

1. Introduction

1.1. Tinker Kit Introduction

ElecFreaks Micro:bit Tinker Kit is a set of Micro:bit accessory that enables you to connect all kinds of modules easily without a group of messy breadboard wires.

1.2. Components

Crystal Battery Box - 2 x AAA x 1

Elecfreaks Micro:bit Breakout Board x 1

Micro-B USB Cable x 1

OLED Display x 1

Mini Servo(1.6kg) x 1

Octopus PIR sensor Brick x 1

Octopus Soil Moisture Sensor Brick x 1

Octopus ADKeypad x 1

Octopus Crash Sensor Brick x 1

Octopus Passive buzzer Brick OBPB01 x 1

Octopus 5mm LED Brick OBLED - Red x 1

Octopus 5mm LED Brick - Green x 1

Octopus 5mm LED Brick OBLED - Blue x 1

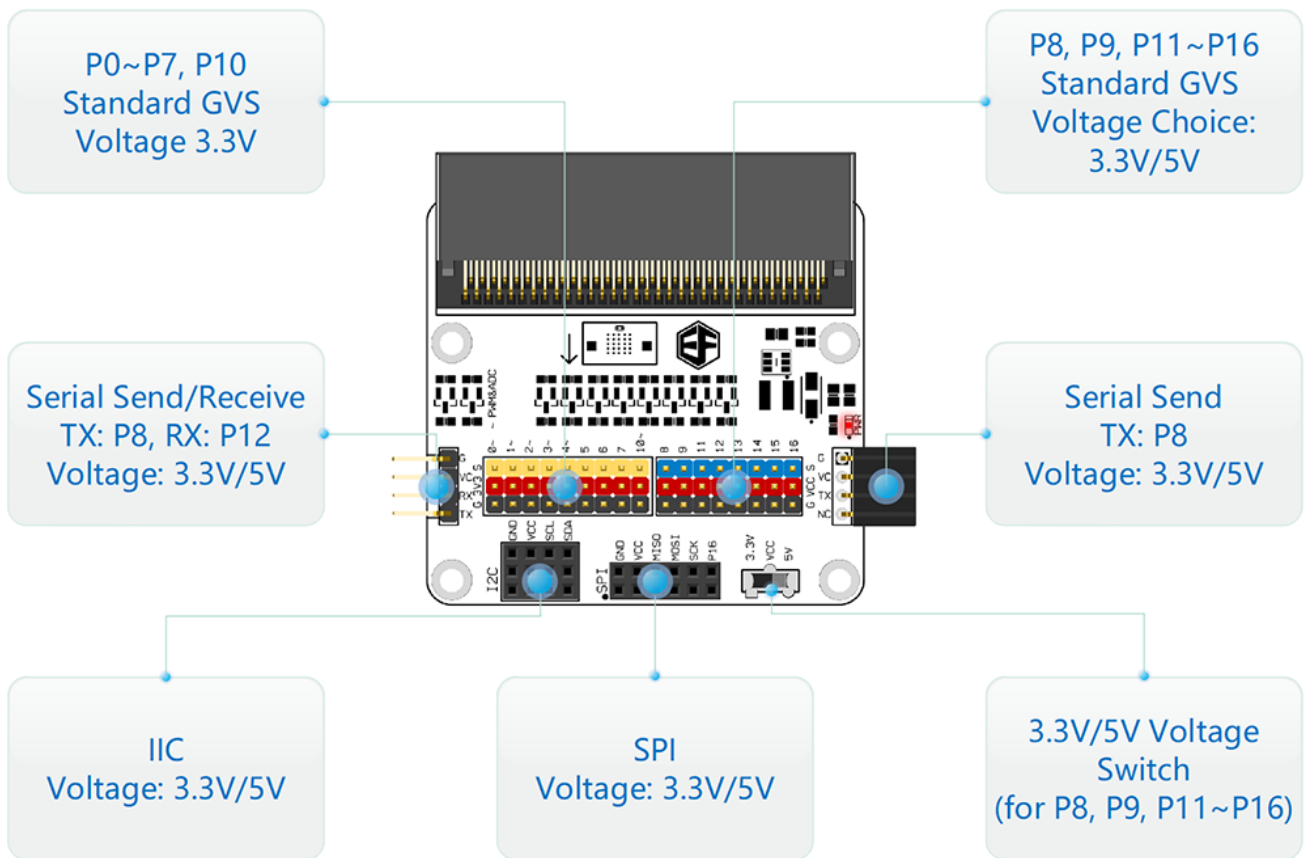
Octopus Analog Rotation Brick OBARot x 1

Octopus Crash Sensor Brick x 1

2. Octopus:bit Introduction

2.1. Introduction

ELECFREAKS Octopus:bit is a kind of breakout boards for micro:bit. It can lead out GPIO port, serial port, IIC port, and SPI port on the micro:bit board. The biggest feature of Octopus:bit is that it can switch electric level for some GPIO ports, which makes micro:bit available to be adapted to 5V sensors.



2.2. Shipping List

1 x ELECFREAKS Octopus:bit

2.3. Hardware

Features

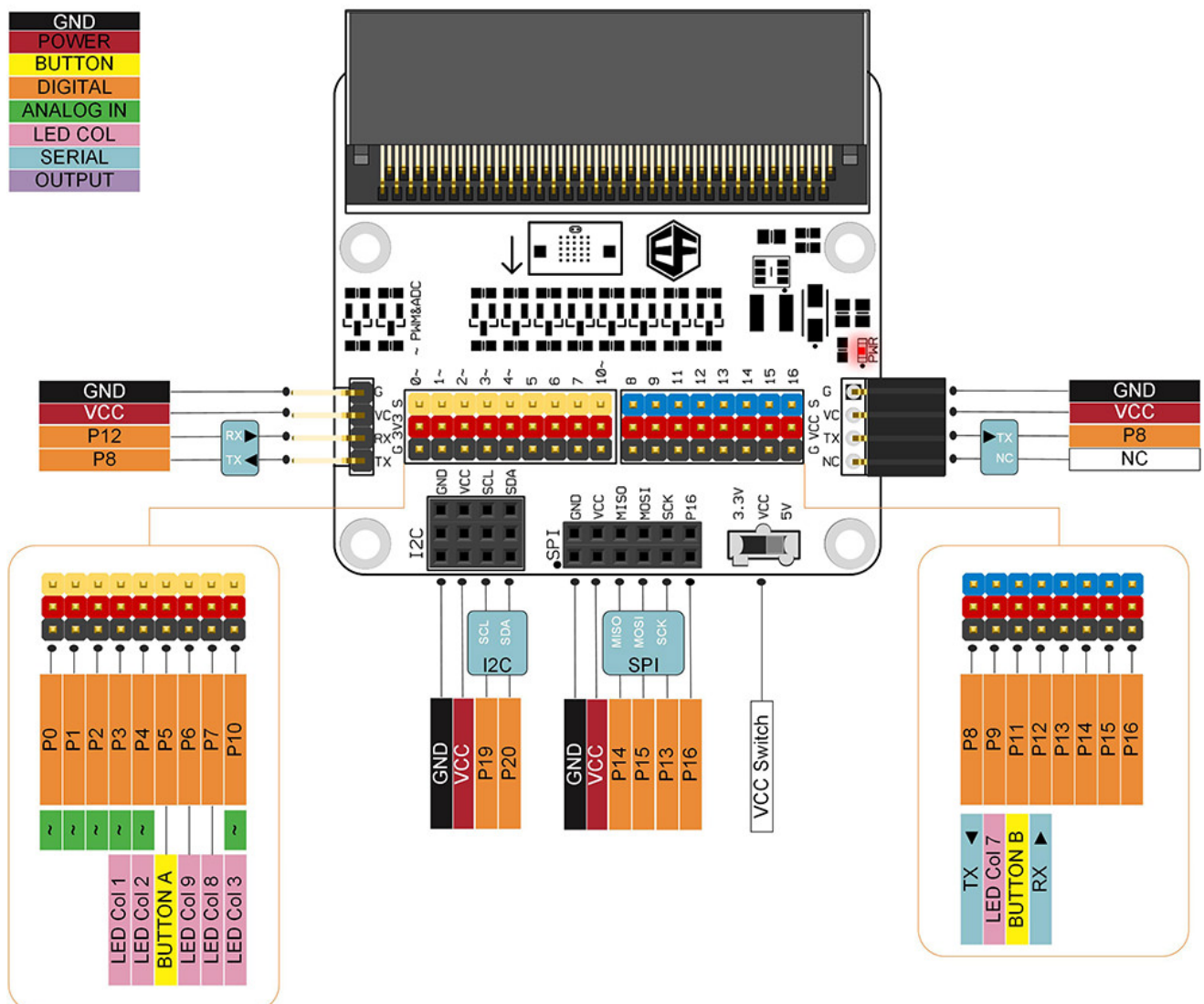
- Input voltage: 3.3V (powered by the edge connector of micro:bit)
- Extend all of GPIO ports(P0~P16, P19~P20).
- Beneath each I/O port, there are pins for VCC and GND. These pins are differentiated by different colors, which enable you to connect your extension module easily. The spread of pins is fully compatible with Octopus series' products.
- With a voltage boosting module, you can shift the working voltage of P8, P9, P11~P16 between 3.3V and 5V through the voltage switch.
- Lead out serial port, I2C port and SPI port, among which I2C can connect 3 channels of I2C devices and SPI can connect 2 channels of SPI devices.
- Available for direct serial port communication between two breakout boards.

Application

It is suitable for all conditions that require micro:bit GPIO such as programming education, smart device creation, and so on.

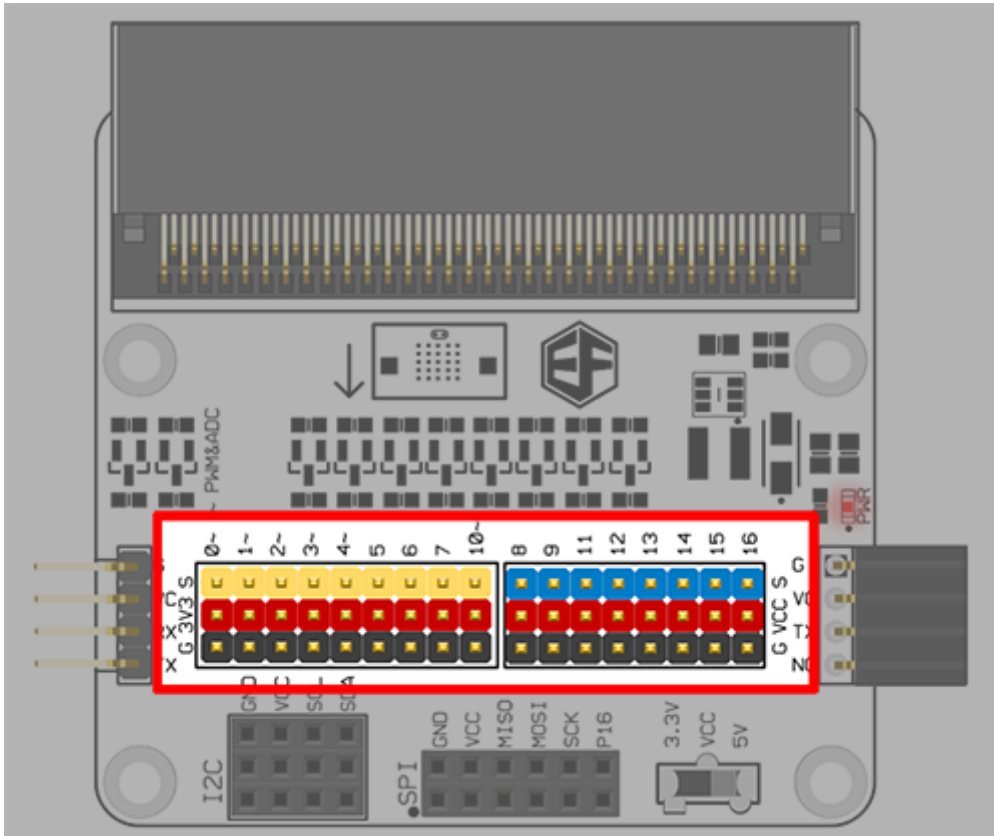
Pins & Connectors

ELECFREAKS OCTOPUS:BIT V1.6



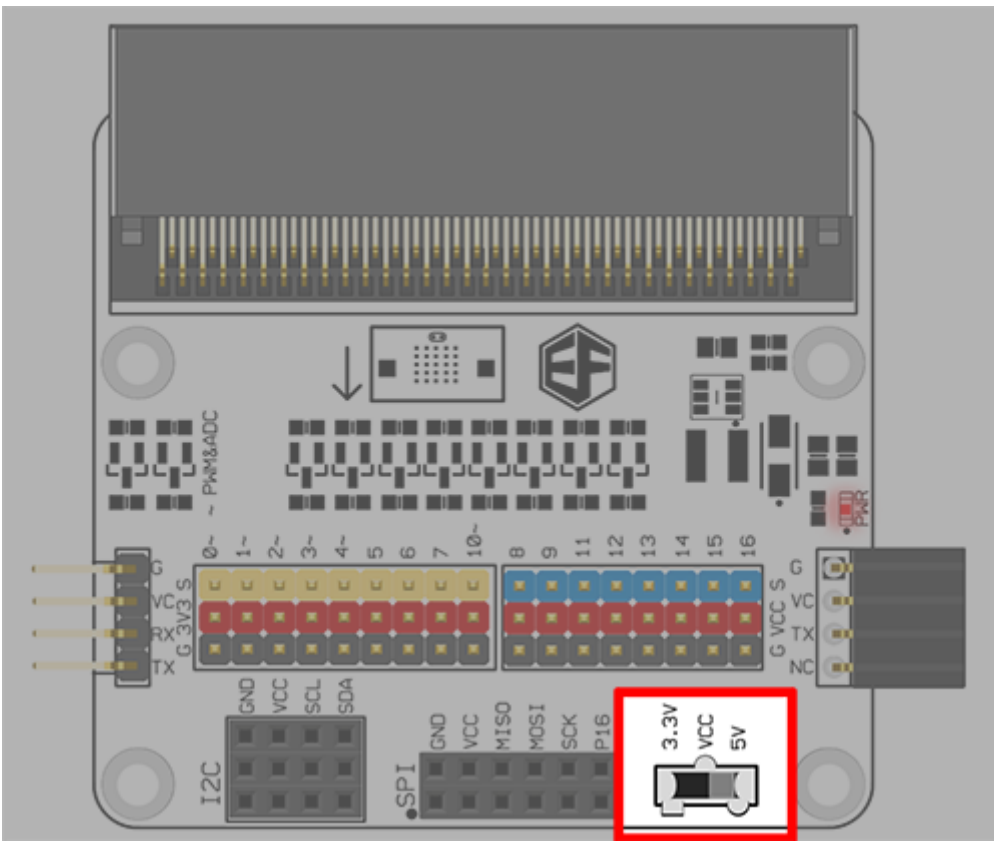
2.4. More Details

Standard GVS Port



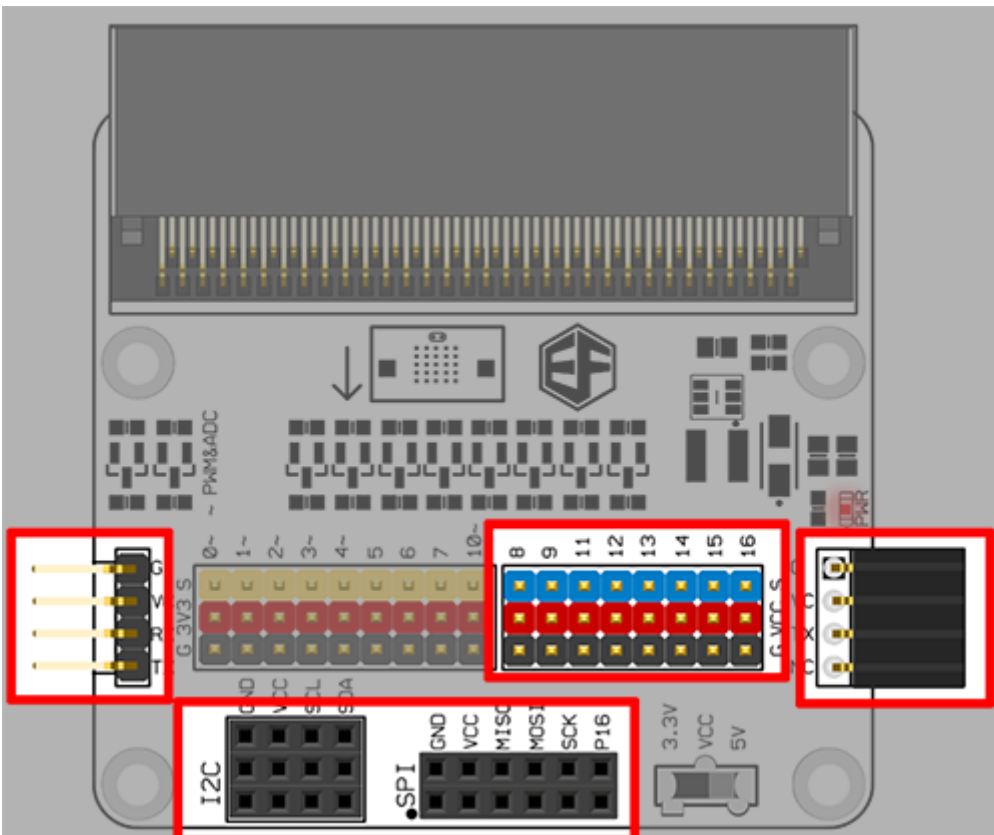
Among the standard GVS ports, the working voltage of the yellow part (P0~P7, P10) is 3.3V, while the working voltage of the blue part (P8, P9, P11~P16) can be shifted between 3.3V and 5V through a voltage switch. Beneath each I/O port, there are pins for VCC and GND. These pins are differentiated by different colors, which enable you to connect your extension module easily. The spread of pins is fully compatible with Octopus series' products.

Voltage Switch

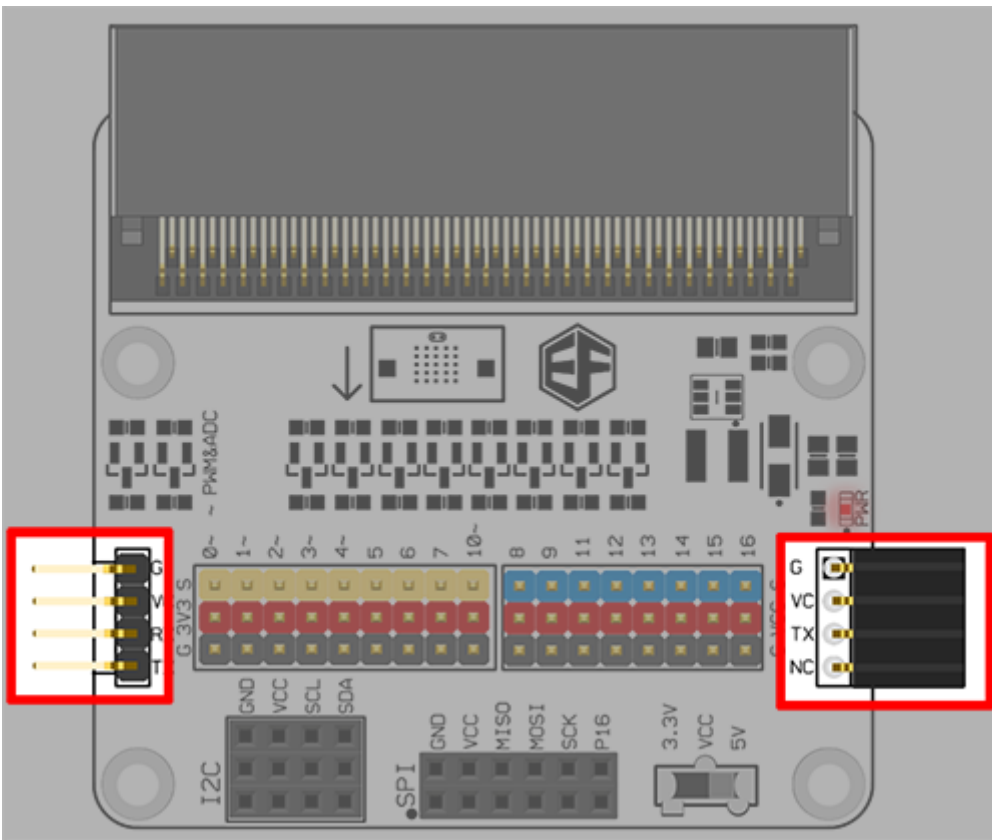


Sliding this switch, we can change the voltage of the blue IO ports (P8, P9, P11~P16) between 3.3V and 5V.

You can see its working range in the below:

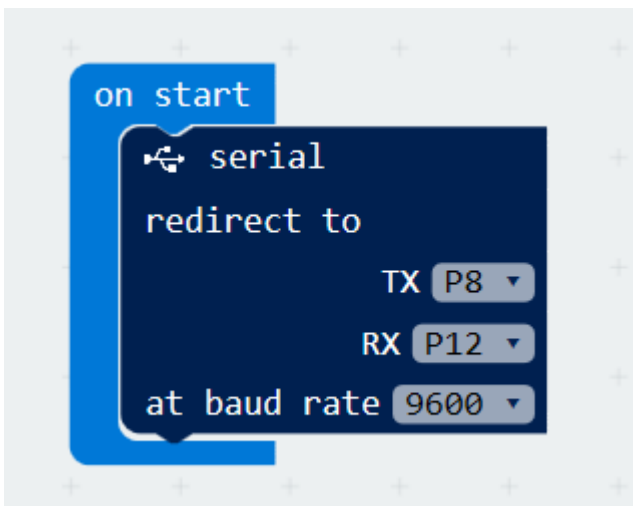


Serial Port

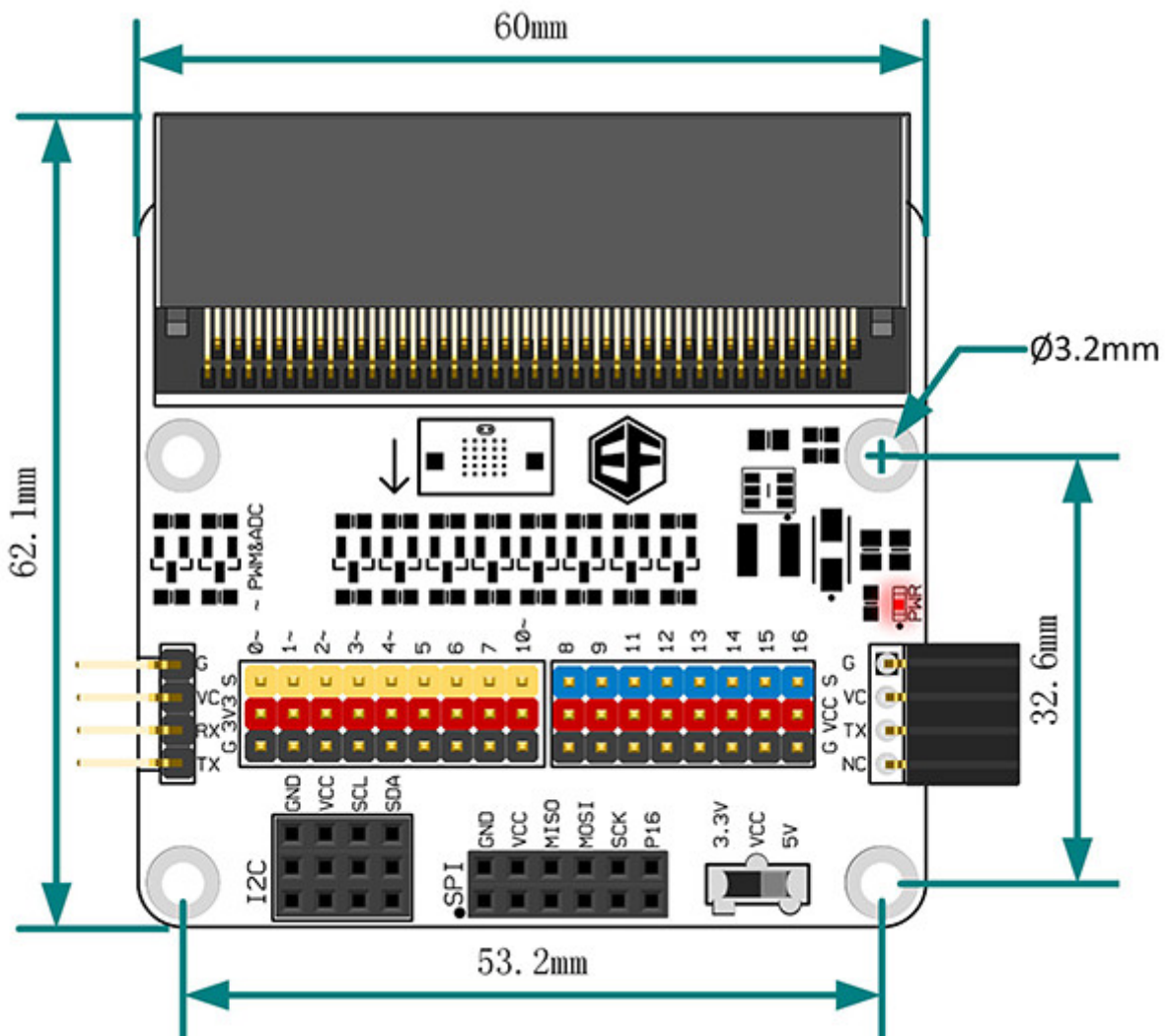


The working voltage of serial port is available to be shifted between 3.3V and 5V through the voltage switch. Connect TX to P8, RX to P12. The left pins are bidirectional serial port, which can run both input and output. The right female header is a one-way output serial port.

Note : To use this port, we have to initialize it according to the program in the below:



2.5. Dimension

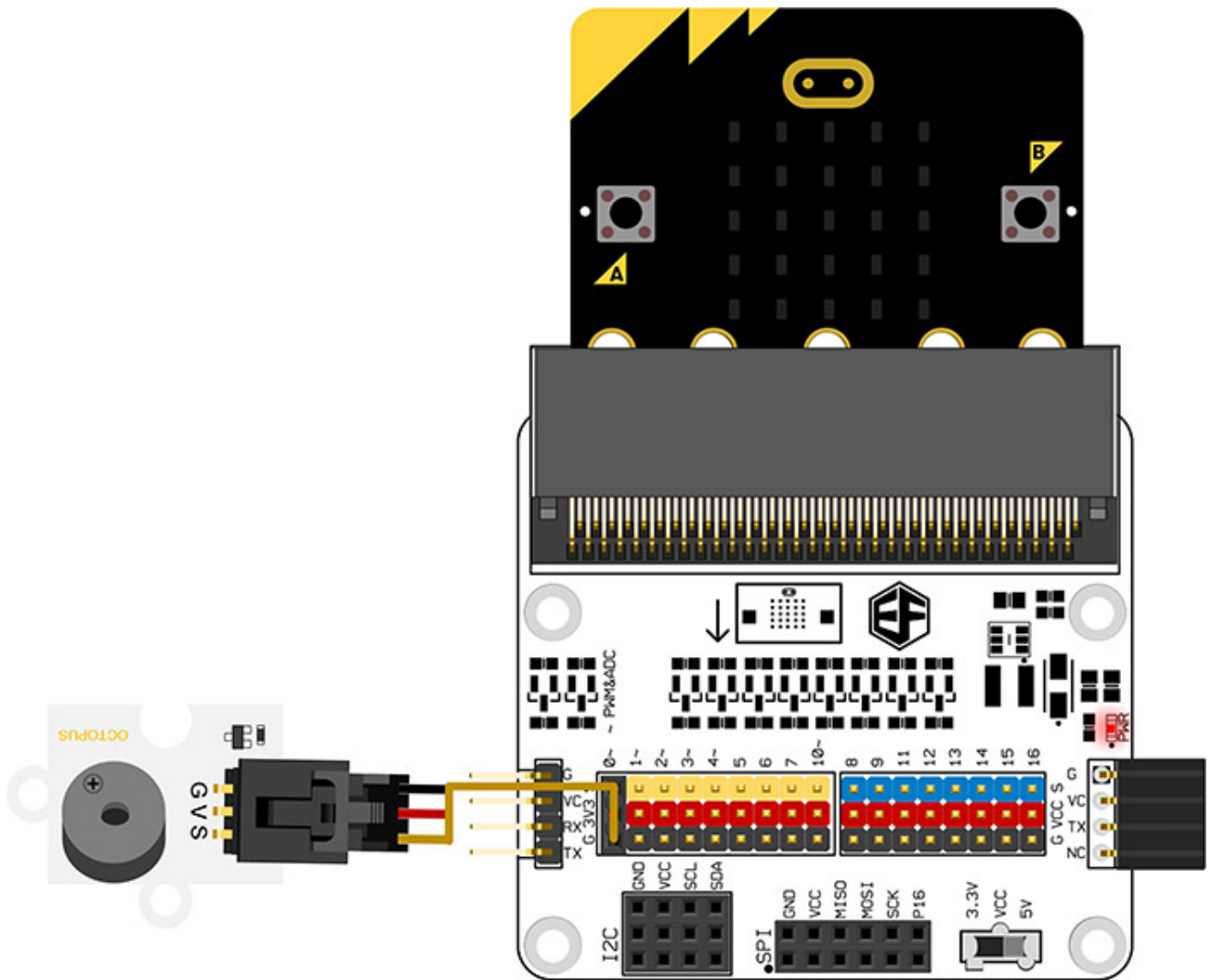


2.6. Software

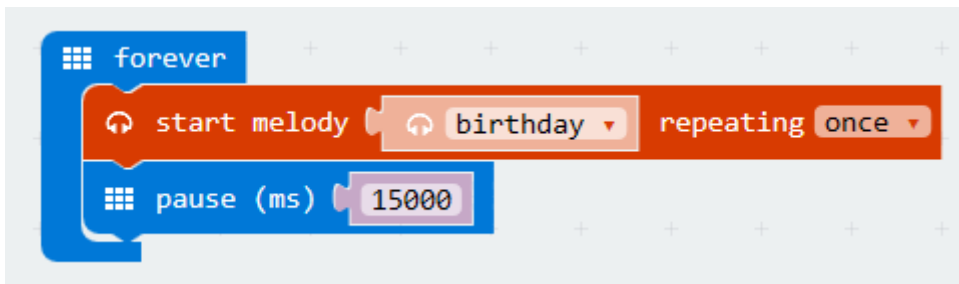
Example 1 Music Broadcast

Hardware Connection

Connect passive buzzer module to PO.



Code Example



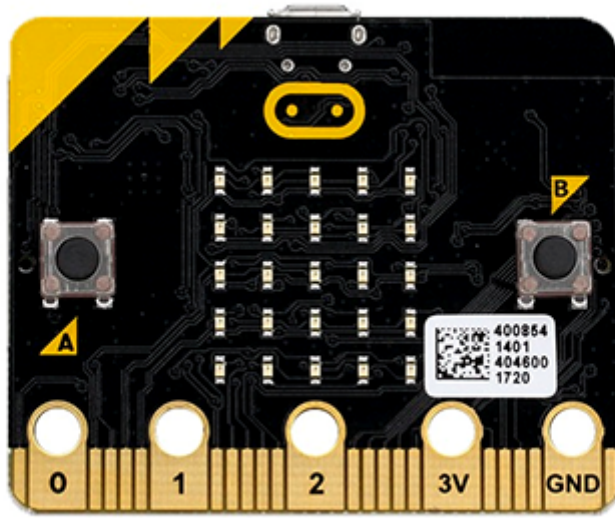
You can download the whole program from the link here:

https://makecode.microbit.org/_fAmC3WERHdR2

Download the whole program into your micro:bit, the buzzer will play Happy Birthday again and again in round.









2.7. Relative Components

BBC micro:bit



Octopus Bricks Series

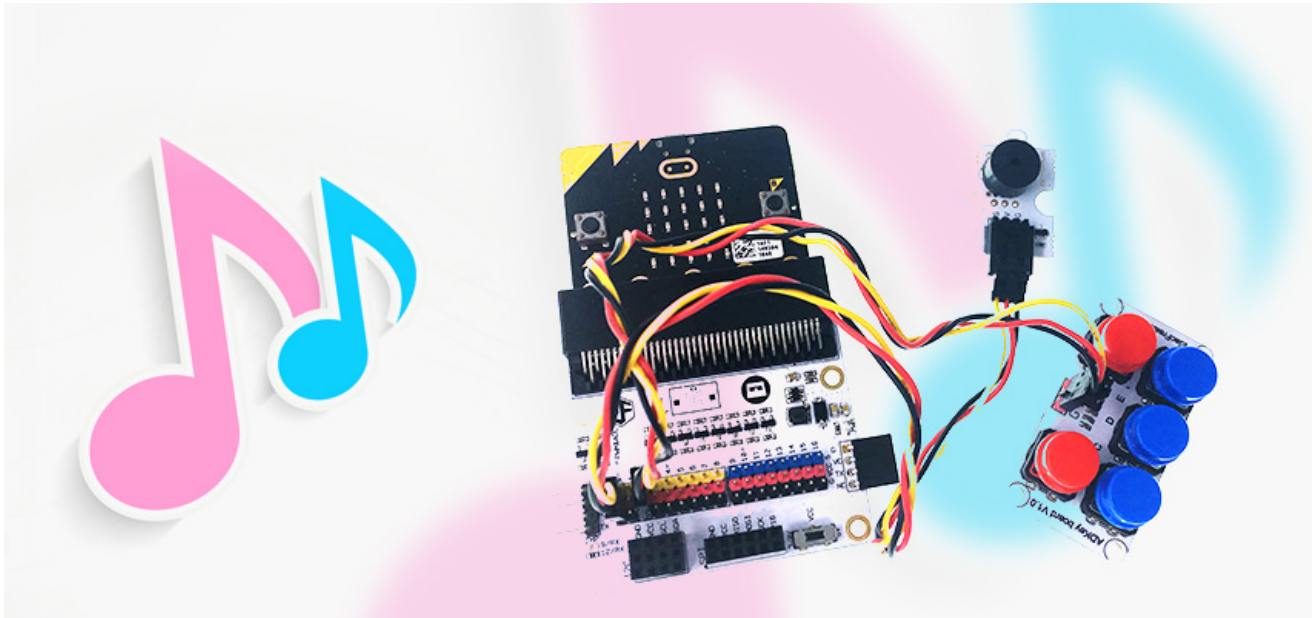
Arduino	+				
Octopus Bricks	+				
Micro:bit	+				
Raspberry Pi	+				
Modules	+				
Freaks X	+				
Wireless	+				
Robotics	+				
Prototyping	+				
Kits	+				
Tools	+				
PCB Service					
Stock Clearance					

 Joystick breakout module BKOUT_MD01 \$3.20 Add to Wish List	 Octopus Linear Slider Potentiometer Brick \$4.98	 Octopus ADKeypad \$4.50	 Octopus Touch PAD Sensor \$3.07
 Octopus Mercury Sensor \$2.91	 Octopus Sound Sensor \$3.15	 Octopus Tilt Sensor \$2.29	 Octopus Crash Sensor Brick \$1.88

ElecFreaks Micro:bit Tinker Kit



3. case 01 Music Machine

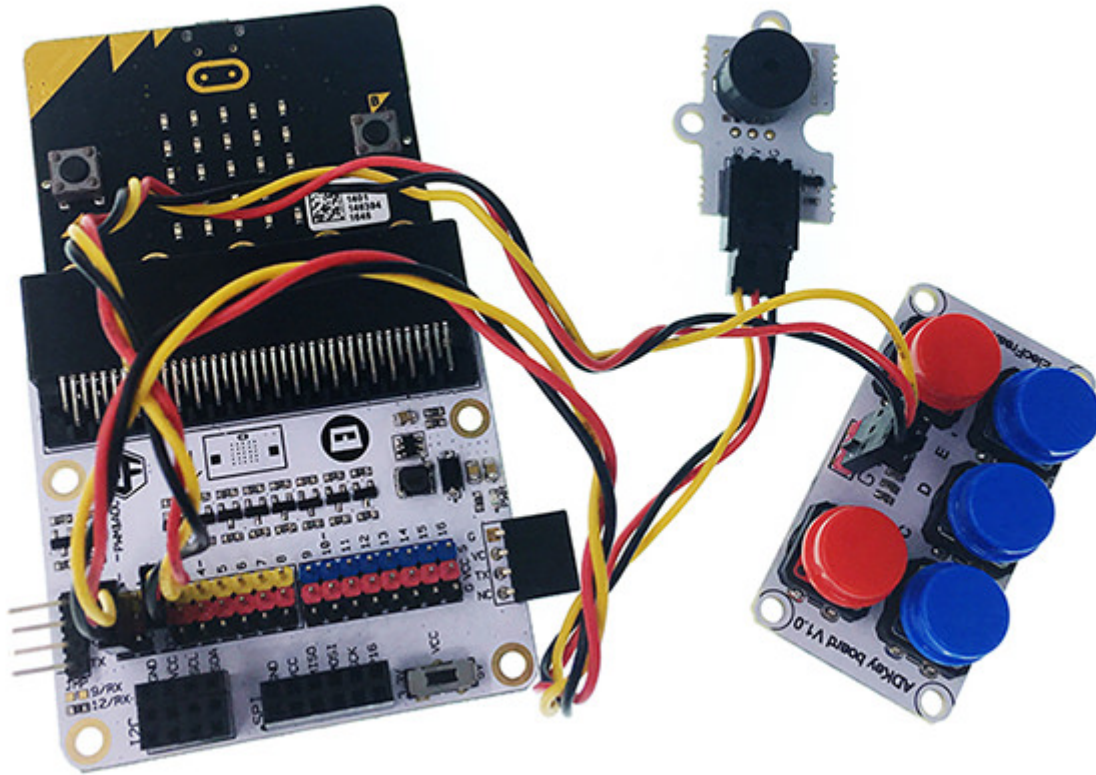


3.1. Goals

- Get to know the ADKeypad.
- Make something with ADKeypad.
- Make something with Buzzer.

3.2. Materials

- 1 x BBC Micro:bit Board
- 1 x Micro USB cable
- 1 x ElecFreaks Micro:bit Breakout Board
- 1 x Octopus Passive buzzer Brick
- 1 x Octopus ADKeypad



Tips: If you want all components above, you may need ElecFreaks Micro:bit Tinker Kit

3.3. How to Make

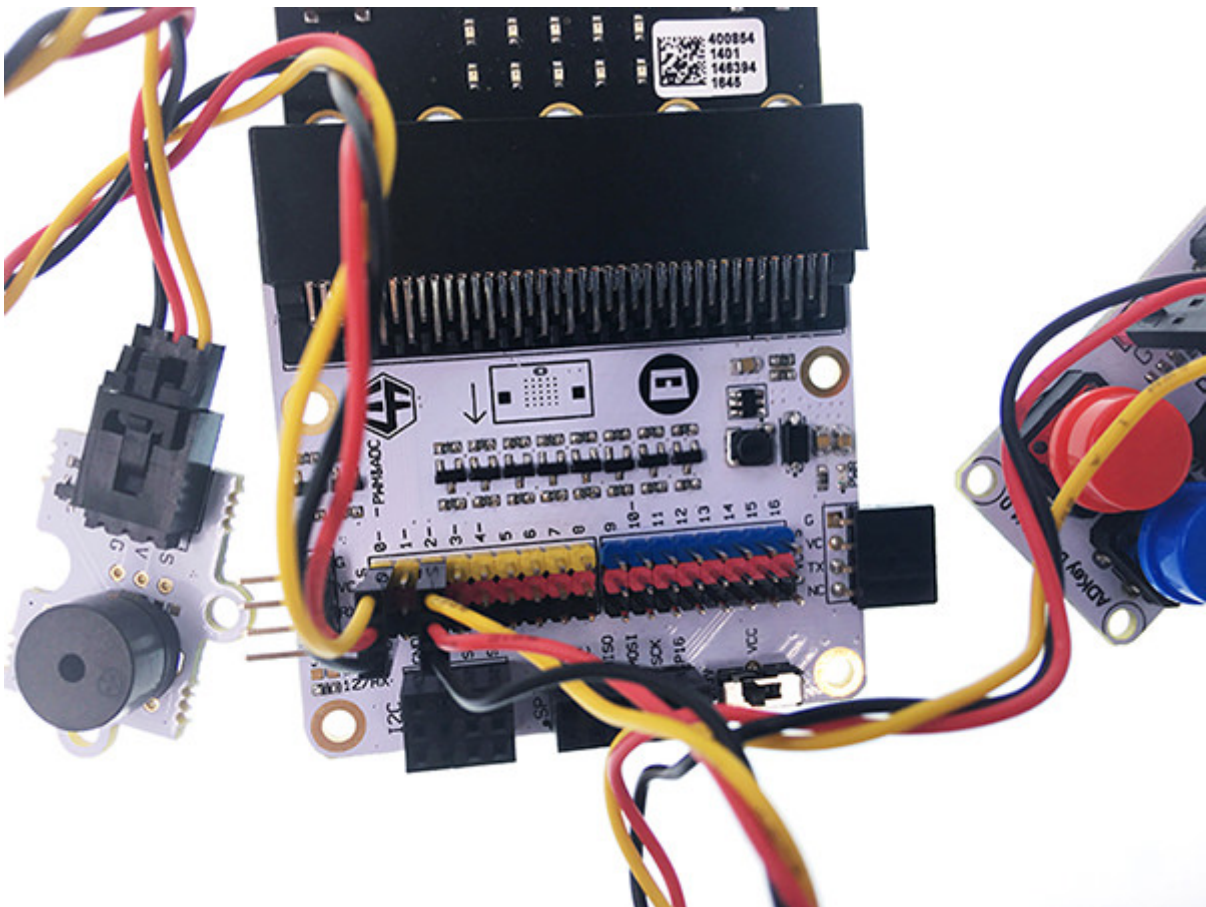
Step 1

After connecting one end of the USB cable to your computer, connect the other end to the micro:bit as shown in the picture below. Then connect the side of the micro:bit where the pins are located to the breakout board.



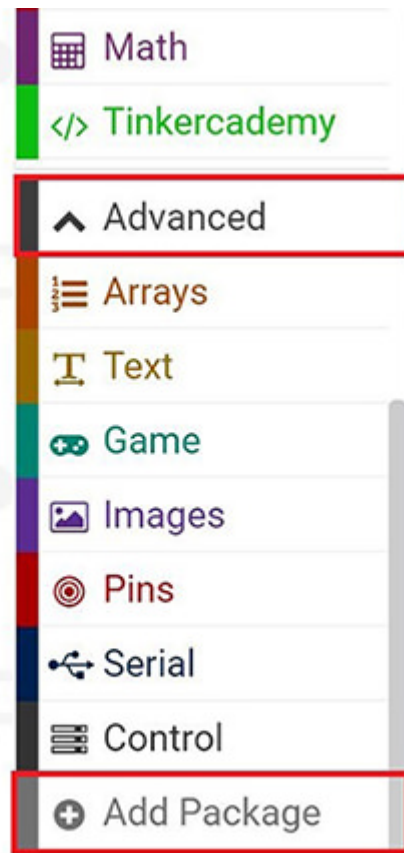
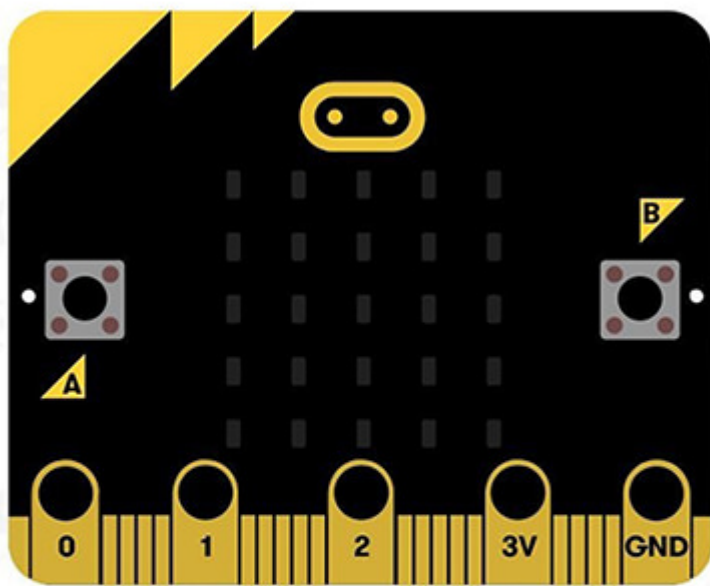
Step 2

Plug in the buzzer Brick to Pin 0. Plug in the ADKeypad to Pin 2. Make sure the colour of the wire of the buzzer and the ADKeypad follows the colour of the pins on the breakout board.

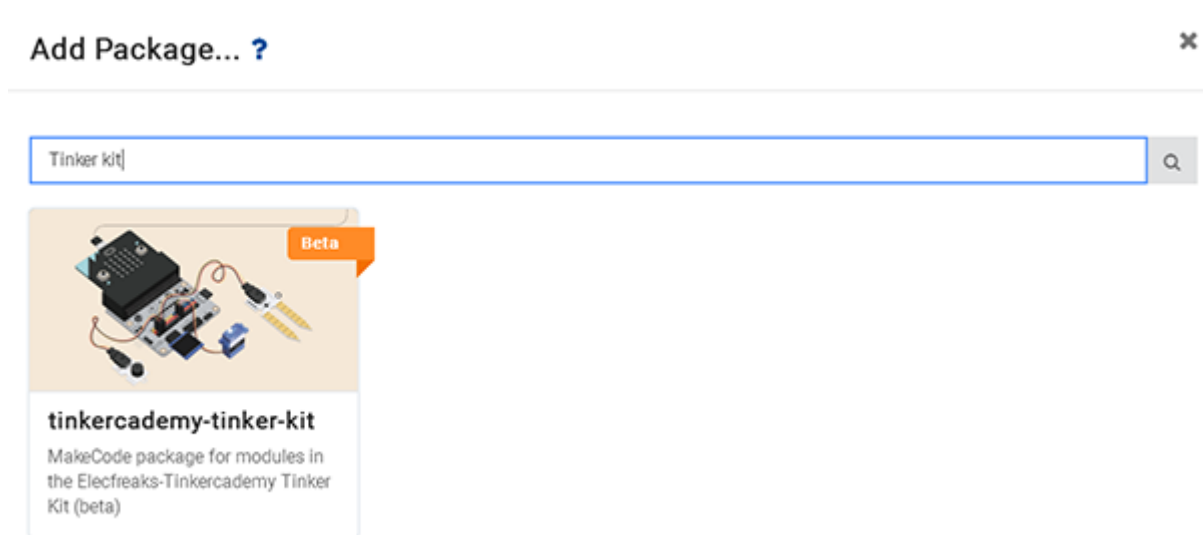


Step 3

Click on Advanced in the Code Drawer to see more code sections.



To code for our extra kit components (the ADKeypad and the buzzer), we have to add a package of code. Look at the bottom of the Code Drawer for “Add Package” and click it. This will open up a dialogue box. Search for “tinker kit” and click on it to download this package.



Note: If you get a warning telling you some packages will be removed because of incompatibility issues, either follow the prompts or create a new project in the Projects file menu.

Step 4

Next, let's create a conditional statement as shown in the picture. This 'if-then' block of code is under the code section “Logic” of the code drawer. The code shown below means that when button A is pressed on the ADKeypad while the ADKeypad is plugged in at pin P2 of the breakout board, the buzzer will play a sound of 175 hertz.

```
if key A is pressed on ADKeyboard at pin P2
then
  play tone 175 for 1 beat
```

Since there are 5 buttons, we need to code 5 similar conditional statements. Each button controls a sound of a particular pitch. So press each button, we will get sounds of different pitches.

```
forever
  show icon [grid icon]
  if key A is pressed on ADKeyboard at pin P2
  then
    play tone 175 for 1 beat
  if key B is pressed on ADKeyboard at pin P2
  then
    play tone 196 for 1 beat
  if key C is pressed on ADKeyboard at pin P2
  then
    play tone 220 for 1 beat
  if key D is pressed on ADKeyboard at pin P2
  then
    play tone 247 for 1 beat
  if key E is pressed on ADKeyboard at pin P2
  then
    play tone 262 for 1 beat
```

If you don't want to type these code by yourself, you can download it directly from the link below:

https://makecode.microbit.org/_3VaHYtgxqRb9

Or, you can download from the page below:

▶ Simulator 📄 Blocks JS JavaScript Edit

```
on start
  show icon [grid icon]

forever
  if key A is pressed on ADKeyl
  play tone Low F for 1 beat
  if key B is pressed on ADKeyl
```

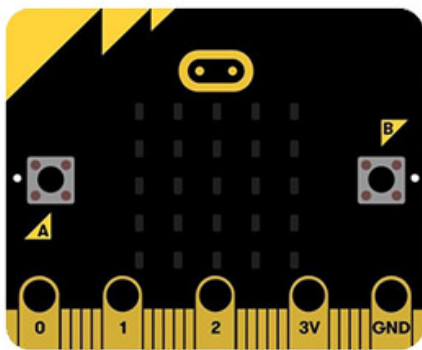



```
play tone ( Low G ) for ( 1 ▾ beat )
```

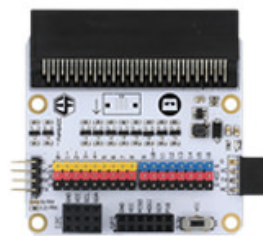
Success! Now you have your own Micro:bit Music Machine.

4. case 02 Smart Light

Make a smart Light



Micro:bit



Breakout Board



Octopus PIR Sensor

In this project, we are going to create a smart light with ElecFreaks Micro:bit Tinker Kit. It will use Octopus PIR Sensor and LED light. When there is someone in the room and detected, the LED will light up.

4.1. Goals

- Make something with Octopus PIR sensor.
- Make something which is easily to be used in real life.

4.2. Materials:

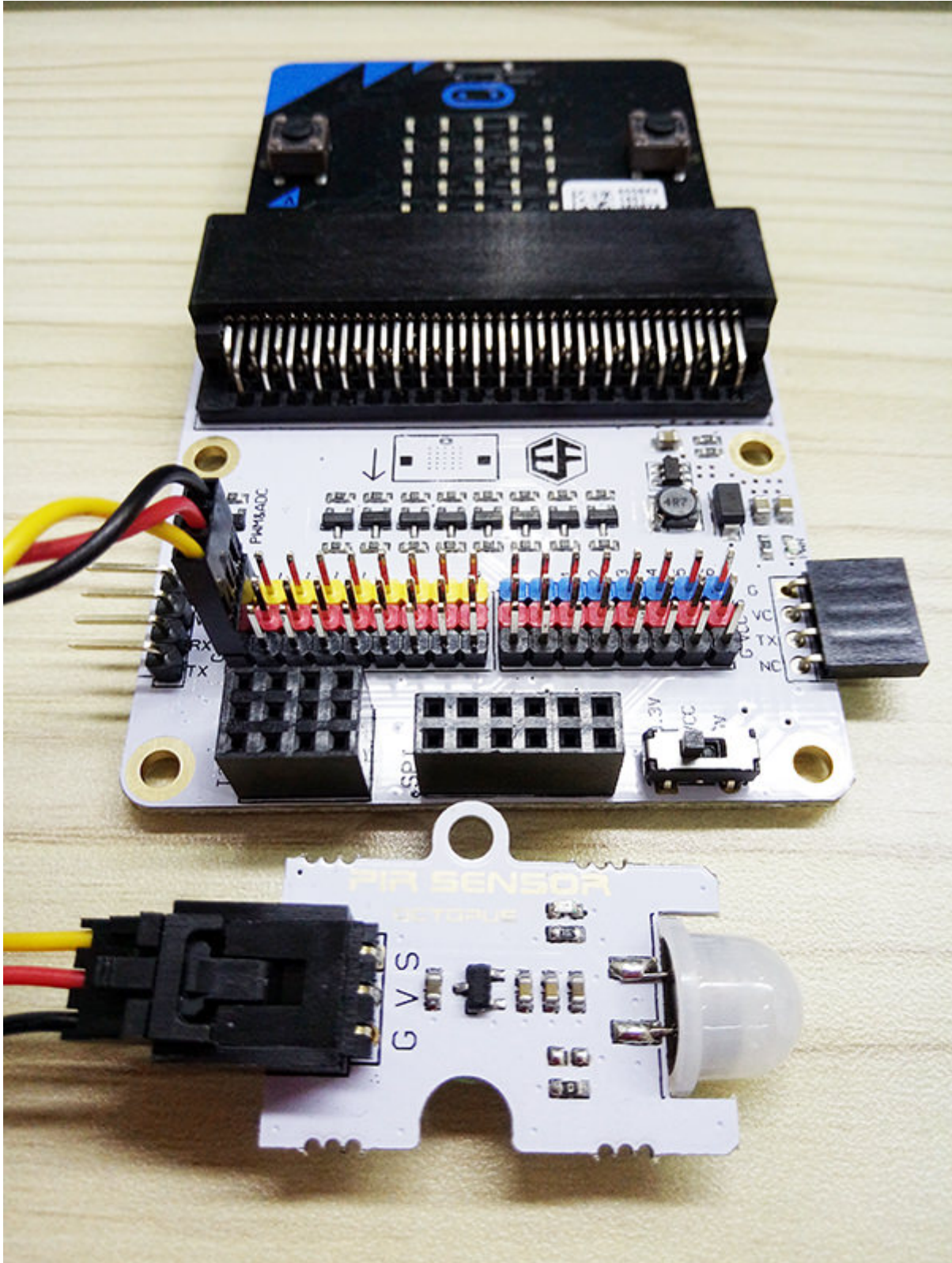
—

- 1 x BBC micro:bit
- 1 x Micro USB cable
- 1 x Breakout board
- 1 X Octopus PIR sensor Brick
- 1 x Octopus 5mm LED Brick OBLED - Red

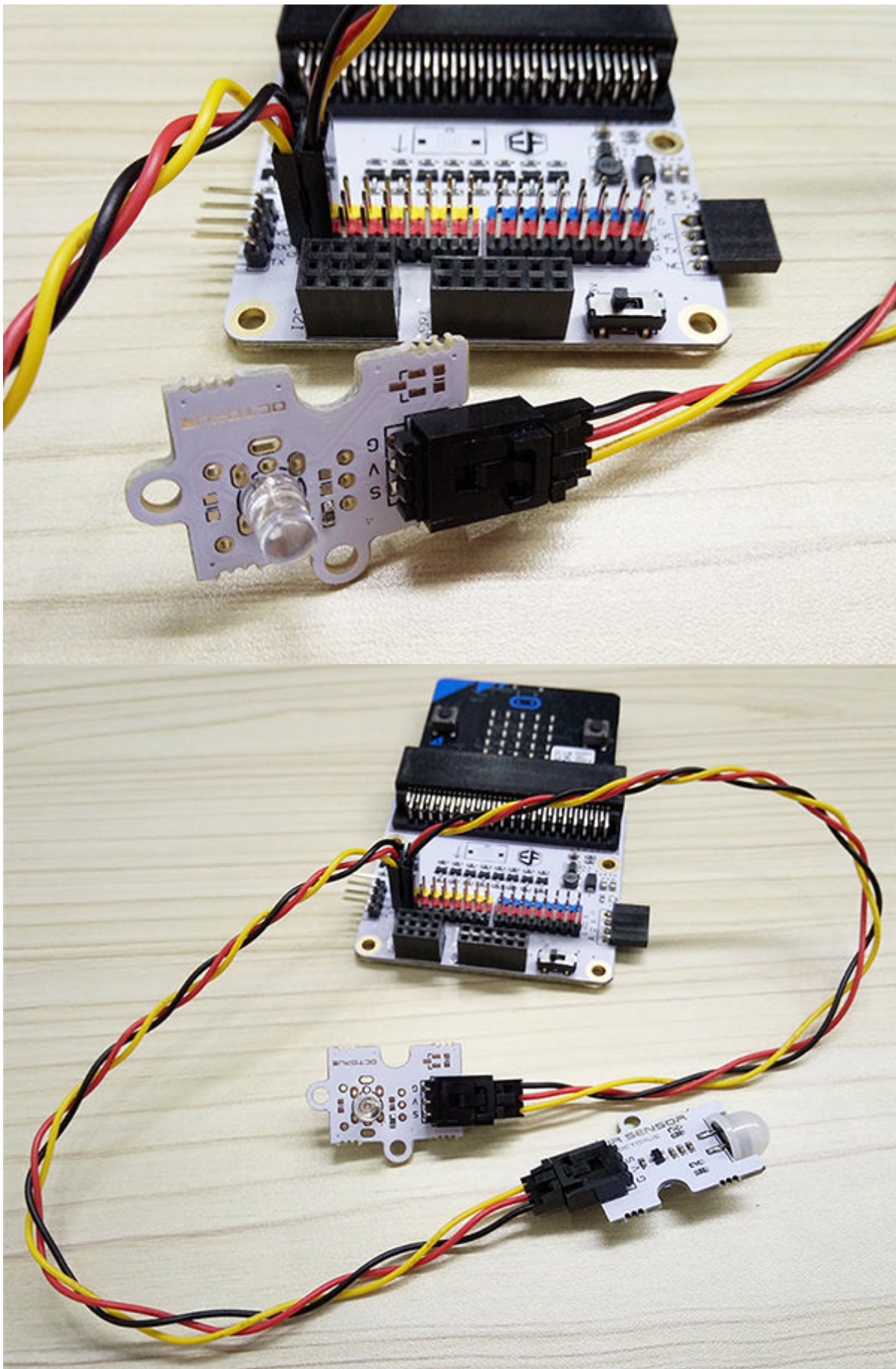
4.3. How to Make

Step 1: Components

Insert the micro:bit into the breakout board and plug Octopus PIR sensor into Pin 0.

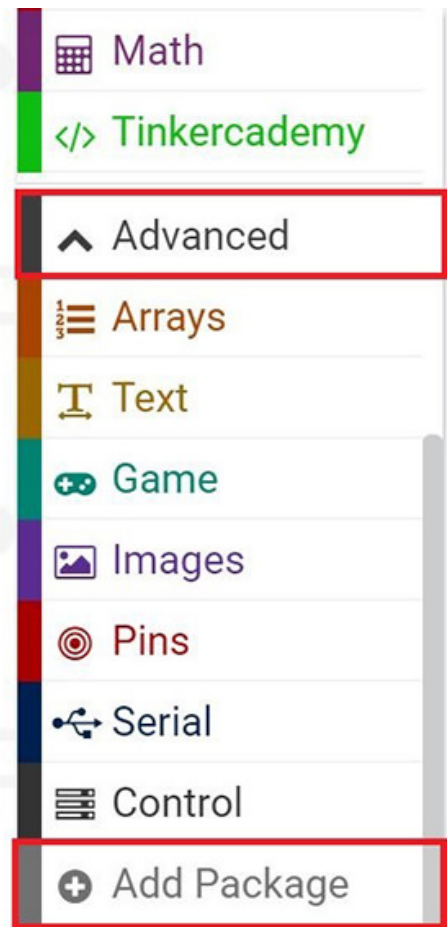
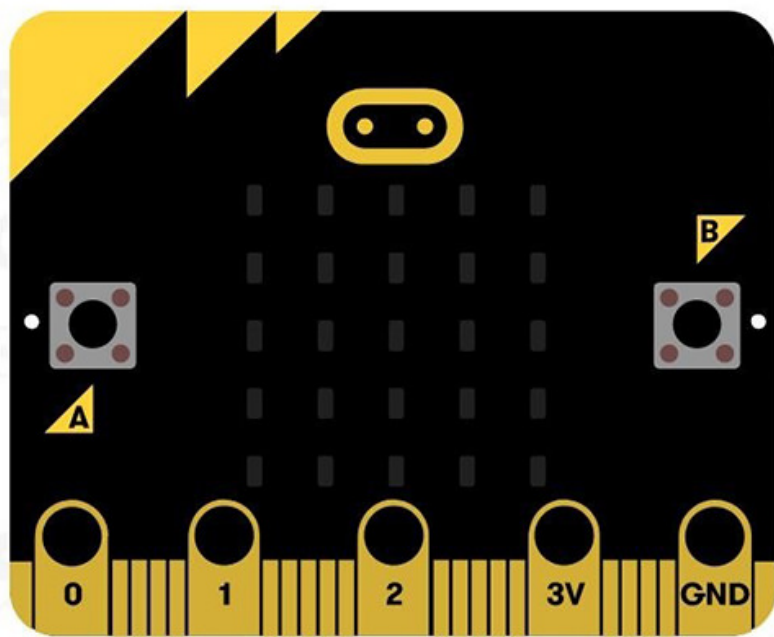


Plug LED to Pin 1. Make sure the wire colors matches pin colors.

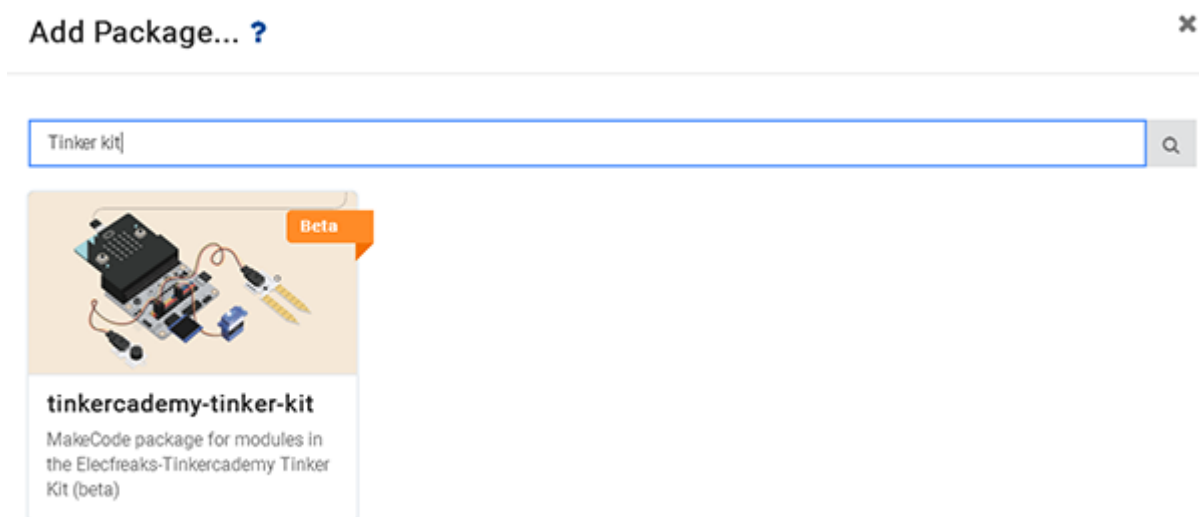


Step 2: Pre-coding

We will add a package of code to enable us to use our kit components. Click on “Advanced” in the Code Drawer to see more code section and look at the bottom of the Code Drawer for “Add Package”.

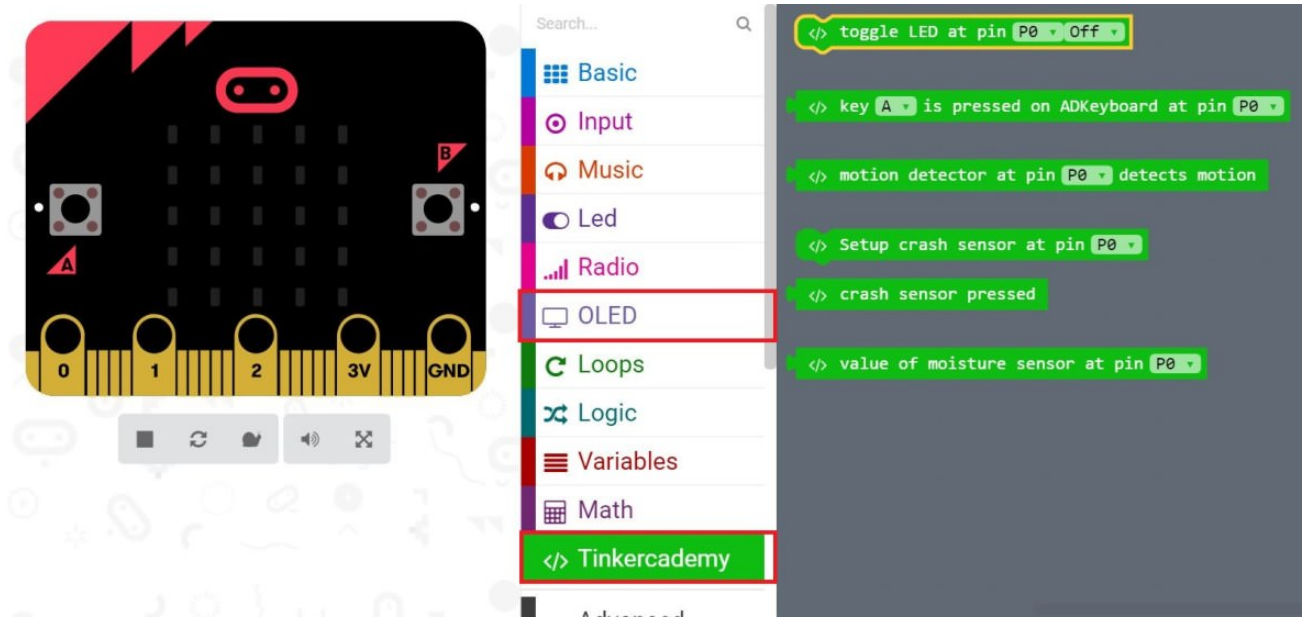


This will open a dialog box. Search for “tinker kit”, and then click it to download this package.



Note: If you get a warning telling you some packages will be removed because of incompatibility issues. You have to either follow the prompts or create a new project in the projects file menu.

Step 3: Coding



Click on Tinkercademy inside the Code Drawer to find blocks for the components in your kit.

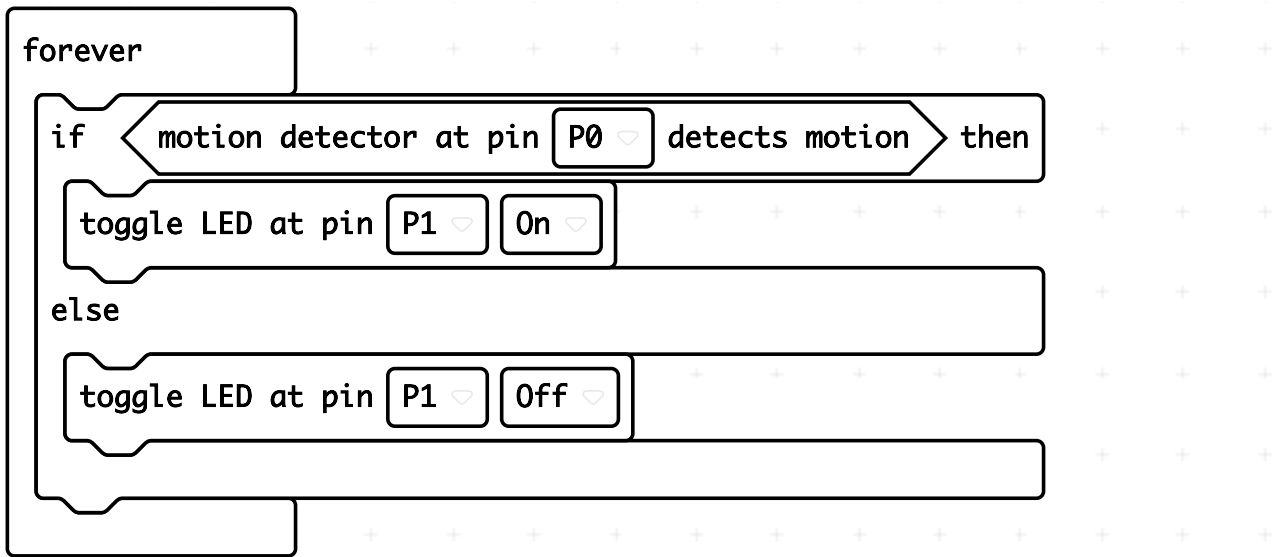


For this project, no initialization is required, and there isn't much code anyway. If you don't want to type these code, you can download it directly from the link below:

https://makecode.microbit.org/_aFUP7wcy94sv

Or, you can download from the page below:





[Microsoft MakeCode](#) | [Terms of Use](#) | [Privacy](#) | [Download](#)

If any motion is detected by the PIR sensor, the light is triggered. Or else, the light is turned off. Quite simple enough.

Step 4: Success

Voilà! You have created a simple smart light! Let's light it up !

5. case 03 Electro-Theremin



5.1. Goals

- Learn to use an analog sensor with the micro:bit.
- Make an electro-theremin!

5.2. Materials

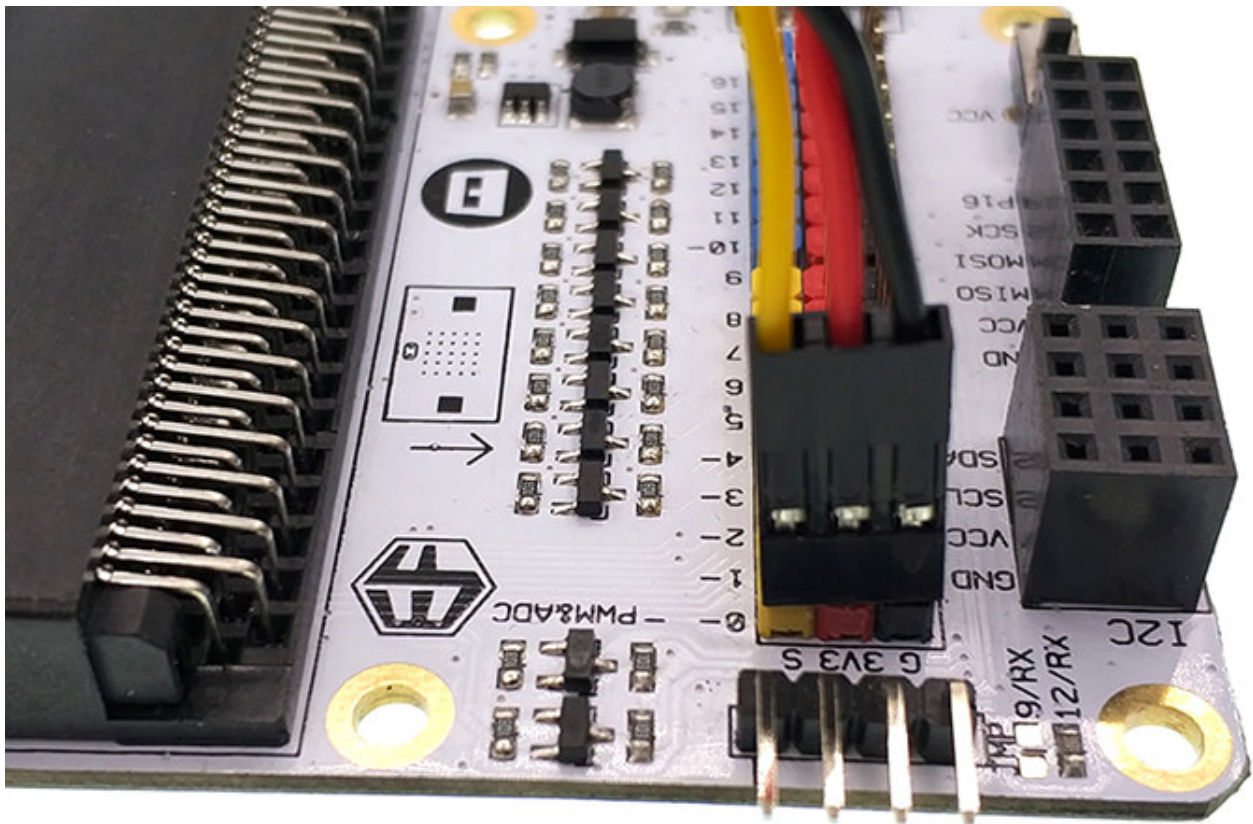
- 1 x BBC micro:bit
- 1 x Micro USB cable
- 1 x Buzzer
- 2 x F-F Jumper Wires
- 1 x Potentiometer

5.3. Procedure

Step 1

Plug in your Buzzer to Pin0. Make sure the positive lead is connected to the yellow signal pin and the negative lead is connected to the black ground pin on the breakout board. Plug in the potentiometer to Pin1. You can plug according to the color. Make sure that the

wire colors and the pin colors on breakout board are well matched!



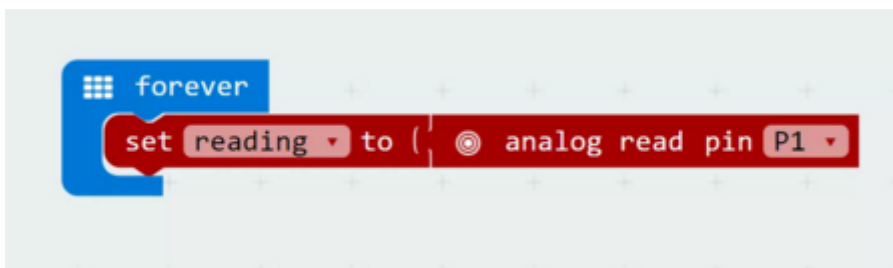
Step 2

In Makecode, we'll track the value of the potentiometer using a variable. Variables are like buckets that can hold changing values.

Make a new variable called reading (or anything you like, really) in the Variable drawer.

We want to constantly set our reading variable to the analog value of the potentiometer instead of the digital.

Reading the analog value allows us to access a whole range of signals from the potentiometer, instead of just a digital 1 or 0. Find this block in the Pins drawer.

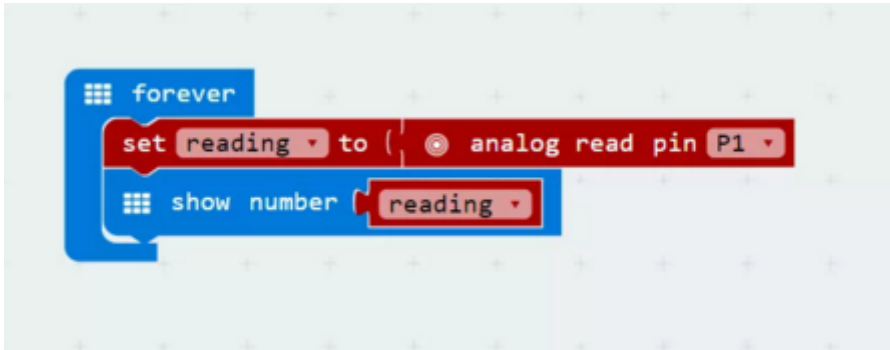


Step 3

Check your minimum and maximum values for your potentiometer by showing the number of the reading variable.

Turning the knob anti-clockwise all the way gives you the minimum, and clockwise all the way gives you maximum.

Notice how the values jump? That's because the micro:bit takes some time to scroll a large number across the screen, and by the time you read a new value, the potentiometer would be way ahead!



Step 4

Now we're going to use those values you just read from your potentiometer to map out your notes!

Our music blocks may not have a range as wide as your potentiometer. In this instance, we want to make sure the highest potentiometer value still corresponds to the highest note we can play.

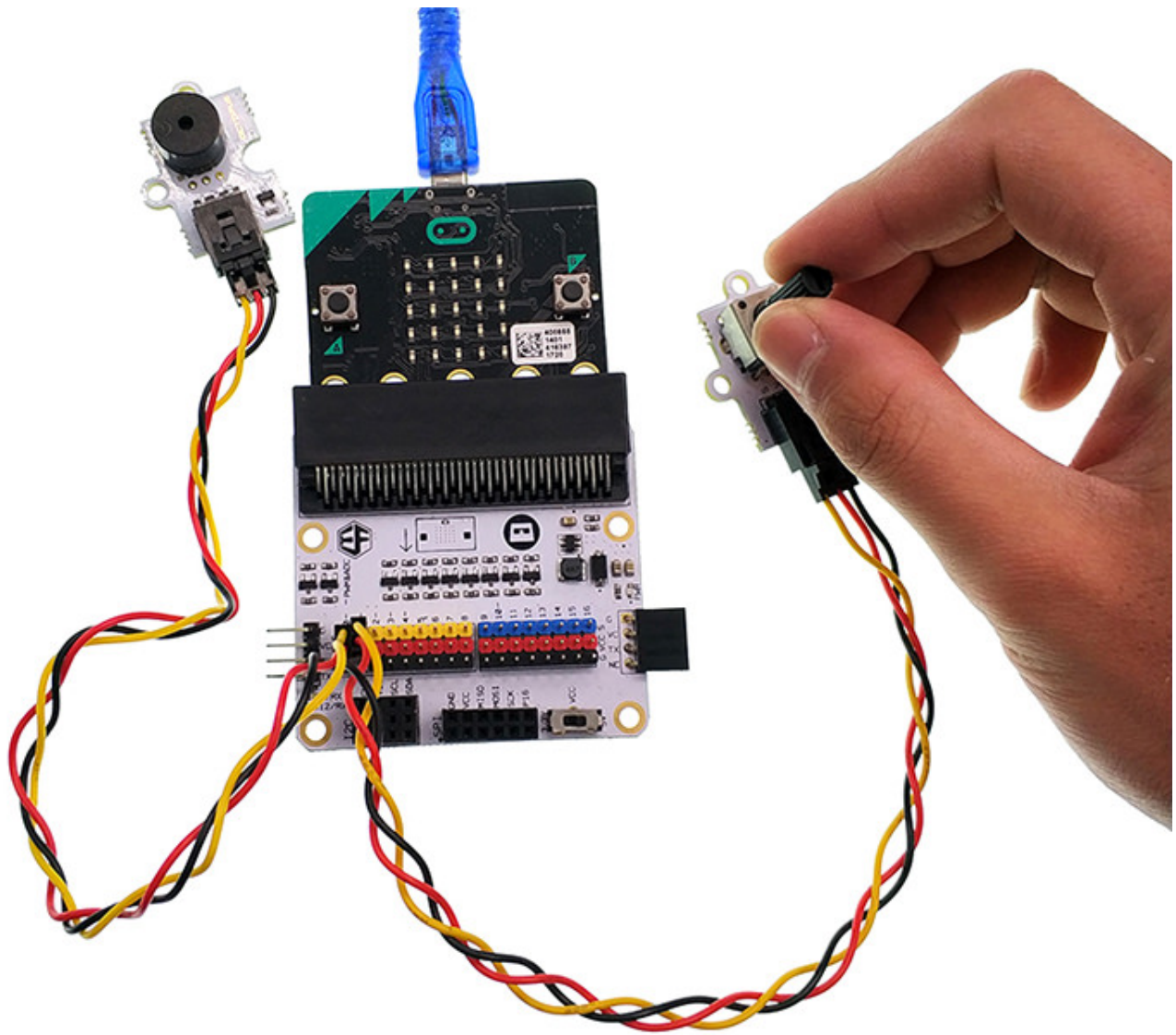
Check out the value of the lowest and highest notes in the micro:bit piano keys.

Using the map block from the Pins drawer to key in all the values.



Step 5

You may have noticed we made another variable called note in the previous step. Make sure you set the note variable to the mapped values. Ring the tone using the note variable. Save these code into your micro:bit and you are ready to make some noise!



If you don't want to type these code by yourself, you can directly download the whole program from the link below:

https://makecode.microbit.org/_5jUeetL6oKqi

Or, you can download from the page below:

▶ Simulator 🧩 Blocks JS JavaScript ▼ ↗ Edit

```
forever
  set reading to analog read pin P1
  show number reading
  set note to map reading from low 4 high 1023 to low 131 h
  ring tone (Hz) High B
```



[Microsoft MakeCode](#) | [Terms of Use](#) | [Privacy](#) | [Download](#)

Cool stuff!

Now you've learned how to play around with the potentiometer, you can try to use it to control LEDs, servos, and other components! And if you get your hands on another analog sensor, you'll know just how to use it!

6. case 04 Simple Alarm Box

6.1. Step 0: Pre-build Overview



In this project, we are going to create a simple alarm device which will alert the owner if someone has stolen his or her property. The red LED will blink when the crash sensor detects that the object has been taken away. Otherwise, the green LED will light up continuously. The OLED will display the status of the device.

6.2. Materials:

-
- 1 x BBC micro:bit

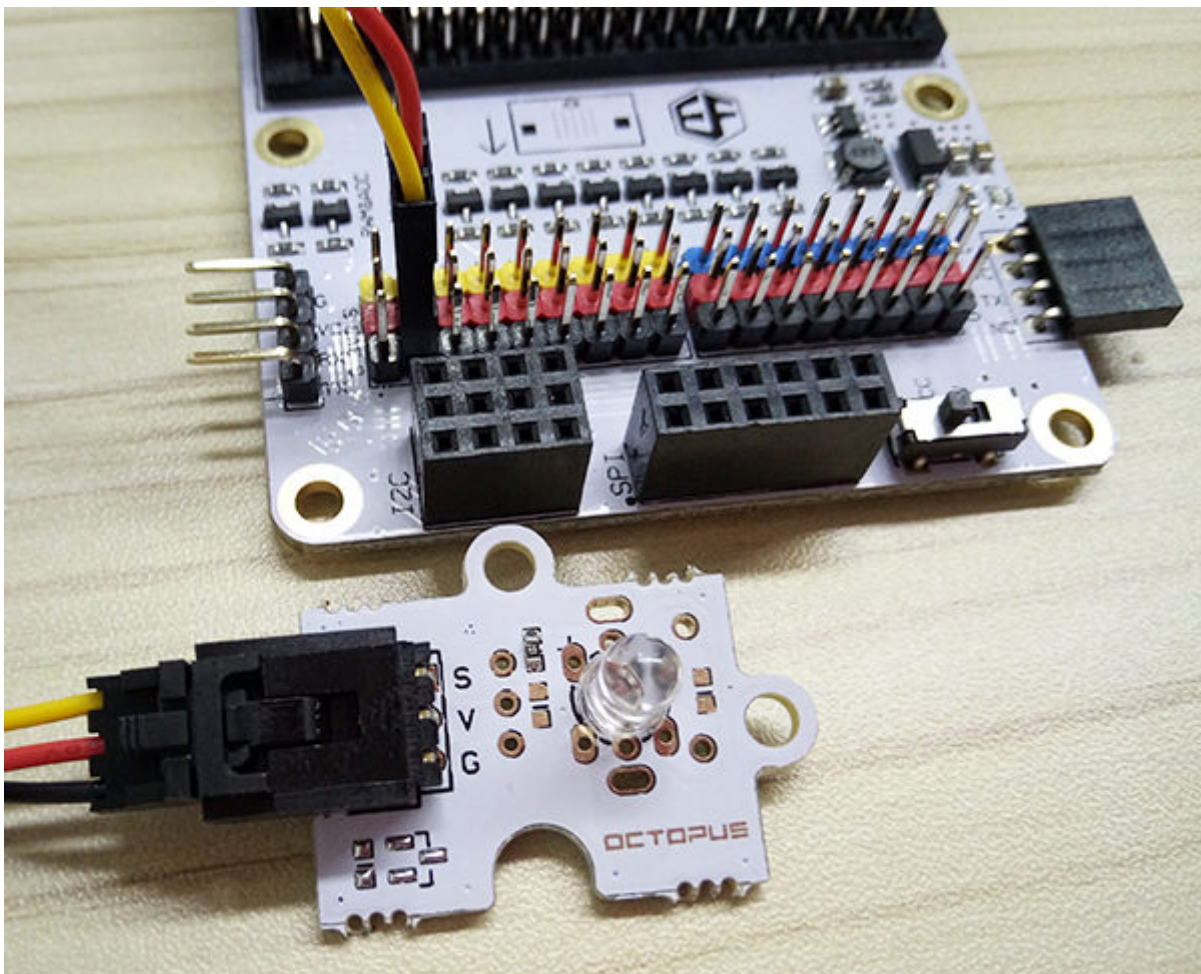
- 1 x Micro USB cable
- 1 x Breakout board
- 1 x Octopus LED
- 1 x Crash Sensor
- 1 x OLED
- 1 x LED
- 2 x Female-Female jumper wires

6.3. Goals:

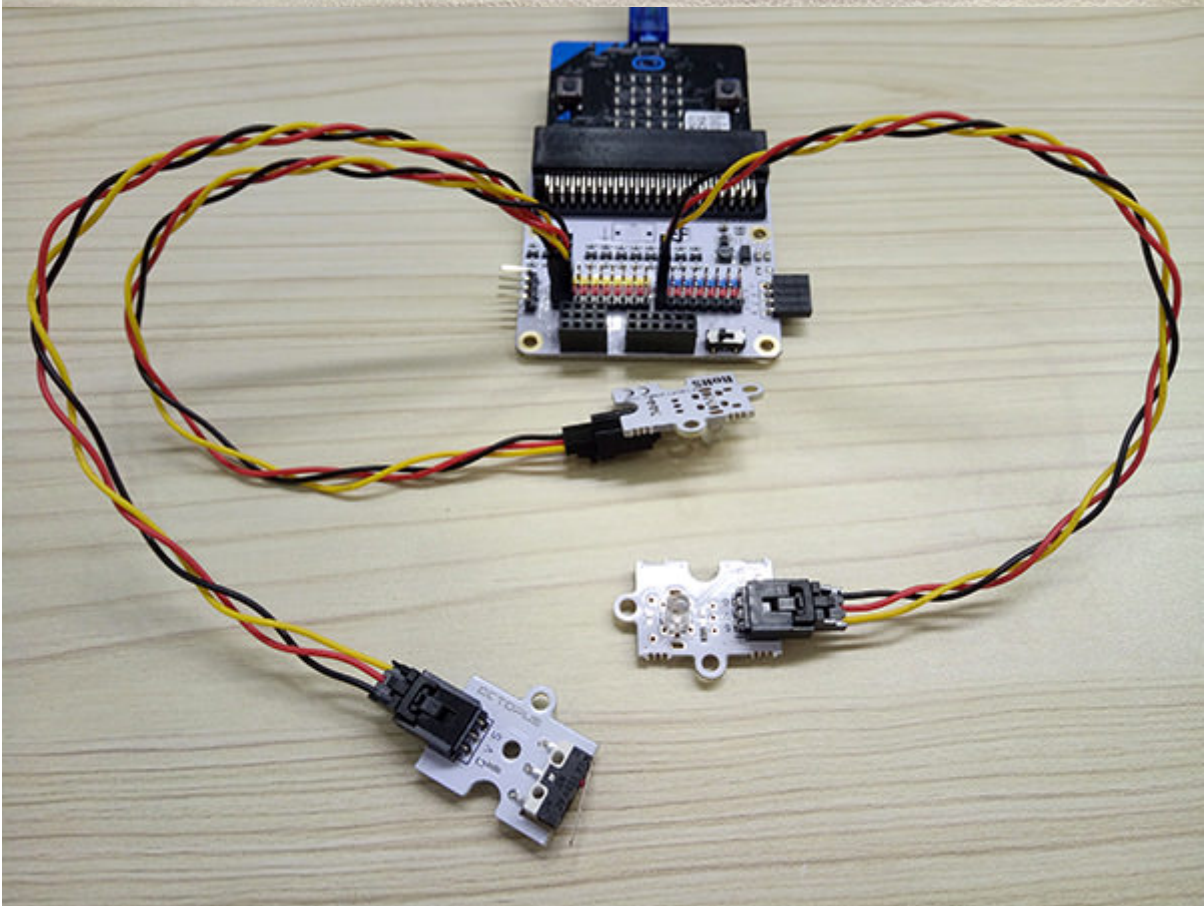
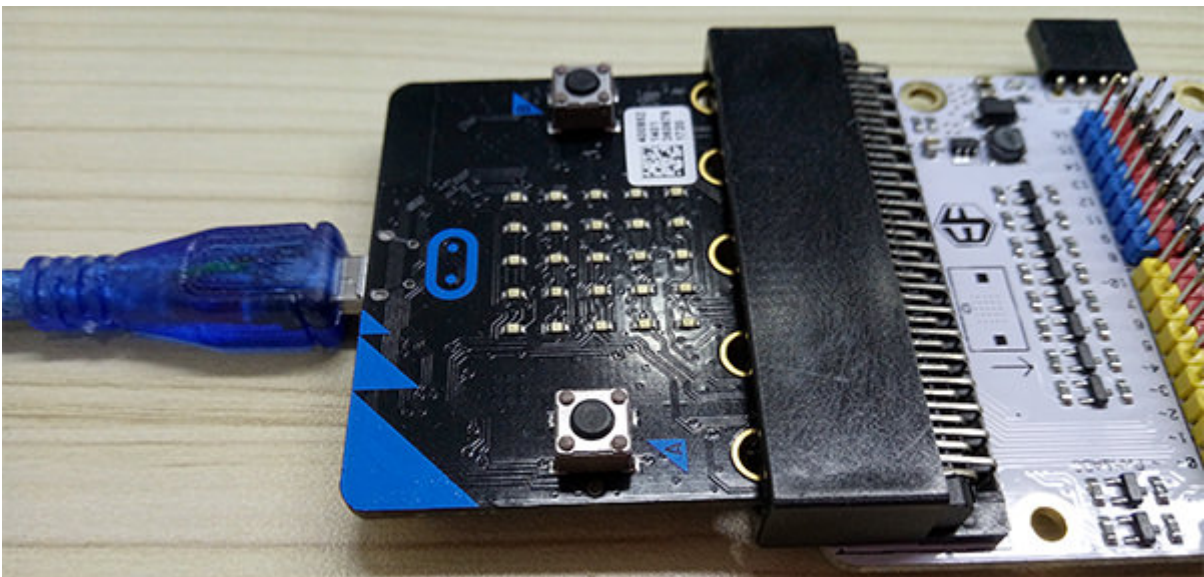
- Get to know the Octopus LED, normal LED, Crash sensor and OLED.
- Make something with different types of LED
- Make something with Crash sensor and OLED

6.4. How to Make

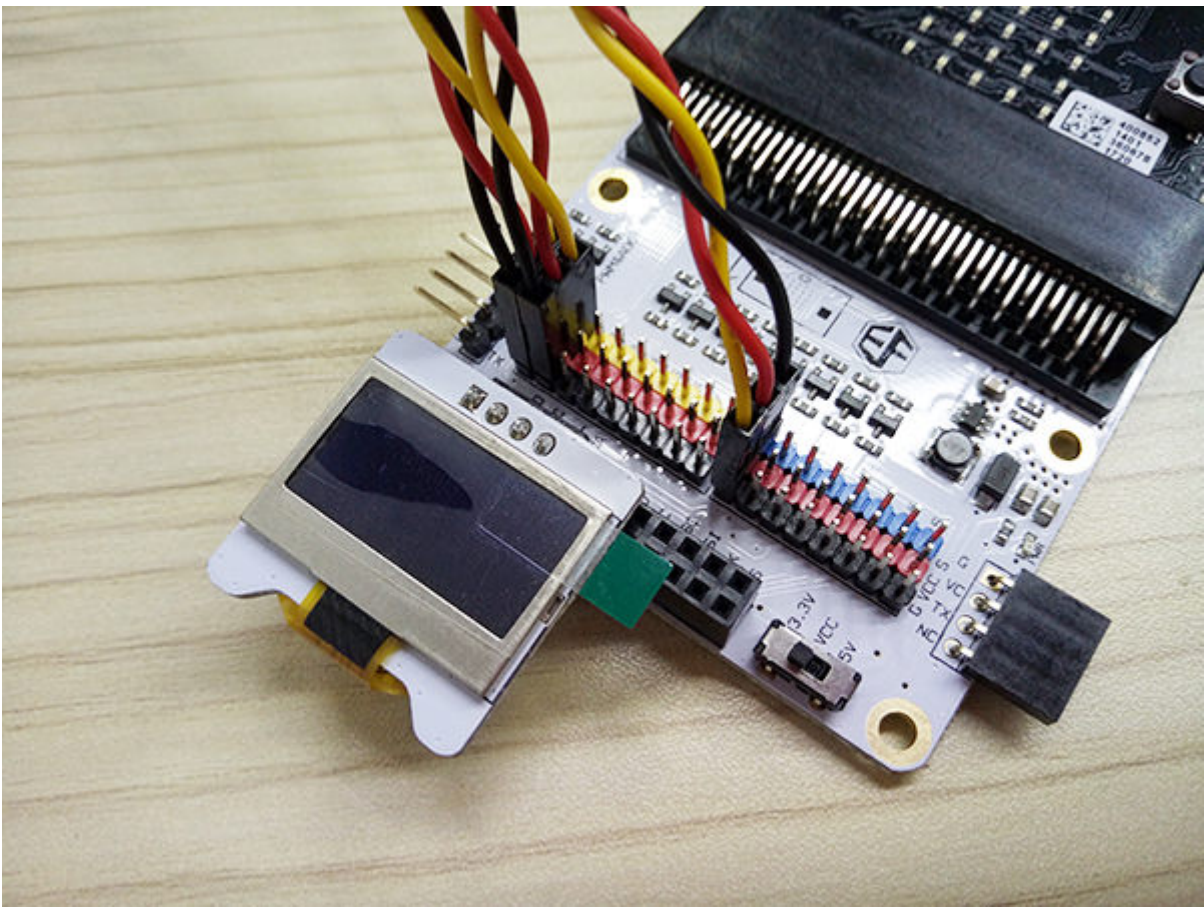
Step 1 – Components



Connect LED module to pin 1.

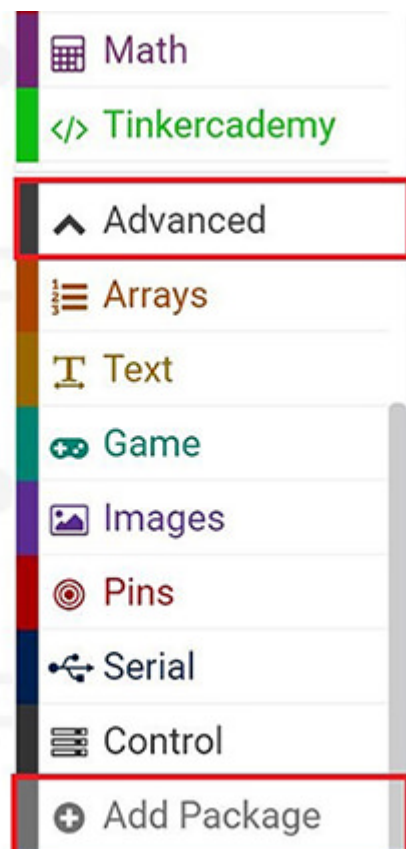
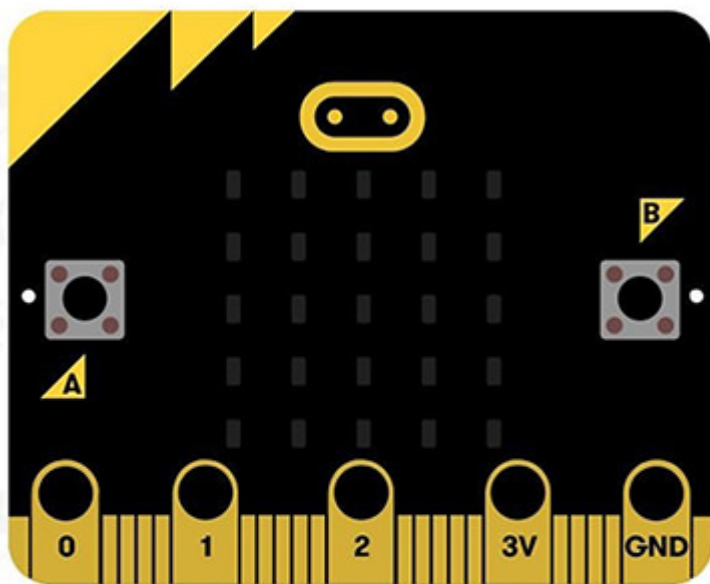


Connect the USB cable to the micro:bit and then to the breakout board as shown in the picture above. Making sure that the colour of the wire follows the colour of the pins on the break out board, plug in the crash sensor to Pin 0 and the Octopus LED to Pin 8. Lastly, plug in the OLED as shown in the picture above. You should be able to plug it into any of the three rows.

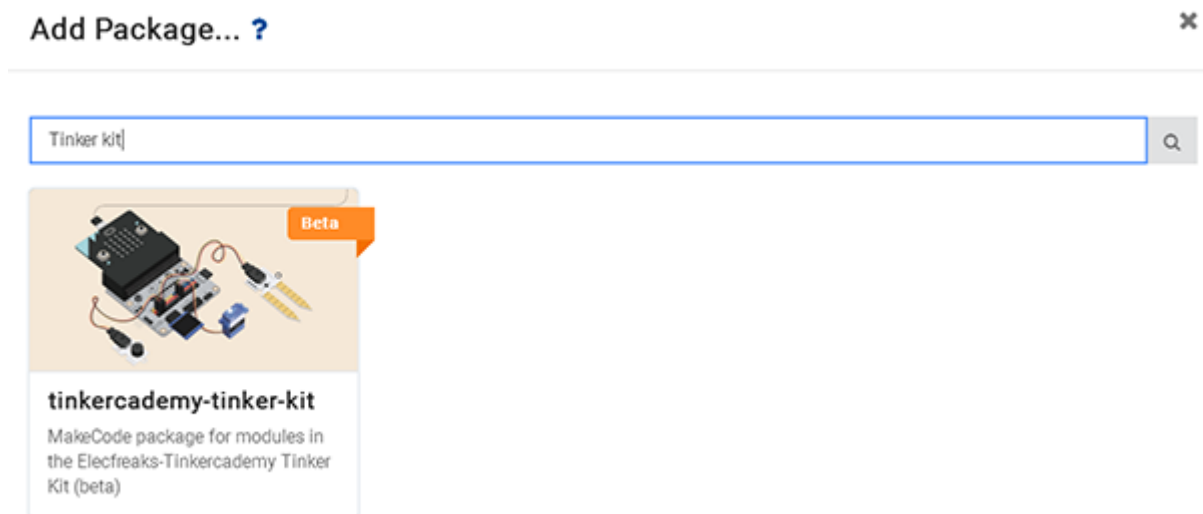


Step 2 – Pre-coding

We'll need to add a package of code to be able to use our kit components. Click on Advanced in the Code drawer to see more code sections and look at the bottom of the Code Drawer for Add Package.

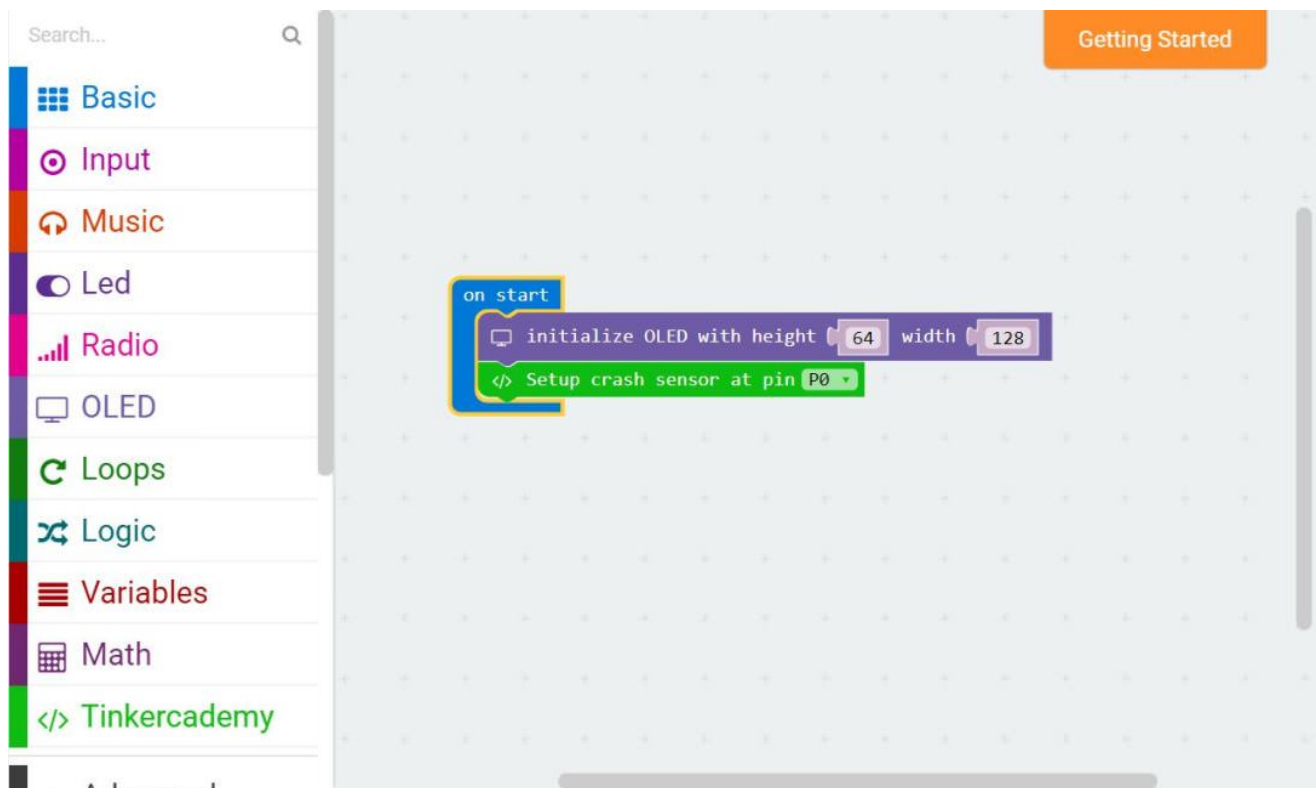


This will open up a dialog box. Search for “tinker kit” and then click it to download this package.

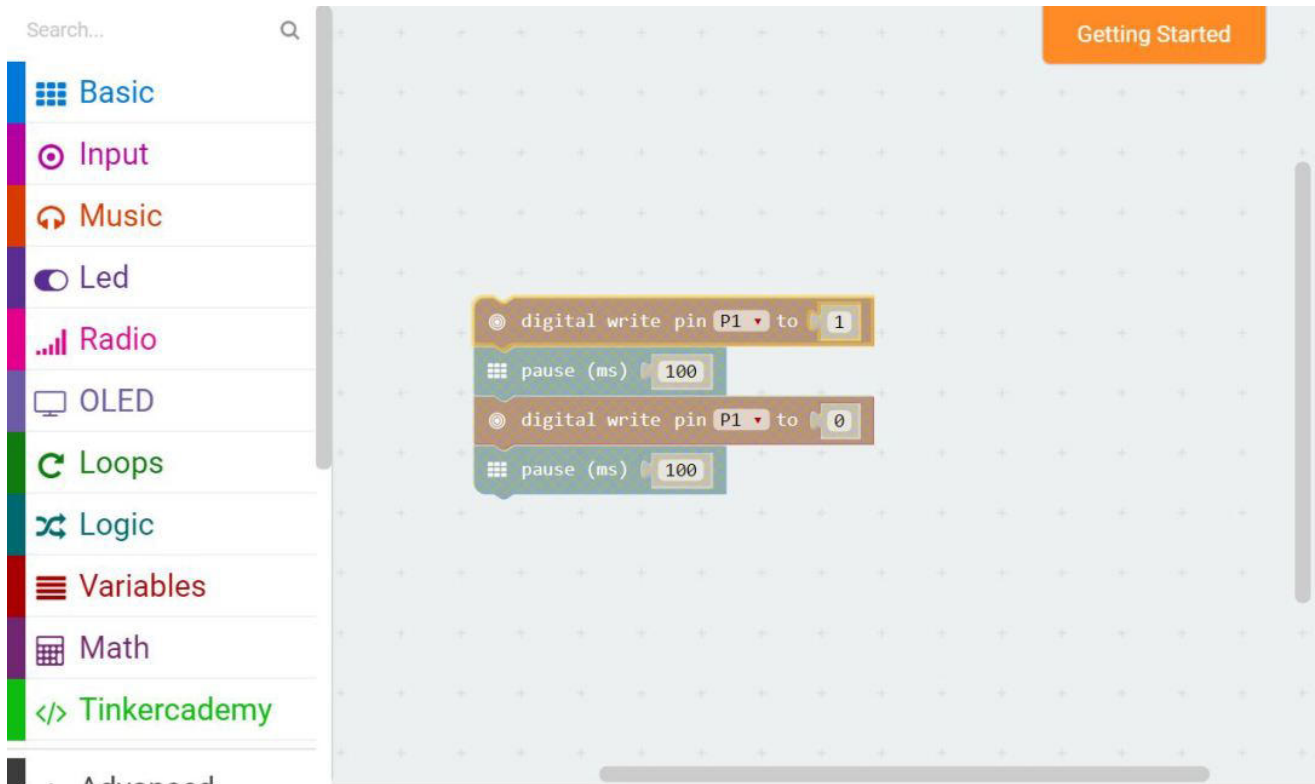


Note: If you get a warning telling you some packages will be removed because of incompatibility issues, either follow the prompts or create a new project in the Projects file menu.

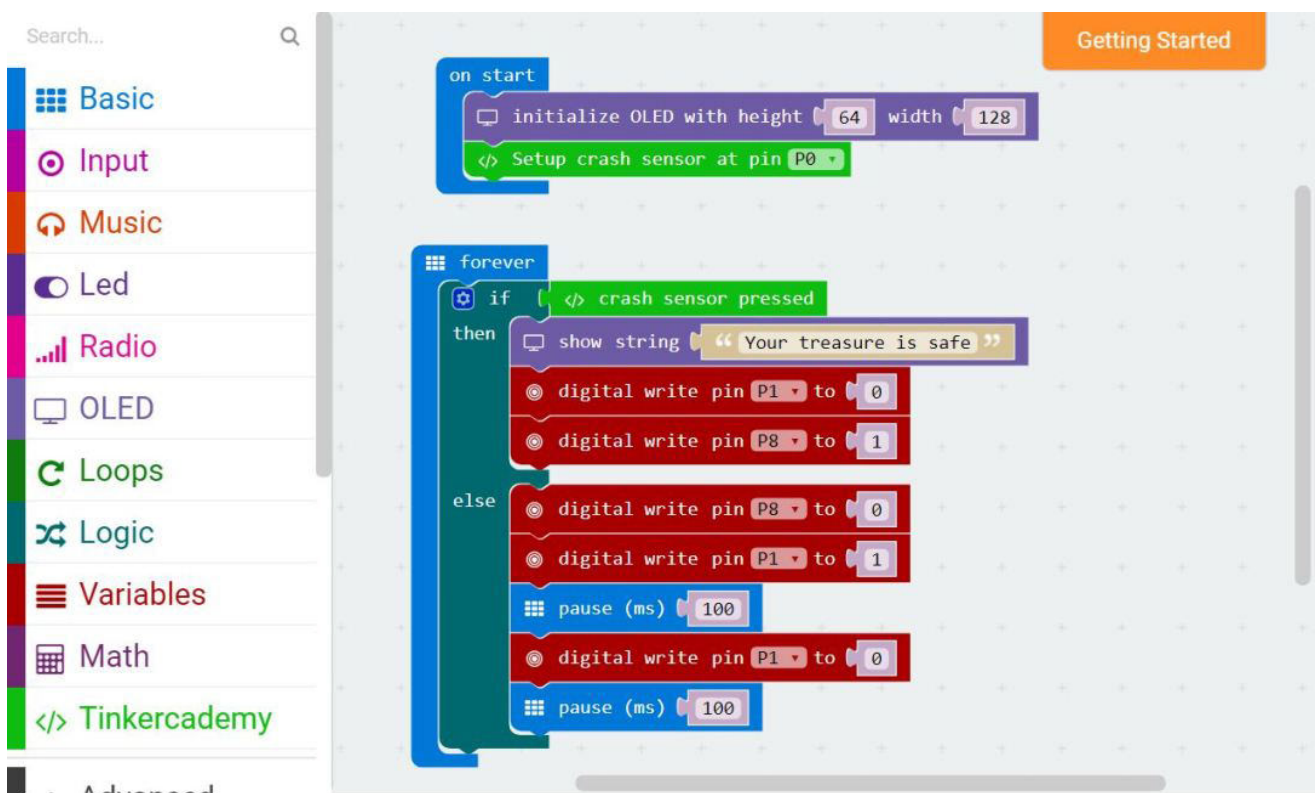
Step 3 – Coding



After that, use blocks under the Tinkercademy section to initialize the OLED and Crash Sensor as shown in the picture



This part of the code allows the red LED to blink continuously. You can adjust the speed of blinking by changing the pause period.



Since there are only two conditions, we need only one 'else-if' statement. When the Crash Sensor is pressed, the green Octopus LED will light up. Or else, if no force is applied to the Crash Sensor, the red LED will blink continuously.

If you don't want to type these code by yourself, you can directly download the whole program from the link below:

Or, you can download from the page below:

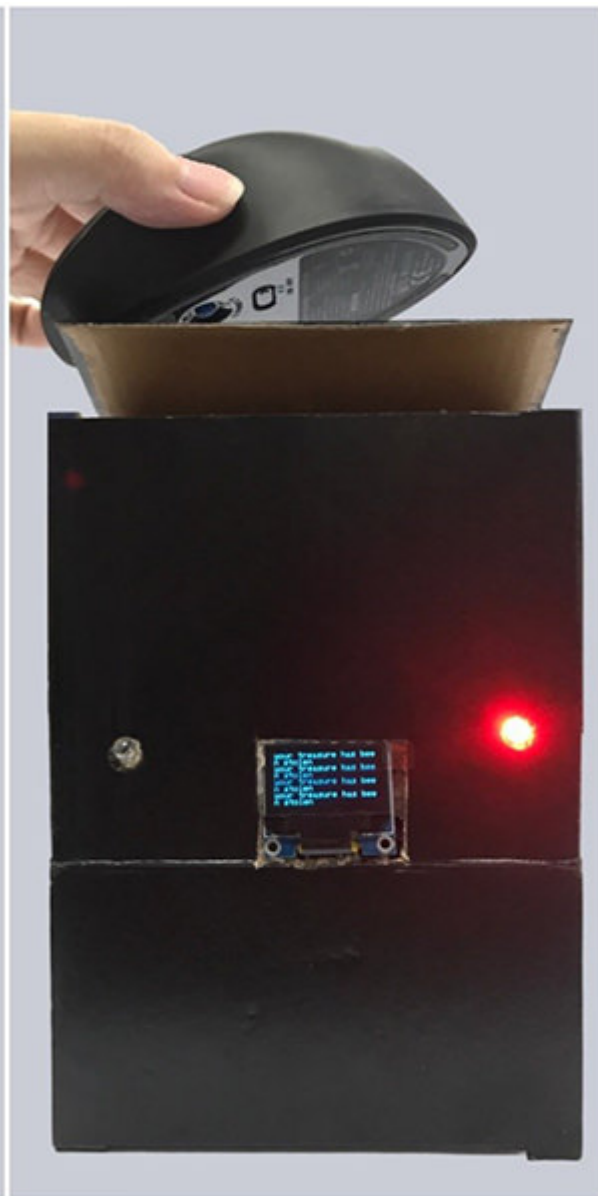
```
▶ Simulator  Blocks  JS JavaScript  Edit
2 tinkercademy.crashSensorSetup(DigitalPin.P0)
3 basic.forever(() => {
4     if (tinkercademy.crashSensor()) {
5         OLED.showString("Your treasure is safe")
6         pins.digitalWritePin(DigitalPin.P1, 0)
7         pins.digitalWritePin(DigitalPin.P8, 1)
8     } else {
9         pins.digitalWritePin(DigitalPin.P8, 0)
10        pins.digitalWritePin(DigitalPin.P1, 1)
11        basic.pause(100)
12        pins.digitalWritePin(DigitalPin.P1, 0)
13        basic.pause(100)
14    }
15 })
```

Problems 1

Microsoft MakeCode | Terms of Use | Privacy | Download

Step 4 – Succeed!

Let's download code into it and run it. Let's find a book or something else and place it on the top of device, then see what will happen. We can see the green light illuminates as showed in the picture below. When we take away the book or something else you placed, you can see the red light starts to flash while the green light turned off.



7. case 05 Plant Monitoring Device



In this article, we are going to talk about how to use micro:bit with buzzer, OLED and moisture sensor to build a case of plant detection device.

7.1. Goals:

- Get to know the buzzer, OLED and moisture sensor.
- Make something with a moisture sensor.

7.2. Material Needed:

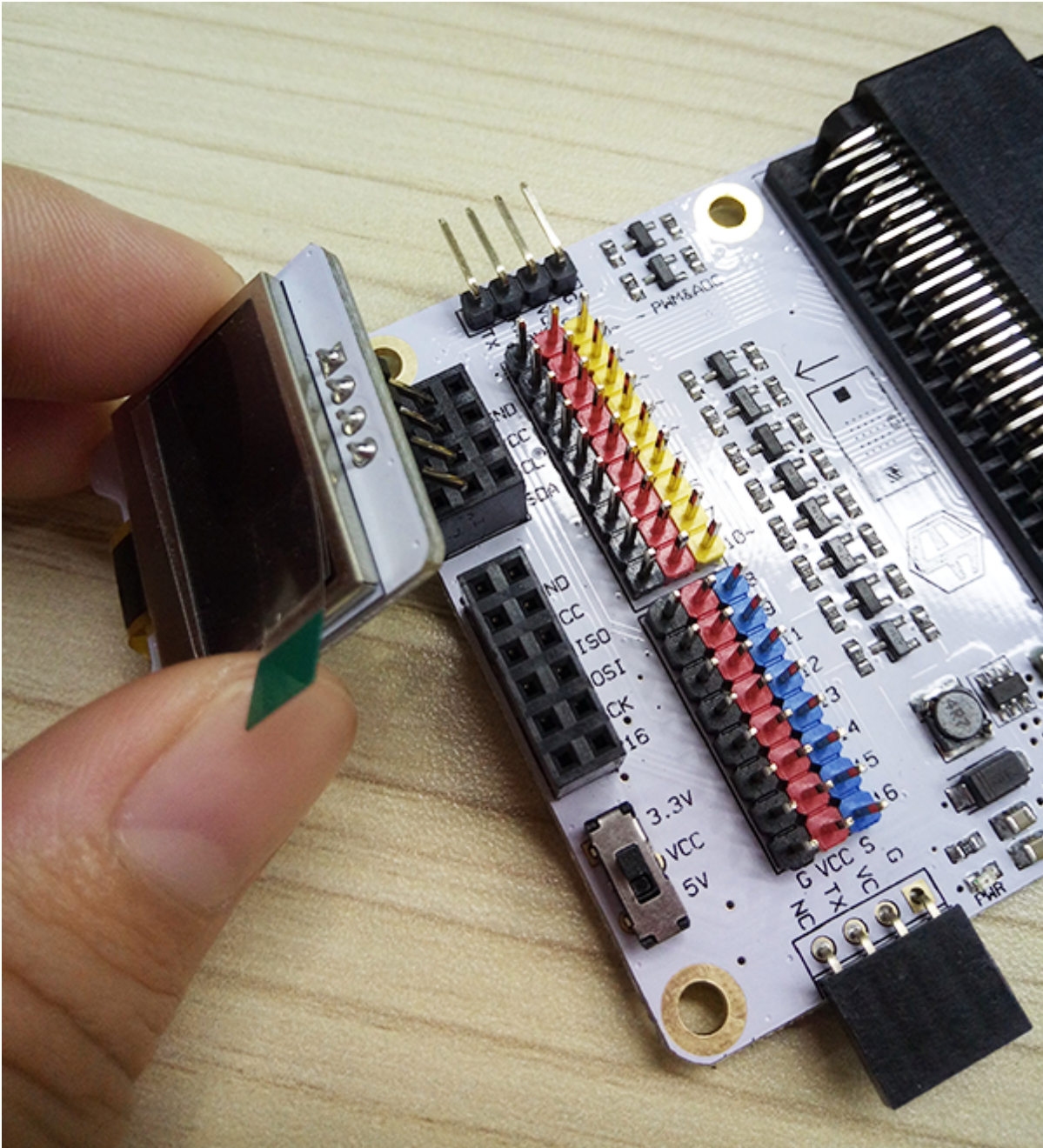
- 1 x BBC Micro:bit
- 1 x Micro USB Cable
- 1 x Breakout Board
- 1 x Mini Buzzer
- 1 x OLED
- 1 x Moisture Sensor
- 2 x Female-Female Jumper Wires

Note: You can plug components in any sequence.

