Brain Pad™

BEGINNERS' GUIDE

CONTENTS

INTRODUCTION

The BrainPad is designed as a powerful educational STEM tool that can be used to teach everyone, from kids to college students and professionals.

This free e-book is provided as a beginner's guide to the world of the BrainPad. More materials can be found on https://www.brainpad.com.

THE AUTHOR

Gus Issa is founder and CEO of GHI Electronics. He started tinkering with programming and electronics as a teenager. Early on, he struggled to educate himself with limited resources and no Internet access. This struggle, along with the profound impact education has had on his life, have made him passionate about teaching others. He has spent countless hours of his own time while leveraging the resources of his company to create an affordable way to share his knowledge, resulting in the BrainPad.

GLOBAL THINKING

We work hard on making the BrainPad available to every culture and language. If you feel up to the challenge and want to translate this book to your language, please let us know through https://www.brainpad.com/contact-us.

LICENSE

This book is free and licensed under CC BY-SA 4.0. You are free to edit and print, repurpose and reuse. Learn more about what you can and can't do here https://creativecommons.org/licenses/by-sa/4.0/.

THE CONCEPT

The BrainPad is self-explanatory. It logically shows how computers work. There are four inputs that are routed to the brain. The brain (the processor) then thinks about what is happening to control the four outputs.
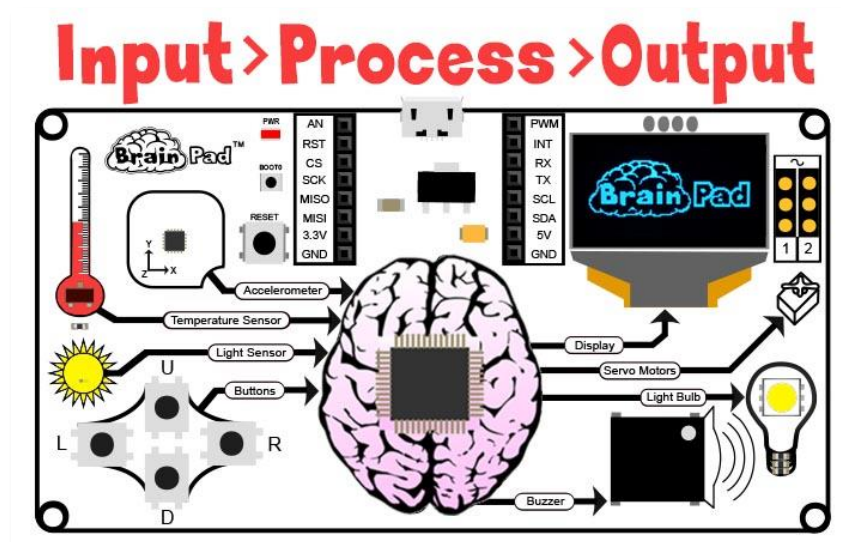
STEM

"STEM" stands for Science, Technology, Engineering, and Mathematics. The STEM acronym was introduced in 2001 by the U.S. National Science Foundation. STEM is now one of the most talked about topics in education. STEM education, however, is more than just these four fields of study. The STEM educational approach is aimed at connecting classroom learning to the real world by emphasizing communication, collaboration, critical thinking, and creativity while teaching the engineering design process. The term "STEAM" is the same as STEM with additional emphasis on the arts.

THE PHILOSOPHY

STEM education requires an evolving platform, not a toy. The BrainPad evolves to match your skill level. Many toys are marketed as STEM tools, but most lack the versatility to keep students engaged for any length of time. The BrainPad consolidates programming (from beginner to professional), electronics, engineering, robotics, and more into an inexpensive platform that can be used from grade school through college. It is backed by our 15 years of professional engineering experience. The same tools commercial customers use to program our industrial controllers can also be used to program the BrainPad. No other STEM platform can lead students from drag and drop block programming to professional programming and engineering like the BrainPad.

TWO PATHS

The BrainPad provides two paths for learning programming called **Start Making** and **Go Beyond**. As you may have guessed, the first option makes it very easy to "Start Making" projects and the second option takes you beyond and lets you use the same programming tools that are used by professionals.

START MAKING

With the Start Making option there is nothing to install on your computer -- everything works through your Internet browser using Microsoft MakeCode. The Go Beyond option requires some setup and software installation. The most popular option to go beyond MakeCode is Microsoft Visual Studio.

With Microsoft MakeCode (Start Making), you can program by simply arranging blocks and optionally by typing JavaScript code right into your web browser. Here is a program that sets the LightBulb to blue and then to red every second, and it does this "forever".
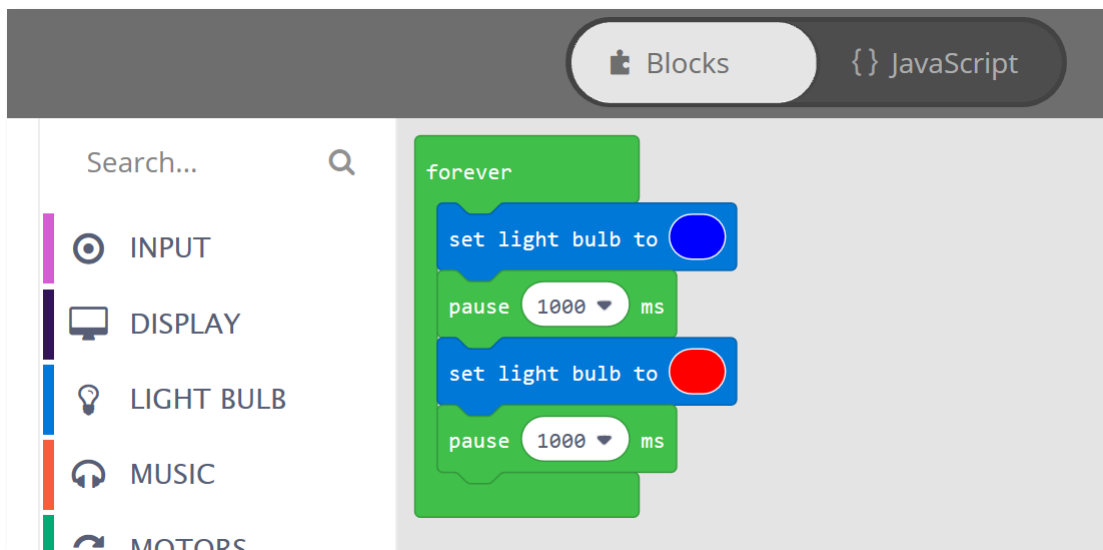
The included simulator (shown below) is basically a virtual BrainPad that runs your programs right on your computer screen. You can see it at the left side of the Microsoft MakeCode Window. The simulator allows you to try programs right in your browser without loading the program onto the BrainPad. This is also an excellent option if you haven't received your BrainPad yet.
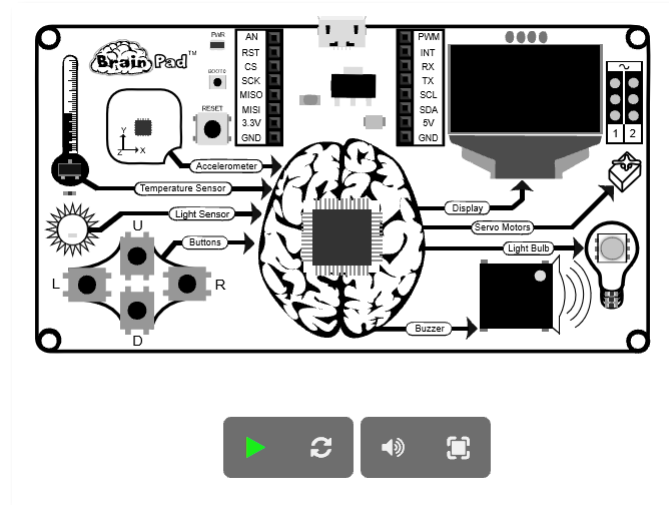
GO BEYOND

Things get more serious with the Go Beyond option. Here you will setup a computer with Microsoft Visual Studio to code and debug programs running on the BrainPad.

Thanks to GHI Electronics TinyCLR OS™, the BrainPad can be .NET programmed in C# and Visual Basic. Full debugging capabilities are included as well, such as stepping through code, breakpoints and inspecting variables at runtime. You will be programming the BrainPad the same way professionals program a PC or a phone app. This book will introduce you to both paths.

GOING BEYOND THE BEYOND!

The BrainPad can also be programmed with other options that are not documented in this book. For example, the Arduino tools https://www.arduino.cc, MicroPython https://www.micropython.org and Arm Mbed https://www.mbed.com.

START MAKING

Enough talking – let's start having fun and make some projects! We will be using Microsoft MakeCode so head on to https://makecode.brainpad.com/. The website includes a lot of material and projects, but we will just dig right into starting our own project. Go ahead and click the New Project button.
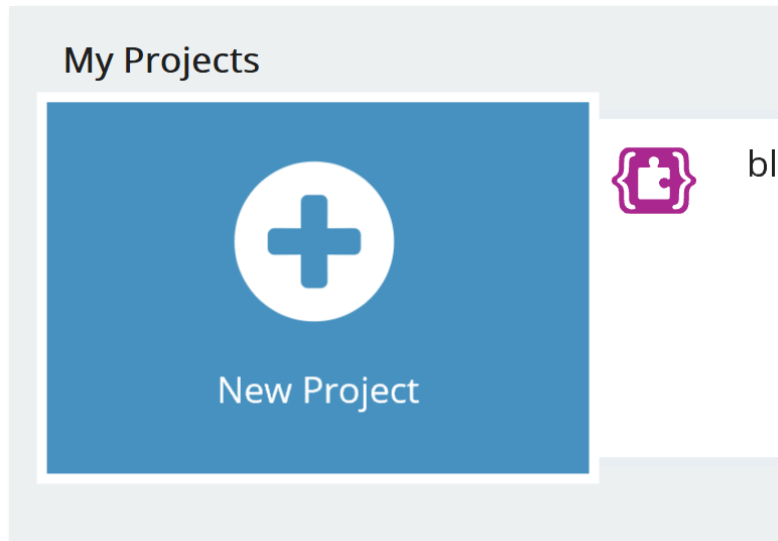
Now, from under the LIGHT BULB option in the menu, drag the block "set light bulb to" and place it into the "forever" block. The forever block is used for any code that you want running all the time. The instructions within the forever block will always execute while the BrainPad has power. Once all the instructions have executed, the BrainPad runs the code again starting with the first instruction in the forever block. This is known as an infinite loop and is used often in computer programming.

Repeat the same step but in the second block click on the color oval to change the color. This is what you should have so far:
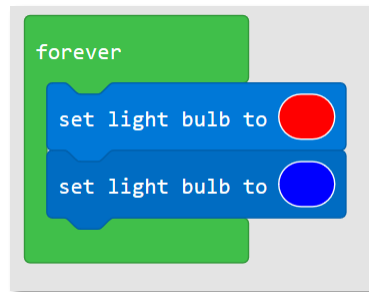
The program we have created will set the Light Bulb to red and then change it to blue repeatedly, forever! However, if you look at the simulator, the Light Bulb may not work as expected. If you load the program on the BrainPad itself, it will also not work properly, but why? You simply did not tell it how long to wait before changing the Light Bulb color. It is now changing color too quickly to see the individual colors.

From under the LOOPS menu, drag the "pause" block.

We will need a pause after the Light Bulb turn red and one after it turns blue. Let's also change the pause time to one second.
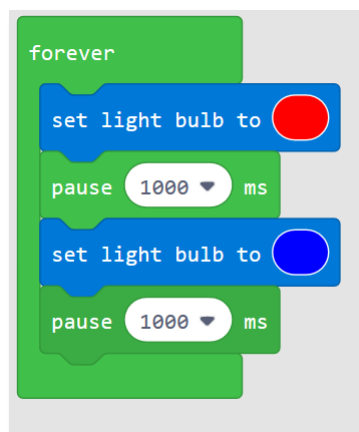
Observe the simulator to verify the Light Bulb is working as expected. Not bad right? How about we load this program on the BrainPad now?

COPYING TO THE BRAINPAD

The process of copying a program onto the BrainPad is easy but can be tricky the first time you do it. First, you must connect the BrainPad to your computer using a Micro-USB cable. By the way, you can use Windows, a Chromebook, a Mac, or almost anything with a modern web browser and a USB port.

A Micro-USB cable should have been included with your BrainPad. You might also have an extra one at home. If you don't have one, they are readily available. Micro-USB cables are even sold at most gas stations!

The smaller end of the USB cable plugs into the BrainPad.
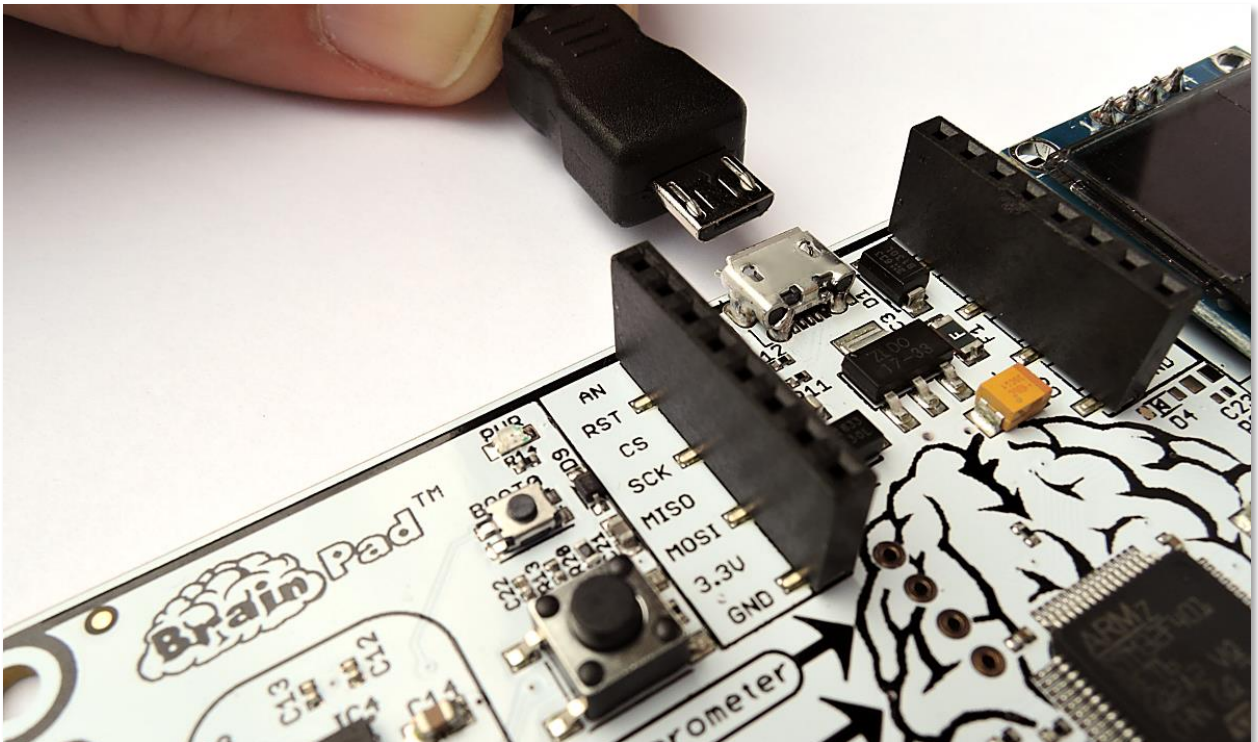
The larger end of the cable plugs into your computer.

When the BrainPad is connected to your computer the red power LED (PWR) should be on.
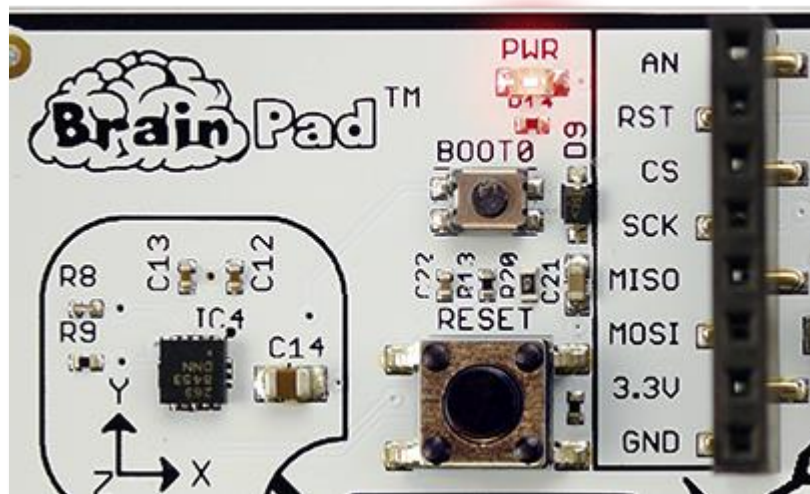
Go back to the browser with the program we made earlier. Click the download button and save the file on your computer.

If you are using Microsoft Edge, you should see a dialog box like the one shown below:

| What do you want to do with brainpad-Untitled.uf2 (250 KB)? From: makecode.brainpad.com | Open | Save | ∧ | Cancel | ✕ |
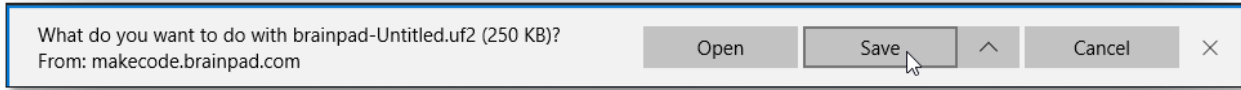
Click the save button and then the "Open folder" button in the next dialog box to show the downloaded file.

