
FIRST Robotics Competition

WPILib

Apr 13, 2021

Zero to Robot

1	Introduction	3
2	Step 1: Building your Robot	5
3	Step 2: Installing Software	43
4	Step 3: Preparing Your Robot	97
5	Step 4: Programming your Robot	117
6	Hardware Component Overview	133
7	Software Component Overview	159
8	What is WPILib?	175
9	2021 Overview	177
10	VS Code Overview	191
11	FRC LabVIEW Programming	229
12	Actuators	281
13	Sensors	307
14	CAN Devices	351
15	Basic Programming	361
16	Support Resources	377
17	FRC Glossary	379
18	Driver Station	381
19	Shuffleboard	439
20	SmartDashboard	499

21	Glass	531
22	LabVIEW Dashboard	549
23	PathWeaver	569
24	RobotBuilder	579
25	Robot Simulation	657
26	Robot Characterization	679
27	OutlineViewer	713
28	Vision Processing	715
29	Command-Based Programming	817
30	[Old] Command Based Programming	899
31	Kinematics and Odometry	941
32	NetworkTables	961
33	roboRIO	973
34	Advanced GradleRIO	991
35	Advanced Controls	1003
36	Convenience Features	1097
37	WPILib Example Projects	1099
38	Trajectory Tutorial	1105
39	Drivetrain Simulation Tutorial	1131
40	Hardware - Basics	1145
41	Hardware Tutorials	1193
42	Sensors	1195
43	Getting Started with Romi	1249
44	Networking Introduction	1279
45	Networking Utilities	1329
46	Contributing to frc-docs	1331
47	Developing with allwpilib	1349
	Index	1351

Welcome to the *FIRST*® Robotics Competition Control System Documentation! An overview of the changes from 2020 to 2021 is available on the [New for 2021](#) document.

Introduction

Welcome to the official documentation home for the *FIRST*® Robotics Competition Control System and WPILib software packages. This page is the primary resource documenting the use of the FRC® Control System (including wiring, configuration and software) as well as the WPILib libraries and tools.

1.1 New to Programming?

These pages cover the specifics of the WPILib libraries and the FRC Control System and do not describe the basics of using the supported programming languages. If you would like resources on learning the supported programming languages check out the recommendations below:

Note: You can continue with this Zero-to-Robot section to get a functioning basic robot without knowledge of the programming language. To go beyond that, you will need to be familiar with the language you choose to program in.

1.1.1 Java

- [Code Academy](#)
- [Head First Java 2nd Edition](#) is a very beginner friendly introduction to programming in Java (ISBN-10: 0596009208).

1.1.2 C++

- [LearnCPP](#)
- [Programming: Principles and Practice Using C++ 2nd Edition](#) is an introduction to C++ by the creator of the language himself (ISBN-10: 0321992784).
- [C++ Primer Plus 6th Edition](#) (ISBN-10: 0321776402).

1.1.3 LabVIEW

- [NI Learn LabVIEW](#)

1.2 Zero to Robot

The remaining pages in this tutorial are designed to be completed in order to go from zero to a working basic robot. The documents will walk you through wiring your robot, installation of all needed software, configuration of hardware, and loading a basic example program that should allow your robot to operate. When you complete a page, simply click **Next** to navigate to the next page and continue with the process. When you're done, you can click **Next** to continue to an overview of WPILib in C++/Java or jump back to the home page using the logo at the top left to explore the rest of the content.

Step 1: Building your Robot

An overview of the available control system hardware can be found [here](#).

2.1 How to Wire an FRC Robot

Note: This document details the wiring of a basic electronics board for bench-top testing.

Some images shown in this section reflect the setup for a Robot Control System using Victor SPX Motor Controllers. Wiring diagram and layout should be similar for other motor controllers. Where appropriate, two sets of images are provided to show connections using controllers with and without integrated wires.

2.1.1 Overview

2.1.2 Gather Materials

Locate the following control system components and tools

- Kit Materials:
 - Power Distribution Panel (PDP)
 - roboRIO
 - Pneumatics Control Module (PCM)
 - Voltage Regulator Module (VRM)
 - OpenMesh radio (with power cable and Ethernet cable)
 - Robot Signal Light (RSL)
 - 4x SPARK MAX or other motor controllers
 - 2x PWM y-cables
 - 120A Circuit breaker

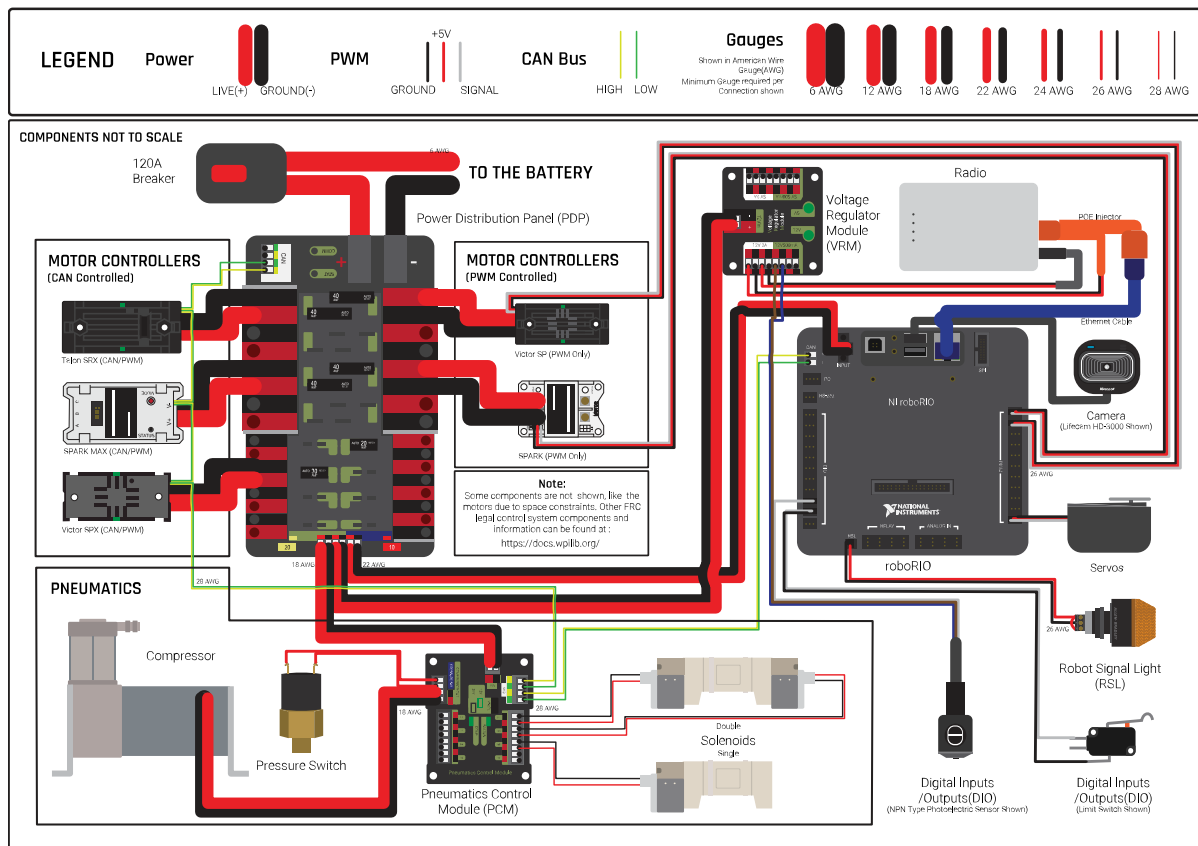


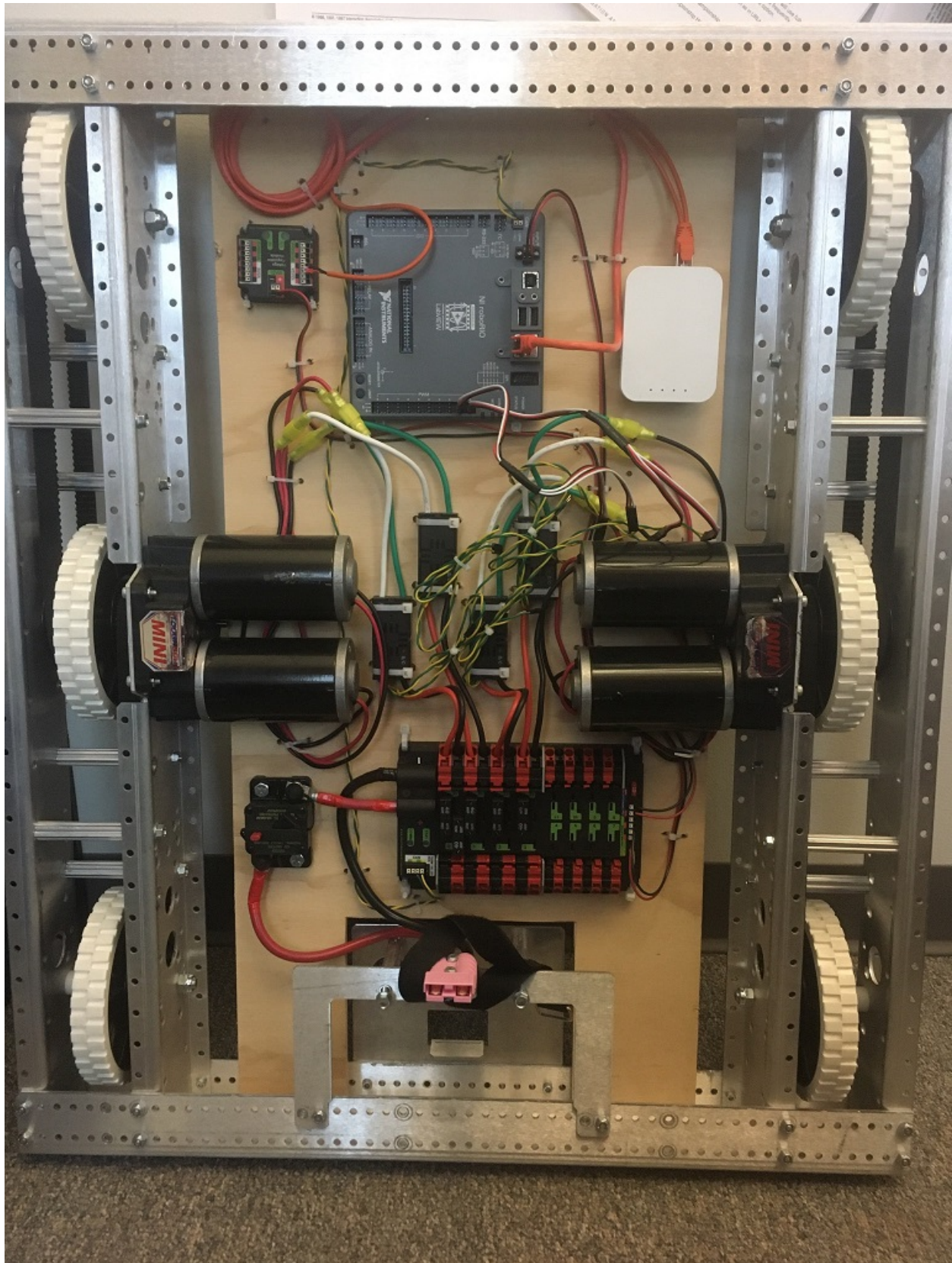
Fig. 1: Diagram courtesy of FRC® Team 3161 and Stefen Acepcion.

- 4x 40A Circuit breaker
- 6 AWG (16 mm^2) Red wire
- 10 AWG (6 mm^2) Red/Black wire
- 18 AWG (1 mm^2) Red/Black wire
- 22 AWG (0.5 mm^2) Yellow/Green twisted CAN cable
- 16x 10-12 AWG (4 - 6 mm^2) (Yellow) ring terminals (8x quick disconnect pairs if using integrated wire controllers)
- 2x Anderson SB50 battery connectors
- 6 AWG (16 mm^2) Terminal lugs
- 12V Battery
- Red/Black Electrical tape
- Dual Lock material or fasteners
- Zip ties
- 1/4" or 1/2" (6-12 mm) plywood
- Tools Required:
 - Wago Tool or small flat-head screwdriver
 - Very small flat head screwdriver (eyeglass repair size)
 - Philips head screw driver
 - 5 mm Hex key (3/16" may work if metric is unavailable)
 - 1/16" Hex key
 - Wire cutters, strippers, and crimpers
 - 7/16" (11 mm may work if imperial is unavailable) box end wrench or nut driver

2.1.3 Create the Base for the Control System

For a benchtop test board, cut piece of 1/4" or 1/2" (6-12 mm) material (wood or plastic) approximately 24" x 16" (60 x 40 cm). For a Robot Quick Build control board see the supporting documentation for the proper size board for the chosen chassis configuration.

2.1.4 Layout the Core Control System Components



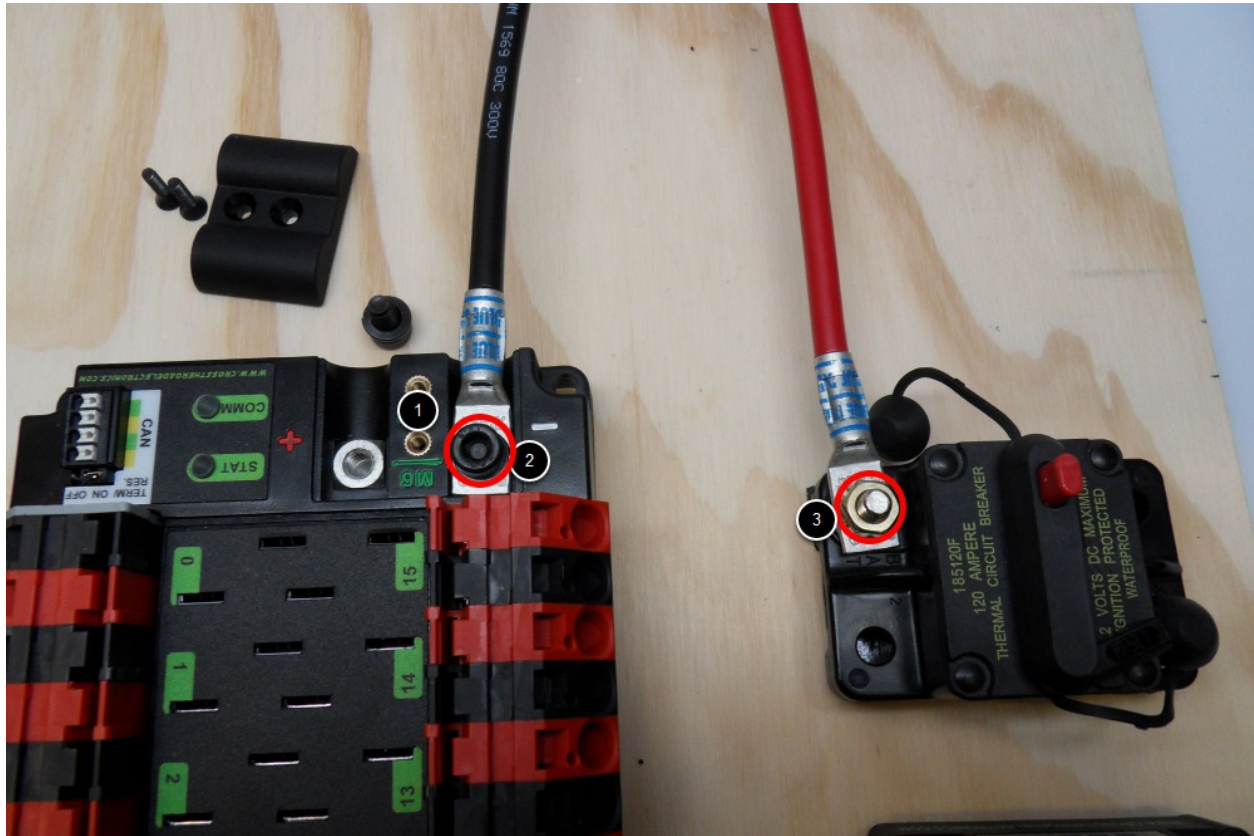
Lay out the components on the board. An example layout is shown in the image above.



2.1.5 Fasten Components

Using the Dual Lock or hardware, fasten all components to the board. Note that in many FRC games robot-to-robot contact may be substantial and Dual Lock alone is unlikely to stand up as a fastener for many electronic components. Teams may wish to use nut and bolt fasteners or (as shown in the image above) cable ties, with or without Dual Lock to secure devices to the board.

2.1.6 Attach Battery Connector to PDP

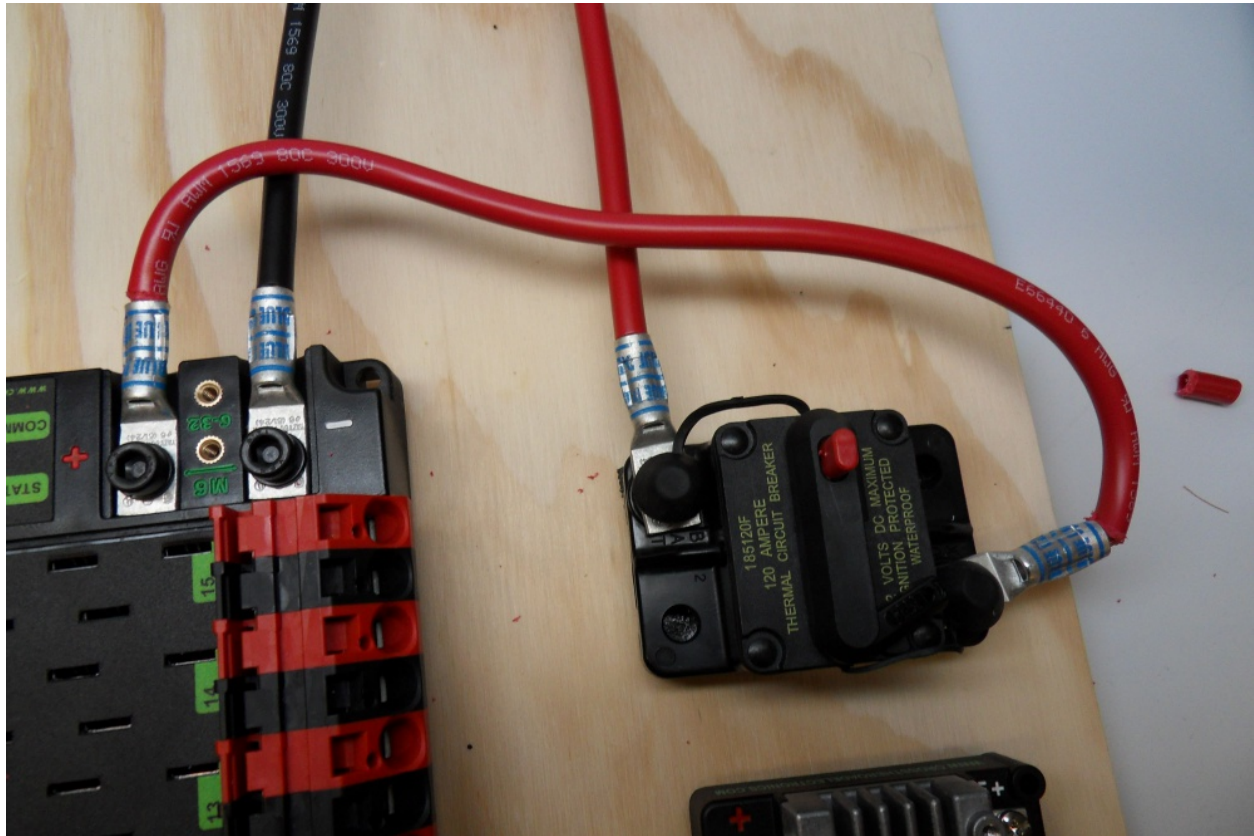


Requires: Battery Connector, 6 AWG (16 mm^2) terminal lugs, 1/16" Allen, 5 mm Allen, 7/16" (11 mm) Box end

Attach terminal lugs to battery connector:

1. Using a 1/16" Allen wrench, remove the two screws securing the PDP terminal cover.
2. Using a 5 mm Allen wrench (3/16"), remove the negative (-) bolt and washer from the PDP and fasten the negative terminal of the battery connector.
3. Using a 7/16" (11 mm) box end wrench, remove the nut on the "Batt" side of the main breaker and secure the positive terminal of the battery connector

2.1.7 Wire Breaker to PDP

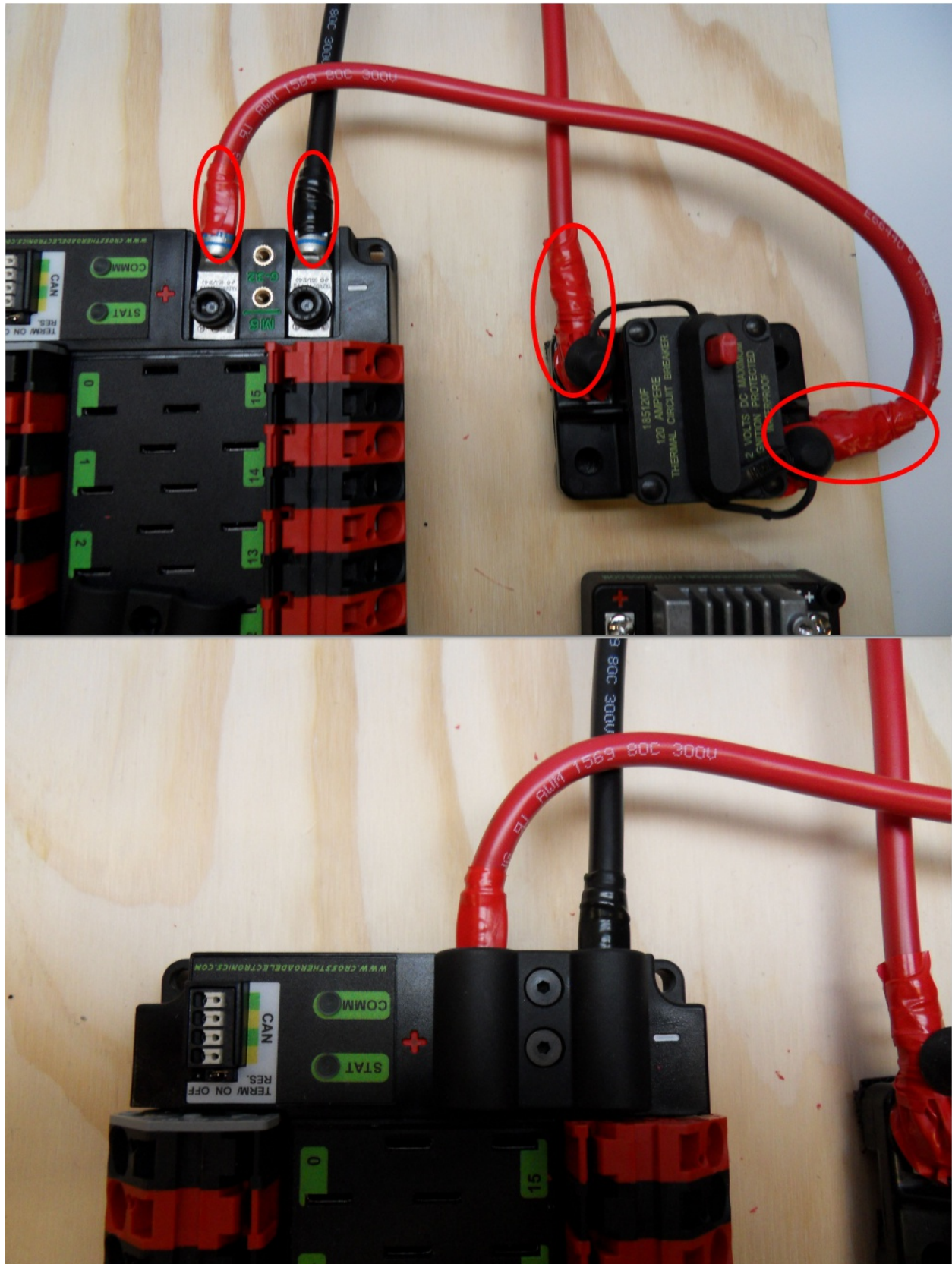


Requires: 6 AWG (16 mm^2) red wire, 2x 6 AWG (16 mm^2) terminal lugs, 5 mm Allen, 7/16" (11 mm) box end

Secure one terminal lug to the end of the 6 AWG (16 mm^2) red wire. Using the 7/16" (11 mm) box end, remove the nut from the "AUX" side of the 120A main breaker and place the terminal over the stud. Loosely secure the nut (you may wish to remove it shortly to cut, strip, and crimp the other end of the wire). Measure out the length of wire required to reach the positive terminal of the PDP.

1. Cut, strip, and crimp the terminal to the 2nd end of the red 6 AWG (16 mm^2) wire.
2. Using the 7/16" (11 mm) box end, secure the wire to the "AUX" side of the 120A main breaker.
3. Using the 5 mm Allen wrench, secure the other end to the PDP positive terminal.

2.1.8 Insulate PDP connections



Requires: 1/16" Allen, Electrical tape

1. Using electrical tape, insulate the two connections to the 120A breaker. Also insulate any part of the PDP terminals which will be exposed when the cover is replaced. One method for insulating the main breaker connections is to wrap the stud and nut first, then use the tape wrapped around the terminal and wire to secure the tape.
2. Using the 1/16" Allen wrench, replace the PDP terminal cover

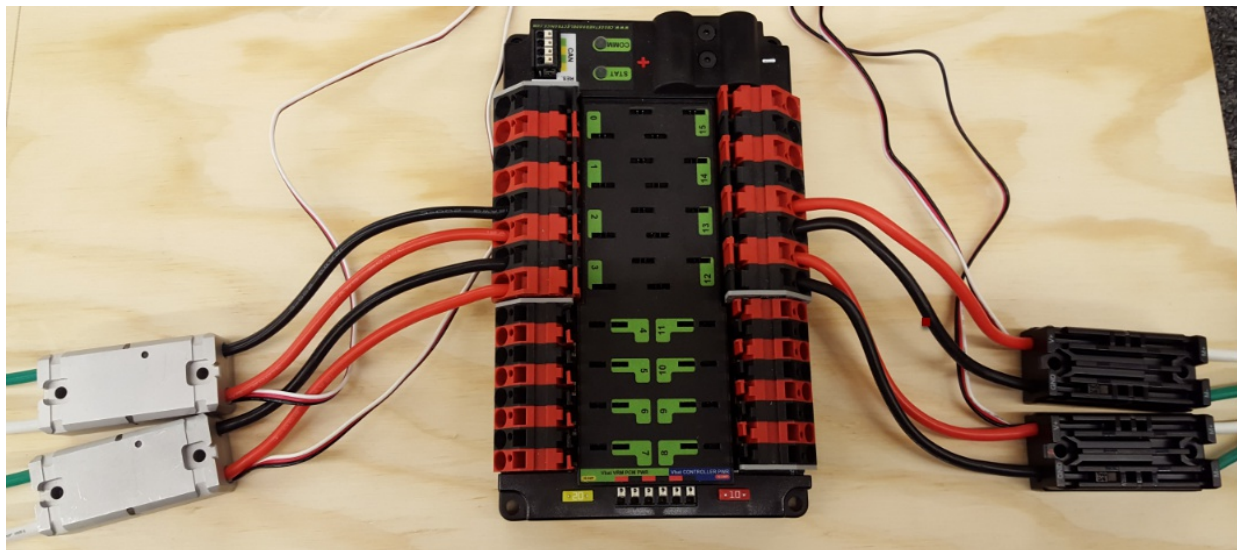
2.1.9 Wago connectors

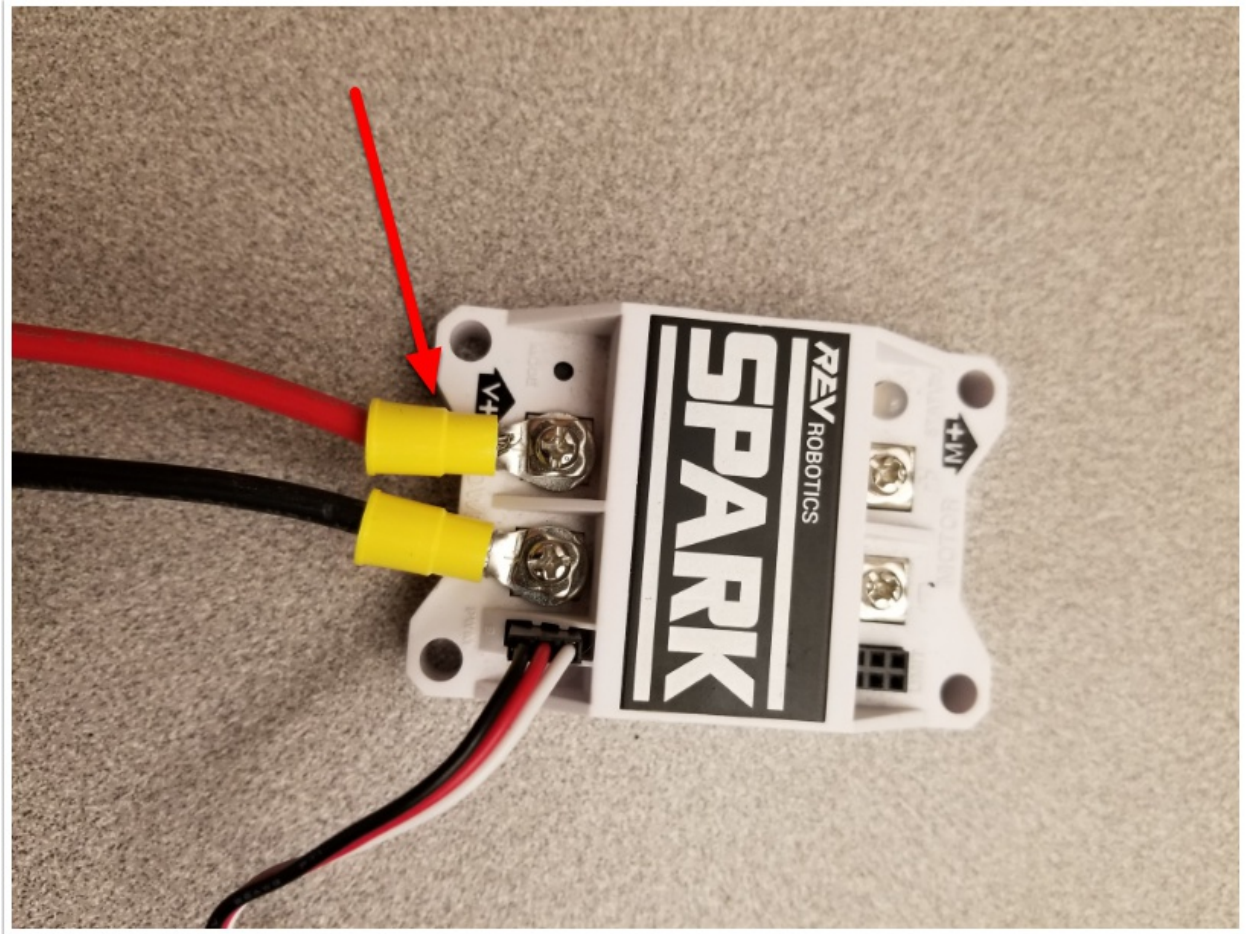
The next step will involve using the Wago connectors on the PDP. To use the Wago connectors, insert a small flat blade screwdriver into the rectangular hole at a shallow angle then angle the screwdriver upwards as you continue to press in to actuate the lever, opening the terminal. Two sizes of Wago connector are found on the PDP:

- Small Wago connector: Accepts 10 - 24 AWG ($0.25 - 6 \text{ mm}^2$), strip 11-12 mm ($\sim 7/16"$)
- Large Wago connector: Accepts 6 - 12 AWG ($4 - 16 \text{ mm}^2$), strip 12-13 mm ($\sim 1/2"$)

To maximize pullout force and minimize connection resistance wires should not be tinned (and ideally not twisted) before inserting into the Wago connector.

2.1.10 Motor Controller Power





Requires: Wire Stripper, Small Flat Screwdriver, 10 or 12 AWG (4 - 6 mm^2) wire, 10 or 12 AWG (4 - 6 mm^2) fork/ring terminals (terminal controllers only), wire crimper

For SPARK MAX or other wire integrated motor controllers (top image):

- Cut and strip the red and black power input wires, then insert into one of the 40A (larger) Wago terminal pairs.

For terminal motor controllers (bottom image):

1. Cut red and black wire to appropriate length to reach from one of the 40A (larger) Wago terminal pairs to the input side of the motor controller (with a little extra for the length that will be inserted into the terminals on each end)
2. Strip one end of each of the wires, then insert into the Wago terminals.
3. Strip the other end of each wire, and crimp on a ring or fork terminal
4. Attach the terminal to the motor controller input terminals (red to +, black to -)

2.1.11 Weidmuller Connectors

Warning: The correct strip length is $\sim 5/16''$ (~ 8 mm), not the $5/8''$ (~ 16 mm) mentioned in the video.

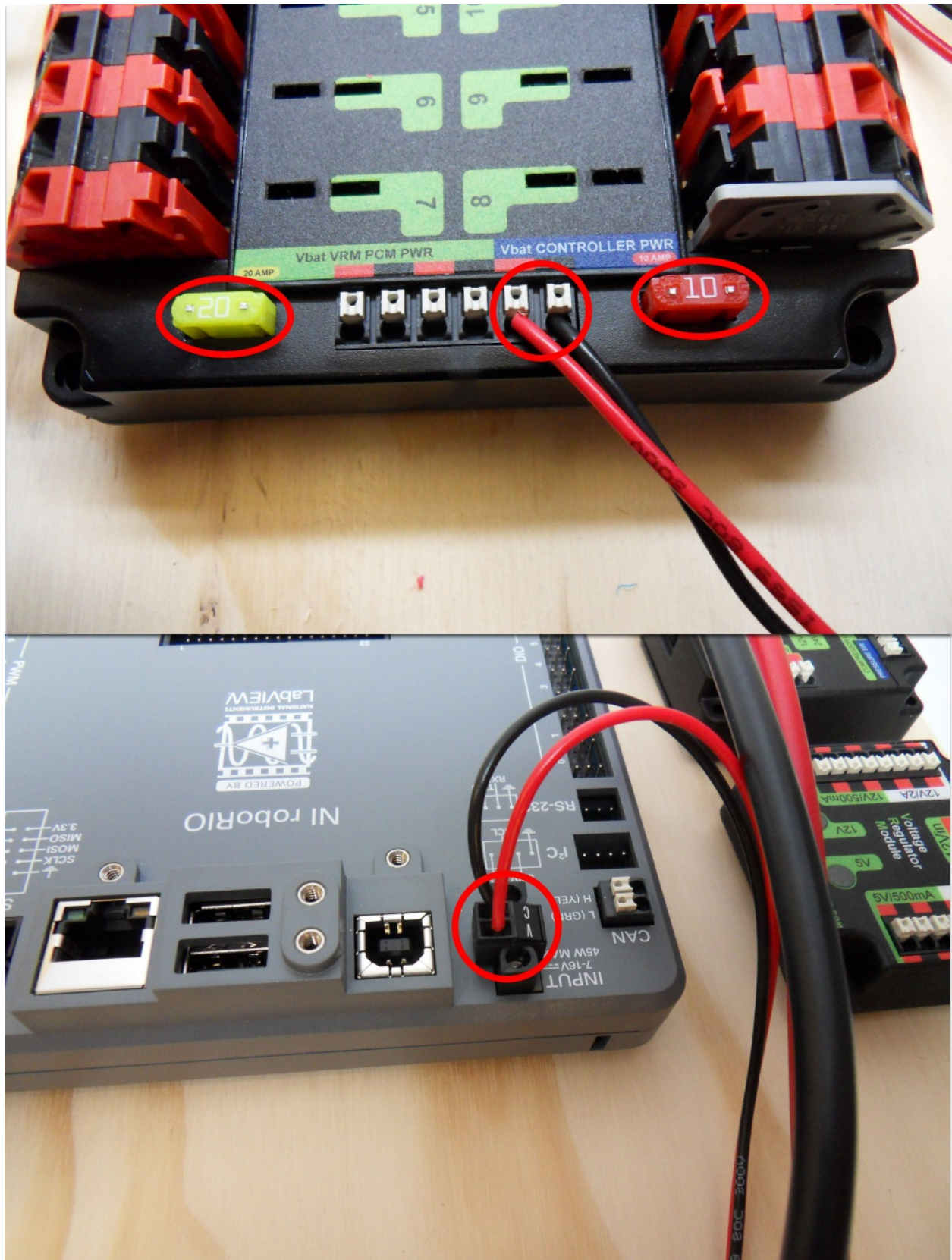
A number of the CAN and power connectors in the system use a Weidmuller LSF series wire-to-board connector. There are a few things to keep in mind when using this connector for best results:

- Wire should be 16 AWG (1.5 mm^2) to 24 AWG (0.25 mm^2) (consult rules to verify required gauge for power wiring)
- Wire ends should be stripped approximately $5/16''$ (~ 8 mm)
- To insert or remove the wire, press down on the corresponding “button” to open the terminal

After making the connection check to be sure that it is clean and secure:

- Verify that there are no “whiskers” outside the connector that may cause a short circuit
- Tug on the wire to verify that it is seated fully. If the wire comes out and is the correct gauge it needs to be inserted further and/or stripped back further.

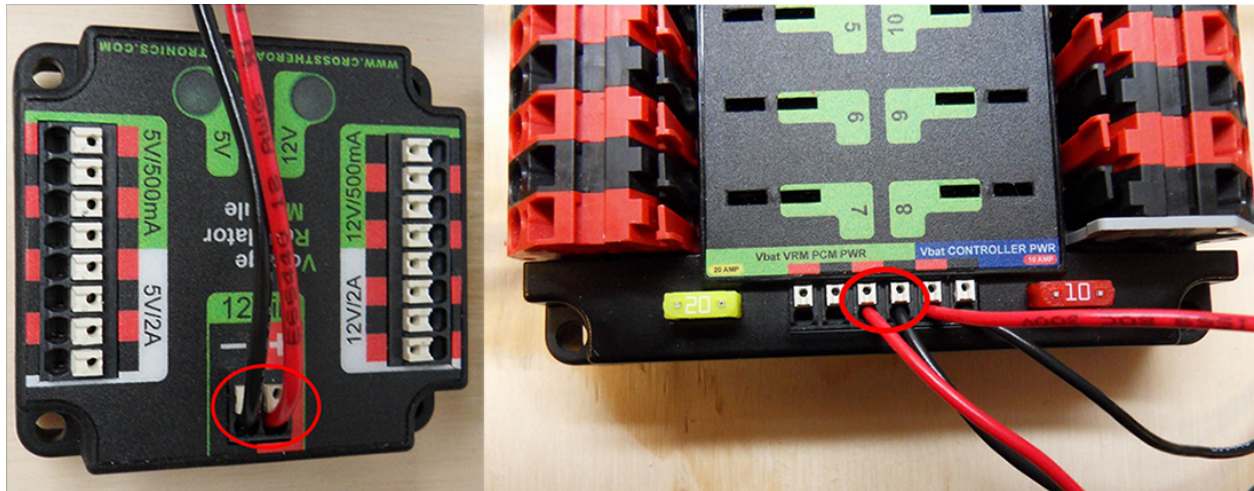
2.1.12 roboRIO Power



Requires: 10A/20A mini fuses, Wire stripper, very small flat screwdriver, 18 AWG (1 mm^2) Red and Black

1. Insert the 10A and 20A mini fuses in the PDP in the locations shown on the silk screen (and in the image above)
2. Strip $\sim 5/16"$ ($\sim 8 \text{ mm}$) on both the red and black 18 AWG (1 mm^2) wire and connect to the "Vbat Controller PWR" terminals on the PDB
3. Measure the required length to reach the power input on the roboRIO. Take care to leave enough length to route the wires around any other components such as the battery and to allow for any strain relief or cable management.
4. Cut and strip the wire.
5. Using a very small flat screwdriver connect the wires to the power input connector of the roboRIO (red to V, black to C). Also make sure that the power connector is screwed down securely to the roboRIO.

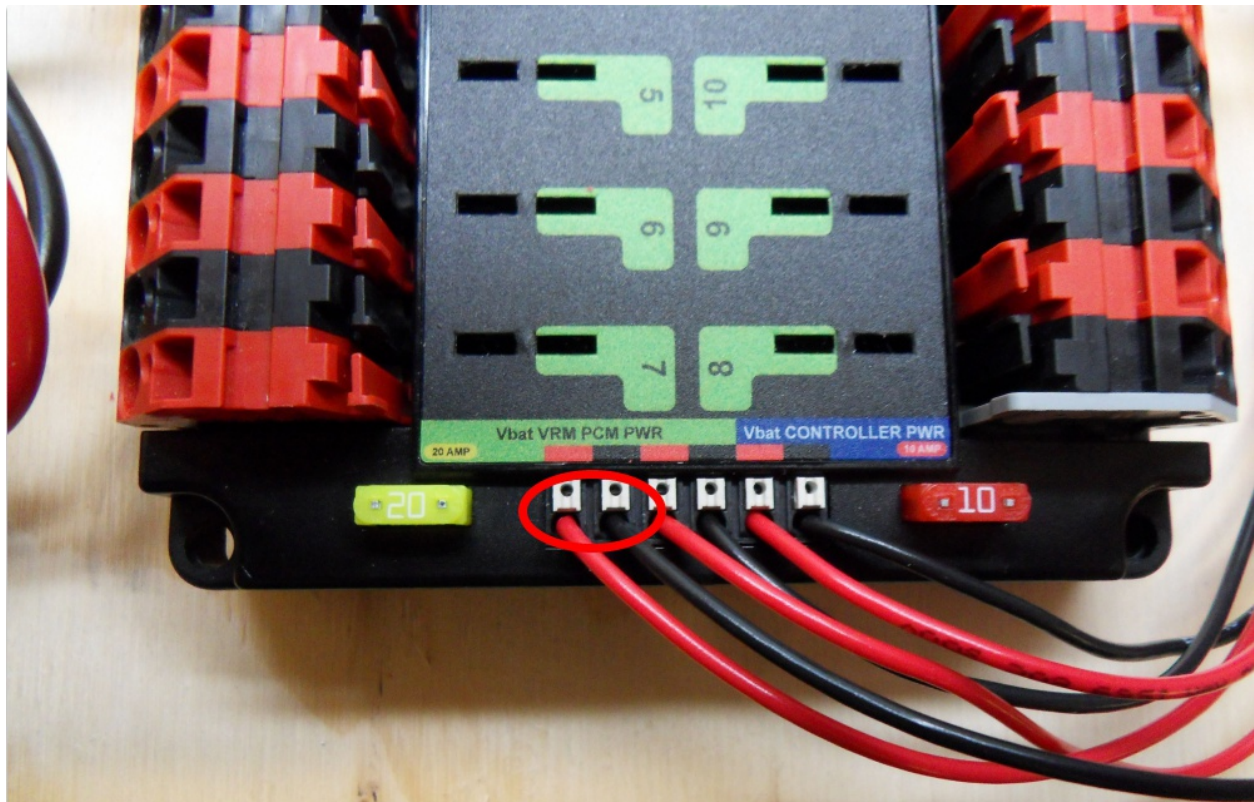
2.1.13 Voltage Regulator Module Power



Requires: Wire stripper, small flat screwdriver (optional), 18 AWG (1 mm^2) red and black wire:

1. Strip $\sim 5/16"$ ($\sim 8 \text{ mm}$) on the end of the red and black 18 AWG (1 mm^2) wire.
2. Connect the wire to one of the two terminal pairs labeled "Vbat VRM PCM PWR" on the PDP.
3. Measure the length required to reach the "12Vin" terminals on the VRM. Take care to leave enough length to route the wires around any other components such as the battery and to allow for any strain relief or cable management.
4. Cut and strip $\sim 5/16"$ ($\sim 8 \text{ mm}$) from the end of the wire.
5. Connect the wire to the VRM 12Vin terminals.

2.1.14 Pneumatics Control Module Power (Optional)



Requires: Wire stripper, small flat screwdriver (optional), 18 AWG (1 mm^2) red and black wire

1. Strip $\sim 5/16"$ ($\sim 8\text{ mm}$) on the end of the red and black 18 AWG (1 mm^2) wire.
2. Connect the wire to one of the two terminal pairs labeled "Vbat VRM PCM PWR" on the PDP.
3. Measure the length required to reach the "Vin" terminals on the PCM. Take care to leave enough length to route the wires around any other components such as the battery and to allow for any strain relief or cable management.
4. Cut and strip $\sim 5/16"$ ($\sim 8\text{ mm}$) from the end of the wire.
5. Connect the wire to the PCM 12Vin terminals.

2.1.15 Radio Power and Ethernet

Warning: DO NOT connect the Rev passive POE injector cable directly to the roboRIO. The roboRIO MUST connect to the female end of the cable using an additional Ethernet cable as shown in the next step.



Requires: Small flat screwdriver (optional), Rev radio PoE cable

1. Insert the ferrules of the passive PoE injector cable into the corresponding colored terminals on the 12V/2A section of the VRM.
2. Connect the male RJ45 (Ethernet) end of the cable into the Ethernet port on the radio closest to the barrel connector (labeled 18-24v POE)

2.1.16 roboRIO to Radio Ethernet

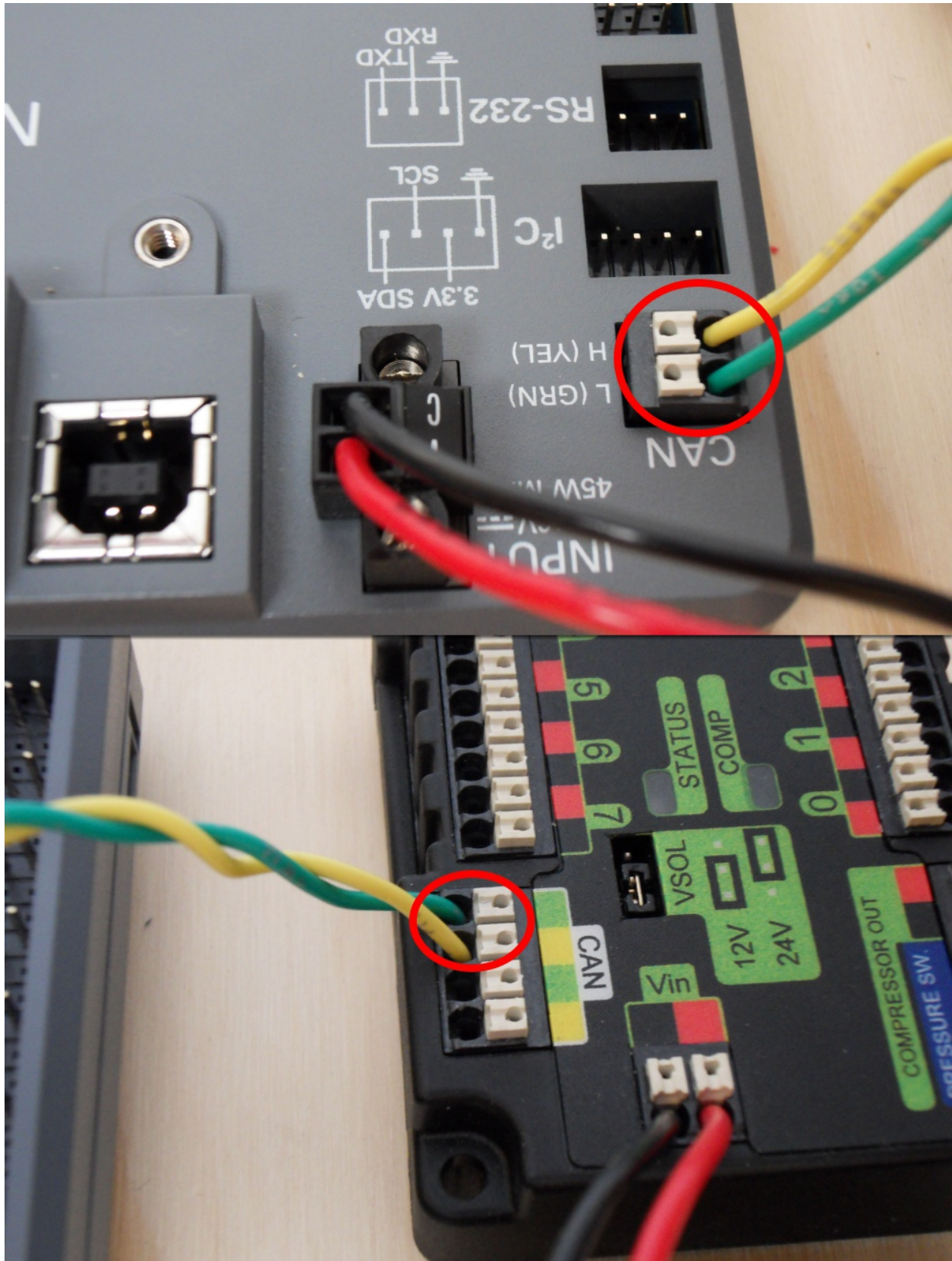


Requires: Ethernet cable

Connect an Ethernet cable from the female RJ45 (Ethernet) port of the Rev Passive POE cable to the RJ45 (Ethernet) port on the roboRIO.

2.1.17 CAN Devices

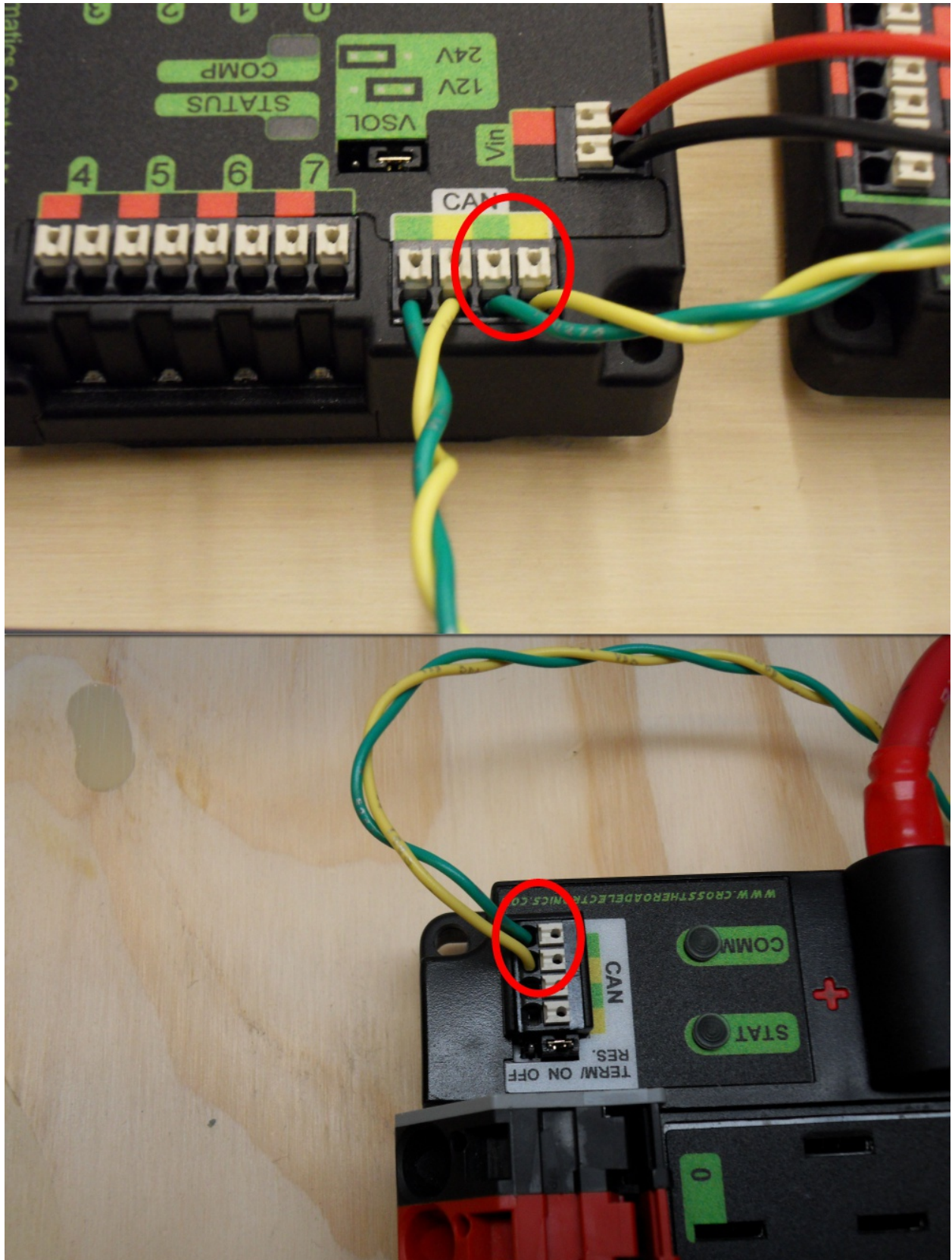
roboRIO to PCM CAN



Requires: Wire stripper, small flat screwdriver (optional), yellow/green twisted CAN cable

1. Strip $\sim 5/16"$ (~ 8 mm) off of each of the CAN wires.
2. Insert the wires into the appropriate CAN terminals on the roboRIO (Yellow->YEL, Green->GRN).
3. Measure the length required to reach the CAN terminals of the PCM (either of the two available pairs). Cut and strip $\sim 5/16"$ (~ 8 mm) off this end of the wires.
4. Insert the wires into the appropriate color coded CAN terminals on the PCM. You may use either of the Yellow/Green terminal pairs on the PCM, there is no defined in or out.

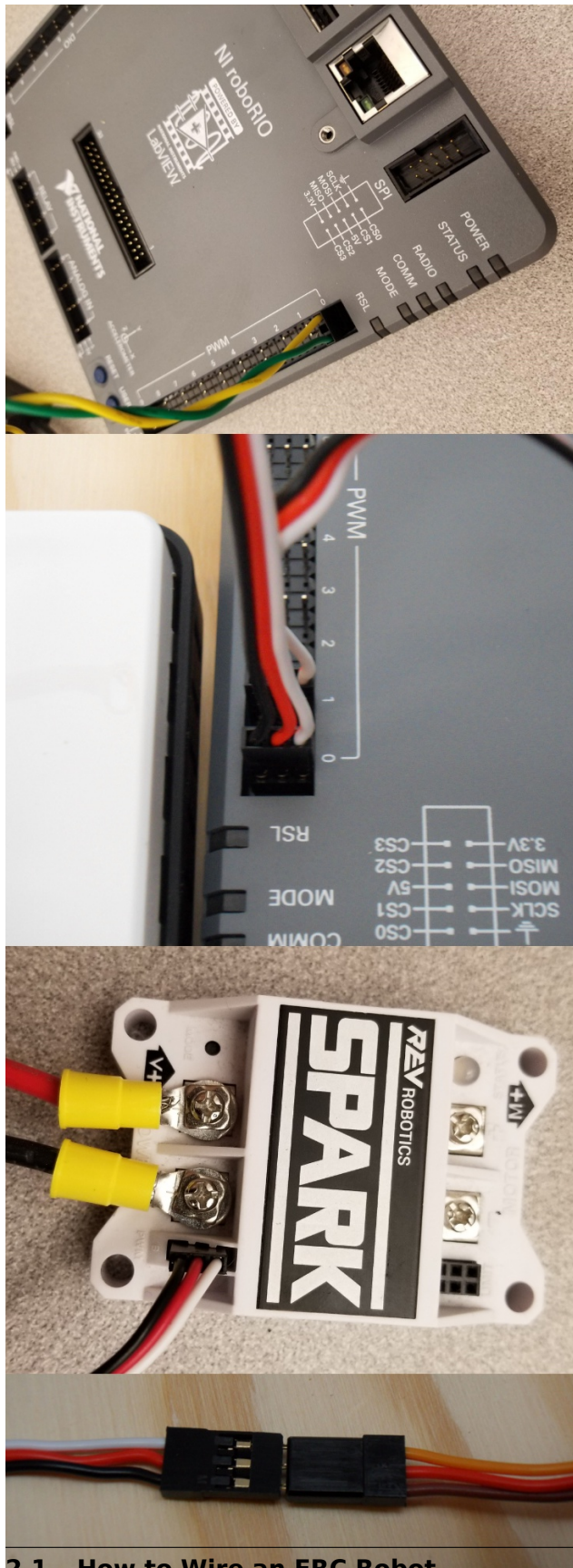
PCM to PDP CAN



Requires: Wire stripper, small flat screwdriver (optional), yellow/green twisted CAN cable

1. Strip $\sim 5/16"$ (~ 8 mm) off of each of the CAN wires.
2. Insert the wires into the appropriate CAN terminals on the PCM.
3. Measure the length required to reach the CAN terminals of the PDP (either of the two available pairs). Cut and strip $\sim 5/16"$ (~ 8 mm) off this end of the wires.
4. Insert the wires into the appropriate color coded CAN terminals on the PDP. You may use either of the Yellow/Green terminal pairs on the PDP, there is no defined in or out.

2.1.18 PWM Cables



This section details how to wire the SPARK MAX controllers using PWM signaling. This is a recommended starting point as it is less complex and easier to troubleshoot than CAN operation. The SPARK MAXs (and many other FRC motor controllers) can also be wired using [CAN](#) which unlocks easier configuration, advanced functionality, better diagnostic data and reduces the amount of wire needed.

Requires: 4x SPARK MAX PWM adapters (if using SPARK MAX), 4x PWM cables (if controllers without integrated wires or adapters, otherwise optional), 2x PWM Y-cable (Optional)

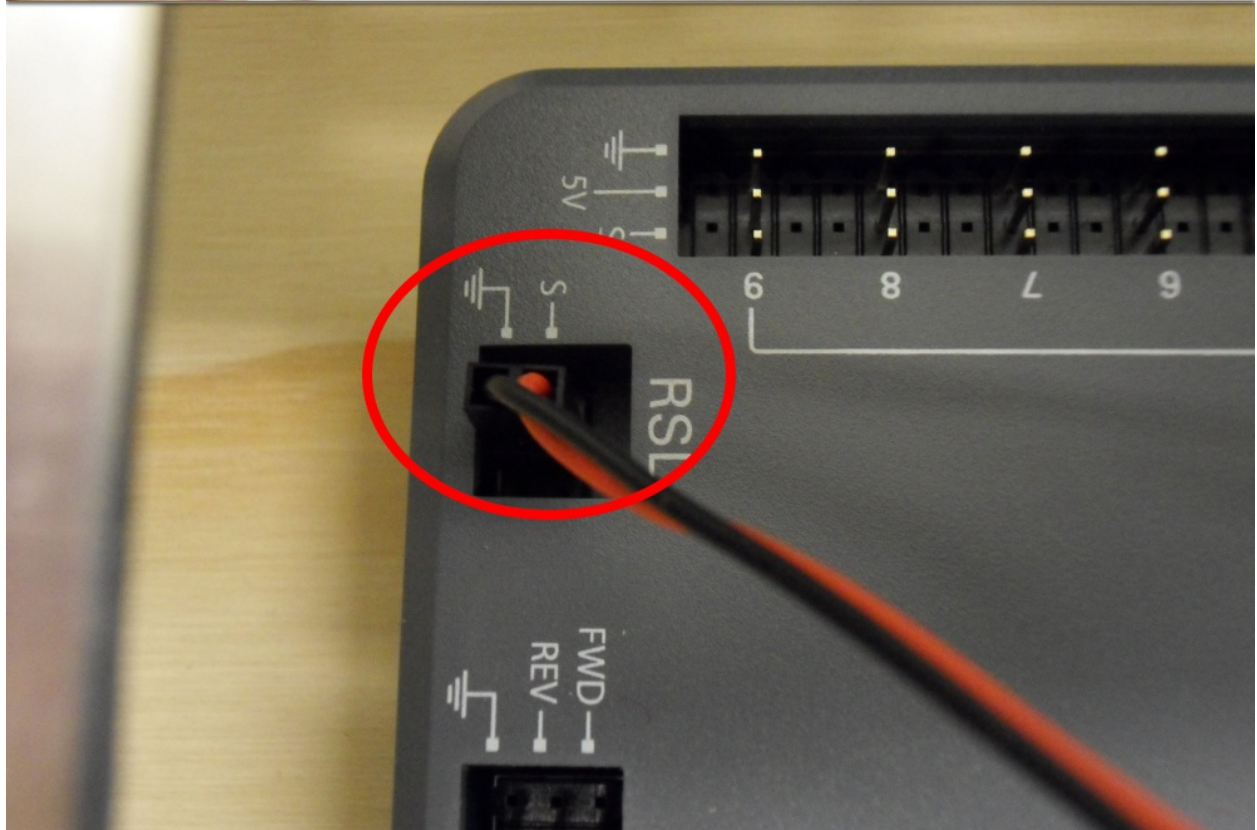
Option 1 (Direct connect):

1. If using SPARK MAX, attach the PWM adapter to the SPARK MAX (small adapter with a 3 pin connector with black/white wires).
2. If needed, attach PWM extension cables to the controller or adapter. On the controller side, match the colors or markings (some controllers may have green/yellow wiring, green should connect to black).
3. Attach the other end of the cable to the roboRIO with the black wire towards the outside of the roboRIO. It is recommended to connect the left side to PWM 0 and 1 and the right side to PWM 2 and 3 for the most straightforward programming experience, but any channel will work as long as you note which side goes to which channel and adjust the code accordingly.

Option 2 (Y-cable):

1. If using SPARK MAX, attach the PWM adapter to the SPARK MAX (small adapter with a 3 pin connector with black/white wires).
2. If needed, attach PWM extension cables between the controller or adapter and the PWM Y-cable. On the controller side, match the colors or markings (some controllers may have green/yellow wiring, green should connect to black).
3. Connect 1 PWM Y-cable to the 2 PWM cables for the controllers controlling each side of the robot. The brown wire on the Y-cable should match the black wire on the PWM cable.
4. Connect the PWM Y-cables to the PWM ports on the roboRIO. The brown wire should be towards the outside of the roboRIO. It is recommended to connect the left side to PWM 0 and the right side to PWM 1 for the most straightforward programming experience, but any channel will work as long as you note which side goes to which channel and adjust the code accordingly.

2.1.19 Robot Signal Light

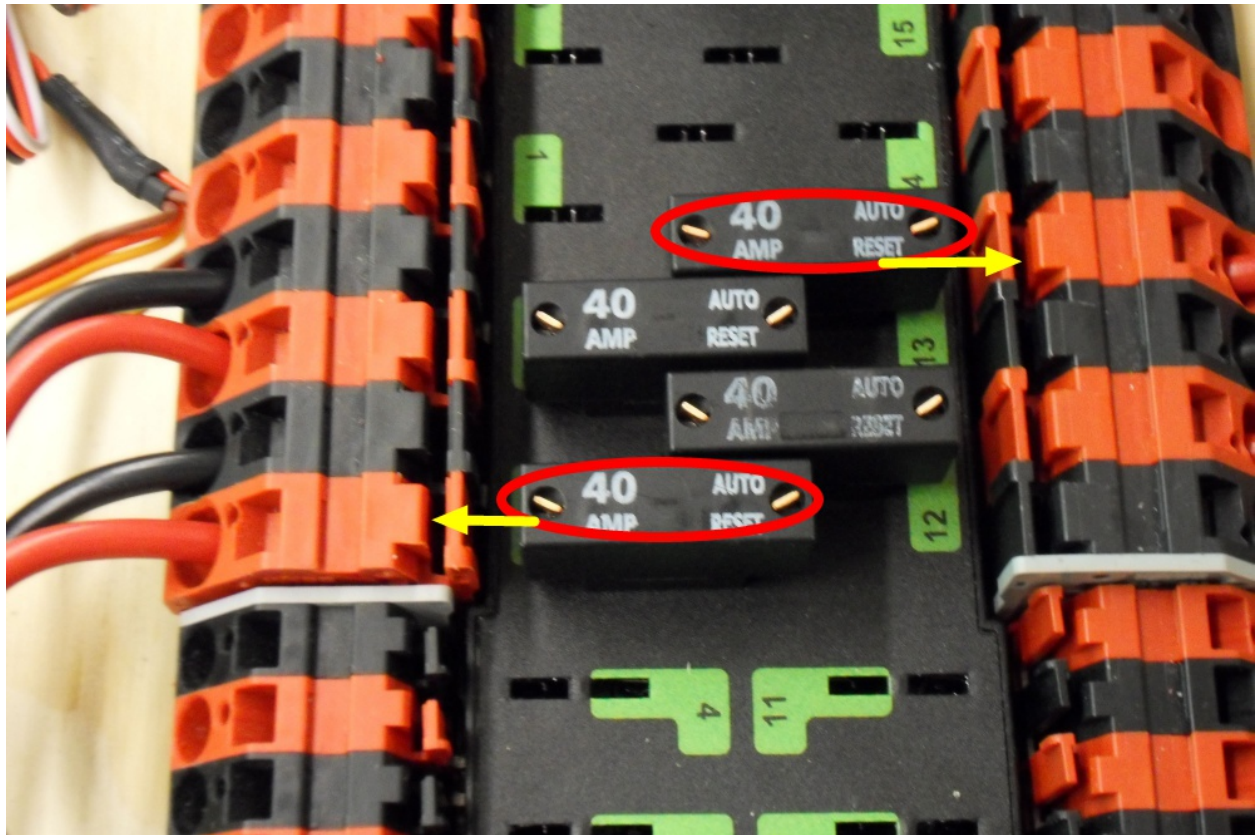


Requires: Wire stripper, 2 pin cable, Robot Signal Light, 18 AWG (1 mm^2) red wire, very small flat screwdriver

1. Cut one end off of the 2 pin cable and strip both wires
2. Insert the black wire into the center, “N” terminal and tighten the terminal.
3. Strip the 18 AWG (1 mm^2) red wire and insert into the “La” terminal and tighten the terminal.
4. Cut and strip the other end of the 18 AWG (1 mm^2) wire to insert into the “Lb” terminal
5. Insert the red wire from the two pin cable into the “Lb” terminal with the 18 AWG (1 mm^2) red wire and tighten the terminal.
6. Connect the two-pin connector to the RSL port on the roboRIO. The black wire should be closest to the outside of the roboRIO.

Tip: You may wish to temporarily secure the RSL to the control board using cable ties or Dual Lock (it is recommended to move the RSL to a more visible location as the robot is being constructed)

2.1.20 Circuit Breakers



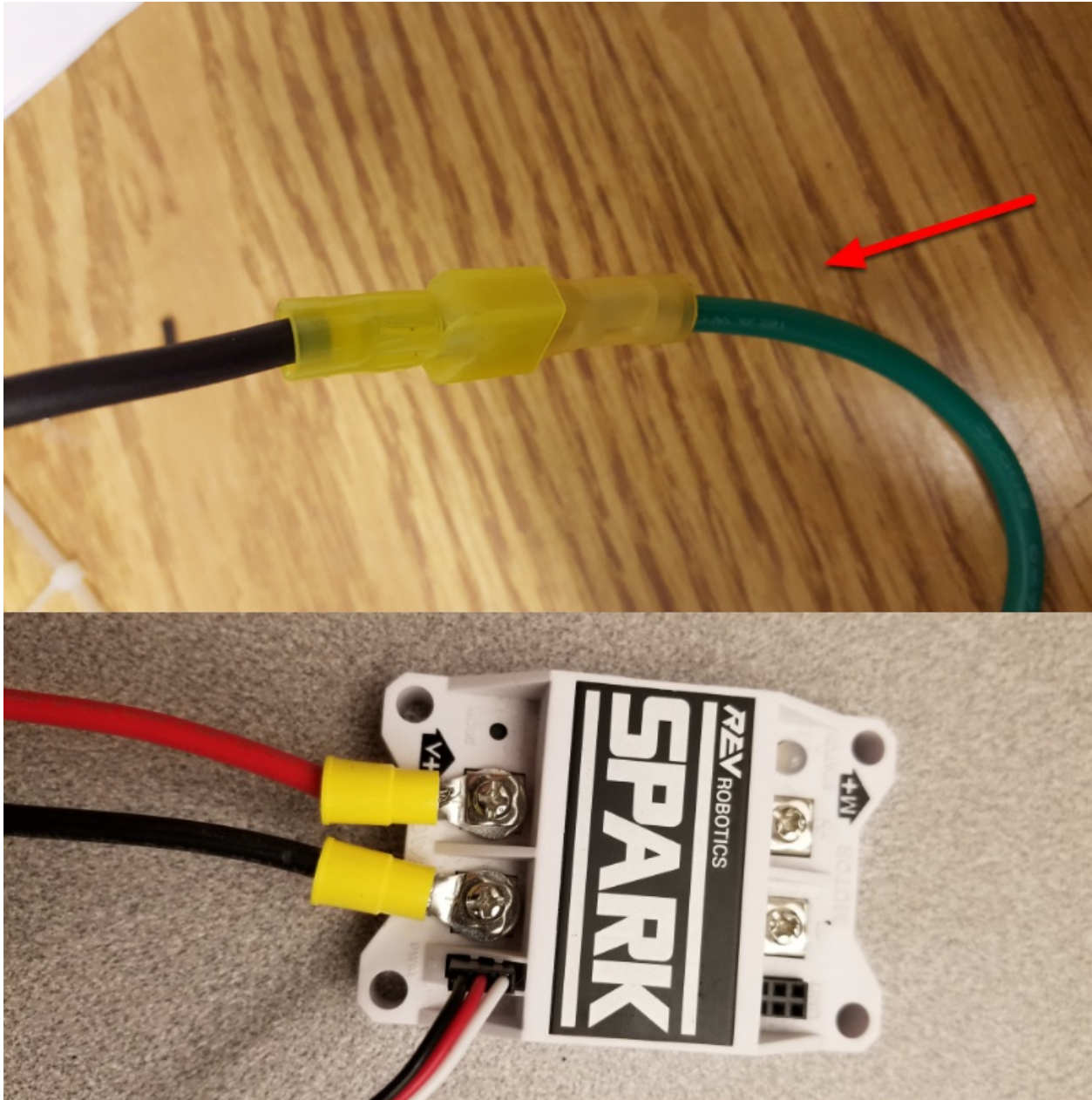
Requires: 4x 40A circuit breakers

Insert 40-amp Circuit Breakers into the positions on the PDP corresponding with the Wago connectors the Talons are connected to. Note that, for all breakers, the breaker corresponds

with the nearest positive (red) terminal (see graphic above). All negative terminals on the board are directly connected internally.

If working on a Robot Quick Build, stop here and insert the board into the robot chassis before continuing.

2.1.21 Motor Power



Requires: Wire stripper, wire crimper, phillips head screwdriver, wire connecting hardware
For each CIM motor:

- Strip the ends of the red and black wires from the CIM

For integrated wire controllers including SPARK MAX (top image):

1. Strip the red and black wires (or white and green wires) from the controller (the SPARK MAX white wire is unused for brushed motors such as the CIM, it should be secured and the end should be insulated such with electrical tape or other insulation method).
2. Connect the motor wires to the matching controller output wires (for controllers with white/green, connect red to white and green to black). The images above show an example using quick disconnect terminals which are provided in the Rookie KOP.

For the SPARK or other non-integrated-wire controllers (bottom image):

1. Crimp a ring/fork terminal on each of the motor wires.
2. Attach the wires to the output side of the motor controller (red to +, black to -)

2.1.22 STOP



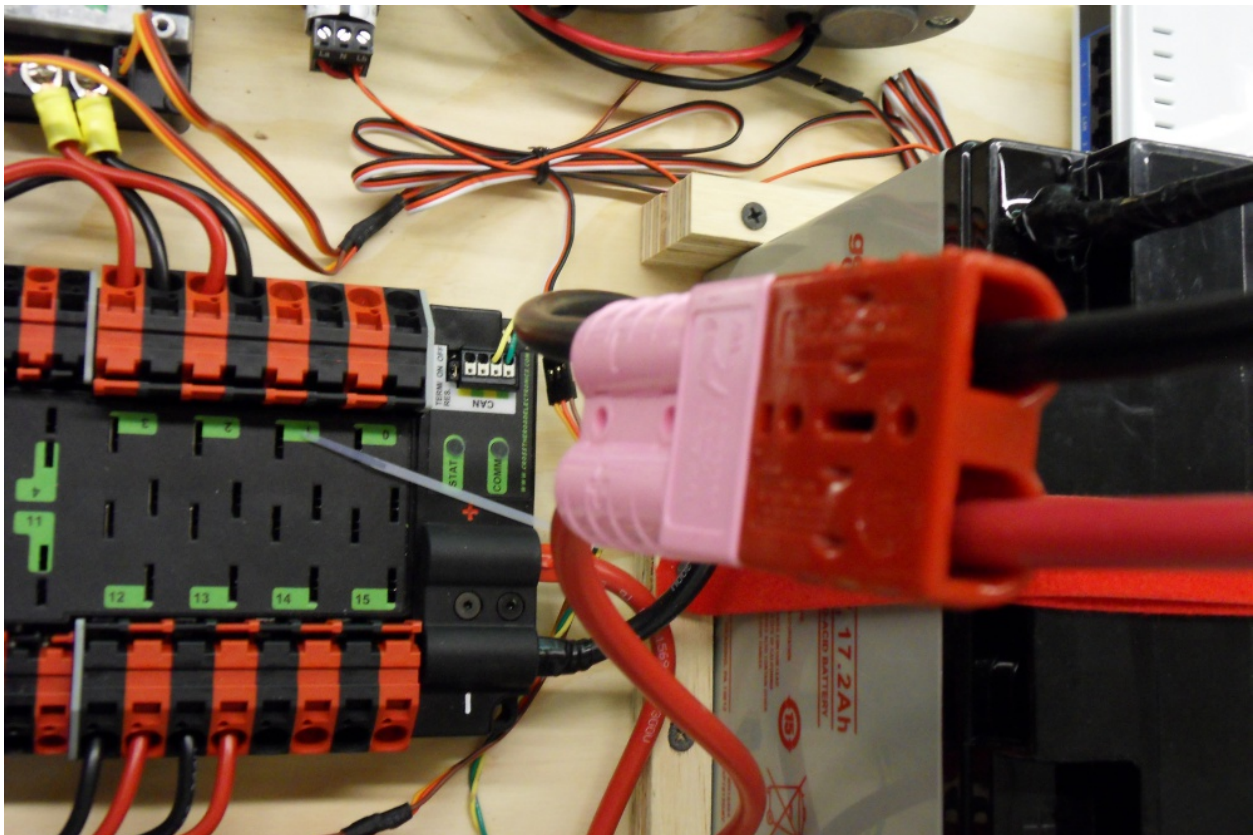
Danger: Before plugging in the battery, make sure all connections have been made with the proper polarity. Ideally have someone that did not wire the robot check to make sure all connections are correct.

Before plugging in the battery, make sure all connections have been made with the proper polarity. Ideally have someone that did not wire the robot check to make sure all connections are correct.

- Start with the battery and verify that the red wire is connected to the positive terminal
- Check that the red wire passes through the main breaker and to the + terminal of the PDP and that the black wire travels directly to the - terminal.
- For each motor controller, verify that the red wire goes from the red PDP terminal to the Red wire on the Victor SPX (not the white M+!!!!)
- For each device on the end of the PDP, verify that the red wire connects to the red terminal on the PDP and the red terminal on the component.
- Make sure that the orange Passive PoE cable is plugged directly into the radio NOT THE roboRIO! It must be connected to the roboRIO using an additional Ethernet cable.

Tip: It is also recommended to put the robot on blocks so the wheels are off the ground before proceeding. This will prevent any unexpected movement from becoming dangerous.

2.1.23 Manage Wires



Requires: Zip ties

Tip: Now may be a good time to add a few zip ties to manage some of the wires before

proceeding. This will help keep the robot wiring neat.

2.1.24 Connect Battery

Connect the battery to the robot side of the Anderson connector. Power on the robot by moving the lever on the top of the 120A main breaker into the ridge on the top of the housing.