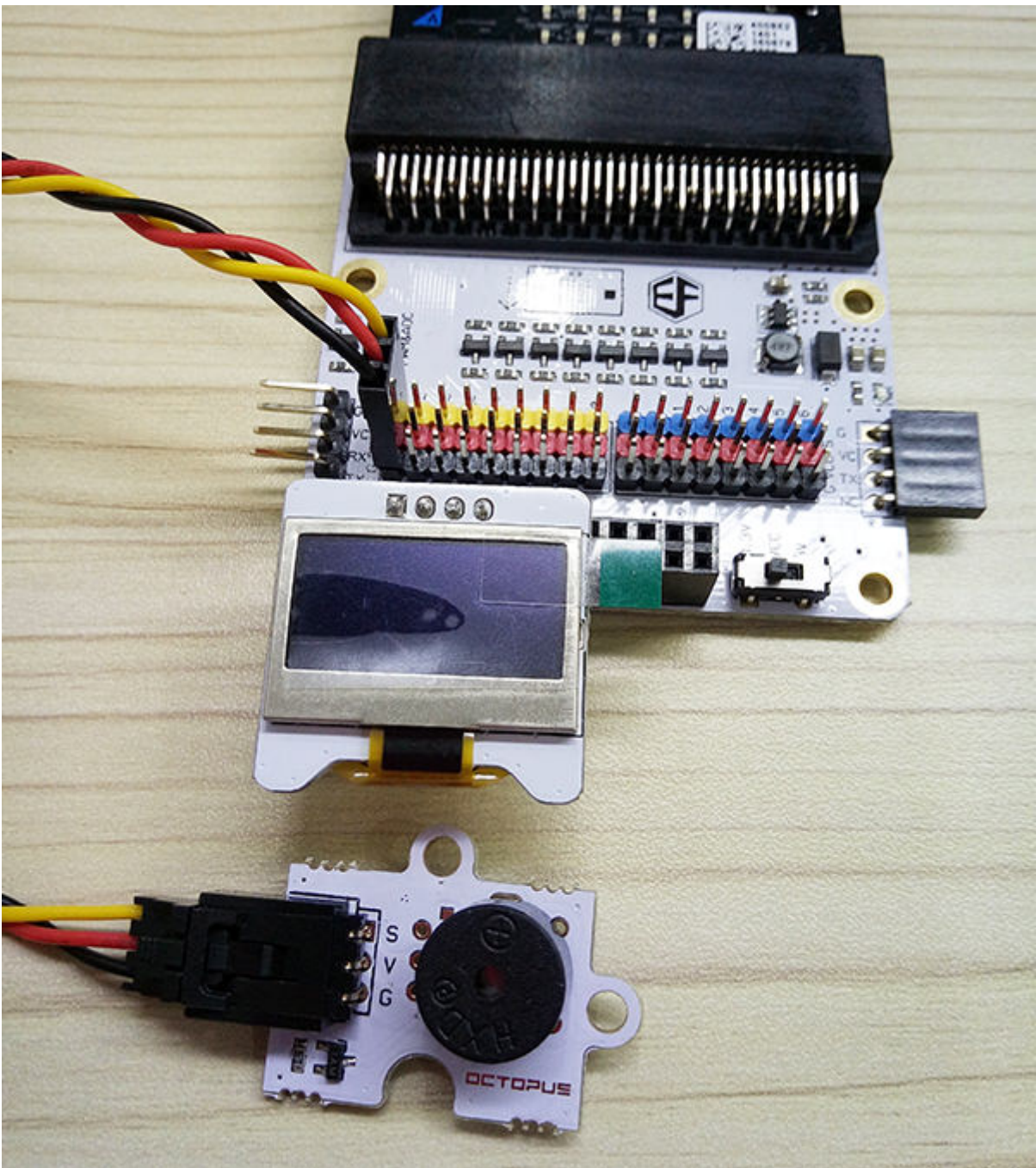
In this project, we are going to create a plant monitoring which the buzzer will sound when there is not enough water.

A message will always be displaying on the OLED, showing the moisture level.

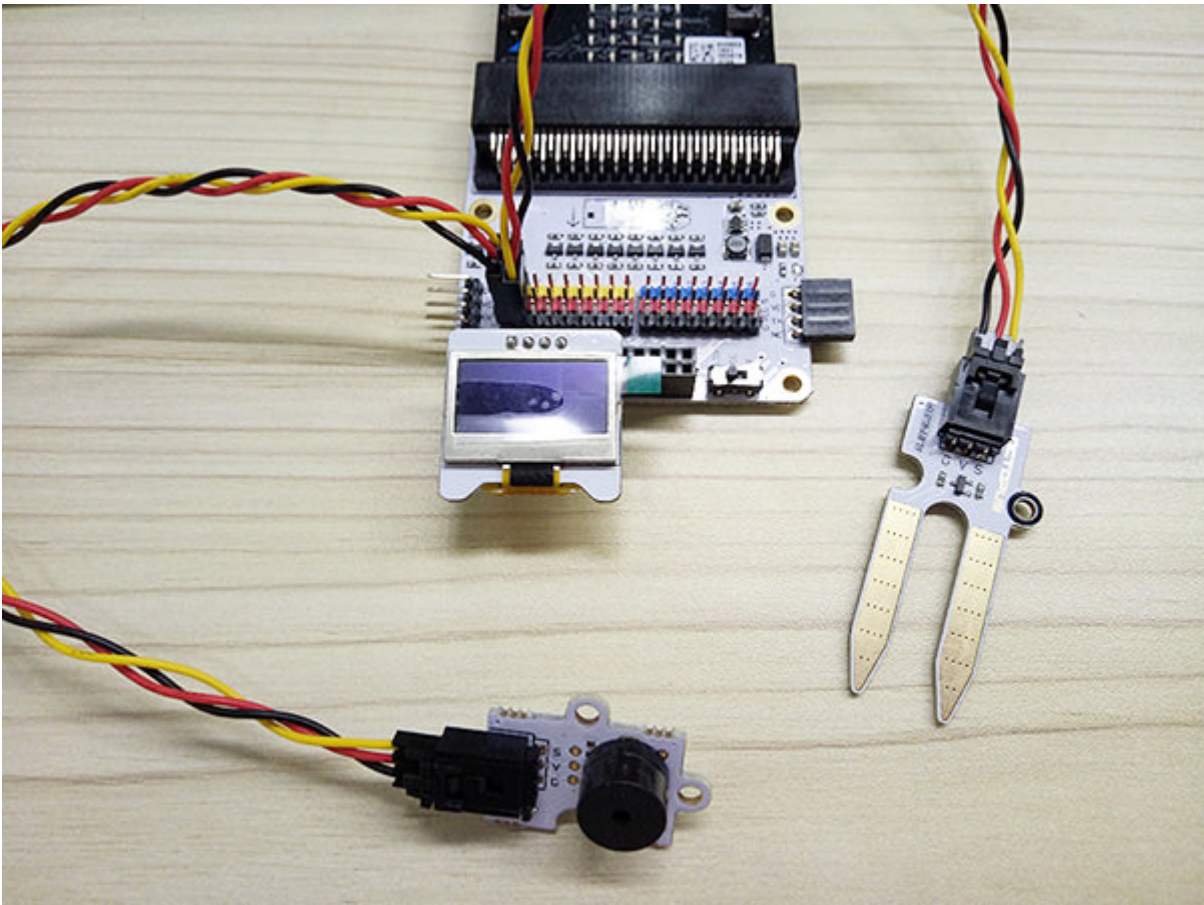
Firstly, plug in the OLED. You are able to plug it into any of the three rows.



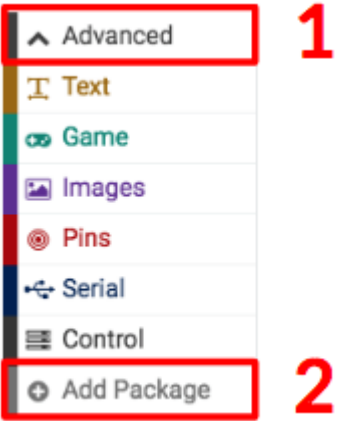
Connect buzzer to P0. Make sure the color of wire follows the pin color on breakout board.



Plug in the moisture sensor to P1.



Click on Advanced in the Code Drawer to see more code sections. We'll add a package of code to be able to use our kit components. Look at the bottom of the Code Drawer for "Add Package" and click it.

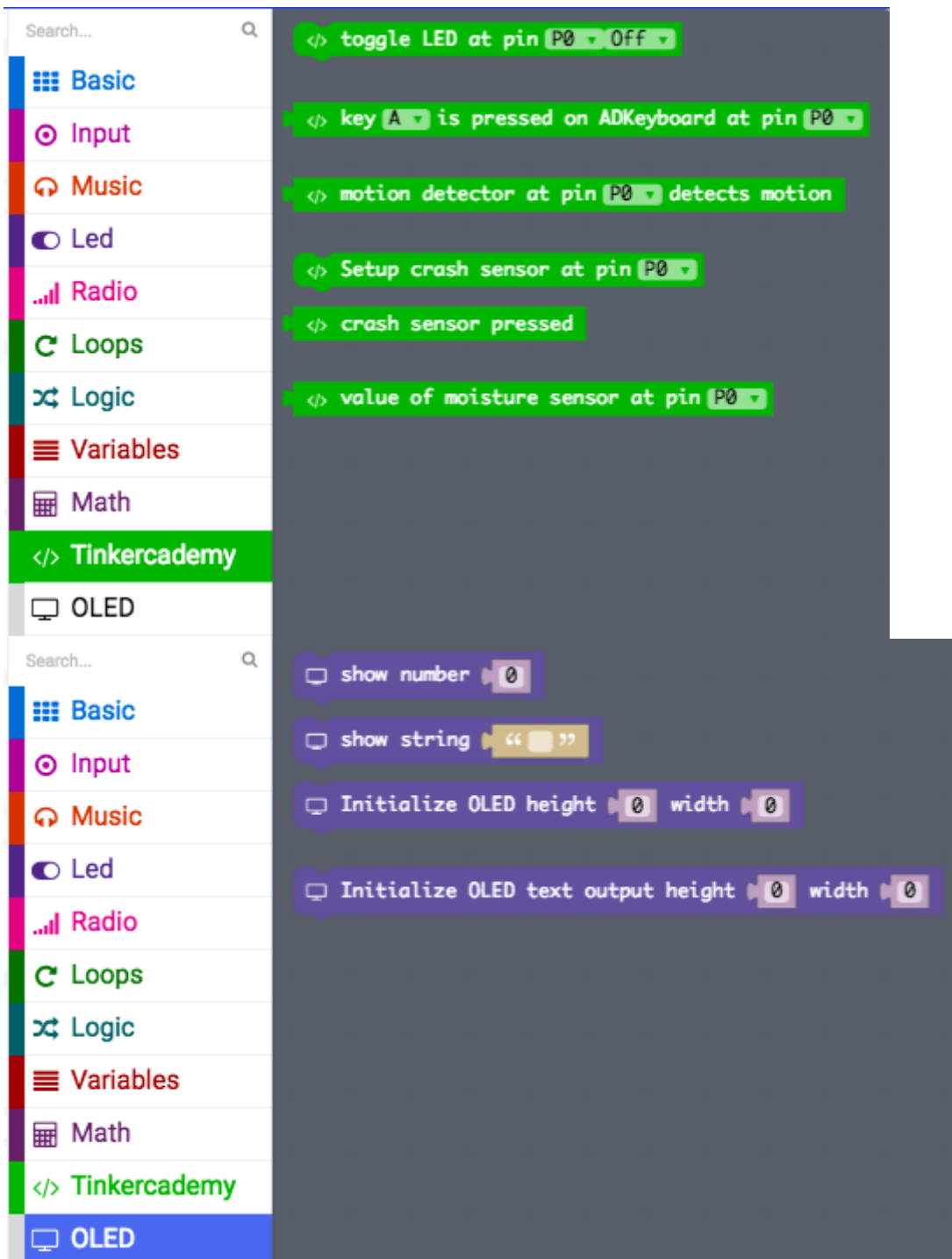


At this time, a dialogue box appears. Search "tinker kit" in the box and then click on the "tinkercademy-tinker-kit" for downloading this package.

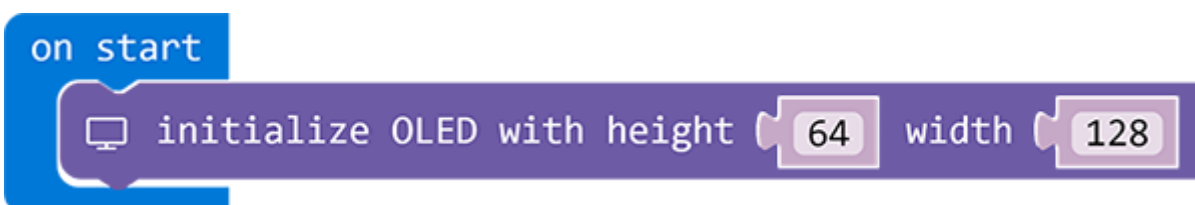
Q

tinkercademy-tinker-kit
MakeCode package for modules in the ElecFreaks-Tinkercademy Tinker Kit (beta)

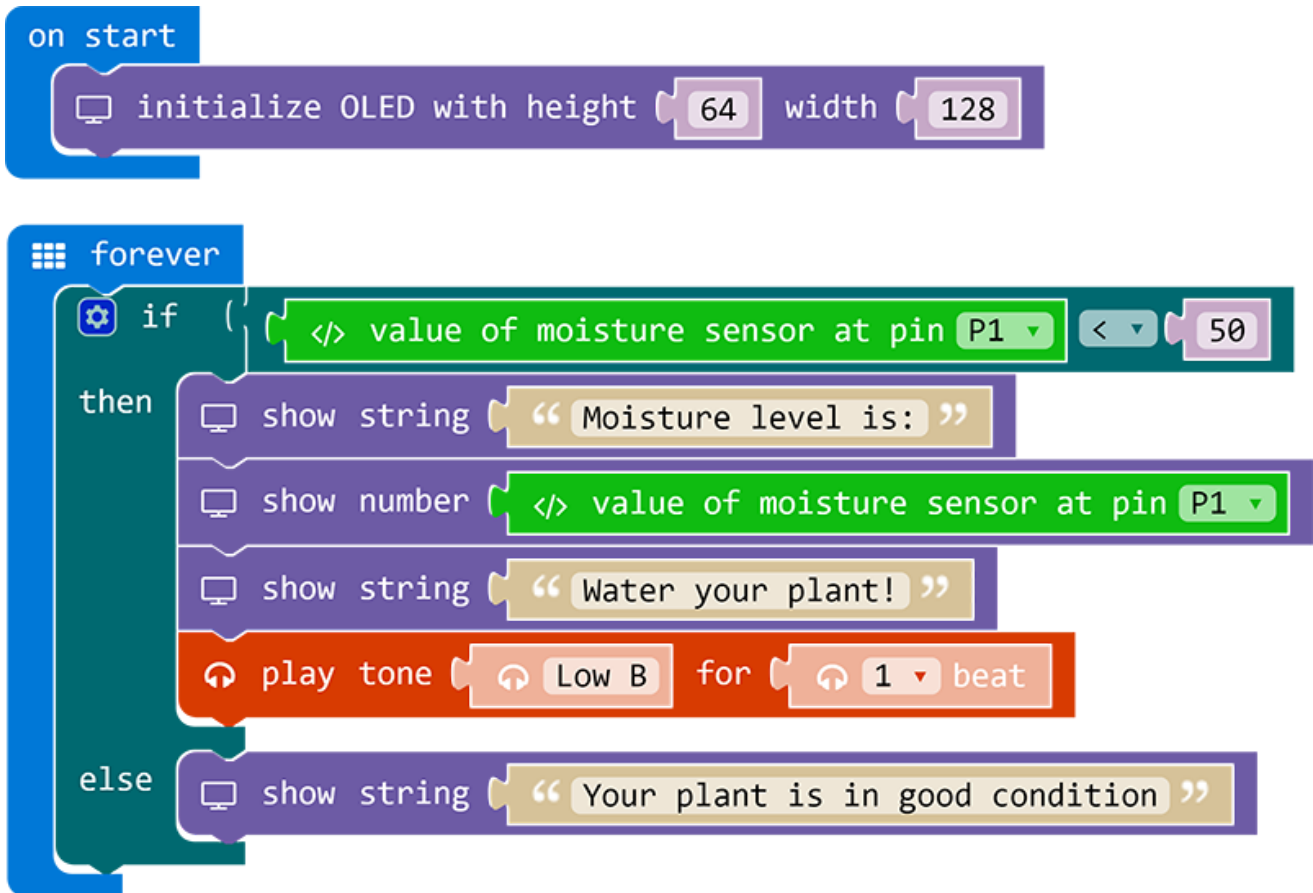
Click on Tinkercademy inside the Code Drawer to find our custom blocks for the various components in your kit.



After that, use blocks under the Tinkercademy section to initialize the OLED.



Since there are only two conditions, we need only one “else-if” statement. Micro:bit reads values from moisture sensor continuously. When the moisture sensor value is less than 50, this indicates that there is not enough water in the pot. As a result, the buzzer will sound and a message “Water your plant” will be displayed on OLED. Else if the moisture sensor value is larger than 50, the buzzer will be in silence and a message “Your plant is in good condition” will be displayed on OLED.



If you don't want to type these code by yourself, you can download the whole program in the link below: https://makecode.microbit.org/_DV547gK8j9ms

Or you can download from this page:

▶ Simulator
🧩 Blocks
JS JavaScript
▼
Edit

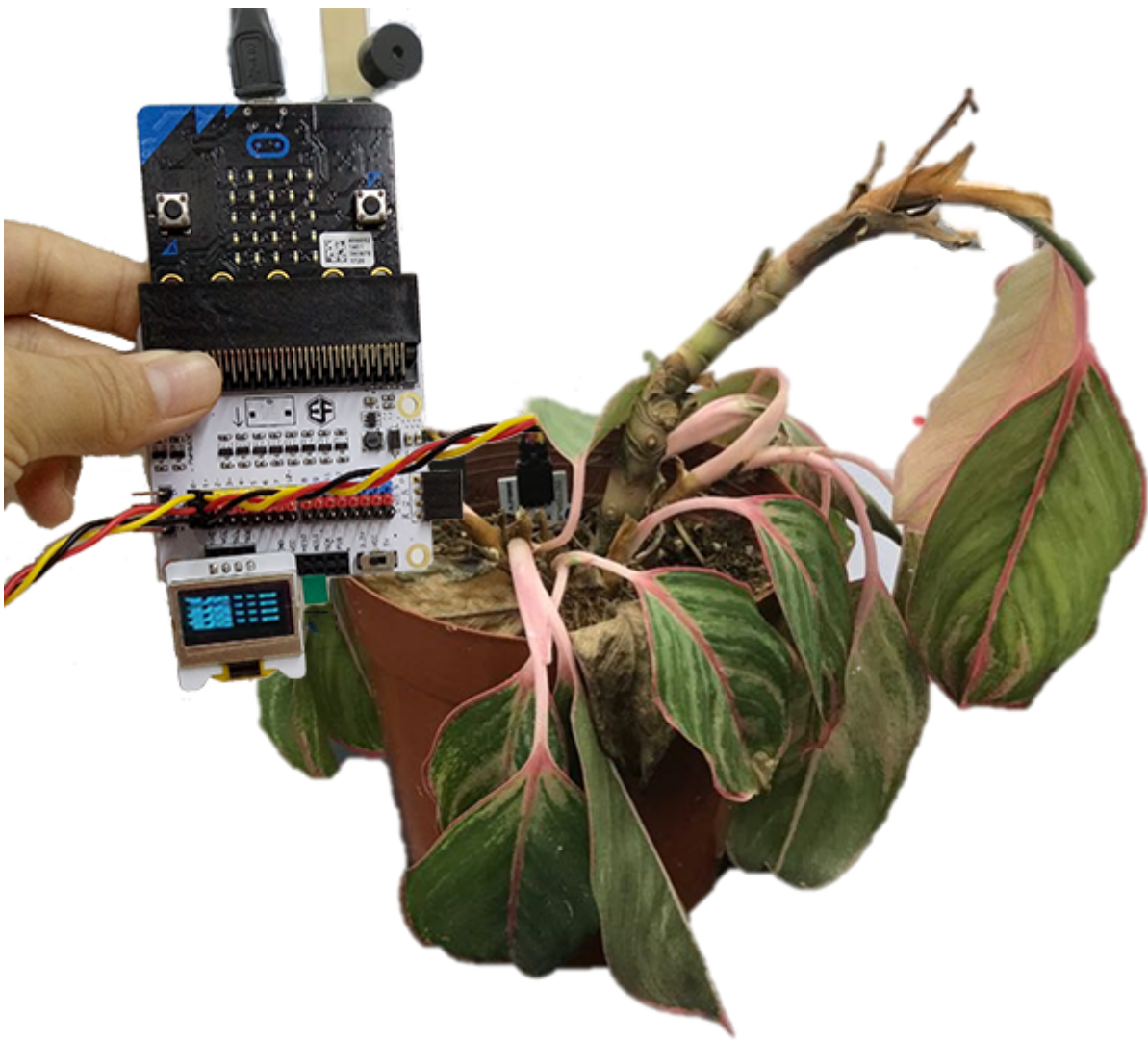
```

1 OLED.init(64, 128)
2 basic.forever(() => {
3     if (tinkercademy.MoistureSensor(AnalogPin.P1)
4         OLED.showString("Moisture level is:")
5         OLED.showNumber(tinkercademy.MoistureSens
6         OLED.showString("Water your plant!")
7         music.playTone(247, music.beat(BeatFraction
8     } else {
9         OLED.showString("Your plant is in good con
10    }
11 })
12

```

Problems 4
⬆

Finally! You have created a device to monitor your plant! Now, let's try it!



Download these code into micro:bit. Find a green plant and plug moisture sensor panel into the soil and watch. When there is not enough water, the buzzer will alarm to tell you “it’s time to water your plant!”. And when the plant has enough water, then the OLED panel will show you water is enough and no need to water the plant. Isn’t it very interesting?

8. case 06 Intruder Detection



Swiper no swiping! Stop burglars with this simple intruder detection system.

8.1. Step 0 – Pre-build Overview

In this project, we are going to create an intruder detection system which will sound when someone opens the door. The status of the house will be displayed on the OLED.

8.2. Materials:

- 1 x BBC micro:bit
- 1 x Micro USB cable
- 1 x Breakout board
- 1 x Crash Sensor
- 1 x OLED
- 1 x Buzzer

- 2 x Female-Female jumper wires

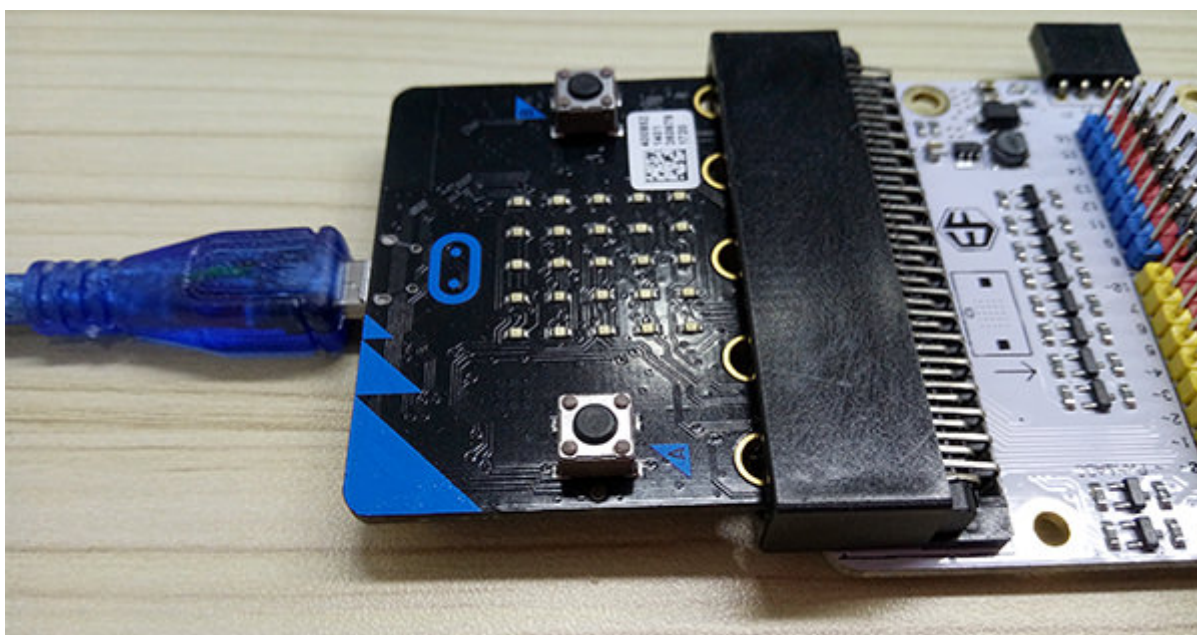
8.3. Goals:

- Get to know the Crash Sensor, OLED and Buzzer
- Make something with a OLED
- Make something with a Crash Sensor

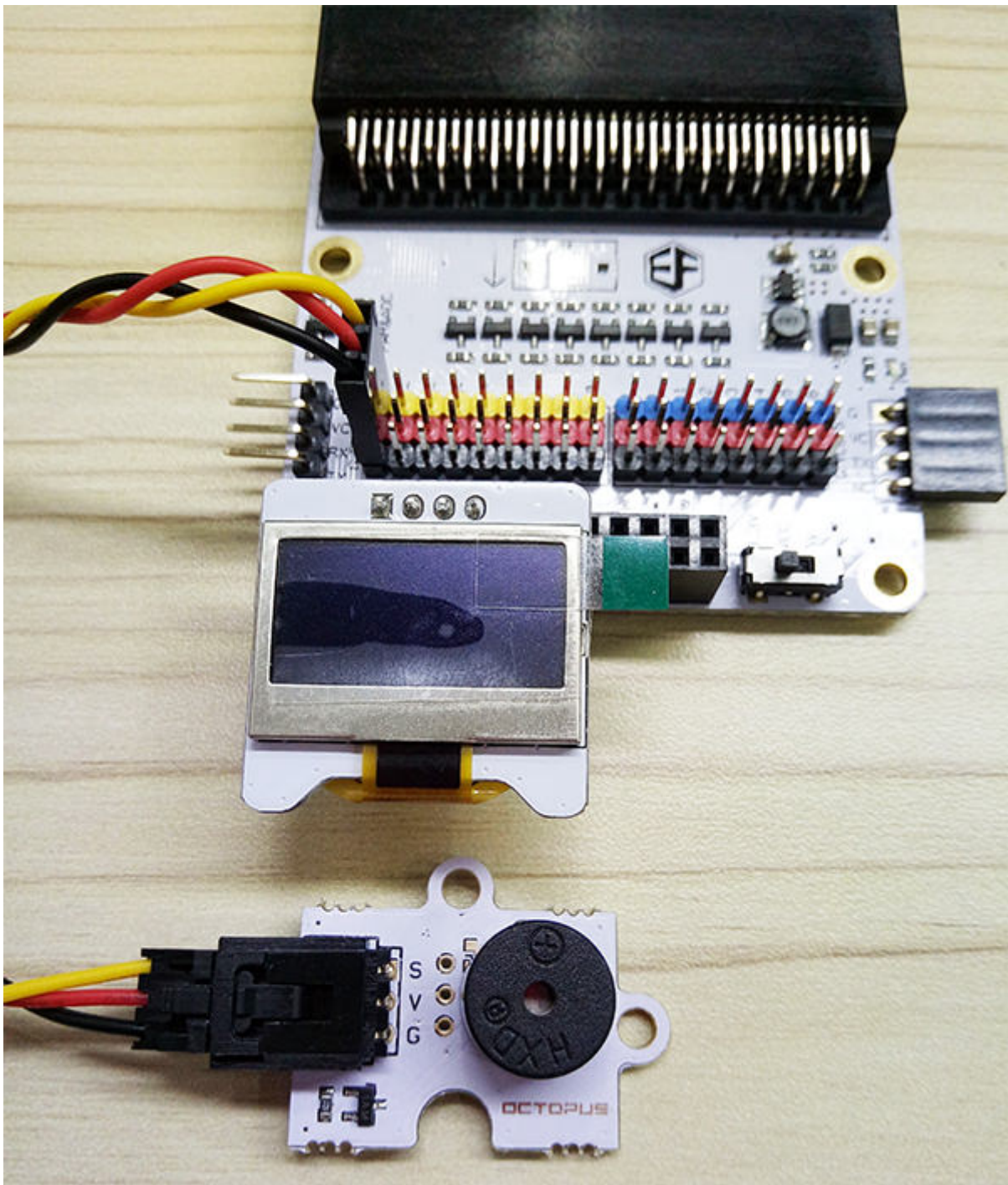
8.4. How to Make

Step 1 – Components

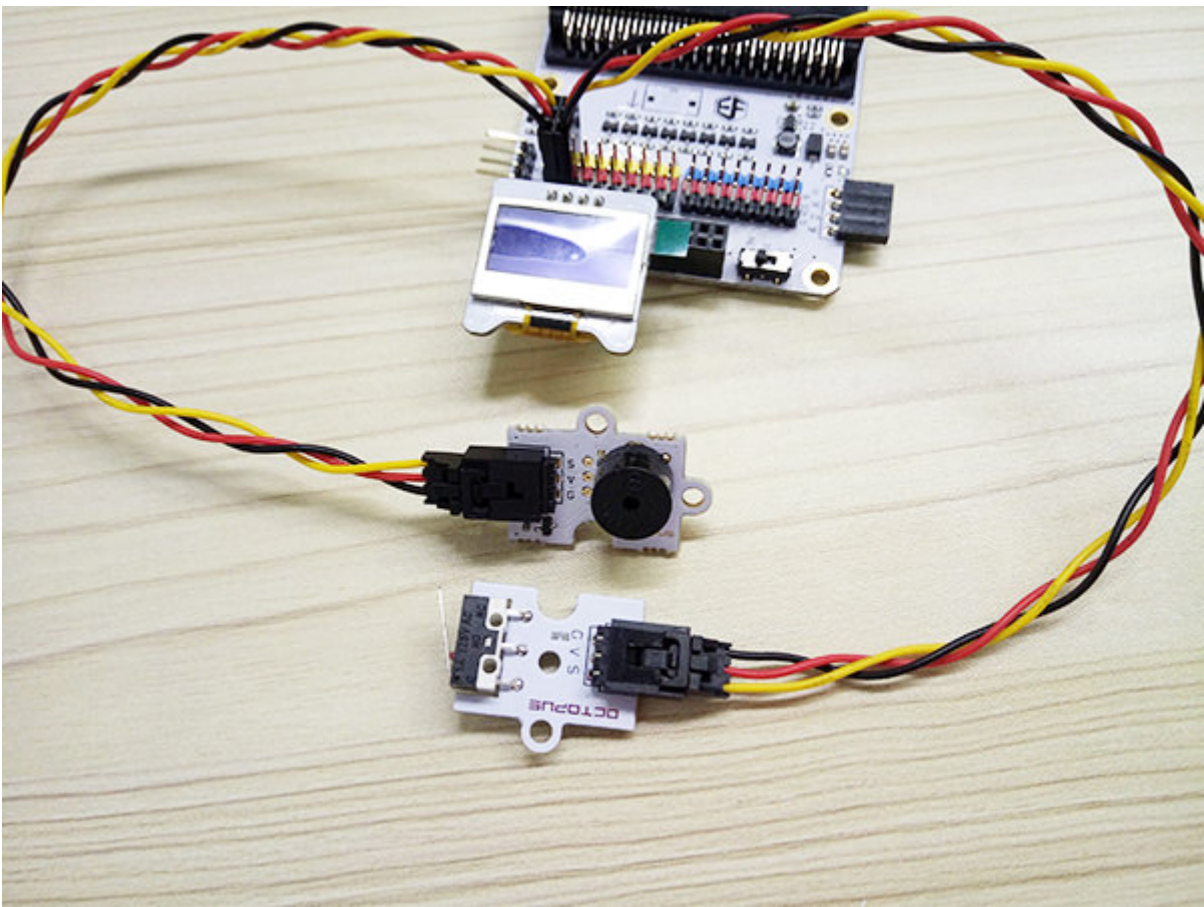
Insert the micro:bit into the Breakout Board and plug in the micro USB cable.



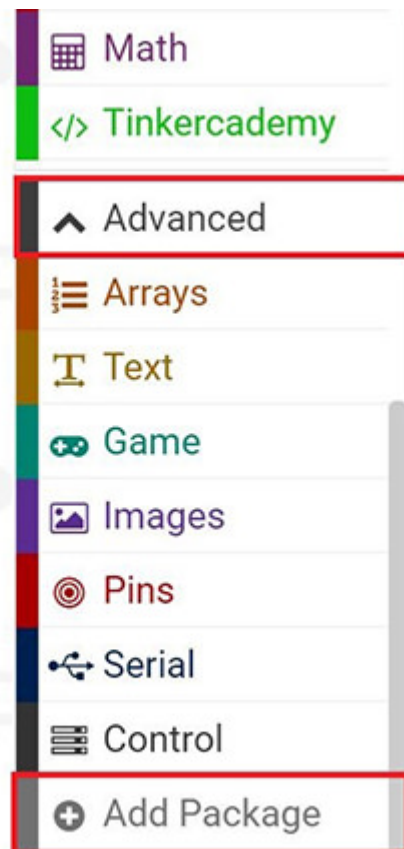
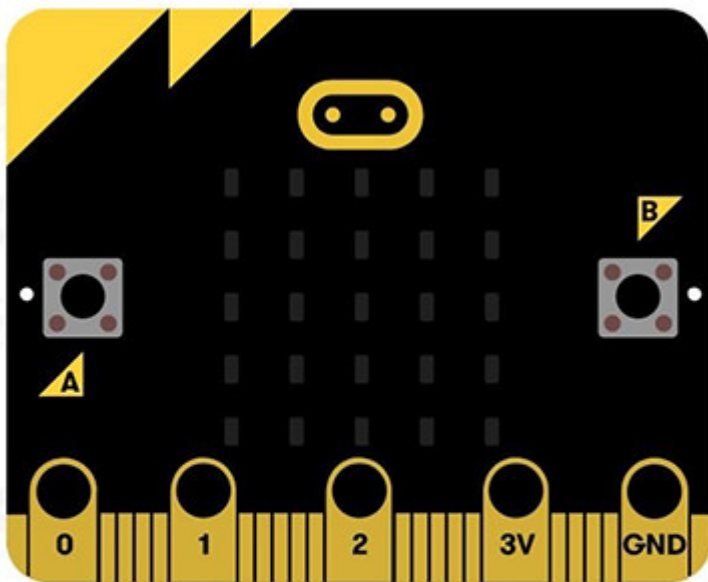
Then connect the buzzer to Pin 0 using the jumper cables. Plug in the OLED as shown in the picture below. You should be able to plug it into any of the three rows.



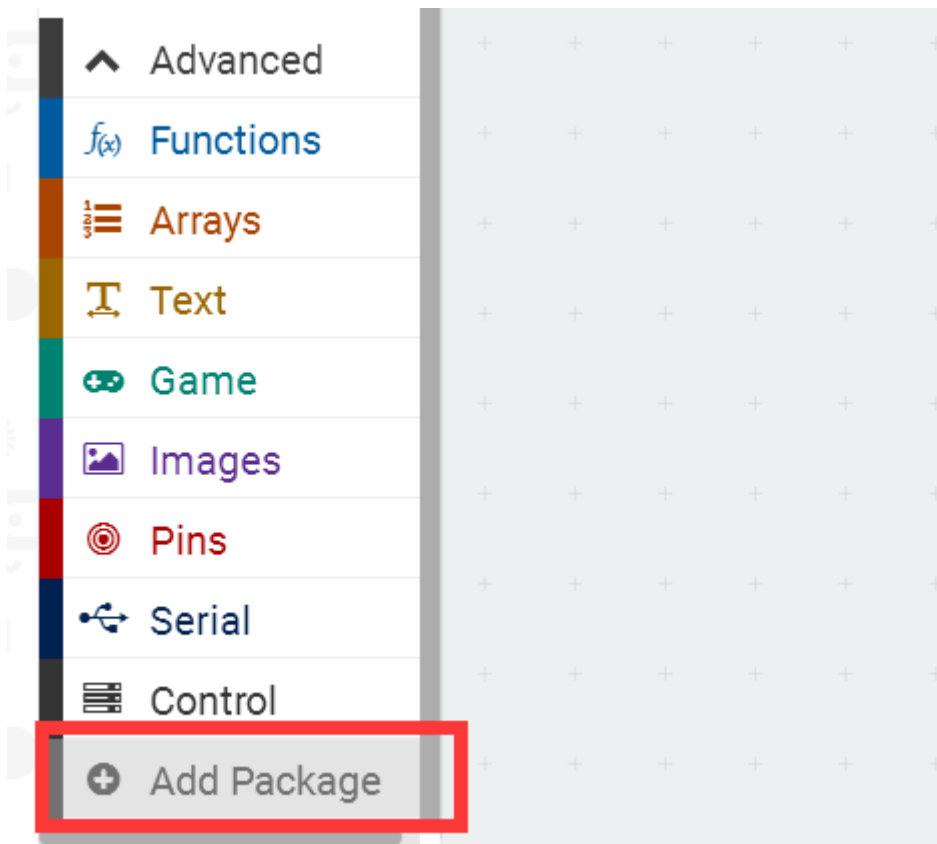
Plug in the crash sensor to Pin 1. Make sure the colour of the wire follows the coloured pins on the breakout board.



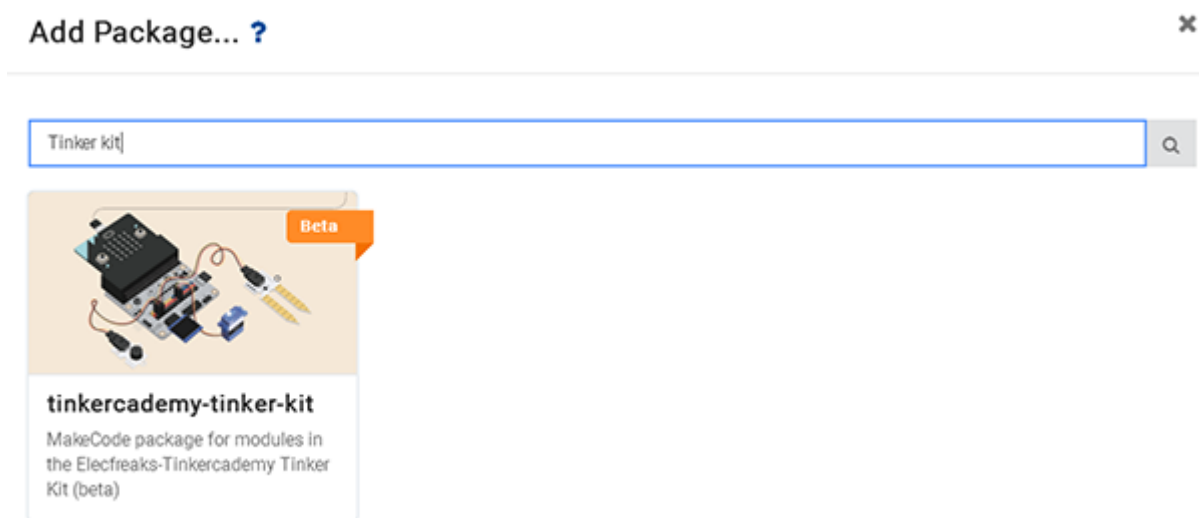
Step 2 – Pre-coding



We will add a package of code to enable us to use our kit components. Click on Advanced in the Code Drawer to see more code section and look at the bottom of the Code Drawer for Add Package.

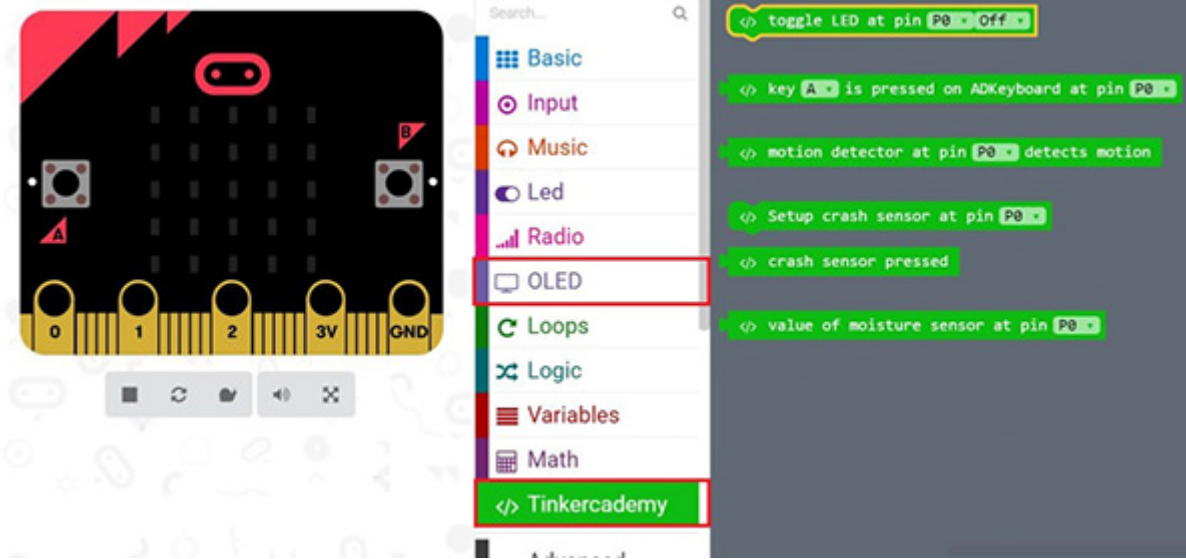


This will open up a dialog box. Search for “tinker kit” and then click it for downloading this package.

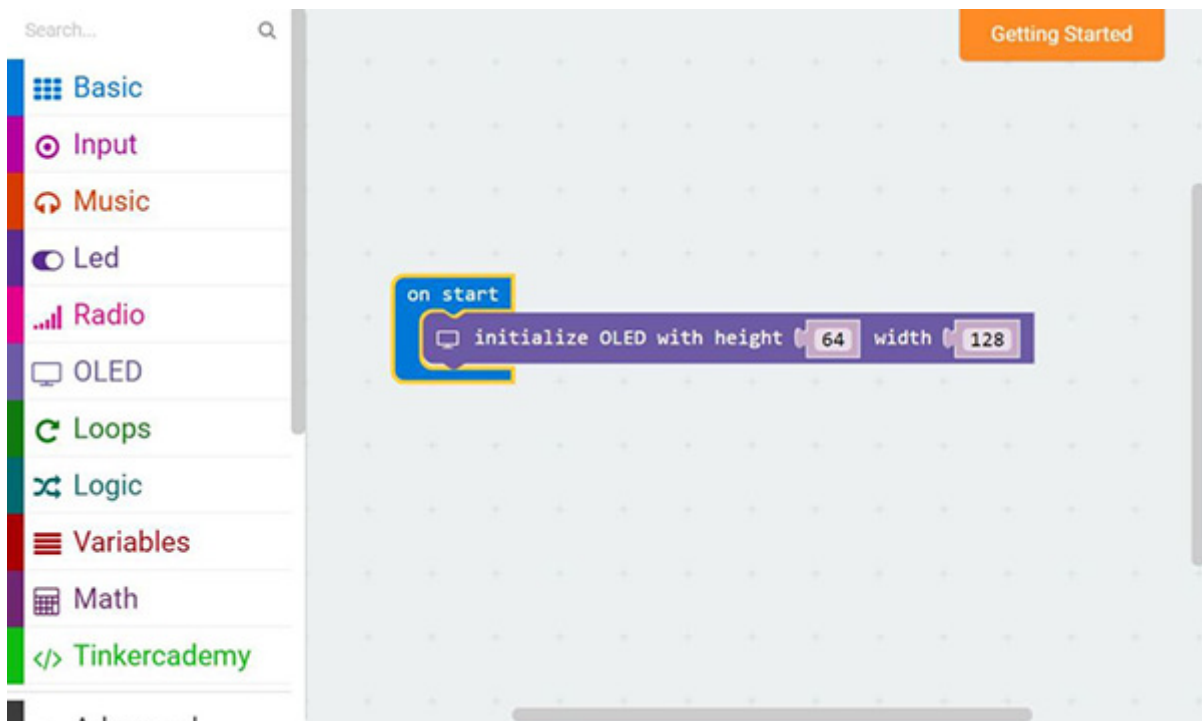


Note: If you get a warning telling you some packages will be removed because of incompatibility issues, either follow the prompts or create a new project in the Projects file menu.

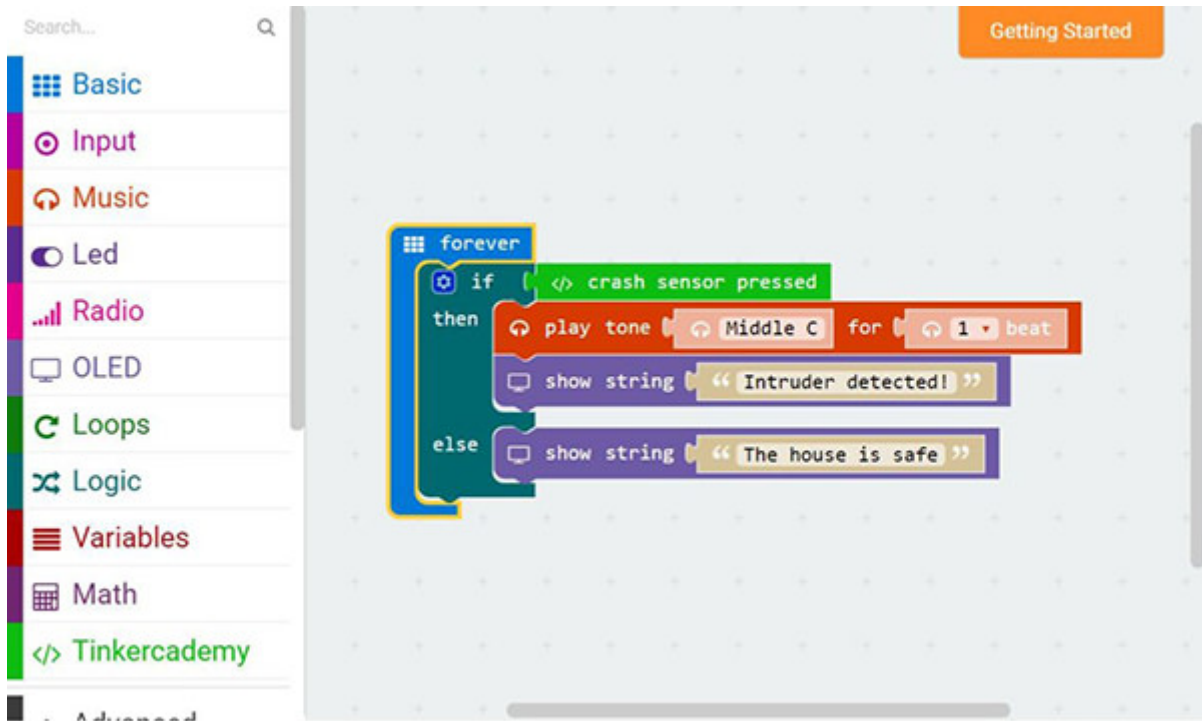
Step 3 – Coding



Click on Tinkercademy inside the Code Drawer to find our custom blocks for the various components in your kit.



You should always initialize the OLED at the beginning. 64 and 128 represent the height and width of the OLED respectively.

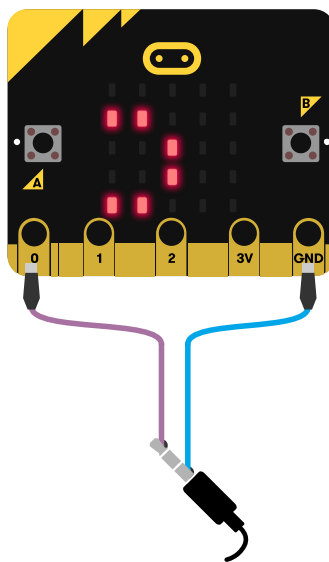


Since there are only two conditions, we need only one “else-if” statement.

When crash sensor is triggered, the buzzer will sound and the OLED will display the message “Intruder Detected”. Or else, if there is no force is applied to the crash sensor, the buzzer will not sound and the OLED will display the message “The house is safe”.

If you don't want to type these code by yourself, you can download the whole program directly from the link below. https://makecode.microbit.org/_A0zFxqMPMXbo

Or you can download from the page below.



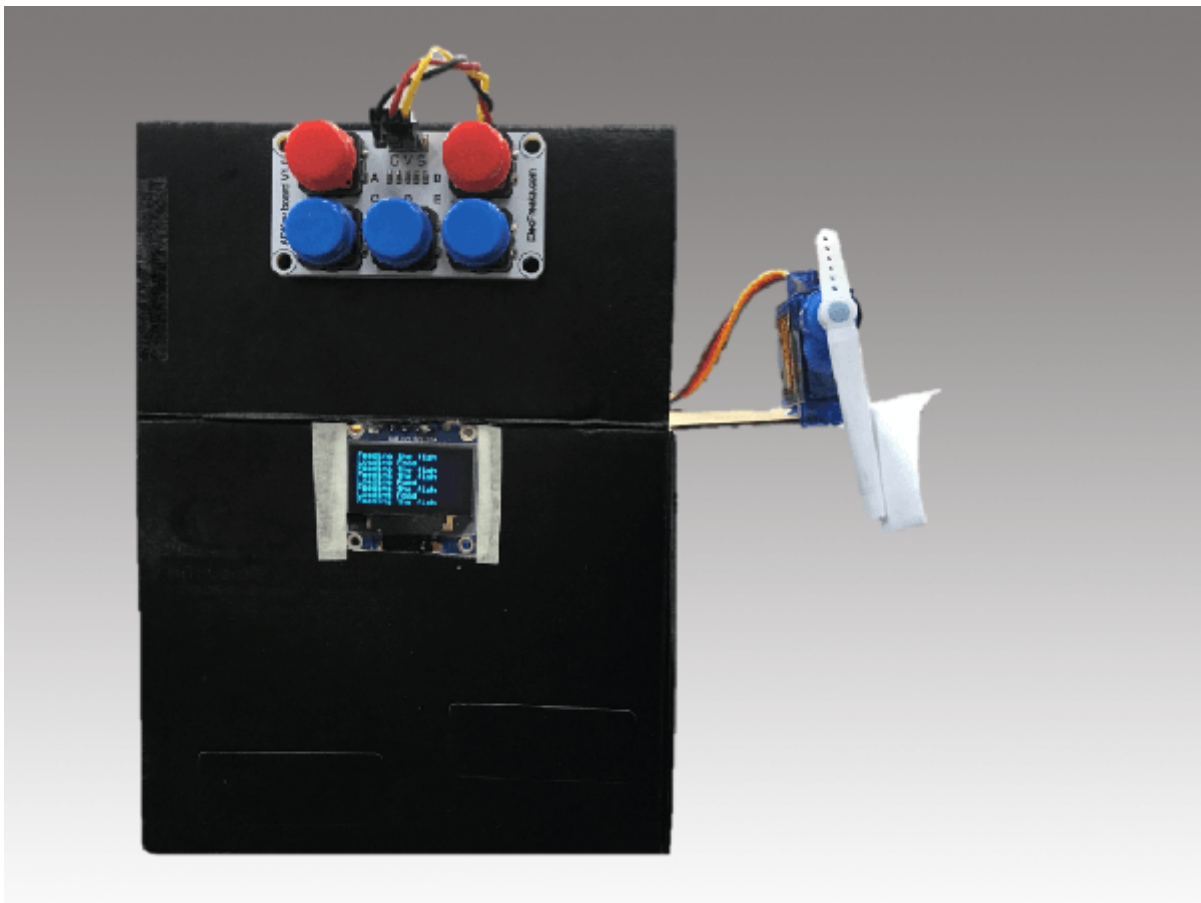
Step 4 – Success!

Voilà! You have created a intruder detector!

9. case 07 Fish Feeder

Tired of feeding your fish by hand? Here's the micro:bit project for you!
In this course, we will use a ADKeypad to control the motion of a servo to feed fish.

9.1. Step 0 – Pre-build Overview



In this project, we are going to create a fish feeding machine. The movement of the servo will be controlled by the two red buttons on the ADKeypad and the OLED will display a message showing the status of the servo.

9.2. Materials:

- 1 x BBC micro:bit
- 1 x Micro USB cable
- 1 x Breakout board
- 1 x ADKeypad

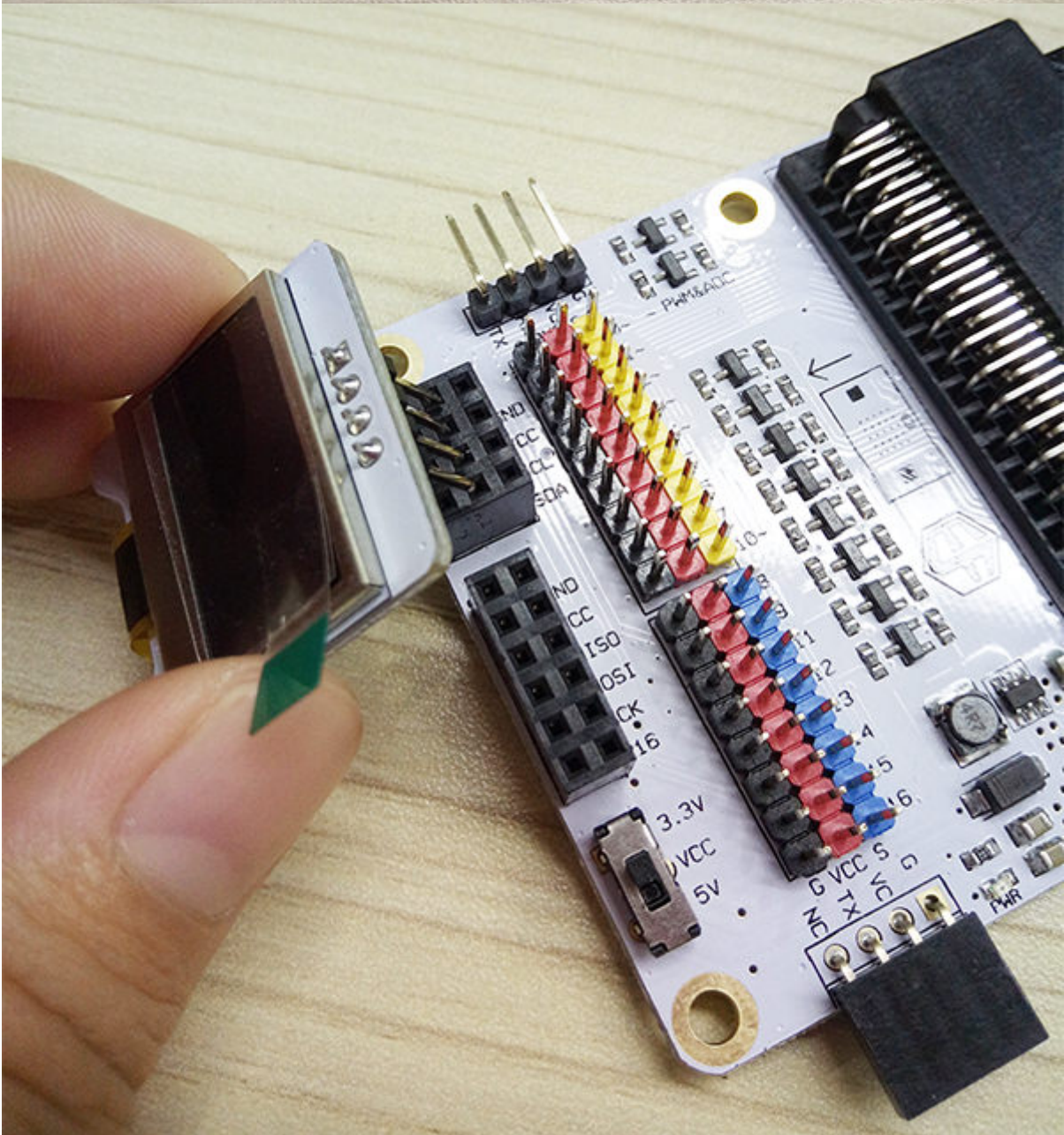
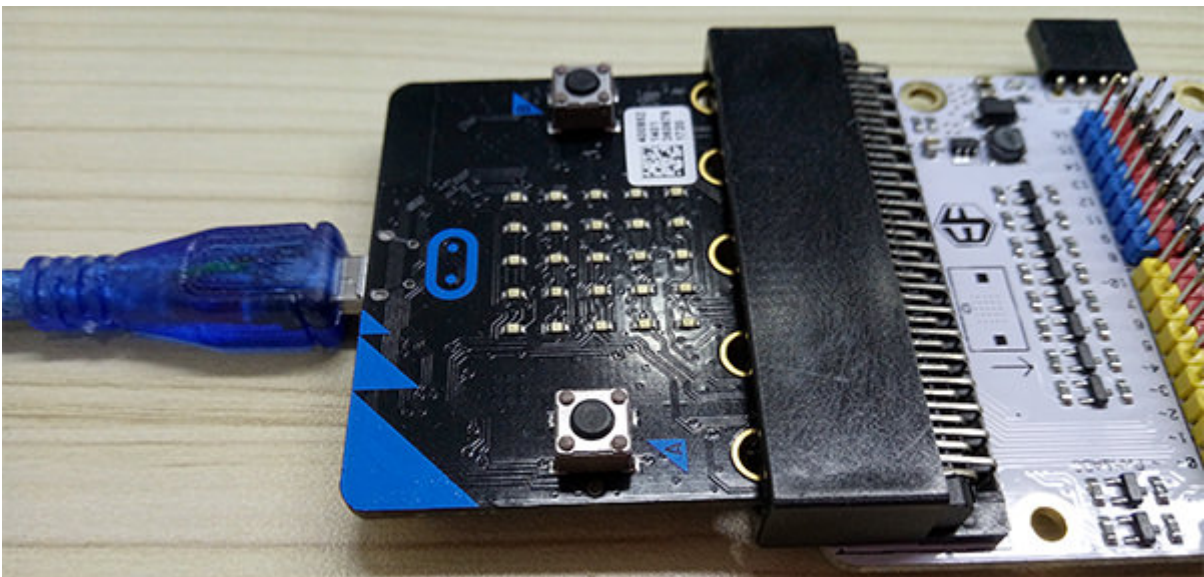
- 1 x OLED
- 1 x Servo
- Wires

9.3. Goals:

- Get to know the ADKeypad, OLED and servo
- Make something with a servo
- Make something with a OLED

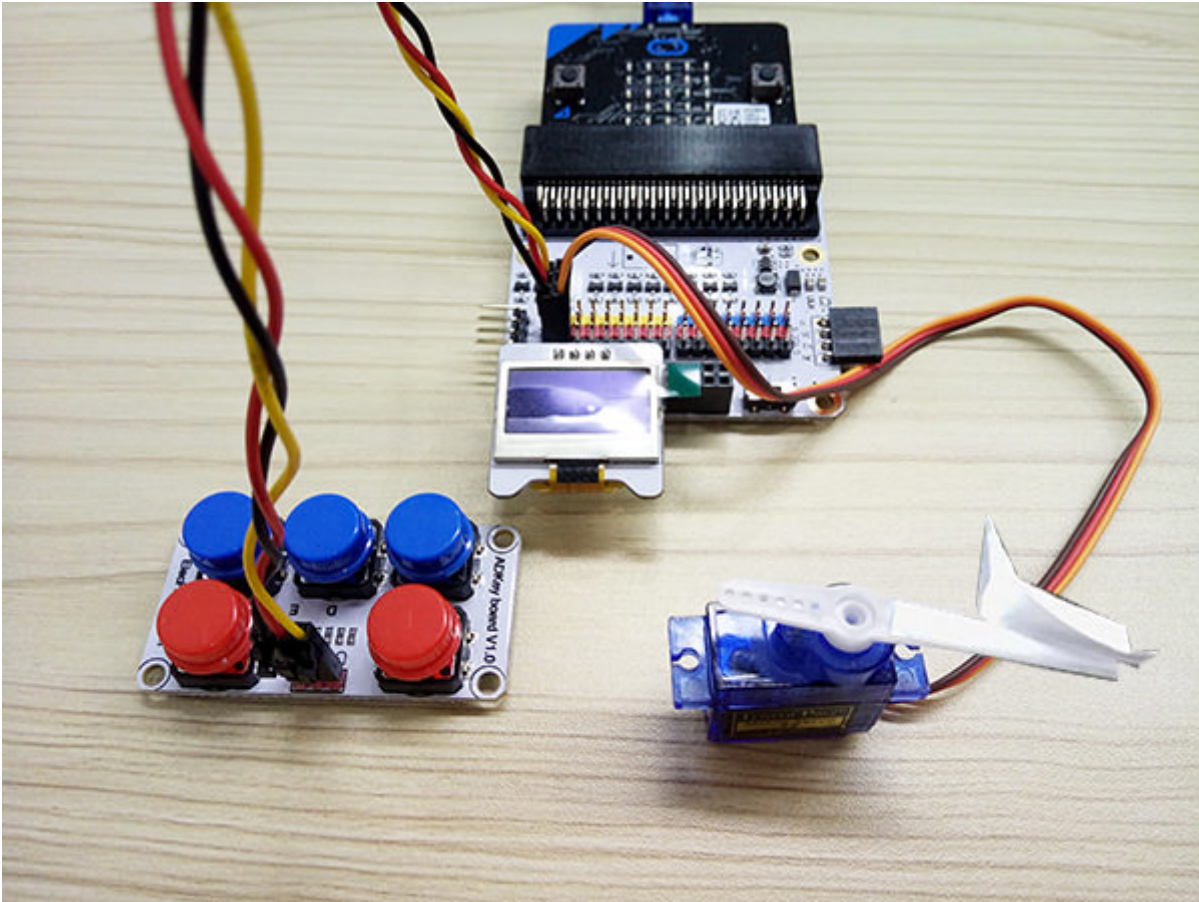
9.4. How to Make

Step 1 – Components



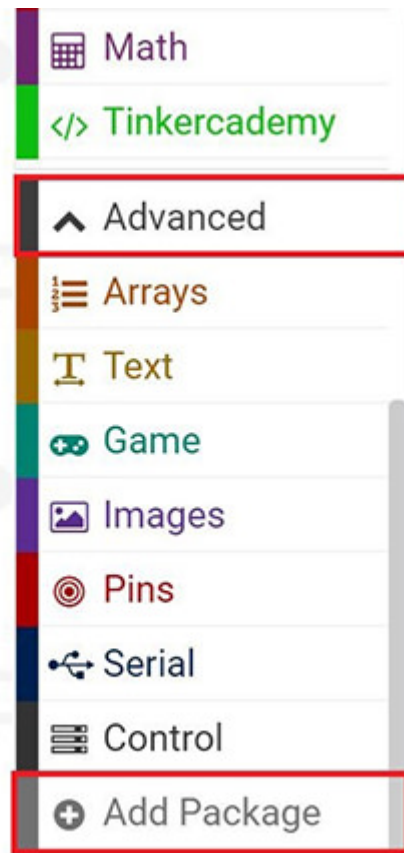
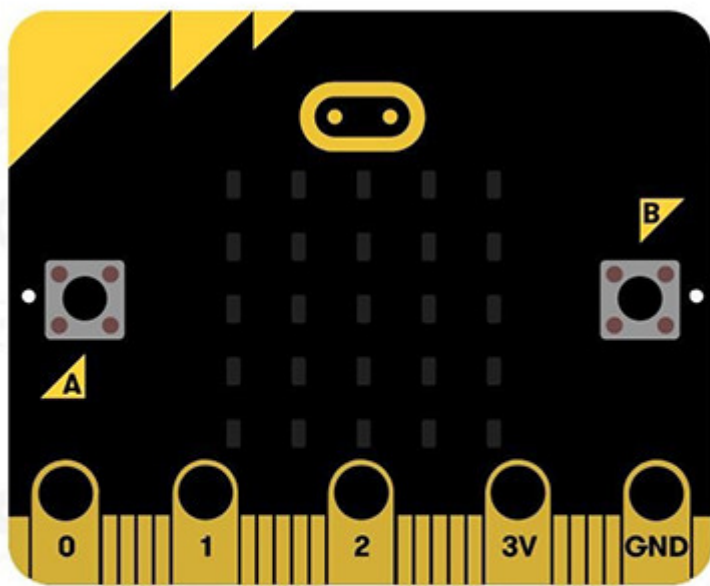
Insert the micro:bit into the Breakout Board and plug in the micro USB cable, then plug in the OLED as shown in the picture above. You should be able to plug it into any of the three rows

Connect the ADKeypad to Pin 0 and the servo to Pin 1. Make sure the colour of the wire matches the colour of the pins on the Breakout Board.

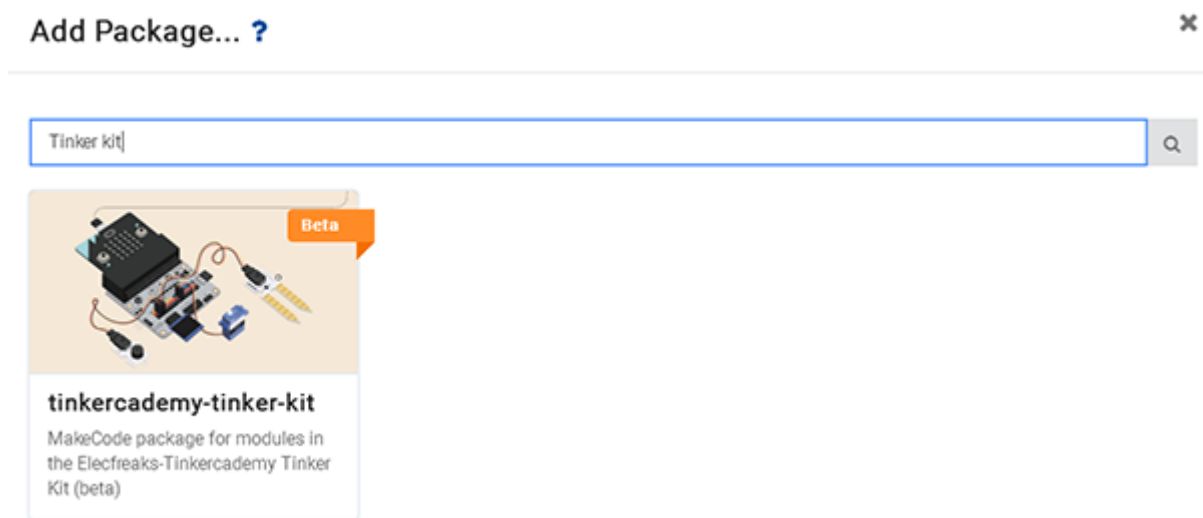


Step 2 – Pre-coding

We will add a package of code to enable us to use our kit components. Click on Advanced in the Code Drawer to see more code section and look at the bottom of the Code Drawer for Add Package.

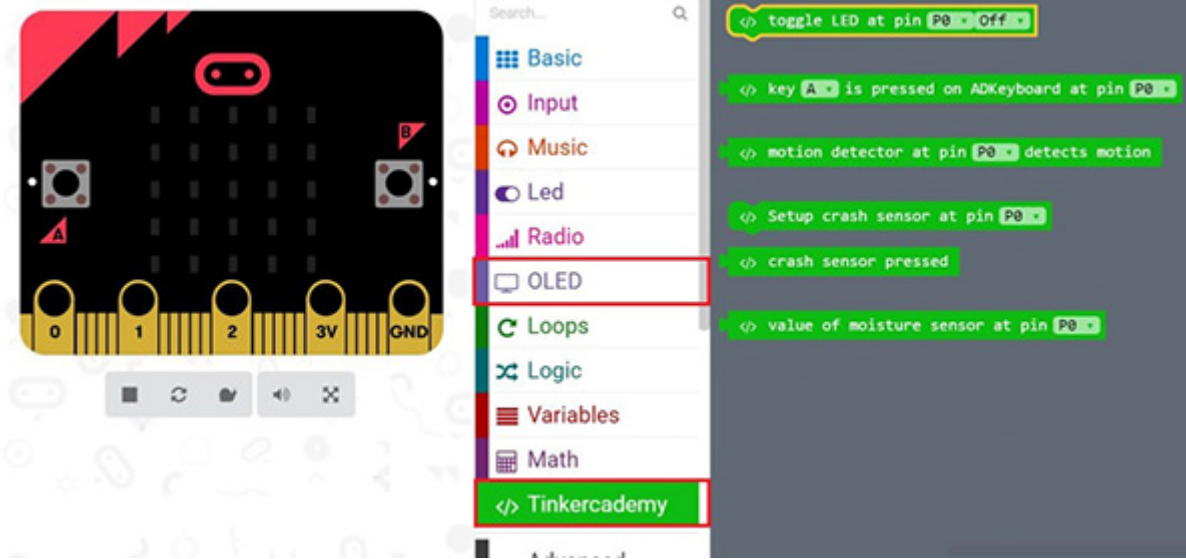


This will open up a dialog box. Search for “tinker kit” and then click it to downloading this package.

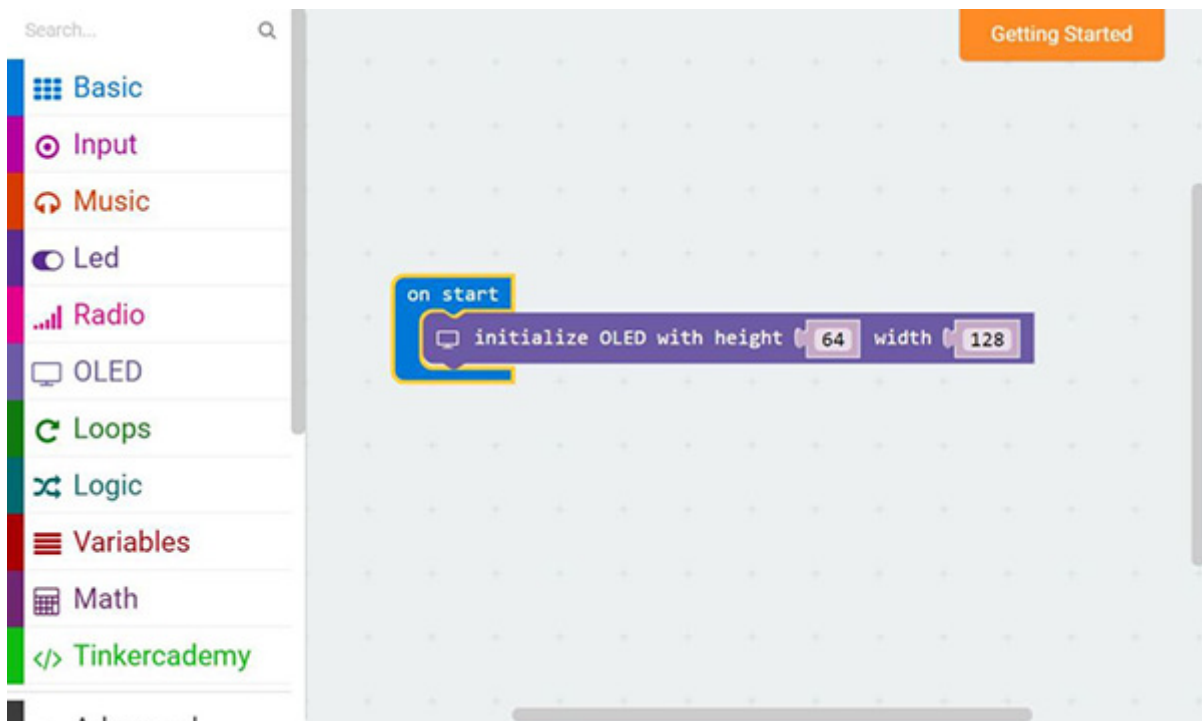


Note: If you get a warning telling you some packages will be removed because of incompatibility issues, either follow the prompts or create a new project in the Projects file menu.

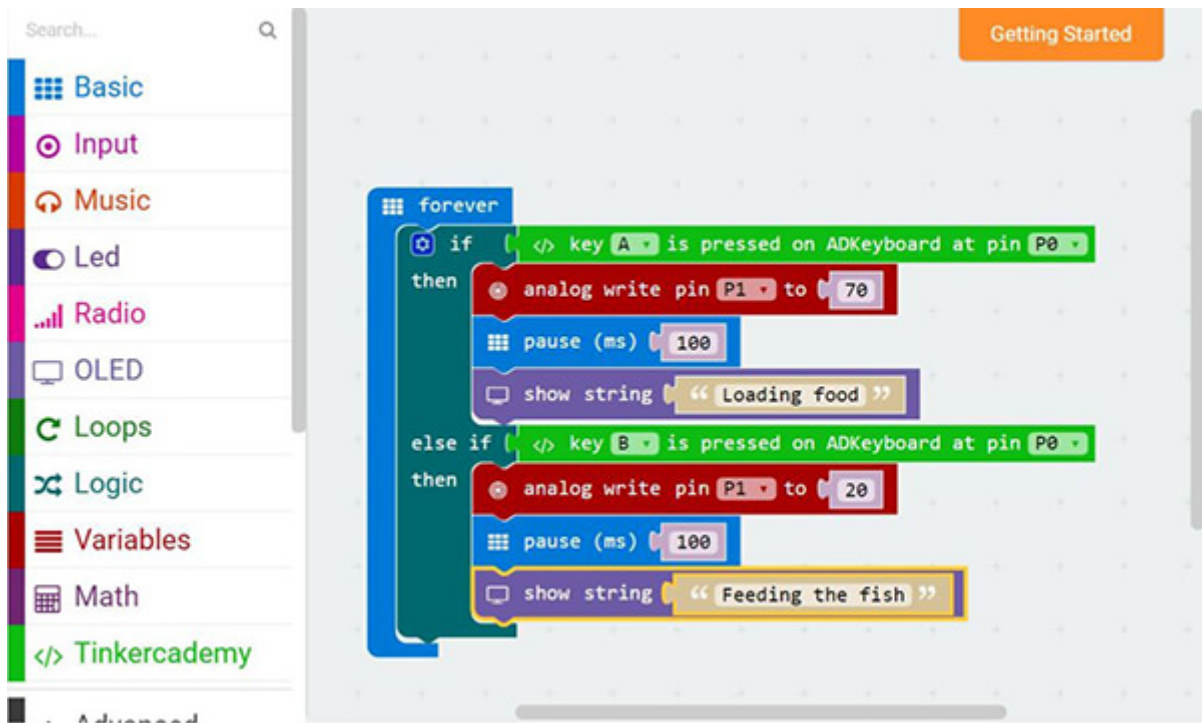
Step 3 – Coding



Click on Tinkercademy inside the Code Drawer to find our custom blocks for the various components in your kit.



You should always initialize the OLED at the beginning. 64 and 128 represent the height and width of the OLED respectively.

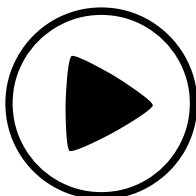


Since there are only two conditions, we need only one 'else-if' statement. If the button A of the ADKeypad is pressed, the servo will turn to angle 70 and the OLED will display "Loading food". Or else,if button B of the ADKeypad is pressed, the servo will turn to angle 20 and the OLED will display "Feeding the fish". You can adjust the servo angle to suit your requirement.

If you don't want to type these code by yourself, you can download the whole program from the link below.

https://makecode.microbit.org/_3HJDazbma3H4

Or you can download from the page below.





Step 4 – Success!

Voilà! You have created a fish feeding machine!

10. case 08 Motion Detector

Don't like people sneaking up on you? Here's just the right micro:bit project for you! In this course, we learn how to make use of the motion sensor, the moisture sensor as well as how they can be coded for.

10.1. Goals:

- Get to know the PIR Sensor Brick and moisture sensor.
- Make something with a PIR Sensor Brick.
- Make something with a Moisture Sensor.

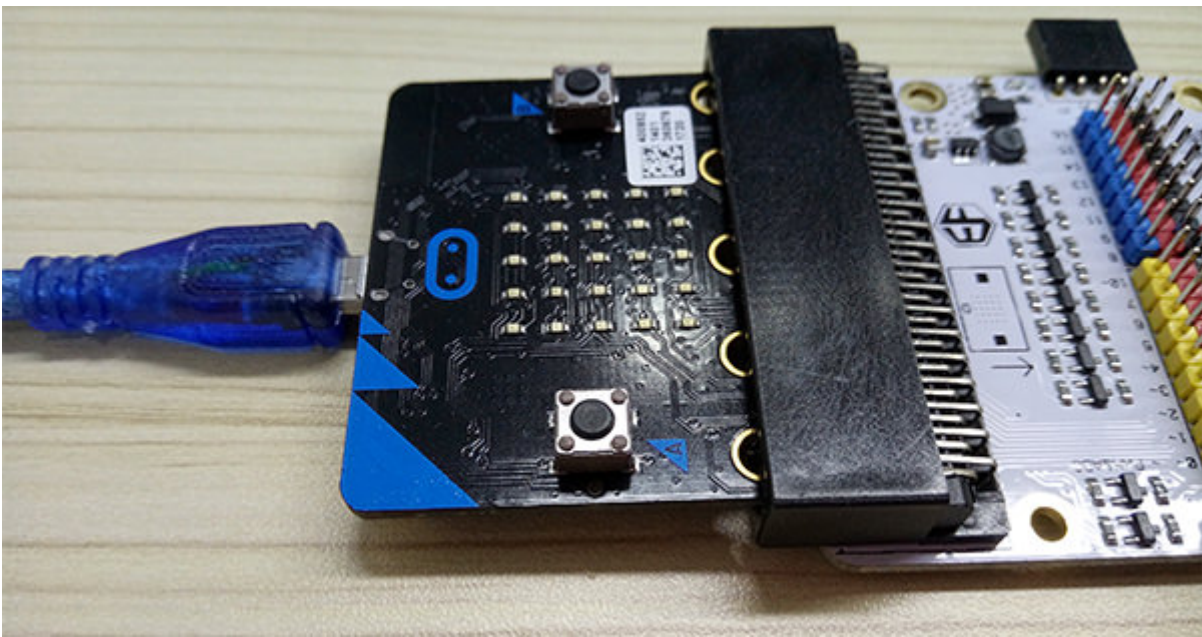
10.2. Materials:

- 1 x BBC micro:bit
- 1 x Micro USB cable
- 1 x Breakout board
- 1 X Mini buzzer
- 1 X Octopus PIR sensor Brick
- 1 X Moisture sensor
- 2 X Female-Female jumper wires

10.3. How to Make

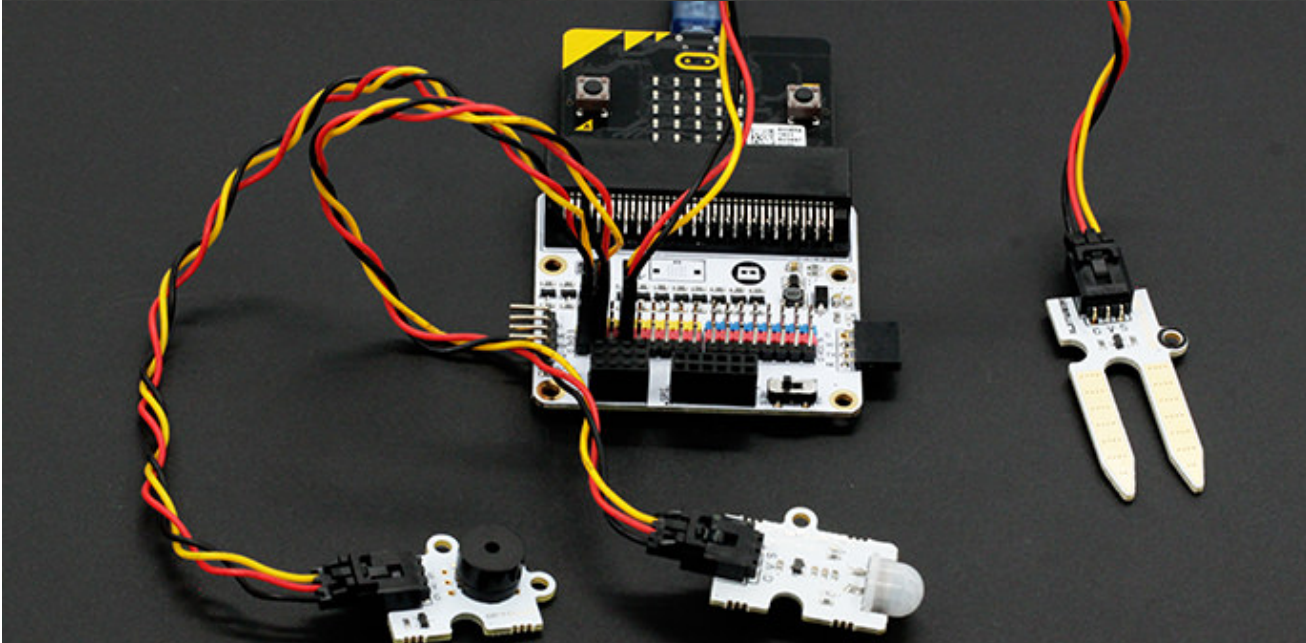
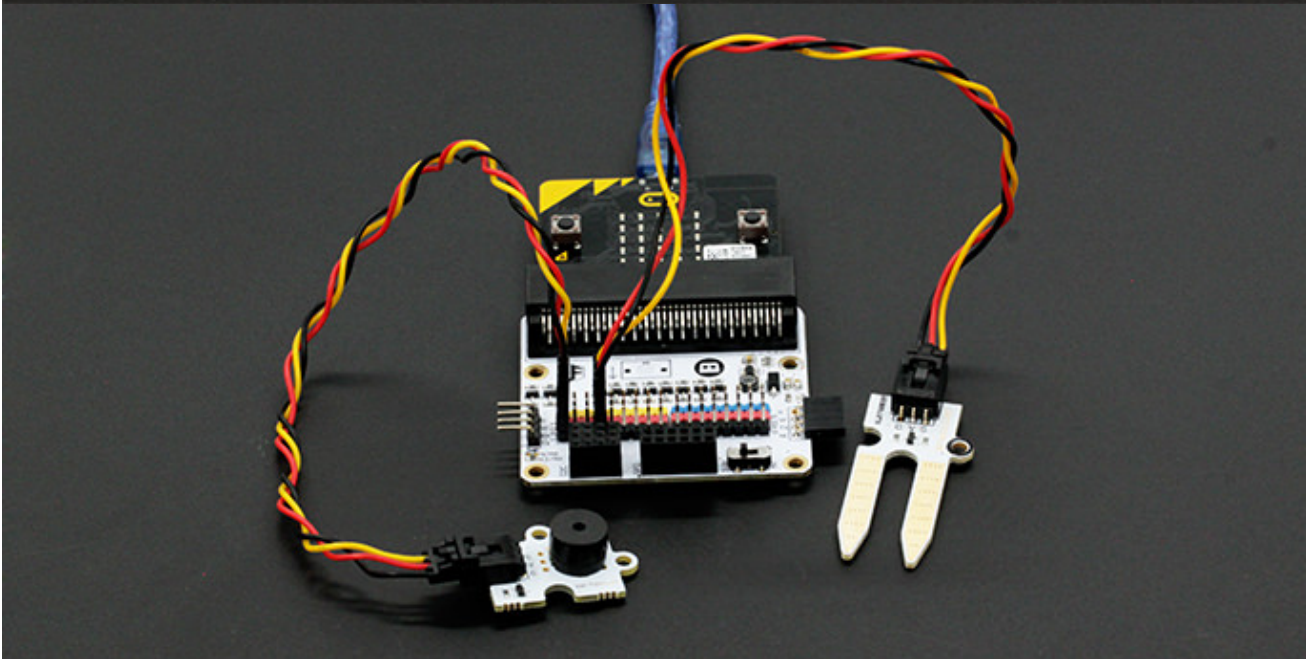
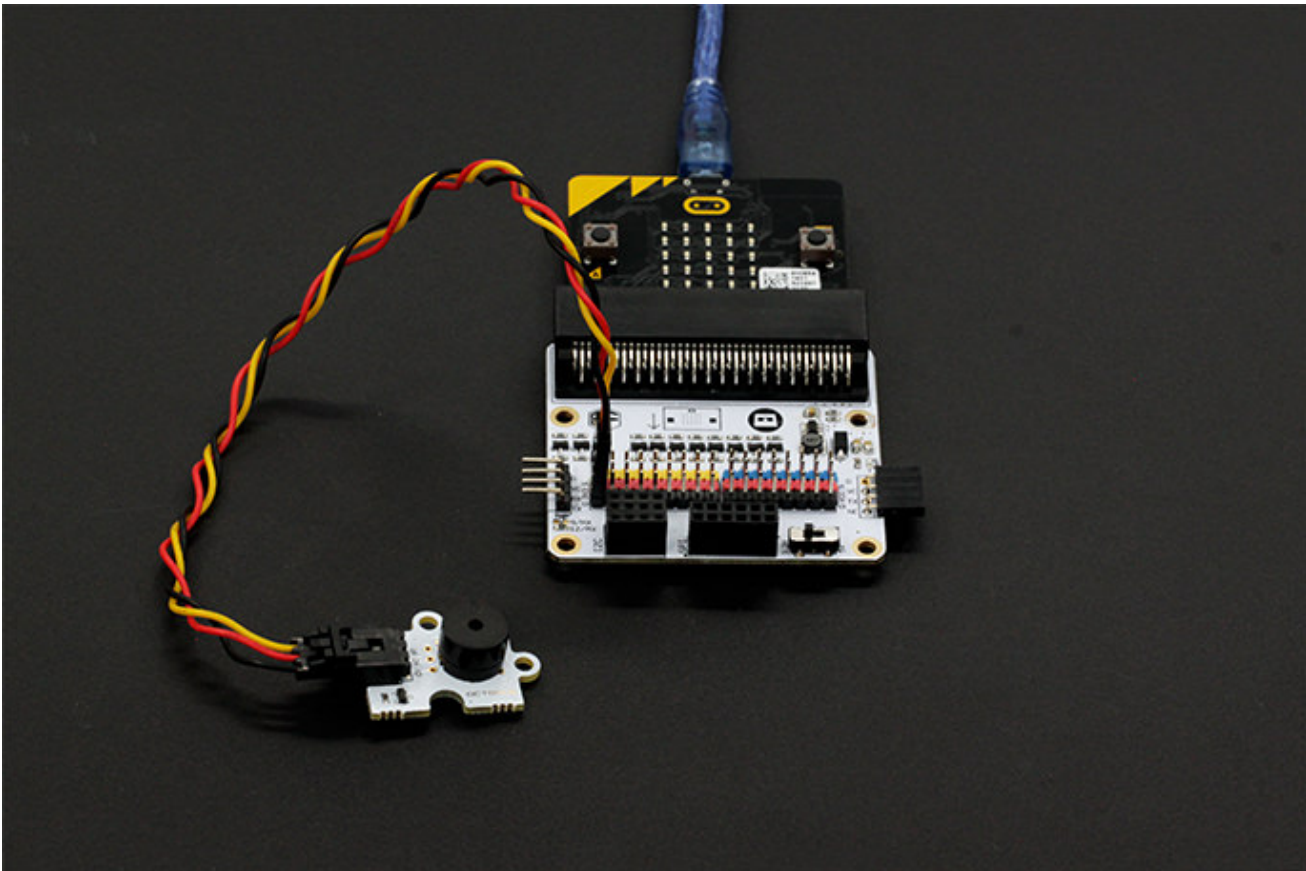
Step 1:

After connecting one end of the USB cable to your computer, connect the other end to the micro:bit as shown in the picture. Connect the side of the micro:bit where the pins are located to the breakout board.



Step 2:

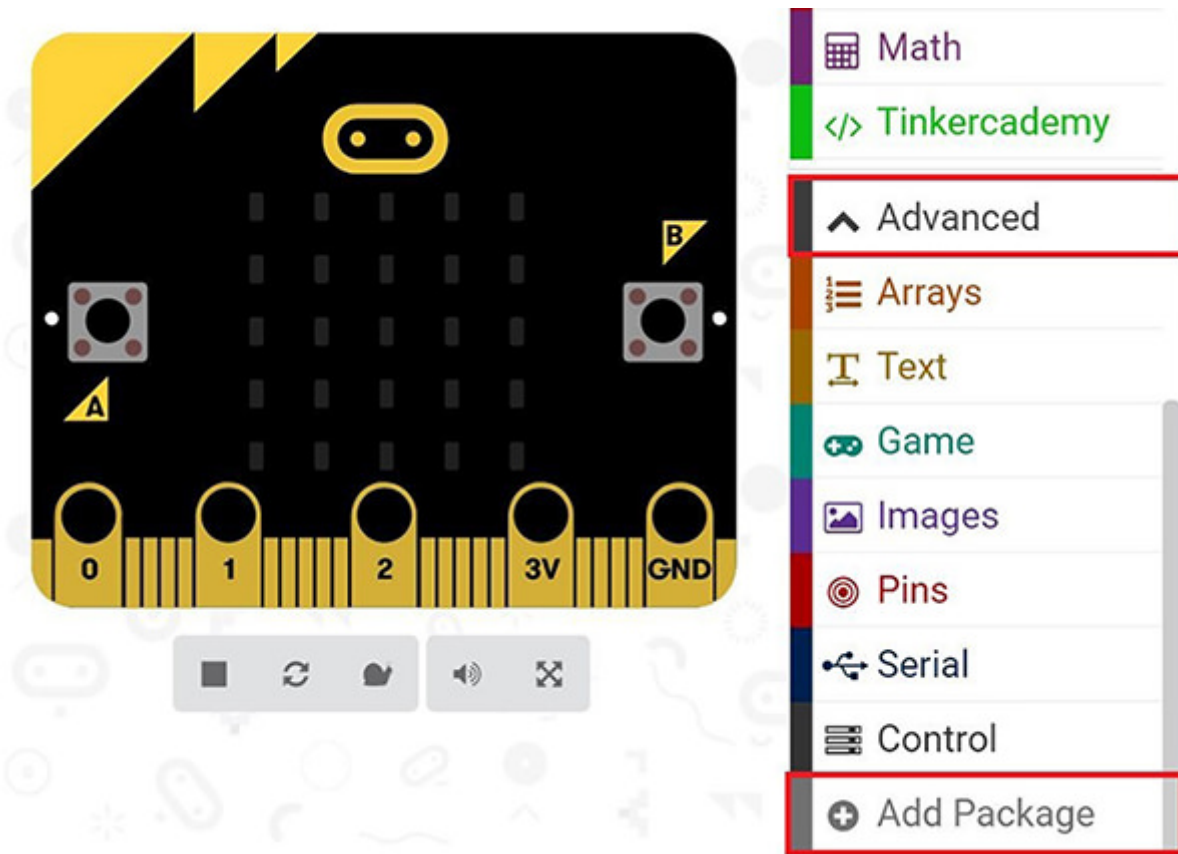
Plug in the buzzer to Pin 0 (the pins beside the number '0' on the breakout board) . Plug in the moisture sensor to Pin 3. Plug in the motion sensor to Pin 1. Make sure the colour of the wire of the buzzer and the ADKeyboard follows the colour of the pins on the breakout board.



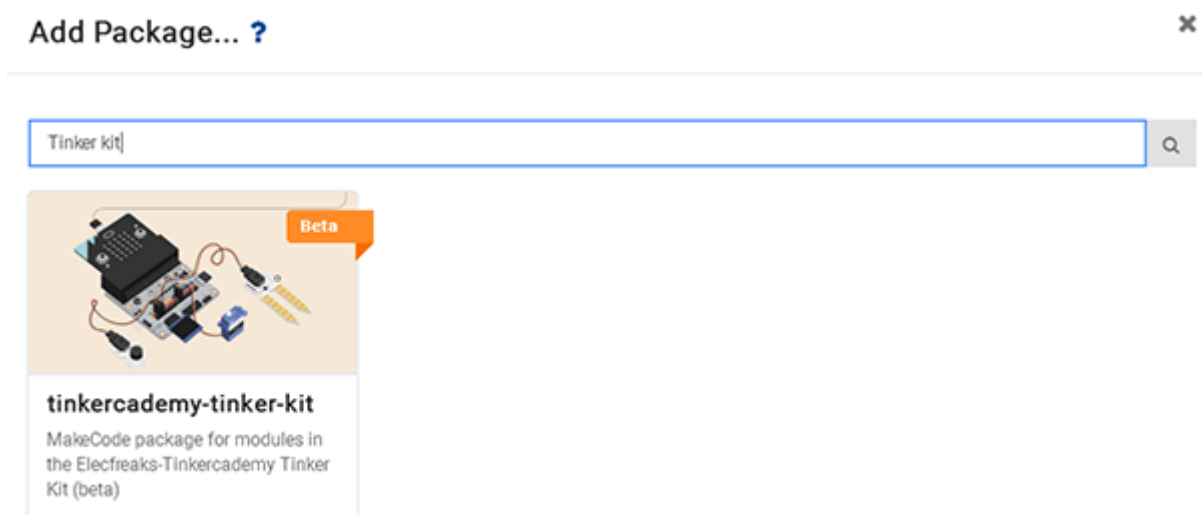
Step 3:

Click on “Advanced” in the code drawer to see more code sections.

To code for our extra kit components (the ADKeyboard and the buzzer), we will need to add a package of code.

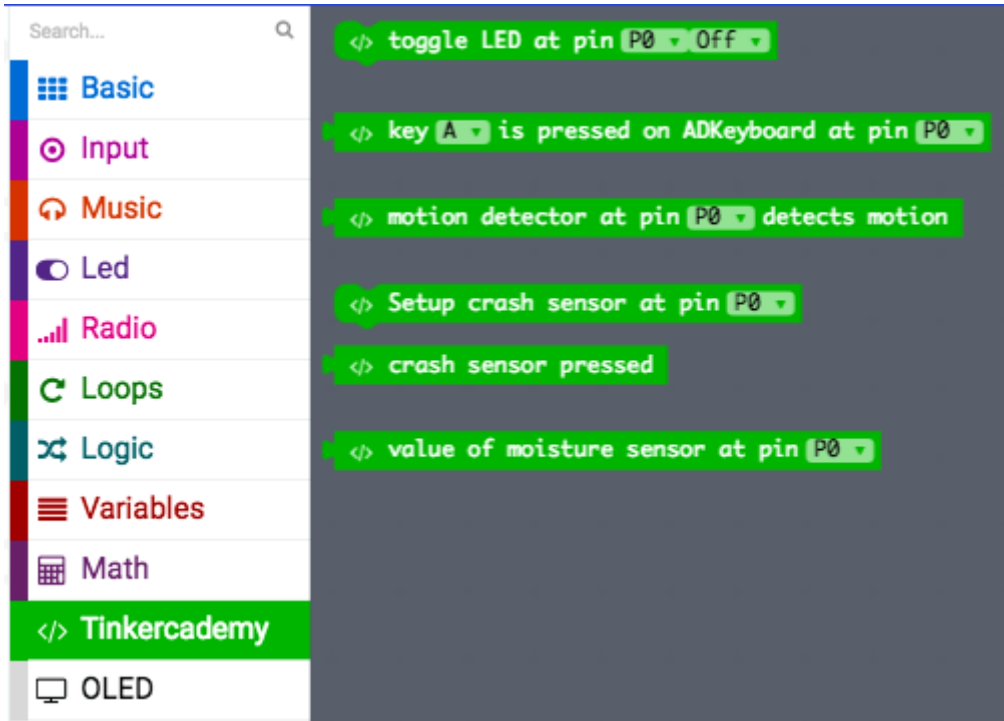


Look at the bottom of the code drawer for “Add Package” and click it to open up a dialogue box. Search for “tinker kit” and then click on it to download this package.



Note: If you get a warning telling you some packages will be removed because of incompatibility issues, either follow the prompts or create a new project in the Projects file menu.

Click on Tinkercademy inside the Code Drawer to find our custom blocks for the various components in your kit.

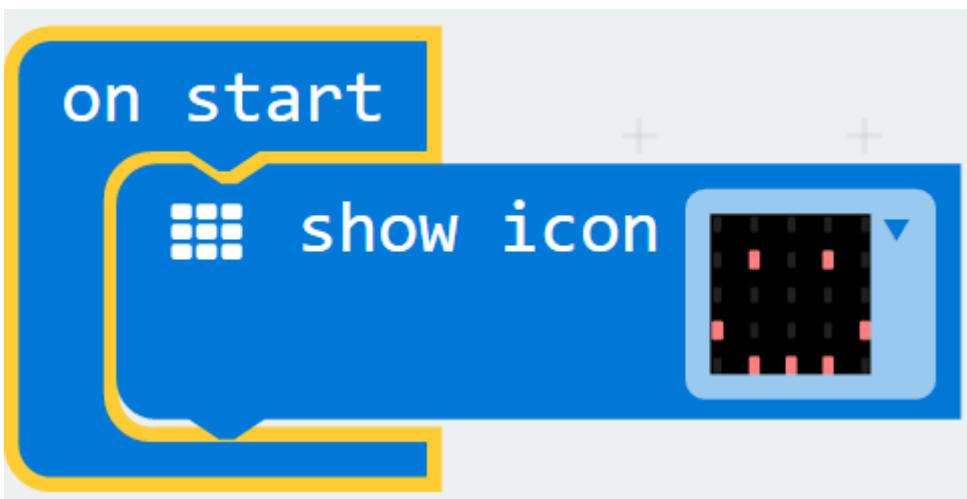


For this project, we are going to use the blocks read value from moisture sensor and motion sensor.

Step 4

In this step, we will code the Micro:bit with Block Editor. We begin by coding a starting screen, by placing the “Show Icon” block under the “On Start” block as shown in the picture on the right.

This causes the icon to appear on the screen whenever the micro:bit is powered on.



Step 5

Next, let's create some music using the moisture sensor values.

Select the "Play Tone" block under the "Music" code section and place the value of Moisture Sensor code block in it. The pitch can be adjusted by multiplying the sensor values by different numbers, as shown in the image on the left.



Step 6

Finally, make the buzzer sound when the motion sensor detects movements. The micro:bit will only show an icon on the screen if there is no movement.

This can be done by using a conditional (if-then-else) statement and inserting the relevant blocks in the appropriate spots, as shown in the picture on the right.

```

on start
  show icon [LED Matrix]

forever
  if (</> motion detector at pin P1 detects motion)
  then
    show icon [LED Matrix]
    show icon [LED Matrix]
    show icon [LED Matrix]
    play tone [value of moisture sensor at pin P3 x 1] for 1 beat
    play tone [value of moisture sensor at pin P3 x 2] for 1 beat
    play tone [value of moisture sensor at pin P3 x 3] for 1 beat
    play tone [value of moisture sensor at pin P3 x 4] for 1 beat
    play tone [value of moisture sensor at pin P3 x 5] for 1 beat
    play tone [value of moisture sensor at pin P3 x 4] for 1 beat
    play tone [value of moisture sensor at pin P3 x 3] for 1 beat
    play tone [value of moisture sensor at pin P3 x 2] for 1 beat
    play tone [value of moisture sensor at pin P3 x 1] for 1 beat
    play tone [value of moisture sensor at pin P3 x 5] for 1 beat
    play tone [value of moisture sensor at pin P3 x 1] for 1 beat
  else
    show icon [LED Matrix]

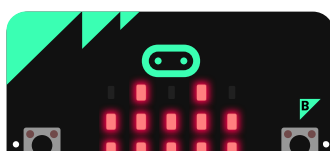
```

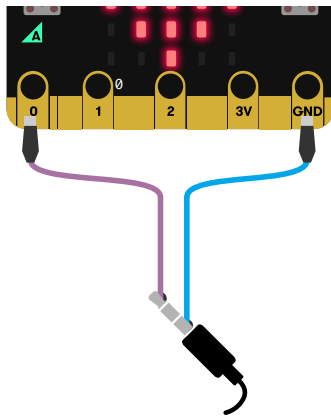
If you don't want to type these code by yourself, you can download the whole code directly from the link below.

https://makecode.microbit.org/_8xYPibiLdeYR

Or you can download from the page below.

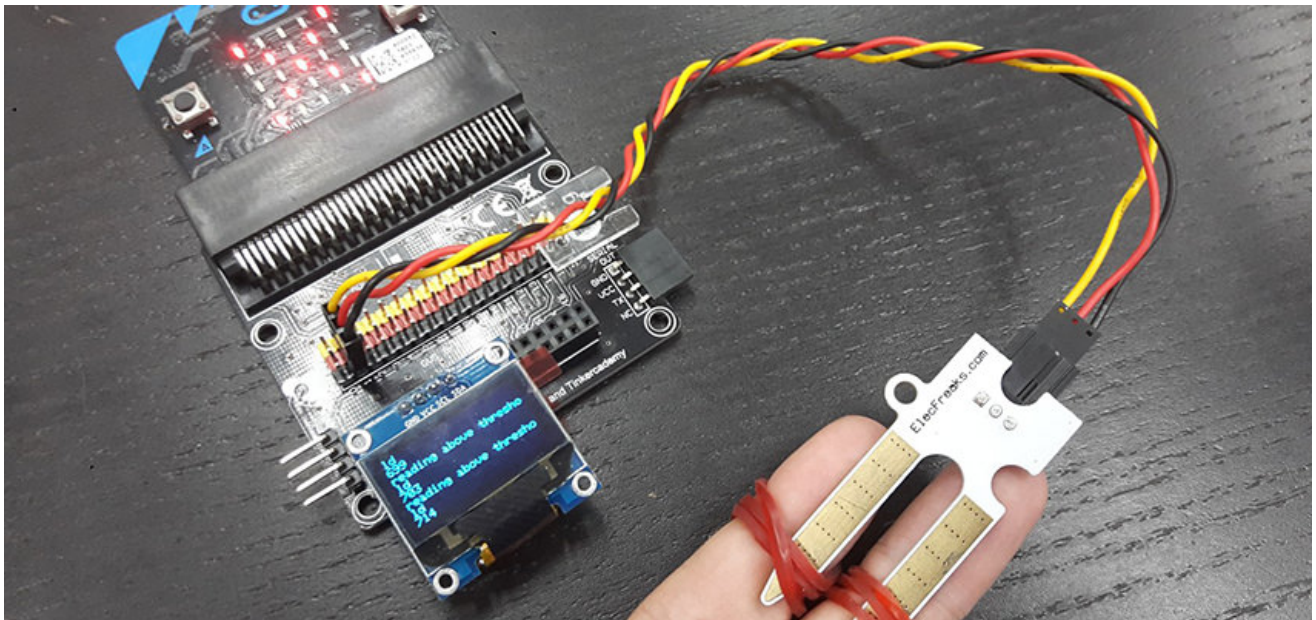
■ Simulator
🧩 Blocks
JS JavaScript
▼
🔗 Edit





Now save these code into your micro:bit and have a try!
Succeed! You now have your very own Micro:bit motion detector!

11. case 09 Lie Detector



With this machine, the truth will never escape you! As long as you can make your subject hold a moisture sensor for a while.

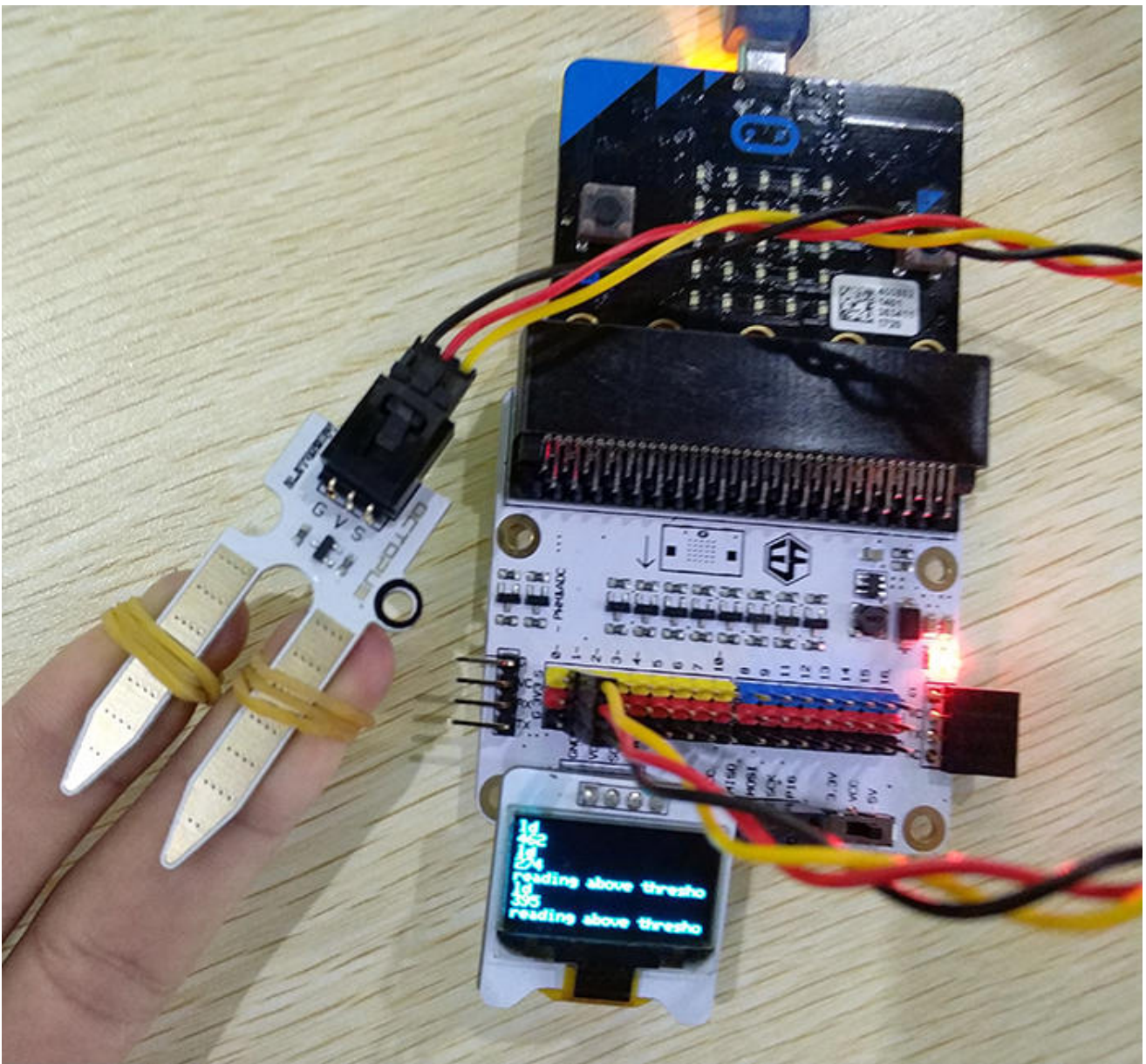
11.1. Step 0 – Pre Build Overview

In this project, we will create a simple lie detector machine, which works by measuring the electrical conductivity of our skin. Upon feeling nervous, our skin's electrical conductivity will increase, and the moisture sensor can pick up on that. This allows us to determine if a person is telling the truth or not.

11.2. Material Needed

- 1 x BBC micro:bit
- 1 x Micro USB cable
- 1 x Breakout board
- 1 x Octopus OLED
- 1 x Moisture Sensor

Tips: If you want all components above, you may need ElecFreaks Micro:bit Tinker Kit

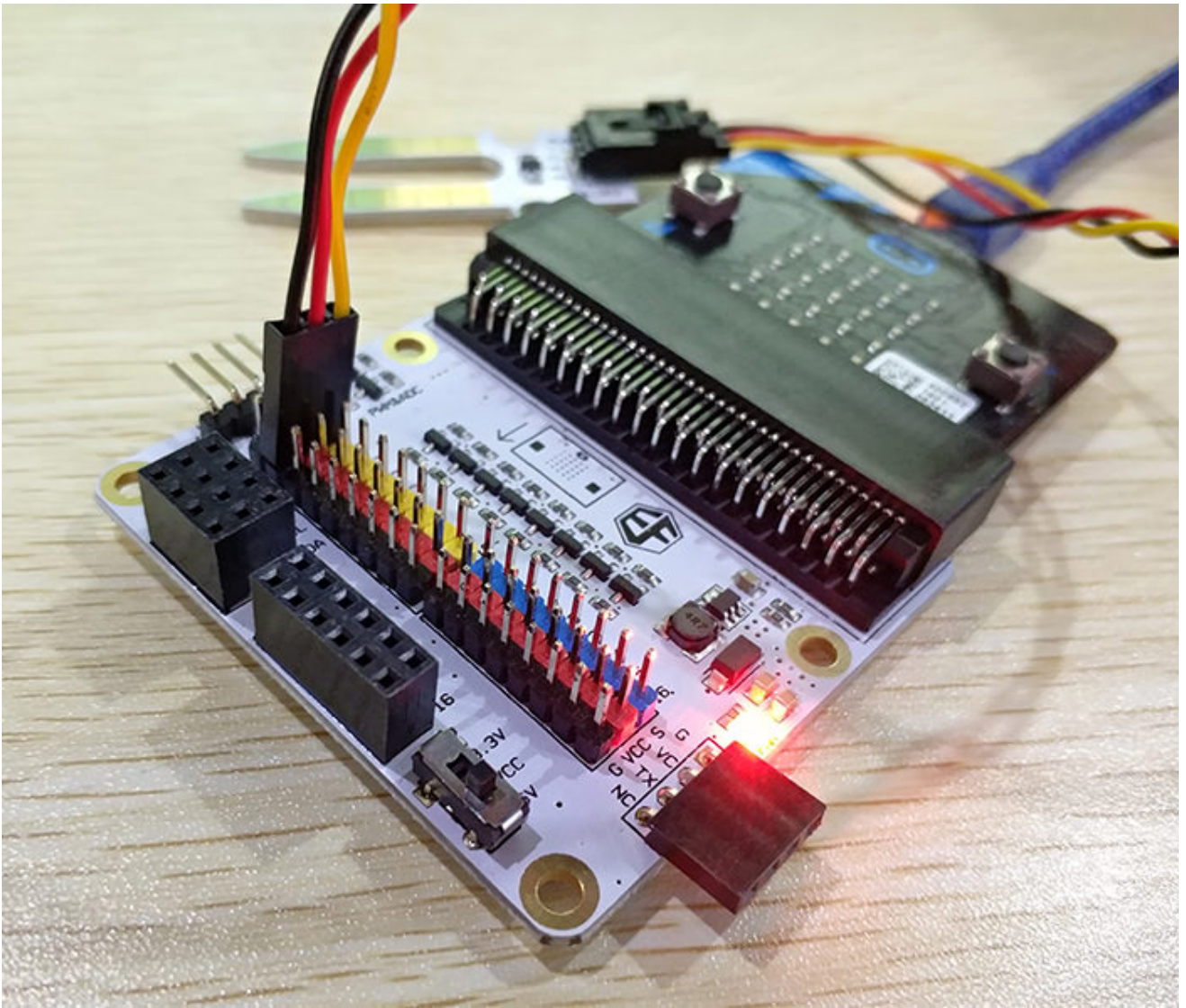


11.3. Goals

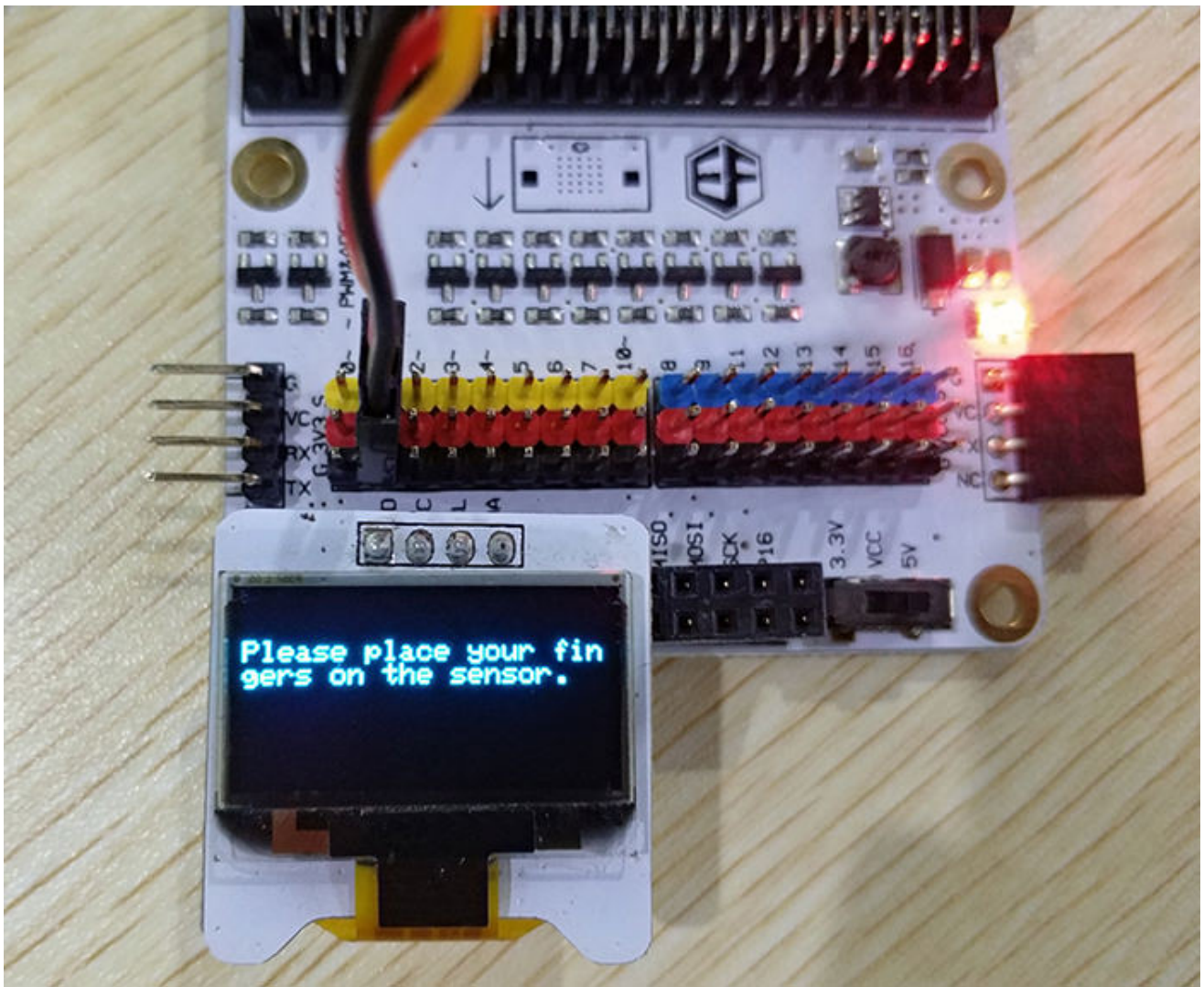
- Get to know the Octopus LED and Soil Moisture Sensor
- Learn basic statistics
- Make something using the moisture sensor

Step 1 Components

First of all, plug in the soil moisture sensor. Match the colors and note down what pin you plug for it will influence later procedures.

