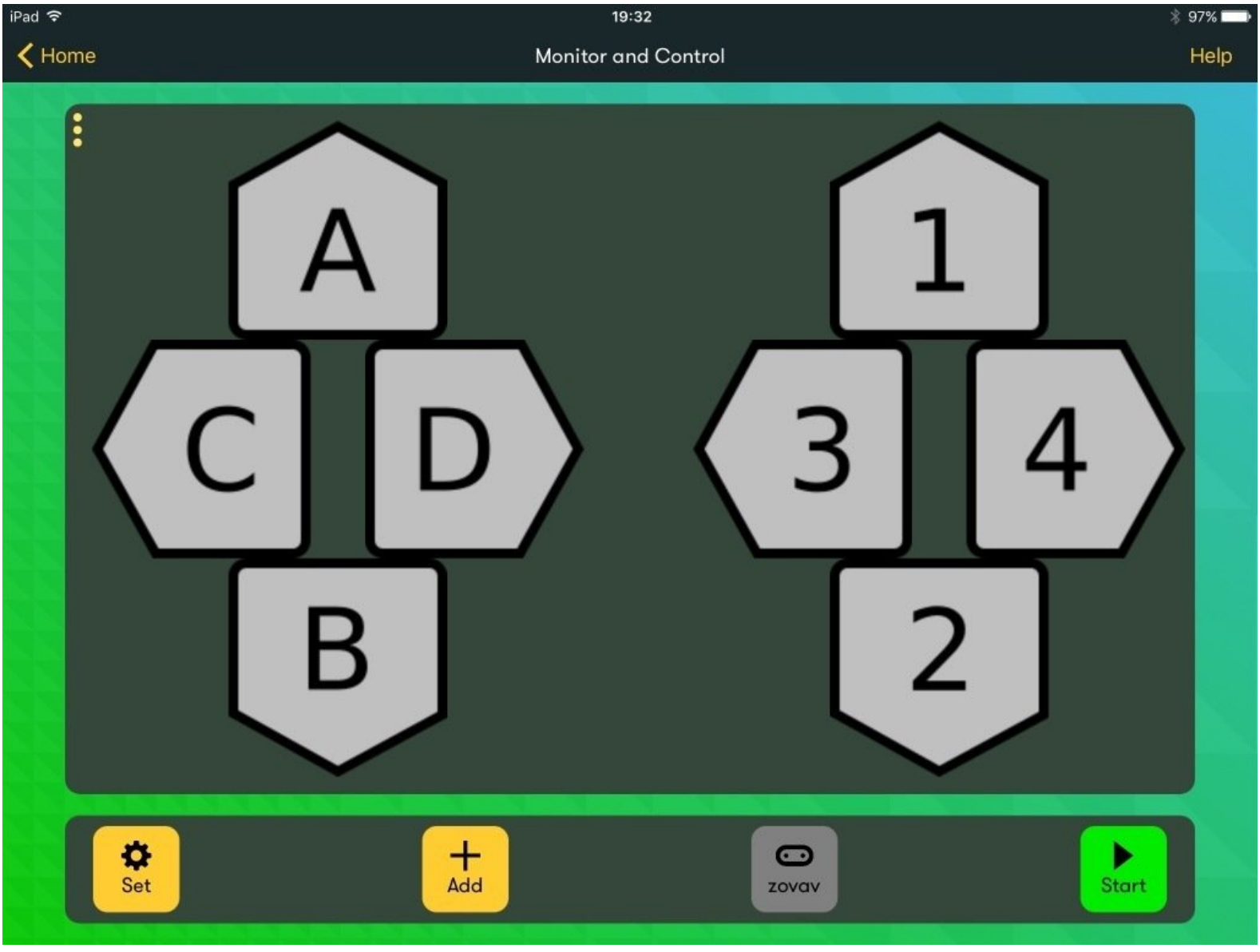
5 - Phone Control (iOS)

Use your iOS device and the micro:bit app to control your car!