| brainpad-Untitled.uf2 finished downloading. | Open | Open folder | View downloads | ✕ |
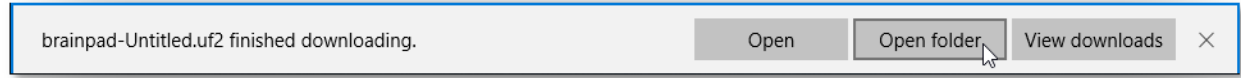
The file will be highlighted. You can right click on the file to copy it.
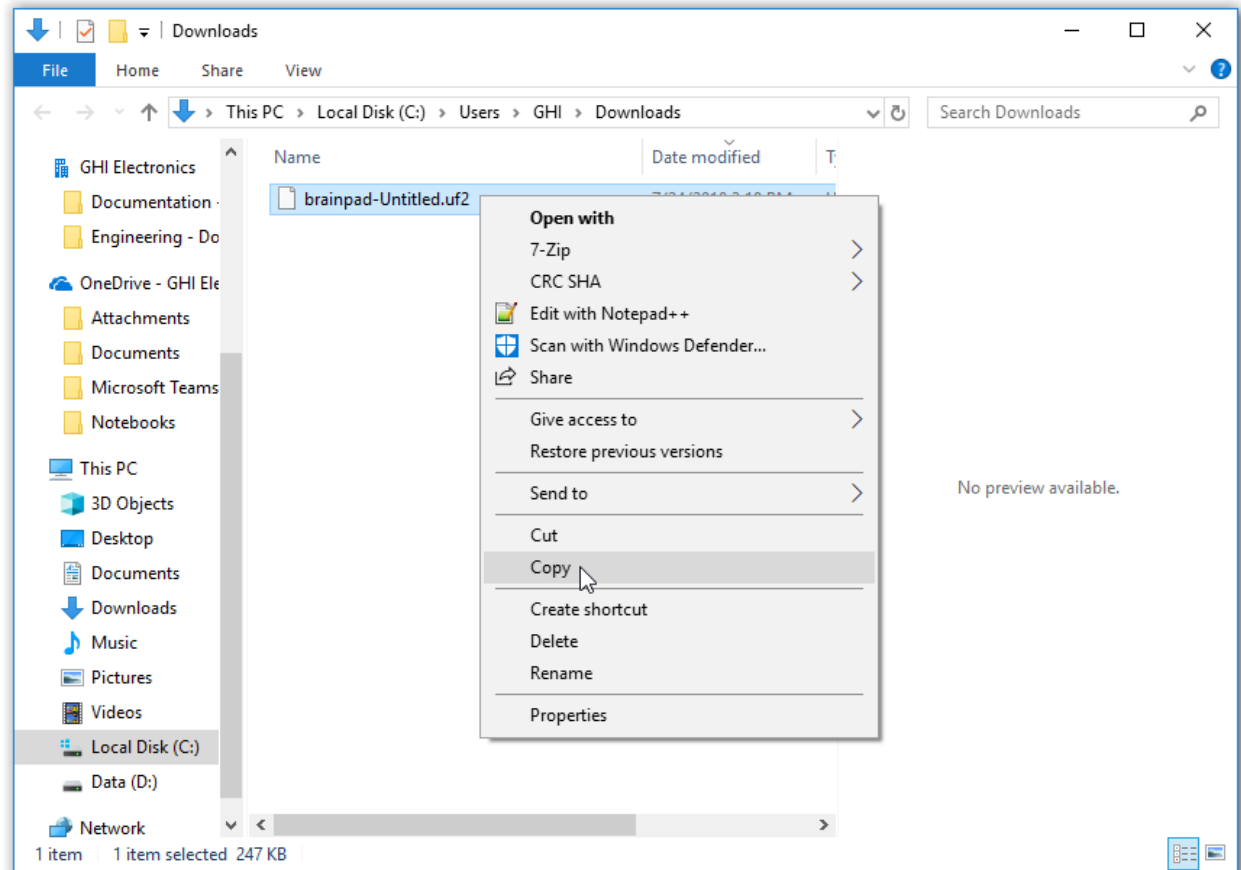
If you are using the Chrome browser, the file will be shown at the lower left corner of the screen:
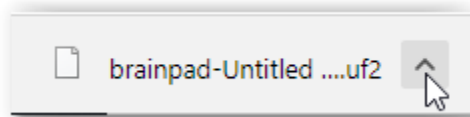
Click on the small up arrow and then select "Show in folder."
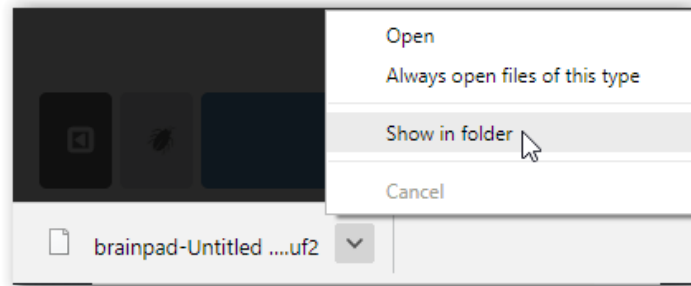
The downloaded file will be highlighted. You can right click on the file to copy it.
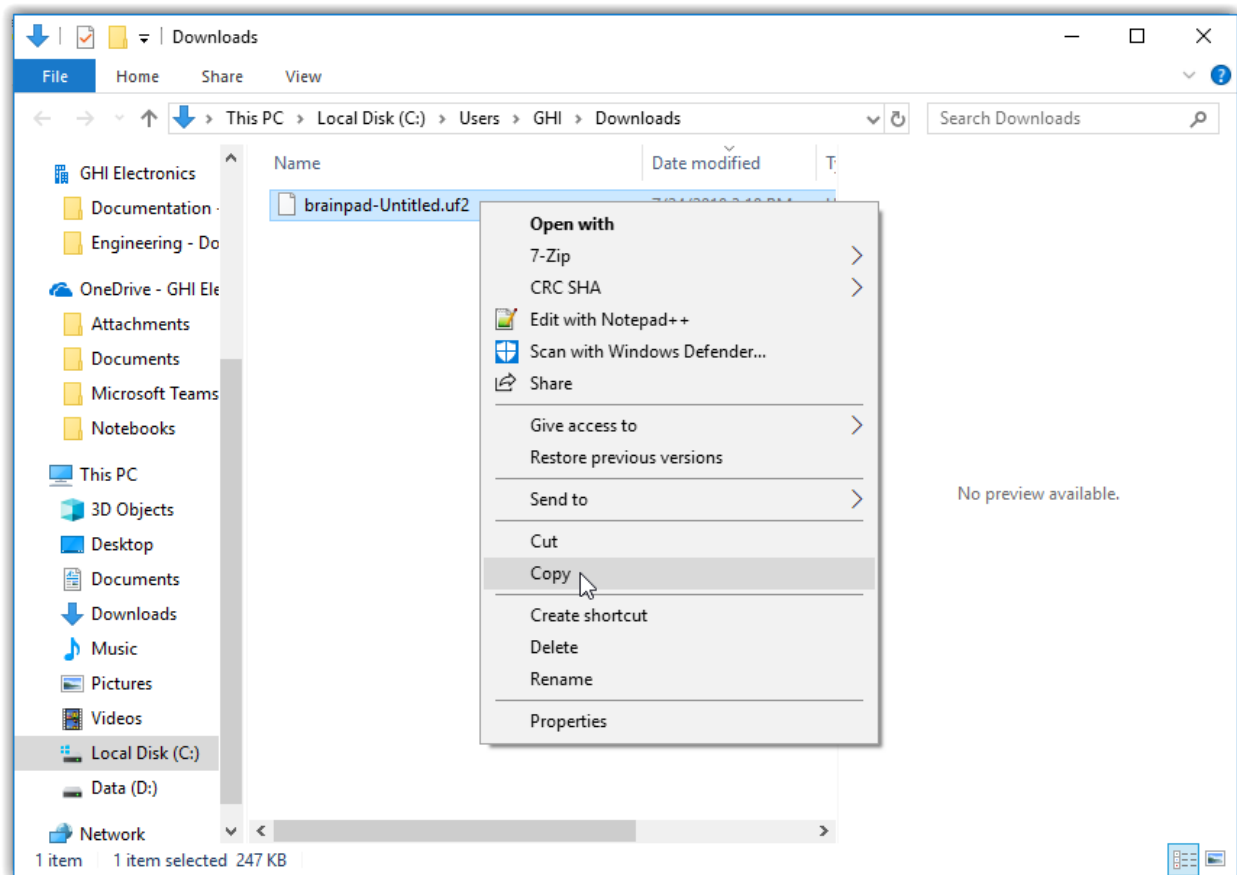
With either browser, Microsoft MakeCode does not delete your older files. If you download the same program multiple times, a number will be added to the filename (for example "brainpad-Untitled (1).uf2"). Make sure you copy the latest file. The latest file will be the highlighted file. It will also have the largest number in parenthesis and the latest date/time.

Other browsers should be similar.

The BrainPad should be connected to your computer, and the red power light should be on. Press and hold the reset button for about three seconds. The Light Bulb will turn green.
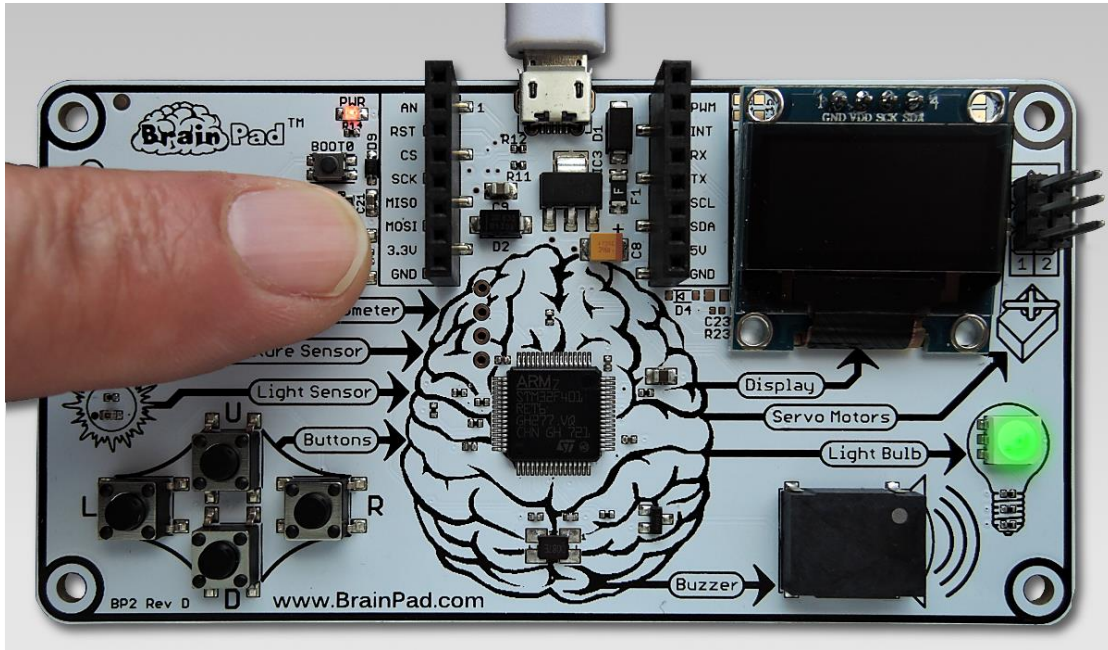
Your computer will now detect a USB storage device. It may take a minute for it to come up, but you will see a window named "BrainPad" appear on your computer screen.
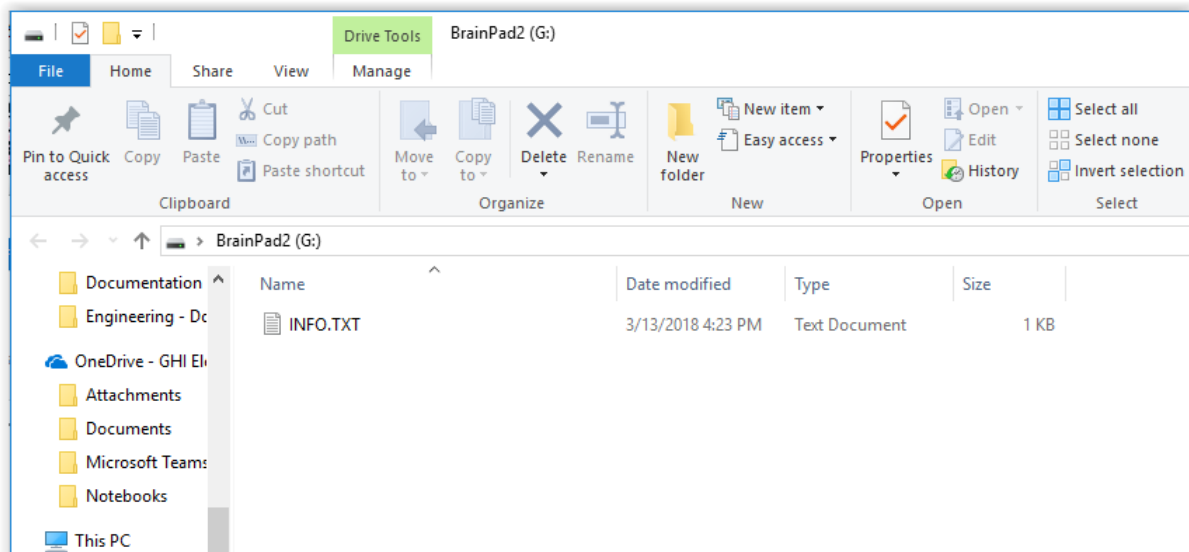
This window shows the contents of the memory inside the BrainPad. Now, right click and select paste to copy the file into the BrainPad.
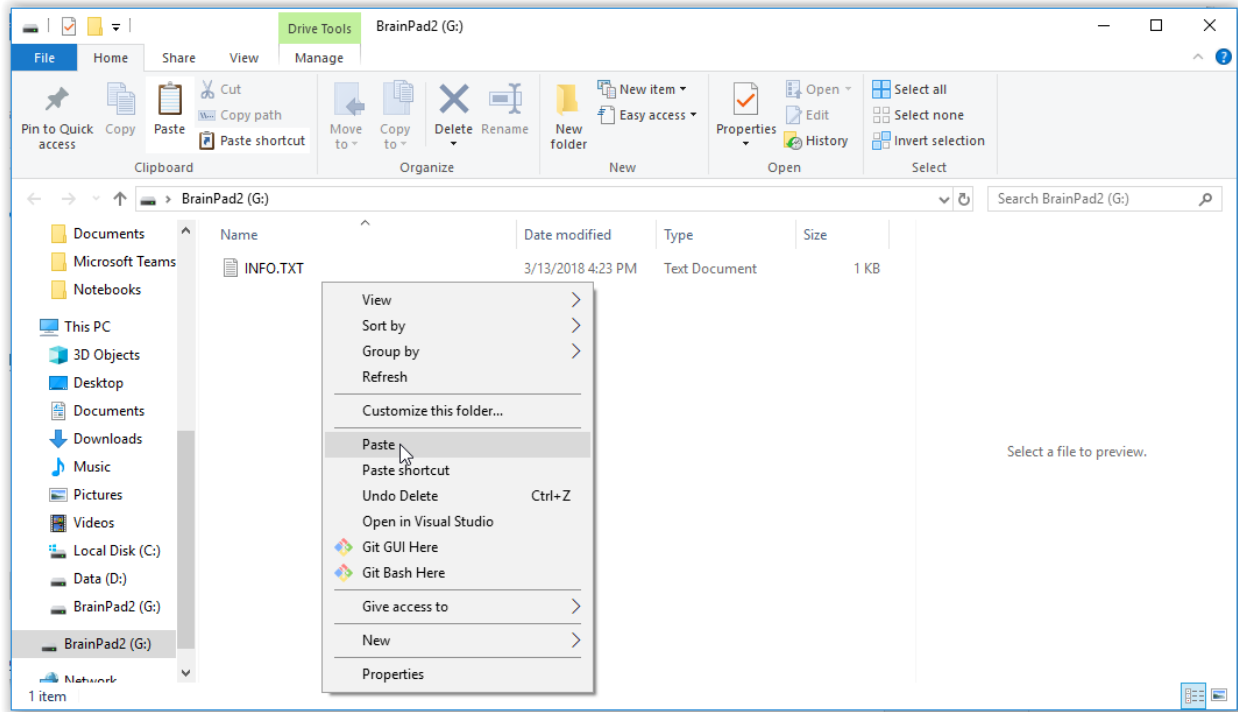
You can also drag and drop the downloaded file into the BrainPad window.
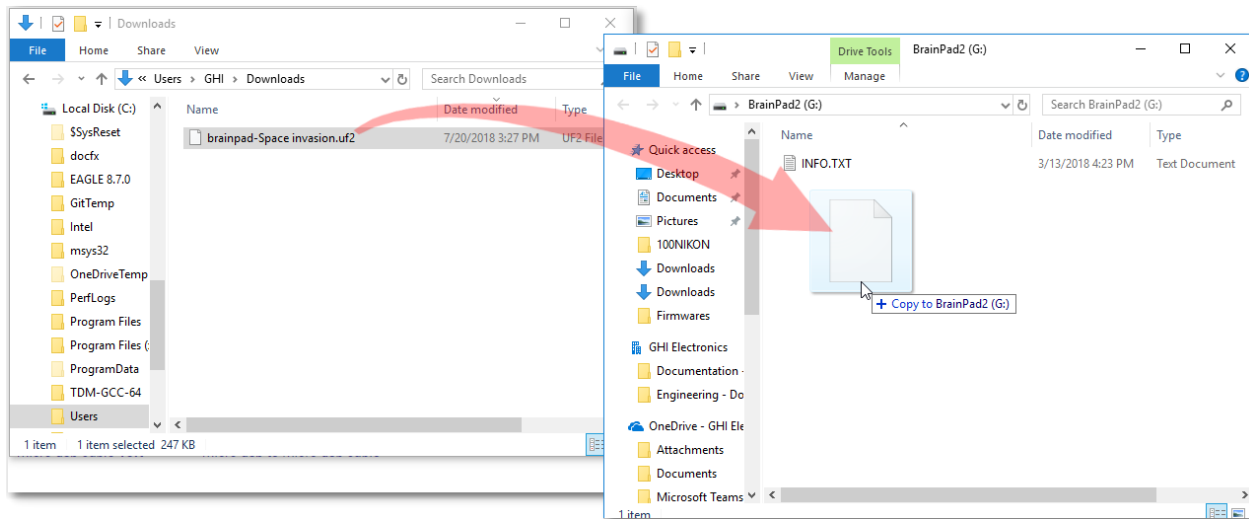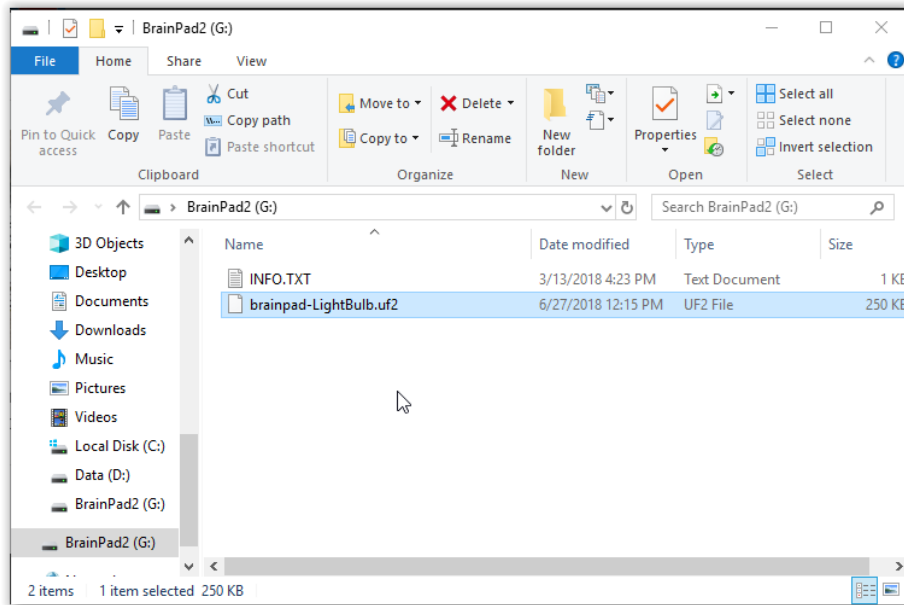
Once you do that, the Light Bulb will flicker for a moment and then the BrainPad will be running the loaded program. Now things are getting more interesting!

JAVASCRIPT

Blocks are fun to start with, but once your programs get larger you will need to brave up and start typing code. Don't worry, you can do that whenever you are ready. One of the amazing features about Microsoft MakeCode is how it translates blocks to JavaScript text-based code and text-based code back to blocks. Go back to the program we made before and click the JavaScript tab.


```javascript
1  forever(function () {
2      lightbulb.setColor(0xff0000)
3      pause(1000)
4      lightbulb.setColor(0x0000ff)
5      pause(1000)
6  })
7
```
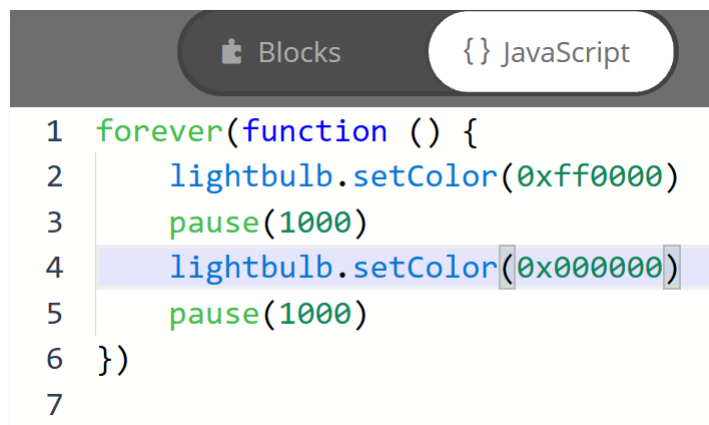
Take a look at the code. Do you see the correlation between blocks and code? One thing that will not be clear to you is the color as the code uses a hexadecimal value to represent color. Do you understand the rest of the code?

Hexadecimal is a numbering system based on sixteen values per digit instead of the usual 10. Every digit in a hexadecimal number can go from 0 to F instead of the 0 to 9 used for decimal (the numbers we use every day). Counting from 0 to 16 in hexadecimal would look like this: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10.

The "Ox" the beginning of a hexadecimal number is put there to let you know for sure it is a hexadecimal number to avoid any confusion. Hexadecimal numbers are also called "hex" for short and are also referred to as "base 16." Binary is also called "base 2," and decimal is called "base 10." If this is confusing, don't worry. It takes some time to become comfortable with different numbering systems.
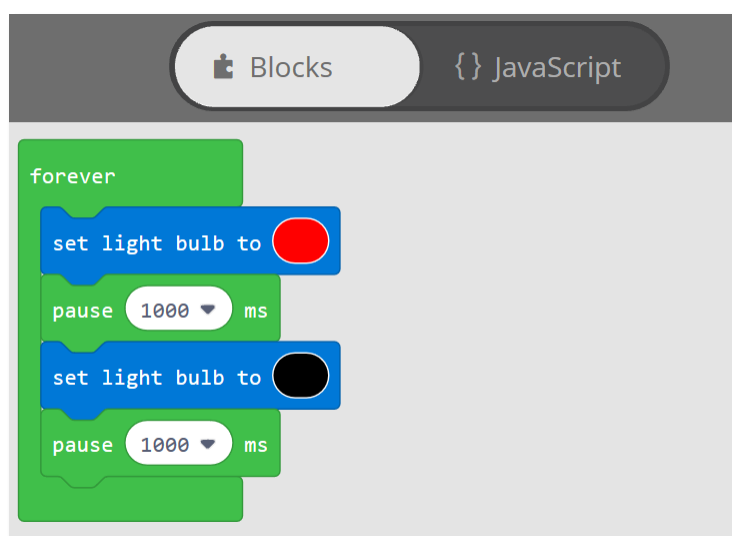
When a hexadecimal number is used to refer to a color, the first two digits represent the amount of red in the color, the second two digits the amount of green, and the third two digits the amount of blue. "00" means that color is at zero intensity, or off. "FF" means the color is at full intensity. Bright red would be 0xFF0000. White is 0xFFFFFF.

