

# DTiF

Digital Technologies in focus

Initiative of and funded by the Australian Government Department of Education and Training

**acara** AUSTRALIAN CURRICULUM,  
ASSESSMENT AND  
REPORTING AUTHORITY



## CLASSROOM IDEAS: YEARS 5–8

### INVESTIGATING ENVIRONMENTAL DATA WITH MICRO:BITS

According to the research of Professor Stephen Heppell:  
'A poor physical environment hurts learning.'

Source: [www.learnometer.net](http://www.learnometer.net)

To be more specific, poor light levels, the wrong temperatures, inappropriate sound volumes and rhythms, humidity, air pollution, carbon dioxide (CO<sub>2</sub>) and air pressure can all impair learning. On their own, each of these factors can affect a student's ability to learn. In combination, current research is expected to show that learning outcomes are even worse.

So, what can you do about it? This knowledge provides a great opportunity for students to participate in some authentic transdisciplinary activities focused on Technologies, Science and Mathematics to measure environmental factors and improve the spaces in which they learn. 'If we can optimise that environment students learn more effectively and it also encourages them to become reflective learners, which improves their learning further.'

Source: [www.learnometer.net](http://www.learnometer.net)

Some of these activities could be done using mobile phone apps and devices such as the Learnometer (Figures 1 and 2). Alternatively, your students could measure some of these things for themselves by creating digital solutions (such as a micro:bit with MonkMakes Sensor Board as shown in Figure 3 – a powerful, authentic learning project. See tutorial page 2.

#### **The Learnometer**

Partners of Stephen Heppell have produced the 'Learnometer' – a device which sits happily in your classroom and measures all the physical factors listed earlier. Both versions of the device (Figures 1 and 2) display readouts of the physical environment and can store data in the cloud for later use. For more information about these devices see

<https://gratnellslearnometer.com>



Figure 1: Early model Learnometer



Figure 2: Learnometer

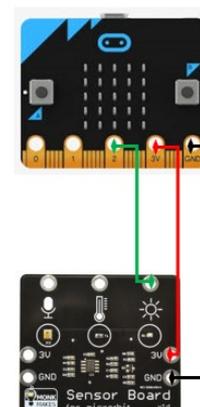


Figure 3: A micro:bit with MonkMakes Sensor Board

## TUTORIAL

This tutorial shows the coding needed for digital solutions to some of the many environmental issues mentioned in the introduction. They can be created using pseudocode/English, visual programming and general-purpose programming.

It is organised into parts as follows:

Part A: Measuring light level – Years 5–6, Years 7–8

Part B: Measuring temperature – Years 5–6, Years 7–8

Part C: Measuring sound level – Years 5–6 or 7–8

Part D: Extension activities (optional)

**Context:** environmental factors affecting learning

**Challenge:** Create a digital device that can measure and display one or more of the following environmental factors in the classroom:

- light levels
- temperature
- sound levels.

**Optional (requires extra sensors)**

- CO<sub>2</sub> levels
- air pressure

**Materials list (Figure 4):**

1 x micro:bit

1 x micro:bit power supply

1 x micro:bit USB connector (not shown)

1 x MonkMakes Sensor Board

10 x alligator leads

1 x buzzer or speaker for micro:bit

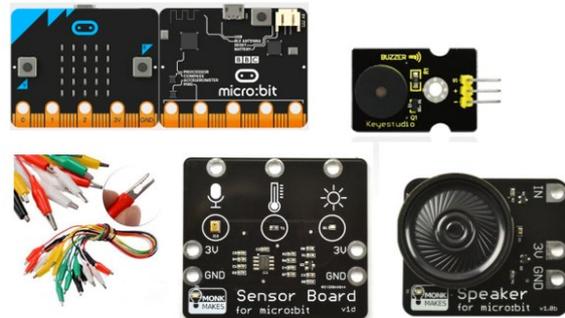


Figure 4: L–R: micro:bit, buzzer, alligator clips, MonkMakes Sensor Board, speaker

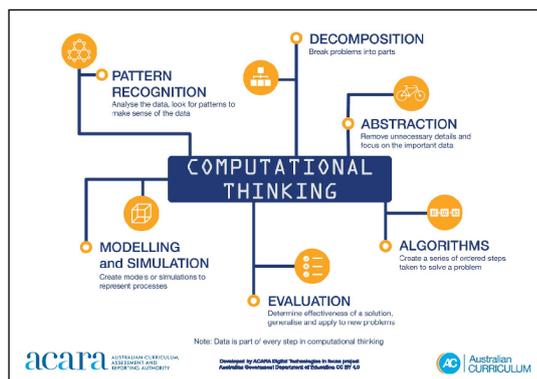
<https://makecode.microbit.org> website or Mu editor for MicroPython Download site:

<https://codewith.mu/en/download>

### Suggested introductory activity

Use the ACARA computational thinking poster as a stimulus to identify the aspects of computational thinking involved in this activity. See

[https://www.australiancurriculum.edu.au/media/5013/computational-thinking\\_poster\\_v3.pdf](https://www.australiancurriculum.edu.au/media/5013/computational-thinking_poster_v3.pdf)



## Part A: Measuring light level

**Intended cohort:** Years 5–6

**Context:** Poor lighting is a significant barrier to learning. Recent research (Barrett et al. 2015) shows that good lighting significantly influences reading, vocabulary and science test scores. Above 500 lux is acceptable but above 1,000 lux is better.

**Challenge:** Create a light meter with your students. To do this we will first need a micro:bit and a clear idea of what we want it to do.

### **Algorithm: Expressed as a simple sequence of steps**

*What is the sequence of steps needed to achieve this digital solution?*

- Have the micro:bit report on the light level.
- Compare that light level to the lux light level indicated by an app on a smart phone.
- Program the micro:bit to convert its reported light level to something similar to the lux readout of the app; that is, we need to code the micro:bit to reflect the true temperature of the room.
- When the user presses button A the calibrated lux level is shown.

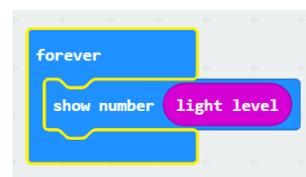


Figure 5

We will use the Microsoft MakeCode website [www.makecode.microbit.org](http://www.makecode.microbit.org) to create this in visual programming. The code to get a light level can be as simple as that shown in Figure 5.

However, we want to do a few things with the light level value, so we will:

1. store it in a variable (see glossary and useful links on page 37)
2. apply a formula to convert the level to something simple that says if it is too dark or at an 'OK' lux level ('lux' may need to be explained before you proceed, depending on students' age/knowledge/ability – see glossary)
3. create a table (such as the one shown in Table 1) with micro:bit reported light levels and lux readings from a phone app placed next to each other. This way these values can be compared easily. For this activity we don't need to go into too much depth or accuracy.

*Table 1: micro:bit and phone app lux values on a scale of healthy light\**

Micro:bit value	Phone app lux value	Healthy light?
12	41	Too dark
17	57	Too dark
35	190	Too dark
44	230	Too dark
53	307	Still too dark
56	345	Still too dark
84	429	Still too dark
88	471	Still too dark
102	592	Light level OK (boundary level)
224	2017	Light level OK
255	5000	Light level OK

\* Above 500 lux is acceptable but above 1,000 lux is better.

The data in Table 1 have been grouped in this way:

- Dark blue – the lux values are around 3–5 times the micro:bit value (too dark)
- Mid blue – the lux values are around 6 times the micro:bit value (still too dark)
- Light blue – the boundary level – at around 100 indicates a lux value of about 500 (light level is OK)
- White – the lux values are around 10 times or more the micro:bit value (light level is OK).

So, using micro:bit values, 88 is becoming a reasonable light level, 102 is a reasonable light level and 224 is desirable.

### **Algorithms: Expressed in pseudocode/English**

How could these steps be expressed in pseudocode?

```
Get the light level
If the light level is below 100
    Then it's too dark
Else
    The light level is fine
```

### **Coding the micro:bit using visual programming**

Students can use the [www.makecode.microbit.org](http://www.makecode.microbit.org) website to create the visual program as shown in Figure 6. This can then be tested on screen with the emulator (virtual micro:bit) and finally downloaded to a physical micro:bit for testing.

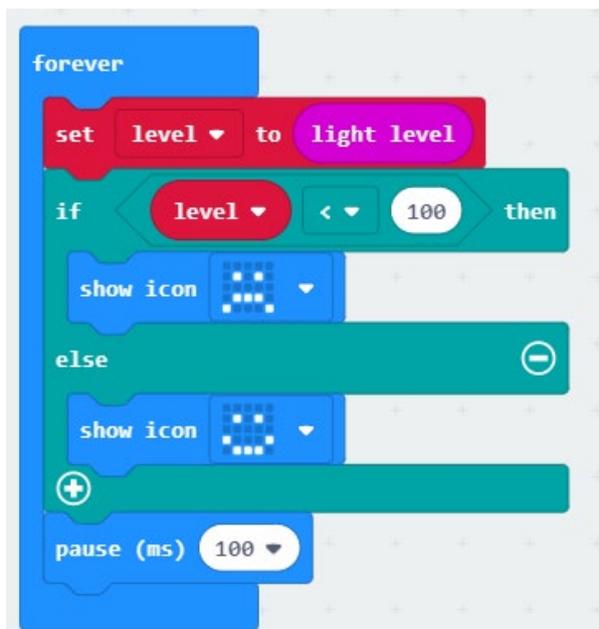


Figure 6

## Part A: Measuring light level

**Intended cohort:** Years 7–8

**Context:** Poor lighting is a significant barrier to learning. Recent research (Barrett et al. 2017) shows that good lighting significantly influences reading, vocabulary and science test scores. Above 500 lux is reasonable but above 1,000 lux is desirable.

**Challenge:** Create a light meter with your students.

This activity provides a great opportunity to discuss why calibration of a device is important; that is, how this will set up the micro:bit to report approximate lux values.

- Dark blue – the lux values are between 3 and 5 times the micro:bit value – we will use 4
- Mid blue – the lux values are around 6 times the micro:bit value – we will use 6
- Light blue – the boundary level – at around 100 that indicates a lux value of about 500
- White – the lux values are 10 times or more the micro:bit value – we will use 10

Initially you could take the students through the following visual programming exercise to explore calibration using the in-built light sensor and Table 1.

### ***Algorithms: Expressed in pseudocode/English***

How could these steps be expressed in pseudocode?

```
Create a variable called level to store the light level
Create a variable called lux to store the lux calculation
Get the light level
If level is below 50
    Then set lux to level x 4
Else if the level is below 100
    Then set lux to level x 6
Else
    Set lux to level x 10
```

### ***Coding the micro:bit using visual programming***

Students can use the [www.makecode.microbit.org](http://www.makecode.microbit.org) website to create the visual program shown in Figure 7. This can then be tested on screen with the emulator (virtual micro:bit) and finally downloaded to a physical micro:bit for testing.

Of course, students should be able to come up with their own tables and boundary values after a bit of experimentation. Some of that experimentation is explained further on in this tutorial.

At this point we will introduce an inexpensive external board called MonkMakes Sensor Board (Figure 9) for students to explore ways to extend the micro:bit's capabilities. This is beneficial for students but optional. Therefore, you may choose to ignore the sensor board code that follows and continue with just the micro:bit.