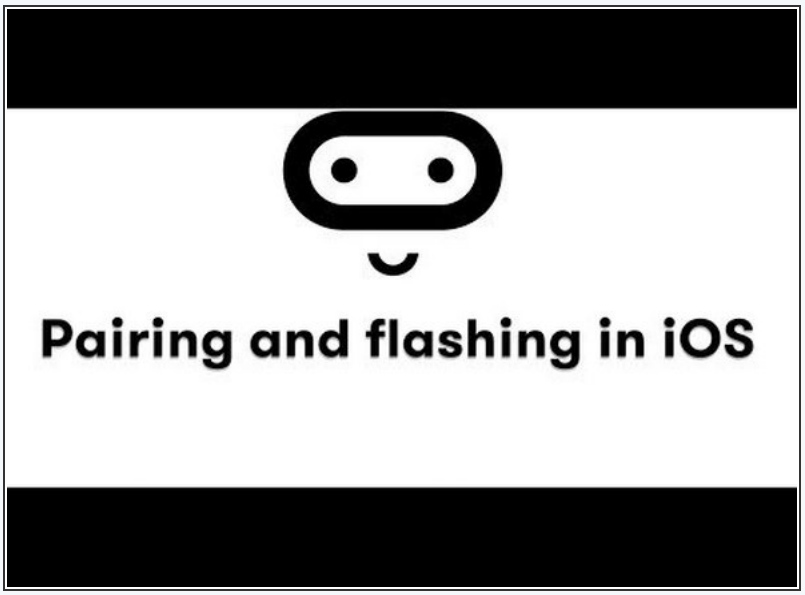
INTRODUCTION

Use your iOS device and the micro:bit app to control your car!

Step 1

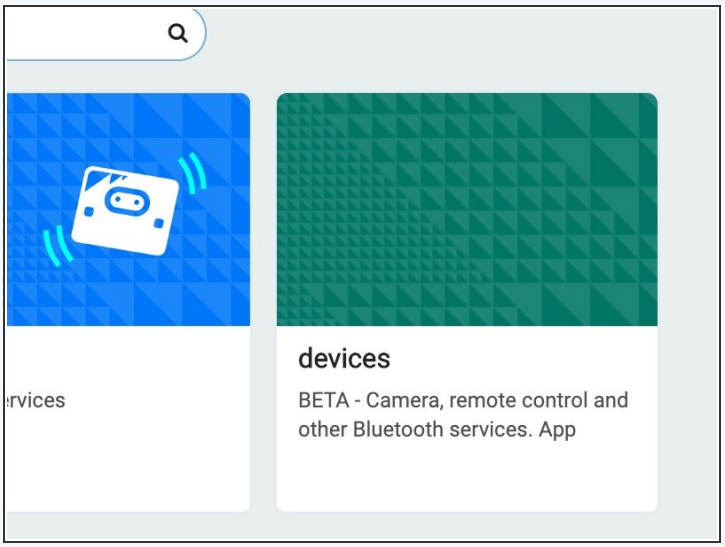
Pairing your device

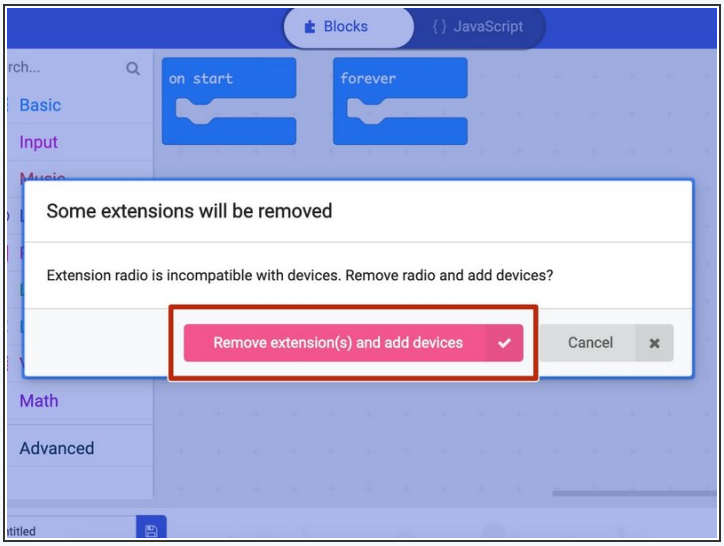
- To control the robot using your phone, we'll be using the micro:bit's internal bluetooth module.
- Follow the video to learn how to pair your micro:bit to your phone!
- You'll also need to install this app. (https://apps.apple.com/gb/app/micro-bit/id1092687276)



