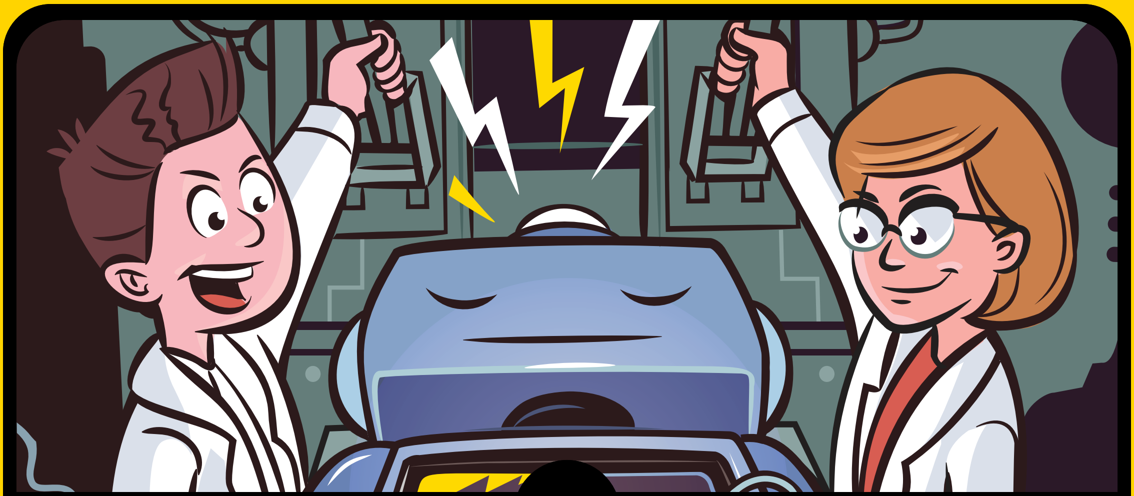


2 SUPER SONIC

In this chapter, we'll explore using sound with the micro:bit. We'll teach the micro:bit to play music and even imitate speech, and we'll get it to hear sound by connecting it to a microphone. You'll try out a couple of experiments and create two simple projects: the first project is a musical doorbell that lets the Mad Scientist know when visitors have arrived, and the second is a Shout-o-meter that measures and displays the volume of sounds it detects.

CONNECTING A LOUDSPEAKER TO A MICRO:BIT

There are a couple of ways to hear sound from your micro:bit. Which one you should choose depends on how much sound you want to make.

The Quiet Method: Headphones

Perhaps the easiest way to get sound from your micro:bit is to use alligator clip cables to connect the micro:bit to a pair of headphones (see Figure 2-1).

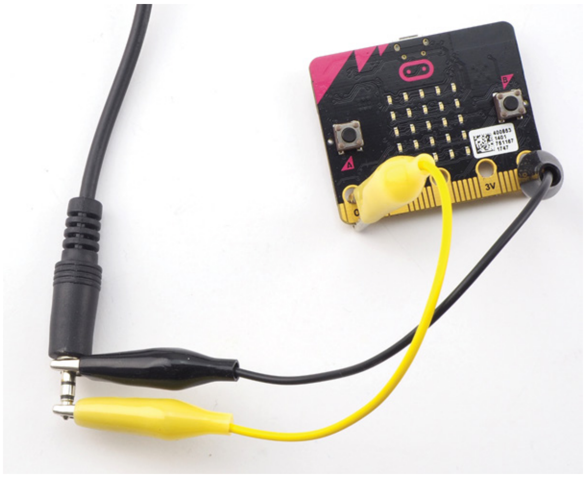


Figure 2-1: Alligator clips attached to headphones

If you look closely at the metal plug on the headphones, you should see that it is really made of three pieces separated by rings of plastic. This means the plug has three connections. The section closest to the headphones' wire is the ground connection. Connect this to the micro:bit's GND (0V) connection with an alligator clip.

The other two connectors are the audio signals for your left and right ears. If you want to hear sound in both ears, place the alligator clip so that it spans both of the two connectors on the end. You can also attach the alligator clip to the very tip for sound in just one ear (as shown in Figure 2-1). Either way, clip the other end of the alligator clip to any of the three micro:bit pins: 0, 1, or 2. Micro:bit users traditionally use pin 0 for audio.

NOTE *Headphones designed for use with a cellphone that include a microphone will have four connectors on the plug rather than three. This shouldn't make a difference. You can still use the tip as the audio connection and the connector closest to the plug body as the GND connection.*

To upgrade this method slightly, you can use an *audio jack adapter* like the one shown in Figure 2-2. Just plug your headphones straight into the adapter, with the black wire connected to GND and the other to pin 0. Adapters like this fit directly onto the headphones and provide a more reliable connection than alligator clips.

