DUO Control System

Introduction

The REV Control Hub is an affordable robotics controller providing a platform for the interfaces required for building robots. The Control Hub works with the Expansion Hub and Driver Hub to create a complete robotics control system for both the classroom and the competition. These devices are most commonly used within the FIRST Tech Challenge (FTC), FIRST Global Challenge (FGC), and in the classroom for educational purposes.

How to use this documentation?

This documentation is intended as the place to answer any questions related to the REV Robotics Control Hub, Driver Hub, and Expansion Hub used in the FIRST Tech Challenge and FIRST Global Challenge.

- **Looking to get an idea of how to use the system before your Control Hub arrives?** Reading through each section will help, but we specifically recommend the guides on getting started with the Control Hub and the programming language options section.
- **Have a specific question?** Feel free to head straight to it using the navigation bar to the left. Each section is grouped with other topics that are similar.
- **Having trouble finding what you are looking for?** Try the search bar in the upper right or read the section descriptions below to find the best fit.

Getting started building robots can be an intimidating process. The following documentation is here to make getting started a bit easier. There are a number of examples to get started with the Control System and we are committed to adding content to make it more accessible for people to use REV.

If there is a question that is not answered by this space, send our support team an email; **support@revrobotics.com**. We are happy to help point you in the right direction.

What is in each section?

System Overview

This section contains information regarding all of the major mechanical specifications of the REV Control Hub and Expansion Hub. These sections include port pinout information, protection features, and the types of cables used with the devices.

Getting Started

Take the Control Hub or Expansion Hub from out of the box through generating the first configuration file. This includes the process for changing your Control Hub's Name and Password as well as connecting to your Driver Hub. Also includes information on ways to add additional motors to the control system through adding a SPARKmini Motor Controller or an Expansion Hub.

Updating and Managing

This section covers how the information needed to keep your Control Hub, Expansion Hub, and Driver Hub up to date with the latest software. This section also includes information on using the REV Hardware Client to update, program, and manage these devices as well.

Programming

From just getting started by writing your first Op Mode to working with closed loop control, this section covers the information needed to start programming.

Sensors

Sensors are often vital for robots to gather information about the world around them. Use this section to find how to use REV sensors and information on the different sensor types.

Getting Started with Control Hub

After receiving the Control Hub it is advised to unbox the device, power the Control Hub on, and start the configuration process. Below are the required materials to run through the initial bring up of the Control Hub and links to the different steps of the process.

Section	Summary
Connect to the Robot Controller Console	In order to manage the Control Hub (REV-31-159 or programming using the onboard programming languages you must have access to the Robot Controller Console. Follow through the steps in th section to ensure your Control Hub is connecting properly
Updating Wi-Fi Settings	Once in the Robot Controller Console, update you Control Hub's Wi-Fi settings for better performanc and network security.
Connecting Driver Station to Control Hub	A Driver Station is required to in the REV Control System, to run code remotely. This section walks through the steps of connecting a Driver Station device to a Control Hub.
	Showcases what hardware components plug into

Wiring Diagram	which ports on the Control Hub.
Next Steps	Once the hardware components are connected to the Control Hub, the basic steps for getting started have been covered. This section covers the important next steps you should take for working with and maintaining your Control System.

Required Materials

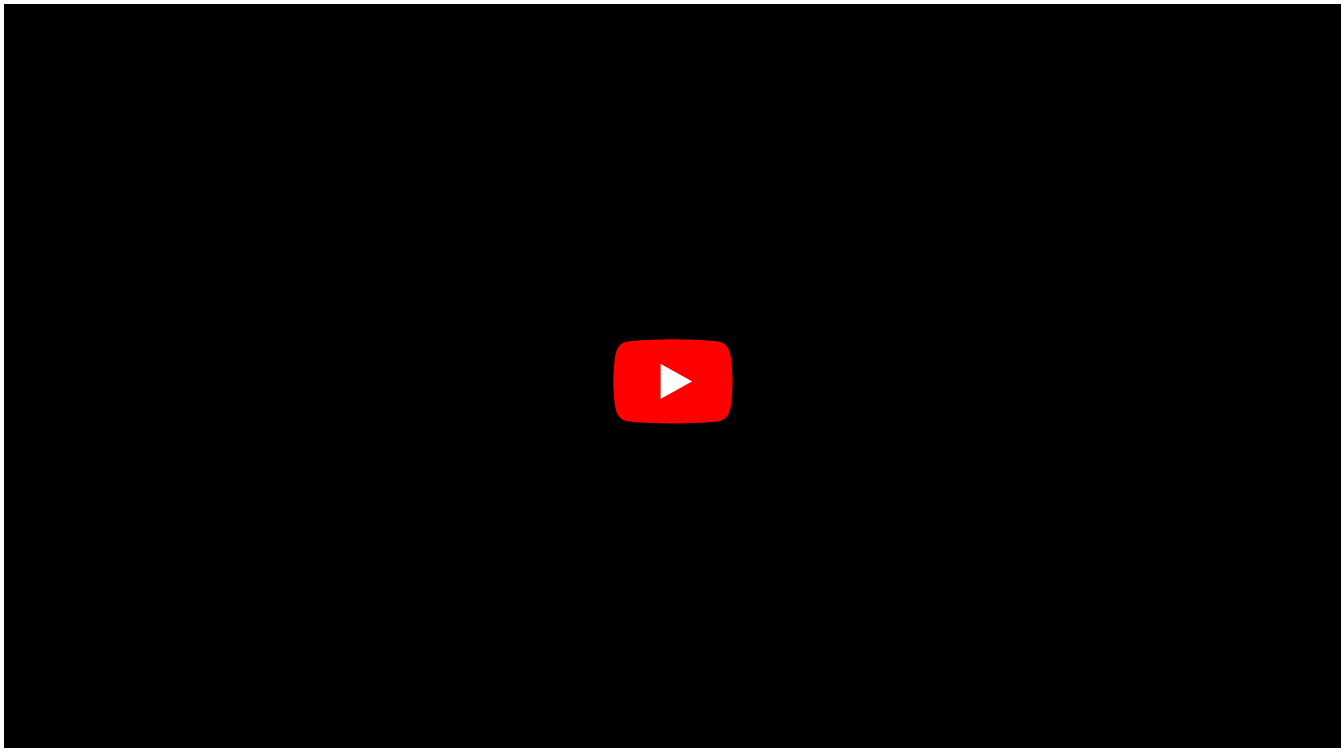
- Control Hub (REV-31-1595)
- 12v Slim Battery (REV-31-1302)
- Driver Hub (REV-31-1596)
- Etpark Wired Controller for PS4 (REV-39-1865)
- USB A Female to Micro USB (REV-31-1807)
- Windows PC running the REV Hardware Client

Optional Additional Materials needed to Connect an Expansion Hub:

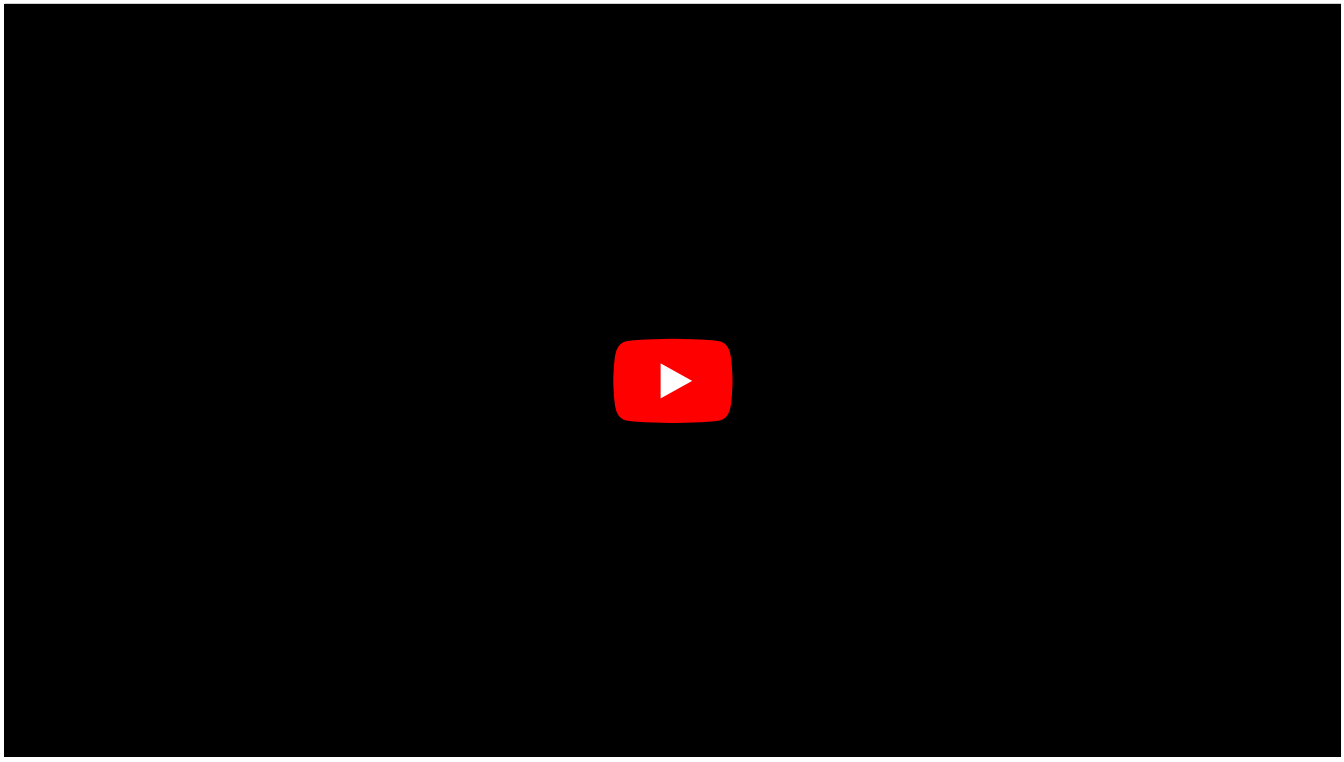
- Expansion Hub (REV-31-1153)
- XT30 Extension Cable (REV-31-1392, included with Expansion Hub)
- JST PH 3-pin Communication Cable (REV-31-1417, included with Expansion Hub)

Videos

Using a Web Browser



Using the REV Hardware Client



Connect to the Robot Controller Console

In order to manage the Control Hub (REV-31-1595) or programming using the onboard programming languages, a computer or other Wi-Fi enabled device will need to connect to the Control Hub's Robot Controller Console. The Robot Control Console is a local network created by the Control Hub to program and manage the device.

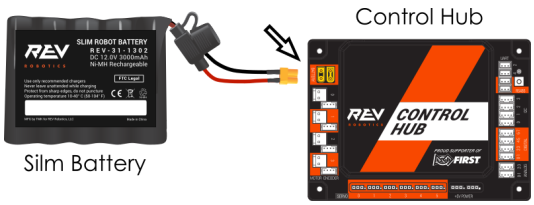
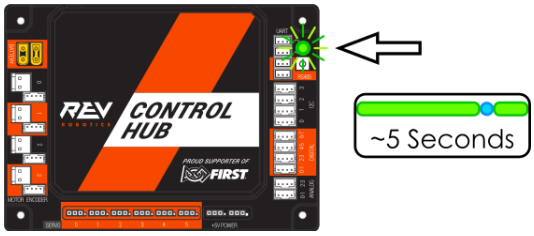
> ⓘ This example assumes the user uses Windows 10 as their operating system. If you are not using a Windows 10, the procedure to connect to the network will differ. Refer to your device's documentation for details on how to connect to a Wi-Fi network.

By default, the Control Hub has a name that begins with "FTC-" or "FIRST-" followed by four characters that are assigned randomly. The default password for the network is "password". If either of these is forgotten, there are a few ways to recovery or reset the password on the Control Hub.

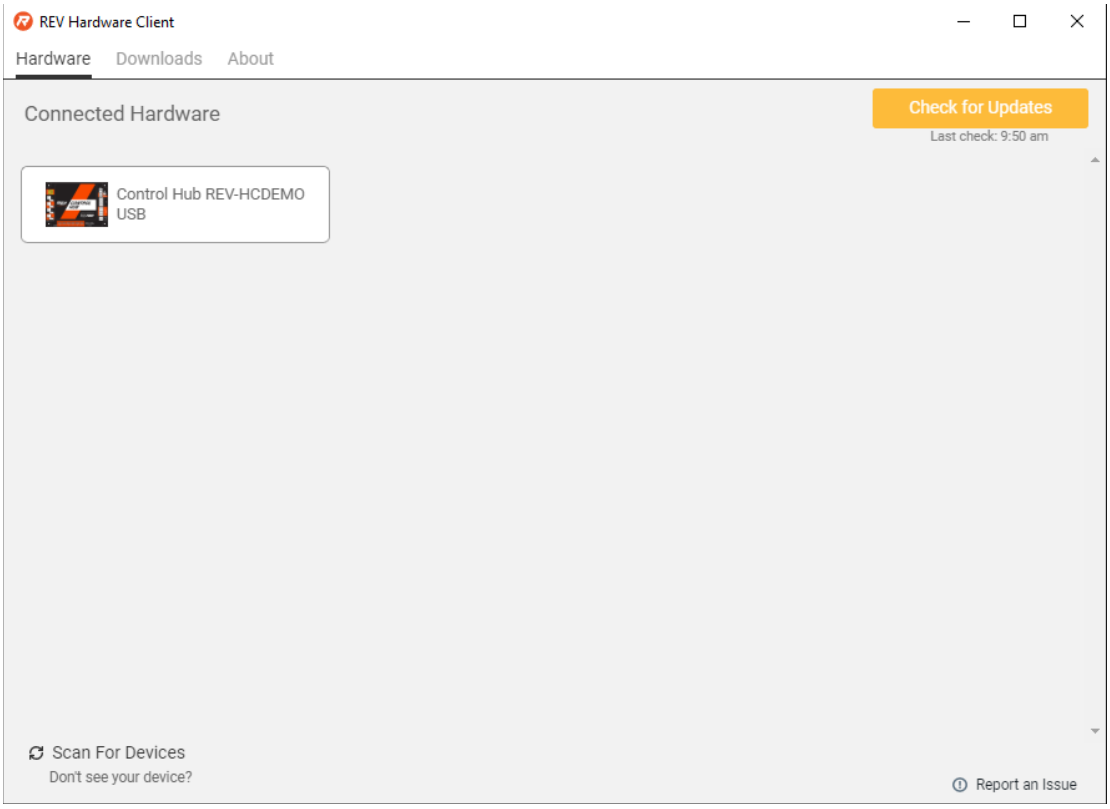
There are two ways to access the Robot Controller Console. The first will cover how to access the Robot Controller Console with the REV Hardware Client. It is recommended to use the REV Hardware Client as it will allow the user to access the Robot Controller Console over a wired connection. The second will run through accessing the Robot Controller Console via a web browser.

REV Hardware Client

Download the latest version of the REV Hardware Client and install on a Windows PC.

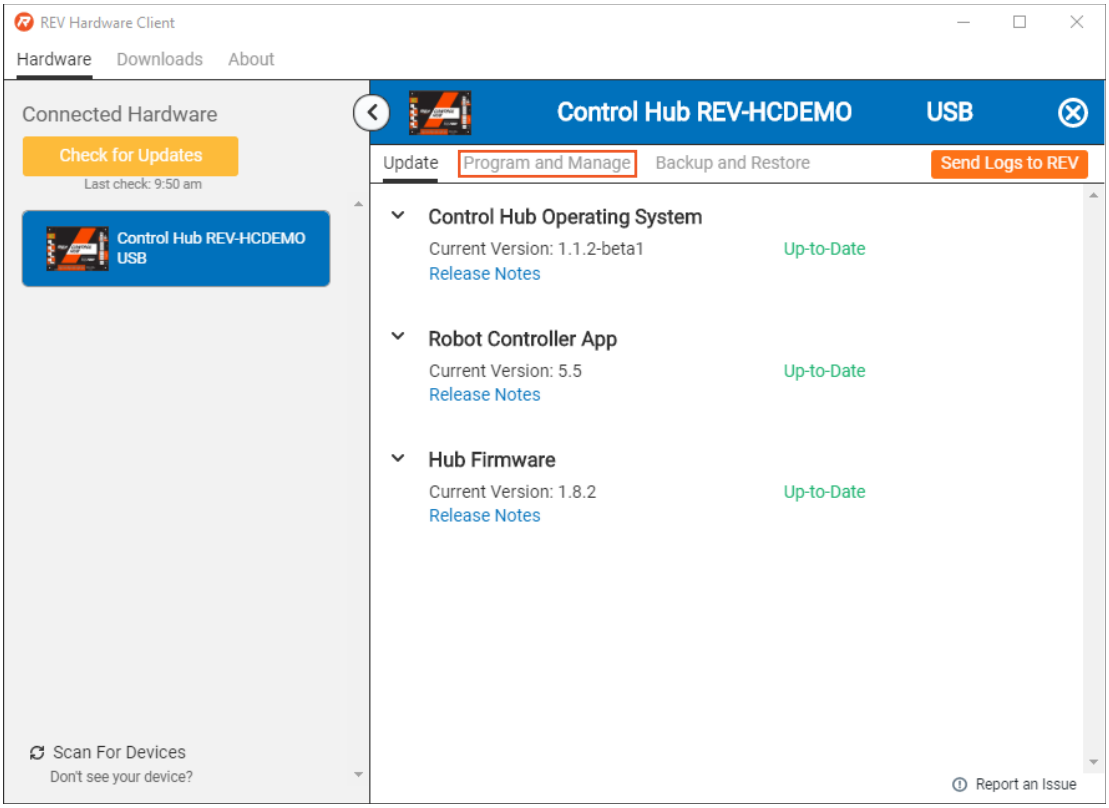
Steps	
Power on the Control Hub, by plugging the 12V Slim Battery (REV-31-1302) into the XT30 connector labeled “BATTERY” on the Control Hub.	
The Control Hub is ready to connect with a PC when the LED turns green. Note: the light blinks blue every ~5 seconds to indicate that the Control Hub is healthy.	
Plug the Control Hub into the PC using a USB-A to USB-C Cable (REV-11-1232)	

Startup the REV Hardware Client. Once the hub is fully connected it will show up on the front page of the UI under the **Hardware Tab**. Select the Control Hub.



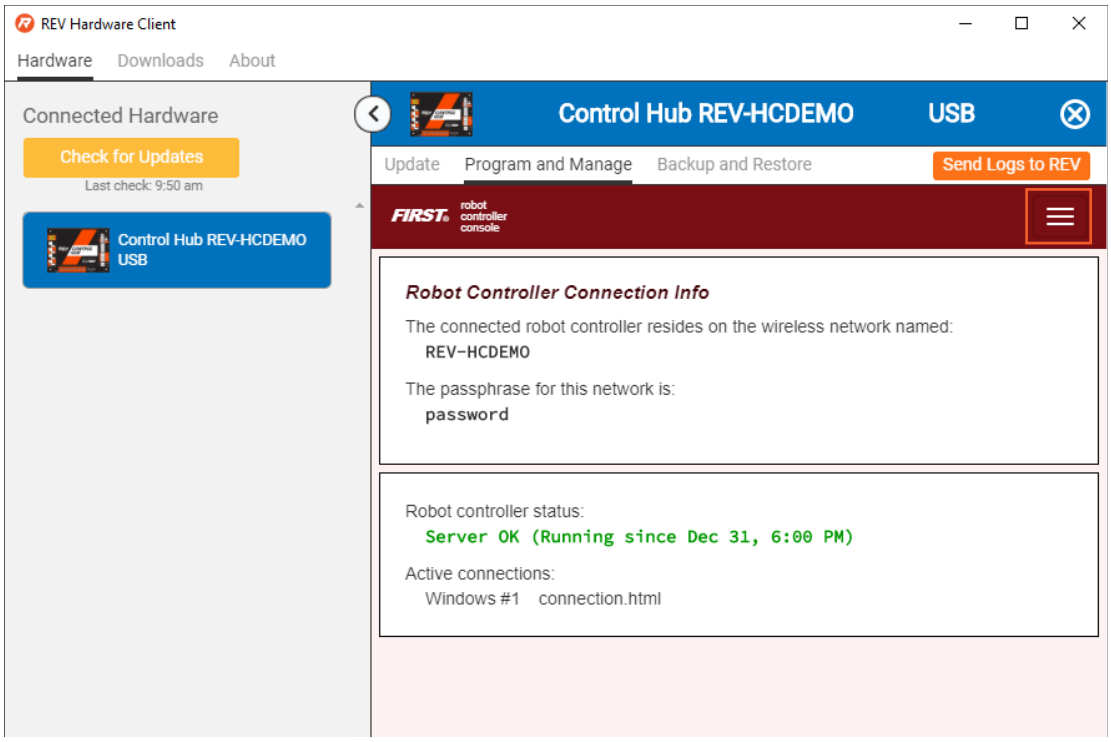
After selecting the Connected Hardware the Update tab will pop up. Select the Program and Manage tab.

This will take you to the Robot Controller Console build into the REV Hardware Client.



> ⓘ At this point it is useful to update the Control Hub Operating System, Robot Controller App, and the Hub Firmware.

Once in the Robot Controller Console, the homepage of the console will appear. In the upper right corner is the navigation menu which will allow users to access the Blocks, OnBot Java, and Manage pages within the console.

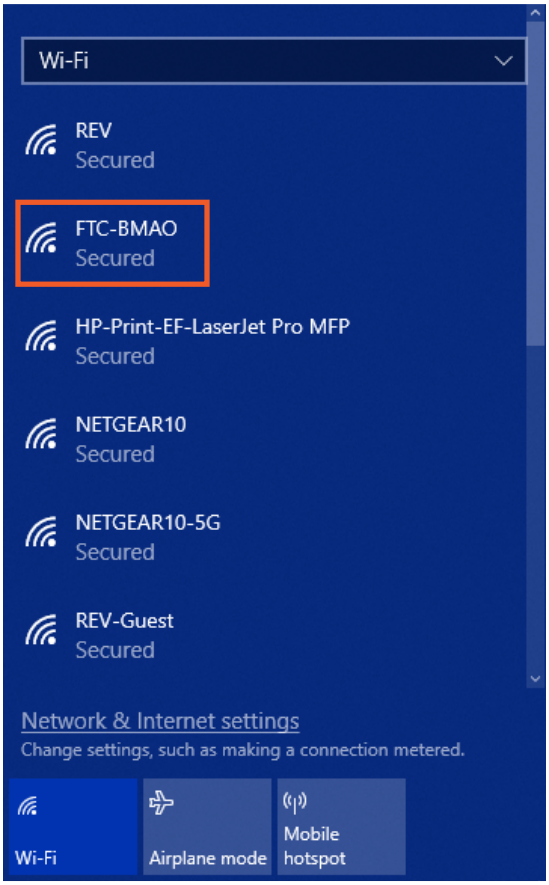


Web Browser

With the Control Hub powered, access the Wi-Fi network selector. For Windows 10 devices, click the Wi-Fi Network icon in the lower right corner of the desktop.

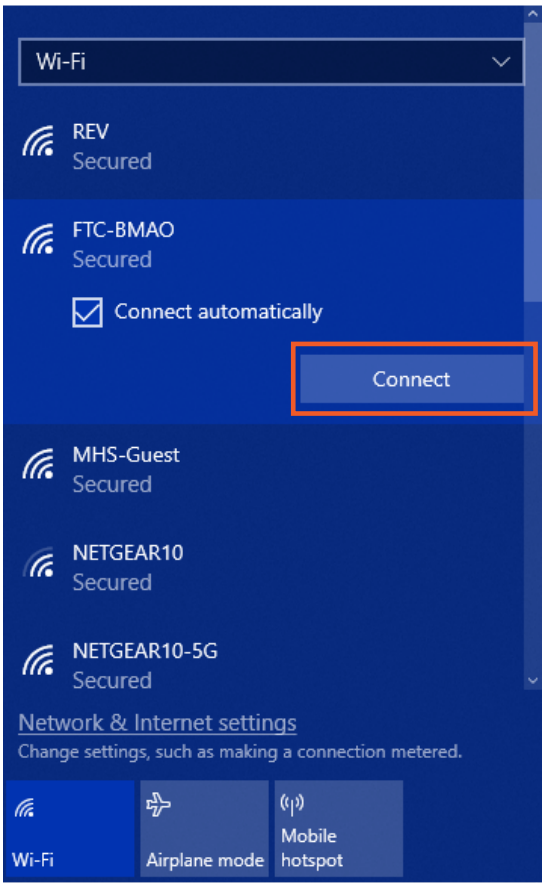
Look for the Wi-Fi that matches the naming protocol of the device.

> To ensure you are able to locate the correct device, it is recommended that you first connect in a location without other active Control Hubs or significant Wi-Fi connections.

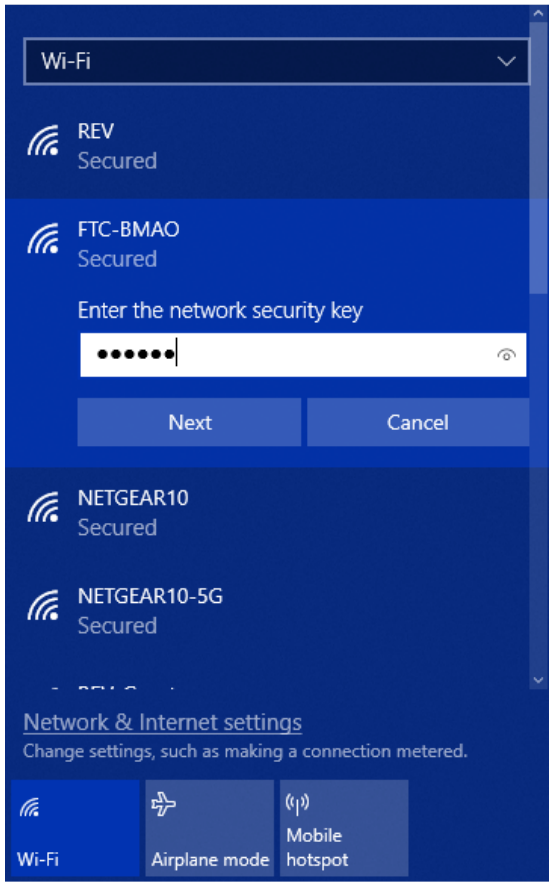


> Depending on your version of Windows or other theme settings your Wi-Fi Networks list may vary in appearance.

Once you have found the target network in the list, click on it to select it then press connect.



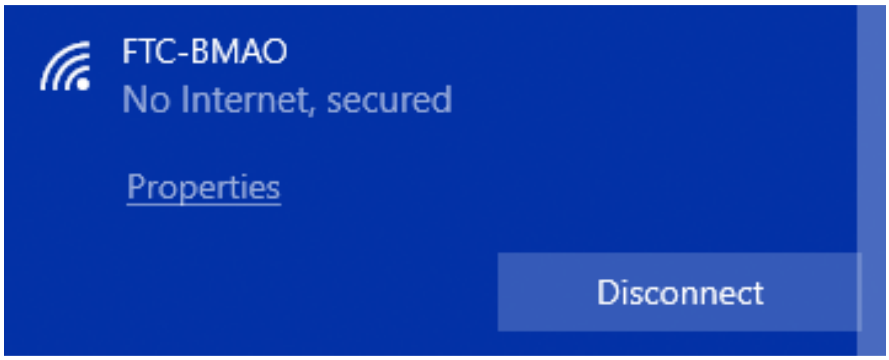
Provide the network password (in this example “password”) and press “Next” to continue.



Passwords are case sensitive. Make sure that your spelling and capitalization matches the

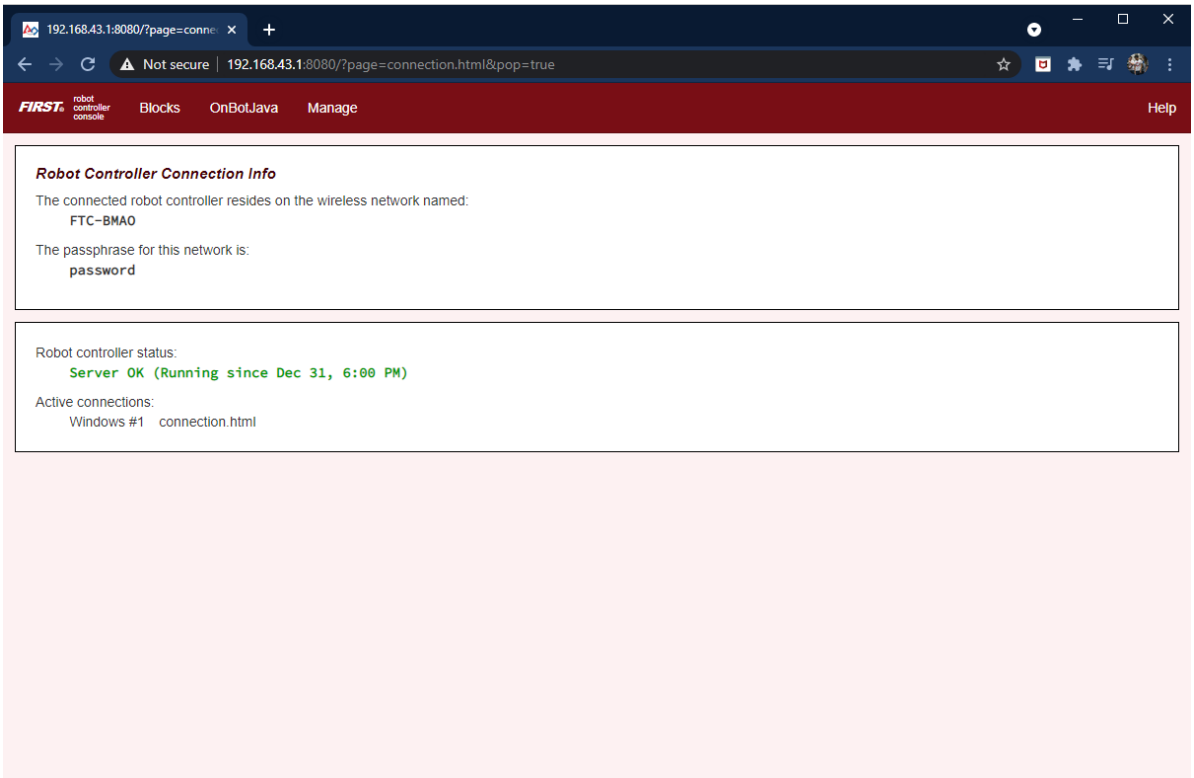
> original spelling and capitalization for the password.

Once a wireless connection is established, the status is displayed in the wireless settings for the device.



> ⚠ When connected to the Control Hub, the connected device will not have access to the Internet. It only has direct access to the Control Hub.

Open a web browser (Chrome, Firefox, Internet Explorer) and navigate to "192.168.43.1:8080" through the address bar.



From the Robot Controller Console users can update the Wi-Fi settings, upgrade the operating system and firmware, as well as program the device. It is strongly recommended that you go through all steps above before you begin programming.

Updating Wi-Fi Settings

One of the first recommendations made to users of the REV Control System is to update Wi-Fi settings, specifically the name and the password. All REV Control Hub's come with a default network name and password. It is useful to change the name and password especially in environments where there are multiple Control Hubs running like at an event or in a classroom. Changing from the default adds the element of network security back to the Hub by reducing the potential for access from outside sources.

With the release of Robot Controller Application 5.5 there have been some major changes to the process of changing Control Hub name, password, Wi-Fi Channel, and Wi-Fi band. Previously changes to the name and password had to be made separately. Each change would reset the network and require users to reconnect to the network in order to change anything else. With 5.5 all changes can be made at once.

The Control Hub (REV-31-1595) can utilize either the 2.4 GHz or 5 GHz Wi-Fi band. In OS versions 1.1.1 and older the Control hub defaults to a channel on the 2.4 GHz band. REV Robotics advises that during competition teams utilize a 5 GHz channel for robot communication. Consult the table below for Driver Station devices that can operate on the 5 GHz band.

> When using OS 1.1.2 the Control Hub operates by default on the 5Ghz band. To switch to the 2.4 Ghz band without the REV Hardware Client, see the Managing the Wi-Fi Network section.

Supported Android Devices and Wi-Fi Band Capabilities

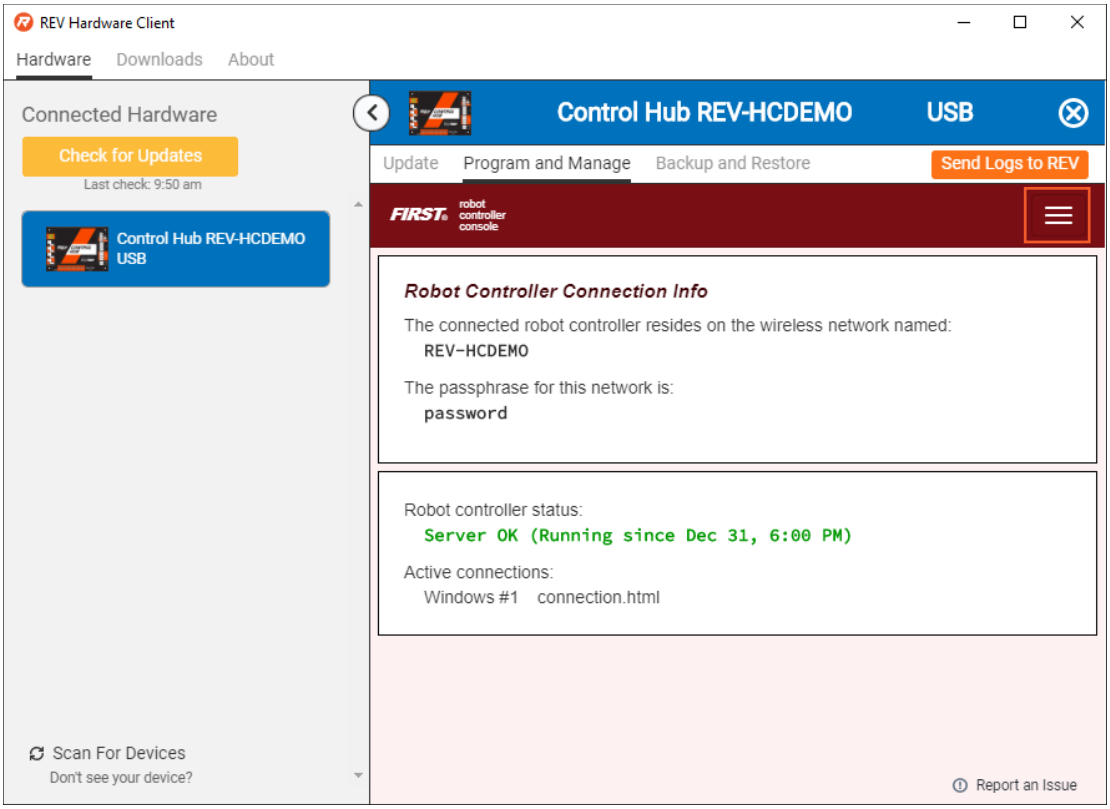
Phone	Wi-Fi Band
REV Driver Hub (REV-31-1596)	2.4 GHz & 5 GHz (Dual Band)
Moto G (2nd generation)	2.4 GHz (Single Band)
Moto G (3rd generation)	2.4 GHz (Single Band)
Moto G (4th generation)	2.4 GHz (Single Band)
Moto G5	2.4 GHz & 5 GHz (Dual Band)
Moto G5 Plus	2.4 GHz & 5 GHz (Dual Band)
Moto E4	2.4 GHz & 5 GHz (Dual Band)
Moto E5	2.4 GHz & 5 GHz (Dual Band)
Moto E5 Play	2.4 GHz & 5 GHz (Dual Band)

The following section will highlight how to access and make changes within the Wi-Fi settings. This section will use the REV Hardware Client to showcase how to make these changes. Once a user has connected to the Robot Controller Console, either via the Hardware Client or a web browser, the steps for accessing Wi-Fi settings are the same.

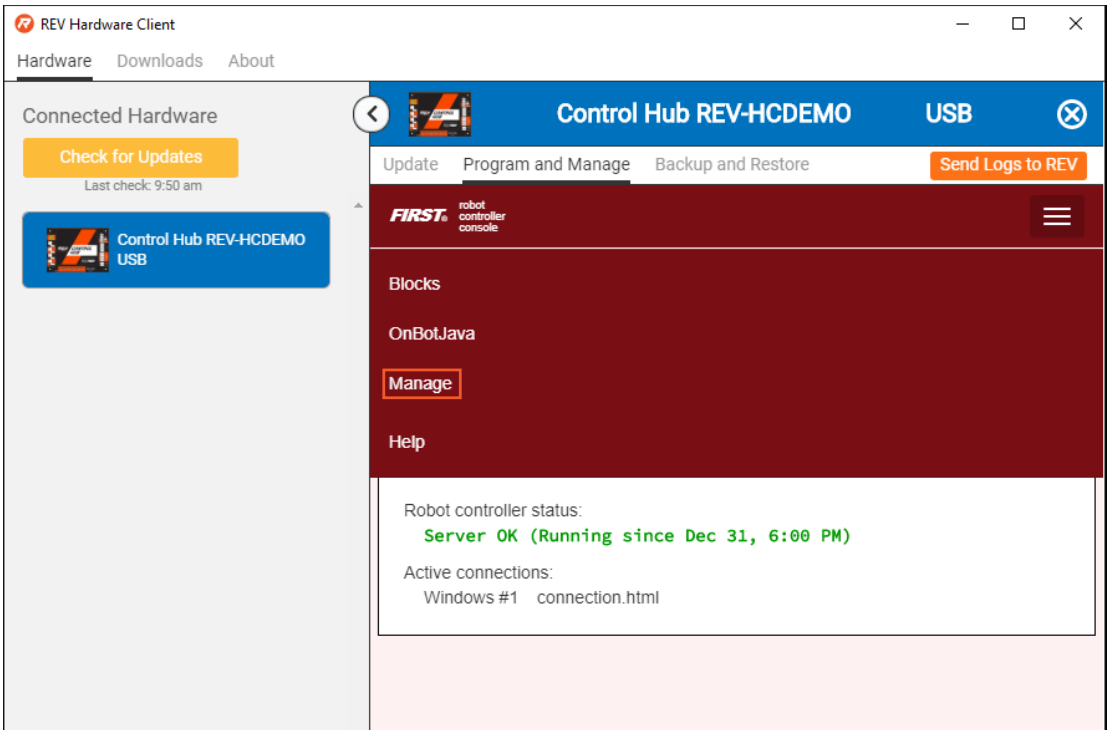
> The following steps assume that users have already connected to the Robot Controller Console. Please go to the Connect to the Robot Controller Console if this is not the case.

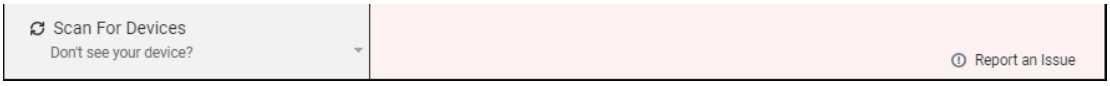
Steps to Updating Wi-Fi Settings

While in the Robot Controller Console select the menu button. In the image below the menu button is highlighted by an orange square in the upper right-hand corner.



When the menu opens, select Manage.

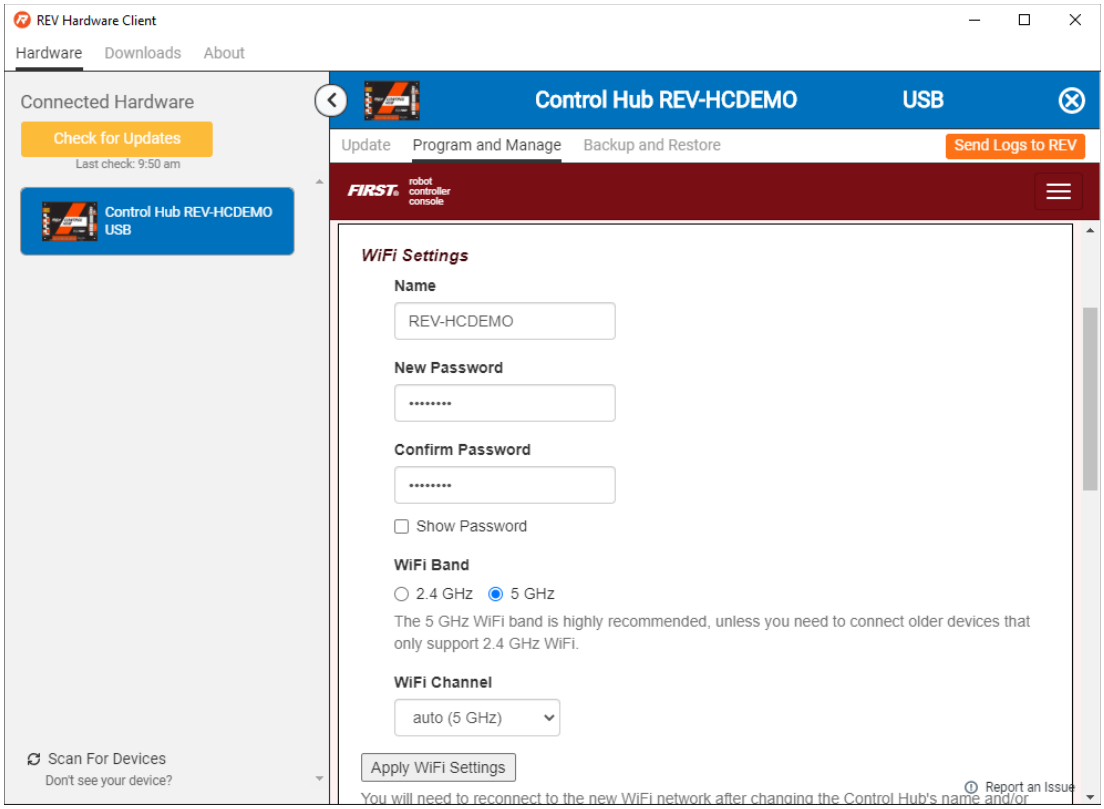




The Manage page is where the Wi-Fi Settings live. The following steps will show and discuss each change as it is made. Please keep in mind the following warning while moving through the steps:

You will need to reconnect to the new Wi-Fi network after changing the name and/or password. This is true for any Wi-Fi connection, but if you are accessing the REV Hardware Client via a USB connection the Hub will stay connected. Though, you may need to close and reopen the Hardware Client in order to see the changes.

> Not all aspects of the Wi-Fi settings need to be changed. If you need to change name and password and do not need to mess with the Wi-Fi band or channel, leave those settings at default, and click **Apply Wi-Fi Settings**.



Changing Control Hub Name

Under Wi-Fi Settings, there is an option to change the name of the Control Hub.

> It is useful to change the Control Hub name to something unique, especially in environments where there are multiple Control Hubs running like at an event or in a classroom. Changing from the default adds the element of network security back to the Hub by reducing the potential for access from outside sources.

For FTC teams you will want to change the name from the default to team number - RC. (i.e. 99999-RC)

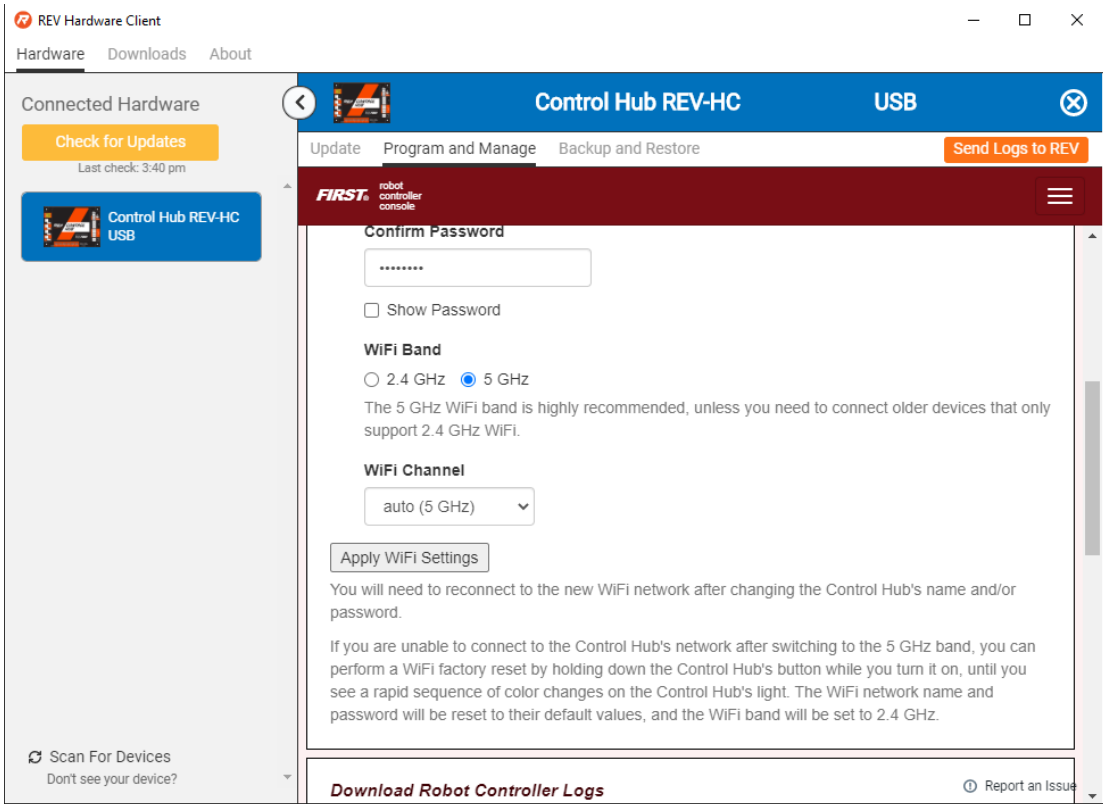
Changing the Control Hub Password

Under Wi-Fi Settings, there is an option to change the password of the Control Hub. There are not any restrictions on the password. Changing it from the default is advised but it does not have to change to anything complicated.

> The default password 'password' is a well know password by Control Hub users, since it is the default for all Control Hubs. Staying with the default password significantly reduces network security. Changing from the default adds the element of network security back to the Hub by reducing the potential for access from outside sources.

Changing the Wi-Fi Band and Channel

As mentioned in the introduction section of this page, the Control Hub is capable of utilizing either the 2.4 GHz or 5 GHz Wi-Fi band. This change is also made within the Wi-Fi Settings.



The Robot Controller Console makes it easy to change between the 2.4 GHz an 5GHz bands. It is advised to check the Legal Android and Wi-Fi Band Capabilities table to determine which band to operate in.

Once a Wi-Fi band is chosen there are two options for dealing with Wi-Fi channels. One option is to let the Control Hub auto default on a channel. The other is to set a specific channel. Both options can be accessed via the drop down menu under the Wi-Fi channel section of the Wi-Fi settings.

It is valuable to know how to change the Wi-Fi Band and Channel as technical staff at an event can request to change those settings.

> The Wi-Fi band and channel can be changed via the Driver Station Application. For more information on how to make these changes from the Driver Station please see Managing the Wi-Fi Network section.

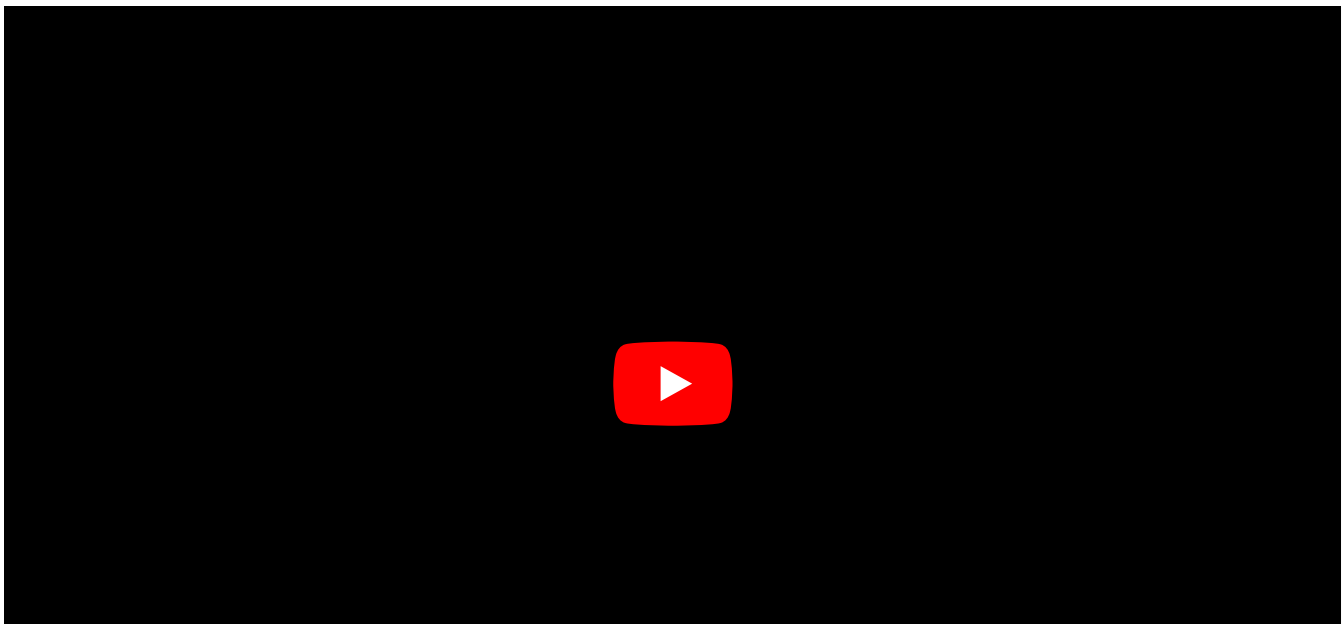
Connecting Driver Station to Control Hub

When you first receive your Control Hub (REV-31-1595), you will have to connect it to a supported Android Device, like a Driver Hub. The following section of the page will walk through how to pair a Driver Hub or Driver Station phone to a Control Hub.

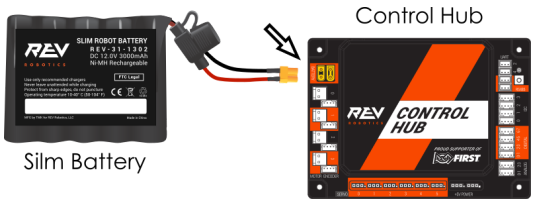
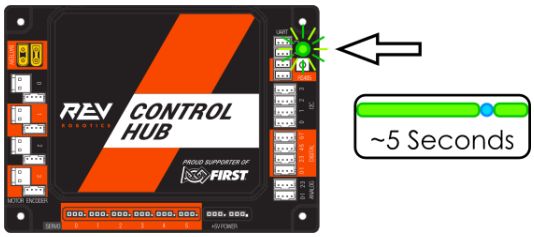
> This section assumes you have already gone through the process of setting up your Driver Station device. If you have not please go through the following guides for more information on getting started with a Driver Station:
>
> - Supported Android Devices and Wi-Fi Band Capabilities - To know what supported Android Devices can be used as a Driver Station
> - Getting Started with Driver Hub - To setup a Driver Hub
> - Configuring Your Android Devices - To setup a non Drive Hub supported Android Devices as a Driver Station

---

Connecting the Driver Station with the Control Hub



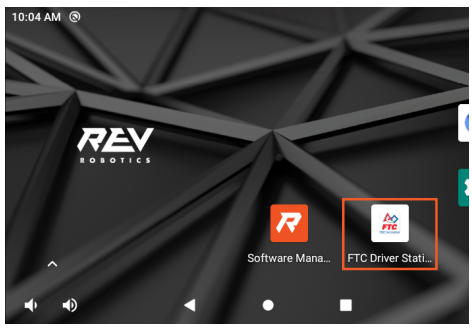
> The procedure for pairing the Driver Hub and the Control Hub only needs to be performed once for each set of hardware. If you replace your Driver Hub or Control Hub, this procedure will need to be repeated.

Power on the Control Hub by plugging the 12V Slim Battery into the XT30 connector labeled “BATTERY” on the Control Hub. You may also choose to include a switch between the Battery and Control Hub, if you prefer.	
The Control Hub is ready to pair with the Driver Station when the LED turns green. Note: the light blinks blue every ~5 seconds to indicate that the Control Hub is healthy.	

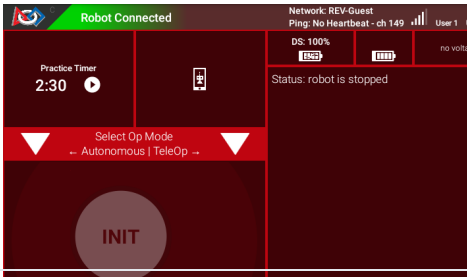
Once you have powered on your Control Hub follow through the process for connection to either a Driver Hub or a Driver Station phone.

Driver Hub

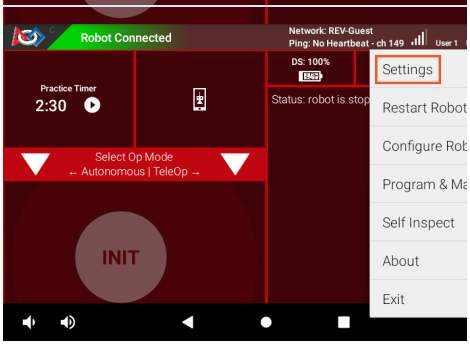
> This section assumes you have gone through the process of setting up your Driver Hub. If this is not the case please go to Getting Started with the Driver Hub and go through the process of bringing up your Driver Hub.

Open the Driver Station application from the HOME Screen.	

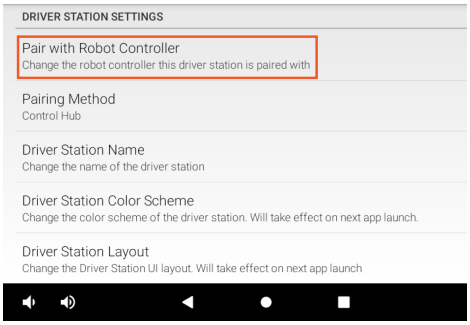
In the Driver Station application, click the three dots in the upper right corner to open the drop down menu.



In the drop down menu select **Settings.**

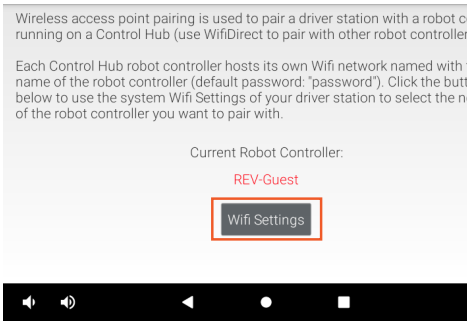


Select, “Pair with Robot Controller”.



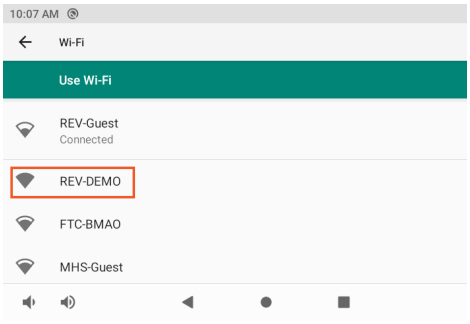
Select **Wi-Fi Settings.**

Note: In initial bring up for the Driver Hub you are asked to connect to a Wi-Fi network with internet, which is why this Driver Hub is already connected to a network. However, now the focus is on connecting to the Control Hub.



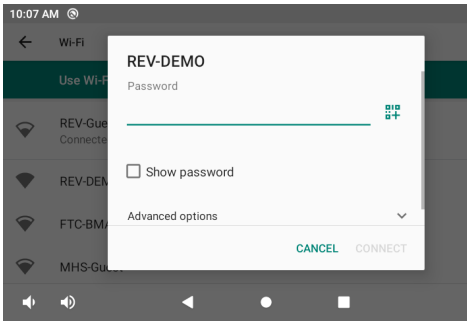
Select the name of the Wi-Fi network generated by your Control Hub. The default SSID name starts

with either “FIRST-“ or “FTC-“. In this example we want to choose our REV-DEMO Control Hub.

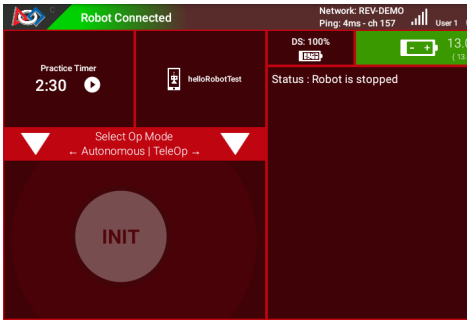


Enter the password to the Wi-Fi network in the password field. This defaults to “password”. Press **CONNECT**.

After pressing connect, press the back arrow at the bottom of the display until you return to the main driver station screen.



After a couple of seconds, the Driver Station page will indicate the network name, a ping time, and battery voltage.



Your Driver Hub is now paired with your Control Hub!

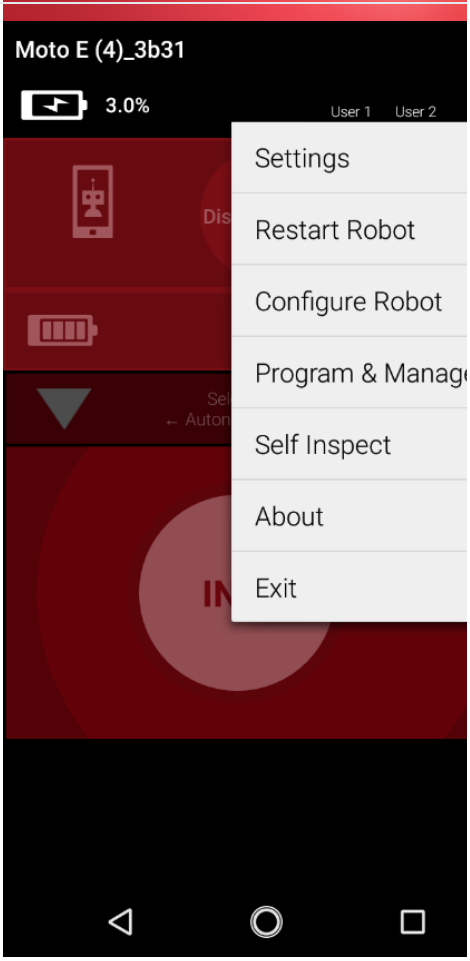
Other Supported Android Device

> ⚠ This section assumes you have gone through the process of setting up your Driver Station Android Device. If this is not the case please go to Configuring Your Android Device and go through the process of configuring an Android Device to act as the Driver Station.

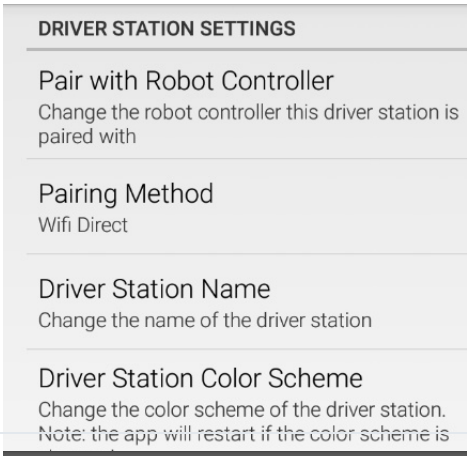
Power on your Android Device by holding down the power button.

Open the Driver Station application from the HOME Screen.

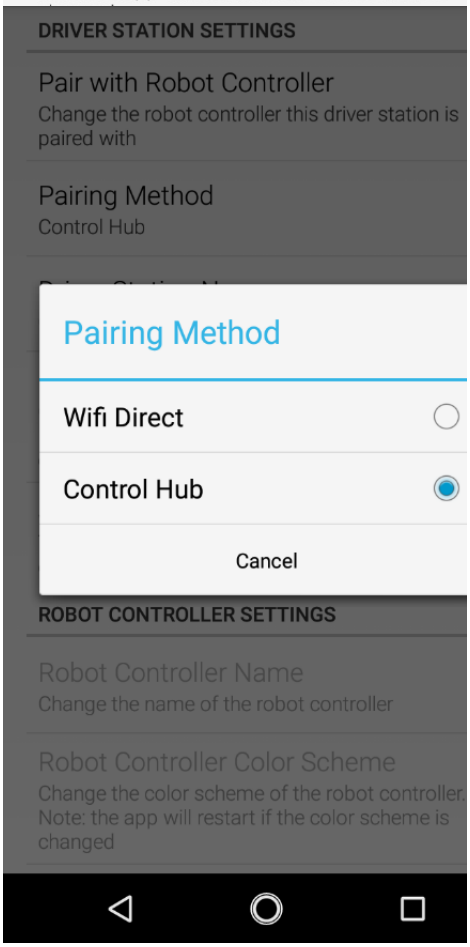
On the Driver Station page, open the menu from the top right corner, then select **Settings.**



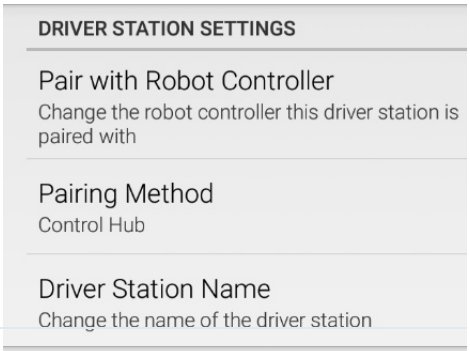
Select, **Pairing Method.**



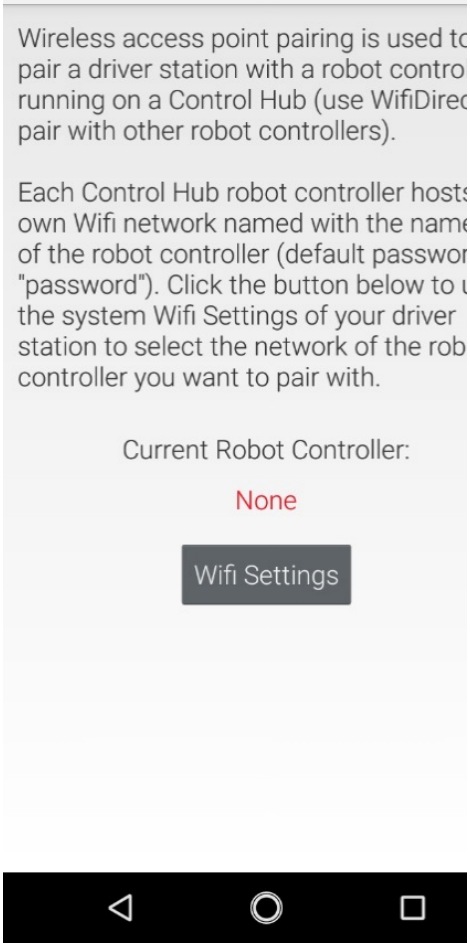
Select, **Control Hub**.



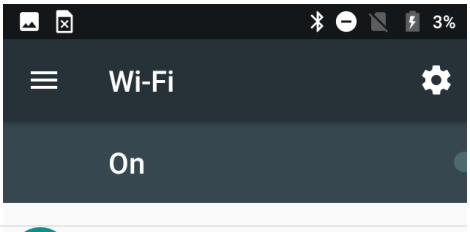
Select, **Pair with Robot Controller**.

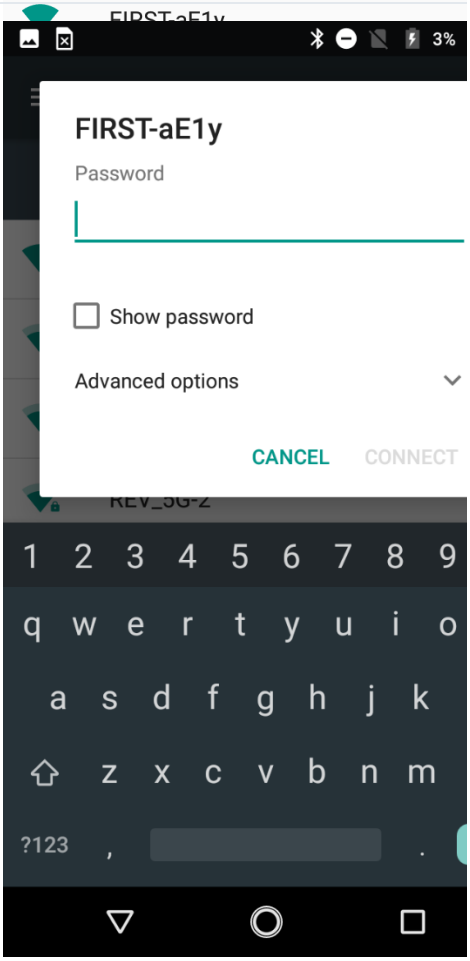


Select **Wifi Settings**.



Select the name of the Wifi network generated by your Control Hub. The default SSID name starts with either “FIRST-“ or “FTC-“.





Enter the password to the Wifi network in the password field. This defaults to “password”. Press **CONNECT**.

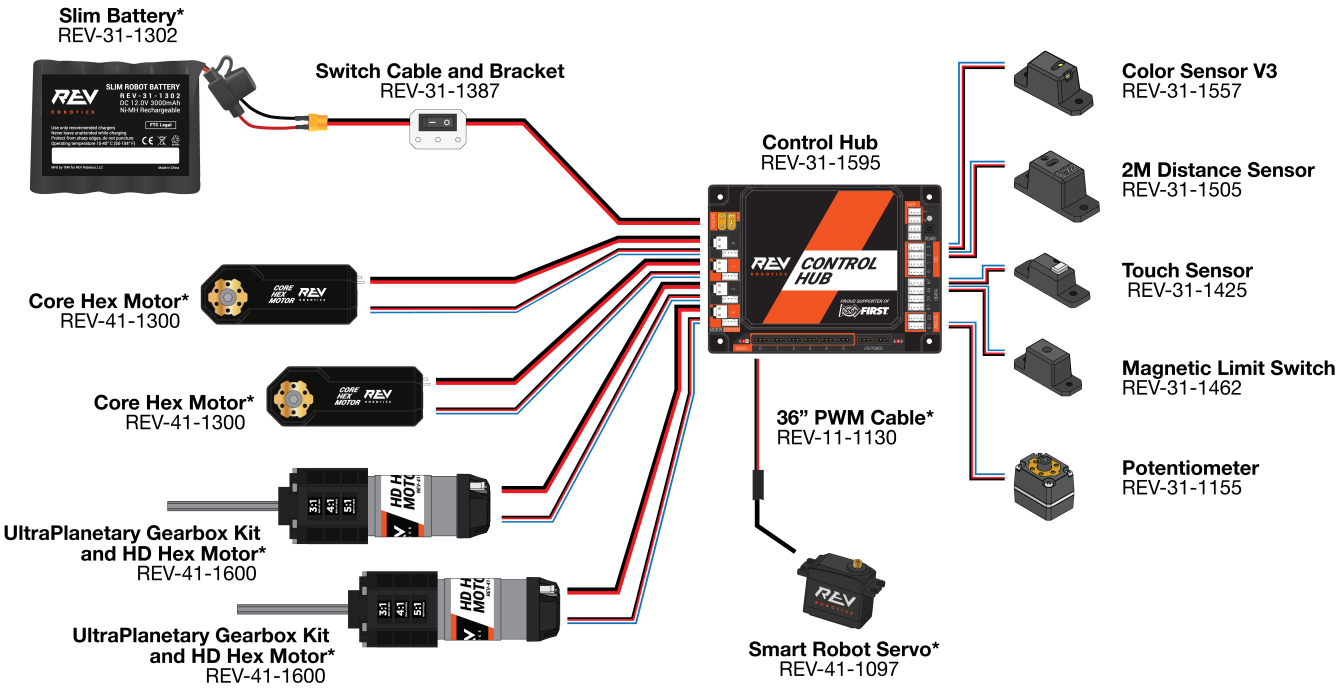
After pressing connect, press the back arrow at the bottom of the display until you return to the main driver station screen.

After a couple of seconds, the Driver Station page will indicate the network name, a ping time, and battery voltage.

Your Driver Station is now paired with your Control Hub!

Wiring Diagram

Before configuring your Control Hub, devices must be connected to the Control Hub. Below is a sample wiring diagram to show a sample of actuators and sensors usable with the Control Hub.



For more information on the connectors and cables used with the Control Hub see the links below:

- XT-30 - Power Cable
- JST VH - Motor Power
- JST PH - Sensors and RS485

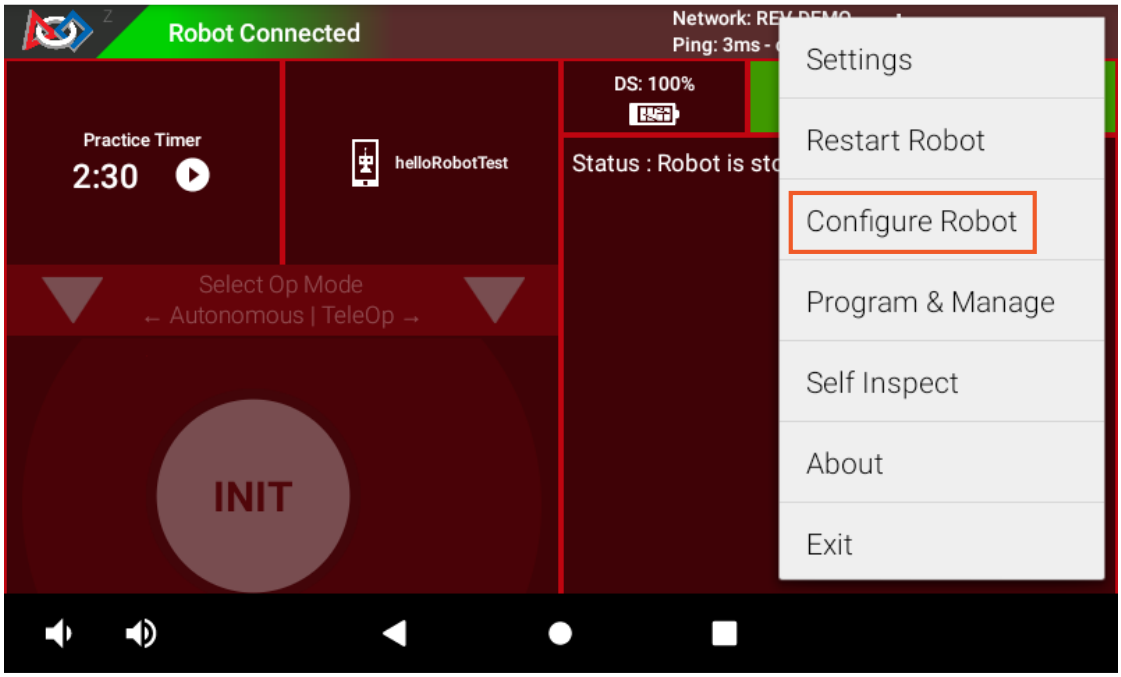
Next Steps

Being able to connect to the Robot Controller Console, connect a Driver Station to a Control Hub, and the basics of connecting Control Hub to different actuators and sensors is just the start. This section focuses on the next steps for using the REV Control System, including getting started with programming and best practices for managing the Control Hub and Slim Batteries.

Getting Started with Programming

Now that the Control Hub is setup, it is ready to start programming to control a robot! The Hello Robot programming guide walks through the necessary steps for getting started with programming. The guide has suggestions to choose the right programming tool, configuring your robot, and the basics of programming.

In order for the Control Hub to properly communicate with hardware components, you must perform a two part process known as hardware mapping. One of the most important, and commonly forgotten steps, when getting started programming is the creation of the configuration file, which is the first part of the hardware mapping process. A properly created configuration file, defines each hardware component with a unique name and a port type and number. After attaching hardware components to the Hub, use the Driver Station application to create a configuration, before beginning to program.



> For more information on the important of hardware mapping and how to configure your robot please see the Hello Robot - Configuration page.

Adding a Expansion Hub

Depending on the application more motor, sensor, or servo ports maybe needed. If your robot needs more motors adding an Expansion Hub might be necessary. Adding an Expansion Hub adds the same amount of hardware ports as one Control Hub (an additional four motor ports, six servo ports, and all the sensor ports) to the system.

> ⓘ For more information on how to add a secondary Expansion Hub please visit our Adding an Expansion Hub page.

Managing the Control Hub

The Control Hub and Expansion Hub are field upgradable devices. When new software is released with new features, bug fixes, and season specific changes users can update the device themselves. Checking for software updates at the start of September and then about every 6-8 weeks is recommended. To check for software updates you can use the REV Hardware Client or check the Managing the Control System section of the documentation.

> ⓘ Information on updating various pieces of software for the Control Hub, Expansion Hub, and Driver Hub can be found in the Managing the Control System section.

Slim Battery Best Practices

To maintain and care for your battery, reference the general best practices on the 12V Slim Battery (REV-31-1302) product page or the information below. This includes how to properly store, charge, and care for your battery on the long term.

All rechargeable batteries have a finite lifespan. Factors that affect lifespan include the number of discharge/charge cycles and the average loading of the battery. The following best practices can help maximize the lifespan of your battery:

- Charge rate
  - Minimum: 1.5A
  - Maximum: 3.0A
  - Recommended: 1.8A or 2.0A
- Do not overcharge
  - Disconnect the battery from the charger once it indicates a full charge.
  - Typical charge time does not exceed 2 hours.
  -

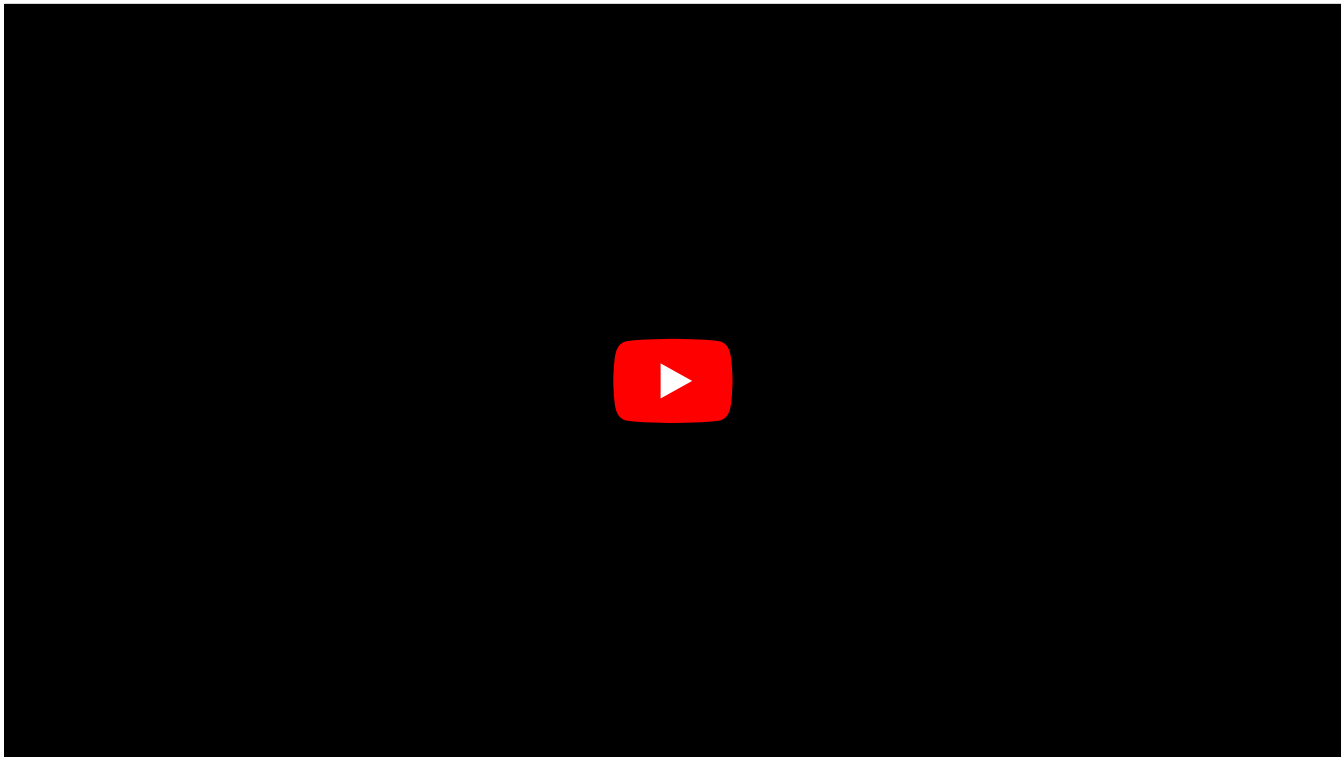
Do not charge a battery that hasn't been discharged significantly.
    - For example, running the robot under minimal load for a few minutes will not significantly discharge the battery.
- Minimum no-load voltage: 9.0V
    - Discharging the battery past 9.0V can reduce the lifespan of the battery and can permanently damage the cells.
    - Periodic dips below 9.0V when under load is expected and OK.
        - For example, don't forget to unplug your battery after you are finished running the robot and don't run your robot until it completely stops responding!
- Temperature
    - Let the battery cool before and after charging.
    - The battery may feel warm after heavy loading or after charging. This is normal.

Getting Started with Driver Hub

After receiving the Driver Hub it is advised to unbox the device, plug the Driver Hub in to charge over USB-C, and power on the Driver Hub. Below is the initial bring up process of the Driver Hub.

Required Materials

- Driver Hub (REV-31-1596)
- USB-A to USB-C Cable
- USB-A Wall Charger



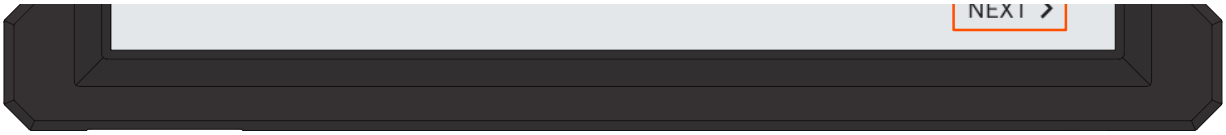
Battery Installation

To install the battery place it with the REV Logo out and the -/+ located near the contacts for the device. Add on the rear door and screw in using the included M3 hardware.

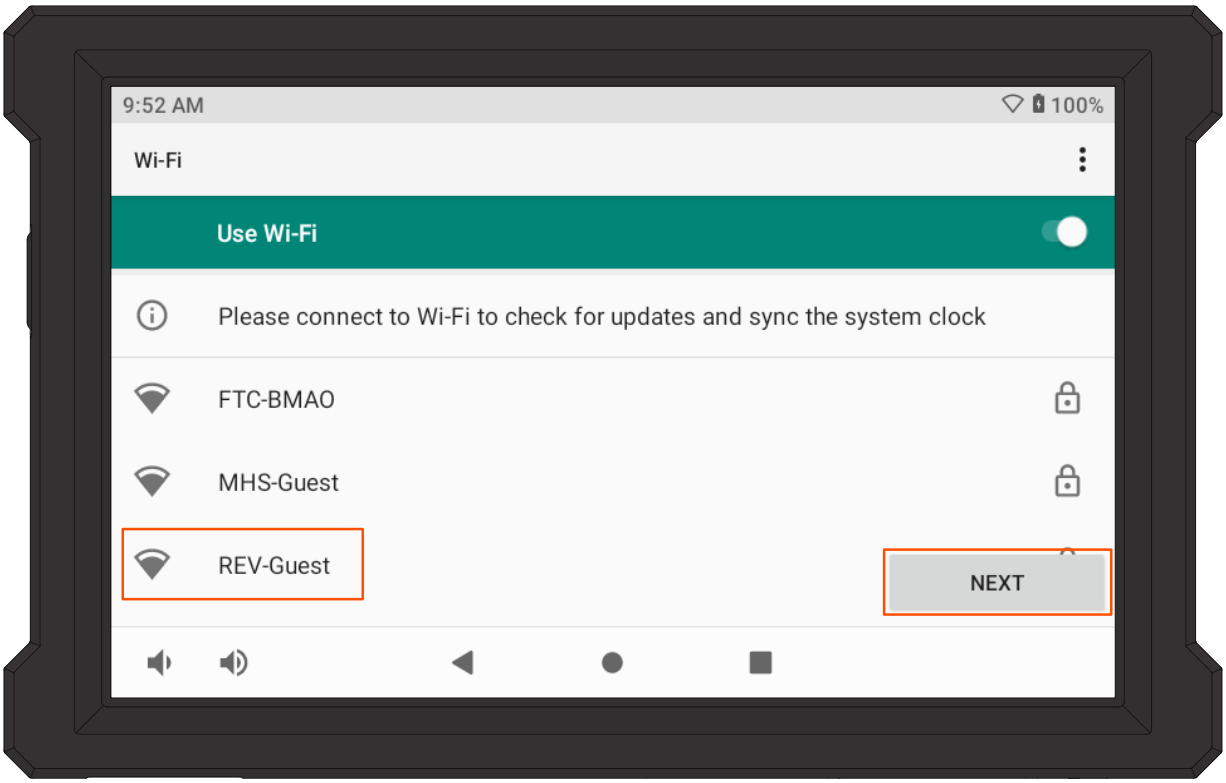
> Before continuing to set up the Driver Hub allow the battery to charge over USB-C or keep the Driver Hub plugged into a power source during set up.

Setting up the Driver Hub

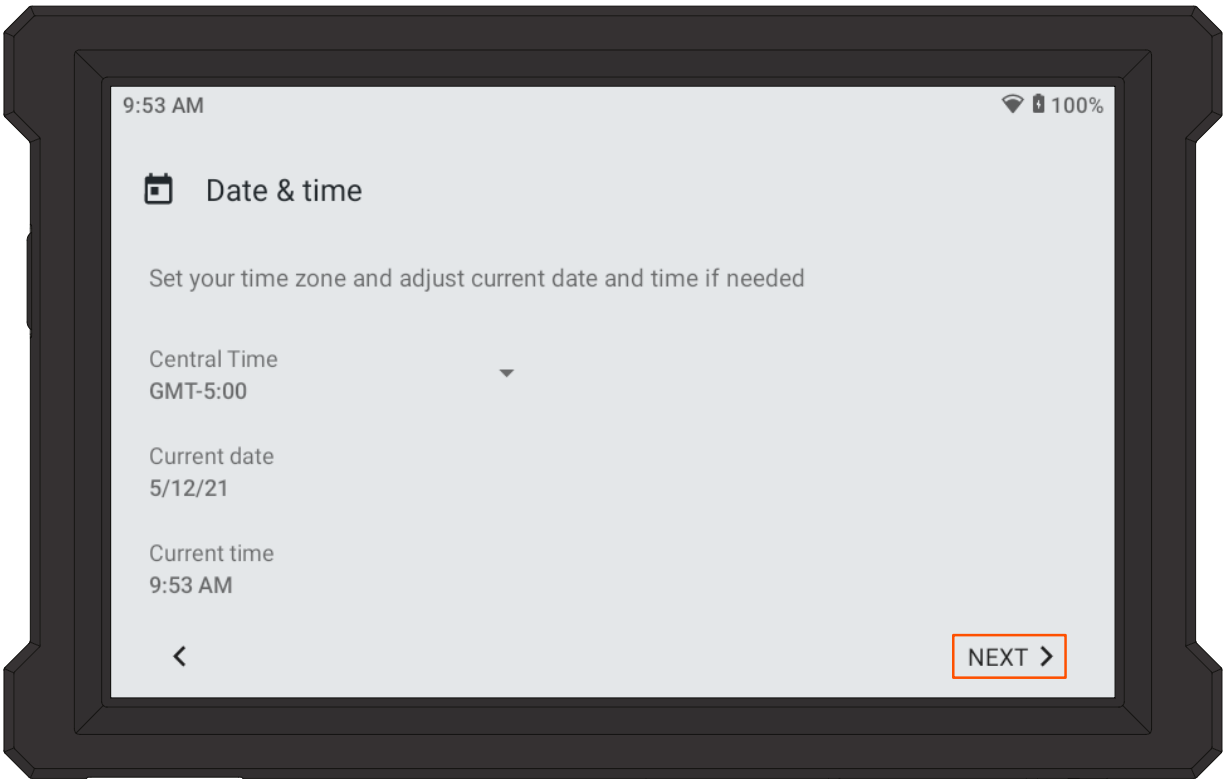
When the Driver Hub is first powered up, or a factory reset is performed, an initial set up process is needed. Start by selecting next on the main screen to continue.



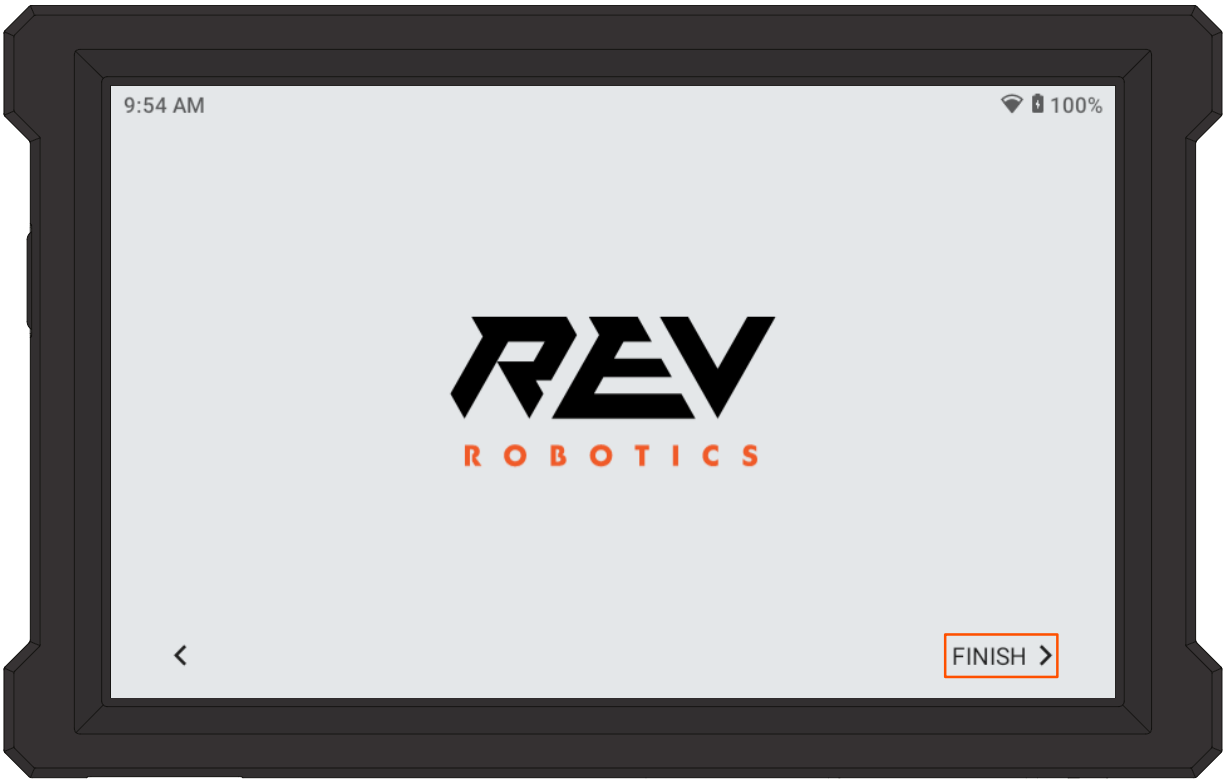
Select a local Wi-Fi network that has access to the internet, enter in the password for that network if required, and select next.



Time zone and date of the device are set by the local Wi-Fi network. Confirm these settings are correct before proceeding by the Next button.



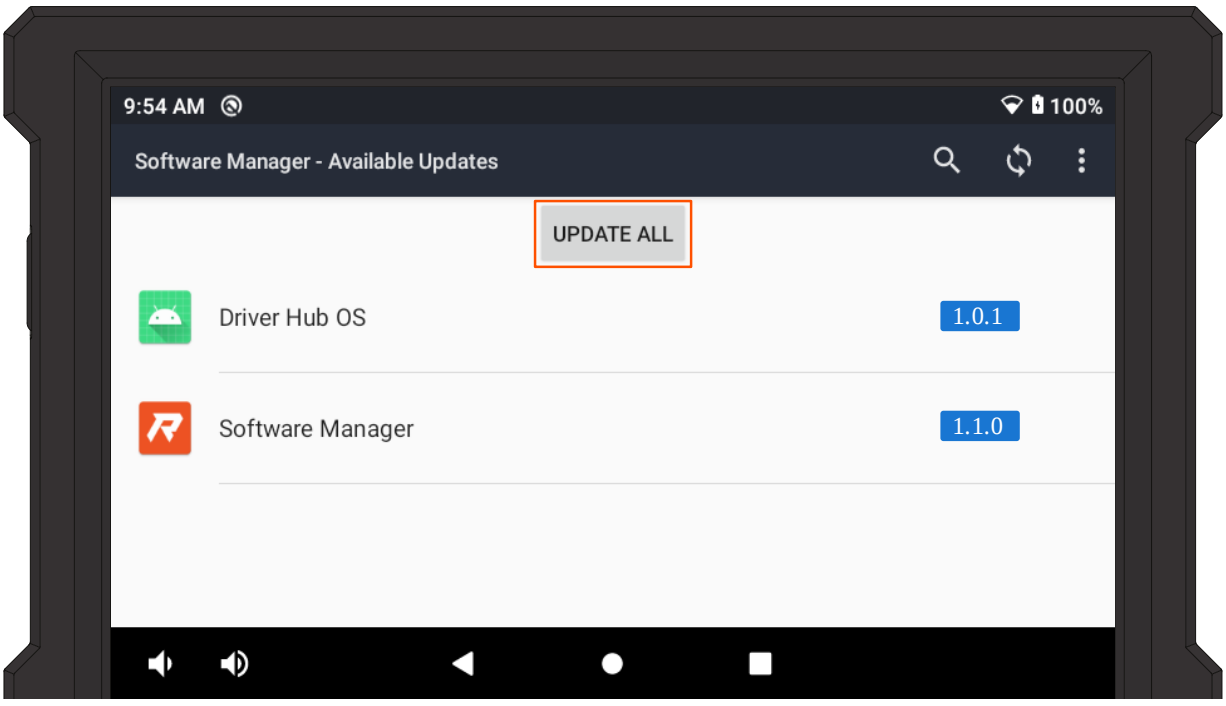
Initial set up is complete! Select Finish to operate the Driver Hub.

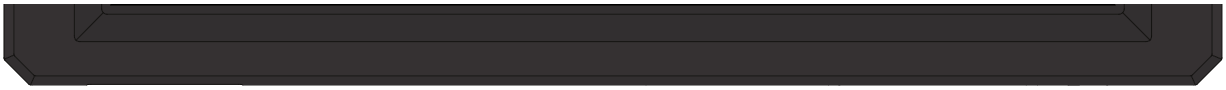


Initial Update

After setting up the Driver Hub, the Software Manager application will open. Select the Update All button to start the download and installation of software updates for the Driver Hub.

> The updates can take several minutes to complete. Make sure the Driver Hub is charged or plug in the Driver Hub during the updating process.

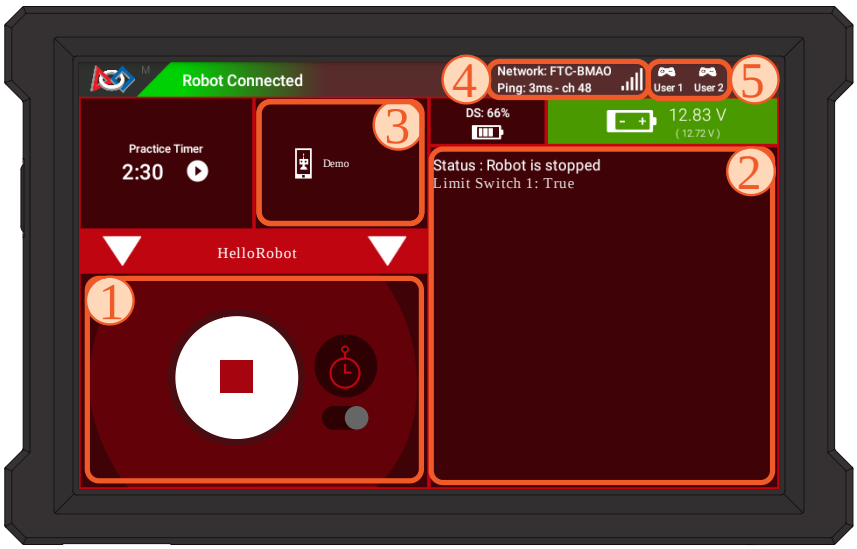


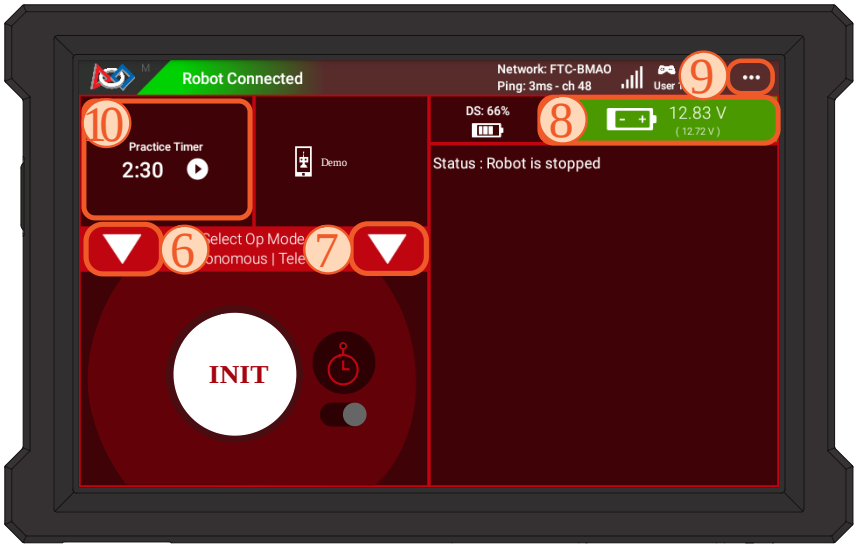


Now the Driver Hub is ready to connect to a Control Hub!

Navigating the Driver Station Application

Once the Driver Hub is connected to a Control Hub, you will have access to the entire Driver Station Application interface. Like any application, understanding the major components that make up the Driver Station Application interface, will maximize your ability to utilize the application efficiently. Consider the following components:

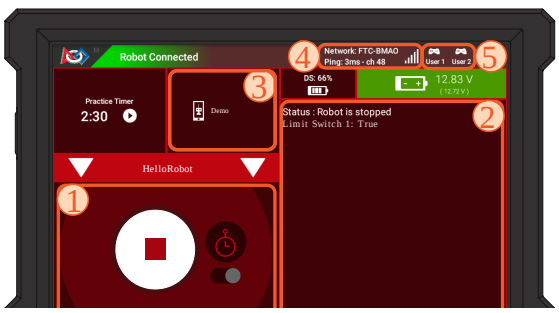


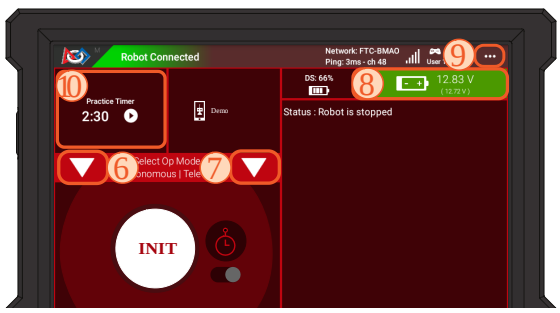


1	Initialize, start, and stop programs	Only available when a program has been selected.
		Displays telemetry outputs.

2	Telemetry display	Displays any system warnings and error codes
3	Active configuration	Displays which configuration fi is currently active. If this section says **<no config file>** you will need to activate o create a configuration file.
4	Network information	Displays Control Hub SSID Name, signal strength, and pin time.
5	Gamepad connections.	See Connecting Gamepads fo more information.
6	Autonomous drop down menu	Drop down menu that displays all autonomous programs save on the Control Hub.
7	Teleop drop down menu	Drop down menu that displays all teleop programs saved on t Control Hub.
8	System power display	Displays the amount of battery voltage powering the robot, when connected to a Control Hub.
9	Settings drop down menu	Access settings, configure the robot, restart the robot, check t see if your system meets competition inspection requirements and more.
10	Practice Timer	A built in timer that can be use to to practice for different portions of a match.

Tips and Tricks





If you tap on area 4, it will switch to displaying the link speed and signal strength. It will go back to showing the signal strength and ping time if you tap it again.

The smaller number in area 8 is the lowest voltage that the Driver Station has observed from the Robot Controller. If you tap area 8, the lowest voltage will be reset to the current voltage.

Connecting Gamepads

The Driver Station Application allows for the connection of two gamepads. When working with the Driver Hub these gamepads can be plugged into any of the three USB 2.0 ports. Once the gamepads are plugged in, you will need to initialize them. For the following example we will use PS4 controllers, such as the Etpark Wired Controller for PS4 (REV-39-1865).

To initialize the gamepad that will act as User 1 (gamepad1, in code) press the **options** button and the `X` button on the gamepad at the same time. To initialize User 2 (gamepad2, in code) press the **options** button and the `O` button at the same time.

> For the Logitech F310 Gaming Controller and Xbox 360 Controller for Windows, press **start** and **A** at the same time to initialize User 1 and **start** and **B** at the same time to initialize User 2.

Adding More Motors

The Control Hub (REV-31-1595) and Expansion Hub (REV-31-1153) can each drive up to four DC brushed motors. As mechanisms are added to the robot the number of motor ports may not be sufficient. There are two ways to add more motors to the Control System, either the SPARKmini Motor Controller (REV-31-1230) or adding an Expansion Hub. The Following two rules give a general idea of when to choose one method over another:

- If one or two motors are needed, consider using the SPARKmini Motor Controller.
- If three to four additional motors are needed, consider adding an Expansion Hub.

For additional information on how to use a SPARKmini or how to add an Expansion Hub, visit the linked pages!

SPARKmini Motor Controller

Adding an Expansion Hub

SPARKmini Motor Controller

The SPARKmini Motor Controller (REV-31-1230) is an inexpensive in-line brushed DC motor controller designed to give *FIRST*® Tech Challenge teams more bang for their buck. It offers the same performance characteristics as the REV Control Hub (REV-31-1595) or Expansion Hub (REV-31-1153) motor ports in a small 60mm x 22mm footprint. Now FTC teams can add a SPARKmini Motor Controller to utilize more than four DC motors from a single Hub in a space-efficient package.

Power and Motor Connections

The SPARKmini has three integrated wires with connectors dedicated to power, control, and the motor; one XT30 connector for power, one 3-wire servo-PWM connector for control, and one JST-VH connector for the motor. The figure below shows each of these connections.

Connect the power wire to a free XT30 port on the REV Control Hub , REV Expansion Hub (REV-31-1153), or through an XT30 Power Distribution Block (REV-31-1293) that is connected to a free Control/Expansion

Hub XT30 port. Connect the control wire to an open servo port on the hub and the motor wire to a JST-VH port on a motor, like the REV HD Hex Motor (REV-41-1301) or the REV Core Hex Motor (REV-41-1300).

> ⚠ DO NOT reverse polarity on the power input connections. The SPARKmini does not contain reverse polarity protection. This can permanently damage the SPARKmini and will void the warranty.

> ⚠ DO NOT swap the motor and power connections. This can result in uncontrolled motor operation and can permanently damage the SPARKmini, voiding the warranty.

Servo-PWM Input

A motor's speed is controlled by varying the voltage that is applied to it. The SPARKmini's output voltage can be controlled by sending it an extended-range servo-PWM pulse. The extended 500µs to 2500µs servo-pulse corresponds to full-reverse and full-forward rotation with 1500µs as the neutral position (no rotation). The pulses are proportionally related to the motor output duty cycle, therefore variable speed can be achieved with pulses in between the extremes. The following table describes the pulse ranges in more detail.

Table - Control Signal Pulse Ranges

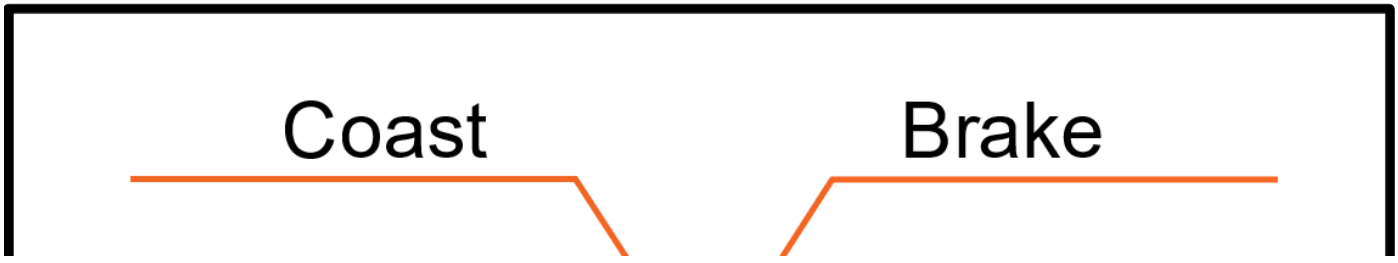
Pulse Width (*p* in µs)				
Full Reverse	**Prop. Reverse**	**Neutral**	**Prop. Forward**	**Full Forward**
$p \leq 500$	$500 < p < 1490$	$1490 \leq p \leq 1510$	$1510 < p < 2500$	$2500 \leq p$

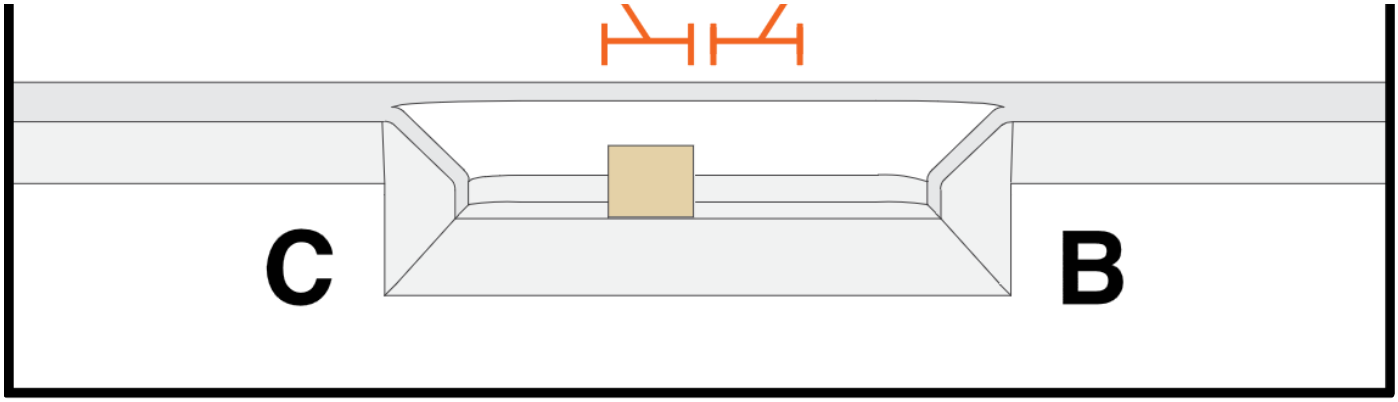
Zero-Power Behavior

When the SPARKmini is receiving a neutral command it will not provide any power to the attached motor. There are two options for how the SPARKmini handles this zero-power state:

Brake - Motor terminals are shorted to each other to dissipate electrical energy, effectively braking the motor.
Coast - Motor terminals are disconnected, allowing the motor to spin down at its own rate.

The zero-power behavior can be selected via a switch located towards the center of the SPARKmini housing, shown in Figure 2. Each mode can be selected by sliding the switch to either the Brake (B) or Coast (C) positions.

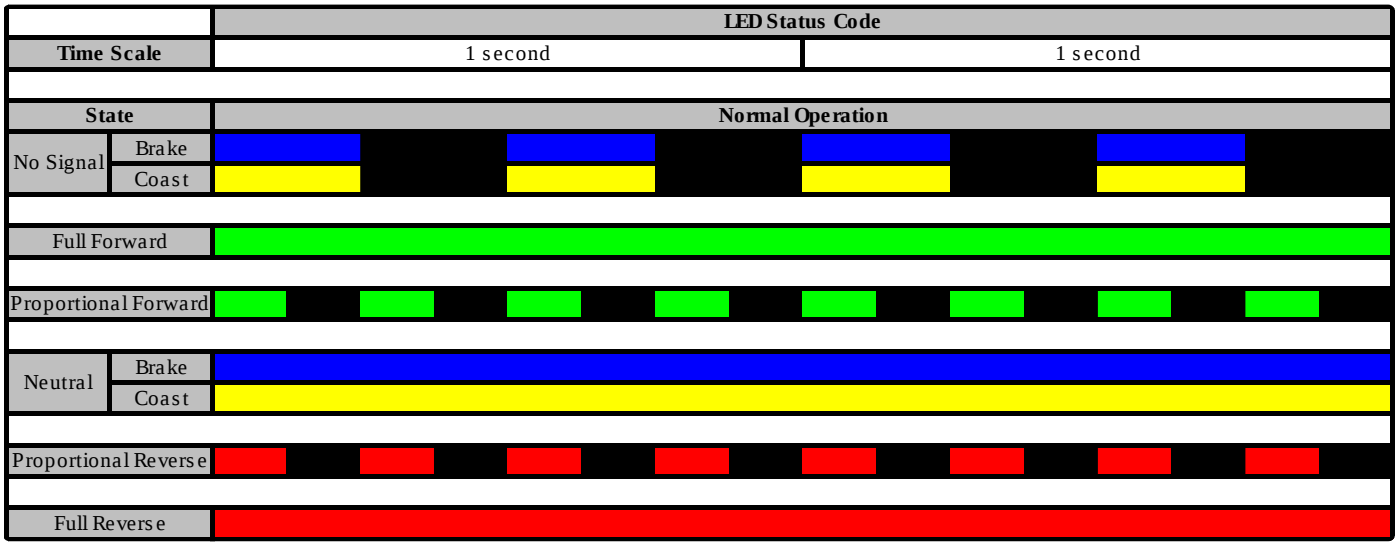




Coast/Brake Switch

The SPARKmini will indicate whether it is in Brake or Coast mode via the Status LED, located in the center of the housing, whenever it is outputting zero-power. Solid or flashing blue indicates Brake Mode while solid or flashing yellow indicates Coast Mode. See the LED Status Codes section for more details.

LED Status Codes



Specifications

Parameter	Min	Typ	Max	Unit
Supply voltage range (VIN)	6.0	12	20	V
Supply voltage absolute maximum	-	-	25	V
Continuous output current	-	-	15	A
Peak output current	-	-	20	A

Output voltage range	- VIN	-	+ VIN	V
Output frequency	-	10	-	kHz
Input pulse width range	500	-	2500	µs
Input frequency	16	50	200	Hz
Input timeout	-	65.5	-	ms
Input deadband	-	±10	-	µs
Input low-level voltage	-0.3	-	0.8	V
Input high-level voltage	2.0	5.0	5.3	V
Weight	-	0.87	-	oz
Dimensions (excluding wires)	-	60 x 22 x 12	-	mm

Adding an Expansion Hub

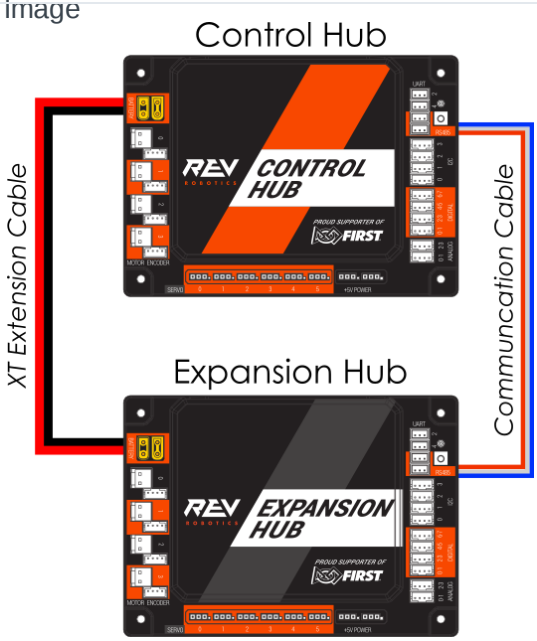
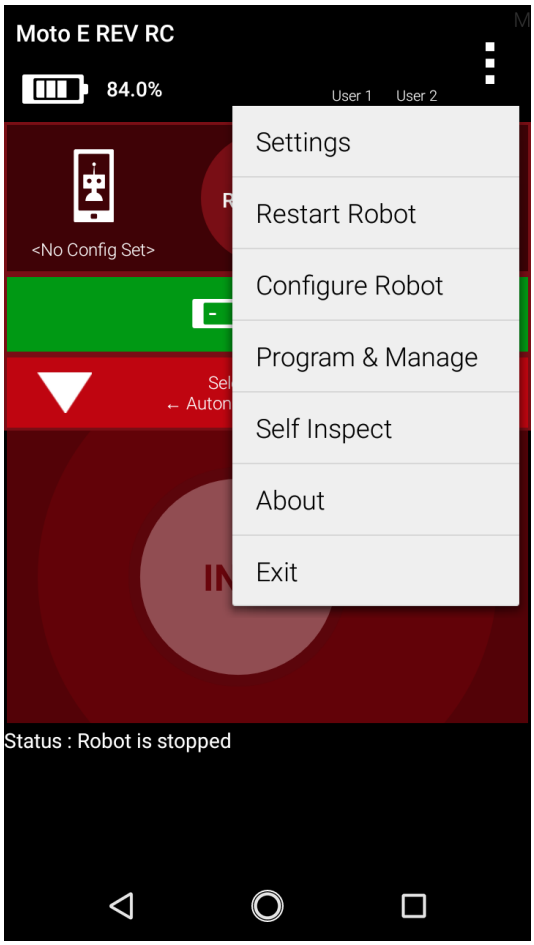
If you want to use more than 4 motors or 6 servos, you can add an Expansion Hub to your robot. An Expansion Hub (REV-31-1153) can be added to a Control Hub (REV-31-1595) or another Expansion Hub. The Expansion Hub has all of the same ports as the Control Hub but without the wireless capability.

Control Hub vs Expansion Hub in FIRST

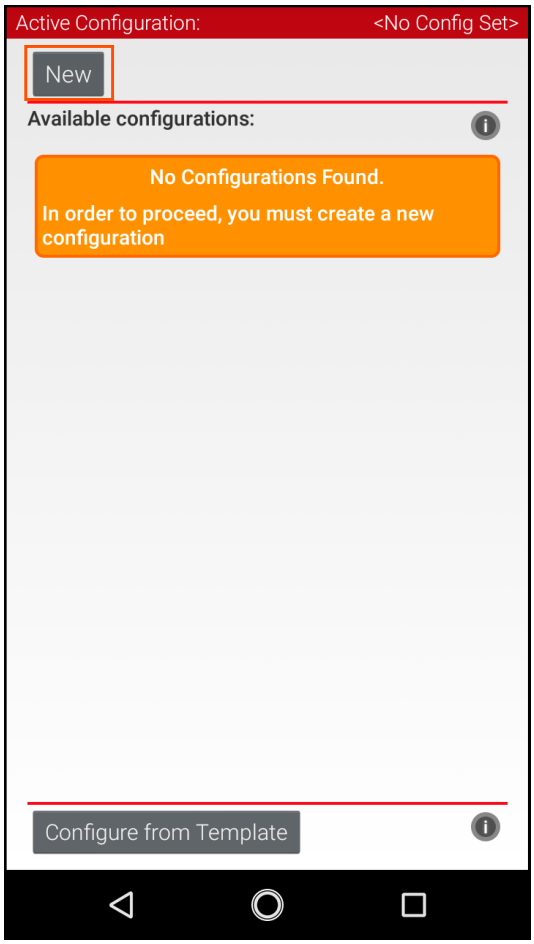
FIRST Tech Challenge	FIRST Global
FIRST Tech Challenge teams may use one (1) Control Hub and may add one (1) Expansion Hub starting in the 2020-2021 season. Read the official FTC Game Manuals for complete game rules.	FIRST Global teams must use one (1) Control Hu and may add one (1) Expansion Hub to their robo Read the official FIRST Global manual for complete game rules.

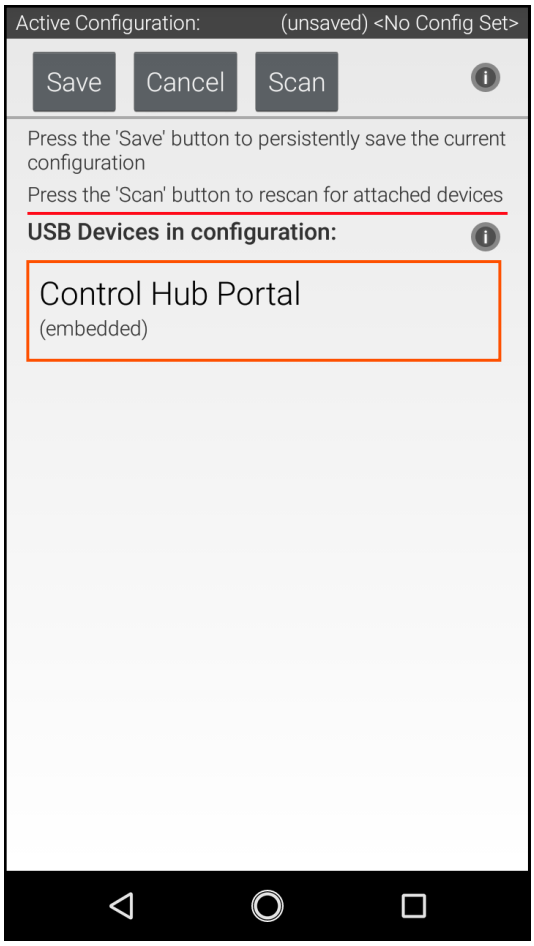
> If you are using a configuration file from a 5.5 or earlier version of the Robot Controller Application, you will need to create a new configuration file.

Adding an Expansion Hub to your Robot

Step	image
Use the XT Extension Cable to connect power between the Control Hub and the Expansion Hub. Use a 3-pin JST PH cable to connect the RS485 port on the Control Hub to the Expansion Hub.	
From the Driver Station choose “Configure Robot”	

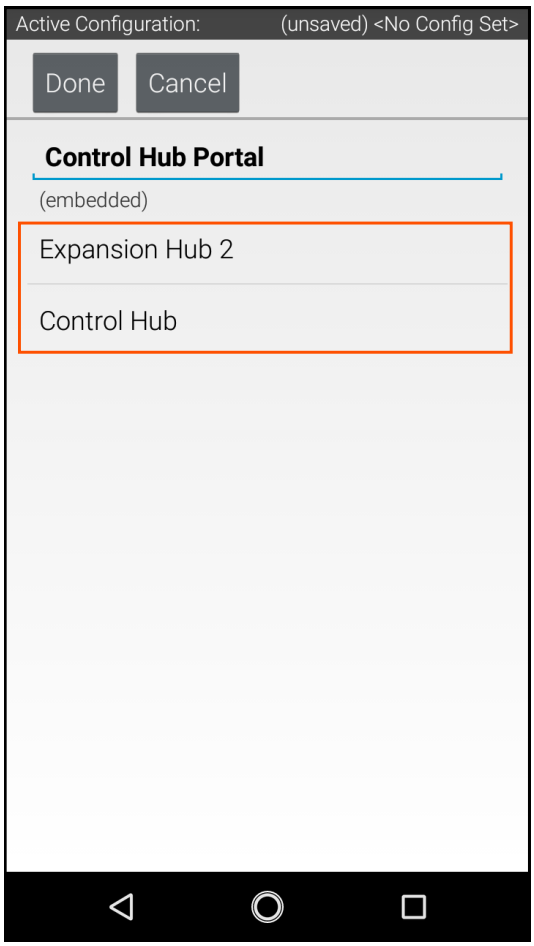
Select “New” in the top left hand corner.





Select “Control Hub Portal”

Note: This will show an Expansion Hub Portal if using an Android Device as a Robot Controller



Now you have two Hubs to choose from. Either the Control Hub or the Expansion Hub.

“Expansion Hub 2” is the connected Expansion Hub that is communicating over RS485.

Configure and program as necessary. Please see the Configuration section of for an overview of configuration.

Note: If using an Android Device as a Robot Controller there will be two Expansion Hubs located here. The Expansion Hub Address may need to change so they do not conflict.

Troubleshooting the Control System

General Troubleshooting

One of the key aspects of troubleshooting is understanding the most common issues that occur in a system. Once those problems, and their indicators, are defined a flow has to be created. For example, a check engine light in a car indicates any number of issues. When a cars check engine light comes on, a mechanic pulls the codes from the car to narrow down the issue to a specific part of the engine. Even if the code leads to a specific part of the engine, like the transmission, it is not always indicative of the exact problem. However, there is a process flow. Each step narrows down the problem to a potential solution. Troubleshooting the REV Control system is no different!

> ⓘ The status LED is the REV Control System equivalent to the check engine light mentioned in the example. *Visit the LED Blink Code section to understand what each code is and what it indicates.*

Many issues can be solved by systematic troubleshooting without needing to contact REV Support. Take a look at the troubleshooting tips below for help in determining the cause of the issue you are seeing. Should you need to contact us, describing the steps you've taken in detail will help us get you up and running quickly. The section is divided into general best practices, Control Hub (REV-31-1595) troubleshooting and Expansion Hub (REV-31-1153) troubleshooting.

General Best Practices

Before diving into common troubleshooting paths its important to understand the general guidelines, or best practices, for Control System Health.

- **Charge the Battery** - While a charged battery and phone are crucial to a healthy control system in general; it is also helpful to ensure batteries and phones are charged before a match.
- **Update** - The applications, firmware, and operating system have periodic updates to improve the control system. Keeping the control system up to date ensures the best performance!
- **Isolate the Issue -** This is key to effective troubleshooting. Many issues can show the same symptom, so eliminating failure points one at a time is critical to finding the root cause.

> ⚠ DO NOT plug a battery charger into either the Control Hub or Expansion Hub. It will damage the Hub and cause eventual device failure

Maintaining and taking care of the 12V Slim Battery is also important for troubleshooting purposes. All

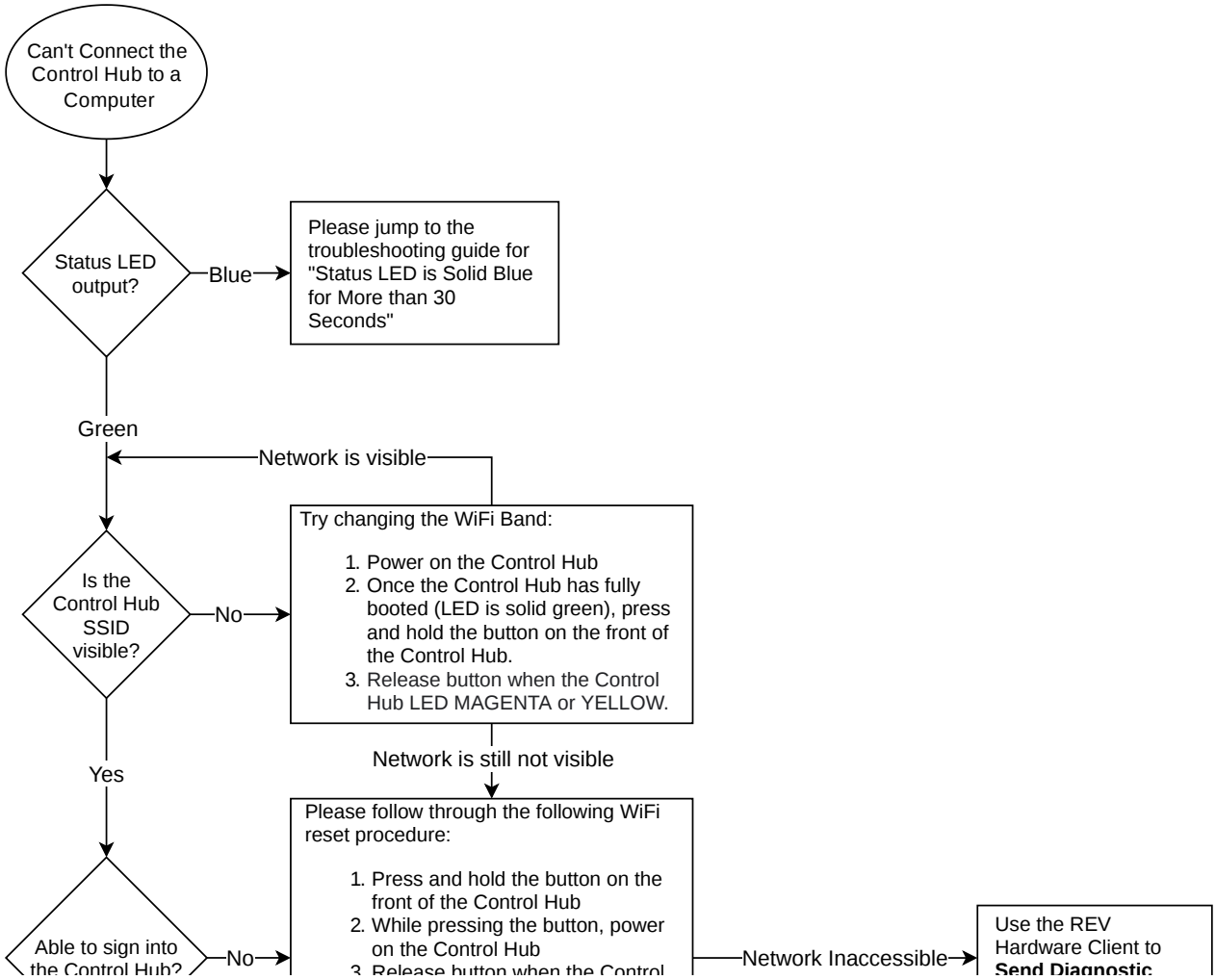
rechargeable batteries have a finite lifespan however following the best practices for the 12V Slim Battery can extend the lifespan of the battery.

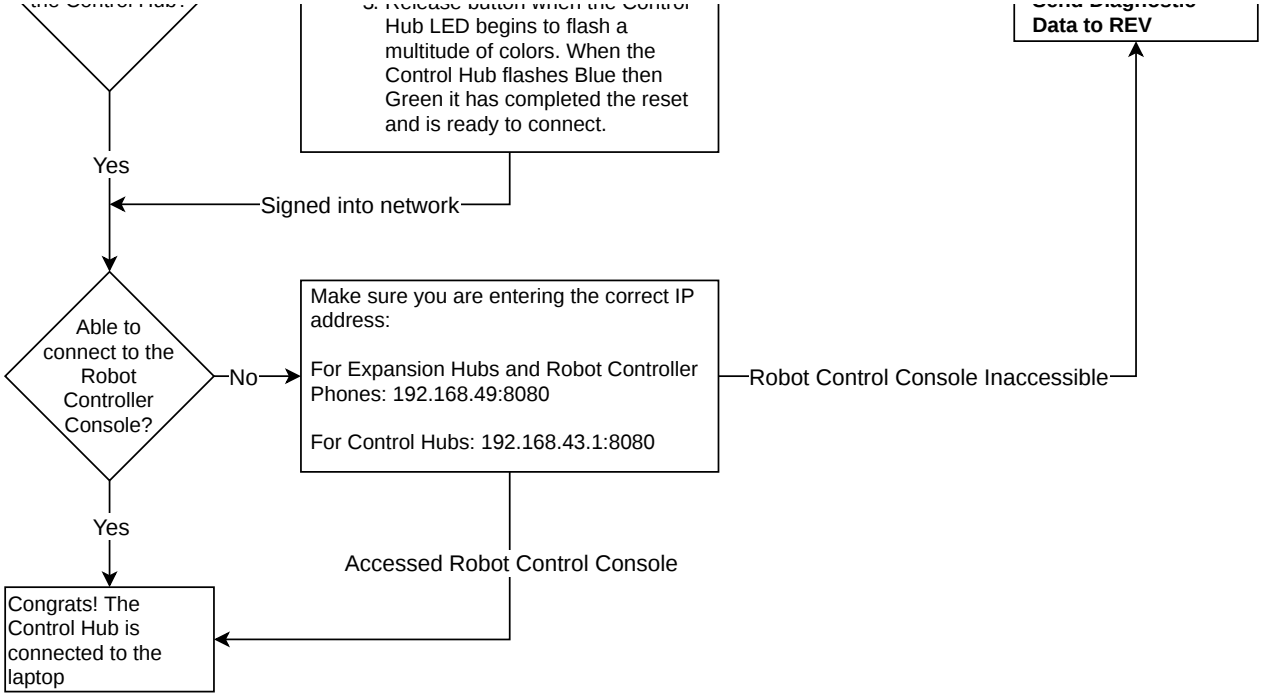
Control Hub Troubleshooting

The following questions consider common indicators of issues seen in the Control Hub. Think about the potential indicators your Hub is currently exhibiting and consider the following questions:

Is the Driver Station device unable to connect to to the Control Hub Wi-Fi?	Yes
Is the Driver Station connected to the Wi-Fi but not showing a ping or any other signs of communication?	Yes
Has the Status LED been solid blue for longer than 30 seconds (after start up)?	Yes

Can't Connect the Control Hub to a Computer





> The Wi-Fi reset will down grade the Wi-Fi connection to 2.4GHz. If you have an android device with 5GHz you may want to switch the Wi-Fi Band in order to run on 5GHz. *Check out the Updating Wi-Fi Settings Section to learn more about making this switch.*

External factors, such as local Wi-Fi environment, play a part in the ability to establish or maintain a connection between a Control Hub and a computer. Like all aspects of of troubleshooting its important to isolate an issue by asking questions and discovering the answers! As you work on troubleshooting consider the following questions:

- **What is your local Wi-Fi environment like?**
  - Local Wi-Fi environment effects the consistency of a connection to the Control Hub. Use a Wi-Fi analyzer to check the local environment for channels that are cluttered with Wi-Fi networks. Change the Control Hubs Wi-Fi channel to a channel with the least amount of overlap with other networks.
- **Are you connected to another Wi-Fi network?**
  - The Control Hub produces a non internet Wi-Fi connection. Settings on the individual computer may cause the device to jump to a local, remembered network that produces an internet connection.
- **Are you in a school or a place of business?**
  - In addition to the amount of local networks in an environment its important to understand what those local networks are capable of. For instance, some school districts have security measures in place that block unauthorized Wi-Fi access points. Talk to your local Wi-Fi adminstrator to find out what you need to get the Control Hub as an approved network.