Step 2

Installing the extensions





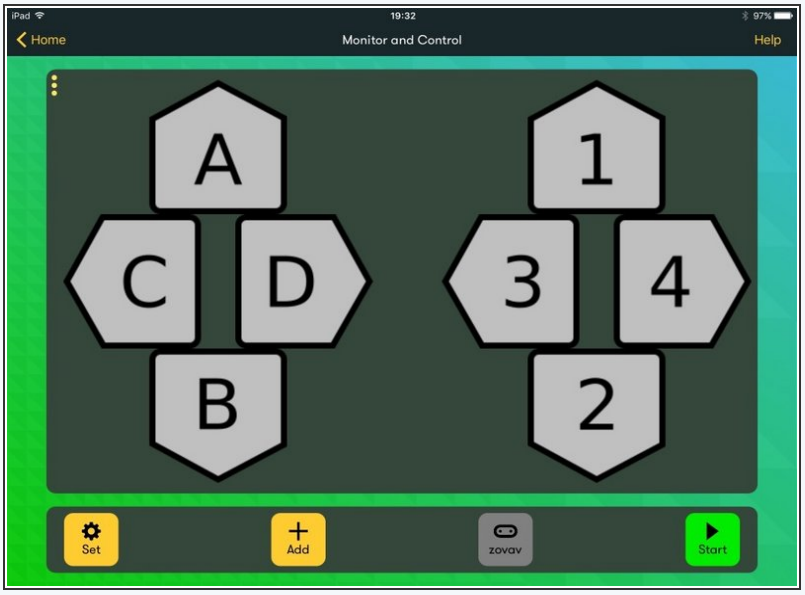
- Like before, we need to install an extension so that the micro:bit knows how to work with our phone.
- Install the "devices" extension by clicking on it.
- When you get this message, click "Remove extension(s) and add devices".

⚠ Make sure you're using the https://www.techcamp.org.uk/invent (https://www.techcamp.org.uk/invent) version of MakeCode!

Step 3

Planning our code

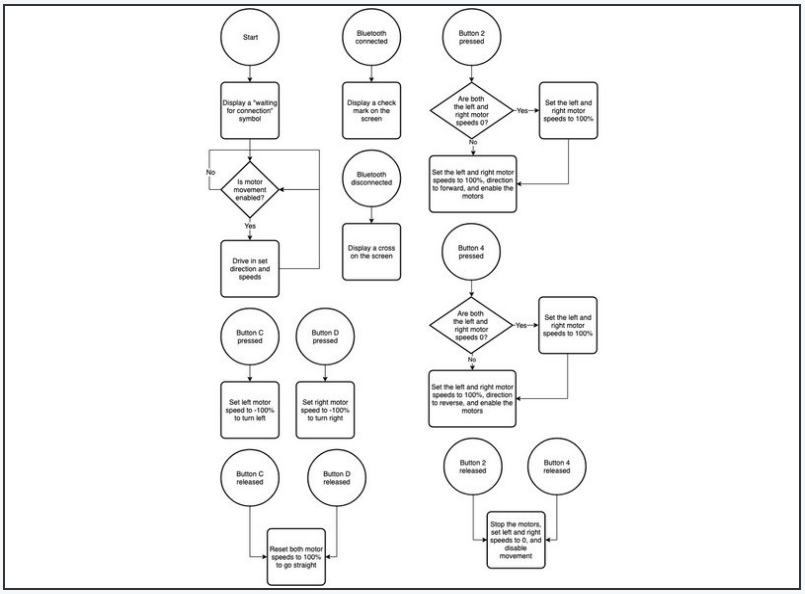
- When working on larger code projects like this one, it's always good to have a plan of what exactly your code will do.
- The micro:bit app has a simple game pad with a D-Pad and 4 buttons. In our code, we want the car to go forward as long as button 2 is held down and backwards when 4 is held down. C and D should make the car go left or right, but only if 2 or 4 is also pressed down.



Step 4

Flowcharting our code

- Now we know *what* we want our code to do, it's good to flowchart our code so we know the exact process to get there.
- In a flowchart, a circle represents an "event" in our code, like a button being pressed or even the program starting, a square represents a process, such as moving forwards, and a diamond represents a decision, just like an if statement.
- Arrows in a flowchart represent the flow of the program between actions.
- The image here is an overview of our program. Let's take a closer look at each section.