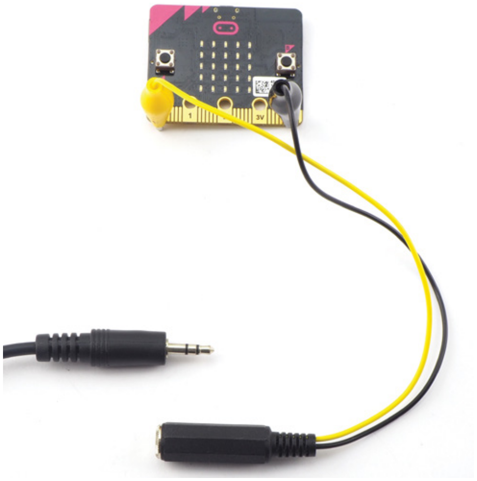


Figure 2-2: An adapter to connect an alligator clip to a 3.5 mm audio jack

The Ghetto Blaster Method: Speaker

With an amplified speaker such as the one shown in Figure 2-3, you can produce a lot more sound using the same connection methods described earlier: either connecting directly to the speaker plug or using an audio jack adapter.

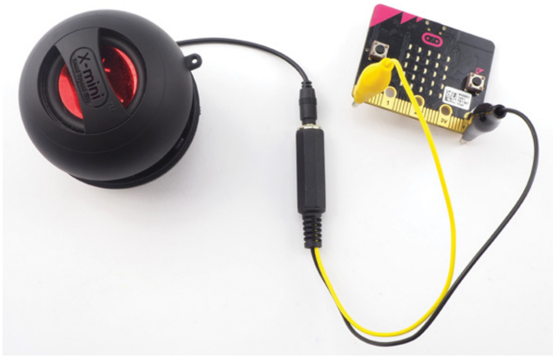


Figure 2-3: Connecting a micro:bit to an amplified speaker

Some speakers are designed especially for use with micro:bits. Some of these have cables that end in alligator clips to attach to your micro:bit, while others, like the Monk Makes Speaker for micro:bit shown in Figure 2-4, end in pins similar to the micro:bit's, making it easy to connect the two with alligator clip cables.

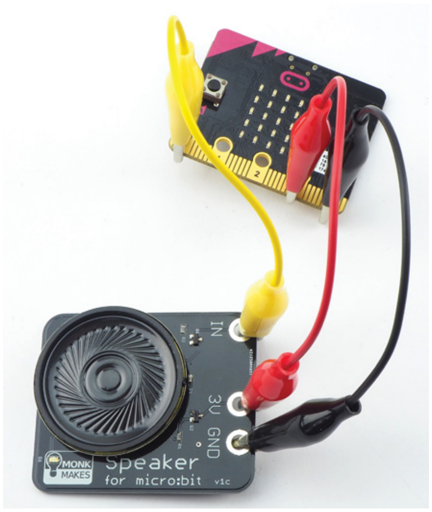


Figure 2-4: The Monk Makes Speaker for micro:bit

Amplified speakers need a power source. In some cases, the amplified speaker may have its own batteries or USB cable. Otherwise, the micro:bit itself could power the speaker, in which case the devices will have to connect in three places: to GND (0V) and 3V on the micro:bit in order to power the speaker and to pin 0 (or one of the other pins) for the audio signal coming from the micro:bit.

Whatever you're using for audio output, let's test it out!

EXPERIMENT 1: GENERATING SOUNDS

In this experiment, you'll learn how to generate sounds using your micro:bit and a loudspeaker or headphones.

What You'll Need

To carry out this experiment, you just need:

Micro:bit

Speaker or headphones

Alligator clip cables

You can find sources for these in the appendix.

Here we'll assume you're using a Monk Makes Speaker for micro:bit and a set of alligator clips, but any of the speaker connection methods listed earlier will work.

Construction

1. Connect the speaker using one of the methods shown in Figures 2-1 to 2-4. Then plug your micro:bit into your computer.
2. Go to *https://github.com/simonmonk/mbms/* to access this book's code repository and click the link for **Experiment 1: Generating Sounds**. Once the program has opened, click **Download** and then copy the hex file onto your micro:bit. If you get stuck, head back to Chapter 1, where we discuss the process of getting programs onto your micro:bit in full.

If you prefer to use Python, download the code from the same website. For instructions for downloading and using the book's examples, see "Downloading the Code" on page 34. The Python file for this experiment is *Experiment_01.py*.