What happens if you change one of the "setColor" methods to 0x000000? Let's try it.

```
                    Blocks        {} JavaScript

1   forever(function () {
2       lightbulb.setColor(0xff0000)
3       pause(1000)
4       lightbulb.setColor(0x000000)
5       pause(1000)
6   })
7
```

Now switch back to Blocks to see what happened.

```
                    Blocks        {} JavaScript

forever
  set light bulb to  ●(red)
  pause  1000 ▾  ms
  set light bulb to  ●(black)
  pause  1000 ▾  ms
```

Looks like the color has changed from blue to black. You may have guessed that setting the Light Bulb to black is the same as turning it off. You can see that in the simulator. The Light Bulb should turn red for a second and then turn off for a second.

DISPLAY FUN

If you think the Light Bulb was exciting, wait until you try the display! If you still have the blocks from the previous example, drag them back into the menu to delete them.
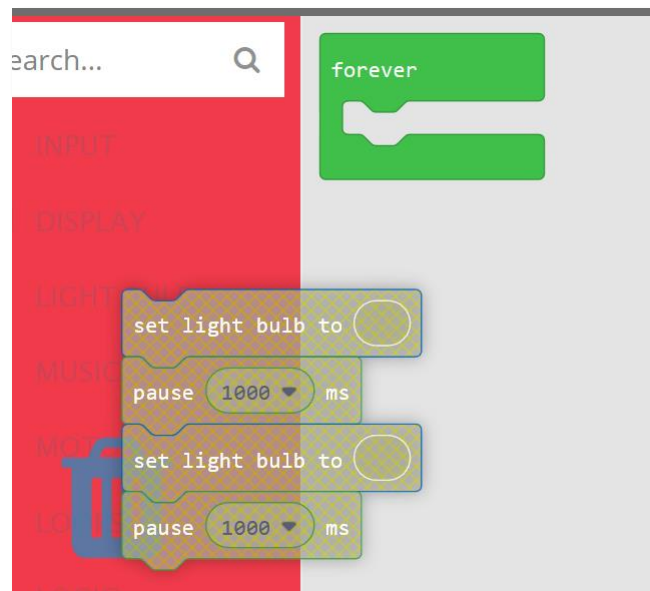
If you accidently deleted the forever block, you can simply bring it back from the LOOPS menu. From under the DISPLAY menu, drag the "show image" block and select the first of the four "walking man" images. Add the other three images and add a 10ms delay in between. There is no 10ms option in the pause block, but you can simply type it in.
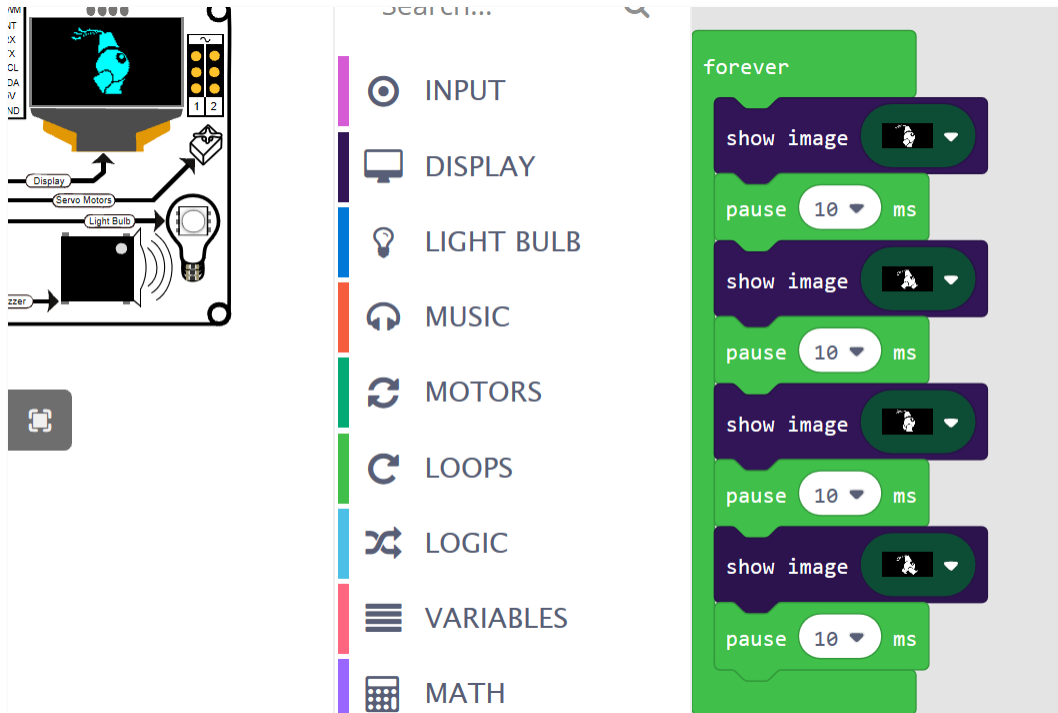
Our guy should be walking on the screen of the simulator by now. Go ahead and deploy it to the actual BrainPad.

PLAYING MUSICAL NOTES

From under MUSIC menu, you can find individual notes. . .

. . .or sounds
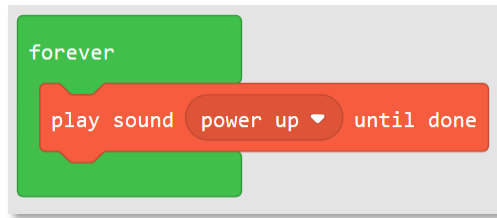
Note how we used the "play sound until done" block. This block stops the rest of the program from running until the sound is done playing. The "play sound" block (without "until done") starts executing the next block while the sound is still playing. If you try both blocks you will see the results are very different.

USING THE BUTTONS

There are two ways to read the buttons. One is to simply check if the button is pressed and to take the appropriate action when it is. The problem with this method is that you must check the button repeatedly to make sure you don't miss a button press. Another way is by letting the system do that for you, and in the "event" the button is pressed automatically do something.

We will first try reading a button by using an event. I want to play the power up sound when the up button is pressed and the power down sound when the down button is pressed.
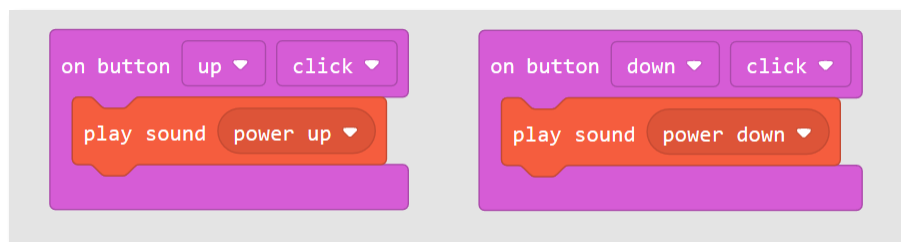
Did you notice that there is no "forever" block in this example? You can still have a forever block in your code, but it is not needed for this example. This is because the system will handle the "event" automatically when the button is pressed.

Here is the JavaScript code that represent the above blocks:

```
input.buttonU.onEvent(ButtonEvent.Click, function () {
    music.playSound(music.sounds(Sounds.PowerUp))
})


input.buttonD.onEvent(ButtonEvent.Click, function () {
    music.playSound(music.sounds(Sounds.PowerDown))
})
```

That worked well, but let's go back to our old trusty forever loop. We will create a program that does the exact same thing but this time without events. To check "if" a button is pressed you will need a. . . Ready for it?. . . You will need an "if" statement. You will find the if statement block under LOGIC. Drag it inside the forever loop.

The if statement checks the status of something that can be true or false. By default it shows "true," but we want to replace that with the button status. Drag the "button. . . is pressed" block from the input section to the "if. . . then" block as shown below. Note that this is not the "on button" event we used before.

The above blocks will check the up button over and over. If the up button is being pressed at the same time the button is checked, the program will execute any blocks inside the if statement.

Now add more blocks to make the program as shown below:
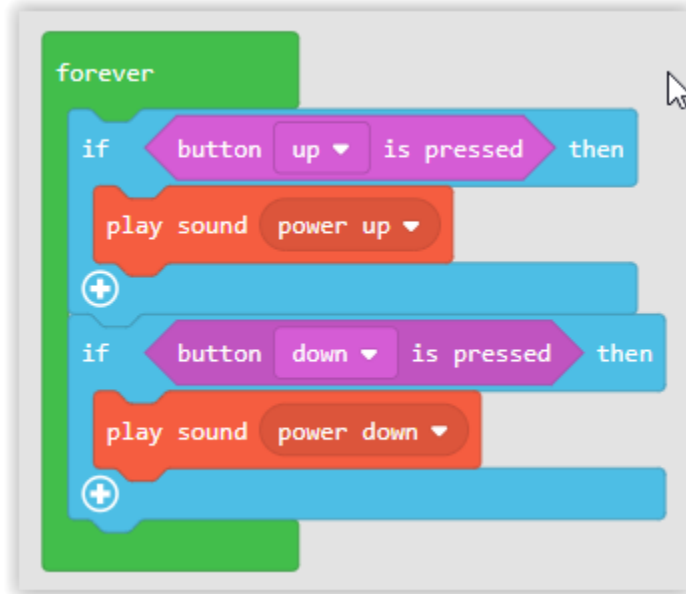
When you run this program, you will notice that the sound doesn't play properly until you release the button. The reason for this is that every time the BrainPad detects the up or down button being pressed it starts playing a new sound -- even if it is in the middle of playing the previous sound. The BrainPad will start playing the sound many times each second while the button is being pressed.

There are two ways to fix this. One way is to replace the "button. . . is pressed" blocks with "button. . . was pressed" blocks. The button was pressed block will be true if the button was pressed at least one time since the button was last checked. Even if the button was pressed multiple times, it will only register as a single button press. Pressing the up or down button while a sound is being played will interrupt the sound, but the sound won't repeat while holding the button down.

The other way is to replace the "play sound. . ." Blocks with "play sound. . . until done" blocks. The play sound until done block stops the program from doing anything else until the sound is done playing. Doing this stops the program from checking for a button press or playing any other sounds until the current sound is done playing. Using play sound until done will make it so the sound cannot be interrupted. The blocks are shown below.
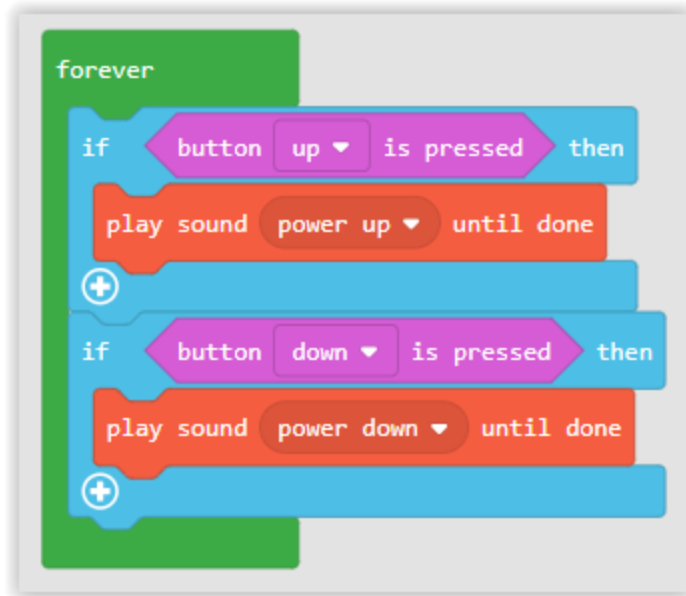
IS IT DARK IN HERE?

Just like the buttons, you can read the light level and do something with it, or you can have an event that fires up when it is bright or dark. However, reading a button is slightly different than reading a light.

A button is either pressed or not pressed. So, when I say, is the button pressed? Your answer will be true if it is pressed and false if it is not. With the Light Sensor, the sensor gives the light level as a value. For programmers, this is known as an analog value. Let's read the light intensity and show it on the screen. We want this in a forever loop, so it is constantly reading the light level and updating the screen.

Since you are an expert in using the if-statement, can you print bright and dark on the screen depending on the Light Sensor reading? This is where things get a bit tricky, but I will explain. The if statement wants something that is true or false, like "is the button being pressed?". The light level is not true or false -- it can be one of many values. If you think about it, it is doesn't make sense to say if light then do something. You have to say if the light is brighter (or dimmer) than a certain level, then do something. Go to the LOGIC menu and add a comparison. Your if statement will also now have and else statement. Take a look at this:
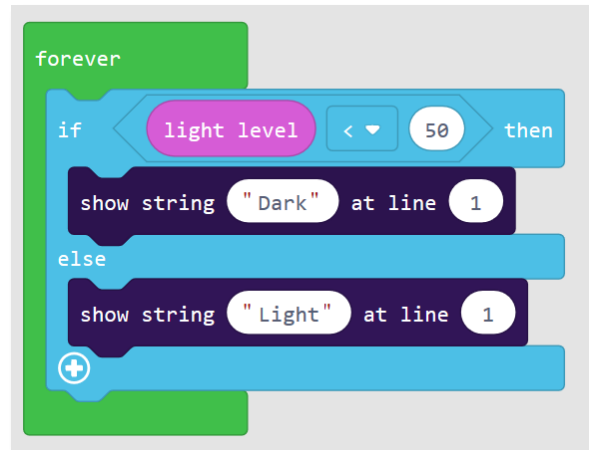
What did that say? If the light level is less than 50 then print "Dark" on line one; otherwise, print "Light" on line one.

Don't be afraid to visit the JavaScript tab often and see what the code looks like.

```javascript
forever(function () {
    if (input.lightLevel() < 50) {
        display.showString("Dark", 1)
    } else {
        display.showString("Light", 1)
    }
})
```

The simulator also works with the Light Sensor. Once the Light Sensor is added to a project, the simulator will start showing an option to set the light level. First, start by making the simulator larger by clicking the full screen button.
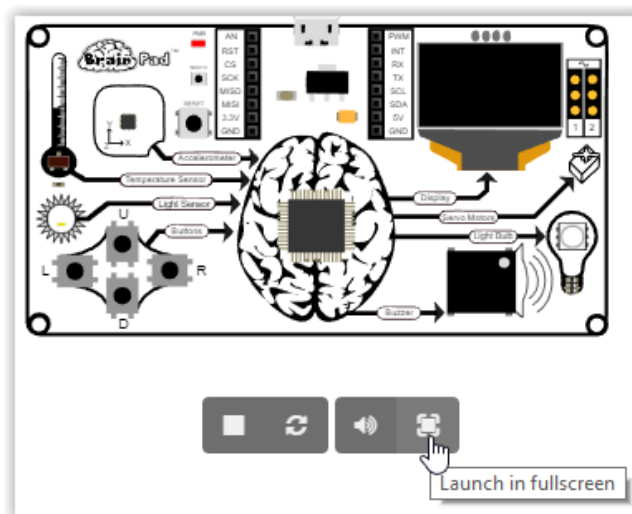
The Light Sensor shows the light level which can be changed by dragging the line up and down. This is similar to covering the actual light sensor on the BrainPad with your hand. As you drag the light level up and down, the screen should alternately display "Light" and "Dark."
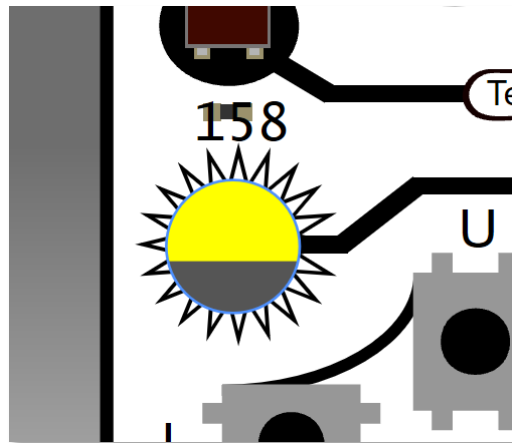
HEARING TEST

Human hearing extends from a low frequency of about 20 hertz (or cycles per second) to a high frequency of about 20,000 hertz. Dogs can hear higher frequencies than humans, with their hearing extending to 44,000 hertz. That's why we can't hear high-pitched whistles made for dogs, but dogs hear them very well!

As we age, we lose the ability to hear very high frequencies. Also, noise induced hearing loss affects the ability to hear high frequencies more than low frequencies. Want to see how high of a frequency you can hear? Let's find out!

We need to write a program that uses a "variable." A variable is simply a named storage location that can hold a piece of information. In this case, our variable will hold a number that represents the frequency we want to play on the Buzzer.

Go to VARIABLES and click on the "Make a Variable. . ." button.
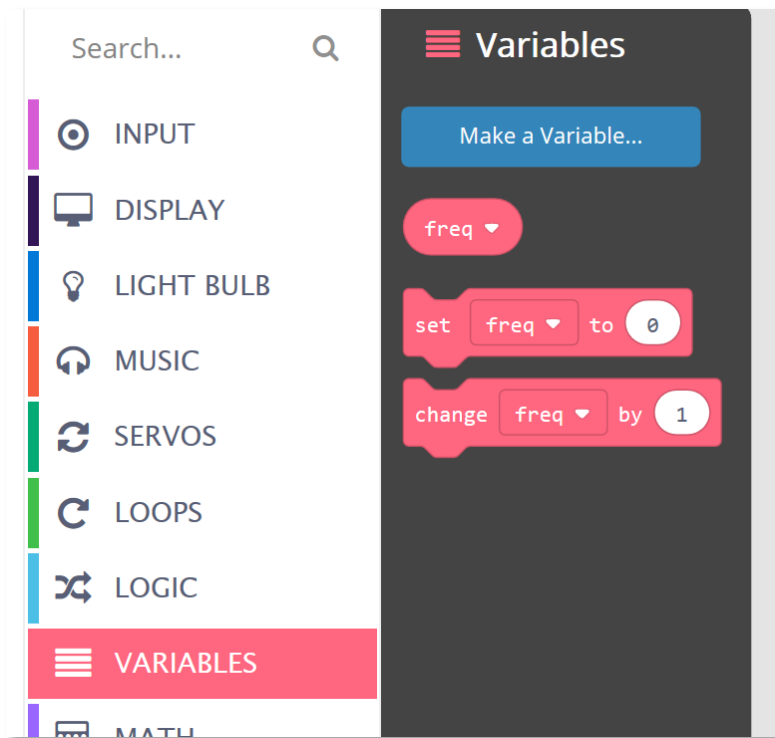
Now give your variable a name. I will use the name "freq". This variable will store the frequency I want to play on the Buzzer.
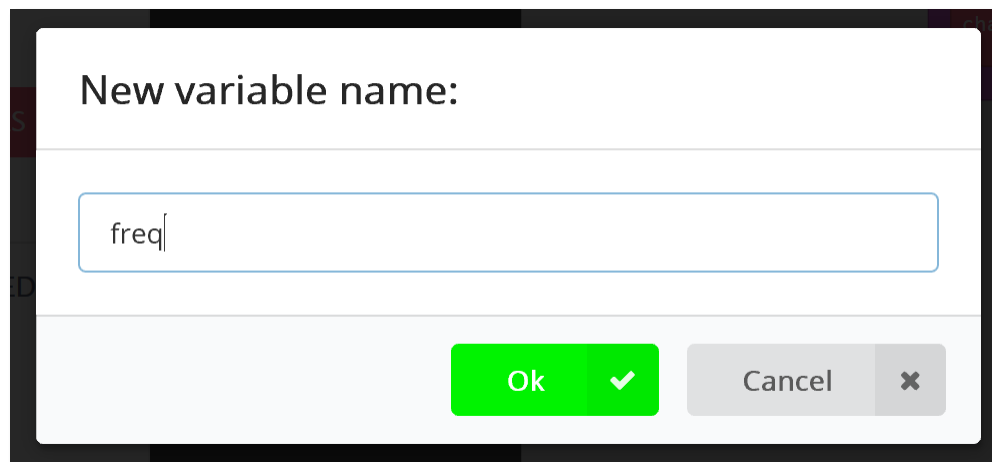
Now go to LOOPS and add an "on start" block. Whatever you add in this block will happen when the program starts. Here we want to set the variable to 2000. We all can hear a frequency of 2000 hertz, so it's a good starting point. The "set" block is found under VARIABLES.
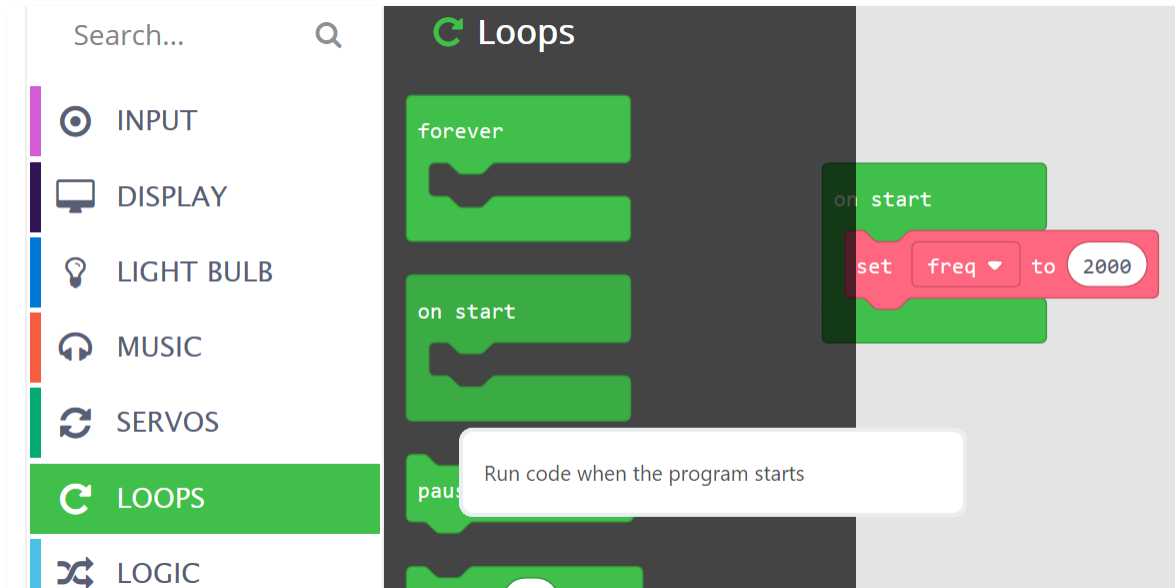
Now, we need to make a sound at that frequency, but we also want to increase the frequency when the up button is pressed and decrease the frequency when the down button is pressed. You will need the "change" block from under variables. Set it to change by 100 when up is pressed and by -100 when down is pressed.
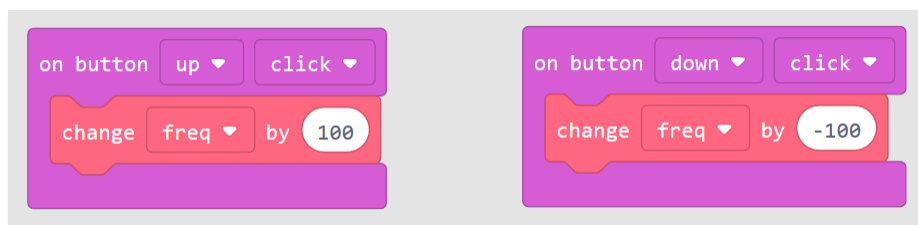
Now that we have the variables changing as we like, we can generate the sound (ring tone at) and print the frequency on the screen (show number) as well.
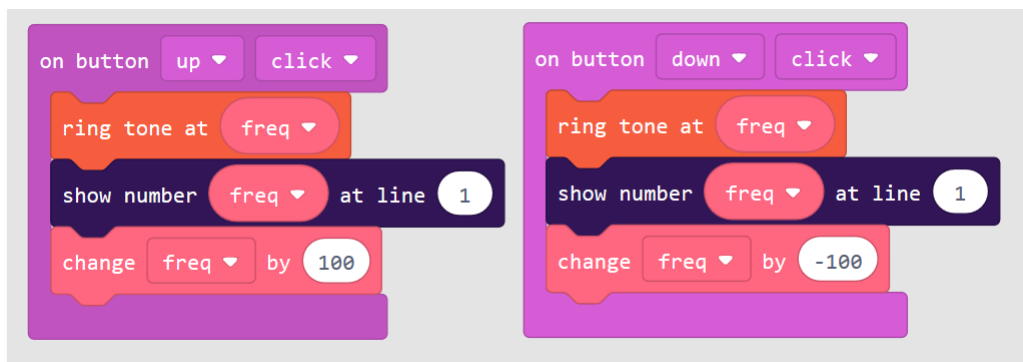
One last thing! The noise can be very annoying so let's stop the sound when the left button is pressed.