If stuff blinks, you probably did it right.

Before moving on, if using SPARK MAX controllers, there is one more configuration step to complete. The SPARK MAX motor controllers are configured to control a brushless motor by default. You can verify this by checking that the light on the controller is blinking either cyan or magenta (indicating brushless brake or brushless coast respectively). To change to brushed mode, press and hold the mode button for 3-4 seconds until the status LED changes color. The LED should change to either blue or yellow, indicating that the controller is in brushed mode (brake or coast respectively). To change the brake or coast mode, which control how quickly the motor slows down when a neutral signal is applied, press the mode button briefly.

Tip: For more information on the SPARK MAX motor controllers, including how to test your motors/controllers without writing any code by using the REV Hardware Client, see the [SPARK MAX Quickstart guide](#).

From here, you should connect to the roboRIO and try uploading your code!

Step 2: Installing Software

An overview of the available control system software can be found [here](#).

3.1 Offline Installation Preparation

This article contains instructions/links to components you will want to gather if you need to do offline installation of the FRC® Control System software.

Tip: This document compiles all the download links from the following documents to make it easier to install on offline computers or on multiple computers. If you are installing on a single computer that is connected to the internet, you can skip this page.

Note: The order in which these tools are installed does not matter for Java and C++ teams. LabVIEW should be installed before the FRC Game Tools or 3rd Party Libraries.

3.1.1 Documentation

This documentation can be downloaded for offline viewing. The link to download the PDF can be found [here](#).

3.1.2 Installers

All Teams

- [2021 FRC Game Tools](#) (Note: Click on link for “Individual Offline Installers”)
- [2020 FRC Radio Configuration Utility](#) or [2020 FRC Radio Configuration Utility Israel Version](#)
- (Optional - Veterans Only!) [Classmate/Acer PC Image](#)

Note: There are no changes to the radio tool for the 2021 season so the 20.0.0 version remains the latest available.

LabVIEW Teams

- LabVIEW USB (from *FIRST*® Choice) or [Download](#) (Note: Click on link for “Individual Offline Installers”)

Note: The dropdown on this page will say 2020 because the 2021 base LabVIEW version is identical to what was used in 2020. If you already have the 2020 version installed see [Re-licensing LabVIEW for 2021 Season](#) for info on re-licensing your installation.

Java/C++ Teams

- [Java/C++ WPILib Installer](#)

Note: After downloading the Java/C++ WPILib installer, run it once while connected to the internet and download VS Code for all platforms and save the downloaded VS Code zip file for future offline installations.

3.1.3 3rd Party Libraries/Software

A directory of available 3rd party software that plugs in to WPILib can be found on [3rd Party Libraries](#).

3.2 Installing LabVIEW for FRC (LabVIEW only)



Note: This installation is for teams programming in LabVIEW or using NI Vision Assistant only. C++ and Java teams not using these features do not need to install LabVIEW and should proceed to [Installing the FRC Game Tools](#).

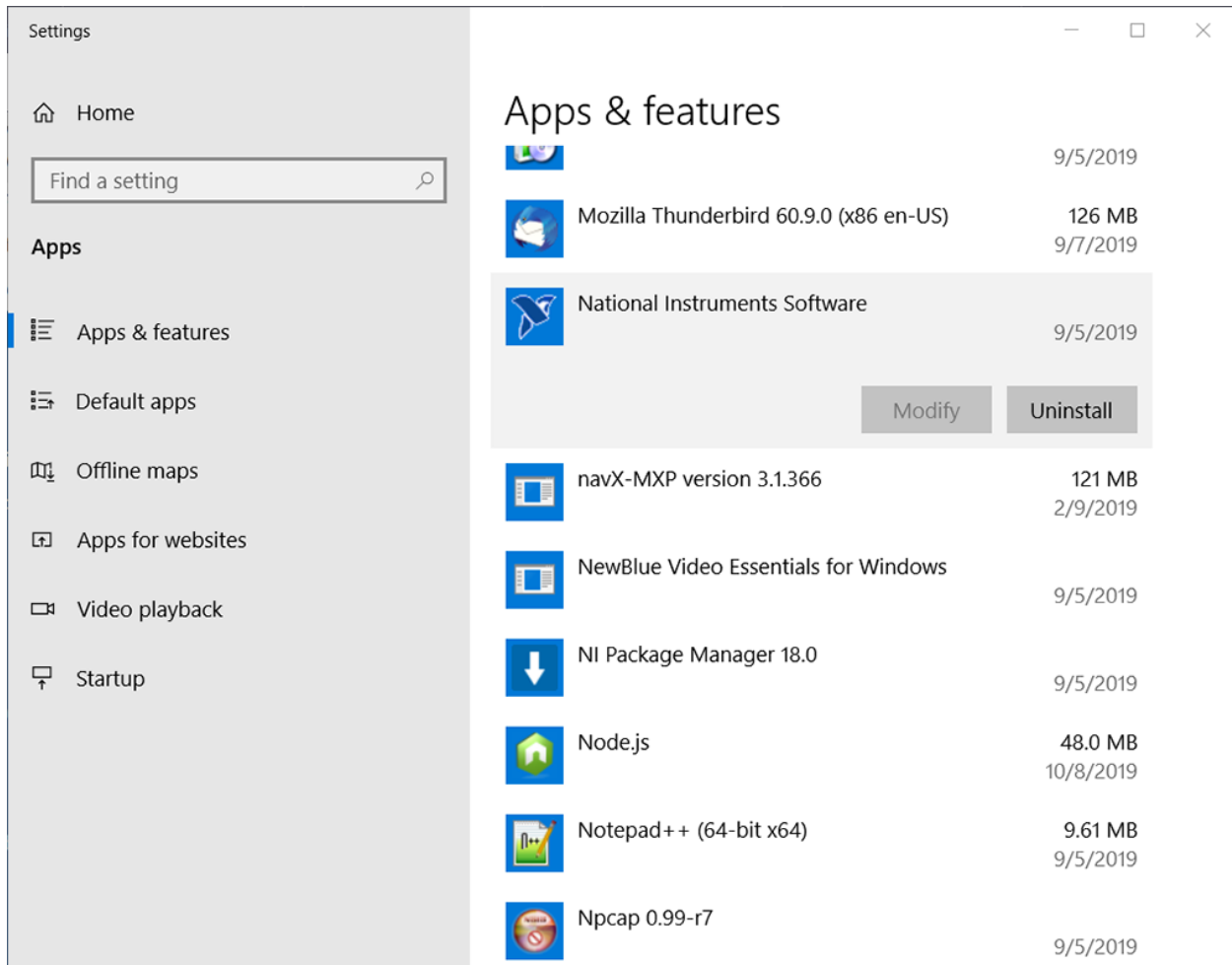
Download and installation times will vary widely with computer and internet connection specifications, however note that this process involves a large file download and installation and will likely take at least an hour to complete.

Warning: The version of LabVIEW used for the 2021 season is the same as the version used for the 2020 season. If you already have LabVIEW installed, it does not need to be re-installed. Instead see [Re-licensing LabVIEW for 2021 Season](#) for info on re-licensing your software.

3.2.1 Uninstall Old Versions (Recommended)

Note: If you wish to keep programming cRIOs you will need to maintain an install of LabVIEW for FRC® 2014. The LabVIEW for FRC 2014 license has been extended. While these versions should be able to co-exist on a single computer, this is not a configuration that has been extensively tested.

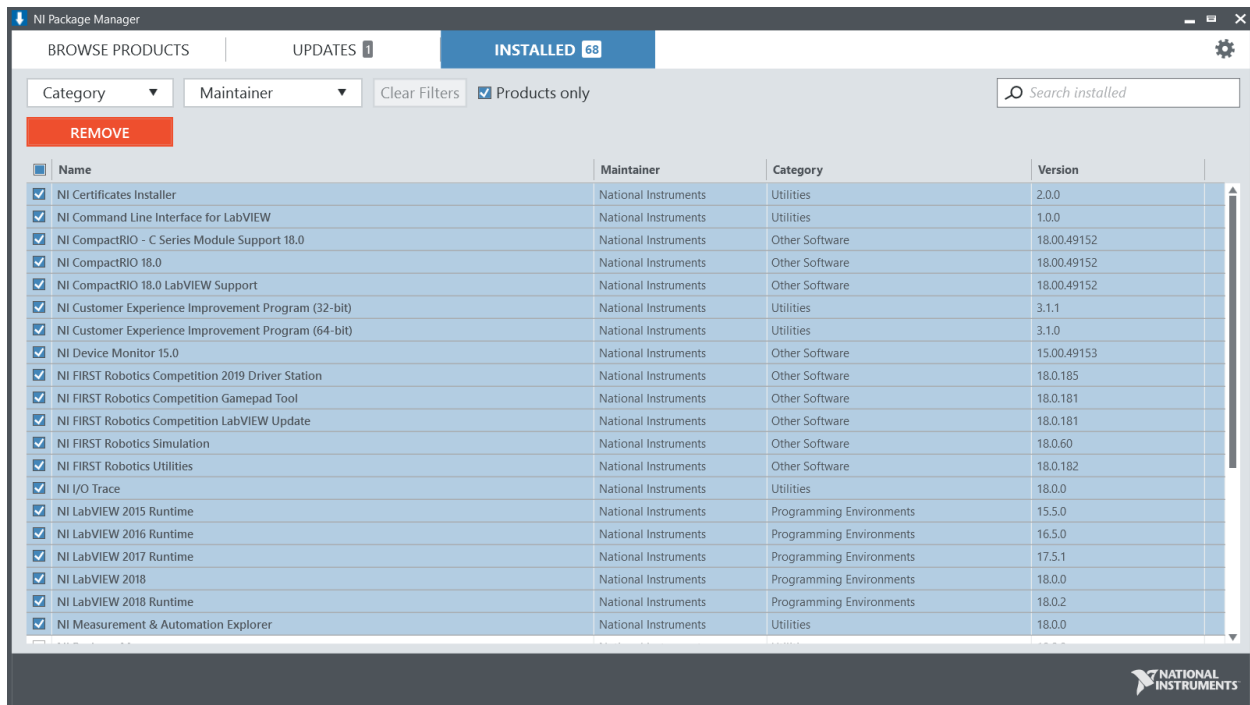
Before installing the new version of LabVIEW it is recommended to remove any old versions. The new version will likely co-exist with the old version, but all testing has been done with FRC 2020 only. Make sure to back up any team code located in the “User\LabVIEW Data” directory before un-installing. Then click Start >> Add or Remove Programs. Locate the entry labeled “National Instruments Software”, and select Uninstall.



Select Components to Uninstall

In the dialog box that appears, select all entries. The easiest way to do this is to de-select the “Products Only” check-box and select the check-box to the left of “Name”. Click Remove. Wait for the uninstaller to complete and reboot if prompted.

Warning: These instructions assume that no other NI software is installed. If you have other NI software installed, it is necessary to uncheck the software that should not be uninstalled.



3.2.2 Getting LabVIEW installer

Either locate and insert the LabVIEW USB Drive or download the [LabVIEW for FRC 2020 installer](#) from NI. Be sure to select the correct version from the drop-down.

DOWNLOAD

Online installer

File Size

5.37 MB

Note: If you need to download individual versions or patches, you can select from [Individual Offline Installers](#)

Offline Installer

If you wish to install on other machines offline, do not click the Download button, click **Individual Offline Installers** and then click Download, to download the full installer.

Note: This is a large download (~8GB). It is recommended to use a fast internet connection and to use the NI Downloader to allow the download to resume if interrupted.

3.2.3 Installing LabVIEW

NI LabVIEW requires a license. Each season's license is active until January 31st of the following year (e.g. the license for the 2020 season expires on January 31, 2021)

Teams are permitted to install the software on as many team computers as needed, subject to the restrictions and license terms that accompany the applicable software, and provided that only team members or mentors use the software, and solely for FRC. Rights to use LabVIEW are governed solely by the terms of the license agreements that are shown during the installation of the applicable software.

Starting Install

Online Installer

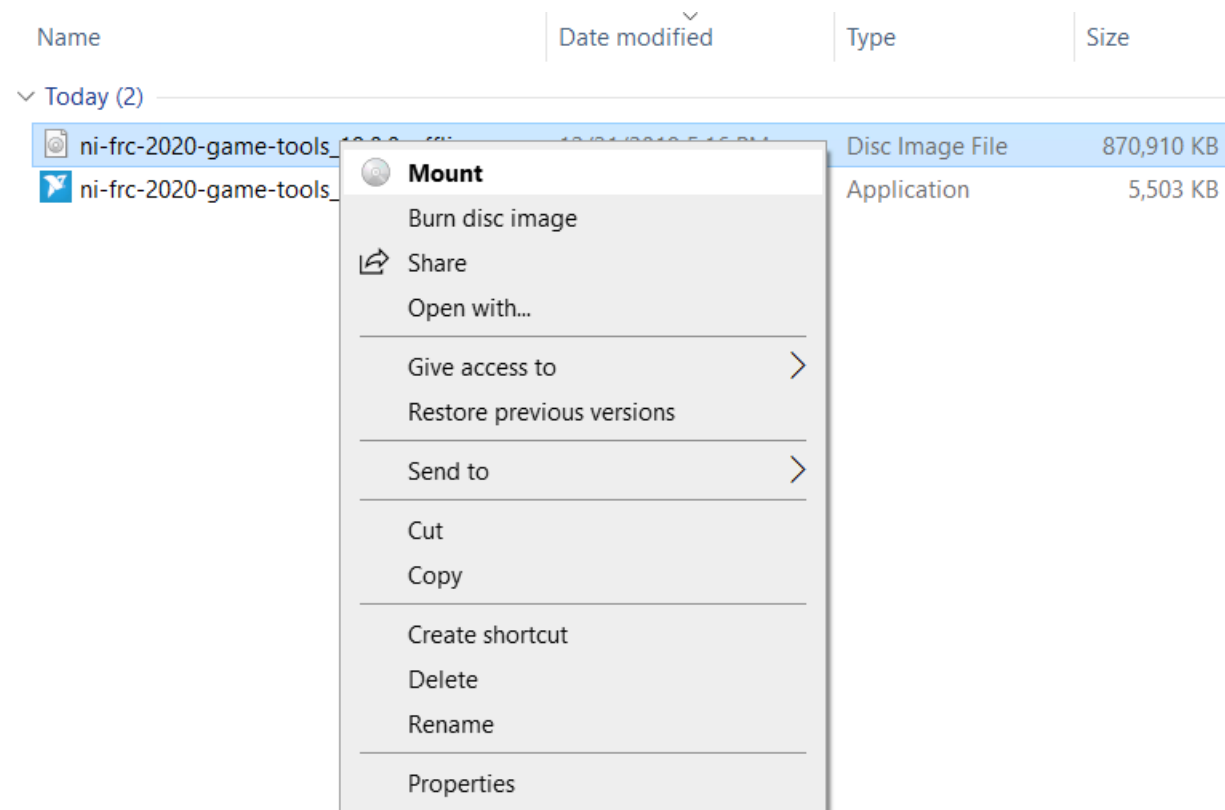
Offline Installer (Windows 10)

Offline Installer (Windows 7, 8, & 8.1)

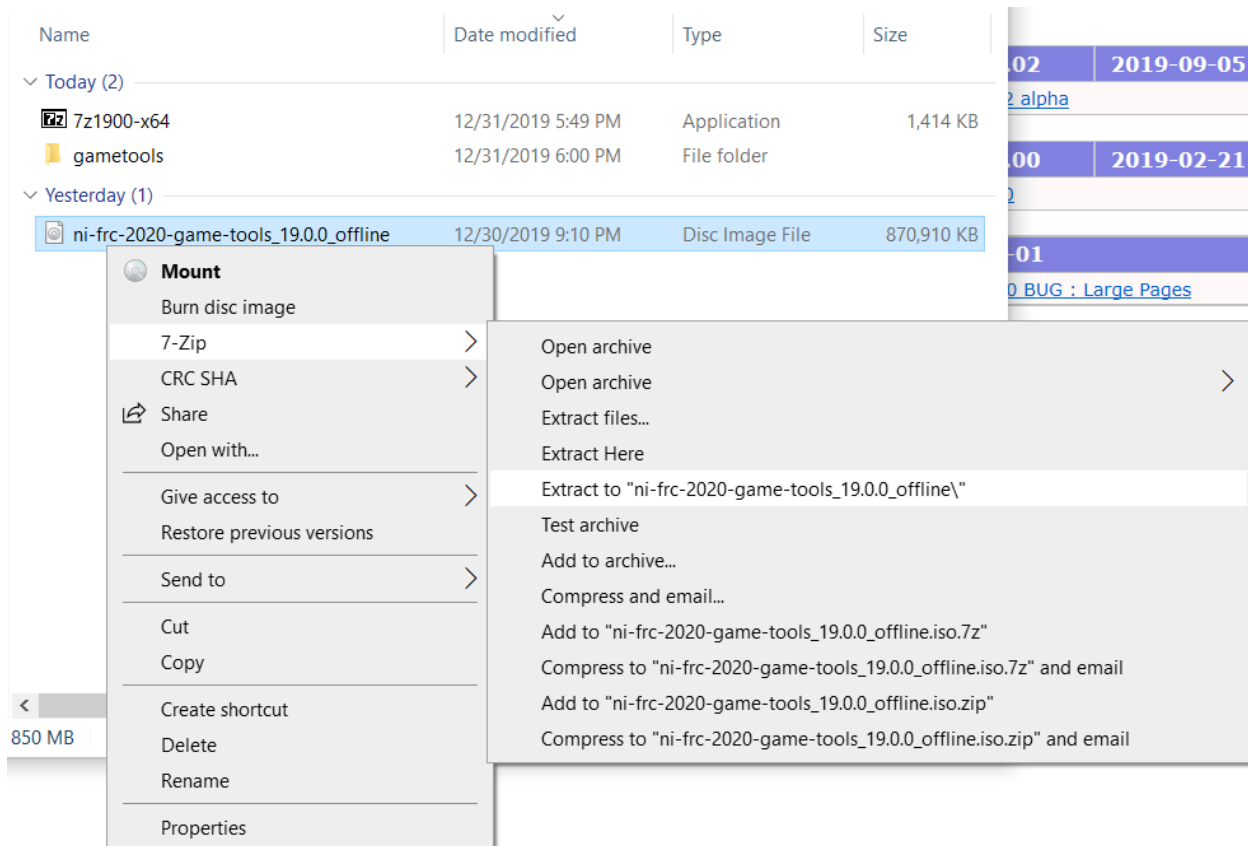
Run the downloaded exe file to start the install process. Click Yes if a Windows Security prompt

Right click on the downloaded iso file and select mount. Run install.exe from the mounted iso. Click "Yes" if a Windows Security prompt

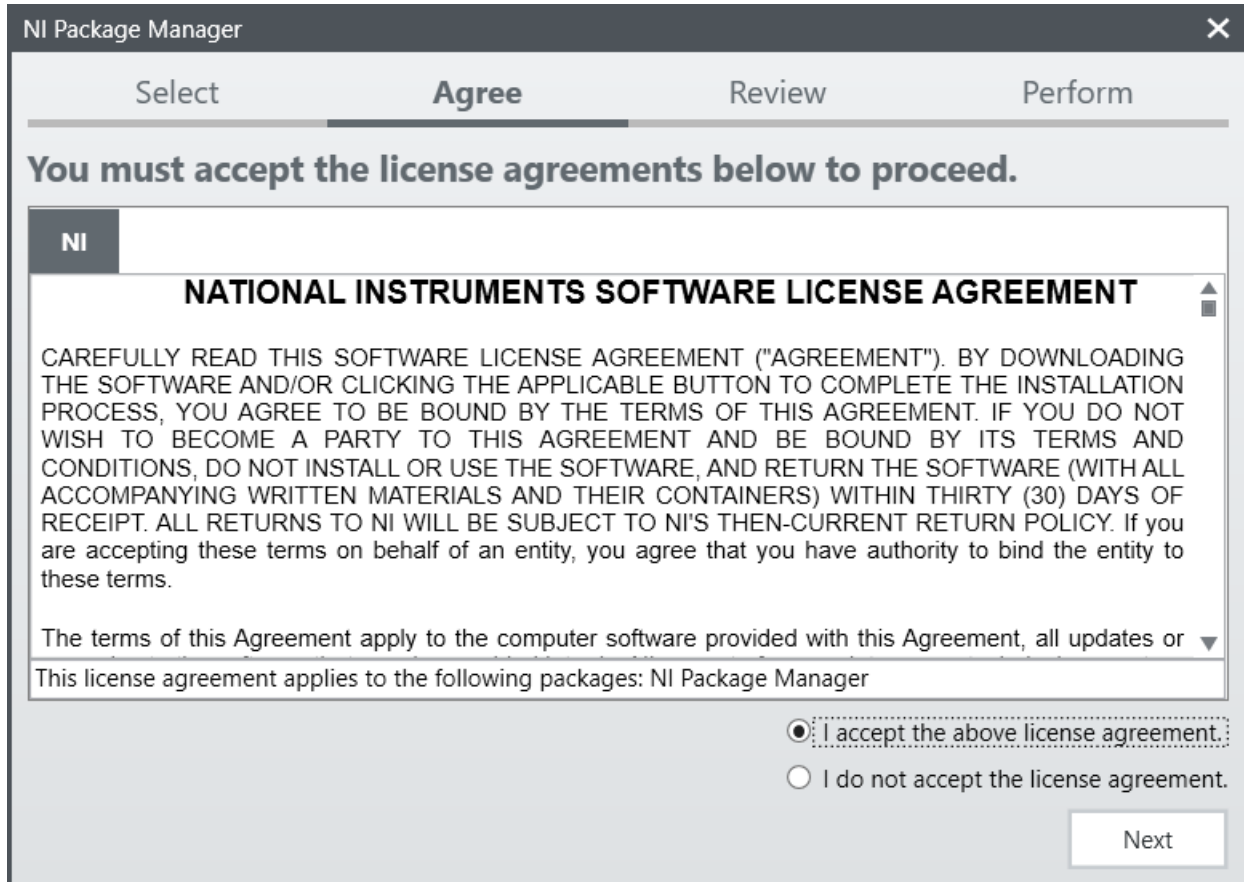
Note: other installed programs may associate with iso files and the mount option may not appear. If that software does not give the option to mount or extract the iso file, then follow the directions in the "Offline Installer (Windows 7, 8, & 8.1)" tab.



Install 7-Zip (download [here](#)). As of the writing of this document, the current released version is 19.00 (2019-02-21). Right click on the downloaded iso file and select Extract to.



Run install.exe from the extracted folder. Click Yes if a Windows Security prompt appears.

NI Package Manager License

The image shows a screenshot of the 'NI Package Manager' window. At the top, there is a title bar with the text 'NI Package Manager' and a close button. Below the title bar is a navigation bar with four tabs: 'Select', 'Agree', 'Review', and 'Perform'. The 'Agree' tab is currently selected. The main content area has a heading 'You must accept the license agreements below to proceed.' followed by a list of license agreements. The first agreement is from 'NI' (National Instruments) and is titled 'NATIONAL INSTRUMENTS SOFTWARE LICENSE AGREEMENT'. The text of the agreement is displayed in a scrollable area. Below the agreement text, there is a section for 'The terms of this Agreement apply to the computer software provided with this Agreement, all updates or...' and a list of packages: 'NI Package Manager'. At the bottom right, there are two radio buttons: 'I accept the above license agreement.' (which is selected) and 'I do not accept the license agreement.'. A 'Next' button is located at the bottom right of the window.

NI Package Manager

Select Agree Review Perform

You must accept the license agreements below to proceed.

NI

NATIONAL INSTRUMENTS SOFTWARE LICENSE AGREEMENT

CAREFULLY READ THIS SOFTWARE LICENSE AGREEMENT ("AGREEMENT"). BY DOWNLOADING THE SOFTWARE AND/OR CLICKING THE APPLICABLE BUTTON TO COMPLETE THE INSTALLATION PROCESS, YOU AGREE TO BE BOUND BY THE TERMS OF THIS AGREEMENT. IF YOU DO NOT WISH TO BECOME A PARTY TO THIS AGREEMENT AND BE BOUND BY ITS TERMS AND CONDITIONS, DO NOT INSTALL OR USE THE SOFTWARE, AND RETURN THE SOFTWARE (WITH ALL ACCOMPANYING WRITTEN MATERIALS AND THEIR CONTAINERS) WITHIN THIRTY (30) DAYS OF RECEIPT. ALL RETURNS TO NI WILL BE SUBJECT TO NI'S THEN-CURRENT RETURN POLICY. If you are accepting these terms on behalf of an entity, you agree that you have authority to bind the entity to these terms.

The terms of this Agreement apply to the computer software provided with this Agreement, all updates or

This license agreement applies to the following packages: NI Package Manager

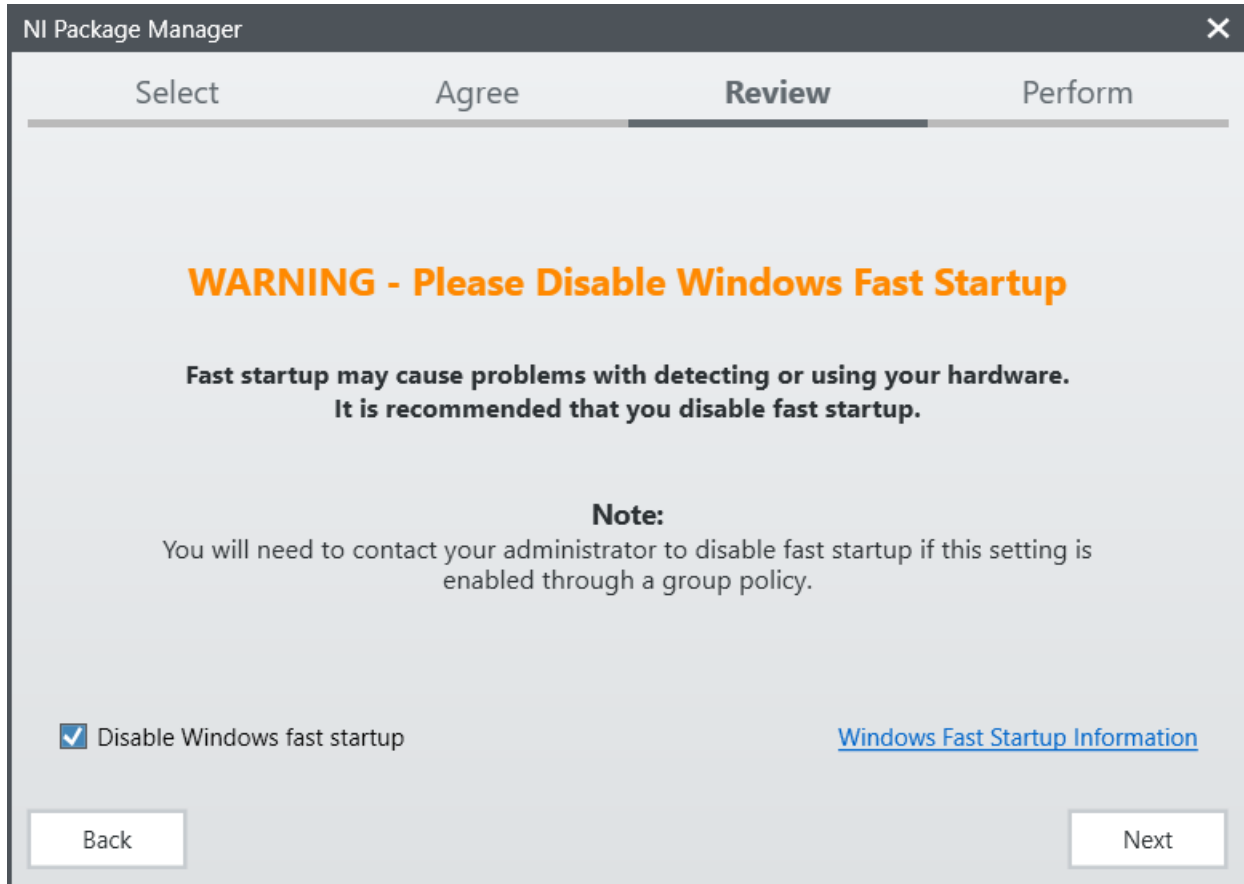
☒ I accept the above license agreement.

☐ I do not accept the license agreement.

Next

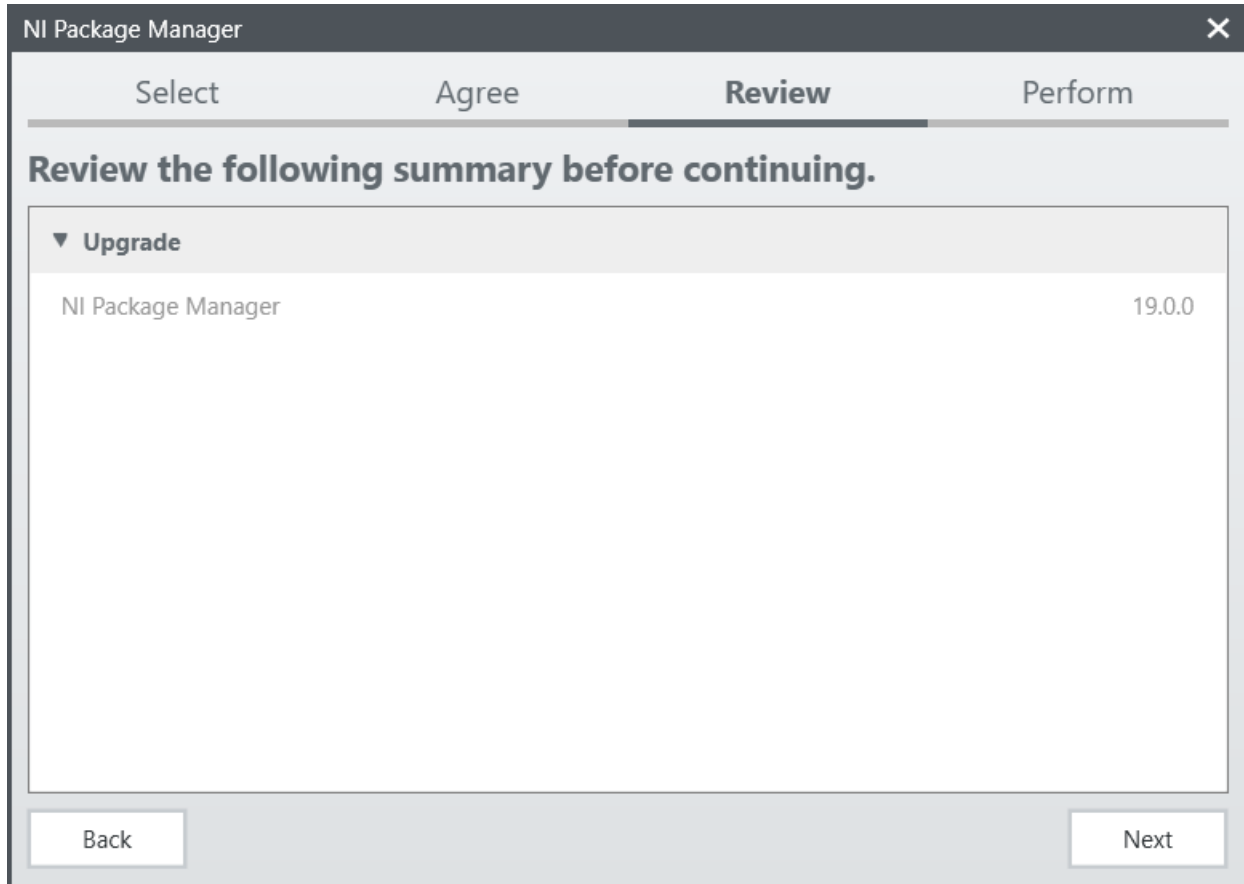
If you see this screen, click *Next*

Disable Windows Fast Startup



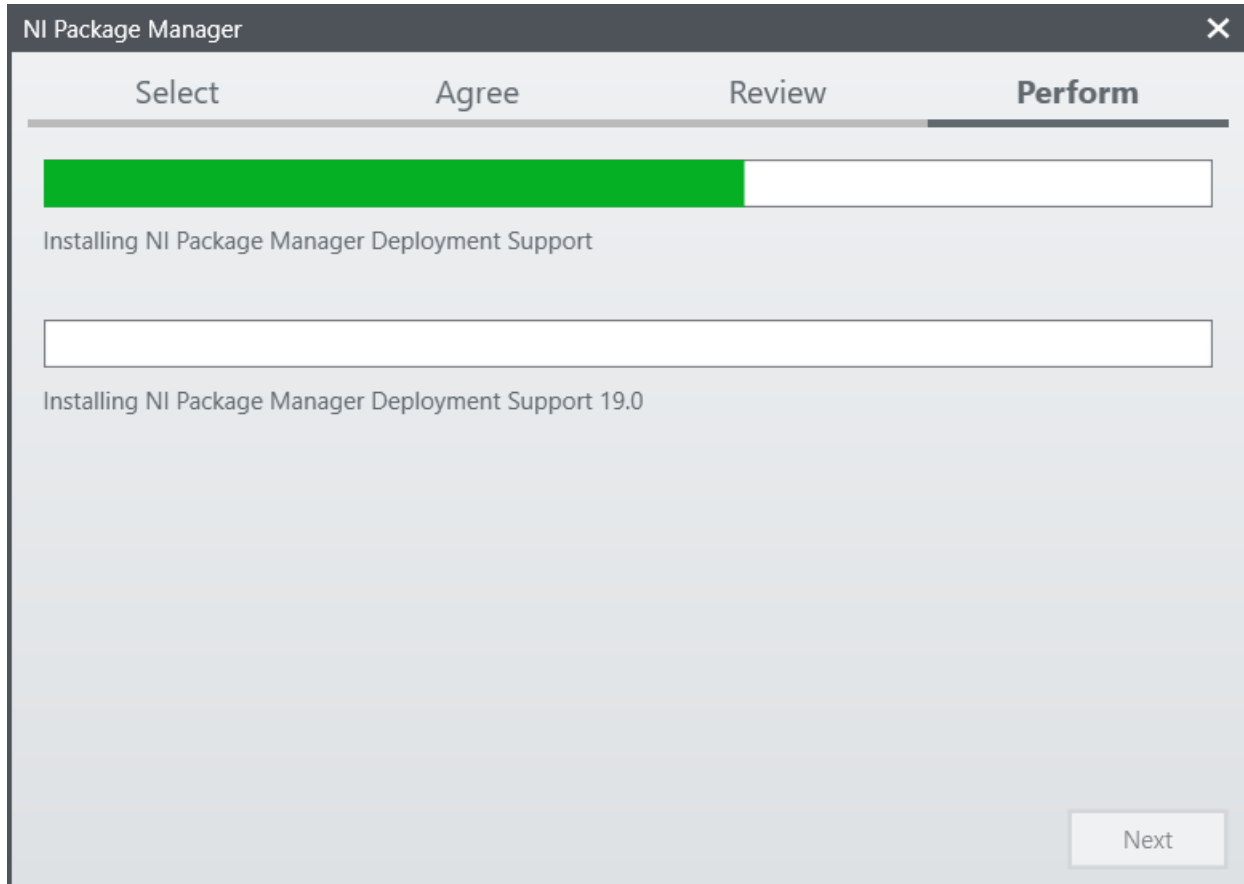
If you see this screen, click *Next*

NI Package Manager Review



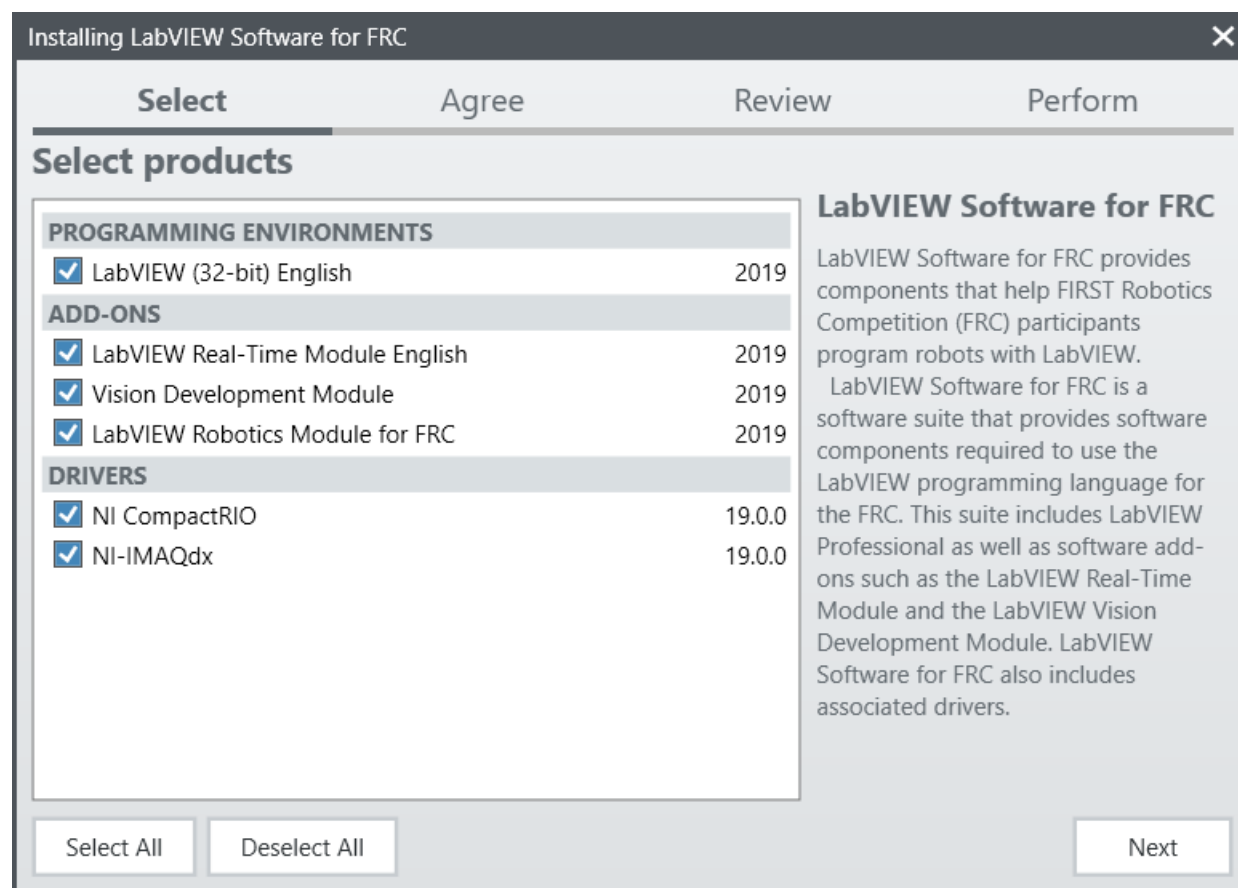
If you see this screen, click *Next*

NI Package Manager Installation



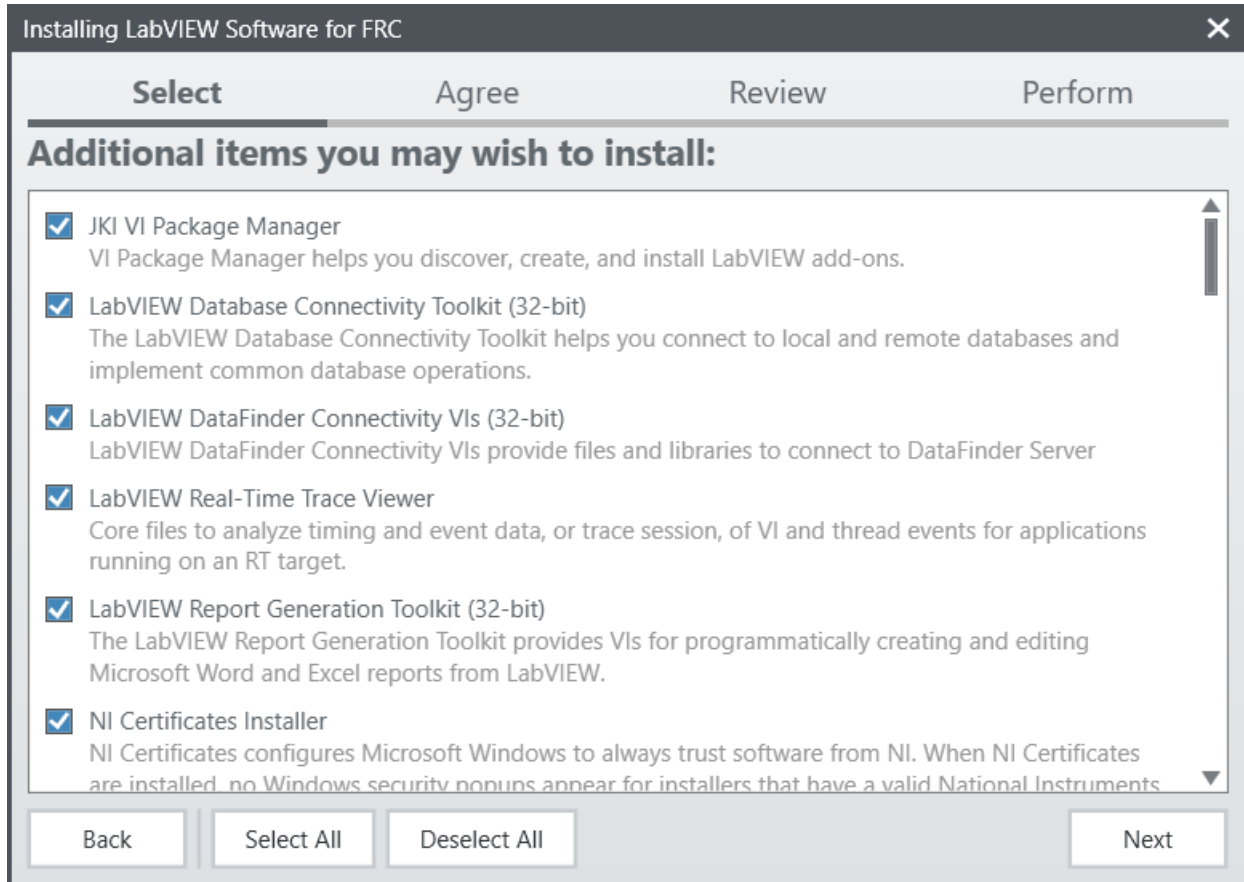
Installation progress of the NI Package Manager will be tracked in this window

Product List



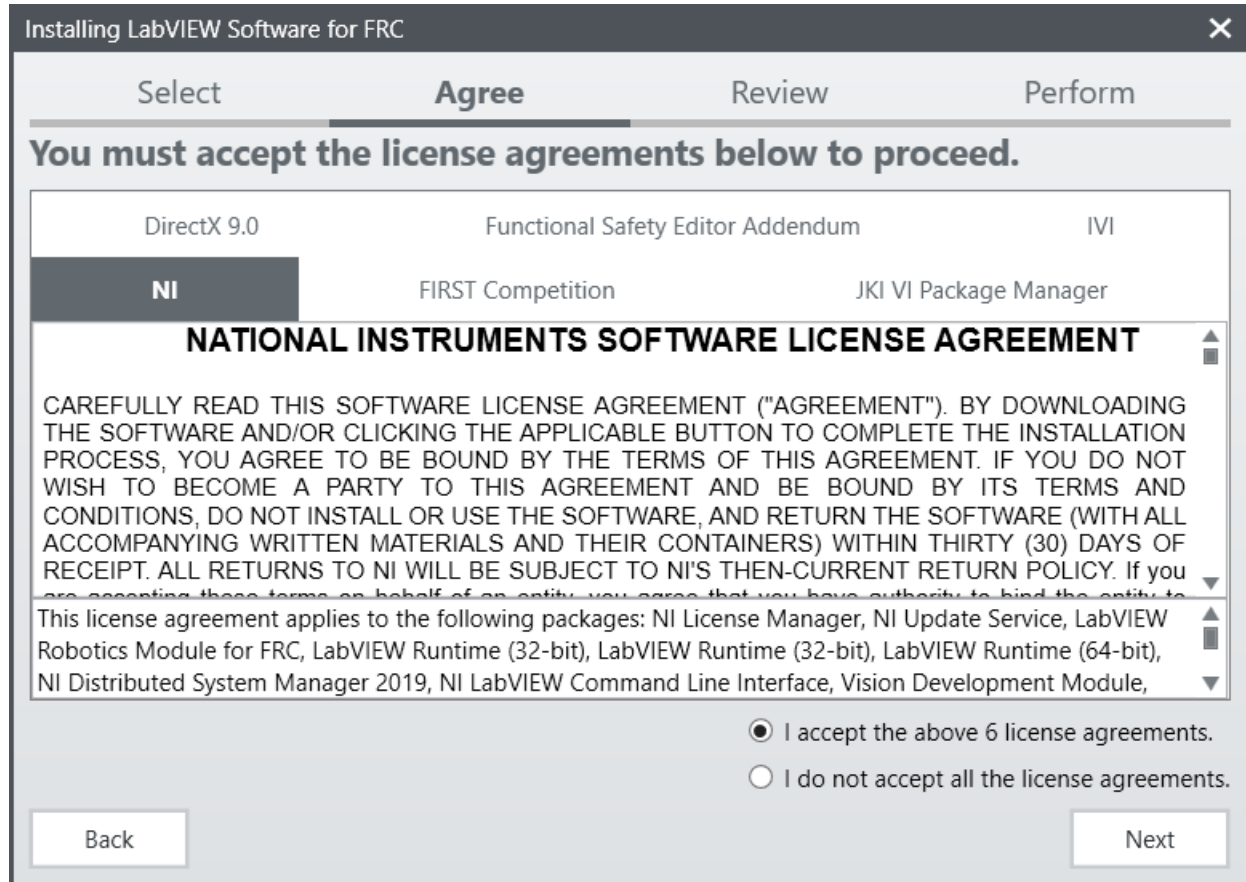
Click *Next*

Additional Packages

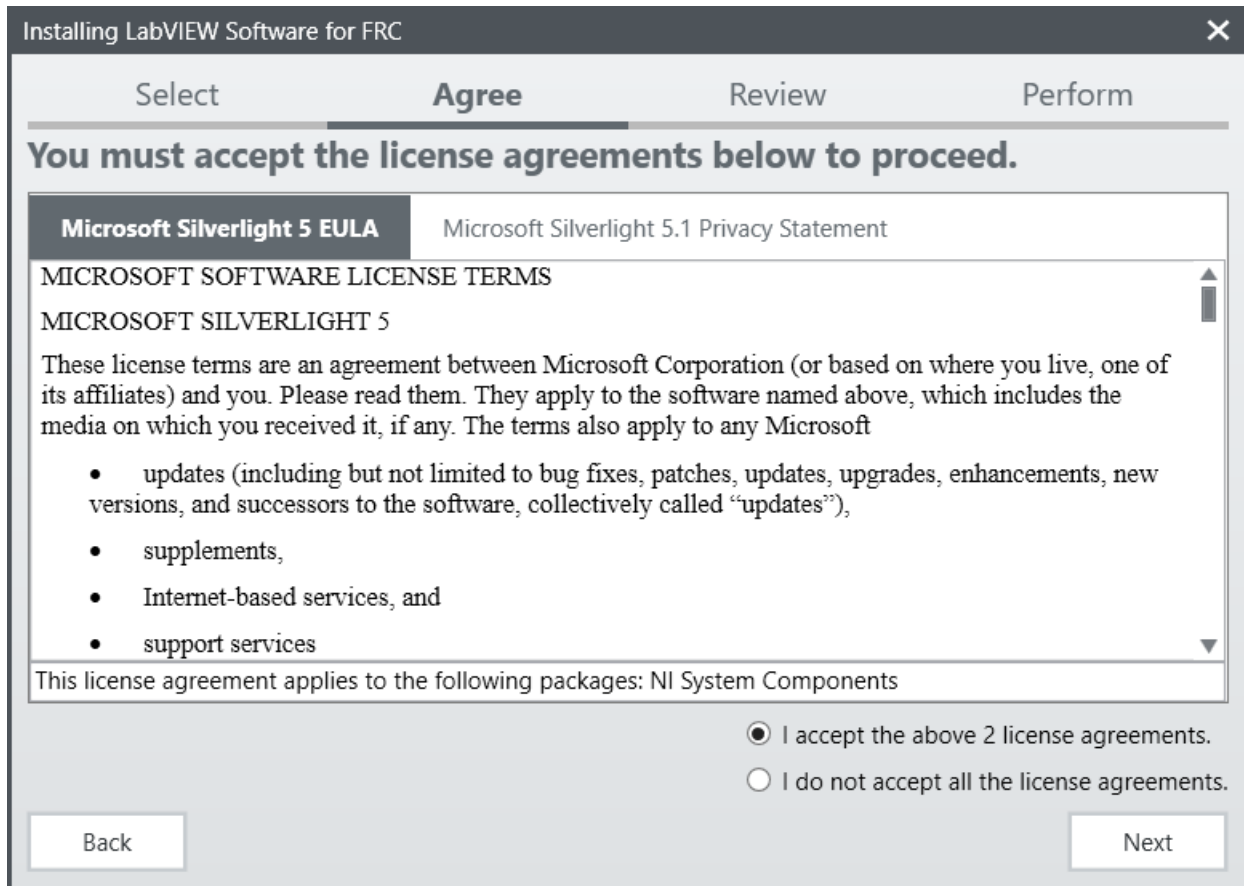


Click *Next*

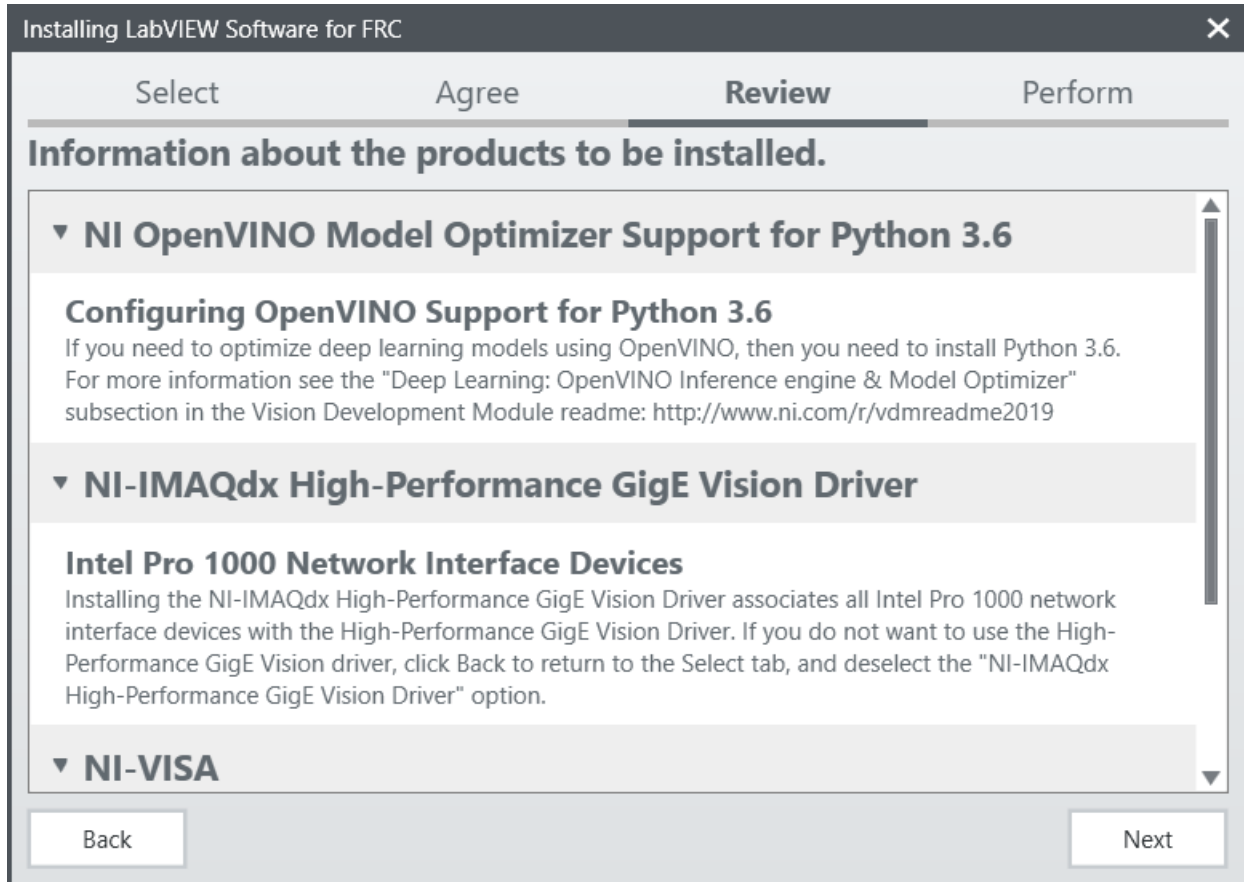
License agreements



Check "I accept..." then Click *Next*

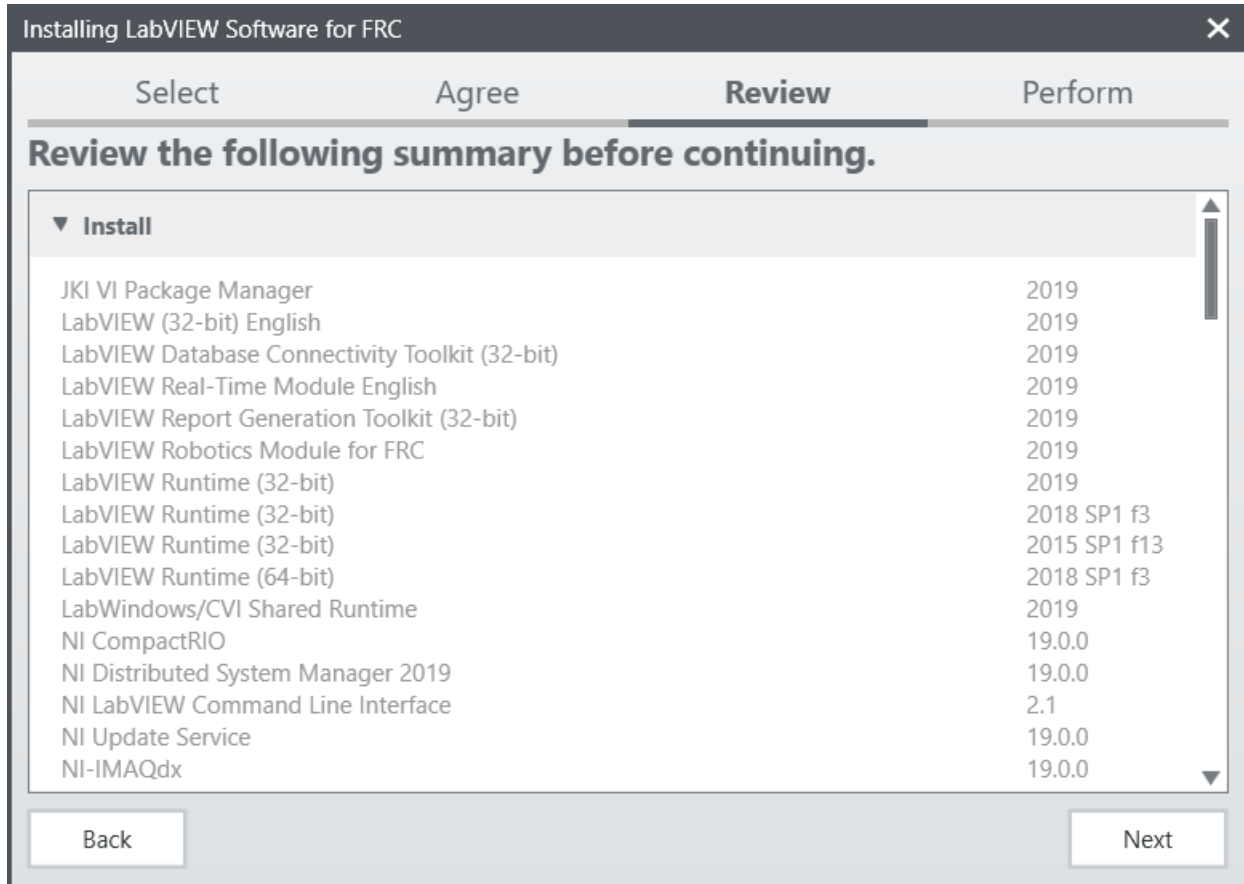


Check “I accept...” then Click *Next*

Product Information

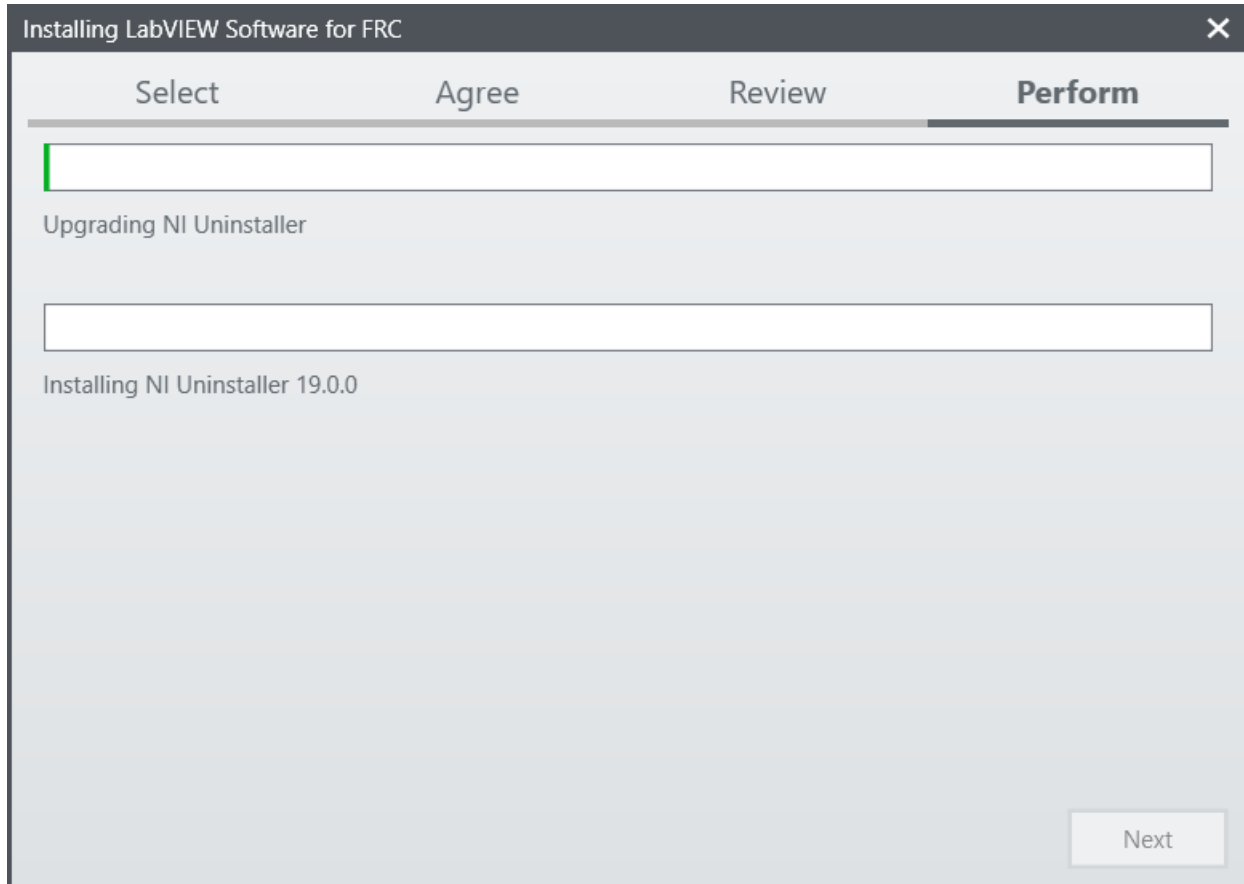
Click *Next*

Start Installation



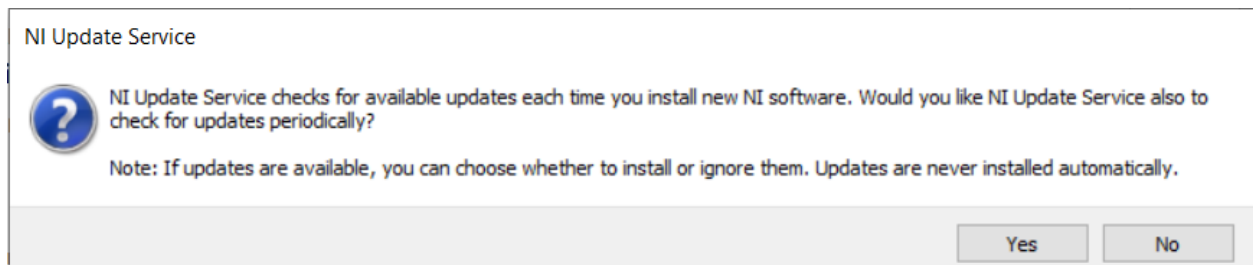
Click *Next*

Overall Progress



Overall installation progress will be tracked in this window

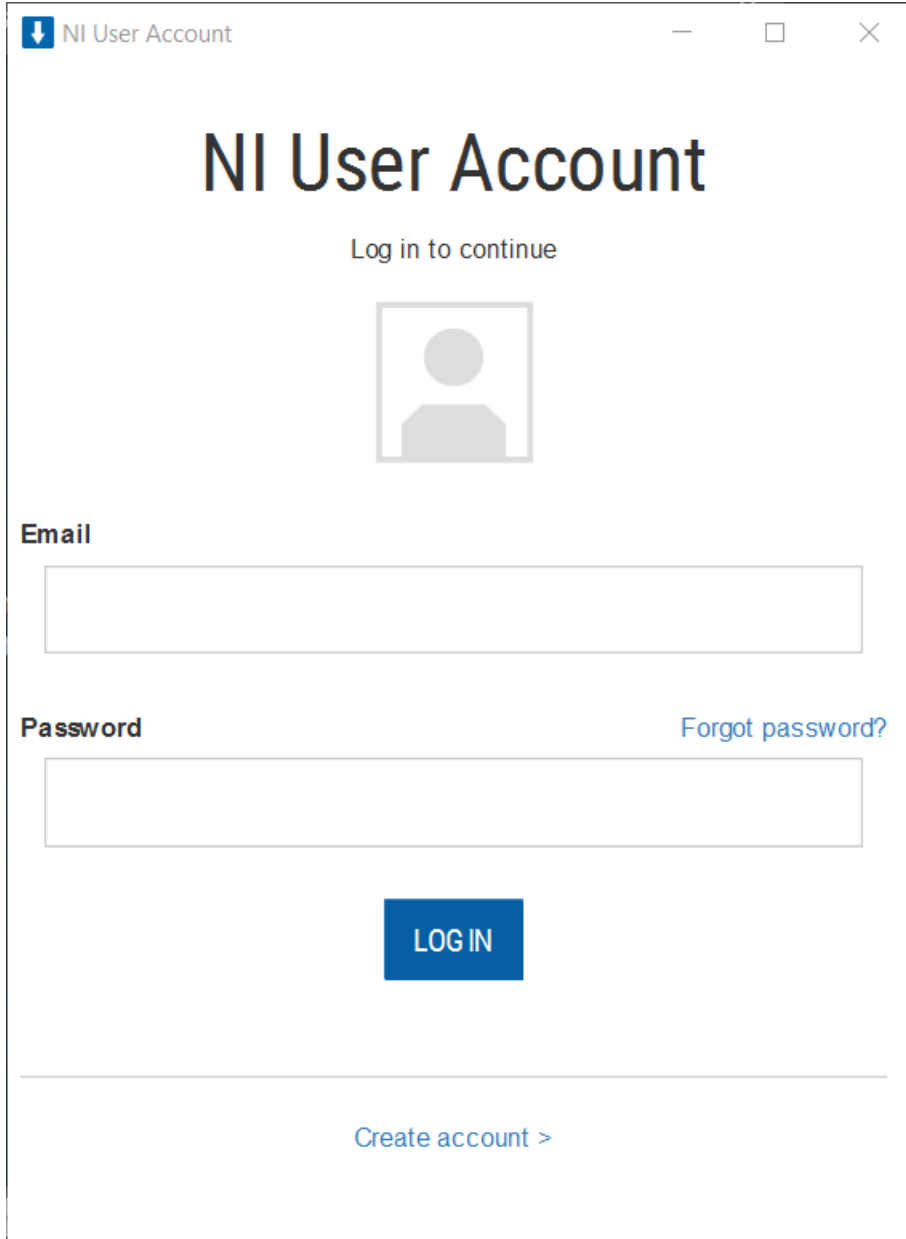
3.2.4 NI Update Service



You will be prompted whether to enable the NI update service. You can choose to not enable the update service.

Warning: It is not recommended to install these updates unless directed by FRC through our usual communication channels (FRC Blog, Team Updates or E-mail Blasts).


NI Activation Wizard

A screenshot of the NI User Account login window. The window has a title bar with a blue square icon containing a white downward arrow, followed by the text "NI User Account". The main content area has a large heading "NI User Account" in a dark blue font. Below the heading is the text "Log in to continue". Underneath is a placeholder for a profile picture, represented by a gray circle and a gray rectangle. Below the profile picture are two input fields: "Email" and "Password". To the right of the "Password" field is a blue link "Forgot password?". Below the input fields is a blue button with the text "LOG IN". At the bottom of the window is a blue link "Create account >".

NI User Account

NI User Account

Log in to continue



Email

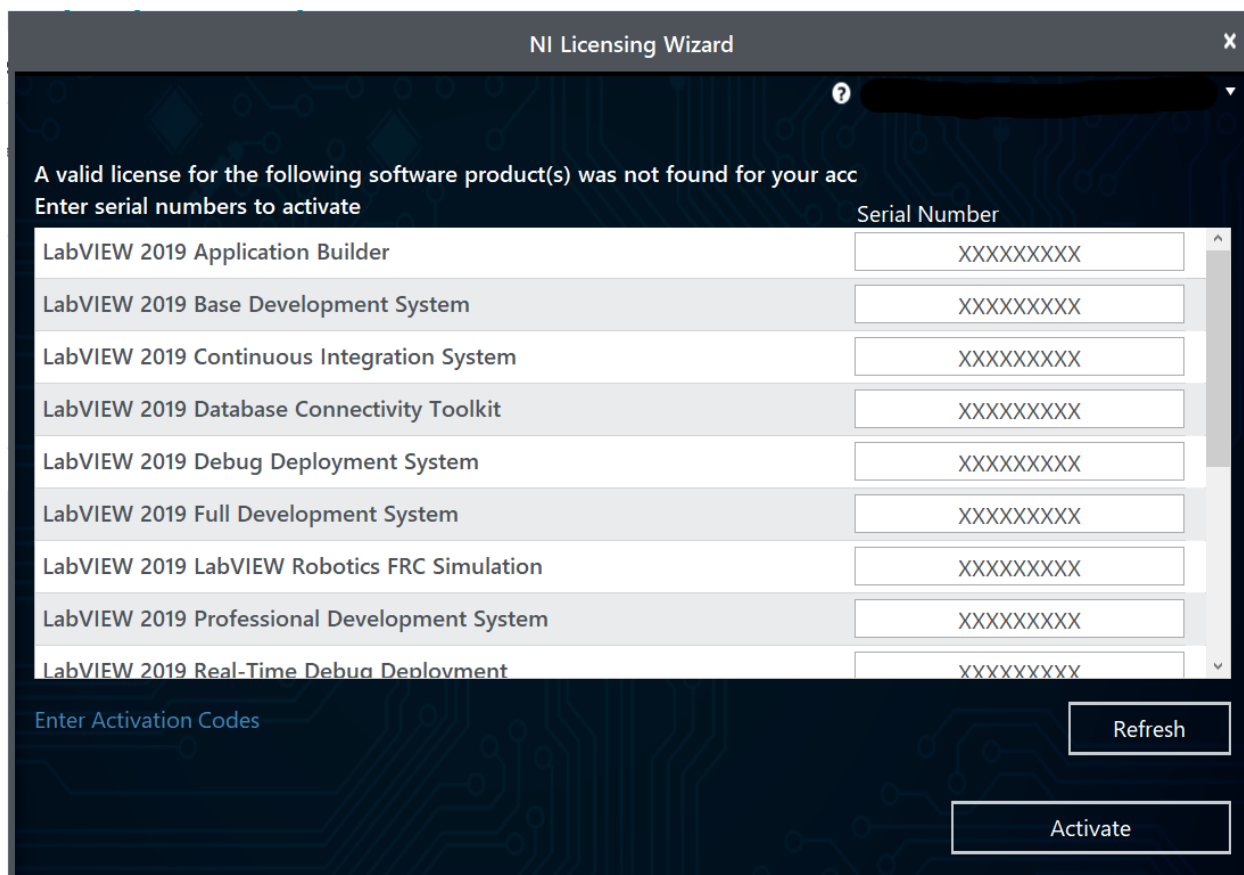
Password

[Forgot password?](#)

LOG IN

[Create account >](#)

Log into your ni.com account. If you don't have an account, select *Create account* to create a free account.



NI Licensing Wizard

A valid license for the following software product(s) was not found for your acc
Enter serial numbers to activate

	Serial Number
LabVIEW 2019 Application Builder	XXXXXXXXXX
LabVIEW 2019 Base Development System	XXXXXXXXXX
LabVIEW 2019 Continuous Integration System	XXXXXXXXXX
LabVIEW 2019 Database Connectivity Toolkit	XXXXXXXXXX
LabVIEW 2019 Debug Deployment System	XXXXXXXXXX
LabVIEW 2019 Full Development System	XXXXXXXXXX
LabVIEW 2019 LabVIEW Robotics FRC Simulation	XXXXXXXXXX
LabVIEW 2019 Professional Development System	XXXXXXXXXX
LabVIEW 2019 Real-Time Debug Deployment	XXXXXXXXXX

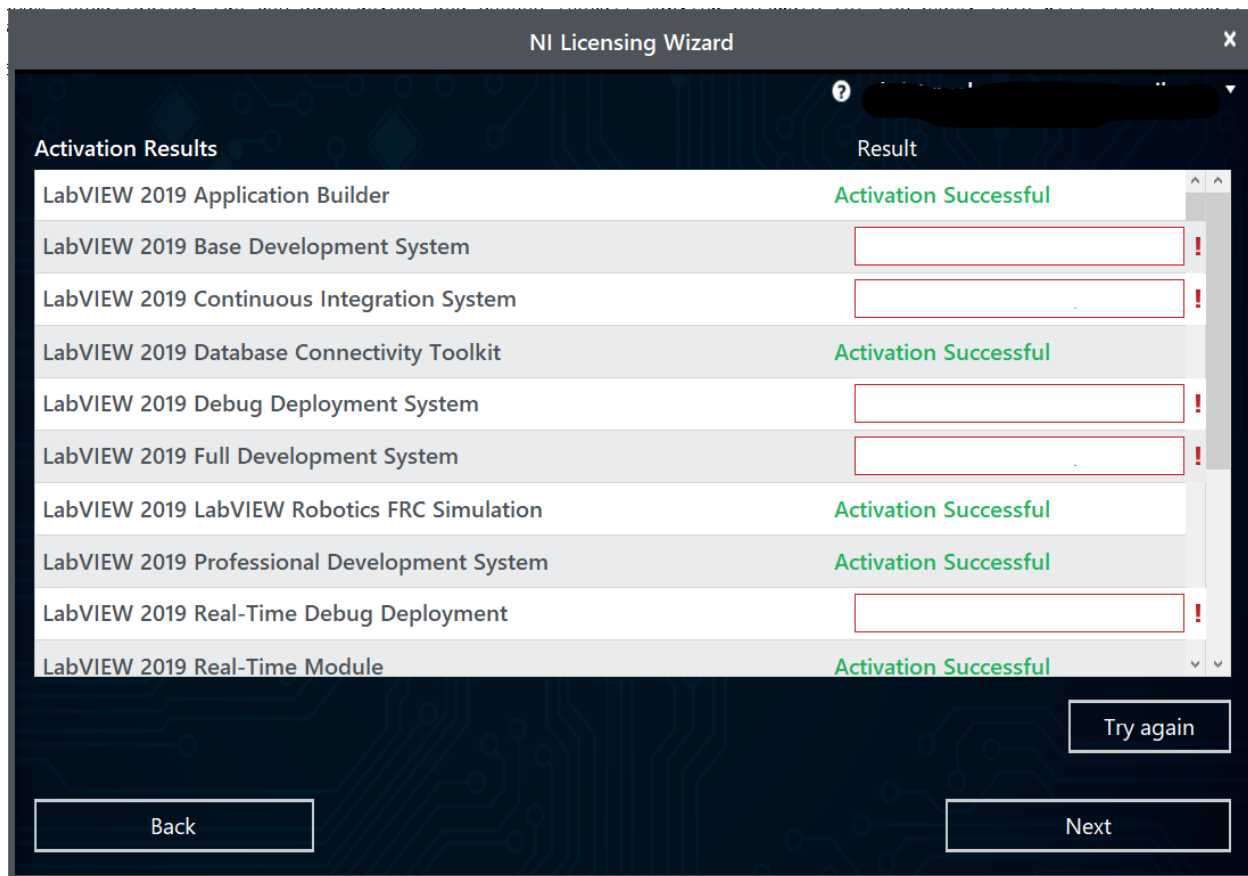
Enter Activation Codes

Refresh

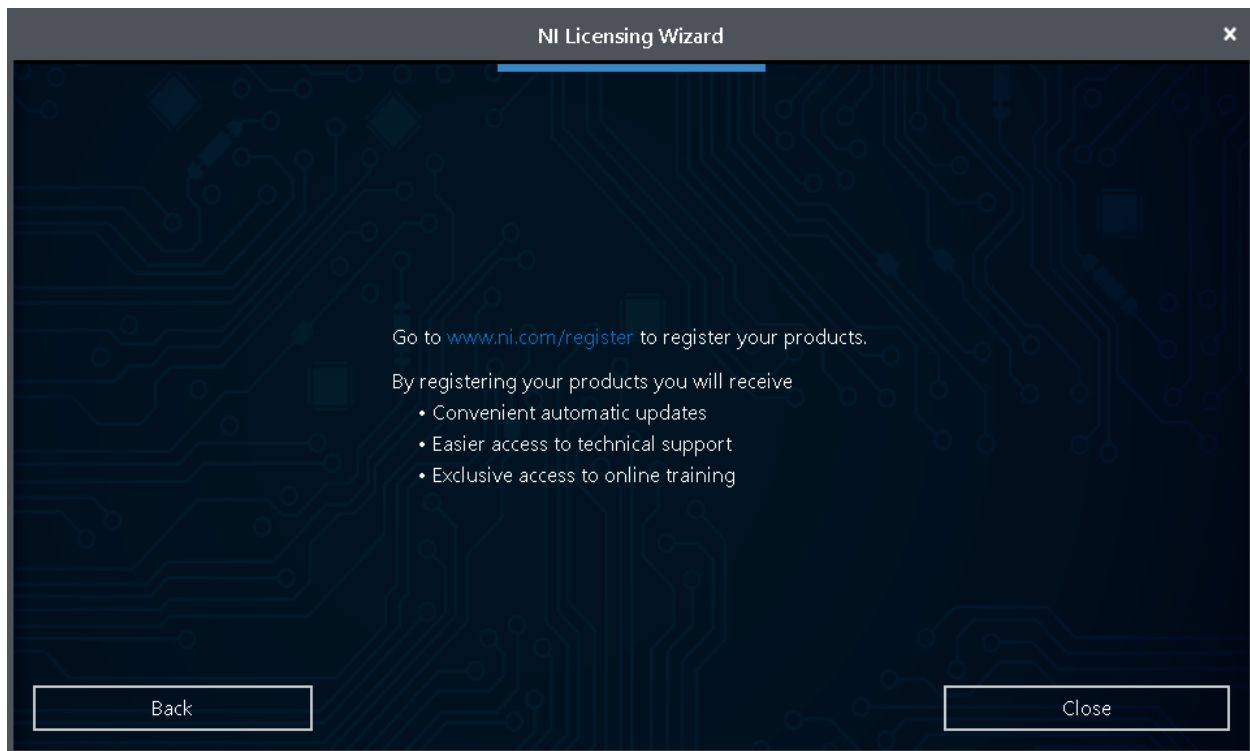
Activate

The serial number you entered at the “User Information” screen should appear in all of the text boxes, if it doesn’t, enter it now. Click “Activate”.

