

---

# **Beak Squad Programming Manual**

***Release 0.1***

**Carson Rueter**

**Nov 23, 2022**



# CONTENTS

<b>1</b>	<b>Contents</b>	<b>3</b>
1.1	Section 0: WPILib & Vendor installation . . . . .	3
1.2	Section 1: Creating a Project . . . . .	4
1.3	Section 2: Your First Motor . . . . .	6
1.4	Section 3: Servos & Solenoids . . . . .	9
1.5	Section 4: Controllers . . . . .	11
1.6	Section 5: Encoders . . . . .	15
1.7	Section 6: Control - Limits & Sensing . . . . .	15
1.8	Section 7: Subsystems . . . . .	15
1.9	Section 8: Commands . . . . .	15
1.10	Section 9: Dashboards & Debugging . . . . .	15
1.11	Section 10: Cameras . . . . .	15
1.12	Section 11: PID Control . . . . .	15
1.13	The Basics of Wiring & Electronics . . . . .	15
1.14	Batteries . . . . .	16
1.15	BeakLib . . . . .	16



The Beak Squad Programming Manual (informally known as “Carson Professor”) is a complete Java, WPILib, and BeakLib how-to, from basic motor control to complete PID, command, and advanced-control robots.

---

**Note:** This manual is currently early in development.

---



**CONTENTS**

## **1.1 Section 0: WPILib & Vendor installation**

### **1.1.1 WPILib**

WPILib contains the WPILib Java library itself, several tools like dashboards, debugging utilities, extra robot communication, tools for system characterization, VS Code, and the JDK.

Instructions for installing WPILib can be found on the [WPILib Docs](#). Ensure to select “Everything” and “Download for this computer only”.

Additionally, the WPILib installer may be available offline on a flash drive. Ask a veteran member for help.

### **1.1.2 CTRE**

The CTRE Phoenix Framework contains offline copies of the Phoenix library for use in robot projects, as well as the Phoenix Tuner utility. The Phoenix Tuner utility gives the ability to configure, test, change CAN IDs, and log data from CTRE’s VictorSPX, TalonSRX, and TalonFX motor controllers.

The Phoenix Framework can be downloaded from [CTRE’s GitHub](#). Download the file for your computer’s hardware, and run the executable. From there, follow the onscreen instructions.

Additionally, the Phoenix Framework may be available offline on a flash drive. Ask a veteran member for help.

### **1.1.3 REV**

The REV Hardware Client is a utility for configuring, testing, changing CAN IDs, and logging data from REV Spark MAX motor controllers.

The REV Hardware Client can be downloaded from [REV’s website](#). Download the executable and run it. Follow the onscreen instructions.

Additionally, the REV Hardware Client may be available offline on a flash drive. Ask a veteran member for help.