on button [left ▼] [click ▼]
    stop all sounds
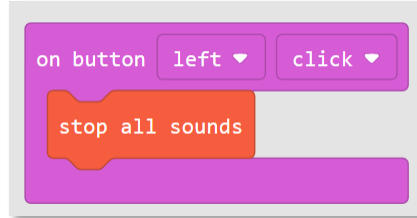
Here is the complete program.

on start
    set [freq ▼] to (2000)

on button [left ▼] [click ▼]
    stop all sounds

on button [up ▼] [click ▼]
    ring tone at (freq ▼)
    show number (freq ▼) at line (1)
    change [freq ▼] by (100)

on button [down ▼] [click ▼]
    ring tone at (freq ▼)
    show number (freq ▼) at line (1)
    change [freq ▼] by (-100)
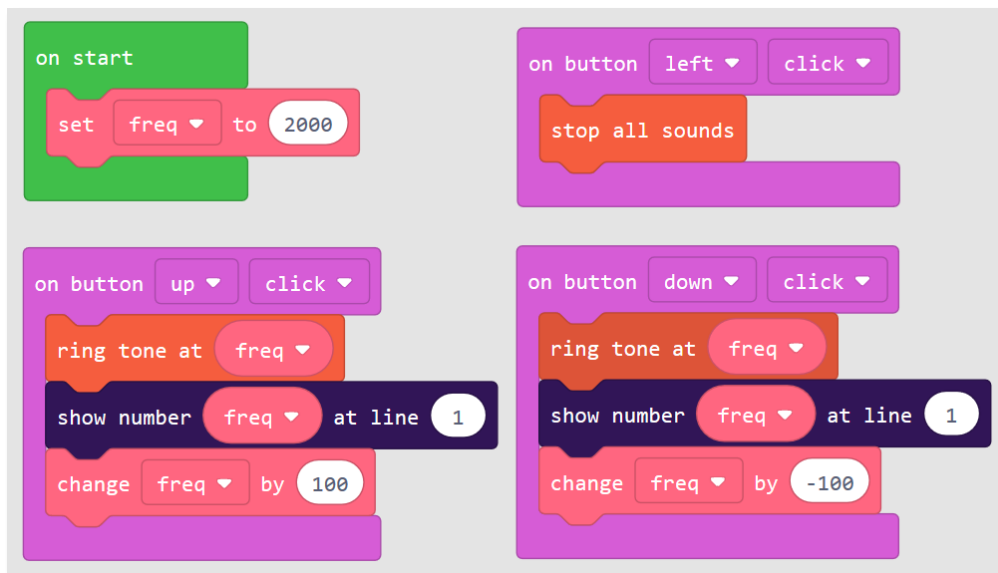
SERVO MOTORS

A servo motor is basically a motor that has a small internal circuit that allows you to control the servo using simple electrical pulses.

Servos have a three-pin connector that plugs directly into the BrainPad. Unfortunately, different servos often use different colored wires. To identify the proper way to plug it in, always ignore the middle wire and look at the wires on each side. The lighter color wire is the one that needs to be next to the ~ symbol printed on the BrainPad. This is the signal wire, by the way.

Servo motors (or servos) are often used in radio-controlled models and can be found online and at local hobby stores. These servos draw power from the BrainPad, which in turn draws power from your PC or a battery. Therefore, we recommend using the smaller servos, called micro servos. Some of the larger servo motors draw too much power to work properly without adding an additional power supply.

Servo motors are available as either continuous or positional servos. While they look identical, a positional servo only turns to a given position and then holds that position until you tell it to move to another position. A continuous servo motor will rotate continuously in one direction until it is told to either stop or reverse direction.

If you turn an unpowered positional servo by hand it will only move approximately one half of a rotation in either direction before stopping. When you rotate a continuous servo in either direction it does not stop. Because of the internal gearing of the servo, you will need to apply a little force to turn either type of servo motor.

POSITIONAL SERVO MOTORS

As stated above, a positional servo motor turns to the position (or angle) you tell it to and stays there until it is told to move to another position. If you try to turn a servo motor by hand and it suddenly stops, it is a positional servo motor. A pulse sent from the BrainPad tells the servo which angle or position to move to.
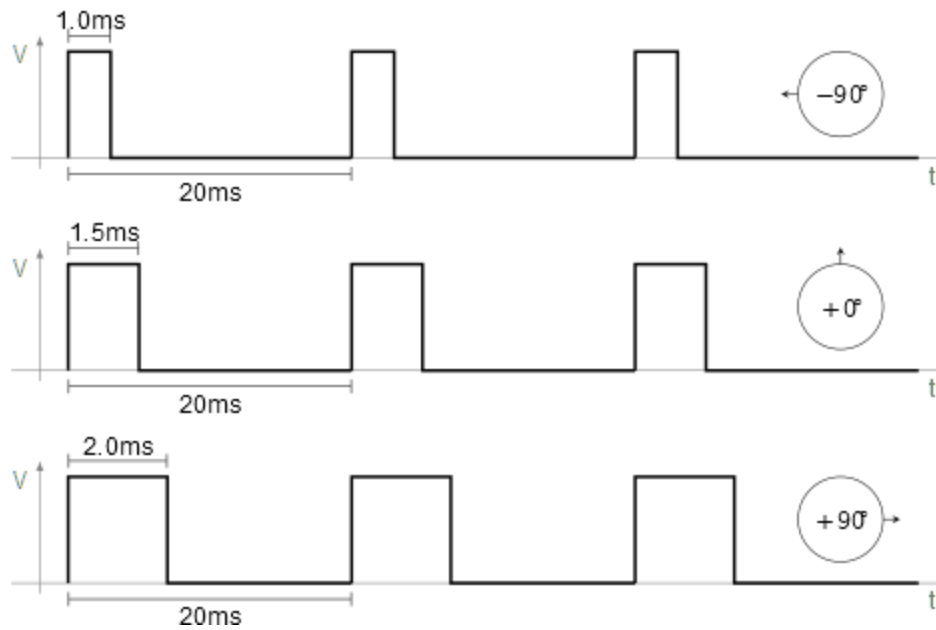
There are many ways to use a positional servo. In this example, we will create a large dial thermometer. The code we use is very simple and doesn't even use a variable!

In a forever loop add a "show value" block (found within the display section). In the first oval of the show value block type " Temp (F)" within the parenthesis as shown below:
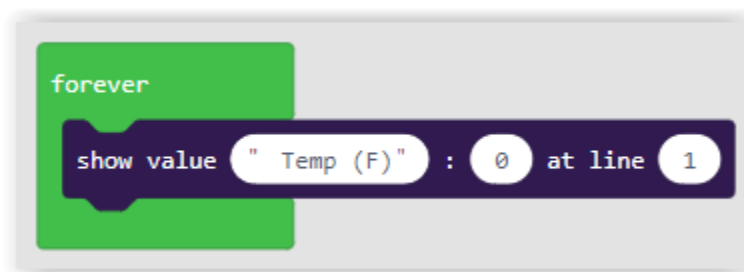
From the input category of blocks select the "temperature in °C" block and drag it into the second oval. Our thermometer will display the temperature in Fahrenheit, so we changed the "°C" to "°F," but you can use Celsius if you prefer. Now change the third oval to "3." This will print the temperature at the third line of the BrainPad display. Your blocks should look like the screen shot below:

So far, we have made a thermometer that will display the temperature on the BrainPad's display. Now we must convert the temperature into an angle and send it to a servo motor to move the dial on our thermometer. Let's make our thermometer work over a range of -20° Fahrenheit to 120° Fahrenheit.

We must convert the temperature (-20 to 120) into an angle (0 to 180). If you are not a mathematician, fear not! MakeCode has a map block that will do the math for you. It is located under the MATH menu. We can complete our program with the addition of one more line:

You will notice that the angle goes from a low of 180 to a high of 0 in our map block. Isn't this backwards? The reason is that when the servo motor is set to 0 degrees the motor is turned to the right, and when the servo is set to 180 degrees it is turned to the left. Mapping the temperature of -20° Fahrenheit to 0 degrees on our servo would make the thermometer read backwards.

Here is finished thermometer made from a box and a straw glued to a servo motor.
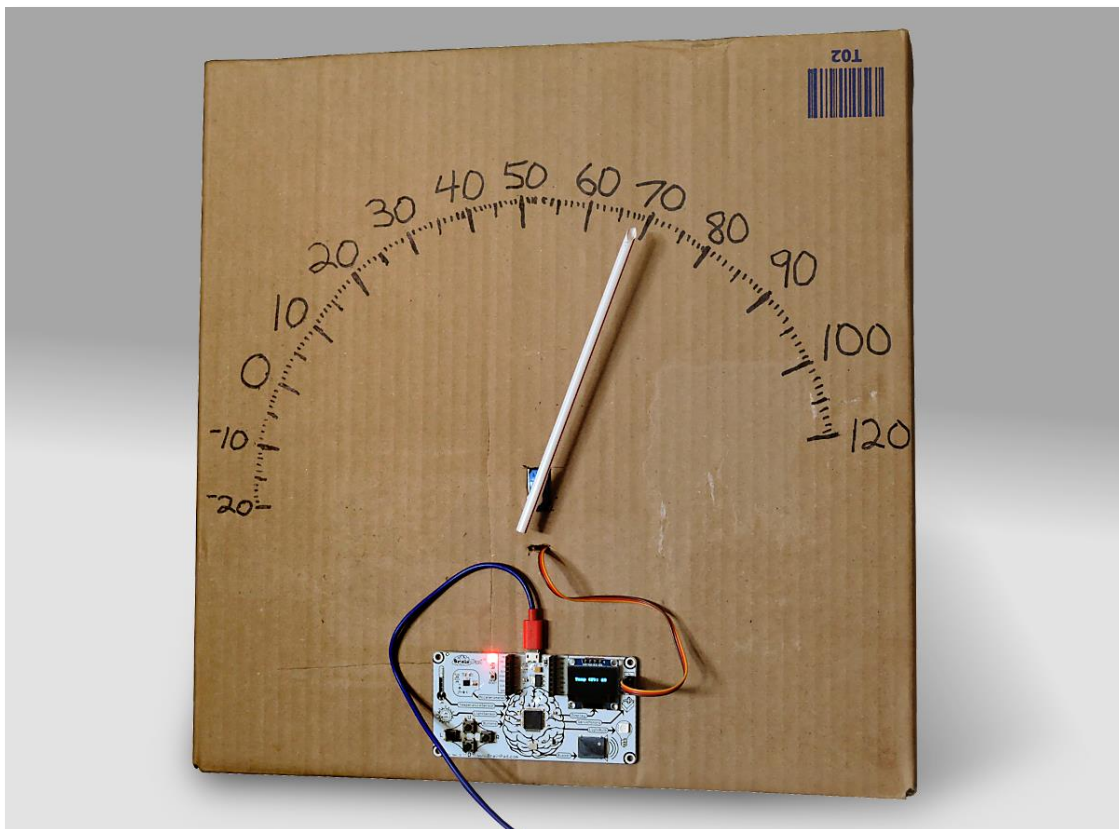
CONTINUOUS SERVO MOTORS

The other type of servo motor is continuous, which, as the name indicates, can constantly turn. In this case, the pulse sets the speed and direction of the motor. The BrainPad has connections for two motors. By controlling the direction and speed of two rotational servos, you can easily move a robot. I searched Amazon.com for "continuous micro servo with wheels" and found exactly what I need.

Now I could have mounted these on a cardboard box, but then I found this chassis by FEETECH.

Here is where things got a bit tricky! The robot comes with wheels and motors. The motors look like servo motors, but they turned out to be not servos, just regular motors. I could tell because they only had two wires instead of

three. The kit also came with a little circuit, which I then realized that circuit converts the regular motors into servo motors!
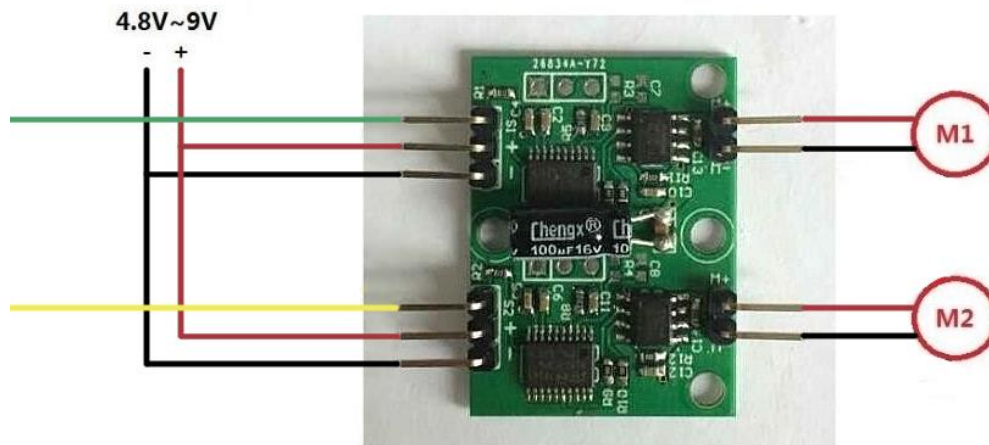
This is doable but too complicated for most classrooms, so I put the circuit and the included motors aside and decided to stick with proper continuous servo motors. My BrainPad fit perfectly on this chassis as if it was made for it! I then found a flat phone charging battery that fit very well under the BrainPad.

This robot is finally ready to dance! To go forward, both motors need to turn forward. However, as one of the motors is facing right and the other is facing left, one will have to turn opposite from the other for the wheels to

turn in the same direction. This may sound confusing, but once you try it you will understand how it works. This code will move both servos forward at full speed for one second, stop the motors for one second, and then repeat.

If the motors continue to turn when the speed is set to zero they need calibration. Most servo motors will have a tiny adjustment inside. You will need a tiny screwdriver for this task.
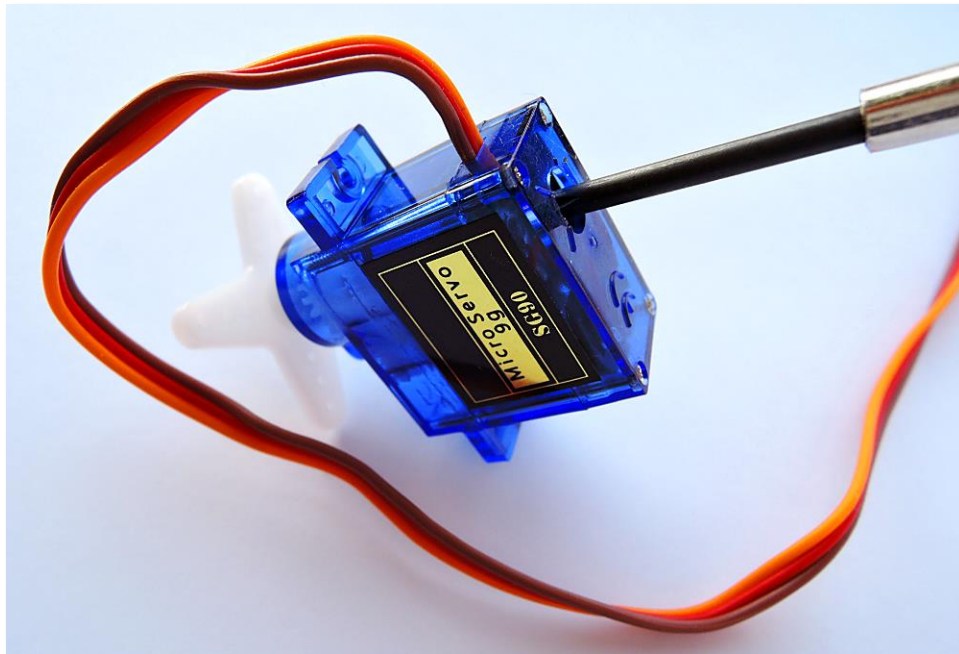
Back to moving the robot forward. Do you see now that one motor is moving in the wrong direction? Change one of the servos to go in reverse at full speed.

```
forever
    continuous servo  1 ▼  run at  100  %
    continuous servo  2 ▼  run at  -100  %
    pause  1000 ▼  ms
    continuous servo  1 ▼  run at  0  %
    continuous servo  2 ▼  run at  0  %
    pause  1000 ▼  ms
```
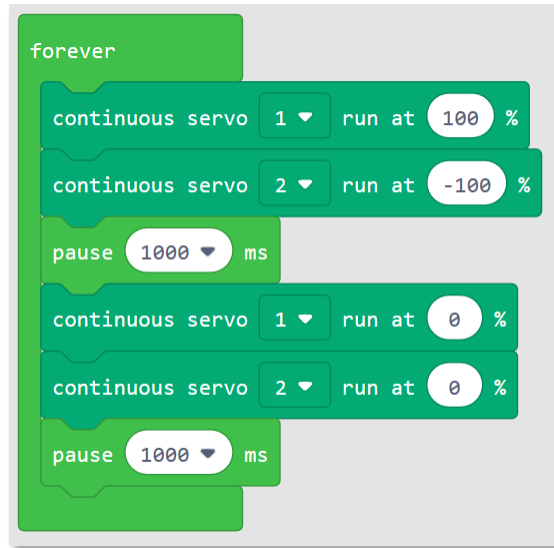
This work was the start of our "Sun Seeker" robot project. You can find it here
https://docs.brainpad.com/projects/sun-seeker.html. Make sure to watch the video. The robot uses the light
sensor to detect if the robot is in the sun or the shade. If it is in the shade, it will turn the robot until it finds a
sunny spot to move into.