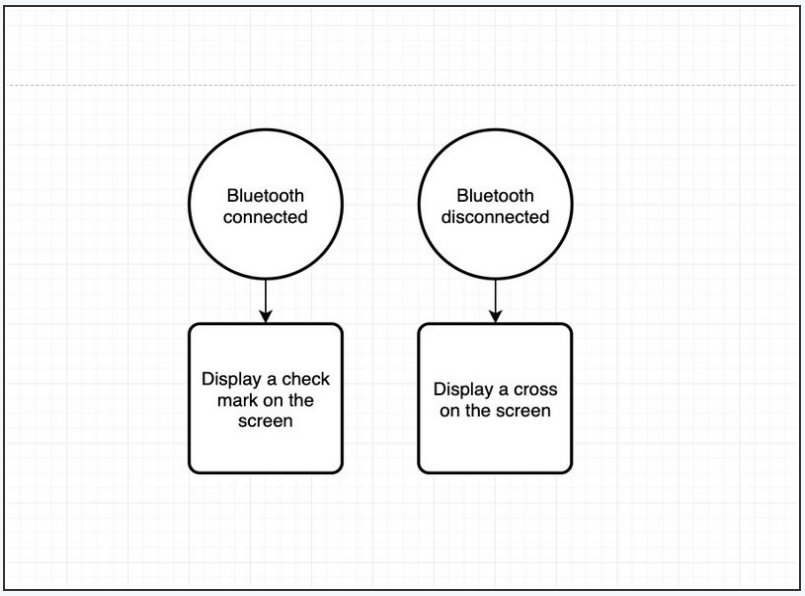


Step 5

Handling bluetooth

- It's useful to know whether our phone is actually connected to the car or not.
- We can put two *event handlers* in our program which will display a check mark if bluetooth is connected, or a cross if it is disconnected.

ⓘ In TypeScript, an event handler is simply a function to run when an event occurs.

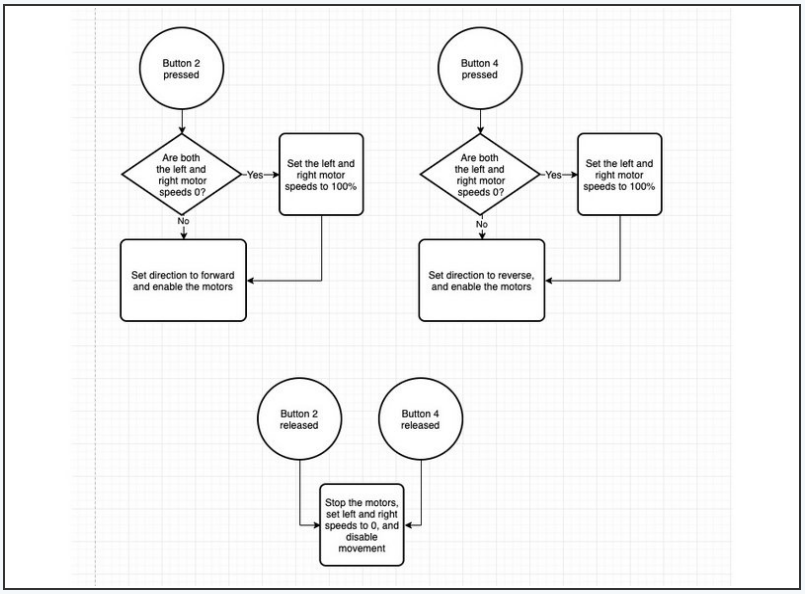


Step 6

Handling the forwards and backwards buttons

- If button 2 or 4 is pressed, we want to first check if the left and right speeds are both 0 (or off). If they are, set them both to 100% speed.
- Next, just set the direction to forwards or backwards and enable the motors.
- We also need code to run if the buttons are released, otherwise the car would go forwards or backwards forever! Here, we have it just set everything back to off if either button is released.