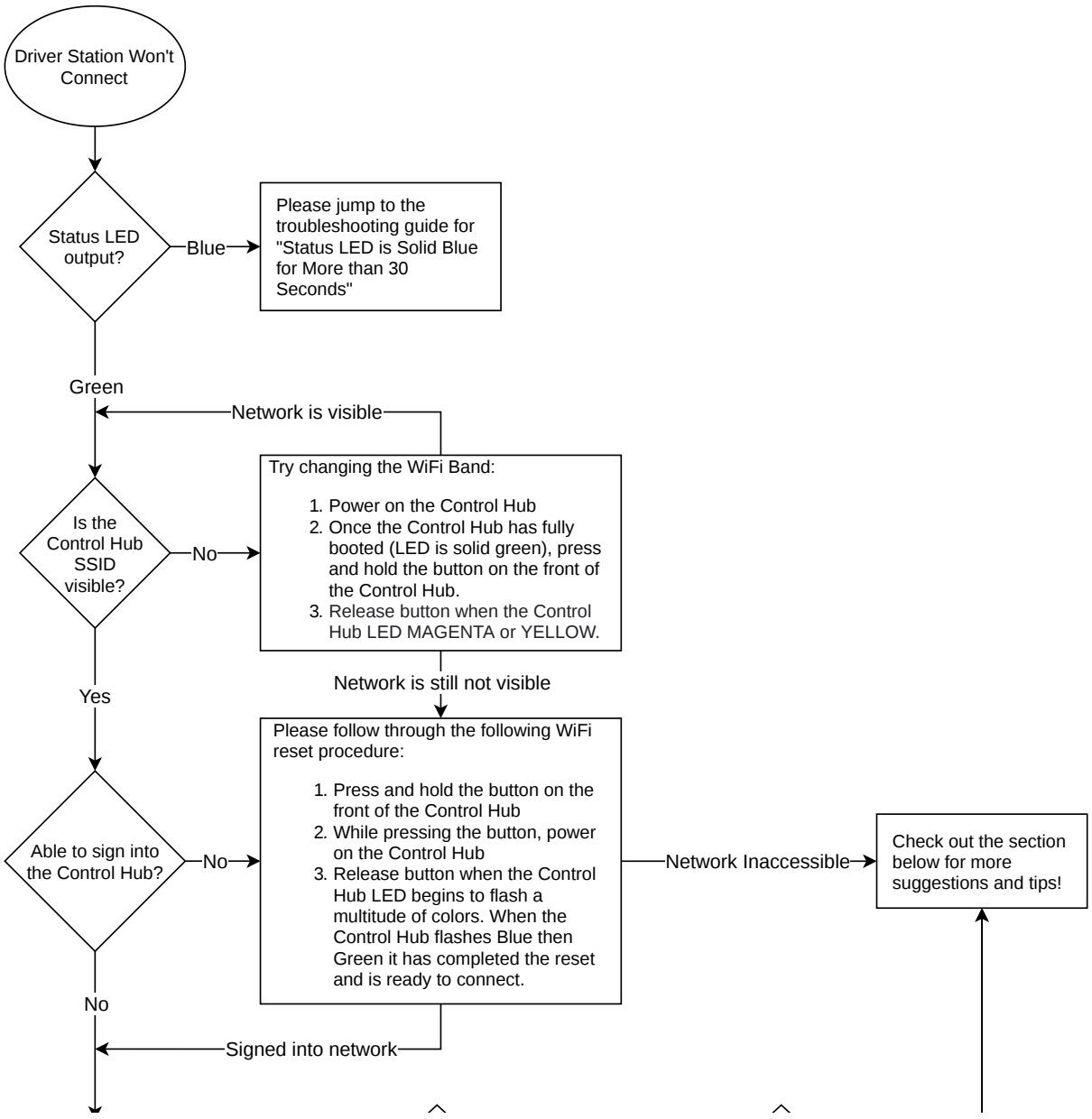
> If the Control Hub SSID is not shown in the list of available Wi-Fi networks, try manually entering the Control Hub SSID to see if that allows you to connect.

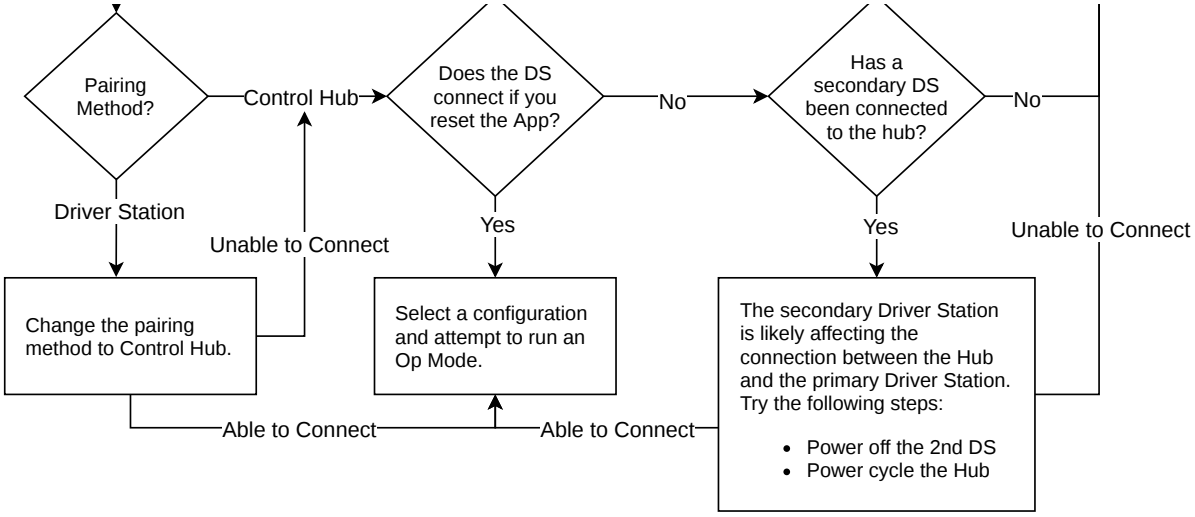
Contact REV Support with details of the troubleshooting information you have collected such as the answers to the questions above and the outcome of your troubleshooting thus far. It will also help to send logs or other diagnostic data to REV Support.

> Need help getting the Log Files to send to REV Support? See Downloading Log File for more information.

Driver Station Won't Connect

> Information in this flowchart is for the initial bring up of connecting the Control Hub with a Driver Station. For issues with intermittent connection or periodic connection drops please check out the information below this flowchart.





> The Wi-Fi reset will down grade the Wi-Fi connection to 2.4GHz. If you have an android device with 5GHz you may want to switch the Wi-Fi band in order to run on 5GHz. *Check out the Updating Wi-Fi Settings Section to learn more about making this switch.*

External factors, such as local Wi-Fi environment, play a part in the ability to establish or maintain a connection between a Control Hub and a Driver Station device. Like all aspects of of troubleshooting its important to isolate an issue by asking questions and discovering the answers! As you work on troubleshooting consider the following questions:

- **Is your system operating on a 2.4 GHz band or 5GHz band?**
  - REV recommends, if you have a dual band Driver Station device, that you operate on the 5GHz Wi-Fi band. Check out the Updating Wi-Fi Settings section to learn more about making this switch.
- **What is your local Wi-Fi environment like?**
  - Local Wi-Fi environment effects the consistency of a connection to the Control Hub. Use a Wi-Fi analyzer to check the local environment for channels that are cluttered with Wi-Fi networks. Change the Control Hubs Wi-Fi channel to a channel with the least amount of overlap with other networks.
- **Are you in a school or a place of business?**
  - In addition to the amount of local networks in an environment its important to understand what those local networks are capable of. For instance, some school districts have security measures in place that block unauthorized Wi-Fi access points. Talk to your local Wi-Fi administrator to find out what you need to get the Control Hub as an approved network.
- **Does the the Driver Station connect to the Control Hub until a mechanism is run?**
  - Certain mechanisms draw enough power from the Control Hub to put a strain on the battery. If you notice a drop in displayed voltage when you start a code, or when a particular mechanism is run, this may be indicative of a brown out condition. Other indicators include:
    - The Driver Station throwing errors about power to the system
    - The Driver Station making a disconnect sound
    - The voltage on the Driver Station showing 9 volts or lower when running code
    - Motors running at lower speeds then what they have been set to run
  -

To remedy this issue check out our instructions on proper battery care.

> If the Control Hub SSID is not shown in the list of available Wi-Fi networks, try manually entering the Control Hub SSID on the Driver Station to see if that allows you to connect.

If you are still experiencing connection issues, once you have gone through the flowchart and worked on addressing the potential root of connection issues describe in the list above, start looking for patterns in the behavior. How often does this behavior appear? Are there certain things that happen around the same time the disconnects happen? The following list provides some ideas on what sort of patterns you might see:

- The Control Hub connects fine when a team member takes it home but doesn't seem to like to connect at school.
- The Control Hub connects fine until you start driving the robot around.

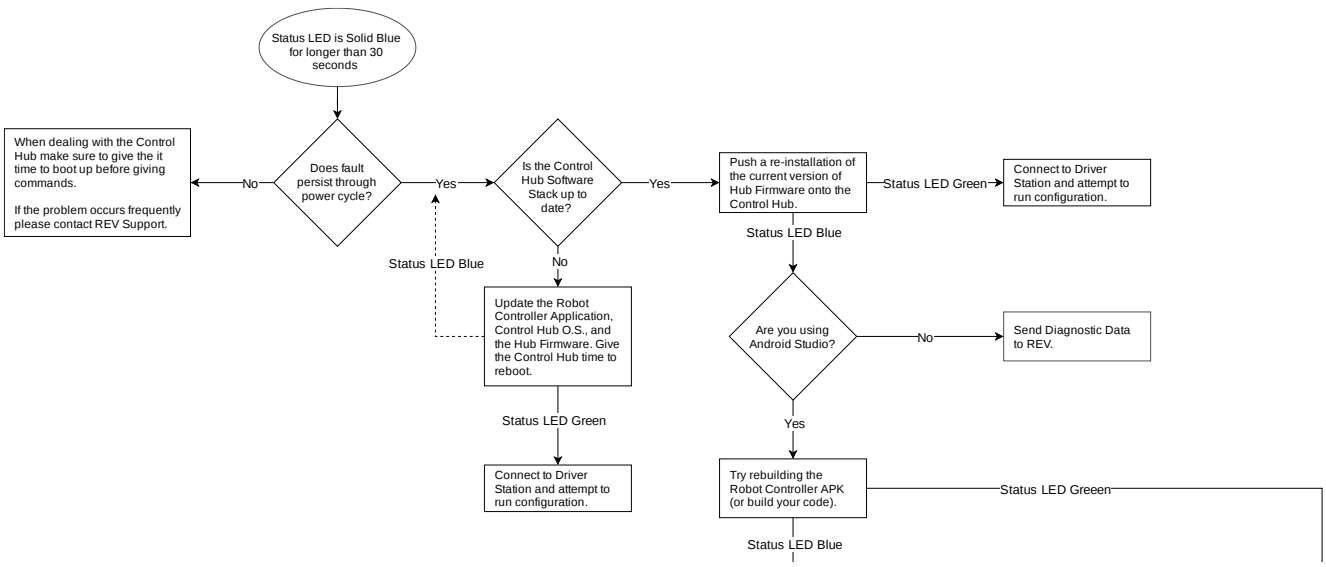
> Just remember correlation does not equal causation of an event but is useful data to further troubleshooting

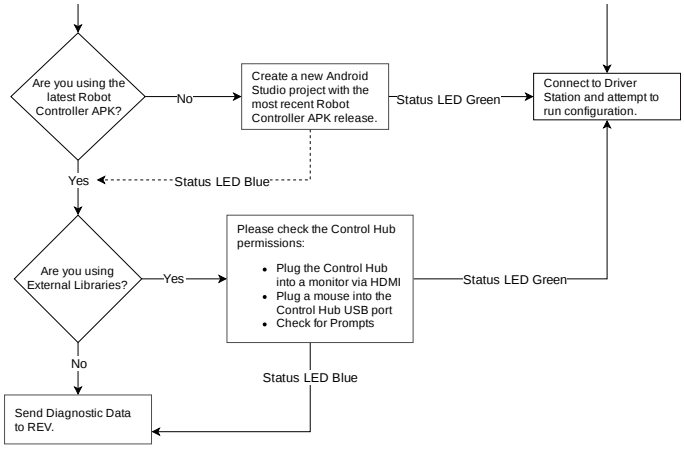
Contact REV Support with details of the troubleshooting information you have collected such as the answers to the questions above and the outcome of your troubleshooting thus far. It will also help to send logs or other diagnostic data to REV Support.

> Need help getting the Log Files to send to REV Support? See Downloading Log File for more information.

Status LED is Solid Blue for Longer than 30 Seconds

> This section is for troubleshooting a Control Hub. If you have an Expansion Hub please refer to the Expansion Hub Troubleshooting guide for help solving Expansion Hub related issues.





The status LED on the Control Hub is similar to a check engine light on a car. A solid blue status LED indicates the Robot Controller is not communicating to the I/O of the Control Hub, but not what the root cause is. Updating the Control Hub to the latest version of all the software is a first step to resolving this issue, listed below are two ways to update.

Using the REV Hardware Client

The REV Hardware Client is software designed to make managing REV devices easier for the user. This Client automatically detects connected device(s), downloads the latest software for those device(s), and allows for seamless updating of the device(s). Using the REV Hardware Client allows you to perform any required updates that may be needed to recover your Control Hub. The Hardware Client can also be used to Send Diagnostic Data to REV.

> If you do not have a Windows 10 or higher PC, see Downloading Log File for more options on getting your diagnostic data to REV, and Updating Firmware, Updating Operating System, and Updating Robot Controller Application for steps to update the software.

Using Android Studio

> The Control Hub must run version 5.0 or higher of the Robot Controller Application. If using Android Studio, make sure you are using a 5.0 or higher project.

If you use Android Studio for coding you will need to update your Robot Controller application by creating a new Android Studio project with the most recent version of the Robot Controller APK. Information on this process can be found in FTC Wiki Android Studio Tutorial.

Still Need Assistance?

Contact REV Support with details of the troubleshooting information you have collected such as the answers to the questions above and the outcome of your troubleshooting thus far. It will also help to send logs or other diagnostic data to REV Support.

>

Need help getting the Log Files to send to REV Support? See Downloading Log File for more information.

Driver Hub Troubleshooting

In this troubleshooting guide we will use specific language to describe different ways of power cycling the Driver Hub.

Turn Off/Power Off - Long press (1-2 seconds) the power button so that a drop down menu appears, then tap "power off" on the screen

Hard Reboot - Hold power button for at least 10 seconds and do not touch anything on the screen. Once the green LED light turns off and the screen goes dark, release the power button, and the hard reboot is complete.

Most Common Issues

Updating the OS

Updating the Driver Hub Operating System

When Updating your Driver Hub to the newest operating system, version 1.2.0, please be sure to follow these steps:

- Install the update on a fully charged Driver Hub. If the update fails, please plug in your hub and try again after fully charging.
- Don't touch the screen when a loading bar is displayed on the Driver Hub during the update process. If you touch the screen you will be directed to a menu after installation completes. Do not touch the screen and hard reboot your Driver Hub.
- Once you have updated your hub, please verify that your device is showing the current version 1.2.0, in the REV Hardware Client.

Unexpected Shut Down

Driver Hub Intermittent Battery Power Loss

Some Driver Hubs have a slight amount of extra space inside the battery bay that may cause a loss of power or intermittent battery charging. We have two quick fix options we are suggesting as solutions. The first is to use a small piece of folded paper or a few layers of tape to provide a more secure connection between the contacts. The second is a piece of foam tape we can ship

to teams which will accomplish the same goal. Suggested installation steps are highlighted below:

Option 1: Tape Quick Fix

Tape (painters tape or masking tape) is placed on the thin edge above the battery on the side opposite the contacts

Any tape or paper needs to sit inside the battery bay door edge

Option 2: Foam Tape

1. Cut foam tape into small pieces, approximately 2 inches or less long. The foam tape recommended is approximately 1/4 inch or less wide and 1/16 inch or less thick

2. Foam tape will be applied inside the battery case, opposite battery contacts and below the ridge that the battery door sits within.

3. Stick foam strip in the middle, both side to side and top to bottom, of the vertical surface opposite the battery contact switch.

4. Press foam strip down firmly to make sure it sticks.

5.1 Insert battery by inserting top of battery towards foam, and gently squeezing battery towards foam with thumb until battery can easily drop into battery case.

5.2 Continue to push the battery down until it is flush in the case.

6. Done

Common Charging/Power Issue Symptoms

The symptoms listed below can have a number of causes.

- Driver Hub only turns on when plugged into a charger
- Battery is discharging rapidly
- Battery reports low-battery at levels significantly above 0% and shuts off
- Device will not boot due to low battery even when Driver Hub is charged
- Driver Hub is on charger but will not turn on
- Device stopped charging and will not continue to charge

To properly troubleshoot, please start with the steps below

1. Check the orientation of the battery - see Battery Installation
2. Ensure you are using the charger that came with the Driver Hub - the charger must specifically be a non-PD charger for these troubleshooting steps, and using the charger that was shipped with the Driver Hub is the simplest way to confirm that.
3. Unplugging and replugging in the charger from the Driver Hub may resolve some symptoms
4. Ensure your Driver Hub is fully updated
5. Perform a Battery Recalibration
6. If possible, swap the battery with a known good battery to see if the issue follows the battery or follows the Driver Hub unit

Known Software Issues

The following are known issues that we are working to resolve via a future software update:

Waking Wi-Fi from a Sleep State

There is a known issue with the Wi-Fi driver not restarting correctly when the Driver Hub is woken from a "sleep" state. The current resolution is to perform a hard reboot on the device when the Driver Hub is having issues connecting to a Wi-Fi network.

You can make sure this issue doesn't happen before a match by leaving the screen on, and the Driver Station app open. This will prevent the Driver Hub from going to sleep.

Unlock Times are Inconsistent

Unlock can take anywhere from 2-10 seconds to occur, this is normal behavior.

Device Froze or Crashed while in Sleep Mode

Perform a hard reboot to wake up the device. This includes some cases where status LED B is solid green, indicating that the device is on, but the screen will not wake.

Inconsistent Battery Drain

Inconsistent battery draining while in a "sleep" state is a known issue. Devices may also shut off

while in a "sleep" state due to this. Future software updates are in the works to resolve this.

Additional Troubleshooting

"App Not Installed" Error

On the homepage the FTC Driver Station app can report an "app not installed" error after updating the OS and the app. This can also cause the Driver Hub to not allow you to open the FTC Driver Station app. To fix this do the following:

1. Remove the Driver Station app icon on the home page by clicking and dragging to the X icon
2. Drag the new icon from the app drawer on the home screen. The app drawer is accessed by swiping up on the home screen of the device.

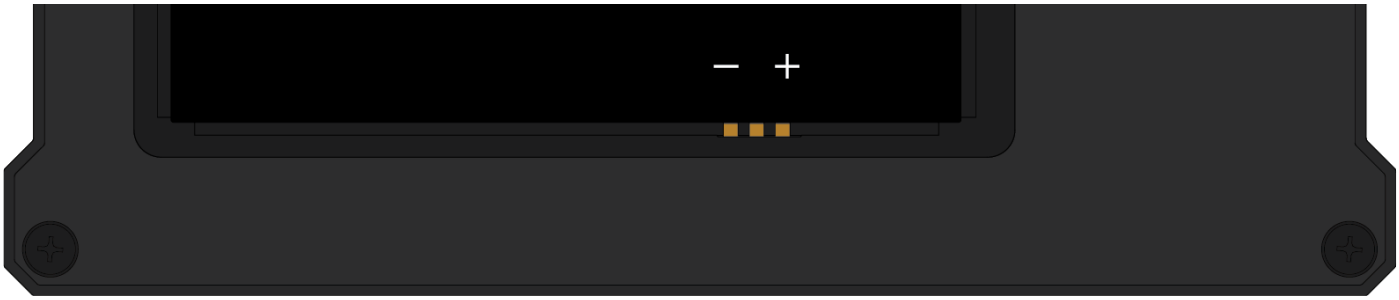
Android Permissions Lock Out

If the FTC Driver Station app is locked out due to android permissions, a factory reset of the Driver Hub should resolve this issue. Please power on the device, then follow the steps below to perform a factory reset:

1. Tap the "Setting" icon
2. Tap the "System" icon
3. Tap "reset options"
4. Tap "erase all data" (factory reset)

Battery Installation

To install the battery, place it with the REV Logo facing out and the -/+ located near the contacts for the device. Add on the rear door and screw in using the included M3 hardware.



A battery that is properly installed

Battery Calibration

We are aware of some Driver Hubs that were shipped from the factory without having their batteries properly calibrated. If you are experiencing power issues such as trouble charging or being unable to power on the device, try the following:

1. Plug Driver Hub into a charger without battery (Please use the charger that came with the Driver Hub to ensure a proper calibration)
2. Turn on Driver Hub and verify that the Driver Hub reports 100% battery charge. If the Driver Hub does not report 100% charge, you may be using a PD charger and not the one that came with the Driver Hub.
3. Install battery into Driver Hub while device is still on and charging
4. Charge for at least 8 hours and do not remove battery or charge cable
5. Remove Driver Hub from Charger
6. Hard Reboot

Battery Verification

After completing a battery calibration, use these steps to verify that your battery is functioning as expected.

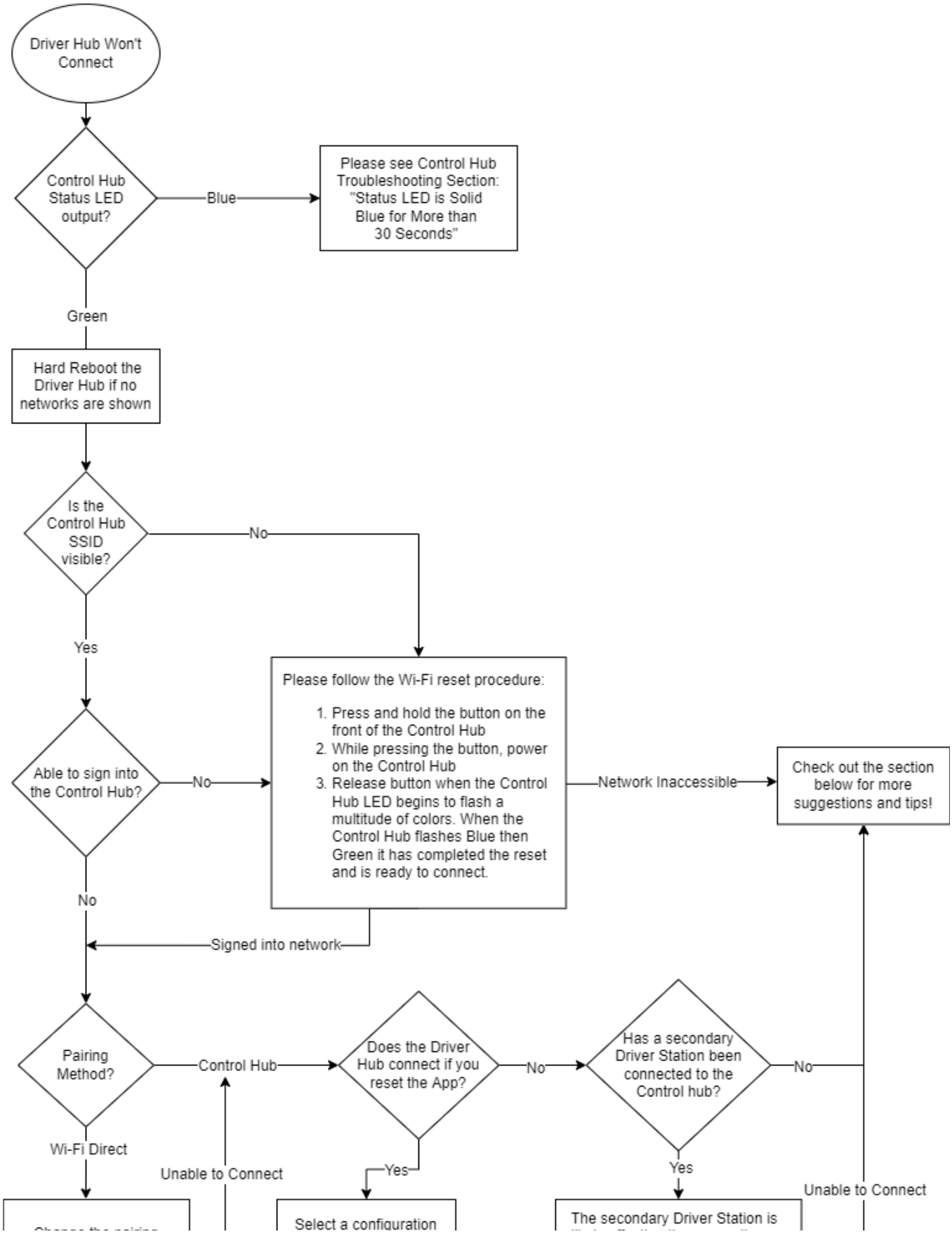
1. Place the battery in a Driver Hub and verify that the Driver Hub turns on.
2. Shake the Driver Hub with the screen still on and verify that the battery does not lose physical contact with the Driver Hub's contacts. If power drops, please see instructions for Unexpected Shutdown above.
3. Take note of the indicated battery charge level, charge the Driver Hub for 10 minutes, and verify that the battery charge level increased.
4. If you have the time, perform a full charge/discharge cycle with the battery to verify that the battery behaves normally.

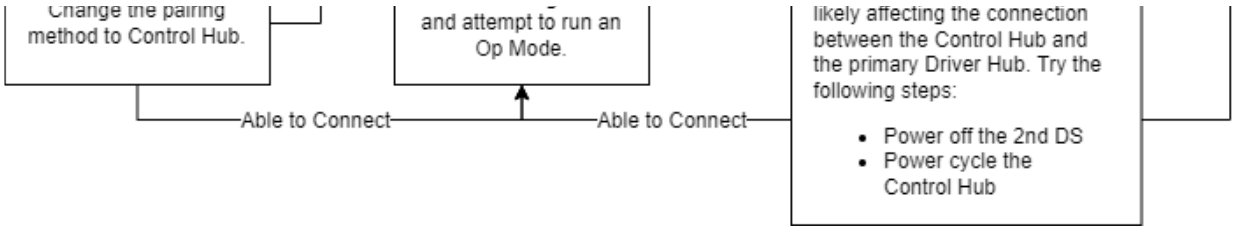
Digitizer Lines

Due to variances in the manufacturing process related to screen digitizer installation, some Driver Hubs have minor visible digitizer lines on the screens when the device is powered off. These lines are more prevalent in some units than others, but the presence or absence of digitizer lines does not impact the performance of the touch screen or unit in any way. Please contact us at support@revrobotics.com if you have any concerns about your specific unit.

Connecting to Control Hub

⚠ Information in this flowchart is for the initial bring up of connecting the Control Hub with a Driver Hub. For issues with intermittent connection or periodic connection drops please check out the information below this flowchart.





> The Wi-Fi reset will down grade the Wi-Fi connection to 2.4GHz. If you have an android device with 5GHz you may want to switch the Wi-Fi band in order to run on 5GHz. *Check out the Updating Wi-Fi Settings Section to learn more about making this switch.*

External factors, such as local Wi-Fi environment, play a part in the ability to establish or maintain a connection between a Control Hub and a Driver Station device. Like all aspects of of troubleshooting its important to isolate an issue by asking questions and discovering the answers! As you work on troubleshooting consider the following questions:

- **Is your system operating on a 2.4 GHz band or 5GHz band?**
  - REV recommends, if you have a dual band Driver Station device, that you operate on the 5GHz Wi-Fi band. Check out the Updating Wi-Fi Settings section to learn more about making this switch.
- **What is your local Wi-Fi environment like?**
  - Local Wi-Fi environment effects the consistency of a connection to the Control Hub. Use a Wi-Fi analyzer to check the local environment for channels that are cluttered with Wi-Fi networks. Change the Control Hubs Wi-Fi channel to a channel with the least amount of overlap with other networks.
- **Are you in a school or a place of business?**
  - In addition to the amount of local networks in an environment its important to understand what those local networks are capable of. For instance, some school districts have security measures in place that block unauthorized Wi-Fi access points. Talk to your local Wi-Fi administrator to find out what you need to get the Control Hub as an approved network.
- **Does the the Driver Hub connect to the Control Hub until a mechanism is run?**
  - Certain mechanisms draw enough power from the Control Hub to put a strain on the battery. If you notice a drop in displayed voltage when you start a code, or when a particular mechanism is run, this may be indicative of a brown out condition. Other indicators include:
    - The Driver Hub throwing errors about power to the system
    - The Driver Hub making a disconnect sound
    - The voltage on the Driver Hub showing 9 volts or lower when running code
    - Motors running at lower speeds then what they have been set to run
  - To remedy this issue check out our instructions on proper battery care.

> If the Control Hub SSID is not shown in the list of available Wi-Fi networks, try manually entering the Control Hub SSID on the Driver Hub to see if that allows you to connect.
>
> If no networks are shown at all, you should reboot the Driver Hub. See Most Common Issues section.

If you are still experiencing connection issues, once you have gone through the flowchart and worked on addressing the potential root of connection issues describe in the list above, start looking for patterns in the behavior. How often does this behavior appear? Are there certain things that happen around the same time the disconnects happen? The following list provides some ideas on what sort of patterns you might see:

- The Driver Hub connects to Wi-Fi and the Control Hub when a team member takes it home but doesn't connect consistently at school.
- The Driver Hub connects to the Control Hub until you start driving the robot around.

> Correlation does not equal causation of an event but is useful to take note of for further troubleshooting

Foam Tape Installation

1. Cut foam tape into small pieces, approximately 2 inches or less long. The foam tape recommended is approximately 1/4 inch or less wide and 1/16 inch or less thick

2. Foam tape will be applied inside the battery case, opposite battery contacts and below the ridge that the battery door sits within.

3. Stick foam strip in the middle, both side to side and top to bottom, of the vertical surface opposite the battery contact switch.

4. Press foam strip down firmly to make sure it sticks.

5.1 Insert battery by inserting top of battery towards foam, and gently squeezing battery towards foam with thumb until battery can easily drop into battery case.

5.2 Continue to push the battery down until it is flush in the case.

6. Done

Still Need Assistance?

Contact REV Support with details of the troubleshooting information you have collected such as the answers to the questions above and the outcome of your troubleshooting thus far. It will also help to send logs or other diagnostic data to REV Support.

If you encounter any of these issues below, please email support@revrobotics.com

- Device freezes on boot, then restarts the boot process in a loop
- Device freezes on boot and never gets into the OS, even after a hard reboot
- Charging and Power issues persist after multiple battery calibrations

> Need help getting the Log Files to send to REV Support? See Downloading Log File for more information.

Expansion Hub Troubleshooting

The following sections, "Common Indicators and their Solution Steps," provides common indicators of issues seen in the Expansion Hub. Think about what the potential indicators your Hub is currently exhibiting and consider the following questions:

- Did you perform a firmware update before the Hub began to have issues?
- What is the behavior of the Status LED on the Expansion Hub?
- Is the Driver Station showing an error message 'Cant find the Expansion Hub Portal"?
- Did the Robot Controller app open when you plugged in the RC phone and gave power to the Hub?
- Are you experiencing issues with communication between a primary and secondary Hub?

> *If a path in this guide does not resolve the issue please contact REV Robotics Support at support@revrobotics.com*

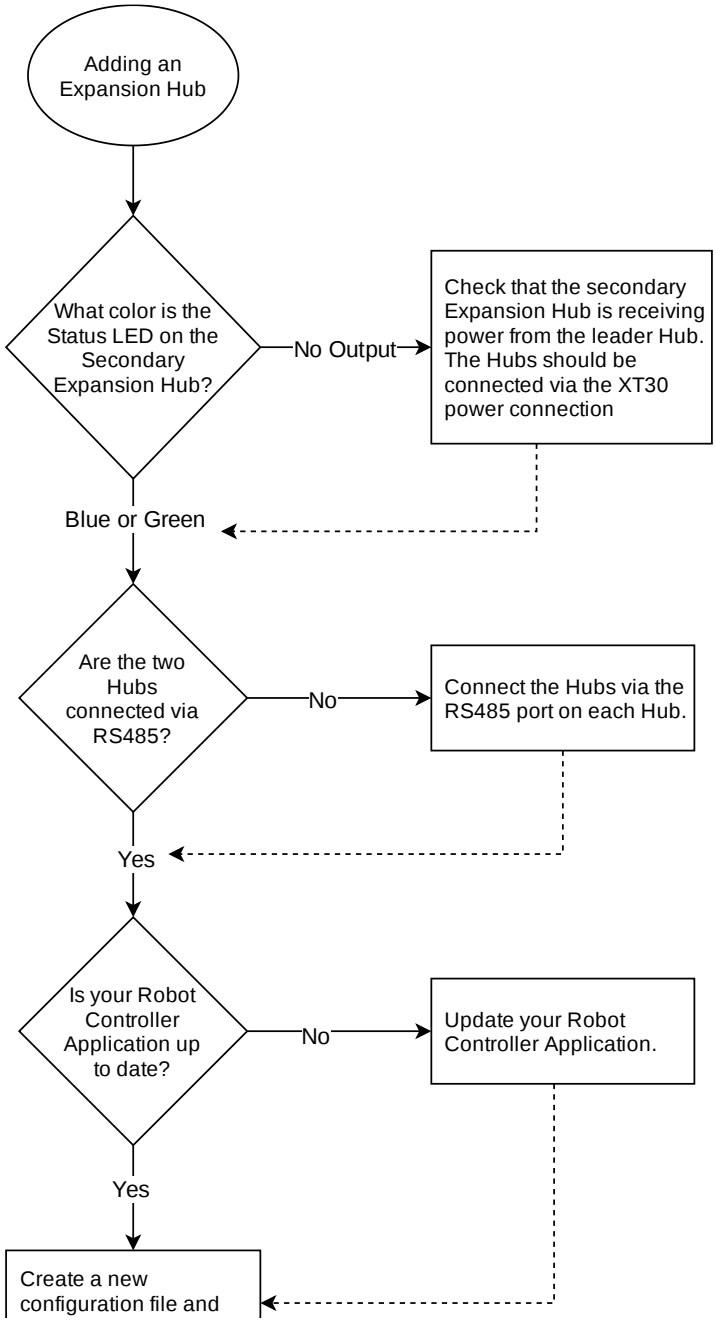
Common Indicators and their Solution Steps

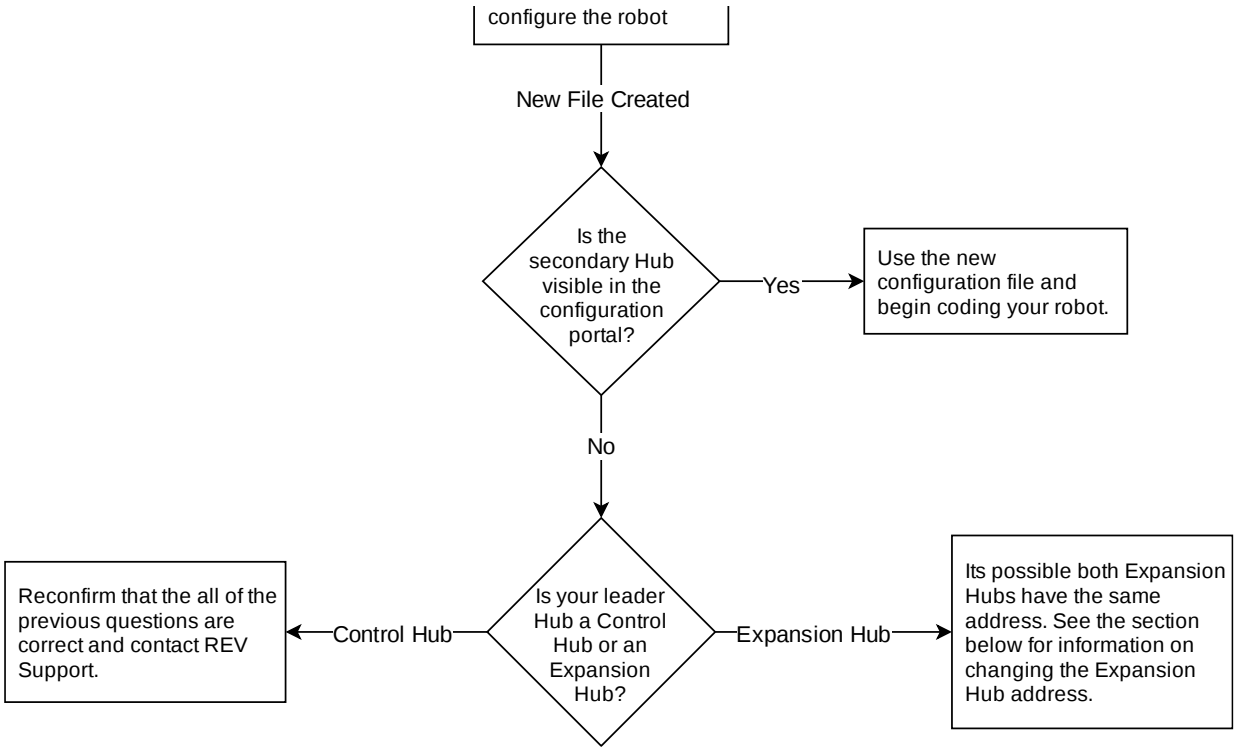
- The firmware update failed and the Hub is unresponsive
  - Try a Firmware Update
- The LED on the Expansion Hub is not lighting up
  - Try a Firmware Update
  - The LED is still not lighting up
- The Hub is not being recognized or communicating with the phones
  - Try doing the Hub Startup Procedure
- There are issues seeing a secondary Expansion Hub

> **Expansion Hubs purchased AFTER December 2021 no longer include an internal IMU**

Issues Seeing a Secondary Expansion Hub

The steps below utilize information provided in the Adding an Expansion Hub article. Use this article to help you navigate as you run through the troubleshooting flowchart.





> To update a Robot Controller check out the article on Updating the Robot Controller Application.

If you are attempting to connect two Expansion Hubs together please confirm that the first Expansion Hub is connected to the Robot Controller. From there change the Expansion Hub address. For information on how to change the Expansion Hub address check out the FTC Wiki Using a Second Expansion Hub article.

Firmware Update

Use the REV Hardware Client to update the Expansion Hub.

USB Serial Converter Check

1. Plug your Expansion Hub into a Windows PC
2. Open the Device Manager in Settings
3. Click the arrow next to Universal Serial Bus Controllers
4. Find USB Serial Converter under the menu
5. If this is not present there maybe a larger issue with your hub. Email support@revrobotics.com with details of the steps you have taken so far,and any order numbers for the Expansion Hub (if you have them)

> If you are using a Mac you can use System Information in Lion or later (or System Profiler in Snow Leopard and earlier versions of Mac OS) in Spotlight (press ⌘ and Space). The program is in /Applications/Utilities and is the tool to see the connected USB devices and other hardware details.

Hub Startup Procedures

1. Unplug the USB from your RC phone
2. Power off the main robot switch (turn off 12V power from the Expansion Hub(s))
3. Wait a few seconds
4. Turn on the Main Robot Switch (supply 12V power to the Expansion Hub(s))
5. On your RC phone, press the square button and the swipe to close the FTC RC app
6. Plug your RC phone into the USB-- the FTC app should automatically open
    1. If the app doesn't automatically open you do not have a good connection from the Expansion Hub to the Phone. Check your cables first, followed by the micro and mini USB connections.
    2. Consider using some form of strain relief (like the REV USB Retention Mount or one of the many 3d printable options available on places like Thingiverse) to keep the USB-mini port from being damaged.

> ⓘ If the issues persists after applying the Retention Mount try running through the Firmware Update procedure.

Still Need Assistance?

Contact REV Support with details of the troubleshooting information you have collected such as the answers to the questions above and the outcome of your troubleshooting thus far. It will also help to send logs or other diagnostic data to REV Support.

> ⓘ Need help getting the Log Files to send to REV Support? See Downloading Log File for more information.

Status LED Blink Codes

The RGB LED located on the Control Hub (REV-31-1595) and Expansion Hub (REV-31-1153) near the RS485 ports and on the bottom of the Driver Hub (REV-31-1956) provide user feedback regarding the status of the Hub. Below is a Table of the Blink Codes.

Control Hub

> ⓘ All Control Hub Blink Codes assume the latest Control Hub Operating System is running on the device

Robot Controller Application 6.0 or Higher

If a Control Hub is running Robot Controller Application 5.5 or lower the LED Blink Codes for the Hub will be the same as an Expansion Hub running Firmware Version 1.7.0 or higher.

LED Status	LED Description	When	Hub Status
	Solid Blue	At Boot	Control Hub has power; Battery is >7V and is waiting to initialize communications.
	Solid Blue	Anytime	Hub is waiting for communication with the Driver Station Hos Control Hub has power; Battery is >7V
	Solid Green	Anytime	Hub has power and active communicatior with the Android Platform.
	Blinking Blue	Anytime	Keep alive has timed out. Fault will clear when communication resumes.
	Blinking Orange	Anytime	Battery Voltage is lower than 7V. Either the 12V battery needs to be charged, or the Expansion Hub is running on USB powe only. This fault will clear when battery voltage is raised abov 7V. This will not be overwritten by the kee alive timeout pattern.
	Blinking Magenta	During Wi-Fi Reset	Control Hub changed Wi-Fi Band to 5GHz after pressing the button
			Control Hub changed

	Blinking Yellow	During Wi-Fi Reset	Wi-Fi Band to 2.4GHz after pressing the button

Driver Hub

> All Driver Hub Blink Codes assume the latest Driver Hub Software is running on the device

LED A

LED Status	LED Description	Hub Status
	Blinking White	Operating System is Booting
	Blinking Green	General Activity

LED B

LED Status	LED Description	Hub Status
	Solid Green	Device is on

Battery Status LED

LED Status	LED Description	Hub Status
	Blinking Red	Battery Charging
	Solid Red	Battery Charged

Expansion Hub

Firmware Version 1.7.0 or Higher

LED Status	LED Description	When	Hub Status

	Solid Blue	At Boot	Hub has power; Battery is >7V and is waiting to initialize communications.
	Solid Blue	Anytime	Hub is waiting for communication with the Robot Controller. Hub has power; Battery is >7V.
	Solid Green with one or more blue blinks every ~5 Seconds	Anytime	Hub has power and active communicatior with the Android Platform. The numbe of blue blinks is the same as the Expansion Hub's address. The factory default address is 2 ( ).
	Blinking Blue	Anytime	Keep alive has timed out. Fault will clear when communication resumes.
	Blinking Orange	Anytime	Battery Voltage is lower than 7V. Either the 12V battery needs to be charged, or the Expansion Hub is running on USB pow only. This fault will clear when battery voltage is raised abo 7V. This will not be overwritten by the kee alive timeout pattern.

System Overview

Control Hub Specifications

The REV Robotics Control Hub (REV-31-1595) is an affordable all in one educational robotics controller that provides the interfaces required for building robots, as well as other mechatronics, with multiple programming language options. The Control Hub was designed and built as an easy to use, dependable, and durable device for use in classroom and the competition. It features an Android operating system, built-in dual band Wi-Fi (802.11 ac/b/g/n/w), and a mature software package designed for both basic and advanced use cases. When the Control Hub software is updated with new features, the controller can receive a "field upgrade," through an update process that is fast and simple.

The Control Hub is an approved device for use in FIRST® Global and FIRST Tech Challenge.

Port Label	Qty	Connector	Description
Battery	2	XT-30	Connect one 12V NiMh battery, add an Expansion Hub with second port
Motor	4	JST VH, 2-pin	Motor power output
Encoder	4	JST PH, 4-pin	Quadrature encoder input
Servo	6	0.1” Header	Extended range 5V servo output
+5V Power	2	0.1” Header	Power for auxiliary device(s)
Analog	4	JST PH, 4-pin	Analog input with two channels per connector.
Digital	8	JST PH, 4-pin	Digital Input/Output with two channels pe connector
			Four separate I2C

I2C	4	JST PH, 4-pin	busses
RS485	2	JST PH, 3-pin	Use this serial communication port t add an Expansion Hu
UART	2	JST PH, 3-pin	Debugging only
USB C	1	USB C	Connect directly to th Control Hub via PC, USB 2.0
USB 2.0	1	USB A	Connect USB camera and other USB peripherals to the Control Hub
USB 3.0	1	USB A	Connect USB camera and other USB peripherals to the Control Hub
HDMI	1	HDMI A	Supports 4k @ 60Hz

Specifications

The following tables provide the operating and mechanical specifications for the Control Hub.

General Specifications

Feature type	Description
Processor(s)	RK3328 Quad-core ARM® Cortex-A53 Texas Instruments ARM® Cortex®-M4
Memory	1GB LPDDR3
Storage†	8GB eMMC 4.51
Wireless	802.11 ac/b/g/n/w Wi-Fi; Dual Band 2.4 & 5 GHz Bluetooth 4.1
Graphics‡	GPU - ARM® Mali 450MP4 HDMI 2.0 support for 4k @ 60Hz

†	Supports expandable storage through the SD Car slot
‡	Display graphics supported through an external display over HDMI

> ⚠ DO NOT exceed the absolute maximum electrical specifications. Doing so will cause permanent damage to the Control Hub and will void the warranty.

Input Power Specifications

Parameter	Min	Typ	Max	Units
Operating voltage range (V_{IN})	8	12	15	V
Absolute maximum supply voltage	-	-	15	V

Motor Port Specifications

Parameter	Min	Typ	Max	Units
Continuous output current †	-	-	10	A
Absolute maximum output current ‡	-	-	20	A

†	Exceeding the continuous current maximum depends on many thermal factors. The outputs wi self protect once they approach their thermal limit.
‡	Maximum current is ultimately limited by the in-lin battery fuse.

Encoder Port Specifications

Parameter	Min	Typ	Max	Units

Encoder port input voltage	0	-	3.3	V
Encoder port supply voltage	-	-	3.3	V
Encoder port total supply current	-	-	500	mA

ⓘ See Sensors - Encoders for more information on encoders and using the encoder ports. For using non-REV motor encoders see Using 5V Sensors - Encoders for more details.

Digital Port Specifications

Parameter	Min	Typ	Max	Units
Digital port input voltage	0	-	3.3	V
Digital port supply voltage	-	-	3.3	V
Digital port total supply current	-	-	1	A

ⓘ See Sensors - Digital for more information on using the digital ports. See Using 5V Sensors for information on using 5V logic level devices with the digital ports.

Analog Port Specifications

Parameter	Min	Typ	Max	Units
Analog port input voltage range †	0	-	5	V
Analog port supply voltage	-	-	3.3	V
Analog port total supply current	-	-	500	mA

	The analog input will accept up to 5V. When using

†	5V analog sensors, a custom wiring harness is needed to provide 5V of power for the sensor as the power pin provides 3.3V.

ⓘ See Sensors - Analog for more information on using the analog ports.

I2C Port Specifications

Parameter	Min	Typ	Max	Units
I2C port input voltage range	0	-	3.3	V
I2C port supply voltage	-	-	3.3	V
I2C port total supply current	-	-	500	mA
Bus speed	-	100/400	-	kHz

ⓘ See Sensors - I2C for more information on using the I2C ports. See Using 5V Sensors for information on using 5V logic level devices with the I2C ports.

Servo Port Specifications

Parameter	Min	Typ	Max	Units
Servo output signal voltage	0	-	5	V
Servo port supply voltage	-	5	-	V
Servo port pair total supply current †	-	-	2	A
Absolute maximum total supply current ‡	-	-	5	A
Servo port output pulse range	500	-	2500	µs

†	Total supply is shared across pairs of ports (0-1, 2 3, 4-5)
‡	The 5A total supply current for all servo ports and +5V power ports is shared.

+5V Power Port Specifications

Parameter	Min	Typ	Max	Units
+5V power port output voltage	-	5	-	V
+5V power port pair total supply current †	-	-	2	A
Absolute maximum total supply current ‡	-	-	5	A

†	Total supply current is shared across both ports
‡	The 5A total supply current for all servo ports and +5V power ports is shared.

Mechanical Specifications

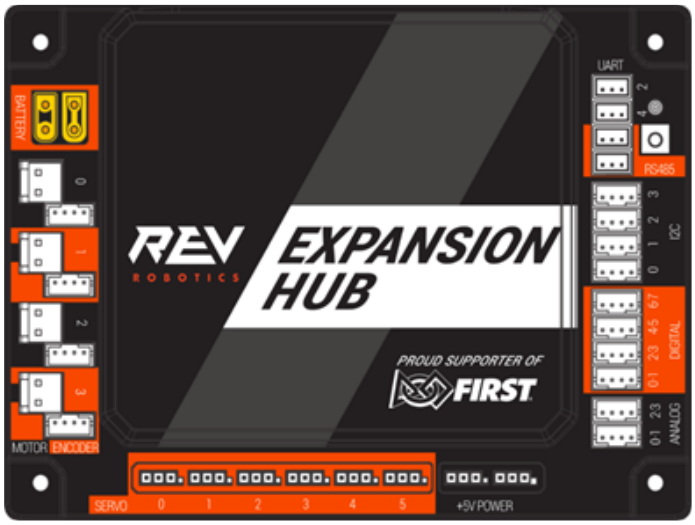
Parameter	Min	Typ	Max	Units
Body length	-	103	-	mm
Body width	-	143	-	mm
Body height	-	29.5	-	mm
Weight	-	209	-	g
Mounting hole pitch	-	16	-	mm

Expansion Hub Specifications

The REV Robotics Expansion Hub (REV-31-1153) is a low-cost educational device that can communicate with any computer (commonly the REV Robotics Control Hub or an Android Phone) to provide the interfaces required for building robots and other mechatronics. The Expansion Hub was purpose built to stand up to the rigors of the classroom and competition field. It features a mature firmware designed for basic and advanced use cases with the ability to be field upgraded in the future.

The IO ports of the Expansion Hub are identical in specification to the Control Hub. Within this documentation, many sections may refer to the Control Hub, but the connections are the same for the Expansion Hub.

The REV Robotics Expansion Hub is an approved device for use in the FIRST Tech Challenge and FIRST Global.



Port Label	Qty	Connector	Description
Battery	2	XT30	Connect one 12V NiMh battery, add an Expansion Hub with second port
Motor	4	JST VH, 2-pin	Motor power output
Encoder	4	JST PH, 4-pin	Quadrature encoder input
Servo	6	0.1” Header	Extended range 5V servo output (500-2500ms)
5V Aux Power	2	0.1” Header	Auxiliary device 5V/2
			Analog input 0-5.0V measurement range with two channels pe

Analog	4	JST PH, 4-pin	connector. 3.3V provided on the connector power pin.
Digital	8	JST PH, 4-pin	Digital Input/Output with two channels pe connector
I2C	4	JST PH, 4-pin	Four separate I2C busses, 100kHz/400kHz bus speed
RS485	2	JST PH, 3-pin	Serial communicatior port to add a Hub (Control or Expansior
UART	2	JST PH, 3-pin	Debugging only
MINI USB	1	USB Mini-B	Connect directly to th Robot Controller Android device or PC

Specifications

The following tables provide the operating and mechanical specifications for the Expansion Hub.

> ⚠ DO NOT exceed the absolute maximum electrical specifications. Doing so will cause permanent damage to the Expansion Hub and will void the warranty.

Input Power Specifications

Parameter	Min	Typ	Max	Units
Operating voltage range (V_{IN})	8	12	15	V
Absolute maximum supply voltage	-	-	15	V

Motor Port Specifications

Parameter	Min	Typ	Max	Units
Continuous output current †	-	-	10	A
Absolute maximum output current ‡	-	-	20	A

†	Exceeding the continuous current maximum depends on many thermal factors. The outputs wi self protect once they approach their thermal limit.
‡	Maximum current is ultimately limited by the in-lin battery fuse.

Encoder Port Specifications

Parameter	Min	Typ	Max	Units
Encoder port input voltage	0	-	3.3	V
Encoder port supply voltage	-	-	3.3	V
Encoder port total supply current	-	-	500	mA

> See Sensors - Encoders for more information on encoders and using the encoder ports. For using non-REV motor encoders see Using 3rd Party Sensors - Encoders for more details.

Digital Port Specifications

Parameter	Min	Typ	Max	Units
Digital port input voltage	0	-	3.3	V
Digital port supply voltage	-	-	3.3	V
Digital port total supply current	-	-	1	A

ⓘ See Sensors - Digital for more information on using the digital ports. See Using 5V Sensors for information on using 5V logic level devices with the digital ports.

Analog Port Specifications

Parameter	Min	Typ	Max	Units
Analog port input voltage range †	0	-	5	V
Analog port supply voltage	-	-	3.3	V
Analog port total supply current	-	-	500	mA

†	The analog input will accept up to 5V. When using 5V analog sensors, a custom wiring harness is needed to provide 5V of power for the sensor as the power pin provides 3.3V.

ⓘ See Sensors - Analog for more information on using the analog ports.

I2C Port Specifications

Parameter	Min	Typ	Max	Units
I2C port input voltage range	0	-	3.3	V
I2C port supply voltage	-	-	3.3	V
I2C port total supply current	-	-	500	mA
Bus speed	-	100/400	-	kHz
I2C pull-up resistor	-	2.49	-	kΩ

Expansion Hubs purchased AFTER December 2021 no longer include an internal IMU

See Sensors - I2C for more information on using the I2C ports. See Using 5V Sensors for information on using 5V logic level devices with the I2C ports.

Servo Port Specifications

Parameter	Min	Typ	Max	Units
Servo output signal voltage	0	-	5	V
Servo port supply voltage	-	5	-	V
Servo port pair total supply current †	-	-	2	A
Absolute maximum total supply current ‡	-	-	5	A
Servo port output pulse range	500	-	2500	µs

†	Total supply is shared across pairs of ports (0-1, 2 3, 4-5)
‡	The 5A total supply current for all servo ports and +5V power ports is shared.

+5V Power Port Specifications

Parameter	Min	Typ	Max	Units
+5V power port output voltage	-	5	-	V
+5V power port pair total supply current †	-	-	2	A

Absolute maximum total supply current ‡	-	-	5	A

†	Total supply current is shared across both ports
‡	The 5A total supply current for all servo ports and +5V power ports is shared.

Mechanical Specifications

Parameter	Min	Typ	Max	Units
Body length	-	103	-	mm
Body width	-	143	-	mm
Body height	-	29.5	-	mm
Weight	-	209	-	g
Mounting hole pitch	-	16	-	mm

Driver Hub Specifications

The REV Robotics Driver Hub (REV-31-1596) is a compact mobile computing device designed for interfacing with the Control Hub (REV-31-1595). The Driver Hub was designed and built as an easy to use, dependable, and durable device for use in classroom and the competition. It features an Android operating system, built-in dual band Wi-Fi (802.11 ac/b/g/n/w), and support for many off-the-shelf gamepads and HID devices connected through built-in USB ports. When the Driver Hub software is updated with new features, the device can receive a "field upgrade," through a fast and simple update through the REV Hardware Client.

The Driver Hub is an approved device for use in FIRST® Global and FIRST Tech Challenge.

Label	Qty	Interface	Description
Power	1	Button	Turns the device on and off
USB C	1	USB C	Connect directly to th Driver Hub via PC, USB 2.0 Supports fast chargin the Driver Hub over USB PD
USB 2.0	3	USB A	Connect USB controllers and other HID devices to the Driver Hub
Ethernet	1	RJ45	10/100 base-T Supports 12V DC passive POE

Specifications

The following tables provide the mechanical specifications for the Driver Hub.

General Specifications

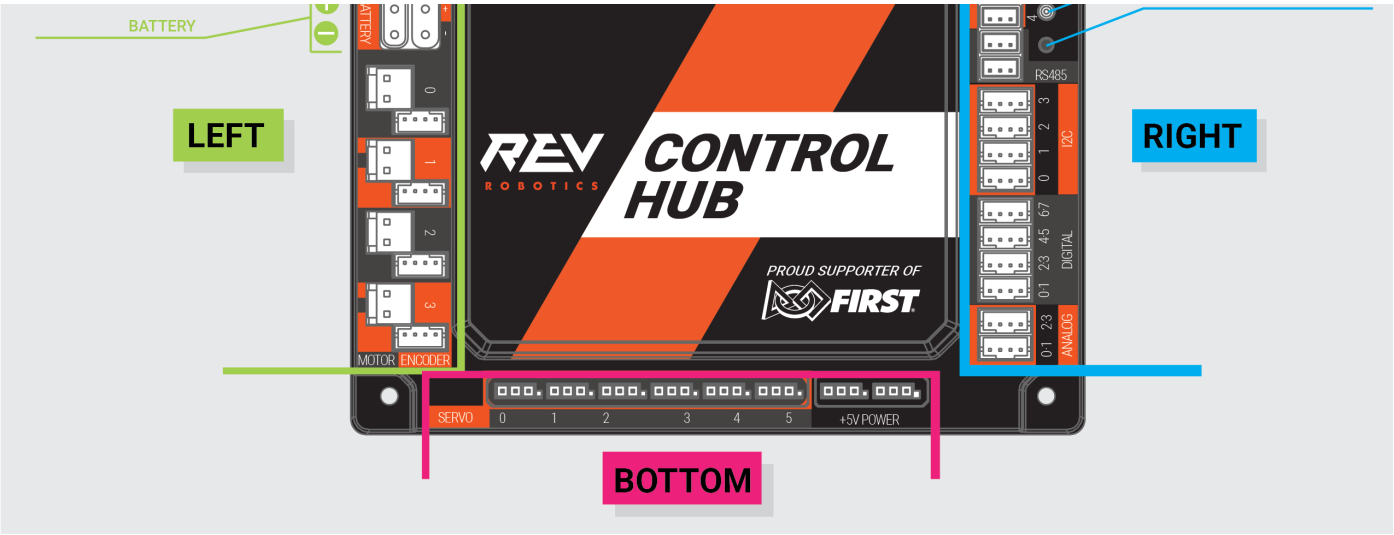
General Specifications

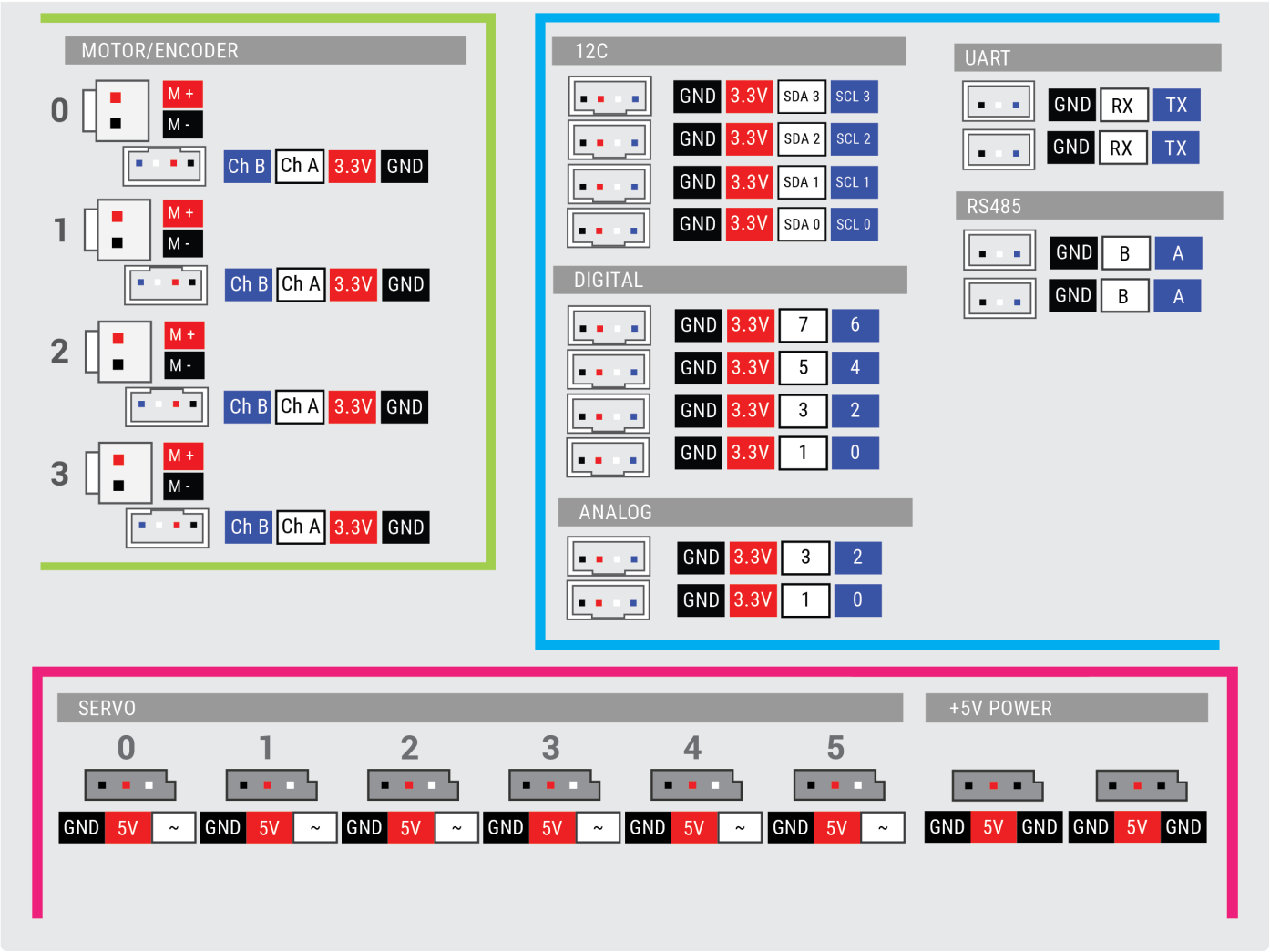
Feature type	Description
Processor	RKPX30 Quad-core ARM A35
Memory	1GB LPDDR3
Storage†	8GB eMMC 4.51
Wireless	802.11 ac/b/g/n/w Wi-Fi; Dual Band 2.4 & 5 GHz Bluetooth 4.1
Graphics	ARM® Mali 450MP4
†	Supports expandable storage through the SD Car slot

Mechanical Specifications

Parameter	Min	Typ	Max	Units
Body length	-	3.375	-	in
Body width	-	5.25	-	in
Body height	-	1.0	-	in
Weight	-	9.8	-	oz
Mounting hole pitch	-	16	-	mm
Screen size (diagonal)		5		in
Screen resolution		800 x 600		px

Port Pinouts





Protection Features

The Control (REV-31-1595) and Expansion Hub (REV-31-1153) were designed with a number of protection features built into the device. These include the following:

- Reverse battery input protection
- Electrostatic discharge (ESD) protection on all connections

- Over-current protection
  on all power buses
  - Digital I/O bus
  - I2C bus
  - Analog bus
  - USB
  - Servo bus per pair (0-1, 2-3, 4-5)
  - Encoder bus
- Over-current monitoring for individual Motor Channels
- Keyed and locking connectors
- Fail-safe mode at communication loss

Cables and Connectors

The REV Robotics Control Hub (REV-31-1595) connector selection provides a robust high-density solution for the user. All connectors are keyed and locking except for the Servo, 5V auxiliary power, HDMI , and USB ports.

XT-30 - Power Cable

JST VH - Motor Power

JST PH - Sensors and RS485

XT-30 - Power Cable

The XT30 connector is used for connecting a battery and powering a Control/Expansion Hub. Each Control/Expansion Hub has both a Male and Female XT30 connector, as determined from the metal contacts, not the plastic housing. While either connector can provide power to the hub, it is the convention to use the male connector for "power in" to the hub, and to use the female connector for "power out" to a connected secondary device, like an Expansion Hub or XT30 Power Distribution Block, from the single

Most teams will want to use pre-made cables which can be conveniently sourced from the REV Robotics website. However, teams can also make their own cables. These connectors are solder-cup style, do not require any crimping tools, and are available from various online vendors. Because these connectors are an open design, they are manufactured by a variety of sources and quality may vary. AMASS branded connectors are recommended, and are what is used on REV products, but there are many other quality vendors available.

Table 1: Premade XT-30 Cables and Accessories

Cable Type	Length	REV Robotics Part Number
XT-30 Male - XT-30 Female	30 cm	REV-31-1392
XT-30 Male - XT-30 Female	50 cm	REV-31-1394
XT-30 Female - Tamiya	8 cm	REV-31-1382
XT-30 Female - Anderson Power Pole Style	8 cm	REV-31-1385
Power Switch Cable (XT30 Male – XT30 Female)	12 cm	REV-31-1387
XT30 Connector Pack – 5 Pairs	-	REV-31-1399

JST VH - Motor Power

Motor Power connections on the Control Hub (REV-31-1595) use the JST VH style connector. This connector is keyed and locking with a small latch, seen below, which must be depressed to release the cable.

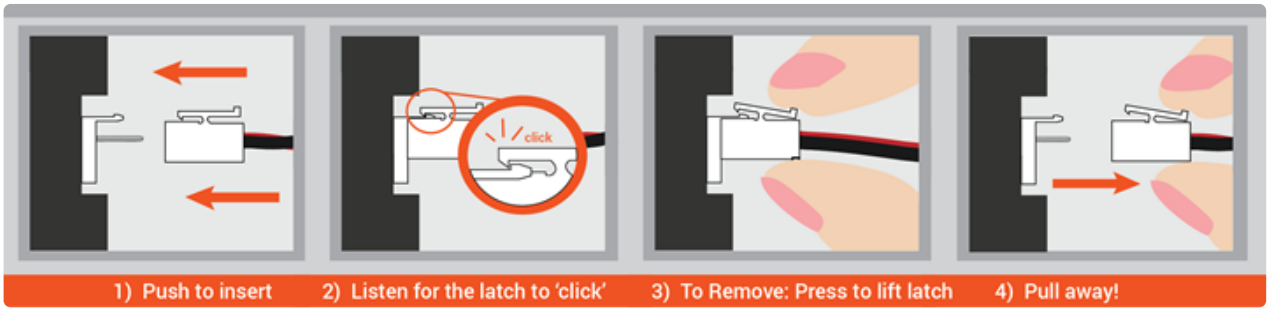


Figure 1: How to Use a JST VH Cable

REV Robotics recommends, in most cases, that teams use pre-made cables because crimp quality is better when made using industrial tooling. These cables can be purchased directly from the REV Robotics website or through other online vendors.

Premade JST VH Cables and Accessories

Cable/Accessory	Pins	Length	REV Robotics Part Number
JST VH 2-Pin Motor Cable	2 pins	30 cm	REV-31-1412
JST VH 2-Pin Motor Cable	2 pins	50 cm	REV-31-1413
JST VH 2-Pin Motor Cable	2 pins	100 cm	REV-31-1526
Anderson to JST VH Cable	2 pins	12 cm	REV-31-1381
JST VH 2-pin Joiner Board	2 pins	-	REV-31-1429

For teams that want to try crimping their own cables, or to find more information about the connectors, Table 3 lists the appropriate part numbers.

Connector Specifications

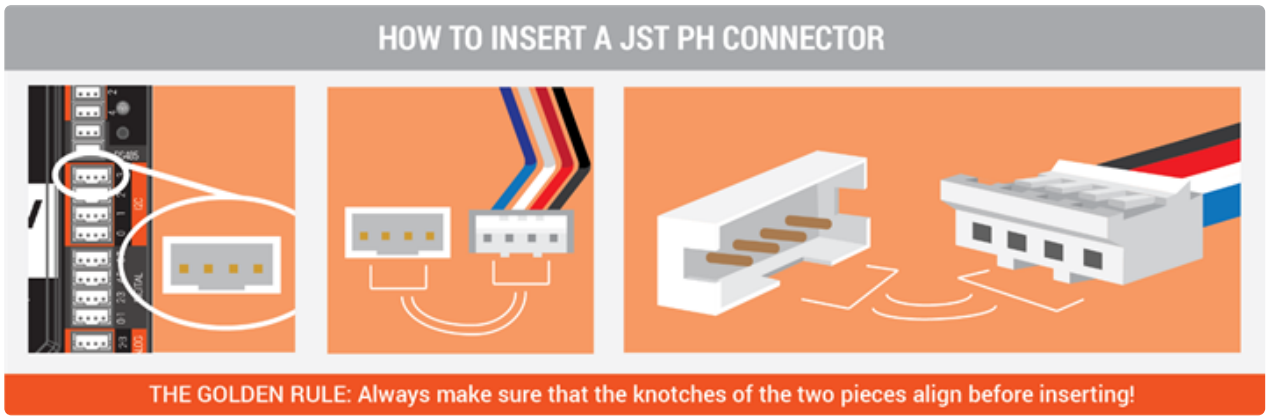
- 10A Continuous Current (16AWG)
- 3.96mm Pitch
- Accepts 22-16AWG Wire

JST VH Connector Part Number Reference

	Manufacturer Part Number	DigiKey Part Number
Contact, JST VH, 18-22AWG	SVH-21T-P1.1	455-1133-1-ND
Contact, JST VH, 16-20AWG	SVH-41T-P1.1	455-1319-1-ND
Housing, JST VH, 2-pin	VHR-2N	455-1183-ND
Header, JST VH, 2-pin, Top Entry	B2P-VH	455-1639-ND
Header, JST VH, 2-pin, Side Entry	B2PS-VH	455-1648-ND

JST PH - Sensors and RS485

The JST PH style connector is used for motor encoder, analog, digital, I2C, RS485, and UART connections on the Control Hub and Expansion Hub. These are all 4-pin connections except for the RS485 and UART which are 3 pin. The connectors are keyed (they only insert in one orientation) and are friction locking. Below the keying feature aligned with the cable is shown.



REV Robotics recommends in most cases that teams use pre-made cables because the quality of the crimp is better when made using industrial tooling. These cables can be bought directly from the REV Robotics Website or through other online vendors.

Premade 4-pin JST PH Cables

Cable/Accessory	Pins	Length	REV Robotics Part Number
JST PH 4-Pin Sensor Cable	4	30 cm	REV-31-1407
JST PH 4-Pin Sensor Cable	4	50 cm	REV-31-1408
JST PH 4-Pin Sensor Cable	4	100 cm	REV-31-1409
JST PH 4-pin Joiner Board	4		REV-31-1388

Premade 3-pin JST PH Cables

Cable	Pins	Length	REV Robotics Part Number
JST PH 3-pin Communication Cable	3	30 cm	REV-31-1417
JST PH 3-pin Communication Cable	3	50 cm	REV-31-1418

JST PH 3-pin Communication Cable	3	100 cm	REV-31-1565

For teams that want to try crimping their own cables, or to find more information about the connectors, the table below lists the appropriate part numbers.

Connector Specifications

- 2A continuous current (24AWG)
- 2.0mm pitch
- Accepts 32-24AWG wire

JST PH Connector Part Number Reference

Connector Parts	**Manufacturer Part Number**	Vendor	**Part Number**
Contact, JST PH, 30-24AWG	SPH-002T-P0.5S	DigiKey	455-1127-1-ND
Contact, JST PH, 28-24AWG	SPH-002T-P0.5L	DigiKey	455-2148-1-ND
Housing, JST PH, 4-pin	PHR-4	DigiKey	455-1164-ND
Header, JST PH, 4-pin, Top Entry	B4B-PH-K-S	DigiKey	455-1706-ND
Header, JST PH, 4-pin, Side Entry	S4B-PH-K-S	DigiKey	455-1721-ND
Housing, JST PH, 3-pin	PHR-3	DigiKey	455-1126-ND
Header, JST PH, 3-pin, Top Entry	B3B-PH-K-S	DigiKey	455-1705-ND
Header, JST PH, 3-pin, Side Entry	S3B-PH-K-S	DigiKey	455-1720-ND
Recommended Crimping Tool	IWISS SN-2549	Amazon	SN-2549

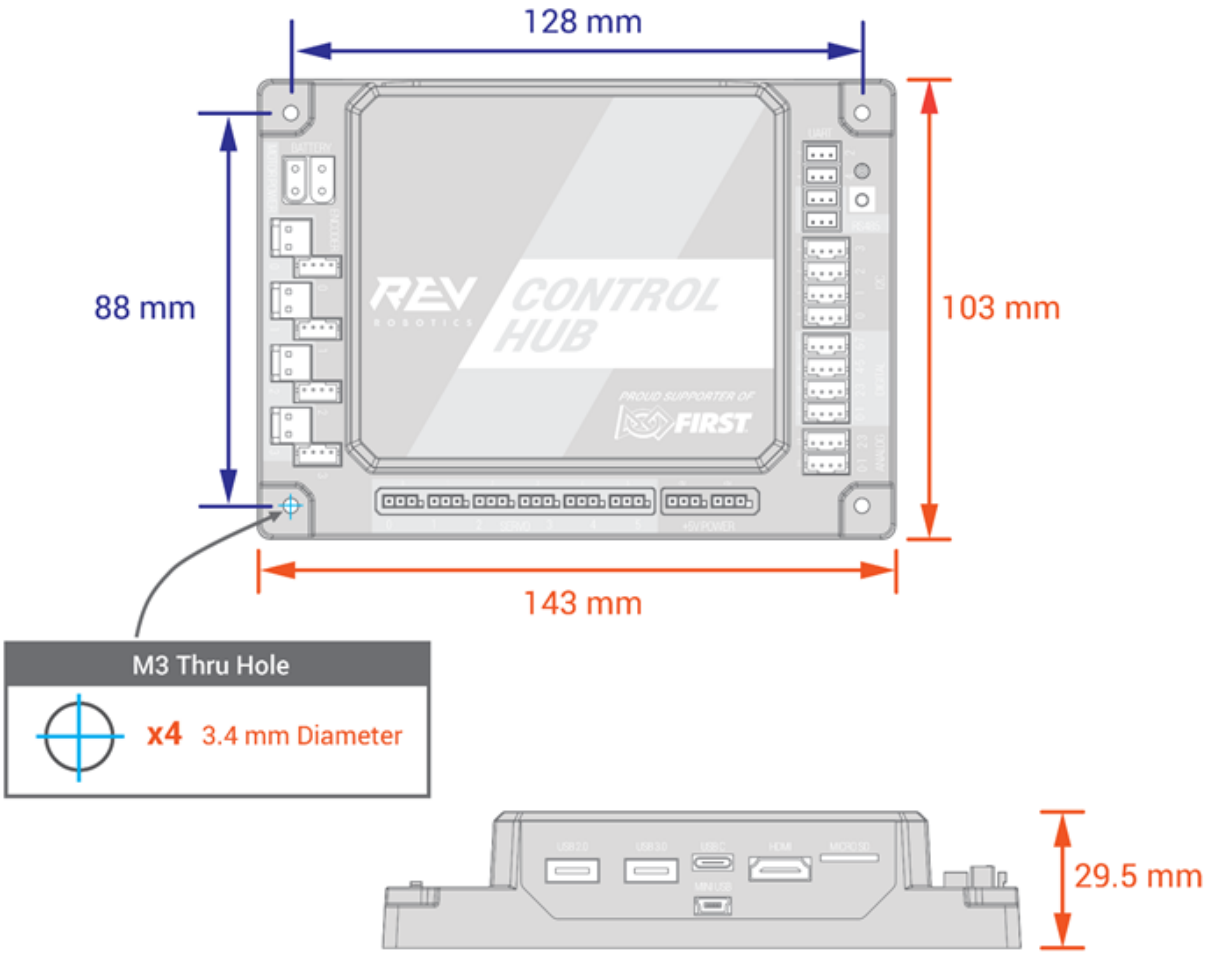
Integrated Sensors

The REV Robotics Control Hub (REV-31-1595) and Expansion Hub (REV-31-1153) integrate a number of

feedback sensors. Some of these are user accessible in the latest FTC Android Studio SDK, but others are not yet directly user accessible. These sensors are in some cases also used by the Control Hub and Expansion Hub for internal safety monitoring.

- Battery Voltage Monitoring [**Accessible**]
- Integrated 9-axis IMU [**Accessible**]
  - Bosch BNO055 9-axis absolute orientation sensor
  - Internally connected to I2C port 0 and configured to address 0x28
- Current Monitoring
  - Battery [**Accessible**]
  - I2C Bus [**Accessible**]
  - Digital Power Bus [**Accessible**]
  - Servo Power Bus [Not Accessible]
- Per Motor Channel Current Monitoring [**Accessible**]

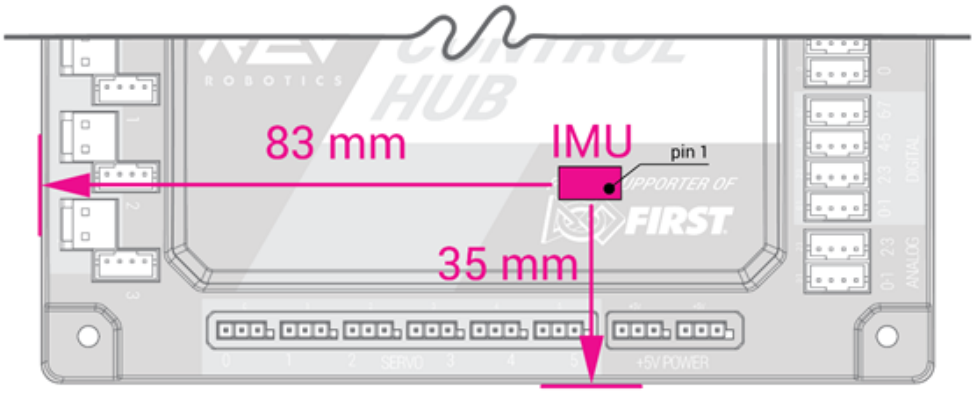
Dimensions and Important Component Locations



IMU Location

When using the Control Hub (REV-31-1595) or Expansion Hub (REV-31-1153) please note the location of the IMU in the graphic below. The Hub's orientation may impact the values received from the embedded IMU.

BELOW: IMU Details Shown in Enlarged View

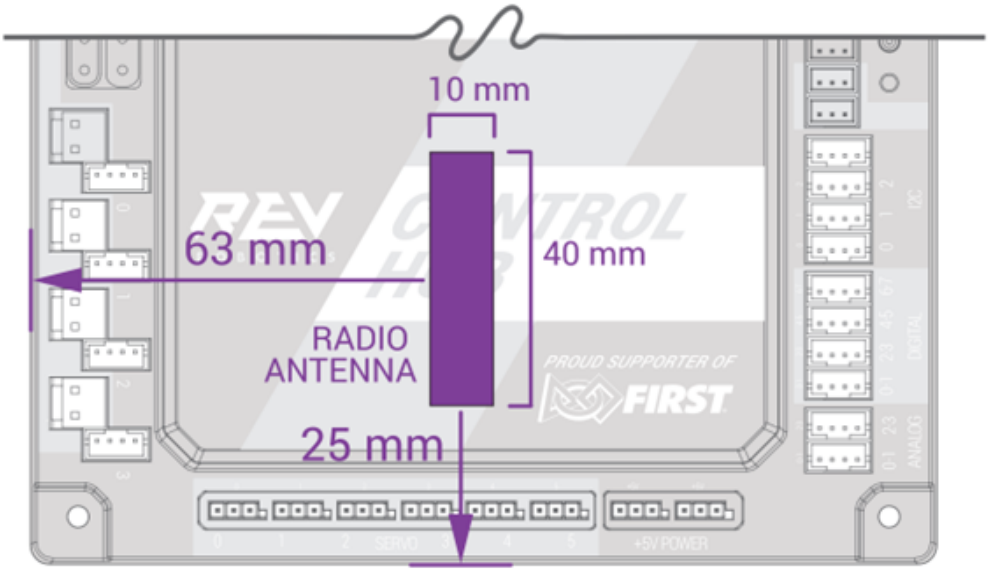


Wi-Fi Radio Location

The Control Hub has an embedded Wi-Fi radio for wireless communication. The antenna is located towards the top of the Control Hub itself. The graphic below shows the location of antenna.

> DO NOT put a battery or other Wi-Fi blocking object on top of the Control Hub. This can lead to higher ping times for communication between the Control Hub and the Driver Station.

BELOW: Radio Antenna Shown in Enlarged View



Updating and Managing

Managing Wi-Fi on the Control Hub

The Control Hub creates a Wi-Fi access point to connect a Driver Station device or laptop to the Control Hub for programming and operation. Settings for the Control Hub access point are managed through the Robot Controller Console or the User Button on the Control Hub.

Before making changes to the Control Hub's Wi-Fi network checking what Wi-Fi bands are supported by the devices being used is important to ensure they will work as expected. Below are the Android Devices that are officially supported:

Supported Android Devices and Wi-Fi Band Capabilities

Android Device	Wi-Fi Band
REV Driver Hub (REV-31-1596)	2.4 GHz & 5 GHz (Dual Band)
Moto G (2nd generation)	2.4 GHz (Single Band)
Moto G (3rd generation)	2.4 GHz (Single Band)
Moto G (4th generation)	2.4 GHz (Single Band)
Moto G5	2.4 GHz & 5 GHz (Dual Band)
Moto G5 Plus	2.4 GHz & 5 GHz (Dual Band)
Moto E4	2.4 GHz & 5 GHz (Dual Band)
Moto E5	2.4 GHz & 5 GHz (Dual Band)
Moto E5 Play	2.4 GHz & 5 GHz (Dual Band)

The following page is split into two sections. The first will cover how to access the Wi-Fi Settings through the Robot Controller Console. It is recommended to use the REV Hardware Client as it will allow the user to access the Wi-Fi settings over a wired connection. The second will run through the steps for using the Control Hub's User Button to preform a Wi-Fi reset or Wi-Fi band change.

> ⓘ If you run into any problems trying to use the Hardware Client or when resetting the Wi-Fi, please contact support@revrobotics.com

---

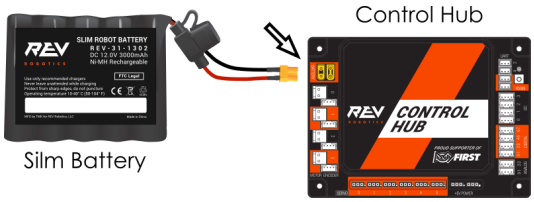
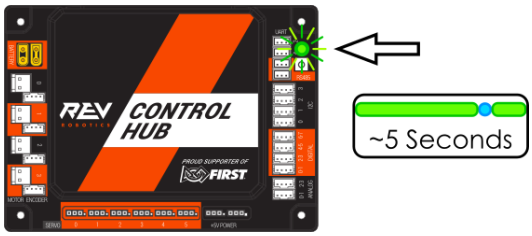
Using the Robot Controller Console

The Robot Controller Console gives access to the Wi-Fi settings of the Control Hub. Below are the steps to access the Robot Controller Console through the REV Hardware Client and the Driver Station application for updating Wi-Fi settings.

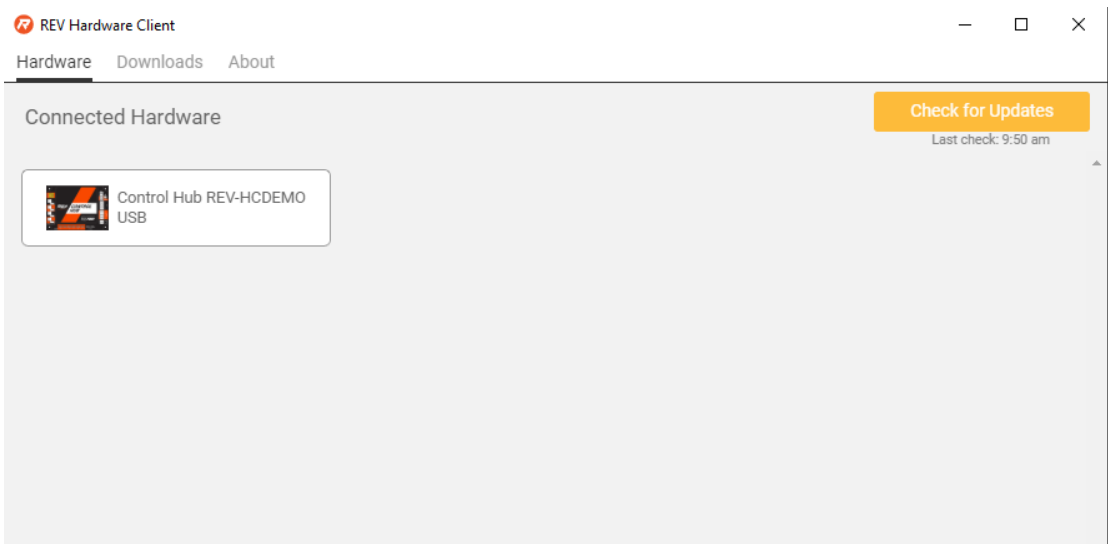
REV Hardware Client

The REV Hardware Client allows teams access to the Hub's Wi-Fi Settings information through a wired connection. The information is visible through the main page of the Robot Control Console and updated through the Program and Manage tab.

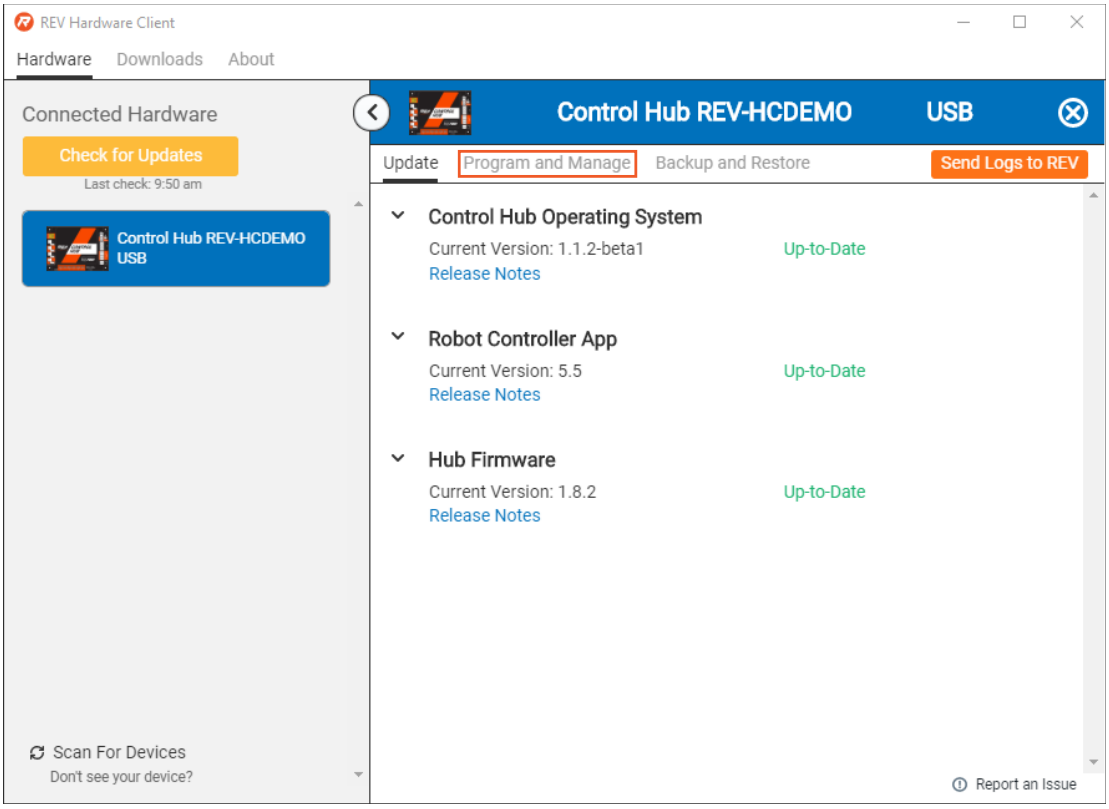
Download the latest version of the REV Hardware Client and install on a Windows PC. Skip this step if completed already.

Steps	
Power on the Control Hub, by plugging the 12V Slim Battery (REV-31-1302) into the XT30 connector labeled “BATTERY” on the Control Hub.	
The Control Hub is ready to connect with a PC when the LED turns green. Note: the light blinks blue every ~5 seconds to indicate that the Control Hub is healthy.	
Plug the Control Hub into the PC using a USB-A to USB-C Cable (REV-11-1232)	

Startup the REV Hardware Client. Once the hub is fully connected it will show up on the front page of the UI under the **Hardware Tab**. Select the Control Hub.



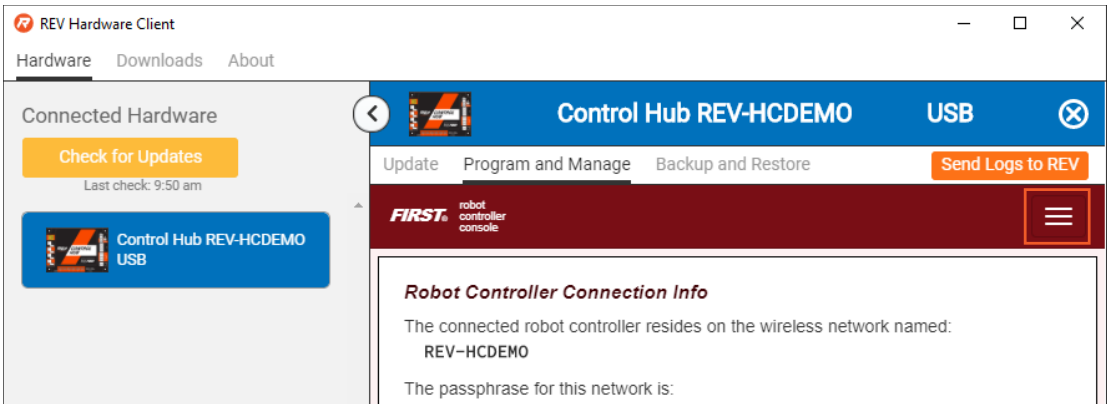
After selecting the Connected Hardware the Update tab will pop up. Select the Program and Manage tab. This will take you to the Robot Controller Console build into the REV Hardware Client.

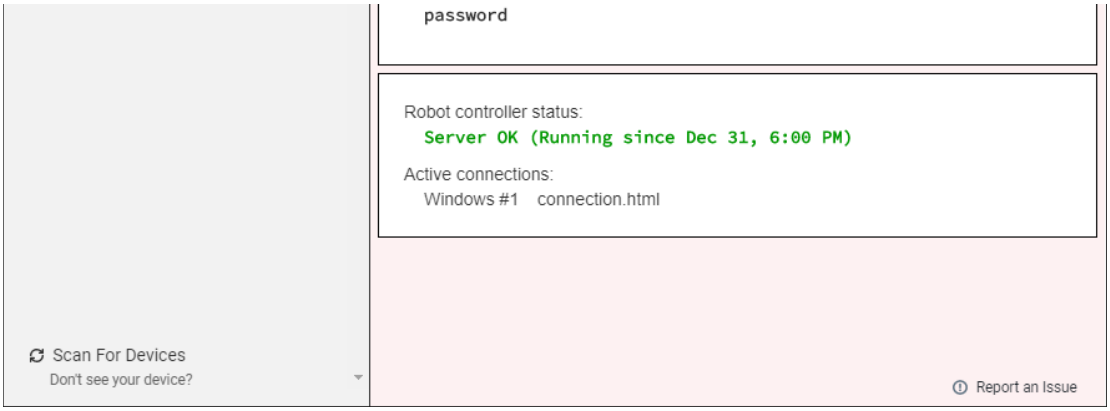


Once in the Robot Controller Console, there are two options.

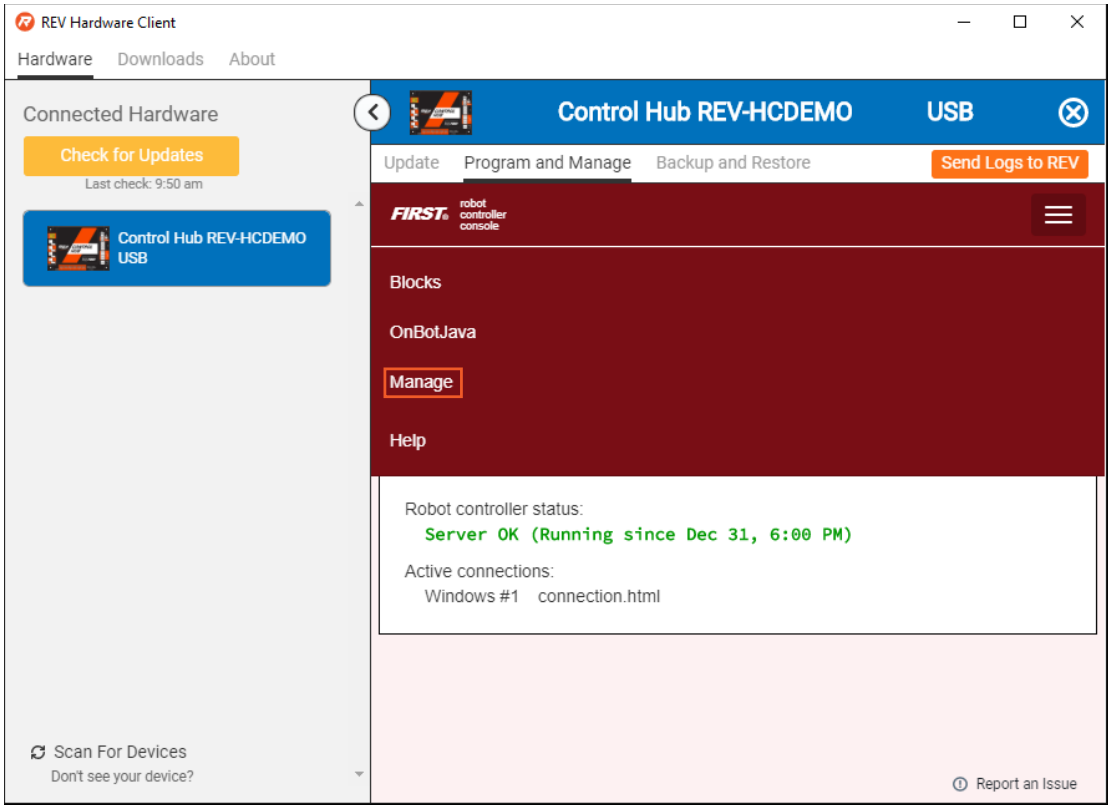
If just the Wi-Fi Access Point name and password need to be found, they can be seen on the main page of the Robot Controller Console.

If any of the Wi-Fi Access Point information needs to be changed, select the menu button in the upper right-hand corner of the page, indicated in the image below.



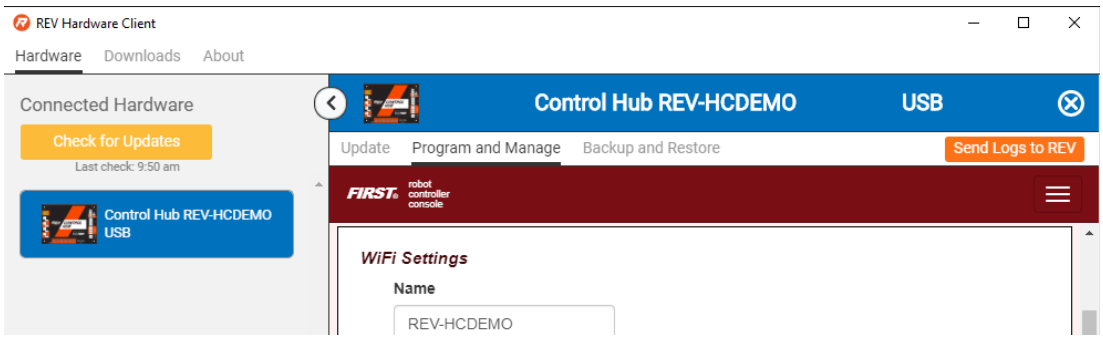


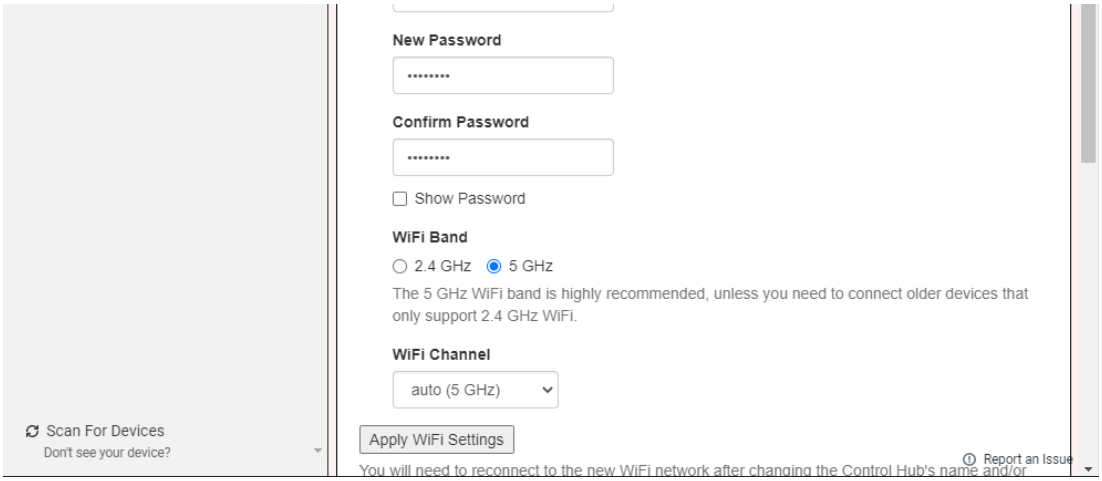
When the menu opens, select Manage.



The Manage page is where the Wi-Fi Access Point information for the Hub can be viewed and changed. In the image below, the Hub's Wi-Fi name, password, band, and channel can be changed. Editing these settings can help when the Hub is not showing up as a potential connection point from a computer or Driver Station device.

Once changes have been made select **Apply Wi-Fi Settings**.



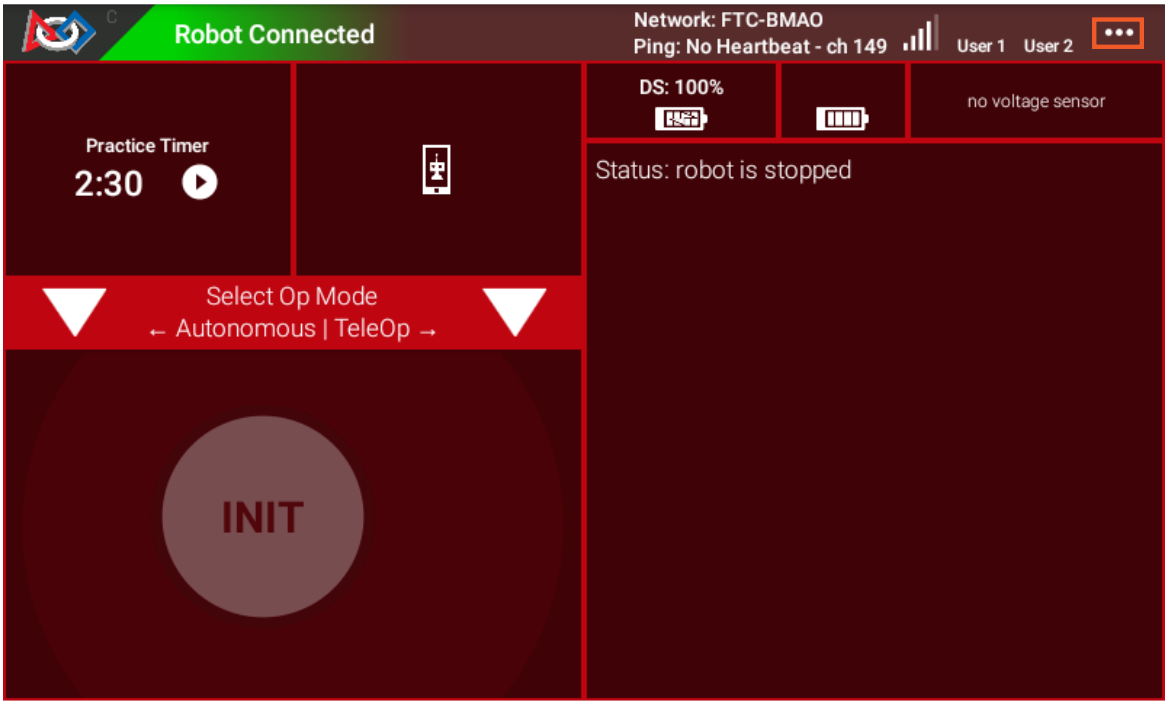


> Once updates are made to the network reconnection to the new Wi-Fi network is needed. When accessing the REV Hardware Client via a USB connection the Control Hub will stay connected to the REV Hardware Client. Rescanning for devices is necessary for changes to show in the Hardware Client.

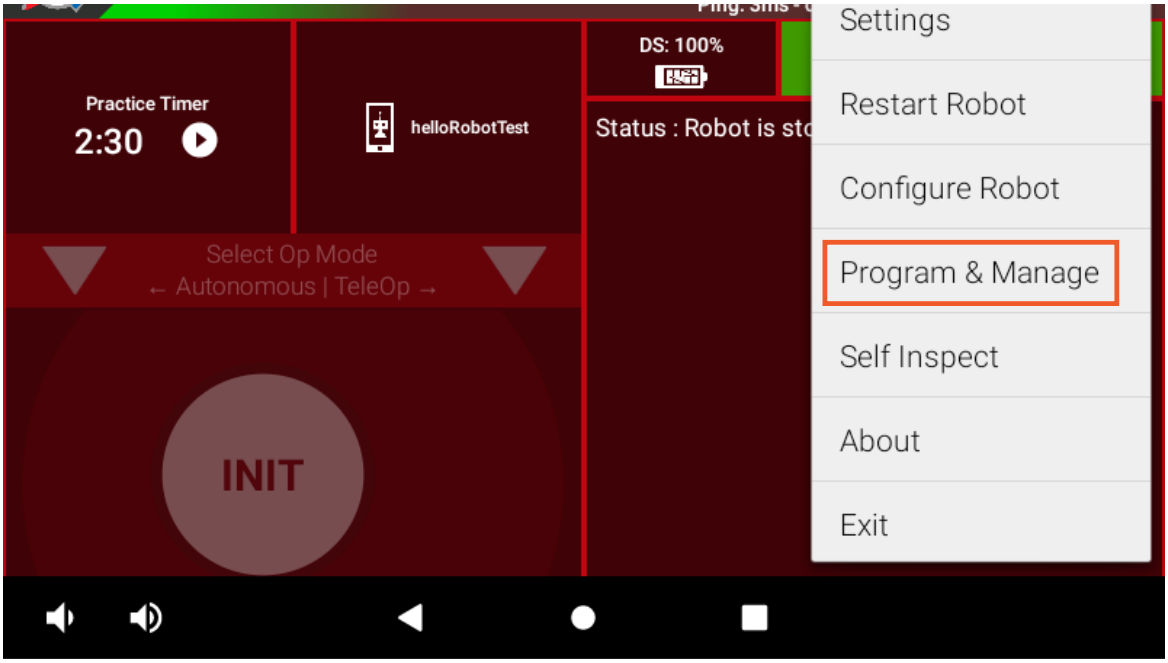
Driver Station Application

The Manage page of the Robot Controller Console can also be accessed via the Driver Station Application. This is helpful in event environments, where Field Technical Staff may request that you change Wi-Fi bands or channels to mitigate disconnections.

Select the three horizontal dots in the upright corned of the Driver Station Application



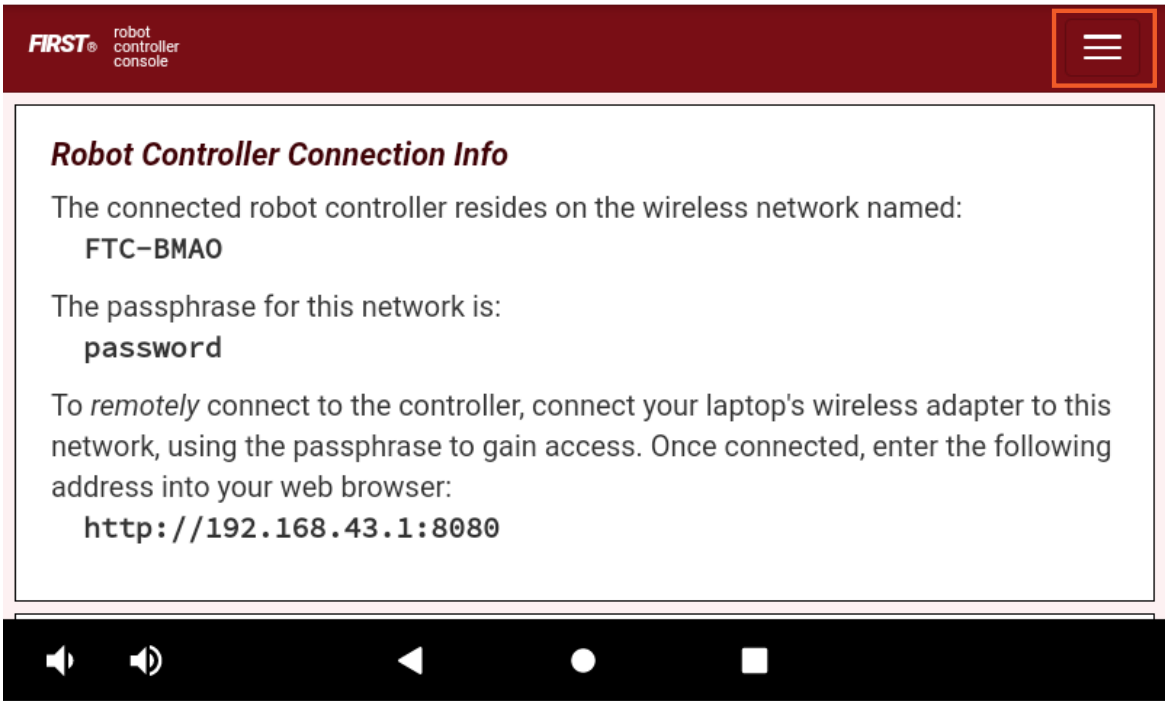
In the drop down menu select **Program & Manage**.



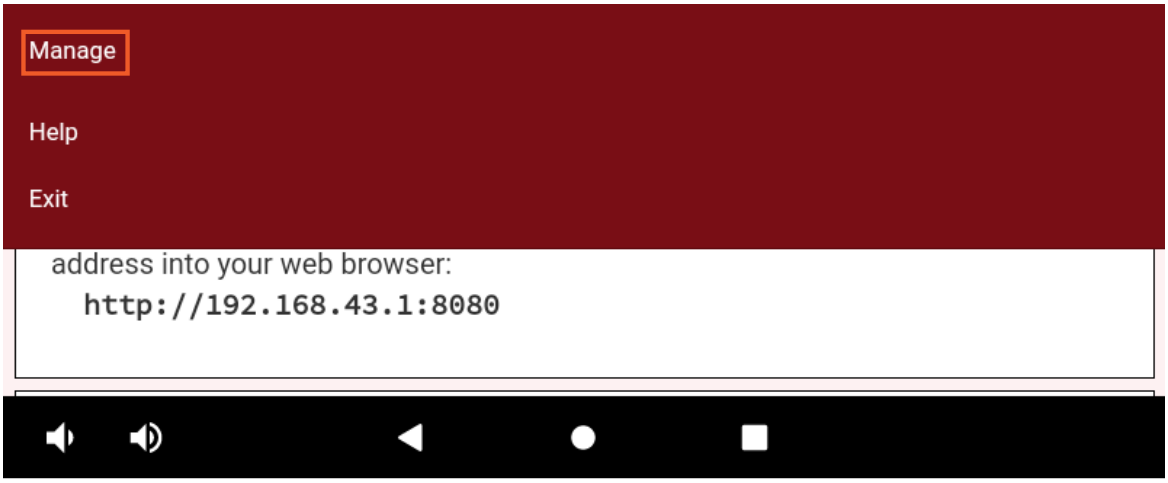
Once in the Robot Controller Console, there are two options.

If just the Wi-Fi Access Point name and password need to be found, they can be seen on the main page of the Robot Controller Console.

If any of the Wi-Fi Access Point information needs to be changed, select the menu button in the upper right-hand corner of the page, indicated in the image below.

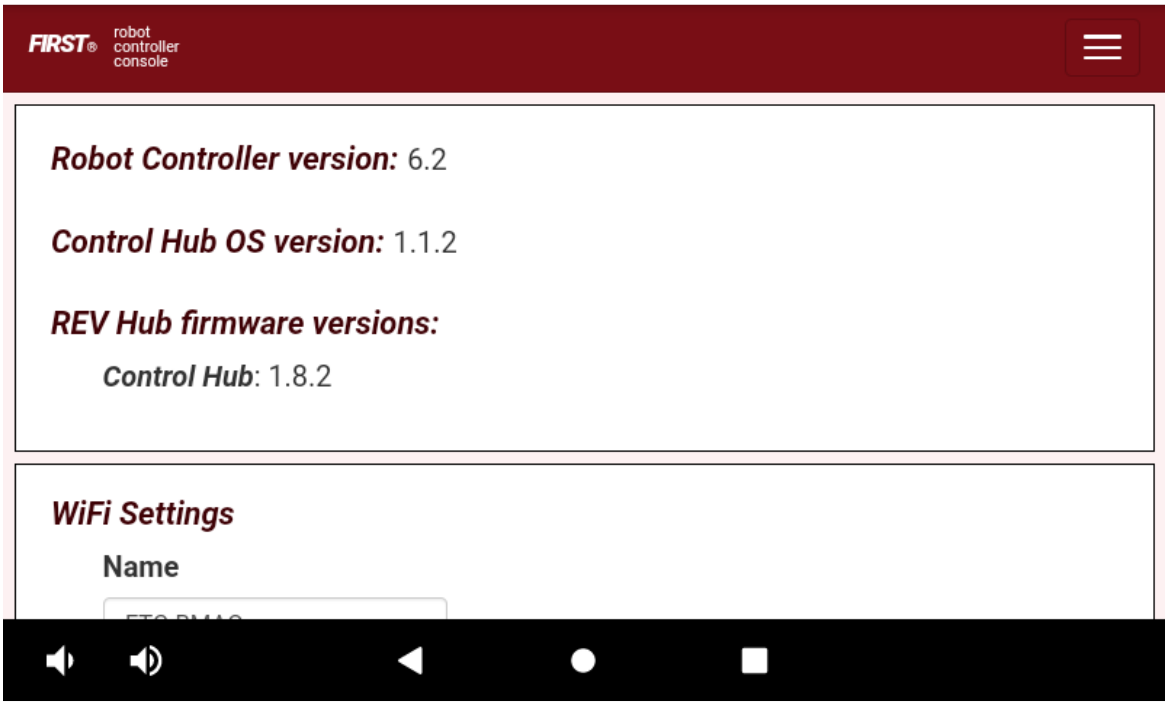


When the menu opens, select Manage.



The Manage page is where the Wi-Fi Access Point information for the Hub can be viewed and changed. In the image below, the Hub's Wi-Fi name, password, band, and channel can be changed.

Once changes have been made select **Apply Wi-Fi Settings**.



> ⚠ You will need to reconnect to the new Wi-Fi network after changing the name/and or password.

Using the User Button

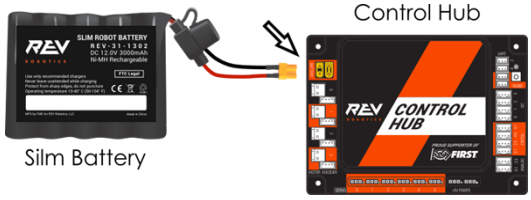
The Control Hub has a user button underneath the LED on the right side of the device. This button allows for a Wi-Fi reset or changing the Wi-Fi band currently being used on the Control Hub.

Wi-Fi Reset

If you are unable to connect to the Control Hub's Wi-Fi after switching to the 5 GHz band, you can perform a

Wi-Fi factory reset. The Wi-Fi network name and password will be reset to their default values, and the Wi-Fi band will be set to 2.4 GHz. To perform a Wi-Fi reset, please follow the steps below.

> The Wi-Fi reset can take several minutes to complete.

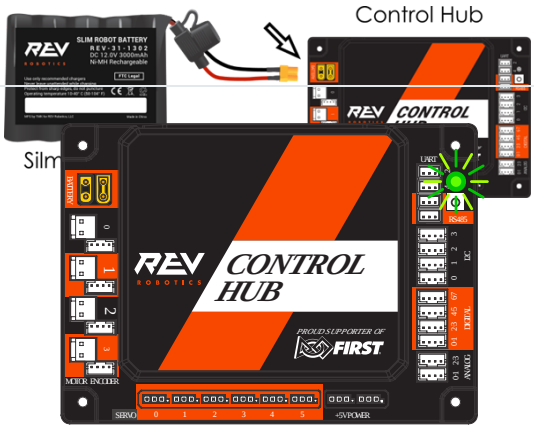
Step	Image
Press and hold the button on the front of the Control Hub.	
While pressing the button, power on the Control Hub.	
Release button when the Control Hub LED begins to flash a multitude of colors. When the Control Hub flashes Blue then Green it has completed the reset and is ready to connect.	

> When the Control Hub flashes Blue then Green it has completed the reset and is ready to connect. The Wi-Fi network will reset back to the default name and password.

Changing Wi-Fi Band

When running version 1.1.2 or later of the Operating System, the Control Hub can switch between the 2.4GHz and 5GHz Wi-Fi bands without access to the REV Hardware Client or the Robot Controller Console. This will only change the Wi-Fi band. When switching to a Wi-Fi band this way, the most recent channel selected on that band will be used (defaulting to auto).

Step	Image
While pressing the button, power on the Control Hub.	

Press and hold the button on the front of the Control Hub after the Control Hub has fully booted (LED is solid green)	
Release button when the Control Hub LED flashes MAGENTA or YELLOW.	

> ⓘ The Control Hub's LED blinks magenta when the band is switched to 5 GHz and yellow when the band is switched to 2.4 GHz.

REV Hardware Client

The REV Hardware Client is software designed to make managing REV devices easier for the user. This Client automatically detects connected device(s), downloads the latest software for those device(s), and allows for seamless updating of the device(s).

For more information on the REV Hardware Client, see the User's Manual.

Latest REV Hardware Client - Version 1.4.3
Download Latest REV Hardware Client

Feature Summary

- Automatically detect supported devices when connected via USB
- Connect a REV Control Hub via Wi-Fi
- One Click update of all software on connected devices
- Pre-download software updates without a connected device
- Back up and restore user data from Control Hub
- Install and switch between DS and RC applications on Android Devices
- Access the Robot Control Console on the Control Hub
- Auto-update to latest version of the REV Hardware Client
-

Display devices connected via RS485

Supported Devices

- REV Control Hub (REV-31-1595)
- REV Expansion Hub (REV-31-1153)
- REV Driver Hub (REV-31-1596)
- Android Device via ADB

Updating Firmware

Updating the Expansion Hub Firmware

There are two boards within the Control Hub: an Expansion Hub and an Android controller. The Expansion Hub board built into the Control Hub, facilitates a line of communication between the built in Robot Controller and the motors, servos, and sensors. In order to improve the quality of the Hubs, REV Robotics will release firmware updates for the Expansion Hub. When a firmware release occurs, both Control Hub and Expansion Hub users will need to update their Expansion Hub firmware to the newest version.

There are two ways to update the Expansion Hub Firmware. It is recommended to use the REV Hardware Client as it will automatically notify the user if the Hub's firmware is out of date, download the latest firmware, and install on the device. The second set of steps utilizes the FIRST Robot Controller Console.

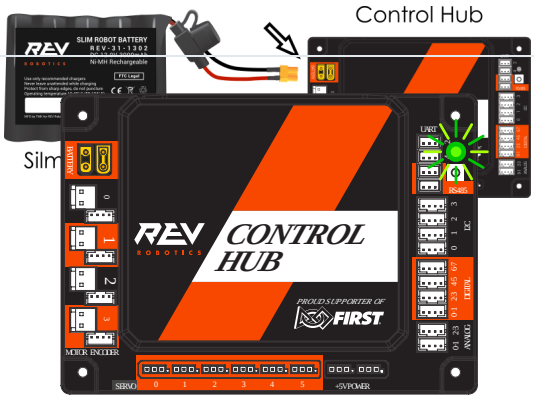
To use the FIRST Robot Controller Console, the *Manage* interface is needed to upload the firmware file to the Control Hub. You can then use a Driver Station that is connected to the Control Hub to initiate the firmware update. You can download the latest firmware below.

Using the REV Hardware Client

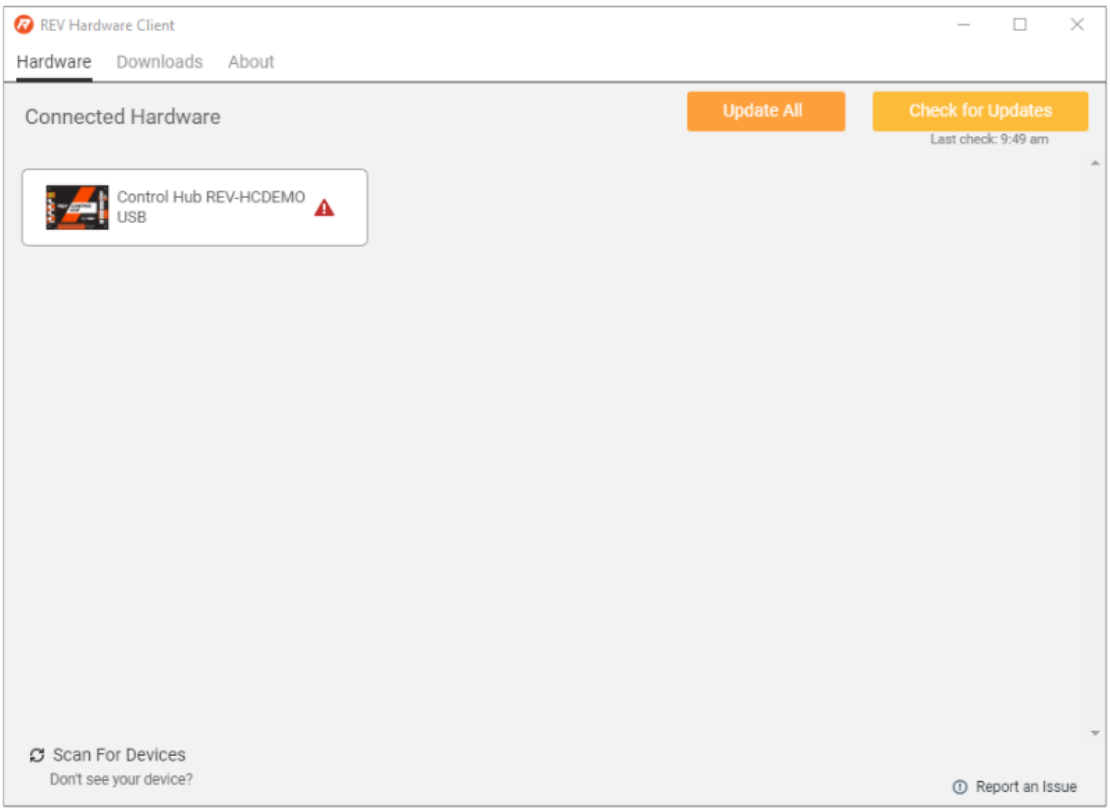
Control Hub

> In order to use the REV Hardware Client for firmware updates, the Robot Controller Application must first be updated to version 5.5. After updating the application you may need to close out of the REV Hardware Client in order for the firmware update to be available.

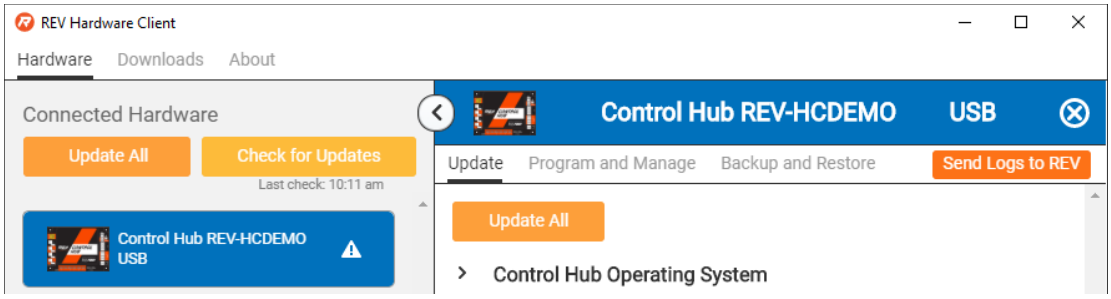
Steps	
Power on the Control Hub, by plugging the 12V Slim Battery (REV-31-1302) into the XT30	

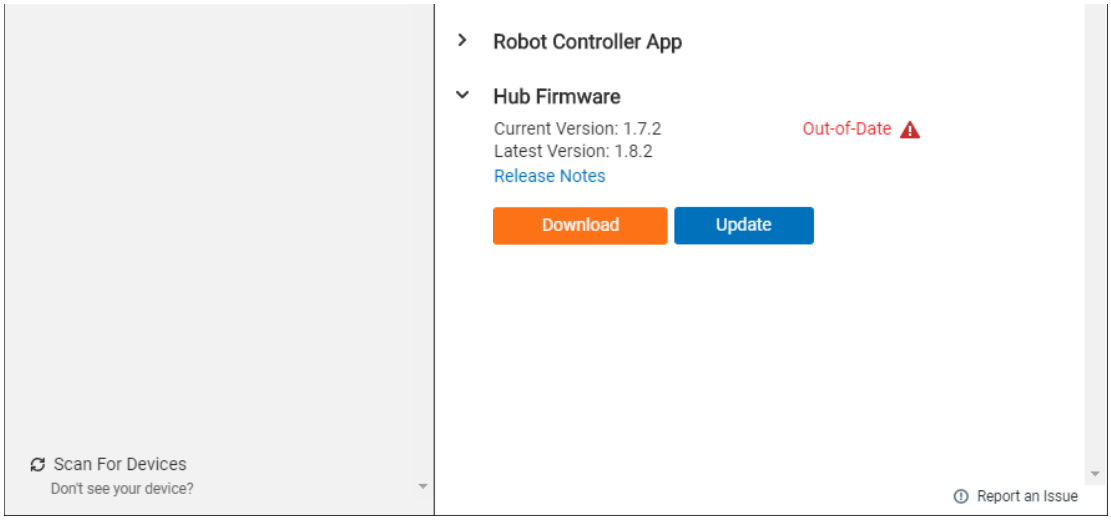
connector labeled “BATTERY” on the Control Hub.	
The Control Hub is ready to connect with a PC when the LED turns green. **Note:** With Robot Controller Application versions 5.5 and below the light will blink blue every ~5 seconds. Please update to 6.0.	
Plug the Control Hub into the PC using a USB-A to USB-C Cable (REV-11-1232)	

Startup the REV Hardware Client. Once the Control Hub is fully connected it will show up on the front page of the UI under the **Hardware Tab**. Select the Control Hub.

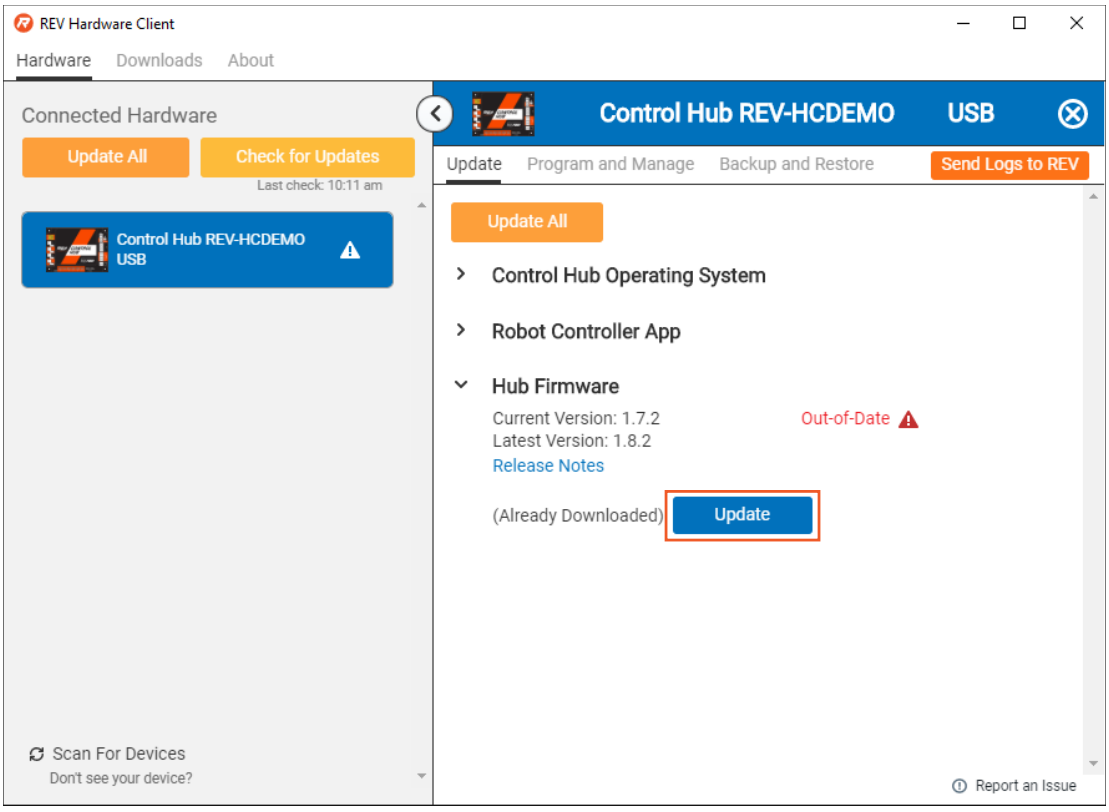


After selecting the Connected Hardware the Update tab will pop up. Under **Hub Firmware** select Download.

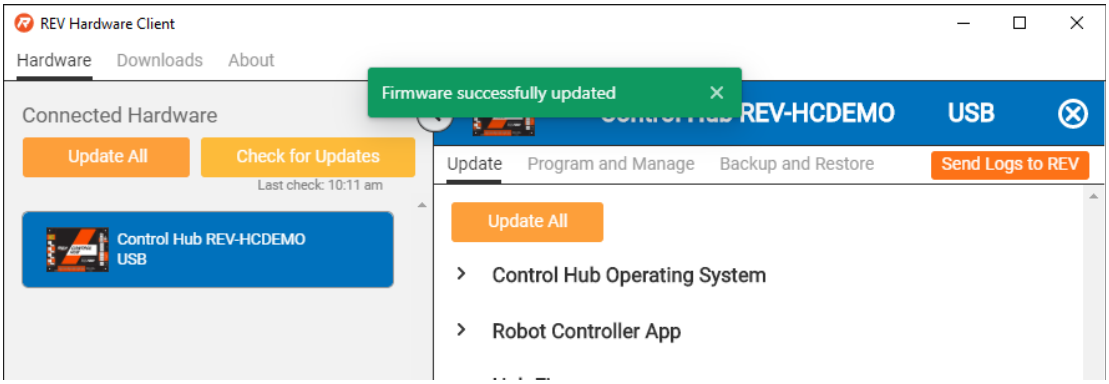


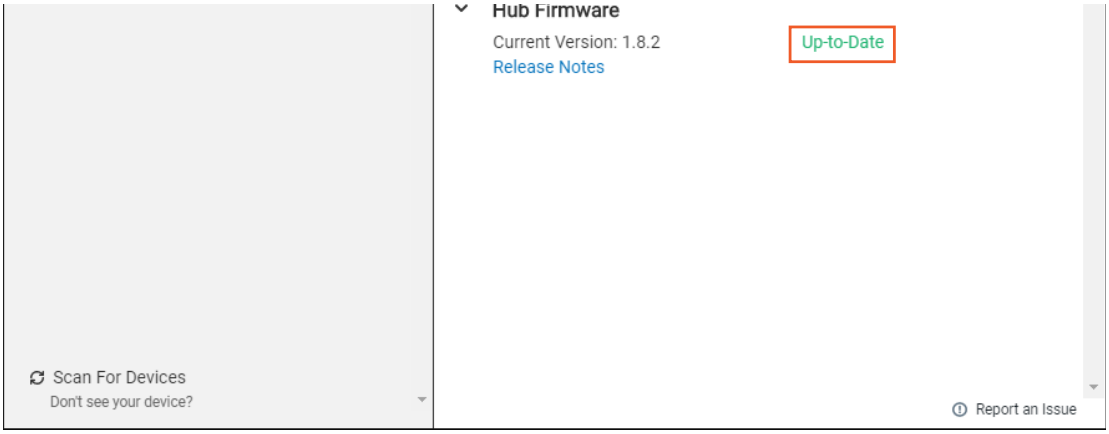


Once the firmware has downloaded, select Update.