Note: If this is the first time activating the 2020 software on this account, you will see the message shown above about a valid license not being found. You can ignore this.

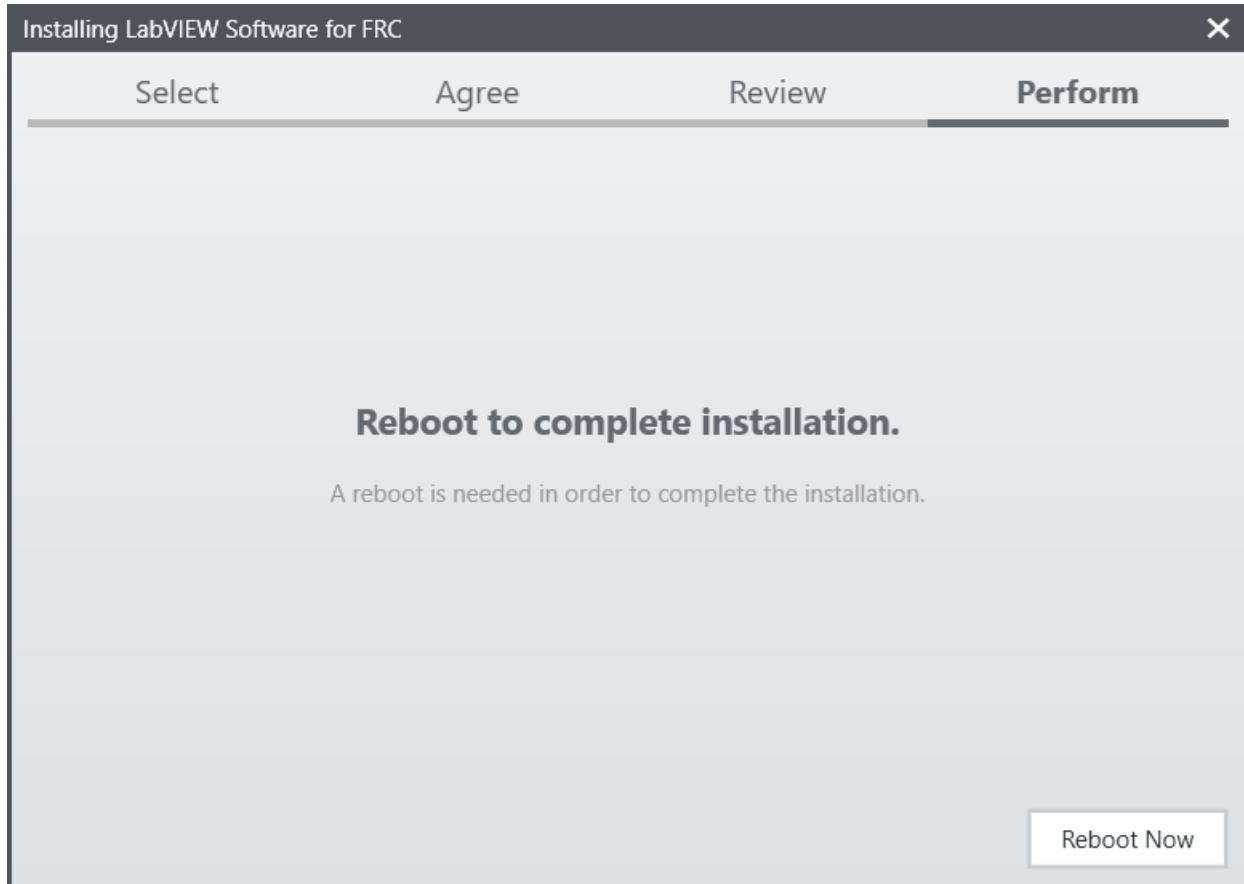


If your products activate successfully, an “Activation Successful” message will appear. If the serial number was incorrect, it will give you a text box and you can re-enter the number and select *Try Again*. The items shown above are not expected to activate. If everything activated successfully, click *Next*.



Click "Close".

Restart



Select *Reboot Now* after closing any open programs.

3.3 Installing the FRC Game Tools

The FRC® Game Tools contains the following software components:

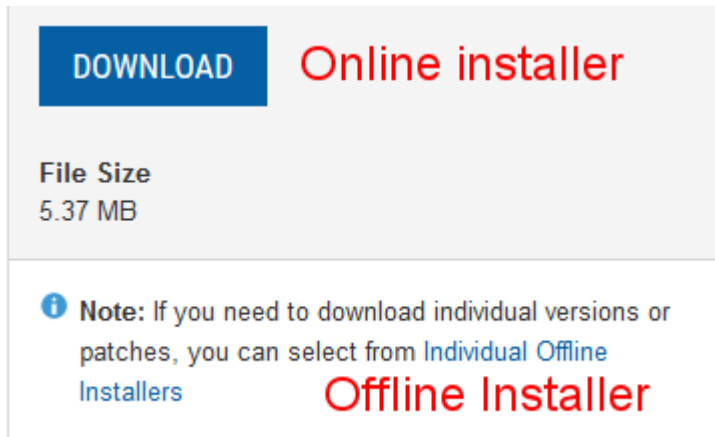
- LabVIEW Update
- FRC Driver Station
- FRC Utilities

The LabVIEW runtime components required for the Driver Station and Utilities are included in this package.

Note: No components from the LabVIEW Software for FRC package are required for running either the Driver Station or Utilities.

3.3.1 Requirements

- Windows 7 or higher (Windows 7, 8, 8.1, 10). Windows 10 is the recommended OS.
- Download the [FRC Game Tools](#) from NI.



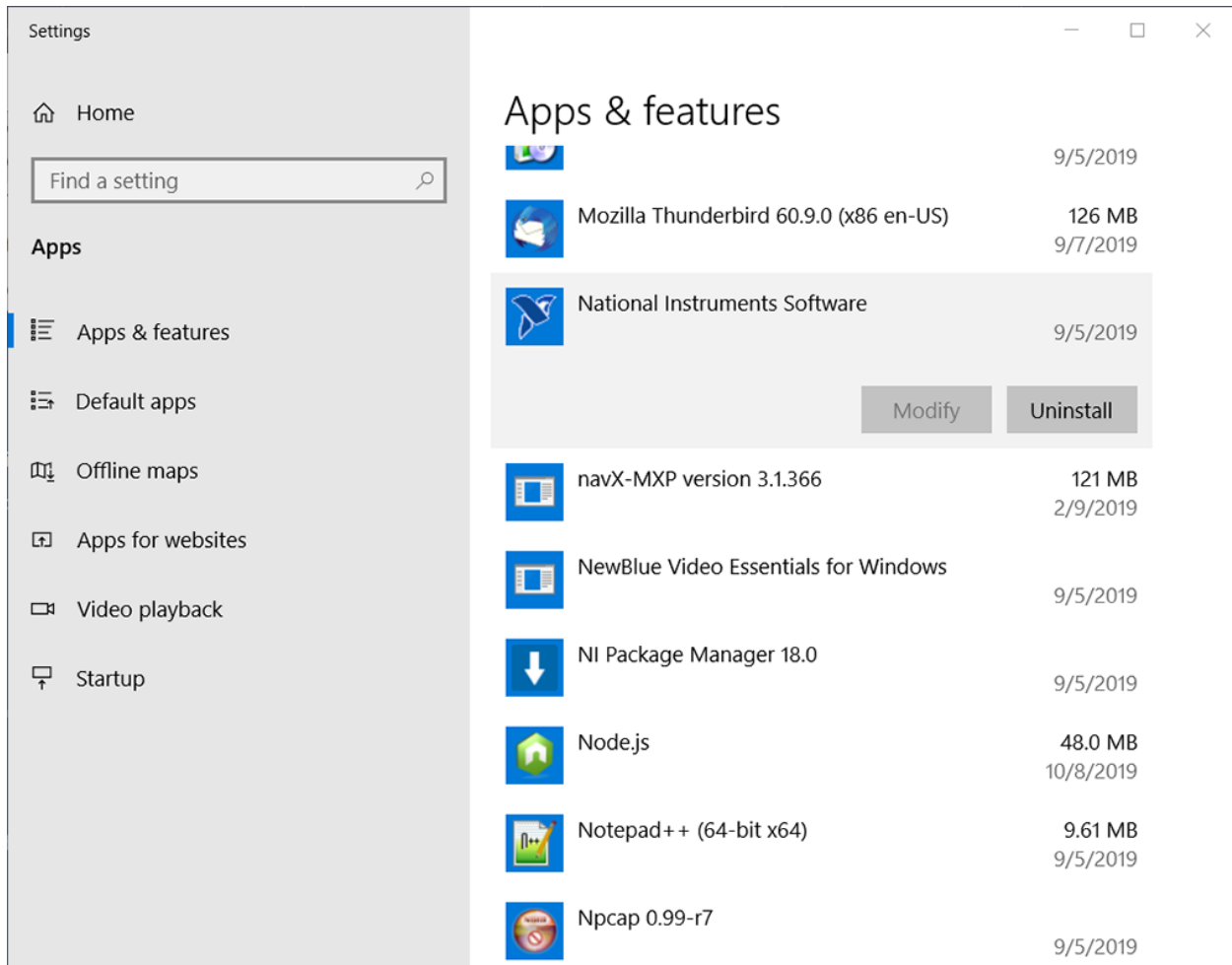
If you wish to install on other machines offline, click *Individual Offline Installers* before clicking *Download* to download the full installer.

3.3.2 Uninstall Old Versions (Recommended)

Important: LabVIEW teams have already completed this step, do not repeat it. LabVIEW teams should skip to the [Installation](#) section.

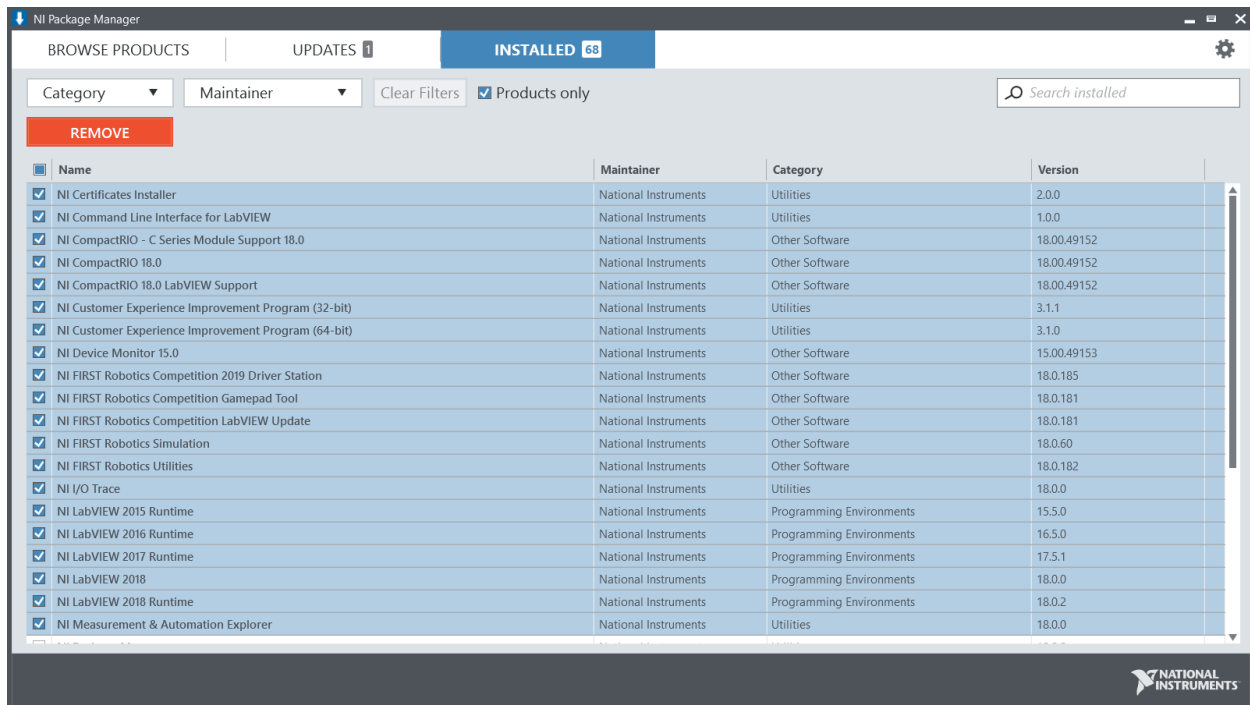
Before installing the new version of the FRC Game Tools it is recommended to remove any old versions. The new version will likely co-exist with the old version (note that the DS will overwrite old versions), but all testing has been done with FRC 2021 only. Then click Start >> Add or Remove Programs. Locate the entry labeled “National Instruments Software”, and select *Uninstall*.

Note: It is only necessary to uninstall previous versions when installing a new year’s tools. For example, uninstall the 2020 tools before installing the 2021 tools. It is not necessary to uninstall before upgrading to a new update of the 2021 game tools.



Select Components to Uninstall

In the dialog box that appears, select all entries. The easiest way to do this is to de-select the *Products Only* check-box and select the check-box to the left of “Name”. Click *Remove*. Wait for the uninstaller to complete and reboot if prompted.



3.3.3 Installation

Important: The Game Tools installer may prompt that .NET Framework 4.6.2 needs to be updated or installed. Follow prompts on-screen to complete the installation, including rebooting if requested. Then resume the installation of the FRC Game Tools, restarting the installer if necessary.

Extraction

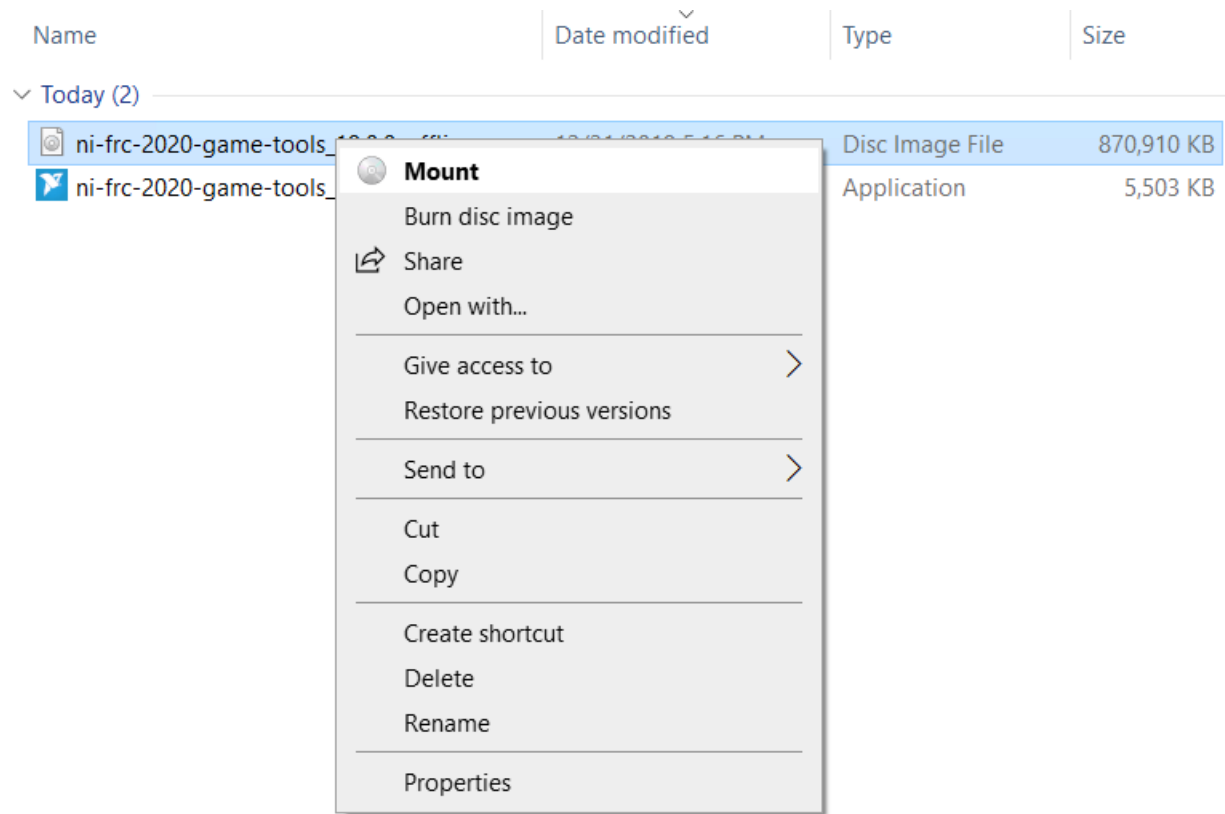
Online

Offline (Windows 10)

Offline (Windows 7, 8, 8.1)

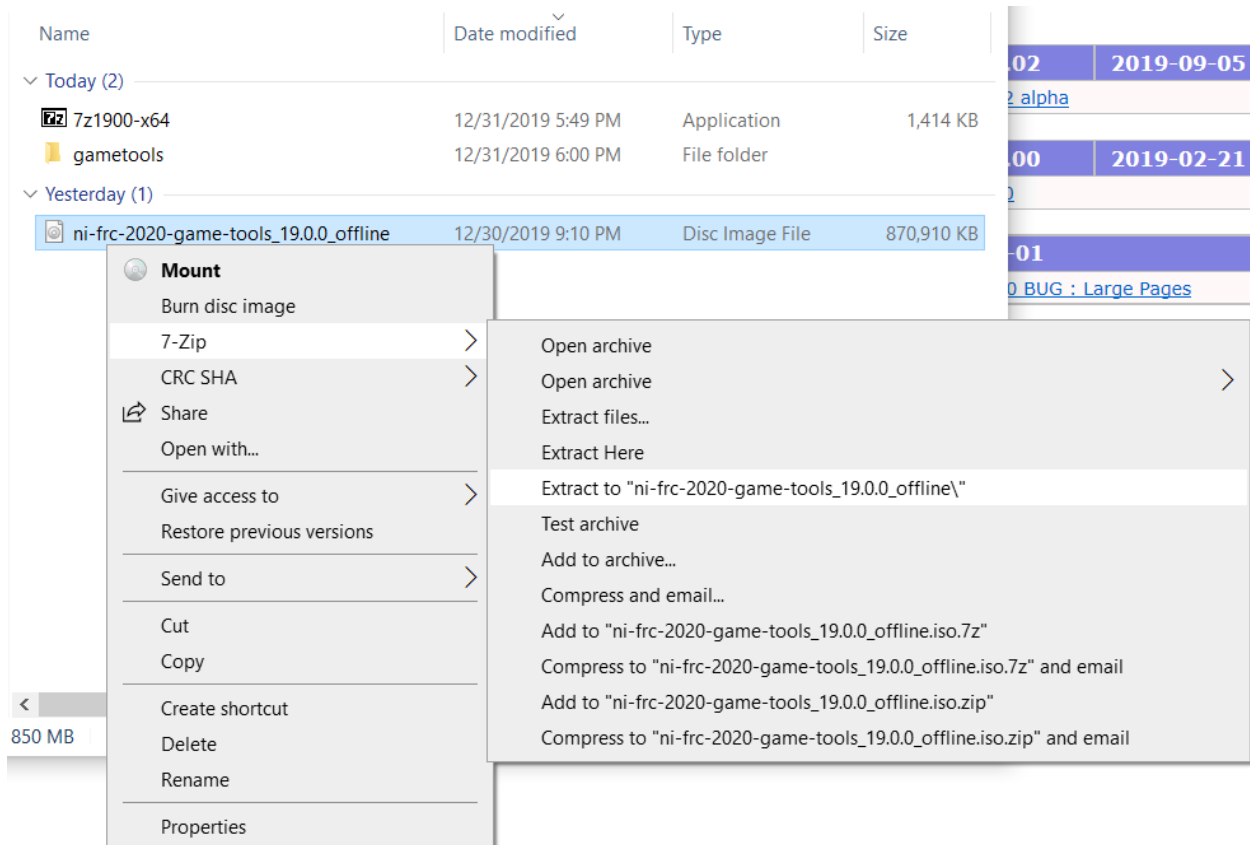
Run the downloaded executable file to start the install process. Click **Yes** if a Windows Security prompt appears.

Right click on the downloaded iso file and select *mount*. Run `install.exe` from the mounted iso. Click **Yes** if a Windows Security prompt appears.



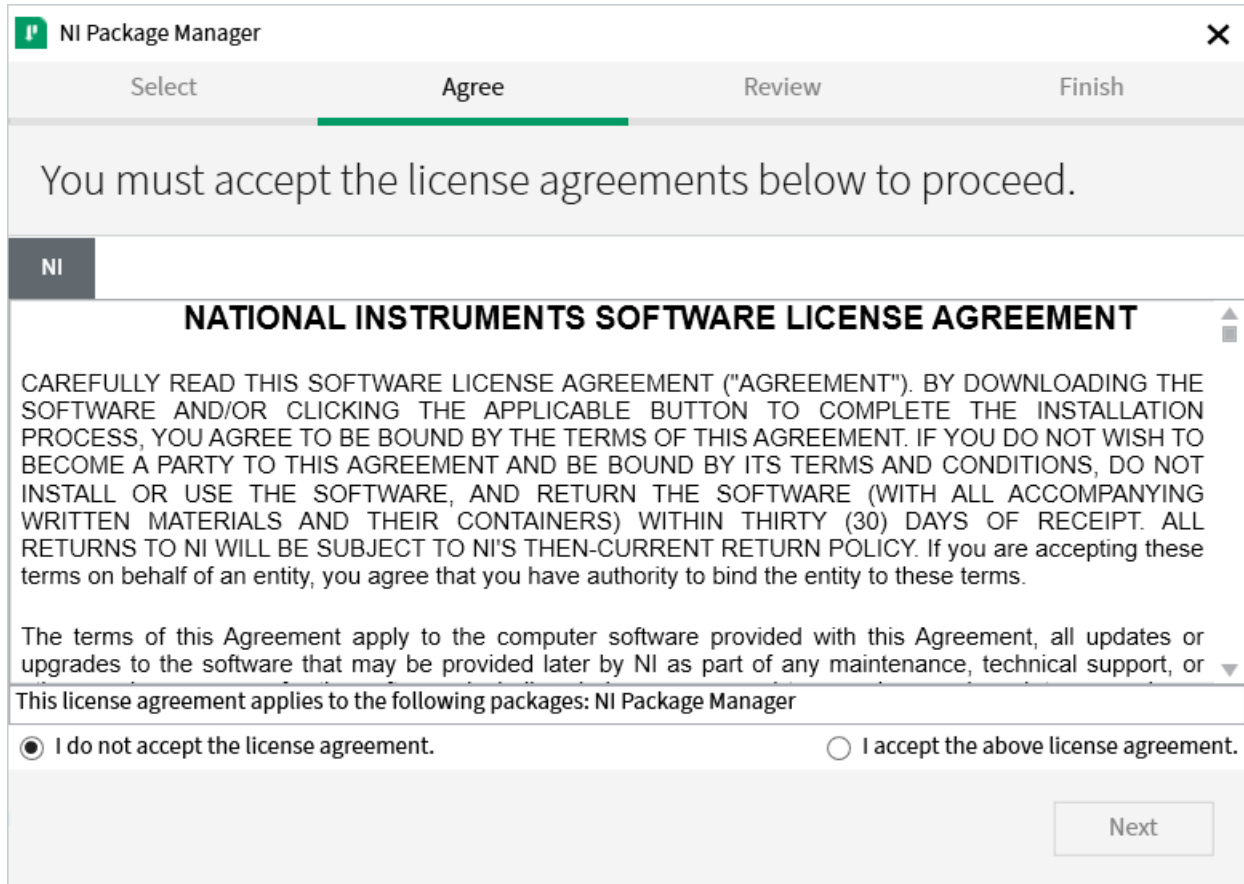
Note: Other installed programs may associate with iso files and the *mount* option may not appear. If that software does not give the option to mount or extract the iso file, then follow the directions in the “Offline Installer (Windows 7, 8, & 8.1)” tab.

Install 7-Zip (download [here](#)). As of the writing of this document, the current released version is 19.00 (2019-02-21). Right click on the downloaded iso file and select *Extract to*.



Run `install.exe` from the extracted folder. Click *Yes* if a Windows Security prompt appears.

NI Package Manager License



The screenshot shows the 'NI Package Manager' window with a progress bar at the top indicating the 'Agree' step is active. Below the progress bar, a message states: 'You must accept the license agreements below to proceed.' A tab labeled 'NI' is selected. The main content area displays the 'NATIONAL INSTRUMENTS SOFTWARE LICENSE AGREEMENT'. The text of the agreement is as follows:

CAREFULLY READ THIS SOFTWARE LICENSE AGREEMENT ("AGREEMENT"). BY DOWNLOADING THE SOFTWARE AND/OR CLICKING THE APPLICABLE BUTTON TO COMPLETE THE INSTALLATION PROCESS, YOU AGREE TO BE BOUND BY THE TERMS OF THIS AGREEMENT. IF YOU DO NOT WISH TO BECOME A PARTY TO THIS AGREEMENT AND BE BOUND BY ITS TERMS AND CONDITIONS, DO NOT INSTALL OR USE THE SOFTWARE, AND RETURN THE SOFTWARE (WITH ALL ACCOMPANYING WRITTEN MATERIALS AND THEIR CONTAINERS) WITHIN THIRTY (30) DAYS OF RECEIPT. ALL RETURNS TO NI WILL BE SUBJECT TO NI'S THEN-CURRENT RETURN POLICY. If you are accepting these terms on behalf of an entity, you agree that you have authority to bind the entity to these terms.

The terms of this Agreement apply to the computer software provided with this Agreement, all updates or upgrades to the software that may be provided later by NI as part of any maintenance, technical support, or

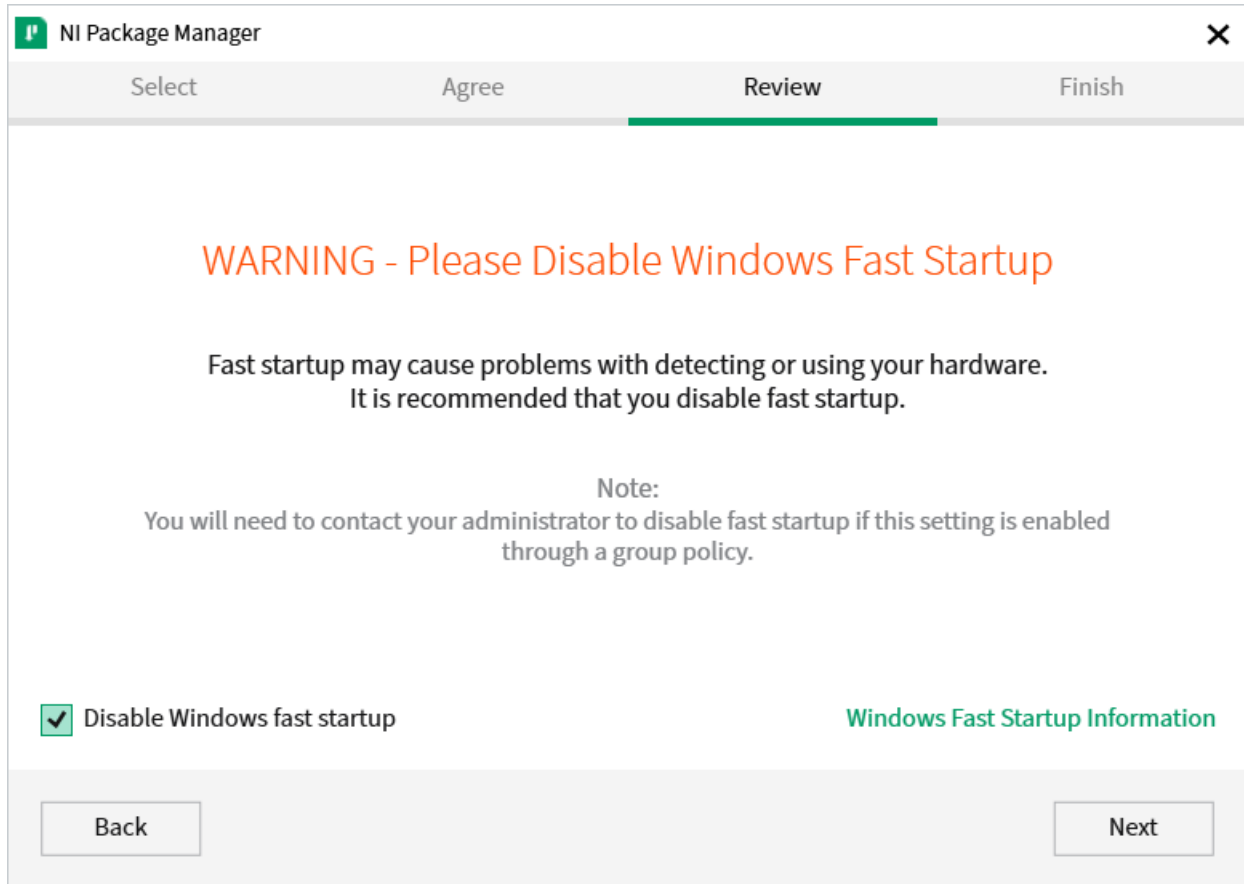
This license agreement applies to the following packages: NI Package Manager

☒ I do not accept the license agreement. ☐ I accept the above license agreement.

A 'Next' button is located at the bottom right of the window.

If you see this screen, click *Next*. This screen confirms that you agree to NI Package Manager License agreement.

Disable Windows Fast Startup



The screenshot shows the 'NI Package Manager' window with a progress bar at the top containing four steps: 'Select', 'Agree', 'Review' (which is highlighted with a green line), and 'Finish'. The main content area has a large orange heading 'WARNING - Please Disable Windows Fast Startup'. Below this, it states: 'Fast startup may cause problems with detecting or using your hardware. It is recommended that you disable fast startup.' A 'Note:' follows, stating: 'You will need to contact your administrator to disable fast startup if this setting is enabled through a group policy.' At the bottom left, there is a checked checkbox labeled 'Disable Windows fast startup'. To the right of this is a green link 'Windows Fast Startup Information'. At the very bottom, there are two buttons: 'Back' on the left and 'Next' on the right.

NI Package Manager

Select Agree **Review** Finish

WARNING - Please Disable Windows Fast Startup

Fast startup may cause problems with detecting or using your hardware.
It is recommended that you disable fast startup.

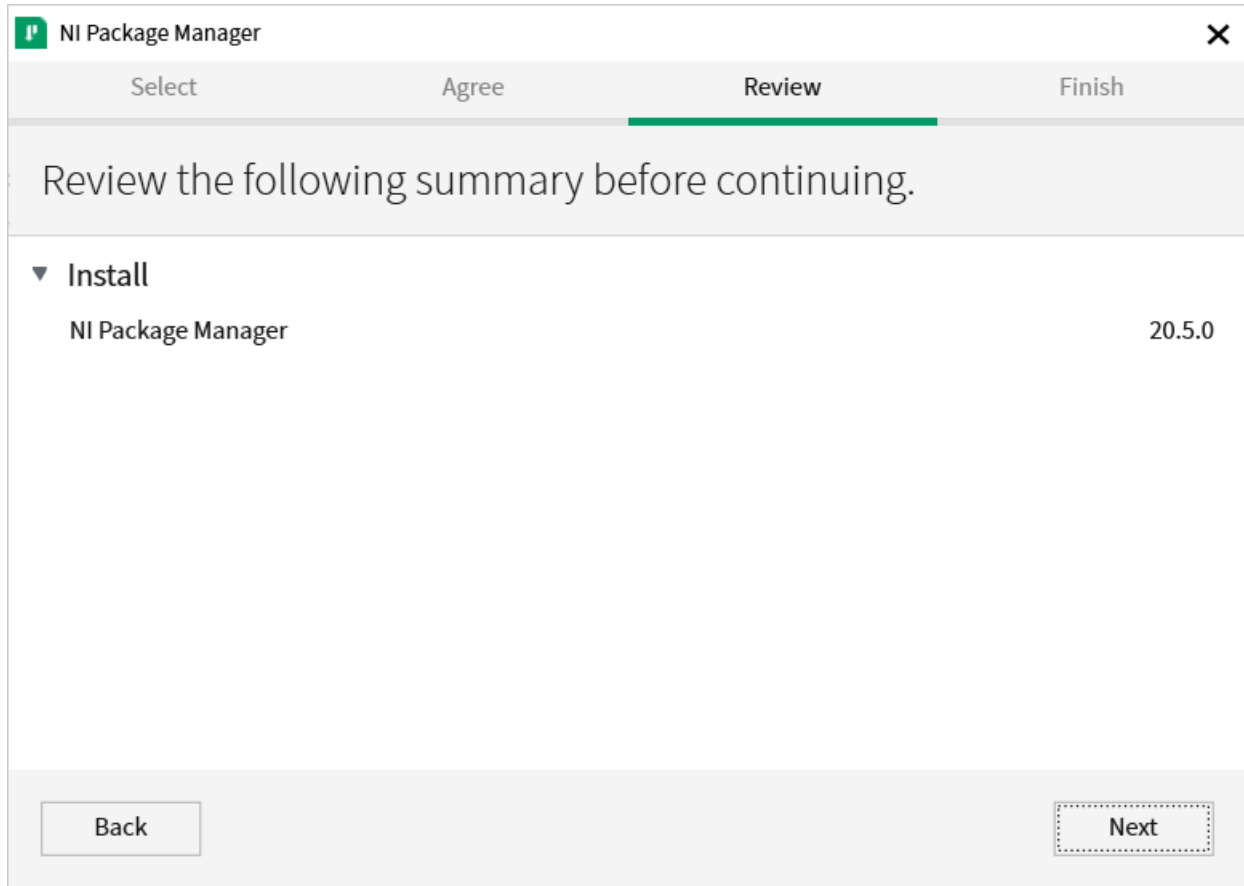
Note:
You will need to contact your administrator to disable fast startup if this setting is enabled through a group policy.

☒ Disable Windows fast startup [Windows Fast Startup Information](#)

Back Next

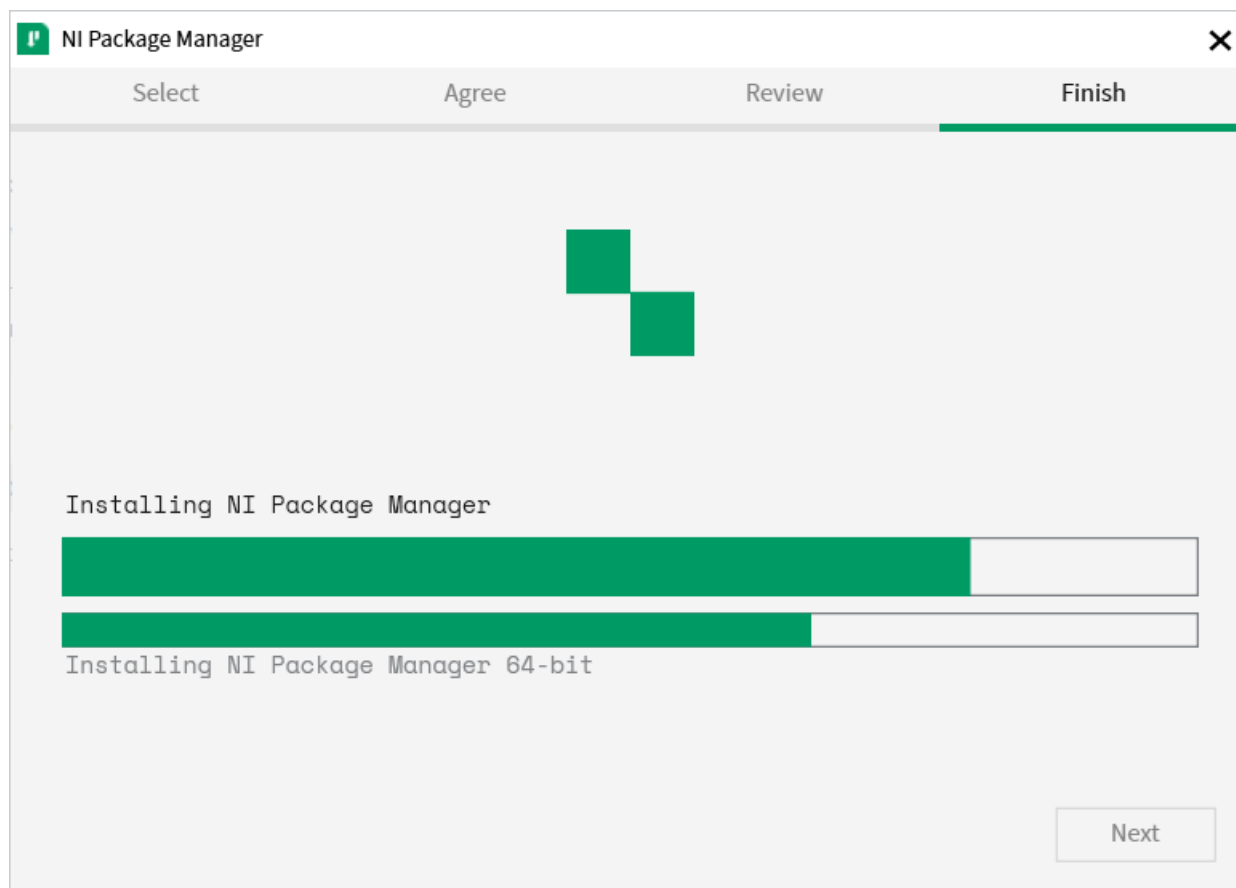
It is recommended to leave this screen as-is, as Windows Fast Startup can cause issues with the NI drivers required to image the roboRIO. Go ahead and click *Next*.

NI Package Manager Review



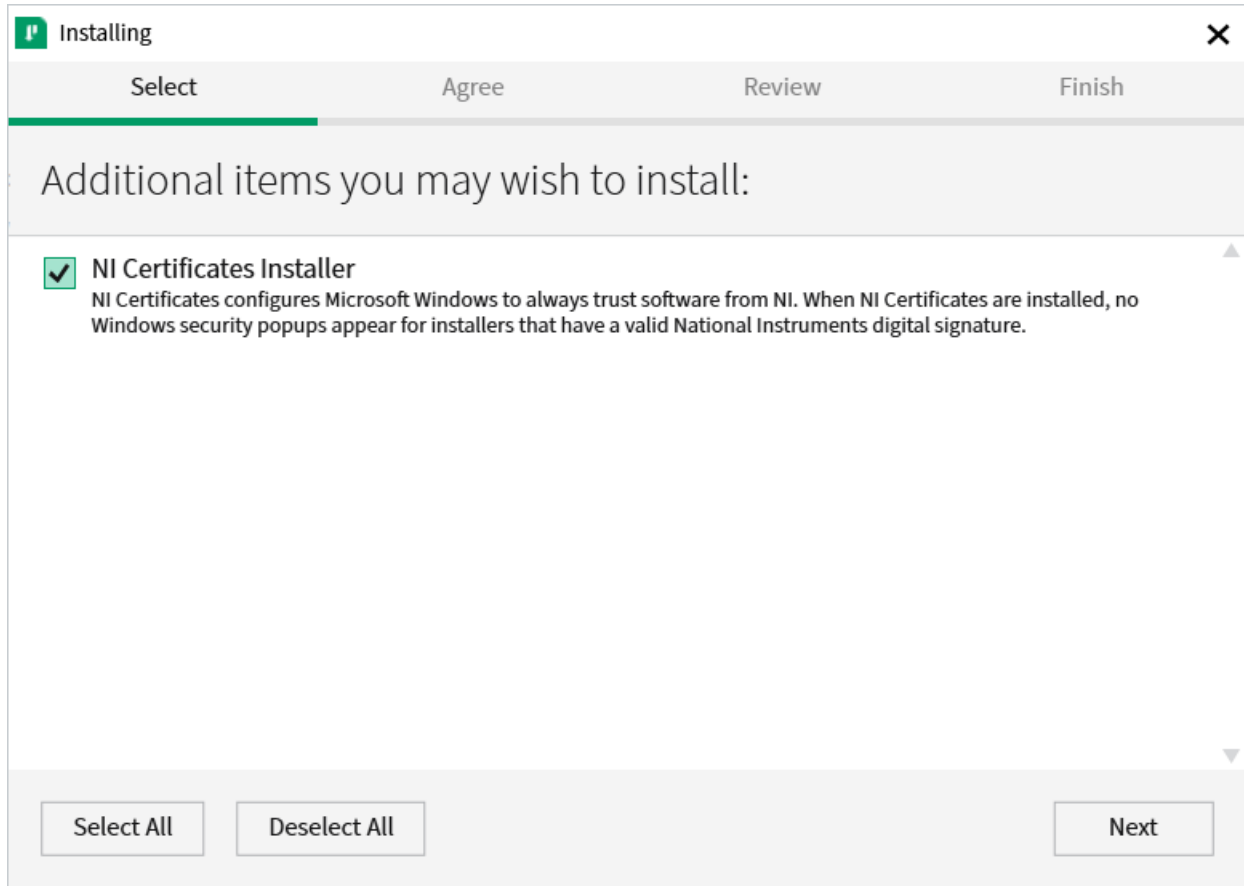
If you see this screen, click *Next*.

NI Package Manager Installation



Installation progress of the NI Package Manager will be tracked in this window.

Additional Software



If you see this screen, click *Next*.

License Agreements

The screenshot shows a software installation window titled "Installing" with a close button (X) in the top right corner. Below the title bar is a progress bar with four steps: "Select", "Agree", "Review", and "Finish". The "Agree" step is currently active, indicated by a green underline. The main content area displays the text: "You must accept the license agreements below to proceed." Below this is a tabbed interface with two tabs: "NI" (selected) and "FIRST Competition". The "NI" tab contains the "NATIONAL INSTRUMENTS SOFTWARE LICENSE AGREEMENT". The agreement text reads: "CAREFULLY READ THIS SOFTWARE LICENSE AGREEMENT ('AGREEMENT'). BY DOWNLOADING THE SOFTWARE AND/OR CLICKING THE APPLICABLE BUTTON TO COMPLETE THE INSTALLATION PROCESS, YOU AGREE TO BE BOUND BY THE TERMS OF THIS AGREEMENT. IF YOU DO NOT WISH TO BECOME A PARTY TO THIS AGREEMENT AND BE BOUND BY ITS TERMS AND CONDITIONS, DO NOT INSTALL OR USE THE SOFTWARE, AND RETURN THE SOFTWARE (WITH ALL ACCOMPANYING WRITTEN MATERIALS AND THEIR CONTAINERS) WITHIN THIRTY (30) DAYS OF RECEIPT. ALL RETURNS TO NI WILL BE SUBJECT TO NI'S THEN-CURRENT RETURN POLICY. If you are accepting these terms on behalf of an entity, you agree that you have authority to bind the entity to these terms." Below the agreement text, it states: "The terms of this Agreement apply to the computer software provided with this Agreement, all updates or upgrades to the software that may be provided later by NI as part of any maintenance, technical support, or" followed by a downward arrow. A line below that states: "This license agreement applies to the following packages: NI License Manager, LabVIEW Runtime (32-bit), FRC Game Tools". At the bottom, there are two radio button options: "I do not accept all the license agreements." (unselected) and "I accept the above 2 license agreements." (selected). At the very bottom are two buttons: "Back" and "Next".

Installing

Select Agree Review Finish

You must accept the license agreements below to proceed.

NI FIRST Competition

NATIONAL INSTRUMENTS SOFTWARE LICENSE AGREEMENT

CAREFULLY READ THIS SOFTWARE LICENSE AGREEMENT ("AGREEMENT"). BY DOWNLOADING THE SOFTWARE AND/OR CLICKING THE APPLICABLE BUTTON TO COMPLETE THE INSTALLATION PROCESS, YOU AGREE TO BE BOUND BY THE TERMS OF THIS AGREEMENT. IF YOU DO NOT WISH TO BECOME A PARTY TO THIS AGREEMENT AND BE BOUND BY ITS TERMS AND CONDITIONS, DO NOT INSTALL OR USE THE SOFTWARE, AND RETURN THE SOFTWARE (WITH ALL ACCOMPANYING WRITTEN MATERIALS AND THEIR CONTAINERS) WITHIN THIRTY (30) DAYS OF RECEIPT. ALL RETURNS TO NI WILL BE SUBJECT TO NI'S THEN-CURRENT RETURN POLICY. If you are accepting these terms on behalf of an entity, you agree that you have authority to bind the entity to these terms.

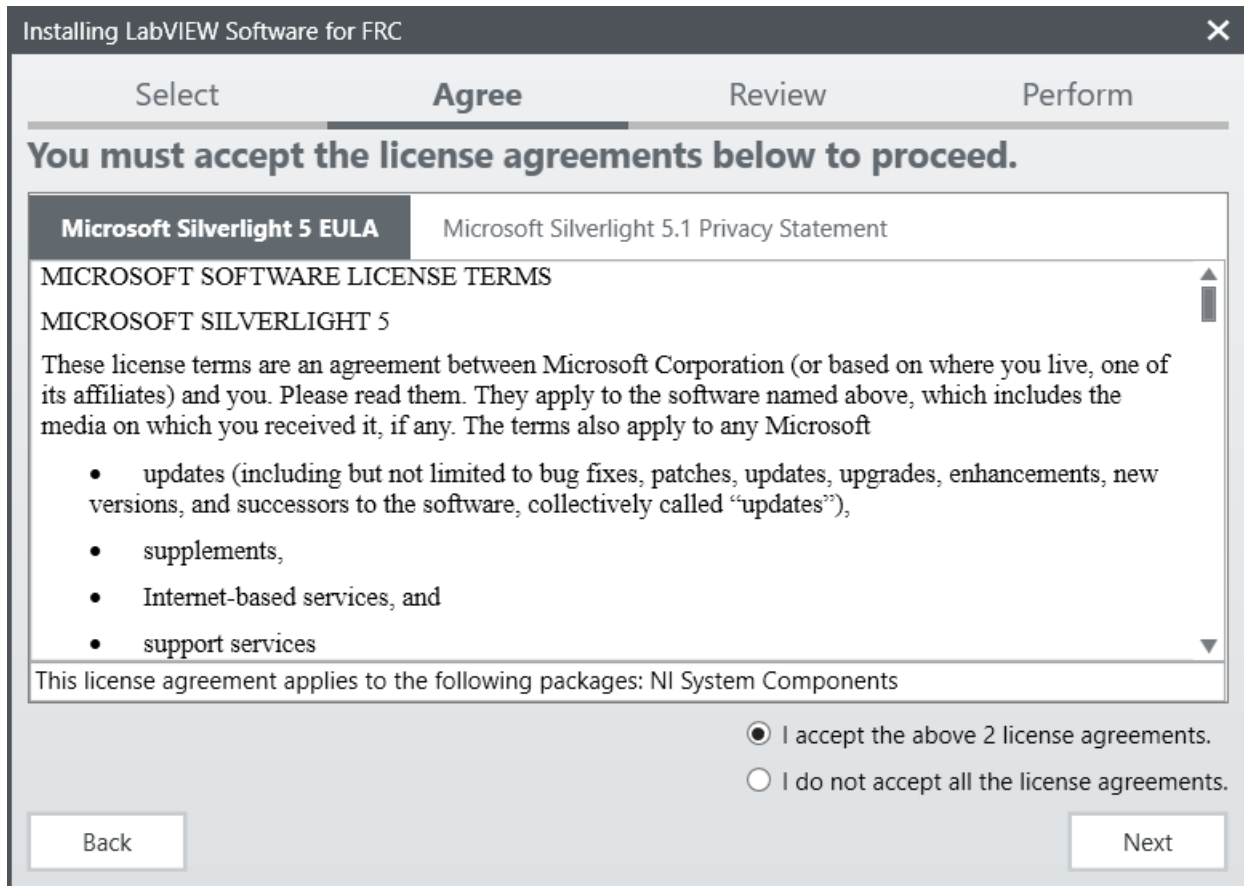
The terms of this Agreement apply to the computer software provided with this Agreement, all updates or upgrades to the software that may be provided later by NI as part of any maintenance, technical support, or

This license agreement applies to the following packages: NI License Manager, LabVIEW Runtime (32-bit), FRC Game Tools

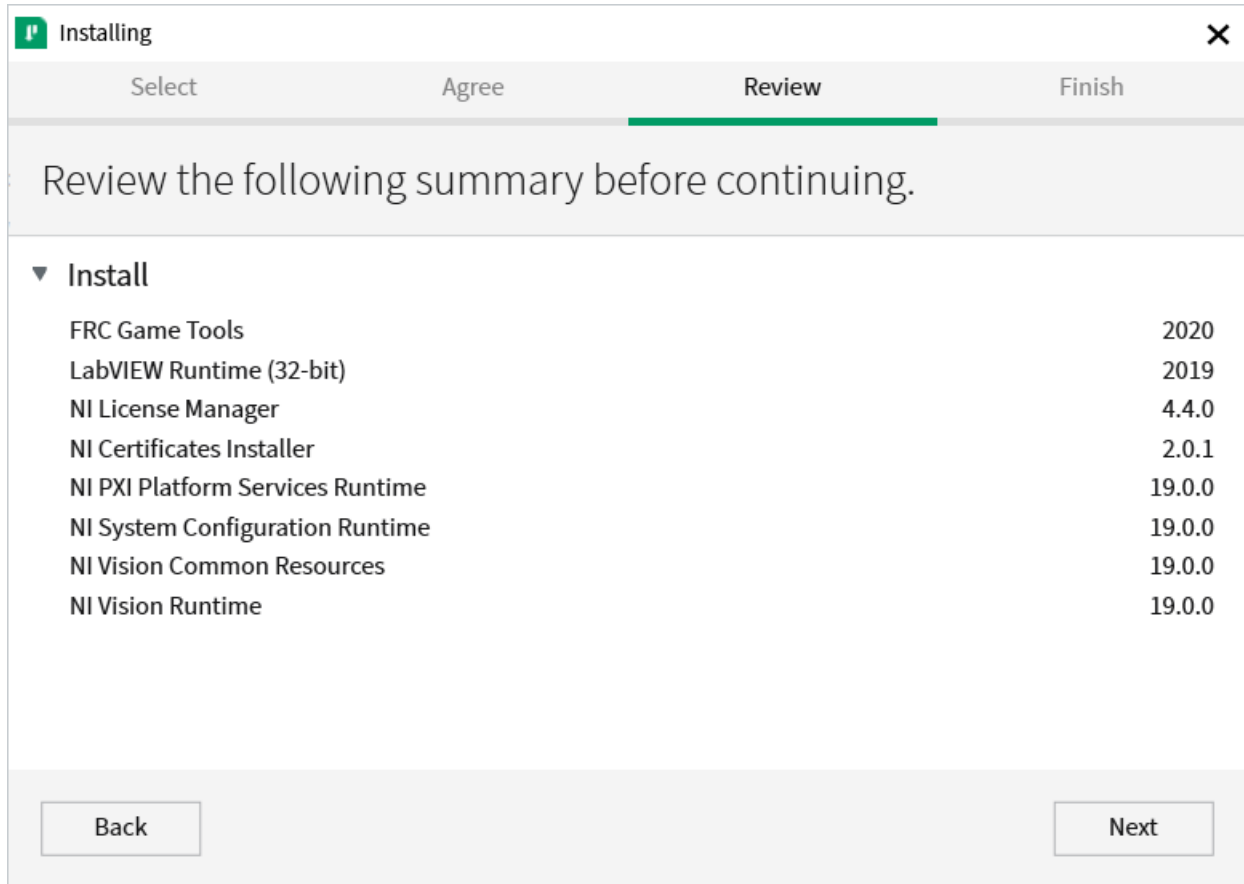
☐ I do not accept all the license agreements. ☒ I accept the above 2 license agreements.

Back Next

Select *I accept...* then click *Next*

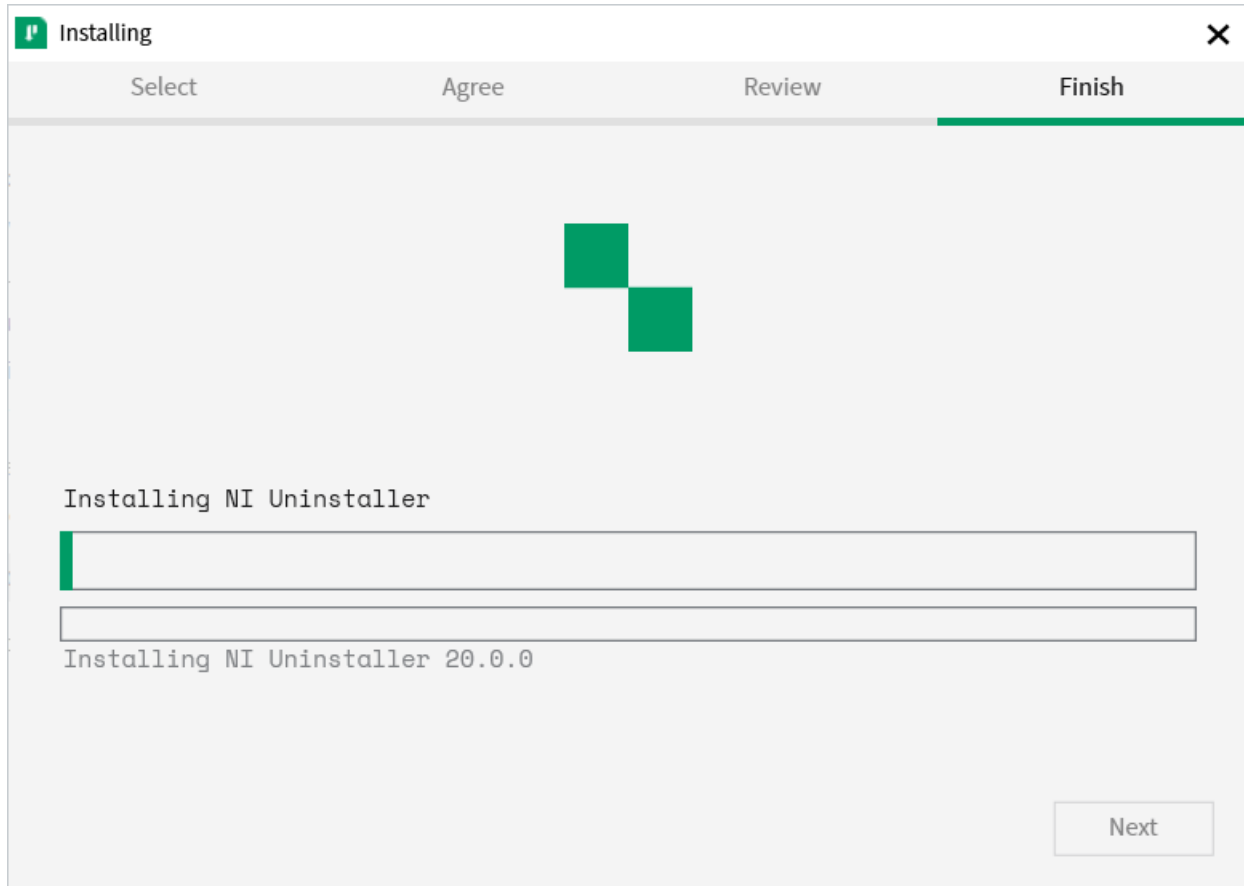


Go ahead and press *I accept...* then click *Next*, confirming that you agree to the NI License agreement.

Review Summary

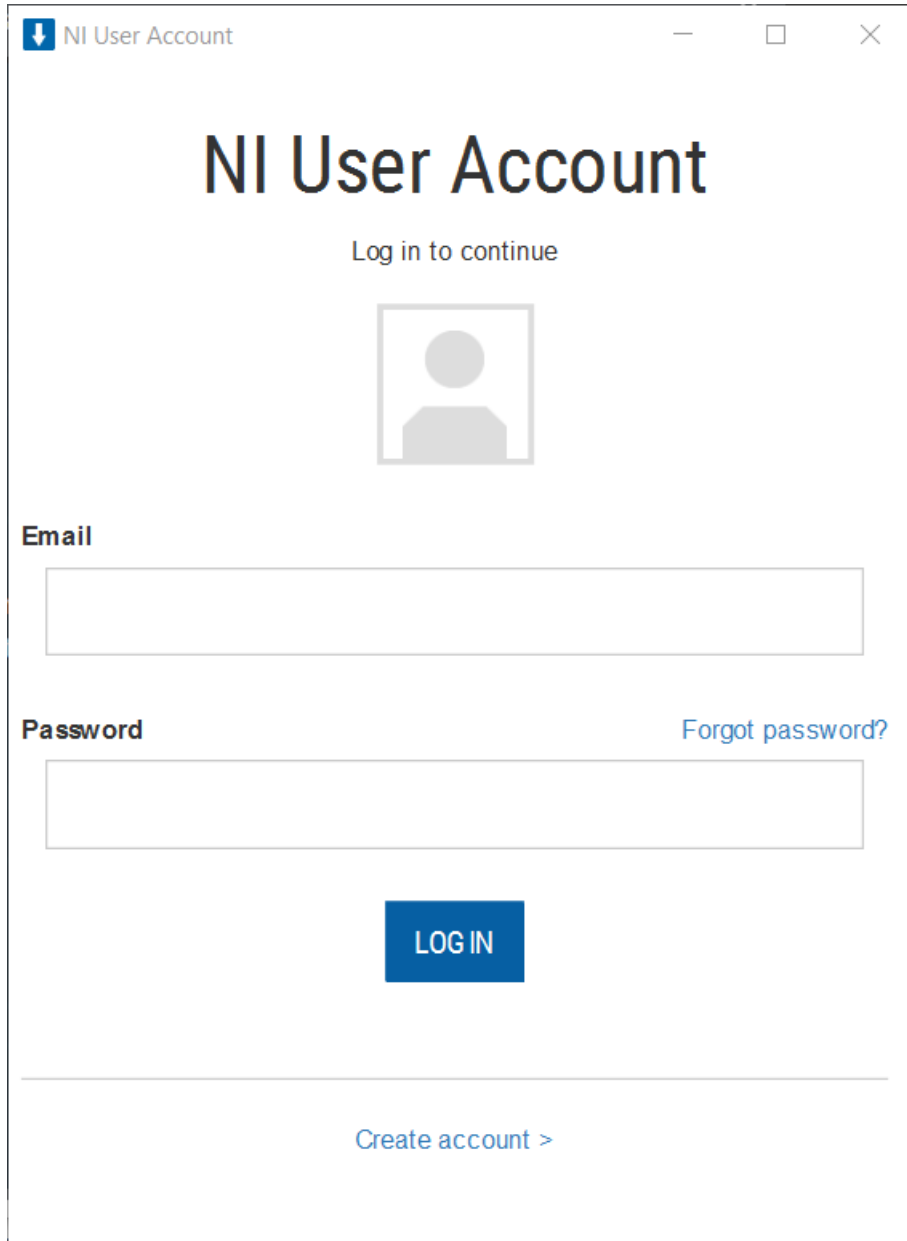
Click *Next*.

Detail Progress



This screen showcases the installation process, go ahead and press *Next* when it's finished.


NI Activation Wizard

A screenshot of the NI User Account login window. The window has a title bar with a blue square icon, the text "NI User Account", and standard window controls (minimize, maximize, close). The main content area has a large "NI User Account" title, followed by the text "Log in to continue". Below this is a placeholder for a user profile picture. There are two input fields: "Email" and "Password". To the right of the password field is a link "Forgot password?". Below the input fields is a blue "LOGIN" button. At the bottom, separated by a horizontal line, is a link "Create account >".

NI User Account

NI User Account

Log in to continue



Email

Password

[Forgot password?](#)

LOGIN

[Create account >](#)

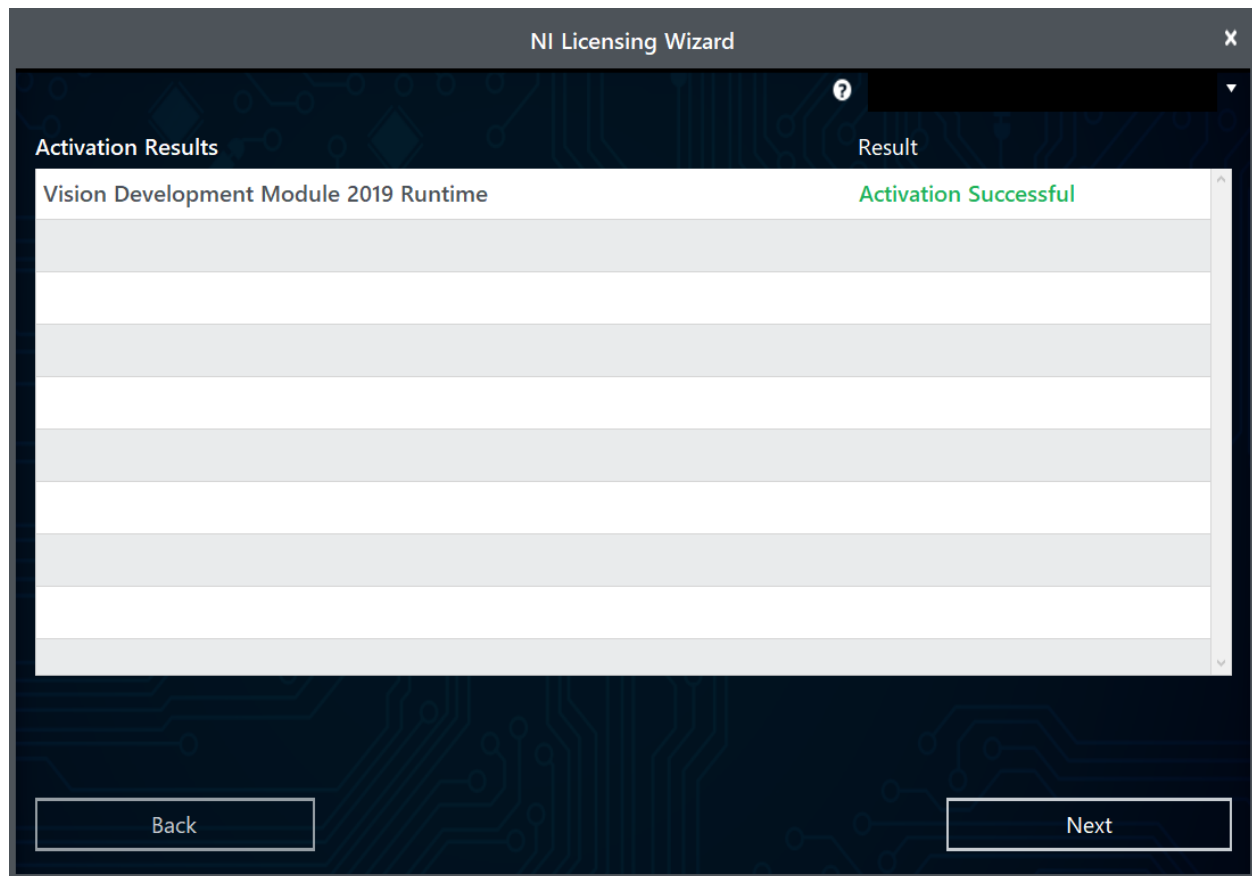
Log into your ni.com account. If you don't have an account, select *Create account* to create a free account.

The image shows a screenshot of the 'NI Licensing Wizard' window. The window has a dark blue background with a circuit-like pattern. At the top, the title bar says 'NI Licensing Wizard' with a close button (X) on the right. Below the title bar, there is a message: 'A valid license for the following software product(s) was not found for your acc' followed by 'Enter serial numbers to activate'. To the right of this message is a 'Serial Number' label. Below the message is a table with multiple rows. The first row contains 'Vision Development Module 2019 Runtime' in the first column and 'XXXXXXXXXX' in the second column. The other rows are empty. At the bottom left, there is a link 'Enter Activation Codes'. At the bottom right, there are two buttons: 'Refresh' and 'Activate'.

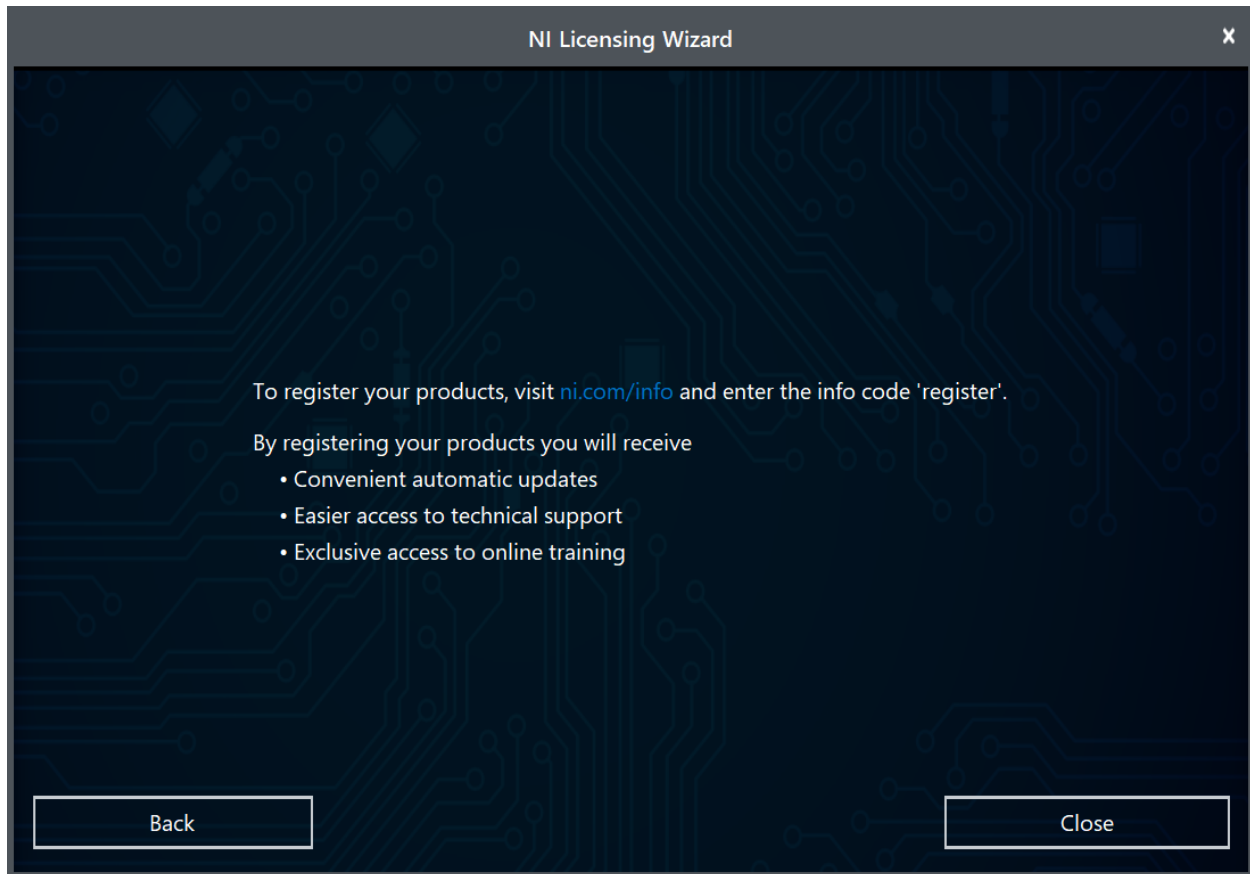
	Serial Number
Vision Development Module 2019 Runtime	XXXXXXXXXX

Enter the serial number. Click *Activate*.

Note: If this is the first time activating this year's software on this account, you will see the message shown above about a valid license not being found. You can ignore this.

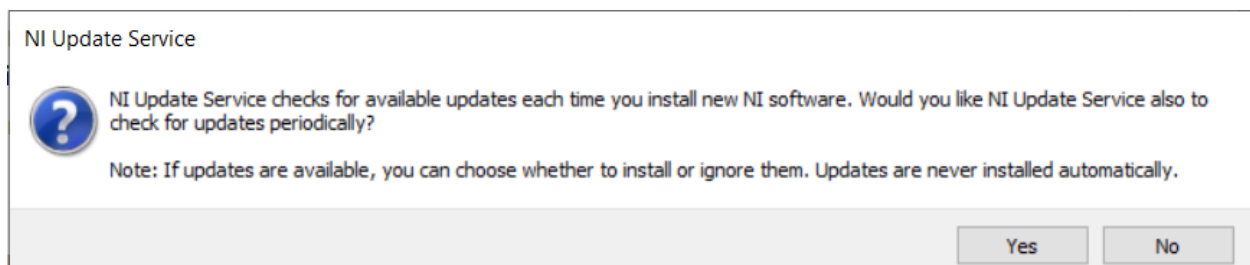


If your products activate successfully, an *Activation Successful* message will appear. If the serial number was incorrect, it will give you a text box and you can re-enter the number and select *Try Again*. If everything activated successfully, click *Next*.



Click *Close*.

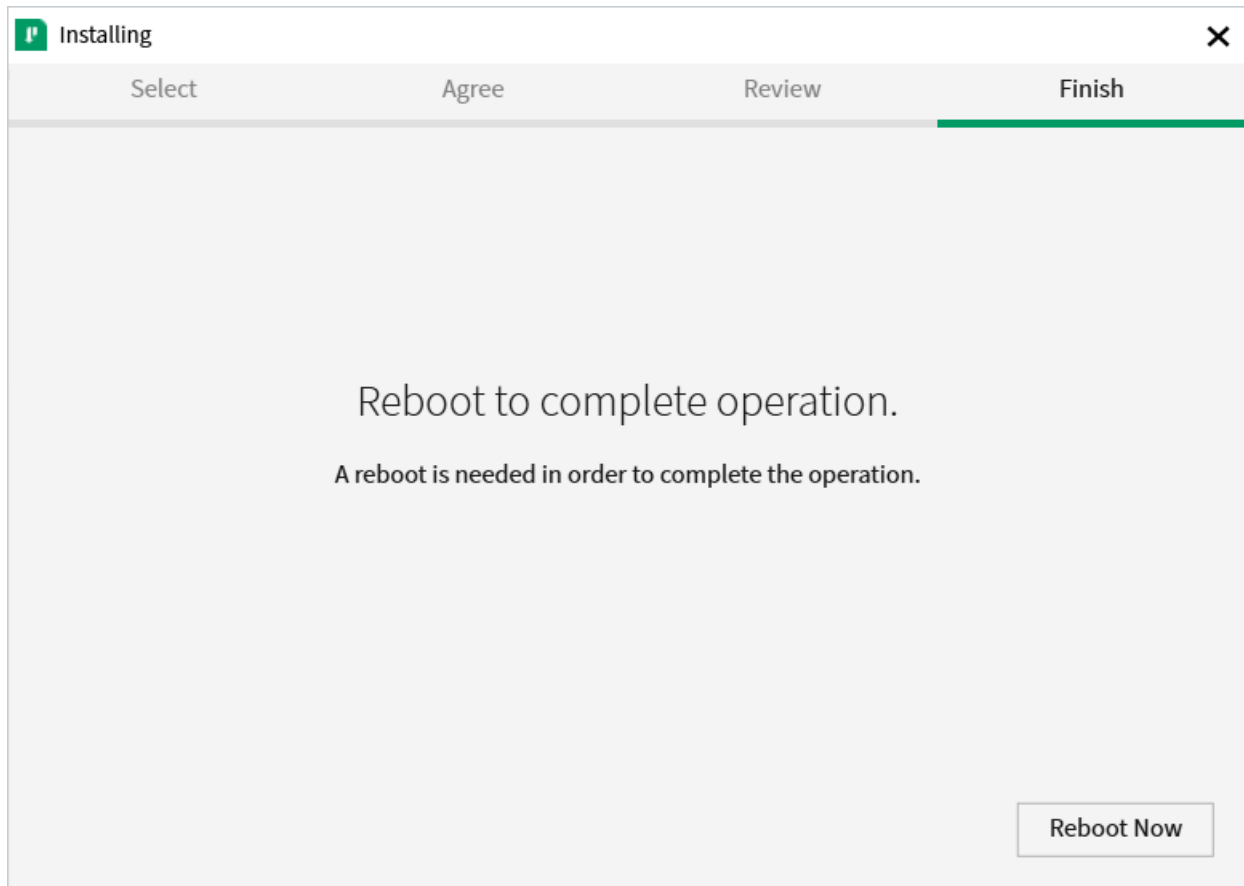
NI Update Service



You will be prompted whether to enable the NI update service. You can choose to not enable the update service.

Warning: It is not recommended to install these updates unless directed by FRC through our usual communication channels (FRC Blog, Team Updates or E-mail Blasts).

3.3.4 Reboot to Complete Installation



If prompted, select *Reboot Now* after closing any open programs.

3.4 WPILib Installation Guide

This guide is intended for Java and C++ teams. LabVIEW teams can skip to [Installing LabVIEW for FRC \(LabVIEW only\)](#). Additionally, the below tutorial shows Windows 10, but the steps are identical for all operating systems. Notes differentiating operating systems will be shown.

3.4.1 Prerequisites

You can download the latest release of the installer from [GitHub](#). Ensure that you download the correct binary for your OS and architecture.

Note: Windows 7 users must have an updated system with [this](#) update installed.

Important: The minimum supported macOS version is Mojave (10.14.x).

3.4.2 Extracting the Installer

When you download the WPILib installer, it is distributed as a disk image file `.iso` for Windows, `.tar.gz` for Linux, and distributed as a DMG for MacOS.