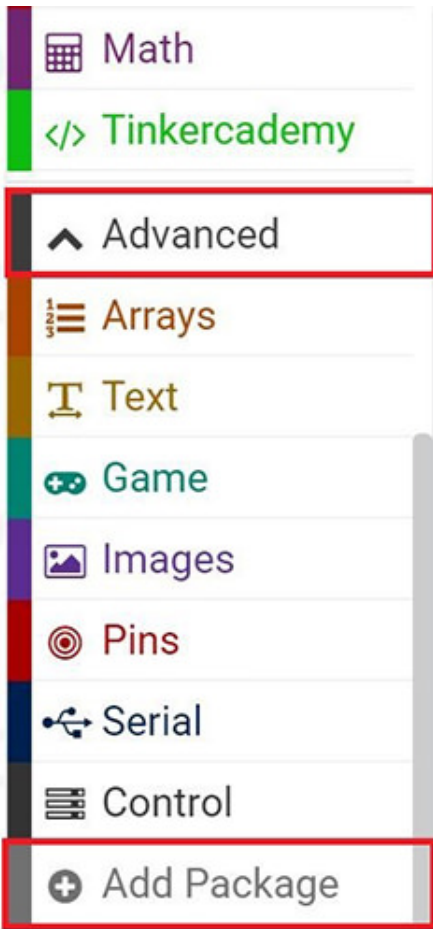
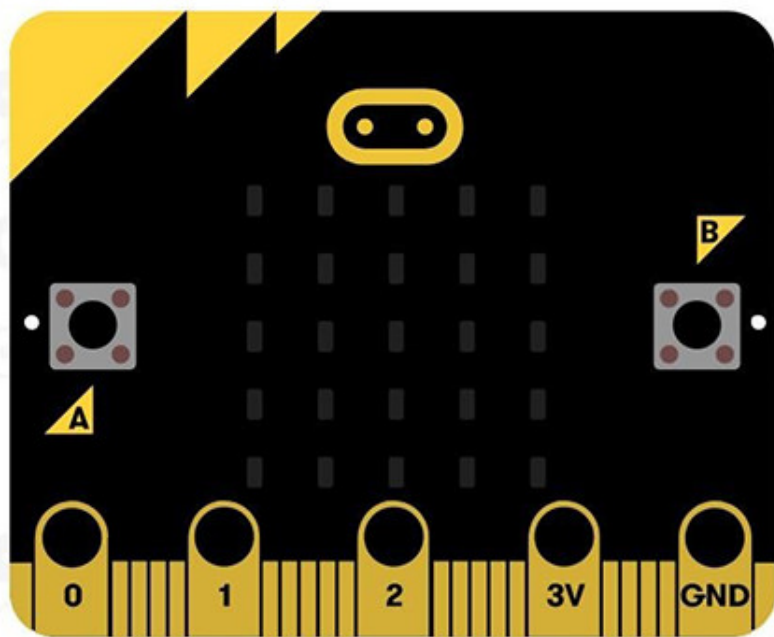


Next, plug in the Octopus LED. Any of the three slots should do.

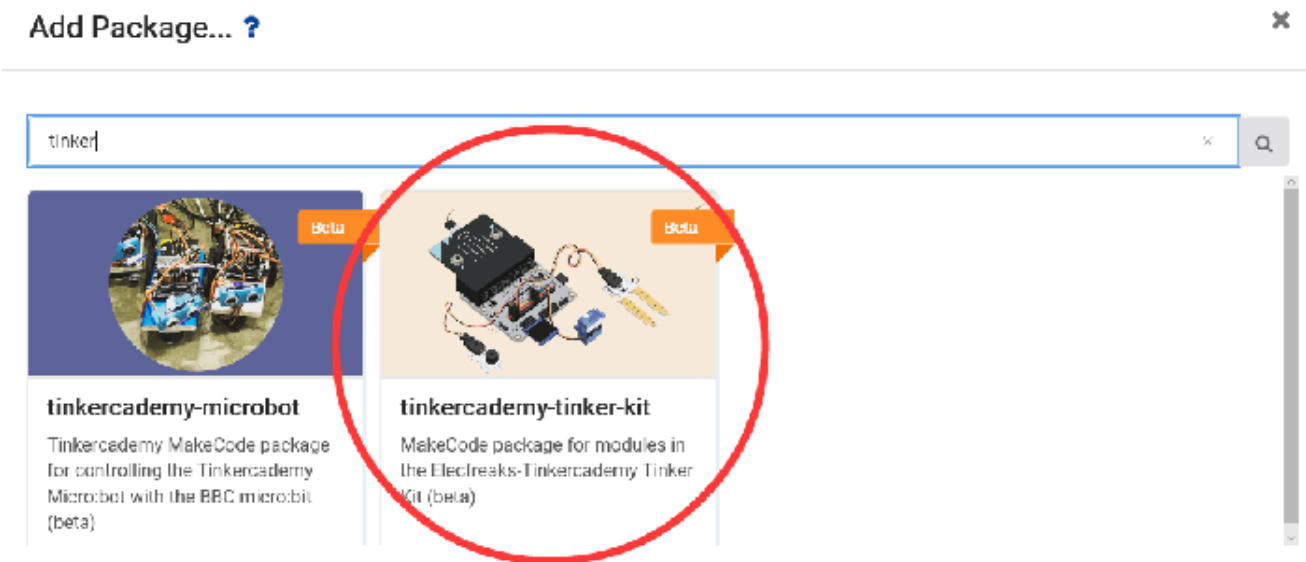


Step 2 Pre-Coding

We'll need to add a package of code to be able to use our kit components. Click on "Advanced" in the Code drawer to see more code sections and look at the bottom of the Code Drawer for "Add Package".



This will open a dialog box. In “Add Package” text field search tinker kit.



Note: If you get a warning telling you some packages will be removed because of incompatibility issues, either follow the prompts or create a new project in the Projects file menu.

Step 3 Coding

First of all, initialize the OLED using blocks in the OLED section as shown in the picture.

```
on start
  initialize OLED with height 64 width 128
  show string "Please place your fingers on the sensor."
  pause (ms) 3000
```

This part of the code allows the soil moisture sensor to measure and record down the electrical conductivity between the two fingers every few seconds for about 45 seconds. Then, it calculates the average. This is the “calm” value when the user has not told any lies.

```
set list to create empty array
repeat 30 times
  do
    list add value analog read pin P1 to end
    change elec by analog read pin P1
    pause (ms) 1500
  end
set average to elec divided by 30
show string "average is:"
show number elec divided by 30
```

This part of the code calculates the standard deviation of the readings obtained in that 45 seconds. The standard deviation indicates how different the readings were. A larger standard deviation means more variance in the readings. The “Math.sqrt” block square roots the given value and was added in Javascript.

```

for element value3 of list
do
  change stddev by (value3 - average) x (value3 - average)
  show string "stddev is:"
  show number Math.sqrt(stddev / 30)
  set std to Math.sqrt(stddev / 30)

```

After the initial readings have been made and recorded, the moisture sensor now measures the average electrical conductivity over 2.5 seconds. If it is higher than the average added to the standard deviation, we can conclude that the user has an abnormally high electrical conductivity and is thus lying. Then, the LED screen would show an "X" shape.

```

while true
do
  set readings to 0
  repeat 5 times
  do
    change readings by analog read pin P1
    pause (ms) 500
  if
    readings ÷ 5 ≥ average + std
  then
    show string "reading above threshold"
    show number readings ÷ 5
    show icon [LED icon]
  else
    show number readings ÷ 5

```

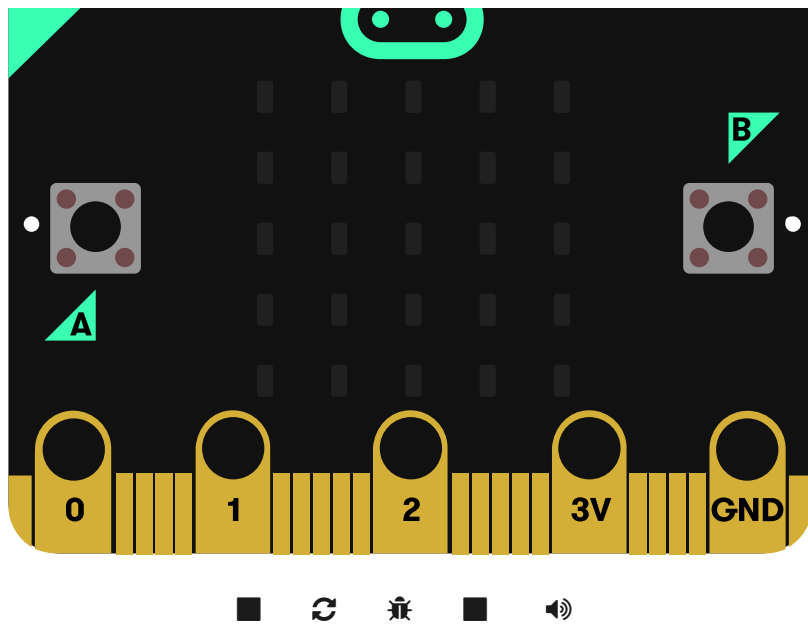
To relieve your tired fingers, you can download the code below.

https://makecode.microbit.org/_fadAyyh27Eo3

Or you can download from the page below.

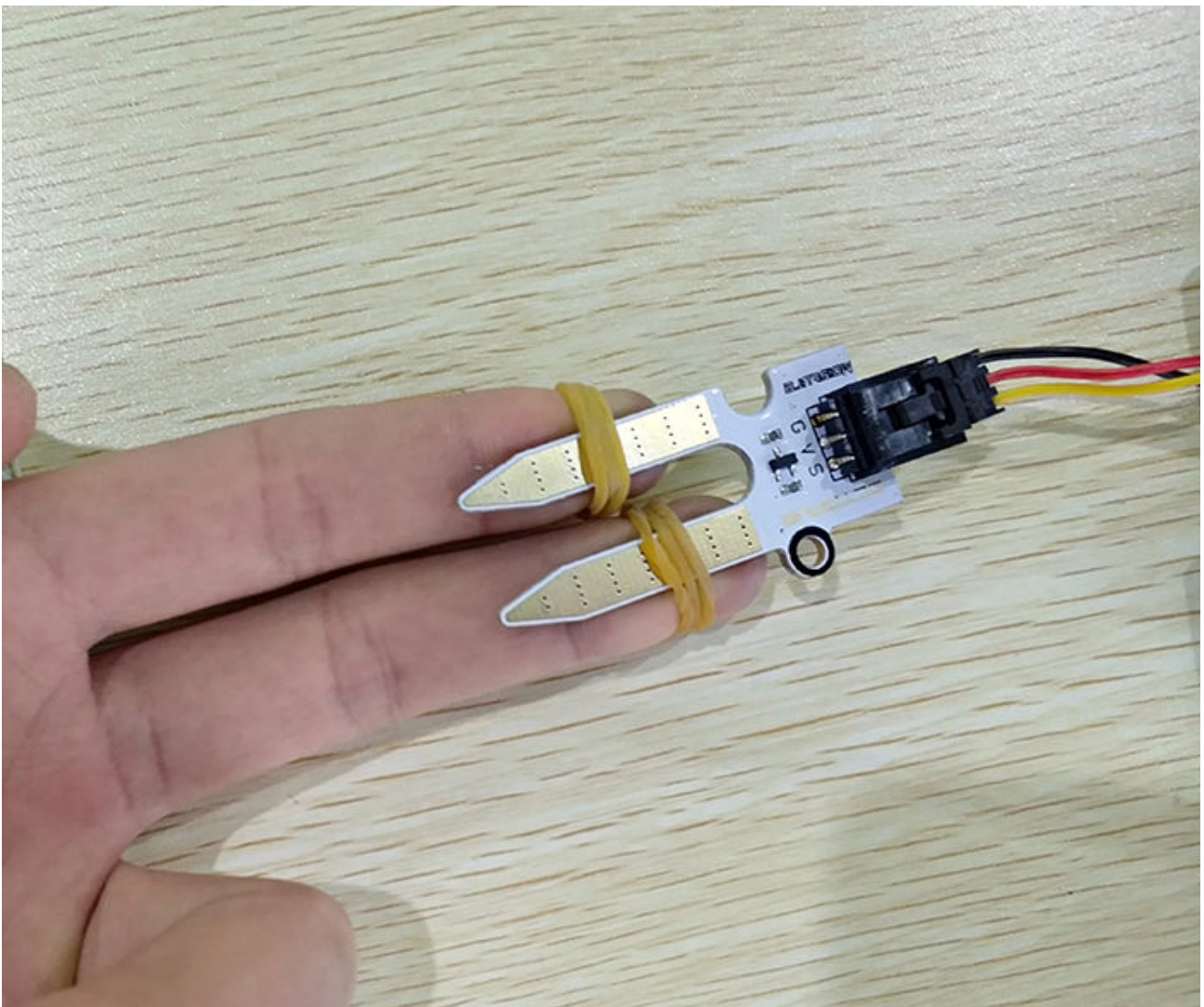
■ Simulator
🧩 Blocks
JS JavaScript
▼
🔗 Edit



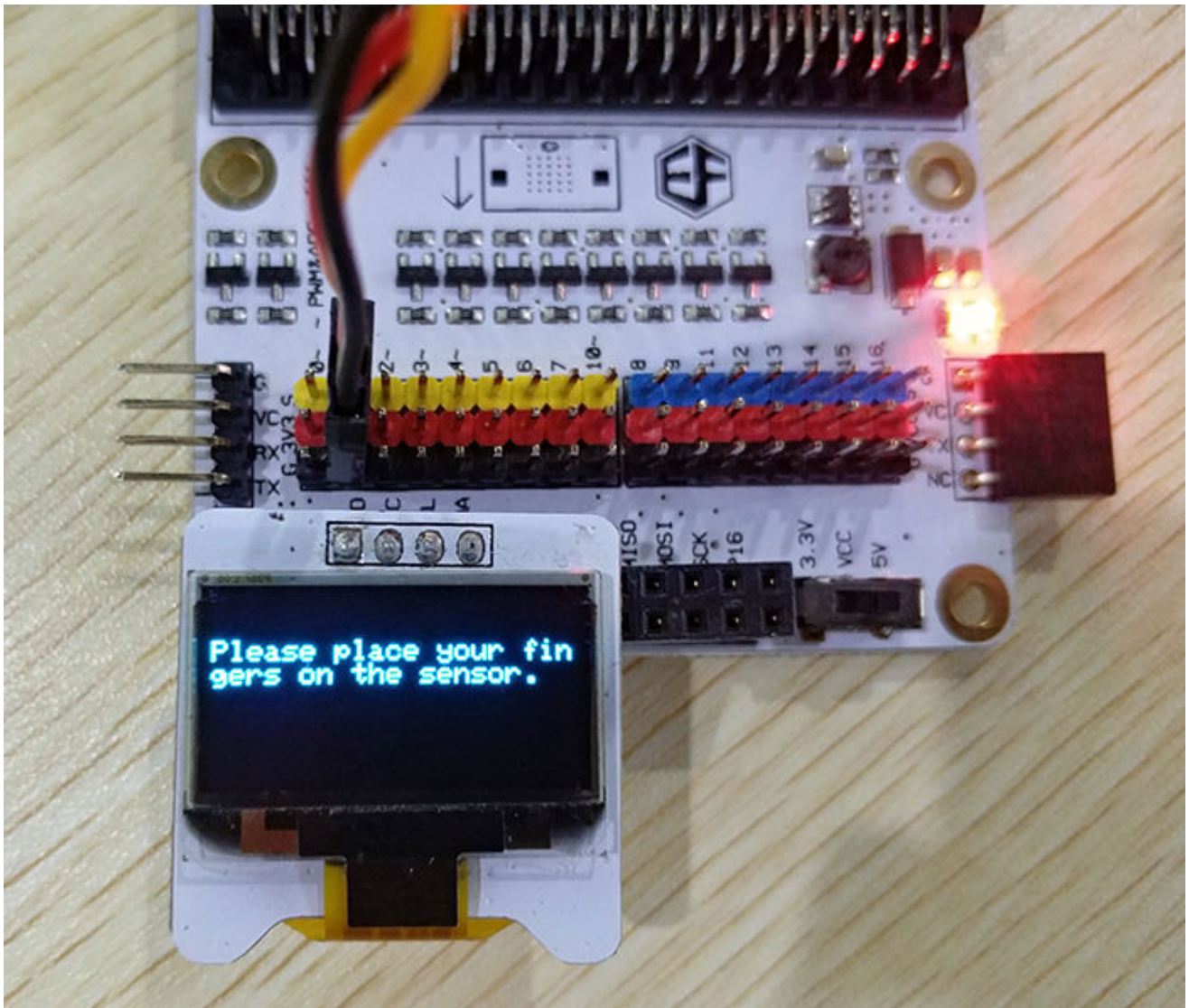


Step 4 Using It

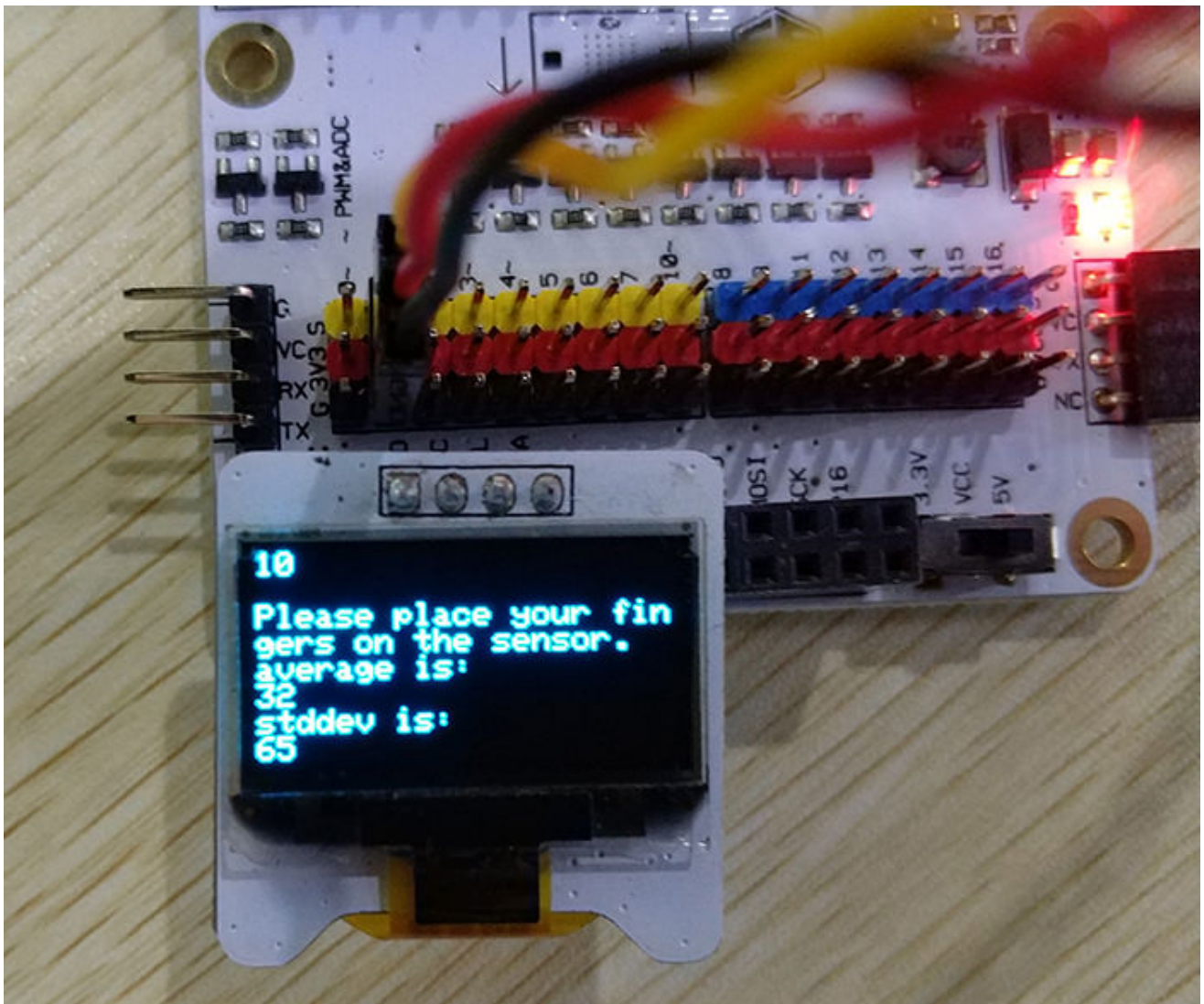
First of all, you will have to attach each prong of the soil moisture sensor to one of your fingers. Personally, I found that using rubber bands was a simple and effective way to do so. You can experiment with different methods, such as using crocodile clips or tape.



Now, turn on the device. The device will record the electrical conductivity of your skin under calm circumstances. Then, it will give the average value and its standard deviation.



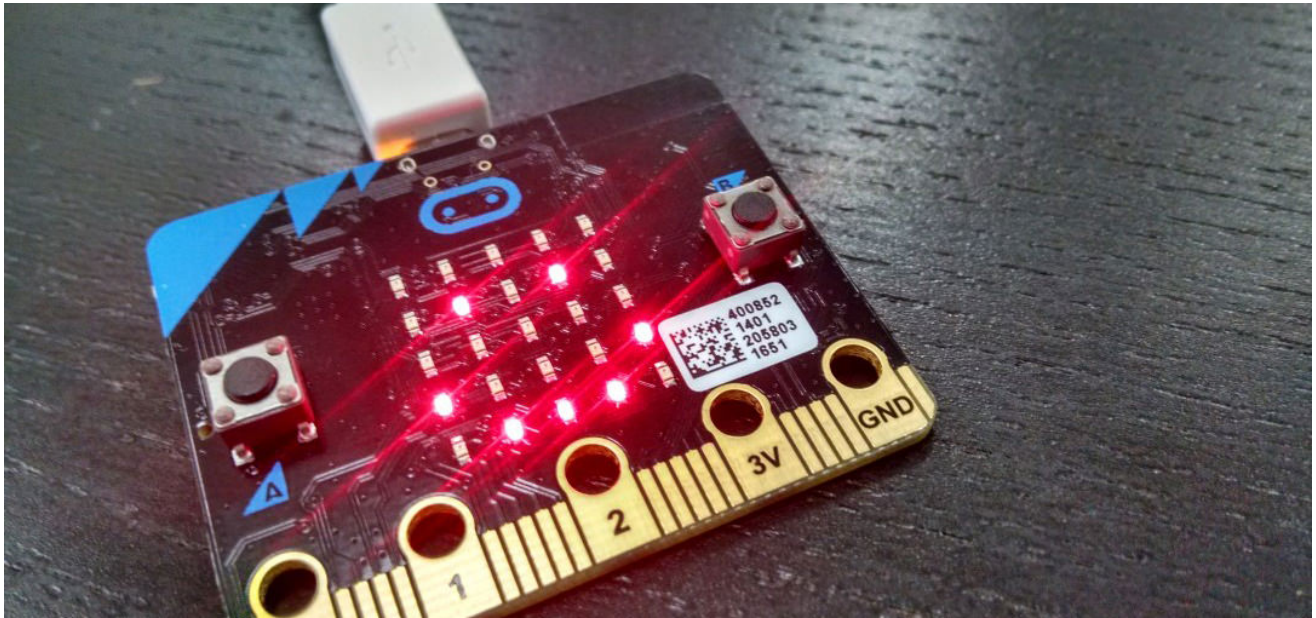
After the initial readings have been made, ask again! If the person has lied, he will become nervous and the device can pick up on that, resulting in a cross being displayed.



Step 5 Success!

Voila! Now you can test lies with this machine easily.

12. case 10 PADDLEBALLSUPERSMASHEM



Learn to program a simple yet fun game on a 5 by 5 display, using JavaScript! PADDLEBALLSUPERSMASHEM may bear unintentional similarities to other, more graphical, games.

12.1. Step 0 – Pre Build Overview

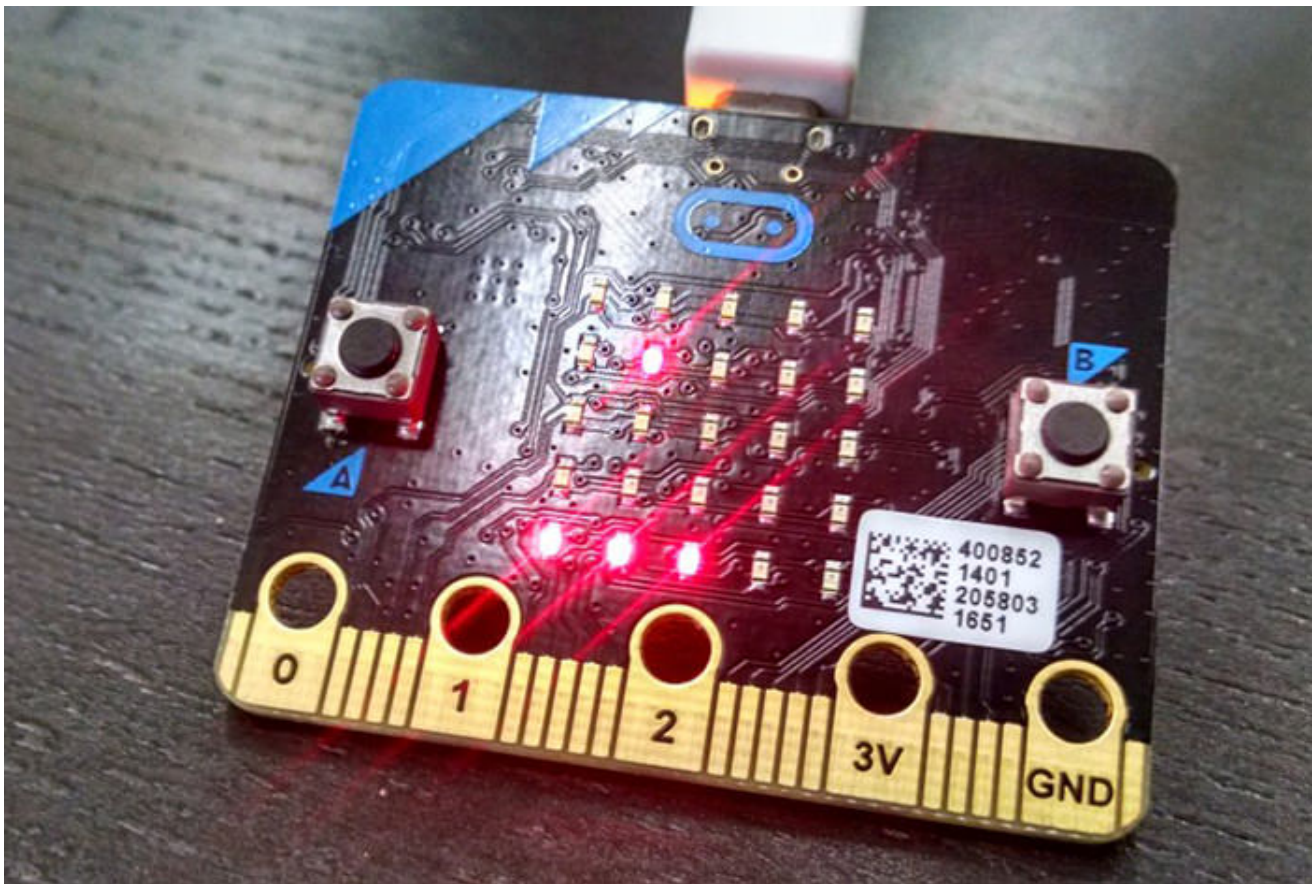
In this project, we will create a simple game, in which you bounce a ball against a wall. If you miss, you die. Too bad. For those of you who appreciate a challenge, the game increases in difficulty with each level!

12.2. Goals

- Get to know more about the microbit microcomputer
- Learn how to program a simple game
- Consider all cases

12.3. Material

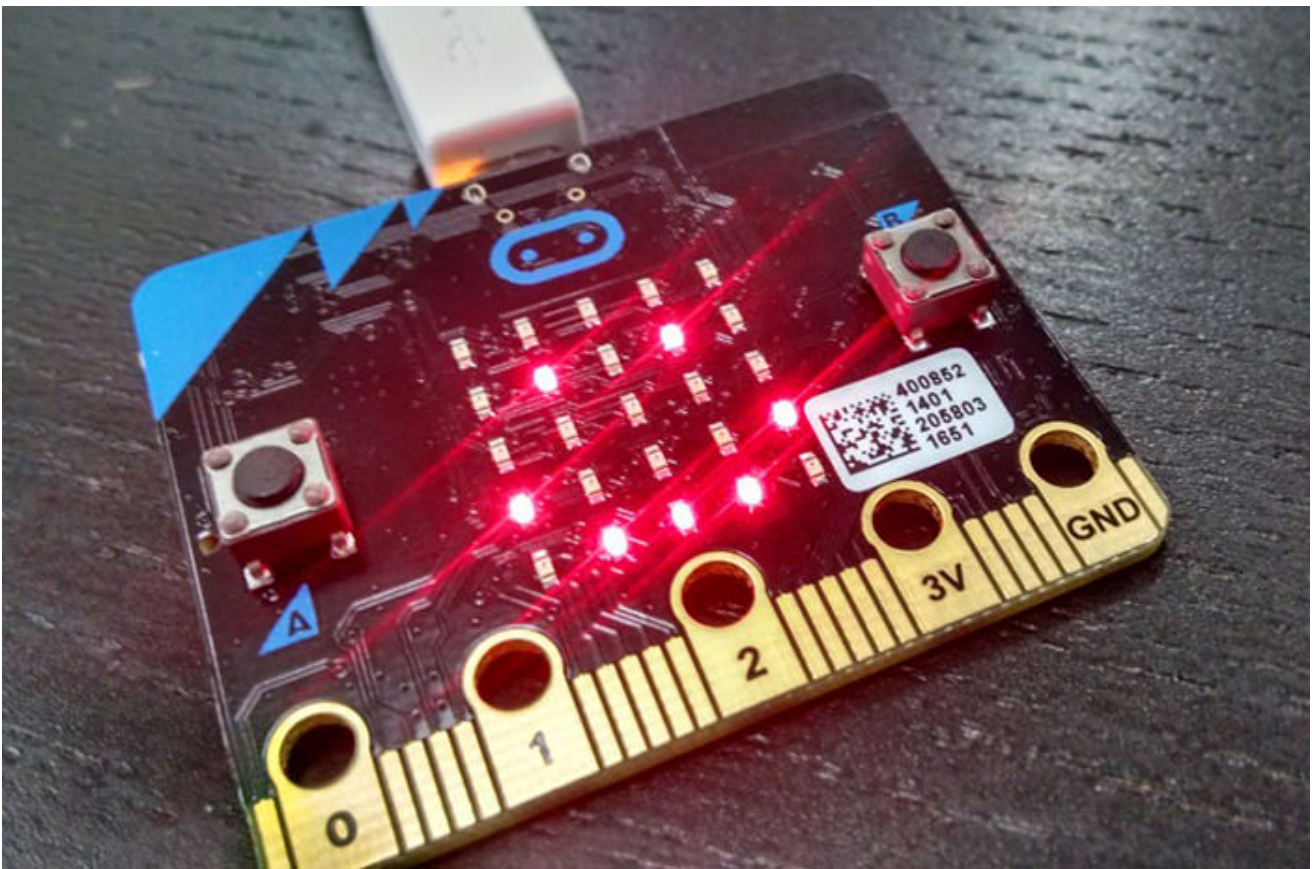
- 1 x BBC micro:bit
- 1 x Micro USB cable



12.4. How to Make

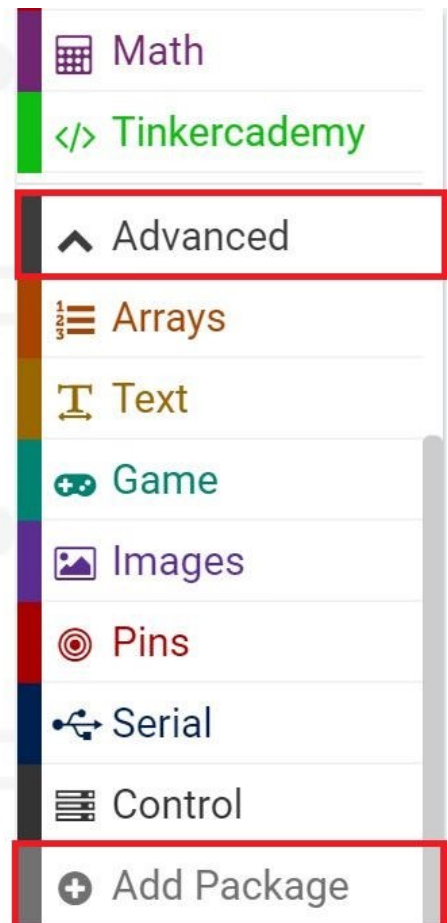
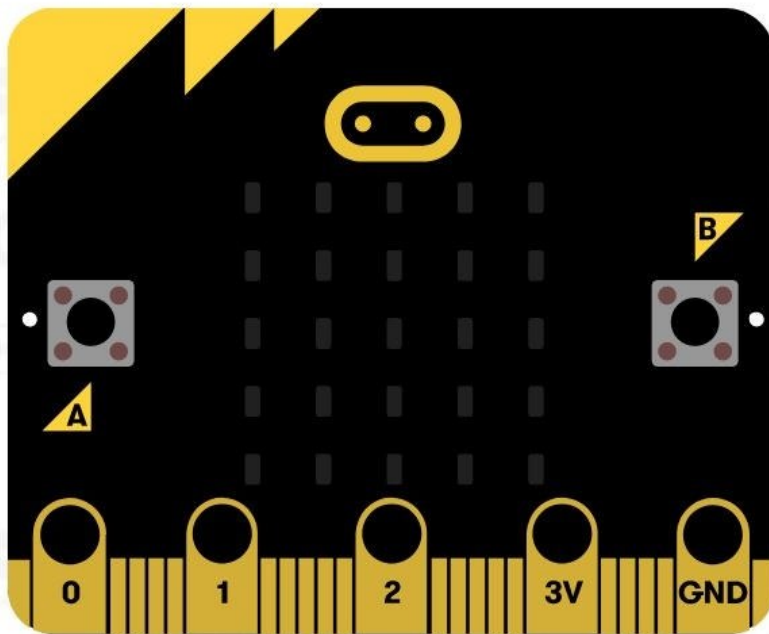
Step 1: Components

First of all, plug the microbit microcomputer into your own computer. No other components are required.

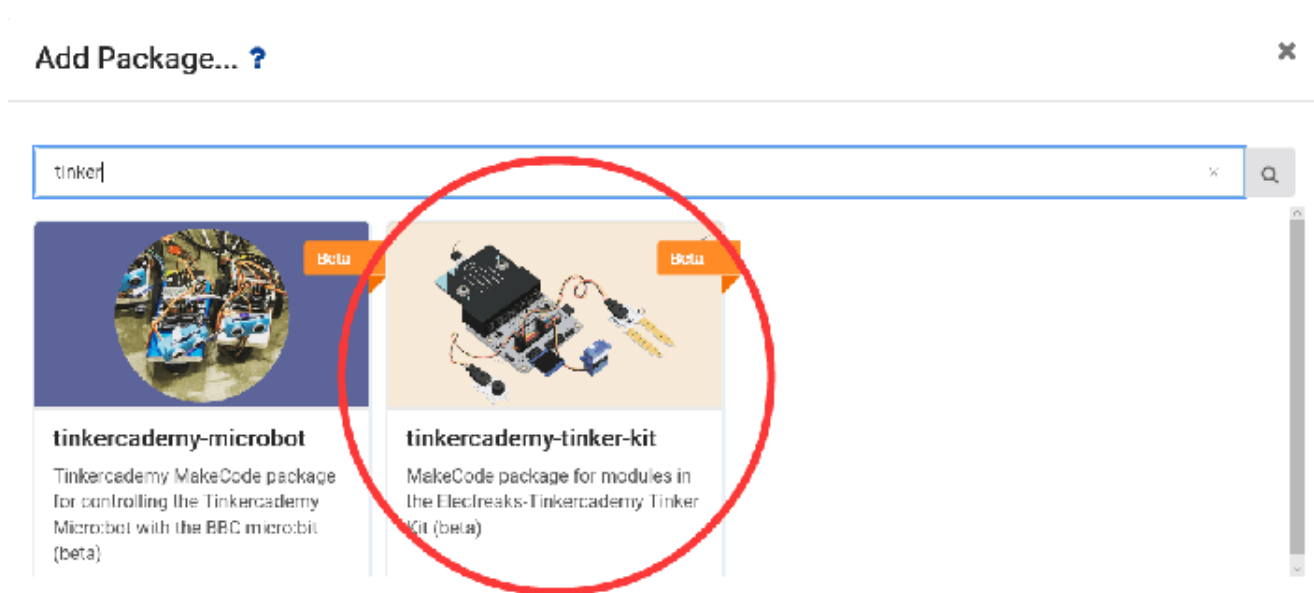


Step 2: Pre-coding

We will add a package of code to enable us to use our kit components. Click on “Advanced” in the Code Drawer to see more code section and look at the bottom of the Code Drawer for “Add Package”.



This will open up a dialog box. Search for “tinker kit” and then click on it to download this package.



Note: If you get a warning telling you some packages will be removed because of incompatibility issues, you should either follow the prompts or create a new project in the Projects file menu.

Step 3: Coding

```
102 let x: number
103 x = 0
104 let y: number
105 y = 0
106 let xb: number
107 xb = 0
108 let yb: number
109 yb = 0
110 let xdir: number
111 xdir = 0
112 let ydir: number
113 ydir = 0
114 let scor: number
115 scor = 0
116 let gam: boolean
117 gam = true
118 let time: number
119 time = 1000
```

First of all, define your variables! We are going to need many variables to store the location, speed and direction of the ball, the length and position of the paddle, and last but not least, your score!

```
1 function board() {
2   for (let i = xb; i < (xb + yb); i++) {
3     led.plot(i, 4)
4   }
5 }
6 function left() {
7   xb += 0 - 1
8   basic.clearScreen()
9   board()
10  ball()
11 }
12 function right() {
13   xb += 1
14   basic.clearScreen()
15   board()
16   ball()
17 }
```

Next, we will program the functions that control the paddle. `xb` represents the position of the first pixel of the paddle from the left, and `yb` represents the length of the paddle. The `left` and `right` functions control `xb` and shift the paddle, and the `board` function prints the paddle on the screen.

```

60 function moveball() {
61     x += xdir
62     y += ydir
63     basic.clearScreen()
64     ball()
65     board()
66     if (y == 0 && (x == 0 || x == 4)) {
67         |   corners()
68     } else if (x == 0 && y > 0 && y < 3) {
69         |   leftside()
70     } else if (x == 4 && y > 0 && y < 3) {
71         |   rightside()
72     } else if (y == 0) {
73         |   topside()
74     } else if (y == 3) {
75         |   checkhit()
76     } else if (y == 4) {
77         |   gam = false
78         |   basic.clearScreen()
79         |   basic.pause(1000)
80         |   basic.showNumber(scor)
81         |   basic.pause(1000)
82         |   basic.clearScreen()
83         |   if (scor < 12) {
84         |       |   basic.showIcon(IconNames.Sad)
85         |       } else {
86         |           |   basic.showIcon(IconNames.Happy)
87         |           }
88     }

```

Next, we include the function that controls when the ball moves. At the beginning, the ball moves every second but as you advance, the ball moves at shorter and shorter intervals! How exciting!

```

18 function ball() {
19     led.plot(x, y)
20 }
21 function topside() {
22     ydir = 1
23     if (x == 0) {
24         xdir = Math.random(2)
25     } else if (x == 4) {
26         xdir = Math.random(2) - 1
27     } else {
28         xdir = Math.random(3) - 1
29     }
30 }
31 function leftside() {
32     xdir = 1
33 }
34 function rightside() {
35     xdir = -1
36 }
37 function corners() {
38     xdir = 0 - xdir
39     ydir = 0 - ydir
40 }

```

We now program the functions that control how the ball interacts with its surroundings. When the ball hits the side, its horizontal movement is reversed but its vertical movement remains the same. When the ball hits the ceiling, it can rebound in any direction, to make the game more fun.

```

41 function checkhit() {
42   if (xb - 1 < x && xb + yb > x) {
43     scor += 1
44     ydir = -1
45     if (x == 0) {
46       xdir = Math.random(2)
47     } else if (x == 4) {
48       xdir = Math.random(2) - 1
49     } else {
50       xdir = Math.random(3) - 1
51     }
52     if (scor > 1 && scor < 4) {
53       yb += 0 - 1
54     }
55     if (scor > 5 && scor < 12) {
56       time += 0 - 150
57     }
58   }
59 }

```

Most importantly, we need to see if the ball hits the paddle. If it misses, you lose, displaying your score! If it doesn't miss, the ball will also rebound in a random direction, and the difficulty of the game will increase.

```

90 input.onButtonPressed(Button.B, () => {
91   right()
92 })
93 input.onButtonPressed(Button.A, () => {
94   left()
95 })
96 basic.forever(() => {
97   while (gam == true) {
98     basic.pause(time)
99     moveball()
100  }
101 })

```

Lastly, we have a for loop which acts as a clock so that the ball keeps moving. Also, we have the `onButtonPressed()` functions that move the paddle.

Save your tired fingers and download the code from the link below.

<https://makecode.microbit.org/63331-03858-42547-81536>

Or you can download from this page.

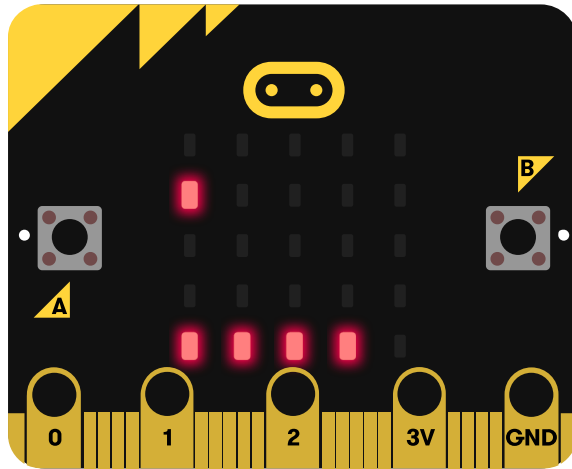
■ Simulator

🧱 Blocks

JS JavaScript



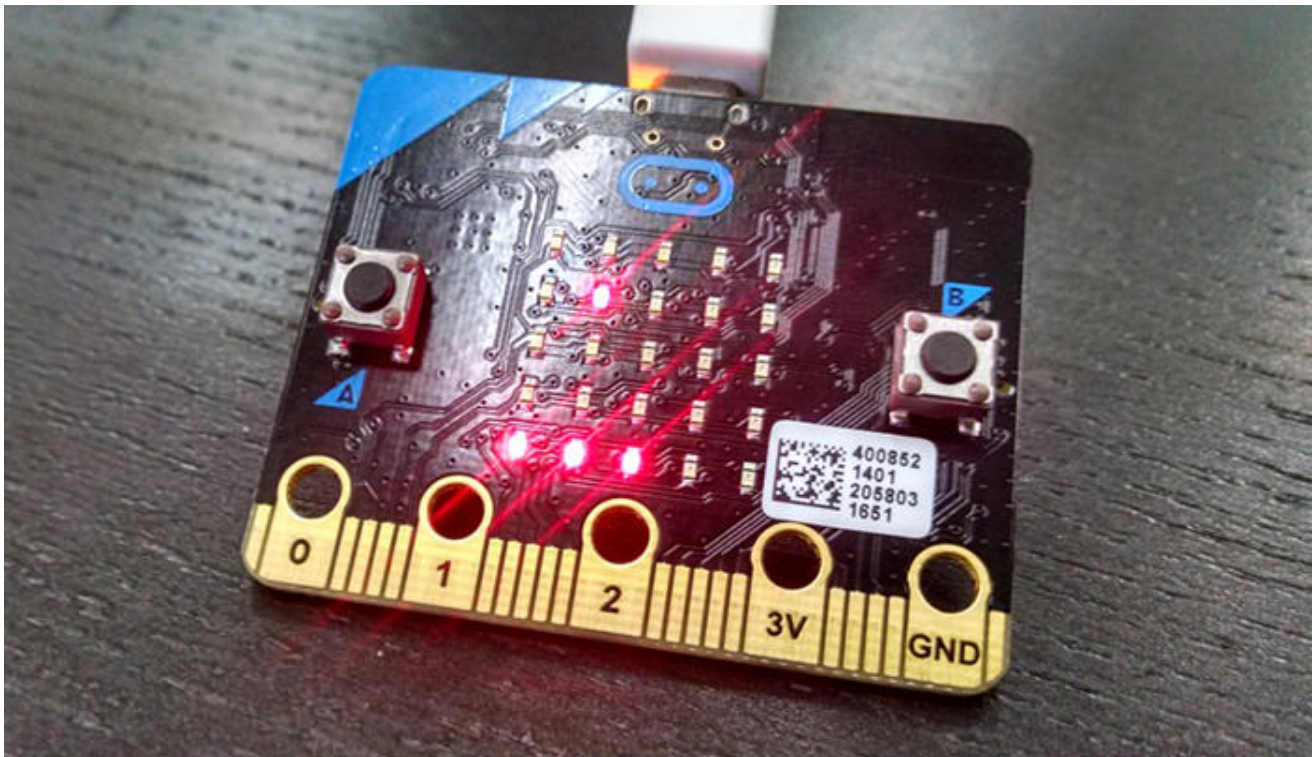
🔗 Edit



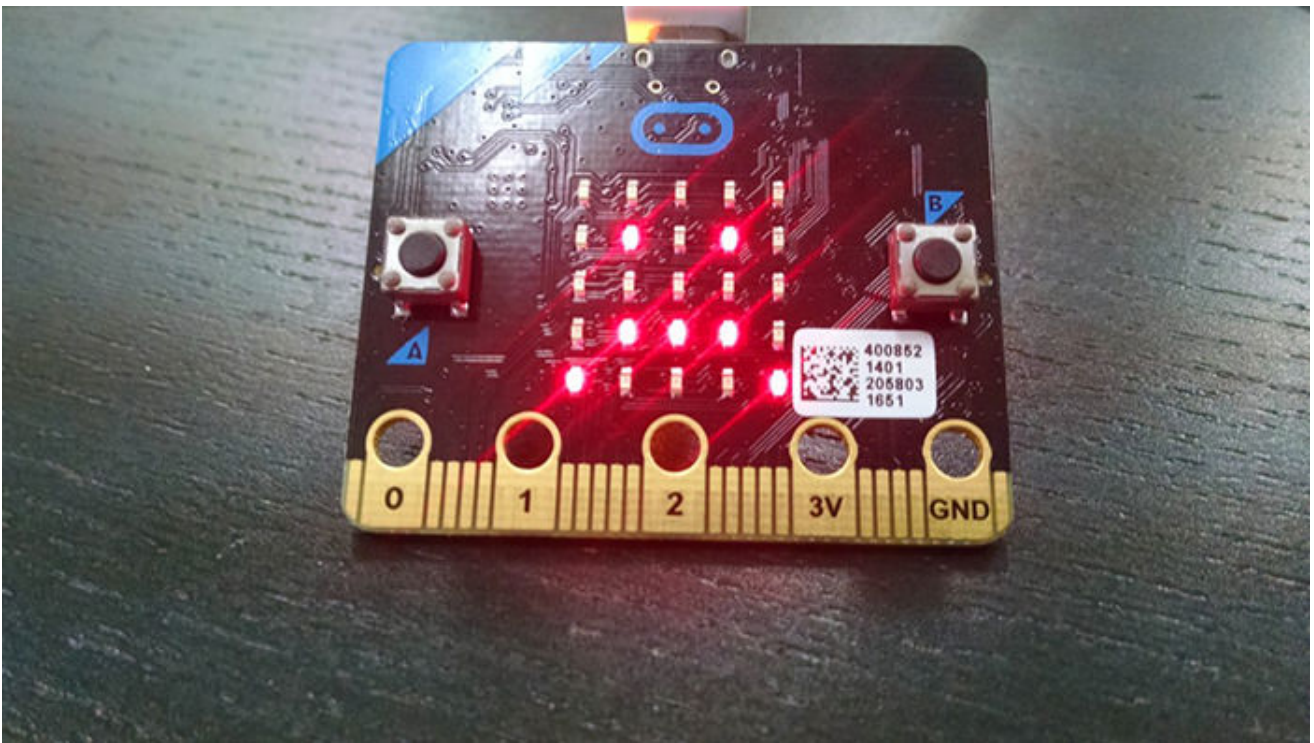
—



Step 4: Using It



Just connect the microcontroller to your computer, and run the program! Easy!

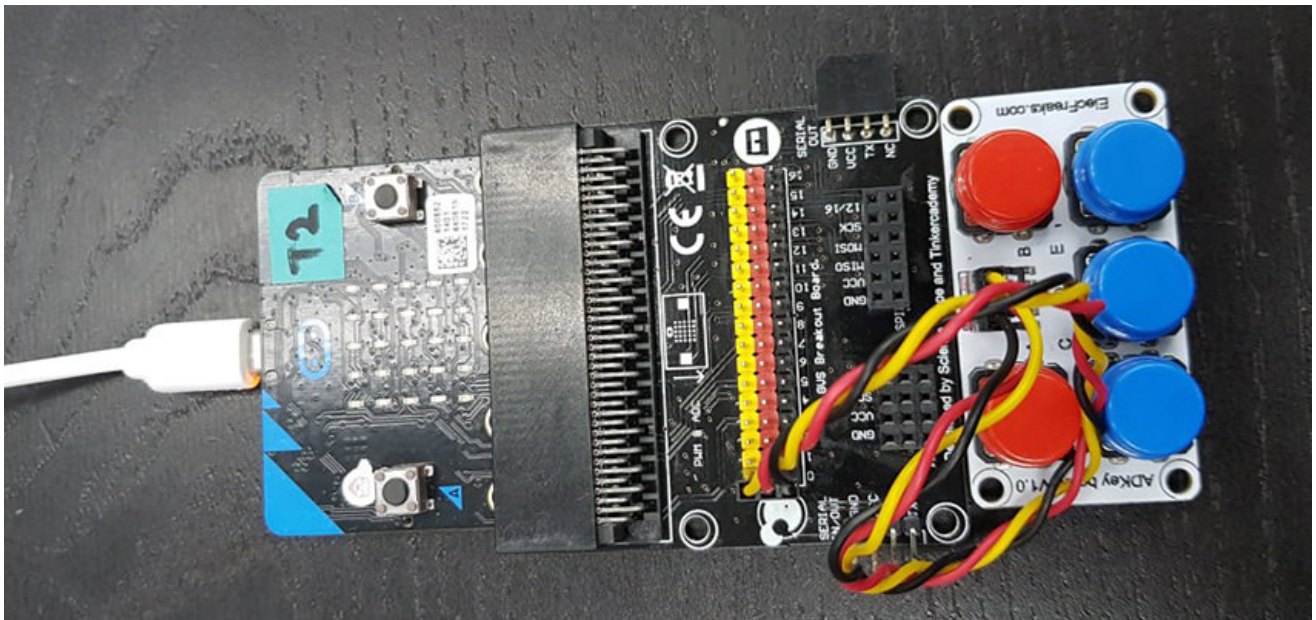


If you score more than 12 points, you will be rewarded with a smiley face! Otherwise, the program may not be very pleased...

Step 5: Success!

Voila! You have now programmed PADDLEBALLSUPERSMASHEM on a 5 by 5 display. You should be proud of yourself.

13. case 11 Avoid Asteroids



Make your own little arcade game on the Micro:bit, and admire its 5 by 5 pixelate glory! This tutorial was contributed by Josh Ho from Raffles Institution.

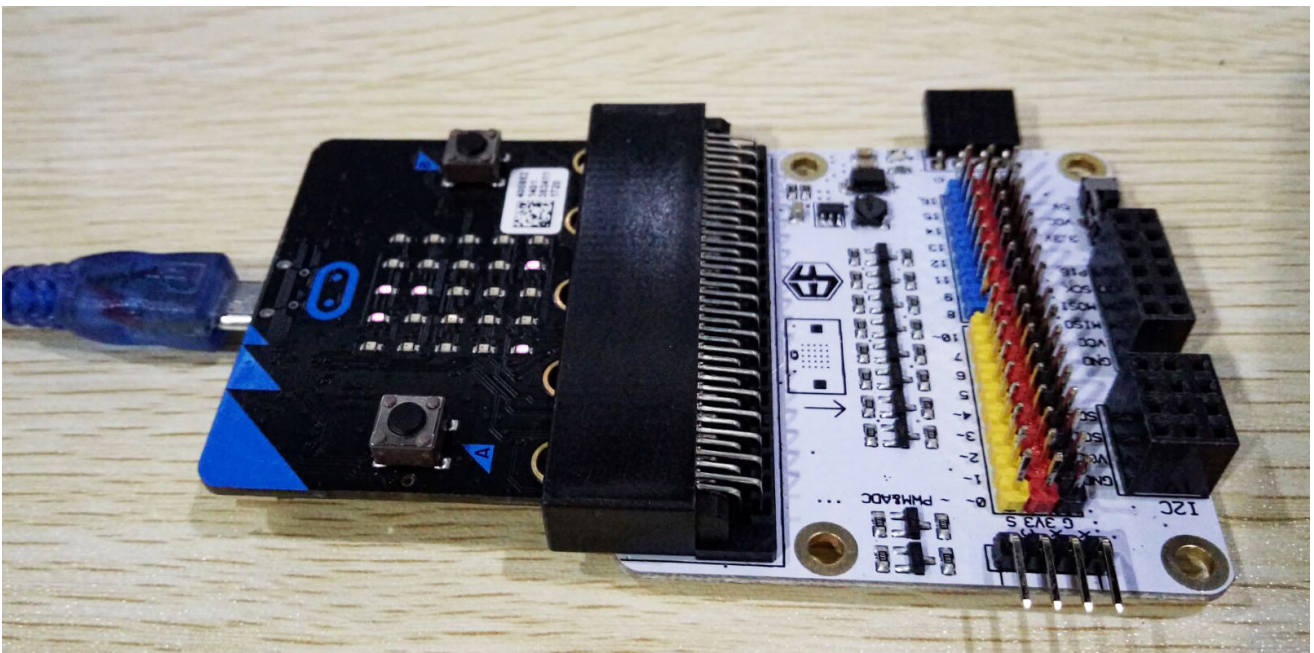
13.1. Step 0 – Pre-build Overview

In this project, we are going to create a Raiden-esque game using Micro:bit and an ADKeyboard only . The aim of this game is to dodge the incoming projectiles, which increase in speed as the game goes on, for as long time as possible. The Micro:bit LED will be our screen and the ADKeyboard will be the controller.

13.2. Materials:

- 1 x BBC Micro:bit
- 1 x Micro USB cable
- 1 x Breakout board
- 1 x ADKeyboard

Tips: If you want all components above, you may need ElecFreaks Micro:bit Tinker Kit

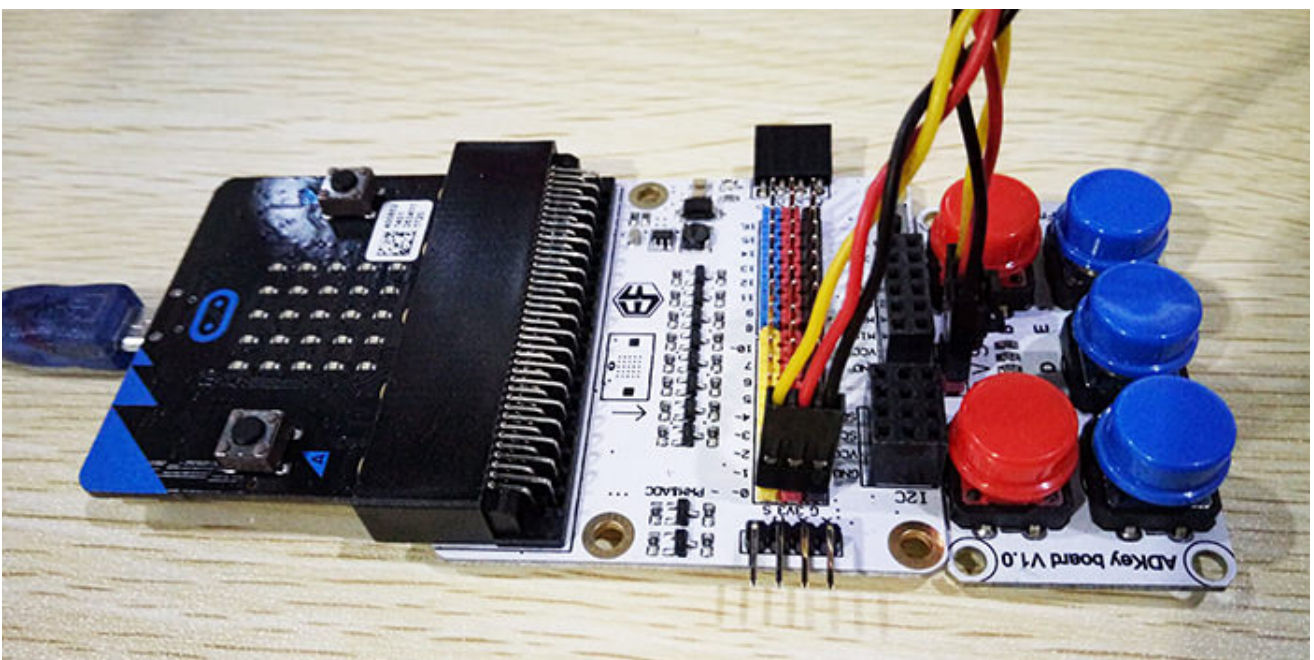


13.3. Goals:

- Make a simple game with Micro:bit.
- Learn advanced programming logic.
- Experiment with sprites.

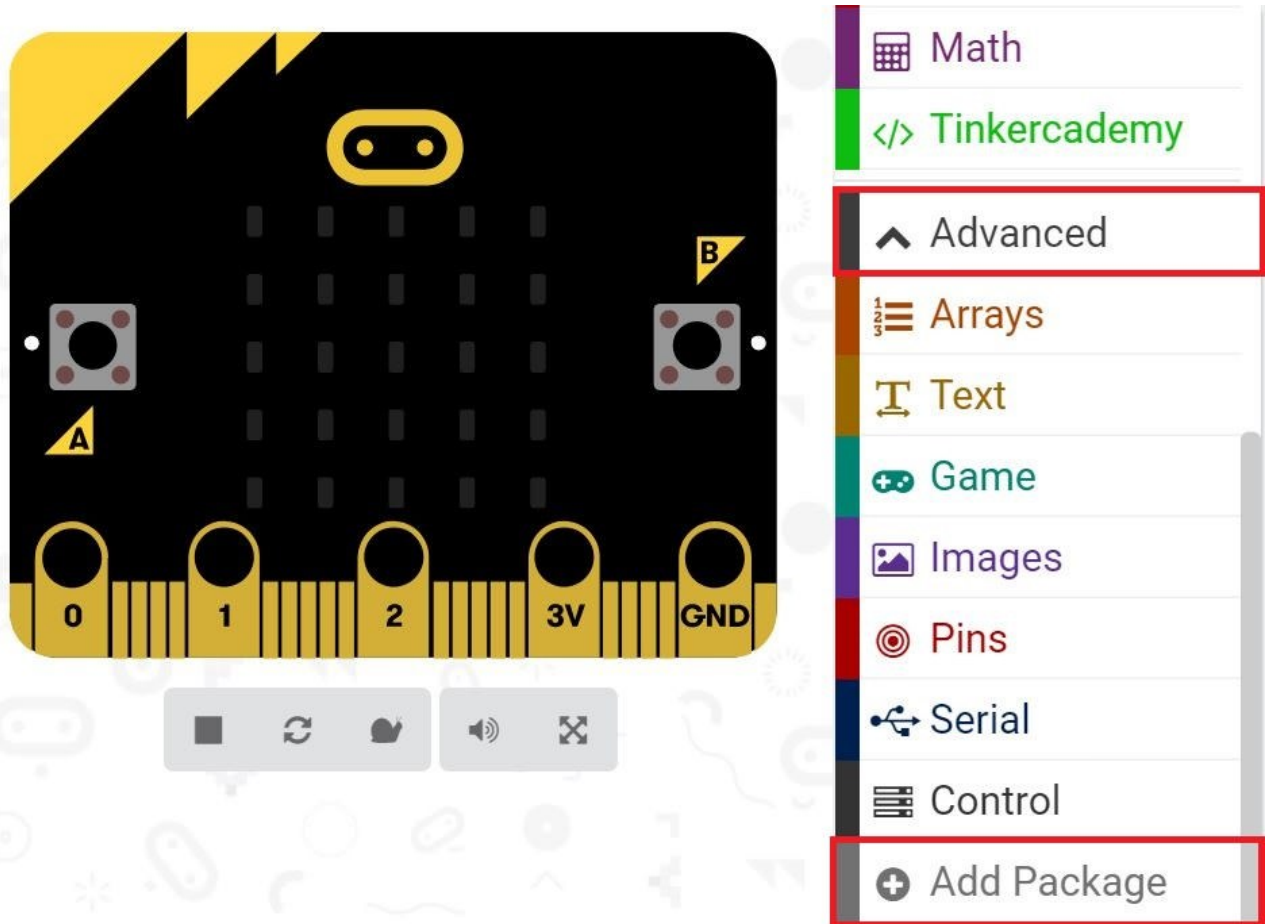
Step 1: Components

The ADKeyboard is the only external component in this project. Just insert the Micro:bit into the breakout board before connecting the ADKeyboard. Make sure that the colors of the wires match the colors of the pins. Quite simple!

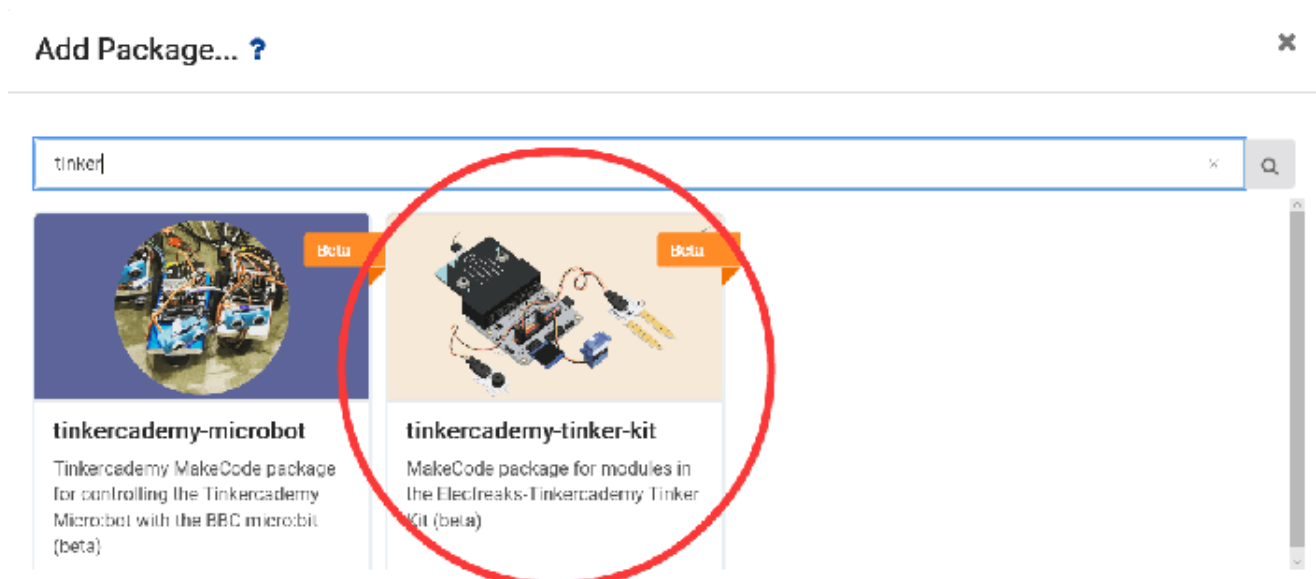


Step 2: Pre-coding

We will add a package of code to enable us to use our kit components. Click on “Advanced” in the Code Drawer to see more code section and look at the bottom of the Code Drawer for “Add Package”.



This will open up a dialog box. Search for “tinker kit” and then click on it to download this package.



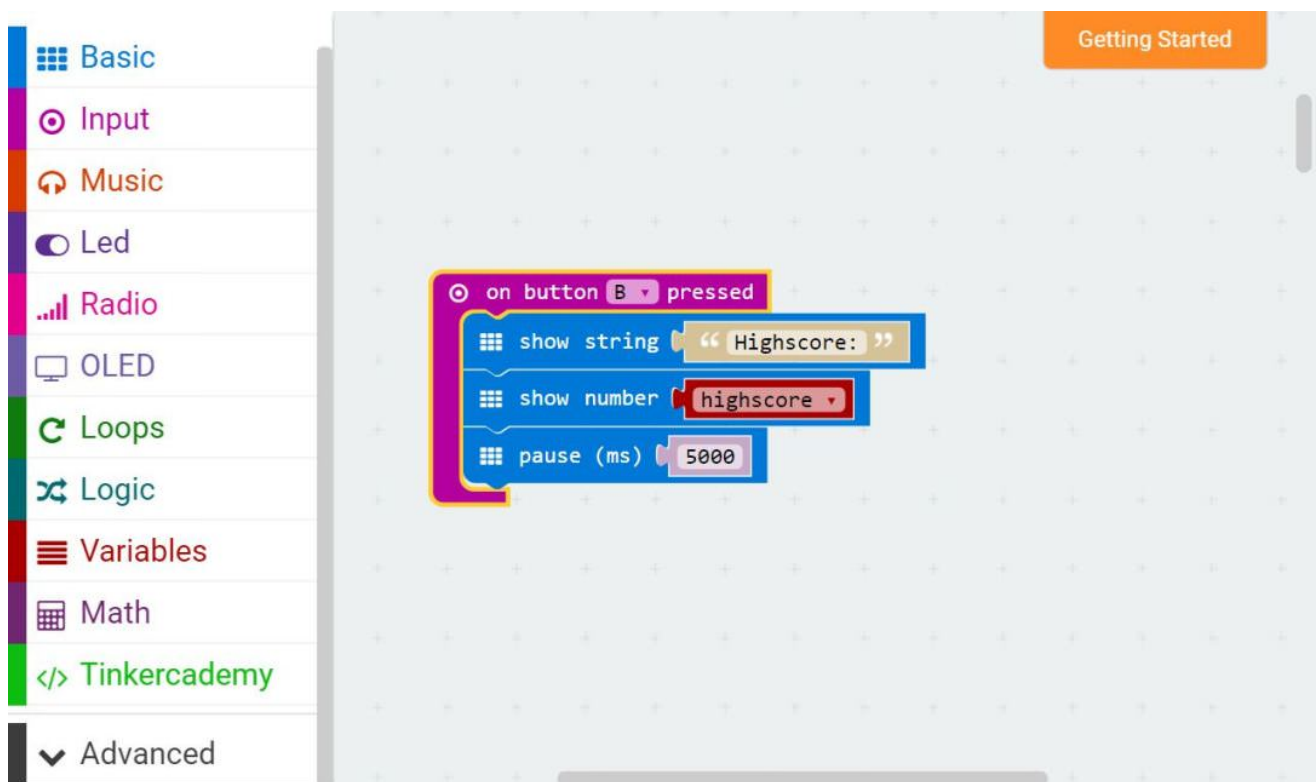
Note: If you get a warning telling you some packages will be removed because of incompatibility issues, you should either follow the prompts or create a new project in the Projects file menu.

Step 3: Coding

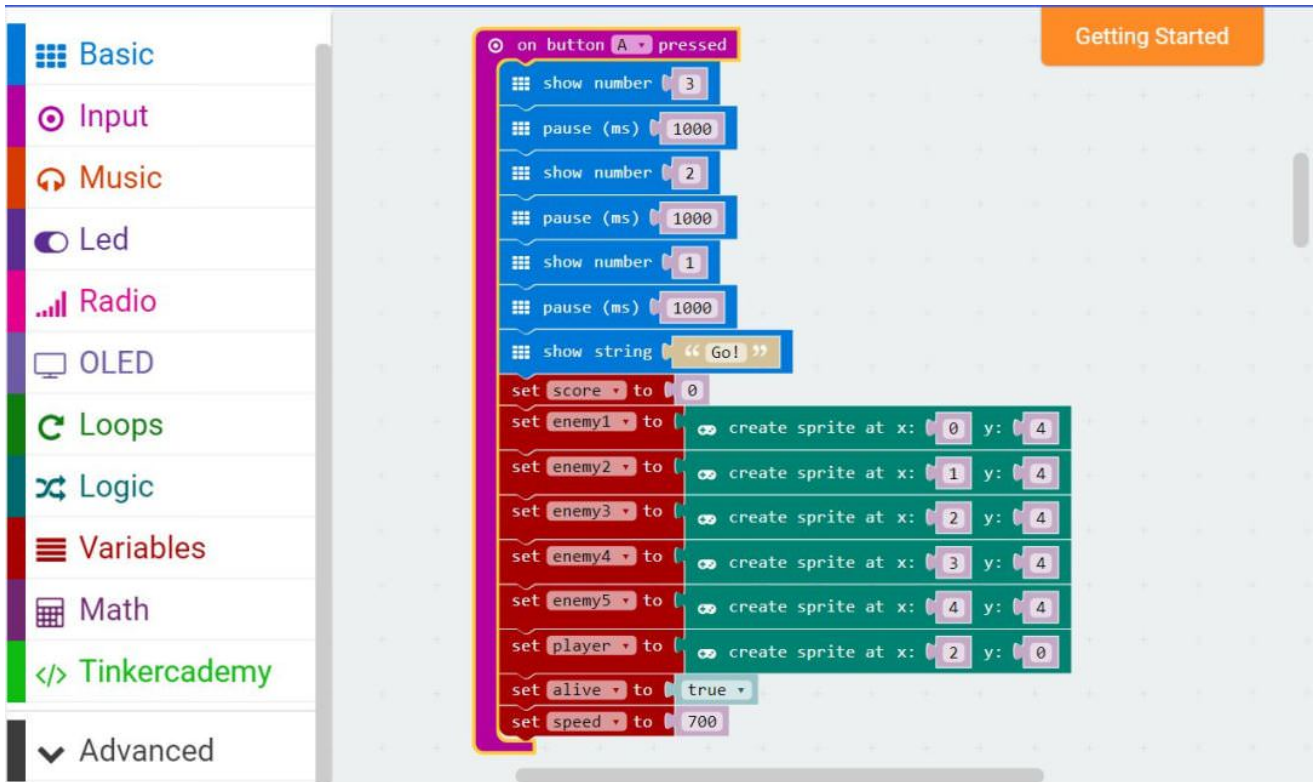
Variables allow us to store data in the program. We will use it to store our scores in the game.



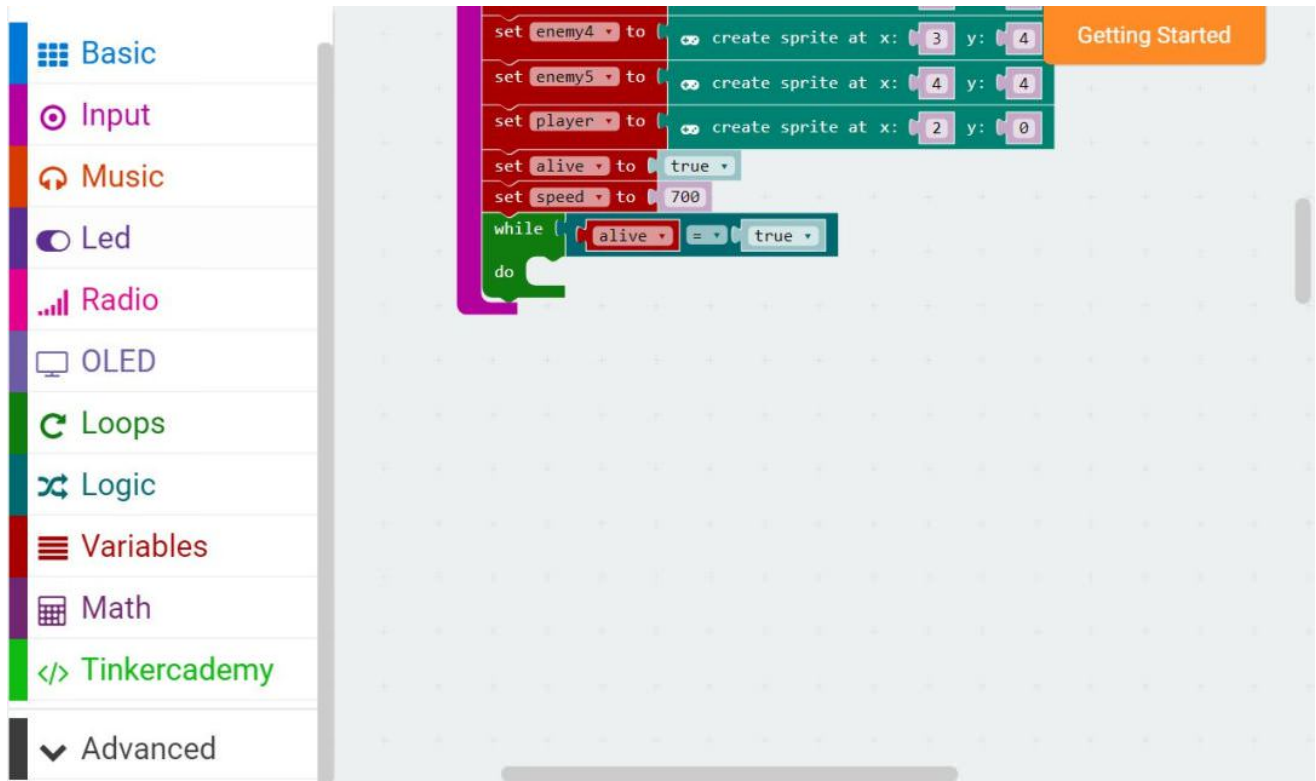
We will use a button on the Micro:bit (button B) to show the high score when the game is not in progress. The code block “On button B pressed” fulfills this condition, and within that block, the variable “highscore” will be displayed.



We will use button A to start the game, which will trigger the countdown. Before anything happened, we must initialize the sprites in the game. Sprites are basically entities represented by a single LED on the Micro:bit screen. They can move around and change direction using the code blocks provided in MakeCode. We will also initialize the variables “alive”, a boolean which accounts for whether the player is still alive, and “speed”, which determines how fast the projectiles move. Counter-intuitively, the lower the number, the faster the projectiles move.

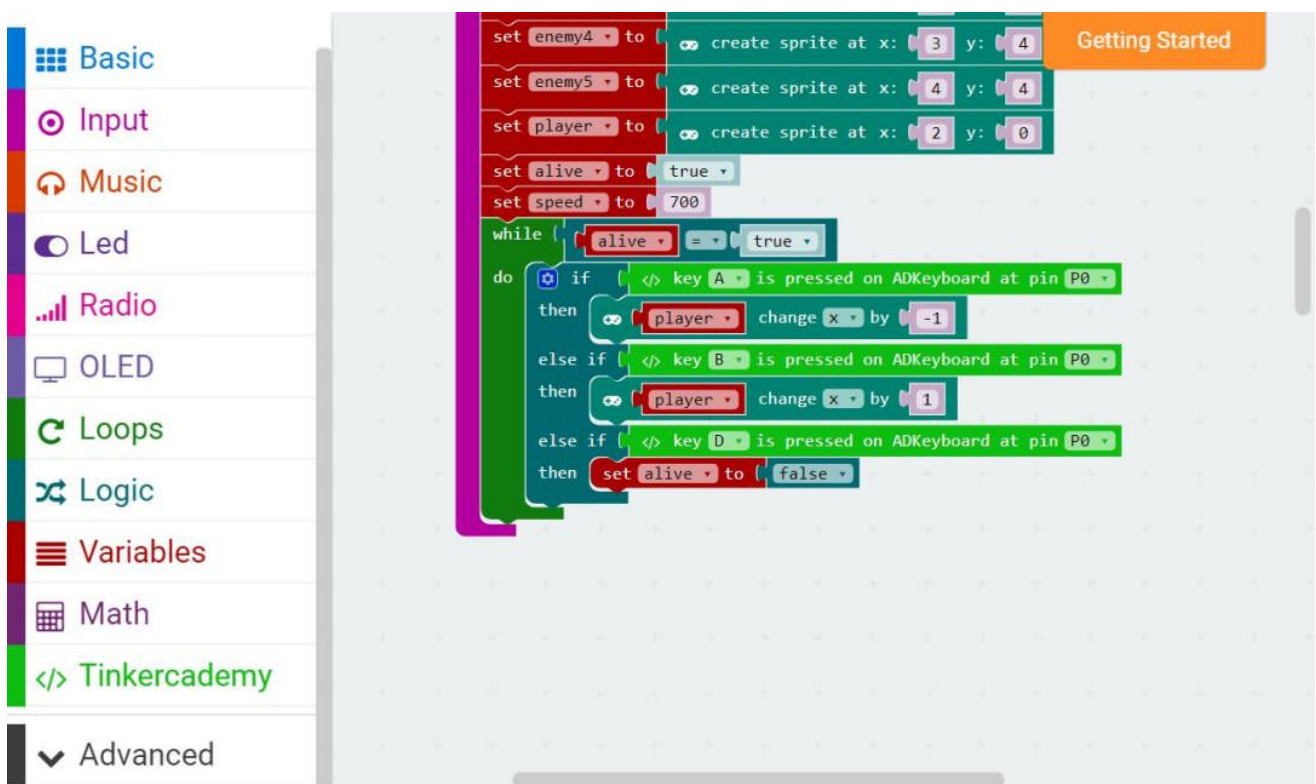


Next, we will add a while loop. A while loop will repeatedly run itself as long as the conditions specified are met. In this case, only if the player is still alive, the game will continue to run.



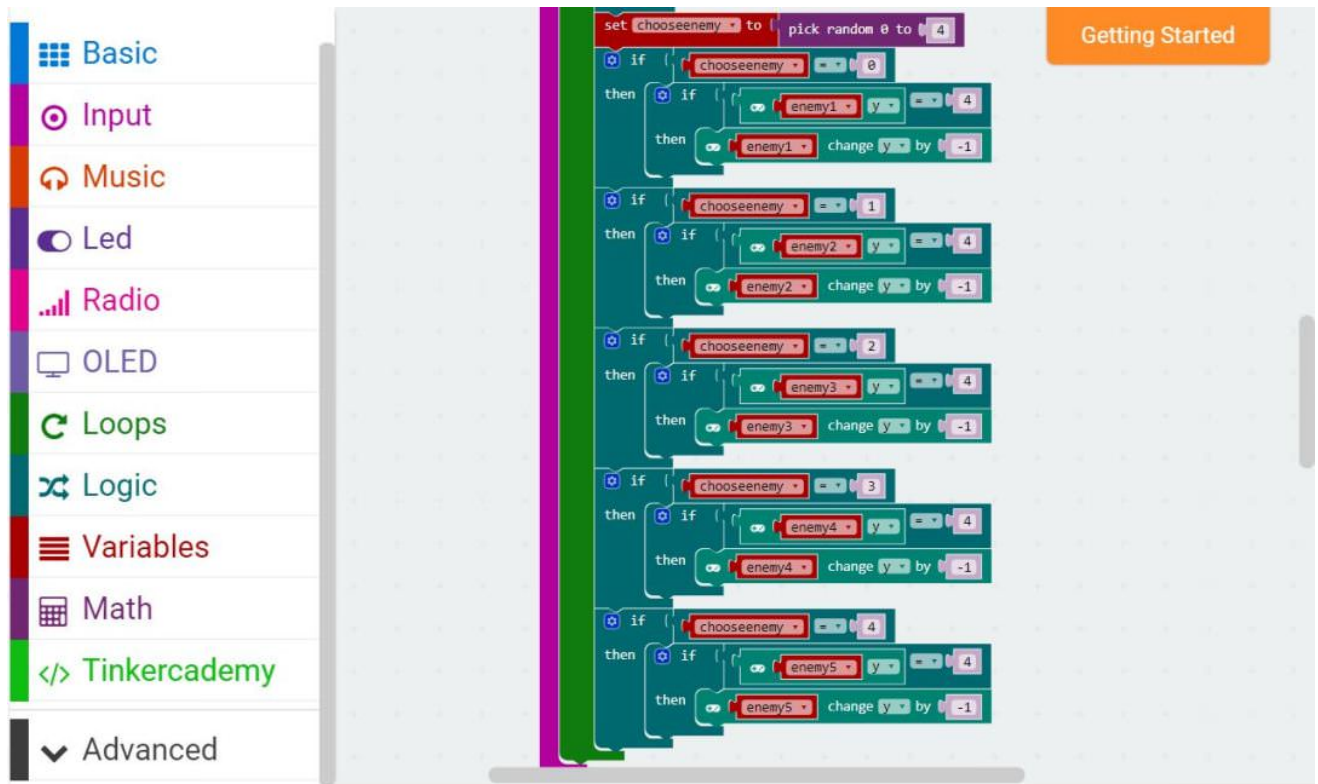
Note: Be careful here! Because the while loops do have the potential to crash your Micro:bit.

Inside the loop, we will add a group of code which governs the control of the game – the ADKeyboard. When the red A button is pressed, the sprite will move left. When the red B button is pressed, the sprite will move right. When the blue D button is pressed, the game will immediately stop.



After that, we will code for the enemy projectile's movement. First we will choose a random number using the pick random block from the Math module. This number will determine which projectile will start moving up by 1. However, this only applies if the projectiles are on

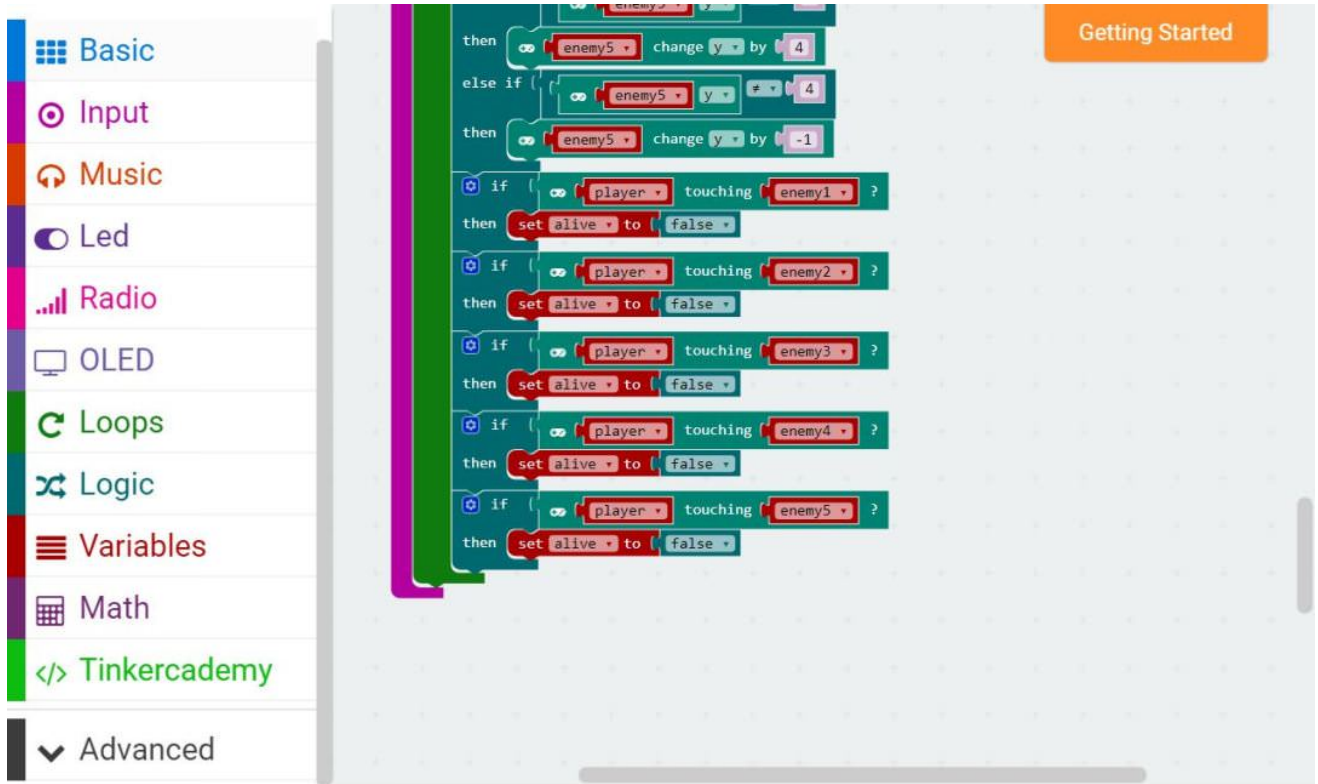
the bottom row, as we will have more code which determines the behaviour of the projectiles when they are off of the bottom row.



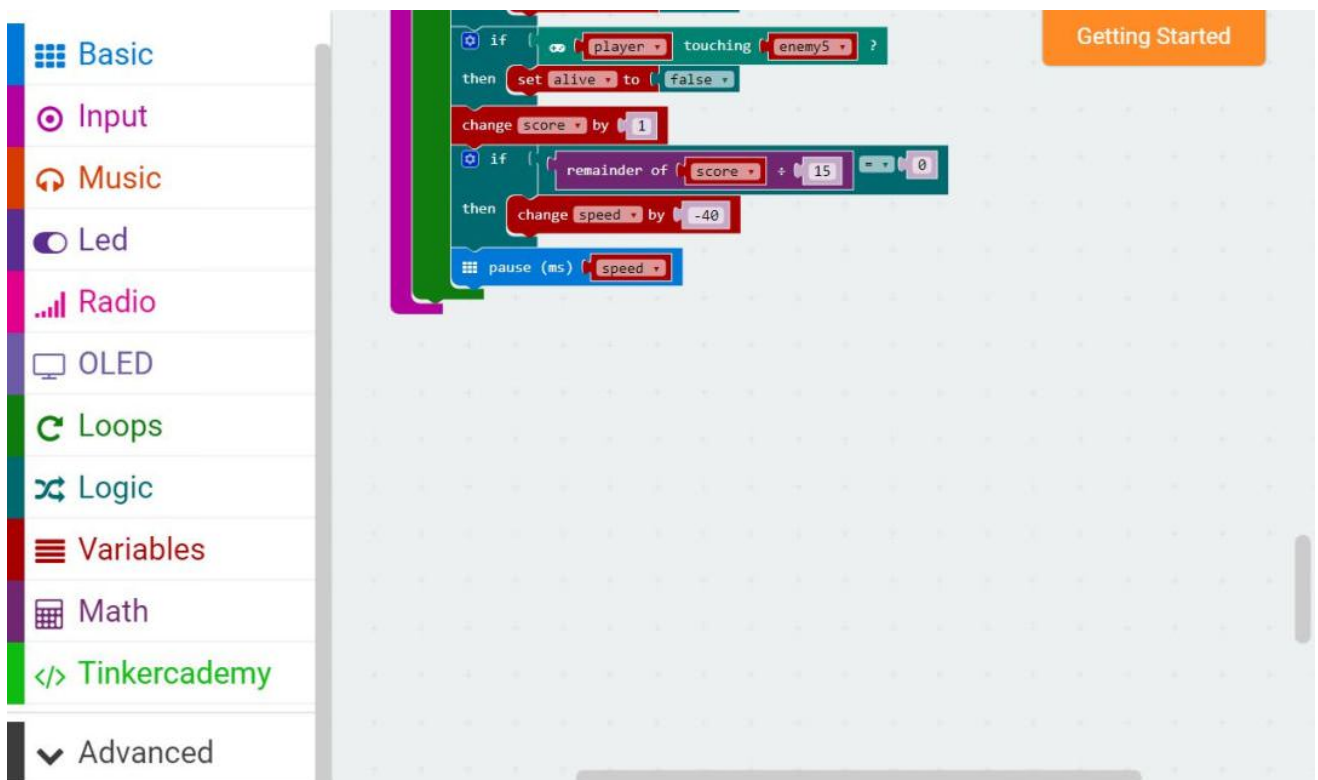
This bit codes for the behaviour of the projectiles when they are in motion already. They move up by 1 if they are in the middle three rows, and they return to the bottom row if they are already at the top row.



We also have to check if the sprites are touching the player so that we can know when the player is hit. If the player is hit, the variable "alive" is changed to "False", The while loop will stop looping, and the game will stop too.



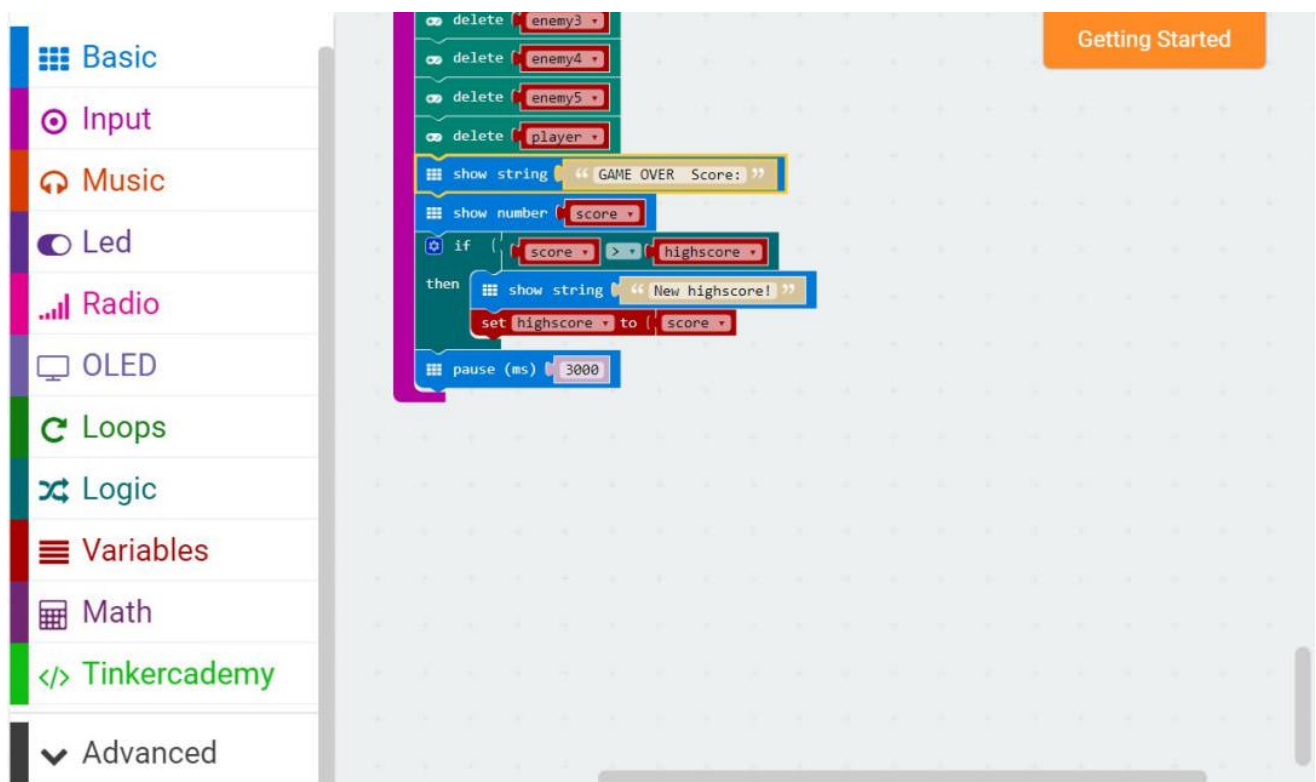
Finally, we will increase the score by 1 for every loop. For every 15 points gained, the variable "speed" will be decreased by 40, causing the projectiles to speed up. The pause controls said speed of the projectiles.



After the game is over, we must delete the sprites so that they do not clog up the LED screen.



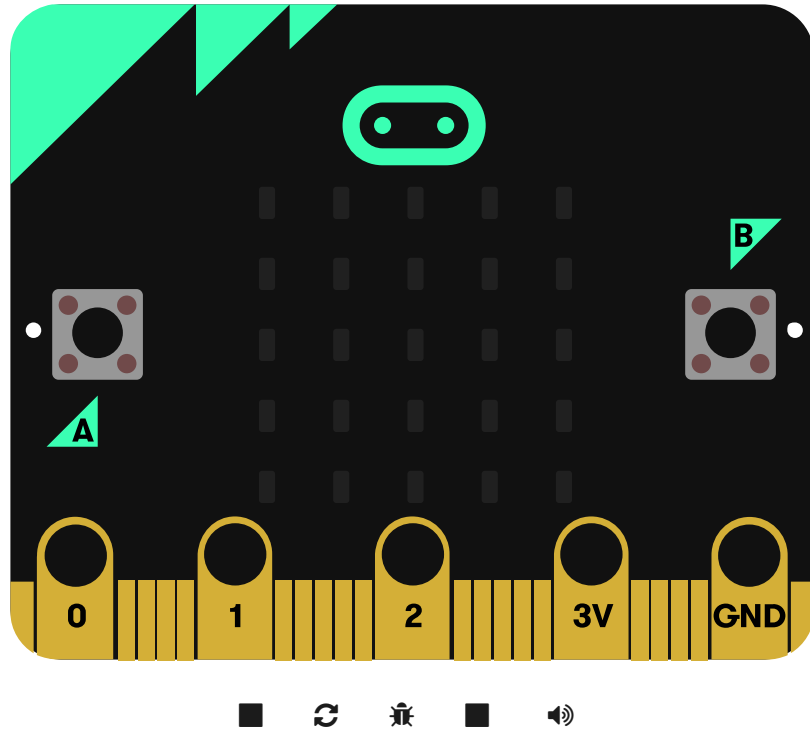
The game will display “Game Over” on the LED screen, followed by the score attained. If the score is higher than the current highscore, then the highscore will be replaced.



If you don't want to type these code by yourself, you can download the whole program from the link below :

https://makecode.microbit.org/_i92YKmDhr9Tf

Or you can download from the page below :



Step 4 – Success!

Voilà! You have created your own mini video game console with your Micro:bit. Now go out there and show your friends who's the real boss!

14. case 12 Remote Control Everything



Do you already have a micro:bit project you'd like to control from afar? Partner up with a friend, or grab a spare micro:bit, to make a remote controlled project with 2 micro:bits. (Don't grab a friend's micro:bit. Be nice.)

14.1. Goals

- Enable remote-controlling for the micro:bit car from this tutorial.
- Use a spare micro:bit to make a remote control for an existing project!
- Remote control everything!

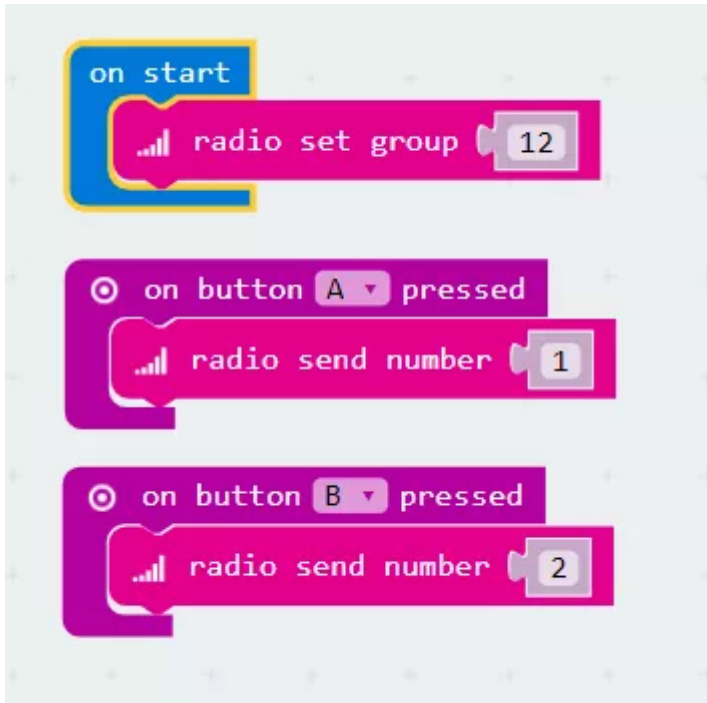
14.2. Materials

- 1 x BBC micro:bit
- 1 x Micro USB cable
- 1 x Battery box
- 2 x AA batteries
- 1 x micro:bit car OR
- 1 x a project you'd like to remote control

14.3. How to Make

Step 1

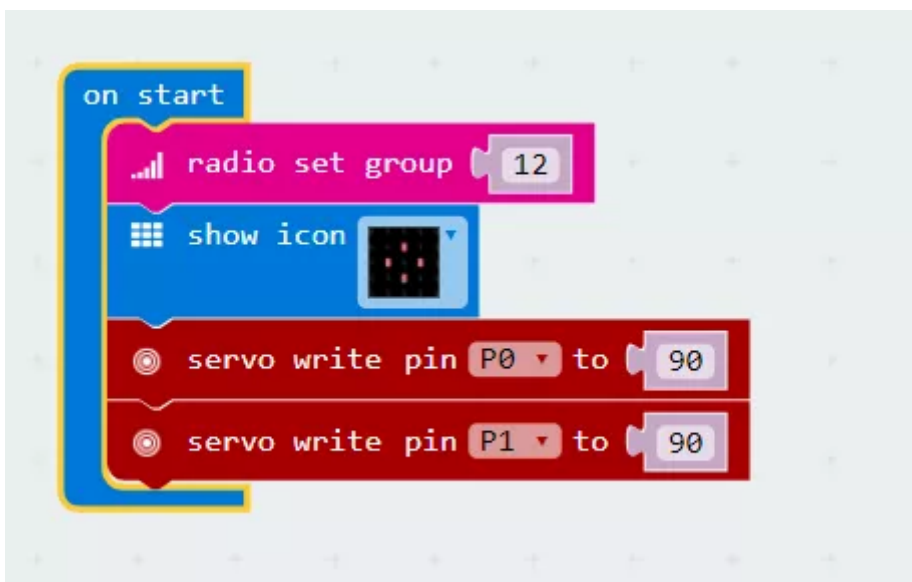
Set your radio group in makecode. This ensures your transmitter and receiver are in the same channel. Think about the usage of each button on your remote controller. Radio send a different number with each button press event block by using the blocks shown. You can find these blocks under the Radio drawer. Download this into the micro:bit you'll use as your remote controller. Now each button on your micro:bit remote controller will send a different command!



Step 2

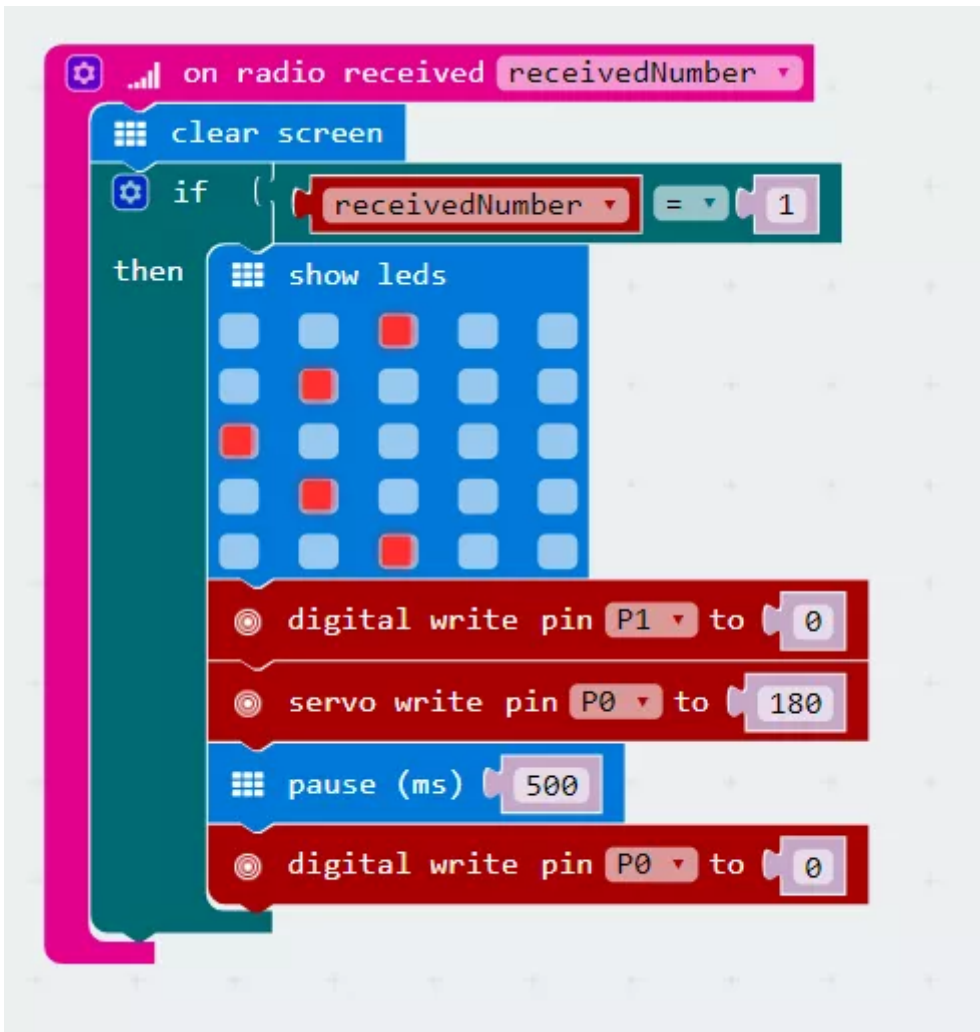
In your micro:bit car project (or the particular project you're trying to remote control), add the same radio group block to your On Start block.

This ensures the project you're trying to remote control will listen to the right commands!



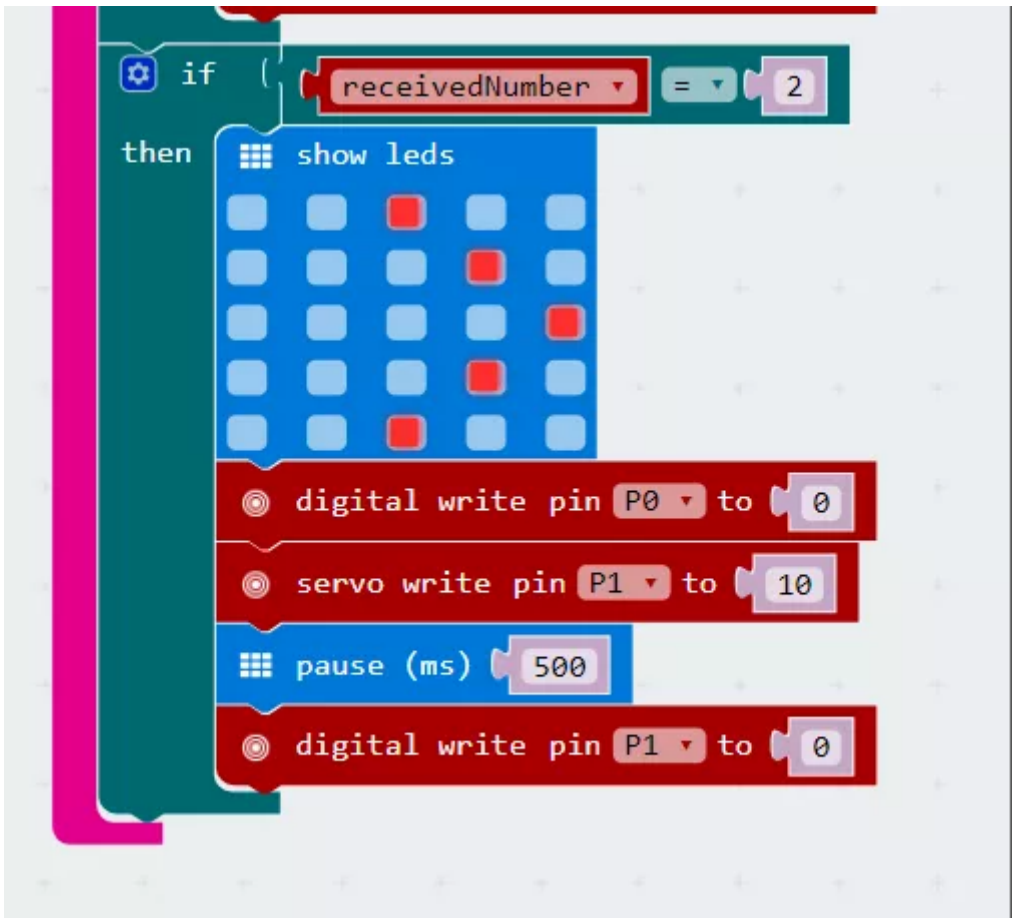
Step 3

Remember the numbers sent from our remote controller every time we pressed a button? We're going to use that to trigger an action. Find the radio received block as shown in the Radio drawer. Use an if-then block to check if the number you received is the number you sent when pressing button A. Take the code that turns your micro:bit car left, and place it within this if-then block. We have also added an led indication pointing left just to show what was supposed to happen. Turn off the left servo afterward by digitally writing the pin to 0.



Step 4

Do the same to the code that turns your micro:bit car to the right! Make sure you turn the right wheel off afterward. You can always choose to leave the wheels on without stop after receiving each command. But you'll face with a situation that the car keeps spinning in circles. Download this program into your micro:bit car.

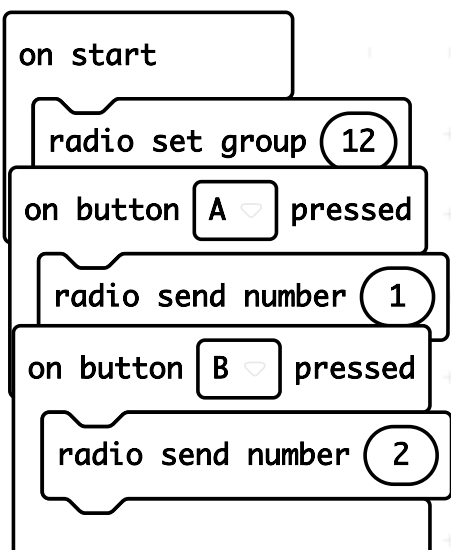


If you don't want to type these code by yourself, you can download the whole program from the link below.

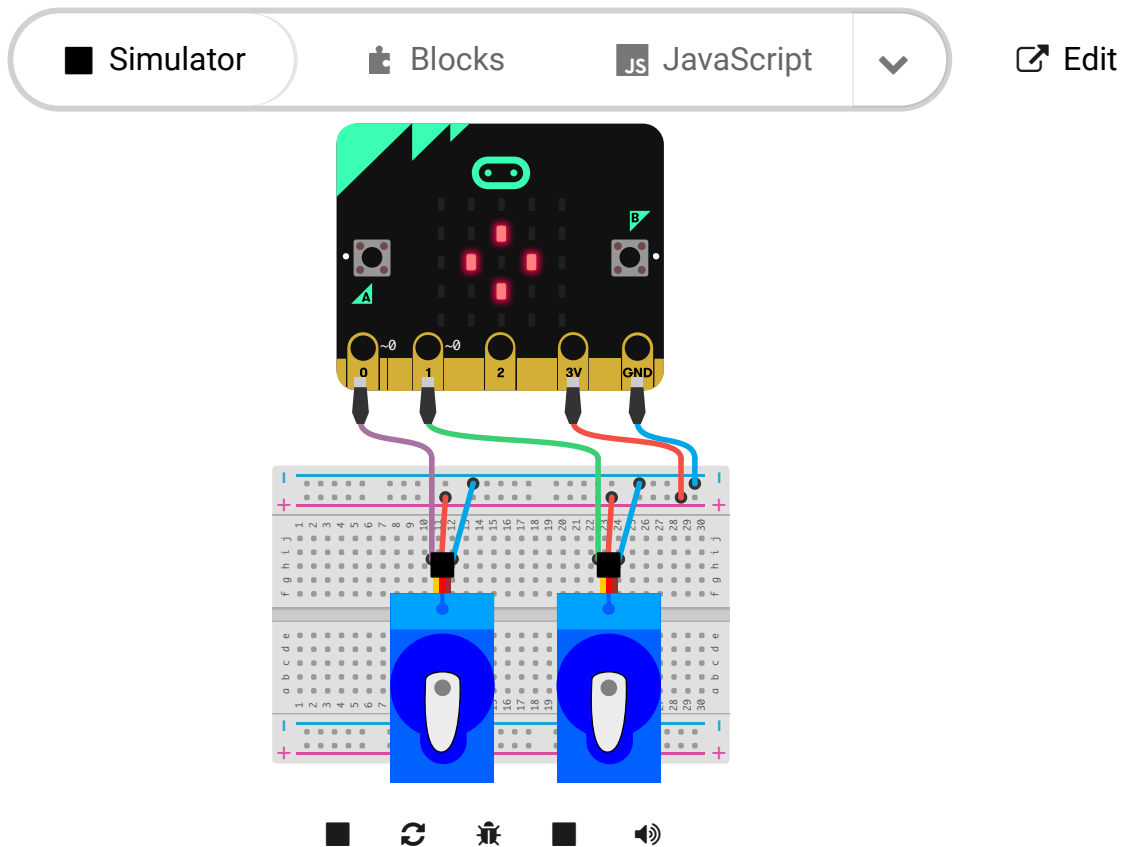
Remote Control: https://makecode.microbit.org/_gH73AW4Dy1rP Receiver: https://makecode.microbit.org/_4am87cCWb0e9

Or you can download from the page below.

Remote Control:



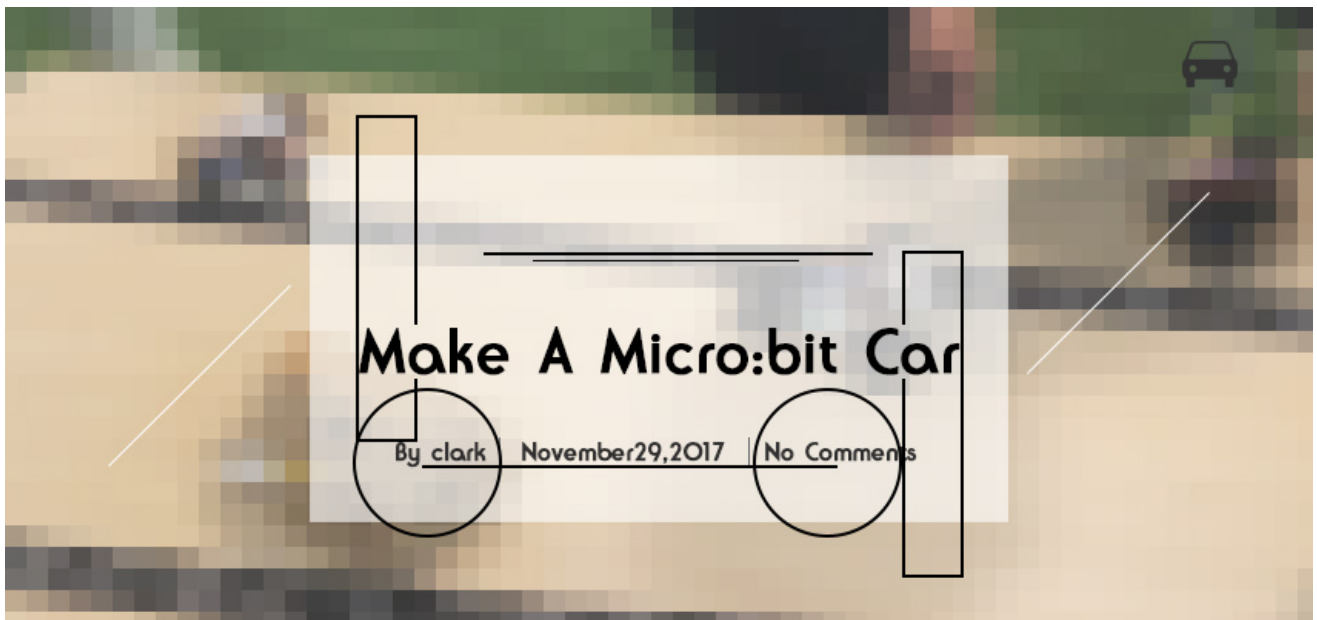
Receiver:



Awesome!

Now that all your code is snugly tucked into your micro:bits, attach your battery packs and get going! Play around and see what other commands you can send with the A+B button, or try different kinds of inputs instead of buttons. Then remote control all your other micro:bit projects. Woo-hoo! World domination without leaving your seat!

15. case 13 micro:bit Car



Make your very own self-driving micro:bit car! (Disclaimer: It's only "self-driving" inasmuch as a ball rolling down a hill is "self-rolling".)