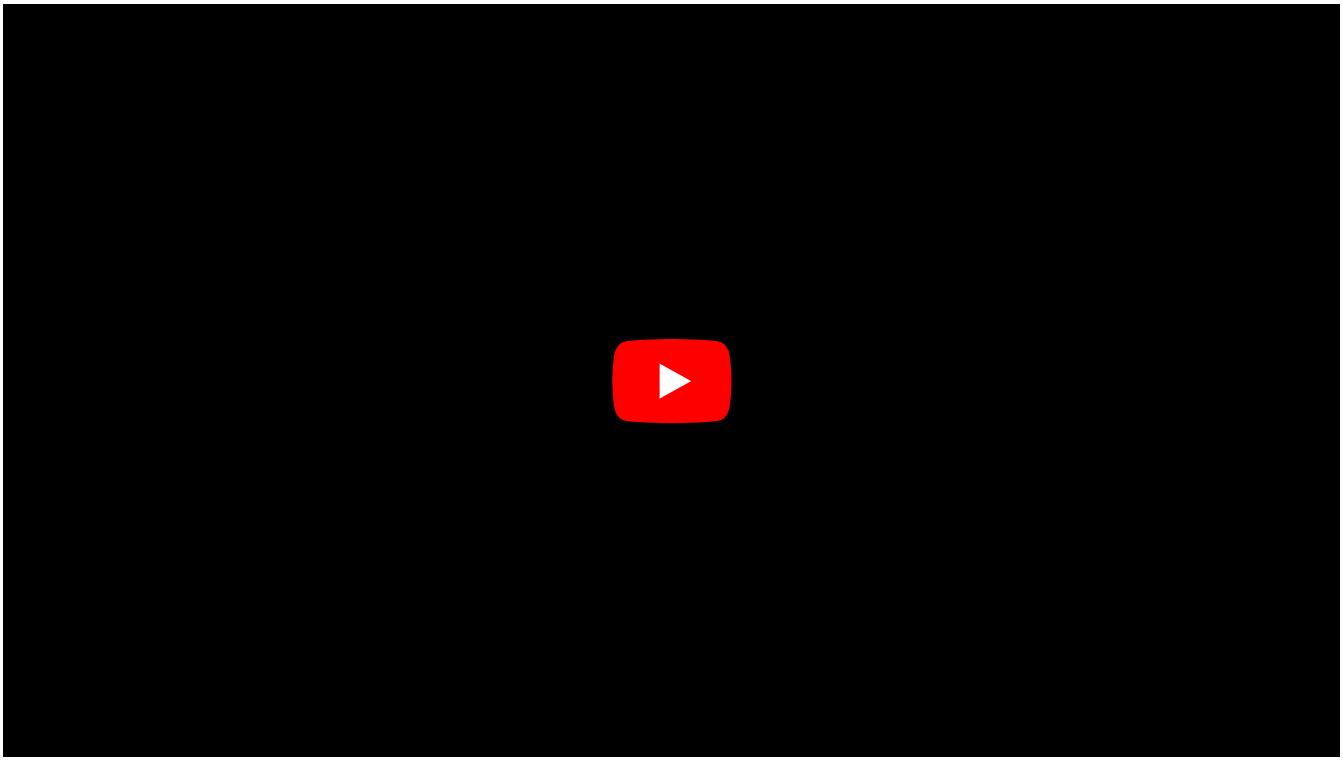


When the firmware update has completed a status message "Firmware successfully updated" The status for the Hub Firmware will also change to "Up-to-Date."



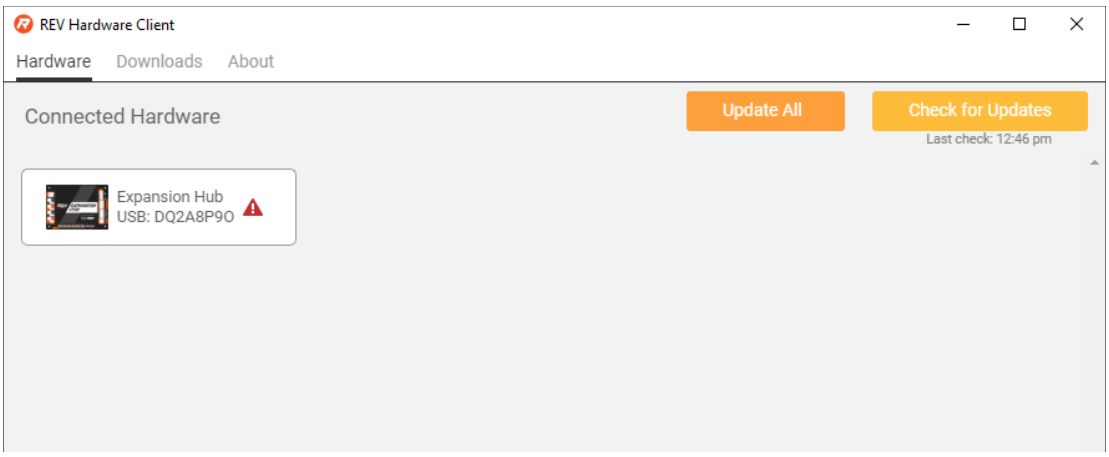


Expansion Hub

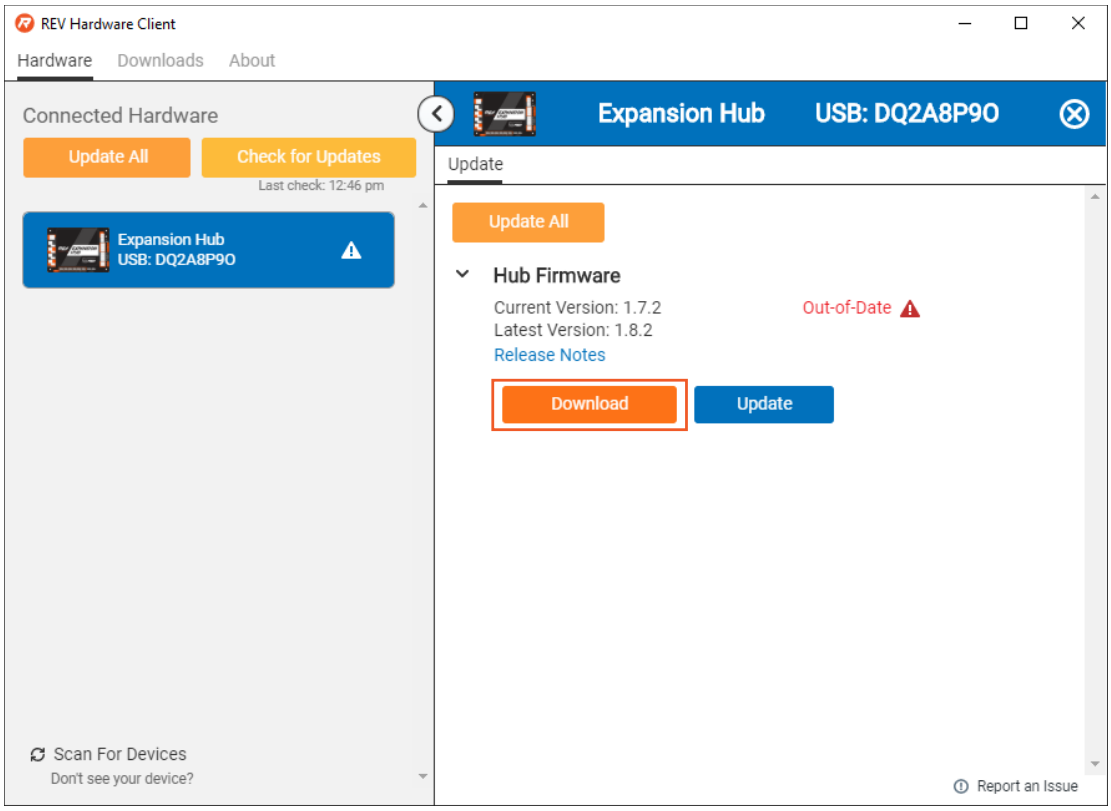


Plug the Expansion Hub into a PC using a USB-A to Mini USB Cable.

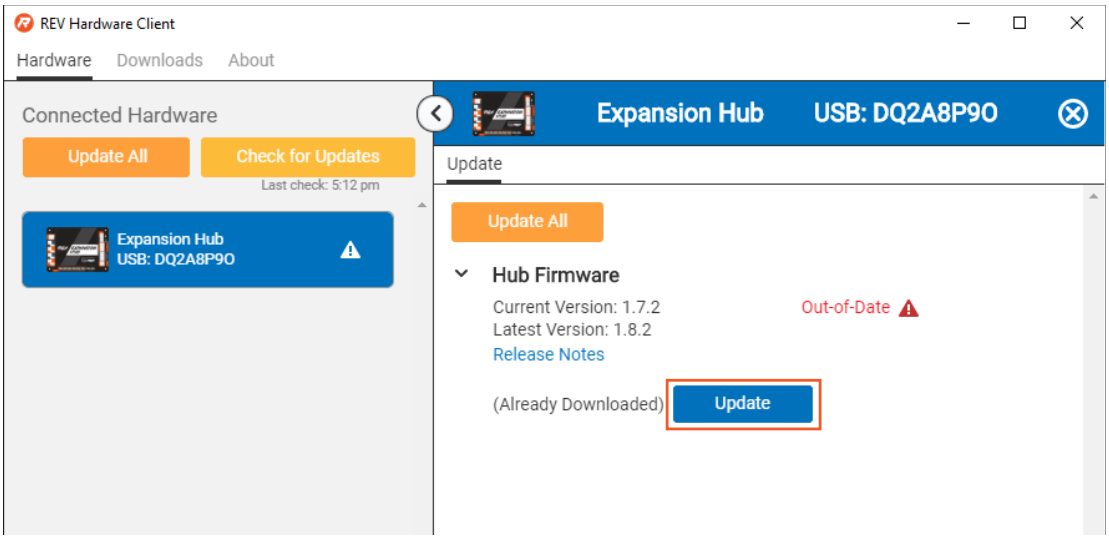
Startup the REV Hardware Client. Once the hub is fully connected it will show up on the front page of the UI under the **Hardware Tab**. Select the Expansion Hub.



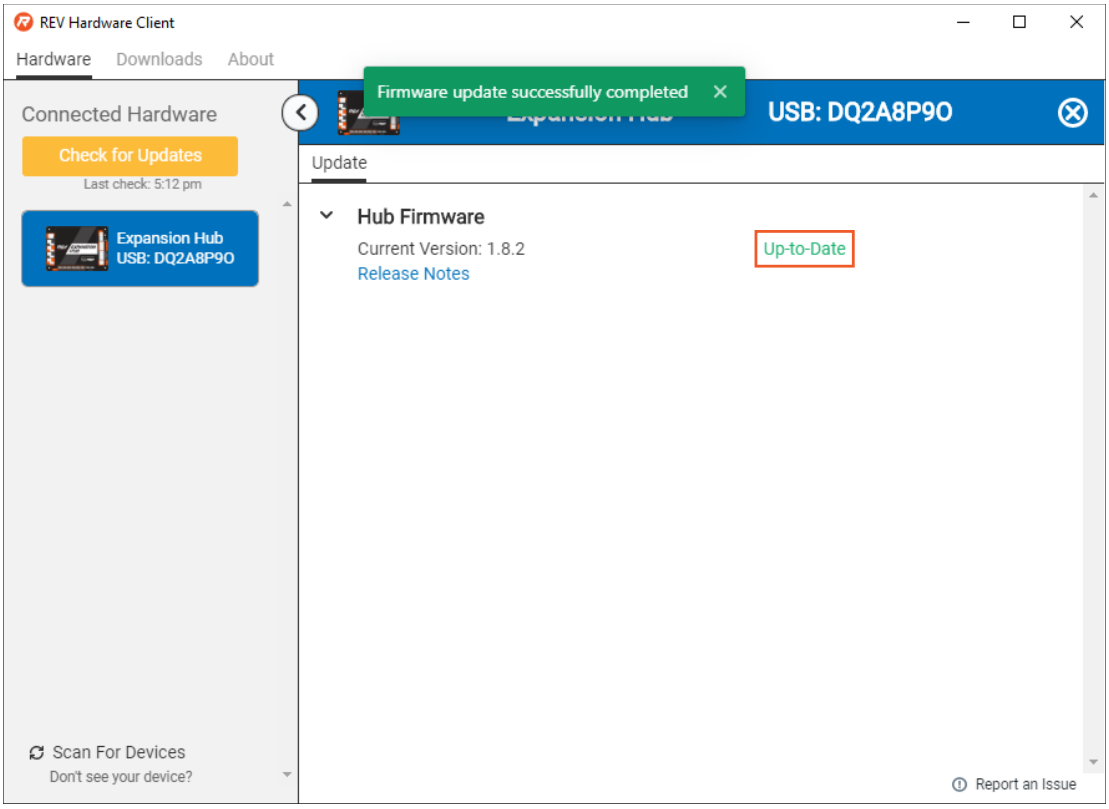
After selecting the Connected Hardware the Update tab will pop up. Under **Hub Firmware** select Download.



Once the firmware has downloaded, select Update.



When the firmware update has completed a status message "Firmware successfully updated" The status for the Hub Firmware will also change to "Up-to-Date."



Using the Robot Controller Console

Download the Latest REV Hub Firmware - Version 1.08.02

Updating the Expansion Hub Firmware

1. On the *Manage* page of the Control Hub user interface, press the *Select Firmware* button to to select the firmware file that you would like to upload.

An *Upload* button should appear after you successfully selected a file.

2. Press the *Upload* button to upload the firmware file from your computer to the Control Hub.

The words "Firmware upload complete" should appear once the file has been uploaded successfully.

3. On the Driver Station, touch the three dots in the upper right hand corner to display a pop-up menu.

4. Select *Settings* from the pop-up menu to display the Settings activity.

5. On the Driver Station, scroll down and select the *Advanced Settings* item (under the *ROBOT CONTROLLER SETTINGS* category).

6. Select the *Expansion Hub Firmware Update* item on the *ADVANCED ROBOT CONTROLLER SETTINGS* activity.

7. If a firmware file that is different from the version currently installed on the Expansion Hub was successfully uploaded, the Driver Station should display some information about the current firmware version and the new firmware version. Press the *Update Expansion Hub Firmware* button to start the updat process.

8. A progress bar will display while the firmware is being updated. Do not power off the Control Hub/Expansion Hub during this process. The Driver Station will display a message when the update process is complete.

Firmware Changelog

Version 1.8.2 (Latest Version)

- Improved USB recovery in case of fault event (e.g. ESD fault)
- Improved DC motor output linearity
-

Improved closed-loop control modes
- Improved I2C speeds
- Minor bug fixes

Download REV Hub Firmware Version 1.8.2

Version 1.7.2

- Fixes a bug where encoder counts would occasionally reset.

Download REV Hub Firmware Version 1.7.2

Version 1.7.0

- Fixes a bug where some I2C sensors can lock up the bus causing other additional performance issues.
- Added new status LED blink code:
  - Blinking orange indicates the Hub is only powered over USB. In other words, turn on your main power switch!
- Other minor performance tweaks.

Download REV Hub Firmware Version 1.7.0

Version 1.6.0

- Original Release

Download REV Hub Firmware Version 1.6.0

Updating Operating System

The Control Hub's Operating System is field upgradable. New updates are released to incorporate fixes, improvements, and new features as they are developed.

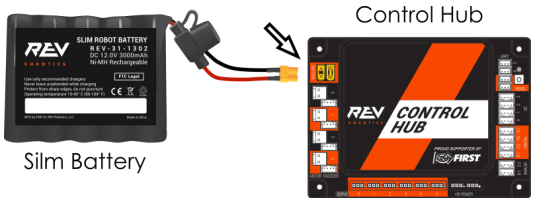
There are two ways you can update the Operating System. It is recommended to use the REV Hardware Client as it will automatically notify the user if the Hub's Operating System is out of date, download the latest OS, and install the OS on the device. The second way utilizes the FIRST Robot Controller Console. For using the FIRST Robot Control Console, you will need to download the latest Operating System.

> ⓘ Updating the Operating System can take some time depending on the size of the update. Expect the update to take approximately 5 minutes to fully complete and keep the Control Hub powered during this process.

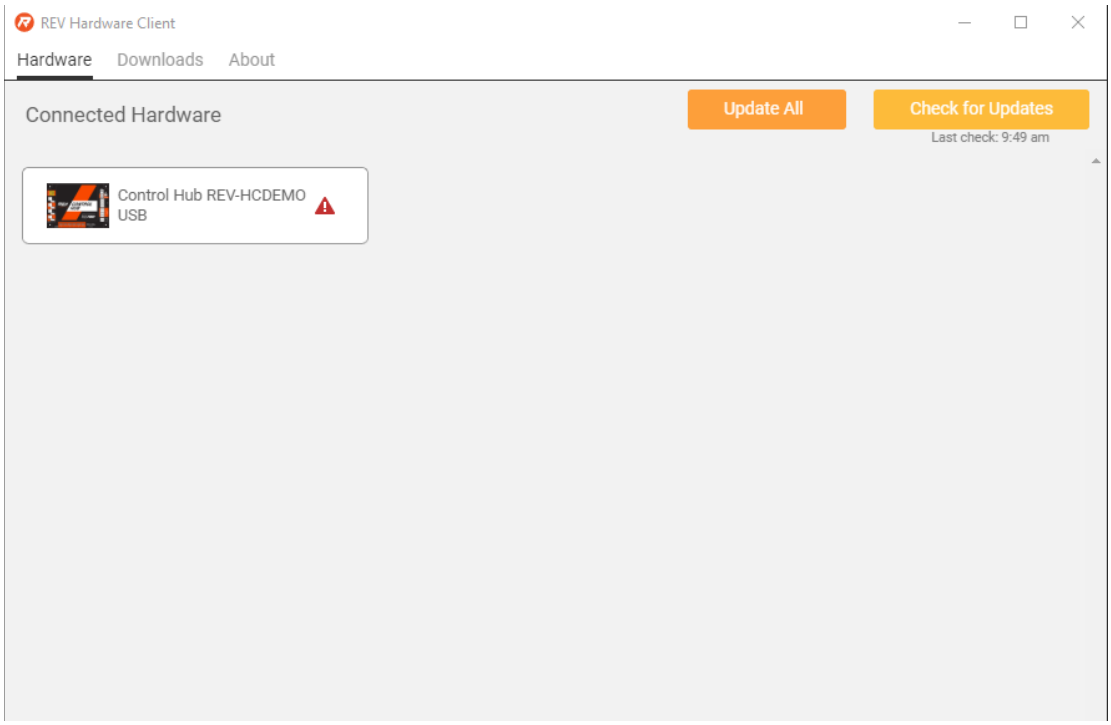
> ⚠ The following procedure works with Control Hubs with the part number REV-31-1595. For

> support using the REV-31-1152 Control Hub v0 please reach out to REV support (support@revrobotics.com).

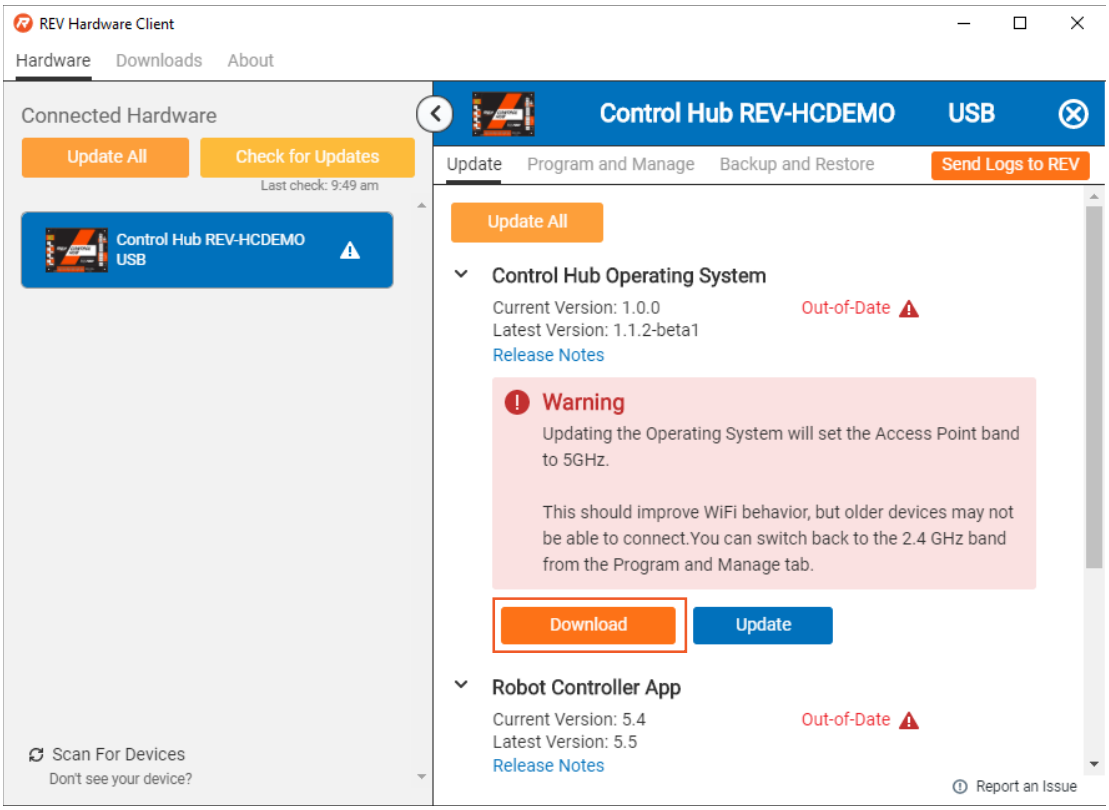
Using the REV Hardware Client

Steps	
Power on the Control Hub, by plugging the 12V Slim Battery (REV-31-1302) into the XT30 connector labeled “BATTERY” on the Control Hub (REV-31-1595).	
The Control Hub is ready to connect with a PC when the LED turns green. **Note:** With Robot Controller Application versions 5.5 and below the light will blink blue every ~5 seconds. Please update to 6.0.	
Plug the Control Hub into the PC using a USB-A to USB-C Cable (REV-11-1232)	

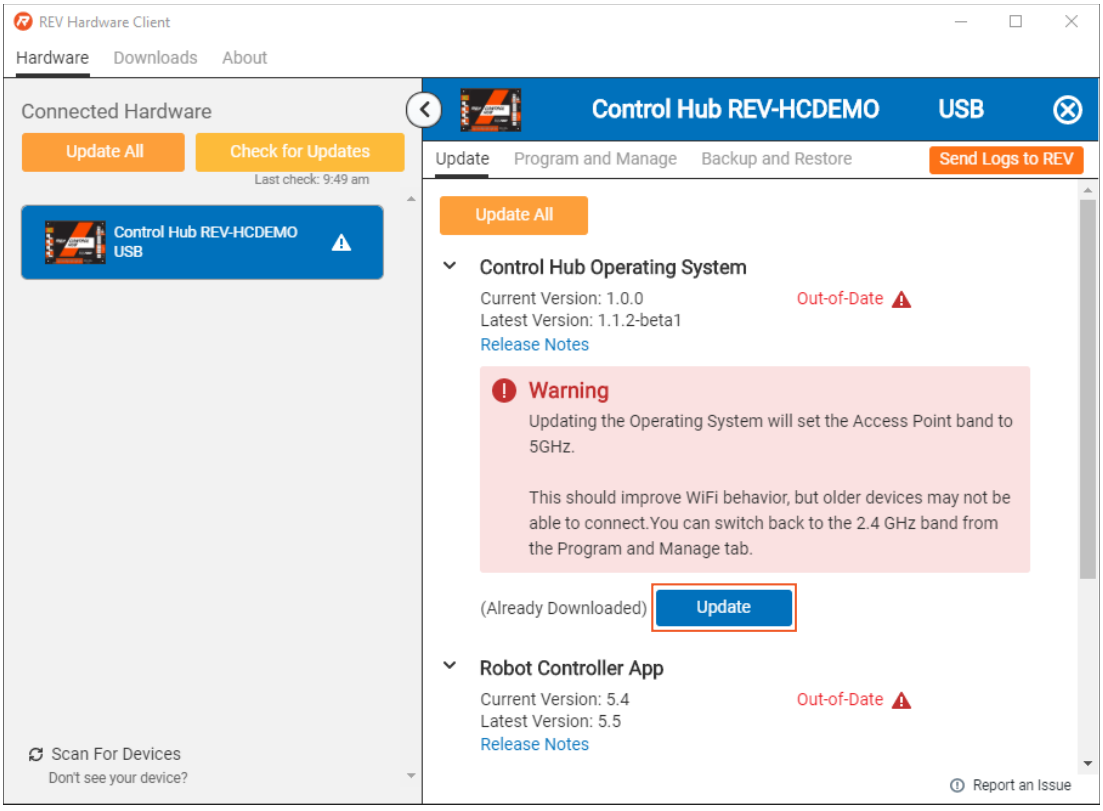
Startup the REV Hardware Client. Once the hub is fully connected it will show up on the front page of the UI under the **Hardware Tab**. Select the Control Hub.



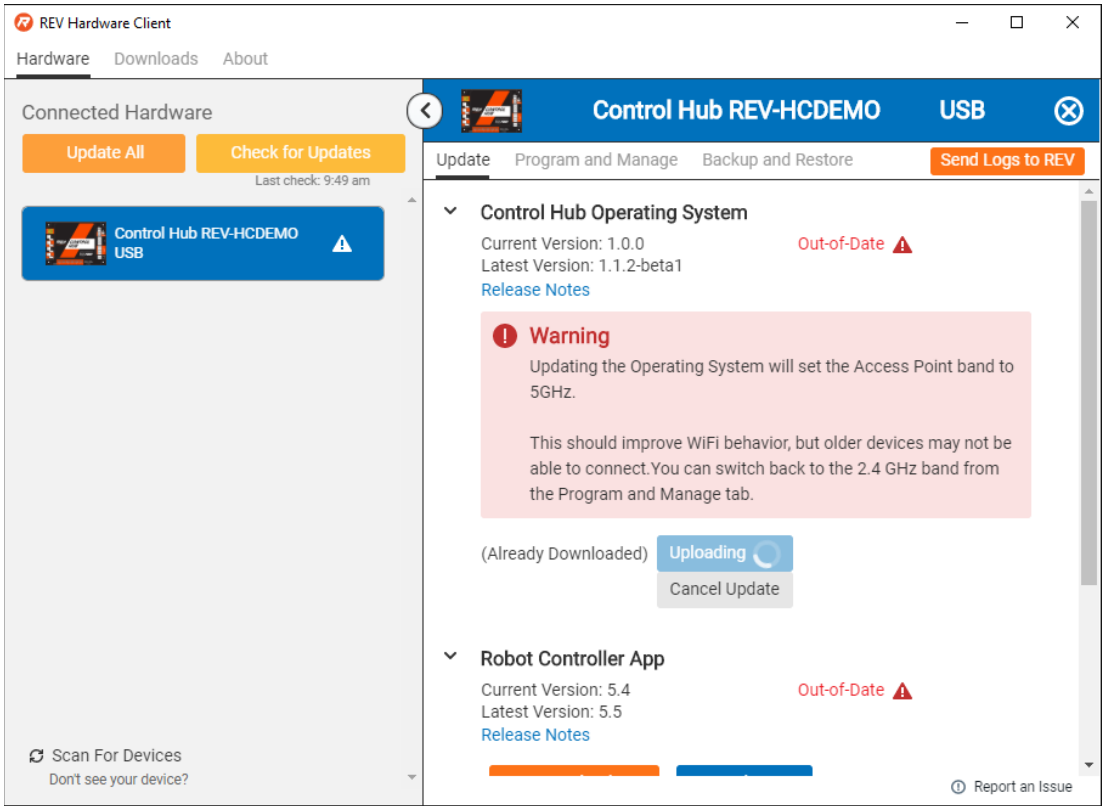
After selecting the Connected Hardware the Update tab will pop up. Under **Control Hub Operating System** select Download.



Once the OS has downloaded, select Update.

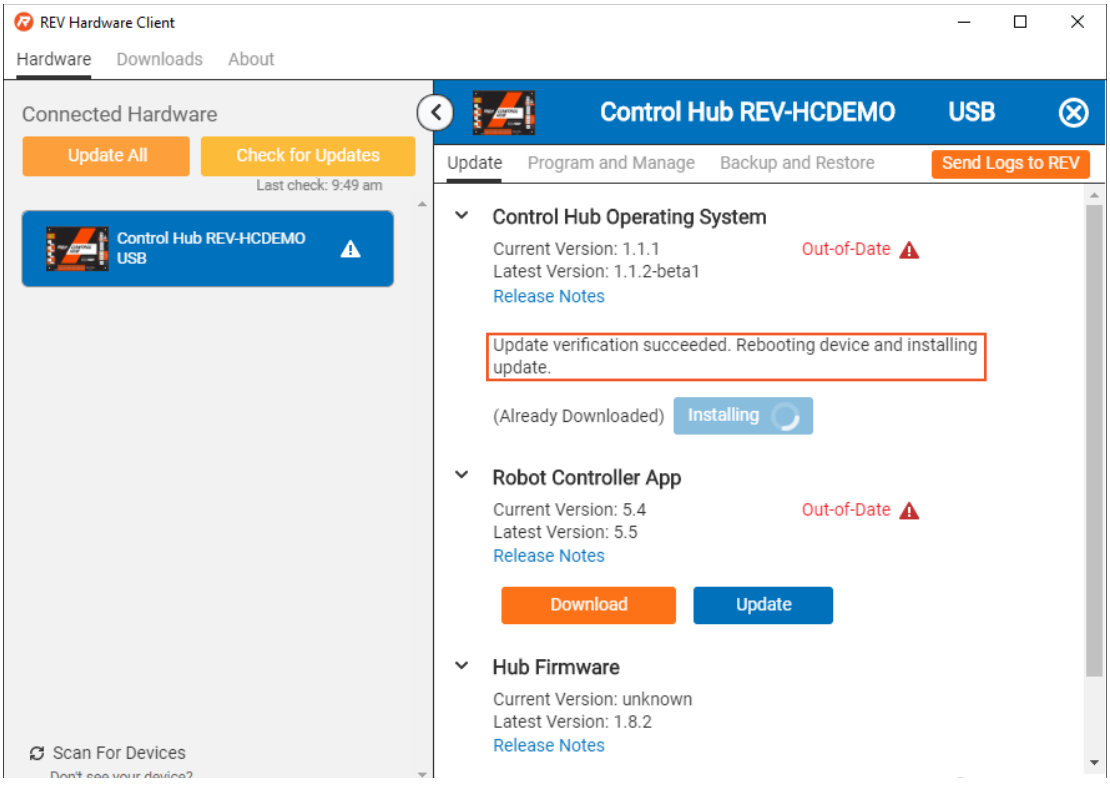


Keep the Control Hub powered while the upload finishes.



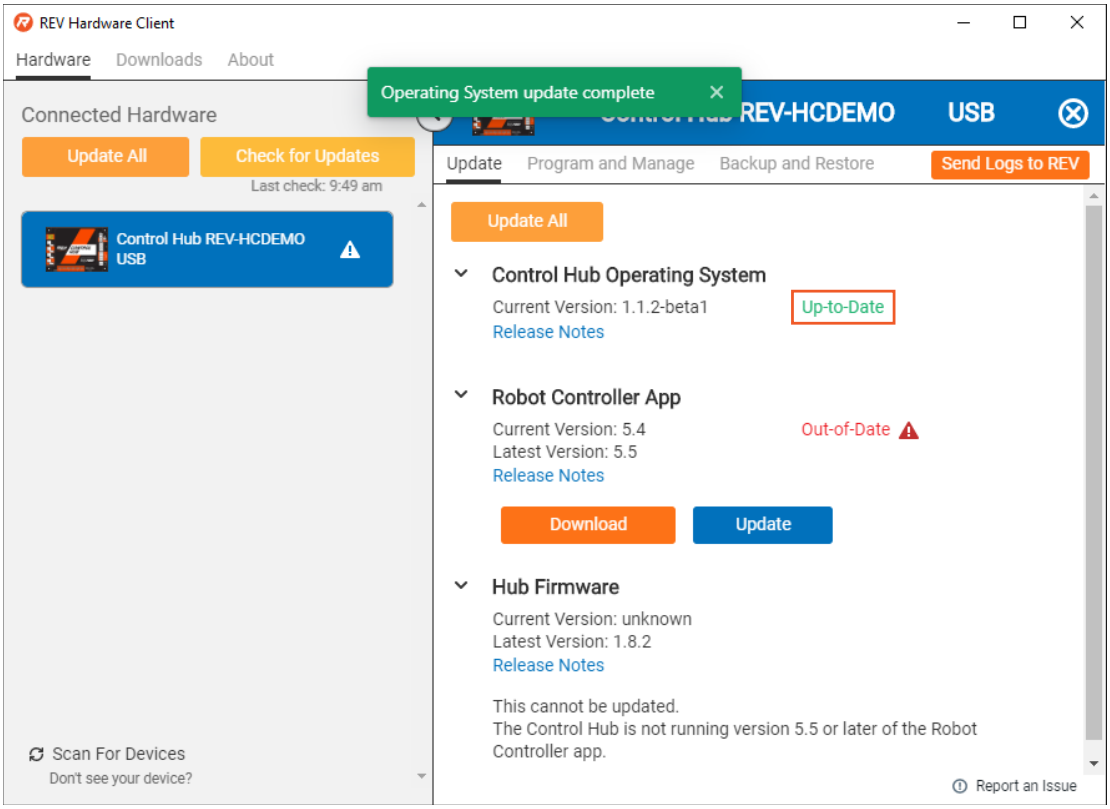
A successful upload will be denoted by the "Update Verification Succeeded" message highlighted in the image below. Once the upload is successful the install will begin.

Keep the Control Hub powered while the update is installed. The Control Hub will reboot to complete the update.



When the OS update has completed a status message "Operating System update complete." The status for the Control Hub Operation System will also change to "Up-to-Date."

> When using OS 1.1.2 the Control Hub operates by default on the 5Ghz band. You may need to update the Wi-Fi settings depending on what Driver Station device you are using.



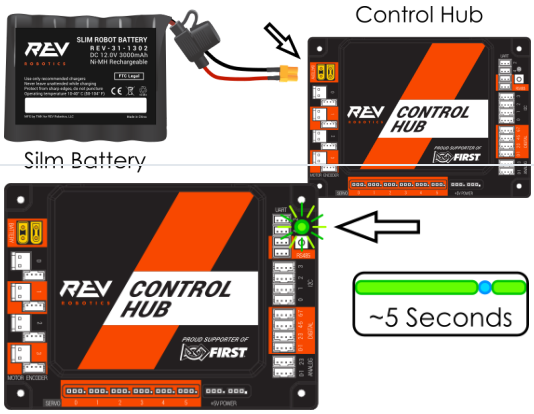
Using the Robot Controller Console

Download the Latest REV Control Hub Operating System - Version 1.1.3

> When updating from OS 1.1.1 or earlier to OS 1.1.2 or later, the Control Hub will switch to the 5 GHz band, regardless of the previous Wi-Fi band setting. Some devices do not support 5 GHz Wi-Fi, and will not be able to connect to the Control Hub wirelessly while it is using the 5 GHz Wi-Fi band. To switch to the 2.4 GHz band without needing a computer, see the Changing Wi-Fi Band section.

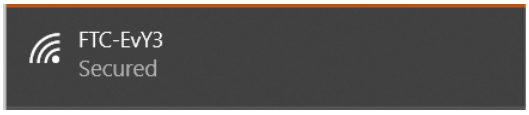
Step	Image

Power on the Control Hub, by plugging the 12V Slim Battery (REV-31-1302) into the XT30 connector labeled “BATTERY” on the Control Hub (REV-31-1595).



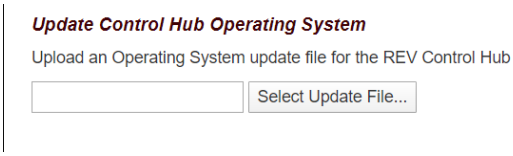
The Control Hub is ready to connect with a PC when the LED turns green. Note: the light blinks blue every ~5 seconds to indicate that the Control Hub is healthy.

Connect to the Control Hub’s Wi-Fi Network. If it is not renamed, the name will begin with either “FIRST-“ or “FTC-“.

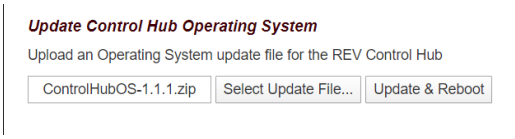


Open a browser and navigate to the FIRST Robot Controller Console (type 192.168.43.1:8080 in the navigation bar). Select the Manage Tab.

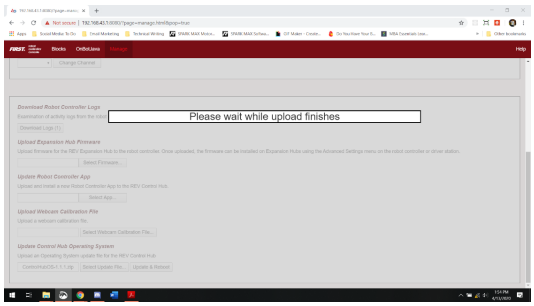
Scroll down to “Update Control Hub Operating System” and press the “Select Update File” button.

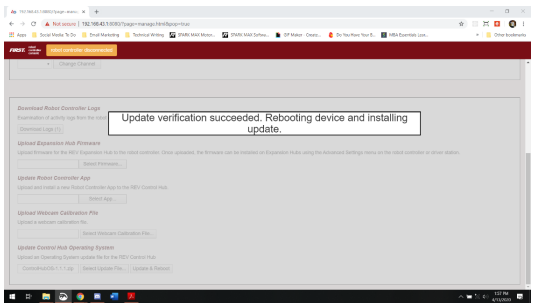
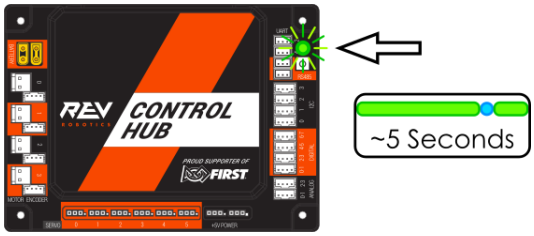
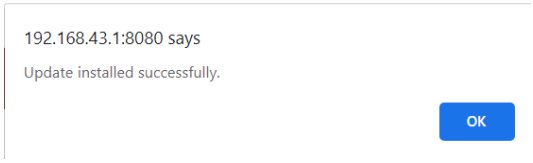


Choose the latest version downloaded in Step 1 and press the “Update & Reboot” button.



Keep the Control Hub powered while the upload finishes.



Keep the Control Hub powered while the update is installed. The Control Hub will reboot to complete the update.	
When the OS update has completed, the Control Hub LED will switch from blue, back to its normal blink pattern.	
Reconnect your computer to the Control Hub network and verify that the update was a success.	

Control Hub Operating System Changelog

> When updating from OS 1.1.1 or earlier to OS 1.1.2 or later, the Control Hub will switch to the 5 GHz band, regardless of the previous Wi-Fi band setting. Some devices do not support 5 GHz Wi-Fi, and will not be able to connect to the Control Hub wirelessly while it is using the 5 GHz Wi-Fi band. To switch to the 2.4 GHz band without needing a computer, see the Changing Wi-Fi Band section.

Version 1.1.3 - Latest Version

- Adds support for new alternative built-in BHI260AP IMU on Control Hub
- Improves reliability of the Wi-Fi access point monitoring feature

Version 1.1.2

-

Adds support for Auto Channel Selection, where the Control Hub will pick the least busy Wi-Fi channel on the selected Wi-Fi band when it starts up
- Migrates all users to Auto Channel Selection on the 5 GHz band by default.
    - If you find that you are unable to connect to the Control Hub after updating, you should perform a Wi-Fi Factory Reset by holding down the Control Hub's button as it boots, until you see a colorful light sequence. That will reset the Wi-Fi settings and switch to the 2.4 GHz Wi-Fi band.
- Allows switching the Wi-Fi band between 2.4 GHz and 5 GHz by holding down the Control Hub's button when the hub has been booted for at least 20 seconds
    - If version 5.5 or later of the Robot Controller app is installed, the Control Hub's light will blink magenta when the band is switched to 5 GHz, or yellow when the band is switched to 2.4 GHz.
- Continuously monitors the Wi-Fi access point status, and will attempt to restart it if it goes down for any reason
- Continuously monitors the Robot Controller app, and restarts it if it crashes or hangs (requires version 6.1 or later of the Robot Controller app)
- Allows the Robot Controller app to access the current Wi-Fi band and channel
- Always backs up the FTC Robot Controller app data before it is uninstalled, in order to preserve Wi-Fi settings
- Improves Wi-Fi reliability
- Prevents issue that could cause device to boot into recovery mode
- Enables use of mouse button in recovery mode

Version 1.1.1

- Fixed bug where Wi-Fi access point would sometimes fail to start after an Operating System update
- Stopped the FtcAccessPointService UI auto-starting on boot
- Allowed Wi-Fi beacon rate to be changed by the FTC Robot Controller app

Version 1.1.0

- Improved reliability of making changes to Wi-Fi access point settings
- Updated to latest Realtek Wi-Fi driver
- Increased Wi-Fi beacon rate to 6mbps, which reduces congestion when many Control Hubs are being used in an area
- Enabled 802.11w, which prevents Wi-Fi deauthentication attacks when the Control Hub is used with a client device that also supports 802.11w
- Added WifiLog.txt file for debugging and disconnection analysis
- Improved reliability of FtcAccessPointService UI (accessed through an HDMI monitor)
- Added 5 GHz channels to FtcAccessPointService UI
- Ensured app data is not lost when installing a Robot Controller with a different signature via the Manage webpage
- Fixed issue where Wi-Fi SSID would sometimes be AndroidAP

Source Files for Control Hub OS:

- Linux Kernel Source
- U-Boot Source

Updating Robot Controller Application

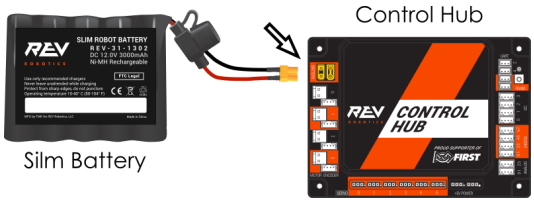
The Robot Controller Application is periodically updated to incorporate fixes, improvements, and new features as they are developed.

> If you update your Robot Controller, then you should also update your Driver Station software to the same version number.

There are two ways you can update the Operating System. It is recommended to use the REV Hardware Client as it will automatically notify the user if the Robot Controller Application is out of date, download the latest APK, and install the APK on the device. The second way utilizes the FIRST Robot Controller Console. For using the FIRST Robot Control Console, you will need to download the latest version of the Robot Controller Application from the GitHub repository.

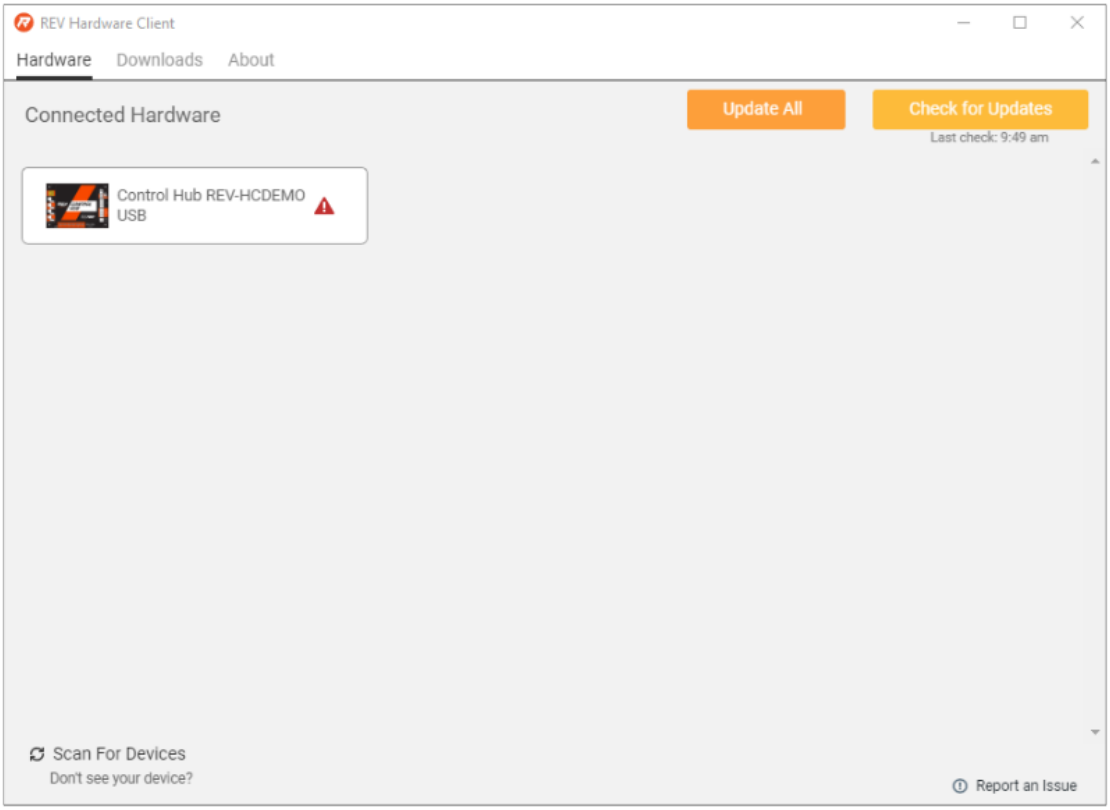
> The following procedure works with Control Hubs with the part number REV-31-1595. For support using the REV-31-1152 Control Hub v0 please reach out to REV support (support@revrobotics.com).

Using the REV Hardware Client

Steps	
Power on the Control Hub, by plugging the 12V Slim Battery (REV-31-1302) into the XT30 connector labeled “BATTERY” on the Control Hub.	
The Control Hub is ready to connect with a PC when the LED turns green. **Note:** With Robot Controller Application versions 5.5 and below the light will blink blue every ~5	

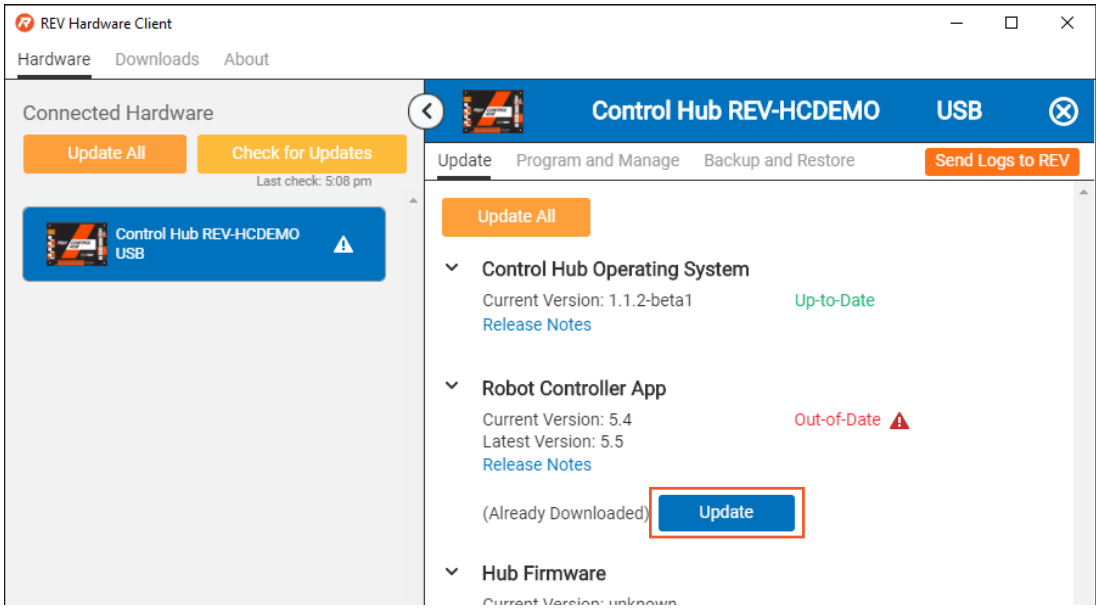
seconds. In 6.0 and above the LED is solid green.	
Plug the Control Hub into the PC using a USB-A to USB-C Cable (REV-11-1232)	

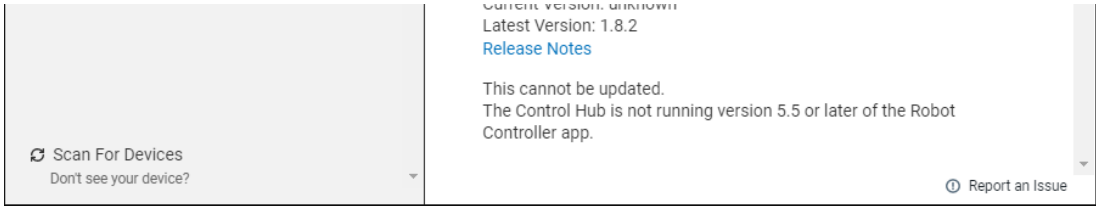
Startup the REV Hardware Client. Once the hub is fully connected it will show up on the front page of the UI under the **Hardware Tab**. Select the Control Hub.



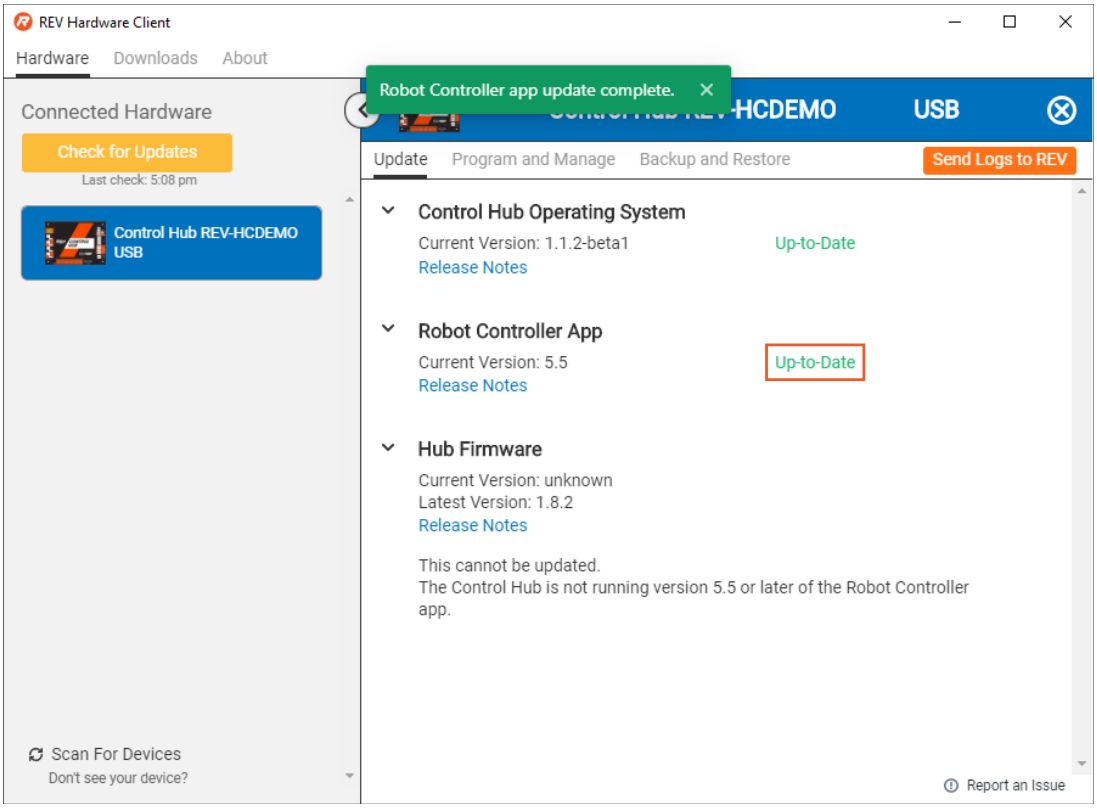
After selecting the Connected Hardware the Update tab will pop up. Under **Robot Controller App** select Download.

Once the app has downloaded, select Update.





When the Robot Controller Application update has completed a status message "Robot Controller app update complete." The status of the **Robot Controller App** will also change to "Up-to-Date."



Using the Robot Controller Console

Download the Latest Robot Controller APK - FtcRobotController-release v8.0

Updating the Robot Controller App

Click on the *FtcRobotController-release.apk* link in the repository to access the Robot Controller file.

Click on the *Download* button to download the Robot Controller app as an APK file to your computer.

On the *Manage* page, click on the *Select App* button to select the Robot Controller app that you would like upload to the Control Hub.

An *Update* button should appear if an APK file was successfully selected.

Click on the *Update* button to begin the update process.

During the update process, if the Control Hub detects that the digital signature of the APK that is being installed is different from the digital signature of the APK that is already installed, the Hub might prompt you to ask if it is OK to uninstall the current app and replace it with the new one.

This difference in digital signatures can occur, for example, if the previous version of the app was built and installed using Android Studio, but the newer app was downloaded from the GitHub repository.

Press *OK* to uninstall the old app and continue with the update process.

If the update process had to uninstall the previous version of the Robot Controller app, the network name and password for the Control Hub will be reset back to their factory values. If this happens, then you will need to reconnect your computer to the Control Hub using the factory default values.

When the update process is complete and you have successfully reconnected your computer to the Control Hub's network, you should see an "installed successfully" message on the *Manage* web page.

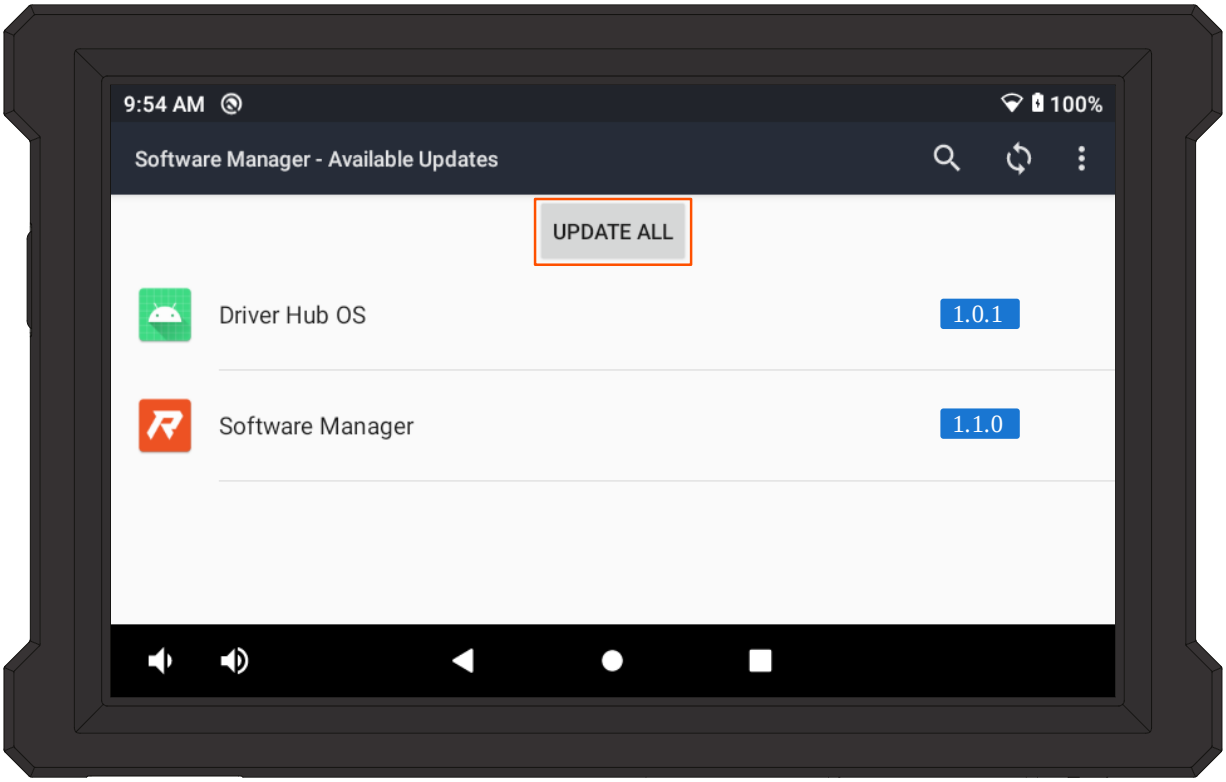
Updating the Driver Hub

The Driver Hub has two pieces of software that are field upgradable, the Driver Hub Operating System and the Driver Station Application. Both pieces of software are updatable either through the REV Hardware Client or directly on the Driver Hub with the Software Manager.

Driver Hub Software Manager

The Driver Hub has a Software Manager Application pre-installed for updating the Driver Hub. Open the application by pressing on the Software Manager icon. Select the Update All button to update all the software that requires updating.

> ⓘ Make sure the Driver Hub is connected to a Wi-Fi network with access to the internet to download and install the latest software.

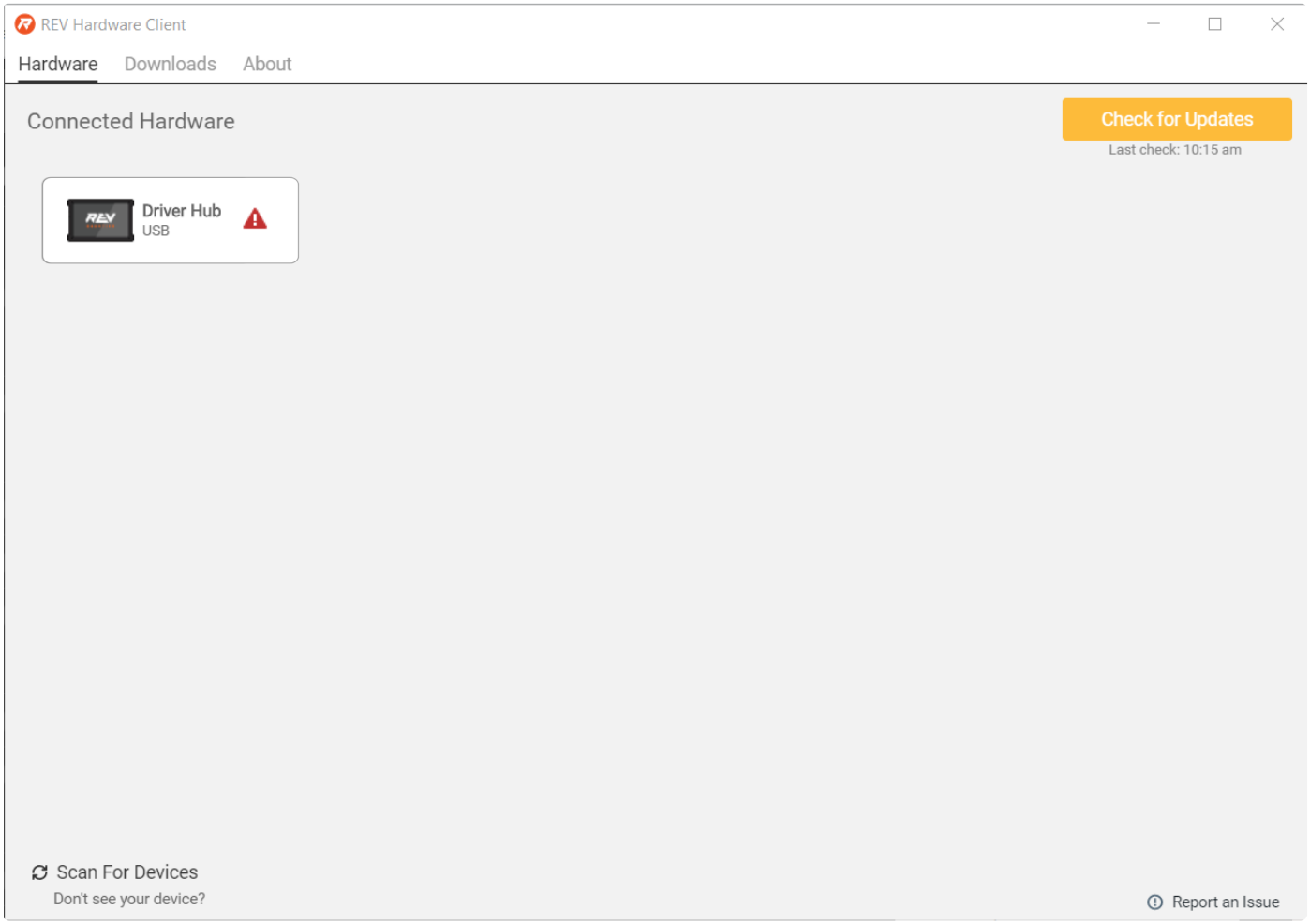


> ⓘ The updates can take several minutes to complete. Make sure the Driver Hub is charged or plug in the Driver Hub during the updating process.

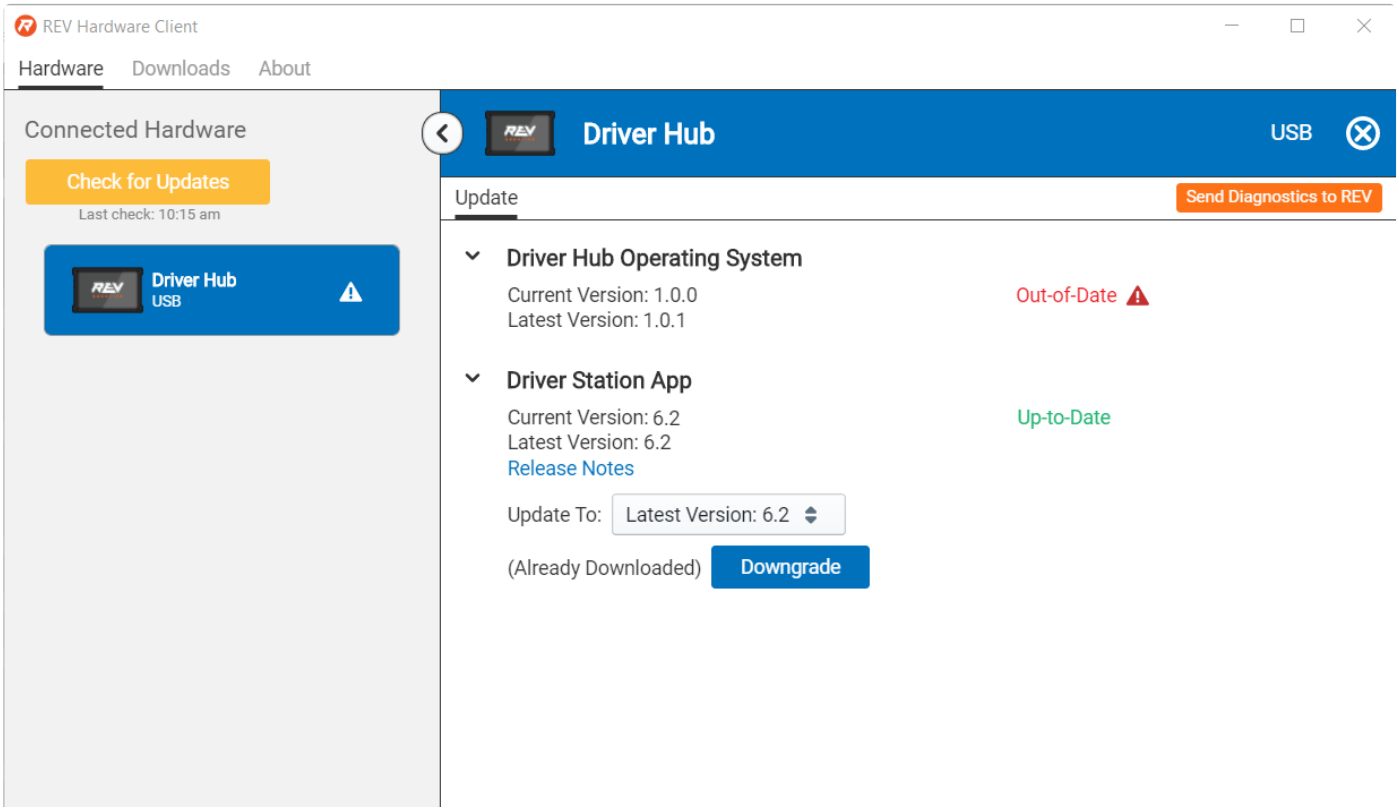
REV Hardware Client

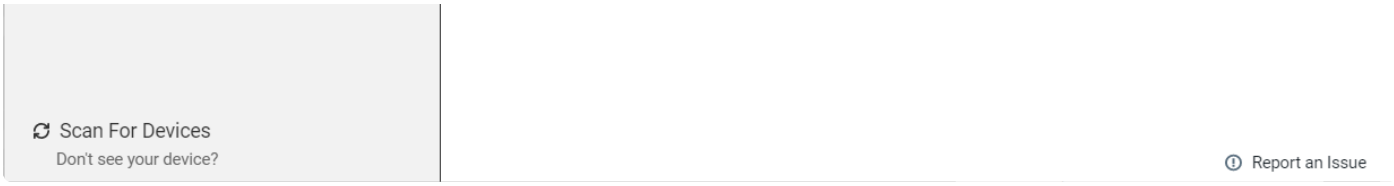
Startup the REV Hardware Client and connect the Driver Hub to the PC using the USB-A to USB-C cable. Once the Driver Hub is connected it will show up on the front page of the UI under the **Hardware Tab**. Select

the Driver Hub.



After selecting the Connected Hardware the Update tab will pop up. Any software the needs updating will have an Out-of-Date notification. Pressing the update button allows the REV Hardware Client to download the software update and install on the Driver Hub.





Once all the Out-of-Date notifications are cleared the Driver Hub is fully up to date.

Accessing Log Files

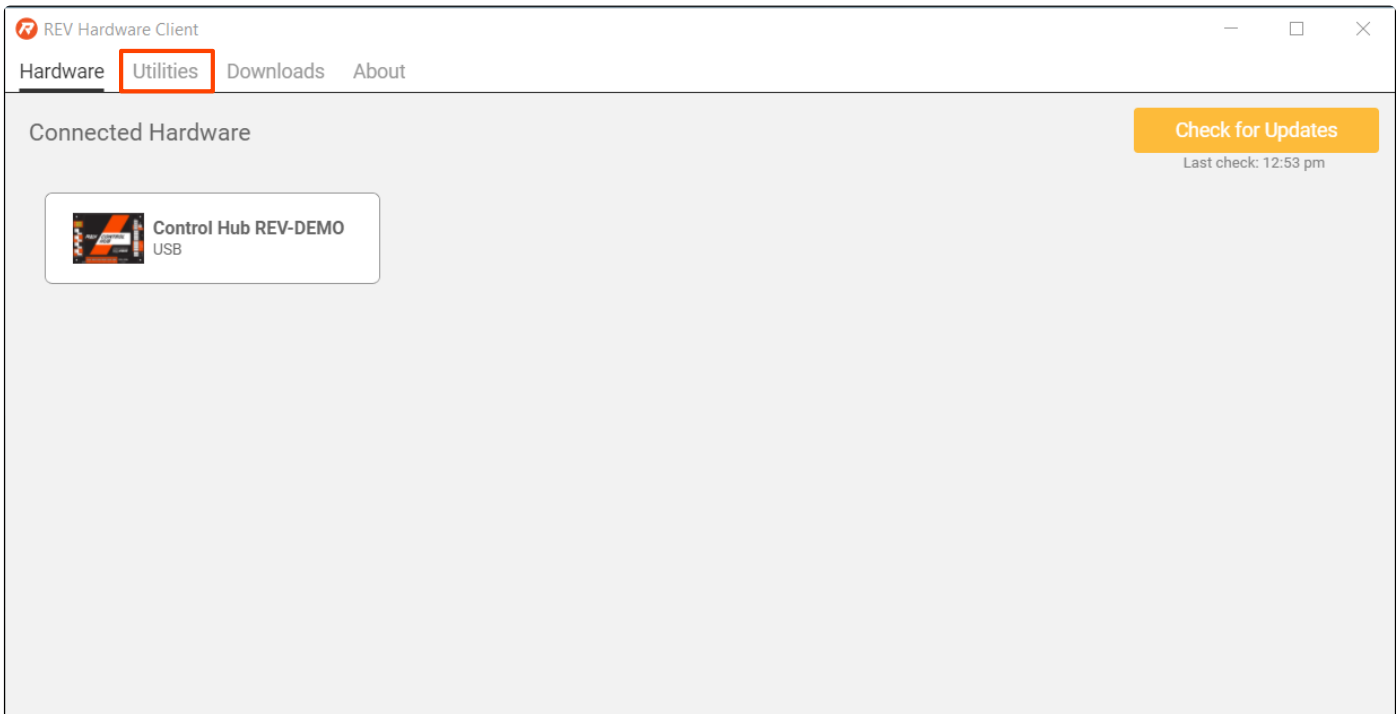
When troubleshooting problems with the REV Control System log files provide indicators of what the status of the Control Hub or Expansion Hub were during an event. Often the first log that is considered is the Robot Controller log, as they are relatively easy to decipher and can be pulled from the Control Hub or Robot Controller. While working through the troubleshooting process looking through the XML files, Wi-Fi Log, or Updater Logs in addition to the Robot Controller logs help to paint a full picture.

There are a few ways to access the log files depending on if you are looking to troubleshoot or downloading the log files for REV support to help.

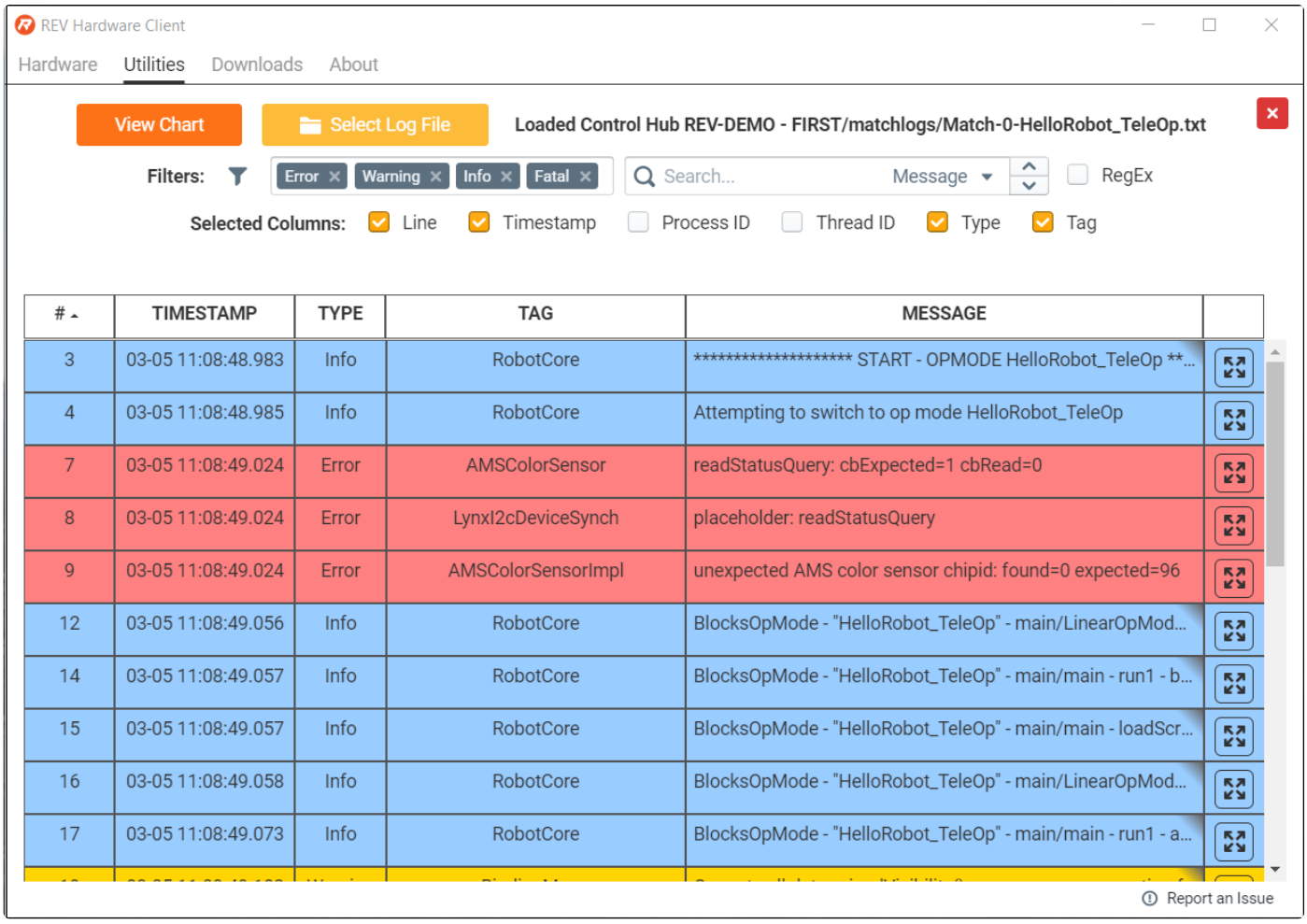
Log Viewer - REV Hardware Client

The REV Hardware Client has a Log Viewer that makes it easier to parse overall log files. Through a series of filters, tags, and a search function makes it easy to see what is happening on the Control Hub or Driver Hub during any opmode run.

To access the Log Viewer, head to the Utilities Tab.



From there you can select and open log files for connected devices or for ones downloaded onto the computer.



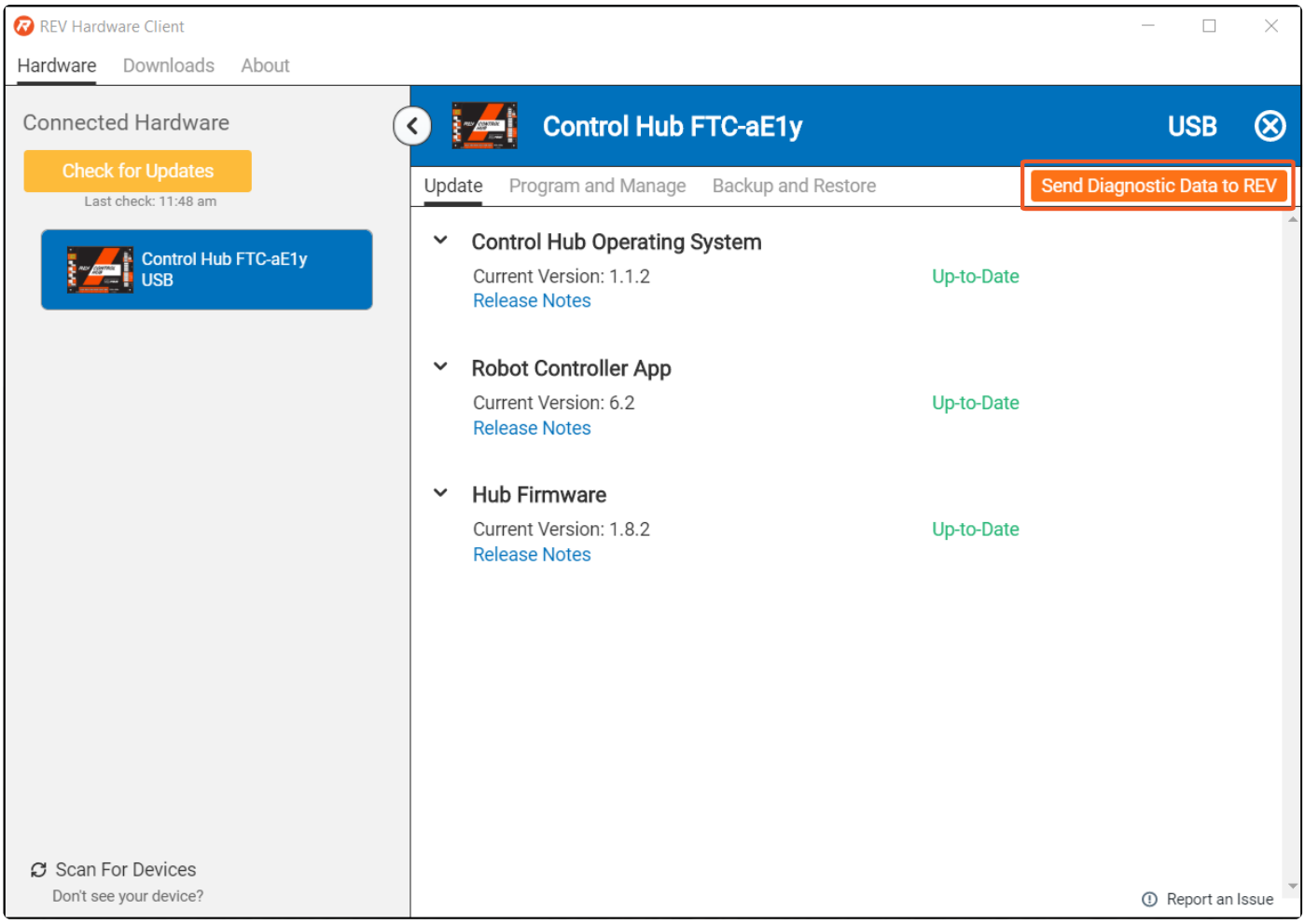
> For more information on the Log Viewer check out the REV Hardware Client User's Manual.

Downloading Log Files

When sending logs to REV Support use the REV Hardware Client. The Client will zip all relevant log files, collect some additional information from a form, and then send them to REV for diagnosis. When running through the troubleshooting process at an event, physically connecting to a Control Hub (REV-31-1595) and using the file search of the computer allows access to the files. Alternatively connecting to the Robot Controller Console allows downloading the logs through the manage tab.

REV Hardware Client

1. Provide 12v Power to the Control Hub.
2. Plug the USB-C Cable into the top board of the Control Hub and into a PC with the REV Hardware Client installed.
3. Select the Control Hub from the Connect Hardware.
4. Click the "Send Diagnostics to REV" Button



There is a short form to fill out with additional information to help REV Support troubleshoot the issue.

File Search

Using a PC

Mac computers do not support MTP natively, the protocol used to browse files on Android devices. You need to use the Android File Transfer app: https://www.android.com/filetransfer/

Windows devices will operate without the need for an additional application.

1.Provide 12v Power to the Control Hub.
2.Plug the USB-C Cable into the top board of the Control Hub and into a PC
3.Navigate to This PC\Control Hub v1.0\Internal shared storage. Robot Controller, Wi-Fi, and Updater logs can be found on this level of the file hierarchy.

> ⓘ The logs are all text files that can either be open via Notepad++ and looked over or sent to REV Support via an email to be further troubleshot.

4.While in the This PC\Control Hub v1.0\Internal shared storage location navigate to a folder called "FIRST." The folder should have XML files with a naming convention that mirrors the names of the robot configuration.

Using a Mac

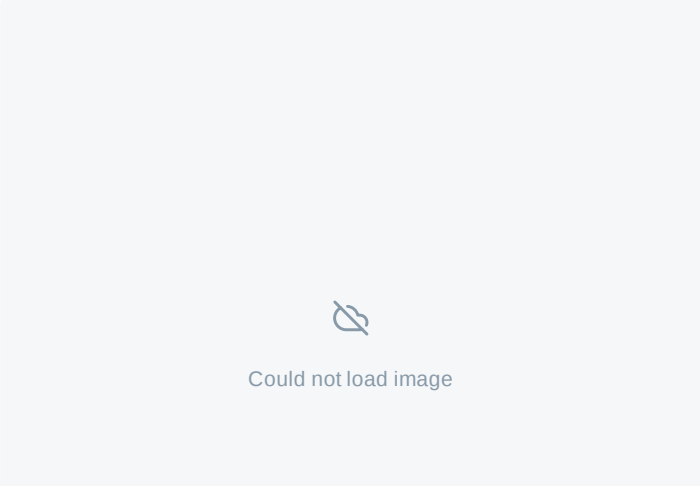
1. Download the Android File Transfer App on your MAC
2. Open Android File Transfer.dmg
3. Drag Android File Transfer to Applications
4. Use the USB-C to USB-A cable that came with your Control Hub (or other relevant Android Device)
5. Double click Android File Transfer
6. Navigate to Control Hub v1.0\Internal shared storage. Robot Controller, Wi-Fi, and Updater logs can be found on this level of the file hierarchy.

> ⓘ The logs are all text files that can either be open via Notepad++ and looked over or sent to REV Support via an email to be further troubleshot.

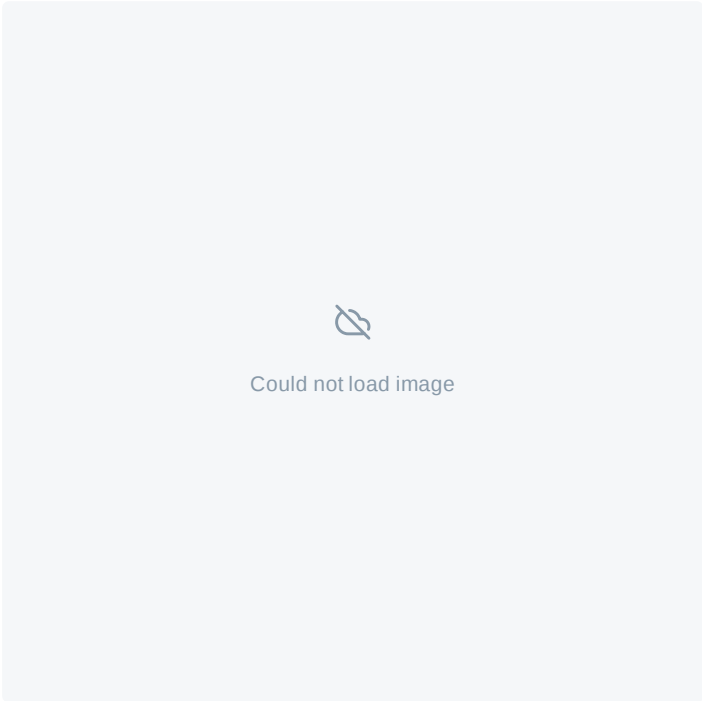
7. While in the Control Hub v1.0\Internal shared storage location navigate to a folder called "FIRST." The folder should have XML files with a naming convention that mirrors the names of the robot configuration.

Robot Controller Console

1.Open the Robot Controller Console
2. Select the **Manage** page
3. Press the Download Logs button



4. Check for the robotControllerLog.txt in the Downloads Directory of the Computer
5. Open the Logs via a text editor, like Notepad++, to view the contents of the log or send the logs to REV Support



Programming

Hello Robot - Introduction to Programming

Hello Robot - Choosing Your Path

In almost every programming class, the first lesson taught is some variation of the **Hello World** code. Hello World, often a one to two line segment of code, displays the line Hello World when the code is built and run. Though this code may seem like a very simple introduction to programming, it introduces several crucial concepts in programming. Hello World is the first lesson many students get in the **logic** of programming, as well as, language specific **syntax**. But, most importantly, the simplicity of Hello World allows it to be a **testing point** for the system used to execute the code.

Though it is possible to display Hello World or Hello Robot on an Android Device in the REV Control System, it doesn't serve quite the same purpose. In order to properly consider syntax, logic, and testing in

the REV Control System; consideration has to be paid to a multitude of system elements like actuators and sensors. For that reason the Hello World lesson has been edited into Hello Robot.

By the end of this guide users should understand how to configure their robot and test their robot mechanisms. The following outline walks through the flow and goals of this section. Choose the path that best fits your needs.

> ⓘ If you are new to programming or the REV Control System we recommend that you follow through the whole guide to learn how to properly utilize the system.

Section	Sub Section	Goals
Introduction		
	Programming Tools	There are three programming tools for the REV Control System. Learn about the benefits of each option and choose the best option to fit you needs. Section also includes instructions on how to access the option you choose.
	Op Modes	What are Op Modes? Learn about the different types of Op Modes in the REV Control System
Configuration		
	Importance of Configuration	What is Configuration and why should you configure before you begin to program?
	Configuring Common Hardware	Learn how to configure commonly used hardware like motors, servos, and sensors.
	Common Errors in Hardware Mapping	Understand and solve the common errors that occur when configuring and mapping hardware.
Test Bed: Introduction		
	Test Bed	Why creating a test bed of actuators and sensors can help with programming. This test be

		or something equivalent, will b
	Testing Basics	Learn why testing is one of the most important aspects of Software Development and how it differs from troubleshooting.
Test Bed: Blocks		
	Creating an Op Mode	Focuses on how to navigate th Blocks interface and create an op mode.
	Programming Essentials	Breaks down the structure and key elelments needed for an op mode, as well as some of the essential components of Block and programming logic.
	Programming Actuators	How to code servos and motors This section walks through the basic logic of coding actuators, controlling actuators with a gamepad, and using telemetry.
	Programming Sensors	How to code a digital device. The section focuses on the basic logic of coding a digital device, like a REV Touch Sensor.
Test Bed: OnBot Java		
	Creating an Op Mode	Focuses on how to navigate th OnBot Java interface and creat an op mode.
	Programming Essentials	Breaks down the structure and key elelments needed for an op mode, as well as some of the essential components of Java.
	Programming Actuators	How to code servos and motors This section walks through the basic logic of coding actuators, controlling actuators with a gamepad, and using telemetry.
	Programming Sensors	How to code a digital device. The section focuses on the basic logic of coding a digital

		device, like a REV Touch
Robot Control		Sensor.
	Create a Basic Robot	Introduces a potential robot to work with as well as the configuration file used in the following sections.
	Drivetrain Basics	Differences between differentia and omnidirectional drivetrains and their affect on teleoperated control types.
Robot Navigation: Blocks		
	Basics of Programming Drivetrains	What to consider when programming drivetrain motors and how to apply this to an arcade style teleoperated control.
	Elapsed Time	Learn how to use the concept o elapsed time to create time controlled autonomous programs.
	Encoder Navigation	Learn how to use encoders to create more consistent autonomous pathing.
Robot Navigation: OnBot Java		
	Basics of Programming Drivetrains	What to consider when programming drivetrain motors and how to apply this to an arcade style teleoperated control.
	Elapsed Time	Learn how to use the concept o elapsed time to create time controlled autonomous programs.
	Encoder Navigation	Learn how to use encoders to create more consistent autonomous pathing.
Arm Control: Blocks		
	Basics of Programming an Arm	Introduction to coding an arm fo teleoperated control and working with a limit switch

	Programming an Arm to a Position	Using motor encoders to move an arm to a specific position, such as from 45 degrees to 90 degrees.
	Using Limits to Control Range of Motion	Working with the basics of arm control, motor encoder, and lim switches to control the range o motion for an arm.
Arm Control: OnBot Java		
	Basics of Programming an Arm	Introduction to coding an arm f teleoperated control and working with a limit switch
	Programming an Arm to a Position	Using motor encoders to move an arm to a specific position, such as from 45 degrees to 90 degrees.
	Using Limits to Control Range of Motion	Working with the basics of arm control, motor encoder, and lim switches to control the range o motion for an arm.

Programming Tools

Choosing the appropriate programming tool or language is one of the most crucial decisions a user can make. In the REV Control System there are three programming tools to choose from: Blocks, OnBot Java, and Android Studio. Each tool comes with different benefits and difficulty levels.

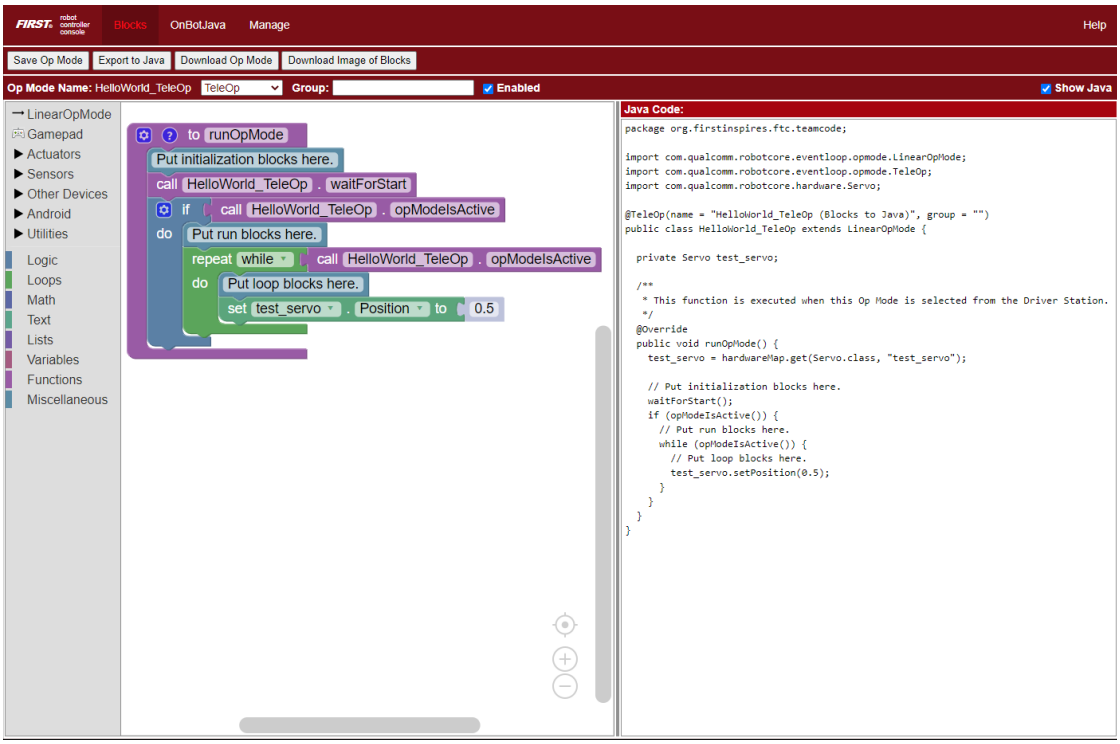
Basic	Intermediate	Advanced
Blocks	Onbot Java	Android Studio

The programming tools for the Software Development Kit (SDK) were chosen as a means of giving users the ability to choose among alternatives, but also as a means of allowing users to naturally move from basic to advanced programming. A user can reasonably start with Blocks and build their way up to Android Studio. In fact, this is often our suggestion to rookie users. Android Studio is a very powerful tool but, as you are learning the basics of programming, has the potential to be a hindrance rather than a help.

Review the following sections to learn about each programming tool and the benefits that come with it. Once you are done reviewing make an educated choice about what tool will work best for you!

Blocks

The Blocks Programming Tool is a visual, programming tool that lets programmers uses a web browser to create, edit and save their op modes. Blocks, like other scratch based programming tools, is a collection of preset code snippets that users can drag-and-drop into the appropriate code line.



Blocks was created to cater to users who have little to no experience programming. Unlike OnBot Java or Android Studio, Blocks works to insulate and protect users from the complexities of the SDK. The Blocks interface accomplishes this by hiding, or abstracting, some of the more complex overhead the system requires, like calls to specific libraries or classes. The code snippets make those connections and assumptions for the user.

One of the other major benefits of Blocks are the built-in features that allow users to naturally transition from little to no programming knowledge to a basic understanding of Java. Blocks teaches users the logic of programming, while protecting them from syntax mistakes. As users gain more confidence and ability they can use the "Show Java" option. "Show Java" allows users to see the Java syntax that corresponds with each Block that is added to the code.

Summarization of Benefits
Hides complexities from the user allowing them to focus on learning the logic
Has an option to Show Java which allows users to see what the corresponding syntax would be in Java
Web-based interface - accessible on most devices
Saves directly to the robot
Just as powerful as OnBot Java

Accessing Blocks

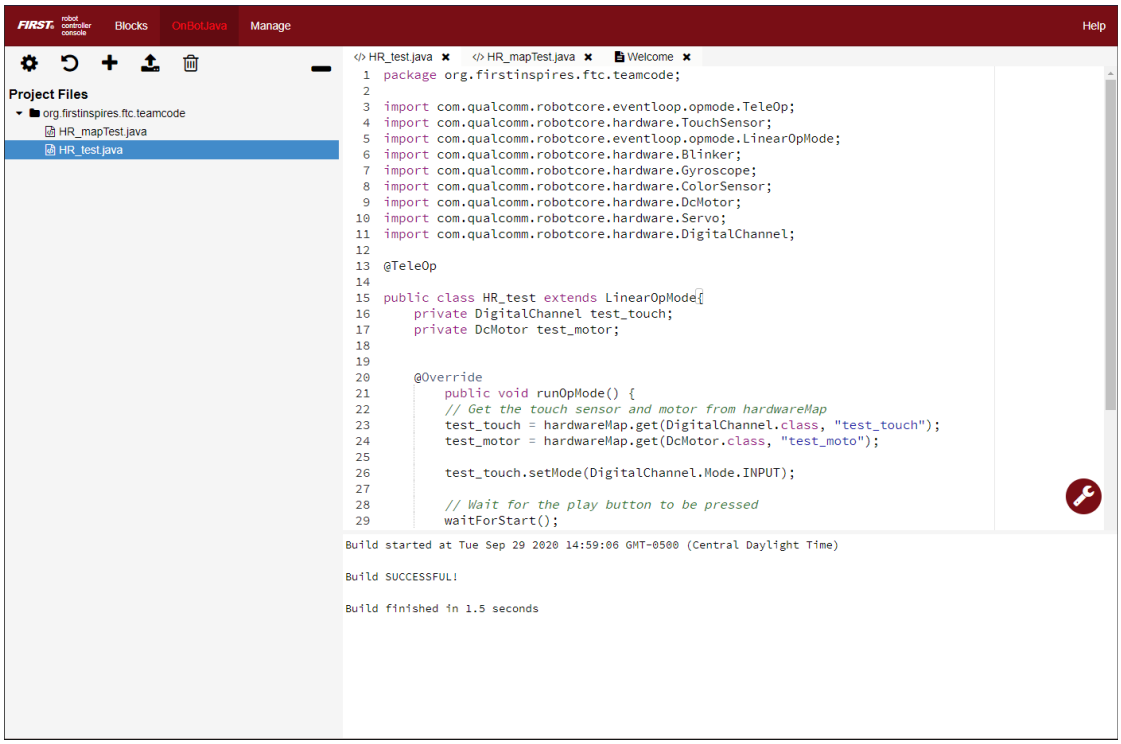
> ⚠ This section assumes that you have already gone through the steps of setting up your REV Control System. For more information on how to setup your control system check out the Getting Started with the Control Hub guide.
>
> This section also assumes that you have a JavaScript enabled web browser.

1. Go to Wi-Fi Settings, on a Windows 10 Computer, by clicking on the Wi-Fi symbol.
2. Once the list of available Wi-Fi networks in the vicinity is displayed select the network that matches the name of your Wi-Fi access point.
3. Enter the password that you set when setting up the Control System.
4. Once connected, open a JavaScript enabled browser (FIRST recommends Google Chrome).
5. Go to IP Address **http://192.168.43.1:8080**
6. At the top of the Robot Controller Console Page, there should be 3 menu options: Blocks, OnBot Java, and Manage. Choose Blocks.

> ⓘ Passwords are case sensitive. If you do not remember your password, use the Hardware Client to check the Program and Manage section of the Robot Controller Console

OnBot Java

A text-based programming tool that lets programmers use a web browser to create, edit and save their Java op modes.



OnBot Java is great for programmers with basic to advanced Java skills who would like to write text-based op modes. OnBot Java shares some of insulative properties of Blocks, but gives users access to the more

complicated elements of the SDK libraries.For instance, OnBot requires users to make calls to classes like the hardwareMap, which are hidden within the Blocks code snippets.

OnBot Java shares a web-based interface with the Blocks Programming tool. The web-based model is easy to access on most devices to make code change and reduces the need to have one set device for code changes.

Summarization of Benefits
Access to more complicated library classes for more advance programming
Reduces coding issues by hiding complex classes from the user
Allows users to learn Java in simplified interface
Web-based interface - accessible on most devices
Saves directly to the robot

Accessing OnBot Java

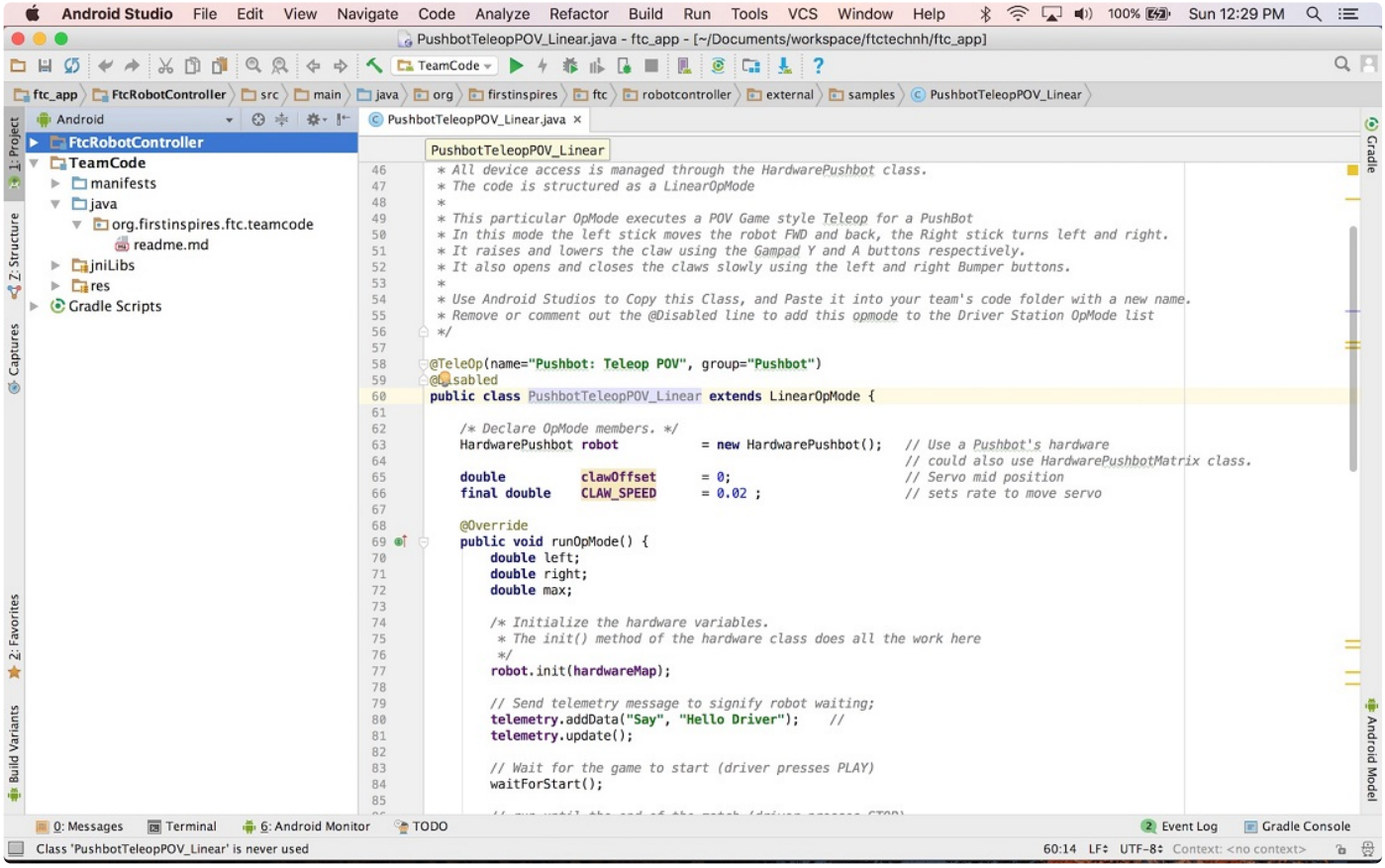
> ⚠ This section assumes you have already gone through the steps of setting up your REV Control System. For more information on how to setup your control system check out the Getting Started or Managing the Control System sections of the Control System Guide.
>
> This section also assumes that you have a JavaScript enabled web browser.

1. Go to Wi-Fi Settings, on a Windows 10 Computer, by clicking on the Wi-Fi symbol.
2. Once the list of available Wi-Fi networks in the vicinity is displayed select the network that matches the name of your Wi-Fi access point.
3. Enter the password that you set when setting up the Control System.
4. Once connected, open a JavaScript enabled browser (FIRST recommends Google Chrome).
5. Go to IP Address **http://192.168.43.1:8080**
6. At the top of the Robot Controller Console Page, there should be 3 menu options: Blocks, OnBot Java, and Manage. Choose OnBot Java

> ⓘ Passwords are case sensitive. If you do not remember your password, use the Hardware Client to check the Program and Manage section of the Robot Controller Console

Android Studio

An advanced integrated development environment for creating Android apps. This tool is the same tool that professional Android app developers use. Android Studio is only recommended for advanced users who have extensive Java programming experience.



Android Studio allows programmer with an advanced understanding of Java a more powerful development environment to work in. It offers enhanced editing and debugging features not available with OnBot Java or Blocks. It also allows programmers the ability to work with 3rd Party libraries not included within the SDK. However, Android Studio is not a web-based software and will need a dedicated laptop to run on.

Summarization of Benefits
Access to more complicated library classes for more advance programming
Enhanced editing and debugging features
Enables access to 3rd Party Libraries

Accessing Android Studio

To learn about how to properly download and work with Android Studio please visit the FTC Wiki.

> ⚠ FIRST Global does not have support for Android Studio.

Op Modes

Op modes (or operational modes) are computer programs that are used to customize or specify the

behavior of a robot. The Robot Controller, either the Control Hub (REV-31-1595) or an Android device paired with an Expansion Hub (REV-31-1153), stores and executes op modes. The Driver Station allows users to select from any of the op modes stored on the Robot Controller and initialize, start, or stop the op modes.

In the SDK there are two types of op modes: **autonomous** and **teleoperation**. Both types of op modes have initialization, start, and stop features on the Driver Station phone. Each feature corresponds with different types of code segments that will be discussed in detail in the programming tool specific Test Bed sections of this document.

The main difference between autonomous and teleoperation op modes is how they show up in the Driver Station application. Autonomous op modes show up in a drop down menu on the left side of the Driver Station application. The Driver Station assigns a 30 second timer to autonomous op modes. If the autonomous op mode is not manually stopped prior to the end of the 30 seconds the Driver Station will automatically stop the code. TeleOp op modes will appear in a drop down menu on the right side of the Driver Station application. These op modes will run until they are manually stopped.

It is also worth noting that in the SDK op modes can be **linear op modes** or **iterative op modes**. This guide focuses on Linear op modes, which execute code lines in a sequential order. In order to repeatedly call actions within in a linear op mode, a loop function must be used. This topic will be discussed in further detail as you follow along this guide.

Hello Robot - Configuration

Configuration is one of the most commonly misunderstood, or forgotten, steps required for programming a robot. This section sets out to explain the importance of configuration and common misconceptions of configuration by answering the following questions:

1. What is configuration?
2. How do you configure hardware elements?
3. What are common issues that are caused by a problem with the configuration file?

The Importance of Configuration

While every REV Control Hub is the same, the robots being controlled by the Control Hub are not. Each Control Hub has the same number of motor ports, servo ports, digital ports, and the like, but how each user utilizes these ports varies from system to system. For instance, a Color Sensor V3 may be plugged in to I2C Bus 1 on one users Hub, but another user might use the same bus to host a 2m Distance Sensor.

The Control Hub knows that there is an I2C device attached to the port. But it doesn't naturally have the information needed to translate that information to an Op Mode or tell the op mode which drivers need to be accessed in order to use this sensor. A user needs to provide additional information, so that the internal

software in the Hub can take information from the Op Mode and apply it to a corresponding external hardware port and vice versa. This process is known as hardware mapping. Hardware mapping is a two step process that includes: the creation of a readable file known as a **configuration file** and calls to the **hardware map** within an Op Mode

The Configuration File

The configuration file is a readable file created by the user through the Driver Station Application. When creating a configuration file users are required to assign each device to a port, select the type of device it is from options provided by the SDK, and give it a **unique** name.

> ⓘ In programming its important to distinguish between variables, by giving each variable a different name.

Once a configuration file is saved or activated the robot will restart. This restart is so the SDK can read the file, determine what devices are present, and add the devices to the hardwareMap class.

The Hardware Map

On the user-created op mode side of the fence is the hardwareMap class. This class is where the information created in the configuration is available for use in Blocks, OnBot Java, or Android Studio code.

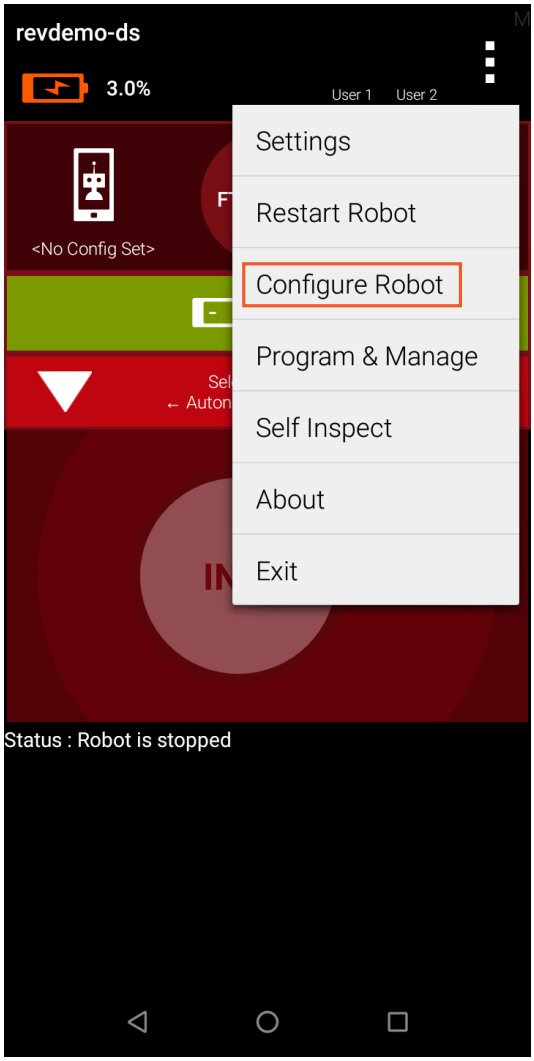
The level of access or interaction a user has with the hardwareMap class depends on which programming tool they are using. Since Blocks is a collection of predetermined code snippets, it creates references to the hardwareMap whenever a variable code snippet, corresponding to an external hardware, is first referenced. However, with Onbot Java and Android Studio the reference to the hardwareMap requires that a variable be created and assigned to an external hardware unit within the hardwareMap

> ⓘ Information on referencing the hardwareMap class in Java will be further explained in the Test Bed - OnBot Java section.

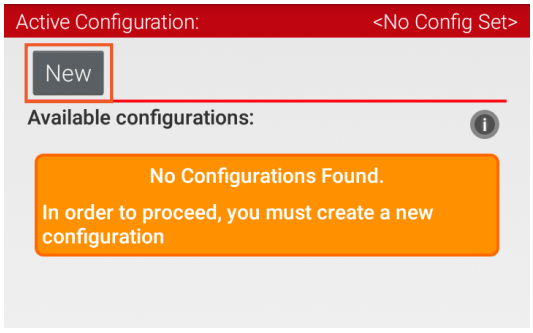
Configuring Common Hardware Devices

Accessing the Configuration Utility

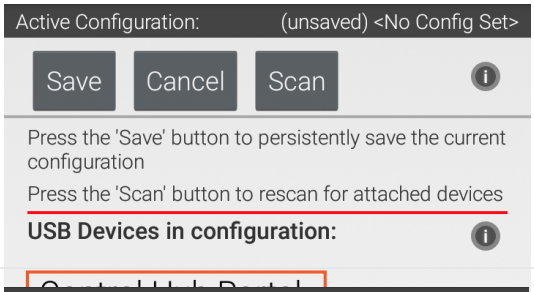
Select the menu in the stop right corner of the Driver Station. Then select **Configure Robot**.



In the **Available configurations** page, select New.

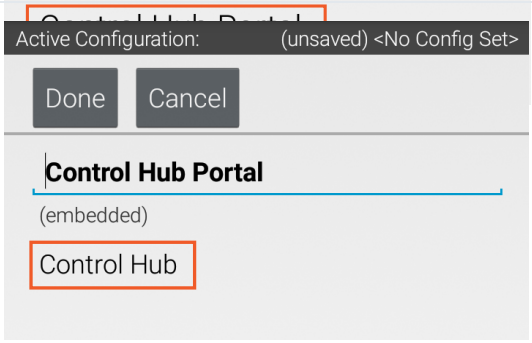


In the **USB Devices in configuration** page select the **Control Hub Portal**.
Note: If you have an Expansion Hub it will appear as an **Expansion Hub Portal**.



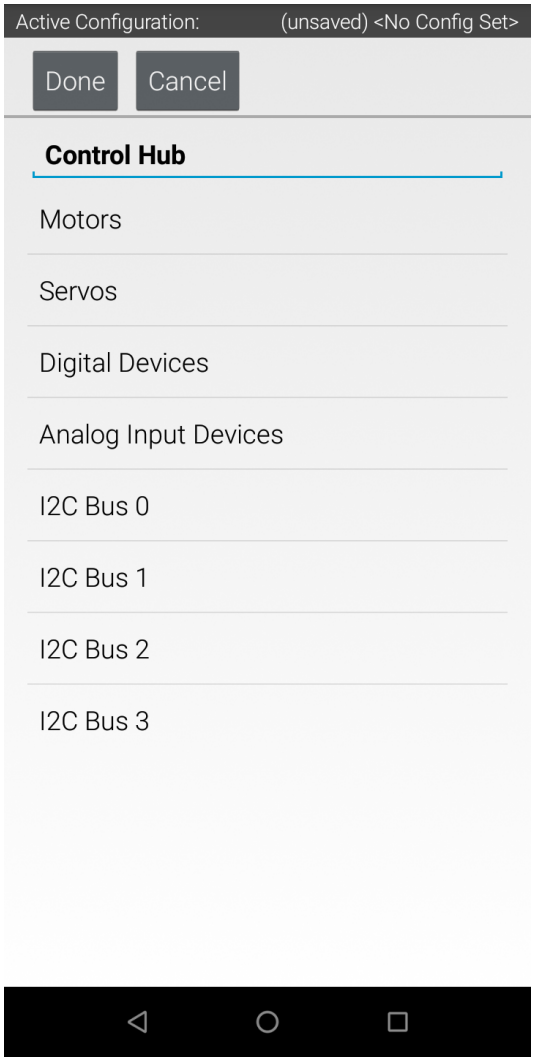
Within the Hub Portal select the device you want to configure. In this use case, select the Control Hub.

Note: if you have an Expansion Hub connected to a Control Hub, the Expansion Hub will also appear as a configurable device in the portal.



This will bring you to the page shown in the image. From here you can configure motors, servos and sensors that you are using. Follow through the rest of the guide to figure out how to configure devices that will be used in the Test Bed section.

Note: The way that Digital and Analog devices are configured versus how I2C devices are configure differ significantly. This is because each physical I2C port is a different bus that can host multiple different sensors. For more information on the different types of sensors check out the sensors section.



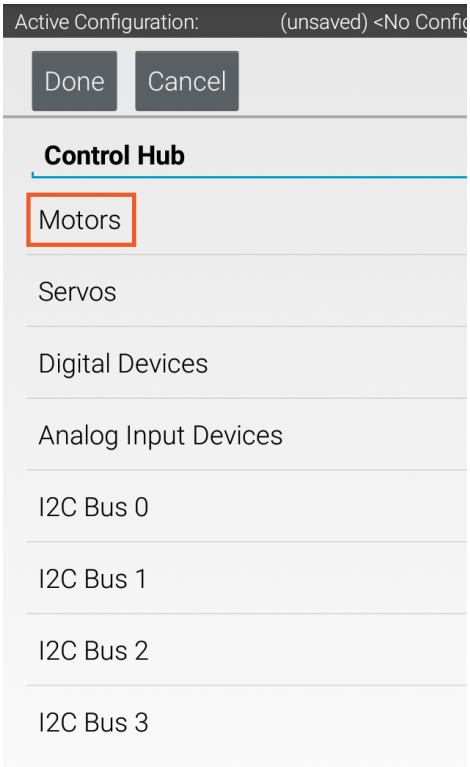
Configuring Hardware

The following section will show how to configure components that will be used in the Test Bed. The hardware type and names have been chosen in consideration for the Hello World lesson plan. Users should heed notes within the steps to consider when creating configuration files for other instances.

Motor

Configuring a Motor

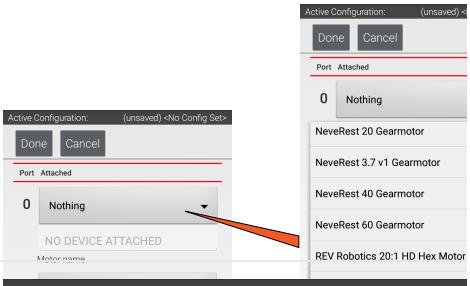
Select **Motors.**



The Motor page will allow you to configure all four motor ports on the Hub. On **Port 0** open the drop

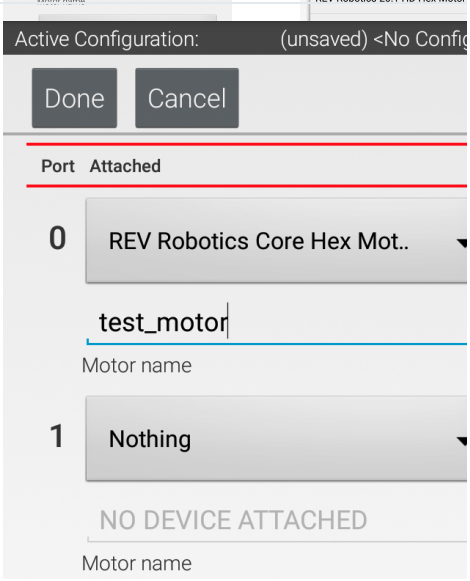
down menu and select **REV Robotics Core Hex Motor**.

Note: In your configuration file you should configure the motor ports to the type of motor you are using.



Name the motor **test_motor**. Select **done.**

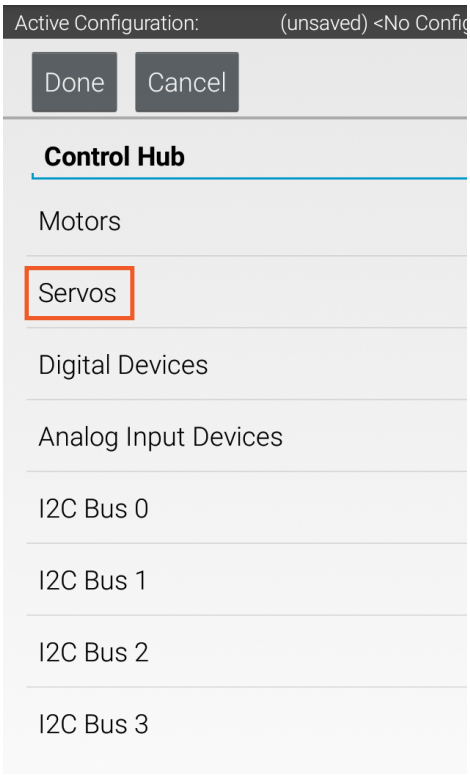
Note: remember when naming hardware in the configuration file that the REV Control System is **Case Sensitive.**



Servo

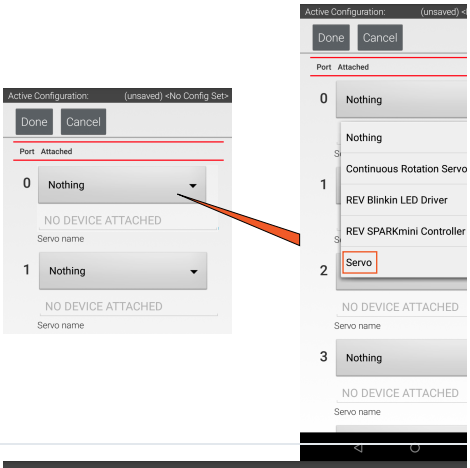
Configuring a Servo

Select **Servos.**



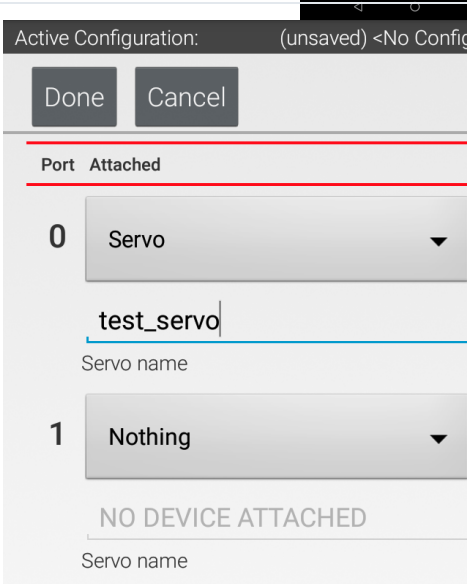
The Servo page will allow you to configure all six servo ports on the Hub. On **Port 0** open the drop down menu and select **Servo**.

Note: REV Servos can be configured as a **Servo** or a **Continuous Rotation Servo**. The type of device a servo is configured as should correspond with the mode the sensor is in. For more information on Sensor modes visit the Sensor section.



Name the servo **test_servo**. Select **done.**

Note: remember when naming hardware in the configuration file that the REV Control System is **Case Sensitive.**



Digital Device

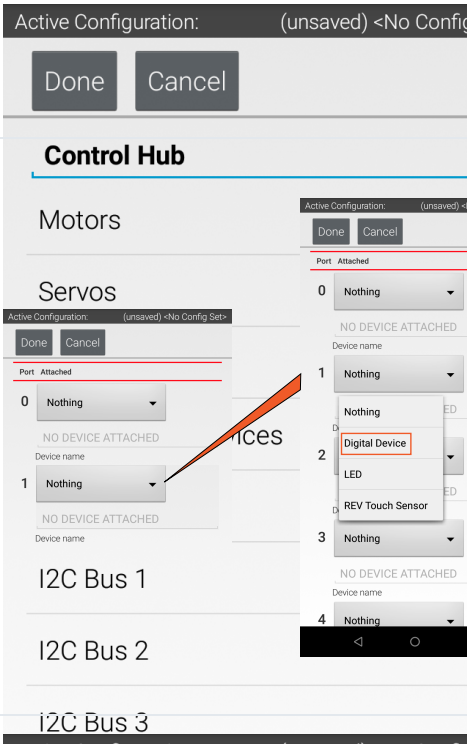
Configuring a Digital Device

Select **Digital Devices**.

The Digital Devices page will allow you to configure all eight digital ports on the Hub. On **Port 1** open the drop down menu and select **Digital Device** .

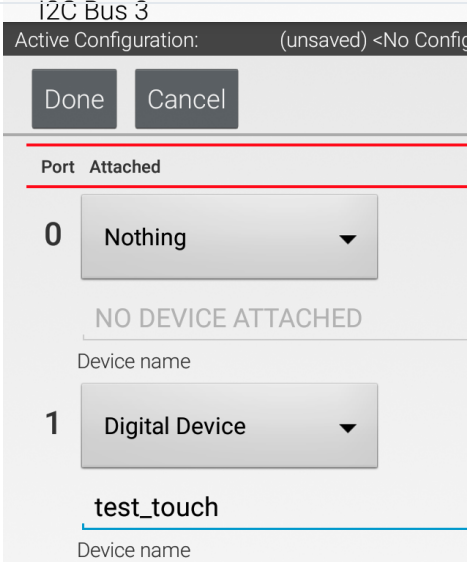
Note: Touch Sensors must always be configured on odd number ports. Check out the Digital Sensor section for more information.

Note: Touch Sensors can be configured as a REV Touch Sensor or a Digital Device. In the FTC SDK the type of device it is configured as changes the classes and methods that can be used.



Name the motor **test_touch**. Select **done.**

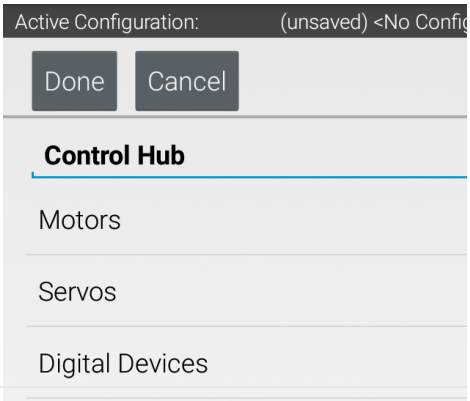
Note: remember when naming hardware in the configuration file that the REV Control System is **Case Sensitive.**



I2C Device

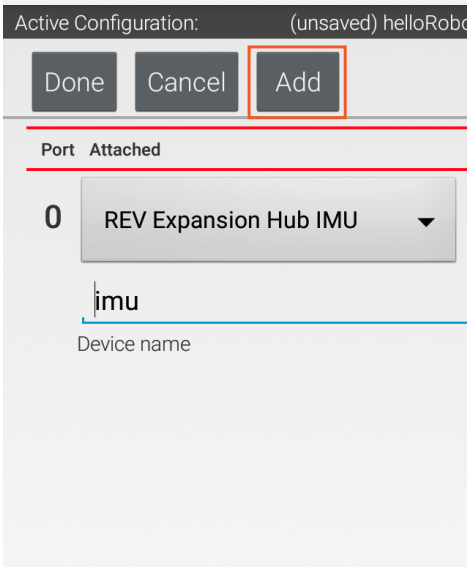
Configuring an I2C Device

Select **I2C Bus 0**.



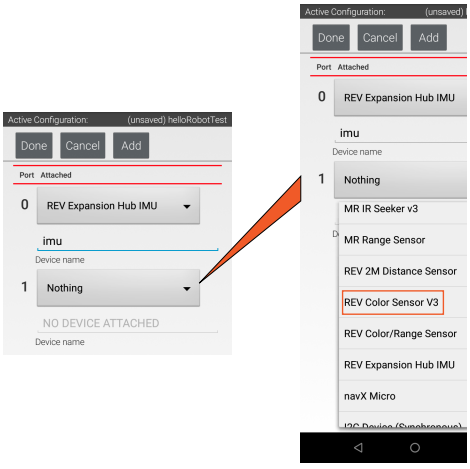
Select Add.

Note: Each I2C Bus can host more than one I2C sensor as long as the I2C addresses do not conflict. Bus 0 will always host the internal IMU. For more information on I2C sensors visit the I2C section.



On **Port 1**, which was created in the previous step**,** open the drop down menu and select **REV Color Sensor V3.**

Note: If you are using Color Sensors V1 or V2 select **REV Color/Range Sensor.** For more information on configuring with the REV Color Sensors visit the Color Sensor Datasheets.

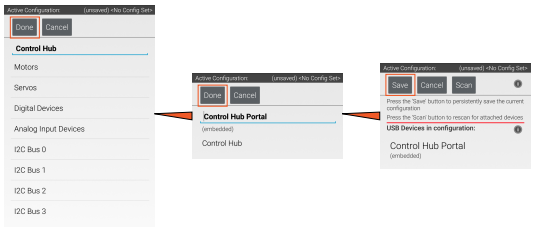


Name the motor **test_color**. Select **done.**

Note: remember when naming hardware in the configuration file, that the REV Control System is **Case Sensitive.**

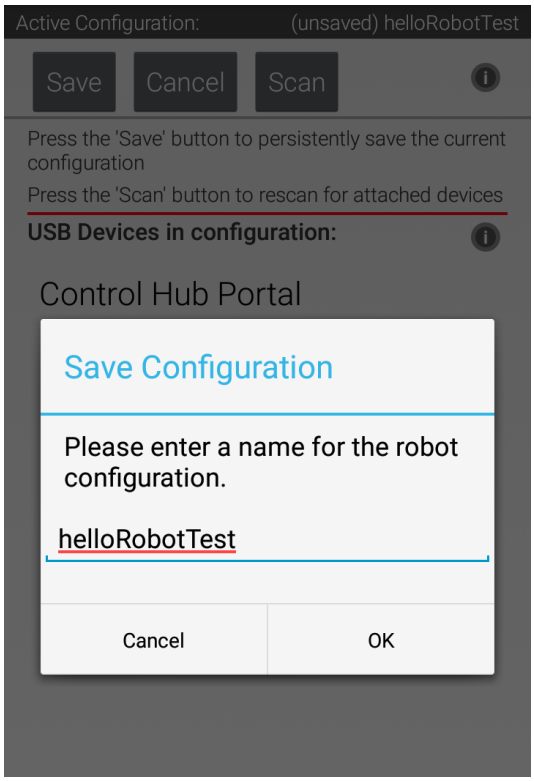
Saving the Configuration File

Hit **Done** twice until you reach the USB Devices in configuration page**.** On the USB Devices in configuration page hit **Save.**

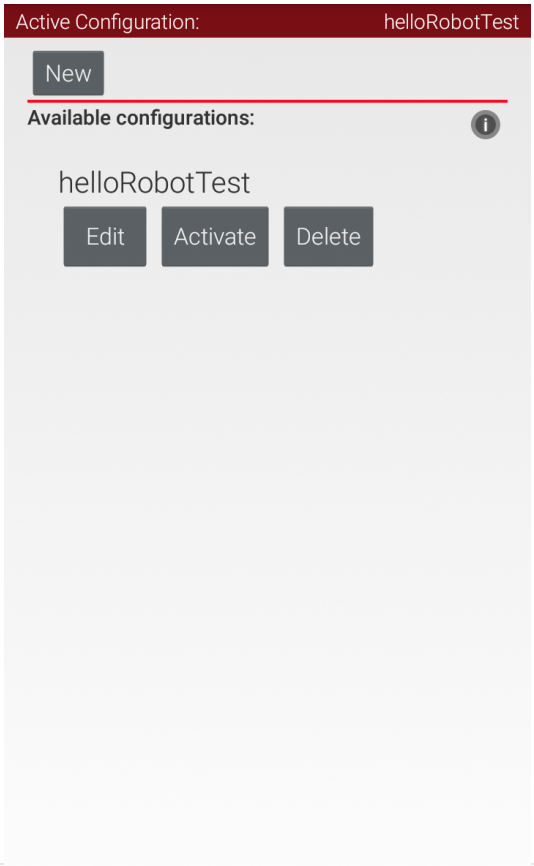


Name the configuration **helloRobotTest** and then select **Ok.**

Note: The FTC SDK does not force you to abide by a naming convention for but it is common to name configurations in lowerCamelCase.