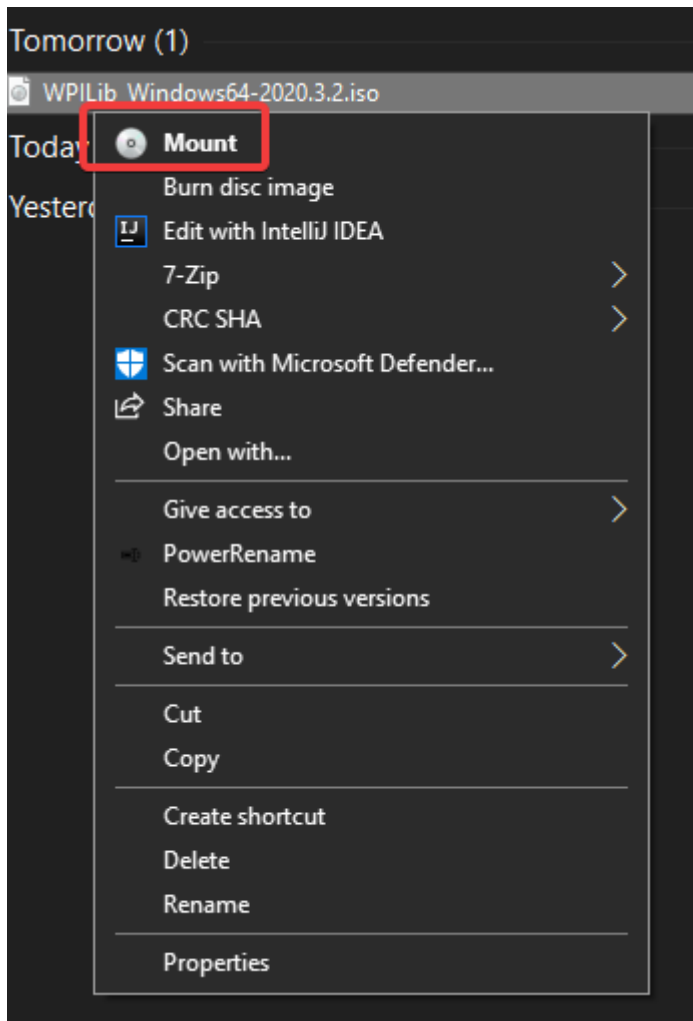
Windows 10

Windows 7

macOS

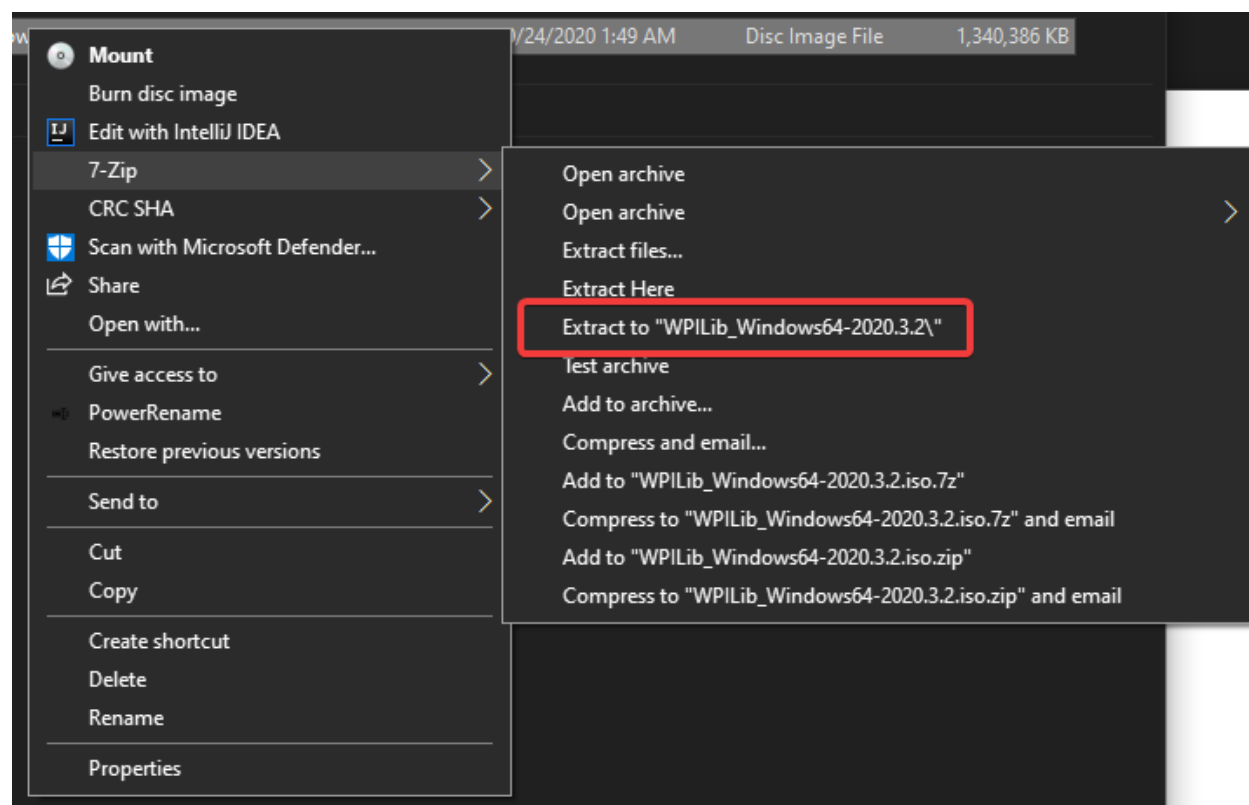
Linux

Windows 10 users can right click on the downloaded disk image and select *Mount* to open it. Then launch `WPILibInstaller.exe`.

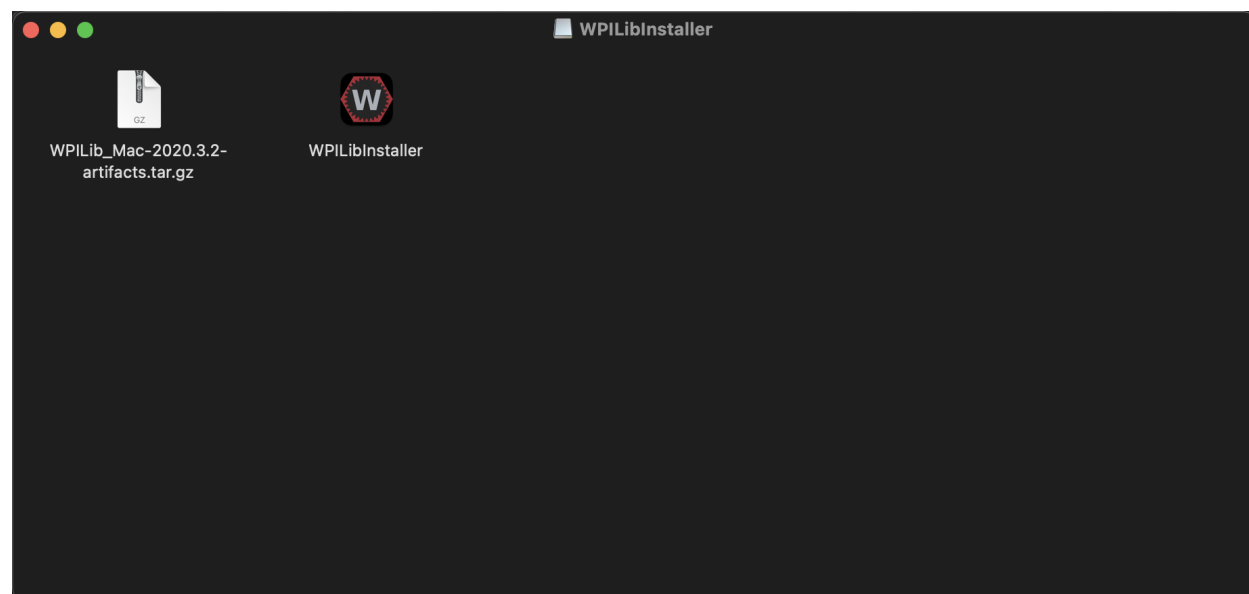


Note: Other installed programs may associate with iso files and the *mount* option may not appear. If that software does not give the option to mount or extract the iso file, then follow the directions in the “Windows 7” tab.

You can use [7-zip](#) to extract the disk image by right-clicking, selecting *7-Zip* and selecting *Extract to....* Then launch `WPILibInstaller.exe`



macOS users can double click on the downloaded DMG and then select WPILibInstaller to launch the application.



Linux users should extract the downloaded .tar.gz and then launch WPILibInstaller. Ubuntu treats executables in the file explorer as shared libraries, so double-clicking won't run them. Run the following commands in a terminal instead with <version> replaced with the version you're installing.

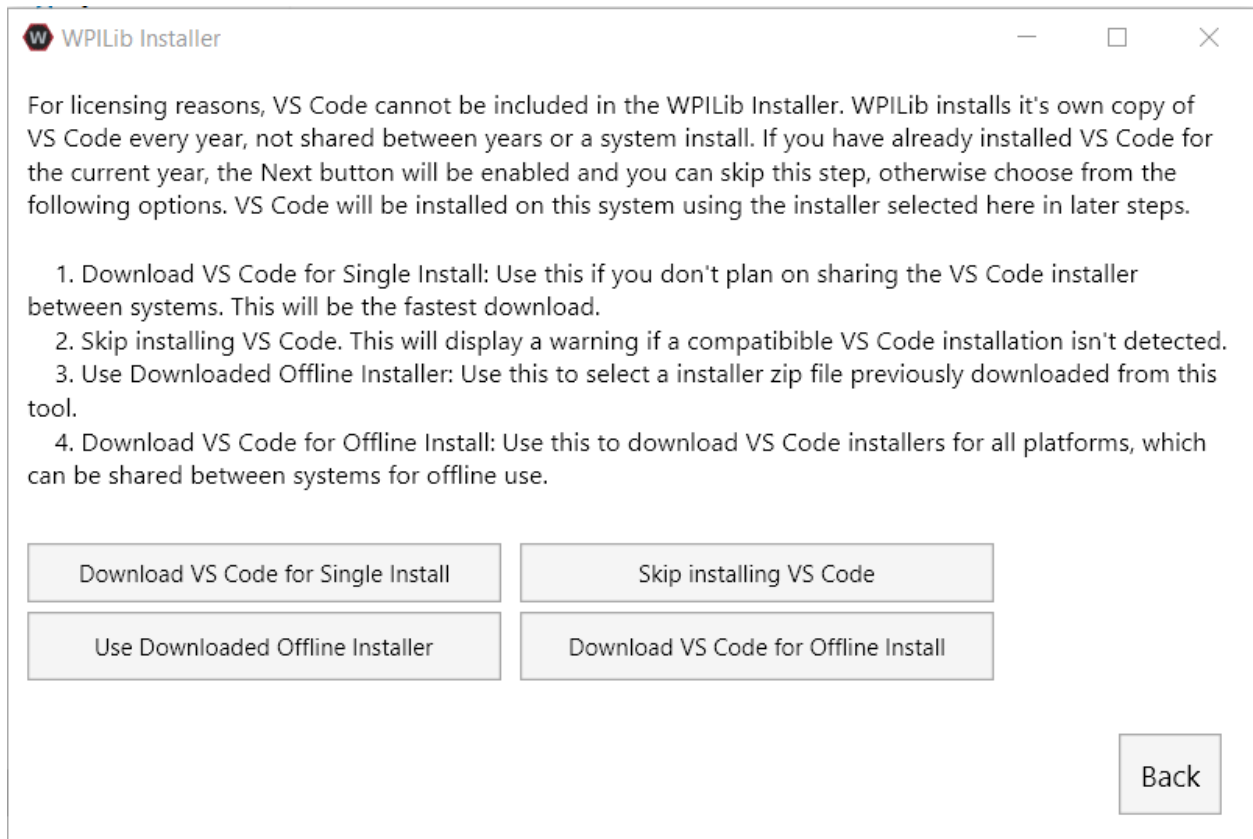
```
$ tar -xf WPILib_Linux-<version>.tar.gz  
$ cd WPILib_Linux-<version>/  
$ ./WPILibInstaller
```

3.4.3 Running the Installer

Upon opening the installer, you'll be presented with the below screen. Go ahead and press *Start*.

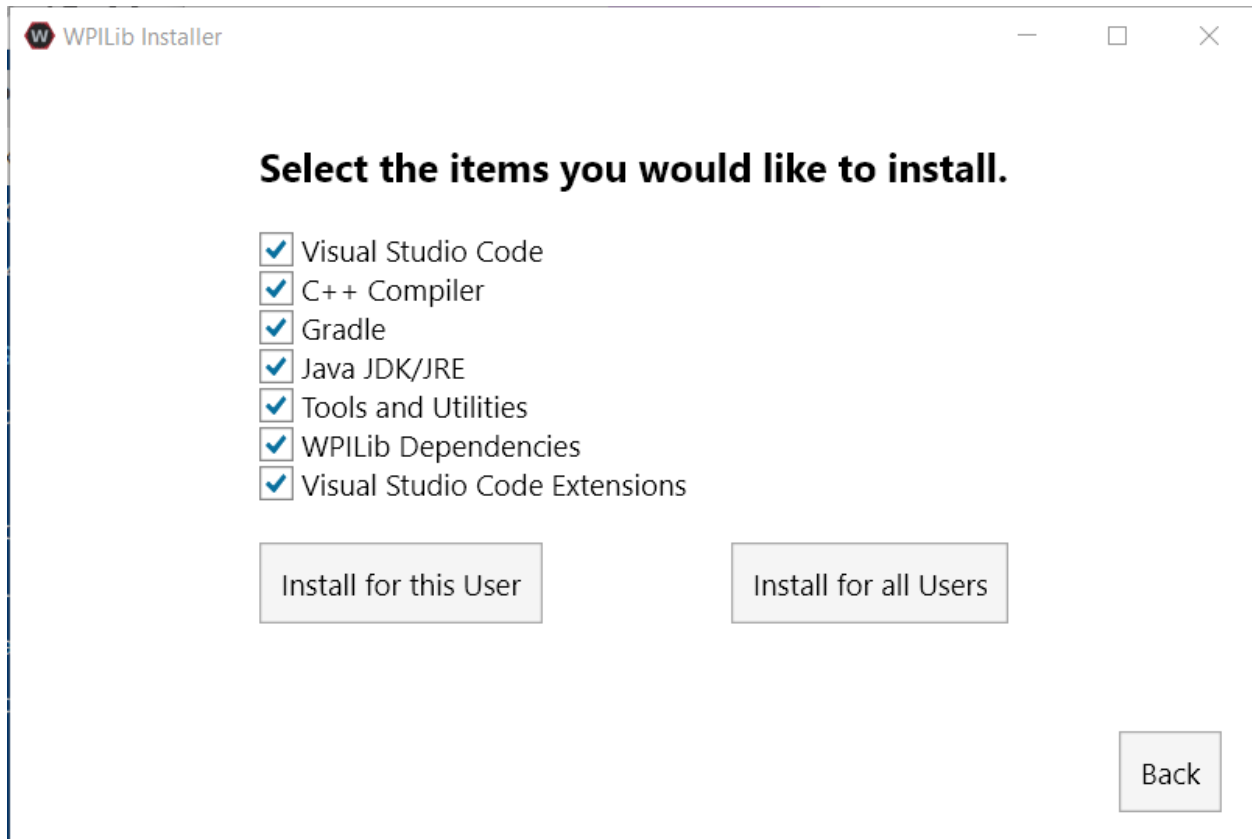


This next screen involves downloading VS Code. Unfortunately, due to licensing reasons, VS Code can not be bundled with the installer.



- Download VS Code for Single Install
 - This downloads VS Code only for the current platform, which is also the smallest download.
- Skip installing VS Code
 - Skips installing VS Code. Useful for advanced installations or configurations. Generally not recommended.
- Use Downloaded Offline Installer
 - Selecting this option will bring up a prompt allowing you to select a pre-existing zip file of VS Code that has been downloaded by the installer previously. This option does **not** let you select an already installed copy of VS Code on your machine.
- Download VS Code for Offline Install
 - This option downloads and saves a copy of VS Code for all platforms, which is useful for sharing the copy of the installer.

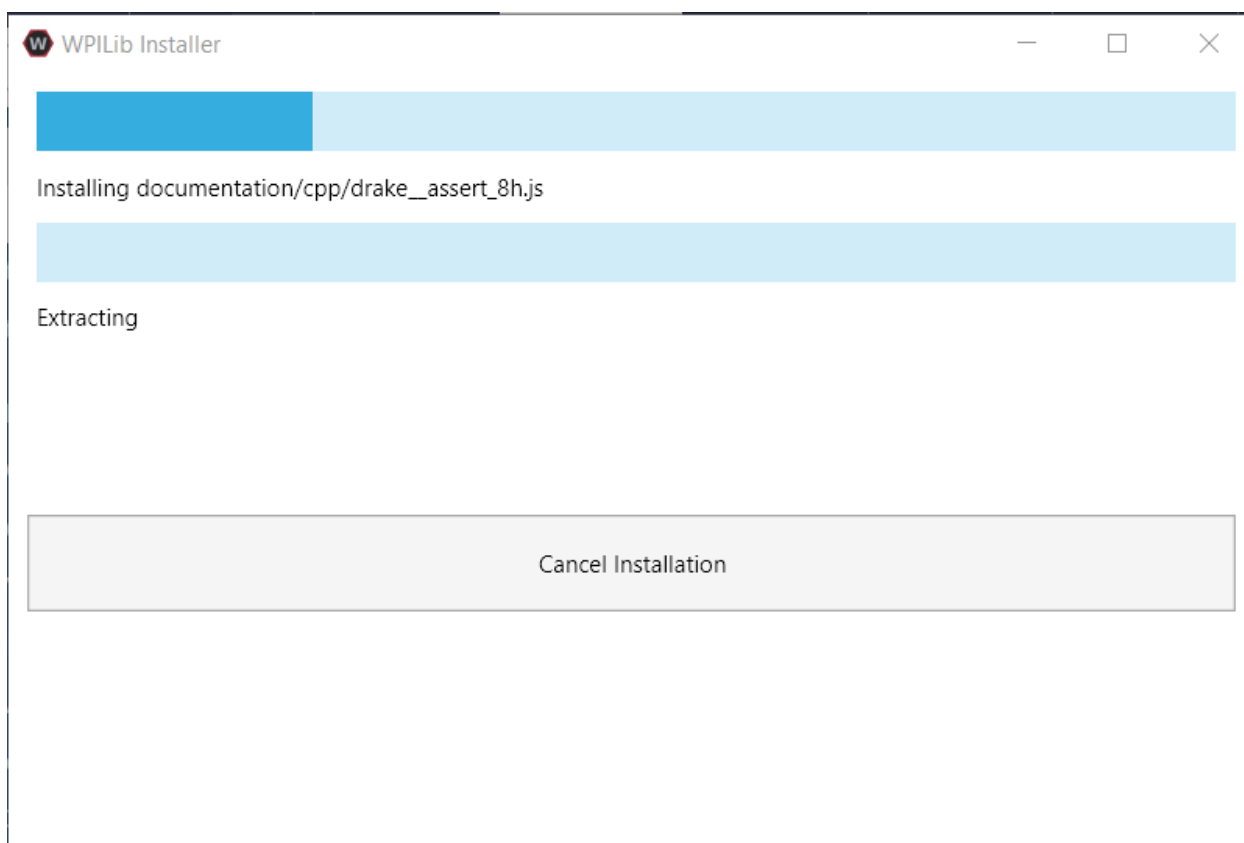
Go ahead and select *Download VS Code for Single Install*. This will begin the download process and can take a bit depending on internet connectivity (it's ~60MB). Once the download is done, select *Next*. You should be presented with a screen that looks similar to the one below.



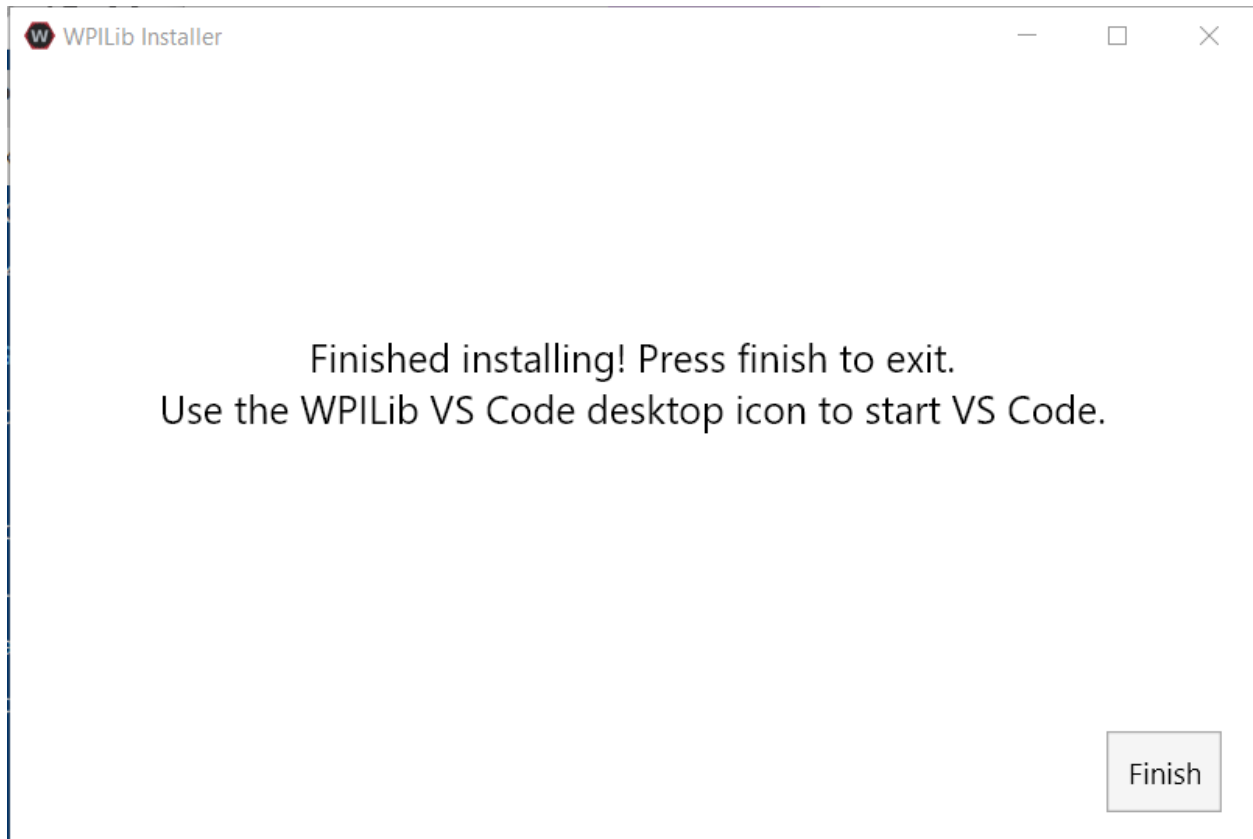
This showcases a list of options included with the WPILib installation. It's advised to just leave the default options selected.

You will notice two buttons, *Install for this User* and *Install for all Users*. *Install for this User* only installs it on the current user account, and does not require administrator privileges. However, *Install for all Users* installs the tools for all system accounts and *will* require administrator access. *Install for all Users* is not an option for macOS and Linux.

Select the option that is appropriate for you, and you'll be presented with the following installation screen.



After installation is complete, you will be presented with the finished screen.



Important: WPILib installs a separate version of VS Code than into an already existing installation. Each year has it's own copy of the tools appended with the year. IE: WPILib VS Code 2021. Please launch the WPILib VS Code and not a system installed copy!

Congratulations, the WPILib development environment and tooling is now installed on your computer! Press Finish to exit the installer.

3.4.4 Post-Installation

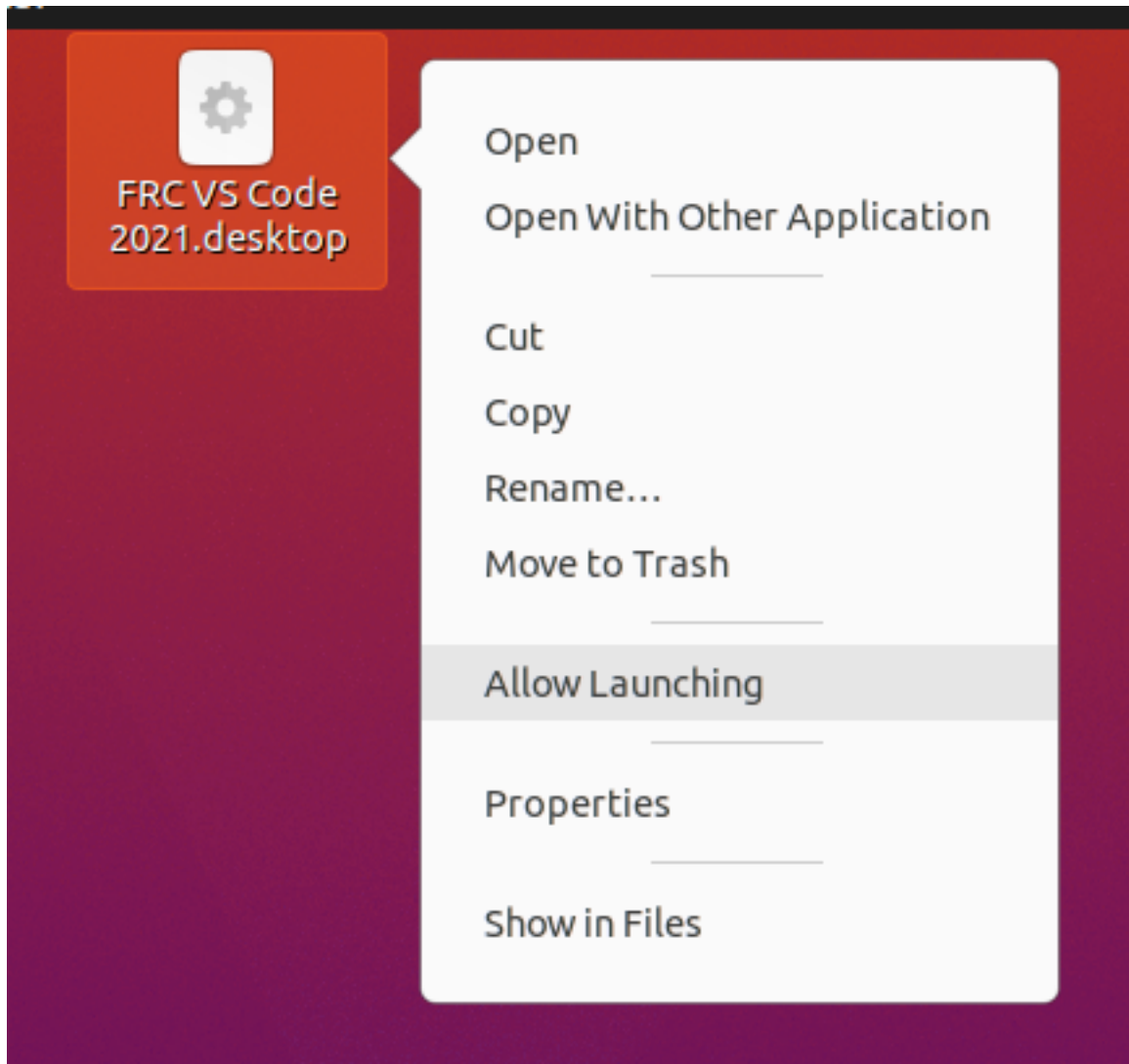
Some operating systems require some final action to complete installation.

macOS

Linux

After installation, the installer opens the WPILib VS Code folder. Drag the VS Code application to the dock. Eject WPILibInstaller image from the desktop.

Some versions of Linux (e.g. Ubuntu 20.04) require you to give the desktop shortcut the ability to launch. Right click on the desktop icon and select Allow Launching.



Note: Installing desktop tools and rebooting will create a folder on the desktop called YYYY WPILib Tools, where YYYY is the current year. Desktop tool shortcuts are not available on Linux and MacOS.

3.4.5 What is Installed?

The Offline Installer installs the following components:

- **Visual Studio Code** - The supported IDE for 2019 and later robot code development. The offline installer sets up a separate copy of VS Code for WPILib development, even if you already have VS Code on your machine. This is done because some of the settings that make the WPILib setup work may break existing workflows if you use VS Code for other projects.
- **C++ Compiler** - The toolchains for building C++ code for the roboRIO
- **Gradle** - The specific version of Gradle used for building/deploying C++ or Java robot

code

- **Java JDK/JRE** - A specific version of the Java JDK/JRE that is used to build Java robot code and to run any of the Java based Tools (Dashboards, etc.). This exists side by side with any existing JDK installs and does not overwrite the JAVA_HOME variable
- **WPILib Tools** - SmartDashboard, Shuffleboard, RobotBuilder, Outline Viewer, Pathweaver, Glass
- **WPILib Dependencies** - OpenCV, etc.
- **VS Code Extensions** - WPILib extensions for robot code development in VS Code

3.4.6 Uninstalling

WPILib is designed to install to different folders for different years, so that it is not necessary to uninstall a previous version before installing this year's WPILib. However, the following instructions can be used to uninstall WPILib if desired.

Windows

macOS

Linux

1. Delete the appropriate wpilib folder (2019: c:\Users\Public\frc2019, 2020 and later: c:\Users\Public\wpilib\YYYY where YYYY is the year to uninstall)
2. Delete the desktop icons at C:\Users\Public\Public Desktop
3. Delete the path environment variables.
 1. In the start menu, type environment and select "edit the system environment variables"
 2. Click on the environment variables button (1).
 3. In the user variables, select path (2) and then click on edit (3).
 4. Select the path with roborio\bin (4) and click on delete (5).
 5. Select the path with frccode and click on delete (5).
 6. Repeat steps 3-6 in the Systems Variable pane.

3.5 Next Steps

Congratulations! You have completed step 2 and should now have a working software development environment! Step 3 of this tutorial covers updating the hardware so that you can program it, while Step 4 showcases programming a robot in the VS Code Integrated Development Environment (IDE). For further information you can read through the [VS Code section](#) to familiarize yourself with the IDE.

Specific articles that are advised to be read are:

- [Visual Studio Code Basics](#)
- [WPILib Commands in Visual Studio Code](#)
- [Creating a Robot Program](#)
- [Building and Deploying Robot Code](#)
- [Installing 3rd Party Libraries](#)

Additionally, you may need to do extra configuration that is applicable to your team's robot. Please utilize the search feature to find necessary documentation.

Note: It's important that teams using 3rd-party CAN motor controllers look at the [Installing 3rd Party Libraries](#) article as extra steps are required to code for these devices.

Step 3: Preparing Your Robot

Warning: The CAN Web Dashboard plugin is no longer supported by the roboRIO Web Dashboard. To configure CTRE CAN devices such as the PCM and PDP, use [CTRE Phoenix Tuner](#)

4.1 Imaging your roboRIO

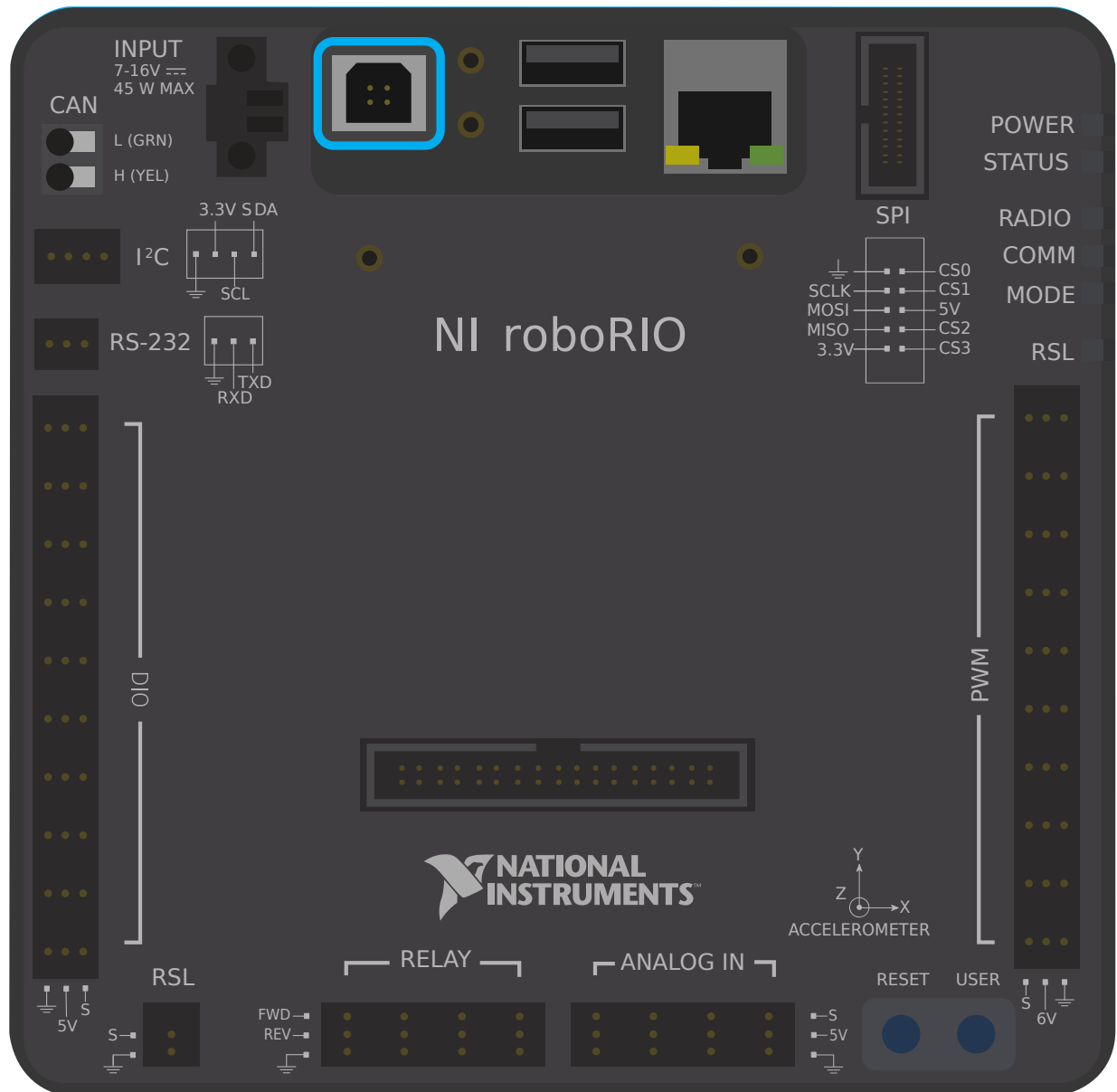
Warning: Before imaging your roboRIO, you must have completed installation of the [FRC Game Tools](#). You also must have the roboRIO power properly wired to the Power Distribution Panel. Make sure the power wires to the roboRIO are secure and that the connector is secure firmly to the roboRIO (4 total screws to check).

Important: An issue was discovered with the roboRIO image version 2021_v3.1 included in the initial release of the 2021 NI Game Tools that prevents successful imaging of roboRIOs. Teams that downloaded the installer prior to the morning of January 11, 2021 should [re-download and re-install](#) the latest installer (no need to uninstall first). The new version is 2021 f1. Users with the correct version installed will not see 2021_v3.1 as an option in the roboRIO imaging tool.

4.1.1 Configuring the roboRIO

The roboRIO Imaging Tool will be used to image your roboRIO with the latest software.

USB Connection



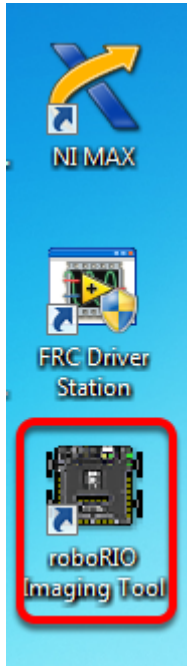
Connect a USB cable from the roboRIO USB Device port to the PC. This requires a USB Type A male (standard PC end) to Type B male cable (square with 2 cut corners), most commonly found as a printer USB cable.

Note: The roboRIO should only be imaged via the USB connection. It is not recommended to attempt imaging using the Ethernet connection.

Driver Installation

The device driver should install automatically. If you see a “New Device” pop-up in the bottom right of the screen, wait for the driver install to complete before continuing.

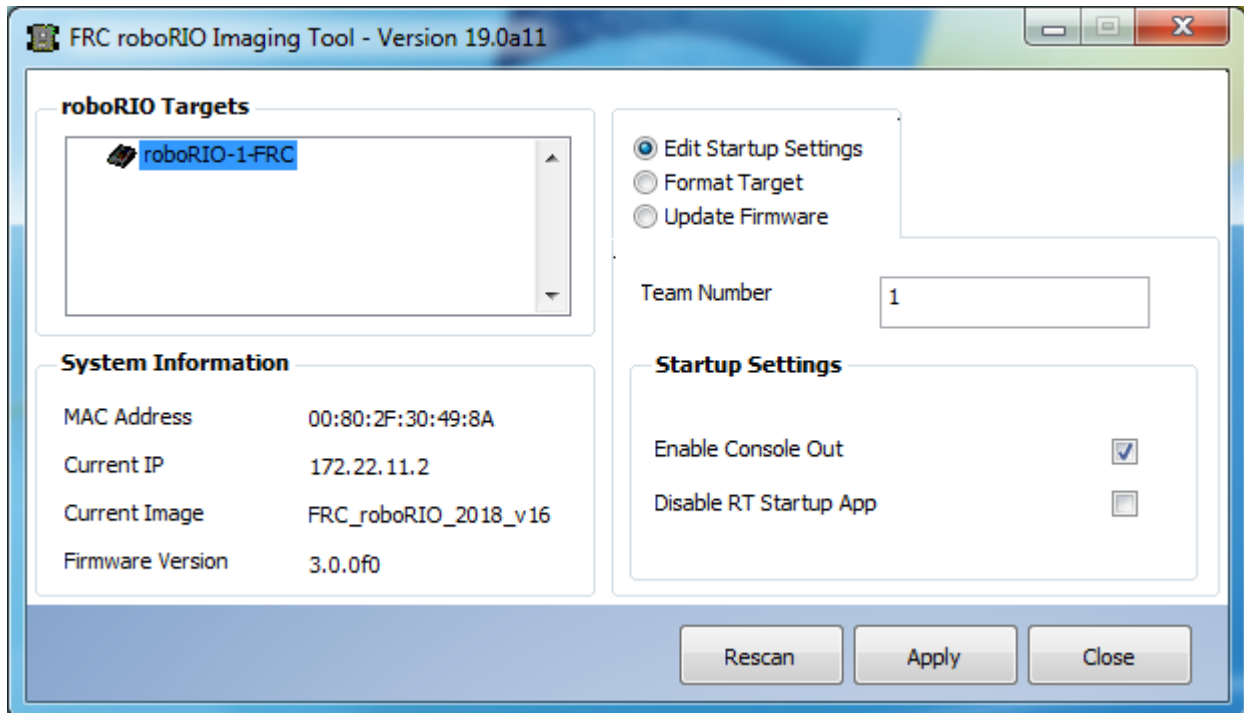
4.1.2 Launching the Imaging Tool



The roboRIO imaging tool and latest image are installed with the NI FRC® Game Tools. Launch the imaging tool by double clicking on the shortcut on the Desktop. If you have difficulties imaging your roboRIO, you may need to try right-clicking on the icon and selecting Run as Administrator instead.

Note: The roboRIO imaging tool is also located at C:\Program Files (x86)\National Instruments\LabVIEW YYYY\project\roboRIO Tool where YYYY is the current year - 1. If it's 2020, the directory would be LabVIEW 2019.

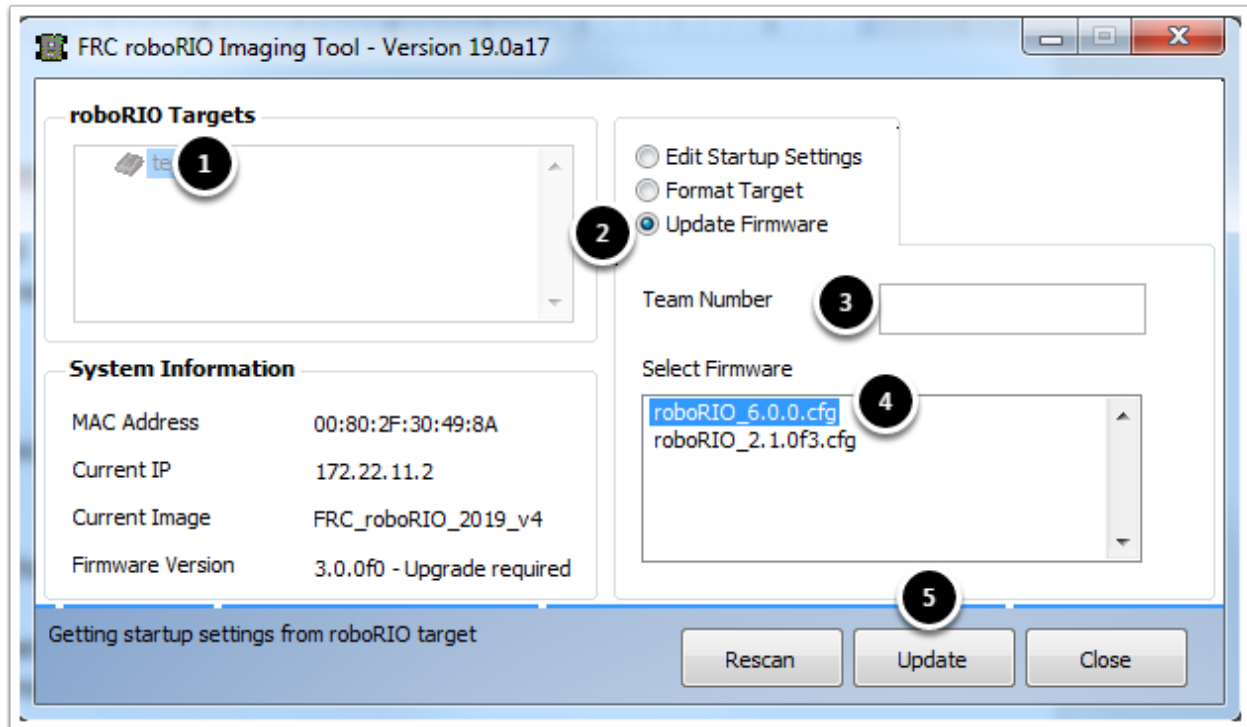
4.1.3 roboRIO Imaging Tool



After launching, the roboRIO Imaging Tool will scan for available roboRIOs and indicate any found in the top left box. The bottom left box will show information and settings for the roboRIO currently selected. The right hand pane contains controls for modifying the roboRIO settings:

- **Edit Startup Settings** - This option is used when you want to configure the startup settings of the roboRIO (the settings in the right pane), without imaging the roboRIO.
- **Format Target** - This option is used when you want to load a new image on the roboRIO (or reflash the existing image). This is the most common option.
- **Update Firmware** - This option is used to update the roboRIO firmware. For this season, the imaging tool will require roboRIO firmware to be version 5.0 or greater.

Updating Firmware

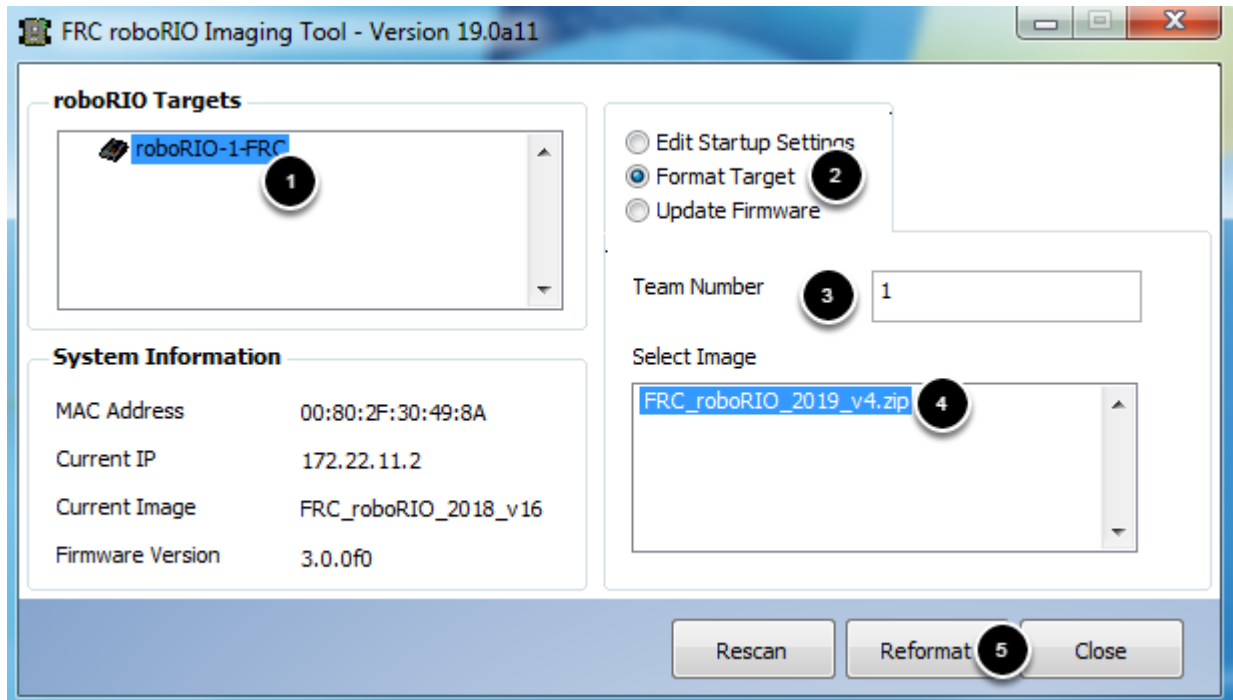


roboRIO firmware must be at least v5.0 to work with the 2019 or later image. If your roboRIO is at least version 5.0 (as shown in the bottom left of the imaging tool) you do not need to update.

To update roboRIO firmware:

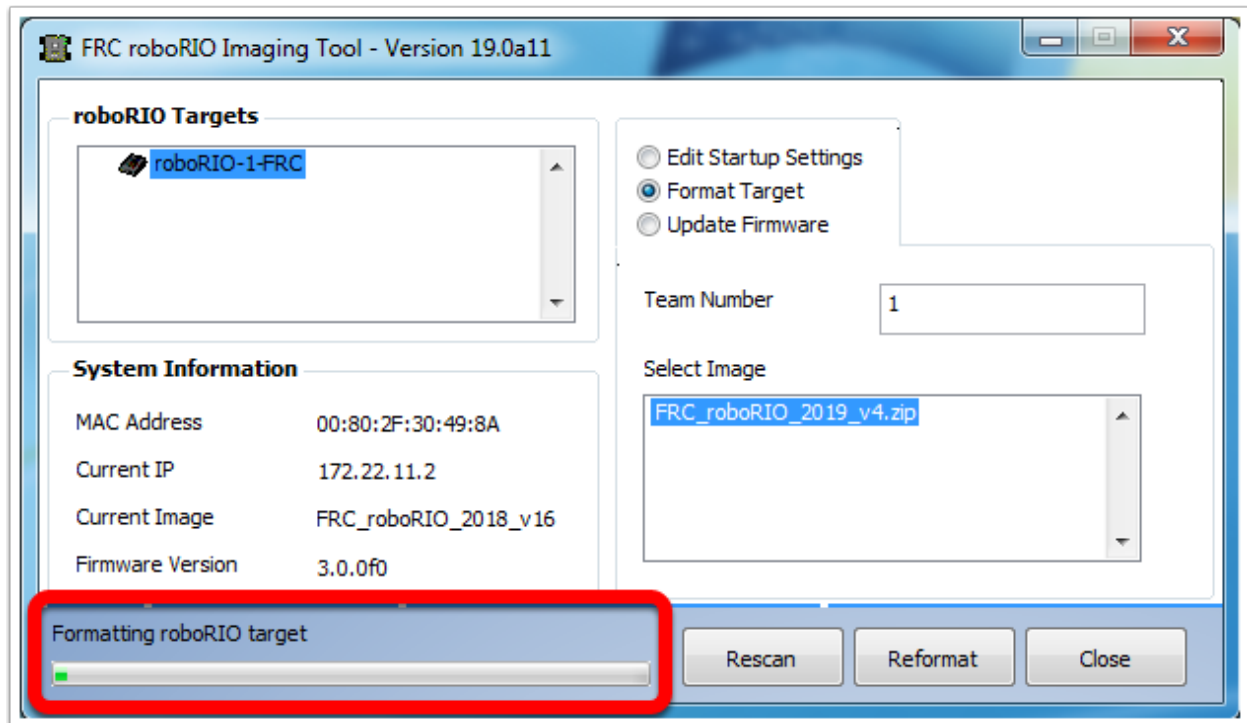
1. Make sure your roboRIO is selected in the top left pane.
2. Select Update Firmware in the top right pane
3. Enter a team number in the Team Number box
4. Select the latest firmware file in the bottom right
5. Click the **Update** button

4.1.4 Imaging the roboRIO



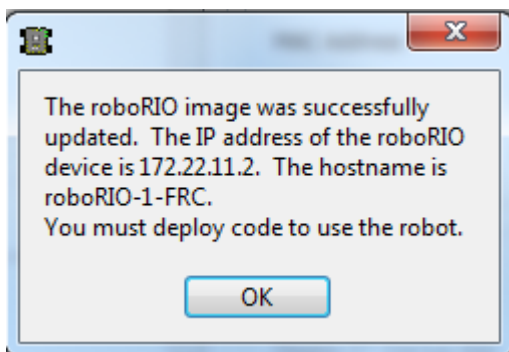
1. Make sure the roboRIO is selected in the top left pane
2. Select Format Target in the right pane
3. Enter your team number in the box
4. Select the latest image version in the box.
5. Click Reformat to begin the imaging process.

4.1.5 Imaging Progress



The imaging process will take approximately 3-10 minutes. A progress bar in the bottom left of the window will indicate progress.

4.1.6 Imaging Complete



When the imaging completes you should see the dialog above. Click Ok, then click the Close button at the bottom right to close the imaging tool. Reboot the roboRIO using the Reset button to have the new team number take effect.

Note: The default CAN webdash functionality has been removed from the image (CAN devices will still work from robot code). You will need to use the tools provided by individual vendors to service their CAN devices.

4.1.7 Troubleshooting

If you are unable to image your roboRIO, troubleshooting steps include:

- Try running the roboRIO Imaging Tool as Administrator by right-clicking on the Desktop icon to launch it.
- Try accessing the roboRIO webpage with a web-browser at <http://172.22.11.2/> and/or verify that the NI network adapter appears in your list of Network Adapters in the Control Panel. If not, try re-installing the NI FRC Game Tools or try a different PC.
- *Disable all other network adapters*
- Make sure your firewall is turned off.
- Some teams have experienced an issue where imaging fails if the device name of the computer you're using has a dash (-) in it. Try renaming the computer (or using a different PC).
- Try booting the roboRIO into Safe Mode by pressing and holding the reset button for at least 5 seconds.
- Try a different PC

4.2 Programming your Radio

This guide will show you how to use the FRC® Radio Configuration Utility software to configure your robot's wireless bridge for use outside of FRC events.

Before you begin using the software:

1. *Disable all other network adapters*
2. Plug directly from your computer into the wireless bridge ethernet port closest to the power jack and make sure no other devices are connected to your computer via ethernet.

Warning: The OM5P-AN and AC use the same power plug as the D-Link DAP1522, however they are 12V radios. Wire the radio to the 12V 2A terminals on the VRM (center-pin positive).

4.2.1 Prerequisites

The FRC Radio Configuration Utility requires administrator privileges to configure the network settings on your machine. The program should request the necessary privileges automatically (may require a password if run from a non-administrator account), but if you are having trouble, try running it from an administrator account.

Download the latest FRC Radio Configuration Utility Installer from the following links:

Note: There are no changes to this tool for the 2021 season so the 20.0.0 version is still the latest available.

[FRC Radio Configuration 20.0.0](#)

FRC Radio Configuration 20.0.0 Israel Version

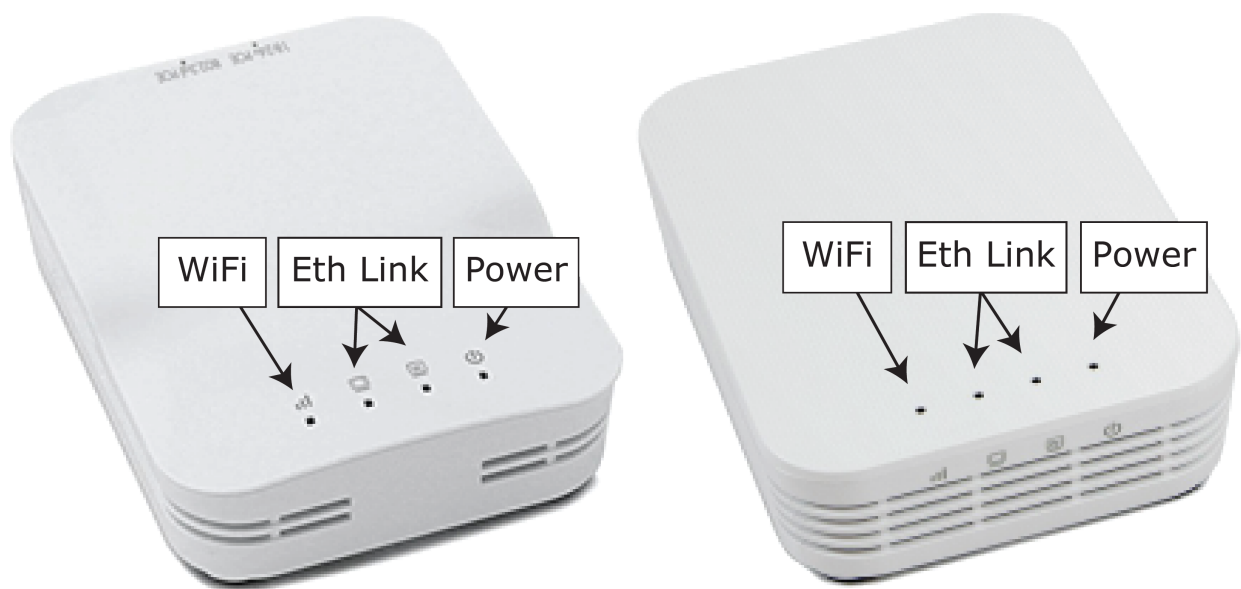
Note: The _IL version is for Israel teams and contains a version of the OM5PAC firmware with restricted channels for use in Israel.

4.2.2 Application Notes

By default, the Radio Configuration Utility will program the radio to enforce the 4Mbps bandwidth limit on traffic exiting the radio over the wireless interface. In the home configuration (AP mode) this is a total, not a per client limit. This means that streaming video to multiple clients is not recommended.

The Utility has been tested on Windows 7, 8 and 10. It may work on other operating systems, but has not been tested.

Programmed Configuration



The Radio Configuration Utility programs a number of configuration settings into the radio when run. These settings apply to the radio in all modes (including at events). These include:

- Set a static IP of 10.TE.AM.1
- Set an alternate IP on the wired side of 192.168.1.1 for future programming
- Bridge the wired ports so they may be used interchangeably
- The LED configuration noted in the graphic above.
- 4Mb/s bandwidth limit on the outbound side of the wireless interface (may be disabled for home use)
- QoS rules for internal packet prioritization (affects internal buffer and which packets to discard if bandwidth limit is reached). These rules are:
 - Robot Control and Status (UDP 1110, 1115, 1150)

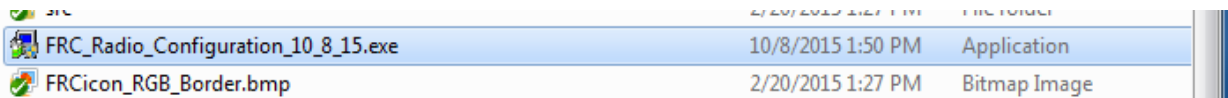
- Robot TCP & *NetworkTables* (TCP 1735, 1740)
- Bulk (All other traffic). (disabled if BW limit is disabled)
- DHCP server enabled. Serves out:
 - 10.TE.AM.11 - 10.TE.AM.111 on the wired side
 - 10.TE.AM.138 - 10.TE.AM.237 on the wireless side
 - Subnet mask of 255.255.255.0
 - Broadcast address 10.TE.AM.255
- DNS server enabled. DNS server IP and domain suffix (.lan) are served as part of the DHCP.

At home only:

- SSID may have a “Robot Name” appended to the team number to distinguish multiple networks.
- Firewall option may be enabled to mimic the field firewall rules (open ports may be found in the Game Manual)

Warning: It is not possible to modify the configuration manually.
--

4.2.3 Install the Software

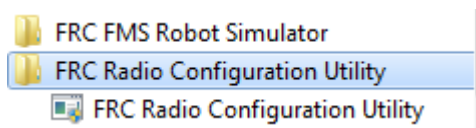


	DATE/TIME	FILE TYPE
FRC_Radio_Configuration_10_8_15.exe	10/8/2015 1:50 PM	Application
FRCicon_RGB_Border.bmp	2/20/2015 1:27 PM	Bitmap Image

Double click on FRC_Radio_Configuration_VERSION.exe to launch the installer. Follow the prompts to complete the installation.

Part of the installation prompts will include installing Npcap if it is not already present. The Npcap installer contains a number of checkboxes to configure the install. You should leave the options as the defaults.

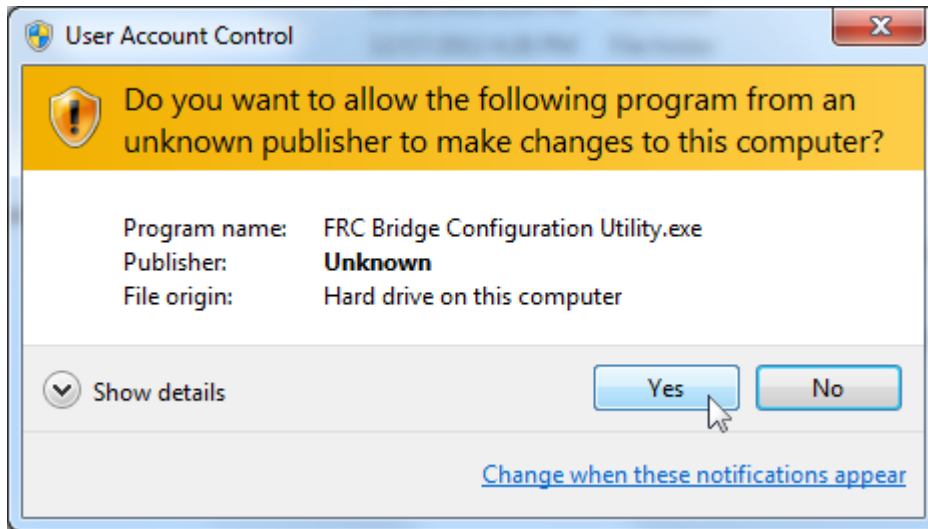
4.2.4 Launch the software



Use the Start menu or desktop shortcut to launch the program.

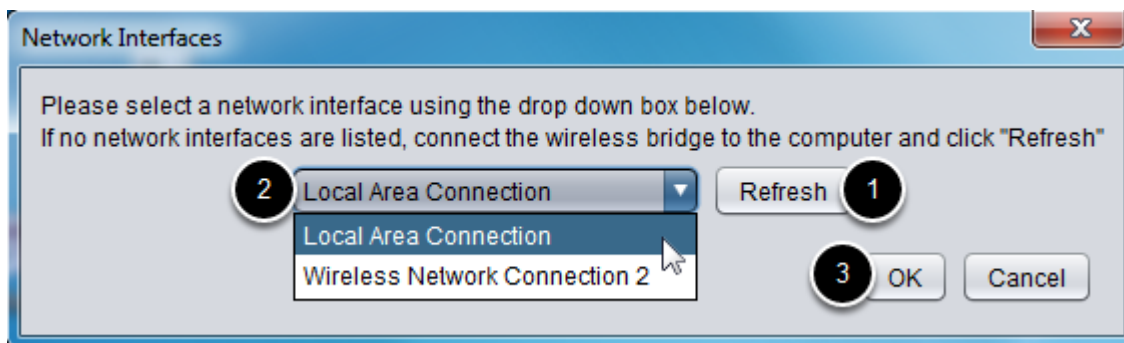
Note: If you need to locate the program, it is installed to C:\Program Files (x86)\FRC Radio Configuration Utility. For 32-bit machines the path is C:\Program Files\FRC Radio Configuration Utility

4.2.5 Allow the program to make changes, if prompted



If your computer is running Windows, a prompt may appear about allowing the configuration utility to make changes to the computer. Click Yes if the prompt appears.

4.2.6 Select the network interface



Use the pop-up window to select the which ethernet interface the configuration utility will use to communicate with the wireless bridge. On Windows machines, ethernet interfaces are typically named "Local Area Connection". The configuration utility can not program a bridge over a wireless connection.

1. If no ethernet interfaces are listed, click *Refresh* to re-scan for available interfaces.
2. Select the interface you want to use from the drop-down list.
3. Click *OK*.

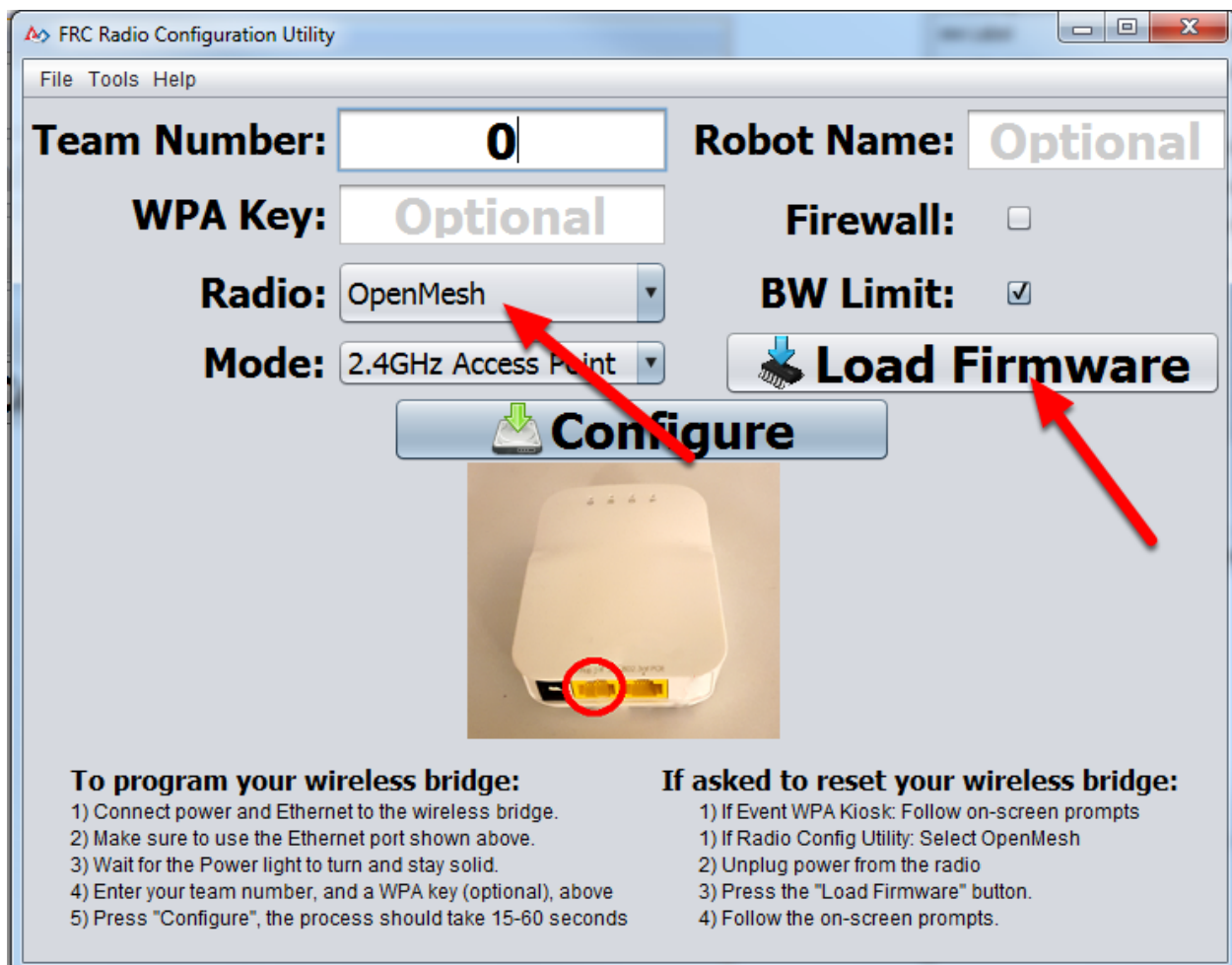
4.2.7 Open Mesh Firmware Note

For the FRC Radio Configuration Utility to program the OM5P-AN and OM5P-AC radio, the radio must be running an FRC specific build of the OpenWRT firmware. OM5P-AC radios in the 2019 KoP should not need an update.

If you do not need to update or re-load the firmware, skip the next step.

Warning: Radios used in 2019 **do not** need to be updated before configuring, the 2020 tool uses the same 2019 firmware.

4.2.8 Loading FRC Firmware to Open Mesh Radio



If you need to load the FRC firmware (or reset the radio), you can do so using the FRC Radio Configuration Utility.

1. Follow the instructions above to install the software, launch the program and select the Ethernet interface.
2. Make sure the Open Mesh radio is selected in the Radio dropdown.
3. Make sure the radio is connected to the PC via Ethernet.

4. Unplug the power from the radio. (If using a PoE cable, this will also be unplugging the Ethernet to the PC, this is fine)
5. Press the Load Firmware button
6. When prompted, plug in the radio power. The software should detect the radio, load the firmware and prompt you when complete.

Warning: If you see an error about NPF name, try disabling all adapters other than the one being used to program the radio. If only one adapter is found, the tool should attempt to use that one. See the steps in [Disabling Network Adapters](#) for more info.

Teams may also see this error with foreign language Operating Systems. If you experience issues loading firmware or programming on a foreign language OS, try using an English OS, such as on the KOP provided PC or setting the Locale setting to “en_us” as described on [this page](#).

4.2.9 Select Radio and Operating Mode

Team Number: **Robot Name:**

WPA Key: **Firewall:** ☐

Radio: **BW Limit:** ☒

Mode: **Load Firmware**

Configure

To program your wireless bridge:

- 1) Connect power and Ethernet to the wireless bridge.
- 2) Make sure to use the Ethernet port shown above.
- 3) Wait for the Power light to turn and stay solid.
- 4) Enter your team number, and a WPA key (optional), above
- 5) Press "Configure", the process should take 15-60 seconds

If asked to reset your wireless bridge:

- 1) If Event WPA Kiosk: Follow on-screen prompts
- 1) If Radio Config Utility: Select OpenMesh
- 2) Unplug power from the radio
- 3) Press the "Load Firmware" button.
- 4) Follow the on-screen prompts.

1. Select which radio you are configuring using the drop-down list.

2. Select which operating mode you want to configure. For most cases, the default selection of 2.4GHz Access Point will be sufficient. If your computers support it, the 5GHz AP mode is recommended, as 5GHz is less congested in many environments.

4.2.10 Select Options

FRC Radio Configuration Utility

File Tools Help

Team Number:

WPA Key:

Radio:

Mode:

Robot Name:

Firewall: ☐

BW Limit: ☒

Load Firmware

Configure

To program your wireless bridge:

- 1) Connect power and Ethernet to the wireless bridge.
- 2) Make sure to use the Ethernet port shown above.
- 3) Wait for the Power light to turn and stay solid.
- 4) Enter your team number, and a WPA key (optional), above
- 5) Press "Configure", the process should take 15-60 seconds

If asked to reset your wireless bridge:

- 1) If Event WPA Kiosk: Follow on-screen prompts
- 1) If Radio Config Utility: Select OpenMesh
- 2) Unplug power from the radio
- 3) Press the "Load Firmware" button.
- 4) Follow the on-screen prompts.

The default values of the options have been selected to match the use case of most teams, however, you may wish to customize these options to your specific scenario:

1. **Robot Name:** This is a string that gets appended to the SSID used by the radio. This allows you to have multiple networks with the same team number and still be able to distinguish them.
2. **Firewall:** If this box is checked, the radio firewall will be configured to attempt to mimic the port blocking behavior of the firewall present on the FRC field. For a list of open ports, please see the FRC Game Manual.
3. **BW Limit:** If this box is checked, the radio enforces a 4 Mbps bandwidth limit like it does when programmed at events. Note that in AP mode, this is a total limit, not per client, so streaming video to multiple clients simultaneously may cause undesired behavior.

Note: Firewall and BW Limit only apply to the Open Mesh radios. These options have no

effect on D-Link radios.

Warning: The “Firewall” option configures the radio to emulate the field firewall. This means that you will not be able to deploy code wirelessly with this option enabled. This is useful for simulating blocked ports that may exist at competitions.

4.2.11 Starting the Configuration Process

Team Number: **Robot Name:**

WPA Key: **Firewall:** ☐

Radio: **BW Limit:** ☒

Mode: **Load Firmware**

Configure

To program your wireless bridge:

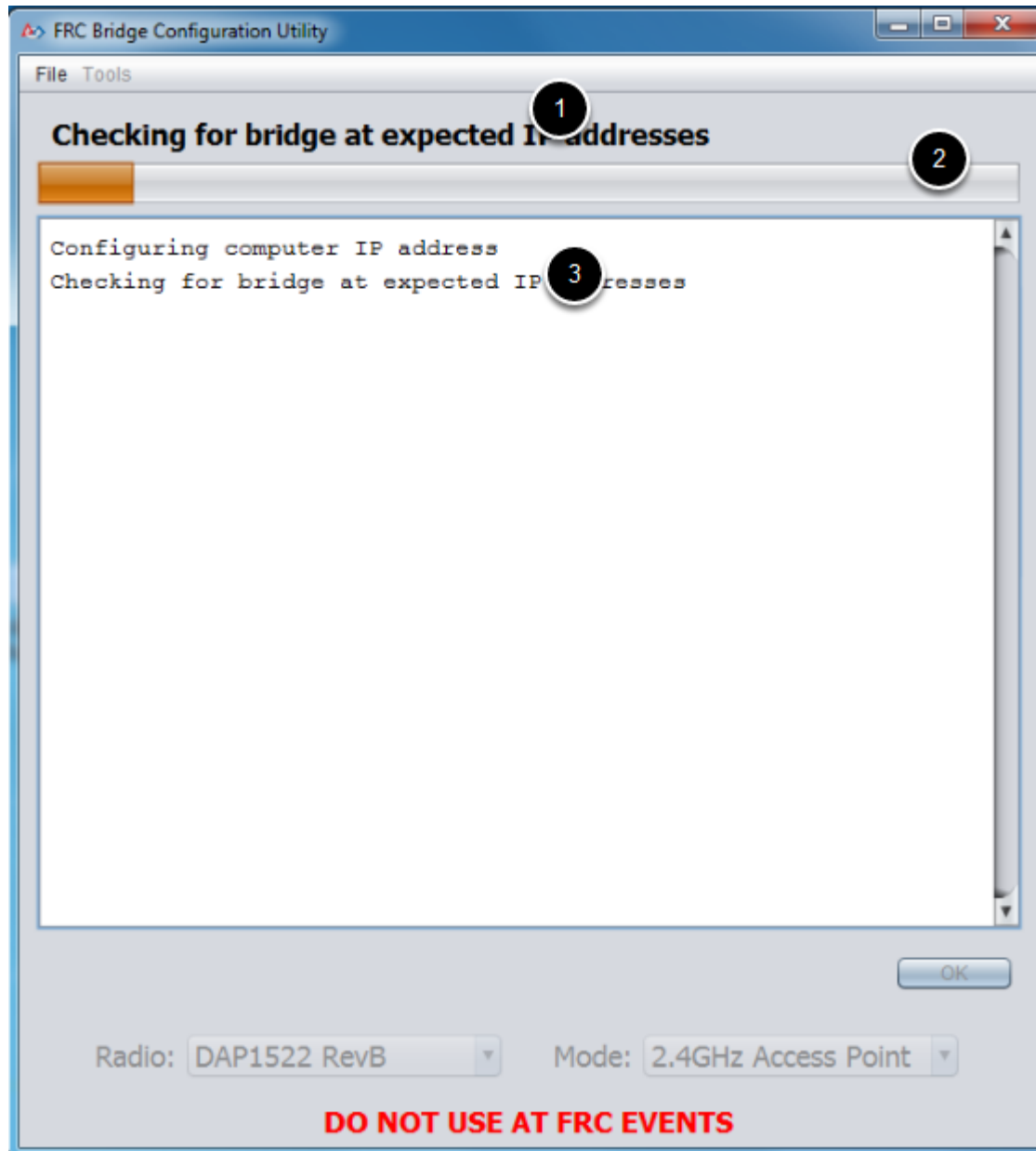
- 1) Connect power and Ethernet to the wireless bridge.
- 2) Make sure to use the Ethernet port shown above.
- 3) Wait for the Power light to turn and stay solid.
- 4) Enter your team number, and a WPA key (optional), above
- 5) Press "Configure", the process should take 15-60 seconds

If asked to reset your wireless bridge:

- 1) If Event WPA Kiosk: Follow on-screen prompts
- 1) If Radio Config Utility: Select OpenMesh
- 2) Unplug power from the radio
- 3) Press the "Load Firmware" button.
- 4) Follow the on-screen prompts.

Follow the on-screen instructions for preparing your wireless bridge, entering the settings the bridge will be configured with, and starting the configuration process. These on-screen instructions update to match the bridge model and operating mode chosen.

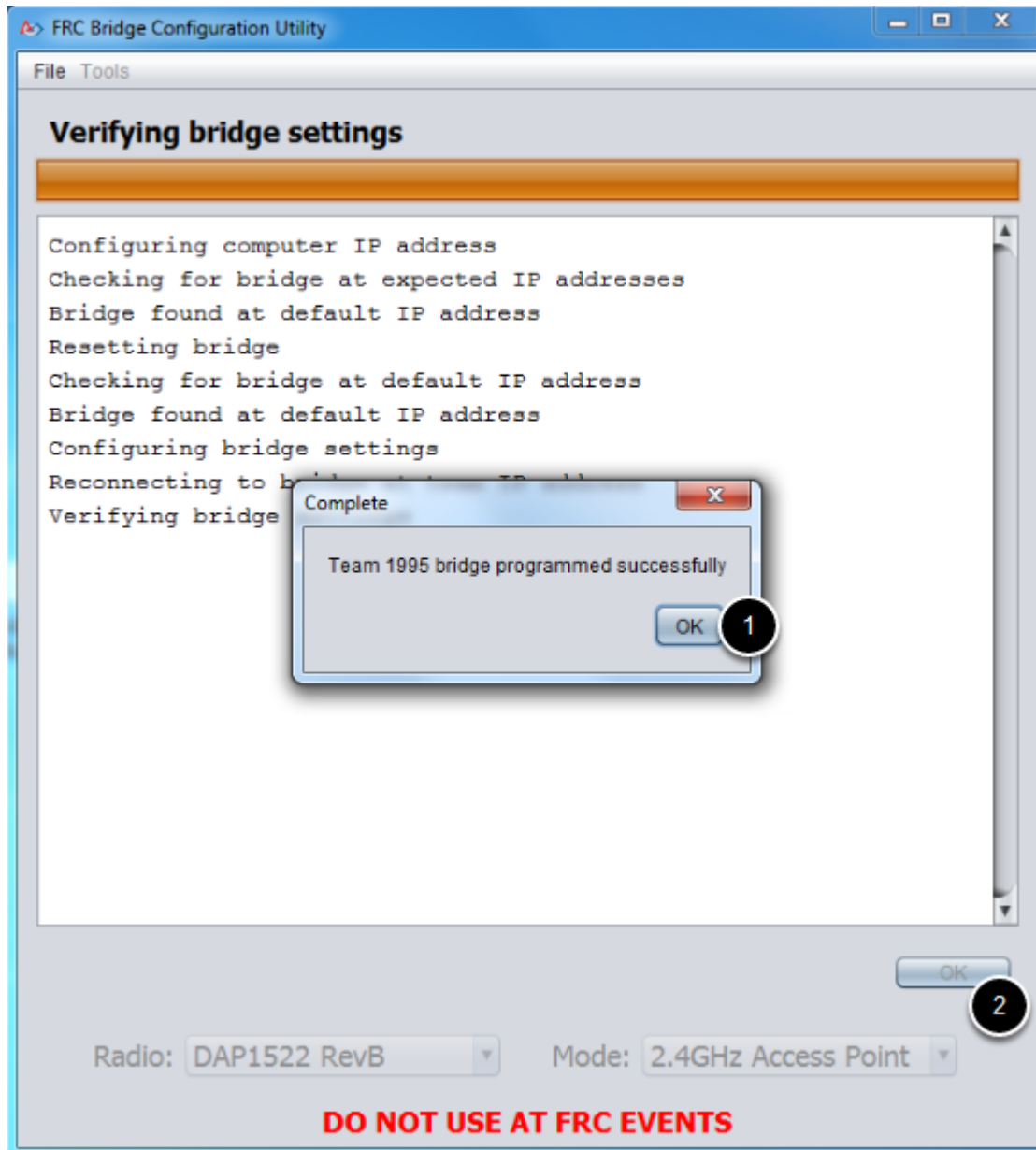
4.2.12 Configuration Progress



Throughout the configuration process, the window will indicate:

1. The step currently being executed.
2. The overall progress of the configuration process.
3. All steps executed so far.