---

**Note:** The REV Hardware Client is currently only available for Windows.

---

## 1.1.4 NI FRC Game Tools

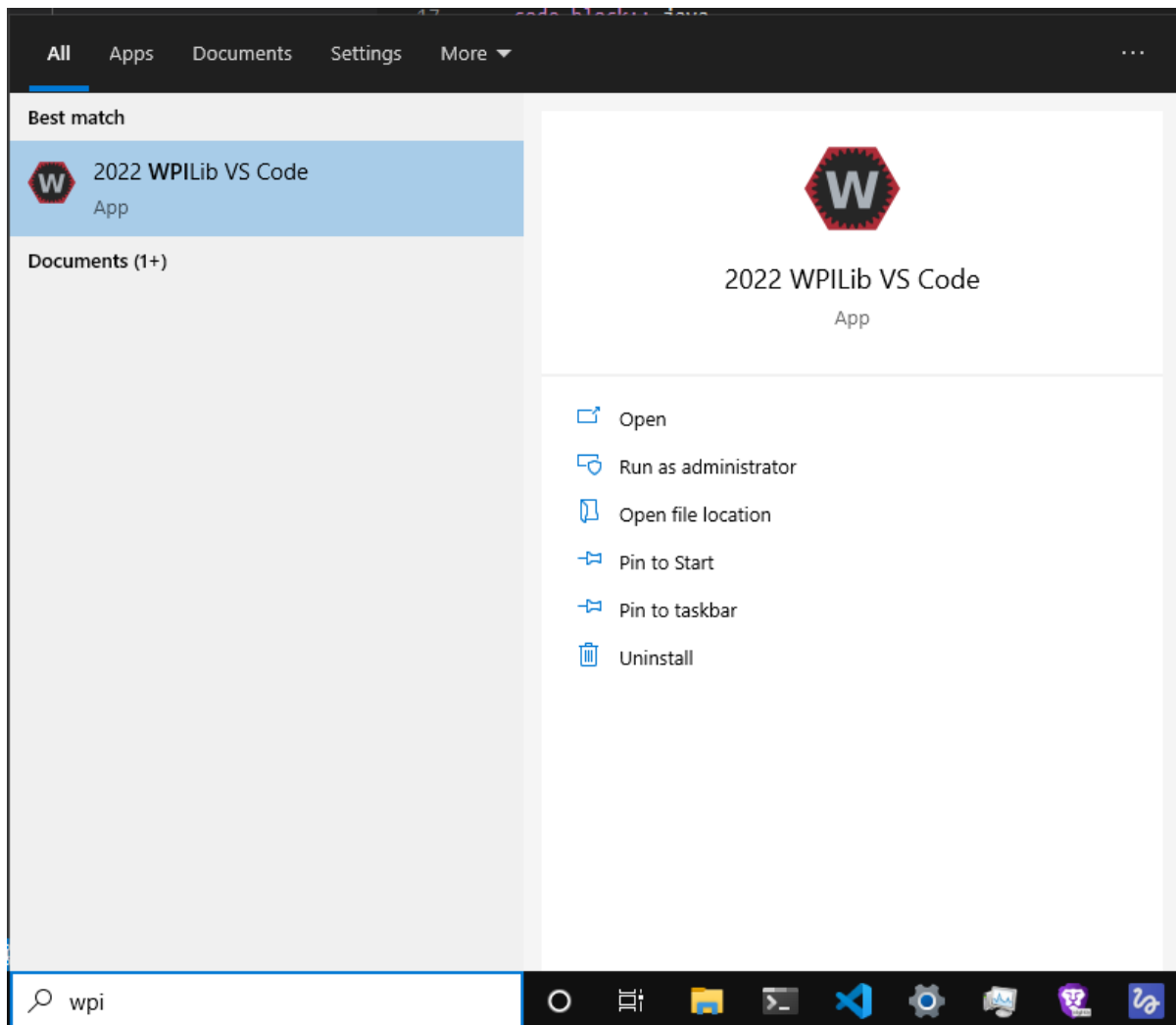
The NI FRC Game Tools contain tools to control, bring up, and run robots from your computer.

Instructions for installing the NI FRC Game Tools can be found on the [WPILib Docs](#).

## 1.2 Section 1: Creating a Project

### 1.2.1 Creating a Project

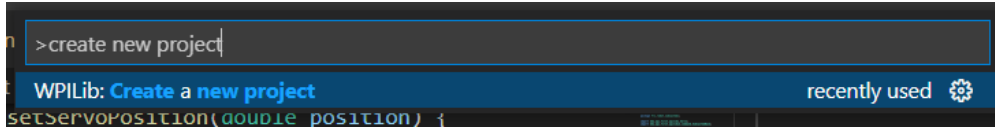
Begin by opening the WPILib VSCode installed in the previous section. This can be done through the start menu (see below) on Windows, or through Spotlight Search on MacOS.



VSCode may guide you through some initial setup. You can ignore most of this. When you're done with the setup, begin by hitting *Ctrl+Shift+P* (Note: most of the time, on MacOS, replace *Ctrl* with *Command*. If this is not the case, it will be noted.)

This should open up the “Command Palette”. From here, type “create new project”. You should see something akin to this:





Select this option, and you will be greeted with a “New Project Creator” menu. Begin by selecting a project type. Select “template”, then “java”, and finally, “Command Robot”. From here, select your base folder; it’s recommended to create a separate “code” folder somewhere—i.e. in your Documents folder.