CHRISTMAS LIGHTS

There are many things you can do with the devices built into the BrainPad, but at some point you might want to
wire the BrainPad to external components. While this is beyond the scope of this book, here is a quick introduction
to give you an idea of what is possible.

For this project we will use addressable LED (Light Emitting Diodes) strips. These LED strips consist of a line of lights
that can be controlled individually. They can be set to virtually any color and brightness combination using only
two wires. These wires send data pulses to the LEDs to command them as needed.

What you need to find are strips with the APA102 controller. This controller uses a SPI bus, which is available on the BrainPad. These strips are powered through different voltages. We will need 5V ones since we have access to 5V.
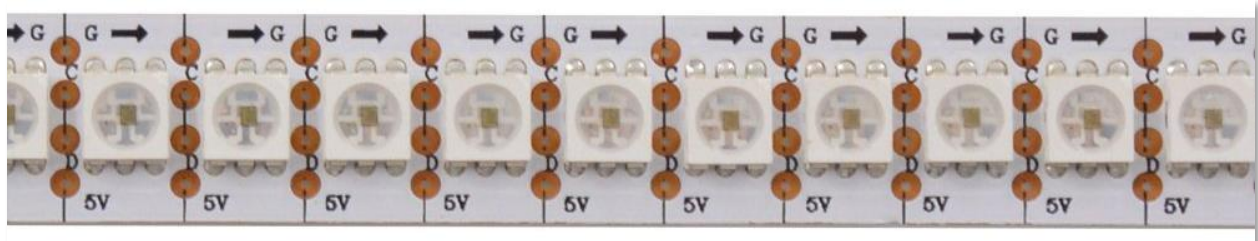
The strip can be cut to any needed size. To keep my setup from drawing too much power, I will cut it down to 25 LEDs. Now to soldering some wires. The arrow on the strip needs to point away from the side you solder the wires to. The 4 wires are G-Ground, C-Clock, D-Data and 5V.
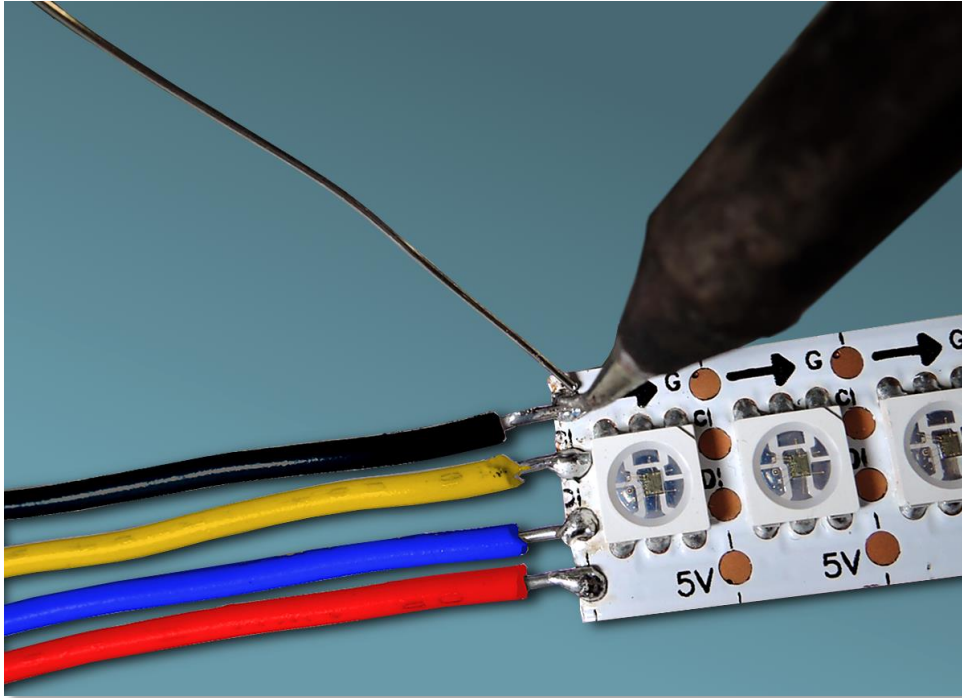
Ground (GND) and 5V are clearly marked on BrainPad's expansion header. As for C-Clock, it should connect to SCK on the BrainPad. D-Data should connect to MOSI on the BrainPad.
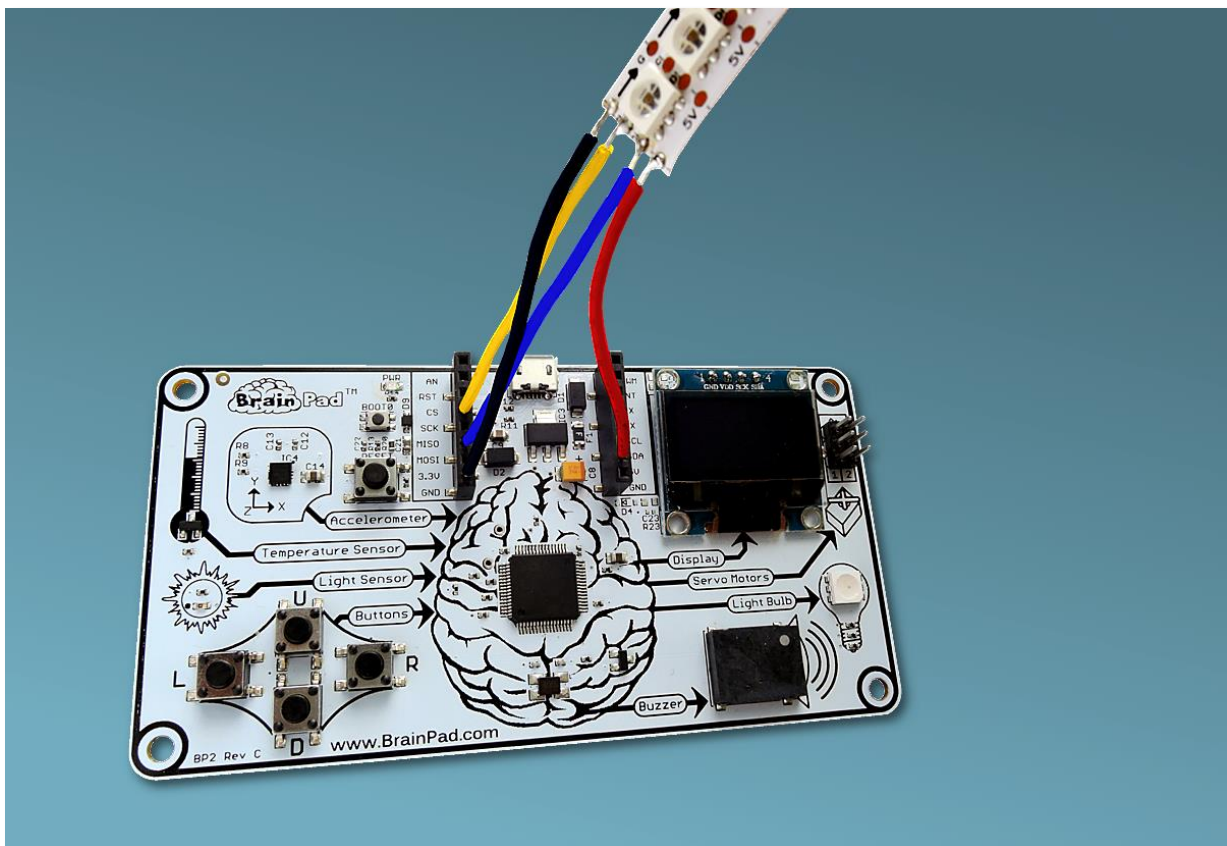
Each LED has three internal LEDs. They are red, blue and green. By controlling the levels of these base colors, you can create any color you wish. There are several online tools to help with color picking. A good example can be found at http://www.rgbtool.com. What you care about are the R, G, and B values.

Color picker

H:	292	
S:	69	
V:	78	
R:	181	0.7082
G:	62	0.2418
B:	199	0.78
HEX:	B53EC7	
HEX (#):	#B53EC7	
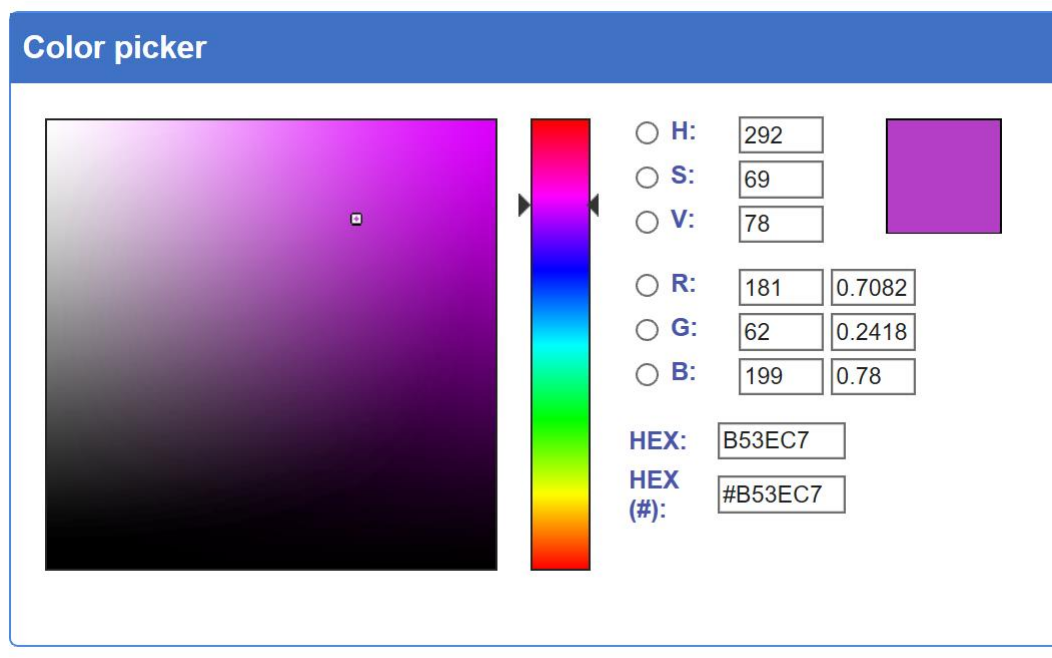
Commands are sent to the LEDs using the BrainPad's SPI bus. To start writing a sequence of colors to the LED strip, a start frame must be sent first. A start frame is just four zeros.

After the start frame, each LED requires four numbers to control it. The first number sets the global brightness for the entire string of LEDs. We don't want to get bogged down in details, so we will always make the first number 255, which is full brightness.

After the first number, the remaining three numbers range from 0 to 255 and control the brightness of blue, green, and red in each LED. To turn an LED bright blue, you would send 255, 0, 0. To turn an LED off, you would send 0,0,0. To make an LED bright white, you would send 255,255,255.

The LEDs in the strip are simply chained together. The first color is sent to the first LED, the second color to the second LED, and so on. The LEDs will continue lighting in sequence until a new start frame is sent (four zeros). After the start frame the sequence starts over with the first LED.

Because of the number of blocks needed, it is easier to switch to JavaScript and copy and paste the code. Copy and paste the following code into the Microsoft MakeCode JavaScript editor:

```
let center = 12
let startRed = 0
let startGreen = 0
let startBlue = 0
```

```
let red = 0

let green = 0

let blue = 0

let data: Buffer = pins.createBuffer(4)

let dummy: Buffer = pins.createBuffer(4)


pins.spiFrequency(1000000)


function SendStart() {

    data[0] = 0

    data[1] = 0

    data[2] = 0

    data[3] = 0


    pins.spiTransfer(data, dummy)

}


function SendRGB(red: number, green: number, blue: number) {

    data[0] = 255

    data[1] = blue

    data[2] = green

    data[3] = red


    pins.spiTransfer(data, dummy)

}


forever(function () {

    startRed = Math.constrain(startRed + Math.randomRange(-20, 20), 0, 255)

    startGreen = Math.constrain(startGreen + Math.randomRange(-20, 20), 0, 255)

    startBlue = Math.constrain(startBlue + Math.randomRange(-20, 20), 0, 255)


    SendStart()
```

```
    for (let index = 0; index <= 24; index = index + 1) {

        red = Math.constrain(startRed - Math.abs(index - center) * 16, 0, 255)

        green = Math.constrain(startGreen - Math.abs(index - center) * 16, 0, 255)

        blue = Math.constrain(startBlue - Math.abs(index - center) * 16, 0, 255)


        SendRGB(red, green, blue)

    }

})
```

If you did everything correctly you should see a brightly colored random light show. Try changing some of the above code to see how it impacts the lighting.
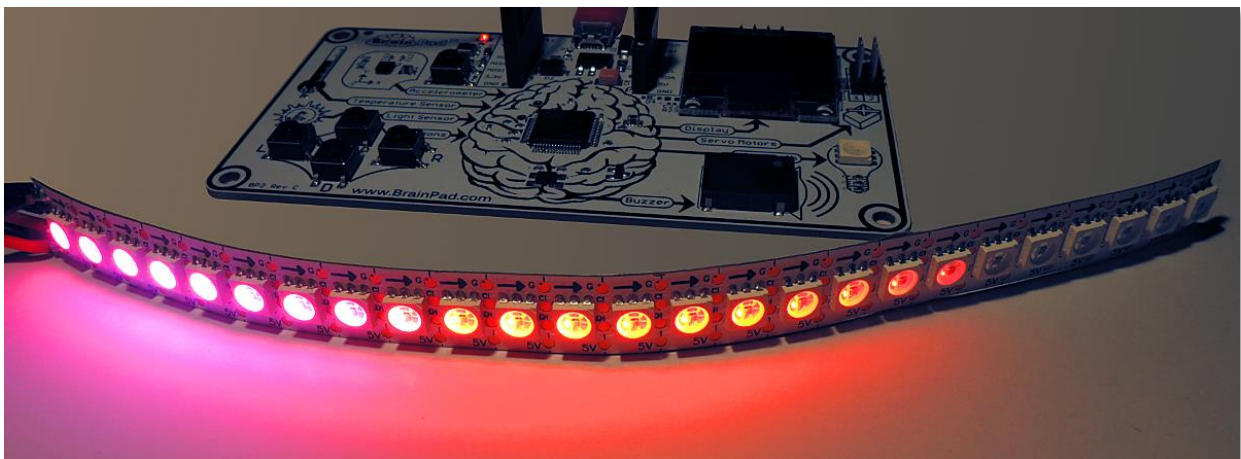
CONCLUSION

Microsoft MakeCode and the BrainPad make a great combo to learn programming. The simulator means you can easily test code without even having a BrainPad. Then you can load the program on the BrainPad by simply copying the downloaded program file to the BrainPad window.

In the next chapters we will Go Beyond and code using C# or Visual Basic in Microsoft Visual Studio. Going Beyond is optional but highly recommended once you feel comfortable with coding in MakeCode. In many cases, using MakeCode is all you need.

GO BEYOND

You are here because you are interested in learning how to program like a pro. You will be using Microsoft Visual Studio to program in C# or Visual Basic. You will have full debugging capabilities along with a feature rich development environment.

Using Microsoft MakeCode is not a prerequisite but highly recommended. After all, we named it **Start Making** for a reason. Either way, put on your seatbelt as you are about to go on a serious ride into true professional coding that could turn into a future career!

TINYCLR OS™

For the BrainPad to work with .NET in C# and Visual Basic, you will need TinyCLR OS. This tiny operating system does all the magic. It interprets compiled .NET assemblies and contains thousands of services like threading, memory management, and debugging. This is the same .NET that's used to develop software for phones or computers. TinyCLR OS is, however, made to run on small computers like the BrainPad, so it is a subset of the full .NET standard.
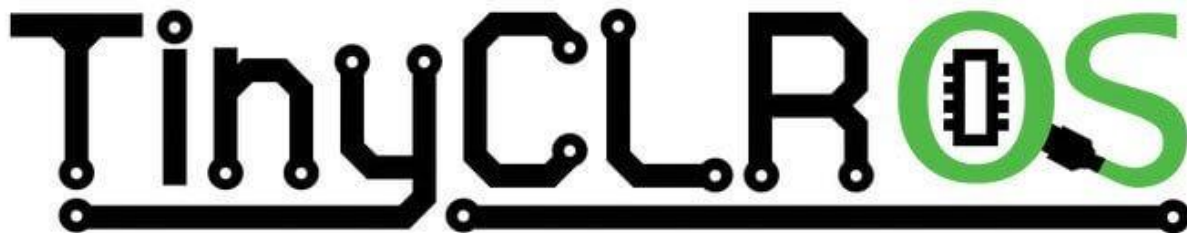
VISUAL STUDIO

In theory, any .NET compiler can be used with TinyCLR OS; however, TinyCLR is only supported by Microsoft Visual Studio. The community edition of Visual Studio is a full-featured programming environment that is considered one of the best (if not the best) available -- and best of all it's free! To install Visual Studio, you will need a modern PC with Windows. Windows version 10 is recommended.

Note that you can use the same Microsoft Visual Studio and the same skills you are learning programming the BrainPad to start writing apps for many other devices, like smartphones, PCs and even Microsoft Xbox gaming consoles!

SYSTEM SETUP

The Microsoft Visual Studio Community edition is free, but it is not a toy and it is not small. You will need some time to setup the system. Full instructions are found on the BrainPad documentation website at https://docs.brainpad.com/go-beyond/system-setup.html.

In summary, you will need to download and install:

1. Microsoft Visual Studio Community edition. Do not forget to select the ".NET desktop development" option during setup.
2. GHI Electronics TinyCLR OS project system

If you get lost you can always visit our forum for further assistance https://forums.ghielectronics.com/c/brainpad

Now that the PC is setup, you need to load TinyCLR OS on the BrainPad. Download the TinyCLR OS firmware file from here https://docs.brainpad.com/resources/downloads.html and then load it on the BrainPad just like you would load any file you created with Microsoft MakeCode:

1. Hold the reset button for about three seconds. The Light Bulb will turn green.
2. The PC will detect a storage device and open a window for it.
3. Save TinyCLR OS firmware onto the BrainPad storage device. You can also drag the file or copy and paste the file into the BrainPad window.

HELLO WORLD

It is common when programming a new device, or when using new programming tools, to start with a very simple program to make sure that everything is working properly. This program is usually called a "hello world" program. Let's do that now.