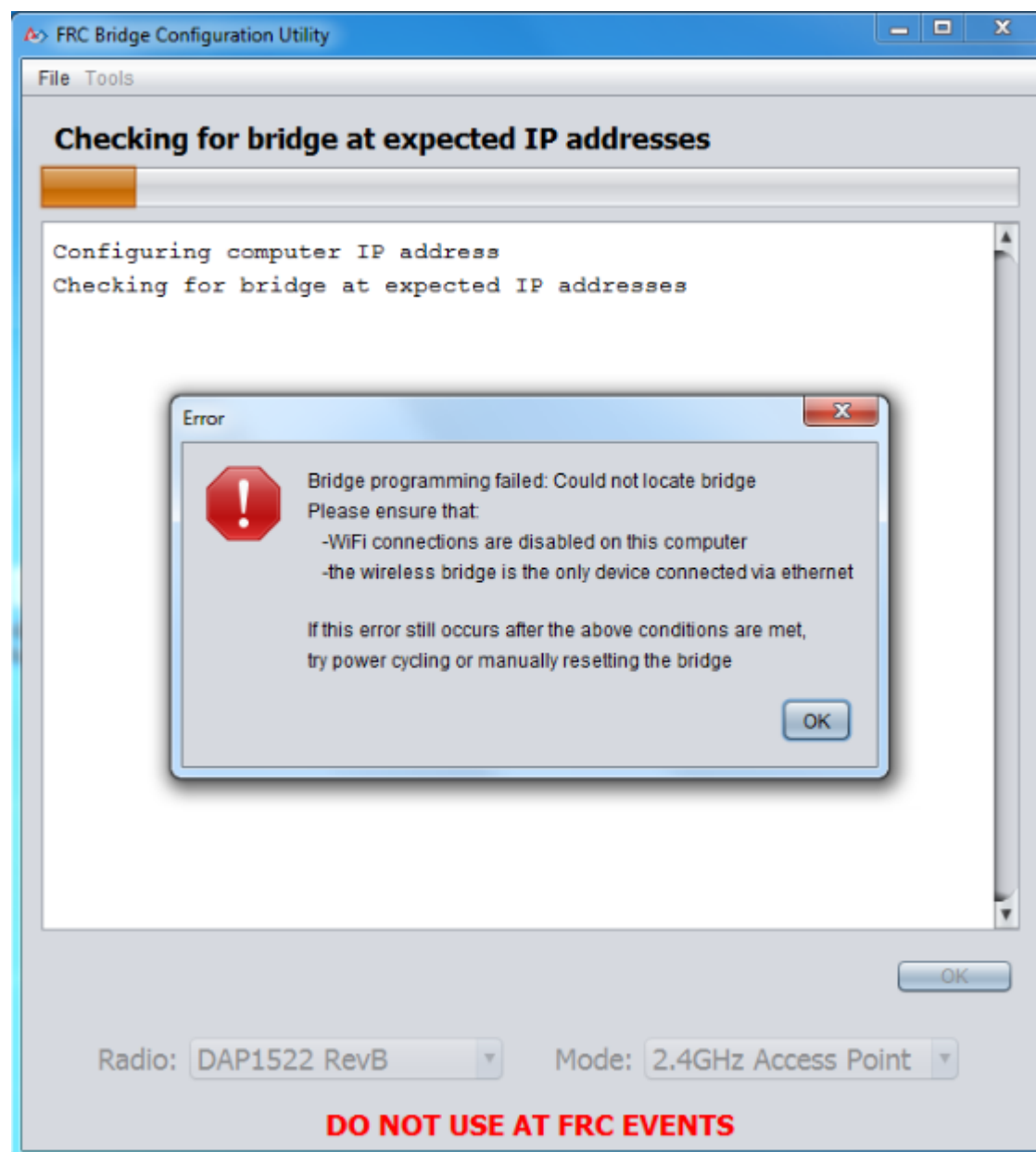
4.2.13 Configuration Completed



Once the configuration is complete:

1. Press *OK* on the dialog window.
2. Press *OK* on the main window to return to the settings screen.

4.2.14 Configuration Errors



If an error occurs during the configuration process, follow the instructions in the error message to correct the problem.

4.2.15 Troubleshooting

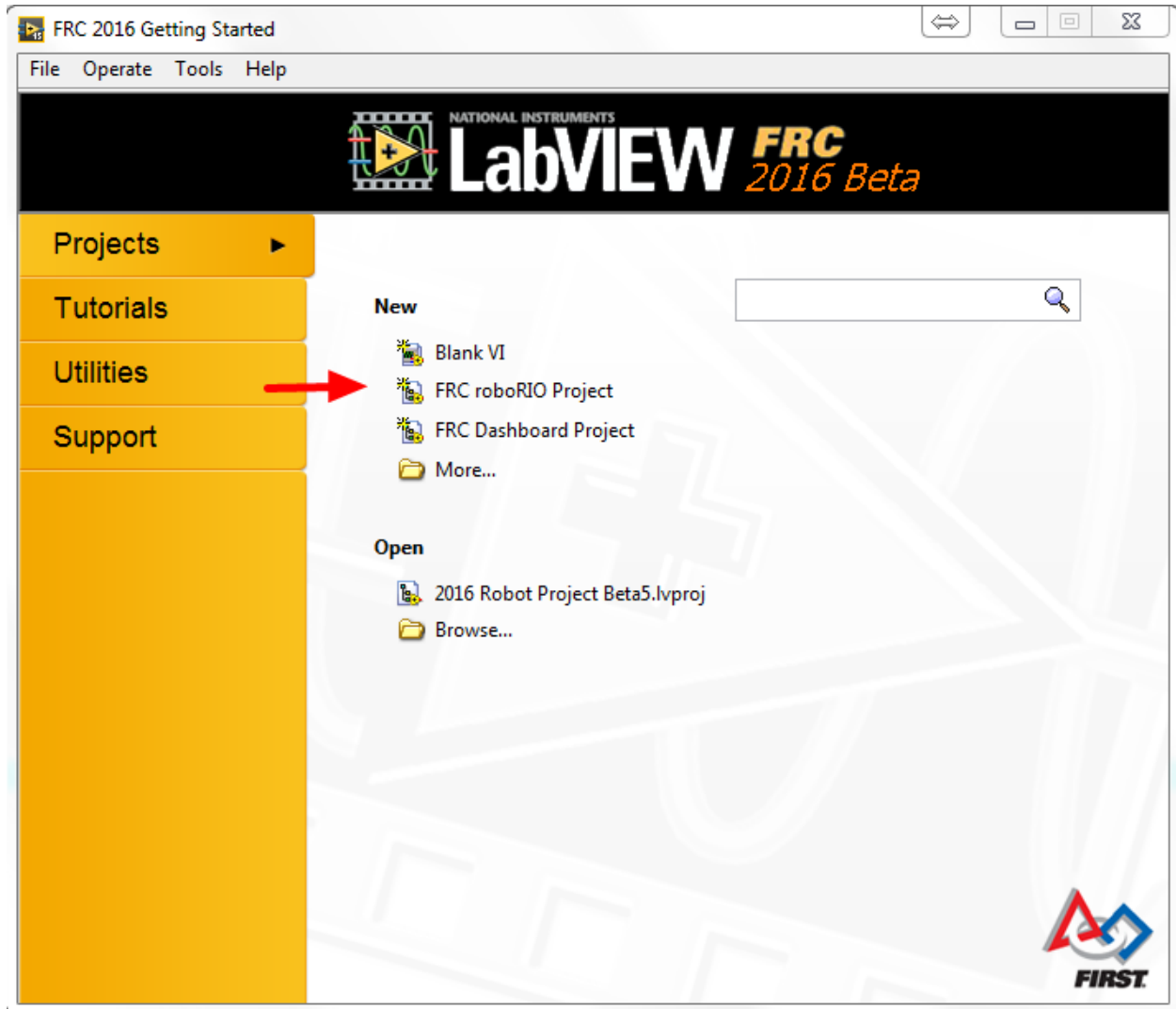
- *Disable all other network adapters.*
- Make sure you wait long enough that the power light has stayed solid for 10 seconds.
- Make sure you have the correct network interface, and only one interface is listed in the drop-down.
- Plug directly from your computer into the wireless bridge and make sure no other devices are connected to your computer via ethernet.
- Ensure the ethernet is plugged into the port closest to the power jack on the wireless bridge.
- If using a foreign language Operating System, try using an English OS, such as on the KOP provided PC or setting the Locale setting to “en_us” as described on [this page](#).

Step 4: Programming your Robot

5.1 Creating your Benchtop Test Program (LabVIEW)

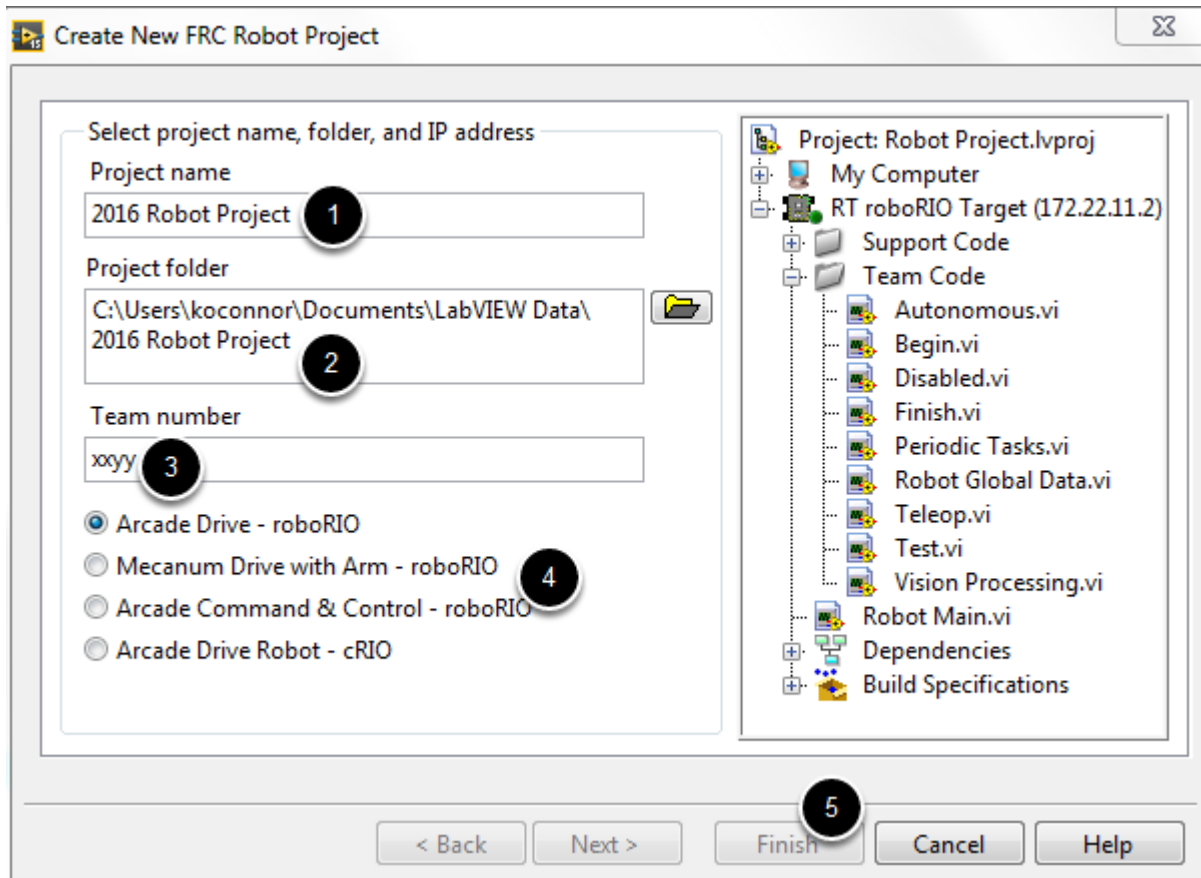
This document covers how to create, build and load an FRC® LabVIEW program onto a roboRIO. Before beginning, make sure that you have installed LabVIEW for FRC and the FRC Driver Station and that you have configured and imaged your roboRIO as described previously.

5.1.1 Creating a Project



Launch LabVIEW and click the FRC roboRIO Robot Project link in the Projects window to display the Create New FRC Robot Project dialog box.

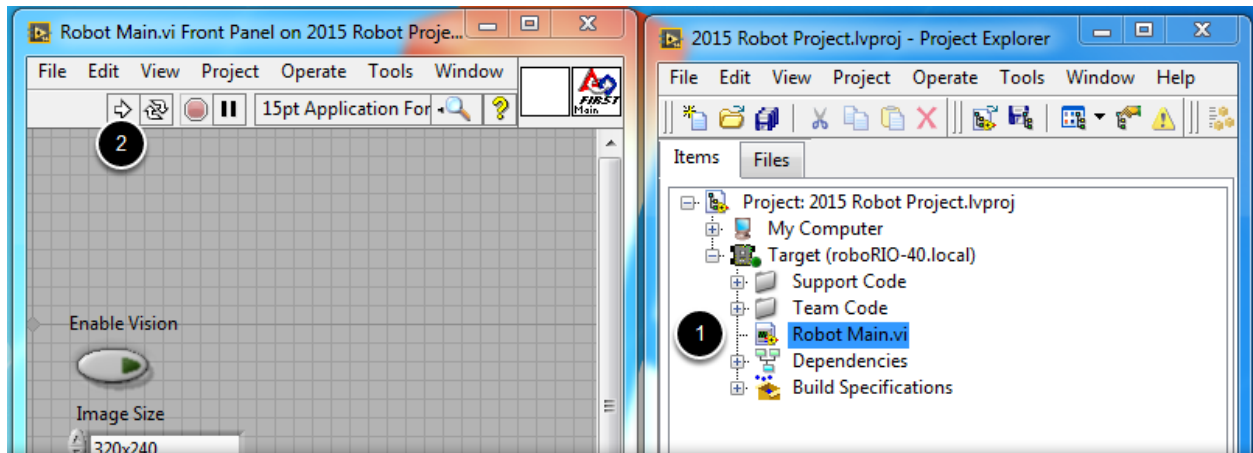
5.1.2 Configuring Project



Fill in the Create New FRC Project Dialog:

1. Pick a name for your project
2. Select a folder to place the project in.
3. Enter your team number
4. Select a project type. If unsure, select Arcade Drive - roboRIO.
5. Click Finish

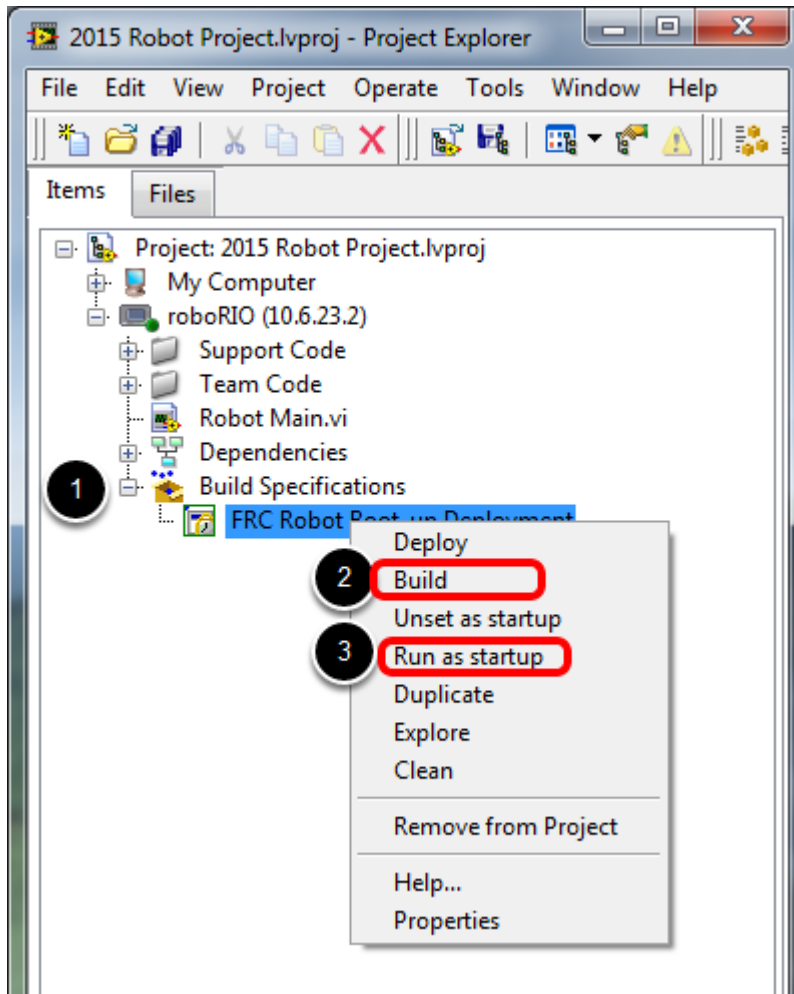
5.1.3 Running the Program



1. In the Project Explorer window, double-click the Robot Main.vi item to open the Robot Main VI.
2. Click the Run button (White Arrow on the top ribbon) of the Robot Main VI to deploy the VI to the roboRIO. LabVIEW deploys the VI, all items required by the VI, and the target settings to memory on the roboRIO. If prompted to save any VIs, click Save on all prompts.
3. Click the Abort button of the Robot Main VI. Notice that the VI stops. When you deploy a program with the Run button, the program runs on the roboRIO, but you can manipulate the front panel objects of the program from the host computer.

Note: A program deployed in this manner will not remain on the roboRIO after a power cycle. To deploy a program to run every time the roboRIO starts follow the next step, Deploying the program.

5.1.4 Deploying the program



To run in the competition, you will need to deploy a program to your roboRIO. This allows the program to survive across reboots of the controller, but doesn't allow the same debugging features (front panel, probes, highlight execution) as running from the front panel. To deploy your program:

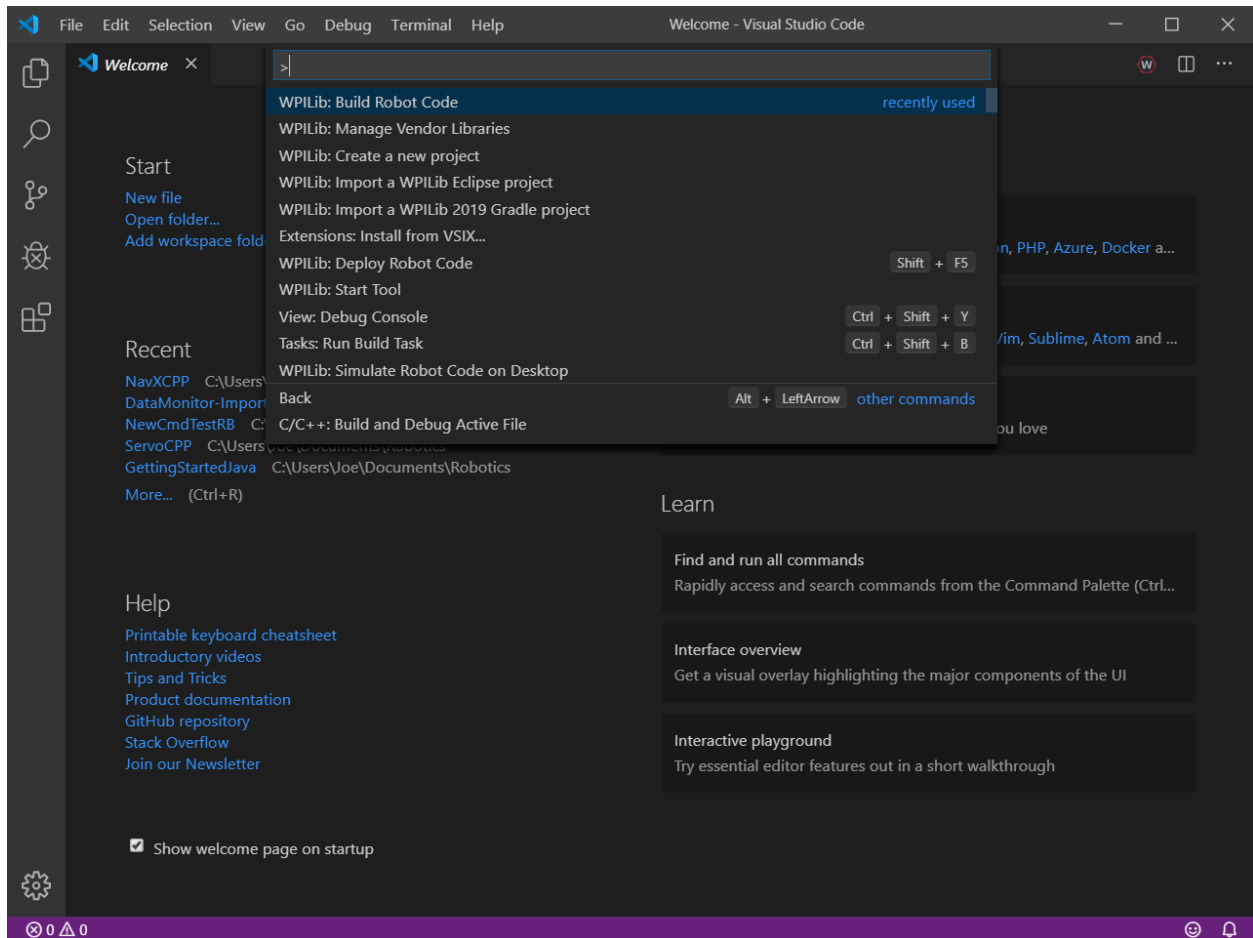
1. In the Project Explorer, click the + next to Build Specifications to expand it.
2. Right-click on FRC Robot Boot-up Deployment and select Build. Wait for the build to complete.
3. Right-click again on FRC Robot Boot-Up Deployment and select Run as Startup. If you receive a conflict dialog, click OK. This dialog simply indicates that there is currently a program on the roboRIO which will be terminated/replaced.
4. Either check the box to close the deployment window on successful completion or click the close button when the deployment completes.
5. The roboRIO will automatically start running the deployed code within a few seconds of the dialog closing.

5.2 Creating your Benchtop Test Program (C++/Java)

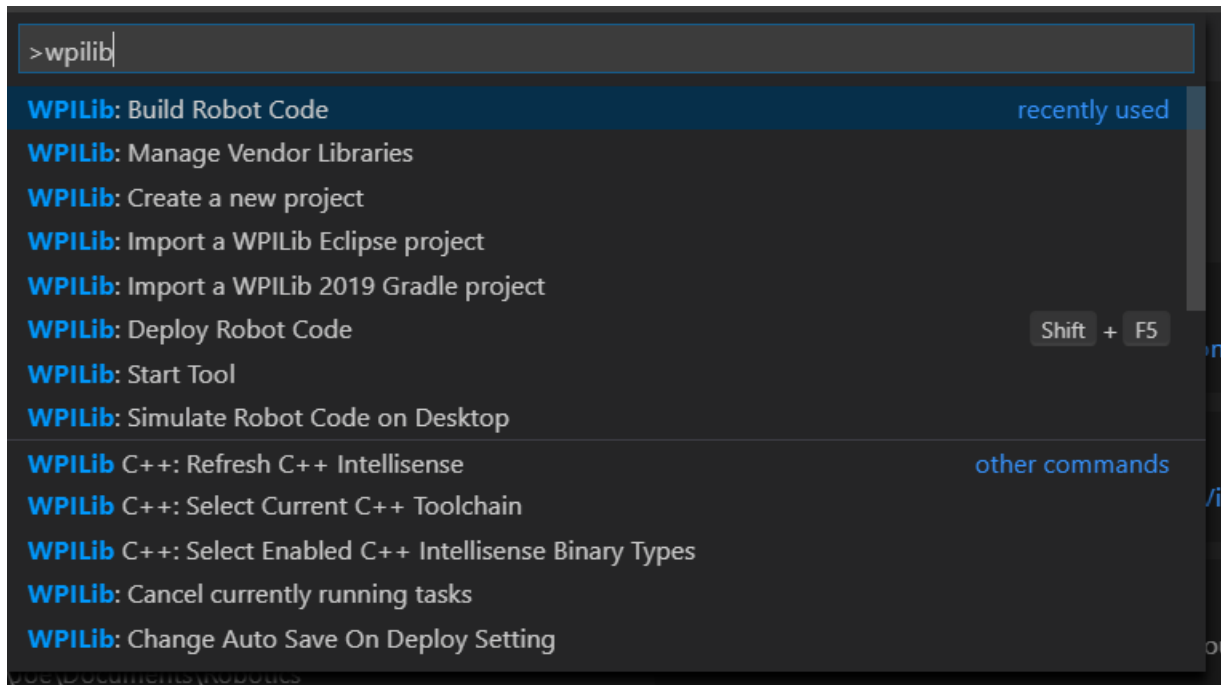
Once everything is installed, we're ready to create a robot program. WPILib comes with several templates for robot programs. Use of these templates is highly recommended for new users; however, advanced users are free to write their own robot code from scratch. This article walks through creating a project from one of the provided examples which has some code already written to drive a basic robot.

5.2.1 Creating a New WPILib Project

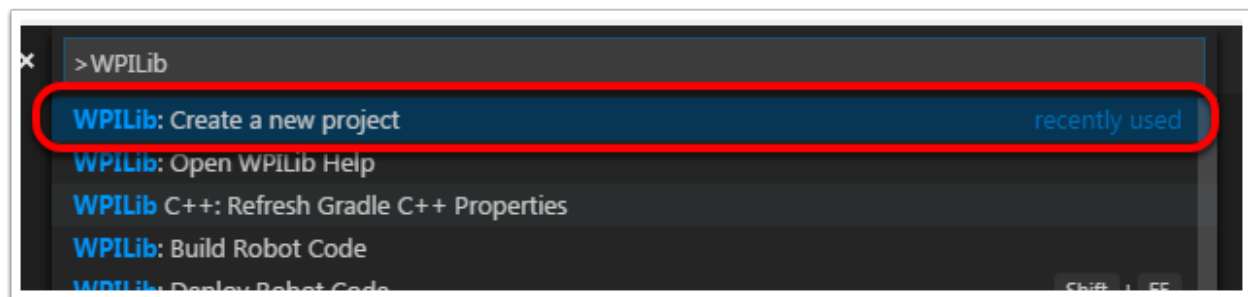
Bring up the Visual Studio Code command palette with `Ctrl+Shift+P`:



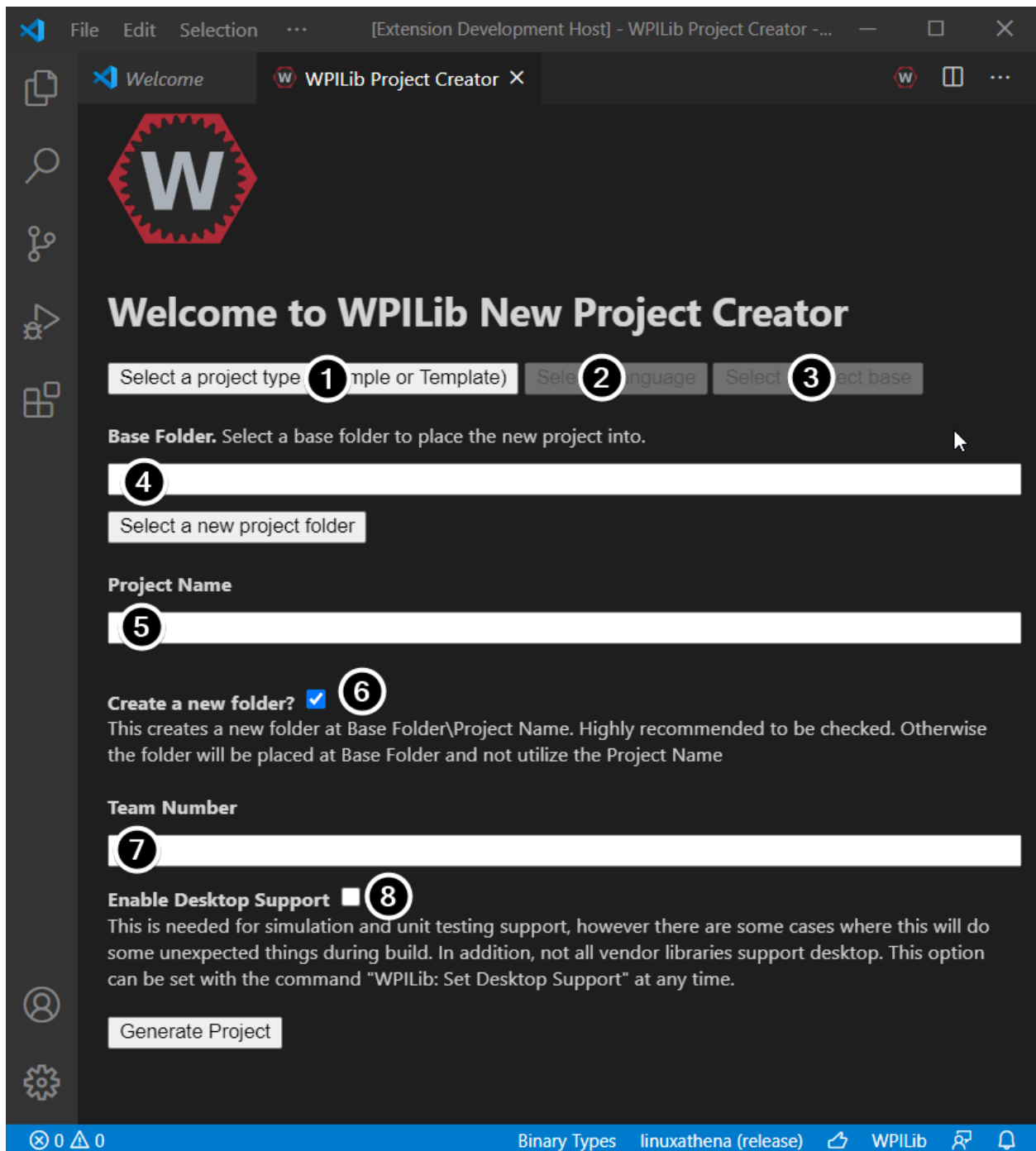
Then, type “WPILib” into the prompt. Since all WPILib commands start with “WPILib,” this will bring up the list of WPILib-specific VS Code commands:



Now, select the "Create a new project" command:



This will bring up the "New Project Creator Window:"



The elements of the New Project Creator Window are explained below:

1. **Project Type:** The kind of project we wish to create. For this example, select **Example**
2. **Language:** This is the language (C++ or Java) that will be used for this project.
3. **Project Base:** This box is used to select the base class or example to generate the project from. For this example, select **Getting Started**
4. **Base Folder:** This determines the folder in which the robot project will be located.
5. **Project Name:** The name of the robot project. This also specifies the name that the

6. **Create a New Folder:** If this is checked, a new folder will be created to hold the project within the previously-specified folder. If it is *not* checked, the project will be located directly in the previously-specified folder. An error will be thrown if the folder is not empty and this is not checked. project folder will be given if the Create New Folder box is checked.
7. **Team Number:** The team number for the project, which will be used for package names within the project and to locate the robot when deploying code.
8. **Enable Desktop Support:** Enables unit test and simulation. While WPILib supports this, third party software libraries may not. If libraries do not support desktop, then your code may not compile or may crash. It should be left unchecked unless unit testing or simulation is needed and all libraries support it. For this example, do not check this box.

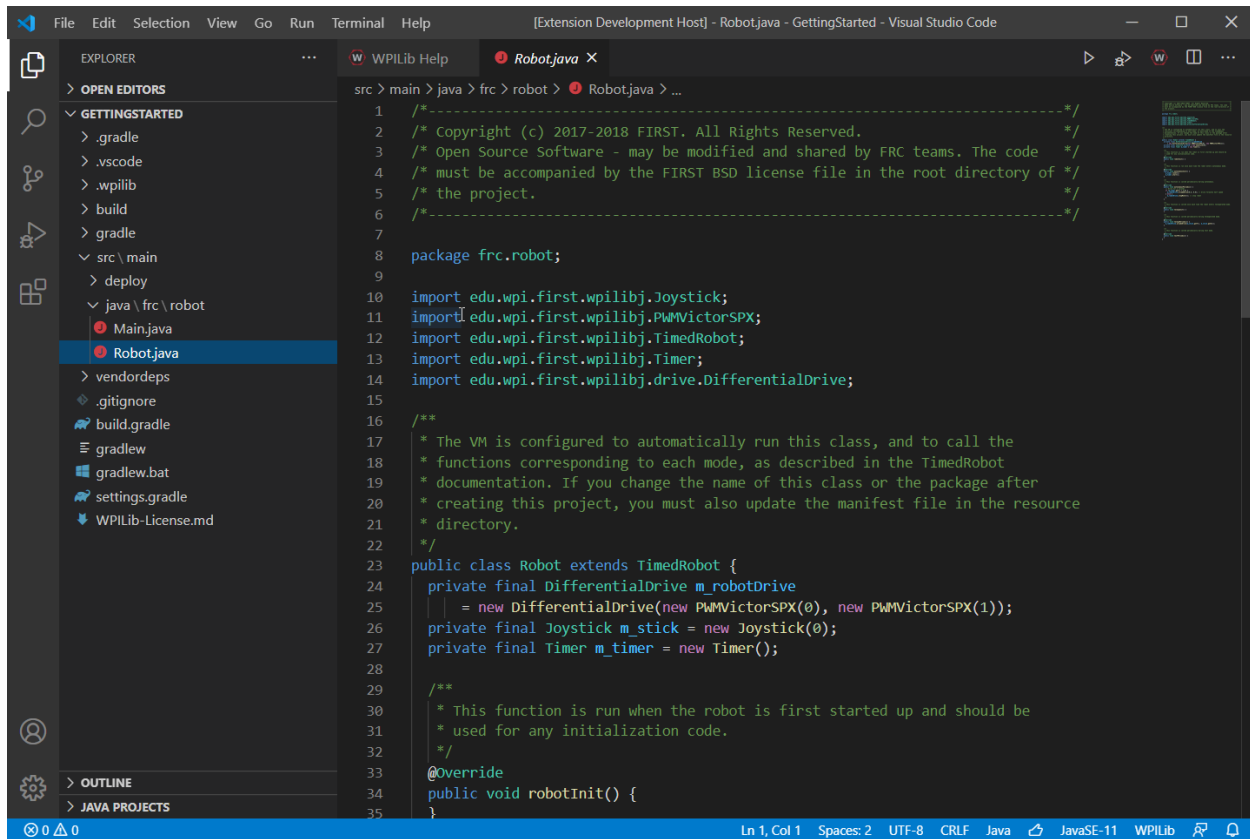
Once all the above have been configured, click “Generate Project” and the robot project will be created.

Note: Any errors in project generation will appear in the bottom right-hand corner of the screen.

5.2.2 Opening The New Project

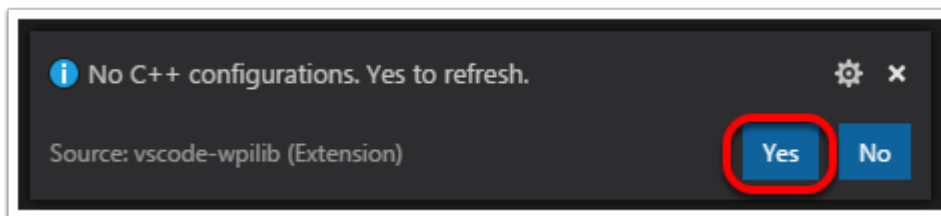
After successfully creating your project, VS Code will give the option of opening the project as shown below. We can choose to do that now or later by typing Ctrl+K then Ctrl+O (or just Command+O on macOS) and select the folder where we saved our project.

Once opened we will see the project hierarchy on the left. Double clicking on the file will open that file in the editor.



5.2.3 C++ Configurations (C++ Only)

For C++ projects, there is one more step to set up IntelliSense. Whenever we open a project, we should get a pop-up in the bottom right corner asking to refresh C++ configurations. Click “Yes” to set up IntelliSense.



5.2.4 Imports/Includes

Java

C++

```

7 import edu.wpi.first.wpilibj.Joystick;
8 import edu.wpi.first.wpilibj.PWMVictorSPX;
9 import edu.wpi.first.wpilibj.TimedRobot;
10 import edu.wpi.first.wpilibj.Timer;
11 import edu.wpi.first.wpilibj.drive.DifferentialDrive;

```



```

5 #include <frc/Joystick.h>
6 #include <frc/PWMVictorSPX.h>
7 #include <frc/TimedRobot.h>
8 #include <frc/Timer.h>
9 #include <frc/drive/DifferentialDrive.h>
10 #include <frc/livewindow/LiveWindow.h>

```

Our code needs to reference the components of WPILib that are used. In C++ this is accomplished using `#include` statements; in Java it is done with `import` statements. The program references classes for Joystick (for driving), PWMVictorSPX (for controlling motors), TimedRobot (the base class used for the example), Timer (used for autonomous), DifferentialDrive (for connecting the joystick control to the motors), and LiveWindow (C++ only).

5.2.5 Defining the variables for our sample robot

Java

C++

```

19 public class Robot extends TimedRobot {
20     private final DifferentialDrive m_robotDrive =
21         new DifferentialDrive(new PWMVictorSPX(0), new PWMVictorSPX(1));
22     private final Joystick m_stick = new Joystick(0);
23     private final Timer m_timer = new Timer();

```

```

12 class Robot : public frc::TimedRobot {
13     public:
14         Robot() {
15             m_robotDrive.SetExpiration(0.1);
16             m_timer.Start();
17         }

```

```

46 private:
47     // Robot drive system
48     frc::PWMVictorSPX m_left{0};
49     frc::PWMVictorSPX m_right{1};
50     frc::DifferentialDrive m_robotDrive{m_left, m_right};
51
52     frc::Joystick m_stick{0};
53     frc::LiveWindow& m_lw = *frc::LiveWindow::GetInstance();
54     frc::Timer m_timer;

```

The sample robot in our examples will have a joystick on USB port 0 for arcade drive and two motors on PWM ports 0 and 1. Here we create objects of type `DifferentialDrive` (`m_robotDrive`), `Joystick` (`m_stick`) and time (`m_timer`). This section of the code does three things:

1. Defines the variables as members of our Robot class.
2. Initializes the variables.

Note: The variable initializations for C++ are in the `private` section at the bottom of the program. This means they are private to the class (`Robot`). The C++ code also sets the Motor

Safety expiration to 0.1 seconds (the drive will shut off if we don't give it a command every .1 seconds) and starts the Timer used for autonomous.

5.2.6 Robot Initialization

Java

C++

```
@Override
public void robotInit() {}
```

```
void RobotInit() {}
```

The RobotInit method is run when the robot program is starting up, but after the constructor. The RobotInit for our sample program doesn't do anything. If we wanted to run something here we could provide the code above to override the default).

5.2.7 Simple Autonomous Example

Java

C++

```
32  /** This function is run once each time the robot enters autonomous mode. */
33  @Override
34  public void autonomousInit() {
35      m_timer.reset();
36      m_timer.start();
37  }
38
39  /** This function is called periodically during autonomous. */
40  @Override
41  public void autonomousPeriodic() {
42      // Drive for 2 seconds
43      if (m_timer.get() < 2.0) {
44          m_robotDrive.arcadeDrive(0.5, 0.0); // drive forwards half speed
45      } else {
46          m_robotDrive.stopMotor(); // stop robot
47      }
48  }
```

```
19  void AutonomousInit() override {
20      m_timer.Reset();
21      m_timer.Start();
22  }
23
24  void AutonomousPeriodic() override {
25      // Drive for 2 seconds
26      if (m_timer.Get() < 2.0) {
27          // Drive forwards half speed
28          m_robotDrive.ArcadeDrive(-0.5, 0.0);
29      } else {
```

(continues on next page)

(continued from previous page)

```

30     // Stop robot
31     m_robotDrive.ArcadeDrive(0.0, 0.0);
32 }
33 }

```

The AutonomousInit method is run once each time the robot transitions to autonomous from another mode. In this program, we reset the Timer and then start it in this method.

AutonomousPeriodic is run once every period while the robot is in autonomous mode. In the TimedRobot class the period is a fixed time, which defaults to 20ms. In this example, the periodic code checks if the timer is less than 2 seconds and if so, drives forward at half speed using the ArcadeDrive method of the DifferentialDrive class. If more than 2 seconds has elapsed, the code stops the robot drive.

5.2.8 Joystick Control for teleoperation

Java

C++

```

50  /** This function is called once each time the robot enters teleoperated mode. */
51  @Override
52  public void teleopInit() {}
53
54  /** This function is called periodically during teleoperated mode. */
55  @Override
56  public void teleopPeriodic() {
57      m_robotDrive.arcadeDrive(m_stick.getY(), m_stick.getX());
58  }

```

```

35  void TeleopInit() override {}
36
37  void TeleopPeriodic() override {
38      // Drive with arcade style (use right stick)
39      m_robotDrive.ArcadeDrive(m_stick.GetY(), m_stick.GetX());
40  }

```

Like in Autonomous, the Teleop mode has a TeleopInit and TeleopPeriodic function. In this example we don't have anything to do in TeleopInit, it is provided for illustration purposes only. In TeleopPeriodic, the code uses the ArcadeDrive method to map the Y-axis of the Joystick to forward/back motion of the drive motors and the X-axis to turning motion.

5.2.9 Test Mode

Java

C++

```

60  /** This function is called once each time the robot enters test mode. */
61  @Override
62  public void testInit() {}
63
64  /** This function is called periodically during test mode. */

```

(continues on next page)

(continued from previous page)

```
65  @Override
66  public void testPeriodic() {}
67  }
```

```
42  void TestInit() override {}
43
44  void TestPeriodic() override {}
```

Test Mode is used for testing robot functionality. Similar to TeleopInit, the TestInit and TestPeriodic methods are provided here for illustrative purposes only.

5.2.10 Deploying the Project to a Robot

Please see the instructions [here](#) for deploying the program onto a robot.

5.3 Running your Benchtop Test Program

5.3.1 Overview

You should create and download a Benchtop Test Program as described for your programming language:

C++/Java

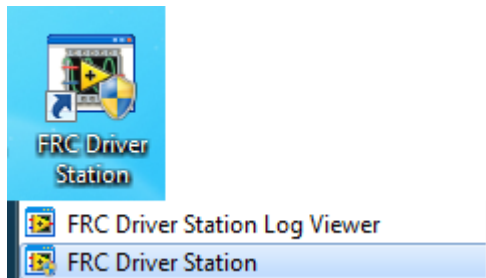
LabVIEW

5.3.2 Tethered Operation

Running your benchtop testing program while tethered to the Driver Station via ethernet or USB cable will confirm the the program was successfully deployed and that the driver station and roboRIO are properly configured.

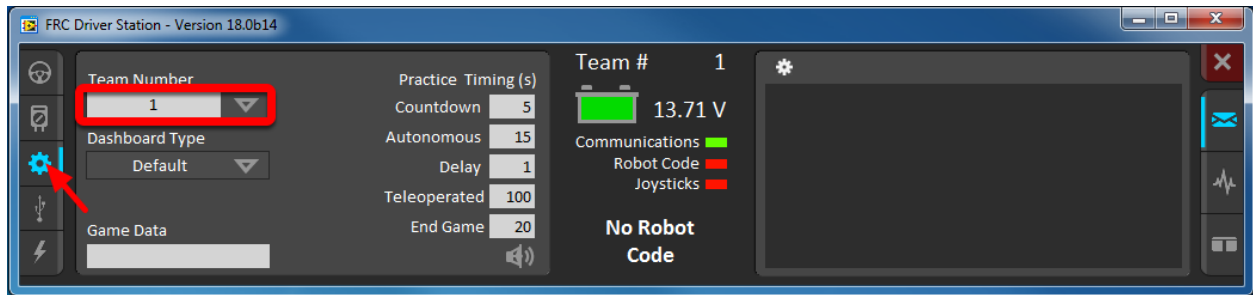
The roboRIO should be powered on and connected to the PC over Ethernet or USB.

5.3.3 Starting the FRC Driver Station



The FRC® Driver Station can be launched by double-clicking the icon on the Desktop or by selecting Start->All Programs->FRC Driver Station.

5.3.4 Setting Up the Driver Station



The DS must be set to your team number in order to connect to your robot. In order to do this click the Setup tab then enter your team number in the team number box. Press return or click outside the box for the setting to take effect.

PCs will typically have the correct network settings for the DS to connect to the robot already, but if not, make sure your Network adapter is set to DHCP.

5.3.5 Confirm Connectivity

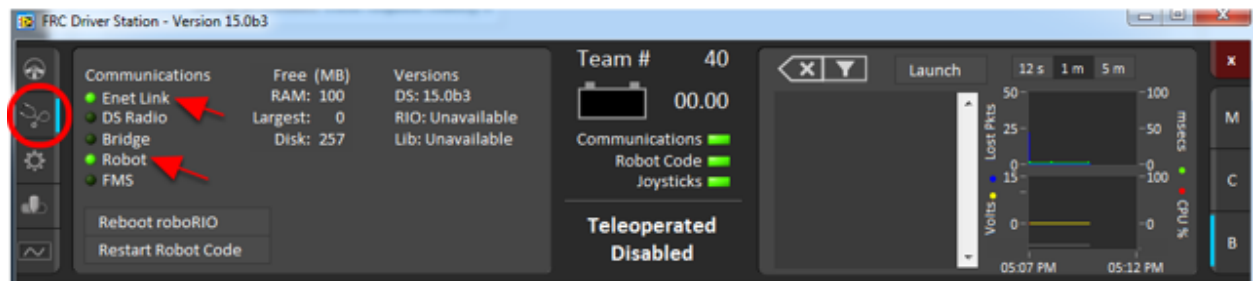


Fig. 1: Tethered

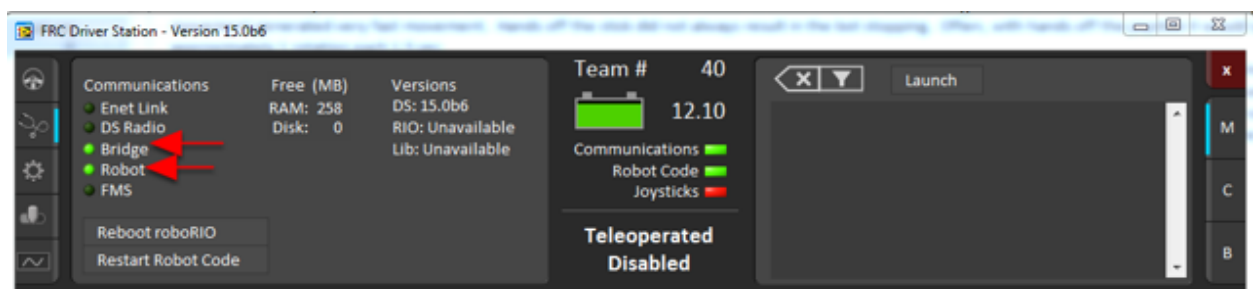
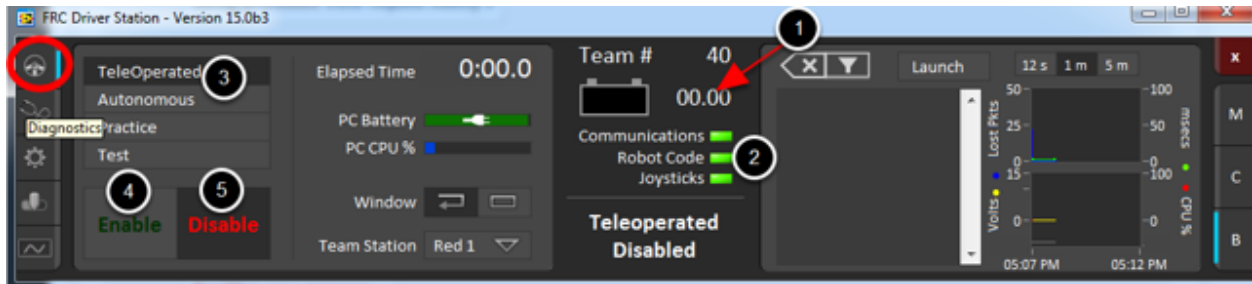


Fig. 2: Wireless

Using the Driver Station software, click Diagnostics and confirm that the Enet Link (or Robot Radio led, if operating wirelessly) and Robot leds are green.

5.3.6 Operate the Robot



Click the Operation Tab

1. Confirm that battery voltage is displayed
2. Communications, Robot Code, and Joysticks indicators are green.
3. Put the robot in Teleop Mode
4. Click Enable. Move the joysticks and observe how the robot responds.
5. Click Disable

5.3.7 Wireless Operation

Before attempting wireless operation, tethered operation should have been confirmed as described in [Tethered Operation](#). Running your benchtop testing program while connected to the Driver Station via WiFi will confirm that the access point is properly configured.

Configuring the Access Point

See the article [Programming your radio](#) for details on configuring the robot radio for use as an access point.

After configuring the access point, connect the driver station wirelessly to the robot. The SSID will be your team number (as entered in the Bridge Configuration Utility). If you set a key when using the Bridge Configuration Utility you will need to enter it to connect to the network. Make sure the computer network adapter is set to DHCP ("Obtain an IP address automatically").

You can now confirm wireless operation using the same steps in **Confirm Connectivity** and **Operate the Robot** above.

Hardware Component Overview

The goal of this document is to provide a brief overview of the hardware components that make up the FRC® Control System. Each component will contain a brief description of the component function and a link to more documentation.

Note: For complete wiring instructions/diagrams, please see the [Wiring the FRC Control System](#) document.

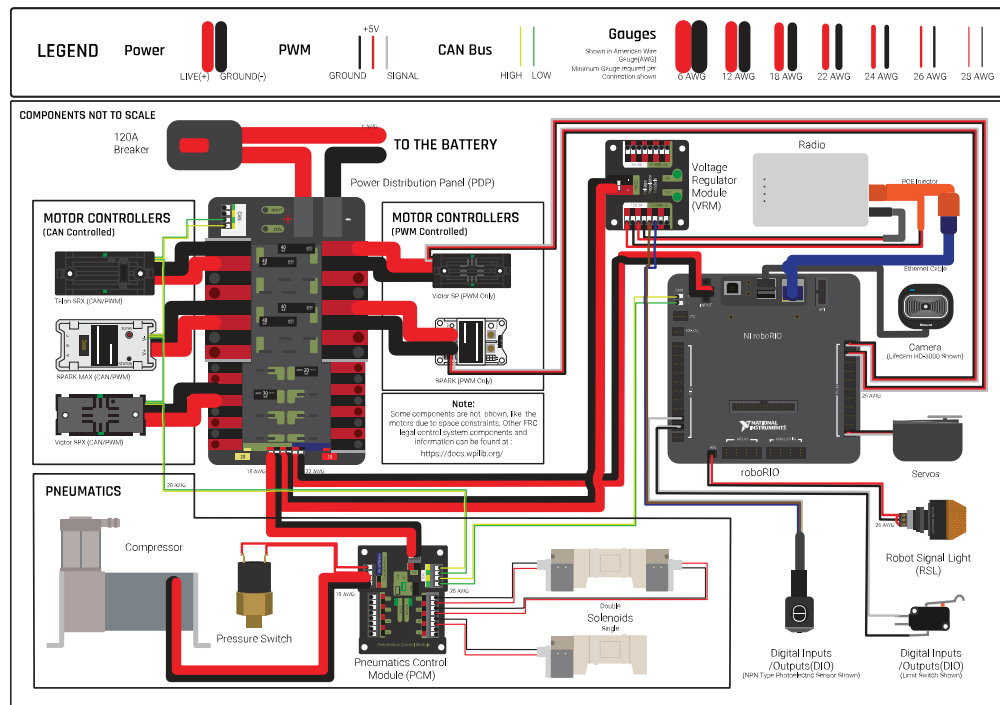
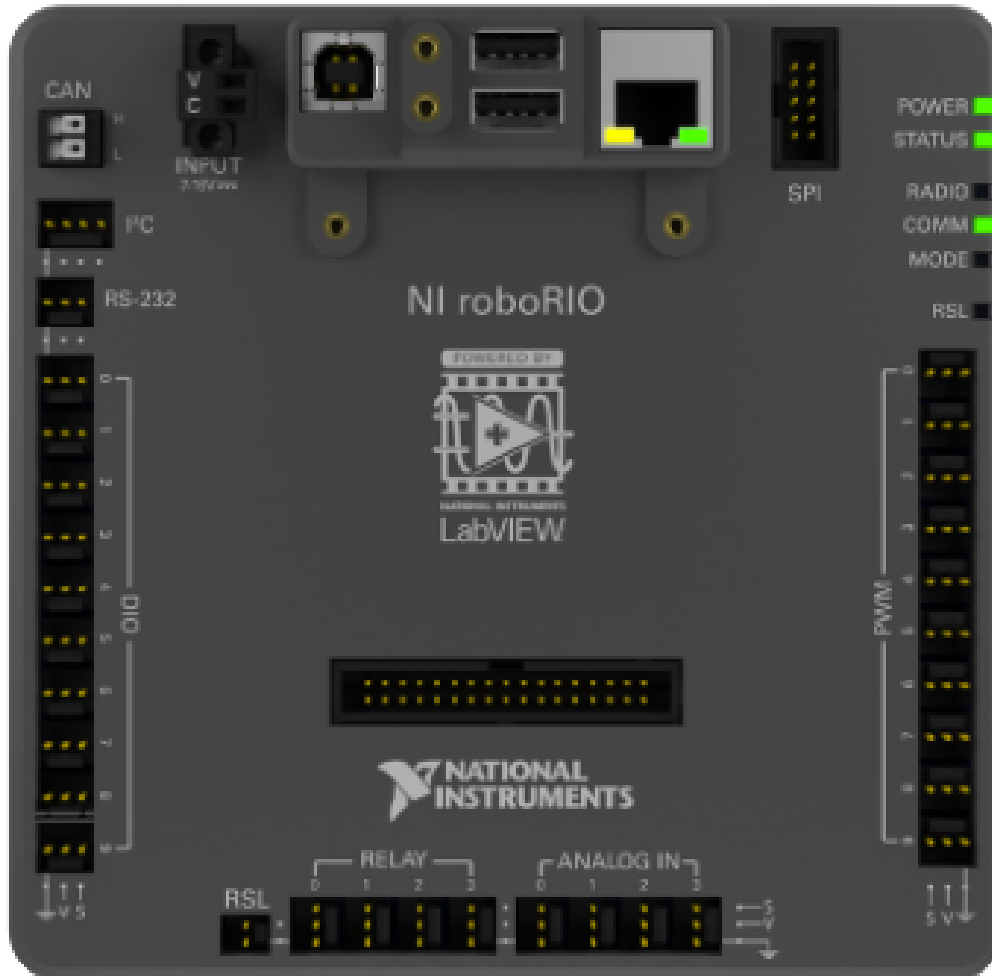


Fig. 1: Diagram courtesy of FRC Team 3161 and Stefen Acepcion.

6.1 Overview of Control System

6.2 NI roboRIO



The *NI-roboRIO* is the main robot controller used for FRC. The roboRIO serves as the “brain” for the robot running team-generated code that commands all of the other hardware.

6.3 Power Distribution Panel



The *Power Distribution Panel* (PDP) is designed to distribute power from a 12VDC battery to various robot components through auto-resetting circuit breakers and a small number of special function fused connections. The PDP provides 8 output pairs rated for 40A continuous current and 8 pairs rated for 30A continuous current. The PDP provides dedicated 12V connectors for the roboRIO, as well as connectors for the Voltage Regulator Module and Pneumatics Control Module. It also includes a CAN interface for logging current, temperature, and battery voltage. For more detailed information, see the [PDP User Manual](#).

6.4 Voltage Regulator Module



The Voltage Regulator Module (VRM) is an independent module that is powered by 12 volts. The device is wired to a dedicated connector on the PDP. The module has multiple regulated 12V and 5V outputs. The purpose of the VRM is to provide regulated power for the robot radio, custom circuits, and IP vision cameras. For more information, see the [VRM User Manual](#).

6.5 OpenMesh OM5P-AN or OM5P-AC Radio



Either the OpenMesh OM5P-AN or [OpenMesh OM5P-AC](#) wireless radio is used as the robot radio to provide wireless communication functionality to the robot. The device can be configured as an Access Point for direct connection of a laptop for use at home. It can also be configured as a bridge for use on the field. The robot radio should be powered by one of the 12V/2A outputs on the VRM and connected to the roboRIO controller over Ethernet. For more information, see [Programming your Radio](#).

The OM5P-AN is [no longer available for purchase](#). The OM5P-AC is slightly heavier, has more cooling grates, and has a rough surface texture compared to the OM5P-AN.

6.6 120A Circuit Breaker



The 120A Main Circuit Breaker serves two roles on the robot: the main robot power switch and a protection device for downstream robot wiring and components. The 120A circuit breaker is wired to the positive terminals of the robot battery and Power Distribution boards. For more information, please see the [Cooper Bussmann 18X Series Datasheet \(PN: 185120F\)](#)

6.7 Snap Action Circuit Breakers



The Snap Action circuit breakers, [MX5 series](#) and [VB3 Series](#), are used with the Power Distribution Panel to limit current to branch circuits. The ratings on these circuit breakers are for continuous current, temporary peak values can be considerably higher.

6.8 Robot Battery



The power supply for an FRC robot is a single 12V 18Ah Sealed Lead Acid (SLA) battery, capable of meeting the high current demands of an FRC robot. For more information, see the [Robot Battery page](#).

Note: Multiple battery part numbers may be legal, consult the [FRC Manual](#) for a complete list.

6.9 Robot Signal Light



The Robot Signal Light (RSL) is required to be the Allen-Bradley 855PB-B12ME522. It is directly controlled by the roboRIO and will flash when enabled and stay solid while disabled.

6.10 Pneumatics Control Module



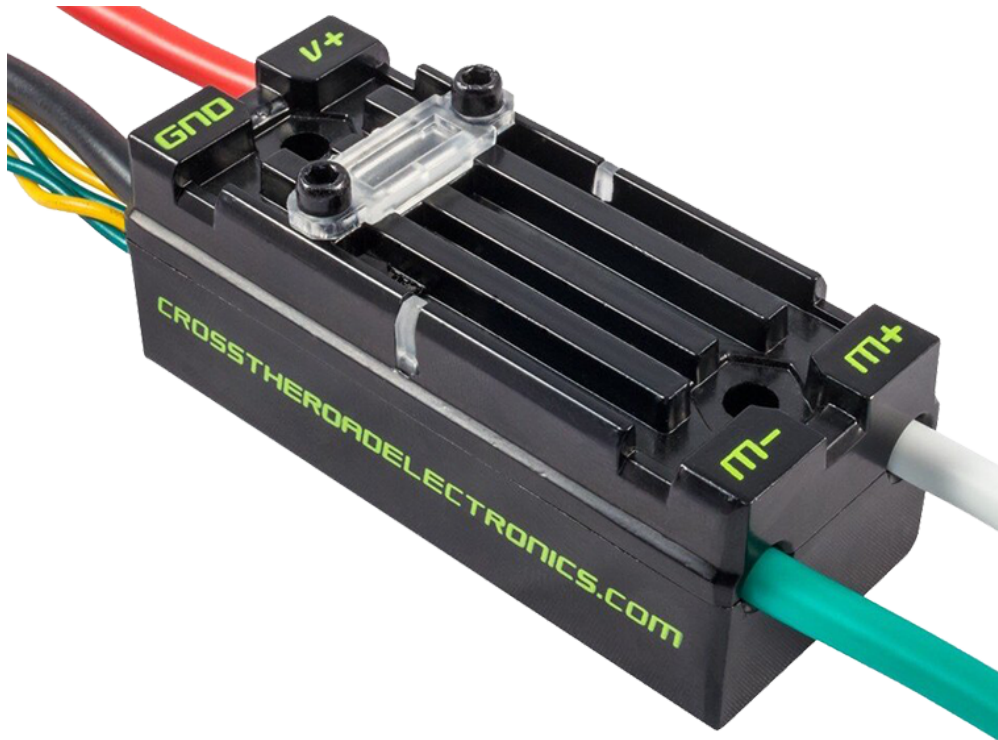
The *Pneumatics Control Module* (PCM) contains all of the inputs and outputs required to operate 12V or 24V pneumatic solenoids and the on board compressor. The PCM contains an input for the pressure sensor and will control the compressor automatically when the robot is enabled and a solenoid has been created in the code. For more information see the [PCM User Manual](#).

6.11 Motor Controllers

There are a variety of different *motor controllers* which work with the FRC Control System and are approved for use. These devices are used to provide variable voltage control of the brushed and brushless DC motors used in FRC. They are listed here in order of [usage](#).

Note: 3rd Party CAN control is not supported from WPILib. See this section on [Third-Party CAN Devices](#) for more information.

6.11.1 Talon SRX



The [Talon SRX Motor Controller](#) is a “smart motor controller” from Cross The Road Electronics/VEX Robotics. The Talon SRX can be controlled over the CAN bus or PWM interface. When using the CAN bus control, this device can take inputs from limit switches and potentiometers, encoders, or similar sensors in order to perform advanced control. For more information see the [Talon SRX User’s Guide](#).

6.11.2 Victor SPX



The [Victor SPX Motor Controller](#) is a CAN or PWM controlled motor controller from Cross The Road Electronics/VEX Robotics. The device is connectorized to allow easy connection to the

roboRIO PWM connectors or a CAN bus. The case is sealed to prevent debris from entering the controller. For more information, see the [Victor SPX User Guide](#).

6.11.3 SPARK MAX Motor Controller



The [SPARK MAX Motor Controller](#) is an advanced brushed and brushless DC motor controller from REV Robotics. When using CAN bus or USB control, the SPARK MAX uses input from limit switches, encoders, and other sensors, including the integrated encoder of the REV NEO Brushless Motor, to perform advanced control modes. The SPARK MAX can be controlled over PWM, CAN or USB (for configuration/testing only). For more information, see the [SPARK MAX User's Manual](#).

6.11.4 TalonFX Motor Controller



The [TalonFX Motor Controller](#) is integrated into the Falcon 500 brushless motor. It features an integrated encoder and all of the smart features of the Talon SRX and more! For more information see the [Falcon 500 User Guide](#).

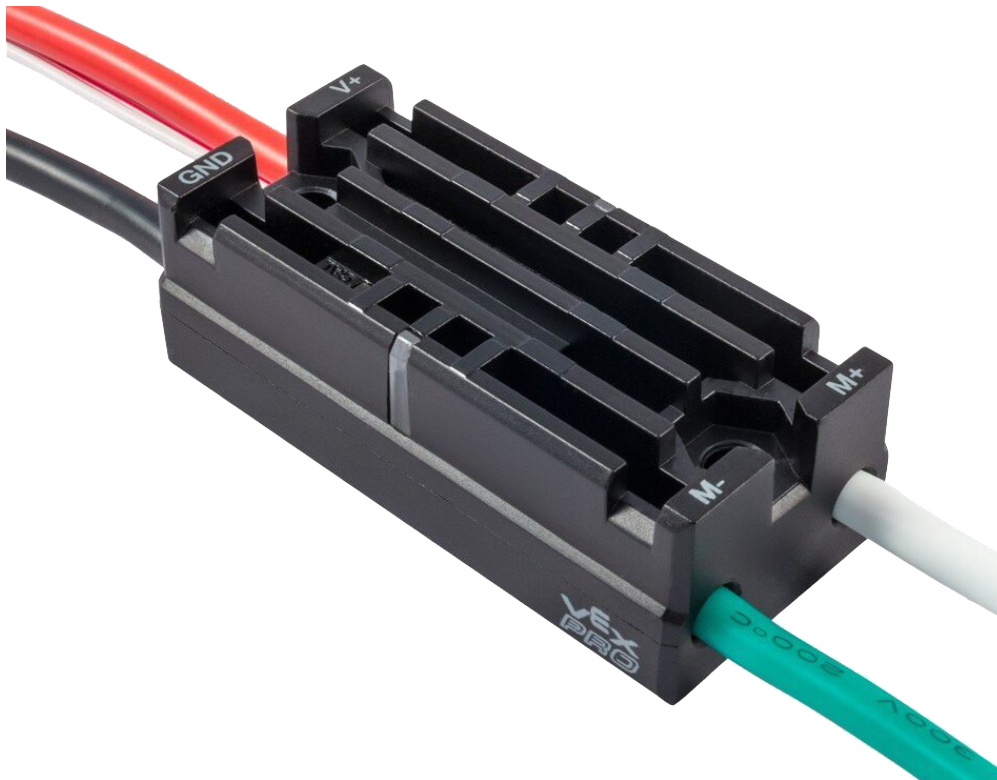
6.11.5 SPARK Motor Controller



Warning: While this motor controller is still legal for FRC use, the manufacturer has discontinued this product.

The [SPARK Motor Controller](#) from REV Robotics is an inexpensive brushed DC motor controller. The SPARK is controlled using the PWM interface. Limit switches may be wired directly to the SPARK to limit motor travel in one or both directions. For more information, see the [SPARK User's Manual](#).

6.11.6 Victor SP



Warning: While this motor controller is still legal for FRC use, the manufacturer has discontinued this product.

The [Victor SP Motor Controller](#) is a PWM motor controller from Cross The Road Electronics/VEX Robotics. The Victor SP has an electrically isolated metal housing for heat dissipation, making the use of the fan optional. The case is sealed to prevent debris from entering the controller. The controller is approximately half the size of previous models.

6.11.7 Talon Motor Controller



Warning: While this motor controller is still legal for FRC use, the manufacturer has discontinued this product.

The [Talon Motor Controller](#) from Cross the Road Electronics is a PWM controlled brushed DC motor controller with passive cooling.

6.11.8 Victor 888 Motor Controller / Victor 884 Motor Controller



Warning: While this motor controller is still legal for FRC use, the manufacturer has discontinued this product.

The [Victor 884](#) and [Victor 888](#) motor controllers from VEX Robotics are variable speed PWM motor controllers for use in FRC. The Victor 888 replaces the Victor 884, which is also usable in FRC.

6.11.9 Jaguar Motor Controller



Warning: While this motor controller is still legal for FRC use, the manufacturer has discontinued this product.

The [Jaguar Motor Controller](#) from VEX Robotics (formerly made by Luminary Micro and Texas Instruments) is a variable speed motor controller for use in FRC. For FRC, the Jaguar may only be controlled using the PWM interface.

6.11.10 DMC-60 and DMC-60C Motor Controller



Warning: While this motor controller is still legal for FRC use, the manufacturer has discontinued this product.

The DMC-60 is a PWM motor controller from Digilent. The DMC-60 features integrated thermal sensing and protection including current-foldback to prevent overheating and damage, and four multi-color LEDs to indicate speed, direction, and status for easier debugging. For more information, see the [DMC-60 reference manual](#)

The DMC-60C adds CAN smart controller capabilities to the DMC-60 controller. Due to the manufacturer discontinuing this product, the DMC-60C is only usable with PWM. For more information see the [DMC-60C Product Page](#)

6.11.11 Venom Motor Controller



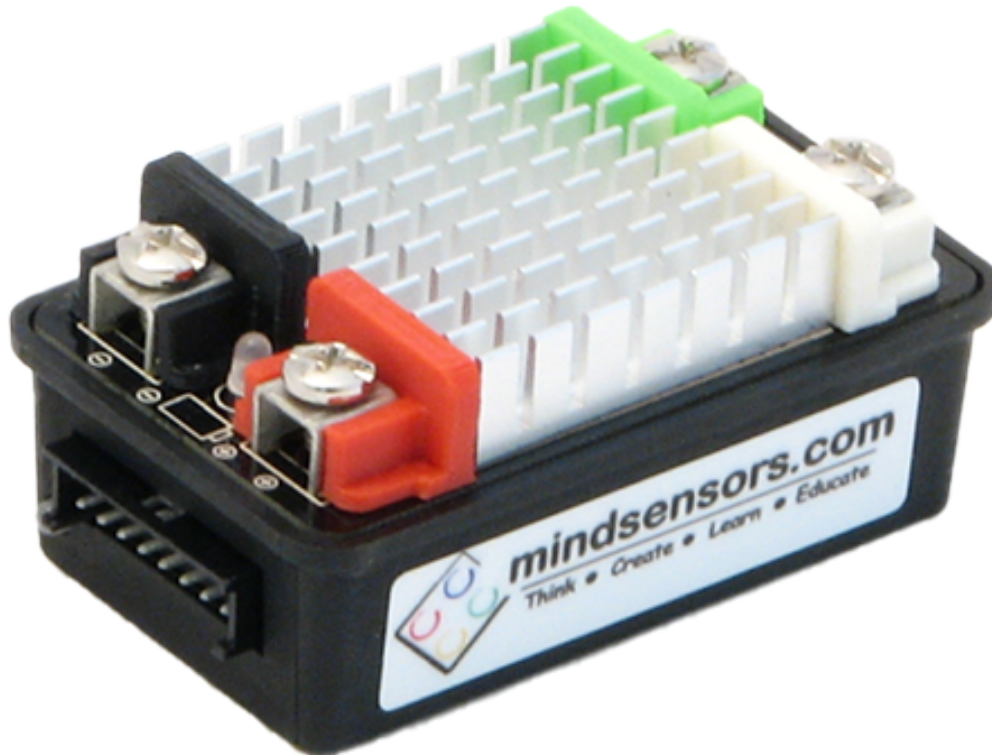
The [Venom Motor Controller](#) from Playing With Fusion is integrated into a motor based on the original CIM. Speed, current, temperature, and position are all measured onboard, enabling advanced control modes without complicated sensing and wiring schemes.

6.11.12 Nidec Dynamo BLDC Motor with Controller



The [Nidec Dynamo BLDC Motor with Controller](#) is the first brushless motor and controller legal in FRC. This motor's controller is integrated into the back of the motor. The [motor data sheet](#) provides more device specifics.

6.11.13 SD540B and SD540C Motor Controllers



The SD540B and SD540C Motor Controllers from Mindsensors are controlled using PWM. CAN control is no longer available for the SD540C due to lack of manufacturer support. Limit switches may be wired directly to the SD540 to limit motor travel in one or both directions. For more information see the [Mindsensors FRC page](#)

6.12 Spike H-Bridge Relay



Warning: While this relay is still legal for FRC use, the manufacturer has discontinued this product.

The Spike H-Bridge Relay from VEX Robotics is a device used for controlling power to motors or other custom robot electronics. When connected to a motor, the Spike provides On/Off control in both the forward and reverse directions. The Spike outputs are independently controlled so it can also be used to provide power to up to 2 custom electronic circuits. The Spike H-Bridge Relay should be connected to a relay output of the roboRIO and powered from the Power Distribution Panel. For more information, see the [Spike User's Guide](#).

6.13 Servo Power Module



The Servo Power Module from Rev Robotics is capable of expanding the power available to servos beyond what the roboRIO integrated power supply is capable of. The Servo Power Module provides up to 90W of 6V power across 6 channels. All control signals are passed through directly from the roboRIO. For more information, see the [Servo Power Module web-page](#).

6.14 Axis M1013/M1011/206 Ethernet Camera



Warning: While this camera is legal for FRC use, the manufacturer has discontinued support.

The Axis M1013, M1011 and Axis 206 Ethernet cameras are used for capturing images for vision processing and/or sending video back to the Driver Station laptop. The camera should be wired to a 5V power output on the Voltage Regulator Module and an open ethernet port on the robot radio. For more information, see [Configuring an Axis Camera](#) and the [Axis 206](#), [Axis M1011](#), [Axis M1013](#) pages.

6.15 Microsoft Lifecam HD3000



The Microsoft Lifecam HD3000 is a USB webcam that can be plugged directly into the roboRIO. The camera is capable of capturing up to 1280x720 video at 30 FPS. For more information about the camera, see the [Microsoft product page](#). For more information about using the camera with the roboRIO, see the [Vision Processing](#) section of this documentation.

6.16 Image Credits

Image of roboRIO courtesy of National Instruments. Image of DMC-60 courtesy of Digi-lent. Image of SD540 courtesy of Mindsensors. Images of Jaguar Motor Controller, Talon SRX, Talon FX, Victor 888, Victor SP, Victor SPX, and Spike H-Bridge Relay courtesy of VEX Robotics, Inc. Image of SPARK MAX courtesy of REV Robotics. Lifecam, PDP, PCM, SPARK, and VRM photos courtesy of FIRST®. All other photos courtesy of AndyMark Inc.

Software Component Overview

The FRC® software consists of a wide variety of mandatory and optional components. These elements are designed to assist you in the design, development, and debugging of your robot code as well as assist with control robot operation and to provide feedback when troubleshooting. For each software component this document will provide a brief overview of its purpose, a link to the package download, if appropriate, and a link to further documentation where available.

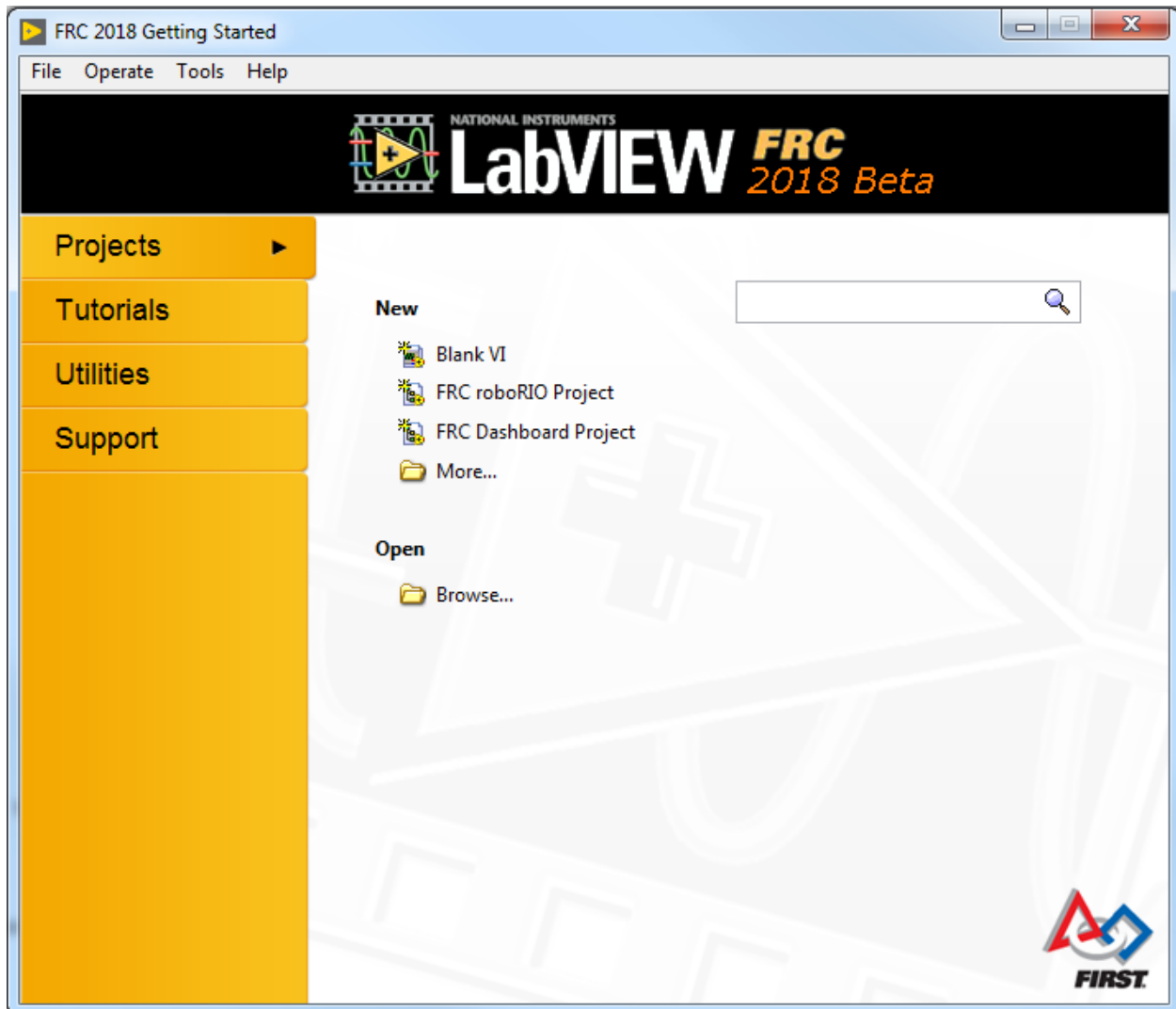
7.1 Operating System Compatibility

The primary supported OS for FRC components is Windows. All required FRC software components have been tested on Windows 7, 8, and 10. Windows XP is not supported.