The project name can be anything—for now, we can call it “First Project”. Make sure “Create a new folder?” is checked, and input your team number (i.e. 4028). In the end, your screen should look something like this:

**Welcome to WPILib New Project Creator**

template java Command Robot

**Base Folder.** Select a base folder to place the new project into.

c:\Users\botto\Documents\Code

Select a new project folder

**Project Name**

First Project

**Create a new folder?** ☒

This creates a new folder at Base Folder\Project Name. Highly recommended to be checked. Otherwise the folder will be placed at Base Folder and not utilize the Project Name

**Team Number**

4028

**Enable Desktop Support** ☐

This is needed for simulation and unit testing support, however there are some cases where this will do some unexpected things during build. In addition, not all vendor libraries support desktop. This option can be set with the command "WPILib: Set Desktop Support" at any time.

Generate Project

Finally, select “Yes (Current Window)”.

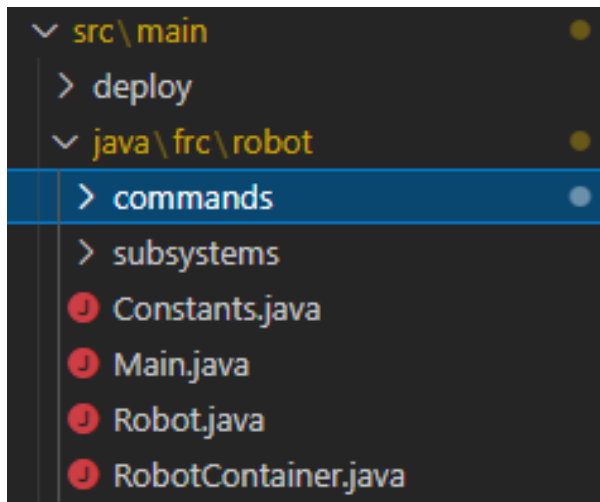
Congratulations! You’ve successfully created your first robot project!

## 1.3 Section 2: Your First Motor

### 1.3.1 Running Your First Motor

**Warning:** Make ABSOLUTELY SURE that you are using a Talon SRX motor controller for this! If you try to run a different motor controller, it will fail. Ask a mentor/teacher for help determining whether or not a motor controller is a TalonSRX.

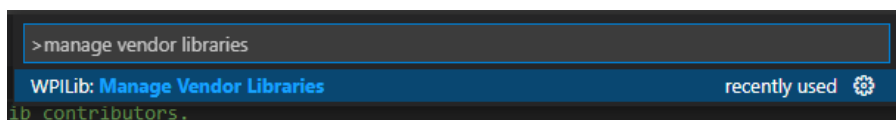
Firstly, we want to get into the main robot code. In your file tree on the left, first hit “src”, then “java”. You will see 4 files there, and 2 folders.



Open “Robot.java” by clicking on it. This file contains the main logic for the robot to run all of its code!

Before we create and run our motor, we need to install a “vendor dependency” for CTRE. Vendor dependencies are libraries developed not by WPILib, but by external “vendors”, who manufacture motor controllers, gyroscopes, etc. In this case, we are running a CTRE motor controller, so we need the CTRE vendor dependency.

Begin by opening the Command Palette again with *Ctrl+Shift+P*. Now type in “manage vendor libraries”. You should see the following:



Select it by pressing enter. Another menu should pop up. Hit “Install new libraries (offline)”. Select “CTRE Phoenix” and press enter. Select “Yes” to build.

Now, we can begin creating our project! Still in Robot.java, begin by going to the line that says:

```
private RobotContainer m_robotContainer;
```

Put your cursor on the end of the line, and press Enter/Return twice. Now, we need to create our motor controller object.

In this case, we are creating a *TalonSRX* (our motor controller). We can call it anything we want; for the time being, let’s call it *turretMotor*. Thus, on our new line, we want to type:

```
TalonSRX turretMotor;
```

---

**Note:** When you type *TalonSRX*, a menu will pop up, with the variable name showing up first. Hit “enter” when you see this, and VSCode will automatically import the needed files to use the TalonSRX class.