15.1. Goals

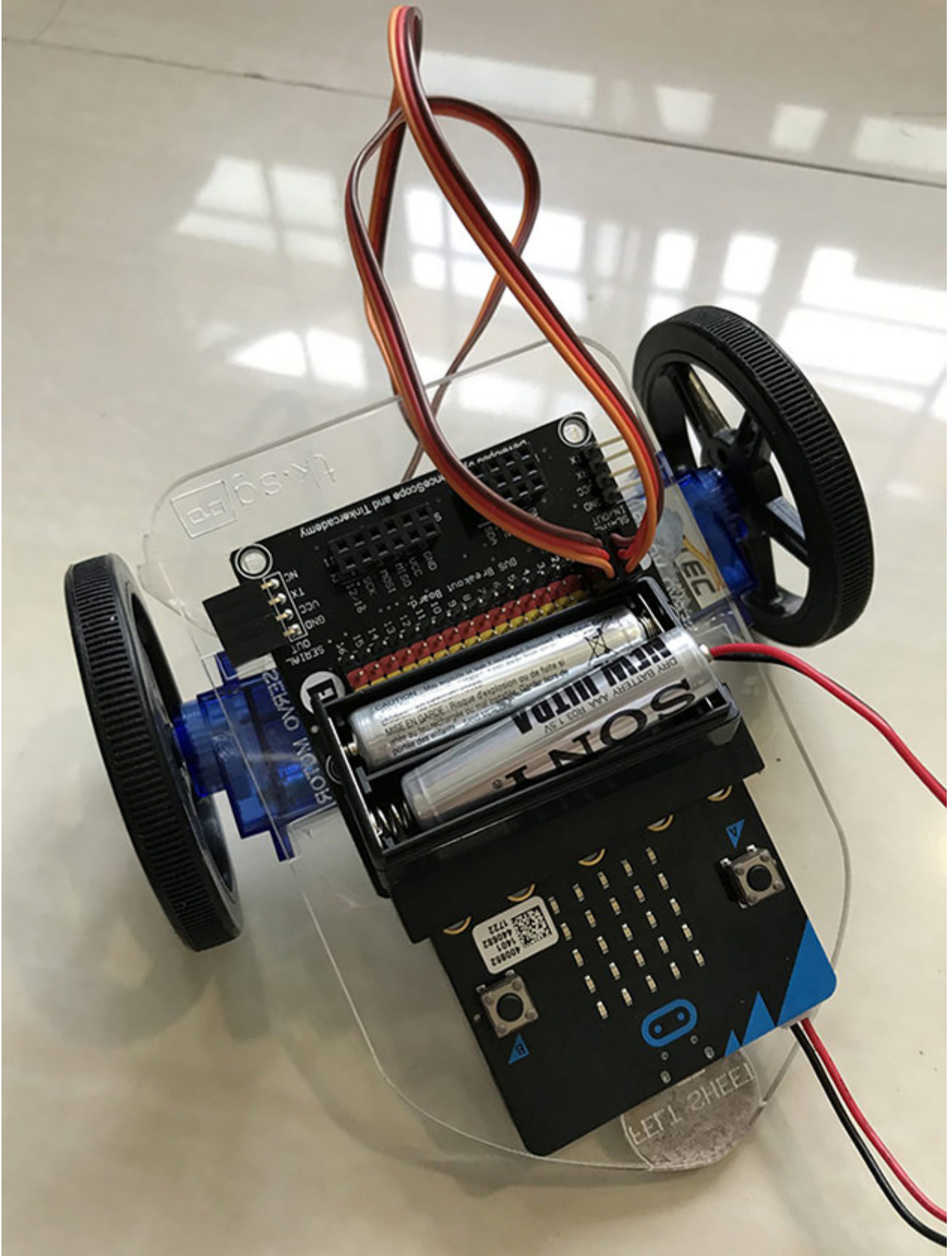
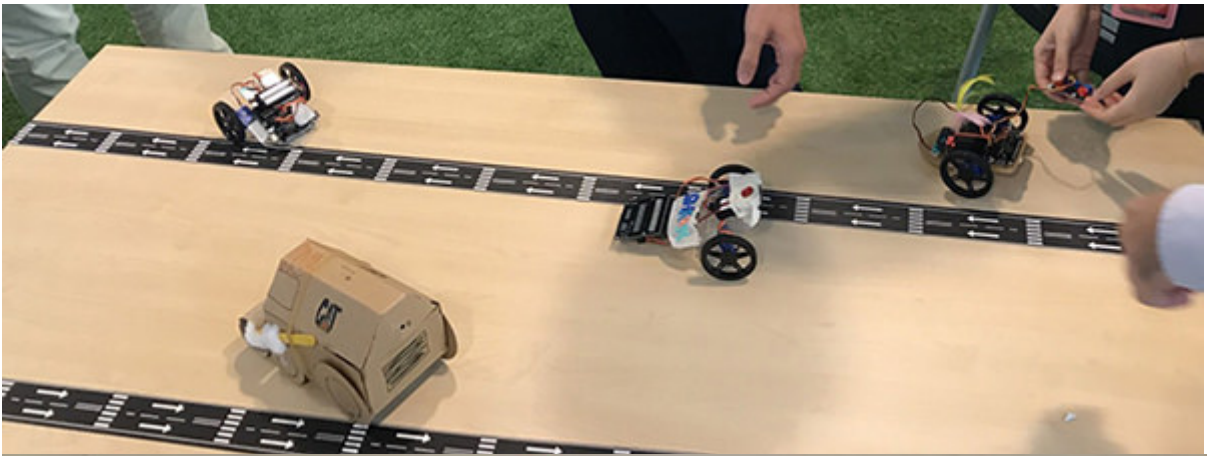
- In this project, we're going to use the Micro:bit, Breakout Board, and Servos to make a self-driving car!
- Get to know the Servo and how to use it with the Micro:bit, Breakout Board and MakeCode.
- Marvel at how ridiculous this thing is!

Note: This activity uses extra parts not found in the Tinker Kit. (Stay tuned to our Online Store for our Car Kit !)

15.2. Materials

- 1 x BBC micro:bit
- 1 x Micro USB Cable
- 1 x Battery Box
- 2 x AA Batteries
- 1 x Breakout Board

- 2 x Servo
- 1 x Acrylic Car Body
- 2 x Wheels
- 1 x Felt Pad
- Sticky Tape



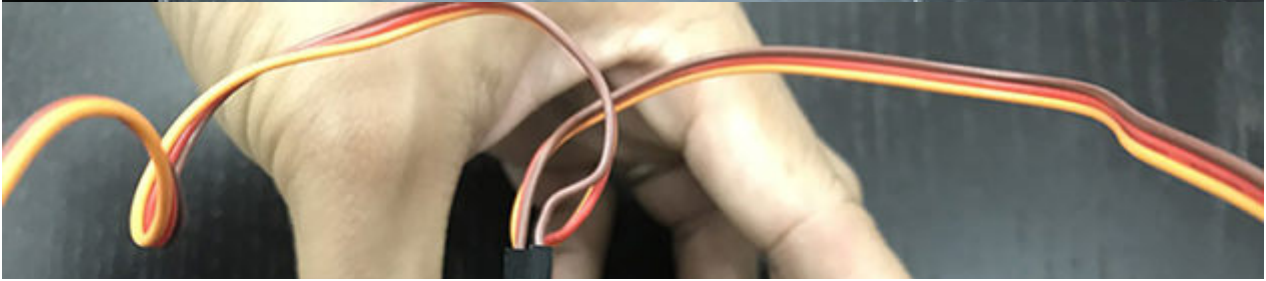
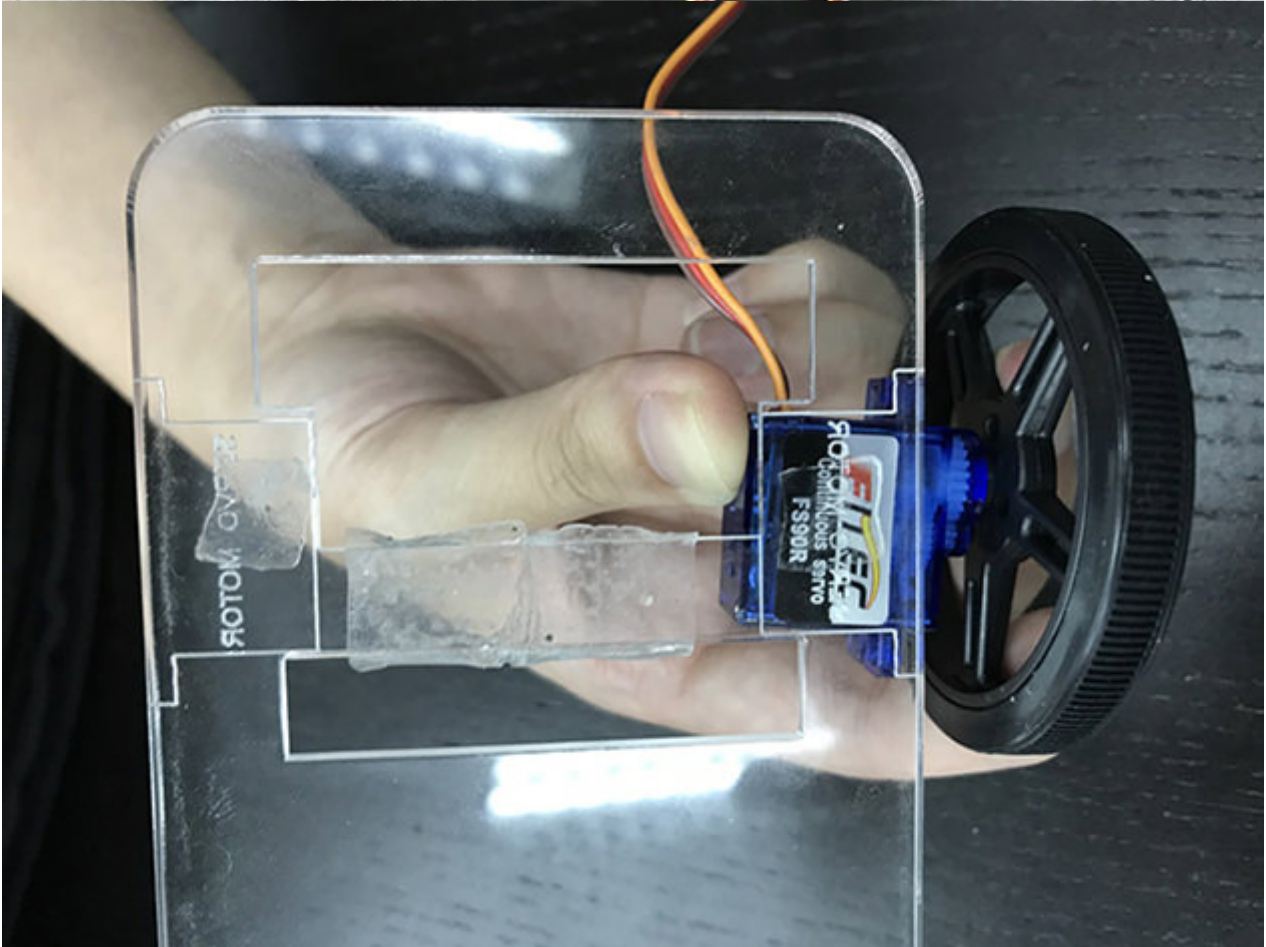
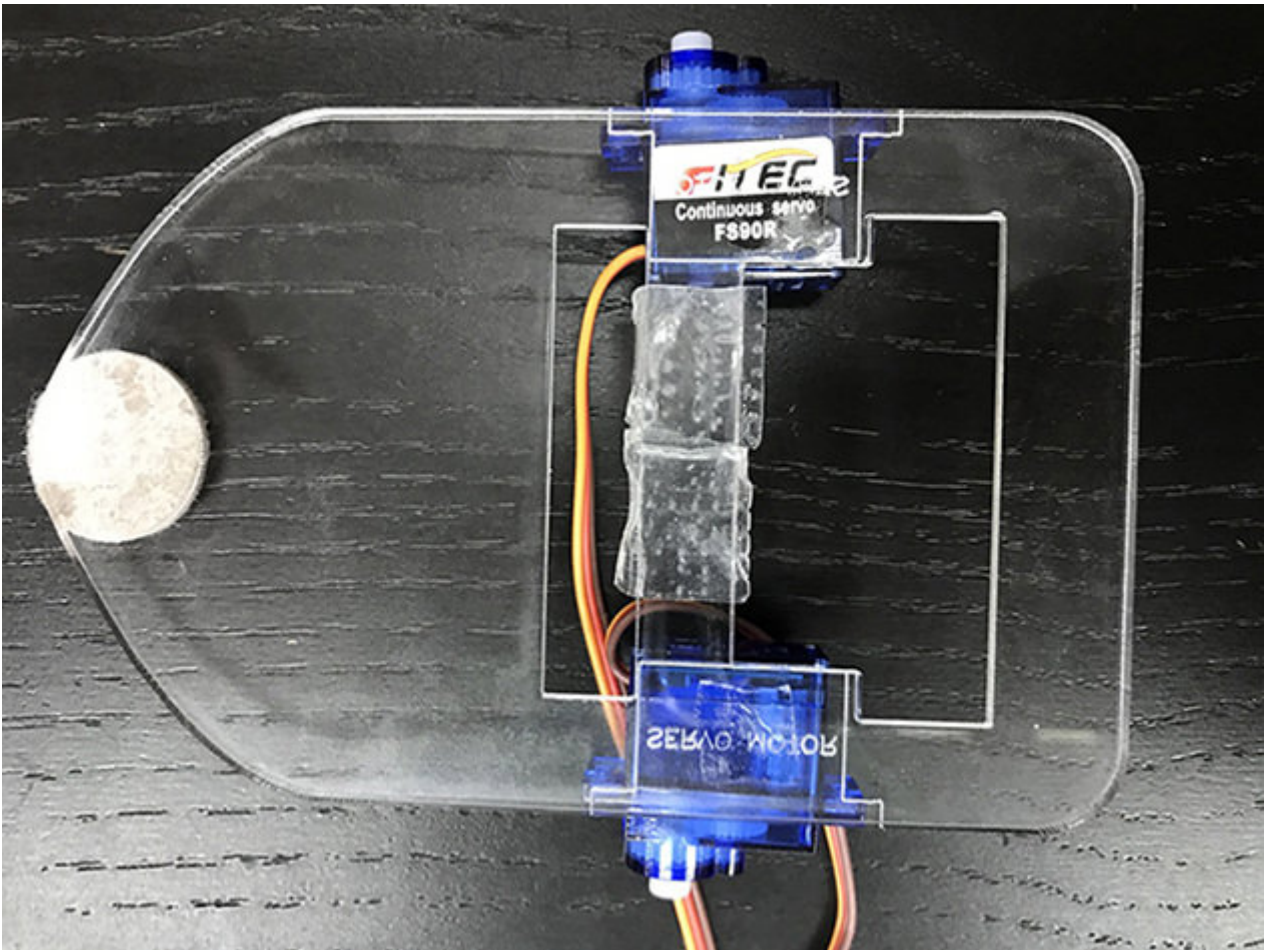
15.3. How to Make

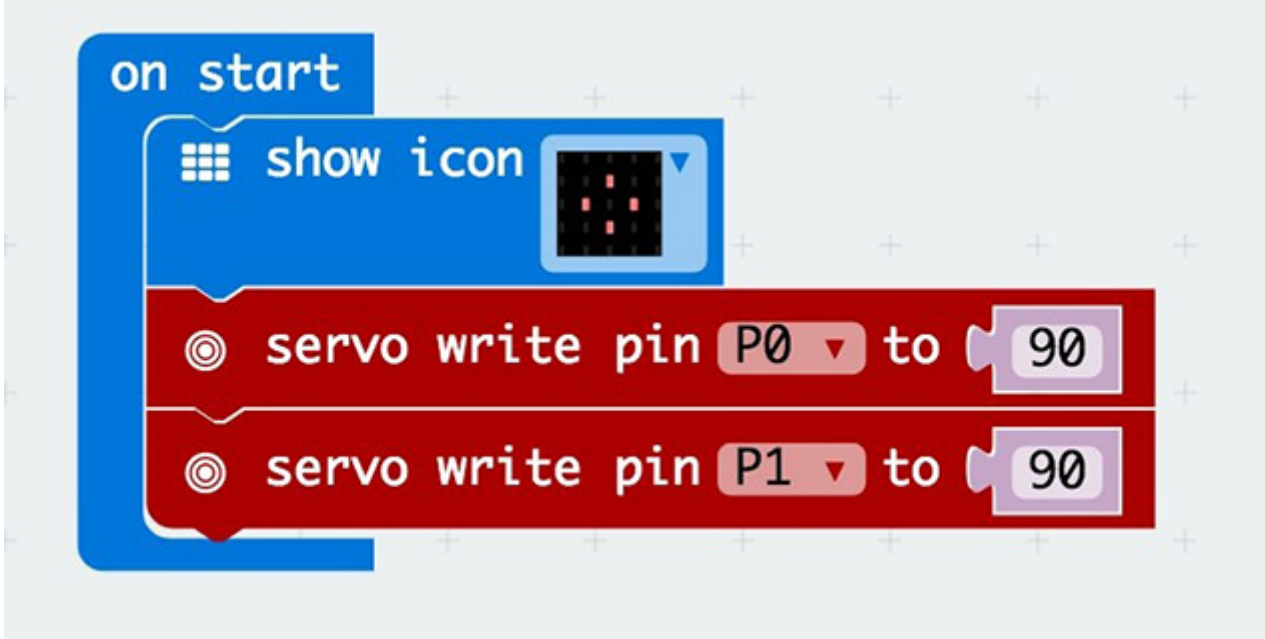
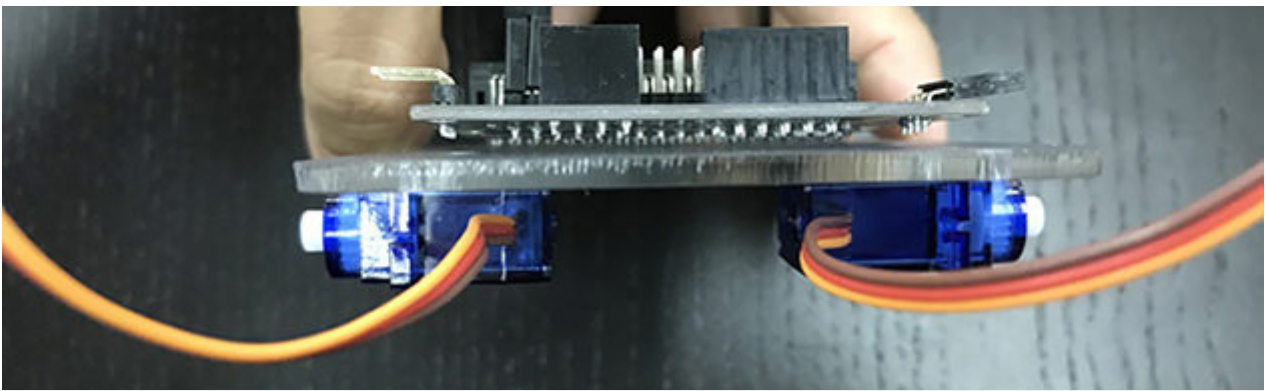
Step 1

Connect your car parts as shown in the following pictures. If you're using our car kit, follow the labels on the car body to insert the components correctly with the sticky tape.

Connect the servo connectors to Pin 0 and Pin 1 on the Breakout Board.

Note that the colours of common servo cables don't match the yellow, red, and black colour scheme of the breakout board exactly. Match the orange servo cable to the yellow pin, and the brown servo cable to the black pin.





Step 2

Add the blocks shown to your On Start block. What this does? Reset the servos to fixed positions whenever we start! The Servo block (coloured red) in MakeCode takes values from 0 to 180. You can find it under Advanced, then Pins. For the continuous servos we're using, a value of 90 is right in the middle. In other words, we're telling the servo to "stay still". We display an image to make a visual indication that we've downloaded our code into the Micro:bit.

Step 3

Let's make the wheels move! Add the code shown on the right to your Forever block. The Digital Write Pin to 0 block is also found under Advanced, Pins. What's happening here? We're turning one servo clockwise (180), while turning off the other servo. Then, after a short pause, we're turning off the former servo, and turning the latter servo anti-clockwise (0). Remember, 90 is straight ahead! Why do we need to turn off one servo at a time? That's because of battery power requirements—your micro:bit has trouble in powering both servos at once. If you're interested, you can explore by using a DC motor with an external power source. Or you can email us to find out more! Make sure to check that your motors are facing the right directions—you can change the travel directions of the motors by swapping the 0 and 180 values.

```

forever
  digital write pin P0 to 0
  servo write pin P1 to 180
  pause (ms) 500
  digital write pin P1 to 0
  servo write pin P0 to 0
  pause (ms) 500

```

If you don't want to type these code by yourself, you can directly download from the link below.

https://makecode.microbit.org/_Ef87EJAepcve

Or you can download from the page below.

Simulator
 Blocks
 JS JavaScript

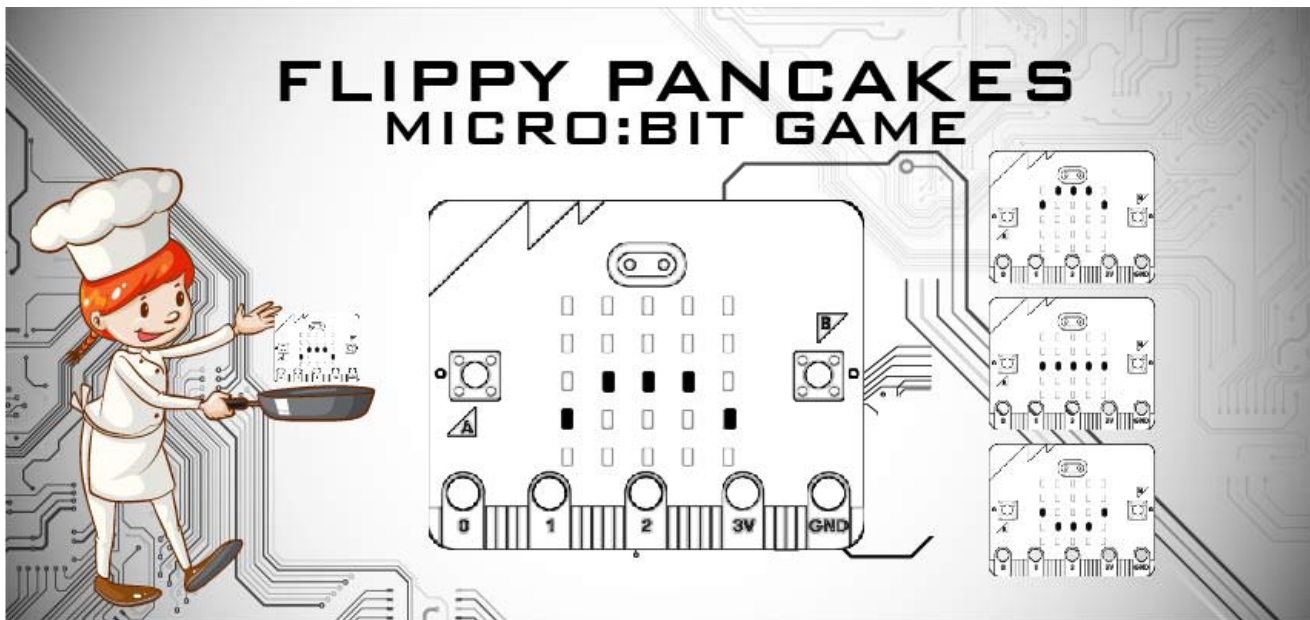
Succeed!

When you're ready to run your car, attach your battery pack to your micro:bit, and your car runs ! Besides, you can personalise your car with some craft material to improve its aerodynamic properties! For further extension, you can also hook up an ADKeyboard to control the motors manually, instead of having the car move autonomously.





16. case 14 Flipping Pancakes



Do you have what it takes to flip the perfect pancake?

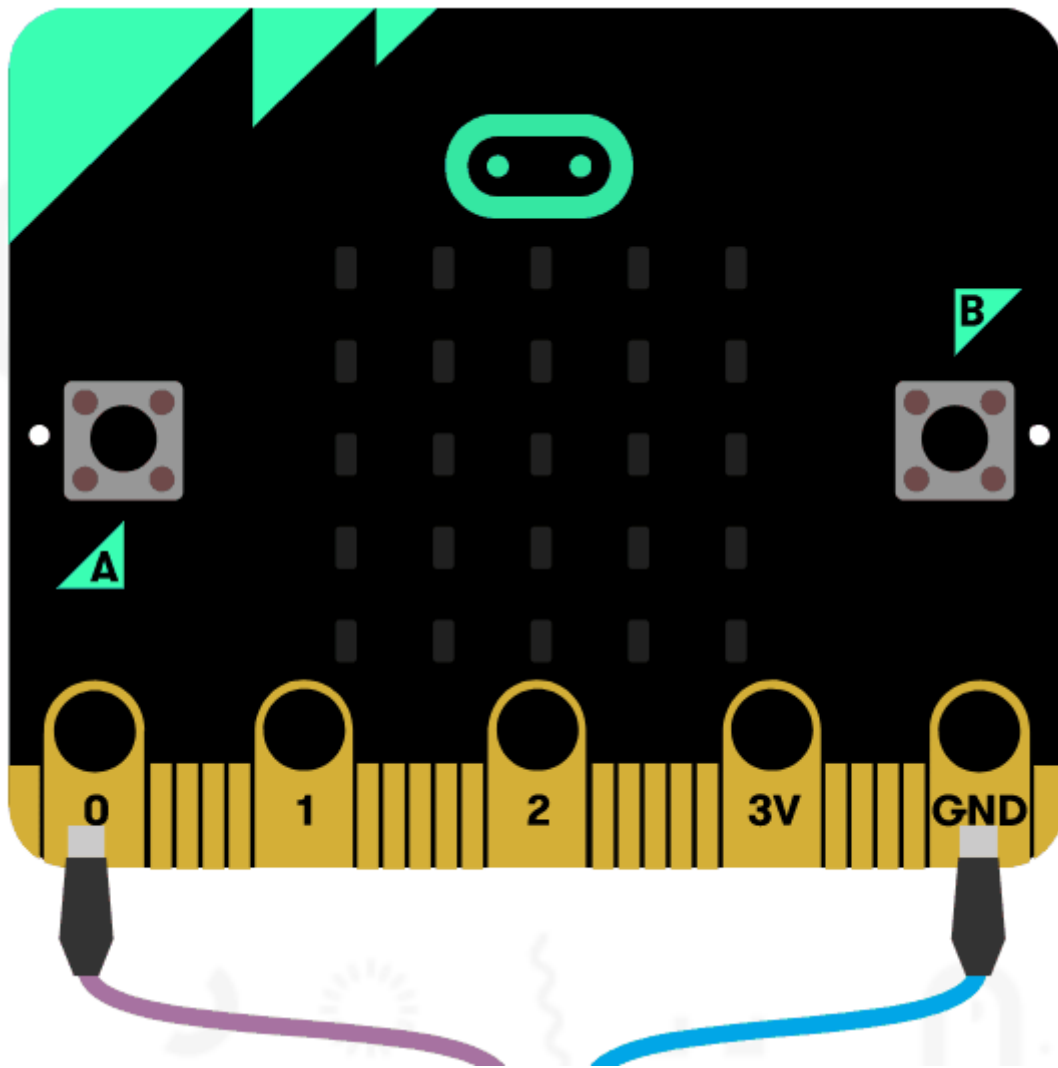
16.1. Goals

We're going to create a flippy pancake game where you must flip the pancake at the perfect time. Too fast, your pancake will be mushy; too slow, it will be burnt! You'll learn how to...

- Use a Buzzer and ADKeypad with the micro:bit.
- Use if-else statements to evaluate conditions.
- Create your own function on MakeCode.
- Customise your game!

16.2. Materials

- 1 x BBC micro:bit
- 1 x Micro USB cable
- 1 x Buzzer
- 2 x F-F Jumper Wires
- 1 x ADKeypad

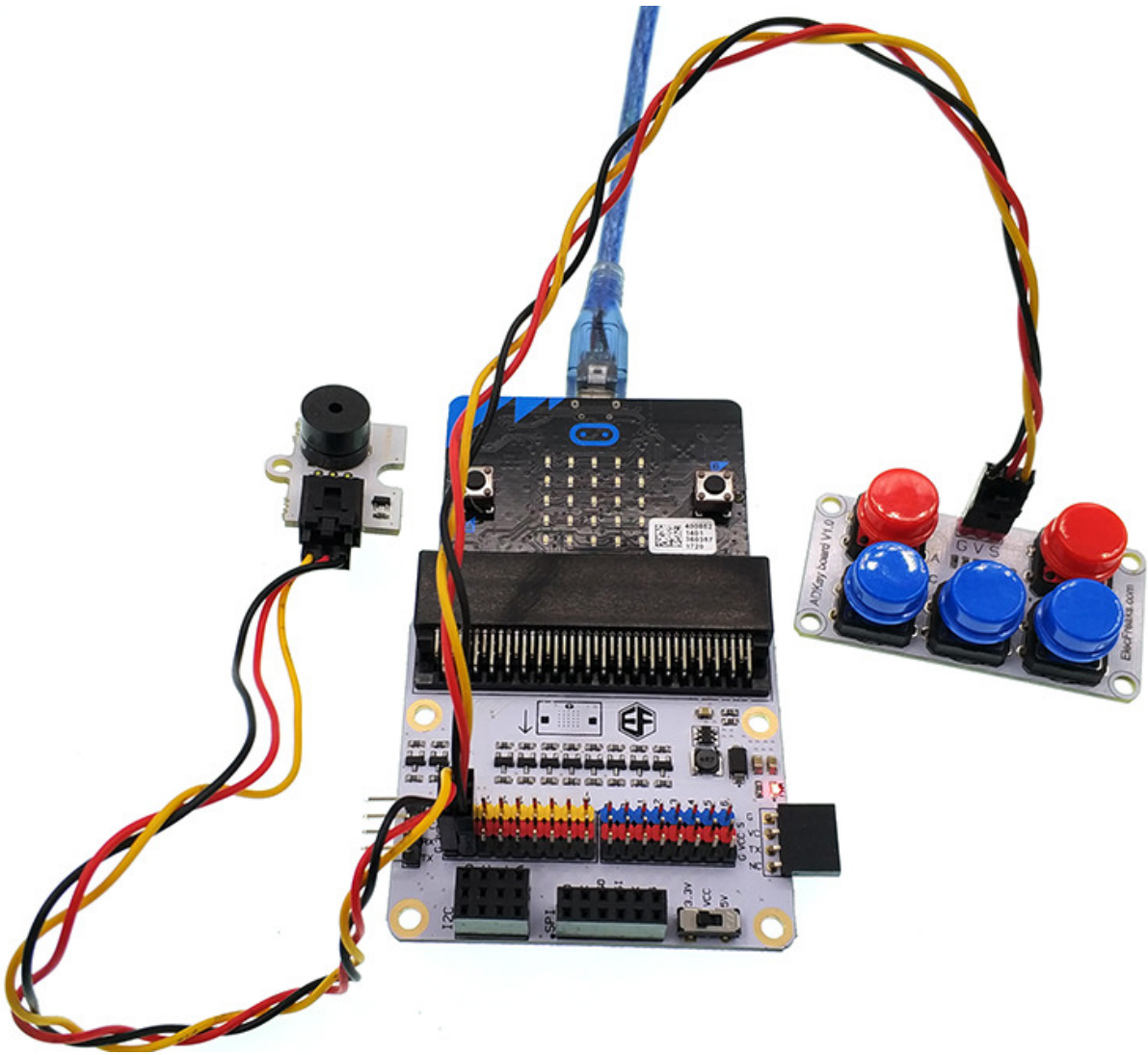


16.3. How to Make

Step 1

Plug in your Buzzer to Pin0. Make sure the positive lead is connected to the yellow signal pin and the negative lead is connected to the black ground pin on the breakout board.

Plug in the ADKeypad to Pin1. Match the colours of the wires to the ones on the breakout board!



Step 2

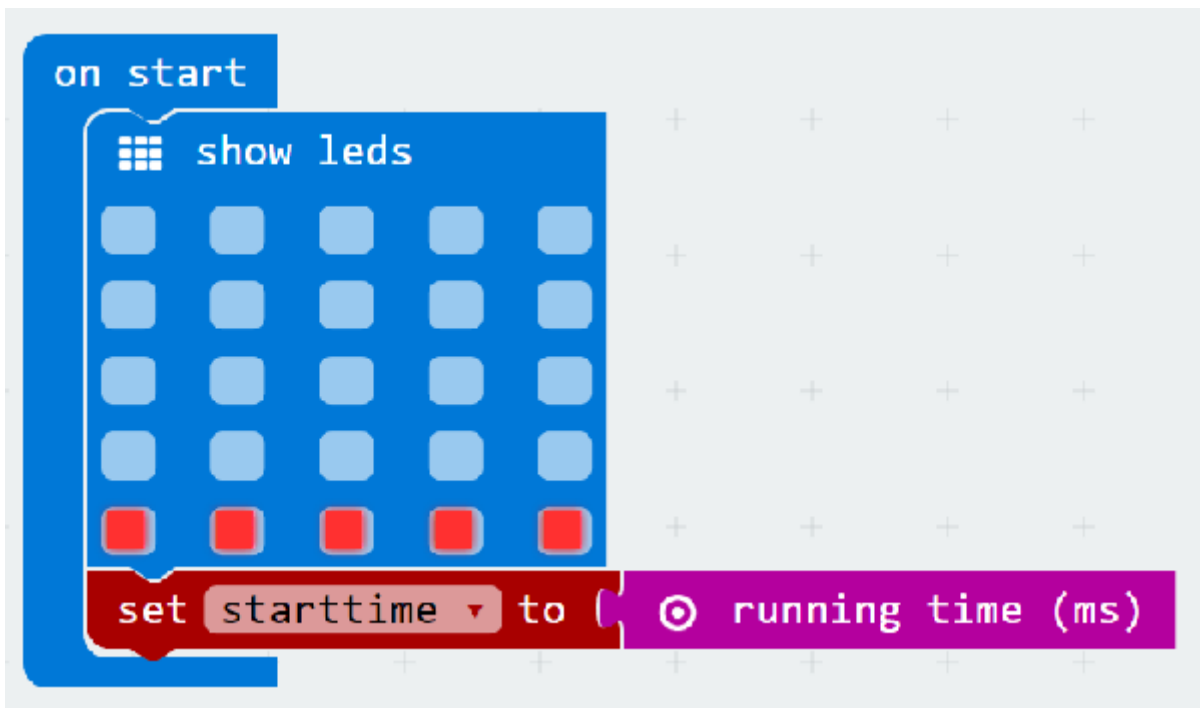
In Makecode, we'll track the length of time using two variables. Variables are like buckets that can hold changing values.

Every time we turn on micro:bit, a hidden timer keeps track of how long it has been on. We're going to use this hidden timer to calculate the start time, end time, and total length of the game.

Create a new variable called `startTime` (or anything you like, really) in the Variable drawer.

When we start the game, we want to set our variable `startTime` to the running time of the micro:bit.

We also want to display a flat pancake on the screen with the LEDs.



Step 3

We want to set up the game so that when you press the A button on the ADKeypad, a pancake-flipping animation will be played on the micro:bit.

To do this, we need to create a function. A function is a piece of code that performs a specific task every single time it's called. In this case, our task is to display the pancake-flipping animation.

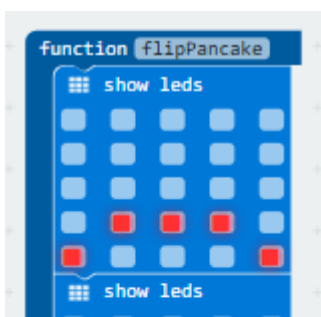
To create your own function, click on the Functions category and choose the 'Make a Function' button. I named my function 'flipPancake'.

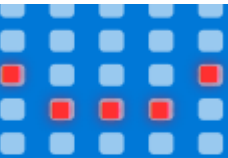
When you do this, a new block will be added to your screen called 'function flipPancake.' This is where we define our function, i.e. what will happen whenever we run the function block!

In this case, we're going to add multiple LED blocks inside our flipPancake function so it appears as if our pancake is being tossed into the air and is wobbling as it falls back down.

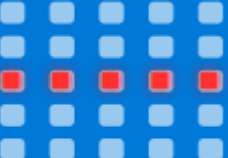
Once you have your function defined, you can run it anytime by using the new 'call function flipPancake' block inside the Functions category.

Feel free to customise your own pancake-flipping animation. This is just one example!





show leds



show leds



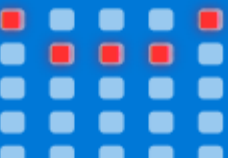
show leds



show leds



show leds



show leds



show leds



show leds



show leds



show leds





Step 4

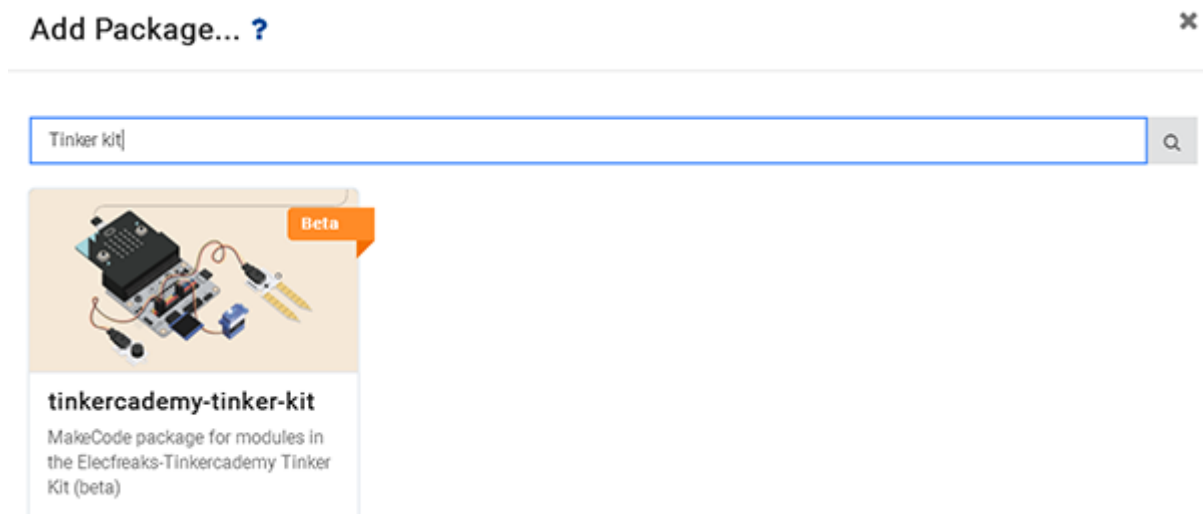
Now we're going to sense button press on the ADKeypad. To do so, we need to import a special package into MakeCode.

Expand the 'Advanced' section, scroll to the bottom and click on 'Add Packages'.

In the search box, type in 'tinker kit'. Then click on the box labelled 'tinker kit'.

Now you'll see something new in MakeCode – a bright green Tinkercademy category has been added!

Inside this category, you'll find blocks to sense button press on the ADKeypad. This package importation only happens in current project. So if you start a new project, you'll need to re-import it.



Step 5

Now that we can sense button press, let's create the main code for the game! We need to calculate the time whenever a player presses button A and figure out if the pancake is mushy, perfectly cooked, or burnt.

We start off with a forever loop. Inside the forever loop, we put an if-else statement to test if button A is pressed. If-else statement judges if a condition is true. If it is true, then implement the program; if it is false, then skip it. Because this if-else statement is inside a forever loop, it will forever test to see if button A is pressed.

To do this, we're going to check the current running time of the micro:bit and subtract the value of our `startTime` variable. This gives us the total time the current game has been running. We store this in another variable called `totalTime` (you can create this in the Variables category).

After calculating the `totalTime`, we call our `flipPancake` function! This will animate the pancake!

Next, we need to test the `totalTime` to judge if the pancake is mushy, perfect, or burnt. We use if-else statement again for this. But in this case, we're going to link the if-else statements together to test multiple conditions back-to-back. When if-else statements are linked together, only one can be run. As soon as the micro:bit finds one that is true, it skips testing all others that are linked. We can add linked if-else statements by clicking on the gear icon on the if-else block and dragging in more if-else blocks.

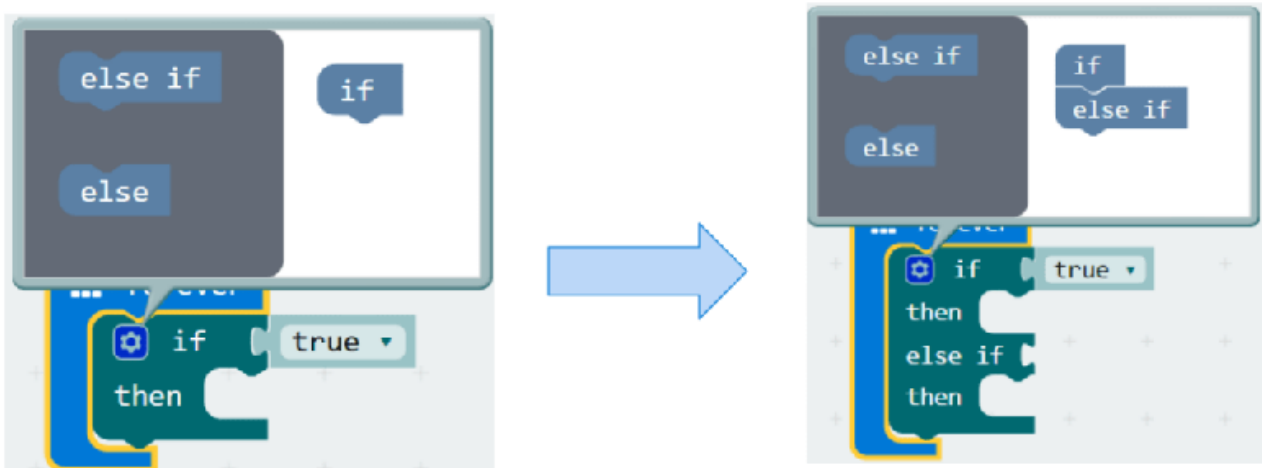
In this case, we start by testing if `totalTime` is less than 7000 (keep in mind this is milliseconds! So 7000 milliseconds = 7 seconds). If `totalTime` is less than 7000, we determine that the pancake is mushy and display a frowny face and a MUSHY message. If this first condition is true, the micro:bit will skip testing all the following conditions!

If `totalTime` is not less than 7000, we test if `totalTime` is less than 8000 milliseconds. We know at this point that `totalTime` is greater than 7000. So if it is also less than 8000, we determine that the pancake is perfect and display a happy face and a PERFECT message.

Finally, if neither of the previous two conditions are true, then we know that `totalTime` must be greater than 8000. So we determine that the pancake is overcooked and display an angry face and a BURNT message.


```

forever
  if (key A is pressed on ADKeyboard at pin P1)
  then
    set totaltime to (running time (ms) - starttime)
    call function flipPancake
    if (totaltime < 7000)
    then
      start melody (wawawawaa) repeating (once in background)
      show icon (MUSHY!)
      show string ("MUSHY!")
    else if (totaltime < 8000)
    then
      start melody (nyan) repeating (once in background)
      show icon (PERFECT!)
      show string ("PERFECT!")
    else
      start melody (baddy) repeating (once in background)
      show icon (BURNT!)
      show string ("BURNT!")
  
```

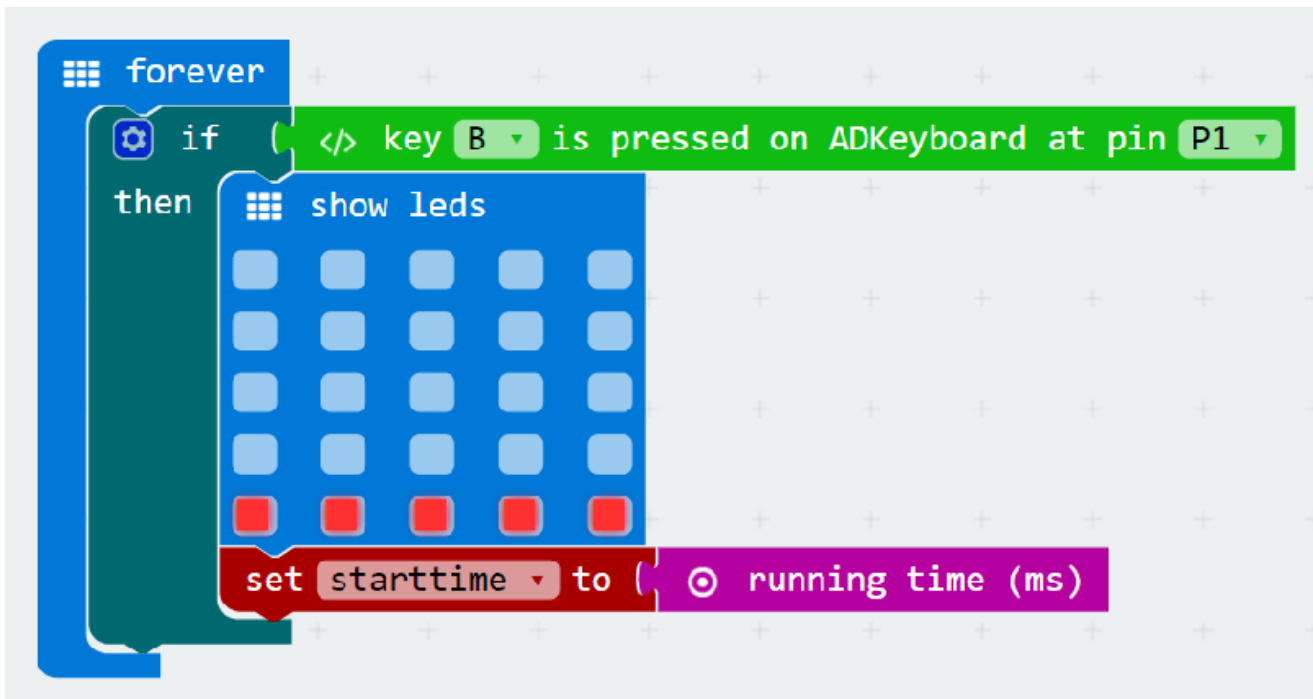


Step 6

Now that we have the game running. Let's make it so that players can play more than once without resetting the micro:bit.

To do this, we'll reset the game when button B is pressed on the ADKeypad. Once again, we use a forever loop and place an if-else statement inside to test if button B is pressed (don't forget to specify the correct Pin number again!).

What else do we need to do when we restart the game? In this case, all we need to do is to display a new pancake, and reset the starting time.



Step 7

Now our game is working (hopefully)! Let's add some more fun to the game! We have a buzzer attached to the micro:bit but haven't used it yet! Note: Add this onto your existing code. It's not a brand new section!

Micro:bit has lots of pre-programmed music melodies available for us to use. Let's add music to our game! Here we add a different melody to each outcome of the game. For mushy or burnt pancakes, we add sad melodies. But for perfect pancakes, we play the nyan-cat melody!

We need to set these melodies to play 'once in the background', otherwise it could pause the entire game until the melody is finished playing.

```
forever
  if (key A is pressed on ADKeyboard at pin P1)
  then
    set totaltime to (running time (ms) - starttime)
    call function flipPancake
    if (totaltime < 7000)
    then
      start melody (wawawawaa) repeating (once in background)
      show icon (micro:bit)
    else if (totaltime < 8000)
    then
      start melody (nyan) repeating (once in background)
      show icon (micro:bit)
      show string (" PERFECT! ")
    else
      start melody (baddy) repeating (once in background)
      show icon (micro:bit)
```

Step 8

Finally, let's add some starting music and a starting message when we first start up the micro:bit. Note: Add this onto your existing code. It's not a brand new section!

We can use the buzzer once again to play a melody (once again we want it to play 'once in background'). We can also display the name of the game as well!

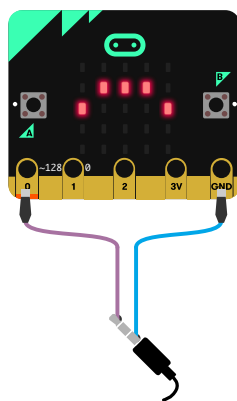
```
on start
  start melody { funk } repeating once in background
  show string "FLIPPY PANCAKES"
  show leds
  set starttime to (running time (ms))
```

If you don't want to type these code by yourself, you can download directly from the link below.

https://makecode.microbit.org/_aepYrcgwLFEy

Or you can download from the page below.

■ Simulator 🧩 Blocks JS JavaScript ▾ 📄 Edit



■ ↻ 🛑 ■ 🔊

—

⊖ ⊕

Cool Stuff!

Now you've learned how to use the ADKeypad, you can use it to control LEDs, servos, and other components! You have also learned about if-else statements and creating your own functions, which can be useful in many micro:bit projects! Try customising your pancake game as well!

17. case 15 Maze Runner



Can you make it through all levels?

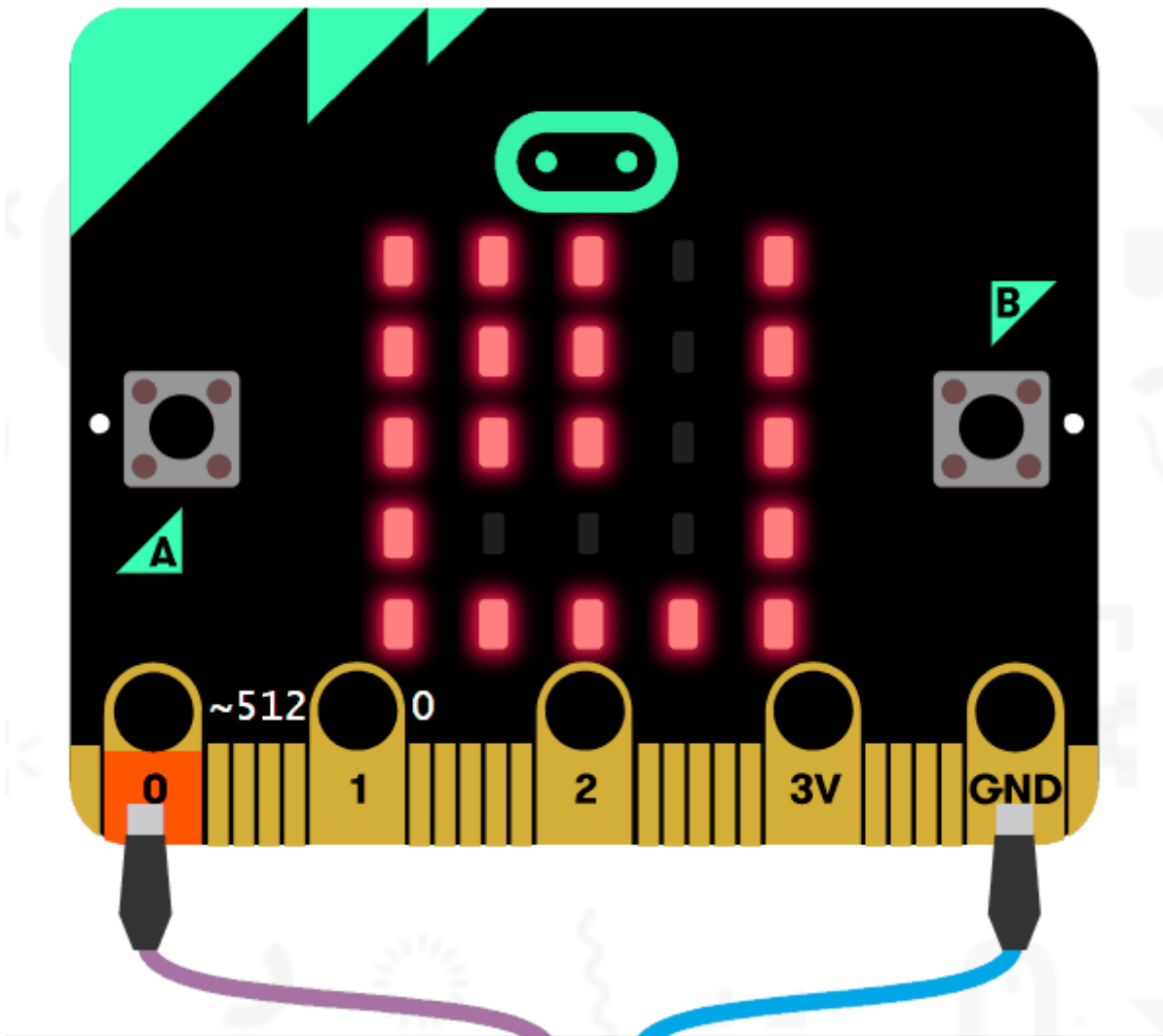
17.1. Goals

We're going to create a maze game where you must navigate a series of mazes without running into the walls. You'll learn how to:

- Use buzzer, ADKeypad and micro:bit board
- Use if statements to evaluate conditions
- Use variables to track game states such as player location
- Customize your game and add your own levels!

17.2. Materials

- 1 x BBC micro:bit
- 1 x Micro USB cable
- 1 x Buzzer
- 2 x F-F Jumper Wires
- 1 x ADKeypad

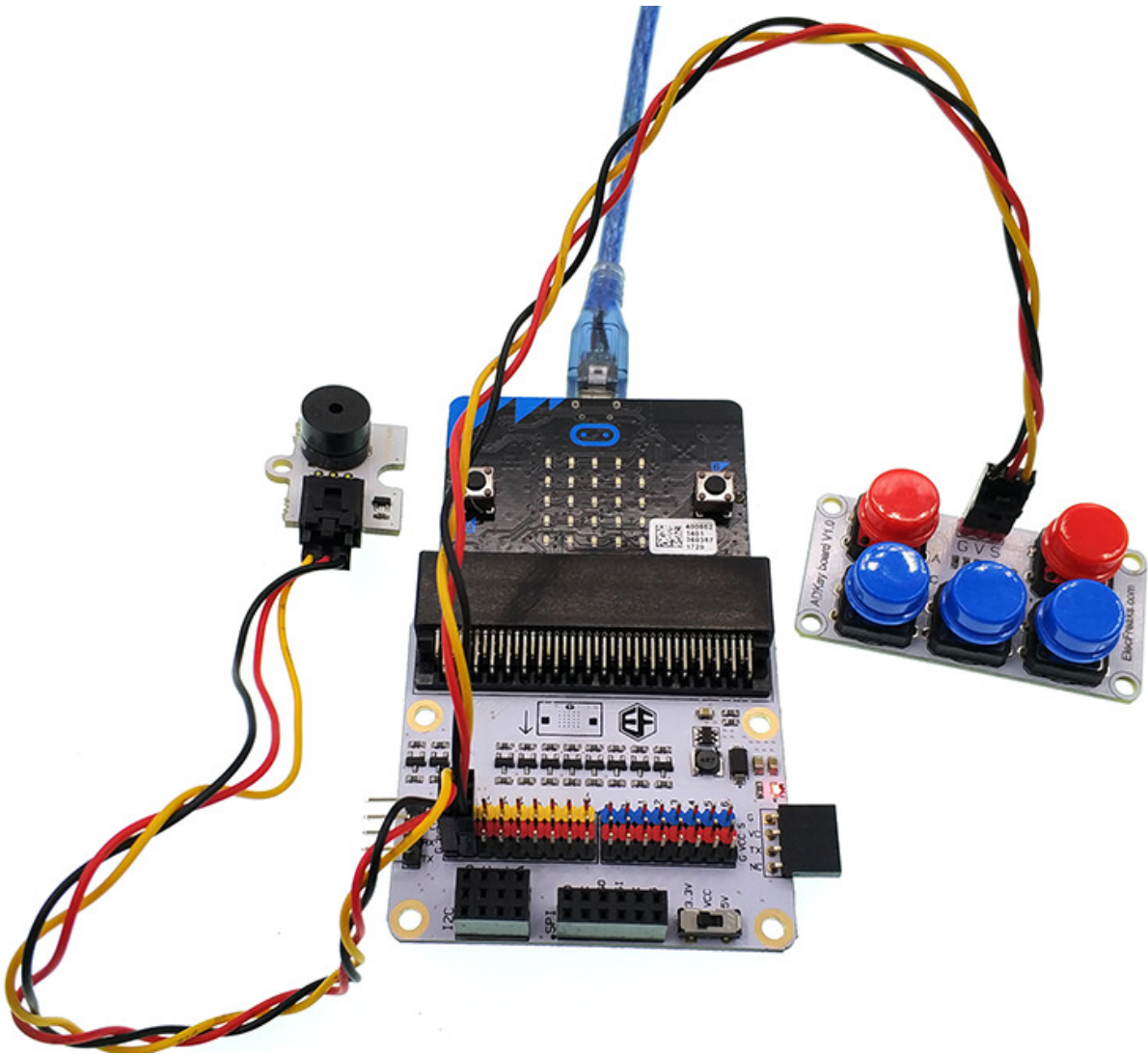


17.3. How to Make

Step 1

Plug in Buzzer to Pin0. Connect the positive lead to the yellow signal pin and the negative lead to the black ground pin on the breakout board.

Plug in the ADKeypad to Pin1. Match wire colors to pin colors on the breakout board!



Step 2

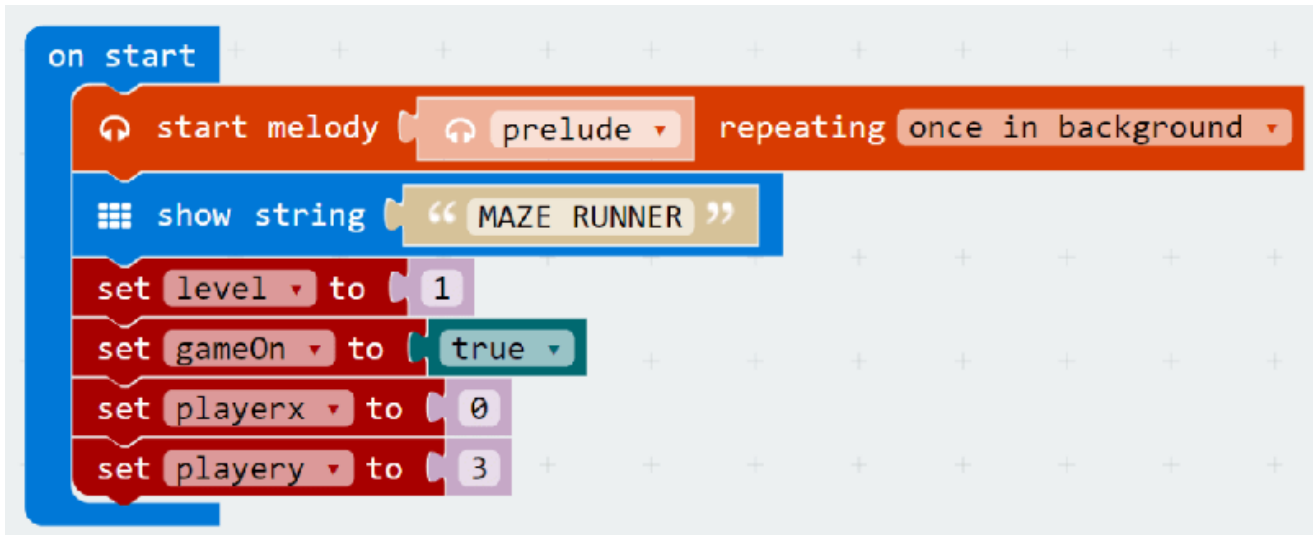
How to create a maze game on the micro:bit? We are going to display player's position, LED walls and the maze paths with LEDs on the micro:bit board.

How to keep track of the player's location on the screen? The 5*5 LED dot matrix on micro:bit can be regarded as a coordinate system. The x coordinate axle starts from 0 on the left to 4 on the right. The y coordinate axle starts from 0 on the top to 4 on the bottom. This means that the upper left LED is $x=0, y=0$. Likewise, the bottom right LED is $x=4, y=4$.

To do this, we need to create a few variables. Variables are like buckets that store pieces of information for us. Whenever we want the stored information, we can just look at the variable. We need two variables to keep track of the player's location. Why two variables? One will keep track of the player's x position and the other will keep track of the player's y position. We also need a variable to keep track of the maze level (yes, we can have multiple levels!) and also another variable to keep track of whether the game is active (opposite to game over).

So let's set these up. Inside 'on start' event, we (optionally) play a melody and display the name of the game on the micro:bit (MAZE RUNNER!). We also set up the 4 variables mentioned above, using the names: level, playerx, playery, and gameOn.

What do we set these variables to? We start at level 1 (of course), and we set gameOn to True because when we power on the micro:bit, we want to start the game right away. We can choose any starting point for our player location, but we'll need to remember this location later on when we set up our maze level (we don't want the player to start inside a wall!). In this example, I choose to start the player at x=0 and y=3.



```
on start
  start melody prelude repeating once in background
  show string "MAZE RUNNER"
  set level to 1
  set gameOn to true
  set playerx to 0
  set playery to 3
```

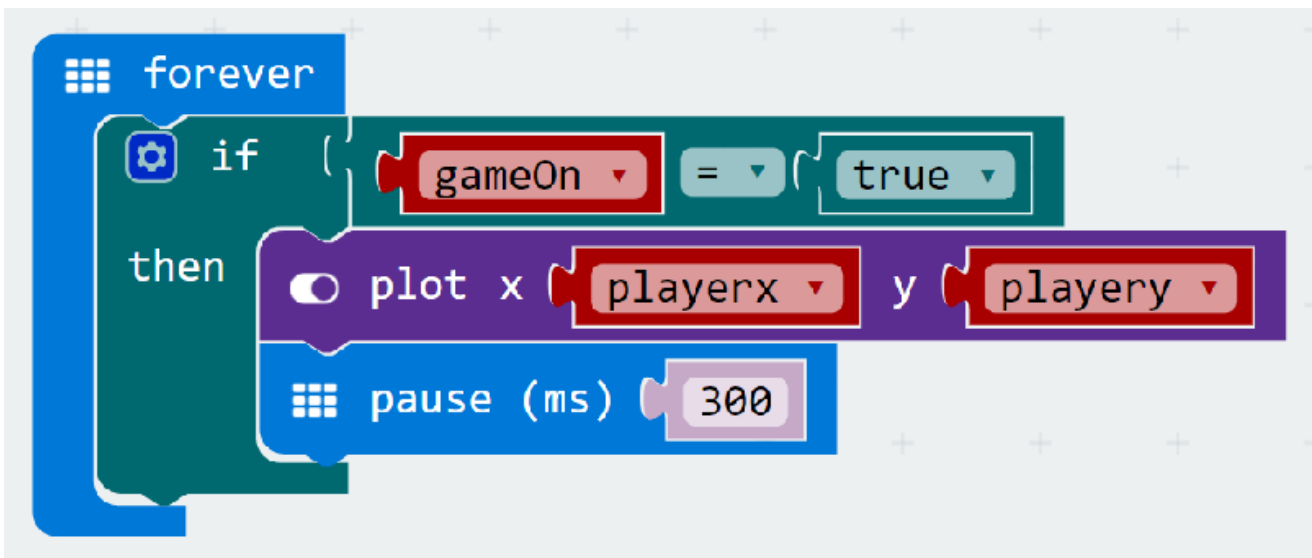
Step 3

Now that we've set up our starting variables, let's get our player to display on the micro:bit screen!

We want the player LED to blink on and off so that it is easy to be identified. To do this, we'll use the 'plot x y' block alternating with the 'pause' block inside a forever loop. Remember, we want the player to forever blink on and off! However, this won't work immediately. In step 6 when we add in the maze walls, the micro:bit will overwrite the player every time it draws the maze walls. By adding a pause block here, we make it so that the player won't immediately be re-plotted, creating a blinking effect.

We use the playerx and playery variables that we created above. Why? If we typed in numbers here, we wouldn't be able to easily make our player move! Using variables allows us to change the values of playerx and playery so that the forever loop will plot the new location of the player.

Remember the pause block is in milliseconds (so 300 ms = .3 seconds)! You can customize the speed at which it flashes by modifying the length of the pause.



Step 4

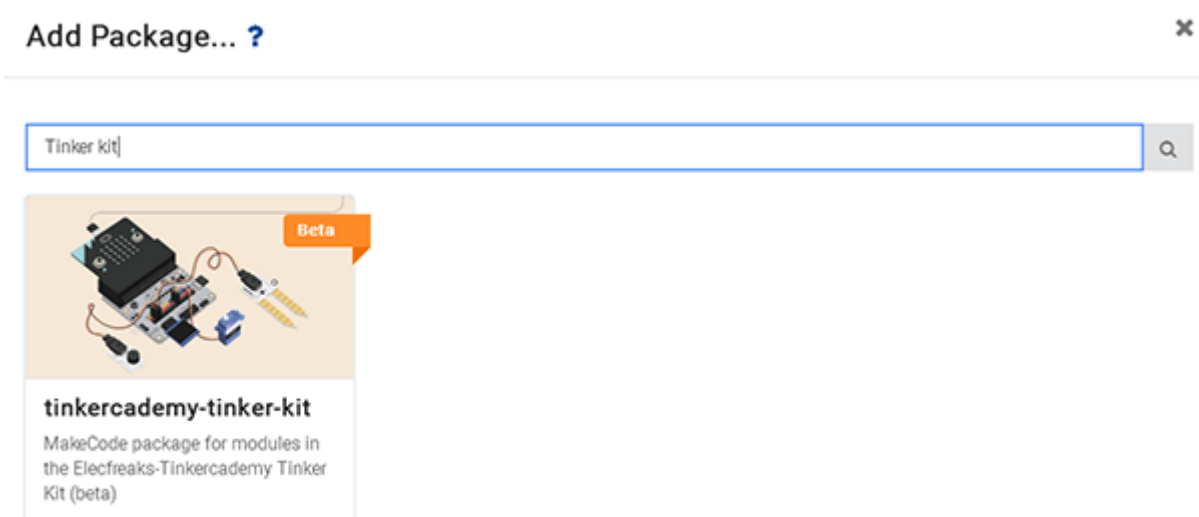
We've displayed the player on the micro:bit, but we can't move it yet! Let's add in player movement. We're going to sense button press on the ADKeypad. But to do so, we need to import a special package into MakeCode.

Expand the 'Advanced' section and scroll to the bottom and click on 'Add Packages'.

In the search box, type in "tinker kit". Click on the box labelled "tinkercademy-tinker-kit".

Now you'll see something new in MakeCode – a bright green Tinkercademy category has been added!

Inside this category you'll find blocks to sense button press on the ADKeypad. Note that importing this package only happens in the current project. So if you start a new project and want to use the category, you'll need to re-import it.



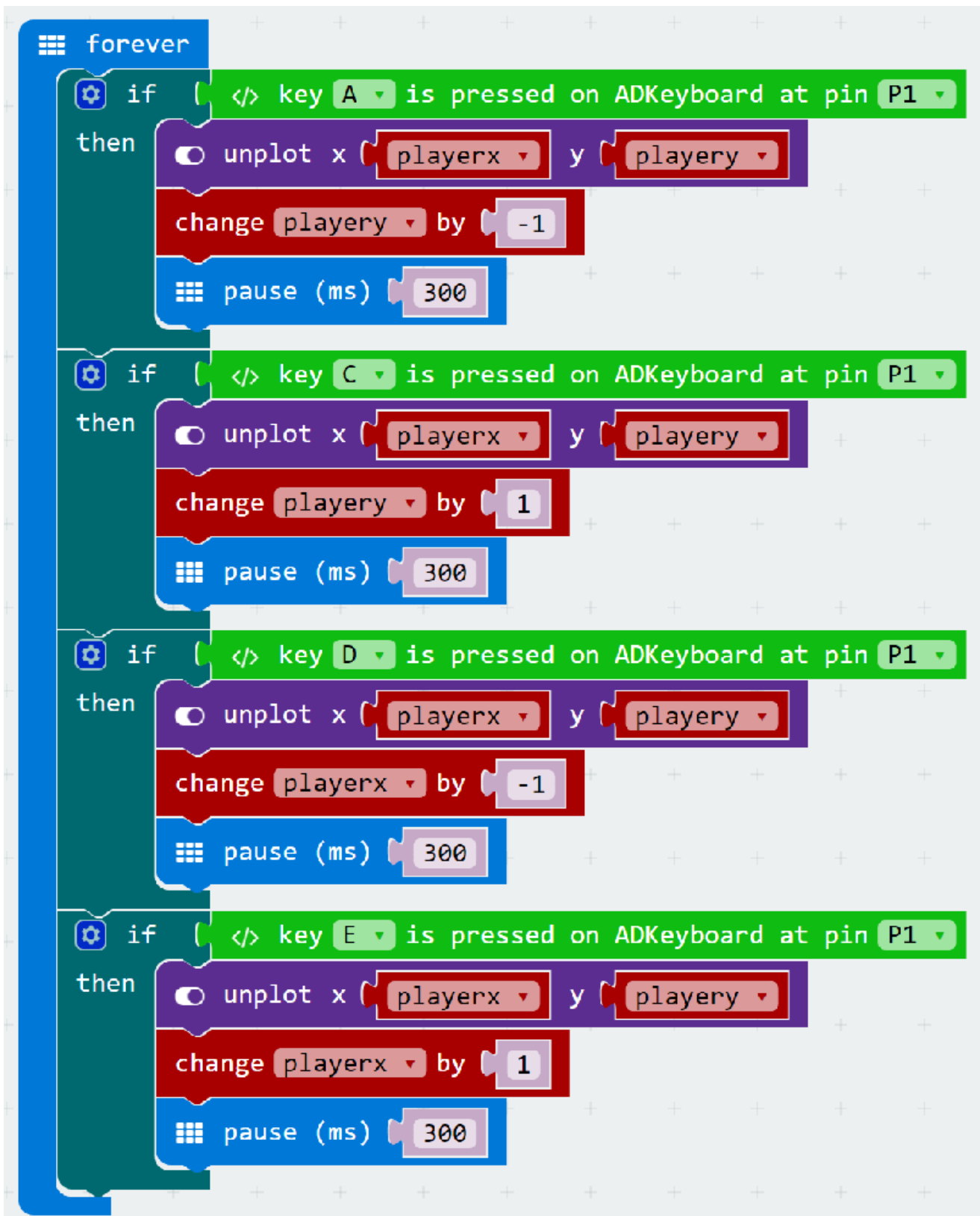
Step 5

Now that we have added Tinkercademy category, we can use the ADKeypad to move up, down, left, and right. In this example, we'll set button A to move up, button C to move down, button D to move left, and button E to move right.

To do this, we use if statements. If statements test if a condition is true. If it is true, then they run any blocks inside the if block. When we place an if statement inside a forever loop, we forever test if the condition is true.

To move the player, we simply change the player x or player y variables. Remember, decreasing or increasing playerx will cause the player to move left or right respectively. While decreasing or increasing playery will cause the player to move up or down respectively. We're constantly plotting the location of the player using these variables. So when we change them, it automatically changes the player's location!

We need to add a short 300ms pause after each button pressed, otherwise the player would move many spaces every time you pressed a button because the program runs so fast.



Step 6

Now that we can move the player, let's start creating our maze levels! Every time we start a level, we need to do a few things.

First we need to display the maze walls on the micro:bit screen; Second, we need to forever check if the player runs into a wall (if they do, it's gameover!). And third, we need to forever check if the player makes it to the end of the maze level (if they do, let them know they succeeded and move on to the next level!).

For each level, we're going to use a forever loop. Inside the loop, we use an 'if' statement to check if the level variable equals 1. This means this code will only ever run if the level variable equals 1.

Inside the if statement, we first display the maze walls. We light up LEDs to serve as maze walls, and leave them turned off to represent the maze path. This can be done using the 'show leds' block. One thing to be careful about though: remember above we set the starting position of the player? Make sure starting position of your player is not inside a maze wall! In this example, the starting position of the player is $x=0, y=3$.

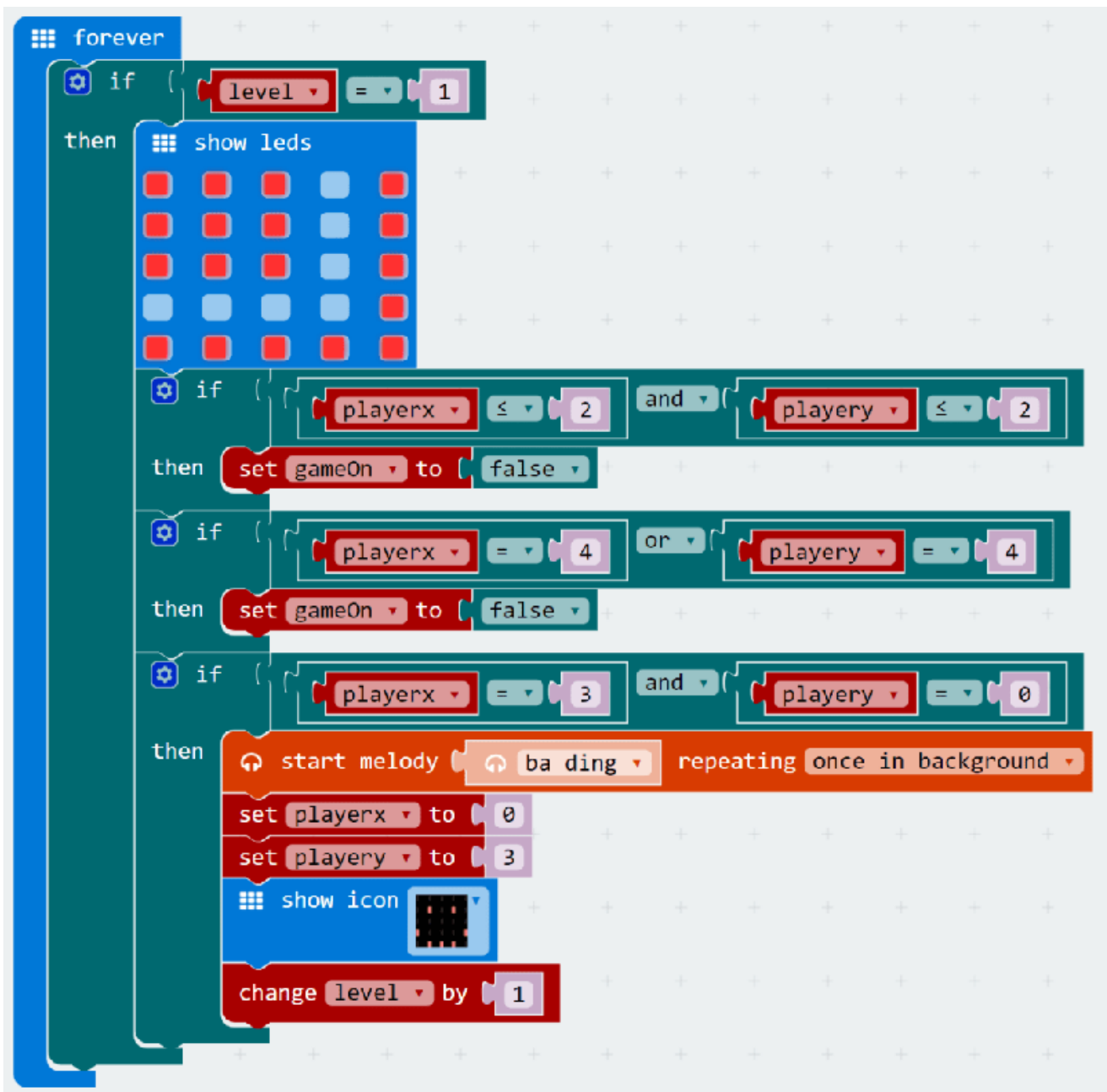
Next, we need to check if the player ever runs into a wall. How to do this? Once again we'll use if statements to check if our `playerx` and `playery` variables are ever in the same place as a wall. We do this using the coordinate system of the 5x5 LED grid. In this example, there are two sections of walls.

The first wall exists where both `playerx` and `playery` is less than or equal to 2. We create an if statement with these conditions, inside which we set `gameOn` to 'false' (since if it's ever 'true', it means the player ran into a wall and should get a Game Over).

The second wall exists where `playerx` or `playery` equals 4. We create another if statement with these conditions, and inside we set `gameOn` to 'false' (because once again if it's ever true, it means the player ran into a wall and game over).

Finally, the last test we need to add is to see if the player makes it successfully through the maze! In this example level, the end of the maze is at $x=3, y=0$. We create another if statement to check if $x=3$ and $y=0$, and inside we do a few things:

First, we play a success melody in the background; Second, we set the starting position of the player for the next level (in this example, we use the same starting position, but it can be different!). Third, we show a smile face to tell the player they succeeded! And fourth, we change the level variable by 1 (this will cause the next level to display).

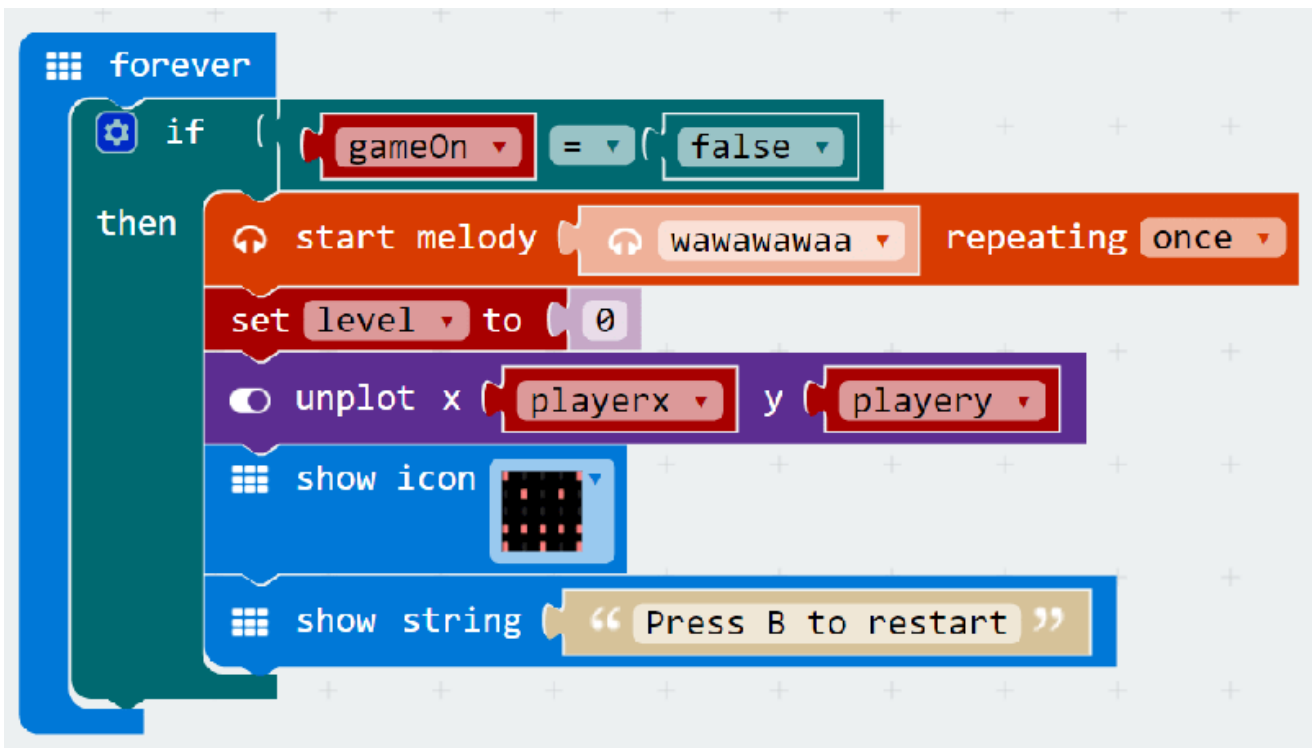


Step 7

Setting up a level costs us a lot of work! Now that we have a single level, let's make something happen when a player gets a game over. This will happen whenever they run into a wall, and it's tracked by the 'gameOn' variable.

Inside a forever loop, we use an if statement to check the value of the 'gameOn' variable. If it equals 'false', then we want our game over code to run!

In this example, we play a sad melody in the background, reset the 'level', unplot the player LED, show an angry face, and finally display a string telling the player they can press B to restart the game.



```
forever
  if (gameOn = false)
  then
    start melody (wawawawaa) repeating (once)
    set level to 0
    unplot x (playerx) y (playery)
    show icon (maze icon)
    show string ("Press B to restart")
```

Step 8

Speaking of pressing B to restart the game, we haven't yet created the code to do that!

Inside a forever loop, we test if button B on the ADKeypad is pressed. If it is, we want to set 'level' to 1, reset the player's starting location by setting the 'playerx' and 'playery' variables to 0 and 3 respectively, and set the 'gameOn' variable back to 'true'.



```
forever
  if (</> key B is pressed on ADKeyboard at pin P1)
  then
    set level to 1
    set playerx to 0
    set playery to 3
    set gameOn to true
```

Step 9

Now our game should work as intended! The only thing missed is more levels!

It's quite easy to add more levels by duplicating our level 1 code from above. The only thing that will change is the maze walls and the coordinates for our if statements (for testing if the player moves into a wall or completes the level).

Tips: sometimes it can be complicated to create if statements to test for every wall. In these cases, try to break down your walls into separate rectangles and create an if statement for each rectangle.

One thing to watch out: after the player has completed a level, you have to reset its `playerx` and `playery` variables, making sure the position matches your next level. Otherwise it would start inside a wall!


```
forever
  if (level = 2)
  then
    show leds
    if (playery = 0 or playery = 4)
    then set gameOn to false
    if (playerx = 0 and playery ≤ 2)
    then set gameOn to false
    if (playerx = 4 and playery ≤ 2)
    then set gameOn to false
    if (playerx = 2 and playery ≥ 2)
    then set gameOn to false
    if (playerx = 4 and playery = 3)
    then
      start melody ba ding repeating once in background
      show icon
      set playerx to 0
      set playery to 3
      change level by 1
```

```
forever
  if (level = 3)
  then
    show leds
    if (playerx = 4 or playery = 4)
```

```

then set gameOn to false
if (playerx = 0 and playery ≤ 2)
then set gameOn to false
if (playerx ≤ 2 and playery = 2)
then set gameOn to false
if (playerx ≥ 2 and playery = 0)
then set gameOn to false
if (playerx = 1 and playery = 0)
then
  start melody ba ding repeating once in background
  set playerx to 0
  set playery to 3
  show icon [Nyan Cat]
  change level by 1

```

Step 10

Once you have done this, you can optionally create a victory section. In this example, once the player have successfully completed the first 3 levels and level equals 4, we unplot the player by playing a victory melody in the background, and showing a victory message!

```

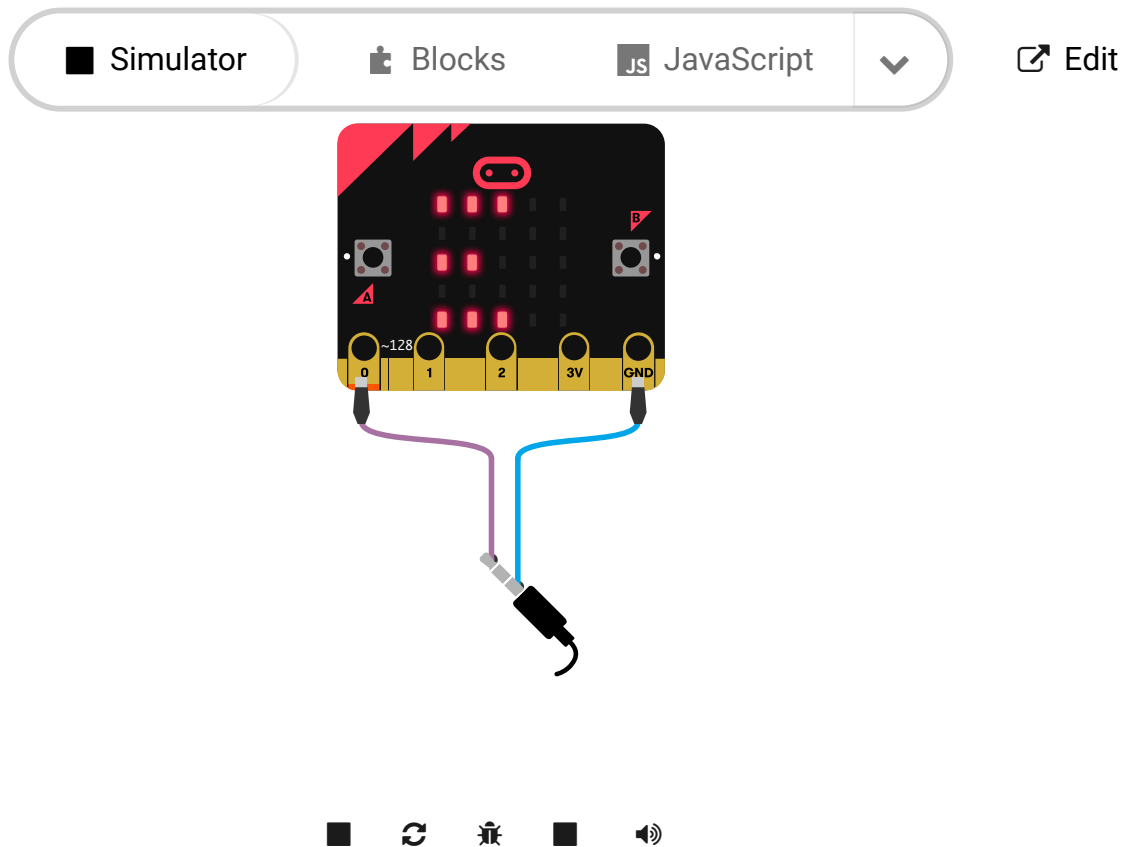
forever
  if (level = 4)
  then
    unplot x playerx y playery
    start melody nyan repeating once
    show string " You win! "
    show string " Press B to restart "

```

If you don't want to type these code by yourself, you can download directly from the link below:

https://makecode.microbit.org/_fCqa4399XUpv

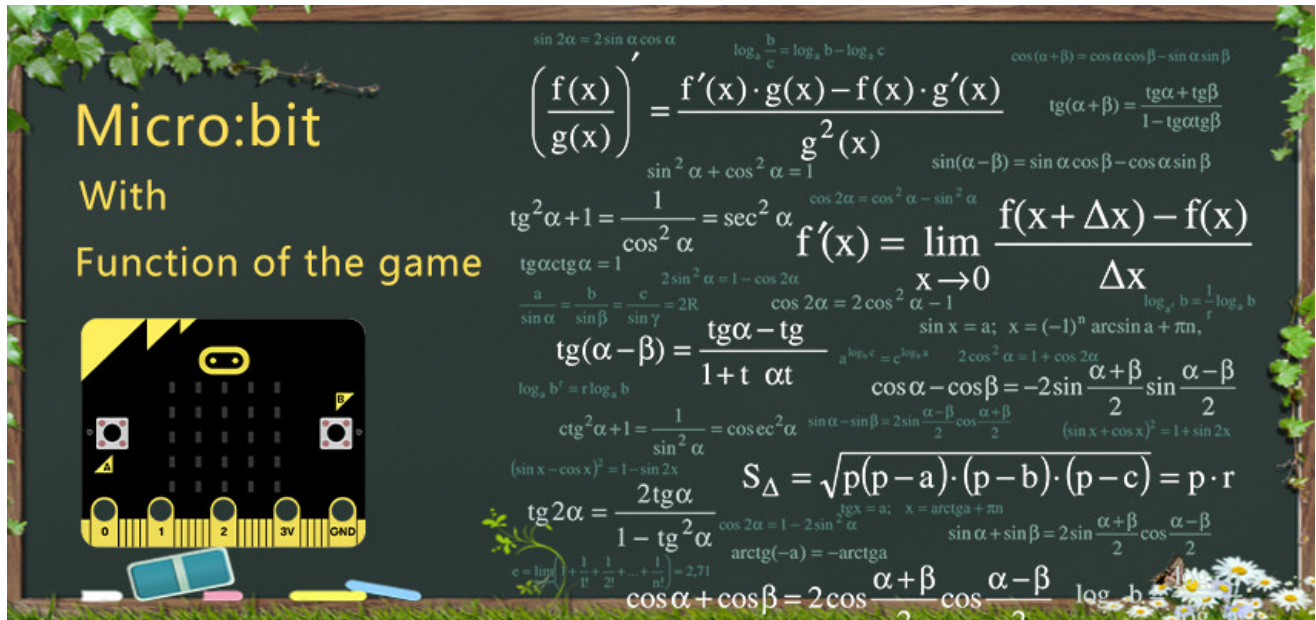
Or you can download from the page below:



17.4. Cool stuff!

Now that you've learned how to use the ADKeypad, you can try to control LEDs, servos, and other components! You've also learned about if statements which are useful in many micro:bit projects! Try to customize your maze runner game by adding more levels!

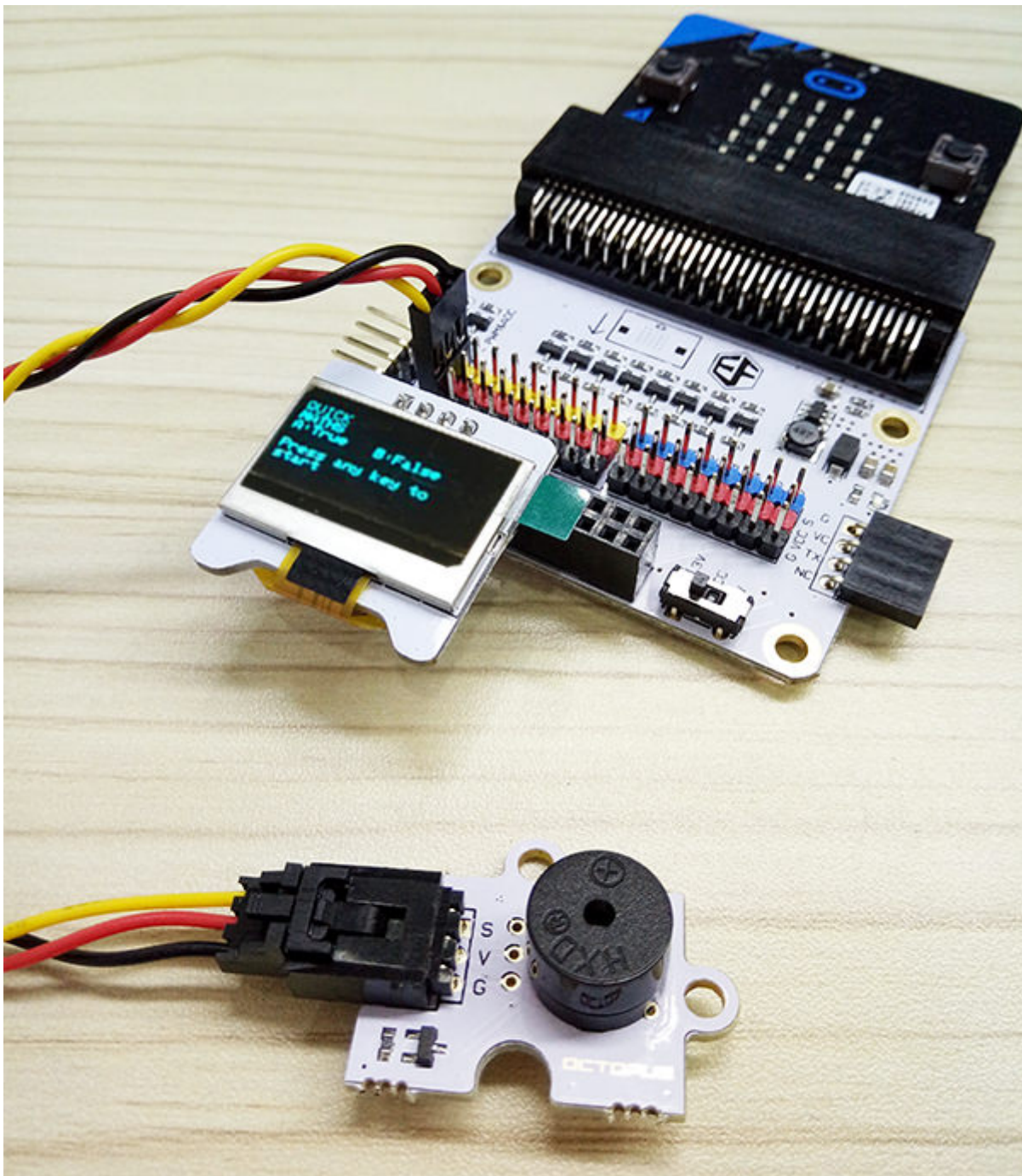
18. case 16 QUICK MATHS



QUICK MATHS is a game where its all about testing your mental calculation. Too slow, you lose; too fast, you may make mistakes.

18.1. Step 0 – Pre Build Overview

- Use a Buzzer and OLED with the micro:bit.
- Use if-else statements to evaluate conditions.
- Create your own function on MakeCode.

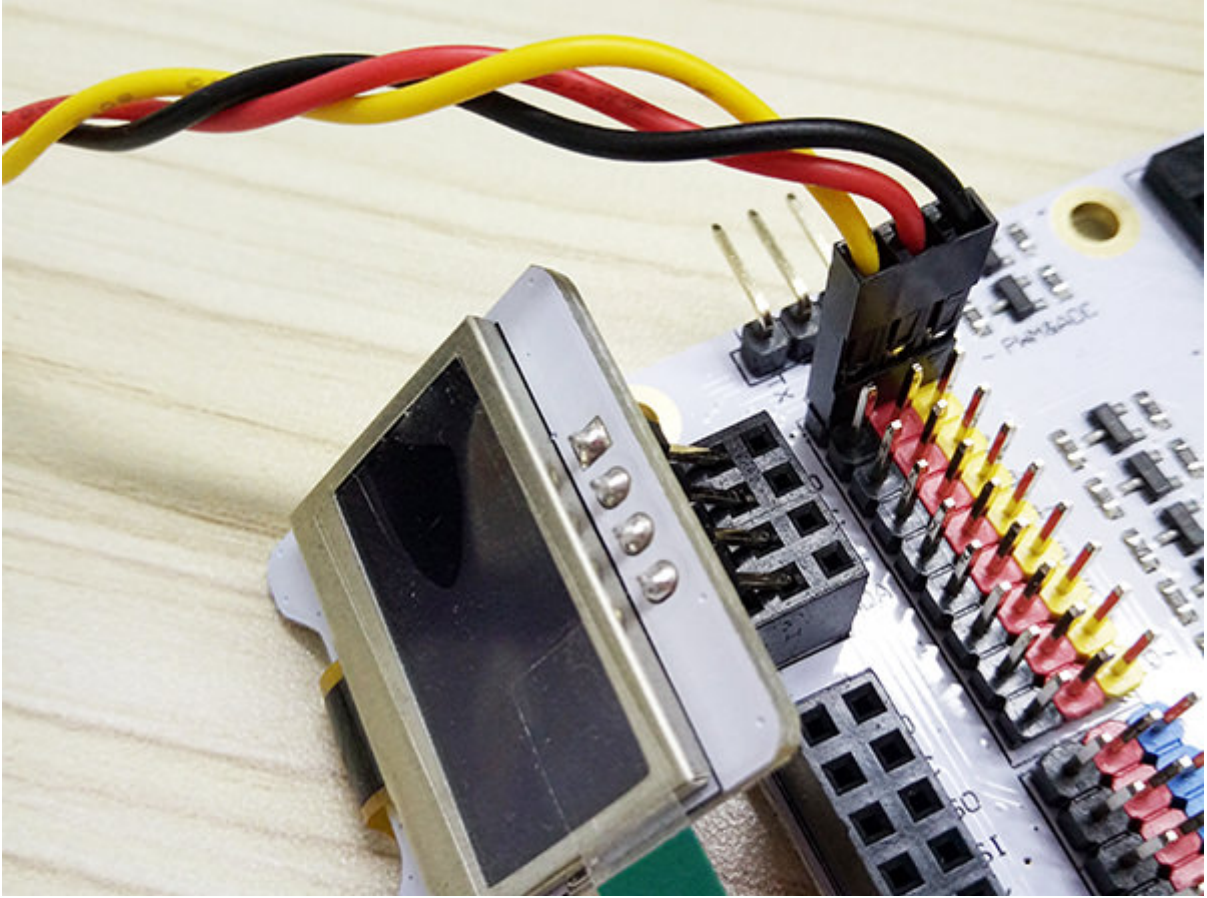
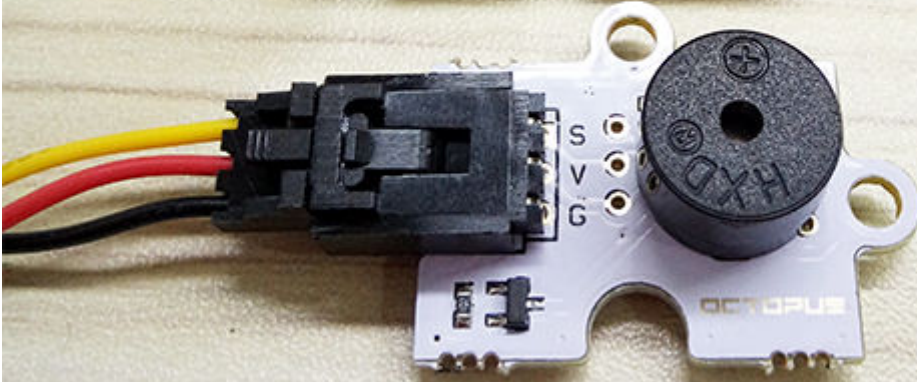
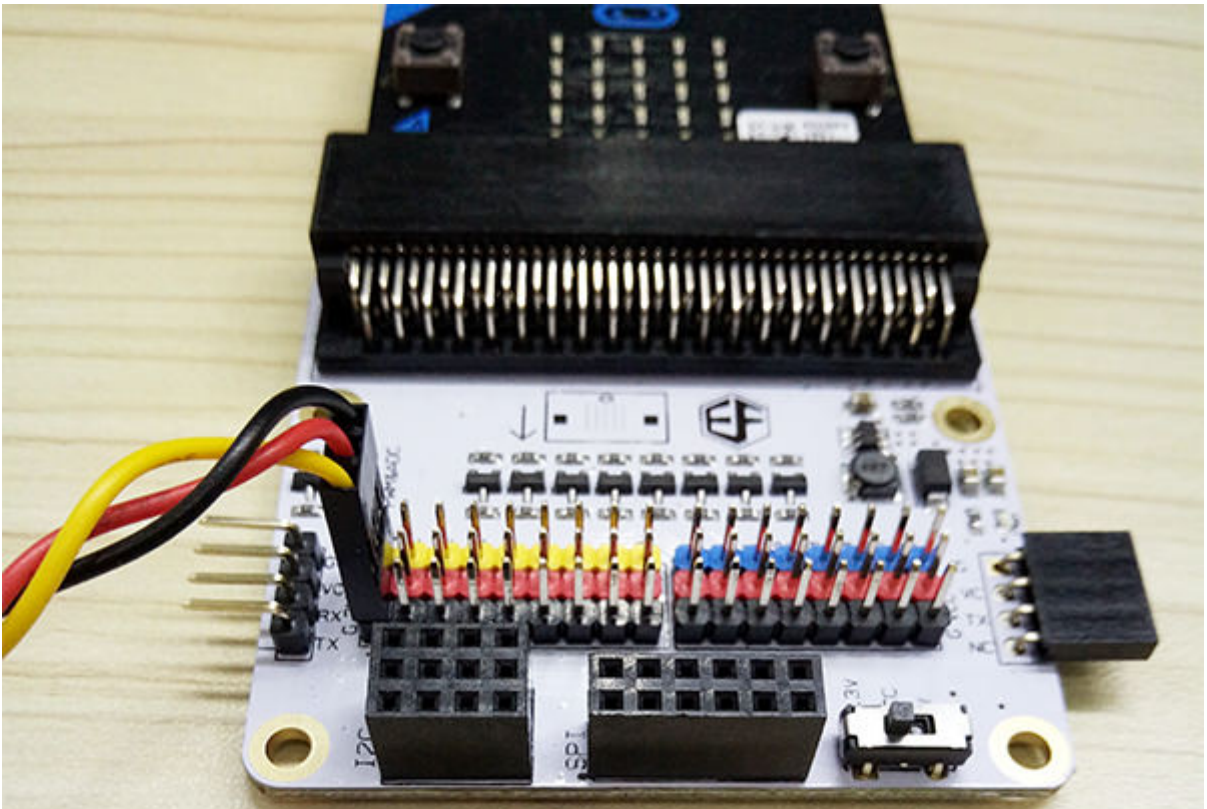


18.2. Materials required

- 1 x BBC micro:bit
- 1 x Micro USB cable
- 1 x Buzzer
- 1 x Octopus:bit
- 1 x OLED

18.3. How to Make

Step 1 – Components





Connect the buzzer to P0.

Plug in the OLED as shown in the picture above. You can plug it into any of the three rows.

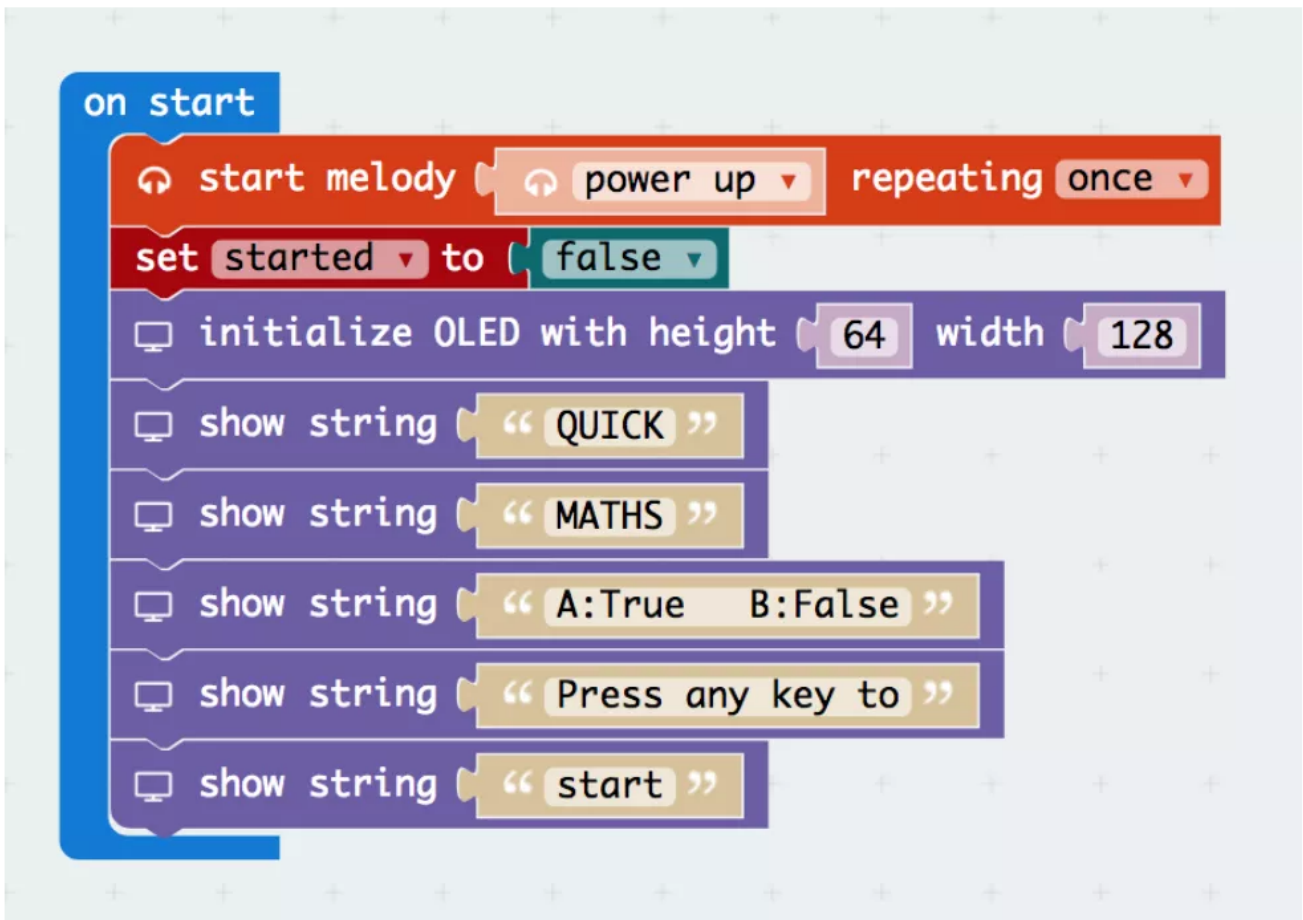
Step 2 – Pre-coding

The screenshot shows the Tinkercad interface. On the left, the Code Drawer is open, showing various code sections. The 'Advanced' section is highlighted with a red box, and the 'Add Package' button at the bottom is also highlighted with a red box. On the right, the 'Add Package...' dialog box is open, showing a search bar with 'OLED' entered. Below the search bar, two packages are listed: 'oled-ssd1306' and 'iot-environment-kit'. Below the dialog box, there is a text instruction: "This will open up a dialog box. Search for OLED. Click on the search icon or press enter, then select the oled-ssd1306."

We'll need to add a package of code to be able to use our kit components. Click on Advanced in the Code drawer to see more code sections and look at the bottom of the Code Drawer for Add Package.

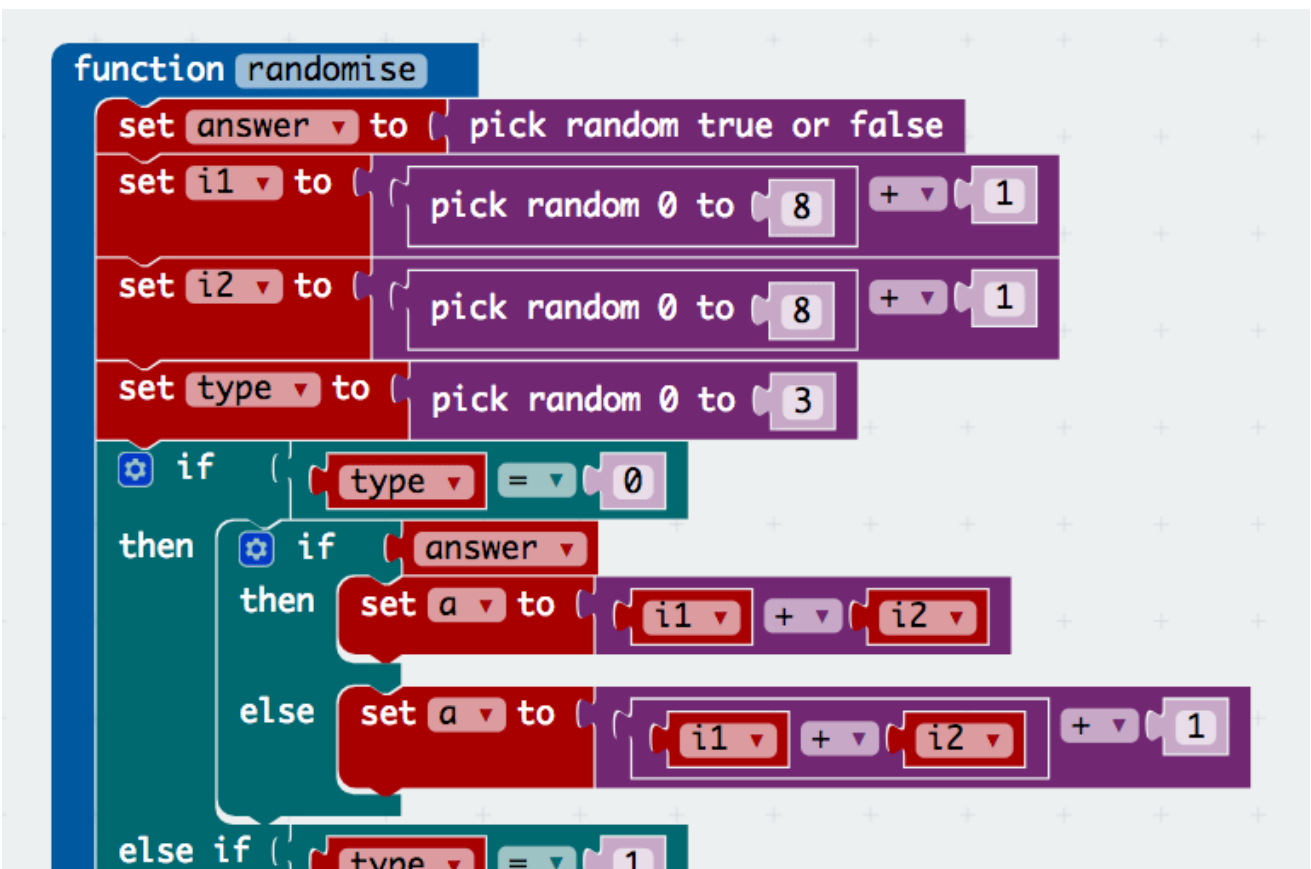
This will open up a dialog box. Search for OLED. Click on the search icon or press enter, then select the oled-ssd1306.

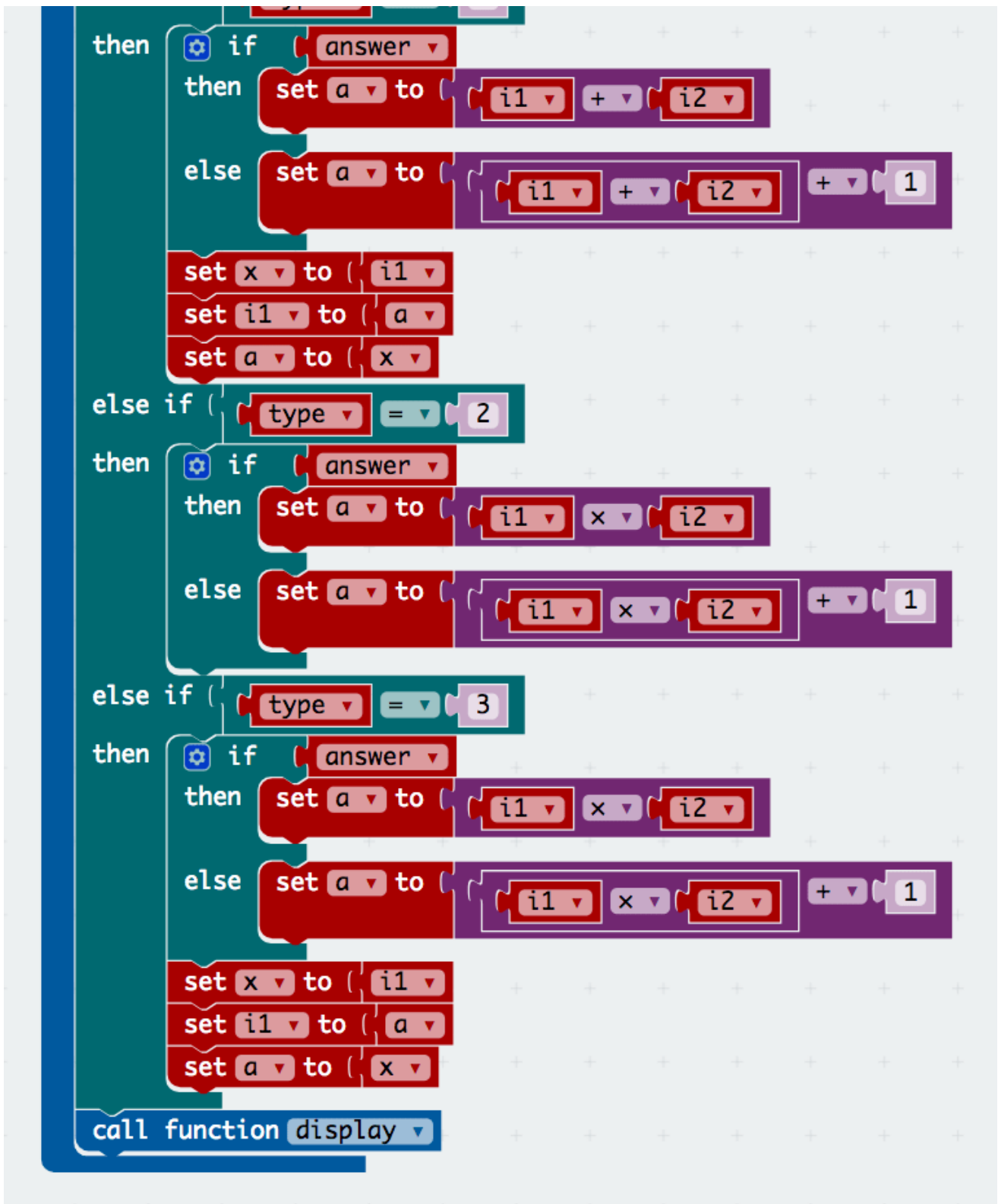
Step 3 – Coding Initial Screen



From the music section start the power up melody, this is the game's introduction music. After that, create a variable named started and set it to false, as the game has not started. Finally, use the blocks under the Tinkercademy section to initialise the OLED and display the messages as shown in the picture above.

Step 4: Coding Randomiser Function





In this step, we will randomise the questions that will be displayed. Firstly, create a variable named answer. In this, we will store whether the answer should be true or false. We determine this by using the randomiser block under Math.

Next, we create 3 more variables – i1, i2 and a. $i1 + i2 = a$ this is an example of what these variables would be used for. We then assign a random value from 1 to 9 to i1 and i2. The value of a would be set later.

After this, we create a variable name type, which will be used to store what type of question this is. (0: Addition, 1: Subtraction, 2: Multiplication, 3: Division) type would then be given a random number from 0 to 3 using the block under math.

From here, there is an if-else statement that checks what type of question it is in order to generate an answer.

For Addition (0), if the answer for this is supposed to be true, we set a to the sum of i1 and i2. However, if this is supposed to be false, we add 1 to correct answer. For Subtraction (1), if the answer for this is supposed to be true, we set a to the sum of i1 and i2, afterwards we swap the values of a and i1. However, if this is supposed be false, we add 1 to i1. For Multiplication (2), if the answer for this is supposed to be true, we set a to the product of i1 and i2.

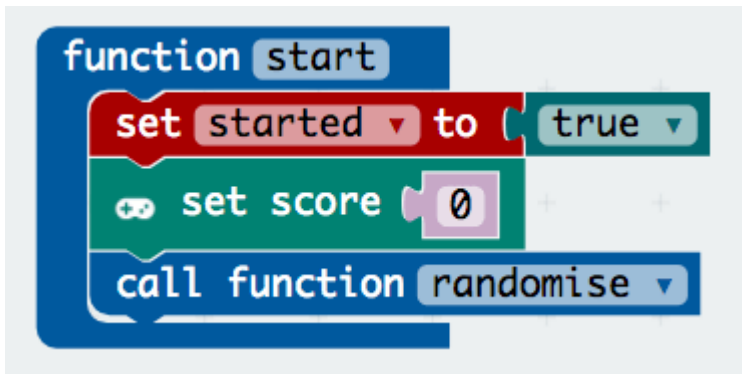
However, if this is supposed to be false, we add 1 to correct answer. For Division (3), if the answer for this is supposed to be true, we set a to the product of i1 and i2, afterwards we swap the values of a and i1. However, if this is supposed be false, we add 1 to i1.