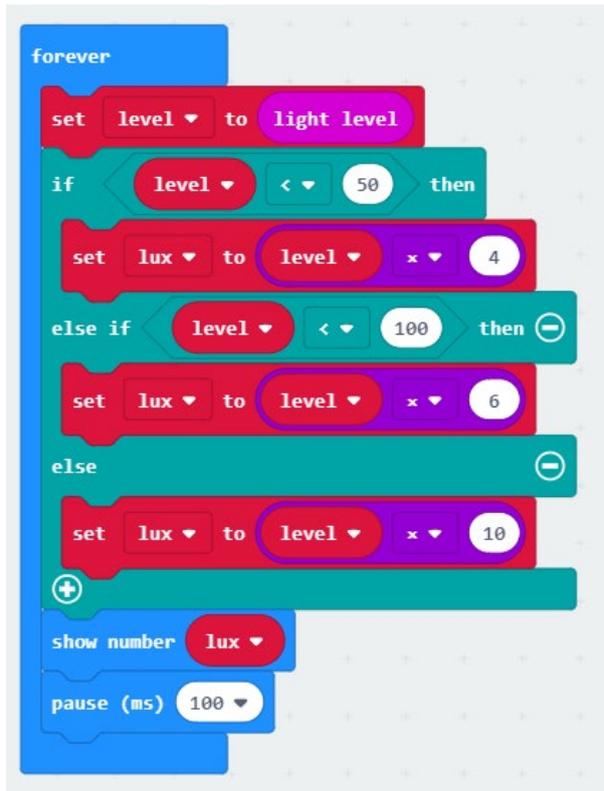


Figure 7

### Coding the micro:bit using general-purpose programming (MicroPython)

Students can code in a general-purpose programming language such as Python. In this tutorial we have used MicroPython, which can be used with Mu editor [www.codewith.mu/en/download](http://www.codewith.mu/en/download), as shown in Figure 8.

```

light_level_3.py x
1  # light_level_3
2
3  from microbit import *
4
5  lux = 0
6  while True:
7      level = display.read_light_level()
8      if level < 50:
9          lux = level * 4
10     elif level < 100:
11         lux = level * 6
12     else:
13         lux = level * 10
14     display.scroll(lux)
15     sleep(100)
  
```

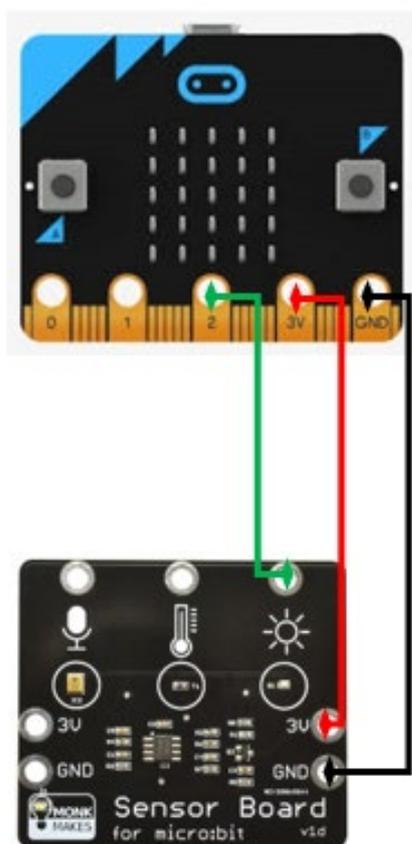
Figure 8

## Adding an external sensor

For this tutorial we are using a MonkMakes Sensor Board. This board has three built-in sensors which we will be using: a thermometer, a light sensor and a sound sensor. We will be combining code using these sensors, and explain why along the way, so that students (and you) get a good understanding of what is going on.

Remember our aim is to create environmental monitors for the classroom to help ensure an optimal learning space. Introducing students to external sensors is a necessary part of this whole exercise.

## Connecting the micro:bit and the sensor board



sensor board (bottom)



Figure 10



The MonkMakes Sensor Board must connect to the micro:bit. The diagram at Figure 9 shows how this is done using alligator clips that connect to the gold teeth at the bottom of the micro:bit. When students do this, remind them to ensure that they have not put the alligator clips over any adjacent teeth (the fine vertical lines between the labelled larger pins).

Once connected, the sensor board is powered via the micro:bit (3V and GND) and for this example the light sensor is connected to pin 2. The next step is to code the micro:bit to display what the sensor is reporting.

The initial values can be read using visual programming (as simple as the example in Figure 10 which has been created in MakeCode). This code doesn't mean much though if the equivalent lux value of the environment isn't known. We created the following example of how students could do this.

First, we used a lux meter (mobile phone app) to find an area in a room that was about 500 lux. We then put the sensor board in the same spot with the same angle – we found the angle of the sensor can affect readings – so we placed it flat on the desk and made sure our shadow was in the way. We did the same for 1,000 lux. The readings provided boundary values to inform our visual programming.

We used the code shown in Figure 10 (visual programming) and collected the data shown in Table 2. You could do this in MicroPython, general-purpose programming language (Figure 11).

Students should collect the same data if they have access to a smart phone and a relevant lux app.

Note: The red 'analog read pin ...' block shown in Figure 10 is located in Pins.

```

light_level_4.py x
1 # light_level_4
2
3 from microbit import *
4
5 while True:
6     display.scroll(pin2.read_analog()) # getlight value reading
7     sleep(1000)
8

```

Figure 11

Table 2: Indoor sensor board and lux values

Sensor board value	Phone app lux value	Healthy light?
< 17	< 500	Too dark
17	503	Light level OK
23	1067	Light level ideal

### Algorithms: Expressed in pseudocode/English

How could these steps be expressed in pseudocode?

Our algorithmic thinking to convert this into an environmental monitor will be very similar to what we have already used:

```

Get the value from pin 2
Create a variable called level to store the value from pin 2
If level is below 17
    Then show sad face and show level
Else if the level is below 23
    Then show happy face and show level
Else
    Show heart and show level

```

### Coding the micro:bit using visual programming

The code using [www.makecode.microbit.org](http://www.makecode.microbit.org) website to carry out our algorithmic thinking is shown in Figure 12.

Students may ask: “Why does it work when 7 is less than 17 but it is also less than 23?” The answer is that the micro:bit goes through the code line by line. As soon as it finds a comparison that is true (7 is less than 17) it doesn’t bother looking at any other parts of the IF ELSE IF ELSE block.

When students can successfully measure suitable light levels in their classroom and a visual alert warns them if the light is too low, we can move on to measuring another factor that can affect learning: temperature.

For Years 5–6 students we will just use the in-built micro:bit temperature sensor. For Years 7–8 students, instead of using the in-built thermometer in the micro:bit, we are going to use the MonkMakes Sensor Board. Why? Well, basically because it may prove more accurate and it introduces students to a whole new world of experimentation and control of their environment that they cannot get just from the micro:bit.

### ***Coding the micro:bit using general-purpose programming (MicroPython)***

Students can code in a general-purpose programming language such as MicroPython, which can be used with Mu editor [www.codewith.mu/en/download](http://www.codewith.mu/en/download), as shown in Figure 13.

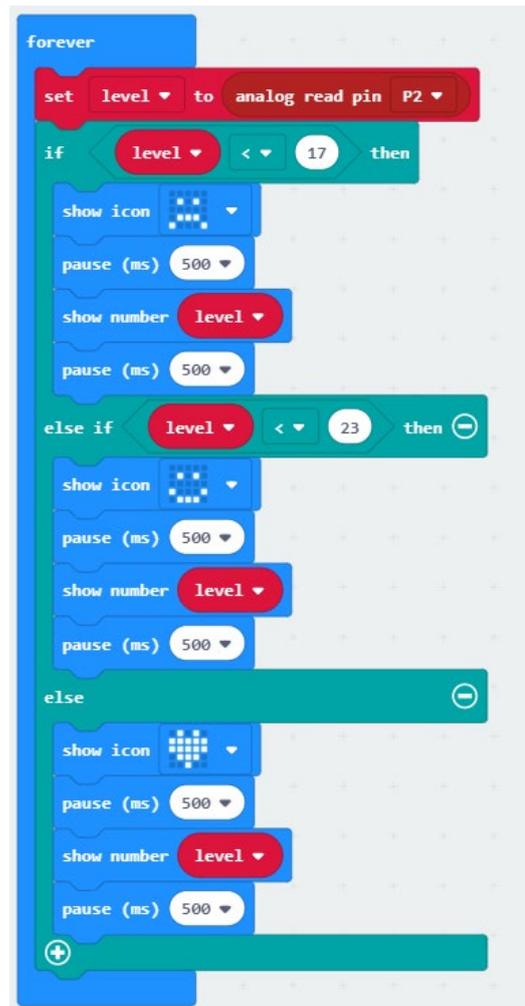


Figure 12

```

light_level_5.py x
1  # light_level_5
2
3  from microbit import *
4
5  while True:
6      level = pin2.read_analog() # getlight value reading
7      if level < 17:
8          display.show(Image.SAD)
9          sleep(500)
10         display.scroll(level)
11         sleep(500)
12     elif level < 23:
13         display.show(Image.HAPPY)
14         sleep(500)
15         display.scroll(level)
16         sleep(500)
17     else:
18         display.show(Image.HEART)
19         sleep(500)
20         display.scroll(level)
21         sleep(500)

```

Figure 13

## Part B: Measuring temperature

**Intended cohort:** Years 5–6

**Context:** Research by Graff Zivin et al. (2018) suggests that warmer classrooms (above 21 °C) have a negative effect on learning and this becomes statistically significant above 26 °C. Another study <https://tinyurl.com/y8pzrdod> confirms that students who experience more hot days during the year perform worse on subsequent standardised exams.

**Challenge:** Create a digital thermometer with your students. For the younger students we will just use the in-built micro:bit temperature sensor.

### *Algorithms: Expressed in English/pseudocode*

How could these steps be expressed in pseudocode?

```
Get the temperature level
    If the temperature level is below 26 degrees Celsius
        Then that's fine
    Else
        The temperature is too high
```

### *Coding the micro:bit using visual programming*

Students can use the [www.makecode.microbit.org](http://www.makecode.microbit.org) website to create the visual program shown in Figure 14.

We could add another readout to make the device more informative. Underneath the tick and the cross you could add two lines saying to pause for a second and then show the temperature.

This addition to the code is shown in Figure 15.

### **Possible extension**

It is easy enough to get the micro:bit to display the temperature, but wouldn't it be great to have it sound an alarm if the temperature gets too warm or too cold?

We will get to that with the approach for Years 7–8 or more advanced students. You could do this with Years 5–6 students if you wish.

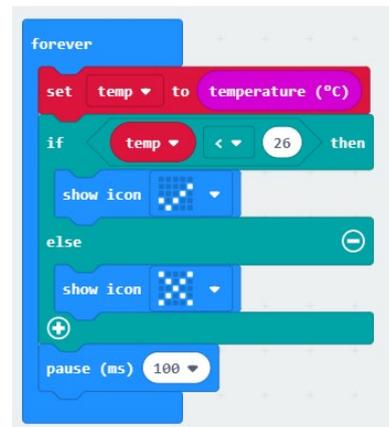


Figure 14

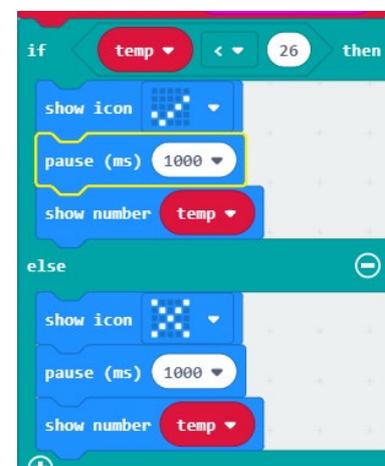


Figure 15

## Part B: Measuring temperature

**Intended cohort:** Years 7–8

**Context:** Research by Graff Zivin et al. (2018) suggests that warmer classrooms (above 21 °C) have a negative effect on learning and this becomes statistically significant above 26 °C. Another study [www.tinyurl.com/y9m3jbxw](http://www.tinyurl.com/y9m3jbxw) confirms that students who experience more hot days during the year perform worse on subsequent standardised exams.

**Challenge:** Create a digital thermometer with your students.

**Preparation:** The micro:bit has a number of gold teeth, as you have already discovered. We can use those teeth to attach to the temperature sensor on the sensor board.