Open Microsoft Visual Studio and start a new project. In the "File" menu select "New" and then "Project" (File > New > Project).
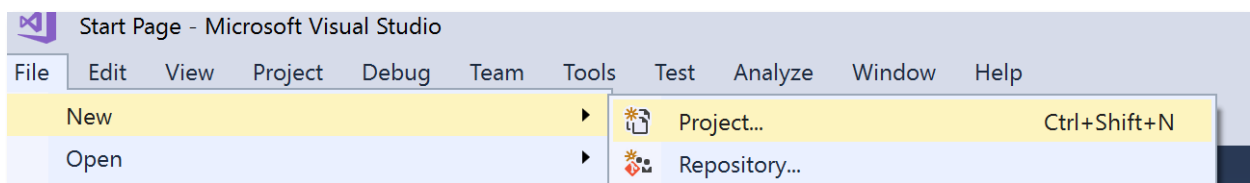
There should be a TinyCLR option under Visual C#. If not, then you did not install the TinyCLR OS project system. Here I've named the project "HelloWorld" and placed it on my desktop.
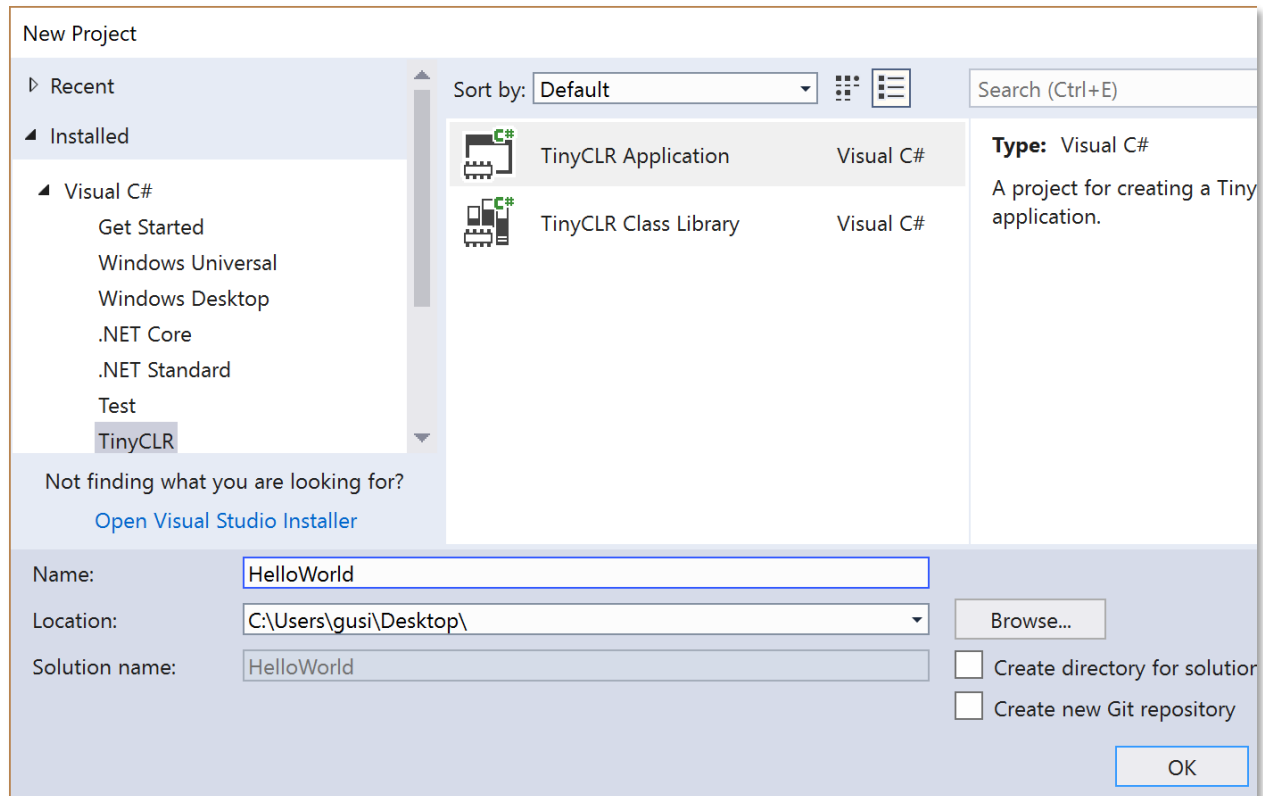
You can now load this empty program onto your BrainPad. It won't do much, but it will verify that everything is working as expected. Go ahead and plug in BrainPad and click Start.
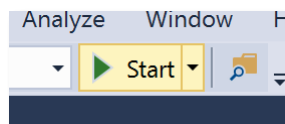
Observe the activity showing in the output window. If that window is not visible, hit Alt+Ctrl+O on your keyboard or select "Output" in the "View" menu (View > Output).

Now we are ready for BrainPad to say, "Hello World." We will first display "Hello World" in the Visual Studio output window. Later we will show it on the BrainPad display. Add this line to the program, right under the left bracket ({) following `Main()`.

```
System.Diagnostics.Debug.WriteLine("Hello World");
```

Note how Visual Studio automatically gives you word suggestions.

```
namespace HelloWorld
{
    class Program
    {
        static void Main()
        {
            System.Diagnostics.|
        }                    ┌─────────────────────────────┐
    }                        │ {} CodeAnalysis             ▲│
}                            │ 🔧 ConditionalAttribute      │
                             │ {} Contracts                 │
                             │ 🔧 Debug                     │
                             │ 🔧 DebuggableAttribute       │
                             │ 🔧 Debugger                  │
                             └─────────────────────────────┘
```
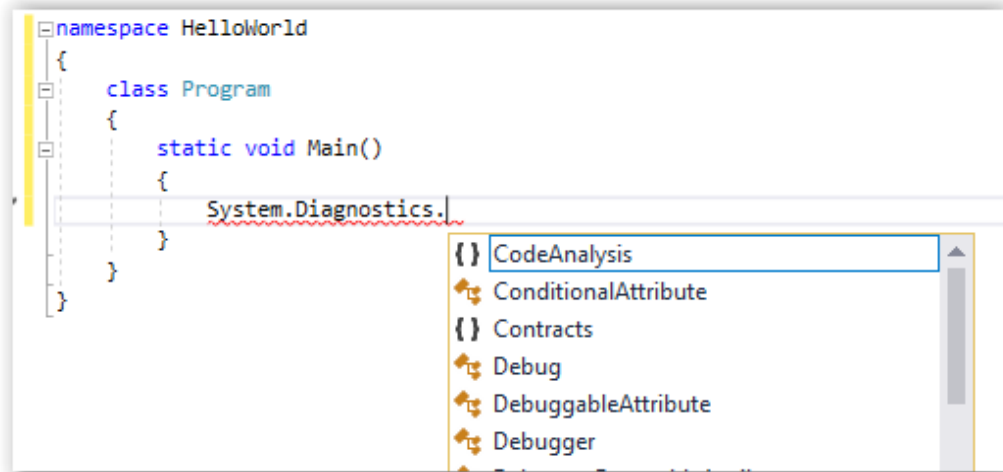
When done, the code should look like this:

```
using System;
using System.Collections;
using System.Text;
using System.Threading;

namespace HelloWorld
{
    class Program
    {
        static void Main()
        {
            System.Diagnostics.Debug.WriteLine("Hello World");
        }
    }
}
```
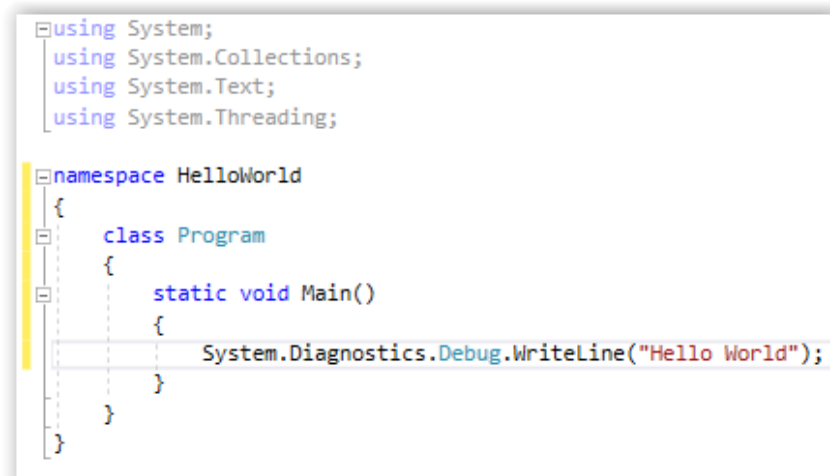
Start the program and observe the output window again.

```
The debugging target runtime is loading the application assemblies and starti
Ready.

Done.

Waiting for debug commands...

'GHIElectronics.TinyCLR.VisualStudio.dll' (Managed): Loaded 'C:\Users\gusi\De
The thread '<No Name>' (0x2) has exited with code 0 (0x0).
Hello World
The thread '<No Name>' (0x1) has exited with code 0 (0x0).
The program '[3] TinyCLR application: Managed' has exited with code 0 (0x0).
```
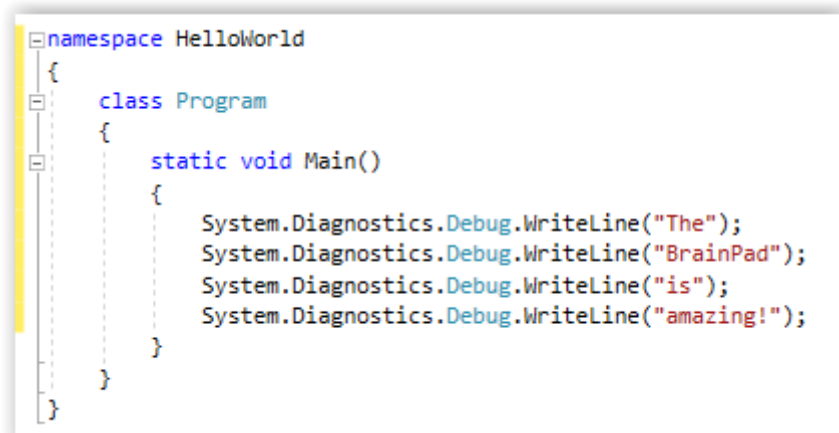
Did you find Hello World? That was a good start but not very exciting. Let's try printing multiple lines. Replace the line we added before with these four lines.

```
System.Diagnostics.Debug.WriteLine("The");
System.Diagnostics.Debug.WriteLine("BrainPad");
System.Diagnostics.Debug.WriteLine("is");
System.Diagnostics.Debug.WriteLine("amazing!");
```

Put the cursor on the first line and hit F9 function key. This will add a breakpoint. You can do the same from under the Debug menu.

```
namespace HelloWorld
{
    class Program
    {
        static void Main()
        {
            System.Diagnostics.Debug.WriteLine("The");
            System.Diagnostics.Debug.WriteLine("BrainPad");
            System.Diagnostics.Debug.WriteLine("is");
            System.Diagnostics.Debug.WriteLine("amazing!");
        }
    }
}
```

Start the program just like before. As soon as execution hits the breakpoint, it will pause at that line.

```
6   namespace HelloWorld
7   {
8       class Program
9       {
10          static void Main()
11          {
12              System.Diagnostics.Debug.WriteLine("The");
13              System.Diagnostics.Debug.WriteLine("BrainPad");
14              System.Diagnostics.Debug.WriteLine("is");
15              System.Diagnostics.Debug.WriteLine("amazing!");
16          }
17      }
18  }
19
```

It will stop right before that line is executed. Press the F10 key or click the step over button

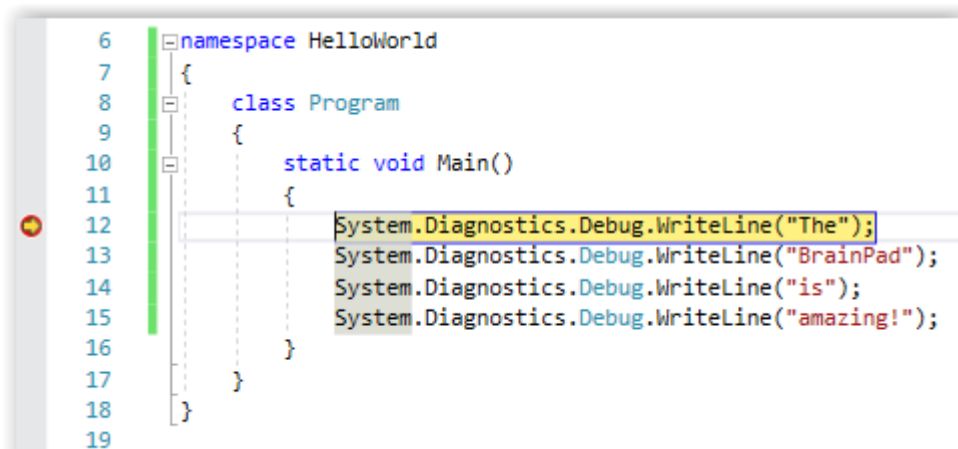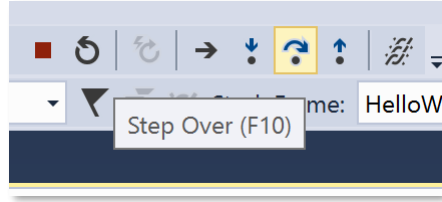This will make the program execute one line of code, or one step. Since the line we are on prints "The" in the output window, you will see it there. Step further in the code (F10) while observing the output window. This is called stepping through code and is extremely useful when trying to debug your programs to figure out why something is not working as expected.

NON-ENDING PROGRAMS

It makes sense for a program to close or end on a PC or a phone. However, on smaller devices, called embedded devices, programs do not usually end. Think of a microwave oven for example. The microwave has a small "brain" inside that is always monitoring the keys for presses and running the clock on the screen. Unless you unplug the microwave, this program is always running. This is known as an infinite loop and is used often in computer programming.

Let's create a counter on the BrainPad that runs "forever." Basically, the BrainPad will count once every second and never stop. Below is what the code should look like.

Note that you need to stop any running program before modifying it. You can do so by pressing the stop button.
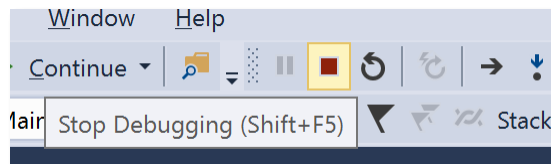

```csharp
namespace HelloWorld {
    class Program {
        static void Main()
        {
            var count = 0;

            while (true)
            {
                System.Diagnostics.Debug.WriteLine("Counting: " + count);
                Thread.Sleep(1000); // Wait one second

                count = count + 1;
            }
        }
    }
}
```

The output window will show the counter.

```
The thread '<No
Counting: 0
Counting: 1
Counting: 2
Counting: 3
Counting: 4
Counting: 5
```

While the program is running, insert a breakpoint on the counter line. From there you can step through the code like we did before. Now, hover the mouse over our variable "count," and Visual Studio will show you the current value of the variable. When I stopped the program, it was at 6. By the way, inspecting variables is another very useful feature for debugging programs.

```
static void Main() {
    var count = 0;
    while ( count 6
        System.Diagnosti
```

BRAINPAD LIBRARIES

So far, we have used pure TinyCLR OS to run simple programs. The BrainPad ships with libraries to help in the use of all the available inputs and outputs. Right-click on the project in the Solution Explorer window and then select Manage NuGet Packages…
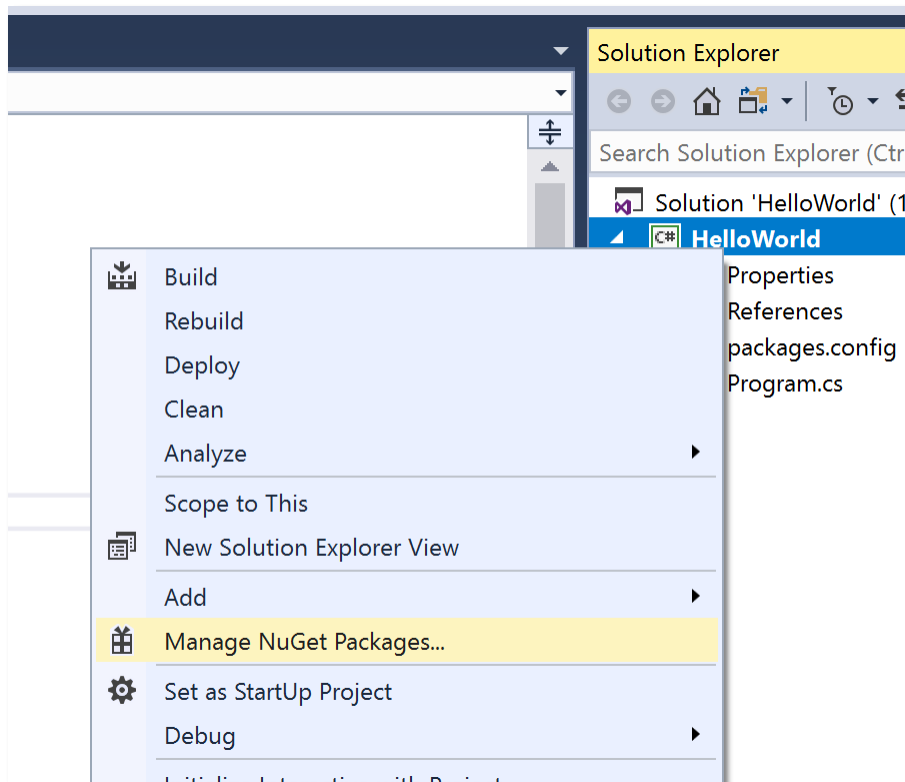
NuGet is an online service for hosting libraries. You can also download the libraries and host them locally. Select where you want the libraries to come from. For example, I have the libraries hosted locally in a directory named TinyCLR.
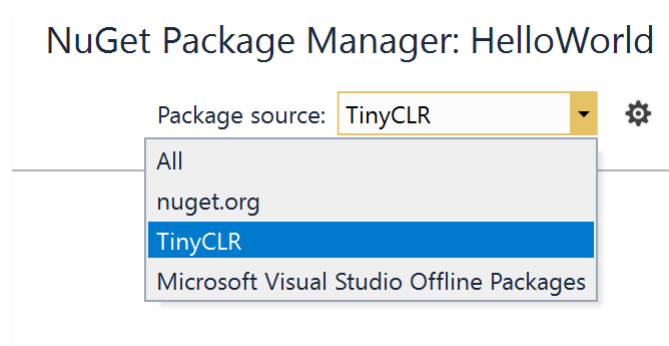
Under Browse, enter "brainpad" in the search bar.
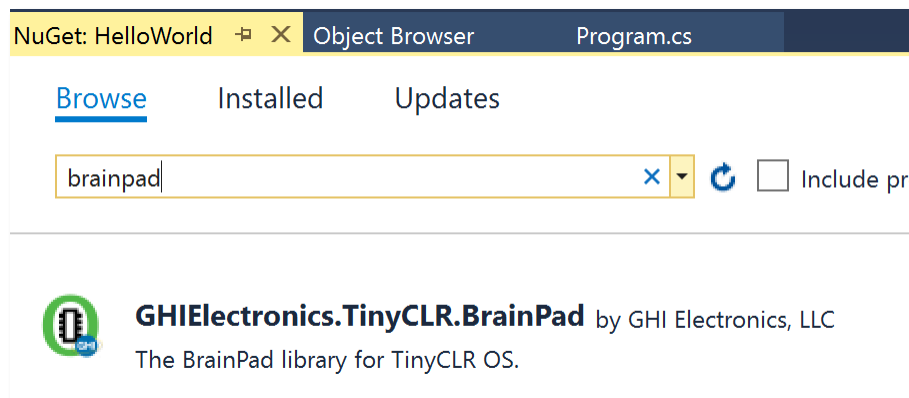
This will reveal the `GHIElectronics.TinyCLR.BrainPad` library. Select it and click install
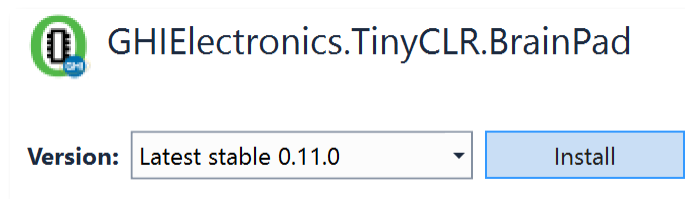
This step will add the needed libraries under references in the solution explorer.
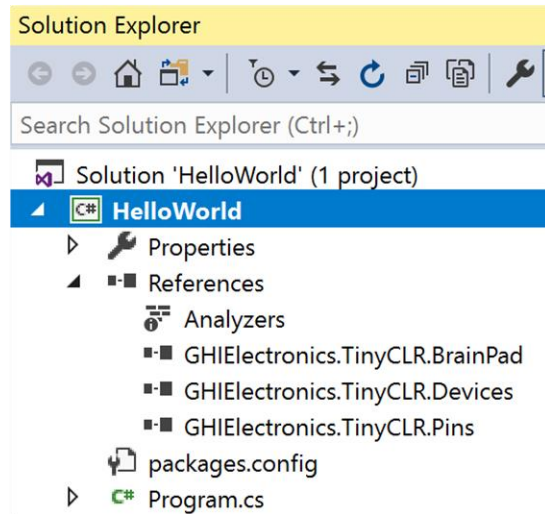
To ease adding libraries to your code, a helper class is provided. Right click on the project the select Add > New Item. . . You can also use the Ctrl+Shift+A shortcut.
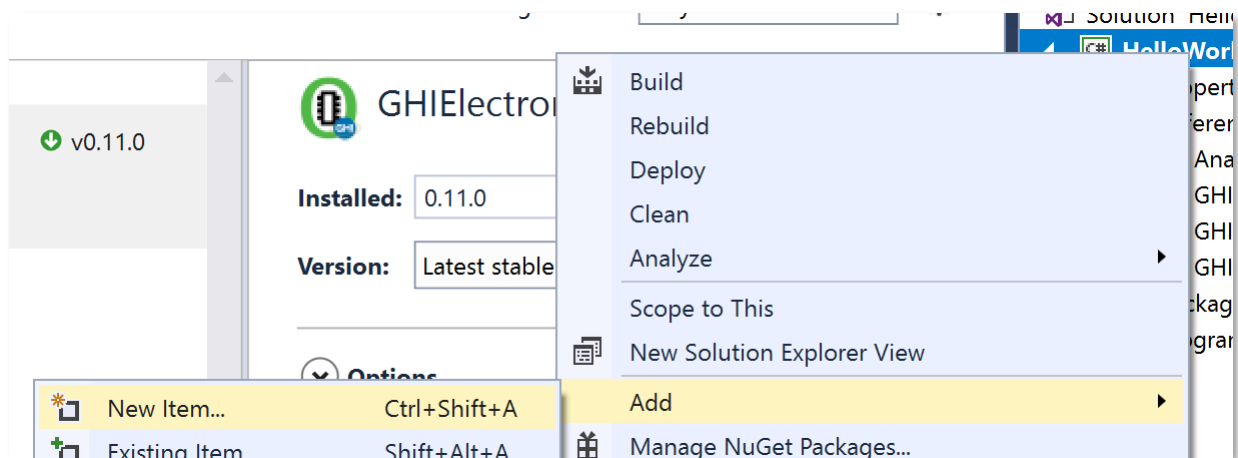
From the available options, select BrainPad helper and click the Add button.
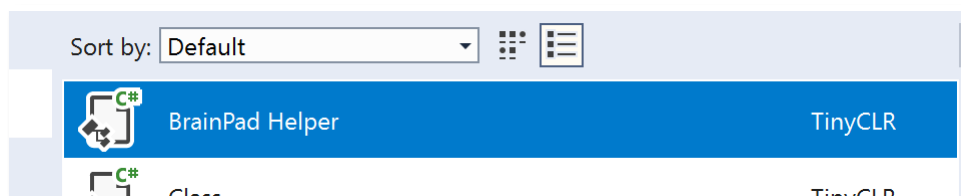
The fun begins now! Go back to Program.cs, where we added the counter code before. Change the code to show the counter on the BrainPad display.

```
namespace HelloWorld
{
    class Program
    {
```

```
        static void Main()
        {
            var count = 0;

            while (true)
            {
                BrainPad.Display.DrawSmallText(0, 0, "Count: " + count);
                BrainPad.Display.RefreshScreen();
                BrainPad.Wait.Seconds(1);

                count = count + 1;
            }
        }
    }
}
```

The line that prints the counter is similar to the Debug.WriteLine command we used before, except this time it is printing to the BrainPad display. The second line is needed to refresh the screen. Displays are much slower than processors. If we were to refresh the display every time we draw, the display would run very slowly. Instead, the processor draws to its memory. Only when RefreshScreen is called is the display refreshed/updated. Think of a video game where you need to draw a ship, obstacles, enemies…etc. You will do all this drawing to memory only, which is fast, and only refresh the display as necessary.