Step 5: Coding Display Function

```
function display
  initialize OLED with height 64 width 128
  if (type = 0)
    then set sign to "+"
  else if (type = 1)
    then set sign to "-"
  else if (type = 2)
    then set sign to "x"
  else if (type = 3)
    then set sign to "/"
  show string join i1 sign i2
  show string join "=" a
```

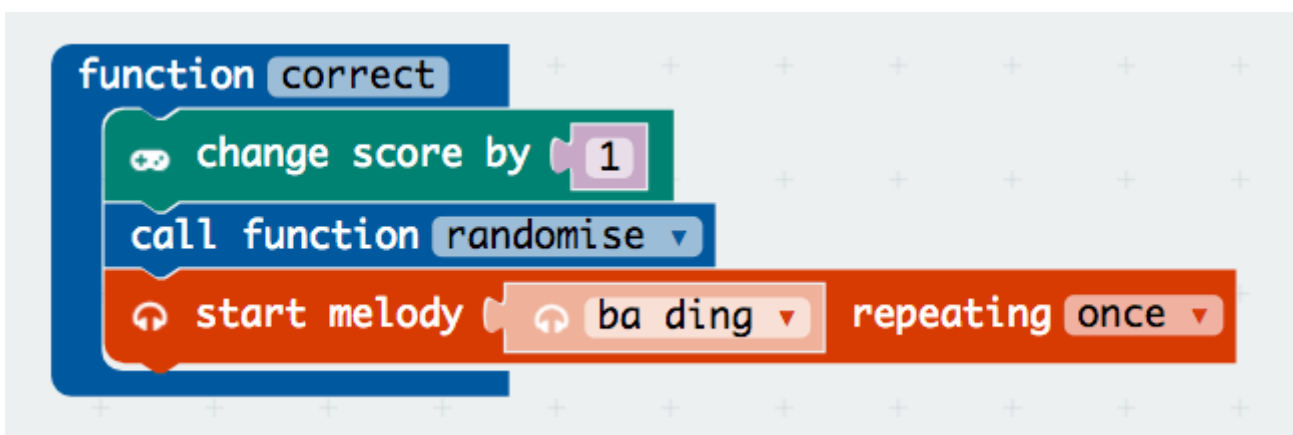
In this step, we will create a function that displays the question on the screen. First, initialise the OLED as shown in the picture. Next, we want to determine what sign to use, so we create a variable named sign. Afterwards set the value of sign by using an if-else statement that checks what type of question it is. (0: +, 1: -, 2: x, 3: /) Now we have what we need to display the equation. Under the OLED section, select the show string block and add the variables i1, sign, i2. Now that the display function is done, call the function at the end of the randomise function, as you would want the question to be displayed after the values have been randomised.

Step 6: Coding the Start Action



Now that we can randomise the questions, it is time we start the game. Firstly, create a start function. In this function, set the start value to be true and call the randomise function. Afterwards, from the game section in **advanced**, add the **set score** block and set the value to 0. Now that the function is complete, add the 2 button pressed blocks under the input section for both buttons A and B. In both blocks, create an if-else statement to check if the game has started. If it hasn't, call function start.

Step 7: Coding Check Function

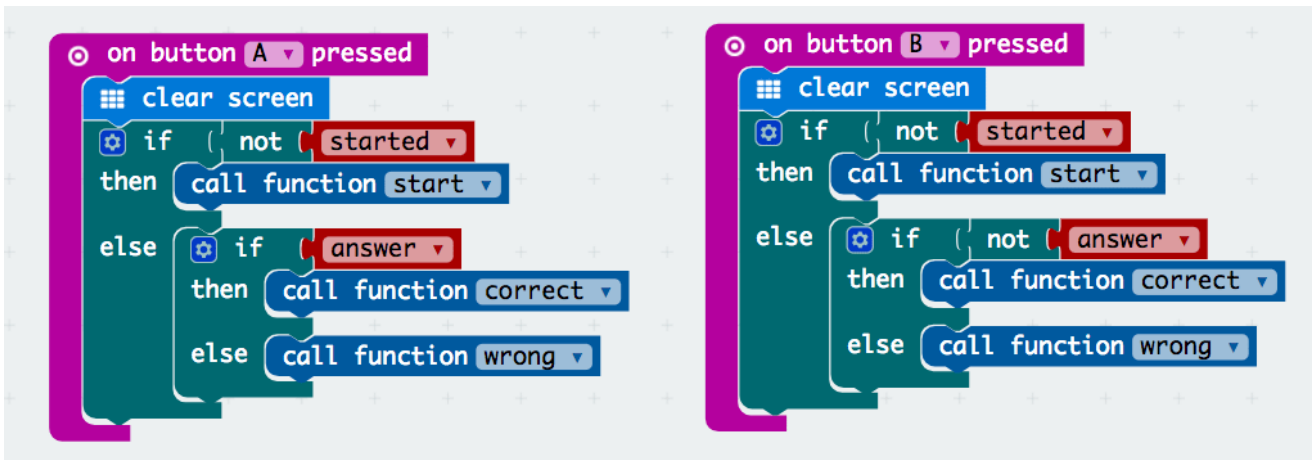


```

function display
  initialize OLED with height 64 width 128
  if (type = 0)
  then set sign to "+"
  else if (type = 1)
  then set sign to "-"
  else if (type = 2)
  then set sign to "x"
  else if (type = 3)
  then set sign to "/"
  show string join i1 sign i2
  show string join "=" a

```

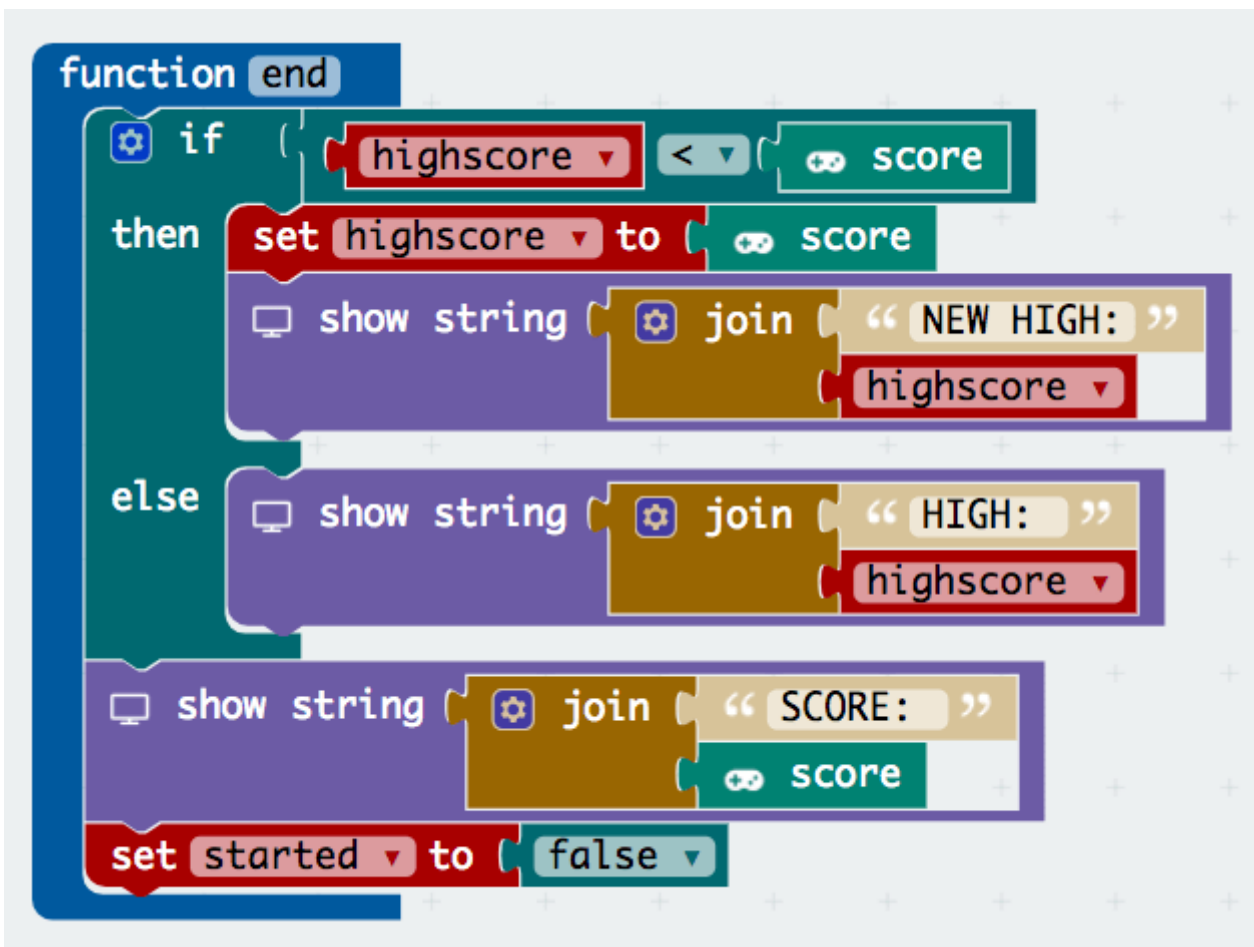
Continuing from step 6, now we will be working on the part if the game has started. Firstly, create 2 functions – correct and wrong. In the correct function, select the change score block from under the game section and change the score by 1. Next call the randomise function to get the next question and lastly start melody ba ding that repeats once for additional sound effects. Moving on to the wrong function, start melody wawawawaa repeating once and show icon X to indicate that the player has chosen the wrong answer. Afterwards, initialise OLED as shown in image. Lastly, we want to check the high score. Start off by creating a high score variable. Next create an if-else statement as shown above, in this logic gate we are checking if the score is higher than the player’s high score. If it is, then the high score value will be set to the current score. Don’t forget to set the start variable to be false when the game has ended. Now that we have completed the correct and wrong functions, we need to call them as shown in the image.



Bonus step 8: Coding Timer

Now you have a functioning game. But to make things more exciting, we should add a timer. Before we dive into that, there are a few things we have to do first.

Firstly, create an end function. Set up in the function accordingly. This may seem familiar as this is the last part of the wrong function. You can replace that portion by calling this function. This function would be called again to prevent reprogramming.



Next, create a slow function. This would be called if the player does not answer in time. Create the blocks as shown in the image.

```

function slow
  start melody wawawawaa repeating once
  show icon [LED icon]
  initialize OLED with height 64 width 128
  show string " "
  show string " T0000 "
  show string " SLOW! "
  show string " "
  call function end

```

Following that, we have to create a new variable called time. Time stores when the player started a specific question. Afterwards, set its value to the running time (ms) block which can be found under the more tab of the input section.

```

function start
  set started to true
  set time to [running time (ms)]
  set score 0
  call function randomise

```

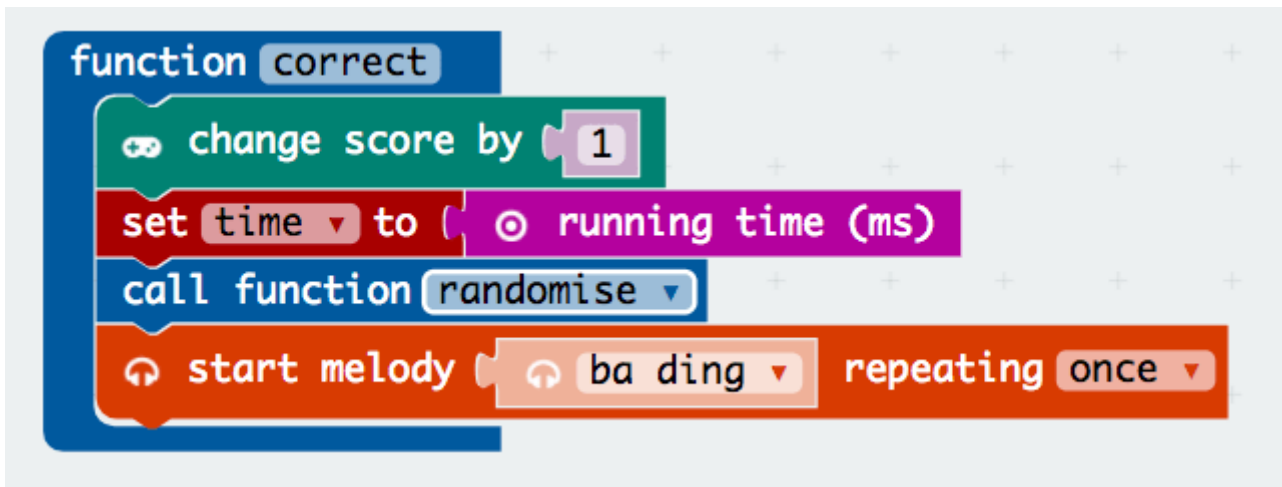
```

forever
  if (started and [running time (ms)] - time > 2500)
  then call function slow

```

Micro:bit does not have a built-in timer, thus we have to design one by using what they offer. Now we know the starting time is when the player started the question, and running time is how long the program has been running. From this, if we subtract them we get how long the

player has spent on that question. For this game, we only allow the player to have 2.5s (2500ms) for each question. Hence, they will lose if they are too slow.



```
function correct
  change score by 1
  set time to running time (ms)
  call function randomise
  start melody ba ding repeating once
```

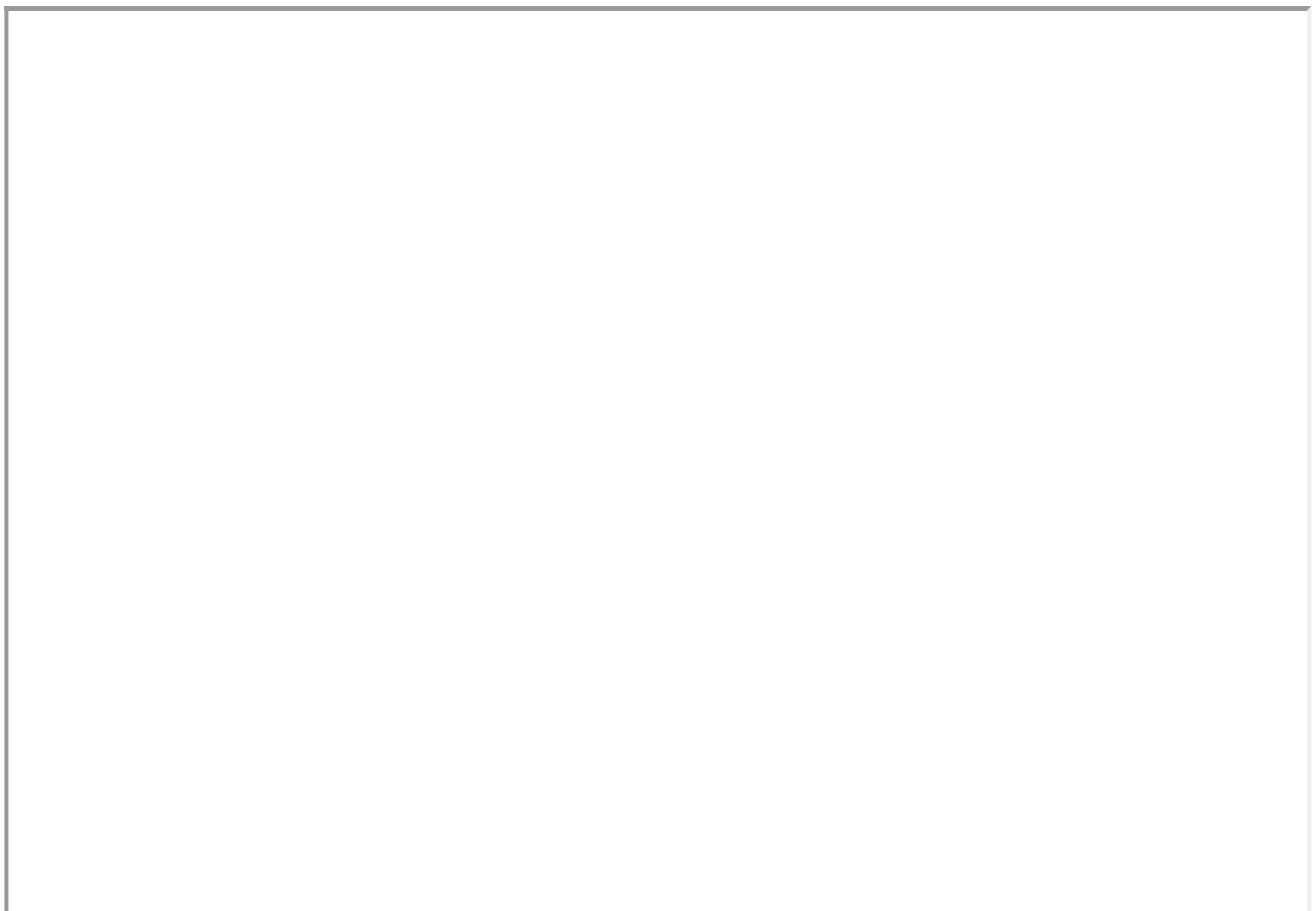
Lastly, in the correct function add a block that sets the value of time to current running time. This is to refresh the starting time for a new question.

That's it! You've officially completed this tutorial.

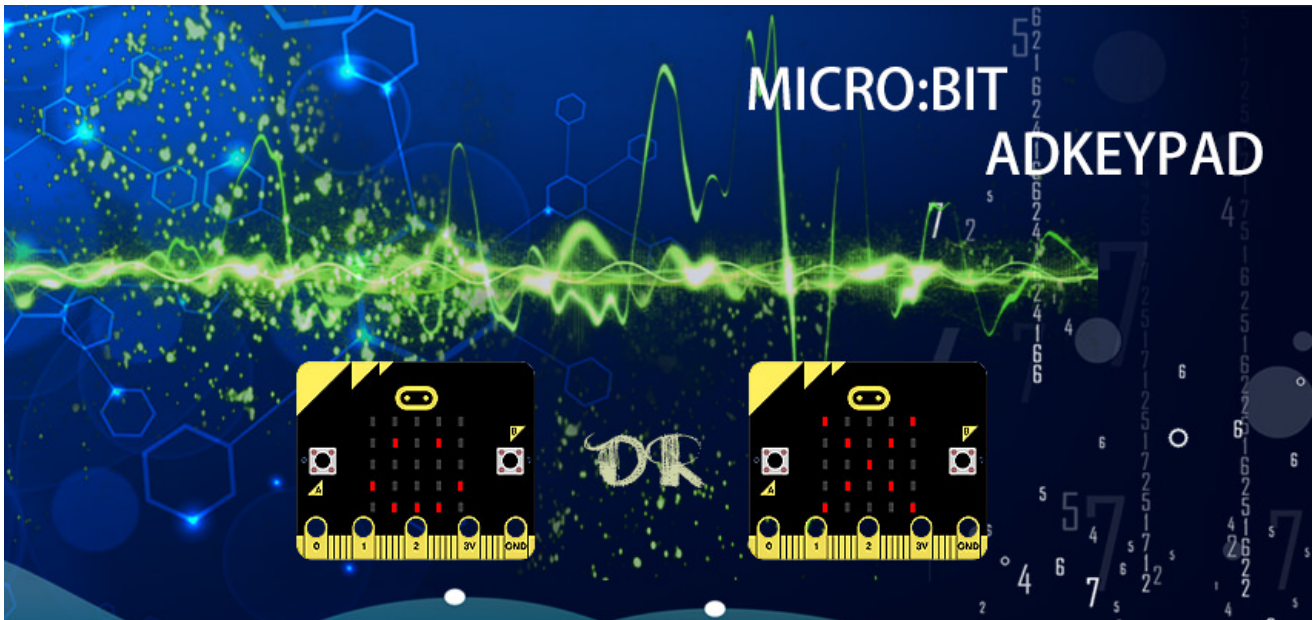
If you don't want to type these code by yourself, you can directly download the whole program from the link below.

https://makecode.microbit.org/_ThdfipEwFbWs

Or you can download from the page below.



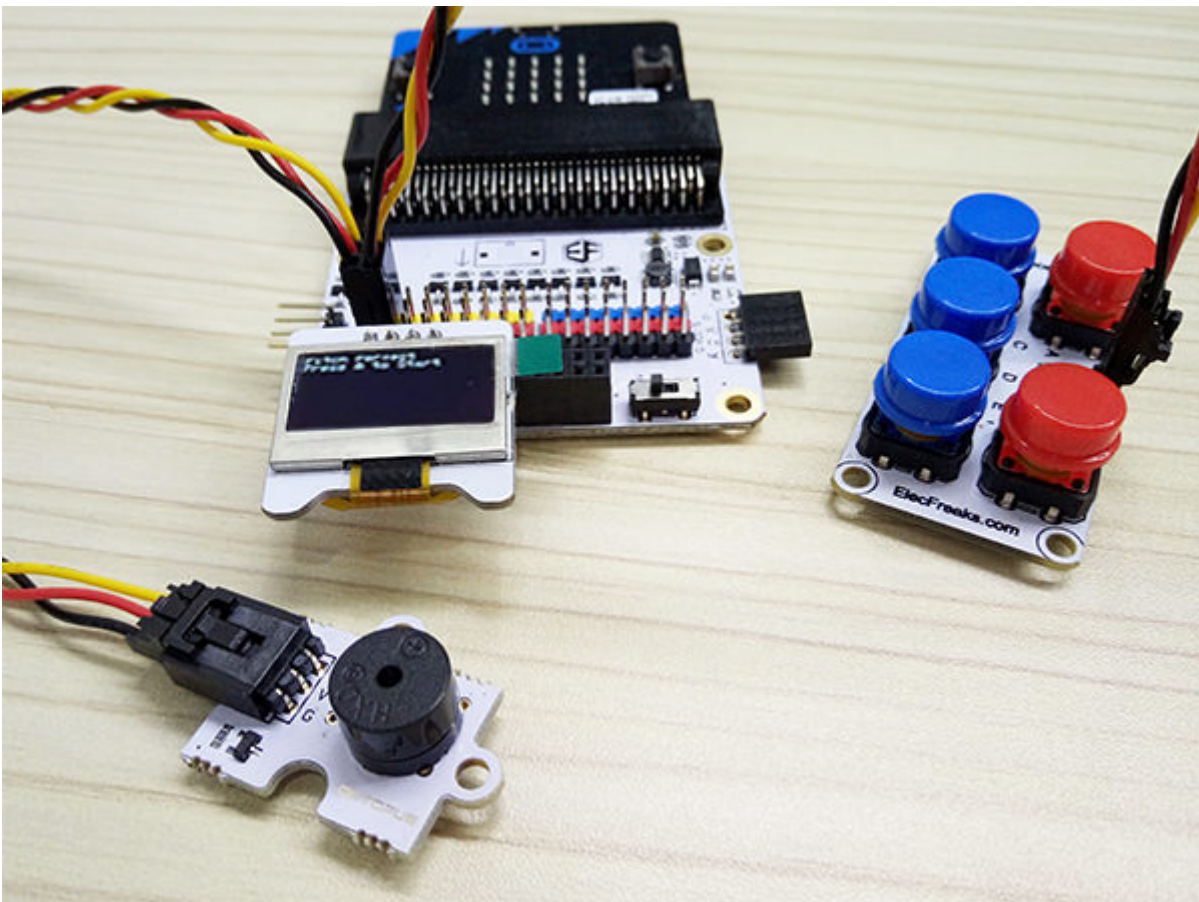
19. case 17 Pitch Perfect



Do you think your ears are pitch perfect, then try my game. Or even better, create one!

19.1. Goals

- Learn how to use a ADKeypad, the OLED screen and the buzzer.
- Make something with a ADKeypad, OLED screen and the buzzer.
- Learn the if-else statement functionality.



19.2. Materials Needed

- 1 x BBC micro:bit
- 1 x Micro USB cable
- 1 x Buzzer
- 2 x F-F Jumper Wires
- 1 x OLED
- 1 x ADKeypad
- 1 x Breakout Board

19.3. How to Make

Step 1:


Firstly, plug in your buzzer to Pin 0, making sure the positive side (usually the longer end) is connected to the yellow signal pin and the negative end is connected to the black ground pin on the breakout board.

Plug in the ADKeypad to Pin 1, making sure that the colour of the wire and breakout board matches. Then, attach the OLED screen at the bottom left socket of the breakout board.

Step 2:

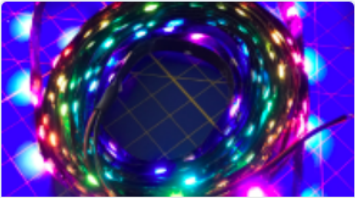
Add Package... ?



Search or enter project URL... 

devices
Camera, remote control and other
Bluetooth services

bluetooth
Bluetooth services




neopixel
AdaFruit NeoPixel driver

We will need to add a package to the code editor to use the kit components. Click on the advanced in the micro bit text editor and you will see a section that says Add Package.

This will open up a dialog box. Search for OLED. Click on the search icon or press enter, then select the oled-ssd1306.

Add Package... ?

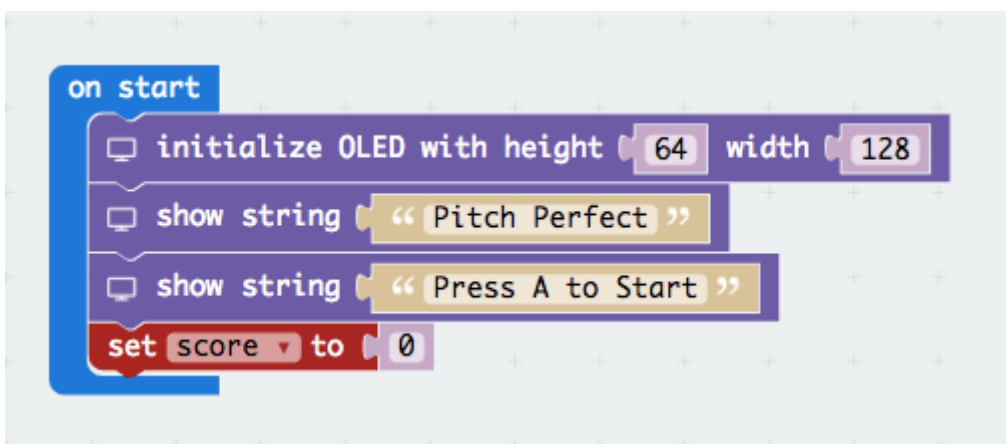


<https://pxt.microbit.org/50544-64675-33322-24641> 

SSD1306_OLED
Tinkercademy package for SSD1306
OLED controller

Note: If you get a warning telling you some packages will be removed because of incompatibility issues, either follow the prompts or create a new project in the Projects file menu.

Step 3:



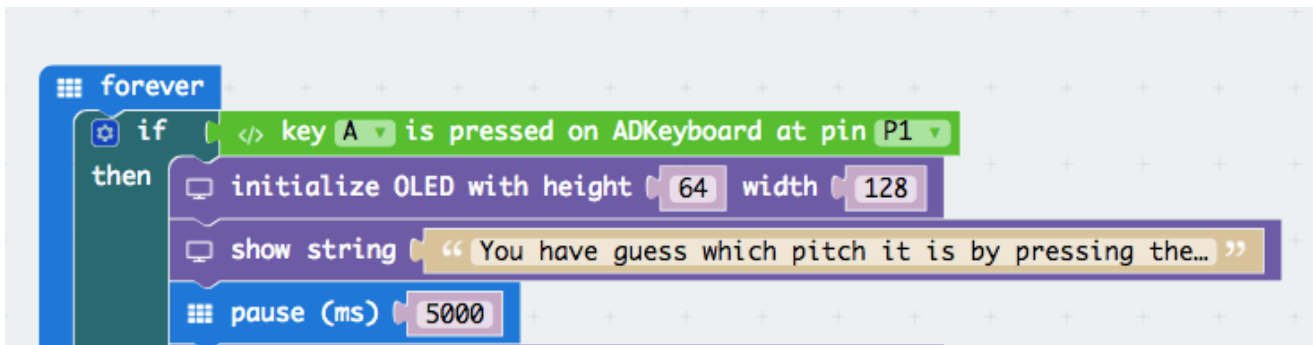
```
on start
  initialize OLED with height 64 width 128
  show string "Pitch Perfect"
  show string "Press A to Start"
  set score to 0
```

First, you have to initialise the OLED screen to a height of 64 and width of 128 so to run the screen in the proper sizing.

Next, you have to set a variable starting score to 0 for the initial play. This means you have a score of 0 at the start of your game. Then you need the OLED display show a text of "Pitch Perfect".

You need to write a simple instruction on how to start. Thus, a simple sentence "Press A to start " will do.

Step 4:



Since at step 3 we wrote that you need to press A to start, we need to write a condition for it. A condition basically means a requirement for a program to start loading its instructions. Thus, an if-else statement of the A button being pressed would suffice. Moreover, this will be nested on a forever bracket.

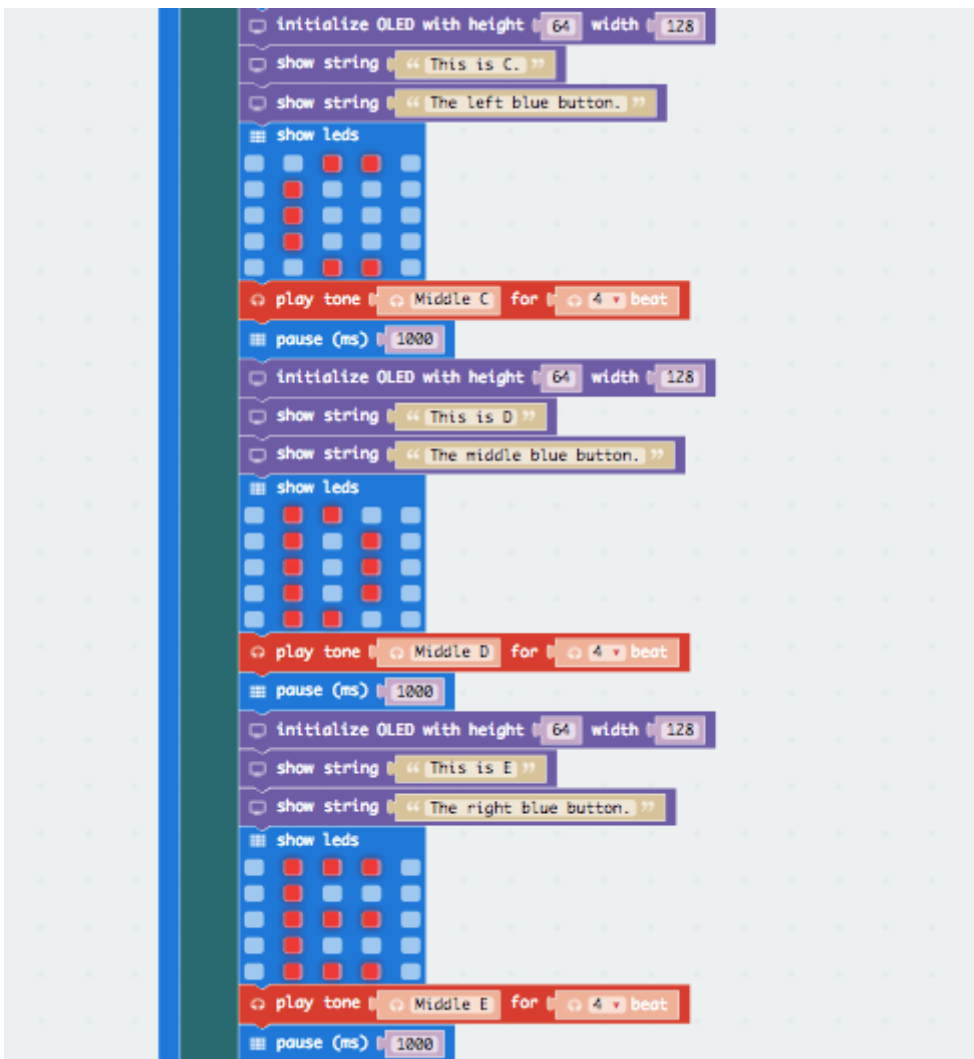
Next, you have to write another set of instructions on how to play the game. What I wrote was : "You have guess which pitch it is by pressing the correct key". Then you have to have a timeout around 5000ms (5 seconds) to let the user read the instructions.

Step 5:

You are gonna introduce the user to listen to the pitch of the sound being played. What I did was to play the pitch (for example: C) for 4 beat (4 seconds), flash the led on the MicroBit of the pitch C and OLED display on the screen itself.

After that, I will put in a timeout so the user can process the pitch to the correct alphabet and the ADKeyboard. Additionally, you can put in the OLED screen on what buttons are to be pressed for a certain pitch. Example, when Pitch C is being played, I wrote "Left blue button" to indicate that is the button.

If you are wondering, why the initialise OLED display and show string block is repeated, it is because it would simulate a refresh in web browser. If you do not initialise the display, the text would just be brought down instead of new text being created.



Step 6:

Once the user have gone through the mini-briefing of how the pitch sounds, you can get them ready. You can have a countdown for them to get ready on the game itself.

Now, you can build your pitch tests. So, to do that, you need to play a pitch and you can customise by displaying by any image on the MicroBit and a message "Key #1" at the same time.

Then, if the user pressed the correct button on the ADKeypad, they would get a point. If not, no points. Thus you set the variable score to change by 1 if the get it correct and otherwise, a -1. Thus, an if-else statement on whether the user pressed the correct button will do.

To let the user know if they got the correct answer, you can display of an image tick for a correct answer and a cross for a wrong answer.

Repeat this step so you can have many tests to play with!



```
initialize OLED with height 64 width 128
show string "Ready?"
show number 3
pause (ms) 1000
show number 2
pause (ms) 1000
show number 1
pause (ms) 1000

initialize OLED with height 64 width 128
show string "Key #1"
play tone Middle C for 4 beat
if key C is pressed on ADKeyboard at pin P1
then
  change score by 1
  show icon [3x3 grid of dots]
else
  change score by 0
  show icon [3x3 grid of dots]
```

Step 7:

Once you are contented with your tests, you can end the game by showing the latest scores. You can display in the OLED screen "Your score is:" with the variable score shown. Put a smiley for fun sake. And you are done! Enjoy the game.

```
start melody nyan repeating once
show icon
show string "Your score is:"
show number score
```

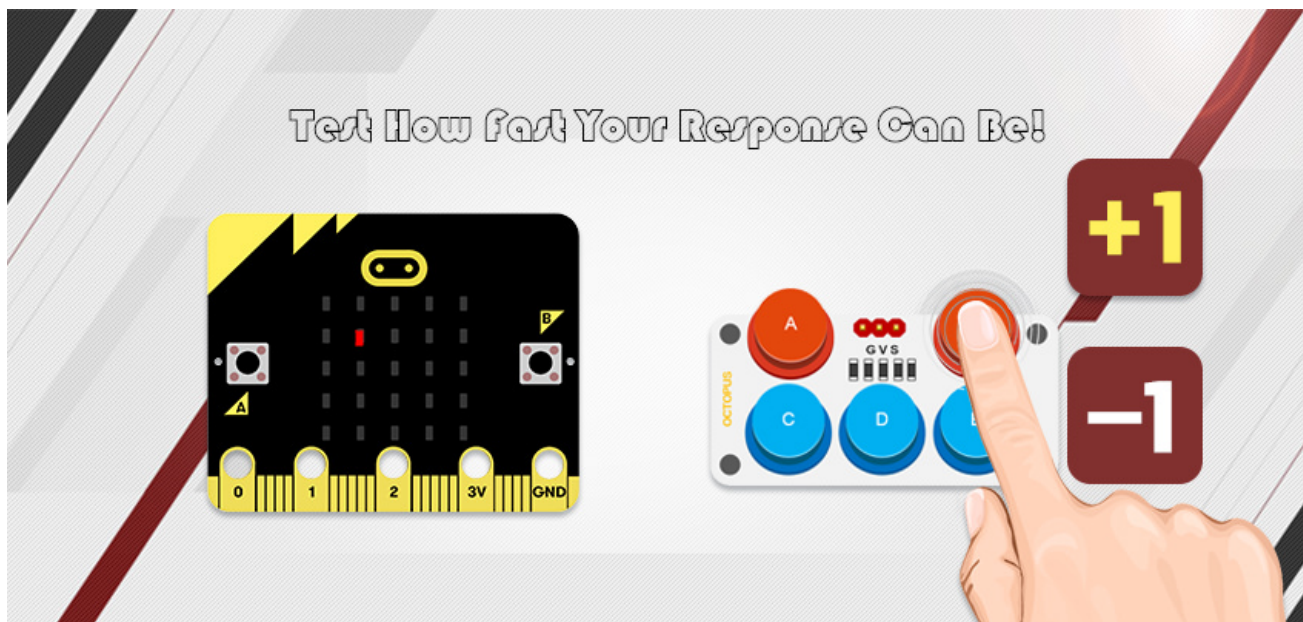
If you don't want to type these code by yourself, you can directly download the whole program from the link below.

https://makecode.microbit.org/_A26fCxRz1P1g

Or you can download from the page below.

The image shows the MakeCode Microbit simulator interface. At the top, there is a navigation bar with tabs for "Simulator", "Blocks", and "JavaScript". To the right of the "JavaScript" tab is an "Edit" button with an external link icon. Below the navigation bar is a 3D rendering of a black Microbit board. The board has a yellow top-left corner and a yellow bottom edge. The bottom edge has five pins labeled "0", "1", "2", "3V", and "GND". A purple wire is connected to pin "0", a blue wire to pin "1", and a black wire to pin "GND". These wires are connected to a black USB-C cable. Below the board rendering is a control bar with five icons: a square, a refresh symbol, a play symbol, a square, and a speaker icon.

20. case 18 Finger Dexterity

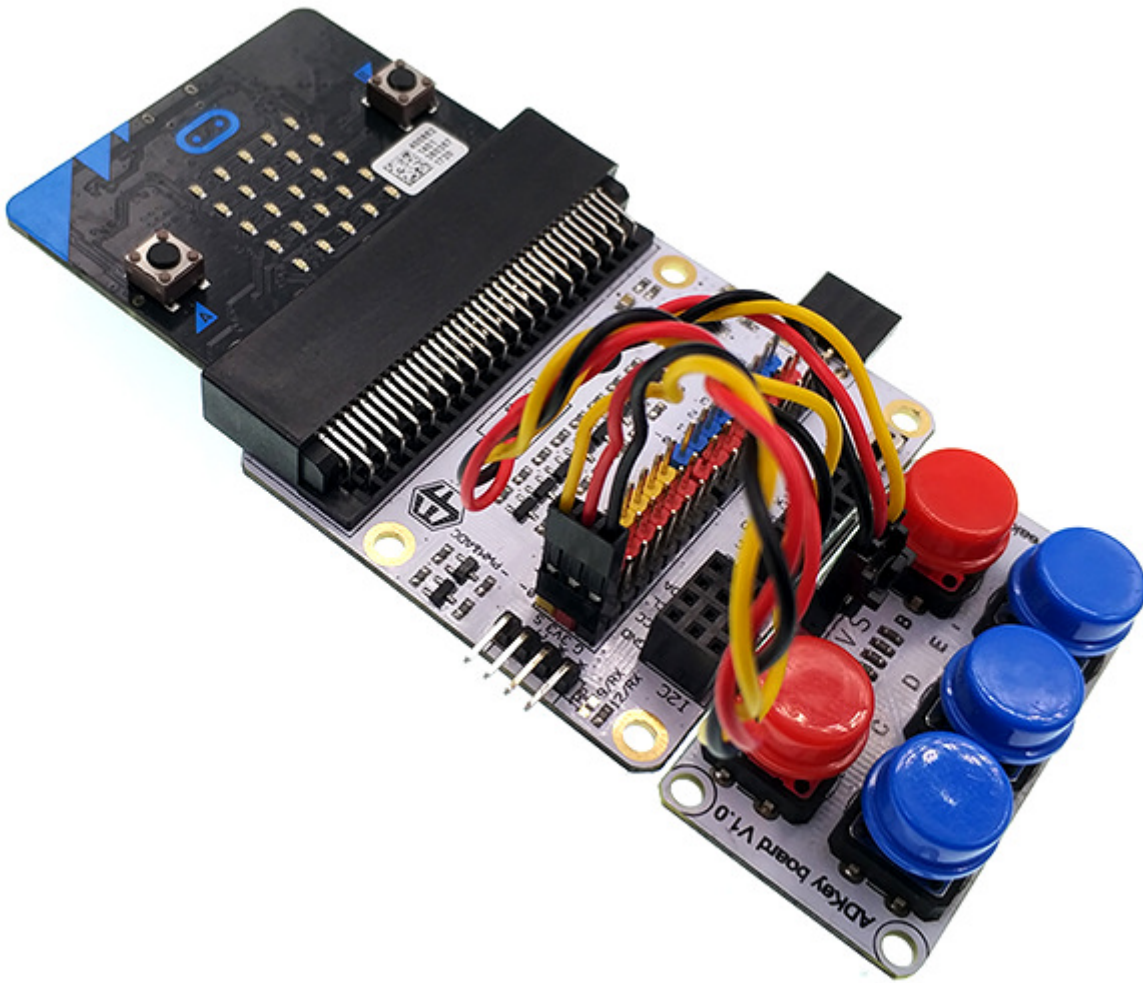


Are your psychomotor skills as bad as mine?

20.1. Goals

We are going to create a game where the player must click on a key (on the ADKeypad) that corresponds to the column on which a random LED lights up (A for the first column and E for the last). The pace at which the LED lights up gets quicker and quicker as the game goes on. You'll learn how to:

- use an ADKeypad with the micro:bit.
- use functions recursively.
- use while loops.
- improve your finger dexterity!



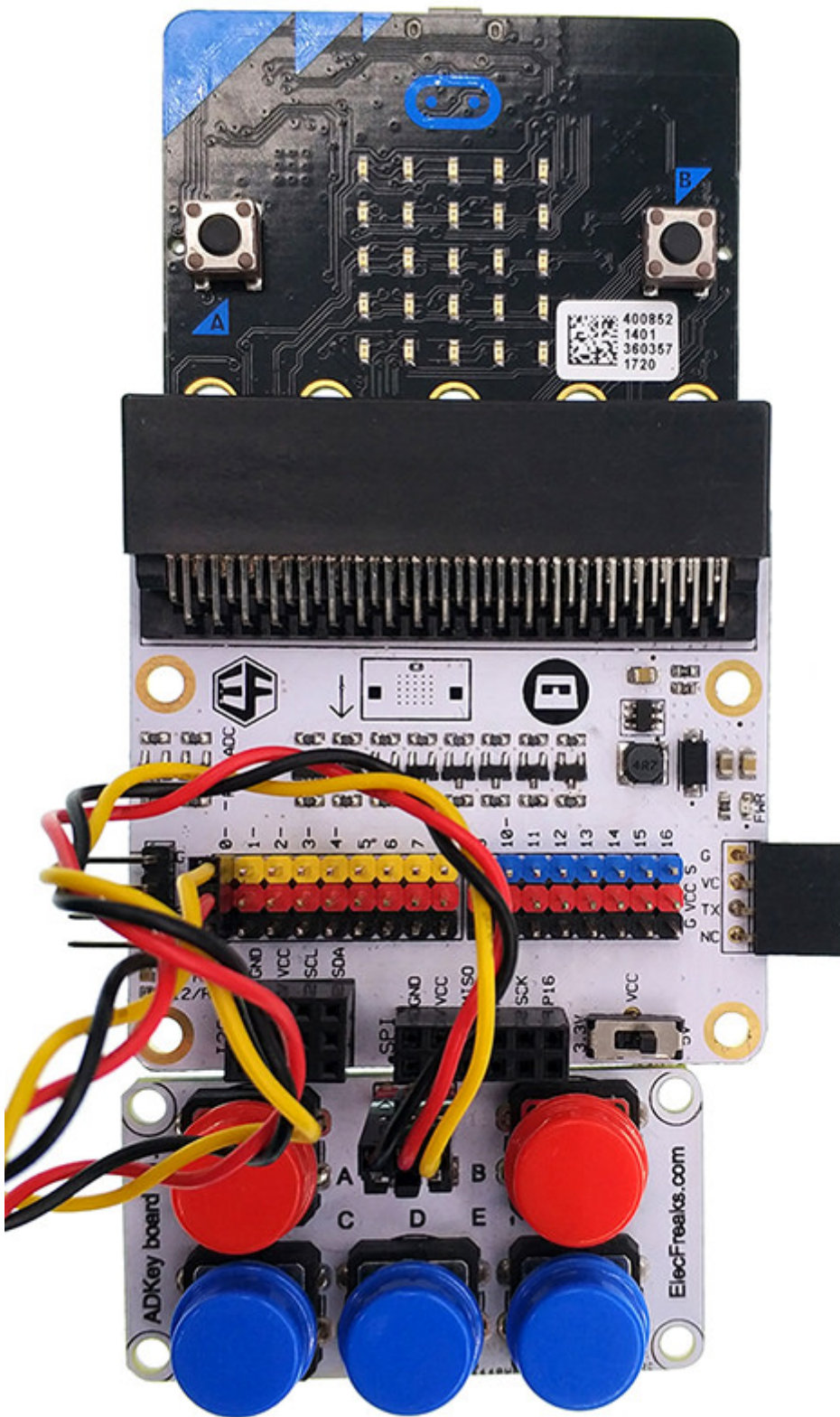
20.2. Materials and Pre-requisites

- 1 x BBC micro:bit
- 1 x Micro USB cable
- 1 x F-F Jumper Wires
- 1 x ADKeypad Or ElecFreaks Micro:bit Tinker Kit (contains all components in the above.)

You also need some experience about if-else statements, variables etc.

20.3. How to Make

Step 1

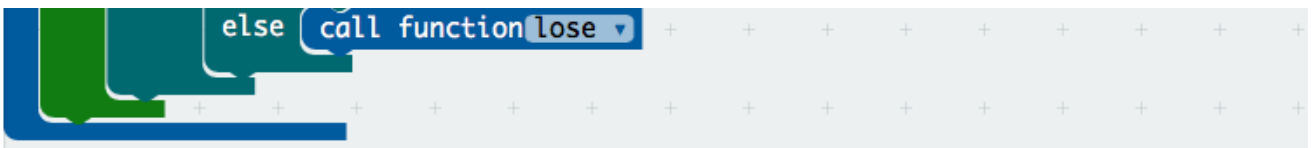


Plug in your ADKeypad to Pin0, making sure the positive lead is connected to the yellow signal pin and the negative lead is connected to the black ground pin on the breakout board.

Step 2

```
function plotLight
  set bool to true
  set randomLightXIndex to pick random 0 to 4
  set randomLightYIndex to pick random 0 to 4
  plot x randomLightXIndex y randomLightYIndex
```

```
pause (ms) time
if (time > 500)
then set time to (time ÷ 10 × 9)
while (bool)
do if (<> key A is pressed on ADKeyboard at pin P0)
then if (randomLightXIndex = 0)
then unplot x randomLightXIndex y randomLightYIndex
call function plotLight
set bool to false
else call function lose
if (<> key B is pressed on ADKeyboard at pin P0)
then if (randomLightXIndex = 1)
then unplot x randomLightXIndex y randomLightYIndex
call function plotLight
set bool to false
else call function lose
if (<> key C is pressed on ADKeyboard at pin P0)
then if (randomLightXIndex = 2)
then unplot x randomLightXIndex y randomLightYIndex
call function plotLight
set bool to false
else call function lose
if (<> key D is pressed on ADKeyboard at pin P0)
then if (randomLightXIndex = 3)
then unplot x randomLightXIndex y randomLightYIndex
pause (ms) 2000
call function plotLight
set bool to false
else call function lose
if (<> key E is pressed on ADKeyboard at pin P0)
then if (randomLightXIndex = 4)
then unplot x randomLightXIndex y randomLightYIndex
call function plotLight
set bool to false
```



In order for the ease of randomisation of the LED that lights up, we will use a function recursively. A function used recursively will call itself (!) so as to achieve the end goal. I created the function plotLight for this reason. If you have not covered functions, go here.

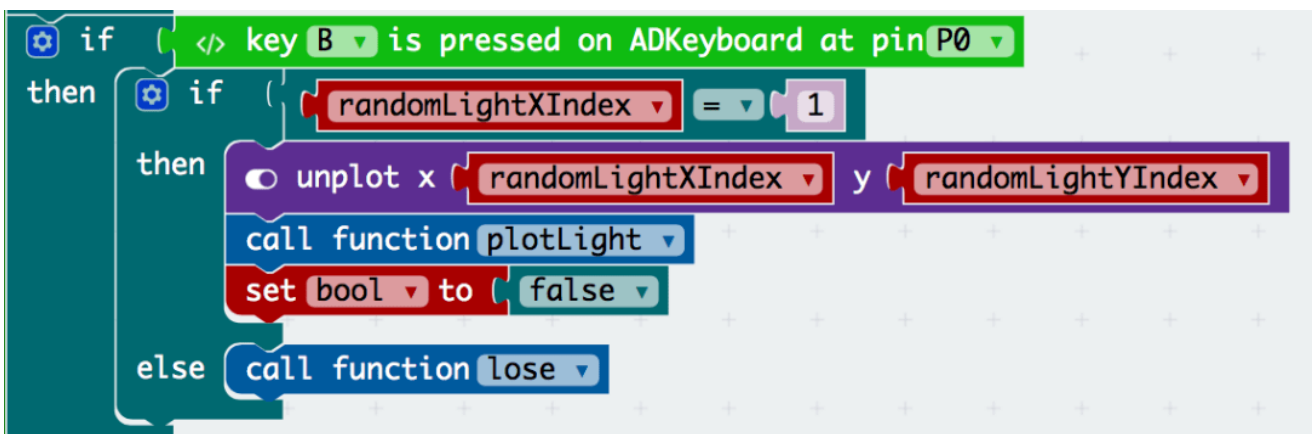
Then I set two variables randomLightXIndex and randomLightYIndex to integers between 0 and 4. This will correspond to the specific LED that lights up. Doing this will ensure randomness (let us not get into the discussion as to where true randomness can really be generated) of the LED that lights up so the game will be different and unpredictable every time.

I also set the variable bool to true. While this may not be obvious now, it will come in handy later (in reality this was a later addition that I decided to add after the rest of the function was fleshed out. The reason for this will come to light later). This is a common technique in coding (especially with while loops).

In order to increase the difficulty, it was my judgement that a time variable could be useful. We use this later to decrease the pause time between one LED lighting up and the next. We have set a lower limit for the pause time at half a second so as to not make the game impossible. When we call the function recursively, the if-statement modifying the pause time is what will decrease the pause time everytime the function is called.

I have created a bunch of if-else statements inside a loop. These statements periodically check if a button on the keyboard was pressed and if the button corresponds to the x-coordinate of the LED that lights up. We have to do this because the pressing of the keypad does not emit an event that our event listeners in micro:bit's core modules can respond to (like how it does for shaking or button presses). Thus, we had to create our own event listener. This event listener only runs as long as bool (which we created earlier) is true.

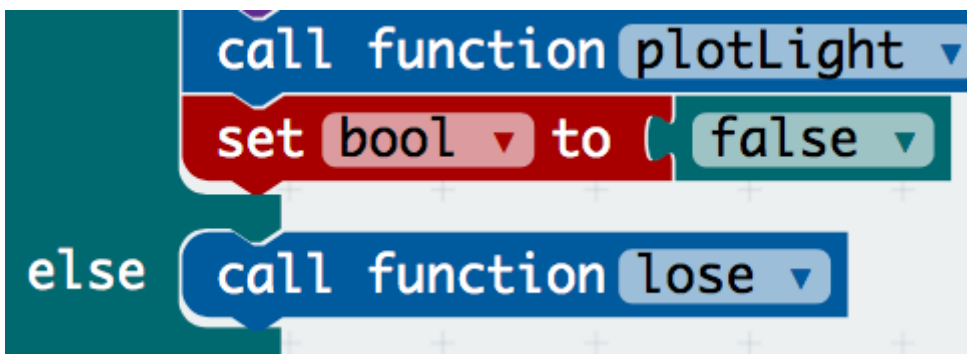
Step 3



Inside the if-else statement, we check to see which key was pressed and if it corresponds to the column of the LED (x-coordinate). If it was, we CALL THE FUNCTION AGAIN. This is how recursive programming works. By calling the function again, we basically start over with a new LED. Note that when we call the function again we decrease the value of the time variable and thus the pause duration will be shorter.

Note that I unplotted the point first LED. This is to ensure that we don't have more than one LED in each round so as to not confuse the player. If you wanted to make the game more difficult, you could show multiple LEDs and play for only the most recent LED that lights up. Treat that as an extension! Interestingly, I have set bool to false. Why?

Step 4



The bool is set to false so as to terminate the above while loop. This is not strictly necessary and I initially disregarded this. However, it is important to note that terminating the while loop greatly improves the efficiency of your program and efficiency of our programs is something generally worth considering.

I have also created and called a function to handle the case where the player types the wrong key. This will be covered later.

Step 5

```

function plotLight
  set bool to true
  set randomLightXIndex to pick random 0 to 4
  set randomLightYIndex to pick random 0 to 4
  plot x randomLightXIndex y randomLightYIndex
  pause (ms) time
  if (time > 500)
  then set time to (time ÷ 10) × 9
  while bool
  do if (<> key A is pressed on ADKeyboard at pin P0)
  then if randomLightXIndex = 0
  then unplot x randomLightXIndex y randomLightYIndex
  call function plotLight
  set bool to false
  else call function lose

```

That was quite a lot for one function! It can be quite a bit for a newbie at programming so let me go through that one more time.

We use random integers between 0 and 4 for the determination of the LED that lights up. (Note that we use an index that starts with 0 – this means that the top left corner is (0,0))

In anticipation of the function being called in some point in the future, we decrease the pause time so that when that happens the game is more difficult.

We run our own homemade event listener (the name betrays its function – it simply waits for an event to happen and acts with our preset code when it does). We use a while loop to listen for an event. If it does not find an event in one loop the if-else statements inside will not be activated and thus, it will go on to the next iteration. When the event does happen (in this case the pressing of the key), the if-else statement is activated from its slumber and thus, in this rather ingenious way, we have created an event listener. (Extension: Browsers listen for events like clicks or keypad presses in the same way).

Step 6

```

while (bool)
do
  if (key A is pressed on ADKeyboard at pin P0)
  then
    if (randomLightXIndex = 0)
    then
      unplot x (randomLightXIndex) y (randomLightYIndex)
      call function plotLight
      set bool to false
    else
      call function lose
  if (key B is pressed on ADKeyboard at pin P0)
  then
    if (randomLightXIndex = 1)
    then
      unplot x (randomLightXIndex) y (randomLightYIndex)
      call function plotLight
      set bool to false
    else
      call function lose
  if (key C is pressed on ADKeyboard at pin P0)
  then
    if (randomLightXIndex = 2)
    then
      unplot x (randomLightXIndex) y (randomLightYIndex)
      call function plotLight
      set bool to false

```

Inside each if-else statement, we have decided to end the game if the wrong keypad was pressed and tell the player what we think of him/her.

If the right key was pressed, we immediately go on to the next LED light whilst ending the previous while loop or efficiency purposes (just to be clear, your code will still work but it's best not to foster such bad habits). Whilst going on to the next LED light, we make use of a concept called recursion. To fully understand the inner workings of recursions we must be familiar with concepts like execution contexts, which is beyond the scope of this tutorial.

Step 7

```

function plotLight
  set bool to true
  set randomLightXIndex to pick random 0 to 4
  set randomLightYIndex to pick random 0 to 4
  plot x (randomLightXIndex) y (randomLightYIndex)
  pause (ms) (time)
  if (time > 500)
  then
    set time to (time ÷ 10) × 9

```

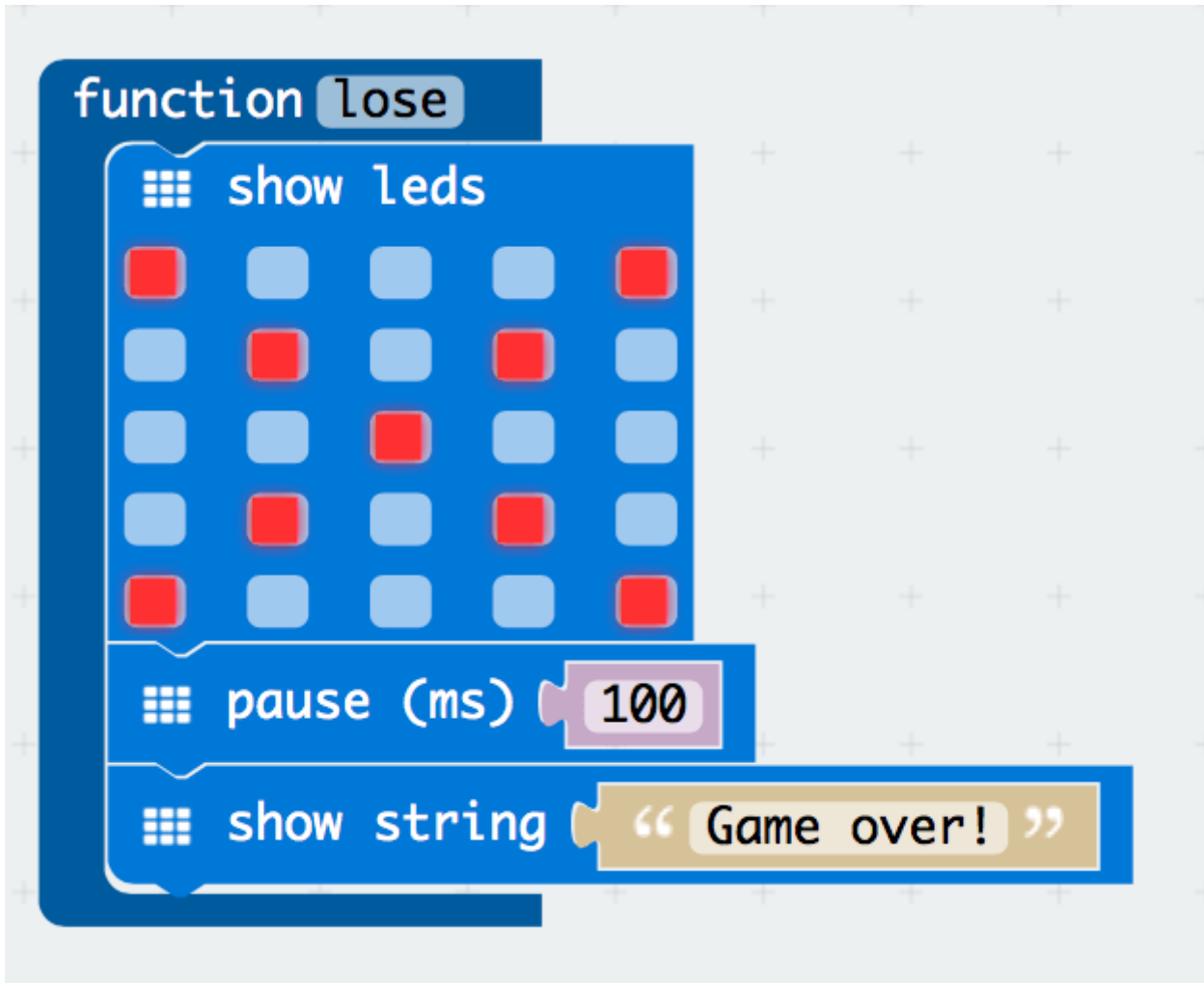


```
while (bool)
do
  if (<> key A is pressed on ADKeyboard at pin P0)
  then
    if (randomLightXIndex = 0)
    then
      unplot x randomLightXIndex y randomLightYIndex
      call function plotLight
      set bool to false
    else
      call function lose
  if (<> key B is pressed on ADKeyboard at pin P0)
  then
    if (randomLightXIndex = 1)
    then
      unplot x randomLightXIndex y randomLightYIndex
      call function plotLight
      set bool to false
    else
      call function lose
  if (<> key C is pressed on ADKeyboard at pin P0)
  then
    if (randomLightXIndex = 2)
    then
      unplot x randomLightXIndex y randomLightYIndex
      call function plotLight
      set bool to false
    else
      call function lose
  if (<> key D is pressed on ADKeyboard at pin P0)
  then
    if (randomLightXIndex = 3)
    then
      unplot x randomLightXIndex y randomLightYIndex
      pause (ms) 2000
      call function plotLight
      set bool to false
    else
      call function lose
  if (<> key E is pressed on ADKeyboard at pin P0)
  then
    if (randomLightXIndex = 4)
    then
      unplot x randomLightXIndex y randomLightYIndex
      call function plotLight
      set bool to false
    else
      call function lose
```

That was a lot of work!!

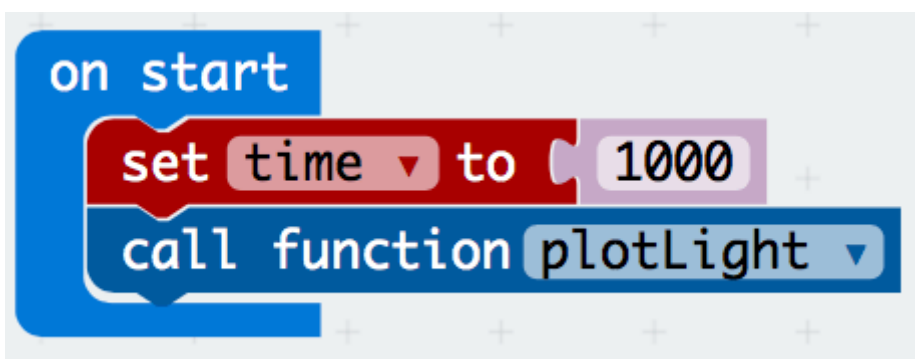
But in the end, we have created a wonderful function that can be called recursively. It is remarkable that such a game can be simplified so much so that its crux is in one block of code!

Step 8



Now we just want to tie up some loose strings. The lose function is one that we will call when the player presses the wrong key. It is mostly self-explanatory and if you could get past the previous parts, it should be obvious what the code does.

Step 9



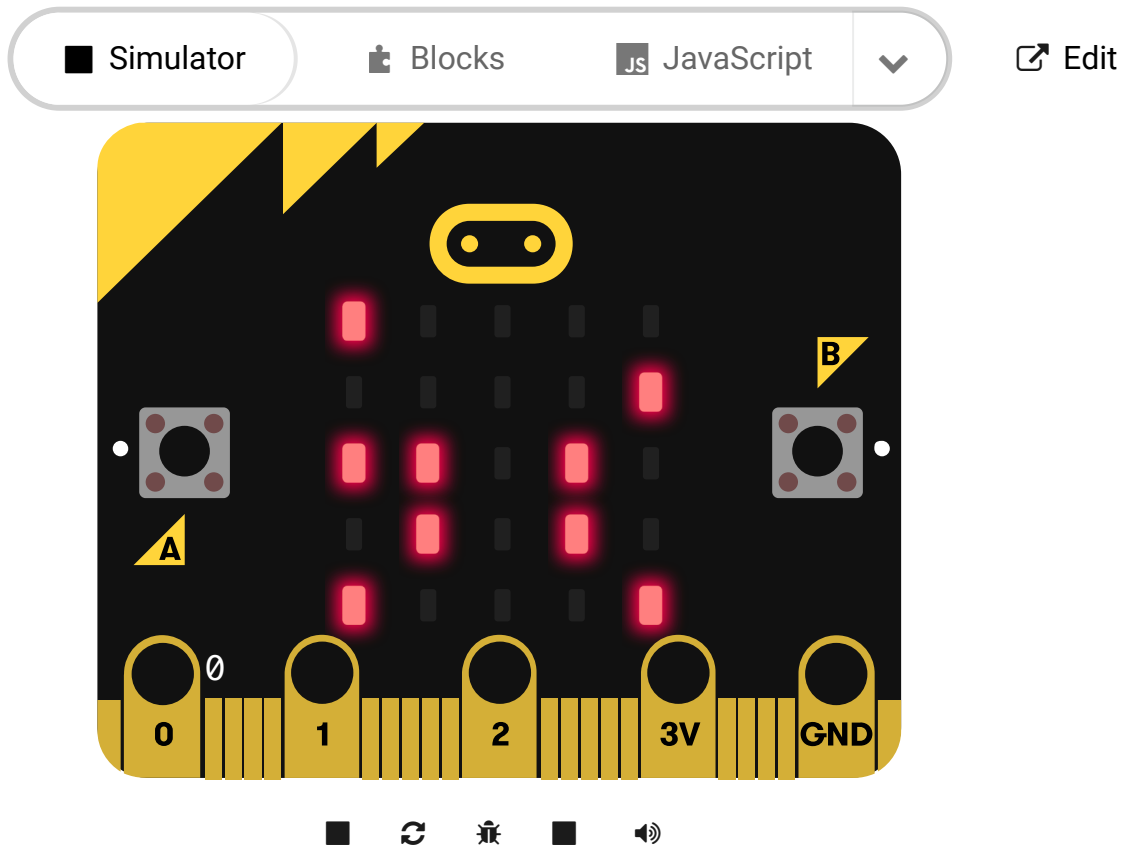
Now to start the first LED.

We call the function when the file loads. Due to the recursiveness of the function, the game will take care of itself thereafter with minimal effort from us. How is that for efficiency!

If you don't want to type these code by yourself, you can download the whole program from the link below.

https://makecode.microbit.org/_eeyAFJMcg8z5

Or you can download from the page below.



Wonderful!

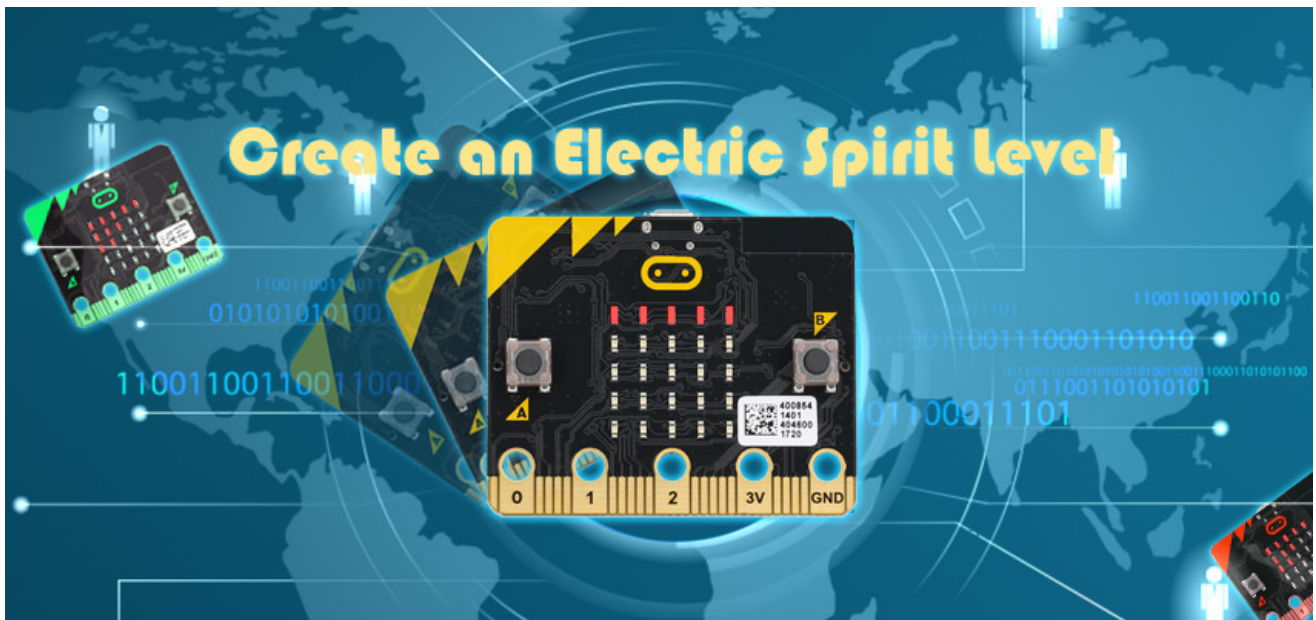


You have completed this tutorial! If you wish to challenge yourself further, go ahead and add a counter that counts the number of points a player gets before he/she loses. Clue: Create a variable called counter and increment it as you see fit. Remember to display it as well!

Congratulations!

This tutorial was possibly a level higher than the rest and if you got here you are definitely rocking it. If you didn't, take solace in the fact that it took me weeks to get my head around concepts like recursions too. Good luck!

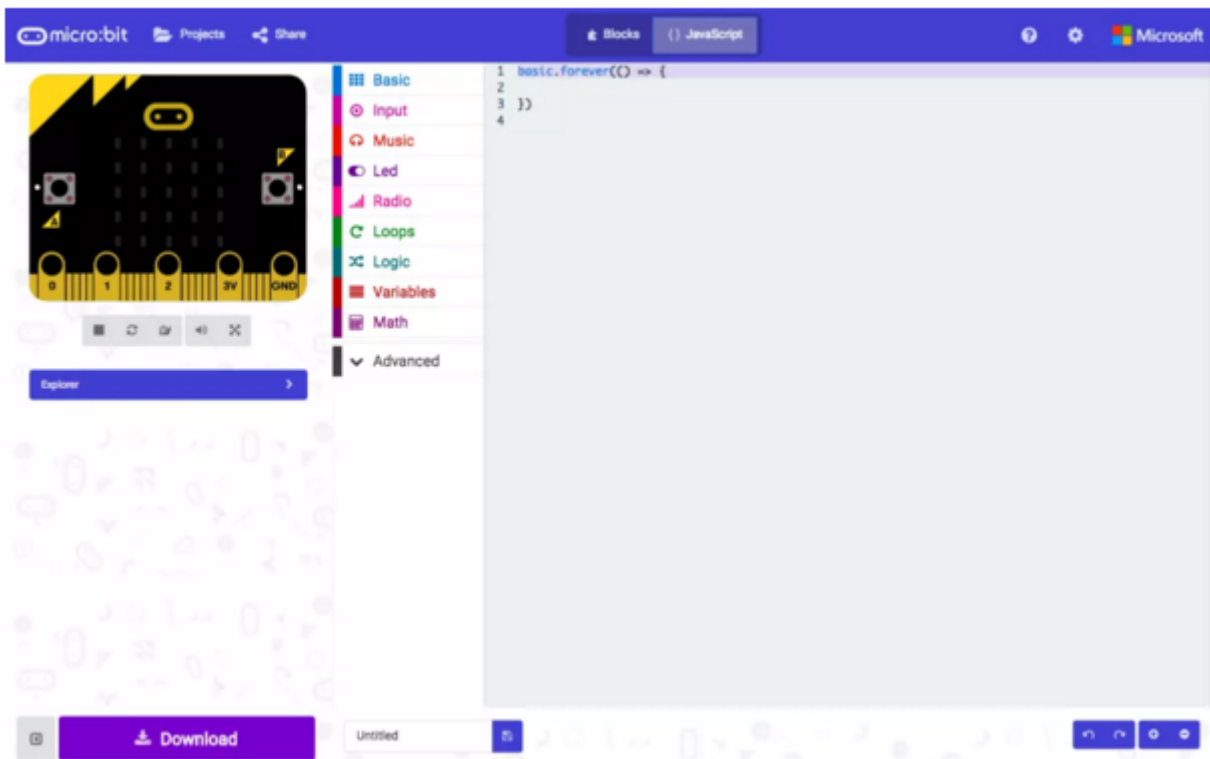
21. case 19 Electric Spirit Level



Use this spirit level to quickly and easily display the tilt of any object attached! Created by Kaitlyn from Raffles Institution.

21.1. Goals

- Learn to read tilt with micro:bit's built-in accelerometer.
- Learn to work with micro:bit's 5x5 LED Display!



21.2. Materials

- 1 x BBC micro:bit
- 1 x Micro USB cable
- 2 x AA Batteries
- 1 x Double AA Battery Pack

21.3. Pre Coding: Connect your Micro:Bit

- Connect the BBC micro:bit to your computer using a micro USB cable.
- Access the javascript editor for the micro:bit at makecode.microbit.org.

Step 0: Code Flow

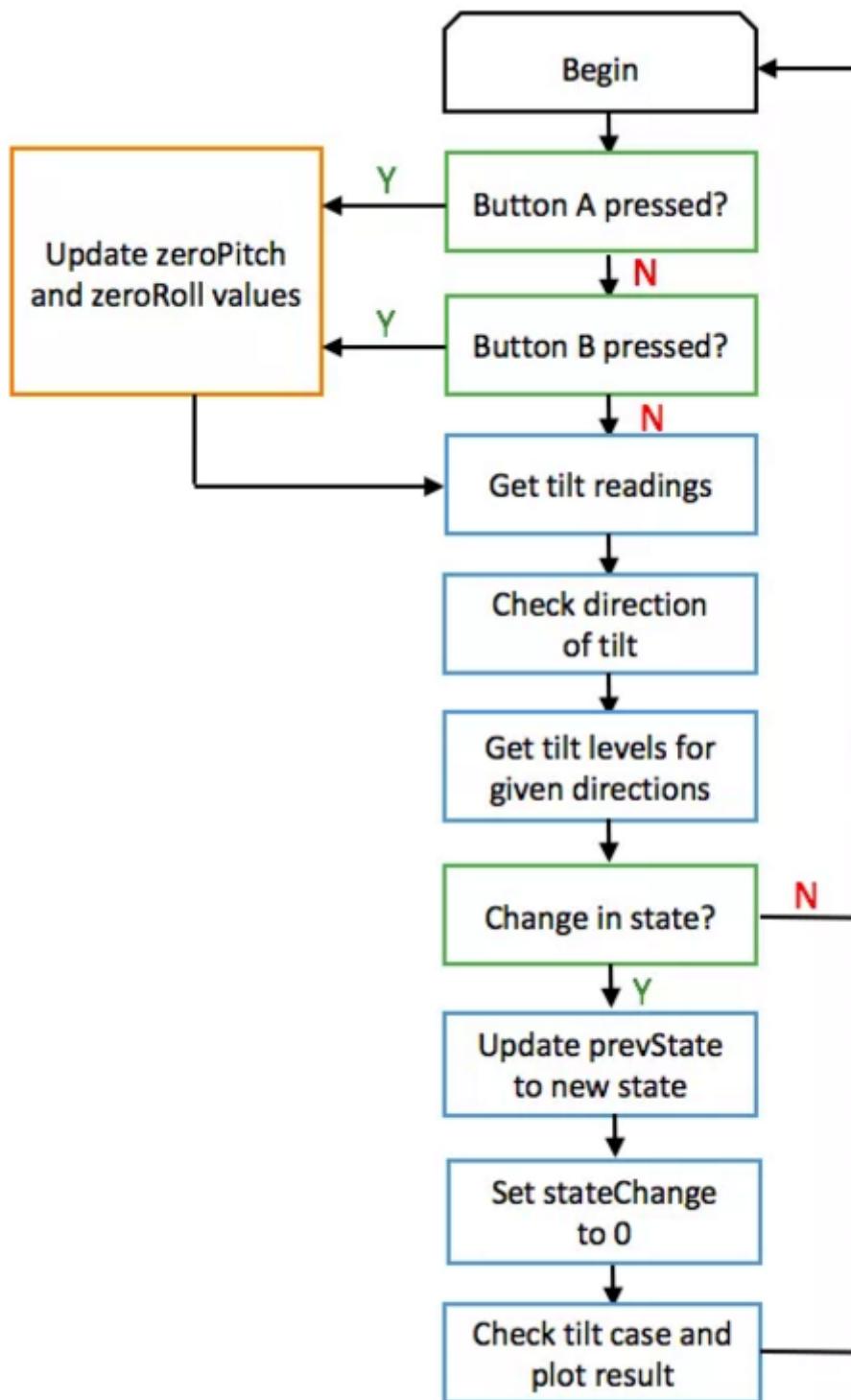
Before we begin writing the code, we need to decide what we want to achieve with the program and in what order should each component run.

For the electric spirit level, the steps that we will take in the code for each loop are:

Read tilt readings from accelerometer. Convert tilt readings to tilt levels to be displayed on LED matrix. Check for change in tilt level readings from previous loop. Create array of LED coordinates for different tilt cases and directions. Plot LED coordinates onto micro:bit LED matrix.

A few additional functions we need to include are:

Calibration for initial tilt position. Returning to default tilt calibration.



21.4. How to Make

Step 1: Defining Variables

We start by defining variables needed as shown. A breakdown of a few variables are:

tiltList: Array that stores extent of tilt from values 0-4 in the order [Left, Right, Forward,

Backward] tiltBoundary: Boundary of the first tilt level between 0 (no tilt) and 1 (slight tilt)

prevState: Array that stores the tilt values of the micro:bit from a previous loop in the same

format as tiltList, used to check for a change in tilt between iterations ledPlotList: Plot led coordinate arrays in the form (x, y). To define an array, we use the type number[][] to indicate a nested array of variables of type: number.

```
1 //tilt variables
2 let tiltList = [0, 0, 0, 0]
3 let tiltBoundary = 1
4 let tiltSensitivity = 5
5
6 //tilt calibration
7 let zeroPitch = 0
8 let zeroRoll = 0
9 let calibratedPitch = 0
10 let calibratedRoll = 0
11
12 //state variables
13 let prevState = [0, 0, 0, 0]
14 let stateChange = 0
15
16 //led variables
17 let ledPlotList: number[][]
18
```

Step 2: Convert tilt values to levels

As the 5x5 LED matrix can only display so much information, the actual tilt values will not be useful for display.

Instead, a function tiltExtent() takes the parameter num, which refers to the tilt value from the accelerometer, and converts these tilt values (num) to tilt levels from 0 to 4.

0 indicates no tilt in the given direction and 4 indicates very large tilt, while -1 is returned when there is an error.

Here, tiltBoundary and tiltSensitivity are used as the boundary values between tilt levels.


```

29 function tiltExtent(num: number) {
30     if (num <= tiltBoundary) {
31         return 0
32     } else if (num > tiltBoundary && num <= tiltBoundary + tiltSensitivity - 1) {
33         return 1
34     } else if (num > tiltBoundary + tiltSensitivity - 1 && num <= tiltBoundary + (tiltSensitivity * 2) - 1) {
35         return 2
36     } else if (num > tiltBoundary + (tiltSensitivity * 2) - 1 && num <= tiltBoundary + (tiltSensitivity * 3) - 1) {
37         return 3
38     } else if (num > tiltBoundary + (tiltSensitivity * 3) - 1) {
39         return 4
40     } else {
41         return 0 - 1
42     }
43 }
44

```

Step 3: Compile tilt levels

The two functions checkRoll() and checkPitch() write the tilt levels obtained from tiltExtent() into tiltList for the roll (left-right) and the pitch (forward-backward) axes respectively.

Before using the tilt values, we calibrate them using a zeroed value for both pitch (zeroPitch) and roll (zeroRoll) obtained from a calibration function written later.

As the accelerometer readings are negative for both left and forward tilt, we need to use the Math.abs() function to obtain the modulus of the negative value to be given to the tiltExtent() function as a parameter for these two directions.

```

45 function checkRoll() {
46     calibratedRoll = input.rotation(Rotation.Roll) - zeroRoll
47     if (calibratedRoll < 0) { //tilt to left
48         tiltList[0] = tiltExtent(Math.abs(calibratedRoll))
49         tiltList[1] = 0
50     } else if (calibratedRoll > 0) { //tilt to right
51         tiltList[0] = 0
52         tiltList[1] = tiltExtent(Math.abs(calibratedRoll))
53     } else if (calibratedRoll == 0) { //no left-right tilt
54         tiltList[0] = 0
55         tiltList[1] = 0
56     } else { //if there is error
57         tiltList[0] = 0 - 1
58         tiltList[1] = 0 - 1
59     }
60 }
61
62 function checkPitch() {
63     calibratedPitch = input.rotation(Rotation.Pitch) - zeroPitch
64     if (calibratedPitch < 0) { //tilt forward
65         tiltList[2] = tiltExtent(Math.abs(calibratedPitch))
66         tiltList[3] = 0
67     } else if (calibratedPitch > 0) { //tilt backward
68         tiltList[2] = 0
69         tiltList[3] = tiltExtent(Math.abs(calibratedPitch))
70     } else if (calibratedPitch == 0) { //no forward-backward tilt
71         tiltList[2] = 0
72         tiltList[3] = 0
73     } else { //if there is error
74         tiltList[2] = 0 - 1
75         tiltList[3] = 0 - 1
76     }
77 }
78

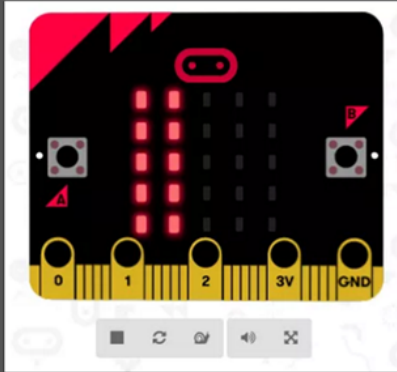
```

Step 4: Write LEDPlotList Functions

Having obtained the tilt levels in tiltList we can now write the led plotting functions for the different cases that can arise, namely

plotSingle(): Tilt only in a single direction, taking extent of tilt in given direction as parameter.
plotDiagonal(): Tilt in two directions of the same magnitude, taking extent of tilt in either direction as parameter. plotUnequal(): Tilt in two directions of different magnitudes, taking extent of tilt in each direction as parameter. Uses plotDiagonal() first and adds on to ledPlotList array afterwards.

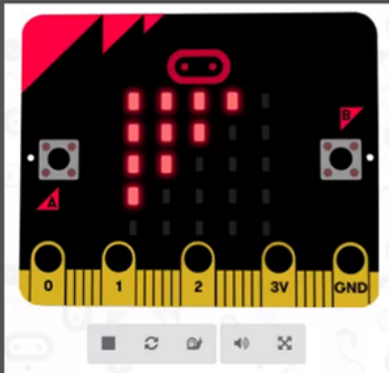
These plotting functions write an array of led coordinates to ledPlotList to be plotted later on.



```

90 function plotSingle(num: number) {
91   for (let x = 0; x < num; x++) {
92     for (let y = 0; y < 5; y++) {
93       ledPlotList.push([x, y])
94     }
95   }
96 }
97

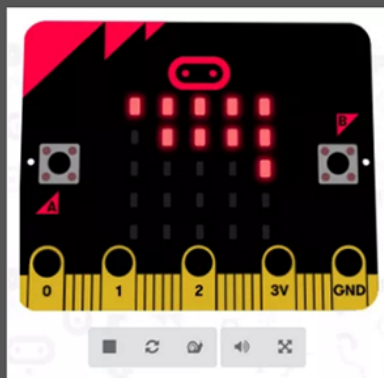
```



```

98 function plotDiagonal(num: number) {
99   for (let x = 0; x < num + 1; x++) {
100     for (let y = 0; y < num + 1 - x; y++) {
101       ledPlotList.push([x, y])
102     }
103   }
104 }
105

```



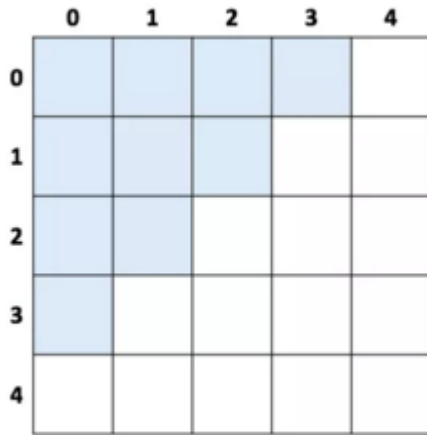
```

106 function plotUnequal(maxX: number, maxY: number, num: number) {
107   if (maxX > maxY) {
108     for (let i = maxX; i > num; i--) {
109       for (let x = i; x > 0; x--) {
110         for (let y = 0; y < num; y++) {
111           if (i - x == y) {
112             ledPlotList.push([x, y])
113           }
114         }
115       }
116     }
117   } else if (maxX < maxY) {
118     for (let i = maxY; i > num; i--) {
119       for (let y = i; y > 0; y--) {
120         for (let x = 0; x < num; x++) {
121           if (i - y == x) {
122             ledPlotList.push([x, y])
123           }
124         }
125       }
126     }
127   }
128 }
129

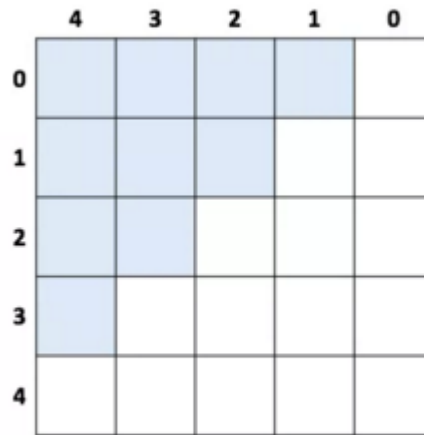
```

Step 5: Plot LED Matrix for Each Case

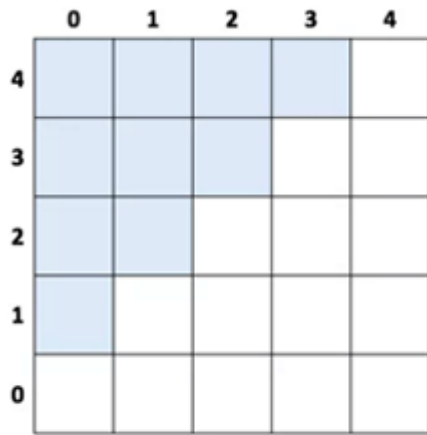
Using the plotting functions from the three cases in step 4, we can now plot the actual LED matrix for the different possible combinations of tilt levels. As the three functions in step 4 do not discriminate with direction, we need to adjust the coordinate values passed to the LED matrix to plot the LEDs in the right directions.



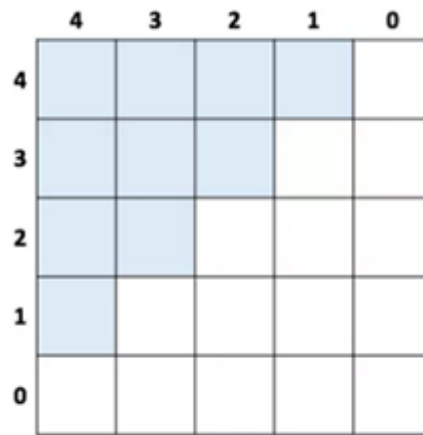
Forward - left



Forward - right



Backward - left



Backward - right

PlotResult() contains multiple if conditions that check the kind of tilt and plot the LED matrix accordingly using led.plot(x, y). The possible combinations of tilt are:

Single direction: Left Only or Right Only.

```

130 function plotResult() {
131
132     // left
133     if (tiltList[0] != 0 && tiltList[1] == 0 && tiltList[2] == 0 && tiltList[3] == 0) {
134         let plotVal = tiltList[0]
135         plotSingle(plotVal)
136         for (let j = 0; j <= ledPlotList.length - 1; j++) {
137             led.plot(ledPlotList[j][0], ledPlotList[j][1])
138         }
139     }
140
141     //right
142     else if (tiltList[0] == 0 && tiltList[1] != 0 && tiltList[2] == 0 && tiltList[3] == 0) {
143         let plotVal = tiltList[1]
144         plotSingle(plotVal)
145         for (let j = 0; j <= ledPlotList.length - 1; j++) {
146             led.plot(4 - ledPlotList[j][0], ledPlotList[j][1])
147         }
148     }
149 }

```

Single direction: Forward Only or Backward Only.

```

151 //forward
152 else if (tiltList[0] == 0 && tiltList[1] == 0 && tiltList[2] != 0 && tiltList[3] == 0) {
153     let plotVal = tiltList[2]
154     plotSingle(plotVal)
155     for (let j = 0; j <= ledPlotList.length - 1; j++) {
156         led.plot(ledPlotList[j][1], ledPlotList[j][0])
157     }
158 }
159
160 //backward
161 else if (tiltList[0] == 0 && tiltList[1] == 0 && tiltList[2] == 0 && tiltList[3] != 0) {
162     let plotVal = tiltList[3]
163     plotSingle(plotVal)
164     for (let j = 0; j <= ledPlotList.length - 1; j++) {
165         led.plot(ledPlotList[j][1], 4 - ledPlotList[j][0])
166     }
167 }
168

```

Two directions: Forward-left or Backward-left.

```

169 //forward left
170 else if (tiltList[0] != 0 && tiltList[1] == 0 && tiltList[2] != 0 && tiltList[3] == 0) {
171     let maxX = tiltList[2]
172     let maxY = tiltList[0]
173     let diagNum = Math.min(maxX, maxY)
174     plotDiagonal(diagNum)
175     if (maxX != maxY) {
176         plotUnequal(maxX, maxY, diagNum)
177     }
178     for (let k = 0; k <= ledPlotList.length - 1; k++) {
179         led.plot(ledPlotList[k][0], ledPlotList[k][1])
180     }
181 }
182
183 //backward left
184 else if (tiltList[0] != 0 && tiltList[1] == 0 && tiltList[2] == 0 && tiltList[3] != 0) {
185     let maxX = tiltList[3]
186     let maxY = tiltList[0]
187     let diagNum = Math.min(maxX, maxY)
188     plotDiagonal(diagNum)
189     if (maxX != maxY) {
190         plotUnequal(maxX, maxY, diagNum)
191     }
192     for (let k = 0; k <= ledPlotList.length - 1; k++) {
193         led.plot(ledPlotList[k][0], 4 - ledPlotList[k][1])
194     }
195 }
196

```

Two directions: Forward-right or Backward-right.

```

197 //forward right
198 else if (tiltList[0] == 0 && tiltList[1] != 0 && tiltList[2] != 0 && tiltList[3] == 0) {
199     let maxX = tiltList[2]
200     let maxY = tiltList[1]
201     let diagNum = Math.min(maxX, maxY)
202     plotDiagonal(diagNum)
203     if (maxX != maxY) {
204         plotUnequal(maxX, maxY, diagNum)
205     }
206     for (let k = 0; k <= ledPlotList.length - 1; k++) {
207         led.plot(4 - ledPlotList[k][0], ledPlotList[k][1])
208     }
209 }
210
211 //backward right
212 else if (tiltList[0] == 0 && tiltList[1] != 0 && tiltList[2] == 0 && tiltList[3] != 0) {
213     let maxX = tiltList[3]
214     let maxY = tiltList[1]
215     let diagNum = Math.min(maxX, maxY)
216     plotDiagonal(diagNum)
217     if (maxX != maxY) {
218         plotUnequal(maxX, maxY, diagNum)
219     }
220     for (let k = 0; k <= ledPlotList.length - 1; k++) {
221         led.plot(4 - ledPlotList[k][0], 4 - ledPlotList[k][1])
222     }
223 }
224 }
225

```

Note: For tilt in two directions, each combination can have the same or different magnitude (checked by comparing `maxX` and `maxY`), and hence plotted using `plotDiagonal()` or `plotUnequal()` respectively.

Step 6: Write Calibration Functions

Having completed the bulk of the code, we now add in the `calibTilt()` and the `resetTilt()` functions.

`calibTilt()` allows users to tare the tilt to zero at the micro:bit's current position `resetTilt()` resets the calibration of the board to its original state.

```
19 function calibrateTilt() {
20     zeroPitch = input.rotation(Rotation.Pitch)
21     zeroRoll = input.rotation(Rotation.Roll)
22 }
23
24 function resetTilt() {
25     zeroPitch = 0
26     zeroRoll = 0
27 }
28
```

Step 7: Write State Function

We add a simple function `checkState()` to check whether the tilt levels have changed from a previous iteration.

If there is no change in tilt levels from a previous iteration i.e. `stateChange == 0`, we can directly move on to the next iteration and skip the plotting of the LED matrix, reducing computation needed.

```
79 function checkState() {
80     for (let n = 0; n < 4; n++) {
81         if (prevState[n] == tiltList[n]) {
82             stateChange = 0
83         } else {
84             stateChange = 1
85             break
86         }
87     }
88 }
89
```

Step 8: Putting It All Together Part 1!

Now we can finally place all the necessary functions into the micro:bit's infinite loop to run it repeatedly.

Firstly, we set button A and B on the micro:bit to the `calibTilt()` and `resetTilt()` functions respectively using `input.onButtonPressed()`, and plot a tick on the LED matrix when calibration is completed.

```
225 basic.forever(() => {
226     input.onButtonPressed(Button.A, () => {
227         basic.clearScreen()
228         calibrateTilt()
229         ledPlotList = [[0, 3], [1, 4], [2, 3], [3, 2], [4, 1]]
230         for (let k = 0; k <= ledPlotList.length - 1; k++) {
231             led.plot(ledPlotList[k][0], ledPlotList[k][1])
232         }
233         control.waitMicros(1000000)
234     })
235
236     input.onButtonPressed(Button.B, () => {
237         basic.clearScreen()
238         resetTilt()
239         ledPlotList = [[0, 3], [1, 4], [2, 3], [3, 2], [4, 1]]
240         for (let k = 0; k <= ledPlotList.length - 1; k++) {
241             led.plot(ledPlotList[k][0], ledPlotList[k][1])
242         }
243         control.waitMicros(1000000)
244     })
245 }
```

Step 9: Putting it All Together Part 2!

Next run the necessary functions according to our code flow in Step 0 and check for a state change (meaning that there has a change in the tilt of micro:bit since the last iteration).

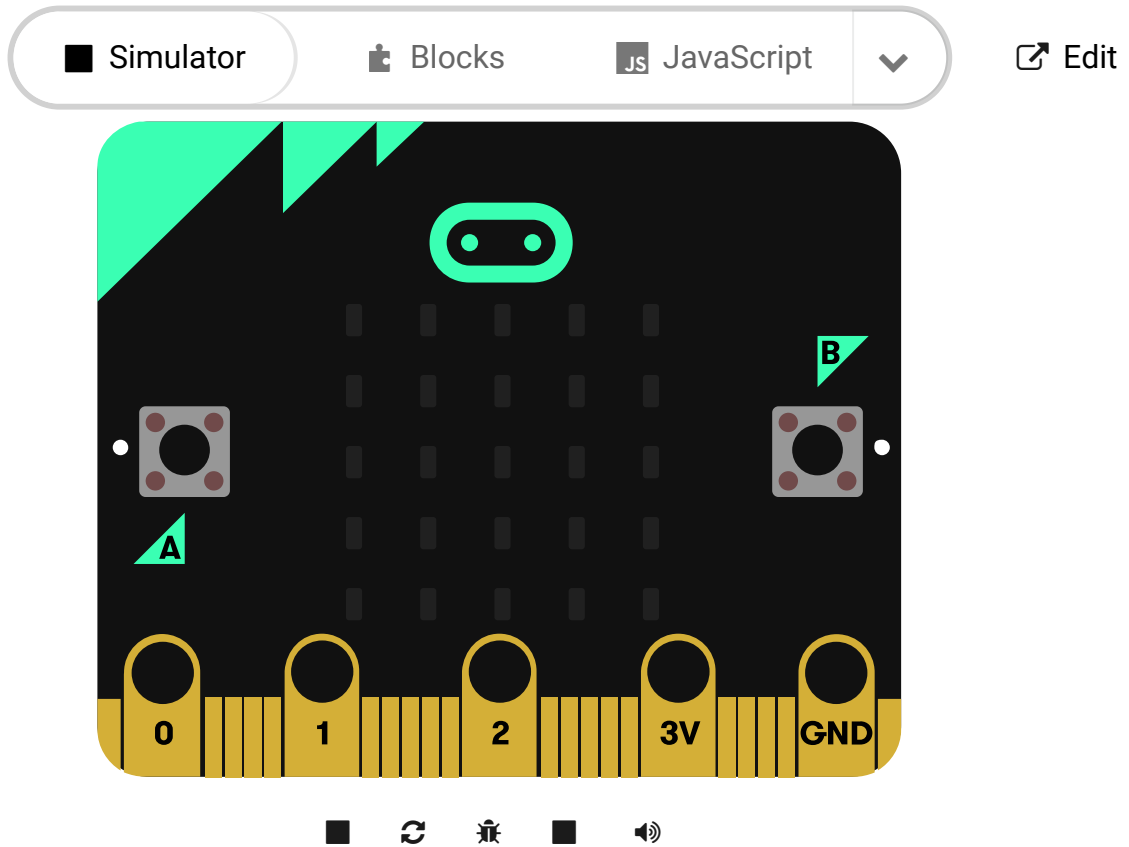
If there is a change in tilt levels i.e. `stateChange == 1`, the code will update `prevState` to the new tilt levels and set `stateChange` back to 0 for the next iteration, and plot the updated tilt levels on the LED matrix using `PlotResult()`.

```
246     checkRoll()
247     checkPitch()
248     checkState()
249
250     if (stateChange == 1) {
251         for (let m = 0; m < tiltList.length; m++) {
252             prevState[m] = tiltList[m]
253         }
254         stateChange = 0
255         basic.clearScreen()
256         plotResult()
257     }
258     control.waitMicros(10000)
259 }
260 }
```


If you don't want to type these code by yourself, you can directly download from the link below.

<https://makecode.microbit.org/56811-31458-64502-76623>

Or you can download from the page below.



Step 10: Assembly

Flash the completed code to your micro:bit.

Attach your micro:bit and the battery pack securely to any object and it is ready for use!



Awesome!

Have fun with your electric spirit level! And while you're at it, why not try to extend the capabilities of the tilt sensor or even turn it into a game?

22. case 20 Space Shooter



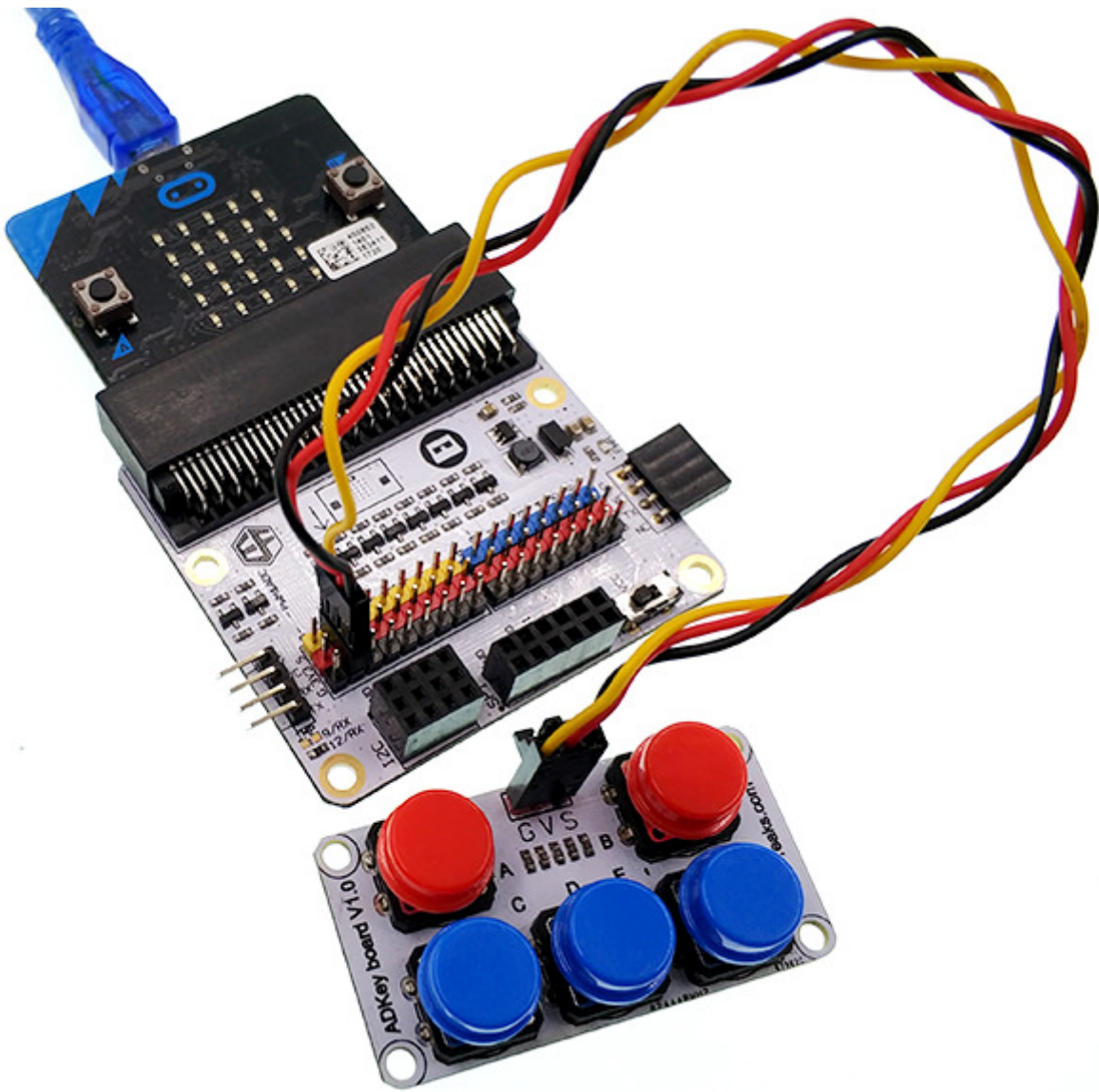
Are you tired of complicated flashy modern games? Prefer to play your games on a 5 x 5 resolution rather than a 4K resolution? Have some arcade fun on your micro:bit with Space Shooter! This tutorial is in JavaScript. Typing! Many typing!

22.1. Step 0: Pre Build Overview

In this project, we will create a simple space shooter game where you have to try to shoot and avoid falling projectiles.

22.2. Materials:

- 1 x BBC micro:bit
- 1 x Micro USB cable
- 1 x Breakout board
- 1 x ADKeypad

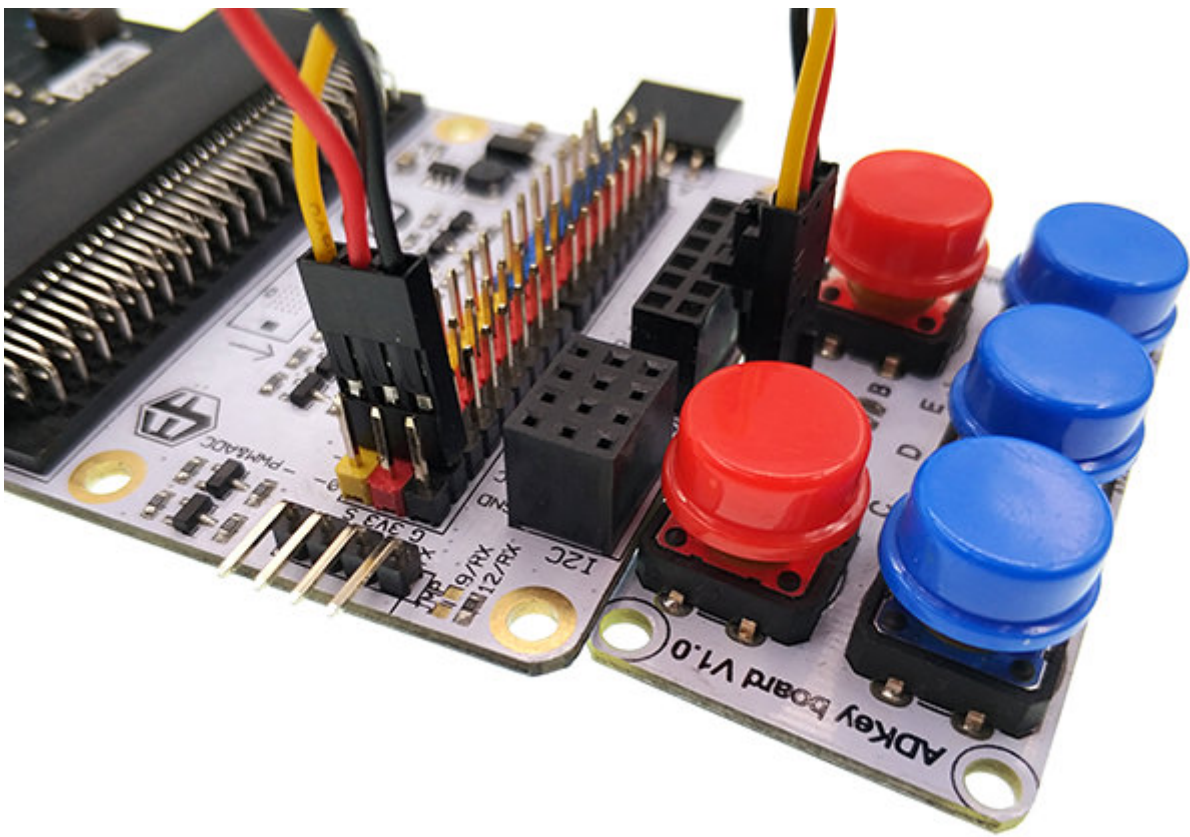


22.3. Goals

- Learn to use the ADKeyboard.
- Learn basic game programming.
- Learn more about programming with Javascript.

22.4. How to Make

Step 1 – Components



First of all, plug in the ADKeypad. Ensure the colours match and take note of what pin you plug them into as it will be relevant later.

Step 2 – Coding



In this project, we will use Javascript to code, so first of all, switch to Javascript mode on the top of the page.

```

let temp2 = 0
let temp = 0
let playerscore = 0
let noalien = 0
let hi = 0
let gamestart = 0
let destroyedpos: number[] = []
let aliens: number[][] = []
let shots: number[][] = []
let pos = 2
let toprow = [0, 0, 0, 0, 0]

```

At the beginning of our code, we will need to initialise the variables we will use: playerscore for the current player's score, noalien as a storage of whether or not there are aliens currently on the screen, hi for the current highscore, gamestart to keep track of what state the game is currently in, destroyedpos as an array to store where the player collided with an alien, shots to store the position of the shots that the player has shot, aliens to store the positions of the aliens, pos to keep track of the player's current position, toprow to store if there is an alien currently in each position of the top row, as well as a few temporary variables. We will need to use all these variables later.

```

function unrendership() {
  led.unplot(pos, 4)
  led.unplot(pos, 3)
  if (pos > 0) {
    led.unplot(pos - 1, 4)
  }
  if (pos < 4) {
    led.unplot(pos + 1, 4)
  }
}

```

This function, unrendership, turns off the LEDs that represents our spaceship. We will use this to move the spaceship around.

```
function rendership() {
  led.plot(pos, 4)
  led.plot(pos, 3)
  if (pos > 0) {
    led.plot(pos - 1, 4)
  }
  if (pos < 4) {
    led.plot(pos + 1, 4)
  }
}
```

This function, rendership, turns on the LEDs that represents our spaceship. After using unrendership, we can update the player's position then use rendership to change the spaceship's position.

```
function unrendershots() {
  for (let i = 0; i <= shots.length - 1; i++) {
    if (shots[i][1] > -1) {
      led.unplot(shots[i][0], shots[i][1])
    }
  }
}
```

This function, unrendershots, turns off the LEDs that represents the player's fired shots. Similarly, we will use this function to update the positions of the shots.

```
function rendershots() {
  for (let j = 0; j <= shots.length - 1; j++) {
    if (shots[j][1] > -1) {
      led.plot(shots[j][0], shots[j][1])
    }
  }
}
```

This function, `rendershots`, turns on the LEDs that represents the player's fired shots. Similar to `unrendership` and `rendership`, after using `unrendershots`, we can update the shots' positions then use `rendershots` to change all fired shots' position.

```
function checkcollision() {
  for (let k = 0; k <= aliens.length - 1; k++) {
    if ((aliens[k][0] == pos && (aliens[k][1] == 4 || aliens[k][1] == 3)) ||
        (aliens[k][1] == 4 && (aliens[k][0] == pos - 1 || aliens[k][0] == pos + 1))) {
      destroyedpos = aliens[k]
      gamestart = 2
    }
  }
  for (let l = 0; l <= shots.length - 1; l++) {
    if (aliens[k][0] == shots[l][0] && aliens[k][1] == shots[l][1]) {
      led.unplot(aliens[k][0], aliens[k][1])
      if (aliens[k][1] == 0) {
        toprow[aliens[k][0]] = 0
      }
      shots[l][1] = -1
      aliens[k][1] = 5
      playerscore += 1
    }
  }
}
}
```

This function, `checkcollision`, checks for two types of collisions: between the player and an alien, and between an alien and a shot fired by the player. Firstly, for the collision between the player and an alien, due to the shape of our ship, we have to check if there is an alien on the lowest or second lowest row and in the same column of the centre of the spaceship, as well as if there is an alien on the lowest row and to the left or right of the centre of the spaceship. Note that in Javascript, `&&` represents “and” and `||` represents “or”. If there is indeed an alien in one of these positions, we will set `destroyedpos` to the position where this alien collided with the spaceship and change the value of `gamestart` to 2 to signify that the game is over. Next, for collision between the player and a shot fired by the player, since each of these only occupy one LED, we just have to check if their positions are exactly the same. However, since there can be multiple shots and/or aliens, we have to loop through every shot for every alien and check if they are in the same position. If they are, we turn off the LED for that position. If the alien was in the `toprow`, we set the value of `toprow` for that column to 0 to signify that there is no longer an alien in the top row of that column. Then, we set the height of the shot to -1 and the height of the alien to 5, moving both of them out of the screen, where we will remove them later. Lastly, we increase the player's score by 1 for shooting an alien.


```

basic.forever(() => {
  if (gamestart == 0) {
    basic.showNumber(hi)
  }
  if (gamestart == 2) {
    aliens = []
    shots = []
    for (let i = 0; i < 3; i++) {
      led.unplot(destroyedpos[0], destroyedpos[1])
      basic.pause(500)
      led.plot(destroyedpos[0], destroyedpos[1])
      basic.pause(500)
    }
    if (hi < playerscore) {
      basic.showIcon(IconNames.Happy)
      basic.pause(1000)
      basic.showString("NEW HISCORE")
      basic.showNumber(playerscore)
      hi = playerscore
      basic.pause(2000)
      gamestart = 0
    } else {
      basic.showIcon(IconNames.Sad)
      basic.pause(2000)
      basic.showNumber(playerscore)
      basic.pause(2000)
      gamestart = 0
    }
  }
}
}

```

Now that we have written all the required functions, we can finally start linking them together! You may already be familiar with the forever function from coding in blocks mode. In Javascript, we use this by typing `basic.forever(() => {Code to run forever here})`. Firstly, if the value of `gamestart` is 0, we show the current highscore on the display. Next, if the value of `gamestart` is 2, that means that the player's game has just ended. We make the position where the player collided with an alien blink 3 times to let the player know where the collision happened, then display a happy face if the player set a new highscore, and a sad face if not. After that, we display the player's score, and set the value of `gamestart` back to 0.

```

if (tinkercademy.ADKeyboard(ADKeys.A, AnalogPin.P1) && gamestart == 0) {
  gamestart = 1
  pos = 2
  playerscore = 0
  basic.clearScreen()
  rendership()
}

```

To start the game, we detect if the player pressed the “A” button on the ADKeypad and the value of gamestart is 0. If so, we set the value of gamestart to 1 to signify that the game has started, reset the player’s position to the centre, set the player’s score to 0, turn off all the LEDs on the screen then render the ship using the function we made earlier.

```

if (gamestart == 1) {
  unrendershots()
  noalien = 0
  if (Math.random(15) == 0) {
    temp = Math.random(5)
    if (toprow[temp] == 0) {
      aliens.push([temp, -1, 4])
    }
  }
  for (let n = 0; n <= aliens.length - 1; n++) {
    aliens[n][2]++
    if (aliens[n][1] < 5) {
      noalien = 1
    }
    if (aliens[n][2] > 4 && aliens[n][1] < 5) {
      led.unplot(aliens[n][0], aliens[n][1])
      aliens[n][1]++
      if (aliens[n][1] == 1) {
        toprow[aliens[n][0]] = 0
      }
      aliens[n][2] = 0
      led.plot(aliens[n][0], aliens[n][1])
    }
    if (aliens[n][1] == 0) {
      toprow[aliens[n][0]] = 1
    }
  }
}

```

Finally, if the value of `gamestart` is 1, the player is currently playing the game! At the start of each loop, we turn off the LEDs of each shot the player fires, since they need to move upwards by 1 position. We use the `Math.random()` function to randomly determine if we should spawn an alien this frame. You can lower the number to make game harder, or increase the number to make it easier. Here, the value we use is 15, which means that there is a 1/15 chance that an alien will spawn every loop. However, we need to check that the top row of the column we want to spawn in is not occupied, or there will be overlapping aliens! Next, we loop through the aliens and move them downwards by 1 position every 5 times the forever loop runs. If an alien enters the top row, we set the value of `toprow` of that column to 1, and if an alien leaves the top row, we set it to 0.

```
for (let o = aliens.length - 1; o >= 0; o--) {
  if (aliens[o][1] >= 5) {
    aliens.removeAt(o)
  }
}
if (noalien == 0) {
  temp2 = Math.random(5)
  if (toprow[temp2] == 0) {
    aliens.push([temp2, -1, 4])
  }
}
```

Then, we check if each alien is outside the screen, and, if it is, we remove it from our aliens array. Lastly, if there are no aliens on screen, we spawn an alien in the same way we would if it spawned from the 1/15 chance.

```
checkcollision()
for (let p = 0; p <= shots.length - 1; p++) {
  shots[p][1]--
}
if (tinkercademy.ADKeyboard(ADKeys.D, AnalogPin.P1)) {
  shots.push([pos, 2])
}
checkcollision()
for (let q = shots.length - 1; q > 0; q--) {
  if (shots[q][1] < 0) {
    shots.removeAt(q)
  }
}
```

After updating the aliens' positions, we check for collisions, then update the shots' positions. Then, we spawn a shot if the player pressed the D button on the ADKeyboard. After that, we remove any shots which are outside the screen.

```

if (gamestart == 1) {
  if (tinkercademy.ADKeyboard(ADKeys.C, AnalogPin.P1)) {
    if (pos > 0) {
      unrendership()
      pos += -1
      rendership()
    }
  }
  if (tinkercademy.ADKeyboard(ADKeys.E, AnalogPin.P1)) {
    if (pos < 4) {
      unrendership()
      pos += 1
      rendership()
    }
  }
}
rendershots()
basic.pause(80)
})

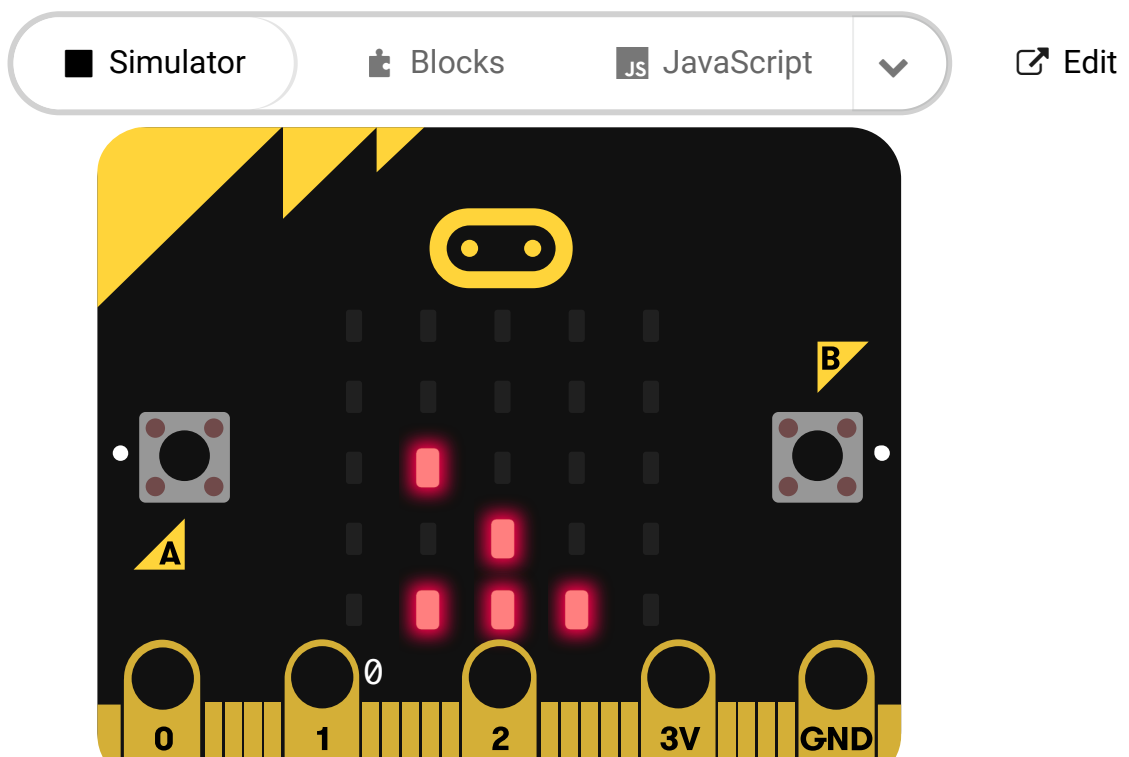
```

Finally, we check if the player has pressed the C or E keys on the ADKeyboard, and update the ship's position accordingly. After that, we render the shots that the player has fired, then set a pause of 0.08 seconds per loop so that the game advances at a playable speed.