Many of the tools for C++/Java programming are also supported and tested on macOS and Linux. Teams programming in C++/Java should be able to develop using these systems, using a Windows system for the Windows-only operations such as the Driver Station, Radio Configuration Utility, and roboRIO Imaging Tool.

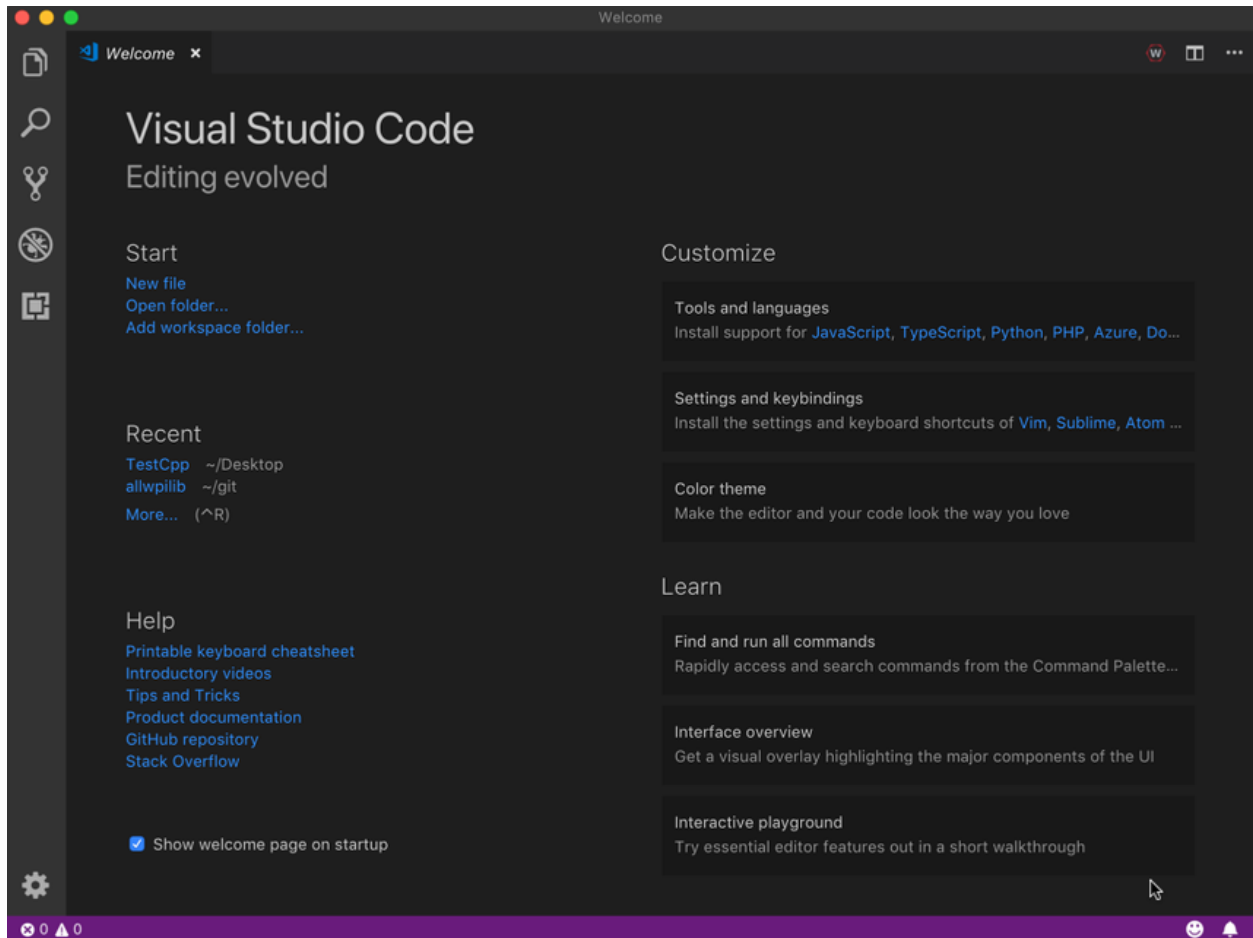
Important: The minimum supported macOS version is Mojave (10.14.x).

7.2 LabVIEW FRC (Windows Only)



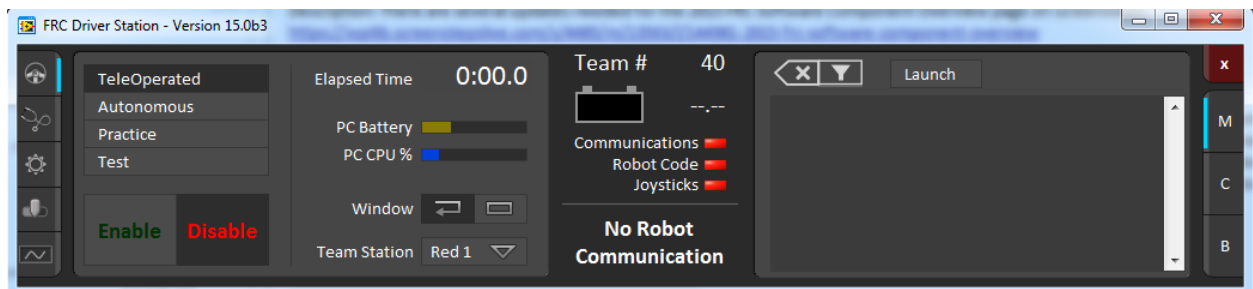
LabVIEW FRC, based on a recent version of LabVIEW Professional, is one of the three officially supported languages for programming an FRC robot. LabVIEW is a graphical, dataflow-driven language. LabVIEW programs consist of a collection of icons, called VIs, wired together with wires which pass data between the VIs. The LabVIEW FRC installer is distributed on a DVD found in the Kickoff Kit of Parts and is also available for download. A guide to getting started with the LabVIEW FRC software, including installation instructions can be found [here](#).

7.3 Visual Studio Code



Visual Studio Code is the supported development environment for C++ and Java (the other two supported languages). Both are object-oriented text based programming languages. A guide to getting started with C++ or Java for FRC, including the installation and configuration of Visual Studio Code can be found [here](#).

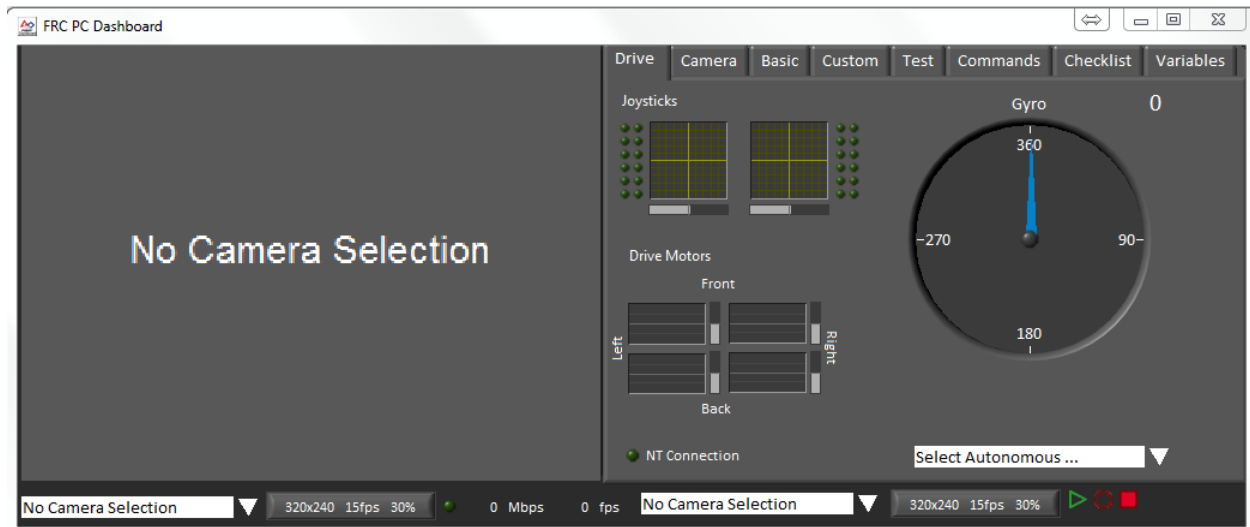
7.4 FRC Driver Station Powered by NI LabVIEW (Windows Only)



This is the only software allowed to be used for the purpose of controlling the state of the robot during competition. This software sends data to your robot from a variety of input devices. It also contains a number of tools used to help troubleshoot robot issues. More information about the FRC Driver Station Powered by NI LabVIEW can be found [here](#).

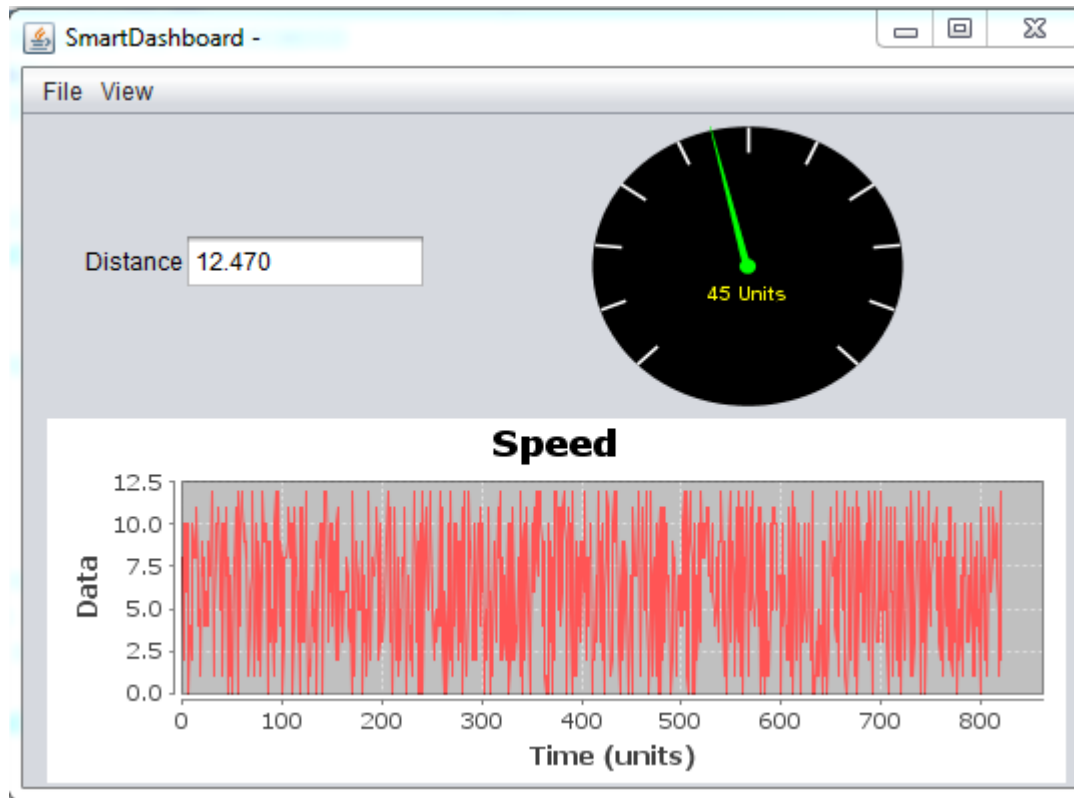
7.5 Dashboard Options

7.5.1 LabVIEW Dashboard (Windows Only)



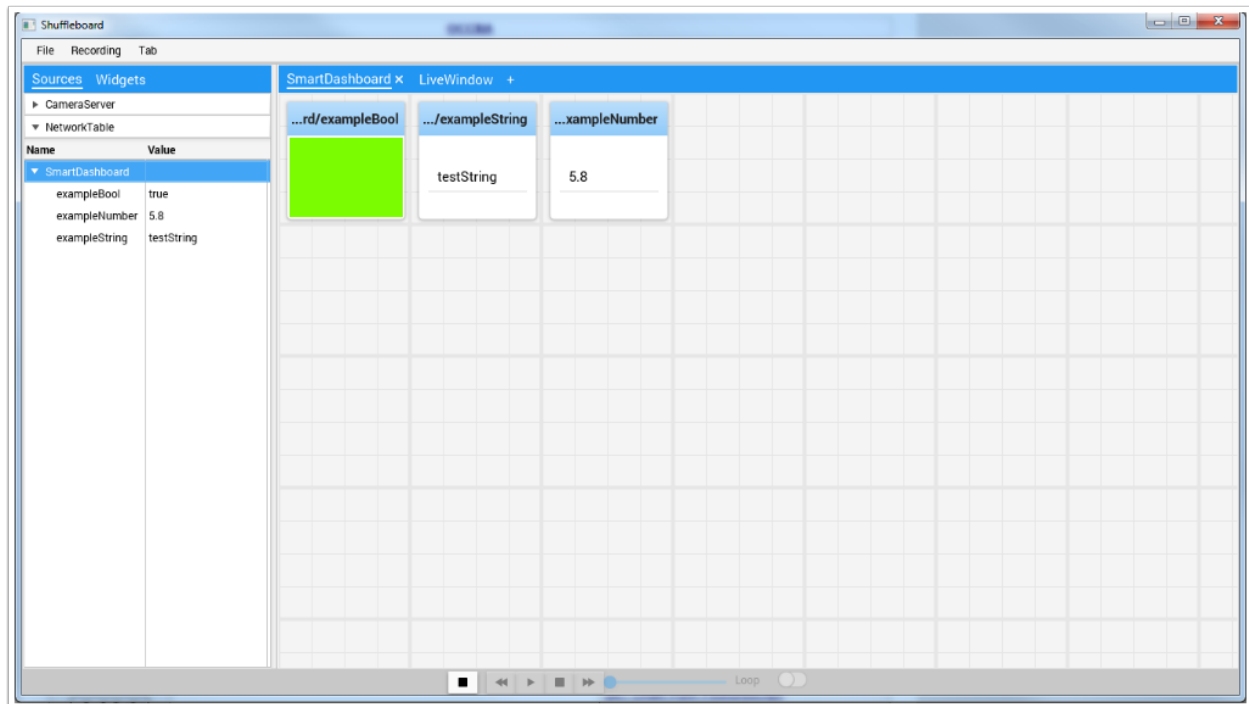
The LabVIEW Dashboard is automatically launched by the FRC Driver Station by default. The purpose of the Dashboard is to provide feedback about the operation of the robot using tabbed display with a variety of built in features. More information about the FRC Default Dashboard software can be found [here](#).

7.5.2 SmartDashboard



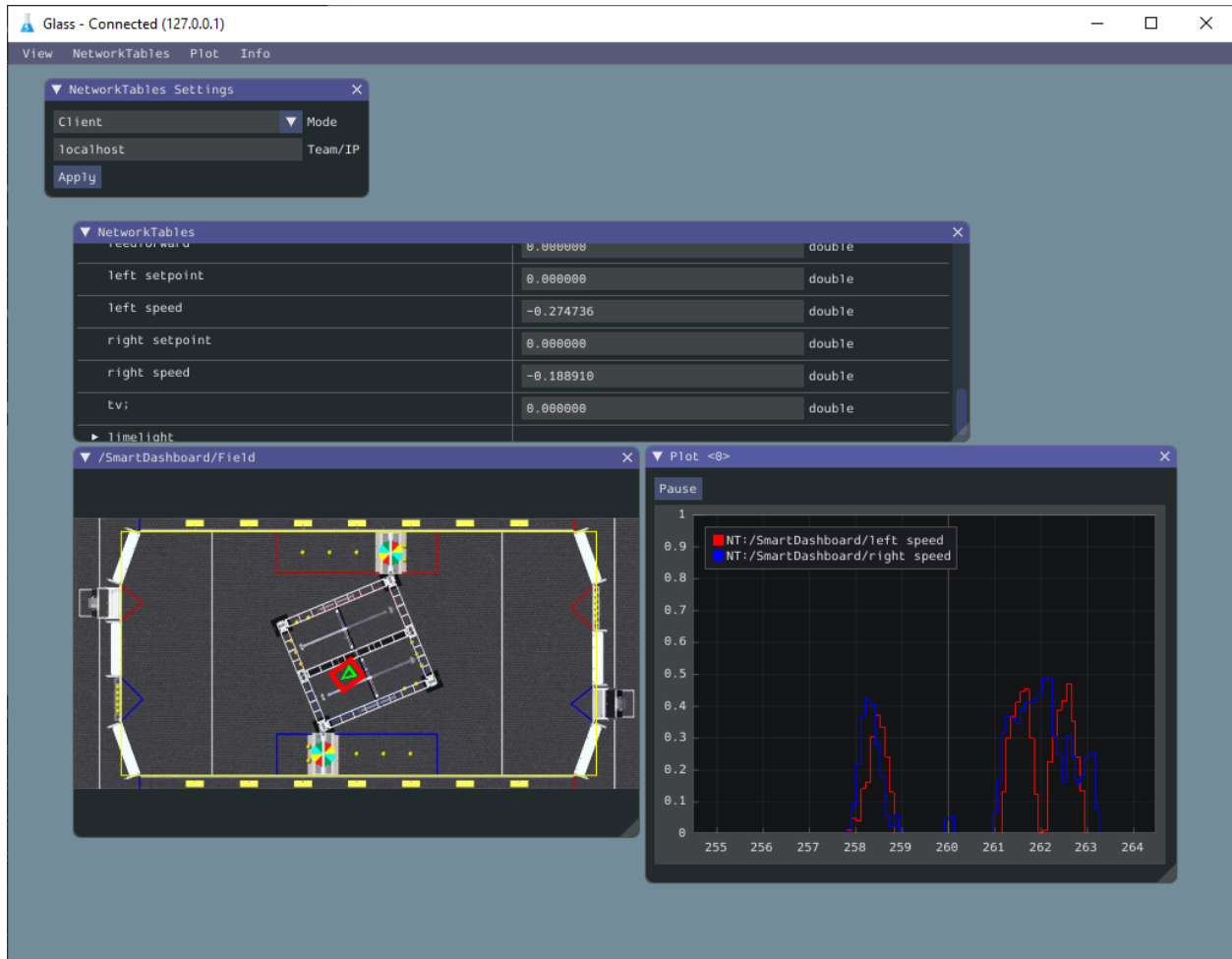
SmartDashboard allows you to view your robot data by automatically creating customizable indicators specifically for each piece of data sent from your robot. Additional documentation on SmartDashboard can be found [here](#).

7.5.3 Shuffleboard



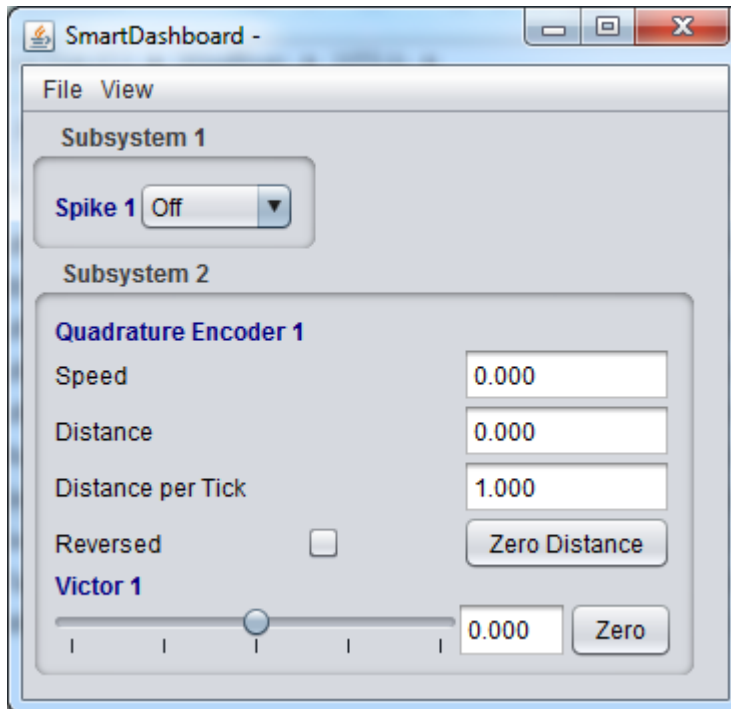
Shuffleboard has the same features as SmartDashboard. It also improves on the setup and visualization of your data with new features and a modern design at the cost of being less resource efficient. Additional documentation on Shuffleboard can be found [here](#).

7.5.4 Glass



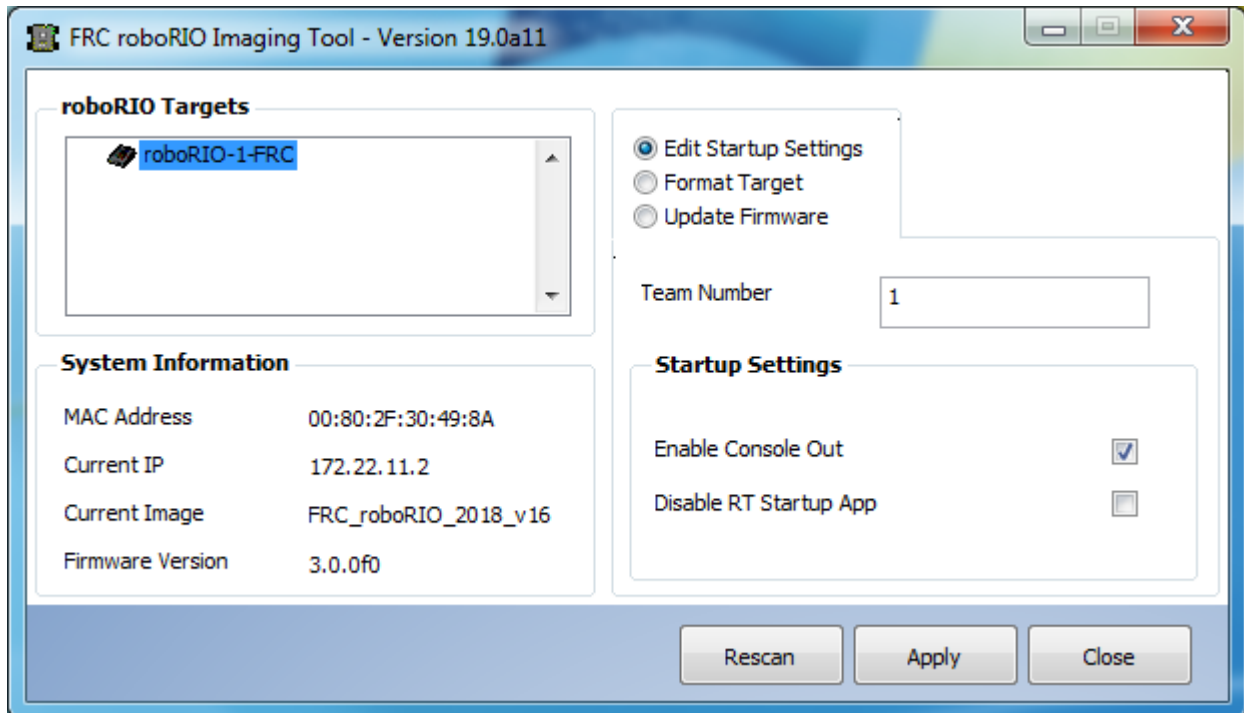
Glass is a Dashboard focused on being a programmer's tool for debugging. The primary advantages are the field view, pose visualization and advanced signal plotting tools.

7.6 LiveWindow



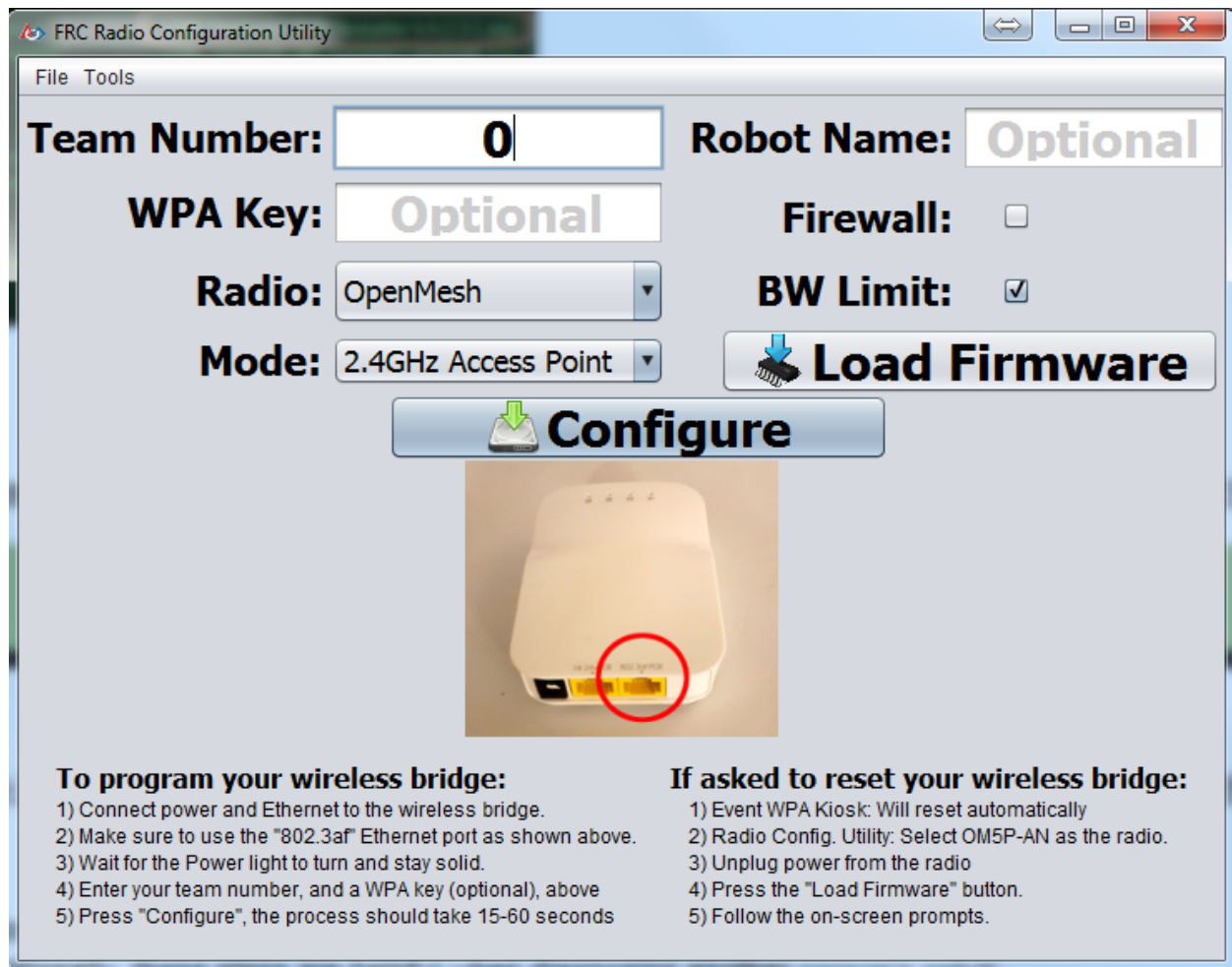
LiveWindow is a feature of SmartDashboard and Shuffleboard, designed for use with the Test Mode of the Driver Station. LiveWindow allows the user to see feedback from sensors on the robot and control actuators independent of the written user code. More information about LiveWindow can be found [here](#).

7.7 FRC roboRIO Imaging Tool (Windows Only)



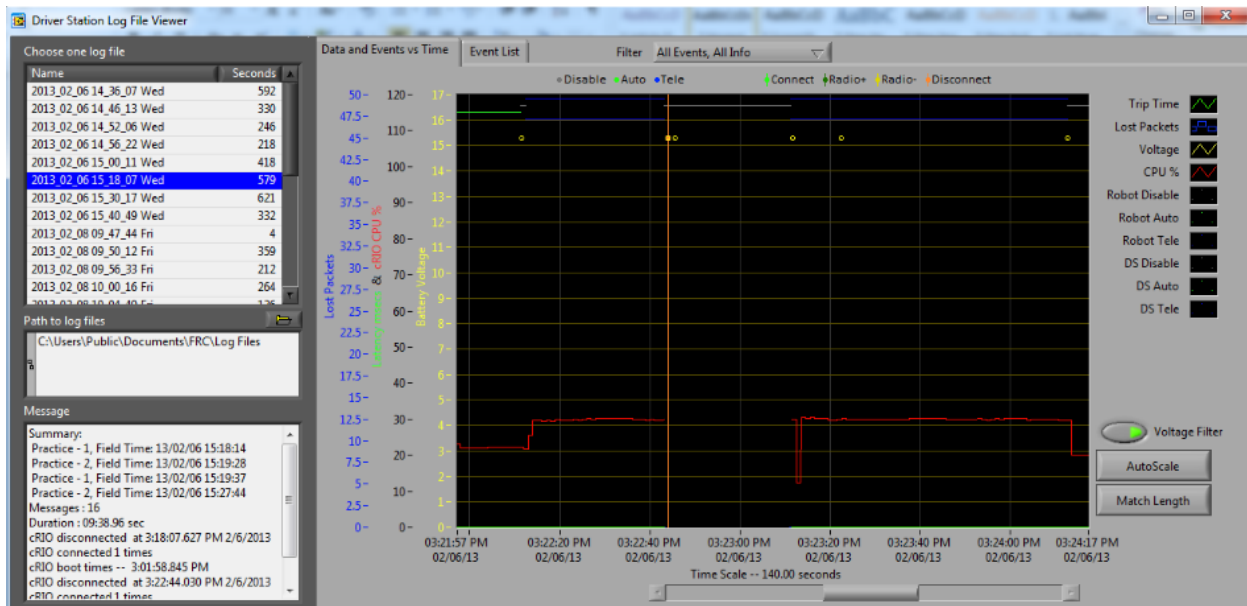
This tool is used to format and setup a roboRIO for use in FRC. Installation instructions can be found [here](#). Additional instructions on imaging your roboRIO using this tool can be found [here](#).

7.8 FRC Radio Configuration Utility (Windows Only)



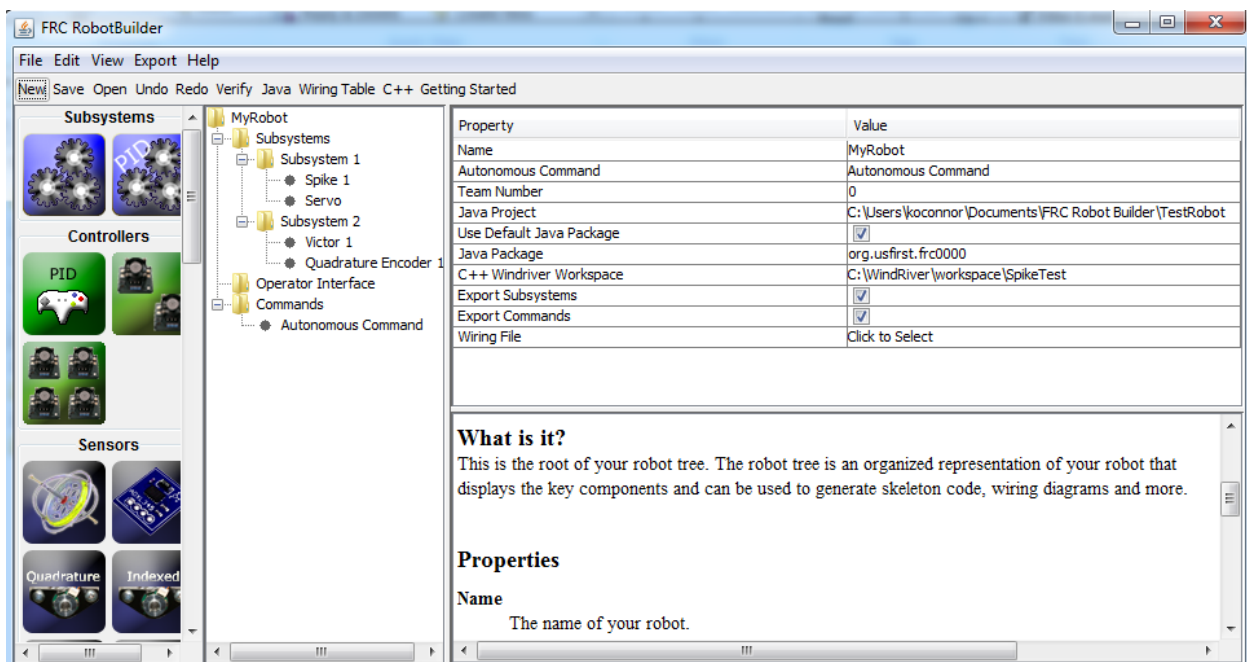
The FRC Radio Configuration Utility is a tool used to configure the standard radio for practice use at home. This tool sets the appropriate network settings to mimic the experience of the FRC playing field. The FRC Radio Configuration Utility is installed by a standalone installer that can be found [here](#).

7.9 FRC Driver Station Log Viewer (Windows Only)



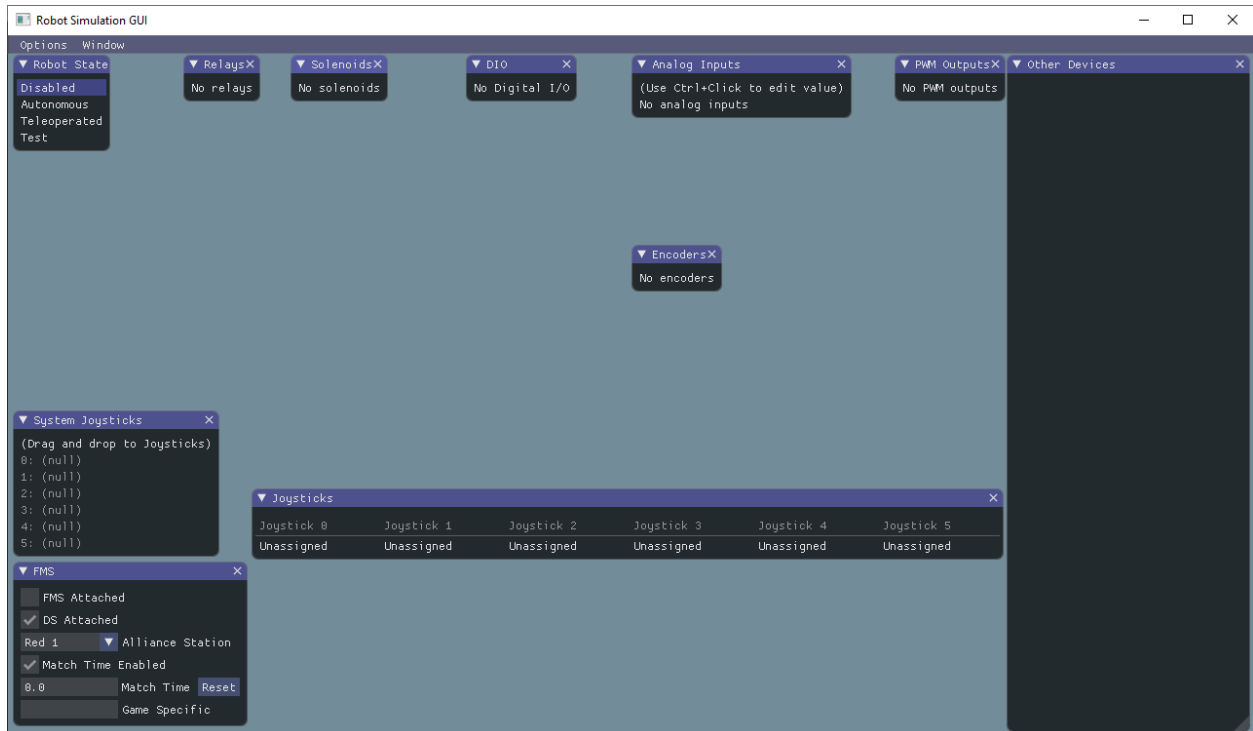
The FRC Driver Station Log Viewer is used to view logs created by the FRC Driver Station. These logs contain a variety of information important for understanding what happened during a practice session or FRC match. More information about the FRC Driver Station Log Viewer and understanding the logs can be found [here](#)

7.10 RobotBuilder



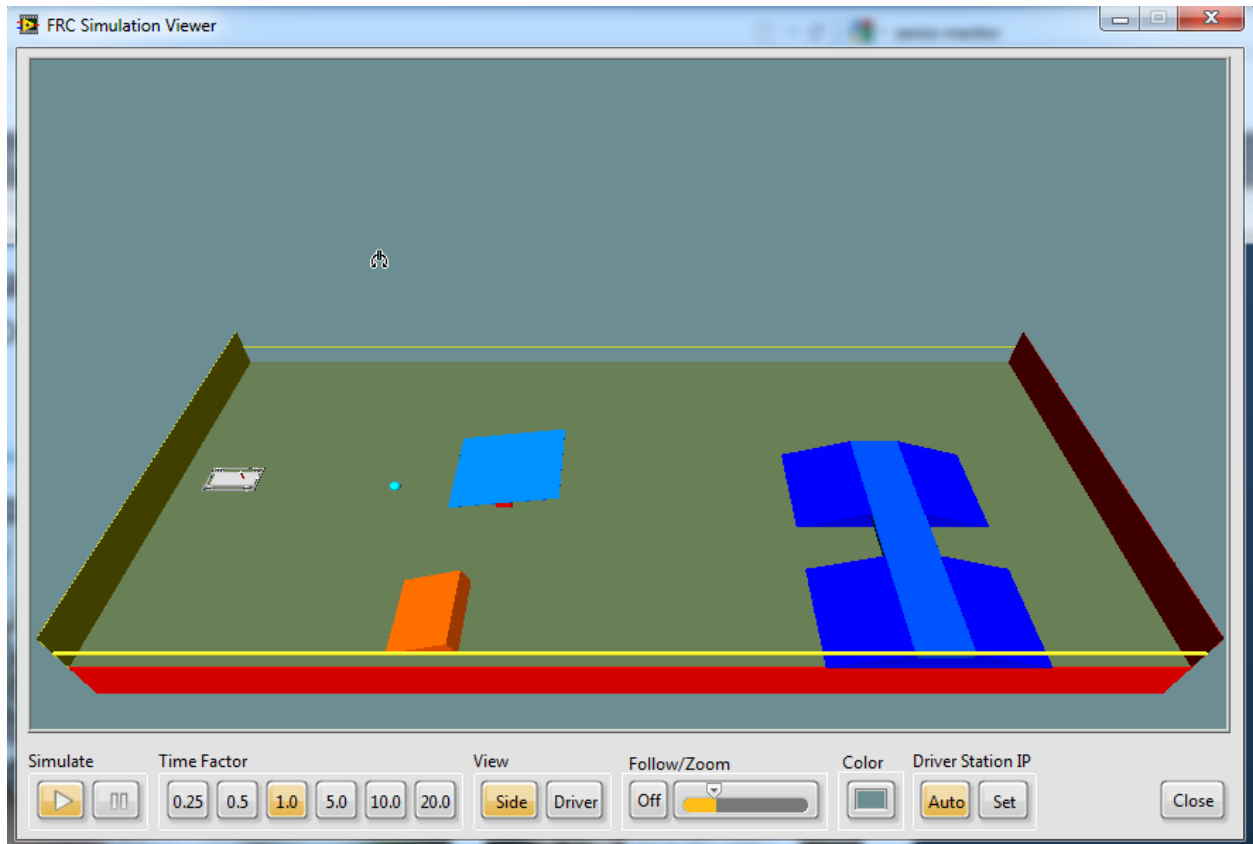
RobotBuilder is a tool designed to aid in setup and structuring of a Command Based robot project for C++ or Java. RobotBuilder allows you to enter in the various components of your robot subsystems and operator interface and define what your commands are in a graphical tree structure. RobotBuilder will then generate structural template code to get you started. More information about RobotBuilder can be found [here](#). More information about the Command Based programming architecture can be found [here](#).

7.11 Robot Simulation



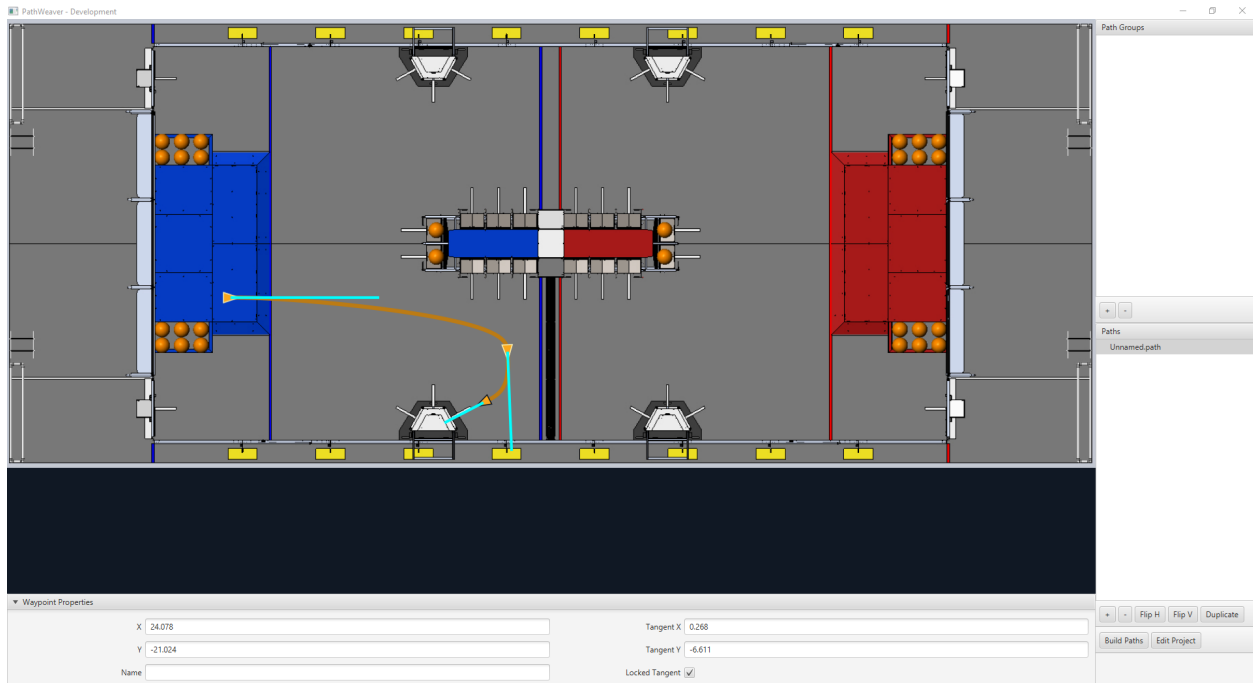
Robot Simulation offers a way for Java and C++ teams to verify their actual robot code is working in a simulated environment. This simulation can be launched directly from VS Code and includes a 2D field that users can visualize their robot's movement on. For more information see the [Robot Simulation section](#).

7.12 FRC LabVIEW Robot Simulator (Windows Only)



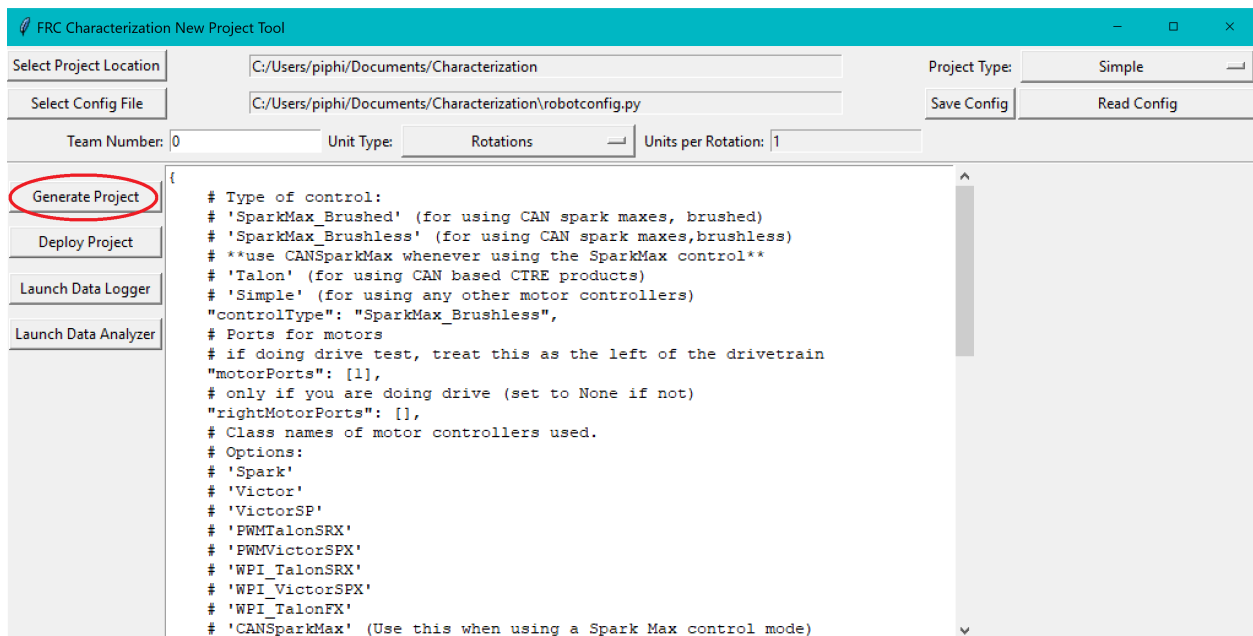
The FRC Robot Simulator is a component of the LabVIEW programming environment that allows you to operate a predefined robot in a simulated environment to test code and/or Driver Station functions. Information on using the FRC Robot Simulator can be found [here](#) or by opening the Robot Simulation Readme.html file in the LabVIEW Project Explorer.

7.13 PathWeaver



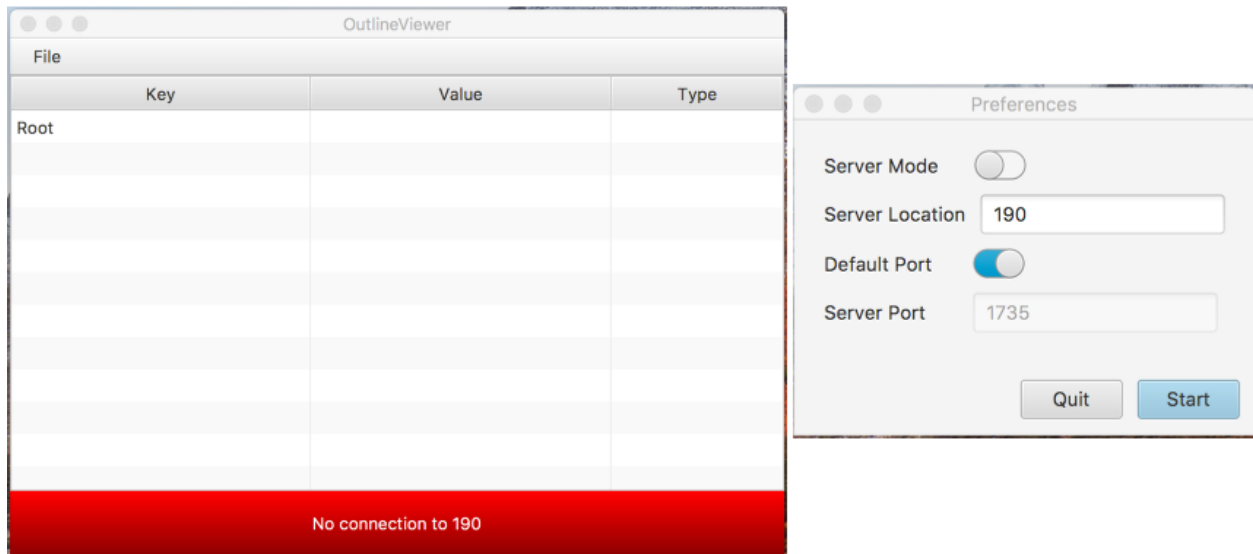
PathWeaver allows teams to quickly generate and configure paths for advanced autonomous routines. These paths have smooth curves allowing the team to quickly navigate their robot between points on the field. For more information see the [PathWeaver section](#).

7.14 Robot Characterization



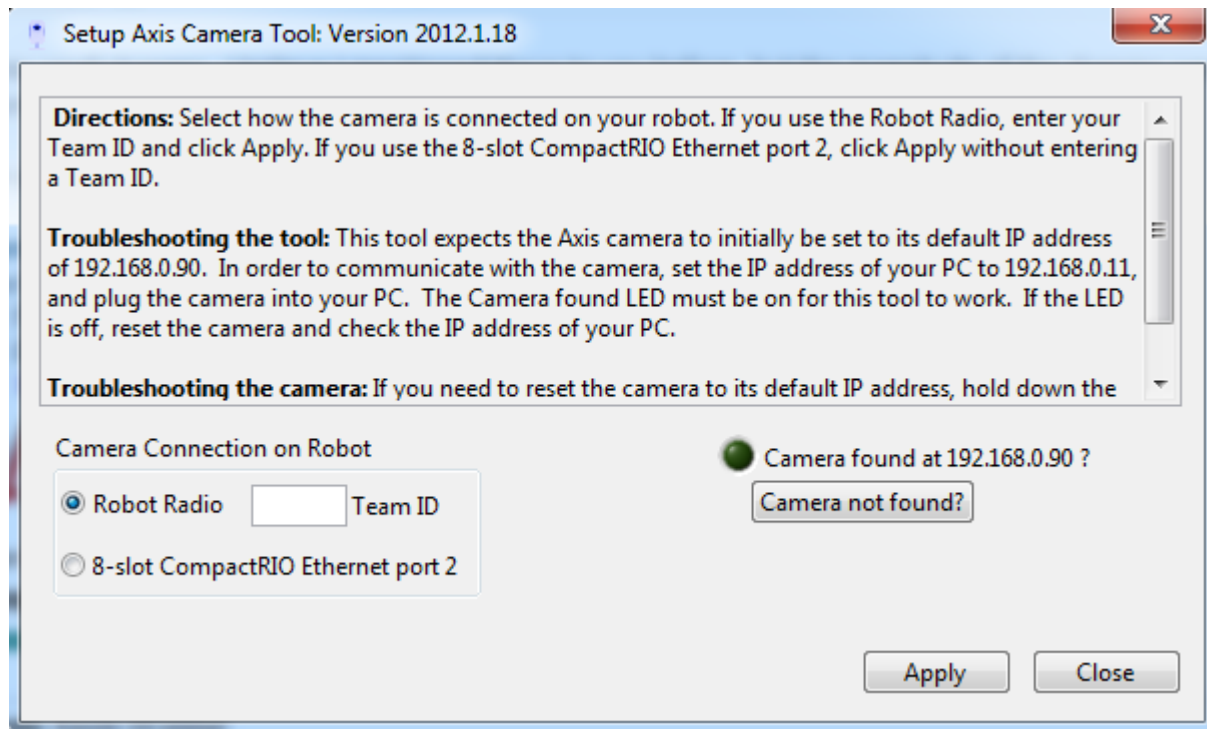
This tool helps teams automatically calculate constants that can be used to describe the physical properties of your robot for use in features like robot simulation, trajectory following, and PID control. For more information see the [Robot Characterization section](#).

7.15 OutlineViewer



OutlineViewer is a utility used to view, modify and add to all of the contents of the NetworkTables for debugging purposes. LabVIEW teams can use the Variables tab of the LabVIEW Dashboard to accomplish this functionality. For more information see the Outline Viewer section.

7.16 Setup Axis Camera (Windows Only)



The Setup Axis Camera utility is a LabVIEW program used to configure an Axis 206, M1011 or M1013 camera for use on the robot. The Setup Axis Camera tool is installed as part of the FRC Game Tools.

What is WPILib?

The WPI Robotics Library (WPILib) is the standard software library provided for teams to write code for their FRC® robots. A [software library](#) is a collection of code that can be imported into and used by other software. WPILib contains a set of useful classes and sub-routines for interfacing with various parts of the FRC control system (such as sensors, motor controllers, and the driver station), as well as an assortment of other utility functions.

8.1 Supported languages

There are two versions of WPILib, one for each of the two officially-supported text-based languages: WPILibJ for java, and WPILibC for C++. A considerable effort is made to maintain feature-parity between these two languages - library features are not added unless they can be reasonably supported for both Java and C++, and when possible the class and method names are kept identical or highly-similar. While unofficial community-built support is available for some other languages, notably [python](#), this documentation will only cover Java and C++. Java and C++ were chosen for the officially-supported languages due to their appropriate level-of-abstraction and ubiquity in both industry and high-school computer science classes.

In general, C++ offers better high-end performance, at the cost of increased user effort (memory must be handled manually, and the C++ compiler does not do much to ensure user code will not crash at runtime). Java offers lesser performance, but much greater convenience. New/inexperienced users are strongly encouraged to use Java.

8.2 Source code and documentation

WPILib is an open-source library - the entirety of its source code is available online on the WPILib GitHub Page:

- [Official WPILib GitHub](#)

The Java and C++ source code can be found in the WPILibJ and WPILibC source directories:

- [Java source code](#)
- [C++ source code](#)

While users are strongly encouraged to read the source code to resolve detailed questions about library functionality, more-concise documentation can be found on the official documentation pages for WPILibJ and WPILibC:

- [Java documentation](#)
- [C++ documentation](#)

2021 Overview

9.1 Known Issues

This article details known issues (and workarounds) for FRC® Control System Software.

9.1.1 Open Issues

No such host is known in the WPILib Installer

The following error message will show up when downloading VS Code with WPILib Installer version 2021.2.1 or earlier **or** version 2021.2.2 that was downloaded before 3/24/2021.

```
System.Net.Http.HttpRequestException: No such host is known.
```

This is due to the VS Code download URL being changed. A hotfix has been published and users should redownload the 2021.2.2 release or later. This fix can be downloaded [here](#).

Invalid build due to missing GradleRIO

Issue: Rarely, a user's Gradle cache will get broken and they will get shown errors similar to the following:

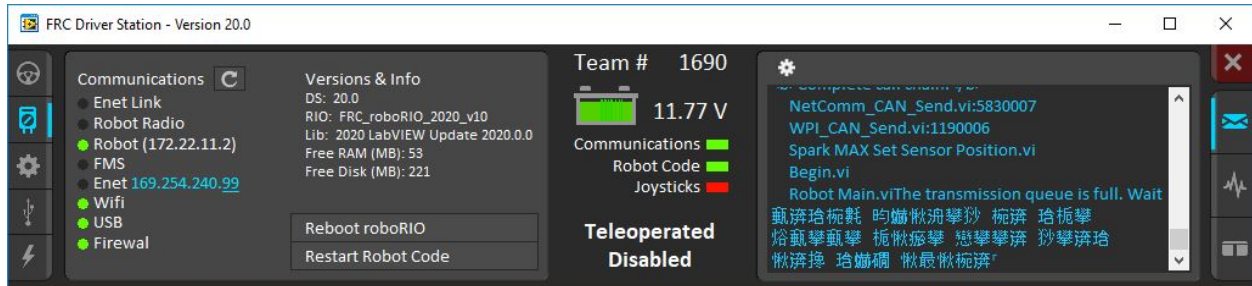
```
Could not apply requested plugin [id: 'edu.wpi.first.GradleRIO', version: '2020.3.2']  
↳ as it does not provide a plugin with id 'edu.wpi.first.GradleRIO'
```

Workaround:

Delete your Gradle cache located under ~\$USER_HOME/.gradle. Windows machines may need to enable the ability to [view hidden files](#). This issue has only shown up on Windows so far. Please [report](#) this issue if you get it on an alternative OS.

Chinese characters in Driver Station Log

Issue: Rarely, the driver station log will show Chinese characters instead of the English text. This appears to only happen when Windows is set to a language other than English.



Workaround: There are two known workarounds:

1. Copy and paste the Chinese characters into notepad, and the English text will be shown.
2. Temporarily change the Windows language to English.

C++ Intellisense - Files Open on Launch Don't Work Properly

Issue: In C++, files open when VS Code launches will have issues with Intellisense showing suggestions from all options from a compilation unit and not just the appropriate ones or not finding header files. This is a bug in VS Code.

Workaround:

1. Close all files in VS Code, but leave VS Code open
2. Delete `c_cpp_properties.json` file in the `.vscode` folder, if it exists
3. Run the "Refresh C++ Intellisense" command in vscode.
4. In the bottom right you should see something that looks like a platform (linuxathena or windowsx86-64 etc). If it's not linuxathena click it and set it to linuxathena (release)
5. Wait ~1 min
6. Open the main cpp file (not a header file). Intellisense should now be working

SmartDashboard and Simulation fail to launch on Windows N Editions

Issue: WPILib code using CSCore (dashboards and simulated robot code) will fail to launch on Education N editions of Windows.

Solution: Install the [Media Feature Pack](#)

NetworkTables Interoperability

There is currently an issue with inter-operating C++/Java [NetworkTables](#) clients (dashboards or co-processors) with LabVIEW servers (LV robot code). In some scenarios users will see updates from one client fail to be replicated across to other clients (e.g. data from a co-processor will not be properly replicated out to a dashboard). Data still continues to return correctly when accessed by code on the server.

Workaround: Write code on the server to mirror any keys you wish to see on other clients (e.g. dashboards) to a separate key. For example, if you have a key named `targetX` being published by a co-processor that you want to show up on a dashboard, you could write code on the robot to read the key and re-write it to a key like `targetXDash`.

9.1.2 Fixed in WPILib 2021.2.2

macOS Mojave Simulation Crash

Issue: The simulation GUI fails to launch on macOS Mojave (10.14.x) due to a dynamic linker error (example below):

```
dyld: lazy symbol binding failed: Symbol not found: _objc_opt_respondToSelector
```

Solution: Upgrade to WPILib 2021.2.2 by using the latest installer or the “Check for WPILib Updates” feature in VS Code.

9.1.3 Fixed in Game Tools 2021 f1

2021_v3.0 Imaging Issue

Issue: An issue was discovered with the roboRIO image version 2021_v3.1 included in the initial release of the 2021 NI Game Tools that prevents successful imaging of roboRIOs.

Solution: Teams that downloaded the 2021 Game Tools installer prior to the morning of January 11, 2021 should [re-download and re-install](#) the latest installer (no need to uninstall first). The new version is 2021 f1. Users with the correct version installed will not see 2021_v3.1 as an option in the roboRIO imaging tool. Once the new version is installed, the roboRIO can be imaged with 2021_v3.0.

9.2 New for 2021

A number of improvements have been made to FRC® Control System software for 2021. This article will describe and provide a brief overview of the new changes and features as well as a more complete changelog for C++/Java WPILib changes. This document only includes the most relevant changes for end users, the full list of changes can be viewed on the various [WPILib](#) GitHub repositories.

Important: Due to internal GradleRIO changes, it is necessary to update previous years projects. After [Installing WPILib for 2021](#), any 2020 projects must be [imported](#) to be compatible.

9.2.1 Major Features

- A hardware-level [WebSocket interface](#) has been added to allow remote access to robot code being simulated in a desktop environment.
- Support for the [Romi](#) robot platform. Romi robot code runs in the desktop simulator environment and talks to the Romi via the new WebSocket interface.
- A new robot data visualizer – [Glass](#) – has been added. Glass has a similar UI to the simulator GUI and supports much of the same features; however, Glass can be used as a standalone dashboard and is not tied in to the robot program.
- The WPILib installer has been rewritten to support macOS and Linux and to improve ease of use.
 - The macOS installer is notarized, eliminating the need for Gatekeeper bypass.
 - Please see the [installation instructions](#) as it differs from previous years.
- Added support for model-based control with Kalman filters, extended Kalman filters, unscented Kalman filters, and linear-quadratic regulators. See [Introduction to State-Space Control](#) for more information.

9.2.2 WPILib

Breaking Changes

- `curvature_t` moved from `frc` to `units` namespace (C++)
- Trajectory constraint methods are now `const` in C++. Teams defining their own custom constraints should mark the `MaxVelocity()` and `MinMaxAcceleration()` methods as `const`.
- The `Field2d` class (added midway through the 2020 season) was moved from the simulation package (`edu.wpi.first.wpilibj.simulation/frc/simulation/`) to the SmartDashboard package (`edu.wpi.first.wpilibj.smartdashboard/frc/SmartDashboard/`). This allows teams to send their robot position over `NetworkTables` to be viewed in Glass. The `Field2d` instance can be sent using `SmartDashboard.putData("Field", m_field2d) / frc::SmartDashboard::PutData("Field", &m_field2d)` or by using one of the [Shuffleboard methods](#). This must be done in order to see the `Field2d` in the Simulator GUI.
- PWM Speed Controllers `get()` method has been modified to return the same value as was `set()` regardless of inversion. The value that still takes into account the inversion can be retrieved with the `getSpeed()` method. This affects the following classes: `DMC60`, `Jaguar`, `PWMSparkMax`, `PWMTalonFX`, `PWMTalonSRX`, `PWMVenom`, `PWMVictorSPX`, `SD540`, `Spark`, `Talon`, `Victor`, and `VictorSP` classes.

New Command-Based Library

- Watchdog and epoch reporting has been added to the command scheduler. This will let teams know exactly which command or subsystem is responsible for a loop overrun if one occurs.
- Added a `withName()` command decorator for Java teams. This lets teams set the name of a particular command using the *decorator pattern*.
- Added a `NetworkButton` class, allowing users to use a boolean `NetworkTableEntry` as a button to trigger commands.
- Added a `simulationPeriodic()` method to `Subsystem`. This method runs periodically during simulation, in addition to the regular `periodic()` method.

General Library

- Holonomic Drive Controller - A controller that teams with holonomic drivetrains (i.e. swerve and mecanum) can use to follow trajectories. This also supports custom `Rotation2d` heading inputs that are separate from the trajectory because heading dynamics are decoupled from translational movement in holonomic drivetrains.
- Added support for scheduling functions more often than the robot loop via `addPeriodic()` in `TimedRobot`. Previously, teams had to make a `Notifier` to run feedback controllers more often than the `TimedRobot` loop period of 20ms (running `TimedRobot` more often than this is not advised). Now, users can run feedback controllers more often than the main robot loop, but synchronously with the `TimedRobot` periodic functions so there aren't any thread safety issues. See an example *here*.
- Added a `toggle()` function to `Solenoid` and `DoubleSolenoid`.
- Added a `SpeedControllerGroup` constructor that takes a `std::vector<> (C++) / SpeedController[] (Java)`, allowing the list to be constructed dynamically. (Teams shouldn't use this directly. This is only intended for bindings in languages like Python.)
- Added methods (`isOperatorControlEnabled()` and `isAutonomousEnabled()`) to check game and enabled state together.
- Added a `ScopedTracer` class for C++ teams to be able to time pieces of code. Simply instantiate the `ScopedTracer` at the top of a block of code and the time will be printed to the console when the instance goes out of scope.
- Added a static method `fromHSV(int h, int s, int v)` to create a `Color` instance from HSV values.
- Added RT priority constructor to `Notifier` in C++. This makes the thread backing the `Notifier` run at real-time priority, reducing timing jitter.
- Added a `DriverStation.getInstance().isJoystickConnected(int)` method to check if a joystick is connected to the Driver Station.
- Added a `DriverStation.getInstance().silenceJoystickConnectionWarning(boolean)` method to silence the warning when a joystick is not connected. This setting has no effect (i.e. warnings will continue to be printed) when the robot is connected to a real FMS.
- Added a constructor to `Translation2d` that takes in a distance and angle. This is effectively converting from polar coordinates to Cartesian coordinates.

- Added `EllipticalRegionConstraint`, `RectangularRegionConstraint`, and `MaxVelocityConstraint` to allow constraining trajectory velocity in a certain region of the field.
- Added `equals()` operator to the `Trajectory` class to compare two or more trajectories.
- Added zero-arg constructor to the `Trajectory` class in Java that creates an empty trajectory.
- Added a special exception to catch trajectory constraint misbehavior. This notifies users when user-defined constraints are misbehaving (i.e. min acceleration is greater than max acceleration).
- Added a `getRotation2d()` method to the `Gyro` interface. This method automatically takes care of converting from gyro conventions to geometry conventions.
- Added angular acceleration units for C++ teams. These are available in the `<units/angular_acceleration.h>` header.
- Added X and Y component getters in `Pose2d` - `getX()` and `getY()` in Java, `X()` and `Y()` in C++.
- Added implicit conversion from `degree_t` to `Rotation2d` in C++. This allows teams to use a degree value (i.e. `47_deg`) wherever a `Rotation2d` is required.
- Fixed bug in path following examples where odometry was not being reset to the starting pose of the trajectory.
- Fixed some spline generation bugs for advanced users who were using control vectors directly.
- Fixed theta controller continuous input in swerve examples. This fixes the behavior where the shortest path is not used during drivetrain rotation.
- Deprecated `units.h`, use individual *units headers* instead which speeds compile times.

9.2.3 Simulation

- Added keyboard virtual joystick simulation support.
- Added `Mechanism2D` for visualizing mechanisms in simulation.
- Added simulation physics classes for common robot mechanisms (`DrivetrainSim`, `ElevatorSim`, `SingleJointedArmSim`, and `FlywheelSim`)

9.2.4 Shuffleboard

- Number Slider now displays the text value
- Graphing Widget now uses `ChartFX`, a high performance graphing library
- Fixed decimal digit formatting with large numbers
- Size and position can now be set separately in the Shuffleboard API
- Analog Input can now be viewed with a Text Widget

9.2.5 SmartDashboard

- Host IP can be specified in configuration.

9.2.6 PathWeaver

- Added support for reversed splines
- The coordinate system in the exported JSON has changed to be compatible with the simulator GUI. See [Importing a PathWeaver JSON](#) for more information.

9.2.7 GradleRIO

- Added a vendordep task for downloading vendor JSONs or fetching them from the user *wpi*lib folder
- Added a gradlerio.vendordep.folder.path property to set a non-default location for the vendor JSON folder
- Renamed the wpi task (that prints current versions of WPILib and tools) to *wpiVersions*
- Added the ability to set environment variables during simulation
 - To set the environment variable HALSIMWS_HOST use:

```
sim {  
    envVar "HALSIMWS_HOST", "10.0.0.2"  
}
```

9.2.8 CSCore

- Now only lists streamable devices on Linux platforms.

9.2.9 Visual Studio Code Extension

- Visual Studio Code has been updated to 1.52.1
- Updated Java and C++ language extensions
- Driverstation sim extension is now enabled by default
- Project importer now retains the commands version used in the original project
- Clarified the text on the new project and project importer screens
- Fixed import corrupting binary files
- Fixed link order in C++ build.gradle projects
- Updated “Change Select Default Simulate Extension Setting” command to work with multiple sim extensions

9.2.10 RobotBuilder

- Updated to be compatible with the new command based framework and PID Controller.
 - Due to the major changes in templates, RobotBuilder will not accept a save file from a previous year. You must regenerate the yaml save file and export to a new directory.
 - A version of RobotBuilder that still exports to the old command based framework has included with the installer and is called RobotBuilder-Old
- C++: use uniform initialization of objects in header
- C++: fixed case of includes so that code compiles on case-sensitive filesystems
- Use project name as default for save file
- Fixed export of wiring file
- Fixed line-endings for scripts so they work on MacOS/Linux
- Added XboxController

9.2.11 Robot Characterization

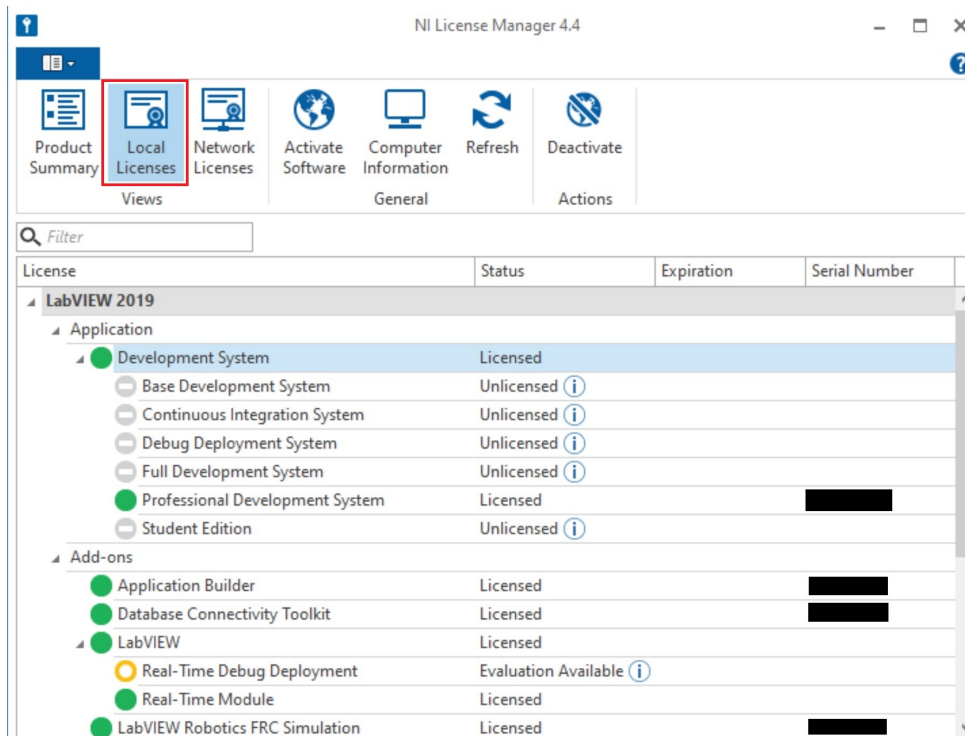
- Added LQR latency compensation
- The tool backend was improved to be more approachable for developers. Configuration and JSON files from the old tool will no longer work with the new version.
- Deploy code in a new thread to avoid causing the GUI to hang.

9.3 Re-licensing LabVIEW for 2021 Season

For the 2021 season we have chosen to remain on LabVIEW for FRC® 2020 (based on LabVIEW 2019) in order to minimize changes for teams in a year where robot re-use is expected. Because this was not anticipated when the 2020 licenses were issued, teams will need to re-license their software with an updated license.

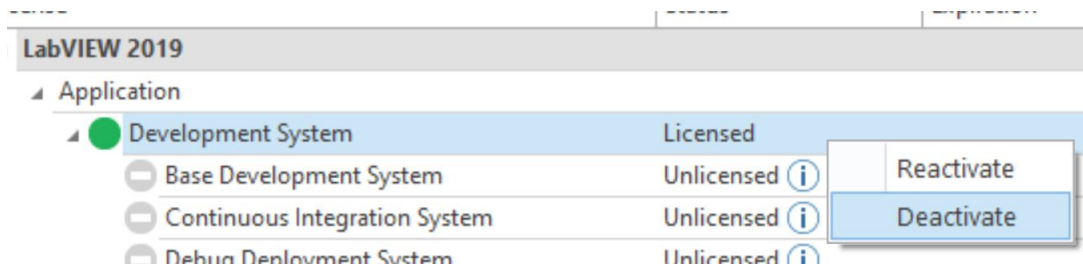
9.3.1 License Manager

Locate and run the NI License Manager software, either from the Start menu, or from Program Files (x86)\National Instruments\Shared\License Manager. Then click *Local Licenses* in the top menu bar.



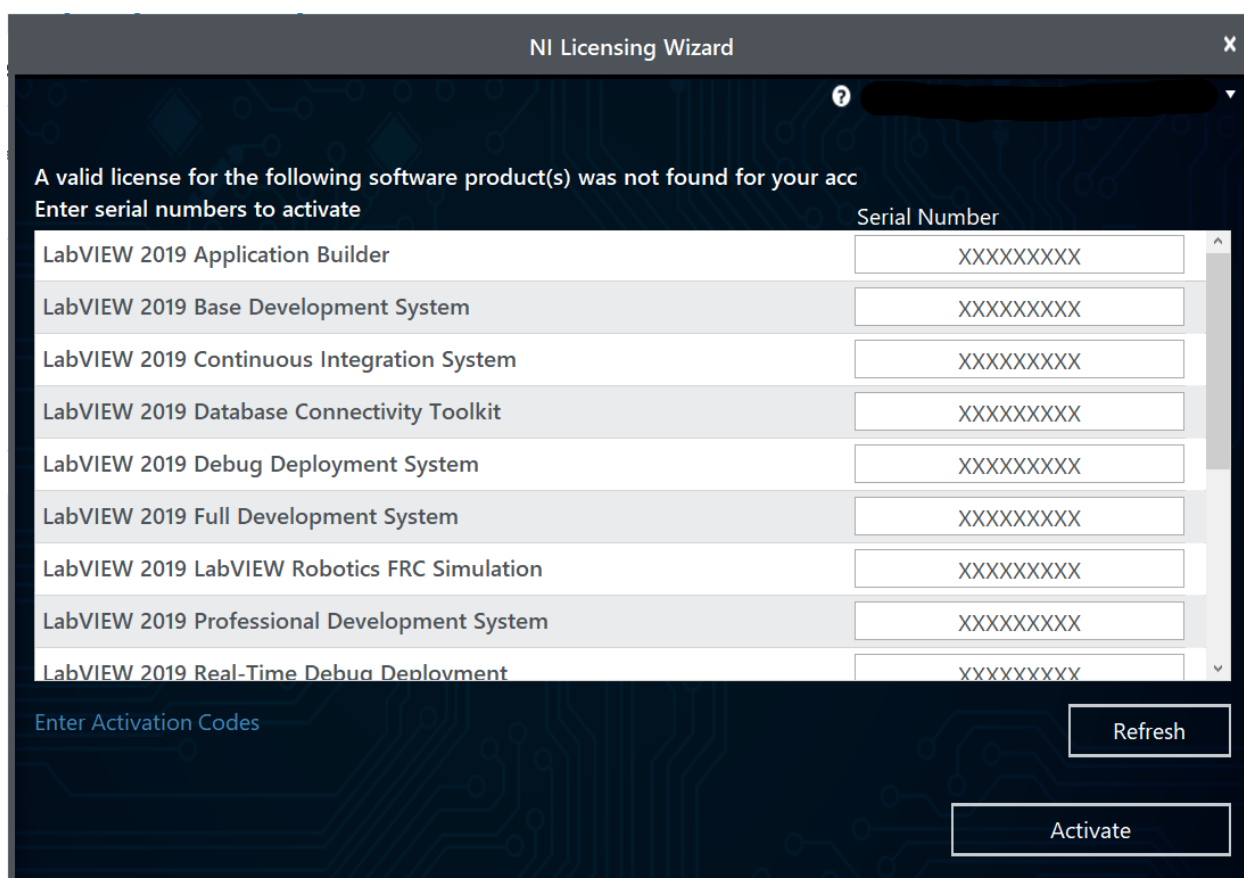
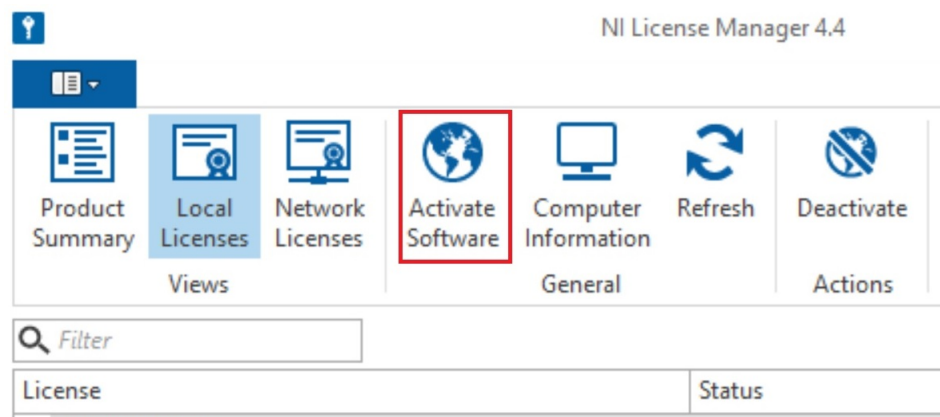
9.3.2 Deactivate Software

For each item that shows a green dot, right click and select *Deactivate*



9.3.3 Reactivate Software

Click *Activate Software* to launch the activation window. Enter your new license key (found in the Lead Mentor Dashboard for your team) in each of the boxes and select *Activate*.