Press back to activate the saved configuration. Your Robot Controller will restart once you activate a new configuration.



Common Errors in Hardware Mapping

Within the programming and software world errors come in many different forms and types. When hardware mapping there are two major errors that you may run into. Both errors fall into common categories of software errors:

- Interface Errors - are errors between how an interface should work and how it actually behaves
- Runtime Errors - are errors that occur when a program is being executed

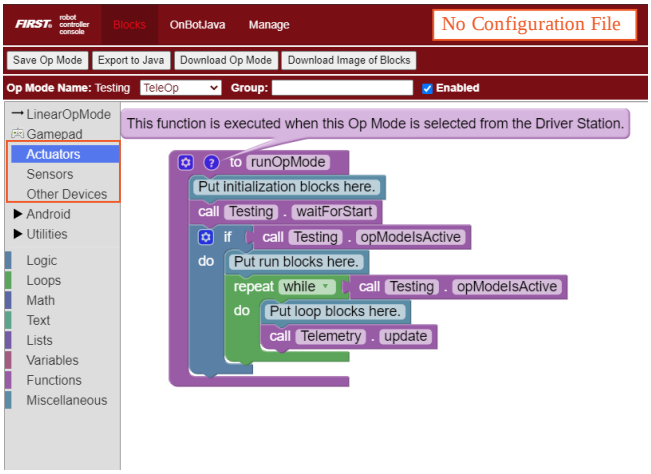
Interface Errors

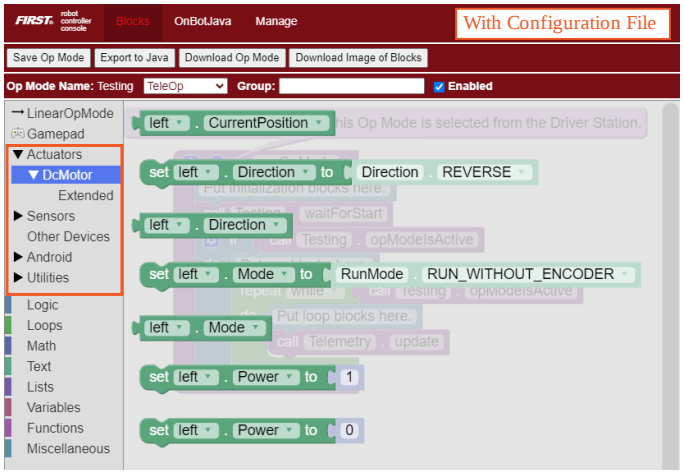
Interface errors occur in the SDK when the parameters of the SDK interface are not met. In the hardware mapping process the most common interface error occurs with the Blocks Programming Tool. As mentioned in the Introduction to Programming section, Blocks hides complexities of the SDK library from the users. One way it does this is by automatically creating references to the hardwareMap when code snippets for an external hardware unit are used.

In order to automate the hardwareMap calls the Blocks interface reads the configuration file and creates hardware variables based off of the information it finds. For this reason it is important to create a configuration file before trying to code.

The image below shows two different interface versions of Blocks. In the version with no configuration file there are no drop down menus to access code snippets specific to actuators or sensors. In the version of the

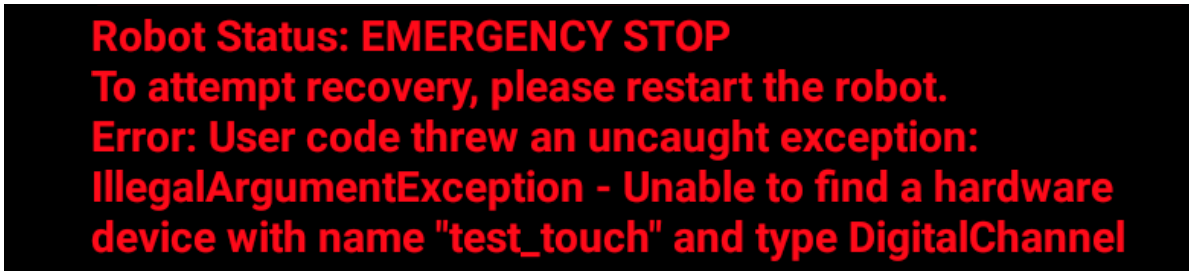
interface with a configuration file the drop down menus are present and the motor-specific code snippets are





Runtime Errors

Within the SDK runtime errors occur during initialization or run. One of the most common runtime errors within the REV Control System is exhibited in the image below.



This error is indicative of an inconsistency between how a hardware device is called within the code and how that compares against the name used in the configuration file. There are two different ways this error can occur.

The first occurrence of this error is when there is no configuration file found. This can mean that a configuration file has not been created, a file has been created but is not active, or the wrong file is being used. When any of these instances happen, the code is requesting a device name and type that the hardware map is unable to locate in the configuration file. The program stops on the first such device name it's unable to locate.

An incorrect reference to the hardwareMap can also cause this error to occur. Unlike Blocks, OnBot Java and Android Studio require that a programmer hard code the hardwareMap call. If the reference name in the call does not correspond with the name of the device in the configuration file (it is case sensitive) the code will build without failure but the runtime error will occur. Lets use the configuration file from the previous section as an example; where there is a touch sensor named `"test_touch"` and a motor name `"test_motor"` .

> ⓘ The quotations marks indicate that `"test_touch"` and `"test_motor"` represent a string

> variable that corresponds with the names of the devices in the configuration.

```
public class HR_test extends LinearOpMode{
    private DigitalChannel test_touch;
    private DcMotor test_motor;

    @Override
    public void runOpMode(){
        //get the touch sensor and motor from hardwareMap
        test_touch = hardwareMap.get(DigitalChannel.class, "test_touch");
        test_motor = hardwareMap.get(DcMotor.class, "test_moto");

    }
}
```

Notice in the example lines of code that the `hardwareMap.get()` for test_motor is written as "test_moto" rather than "test_motor." When the code is building, there is no immediate check that the name requested is in the hardwareMap. This check is done when code is run on the robot. When the communication to the hardwareMap is initiated it looks for "test_moto" and when it can not find it, it creates the runtime error referenced above.

Hello Robot - Test Bed

One of the most important steps in the engineering design process and the software development lifecycle is testing. When working with code, ensuring that it works without errors and works to the standard decided upon in the planning stage of the process is crucial. In order to ensure that the code is working as intended testing needs to be performed.

Before delving into the introduction to programming sections, Test Bed - Blocks or Test Bed - OnBot Java; its important to understand testing, the benefits of creating a test bed, the components needed for the next sections, and how to use gamepads. Follow through the rest of this section to learn more about testing!

Section	Goals of Section
Testing Basics	Learn why is one of the most important aspects of Software Development and how it differs from troubleshooting.
Test Bed	Why creating a test bed of actuators and sensors can help with programming. This test bed, or something equivalent, will be used in following sections.
Using Gamepads	Understanding the naming conventions for programming a gamepad.

> Keep in mind that this is the introduction to the basic programming guide. Test Best - Blocks and Test Bed - OnBot Java will walk you through the basics of programming with the REV Control System.

Testing Basics

The purpose of testing is to identify, isolate, and correct potential issues in a design before the design is put into use. Testing takes on different forms or provides different metrics for various intents in design. A mechanism, like a shooter for instance, might be tested to confirm that it is running reliably. During the planning phase of the design process you should create various performance, quality, and reliability metrics. When the design is built, or the program is written, these metrics will help you identify whether the mechanism meets the standards you expect it to. If the standards of operation are not met then the problem needs to be isolated.

In order to fix a problem in the design process, you must isolate the source of the issue. To understand how this works consider the following example:

> A team has recently purchased a Control Hub and a Core Hex Motor. They plug the Core Hex Motor into the Control Hub using the correct wiring, but when they go to run their code the motor doesn't move. What is the most likely reason for this failure:
>
> 1. The program is the issue
> 2. The motor is the issue
> 3. The wire connecting the motor to the Hub is the issue
> 4. The Hub is the issue.

Without more information there is not a good way to discover why the motor is not running. In order to narrow things down the different components have to be tested until the root of the issue is found. Common practice is to start with a code that is known to work, such as one of the sample codes in the SDK. If the motor still doesn't run the next thing the team should check is whether or not the wires are working as intended. One by one the team should go through and test, or troubleshoot, the different potential origins of the problem to see what is working and what isn't.

Once the source of an issue has been isolated, the issue needs to be corrected. The duration of the fix depends on the sources of the problem and how deep it runs. For instance, if an op mode doesn't work as intended the fix may be a simple change, like to the configuration file or the hardwareMap. A larger issue that requires a redesign, like a mechanism not meeting performance metrics, triggers a restart of the engineering design process.

Testing vs. Troubleshooting

Previously, testing was defined as the process of identifying, isolating, and correcting potential issues during

the design process. This differs from troubleshooting which is the process of identifying, isolating, and correcting issues of a mechanism that went through the testing process and worked as intended.
In the troubleshooting section the examples of a cars check engine light was used. In this example, the known indicator of a failure was the cars engine light. The check engine light informs the driver that something is wrong with the car but in order to find the cause of the issue troubleshooting and diagnostic steps must be performed. To maintain that comparison, testing is what the engineers of the car use to establish the metrics of expected engine performance. If those standards are not met then the check engine light turns on to warn the driver of the issue.

Test Bed

One of the fallbacks to **testing** code in a system of components, like the REV Control System, is that there is not a guarantee that all components are functioning as they should be. For instance, if a motor on the robot isn't working there are several potentials reasons for the failure. The motor, the motor port on the Control Hub, the wire connecting the motor to the port, and the code are all potential causes of motor failure.

If a failure occurs after the Robot is assembled it can be hard to go back and make changes, or **troubleshoot** without having to disassemble the robot. One of the ways to plan ahead for this circumstance is to create a test bed prior to creating a robot.

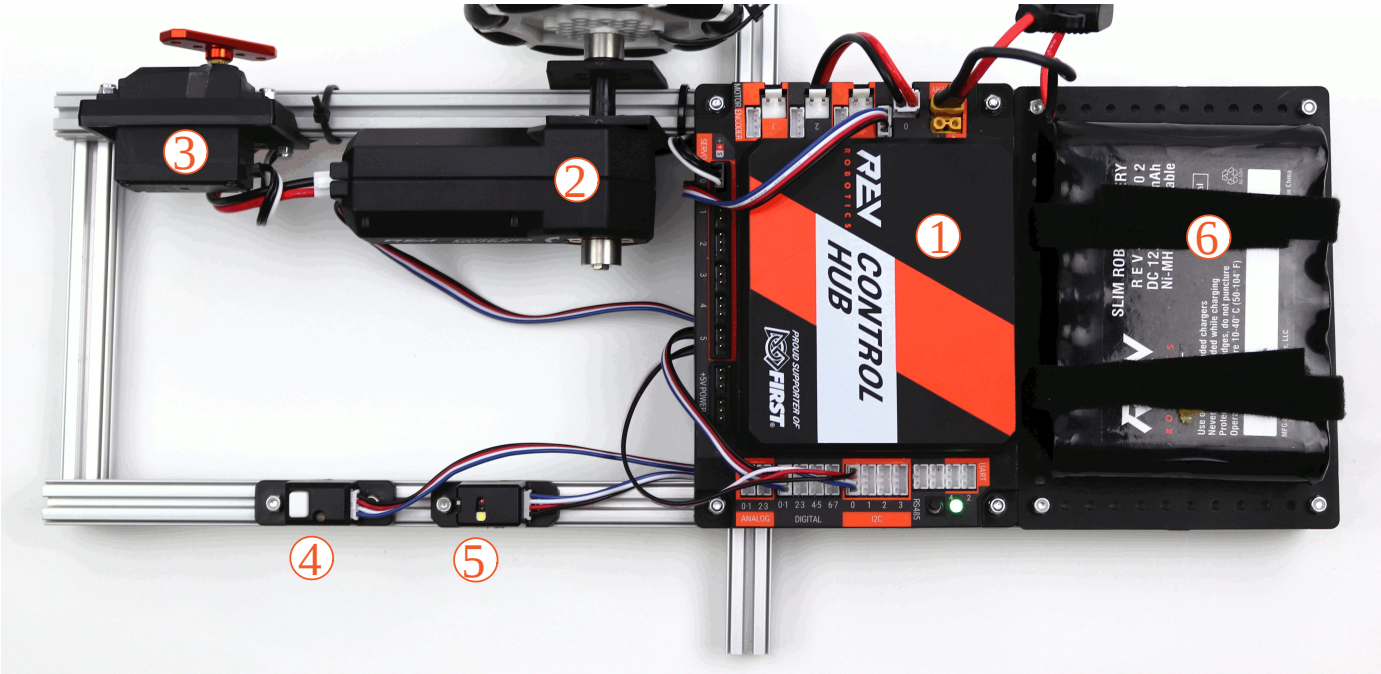
> When testing code do not assume that a failure is due to the mechanism rather than the code. Testing and troubleshooting, while being similar concepts, are fundamentally different. Checking the code or using a known code that works should always occur before troubleshooting components like actuators and sensors.

A **test bed** is a testing environment for hardware and software components, commonly used in the engineering world. Test bed applications includes a broad range of different equipment and measurement testing. In some cases a test bed is a piece of equipment for testing a specific product, in other cases it is a system of components that create a testing environment. Regardless, the end goal of a test bed is to ensure a component is working before it is used for its intended purpose.

Creating a test bed eases the process of troubleshooting if there is a failure during code testing. The purpose of this section is to create a test bed to test basic code in the Test Bed - Blocks and Test Bed - OnBot Java sections.

Creating a Test Bed

The design of a test bed depends on the use case and available resources. For instance, one of the design requirements for the test bed featured below was accessibility. Notice that the placement of the hardware components on the Extrusion allows for the actuators, sensors, and Control Hub to be removed or swapped out with ease.



Another major design consideration for this test bed was that it include the common components necessary to teach users the basics of programming with the REV Control System. In this case components were chosen from the REV FTC Starter Kit.

1. Control Hub
2. REV Core Hex Motor
3. Smart Robot Servo
4. Touch Sensor
5. Color Sensor V3
6. Battery

> Any one of these test beds components can be swapped out for an equivalent component. For instance, if you have an Expansion Hub rather than a Control Hub. However, with an Expansion Hub you may need to consider placement for the Robot Controller Phone.

There are other minor, but important, design considerations to make for a test bed. For example, when adding an actuator to a test bed consider the following questions:

- **What level of constraint does the actuator need?** One of the benefits of creating a test bed for motors, or other actuators, is that the motors can be properly constrained during the testing process. In this case providing basic motion support and constraint is valuable.
- **How will you be able to tell the behavior of the actuator?** The example test bed uses a wheel with a zip tie to help users visualize the behavior of the motor. Tape or other markers can be used, as well.

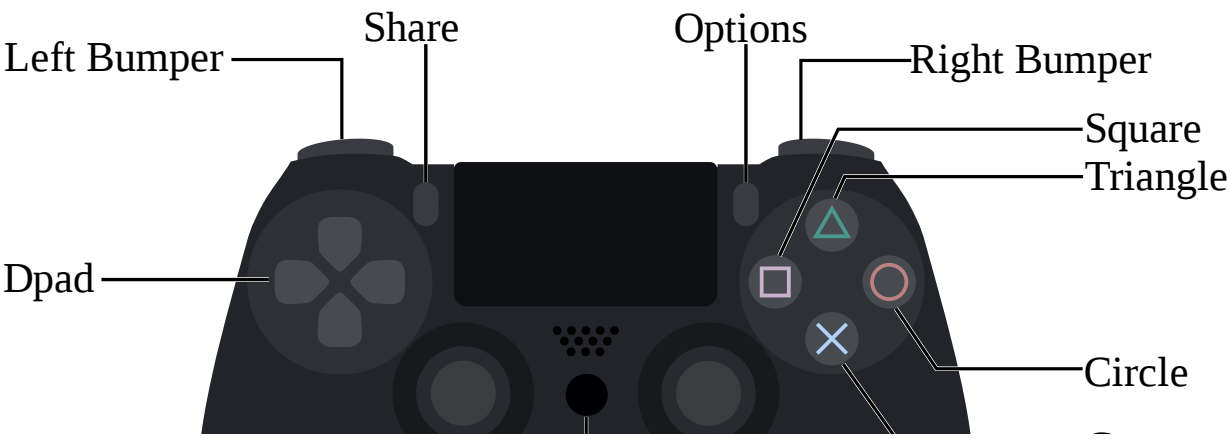
For the purpose of this guide a test bed similar to the example one can be built.

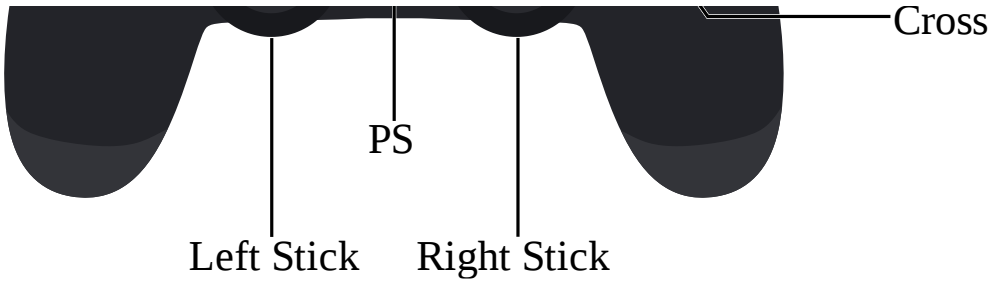
Using Gamepads

The Test Bed sections highlights the necessary robot components needed to learn the basic programming concepts used in the Test Bed - Blocks and Test Bed - OnBot Java sections. However, there are two more components needed to succeed in testing your code: A Driver Hub (or equivalent Driver Station Android Device) and a gamepad.

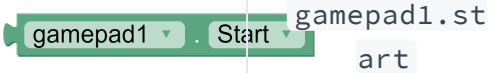
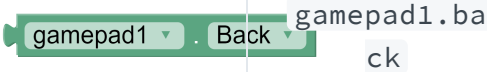
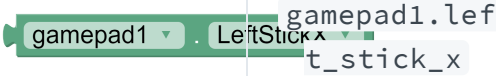
> For information on setting up a Driver Hub and gamepad please visits the Getting Start with Driver Hub guide.

All buttons on a gamepad can be programmed to a specific task or behavior. Throughout the Hello Robot Guide you will encounter several different places where the gamepad is utilized. Knowing the general naming convention for the gamepads will help you program them correctly. The guide assumes you are using either a Logitech gamepad or a PS4 gamepad, like Etpark Wired Controller for PS4 (REV-39-1865). To understand how to program a gamepad, especially with difference in the way certain buttons are named, please see the following graphic and table, showcasing what the code lines correspond with which button.





PS4 Controllers	Default (Logitech Gamepad)	Blocks	Java	Data Type
Cross	a	gamepad1 . A	gamepad1.a	Boolean
Circle	b	gamepad1 . B	gamepad1.b	Boolean
Triangle	y	gamepad1 . Y	gamepad1.y	Boolean
Square	x	gamepad1 . X	gamepad1.x	Boolean
Dpad Up	Dpad Up	gamepad1 . DpadUp	gamepad1.dpad_up	Boolean
Dpad Down	Dpad Down	gamepad1 . DpadDown	gamepad1.dpad_down	Boolean
Dpad Left	Dpad Left	gamepad1 . DpadLeft	gamepad1.dpad_left	Boolean
Dpad Right	Dpad Right	gamepad1 . DpadRight	gamepad1.dpad_right	Boolean
Left Bumper	Left Bumper	gamepad1 . LeftBumper	gamepad1.left_bumper	Boolean
Right Bumper	Right Bumper	gamepad1 . RightBumper	gamepad1.rig	Boolean

		ht_bumper	
Left Trigger	Left Trigger	 gamepad1.left_trigger	Float
Right Trigger	Right Trigger	 gamepad1.right_trigger	Float
PS	n/a	 gamepad.ps	Boolean
Options	Start	 gamepad1.start	Boolean
Share	Back	 gamepad1.back	Boolean
Left Stick Button	Left Stick Button	 gamepad1.left_stick_button	Boolean
Left Stick X Axis	Left Stick X Axis	 gamepad1.left_stick_x	Float
Left Stick Y Axis	Left Stick Y Axis	 gamepad1.left_stick_y	Float
Right Stick Button	Right Stick Button	 gamepad1.right_stick_button	Boolean
Right Stick X Axis	Right Stick X Axis	 gamepad1.right_stick_x	Float
Right Stick Y Axis	Right Stick Y Axis	 gamepad1.right_stick_y	Float

Data Types

Boolean

Boolean

Boolean data has two possible values: **True and False**. These two values can also be represented by **On and Off** or **1 and 0**. Buttons, bumpers, and triggers on the gamepad provide boolean data to your robot. For example, a button that is not pressed will return a value of False and a button that is pressed will return the value True.

Float

Float data is a number that can include decimal places and positive or negative values. On the gamepad, the float data returned will be between 1 and -1 for the joystick's position on each axis. Some examples of possible values are 0.44, 0, -0.29, or -1.

Test Bed - Blocks

The Blocks Programming Tool is a visual, programming tool that lets programmers use a web browser to create, edit and save their op modes. Blocks, like other scratch based programming tools, is a collection of preset code snippets that users can drag-and-drop into the appropriate code line. In this section users can learn how to create an op mode, as wells as the basics of programming the actuators and sensors featured on the test bed.

Follow the guide in order to get an in depth understanding of working with Blocks or navigate to the section that fits your needs:

Section	Goals of Section
Creating an Op Mode	Focuses on how to navigate the Blocks interface and create an op mode.
Programming Essentials	Breaks down the structure and key elelments needed for an op mode, as well as some of the essential components of Blocks and programming logic.
Programming Actuators	How to code servos and motors. This section wall through the basic logic of coding actuators, controlling actuators with a gamepad, and using telemetry.
Programming Sensors	How to code a digital device. The section focuses on the basic logic of coding a digital device, like a REV Touch Sensor.

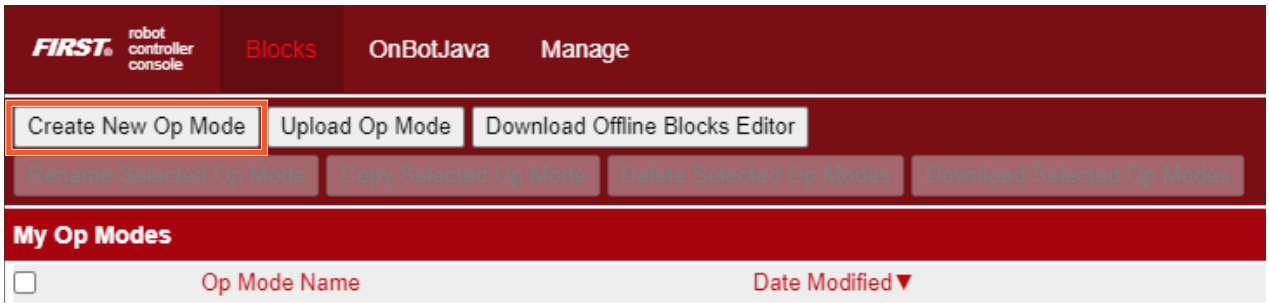
Creating an Op Mode

Before diving in and creating your first op mode, you should consider the concept of **naming conventions**.

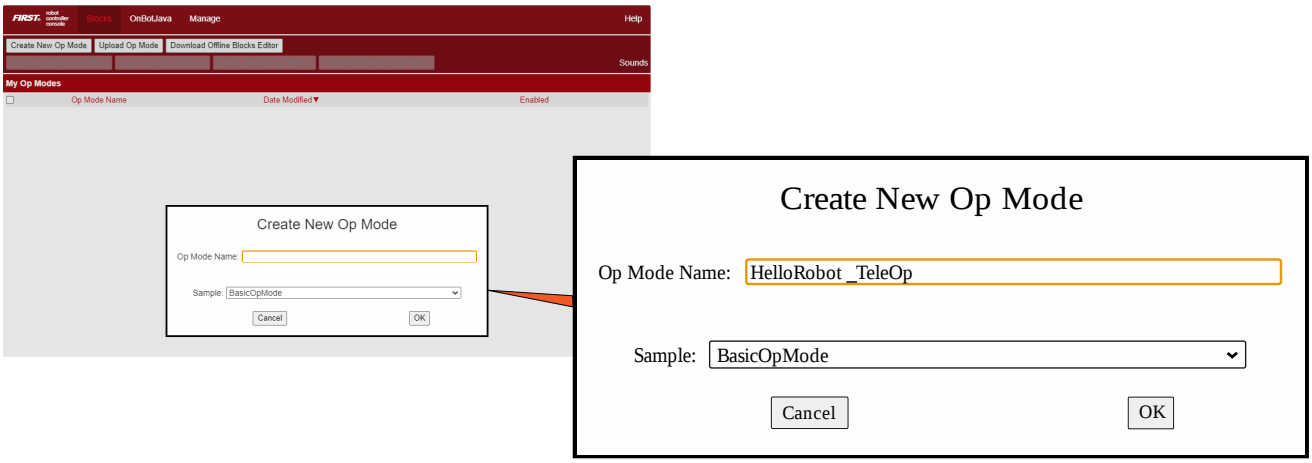
When writing code the goal is to be as clear as possible about what is happening within the code. This is where the concept of naming conventions comes into play. Common naming conventions have been established by the programming world to denote variables, classes, functions, etc. Op modes share some similarities to **classes.** Thus the naming convention for op modes tends to follow the naming convention for classes; where the first letter of every word is capitalized.

> This section assumes that you have already accessed the Blocks platform during the Hello Robot - Introduction to Programming. If you are unsure how to access blocks please revisit this section before proceeding.

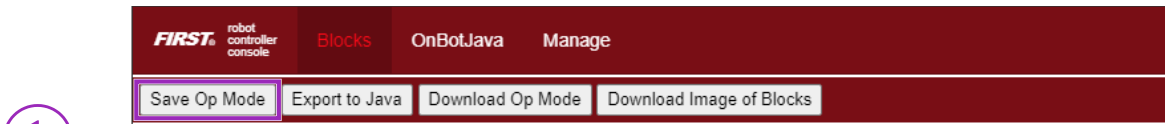
To start, access the Robot Controller Console and go to the Blocks page. In the upper right-hand corner of there is a ***Create New Op Mode*** button, click it.

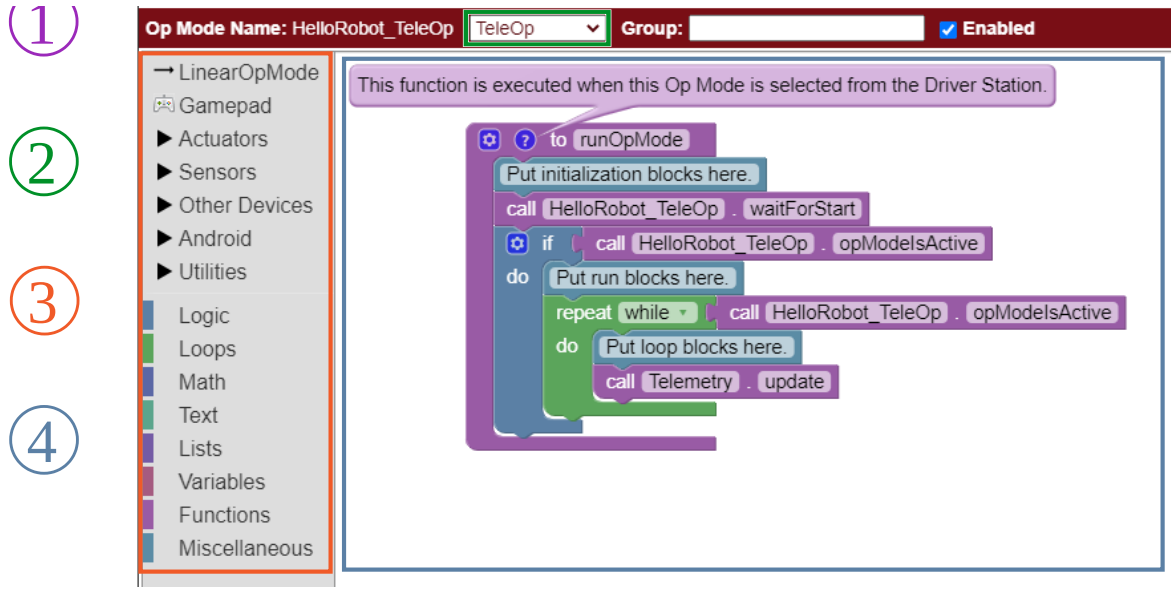


Clicking the ***Create New Op Mode*** button will open up the ***Create New Op Mode*** window. This window allows users to name their op modes and select a sample code to build off of. For this guide use the default **BasicOpMode** sample and name the op mod **HelloRobot_TeleOp** as shown in the image below.



Once the op mode has been named click 'OK' to proceed forward. Creating an op mode will open up the main Blocks programming page. Before moving on to programming, take some time to learn and understand the following key components of Blocks featured in the image below.



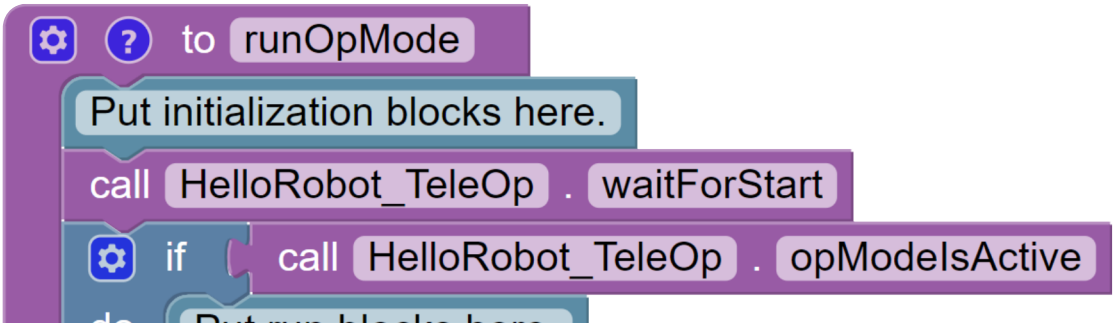


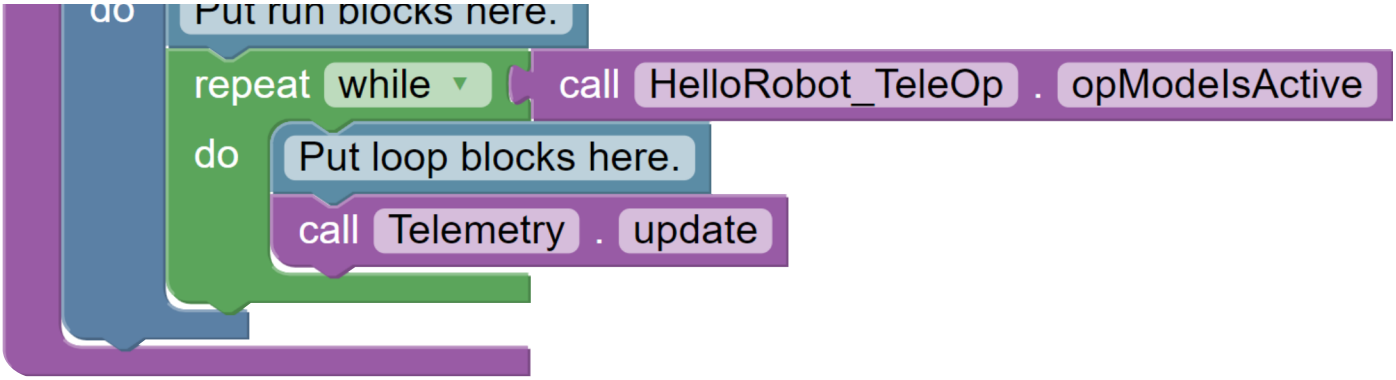
1. **Save Op Mode** - Click this button to save an op mode to the robot. It is important to save the op mode any time you stop working on a code, so that progress is not lost.
2. **TeleOp/Autonomous** - This section of blocks allows users to change between the two types of op modes: teleop and autonomous.
3. **Categorized Blocks** - This section of the screen is where the programming blocks are categorized and accessible. For instance, clicking **Logic** will open access to programming blocks like if/else statements.
4. **Programming Space** - This space is where blocks are added to build programs.

> ⚠ If a configuration has been made then the Actuators, Sensors, and Other Devices in the Categorized Blocks section should appear as drop down menus, where blocks that are unique to specific hardware can be accessed. If this is not the case a configuration file has not been made. For more information visit the Configuration page, before moving forward with programming.

Programming Essentials

During the process of creating an op mode the Blocks tool prompted the selection of a sample code. In Blocks these samples act as templates; providing the blocks and logical structure for different robotics use cases. In the previous section the sample code **BasicOpMode** was selected. This sample code, seen in the image below, is the structural shell needed in order to have a working op mode.

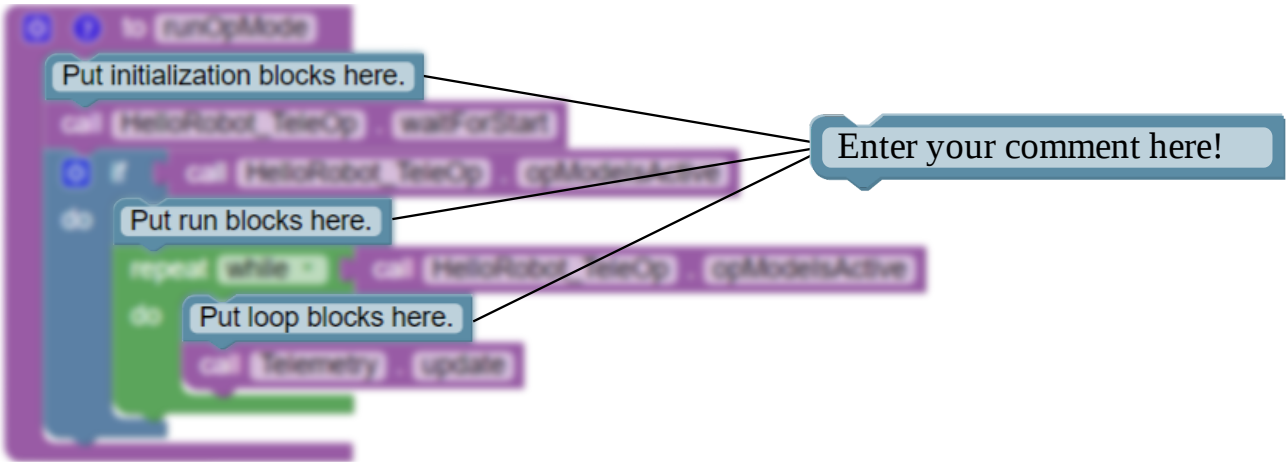




An op mode can often be considered a set of instructions for a robot to follow in order to understand the world around it. The BasicOpMode provides the initial set of instructions that are needed in order for an op mode to properly function.

Though this sample is given to users to reduce some of complexities of programming as they learn; it introduces some of the most important code blocks. It is also important to understand what is happening in the structure of the BasicOpMode, so that code blocks are put in the correct area.

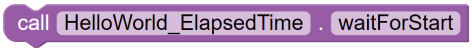
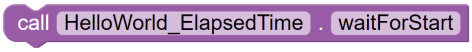
Key Op Mode Blocks

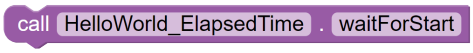
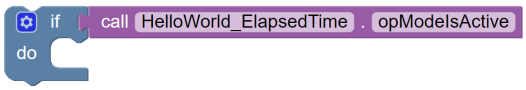


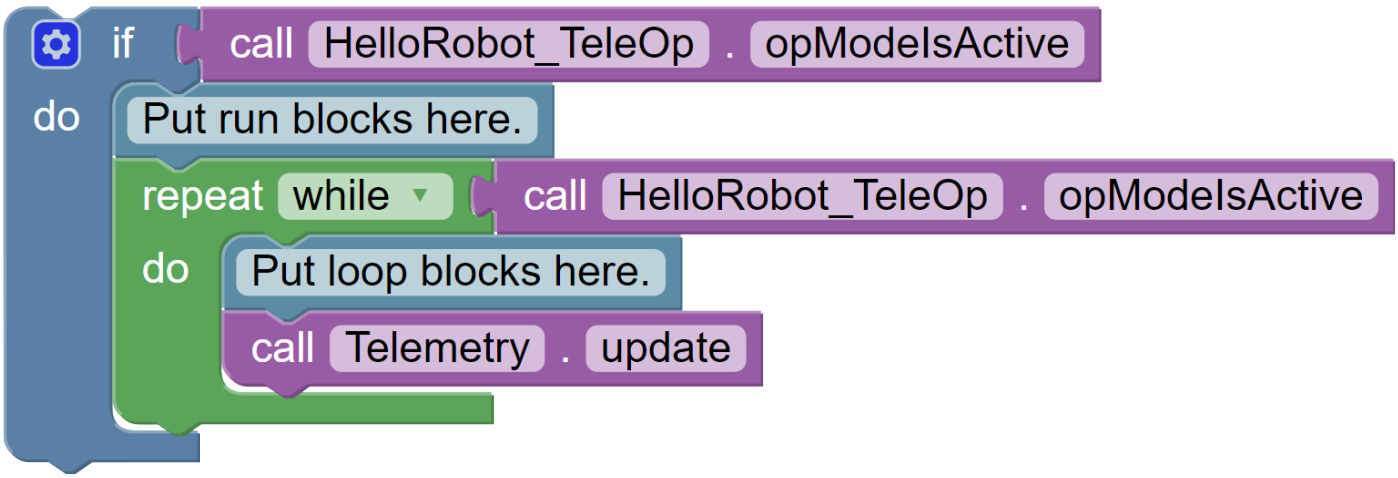
Comments are blocks of code that benefit the human user. They are used by programmers to explain the function of a section of code. This is especially helpful in collaborative programming environments. If code is handed from one programmer to another, comments communicate the intent of the code to the other programmer. Blocks like Put initialization blocks here. are comments written by the FIRST Tech Team to inform the user what will happen when blocks are added directly beneath the comment.

For instance, any programming blocks that are placed after the Put initialization blocks here. comment (and before the call HelloWorld_ElapsedTime . waitForStart block) will be executed when the op mode is first selected by a user at the Driver Station. Typically, blocks put in this section are meant to create and define **variables** between the initialization and start phases of the op mode.

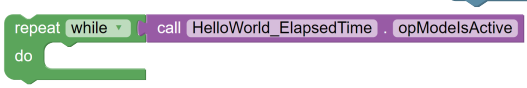
> A variable is a storage location with an associated symbolic name, which contains some known or unknown quantity of information referred to as a value. Variables can be numbers, characters, or even motors and servos.

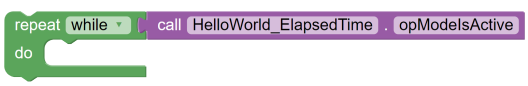
When the Robot Controller reaches the block  it will stop and wait until it receives a Start command from the Driver Station. A Start command will not be sent until the user pushes the Start button on the Driver Station. Any code after the  block will get executed after the Start button has been pressed.

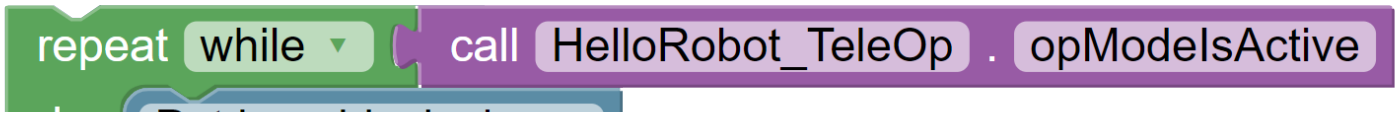
After the  , there is a conditional **if** block  that only gets executed if the op mode is still active (i.e., a stop command hasn't been received).

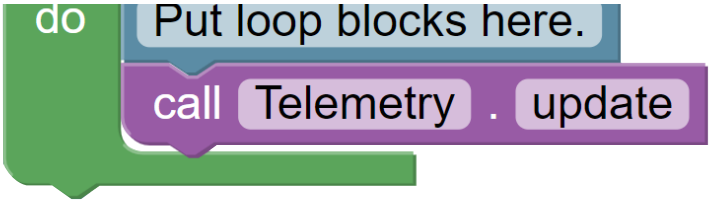


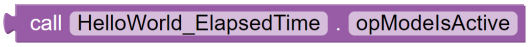
> **If-then** (if-else) statements are similar to the concept of cause and effect. If cause (or condition) happens, then perform effect.

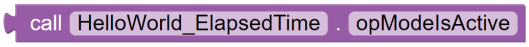
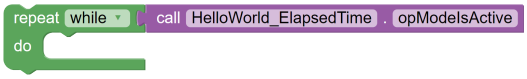
Any blocks that are placed after the  comment and before the  will be executed sequentially by the Robot Controller after the Start button has been pressed.

The  is an **iterative or looping** control structure.



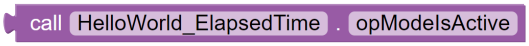


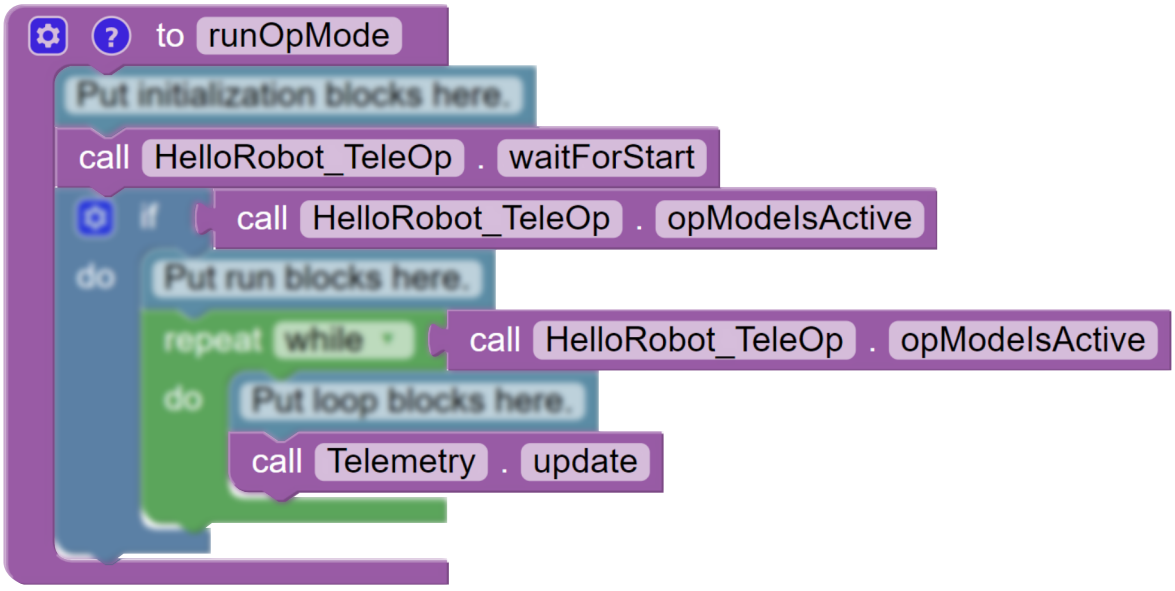
This control will perform the steps listed under the “do” portion of the block as long as the condition  is true. What this means is that the statements included in the “do” portion of the block will repeatedly be executed as long as the op mode *HelloRobot_TeleOp* is running.

Once the user presses the Stop button, the  clause is no longer true and the  loop will stop repeating itself.

Functions and Methods

The previous section did not go into a detailed discussion of the purple **function** (or **method**) blocks. Functions and methods are similar procedures in programming that are more advance than what will be covered in this guide.

For now the most important thing to know is that occasionally methods within the SDK libraries will need to be called in order to perform a certain task. For instance, the  line calls the method *opModeIsActive*, which is the procedure in the SDK that is able to tell when the robot was been started or stopped.



When your programming skills have advanced take sometime to visit the concepts of functions and methods and explore how they can help you enhance your code.

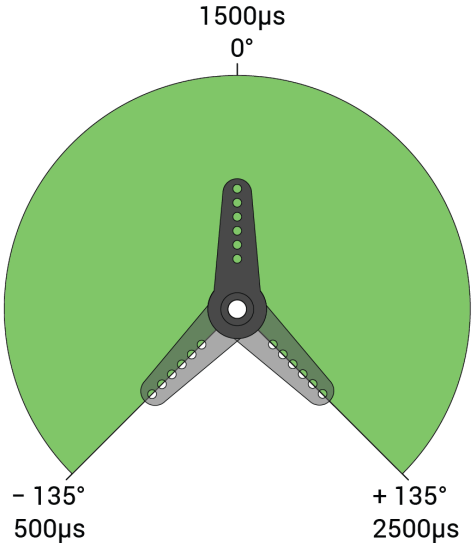
Programming Actuators

Servo Basics

The goal of this section is to cover some of the basics of programming a servo within Blocks. By the end of this section users should be able to control a servo with a gamepad, as well as understand some of the key programming needs of the servo.

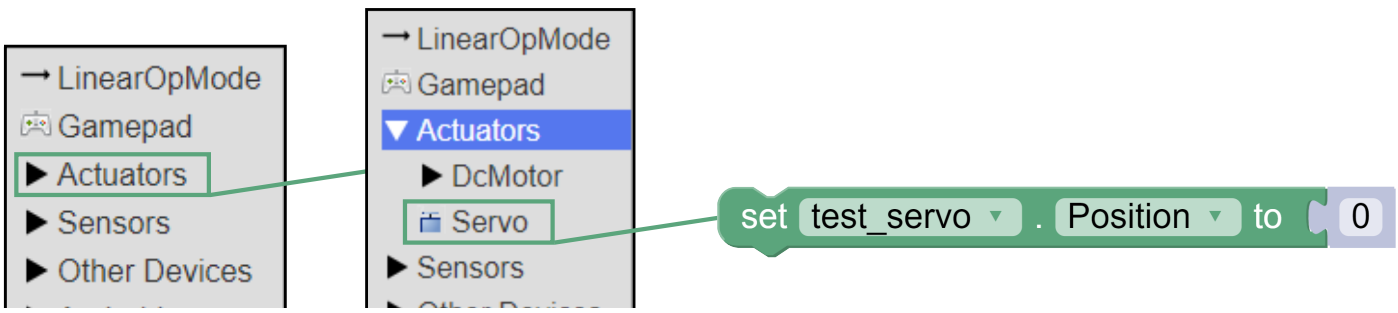
> This section is considering the Smart Robot Servo in its default mode. If your servo has been changed to function in continuous mode or with angular limits it will not behave the same using the code examples below. You can learn more about the Smart Robot Servo or changing the Servo's mode via the SRS Programmer by clicking the hyperlinks.

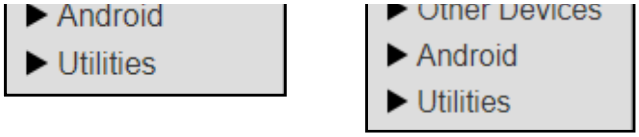
With a typical servo, you can specify a target position for the servo. The servo will turn its motor shaft to move to the target position, and then maintain that position, even if moderate forces are applied to try and disturb its position.



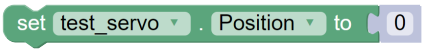
For both Blocks and OnBot Java, you can specify a target position that ranges from 0 to 1 for a servo. For a servo with a 270° range, if the input range was from 0 to 1 then a signal input of 0 would cause the servo to turn to point -135°. For a signal input of 1, the servo would turn to +135°. Inputs between the minimum and maximum have corresponding angles evenly distributed between the minimum and maximum servo angle. This is important to keep in mind as you learn how to code servos.

Since this section will focus on servos it is important to understand how to access servos within Blocks. At the top of the Categorize Blocks section there is a drop down menu for **Actuators**. When the menu is selected it will drop down two choices: **DcMotor** or **Servo**. Selecting Servo will open a side window filled with various servo related blocks.

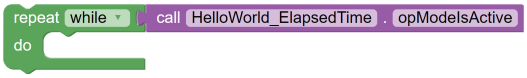


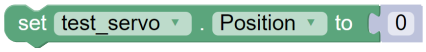


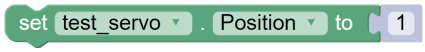
Programming a Servo

From the Servo menu select the block 

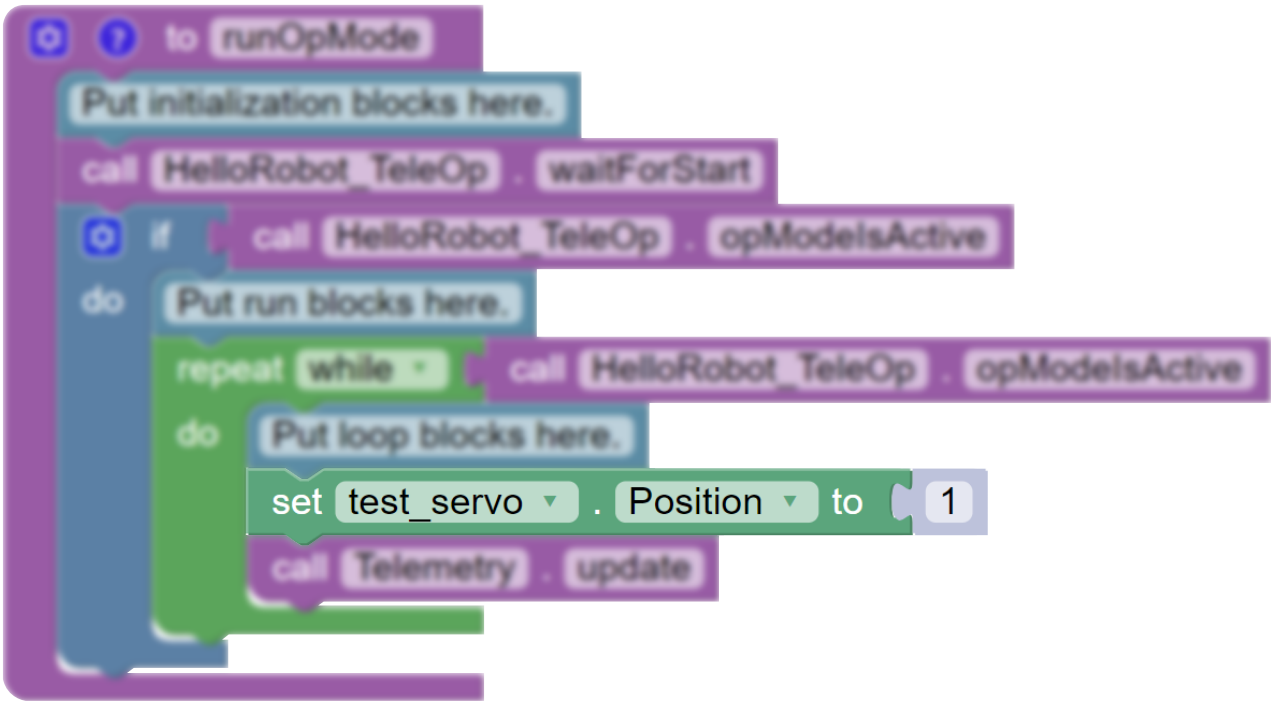
> The block above will change names depending on the name of the servo in a configuration file. If there are multiple motors in a configuration file the arrow next to test_servo will drop down a menu of all the servos in a configuration.

Add this block to the op mode code within the 

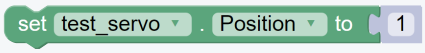
. Click on the number block to change from 

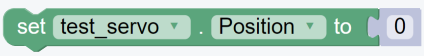
to 

.

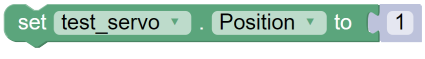


Select **Save Op Mode** in the upper right corner in the Robot Controller Console.

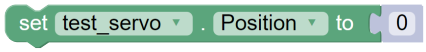
> Try running this op mode on the test bed two times and consider the following questions:
>
> - Did the servo move during the first run?
> - Did the servo move during the second run?
>
> If the servo did not move switch from 

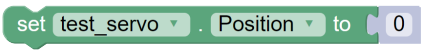
back to 

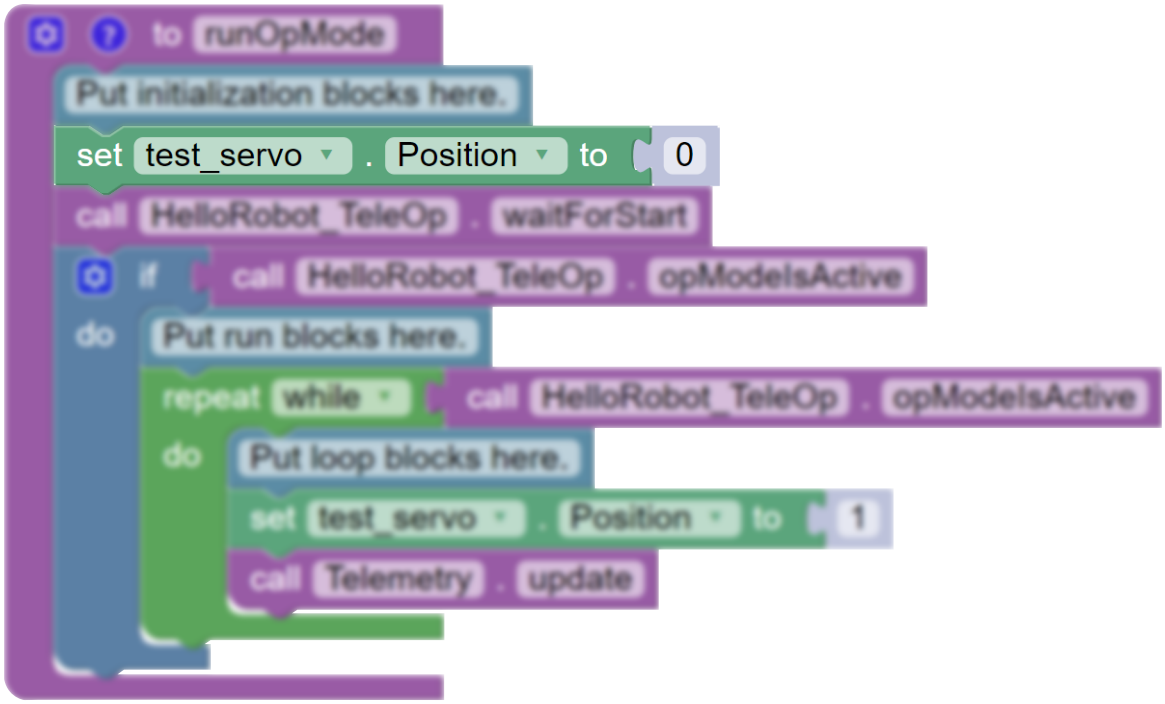
and try again.

The intent of the 

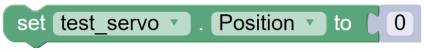
is to set the position of the servo. If the servo is already in the set position when a code is run, it will not change positions. Lets try adding another

 block and see what changes.

Drag an additional  block into the op mode code under the  comment.



> Try running this op mode on the test bed and consider the following question:
>
> - What is different from the previous run?

The  that was added in the step above changes the servo position to 0 during the initialization phase, so when the op mode is run the servo will always move to position 1. For some applications starting the servo in a **known state**, like at position zero, is beneficial to the operation of a mechanism. Setting the servo to the known state in the initialization ensures it is in the correct position when the op mode runs.

Programming a Servo with a Gamepad

The focus of this example is to assign certain servo positions to buttons on the gamepad. For this example the known state will stay at position 0, so that after initialization the servo will be a the -135 degree position of the servo range. The following list shows what buttons correspond with which servo position.

> If you are using a PS4 Controller, like the Etpark Wired Controller for PS4 (REV-39-1865), see the Using Gamepads section to determine how the gamepad code used in this section translates to the PS4 Gamepad.

Button	Degree Position	Code Position
Y	-135	0

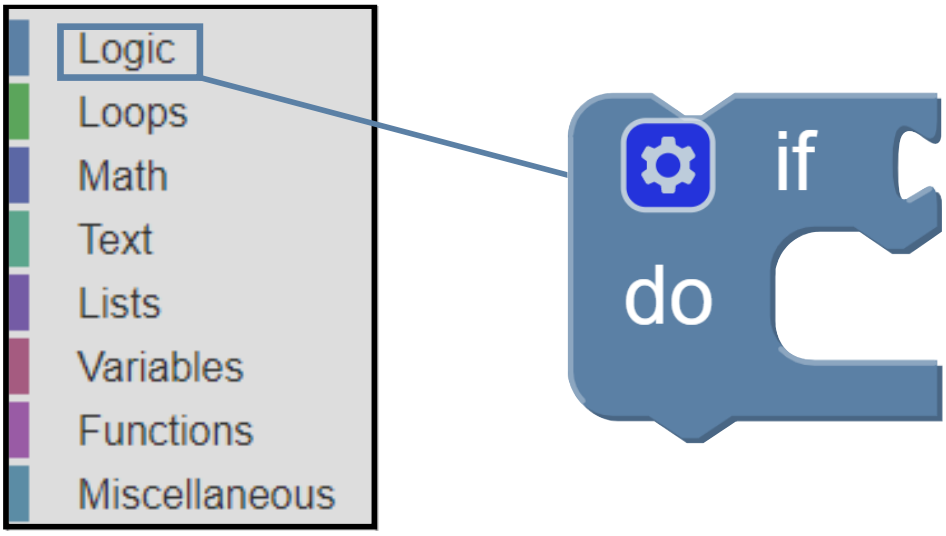
X	0	0.5
B	0	0.5
A	135	1

The best way to switch the servo position will be to use a conditional `if/ else if` statement. An if statement considers whether a conditional statement is true or false. If the conditional statement is true a defined action (like the servo moving) is performed. If the conditional statement is false the action is not performed.

An `if/ else if` statement takes in multiple different conditional statements. If the first conditional statement is found to be false then the second conditional state is analyzed. Each statement in the `if/ else if` will be analyzed one by one until a statement is found true or all statements are found false. For this example, there will be three conditions that will need to be checked.

From the **Logic** Menu in Blocks select the 

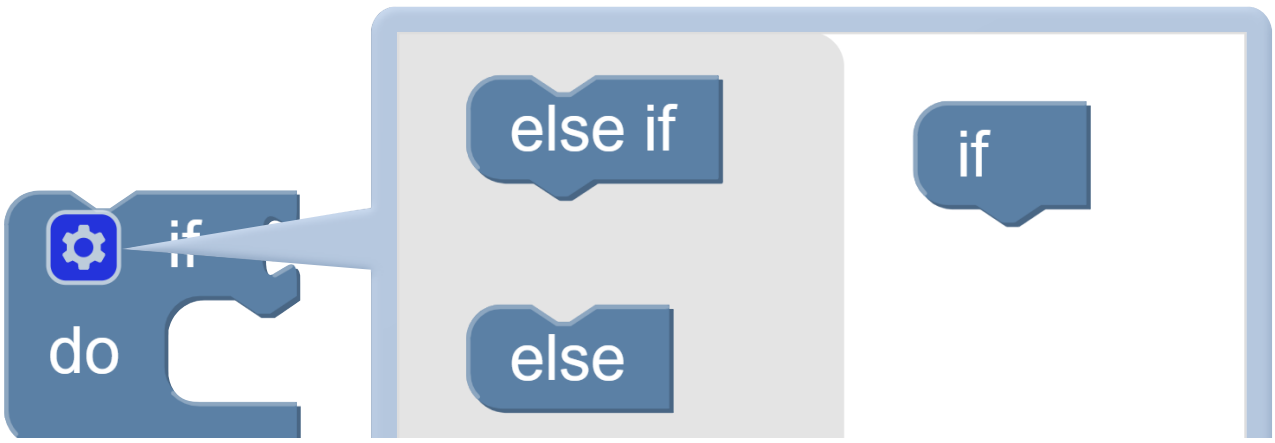
block and drag in into the op mode's while loop.



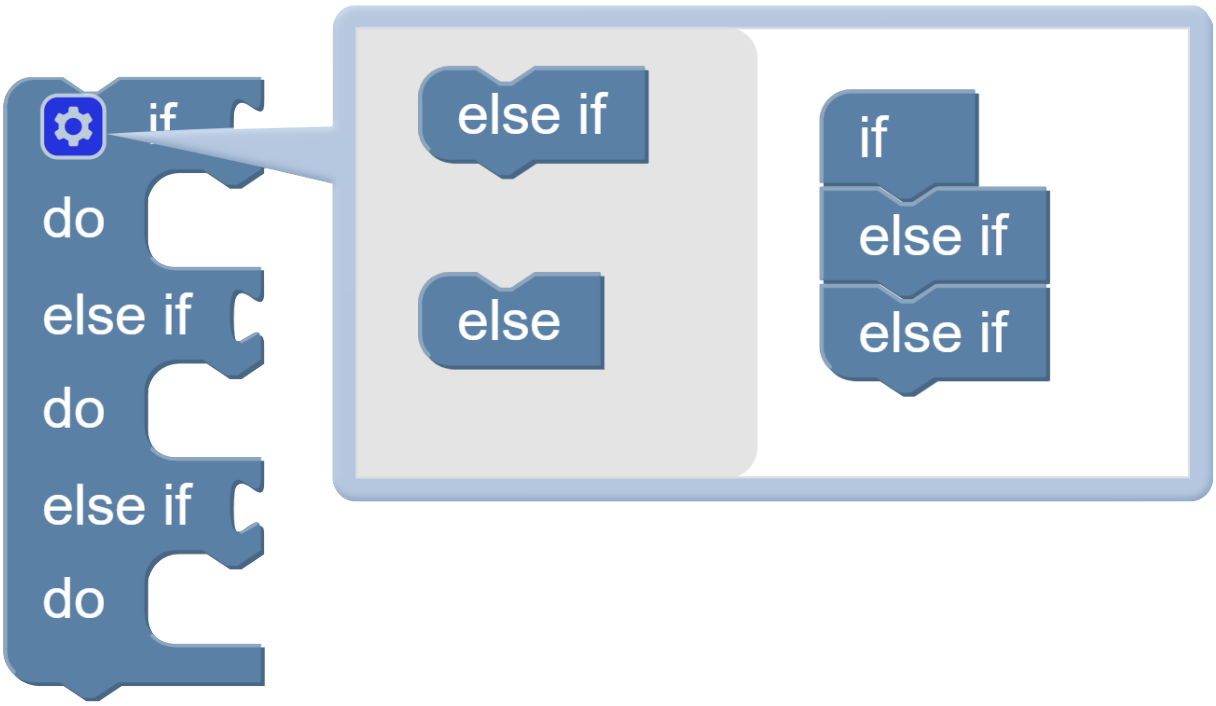
Click on the blue and white Settings icon for the 

block. This will display a pop-up menu that lets you modify the 

block.



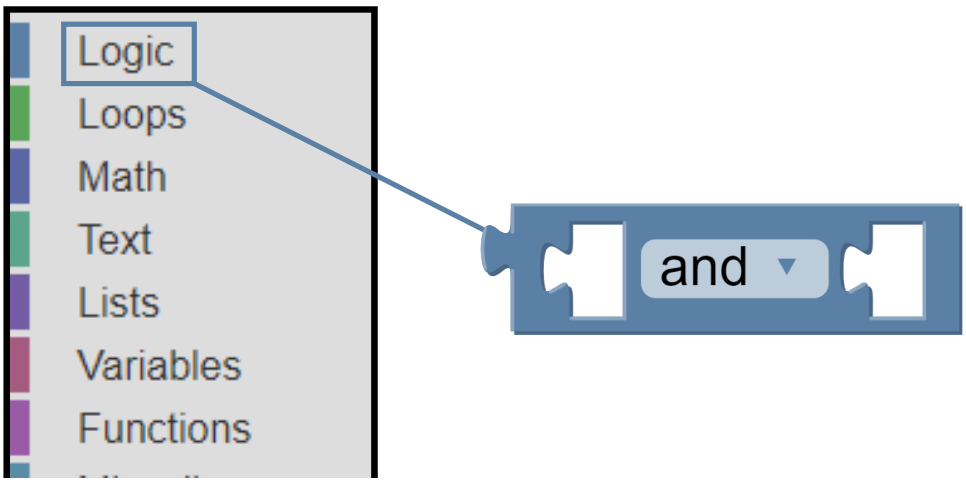
Drag an else if block from the left side of the pop-up menu and snap it into place under the if block. Drag a second else if block from the left side and snap it into place on the right side under the first else if block.

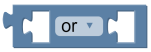


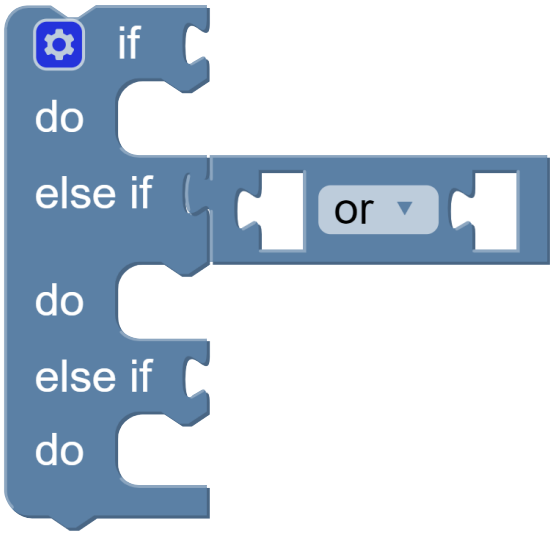
There are three different paths in this `if/else if` block. Each one corresponds with one of the three chosen servo positions 0, 0.5, and 1. However, there are four different buttons that will be used for this example. Both button B and button X should be able to move the servo to position 0.5. In order to do this the logical operator **or** needs to be used.

> The logical operator **or** considers two operands if either (or both) are true the or statement is true. If both operands are false the *or* statement is false.

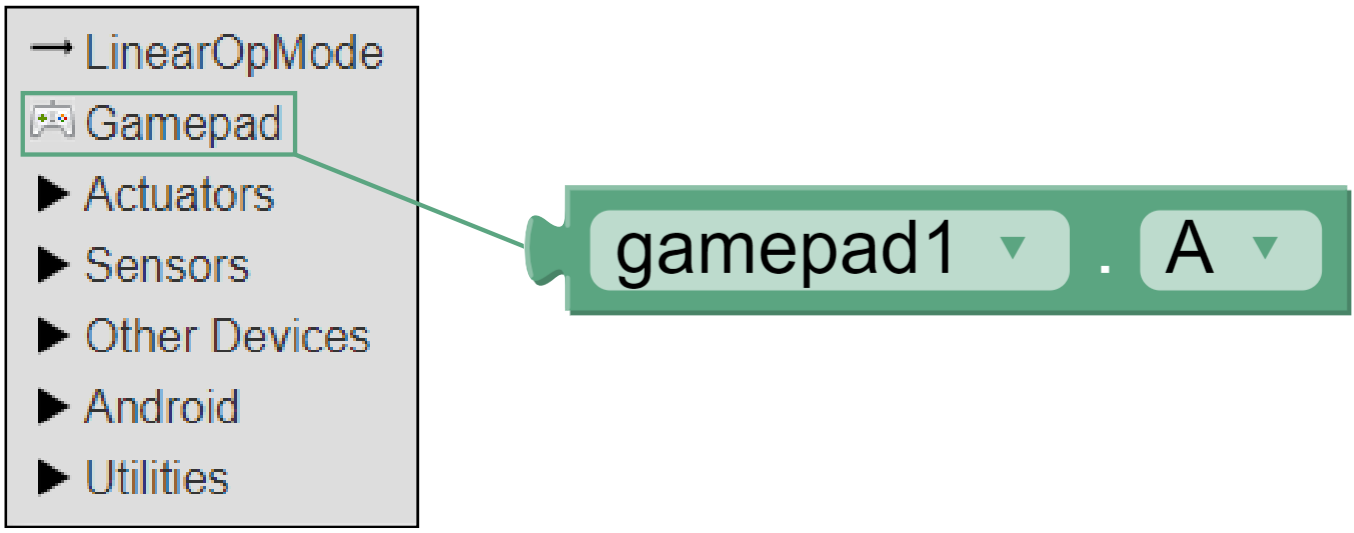
From the **Logic** Menu in Blocks select the and block.



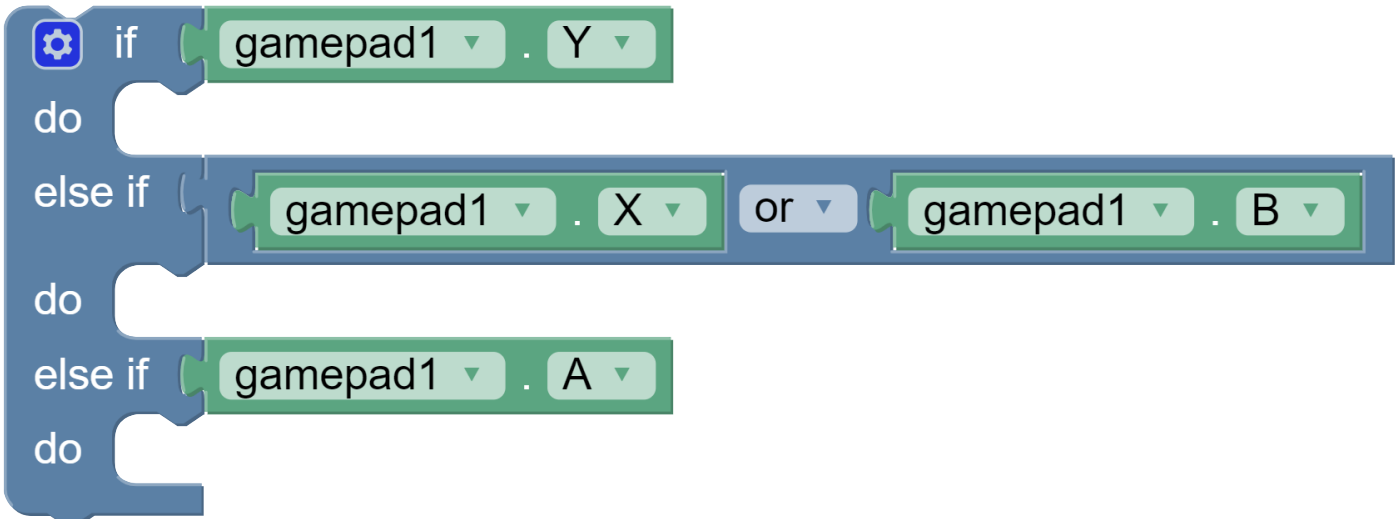
Add this block to the `If/else if` block, as shown in the image below. Use the drop down menu on the block to change it from an  block to an  block.

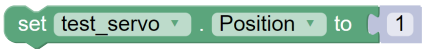


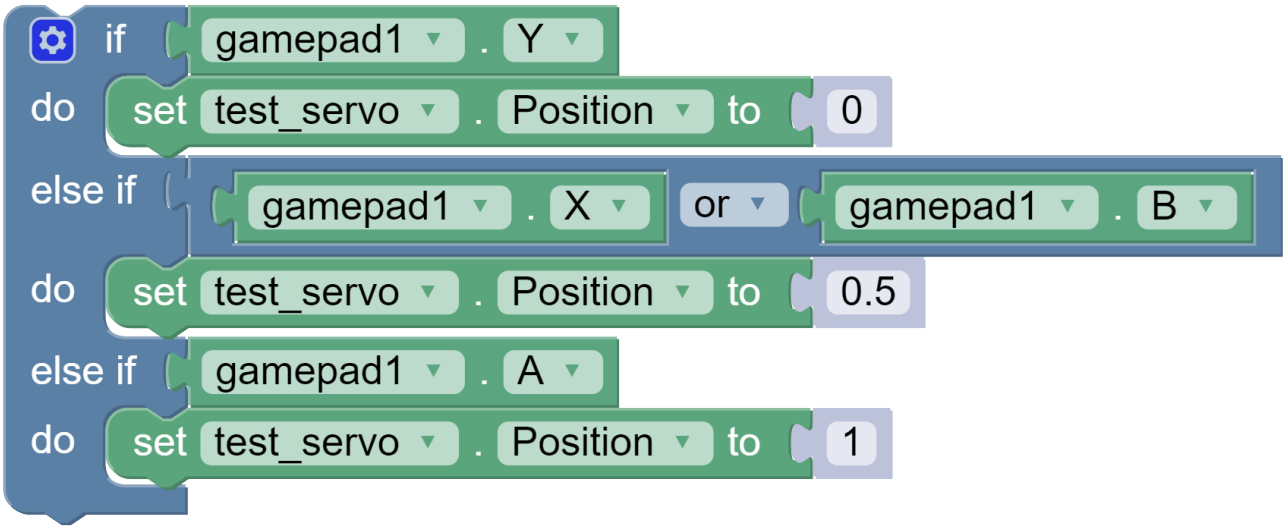
All gamepad related blocks are in the **Gamepad** Menu.



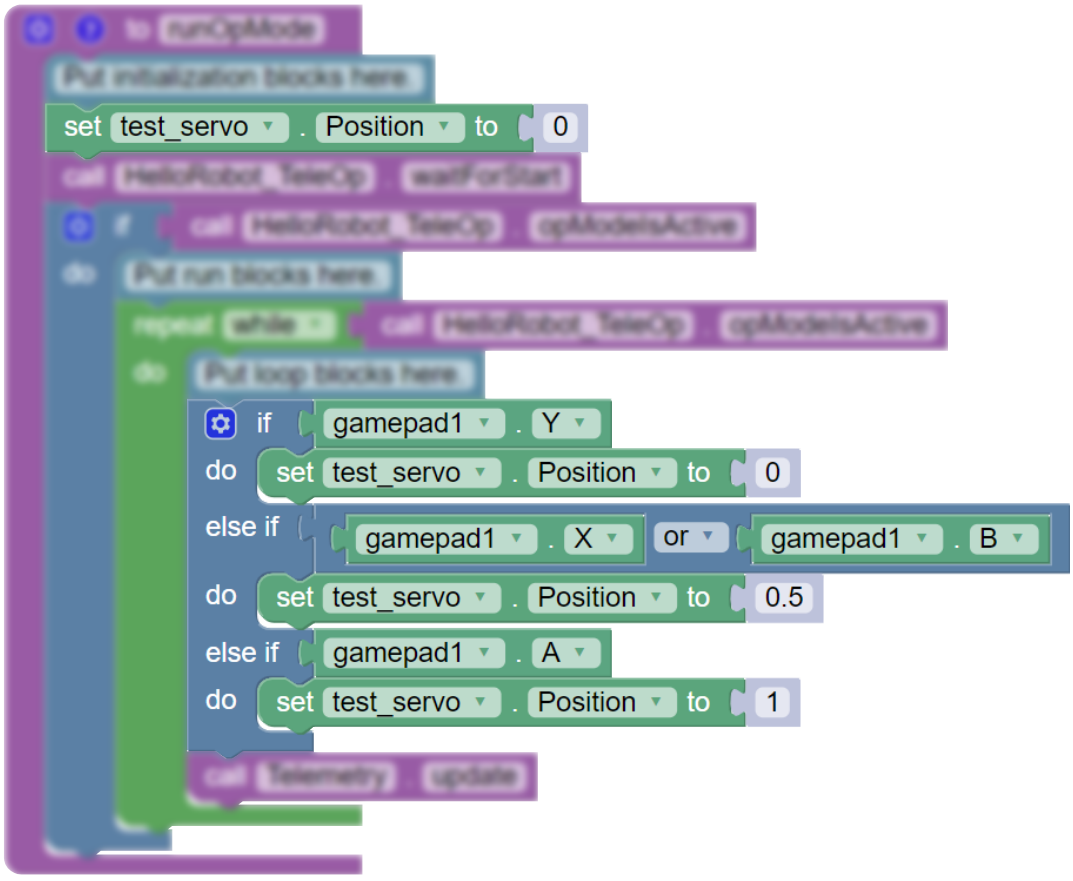
Add each button to the `if/else if` block as seen in the image below.



Add  blocks to each section of the *If/else if* block. Set the servo position to correspond with the assigned gamepad button.



There are three different paths in this `if/else if` statement. If the first conditional statement is true (the Y button is pressed) the servo moves to code position 0 and the other conditional statements are ignored. If the first condition is false (the Y button is not pressed) the second condition is analyzed. Recall that this behavior repeats until a condition is met or all conditions have been tested and found false.



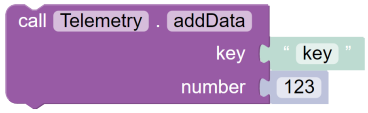
Servos and Telemetry

Telemetry is the process of collecting and transmitting data. In Robotics telemetry is used to output internal

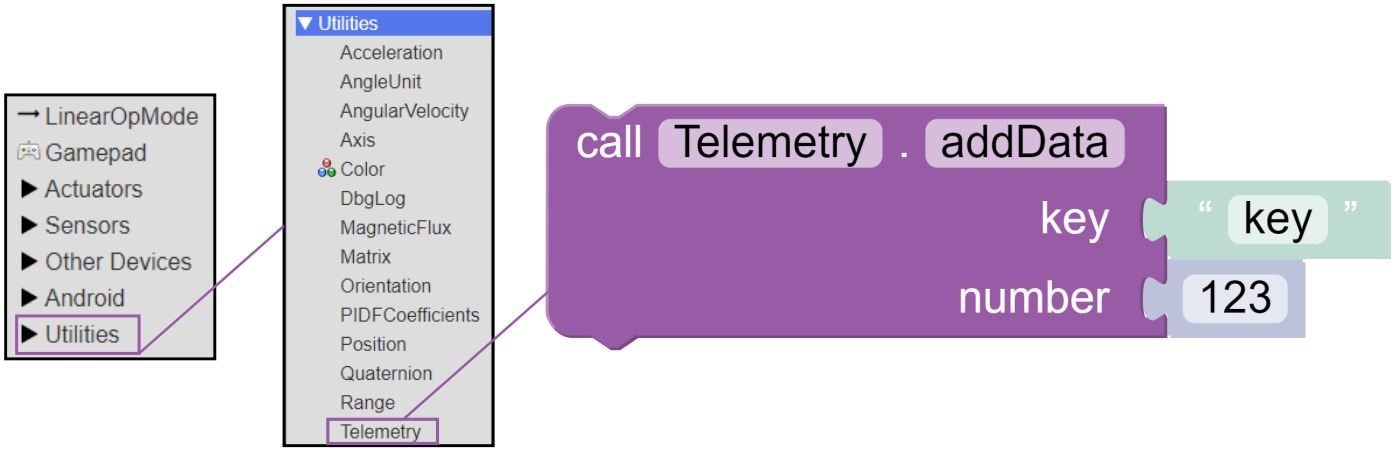
data from actuators and sensors to the Driver Station. This data can then be analyzed by users to make decisions that can improve code.

The most useful telemetry from the servo is the the position of the servo along its 270 degree range. In order to get that information the following line needs to be used.

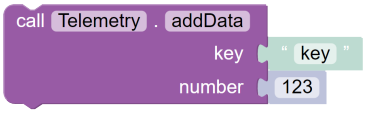
In order to access the telemetry blocks select the **Utilities** drop down. The utilities drop down is in alphabetical order, so telemetry is towards the bottom of the drop down options. Select the



block from the telemetry menu.



Drag the



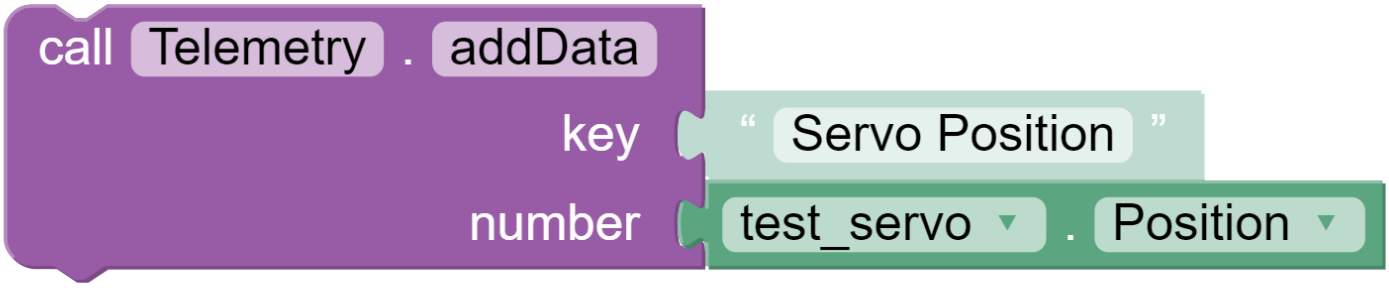
block and place it beneath the `if/else if` block set.

From the **Servo** menu pullout the block

Drag the Block and attach it to the

number **parameter** on the telemetry blocks.

Change the key parameter to "Servo Position"



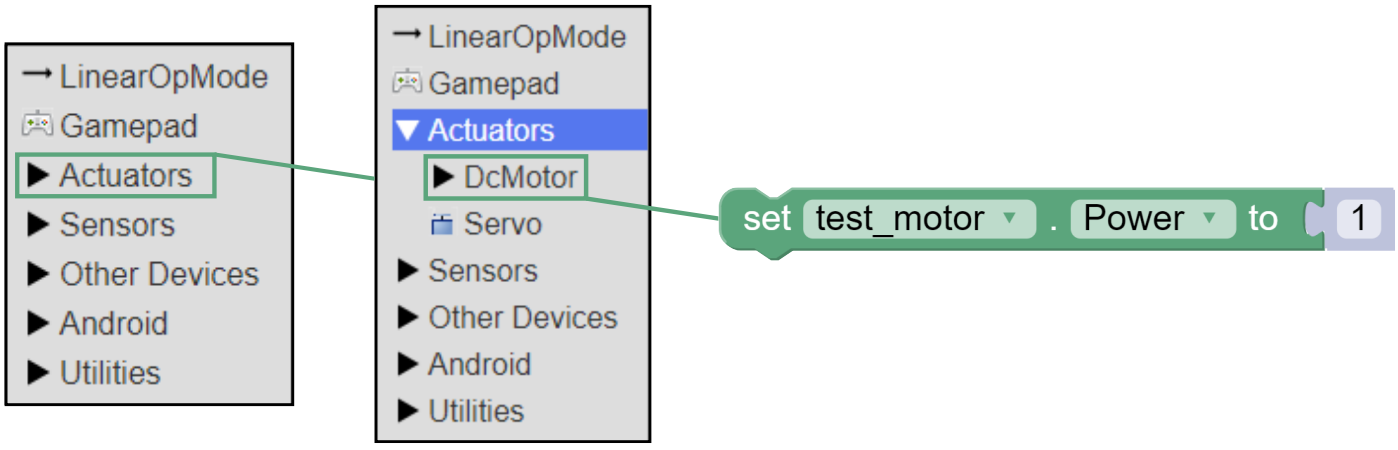
When the op mode is run the telemetry block will display the current position information will be displayed with the Servo Position Key. The number that corresponds with the current position will change as the servo shaft position changes.

Motor Basics

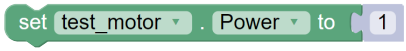
> Modify your op mode to add the motor related code. This can be done by clearing out your current code modifications or adding the motor related code to your current op mode.

The goal of this section is to cover some of the basics of programming a motor within Blocks. By the end of this section users should be able to control a motor using a gamepad, as well as understand some of the basics of working with motor encoders.

Since this section will focus on motors it is important to understand how to access motors within Blocks. At the top of the Categorize Blocks section there is a drop down menu for **Actuators**. When the menu is selected it will drop down two choices: **DcMotor** or **Servo**. Selecting DC Motor will open a side window filled with various motor related blocks.



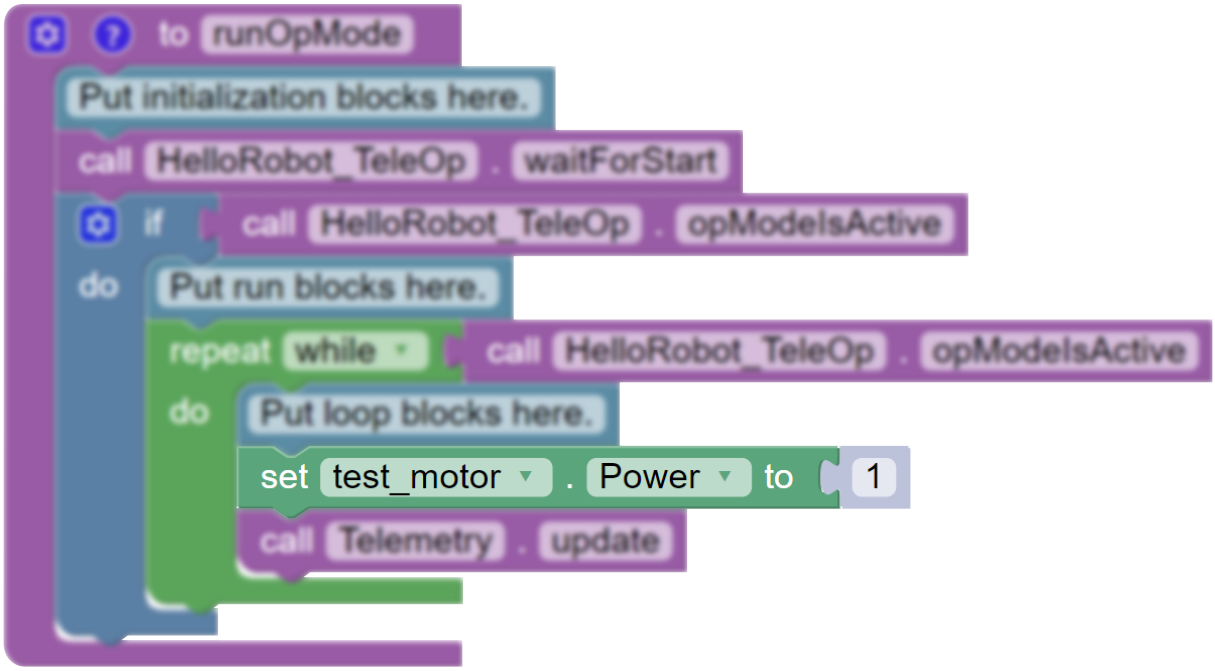
Driving Motors

From the Dc Motor menu in Blocks select the block 

.

> The block above will change names depending on the name of the motor in a configuration file. If there are multiple motors in a configuration file the arrow next to test_motor will drop down a menu of all the motors in a configuration.

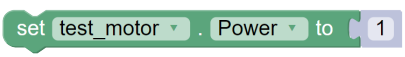
Add this block to the op mode code within the **while loop**.



Select **Save Op Mode** in the upper right corner in the Robot Controller Console.

> Try running this op mode on the test bed and consider the following questions:
>
> - How fast is the motor running?
> - What happens if you change the power from **1** to **0.3**?
> - What happens if you change the power to **-1**?

The level of power sent to the motor is dependent on the numerical number assigned to the motor. The change from 1 to 0.3 decreased the motors speed from 100% of duty cycle to 30% of duty cycle. Meanwhile, the change to -1 allowed the motor to rotate at 100% duty cycle in the opposite direction. So, power can be fluctuated to drive a motor *forward* or *backwards*.

However, the 

block will run the motor in the assigned direction until something in the code stops the motor or causes a change in direction.

> To better understand motors and the concept of duty cycle check out the our Motors and Choosing an Actuator documentation.

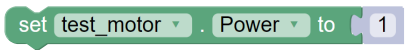
Driving Motors with the Gamepad

In the previous section you learned how to set the motor to run at a specific power level in a specific direction. However, in some applications, it may be necessary to control the motor with a gamepad, to easily change the direction or power level of a mechanism.

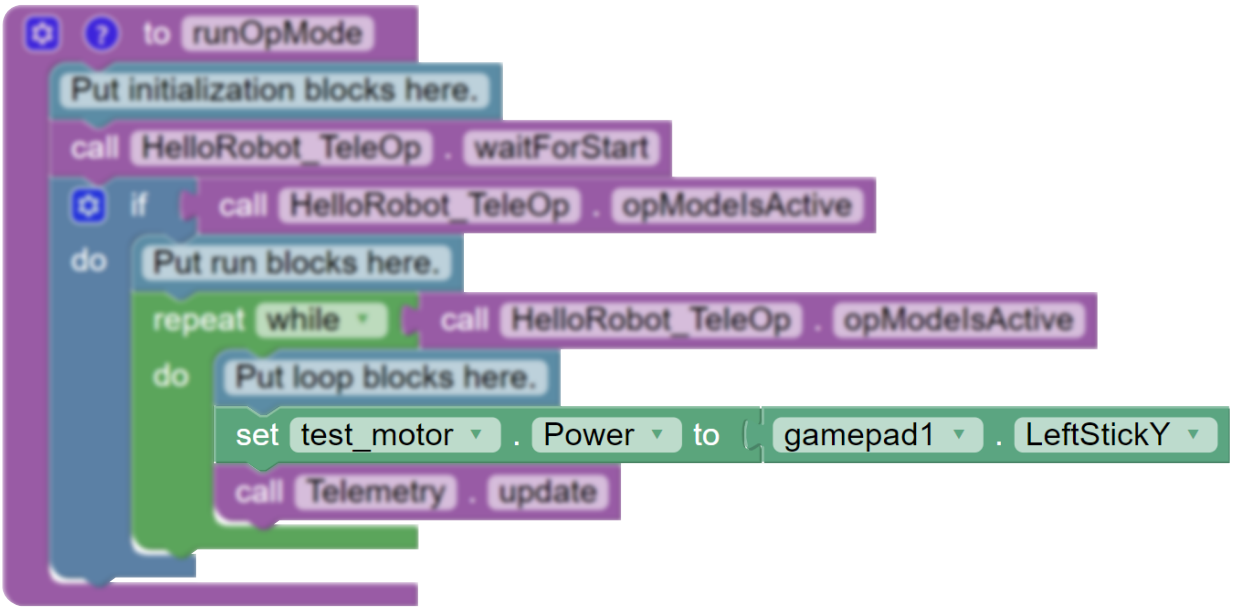
From the Gamepad Menu in Blocks select the 

Block.

Drag the 

block so it snaps in place onto the right side of the 

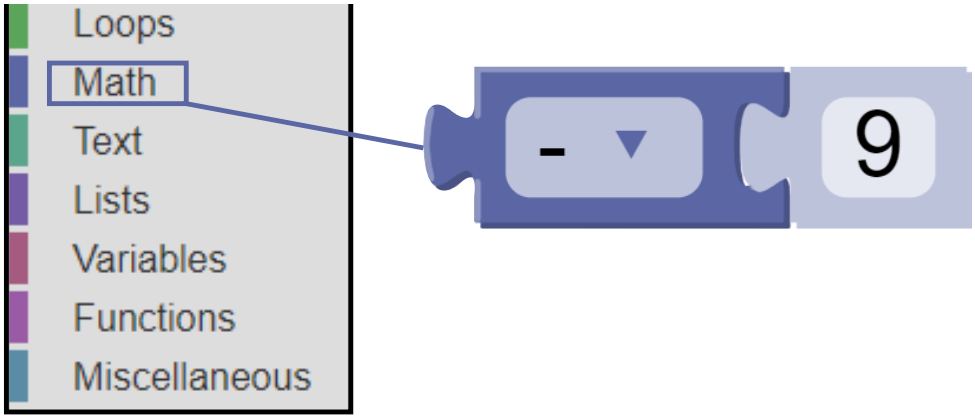
block. This set of blocks will continually loop and read the value of gamepad #1's left joystick (the y position) and set the motor power to the Y value of the left joystick.



Note that for the Logitech F310 gamepads, the Y value of a joystick ranges from -1, when a joystick is in its topmost position, to +1, when a joystick is in its bottommost position. If the motor is not running in the intended direction adding a **negative symbol**, or negation operator, to the line of code will change the direction of the motor in relation to the gamepad.

From the **Math** Menu in Blocks select the 

block in the image below.

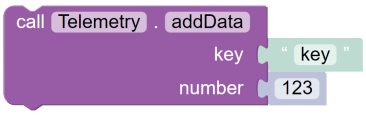


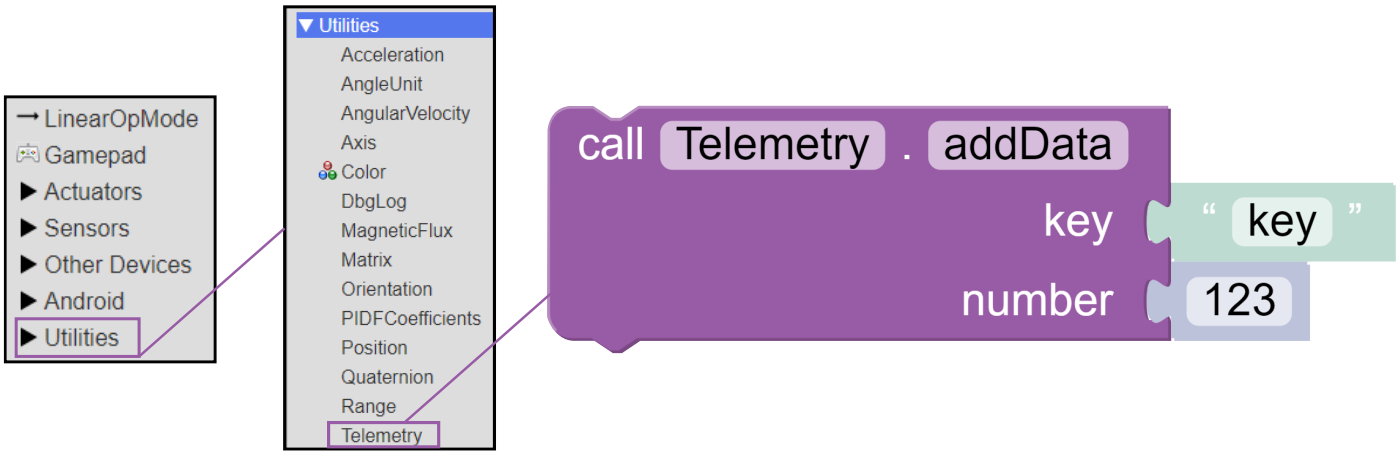
Drag the negative symbol block so it snaps in place between the  and  blocks.

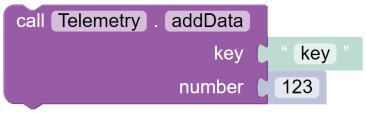
Motors and Telemetry

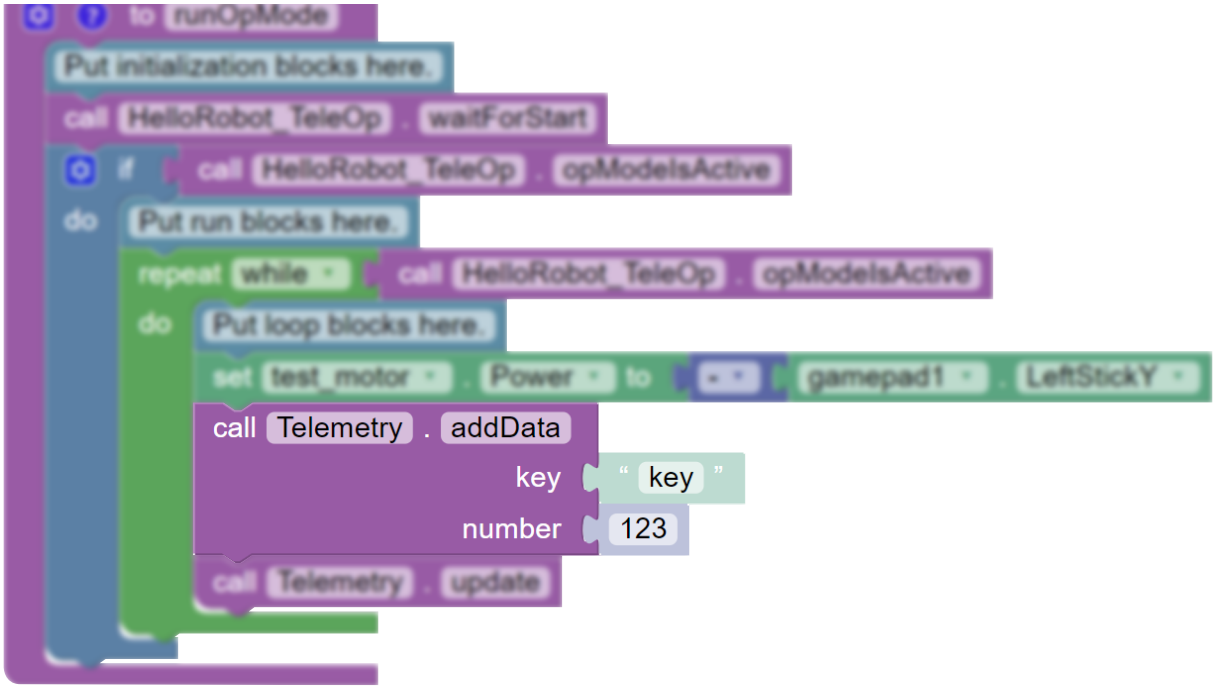
Recall that **telemetry** is the process of collecting and transmitting data. In Robotics telemetry is used to output internal data from actuators and sensors to the Driver Station. This data can then be analyzed by users to make decisions that can improve code.

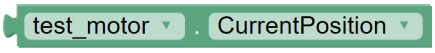
In order to gain telemetry data from the motor, motor encoders need to be used. REV DC Motors, like the Core Hex Motor, are equipped with internal encoders that relay information in the form of counts.

In order to access the telemetry blocks select the **Utilities** drop down. The utilities drop down is in alphabetical order, so telemetry is towards the bottom of the drop down options. Select the  block from the telemetry menu.



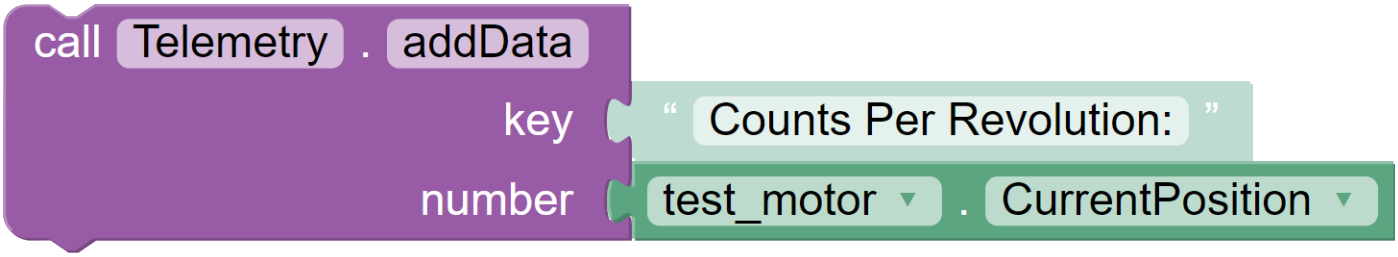
Drag the  block and place it beneath the  block set.



From the DC Motor menu pullout the block 

. Drag the Block and attach it to the number **parameter** on the telemetry blocks.

Change the key parameter to "Counts Per Revolution: "



When the op mode is run the telemetry block will display the current position information will be displayed with the Counts Per Revolution Key. The number that corresponds with the current position will change as the motor shaft position is changed.

> ⓘ For more information on programming encoders check out the Using Encoders page. For more information the counts per revolution metric and how to use it check out the Encoders page.

Programming Sensors

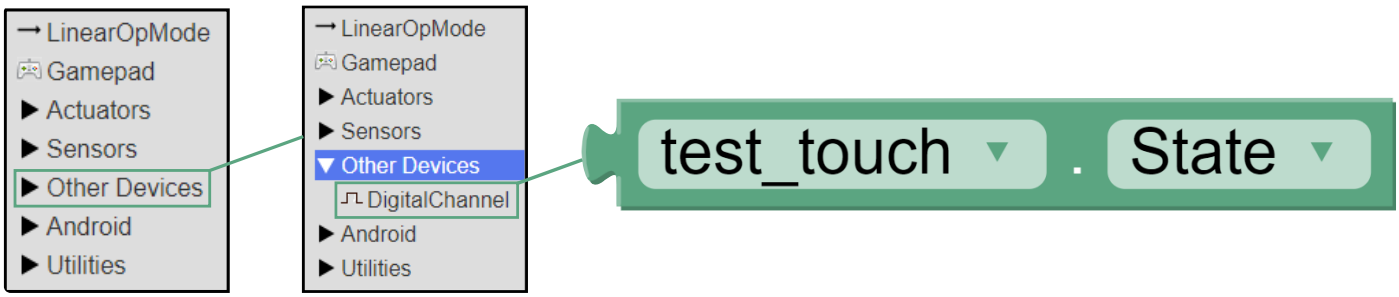
Touch Sensor Basics

> Modify your op mode to add the digital device related code. This can be done by clearing out your current code modifications or adding the digital device code to your op mode.

The goal of this section is to cover some of the basics of programming a digital device, or Touch Sensor, within Blocks.

Since this section will focus on digital devices it is important to understand how to access digital device specific blocks. At the top of the Categorize Blocks section there is a drop down menu for **Other Devices**. When the menu is selected it will drop down an option for **Digital Devices.** Selecting Digital Devices will open a side window filled with various digital device related blocks. The one that will most commonly be used is 

.



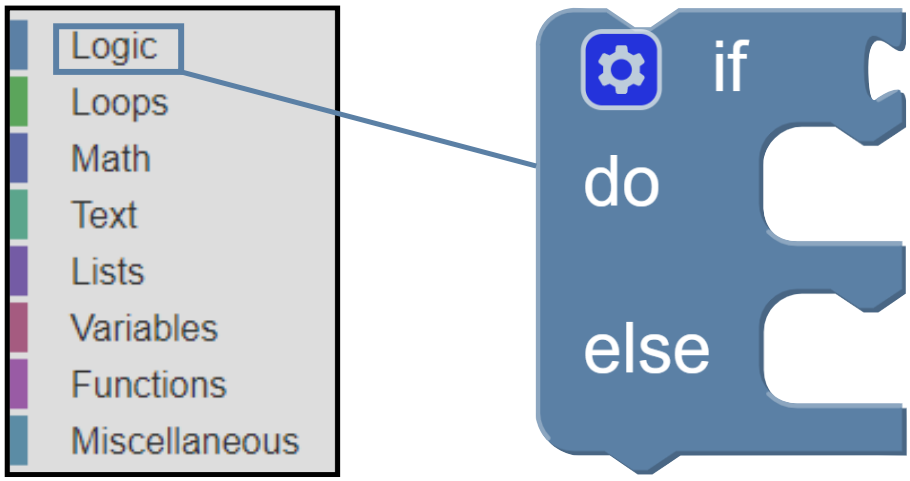
> Before programming with a Touch Sensor or other digital device it is important to understand what a digital device is and what the common applications for digital devices are. Visit the Digital Sensors page for more info.

Programming a Digital Device

The information from digital devices comes in two states, also known as binary states. The most common way to utilize this information is to use a conditional statement like an `if/else` statement.

From the **Logic** Menu in Blocks select the 

block.



Drag the  block and place it beneath the  comment.

Select a  block from the Digital Devices menu and add it to the *if/do/else* block as shown in the image below.

The  block stores the binary `FALSE/TRUE` information from the touch sensor and acts as the condition for the  statement. If  is true, any code placed in the do portion of the block will be activated. If  is false anything placed in the else portion of the clock will be activated

The `FALSE/TRUE` state of a REV Touch Sensor corresponds with whether or not the button on the Touch Sensor is pressed. When the button is not pressed the state of the Touch Sensor is True. When the button is pressed the state of the Touch Sensor is False,

To help remember how the physical and digital states of the sensor correspond in the next few sections lets use some comments.

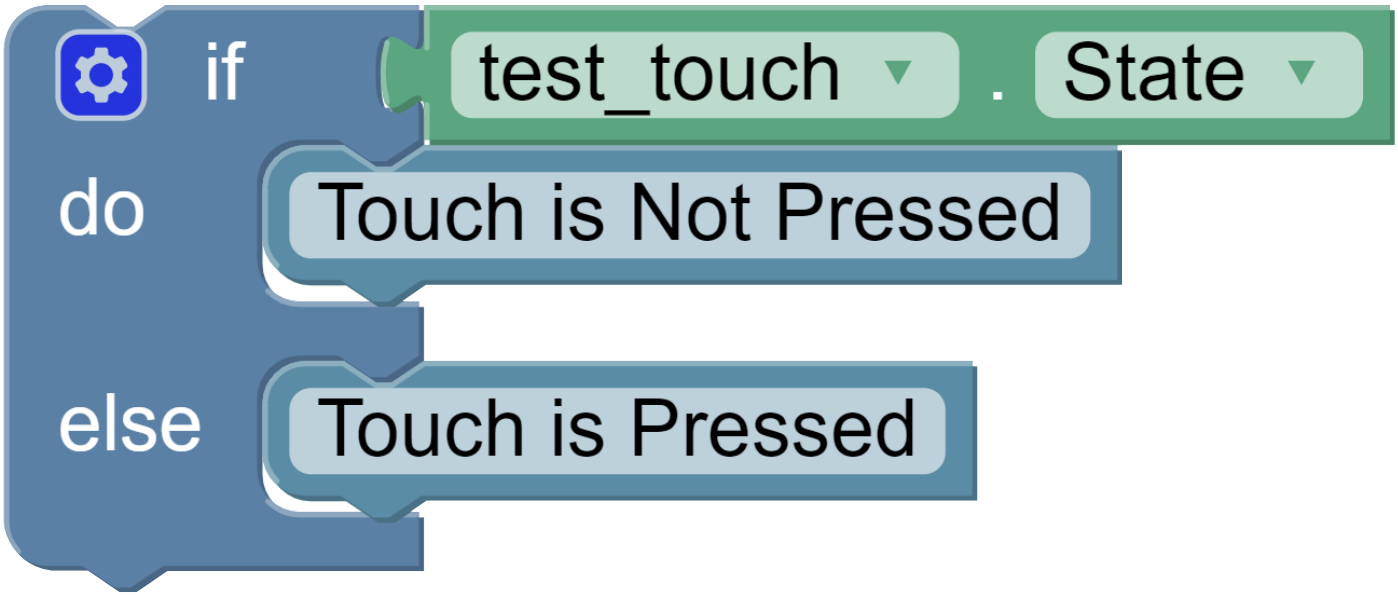
Comment blocks can be found in the **Miscellaneous** menu.

Place one comment block in the do portion of the 

block and change the comment to say 

Add another comment to the else portion of the block and change that comment to say 

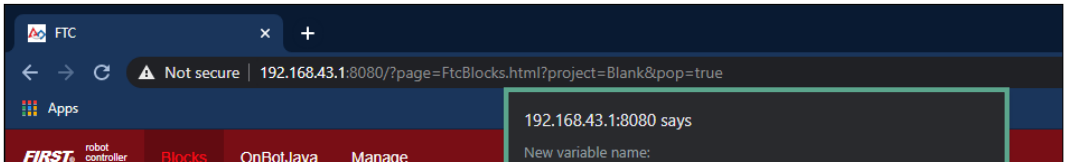
, as shown in the image below.

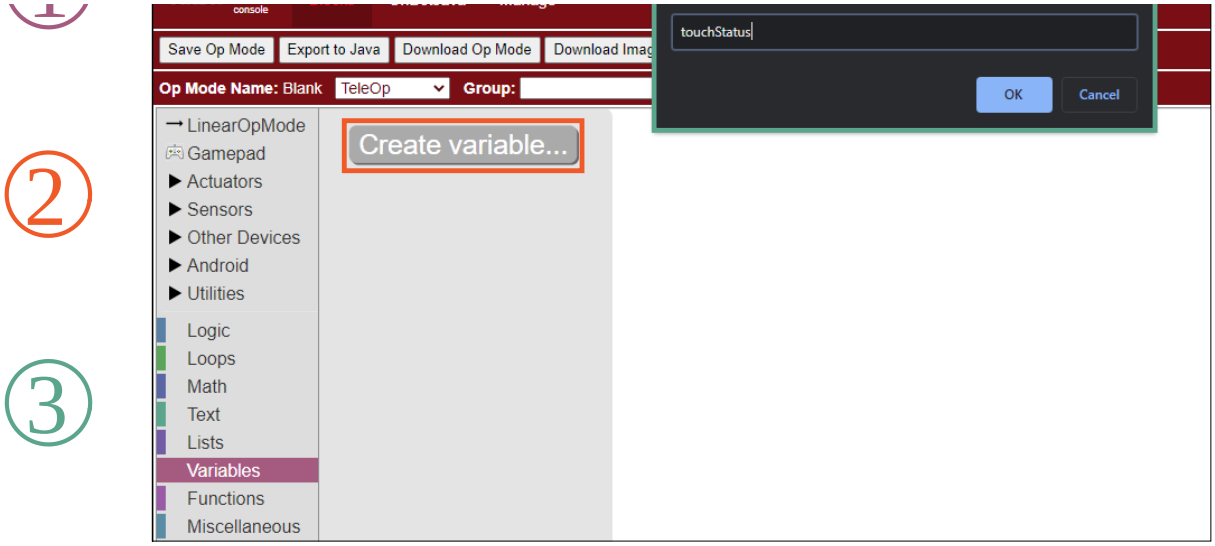


The next step in the process is to use telemetry to display the status of the Touch Sensor on the Driver Station phone. To do this, lets create a **string** variable called `touchStatus`.

> String refers to data that consists of a sequence of characters.

1

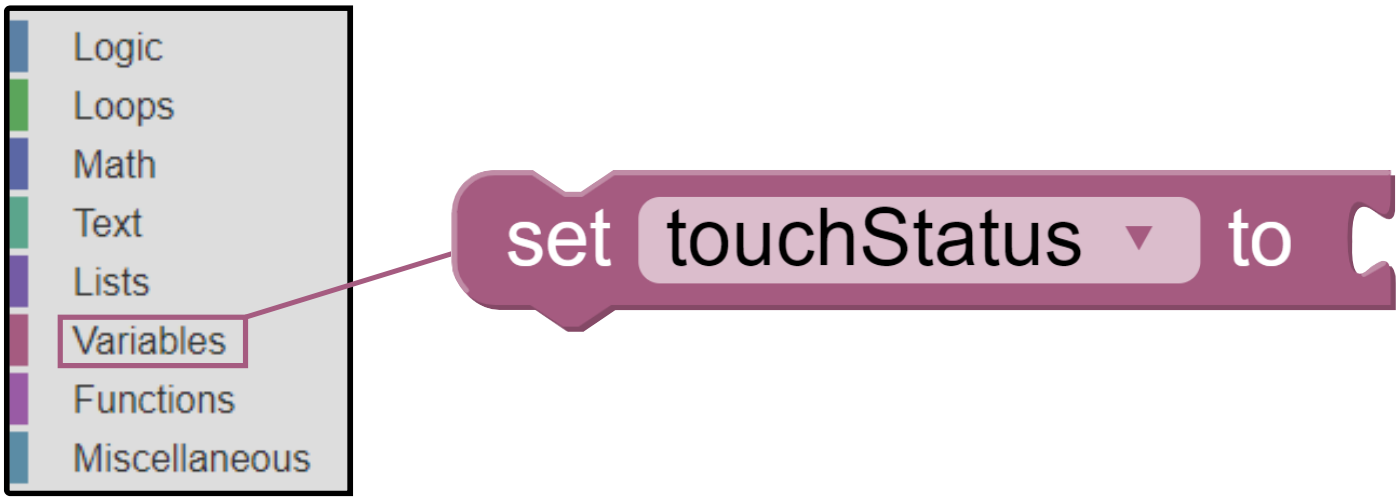




1. Click on the **Variables** menu. This will open a side window
2. Select the *Create variable...* block
3. A prompt from the FIRST Robot Controller will appear asking for a name for the variable. Name the variable `touchStatus`. Click okay

This process created a variable named `touchStatus`. Currently `touchStatus` is undefined, in order to define it the 

block needs to be used. This block can be found in the **Variables** menu now that the variable has been created.

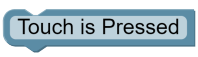


Drag a 

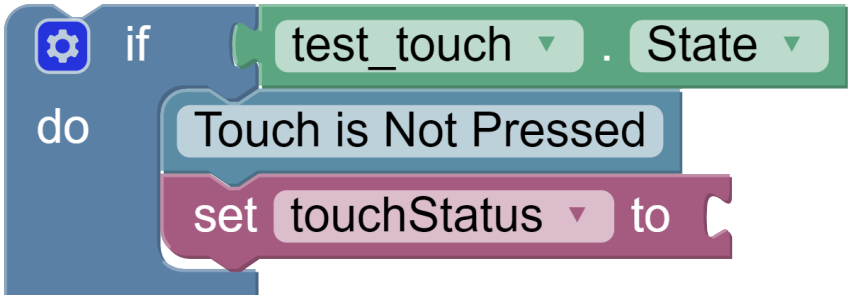
block and place it beneath the 

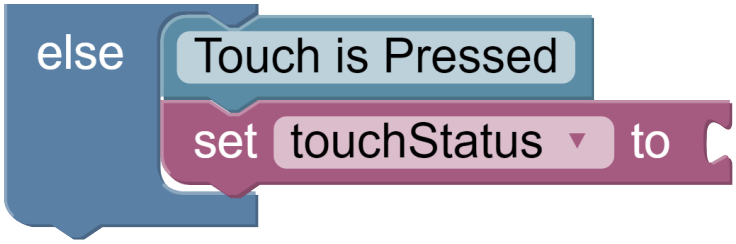
comment block. Add another 

block to the 

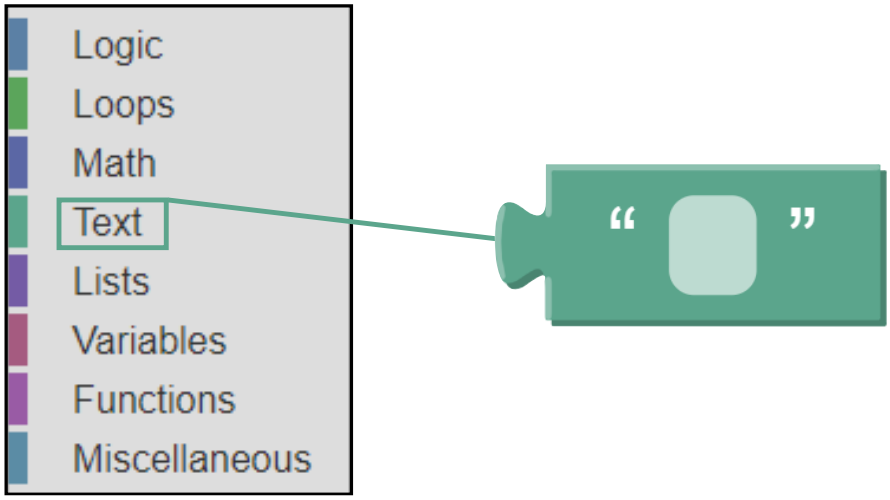
block set under the 

comment.

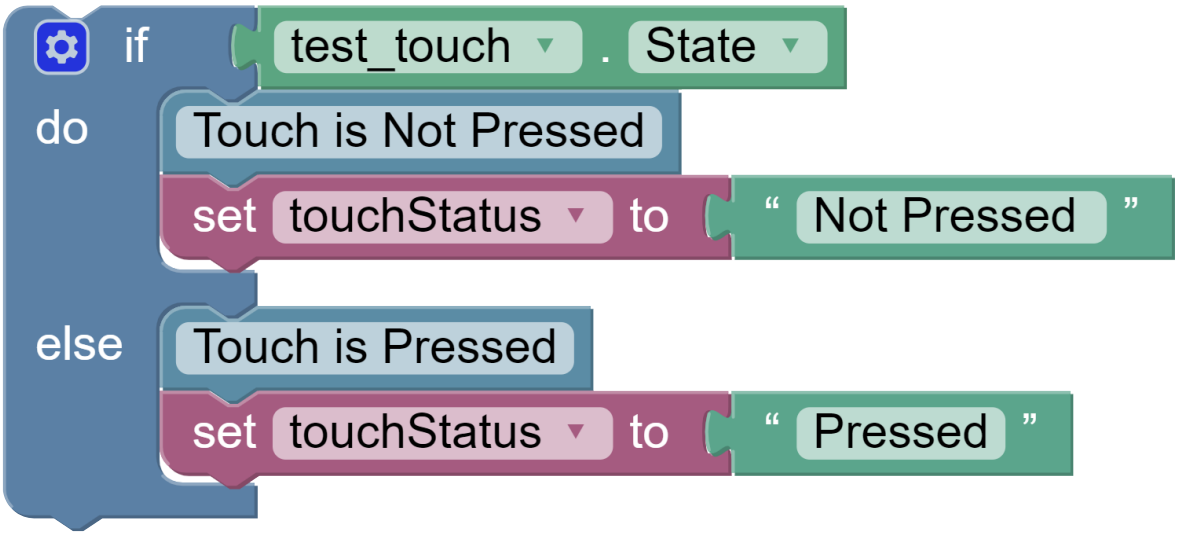


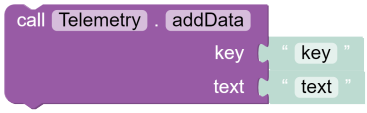


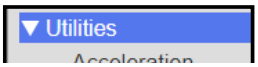
The  block allows you to define the `touchStatus` variable. Depending on what the status is of the touch sensor is, `touchStatus` will be set to a different string. For this select the string  block from the **Text** menu, as seen in the image below.

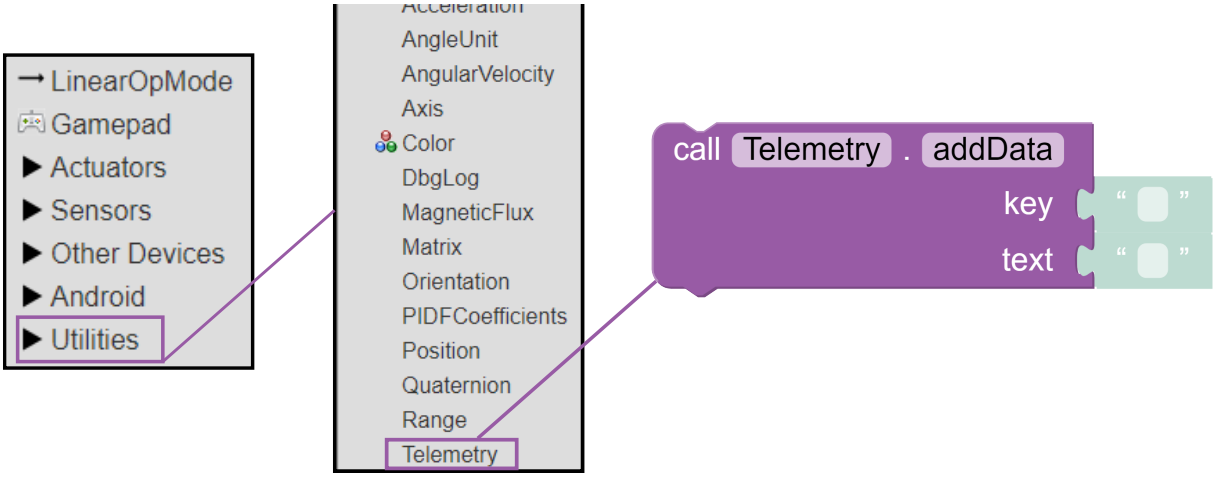


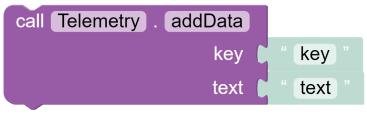
Attach a string block to both  blocks. Fill the  blocks with a status message that relates to the state of the Touch Sensor. For instance,  and  .



To display this information on the Driver Station phone telemetry must be used. In order to access the telemetry blocks select the **Utilities** drop down. The utilities drop down is in alphabetical order, so telemetry is towards the bottom of the drop down options. Select the  block from the telemetry menu.

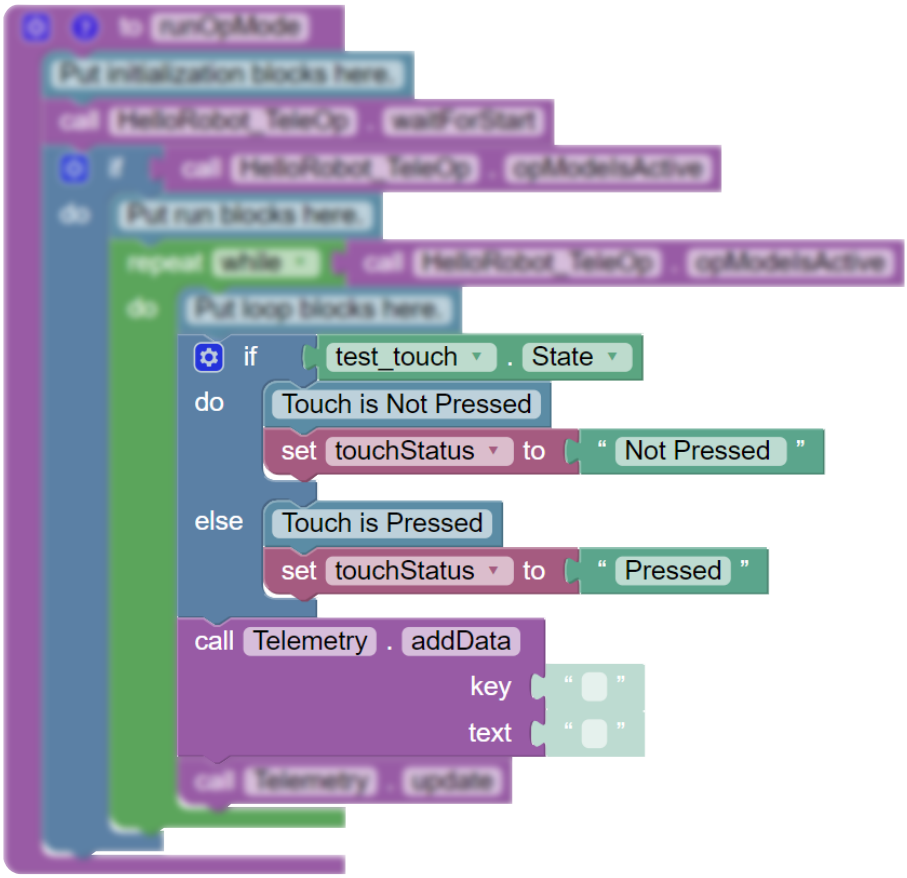




Drag the 

block and place it beneath the 

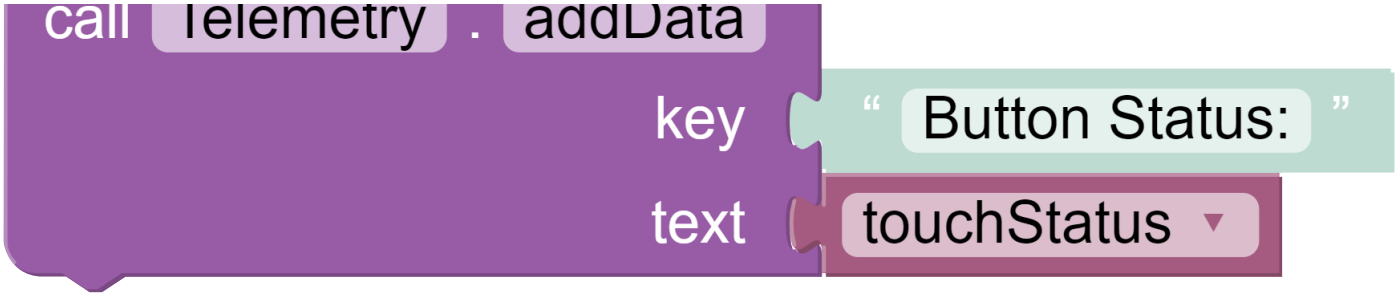
block set.



From the Variables menu select the 

block. Drag the Block and attach it to the text **parameter** on the telemetry block.

Change the key parameter to "Button Status: "

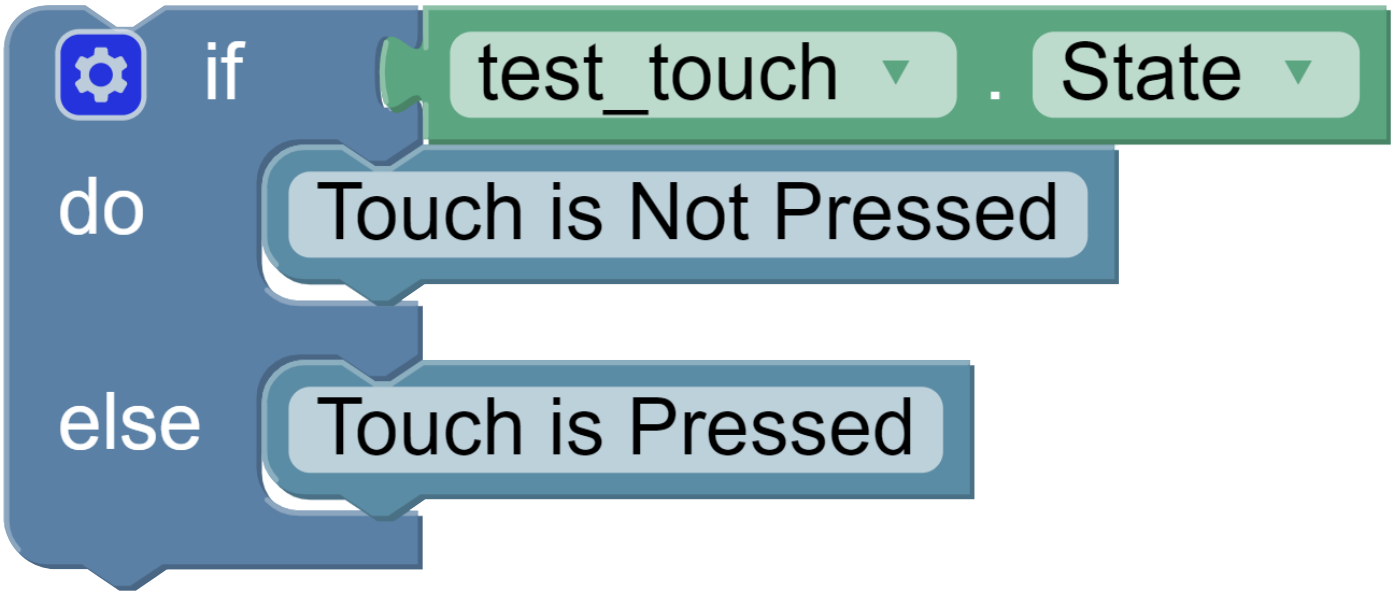


When this program is run the `touchStatus` telemetry will appear on the Driver Station phone. The `touchStatus` information will change based on the state of the Touch Sensor button.