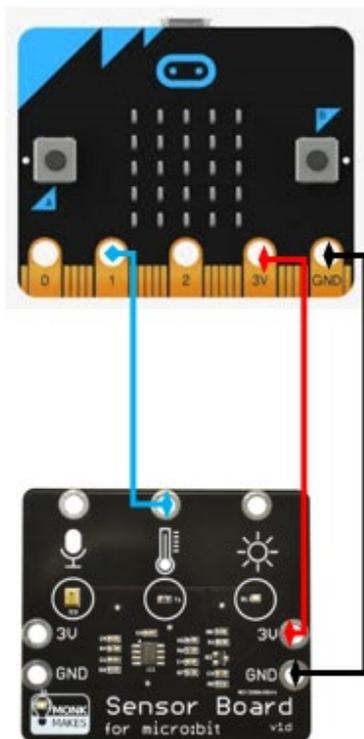


Figure 16

This time, the temperature level is going to be coming from an external sensor via pin 1. At this stage, tell students that there are three pins we can use to collect data from external sensors. On the micro:bit they are labelled 0, 1 and 2 and are at the bottom on the gold 'teeth'. There are many more pins, but students don't need to know about those at this point. To keep it simple at this stage, disconnect the light sensor from the light level activity if it isn't already. We will combine the two at the end of this section. To get the temperature we need to do a bit of mathematics. First let's work out why.



Figure 17

At Figure 16 is a diagram of a sensor board attached to the micro:bit showing it just getting information from the temperature sensor. Notice that there are two other sensors: one for sound and the other (which we have already used) for light.

An activity might be for your students to find out what is more accurate – onboard temperature sensors on the micro:bit or external sensors such as the MonkMakes Sensor Board sensing the same conditions.

NB: If you don't have a MonkMakes sensor (they are about \$15) then you could code this just using the in-built temperature sensor.

The reason we are introducing this sensor board at this time is that it will be needed for sound levels later on, and it avoids a MakeCode issue we discovered when combining the onboard light sensor and the sensor board temperature sensor into one piece of code. You can see the issue in this short video <https://youtu.be/mqbrFcdi0Es> (4 min).

Attach the sensor board to the micro:bit just like in the diagram in Figure 16. Use the visual programming code in Figure 17 or the MicroPython general-purpose programming in Figure 18 to get a reading from pin 1.

```
temperature_3.py x
1 # temperature_3
2
3 from microbit import *
4
5 while True:
6     reading = pin1.read_analog()
7     display.scroll(reading)
8
```

Figure 18

The number that is shown is an analog number between 0 and 1023 which reflects how much electricity is going through the sensor.

When we ran the code the numbers 465 and 466 came up. These are a measurement of electrical (kinetic) energy. We need to work out how to convert those numbers into degrees Celsius.

So students can see the value the sensor board is returning, have them gently put their finger on the temperature sensor: they should be able to see the reading increase.

You may want to explain that the values generated indicate a measurement (in volts) of how much energy is able to pass through the sensor.

### Managing and interpreting the sensor data

To convert the numbers that the sensor is reporting, we need to turn them into something that makes much more sense than a measure of electrical (kinetic) energy.

There are three approaches we could take:

- Use a 'black box' approach that allows students to use a pre-built code block to display a fairly accurate temperature. Students just apply the code block without understanding the mathematics behind it.
- Collect some data and calibrate the micro:bit – the 'calibration approach'.
- Apply the [Steinhart – Hart equation](#) to the data we are reading.

The first two approaches are described below. The Steinhart – Hart equation is beyond the level of this tutorial but would have its place in Years 9–12.

## The black box approach

If you want your students to just get the readings from the sensor, which are then automatically converted to an approximate temperature value:

1. Open [www.makecode.microbit.org](http://www.makecode.microbit.org), click on 'New Project', then complete the Create a Project box. Next click on Extensions (Figure 19) under Advanced (Figure 20).
2. Search for 'Monk' in the search bar (Figure 21) and choose the sensor extension that comes up (Figure 22).

You will then have a new set of blocks called Sensor (Figure 23). The video at the following link explains the process <https://youtu.be/sx6OIfdq3sE> (1 min).

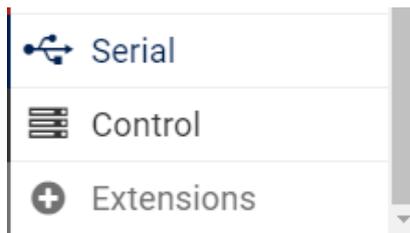


Figure 19

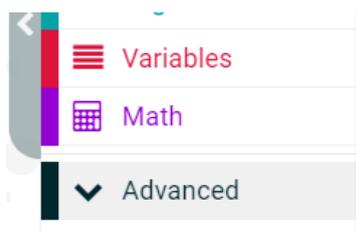


Figure 20

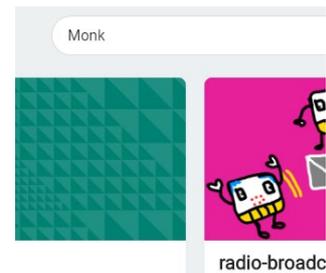


Figure 21

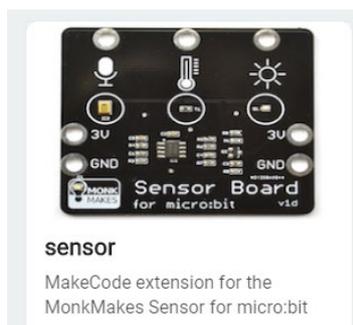


Figure 22

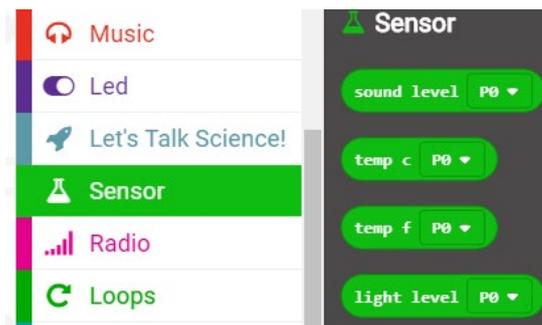


Figure 23

With the temperature block (Figure 23) we can quickly get the temperature from the sensor board. Again, our code will be based on the following algorithmic thinking/pseudocode.

### Algorithms: Expressed in pseudocode/English

How could these steps be expressed in pseudocode?

Get the temperature level

If the temperature level is below 26 degrees Celsius

Then that's fine, display a tick, then the temperature

Else

The temperature is too high – display a cross, then the temperature

## Coding the micro:bit using visual programming

The code that we will use is not too dissimilar to the code used for Years 5–6, except that we are asking an external device to provide the values (Figure 24).

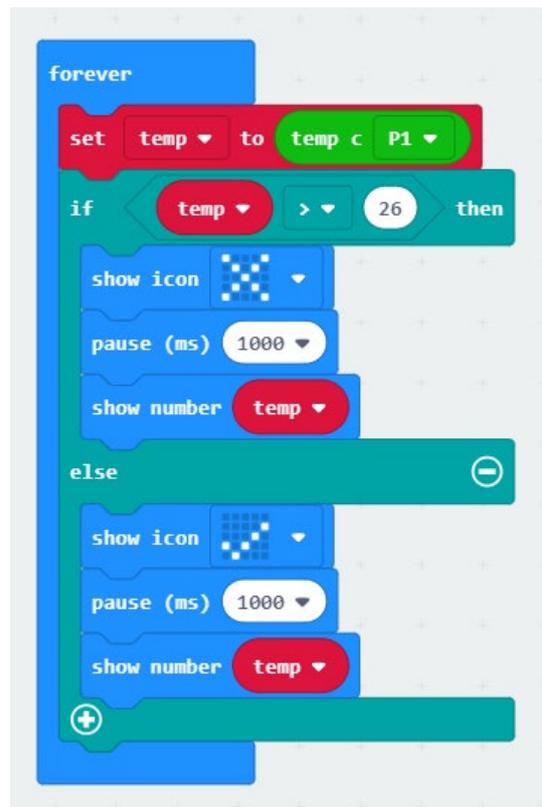


Figure 24

### The calibration approach

Why bother with this approach if the black box approach works? Well, this approach covers a lot of the collection, analysis and representation of data key concepts in Digital Technologies, as well as the specification, algorithms and implementation key concepts. The black box approach does as well but not as explicitly nor to the same degree; however, it does illustrate the concept of abstraction. The calibration approach basically explains the formula in the black box approach (page 13), taking the 'black box' away so students understand what is actually happening.

To complete this approach, the students need access to a digital thermometer. We picked one up at a local supermarket for about \$10.

The way for students to do this is to get two known different temperatures and the sensor values for those same temperatures. We recorded an indoor temperature as well as an outdoor temperature that was in the shade. Students could record a temperature first thing in the morning and another around lunchtime (provided they were quite different readings). Students then apply a fairly simple formula to convert the sensor value to a temperature within the micro:bit code.

We will use a table to collect our data and apply our formula:

**Celsius = (reading x C) – D** (see Table 3)

This formula comes from MonkMakes and it provides a simple way to apply a formula that does just about the same thing as the Steinhart – Hart equation, without all the difficult mathematics.

The filled in data are shown in Table 3.

Table 3: Sample temperature data

Data collected	Name we give it	Value reported
Thermometer reading (current room temperature)	t1	18
Sensor reading (current room temperature)	r1	430
Thermometer reading (different temperature)	t2	10
Sensor reading (different temperature)	r2	330
<b>Formula application</b>		
$A = t1 - t2$	A	$18 - 10 = 8$
$B = r1 - r2$	B	$430 - 330 = 100$
$C = A / B$	C	0.08
$D = t2 - (C * r2)$	D	$10 - (0.08 \times 330)$ $10 - 26$ $= -16$

So in our example,  $A = 8$ ,  $B = 100$ ,  $C = 0.08$  and  $D = -16$ .

A is the difference between the two thermometer readings.

B is the difference between the two sensor readings.

C represents the voltage change per degree.

D is the second thermometer reading minus the change per degree multiplied by the second sensor reading.

To get the temperature we use C and D as well as the voltage reading. Our final formula based on our collected data is: **Celsius = reading x 0.08 – 16.**

So if the voltage is 427 we get a temperature of 18 degrees Celsius.

The final formula we used is for the readings taken for the conditions we were in. *Your readings/numbers will of course be different from these.* It is the process that the students go through that is important. If you are using a different temperature sensor than the one used in this tutorial, you may need to apply a different formula.

To make it simpler to refer to the temperature that is calculated by the formula we will create a variable called 'temp' to store the calculated temperature and create the code (Figure 25).

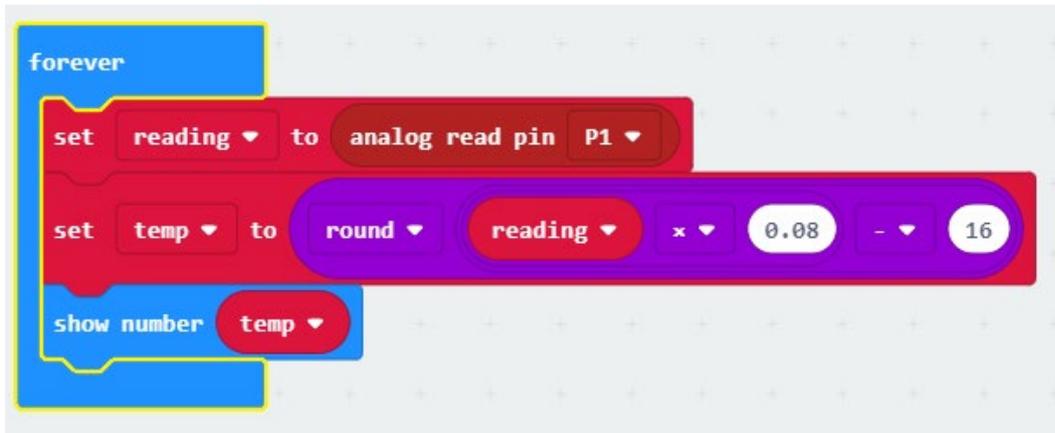


Figure 25

We used the 'round' function, otherwise there would be many decimal places shown. Students could experiment with and without the round block to see which is more user friendly.