---

---

**Note:** In Java and most other programming languages, at the end of each line (or “statement”), we use semicolons (;) to determine that this is the end of the line. Semicolons are NOT OPTIONAL! When you have an error in your code, ALWAYS check your semicolons first and foremost!

---

---

**Note:** In Java, almost all variables are named according to “camel case” notation. This specifies that the first “word” of the name is lowercase, and any subsequent “words” within the variable name have their first letters capitalized. There are exceptions, such as constants; these will be discussed later.

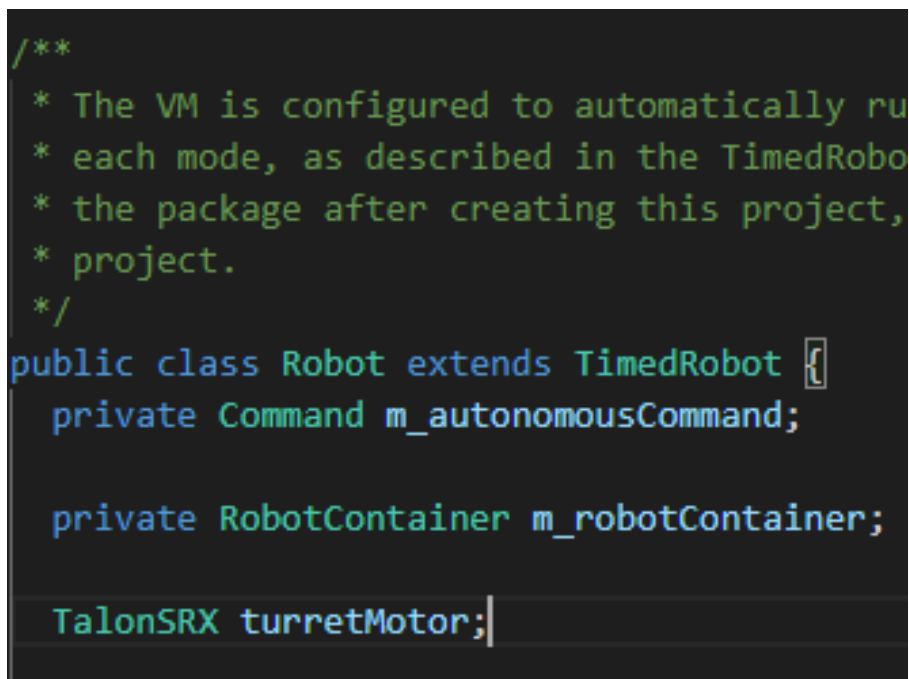
---

Your code should look like this:

---

**Note:** TODO: Should we use codeblocks or images? Codeblocks are a bit easier to maintain...

---



```
/**
 * The VM is configured to automatically run
 * each mode, as described in the TimedRobot
 * the package after creating this project,
 * project.
 */
public class Robot extends TimedRobot {
    private Command m_autonomousCommand;

    private RobotContainer m_robotContainer;

    TalonSRX turretMotor;
```

If your code looks like this, you’re good to move on. Scroll down until you see a “robotInit” function.

---

**Note:** *teleopInit* is the code that runs whenever you start the robot in “tele-operated” mode, shortened to “teleop”, where in a real robot, the drivers have control over it.

---

Set your cursor at the end of the line that says *m\_robotContainer = new RobotContainer();*, and hit Enter twice. We now need to “assign” a value to our motor variable. Assignment is done through the = operator, i.e. *variableName = someValue;*

With this in mind, to assign to our *turretMotor* object, we must create a new *TalonSRX* object. The *TalonSRX* constructor takes a single argument; the CAN ID of the motor controller. To determine the CAN ID... etc. Phoenix Tuner

stuff

In this case, our CAN ID should be 4. Thus, to “instantiate” (i.e. create) the *turretMotor*, we use:

```
turretMotor = new TalonSRX(4);
```

Don’t forget the semicolon!