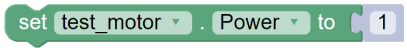
Digital Devices as Limit Switches

One of the most common uses for a digital device like a touch sensor is to use it as a limit switch. The intent of a limit switch is to stop a mechanism, like an arm or lift, before it exceeds its physical limitations. In this application power needs to be cut from the motor when the limit is met.

The concept of a limit switch involves many of the same steps from the previous section on programming a digital device. For that reason lets pick up from the following block set:

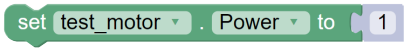
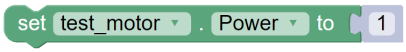
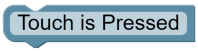


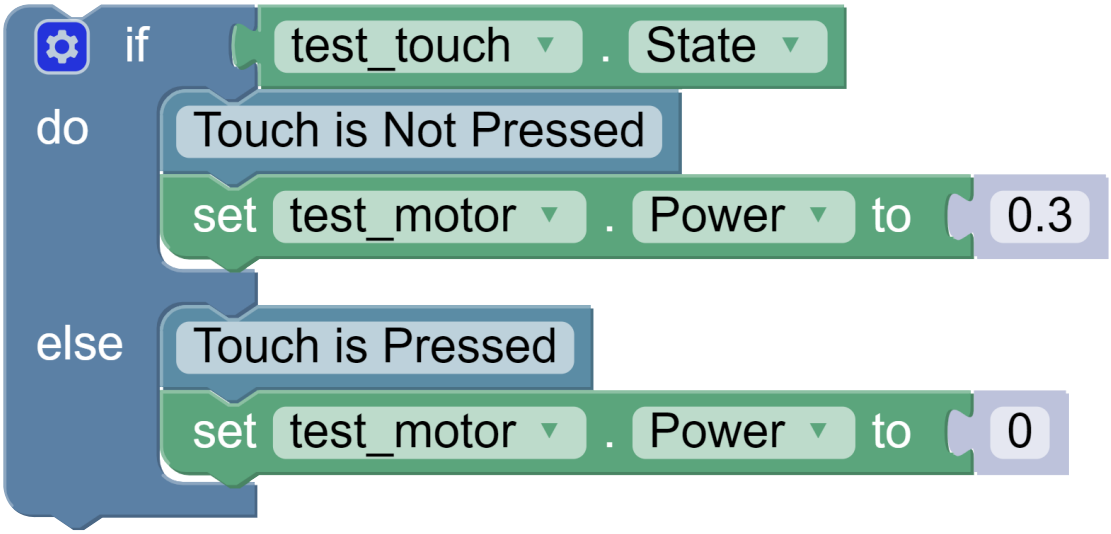
The 

block establishes a conditional environment for the limit switch. If the touch sensor is not pressed the motor can run, however, if it is pressed the motor can not run. To add this to the code the 

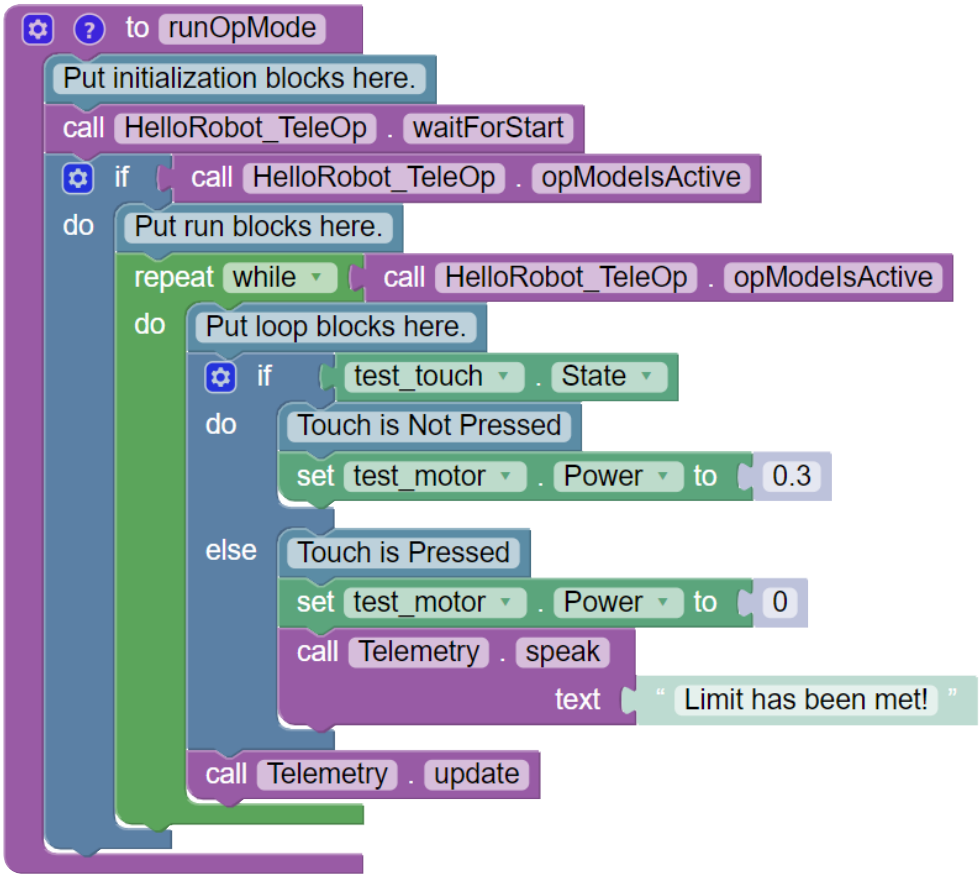
block needs to be used.

For information on where to find motor specific blocks please revisit the motor section.

Add a  block under the  comment. Change the power to 0.3. Add another  block under the  comment. Change the power to 0.



This  block introduces the basics of a limit switch. Like with most sensors, its good to have telemetry that updates the Driver Station on the status of the sensor. Consider the code from the previous section, or the following code as potential ideas for telemetry.



Test Bed - OnBot Java

OnBot Java is a text-based programming tool that lets programmers use a web browser to create, edit and save their Java op modes. In this section users can learn how to create an op mode, as wells as the basics of programming the actuators and sensors featured on the test bed.

Follow the guide in order to get an in depth understanding of working with OnBot Java or navigate to the section that fits your needs:

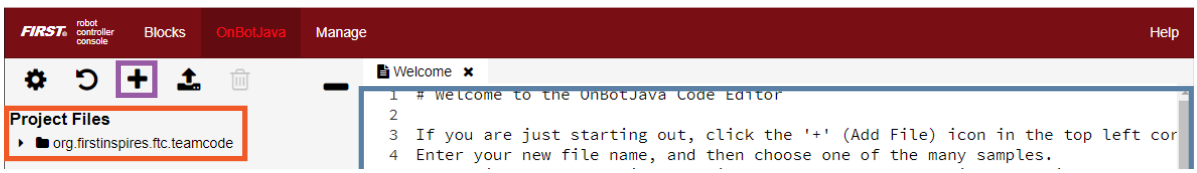
Section	Goals of Section
Creating an Op Mode	Focuses on how to navigate the OnBot Java interface and create an op mode.
Programming Essentials	Breaks down the structure and key elelments needed for an op mode, as well as some of the essential components of Java.
Programming Actuators	How to code servos and motors. This section wall through the basic logic of coding actuators, controlling actuators with a gamepad, and using telemetry.
Programming Sensors	How to code a digital device. The section focuses on the basic logic of coding a digital device, like a REV Touch Sensor.

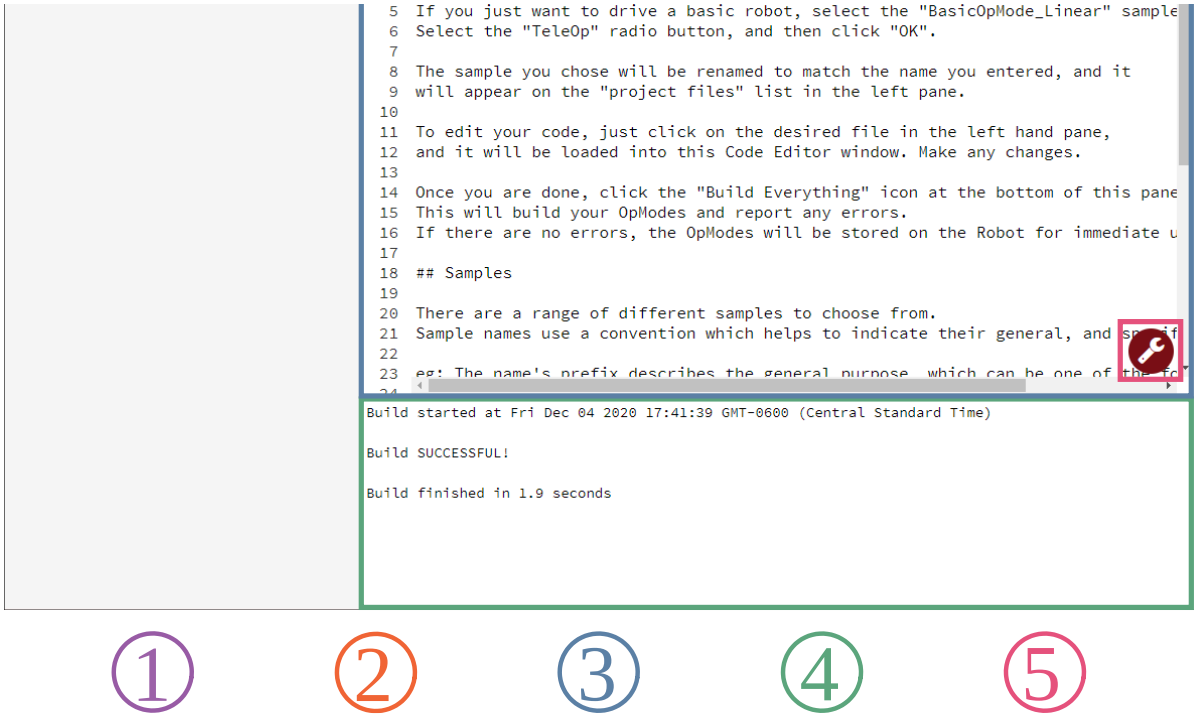
Creating an Op Mode

Before diving in and creating your first op mode, you should consider the concept of **naming conventions**. When writing code the goal is to be as clear as possible about what is happening within the code. This is where the concept of naming conventions comes into play. Common naming conventions have been established by the programming world to denote variables, classes, functions, etc. Op modes share some similarities to **classes.** Thus the naming convention for op modes tends to follow that naming convention for classes; where the first letter of every word is capitalized.

> This section assumes that you have already accessed the OnBot Java platform during the Hello Robot - Introduction to Programming. If you are unsure how to access OnBot Java please revisit this section before proceeding.

To start, access the Robot Controller Console and go to the OnBot Java page. There are a few key things to take note of on the main Onbot Java page.



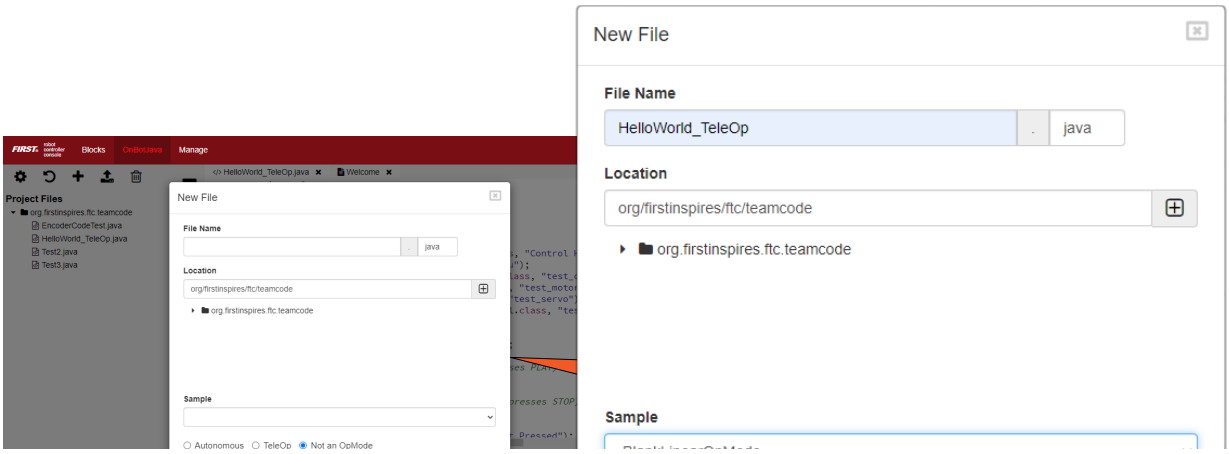


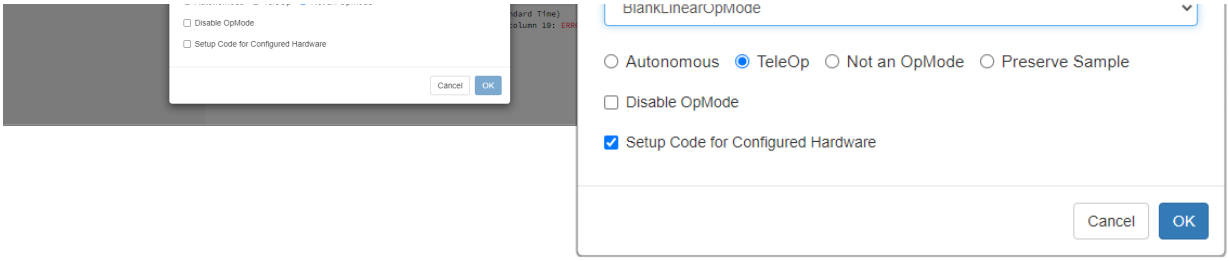
1. **Create New Op Mode** - The plus sign button opens up a window to create a new op more.
2. **Project Browser Pane** - This pane shows all the java project files on the Robot Controller.
3. **Source Code Editing Pane** - This pane is the main code editing area.
4. **Message Pane** - This pane provides messages on the success or failure of code builds.
5. **Build Everything** - Builds all of the .java files on a Robot Controller.

> When an op mode is created or edited the OnBot Java editor will auto-save the .java file to the file of system of the Robot Controller. However, in order to execute the code on the Robot Controller the .java text file needs to be converted to a binary that can be loaded dynamically onto the FTC Robot Controller app. This conversion is done by building the op modes.

Select the ***Create New Op Mode*** button. This will open the ***New File*** window. This window allows users to choose settings like: naming their op modes, selecting a sample code to build off of, or choosing op mode type.

For this guide select the following sections:





- File Name: HelloRobot_TeleOp
- Sample: BlankLinearOpMode
- Op Mode Type: TeleOp
- Setup for Configured Hardware: on

Once the proper settings have been choose, select "OK" to create the op mode. The new file will populate the Project Browser Pane.

Programming Essentials

During the process of creating an op mode the Onbot Java tool had several options to choose from. Those options define what information is already included in the op mode, which can simplify what a programmer has to do on their end. For instance, an option was given to select a sample. In OnBot Java these samples act as templates; providing statements, logical structure, and syntax for different robotics use cases.

In the previous section the following settings were selected: the *Setup Code for Configured Hardware* option, the *TeleOp* option, and a sample code called **BlankLinearOpMode.** These options combined setup the shell of code needed to have a functional op mode.

An op mode is considered a set of instructions for a robot to follow in order to understand the world around it. Even though the SDK provides readily available op mode structures, understanding what concepts the template is utilizing, and why, helps increase programming knowledge. Follow through this section to learn more about the op mode template and the programming concepts that make up its structure.

```
package org.firstinspires.ftc.teamcode;

import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.hardware.Blinker;
import com.qualcomm.robotcore.hardware.Gyroscope;
import com.qualcomm.robotcore.hardware.ColorSensor;
import com.qualcomm.robotcore.hardware.Servo;
import com.qualcomm.robotcore.hardware.DigitalChannel;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.eventloop.opmode.Disabled;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.DcMotorSimple;
import com.qualcomm.robotcore.util.ElapsedTime;

@TeleOp
```

```
public class HelloWorld_TeleOp extends LinearOpMode {
    private Gyroscope imu;
    private ColorSensor test_color;
    private DcMotor test_motor;
    private Servo test_servo;
    private DigitalChannel test_touch;


    @Override
    public void runOpMode() {
        imu = hardwareMap.get(Gyroscope.class, "imu");
        test_color = hardwareMap.get(ColorSensor.class, "test_color");
        test_motor = hardwareMap.get(DcMotor.class, "test_motor");
        test_servo = hardwareMap.get(Servo.class, "test_servo");
        test_touch = hardwareMap.get(DigitalChannel.class, "test_touch");

        telemetry.addData("Status", "Initialized");
        telemetry.update();
        // Wait for the game to start (driver presses PLAY)
        waitForStart();

        // run until the end of the match (driver presses STOP)
        while (opModeIsActive()) {
            telemetry.addData("Status", "Running");
            telemetry.update();

        }
    }
```

> The code block provides the structure of the template op mode based on the Hello Robot Configuration and with some comments missing. If another configuration is being used the code will be slightly different but many of the underlying concepts are the same.

Programming Concepts

At the start of the op mode there is an **annotation** that occurs before the class definition. This annotation states that this is a tele-operated (i.e., driver controlled) op mode:

```
@TeleOp
```

In Java annotations are metadata, or descriptive information about the code. In this case the annotation is being used to tell the system that this op mode is tele-operated. Changing the annotation from `@TeleOp` to `@Autonomous` will change the code to an autonomous op mode.

```
public class HelloWorld_TeleOp extends LinearOpMode {
```

You can also see that the OnBot Java editor created five **private member variables** for this op mode. These variables will hold references to the five configured devices that the OnBot Java editor detected in the active configuration.

```
    private Gyroscope imu;
    private ColorSensor test_color;
    private DcMotor test_motor;
    private Servo test_servo;
    private DigitalChannel test_touch;
```

Next, there is an **overridden method** called `runOpMode`. Every op mode of type `LinearOpMode` must implement this method. This method gets called when a user selects and runs the op mode.

```
    @Override
    public void runOpMode() {
```

Hardware mapping was introduced in the configuration section, as a two part process. The first part of the process was creating a configuration file. The second part of the process is retrieving references to hardware devices from the `hardwareMap` **object**.

> ⓘ The `hardwareMap` object is available to use in the `runOpMode` method. It is an object of type `hardwareMap` class.

At the start of the `runOpMode` method, the op mode uses the `hardwareMap` object to get references to the hardware devices that are listed in the Robot Controller's configuration file:

```
        imu = hardwareMap.get(Gyroscope.class, "imu");
        test_color = hardwareMap.get(ColorSensor.class, "test_color");
        test_motor = hardwareMap.get(DcMotor.class, "test_motor");
        test_servo = hardwareMap.get(Servo.class, "test_servo");
        test_touch = hardwareMap.get(DigitalChannel.class, "test_touch");
```

The `hardwareMap.get()` **method call** is used to retrieve the hardware devices and assign them to variables. The method call accepts two **arguments**: a reference to the particular class of hardware devices the device belongs to and the name of the hardware device in the configuration file. The name in the `hardwareMap.get()` needs to match the name of the device in the configuration file. If the names do not match, the op mode will throw a runtime error indicating that it can not find the device.

> ⓘ For more information on the runtime error check out the Common Errors in Hardware Mapping section.

In the next few statements of the example, the op mode prompts the user to push the start button to continue. It uses another object that is available in the `runOpMode` method. This object is called **telemetry** and the

op mode uses the `addData` method to add a message to be sent to the Driver Station. The op mode then calls the update method to send the message to the Driver Station. Then it calls the `waitForStart` method, to wait until the user pushes the start button on the driver station to begin the op mode run.

> **Telemetry** is the process of collecting and transmitting data. In Robotics telemetry is often used to output internal data from actuators and sensors to the Driver Station. This data can then be analyzed by users to make decisions that can improve code.

```
        telemetry.addData("Status", "Initialized");
        telemetry.update();
        // Wait for the game to start (driver presses PLAY)
        waitForStart();
```

> All linear op modes should have a `waitForStart` statement to ensure that the robot will not begin executing the op mode until the driver pushes the start button.

After a start command has been received, the op mode enters a **while loop** and keeps iterating in this loop until the op mode is no longer active (i.e., until the user pushes the stop button on the Driver Station):

```
        // run until the end of the match (driver presses STOP)
        while (opModeIsActive()) {
            telemetry.addData("Status", "Running");
            telemetry.update();

        }
```

As the op mode iterates in the while loop, it will continue to send telemetry messages with the index of “Status” and the message of “Running” to be displayed on the Driver Station.

Syntax

Programming languages, much like any language, have a set of guiding rules and principals that allow statements to be universally understood. Things like punctuation, word structure, and formatting all play a part in how a line of code is interpreted. In linguistics and computer science the rules that govern the structure of a sentence are known as syntax.

It is important to understand the syntax for Java, as syntax errors will be common and hard to track without a basic level of understanding.

Object Oriented Programming

This section dropped a lot of references to methods, object, and classes. These are all intermediate to advance programming topics often centered around the concept of object oriented programming. The purpose of the Hello Robot guide is to act as a introductory course to robotics programming rather than deep dive into programming concepts.

However, keep object oriented programming in mind as your skills grow. For now the most important thing to know is that occasionally methods within the SDK libraries will need to be called in order to perform a certain task. For instance, the line `HelloRobot_TeleOp.opModeIsActive()` line calls the method `opModeIsActive`, which is the procedure in the SDK that is able to tell when the op mode has been activate by the driver station phone.

Going forward many of the motor, servo, or sensor specific code will deal with calls to other methods or classes.

> For more information on classes and methods in the SDK check out the Java Doc.

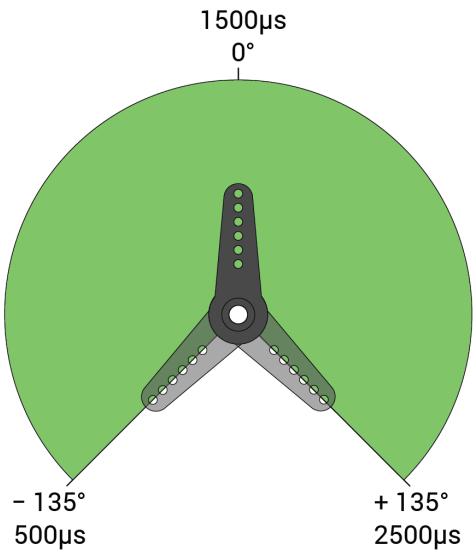
Programming Actuators

Servo Basics

The goal of this section is to cover some of the basics of programming a servo within OnBot Java. By the end of this section users should be able to control a servo with a gamepad, as well as understand some of the key programming needs of the servo.

> This section is considering the Smart Robot Servo in its default mode. If your servo has been changed to function in continuous mode or with angular limits it will not behave the same using the code examples below. You can learn more about the Smart Robot Servo or changing the Servo's mode via the SRS Programmer by clicking the hyperlinks.

With a typical servo, you can specify a target position for the servo. The servo will turn its motor shaft to move to the target position, and then maintain that position, even if moderate forces are applied to try and disturb its position.



For both Blocks and OnBot Java, you can specify a target position that ranges from 0 to 1 for a servo. For a servo with a 270° range, if the input range was from 0 to 1 then a signal input of 0 would cause the servo to turn to point -135°. For a signal input of 1, the servo would turn to +135°. Inputs between the minimum and maximum have corresponding angles evenly distributed between the minimum and maximum servo angle. This is important to keep in mind as you learn how to code servos.

Programming a Servo

Add the line `test_servo.setPosition(1);` to the op mode while loop.

```
        while (opModeIsActive()) {
            test_servo.setPosition(1);
            telemetry.addData("Status", "Running");
            telemetry.update();

        }
```

Select **Build Everything** to build the code.

> Try running this op mode on the test bed two times and consider the following questions:
>
> - Did the servo move during the first run?
> - Did the servo move during the second run?
>
> If the servo did not move switch the `test_servo.setPosition(1);` to `test_servo.setPosition(0);` and try again.

The intent of the `test_servo.setPosition();` is to set the position of the servo. If the servo is already in the set position when a code is run, it will not change positions. Lets try adding the line `test_servo.setPosition(0);` to the code in the initialization section.

```
public void runOpMode() {
        imu = hardwareMap.get(Gyroscope.class, "imu");
        test_color = hardwareMap.get(ColorSensor.class, "test_color");
        test_motor = hardwareMap.get(DcMotor.class, "test_motor");
        test_servo = hardwareMap.get(Servo.class, "test_servo");
        test_touch = hardwareMap.get(DigitalChannel.class, "test_touch");

        test_servo.setPosition(0);

        telemetry.addData("Status", "Initialized");
        telemetry.update();
        // Wait for the game to start (driver presses PLAY)
        waitForStart();

        // run until the end of the match (driver presses STOP)
```

```
        while (opModeIsActive()) {
            test_servo.setPosition(1);
            telemetry.addData("Status", "Running");
            telemetry.update();

        }
    }
}
```

> Try running this op mode on the test bed. Give some time between hitting init and hitting play and consider the following question:
>
> - What is different from the previous run?

The `test_servo.setPosition(0);` that was added in the step above changes the servo position to 0 during the initialization phase, so when the op mode is run the servo will always move to position 1. For some applications starting the servo in a **known state**, like at position zero, is beneficial to the operation of a mechanism. Setting the servo to the known state in the initialization ensures it is in the correct position when the op mode runs.

Programming a Servo with a Gamepad

The focus of this example is to assign certain servo positions to buttons on the gamepad. For this example the known state will stay at position 0, so that after initialization the servo will be a the -135 degree position of the servo range. The following list shows what buttons correspond with which servo position.

Button	Degree Position	Code Position
Y	-135	0
X	0	0.5
B	0	0.5
A	135	1

The best way to switch the servo position will be to use a conditional `if/ else if` statement. An if statement considers whether a conditional statement is true or false. If the conditional statement is true a defined action (like the servo moving) is performed. If the conditional statement is false the action is not performed.

An `if/else if` statement takes in multiple different conditional statements. If the first conditional statement is found to be false then the second conditional state is analyzed. To better understand this concept consider the following code:

```
if (gamepad1.y){
    //move to -135 degrees
```

```
    test_servo.setPosition(0);

} else if (gamepad1.x || gamepad1.b) {
    //move to 0 degrees
    test_servo.setPosition(0.5);

} else if (gamepad1.a) {
    //move to 135 degrees
    test_servo.setPosition(1);

}
```

There are three different paths in this `if/else if` statement. If the first conditional statement is true (the Y button is pressed) the servo moves to code position 0 and the other conditional statements are ignored. If the first condition is false (the Y button is not pressed) the second condition is analyzed. This behavior repeats until a condition is met or all conditions have been tested and found false.

> ⓘ `||` is a logical operator in Java. This symbol is the Java equivalent of "or." Using this in a conditional statement says that either button x or button b can be pressed for this condition to be considered true.

```
public void runOpMode() {
        imu = hardwareMap.get(Gyroscope.class, "imu");
        test_color = hardwareMap.get(ColorSensor.class, "test_color");
        test_motor = hardwareMap.get(DcMotor.class, "test_motor");
        test_servo = hardwareMap.get(Servo.class, "test_servo");
        test_touch = hardwareMap.get(DigitalChannel.class, "test_touch");

        test_servo.setPosition(0);

        telemetry.addData("Status", "Initialized");
        telemetry.update();
        // Wait for the game to start (driver presses PLAY)
        waitForStart();

        // run until the end of the match (driver presses STOP)
        while (opModeIsActive()) {
            if (gamepad1.y){
                //move to -135 degrees
                test_servo.setPosition(0);

            } else if (gamepad1.x || gamepad1.b) {
                //move to 0 degrees
                test_servo.setPosition(0.5);

            } else if (gamepad1.a) {
                //move to 135 degrees
                test_servo.setPosition(1);
                        }
```

```
            telemetry.addData("Status", "Running");
            telemetry.update();

        }
    }
}
```

Servos and Telemetry

Recall that **telemetry** is the process of collecting and transmitting data. In Robotics telemetry is used to output internal data from actuators and sensors to the Driver Station. This data can then be analyzed by users to make decisions that can improve code.

The most useful telemetry from the servo is the the position of the servo along its 270 degree range. In order to get that information the following line needs to be used.

```
test_servo.getPosition();
```

In the programming essentials section the `telemetry.addData();` line was briefly discussed. This method call takes in a key and variable parameter and outputs the information to the Driver Station. The key is a string, or a line of text, that should define the variable. In this case the `telemetry.addData();` is being used to output the position of the servo as it is changed so the key can be "Servo Position" The parameter however will be the the `test_servo.getPosition();` method call.

```
double motorPower = 0;
while (opModeIsActive()) {
           if (gamepad1.y){
                //move to -135 degrees
                test_servo.setPosition(0);

            } else if (gamepad1.x || gamepad1.b) {
                //move to 0 degrees
                test_servo.setPosition(0.5);

            } else if (gamepad1.a) {
                //move to 135 degrees
                test_servo.setPosition(1);

            telemetry.addData("Servo Position", test_servo.getPosition());
            telemetry.addData("Status", "Running");
            telemetry.update();

        }
```

Motor Basics

> Modify your op mode to add the motor related code. This can be done by clearing out your current

> code modifications or adding the motor related code to your current op mode.

The goal of this section is to cover some of the basics of programming a motor within OnBot Java. By the end of this section users should be able to control a motor using a gamepad, as well as understand some of the basics of working with motor encoders.

Driving Motors

Add the line `test_motor.setPower(1);` to the op mode while loop.

```
        while (opModeIsActive()) {
            test_motor.setPower(1);

            telemetry.addData("Status", "Running");
            telemetry.update();

        }
```

Select **Build Everything** to build the code.

> Try running this op mode on the test bed and consider the following questions:
>
> - How fast is the motor running?
> - What happens if you change the power from **1** to **0.3**?
> - What happens if you change the power to **-1**?

The level of power sent to the motor is dependent on the numerical number assigned to the motor. The change from 1 to 0.3 decreased the motors speed from 100% of duty cycle to 30% of duty cycle. Meanwhile, the change to -1 allowed the motor to rotate at 100% duty cycle in the opposite direction. So, power can be fluctuated to drive a motor *forward* or *backwards*.

However, the `test_motor.setPower(1);` line will run the motor in the assigned direction until something in the code stops the motor or causes a change in direction.

Driving Motors with the Gamepad

In the previous section you learned how to set the motor to run at a specific power level in a specific direction. However, in some applications, it may be necessary to control the motor with a gamepad, to easily change the direction or power level of a mechanism.

For this section lets create a double variable `motorPower`. This variable will be created within the op mode but outside of the while loop.

```
public void runOpMode() {
        imu = hardwareMap.get(Gyroscope.class, "imu");
        test_color = hardwareMap.get(ColorSensor.class, "test_color");
        test_motor = hardwareMap.get(DcMotor.class, "test_motor");
```

```
        test_touch = hardwareMap.get(DigitalChannel.class, "test_touch");

        double motorPower = 0;

        telemetry.addData("Status", "Initialized");
        telemetry.update();
        // Wait for the game to start (driver presses PLAY)
        waitForStart();
        // run until the end of the match (driver presses STOP)
        while (opModeIsActive()) {

            telemetry.addData("Status", "Running");
            telemetry.update();

        }
    }
}
```

A **double** is numerical data type that can store numbers with decimal points. Since the power, or duty cycle, of the motor runs on a scale between 1 to -1; the `motorPower` variable will need to be able to hold numerical data with decimal points.

Consider the following lines of code:

```
  motorPower = - this.gamepad1.left_stick_y;
  test_motor.setPower(motorPower);
```

The line `motorPower = - this.gamepad1.left_stick_y;` takes an numerical input that corresponds with the position of the gamepad joystick as it moves along the y-axis and assigns it as the `motorPower` variable. The next line `test_motor.setPower(motorPower);` sets the motor power equal to the `motorPower` variable.

> Note that for the Logitech F310 gamepads, the Y value of a joystick ranges from -1, when a joystick is in its topmost position, to +1, when a joystick is in its bottommost position. In order to change the directional relationship between the motor and the joystick, so that the topmost position of the joystick correlates with the forward direction of the motor, a **negative symbol**, or negation operator needs to be used.

```
// run until the end of the match (driver presses STOP)
double motorPower = 0;
while (opModeIsActive()) {
            motorPower = - this.gamepad1.left_stick_y;
            test_motor.setPower(motorPower);

            telemetry.addData("Status", "Running");
            telemetry.update();

        }
```

Motors and Telemetry

Recall that **telemetry** is the process of collecting and transmitting data. In Robotics telemetry is used to output internal data from actuators and sensors to the Driver Station. This data can then be analyzed by users to make decisions that can improve code.

One of the most common forms of telemetry data from motors is the data pulled from the motor encoder. REV DC Motors, like the Core Hex Motor, are equipped with internal encoders that relays positional information in the form of counts. In order to get information from the encoders the following line needs to be used:

```
test_motor.getCurrentPosition();
```

In the programming essentials section the `telemetry.addData();` line was briefly discussed. This method call takes in a key and variable parameter and outputs the information to the Driver Station. The key is a string, or a line of text, that should define the variable. In this case the `telemetry.addData();` is being used to output the position of the motor in the form of encoder counts so the key can be "Encoder Value." The parameter however will be the the `test_motor.getCurrentPosition();` method call.

```
double motorPower = 0;
while (opModeIsActive()) {
            motorPower = - this.gamepad1.left_stick_y;
            test_motor.setPower(motorPower);

            telemetry.addData("Encoder Value", test_motor.getCurrentPosition());
            telemetry.addData("Status", "Running");
            telemetry.update();

        }
```

> ⓘ For more information on programming encoders check out the Using Encoders page. For more information the counts per revolution metric and how to use it check out the Encoders page.

Programming Sensors

Touch Sensor Basics

The goal of this section is to cover some of the basics of programming a digital device, or Touch Sensor, within Blocks.

> ⚠ Before programming with a Touch Sensor or other digital device it is important to understand what a digital device is and what the common applications for digital devices are. Visit the Digital

> Sensors page for more info.

Programming a Digital Device

> Modify your op mode to add the digital device related code. This can be done by clearing out your current code modifications or adding the digital device code to your op mode.

The information from digital devices comes in two states, also known as binary states. The most common way to utilize this information is to use a conditional statement like an `if/else` statement. The line `test_touch.getState();` collects the binary `FALSE/TRUE` state from the touch sensor and acts as the condition for the `if/else` statement.

```
if (test_touch.getState()){
    //Touch Sensor is not pressed
} else {
    //Touch Sensor is pressed
        }
```

The code above highlights the basics structure of the `if/else` statement for a digital device. The `FALSE/TRUE` state of a REV Touch Sensor corresponds with whether or not the button on the Touch Sensor is pressed. When the button is not pressed the state of the Touch Sensor is true. When the button is pressed the state of the Touch Sensor is false. This status is reflected by the comments in the code.

The most basic way to use a digital device is to use telemetry to output information, like the status of the Touch Sensor button. To do this, lets create a **string** variable called `touchStatus`. This variable will be created within the op mode.

> String refers to data that consists of a sequence of characters. String datatypes are indicated in code by a set of quotation marks. For instance, "Hello Robot" is a string but Hello Robot is not.

```
public void runOpMode() {
        imu = hardwareMap.get(Gyroscope.class, "imu");
        test_color = hardwareMap.get(ColorSensor.class, "test_color");
        test_motor = hardwareMap.get(DcMotor.class, "test_motor");
        test_servo = hardwareMap.get(Servo.class, "test_servo");
        test_touch = hardwareMap.get(DigitalChannel.class, "test_touch");

        String touchStatus = "";
```

The line `String touchStatus = "";` declares that the variable touchStatus is an empty string variable. Which means that `touchStatus` is currently holding a string with zero characters in it.

Add the `if/else` statement to the while loop.

```
public void runOpMode() {
        imu = hardwareMap.get(Gyroscope.class, "imu");
        test_color = hardwareMap.get(ColorSensor.class, "test_color");
        test_motor = hardwareMap.get(DcMotor.class, "test_motor");
        test_servo = hardwareMap.get(Servo.class, "test_servo");
        test_touch = hardwareMap.get(DigitalChannel.class, "test_touch");

        String touchStatus = "";

        telemetry.addData("Status", "Initialized");
        telemetry.update();

        // Wait for the game to start (driver presses PLAY)
        waitForStart();

        // run until the end of the match (driver presses STOP)
        while (opModeIsActive()) {

            if (test_touch.getState()){
                //Touch Sensor is not pressed
            } else {
                //Touch Sensor is pressed
                        }
            telemetry.addData("Status", "Running");
            telemetry.update();

        }
    }
}
```

Right now the variable `touchStatus` is empty, but for this example it should change to reflect the status of the touch sensor. To do this `touchStatus` should be set to either `"Not Pressed"` or `"Pressed"`.

```
 if (test_touch.getState()){
    //Touch Sensor is not pressed
    touchStatus = "Not Pressed";

 } else {
    //Touch Sensor is pressed
    touchStatus = "Pressed";
                        }
```

To display in the information assigned to `touchStatus`, telemetry needs to be used. In the programming essentials section the `telemetry.addData()` line was briefly discussed. This method call takes in a key and variable parameter and outputs the information to the Driver Station. The key is a string, or a line of text, that should define the variable. In this case the `telemetry.addData();` is being used to output changes in the `touchStatus` variable so `"Touch Status"` would be a good key. The parameter will be the `touchStatus` variable. Add this line above the `telemetry.update();` line in the while loop.

```
telemetry.addData("Touch Sensor", touchStatus);
```

Digital Devices as Limit Switches

One of the most common uses for a digital device like a touch sensor is to use it as a limit switch. The intent of a limit switch is to stop a mechanism, like an arm or lift, before it exceeds its physical limitations. In this application power needs to be cut from the motor when the limit is met.

Programming a limit switch requires the same *if/else* logic applied in the previous section. If the touch sensor state is true (it is not pressed) the motor will have power. Else (it is pressed) the motor will not have power.

```
 if (test_touch.getState()){
    //Touch Sensor is not pressed
     test_motor.setPower(0.3);

} else {
    //Touch Sensor is pressed
    test_motor.setPower(0);
                        }
```

The code block above introduces the basics of a limit switch. Like with most sensors, its good to have telemetry that updates the Driver Station on the status of the sensor. Consider the following code:

```
public void runOpMode() {
        imu = hardwareMap.get(Gyroscope.class, "imu");

        test_color = hardwareMap.get(ColorSensor.class, "test_color");
        test_motor = hardwareMap.get(DcMotor.class, "test_motor");
        test_servo = hardwareMap.get(Servo.class, "test_servo");
        test_touch = hardwareMap.get(DigitalChannel.class, "test_touch");

        String touchStatus = "";

        telemetry.addData("Status", "Initialized");
        telemetry.update();

        // Wait for the game to start (driver presses PLAY)
        waitForStart();

        // run until the end of the match (driver presses STOP)
        while (opModeIsActive()) {

         if (test_touch.getState()){
            //Touch Sensor is not pressed
           test_motor.setPower(0.3);
           touchStatus = "Not Pressed";

          } else {
            //Touch Sensor is pressed
```

```
            testMotor.setPower(0);
            touchStatus = "Pressed";
                        }

            telemetry.addData("Touch Sensor:", touchStatus);
            telemetry.addData("Status", "Running");
            telemetry.update();

        }
    }
}
```

Hello Robot - Robot Control

With the basics of controlling actuators and getting feedback from sensors is understood from Hello Robot - Test Bed, it is time to start configuring and programming our robot for Teleoperated and Autonomous control!

Section	Goals of Section
Create a Basic Robot	Introduces a potential robot to work with as well as the configuration file used in the following section
Drivetrain Basics	Differences between differential and omnidirectional drivetrains and their affect on teleoperated control types.

> ⓘ Before continuing it is recommended to complete, at minimum, a drivetrain. There are a few different options depending on the kit being used. For this guide the Class Bot V2 is used. Check out the build guide for full building instructions for the Class Bot V2!

Create a Basic Robot

The graphic below highlights the major hardware components of the Class Bot V2. These components are important to understand for the configuration process.

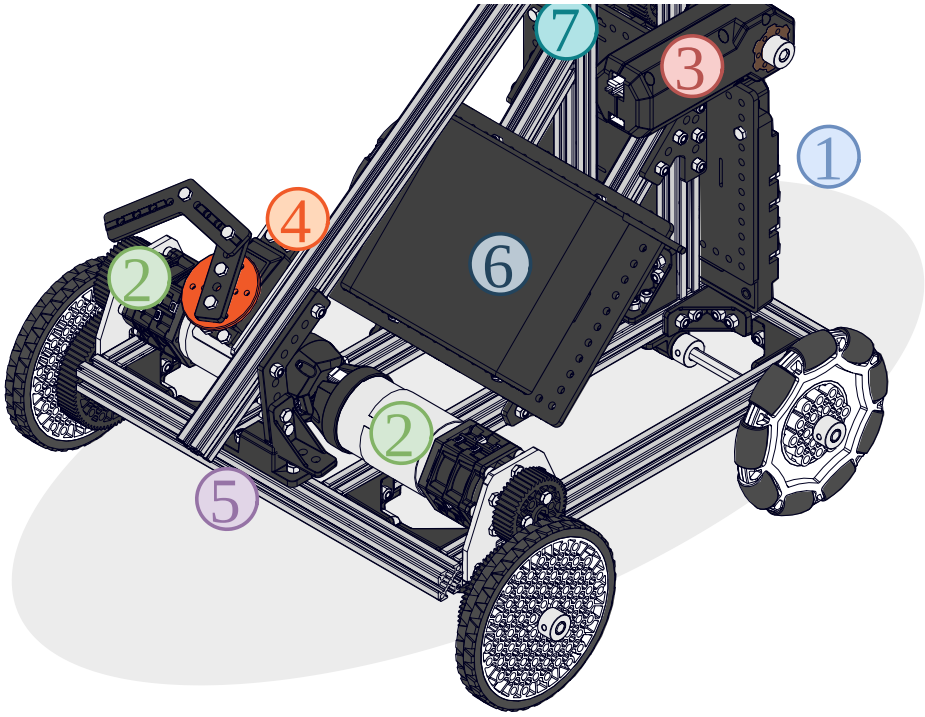
(3) Arm Motor

(4) Claw Servo

(5) Touch Sensor

(6) Battery

(7) Switch



The Hello Robot - Configuration section focused on configuring the components in the Test Bed. In order to continue forward with the Robot Control programming sections, a new configuration file must be made for the components on the robot. It is your choice what variable names you would like to assign to your robot, but for reference this guide will use the following names for each hardware component.

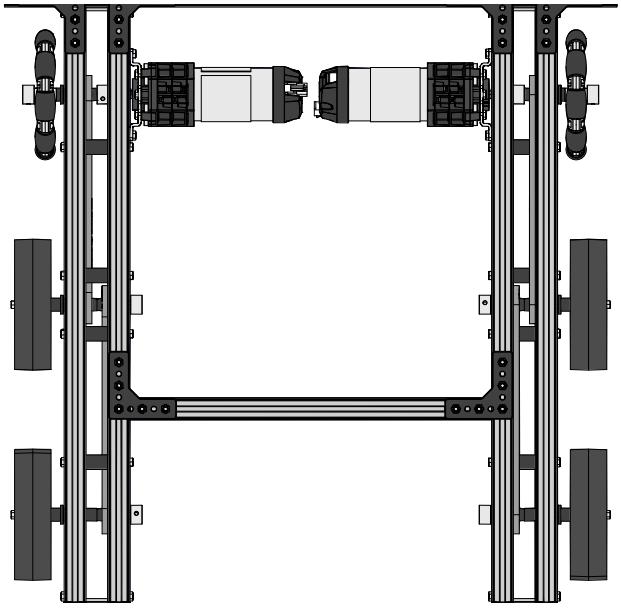
Hardware Component	Hardware Type	Name
Right Drive Motor	REV Robotics UltraPlanetary HD Hex Motor	rightmotor
Left Drive Motor	REV Robotics UltraPlanetary HD Hex Motor	leftmotor
Arm Motor	REV Robotics Core Hex Motor	arm
Claw Servo	Servo	claw
Touch Sensor	REV Touch Sensor	touch

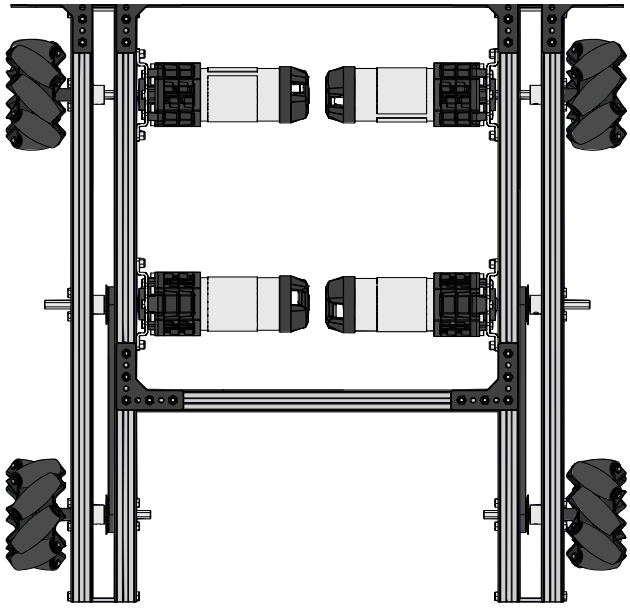
Drivetrain Basics

Before continuing it is important to understand the mechanical behavior of different drivetrains. The two most common drivetrain categories types are Differential and Omnidirectional. The Class Bot's drivetrain is a differential drivetrain. The table below highlights the main features of these two types of drivetrains.

Differential

Omnidirectional





Differential Drivetrains	Omnidirectional Drivetrains
Most common type of drivetrain	Can move in any direction due to rollers on the wheels
Moves along a central axis	Varies power to each wheel to change heading or strafe
Applies more power to one side of the drivetrain than the other to change heading	More complex programming
Can have different names depending on the number of motors, wheels, and wheel types used (4WD, 6WD, West Coast)	Requires more than 2 motors (depending on specific type and configuration)

Teleoperated Control Types

There are a number of different ways to control a robot teleoperated. When using the REV Control System this is done with a Driver Station Device and gamepads. There are various ways to use a controller to drive a differential drivetrain. Two of the conventional ways are **Tank Drive** and **Arcade Drive**.

Tank Drive

For tank drive, each side of the differential drivetrain is mapped to its own joystick. Changing the position of each joystick allows the drivetrain to steer and change its heading. Sample code exists in the Robot Controller Application to control a differential drivetrain in this way.

Arcade Drive

For arcade drive, each side of the differential drivetrain is controlled by a single joystick. Changing position of the joystick changes the power applied to each side of the drivetrain allowing for a given command. Arcade drives typically have left/right movement of the joystick set to spin the robot about its axis with

forward/back moving the robot forward and reverse. More information on Arcade drive are found in the Robot Navigation - Blocks and Robot Navigation - OnBot Java sections.

With the robot configured, a basic understanding of drivetrains, and teleoperated control types, we can move forward to programming the drivetrain to get the robot moving.

Robot Navigation - Blocks

Introduction to Robot Navigation

As alluded to in the Hello Robot - Robot Control section, robot control comes in many different forms. One of the control types to consider for robots with drivetrains, is robot navigation.

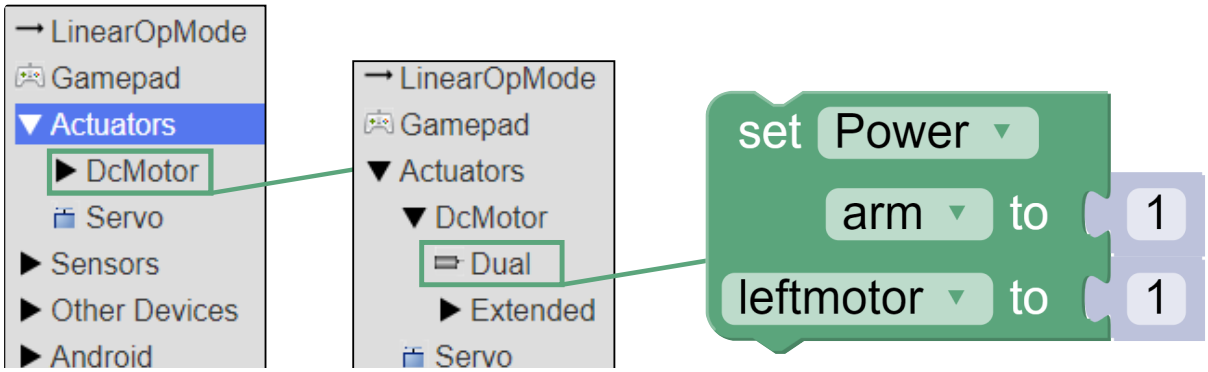
Robot navigation as a concept is dependent on the type of drivetrain and the type of operation mode. For instance, the code to control a mecanum drivetrain differs from the code used to control a differential drivetrain. There is also a difference between coding for teleoperated driving, with a gamepad, or coding for autonomous, where each movement of the robot must be defined within code.

The following section goes through some of the basics of programming for a differential drivetrain, as well as how to set up a teleoperated arcade style drivetrain code. The concepts and logic highlighted in this section are applicable to autonomous control, including the section Elapsed Time.

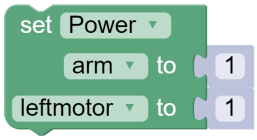
Sections	Goals of Section
Basics of Programming Drivetrains	What to consider when programming drivetrain motors and how to apply this to an arcade style teleoperated control.

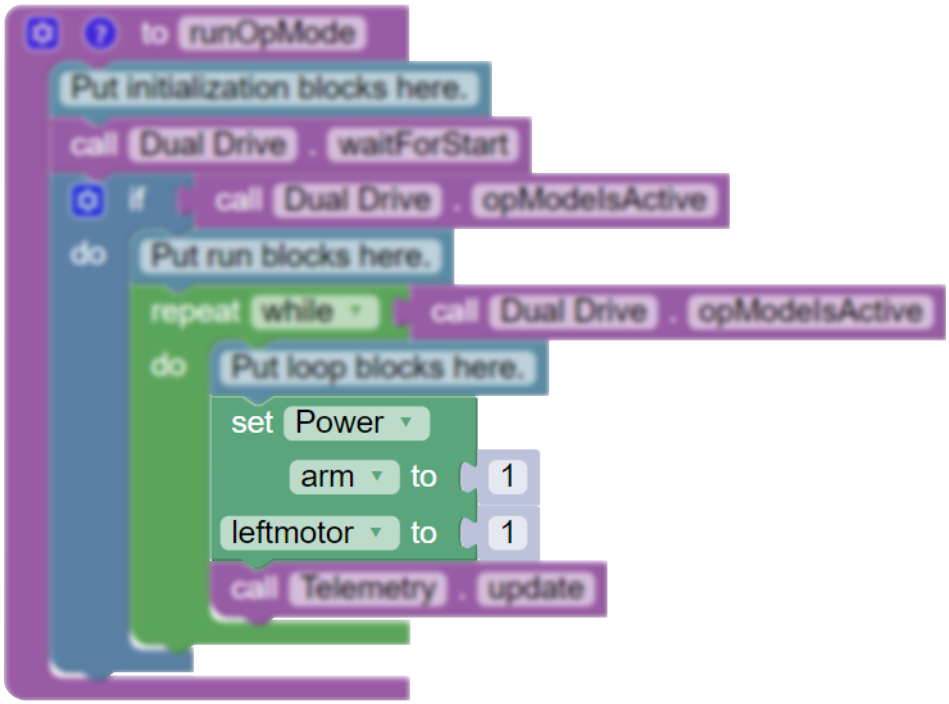
Basics of Programming Drivetrains

For controlling the Class Bot V2 drivetrain, being able to control two motors simultaneously is important. This is done through the **dual** motor block within Blocks. To access the **dual** motor block, at the top of the Categorize Blocks section there is a drop down menu for **Actuators**. Selecting DcMotor will drop down the options **Dual** and another drop down menu **Extended**. Select Dual to access the dual motor blocks.



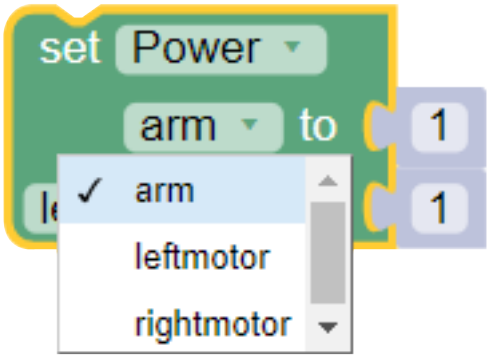
Programming Drivetrain Motors

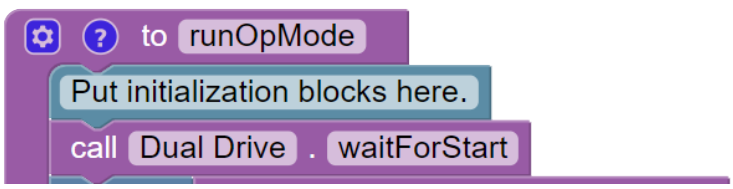
Add the  block to op mode while loop.

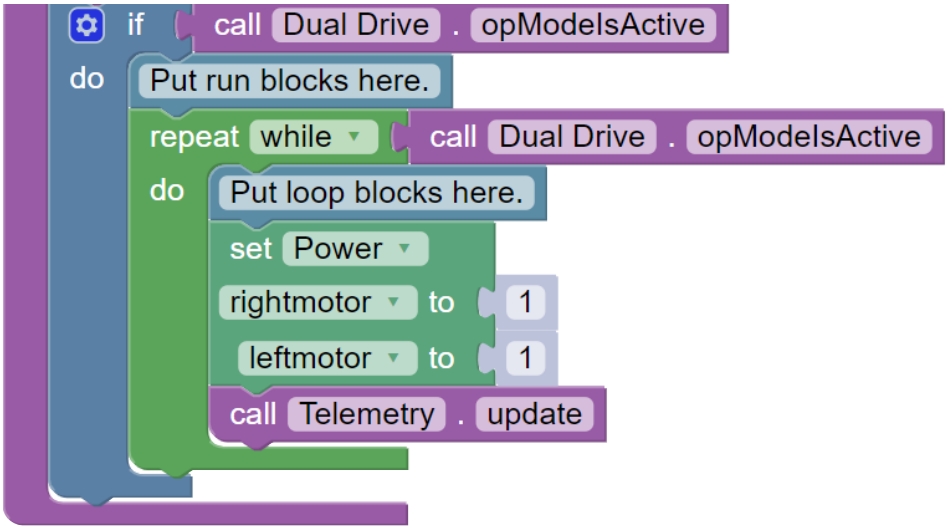


> ⓘ When there are multiple of the same type of variable (such as multiple Dc Motor variables) the variable specific blocks will choose a default variable based on alphabetical order. For this example Op Mode Dc Motor blocks will default to the arm variable.

Use the variable drop down menu on the block to change from arm to rightmotor.







Before moving on try running the code as is and consider the following questions:

- What behavior is the robot exhibiting?
- What direction is the robot spinning in?

When motors run at different speeds they spin along their center pivot point. But the motors are both set to a power (or duty cycle) of 1?

DC Motors are capable of spinning in two different directions depending on the current flow: clockwise and counter clockwise. When using a positive power value the Control Hub sends current to the motor for it to spin in a clockwise direction.

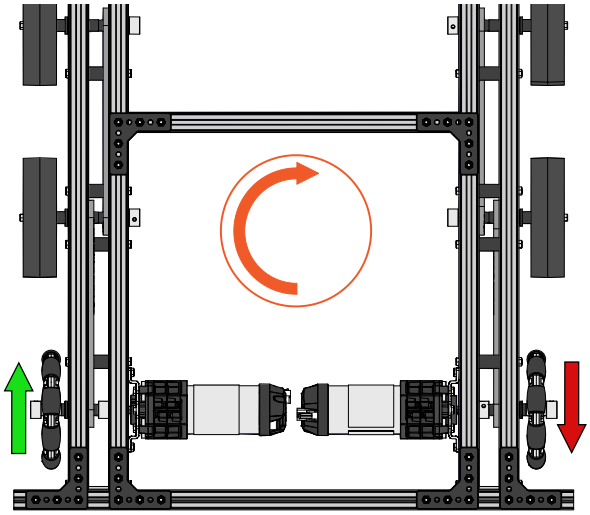
With the Class Bot and current code, both motors are currently set to run in the clockwise direction. If you set the robot on blocks and run the code again though, you can see that the motors run in opposing directions. With the mirrored way the motors mount to the drivetrain, one motor is naturally the inverse of the other.

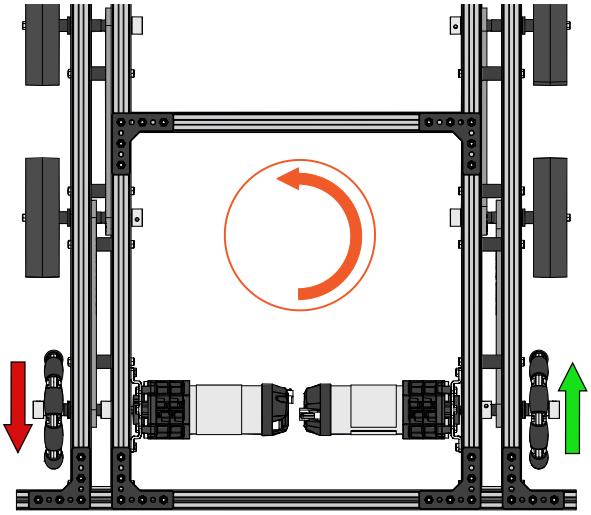
Why would the inverse motor cause the robot to spin in a circle? Both speed and direction of rotation of the **wheels** impact the overall direction the robot moves in. In this case, both **motors** were assigned to have the same power and direction but how the motors transfer motion to the **wheels**, causes the robot to spin instead of moving forward.

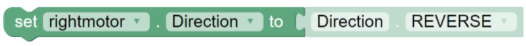
Check the Introduction to Motion section for more information on the mechanics of transferring motion and power.

In the info block above you were asked to determine which direction the robot spun in. The robot pivots in the direction of the inversed motor. For instance, when the right motor is the inversed motor the robot will pivot to the right. If the left motor is the inversed motor the robot will pivot to the left.

The Affect of Drivetrain Motors on Drivetrain Movement

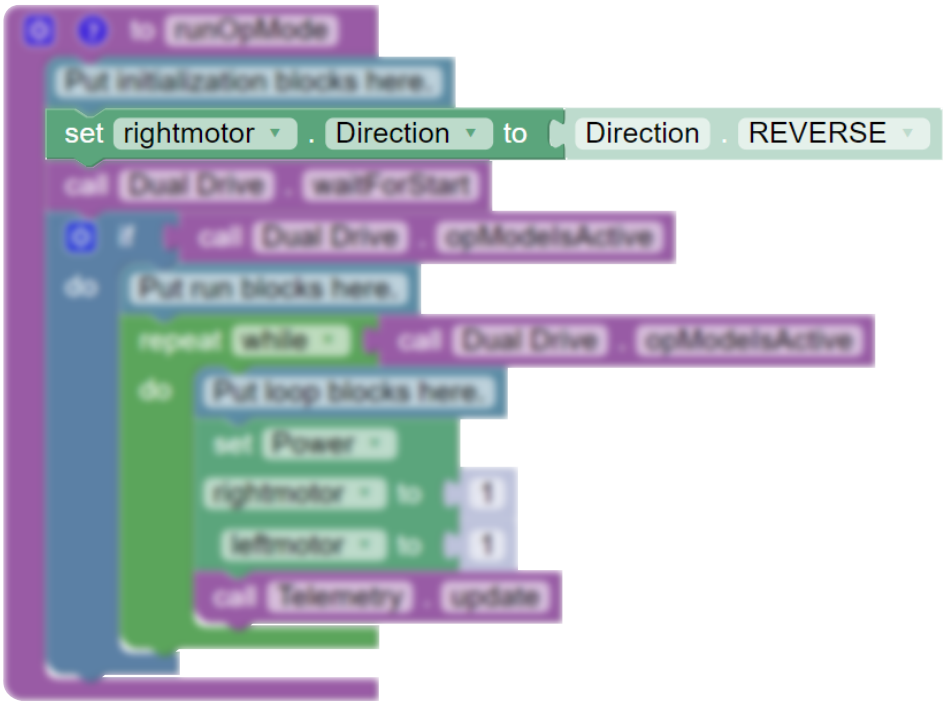


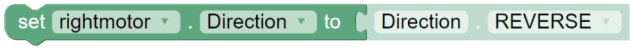


For the Class Bot, the robot pivots to the right, so the right motor will be reversed. Add 

to the op mode class, under the 

comment block.



Adding the 

block changes the direction of the right motor so that both motors will run in the same direction when power is set to one.

Teleoperated Driving - Arcade Style

Recall that when the motors were running in opposing directions the robot spun in circles. This same logic will be used to control the robot using the arcade style of control mentioned in the Hello Robot - Autonomous Robot section.

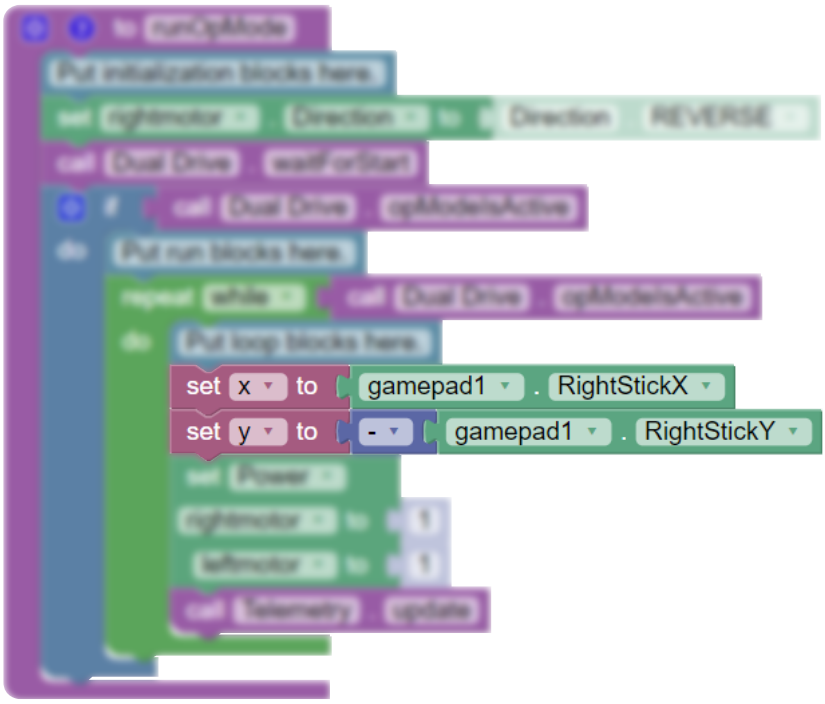
Programming with gamepads

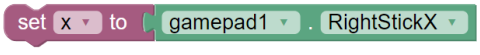
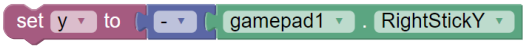
To start, create two variables x and y . Add the  and  blocks to the while loop.

y will be assigned as  , which is the y-axis of the right joystick.

> ⓘ Remember positive/negative values inputted by the gamepad's y-axis are inverse of the positive/negative values of the motor.

Assign x as the  , which is the x axis of the right gamepad joystick. The x-axis of the joystick does not need to be inverted.



The  and  block sets

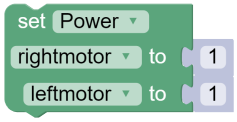
assign values from the gamepad joystick to x and y . As previously mentioned, the joystick gives values along a two dimension coordinate system. y receives the value from the y- axis and x receives the value from the x-axis. Both axis output values between -1 and 1.

To better understand consider the following table. The table shows the expected value generated from moving the joystick all the way in one direction, along the axis. For instance, when the joystick is pushed all the way in the upwards direction the coordinate values are (0,1).

> The table below assumes that the the y values from the gamepad have been inverted in code when assigned to the variable y .

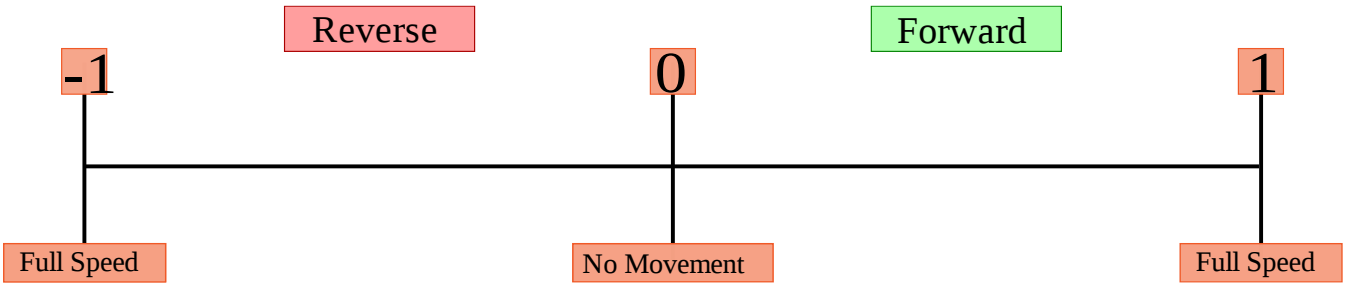
Joystick Direction	x	y
	0	1
	0	-1
	-1	0

	1	0

Now that you have a better understanding of how the physical movement of the gamepad affects the numerical inputs given to your Control System; its time to consider how to control the drivetrain using the joystick. Recall, from the Programming Drivetrain Motors section, that the speed and direction of a motor plays a large part in how the drivetrain moves. The numerical outputs for 

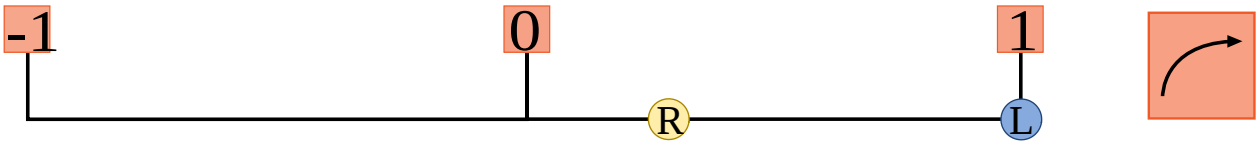
determine the speed and direction of the motors. For instance, when both motors are set to 1 they move in the forward direction at full speed (or 100% of duty cycle).

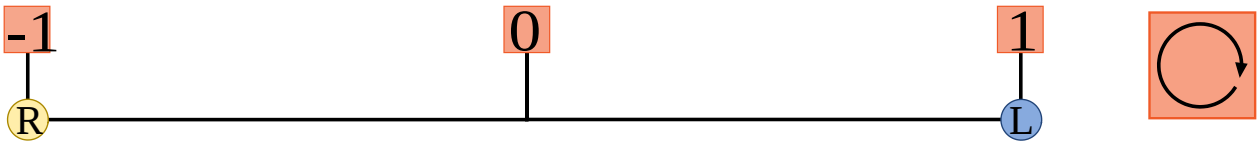
Much like the gamepads, the numerical value for `setPower` is in a range of -1 to 1. The absolute value of the assigned number determines percentage of duty cycle. As an example, 0.3 and -0.3 both indicate that the motor is operating at a duty cycle of 30%. The sign of the number indicates the direction the motor is rotating in. To better understand, consider the following graphic.

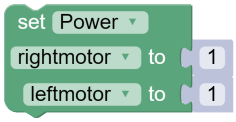


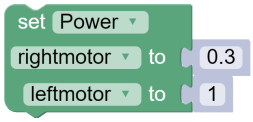
When a motor is assigned a `setPower` value between -1 and 0, the motor will rotate in the direction it considers to be reverse. When a motor is assigned a value between 0 and 1, it will rotate forward.

In the Programming Drivetrain Motors section, it was discussed that a robot rotates when the motors are moving in opposing directions. However, this has more to do with both speed and direction. To think of it numerically, a differential drivetrain will turn to the right when the `setPower` value for the right motor is less than that of the left motor. This is exhibited in the following example.





When both motors are 

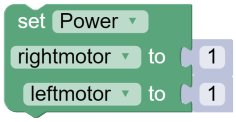
the robot will run at full speed in a mostly straight line. However, when the rightmotor is running in the same direction but at a lower speed, such as 

the robot will turn or rotate to the right. This is likely to be an arching movement that is not as sharp as a full pivot. In contrast when the rightmotor is set to full speed but in the opposite direction of the leftmotor, the robot pivots to the right. So, mathematically the following is considered to be true:

rightMotor = leftMotor	Forward or Reverse
rightMotor > leftMotor	Left Turn
rightMotor < leftMotor	Right Turn

As previously implied, 

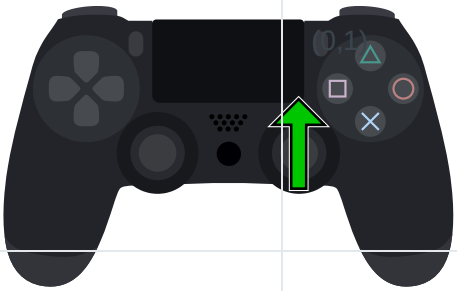
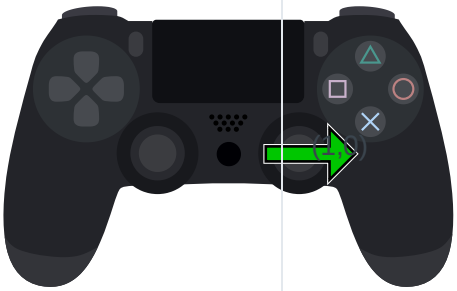
and 

send values to the Control System from the game pad joystick. In contrast, 

interprets numerical information set in the code and sends the appropriate current to the motors to dictate how the motors behave.

In an arcade drive, the following joy stick inputs (directions) need to correspond with the following outputs (motor power values).

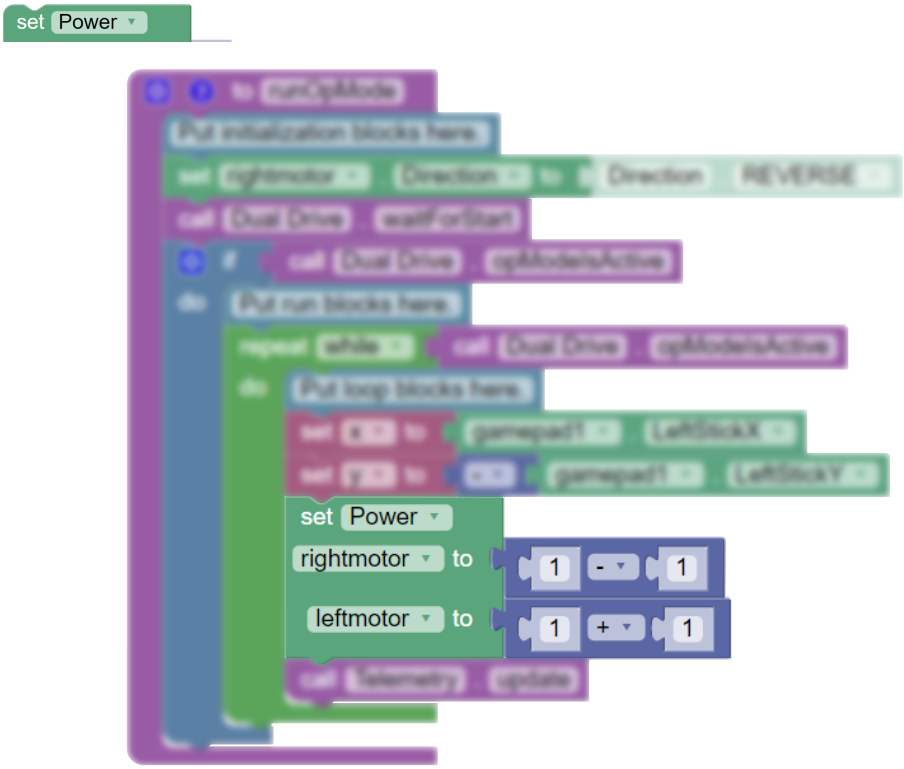
Joystick Direction	(x , y)	rightmotor	leftmotor

	1	1
	-1	-1
	1	-1
	-1	1

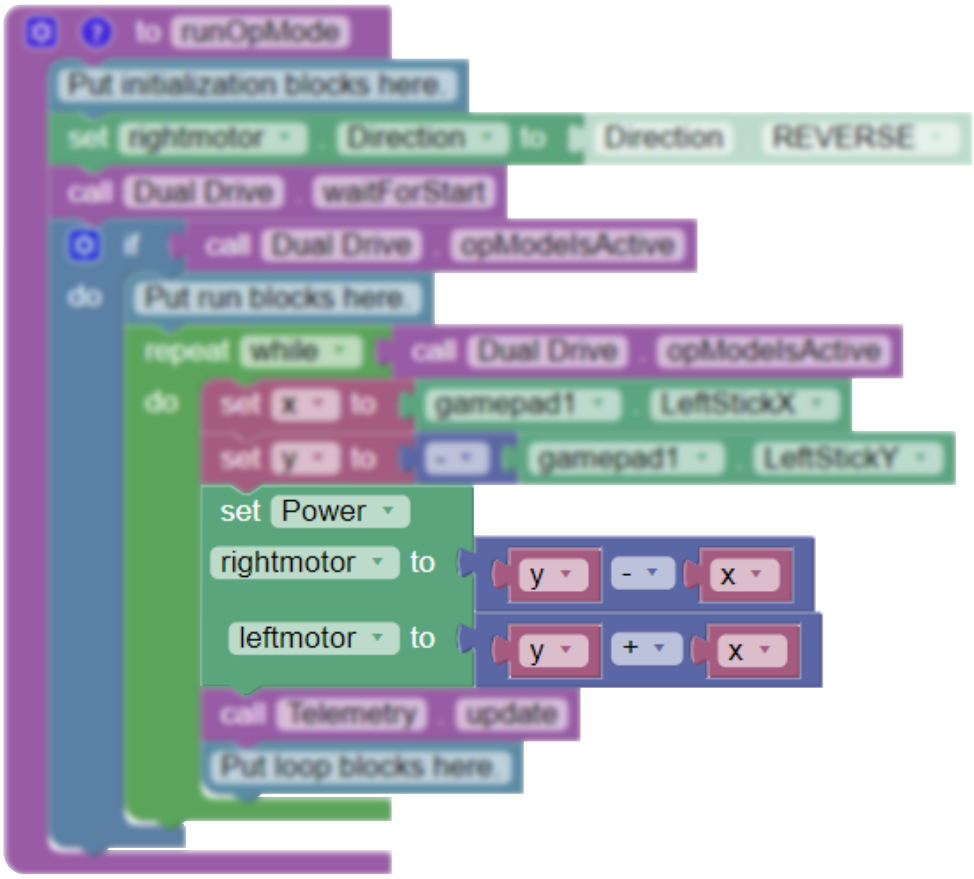
To get the outputs expressed in the table above, the gamepad values must be assigned to each motor in a meaningful way, where. Algebraic principles can be used to determine the two formulas needed to get the values. However, the formulas are provided below.

$$rightmotor = y - x$$
$$leftmotor = y + x$$

From the **math** menu grab the  and  blocks and add them to the respective



Next add the x and y to the formula blocks.



With this you now have a functional teleoperated arcade drive!

Elapsed Time - Blocks

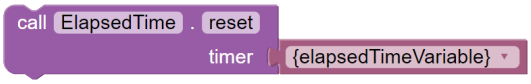
Introduction to Elapsed Time

One way to create an autonomous code is to use a timer to define which actions should occur when. Within the SDK actions can be set to a timer by using **ElapsedTime.**

Timers consist of two main categories: count up and count down. In most applications a timer is considered to be a device that counts down from a specified time interval. For instance, the timer on a phone or a microwave. However, some timers, like stopwatches, count upwards from zero. These types of timers measure the amount of time that has elapsed.

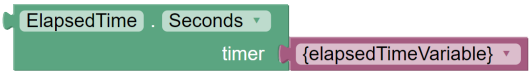
ElapsedTime is a count up timer. Registering the amount of time elapsed from the start of a set event, like the starting of a stopwatch. In this case, it is the amount of time elapsed from when the timer is created or reset within the code.

The ElapsedTime timer starts counting the amount of time elapsed from the point of its creation within a code. For instance, in this section ElapsedTime will be created in the section of code that occurs when the op mode is initialized. There is no option to stop the ElapsedTime timer. Instead, the



block can be used within your code to reset the timer at various intervals.

Once the timer has been reset, the amount of time that has elapsed is queried by calling blocks like

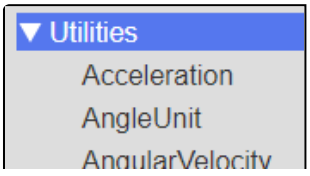


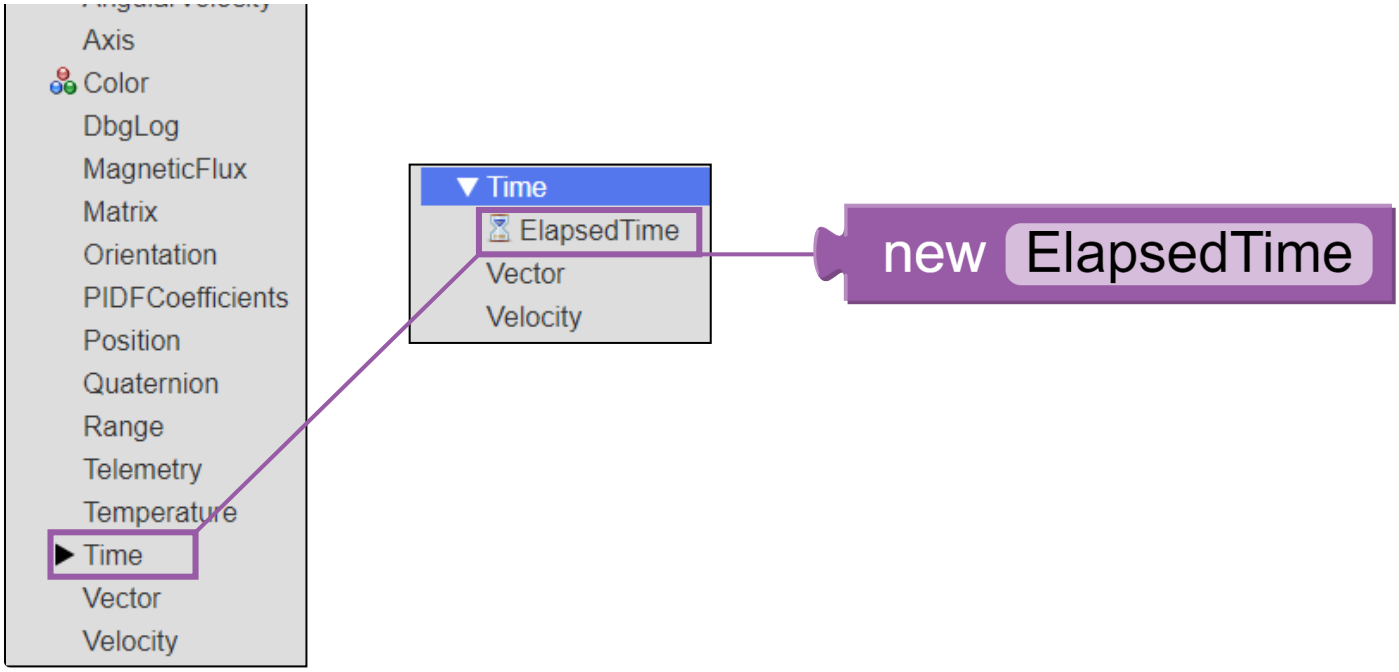
. The time given by the queried blocks can be used in loops to dictate how long a specific action should take place.

Sections	Goals of Section
Basics of programming with Elapsed Time	Learning the logic needed to use elapsed time for autonomous control.

Basics of Programming with Elapsed Time

Since this section focuses on creating an autonomous program using ElapsedTime it is important to understand where the elapsed time related blocks are located. At the top of the Categorize Blocks section there is a drop down menu for **Utilities**. The utilities drop down is a list of various utilities in alphabetical order. Towards the bottom of the the list select **Time** drop down menu. From there you can select **Elapsed Time**.

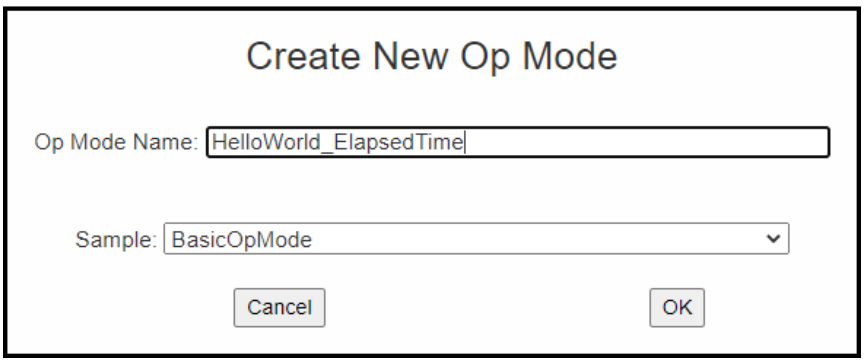




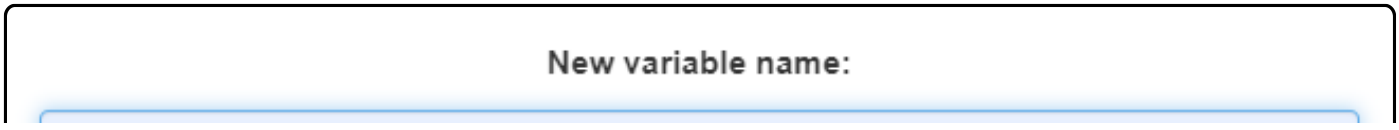
Programming with Elapsed Time

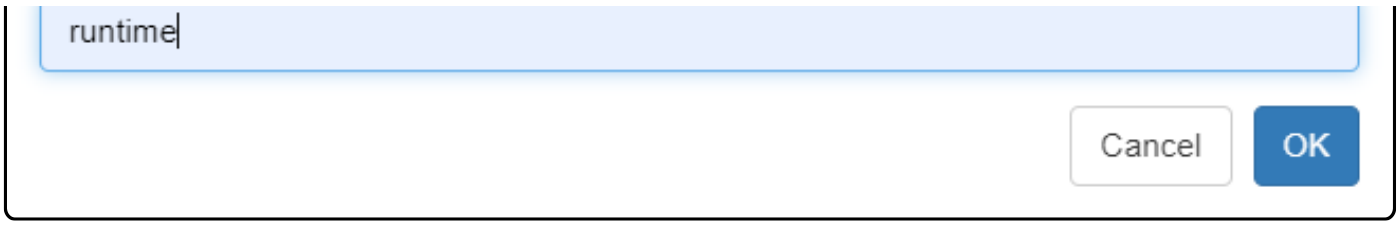
Start by creating a new op mode call `HelloWorld_ElapsedTime` using the `BasicOpMode` sample.

> When creating an op mode a decision needs to be made on whether or not to set it to autonomous mode. For applications under 30 seconds, typically required for competitive game play changing the op mode type to autonomous is recommended. For applications over 30 seconds, setting the code to the autonomous op mode type will limit your autonomous code to 30 seconds of run time. If you plan on exceeding the 30 seconds built into the SDK, keeping the code as a teleoperated op mode type is recommended.
>
> For information on how op modes work please visit the Introduction to Programming section.
>
> For more information on how to change the op mode type check out the Test Bed - Blocks section.



Create a variable named `runtime`.

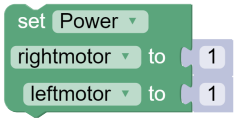




For information on creating variables in blocks please revisit the Test Bed - Blocks section.

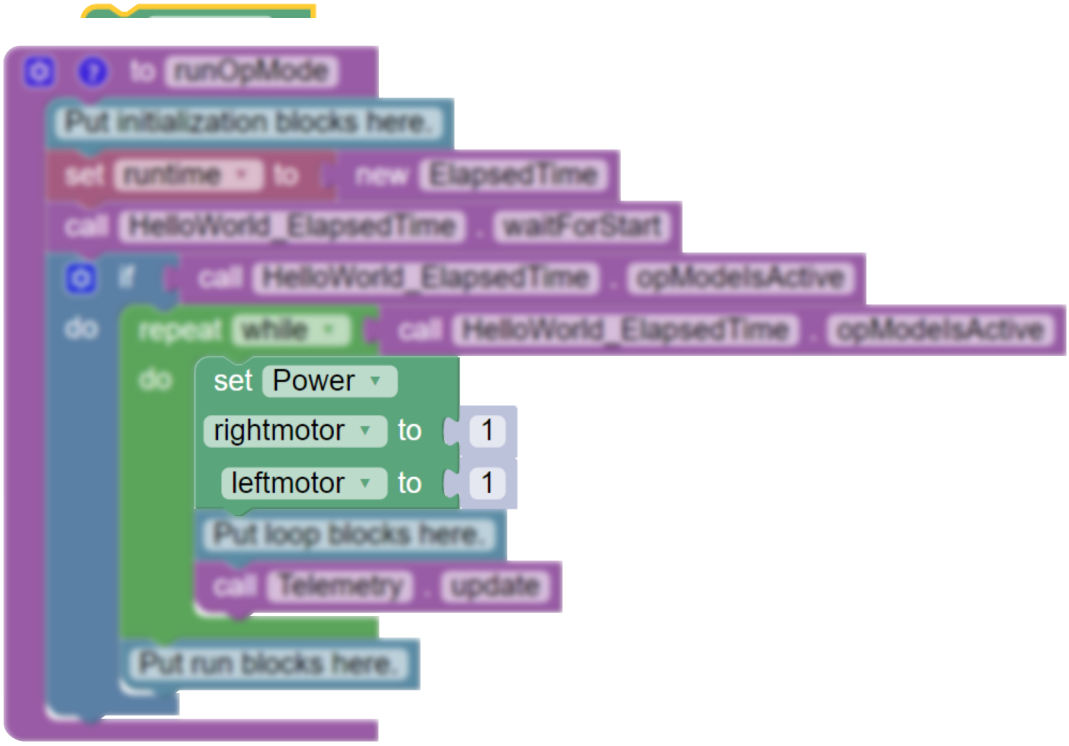
Add the  block to the op mode below the  comment block.

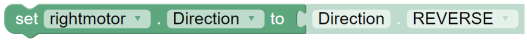
In order to utilize elements of the `ElapsedTime`, `runtime` will act as the `ElapsedTime` variable. Add the  block to the  block.

Before moving on to the rest of the `ElapsedTime` structure lets go ahead and add the motor related blocks. Add  to the op mode to the while loop.

When there are multiple of the same type of variable (such as multiple Dc Motor variables) the variable specific blocks will choose a default variable based on alphabetical order. For this example Op Mode Dc Motor blocks will default to the arm variable. Click the arrow next to the motor name to change the arm motor variable to the rightmotor variable. Use the variable drop

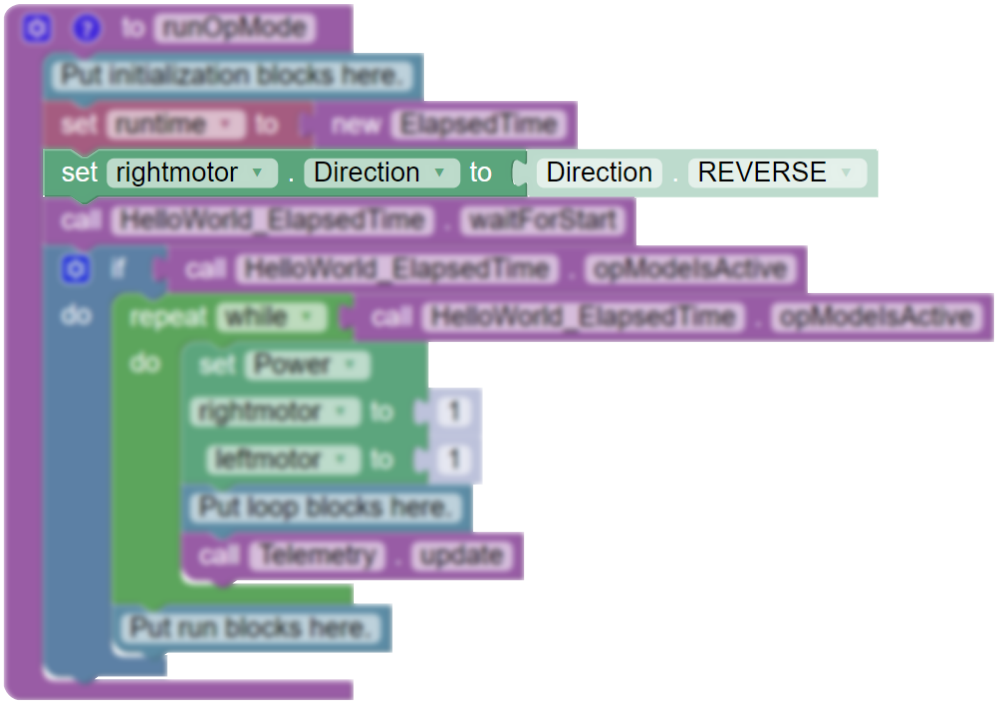
down menu on the block to change from arm to rightmotor.



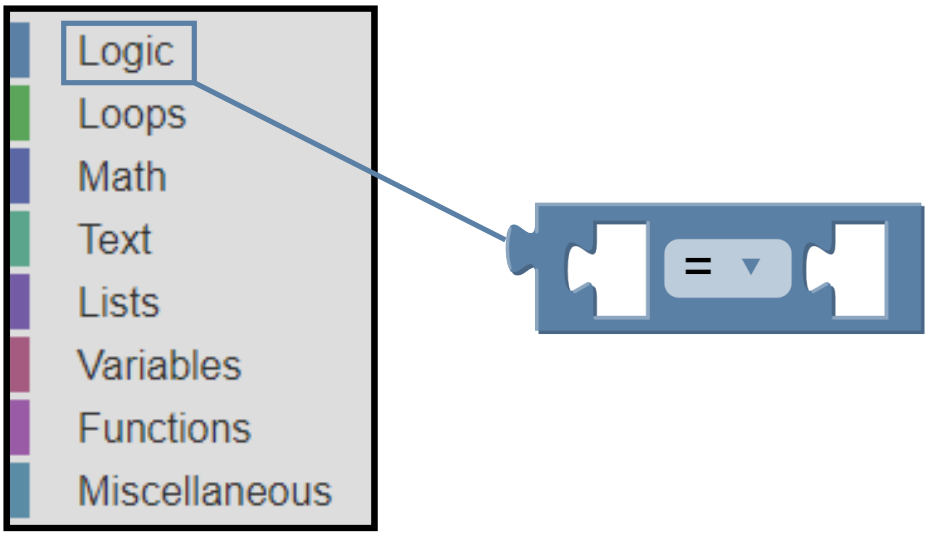
If you recall from Programming Drivetrain Motors article; the motors on the drivetrain mirror each other. The mirrored nature of the motor mounting causes the motors to rotate in opposing directions. In order to remedy this discrepancy the direction of the right motor needs to be reversed. Add the 

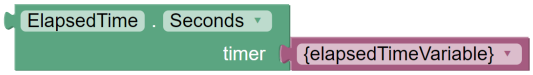
block to the op mode under the the 

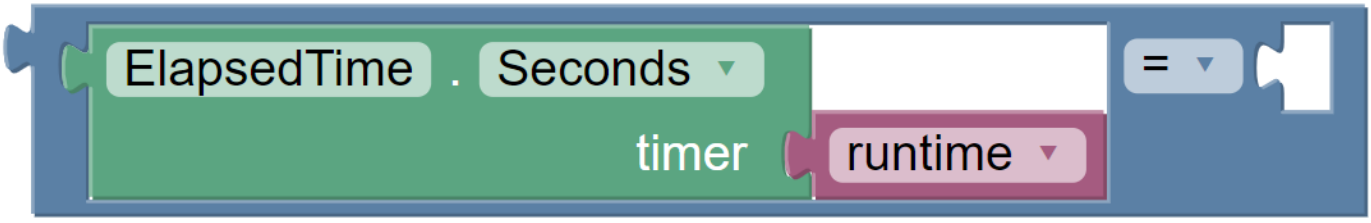
block set.



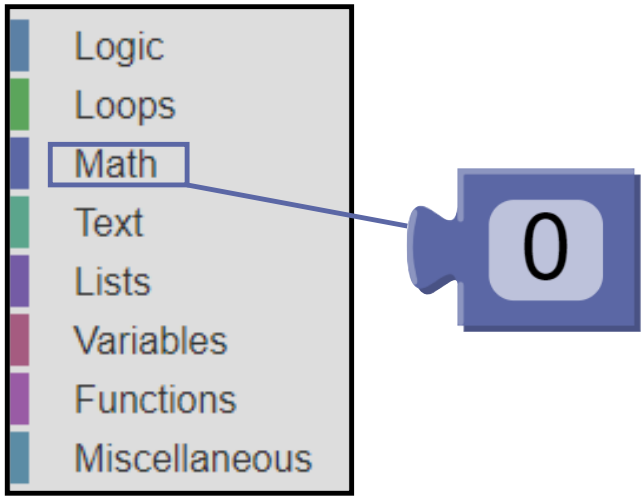
The goal is to have the motor move forward for 3 seconds. To accomplish this the While loops needs to be edited so that it triggers when the op mode is active and the ElapsedTime timer is less than or equal to 3 seconds. Lets start by creating the less than or equal to condition. Grab the  from the **Logic** menu.



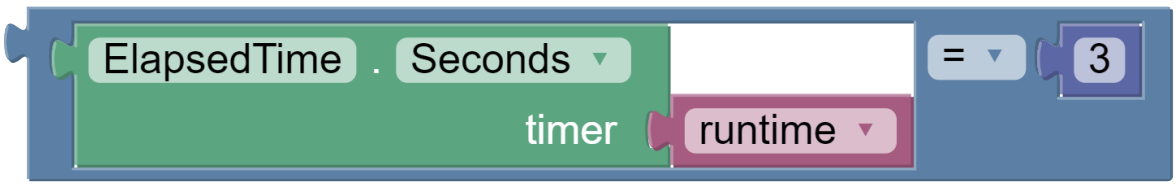
Select the  block from the **Elapsed Time** menu. Drop the block into the left side of the  block. Use the drop down menu to change the generic  to the  variable.

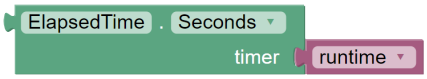


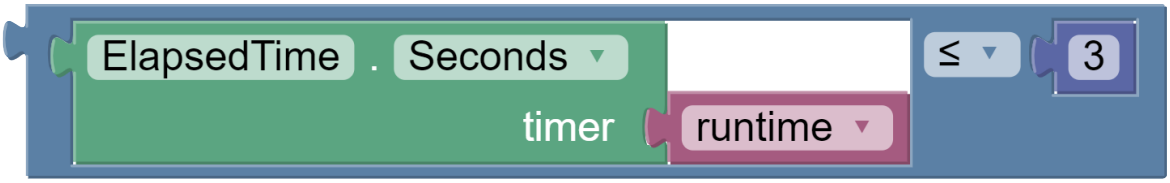
Grab the  block from the **Math** menu.



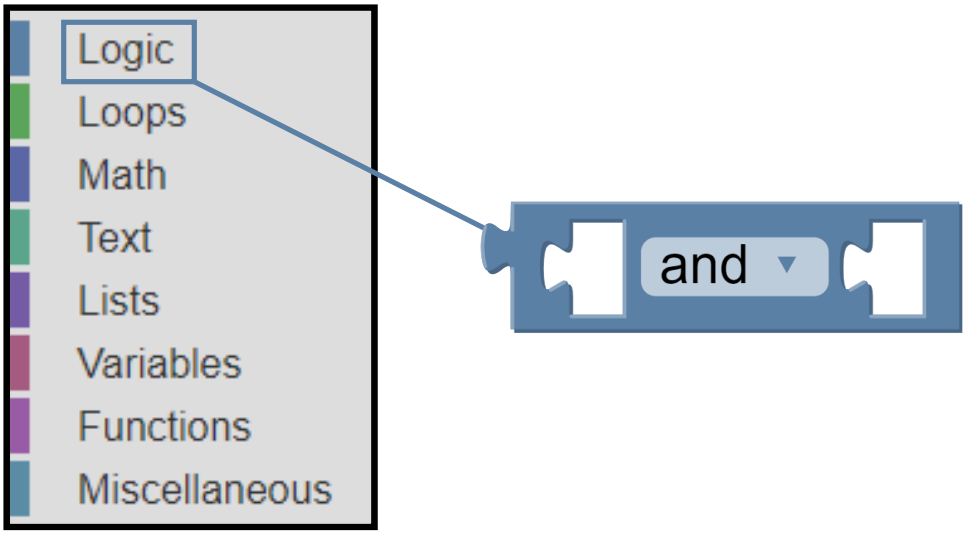
Add the number block to the right side of the  block. Change the number block to 3.

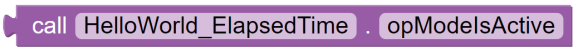
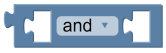


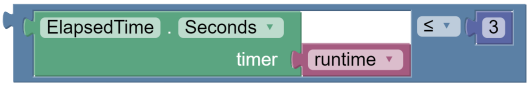
Right now the  is equal to three. Use the arrow next to the equal sign to choose the less than or equal to sign from the drop down menu.

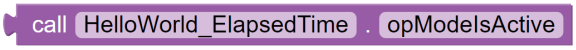


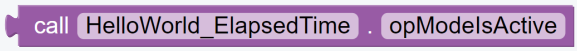
Set this block set to the side for now. Grab an  block from the **Logic** menu

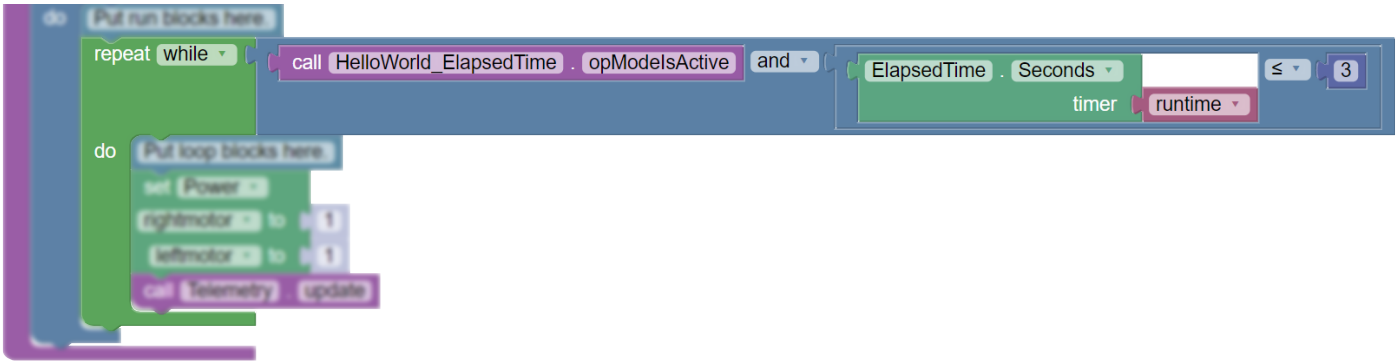


Add the call  block to the left side of the  block.

Add the  block set to the right side of the  block.

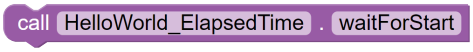
This block set will replace the  block that is currently attached to the while loop. With this block set in place the while loop will now activate when both conditions of the *and* block are true.

> It is important to know that, within a linear op mode, a while loop must always have the  Boolean as a condition. This condition ensures that the while loop will terminate when the stop button is pressed.

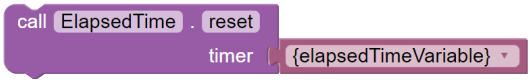


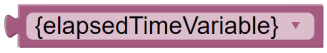
> Use tape to mark the distance from where the robot starts to where you would like it to end up. Try running the code using the following conditions:
>
> - Press the init button and immediately press play
> - Press the init button, wait 30 seconds and then press play
>
> What difference in behavior did you notice?

Recall that the `ElapsedTime` timer starts when the timer is created, which occurs where the 

block is placed. Since the timer is created prior to 

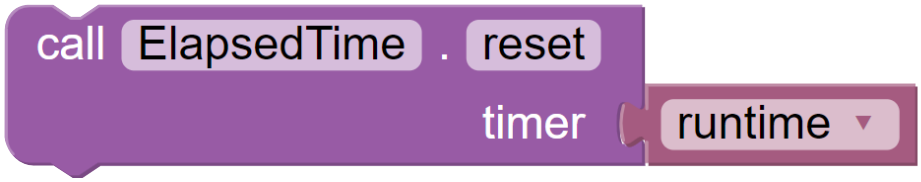
, the timer will start when the program is initialized.

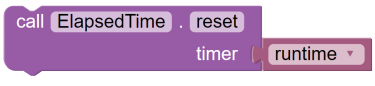
If you tested the program you may have noticed that the robot didn't move the during the second run. Depending on how long you wait to start after initialization the timer may be close to or past 3 seconds by the time the program is played. To keep this from happening the timer should be reset once the op mode is active. Grab the call 

block. Use the drop down menu on the variable block to change the 

to 

.

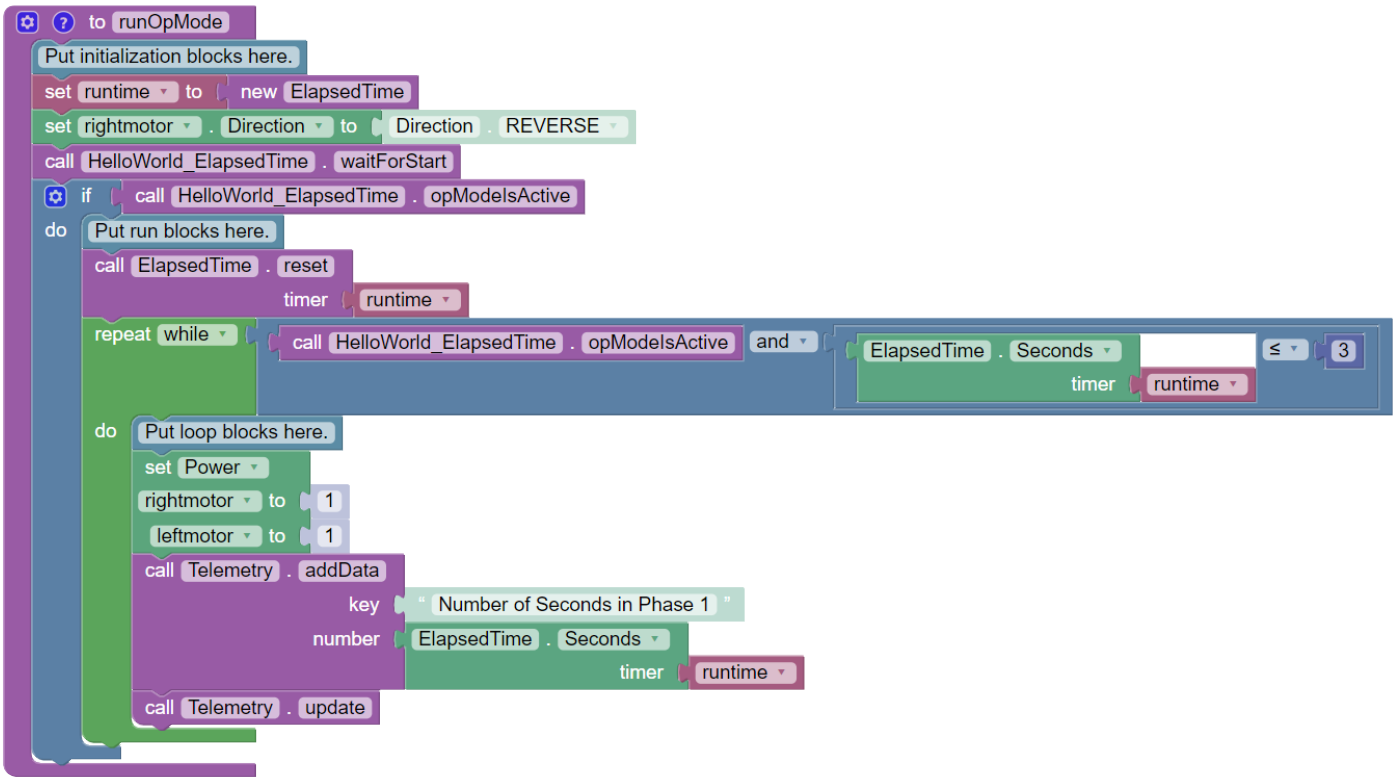


Add the 

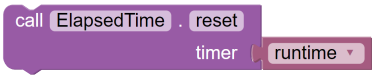
to the op mode beneath the 

comment and above the while loop.

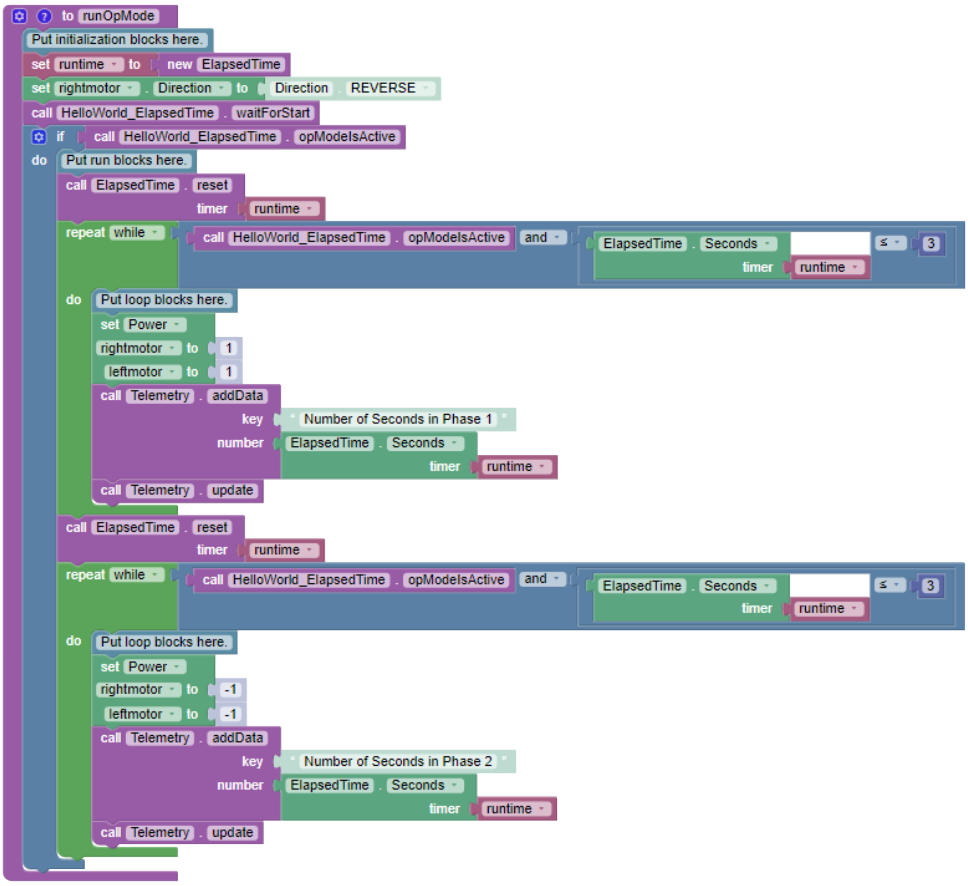
As mentioned in previous sections, it can be beneficial to have a telemetry output when testing code. In the following example telemetry is used to output the amount of time that has passed with the timer.



The above code will allow your motor to drive straight for 3 seconds. Additional movements can be added by duplicating the while loop. Right click the while loop block and select duplicate

Once you have duplicated the while loop you can change some of the basic information like motor power or the time interval of the loop. You will also need to add a  block between the two loops.

Full Code Example



Encoder Navigation - Blocks

In the previous section you learned about how to use Elapsed Time to allow your robot to navigate the world around it autonomously. When starting out many of the robot actions can be accomplished by turning on a motor for a specific amount of time. Eventually, these time-based actions may not be accurate or repeatable enough. Environmental factors, such as the state of battery charge during operation and mechanisms wearing in through use, can all affect time-based actions. Fortunately, there is a way to give feedback to the robot about how it is operating by using sensors; devices that are used to collect information about the robot and the environment around it.

With Elapsed Time, in order to get the robot to move to a specific distance, you had to estimate the amount of time and the percentage of duty cycle needed to get from point a to point b. However, the REV motors come with built in encoders, which provide feedback in the form of ticks (or counts) per revolution of the motor. The information provided by the encoders can be used to move the motor to a target position, or a target distance.

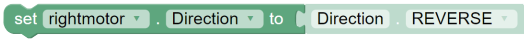
Moving the motors to a specific position, using the encoders, removes any potential inaccuracies or inconsistencies from using Elapsed Time. The focus of this section is to move the robot to a target position using encoders.

> There are two articles in that go through the basics of Encoders. Using Encoders goes through the basics of the different types of motor modes, as well as a few application examples of using these modes in code. In this section we will focus on using `RUN_TO_POSITION`.
>
> The other article, Encoders, focuses on the general functionality of an encoder.
>
> It is recommended that you review both articles before moving on with this guide.

Basics of Programming with Encoders

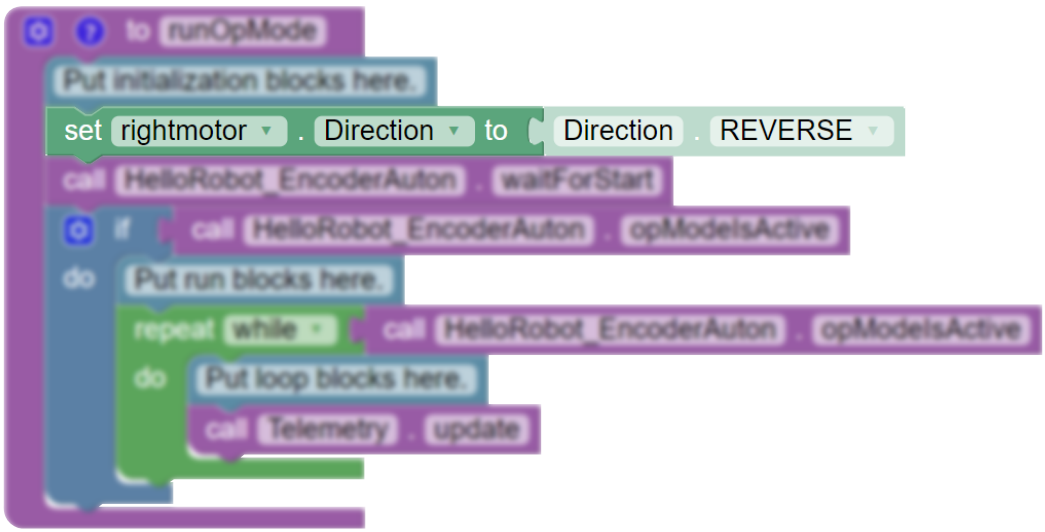
Start by creating a basic op mode called HelloRobot_EncoderAuton.

> When creating an op mode a decision needs to be made on whether or not to set it to autonomous mode. For applications under 30 seconds, typically required for competitive game play changing the op mode type to autonomous is recommended. For applications over 30 seconds, setting the code to the autonomous op mode type will limit your autonomous code to 30 seconds of run time. If you plan on exceeding the 30 seconds built into the SDK, keeping the code as a teleoperated op mode type is recommended.
>
> For information on how op modes work please visit the Introduction to Programming section.
>
> For more information on how to change the op mode type check out the Test Bed - Blocks section.

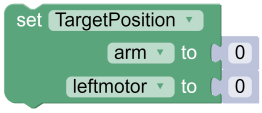
Add the 

block to the op mode under the 

.This will change the direction of the rotation of the right motor to be the same direction as the left motor.

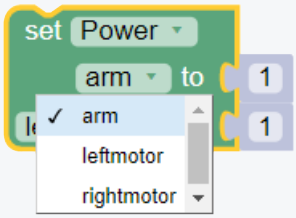


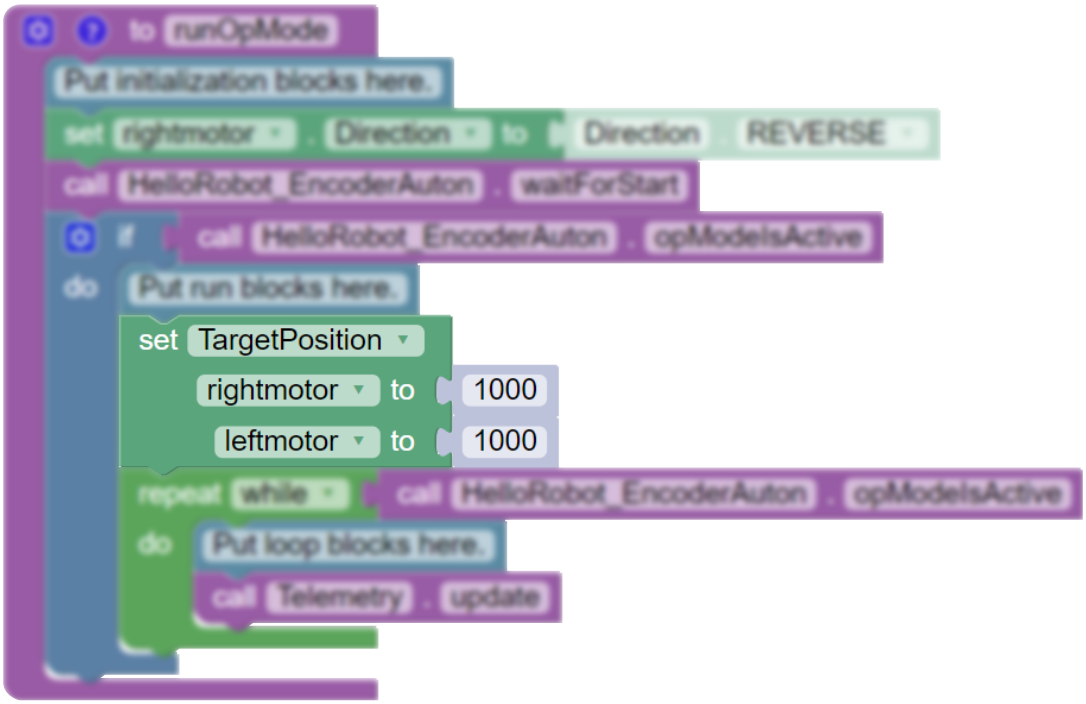
> For more information on the directionality of motor check out the Basics of Programming Drivetrains section.

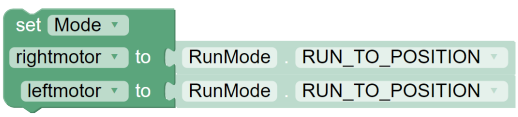
Recall from Using Encoders that using `RUN_TO_POSITION` mode requires a three step process. The first step is setting target position. To set target position, grab the 

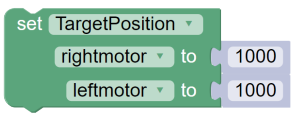
block and add it to the op mode under the 

comment. To get a target position that equates to a target distance requires so calculations, which will be covered later. For now, set target position to 1000 ticks.

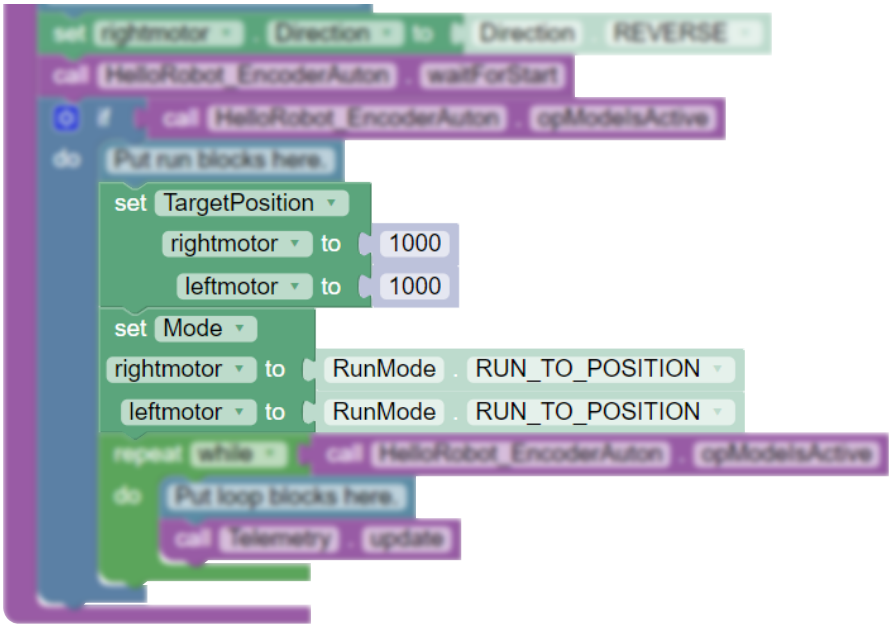
> When there are multiple of the same type of variable (such as multiple Dc Motor variables) the variable specific blocks will choose a default variable based on alphabetical order. For this example Op Mode Dc Motor blocks will default to the arm variable. Click the arrow next to the motor name to change the arm motor variable to the rightmotor variable. Use the variable drop down menu on the block to change from arm to rightmotor.
>
> 
>

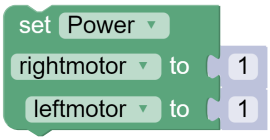


The next step is to set both motors to the `RUN_TO_POSITION` mode. Place the 

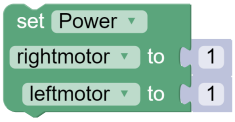
block beneath the 

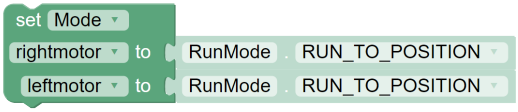
block.



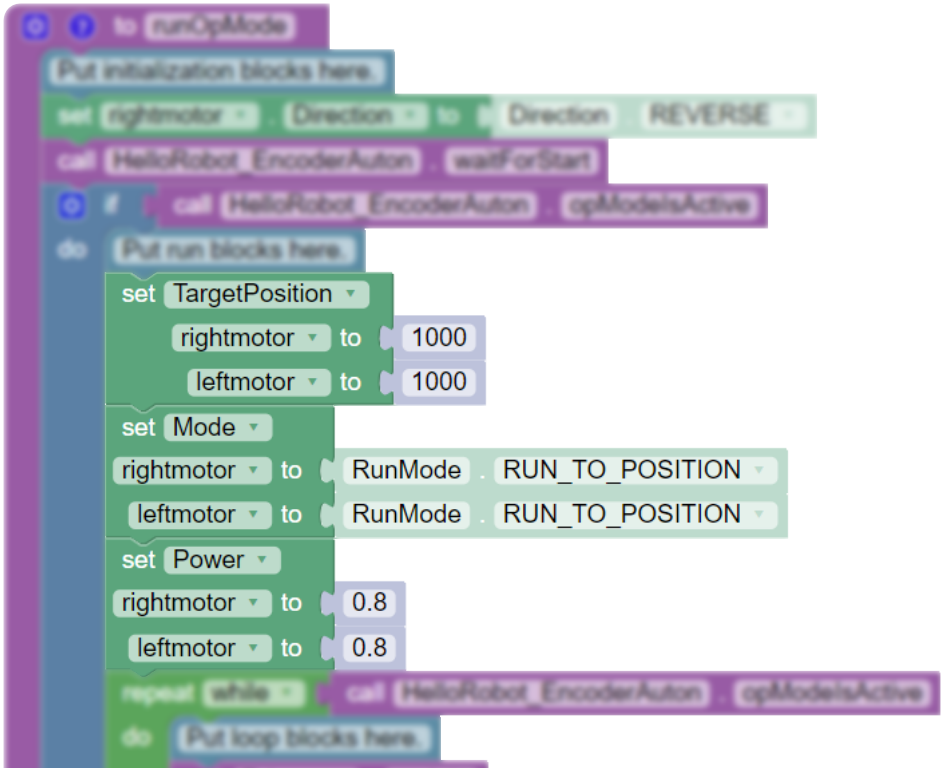
The main focus of the three step process is to set a target, tell the robot to move to that target, and at what speed (or velocity) the robot should get to that target. Normally, the recommended next step is to calculate velocity and set a target velocity based on ticks. However, this requires quite a bit of math to find the appropriate velocity. For testing purposes, its more important to make sure that the main part of the code is working before getting too deep into the creation of the code. Since the 

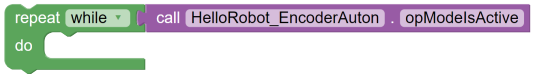
function was covered in previous sections and will communicate to the system what relative speed (or in this case duty cycle) is needed to get to the target, this can be used in the place of velocity for now.

Add the 

block to the op mode beneath the 

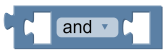
block. Change the duty cycle (or power) of both motors to 0.8, instead of 1.

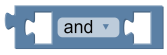


Now that all three `RUN_TO_POSITION` steps have been added to the code the code can be tested. However, if you want to wait for the motor to reach its target position before continuing in your program, you can use a while loop that checks if the motor is busy (not yet at its target). For this program lets edit the 

block.

> Recall that, within a linear op mode, a while loop must always have the 
>
> Boolean as a condition. This condition ensures that the while loop will terminate when the stop button is pressed.

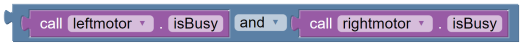
Grab an 

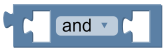
block from the logic menu and add it to the while loop. On the left side of the 

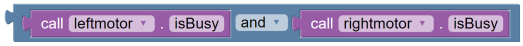
block add the 

block. On the right side add the 

block.

Embed the 

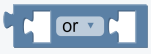
in another 

block. Place the 

on the right side of the 

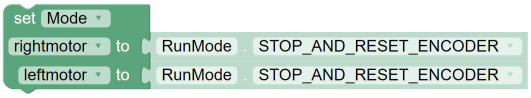
block. On the left side add the 

block.

> Right now the while loop is waiting for the right and left motors to reach their respective targets. There may be occasions when you want to wait for both motors to reach their target position, in this case the 
>
> can be used.
>
> 
>

> Save and run the op mode two times in a row. Does the robot move as expected the second time?

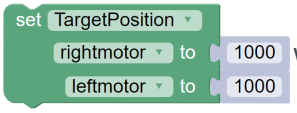
> Try turning the Control Hub off and then back on. How does the robot move?

In the Basic Encoder Concepts section, it is clarified that ell encoder ports start at 0 ticks when the Control Hub is turned on. Since you did not turn off the Control Hub in between runs, the second time you ran the op mode the motors were already at, or around, the target position. When you run a code, you want to ensure that certain variables start in a known state. For the encoder ticks, this can be achieved by setting the mode to 

. Add this block to the op mode in the initialization section. Each time the op mode is initialized, the encoder ticks will be reset to zero.

> ⓘ For more information on the motor mode `STOP_AND_RESET_ENCODERS` check out the STOP_AND_RESET_ENCODERS section of the Using Encoders guide.

Converting Encoder Ticks to a Distance

In the previous section, the basic structure needed to use `RUN_TO_POSITION` was created. The placement of 

within the code, set the target position to 1000 ticks. What is the distance from the starting point of the robot and the point the robot moves to after running this code?

Rather than attempt to measure, or estimate, the distance the robot moves, the encoder ticks can be converted from amount of ticks per revolution of the encoder to how many encoder ticks it takes to move the robot a unit of distance, like a millimeter or inch. Knowing the amount of ticks per a unit of measure allows you to set a specific distance. For instance, if you work through the conversion process and find out that a drivetrain takes 700 ticks to move an inch, this can be used to find the total number of ticks need to move the robot 24 inches.

> Reminder that the basis for this guide is the Class Bot V2. The REV DUO Build System is a metric system. Since part of the conversion process references the diameter of the wheels, this section will convert to ticks per mm.

When using encoders built into motors, converting from ticks per revolution to ticks per unit of measure moved requires the following information:

- Ticks per revolution of the encoder shaft
- Total gear reduction on the motor
  - Including gearboxes and motion transmission components like gears, sprockets and chain, or belts and pulleys
- Circumference of the driven wheels

Ticks per Revolution

The amount of ticks per revolution of the encoder shaft is dependent on the motor and encoder. Manufacturers of motors with built-in encoders will have information on the amount of ticks per revolution. For HD Hex Motors the encoder counts 28 ticks per revolution of the motor shaft.

> Visit the manufacturers website for your motor or encoders for more information on encoder counts. For HD Hex Motors or Core Hex Motors visit the Motor documentation.

Total Gear Reduction

Since ticks per revolution of the encoder shaft is before any gear reduction calculating the total gear reduction is needed. This includes the gearbox and any addition reduction from motion transmission components. To find the total gear reduction use the Compound Gearing formula.

For the Class Bot V2 there are two UltraPlanetary Cartridges, 4:1 and 5:1, and an additional gear reduction from the UltraPlanetary Output to the wheels, 72T:45T ratio.

> The UltraPlanetary Cartridges use the nominal gear ratio as a descriptor. The actual gear ratios can be found in the UltraPlanetary Users Manual's Cartridge Details.

Using the compound gearing formula for the Class Bot V2 the total gear reduction is:

$$\frac{3.61}{1} * \frac{5.23}{1} * \frac{72}{45} = 30.21$$

> Unlike the the spur gears used to transfer motion to the wheels, the UltraPlanetary Gearbox Cartridges are planetary gear systems. To make calculations easier the gear ratios for the Cartridges are already reduced.

Circumference of the Wheel

The Class Bot V2 uses the 90mm Traction Wheels. 90mm is the diameter of the wheel. To get the appropriate circumference use the following formula

$$circumference = diameter * \pi$$

You can calculate this by hand, but for the purpose of this guide, this can be calculated within the code.

> ⓘ Due to wear and manufacturing tolerances, the diameter of some wheels may be nominally different. For the most accurate results consider measuring your wheel to confirm that the diameter is accurate.