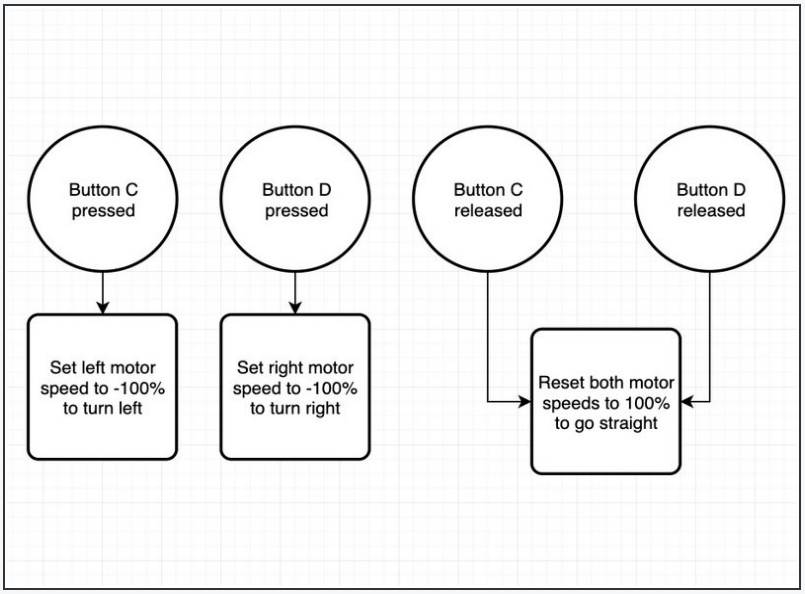
ⓘ Notice the amount of shared logic between the two events. This indicates that we'll want to use functions in our code to avoid repeating ourselves!



Step 7

Handling the left and right buttons

- Finally, we just need to handle the left and right logic. Luckily, this is much simpler.
- All we need to do is set the left and right speeds to go in the direction we need. This will cause the car to go left or right.
- If left or right is released, just set the direction to straight again.

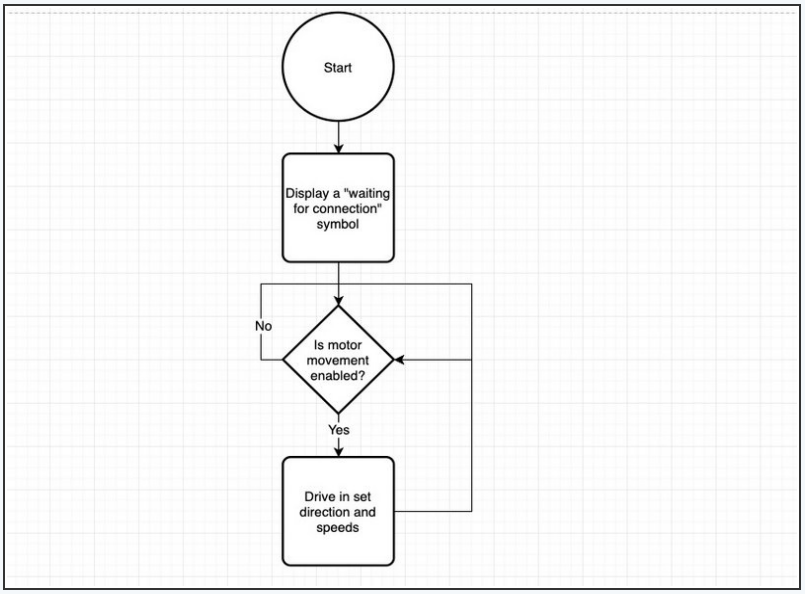


Step 8

Main program loop

- Since the rest of the code is modifying global variables, the main program loop is fairly simple.
- First, we display a "waiting for connection" symbol on the display (we'll use a horizontal line)
- Next, if the motors are enabled, just drive in the set direction and speeds. Otherwise, loop back and wait again.

Notice how the decision loops back on itself. This indicates that this code will go in our basic.forever loop!



Step 9

Writing the bluetooth connection handlers

- Now that we've planned out the program, let's actually write it!
- Let's start by writing the bluetooth connection display handlers.
- We can use the basic.showLeds function to draw out what we want to see on the screen, and the bluetooth.onBluetoothConnected function to display a check when bluetooth is connected.

```
// Waiting for connection
basic.showLeds(`
    . . . . .
    . . . . .
    # # # # #
    . . . . .
    . . . . .
    `)

// Connected to bluetooth
bluetooth.onBluetoothConnected(function () {
    basic.showLeds(`
        . . . . .
        . . . . #
        . . . # .
        # . # . .
        . # . . .
        `)
})

// Disconnected from bluetooth
bluetooth.onBluetoothDisconnected(function () {
    basic.showLeds(`
        # . . . #
        . # . # .
        . . # . .
        . # . # .
        # . . . #
        `)
})
```

Step 10

Writing the helper functions

- Next, we'll define our helper functions and variables. We define variables for the left and right motor speeds, whether the motors are enabled, and the current direction (forwards or backwards).
- In TypeScript, functions are defined using the function keyword, just like def in Python. Notice how our dir parameter has a second part. This is its *type*, and TypeScript needs to know this so it knows what is being passed to the function.