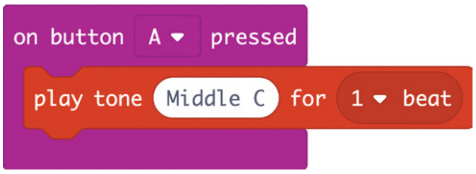
3. Once you've successfully programmed the micro:bit, press **button A**. You should hear a tone through your speaker or headphones!

Code

You won't need much code for this experiment. Whether you use Blocks code or MicroPython, it's just a matter of detecting button A being pressed and then playing a -.

Blocks Code

The Blocks code for this experiment is shown here.



```
on button A ▾ pressed
  play tone Middle C for 1 ▾ beat
```

The code uses the `on button A pressed` block to run the `play tone` block every time button A is pressed. You drop the `play tone` block into the `on button A pressed` block so it clicks into place. Then from the drop-down menu, select the tone you want to hear (in this case `Middle C`) and the duration of the note (`1 beat`).

MicroPython Code

Here's the MicroPython version of the code:

```
from microbit import *
import music

while True:
    if button_a.was_pressed():
        music.pitch(262, 1000)
```

Python has a huge number of *libraries*, which are collections of code that do a specific thing. By asking your code to use these libraries, you get access to a lot of functionality without having to write complicated code yourself. The music library is an example: it contains functions you can use to make your micro:bit make sound. To make MicroPython use the music library, you first import the library using the `import music` command.

While Blocks code will handle some things on its own, like knowing how often to run code and what order to run it in, MicroPython requires you to make that clear in the code itself. Here, you use a `while True:` loop to tell the micro:bit to keep checking whether someone has pressed button A.

When someone does press button A, the note plays using the `pitch` command, which needs two pieces of information: the frequency of the note (`262` is middle C) and the duration of the note in milliseconds (in this case, `1000` milliseconds or 1 second).

Things to Try

You might like to try changing the tone produced. If you are using Blocks code, go back to the browser and click the **Edit** button to alter the code, then click **Middle C**. This will open up a mini keyboard where you can choose a different note to play. To change the note in MicroPython, enter a new number instead of 262 for the frequency. Then click the **Flash** button again. Later in this chapter, you'll learn a better way to choose notes using MicroPython.

You could also try making both buttons A and B play tones and even have them play different tones—a chord!

How It Works: Frequency and Sound

How does the micro:bit create sound in the speaker? Essentially, the micro:bit switches a current (the flow of electricity) on and off incredibly fast, causing part of the speaker to vibrate, creating sound. The speed at which the micro:bit switches the current on and off determines the *frequency* of the sound, and that's what makes different tones. I'll explain this in more detail.

Figure 2-5 shows the parts of a loudspeaker. A rigid, usually metal frame holds a cone in place. The narrow end of this cone is cylindrical and has a coil of wire wrapped around it. Around this coil, fixed to the frame of the loudspeaker, is a strong magnet.

When a current passes through the coil, it—and hence the whole cone—moves back and forth very rapidly. This vibration creates pressure waves in the air that we hear as sound.

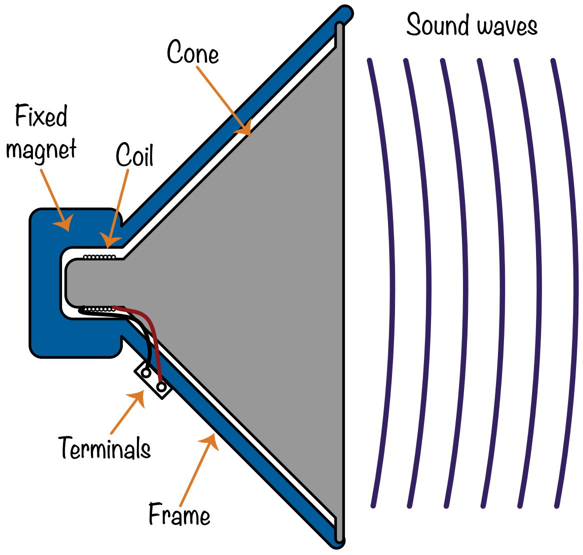


Figure 2-5: A loudspeaker

To make a particular sound, the speaker cone needs to move back and forth a certain number of times per second. The number of times per second the speaker moves is its frequency, measured in *hertz* (shortened to Hz). The higher the frequency, the higher the pitch of the sound. A frequency of 262 Hz corresponds to middle C on a piano. The C an octave higher has a frequency of 524 Hz, or double middle C. In music, when you go up an octave, you double the frequency.