To summarize, for the Class Bot V2 the following information is true:

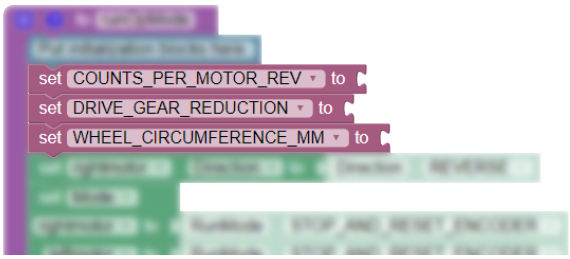
Ticks per revolution	28 ticks
Total gear reduction	30.21
Circumference of the wheel	$90mm * \pi$

Each of these pieces of information will be used to find the number of encoder ticks (or counts) per mm that the wheel moves. Rather than worry about calculating this information by hand, these values can be added to the code as constant variables. To do this create three variables:

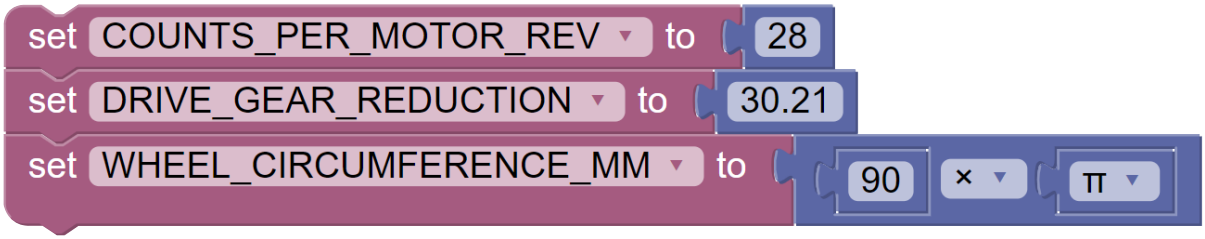
- `COUNTS_PER_MOTOR_REV`
- `DRIVE_GEAR_REDUCTION`
- `WHEEL_CIRCUMFERENCE_MM`

> ⓘ The common naming convention for constant variables is known as CONSTANT_CASE, where the variable name is in all caps and words are separated by and underscore.

Add the variables to the initialization section of the op mode.



Once the variables are created and added to the op mode, use the 0 blocks to set the variables to the respective values. For `WHEEL_CIRCUMFERENCE_MM` a combination of the 1 × 1 , 0 , and π blocks to get the circumference of the wheel. The



Now that these three variables have been defined, we can use them to calculate two other variables: the amount of encoder counts per rotation of the wheel and the number of counts per mm that the wheel moves.

To calculate counts per wheel revolution multiple `COUNTS_PER_MOTOR_REV` by `DRIVE_GEAR_REDUCTION` Use the following formula:

$$y = a * b$$

Where,

- a = `COUNTS_PER_MOTOR_REV`
- b = `DRIVE_GEAR_REDUCTION`
- y = `COUNTS_PER_WHEEL_REV`

Once `COUNTS_PER_WHEEL_REV` is calculated, use it to calculate the counts per mm that the wheel moves. To do this divide the `COUNTS_PER_WHEEL_REV` by the `WHEEL_CIRCUMFERENCE_MM` . Use the following formula.

$$x = \frac{(a * b)}{c} = \frac{y}{c}$$

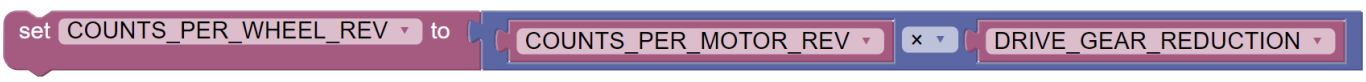
Where,

- a = `COUNTS_PER_MOTOR_REV`
- b = `DRIVE_GEAR_REDUCTION`
- c = `WHEEL_CIRCUMFERENCE_MM`
- y = `COUNTS_PER_WHEEL_REV`
- x = `COUNTS_PER_MM`

> `COUNTS_PER_WHEEL_REV` will be created as a separate variable from `COUNTS_PER_MM` as it is used in calculating a target velocity.

Create these variables in Blocks and add then to the op mode under the other constant variables.

Again math blocks need to be used to define these variables. Lets start with the `COUNTS_PER_WHEEL_REV` variable. Add a 1 × 1 to the set COUNTS_PER_WHEEL_REV to block. Add the COUNTS_PER_MOTOR_REV and DRIVE_GEAR_REDUCTION blocks to either side of the 1 × 1 block.



Since `COUNTS_PER_WHEEL_REV` has been calculated it can be used to calculate `COUNTS_PER_MM` add the 

to the 

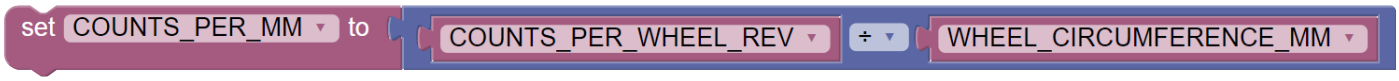
. On the left side of the 

add the 

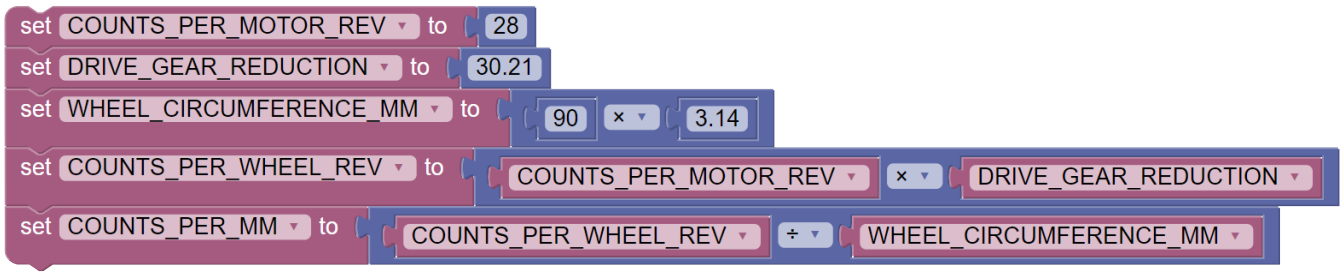
block. On the right side of the 

add the 

.



Once `COUNTS_PER_WHEEL_MM` is set, this completes the conversion process, and all constant variables are set.

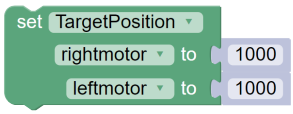


Moving to a Target Distance

Now that you have created the constant variables needed to calculate the amount of ticks per mm moved, you can use this to set a target distance. For instance, if you would like to have the robot move forward two feet, converting from feet to millimeters and multiplying by the `COUNTS_PER_MM` will give you the amount of counts (or ticks) needed to reach that distance.

Create two more variables called `leftTarget` and `rightTarget`. Add the 

and 

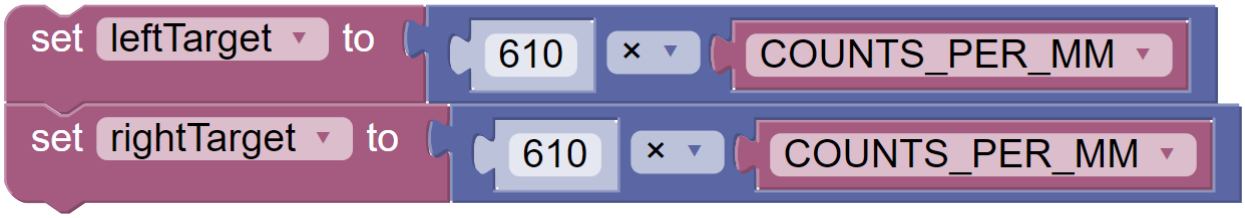
blocks to the op mode above the 

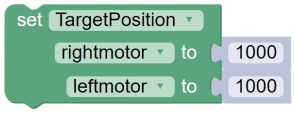
block.

Right now the main distance factor is `COUNTS_PER_MM` , however you may want to go a distance that is in the imperial system, such as 2 feet (or 24 inches). The target distance in this case will need to be converted to mm. To convert from feet to millimeters use the following formula:

$$d_{(mm)} = d_{(ft)} \times 304.8$$

If you convert 2 feet to millimeters, it comes out the be 609.6 millimeters. For the purpose of this guide, lets go ahead an round this to be 610 millimeters. Multiply 610 millimeters by the `COUNTS_PER_MM` variable to get the number of ticks needed to move the robot 2 feet. Since the intent is to have the robot move in a straight line, set both the `leftTarget` and `rightTarget` , to be equal to 610 * `COUNTS_PER_MM`

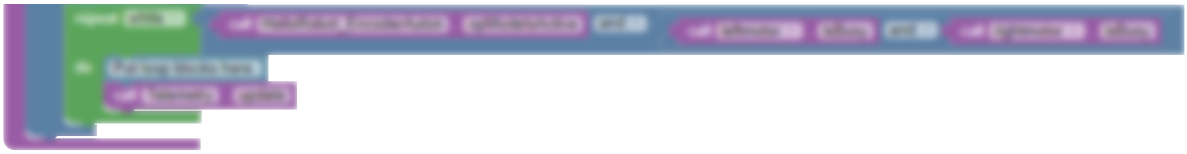


Edit the 

so that both motors are set to the appropriate target position. To do this add the 

and 

blocks to their respective motor.



Setting Velocity

Velocity is a closed loop control within the SDK that uses the encoder counts to determine the approximate power/speed the motors need to go in order to meet the set velocity. When working with encoder setting a velocity is recommended over setting a power level, as it offers a higher level of control.

To set a velocity, its important to understand the maximum velocity in RPM your motor is capable of. For the Class Bot V2 the motors are capable of a maximum RPM of 300. With a drivetrain, you are likely to get better control by setting velocity lower than the maximum. In this case, lets set the velocity to 175 RPM

> ⓘ Recall that `setVelocity` is measure in ticks per second.

Since RPM is the amount of revolutions per minute a conversion needs to be made from RPM to ticks per second. To do this divide the RPM by 60, to get the amount of rotations per second. Rotations per second can the be multiplied by `COUNTS_PER_WHEEL_REV`, to get the amount of ticks per second.

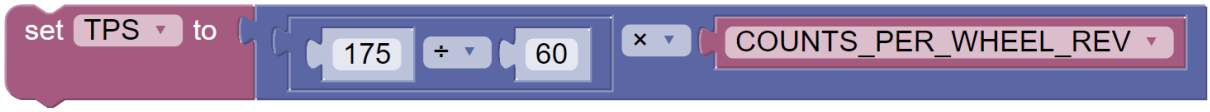
$$TPS = \frac{175}{60} * CPWR$$

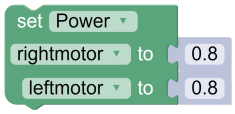
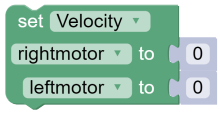
Create a new variable called TPS. Add the 

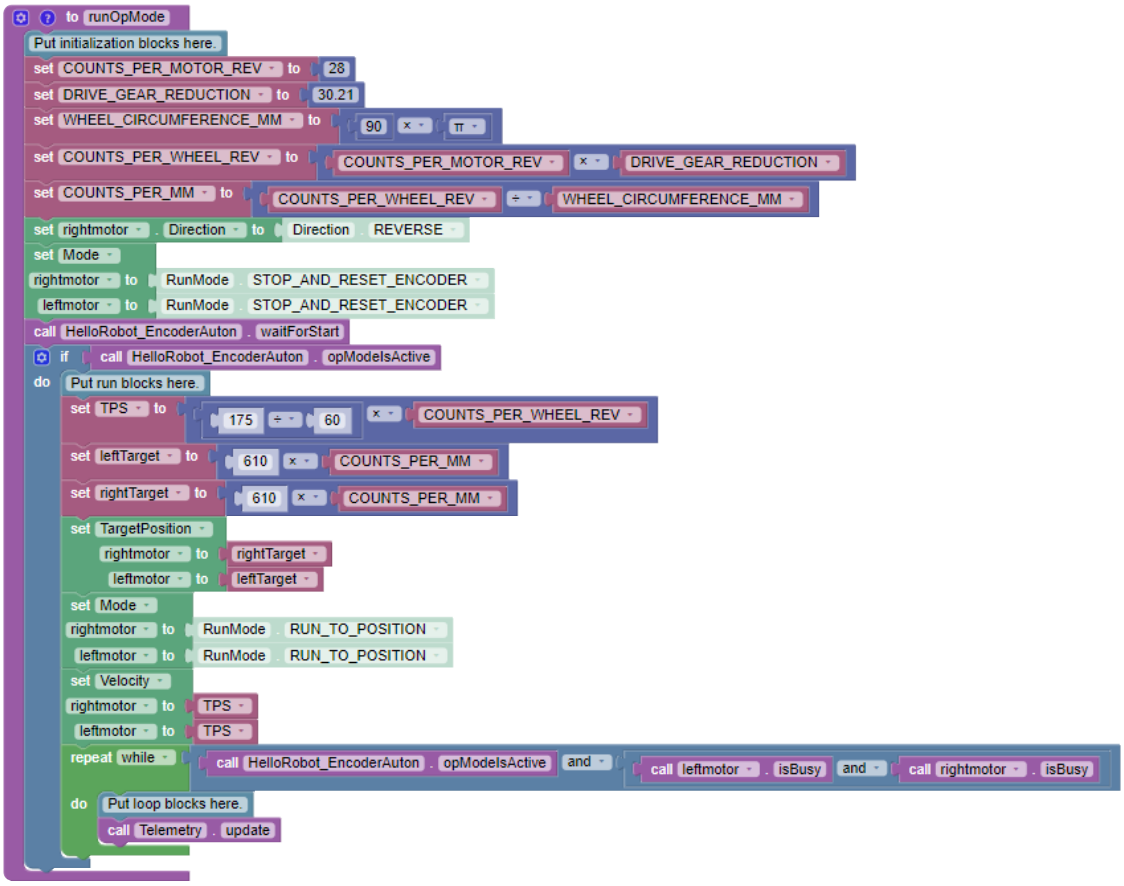
to the op mode under the 

comment block.

Add a  block to the  block. On the right side of the  block add the  . One the left side of the  add the  block. Add the chosen RPM to the left side of the  block and 60 to the right side.

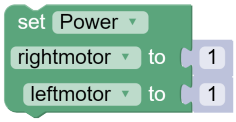


Now that the target ticks per second has been set, swap the  block for a  block. Add the  to both motors.

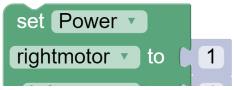


With the velocity set, this is the final thing needed to complete the objective of driving in a straight line. Consider adding telemetry and other hardware components as you begin fleshing out your full autonomous code.

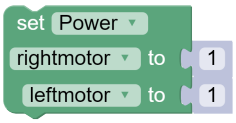
Turning the Drivetrain Using RUN_TO_POSITION

In the Robot Navigation - Blocks section, the mechanism of 

was discussed.

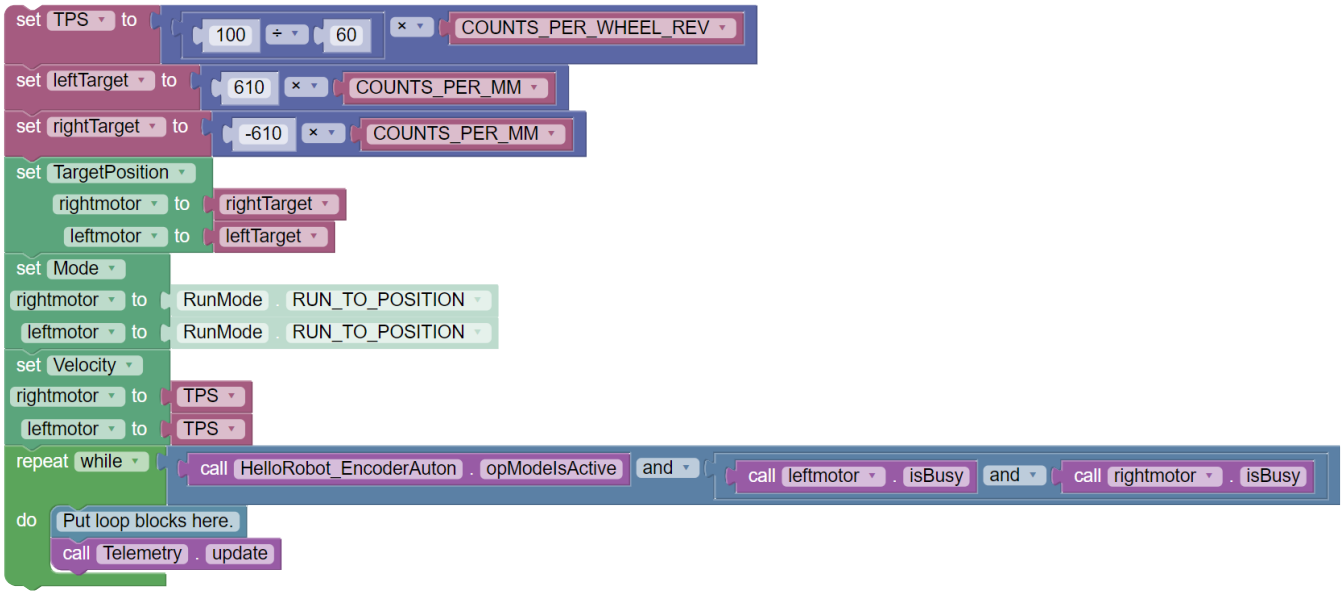


dictates what direction and speed a motor moves in. On a drivetrain the combined

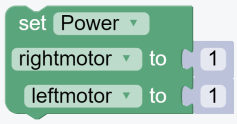
In `RUN_TO_POSITION` mode the encoder counts are used instead 

of to dictate directionality of the motor. If a target position value is greater than the current position of the encoder, the motor moves forward. If the target position value is less than the current position of the encoder, the motor moves backwards

Since speed an directionality impacts how a robot turns, target position and velocity need to be edited to get the robot to turn. Consider the following code:



The `rightTarget` has been changed to be a negative target position. Assuming that the encoder starts at zero due to `STOP_AND_RESET_ENCODER` this causes the robot to turn to the right. Velocity is the same for both motors. If you try running this code, you may notice that the robot pivots along its center of rotation. To get a wider turn changing the velocity so that the right motor is running at a lower velocity than the left motor. Adjust the velocity and target position as needed to get the turn you need.

> For more information on how direction and speed impact the movement of a robot please refer to the explanation of 
>
> in the Robot Navigation section.

Robot Navigation - OnBot Java

Introduction to Robot Navigation

As alluded to in the Hello Robot - Robot Control section, robot control comes in many different forms. One of the control types to consider for robots with drivetrains, is robot navigation.

Robot navigation as a concept is dependent on the type of drivetrain and the type of operation mode. For instance, the code to control a mecanum drivetrain differs from the code used to control a differential drivetrain. There is also a difference between coding for teleoperated driving, with a gamepad, or coding for autonomous, where each movement of the robot must be defined within code.

The following section goes through some of the basics of programming for a differential drivetrain, as well as how to set up a teleoperated arcade style drivetrain code. The concepts and logic highlighted in this section will be applicable the autonomous control section Elapsed Time.

Sections	Goals of Section
Basics of Programming Drivetrains	What to consider when programming drivetrain motors and how to apply this to an arcade style teleoperated control.

Basics of Programming Drivetrains

Programming Drivetrain Motors

Start by creating a basic op mode called `DualDrive`.

> ⓘ Visit the Test Bed - OnBot Java section for more information on creating an op mode. The op mode below focuses on hardware mapping only the relevant drivetrain motors.

```
package org.firstinspires.ftc.teamcode;

import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.DcMotorSimple;

@TeleOp
public class DualDrive extends LinearOpMode {
  private DcMotor rightmotor;
  private DcMotor leftmotor;

  @Override
  public void runOpMode() {
```

```
        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");

        waitForStart();

        while (opModeIsActive()) {

        }
    }
}
```

Since the focus of this section is creating a functional drivetrain in code, lets started by adding `rightmotor.setPower(1);` and `leftmotor.setPower(1);` to the op more while loop.

```
while (opModeIsActive()) {
        rightmotor.setPower(1);
        leftmotor.setPower(1);
        }
```

Before moving on try running the code as is and consider the following questions:

- What behavior is the robot exhibiting?
- What direction is the robot spinning in?

When motors run at different speeds they spin along their center pivot point. But the motors are both set to a power (or duty cycle) of 1?

DC Motors are capable of spinning in two different directions depending on the current flow: clockwise and counter clockwise. When using a positive power value the Control Hub sends current to the motor for it to spin in a clockwise direction.

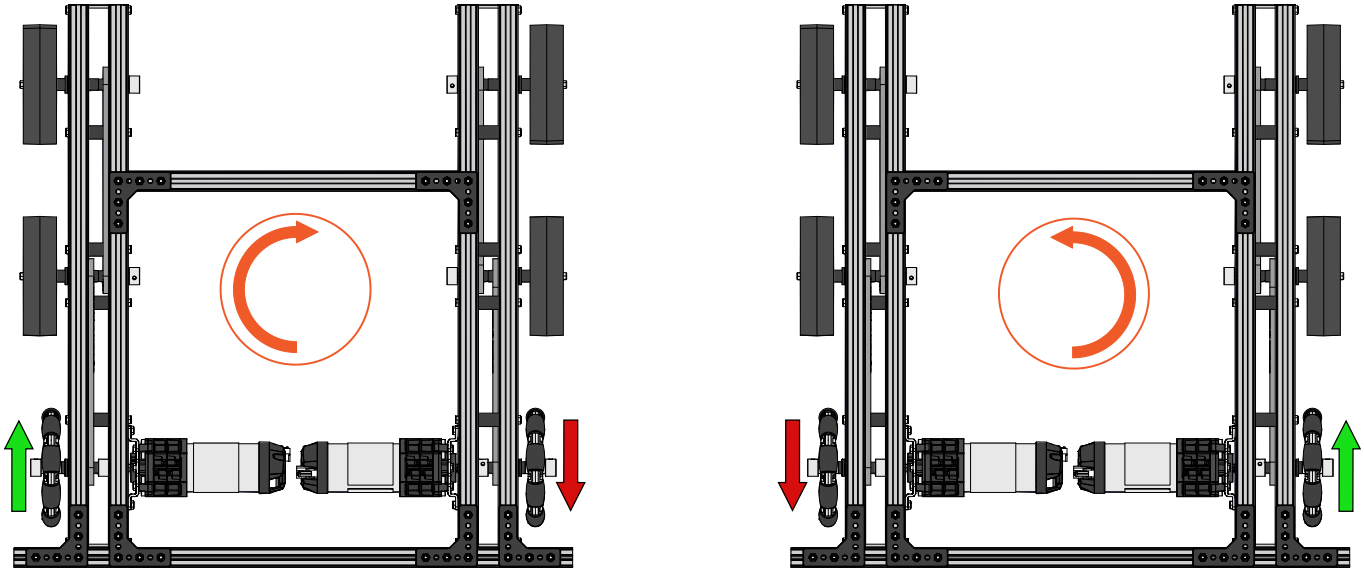
With the Class Bot and current code, both motors are currently set to run in the clockwise direction. If you set the robot on blocks and run the code again though, you can see that the motors run in opposing directions. With the mirrored way the motors mount to the drivetrain, one motor is naturally the inverse of the other.

Why would the inverse motor cause the robot to spin in a circle? Both speed and direction of rotation of the **wheels** impact the overall direction the robot moves in. In this case, both **motors** were assigned to have the same power and direction but how the motors transfer motion to the **wheels**, causes the robot to spin instead of moving forward.

Check the Introduction to Motion section for more information on the mechanics of transferring motion and power.

In the info block above you were asked to determine which direction the robot spun in. The robot pivots in the direction of the inversed motor. For instance, when the right motor is the inversed motor the robot will pivot to the right. If the left motor is the inversed motor the robot will pivot to the left.

The Affect of Drivetrain Motors on Drivetrain Movement



For the Class Bot, the robot pivots to the right, so the right motor will be reversed. Add the line `rightmotor.setDirection(DcMotorSimple.Direction.REVERSE);` to the op mode under the variable declarations.

```
public void runOpMode() {
        float x;
        double y;

        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");

        rightmotor.setDirection(DcMotorSimple.Direction.REVERSE);

        waitForStart();

        while (opModeIsActive()) {
              rightmotor.setPower(1);
              leftmotor.setPower(1);
            }
   }
```

Adding the `rightmotor.setDirection(DcMotorSimple.Direction.REVERSE);` code line reverses (or inverses) the direction of the right motor. Both motors now consider the same direction forward an

Teleoperated Driving - Arcade Style

Recall that when the motors were running in opposing directions the robot spun in circles. This same logic will be used to control the robot using the arcade style of control mentioned in the Hello Robot - Autonomous Robot section.

Programming with gamepads

To start, create two variables x and y. Both variables will be doubles.

```
public void runOpMode() {
        double x;
        double y;

        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");

        rightmotor.setDirection(DcMotorSimple.Direction.REVERSE);

        waitForStart();
```

Assign y as `y = -gamepad1.right_stick_y;`, which is the y-axis of the right joystick.

> ⓘ Remember positive/negative values inputted by the gamepad's y-axis are inverse of the positive/negative values of the motor.

Assign the x as the `x = gamepad1.right_stick_x;`, which is the x axis of the right gamepad joystick. The x-axis of the joystick does not need to be inverted.

```
while (opModeIsActive()) {
        x = gamepad1.right_stick_x;
        y = -gamepad1.right_stick_y;

        rightmotor.setPower(1);
        leftmotor.setPower(1);
        }
```

Setting `x = gamepad1.right_stick_x;` and `y = -gamepad1.right_stick_y;` assigns values from the gamepad joystick to x and y. As previously mentioned, the joystick gives values along a two dimension coordinate system. y receives the value from the y- axis and x receives the value from the x-axis. Both axis output values between -1 and 1.

To better understand consider the following table. The table shows the expected value generated from moving the joystick all the way in one direction, along the axis. For instance, when the joystick is pushed all the way in the upwards direction the coordinate values are (0,1).

> ⓘ The table below assumes that the the y values from the gamepad have been inverted in code when assigned to the variable y.

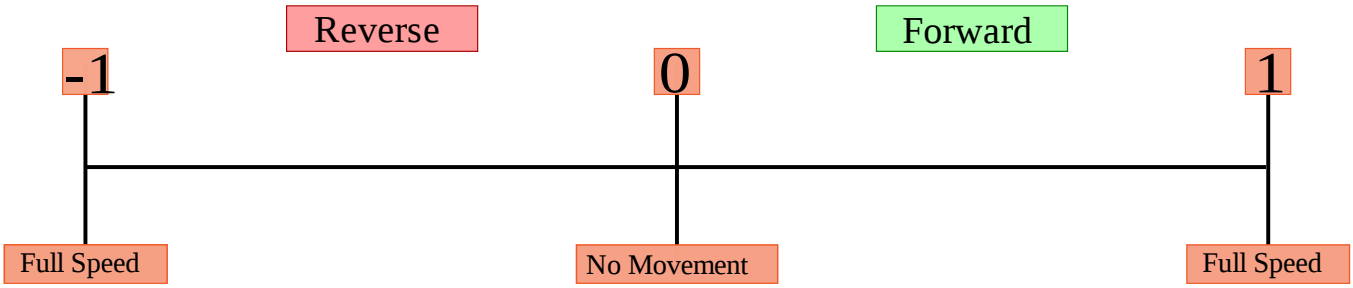
Joystick Direction	x	y
	0	1
	0	-1
	-1	0
	1	0

Now that you have a better understanding of how the physical movement of the gamepad affects the numerical inputs given to your Control System; its time to consider how to control the drivetrain using the joystick.

> Recall, from the Programming Drivetrain Motors section, that the speed and direction of a motor

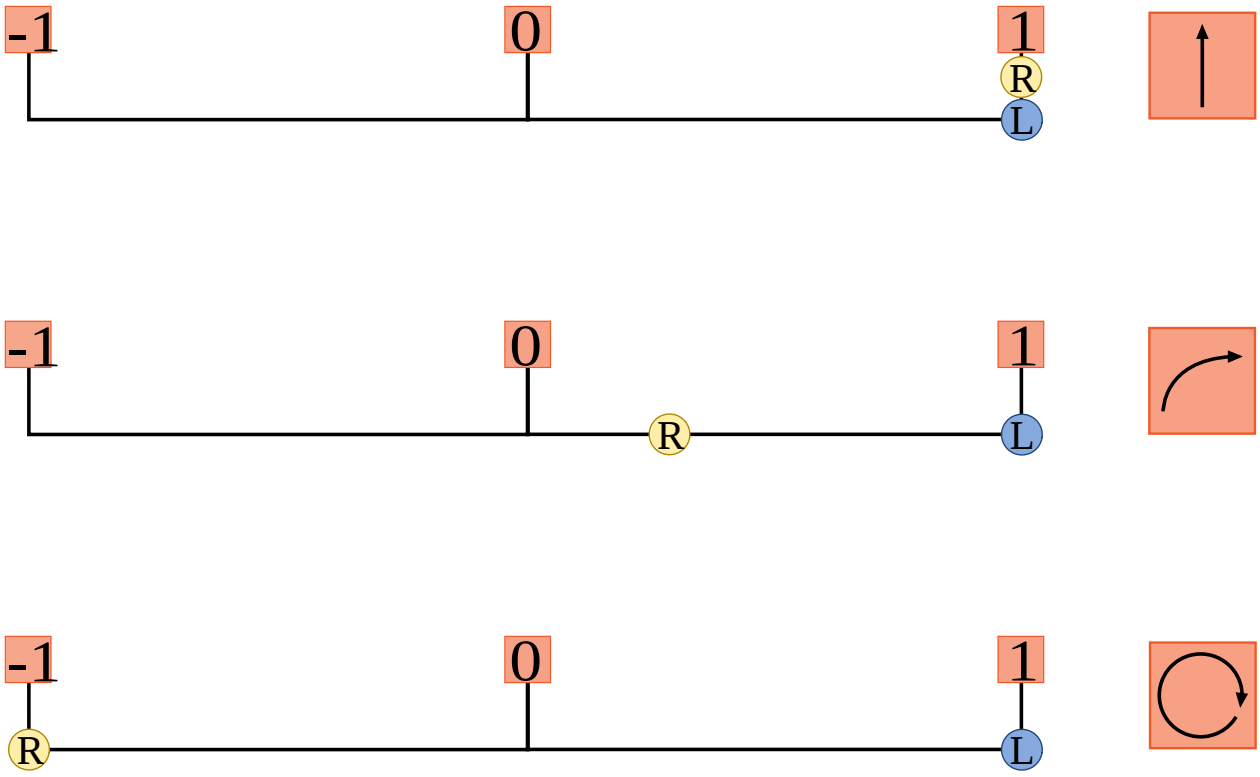
> plays a large part in how the drivetrain moves.

The numerical outputs for `setPower` determine the speed and direction of the motors. For instance, when both motors are set to 1 they move in the forward direction at full speed (or 100% of duty cycle). Much like the gamepads, the numerical value for `setPower` is in a range of -1 to 1. The absolute value of the assigned number determines percentage of duty cycle. As an example, 0.3 and -0.3 both indicate that the motor is operating at a duty cycle of 30%. The sign of the number indicates the direction the motor is rotating in. To better understand, consider the following graphic.



When a motor is assigned a `setPower` value between -1 and 0, the motor will rotate in the direction it considers to be reverse. When a motor is assigned a value between 0 and 1, it will rotate forward.

In the Programming Drivetrain Motors section, it was discussed that a robot rotates when the motors are moving in opposing directions. However, this has more to do with both speed and direction. To think of it numerically, a differential drivetrain will turn to the right when the `setPower` value for the right motor is less than that of the left motor. This is exhibited in the following example.



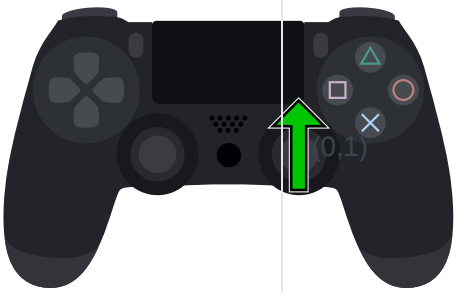
When both motors are `rightmotor.setPower(1); leftmotor.setPower(1);` the robot will run at

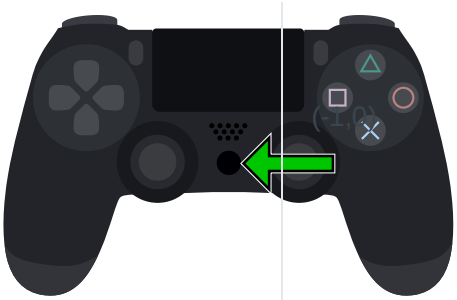
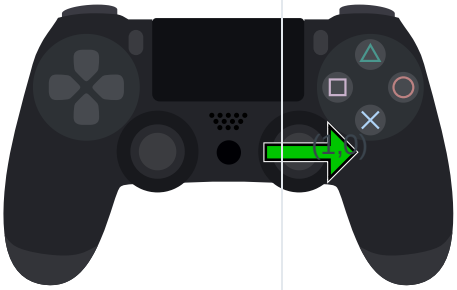
full speed in a mostly straight line. However, when the rightmotor is running in the same direction but at a lower speed, such as `rightmotor.setPower(0.3); leftmotor.setPower(1);` the robot will turn or rotate to the right. This is likely to be an arching movement that is not as sharp as a full pivot. In contrast when the `rightmotor` is set to full speed but in the opposite direction of the `leftmotor`, the robot pivots to the right. So, mathematically the following is considered to be true:

`rightmotor.setPower = leftmotor.setPower`	Forward or Reverse
`rightmotor.setPower > leftmotor.setPower`	Left Turn
`rightmotor.setPower < leftmotor.setPower`	Right Turn

As previously implied, `gamepad1.right_stick_y` and `gamepad1.right_stick_x` send values to the Control System from the game pad joystick. In contrast, the `setPower` function interprets numerical information set in the code and sends the appropriate current to the motors to dictate how the motors behave.

In an arcade drive, the following joy stick inputs (directions) need to correspond with the following outputs (motor power values).

Joystick Direction	(X,Y)	rightmotor	leftmotor
	(0,1)	1	1
	(0,-1)	-1	-1

	1	-1
	-1	1

To get the outputs expressed in the table above, the gamepad values must be assigned to each motor in a meaningful way, where. Algebraic principles can be used to determine the two formulas needed to get the values. However, the formulas are provided below.

$$rightmotor = y - x$$
$$leftmotor = y + x$$

Rather than `setPower(1);` both the motors can be set to the above formulas. For instance, the right motor can be set as `rightmotor.setPower(y-x);`.

```
while (opModeIsActive()) {
        x = gamepad1.right_stick_x;
        y = -gamepad1.right_stick_y;

        rightmotor.setPower(y-x);
        leftmotor.setPower(y+x);
        }
```

With this you now have a functional teleoperated arcade drive. From here you can start adding hardware mapping for the other pieces of robot hardware. Below is an outline of the expected code for the Class Bot with full hardware mapping.

```
package org.firstinspires.ftc.teamcode;

import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.hardware.Blinker;
import com.qualcomm.robotcore.hardware.Servo;
import com.qualcomm.robotcore.hardware.Gyroscope;
import com.qualcomm.robotcore.hardware.DigitalChannel;
```

```java
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.eventloop.opmode.Disabled;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.DcMotorSimple;
import com.qualcomm.robotcore.util.ElapsedTime;

@TeleOp

public class DualDrive extends LinearOpMode {
    private Blinker control_Hub;
    private DcMotor arm;
    private Servo claw;
    private Gyroscope imu;
    private DcMotor leftmotor;
    private DcMotor rightmotor;
    private DigitalChannel touch;


    @Override
    public void runOpMode() {
        double x;
        double y;

        control_Hub = hardwareMap.get(Blinker.class, "Control Hub");
        arm = hardwareMap.get(DcMotor.class, "arm");
        claw = hardwareMap.get(Servo.class, "claw");
        imu = hardwareMap.get(Gyroscope.class, "imu");
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");
        touch = hardwareMap.get(DigitalChannel.class, "touch");

        rightmotor.setDirection(DcMotorSimple.Direction.REVERSE);

        telemetry.addData("Status", "Initialized");
        telemetry.update();

        // Wait for the game to start (driver presses PLAY)
        waitForStart();

        // run until the end of the match (driver presses STOP)
        while (opModeIsActive()) {
              x = gamepad1.right_stick_x;
              y = -gamepad1.right_stick_y;

              rightmotor.setPower(y-x);
              leftmotor.setPower(y+x);

            telemetry.addData("Status", "Running");
            telemetry.update();

        }
    }
```

Elapsed Time - OnBot Java

Introduction to Elapsed Time

One way to create an autonomous code is to use a timer to define which actions should occur when. Within the SDK actions can be set to a timer by using **ElapsedTime.**

Timers consist of two main categories: count up and count down. In most applications a timer is considered to be a device that counts down from a specified time interval. For instance, the timer on a phone or a microwave. However, some timers, like stopwatches, count upwards from zero. These types of timers measure the amount of time that has elapsed.

ElapsedTime is a count up timer. Registering the amount of time elapsed from the start of a set event, like the starting of a stopwatch. In this case, it is the amount of time elapsed from when the timer is instantiated or reset within the code.

The ElapsedTime timer starts counting the amount of time elapsed from the point of its creation within a code. For instance, in this section ElapsedTime will be created (or instantiated) in the section of code that occurs when the op mode is initialized. There is no option to stop the ElapsedTime timer. Instead, the reset() function can be used within your code to reset the timer at various intervals.

One the timer has been reset, the amount of time that has elapsed can be queried by calling methods like time(), seconds(), or milliseconds(). The time given by the queried methods can be used in loops to dictate how long a specific action should take place.

> ⓘ For more information on the ElapsedTime object check out the Java Docs.

Sections	Goals of Section
Basics of programming with Elapsed Time	Learning the logic needed to use elapsed time for autonomous control.

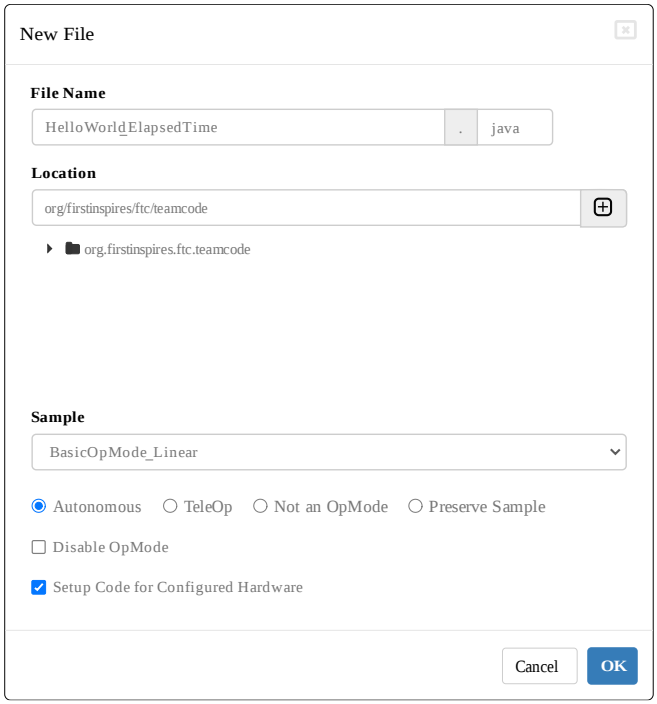
Programming with Elapsed Time

Start by creating a new op mode called `HelloWorld_ElapsedTime` using the `BasicOpMode_Linear` sample. There are other feature you can select that may make things easier as you begin to develop your autonomous op modes. For instance, as you may recall, selecting **Setup Code for Configured Hardware** creates the necessary references to the hardware map. Another selection you can make is for the code to be setup as an autonomous op mode. This adds the `@Autonomous` annotation that distinguishes the code as an autonomous op mode in the Driver Station Application.

> ⓘ

When creating an op mode a decision needs to be made on whether or not to set it to autonomous mode. For applications under 30 seconds, typically required for competitive game play changing the op mode type to autonomous is recommended. For applications over 30 seconds, setting the code to the autonomous op mode type will limit your autonomous code to 30 seconds of run time. If you plan on exceeding the 30 seconds built into the SDK, keeping the code as a teleoperated op mode type is recommended.

For information on how op modes work please visit the Introduction to Programming section.



Selecting the features discussed above will allow you to start with the following code.

```
package org.firstinspires.ftc.teamcode;

import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.hardware.AnalogInput;
import com.qualcomm.robotcore.hardware.Gyroscope;
import com.qualcomm.robotcore.hardware.ColorSensor;
import com.qualcomm.robotcore.hardware.Servo;
import com.qualcomm.robotcore.hardware.DigitalChannel;
import com.qualcomm.robotcore.eventloop.opmode.Autonomous;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.eventloop.opmode.Disabled;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.DcMotorSimple;


@Autonomous

public class HelloWorld_ElapsedTime extends LinearOpMode {
    private DcMotor leftMotor;
    private DcMotor rightMotor;
    private DcMotor arm;
```

```
    private Servo claw;
    private DigitalChannel touch;
    private Gyroscope imu;

    @Override
    public void runOpMode() {
        imu = hardwareMap.get(Gyroscope.class, "imu");
        leftMotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightMotor = hardwareMap.get(DcMotor.class, "rightmotor");
        arm = hardwareMap.get(DcMotor.class, "arm");
        claw = hardwareMap.get(Servo.class, "claw");
        touch = hardwareMap.get(DigitalChannel.class, "touch");


        telemetry.addData("Status", "Initialized");
        telemetry.update();

        // Wait for the game to start (driver presses PLAY)
        waitForStart();

        // run until the end of the match (driver presses STOP)
        while (opModeIsActive()){
            telemetry.addData("Status", "Running");
            telemetry.update();
        }
    }
}
```

Since the focus of this section is Elapsed Time, a variable of `ElapsedTime` and an instance of `ElapsedTime` needs to be created. To do this the following line is needed

```
private ElapsedTime     runtime = new ElapsedTime();
```

The above line performs two actions. A private `ElapsedTime` variable called `runtime` is created. Once `runtime` is created and defined as an `ElapsedTime` variable, it can hold the relevant time information and data. The other part of the line `runtime = new ElapsedTime();` creates an instance of the `ElapsedTime` timer object and assigns it to the runtime variable.

Add this line to the op mode with the other private variables.

```
public class HelloWorld_ElapsedTime extends LinearOpMode {
    private DcMotor leftMotor;
    private DcMotor rightMotor;
    private DcMotor arm;
    private Servo claw;
    private DigitalChannel touch;
    private Gyroscope imu;
    private ElapsedTime     runtime = new ElapsedTime();
```

The goal for this example is to have a series of actions performed on timed intervals, like driving forward for

three seconds. Another way to think about it is that the robot drives forward while the `ElapsedTime` timer is less than or equal to three seconds or `runtime.seconds() <= 3.0`. For this particular example the best way to achieve this goal is to use a while loop. Replace the default op mode while loop with the following loop.

```
        waitForStart();
        while (runtime.seconds() <= 3.0) {

         }
```

> It is important to know that, within a linear op mode, a while loop must always have the `opModeIsActive()` Boolean as a condition. This condition ensures that the while loop will terminate when the stop button is pressed.

While loops run when the condition is true and stop when the condition is false. In this case, the while loop should only start if both conditions (`opModeIsActive()` and `runtime.seconds() <= 3.0`) are true. The while loop should terminate when the `runtime.seconds() > 3` is greater than three seconds or the stop button on the driver station is pressed. To accomplish this the logical operator `&&` needs to be used.

> `&&` is a logical operator in Java. This symbol is the Java equivalent of "and." Using this in a conditional statement requires that both statements need to be true in order for the overall condition to be true.

```
        waitForStart();
        while (opModeIsActive() && (runtime.seconds() <= 3.0)) {

         }
```

Recall that the `ElapsedTime` timer starts when it is instantiated or reset. Since the timer is being instantiated when the runtime variable is being created, and the variable creations are happening before the `waitForStart();` command is written; the timer will start when the op mode is initialized rather than when the op mode is started. This can cause issues on consistency in the robots performance, depending on the delay between initialization and start.

> **Consider the following scenario:**
>
> In a competition setting, teams are often required to initialize their robot prior to the start of a match. This means that a robot can sit in initialization anywhere from a few seconds to a few minutes. If an autonomous code is centered around using an `ElapsedTime` timer that begins upon instantiation, the longer a robot is sitting in initialization the less likely it is to run as expected.

In order to avoid issues from a time delay between initialization and start, a timer reset can be added to the code. Add the line `runtime.reset();` between the `waitForStart();` command and the while loop

```
        waitForStart();
        runtime.reset();
        while (opModeIsActive() && (runtime.seconds() <= 3.0)) {

         }
```

Now the timer is reset , lets go ahead and add the motor related code. If you recall from Programming Drivetrain Motors article; the motors on the drivetrain mirror each other. The mirrored nature of the motor mounting causes the motors to rotate in opposing directions. In order to remedy this discrepancy the direction of the right motor needs to be reversed. Add the following lines of code to the op mode above the `waitForStart();` command.

```
rightMotor.setDirection(DcMotor.Direction.REVERSE);
```

Now, within the while loop add the lines `leftmotor.setPower(1);` and `rightmotor.setPower(1);` to set both motors to run at full speed in the forward direction.

```
package org.firstinspires.ftc.teamcode;

import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.hardware.AnalogInput;
import com.qualcomm.robotcore.hardware.Gyroscope;
import com.qualcomm.robotcore.hardware.ColorSensor;
import com.qualcomm.robotcore.hardware.Servo;
import com.qualcomm.robotcore.hardware.DigitalChannel;
import com.qualcomm.robotcore.eventloop.opmode.Autonomous;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.eventloop.opmode.Disabled;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.DcMotorSimple;
import com.qualcomm.robotcore.util.ElapsedTime;


@Autonomous

public class HelloWorld_ElapsedTime extends LinearOpMode {
    private DcMotor leftMotor;
    private DcMotor rightMotor;
    private DcMotor arm;
    private Servo claw;
    private DigitalChannel touch;
    private Gyroscope imu;
    private ElapsedTime     runtime = new ElapsedTime();

    @Override
    public void runOpMode() {
        imu = hardwareMap.get(Gyroscope.class, "imu");
```

```
        leftMotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightMotor = hardwareMap.get(DcMotor.class, "rightmotor");
        arm = hardwareMap.get(DcMotor.class, "arm");
        claw = hardwareMap.get(Servo.class, "claw");
        touch = hardwareMap.get(DigitalChannel.class, "touch");

        rightMotor.setDirection(DcMotor.Direction.REVERSE);

        telemetry.addData("Status", "Initialized");
        telemetry.update();
        // Wait for the game to start (driver presses PLAY)

        waitForStart();
        // run until the end of the match (driver presses STOP)

        runtime.reset();
        while (opModeIsActive() && (runtime.seconds() <= 3.0)) {
            leftMotor.setPower(1);
            rightMotor.setPower(1);
        }
```

You now have the basic code you need to have your robot drive forward for three seconds. This should give you a basic sense of coding with `ElapsedTime`. Other actions like opening and closing a claw, or lifting an arm can be coded into your autonomous program.

As advised in previous sections, it is beneficial to add telemetry to certain code to get the feedback data you want or need. For this example, the telemetry will display how many seconds have elapsed for each leg of the robots journey.

```
while (opModeIsActive() && (runtime.seconds() <= 3.0)) {
    leftMotor.setPower(1);
    rightMotor.setPower(1);
    telemetry.addData("Leg 1", runtime.seconds());
    telemetry.update();
        }
```

For this particular guide, the end goal is to test the accuracy of a robot driving forward from point a to point b and then driving backwards back to point a. In order to do that another section of code based off the timer needs to be written. One way to do this is to to copy the while loop that you already made and make the necessary edits like switching the direction of power to the motors.

```
runtime.reset();
while (opModeIsActive() && (runtime.seconds() <= 3.0)) {
    leftMotor.setPower(1);
    rightMotor.setPower(1);
    telemetry.addData("Leg 1", runtime.seconds());
    telemetry.update();
        }

runtime.reset();
```

```
while (opModeIsActive() && (runtime.seconds() <= 3.0)) {
    leftMotor.setPower(-1);
    rightMotor.setPower(-1);
    telemetry.addData("Leg 2", runtime.seconds());
    telemetry.update();
        }
```

> Notice that an additional `runtime.reset();` was added to the code above. The other option for a second while loop would have involved adding an additional condition to the while loop. Such as:
>
> `while(opModeIsActive() && (runtime.seconds() > 3.0)  && runtime.seconds() <=6.0)`
>
> The choice to reset the timer before starting a new leg of the robots journey was made to reduce the amount of code changes that may need to be made while testing the code.

Full Code Example

```
package org.firstinspires.ftc.teamcode;

import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.hardware.AnalogInput;
import com.qualcomm.robotcore.hardware.Gyroscope;
import com.qualcomm.robotcore.hardware.ColorSensor;
import com.qualcomm.robotcore.hardware.Servo;
import com.qualcomm.robotcore.hardware.DigitalChannel;
import com.qualcomm.robotcore.eventloop.opmode.Autonomous;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.eventloop.opmode.Disabled;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.DcMotorSimple;
import com.qualcomm.robotcore.util.ElapsedTime;


@Autonomous

public class HelloWorld_ElapsedTime extends LinearOpMode {
    private DcMotor leftMotor;
    private DcMotor rightMotor;
    private DcMotor arm;
    private Servo claw;
    private DigitalChannel touch;
    private Gyroscope imu;
    private ElapsedTime     runtime = new ElapsedTime();

    @Override
    public void runOpMode() {
        imu = hardwareMap.get(Gyroscope.class, "imu");
```

```
        leftMotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightMotor = hardwareMap.get(DcMotor.class, "rightmotor");
        arm = hardwareMap.get(DcMotor.class, "arm");
        claw = hardwareMap.get(Servo.class, "claw");
        touch = hardwareMap.get(DigitalChannel.class, "touch");
        leftMotor.setDirection(DcMotor.Direction.FORWARD); // Set to REVERSE if using AndyMarl
        rightMotor.setDirection(DcMotor.Direction.REVERSE);

        telemetry.addData("Status", "Initialized");
        telemetry.update();
        // Wait for the game to start (driver presses PLAY)
        waitForStart();

        // run until the end of the match (driver presses STOP)

        runtime.reset();
        while (opModeIsActive() && (runtime.seconds() <= 3.0)) {
            leftMotor.setPower(1);
            rightMotor.setPower(1);
            telemetry.addData("Leg 1", runtime.seconds());
            telemetry.update();
        }

        runtime.reset();
        while (opModeIsActive() && (runtime.seconds() <= 3.0)) {
            leftMotor.setPower(-1);
            rightMotor.setPower(-1);
            telemetry.addData("Leg 2", runtime.seconds());
            telemetry.update();
        }

        }
    }
```

Encoder Navigation - OnBot

In the previous section you learned about how to use Elapsed Time to allow your robot to navigate the world around it autonomously. When starting out many of the robot actions can be accomplished by turning on a motor for a specific amount of time. Eventually, these time-based actions may not be accurate or repeatable enough. Environmental factors, such as the state of battery charge during operation and mechanisms wearing in through use, can all affect time-based actions. Fortunately, there is a way to give feedback to the robot about how it is operating by using sensors; devices that are used to collect information about the robot and the environment around it.

With Elapsed Time, in order to get the robot to move to a specific distance, you had to estimate the amount of time and the percentage of duty cycle needed to get from point a to point b. However, the REV motors come with built in encoders, which provide feedback in the form of ticks (or counts) per revolution of the

motor. The information provided by the encoders can be used to move the motor to a target position, or a target distance.

Moving the motors to a specific position, using the encoders, removes any potential inaccuracies or inconsistencies from using Elapsed Time. The focus of this section is to move the robot to a target position using encoders.

> There are two articles in that go through the basics of Encoders. Using Encoders goes through the basics of the different types of motor modes, as well as a few application examples of using these modes in code. In this section we will focus on using `RUN_TO_POSITION`.
>
> The other article, Encoders, focuses on the general functionality of an encoder.
>
> It is recommended that you review both articles before moving on with this guide.

Basics of Programming with Encoders

Start by creating a basic op mode called `HelloRobot_EncoderAuton`.

> When creating an op mode a decision needs to be made on whether or not to set it to autonomous mode. For applications under 30 seconds, typically required for competitive game play changing the op mode type to autonomous is recommended. For applications over 30 seconds, setting the code to the autonomous op mode type will limit your autonomous code to 30 seconds of run time. If you plan on exceeding the 30 seconds built into the SDK, keeping the code as a teleoperated op mode type is recommended.
>
> For information on how op modes work please visit the Introduction to Programming section.
>
> For more information on how to change the op mode type check out the Test Bed - OnBot Java section.

> The op mode structure below is simplified and only includes the necessary components needed to create the encoder based code.

```
package org.firstinspires.ftc.teamcode;

import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.eventloop.opmode.Autonomous;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.eventloop.opmode.Disabled;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.DcMotorSimple;
```

```
@Autonomous //sets the op mode as an autonomous op mode
public class HelloWorld_EncoderAuton extends LinearOpMode {
    private DcMotor leftmotor;
    private DcMotor rightmotor;

    @Override
    public void runOpMode() {
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");

        // Wait for the game to start (driver presses PLAY)
        waitForStart();

        // run until the end of the match (driver presses STOP)
        while (opModeIsActive()){

        }
    }
}
```

As with all drivetrain related navigation, the directionality of one of the motors needs to be reversed in order for both motors to move in the same direction. Since the Class Bot V2 is still being used add the line `rightmotor.setDirection(DcMotor.Direction.REVERSE);` to the code beneath the `rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");` code line.

```
public void runOpMode() {
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");

        rightmotor.setDirection(DcMotor.Direction.REVERSE);

        waitForStart();
```

> ⓘ For more information on the directionality of motor check out the Basics of Programming Drivetrains section.

Recall from Using Encoders that using `RUN_TO_POSITION` mode requires a three step process. The first step is setting target position. To set target position add the lines `leftmotor.setTargetPosition(1000);` and `rightmotor.setTargetPosition(1000);` to the op mode after the `waitForStart();` command. To get a target position that equates to a target distance requires so calculations, which will be covered later. For now, set target position to 1000 ticks.

```
waitForStart();

leftmotor.setTargetPosition(1000);
rightmotor.setTargetPosition(1000);
```

```
while (opModeIsActive()){

        }
```

The next step is to set both motors to the `RUN_TO_POSITION` mode. Add the lines `leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);` and `rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);` to your code, beneath the `setTargetPosition` code lines.

```
waitForStart();

leftmotor.setTargetPosition(1000);
rightmotor.setTargetPosition(1000);

leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);
rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);

while (opModeIsActive()){

        }
```

The main focus of the three step process is to set a target, tell the robot to move to that target, and at what speed (or velocity) the robot should get to that target. Normally, the recommended next step is to calculate velocity and set a target velocity based on ticks. However, this requires quite a bit of math to find the appropriate velocity. For testing purposes, its more important to make sure that the main part of the code is working before getting too deep into the creation of the code. Since the `setPower` function was covered in previous sections and will communicate to the system what relative speed (or in this case duty cycle) is needed to get to the target, this can be used in the place of `setVelocity` for now.

Add the lines to set the power of both motors to 80% of duty cycle.

```
waitForStart();

leftmotor.setTargetPosition(1000);
rightmotor.setTargetPosition(1000);

leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);
rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);

leftmotor.setPower(0.8);
rightmotor.setPower(0.8);

while (opModeIsActive()){

        }
```

Now that all three `RUN_TO_POSITION` steps have been added to the code the code can be tested. However, if you want to wait for the motor to reach its target position before continuing in your program, you

can use a while loop that checks if the motor is busy (not yet at its target). For this program lets edit the

> Recall that, within a linear op mode, a while loop must always have the `opModeIsActive()` Boolean as a condition. This condition ensures that the while loop will terminate when the stop button is pressed.

Edit the while loop to include the `leftmotor.isBusy()` and `righmotor.isBusy()` functions. This will check if the left motor and right motor are busy running to a target position. The while loop will stop when either motor reaches the target position.

```
while (opModeIsActive() && (leftmotor.isBusy() && rightmotor.isBusy())) {

}
```

> Right now the while loop is waiting for the either motor to reach the target. There may be occasions when you want to wait for both motors to reach their target position, in this case the following loop can be used.
>
> `while (opModeIsActive() && (leftmotor.isBusy() || rightmotor.isBusy()))`

> Save and run the op mode two times in a row. Does the robot move as expected the second time?
>
> Try turning the Control Hub off and then back on. How does the robot move?

In the Basic Encoder Concepts section, it is clarified that all encoder ports start at 0 ticks when the Control Hub is turned on. Since you did not turn off the Control Hub in between runs, the second time you ran the op mode the motors were already at, or around, the target position. When you run a code, you want to ensure that certain variables start in a known state. For the encoder ticks, this can be achieved by setting the mode to `STOP_AND_RESET_ENCODER` . Add this block to the op mode in the initialization section. Each time the op mode is initialized, the encoder ticks will be reset to zero.

```
public void runOpMode() {
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");

        rightmotor.setDirection(DcMotor.Direction.REVERSE);

        leftmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
        rightmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);

        waitForStart();
```

> For more information on the motor mode `STOP_AND_RESET_ENCODERS` check out the STOP_AND_RESET_ENCODERS section of the Using Encoders guide.

Converting Encoder Ticks to a Distance

In the previous section, the basic structure needed to use `RUN_TO_POSITION` was created. The placement of `leftmotor.setTargetPosition(1000);` and `rightmotor.setTargetPosition(1000);` within the code, set the target position to 1000 ticks. What is the distance from the starting point of the robot and the point the robot moves to after running this code?

Rather than attempt to measure, or estimate, the distance the robot moves, the encoder ticks can be converted from amount of ticks per revolution of the encoder to how many encoder ticks it takes to move the robot a unit of distance, like a millimeter or inch. Knowing the amount of ticks per a unit of measure allows you to set a specific distance. For instance, if you work through the conversion process and find out that a drivetrain takes 700 ticks to move an inch, this can be used to find the total number of ticks need to move the robot 24 inches.

> Reminder that the basis for this guide is the Class Bot V2. The REV DUO Build System is a metric system. Since part of the conversion process references the diameter of the wheels, this section will convert to ticks per mm.

For the conversion process the following information is needed:

- Ticks per revolution of the encoder
- Total gear reduction on the motor
  - Including gearboxes and motion transmission components like gears, sprockets and chain, or belts and pulleys
- Circumference of the driven wheels

Ticks per Revolution

The amount of ticks per revolution of the encoder shaft is dependent on the motor and encoder. Manufacturers of motors with built-in encoders will have information on the amount of ticks per revolution. For HD Hex Motors the encoder counts 28 ticks per revolution of the motor shaft.

> Visit the manufacturers website for your motor or encoders for more information on encoder counts. For HD Hex Motors or Core Hex Motors visit our Motor documentation.

Total Gear Reduction

Since ticks per revolution of the encoder shaft is before any gear reduction calculating the total gear reduction is needed. This includes the gearbox and any addition reduction from motion transmission

components. To find the total gear reduction use the Compound Gearing formula.
For the Class Bot V2 there are two UltraPlanetary Cartridges, 4:1 and 5:1, and an additional gear reduction from the UltraPlanetary Output to the wheels, 72T:45T ratio.

> The UltraPlanetary Cartridges use the nominal gear ratio as a descriptor. The actual gear ratios can be found in the UltraPlanetary Users Manual's Cartridge Details.

Using the compound gearing formula for the Class Bot V2 the total gear reduction is:

$$\frac{3.61}{1} * \frac{5.23}{1} * \frac{72}{45} = 30.21$$

> Unlike the the spur gears used to transfer motion to the wheels, the UltraPlanetary Gearbox Cartridges are planetary gear systems. To make calculations easier the gear ratios for the Cartridges are already reduced.

Circumference of the Wheel

The Class Bot V2 uses the 90mm Traction Wheels. 90mm is the diameter of the wheel. To get the appropriate circumference use the following formula

$$circumference = diameter * \pi$$

You can calculate this by hand, but for the purpose of this guide, this can be calculated within the code.

> Due to wear and manufacturing tolerances, the diameter of some wheels may be nominally different. For the most accurate results consider measuring your wheel to confirm that the diameter is accurate.

To summarize, for the Class Bot V2 the following information is true:

Ticks per revolution	28 ticks
Total gear reduction	30.21
Circumference of the wheel	$90mm * \pi$

Each of these pieces of information will be used to find the number of encoder ticks (or counts) per mm that the wheel moves. Rather than worry about calculating this information by hand, these values can be added

to the code as constant variables. To do this create three variables:

- `COUNTS_PER_MOTOR_REV`
- `DRIVE_GEAR_REDUCTION`
- `WHEEL_CIRCUMFERENCE_MM`

> The common naming convention for constant variables is known as CONSTANT_CASE, where the variable name is in all caps and words are separated by and underscore.

Add the variables to op mode class, where the hardware variables are defined. Defining the variables within the bounds of the class but outside of the op mode, allows them to be referenced in other methods of functions within the class. To ensure variables are referenceable they are set as `static final double` variables. Static allows references to the variables anywhere within the class and final dictates that these variables are constant and unchanged elsewhere within the code. Since these variables are not integers they are classified as double variables.

```
public class HelloWorld_EncoderAuton extends LinearOpMode {
    private DcMotor leftmotor;
    private DcMotor rightmotor;

    static final double     COUNTS_PER_MOTOR_REV    = 28.0;
    static final double     DRIVE_GEAR_REDUCTION    = 30.21;
    static final double     WHEEL_CIRCUMFERENCE_MM  = 90.0 * Math.PI;
```

Now that these three variables have been defined, they can be used to calculate two other variables: the amount of encoder counts per rotation of the wheel and the number of counts per mm that the wheel moves.

To calculate counts per wheel revolution multiple `COUNTS_PER_MOTOR_REV` by `DRIVE_GEAR_REDUCTION` Use the following formula:

$$y = a * b$$

Where,

- a = `COUNTS_PER_MOTOR_REV`
- b = `DRIVE_GEAR_REDUCTION`
- y = `COUNTS_PER_WHEEL_REV`

Create the `COUNTS_PER_WHEEL_REV` variable within the code. This will also be a `static final double` variable.

```
public class HelloWorld_EncoderAuton extends LinearOpMode {
    private DcMotor leftmotor;
    private DcMotor rightmotor;
```

```
    static final double     COUNTS_PER_MOTOR_REV    = 28.0;
    static final double     DRIVE_GEAR_REDUCTION    = 30.24;
    static final double     WHEEL_CIRCUMFERENCE_MM  = 90.0 * 3.14;

    static final double     COUNTS_PER_WHEEL_REV    = COUNTS_PER_MOTOR_REV * DRIVE_GEAR_REDUC
```

Once `COUNTS_PER_WHEEL_REV` is calculated, use it to calculate the counts per mm that the wheel moves. To do this divide the `COUNTS_PER_WHEEL_REV` by the `WHEEL_CIRCUMFERENCE_MM`. Use the following formula.

$$x = \frac{(a * b)}{c} = \frac{y}{c}$$

Where,

- a = `COUNTS_PER_MOTOR_REV`
- b = `DRIVE_GEAR_REDUCTION`
- c = `WHEEL_CIRCUMFERENCE_MM`
- y = `COUNTS_PER_WHEEL_REV`
- x = `COUNTS_PER_MM`

Create the `COUNTS_PER_MM` variable within the code. This will also be a `static final double` variable.

```
public class HelloWorld_EncoderAuton extends LinearOpMode {
    private DcMotor leftmotor;
    private DcMotor rightmotor;

    static final double     COUNTS_PER_MOTOR_REV    = 28.0;
    static final double     DRIVE_GEAR_REDUCTION    = 30.24;
    static final double     WHEEL_CIRCUMFERENCE_MM  = 90.0 * 3.14;

    static final double     COUNTS_PER_WHEEL_REV    = COUNTS_PER_MOTOR_REV * DRIVE_GEAR_REDUC
    static final double     COUNTS_PER_MM           = COUNTS_PER_WHEEL_REV / WHEEL_CIRCUMFEREI
```

> `COUNTS_PER_WHEEL_REV` will be created as a separate variable from `COUNTS_PER_MM` as it is used in calculating a target velocity.

Moving to a Target Distance

Now that you have created the constant variables needed to calculate the amount of ticks per mm moved, you can use this to set a target distance. For instance, if you would like to have the robot move forward two

feet, converting from feet to millimeters and multiplying by the `COUNTS_PER_MM` will give you the amount of counts (or ticks) needed to reach that distance.

Create two more variables called `leftTarget` and `rightTarget`. These variables can be fluctuated and edited in your code to tell the motors what positions to go to, rather than place them with the constant variables, create these variables within the op mode but above the `waitForStart();` command.

> ⓘ The `setTargetPosition();` function takes in a integer (or int) data type as its parameter, rather than a double. Since both the `leftTarget` and `rightTarget` will be used to set the target position, create both variables as int variables.

```
public void runOpMode() {
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");

        rightmotor.setDirection(DcMotor.Direction.REVERSE);

        leftmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
        rightmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);

        int leftTarget;
        int rightTarget;

        waitForStart();
```

Right now the main distance factor is `COUNTS_PER_MM` , however you may want to go a distance that is in the imperial system, such as 2 feet (or 24 inches). The target distance in this case will need to be converted to mm. To convert from feet to millimeters use the following formula:

$$d_{(mm)} = d_{(ft)} \times 304.8$$

If you convert 2 feet to millimeters, it comes out the be 609.6 millimeters. For the purpose of this guide, lets go ahead an round this to be 610 millimeters. Multiply 610 millimeters by the `COUNTS_PER_MM` variable to get the number of ticks needed to move the robot 2 feet. Since the intent is to have the robot move in a straight line, set both the `leftTarget` and `rightTarget` , to be equal to 610 * `COUNTS_PER_MM`

> ⓘ As previously mentioned the `setTargetPosition();` function requires that its parameter must be an integer data type. The `leftTarget` and `rightTarget` variables have been set to be integers, however the `COUNTS_PER_MM` variable is a double. Since these are two different data types, a conversion of data types needs to be done.
>
> In this case the `COUNTS_PER_MM` needs to be converted to an integer. This is as simple as adding the line (int) in front of the double variable. However, you need to be cautious of potential rounding errors. Since `COUNTS_PER_MM` is part of an equation it is recommended that you

convert to an integer after the result of the equation is found. The example of how to do this is exhibited below.

```
int leftTarget = (int)(610 * COUNTS_PER_MM);
int rightTarget = (int)(610 * COUNTS_PER_MM);
```

Edit the `setTargetPosition();` lines so that both motors are set to the appropriate target position. To do this add the `leftTarget` and `rightTarget` variables to their respective motor.

```
leftmotor.setTargetPosition(leftTarget);
rightmotor.setTargetPosition(rightTarget);
```

Try running the code and observing the behavior of the robot. Consider some of the following

- Is the robot moving forward by two feet?
- Does the robot seem to be moving in straight line?
- Is the code running without error?

Setting Velocity

Velocity is a closed loop control within the SDK that uses the encoder counts to determine the approximate power/speed the motors need to go in order to meet the set velocity. When working with encoder setting a velocity is recommended over setting a power level, as it offers a higher level of control.

To set a velocity, its important to understand the maximum velocity in RPM your motor is capable of. For the Class Bot V2 the motors are capable of a maximum RPM of 300. With a drivetrain, you are likely to get better control by setting velocity lower than the maximum. In this case, lets set the velocity to 175 RPM

Recall that `setVelocity` is measure in ticks per second.

Create a new double variable called `TPS`. Add `TPS` the to the op mode under where `rightTarget` is defined.

```
public void runOpMode() {
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");

        rightmotor.setDirection(DcMotor.Direction.REVERSE);

        leftmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
        rightmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);

        int leftTarget = (int)(610 * COUNTS_PER_MM);
```

```
        int rightTarget = (int)(610 * COUNTS_PER_MM);
        double TPS;

        waitForStart();
```

Since RPM is the amount of revolutions per minute a conversion needs to be made from RPM to ticks per second. To do this divide the RPM by 60, to get the amount of rotations per second. Rotations per second can the be multiplied by `COUNTS_PER_WHEEL_REV`, to get the amount of ticks per second.

$$TPS = \frac{175}{60} * CPWR$$

```
double TPS = (175/60) * COUNTS_PER_WHEEL_REV
```

Exchange the `setPower();` functions for `setVelocity();`. Add `TPS` as the parameter for `setVelocity();`.

```
waitForStart();

leftmotor.setTargetPosition(leftTarget);
rightmotor.setTargetPosition(rightTarget);

leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);
rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);

leftmotor.setVelocity(TPS);
rightmotor.setVelocity(TPS);

while (opModeIsActive() && (leftmotor.isBusy() && rightmotor.isBusy())){

        }
```

> Try to build the code. Do you get errors?

With the current state of the code you are likely to get errors similar to the ones pictured below:

```
org/firstinspires/ftc/teamcode/HelloWorld_EncoderAuton.java line 55, column 18: ERROR: cannot find symbol
  symbol:   method setVelocity(double)
  location: variable leftmotor of type com.qualcomm.robotcore.hardware.DcMotor
org/firstinspires/ftc/teamcode/HelloWorld_EncoderAuton.java line 56, column 19: ERROR: cannot find symbol
  symbol:   method setVelocity(double)
  location: variable rightmotor of type com.qualcomm.robotcore.hardware.DcMotor
```

This is because the `setVelocity();` function is a function of the `DcMotorEx` Interface. The `DcMotorEx` Interface is an extension of the `DcMotor` Interface, that provides enhanced motor

functionality, such as access to closed loop control functions. To use `setVelocity();` the motor variables need to be changed to `DcMotorEx` . To do this both the private variable creation of the motors, and the hardware mapping need to be changed to `DcMotorEx` .

```
public class HelloWorld_EncoderAuton extends LinearOpMode {
    private DcMotorEx leftmotor;
    private DcMotorEx rightmotor;
```

```
public void runOpMode() {
        leftmotor = hardwareMap.get(DcMotorEx.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotorEx.class, "rightmotor");
```

> ⓘ Since `DcMotorEx` is an extension of `DcMotor` , `DcMotor` specific functions can be used by variables defined as `DcMotorEx` .

Once you have made these changes the basic, drive two feet code is done! The code below is the finalized version of the code. In this the other hardware components and telemetry have been added.

```
@Autonomous

public class HelloWorld_EncoderAuton extends LinearOpMode {
    private DcMotorEx leftmotor;
    private DcMotorEx rightmotor;
    private DcMotor arm;
    private Servo claw;
    private DigitalChannel touch;
    private Gyroscope imu;

    static final double     COUNTS_PER_MOTOR_REV    = 28.0;
    static final double     DRIVE_GEAR_REDUCTION    = 30.24;
    static final double     WHEEL_CIRCUMFERENCE_MM  = 90.0 * 3.14;

    static final double     COUNTS_PER_WHEEL_REV    = COUNTS_PER_MOTOR_REV * DRIVE_GEAR_REDUC
    static final double     COUNTS_PER_MM           = COUNTS_PER_WHEEL_REV / WHEEL_CIRCUMFEREI

    @Override
    public void runOpMode() {
        imu = hardwareMap.get(Gyroscope.class, "imu");
        leftmotor = hardwareMap.get(DcMotorEx.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotorEx.class, "rightmotor");
        arm = hardwareMap.get(DcMotor.class, "arm");
        claw = hardwareMap.get(Servo.class, "claw");
        touch = hardwareMap.get(DigitalChannel.class, "touch");

        rightmotor.setDirection(DcMotor.Direction.REVERSE);

        leftmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
```

```
        rightmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);

        int leftTarget = (int)(610 * COUNTS_PER_MM);
        int rightTarget = (int)(610 * COUNTS_PER_MM);
        double TPS = (175/ 60) * COUNTS_PER_WHEEL_REV;

        waitForStart();


        leftmotor.setTargetPosition(leftTarget);
        rightmotor.setTargetPosition(rightTarget);

        leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);
        rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);

        leftmotor.setVelocity(TPS);
        rightmotor.setVelocity(TPS);

        while (opModeIsActive() && (leftmotor.isBusy() && rightmotor.isBusy())) {
            telemetry.addData("left", leftmotor.getCurrentPosition());
            telemetry.addData("right", rightmotor.getCurrentPosition());
            telemetry.update();
        }
    }
}
```

Turning the Drivetrain Using RUN_TO_POSITION

In the Robot Navigation - OnBot Java section, the mechanism of `setPower();` was discussed. `setPower();` dictates what direction and speed a motor moves in. On a drivetrain this dictates whether the robot moves in forward, reverse, or turns.

In `RUN_TO_POSITION` mode the encoder counts (or `setTargetPosition();`) are used instead of `setPower();` to dictate directionality of the motor. If a target position value is greater than the current position of the encoder, the motor moves forward. If the target position value is less than the current position of the encoder, the motor moves backwards

Since speed an directionality impacts how a robot turns, `setTargetPostion();` and `setVelocity();` need to be edited to get the robot to turn. Consider the following code:

```
int leftTarget = (int)(610 * COUNTS_PER_MM);
int rightTarget = (int)(-610 * COUNTS_PER_MM);
double TPS = (100/ 60) * COUNTS_PER_WHEEL_REV;


waitForStart();


leftmotor.setTargetPosition(leftTarget);
rightmotor.setTargetPosition(rightTarget);
```

```
leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);
rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);

leftmotor.setVelocity(TPS);
rightmotor.setVelocity(TPS);
```

The `rightTarget` has been changed to be a negative target position. Assuming that the encoder starts at zero due to `STOP_AND_RESET_ENCODER` this causes the robot to turn to the right. Velocity remains the same for both motors. If you try running this code, you may notice that the robot pivots along its center of rotation. To get a wider turn changing the velocity so that the right motor is running at a lower velocity than the left motor. Adjust the velocity and target position as needed to get the turn you need.

> For more information on how direction and speed impact the movement of a robot please refer to the explanation of `setPower();` in the Robot Navigation section.

The following code walks through adding a turn to the program, after the robot moves forward for 2 feet. After the robot reaches the 2 foot goal, there is a call to `STOP_AND_RESET_ENCODERS` this will reduce the need to calculate what position to go to after a position has been reached.

```
@Autonomous

public class HelloWorld_EncoderAuton extends LinearOpMode {
    private DcMotorEx leftmotor;
    private DcMotorEx rightmotor;
    private DcMotor arm;
    private Servo claw;
    private DigitalChannel touch;
    private Gyroscope imu;

    static final double     COUNTS_PER_MOTOR_REV    = 28.0;
    static final double     DRIVE_GEAR_REDUCTION    = 30.24;
    static final double     WHEEL_CIRCUMFERENCE_MM  = 90.0 * 3.14;

    static final double     COUNTS_PER_WHEEL_REV    = COUNTS_PER_MOTOR_REV * DRIVE_GEAR_REDUC
    static final double     COUNTS_PER_MM           = COUNTS_PER_WHEEL_REV / WHEEL_CIRCUMFEREI

    @Override
    public void runOpMode() {
        imu = hardwareMap.get(Gyroscope.class, "imu");
        leftmotor = hardwareMap.get(DcMotorEx.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotorEx.class, "rightmotor");
        arm = hardwareMap.get(DcMotor.class, "arm");
        claw = hardwareMap.get(Servo.class, "claw");
        touch = hardwareMap.get(DigitalChannel.class, "touch");

        rightmotor.setDirection(DcMotor.Direction.REVERSE);

        leftmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
        rightmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
```

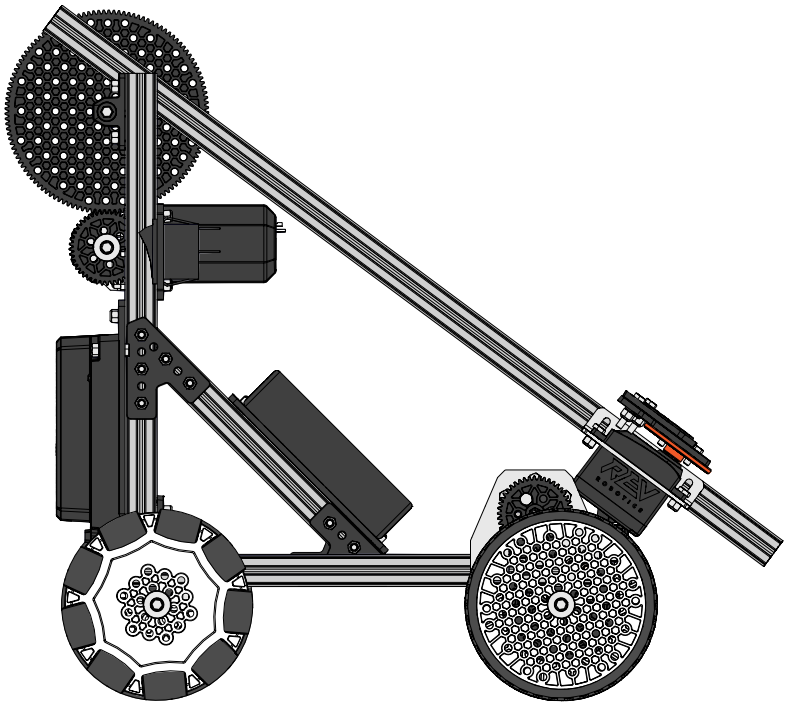
```
        // TPS variable split to change velocity for each motor when necessary

        int leftTarget = (int)(610 * COUNTS_PER_MM);
        int rightTarget = (int)(610 * COUNTS_PER_MM);
        double LTPS = (175/ 60) * COUNTS_PER_WHEEL_REV;
        double RTPS = (175/ 60) * COUNTS_PER_WHEEL_REV;

        waitForStart();


        leftmotor.setTargetPosition(leftTarget);
        rightmotor.setTargetPosition(rightTarget);

        leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);
        rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);

        leftmotor.setVelocity(LTPS);
        rightmotor.setVelocity(RTPS);

        //wait for motor to reach position before moving on
        while (opModeIsActive() && (leftmotor.isBusy() && rightmotor.isBusy())) {
            telemetry.addData("left", leftmotor.getCurrentPosition());
            telemetry.addData("right", rightmotor.getCurrentPosition());
            telemetry.update();
        }
        // Reset encoders to zero
        leftmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
        rightmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);

        // changing variables to match new needs

        leftTarget = (int)(300 * COUNTS_PER_MM);
        rightTarget = (int)( -300 * COUNTS_PER_MM);
        LTPS = (100/ 60) * COUNTS_PER_WHEEL_REV;
        RTPS = (70/ 60) * COUNTS_PER_WHEEL_REV;

        leftmotor.setTargetPosition(leftTarget);
        rightmotor.setTargetPosition(rightTarget);

        leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);
        rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);

        leftmotor.setVelocity(LTPS);
        rightmotor.setVelocity(RTPS);

        //wait for motor to reach position before moving on
        while (opModeIsActive() && (leftmotor.isBusy() && rightmotor.isBusy())) {
            telemetry.addData("left", leftmotor.getCurrentPosition());
            telemetry.addData("right", rightmotor.getCurrentPosition());
            telemetry.update();
        }
    }
```

```
}
```

Arm Control - Blocks

Introduction to Arm Control

Robot control comes in many different forms. Now that you have walked through programming a drivetrain, we can apply those concepts to controlling other mechanisms. Since this guide utilizes the Class Bot the focus will be on the basics of controlling it's main mechanism, a single jointed arm.



Controlling an arm requires a different thought process than the one you used to control the drivetrain. While the drivetrain uses the rotation motion of the motors to drive along a linear distance, an arm rotates along a central point, or joint. When working with an arm you will have to head caution to the physical limitations of the robot this includes load bearing, range of motion, and other forces that may apply.

In this section you will learn how to use the gamepad `Dpad` controls and the installed Touch Sensor to control the arm. However, the focus of this section is using code to limit the range of motion of the arm.

Sections	Goals of Section
Basics of Programming an Arm	Introduction to coding an arm for teleoperated control and working with a limit switch
Programming an Arm to a Position	Using motor encoders to move an arm to a specifi position, such as from 45 degrees to 90 degrees.
Using Limits to Control Range of Motion	Working with the basics of arm control, motor encoder, and limit switches to control the range of

	motion for an arm.

Basics of Programming an Arm

Start by creating a basic op mode called `HelloRobot_ArmControl`.

> For more information on how to create an op mode type check out the Test Bed - Blocks section.

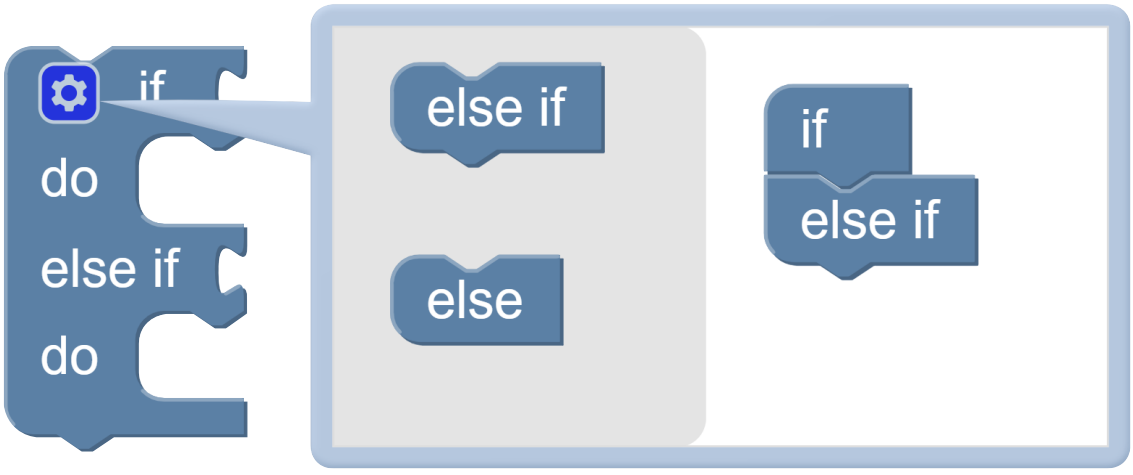
Unlike the joystick, which sends values corresponding to the position of the joystick, the `Dpad` on the gamepad inputs **Boolean** `FALSE/TRUE`. In order to tell the arm to move when `Dpad Up` or `Dpad Down` are selected, an `if/else if` statement needs to be used. Select an 

block. Use the settings drop down to change the block to an 

block. Do this by switching the 

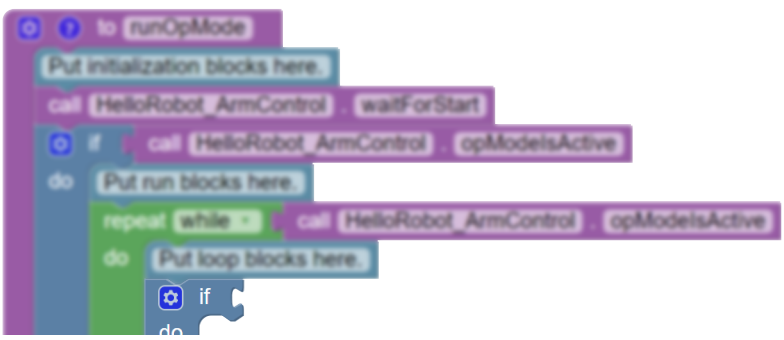
out for an 

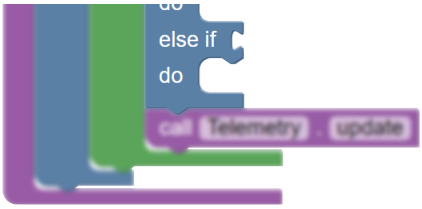
.



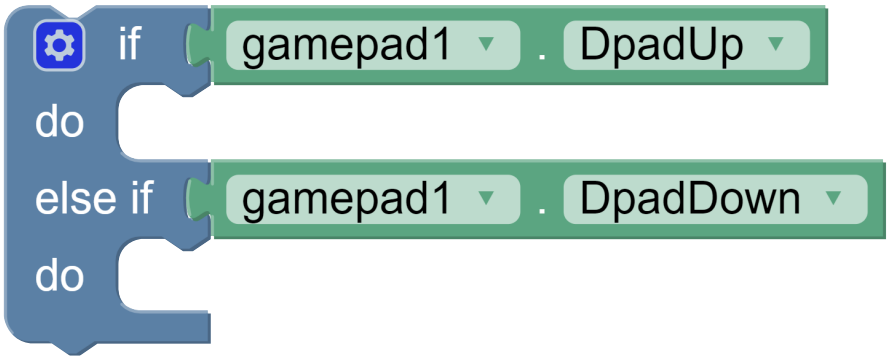
Add the 

block to the op mode while loop.

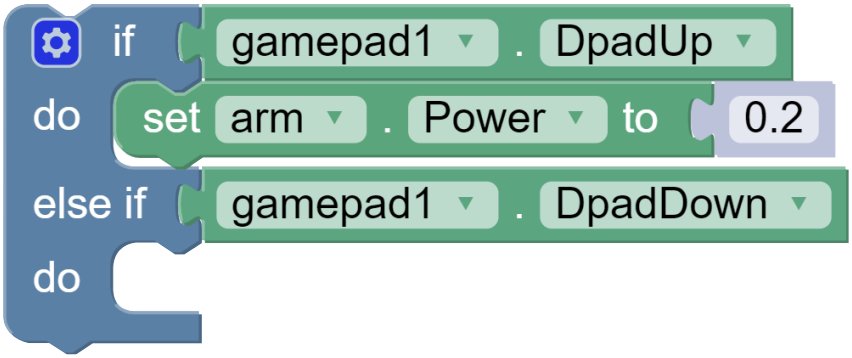




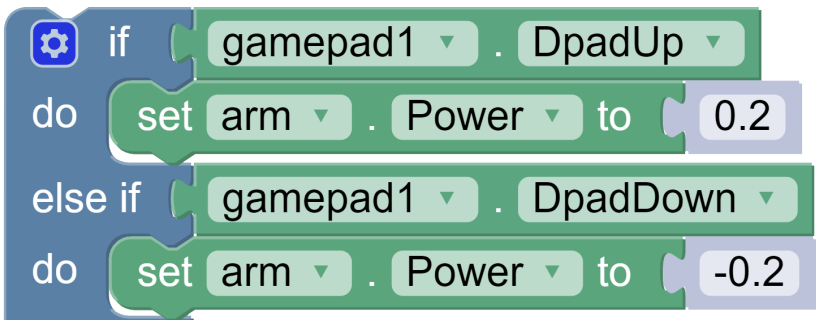
Add the gamepad1 . DpadUp and gamepad1 . DpadDown blocks to the condition statements of the if do else if do block.



One of the purposes of using the `Dpad` is to help delineate which direction the arm needs to move in. In this case, should gamepad1 . DpadUp correspond with the arm moving upwards. Add a set arm . Power to 1 block to the do portion of the if do else if do block below the gamepad1 . DpadUp block. Change the power from 1 to 0.2.



Add another set arm . Power to 1 block to the do portion of the if do else if do block below the gamepad1 . DpadDown block. Change the power from 1 to -0.2.

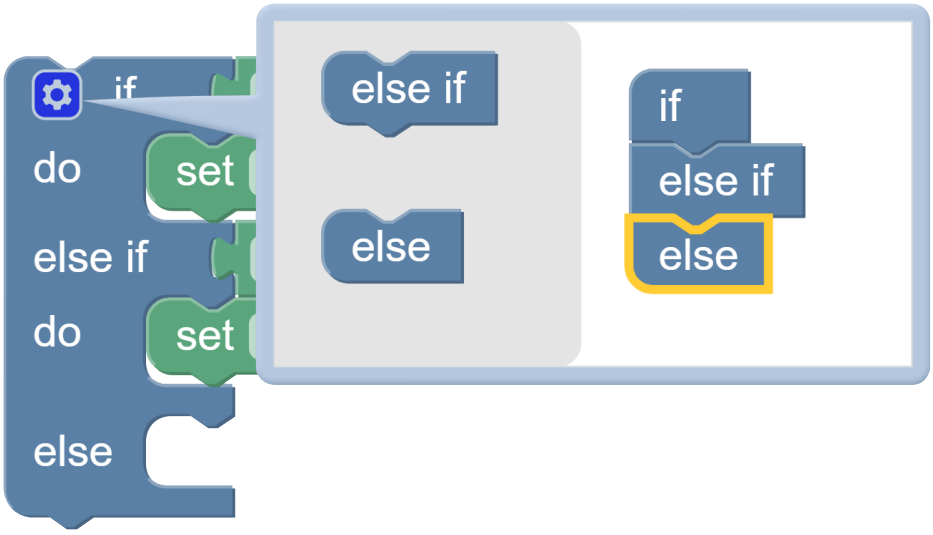


Save the op mode and try running the code. Consider the following questions.

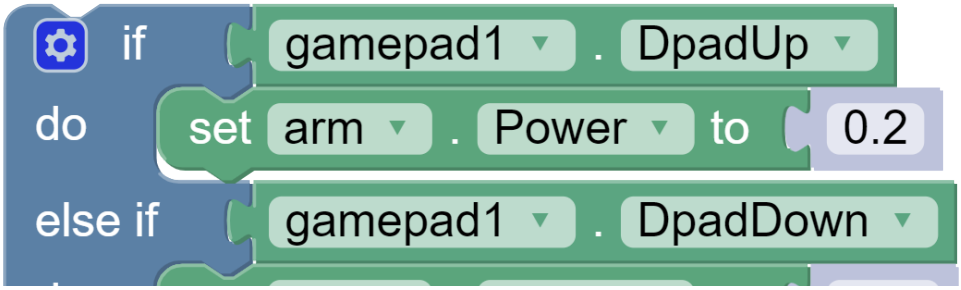
- What happens if you press up on the `Dpad` ?
- What happens if you press down on the `Dpad` ?

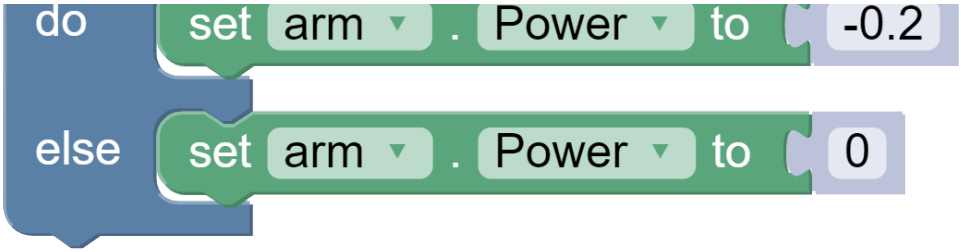
Right now the logic of the  statement declares that when  is true (has been pressed) the motor will run in the forward (or in this case upwards) direction at 20% duty cycle. If  is true the motor will run in reverse at 20% duty cycle. If you ran the code at this stage you may have noticed that even when you released the `Dpad` the motor continued to run in the selected direction. The current  statement tells the robot when the motor should move and in what direction, but nothing tells the motor to stop, thus the arm is continuing to run without limits.

To fix this edit the  block to be a  instead. To do this select the settings icon on the  block and add an  below the  .



Add the  the else portion of the  block.





> Try saving and running the op mode again. Pay attention to the speed of the arm going up versus going down. Does the speed seem the same?

Working with an arm introduces different factors for consideration than what you have seen previously with drivetrains. For instance, did you notice a difference in speeds when moving the arm up or down? Unlike the drivetrain, where the effect of gravity impacts the motors consistently across either direction, gravity plays a significant role in the speed of the arm motor.

Adding a Limit Switch

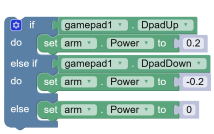
Another consideration to make is the physical limitations of your arm mechanism. Certain mechanisms may have a physical limitation, that when exceeded runs the risk of damaging the mechanism or another component of the robot. There are a few ways to limit the mechanism with sensors that will help reduce the potential of a mechanism exceeding its physical limitations. In this section we will focus on using a limit switch to limit the motion range of the arm.

> This section assumes that you have a basic knowledge of limit switches form the Test Bed section and the Digital Sensors article.

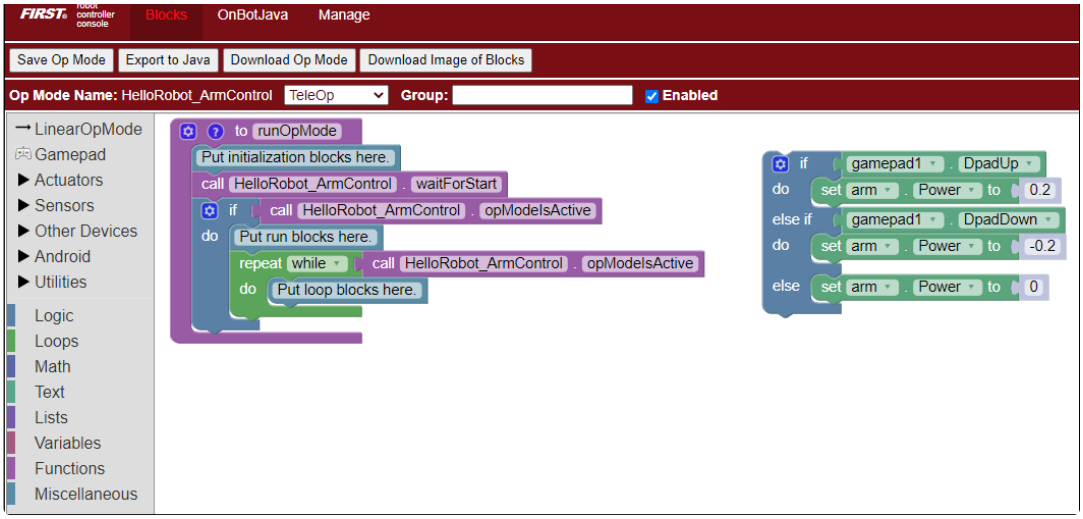
As you may recall from the Test Bed section limit switches use Boolean logic to dictate when a limit has been met. Limit switches typically come in the form of digital sensors, like the Touch Sensor, as digital sensors report a **Boolean** on/off to the system, much like a light switch.

If you are using a Class Bot your robot should have a Touch Sensor mounted to the front of your robot chassis. You also have a Limit Switch Bumper installed. Together these items create a limit switch system. By utilizing the limit switch system you can keep your Class Bot arm from exceeding the lower physical limit, or what will be known as our starting position. Lets go ahead and start coding!

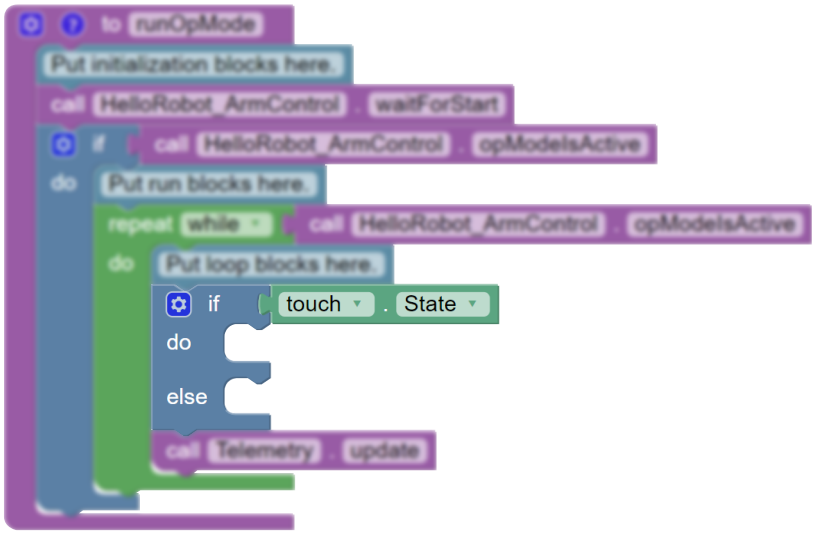
> Before proceeding with code please make sure that your mechanism is interfacing with, and pressing the Touch Sensor. If you have the Class Bot this entails making sure your bumper is actively pressing the Touch Sensor when the arm comes down.

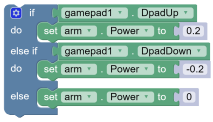
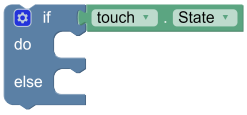
Start by grabbing the 

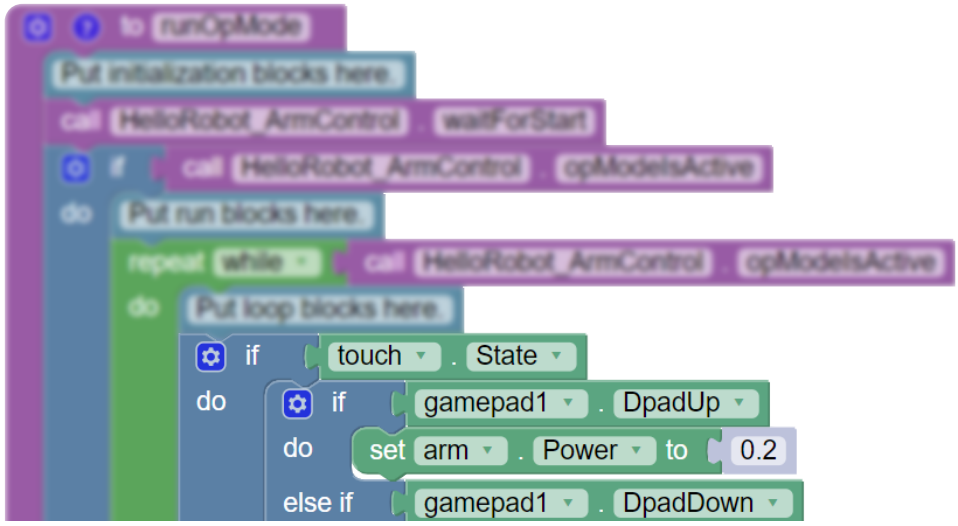
statement made in the previous section, and dragging it to the side of the blocks project.

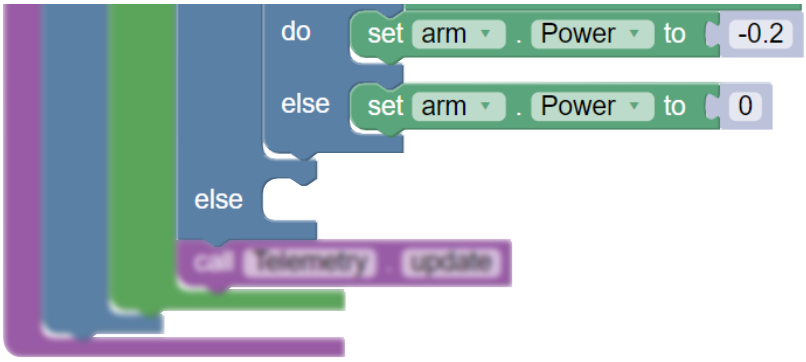


Add a new  block to the while loop. Add the  block to the conditional portion of the  block.



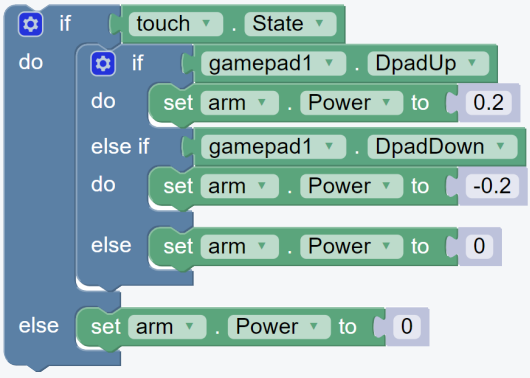
Add the  block set back to the code in the do port of the  block.





If you recall from the initial Limit switch section, the touch sensor operates on a `FALSE/TRUE` binary. When the touch sensors is not pressed the touch . State block reports true; when the touch sensor is pressed the touch . State block reports false. At this point the logic of the code states that when touch sensor is not pressed, the gamepad commands that were previously chosen operate normally. To function as a limit switch the motor needs to stop when the touch sensor is pressed.

Try adding a set arm . Power to 0 block to the else portion of the block.



Save the op mode and run it.

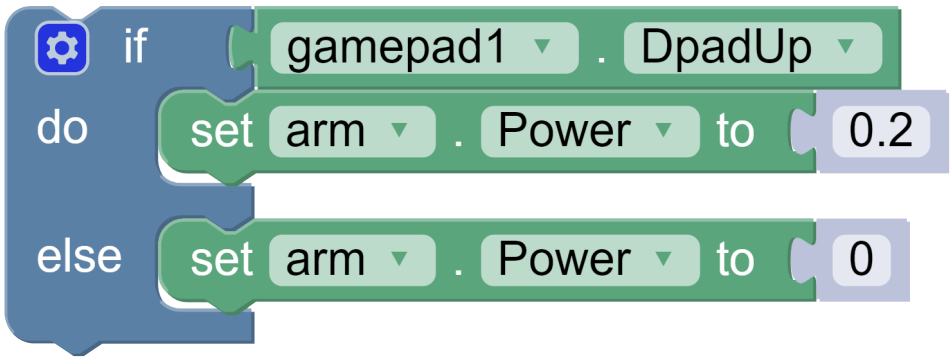
What happens when the Touch Sensor is pressed?

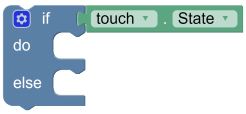
One of the common features of a limit switch, like the Touch Sensor, is the ability to reset to its default state. If you press the Touch Sensor with your finger, you may notice that as soon as you release the pressure you are applying the Touch Sensor will return to its default "not pressed" state. However, you have to release the pressure in order to accomplish this.

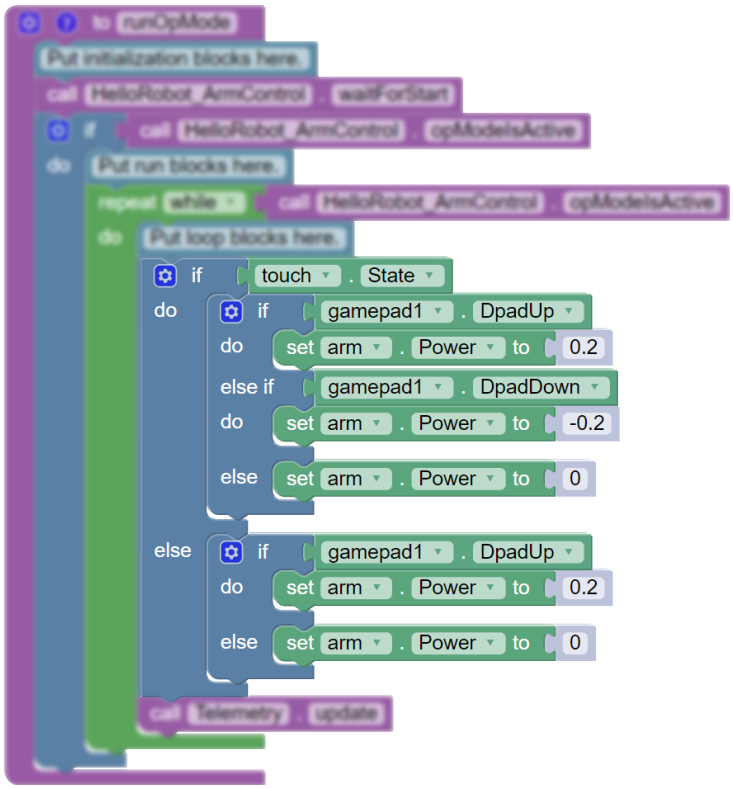
Make sure that the mechanism is actually interfacing with the Touch Sensor. For the Class Bot, you may need to adjust the Touch Sensor so that the Limit Switch Bumper is interfacing with it more consistently.

The code in the info block above dictates that when the Touch Sensor is pressed the arm motor is set to zero. This would work in a mechanism where the Touch Sensor is allowed to return to its default state on its own. However, once the arm presses the Touch Sensor, the weight of the mechanism will keep the Touch Sensor from returning to its default state. The combination of the weight of the mechanism and the logic of the info block code means that once the arm meets its limit it will not be able to move again.

To remedy this, an action to move the arm in the opposite direction of of the limit needs to be added to the else statement. To do this lets use another  block. Since the Touch Sensor is a lower limit for the arm, the arm will need to move up (or the motor in the forward direction) to move away from the touch sensor. Following the earlier convention for moving the arm, add the  as the condition in the  . In the do portion of the block add the  block. Change the duty cycle from 100% to 20%. In the else portion add the  block.



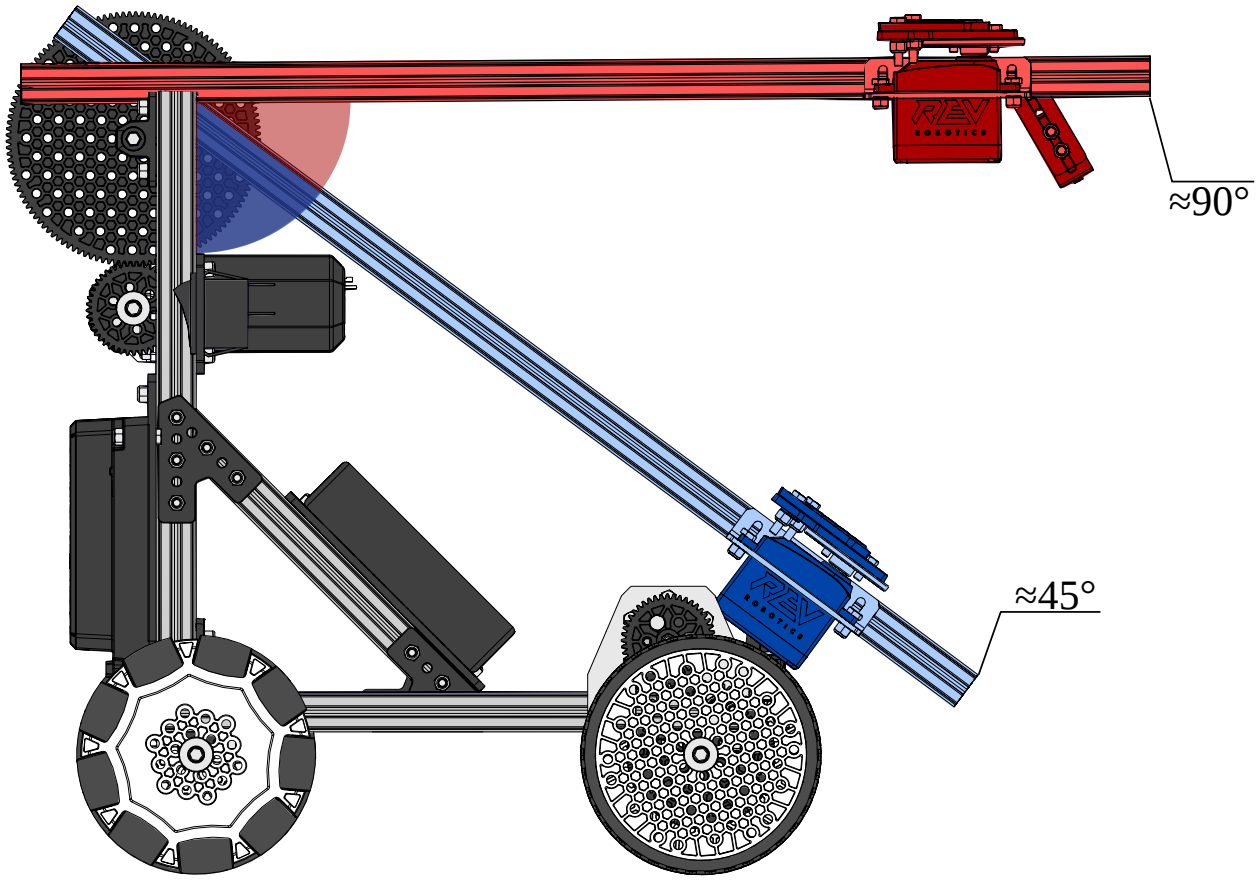
Add this block set to the else portion of the  block set.



Programming an Arm to a Position

In the Encoder Navigation section the concept of moving the motor to a specific position based on encoder

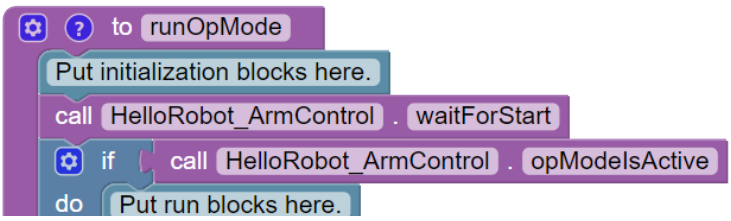
ticks was introduced. The process highlighted in *Encoder Navigation* focused on how to convert from encoder ticks to rotations to a linear distance. A similar procedure can be utilized to move the arm to a particular position. However, unlike the drivetrain, the arm does not follow a linear path. Rather than convert to a linear distance it makes more sense to convert the encoder ticks into an angle measured in degrees. In the image below two potential positions are showcased for the ClassBot arm. One of the positions - highlighted in blue below - is the position where the arm meets the limit of the touch sensor. Due to the limit, this position will be our default or starting position. From the Class Bot build guide, it is known that the Extrusion supporting the battery sits a 45 degree angle. Since the arm is roughly parallel to these extrusion when it is in the starting position, we can estimate that the default angle of the arm is roughly 45 degrees.

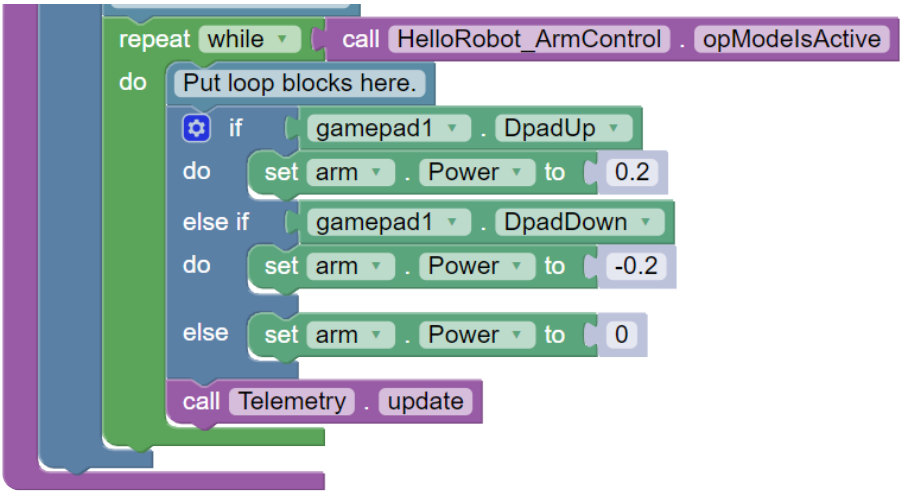


The goal of this section is to determine the amount of encoder ticks it will take to move the arm from its starting position to a position around 90 degrees. There are a few different ways this can be accomplished. An estimation can be done by moving the arm to the desired position and recording the telemetry feedback from the Driver Station. Another option is to do to the math calculations to find the amount of encoder ticks occur per degree moved. Follow through this section to walk through both options and determine which is the best for your team.

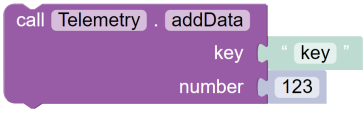
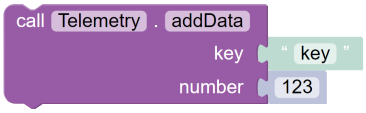
Estimating the Position of the Arm

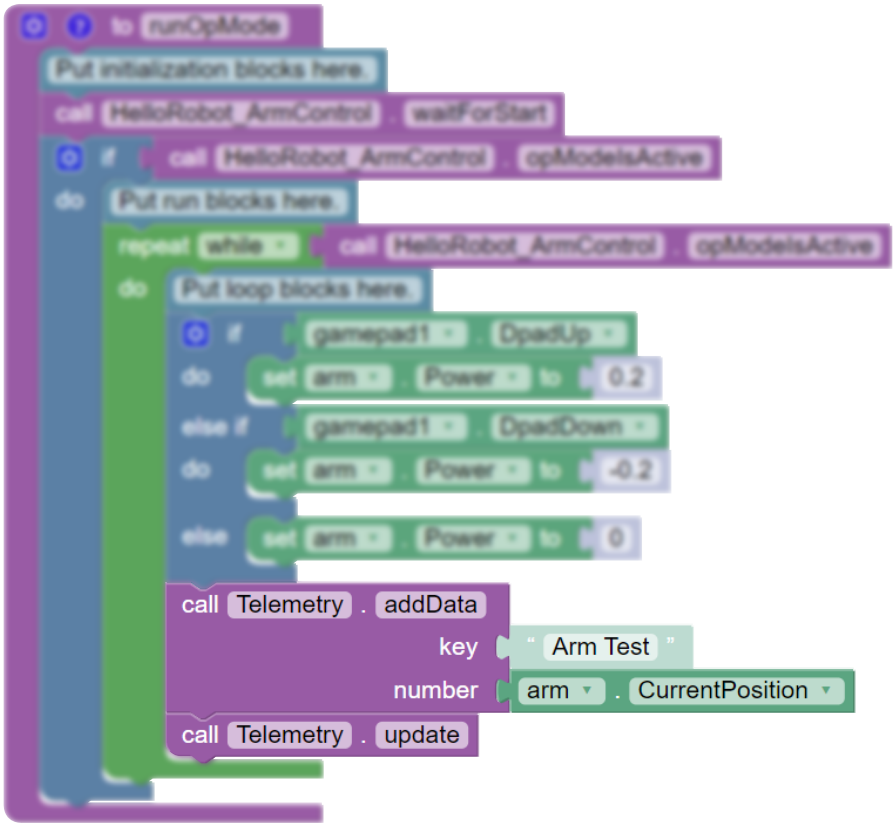
To estimate the position of the arm using telemetry and testing, lets start with the initial code we created at the start of the Basics of Programming an Arm, section.





> For now you can move the limit switch related blocks to the side of your project.

Within the loop add a  block. Add the  to the number portion of the  block. Change the key string from the default `"key"` to `"Arm Test."`



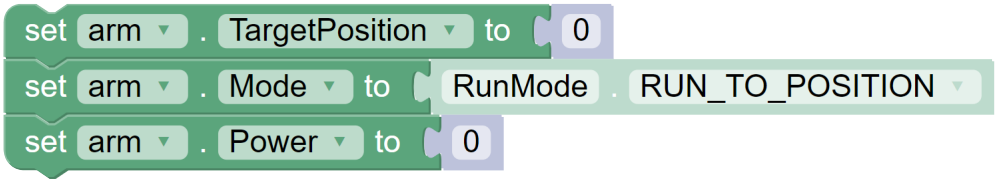
Save the op mode and run it. Use the gamepad commands to move the arm to the 90 degree position. Once you have the arm properly positioned read the telemetry off the Driver Station to determine the encoder count relative to the position of the arm.

> Recall from the Basic Encoder Concepts section that the encoder position is set to 0 each time the Control Hub is turned on. This means that if your arm is in a position other than the starting position when the Control Hub is turned on, that position becomes zero instead of the starting position.
>
> The number given in the image above is not necessarily an accurate encoder count for the 90 degree position. To get the most accurate encoder reading for your robot make sure that your starting position reads as 0 encoder counts. To further increase accuracy consider doing several testing runs before deciding on the number of counts.

To add the `RUN_TO_POSITION` code the `if/else` statement must first be edited back into an 

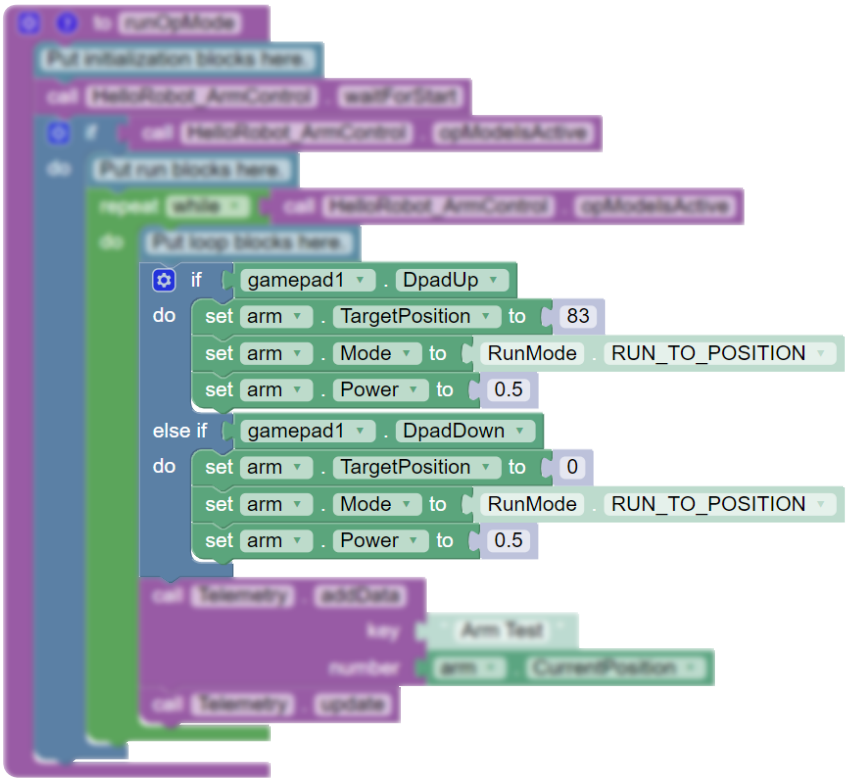
block, as shown in the code below.

Now recall that in order to execute `RUN_TO_POSITION` the following three blocks need to be added to both sections of the  block.



When `DpadUp` is pressed, the arm should move to the the 90 degree position. When `DpadDown` is pressed the arm should move back to the starting position. To do this set the first  equal to the number of ticks it took your arm to get to 90 degrees, for this example we will use 83 ticks.

Since we want `DpadDown` to return the arm to the starting position, keeping the  block set to 0 will allow us to accomplish this. Set both  blocks equal to 0.5.



> Recall that the target position dictates which direction the motor moves, taking over the directionality control from the  block, so both blocks can be set to a positive value since they will control the speed.

If you try running this code you may notice that the arm oscillates around the 90 degree position. When this behavior is present you should also notice the telemetry output for the encoder counts fluctuating. `RUN_TO_POSITION` is a **Closed Loop Control**, which means that if the arm does not perfectly reach the target position, the motor will continue to fluctuate until it does. When motors continue to oscillate and never quite reach the target position this may be a sign that the factors determining tolerances and other aspects of the closed loop are not tuned to this particular motor or mechanism. There are ways to tune the motor, but for now we want to focus on working with the arm and expanding on how limits and positions work with regards to the mechanism.

Calculating Target Position

In the initial introduction to run to position, you worked through the calculations needed to convert the ticks per rotation of a motor into ticks per mm moved. Now we want to focus on how to convert ticks per rotation of the motor to ticks per degree moved. From the previous section you should have a rough estimate of the amount of ticks you need to get to the 90 degree position. The goal of this section is to work through how to get a more exact position.

To start you will need some of the same variables we used in Encoder Navigation:

Ticks per Revolution

Recall, that ticks per revolution of the encoder shaft is different than the ticks per revolution of the shaft that is controlling a mechanism. We saw this in the Encoder Navigation section when the ticks per revolution at the motor was different from the ticks per revolution of the wheel. As motion is transmitted from a motor to a mechanism, the resolution of the encoder ticks changes.

> For more information on the effect of motion transmission across a mechanism check out the Compound Gearing section.

The amount of ticks per revolution of the encoder shaft is dependent on the motor and encoder. Manufacturers of motors with built-in encoders will have information on the amount of ticks per revolution.

> Visit the manufacturers website for your motor or encoders for more information on encoder counts. For HD Hex Motors or Core Hex Motors visit the Motor documentation.

In the Core Hex Motor specifications there are two different Encoder Counts per Revolution numbers:

- At the motor - 4 counts/revolution
- At the output - 288 counts/revolution

At the motor is the number of encoder counts on the shaft that encoder is on. This number is equivalent to the 28 counts per revolution we used for the HD Hex Motor. The 288 counts "at the output" accounts for the change in resolution after the motion is transmitted from the motor to the built in 72:1 gearbox. Lets use the 288 as ticks per revolution so that we do not have to account for the gearbox in our total gear reduction variable.

Total Gear Reduction

Since we built the the gear reduction from the motor gearbox into the ticks per revolution the main focus of this section is calculating the gear reduction of the arm joint. The motor shaft drives a 45 tooth gear that transmits motion to a 125 tooth gear. The total gear ratio is 125T:45T. To calculate the gear reduction for this gear train, we can simply divide 125 by 45.

$$\frac{125}{45} = 2.777778$$

To summarize, for the Class Bot V2 the following information is true:

Ticks per revolution	288 ticks
Total gear reduction	2.777778

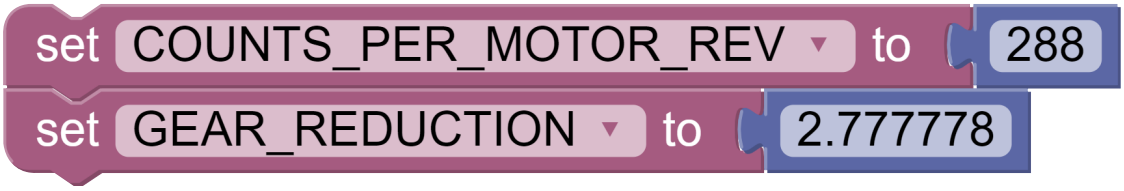
Now that we have this information lets create two constant variables:

- `COUNTS_PER_MOTOR_REV`
- `GEAR_REDUCTION`

> ⓘ The common naming convention for constant variables is known as CONSTANT_CASE, where the variable name is in all caps and words are separated by and underscore.

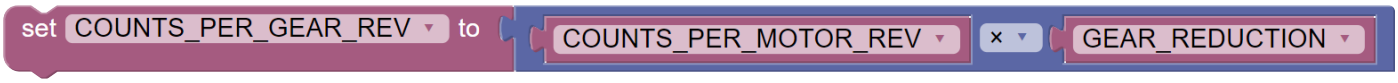
Add the variables `COUNTS_PER_MOTOR_REV` and `GEAR_REDUCTION` variables to the initialization section of the op mode.

Once the variables are created and added to the op mode, use the 0 blocks to set the variables to the respective values

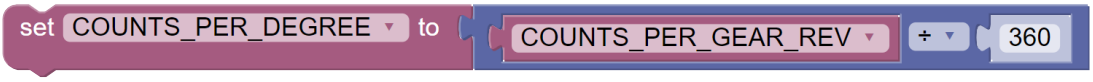


Now that these two variables have been defined, we can use them to calculate two other variables: the amount of encoder counts per rotation of the 125T driven gear and the number of counts per degree moved.

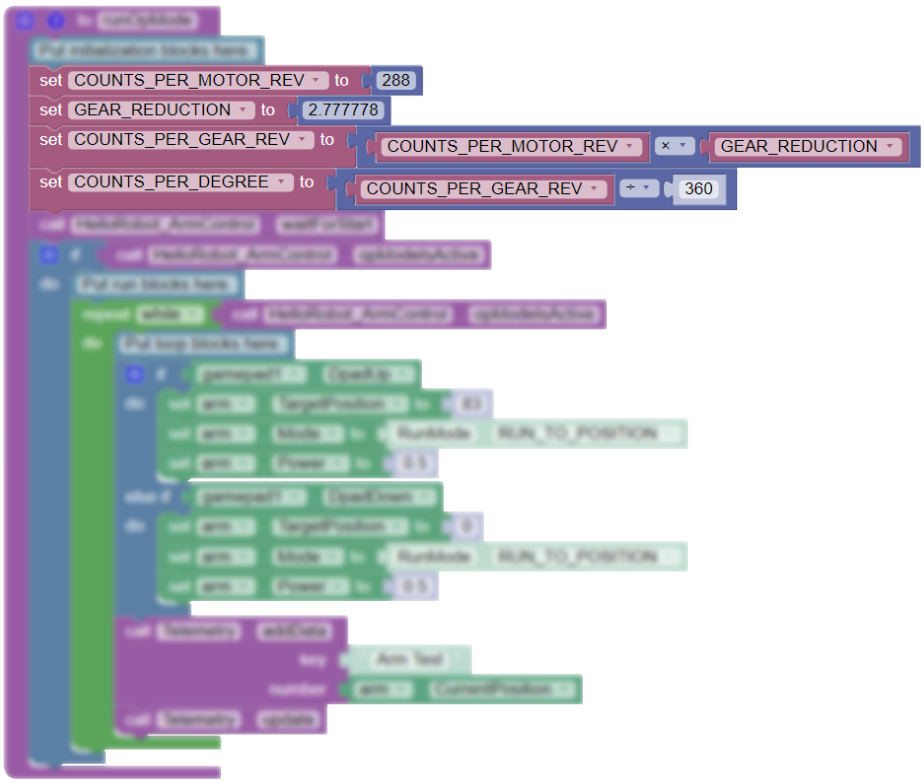
Calculating counts per revolution of the 125T gear (or `COUNTS_PER_GEAR_REV`)is the same formula used in Encoder Navigation for our `COUNTS_PER_WHEEL_REV` variable. So to get this variable we can multiple `COUNTS_PER_MOTOR_REV` by `GEAR_REDUCTION`.



To calculate the number of counts per degree or moved or `COUNTS_PER_DEGREE` divide the `COUNTS_PER_GEAR_REV` variable by 360.



Add both these variables to the op mode in the initialization section of the op mode.



Finally we need to create a non-constant variable that will act as our position. Create a variable called arm position.

To get to the 90 degree position, the arm needs to move roughly 45 degrees. Set arm position equal to `COUNTS_PER_DEGREE` times 45.

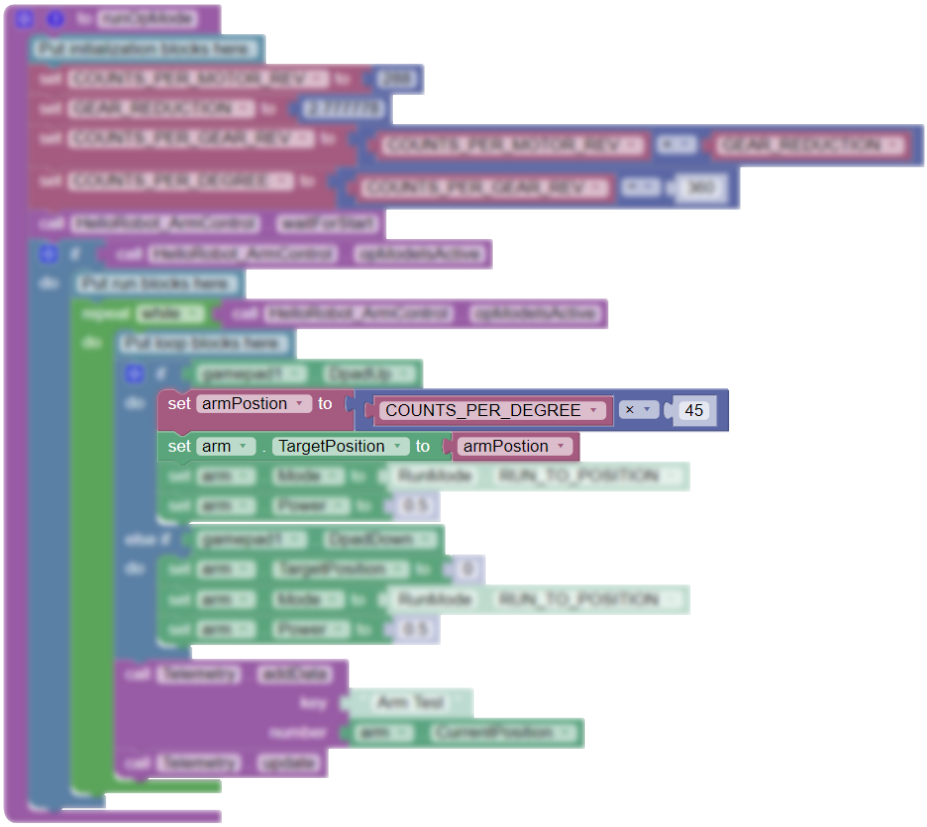
Add this variable to the 

section of the 

statement, as this section dictates the 90 degree position. Add the 

block to the 

block.