If students put the various blocks in the wrong spot, then the readings will be way off. The easiest way to check is to make sure that the section of code between 'round' and 'reading' has a double border line. It is hard to see, so here it is magnified and indicated with an arrow in Figure 26.

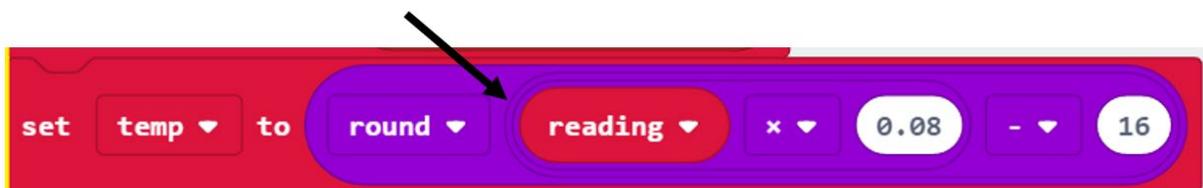


Figure 26

This video shows how to build the set temp block in the correct order: <https://youtu.be/nq1uu2490bk> (2 min).

The sequence for the code is shown step by step in Figure 27.

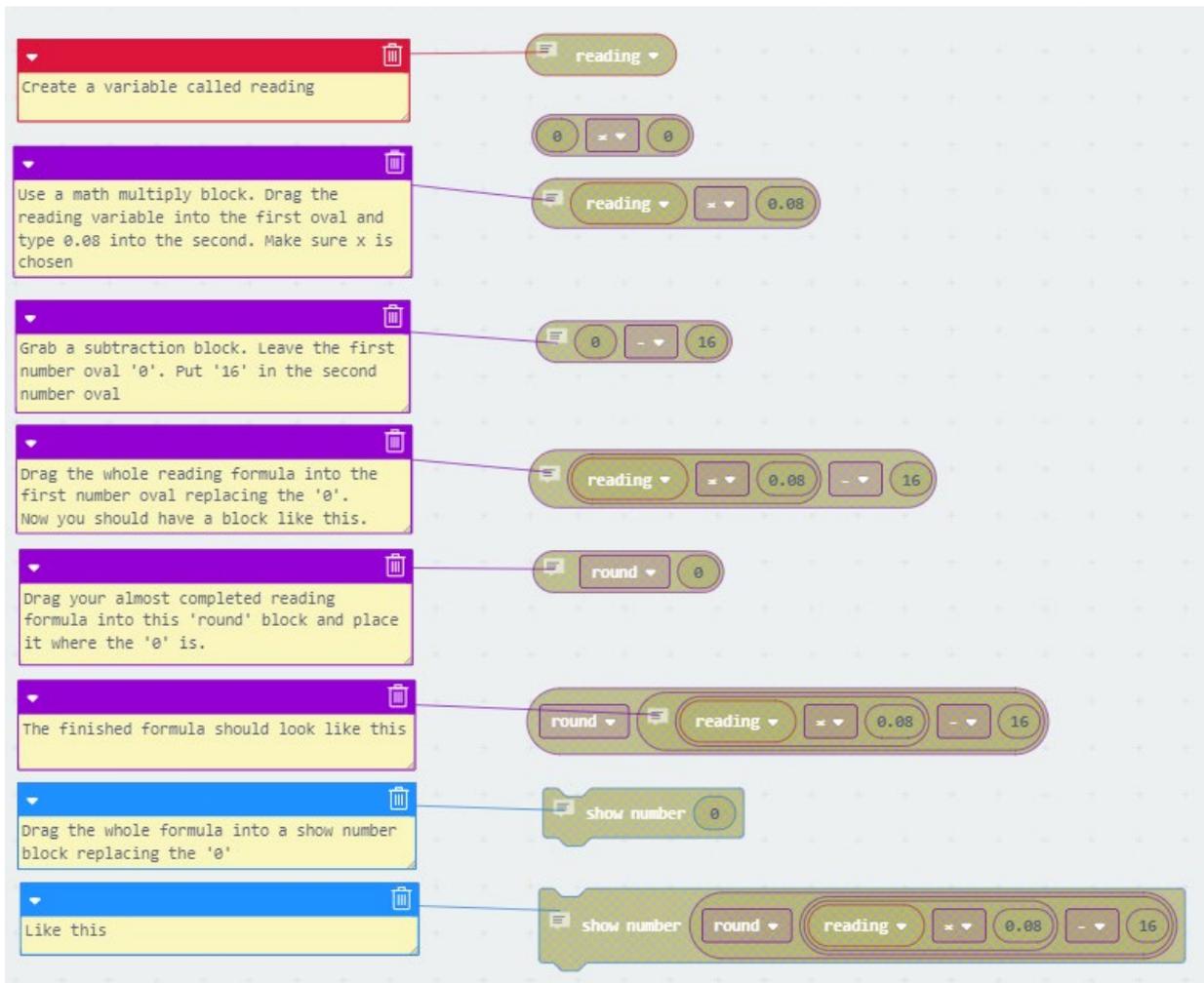


Figure 27: Steps for building the set temp block in the correct order

The code using the calibration method contains only two different lines to the black box method (Figure 28). Encourage students to try working it out for themselves first. Comparing the two methods might also lead to a discussion related to the Digital Technologies key concept, abstraction; that is, hiding unnecessary details.

```

forever
  set reading to analog read pin P1
  set temp to round reading * 0.08 - 16
  if temp > 26 then
    show icon [LED icon]
    pause (ms) 1000
    show number temp
  else
    show icon [LED icon]
    pause (ms) 1000
    show number temp
  
```

Figure 28

## Coding the micro:bit using general-purpose programming (MicroPython)

Students can code in a general-purpose programming language such as MicroPython, which can be used with Mu editor [www.codewith.mu/en/download](http://www.codewith.mu/en/download), as shown in Figure 29.

```
temperature_4.py X
1 # temperature_4
2
3 from microbit import *
4
5 # there isn't a micropython equivalent of the MONK MAKES temp c extension block
6 # so we will build one that acts just the same
7 def temp_c():
8     reading = pin1.read_analog() # get reading from the sensor
9     celsius = reading * 0.08 - 16 # apply our formula
10    return celsius # return the result
11
12
13 # now we can recreate the block code in python
14 while True:
15     temp = temp_c() # temp stores the return value after running temp_c
16     if temp > 26: # perform the conditionals
17         display.show(Image.NO)
18         sleep(1000)
19         display.scroll(temp)
20     else:
21         display.show(Image.YES)
22         sleep(1000)
23         display.scroll(temp)
```

Figure 29

## Adding an alarm

Now that we can get the temperature, we need to add an alarm that will beep if the temperature gets too warm. This could be added to both the Years 5–6 and Years 7–8 student projects.

To begin with we will use a simple Keyestudio buzzer that we sourced from the internet for a few dollars (Figure 30).

Notice that it has three pins:

- connects to GND
- + connects to 3V
- S connects to a spare pin on the micro:bit.



Figure 30: Image source:  
<https://wiki.keyestudio.com/File:361-10.png>

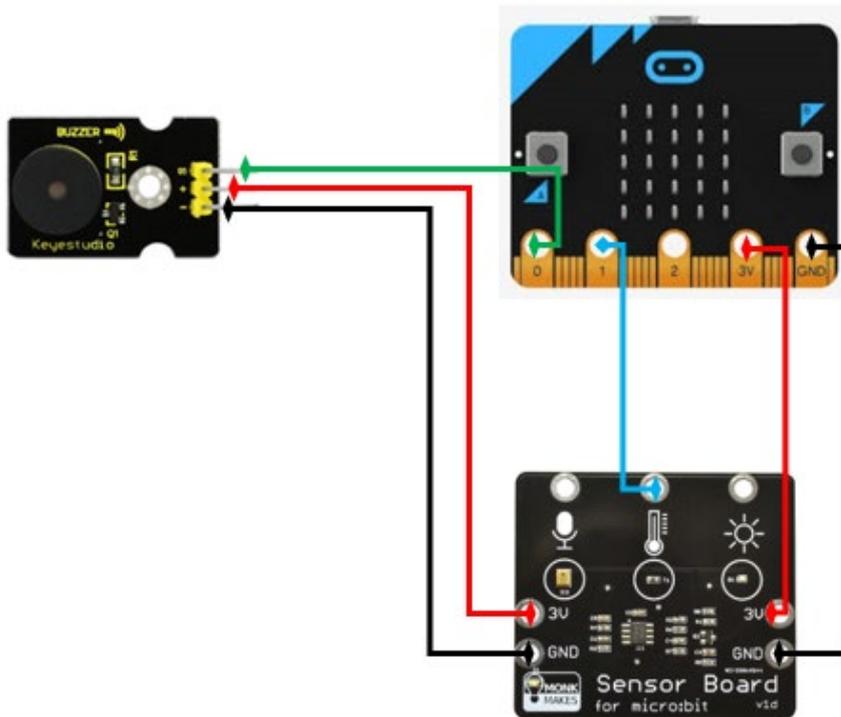


Figure 31

Next, we will connect this to the micro:bit and sensor board we have already set up (Figure 16). To do this, follow the diagram shown in Figure 31.

Note that the green signal lead (S) is connected to pin 0 on the micro:bit. We will need to remember this in our coding. Connecting it to pin 0 wasn't an accident, as we will discover shortly.

Now to get the alarm to work we just need to tell the micro:bit to send a signal out of pin 0 if our boundary value (26 degrees from the research above) is reached.

### **Algorithms: Expressed in pseudocode/English**

How could these steps be expressed in pseudocode?

Get the temperature level

If the temperature level is below 26 degrees Celsius

Then that's fine, display a tick, then the temperature

Else

The temperature is too high – display a cross, then the temperature

Also sound an alarm for a second

## Coding the micro:bit using visual programming

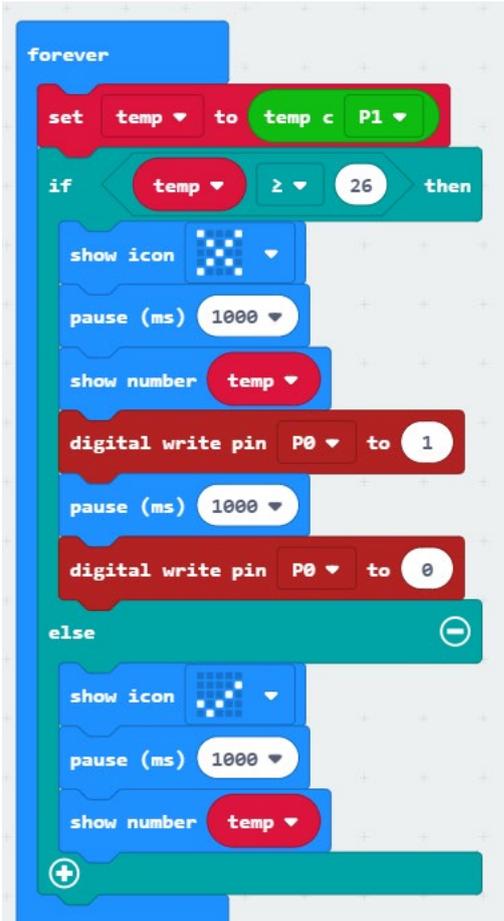


Figure 32

After setting up the code shown in Figure 32, we now have a micro:bit that can sense temperature and alert us if it gets too high for conducive learning to take place in the classroom.

If you are happy for the students to experiment with some other tones or alarms then they could try using some of the music blocks in MakeCode (like the 'dadadum' melody, for example), instead of the digital write pin 0 blocks.

An interesting thing for them to do then would be to connect the buzzer to pin 2 instead, recode and see if the melodies still play. This could lead to a discussion about pin 0 having some different properties compared with pin 2.