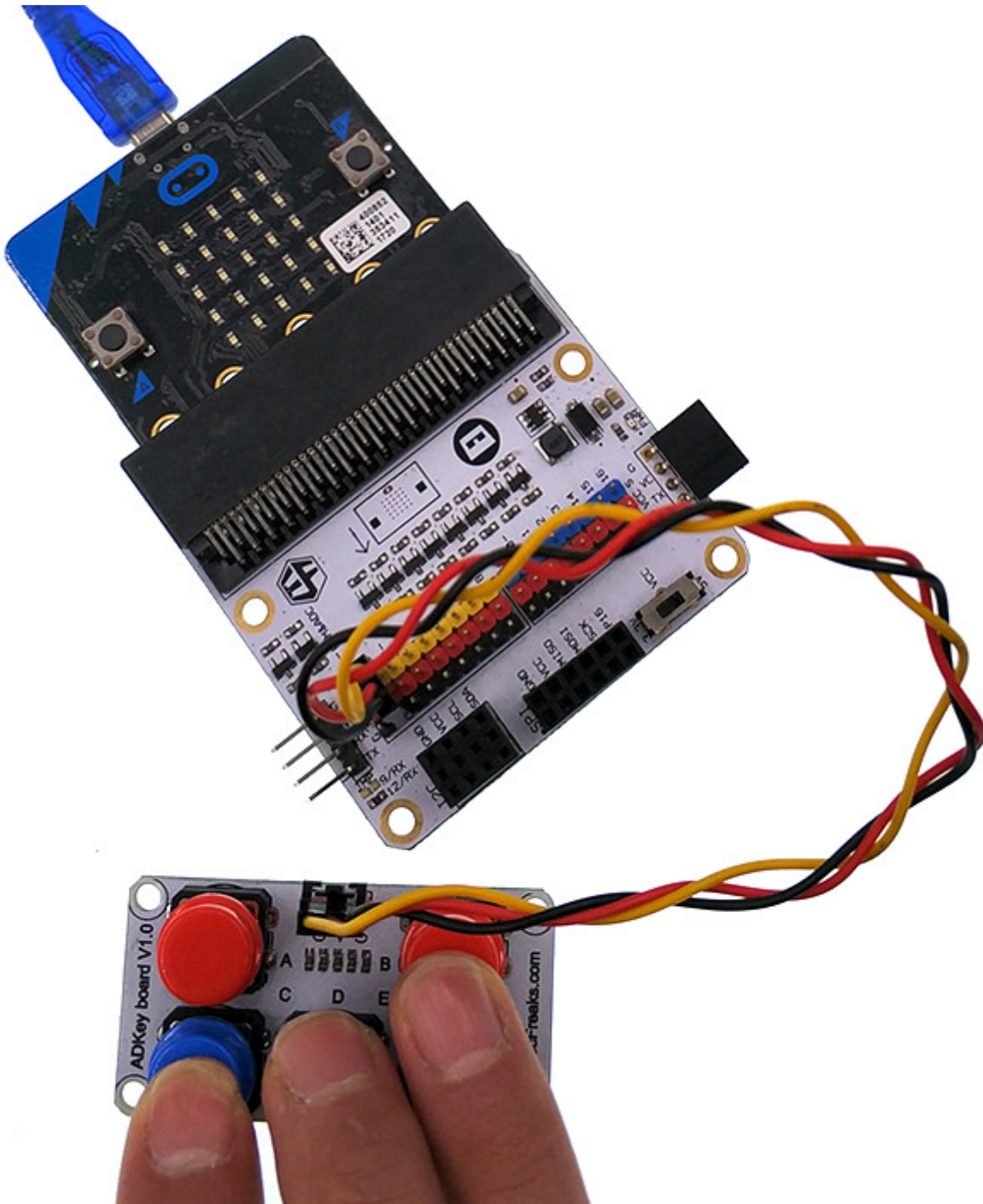
For the whole program, you can download directly from the link below :

https://makecode.microbit.org/_euRV3uHYJAfx

Or download from the page below.



Step 3: Using It

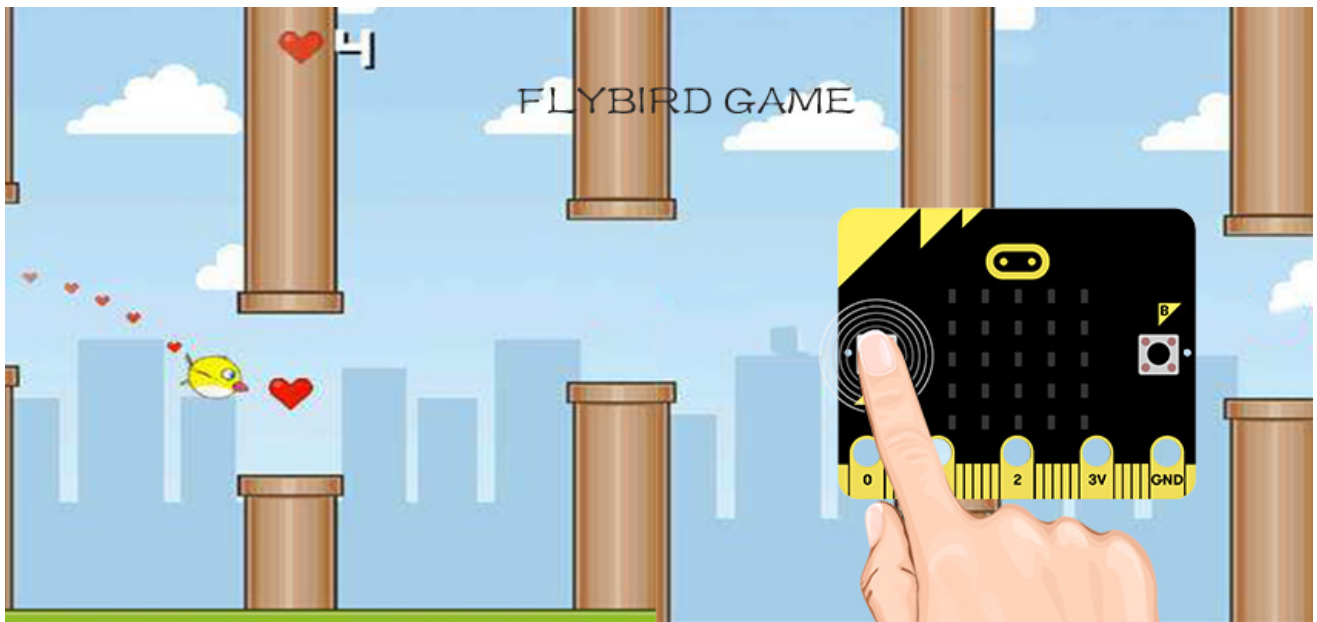


Playing the game is very simple. Just use the A button on the ADKeyboard to start the game, use the C and E buttons to move and the D key to shoot the aliens!

Step 4 – Success!

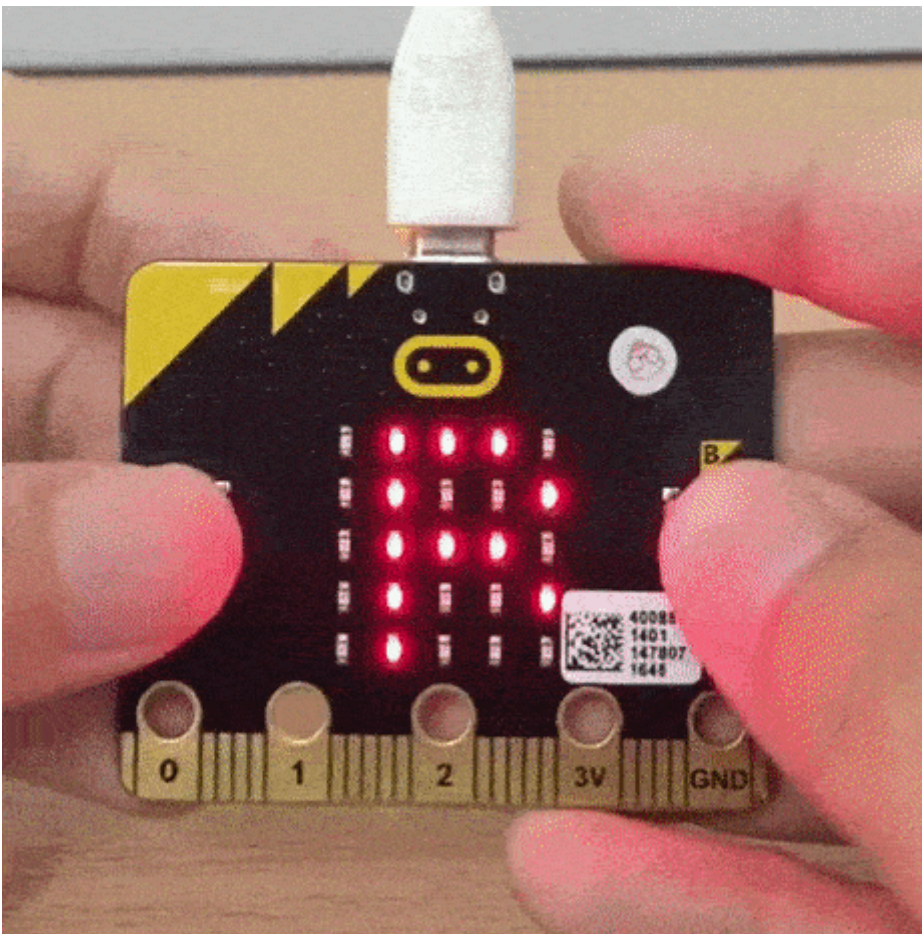
Voila! Time to have some old fashioned arcade fun with your new space shooter. What highscore can you reach?

23. case 21 Flappy Bird



Take flight and achieve your pipe dreams with your own version of the notoriously challenging Flappy Bird game, using nothing but a micro:bit (no extras needed) and some Python code.

Made by Cheryl from Raffles Institution. Warning: heavy dosage of bird puns included.



23.1. Goals

We're going to create a full-fledged interactive game on your 5x5 LED screen, playable for ages 9 days to 90 years old. In the process, you'll learn how to: First step is to import the micro:bit library into Python. Then, let a 'READY' message scroll across the screen and initiate countdown that shows when the game starts.

- Line 1: This imports the micro:bit program.
- Line 4: This initiates the 'READY' message that scrolls across the screen. Double quotation marks indicate a string (in this case 'READY').
- Lines 5-10: This flashes each number on the screen for 1 second (or 1000 milliseconds, the measurement involved) by using the sleep() function.
- Line 11: clears the screen for us to draw the bird and walls later on.

Note:

It's always good to add comments to explain your own code for others or yourself to understand when coming back to it. You add a comment with '#'. Also space out your code when necessary to indicate different segments that do different things.

23.2. Materials

- 1 x BBC micro:bit
- 1 x Micro USB Cable (Seriously, that's all you need.)

23.3. Why Python?

- Reads like English – Python is one of the easiest languages to read, which makes it such a fantastic beginner's language.
- Versatile – Python is industry standard for good reason. It can be used to do so much. This is why Google and YouTube utilise the language for part of its back-end software.
- Active community – Python is one of the most popular languages for beginners. There are tons of resources and many more than willing to help look over your code, which will prove invaluable to helping you get over stumbling blocks in your coding journey.

Actual coding looks cooler than block-based drag-drop coding. I know it's intimidating, but look at these colours! (Demo of Flappy Bird on Sublime Text) [How Do I Start Coding in Python?](#)

If you're a fledgling to programming, you probably don't have Python lying around. Don't worry! Just go to the [official micro:bit Python editor](#) or download the offline Python editor [mu](#) to write code and send it to your micro:bit. You can also use your own text editor (three cheers to Sublime 3 and Atom) but you have to flash it to the micro:bit. This might turn out to be quite troublesome. Alternatively, you can use a [micro:bit simulator](#), which is really useful to test code out without downloading the .hex file each time, and makes it easier to fix errors.

Once set up, connect your micro:bit to your computer using the micro-USB cable. It should connect to the port at the top of the backside of the micro:bit. Once ready to be flashed, the micro:bit should light up bright yellow. Ignore this step if you're on the simulator. Otherwise, stop reading and set it up if you haven't already. Don't worry, I'll wait.

Welcome back. Without feather ado, let's get started! [A Bird's Eye View of What We're Doing.](#)

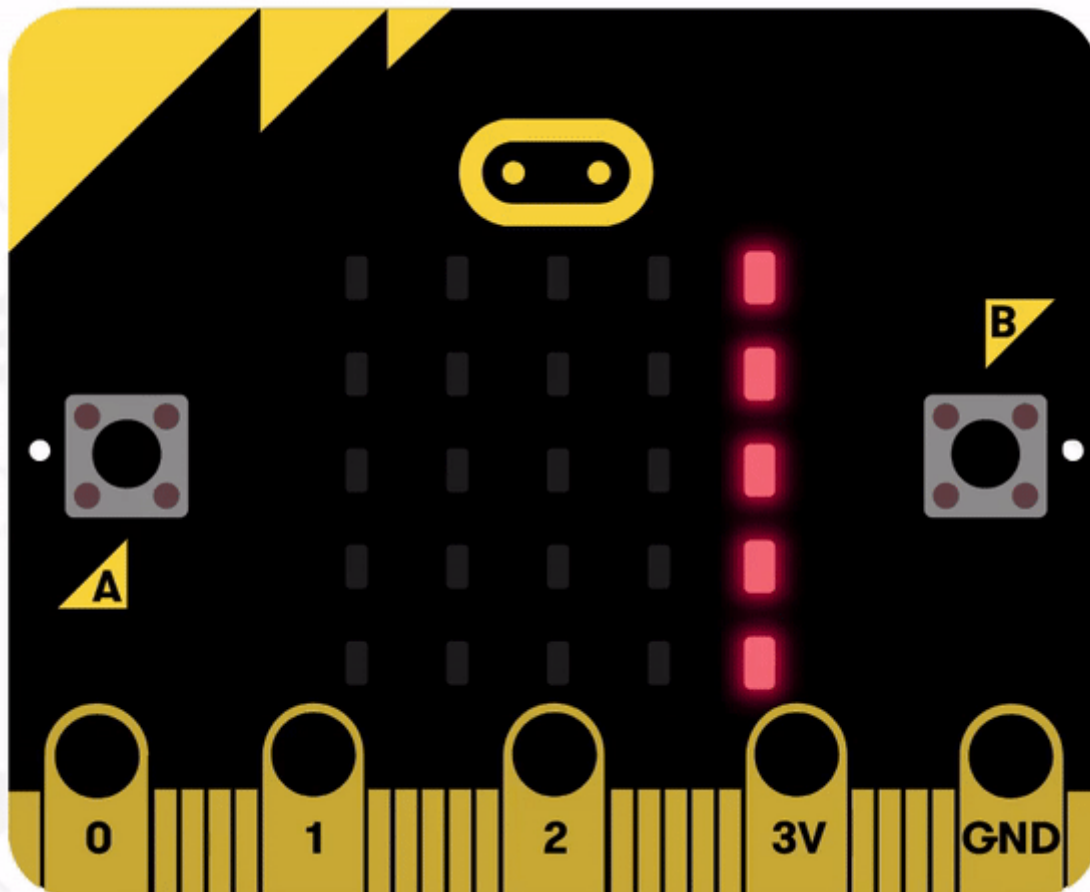
The key to tackling every programming problem is to break it into bite-sized achievable bits. Let's look at what we'll need. Refer to the video to see a demo of the game. As we go through the process, let's ask what elements are within the game. A 'READY' message and countdown shows when the screen starts.

Create a coordinate to indicate the bird. Move the bird around by pressing button A. Keep track of the number of pipes the bird passes. Create walls for the bird to fly past. When the bird collides with a wall, the game is over. You might already know how to do some of these.

Try covering these steps on your own first. If necessary, break the steps down further into smaller steps. There are also game checks which should be the progress you've made by that step. Use these to make sure you're on track.

23.4. How to Make

Step 1 – Hello, World!



```
1  from microbit import *
2
3  # scrolls welcome message and starts countdown
4  display.scroll("READY")
5  display.show("3")
6  sleep(1000)
7  display.show("2")
8  sleep(1000)
9  display.show("1")
10 sleep(1000)
11 display.clear() # clears welcome message
```

First step is to import the micro:bit library into Python. Then, let a 'READY' message scroll across the screen and initiate countdown that shows when the game starts. Line 1: This imports the micro:bit program Line 4: This initiates the 'READY' message that scrolls across the screen. Double quotation marks indicate a string (in this case 'READY') Lines 5-10: This flashes each number on the screen for 1 second (or 1000 milliseconds, the measurement

involved) by using the `sleep()` function. Line 11: clears the screen for us to draw the bird and walls later on. Note: It's always good to add comments to explain your own code for others or yourself to understand when coming back to it. You add a comment with '#'. Also space out your code when necessary to indicate different segments that do different things. What you're doing is applying functions to the object display such that the LCD screen lights up. In Python, you also have the flexibility of slowing down the scrolling rate of text in line 4. `display.scroll("READY", delay = 200)` scrolls the text twice as fast and `display.scroll("READY", delay = 800)` scrolls the text at half the speed. The standard delay setting is 400. Increasing the value decreases scroll speed and decreasing the value increases scroll speed. Congratulations! You finished the pre-game message! Next, we have to actually set up the game for the user to play.

Step 2 – Fly, Birdie!

```
13  y = 50 # set position of bird
14
15  while True:
16      # draw bird
17      led_y = int(y / 20)
18      display.set_pixel(1, led_y, 7)
19      sleep(20)
```

Next, we have to create the image of the bird. For those who never analysed the game, Flappy Bird only allows the bird to move upwards and downwards, and pushes it at a constant speed towards the walls. Of course, our screen only has 5 rows of LED so it's quite limited. To make the bird-flapping more realistic, we'll be splitting these 5 segments into 100 different positions. This gives us more flexibility when adding speed of descent later on. In this case, the top of the screen is position $y=0$ and the bottom is position $y=99$ so there are 100 positions. The start position is $y=50$. Line 13: This sets the start position of the bird right in the middle, as $y=0$ is the top and $y=99$ is at the bottom. Line 17: This determines the actual position of the bird on screen, since there are 100 positions and 5 LED rows. Hence, you divide the value stored in variable y by 20 so you scale the bird down onto the screen. Line 18: This displays the bird on the screen using the `display.set_pixel` function, which has 3 parameters: x , y and brightness. The x -coordinate is 1 so it will appear in the second column. The y -coordinate is presently 2 because we divided 50 by 20 and rounded it down. That's the third row. (Note: Indexes begin at 0 usually for computer programming, so you have rows 0-4 from above to below and columns 0-4 for left to right.) Brightness can be any integer from 0 to 9, with 9 being the brightest. In this case, 7 will suffice to avoid eye-strain. We add a while loop to tell the micro:bit to keep repeating the block of code that is indented. (Python uses indentations to separate code.) The sleep code tells the micro:bit to run this loop every 20ms so it makes your game far more manageable and makes sure your CPU doesn't work too hard and crash the browser, which would otherwise happen. Game check: At this point, a welcome message should appear, then disappear for a bird to appear.

Step 3 – Leaving The Nest

```

13  y = 50 # set position of bird
14  speed = 0 # sets rate of falling
15
16  while True:
17      display.clear()
18
19      # accelerate to terminal velocity
20      speed += 1
21      if speed > 2:
22          speed = 2
23
24      # move bird, but not off the edge
25      y += speed
26      if y > 99:
27          y = 99
28      if y < 0 :
29          y = 0
30
31      # draw bird
32      led_y = int(y / 20)
33      display.set_pixel(1, led_y, 7)
34      sleep(20)

```

The previous step only created the bird image, but it still can't move! This is what we will do in the next step, by simulating realistic gravity. Firstly, let's add a new variable 'speed' right below the y-coordinate. Shift the `display.clear()` into the while loop such that it no longer just clears the welcome message, but also clears the old position of the bird, as it runs before the new position is set each time. Lines 25-29: This sets a new y-coordinate of the bird within the borders (max y=99, min y=0), based on the 'gravity' acting at that point. Why place it all in the while loop? Well, you want this block to continually update the position of the bird every few milliseconds (20 to be exact) so this block will keep repeating itself. Terminal velocity: to make the motion of the bird more realistic, speed reaches a constant rate of 2, but only after two iterations of the code whereby speed = 0 becomes speed =2. The if function ensures that speed does not increase beyond 2. You can play around with this to vary the speed of bird descent.

Step 4 – Defying Gravity

```

13  y = 50 # set position of bird
14  speed = 0 # sets rate of falling
15  score = 0 # sets initial score
16
17  while True:
18      display.clear()
19
20      # flap if button a was pressed
21      if button_a.was_pressed():
22          speed = -8
23
24      if button_b.was_pressed():
25          display.scroll("Scr:" + str(score))
26
27      # accelerate to terminal velocity

```

Now, we have to get the bird to hop by pressing button A. In this step, we also include a new 'score' variable to track the number of walls that the bird flies past. This can be accessed at any point using button B. To react to key-pressing of A, run 'button_a.was_pressed()' under an if-loop like in line 21. If, during that iteration, the A button was pressed at any time, we bring the bird up, reset the falling rate, then let it accelerate back down to the ground, giving the falling and flapping motion. Change the value of speed on flapping, which is currently -8, to see the visual changes to rate of bird's descent. Add variable 'score = 0' to set new variable score to 0, underneath the speed and y variables. As a coding habit, try to set all your variables in one place, above the code that uses it so it's easier to follow, and actually can be inputted for use. Show score when button B is pressed by creating an if loop similar to button A. display.show(score) shows the score at any point in time. We'll learn to vary and count the score after each wall-passing later. Game check: Welcome message appears, disappears, then bird appears that falls down. Press A for it to flap upwards and B to check the score, which should remain at 0 right now.

Step 5 – Pipe Blaster

```

18  # make an image that represents a pipe to dodge
19  def make_pipe():
20      i = Image("00004:00004:00004:00004:00004")
21      gap = random.randint(0,3) # random wall gap position
22      i.set_pixel(4, gap, 0) # blast a hole in the pipe
23      i.set_pixel(4, gap+1, 0) # make hole bigger
24      return i
25
26  # create first pipe
27  i = make_pipe()
28
29  while True:
30      # show pipe
31      display.show(i)

```

We're going to create our first pipe using a make_pipe function! Then we'll assign it to variable i, and show pipe within the while loop. I know it's complicated, but it'll also be the start of owl/our game finally looking complete! Functions are blocks of code that are run

conveniently under the function name. By calling a function, we can run the entire block of code within it. This makes it easier to understand what we're doing at each step. In this case, we'll name our function `make_pipe()` which runs code to make a new pipe each time. Let's break down what each step of the `make_pipe()` function does At line 19, we define the function using `def make_pipe():` – the indented blocks beneath make up the function At line 20, a custom image is drawn, with the '0' indicating 0 brightness for each coordinate, starting from row 1, column 1 then row 1, column 2 and so on. This basically lights up the LED of the entire last column with the brightness of 4. (You can tweak this as you like. I personally like for the bird to be clearly brighter than the wall so you can identify its position.) At line 21, we use the random library to call a random number between and inclusive of 0 and 3. This means 0, 1, 2 and 3. We don't use 4 because we blast two holes, one which is `gap+1`. If 4 was selected, we would blast a hole in column 4, row 5. But there's no row 5 so an error is returned. We have to return this image so that it can be called as the value of `i` later on. The hole is blasted by setting the LED brightness for the `gap` position and the LED above it to be zero. Pretty cool, eh? That's your first function. Good job! Note: always define the functions above the actual code, beneath the variables. This is just a convention, but it makes your program readable! Let's assign variable `i` to the function, as per line 27. Now, in the while loop, if we add a `display.show(i)`, the display now shows the pipe (and hole) `i`. Persevere! We're nearly there. Now, we just have to get the wall moving, count scores and react to bird-wall collisions. Game check: Same as step 4, and now there's an unmoving wall with holes! Check earlier steps if something has gone awol.

Step 6 – Frame Rate

```
14 # game constants
15 DELAY = 20 # ms between each frame (not used, just for programmer's info)
16 FRAMES_PER_WALL_SHIFT = 20 # number of frames between each time a wall moves a pixel to the left
17 FRAMES_PER_NEW_WALL = 100 # number of frames between each new wall
18 FRAMES_PER_SCORE = 100 # number of frames between score rising by 1
19
20 y = 50 # set position of bird
21 speed = 0 # sets rate of falling
22 score = 0 # sets initial score
23 frame = 0
24
25 # make an image that represents a pipe to dodge
26 def make_pipe():
27     i = Image("00004:00004:00004:00004:00004")
28     gap = random.randint(0,3) # random wall gap position
29     i.set_pixel(4, gap, 0) # blast a hole in the pipe
30     i.set_pixel(4, gap+1, 0) # make hole bigger
31     return i
32
33 # create first pipe
34 i = make_pipe()
35
36 while True:
37     frame += 1
```

This step is where we set up the game constants. Here, the frame variable starts at 0, then increases by 1 every 20ms so it takes 400ms or 0.4s for the frames variable to increase by 20. Remember this, it'll be easier for the incoming math. These constants aren't used until Step 7, but let's set them up first. Line 15 just indicates the time taken (in ms) for frame to increase by 1, which is added as part of the while loop in line 37 (`frame += 1`). You can change the `sleep(20)` at the bottom of the code to `sleep(DELAY)` so it corresponds. Line 16 sets the

time taken for the wall to shift by 1 column. This is currently 0.4s or 20 frames. Line 17 sets the time between the occurrence of another wall. This is currently 2.0s or 100 frames. Line 18 sets the time between the score increasing. This should always be equivalent to the `FRAMES_PER_NEW_WALL` value so that each wall you pass is equivalent to one additional score. To make the game harder, you would adjust these game constants, perhaps reducing the distance between each new wall for more walls (but change `FRAMES_PER_SCORE` to correspond to it). The game is currently set for one wall on the screen at any time, but you can definitely make it more chaotic by playing around with the values. Note: The game constants are in uppercase, differentiating them from the other variables used. These are just [standard rules](#) for Python programming. It'll still work without following it, but your code should follow conventions to be readable.

Step 7 – Pipe Dreams

```
61     # draw bird
62     led_y = int(y / 20)
63     display.set_pixel(1, led_y, 9)
64
65     # move wall left
66     if frame % FRAMES_PER_WALL_SHIFT == 0:
67         i = i.shift_left(1)
68
69     # create new wall
70     if frame % FRAMES_PER_NEW_WALL == 0:
71         i = make_pipe()
72
73     # increase score
74     if frame % FRAMES_PER_SCORE == 0:
75         score += 1
76
77     sleep(DELAY)
```

Here, we will compare the frame value with game constants to move the wall left, create a new wall and increase the score. This is all within the while loop so it's checked every 20ms. Ready? Let's go. At this step, we'll use the modulo sign (%). This provides the remainder when a number is divided by another number. So $4 \% 2$ returns 0 but $4 \% 3$ returns 3. Here, we'll use it to check that the frame variable is equal to any of the game constants. Moving wall left: Look at lines 65-67. This means the wall shifts when the frame is equal to 20, 40, 60... since they're divisible by `FRAMES_PER_WALL_SHIFT` value of 20. You can vary this to make the walls move faster and increase the difficulty. Currently, the walls move every 0.4s. Creating new wall: Look at lines 69-71. Every 100 frames, or 2 seconds, a new pipe is made by calling the `make_pipe()` function for `i`. This is the constant used to create and move the wall. Increasing the score: look at lines 73-75. This means that a point is added when the bird travels for 2 seconds, or 1 wall. This value corresponds with the distance between walls so each wall passed is one point. Game check: The game should be almost fully playable, with the welcome message, then the bird moving by pressing button A. You can see score with

button B. There's gravity acting on the bird so it falls down over time. Then the walls created randomly move right past it. Wow, you're nearly done! Now, we just have to react to pipe collisions, ending the game and revealing the score when the bird collides with any pipe.

Step 8 – Collision Course

```
61     # draw bird
62     led_y = int(y / 20)
63     display.set_pixel(1, led_y, 9)
64
65     # check for collision
66     if i.get_pixel(1, led_y) != 0:
67         display.show(Image.SAD)
68         sleep(500)
69         display.scroll("Score: " + str(score))
70         break
71
72     # move wall left
73     if frame % FRAMES_PER_WALL_SHIFT == 0:
74         i = i.shift_left(1)
```

Phew, you made it to the last step! Ready to wing it? Now, we just need to add a collision reaction. This uses a `get_pixel` function that returns the LED brightness value at that position. `!=`, the NOT function is also used. Let's explain how it's used below. Add this collision checking code to the while loop, between the bird-drawing and wall-shifting. This means it checks for collision before new walls are created so there's no extra scores by error. As shown in line 66, we use an if loop. `i.get_pixel(1, led_y) != 0` checks if there is a pipe in the position of column 1 (where the bird is), specifically at `led_y`, the displayed position of the bird. If there is a pipe pixel in the same position as the bird's coordinates, the `i.get_pixel(1, led_y)` returns 4, the brightness of the wall. This is NOT 0 so the function beneath, the collision checker, runs Line 67-68 display the in-built sad face image for 0.5s. You can change how long this lingers, and to whatever other image you like. Python has a lot of images you can input. You can find the entire list [here](#). Line 69 displays the score as a string, behind "Score". Line 70 ends the while loop so the game ends. This means that it's 'game over'.

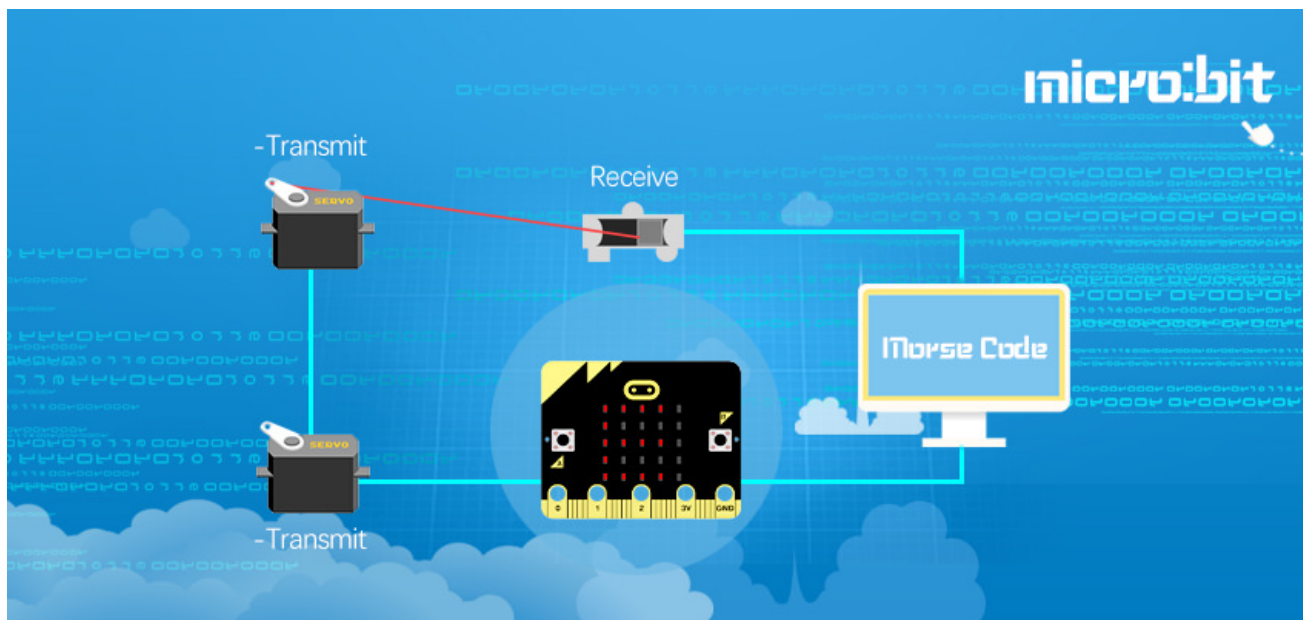
Start Game!

And... that's it! You're done. Your game should be able to run and end, revealing the score at the end. It's now a full-fledged frustratingly simple yet challenging game. Pat yourself on the back! That was a lot of hefty coding and new concepts. Look through your code, and try and figure out what each line. Add comments to explain it to yourself if necessary. This is a good practice for you to easily read your own code when coming back to it months later.

Good job! Have fun frustrating your friends with this novel interface for the annoying game. Now, you're free as a bird to look for other projects, with a better understanding of the Python code. Extension: Add a game loop, such that you can play again without resetting the

device. I suggest changing the while loop's requirements from True to a certain variable, a play_again function which can be changed with the press of a button. Look at other Python game loops for inspiration, like a scissors, paper, stone game.

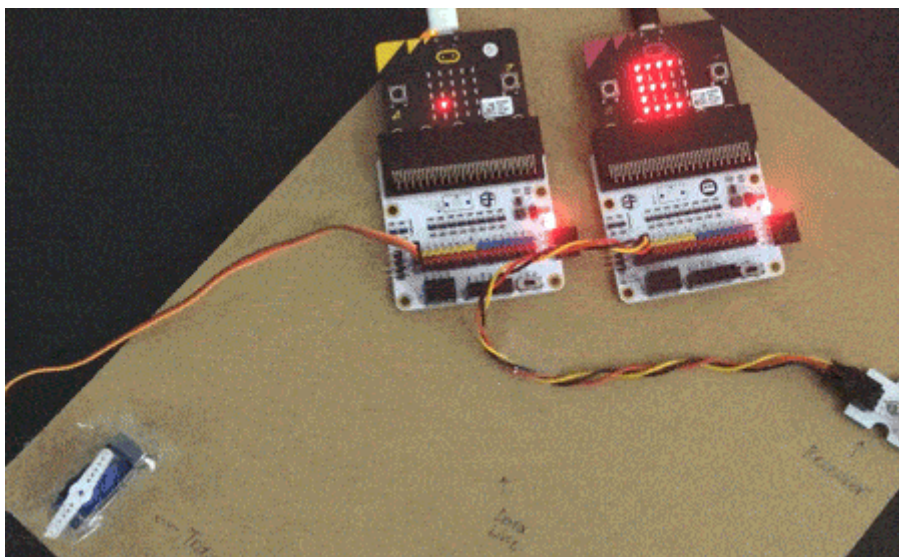
24. case 22 Wire Transmission



Communicate between two micro:bits using Morse code, fishing line, a servo and a sensor!
Why use micro:bit's radio when this is so much cooler?

24.1. Goals

- Use Python to programme the micro:bit
- Use dictionaries to encode and decode Morse code
- Move the servo, and detect using the crash sensor



24.2. Materials

-
- 2 x BBC Micro:bit
 - 2 x Breakout board
 - 2 x Micro-USB cable
 - 1 x Servo
 - 1 x Crash Sensor
 - A thin string (e.g. fishing line)
 - Optional: Cardboard sheet

You can't see the string in this gif, but it's there between the servo and crash sensor!

24.3. Why Python?

- Reads like English – Python is one of the easiest languages to read, which makes it such a fantastic beginner's language.
- Versatile – Python is industry standard for good reason. It can be used to do so much. This is why Google and YouTube utilise the language for part of its back-end software.
- Active community – Python is one of the most popular languages for beginners. There are tons of resources and many more than willing to help look over your code, which will prove invaluable to helping you get over stumbling blocks in your coding journey.

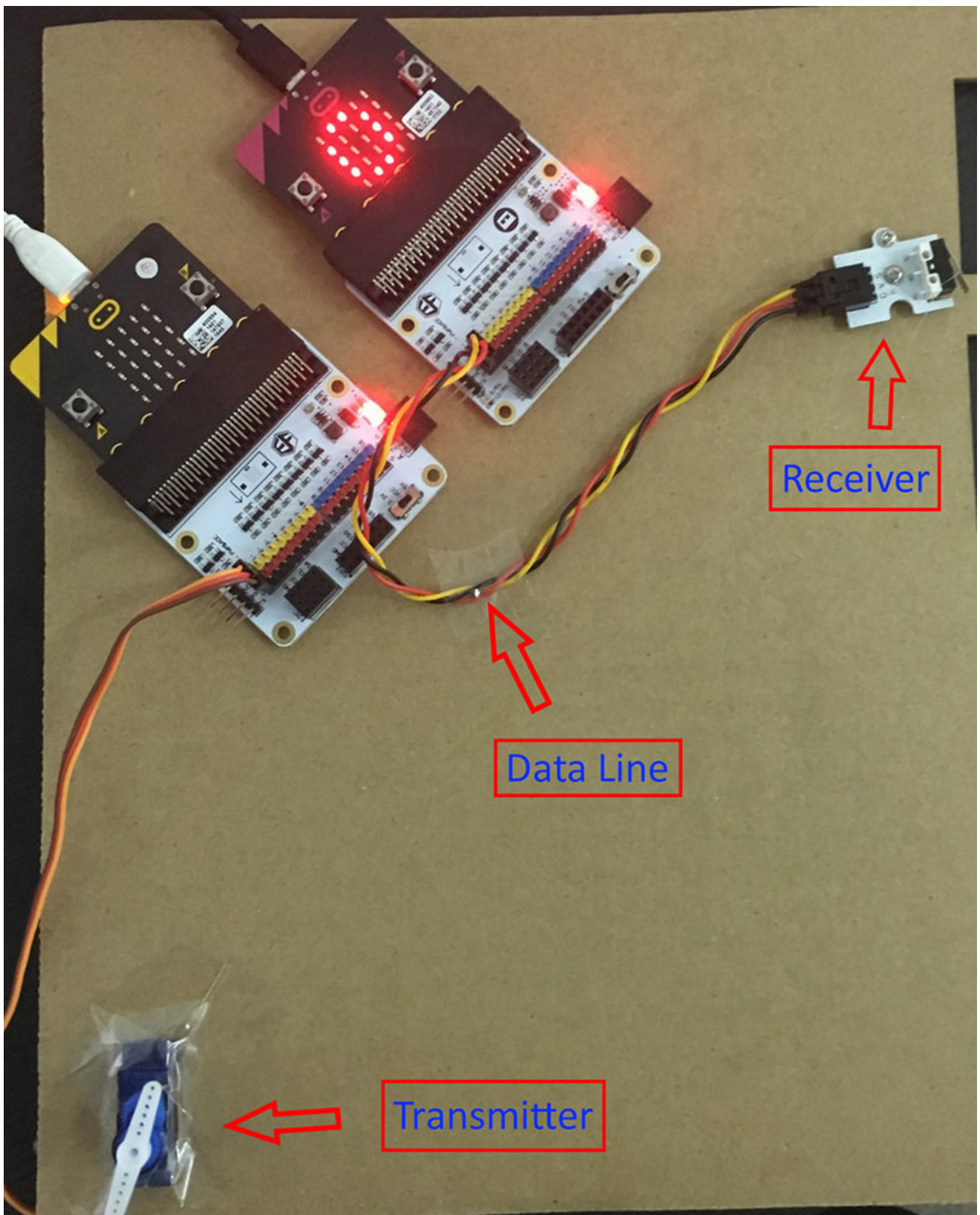
24.4. How Do I Start Coding in Python?

You can write your code in Python on the official micro:bit Python editor. To run a program, click the download button, and drag the .hex file into the MICROBIT drive connected to your computer.

24.5. Overview

We'll be using two micro:bits, one to transmit Morse code and one to receive Morse code. The transmission of data will be done over a length of string. As the servo tugs on the string (based on the encoded input), the crash sensor detects the tugging and decodes it from morse code into letters. Of course, you could transmit data over the radio component of the micro:bit, but where's the fun in that?

24.6. Physical Assembly



Attach the servo to the cardboard sheet, and tie the string around the end of the rotor attached to the servo. Tie the other end of the string around the metal flap of the crash sensor. Attach the crash sensor at a distance such that when the servo turns, the string is pulled and the sensor is activated. If you don't have a cardboard sheet, you could tape everything to a table. For the transmitting micro:bit, attach the servo to pin 0 on the breakout board. For the receiving micro:bit, attach the crash sensor to pin 0 on the breakout board.

24.7. What's Morse Code?

Morse code is a type of code used to transmit text by a combination of short (“.”, or “dit”) and long (“-”, or “dah”) signals. Every letter of the alphabet and number from 0 to 9 has its own Morse code representation. Letters are separated by pauses.

24.8. Transmitter

Step 1: Encoding Text into Morse Code

Suppose we are given the text “HELLO WORLD”, and would like to convert this into Morse code. First, we need to have a ‘table’ of what each letter’s morse code is, so that we could, for example, find that “E” is “.” and “W” is “.-”.

We can use one of Python’s data structures, the dictionary, which allows us to associate keys to values. In this case, the keys should be the letters of the alphabet, and the values should be the Morse code representation of the corresponding letter.

```
from microbit import *

MORSE_CODE = {'A': '.-', 'B': '-...', 'C': '-.-.',
              'D': '-..', 'E': '.', 'F': '..-.',
              'G': '--', 'H': '....', 'I': '..',
              'J': '.---', 'K': '-.-', 'L': '-.-.',
              'M': '--', 'N': '-.', 'O': '---',
              'P': '-.-.', 'Q': '-.-.', 'R': '-.-.',
              'S': '...-', 'T': '-.', 'U': '..-.',
              'V': '...-', 'W': '.--', 'X': '-.-.',
              'Y': '-.-.', 'Z': '--..', '0': '-----',
              '1': '.-----', '2': '..---',
              '3': '...--', '4': '....-', '5': '.....',
              '6': '-....', '7': '-...-', '8': '--...',
              '9': '---..'}

message_to_send = "HELLO WORLD" #insert your own message here
encoded_message = "".join([MORSE_CODE[letter] + ' ' for letter in message_to_send])
```

Here is a dictionary that should do the trick:

```
MORSE_CODE = {'A': '.-', 'B': '-...', 'C': '-.-.', 'D': '-..', 'E': '.', 'F': '..-.', 'G': '--', 'H': '....', 'I': '..', 'J': '.---', 'K': '-.-.', 'L': '-.-.', 'M': '--', 'N': '-.', 'O': '---', 'P': '-.-.', 'Q': '-.-.', 'R': '-.-.', 'S': '...-', 'T': '-.', 'U': '..-.', 'V': '...-', 'W': '.--', 'X': '-.-.', 'Y': '-.-.', 'Z': '--..', '0': '-----', '1': '.-----', '2': '..---', '3': '...--', '4': '....-', '5': '.....', '6': '-....', '7': '-...-', '8': '--...', '9': '---..'}


```

Now that we can translate each individual letter into Morse code, we should assemble the entire message, adding a space to the end of each letter to tell the receiver that a letter has been sent.

Step 2: Moving the Servo based on Morse Code

Once we've converted our message into the Morse code form, the next step is to move the servo based on the encoded message. In this case, dit will represent a 0.6s tug, dah a 1.2s tug, and a space a 1.6s tug.

First, we need to find the correct angles for the servos that will either tug on the sensor to activate it, or release the string to deactivate the sensor. We'll call these values `press_angle` and `release_angle`. For this set-up, their values are 150 and 60, but this will differ based on how you've positioned the sensor and servo.

To move the servo, we'll need to use a class, which can be obtained here. To use this class with the online editor, copy and paste this code at the start of the programme.

For each character (dit, dah or space), we should tug on the string for the appropriate length of time, and then release the string for a short period of time.

```
press_angle = 150
release_angle = 60

Servo(pin0).write_angle(release_angle)

for char in encoded_message: #repeat for each character in the encoded message
    display.show(char, wait=False)
    if char == '.':
        Servo(pin0).write_angle(press_angle) #tug on the string for 0.6s
        sleep(600)
    elif char == '-':
        Servo(pin0).write_angle(press_angle)
        sleep(1200)
    else:
        Servo(pin0).write_angle(press_angle)
        sleep(2000)
    Servo(pin0).write_angle(release_angle) #release the string for 1s
    sleep(1000)
```



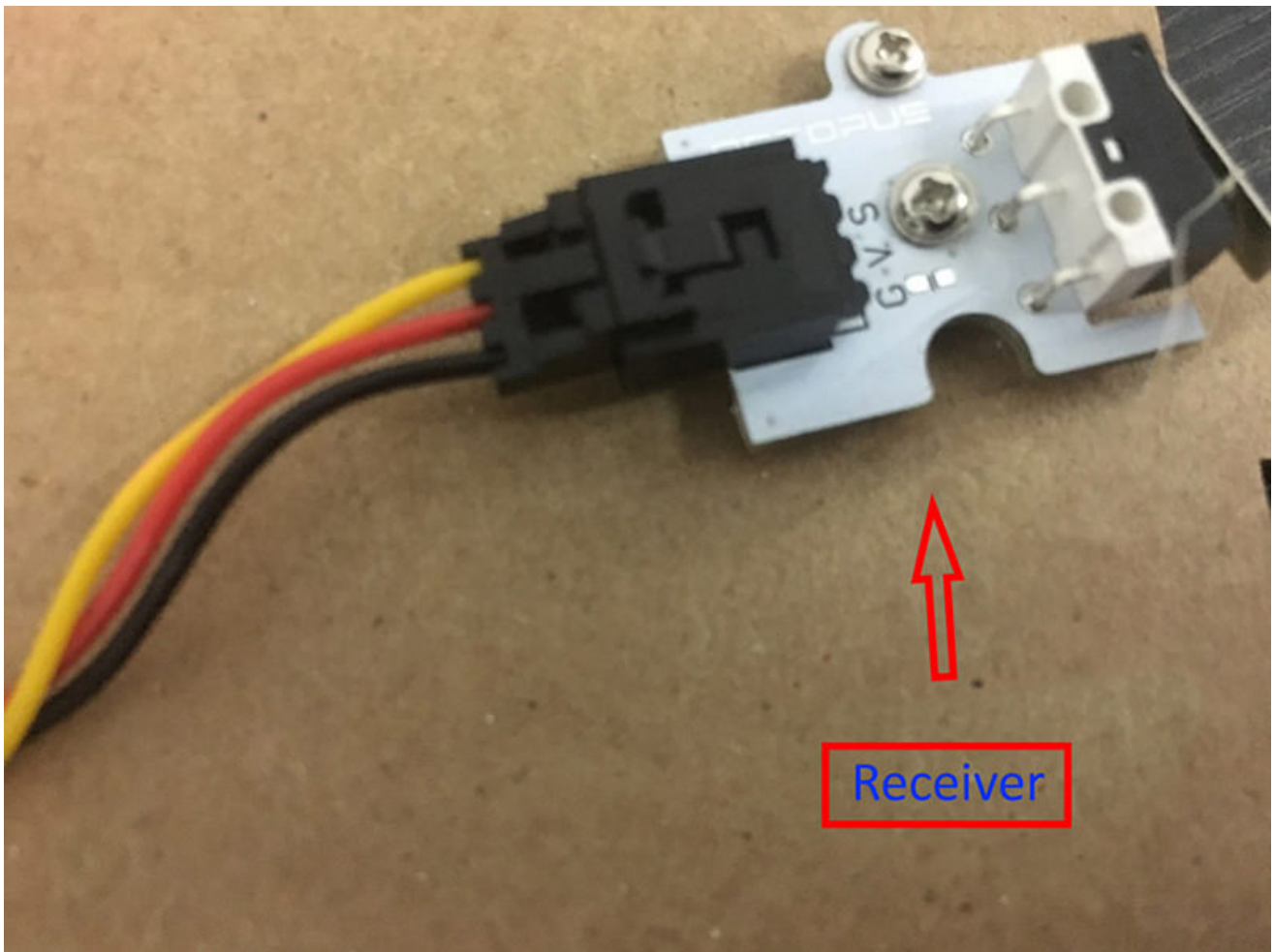
24.9. Receiver

Step 1: Translating Sensor Data into Morse Code

When the string tugs on the sensor, it will press the flap down, and this can be detected using analog input. Whenever the flap is down, the analog reading of the pin drops below a threshold value. In this case, we'll use a threshold value of 100.

While we could use event listeners that trigger events when the flap is pressed, it'll be easier to perform polling, which means checking the analog reading at a certain interval, in this case 0.1s.

If in a cycle, the flap is being held down, we'll increase the `press_length` by 100, to keep track of how long the flap has been pressed so far. If the flap is found to be released, we can use `press_length` to figure out how long the button has been pressed, and use it to determine what character (dit, dah or space) has been transmitted. We'll add this to the variable `cur_letter`, which keeps track of the dits and dahs that have been sent over so far.



```
cur_letter = "" #current string of dits and dahs
press_length = 0
while True:
    if pin0.read_analog() < 100: #string is being tugged
        press_length += 100
    elif press_length != 0: #string has been released
        if press_length < 600:
            cur_letter += "."
        elif press_length < 1200:
            cur_letter += "-"
        else: #letter complete
            #decode current letter here
    else:
        press_length = 0

    sleep(100)
```

Step 2: Translating Morse Code into Letters

Every time a space is detected, it should take the characters (dits or dahs) detected so far, and convert that into a letter. We'll need to use a dictionary again. This time the keys should be the Morse code representation, and the value should be the letter of the alphabet.

Here's the code for the decoding dictionary:

```
MORSE_DECODE = {'.-': 'A', '-...': 'B', '-.-.': 'C', '-..': 'D', '.': 'E', '..-': 'F', '-': 'G', '...': 'H', '.': 'I', '-.-': 'J',
'.-': 'K', '-..': 'L', '-': 'M', '-': 'N', '-': 'O', '-.-': 'P', '-.-': 'Q', '-.-': 'R', '...': 'S', '-': 'T', '-.-': 'U', '...': 'V', '-.-':
'W', '...': 'X', '-.-': 'Y', '-.-': 'Z', '-.-': '0', '-.-': '1', '-.-': '2', '...': '3', '...': '4', '...': '5', '-.-': '6', '-.-':
'7', '-.-': '8', '-.-': '9'}
```

```
from microbit import *
```

```
MORSE_DECODE = {'.-': 'A', '-...': 'B', '-.-.': 'C',
'.-..': 'D', '.': 'E', '..-': 'F',
'.--.': 'G', '...': 'H', '.': 'I',
'.--.-': 'J', '-.-': 'K', '-.-': 'L',
'.--': 'M', '-.-': 'N', '-.-': 'O',
'.--.-': 'P', '-.-.-': 'Q', '-.-': 'R',
'.--.-': 'S', '-': 'T', '-.-': 'U',
'.--.-': 'V', '-.-': 'W', '-.-.-': 'X',
'.--.-': 'Y', '-.-..': 'Z',
'.--.-': '0', '-.-.-': '1', '-.-.-': '2',
'.--.-': '3', '-.-.-': '4', '-.-.-': '5',
'.--.-': '6', '-.-.-': '7', '-.-.-': '8',
'.--.-': '9'}
```

```
cur_letter = "" #current string of dits and dahs
cur_char = "" #current character to display
press_length = 0
```

```
while True:
    display.show(cur_char, wait=False)
    if pin0.read_analog() < 100: #string is being tugged
        press_length += 100
    elif press_length != 0: #string has been released
        if press_length < 600:
            cur_letter += "."
        elif press_length < 1200:
            cur_letter += "-"
        else: #letter complete
            if cur_letter in MORSE_DECODE.keys():
                cur_char = MORSE_DECODE[cur_letter]
            else:
                cur_char = "?"
    else:
        press_length = 0
    sleep(100)
```


Now, whenever a letter is detected (a space is pressed), we can look in the decode dictionary to obtain the original letter. However, sometimes the receiver may not correctly detect the sequence of string tugs, and so the sequence cannot be found in the dictionary. If we try to look for a sequence that cannot be found in the dictionary, Python will throw an error and the programme will stop executing.

Hence, we should first check if the sequence exists in the dictionary's keys, and if it does not, we'll set the current character to "?". Once we have the current character, we can display it on the LEDs, by setting the `cur_char` variable. At each cycle, we'll display the character detected.

24.10. Putting it all Together

If the set-up doesn't work flawlessly at first, that's fine! Try adjusting the positions and orientations of the servo or sensor, as well as the press and release angles of the servo. Also, you can try adjusting the durations of the tugs.

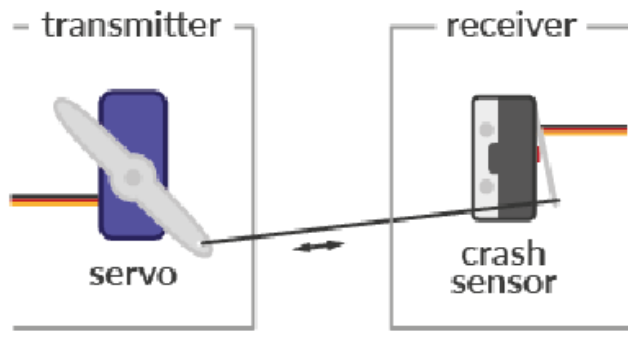
Here is the full code for the [transmitter](#) and [receiver](#).

24.11. Extensions

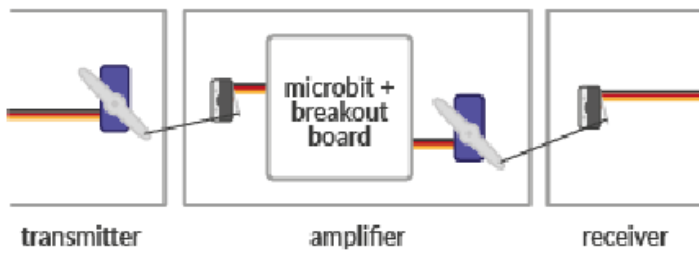
Although this method of data transmission isn't used for ...obvious reasons, many concepts in data transfer are relevant. Try to experiment with the length of string to see how long distance can be reliably transferred, and at what point the "signal" becomes too weak to be detected.

To boost the "signal", a third micro:bit can be used as an amplifier that converts sensor signals into new tugs, similar to how signal amplifiers are installed every 20km in underwater fibre-optic cables.

Morse code certainly isn't the most efficient way to transmit data, nor is it the most reliable way. Experiment with different types of encodings (binary + ASCII, Hamming codes, etc.), as well as explore some error-correcting codes to detect and fix any losses/errors in transmission.



**Morse Code
Transmitter &
Receiver**



**Morse Code
Amplifier
(for longer
distances)**

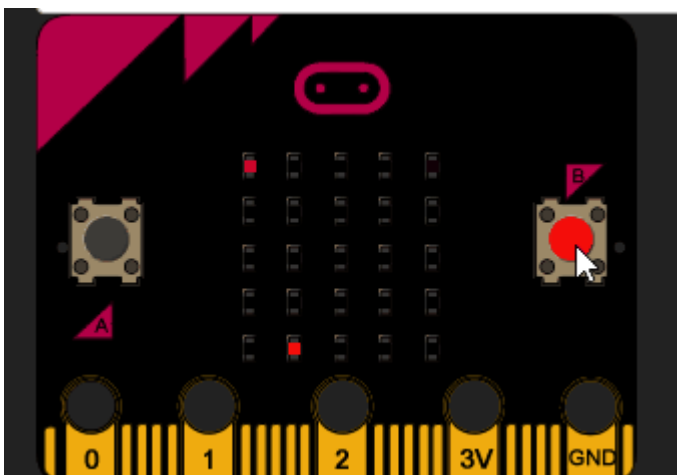
25. case 23 Snake Game



Anyone remembers the Snake game that used to come with old Nokia phones? This micro:bit version in glorious 5×5 resolution is easy to make and fun to play!

25.1. Goals

In this step by step guide, we will build a snake game from scratch, handling controls, movement, win and lose conditions, as well as the drawing of the game board.



25.2. Materials

- 1 x BBC micro:bit
- 1 x Micro USB Cable
- 1x Patience (coding should take approx. 30 min)

25.3. Why Python?

- Reads like English – Python is one of the easiest languages to read, which makes it such a fantastic beginner’s language.
- Versatile – Python is industry standard for good reason. It can be used to do so much. This is why Google and YouTube utilise the language for part of its back-end software.
- Active community – Python is one of the most popular languages for beginners. There are tons of resources and many more than willing to help look over your code, which will prove invaluable to helping you get over stumbling blocks in your coding journey.
- Actual coding looks cooler than block-based drag-drop coding. I know it’s intimidating, but look at these colours!

25.4. How Do I Start Coding in Python?

If you’re a fledgling to programming, you probably don’t have Python lying around. Don’t worry! Just go to the [official micro:bit Python editor](#) or download the offline Python editor [mu](#) to write code and send it to your micro:bit. You can also use your own text editor (three cheers to Sublime 3 and Atom) but you have to flash it to the micro:bit. This might turn out to be quite troublesome. Alternatively, you can use a [micro:bit simulator](#), which is really useful to test code out without downloading the .hex file each time, and makes it easier to fix errors.

Once set up, connect your micro:bit to your computer using the micro-USB cable. It should connect to the port at the top of the backside of the micro:bit. Once ready to be flashed, the micro:bit should light up bright yellow. Ignore this step if you’re on the simulator. Otherwise, stop reading and set it up if you haven’t already.

25.5. Six Simple Steps to SNAKE!

By breaking the code into separate portions, each aspect of the game can be tested individually to ensure that they are all functioning as they should.

- Import libraries
- Initialize variables
- Create the main loop
- Display snake and food
- Move snake every frame

- Set win and game over conditions

By checking the code constantly, we can be sure that what has been written so far is correct.

25.6. How to Make

Step 1 – Import

All necessary libraries for the project.

Since the project is a fairly simple one, we just need the default micro:bit library and this nifty function called `randint` that produces the random numbers we need.

```
from microbit import *  
from random import randint
```

Step 2 – Initialize

All the variables we will need later.

A point on the board is represented by a list `[x, y]` with `x` representing the column and `y` representing the row. The snake is a list of these points (yes, a list of lists!) as it contains more than one point. It starts as a single pixel at the top left of the screen, denoted by `[0,0]`. After which, more points will get appended to the list. The food is a single pixel positioned randomly somewhere else (not in the same row or column).

Each direction is represented by a list containing an increase/decrease in the column, or increase/decrease in the row (In essence, a vector). For example, right is represented by `[1, 0]` – an increase in the column by one, and no increase in the row. The snake is moving right by default, which is the first option in the list of directions. For the snake to turn leftwards, we simply go to the next direction in the list (right -> up -> left -> down -> right). For the snake to turn rightwards, we go to the previous direction in the list.

```
snake = [[0,0]]  
food = [randint(1,4),randint(1,4)]  
#right is [1,0], up is [0,-1]  
#left is [-1,0], down is [0,1]  
directions = [[1,0],[0,-1],[-1,0],[0,1]]  
direction = 0
```

Step 3 – Create


```

#determine location of the next position to move to
next_block = [(snake[0][0] + directions[direction][0]) % 5,
              (snake[0][1] + directions[direction][1]) % 5]

#if collision occurs
if next_block in snake:
    display.scroll("Game Over")
    break
#add next block to front of snake
snake = [next_block] + snake
#if food is eaten
if next_block == food:
    while food in snake:
        food = [randint(0,4),randint(0,4)]
#if food not eaten
else:
    snake.pop()

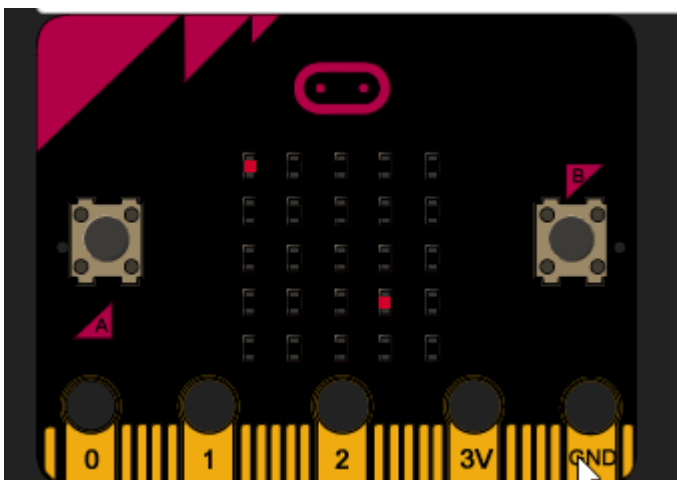
```

The whole code should be placed on top of the previous display code. (See completed code for reference). The first line determines the next pixel the snake will move to. Based on the current location of the head of the snake and adding the direction (in terms of row and column), we can find the next pixel. By obtaining the modulo 5, we can wrap the snake around the edge of the board.

What happens if this next block is already occupied by the body of the snake? In this case, a collision happens and the game ends. Note that break stops the while: True loop from running.

The next block is now made the new head of the snake. Next, we check if a piece of food has been eaten. If so, then a new piece of food should be generated. If not, the tail of the snake should be removed so that the snake is moving, not simply growing longer.

Run the code! Become infuriated as you realize that there is no way to win the game.



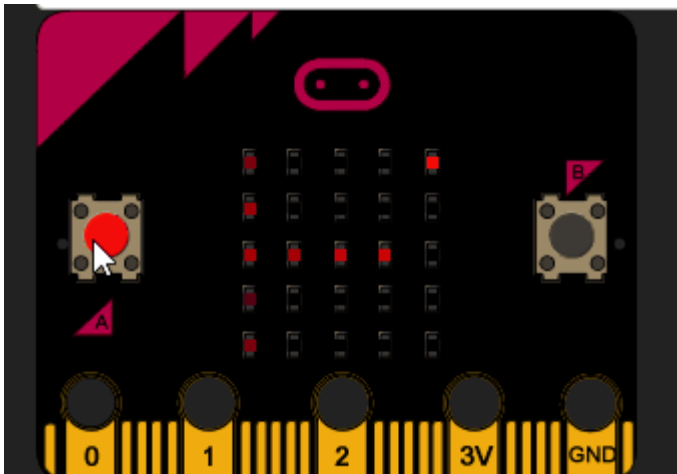
Step 6 – Win the game!

This code should be placed on top of the display code, but below the movement code. (See completed code for reference). What it does it continually check if the snake contains twenty five pixels, which is the entire board. If that is the case, the player wins!

```
#win condition
if len(snake) == 25:
    display.scroll("You Win")
    break
```

Congratulations!

Enjoy your fully functional snake game.



26. case 24 Game bit

26.1. Put together the Game:bit!

- Let's figure out where all those screws are supposed to go.



Goals

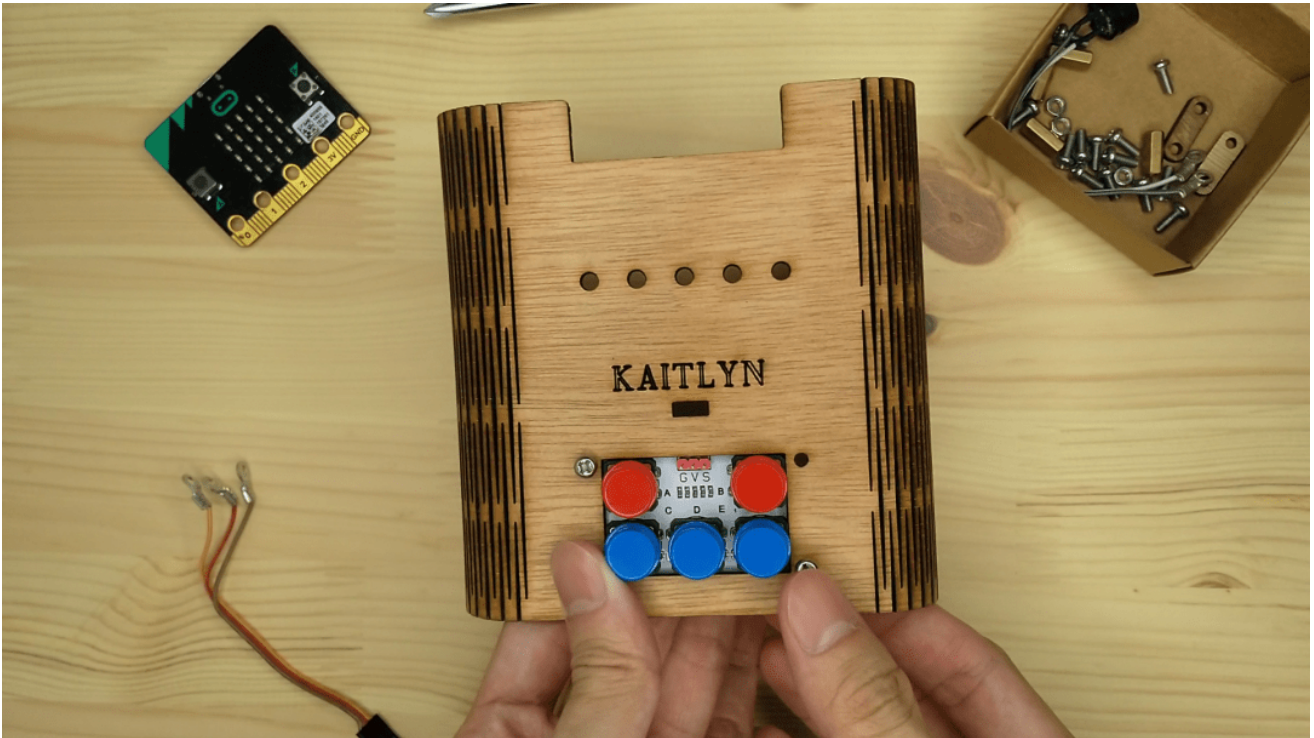
- Assemble the game:bit.
- Try not to break it.
- Helpful Hint: Toggle through pictures for each step for more photographic detail!

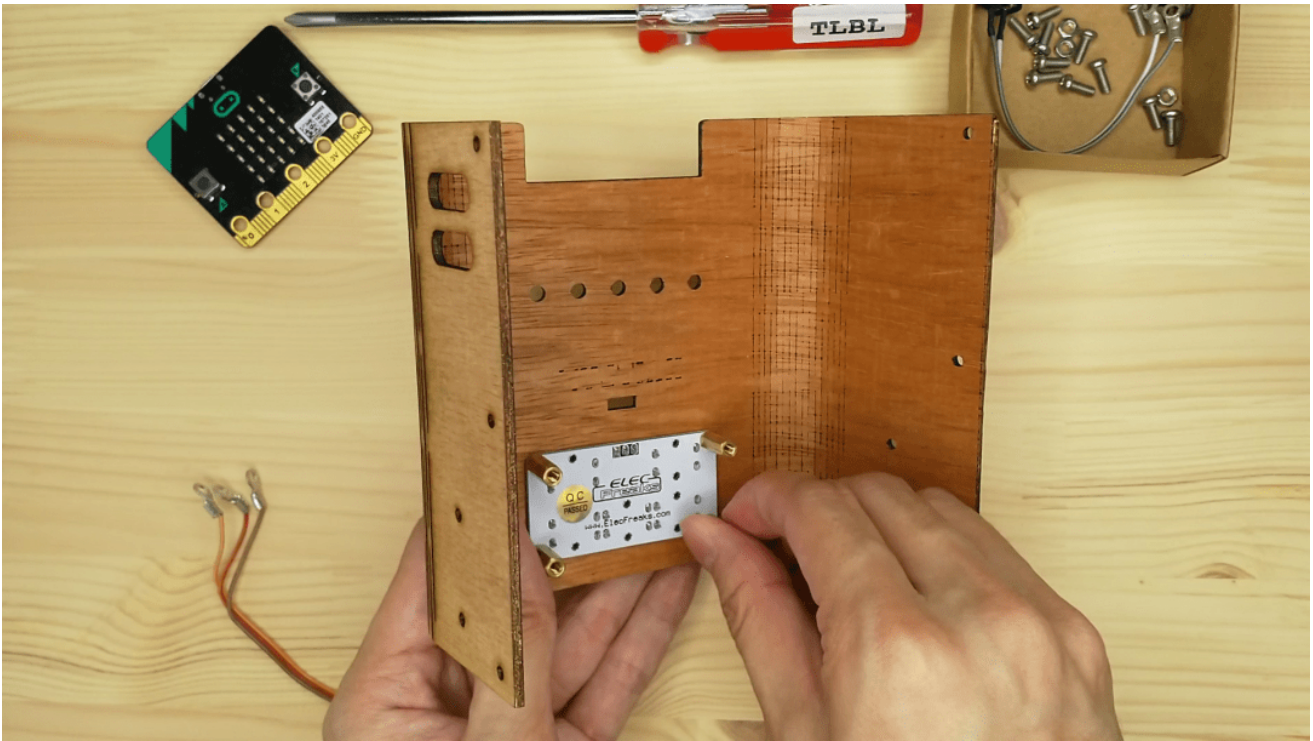
Materials

- 1 x Game:bit kit

- 1 x Screwdriver

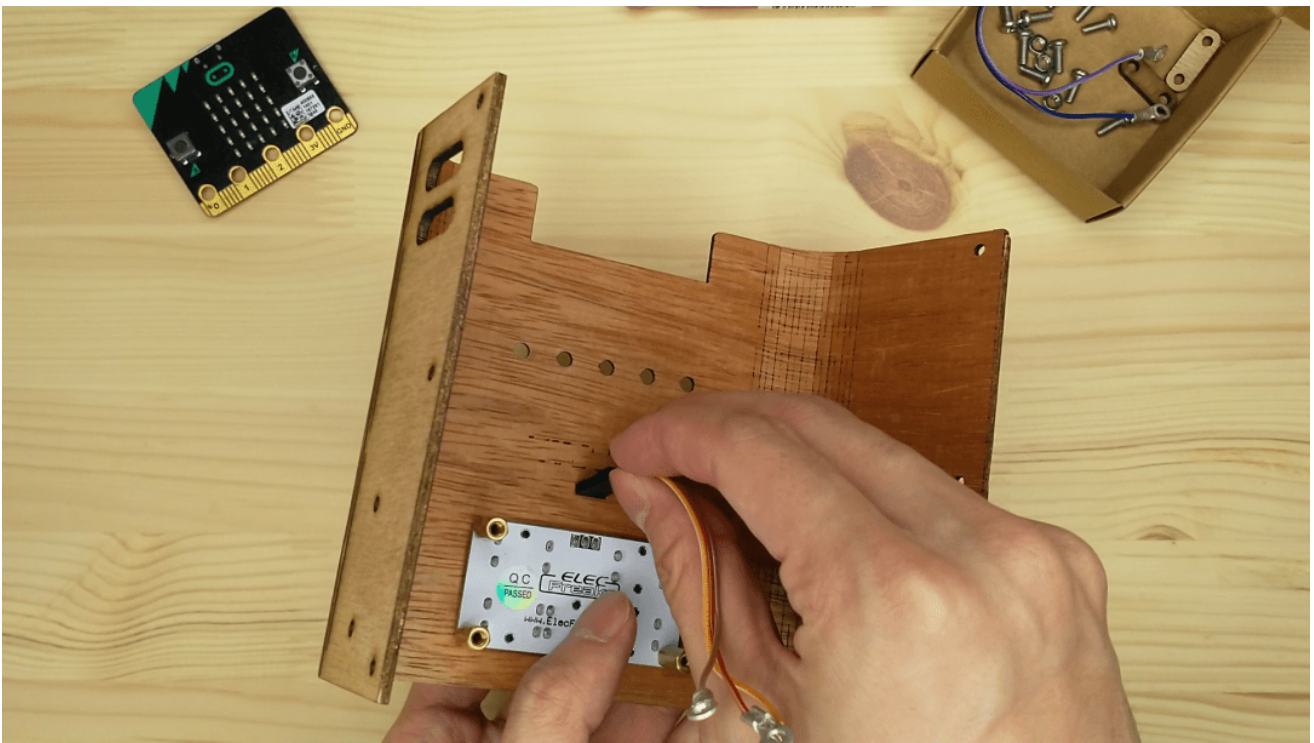
Step 1 – Buttons!





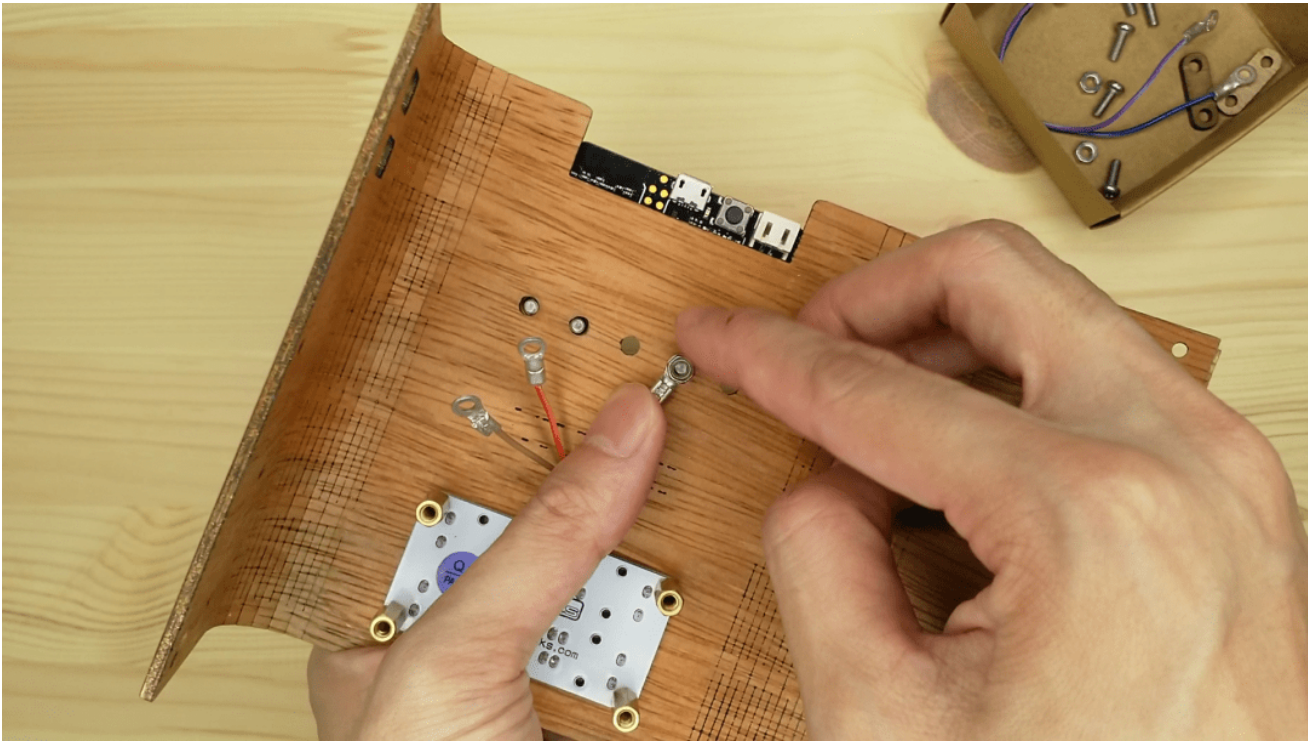
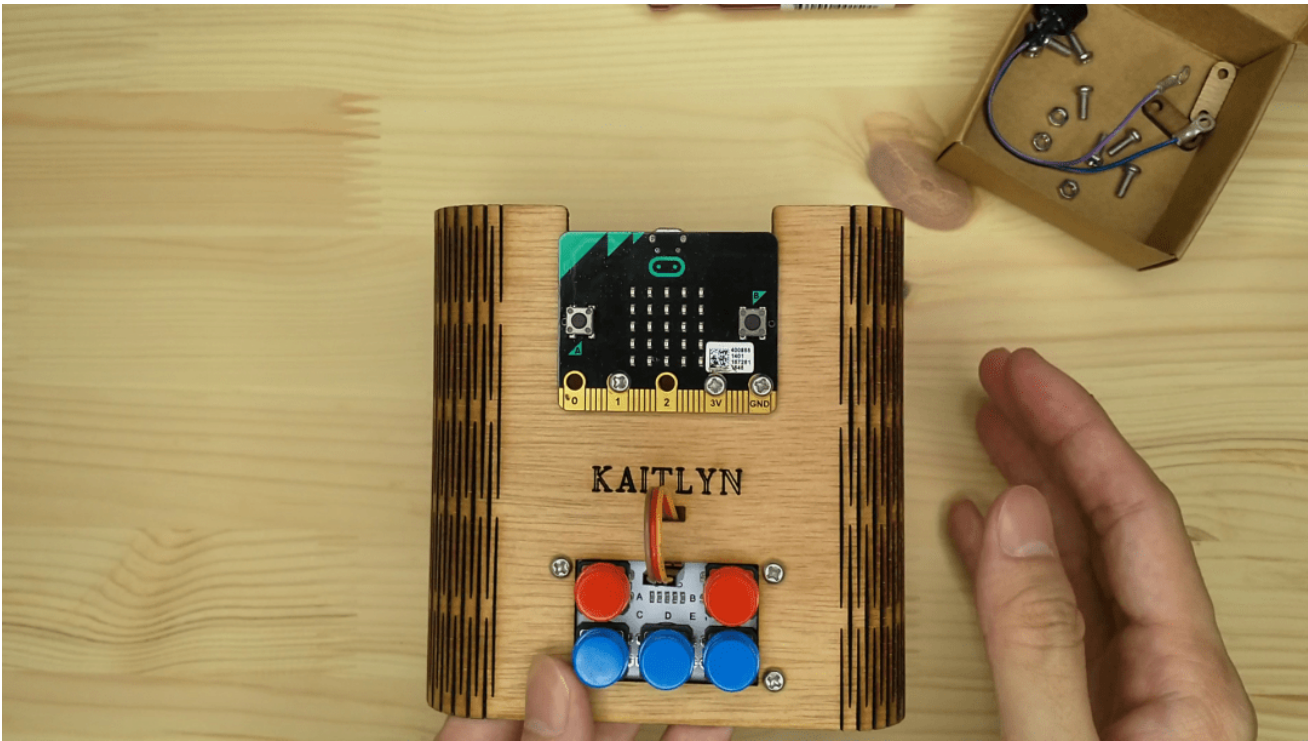
1. Attach the ADKeypad first with the red buttons on top.
2. Screw the 4 corners in and secure them at the back with the golden standoffs

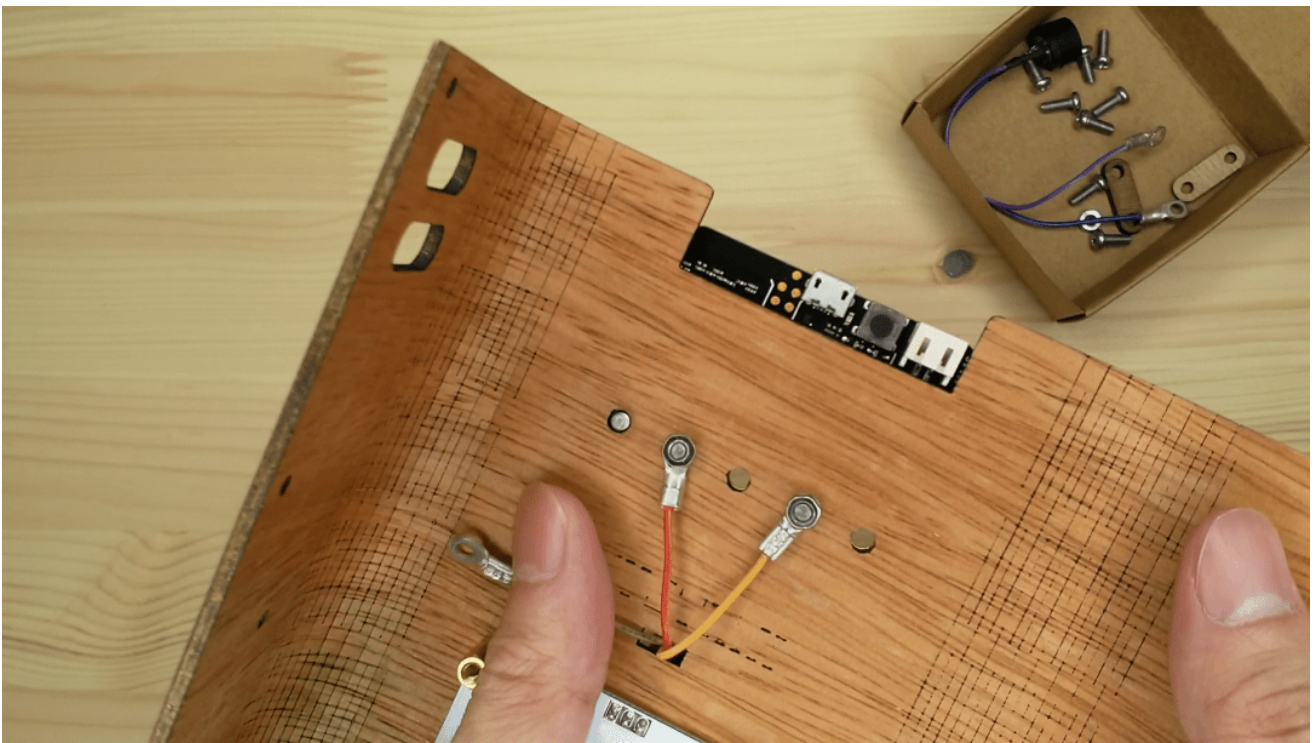
Step 2 – Wire it up!



1. Thread the tri-coloured wire through the hole and attach it to ADKeypad. Brown to G (ground), red to V (voltage) and orange to S (signal).
2. The colours of the jumper wires don't actually affect how the electronics work. But it is good practice to follow a colour convention so that you can easily identify where which cables are attached to.

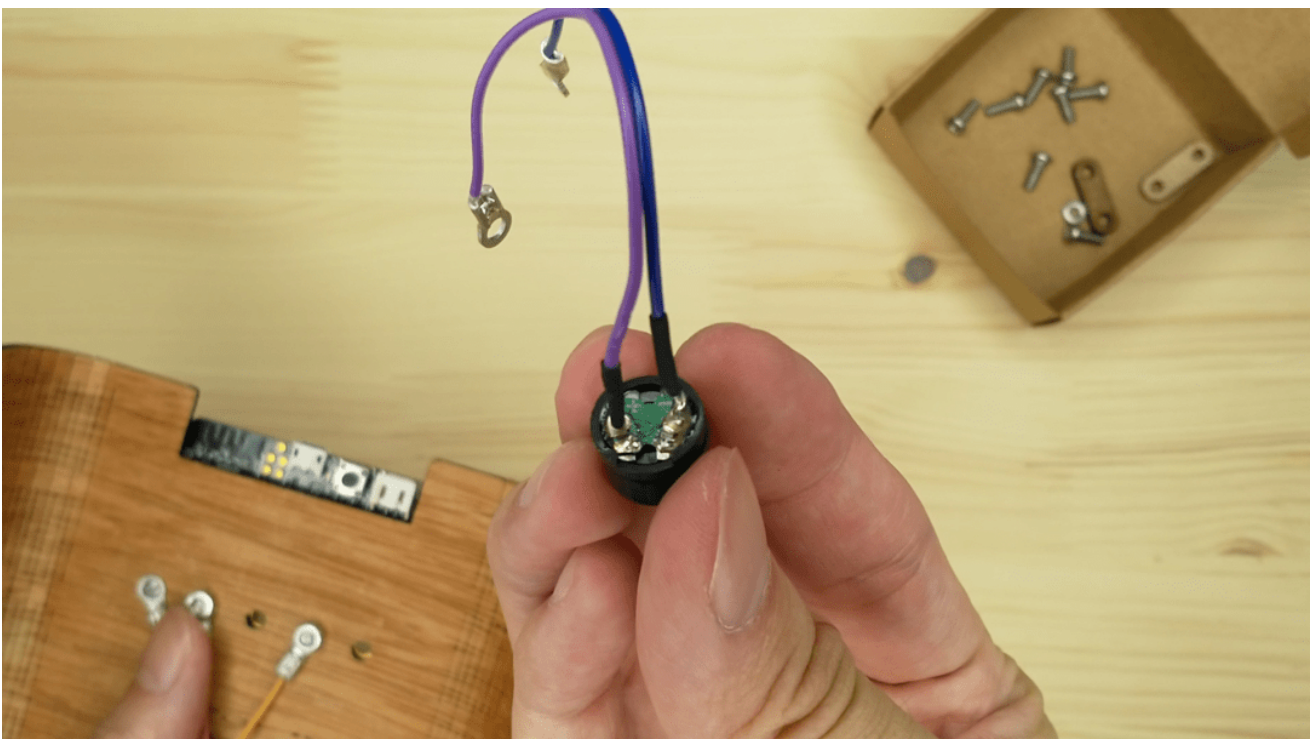
Step 3 – Wiring Firing

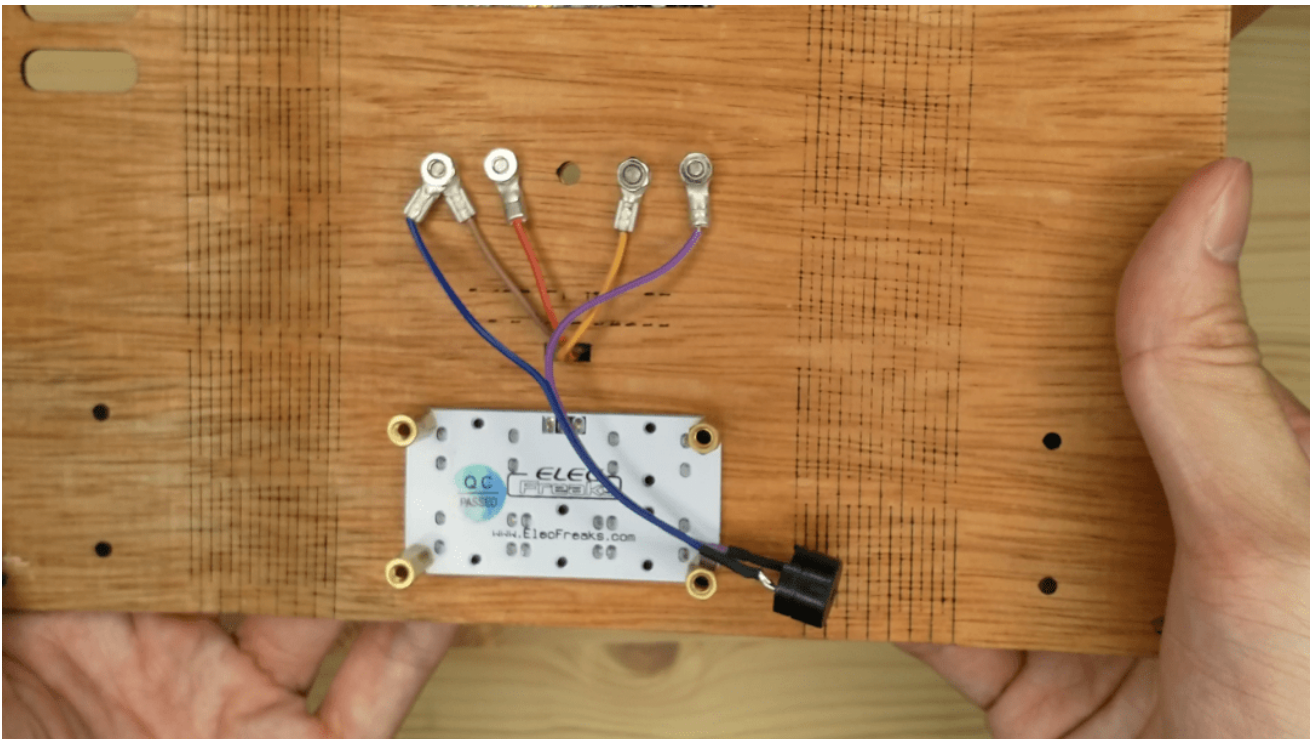
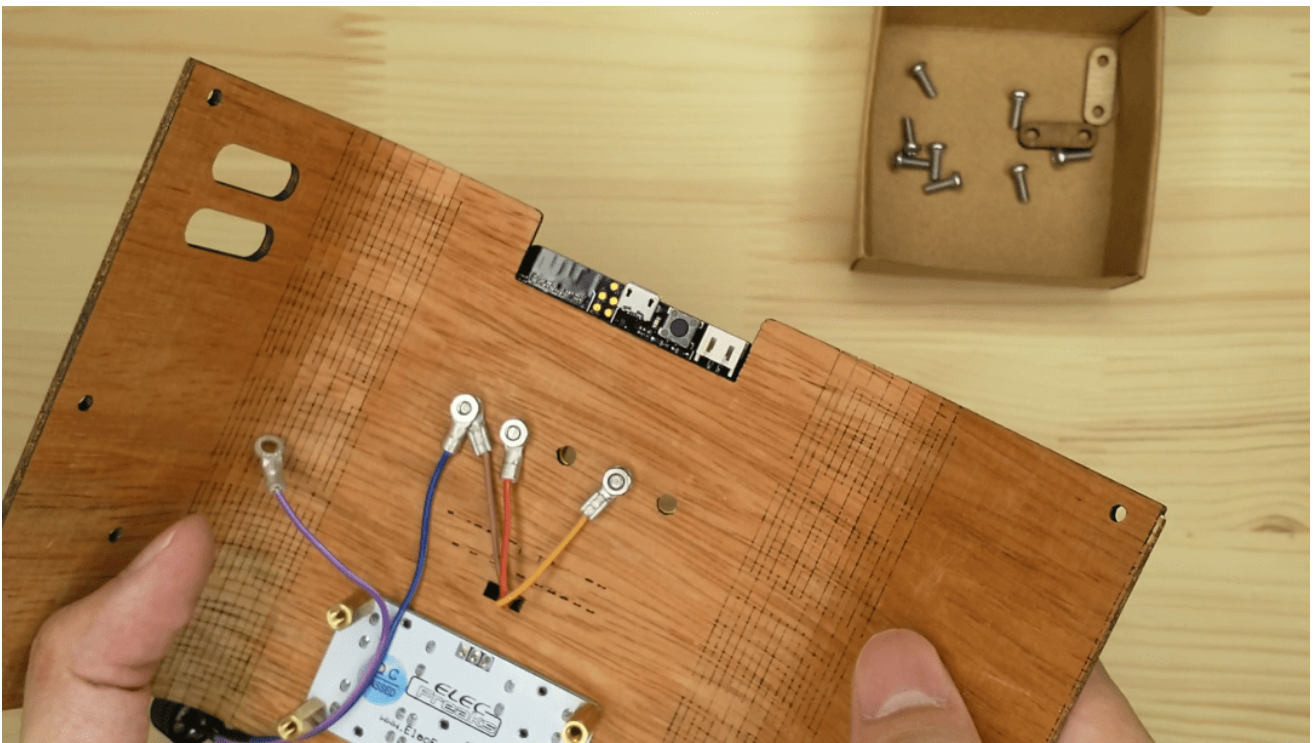




1. Position your micro:bit at the top and on top of your shell.
2. Place a screw into the P1, 3V and GND holes of the micro:bit. We're going to communicate with our ADKeypad through P1 of the micro:bit.
3. On the back, secure the ring terminal of the orange (S) wire from the ADKeypad to the screw on P1 using a nut. Do the same for the red (V) wire with the screw attached to 3V.
4. Position the brown (G) wire to the GND screw but don't attach it yet!

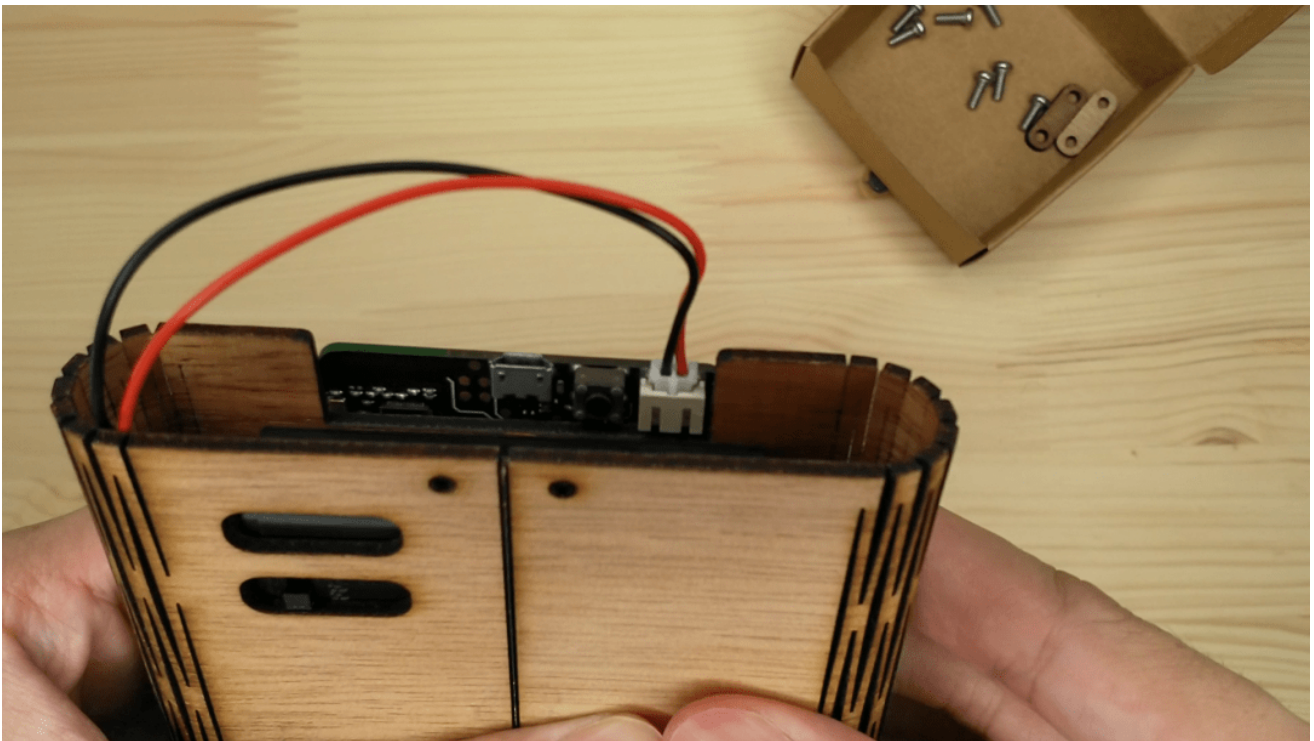
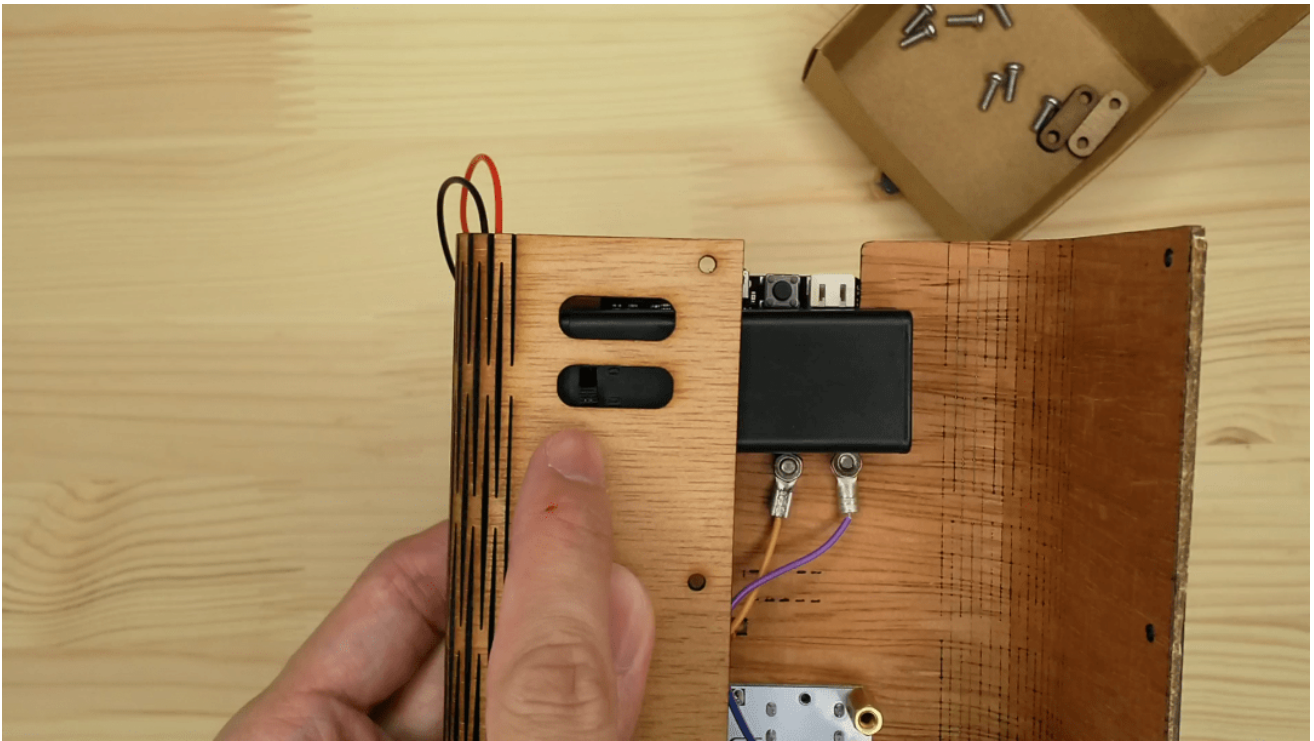
Step 4 – Add a Buzzer





1. The buzzer has both a positive and negative wire! You can find markings on the green bottom of the buzzer. Take note of which colour is positive (+) and which is negative (-). The power supply capabilities and parameters, which better define how you can use the GND and 3V rings.
2. Attach the negative wire to the GND screw above the ring terminal from the ADKeypad. Bolt it in tight!
3. Attach the positive wire to P0 of the micro:bit using the same screw and nut method.
4. Take note that the buzzer will only work with the micro:bit when you attach it to P0! You won't be able to use the makecode Music blocks otherwise.

Step 5 – Battery Powered



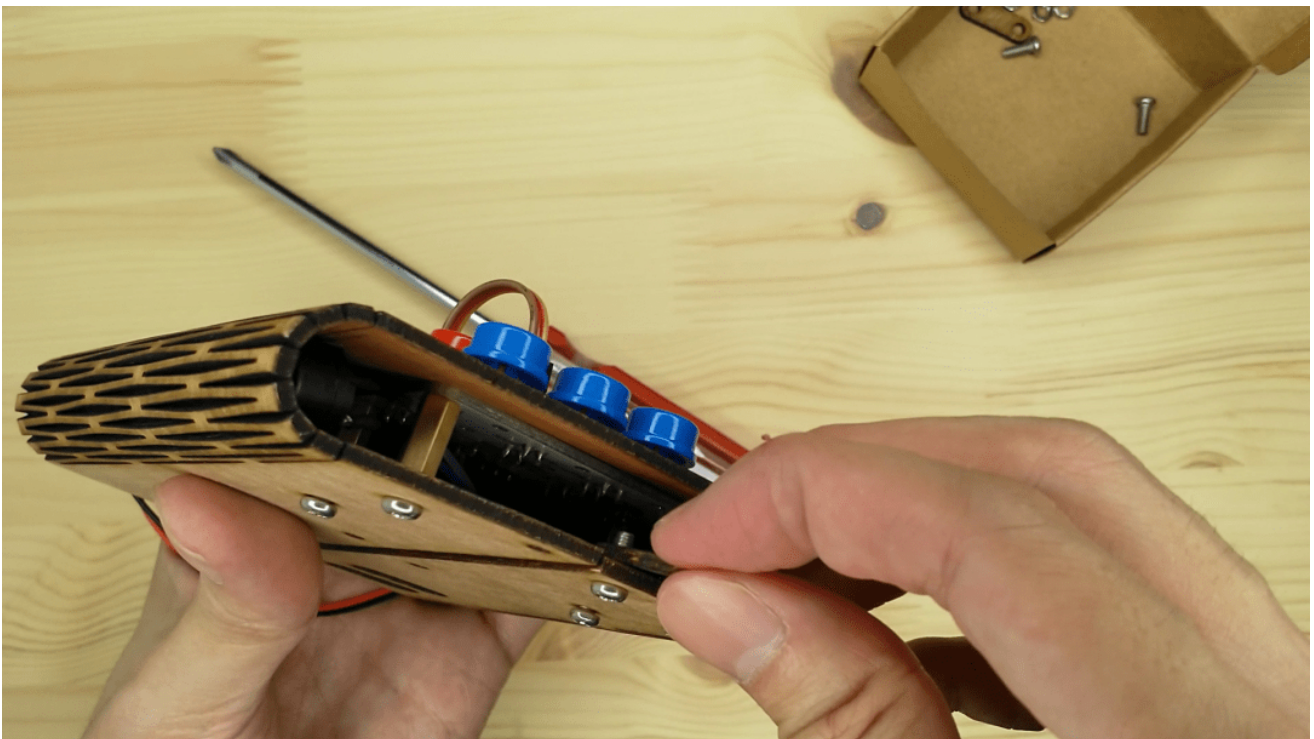
1. Last thing to go into your game:bit will be your battery pack!
2. Add two AAA batteries into your battery pack.
3. Position your battery pack horizontally in the game:bit so that the On-Off switch is accessible from the hole in the back.

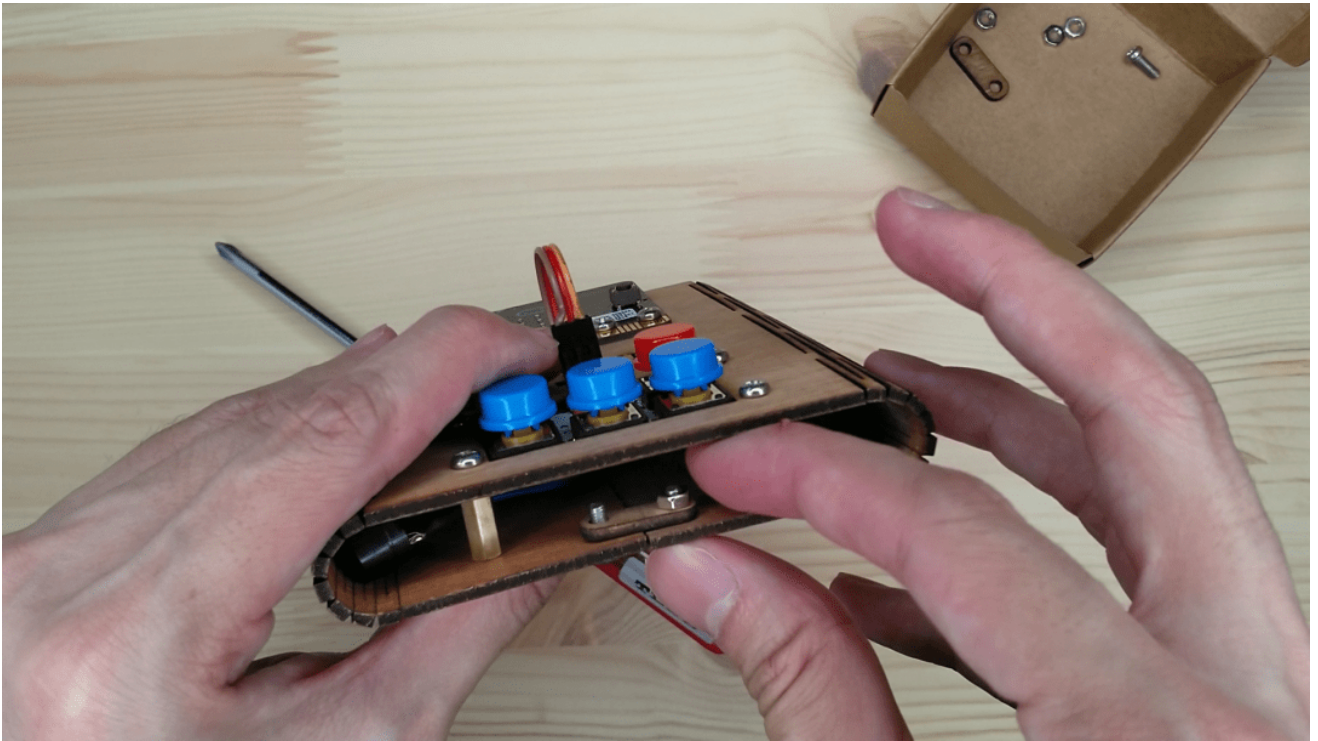
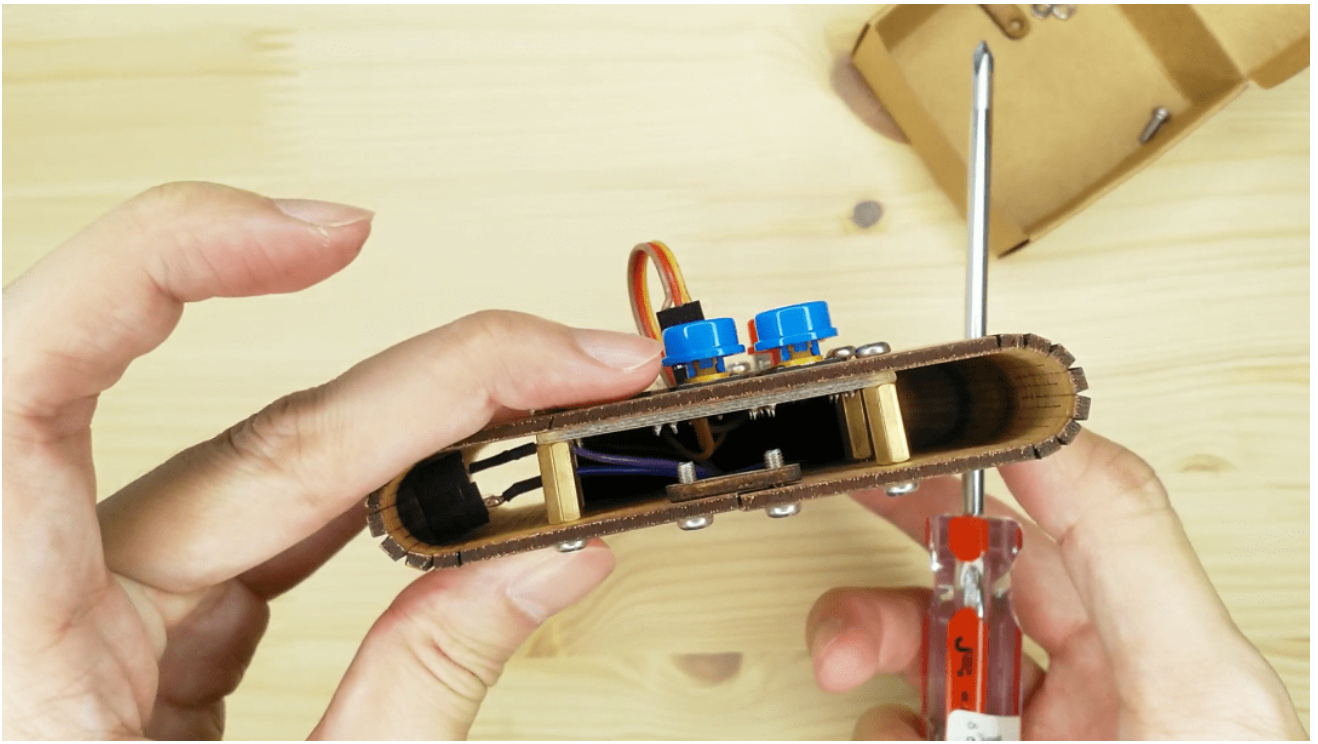
Step 6 – Closing Time

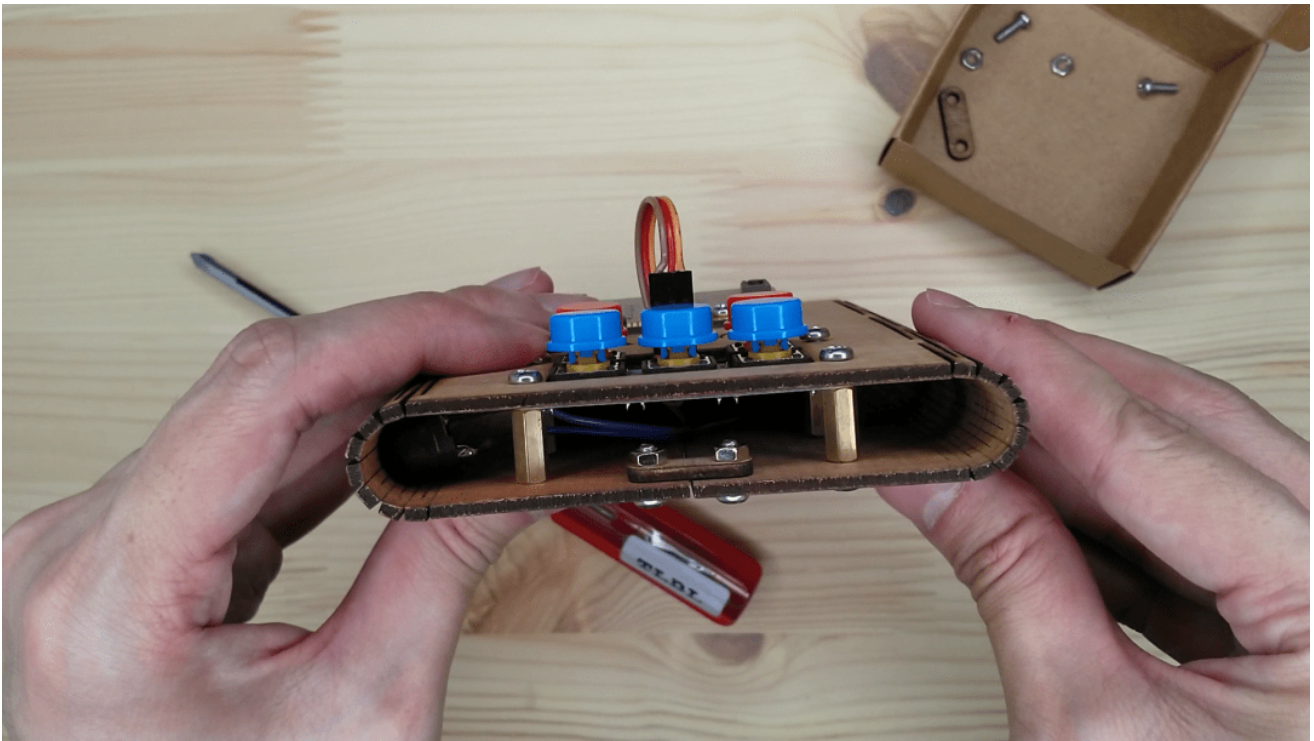


1. Close up the game:bit and align the 4 holes at the back to the standoffs securing the ADKeypad.
2. Screw down into the standoffs to secure the back.

Step 7 – Closing Time







1. Screw two screws into the two holes at the edge of the shell with the lock holder behind. Secure them with nuts.
2. Repeat on the other edge of the shell.
3. The lock holder helps to hold everything together so don't lose it! (Of course this advice is given right at the end of the instructions)

Cool stuff!

Now you've gotten your game:bit fixed together – get your game on and start coding! Follow along with our tutorials and make cool games like Avoid the Asteroids, Maze Runner and Flappy Bird.

27. case 25 μ reMorse

27.1. μ reMorse

- Make a Morse Code “Keyboard”/Editor the hard way.

microbitKit\Tinker_Kit\./images/aSEfIPU.jpg

Goals

- A Morse Code keyboard/Editor made using the C/C++ Micro:bit Runtime
 1. Interprets a combination of short and long button presses into characters using Morse code.
 2. Send characters over the serial interface to your computer, just like a “keyboard”.
 3. Special button combinations for non-visible characters such as spaces and newlines.
 4. Unfortunately, the author use unable to figure out how to send keyboard events, hence a “keyboard” in quotes.
 5. Built using only the Micro:Bit runtime in C/C++
 6. Leverage the built Micro:Bit display provide an interactive typing experience.