ⓘ You may recognise the function keyword from previous code, such as what we define inside the basic.forever loop. This is because when we work with those, we actually *define* a function which then gets passed to the basic.forever function. A function without a name is called an *anonymous* function, and they are incredibly useful!

ⓘ You can think of the basic.forever function as taking another function as a parameter, which it then will call forever. A function that takes a function as an argument is known as a *higher-order function*.

```
let left = 0;
let right = 0;
let enabled = false;
let direction = InventMotorDir.Forward;

function go(dir: InventMotorDir) {
    if (left === 0 && right === 0) {
        left = 100;
        right = 100;
    }

    direction = dir;
    enabled = true;
}

function stop() {
    left = 0;
    right = 0;
    enabled = false;
    invent.motor(InventMotor.Left, direction, left);
    invent.motor(InventMotor.Right, direction, right);
}
```

Step 11

Helper functions continued

- A variable type is simply a piece of information telling the computer *what* kind of information the variable holds, such as a number or string. Here, we tell the computer that the dir parameter is a `InventMotorDir`. If you tried to pass a string to this function, you'd get an error!

ⓘ TypeScript is smart enough to automatically determine types on variable assignment, which is why we can do let left = 0. Technically, the full version of this would be let left: number = 3 though.

- Notice that the logic of the go and stop functions follow what we defined in the flowchart.

```
let left = 0;
let right = 0;
let enabled = false;
let direction = InventMotorDir.Forward;

function go(dir: InventMotorDir) {
    if (left === 0 && right === 0) {
        left = 100;
        right = 100;
    }

    direction = dir;
    enabled = true;
}

function stop() {
    left = 0;
    right = 0;
    enabled = false;
    invent.motor(InventMotor.Left, direction, left);
    invent.motor(InventMotor.Right, direction, right);
}

```

Step 12

Writing our button handlers

- Now we can write the handlers for the buttons. We can use the devices.onGamepadButton function, which takes a button action and a function to call when the event occurs.

- Here, we have two handlers for the _2Down and _4Down events to go forwards and backwards respectively.

- We also have two handlers for CDown and DDown to set the left and right speeds to make the robot turn.

```
// Go forward if button 2 pressed
devices.onGamepadButton(MesDpadButtonInfo._2Down, function () {
    go(InventMotorDir.Forward);
});

// Go backwards if button 4 pressed
devices.onGamepadButton(MesDpadButtonInfo._4Down, function () {
    go(InventMotorDir.Reverse);
})

// Set travel direction to left if C is pressed
devices.onGamepadButton(MesDpadButtonInfo.CDown, function () {
    left = -100;
    right = 100;
})

// Set travel direction to right if D is pressed
devices.onGamepadButton(MesDpadButtonInfo.DDown, function () {
    left = 100;
    right = -100;
})
```

Step 13

Finishing the handlers

- Finally, we add code for the button up events (when the button is released), so that the car can reset itself.

```
76
77  // Stop the car if 2 or 4 is released
78  devices.onGamepadButton(MesDpadButtonInfo._2Up, function () {
79      stop();
80  })
81
82  devices.onGamepadButton(MesDpadButtonInfo._4Up, function () {
83      stop();
84  })
85
86  // Reset travel direction to straight if d-pad released
87  devices.onGamepadButton(MesDpadButtonInfo.CUp, function () {
88      left = 100;
89  })
90
91  devices.onGamepadButton(MesDpadButtonInfo.DUp, function () {
92      right = 100;
93  })
94
```