Following in Figure 35 is the extra code we need added to the 'black box' code. If your students did the calibration method, then they should be able to work out where to place the necessary extra blocks.

## Coding the micro:bit using general-purpose programming (MicroPython)

The MicroPython code is shown in Figure 33 and as an alternative with music that plays when the temperature gets too high in Figure 34.

```
temperature_6.py x
1 # temperature_6
2
3 from microbit import *
4
5
6 while True:
7     reading = pin1.read_analog() # get the sensor value
8     temp = round((reading * 0.08)-16) # temp value is rounded form of formula
9     if temp > 26: # perform the conditionals
10         display.show(Image.NO)
11         sleep(1000)
12         display.scroll(temp)
13         pin0.write_digital(1)
14         sleep(1000)
15         pin0.write_digital(0)
16     else:
17         display.show(Image.YES)
18         sleep(1000)
19         display.scroll(temp)
```

Figure 33

```

temperature_7.py x
1 # temperature_7
2
3 from microbit import *
4 import music # import the music library
5
6 while True:
7     reading = pin1.read_analog() # get the sensor value
8     temp = round((reading * 0.08)-16) # temp value is rounded form of formula
9     if temp > 26: # perform the conditionals
10         display.show(Image.NO)
11         sleep(1000)
12         display.scroll(temp)
13         music.play(music.PUNCHLINE) # play this tune from music library
14         sleep(1000)
15
16     else:
17         display.show(Image.YES)
18         sleep(1000)
19         display.scroll(temp)

```

Figure 34

## Combining codes

If we combine our temperature code with the code that visually alerts us if the light is too low then we have covered two conditions that research proves are important for optimal learning.

We will use the calibration code (including the alarm code) and combine it with the light level code. Initially our code looks like that shown in Figure 35.

This code is starting to get a bit complex and hard to read because it is getting too long and doing two different things which take time to read through and can cause confusion.

It is also hard to understand because the second half refers to level, but what sort of level is it? Is it light? Sound? Water? Air pressure? We will need a better variable name. Computer programmers refer to this as intrinsic documentation and it is really important for readability and maintainability. Maintainability refers to how easy it is for other programmers to make necessary changes to your code.

Computer programmers often break code up into logical smaller blocks called procedures or functions. Basically, these are self-contained pieces of code that can be called upon at any time by other parts of a program.

In the next section we will break our code up into more logical chunks.

```

forever
  set reading to analog read pin P1
  set temp to round reading x 0.08 - 16
  if temp > 26 then
    show icon [x]
    pause (ms) 1000
    show number temp
    digital write pin P8 to 1
    pause (ms) 1000
    digital write pin P8 to 0
  else
    show icon [x]
    pause (ms) 1000
    show number temp
  set level to analog read pin P2
  if level < 17 then
    show icon [x]
    pause (ms) 500
    show number level
    pause (ms) 500
  else if level < 23 then
    show icon [x]
    pause (ms) 500
    show number level
    pause (ms) 500
  else
    show icon [x]
    pause (ms) 500
    show number level
    pause (ms) 500

```

Figure 35

## Coding the micro:bit using general-purpose programming (MicroPython)

The MicroPython code is shown combining light and temperature using the micro:bit in Figure 36 and with the MonkMakes Sensor Board in Figure 37.

```
light_and_temp_1.py X
1 # light_and_temp_1
2
3 from microbit import * # load all the microbit code library into MicroPython
4
5
6 def getLight():
7     light = display.read_light_level() # light = micro:bit light sensor value
8     if light < 100:                    # if light level is too low
9         display.show(Image.SAD)       # show sad face
10    else:                               # or else
11        display.show(Image.HAPPY)     # show happy face
12        sleep(500)                    # pause for half of a second
13
14 def getTemperature():
15     reading = pin1.read_analog()      # get temp sensor value & store in reading
16     temp = int((reading*0.08)-16)    # convert it to Celsius
17     display.scroll(temp)              # show the temperature
18
19     if temp > 26:                     # if temp too hot
20         pin0.write_digital(1)        # sound an alarm
21         sleep(1000)                  # for a second
22         pin0.write_digital(0)        # then turn off
23     else:                              # or else...do nothing really
24         sleep(100)                   # just pause to avoid data overload
25
26 def main():                           # create a main function that calls the other two
27     while True:
28         getLight()
29         sleep(1000)
30         getTemperature()
31         sleep(1000)
32
33 main() # call the main function
```

Figure 36

```
light_and_temp_2.py X
1 # light_and_temp_2
2
3 from microbit import * # load all the microbit code library into MicroPython
4
5
6 def getLight():
7     light = pin2.read_analog() # light = MONK MAKES light sensor value
8     if light < 100:            # if light level is too low
9         display.show(Image.SAD) # show sad face
10    else:                       # or else
11        display.show(Image.HAPPY) # show happy face
12        sleep(500)              # pause for half of a second
13
14
15 def getTemperature():
16     reading = pin1.read_analog() # reading = MONK MAKES temp sensor value
17     temp = int((reading*0.08)-16) # convert it to Celsius
18     display.scroll(temp)         # show the temperature
19
20     if temp > 26:               # if temp too hot
21         pin0.write_digital(1)  # sound an alarm
22         sleep(1000)            # for a second
23         pin0.write_digital(0)  # then turn off
24     else:                       # or else...do nothing really
25         sleep(100)             # just pause to avoid data overload
26
27
28 def main():                     # create a main function that calls the other two
29     while True:
30         getLight()
31         sleep(1000)
32         getTemperature()
33         sleep(1000)
34
35 main() # call the main function
```

Figure 37

## Creating some functions

In Microsoft MakeCode if you go into the advanced blocks at the bottom of the list of blocks (below Math) you will see a block to **Make a Function** (Figure 38).

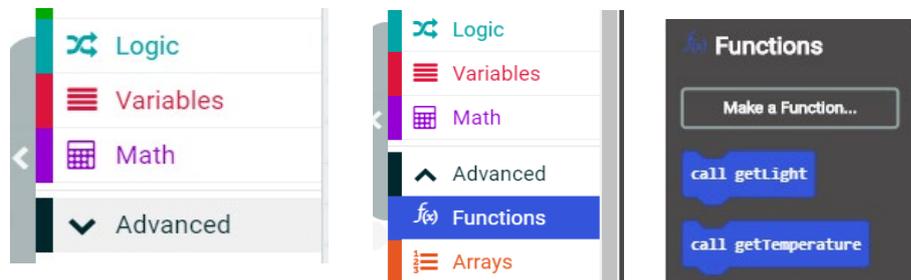


Figure 38

The video at this link <https://youtu.be/LGaTYz022mk> demonstrates the whole process (3 min). Figure 39 shows the same code as previously seen in Figure 35, but broken up into three separate pieces of code:

- a function to getTemperature
- a function to getLight
- a forever loop which cycles between the two with a one-second break in between.

Compare the two different ways to write the same code. Which one do you find easier to understand and read? This could lead to a discussion about abstraction and hidden code, code libraries etc.

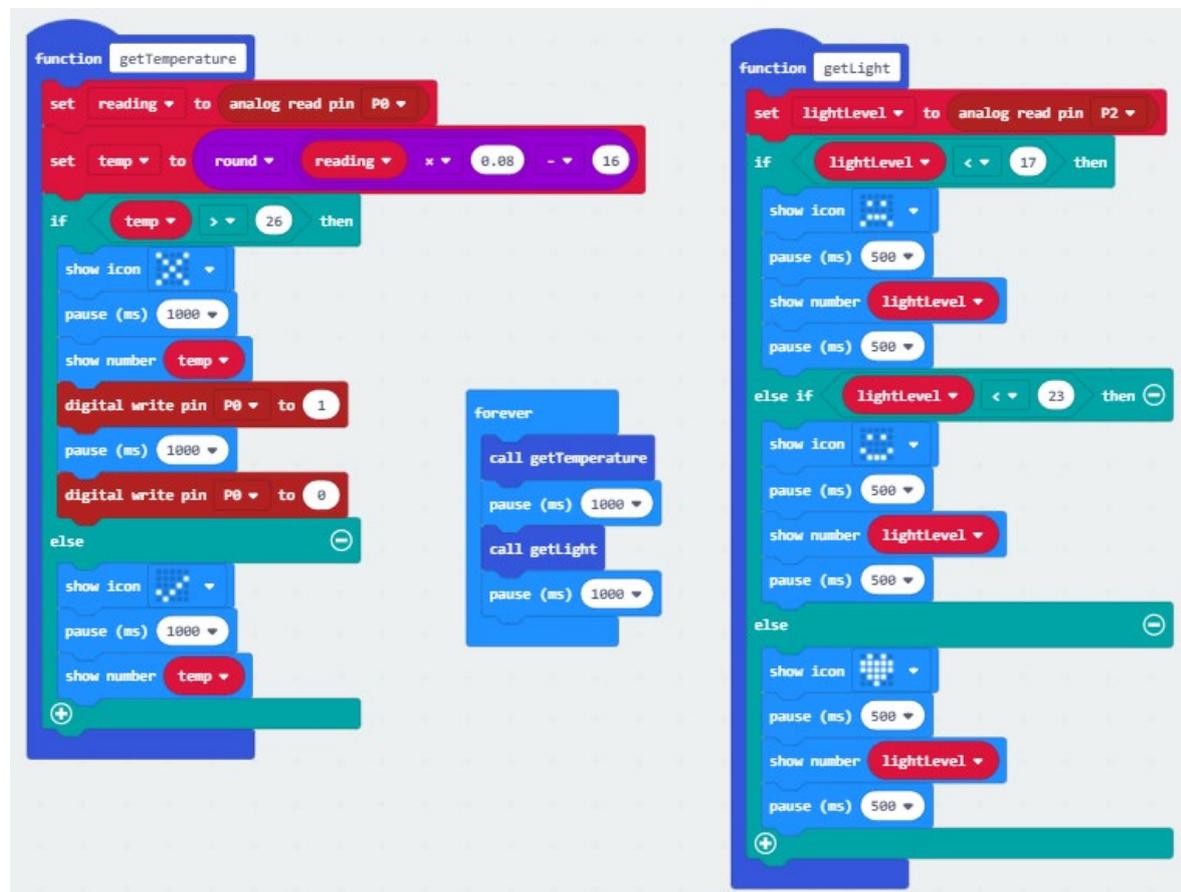


Figure 39

Of course, for the temperature and light to both provide data to the micro:bit and for the alarm to function we need to connect it all up together. Figure 40 is a diagram showing the correct connections. Your students could probably work it out for themselves at this stage.