The micro:bit controls the current and therefore the frequency by turning pin 0 on and off very rapidly. When pin 0 is off, it has an output voltage of 0V, and when it is on, it has a voltage of 3V. If you were to draw a chart of the output voltage against time, it would look like Figure 2-6.

For obvious reasons, this type of wave is called a *square wave*. Since a micro:bit's outputs can only ever be on or off, this is the only kind of wave that we can generate from the micro:bit.

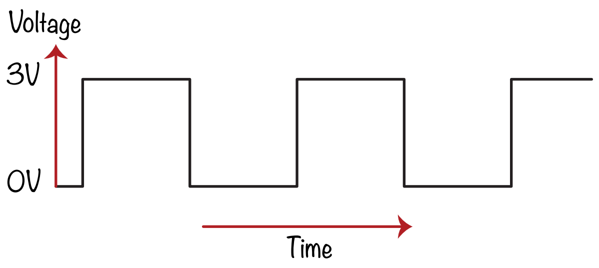


Figure 2-6: A square wave

When the micro:bit sends its signal to the amplified speaker, the speaker takes the low current signal from the micro:bit and increases the current to drive the speaker with more power, making everything louder.

Now let's experiment with making sounds.

EXPERIMENT 2: IT SPEAKS!

The micro:bit's MicroPython software has a really neat feature that allows you to make your micro:bit read out phrases. In this experiment, we will try out this feature and have our micro:bit talk to us.

Although the software that generates the speech was designed for use with English, by experimenting with the spelling, you should be able to make the library speak in other languages.

This feature isn't (at the time of writing) available through the Blocks code, so we'll be using MicroPython.

What You'll Need

This project uses exactly the same hardware as Experiment 1.

Micro:bit

Speaker or headphones

Alligator clip cables

Construction

1. Connect the speaker using one of the methods shown in Figures 2-1 to 2-4.

2. This project uses the speech library, which is not available in Blocks code, so this experiment code is for Python only. Go to *https://github.com/simonmonk/mbms/* and download the *Experiment_02*.py file. You'll also find code for the other projects and instructions for downloading and using the book's examples on the GitHub page. Flash the program onto your micro:bit.
3. Once the micro:bit has been successfully programmed, press **button A** on the micro:bit. You should hear a message being spoken through your speaker or headphones. The Mad Scientist likes to hear this voice as it's a reminder of their dear old friend Professor Hawkins, who alas is no longer with us.

Code

The MicroPython code for the experiment is listed here:

```
from microbit import *
import speech

while True:
    if button_a.was_pressed():
        speech.say("Mad Scientists love micro bits")
```

Aside from importing the speech library, getting the micro:bit to speak is as simple as putting some text for it to say in the say function.

The speech library is quite sophisticated—you can even use it to vary the pitch to make your micro:bit sing! You can find out all about the library at *https://microbit-micropython.readthedocs.io/en/latest/tutorials/speech.html*.

PROJECT: MUSICAL DOORBELL

Difficulty: Easy

The Mad Scientist is particularly partial to a musical doorbell. In fact, you will not be surprised to hear that one of the scientist's favorite tunes is "Imperial March" from *Star Wars*.

In Chapter 10, we will revisit this project, adding a second micro:bit that will make the doorbell work wirelessly.

This project (shown in Figure 2-7) is a variation on Experiment 1, except that instead of playing a single tone when a button is pressed, the doorbell will play tunes. We'll have button A play one tune and button B play another. You can see a short video of the project in action here: *https://youtu.be/xmLupw4PxYQ/*.