Step 14

Finishing our code

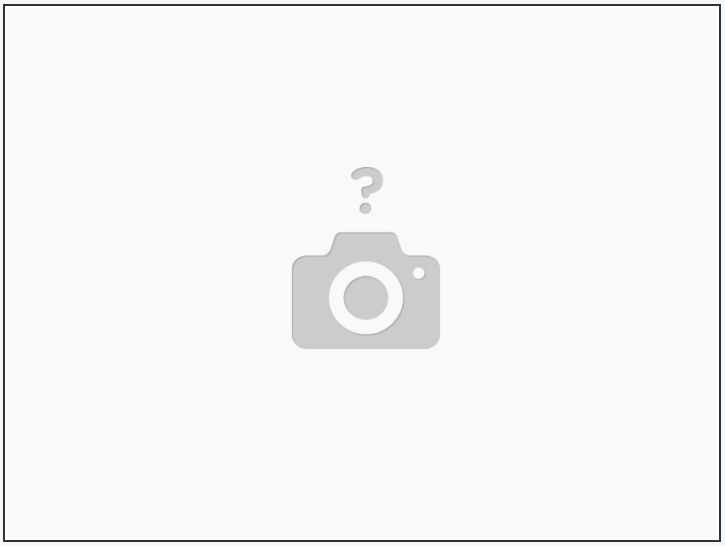
- To finish off our program, we just tell the motors to turn on if they're enabled, just like in the flowchart!
- Install the code to your micro:bit and get ready to connect your phone to it.

```
})

// Move if the motors are enabled by button 2 or 4
basic.forever(function () {
    if (enabled) {
        invent.motor(InventMotor.Left, direction, left);
        invent.motor(InventMotor.Right, direction, right);
    }
})
```

Step 15

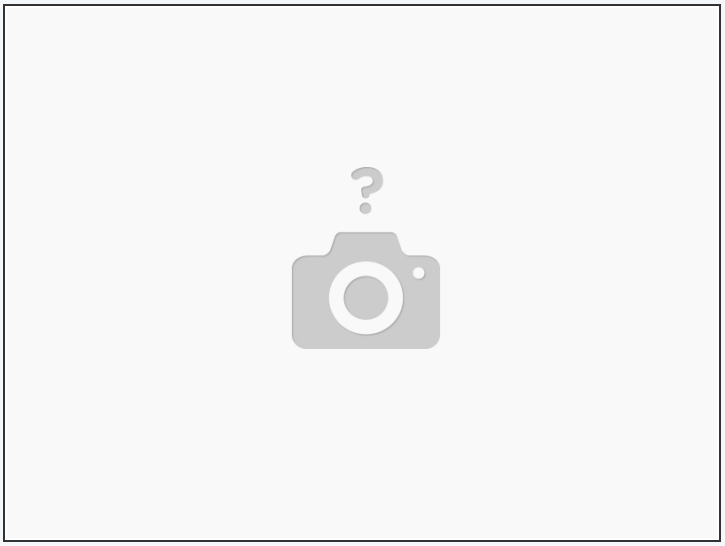
Connecting in app



- In the micro:bit app, go to "Monitor and Control"
- Press the Start button the the lower right corner to connect to the micro:bit (make sure it's on!)
- You'll also need to add a controller by going to "Add" and then selecting "Gamepad"

Step 16

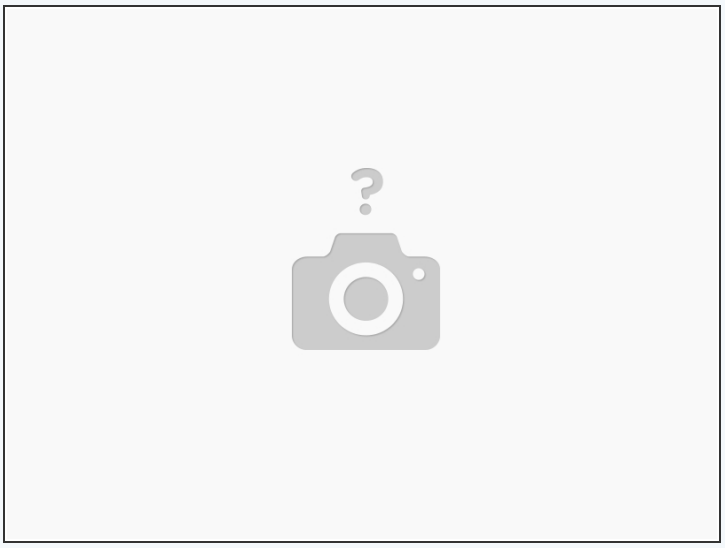
Controlling your car



- Press 2 to accelerate and 4 to reverse. You can use C and D to turn left and right!

Step 17

Challenge: add more features!



- We still have a bunch of buttons that have gone unused in the app!
- Can you think of new features to add? Maybe you could make the screen display a symbol with one button, or activate a buzzer as a horn!
- You can also experiment with the other monitor and control modules, which can read information such as temperature out (as long as you've coded it in!)