> We could set 

equal to 

. However, it is a good practice to create a variable in situations like this. If we want to add another position later, we can easily edit the variable to fit our needs.

Using Limits to Control Range of Motion

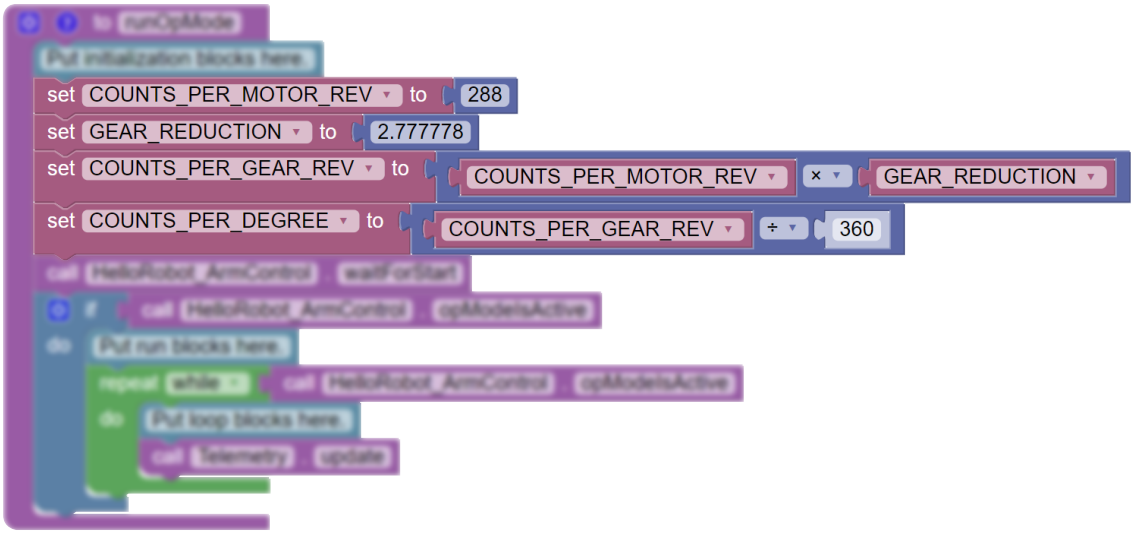
In the previous sections you worked on some of the building blocks for restricting an arms range of motion. From those sections you should have the foundation you need to perform basic arm control. However, there

are some other creative ways you can use encoder positions and limits to expand the control you have over

This section will cover two additional types of control. The first type of control we will explore is the idea of **soft limits**. In the Adding a Limit Switch section we discuss the concept of physical limits of a mechanism however, there may be times you need to limit the range of motion of an arm without installing a physical limit. To do this a position based code can be used to create a range for the arm.

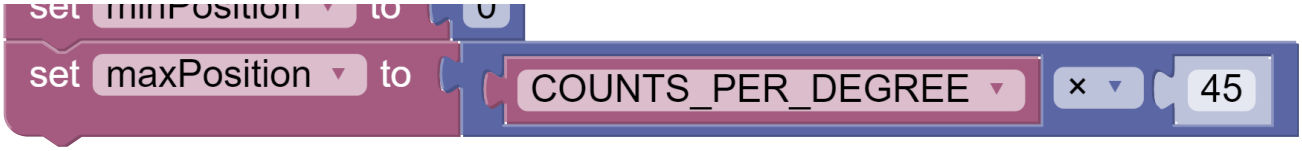
Once you have a basic idea of how to create soft limits, we will explore how to use a limit switch (like a touch sensor) to reset the range of motion. This type of control reduces the risk of getting stuck outside of your intended range of motion, which can affect the expected behavior of your robot.

To set the soft limits we will use some of the basic logic we established in previous sections, with some edited changes. Start with a `Basic Op Mode` and add the constant variables from the Calculating Target Position section to the op mode.

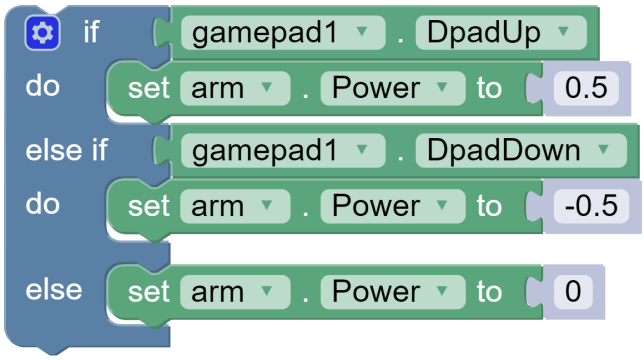


Next we need to create our upper and lower limits. Create two new variables one called `minPosition` and one called `maxPosition`. Add both of these to the initialization section of the op mode.

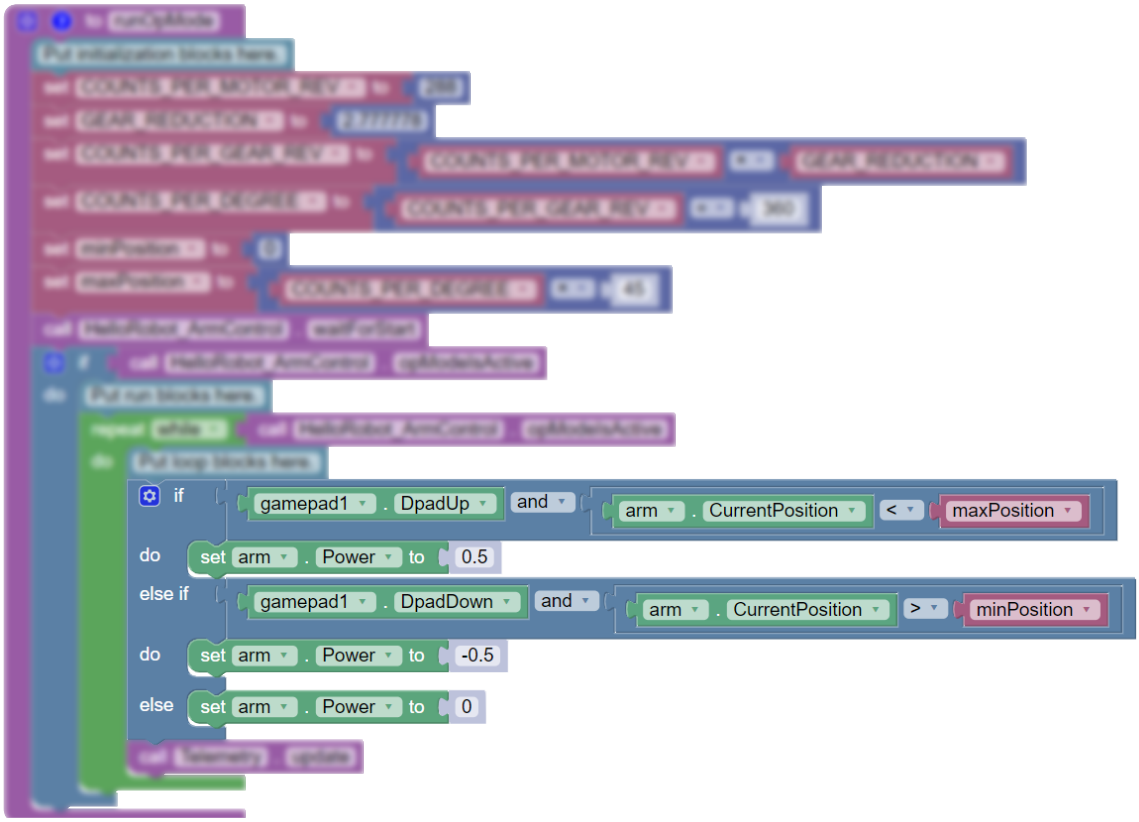
For now we want the `minPosition` set as our starting position and the `maxPosition` set to our 90 degree position. Set `minPosition` equal to  and set `maxPosition` equal to  .



An `if/else if` statement needs to be added to control the arm, for this we can use the same basic logic we use in the Basics of Programming an Arm section.



To set the limit we need to edit our `if/else if` statement so that the limits are built in. If `DpadUp` is selected and the position of the arm is less than the `maxPosition` then the arm will move to the `maxPosition`. If `DpadDown` is selected and the position of the is greater that the `minPosition` then the arm will move towards the `minPosition`.



The current code configuration will stop the motor at any point that the conditions to power the motor are not met. Depending to factors like the weight of the mechanism and any load that it is bearing, when the motor stops the arm may drop below the `maxPosition`. Take time to test out the code and confirm that it behaves in the way you expect it to.

Overriding Limits

One of the benefits of having a soft limit is being able to exceed that limit. Since encoders zero tick position is determined by the position of the arm when the Control Hub powers on; if attention is not payed to what position the arm is on power up the range of motion of the arm is affected. For instance, if we have to reset the Control Hub while the arm is in the 90 degree position, the 90 degree position is equal to 0 encoder ticks. One way around this is to create an override for the range of motion.

There are a few different ways an override of sorts can be created, in our case we are going to use the a button and touch sensor to help reset our range.

Start by editing the 

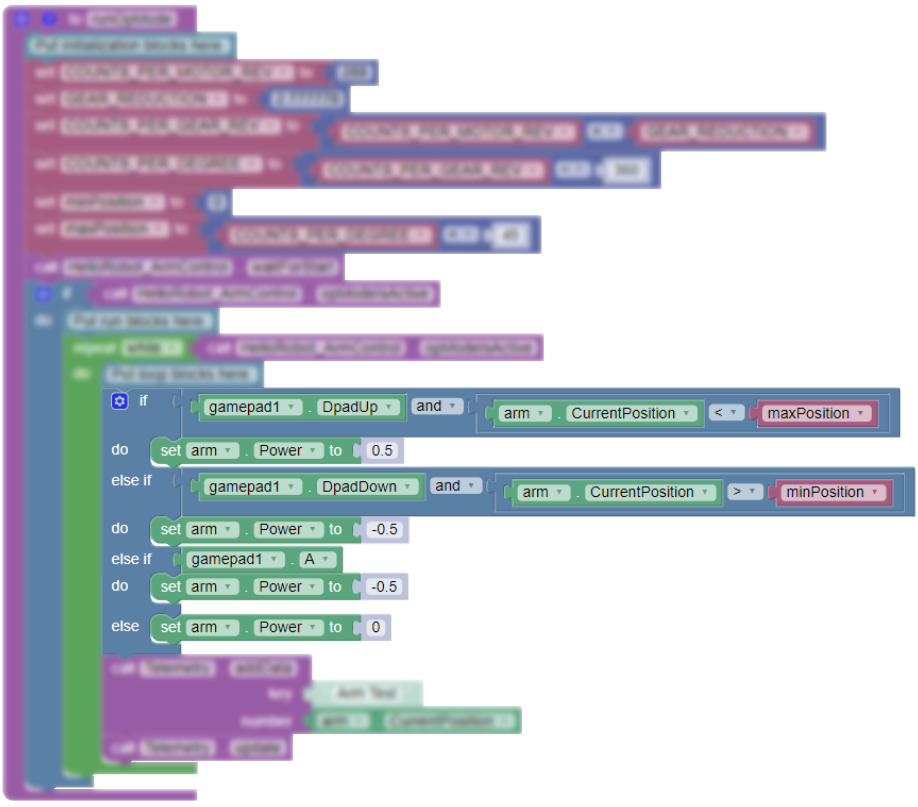
to add another 

condition. Use the 

block as the condition. Add a 

block to the do portion of the 

block.



Now that we have this change in place, when the `a` button is pressed the arm will move toward the starting position. When the arm reaches and presses the touch sensor we want to `STOP_AND_RESET_ENCODER` .

We can create an 

statement that focuses on performing this stop and reset when the touch sensor is pressed. Since the touch sensor reports `true` when its not pressed and `false` when it is, we will need to use the 

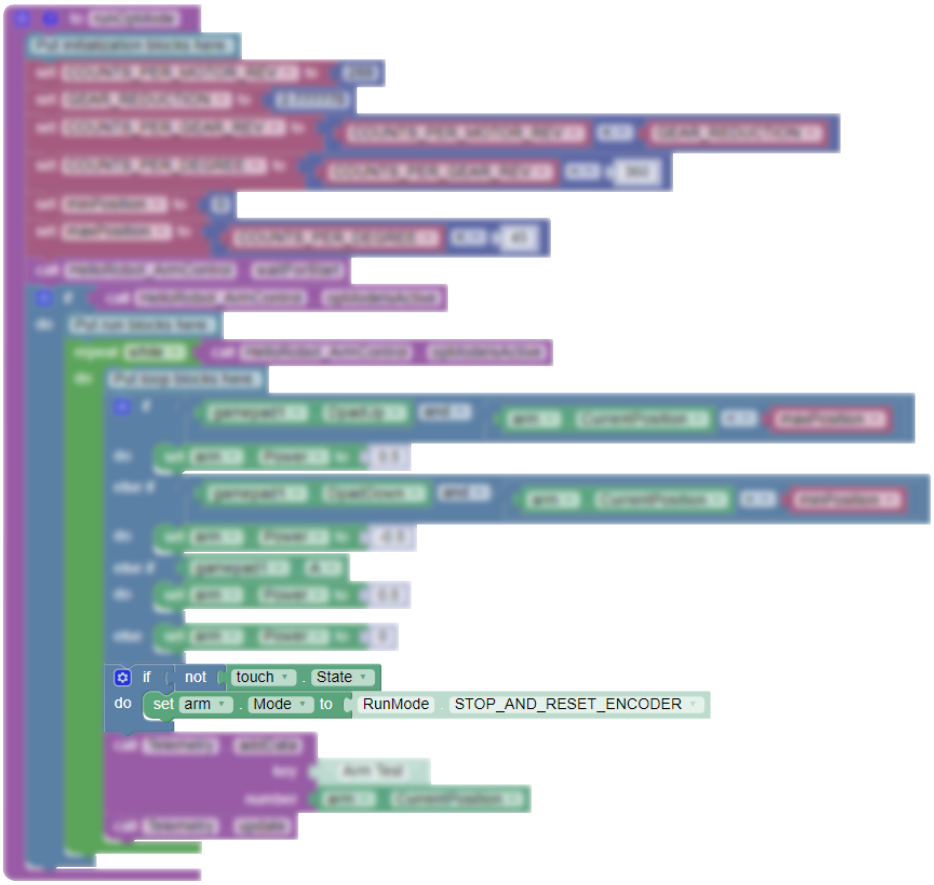
block.

> The **not operator** 

can be used in conditional binary statements when you need inverse

> whether something is `true` of `false`. For instance, an `if` statement activates when something is true, but when the Touch Sensor reports `true` it is not pressed. In our case we want this if statement to activate when the touch sensor is pressed thus we need to use the not operator.

So, if the touch sensor returns `false` (or is pressed) the motor run mode `STOP_AND_RESET_ENCODER` will be activated causing the motor encoder to reset to 0 ticks.

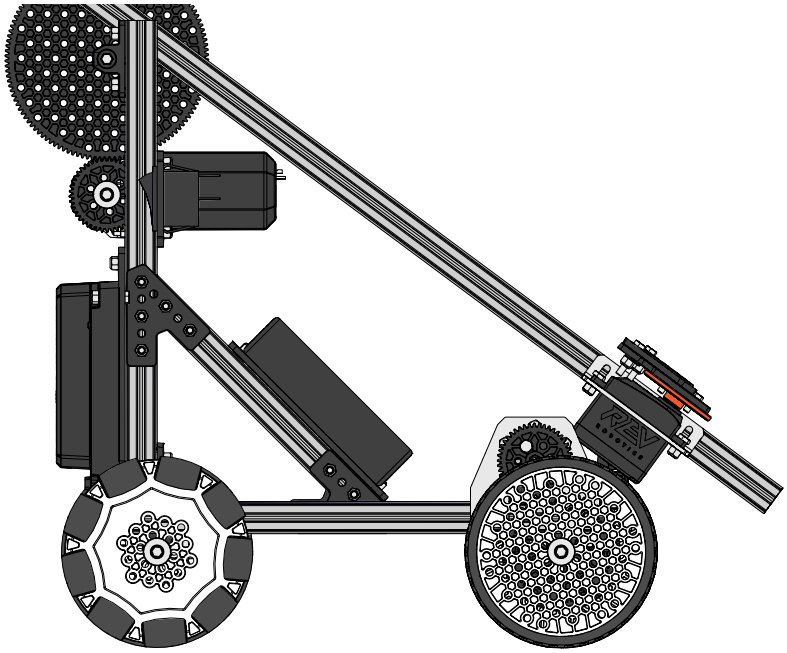


Now that this code is done, try testing it!

Arm Control - OnBot Java

Introduction to Arm Control

Robot control comes in many different forms. Now that you have walked through programming a drivetrain, we can apply those concepts to controlling other mechanisms. Since this guide utilizes the Class Bot the focus will be on the basics of controlling it's main mechanism, a single jointed arm.



Controlling an arm requires a different thought process than the one you used to control the drivetrain. While the drivetrain uses the rotation motion of the motors to drive along a linear distance, an arm rotates along a central point, or joint. When working with an arm you will have to head caution to the physical limitations of the robot this includes load bearing, range of motion, and other forces that may apply.

In this section you will learn how to use the gamepad `Dpad` controls and the installed Touch Sensor to control the arm. However, the focus of this section is using code to limit the range of motion of the arm.

Sections	Goals of Section
Basics of Programming an Arm	Introduction to coding an arm for teleoperated control and working with a limit switch
Programming an Arm to a Position	Using motor encoders to move an arm to a specifi position, such as from 45 degrees to 90 degrees.
Using Limits to Control Range of Motion	Working with the basics of arm control, motor encoder, and limit switches to control the range of motion for an arm.

Basics of Programming an Arm

Start by creating a basic op mode called `HelloRobot_ArmControl`.

> For more information on how to create an op mode check out the Test Bed - Onbot Java section.

Unlike the joystick, which sends values corresponding to the position of the joystick, the `Dpad` on the gamepad inputs **Boolean** `FALSE/TRUE`. In order to dictate how the arm moves when you press `DpadUp`

or `DpadDown`; an `if/else if` statement needs to be used. Create an `if/else if` statement like the one below.

```
while (opModeIsActive()) {
   if(gamepad1.dpad_up){

               }
   else if (gamepad1.dpad_down){

               }
    }
```

Now that the basic structure is in place, we can add the necessary blocks to dictate the direction of the arm. The best practice is to have the arm move up when `DpadUp` is selected and down with `DpadDown` is selected. To do this lets add `arm.SetPower();` to each of the actionable parts of the `if/else if` statement.

> Recall that the value assigned to `setPower` dictates the direction and speed of the motor. Between the motor and the gearing on the class bot the positive value will move the arm the arm upwards and a negative value will move the arm downwards.
>
> If you are unsure which direction your motor will move create the following code and test to ensure that your motor is behaving as expected.

```
if(gamepad1.dpad_up){
       arm.setPower(0.2);
            }
else if (gamepad1.dpad_down){
       arm.setPower(-0.2);
            }
```

> Starting with a lower duty cycle percentage such as the 0.2 exhibited in the code above, will allow for easier testing when making decisions for the arm. We will change to a higher duty cycle later on in this guide.

> Save the op mode and try running the code. Consider the following questions.
>
> - What happens if you press up on the `Dpad` ?
> - What happens if you press down on the `Dpad` ?

Right now the logic of the `if/else if` statement declares that when `gamepad1.dpad_up` is true (has been pressed) the motor will run in the forward (or in this case upwards) direction at 20% duty cycle. If `gamepad1.dpad_down` is true the motor will run in reverse at 20% duty cycle. If you ran the code at this stage you may have noticed that even when you released the `Dpad` the motor continued to run in the

selected direction. The current `if/else if` statement tells the robot when the motor should move and in

To fix this edit the `if/else if` statement to include and action to perform if neither gamepad conditions are true. Since we want the arm to stop moving if neither gamepad conditions are met lets use `arm.setPower(0);` to stop the motor .

```
if(gamepad1.dpad_up){
       arm.setPower(0.2);
            }
else if (gamepad1.dpad_down){
       arm.setPower(-0.2);
            }
else {
       arm.setPower(0);
            }
```

> Try saving and running the op mode again. Pay attention to the speed of the arm going up versus going down. Does the speed seem the same?

Working with an arm introduces different factors for consideration than what you have seen previously with drivetrains. For instance, did you notice a difference in speeds when moving the arm up or down? Unlike the drivetrain, where the effect of gravity impacts the motors consistently across either direction, gravity plays a significant role in the speed of the arm motor.

Adding a Limit Switch

Another consideration to make is the physical limitations of your arm mechanism. Certain mechanisms may have a physical limitation, that when exceeded runs the risk of damaging the mechanism or another component of the robot. There are a few ways to limit the mechanism with sensors that will help reduce the potential of a mechanism exceeding its physical limitations. In this section we will focus on using a limit switch to limit the motion range of the arm.

> This section assumes that you have a basic knowledge of limit switches form the Test Bed section and the Digital Sensors article.

As you may recall from the Test Bed section limit switches use Boolean logic to dictate when a limit has been met. Limit switches typically come in the form of digital sensors, like the Touch Sensor, as digital sensors report a **Boolean** on/off to the system, much like a light switch.

If you are using a Class Bot your robot should have a Touch Sensor mounted to the front of your robot chassis. You also have a Limit Switch Bumper installed. Together these items create a limit switch system. By utilizing the limit switch system you can keep your Class Bot arm from exceeding the lower physical limit, or what will be known as our starting position. Lets go ahead and start coding!

> Before proceeding with code please make sure that your mechanism is interfacing with, and

> pressing the Touch Sensor. If you have the Class Bot this entails making sure your bumper is actively pressing the Touch Sensor when the arm comes down.

In the Test Bed - Onbot Java section, you learned how to create a basic limit switch program, similar to the one below.

Limit Switch if/else

```
 if (touch.getState()){
    //Touch Sensor is not pressed
     arm.setPower(0.2);

} else {
    //Touch Sensor is pressed
    arm.setPower(0);
                        }
```

If you recall from the initial Limit Switch section, the Touch Sensor operates on a `FALSE/TRUE` binary. When the touch sensors is not pressed `touch.getState()` reports `true`; when the touch sensor is pressed `touch.getState()` reports `false`. The logic of the code states that when touch sensor is not pressed, the motor runs at 20% duty cycle.

Rather than have the motor run at 20% of duty cycle when the Touch Sensor isn't pressed and stop when the sensor is pressed, we want to control the arm using the gamepad still. To do this we can nest the `Gamepad if/else if` statement within the `Limit Switch if/else` statement.

> For this next portion we will be utilizing the `if/else if` statement create in the Basics of Programming and Arm. From here on out this basic code logic will be refereed to as the `Gamepad if/else if`. The limit switch code will be know as the `Limit Switch if/else`. Both pieces of code will be referenced again.
>
> Gamepad if/else if
>
> ```
> if(gamepad1.dpad_up){
>        arm.setPower(0.2);
>             }
> else if (gamepad1.dpad_down){
>        arm.setPower(-0.2);
>
>             }
> else {
>        arm.setPower(0);
>             }
> ```

```
if(touch.getState()){
      if(gamepad1.dpad_up){
            arm.setPower(0.2);
                 }
      else if (gamepad1.dpad_down){
```

```
            arm.setPower(-0.2);
                  }
      else {
            arm.setPower(0);
                  }
       }
  else {
      arm.setPower(0);
       }
```

> Save the op mode and run it.
>
> What happens when the Touch Sensor is pressed?

One of the common features of a limit switch, like the touch sensor, is the ability to reset to its default state. If you press the Touch Sensor with your finger, you may notice that as soon as you release the pressure you are applying the Touch Sensor will return to its default "not pressed" state. However, you have to release the pressure in order to accomplish this.

> Make sure that the mechanism is actually interfacing with the Touch Sensor. For the Class Bot, you may need to adjust the Touch Sensor so that the Limit Switch Bumper is interfacing with it more consistently.

The code in the info block above dictates that when the Touch Sensor is pressed the arm motor is set to zero. This would work in a mechanism where the Touch Sensor is allowed to return to its default state on its own. However, once the arm presses the Touch Sensor, the weight of the mechanism will keep the Touch Sensor from returning to its default state. The combination of the weight of the mechanism and the logic of the info block code means that once the arm meets its limit it will not be able to move again.

To remedy this, an action to move the arm in the opposite direction of of the limit needs to be added to the `else` statement. Since the Touch Sensor is a lower limit for the arm, the arm will need to move up (or the motor in the forward direction) to move away from the touch sensor. To do this we can create an `if/else` statement similar to our gamepad `Gamepad if/else if` statement. Instead of having the normal gamepad operations, when the Touch Sensor and `DpadUp` are pressed the arm moves away from the Touch Sensor. Once the Touch Sensor no longer reports false the normal gamepad operations return and the arm can move in either direction again.

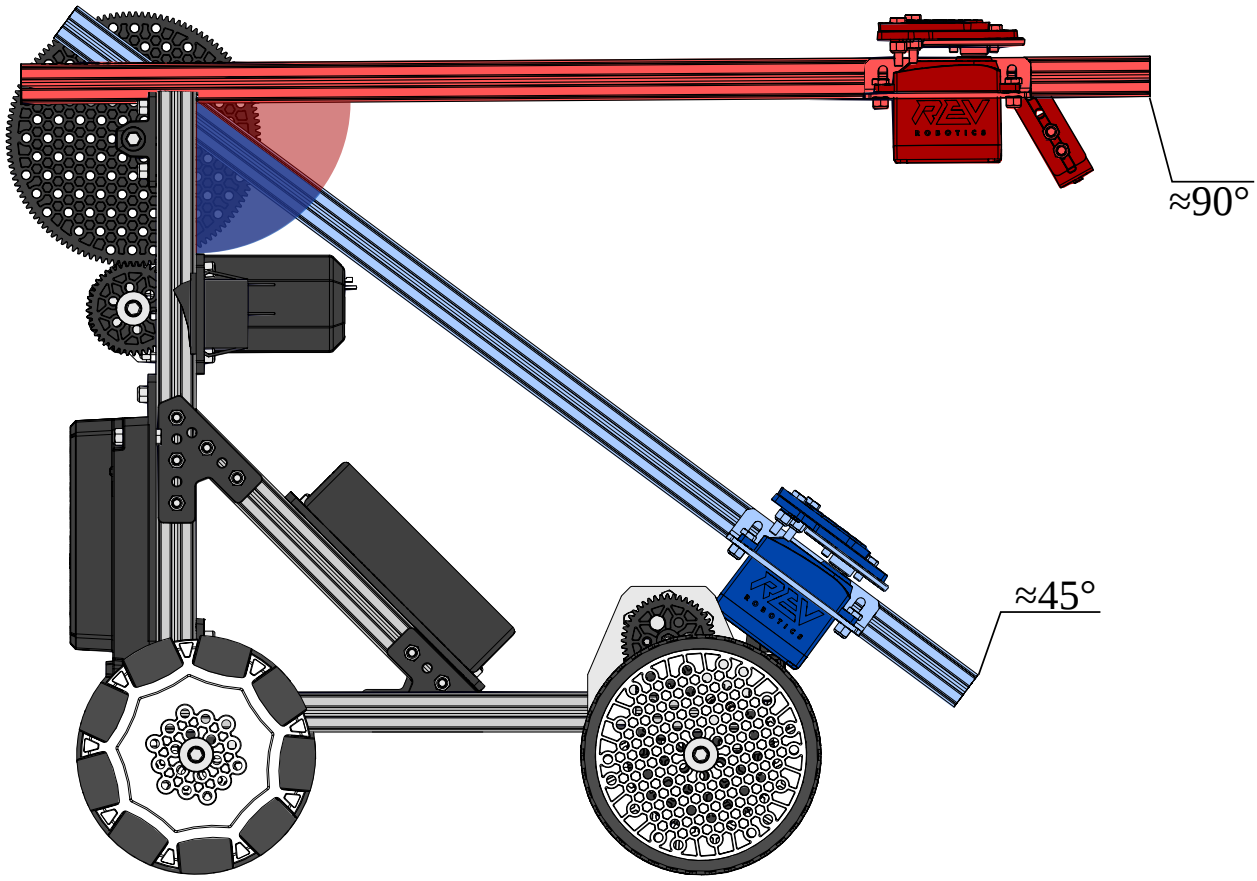
```
if(touch.getState()){
      if(gamepad1.dpad_up){
            arm.setPower(0.2);
                  }
      else if (gamepad1.dpad_down){
            arm.setPower(-0.2);
                  }
      else {
            arm.setPower(0);
```

```
        }          }
    else {
        if(gamepad1.dpad_up){
            arm.setPower(0.2);
                }
        else{
            arm.setPower(0);
        }
```

Programming an Arm to a Position

In the Encoder Navigation section the concept of moving the motor to a specific position based on encoder ticks was introduced. The process highlighted in *Encoder Navigation* focused on how to convert from encoder ticks to rotations to a linear distance. A similar procedure can be utilized to move the arm to a particular position. However, unlike the drivetrain, the arm does not follow a linear path. Rather than convert to a linear distance it makes more sense to convert the encoder ticks into an angle measured in degrees.

In the image below two potential positions are showcased for the ClassBot arm. One of the positions - highlighted in blue below - is the position where the arm meets the limit of the touch sensor. Due to the limit, this position will be our default or starting position. From the Class Bot build guide, it is known that the Extrusion supporting the battery sits a 45 degree angle. Since the arm is roughly parallel to these extrusion when it is in the starting position, we can estimate that the default angle of the arm is roughly 45 degrees.



The goal of this section is to determine the amount of encoder ticks it will take to move the arm from its

starting position to a position around 90 degrees. There are a few different ways this can be accomplished. An estimation can be done by moving the arm to the desired position and recording the telemetry feedback from the Driver Station. Another option is to do to the math calculations to find the amount of encoder ticks occur per degree moved. Follow through this section to walk through both options and determine which is the best for your team.

Estimating the Position of the Arm

To estimate the position of the arm using telemetry and testing, lets start with the `Gamepad if/else if` code.

Gamepad if/else if

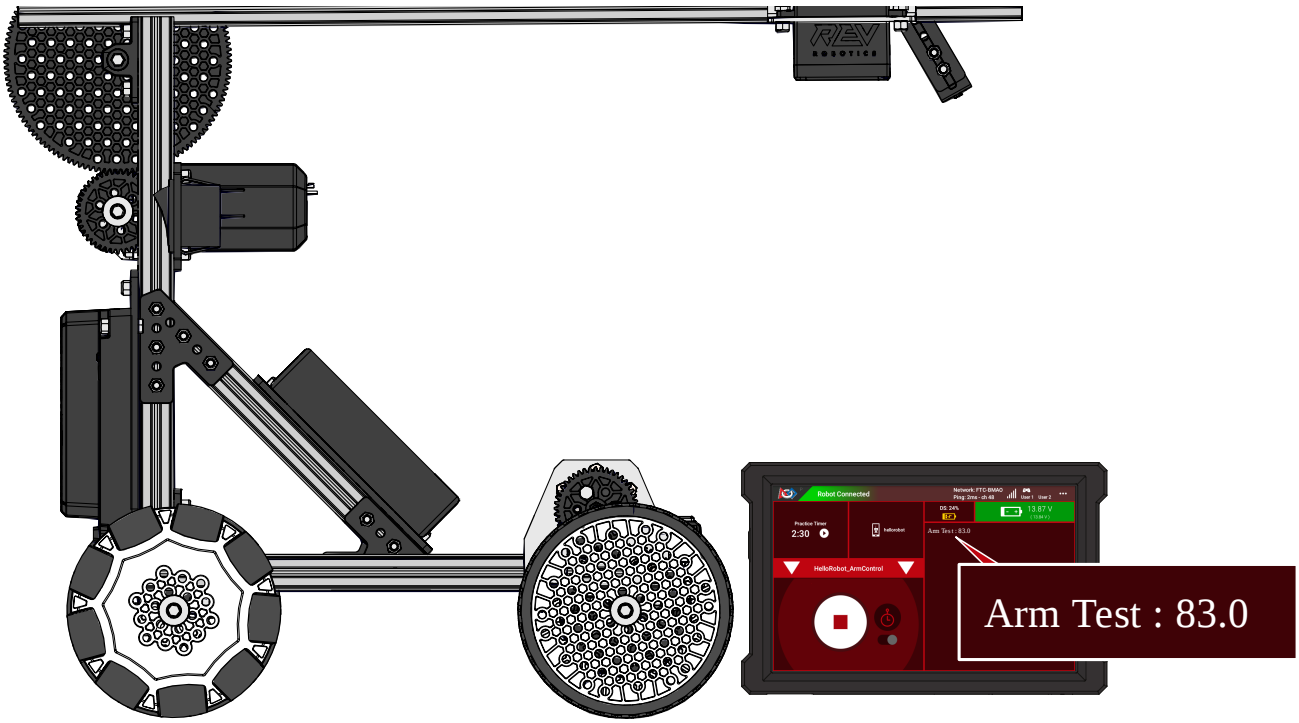
```
if(gamepad1.dpad_up){
       arm.setPower(0.2);
            }
else if (gamepad1.dpad_down){
       arm.setPower(-0.2);
            }
else {
       arm.setPower(0);
            }
```

> For now you can comment out the limit switch related code.

Within the while loop add the lines `telemetry.addData("Arm Test", arm.getCurrentPosition());` and `telemetry.update();`

```
while(opModeIsActive){
    if(gamepad1.dpad_up){
       arm.setPower(0.2);
            }
     else if (gamepad1.dpad_down){
       arm.setPower(-0.2);
            }
     else {
       arm.setPower(0);

            }
     telemetry.addData("Arm Test", arm.getCurrentPosition());
     telemetry.update();

 }
```

Save the op mode and run it. Use the gamepad commands to move the arm to the 90 degree position. Once you have the arm properly positioned read the telemetry off the Driver Station to determine the encoder count relative to the position of the arm.



> Recall from the Basic Encoder Concepts section that the encoder position is set to 0 each time the Control Hub is turned on. This means that if your arm is in a position other than the starting position when the Control Hub is turned on, that position becomes zero instead of the starting position.
>
> The number given in the image above is not necessarily an accurate encoder count for the 90 degree position. To get the most accurate encoder reading for your robot make sure that your starting position reads as 0 encoder counts. To further increase accuracy consider doing several testing runs before deciding on the number of counts.

Recall that in order to execute `RUN_TO_POSITION` the following three lines of cod need to be added to both sections of the Gamepad `if/else if` block.

```
arm.setTargetPosition(0);
arm.setMode(DcMotor.RunMode.RUN_TO_POSITION);
arm.setPower(0);
```

When `DpadUp` is pressed, the arm should move to the the 90 degree position. When `DpadDown` is pressed the arm should move back to the starting position. To do this set the first `arm.setTargetPosition(0);` equal to the number of ticks it took your arm to get to 90 degrees, for this example we will use 83 ticks.

Since we want `DpadDown` to return the arm to the starting position, keeping the `arm.setTargetPosition(0);` set to 0 will allow us to accomplish this. Set both `arm.setPower(0);` equal to 0.5.

Target Position if/else if

```
if(gamepad1.dpad_up){
```

```
      arm.setTargetPosition(90);
      arm.setMode(DcMotor.RunMode.RUN_TO_POSITION);
      arm.setPower(0.5);
            }
else if (gamepad1.dpad_down){
       arm.setTargetPosition(0);
       arm.setMode(DcMotor.RunMode.RUN_TO_POSITION);
       arm.setPower(0.5);
            }
```

> Note: the code above was given a file name `Target Position if/else if` this code will be referenced again.

> Recall that the target position dictates which direction the motor moves, taking over the directionality control from `arm.setPower();` so both blocks can be set to a positive value since they will control the speed.

If you try running this code you may notice that the arm oscillates in the 90 degree position. When this behavior is present you should also notice the telemetry output for the encoder counts fluctuating. `RUN_TO_POSITION` is a **Closed Loop Control**, which means that if the arm does not perfectly reach the target position, the motor will continue to fluctuate until it does. When motors continue to oscillate and never quite reach the target position this may be a sign that the factors determining tolerances and other aspects of the closed loop are not tuned to this particular motor or mechanism. There are ways to tune the motor, but for now we want to focus on working with the arm and expanding on how limits and positions work with regards to the mechanism.

Calculating Target Position

In the initial introduction to run to position, you worked through the calculations needed to convert the ticks per rotation of a motor into ticks per mm moved. Now we want to focus on how to convert ticks per rotation of the motor to ticks per degree moved. From the previous section you should have a rough estimate of the amount of ticks you need to get to the 90 degree position. The goal of this section is to work through how to get a more exact position.

To start you will need some of the same variables we used in Encoder Navigation:

Ticks per Revolution

Recall, that ticks per revolution of the encoder shaft is different than the ticks per revolution of the shaft that is controlling a mechanism. We saw this in the Encoder Navigation section when the ticks per revolution at the motor was different from the ticks per revolution of the wheel. As motion is transmitted from a motor to a mechanism, the resolution of the encoder ticks changes.

> For more information on the effect of motion transmission across a mechanism check out the Compound Gearing section.

The amount of ticks per revolution of the encoder shaft is dependent on the motor and encoder. Manufacturers of motors with built-in encoders will have information on the amount of ticks per revolution.

> ⓘ Visit the manufacturers website for your motor or encoders for more information on encoder counts. For HD Hex Motors or Core Hex Motors visit the Motor documentation.

In the Core Hex Motor specifications there are two different Encoder Counts per Revolution numbers:

- At the motor - 4 counts/revolution
- At the output - 288 counts/revolution

At the motor is the number of encoder counts on the shaft that encoder is on. This number is equivalent to the 28 counts per revolution we used for the HD Hex Motor. The 288 counts "at the output" accounts for the change in resolution after the motion is transmitted from the motor to the built in 72:1 gearbox. Lets use the 288 as ticks per revolution so that we do not have to account for the gearbox in our total gear reduction variable.

Total Gear Reduction

Since we built the the gear reduction from the motor gearbox into the ticks per revolution the main focus of this section is calculating the gear reduction of the arm joint. The motor shaft drives a 45 tooth gear that transmits motion to a 125 tooth gear. The total gear ratio is 125T:45T. To calculate the gear reduction for this gear train, we can simply divide 125 by 45.

$$\frac{125}{45} = 2.777778$$

To summarize, for the Class Bot V2 the following information is true:

Ticks per revolution	288 ticks
Total gear reduction	2.777778

Now that we have this information lets create two constant variables:

- `COUNTS_PER_MOTOR_REV`
- `GEAR_REDUCTION`

> ⓘ The common naming convention for constant variables is known as CONSTANT_CASE, where the variable name is in all caps and words are separated by and underscore.

Add the `COUNTS_PER_MOTOR_REV` and `GEAR_REDUCTION` variables to the op mode beneath where the hardware variables are created.

```
public class HelloRobot_ArmControl extends LinearOpMode {
    private DcMotor arm;

    static final double     COUNTS_PER_MOTOR_REV    = 288;
    static final double     GEAR_REDUCTION     = 2.7778;
```

Now that these two variables have been defined, we can use them to calculate two other variables: the amount of encoder counts per rotation of the 125T driven gear and the number of counts per degree moved.

Calculating counts per revolution of the 125T gear (or `COUNTS_PER_GEAR_REV`)is the same formula used in Encoder Navigation for our `COUNTS_PER_WHEEL_REV` variable. So to get this variable we can multiple `COUNTS_PER_MOTOR_REV` by `GEAR_REDUCTION`.

```
static final double  COUNTS_PER_GEAR_REV    = COUNTS_PER_MOTOR_REV * GEAR_REDUCTION;
```

To calculate the number of counts per degree or moved or `COUNTS_PER_DEGREE` divide the `COUNTS_PER_GEAR_REV` variable by 360.

```
static final double     COUNTS_PER_DEGREE    = COUNTS_PER_GEAR_REV/360;
```

Add both these variables to the op mode.

```
public class HelloRobot_ArmControl extends LinearOpMode {
    private DcMotor arm;

    static final double     COUNTS_PER_MOTOR_REV    = 288;
    static final double     GEAR_REDUCTION     = 2.7778;
    static final double     COUNTS_PER_GEAR_REV    = COUNTS_PER_MOTOR_REV * GEAR_REDUCTION;
    static final double     COUNTS_PER_DEGREE    = COUNTS_PER_GEAR_REV/360;
```

Finally we need to create a non-constant variable that will act as our position. Create a variable called `armPosition` above the `waitForStart();` command.

```
public void runOpMode() {
        arm = hardwareMap.get(DcMotor.class, "arm");

        int armPosition;

        waitForStart();
```

Add this variable to the `if(gaempad1.dpad_up)` section of the `Target Position if/else if` statement, as this section dictates the 90 degree position. To get to the 90 degree position, the arm needs

Recall that `setTargetPosition()` requires an integer to be its parameter. When defining `armPosition` remember to add the line `(int)` in front of the double variable. However, you need to be cautious of potential rounding errors. Since `COUNTS_PER_MM` is part of an equation it is recommended that you convert to an integer after the result of the equation is found.

```
armPosition = (int)(COUNTS_PER_DEGREE * 45);
```

```
while (opModeIsActive()) {

   if(gamepad1.dpad_up){
         armPosition = (int)(COUNTS_PER_DEGREE * 45);
         arm.setTargetPosition(83);
         arm.setMode(DcMotor.RunMode.RUN_TO_POSITION);
         arm.setPower(0.4);
                 }
```

Set target position to `armPostion`.

```
if(gamepad1.dpad_up){
     armPosition = (int)(COUNTS_PER_DEGREE * 45);
     arm.setTargetPosition(armPosition);
     arm.setMode(DcMotor.RunMode.RUN_TO_POSITION);
     arm.setPower(0.4);
            }
else if (gamepad1.dpad_down){
      arm.setTargetPosition(0);
      arm.setMode(DcMotor.RunMode.RUN_TO_POSITION);
      arm.setPower(0.4);
            }
```

We could change what `armPosition` is equal to in the `gamepad1.dpad_down` portion of the `if/else if` statement such as:

```
else if (gamepad1.dpad_down){
       armPosition = (int)(COUNTS_PER_DEGREE * 0);
       arm.setTargetPosition(armPosition);
       arm.setMode(DcMotorEx.RunMode.RUN_TO_POSITION);
       arm.setPower(0.4);
             }
```

In this case we would consistently redefine `armPosition` to match the needs of whatever positions we want to create. Since our only two positions at the moment are our starting position and our 90 degree position it isn't necessary However, it is a good practice to create a variable in situations like this. If we want to add another position later, we can easily edit the variable to fit our needs.

Using Limits to Control Range of Motion

In the previous sections you worked on some of the building blocks for restricting an arms range of motion. From those sections you should have the foundation you need to perform basic arm control. However, there are some other creative ways you can use encoder positions and limits to expand the control you have over your arm.

This section will cover two additional types of control. The first type of control we will explore is the idea of **soft limits**. In the Adding a Limit Switch section we discuss the concept of physical limits of a mechanism however, there may be times you need to limit the range of motion of an arm without installing a physical limit. To do this position based code can be used to create a range for the arm.

Once you have a basic idea of how to create soft limits, we will explore how to use a limit switch (like a touch sensor) to reset the range of motion. This type of control reduces the risk of getting stuck outside of your intended range of motion, which can affect the expected behavior of your robot.

To set the soft limits we will use some of the basic logic we established in previous sections, with some edited changes. Start with a `Basic Op Mode` and add the constant variables from the Calculating Target Position section to the op mode.

```
@TeleOp

public class Basic extends LinearOpMode {
    private DcMotor  arm;

    static final double     COUNTS_PER_MOTOR_REV    = 288;
    static final double     GEAR_REDUCTION    = 2.7778;
    static final double     COUNTS_PER_GEAR_REV    = COUNTS_PER_MOTOR_REV * GEAR_REDUCTION;
    static final double     COUNTS_PER_DEGREE    = COUNTS_PER_GEAR_REV/360;

    @Override
    public void runOpMode() {
        arm = hardwareMap.get(DcMotor.class, "arm");


        waitForStart();

        while (opModeIsActive()) {
            telemetry.addData("Status", "Running");
            telemetry.update();

        }
    }
}
```

Next we need to create our upper and lower limits. Create two new integer variables one called `minPosition` and one called `maxPosition`. Add both of these to the in the initialization section of the

op mode above the `waitForStart()` command.

```
public void runOpMode() {
        arm = hardwareMap.get(DcMotor.class, "arm");

        int minPostion;
        int maxPosition;
        waitForStart();
```

For now we want the `minPosition` set as our starting position and the `maxPosition` set to our 90 degree position. Set `minPosition` equal to 0 and set `maxPosition` equal to `COUNTS_PER_DEGREE` times 45 .

> Remember you need to make a data type conversion!

```
int minPostion = 0;
int maxPosition = (int)(COUNTS_PER_DEGREE *45);
```

An `if/else if` statement needs to be added to control the arm, for this we can use the same basic logic we use in the Basics of Programming and Arm.

```
while(opModeIsActive()){
     if(gamepad1.dpad_up){
            arm.setPower(0.5);
            }
     else if (gamepad1.dpad_down){
            arm.setPower(-0.5);
            }
     else {
            arm.setPower(0);
            }
     }
```

To set the limit we need to edit our `if/else if` statement so that the limits are built in. If `DpadUp` is selected and the position of the arm is less than the `maxPosition` then the arm will move to the `maxPosition` . If `DpadDown` is selected and the position of the is greater that the `minPosition` then the arm will move towards the `minPosition` .

```
while (opModeIsActive()) {
    if (gamepad1.dpad_up && arm.getCurrentPosition() < maxPosition) {
            arm.setPower(0.5);
            }
    else if (gamepad1.dpad_down && arm.getCurrentPosition() > minPosition) {
            arm.setPower(-0.5);
            }
```

```
    else {
            arm.setPower(0);
            }
    }
```

The current code configuration will stop the motor at any point that the conditions to power the motor are not met. Depending to factors like the weight of the mechanism and any load that it is bearing, when the motor stops the arm may drop below the `maxPosition`. Take time to test out the code and confirm that it behaves in the way you expect it to.

Overriding Limits

One of the benefits of having a soft limit is being able to exceed that limit. Since encoders zero tick position is determined by the position of the arm when the Control Hub powers on; if attention is not payed to what position the arm is on power up the range of motion of the arm is affected. For instance, if we have to reset the Control Hub while the arm is in the 90 degree position, the 90 degree position is equal to 0 encoder ticks. One way around this is to create an override for the range of motion.

There are a few different ways an override of sorts can be created, in our case we are going to use the a button and touch sensor to help reset our range.

Start by editing the `if/else if` statement to add another `else if` condition. Use the line `gamepad1.a` as the condition. Add a the line `arm.setPower(-0.5);` as the action item.

```
while (opModeIsActive()) {
    if (gamepad1.dpad_up && arm.getCurrentPosition() < maxPosition) {
            arm.setPower(0.5);
            }
    else if (gamepad1.dpad_down && arm.getCurrentPosition() > minPosition) {
            arm.setPower(-0.5);
            }
    else if(gamepad1.a){
            arm.setPower(-0.5);
    else {
            arm.setPower(0);
            }
    }
```

Now that we have this change in place, when the `a` button is pressed the arm will move toward the starting position. When the arm reaches and presses the touch sensor we want to `STOP_AND_RESET_ENCODER`.

We can create another `if` statement that focuses on performing this stop and reset when the Touch Sensor is pressed. Since the Touch Sensor reports `true` when its not pressed and `false` when it is, we will need to use the **logical not operator** `!`.

The **not operator** `!` can be used in conditional binary statements when you need inverse whether something is `true` of `false`. For instance, an `if` statement activates when

> something is true, but when `touch.getState();` reports `true` it is not pressed. In our case we want this if statement to activate when the Touch Sensor is pressed thus we need to use the not operator.

```
if (!touch.getState()) {
          arm.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
        }
```

So, if the Touch Sensor returns `false` (or is pressed) the motor run mode `STOP_AND_RESET_ENCODER` will be activated causing the motor encoder to reset to 0 ticks.

Now that this code is done, try testing it!

```
@TeleOp

public class HelloRobot_ArmControl extends LinearOpMode {
    private DcMotor arm;
    private Servo claw;
    private Gyroscope imu;
    private DcMotor leftmotor;
    private DcMotor rightmotor;
    private DigitalChannel touch;

    static final double     COUNTS_PER_MOTOR_REV    = 288;
    static final double     GEAR_REDUCTION    = 2.7778;
    static final double     COUNTS_PER_GEAR_REV    = COUNTS_PER_MOTOR_REV * GEAR_REDUCTION;
    static final double     COUNTS_PER_DEGREE    = COUNTS_PER_GEAR_REV/360;



    @Override
    public void runOpMode() {
        arm = hardwareMap.get(DcMotor.class, "arm");
        claw = hardwareMap.get(Servo.class, "claw");
        imu = hardwareMap.get(Gyroscope.class, "imu");
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");
        touch = hardwareMap.get(DigitalChannel.class, "touch");

        int minPostion = 0;
        int maxPosition = (int)(COUNTS_PER_DEGREE *45);

        waitForStart();

        // run until the end of the match (driver presses STOP)
        while (opModeIsActive()) {
            if (gamepad1.dpad_up && arm.getCurrentPosition() < maxPosition) {
                arm.setPower(0.5);
                    }
            else if (gamepad1.dpad_down && arm.getCurrentPosition() > minPosition) {
```

```
                arm.setPower(-0.5);
                    }
            else if (gamepad1.a) {
                arm.setPower(-0.5);
                    }
            else {
                arm.setPower(0);
                }
            if (!touch.getState()) {
              arm.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
                    }
        telemetry.addData("Arm Test", arm.getCurrentPosition());
        telemetry.update();
        }
    }
}
```

Using Encoders

Basic Encoder Concepts

Each motor designed by REV has an encoder built into it that keeps track of its rotation. To use it, you must have a 4-pin JST PH cable connecting the motor to the Control Hub (REV-31-1595) or Expansion Hub (REV-31-1153), next to the 2-pin JST VH cable used to provide power to the motor.

Encoder values are measured in “ticks.” Different motors have different numbers of ticks per rotation of the output shaft based on the gear ratio of the motor. When the Control Hub is turned on, all of its encoder ports are at 0 ticks. As a motor moves forward, its encoder value increases. As a motor moves backwards, its encoder value decreases.

For more information see the section on encoders.

Choosing a Motor Mode

Your programs can always access the encoder values directly, but you can also direct the Control Hub to use the encoder values to maintain a motor’s speed, or maintain a particular position. You do this by changing the motor’s mode.

> ⓘ It is recommended to use the latest Control Hub and Expansion Hub firmware before using RUN_USING_ENCODER mode or RUN_TO_POSITION mode.

STOP_AND_RESET_ENCODER Mode

Place a motor in this mode when you want to set its encoder position back to 0. The motor will stop. To start it again, you need to place the motor into one of the other three modes. It is recommended to place each motor you will be using encoders with into this mode at the start of each program, so that you know what position the motor is starting out in.

RUN_WITHOUT_ENCODER Mode

Use this mode when you don't want the Control Hub to attempt to use the encoders to maintain a constant speed. You can still access the encoder values, but your actual motor speed will vary more based on external factors such as battery life and friction. In this mode, you provide a power level in the -1 to 1 range, where -1 is full speed backwards, 0 is stopped, and 1 is full speed forwards. Reducing the power reduces both torque and speed.

> ⓘ This mode is a good choice for drivetrain motors driven by joysticks on the gamepad.

RUN_USING_ENCODER Mode

In this mode, the Control Hub will use the encoder to take an active role in managing the motor's speed. Rather than directly applying a percentage of the available power, RUN_USING_ENCODER mode targets a specific velocity (speed). This allows the motor to account for friction, battery voltage, and other factors.

> ⓘ This mode is a good choice for operations, like a flywheel, that require a specific speed and can use buttons on a gamepad for control.

RUN_TO_POSITION Mode

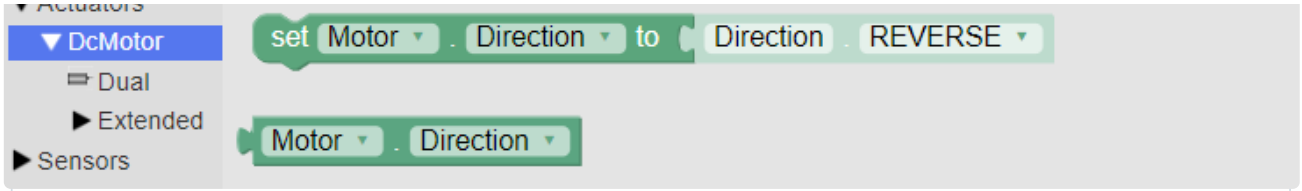
In this mode, the Control Hub will target a specific position, rather than a specific velocity. You still set a velocity, but it is only used as the maximum velocity. The motor will continue to hold its position even after it has reached its target.

> ⓘ This mode is a good choice for operations, like an arm, that require a specific position and can use buttons on a gamepad for control.

Reading the Encoder Value

Blocks

In Blocks, you access the current encoder value by using the DcMotor CurrentPosition block.



Java

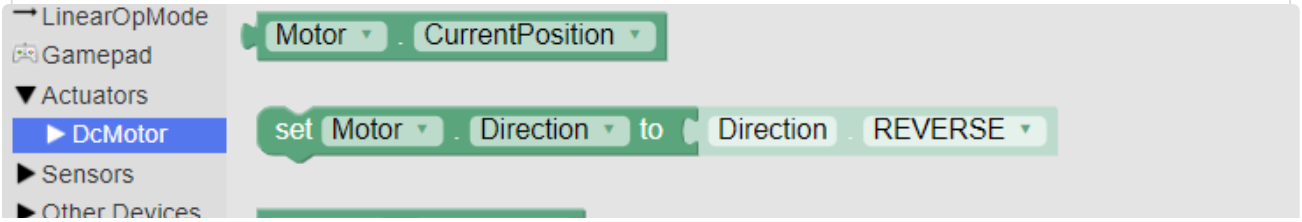
In Java, you access the current encoder value by calling `getCurrentPosition()` on a DcMotor or DcMotorEx object. This sample program prints the encoder value for a motor configured with the name “Motor” to telemetry:

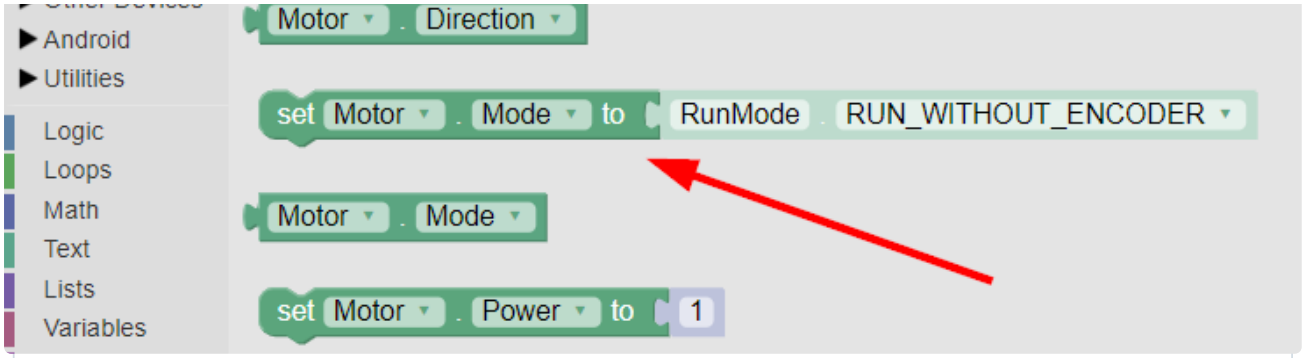
```
package org.firstinspires.ftc.teamcode;
// import lines were omitted. OnBotJava will add them automatically.

@TeleOp
public class JavaEncoderTest extends LinearOpMode {
    DcMotorEx motor;

    @Override
    public void runOpMode() {
        motor = hardwareMap.get(DcMotorEx.class, "Motor");
        waitForStart();
        while (opModeIsActive()) {
            telemetry.addData("Encoder value", motor.getCurrentPosition());
            telemetry.update();
        }
    }
}
```

Setting the Motor Mode

Blocks

In Blocks, you set the motor’s mode with this block. You can select different modes from its dropdown menu.





Java

Here is a snippet of code that demonstrates how to do the same thing in Java. You can skip the first line if you already have retrieved the motor object from hardwareMap. Change RUN_WITHOUT_ENCODER to the desired motor mode (STOP_AND_RESET_ENCODER, RUN_WITHOUT_ENCODER, RUN_USING_ENCODER, or RUN_TO_POSITION).

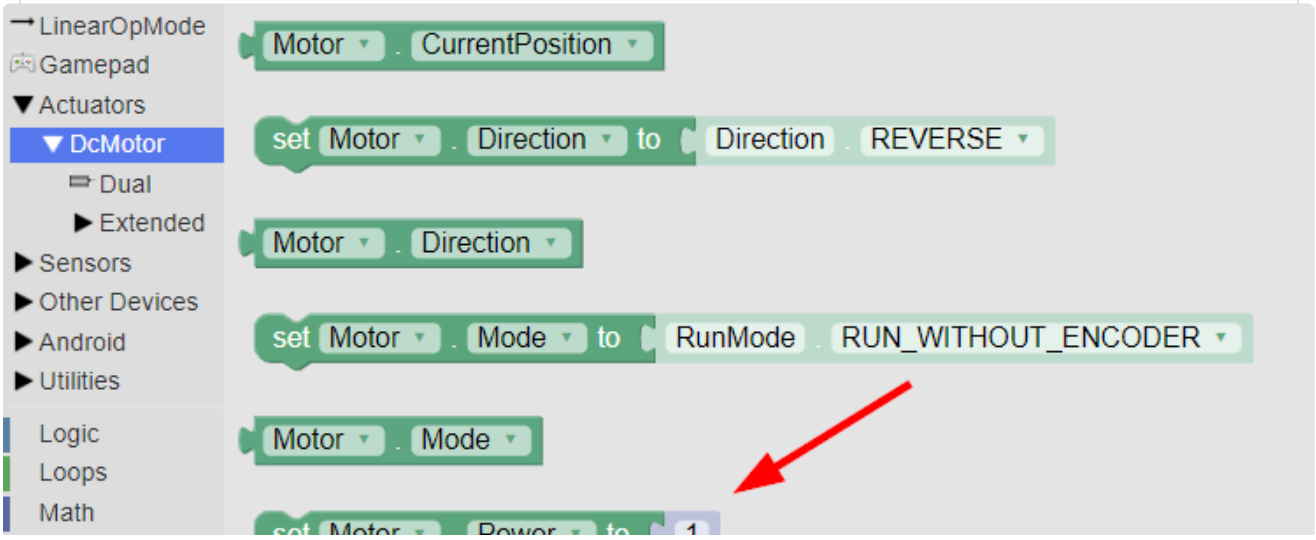
```
DcMotorEx motor = hardwareMap.get(DcMotorEx.class, "Motor");
motor.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODER);
```

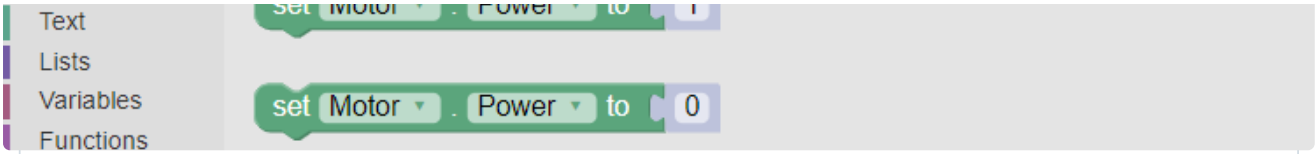
Using RUN_WITHOUT_ENCODER

The RUN_WITHOUT_ENCODER motor mode is very straightforward, you simply set a power in the range of -1.0 to 1.0. However, if you try to set a velocity (which will be covered later on), the motor will automatically be switched into RUN_USING_ENCODER mode.

Blocks

The power level is set in Blocks mode using this block:





Java

The power level is set in Java by calling `setPower()` on a DcMotor or DcMotorEx object, as is shown in this snippet. You can skip the first two lines if you already have retrieved the motor object from hardwareMap and set the mode to RUN_WITHOUT_ENCODER.

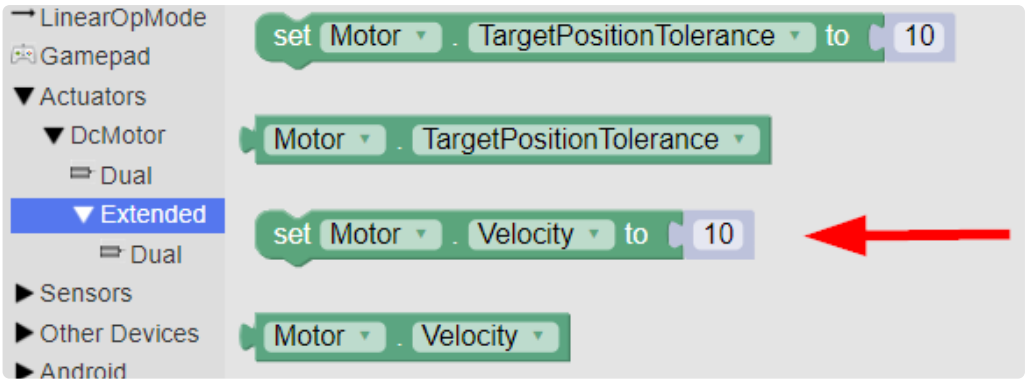
```
DcMotorEx motor = hardwareMap.get(DcMotorEx.class, "Motor");
motor.setMode(DcMotor.RunMode.RUN_WITHOUT_ENCODER);
// This will run the motor forward at half-power
double motorPower = 0.5;
motor.setPower(motorPower);
```

Using RUN_USING_ENCODER

In RUN_USING_ENCODER mode, you should set a velocity (measured in ticks per second), rather than a power level. You can still provide a power level in RUN_USING_ENCODER mode, but this is not recommended, as it will limit your target speed significantly. Setting a velocity from RUN_WITHOUT_ENCODER mode will automatically switch the motor to RUN_USING_ENCODER mode. You should pick a velocity that the motor will be capable of reaching even with a full load and a low battery.

Blocks

Providing a velocity is an extended motor feature, which means that the block for it is located under DcMotor > Extended. You can see it here:



Java

The velocity is set in Java by calling `setVelocity()` on a DcMotorEx object, as is shown in this snippet. You can skip the first two lines if you have already retrieved the motor object as a DcMotorEx from hardwareMap and set the mode to RUN_USING_ENCODER.

```
DcMotorEx motor = hardwareMap.get(DcMotorEx.class, "Motor");
motor.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
// This will turn the motor at 200 ticks per second
double motorVelocity = 200;
motor.setVelocity(motorVelocity);
```

Using RUN_TO_POSITION

To use RUN_TO_POSITION mode, you need to do the following things in this order:

1. Set a target position (in ticks)
2. Switch to RUN_TO_POSITION mode
3. Set the maximum velocity

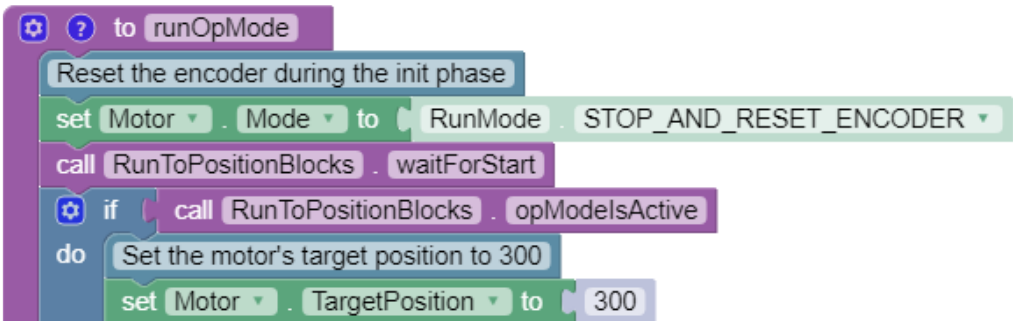
You should reset the encoders (switch to STOP_AND_RESET_ENCODER mode) during initialization when you use RUN_TO_POSITION mode. If you are using it with a mechanism such as a lift, you have to be careful to make sure that you always have the motor in the same physical location when you reset the encoders, or else your target position won't mean the same thing between runs.

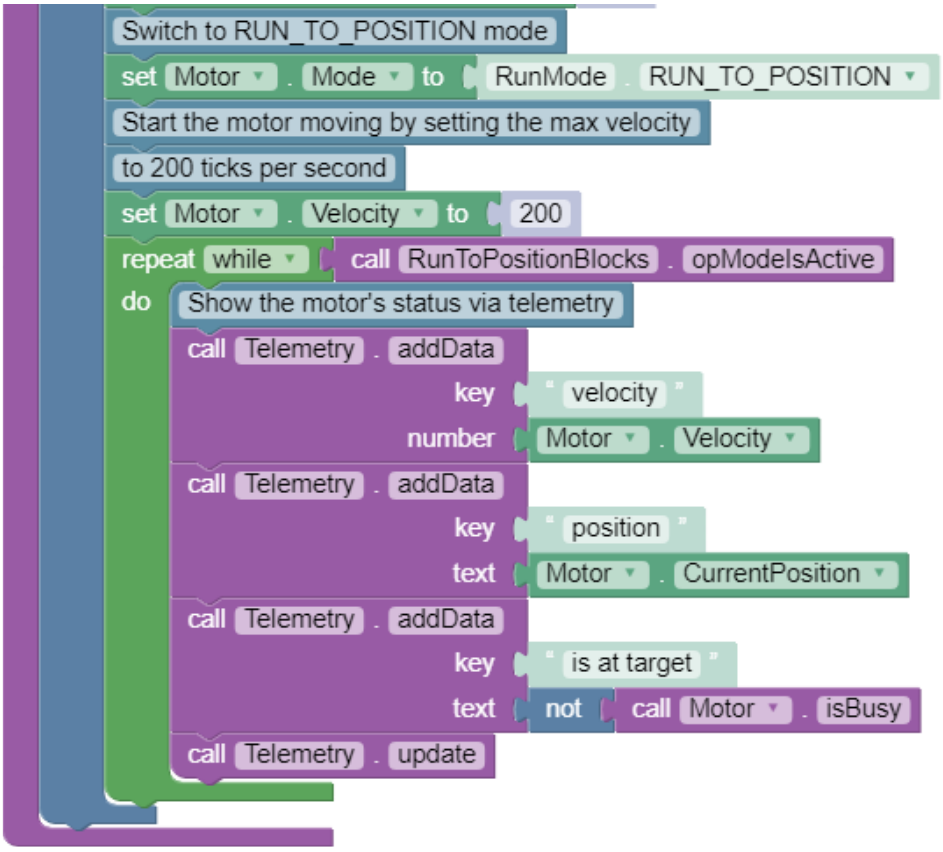
The motor will continue to hold its position even after it has reached its target, unless you set the velocity or power to zero, or switch to a different motor mode.

The following examples assume that the motor used is a Core Hex Motor. If it is a motor that has a more precise encoder, such as an HD Hex Motor, higher velocity and target position values will be more appropriate.

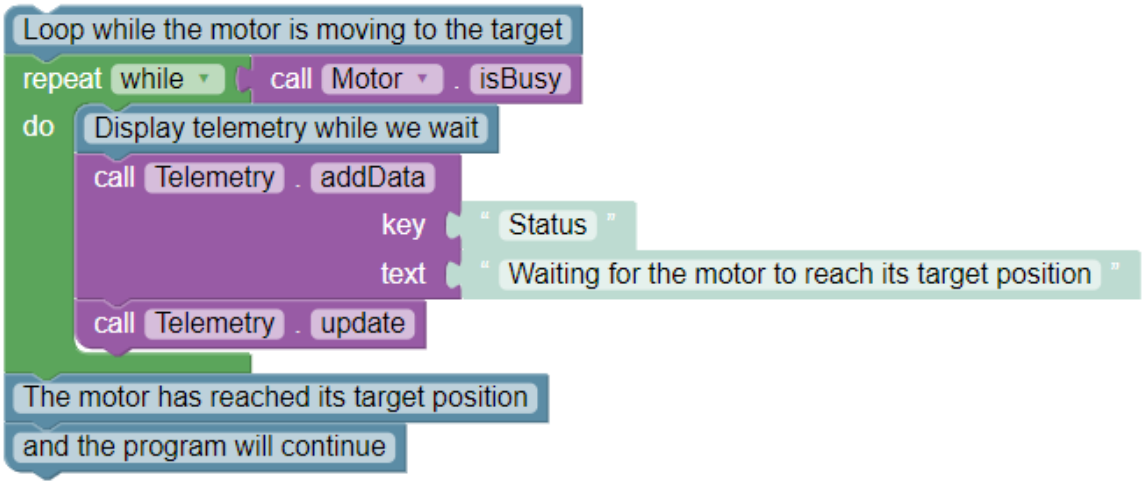
Blocks

Here is a complete Blocks program that uses RUN_TO_POSITION.





If you want to wait for the motor to reach its target position before continuing in your program, you can use a while loop that checks if the motor is busy (not yet at its target):



Java

```java
package org.firstinspires.ftc.teamcode;
// import lines were omitted. OnBotJava will add them automatically.

@TeleOp
public class JavaRunToPositionExample extends LinearOpMode {
    DcMotorEx motor;

    @Override
    public void runOpMode() {
```

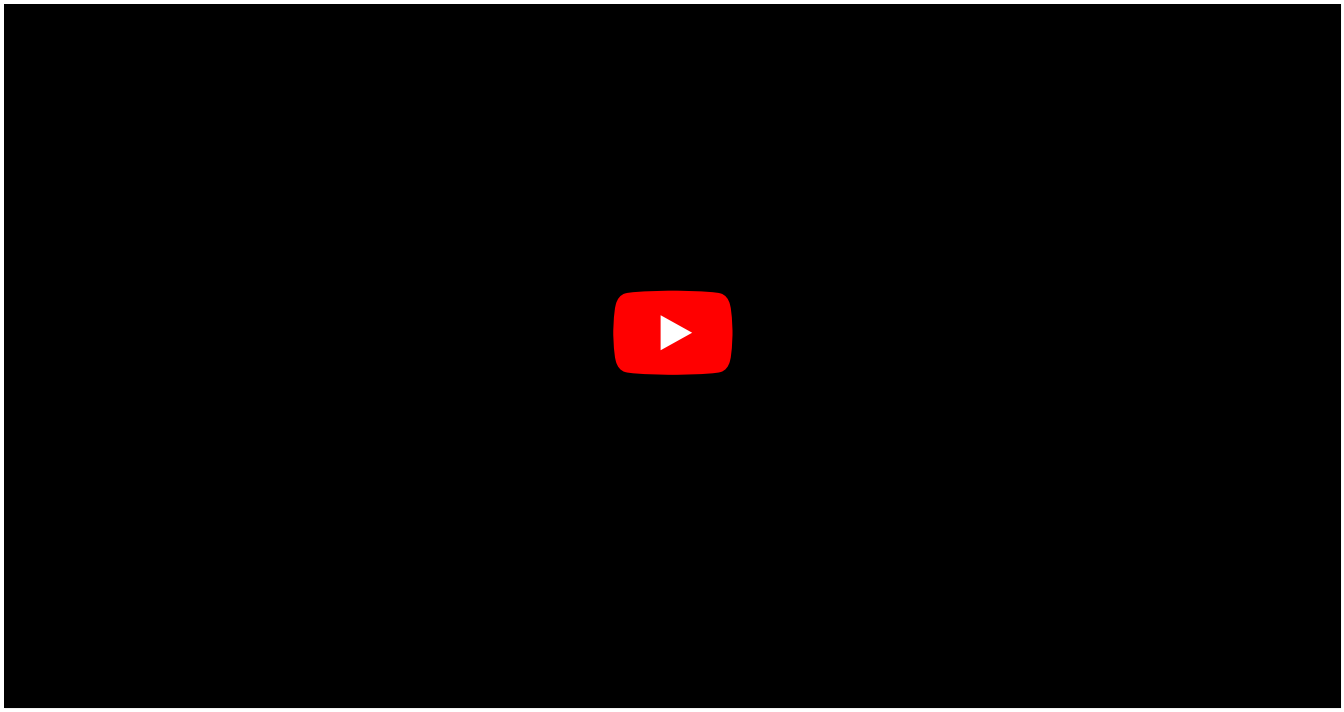
```
        motor = hardwareMap.get(DcMotorEx.class, "Motor");
        // Reset the encoder during initialization
        motor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);

        waitForStart();

        // Set the motor's target position to 300 ticks
        motor.setTargetPosition(300);

        // Switch to RUN_TO_POSITION mode
        motor.setMode(DcMotor.RunMode.RUN_TO_POSITION);

        // Start the motor moving by setting the max velocity to 200 ticks per seco
        motor.setVelocity(200);

        // While the Op Mode is running, show the motor's status via telemetry
        while (opModeIsActive()) {
            telemetry.addData("velocity", motor.getVelocity());
            telemetry.addData("position", motor.getCurrentPosition());
            telemetry.addData("is at target", !motor.isBusy());
            telemetry.update();
        }
    }
}
```

If you want to wait for the motor to reach its target position before continuing in your program, you can use a while loop that checks if the motor is busy (not yet at its target):

```
// Loop while the motor is moving to the target
while(motor.isBusy()) {
 // Let the drive team see that we're waiting on the motor
 telemetry.addData("Status", "Waiting for the motor to reach its target");
 telemetry.update();
}
// The motor has reached its target position, and the program will continue
```

Android Studio - Deploying Code Wirelessly

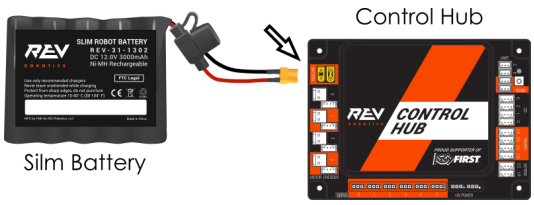
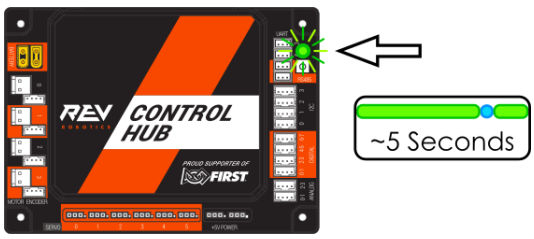
Android Debug Bridge (ADB) utility is the tool used by Android Studio to connect and control Android devices, like the Control Hub. Android Studio, using ADB, allows users to build and install the Robot Controller app onto their Control Hub.



By default ADB supports using a hardwire connection via USB to deploy code to Android Devices. ADB does support a wireless mode where the build and install process is sent over Wi-Fi. The Control Hub is configured to support ADB wireless connections on port 5555. To deploy code over the Wi-Fi connection the user will need to set up Wireless ADB.

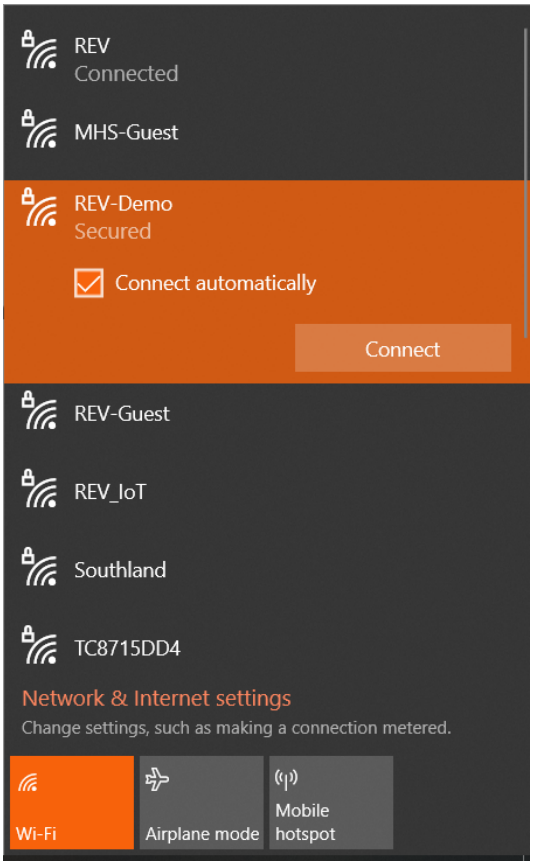
Setting Up Wireless ADB using the REV Hardware Client

To set up wireless ADB using the REV Hardware Client you will need a laptop or PC with both Android Studio and the REV Hardware Client installed.

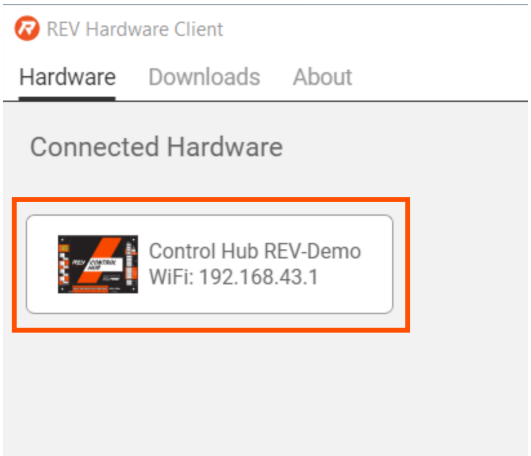
Power on the Control Hub, by plugging the 12V Slim Battery (REV-31-1302) into the XT30 connector labeled “BATTERY” on the Control Hub.	
The Control Hub is ready to connect with a PC when the LED turns green. Note: the light may blink blue every ~5 seconds to indicate that the Control Hub is healthy depending on Robot Controller version.	

Connect to the Wi-Fi Network created by the Control Hub

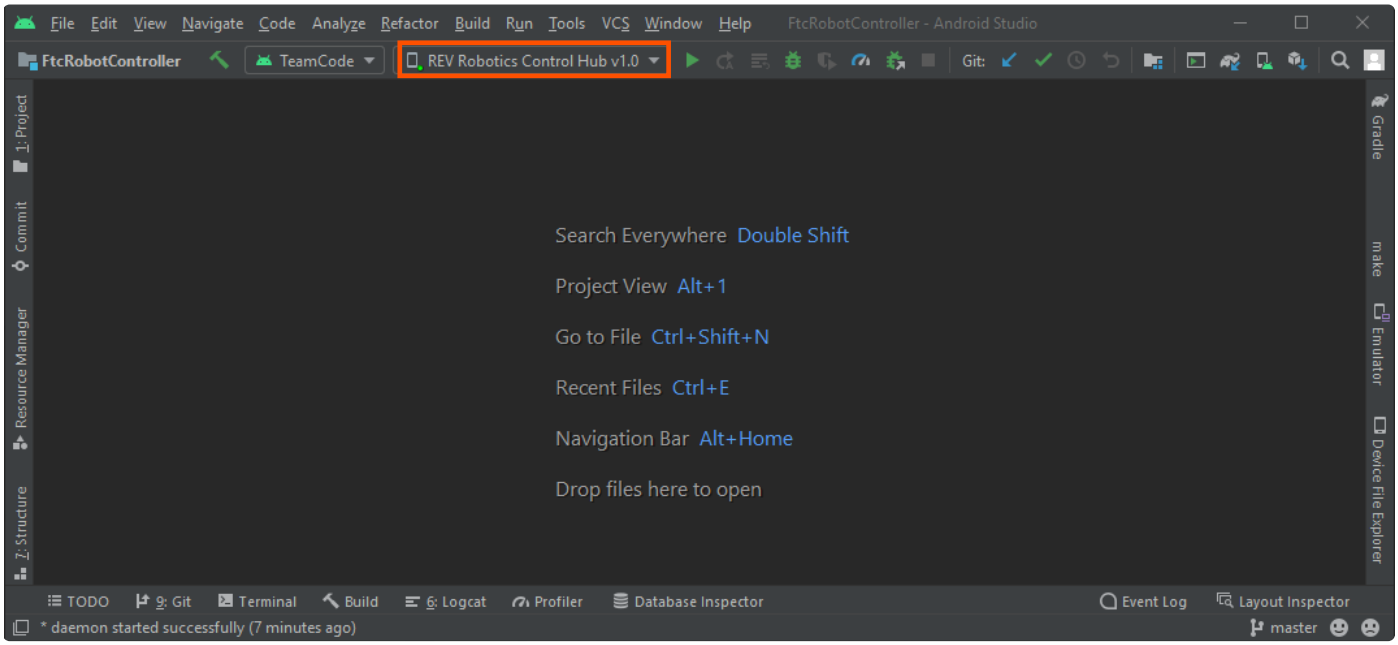
Note: Connect to the REV Hardware Client over USB if the password needs resetting.



Open the REV Hardware Client and confirm the Control Hub is connected over Wi-Fi



The Control Hub should be listed in the Android Studio devices dropdown



Sensors

Introduction to Sensors

Sensor Basics

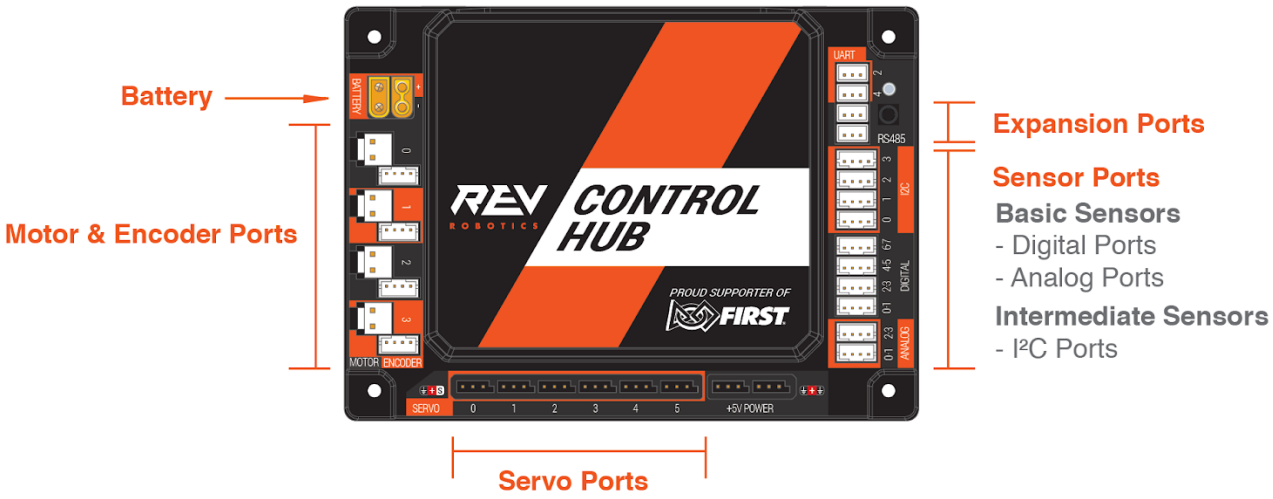
When starting out many of the robot actions can be accomplished by turning on a motor for a specific amount of time. Eventually, these time-based actions may not be accurate or repeatable enough. As battery power drains while the robot is running and mechanisms wearing in through use can all affect time-based actions. Fortunately, there is a way to give feedback to the robot about how it is operating by using sensors; devices that are used to collect information about the robot and the environment around it.

Sensors provide information that allows you to program the robot to use this information to perform specific actions. This allows the robot to perform at its best and in a repeatable manner. A few scenarios that can benefit from a sensors information are listed below.

Scenarios where a sensor is needed:

- The robot needs to autonomously move to a specific location and stop there.
- The robot needs to move forward at a green signal and stop moving at a red signal.
- The robot has an arm that needs to be prevented from rotating too far or it may damage other parts of the robot.
- The robot needs to stop 1 meter away from an opaque wall.
- The robot needs to be able to tell how many game objects it is currently holding inside its hopper.

Different Sensor Types and Uses



Control Hub ports

In the REV Robotics Control System sensors are classified as **basic**, **intermediate**, or **advanced**. This division among sensors is based on programming complexity. **Basic sensors** can typically be coded using a if/else statement. **Intermediate sensors,** like the Color Sensor or Encoders, require a higher level understanding of programming. **Advanced sensors** require an advanced knowledge of programming. Visions sensors and using the Inertial Measurement Unit (IMU) are considered advanced.

Basic

In the REV Robotics Control System, both **Analog** and **Digital** sensors are considered basic sensors.

Digital sensors provide **binary** information: information that can take one of two possible values or states. These states are represented in programming languages as: **TRUE/FALSE** or **1/0**. Electrically, these states are usually represented as two voltages: a **High** voltage and a **Low** voltage. For REV Hubs, High is 3.0V and Low is 0V.

A touch sensor is a common digital sensor. It has two states: pressed and not-pressed.

Analog sensors provide a range of information with an almost infinite set of values, instead of just two. These values are usually represented in programming languages as decimal numbers. Electrically, these values are represented as voltage. REV Hubs can measure voltages on the analog ports between 0V and 5.0V.

Depending on the sensor, the reported voltage can represent anything that can't be represented by two digital states. A potentiometer is a common analog sensor that reports the angle of an attached shaft as voltages.

> *Some sensors in the REV Control System are capable of running up to 5V. To learn more about sensor voltage visit the pages of the individual sensors!*

The table below gives the basic usage scenarios for analog and digital sensors

Digital	Analog
Gives feedback as either on or off. This type of sensor is ideal for setting limits of a mechanism.	Gives feedback as a proportional voltage range. This type of sensor is ideal for knowing exactly where a mechanism is, like a dial on a radio.

Digital Sensors

- Touch Sensor: A sensor with a button. The button press can be used to trigger actions like stopping motors.
- Magnetic Limit Switch: A sensor that detects magnetic fields. When there is sufficient field strength of either magnetic pole detected the sensor is triggers and a limit of movement can be established.

Analog Sensors

- Potentiometer: A sensor that senses the angular position of a shaft.

Intermediate

I2C sensors are considered intermediate because they give feedback through two-way communication with a robot controller. These types of sensors allow for more complex data to communicate to the robot, such as color values of an object.

- Color Sensor: A sensor capable of sensing colors and proximity of objects.
- 2m Distance Sensor: A sensor typically used to detect the distance from the sensor to other opaque objects.

All REV Robotics motors contain a built-in intermediate-level sensor called an Encoder. An **Encoder**, in the context of robotics, is a type of digital sensor that converts rotary motion into digital signal. These type of sensors require “decoding” to get this information into a usable form. The Control Hub and Expansion Hub have built in decoding through the “Encoder Ports” under the motor ports.

Advanced

Advanced sensors, in the REV Control System, are considered advanced as they rely on complex coding and information from other sensors in order to work effectively. Both the IMU and vision sensors require higher level code in order to decipher information being received from the sensor.

Vision	IMU
Gives feedback as images to the robot controller. These types of sensors require the use of image	The IMU incorporates three sensors: a 3-axis accelerometer, a 3-axis gyroscope, and a 3-axis

processing software, like VuForia, to use to their full | geomagnetic sensor. This sensor can be used to

Digital

Digital Sensor Basics

The information from digital sensors comes in two states, also known as binary states. The binary state of a digital sensor is either low or high. This is similar to a light switch being on or off.

> ⓘ Binary information, or states, can be thought of as an "either/or"; a light switch can either be in an 'on' state or an 'off' state. On/off, 0/1, low/high, and FALSE/TRUE are all different ways of presenting binary information. In programming FALSE/TRUE is used most often.

The main difference between a light switch and a digital sensor is that a digital sensor has a **default state.** The default state is typically its **inactive states**. Digital sensor datasheets typically will report the sensors **active behavior**, either **active-low** or **active-high**. With an active-high behavior, when the digital sensor is triggered (or activated) you can detect a change in code from a "logic" low state to a "logic" high state.

> ⓘ Logic Level represents the voltage difference between the signal and ground of the Control and Expansion Hub's sensor ports. Both Hubs and REV Sensors operate on a 3.3V logic level. This means the digital sensor needs an operating voltage of 3.3V for use with the Hub. If you are looking to use a 5V digital sensor you will need a Logic Level Converter. See Using 5V Sensors for more information.

This change is from FALSE to TRUE and you can program your robot to act accordingly with this information. Check the datasheet for the sensor you are using to determine its **active behavior** is and how the behavior is reported in your code.

REV carries the following digital sensors:

- Touch Sensor (REV-31-1425)
- Magnetic Limit Switch (REV-31-1462)

Wiring

Digital sensors connect to the Control Hub (REV-31-1595), or Expansion Hub (REV-31-1153), via a JST PH 4-Pin Sensor Cable and the Digital Ports, shown in the image above. The color-coding of the digital ports in the image corresponds with each wire in the JST PH 4-Pin Sensor Cable. Following convention, the black wire is ground and the red wire is power. The blue **(n)** and white **(n+1)** wires are the communication (signal) channels along which the sensor sends feedback to the Hubs.

Each digital port on the Hub is capable of acting as two separate ports, thanks to the two channels of communication. This is why the ports are marked as 0-1, 2-3, etc. The image above shows which channel of communication corresponds with which port. The n+1 channel operates on odd-numbered ports 1-7 and the n channel operates on the even number ports 0-6.

Two digital sensors may be hosted on the same physical port using the Sensor Splitter Cable. That being said, it is important to check the Pinout Diagram included in the datasheets for each individual sensor, as certain sensors, like the Touch sensor, use only one of the communication channels.

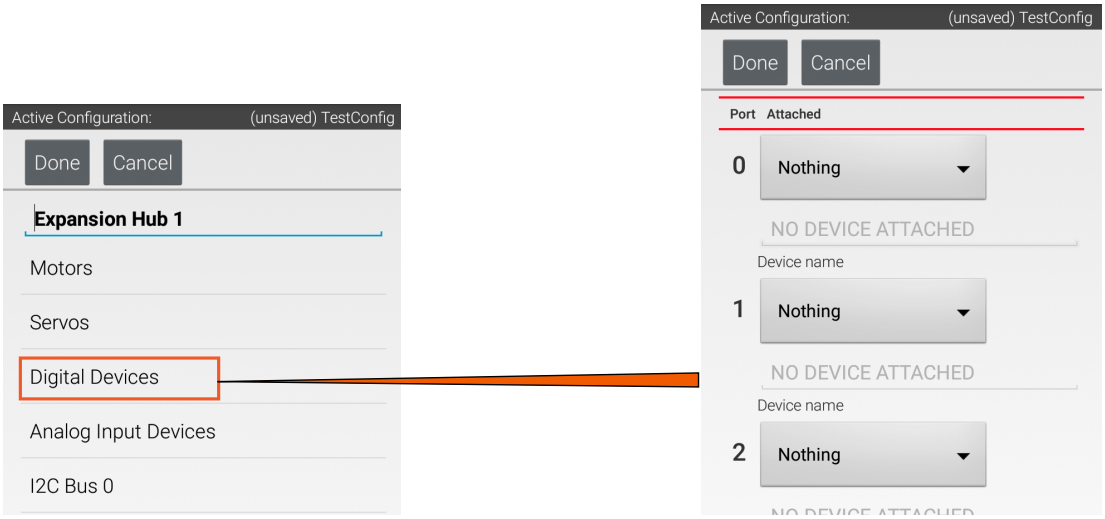
Configuration

Before a sensor can be programmed it must be added to the Robot Configuration. The configuration file stores all configured devices in the Control Hub's "hardwareMap," which can be called to in the code to establish the line of communication between devices.

The steps below show the basic configuration for digital devices. In the example, the Touch Sensor is configured as "REV Touch Sensor" on port 1.

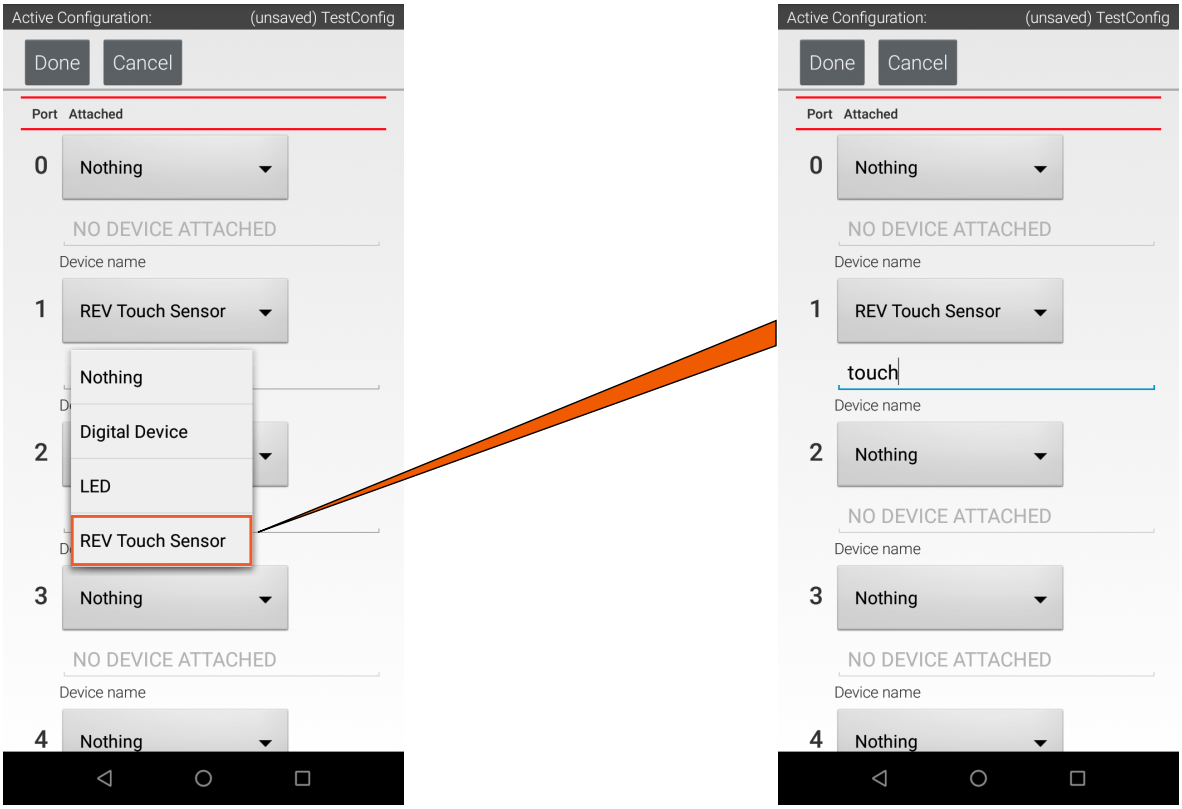
Step 1

While in the configuration select the **Digital Devices** option. This will open a screen that shows the eight digital ports.



Step 2

In the drop-down menu for Port 1 select "REV Touch Sensor." After it is selected name the sensor. In this example, the Touch Sensor is named "touch," but any naming convention can be used.



Step 3

When you have finished configuring the sensor hit **Done**. The app will return to the previous screen

> For more information on configuring the Touch Sensor or Magnetic Limit Switch go to the sensor datasheets.

Applications

How do digital sensors help a robot navigate the world around it? The REV Touch Sensor and REV

Magnetic Limit Switch are most commonly used as **limit switches**! Limit switches can help detect when a mechanism, like an arm and/or a lift, has reached its physical limits. Installing a limit switch can help keep robot mechanisms from overextending and breaking. They can also be used to zero out the position of motor encoders to further reduce mechanical failure.

For more information on how to use the REV Digital Sensors as limit switches, sensor specifications, coding examples, and more; click one of the links below to head to the sensor datasheets

Touch Sensor (REV-31-1425)
Magnetic Limit Switch (REV-31-1462)
Digital LED Indicator (REV-31-2010)

Analog

Analog Sensor Basics

Analog sensors can report an almost infinite number of states unlike digital sensors that report only two states. As the state of the sensor changes, the voltage reporting back to the robot changes as well. Think of a dimmer switch, the brightness of the lights in the room depends on where the slider or knob is positioned along the scale of potential positions. As the knob is adjusted the voltage level adjusts proportionally and the light continuously changes to the output from the knob.

> Can you think of anything that acts like analog sensors around your household? Here are some we thought of: scale, thermometer, volume knob

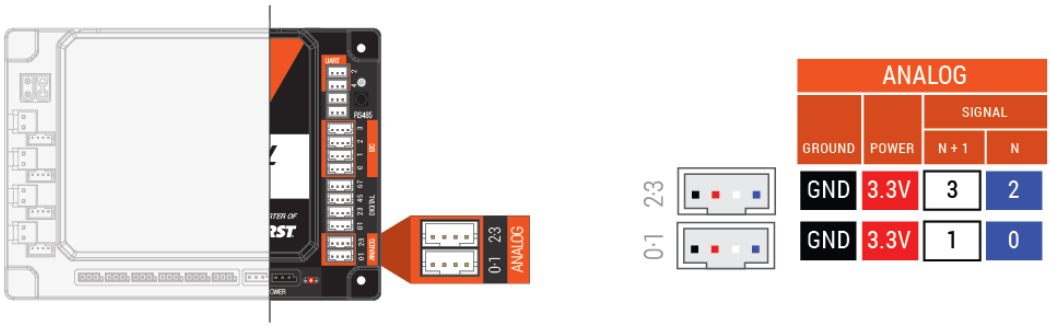
Unlike the binary (low/high) status of digital sensors, analog sensors consider all numbers within a specific, given range. When using an analog sensor the actionable trigger will vary depending on the sensor. Consider a potentiometer attached to an arm, the output voltage (signal) will correspond to an angle of the arm. Knowing the angle of the arm then allows you to decide where to stop the arm along its travel path.

> The Control Hub and Expansion Hub can read voltages ranging from 0V to 5V.

REV Robotics offers analog sensor, known as a Potentiometer (REV-31-1155). The Potentiometer can be used to sense or measure the angular position of a shaft.

Wiring

Wiring



Analog sensors connect to the Control Hub (REV-31-1595), or Expansion Hub (REV-31-1153), via a JST PH 4-Pin Sensor Cable and the Analog Ports, shown in the image above. The color-coding of the analog ports in the image corresponds with each wire in the JST PH 4-Pin Sensor Cable. As a convention, the black wire is ground and the red wire is power. The blue (n) wire and white (n+1) wire are the communication (signal) channels along which the sensor sends feedback to the Hubs.

Each analog port on the Hub is capable of acting as two separate ports, thanks to the two channels of communication. This is why the ports are marked as 0-1 and 2-3. The image above shows which channel of communication corresponds with which port. The n+1 channel operates on odd-numbered ports 1-3 and the n channel operates on the even number ports 0-2.

Two analog sensors may be hosted on the same physical port using the Sensor Splitter Cable (REV-31-1386). That being said, it is important to check the Pinout Diagram included in the datasheets for each individual sensor, as certain sensors, like the REV Potentiometer, use only one of the communication channels.

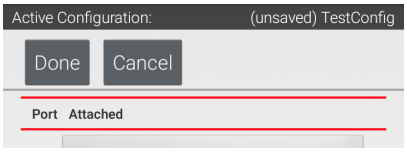
Configuration

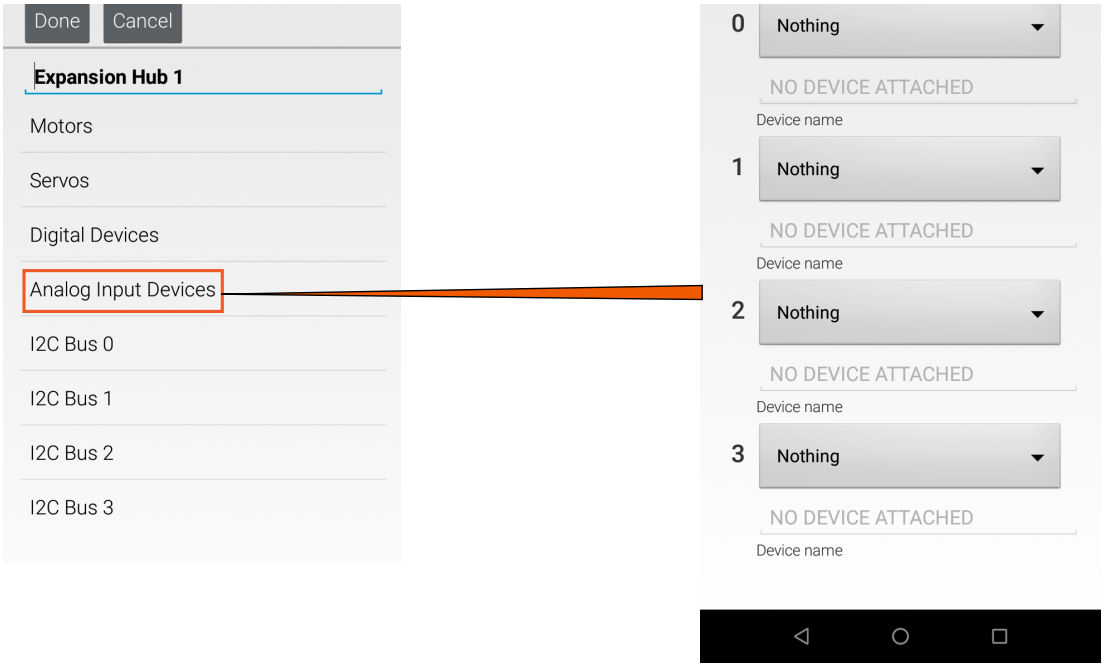
Before a sensor can be programmed it must be added to the Robot Configuration. The configuration file stores all configured devices in the Control Hub's "hardwareMap," which can be called to in the code to establish the line of communication between devices.

The steps below show the basic configuration for analog devices. In the example, the Potentiometer will be configured as "Analog Input`" on port 0.

Step 1

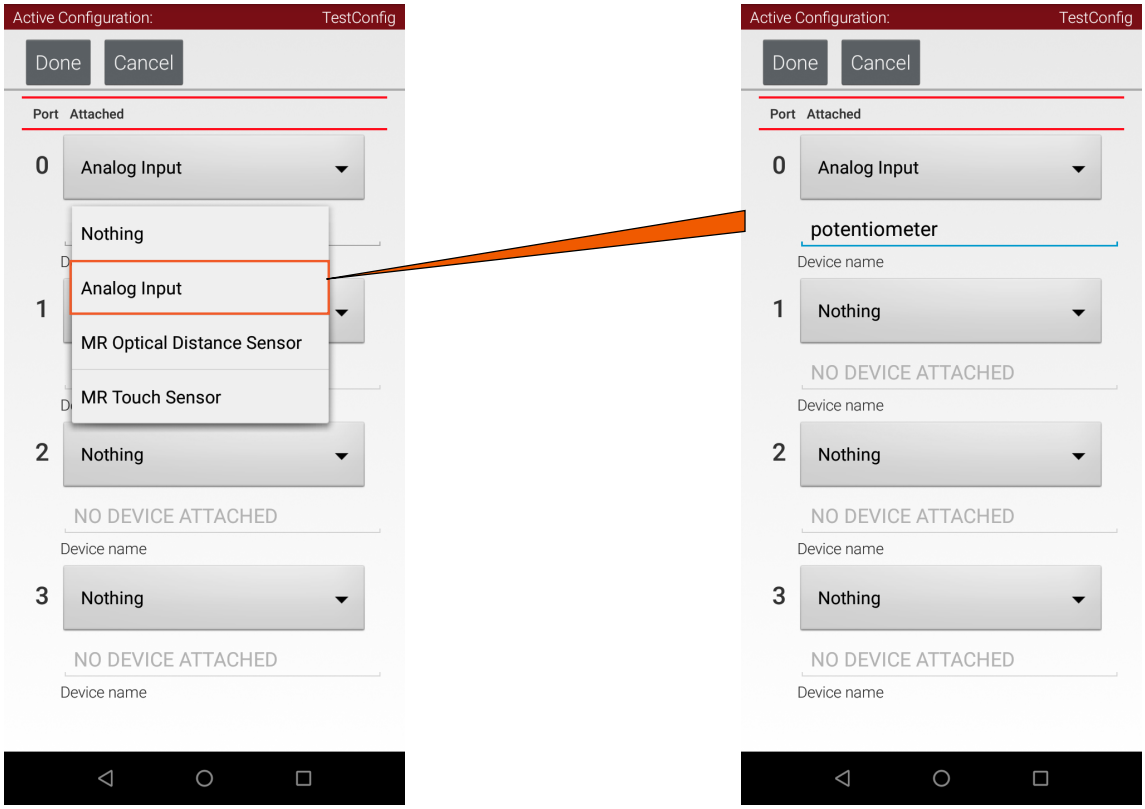
While in the configuration select the **Analog Input Devices** option. This will open a screen that shows the four analog ports.





Step 2

In the drop-down menu for Port 0 select "Analog Input." After it is selected name the sensor. In this example, the Potentiometer is named "potentiometer," but any naming convention can be used.



Step 3

When you have finished configuring the sensor hit **Done**. The app will return to the previous screen.

Applications

How does the Potentiometer help a robot navigate the world around it? Potentiometers are most commonly used to measure the angle of an arm type joint. The angle measurement can be used to set or find a specific position along the arm joint.

For more information on the REV Potentiometer's sensor specifications, coding examples, and more; click one of the links below to head to the sensor datasheets

Potentiometer (REV-31-1155)

I2C

I2C Sensor Basics

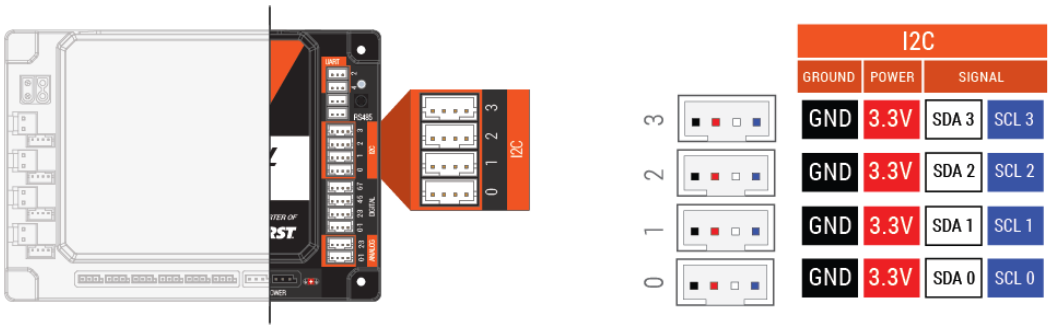
I2C is a common electronic communication standard that allows a **host** (the Hub) to communicate with multiple **devices** on the same I2C bus. Each I2C port on a Hub is its own **I2C bus**. Every I2C device has a unique address, a number that is normally fixed by the manufacturer. All of the devices on an individual I2C bus must have a unique address so that the host can communicate with one device at a time. If two devices have the same address, such as when using two of the same kind of sensors, they must be used on different I2C buses otherwise the communication channels conflict.

> ⓘ While I2C is technically a digital communication protocol, it is more advanced than the simple on/off style of basic digital sensors. I2C sensors require software drivers for the information from a follower (sensor) to be interpreted by the leader (hub).

There are three I2C sensors within the REV system: the Inertial Measurement Unit (IMU), Color Sensor (REV-31-1557), and 2m Distance Sensor (REV-31-1505). The IMU is built into the Control Hub (REV-31-1595) and Expansion Hub (REV-31-1153) and is connected to I2C bus 0.

> ⓘ Logic Level represents the voltage difference between the signal and ground of the Control and Expansion Hub's sensor ports. Both Hubs and REV Sensors operate on a 3.3V logic level. This means the digital sensor needs an operating voltage of 3.3V for use with the Hub. If you are looking to use a 5V I2C sensor you will need a Logic Level Converter. See Using 5V Sensors for more information.

Wiring



I2C sensors connect to the Control Hub (REV-31-1595), or Expansion Hub (REV-31-1153), via a JST PH 4-Pin Sensor Cable and the I2C buses, shown in the image above. The color-coding of the I2C buses in the image corresponds with each wire in the JST PH 4-Pin Sensor Cable. As a convention, the black wire is ground and the red wire is power. The blue (SCLn) wire and white (SDAn) wire are the communication signals for each I2C bus on the Hubs.

Sensor feedback to the Hub works differently for the I2C sensor than it does for Analog or Digital sensors. With the Analog and Digital Sensors, only one communication channel needs to be used by an individual sensor. In contrast, an I2C sensor sends different kinds of information over the SDA (white) and SCL (blue) wires. Since the I2C is transferring more complex data to the Hub then Analog or Digital sensors, there has to be a component of harmonization, or consistency, as the data moves from the sensor to the Hub. The **SCL (Serial Clock)** channel provides consistency by acting as a clock line and time-stamping the data provided by the **SDA (Serial Data)** channel.

While it is possible to host more than one I2C sensor on the same bus, there are a couple of factors to take into account. The Hub keeps track of the information from different sensors by considering the sensor's address in relation to the data being sent. When two sensors have the same address, like the REV Color Sensor V3 and the 2m Distance Sensor, they cannot be hosted on the same bus. Check the sensor datasheets for all I2C sensors to determine what sensors can and cannot be hosted on the same bus.

Currently, REV Robotics does not produce a cable or breakout board to connect two sensors to one I2C port on the Hub. A custom cable will need to be made in order to wire more than one I2C to the Hub.

> ⓘ The internal IMU is hosted on I2C bus 0. See the configuration section below to learn more about configuring a secondary sensor on bus 0.

Configuration

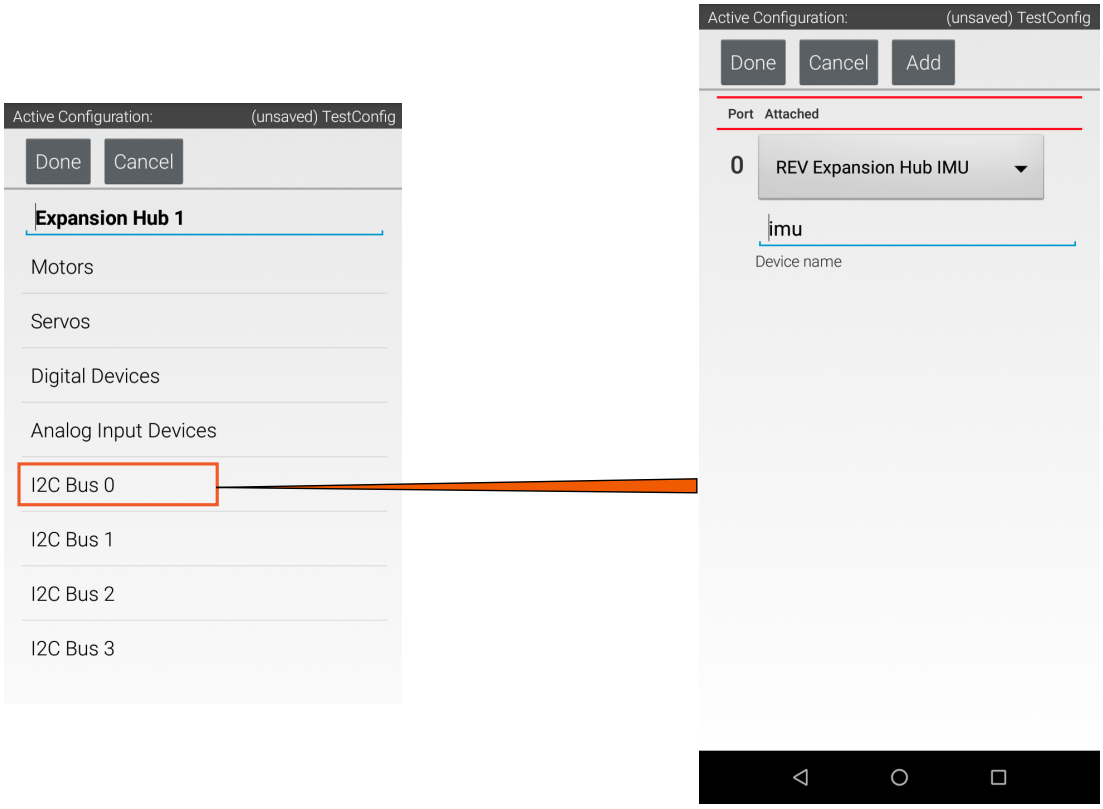
Before a sensor can be programmed it must be added to the Robot Configuration. The configuration file stores all configured devices in the Control Hub's "hardwareMap," which can be called to in the code to establish the line of communication between devices.

In order to function, all FTC legal I2C devices have drivers installed into the SDK. With regards to configuration, this means that the device has to be set to the drop-down menu item that corresponds with its drivers. Visit the datasheet for the sensor you are trying to configure to see how to configure it.

The steps below shows a basic configuration for I2C devices. The I2C Bus 0 hosts the internal IMU sensor within the Hubs. In this example, the Color Sensor V3 is being added to Bus 0 as well.

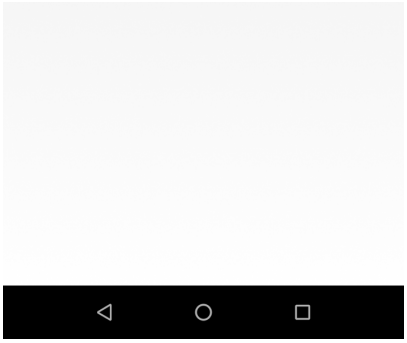
Step 1

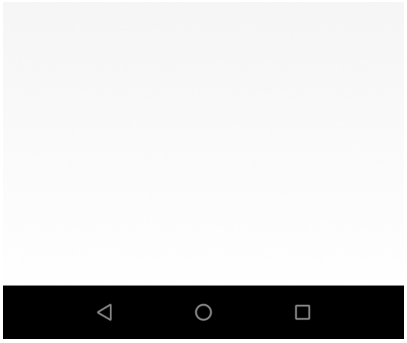
While in the configuration select the **I2C Bus 0** option. This will open a screen that shows the IMU.



Step 2

Press the Add button to add the Color Sensor to this bus. Select "REV Color Sensor V3" from the drop-down menu and name the device.





Step 3

When you have finished configuring the sensor hit **Done**. The app will return to the previous screen.

Applications

How do I2C sensors help a robot navigate the world around it? The answer to this question is a bit more diverse than it was for Analog or Digital sensors.

All three Color Sensors (V1-V3) sense color within a 2cm distance from the sensor. When mounted on the robot this can help in autonomous period tasks where robots have to decide between several different colored objects. Relic Recovery, Rover Ruckus, and Skystone all had autonomous tasks where the Color Sensor helped robots choose between randomized jewels, minerals, and stones!

While the Color Sensors have some proximity sensing capabilities, the 2m Distance Sensor is able to detect proximity with higher accuracy and reliability. When combined with odometry, the 2m Distance Sensor can help the robot navigate obstacles on the field during autonomous!

The IMU has a built-in accelerometer, gyroscope, and magnetometer. There are a multitude of applications for the IMU within autonomous op modes:

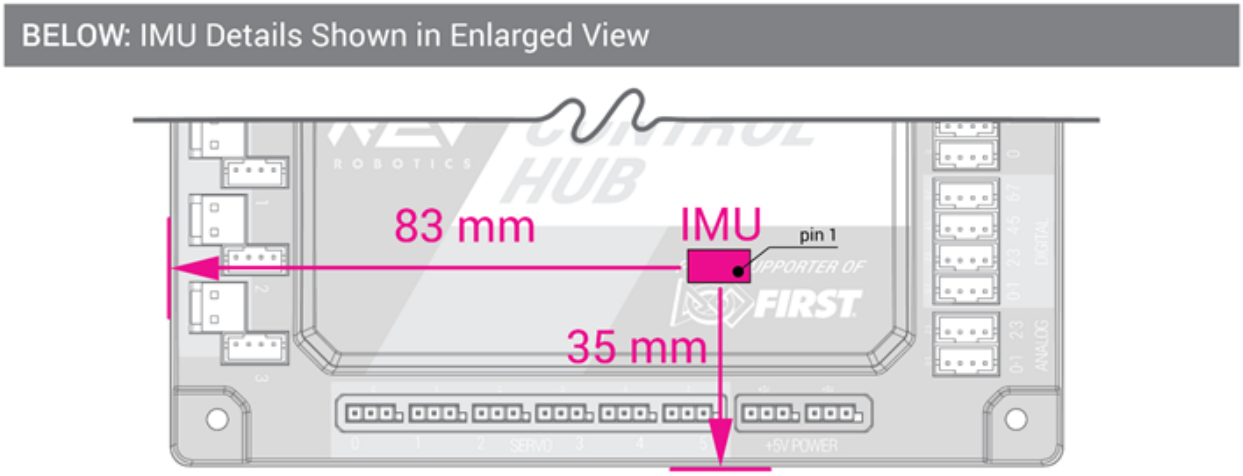
- Use the Gyroscope to drive in the straight lines and turn during autonomous
- Use the Accelerometer in conjunction with the gyroscope to avoid drift and give an approximation of position/travel
- Use the IMU with motor encoders to track and determine robot placement on a field

For more information on the I2C sensor specifications, coding examples, and more; click one of the links below to head to the sensor datasheets

Color Sensor V3 (REV-31-1557)
Color Sensor V2 (REV-31-1537)
Color Sensor V1 (REV-31-1154)

IMU

IMU Basics



Every REV Robotics Control Hub (REV-31-1595), and Expansion Hubs (REV-31-1153) purchased before December 2021, have a built in 9-axis IMU, or inertial measurement unit. The IMU incorporates three sensors: a 3-axis accelerometer, a 3-axis gyroscope, and a 3-axis geomagnetic sensor. The **accelerometer** measures the affect of forces on acceleration along the three axes. The **gyroscope** measures the rotational location of the the Hubs along the axes. The **geomagnetic** sensor (or **magnetometer**) uses the Earth's magnetic field to find orientation.

> ⓘ **Expansion Hubs purchased AFTER December 2021 no longer include an internal IMU**

> ⚠ The accuracy of the magnetometer within the IMU is affected by proximity to surrounding magnetic fields.

The data considered and used by the IMU includes: rotation along each axis, forces of acceleration along each axis, and magnitude of acceleration. The rotational measurements for the gyroscope play an important part in the use of the gyroscope for positioning and location of the robot.

- **Heading** is the measure of rotation along the z-axis. If the Hub is laying flat on a table, the z-axis points upwards through the front plate of the Hub.
- **Pitch** is the measure of rotation along the x-axis. The x-axis is the axis that runs from the bottom of the hub, near the servo ports, to the top of the hub ,where the USB ports are.
- **Roll** is the measure along the y-axis. The y-axis is the axis that runs from the sensor ports on the right to

the motor ports on the left.

The orientation of the hub plays a large part into which measurement will be used to determine the orientation of the robot as it moves.

Product Specifications

- I2C Address: 0x28
- Port: 0

How to Use?

Application

There are a multitude of applications for the IMU within autonomous op modes:

- Use the Gyroscope to drive in the straight lines and turn during autonomous
- Use the Accelrometer in conjunction with the gyroscope to avoid drift and give an approximation of position/travel
- Use the IMU with motor encoders to track and determine robot placement on a field

Configuration

Since some IMUs are already installed with in the Control or Expansion Hub the main concerns are Hub position and configuration. **The orientation of the Hub affects which axis is getting feedback.**

The IMU always exists on I2C Bus 0.

> ⓘ *To learn more on how to configure the IMU check out the I2C introduction page.*

Adding an IMU to your Expansion Hub

> ⓘ **If your Expansion Hub was purchased *BEFORE* December 2021, it already has an internal IMU installed and you do not need to follow these steps.**

Compatible External IMUs

There are a few options that will work for giving your Expansion Hub Gyro/IMU function.

1. Integrating Gyro with our Logic Level Converter and Sensor Cable Adapter - This is directly supported in the FTC Programing environment but is just a single-axis gyro, not a full IMU.
2. navX2 Sensor Bundle - This is currently out of stock, but is also supported in the FTC programming environment. Code examples are listed on AndyMark's page, and this product includes the correct cables to use within FTC.
3. Adafruit 9-DOF Absolute Orientation IMU - This is the same IMU as in the Control Hub, but will require you to either create an adapter cable or solder a cut sensor cable to the board. Plugging this in and configuring the IMU on I2C port zero will allow you to use and program the same as an internal IMU.

Encoders

What is an Encoder?

An encoder is anything (device, software, person) that *converts* information from one format into another. Some examples of encoding include:

- A transducer, like a speaker, which *converts* an electrical signal into sound waves
- Software which *encodes* an audio file into an mp3 to decrease file size
- A stenographer (court reporter) takes courtroom dialog and *converts* it into a written record

This section is about rotary encoders which are electro-mechanical devices which convert the angular position of a shaft, like on a motor, to an electronic signal. These signals can be fed into a microcontroller, which controls all robot functions, and then used to provide real world data to make better programming decisions.

There are two main types of encoders: absolute and relative.

Absolute encoders return the actual angle of the rotation (e.g. 30°). Absolute encoders maintain position information if the power is removed, and position data is immediately available when power is reapplied with no rotation needed to read the current angle. The relationship between the encoder value and the motor shaft is set when assembled and will always stay the same. Commonly these encoders use a specially printed pattern disk which are read and converted to a known angle. Generally, absolute encoders are easier to use when programming, but they are more complicated to manufacture so are larger, or more expensive.

Relative encoders, which are also referred to as **incremental encoders**, provide information about the motion of the shaft (e.g. forward at 5 RPM), and only provide data while the shaft is rotating. One way to remember this is that relative encoders return information on the incremental change of the motor output shaft. Relative encoders only provide pulses as the motor turns, and interpreting these pulses into useful information must be done externally. A relative encoder does not know what position it is in at start-up, but it is possible to create a calibration program that must be run at every start-up to obtain reference point to calculate an angle from.

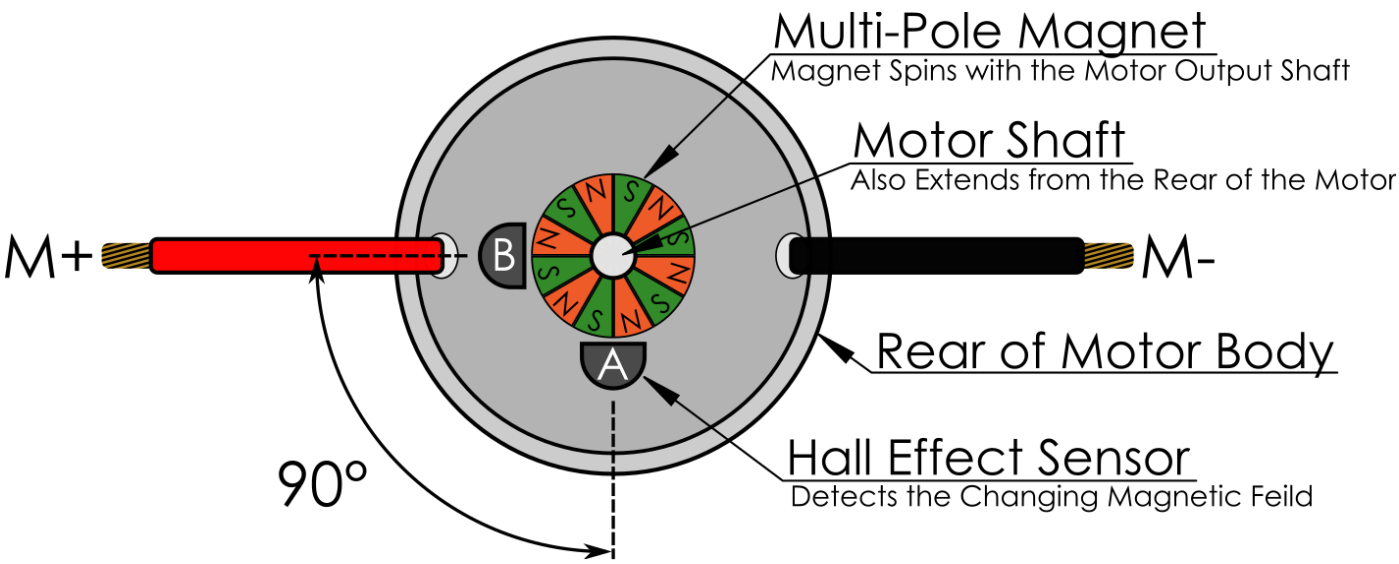
Encoders measure a real world change (shaft rotation) and convert it to an electrical signal. Two common ways to do this are using optical or magnetic feedback:

Optical encoders have a disk with a series of either slots or a reflective pattern around the outside which is attached to the motor shaft. A light shines on or through the disk where the light can pass through or reflect onto a photodiode (device which produces an electric signal when light shines on it). These sensors can be very light and compact, but can be very sensitive to anything that might interfere with the light reaching the photodiode. Finger prints on a reflective disk, or dust from a dirty environment can interfere.

Magnetic encoders have a magnet attached to the shaft of a motor and use Hall effect sensors to detect the changing magnetic field as the shaft rotates. Magnetic encoders are able to operate in harsh or dirty environments.