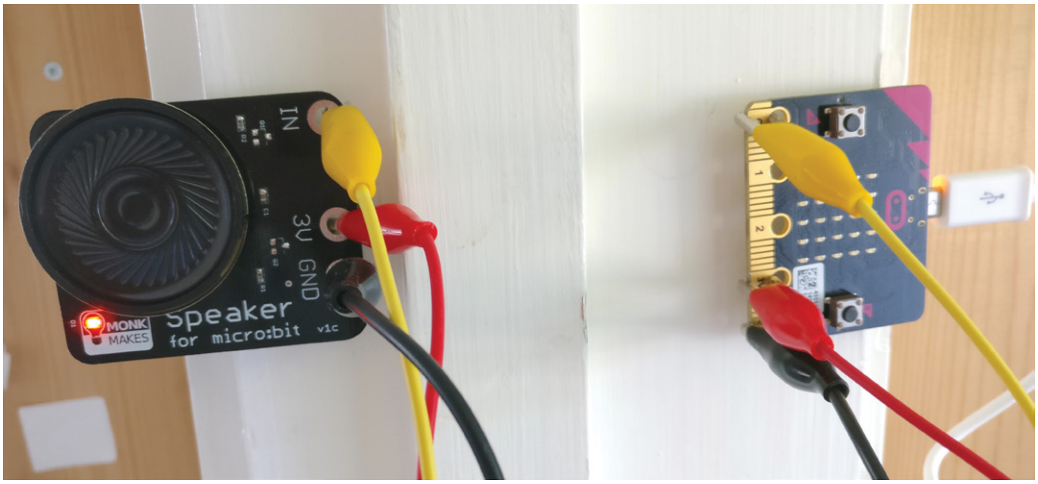


Figure 2-7: The musical doorbell project

Giving the visitor two tunes to choose from allows them to indicate the level of urgency of their visit. Then if the Mad Scientist is busy, they can just ignore the person at the door!

What You'll Need

For this project, you will need the following items:

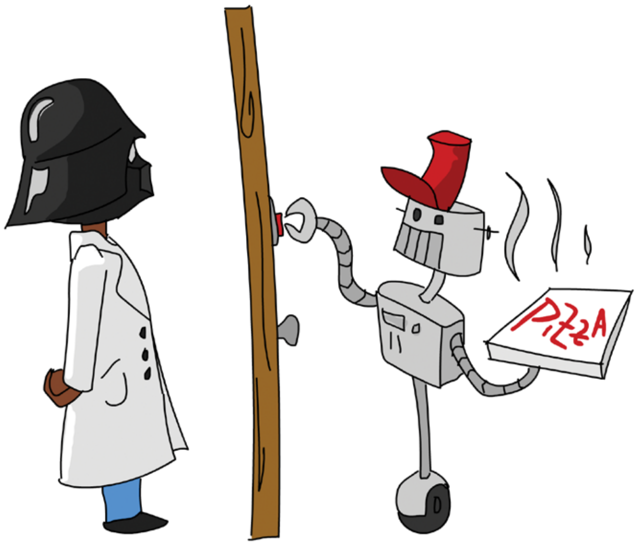
Micro:bit To be the controller for this project and provide two buttons to press

3 × Alligator clip cables To connect the micro:bit to the speaker (Longer cables will make this easier)

USB power adapter or 3V battery pack with power switch To power the micro:bit and speaker

Speaker To play the doorbell tune (I recommend the Monk Makes Speaker for micro:bit)

Adhesive putty or self-adhesive pads To attach the micro:bit to the door frame and the speaker to the inside of the door frame



If you use batteries for this project, it's a good idea to use a battery box with a power switch so that when not in use, the doorbell can be switched off to save the batteries. Otherwise, the batteries will be exhausted after only a day or so of use. A USB power supply offers a longer-term solution that can be left on all the time.

Construction

When building a new project, it's always worth constructing and testing it at your desk before you fit it into place where it will be used.

1. Connect the speaker to the micro:bit using the three alligator cables, as shown in Figure 2-7.

   It's a good idea to use color-coding for your cables, with black for GND, red for 3V, and any other color for the audio connection from pin 0 of the micro:bit. Using different colors will help you keep track of the connections.

2. Go to *https://github.com/simonmonk/mbms/* to access the book's code repository and click the link for **Musical Doorbell**. Once the program has opened, click **Download** and then copy the hex file onto your micro:bit. If you get

stuck on this, head back to Chapter 1, where we discuss the process of getting programs onto your micro:bit in full. If you prefer to use Python, download the code from the same website, along with instructions for downloading and using the book's examples. The Python file for this experiment is *ch_02_Doorbell.py*.

3. Once the micro:bit has been successfully programmed, press **button A** on the micro:bit and you should hear a tune playing (Scott Joplin's "The Entertainer"). Now press **button B** and you will hear Frédéric Chopin's "Funeral March."
4. Once you have everything working, disconnect the micro:bit from your computer and plug it into your USB power adapter or battery box. Test it out again to make sure you've got it working. Then fix the micro:bit part of the project onto one side of your door and the speaker side of the project to the other side of the door. There are a few things to note here:

   Firstly, sticking things to walls, even with adhesive putty, can make a mess, so make sure you get permission if you need to. This is especially true if you are using sticky pads, as these can attach quite permanently to paint.

   Secondly, the alligator clips will need to pass from one side of the door to the other in such a way that they don't get too pinched when the door closes. So work out where they need to go before you start sticking anything down. In Chapter 10, we will make another version of this project that uses a second micro:bit to provide a wireless link.