Your code should now look like this:

```
public class Robot extends TimedRobot {
    private Command m_autonomousCommand;

    private RobotContainer m_robotContainer;

    TalonSRX turretMotor;

    /**
     * This function is run when the robot is first
     * initialized.
     */
    @Override
    public void robotInit() {
        // Instantiate our RobotContainer. This
        // will allow the autonomous chooser on the dashboard.
        m_robotContainer = new RobotContainer();

        turretMotor = new TalonSRX(4);
    }
}
```

Now that we’ve created our motor, it’s time to run it! Scroll down until you find the *teleopInit* function. Inside of this function (below the autonomous command stuff), we need to “call” a method within the TalonSRX object.

In this case, the method is `<TalonSRX>.set()`. The *set()* method takes two parameters: the control mode (you’ll learn more about this later), and the “percent output”, a value from -1 to 1, determining how fast the motor should run (0 = stop, -1/1 = full speed), and which direction (>0 = forward, <0 = reverse).

Begin by making two blank lines after the autonomous command stuff. Then, we need to call the *set()* function, with our desired parameters. Begin by typing *turretMotor.set*. A menu should pop up, with the *set* method showing up, with all of its parameters. Press enter to input this in. Now, replace *Mode* with *ControlMode.PercentOutput* (ensuring to import *ControlMode*), and *demand* with our target speed. For safety and tutorial reasons, run it low, i.e. 0.2. In the end, your code should be:

```
turretMotor.set(ControlMode.PercentOutput, 0.2);
```

And will look like:

```

@Override
public void teleopInit() {
    // This makes sure that the autonomous stops running when
    // teleop starts running. If you want the autonomous to
    // continue until interrupted by another command, remove
    // this line or comment it out.
    if (m_autonomousCommand != null) {
        m_autonomousCommand.cancel();
    }

    turretMotor.set(ControlMode.PercentOutput, 0.2);
}

```

Now, you’ve created your code! It’s time to deploy and run it. First of all, we need to connect to the robot’s radio. Ensure the robot is turned on (you will see the orange light) and go to your Wi-Fi settings in the bottom right, selecting the radio (i.e. 4028\_SNEED).

picture of radio in wifi tab

Now that we’re connected to the robot, it’s time to deploy the code. Go back to your code, and press *Shift+F5*. You may also need to press *Fn*.

You might get a message saying “Starting a Gradle Daemon”. After some time, you should see something like the following:

```

BUILD SUCCESSFUL in 25s
4 actionable tasks: 3 executed, 1 up-to-date

Terminal will be reused by tasks, press any key to close it.

```

If you see any errors, ensure you’re connected to the robot and that your code doesn’t contain any errors (underlined in red in your code).

We’ve now successfully deployed our code! Now, it’s time to run it! Open up the FRC Driver Station installed in section 0. You should see something like this:

picture of driver station with comms

Ensure “TeleOperated” is selected, and press “enable”. The motor should run. If not, ask a veteran member for help.

Congratulations! You’ve written and deployed your first code!

## 1.4 Section 3: Servos & Solenoids

### 1.4.1 What are Servos & Solenoids?

Servos are a type of very basic motor. They can have a linear or rotational movement pattern, and are designed to go to a position (to “servo” to a position) and stay there with high amounts of force. These are typically used for shooter hoods, or for small-scale, low-power rotation of arms or other limbs.

Solenoids control pneumatics systems. Pneumatics are used to control “cylinders”, which are linear actuators with two states: fully extended, or fully retracted. Because they use compressed air to transfer force, they can generally provide

enough force to move an object that a motor can't. They are solely used for two-state systems, i.e. an infeed or a hook.

### 1.4.2 Servos in Code

Servos are extremely simple to use—using them is identical to running a basic motor. First, create your servo variable at the top of `Robot.java`:

```
Servo servoMotor;
```

Now we instantiate our servo. Much like a motor, the servo constructor takes a single argument: the PWM ID. Check the RIO's PWM ports, and trace the servo you want to control to the PWM port it uses, and plug this in (in `robotInit`):

```
servoMotor = new Servo(0);
```

Controlling servos is just like a motor as well, although the “percent output” is really the position to set the servo to. Servos generally have a hard minimum or maximum position. On 4028, the servos we use generally have a range of 0.2-1.0. Because of this, if you want to go to the “zero” position, you must set the servo to 0.2. In `teleopInit`:

```
servoMotor.set(0.2);
```

Deploy and enable as described in the last section. The servo should be fully retracted after a while.