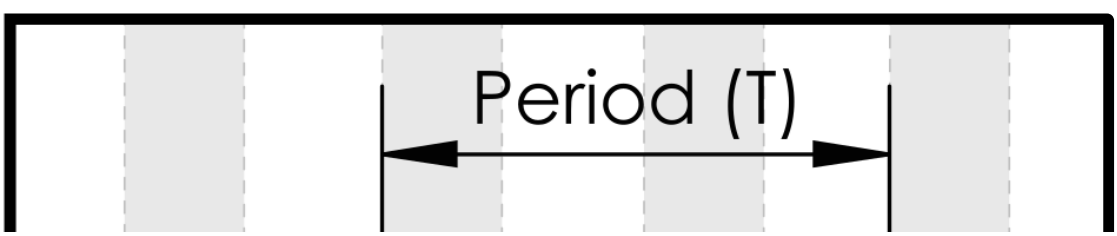
Magnetic Quadrature Encoders

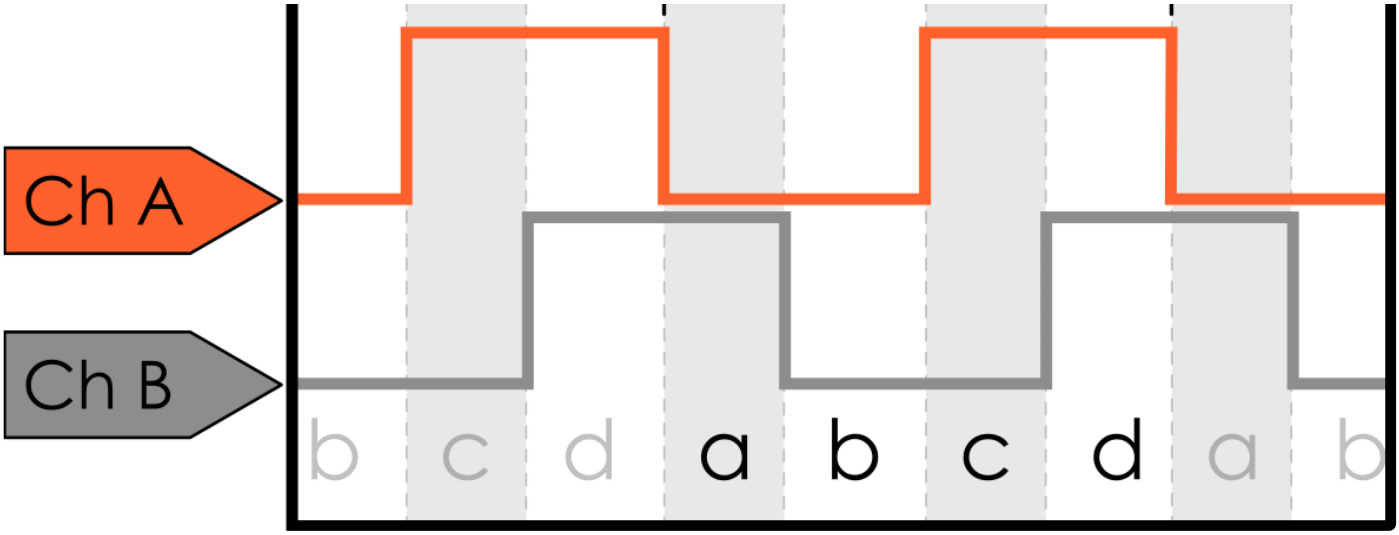
A 12 pole magnetic quadrature encoder is installed on the rear of both the HD Hex Motor and Core Hex Motor. The output shaft of the motor extends from the rear of the motor case and a multi-pole permanent magnet is attached to the shaft. There are two Hall effect sensors, marked ‘A’ and ‘B’, mounted next to the magnet at 90° to each other. As each of the 12 poles passes across one of the Hall effect sensors, it creates a change in the magnetic field causing the sensor to generate a measurable voltage signal.



Typical Encoder Configuration Installed on the Rear of a Motor

Quadrature encoders are a specific type of relative encoder that have four different output states. If the root *quad-*, means four, but there are only two sensors in this encoder, where does the name come from? The output from the two Hall effect sensors are called “Channel A” and Channel B” respectively; an example of the output is shown below. In a single period (T), defined as the duration of time of one complete cycle in a repeating pattern, the timing diagram has four distinct states (see a, b, c, and d below), hence a quadrature encoder.





Clockwise Quadrature Encoder Output Timing Diagram

The offset from Channel A to Channel B is because the sensors are offset from each other by 90°. As the motor rotates one sensor will see the change before the other. When the motor shaft rotates clockwise (CW), Channel A will lead (the edge will rise before) Channel B. When the motor spins counter clockwise (CCW) Channel A will lag (rise after) Channel B. If there was only one sensor it would still be possible to measure the number of rotations, but not to detect the direction of the motor.

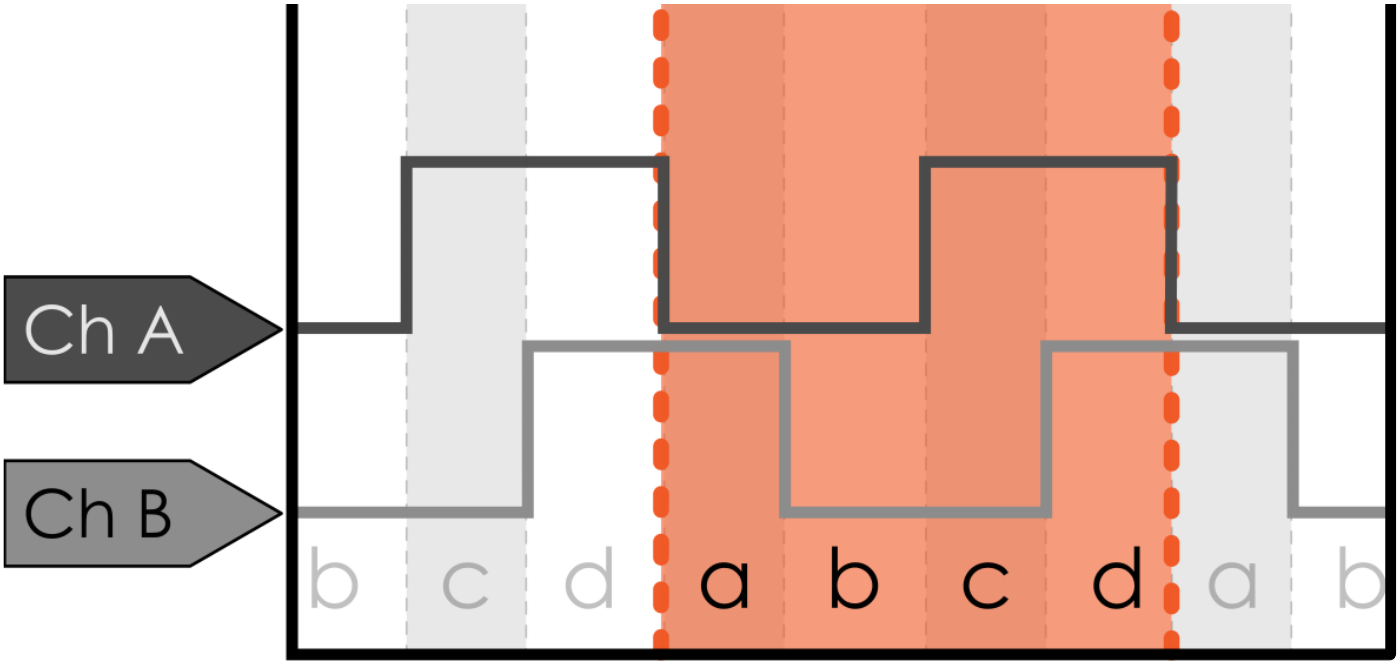
> ⓘ On HD Hex and Core Hex motors Channel A leads Channel B when positive voltage is applied to the M+ terminal. However, there are times when this will not hold true in real life. Different reduction gearboxes, or physically swapping the Channel A and Channel B encoder wires into the controller, can reverse the relationship between the channels. Keep this in mind when programming and troubleshooting your robot.

When the encoder is being read by a microcontroller, the two signals are compared to produce a count up pulse or count down pulse. These pulses are counted as steps forward (CW) or backwards (CCW). Using the specifications for the encoder being used, a count can be converted to degrees. This information can be used to drive a robot arm to a specific angle, or tell a robot to drive a certain distance. Both the Control Hub and Expansion Hub communicate to a microcontroller through the encoder ports.

Encoder Technical Specification Definitions

> ⓘ There is some conflicting terminology difference between encoder suppliers. This document defines one of the most commonly agreed upon set of terms, however be aware that when comparing between encoder specifications from different vendor's terms may vary in meaning.

Every time the output goes through all four distinct combinations of output signals, it's called a **cycle** (see a, b, c, and d below). Encoders have a different **cycles-pre-revolution(CPR)** based on the number of poles on the magnet used. The CPR is how many cycles are generated for one complete revolution of the encoder shaft.



Encoder Cycle

An example output from one complete rotation of a 14 CPR encoder is shown in in the figure below. A 14 CPR rotation encoder may also be referred to as having 14 rises on channel A. Encoders are mounted to the motor shaft, not the gearbox output shaft, so for a motor with a reduction gearbox attached this is less than one full output shaft rotation.

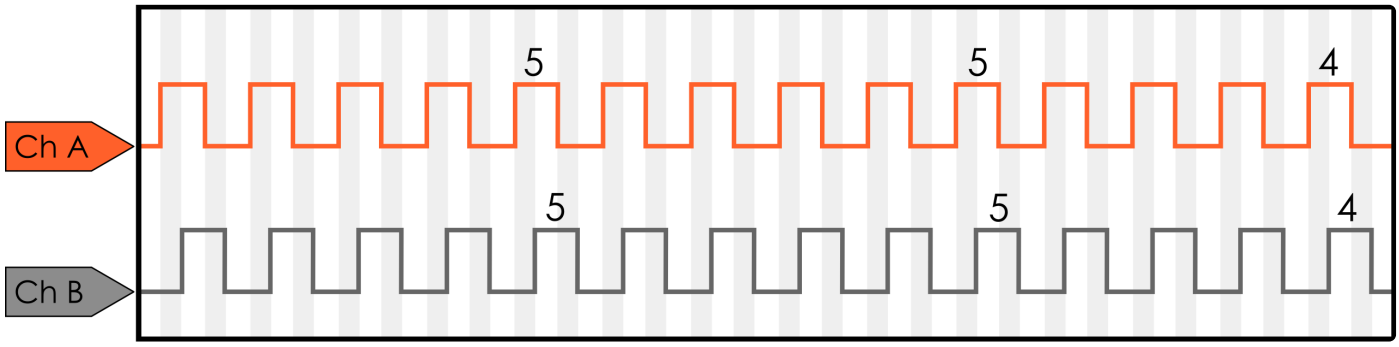


Figure 4: Encoder Output for one Revolution of a 14 CPR Encoder

One reason to use CPR to define an encoder, rather than the commonly used PPR (Pulses per Revolution) is when the encoder signal is decoded by the microcontroller it is possible to do 1x, 2x, or 4x decoding. For 1x decoding the micro controller would only “count” the rising signal on a single channel, while for 4x decoding each rising or falling edge for both channels is measured as a “count.” Although 4x decoding is one of the most common methods, because it’s based on how the electronics decode the signal from the encoder, and not on the encoder hardware itself, it’s not an ideal method of defining the encoder hardware specifications.

If we assume 4x decoding when each cycle is interpreted, the microcontroller can read the four distinct outputs (a, b, c, and d) as individual steps. So for each CPR, the controller can read four counts/ticks. To calculate the number of counts per rotation of the encoder shaft:

$$COUNTSPERROTATION_{(}oftheencodershaft) = CPR(Cyclesperrotation) \times 4$$

The actual cycles per rotation of the output shaft of the motor is depending on the gearbox that's attached

$$COUNTSPERROTATION_{(}oftheoutputshaft) = CPR(Cyclesperrotation) \times 4 \times Reduction$$

This can be calculated into the degrees per count. Assuming no additional reduction is added to the final stage of the motor output (i.e. direct drive) the number of degrees per count is calculated as:

$$DEGREESPERCOUNT = 360°/COUNTSPERROTATION_{(}oftheoutputshaft)$$

REV Motor Encoders

REV Robotics HD Hex Motors (REV-41-1291) and the Core Hex Motors (REV-41-1300) come with a magnetic quadrature encoder already installed and an appropriate cable for connecting the encoder output to the REV Robotics Control Hub (REV-31-1595) or Expansion Hub (REV-31-1153). See Table 1 and Table 2 for relevant encoder details.

Core Hex Motor (REV-41-1300) Encoder Specifications

Core Hex Motor (REV-41-1300) Reduction	72:1
Free Speed (RPM)	125
Cycles per Rotation of the Encoder Shaft	4 (1 Rise of Channel A)
Counts per Rotation of the Output Shaft	288 (72 Rises of Channel A)

HD Hex Motor (REV-41-1291) Encoder Specifications

HD Hex Motor Reduction	Bare Motor	40:1	20:1
Free Speed (RPM)	6000	150	300
Cycles per Rotation of the Encoder Shaft	28 (7 Rises of Channel A)	28 (7 Rises of Channel A)	28 (7 Rises of Chann A)
Counts per Rotation of the Output Shaft	28 (7 Rises of Channel A)	1120 (280 Rises of Channel A)	560 (140 Rises of Channel A)

Through Bore Encoder

The REV Through Bore Encoder (REV-11-1271) is specifically designed with the end user in mind, allowing teams to place the sensor in the locations closest to the rotation that they wish to measure. This rotary sensor measures both relative and absolute position through its ABI quadrature output and its absolute position pulse output.

> ⓘ The FTC Control System (Control Hub and Expansion Hub) only supports incremental encoder input through the motor encoder ports at this time. Absolute pulse input is not supported.

Included with the Through Bore Encoder is a 5mm Hex insert and a 4-Pin JST PH to 6-pin JST PH connector. The 6-pin connector is plugged into the Through Bore Encoder with the 4-pin connector plugging into either the Control Hub (REV-31-1595) and Expansion Hub (REV-31-1153) Encoder Port. Both the A and B channels of the encoder are used.

When using the 5mm Hex insert, press the insert into the 1/2” Hex hole before attaching to a mechanism. If you are having difficulty pressing the insert into the encoder, try flipping the insert over and press it in. There is a slight taper in the insert, so it is recommended to press the insert with the smaller end first. When removing, it is recommended to push the insert out in the reverse order (larger end first).

For more information on the Through Bore Encoder check the Through Bore Encoder Datasheet.

Using 3rd Party Sensors

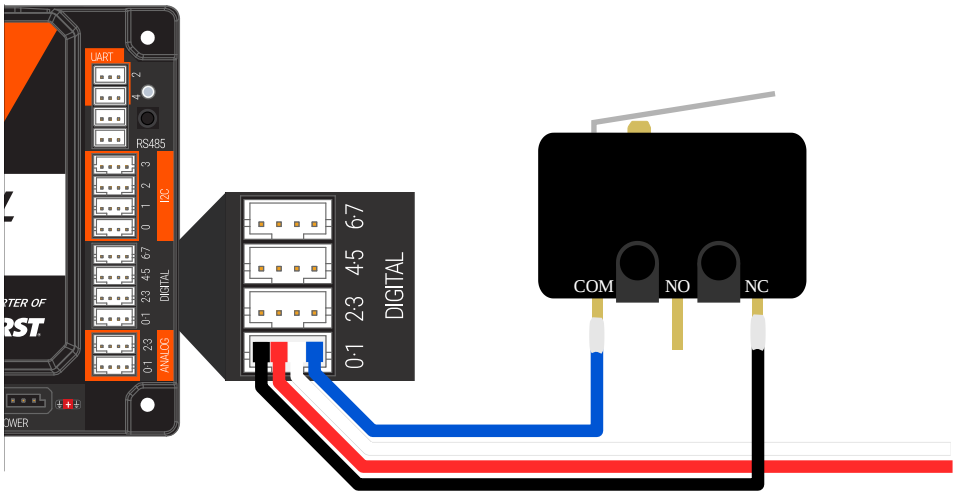
The Control Hub (REV-31-1595) and Expansion Hub (REV-31-1153) are 3.3V logic level devices. Many 3rd party sensors, including ones that teams have previously purchased through vendors such as Modern Robotics, are 5V logic level devices. Many of these legacy sensors are used with the REV system by using a logic level converter. REV Robotics offers a Logic Level Converter (REV-31-1389) and an optional Sensor Adapter Cable (REV-31-1384) so teams can more easily use their legacy sensors with the REV Control System.

Wiring a Limit Switch or Micro Switch

Limit switches are common 3rd Party sensor type used with the REV Control System and require a custom wiring harness. Each of the digital inputs on the Control and Expansion Hub have a pull-up resistor making the digital inputs pulled "high" by default. Incorrect wiring of a limit switch to a digital input can create a conflict making the Control or Expansion Hub unresponsive.

The recommended wiring is to connect the signal wire (n, n+1) to the common pin (COM), the ground wire to the normally closed (NC) pin, and not connect to the normally open pin (NO) of the limit switch. With this wiring when the switch is in its normal state (not pressed), the switch is closed connecting the signal to ground (reporting `FALSE` in code). When pressed, the switch is open and disconnects the signal from ground (reporting `TRUE` in code).

> ⚠ The power wire and the unused signal wire will not be used in this set up process.



If you need the opposite behavior (`FALSE` for pressed, and `TRUE` for not pressed) switch the ground (black) wire to the NO position instead of NC. Alternatively changing the logic in code will have a similar effect.

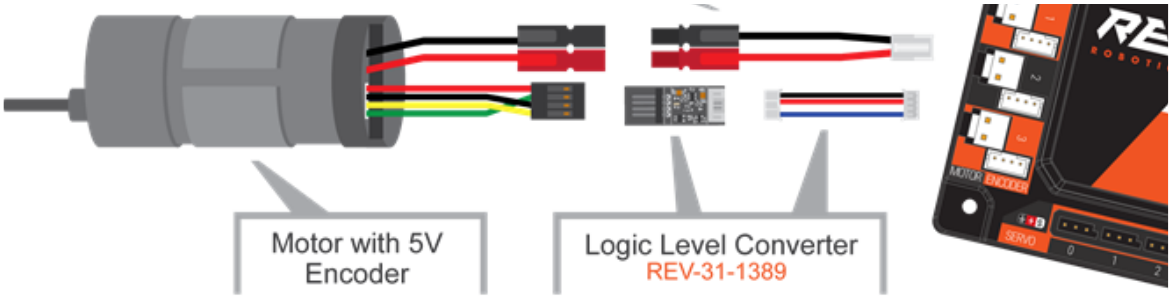
Logic Level Converter

The REV Robotics Logic Level Converter is a circuit board which generates a 5V output from the 3.3V input and uses a MOSFET on each signal line to create a bidirectional communication appropriate for a variety of digital signals include I2C communication. For more information on how bidirectional level shifting communication is accomplished, please reference the NXP Application Note AN10441.

The Logic Level Converter is only needed for the Digital and I2C senor ports on the Control or Expansion Hub when using a 5V device.

Connecting 5V Encoder

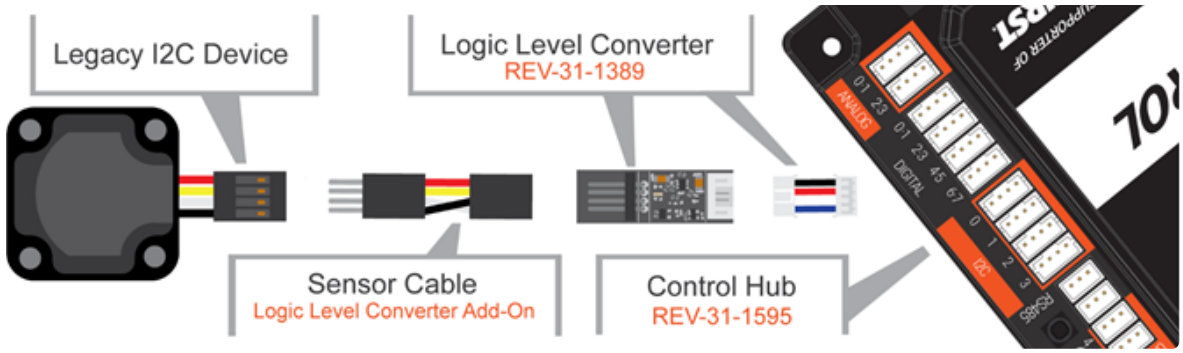
The Logic Level Converter (REV-31-1389) pinout directly matches the encoder cable pin out for FTC legal 3rd party motors. Encoder cables plug directly into the Logic Level Converter board and then the 4-pin JST PH Cable (REV-31-1407), which is included with the Logic Level Converter, is plugged into the appropriate Control Hub (REV-31-1595) Encoder Port. Motors which are terminated with Anderson Power Pole style connectors use the JST VH to Anderson Power Pole Style (REV-31-1381) cable to connect to the motor output port on the Control Hub.



> ⓘ All REV Robotics Motors work directly with the REV Control and Expansion Hubs. No Logic Level Converter is needed for REV Motors.

Connecting a 5V Sensor

A variety of 5V sensors are usable with the Control Hub (REV-31-1595) when used with a Logic Level Converter (REV-31-1389). For some Modern Robotics I2C sensors a Logic Level Converter, and a change in wiring to match the pinout of the Control Hub are needed. Teams can either purchase a Sensor Cable as an add on to the Logic Level Converter Kit which will cross over the correct wires, or they can carefully rearrange the pin order on the sensor cable. If using the Sensor Cable, connect the sensor to the Control Hub as shown below. It is recommended to zip tie the connection between the sensor and the sensor cable to prevent accidental disconnects. See the Sensor Compatibility Chart for more information on hardware required for other sensors.

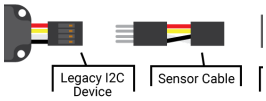
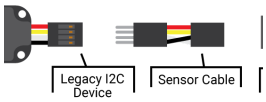
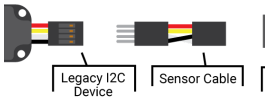
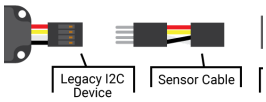
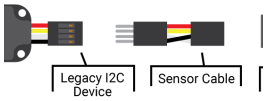
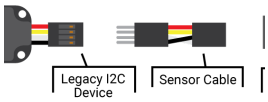
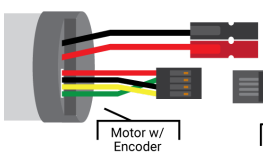


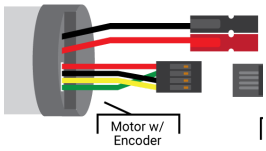
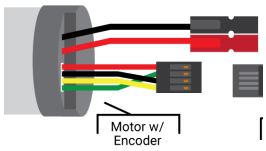
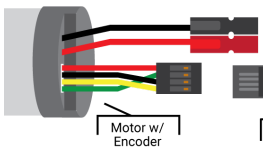
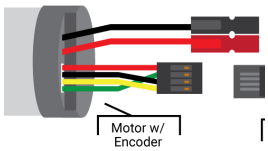
Sensor Compatibility Chart

To determine if your existing sensors are used with the Control Hub (REV-31-1595) or Expansion Hub (REV-31-1153) along with additional hardware needed, see the table below.

Sensor Compatibility Table

Sensor	Type	Compatible	Adapters Needed
Absolute Orientation IMU Fusion Breakout - BNO055	I2C	Yes	3.3V Compatible Custom Wiring

2472 Adafruit			Harness Needed
RGB Color Sensor with IR filter and White LED - TCS34725 1334 AdaFruit	I2C	Yes	3.3V Compatible Custom Wiring Harness Needed
Color Sensor 45-2018 Modern Robotics	I2C	Yes	
Compass 45-2003 Modern Robotics	I2C	Yes	
Integrating Gyro 45-2005 Modern Robotics	I2C	Yes	
IR Locator 360 45-2009 Modern Robotics	I2C	Yes	
IR Seeker V3 45-2017 Modern Robotics	I2C	Yes	
Ranger Sensor 45-2008 Modern Robotics	I2C	Yes	
NeveRest Motor AM-3461, AM-3102, AM-2964a, AM-3103, AM-3104 AndyMark	Quad Encoder	Yes	

HD Hex Motor REV-41-1301 REV Robotics	Quad Encoder	Yes	Directly Compatible No Custom Adapters Needed
Core Hex Motor REV-41-1301 REV Robotics	Quad Encoder	Yes	Directly Compatible No Custom Adapters Needed
12v 4mm Motor Kit 50-0119 MATRIX	Quad Encoder	Yes	
12v 6mm Motor Kit 50-0120 MATRIX	Quad Encoder	Yes	
Standard Motor Kit 50-0001 MATRIX	Quad Encoder	Yes	
Max Motor Shaft Encoder Kit W38000 Tetrix	Quad Encoder	Yes	
Limit Switch 45-2401 Modern Robotics	Digital	Yes	No Adapter Needed Custom Wiring Harness Required.
Rate Gyro 45-2004 Modern Robotics	Analog	No	Not Officially Supported
Optical Distance Sensor 45-2006 Modern Robotics	Analog	No	Not Officially Supported
Touch Sensor			No Adapter Needed

45-2007 Modern Robotics	Analog	Yes	Custom Wiring Harness Required
Light Sensor 45-2015 Modern Robotics	Analog	No	Not Officially Supported
Magnetic Sensor 45-2020 Modern Robotics	Analog	No	Not Officially Supported

Useful Links

Legacy Documentation

Configuring Your Android Devices

When using Android Phones as your Robot Controller and Driver Station devices, there are several steps you need to take in order to get the phones up and running. This section will go through the process of installing a Driver Station and Robot Controller application onto a phone using the REV Hardware Client, as well as the process for renaming your Wi-Fi direct network.

> ⓘ For information on how to pair a configured Android phone with a Control Hub please see our Driver Station Pairing to Control Hub article in the Getting Started with Control Hub section.

Installing the Driver Station Application

Android Developer Options

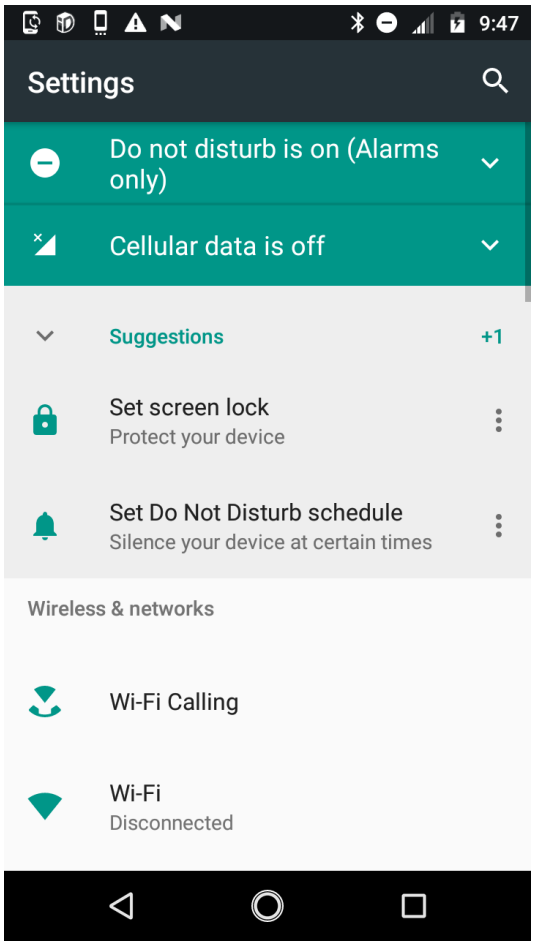
In order to install the Driver Station Application onto and Android phone, the phone's developer settings and USB debugging options need to be turned on.

The developer options on Android Devices are hidden within the phone as a default. Different phone manufactures have different ways of accessing the developer options. However, once the developer options are available in the phone's settings, the steps for activating USB debugging and development settings are similar.

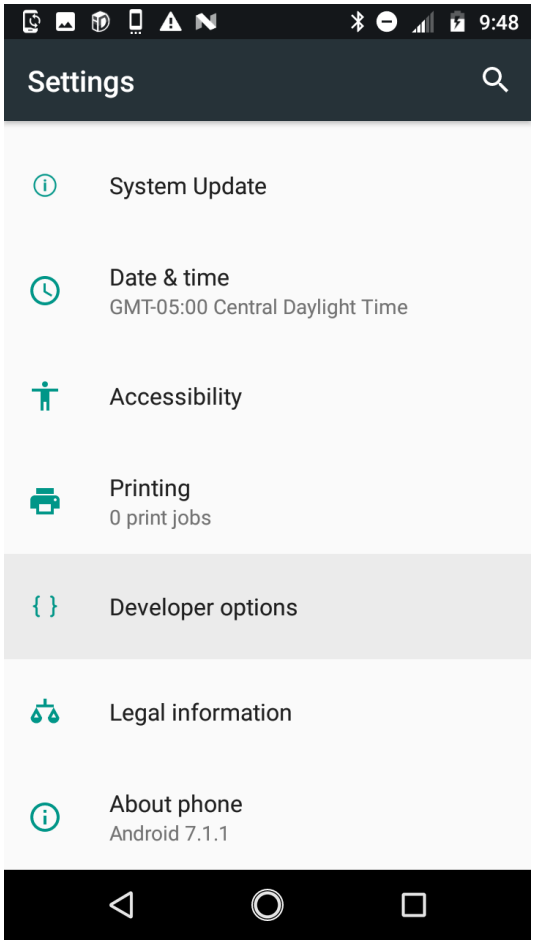
> ⚠ Before moving forward it is advised to look up where the developer options on your Android

Device are located. For Motorola users, the Motorola Support Page has information on how to unlock the developer options.

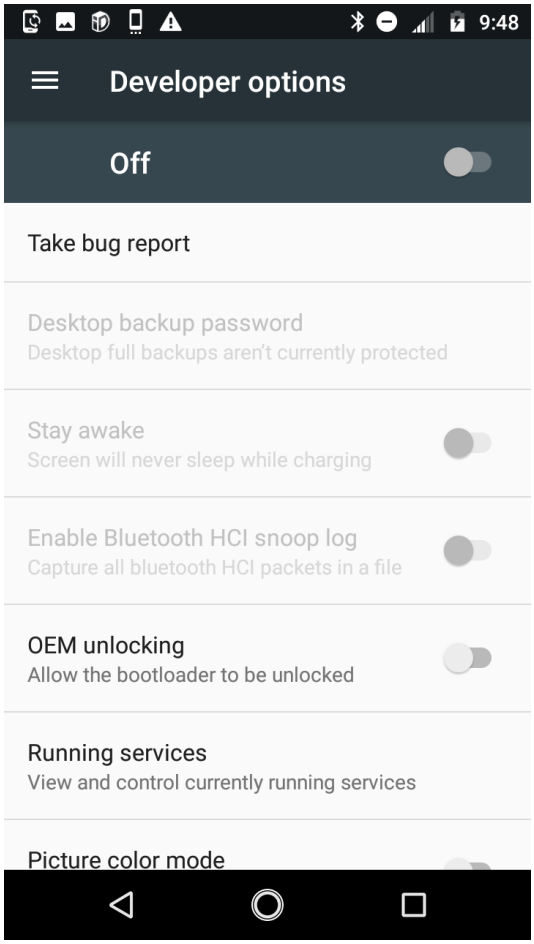
Open the Android Devices settings



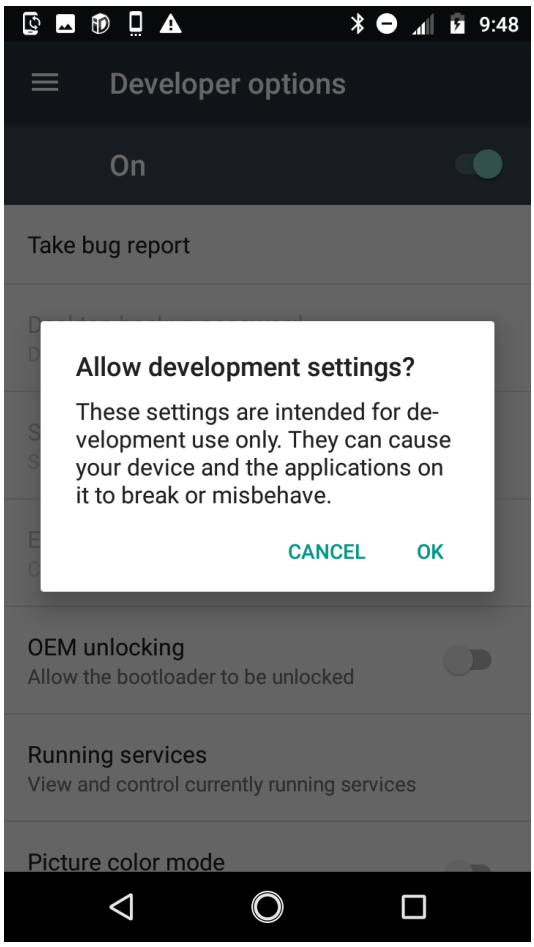
Scroll to the bottom of the settings, where the unlocked developer options are available. Open the developer options



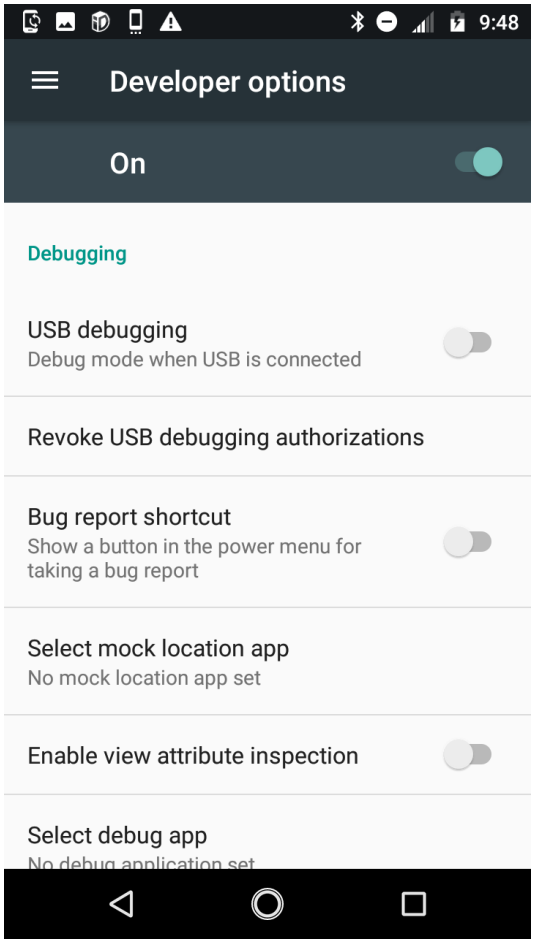
At the top of the developer options page is an on/off switch. Turn the developer options on.



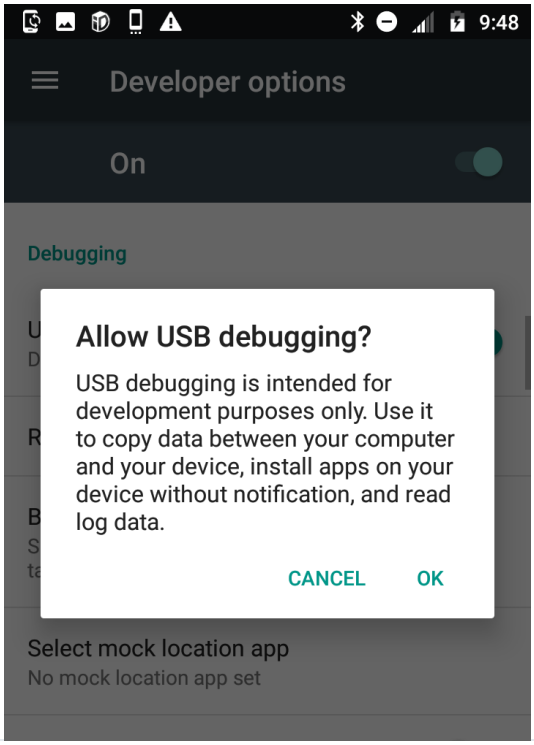
The device will open a confirmation message. Select 'OK.'



Scroll through the developer options until you find the Debugging section. Turn USB Debugging on.



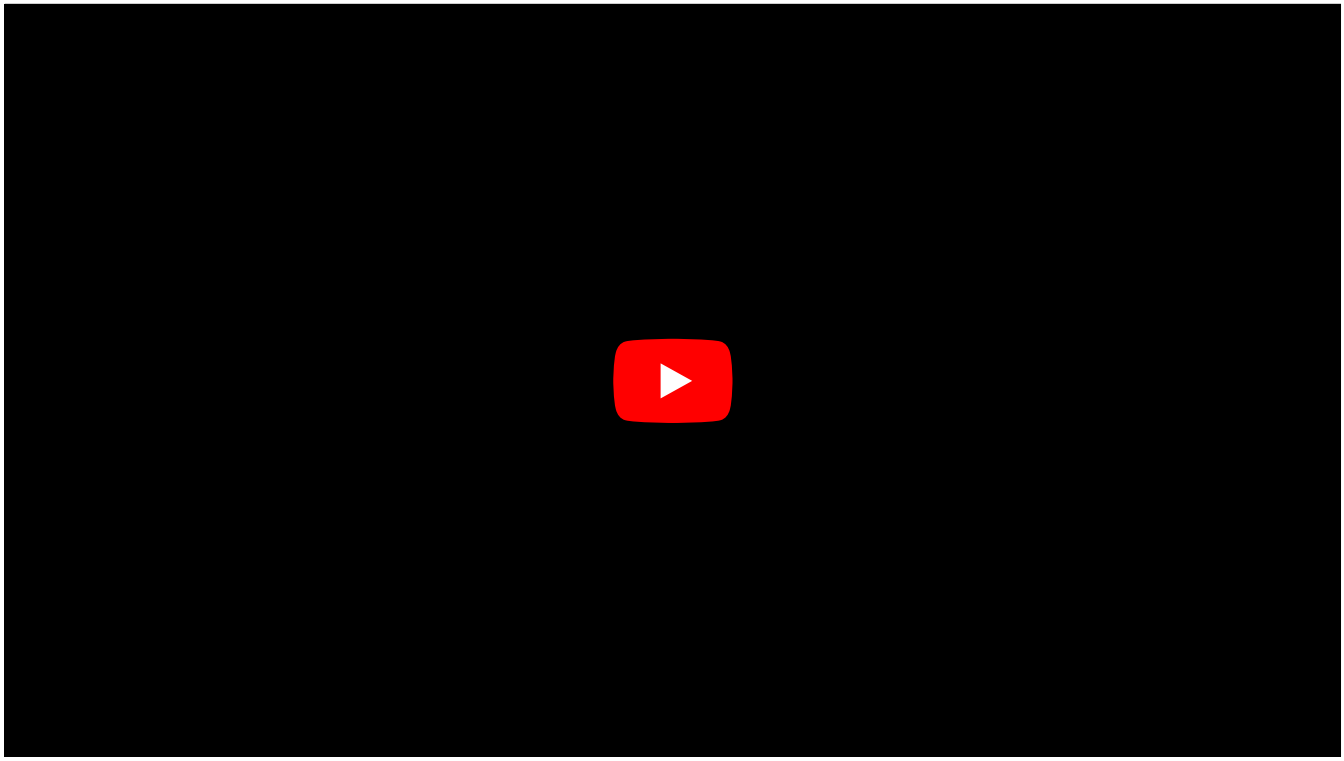
Another confirmation message will appear, click 'OK.'



USB debugging is now on! You can move on to the steps for installing the application.

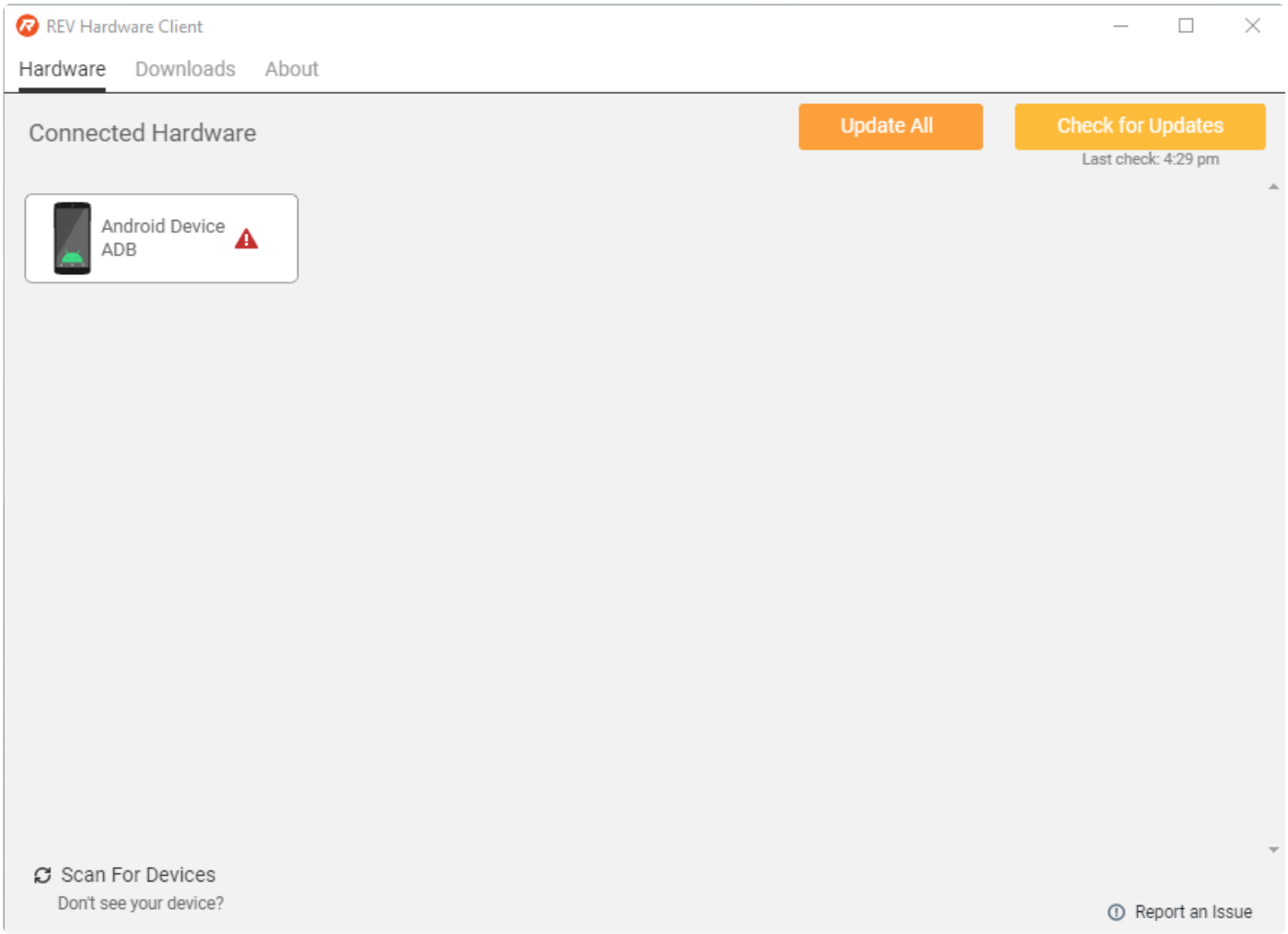
Driver Station Application

> The following steps will go through how to install the Driver Station Application via the REV Hardware Client. It is possible to install the application via the app store or via the FTC GitHub repository as well.

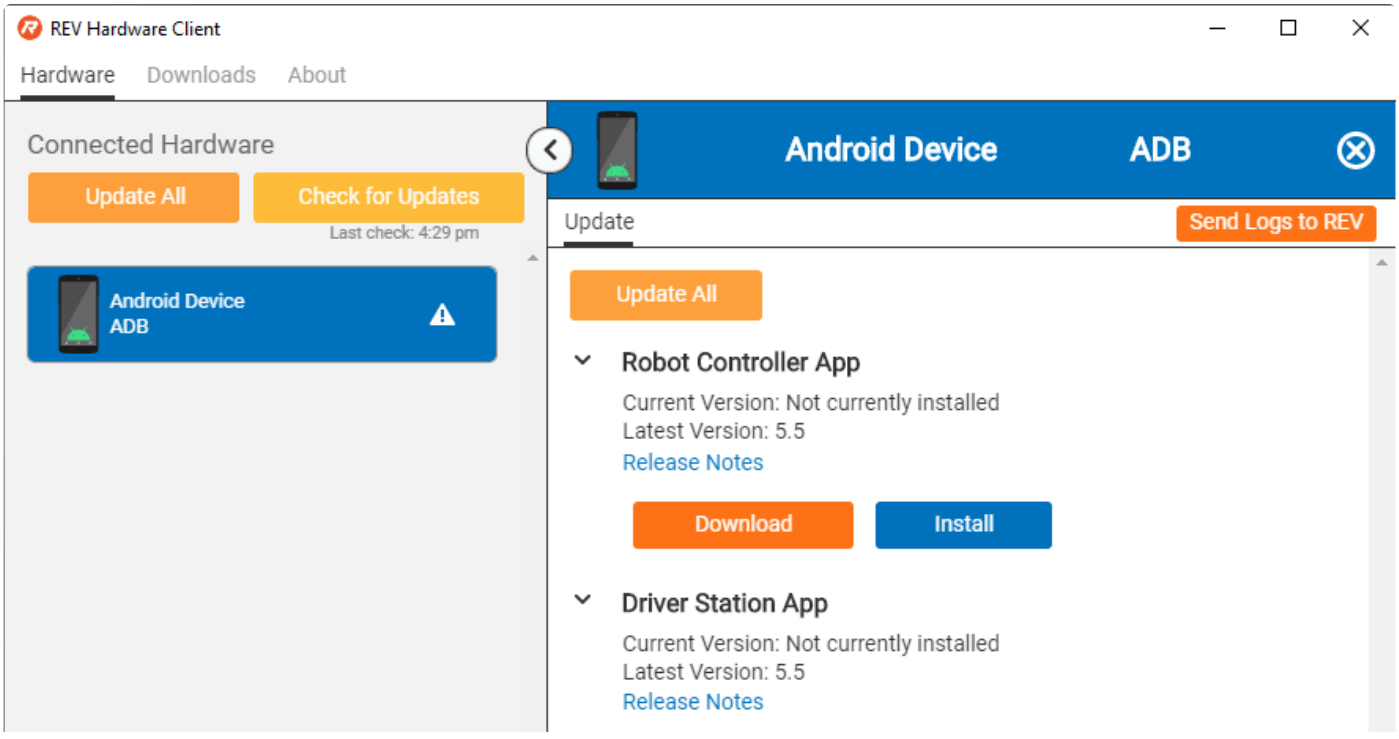


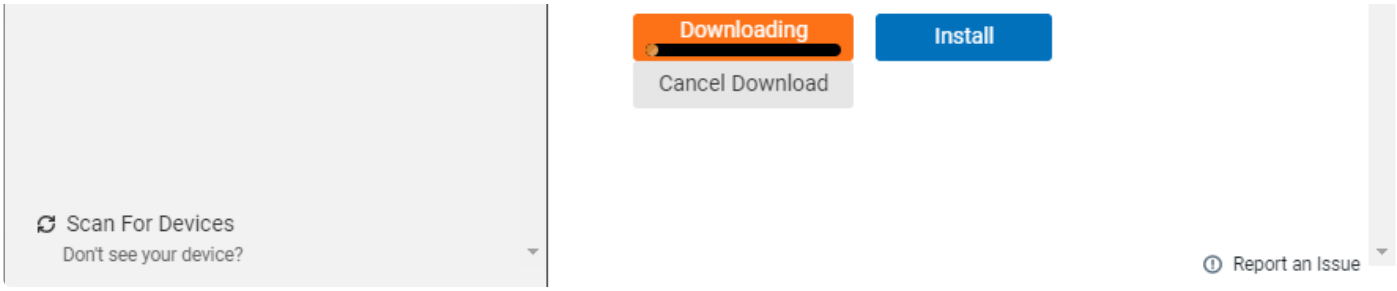
Connect the Android Device to a PC with the REV Hardware Client installed.

Startup the REV Hardware Client. Once the Android Device is fully connected it will show up on the front page of the UI under the **Hardware Tab**. Select the Android Device.

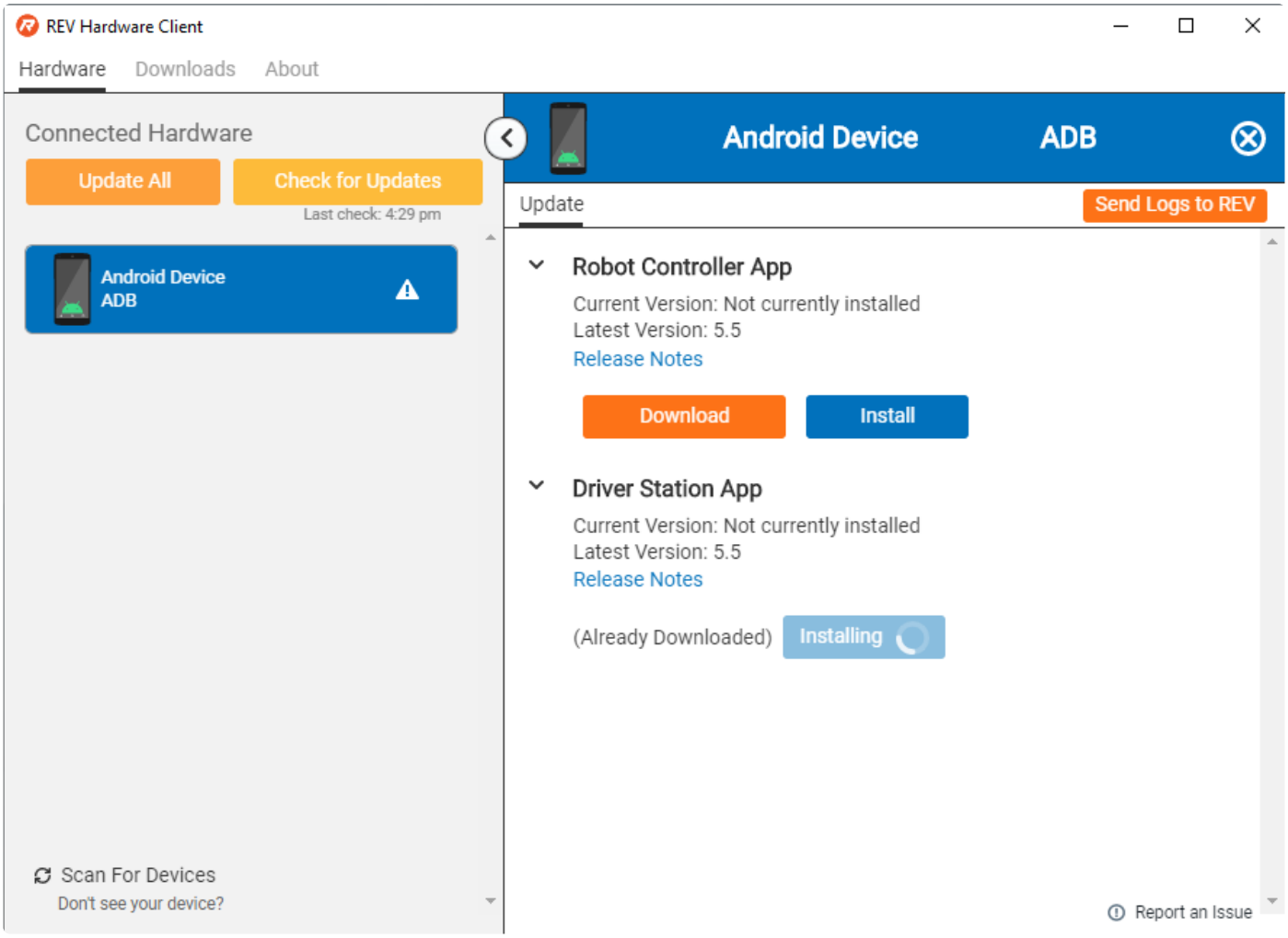


After selecting the Connected Hardware the Update tab will pop up. Under **Driver Station App** select Download.

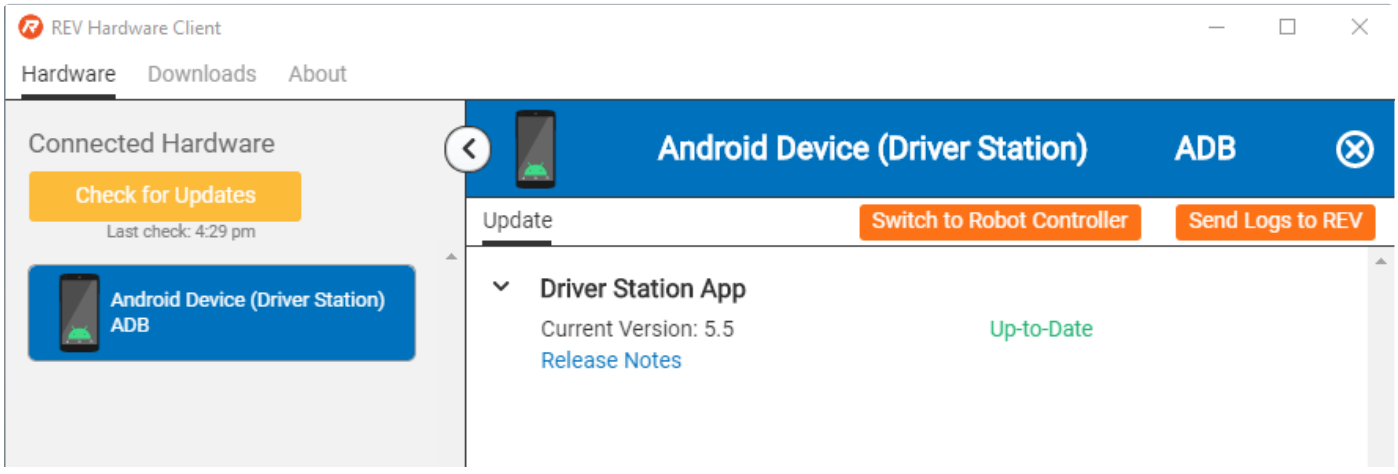




Once the Driver Station App has downloaded, select Install.



When the application installation has completed the status for the Driver Station App will change to "Up-to-Date."



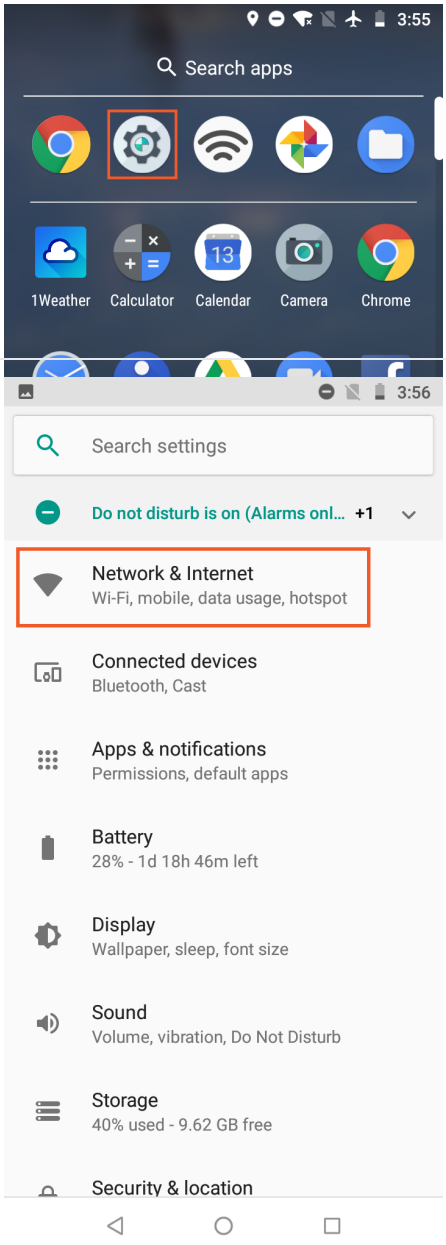
Renaming Your Smartphone

Part of the process for configuring your Android Device is changing the Wi-Fi Direct network. The intent of this process is to give your Robot Controller and Driver Station phones an identifiable and unique network name. This is a general best practice when working with networks, but is also a requirement for FIRST programs.

> FIRST has specific naming convention requirements for Robot Controllers and Driver Stations. Please check your programs game manual for more information on what you need to name your devices.

> Before moving forward it is advised to look up where the Wi-Fi direct options on your Android Device are located. This guide goes over where to make this change on the Moto E5.

Locate settings in the application list for your Android Device. Select the settings application



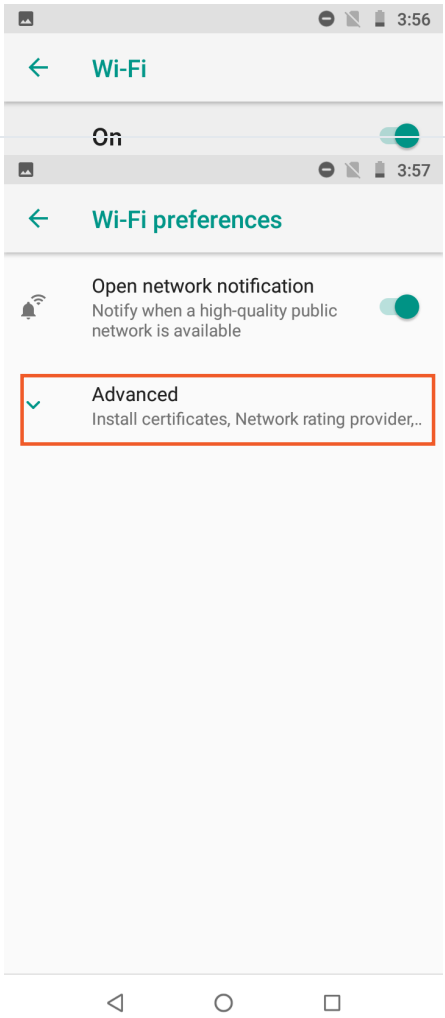
In the settings application, look for the **Wi-Fi** or **Network & Internet settings** and select it.

Note: the naming convention for the network settings will vary depending on device model and manufacturer

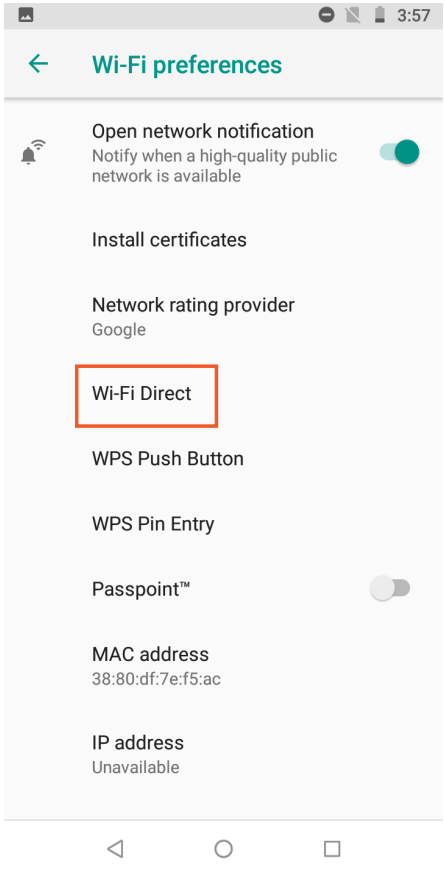
In the network settings on Moto E5, scroll to the bottom and look for **Wi-Fi preferences**. Select Wi-Fi preferences.

Note: on other phone models Wi-Fi Direct settings will likley be found in a different place. Please look up the Wi-Fi direct information for your phone model.

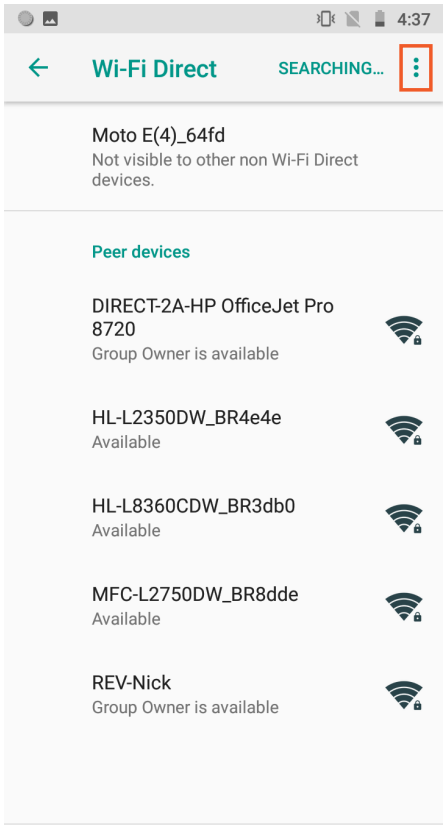
In **Wi-Fi preferences** select **Advanced**.



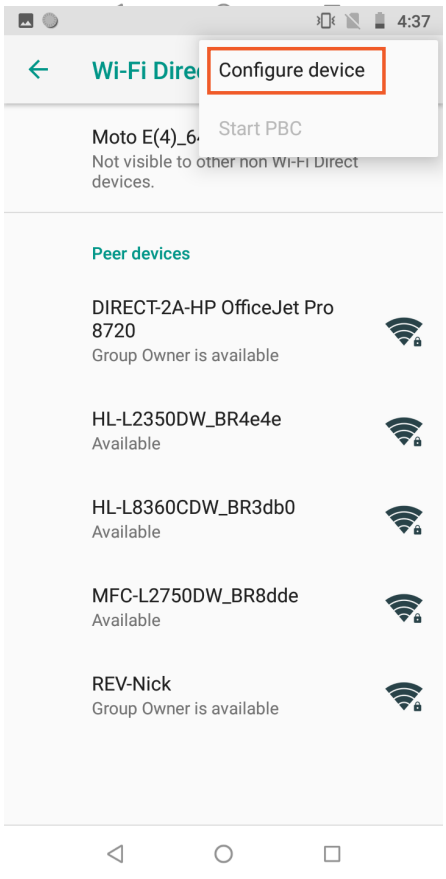
Select **Wi-Fi Direct**.



In the **Wi-Fi Direct** settings select the three vertical dots in the upper right hand corner.



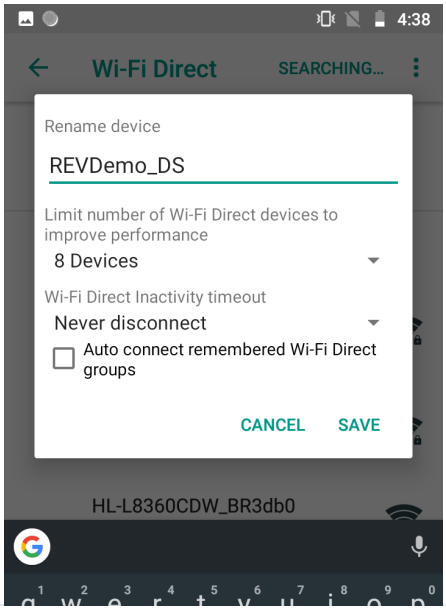
Select **Configure device**.



Change the name of your device to something unique and identifiable. For this example the

device has been renamed to REVDemo_DS. It is also good to check the **Wi-Fi Direct Inactivity timeout** and confirm it is set to **Never disconnect**. Hit 'save' to confirm your changes.

Note: If you are competing in robotics competitions you may need to follow a Wi-Fi Direct naming convention set by the competition rules. Check any relative documentation to confirm that you are following the correct naming convention.



Expansion Hub with Android Device Robot Controller

After receiving the Expansion Hub it is advised to unbox the device, power the Expansion Hub on, and start the configuration process. Below are the required materials to run through the initial bring up of the Expansion Hub and links to the different steps of the process.

Required Materials

- Expansion Hub (REV-31-1153)
- 12v Slim Battery (REV-31-1302)
- Properly Configured Driver Station (DS)
- Properly Configured Robot Controller (RC)
- Etpark Wired Controller for PS4 (REV-39-1865)
- USB A Female to Micro USB (REV-31-1807)

Optional Additional Materials needed to Connect an Expansion Hub:

- Expansion Hub (REV-31-1153)
- XT30 Extension Cable (REV-31-1392)
- JST PH 3-pin Communication Cable (REV-31-1417)

Driver Station and Robot Controller Pairing

When you first receive your Expansion Hub, you will have to install the Driver Station and Robot Controller Applications and pair (link) your Driver Station (Android Device) to your Robot Controller. The following

sections of the page will walk through how to install the applications and how to connect the Driver Station to the Robot Controller's Network.

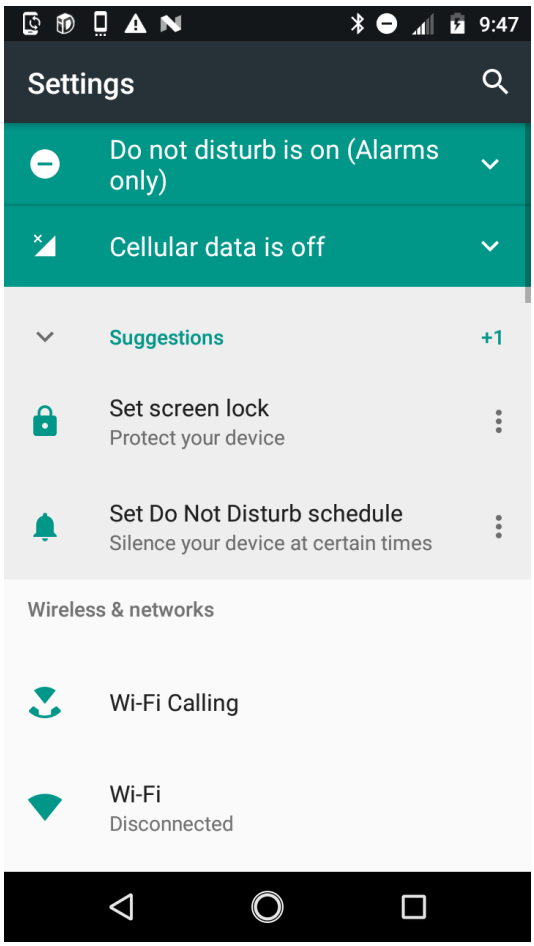
Install Applications

Android Developer Options

In order to install the Driver Station Application or Robot Controller Application onto and Android phone, the phone's developer settings and USB debugging options need to be turned on.

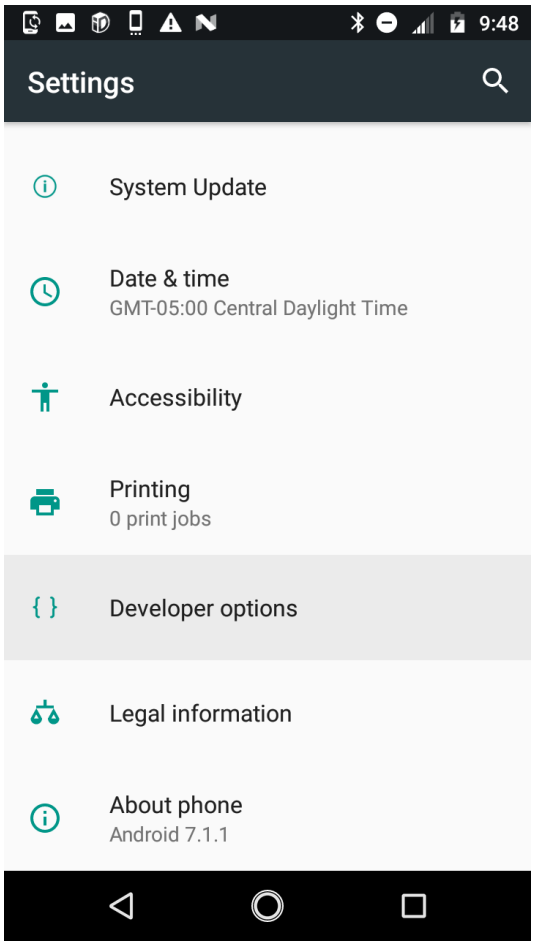
The developer options on Android Devices are hidden within the phone as a default. Different phone manufactures will have different ways of accessing the developer options. However, once the developer options are available in the phone's settings, the steps for activating USB debugging and development settings are similar.

> ⚠ Before moving forward it is advised to look up where the developer options on your Android Device are located. For Motorolla users, the Motorolla Support Page has information on how to unlock the developer options.

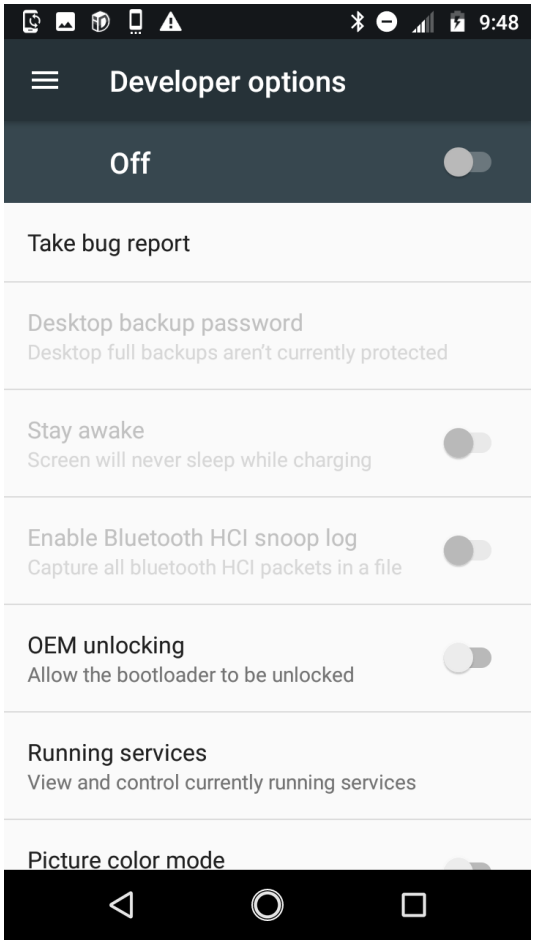


Open the Android Devices settings

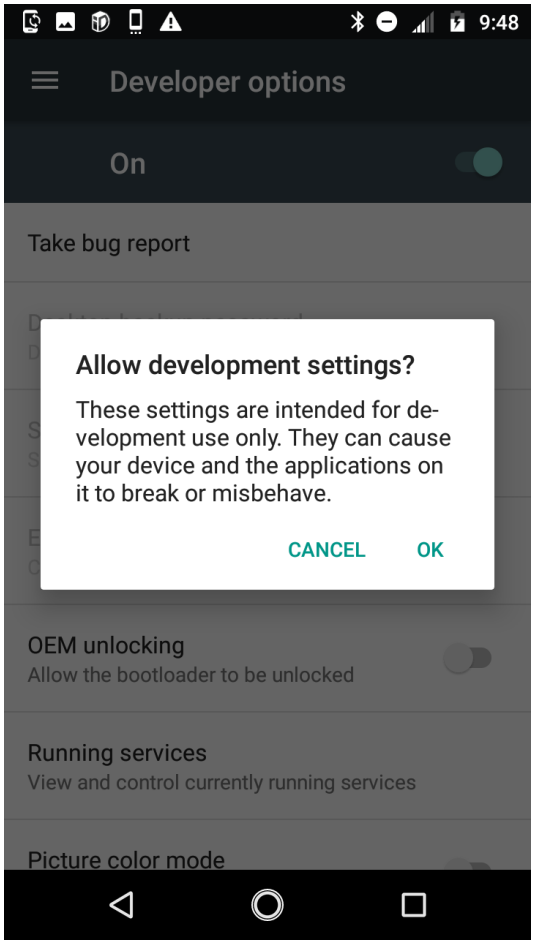
Scroll to the bottom of the settings, where the unlocked developer options are available. Open the developer options



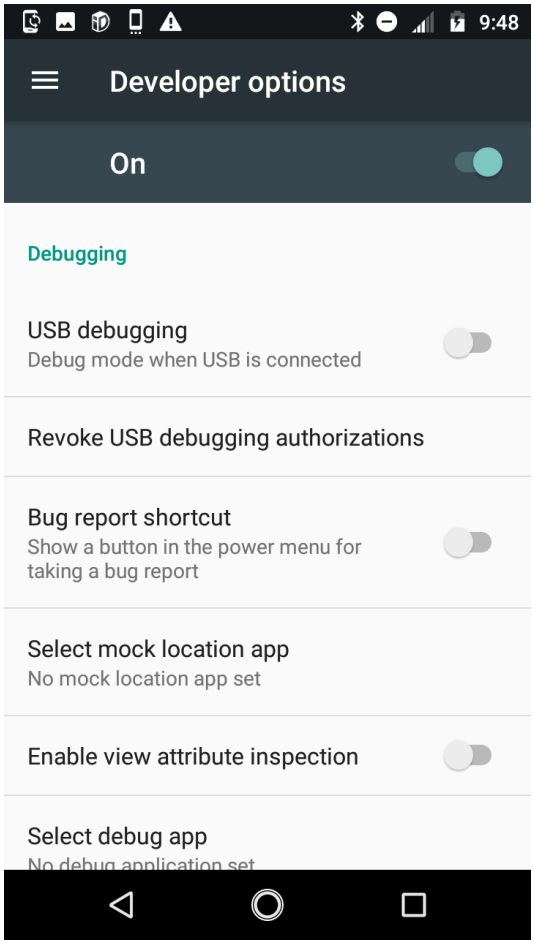
At the top of the developer options page is an on/off switch. Turn the developer options on.



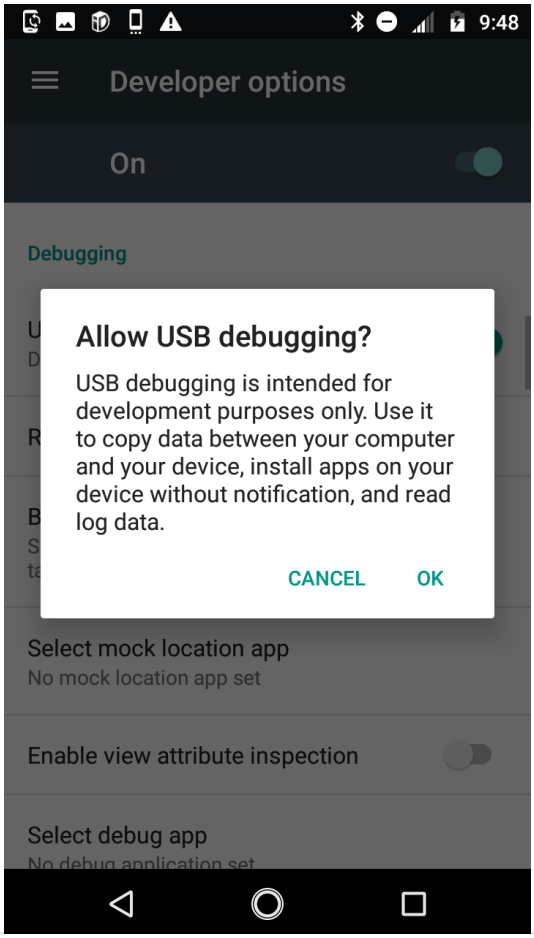
The device will open a confirmation message. Select 'OK.'



Scroll through the developer options until you find the Debugging section. Turn USB Debugging on.



Another confirmation message will appear, click 'OK.'



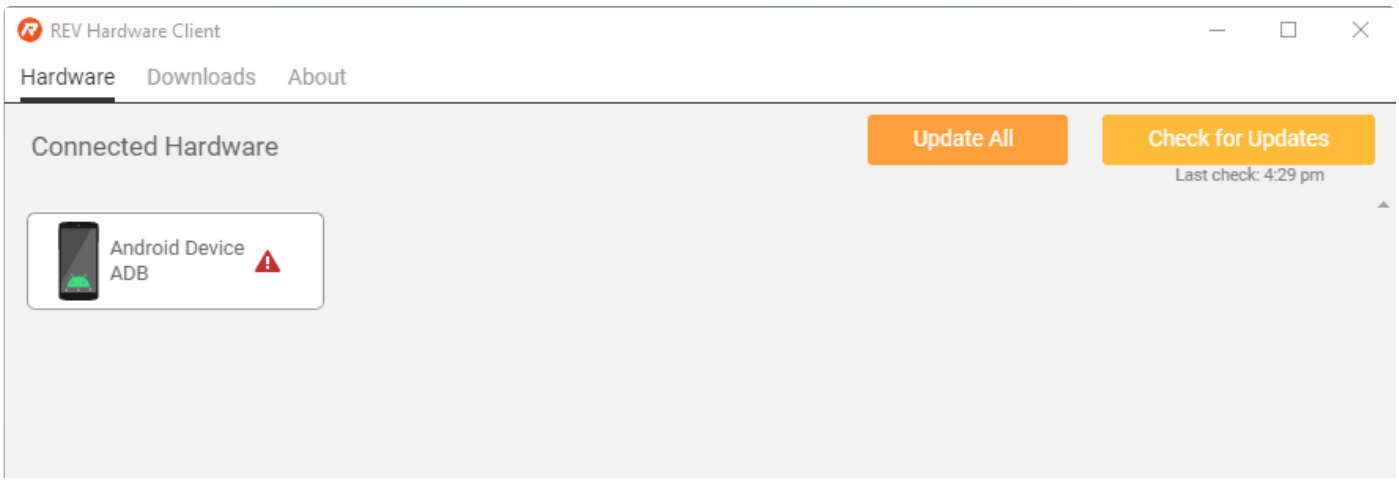
USB debugging is now on! You can move on to the steps for installing the application.

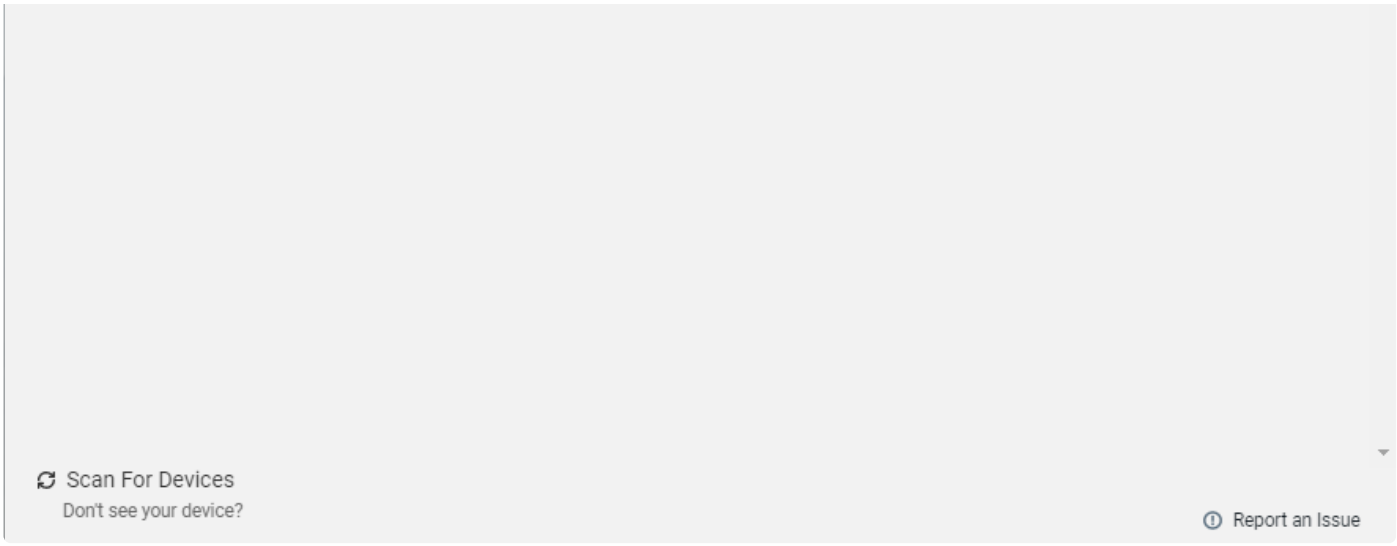
Driver Station Application

> The following steps will go through how to install the Driver Station Application via the REV Hardware Client. It is possible to install the application via the app store or via the FTC GitHub repository as well.

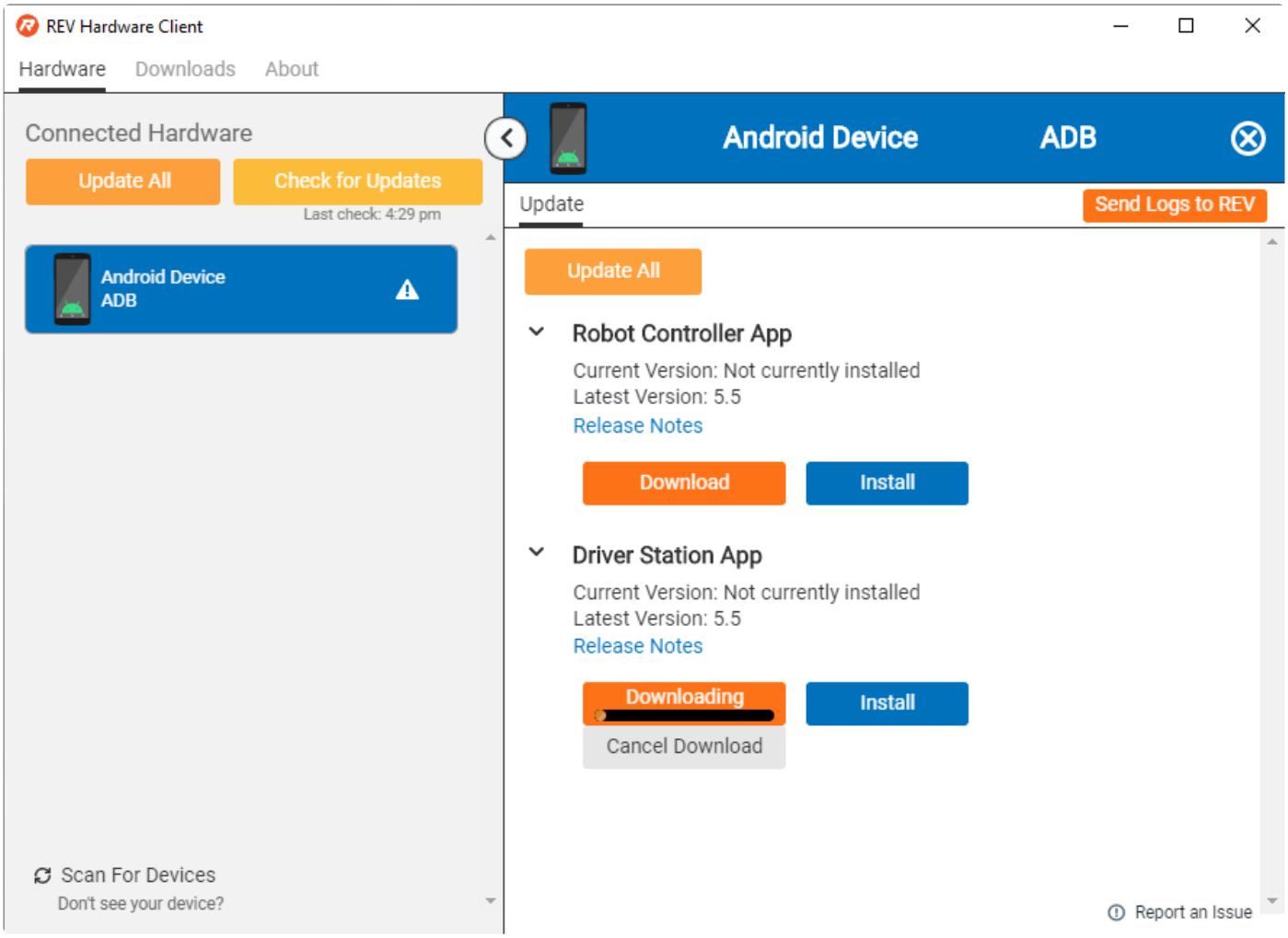
Connect the Android Device to a PC with the REV Hardware Client installed.

Startup the REV Hardware Client. Once the Android Device is fully connected it will show up on the front page of the UI under the **Hardware Tab**. Select the Android Device.

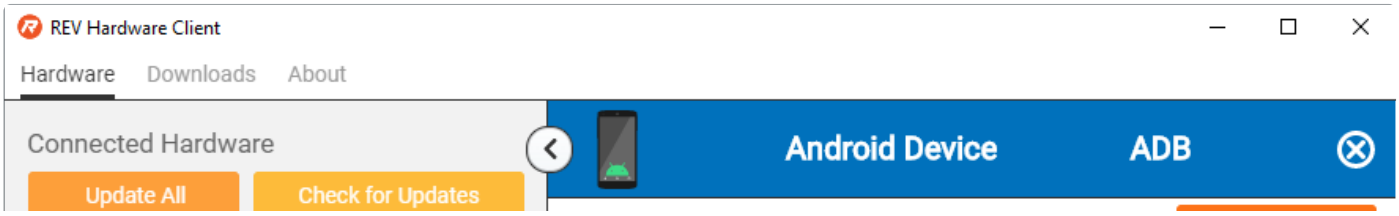


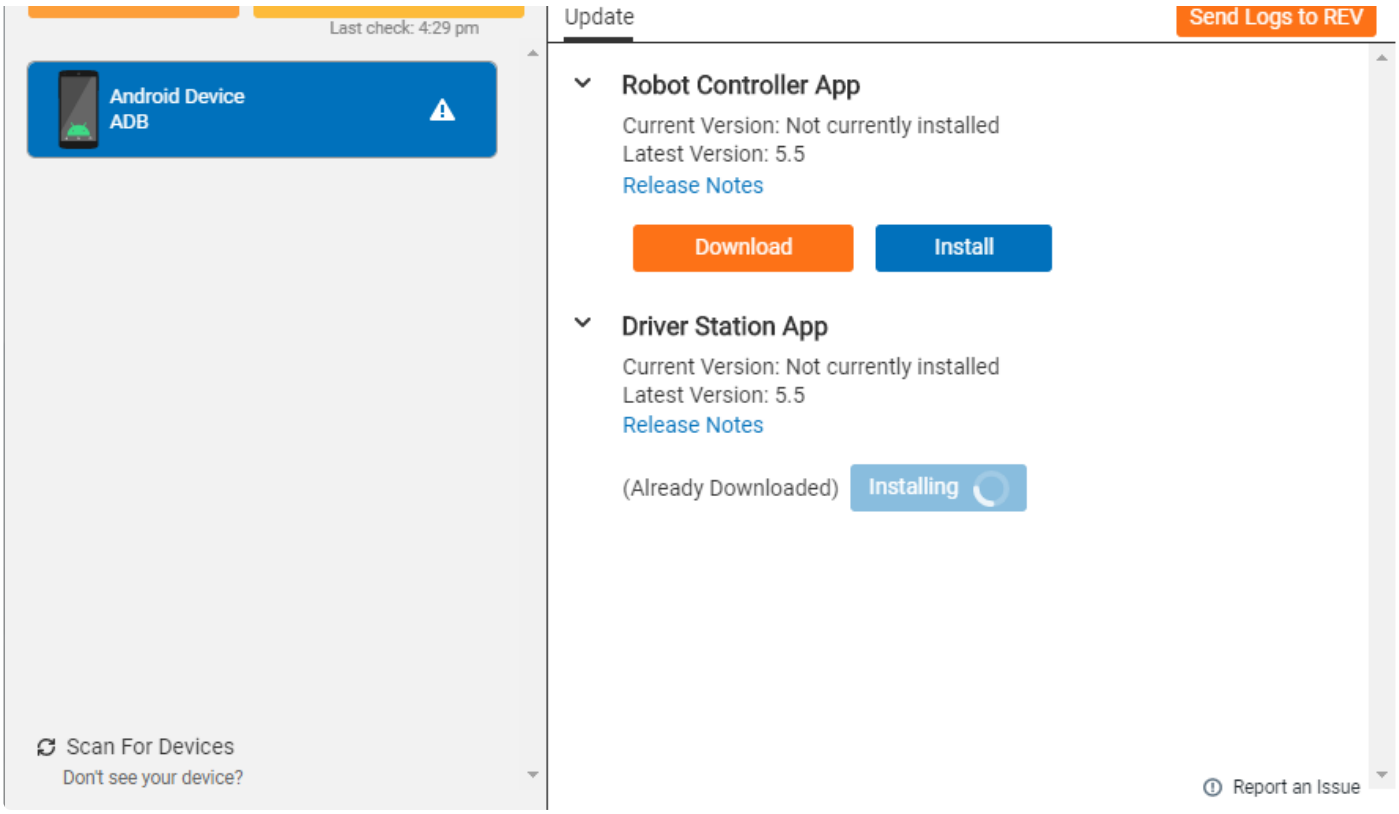


After selecting the Connected Hardware the Update tab will pop up. Under **Driver Station App** select Download.

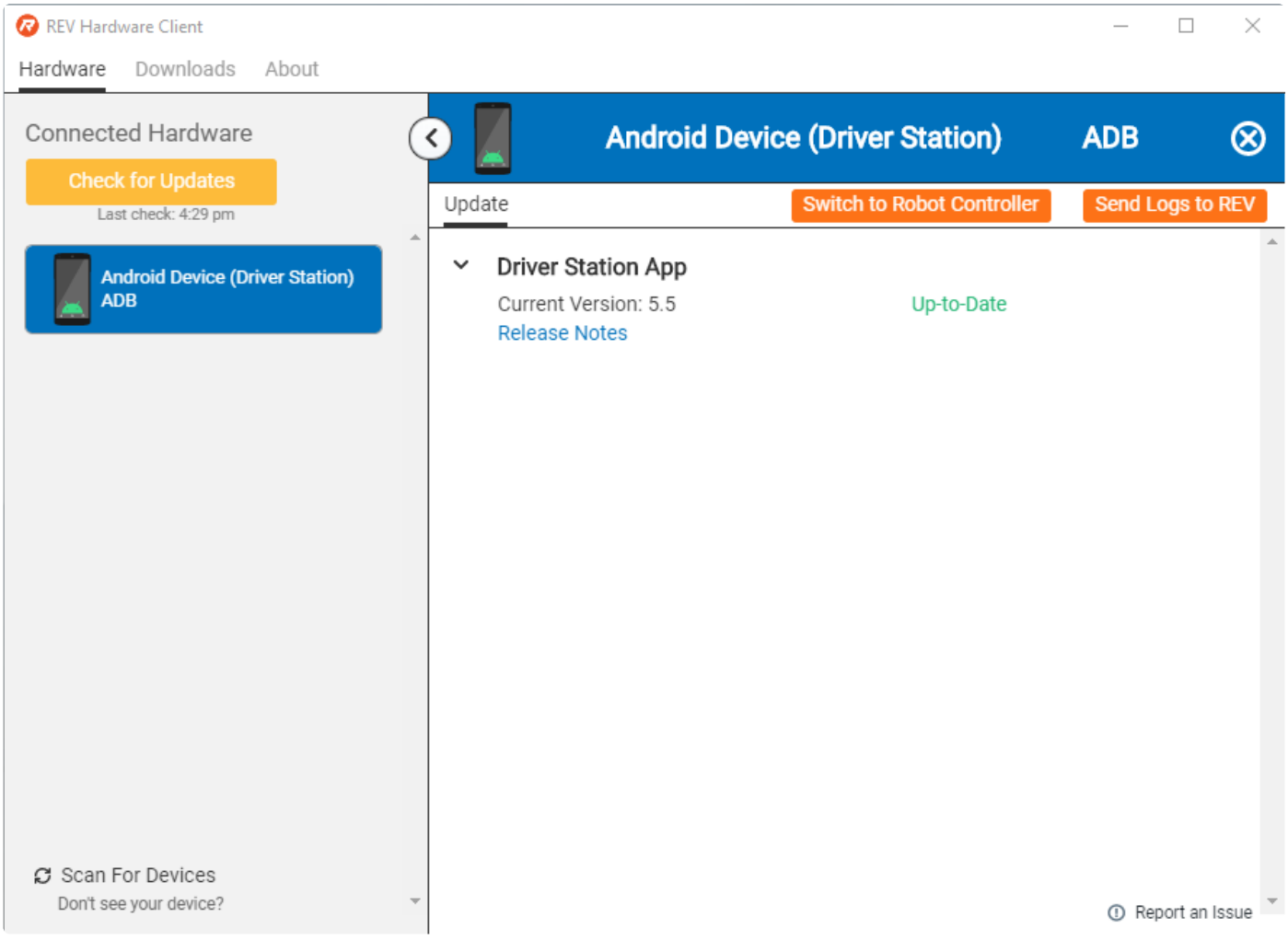


Once the Driver Station App has downloaded, select Install.





When the application installation has completed the status for the Robot Controller App will change to "Up-to-Date."

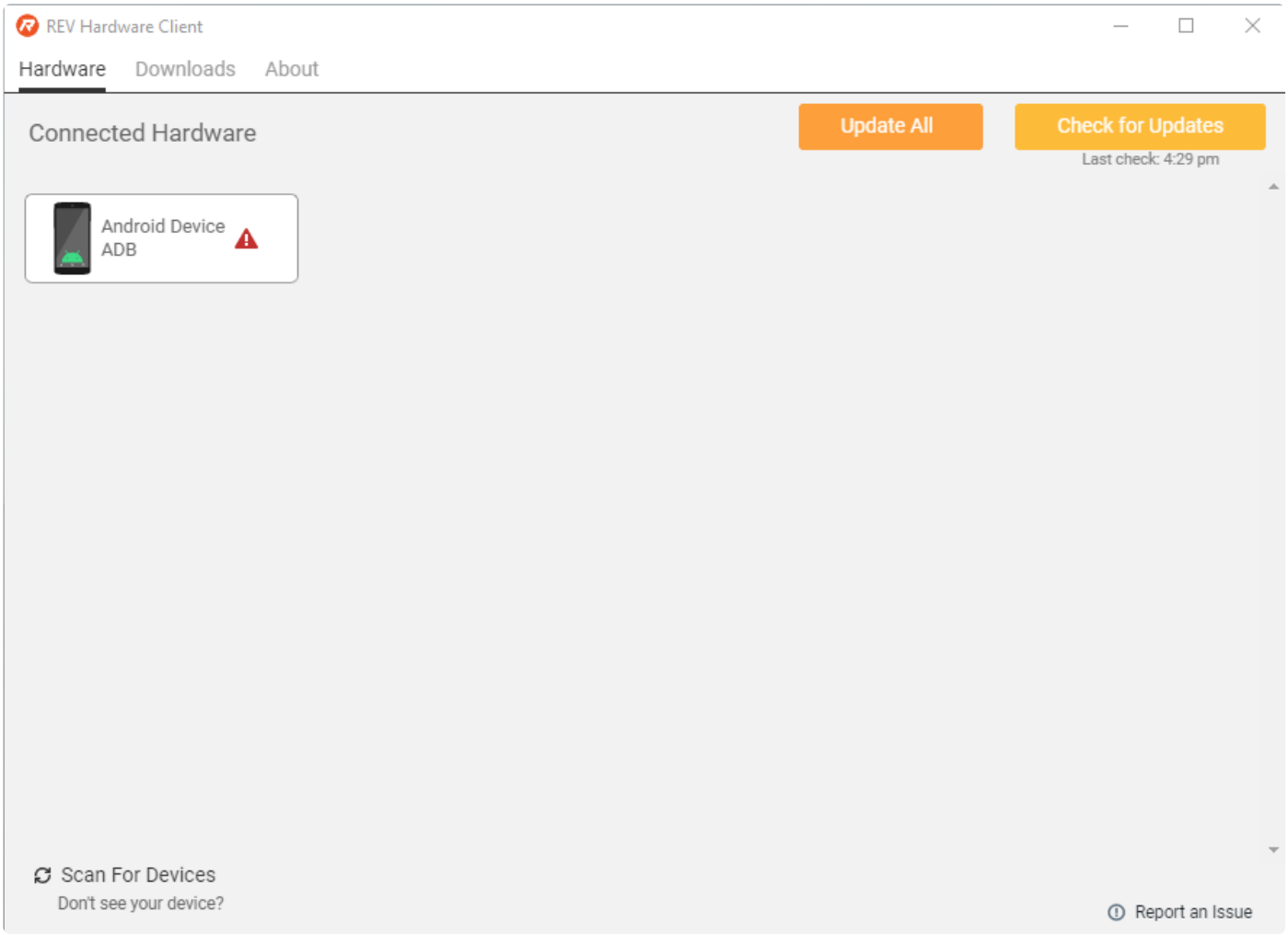


Robot Controller Application

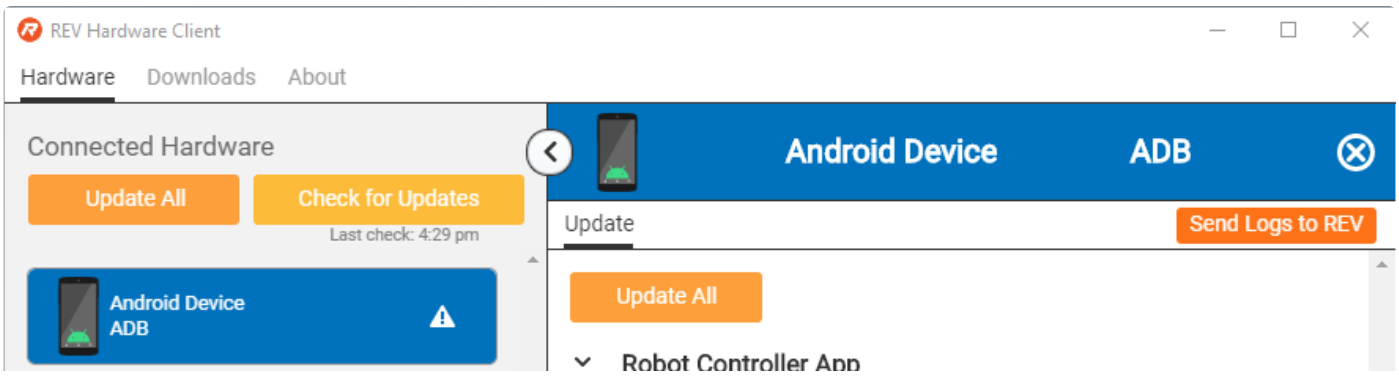
> The following steps will go through how to install the Robot Controller Application via the REV Hardware Client. It is possible to install the application via the app store or via the FTC GitHub repository as well.

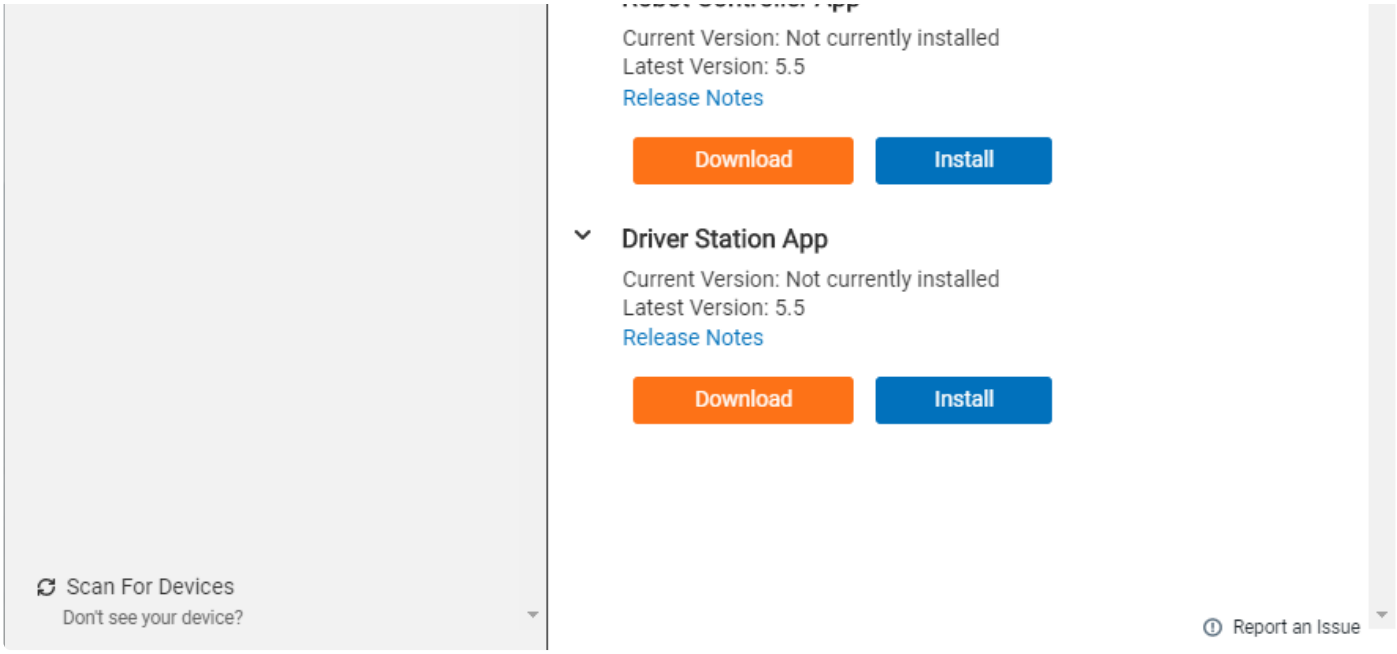
Connect the Android Device to a PC with the REV Hardware Client installed.

Startup the REV Hardware Client. Once the Android Device is fully connected it will show up on the front page of the UI under the **Hardware Tab**. Select the Android Device.

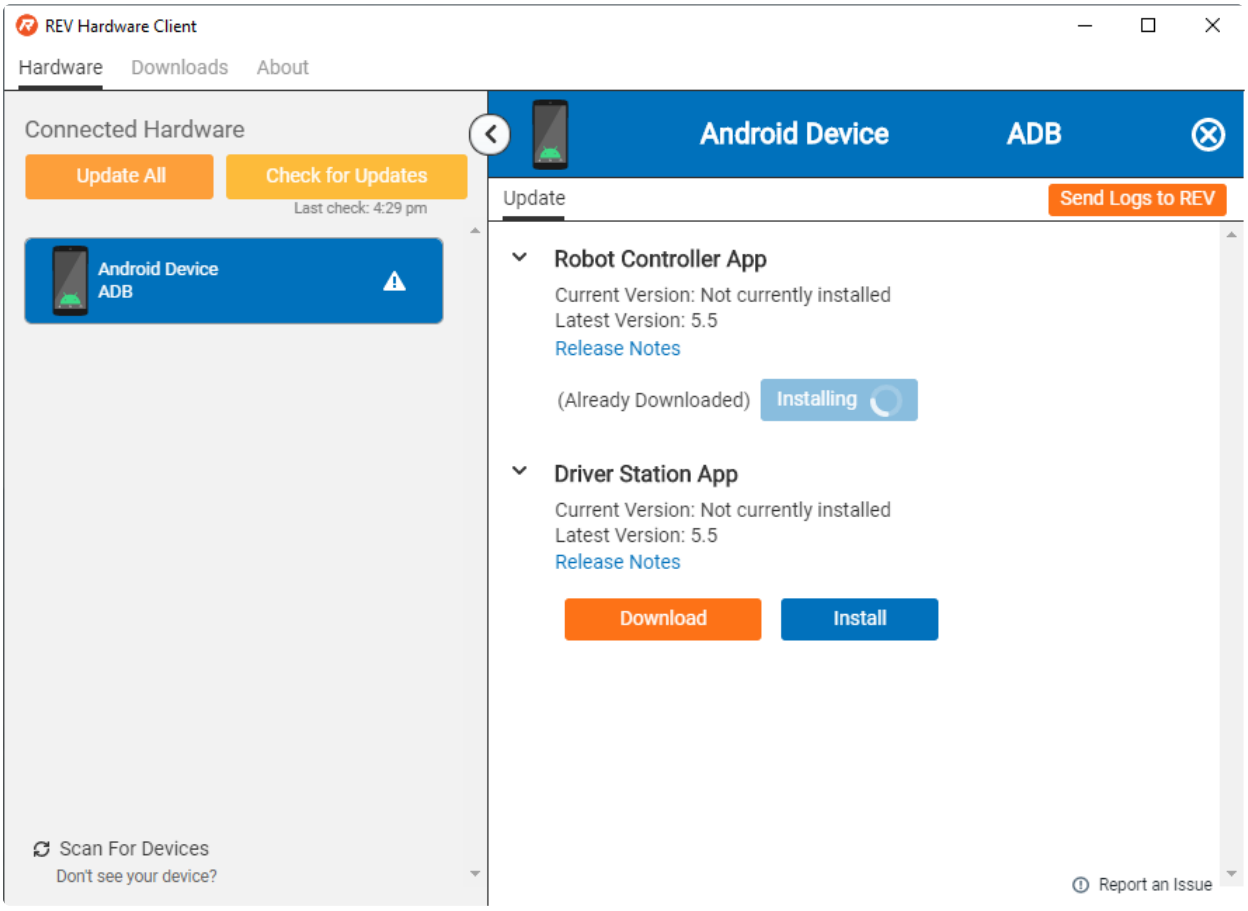


After selecting the Connected Hardware the Update tab will pop up. Under **Robot Controller App** select Download.

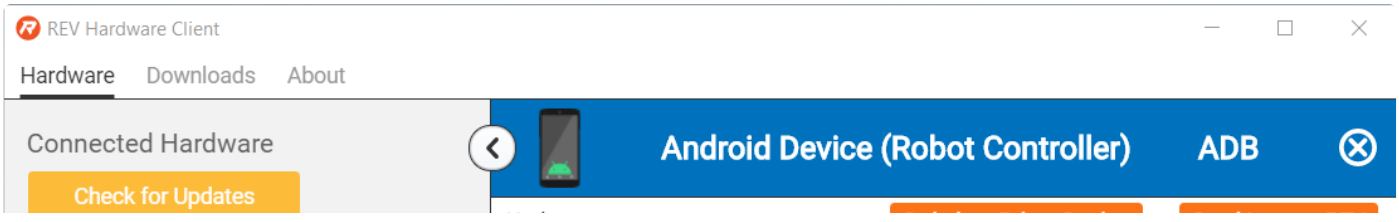


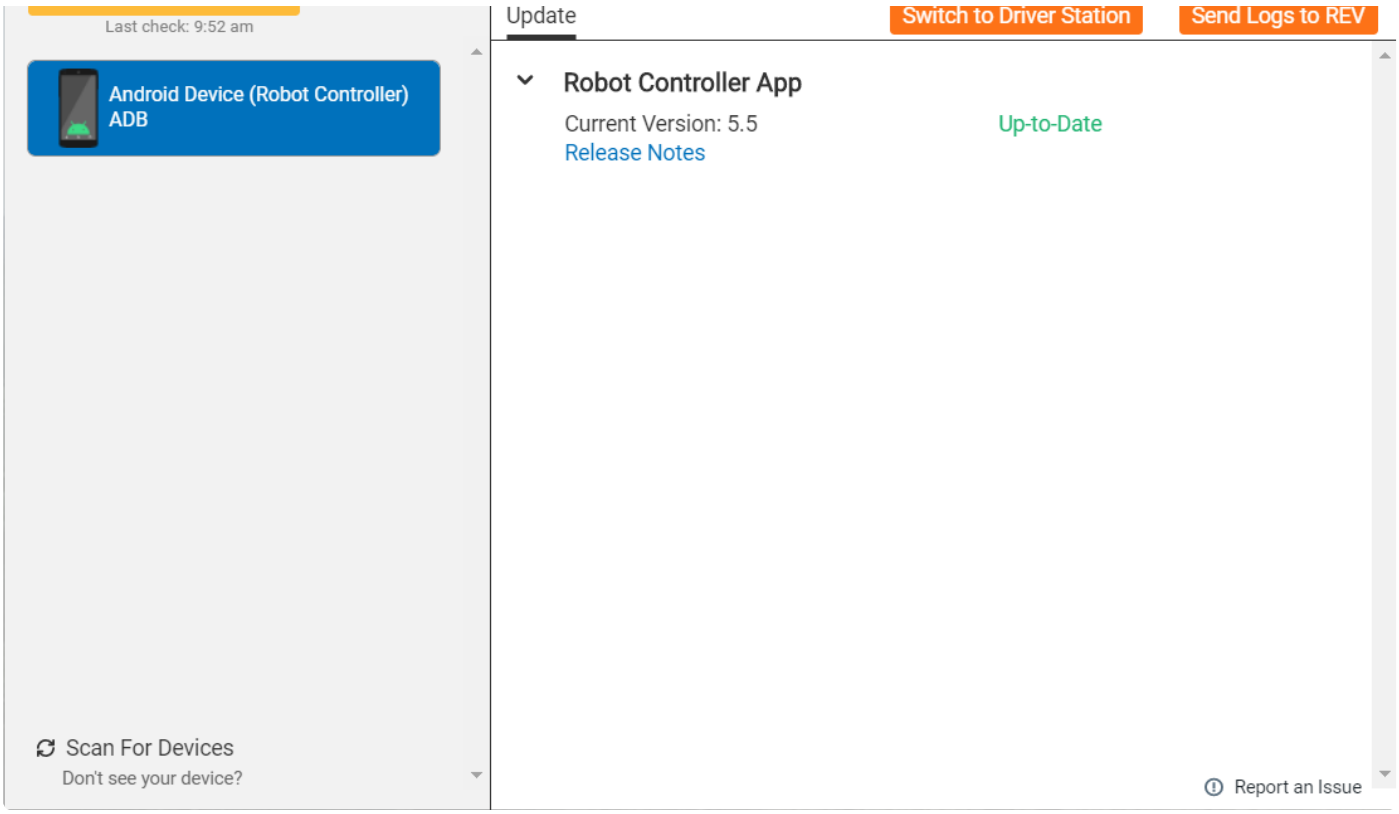


Once the Robot Controller App has downloaded, select Install.



When the application installation has completed the status for the Robot Controller App will change to "Up-to-Date."





Driver Station and Robot Controller Pairing

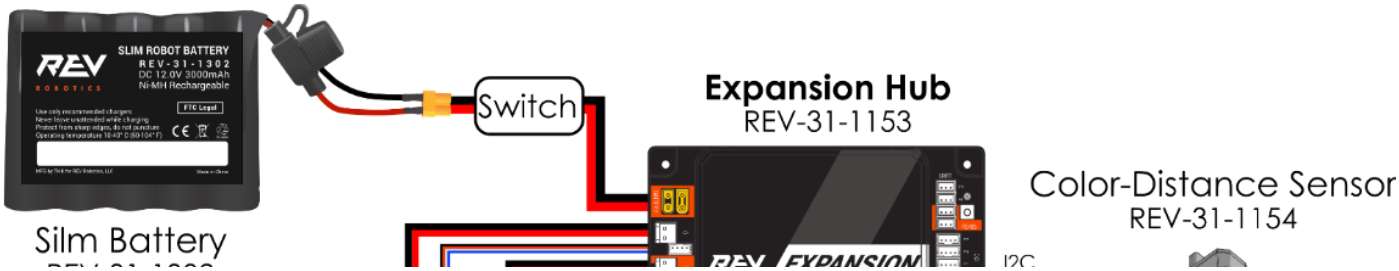
> You should update your Driver Station(DS) and Robot Controller(RC) phones to the latest app version in order to use the Expansion Hub controller. The minimum compatible version is 3.1 released on May 10th, 2017

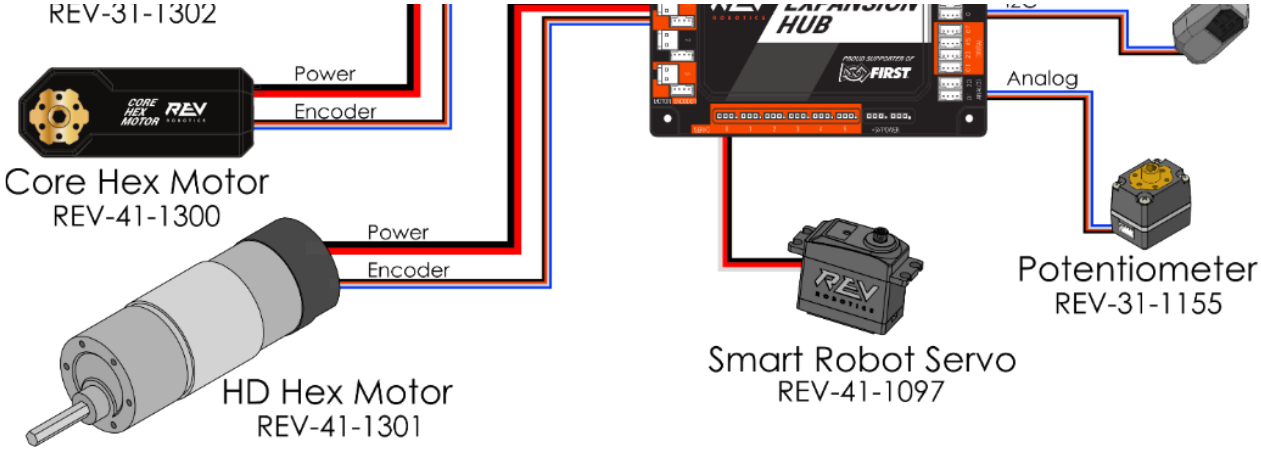
Please ensure that the Driver Station and Robot Controller phones are properly configured and paired. Refer to the latest pairing and troubleshooting instructions provided by in the FTC Control System Wiki.

Wiring Diagram

System Wiring Diagram

Before configuring your Expansion Hub, devices must be connected to the Expansion Hub. Below is a sample wiring diagram to show a sample of actuators and sensors usable with the Expansion Hub.



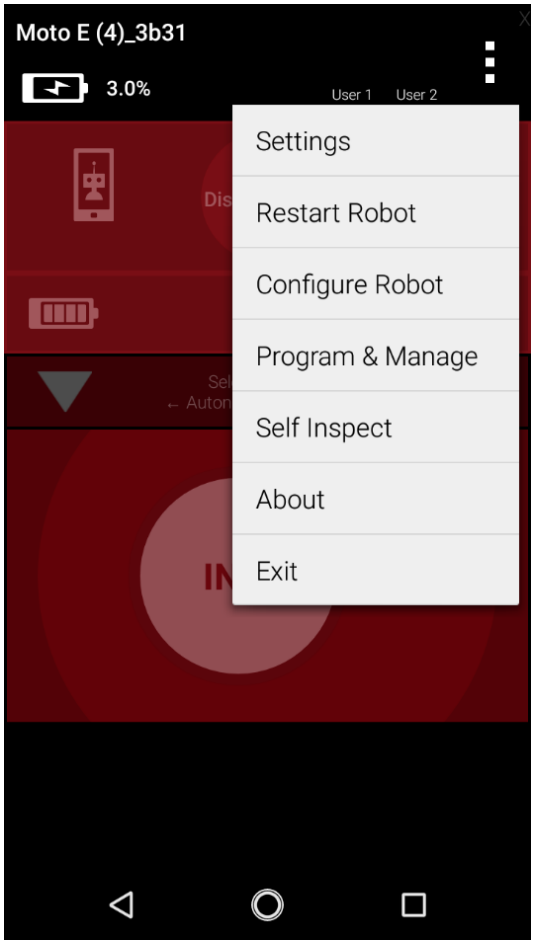


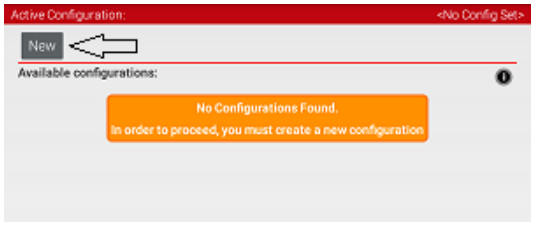
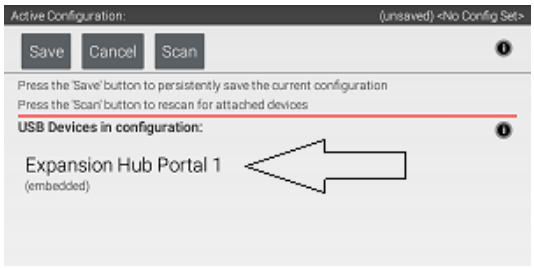
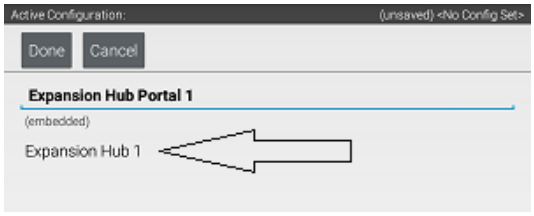
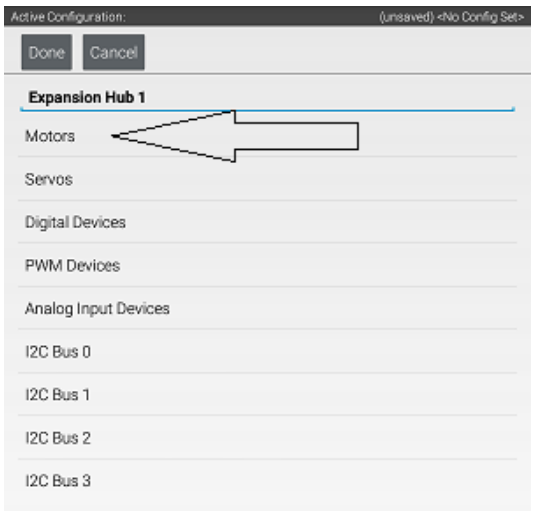
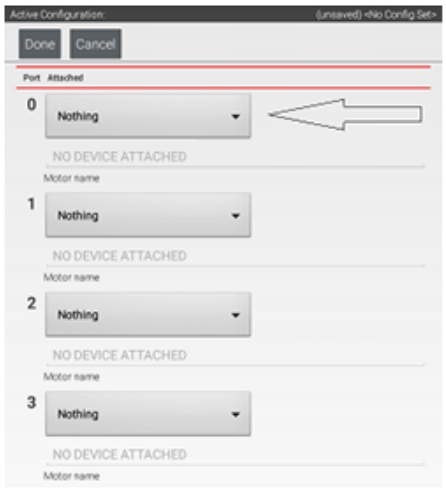
System Wiring Diagram

Configuration

Every device connected to the Expansion Hub (REV-31-1153) will need to be added to the Robot Configuration file before you can use the device in your program. The Robot Configuration will allow you to give your sensors and actuators meaningful names that you can reference while programming.

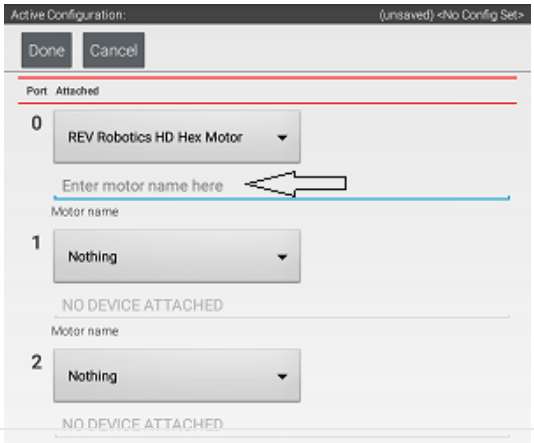
For this example, we will configure a simple two motor robot drivetrain.

Step	Image
Select the menu on either the Driver Station or Robot Controller. Then select “Configure Robot”.	

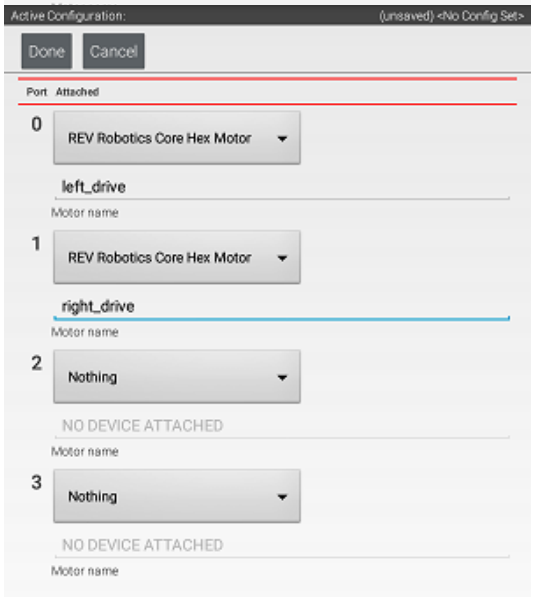
Select “New” in the top left hand corner.	
Select “Expansion Hub Portal 1” (embedded).	
Select “Expansion Hub 1”.	
Select “Motors”.	
Select the Drop Down menu for “Port 0” then select the motor type attached to the port. In the case of the Minibot in Figure 4, select the “Rev Robotics Core Hex Motor”.	

Press “Enter motor name here” and name the motor “left_drive”.

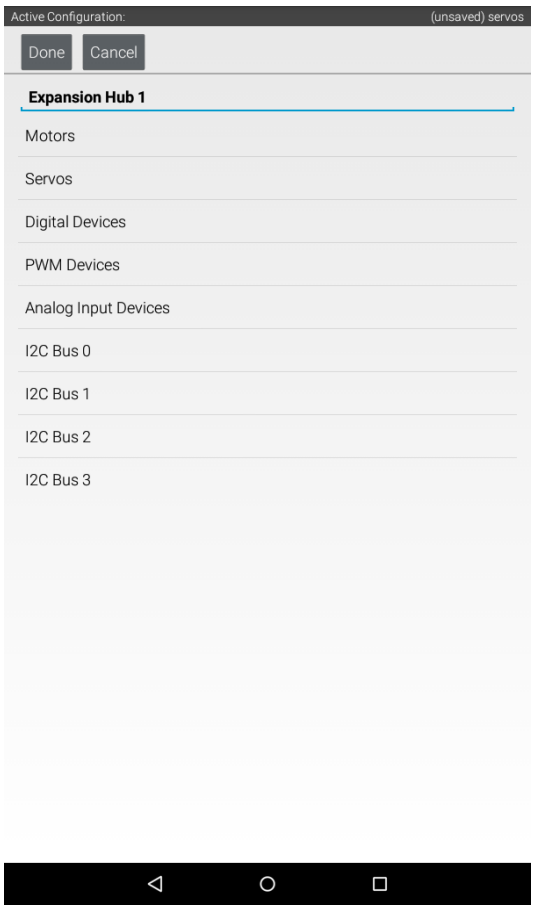
This is the name that you will use when you are programming your robot to control this motor. Always use descriptive names so that you can remember what a device does when you are programming.



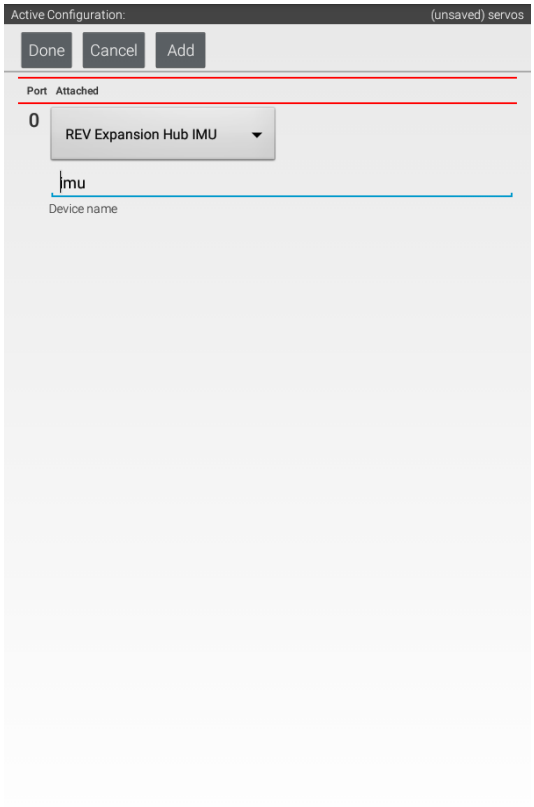
Repeat the process for “Port 1” and name the motor “right_drive”.



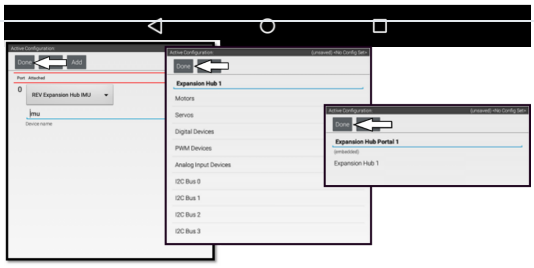
Press “Done” once to go back to the list of device ports and then select I2C Bus 0.



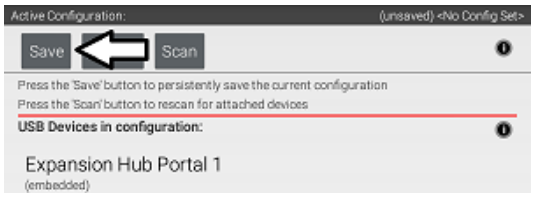
Add the built-in REV Expansion Hub IMU. Name it “imu”



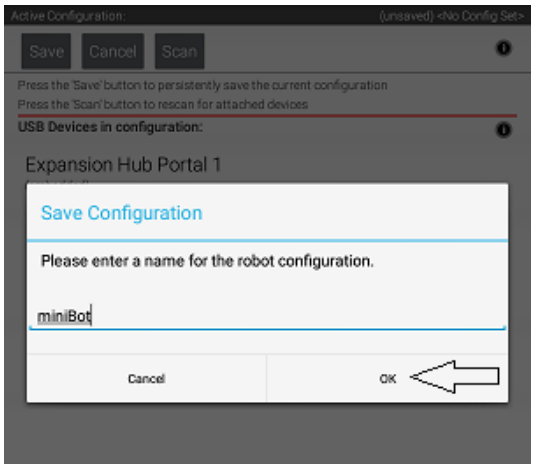
Press the “Done” button (at the top left corner of the page) 3 times.



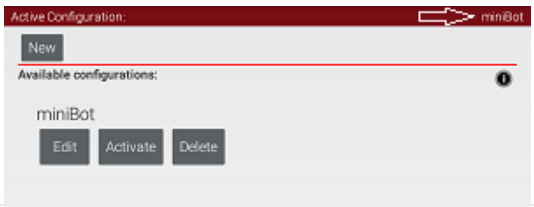
Press “Save”.



Enter “miniBot” as your configuration name, then select “OK”.



You now have an active configuration called

“miniBot”. Press the Android back button to return to the Driver Station page.	

REV Hub Interface Software

The REV Hub Interface is a beta software allowing for a direct connection from a REV Expansion Hub and its peripherals to a Windows PC.

This interface provides a method for teams to prototype with motors, servos, and sensors in a way that is faster and easier than setting up an entire robot control system. It is also a valuable troubleshooting tool that can help isolate the cause of an issue and determine if it is electrical or software related. The REV Hub Firmware can also be updated and recovered through this interface in addition to the Robot Controller Application.

Download the Latest Hub Interface Software - Version 1.2.0

> The REV Hub Interface Software only works with the REV Expansion Hub and not the REV Control Hub

System Requirements

- Operating System: Windows 7 or newer*
- Processor: 64-bit
- RAM: Yes

> The newest versions of Windows should automatically install the required USB drivers. Alternatively, you can download the latest drivers from the FTDI VCP website.

Installation Instructions

1. Download the Hub Interface software installer above.
2. Run the installer.
3. Run the REV Hub Interface Software from the Windows Start Menu or the desktop shortcut

Connecting and Controlling an Expansion Hub

1. Connect your Expansion Hub to the computer with a USB A to USB Mini-B cable.
2. Run the REV Hub Interface Software.
3. The software will scan and connect to the Expansion Hub. The various peripheral tabs will populate with controls once connected.

> Some peripherals, such as DC Motors and Servo Motors, require a battery to be connected to the Expansion Hub in order to operate through the REV Hub Interface.

Alternative Installation Method

You may also download the following zip file if you would rather unzip the application in a directory of your choice. This method shouldn't require administrator privileges.

REV Hub Interface Software Zip File

LATEST HUB INTERFACE SOFTWARE CHANGE LOG - VERSION 1.2.0

- Display encoder values on 'DC motors' tab.
- Added support for REV Color Sensor V3.
- Display proximity values along with RGBC for REV color sensors.
- Display REV Hub Interface version on the 'Firmware' tab.
- Changed behavior of 'INIT' and 'POLL' buttons on 'I2C'. User can no longer poll a device until it has been successfully initialized.
- Added ability to set LED pattern.
- Bug fix where 'POLL' had to be pressed twice to read values from the IMU.
- Bug fix where status LED would continue to flash blue the second time REV Hub Interface is connected.

- Allow user to press enter key to update motor/servo values.
- Fixed gyro labels on IMU tab and corrected units for linear acceleration.