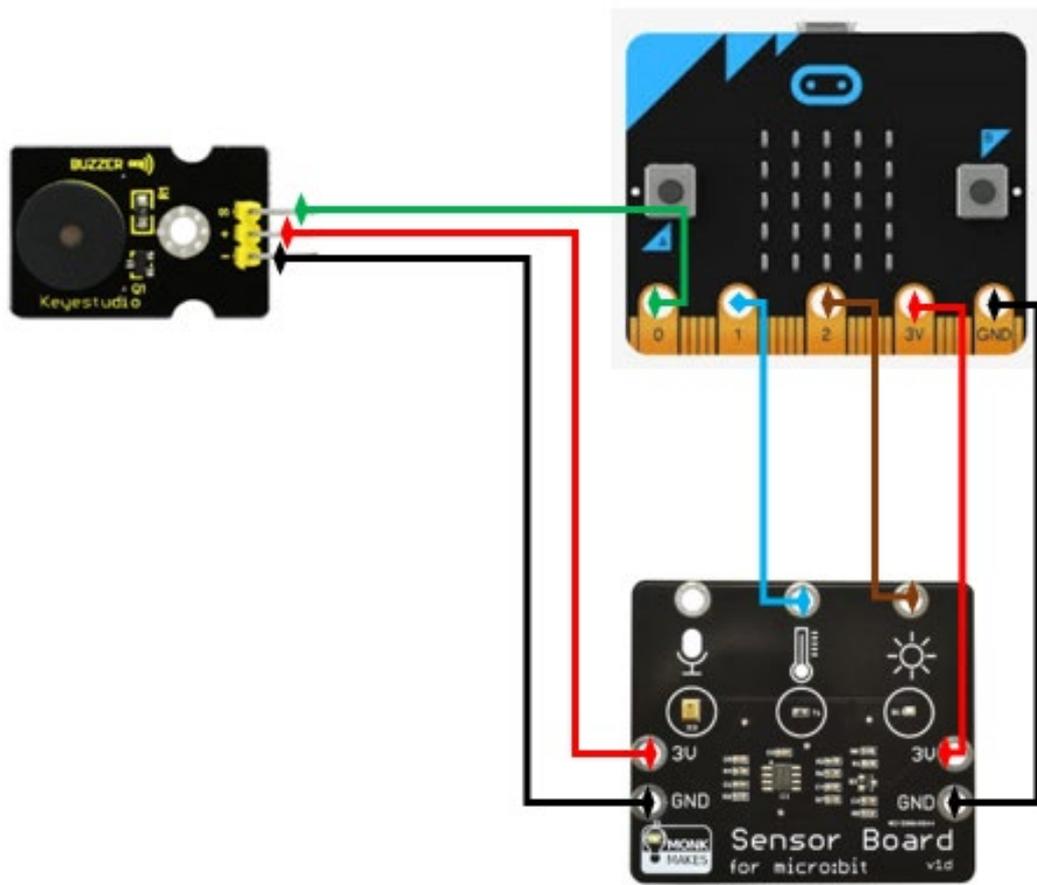


Figure 40

You could get your students to experiment to get the alarm to chirp longer than it does in the code presented. Then again, to save sanity, maybe don't tell them!

## Part C: Measuring sound level

**Intended cohort:** Years 5–6 or 7–8

**Context:** Classroom sound signatures can affect how well students achieve (Picard and Bradley 2001 [www.ncbi.nlm.nih.gov/pubmed/11688542](http://www.ncbi.nlm.nih.gov/pubmed/11688542)). Studies by James et al. (2012) and Anderson (2001) show that ‘children from classrooms with poor acoustics have lower literacy and numeracy skills, are less productive in the workforce, and tend to be in lower paid jobs than those from classrooms with good acoustics’ (in Mealings 2016 [www.tinyurl.com/y8dqypl2](http://www.tinyurl.com/y8dqypl2)). Anything above approximately 72 decibels starts to get disruptive. Above 50 makes concentrating difficult.

**Challenge:** Create a sound monitor with your students.

**Preparation:** On the MonkMakes Sensor Board there is a third sensor (in addition to temperature and light) which can detect sound. Unlike temperature and light, the micro:bit does not have an in-built sound sensor. If students want to monitor their classroom for suitable noise levels, some sort of microphone sensor which works with a micro:bit is needed.

For this tutorial we will use the MonkMakes Sensor Board. What you may have noticed is that the 0, 1, 2, 3V and GND pins have all been used on the micro:bit to get the temperature, light and alarm operating (Figure 40).

There are breakout boards or edge connectors [www.tinyurl.com/y9h4vjyf](http://www.tinyurl.com/y9h4vjyf) available for the micro:bit which allow you to utilise all 21 pins, including all the little pins in between the larger labelled pins just mentioned. We won’t use one of those here. Instead, we will just code another micro:bit and connect it up to the sensor board as a stand-alone sensor.

If you have read the research findings at the start of this part, you will be aware that sounds above 72 decibels are regarded as disruptive, although noise over 50 might be annoying if you are trying to concentrate. We are going to use the sensor board to provide sound levels for the micro:bit and tell it when the noise level is too high and to sound an alarm to alert the students that the environment is no longer conducive to optimal learning. Here is how we will do that.

### Connecting the Sound (microphone) sensor

Connect pin 0 to the S pin on the buzzer. Next, connect the sound sensor to pin 1 or pin 2. In Figure 41 we have used pin 1. Finally, we need to connect the power connectors as we had them before.

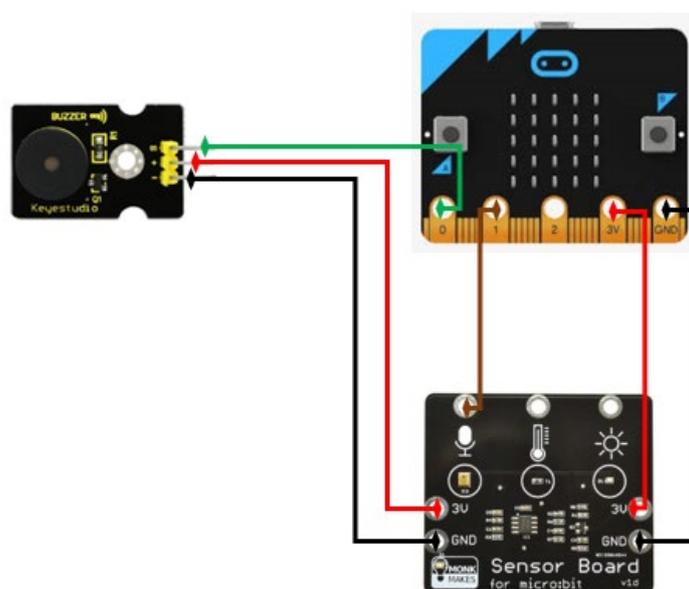


Figure 41

## Experimenting with the sound sensor

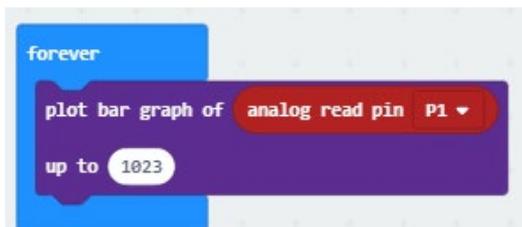


Figure 42

The sound sensor is connected to pin 1. It supplies an analog signal between 0 and 3 V. The signal swings above and below a midpoint of about 1.5 V. Let's code the micro:bit to show this.

We are going to graph the results that the micro:bit receives from the sensor board.

The visual programming code to start on our noise level journey is shown in Figure 42 and in MicroPython in Figure 43.

```
sound_1.py X
1 # sound_1
2
3 from microbit import *
4
5 # there isnt a micropython equivalent of MakeCode plot bar graph block
6 # so we will build one that acts similarly
7 def graph(a): # make a function called graph that mimics plot bar graph block
8     display.clear()
9     for y in range(0, 5): # plot the LED rows and columns
10        if a > y:
11            for x in range(0, 5):
12                display.set_pixel(x, 4-y, 9) # light pixels (x, y, brightness)
13
14 while True:
15     sound_level = (pin1.read_analog()-511)/100 # get the sensor value
16     graph(sound_level) # call the function and provide the parameter(a)
17 # Write your code here :-)
```

Figure 43

You will notice that even when a space is really quiet, about half of the LEDs light up. (On our simulator it looks like the image shown at Figure 44.)

That is because a very quiet environment registers at about 500 (which is about 1.5 volts). It appears as though there is much greater noise when it is actually really quiet.

*NB: Noises that are greater than this register at above or below the 1.5 volts. This is due to the way the sensor works. To find out more about sinusoidal waves created by sound pressure (not necessary to know this to do the activity) see [www.tinyurl.com/y8e9poa6](http://www.tinyurl.com/y8e9poa6).*

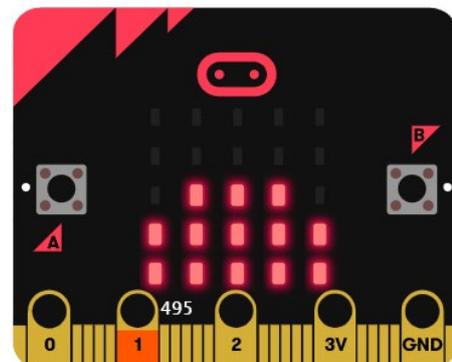


Figure 44



Figure 45

To resolve this issue we can do some mathematics. Since silence starts at around 500, if we take about 500 away then silence will then be represented by about 0 or 1. We are actually going to subtract 511.



Figure 46

You may remember that the analog signals being reported by the sensor board fall between 0 and 1023. If we take 511 from the highest value then that is about half. Have the students try the code shown in Figure 45 to see if they can get the top two lines of the LEDs to light up at all. Of course, they won't be able to because if we are

taking 511 from any reading, then the top reading can only be 512 ( $511 + 512 = 1023$ ). We have told the graph to plot to 1023, which it won't get to.

To get the graph to be more accurate, we need to tell it to make 512 the highest value to expect. The finished graph code is shown in Figure 46. Using this code, students should be able to get all the rows of LEDs to light up, even momentarily. This video shows this whole process <https://youtu.be/EidbZE5NK8Y> (4 min).

The graph shown in Figure 47 shows what happens when sounds are detected. Notice that the readings oscillate above and below the 1.5 V level.

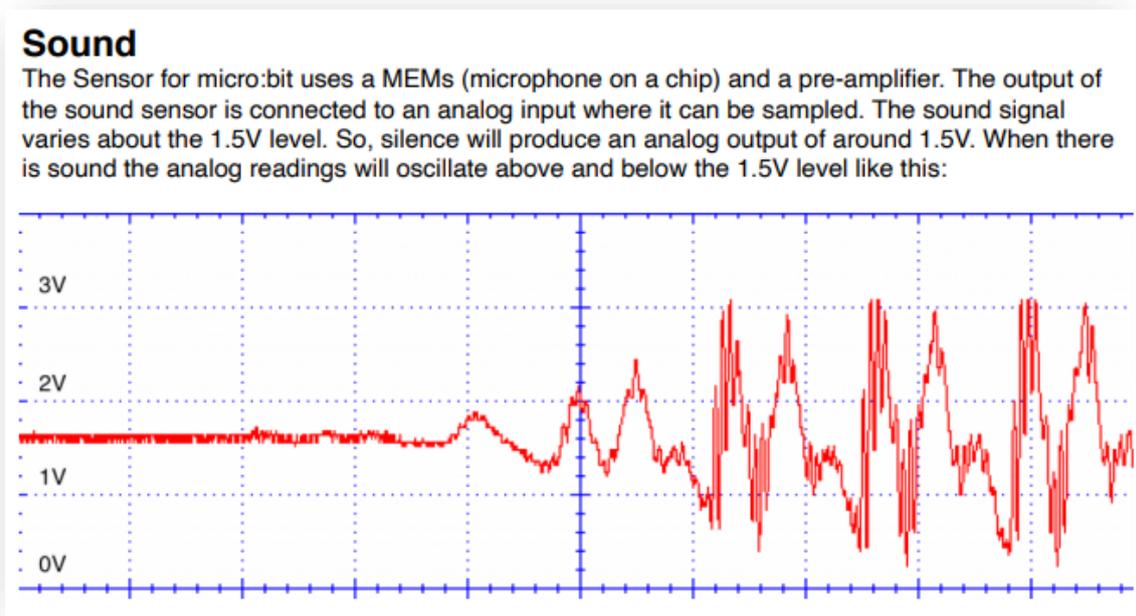


Figure 47. Source: [www.monkmakes.com/downloads/instructions\\_mb\\_sensor.pdf](http://www.monkmakes.com/downloads/instructions_mb_sensor.pdf)

## Capturing the sound level data

We are going to stray for a few minutes to explore a really powerful feature of MakeCode for micro:bits. When we are watching the LEDs light up, they are being controlled by incoming data from the sensor board sound level sensor. There is a lot of data coming in. Wouldn't it be great if we could capture that data for later use or analysis? Well, we can.

To do this, the micro:bit needs to be paired to the computer that is being used to program it. If you haven't discovered this yet, it is the easiest way to download code to the micro:bit.