9.4 2020 Game Data Details

In the 2020 *FIRST*® Robotics Competition game, the Position Control objective requires alliances to select a specific color transmitted to them when specific pre-requisites have been met. The field will transmit the selected color to teams using Game Data. This page details the timing and structure of the sent data and provides examples of how to access it in the three supported programming languages.

9.4.1 The Data

Timing

Color assignment data is sent to both alliances simultaneously once the first alliance in a match has reached Capacity on Stage 3 of the Shield Generator (see the Game Manual for more complete details). Between the beginning of the match and this point, the Game Data will be an empty string.

Data format

The selected color for an alliance will be provided as a single character representing the color (i.e. 'R' = red, 'G' = green, 'B' = blue, 'Y' = yellow). This color indicates the color that must be placed underneath the Control Panel's color sensor in order to complete the Position Control objective (see the Game Manual for information about the location of the Control Panel sensor).

9.4.2 Accessing the Data

The data is accessed using the Game Data methods or VIs in each language. Below are descriptions and examples of how to access the data from each of the three languages. As the data is provided to the Robot during the Teleop period, teams will likely want to query the data in Teleop periodic code, or trigger reading it off a button press or other action after they have reached Stage 3 capacity.

C++/Java

In C++ and Java the Game Data is accessed by using the `GetGameSpecificMessage` method of the `DriverStation` class. Teams likely want to query the data in a Teleop method such as Teleop Periodic in order to receive the data after it is sent during the match. Make sure to handle the case where the data is an empty string as this is what the data will be until the criteria are reached to enable Position Control for either alliance.

Java

C++

```
import edu.wpi.first.wpilibj.DriverStation;

String gameData;
gameData = DriverStation.getInstance().getGameSpecificMessage();
```

(continues on next page)

(continued from previous page)

```
if(gameData.length() > 0)
{
    switch (gameData.charAt(0))
    {
        case 'B' :
            //Blue case code
            break;
        case 'G' :
            //Green case code
            break;
        case 'R' :
            //Red case code
            break;
        case 'Y' :
            //Yellow case code
            break;
        default :
            //This is corrupt data
            break;
    }
} else {
    //Code for no data received yet
}
```

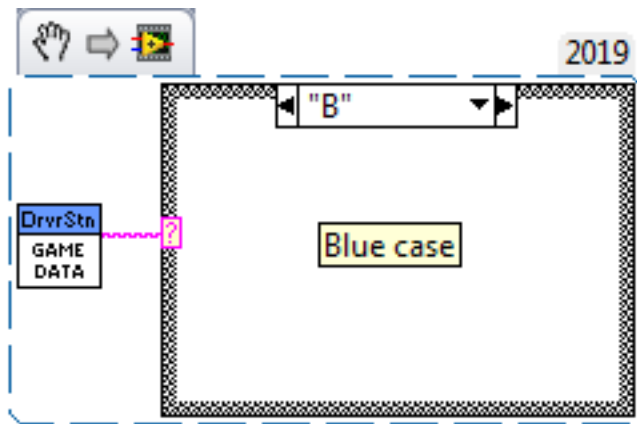
```
#include <frc/DriverStation.h>

std::string gameData;
gameData = frc::DriverStation::GetInstance().GetGameSpecificMessage();
if(gameData.length() > 0)
{
    switch (gameData[0])
    {
        case 'B' :
            //Blue case code
            break;
        case 'G' :
            //Green case code
            break;
        case 'R' :
            //Red case code
            break;
        case 'Y' :
            //Yellow case code
            break;
        default :
            //This is corrupt data
            break;
    }
} else {
    //Code for no data received yet
}
```

LabVIEW

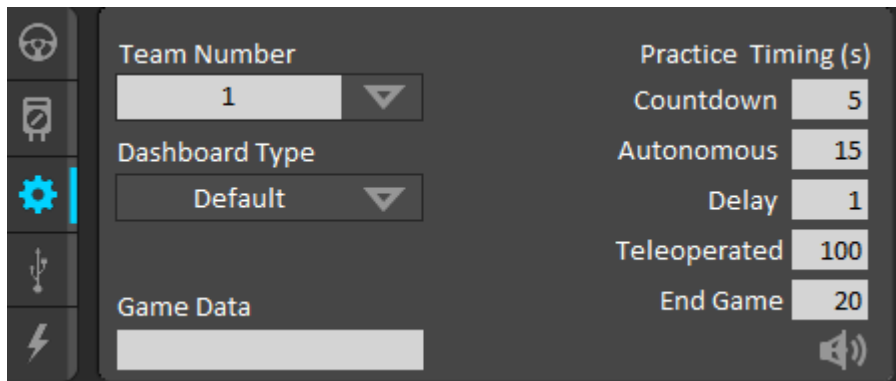
The Game Data in LabVIEW is accessed from the Game Specific Data VI. This VI can be found in the WPI Robotics Library -> Driver Station palette.

LabVIEW teams will likely want to query the data in the Teleop or PeriodicTasks VIs and may choose to gate the query behind a button press or other action. The code below reads the data and then uses a case structure to react differently to each of the 5 possible cases (empty, or any of the 4 letters).



9.4.3 Testing Game Specific Data

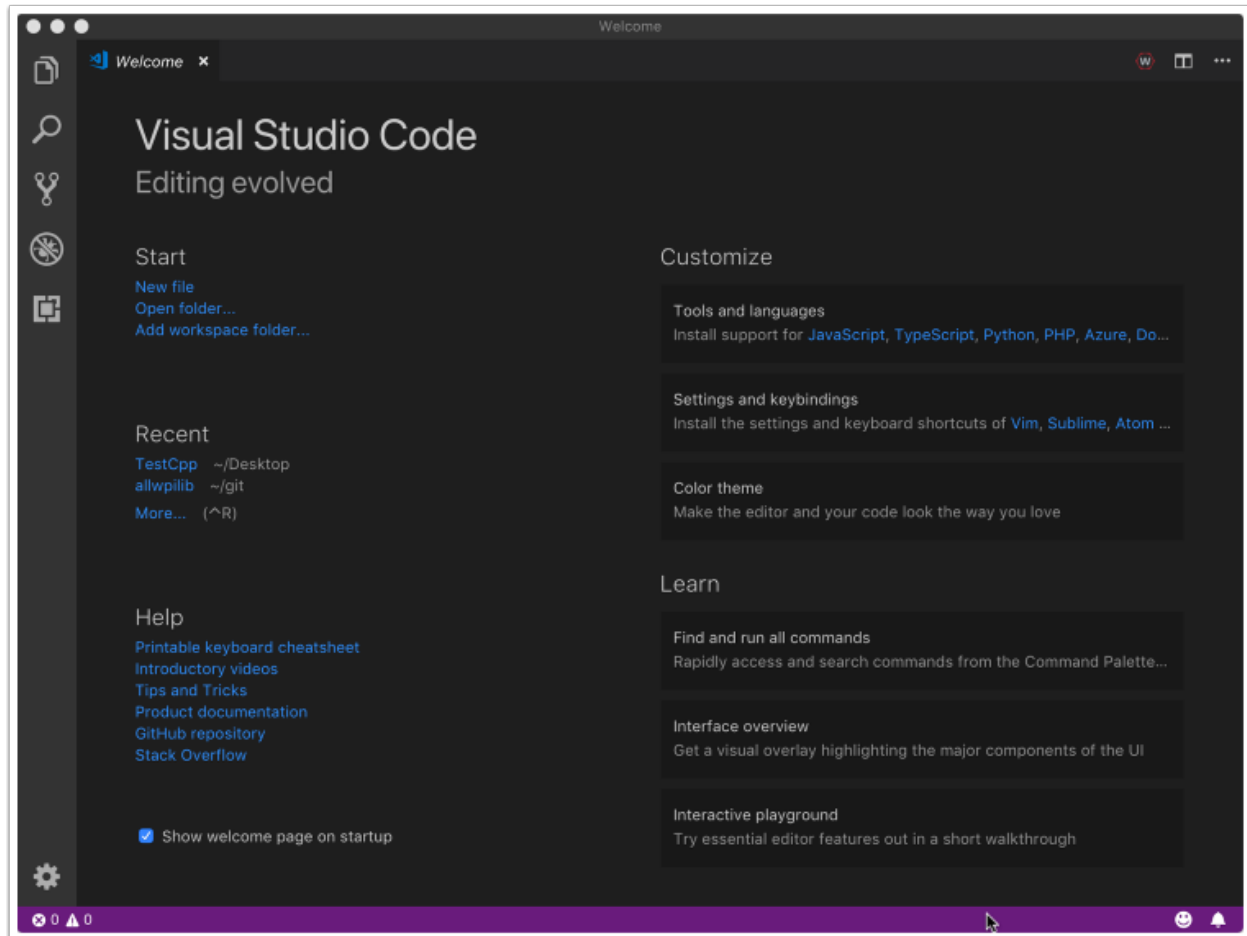
You can test your Game Specific Data code without FMS by using the Driver Station. Click on the Setup tab of the Driver Station, then enter the desired test string into the Game Data text field. The data will be transmitted to the robot in one of two conditions: Enable the robot in Teleop mode, or when the DS reaches the End Game time in a Practice Match (times are configurable on the Setup tab). It is recommended to run at least one match using the Practice functionality to verify that your code works correctly in a full match flow.



10.1 Visual Studio Code Basics and the WPILib Extension

Microsoft's Visual Studio Code is the new supported IDE for C++ and Java development in FRC, replacing the Eclipse IDE used from 2015-2018. This article introduces some of the basics of using Visual Studio Code and the WPILib extension.

10.1.1 Welcome Page



When Visual Studio Code first opens, you are presented with a Welcome page. On this page you will find some quick links that allow you to customize Visual Studio Code as well as a number of links to help documents and videos that may help you learn about the basics of the IDE as well as some tips and tricks.

You may also notice a small WPILib logo way up in the top right corner. This is one way to access the features provided by the WPILib extension (discussed further below).

10.1.2 User Interface

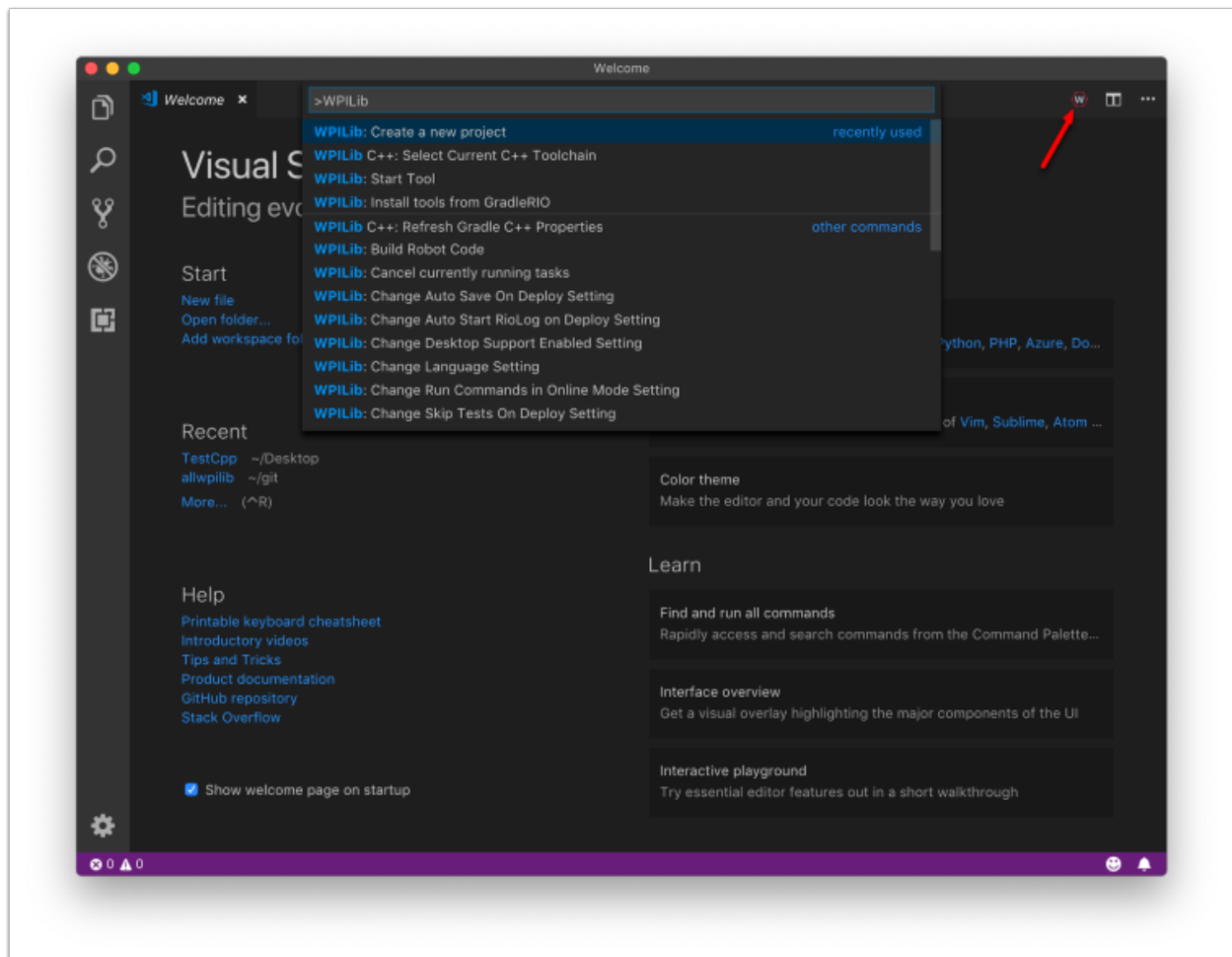
The most important link to take a look at is probably the basic User Interface document. This document describes a lot of the basics of using the UI and provides the majority of the information you should need to get started using Visual Studio Code for FRC.

10.1.3 Command Palette

The Command Palette can be used to access or run almost any function or feature in Visual Studio Code (including those from the WPILib extension). The Command Palette can be accessed from the View menu or by pressing `Ctrl+Shift+P` (`Cmd+Shift+P` on macOS). Typing text into the window will dynamically narrow the search to relevant commands and show them in the dropdown.

In the following example “wpilib” is typed into the search box after activating the Command Palette, and it narrows the list to functions containing WPILib.

10.1.4 WPILib Extension



The WPILib extension provides the FRC® specific functionality related to creating projects and project components, building and downloading code to the roboRIO and more. You can access the WPILib commands one of two ways:

- By typing “WPILib” into the Command Palette
- By clicking on the WPILib icon in the top right of most windows. This will open the Command Palette with “WPILib” pre-entered

Note: It is **not** recommended to install the [Visual Studio IntelliCode](#) plugin with the FRC installation of VS Code as it is known to break IntelliSense in odd ways.

For more information about specific WPILib extension commands, see the other articles in this chapter.

10.2 WPILib Commands in Visual Studio Code

This document contains a complete list of the commands provided by the WPILib VS Code Extension and what they do.

To access these commands, press Ctrl+Shift+P to open the Command Palette, then begin typing the command name as shown here to filter the list of commands. Click on the command name to execute it.

- **WPILib: Build Robot Code** - Builds open project using GradleRIO
- **WPILib: Create a new project** - Create a new robot project
- **WPILib C++: Refresh C++ Intellisense** - Force an update to the C++ Intellisense configuration.
- **WPILib C++: Select current C++ toolchain** - Select the toolchain to use for Intellisense (i.e. desktop vs. roboRIO vs...). This is the same as clicking the current mode in the bottom right status bar.
- **WPILib C++ Select Enabled C++ Intellisense Binary Types** - Switch Intellisense between static, shared, and executable
- **WPILib: Cancel currently running tasks** - Cancel any tasks the WPILib extension is currently running
- **WPILib: Change Auto Save On Deploy Setting** - Change whether files are saved automatically when doing a Deploy. This defaults to Enabled.
- **WPILib: Change Auto Start RioLog on Deploy Setting** - Change whether RioLog starts automatically on deploy. This defaults to Enabled.
- **WPILib: Change Desktop Support Enabled Setting** - Change whether building robot code on Desktop is enabled. Enable this for test and simulation purposes. This defaults to Desktop Support off.
- **WPILib: Change Language Setting** - Change whether the currently open project is C++ or Java.
- **WPILib: Change Run Commands Except Deploy/Debug in Offline Mode Setting** - Change whether GradleRIO is running in Online Mode for commands other than deploy/debug (will attempt to automatically pull dependencies from online). Defaults to enabled (online mode).

- **WPILib: Change Run Deploy/Debug Command in Offline Mode Setting** - Change whether GradleRIO is running in Online Mode for deploy/debug (will attempt to automatically pull dependencies from online). Defaults to disabled (offline mode).
- **WPILib: Change Select Default Simulate Extension Setting** - Change whether simulation extensions are enabled by default (all simulation extensions defined in `build.gradle` will be enabled)
- **WPILib: Change Skip Tests On Deploy Setting** - Change whether to skip tests on deploy. Defaults to disabled (tests are run on deploy)
- **WPILib: Change Stop Simulation on Entry Setting** - Change whether to stop robot code on entry when running simulation. Defaults to disabled (don't stop on entry).
- **WPILib: Check for WPILib Updates** - Check for an update to the WPILib extensions
- **WPILib: Debug Robot Code** - Build and deploy robot code to roboRIO in debug mode and start debugging
- **WPILib: Deploy Robot Code** - Build and deploy robot code to roboRIO
- **WPILib: Import a WPILib 2020 Gradle Project** - Open a wizard to help you create a new project from a existing VS Code Gradle project from 2020. Further documentation is at [importing gradle project](#)
- **WPILib: Import a WPILib Eclipse Project** - Open a wizard to help you create a new VS Code project from an existing WPILib Eclipse project from a previous season. Further documentation is at [importing eclipse project](#)
- **WPILib: Install tools from GradleRIO** - Install the WPILib Java tools (e.g. SmartDashboard, Shuffleboard, etc.). Note that this is done by default by the offline installer
- **WPILib: Manage Vendor Libraries** - Install/update 3rd party libraries
- **WPILib: Open API Documentation** - Opens either the WPILib Javadocs or C++ Doxygen documentation
- **WPILib: Open Project Information** - Opens a widget with project information (Project version, extension version, etc.)
- **WPILib: Open WPILib Command Palette** - This command is used to open a WPILib Command Palette (equivalent of hitting Ctrl+Shift+P and typing WPILib)
- **WPILib: Open WPILib Help** - This opens a simple page which links to the WPILib documentation (this site)
- **WPILib: Reset Ask for WPILib Updates Flag** - This will clear the flag on the current project, allowing you to re-prompt to update a project to the latest WPILib version if you previously chose to not update.
- **WPILib: Run a command in Gradle** - This lets you run an arbitrary command in the GradleRIO command environment
- **WPILib: Set Team Number** - Used to modify the team number associated with a project. This is only needed if you need to change the team number from the one initially specified when creating the project.
- **WPILib: Set VS Code Java Home to FRC Home** - Set the VS Code Java Home variable to point to the Java Home discovered by the FRC extension. This is needed if not using the offline installer to make sure the intellisense settings are in sync with the WPILib build settings.

- **WPILib: Show Log Folder** - Shows the folder where the WPILib extension stores internal logs. This may be useful when debugging/reporting an extension issue to the WPILib developers
- **WPILib: Simulate Robot Code on Desktop** - This builds the current robot code project on your PC and starts it running in simulation. This requires Desktop Support to be set to Enabled.
- **WPILib: Start RioLog** - This starts the RioLog display used to view console output from a robot program
- **WPILib: Start Tool** - This allows you to launch WPILib tools (e.g. SmartDashboard, Shuffleboard, etc.) from inside VS Code
- **WPILib: Test Robot Code** - This builds the current robot code project and runs any created tests. This requires Desktop Support to be set to Enabled.

10.3 Creating a Robot Program

Once everything is installed, we're ready to create a robot program. WPILib comes with several templates for robot programs. Use of these templates is highly recommended for new users; however, advanced users are free to write their own robot code from scratch.

10.3.1 Choosing a Base Class

To start a project using one of the WPILib robot program templates, users must first choose a base class for their robot. Users subclass these base classes to create their primary Robot class, which controls the main flow of the robot program. There are three choices available for the base class:

TimedRobot

Documentation: [Java](#) - [C++](#)

Source: [Java](#) - [C++](#)

The `TimedRobot` class is the the base class recommended for most users. It provides control of the robot program through a collection of `init()` and `periodic()` methods, which are called by WPILib during specific robot states (e.g. autonomous or teleoperated). The `TimedRobot` class also provides an example of retrieving autonomous routines through `SendableChooser` ([Java](#)/ [C++](#))

Note: A *TimedRobot Skeleton* template is available that removes some informative comments and the autonomous example. You can use this if you're already familiar with *Time-dRobot*. The example shown below is of *TimedRobot Skeleton*.

Java

C++

```

import edu.wpi.first.wpilibj.TimedRobot;

public class Robot extends TimedRobot {

    @Override
    public void robotInit() {
        // This is called once when the robot code initializes
    }

    @Override
    public void robotPeriodic() {
        // This is called every period regardless of mode
    }

    @Override
    public void autonomousInit() {
        // This is called once when the robot first enters autonomous mode
    }

    @Override
    public void autonomousPeriodic() {
        // This is called periodically while the robot is in autonomous mode
    }

    @Override
    public void teleopInit() {
        // This is called once when the robot first enters teleoperated mode
    }

    @Override
    public void teleopPeriodic() {
        // This is called periodically while the robot is in teleopreated mode
    }

    @Override
    public void testInit() {
        // This is called once when the robot enters test mode
    }

    @Override
    public void testPeriodic() {
        // This is called periodically while the robot is in test mode
    }

}

```

```

#include <frc/TimedRobot.h>

class Robot : public frc::TimedRobot {
public:
    void RobotInit() override; // This is called once when the robot code
↪initializes
    void RobotPeriodic() override; // This is called every period regardless of
↪mode
    void AutonomousInit() override; // This is called once when the robot first
↪enters autonomous mode
    void AutonomousPeriodic() override; // This is called periodically while the
↪robot is in autonomous mode

```

(continues on next page)

(continued from previous page)

```
void TeleopInit() override; // This is called once when the robot first
↳ enters teleoperated mode
void TeleopPeriodic() override; // This is called periodically while the
↳ robot is in teleopreated mode
void TestInit() override; // This is called once when the robot enters test
↳ mode
void TestPeriodic() override; // This is called periodically while the robot
↳ is in test mode
};
```

Periodic methods are called every 20 ms by default. This can be changed by calling the superclass constructor with the new desired update rate.

Danger: Changing your robot rate can cause some unintended behavior (loop overruns). Teams can also use [Notifiers](#) to schedule methods at a custom rate.

Java

C++

```
public Robot() {
    super(0.03); // Periodic methods will now be called every 30 ms.
}
```

```
Robot() : frc::TimedRobot(30_ms) {}
```

RobotBase

Documentation: [Java](#) - [C++](#)

Source: [Java](#) - [C++](#)

The `RobotBase` class is the most minimal base-class offered, and is generally not recommended for direct use. No robot control flow is handled for the user; everything must be written from scratch inside the `startCompetition()` method. The template by default shows cases how to process the different operation modes (teleop, auto, etc).

Note: A `RobotBase` Skeleton template is available that offers a blank `startCompetition()` method.

Command Robot

Teams using `Command Robot` or `Old Command Robot` should see the [Command-Based Programming Tutorial](#) or [\[Old\] Command Based Programming](#).

Romi

Teams using a *Romi* should use the Romi - Timed or Romi - Command Bot template.

Romi - Timed

The Romi - Timed template provides a `RomiDrivetrain` class that exposes an `arcadeDrive(double xaxisSpeed, double zaxisRotate)` method. It's up to the user to feed this `arcadeDrive` function.

This class also provides functions for retrieving and resetting the Romi's onboard encoders.

Romi - Command Bot

The Romi - Command Bot template provides a `RomiDrivetrain` subsystem that exposes an `arcadeDrive(double xaxisSpeed, double zaxisRotate)` method. It's up to the user to feed this `arcadeDrive` function.

This subsystem also provides functions for retrieving and resetting the Romi's onboard encoders.

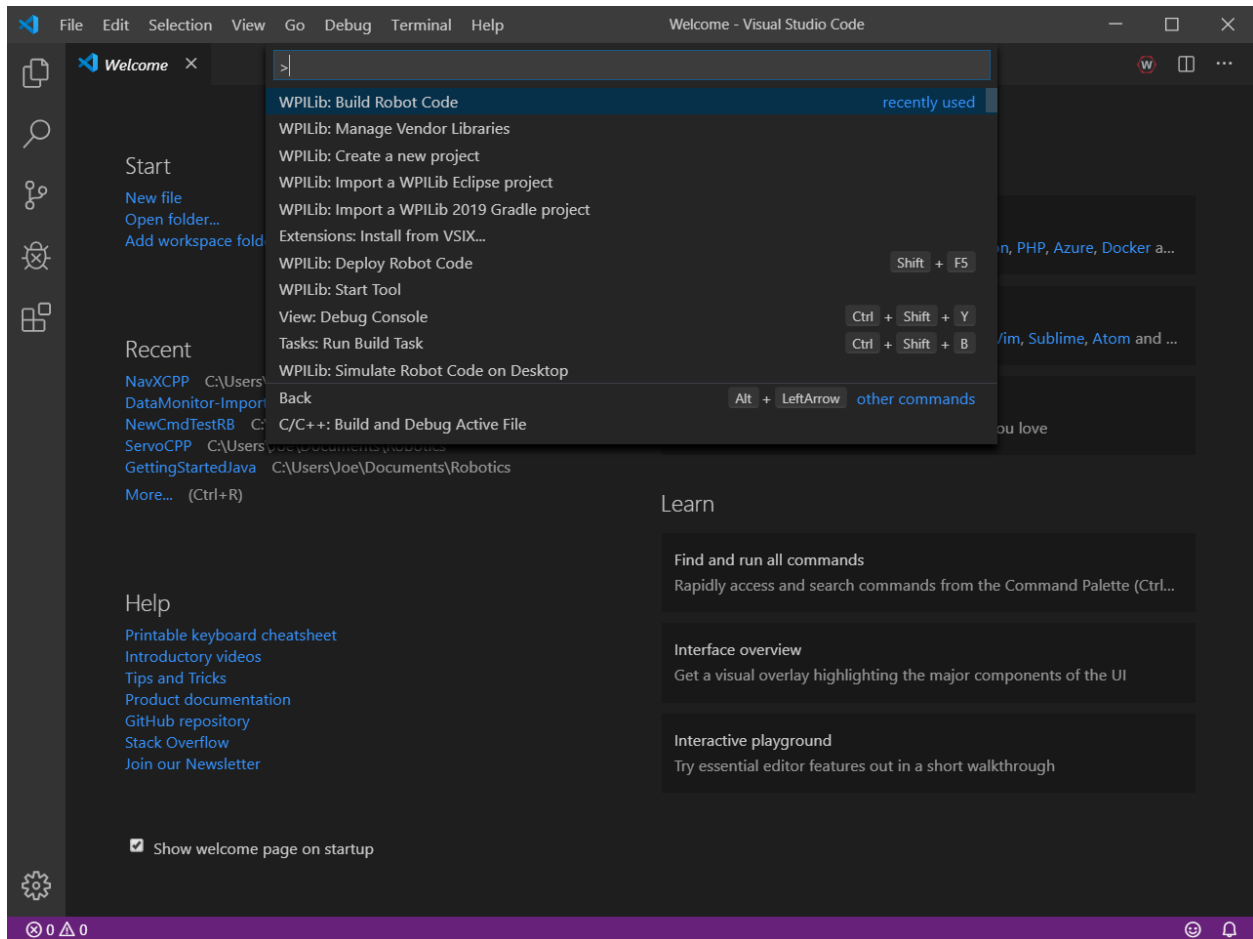
Not Using a Base Class

If desired, users can omit a base class entirely and simply write their program in a `main()` method, as they would for any other program. This is *highly* discouraged - users should not "reinvent the wheel" when writing their robot code - but it is supported for those who wish to have absolute control over their program flow.

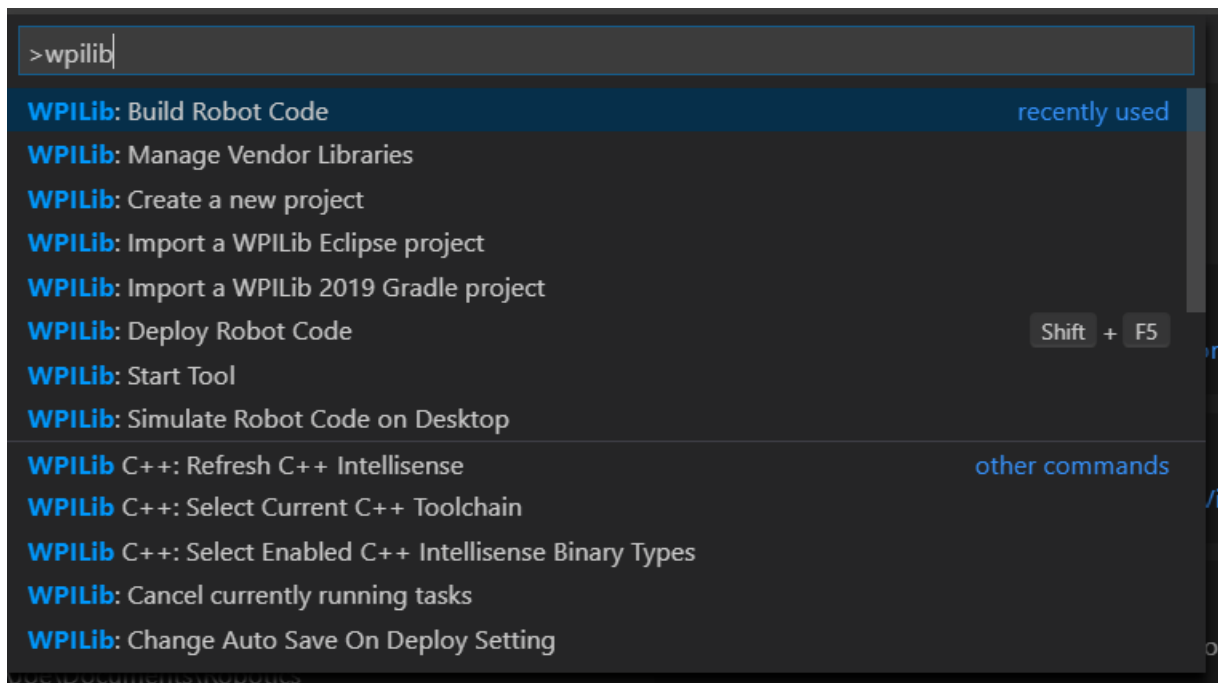
Warning: Users should *not* modify the `main()` method of a robot program unless they are absolutely sure of what they are doing.

10.3.2 Creating a New WPILib Project

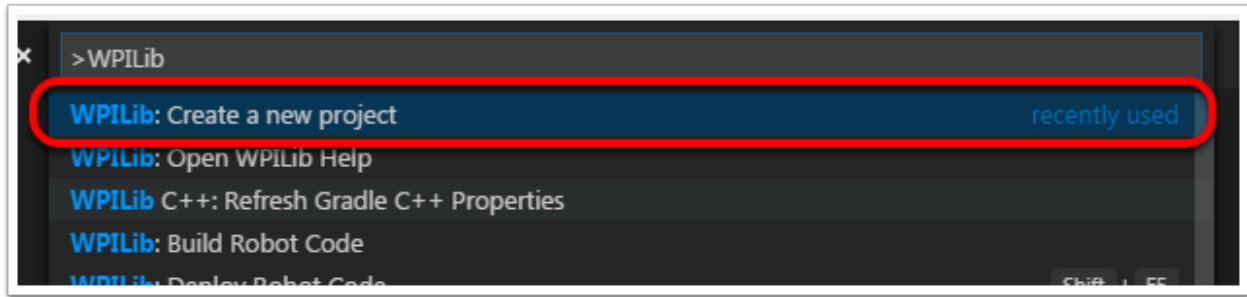
Once we've decided on a base class, we can create our new robot project. Bring up the Visual Studio Code command palette with `Ctrl+Shift+P`:



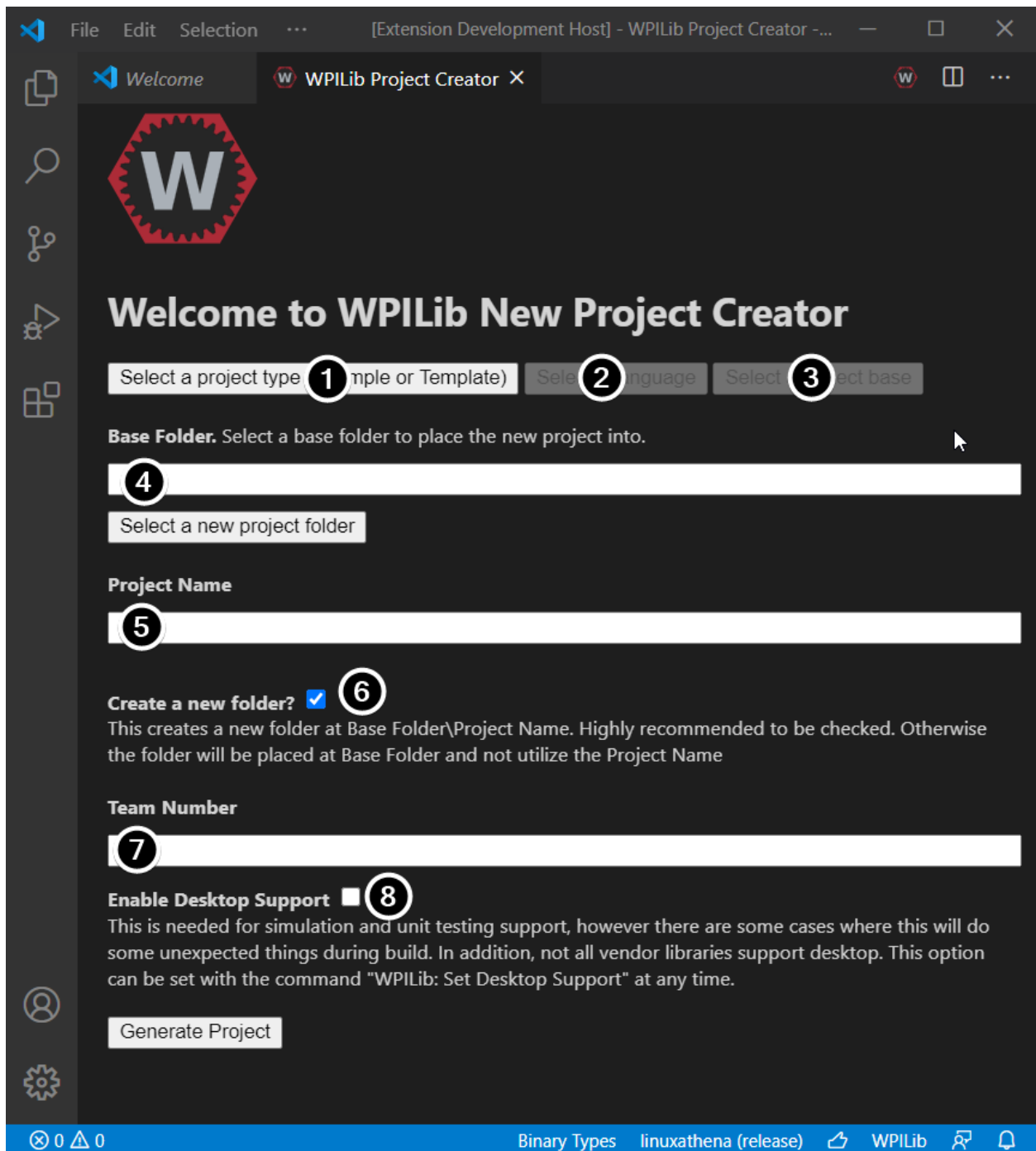
Then, type “WPILib” into the prompt. Since all WPILib commands start with “WPILib,” this will bring up the list of WPILib-specific VS Code commands:



Now, select the *Create a new project* command:



This will bring up the “New Project Creator Window:”



The elements of the New Project Creator Window are explained below:

1. **Project Type:** The kind of project we wish to create. This can be an example project, or one of the project templates provided by WPILib. Templates exist for each of the robot base classes. Additionally, a template exists for *Command-based* projects, which are built on the TimedRobot base class but include a number of additional features - this type of robot program is highly recommended for new teams.
2. **Language:** This is the language (C++ or Java) that will be used for this project.
3. **Base Folder:** If this is a template project, this specifies the type of template that will be

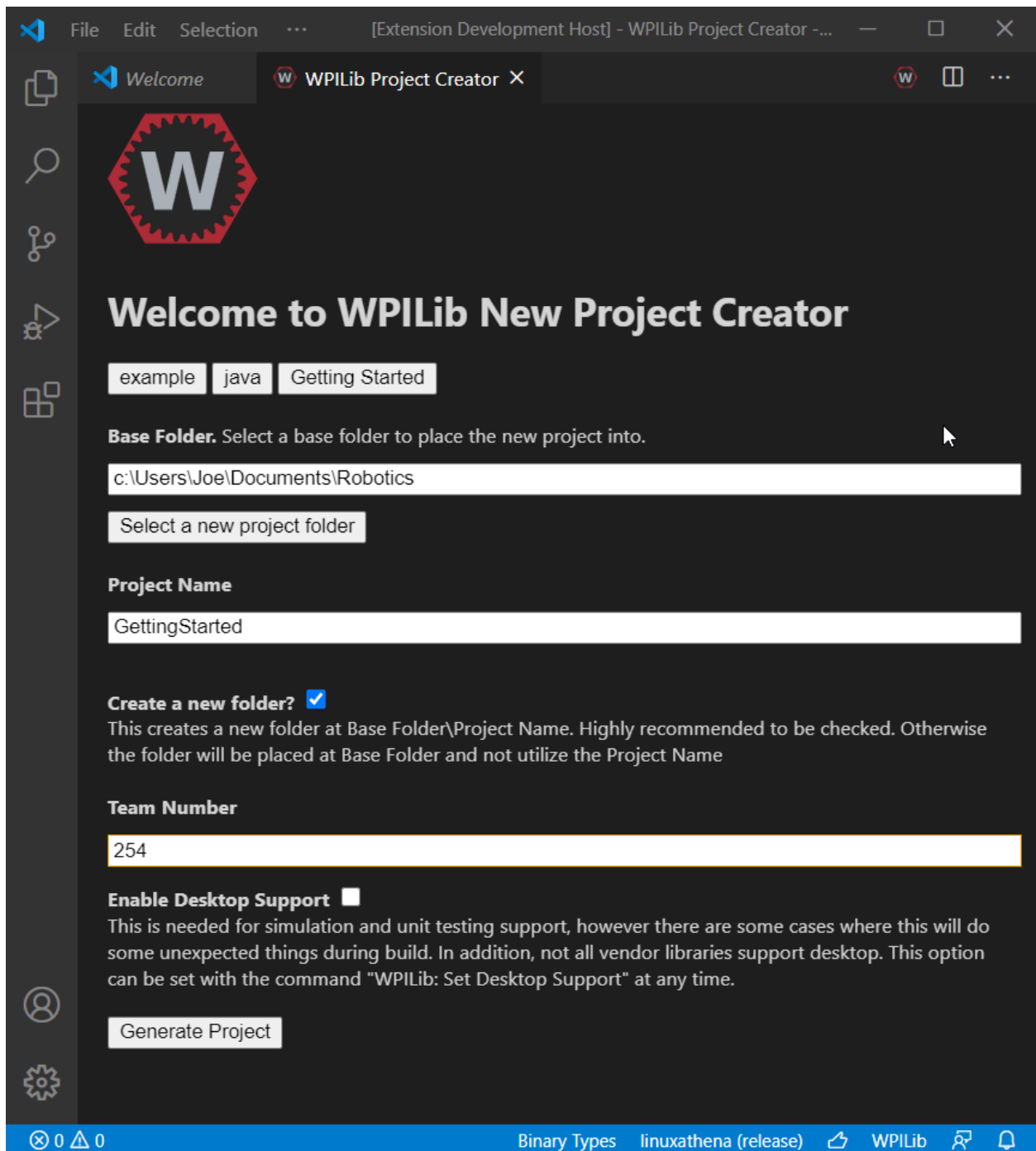
used.

4. **Project Location:** This determines the folder in which the robot project will be located.
5. **Project Name:** The name of the robot project. This also specifies the name that the project folder will be given if the Create New Folder box is checked.
6. **Create a New Folder:** If this is checked, a new folder will be created to hold the project within the previously-specified folder. If it is *not* checked, the project will be located directly in the previously-specified folder. An error will be thrown if the folder is not empty and this is not checked.
7. **Team Number:** The team number for the project, which will be used for package names within the project and to locate the robot when deploying code.
8. **Enable Desktop Support:** Enables unit test and simulation. While WPILib supports this, third party software libraries may not. If libraries do not support desktop, then your code may not compile or may crash. It should be left unchecked unless unit testing or simulation is needed and all libraries support it.

Once all the above have been configured, click “Generate Project” and the robot project will be created.

Note: Any errors in project generation will appear in the bottom right-hand corner of the screen.

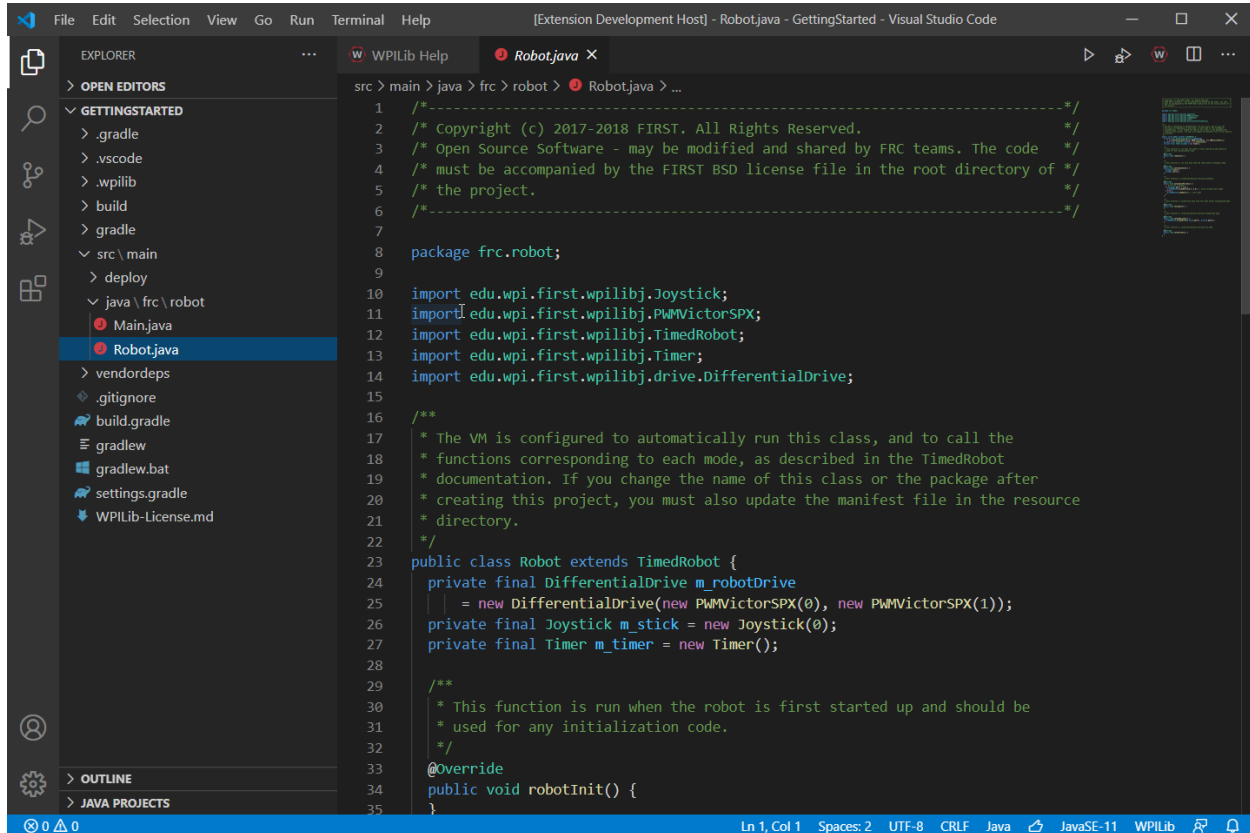
An example after all options are selected is shown below.



10.3.3 Opening The New Project

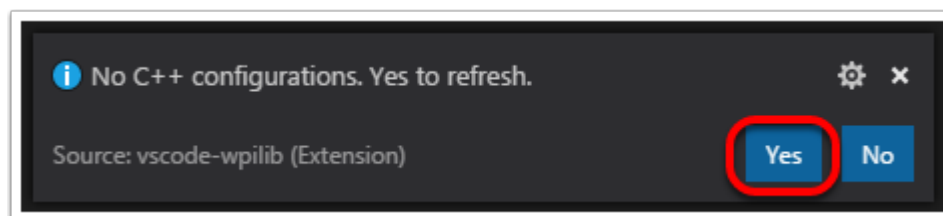
After successfully creating your project, VS Code will give the option of opening the project as shown below. We can choose to do that now or later by typing `Ctrl+K` then `Ctrl+O` (or just `Command+O` on macOS) and select the folder where we saved our project.

Once opened we will see the project hierarchy on the left. Double clicking on the file will open that file in the editor.



10.3.4 C++ Configurations (C++ Only)

For C++ projects, there is one more step to set up IntelliSense. Whenever we open a project, we should get a pop-up in the bottom right corner asking to refresh C++ configurations. Click “Yes” to set up IntelliSense.



10.4 3rd Party Libraries

Teams that are not using parts provided in the KoP will likely need to install external dependencies. Please view the below section to see information on adding an external library.

10.4.1 The Mechanism

In support of this effort NI (for LabVIEW) and FIRST/WPI (for Java/C++) have developed mechanisms that should make it easy for vendors to plug their code into the WPILib software and for teams to use that code once it has been installed. A brief description of how the system works for each language can be found below.

The Mechanism - Java/C++

For Java and C++ a JSON file describing the vendor library is installed on your system to `~/wpilib/YYYY/vendordeps` (where YYYY is the year and ~ is `C:\Users\Public` on Windows). This can either be done by an offline installer or the file can be fetched from an online location using the menu item in Visual Studio Code. This file is then used from VS Code to add to the library to each individual project. Vendor library information is managed on a per-project basis to make sure that a project is always pointing to a consistent version of a given vendor library. The libraries themselves are placed in the Maven cache at `C:\Users\Public\wpilib\YYYY\maven`. Vendors can place a local copy here with an offline installer (recommended) or require users to be online for an initial build to fetch the library from a remote Maven location.

The JSON file allows specification of complex libraries with multiple components (Java, C++, JNI, etc.) and also helps handle some complexities related to simulation. Vendors choosing to provide a remote URL in the JSON also enable users to check for updates from within VS Code.

Note: The vendor JSON files are actually processed by GradleRIO once they are in your projects `vendordeps` folder. If you are using another IDE, you will need to manually create a “`vendordeps`” folder in your project and copy any desired vendor JSON files from the “`wpilib/YYYY`” folder (where they should be placed by an offline installer) or download them directly from the vendor and place them into the folder in the project.

The Mechanism - LabVIEW

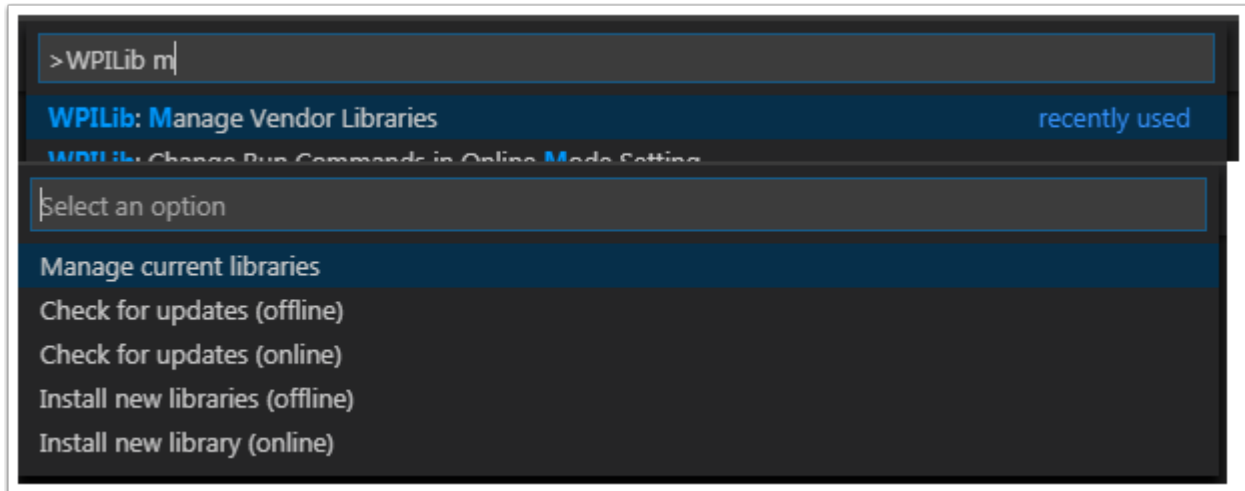
For LabVIEW teams, you may notice a few new Third Party items on various palettes (specifically, one in *Actuators*, one in *Actuators -> Motor Control* labeled *CAN Motor*, and one in *Sensors*). These correspond to folders in `C:\Program Files\National Instruments\LabVIEW\YYYY\vi.lib\Rock Robotics\WPI\Third Party` where YYYY is the current year - 1. If it's 2020, the directory would be `LabVIEW 2019`.

To use installed “Third Party” libraries, simply locate the VIs in one of these 3 locations and drag them into your project as you would with any other VI.

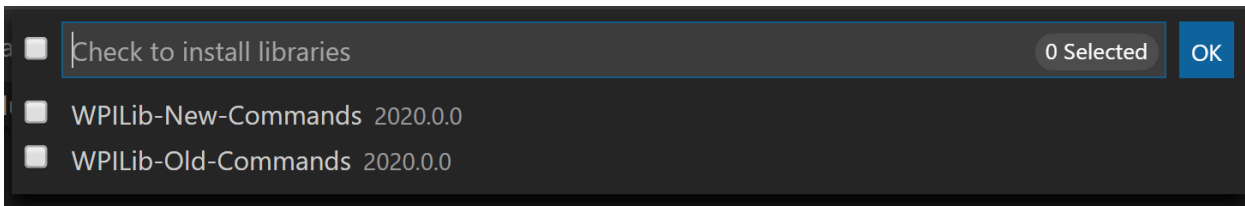
10.4.2 Managing VS Code Libraries

Adding Offline Libraries

VS Code



To add a vendor library that has been installed by an offline installer, press `Ctrl+Shift+P` and type `WPILib` or click on the `WPILib` icon in the top right to open the `WPILib` Command Palette and begin typing *Manage Vendor Libraries*, then select it from the menu. Select the option to *Install new libraries (offline)*.



Select the desired libraries to add to the project by checking the box next to each, then click `OK`. The JSON file will be copied to the `vendordeps` folder in the project, adding the library as a dependency to the project.

Command-line

Adding a vendor library dependency from the vendor URL can also be done through the command-line via a gradle task. Open a command-line instance at the project root, and enter `gradlew vendordep --url=<url>` where `<url>` is the vendor JSON URL. This will add the vendor library dependency JSON file to the `vendordeps` folder of the project. Vendor libraries can be updated the same way.

The `vendordep` gradle task can also fetch `vendordep` JSONs from the user `wpiLib` folder. To do so, pass `FRCLocal/Filename.json` as the file URL. For example, `gradlew vendordep -url=FRCLocal/WPILibNewCommands.json` will fetch the JSON for the new command-based framework.

Checking for Updates (Offline)

Dependencies are now version managed and done on a per-project bases. Even if you have installed an updated library using an offline installer, you will need to *Manage Vendor Libraries* and select *Check for updates (offline)* for each project you wish to update.

Checking for Updates (Online)

Part of the JSON file that vendors may optionally populate is an online update location. If a library has an appropriate location specified, running *Check for updates (online)* will check if a newer version of the library is available from the remote location.

Removing a Library Dependency

To remove a library dependency from a project, select *Manage Current Libraries* from the *Manage Vendor Libraries* menu, check the box for any libraries to uninstall and click *OK*. These libraries will be removed as dependencies from the project.

10.4.3 Libraries

Warning: These are **not** links to directly plug in to the *VS Code -> Install New Libraries (online)* feature. Click these links to visit the vendor site to see whether they offer online installers, offline installers, or both.

Analog Devices ADIS16448 IMU - Driver for ADIS16448 IMU. More info [here](#)

Analog Devices ADIS16470 IMU - Driver for ADIS16470 IMU. More info [here](#)

Copperforge LibCu Software Library - Library for all Copperforge devices including the Laser-shark

CTRE Phoenix Framework - Contains CANcoder, CANifier, Pigeon IMU, Talon FX, Talon SRX, and Victor SPX Libraries and Phoenix Tuner program for configuring CTRE CAN devices

Digilent - DMC-60C library

Playing With Fusion Driver - Library for all PWF devices including the Venom motor/controller

Kauai Labs - Libraries for NavX-MXP, NavX-Micro, and Sensor Fusion

Rev Robotics SPARK MAX - SPARK MAX Library

10.4.4 Community Libraries

PhotonVision - Library for PhotonVision CV software

10.4.5 WPILib Command Libraries

The WPILib *old* and *new* command libraries have been split into vendor libraries in order to reduce the chances of mixing the two which will not work correctly. They are both installed by the wpilib installer for offline installation. They may also be installed with the following online links:

[Old Command Library](#) [New Command Library](#)

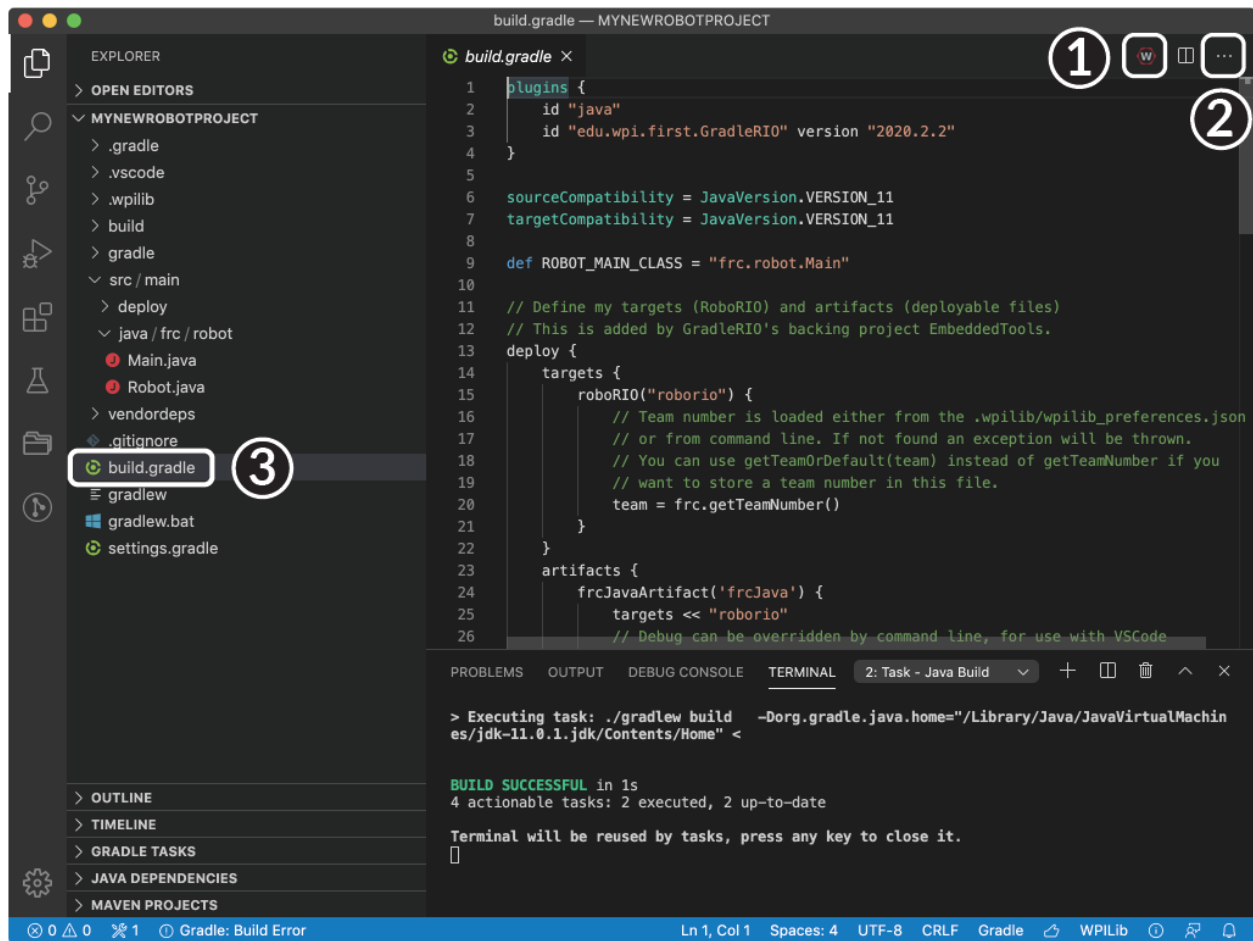
To remove a library dependency from a project, select **Manage Current Libraries** from the **Manage Vendor Libraries** menu, check the box for any libraries to uninstall and click OK. These libraries will be removed as dependencies from the project.

10.5 Building and Deploying Robot Code

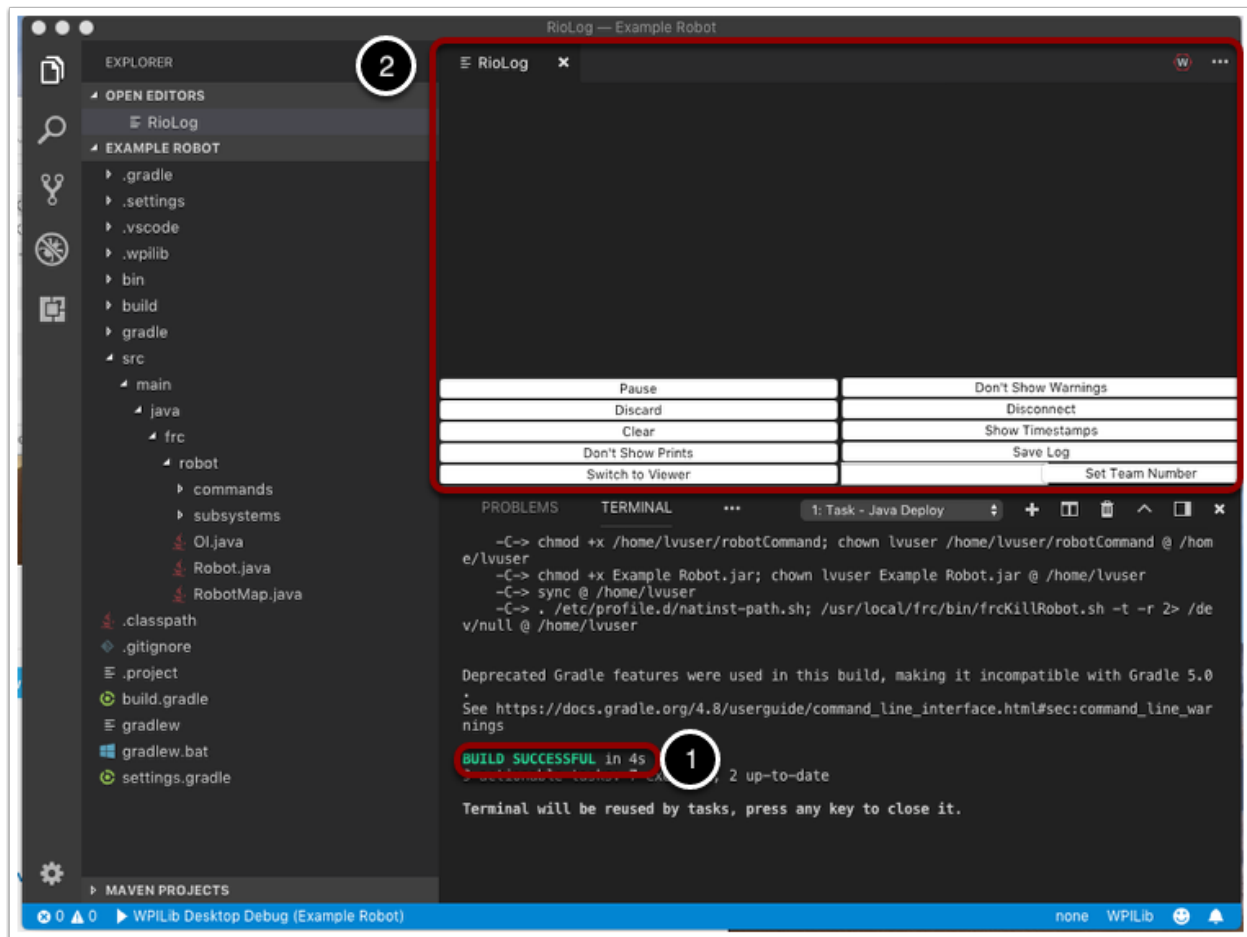
Robot projects must be compiled (“built”) and deployed in order to run on the roboRIO. Since the code is not compiled natively on the robot controller, this is known as “cross-compilation.”

To build and deploy a robot project, do one of:

1. Open the Command Palette and enter/select “Build Robot Code”
2. Open the shortcut menu indicated by the ellipses in the top right corner of the VS Code window and select “Build Robot Code”
3. Right-click on the build.gradle file in the project hierarchy and select “Build Robot Code”



Deploy robot code by selecting “Deploy Robot Code” from any of the three locations from the previous instructions. That will build (if necessary) and deploy the robot program to the roboRIO. If successful, we will see a “Build Successful” message (1) and the RioLog will open with the console output from the robot program as it runs (2).



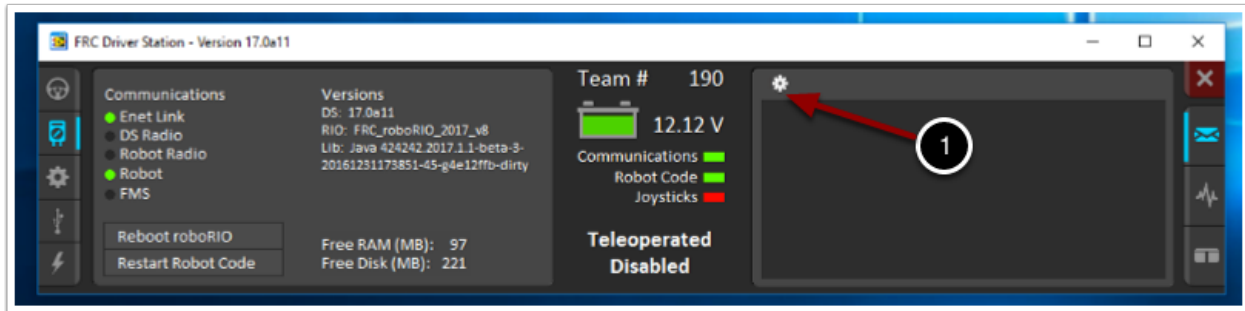
10.6 Viewing Console Output

For viewing the console output of text based programs the roboRIO implements a NetConsole. There are two main ways to view the NetConsole output from the roboRIO: The Console Viewer in the FRC Driver Station and the RioLog plugin in VS Code.

Note: On the roboRIO, the NetConsole is only for program output. If you want to interact with the system console you will need to use SSH or the Serial console.

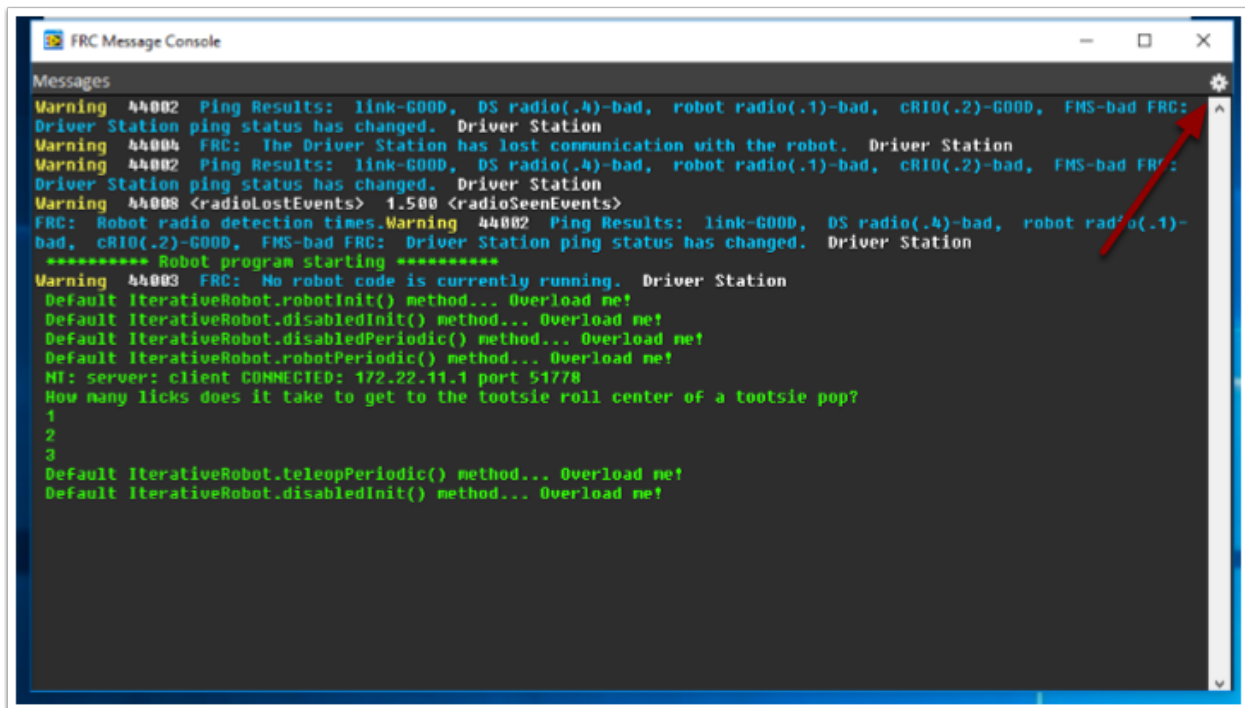
10.6.1 Console Viewer

Opening the Console Viewer



To open Console Viewer, first open the FRC® Driver Station. Then, click on the gear at the top of the message viewer window (1) and select “View Console”.

Console Viewer Window

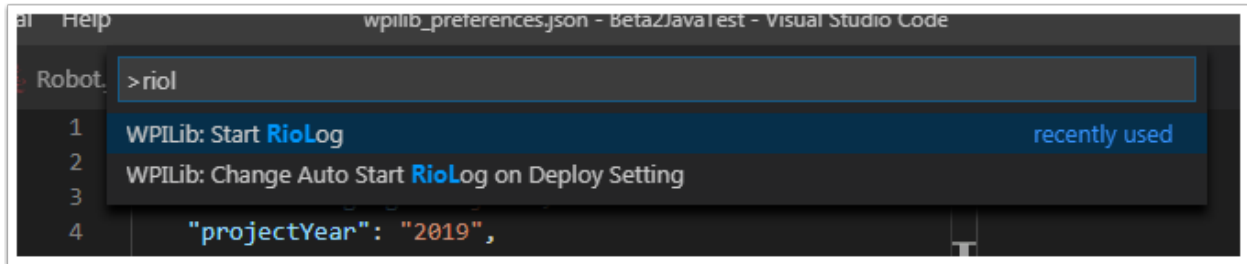


The Console Viewer window displays the output from our robot program in green. The gear in the top right can clear the window and set the level of messages displayed.

10.6.2 Riolog VS Code Plugin

The Riolog plugin is a VS Code view that can be used to view the NetConsole output in VS Code (credit for the original Eclipse version: Manuel Stoeckl, FRC1511).

Opening the Riolog View



By default, the Riolog view will open automatically at the end of each roboRIO deploy. To launch the Riolog view manually, press **Ctrl+Shift+P** to open the command palette and start typing "Riolog", then select the **WPILib: Start Riolog** option.

Riolog Window



The Riolog view should appear in the top pane. The Riolog contains a number of controls for manipulating the console:

- **Pause/Resume Display** - This will pause/resume the display. In the background, the new packets will still be received and will be displayed when the resume button is clicked.

- **Discard/Accept Incoming** - This will toggle whether to accept new packets. When packets are being discarded the display will be paused and all packets received will be discarded. Clicking the button again will resume receiving packets.
- **Clear** - This will clear the current contents of the display.
- **Don't Show/Show Prints** - This shows or hides messages categorized as print statements
- **Switch to Viewer** - This switches to viewer for saved log files
- **Don't Show/Show Warnings** - This shows or hides messages categorized as warnings
- **Disconnect/Reconnect** - This disconnects or reconnects to the console stream
- **Show/Don't Show Timestamps** - Shows or hides timestamps on messages in the window
- **Save Log** - Copies the log contents into a file you can save and view or open later with the RioLog viewer (see Switch to Viewer above)
- **Set Team Number** - Sets the team number of the roboRIO to connect to the console stream on, set automatically if RioLog is launched by the deploy process

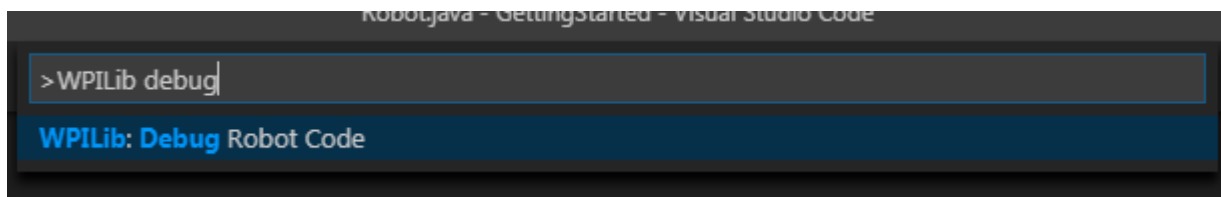
10.7 Debugging a Robot Program

Inevitably, a program will not behave in the way we expect it to behave. When this occurs, it becomes necessary to figure out why the program is doing what it is doing, so that we can make it do what we want it to do, instead. Such an undesired program behavior is called a “bug,” and this process is called “debugging.”

A debugger is a tool used to control program flow and monitor variables in order to assist in debugging a program. This section will describe how to set up a debug session for an FRC® robot program.

Note: For beginning users who need to debug their programs but do not know/have time to learn how to use a debugger, it is often possible to debug a program simply by printing the relevant program state to the console. However, it is strongly recommended that students eventually learn to use a debugger.

10.7.1 Running the Debugger



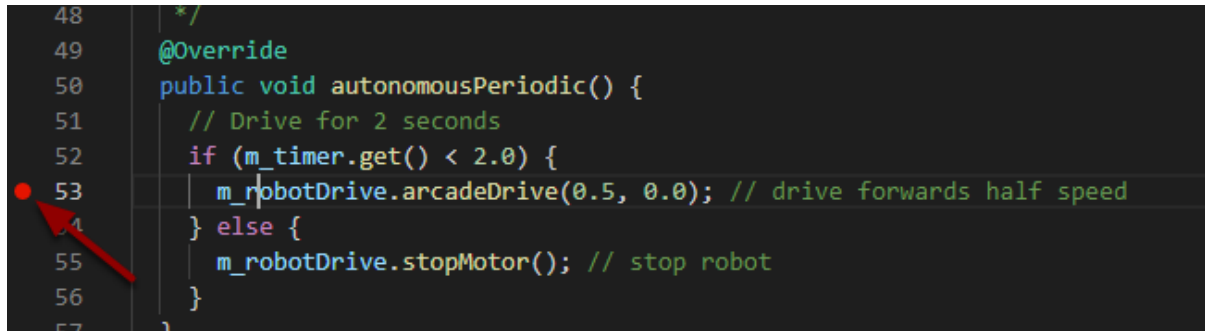
Press **Ctrl+Shift+P** and type **WPILib** or click on the *WPILib Menu Item* to open the Command palette with WPILib pre-populated. Type **Debug** and select the **Debug Robot Code** menu item to start debugging. The code will download to the roboRIO and begin debugging.

10.7.2 Breakpoints

A “breakpoint” is a line of code at which the debugger will pause the program execution so that the user can examine the program state. This is extremely useful while debugging, as it allows the user to pause the program at specific points in problematic code to determine where exactly the program is deviating from the expected behavior.

The debugger will automatically pause at the first breakpoint it encounters.

Setting a Breakpoint



Click in the left margin of the source code window (to the left of the line number) to set a breakpoint in your user program: A small red circle indicates the breakpoint has been set on the corresponding line.

10.7.3 Debugging with Print Statements

Another way to debug your program is to use print statements in your code and view them using the RioLog in Visual Studio Code or the Driver Station. Print statements should be added with care as they are not very efficient especially when used in high quantities. They should be removed for competition as they can cause loop overruns.

Java

C++

```
System.out.print("example");
```

```
wpi::outs() << "example\n";
```

10.7.4 Debugging with NetworkTables

NetworkTables can be used to share robot information with your debugging computer. *NetworkTables* can be viewed with your favorite Dashboard or *OutlineViewer*. One advantage of *NetworkTables* is that tools like *Shuffleboard* can be used to graphically analyze the data. These same tools can then be used with same data to later provide an operator interface for your drivers.

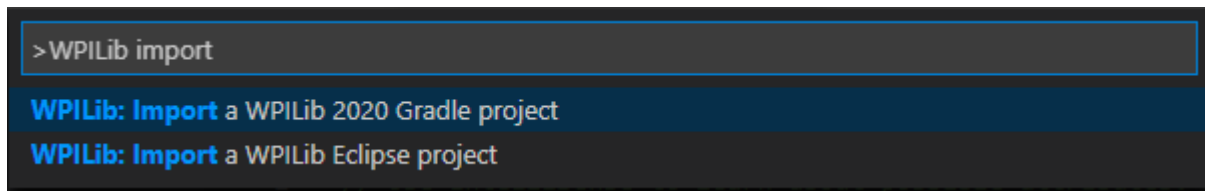
10.7.5 Learn More

- To learn more about debugging with VS Code see this [link](#).
- Some of the features mentioned in this VS Code [article](#) will help you understand and diagnose problems with your code. The Quick Fix (yellow light bulb) feature can be very helpful with a variety of problems including what to import.
- One of the best ways to prevent having to debug so many issues is to do Unit Testing.
- Verifying that your robot works in [Simulation](#) is also a great way to prevent having to do complex debugging on the actual robot.