The last line is to ask the BrainPad to wait one second.

BOUNCY BALL

The BrainPad libraries make everything look more "English" than code. Add some logic and you can create amazing programs. Teaching C#/Visual Basic is not covered in this book but that doesn't mean we can't have a bit of fun!

Let's go back to the original program and draw a 5-pixel diameter circle. We will draw it at location 30,30. Displays on computer systems start at the top left corner, which is location 0,0. Moving things to the right is done by increasing x, the first number. 30,0 describes a point 30 pixels from the left edge of the screen at the very top of the screen.

```
namespace HelloWorld
{
    class Program
    {
        static void Main()
        {
            var x = 30;
```

```
                var y = 30;

                while (true)
                {
                    BrainPad.Display.DrawCircle(x, y, 5);
                    BrainPad.Display.RefreshScreen();
                    BrainPad.Wait.Seconds(1);
                }
            }
        }
    }
}
```

And we have a circle!

In our endless loop, we will increase the x and y variables in every loop. Since we started at 30, the variable will go 31, 32, 33, 34…etc. Are things getting more exciting for you?

The circle moved very slowly because of the one second delay. We will change that to 0.01. We will also add some logic to bounce the circle/ball back. To bounce back, we need to have new variables, dx and dy. Those are basically holding the direction the ball is going. We then check if the ball is at an edge. If so, we reverse direction.

```
namespace HelloWorld
{
    class Program
    {
        static void Main()
        {
            var x = 30;
            var y = 30;
            var dx = 1;
            var dy = 1;

            while (true)
            {
                x = x + dx;
                y = y + dy;

                if (x < 0 || x > BrainPad.Display.Width)
                {
                    dx = dx * -1;
                }

                if (y < 0 || y > BrainPad.Display.Height)
                {
```

```
                dy = dy * -1;
            }

            BrainPad.Display.DrawCircle(x, y, 5);
            BrainPad.Display.RefreshScreen();
            BrainPad.Wait.Seconds(0.01);
        }
    }
  }
}
```

The circle being redrawn as it moves across screen:
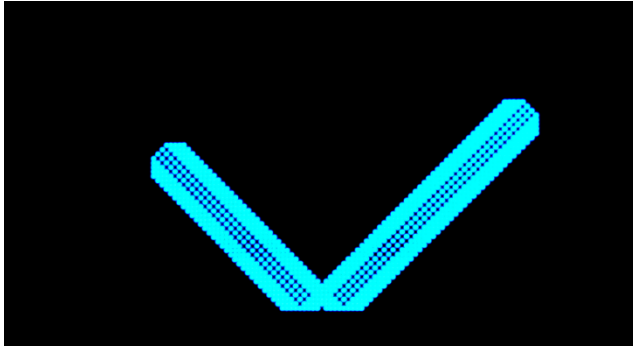
But we want to have a bouncing ball so let us clear the screen before we draw a circle. Add this line right before DrawCircle.

```
 BrainPad.Display.Clear();
```

Do you want to speed it up some more? The delay we have of 0.01 is very small so the delay is not the problem here. We are only moving the circle one pixel in every loop. Move it 5 instead and it will be 5 times faster. This is accomplished by changing the dx and dy initial value to 5.

```
 var dx = 5;
 var dy = 5;
```

What ball bounces without making noise? Have the BrainPad beep every time the ball bounces. Add this line inside each if statement.

```
 BrainPad.Buzzer.Beep();
```

Aha! Now this is a bouncing ball! But this is too loud, especially if you are in a classroom with 30 BrainPads! Add a check to only make sounds if the down button is pressed.

```
 if (BrainPad.Buttons.IsDownPressed())
 {
     BrainPad.Buzzer.Beep();
 }
```

Can you give the ball a cool effect by making get larger and smaller while moving? Think of how we change the ball position within set boundaries. You will do the exact same thing. You need a variable r for radius and a variable dr for direction of radius, growing or shrinking.

```
var r = 3;
var dr = 1;
```

We will then increase the size with boundaries between 1 and 8.

```
r = r + dr;

if (r < 1 || r > 8)
{
    dr = dr * -1;

}
```

Of course, we need to change the `DrawCircle` line to use the variable r instead of 5.

```
BrainPad.Display.DrawCircle(x, y, r);
```

Here is the complete code. But before you copy and paste the code, if you have named your project something other than HelloWorld (no space in between words, case sensitive) then you will have a different default name space. You should keep the default name space of your code and not copy the one from below. For example, if your code shows

```
namespace SomethingElse
```

Then keep the namespace line but copy the rest of the code, everything starting from `class Program` and ending with the second to last right brace(`}`). The last brace is actually part of the namespace.

```
namespace HelloWorld
{
    class Program
    {
        static void Main()
        {
            var x = 30;
            var y = 30;
            var dx = 5;
            var dy = 5;
            var r = 3;
            var dr = 1;

            while (true)
            {
                x = x + dx;
                y = y + dy;

                if (x < 0 || x > BrainPad.Display.Width)
                {
                    if (BrainPad.Buttons.IsDownPressed())
                    {
                        BrainPad.Buzzer.Beep();
                    }

                    dx = dx * -1;
                }

                if (y < 0 || y > BrainPad.Display.Height)
```

```
                {
                    if (BrainPad.Buttons.IsDownPressed())
                    {
                        BrainPad.Buzzer.Beep();
                    }

                    dy = dy * -1;
                }

                r = r + dr;

                if (r < 1 || r > 8)
                {
                    dr = dr * -1;
                }

                BrainPad.Display.Clear();
                BrainPad.Display.DrawCircle(x, y, r);
                BrainPad.Display.RefreshScreen();
                BrainPad.Wait.Seconds(0.01);
            }
        }
    }
}
```

A CHRISTMAS LIGHT

You read that title correctly. We will be creating a Christmas light, as in a single light bulb. For this we will utilize one of the thousands of built in functions found in TinyCLR OS. This function gives us a random number. Unlike real life, in a computer nothing is truly random. This can make it difficult to generate a random number. Luckily, there is a built in random number function that makes it easy. We just create a random object and then ask it for the next random value.

We will use those random numbers to set the Light Bulb primary colors of red, green, and blue. In case you didn't know, those three colors can create any color you want.

```
static void Main()
{
    var rnd = new Random();

    while (true)
    {
        BrainPad.LightBulb.TurnColor(rnd.Next(100), rnd.Next(100), rnd.Next(100));
        BrainPad.Wait.Seconds(0.1);
    }
}
```

SEEING THE LIGHT

We want to set a plant somewhere indoors, and we are curious about how much light that corner gets. We want to see summary of the light level throughout the day in the form of a chart.

The chart is very simple. We will basically draw a vertical line that starts where the level is and ends at the very bottom of the display. The line will shift one pixel to the right in every loop.

```
static void Main()
{
    var x = 300;

    while (true)
    {
        x++;

        if (x > BrainPad.Display.Width)
        {
            x = 0;

            BrainPad.Display.Clear();
            BrainPad.Display.DrawSmallText(30, 0, "Light Level");
        }

        var light = BrainPad.LightSensor.ReadLightLevel() / 2;
        var y = BrainPad.Display.Height – light;

        BrainPad.Display.DrawLine(x, y, x, BrainPad.Display.Height);
        BrainPad.Display.RefreshScreen();
        BrainPad.Wait.Seconds(0.1);
    }
}
```

You probably want the program to pause when it reaches the far right, but in this example we will clear the display and start from the beginning. Let's run the loop quickly so we can see immediate results. If you want to graph two hours, and the display is 128 pixels wide, you will need a one-minute delay in every step. This will record 128 minutes across the width of the display.

You can also keep the program as is and cover the light sensor with your hand to make some cool drawings!

Since we are expert coders by now, can you guess the answers to these questions?

1. Can you guess why we are dividing the light level by 2?
2. Can you guess why we set the start line to `BrainPad.Display.Height - light`?
3. Can you guess why x is initialized with 300? What happens if used 0 instead?

Here are the answers but promise me you will try on your own first. The light level is anywhere 0 to 100. The display is 64 pixels high. If we divide the level in half, the max value will be 100 / 2, or 50. With a 64 pixel high display, we will have 14 pixels available at the top of the screen. This is where we print "Light Level."

The second answer relates to how we want to show increasing light levels. Smaller y-coordinates are shown near the top of the display, but we want lower light levels to appear near the bottom of the display. By subtracting the light level from the height of the display, we are inverting the graph.

The last one is a cool trick to help when the code is first run. We only print Light Level when we clear the screen. So, by setting x to a large value, we will automatically be outside the boundary we had set and "Light Level" will get printed on the screen. Try changing it to 0 and light level will not show on the screen the first time the graph is plotted. It will only show when the display gets refreshed after drawing the first chart. You could just print it in every loop, but this will slow down your program. We should always think smart when we code and only do what is necessary.

MULTITASKING

Asking a system to do multiple things at once is very complex internally. Typically, it's not user-friendly on small systems. The good news is that TinyCLR OS allows for multitasking and it's easy to use. It works exactly how it would on any larger system programmed in .NET. Multitasking is called threading, where each thread is an individual task running part of the program.

Not sure why you would need threading? How would you count on the Display while blinking the Light Bulb? Sure, if you want to increment the count with every light blink then it is easy.

```
static void Main()
{
    var count = 0;

    while (true)
    {
        BrainPad.LightBulb.TurnGreen();
        BrainPad.Wait.Seconds(0.1);
        BrainPad.LightBulb.TurnOff();
        BrainPad.Wait.Seconds(0.9);

        BrainPad.Display.DrawSmallText(0, 0, "Count: " + count);
        BrainPad.Display.RefreshScreen();

        count = count + 1;
    }
}
```

Now, change the program to keep blinking once a second but count as fast as possible. You really can't without threading. Start by moving the blinking code into its own method. A method is a piece of code that does a specific task. You call that method to handle that task. For example, the LightBulb has a method called TurnGreen to make it green.

```
static void Blink()
{
    while (true)
    {
```

```
        BrainPad.LightBulb.TurnGreen();
        BrainPad.Wait.Seconds(0.1);
        BrainPad.LightBulb.TurnOff();
        BrainPad.Wait.Seconds(0.9);
    }

}
```

If you call that method, it will blink the light once a second; however, it will never return due to its while-loop. That's okay because we will call that method from a separate thread. Are you ready for the complex code to create a thread? Here it is!

```
new Thread(Blink).Start();
```

We simply told the system to create a new thread for the method Blink and then start that thread.

STOP! Before running the code, we need one last change. Every thread in the system must have a sleep in it. This helps the system in processing its internal tasks and lets the other threads run smoothly. But you may say that we only have one thread. We really have two threads. The system automatically creates a thread internally that runs Main(), which is the main method to run. If you need to run as fast as possible, just add the minimum required delay.

```
BrainPad.Wait.Minimum();
```

Here are both methods, Main and Blinking.

```
static void Blink()
{
    while (true)
    {
        BrainPad.LightBulb.TurnGreen();
        BrainPad.Wait.Seconds(0.1);
        BrainPad.LightBulb.TurnOff();
        BrainPad.Wait.Seconds(0.9);
    }
}

static void Main()
{
    new Thread(Blink).Start();

    var count = 0;

    while (true)
    {
        BrainPad.Display.DrawSmallText(0, 0, "Count: " + count);
        BrainPad.Display.RefreshScreen();
        BrainPad.Wait.Minimum();

        count = count + 1;
    }
}
```

There are two endless loops and they are both running simultaneously. While this is a very simple example, it shows the possibilities. As an exercise, add another thread to beep every three seconds.

We have used the buttons already, so you know how they work. However, we have always used them in a loop where we checked if the button was pressed. The system may check if the button is pressed millions of times before it is finally pressed. This is very bad for battery operated devices. A good design will go to sleep when it is not handling any tasks to conserve on battery usage. If your phone didn't sleep, it wouldn't last an hour before it needed to be charged again.

This subject can get quite intense, so we will only cover button events. Instead of constantly checking the buttons, we will ask the system to call us when a button is pressed. We want to make a beep sound every time the up button is pressed. We will blink the Light Bulb as well.

```
static void Main()
{
    while (true)
    {
        BrainPad.LightBulb.TurnGreen();
        BrainPad.Wait.Seconds(0.1);
        BrainPad.LightBulb.TurnOff();
        BrainPad.Wait.Seconds(0.9);

        if (BrainPad.Buttons.IsUpPressed())
        {
            BrainPad.Buzzer.Beep();
        }
    }
}
```

Press the up button and nothing will happen. Hold it down and it will beep once a second. This is expected as the loop has delays and the button is checked once a second. You can decrease the delay, but this will still not solve the problem. Instead, we will now use events. Visual Studio can automatically generate the code needed. Start by typing "BrainPad.Buttons.WhenUpButtonPressed" and then add "+=" after. You can now hit the tab key to generate the event method.
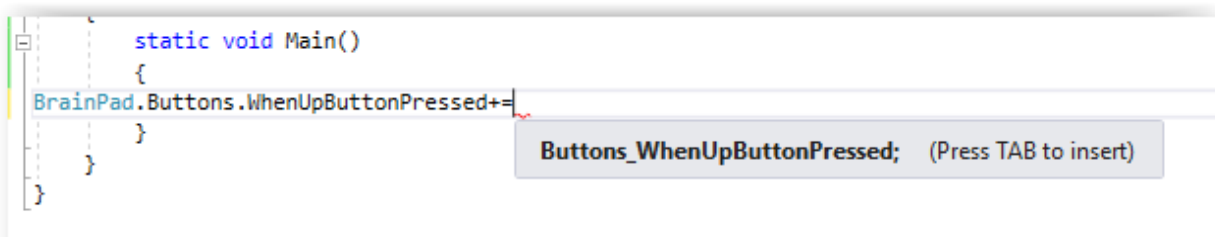
The generated event method looks like this:

```
private static void Buttons_WhenUpButtonPressed()
{
    throw new NotImplementedException();
}
```

The generated code has a line to generate an error (exception). This is only there so we would not forget about it. Replace that line with a beep.

```
private static void Buttons_WhenUpButtonPressed()
{
    BrainPad.Buzzer.Beep();

}
```

Do not forget to remove the code that checks the button in the main loop.

```
static void Main()
{
    BrainPad.Buttons.WhenUpButtonPressed += Buttons_WhenUpButtonPressed;

    while (true)
    {
        BrainPad.LightBulb.TurnGreen();
        BrainPad.Wait.Seconds(0.1);
        BrainPad.LightBulb.TurnOff();
        BrainPad.Wait.Seconds(0.9);
    }
}

private static void Buttons_WhenUpButtonPressed()
{
    BrainPad.Buzzer.Beep();

}
```

By the way, multiple events can call the same event handler. Here is code that beeps when any one of the four buttons is pressed.

```
static void Main()
{
    BrainPad.Buttons.WhenUpButtonPressed += Beeper;
    BrainPad.Buttons.WhenDownButtonPressed += Beeper;
    BrainPad.Buttons.WhenLeftButtonPressed += Beeper;
    BrainPad.Buttons.WhenRightButtonPressed += Beeper;

    while (true)
    {
        BrainPad.LightBulb.TurnGreen();
        BrainPad.Wait.Seconds(0.1);
        BrainPad.LightBulb.TurnOff();
        BrainPad.Wait.Seconds(0.9);
    }
}

private static void Beeper()
{
    BrainPad.Buzzer.Beep();

}
```

SCARY WIRES

It is normal for anyone new to electronics to be intimidated and scared by the idea of wiring things. What if I get electrocuted? What if damage the circuit?

The BrainPad makes this less scary. First, the BrainPad runs off a very low voltage of 5 volts (5V). You can't get hurt by touching 5V. Also, the BrainPad has built in protection and is designed to make it very difficult to damage it by wiring something incorrectly. Still, you should understand what you are doing. There are many courses and online articles that will guide you through understanding circuits.
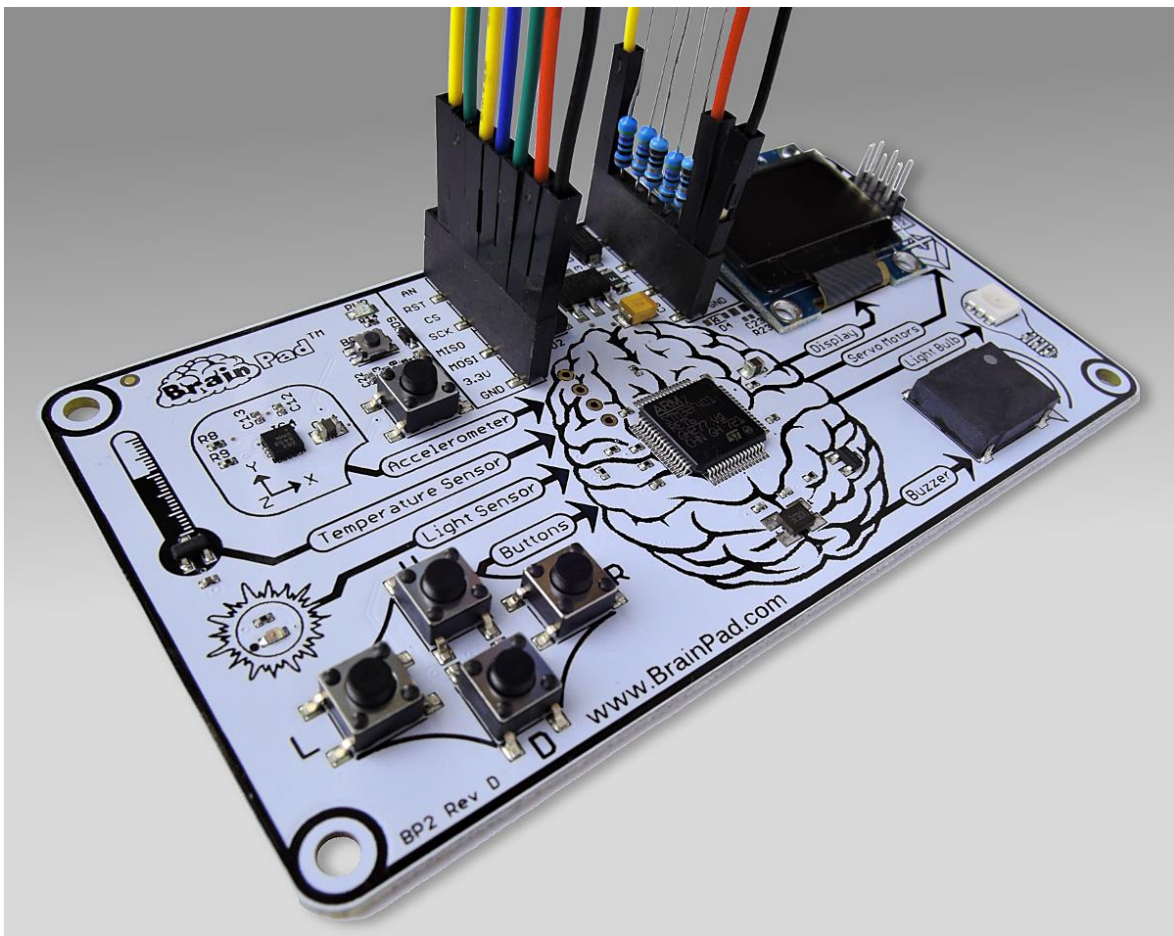

As mentioned in the first chapter of this book, there are multiple ways to expand the BrainPad, thanks to the female headers located near the USB connector.
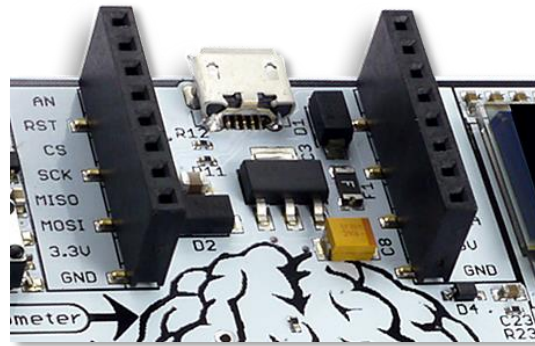
Those pins have many functions and we would need several books to explain completely. We will scratch the surface here by connecting a simple LED (Light Emitting Diode.)