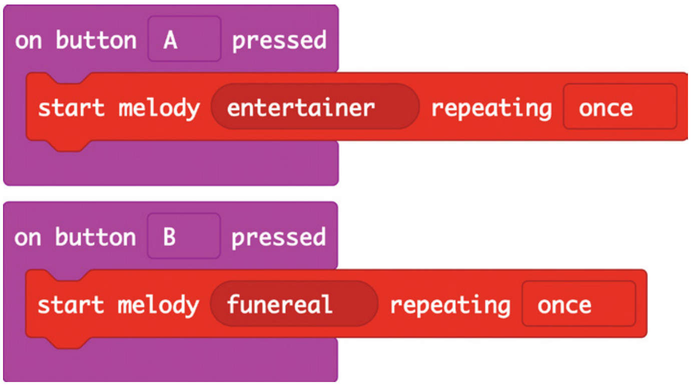
   Finally, if you are using a USB power adapter, you will need a power outlet that's close enough for the USB power adapter to reach your micro:bit.

Code

Now let's talk through the code for the project.

Blocks Code

Here's the Blocks code.



```
on button A pressed
    start melody entertainer repeating once

on button B pressed
    start melody funereal repeating once
```

The code is similar to that of Experiment 1, with a few differences. First, we have two stacks of code: one for button A and one for button B. Second, we choose `once` from the `repeating` menu, because we want the melody to play only once.

Third, we use the `start melody` block to play a whole sequence of notes rather than just a single note. Notice that these tunes are already available in the blocks—you just need to select them from the menu!

MicroPython Code

Here is the MicroPython version of the program:

```
from microbit import *
import music

while True:
    if button_a.was_pressed():
        music.play(music.ENTERTAINER)
    elif button_b.was_pressed():
        music.play(music.FUNERAL)
```

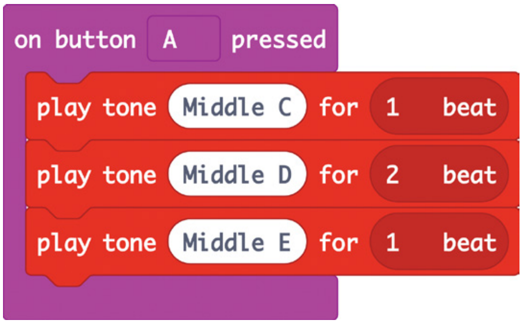
This works exactly the same as the Blocks code. The `music.play` method is equivalent to the `start melody` block, and we use `if` statements to check which button was pressed. The `if` statements allow button A and button B to play different tunes.

The same predefined tunes are available to play in both Blocks and MicroPython code.

Things to Try

Picking from a selection of predefined tunes is all very well, but the Mad Scientist may have particular tastes in music. They may want to compose their own tunes!

If you are using Blocks code, you can make your own tune by creating a list of `play tone` blocks, like the example shown here. You fill out the notes you want played, and each note is played in turn.



```
on button A pressed
    play tone Middle C for 1 beat
    play tone Middle D for 2 beat
    play tone Middle E for 1 beat
```

So if you know all the notes for a particular tune, you can create it like this. You'll also need to specify how long each note needs to play. You may have to experiment a bit before you get your music to sound just the way you want.

Now let's see how to create a tune in MicroPython:

```
from microbit import *
import music

notes = ['A4:4', 'A', 'A', 'F:2', 'C5:2', 'A4:4', 'F:2',  'C5:2',
 'A4:4']

while True:
    if button_a.was_pressed():
        music.play(notes)
```

The music library for MicroPython takes care of playing whole tunes by letting you use a special notation to write your own melodies. Each note is made up of a string of characters (see Chapter 1 for more information on strings). The first character of the string is the note name (a letter A to G). Next comes an octave number—middle C is in octave number 4, so you will probably want to restrict your tune to around octaves 3, 4, and 5. The octave number is optional, and if you don't give it, Python will assume you want the first octave.

Once you specify an octave number, the music library will assume that octave applies to all following notes until you specify a different octave number.

Next, you can optionally put a colon followed by a duration. The duration is measured in quarter-notes. For example, to play middle C for a half-note, you would write `C4:2`.

To string together several notes, you have to create a *list*. So far we've used variables that hold only a single element. A list is like a variable that can hold multiple elements, and you can access and use each element independently. To indicate that the `notes` variable contains a list of values, rather than just a single value, you'll separate the array values by commas and enclose the whole thing between `[` and `]`.

In our array, each element is a note string. To play the whole sequence of notes, you use the `play` function, providing it with the list of notes to play. This example plays the opening few notes from the *Star Wars* "Imperial March."