Now let's try to fully lengthen it. Replace the 0.2 with 1.0, deploy, and enable. It should go to the fully lengthened position.

### 1.4.3 Solenoids in Code

Solenoids are even simpler than servos. They are simple binary actuators: they only have a true and false state. Double solenoids are slightly different, with forward, reverse, and off states. However, they both accomplish the same thing: to either retract or extend a pneumatic cylinder (see the Pneumatics link in the sidebar).

As always, create your solenoid variable and instantiate it. At the top of the `Robot` class:

```
Solenoid solenoid;
```

The Solenoid constructor takes two arguments: the pneumatic hub type (this is either *PneumaticsModuleType.CTREPCM* or *PneumaticsModuleType.REVPH*. Ask a veteran member for help determining which to use), and the channel. For now, we are only using single solenoids. Trace the solenoid you want to control back to the PCM or PH, and find its port. Use this port in your constructor (`robotInit`):

```
solenoid = new Solenoid(PneumaticsModuleType.CTREPCM, 0);
```

Replace 0 with the port you determined.

To set a solenoid, we simply call `solenoid.set()`. This call takes one argument: true or false. The default state of a solenoid is false; so, to see a difference, you will want to set this to *true* (`teleopInit`):

```
solenoid.set(true);
```

Now, deploy and enable. You should hear a small click and the solenoid will light up. If pneumatics are hooked up on your test board, a cylinder should fire as well.

### 1.4.4 Double Solenoids in Code

Bruh

## 1.5 Section 4: Controllers

### 1.5.1 Basic Setup

Controllers do as their namesake: they control things. Ever played video games? You’ve used a controller. Ever used a keyboard and mouse? You’ve used a controller. In FRC, the controllers we use are similar if not identical to those found on consoles like Xbox.



But how do we use them?

For simplicity’s sake, we will be using the `BeakXboxController` class as our wrapper around controllers. The `BeakXboxController` class itself wraps around the built-in WPILib class, `XboxController`, which provides functionality to use Xbox and Xbox-like controllers for robots. Buttons are mapped properly for an Xbox controller, and generally, this is how we give commands to the robot.

To start, get the `BeakXboxController` class. Go to [here](#). In the file that opens, select all and copy. Now, in your *robot* folder, create a new folder and call it *utilities*. From here, right-click on the utilities folder, and click “New File”. Name the file *BeakXBoxController.java* (watch your capitalization!), and press enter. Now, delete whatever may be in the new file, and then press `Ctrl+V` to paste the `BeakXBoxController` class. You have successfully created your controller!

### 1.5.2 Setup in Code

To use the `BeakXboxController`, we first need to create our controller instance. Go to your *RobotContainer.java* file, and in the *RobotContainer*’s member variable definitions, create a *private* `BeakXboxController`, and name it *driverController*.

Now, in your *RobotContainer* constructor, BEFORE `configureButtonBindings` is called, initialize your driver controller. The `BeakXBoxController` constructor takes one argument: the port. Controller ports start at 0, meaning that the “first” controller is actually port 0. We want to use the first controller, so plug in 0 for your port. Your code should now look something like this:

```

23 public class RobotContainer {
24     // The robot's subsystems and commands are defined
25     private final ExampleSubsystem m_exampleSubsystem =
26
27     private final ExampleCommand m_autoCommand = new Ex
28
29     // Controller Setup
30     private BeakXBoxController driverController;
31
32     /**
33      * The container for the robot. Contains subsystems
34      */
35     public RobotContainer() {
36         // Like arrays, port indices start at 0.
37         driverController = new BeakXBoxController(0);
38         // Configure the button bindings
39         configureButtonBindings();
40     }
41
42     /**

```

You've now set up a controller in code! Let's use it now.

### 1.5.3 Usage

You may have learned about inline functions in your Java training. If not, what you need to know is that the basic form is `() -> functionToRun()`. Feel free to look up what inline functions are if you need more info. For now, all you need to know is that we plug this into the controller to run something. So, let's get started!