Materials & Prerequisites

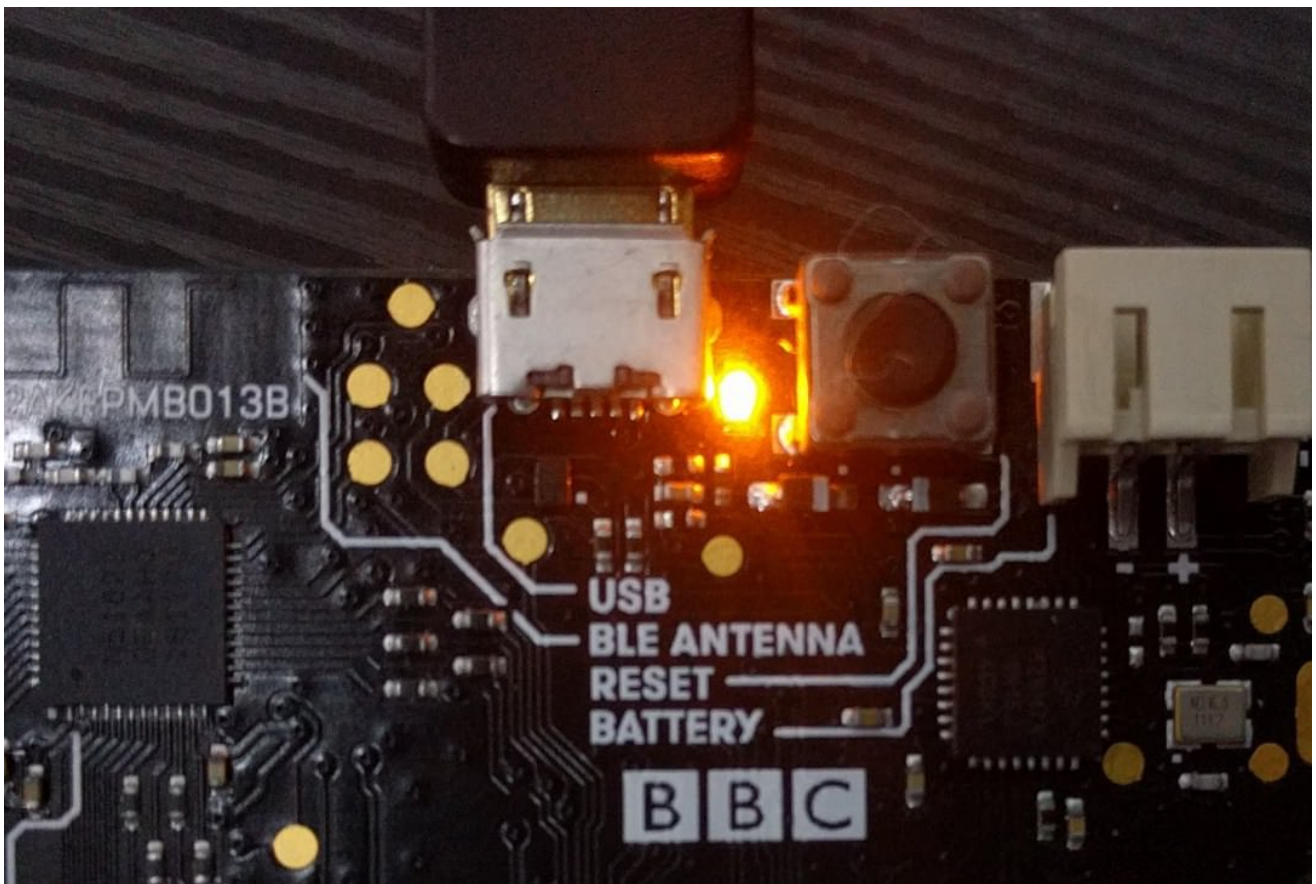
- 1 x BBC micro:bit
- 1 x Micro USB cable
- 1 x Computer with Unix Like OS
- C language experience
- Command Line experience

Step 1 – Development Envrioment

```
→ ureMorse git:(master) brew install srecord
Updating Homebrew...
==> Auto-updated Homebrew!
Updated Homebrew from 987805d3d to f06b54f1b.
Updated 2 taps (homebrew/core, caskroom/cask).
==> Updated Formulae
glib ✓          azure-cli      clojure        entr           gist
tor ✓          bartycrouch   cockroach      fn            goofys
amazon-ecs-cli basex         duck           fonttools     gtk+
artifactory    bitrise      elixir         gauge         hadolint
awscli         buku         emscripten    get-flash-videos imagemagi
==> Deleted Formulae
cloudbees-sdk

==> Downloading https://homebrew.bintray.com/bottles/srecord-1.64.high_sierra.bottle.tar.gz
Already downloaded: /Users/zzy/Library/Caches/Homebrew/srecord-1.64.high_sierra.bottle.tar.gz
==> Pouring srecord-1.64.high_sierra.bottle.tar.gz
🍺 /usr/local/Cellar/srecord/1.64: 203 files, 6.3MB
→ ureMorse git:(master) █
```

microbitKit\Tinker_Kit\.\images\X2ptgqb.png



1. Install the dependencies to build your microbit project. Using your package manager (brew, apt-get, pacman, ...), or any method you fancy, install yotta and srecord. Direct your terminal to the project directory. Here you will write your code in source/main.cpp.
2. To build the program, Micro:Bit runtime program employs the Yotta build system. First we would target the architecture of Micro:Bit by running `yotta target bbc-microbit-classic-gcc`. We can now build the project with `yotta build`. Finally, to install the compiled program into your microbit, plug in your Micro:Bit and find a file ending with `.hex` in the `build//source/` folder. Copy this file into your Micro:Bit, which should now be mounted.
3. Check out `module.json` where you can configure the program's name, version, description, source code folder ... etc.

4. Now that is quite an overwhelming amount of information all at once, fortunately what you need to do would be much simpler. The author has provided a convenience makefile to handle building with yotta and installing the compiled file into the microbit.
5. To build the project, run `make` and to install the compiled program into the microbit, run `make install`. When the micro:bit is receiving instruction, the LED on the back of the microbit near the USB port would flash. Once installation is complete, the LED at the micro-USB port would stop flashing and the program would run automatically.
6. To reset the Micro:Bit at any time and restart the program running the Micro:Bit, press the button next to the micro-USB port on the back of the Micro:Bit
7. For more information, see the runtime documentation on setting up the development environment here.

Step 2 – Hello World

```
1 #include "MicroBit.h"
2
3 MicroBit uBit;
4
5 int main()
6 {
7     uBit.init();
8     uBit.display.print("Hello World!");
9 }
```

1. Let's begin by writing the time-tested "Hello World" program on the microbit. Copy this into `main.cpp`. As usual, but different from the Arduino prototyping platform, code execution starts from the main function.
2. First we would include the `MicroBit.h` to get the Micro:Bit runtime definitions. (The include path would be configured automatically by the build system). The Micro:Bit program is centred around one object/instance, the `MicroBit` object, which you will interact with do almost every action that involves the microbit. In this tutorial, we will name the object "uBit". The first thing any Micro:bit program should do is initialise the Micro:Bit using `uBit.init()`.
3. Now we want the Micro:Bit to scroll "Hello World!" across its screen. This is done using `uBit.display.scroll()` which displays its argument scrolling across the display. Note that the call blocks while the text is displaying scrolls across the screen. Take a look at `uBit.display` for more information on driving the Micro:Bit's built-in display, such as non-blocking calls.
4. That's it, run `make` && `make install` with your Microbit plugged in to install the program. Once the program is installed, "Hello World!" should scroll across the screen.

Step 3 – Buttons & Events

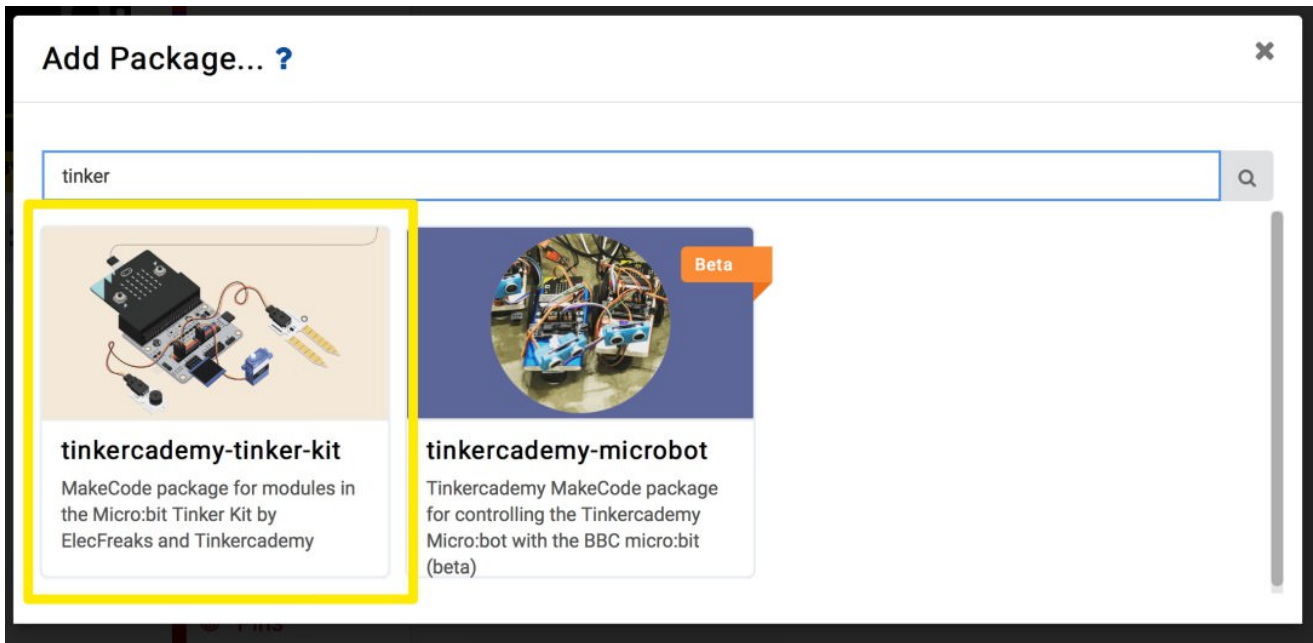
```

1 #include "MicroBit.h"
2
3 MicroBit uBit;
4
5 void handle_button_A(MicroBitEvent e)
6 {
7     uBit.display.scroll("A");
8 }
9
10 void handle_button_B(MicroBitEvent e)
11 {
12     uBit.display.scroll("B");
13 }
14
15 void handle_button_AB(MicroBitEvent e)
16 {
17     uBit.display.scroll("AB");
18 }
19
20 //Core
21 int main()
22 {
23     uBit.init();
24
25     uBit.messageBus.listen(MICROBIT_ID_BUTTON_A, MICROBIT_EVT_ANY, handle_button_A);
26     uBit.messageBus.listen(MICROBIT_ID_BUTTON_B, MICROBIT_EVT_ANY, handle_button_B);
27     uBit.messageBus.listen(MICROBIT_ID_BUTTON_AB, MICROBIT_EVT_ANY, handle_button_AB);
28
29
30     release_fibre();
31 }

```

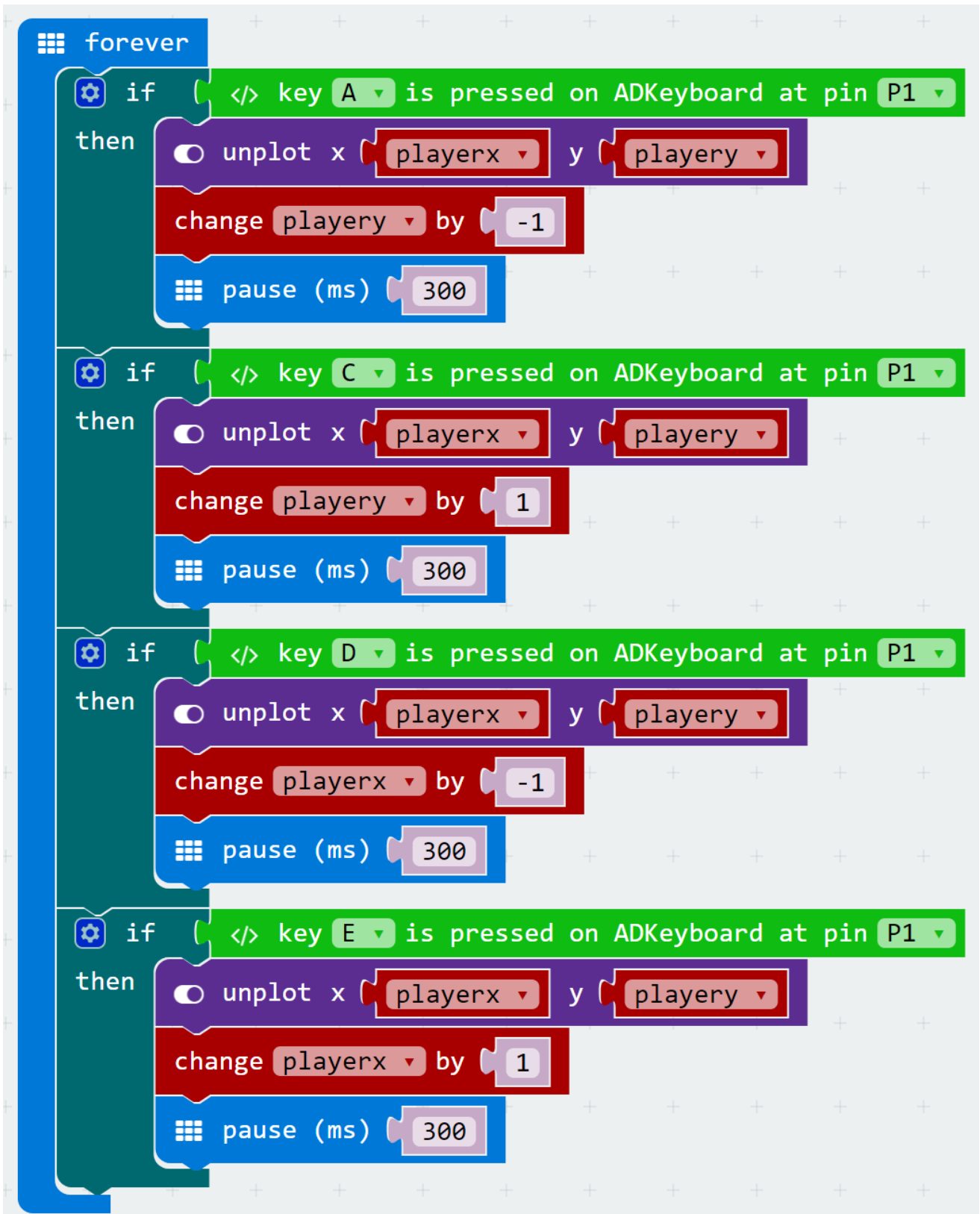
1. We will employ the two buttons on the Micro:Bit, namely button A & B in some combination of clicks & holds to trigger a specific functionality.
2. When something happens to the buttons on the Micro:Bit, this happening is translated into an 'event' into the system. Events are not simply limited to button presses. They could represent anything from a the radio receive a datagram or the accelerometer detecting a change in orientation.
3. To run code whenever a specific event is raised, we write an 'event handler', which is simply a function that contains the code that you want to run. The function takes in an MicroBitEvent argument, which is the event that caused the function to be run and returns nothing.
4. Micro:Bit uses messageBus as it to deliver events run registered event handlers when a certain event happens, such when one of the buttons on the microbit is pressed, which messageBus would call the event handler, provided that the event meets the handlers listening criteria.
5. Use uBit.messageBus.listen() to register your function as an event handler for a specific event criteria. Here we are registering event handlers to the buttons on micro:bit, for any event. This means that any event that is related to the specified button would call the event handler that was registered for that button. In an actual program, we can be more specific like specifying MICROBIT_EVT_BUTTON_HOLD to run the event handler only if the specific button is held down for some time.
6. Note that ALL execution would cease if the Micro:Bit reaches the end of main() function, hence, we release the main thread or this case the main "fibre" to allow the micro:bit to process button events.
7. That's it, run it with make && make install. Once the program is installed, pressing button(s) A and/or B should scroll "A" or "B" depending on which button you press. If you press A and B together, you should see "AB" scrolling across the screen.

Step 4 – Registering Morse Code



1. Morse code is made up of a variable combination short and long signals, or in this case, button presses.
2. Expand the 'Advanced' section and scroll to the bottom and click on 'Add Packages'
3. In the search box, type in 'Tinker'. Click on the box labelled 'tinkercademy-tinker-kit'
4. Now you'll see something new in MakeCode – a bright green Tinkercademy category has been added!
5. Inside this category you'll find blocks to sense button presses on the ADKeypad. Note that importing this package only happens for the current project. So if you start a new project and want to use the category, you'll need to re-import it.

Step 5

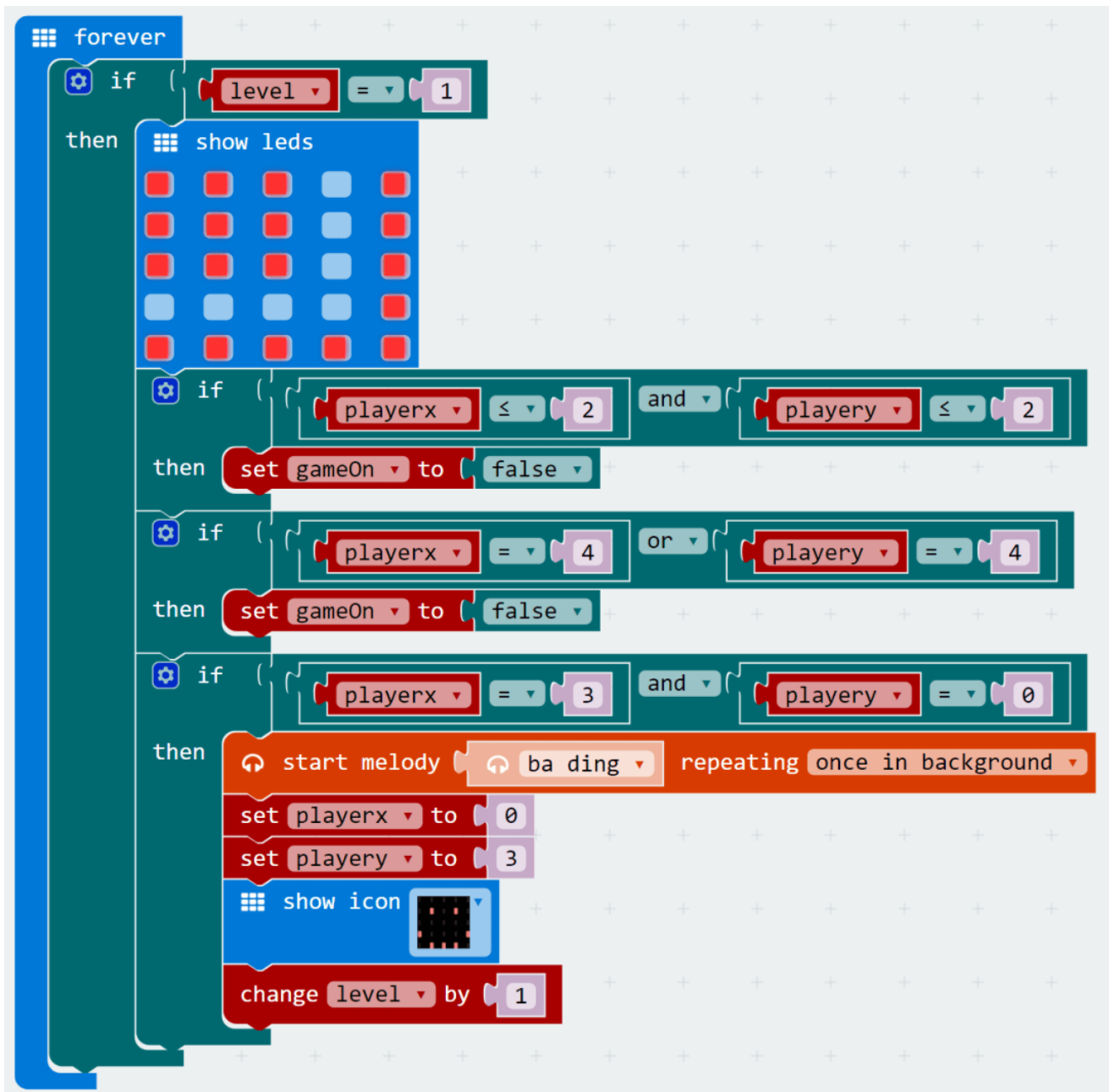


1. Now that we have our Tinkercademy category added, we can use the ADKeypad to move up, down, left, and right. In this example, we'll set the A button to move up, the C button to move down, the D button to move left, and the E button to move right.
2. To do this, we use if statements. If statements test to see if a condition is true; if it is true, then they run any blocks inside the if block. When we place an if statement inside a forever loop, we forever test to see if the condition is true.
3. To move the player, we simply change the player x or player y variables. Remember, decreasing or increasing playerx causes the player to move left or right respectively, and decreasing or increasing playery causes the player to move up or down respectively.

We're constantly plotting the location of the player using these variables, so when we change them, it automatically changes the player's location!

- Note that we add a short 300ms pause after each button press. Otherwise the micro:bit would move you many spaces every time you pressed a button because it runs the code so fast.

Step 6



- Now that we can move our player around, let's start creating our maze levels! Every time we start a level we need to do a few things: First we need to display the maze walls on the LED display. Second, we need to forever check if the player runs into a wall (if they do, it's gameover!). And third, we need to forever check if the player makes it to the end of the maze level (if they do, let them know they succeeded and move onto the next level!).

2. For each level, we're going to use a forever loop. Inside the loop, we use an 'if' statement to check if the level variable equals 1. This means this code will only ever run if the level variable equals 1.
3. Inside the if statement, we first display the maze walls. We light up LEDs to serve as maze walls, and leave them turned off to represent the maze path. This can be done using the 'show leds' block. One thing to be careful about though: remember above we set the starting position of the player? Make sure your player starting position is not inside a maze wall! In this example, the starting position of the player is $x=0, y=3$.
4. Next, we need to check if the player ever runs into a wall. How to do this? Once again we'll use if statements to check if our `playerx` and `playery` variables are ever in the same place as a wall. We do this using the coordinate system of the 5x5 LED grid. In this example, there are two sections of walls.
5. The first wall exists where `playerx` is less than or equal to 2 AND `playery` is less than or equal to 2. We create an if statement with these conditions, and inside we set `gameOn` to 'false' (since if it's ever 'true', it means the player ran into a wall and should get a Game Over).
6. The second wall exists where `playerx` equals 4 OR `playery` equals 4. We create another if statement with these conditions, and inside we set `gameOn` to 'false' (because once again if it's ever True, it means the player ran into a wall and should get a Game Over).
7. Finally, the last test we need to add is to see if the player makes it successfully through the maze! In this example level, the end of the maze is at $x=3, y=0$. We create another if statement to check if $x=3$ AND $y=0$, and inside we do a few things: First, we play a success melody in the background. Second, we set the starting position of the player for the next level (in this example, we use the same starting position, but it can be different!). Third, we show a smily face to tell the player they succeeded! And fourth, we change the level variable by 1 (this will cause the next level to display).

Step 7

```
forever
  if (gameOn = false)
  then
    start melody (wawawawaa) repeating (once)
    set level to 0
    unplot x (playerx) y (playery)
    show icon (angry face)
    show string ("Press B to restart")
```

1. Whew, setting up a level was a lot of work! Now that we have a single level, let's make something happen when a player gets a game over. This will happen whenever they run into a wall, and it's tracked by the 'gameOn' variable.
2. Inside a forever loop, we use an if statement to check the value of the 'gameOn' variable. If it equals 'false', then we want our game over code to run!
3. In this example, we play a sad melody in the background, reset the 'level', unplot the player LED, show an angry face, and finally display a string telling the player they can press B to restart the game.

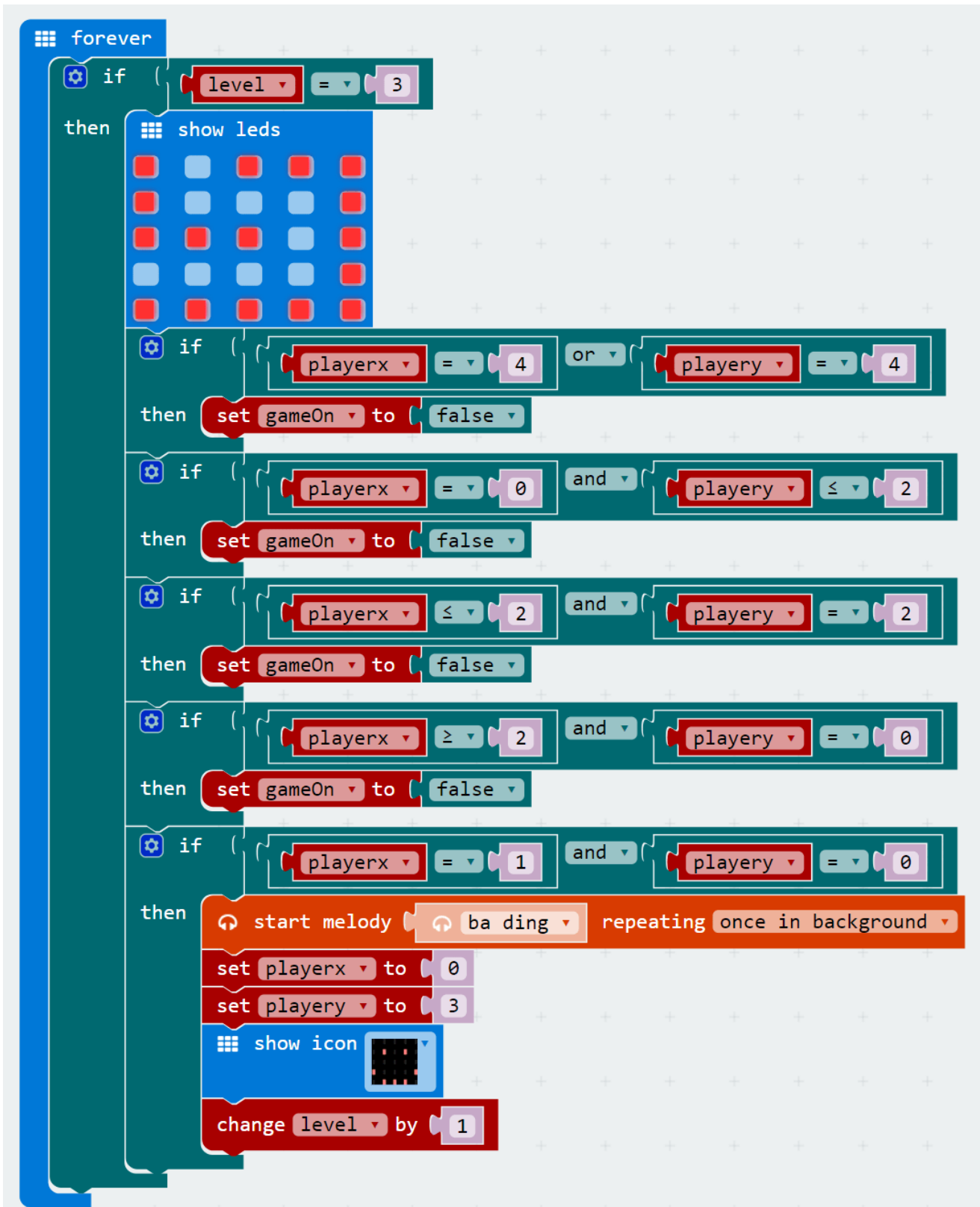
Step 8

```
forever
  if (<> key B is pressed on ADKeyboard at pin P1)
  then
    set level to 1
    set playerx to 0
    set playery to 3
    set gameOn to true
```

1. Speaking of pressing B to restart the game, we haven't yet created the code to do that!
2. Inside a forever loop, we test to see if button B on the ADKeypad is pressed. If it is, we want to set 'level' to 1, reset the player's starting location by setting the 'playerx' and 'playery' variables to 0 and 3 respectively, and set the 'gameOn' variable back to 'true'.

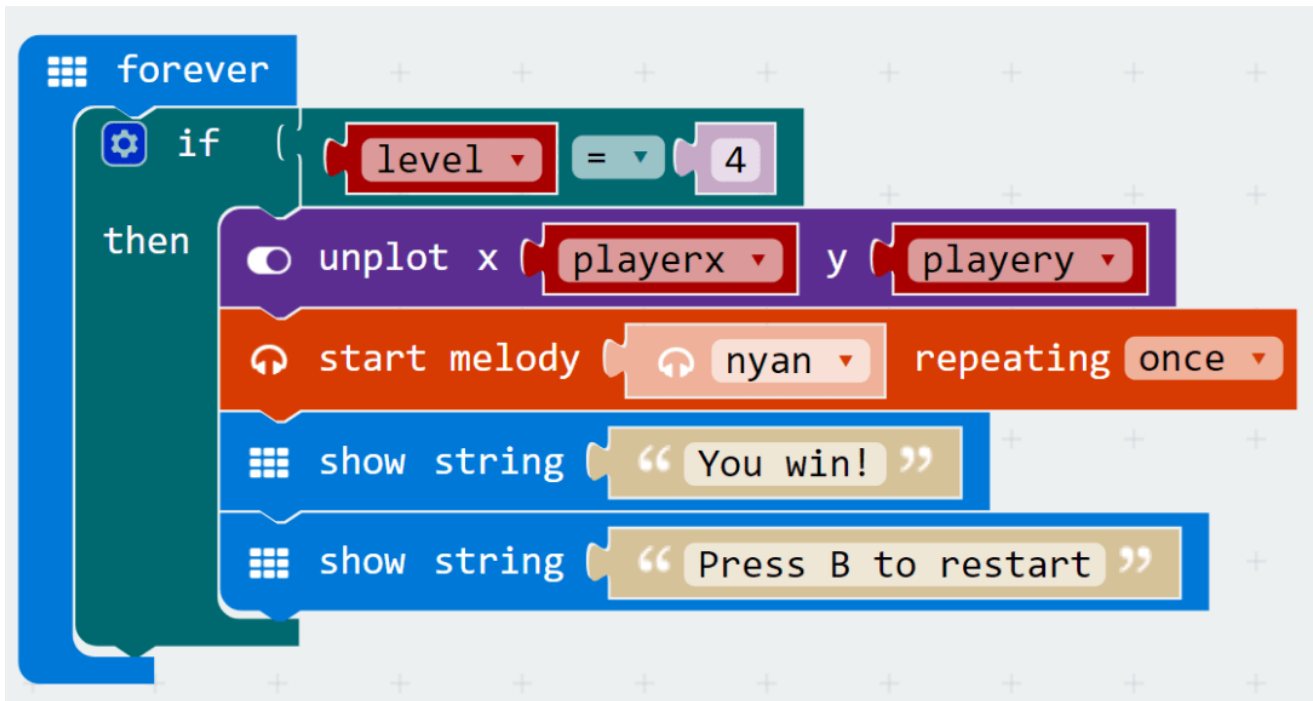
Step 9

```
forever
  if (level = 2)
  then
    show leds
    if (playery = 0 or playery = 4)
    then set gameOn to false
    if (playerx = 0 and playery ≤ 2)
    then set gameOn to false
    if (playerx = 4 and playery ≤ 2)
    then set gameOn to false
    if (playerx = 2 and playery ≥ 2)
    then set gameOn to false
    if (playerx = 4 and playery = 3)
    then
      start melody ba ding repeating once in background
      show icon
      set playerx to 0
      set playery to 3
      change level by 1
```



1. Now our game should be working as intended! The only thing missing is more levels!
2. It's quite easy to add more levels by duplicating our level 1 code from above. The only things that will change are the maze walls and the coordinates for our if statements (for testing if the player moves into a wall or completes the level).
3. Tip: sometimes it can be complicated to create if statements to test for every wall. In these cases, try to break down your walls into separate rectangles and create an if statement for each rectangle.
4. One thing to watch out for: after the player completes the level and you reset their playerx and playery variables, make sure the position matches your next level. Otherwise they could start inside a wall!

Step 10



1. Once you're done adding in levels, you can optionally create a victory section. In this example, once the player successfully completes the first 3 levels and level equals 4, we: unplot the player, play a victory melody in the background, and show a victory message!

Cool stuff!

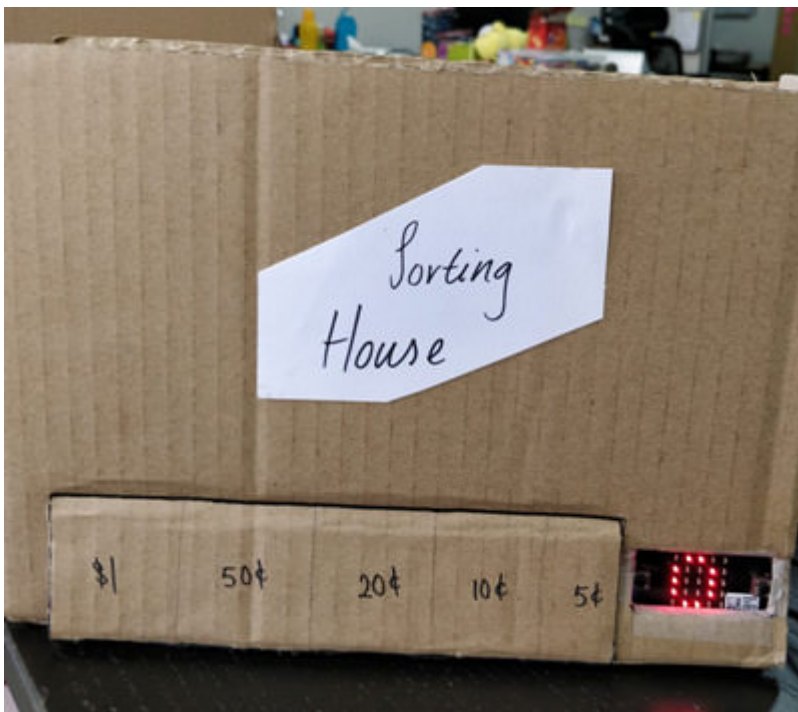
Now that you've learned how to use the ADKeypad, you can try using it to control LEDs, servos, and other components! You also learned about if statements which are useful in many micro:bit projects! Try customising your maze runner game by adding more levels!

28. case 26 Coin Sorter with micro:bit

28.1. Coin Sorter with micro:bit

- Ever just accumulate a bunch of coins in a jar and now want to sort out the giant mess your past self should have foreseen? No? Just me? Ok well let's build a coin sorter for fun then, adding on an ultrasound to count the amount you have sorted. Written by Hannah from Raffles Institution during a job attachment.

Goals [🔗](#)



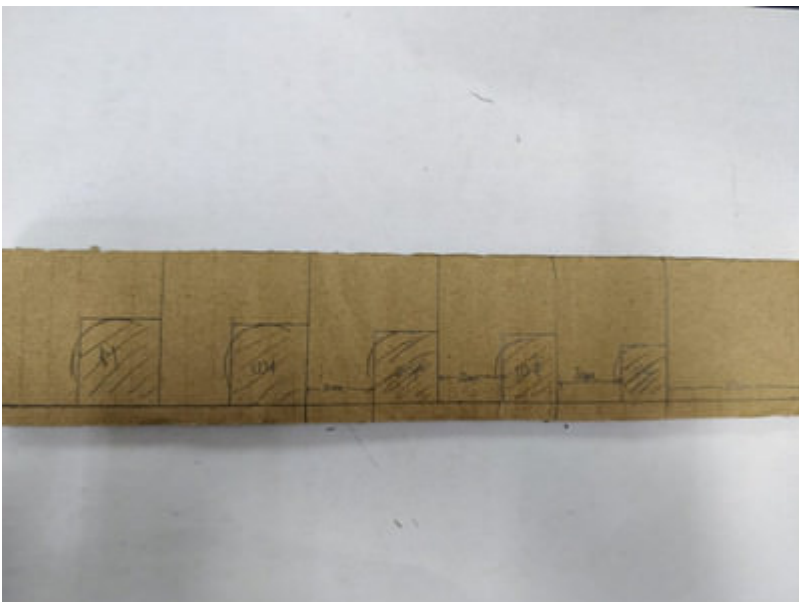
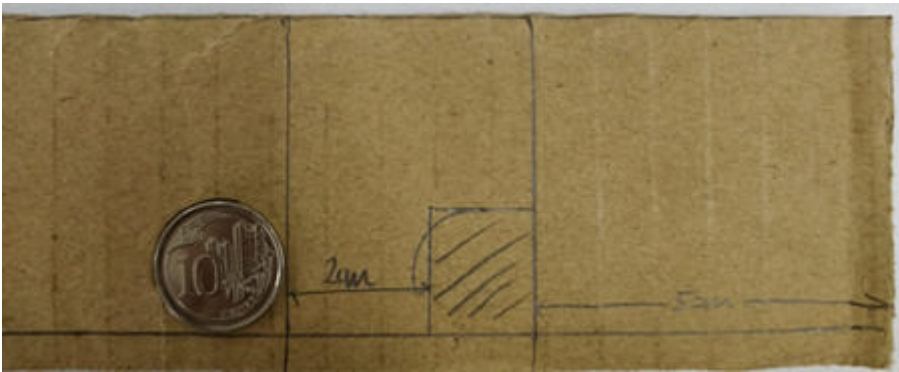
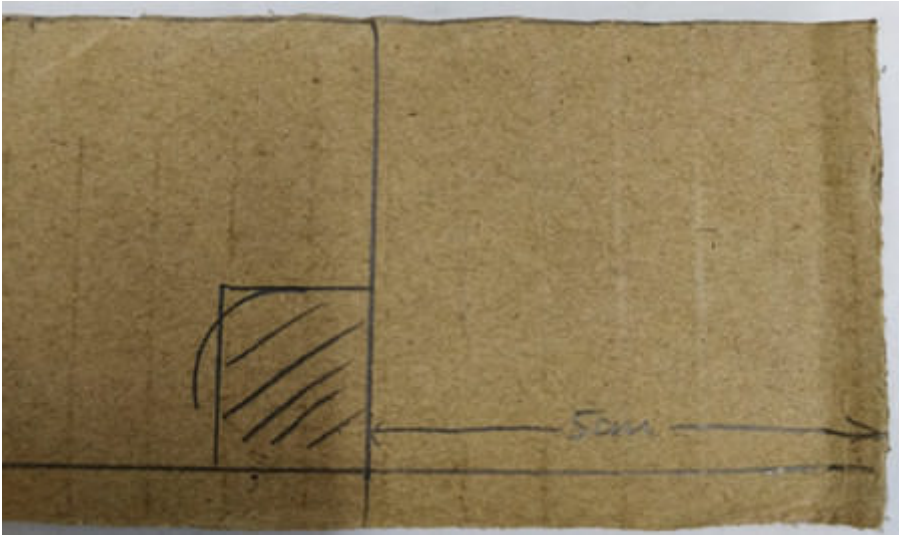


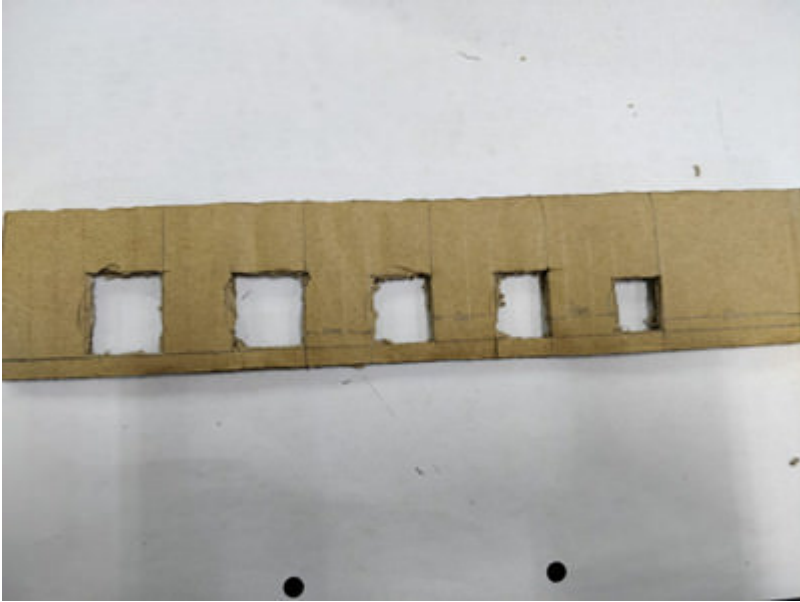
1. Make a simple mechanical coin sorter.
2. Learn how to wire up and use an ultrasound HC SR04.
3. Have fun!

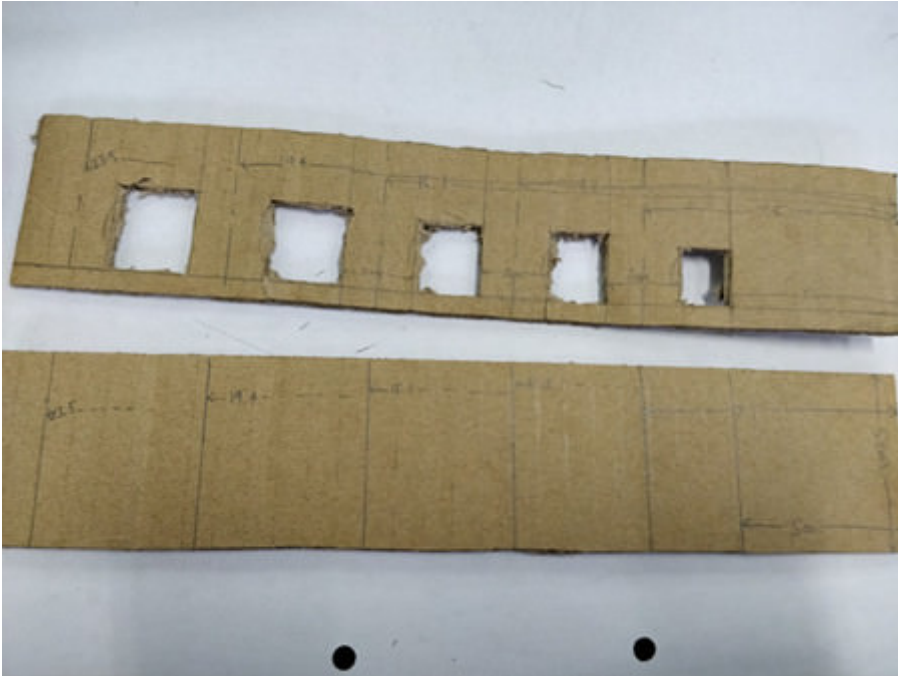
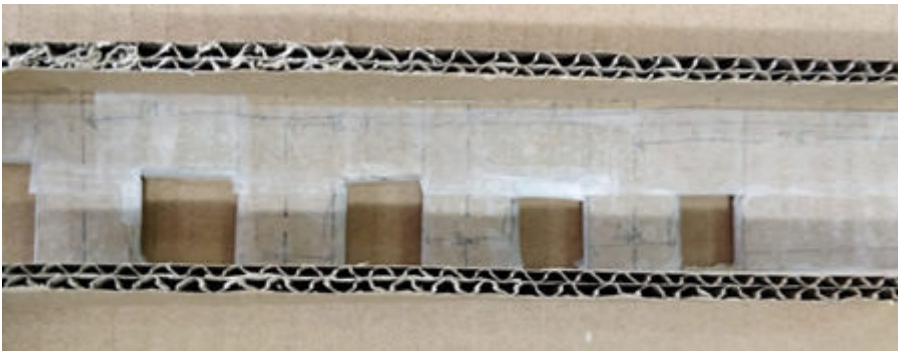
Materials

- 3 5x25cm pieces of cardboard
- 2 25x2.5cm pieces of cardboard
- 2 19x26cm pieces of cardboard
- 2 5x19cm pieces of cardboard
- 6 5x5cm pieces of cardboard
- 1 2x5cm piece of cardboard
- Jumper wires and 3 crocodile clip heads
- Solder
- 5V battery supply or 3AA batteries and a battery holder
- 1 micro:bit
- 1 micro USB cable
- 1 Ultrasound HC SR04
- Superglue

Hardware Step 1 – Creating the slots



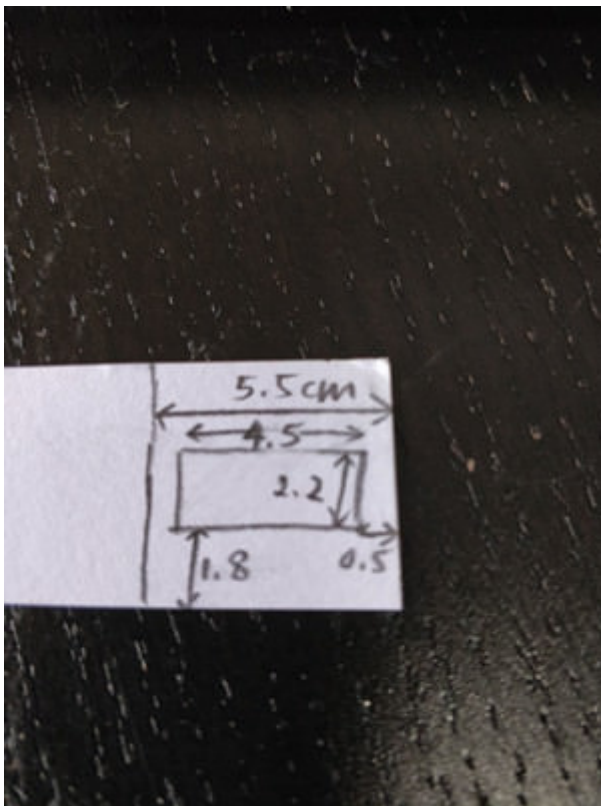




1. Take a 5x25cm piece of cardboard (A) and draw a line 0.5cm away from the long edge of the cardboard.
2. Leave a gap of 5cm from the short edge to the cardboard and draw a reference line.
3. Place a 5cm coin at the corner of these two lines and trace its edges.
4. Draw a rectangle of width which is the diameter of the coin and length 0.2cm shorter than the diameter of the coin. Shade this rectangle to indicate that it is to be cut out later.

5. Leave a 2cm gap from this rectangle and draw another reference line
6. Repeat steps 3 to 5 using a 10 cents, 20 cents, 50 cents and \$1 coins in this order.
7. Cut out the shaded rectangles using a pen knife.
8. Since the cutting of the board left some parts flatter than others, flatten the whole piece of cardboard.
9. Since the cardboard is pretty rough, there might be too much friction for the coins to slide down. Rectified this by taping the cardboard in smooth scotch tape (p.s. The transparent scotch tape might also have too much friction)
10. Measure the distance from the middle of each of the 2cm gap to the end of the cardboard. Taking these measurements, draw reference lines on the other piece of 5x25cm cardboard (B).
11. Paste the two pieces of 2.5x25cm pieces of cardboard on to the edges of A.

Step 2 – Creating the front piece





1. Leave a 5.5cm gap from the edge of the 19x26cm piece of cardboard (C) and draw a reference line.
2. Draw a rectangle at the corner of the C (within the area enclosed by the 5.5cm gap) as shown.
3. Cut out this rectangle.
4. Draw a 5x18cm rectangle from this reference line as shown. This will be the opening to the sorted coins.
5. Cut out the widths of the rectangle as shown to make a door.
6. Using a ruler, press in the side of the door to make it easier to bend outwards

Step 3 – Creating the side piece



1. Take a piece of 5x19cm piece of cardboard (D).
2. Create a rectangular coin slot of 0.3x3cm dimensions whose lower length is 16cm from the bottom of the cardboard. (see picture for better visualisation)
3. Create a hole for the wires by cutting out a 1x2cm rectangle whose lower length is 10cm from the bottom of the cardboard. (see picture for better visualisation)

Step 4 – Setting up the electronics





1. Solder one male header jumper wire to a crocodile clip as shown. Insulate the exposed wire using either electrical tape or shrink tubing.
2. Repeat step 1 2 more times to create 2 signal connection wires and 1 ground wire.
3. Connect the 2 signal connection wires to the trig and echo pins of the ultrasound and pins 0 and 1 respectively on the microbit.
4. Connect the the ground pin of the ultrasound and the ground of the microbit using the ground wire.
5. Paste the ultrasound to the 2x5cm piece of cardboard such that the the large flat back of the ultrasound is flushed against the board and the pins are sticking out.

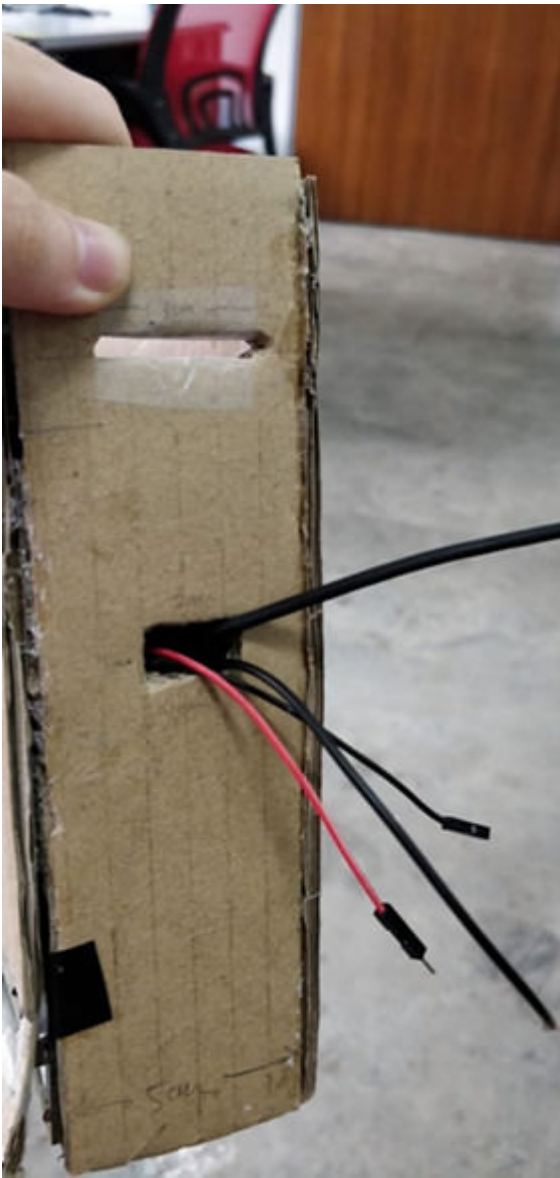
6. Connect the micro USB of the microbit.

7. Connect a jumper wire separately to the 5V and ground pin of the ultrasound.

Step 5 – Putting it altogether







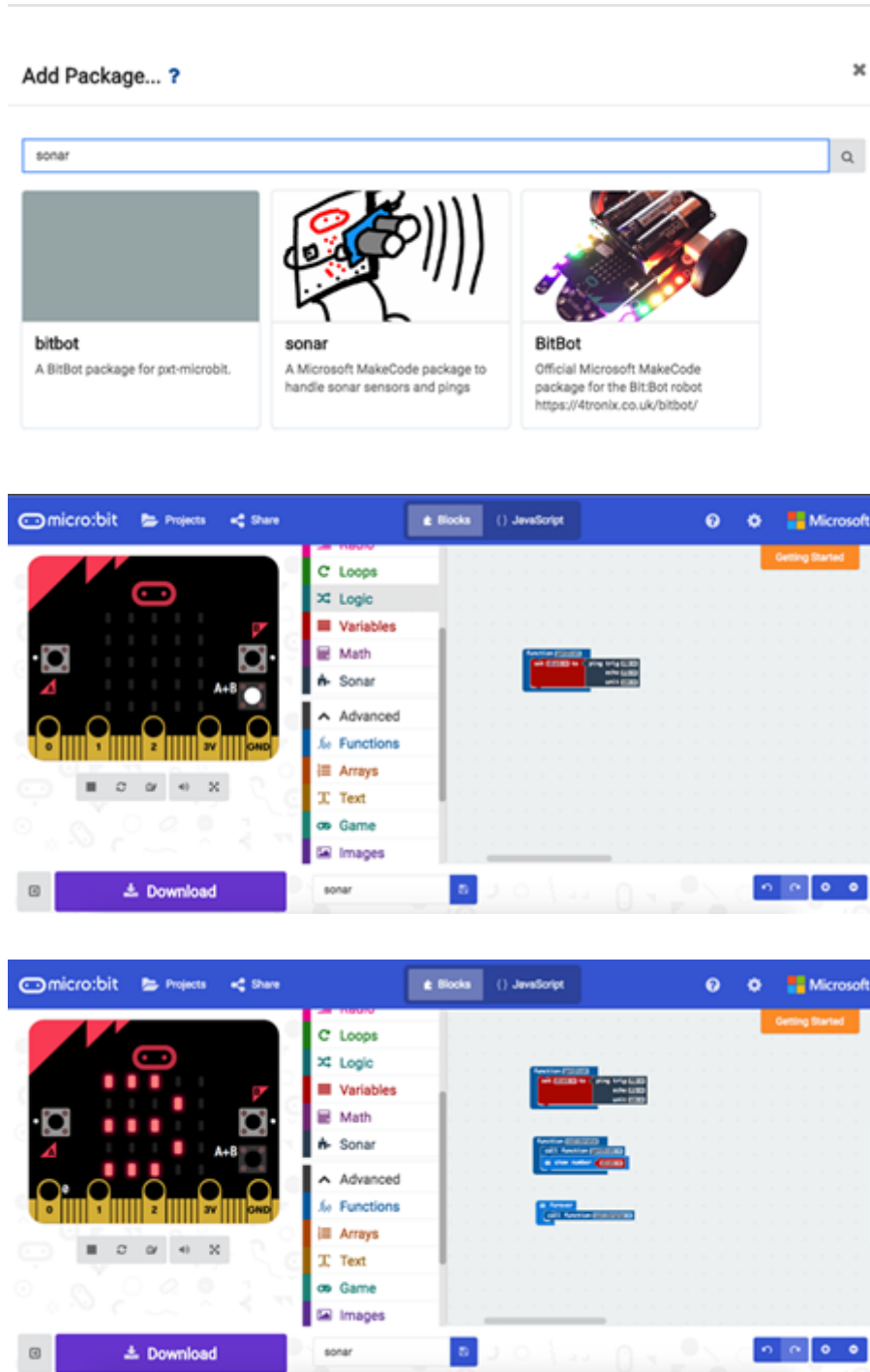
1. Paste the 6 5x5cm pieces of cardboard on B on its reference lines.
2. Paste D on the right side of B
3. Paste the ultrasound about 5.7cm from the bottom of the D
4. Using tape, tape the microbit 0.5cm from the bottom of B, with its screen and buttons facing outwards
5. Thread all the wires through the hole for the wires.
6. Paste the other 5x19cm board (E) at the back of B
7. Orientate A such that the 5 cent slot is on the right. Paste the back of A on B such that the tip of A near the \$1 slot is 9cm from the bottom of B and the 5 cent slot side is 16cm from the bottom of B. (see picture for better visualisation)
8. Paste C to the front of the coin sorter such that A is 8cm and 15cm from the bottom of C.
9. Paste the other 5x19cm piece to the left side of the coin sorter and the last 5x25cm board onto the top of the sorter to close the setup.

Step 6 – Powering the microbit and ultrasound

1. Attach the ground and 5V wires of the ultrasound to a 4.5V to 5V power supply
2. Power the microbit in parallel to the ultrasound using the same power supply.

3. If a different power supply is used, attach the ground wire of the microbit in parallel to the ground from the same 5V power supply. (Sharing of the ground wire)

Software Step 1 – Calibrating the ultrasound



- Every ultrasound is different and the environmental conditions at your place might be different from mine so the values I use might be different from yours. Calibration helps you find what values you should be using in your situation.

1. Download the Sonar package for easy access to the ultrasound function.
2. Create a function to get the distance of the ultrasound as shown.
3. Create a function to show the distance that the ultrasound is sensing on the screen

4. Continuously call this function
5. Drop different coins into the coin sorting machine and record the minimum and maximum distances you get from each type of coin.

Do make sure that the ultrasound is placed neatly vertically and is not obstructed by anything.

Step 2 – Sum of the value of the coins



1. Record the min and max values from above as variables.
2. Initialise the sum of the coins to be 0.
3. Logically, if the ultrasound gets a reading between the min and max of a coin, the coin that has been sorted must be of that particular value. Hence, if the reading is larger than or equal to the min value and smaller than or equal to the max value, it is for example a 5 cent coin. Store the value of the current coin being sorted in a variable.
4. If the coin has been found, we need to increase the total sum value by its value. Create a function that checks for and does this.
5. Continuously call this function.
6. Now we need a way to restart the sum value if we take out our coins, so go ahead and reset the sum to 0 if the buttons A&B are pressed.

Good job!!!

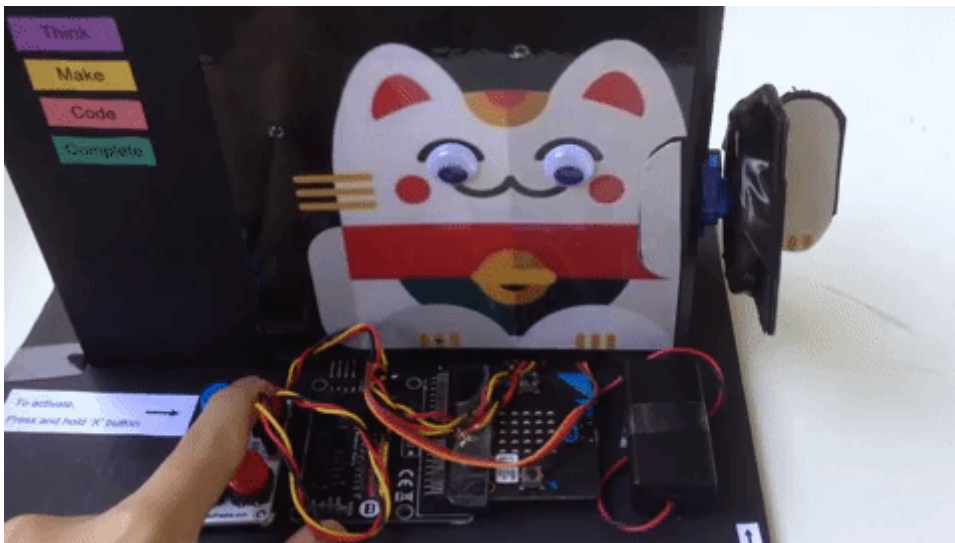
Hopefully, you had fun building the coin sorting machine. Now, think of ways to further develop the project... maybe add a function to see if the machine is full? Either way, enjoy ^
^

29. case 27 Make a Waving Fortune Cat

29.1. Make a Waving Fortune Cat

- Use a micro:bit and mini servo to make your very own Maneki-Neko, or Fortune Cat, who waves its hand when you press a button! Designed and written by Tim Ho from the National University of Singapore.

Goals



1. Make a moving cardboard cat.
2. Give cat an action which you desire.
3. Hint: Follow the steps and pictures during your building process!

Materials

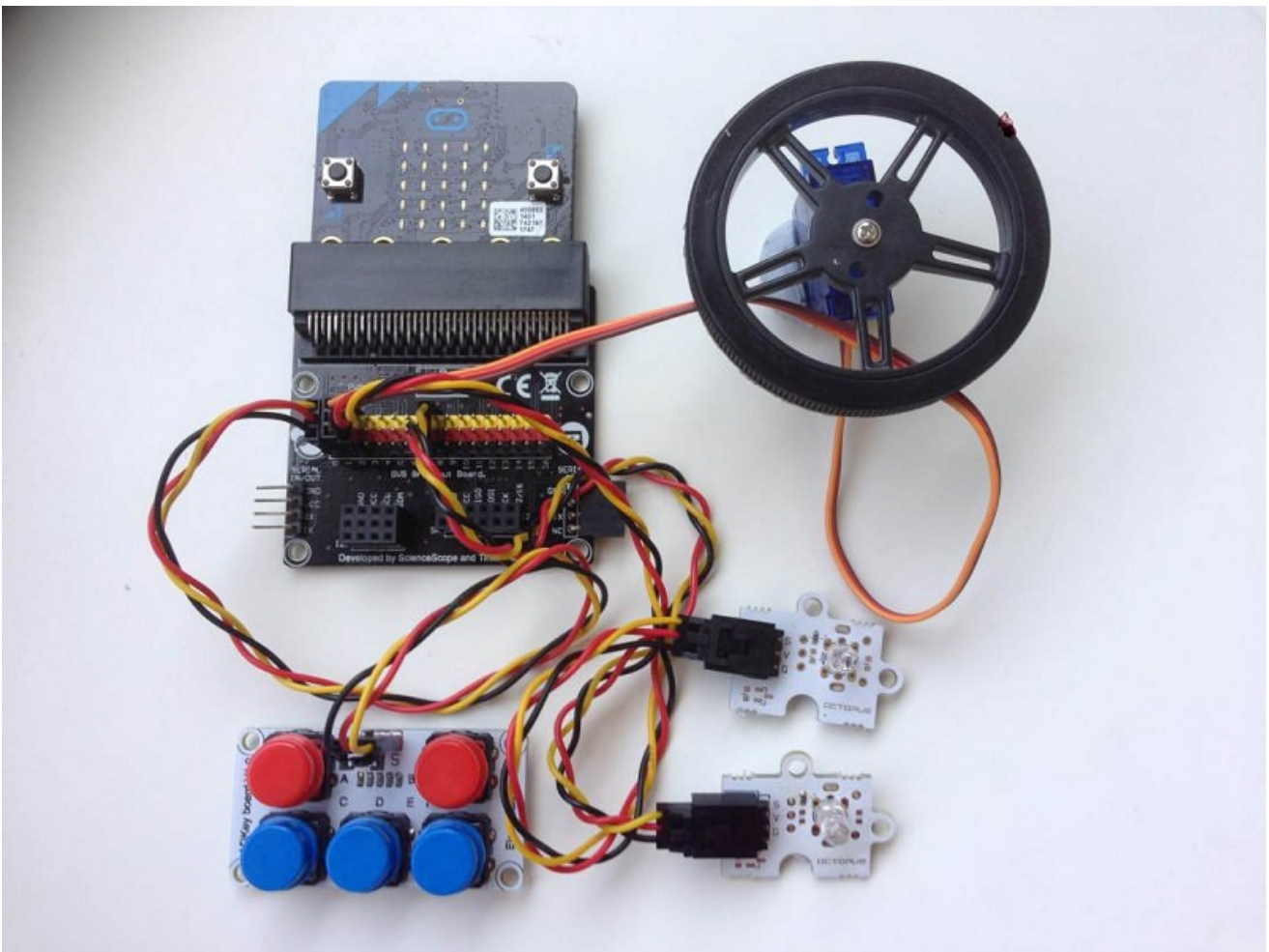
- 1 x ADKeypad
- 1 x Battery Pack
- 2 x Yellow LED
- 1 x Breakout Board
- 1 x micro:bit
- 1 x Mini Servo

Step 1 – Choose a cat



- Find a cat image and stick over a cardboard for support.
- Choose a cat personality of your choice. Happy, relaxed or friendly!

Step 2 – Connect electronic parts

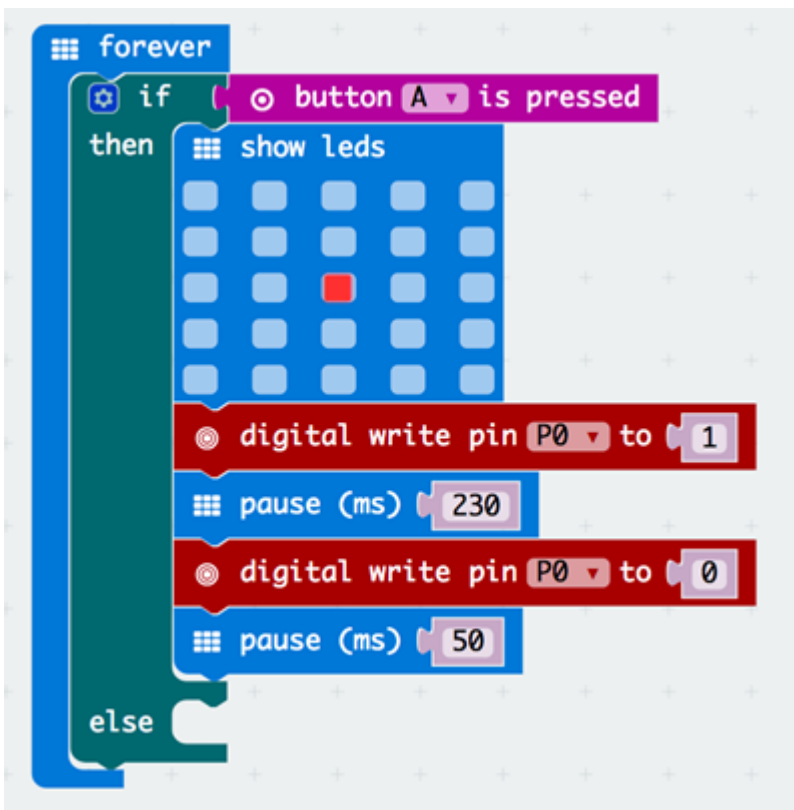


Connect the following on the breakout board

1. ADKeypad to Pin '0'.
2. Mini Servo to Pin '1'.
3. Two LED lights to Pin '2' and Pin '8'.

The colours of the jumper wires don't actually affect how the electronics work. But it is good practice to follow a colour convention so that you can easily identify where which cables are attached to.

Step 3 – Join up electronics to cardboard



1. Mount the mini servo to right side of your cardboard cat.
2. Create two openings on the cardboard for the LED lights to shine through

Step 4 – Code Microbit

The image shows two Scratch code snippets. The top snippet is a 'forever' loop containing an 'if' block. The 'if' block checks if key 'A' is pressed on ADKeyboard at pin P0. If true, it executes a sequence: 'servo write pin P1 to 0', 'pause (ms) 1000', 'servo write pin P1 to 90', 'pause (ms) 1000', 'servo write pin P1 to 0', and 'pause (ms) 1000'. If false, it executes 'digital write pin P1 to 0' followed by 'pause (ms) 1000'. The bottom snippet is also a 'forever' loop with an 'if' block. The 'if' block checks if key 'A' is pressed on ADKeyboard at pin P0. If true, it executes 'toggle LED at pin P2 On' and 'toggle LED at pin P8 On'. If false, it executes 'toggle LED at pin P2 Off' and 'toggle LED at pin P8 Off'.

1. Create block code in make code on the left.
2. When Button 'A' is pushed (Two LED lights up, Servo motor turns)

Cool stuff!

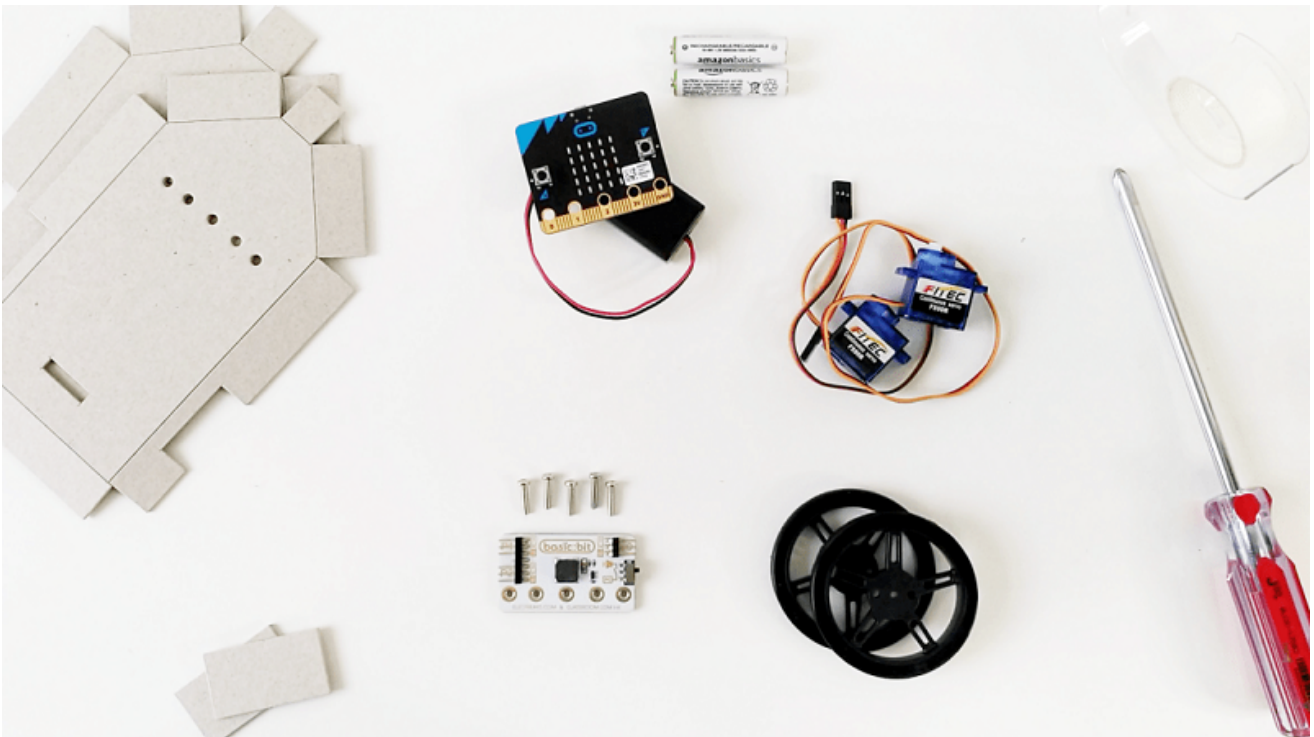
Now you've brought your cat to life. Think of a variety of movements for your cat. Enjoy and add them on to the cat!

30. case 28 Put together the Krazy Kar v2

30.1. Put together the Krazy Kar v2

- Got our Krazy Kar Kit and ready to get started? Follow along to put it together here. Don't like instructions? Use your creativity and make a crazy octopus instead.

Goals



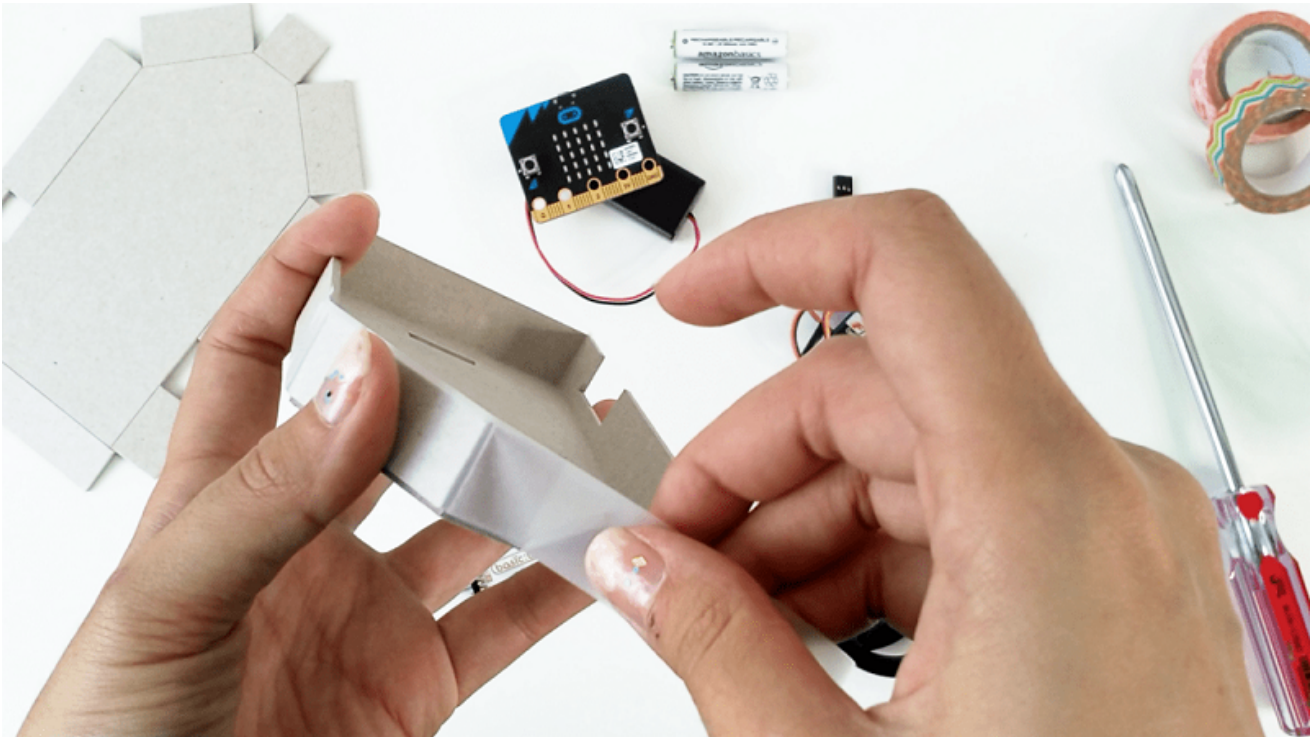
1. Make the Krazy Kar.
2. Don't break anything.
3. (Or if you break anything, learn how to fix it).

Materials

- 1 x Krazy Kar Shell
- 2 x Continuous Servos
- 2 x Wheels for Servos
- Some x Tape
- 1 x Basic:bit

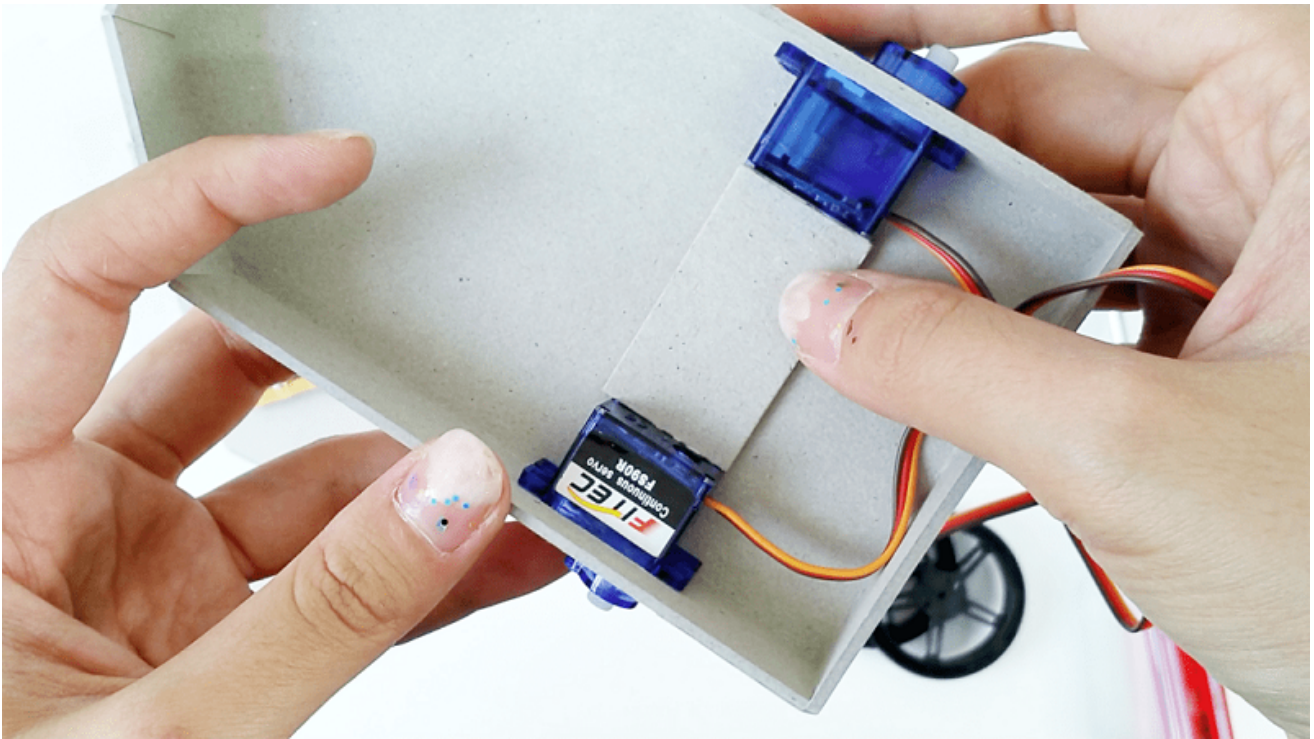
- 5 x Screws
- 1 x micro:bit
- 1 x Battery Pack

Step 1 – Shell it!



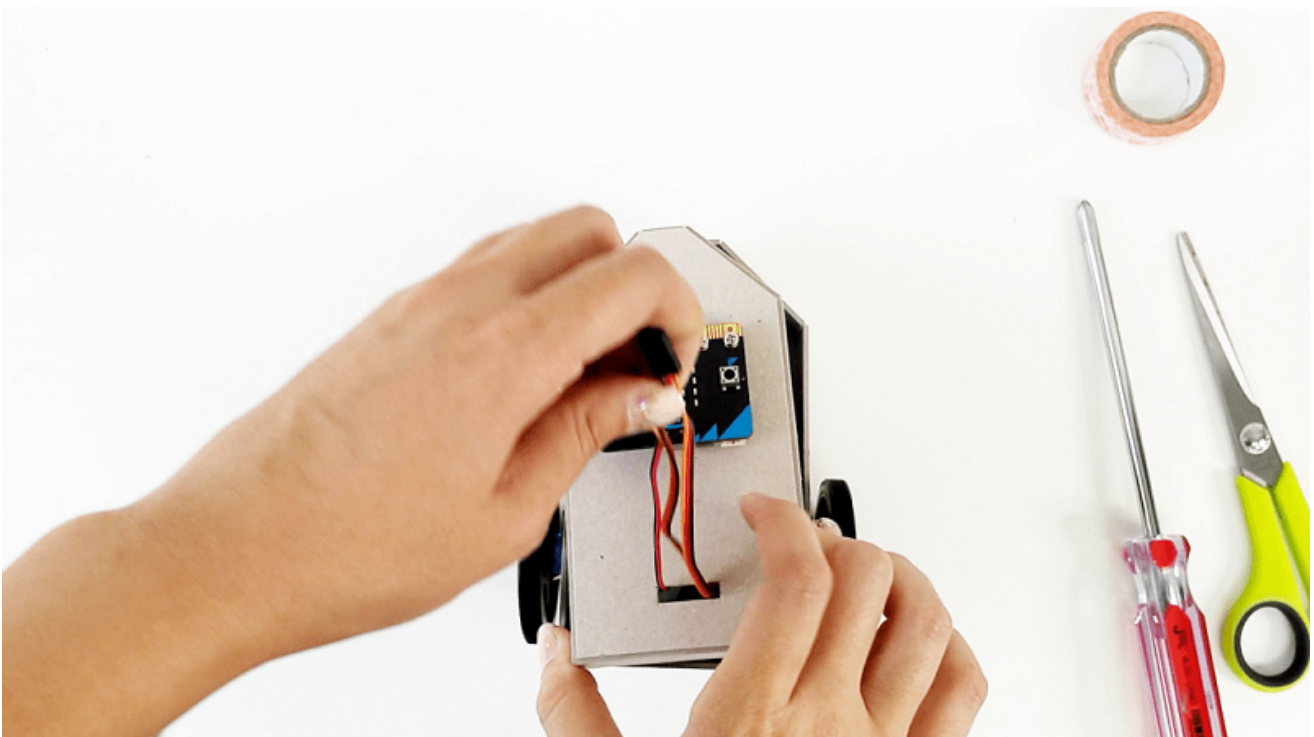
- Fold up the cardboard shell where the line cuts are.
- Tape it such that the sides stand 90 degrees to the base.
- You should be able to slot the top shell (one with 5 holes) into the bottom shell now.

Step 2 – Put in the Innards.



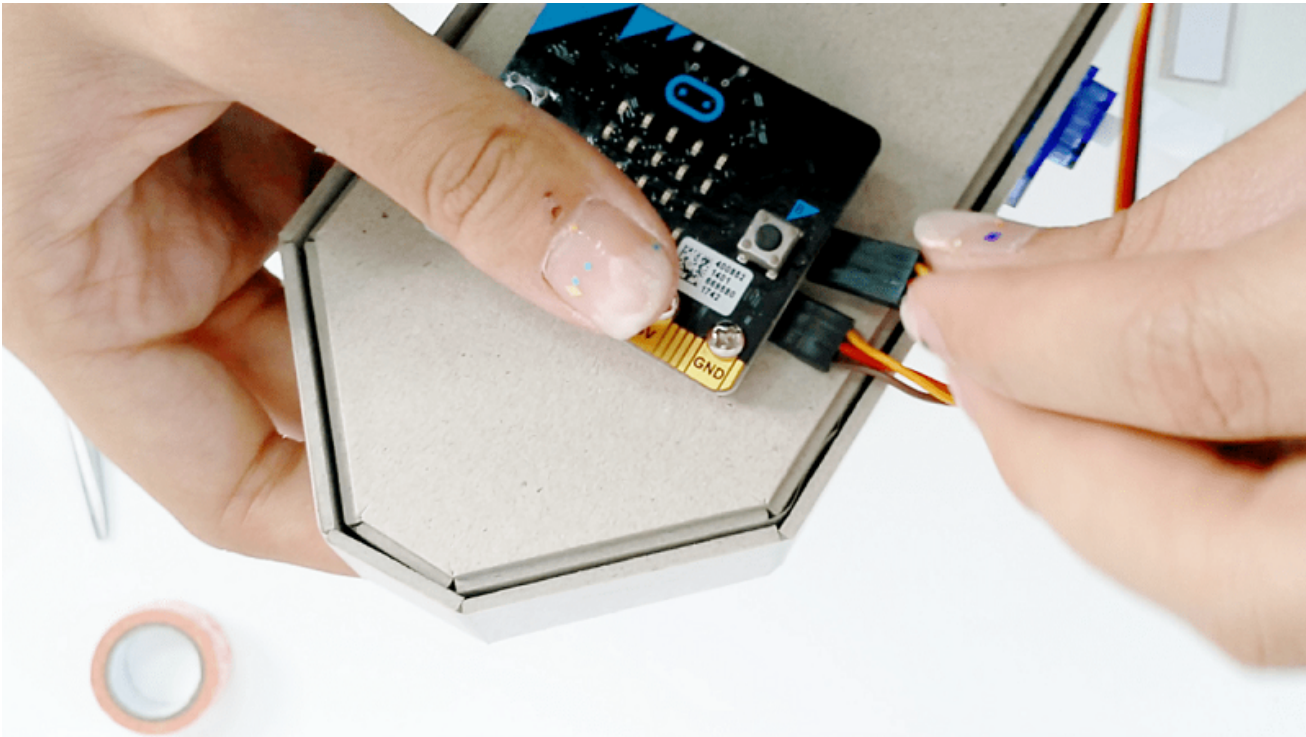
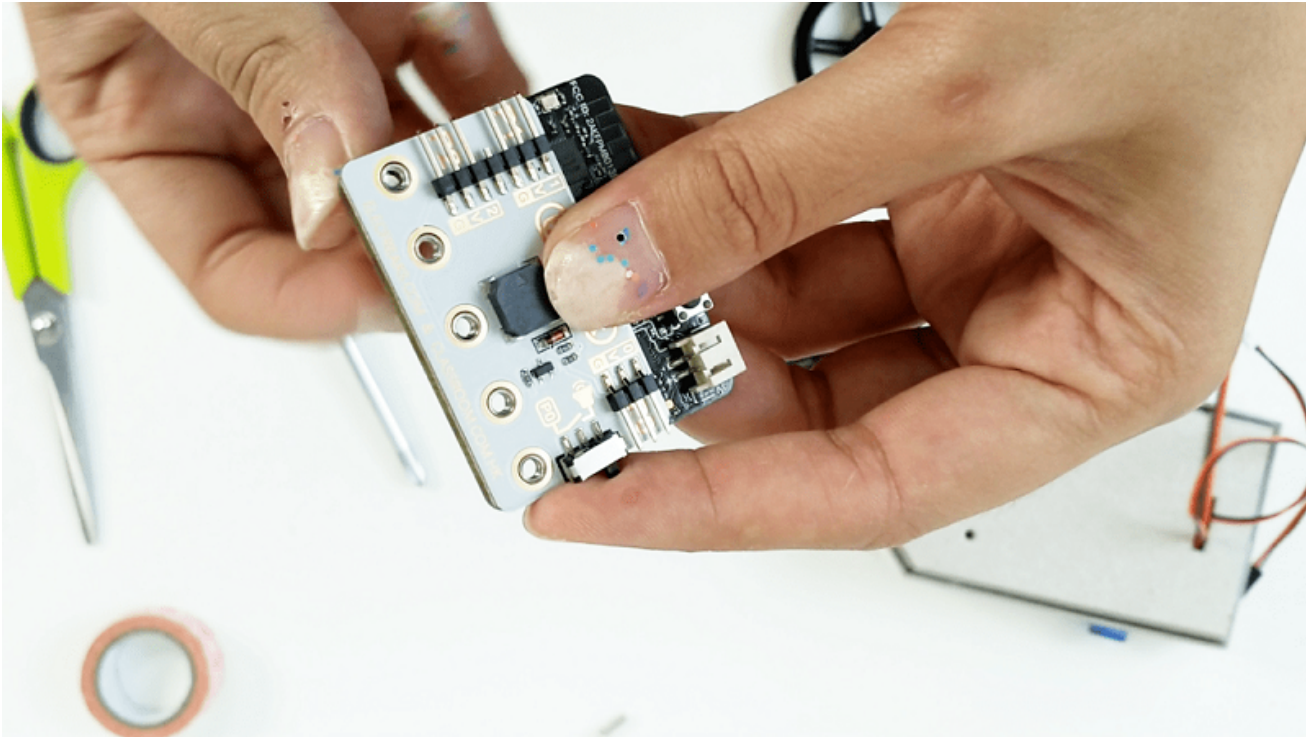
- Insert the two servo motors into the holes at the sides.
- Behind the stabilizing rectangle is some double-sided tape. Peel off the paper and push the rectangle in between the servos to secure them in. Stick it down well onto the base!
- Fill the battery pack with batteries and place it in the front of the crazy kar.

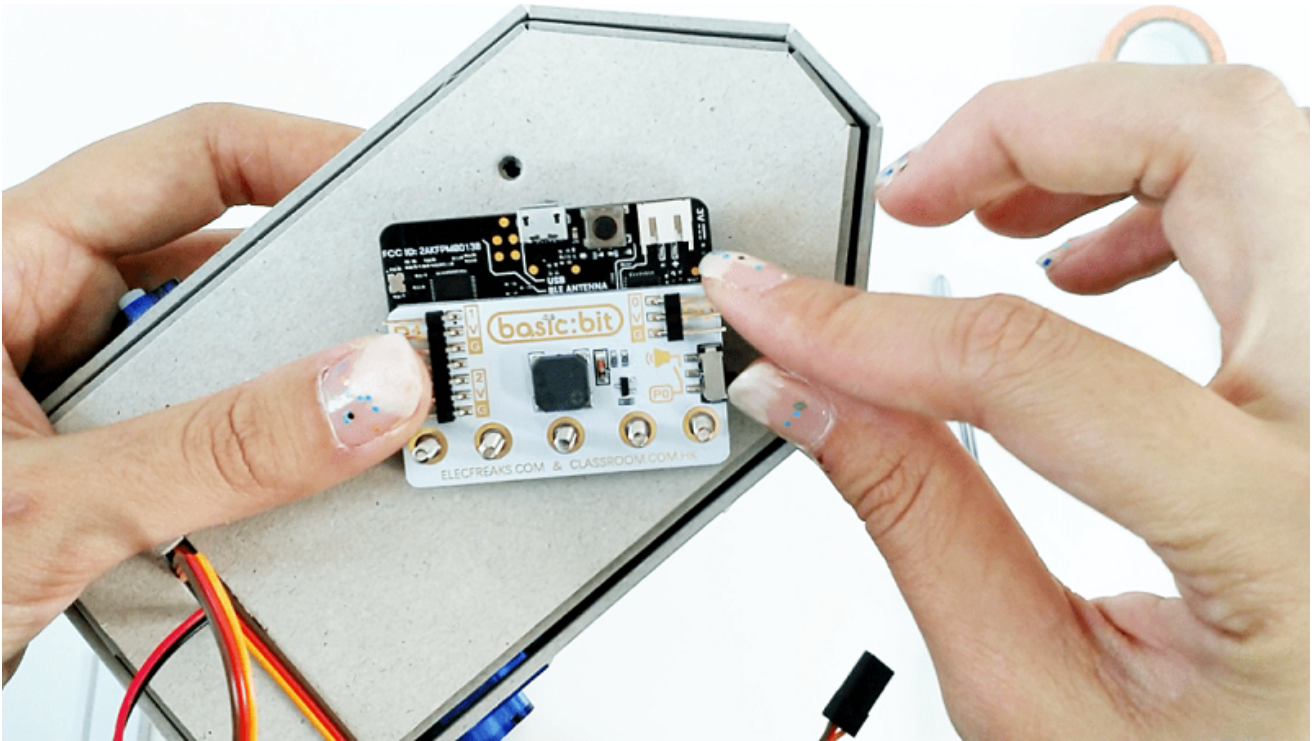
Step 3 – On to the Outside!



- Screw the wheels onto the continuous servo motors.
- Thread the servos and battery pack's wire through the rectangular hole in the top shell. And fit the top shell onto the bottom.

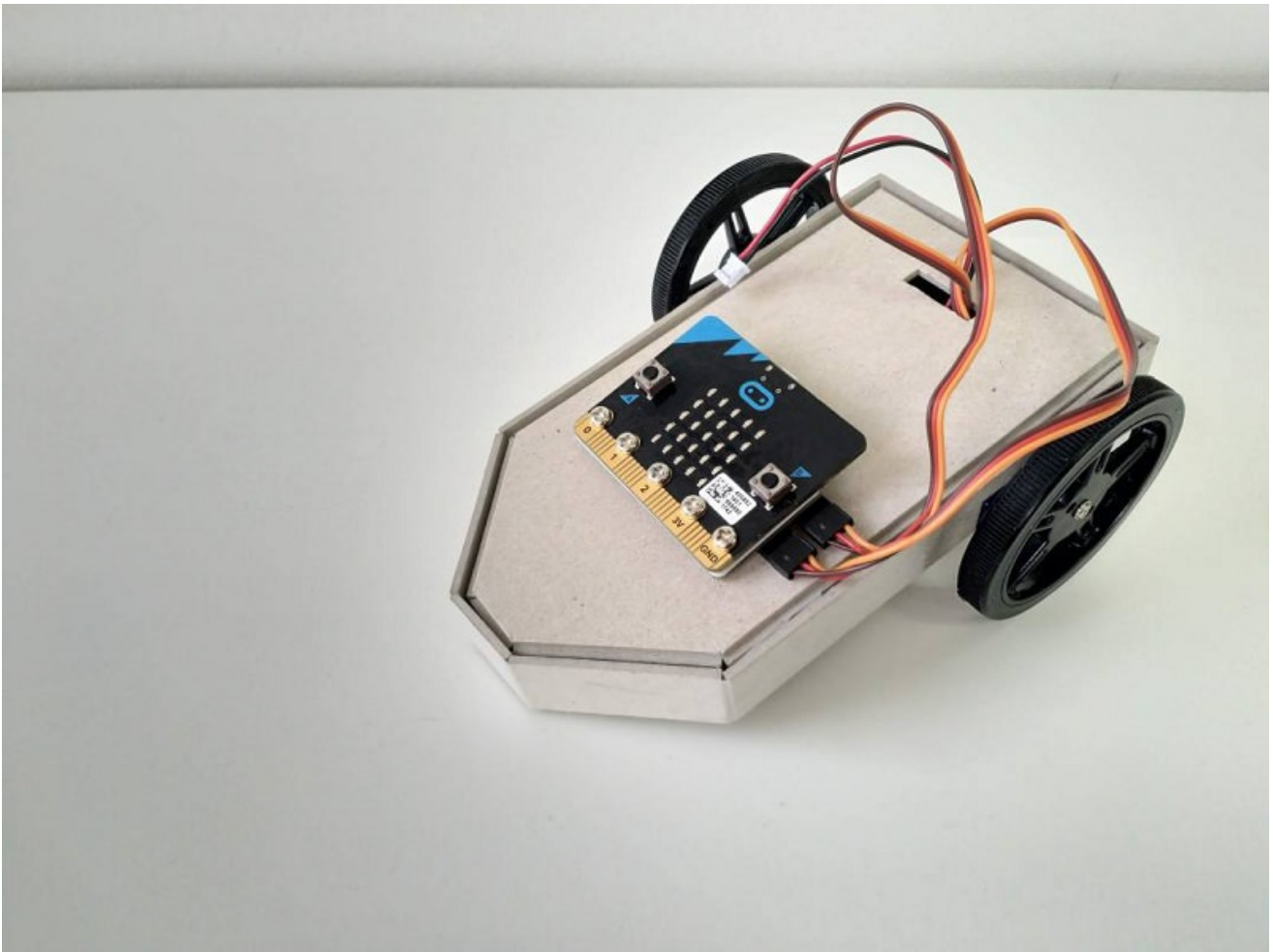
Step 4 – Upgrade the micro:bit





- Screw the micro:bit onto the basic:bit using the longer screws provided. Note that the holes should align, P0 to P0 and G to G.
- Affix the two servos into P1 and P2 of the basic:bit, making sure that the brown wires goes to G and the yellow wires go to S.
- Switch the P0/Buzzer switch on the basic:bit to buzzer. If it was already there – lucky you.

Step 5 – On to the Outside!



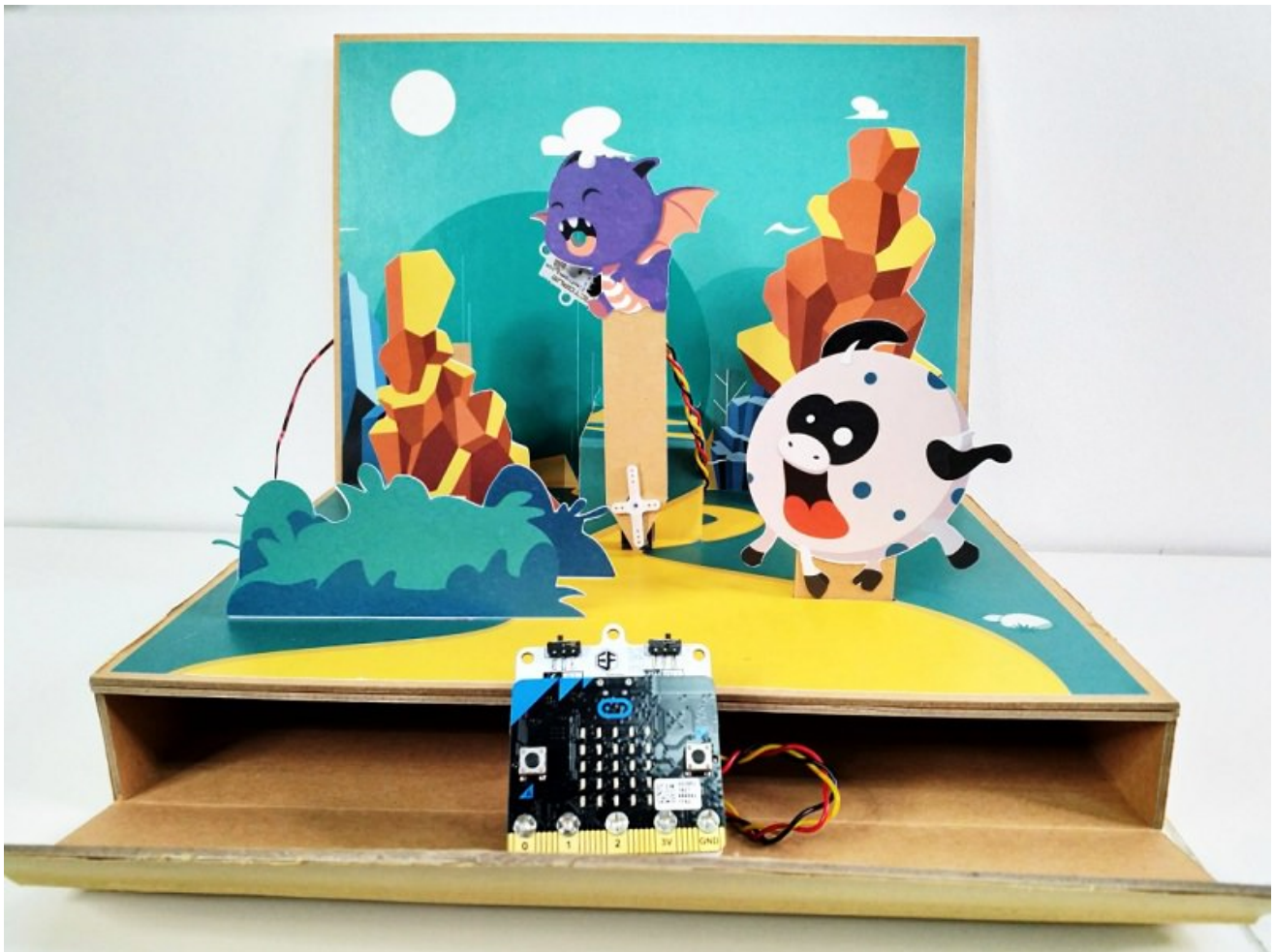
- Using the 5 screws, fit the micro+basic:bit cyborg into the 5 holes on the top of the crazy kar.
- Decorate decadently. And code it to get moving!

31. case 29 Shoot Em Up Kit

31.1. Shoot Em Up Kit

- You've got a town to save and a dragon to shoot! This here is the formula to get your own tabletop shooter arcade working in good form!

Goals



1. Connect all the parts of the Shoot Em Up Kit.
2. Code the micro:bit to fly dragons, detect lasers and score your player.

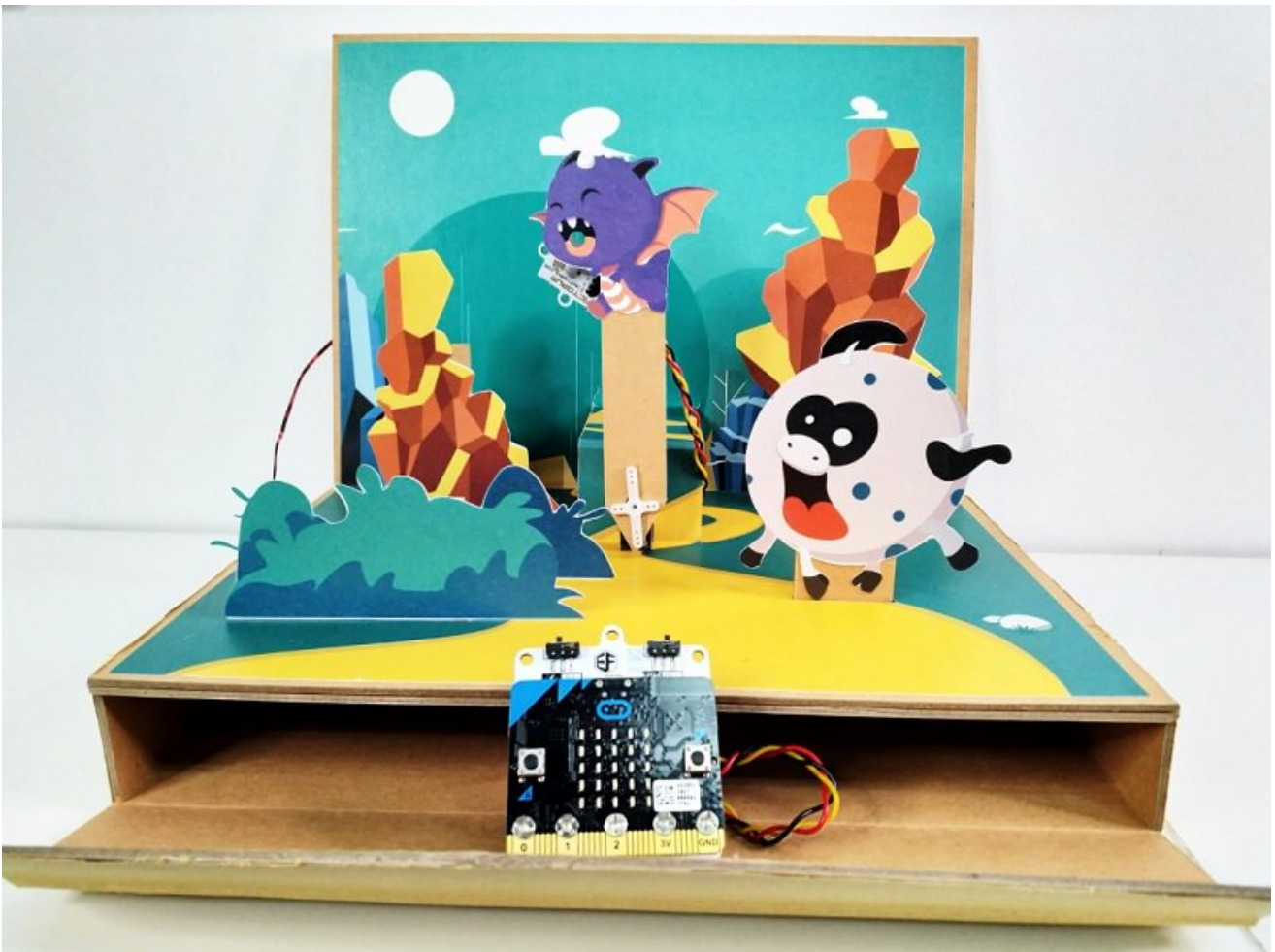
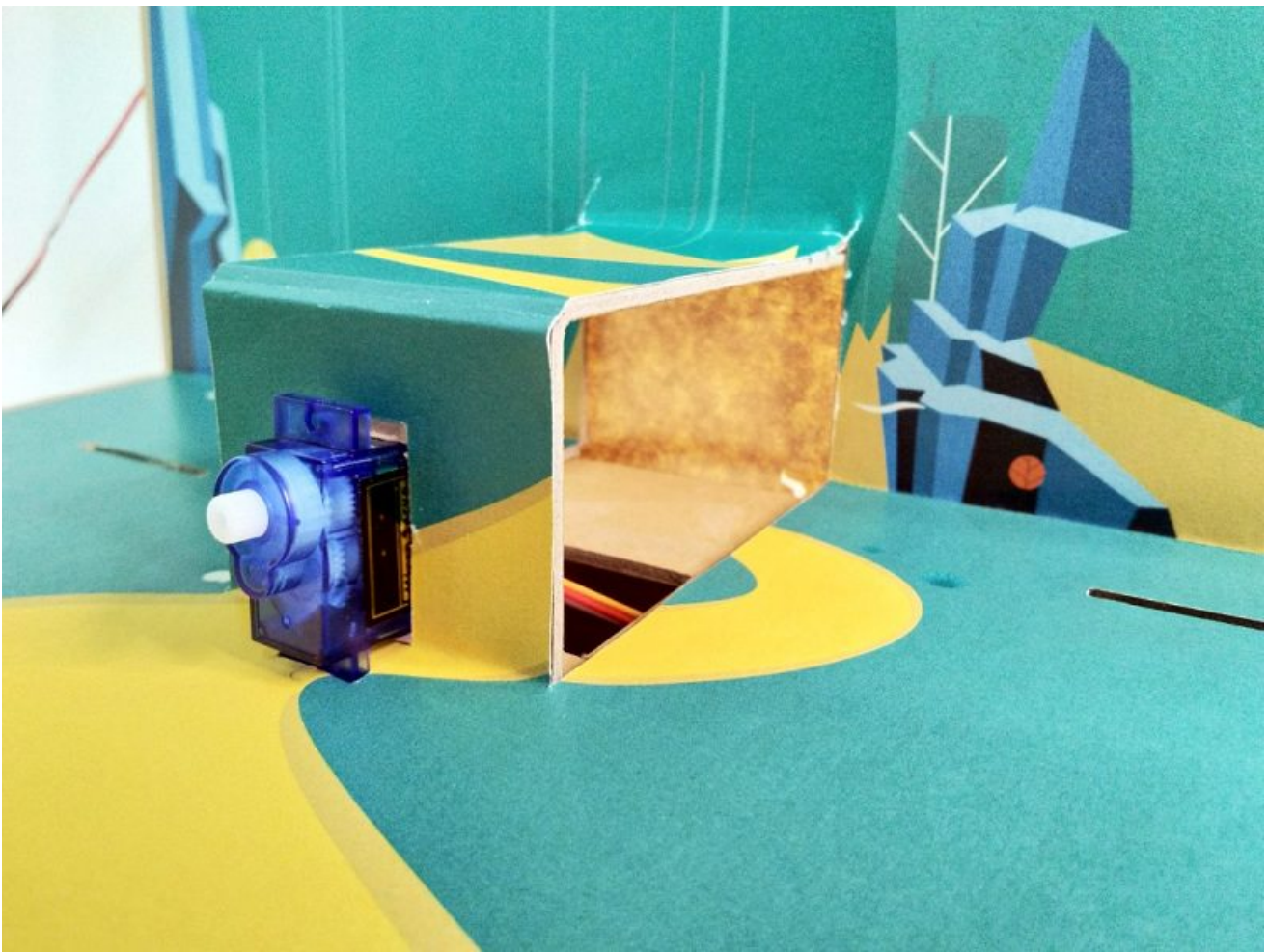
Materials

- 1 x Shoot Em Up Box

- 1 x Light Sensor
- 1 x Servo motor
- 1 x Ring:bit
- 5 x Screws
- 1 x micro:bit
- 3 x AAA batteries

Step 1 – Put your town in order!

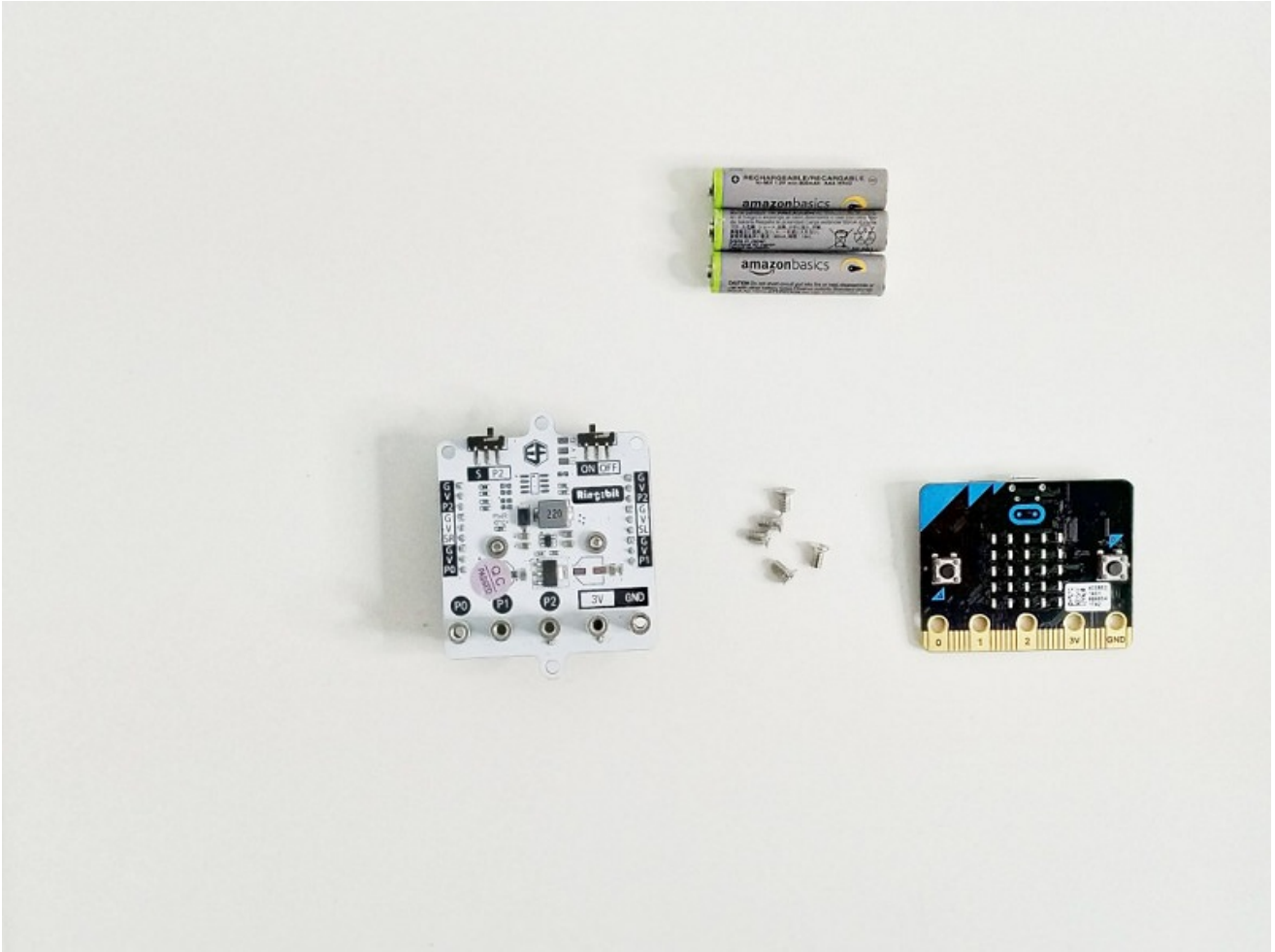


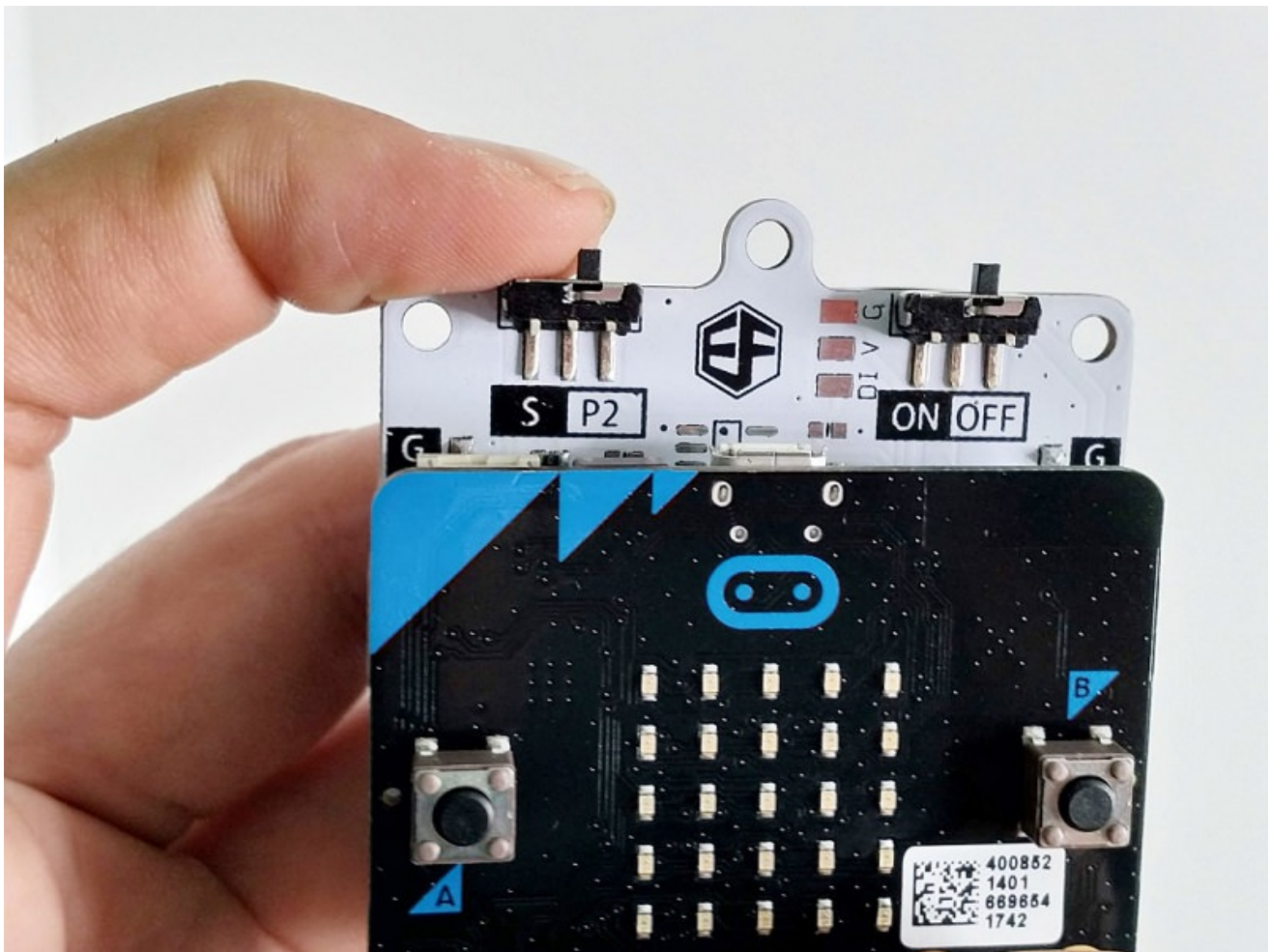


- Insert all the mountain cutouts and bushes around the path to your town.
- Fit the servo motor into the hole at the pop-up center.

- Thread the servo's wires into the hole and out the box at the front.

Step 2 – Attach your ring:bit





- Attach the micro:bit atop the ring:bit and screw it down with all 5 screws.
- Insert all 3 batteries into the back of the pack.
- Switch the left switch to P2 and the right switch to OFF. We'll turn it on after we've coded the micro:bit.

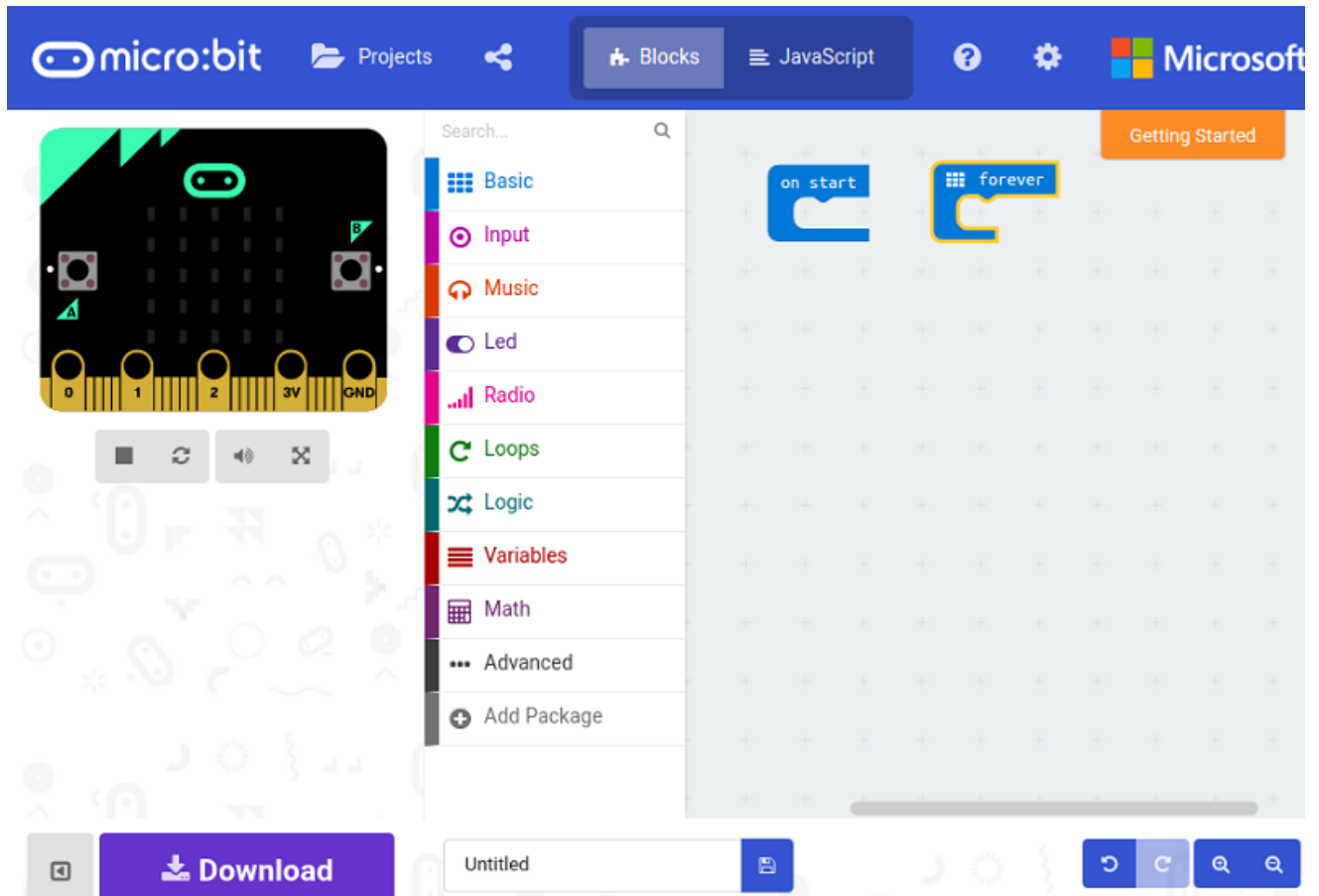
Step 3 – Connect all the parts!!