LEDs are very inexpensive and come in different colors. They have two leads where one should be longer than the other. Some LEDs have more than one color (like the BrainPad's Light Bulb.) We are not covering those LEDs. To control an LED a current-limiting resistor is needed. Some common values are 220, 330, or 470 ohms. You will also need some wires and a breadboard.
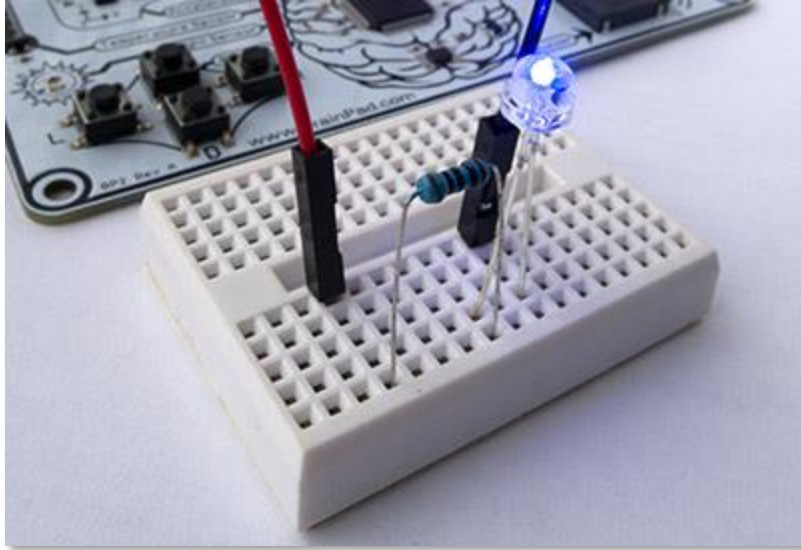
We can use one of the GPIO pins on the expansion header to control the LED. GPIO means General Purpose Input Output. We will use pin labeled PWM.
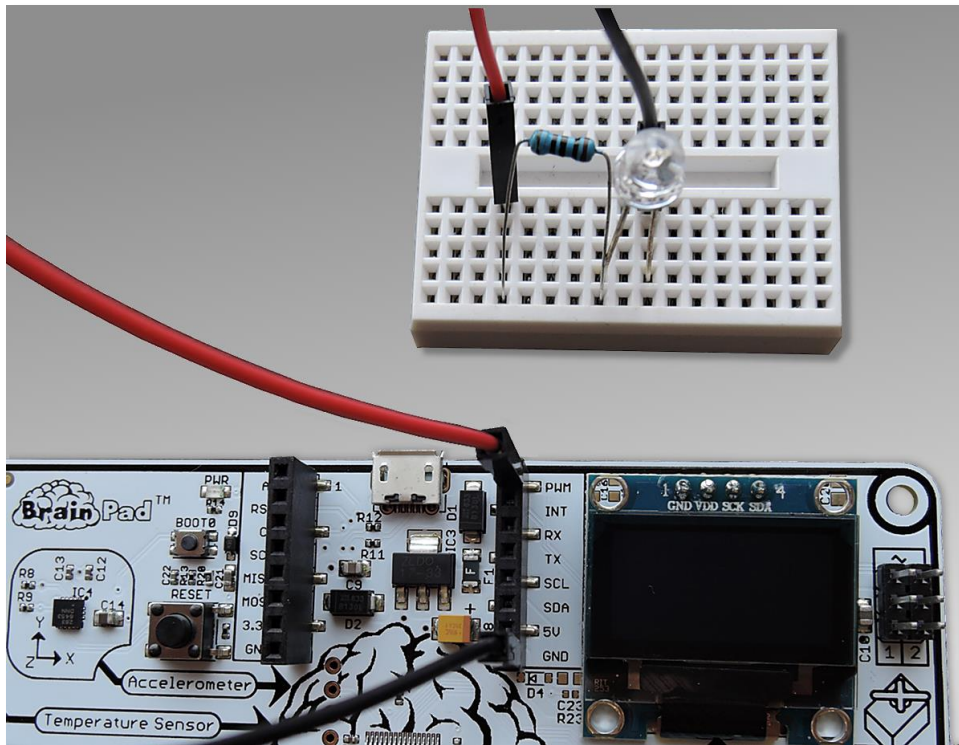

Bread-boards have internal connections where every row of holes is electrically connected. In the image below, the breadboard on the right side has black lines showing the internal electrical connections. Anything plugged into the holes connected by a black line will be electrically connected.
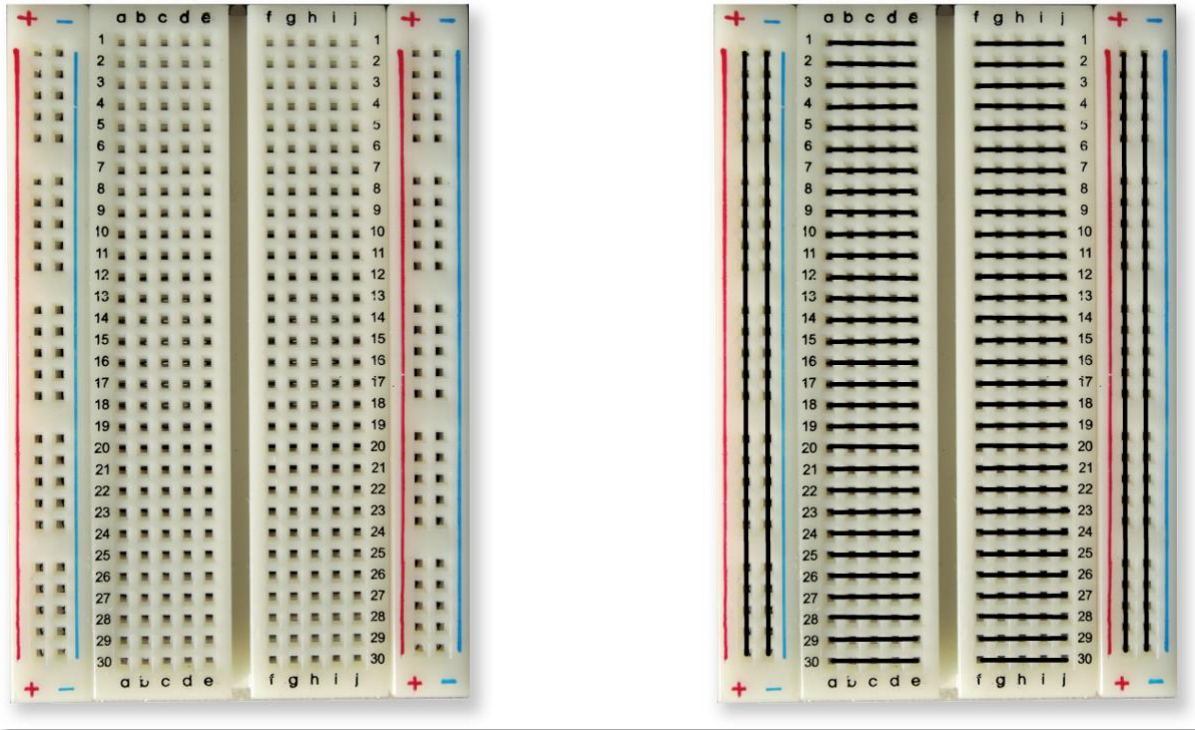
A wire will go from the PWM pin to the breadboard where the long lead from the LED is plugged in. The short lead from the LED will connect to the resistor, and the resistor to the GND (ground) pin. This forms a circle (or circuit) where electrons will have a path between PWM and GND. When the PWM pin is activated, the LED will light up.
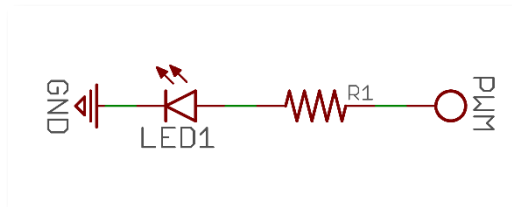
To control the PWM pin, which is a GPIO, we will need to include the GHIElectronics.TinyCLR.Devices and GHIElectronics.TinyCLR.Pins NuGet libraries. These libraries are needed by the GHIElectronics.TinyCLR.BrainPad library we always add, so no further action is needed as they are automatically added.
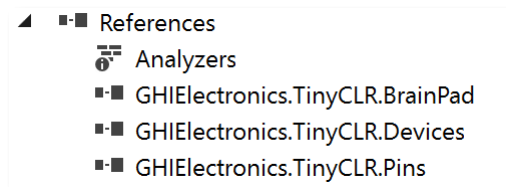
We will need to have access to the GPIO libraries in our code. This is done with one line of code.

```
using GHIElectronics.TinyCLR.Devices.Gpio;
```

We need to access a pin through the GPIO controller that is found inside the processor.

```
var controller = GpioController.GetDefault();
var pwmPin = controller.OpenPin(BrainPad.Expansion.GpioPin.Pwm);
```

Pins can be inputs or outputs, just like the inputs and outputs on the BrainPad. Inputs go into the "brain" and outputs come out of the "brain". An input example is a button, like the buttons found on the BrainPad. An output example is an LED, like the Light Bulb. So the PWM pin must be an output.

```
pwmPin.SetDriveMode(GpioPinDriveMode.Output);
```

We can now simple "write" to this pin, to make it High (activate) or Low (deactivate). Put that in a loop with some delays and you have a blinking LED.

```
while (true)
{
    pwmPin.Write(GpioPinValue.High);
    BrainPad.Wait.Seconds(0.1);
    pwmPin.Write(GpioPinValue.Low);
    BrainPad.Wait.Seconds(0.5);
}
```

The complete main method code should look like this:

```
static void Main()
{
    var controller = GpioController.GetDefault();
    var pwmPin = controller.OpenPin(BrainPad.Expansion.GpioPin.Pwm);

    pwmPin.SetDriveMode(GpioPinDriveMode.Output);

    while (true)
    {
        pwmPin.Write(GpioPinValue.High);
        BrainPad.Wait.Seconds(0.1);
        pwmPin.Write(GpioPinValue.Low);
        BrainPad.Wait.Seconds(0.5);
    }
}
```

Connecting a button is similar, except the pin will need to be an input instead of output. Without getting into a lot of details, the pin needs to be an input with pull up. This keeps the pin high until the button is pressed.

```
pwmPin.SetDriveMode(GpioPinDriveMode.InputPullUp);
```

No need to add any resistors to the button. Simply wire the button between one of the GPIOs and GND.

CONCLUSION

TinyCLR OS from GHI Electronics brings truly professional programming to the BrainPad. The knowledge you will gain is beneficial in coding any device, from phones to computers. This option is made easy to use thanks to our TinyCLR Operating System and Microsoft's .NET Framework.

EXPANDABILITY

The expansion headers open a whole world of possibilities. Here we list some options to get you thinking about what's possible.

MIKROELEKTRONIKA CLICK BOARDS™

MikroElektronika offers hundreds of small modules that plug right onto the BrainPad. Modules include simple sensors like humidity and temperature to more sophisticated devices like voice recognition and electrocardiogram modules. There are also many actuator and control modules for things like motor control and digital lighting control. You can find them all on their website at https://www.mikroe.com/click.

In this sample project, we've made a linear clock that uses a Real Time Clock (RTC) click module to keep track of time even when the BrainPad is not powered (https://docs.brainpad.com/projects/linear-clock.html).
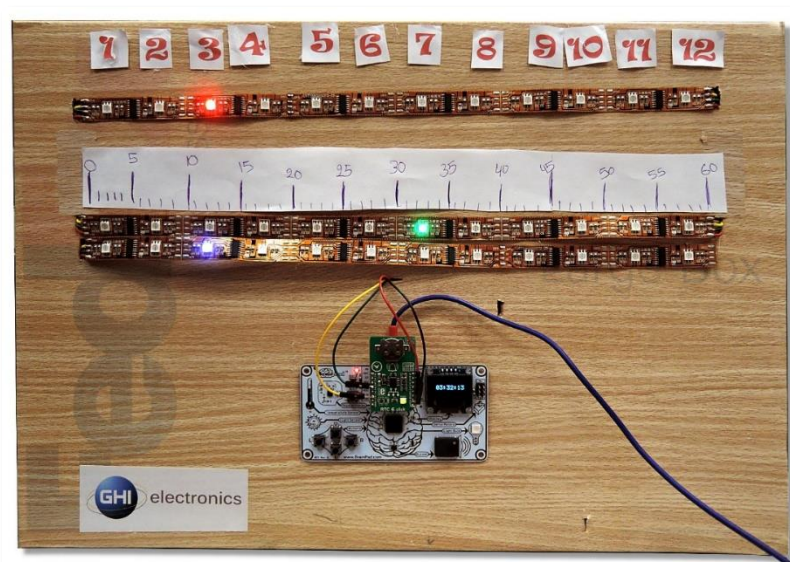
ELENCO® SNAP CIRCUITS®
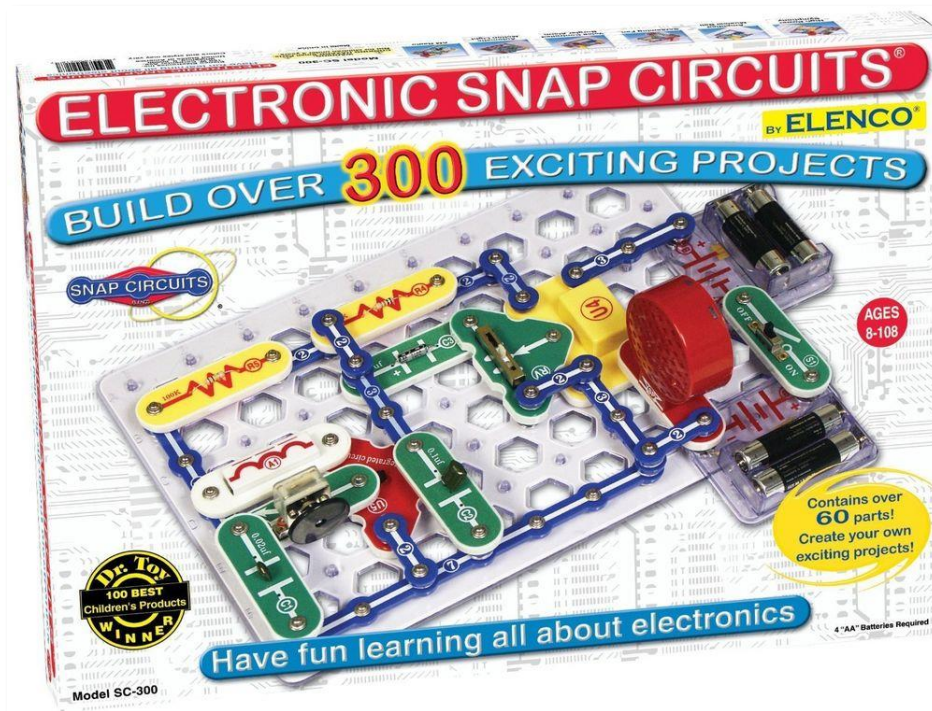
One of the most popular electronics learning platforms is called Snap Circuits, made by Elenco

(https://www.elenco.com). These very popular kits can be found at local and online retailers and include a variety of electronic components and instructions for building many different projects.

With the BrainPad, you can add intelligence and programming to your electronics projects. This sample project is a countdown timer that launches a propeller. Make sure you watch the video. It is one of our favorites (https://docs.brainpad.com/projects/lift-off.html).
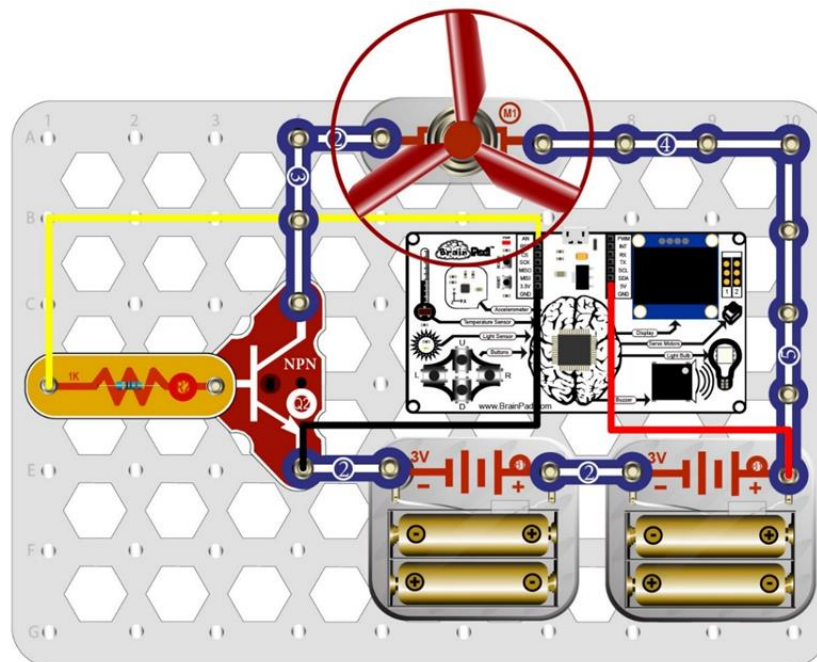
BREADBOARDS

Want to design your own electronic circuits from scratch just like engineers do? There are many ways to create circuits that work with the BrainPad. While designing circuits requires a little more knowledge, it is the most creative and least expensive way to expand your BrainPad. It is also the most educational.

The easiest way to get started making your own circuits is to use solderless breadboards. Wire and components simply plug into the holes in the breadboard. The breadboard holds the components and connects them together electrically. Not only is this the fastest way to test a new circuit, but correcting mistakes is simple as well.
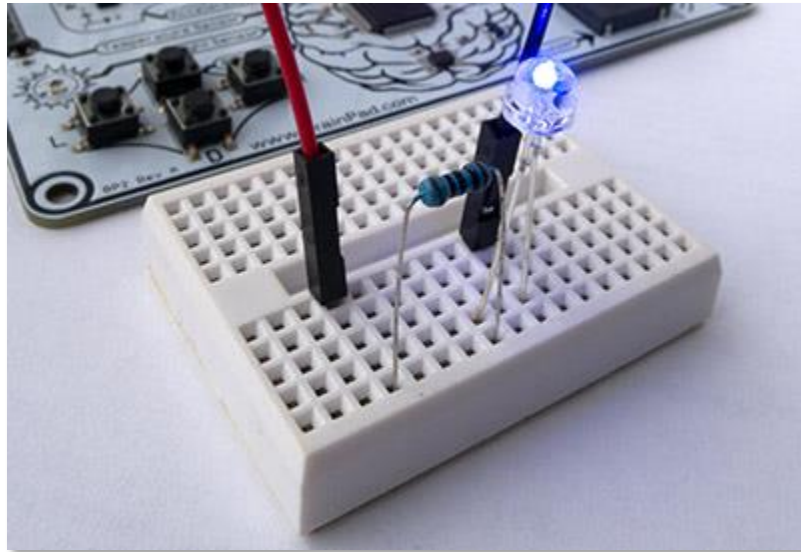
If you have a circuit that you would like to make more permanent, there are several options from easy to use premade prototyping boards to designing your own circuit boards. Circuit board designs can be drawn by hand or drawn using free computer-based design tools. This is what makes the BrainPad so special – it is simple to get started, but you can advance as far as you like while learning at your own pace.

You can also add one or more of many available sensors to your BrainPad project. By searching the web for "sensor kit," you can find many inexpensive sensor assortments. Here is one example:

Some knowledge is necessary for using these modules, however we have some easy to build projects that will get you moving in the right direction. Once you become familiar with one type of sensor, it becomes much easier to understand how other sensors work.

www.BrainPad.com