First of all, remember back to our previous lessons on servos, solenoids, and motors? Remember how you had to redeploy code every time you changed a value? Well, in competition, you *can't* redeploy code to change values! Thus, one of the many ways we change values "on-the-fly" is through controllers. For example, you can press one button to run a servo to a shortened position, and another to run it to an elongated position. That's exactly what we'll be doing here.

To start in code, we first need to stop anything from being done automatically in teleop. Go to *Robot.java* and remove all 3 *set()* calls in *teleopInit*. We won't be needing these anymore. Now, remove your definition and initialization of your *servoMotor*, and place it into *RobotContainer.java*. Your *RobotContainer* should look like this:



```

public class RobotContainer {
    // The robot's subsystems and commands are defined here
    private final ExampleSubsystem m_exampleSubsystem = new ExampleSubsystem();

    private final ExampleCommand m_autoCommand = new ExampleCommand(m_exampleSubsystem);

    // Controller Setup
    private BeakXBoxController driverController;

    // Controlled Components
    private Servo servoMotor;

    /**
     * The container for the robot. Contains subsystems,
     */
    public RobotContainer() {
        // Like arrays, port indices start at 0.
        driverController = new BeakXBoxController(0);

        // Setup controlled components
        servoMotor = new Servo(0);

        // Configure the button bindings
        configureButtonBindings();
    }
}

```

Now, how do we control it? We bind it! Binding means effectively mapping a button press to an action to be performed on the robot. With the *BeakXBoxController* class, this is easy! Scroll down to the *configureButtonBindings* method. This function is where we bind all our buttons.

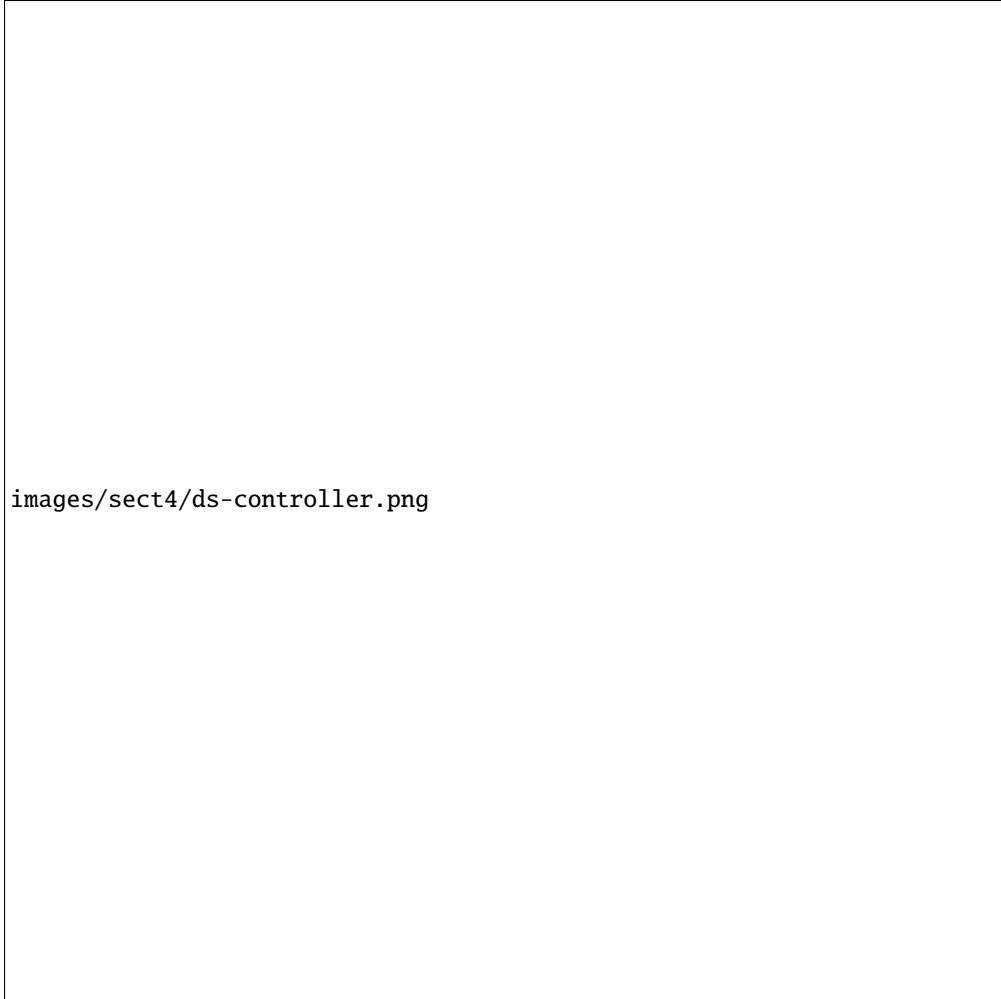
To bind to a specific button with *BeakXBoxController*, you can access the buttons themselves, for example, *driverController.a* accesses the A button. To bind a command to that button, call *.whenPressed* of the button. Now to input your command, use the inline function notation you just learned about. We're going to bind the A button to the shortened position of the servo; thus, we put in *() -> servoMotor.set(0.2)*. Don't forget your semicolon at the very end of the line. Your code should look like this:

```

private void configureButtonBindings() {
    driverController.a.whenPressed(() -> servoMotor.set(0.2));
}