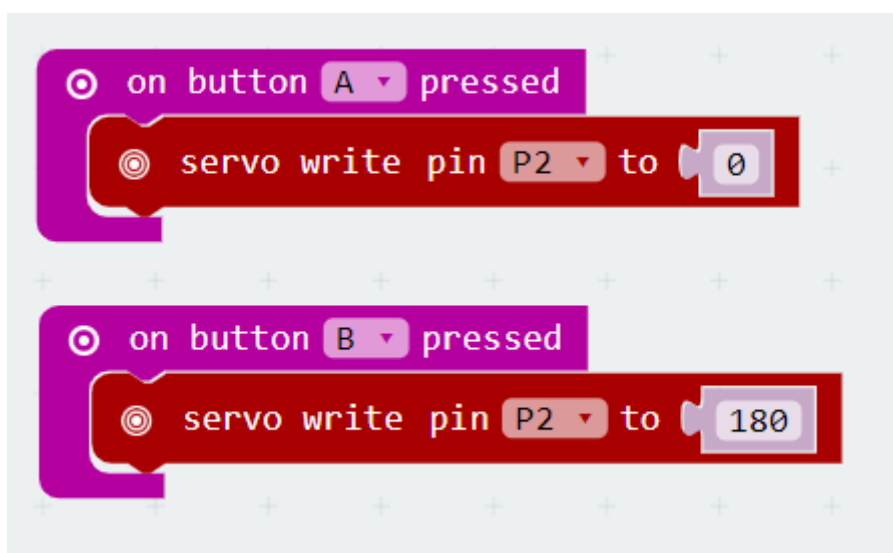
- Connect the servo to P2 on the ring:bit. And make sure that the brown wire goes to G and the yellow wires go to P2.

- Connect the light sensor to P1 on the ring:bit. The black wire should connect to G.

Step 4 – Code it up!



Step 5 – The real challenge – calibration.

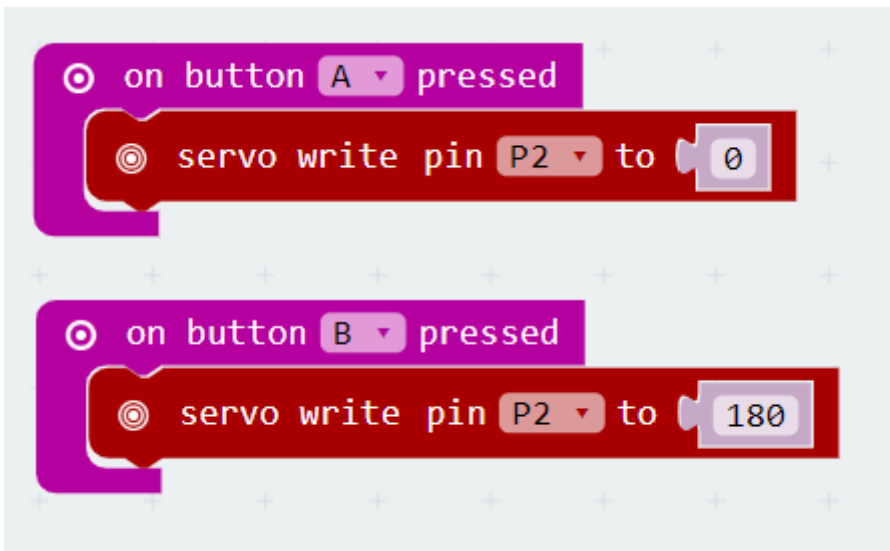
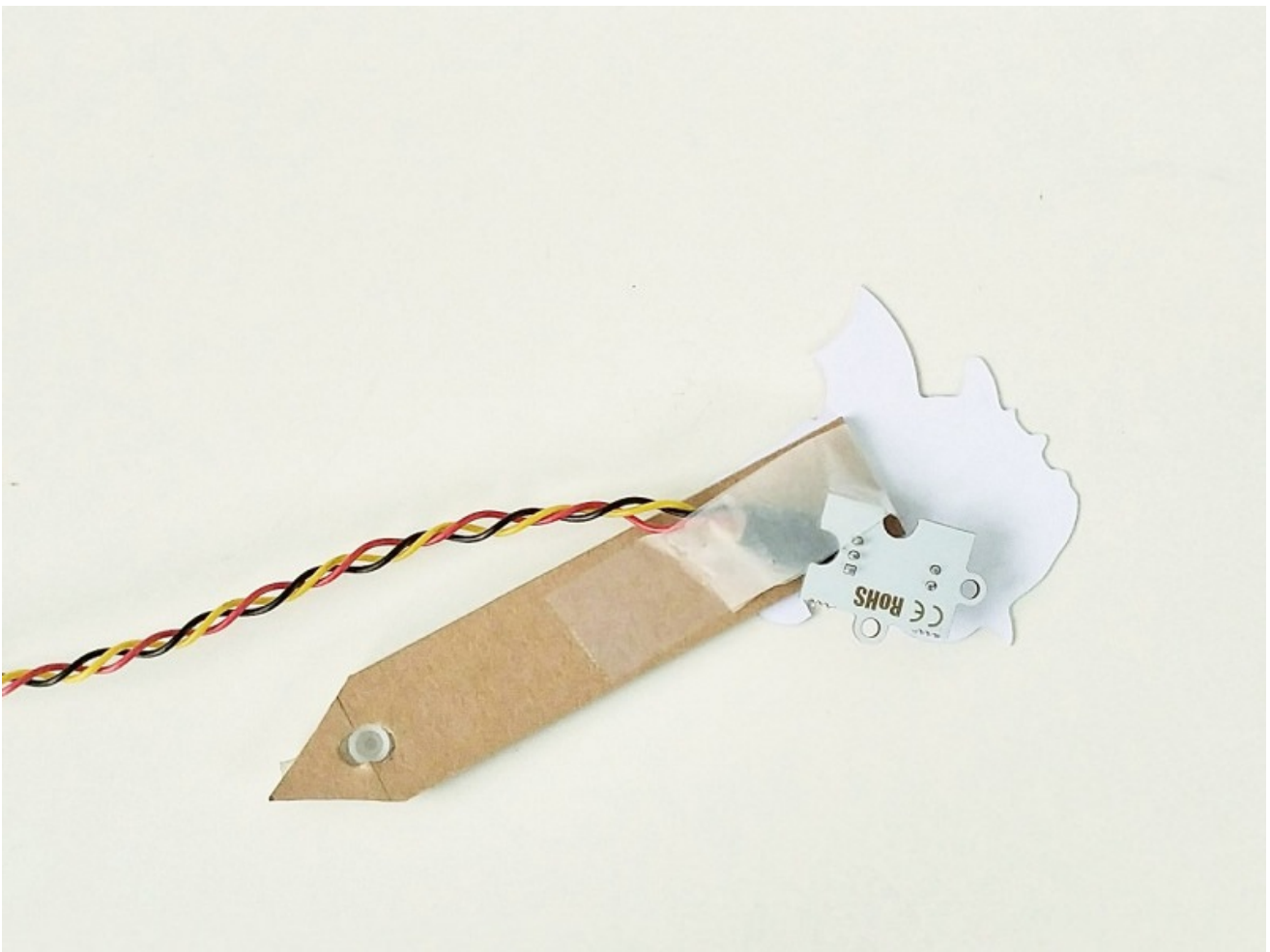


Calibrate your dragon to make sure it's flying back and forth in the sky and not face flat into the ground repeatedly.

- With your servo firmly in place – code the micro:bit to turn your servo to 0 deg when button A is pressed, and to turn to 180 deg when button B is pressed.
- Place a servo arm onto the head of the servo after pressing button A and watch to see where it goes when you press button B. If the servo arm points too far down or to the side, adjust the arm and reduce the angle coded. (e.g. adjust 0 – 180 deg to 20 – 160 deg).
- Replace the servo arm with the dragon arm at the same angle and screw it down to secure it.

Step 6 – Calibration Part 2

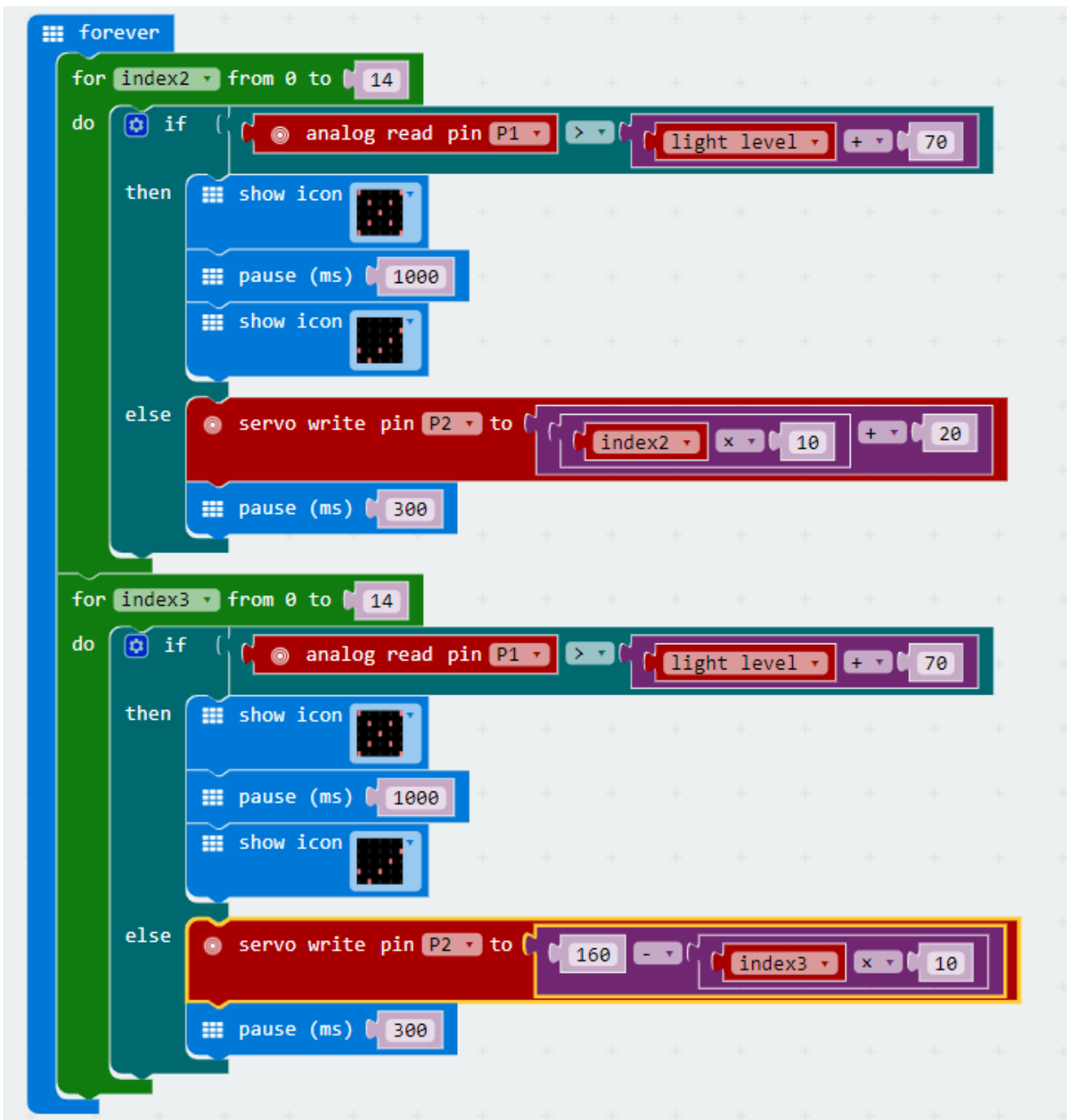




On to the light sensor. We've got to capture the current light levels and when a laser passes over, the sensor will alert the micro:bit that the light levels has suddenly gone way up.

- Thread the light sensor through the hole under the pop-up and to the front of the box.
- Place the light sensor into the dragon's mouth and tape it down to secure it.
- Code the micro:bit to detect the light level through the light sensor when a button is pressed.

Step 7 – Putting it all together.



Now we want to be able to stop the dragon when it gets hit and also score points!

- So we should combine the two pieces of code and use an IF logic block to check if we got hit.
- IF we get hit, then we change the icon on the micro:bit and pause for a while, before changing back to the default icon.
- ELSE (if we don't detect and hits) then we allow the servo to move as per normal.
- Also - add in more blocks so when the dragon gets hit it increases a score variable.

Cool stuff!

You've killed the dragon. Now what? Add extra mountains, make it more challenging. Or write your own story, and see what you can do with your magical micro:bit!

32. case 30 Reaction Time Tester

32.1. Reaction Time Tester

- Test yourself with this Python-based mini-game for the micro:bit and OLED! Written by Jensen from Raffles Institution.

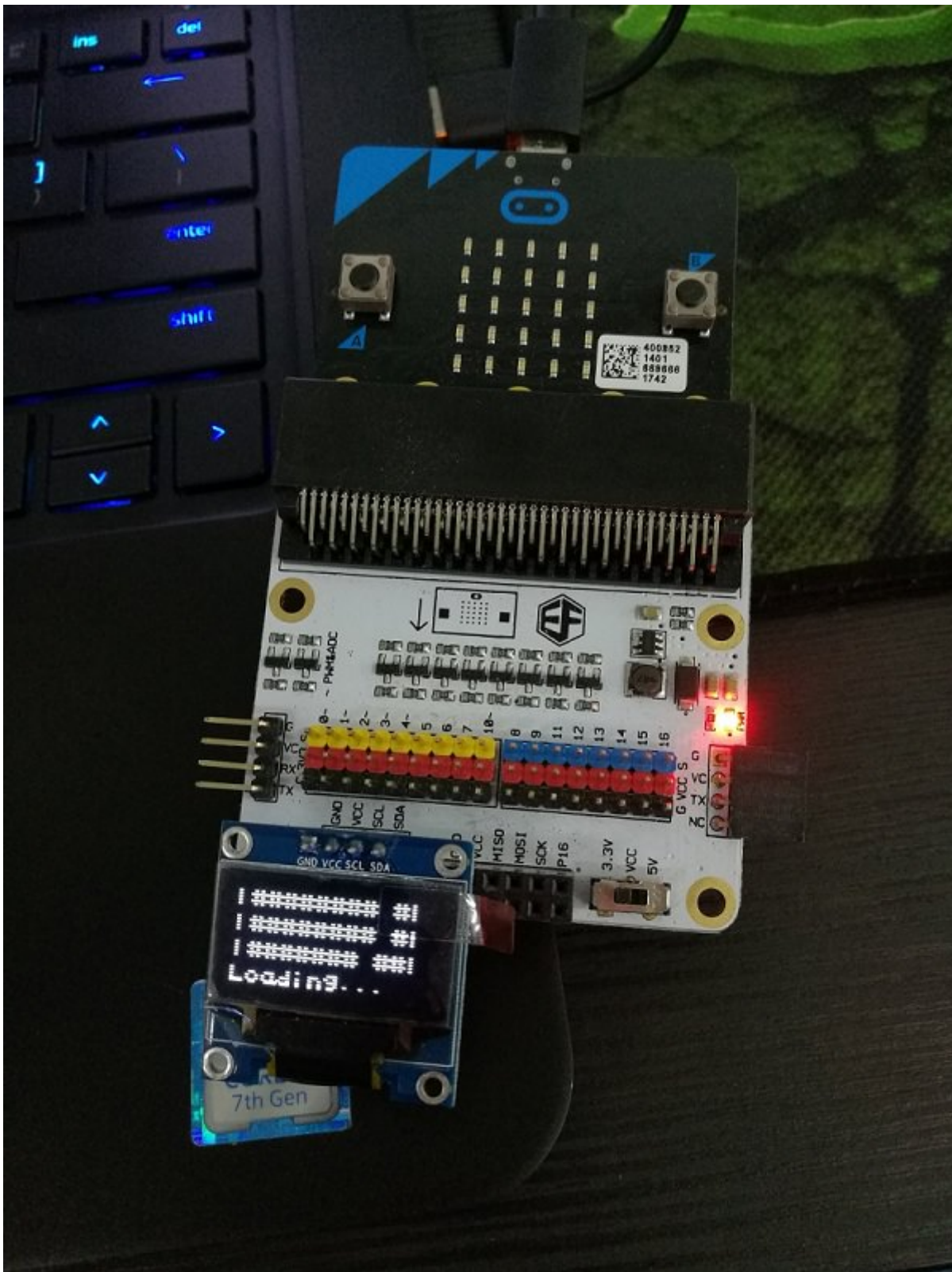
Goals

1. Assemble a reaction time tester.
2. Try not to break it when testing yourself!

Materials

- 1 x Tinker Kit (or OLED display)
- 1 x Brain
- 1 x You

Step 1 – Input/Output



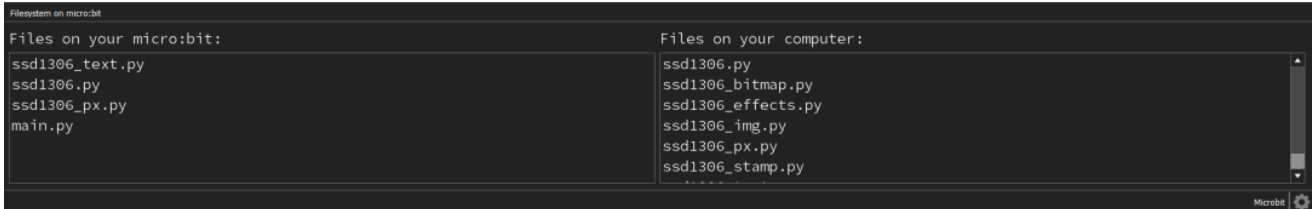
- Attach the MicroBit to the Breakout Board.
- Connect the LED to pin 12, which is a digital pin.
- Connect the light detector to pin 8, another digital pin.
- Brown to G (ground), red to V (voltage) and orange to S (signal).
- These tell us if light is on or off.
- Connect the OLED display to the I²C pins. Any one set will do.
- Connect your micro:bit to the computer, and download the Mu Editor.

Step 2 – File Transfer

```

from microbit import *
from ssd1306 import initialize,clear_oled,draw_screen
from ssd1306_px import set_px
from ssd1306_text import add_text
from utime import ticks_ms,ticks_diff
import random

```



- We need to download some modules to be used in the project.
- From This link, download the module as a zip file.
- Write the code (first screenshot) in Mu Editor to import the files.
- Be sure to have the files in the same folder as the project.
- Transfer a few of the files from your computer to the microbit.
- On Windows, be sure to put the files in a folder under users/"Username"/mu-code for them to be detected my mu-editor.
- These files are the modules that are not originally present on the micro:bit for you to import.

Step 3 – Loading Screens



- This step isn't really necessary , but it adds a little flair into your project.
- We display this loading on the OLED module
- Using the module "add_text", we can display text and other characters onto the display
- We show the animation using the function "loading_screen()"
- If you want more variations of the loading screen, head down to the bottom of the post.

Step 4 – Code the Game

```

#Game itself
seconds_to_wait = random.randint(0,10)
milliseconds_to_wait = str(seconds_to_wait) + '000'
milliseconds_to_wait = int(milliseconds_to_wait)

time1 = ticks_ms()
sleep(milliseconds_to_wait)
display.show("#")
time2 = ticks_ms()

```

- Here's the actual test itself.
- First, we have to have a time to wait before showing the indicator for the player to press the button.
- We randomly generate the number and parse it into milliseconds by adding "000" to the end of the number after turning the original number into a string.
- The variables time1 and time2 refer to 2 arbitrary points in time before the "#" (the indicator to the player) symbol is shown.
- One quirk about micropython and the MicroBit to note is that the time module is replaced by the utime module, and utime has to be imported instead of time.

Step 5 – A Little More Logic

```

84     while True:
85         if button_a.is_pressed():
86             time3 = ticks_ms()
87             display.clear()
88             break
89         else:
90             pass
91     clear_oled()
92     add_text(0,0,str(ticks_diff(time2,time1)))
93     add_text(0,1,"Reaction")
94     add_text(0,2,"Time")
95     add_text(0,3,str(ticks_diff(time3,time2))+ "ms")
96     sleep(10000)
97     continue
98

```

- This is the step where we calculate and display the player's reaction time.
- This is done by calculating the time between when the indicator is displayed and when the player presses the button.
- Then, we display the player's reaction time to the OLED display.



Bonus loading effects:

- This one utilizes the light sensor and the LED to start the game.
- It detects light and once the light is covered, the game will start.

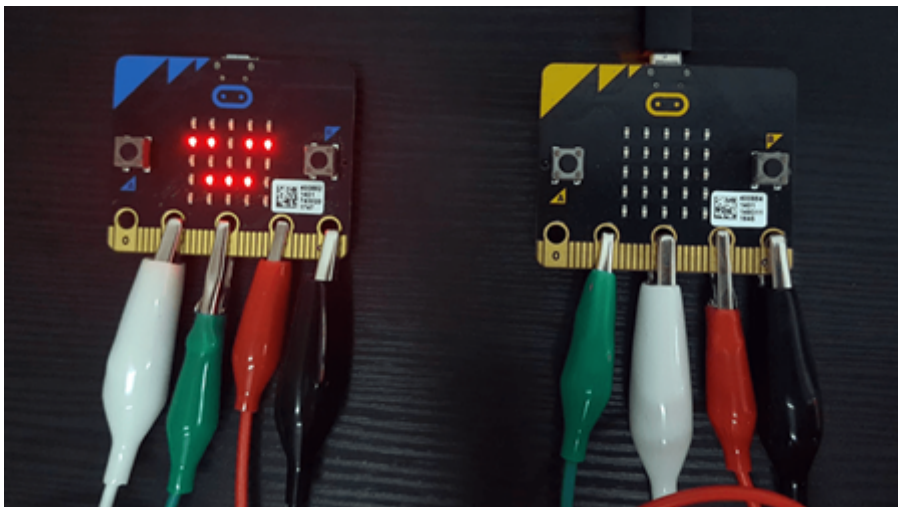
This next animation is simple: display the characters “3”, “2” , and “1”, as a countdown before starting the game. The code is pretty self explanatory.

33. case 31 morse code transmitter

33.1. Morse Code Transmitter

- Make a simple morse code transmitter using MakeCode, micro:bits, and some crocodile clips! This tutorial was written by Anahita from the University of California at Berkeley, during her summer internship in Singapore.

Goals



1. Connect two micro:Bits together.
2. Send signals from the first micro:Bit to the second micro:Bit by pressing the A and B buttons.
3. Receive signals from the first micro:Bit.
4. Learn how to code in MakeCode.

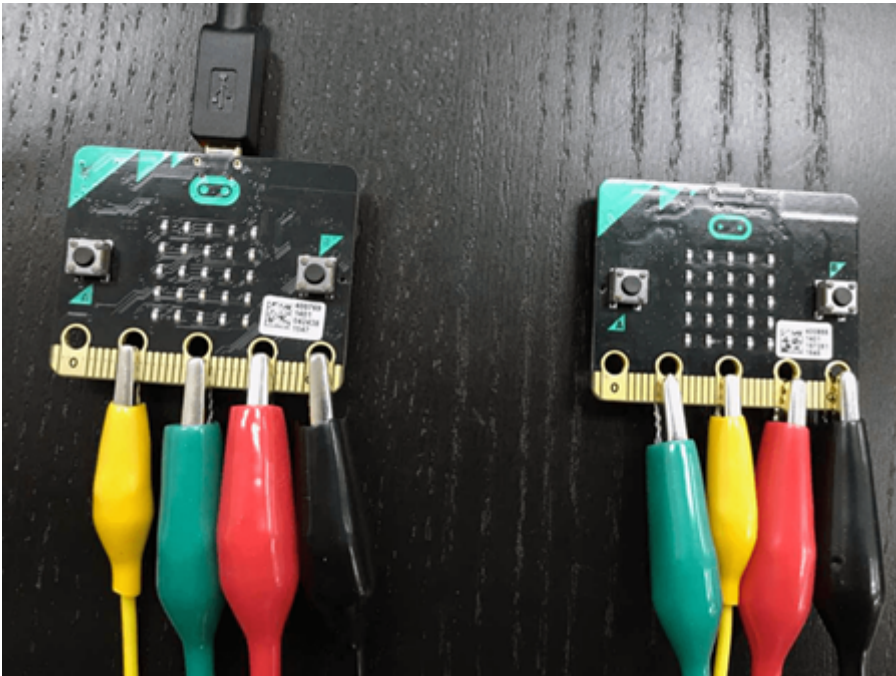
Materials

- 2 x micro:Bits
- 4 x Crocodile Clips
- 1 x micro USB cable

Step 0 – Preview

- We will be writing two sets of code: one for the sending micro:Bit and another for the receiving micro:Bit
- In order for the receiver to know which signal is being sent, we will adjust the length of time between when the signal turns “on” and “off”
- That way, we can differentiate the two signals by the pause length

Step 1 – Crocodile Clips

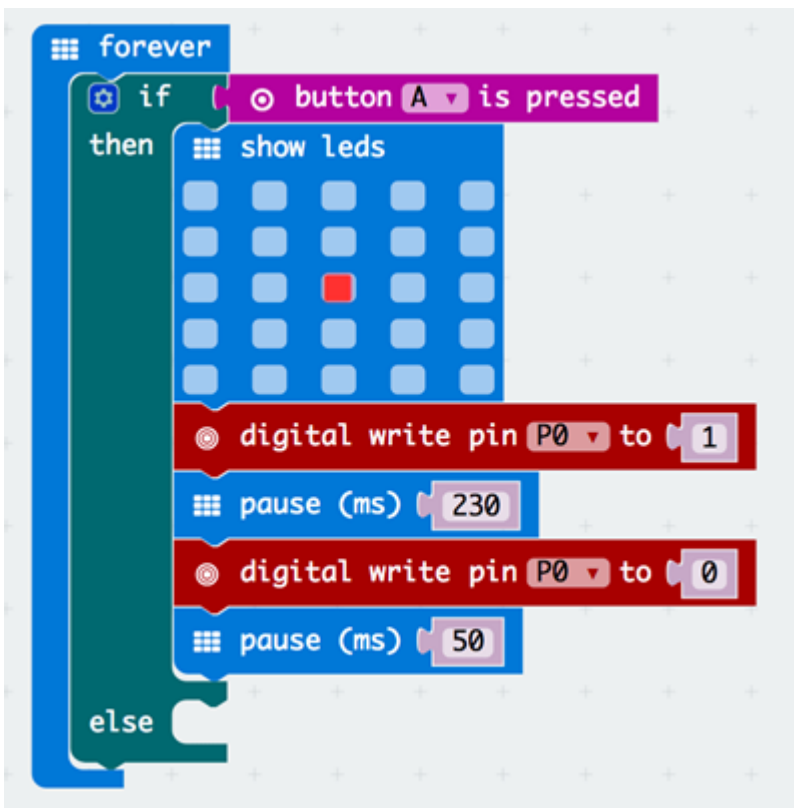


We want to send signals from pin 1 of the first micro:Bit to pin 2 of the second micro:Bit (and vice versa)

Connect:

1. GND to GND
2. 3V to 3V
3. Pin 1 to Pin 2
4. Pin 2 to Pin 1

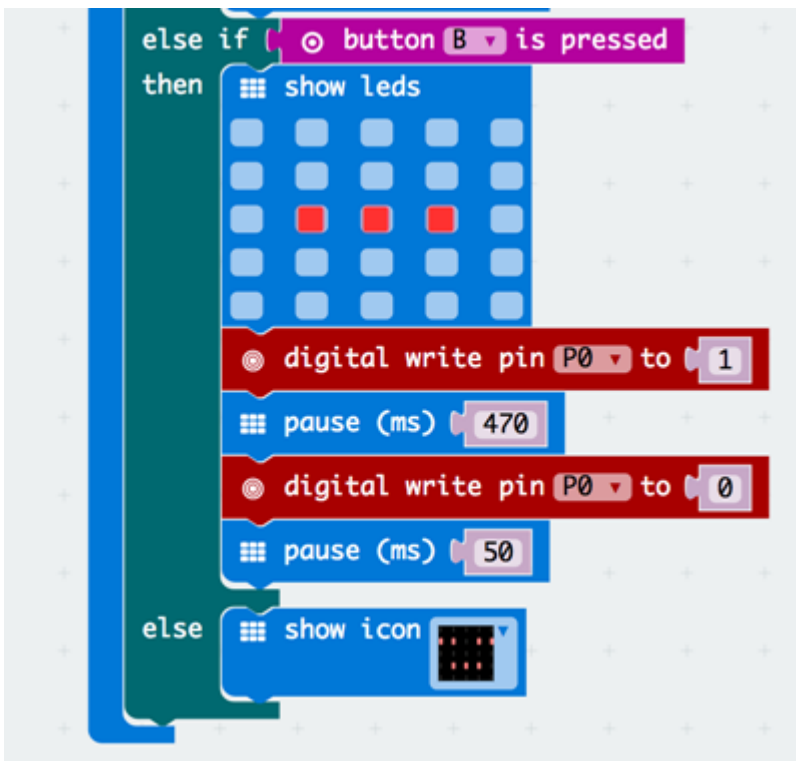
Step 2 – Sender: button A



We want a signal to be sent when the A button is pressed. Let's make this one the "dot" signal.

1. Open up MakeCode and name this file "Sender"
2. Drag and drop an if-then-else block from the Logic drawer to the Forever block
3. From the Input drawer, attach a button A is pressed block to the if section
4. From the Basic drawer, attach a show led block in the then section to display the "dot" on the sender screen
5. Add a digital write pin block from the Pins drawer (under Advanced) and set it to 1(This means the signal will turn "on";Make sure to change it to p1 since that's where we attached the clip)
6. Add a pause block from the Basic drawer and set it to 230 ms(This pause will be associated with the "dot" signal)
7. Add another digital write pin block and set it to 0(This means the signal will turn "off")
8. Add another pause for 50 ms just to give it some time

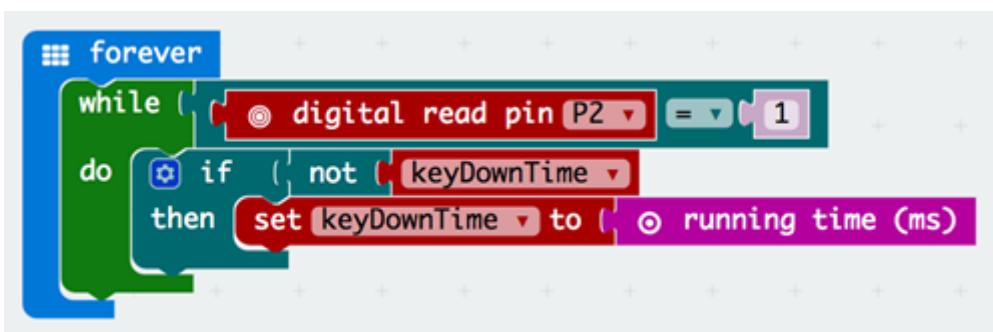
Step 3 – Sender: Button B



We want a “dash” signal to be sent when the B button is pressed.

1. Add an else if block to the if-then-else block from earlier
2. Repeat the steps as with Step 2, except (Use button B is pressed, Display a “dash” in the LED, Pause for 470 ms)
3. Add a clear screen block from Basic or an icon to the else statement

Step 4 – Receiver: detecting the signal

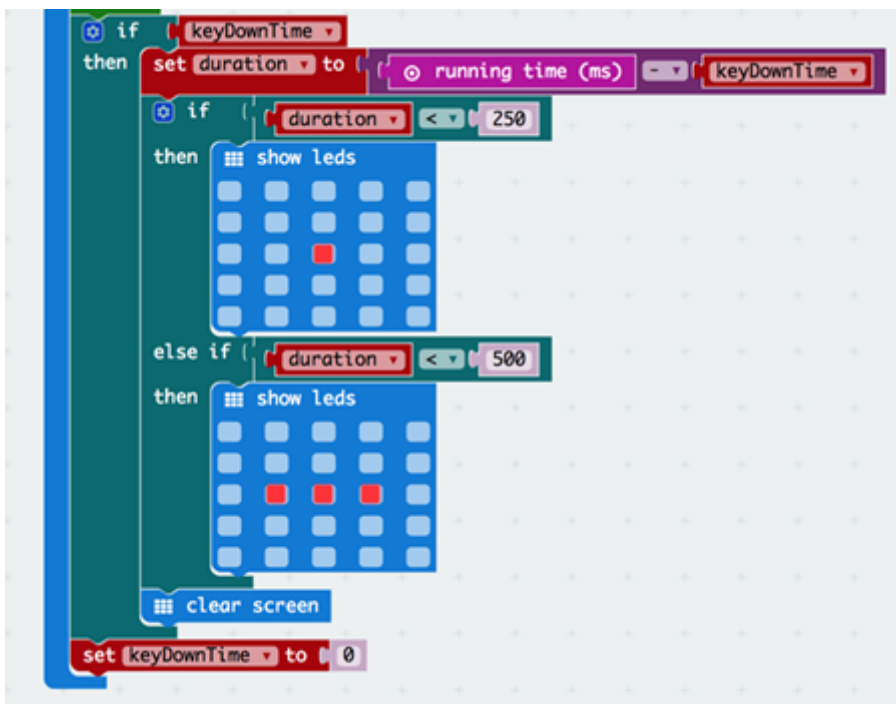


We want to record the duration of time between whenever the signal is received and when it stops. We will be using the running time (ms) block for this.

1. Create another project on MakeCode called “Receiver”
2. Drag a while loop from the Loops drawer
3. From the Logic drawer, attach an equals sign block to the while loop
4. Attach a digital read pin block to the equals sign block and set it equal to 1 (This means that a signal is being detected; Make sure to change it to p2 since that’s where the crocodile clip is)
5. In the Variables drawer, make a variable called “keyDownTime”

6. Attach an if-then block to the body of the while loop
7. Attach a not block from the Logic drawer to the if statement and then attach the keyDownTime variable to it
8. You can find the running time (ms) block by searching for it in the search bar

Step 5 – Receiver: displaying the signal



We want to display the correct signal on the screen.

1. Drag and drop an if-then block underneath the while loop and attach the keyDownTime variable to it (This is so that this block of code will only run if a signal has been detected)
2. Create another variable called “duration” and set it to be the difference between running time and keyDownTime (The minus operation is under the Math drawer, This variable tells us how long it’s been since the program started running and when the signal was detected)
3. Drag an if-then block and attach a less than block from the Logic drawer and make it so that it’s duration < 250 (We chose 250 ms since the “dot” takes 230 ms)
4. Display the “dot” led in the body of the if statement
5. Add an else if block to the if-then-else block from earlier and do the same thing as above except the “dash” threshold is 500 since the “dash” takes 470 ms and then show the “dash” led
6. Add a clear screen so that the screen clear after a signal comes in
7. After the first if-then block make sure to set keyDownTime to 0 so that it works every time you send a new signal

Done!

Make sure to flash the code to the respective micro:bits and test it out! It should display the same signal on both screen when you press a button.

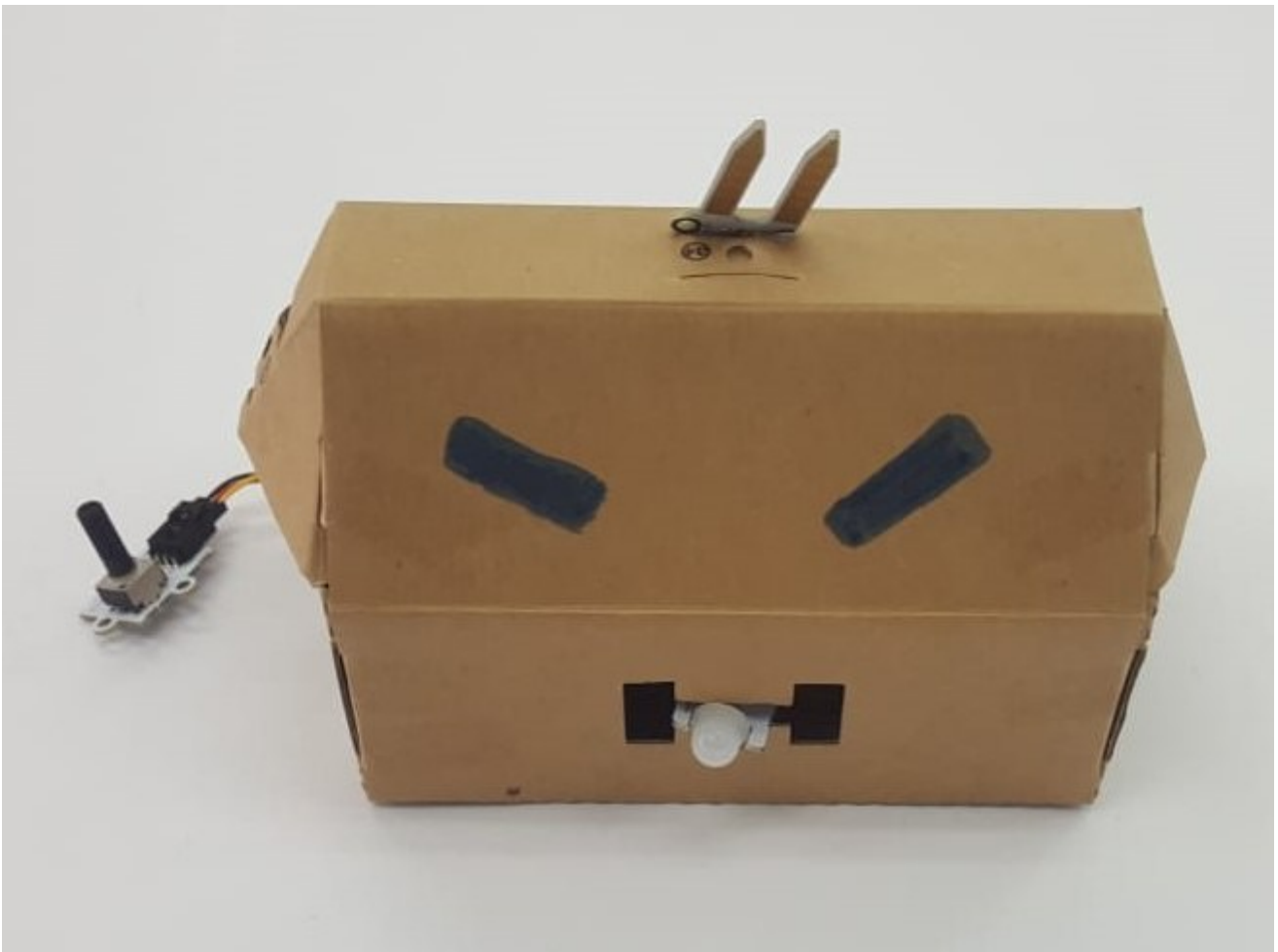
For an added challenge, try to translate the morse code on the second micro:bit.

34. case 32 reclusebot

34.1. Reclusebot

- Make a reclusive robot that squeals when toggled, touched, or when it detects motion
Use the micro:bit to make a robot that squeals when it detects motion, is touched on any of its sensors or when lifted up. Written by Shaun Toh, from the Singapore University of Technology & Design, on his summer internship.

Goals

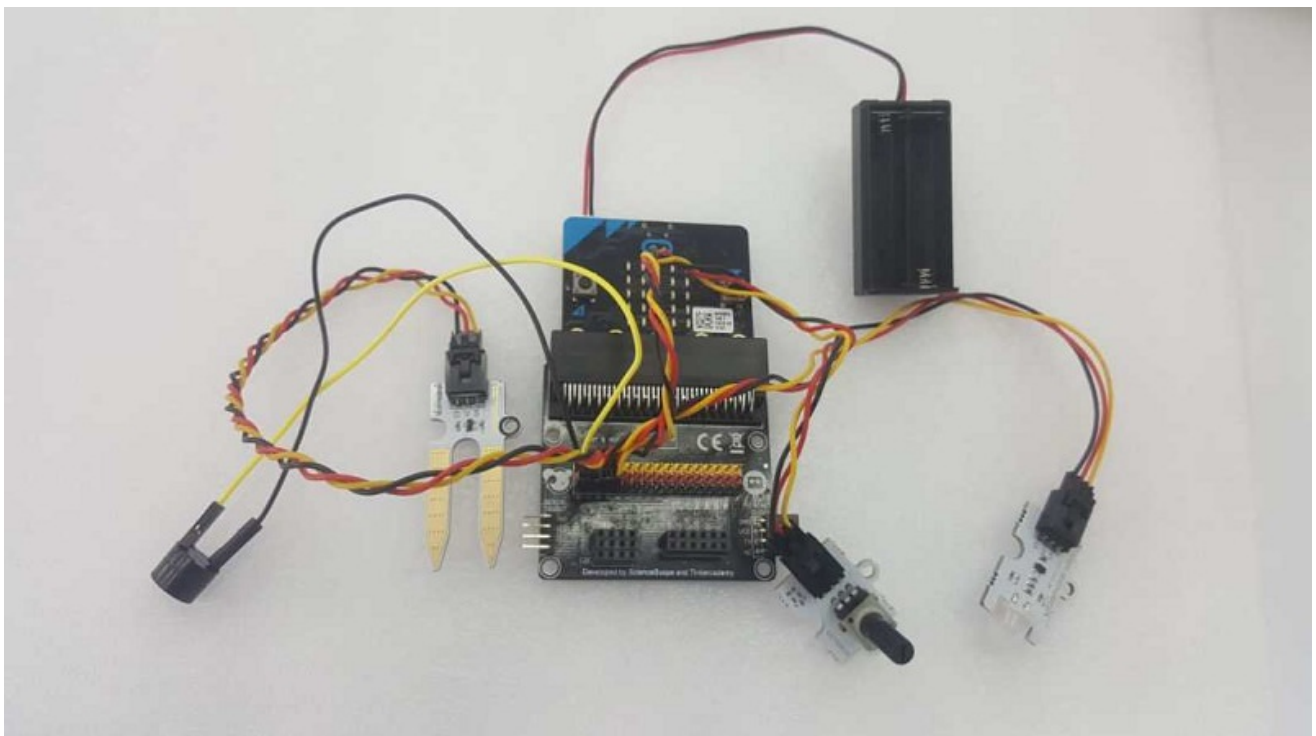
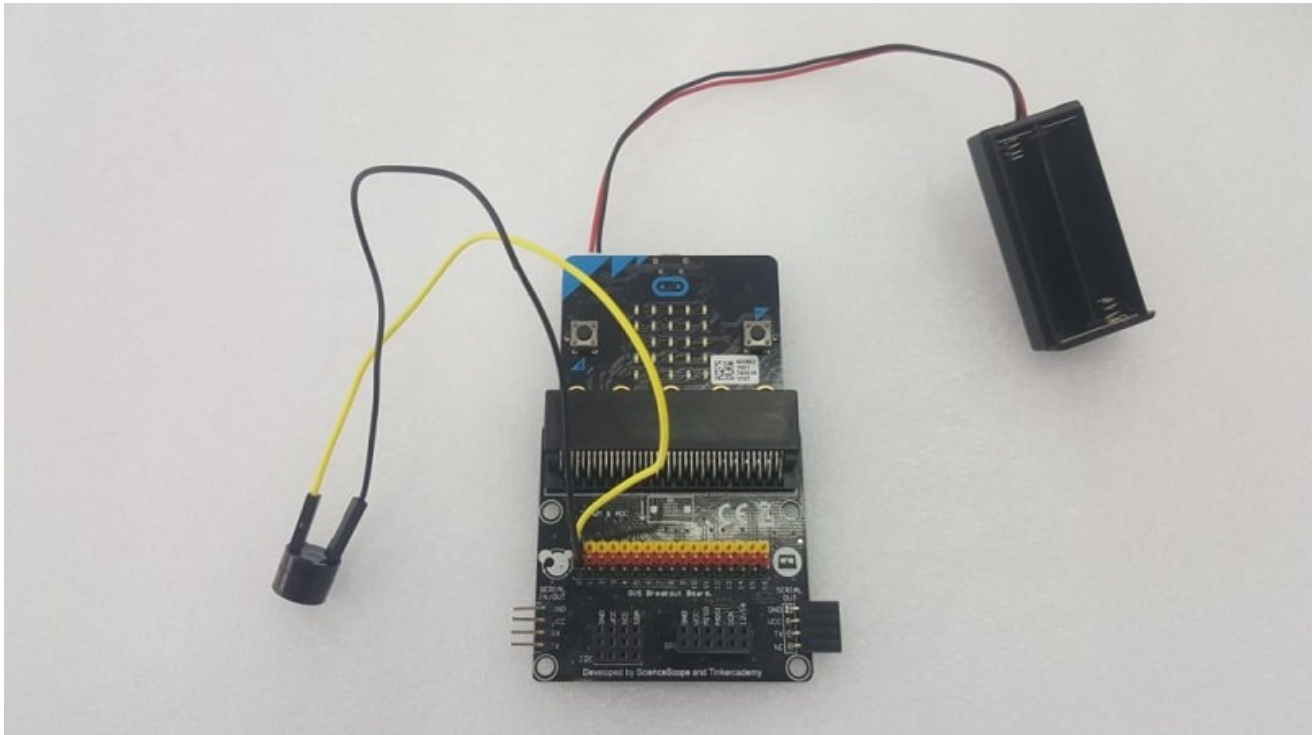


1. Assemble a shy reclusive robot
2. Input code to make a shy robot

Required Materials

- 1 x Micro:bit
- Batteries
- Any Box (Large enough to fit Micro:bit)
- PIR Sensor
- Potentiometer
- Passive Buzzer
- 2 x Female-Female Jumper Wires

Step 1 – Connect the Micro:bit parts



1. Attach the buzzer into pin 0 as shown. The + sign on your buzzer connects with the yellow port on the breaker board.
2. Attach the soil moisture sensor to pin 1.
3. Connect the PIR sensor to pin 2.
4. Connect the Potentiometer into pin 3.

Step 2: Add the Tinker Kit Package

Add Package... ?



Search or enter project URL...

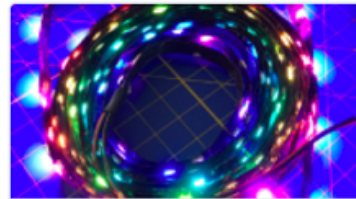


devices

Camera, remote control and other Bluetooth services

bluetooth

Bluetooth services



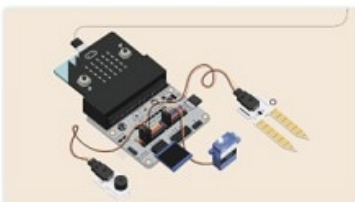
neopixel

AdaFruit NeoPixel driver

Add Package... ?



tinker kit

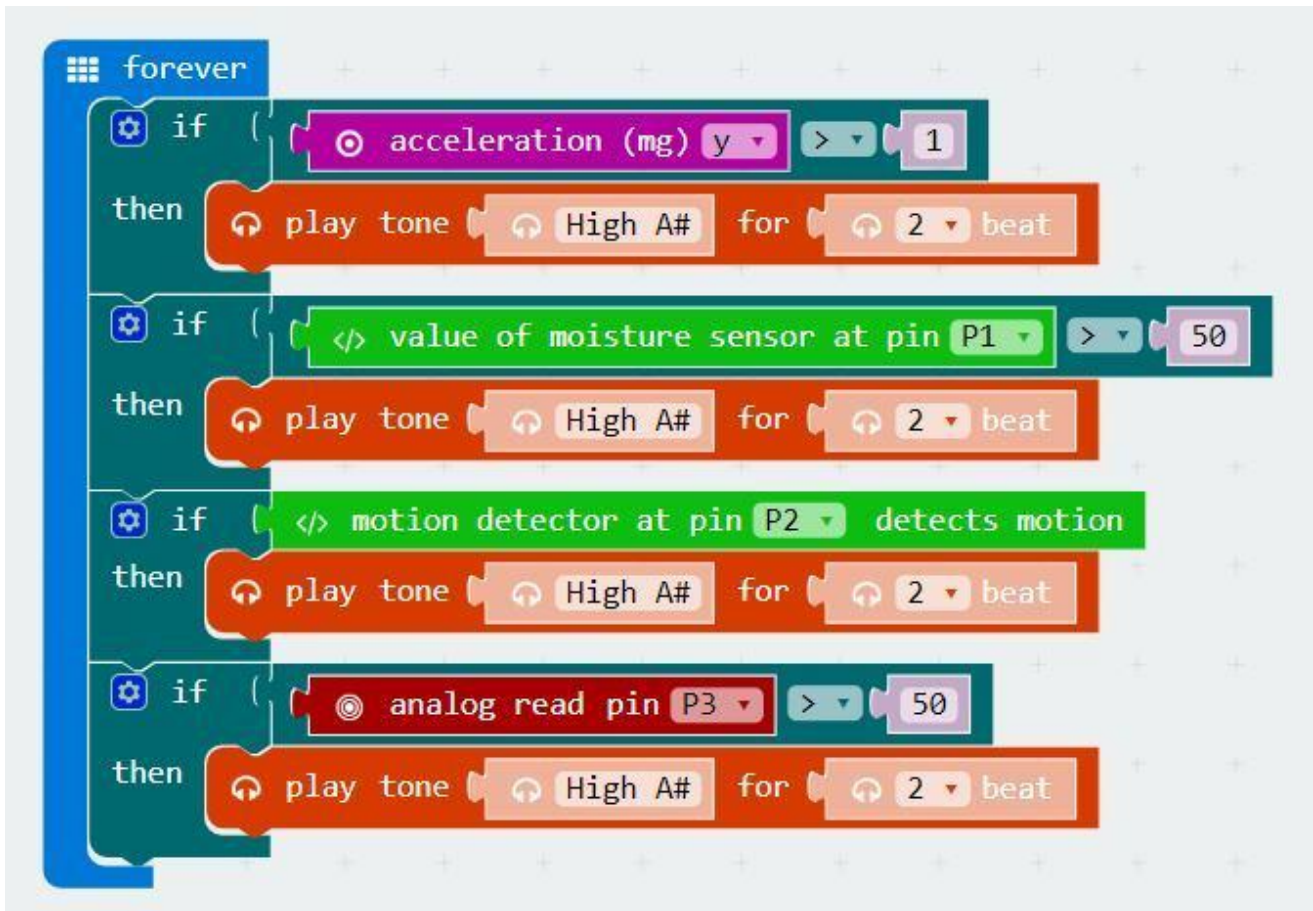


tinkercademy-tinker-kit

MakeCode package for modules in the Micro:bit Tinker Kit by ElecFreaks and Tinkercademy

1. We will need to add a package to the code editor to enable to use the kit components. Click on the advanced in the micro bit text editor and you will see a section that says Add Package.
2. This will open up a dialog box. Search for Tinker Kit. Click on the search icon or press enter, then select tinkercademy-tinker-kit.
3. This will add two libraries: Tinkercademy, for general-purpose sensors found in our kit, and OLED, for the OLED module (ours has a height of 64 and width of 128). We're not using the OLED module in this tutorial, but you can!

Step 3 – Start Coding!

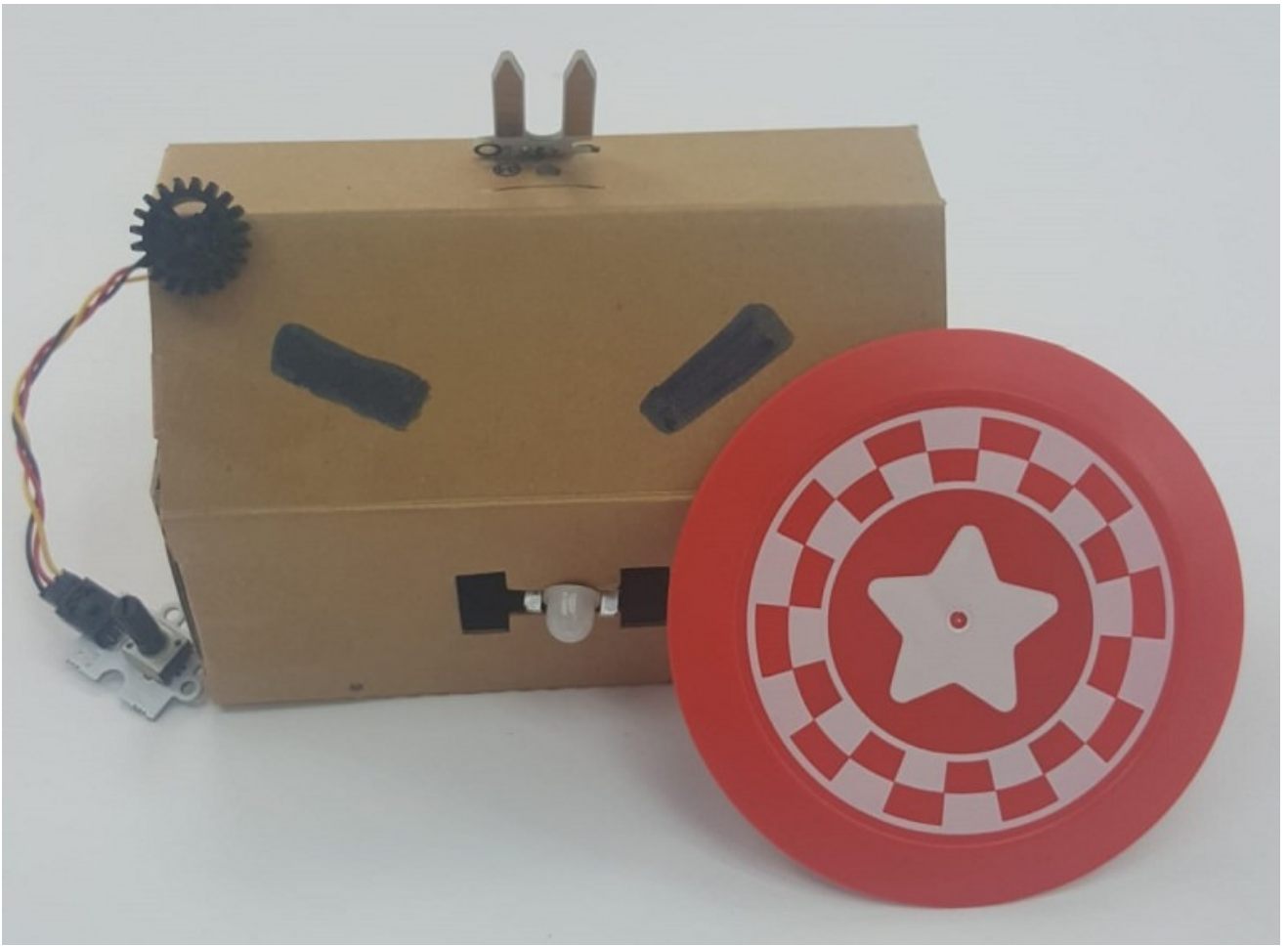


Put on the Micro:bit code- Coding your reclusive robot. Your program consists of a few “if” statements. We want the robot to only react to four conditions.

1. The first condition is the robot being picked up.
2. The second condition is someone touching the soil moisture sensors.
3. The third condition is if the robot detects movement in front of it.
4. And the last condition is someone toggling the potentiometer.

You're Done!

You have finished building all components needed to make a reclusive robot that makes a sound whenever someone surprises it! Place it into any box you large enough to contain the components while leaving some of them sticking out, and you have your very own reclus bot! Feel free to dress it up, but take care or it'll start squealing in surprise!

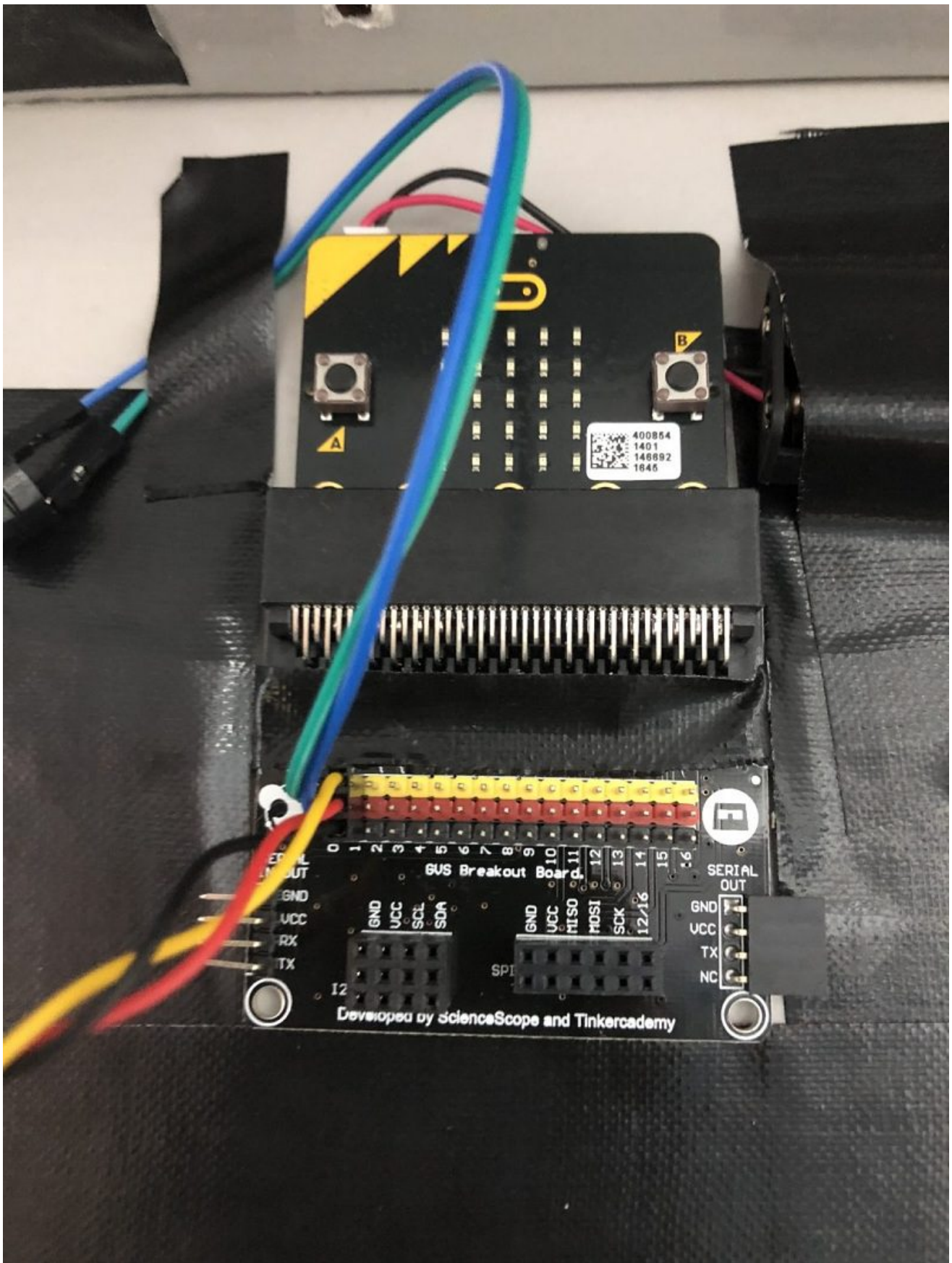


35. case 33 access denied

35.1. Access Denied! A Door Entry Tutorial

- Enhancing door security with micro:bit. This tutorial was written by Sean Lew, from the Singapore University of Technology and Design, during his summer internship with us in 2018.

Goals



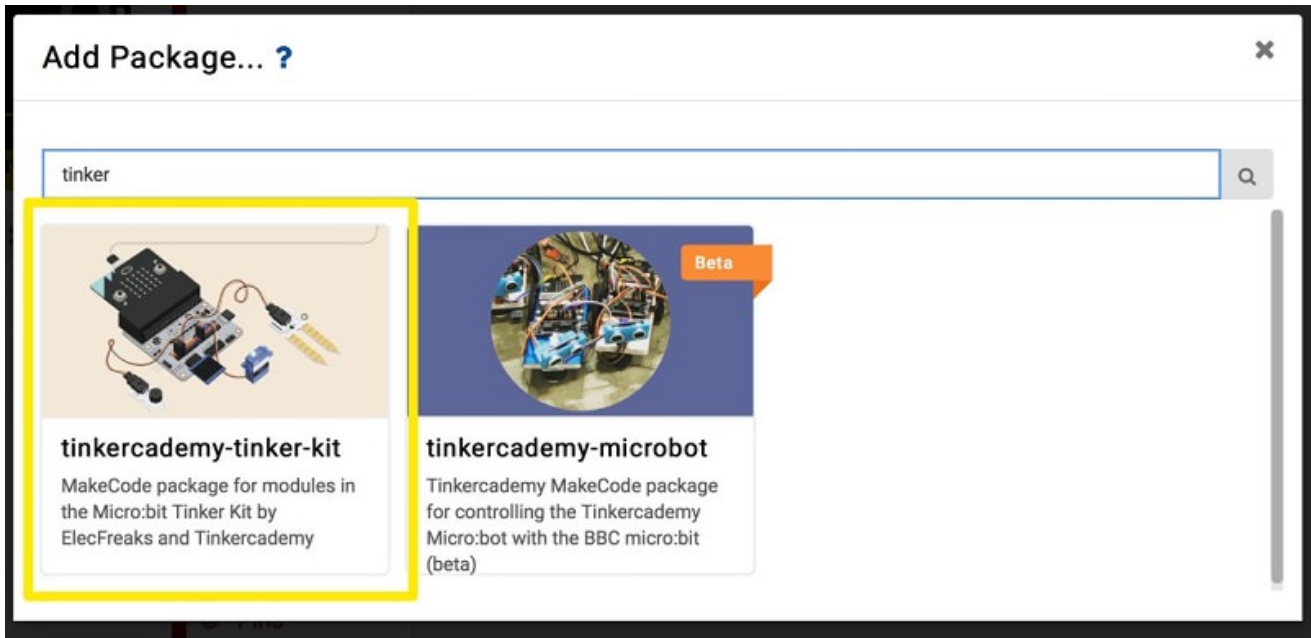
1. Build your own wireless door security!
2. Setting up alerts for any intruders or unlocked doors!

Required Materials

- 2 x micro:bit

- 1 x Breakout Board
- 1 x Crash Sensor
- 1 x Buzzer

Step 1 – Setting Up!



1. Slot one of the micro:bit into the breakout board.
2. Connect the Buzzer to Pin 0 of the breakout board and the crash sensor to Pin 1.
3. Before moving on to Step 2, make sure you download the “tinkercademy-tinker-kit” package.

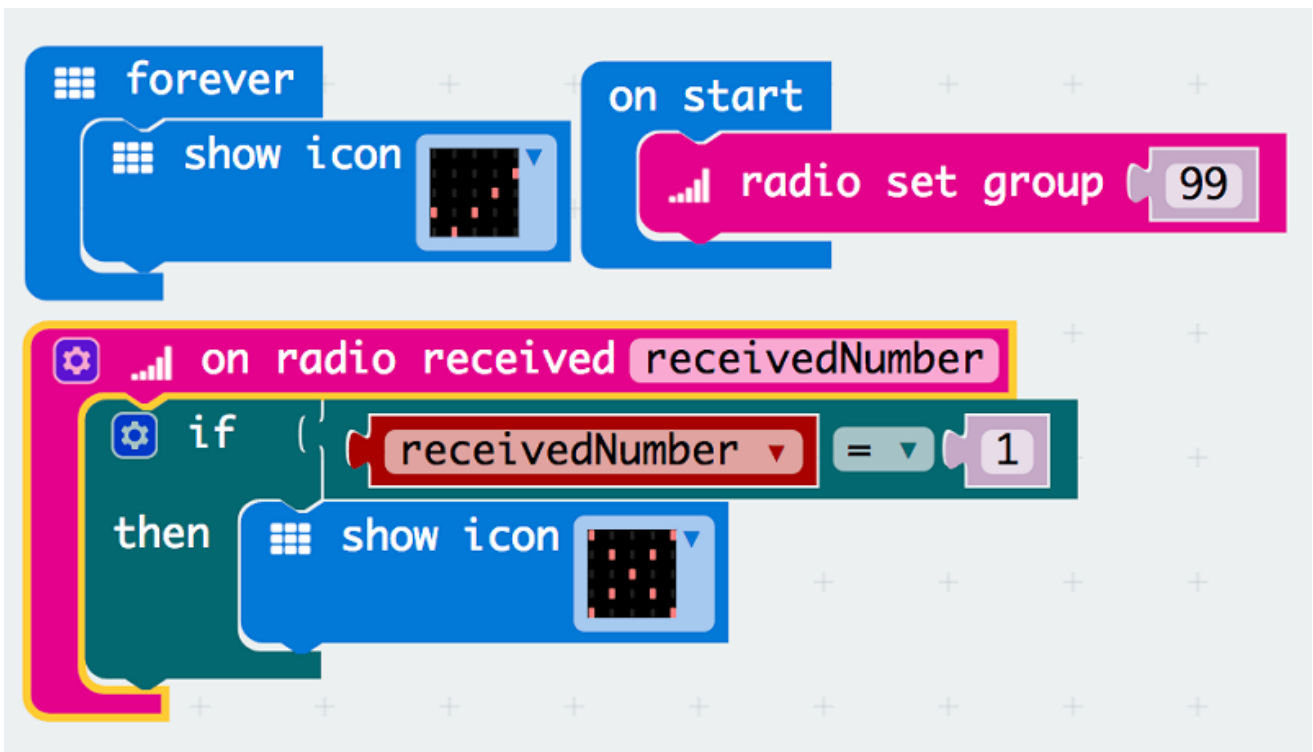
Step 2 – Code away!

```
on start
  Setup crash sensor at pin P1
  radio set group 99

forever
  radio send number 1
  show icon [3x3 grid]
  play tone High B for 1 beat
  while (crash sensor pressed)
    do
      show icon [3x3 grid]
```

1. Drag these set of codes into your makecode platform and download it into the micro:bit connected to your breakout board.
2. Give the crash sensor a few press to see if your code is working correctly!

Step 3 – Almost there!



1. Download this set of codes for the second micro:bit (receiver).
2. Now give test out the crash sensor to see if the receiving micro:bit is working correctly!
3. Now that you have set up your very first wireless door security, you can try it with more than just 1 door!
4. Feel free to also add add other form of alerts on the receiving micro:bit to enhance your own security.

36. case 34 micropython

36.1. Getting Started

Pre-coding:

- Get hold of a Micro:bit Tinker Kit
- Download the Mu editor [Mu editor](#)

36.2. Project 01: Music Machine



Pin Layout

- Buzzer: Pin0
- ADKeypad: Pin2

Small note about the ADKeypad

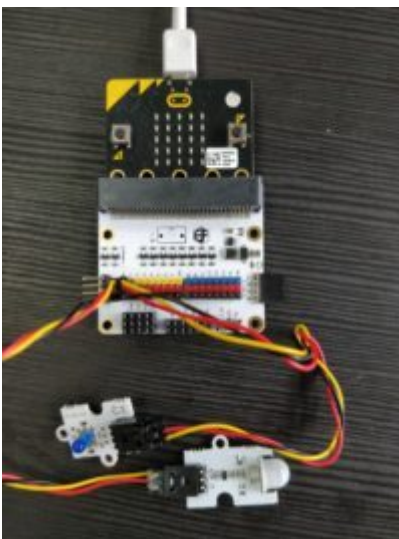
The ADKeypad returns an analog signal when its buttons are pressed. Each button pressed would return a unique integer value ranging from 0 (meaning 0V) to 1023 (meaning 3V). However, it is not uncommon that each button would give a small range of values when pressed at different times and different ADKeypads might give different signals yet again. Hence, in this example code, we provide a range of possible values that your ADKeypad's buttons are likely to return when pressed. Feel free to test out the values that your ADKeypad might return when pressed and change the values in the example code. ^ ^

```

1  from microbit import *
2  import music
3
4  #pins
5  ADKeyboard_pin = pin2
6  Buzzer_pin = pin0
7
8  while True :
9      #buttonA
10     if ADKeyboard_pin.read_analog( )>0 and ADKeyboard_pin.read_analog(<10:
11         music.play('f3:4', pin = Buzzer_pin)
12
13     #buttonB
14     if ADKeyboard_pin.read_analog(>45 and ADKeyboard_pin.read_analog(<55:
15         music.play('g3:4', pin = Buzzer_pin)
16
17     #buttonC
18     if ADKeyboard_pin.read_analog(>90 and ADKeyboard_pin.read_analog(<100:
19         music.play('a3:4', pin = Buzzer_pin)
20
21     #buttonD
22     if ADKeyboard_pin.read_analog(>136 and ADKeyboard_pin.read_analog(<139:
23         music.play('b3:4', pin = Buzzer_pin)
24
25     #buttonE
26     if ADKeyboard_pin.read_analog(>535 and ADKeyboard_pin.read_analog(<545:
27         music.play('c2:4', pin = Buzzer_pin)

```

36.3. Project 02: Smart Light



Pin Layout

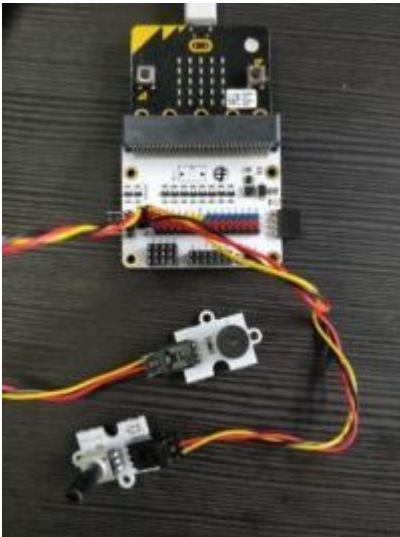
- PIR Sensor: Pin0
- LED: Pin1

```

1  from microbit import *
2
3  #pins
4  PIR_pin = pin0
5  LED_pin = pin1
6
7  while True:
8      # if PIR Sensor detects motion, turn on LED
9      if PIR_pin.read_digital():
10         LED_pin.write_digital(1)
11     else:
12         LED_pin.write_digital(0)

```

36.4. Project 03: Electro-Theremin



Pin Layout

- Buzzer: Pin0
- Potentiometer: Pin1

```
1  from microbit import *
2  import music
3
4  #pins
5  Potentiometer_pin = pin1
6  Buzzer_pin = pin2
7
8  # values for mapping
9  highest_p_note = 1023
10 lowest_p_note = 1
11 highest_note = 988
12 lowest_note = 131
13
14 potentiometer_note = 0
15 modified_note = lowest_note
16
17 #modify the note
18 def modify_note(p_value): # p is potentiometer
19     new_note = (p_value-lowest_p_note)/(highest_p_note-lowest_p_note)*(highest_note-lowest_note)+ lowest_note
20     return int(new_note)
21
22 while True:
23     potentiometer_note = Potentiometer_pin.read_analog()
24     display.show(potentiometer_note)
25     modified_note = modify_note(potentiometer_note)
26     music.pitch(modified_note, pin = Buzzer_pin)
```

36.5. Project 04: Simple Alarm Box



Pin Layout

- Crash Sensor: Pin0
- LED: Pin8
- OLED: I2C row (at the bottom of the BoB)

```

1 from microbit import *
2 import time
3 import math
4
5 # Adapted from https://github.com/fizban99/microbit_ssd1306
6 OLED_ADDR = 0x3c
7 oled_screen = bytearray(' '\x40' + bytearray(512))
8
9 def oled_initialize():
10     for c in ([0xae], [0xa4], [0xd5, 0xf0], [0xa8, 0x3f], [0xd3, 0x00], [0 | 0x0], [0x8d, 0x14], [0x20, 0x00], [0x21, 0, 127], [0x22, 0,
11         i2c.write(OLED_ADDR, b'\x00' + bytearray(c))
12
13 def oled_set_pos(col=0, page=0):
14     i2c.write(OLED_ADDR, b'\x00' + bytearray([0xb0 | page]))
15     c1, c2 = col * 2 & 0x0f, col >> 3
16     i2c.write(OLED_ADDR, b'\x00' + bytearray([0x00 | c1]))
17     i2c.write(OLED_ADDR, b'\x00' + bytearray([0x10 | c2]))
18
19 def oled_clear_screen(c=0):
20     global oled_screen
21     oled_set_pos()
22     for i in range(1, 513):
23         oled_screen[i] = c
24     oled_draw_screen()
25
26 def oled_draw_screen():
27     global oled_screen
28     oled_set_pos()
29     i2c.write(OLED_ADDR, oled_screen)
30
31 def oled_add_text(x, y, text):
32     global oled_screen
33     for i in range(0, min(len(text), 12 - x)):
34         for c in range(0, 5):
35             col = 0
36             for r in range(1, 6):
37                 p = Image(text[i]).get_pixel(c, r - 1)
38                 col = col | (1 << r) if (p != 0) else col
39                 ind = x * 10 + y * 128 + i * 10 + c * 2 + 1
40                 oled_screen[ind], oled_screen[ind + 1] = col, col
41             oled_set_pos(x * 5, y)
42             ind0 = x * 10 + y * 128 + 1
43             i2c.write(OLED_ADDR, b'\x40' + oled_screen[ind0 : (ind+1)])
44
45 #allow overflow to go onto the next line
46 def oled_add_text_new_line(x, y, text):
47     length_text = len(text)
48     separated_text = []
49     counter = 0
50     num_of_lines = math.ceil(length_text/12)
51     letters_in_line = 12
52

```

```

53     for line in range(0,num_of_lines):
54         separated_text.append([])
55         #separated_text[line].append(y*(line+1))
56         for l in range(0,letters_in_line):
57             separated_text[line].append(text[letters_in_line*line+l])
58             counter +=1
59             if counter == length_text:
60                 break
61
62     #draw letters
63     for i in range(0,len(separated_text)):
64         oled_add_text(x,y+i,separated_text[i])
65
66
67     # Screen divided into 12 columns and 4 rows
68
69     oled_initialize()
70     oled_clear_screen()
71
72     # Start Simple Alarm Box Code here
73
74     #pins
75     CrashSensor_pin = pin0
76     LED_pin = pin8
77
78     #set up crash sensor
79     CrashSensor_pin.set_pull(CrashSensor_pin.PULL_UP)
80
81     #other variables
82     has_text = False
83
84     while True:
85         if CrashSensor_pin.read_digital() == 1:
86             if has_text == False : #checks if oled screen has the message already, if not add it
87                 oled_add_text_new_line(0, 0, "Your treasure is safe")
88                 has_text = True
89                 LED_pin.write_digital(1)
90
91         else:
92             #clear oled screen
93             oled_clear_screen()
94             has_text = False
95
96             #make LED blink
97             LED_pin.write_digital(0)
98             time.sleep(0.1)
99             LED_pin.write_digital(1)
100            time.sleep(0.1)

```

36.6. Project 05: Plant Monitoring Device



Pin Layout

- Buzzer: Pin0
- Soil Moisture Sensor: Pin1

- OLED: I2C row (at the bottom of the BoB)

```

1  from microbit import *
2  import time
3  import math
4  import music
5
6  # Adapted from https://github.com/fizban99/microbit_ssd1306
7  OLED_ADDR = 0x3c
8  oled_screen = bytearray('b\x40') + bytearray(512)
9
10 def oled_initialize():
11     for c in ([0xae], [0xa4], [0xd5, 0xf0], [0xa8, 0x3f], [0xd3, 0x00], [0 | 0x0], [0x8d, 0x14], [0x20, 0x00], [0x21, 0, 127], [0x22,
12         i2c.write(OLED_ADDR, b'\x00' + bytearray(c))
13
14 def oled_set_pos(col=0, page=0):
15     i2c.write(OLED_ADDR, b'\x00' + bytearray([0xb0 | page]))
16     c1, c2 = col * 2 & 0x0f, col >> 3
17     i2c.write(OLED_ADDR, b'\x00' + bytearray([0x00 | c1]))
18     i2c.write(OLED_ADDR, b'\x00' + bytearray([0x10 | c2]))
19
20 def oled_clear_screen(c=0):
21     global oled_screen
22     oled_set_pos()
23     for i in range(1, 513):
24         oled_screen[i] = 0
25     oled_draw_screen()
26
27 def oled_draw_screen():
28     global oled_screen
29     oled_set_pos()
30     i2c.write(OLED_ADDR, oled_screen)
31
32 def oled_add_text(x, y, text):
33     global oled_screen
34     for i in range(0, min(len(text), 12 - x)):
35         for c in range(0, 5):
36             col = 0
37             for r in range(1, 6):
38                 p = Image(text[i]).get_pixel(c, r - 1)
39                 col = col | (1 << r) if (p != 0) else col
40                 ind = x * 10 + y * 128 + i * 10 + c * 2 + 1
41                 oled_screen[ind], oled_screen[ind + 1] = col, col
42             oled_set_pos(x * 5, y)
43             ind0 = x * 10 + y * 128 + 1
44             i2c.write(OLED_ADDR, b'\x40' + oled_screen[ind0 : (ind+1)])
45
46 #allow overflow to go onto the next line
47 def oled_add_text_new_line(x, y, text):
48     length_text = len(text)
49     separated_text = []
50     counter = 0
51     num_of_lines = math.ceil(length_text/12)
52     letters_in_line = 12

```

```

53
54     for line in range(0,num_of_lines):
55         separated_text.append([])
56         #separated_text[line].append(y*(line+1))
57         for l in range(0,letters_in_line):
58             separated_text[line].append(text[letters_in_line*line+l])
59             counter +=1
60             if counter == length_text:
61                 break
62
63     #draw letters
64     for i in range(0,len(separated_text)):
65         oled_add_text(x,y+i,separated_text[i])
66
67
68     # Screen divided into 12 columns and 4 rows
69
70     oled_initialize()
71     oled_clear_screen()
72
73     # Start Plant Monitoring Device Code here
74
75     #pins
76     Buzzer_pin = pin0
77     MoistureSensor_pin = pin1
78     Servo_pin = pin8
79
80     #other variables
81     healthWarning = False
82     oled_add_text_new_line(0, 0, "Your plant is in good condition")
83
84     while True:
85         if MoistureSensor_pin.read_analog() <50:
86             if healthWarning == False : #checks if oled screen has the message already, if not add it
87                 oled_clear_screen()
88                 oled_add_text_new_line(0, 0, "Moisture level is: %d" % MoistureSensor_pin.read_analog())
89                 oled_add_text_new_line(0, 2, "Water your plant!")
90                 healthWarning = True
91                 music.play('b3:1', pin = Buzzer_pin)
92
93         else:
94             #clear oled screen
95             if healthWarning == True:
96                 oled_clear_screen()
97                 healthWarning = False
98                 oled_add_text_new_line(0, 0, "Your plant is in good condition")

```

Get Creative!

Mix and match the component in the Tinker Kit to create your own projects.

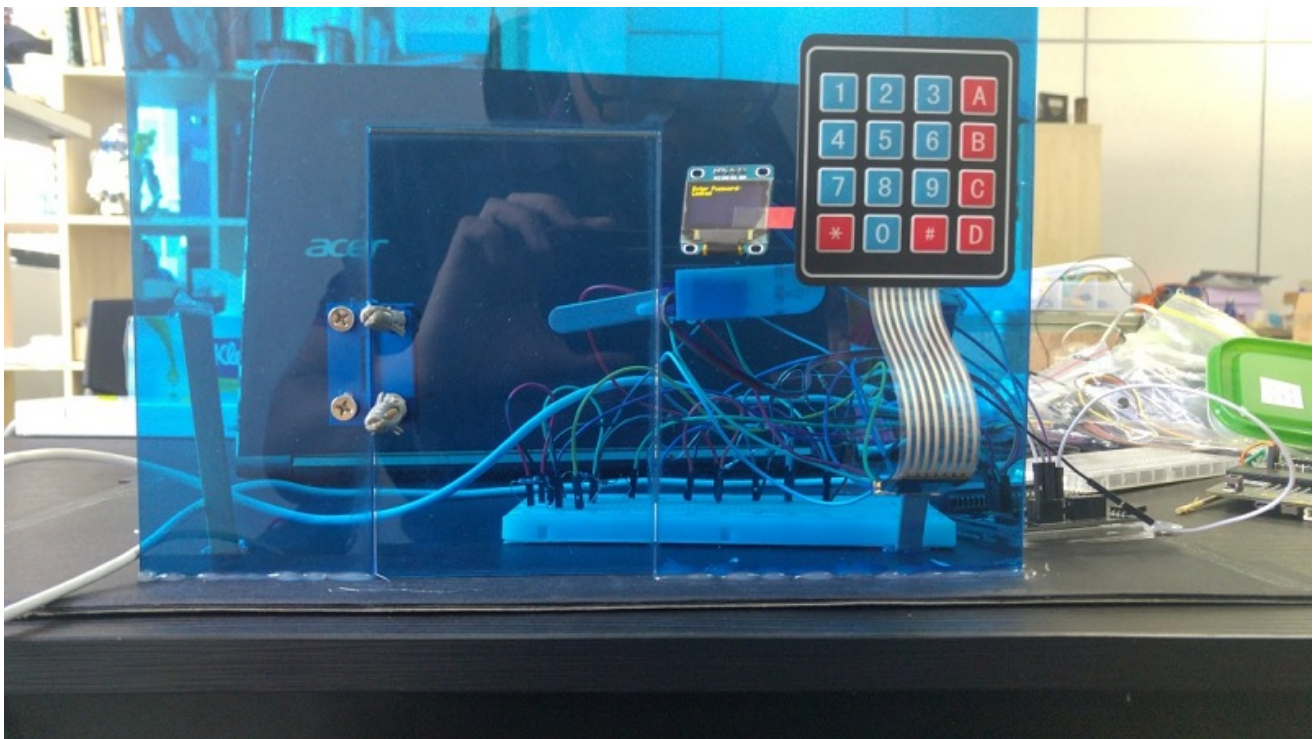
For a more comprehensive explanation of MicroPython, visit the official documentation [here](#)

37. case 35 build your own microbit security Door

37.1. Build your own Micro:bit Security Door!

- Protect your house or valuables with a micro:bit, a servo motor, and a 4x4 keypad!
Created by Mohd Shafiq from NUS.

Goals



1. Connect the wiring to interface the keypad with the microbit
2. Set your own unique password for the lock
3. Add a lock down counter in case an intruder tries to guess your password

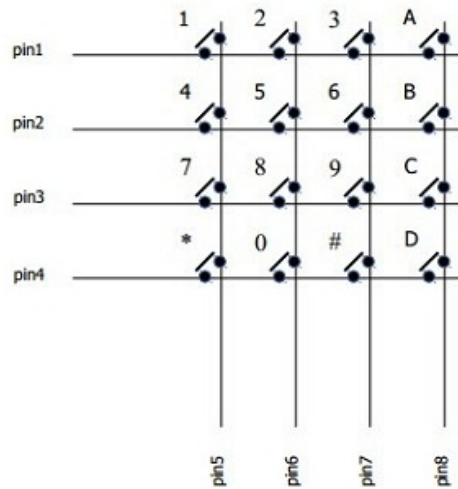
Required Materials

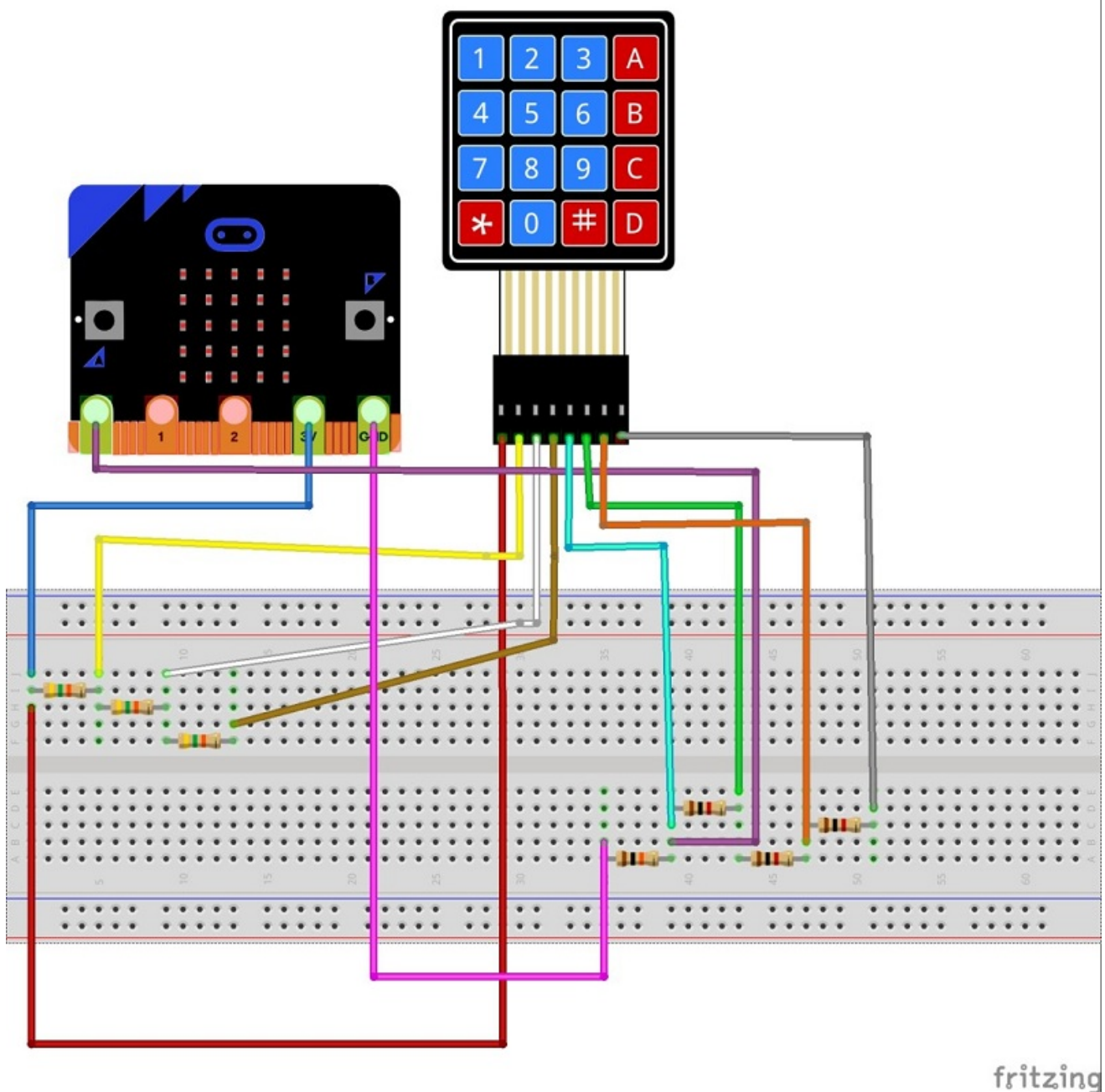
- 1 x Breakout board
- 1 x micro:bit
- 1 x Mini Servo

- 1 x Breadboard
- 4 x 4 Membrane keypad
- 3 x 5kohm resistor
- 3 x 1kohm resistor
- 1 x 10kohm resistor
- Jumper Wires
- 1 x OLED
- 18.5cm x28cm Acrylic
- 1 x small Metal Hinge

Step 1 – Interfacing the Keypad

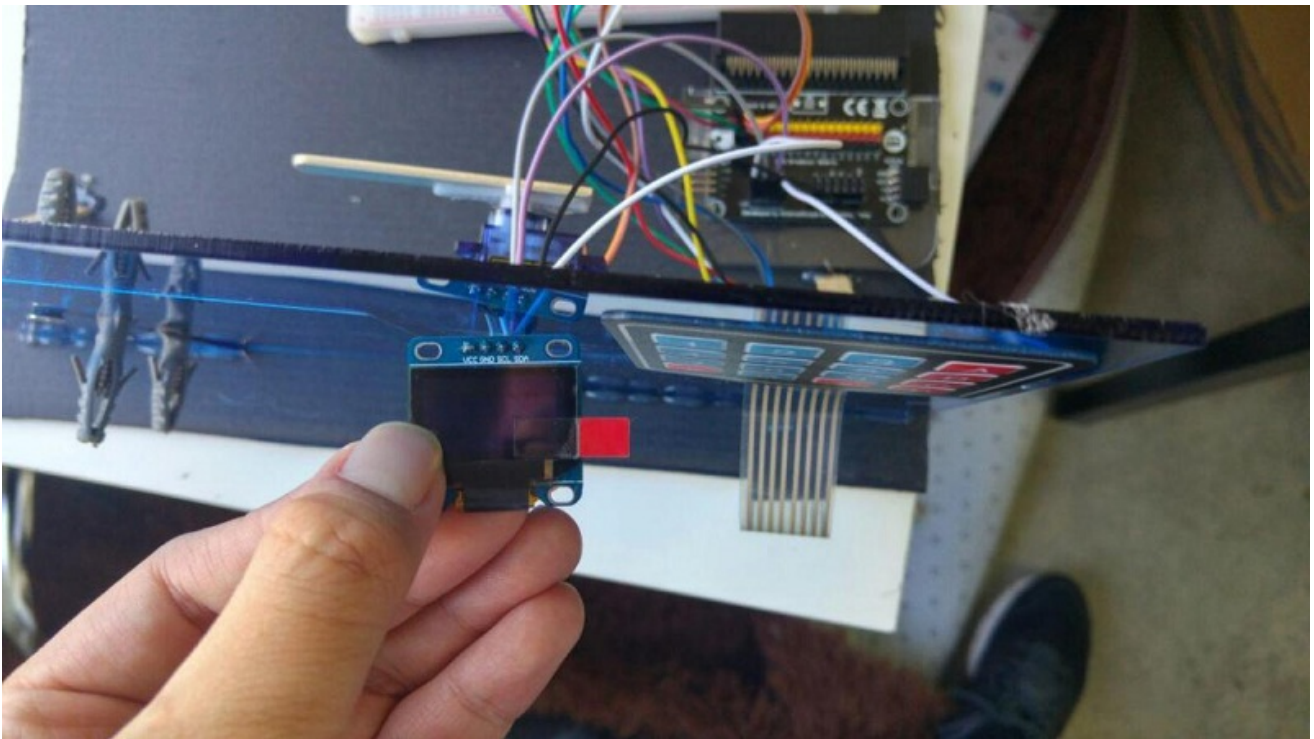
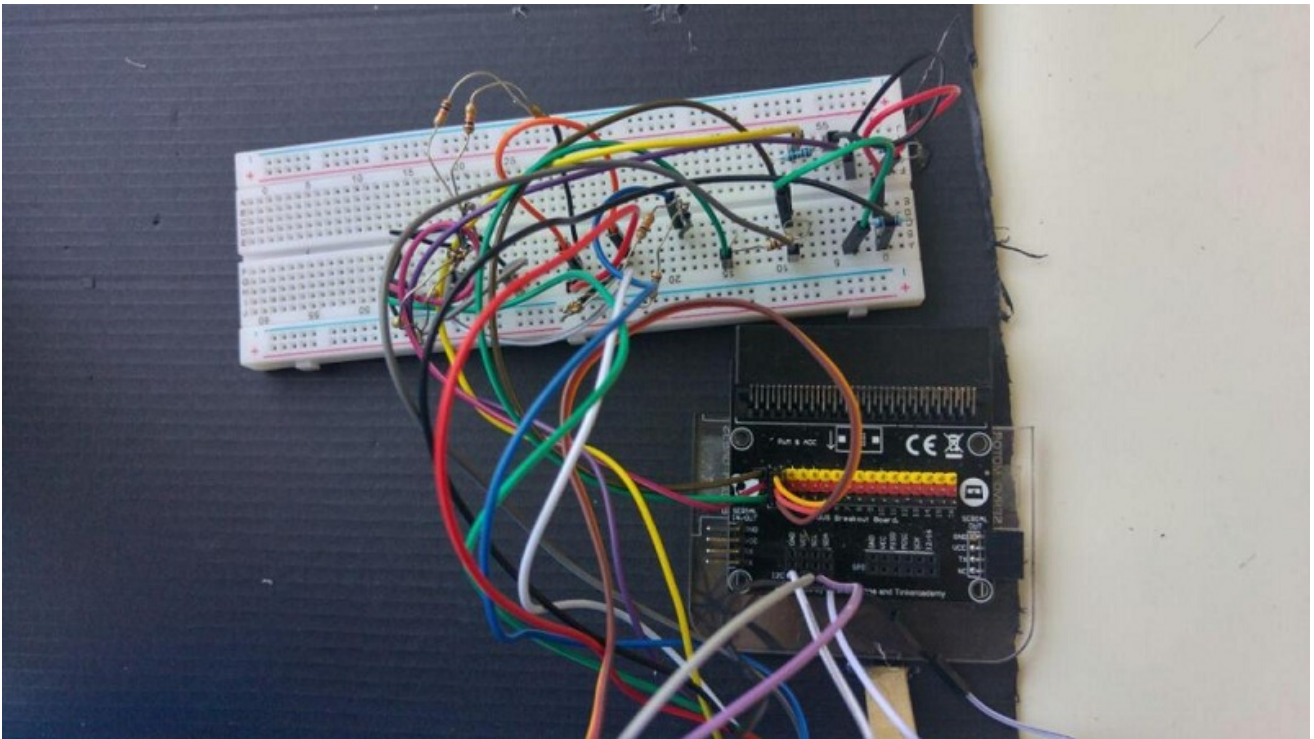
KEY4X4M01





1. The membrane keypad has 16 different switches which have 16 different characters
2. The switches are grouped together by row and column for example (R1,C1) corresponds to 1 and so on. When the controller detects a 1 at Pin 1 and Pin 5 it would mean key 1 is pressed. This is called digital output
3. Therefore we can use the Microbit's digital pins to interface with the keypad. However if we do that it will be quite messy.
4. Let's create a driver circuit for the 4x4 keypad!
5. Follow the diagram as shown:

Step 2 – Wire it Up



1. Attach the 3 pins of the Servo motor to P2 of the breakout board
2. Attach Ground (Black pin) of the Micro:bit to the 10kOh resistor
3. Attach the 3V (Red pin) to the 5k Ohm resistor
4. Attach A0 (Yellow pin) to the point between the 10k Ohm Resistor and 1k Ohm resistor

Connect GND,VCC,SCI and SDA of the breakout board to GND,VCC,SCI and SDA of the OLED respectively.

```
on start
  initialize OLED with height 64 width 128
  show string "Enter Password:"
  show string "Locked"
  servo write pin P1 to 180
  set N to 0
  set LockdownCounter to 61
  set KeyArray to ( create array with ( " "
  ( " "
  ( " "
  ( " "
```

Step 3 – Coding the micro:bit

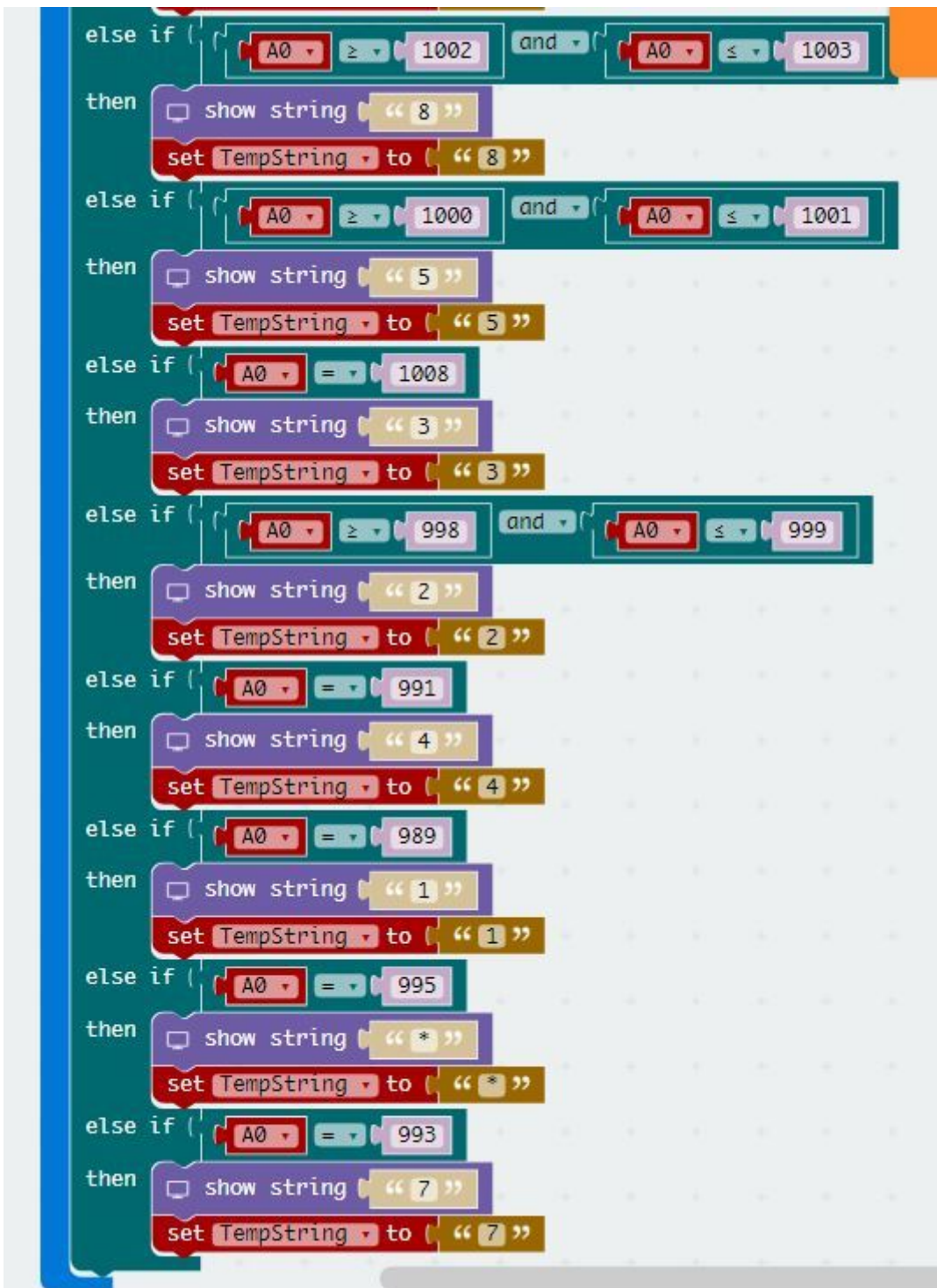
On start we have to:

- Initialize the servo to position 180 (Locked Position)
- Initialize the OLED display
- Initialize a 4x1 array
- Initialize the lockdown counter

```
on button A pressed
  change N by 1
  if (N = 1)
    then KeyArray set value at 0 to TempString
  else if (N = 2)
    then KeyArray set value at 1 to TempString
  else if (N = 3)
    then KeyArray set value at 2 to TempString
  else if (N = 4)
    then KeyArray set value at 3 to TempString
```

- Button A acts as the enter character button
- Every time you press a key on the 4x4 keypad you have to press Button A in order to key in a 4 digit Number


```
forever
  set A0 to (analog read pin P0)
  if (A0 >= 1023)
  then
    show string "D"
    set TempString to "D"
  else if (A0 = 1021)
  then
    show string "C"
    set TempString to "C"
  else if (A0 = 1019)
  then
    show string "B"
    set TempString to "B"
  else if (A0 = 1017)
  then
    show string "A"
    set TempString to "A"
  else if (A0 = 1014)
  then
    show string "#"
    set TempString to "#"
  else if (A0 = 1012)
  then
    show string "9"
    set TempString to "9"
  else if (A0 = 1010)
  then
    show string "6"
    set TempString to "6"
  else if (A0 = 1004)
  then
    show string "0"
    set TempString to "0"
```



```
else if (A0 >= 1002 and A0 <= 1003)
then
  show string "8"
  set TempString to "8"
else if (A0 >= 1000 and A0 <= 1001)
then
  show string "5"
  set TempString to "5"
else if (A0 = 1008)
then
  show string "3"
  set TempString to "3"
else if (A0 >= 998 and A0 <= 999)
then
  show string "2"
  set TempString to "2"
else if (A0 = 991)
then
  show string "4"
  set TempString to "4"
else if (A0 = 989)
then
  show string "1"
  set TempString to "1"
else if (A0 = 995)
then
  show string "*"
  set TempString to "*"
else if (A0 = 993)
then
  show string "7"
  set TempString to "7"
```

Now we need to set up the micro:bit to detect the key presses!

- Each key press corresponds to a unique analog value from 0 to 1023 by using the driver circuit
- The analog value can be read using the analog read function
- The value in the character is stored in the "TempString" variable
- The code block is quite long, so the download link is provided below

```

on button B pressed
  set FirstNumber to KeyArray get value at 0
  set SecondNumber to KeyArray get value at 1
  set ThirdNumber to KeyArray get value at 2
  set FourthNumber to KeyArray get value at 3
  set Final Key to join FirstNumber SecondNumber ThirdNumber FourthNumber
  set Result to compare Final Key to "369#"
  show string Final Key
  if Result = 0
  then
    show string "Unlocked"
    servo write pin P1 to 0
  else
    show string "Wrong Password!"
    change Attempts by 1
  if Attempts = 3
  then
    show string "Initiating Lockdown"
    for index from 0 to 60
    do
      set LockdownCounter to LockdownCounter - 1
      show number LockdownCounter

```

- Button B acts as the final “Enter” button
- Pressing button B causes the program to check if the entered string is equal to “369#” using the compare block
- If the answer is 0 it means that the strings are equal. If it is 1, then the strings are not equal
- The number of wrong Attempts will increase by 1 every time an intruder enters the wrong password
- Once 3 wrong attempts are detected, the program will enter a loop for 60 seconds

```

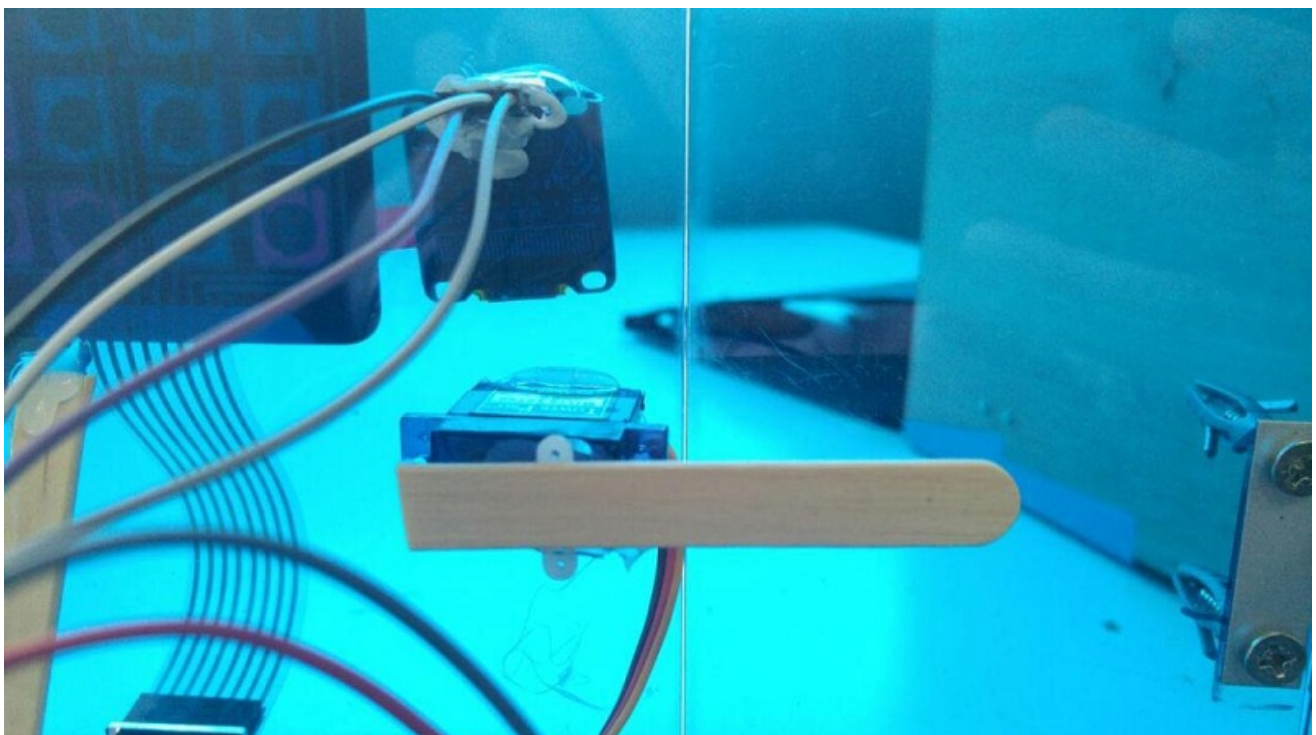
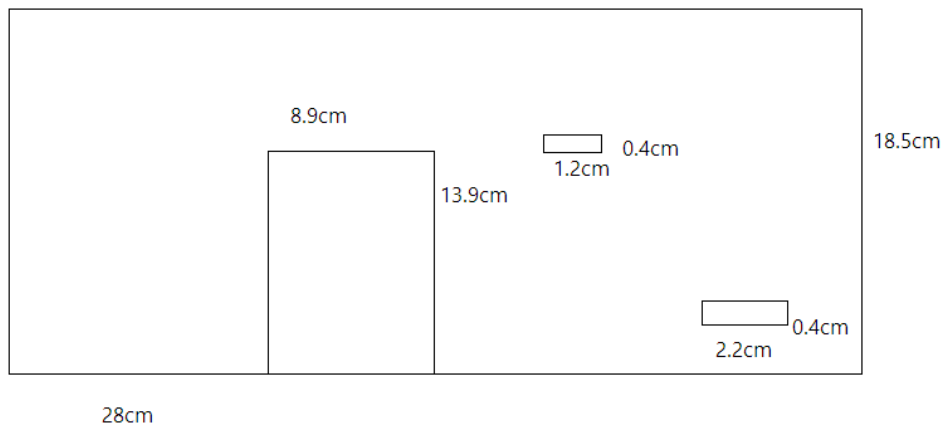
on button A+B pressed
  set N to 0
  set Attempts to 0
  set LockdownCounter to 61
  servo write pin P1 to 180

```

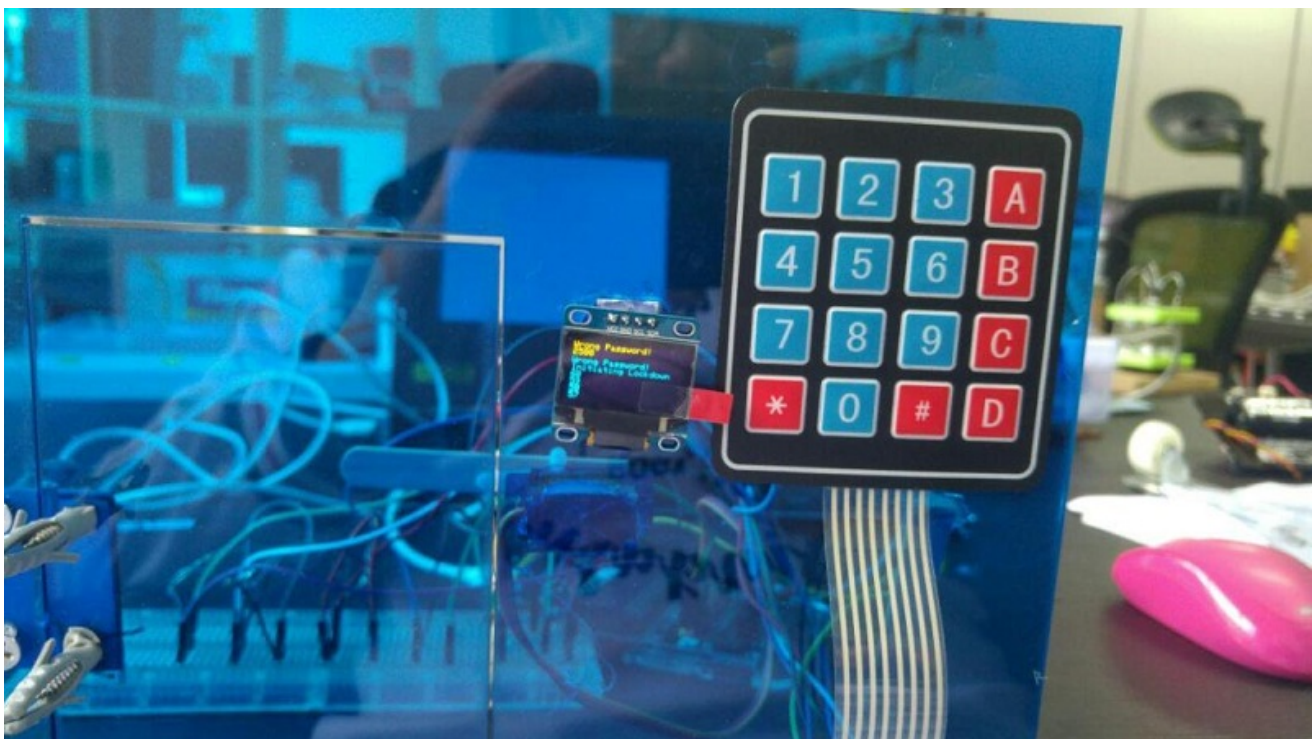
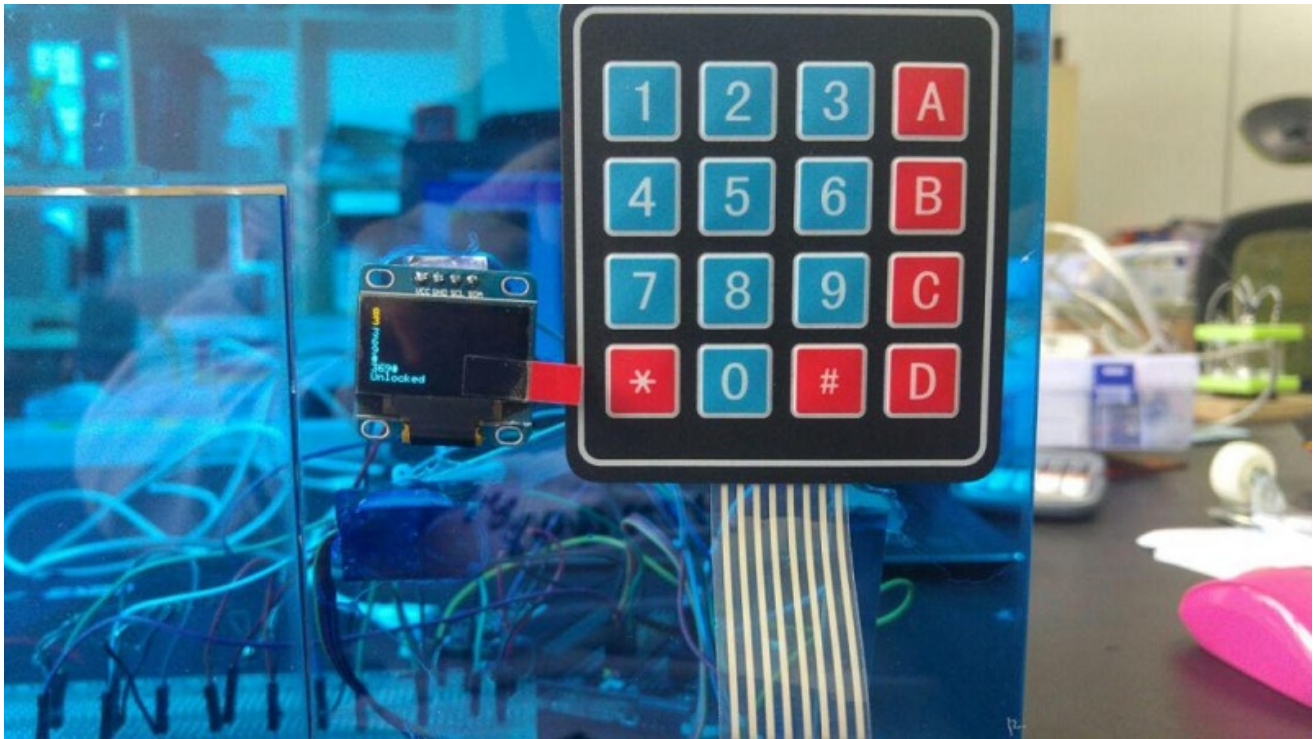
- To reset the number of attempts and the characters entered,press Button A+B
- The reset button also resets the servo to the “Locked Position”

Step 4 – Build it!

1. Using a laser cutter cut out 18.5cm by 28cm of 3mm acrylic
2. If you do not have acrylic you can use cardboard instead
3. Cut out slots for the OLED as well as the keypad wires
4. Drill/poke holes forthe door hinge and use screws to fasten the door
5. Attach an ice-cream stick to the servo.This will serve as the lock
6. Glue the servo with the ice-cream stick on the other side of the door



Step 5 – Demo Time!



1. Each time you press a key, the corresponding character will appear on the OLED
2. To unlock the door key in 3, Button A, 6, Button A, 9, Button A, #
3. Then press button B
4. To reset press A+B
5. If you try to enter the wrong password three times the OLED will display a lockdown timer. You will only be able to enter the password after 60 seconds have passed.
6. Congratulations! You have made your own micro:bit door.