Here, you see we import the usual microbit library, as well as the music library. We save our tune in a variable called `notes`. Then we make another `while True:` loop so that the code keeps running and checking whether the button was pressed. We tell the program that if button A is pressed, it should play the `notes` variable.

PROJECT: SHOUT-O-METER

Difficulty: Easy

The Mad Scientist likes to measure things. To that end, this project makes a simple sound meter that indicates the volume of a noise. Then the scientist can tell the neighbors off for making too much noise—and prove they really are.

What You'll Need

For this project, you need a microphone to pick up sounds so you can measure their volume. I'm going to use the microphone built into the Monk Makes Sensor, which has a bunch of sensors. The sound's volume then appears on the micro:bit's LED display. The louder the sound, the more LEDs will light up.

For this project, you will need the following items:

Micro:bit To be the controller for this project and provide two buttons to press

3 × Alligator clip cables To connect the micro:bit to the speaker (Longer cables will make this easier)

Any micro:bit power source Can be the USB computer cable or a battery box

Monk Makes Sensor for micro:bit To supply a microphone

Construction

1. Connect the sensor board to the micro:bit using the three alligator clips, as shown in Figure 2-8. You need to connect 3V on the sensor to 3V on the micro:bit, GND to GND, and the hole with the microphone picture to pin 0 on the micro:bit.

   It's a good idea to stick to the color-coding of the cables, with black for GND, red for 3V, and any other color for the microphone connection from pin 0 of the micro:bit.

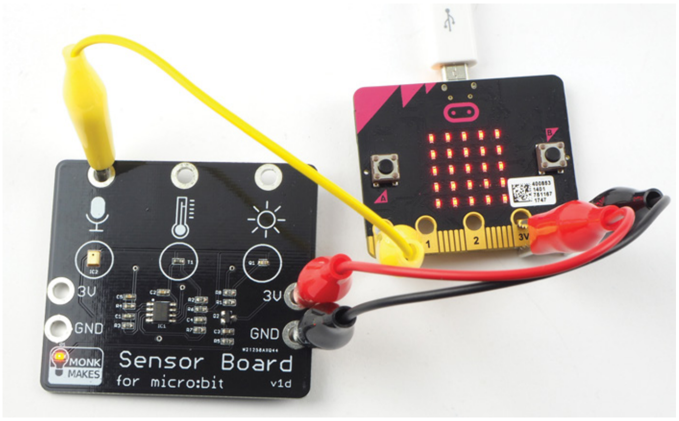


Figure 2-8: The Shout-o-meter project

2. Go to *https://github.com/simonmonk/mbms/* to access the book's code repository and click the link for **Shout-O-Meter**. Once the program has opened, click **Download** and then copy the hex file onto your micro:bit. If you get stuck on this, head back to Chapter 1, where we discuss in detail how to get programs onto your micro:bit.

   If you prefer to use Python, then download the code from the same website, along with instructions for downloading and using the book's examples. The Python file for this experiment is *ch_2_Shoutometer.py*.

3. Once you've programmed the micro:bit, try whistling near the microphone (Figure 2-9) and notice how the LEDs jump in response to the sound level. You can also try tapping the microphone. You can see a short video of the project in action here: *https://youtu.be/6pGDSHmfFng/*.

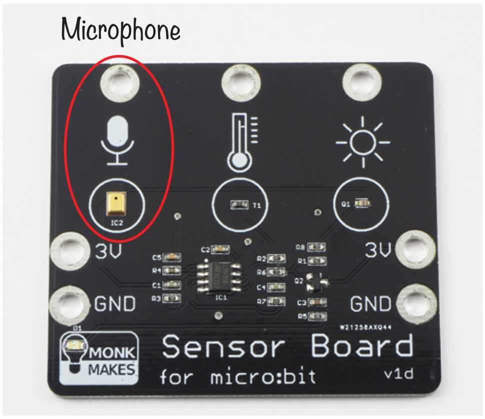


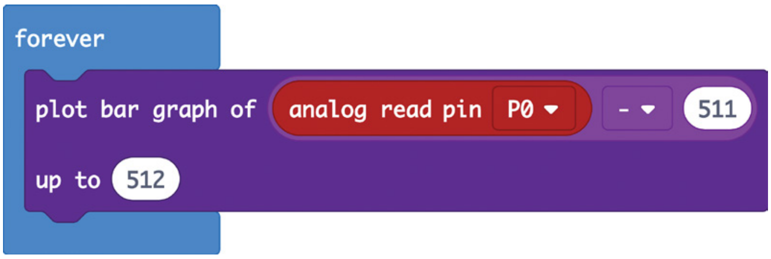
Figure 2-9: The Monk Makes Sensor for micro:bit microphone

Code

The Blocks version of this code is able to make use of the built-in `plot bar graph of` block, whereas the MicroPython version is more complicated because we have to implement our own version of this feature.