```

If everything looks good, deploy your code. Now, you need to plug in a controller to the driver station, and verify that the controller is in the correct place. Plug in a controller to your driver station computer (via USB), and open the driver station. On the left, you will see a USB icon. Press this icon, and you should now see a list, and the first item should be "0 Controller (Gamepad F310)". This means it's plugged in correctly. If the first number is different, click and drag the controller to the first slot until it says 0. Now, to verify that it works and is in the correct port, press any button on the controller. In the driver station, the controller should now light up green, like this:



images/sect4/ds-controller.png

Your controller is now good to go. Enable, and you will see that nothing happens. This is normal—there’s nothing scheduled to happen! To see something happen, press A on your controller, and you will see the servo move to the retracted position!

But if you press it again, nothing happens. If you want to move it between positions, we need another position to be bound. Bind B to the fully lengthened position (1.0). Your code should now look like this:

```
private void configureButtonBindings() {  
    driverController.a.whenPressed(() -> servoMotor.set(0.2));  
    driverController.b.whenPressed(() -> servoMotor.set(1.0));  
}
```

Deploy and enable. Press B and it should go to the fully lengthened position. Then, press A, and it’ll go back! This, fundamentally, is how we manage the state of the robot. We press different buttons, and they do different things. Now, you have control over whatever you want, without having to redeploy!

Congratulations on your work! Controllers will be used exclusively in the next few modules, so make sure you understand everything. Try out some things on your own. Bind X to some other position. Bind Y to running a motor. Bind the right bumper (known in code as *rb*) to toggle the solenoid. The world is your oyster when it comes to controllers!

## **1.6 Section 5: Encoders**

they encode stuff

## **1.7 Section 6: Control - Limits & Sensing**

they sense and limit stuff

## **1.8 Section 7: Subsystems**

they are below the system

## **1.9 Section 8: Commands**

they command stuff to happen

## **1.10 Section 9: Dashboards & Debugging**

they dash stuff to a board

## **1.11 Section 10: Cameras**

they see stuff

## **1.12 Section 11: PID Control**

they control stuff using PID

## **1.13 The Basics of Wiring & Electronics**

### **1.13.1 Electrical Components & How to Wire Them**

PDP, breaker, etc

### **1.13.2 Types, Gauges, & Uses of Wires**

Hi

### **1.13.3 Safety (and what not to do)**

r

### **1.13.4 Powerpole**

crimp be like

### **1.13.5 Ferrules**

crimp be like 2

### **1.13.6 Fuses**

Might not even need this section.

## **1.14 Batteries**

bruh

### **1.14.1 Usage & Placement**

### **1.14.2 Setting Up & Crimping Batteries**

### **1.14.3 Beaking & Testing Batteries**

### **1.14.4 What's a "Good" Battery?**

MK

## **1.15 BeakLib**