There are two things you will need for a PC, both of which are worth the effort:

- the latest Chrome browser (at least version 65) or the latest Microsoft Edge browser (at least version 83 or the beta version)
- firmware version 0249 or above installed on your micro:bits.



Figure 48

When you open the Microsoft micro:bit MakeCode editor and open or start a new program you will see a gear symbol near the top right (Figure 48).

When you click the gear icon an option should come up saying 'Pair device' (Figure 49). If you cannot see that option, you will need to upgrade your browser.

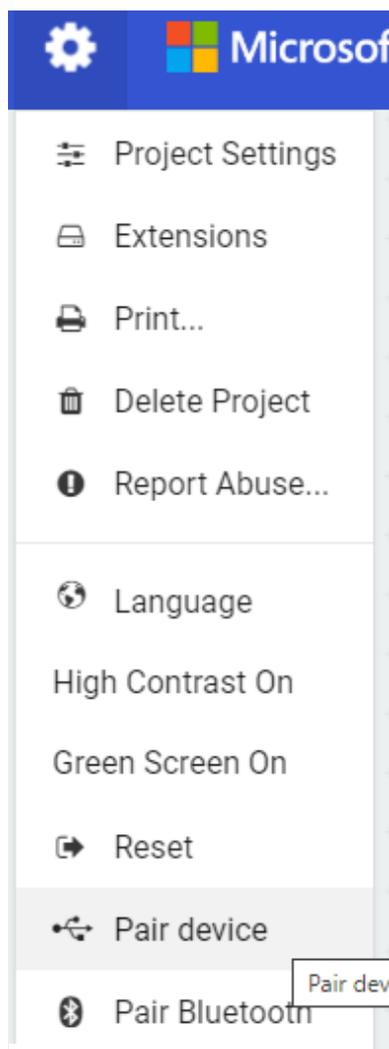


Figure 49

If you have a class set of micro:bits it is best to upgrade them all at once. Just download the firmware file and follow the instructions for each micro:bit. After the initial download it takes only 30 seconds per micro:bit to get them all ready.

Of course, getting at least version 65 of Chrome may need negotiation with whomever looks after your computer network.

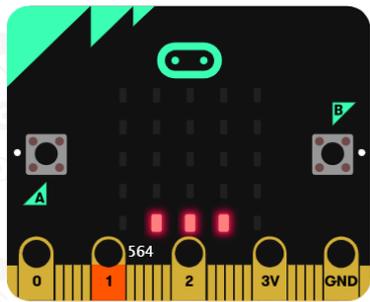
This video explains the process of upgrading your micro:bits if they cannot seem to pair, and then going through the process of pairing <https://youtu.be/r1VgzQV8to0> (5 min).

This is worth doing with your personal or home computer just to see the possibilities, even if your school computers' software may need to be upgraded for it to work for the students.

Make sure the micro:bit and your computer are connected by a USB cable. When you click on pair device (big green button in pop-up), a window comes up and it should have the name of the micro:bit in it. Click on the name and then click connect.

Now when you want to download some code, just click the purple download button and it automatically goes straight to the micro:bit. Better still, when using the plot bar graph block of code, a new button will appear called **Show console Device** (Figure 50).

## Using Show console Device



Make sure that your micro:bit is paired with your computer. Download the latest code we have written (Figure 46). You will see more LEDs lighting up on the micro:bit as the sound level increases.

If you click on the **Show console Device** button under the emulator a graph will appear which is showing in real time what the sensor board is picking up.

Figure 51 shows an example using the initial code.

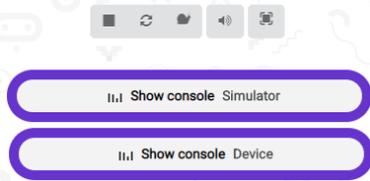


Figure 50

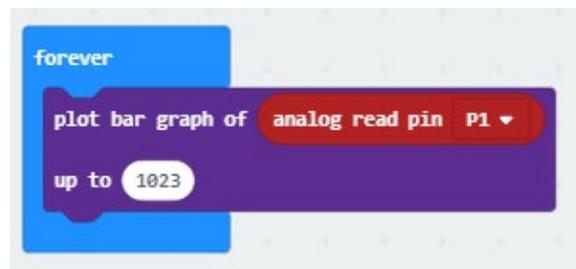


Figure 51

Figure 52 shows what it picked up with the latest code.



Figure 52

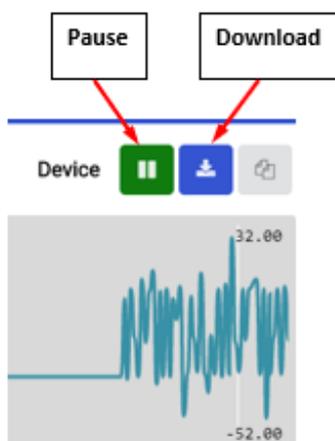


Figure 53

Notice the almost flatline quiet signal on the top graph (Figure 51) is at around 500 (look where 827 and 17 are) and it is around 0 on the bottom graph (Figure 52).

If we pause the real-time graphing we can then download (Figure 53) the data as a (CSV) file. We can then analyse that data in a spreadsheet.

Take a look at this video to see an example of what you can do with these data <https://youtu.be/avL9GUGZpzc> (7 min).

Students could also add a plot bar graph function to the temperature and the light codes as well, and there would be a wealth of data to analyse.

We suggest sampling only once every minute to reduce the amount of data that students have to deal with. Of course, it would depend on what you were collecting the data for.

There is a lot that can be done with this graphing and data capture ability. One idea is micro:bits sending data via their radio functions to a central micro:bit connected to a computer which collects all that data from various parts of the classroom as well as outside.

## Getting the sound sensor to alert when it is too loud



Figure 54

The last thing we will do with this sensor is to get it to play an alarm and give a visual cue when the sound level gets too loud.

Again, we have to load the MonkMakes extension. The process for doing this is described on page 13. Every time we start a new program with the MonkMakes Sensor Board we need to get the extension blocks again.

Once we add the sensor board blocks to MakeCode we can use the new sound level block

to help us collect the sound. It will report a sound level between 0 and 100. We tested this using the code shown in Figure 54 and also in MicroPython in Figure 55.

```
sound_2.py x
1 # sound_2
2
3 from microbit import *
4
5 # there isn't a micropython equivalent of the MONK MAKES sound level extension block
6 # so we will build one that acts just the same
7
8 def sound_level():
9     sound = (pin1.read_analog()-511)/100
10    return sound
11
12 # there isnt a micropython equivalent of MakeCode plot bar graph block
13 # so we will build one that acts similarly
14
15 def graph(a):
16     display.clear()
17     for y in range(0, 5): # plot the LED rows and columns
18         if a > y:
19             for x in range(0, 5):
20                 display.set_pixel(x, 4-y, 9) # light pixels (x, y, brightness)
21
22 while True:
23     sound = sound_level()
24     graph(sound) # call the function and provide the parameter(a)
```

Figure 45

Approximate levels captured just through observation are shown in Table 4.

Table 4: Sample sound level data

Sample	Our perception	Sensor average level	App decibels
A	Soft	14	40
B	Medium	23	63
C	Starting to get loud	34	73
D	Quite loud	48	around 78

For classroom ownership, you could lead your class through a discussion of what are conducive sound levels and measure them with a mobile app and the sensor board attached to the micro:bit.

Perhaps get the students to raise their hand when they think the classroom level is great for learning, getting annoying and really annoying. Students could capture readings at these times and use that data to inform their coding.

That is what we will do with the data we captured in Table 4. We will use a sensor average level value of about halfway between soft and medium (so about 17) to be the boundary for conducive levels, and a value between medium and starting to get loud (about 28) to indicate somewhat annoying noise. Anything between 29 and 36 becomes quite annoying noise and above 36 will trigger the alarm for sound that is disruptive to learning levels. These are approximations and your class may come up with a different scale.

### **Algorithms: Expressed in pseudocode/English**

How could these steps be expressed in pseudocode?

```
Get the sound level
```

```
  If the sound level is below or equal to 17
```

```
    Then sound level is OK, show the centre LED
```

```
  Else If
```

```
    the sound level is below or equal to 28
```

```
    Then sound level is starting to get annoying, flash a small square
```

```
  Else If
```

```
    The sound is below or equal to 36
```

```
    Sound level is quite annoying, flash the large square unfilled square
```

```
  Else
```

```
    Level is too high show all LEDs lit, sound alarm for a second
```

Just note that now that we have our data we don't need the 'plot bar graph' block to allow the console device to run so it won't form part of our code.

You could change your code if you want the students to continue to be able to track the sound levels in real time.

You may find you also need to change the code to better represent noise levels in your classroom. The examples shown in this tutorial were not taken in a classroom environment.

## Coding the micro:bit using visual programming

As code, the program appears as shown in Figure 56.

So now we have devices that can measure the temperature, sound and light levels in a classroom and alert the students and teacher when they get to a level that is no longer conducive to effective learning to take place.

Students can take ownership of their environment, based on science, and create a positive atmosphere in which to learn. This is really powerful as it gives the students agency for their own learning.

## Coding the micro:bit using general-purpose programming (MicroPython)

The MicroPython code is shown in Figure 57.

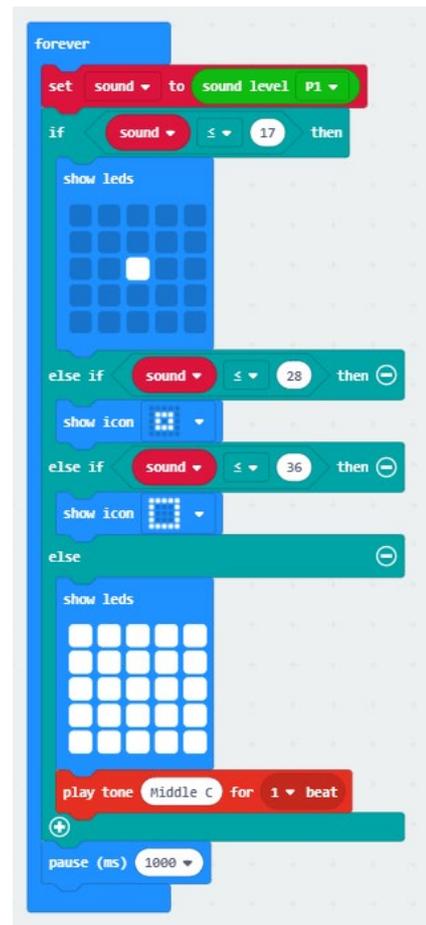


Figure 56

```
sound_3.py X
1 # sound_3
2
3 from microbit import *
4 import math # we need to import a math library to get absolute values
5
6 # there's no micropython equivalent of MONK MAKES sound level extension block
7 # so we will build one that acts just the same
8 def sound_level():
9     sound = math.fabs((pin1.read_analog()-511))
10    return sound
11
12 while True:
13     sound = sound_level()
14     if sound <= 17:
15         display.set_pixel(2, 2, 9)
16         sleep(1000)
17         display.clear()
18     elif sound <= 40:
19         display.show(Image.SQUARE_SMALL, wait=False)
20         sleep(100)
21         display.clear()
22     elif sound <= 60:
23         display.show(Image.SQUARE, wait=False)
24         sleep(100)
25         display.clear()
26     else:
27         for x in range(0, 5):
28             for y in range(0, 5):
29                 display.set_pixel(x, y, 9)
30         sleep(100)
31         display.clear()
32         pin0.write_digital(1)
33         sleep(1000)
34         pin0.write_digital(0)
```

Figure 57

## Part D: Optional extension activities

As extension, you may like your students to attempt to measure carbon dioxide or air pressure levels or to create a master monitoring device.