Blocks Code

The Blocks language includes a useful block called `plot bar graph of` that makes the code for displaying the sound level really easy.



```
forever
  plot bar graph of  analog read pin P0 - 511
  up to 512
```

We put a `forever` block in, so the code inside is constantly running. Then we add the `plot bar graph of` block, which will display the loudness from the microphone.

As you can see, the analog value read from pin 0 of the micro:bit has `511` subtracted from it before being passed to `plot bar graph of` with a maximum value of `up to` set to `512`. The reason for this bit of math is discussed in “How It Works: Microphone Output” on page 59.

Getting the right blocks assembled can be tricky, especially when it comes to math. Fortunately, the editor allows you to freely move blocks around, so if they are not in the right place to give you the results you want, you can just drag them to where they should be. See Chapter 1 for more information on editing code.

MicroPython Code

The MicroPython version of the code is a little more complicated than the Blocks code. MicroPython does not have a built-in bar graph display, so we have to write our own. The `plot bar graph of` block provides a nice, smooth display, despite rapidly changing data. To get the same result in MicroPython, I had to add code to read the maximum sound level from 10 samples.

```
from microbit import *

def sound_level():
    max_level = 0
    for i in range(0, 10):
        sound_level = (pin0.read_analog() - 511) / 100
        if sound_level > max_level:
            max_level = sound_level
    return max_level

def bargraph(a):
    display.clear()
    for y in range(0, 5):
        if a > y:
            for x in range(0, 5):
                display.set_pixel(x, 4-y, 9)

while True:
    bargraph(sound_level())
    sleep(10)
```

We use the `sound_level` function and make a `for` loop to take 10 samples of sound. Each sample value is (as with the Blocks version of the code) the analog value with `511` subtracted from it. However in this case, to scale down the number of rows to be lit to 0 to 4, we divide the resulting value by 100. We then compare the sound level stored in the variable `sound_level` to the variable `max_level` and, if it is greater, `max_level` is changed to be the `sound_level`. When all 10 samples have been taken, the largest one will be in `max_level`, and this value is returned by the function.

The `bargraph` function takes a value, represented by `a`, to display. The higher the value, the more LEDs will be lit, indicating a louder noise. This value should be between 0 and 4. However, if it is greater than 4, it doesn't matter—all the LEDs in the display will turn on, but nothing else will happen. The function works by looping over each row of the display, and, if the value of `a` is greater than the row number, every LED on that row is illuminated by the inner `for` loop that asks whether `x` is in the range of 0 to 4.

All the main `while` loop has to do is call the function `bargraph`, supplying it with the sound level returned by the function `sound_level`.

How It Works: Microphone Output

Figure 2-10 shows a graph of the output of the microphone when it is detecting sound. Voltage is on the vertical axis, and time in on the horizontal axis.

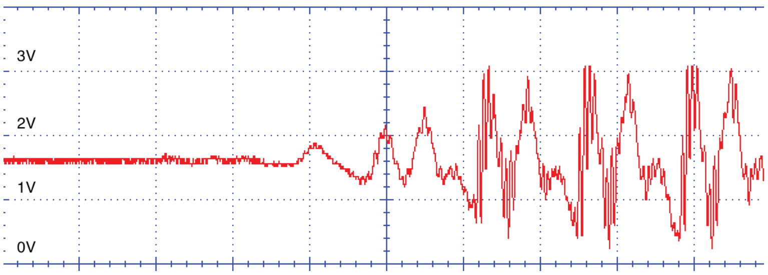


Figure 2-10: A sample of sound

As you can see from the left-hand side of the plot, before the sound starts, the output voltage from the sensor is level at about 1.5V. When the sound starts, the voltage oscillates above and below this 1.5V value as the microphone picks up the pressure waves of the sound. A reading of 1.5V on the micro:bit gives an analog value of 511. This is why we subtract 511 from the reading before displaying it on the micro:bit; otherwise, half the LEDs would be on during silence.

SUMMARY

In this chapter, the Mad Scientist explored the world of sound, both by producing music and speech from the micro:bit and by detecting sound using a microphone. We have started our exploration of the micro:bit with a couple of easy projects.

In the next chapter, we will take a look at light. We'll measure light with a special sensor and use the micro:bit's LED display. Then we'll tackle a large project, using the multicolored NeoPixel display and combining light with sound to make a light-controlled musical instrument. After that, we'll move on to other, even more challenging projects.