### ***Carbon dioxide levels***

**Context:** Carbon dioxide (CO<sub>2</sub>) levels play a major part in students' abilities to learn. With as little as 1,000 parts per million and likely lower still, CO<sub>2</sub> induces sleepiness, poor concentration, abnormal heart rates and even nausea, as expressed in an article about a study from the Harvard School of Public Health [www.tinyurl.com/yclw6kzu](http://www.tinyurl.com/yclw6kzu). Similarly, it appears that air pollution has an enormous effect on learning. A study reported on in *The Guardian* [www.tinyurl.com/y92t7yz9](http://www.tinyurl.com/y92t7yz9) suggests that high levels of urban pollution have a major impact on attainment, with some students dropping a whole year of progress over their school lives.

**Challenge:** measure CO<sub>2</sub> and create an alarm when readings reach a certain level.

### ***Air pressure***

**Context:** Air pressure may play a role in affecting cognitive abilities. This is under research; however, the common complaint of sinus headaches when air pressure changes will obviously affect one's ability to learn. Think about how your students behave on a windy day.

**Challenge:** measure air pressure and create an alarm when readings reach a certain level.

### ***Create a master monitoring device***

You may want students to make a central computer which is monitoring all the sensors covered in this tutorial.

## Links to the Australian Curriculum

Tables 5 and 6 give teachers an opportunity to see related aspects of the Australian Curriculum.

Table 5: Aspects of the Australian Curriculum: Digital Technologies Years 5–6 which may be addressed depending on the task.

<p><b>Digital Technologies Achievement standard</b></p>	<p>By the end of Year 6, students explain the fundamentals of digital system components (hardware, software and networks) and how digital systems are connected to form networks. They explain how digital systems use whole numbers as a basis for representing a variety of data types.</p> <p>Students define problems in terms of data and functional requirements and design solutions by developing algorithms to address the problems. They incorporate decision-making, repetition and user interface design into their designs and implement their digital solutions, including a visual program. They explain how information systems and their solutions meet needs and consider sustainability. Students manage the creation and communication of ideas and information in collaborative digital projects using validated data and agreed protocols.</p>		
<p><b>Strands</b></p>	<p>Digital Technologies knowledge and understanding</p> <ul style="list-style-type: none"> <li>Digital systems</li> </ul> <p>Digital Technologies processes and production skills</p> <ul style="list-style-type: none"> <li>Collecting, managing and analysing data</li> <li>Creating designed solutions by: <ul style="list-style-type: none"> <li>investigating and defining</li> <li>generating and designing</li> <li>producing and implementing</li> <li>evaluating</li> </ul> </li> </ul>		
<p><b>Content descriptions</b></p>	<ul style="list-style-type: none"> <li>Examine the main components of common digital systems and how they may connect together to form networks to transmit data (<a href="#">ACTDIK014</a>)</li> <li>Acquire, store and validate different types of data, and use a range of software to interpret and visualise data to create information (<a href="#">ACTDIP016</a>)</li> <li>Design a user interface for a digital system (<a href="#">ACTDIP018</a>)</li> <li>Design, modify and follow simple algorithms involving sequences of steps, branching, and iteration (repetition) (<a href="#">ACTDIP019</a>)</li> <li>Implement digital solutions as simple visual programs involving branching, iteration (repetition), and user input (<a href="#">ACTDIP020</a>)</li> </ul>		
<p><b>Key concepts</b></p>	<ul style="list-style-type: none"> <li>abstraction</li> <li>data collection</li> <li>data interpretation</li> <li>specification</li> <li>algorithms</li> <li>implementation</li> <li>digital systems</li> <li>interactions</li> <li>impact</li> </ul>	<p><b>Key ideas</b></p>	<p>Thinking in Technologies</p> <ul style="list-style-type: none"> <li>computational thinking</li> <li>systems thinking</li> </ul>
<p><b>Cross-curriculum priorities</b></p>		<p><b>General capabilities</b></p>	<ul style="list-style-type: none"> <li>Information and Communication Technology (ICT) Capability</li> <li>Literacy</li> <li>Numeracy</li> </ul>

Table 6: Aspects of the Australian Curriculum: Digital Technologies Years 7–8 which may be addressed depending on the task.

<p><b>Digital Technologies Achievement standard</b></p>	<p>By the end of Year 8, students distinguish between different types of networks and defined purposes. They explain how text, image and audio data can be represented, secured and presented in digital systems.</p> <p>Students plan and manage digital projects to create interactive information. They define and decompose problems in terms of functional requirements and constraints. Students design user experiences and algorithms incorporating branching and iterations, and test, modify and implement digital solutions. They evaluate information systems and their solutions in terms of meeting needs, innovation and sustainability. They analyse and evaluate data from a range of sources to model and create solutions. They use appropriate protocols when communicating and collaborating online.</p>		
<p><b>Strands</b></p>	<p>Digital Technologies knowledge and understanding</p> <ul style="list-style-type: none"> <li>• Digital systems</li> </ul> <p>Digital Technologies processes and production skills</p> <ul style="list-style-type: none"> <li>• Creating digital solutions by: <ul style="list-style-type: none"> <li>– Investigating and defining</li> <li>– generating and designing</li> <li>– Producing and implementing</li> <li>– Evaluating</li> </ul> </li> </ul>		
<p><b>Content descriptions</b></p>	<ul style="list-style-type: none"> <li>• Investigate how data is transmitted and secured in wired, wireless and mobile networks, and how the specifications affect performance (<a href="#">ACTDIK023</a>)</li> <li>• Define and decompose real-world problems taking into account functional requirements and economic, environmental, social, technical and usability constraints (<a href="#">ACTDIP027</a>)</li> <li>• Implement and modify programs with user interfaces involving branching, iteration and functions in a general-purpose programming language (<a href="#">ACTDIP030</a>)</li> <li>• Evaluate how student solutions and existing information systems meet needs, are innovative, and take account of future risks and sustainability (<a href="#">ACTDIP031</a>)</li> </ul>		
<p><b>Key concepts</b></p>	<ul style="list-style-type: none"> <li>• abstraction</li> <li>• data collection</li> <li>• data interpretation</li> <li>• specification</li> <li>• implementation</li> <li>• digital systems</li> <li>• impact</li> </ul>	<p><b>Key ideas</b></p>	<p>Thinking in Technologies</p> <ul style="list-style-type: none"> <li>• computational thinking</li> <li>• systems thinking</li> </ul>
<p><b>Cross-curriculum priorities</b></p>		<p><b>General capabilities</b></p>	<ul style="list-style-type: none"> <li>• Information and Communication Technology (ICT) Capability</li> <li>• Literacy</li> <li>• Numeracy</li> </ul>

## Useful links

Find out more about the micro:bit [www.microbit.org](http://www.microbit.org)

- Code the micro:bit at [www.makecode.org](http://www.makecode.org)
  - Block code within MakeCode: <https://makecode.microbit.org/>

Find out more about variables:

- <https://makecode.microbit.org/blocks/variables/var>
- <https://makecode.microbit.org/courses/csintro/variables>

Teachers or students wishing to explore these activities in a general-purpose programming language including Python and MicroPython:

- Code in Python inside MakeCode: <https://python.microbit.org/v/1.1>
- Python for beginners <https://www.python.org/about/gettingstarted/>
- Code in MicroPython with Mu editor. Download site: <https://codewith.mu/en/download>

## Glossary

**General-purpose programming languages** Programming languages in common use designed to solve a wide range of problems. They include procedural, functional and object-oriented programming languages, including scripting and/or dynamically typed languages. Examples of *general-purpose programming languages* include C#, C++, Java, JavaScript, Python, Ruby and Visual Basic.

**Lux** A measure of the illumination or amount of light produced by something. For example, we can use a light meter/lux meter to measure the light produced by a light bulb.

**Pseudocode** A way of showing algorithms without use of any specific programming language. This makes the algorithm easy to understand for everyone whatever programming language they might use. Pseudocode may be written in English text with some common operation words used. For example: *if* and *else*.

**Variables** Created by programmers to hold the value of data that may change. For example, a variable may be created to hold the player's score in a game.

**Visual programming** A programming language or environment where a program is represented and manipulated graphically rather than as text. A common visual metaphor represents statements and control structures as graphic blocks that can be composed to form programs, allowing programming without having to deal with textual syntax. Examples of *visual programming* languages include: Alice, GameMaker, Kodu, Lego Mindstorms, MIT App Inventor, Scratch (Build Your Own Blocks and Snap).

See also [www.australiancurriculum.edu.au/f-10-curriculum/technologies/glossary/](http://www.australiancurriculum.edu.au/f-10-curriculum/technologies/glossary/)

**Disclaimer:** ACARA does not endorse any product or make any representations as to the quality of such products. This resource is indicative only. Any product that uses material published on the ACARA website should not be taken to be affiliated with ACARA or have the sponsorship or approval of ACARA. It is up to each person to make their own assessment of the product, taking into account matters including the degree to which the materials align with the content descriptions and achievement standards of the Australian Curriculum. The Creative Commons licence BY 4.0 does not apply to any trademark-protected material.

*All images in this resource used with permission*

## Bibliography

Allen, J. G., MacNaughton, P., Satish, U., Santanam, S., Vallarino, J. & Spengler, J. D. (2016). Associations of cognitive function scores with carbon dioxide, ventilation, and volatile organic compound exposures in office workers: a controlled exposure study of green and conventional office environments. *Environmental health perspectives*, 124(6), 805–812.

Barrett, P., Davies, F., Zhang, Y. & Barrett, L. (2015). The impact of classroom design on pupils' learning: Final results of a holistic, multi-level analysis. *Building and Environment*, 89, 118–133.

Carrington, D. & Kuo, L. (2018). Air pollution causes 'huge' reduction in intelligence, study reveals. *The Guardian*, Aug 27. Retrieved from <https://www.theguardian.com/environment/2018/aug/27/air-pollution-causes-huge-reduction-in-intelligence-study-reveals>

Graff Zivin, J., Hsiang, S. M. & Neidell, M. (2018). Temperature and human capital in the short and long run. *Journal of the Association of Environmental and Resource Economists*, 5(1), 77–105.

Heppell, S. (n.d.). Learnometer. Retrieved from <http://www.learnometer.net/>

Mealings, K. (2016). Classroom acoustic conditions: Understanding what is suitable through a review of national and international standards, recommendations, and live classroom measurements. Conference paper, Acoustics 2016 Brisbane. Available at [https://www.researchgate.net/publication/310651345\\_Classroom\\_acoustic\\_conditions\\_Understanding\\_what\\_is\\_suitable\\_through\\_a\\_review\\_of\\_national\\_and\\_international\\_standards\\_recommendations\\_and\\_live\\_classroom\\_measurements](https://www.researchgate.net/publication/310651345_Classroom_acoustic_conditions_Understanding_what_is_suitable_through_a_review_of_national_and_international_standards_recommendations_and_live_classroom_measurements)

Mooney, C. (2015). Paper finds a surprising link between warmer temperatures and math test scores. *The Washington Post*. Retrieved from <https://www.washingtonpost.com/news/energy-environment/wp/2015/05/12/paper-finds-a-surprising-link-between-warm-temperatures-and-math-test-scores/?postshare=7651431446442153>

Park, R. J. (2019). Heat wave: Air conditioned schools would narrow the racial achievement gap. *USA Today*. Retrieved from <https://www.usatoday.com/story/opinion/2019/08/15/heat-wave-students-need-air-conditioning-close-achievement-gap-column/1996394001/>

Picard, M. & Bradley, J. S. (2001). Revisiting speech interference in classrooms. *Audiology*, 40(5), 221–244. Retrieved from <https://pubmed.ncbi.nlm.nih.gov/11688542/>

Romm, J. (2015). Elevated CO2 Levels Directly Affect Human Cognition. *Climate Progress*, Oct, 26. Retrieved from <https://thinkprogress.org/exclusive-elevated-co2-levels-directly-affect-human-cognition-new-harvard-study-shows-2748e7378941/>

Zhang, X., Chen, X. & Zhang, X. (2018). The impact of exposure to air pollution on cognitive performance. *Proceedings of the National Academy of Sciences*, 115(37), 9193–9197. DOI:10.1073/pnas.1809474115