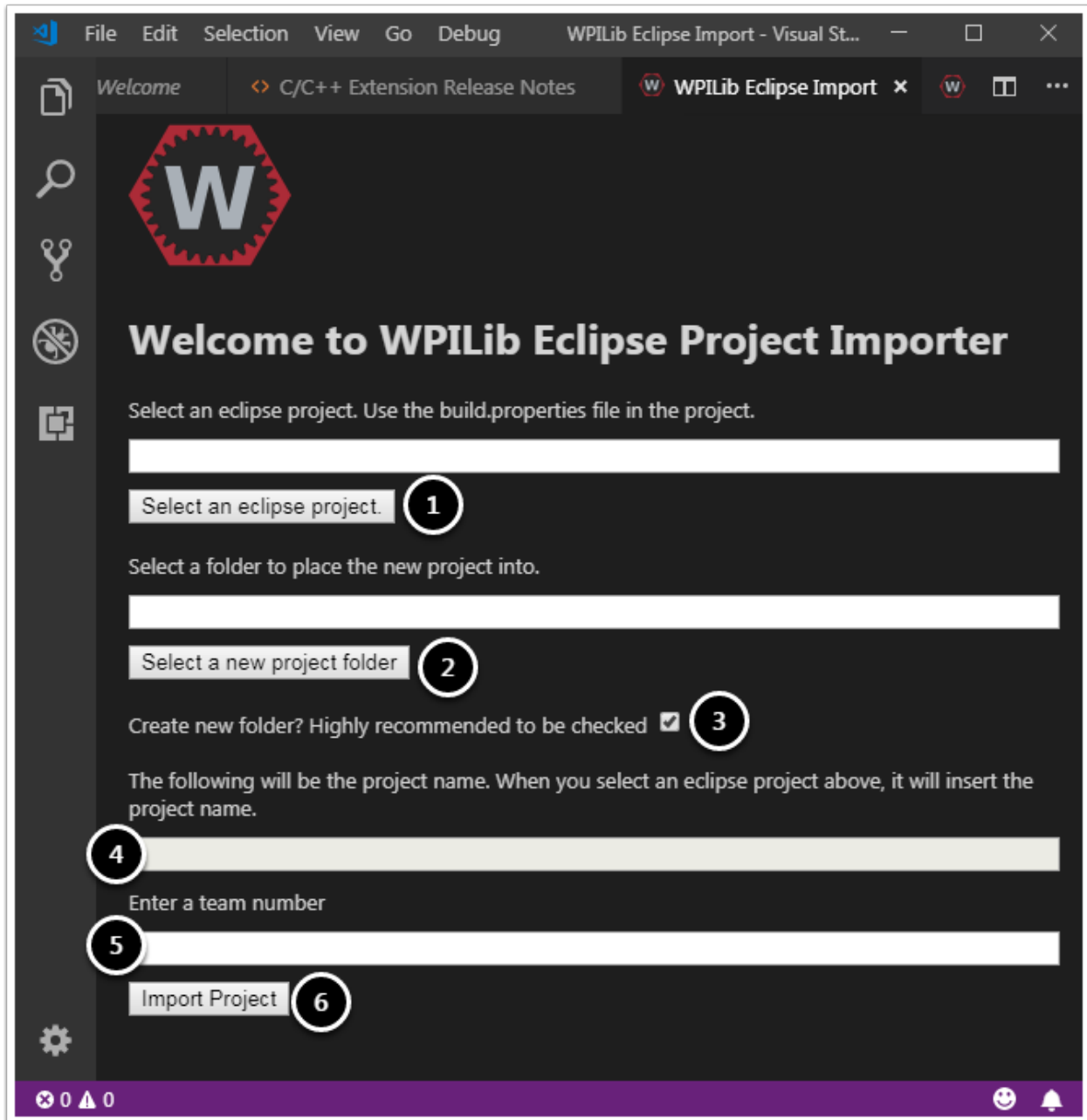
10.8 Importing an Eclipse project into VS Code

To make it easy for teams to use existing projects with the new IDE, WPILib includes a wizard for importing Eclipse projects into VS Code. This will generate the necessary Gradle components and load the project into VS Code. The importer automatically imports the old command framework 3rd party library.

10.8.1 Launching the Import Wizard



Press Ctrl+Shift+P and type “WPILib” or click the WPILib icon to locate the WPILib commands. Begin typing “Import a WPILib Eclipse project” and select it from the dropdown.



You'll be presented with the WPILib Eclipse Project Upgrade window. This is similar to the process of creating a new project and the window and the steps are shown below. This window contains the following elements:

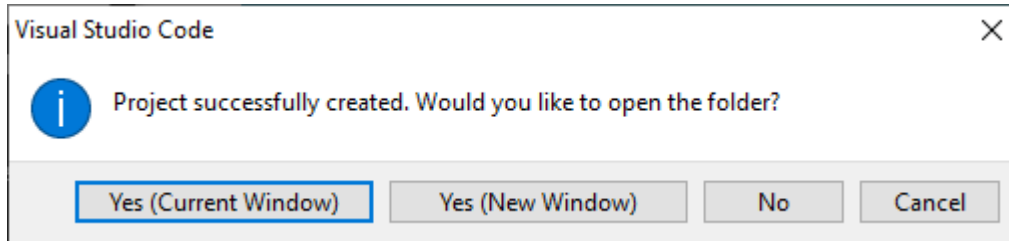
1. **Eclipse Project Selection:** Selects the Eclipse project to be imported. Users should select the `build.properties` file in the root directory of the eclipse project.
2. **Project Location:** This determines the folder in which the robot project will be located.
3. **Create New Folder:** If this is checked, a new folder will be created to hold the project within the previously-specified folder. If it is *not* checked, the project will be located directly in the previously-specified folder. An error will be thrown if the folder is not empty and this is not checked.
4. **Project Name:** The name of the robot project. This also specifies the name that the

project folder will be given if the Create New Folder box is checked.

5. **Team Number:** The team number for the project, which will be used for package names within the project and to locate the robot when deploying code.

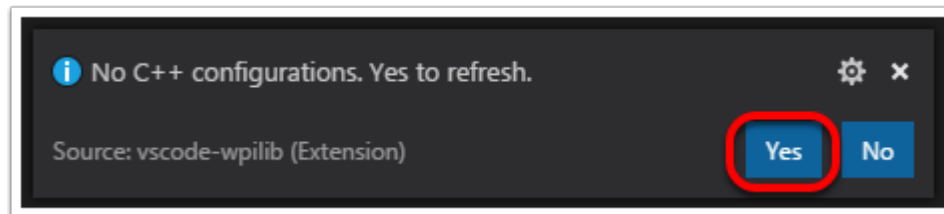
Click “Upgrade Project” to begin the upgrade.

The eclipse project will be upgraded and copied into the new project directory. You can then either open the new project immediately (the pop-up shown below should appear in the bottom right) or open it later using the Ctrl+O (or Command-O for macOS) shortcut.



10.8.2 C++ Configurations (C++ Only)

For C++ projects, there is one more step to set up IntelliSense. Whenever you open a project, you should get a pop-up in the bottom right corner asking to refresh C++ configurations. Click “Yes” to set up IntelliSense.



10.9 Importing a Gradle Project

Due to changes in the project, it is necessary to update the build files for a previous years Gradle project. It is also necessary to import vendor libraries again, since last year’s vendor libraries must be updated to be compatible with this year’s projects.

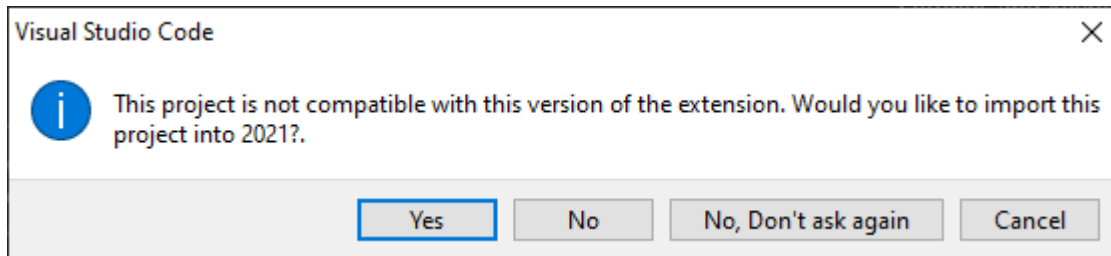
10.9.1 Automatic Import

To make it easy for teams to import previous years gradle projects into the current year’s framework, WPILib includes a wizard for importing previous years projects into VS Code. This will generate the necessary gradle components and load the project into VS Code. In place upgrades are not supported. The importer automatically imports the old or new command framework 3rd party library (which ever one was used on the imported project).

Important: The import process copies your project source files from the current directory to a new directory and completely regenerates the gradle files. If you made non-standard updates to the build.gradle, you will need to make those changes again. For this reason, in

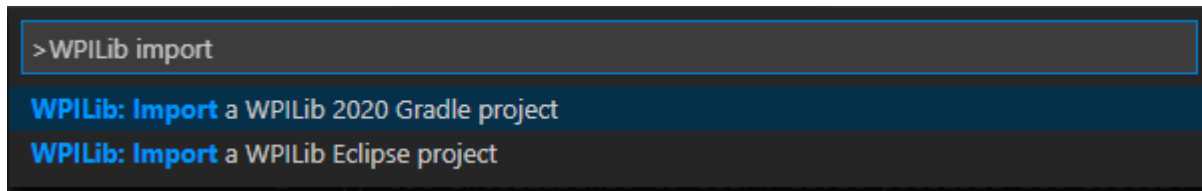
place upgrades are not supported. It is also necessary to import vendor libraries again, since last year's vendor libraries must be updated to be compatible with this year's projects.

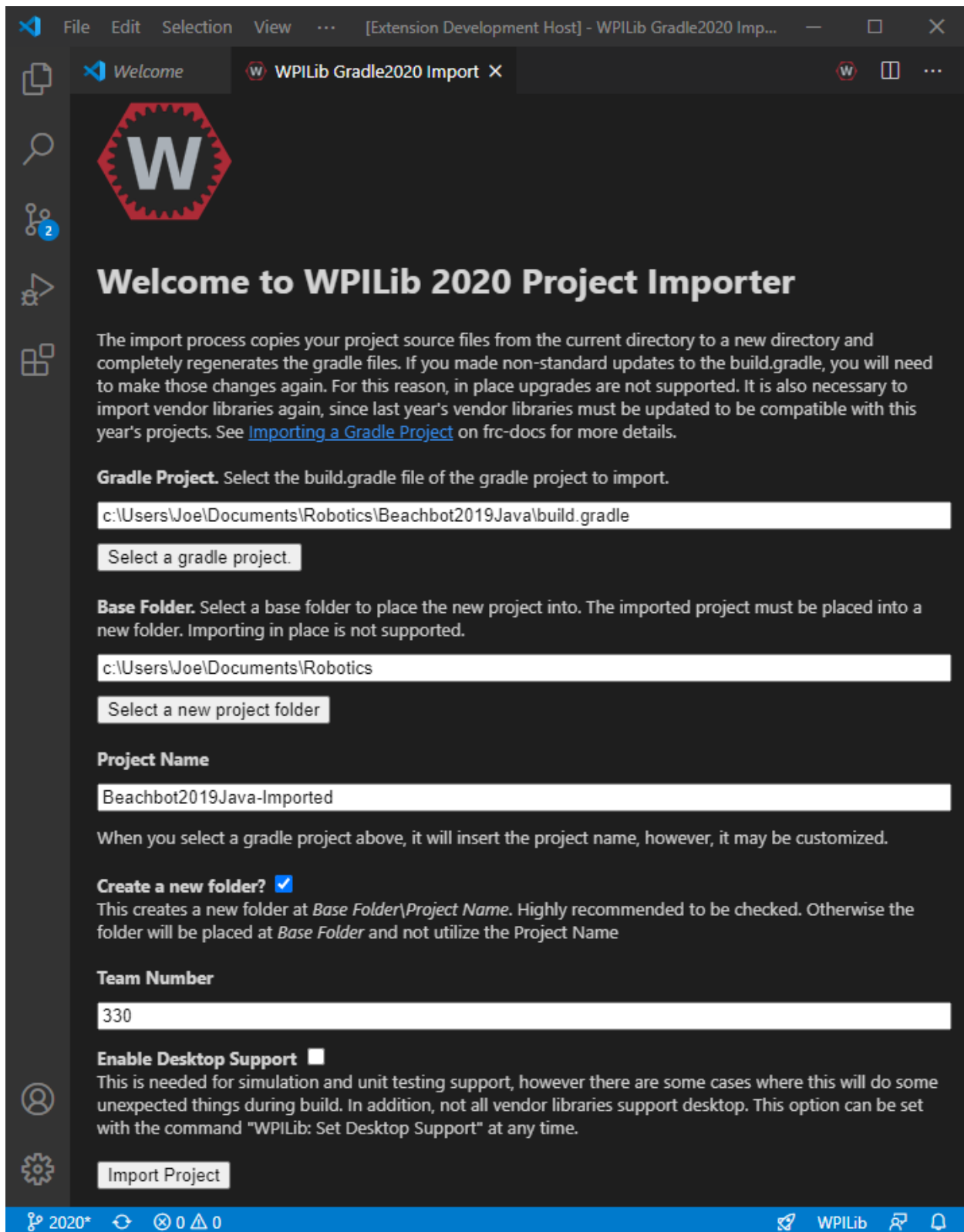
Launching the Import Wizard



When you open a previous year's project, you will be prompted to import that project. Click *yes*.

Alternately, you can chose to import it from the menu. Press `Ctrl+Shift+P` and type "WPILib" or click the WPILib icon to locate the WPILib commands. Begin typing "Import a WPILib 2020 Gradle project" and select it from the dropdown as shown below.



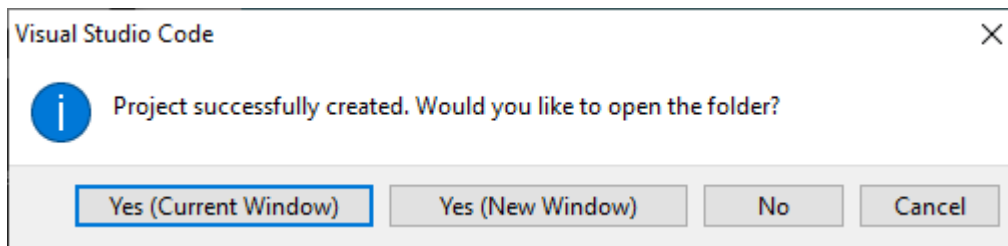


You'll be presented with the WPILib Project Importer window. This is similar to the process of creating a new project and the window and the steps are shown below. This window contains the following elements:

1. **Gradle Project:** Selects the project to be imported. Users should select the build.gradle file in the root directory of the gradle project.
2. **Project Location:** This determines the folder in which the robot project will be located.
3. **Project Name:** The name of the robot project. This also specifies the name that the project folder will be given if the Create New Folder box is checked. This must be a different directory from the original location.
4. **Create a New Folder:** If this is checked, a new folder will be created to hold the project within the previously-specified folder. If it is *not* checked, the project will be located directly in the previously-specified folder. An error will be thrown if the folder is not empty and this is not checked.
5. **Team Number:** The team number for the project, which will be used for package names within the project and to locate the robot when deploying code.
6. **Enable Desktop Support:** If this is checked, simulation and unit test support is enabled. However, there are some cases where this will do some unexpected things. In addition, all vendor libraries need desktop support which not all libraries do.

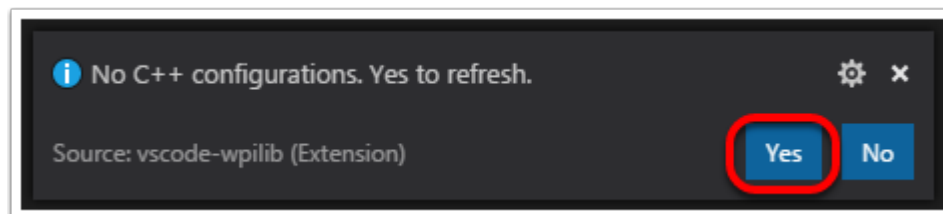
Click *Import Project* to begin the upgrade.

The gradle project will be upgraded and copied into the new project directory. You can then either open the new project immediately (the pop-up shown below should appear in the bottom right) or open it later using the Ctrl+O (or Command+O for macOS) shortcut.



C++ Configurations (C++ Only)

For C++ projects, there is one more step to set up IntelliSense. Whenever you open a project, you should get a pop-up in the bottom right corner asking to refresh C++ configurations. Click Yes to set up IntelliSense.



3rd Party Libraries

It is necessary to update and re-import 3rd party libraries. See *3rd Party Libraries* for details.

10.9.2 Manual Import Process (2020 to 2021)

As there were minimal changes to the gradle templates in 2021, it is possible to manually import a project from a 2020 version of GradleRIO to 2021, which may be convenient if you've heavily customized your build.gradle file. This method does *not* work for importing a 2019 project, and will not work for future years. There are two files that need to be updated.

wpilib_preferences.json

Open .wpilib\wpilib_preferences.json and change the projectYear to 2021.

```
{
  "enableCppIntellisense": false,
  "currentLanguage": "java",
  "projectYear": "2021",
  "teamNumber": 330
}
```

build.gradle

Java

C++

```
1  plugins {
2      id "java"
3      id "edu.wpi.first.GradleRIO" version "2021.1.2"
4  }
5
6  sourceCompatibility = JavaVersion.VERSION_11
7  targetCompatibility = JavaVersion.VERSION_11
8
9  def ROBOT_MAIN_CLASS = "frc.robot.Main"
10
11  // Define my targets (RoboRIO) and artifacts (deployable files)
12  // This is added by GradleRIO's backing project EmbeddedTools.
13  deploy {
14      targets {
15          roboRIO("roborio") {
16              // Team number is loaded either from the .wpilib/wpilib_preferences.json
17              // or from command line. If not found an exception will be thrown.
18              // You can use getTeamOrDefault(team) instead of getTeamNumber if you
19              // want to store a team number in this file.
20              team = frc.getTeamNumber()
21          }
22      }
23      artifacts {
24          frcJavaArtifact('frcJava') {
25              targets << "roborio"
```

(continues on next page)

(continued from previous page)

```

26         // Debug can be overridden by command line, for use with VSCode
27         debug = frc.getDebugOrDefault(false)
28     }
29     // Built in artifact to deploy arbitrary files to the roboRIO.
30     fileTreeArtifact('frcStaticFileDeploy') {
31         // The directory below is the local directory to deploy
32         files = fileTree(dir: 'src/main/deploy')
33         // Deploy to RoboRIO target, into /home/lvuser/deploy
34         targets << "roborio"
35         directory = '/home/lvuser/deploy'
36     }
37 }
38 }
39
40 // Set this to true to enable desktop support.
41 def includeDesktopSupport = false
42
43 // Defining my dependencies. In this case, WPILib (+ friends), and vendor libraries.
44 // Also defines JUnit 4.
45 dependencies {
46     implementation wpi.deps.wpilib()
47     nativeZip wpi.deps.wpilibJni(wpi.platforms.roborio)
48     nativeDesktopZip wpi.deps.wpilibJni(wpi.platforms.desktop)
49
50     implementation wpi.deps.vendor.java()
51     nativeZip wpi.deps.vendor.jni(wpi.platforms.roborio)
52     nativeDesktopZip wpi.deps.vendor.jni(wpi.platforms.desktop)
53
54     testImplementation 'junit:junit:4.12'
55
56     // Enable simulation gui support. Must check the box in vscode to enable support
57     // upon debugging
58     simulation wpi.deps.sim.gui(wpi.platforms.desktop, false)
59     simulation wpi.deps.sim.driverstation(wpi.platforms.desktop, false)
60
61     // Websocket extensions require additional configuration.
62     // simulation wpi.deps.sim.ws_server(wpi.platforms.desktop, false)
63     // simulation wpi.deps.sim.ws_client(wpi.platforms.desktop, false)
64 }
65
66 // Simulation configuration (e.g. environment variables).
67 sim {
68     // Sets the websocket client remote host.
69     // envVar "HALSIMWS_HOST", "10.0.0.2"
70 }
71
72 // Setting up my Jar File. In this case, adding all libraries into the main jar ('fat_
73 jar')
74 // in order to make them all available at runtime. Also adding the manifest so WPILib
75 // knows where to look for our Robot Class.
76 jar {
77     from { configurations.runtimeClasspath.collect { it.isDirectory() ? it :
78 zipTree(it) } }
79     manifest edu.wpi.first.gradlerio.GradleRIOPlugin.javaManifest(ROBOT_MAIN_CLASS)
80 }

```

```

1 plugins {
2     id "cpp"
3     id "google-test-test-suite"
4     id "edu.wpi.first.GradleRIO" version "2021.1.2"
5 }
6
7 // Define my targets (RoboRIO) and artifacts (deployable files)
8 // This is added by GradleRIO's backing project EmbeddedTools.
9 deploy {
10     targets {
11         roboRIO("roborio") {
12             // Team number is loaded either from the .wpilib/wpilib_preferences.json
13             // or from command line. If not found an exception will be thrown.
14             // You can use getTeamOrDefault(team) instead of getTeamNumber if you
15             // want to store a team number in this file.
16             team = frc.getTeamNumber()
17         }
18     }
19     artifacts {
20         frcNativeArtifact('frcCpp') {
21             targets << "roborio"
22             component = 'frcUserProgram'
23             // Debug can be overridden by command line, for use with VSCode
24             debug = frc.getDebugOrDefault(false)
25         }
26         // Built in artifact to deploy arbitrary files to the roboRIO.
27         fileTreeArtifact('frcStaticFileDeploy') {
28             // The directory below is the local directory to deploy
29             files = fileTree(dir: 'src/main/deploy')
30             // Deploy to RoboRIO target, into /home/lvuser/deploy
31             targets << "roborio"
32             directory = '/home/lvuser/deploy'
33         }
34     }
35 }
36
37 // Set this to true to include the src folder in the include directories passed
38 // to the compiler. Some eclipse project imports depend on this behavior.
39 // We recommend leaving this disabled if possible. Note for eclipse project
40 // imports this is enabled by default. For new projects, its disabled
41 def includeSrcInIncludeRoot = false
42
43 // Set this to true to enable desktop support.
44 def includeDesktopSupport = false
45
46 // Enable simulation gui support. Must check the box in vscode to enable support
47 // upon debugging
48 dependencies {
49     simulation wpi.deps.sim.gui(wpi.platforms.desktop, true)
50     simulation wpi.deps.sim.driverstation(wpi.platforms.desktop, true)
51
52     // Websocket extensions require additional configuration.
53     // simulation wpi.deps.sim.ws_server(wpi.platforms.desktop, true)
54     // simulation wpi.deps.sim.ws_client(wpi.platforms.desktop, true)
55 }
56
57 // Simulation configuration (e.g. environment variables).

```

(continues on next page)

(continued from previous page)

```

58 sim {
59     // Sets the websocket client remote host.
60     // envVar "HALSIMWS_HOST", "10.0.0.2"
61 }
62
63 model {
64     components {
65         frcUserProgram(NativeExecutableSpec) {
66             targetPlatform wpi.platforms.roborio
67             if (includeDesktopSupport) {
68                 targetPlatform wpi.platforms.desktop
69             }
70
71             sources.cpp {
72                 source {
73                     srcDir 'src/main/cpp'
74                     include '**/*.cpp', '**/*.cc'
75                 }
76                 exportedHeaders {
77                     srcDir 'src/main/include'
78                     if (includeSrcInIncludeRoot) {
79                         srcDir 'src/main/cpp'
80                     }
81                 }
82             }
83
84             // Defining my dependencies. In this case, WPILib (+ friends), and vendor.
85             ↪libraries.
86             wpi.deps.vendor.cpp(it)
87             wpi.deps.wpilib(it)
88         }
89     }
90     testSuites {
91         frcUserProgramTest(GoogleTestTestSuiteSpec) {
92             testing $.components.frcUserProgram
93
94             sources.cpp {
95                 source {
96                     srcDir 'src/test/cpp'
97                     include '**/*.cpp'
98                 }
99             }
100             wpi.deps.vendor.cpp(it)
101             wpi.deps.wpilib(it)
102             wpi.deps.googleTest(it)
103         }
104     }
105 }

```

1. Change the GradleRIO version to the latest version (e.g. 2021.1.2 for the kickoff release)
2. (C++ Only) move `wpi.deps.vendor.cpp(it)` above `wpi.deps.wpilib(it)` (2 places)
3. (Optional) add simulation `wpi.deps.sim.driverstation(wpi.platforms.desktop, XXXX)` where XXXX is false for Java and true for C++. This allows the driver station to be used in simulation

4. (Optional) add the websocket and simulation configuration blocks to support websockets simulation (i.e. Romi)

imgui.ini

Delete `imgui.ini` (the Simulator GUI ini file) if it exists. The 2020 file format is not compatible with the 2021 format. The file will be regenerated when the Simulator GUI is run).

Update 3rd Party Libraries

It is necessary to update 3rd party libraries. See [3rd Party Libraries](#) for details.

10.10 Using Test Mode

Test mode is designed to enable programmers to have a place to put code to verify that all systems on the robot are functioning. In each of the robot program templates there is a place to add test code to the robot. If you use the TimedRobot template, or use command-based programming you can easily see the operation of all the motors and sensors through the LiveWindow.

10.10.1 Enabling Test Mode

Test mode on the robot can be enabled from the Driver Station just like autonomous or teleop. When in test mode, the `testInit` method is run once, and the `testPeriodic` method is run once per tick, in addition to `robotPeriodic`, the same as teleop and autonomous control modes. To enable test mode in the Driver Station, select the “Test” button and enable the robot. The test mode code will then run.

10.10.2 Adding Test mode code to your robot code

Adding test mode can be as painless as calling your already written Teleop methods from Test. This will allow you to use the LiveWindow tuning features for classes such as `PIDBase` and `PIDController`, as well as `PIDSubsystems` and `PIDCommands`, to change PID constants and setpoints. Make sure to add your subsystems to SmartDashboard with the `putData(subsystem)` or `PutData(subsystem)` method.

Java

C++

```
public class Robot extends TimedRobot {  
  
    @Override  
    public void robotInit() {  
        SmartDashboard.putData(m_aSubsystem);  
    }  
  
    @Override
```

(continues on next page)

(continued from previous page)

```

    public void testInit() {
        teleopInit();
    }

    @Override
    public void testPeriodic() {
        teleopPeriodic();
    }
}

void Robot::RobotInit() {
    SmartDashboard::PutData(m_aSubsystem);
}

void Robot::TestInit() {
    TestInit();
}

void Robot::TestPeriodic() {
    TeleopPeriodic();
}

```

10.10.3 Test Mode in SmartDashboard

The above sample code produces the following output when the Driver Station is put into Test mode then enabled. You can operate the motors by moving the sliders and read the values of sensors such as the wrist potentiometer.

Notice that the values are grouped by the subsystem names to group related actuators and sensors for easy testing. The subsystem names are specified by supplying a name to the `putData()` method, or by calling `SendableRegistry.setName()`. This grouping, while not required, makes it much easier to test one subsystem at a time and have all the values next to each other on the screen.

Using Test Mode with the TimedRobot Template

The `TimedRobot` template lends itself quite nicely to testing since it will periodically call the `testPeriodic()` method (or `TestPeriodic()` in C++) in your robot program. The `testPeriodic()` method will be called every 20ms and it is a good place to test commands or have `LiveWindow` update. The `LiveWindow` updating is built into the `TimedRobot` template so there is very little work to use `LiveWindow`.

Note: This works even if you are using the `TimedRobot` template and not doing Command-based programming.

In this example the sensors are registered with the `LiveWindow` and during the `testPeriodic()` method, simply update all the values by calling the `LiveWindow run()` method. If your program is causing too much network traffic you can call the `run` method less frequently by, for example, only calling it every 5 updates for a 100ms update rate.

10.10.4 PID Tuning in Test Mode

Tuning PID loops is often a challenging prospect with frequent recompiles of the program to get the correct values. When using the command based programming model, subclassing `PIDSubsystem` or `PIDCommand` in your PID commands allows the adjustment of PID constants with immediate feedback of the results.

FRC LabVIEW Programming

11.1 Creating Robot Programs

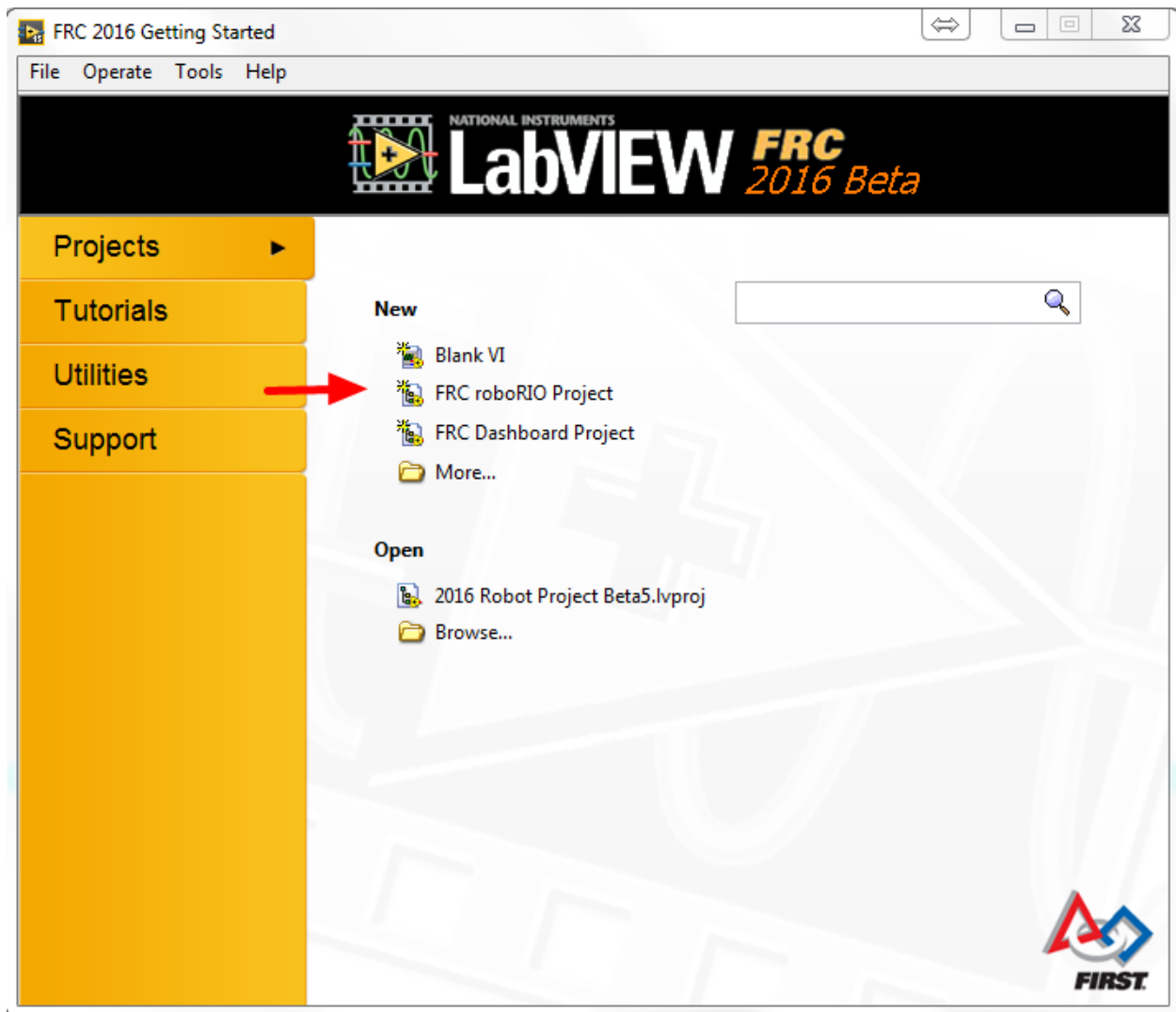
11.1.1 Creating, Building and Loading your Benchtop Test Program



Note: This document covers how to create, build and load an FRC® LabVIEW program onto a roboRIO. Before beginning, make sure that you have installed LabVIEW for FRC and the FRC Driver Station and that you have configured and imaged your roboRIO as described in the [Zero-to-Robot tutorial](#).

Creating a Project

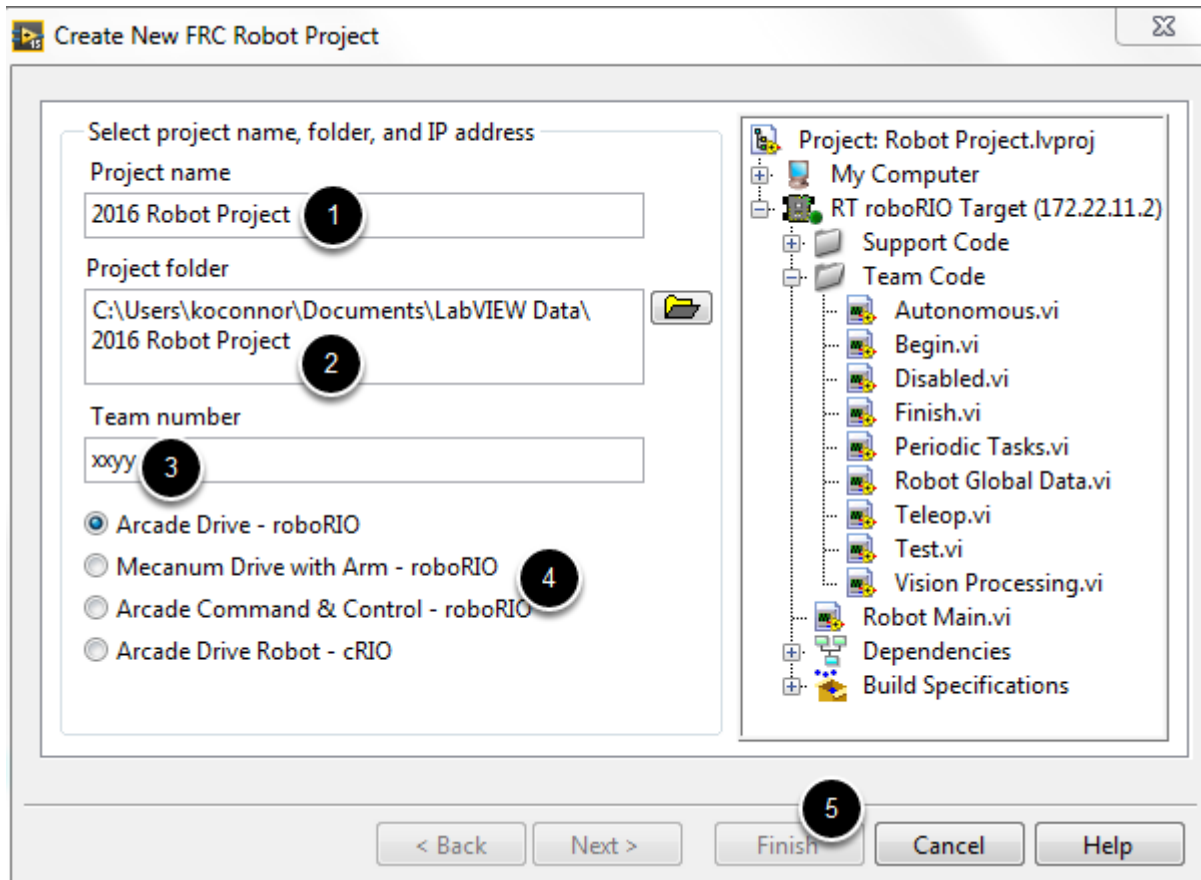
Launch LabVIEW and click the FRC roboRIO Robot Project link in the Projects window to display the Create New FRC Robot Project dialog box.



Configuring Project

Fill in the Create New FRC Project Dialog:

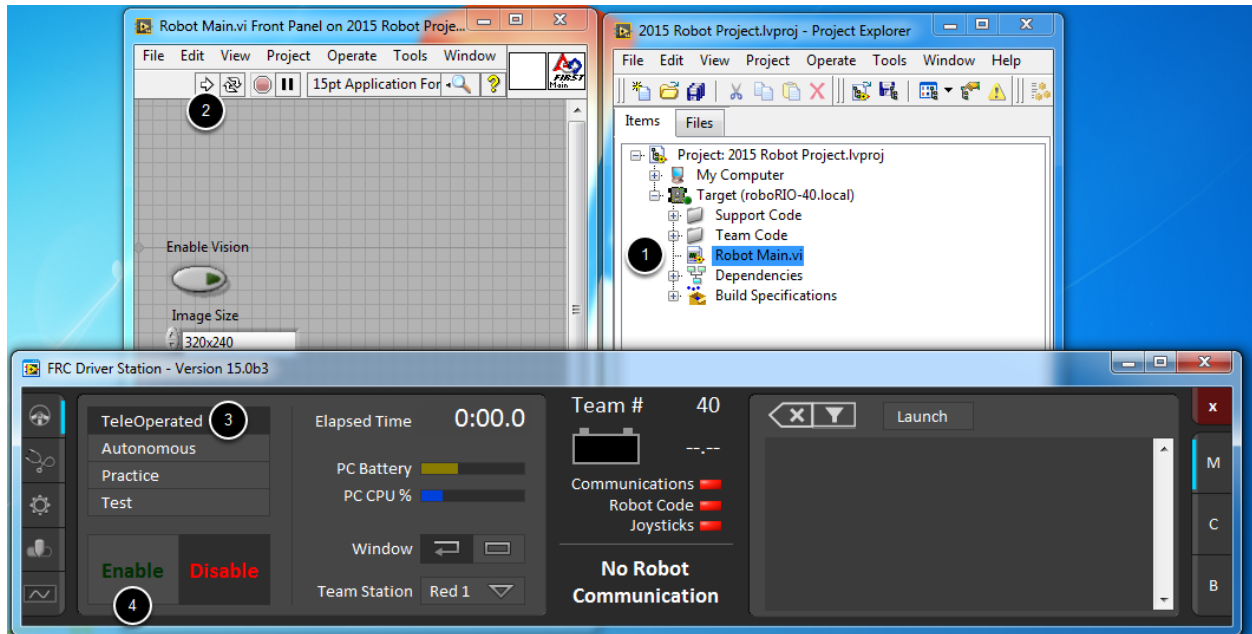
1. Pick a name for your project
2. Select a folder to place the project in.
3. Enter your team number
4. Select a project type. If unsure, select Arcade Drive - roboRIO.
5. Click Finish



Running the Program

Note: Note that a program deployed in this manner will not remain on the roboRIO after a power cycle. To deploy a program to run every time the roboRIO starts follow the next step, Deploying the program.

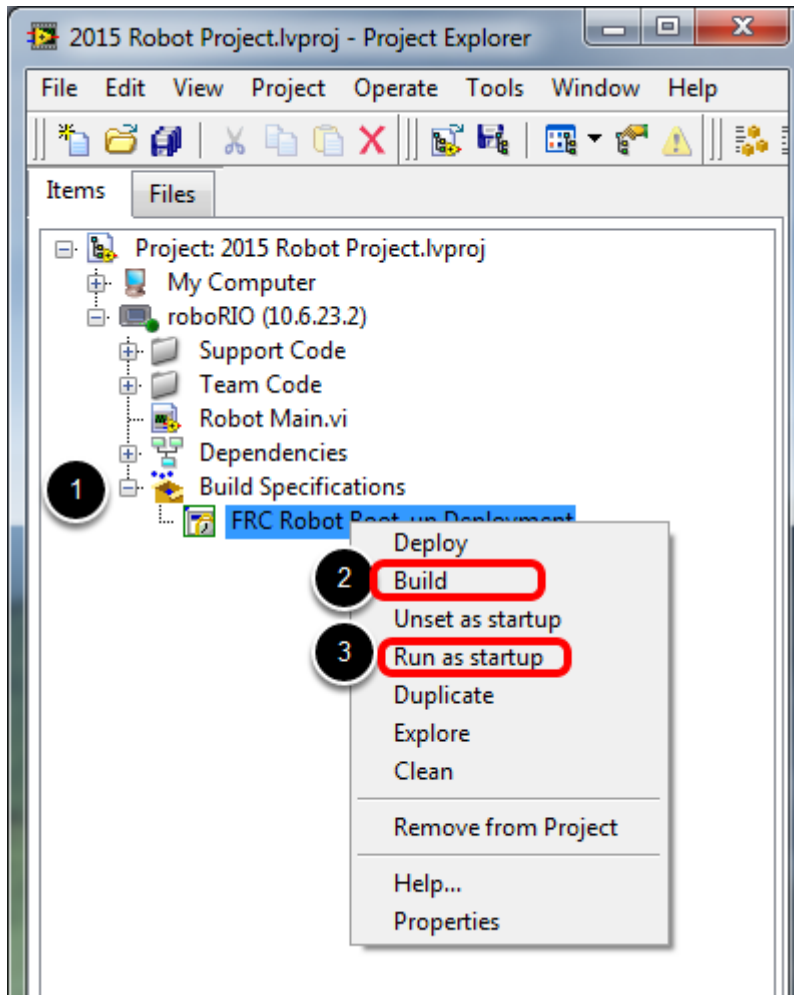
1. In the Project Explorer window, double-click the Robot Main.vi item to open the Robot Main VI.
2. Click the Run button (White Arrow on the top ribbon) of the Robot Main VI to deploy the VI to the roboRIO. LabVIEW deploys the VI, all items required by the VI, and the target settings to memory on the roboRIO. If prompted to save any VIs, click Save on all prompts.
3. Using the Driver Station software, put the robot in Teleop Mode. For more information on configuring and using the Driver Station software, see the FRC Driver Station Software article.
4. Click Enable.
5. Move the joysticks and observe how the robot responds.
6. Click the Abort button of the Robot Main VI. Notice that the VI stops. When you deploy a program with the Run button, the program runs on the roboRIO, but you can manipulate the front panel objects of the program from the host computer.



Deploying the Program

To run in the competition, you will need to deploy a program to your roboRIO. This allows the program to survive across reboots of the controller, but doesn't allow the same debugging features (front panel, probes, highlight execution) as running from the front panel. To deploy your program:

1. In the Project Explorer, click the + next to Build Specifications to expand it.
2. Right-click on FRC Robot Boot-up Deployment and select Build. Wait for the build to complete.
3. Right-click again on FRC Robot Boot-Up Deployment and select Run as Startup. If you receive a conflict dialog, click OK. This dialog simply indicates that there is currently a program on the roboRIO which will be terminated/replaced.
4. Either check the box to close the deployment window on successful completion or click the close button when the deployment completes.
5. The roboRIO will automatically start running the deployed code within a few seconds of the dialog closing.



11.1.2 Tank Drive Tutorial

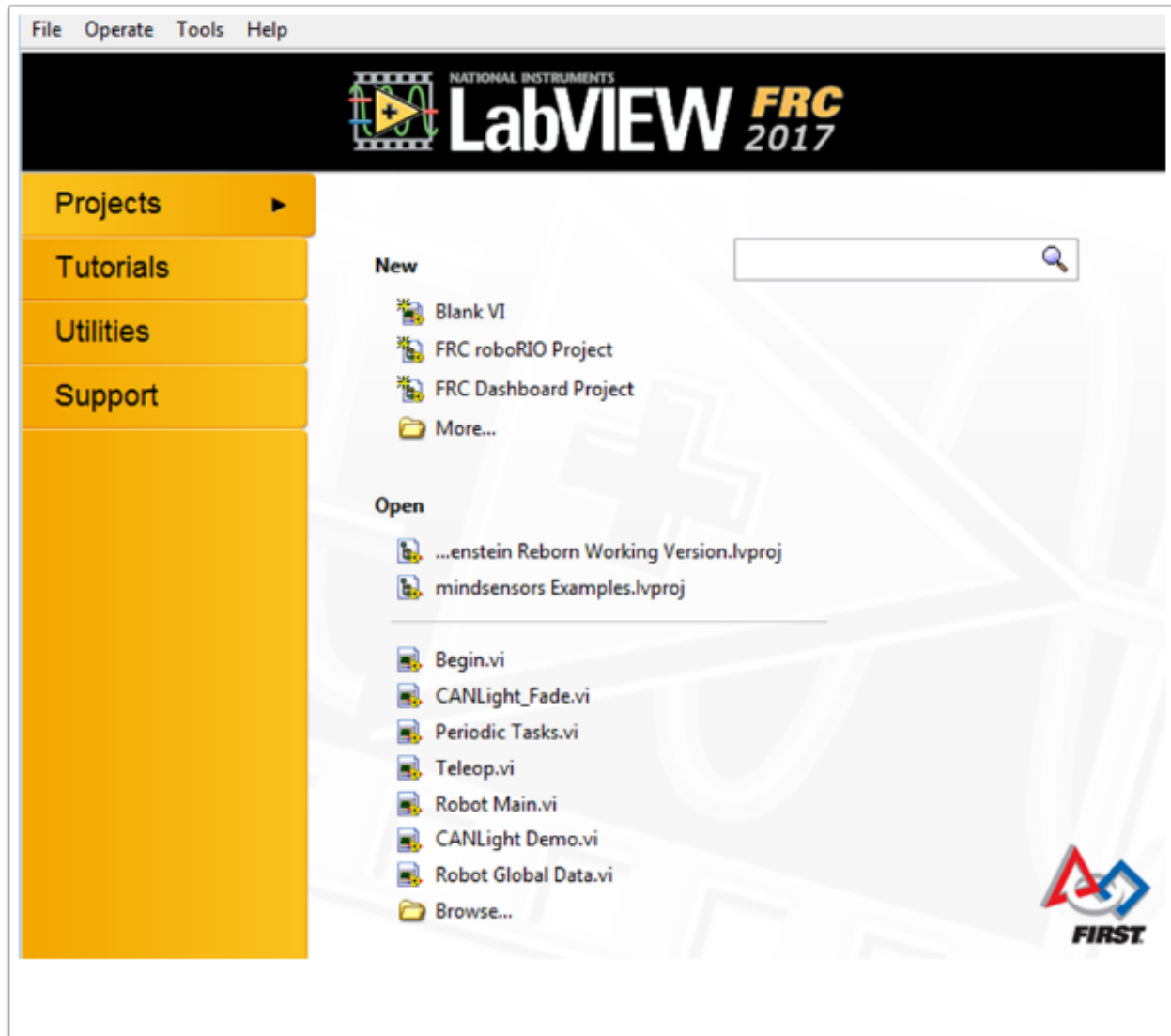


Question: How do I get my robot to drive with two joysticks using tank drive?

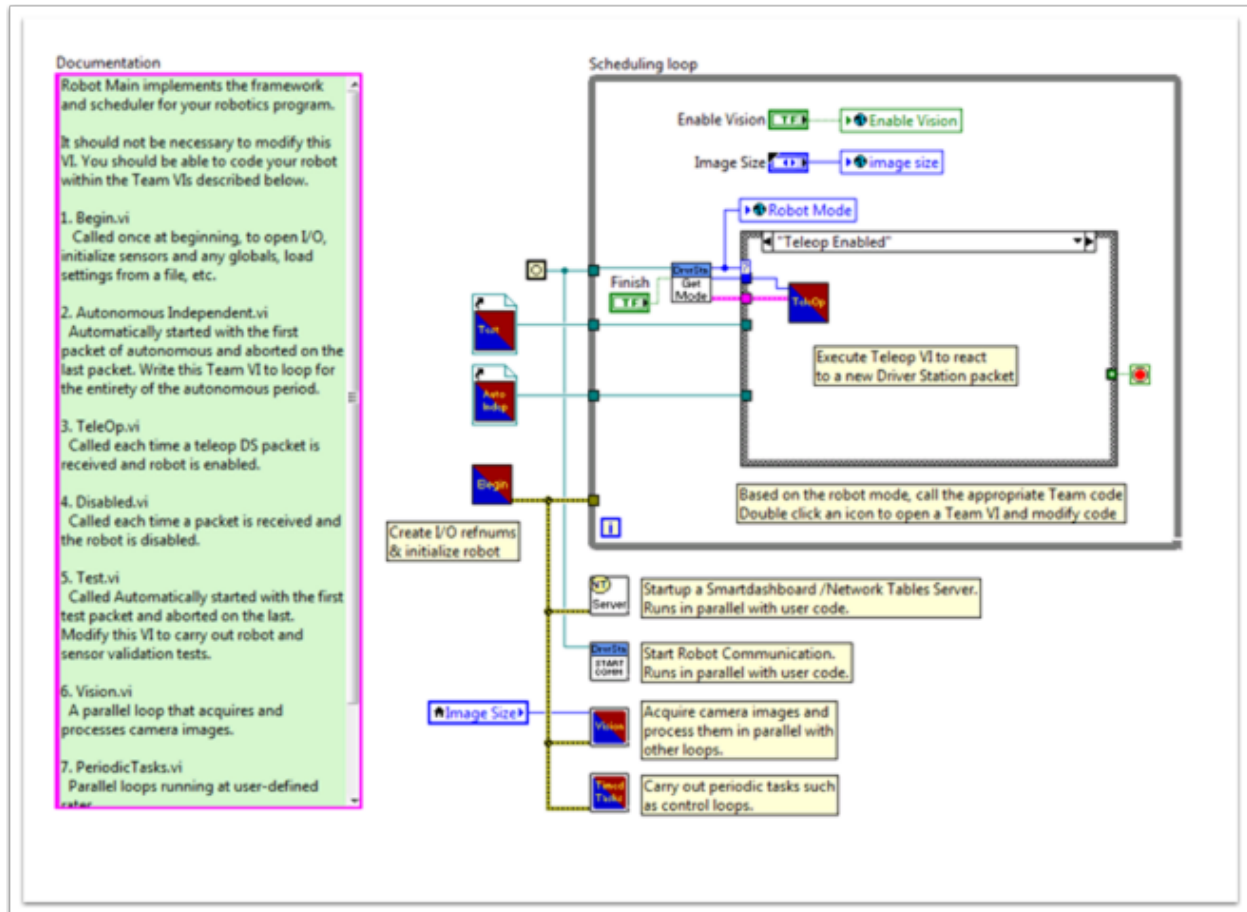
Solution: There are four components to consider when setting up tank drive for your robot. The first thing you will want to do is make sure the tank drive.vi is used instead of the arcade drive.vi or whichever drive VI you were utilizing previously. The second item to consider is how you want your joysticks to map to the direction you want to drive. In tank drive, the left joystick is used to control the left motors and the right joystick is used to control the

right motors. For example, if you want to make your robot turn right by pushing up on the left joystick and down on the right joystick you will need to set your joystick's accordingly in LabVIEW (this is shown in more detail below). Next, you will want to confirm the PWM lines that you are wired into, are the same ones your joysticks will be controlling. Lastly, make sure your motor controllers match the motor controllers specified in LabVIEW. The steps below will discuss these ideas in more detail:

1. Open LabVIEW and double click FRC roboRIO Project.

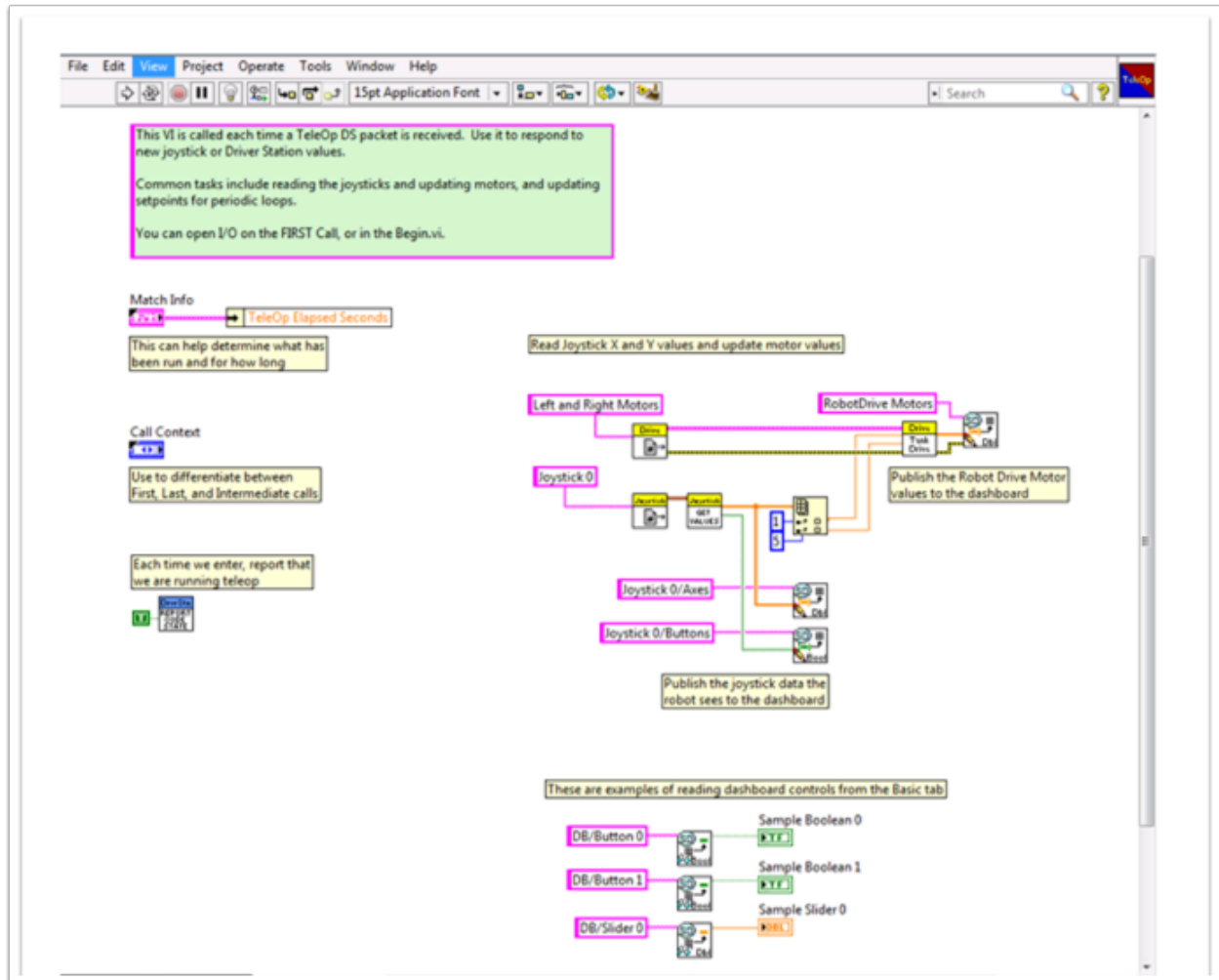


2. Give your project a name, add your team number, and select Arcade Drive Robot roboRIO. You can select another option, however, this tutorial will discuss how to setup tank drive for this project.
3. In the Project Explorer window, open up the Robot Main.vi.
4. Push Ctrl+E to see the block diagram. It should look like the following image:



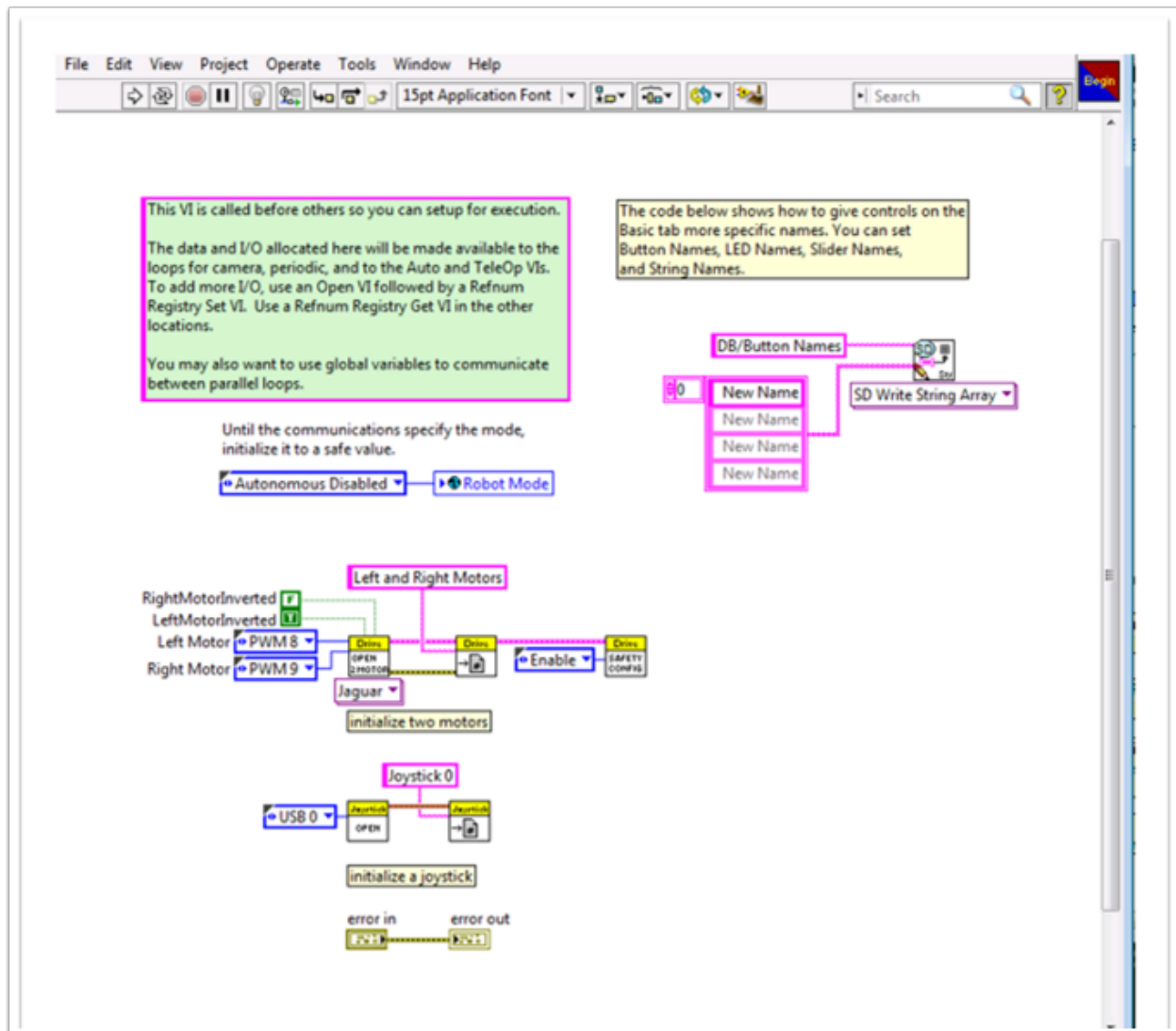
5. Double click the “Teleop” vi inside of the Teleop Enabled case structure. Look at its block diagram. You will want to make two changes here:

- Replace Arcade Drive with the tank drive.vi. This can be found by right clicking on the block diagram >> WPI Robotics Library >> Robot Drive >> and clicking the Tank Drive VI.
- Find the Index Array function that is after the Get Values.vi. You will need to create two numeric constants and wire each into one of the index inputs. You can determine what the values of each index should be by looking at the USB Devices tab in the FRC® Driver Station. Move the two joysticks to determine which number (index) they are tied to. You will likely want to use the Y-axis index for each joystick. This is because it is intuitive to push up on the joystick when you want the motors to go forward, and down when you when them to go in reverse. If you select the X-axis index for each, then you will have to move the joystick left or right (x-axis directions) to get the robot motors to move. In my setup, I’ve selected index 1 for my left motors Y-axis control and index 5 as the right motors Y-axis control. You can see the adjustments in LabVIEW in the following image:



6. Next you will want to go back to your “Robot Main.vi” and double click on the “Begin.vi.”
7. The first thing to confirm in this VI is that your left and right motors are connected to the same PWM lines in LabVIEW as they are on your PDP (Power Distribution Panel).
8. The second thing to confirm in this VI is that the “Open 2 Motor.vi” has the correct motor controller selected (Talon, Jaguar, Victor, etc.).

For example, I am using Jaguar motor controllers and my motors are wired into PWM 8 and 9. The image below shows the changes I need to make:



9. Save all of the VIs that you have made adjustments to and you are now able to drive a robot with tank drive!

11.1.3 Command and Control Tutorial



Introduction

Command and Control is a new LabVIEW template added for the 2016 season which organizes robot code into commands and controllers for a collection of robot-specific subsystems. Each subsystem has an independent control loop or state machine running at the appropriate rate for the mechanism and high-level commands that update desired operations and set points. This makes it very easy for autonomous code to build synchronous sequences of commands. Meanwhile, TeleOp benefits because it can use the same commands without needing to wait for completion, allowing for easy cancellation and initiation of new commands according to the drive team input. Each subsystem has a panel displaying its sensor and control values over time, and command tracing to aid in debugging.

What is Command and Control?

Command and Control recognizes that FRC® robots tend to be built up of relatively independent mechanisms such as Drive, Shooter, Arm, etc. Each of these is referred to as a subsystem and needs code that will coordinate the various sensors and actuators of the subsystem in order to complete requested commands, or actions, such as “Close Gripper” or “Lower Arm”. One of the key principles of this framework is that subsystems will each have an independent controller loop that is solely responsible for updating motors and other actuators. Code outside of the subsystem controller can issue commands which may change the robot’s output, but should not directly change any outputs. The difference is very subtle but this means that outputs can only possibly be updated from one location in the project. This speeds up debugging a robot behaving unexpectedly by giving you the ability to look through a list of commands sent to the subsystem rather than searching your project for where an output may have been modified. It also becomes easier to add an additional sensor, change gearing, or disable a mechanism without needing to modify code outside of the controller.

Game code, primarily consisting of Autonomous and TeleOp, will typically need to update set points and react to the state of certain mechanisms. For Autonomous, it is very common to define the robot’s operation as a sequence of operations – drive here, pick that up, carry it there, shoot it, etc. Commands can be wired sequentially with additional logic to quickly build complex routines. For teleOp, the same commands can execute asynchronously, allowing the robot to always process the latest driver inputs, and if implemented properly, new commands will interrupt, allowing the drive team to quickly respond to field conditions while also taking advantage of automated commands and command sequences.

Why should I use Command and Control?

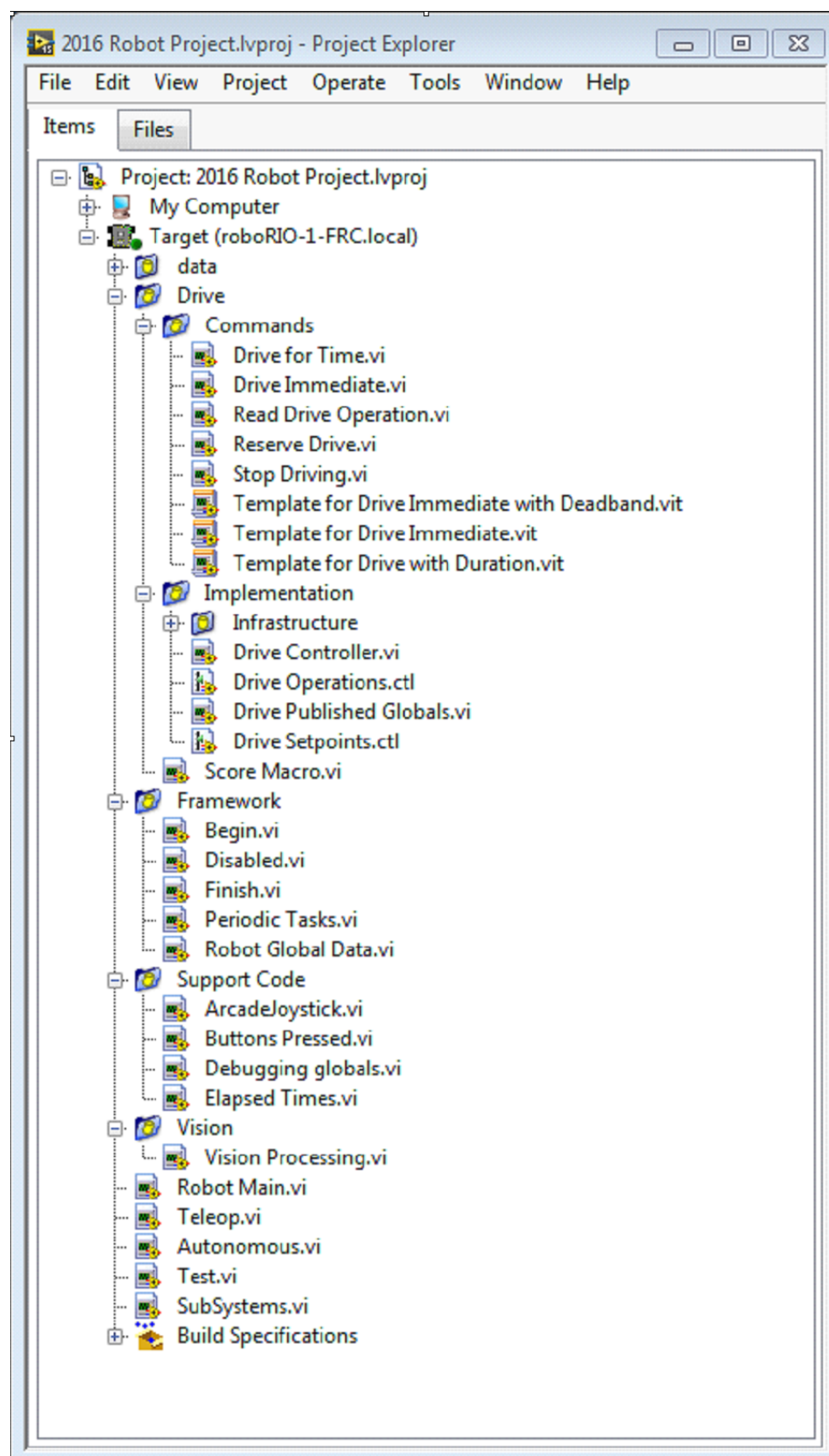
Command and Control adds functionality to the existing LabVIEW project templates, allowing code to scale better with more sophisticated robots and robot code. Subsystems are used to abstract the details of the implementation, and game code is built from sequences of high level command VIs. The commands themselves are VIs that can update set points, perform numerical scaling/mapping between engineering units and mechanism units, and offer synchronization options. If physical changes are made to the robot, such as changing a gearing ratio, changes can be made to just a few command VIs to reflect this change across the entire code base.

I/O encapsulation makes for more predictable operation and quicker debugging when resource conflicts do occur. Because each command is a VI, you are able to single step through commands or use the built in Trace functionality to view a list of all commands sent to each subsystem. The framework uses asynchronous notification and consistent data propagation

making it easy to program a sequence of commands or add in simple logic to determine the correct command to run.

Part 1: Project Explorer

The Project Explorer provides organization for all of the Vis and files you will use for your robot system. Below is a description of the major components in the Project Explorer to help with the expansion of our system. The most frequently used items have been marked in bold.



My Computer The items that define operation on the computer that the project was loaded on. For a robot project, this is used as a simulation target and is populated with simulation files.

Sim Support Files The folder containing 3D CAD models and description files for the simulated robot.

Robot Simulation Readme.html Documents the PWM channels and robot info you will need in order to write robot code that matches the wiring of the simulated robot.

Dependencies Shows the files used by the simulated robot's code. This will populate when you designate the code for the simulated robot target.

Build Specifications This will contain the files that define how to build and deploy code for the simulated robot target.

Target (roboRIO-TEAM-FRC.local) The items that define operation on the roboRIO located at (address).

Drive The subsystem implementation and commands for the robot drive base. This serves as a custom replacement for the WPILib RobotDrive VIs.

Framework VIs used for robot code that is not part of a subsystem that are not used very often.

Begin Called once when robot code first starts. This is useful for initialization code that doesn't belong to a particular subsystem.

Disabled Called once for each disabled packet and can be used to debug sensors when you don't want the robot to move.

Finish During development, this may be called when robot code finishes. Not called on abort or when power is turned off.

Periodic Tasks A good place for ad hoc periodic loops for debugging or monitoring

Robot Global Data Useful for sharing robot information that doesn't belong to a subsystem.

Support Code Debugging and code development aids.

Vision Subsystem and commands for the camera and image processing.

Robot Main.vi Top level VI that you will run while developing code.

Autonomous.vi VI that runs during autonomous period.

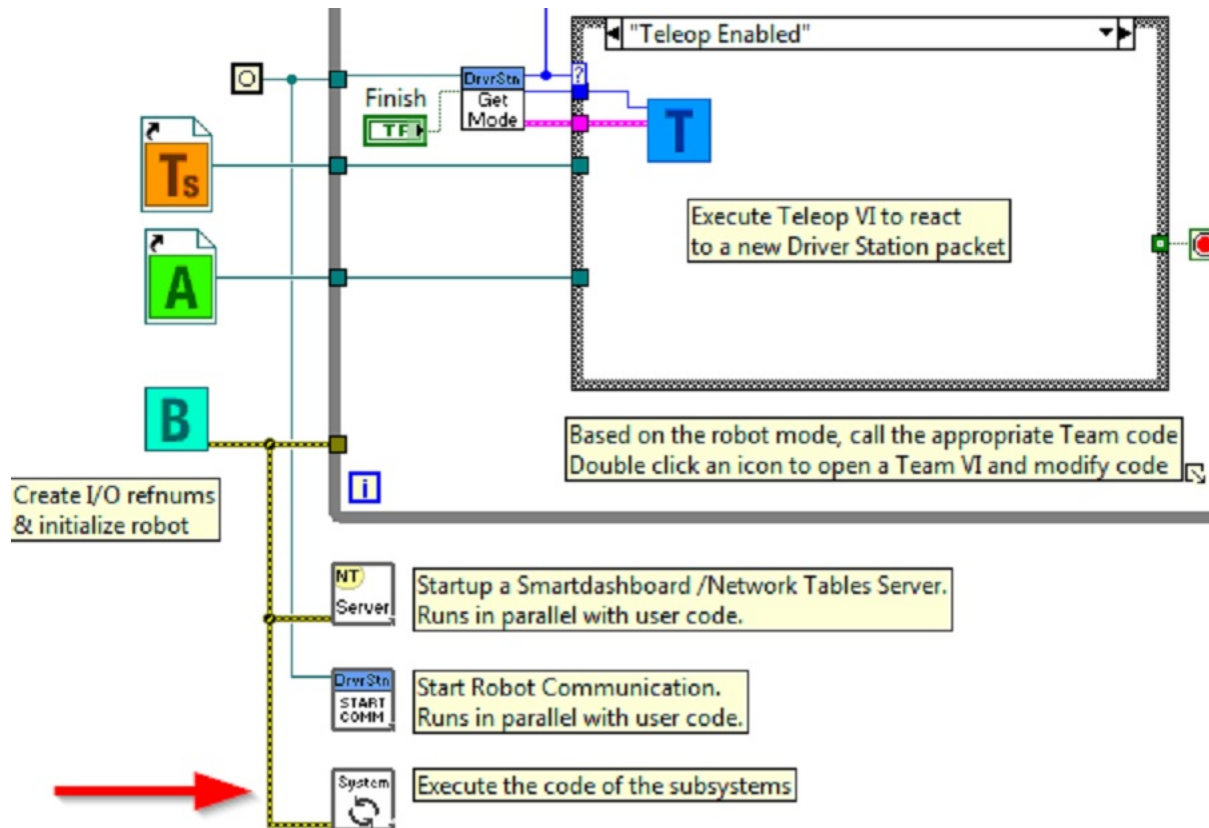
Teleop.vi VI that is called for each TeleOp packet.

Test.vi VI that runs when driver station is in test mode.

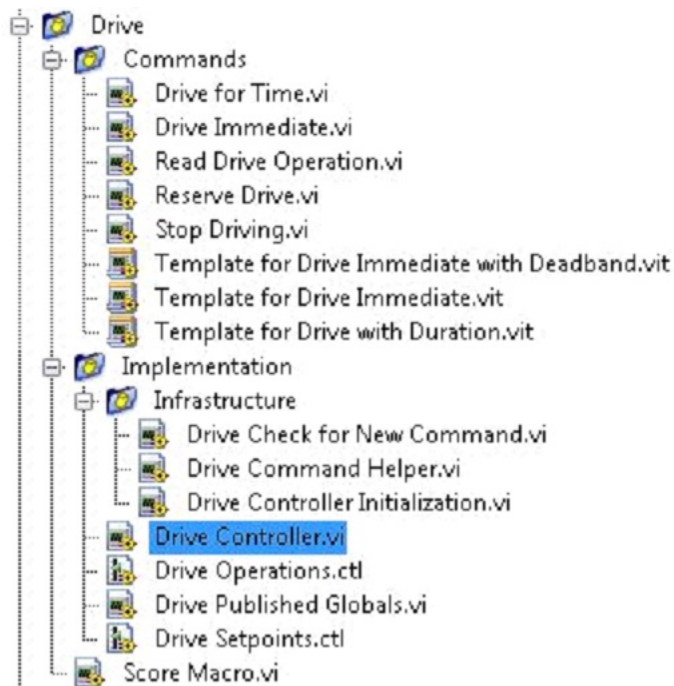
SubSystems.vi VI that contains and starts all subsystems.

Dependencies Shows the files used by the robot code.

Build Specifications Used to build and run the code as a startup application once code works correctly.



Drive Subsystem Project Explorer



Commands: This folder contains the command VIs that request the controller carry out an

operation. It also contains templates for creating additional drive commands.

Note: After creating a new command, you may need to edit `Drive_Setpoints.ctl` to add or update fields that controller uses to define the new operation. You also need to go into the `Drive Controller.vi` and modify the case structure to add a case for every value.

Implementation

These are the VIs and Controls used to build the subsystem.

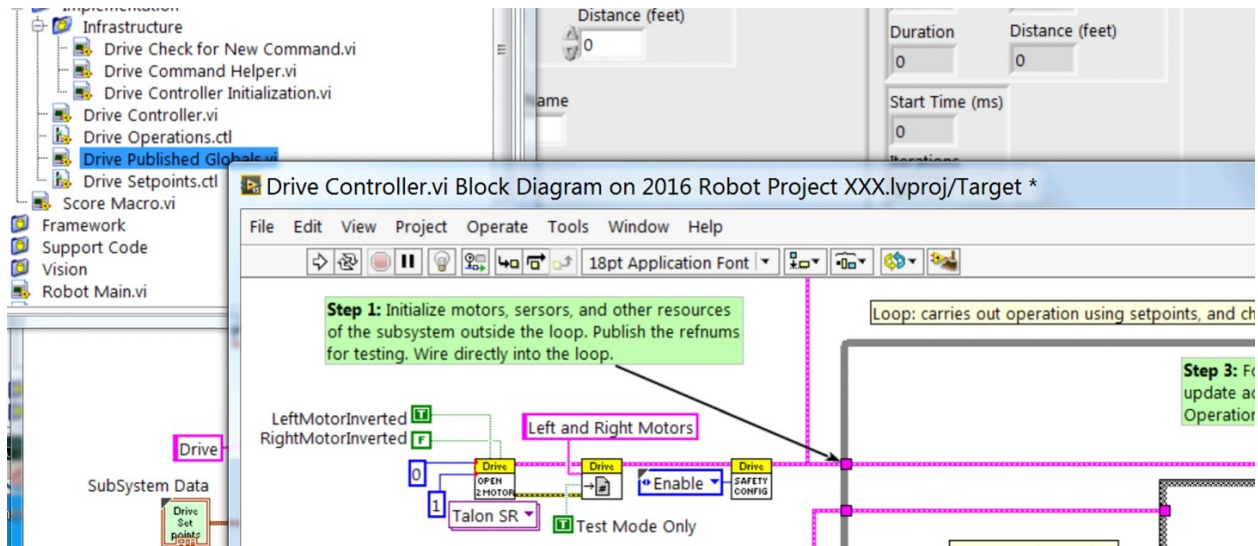
Infrastructure VIs

- **Drive Check for New Command:** It is called each iteration of the controller loop. It checks for new commands, updates timing data, and upon completion notifies a waiting command.
- **Drive Command Helper.vi:** Commands call this VI to notify the controller that a new command has been issued.
- **Drive Controller Initialization.vi:** It allocates the notifier and combines the timing, default command, and other information into a single data wire.
- **Drive Controller.vi:** This VI contains the control/state machine loop. The panel may also contain displays useful for debugging.
- **Drive Operation.ctl:** This typedef defines the operational modes of the controller. Many commands can share an operation.
- **Drive Setpoint.ctl:** It contains the data fields used by all operating modes of the Drive subsystem.
- **Drive Published Globals.vi:** A useful place for publishing global information about the drive subsystem.

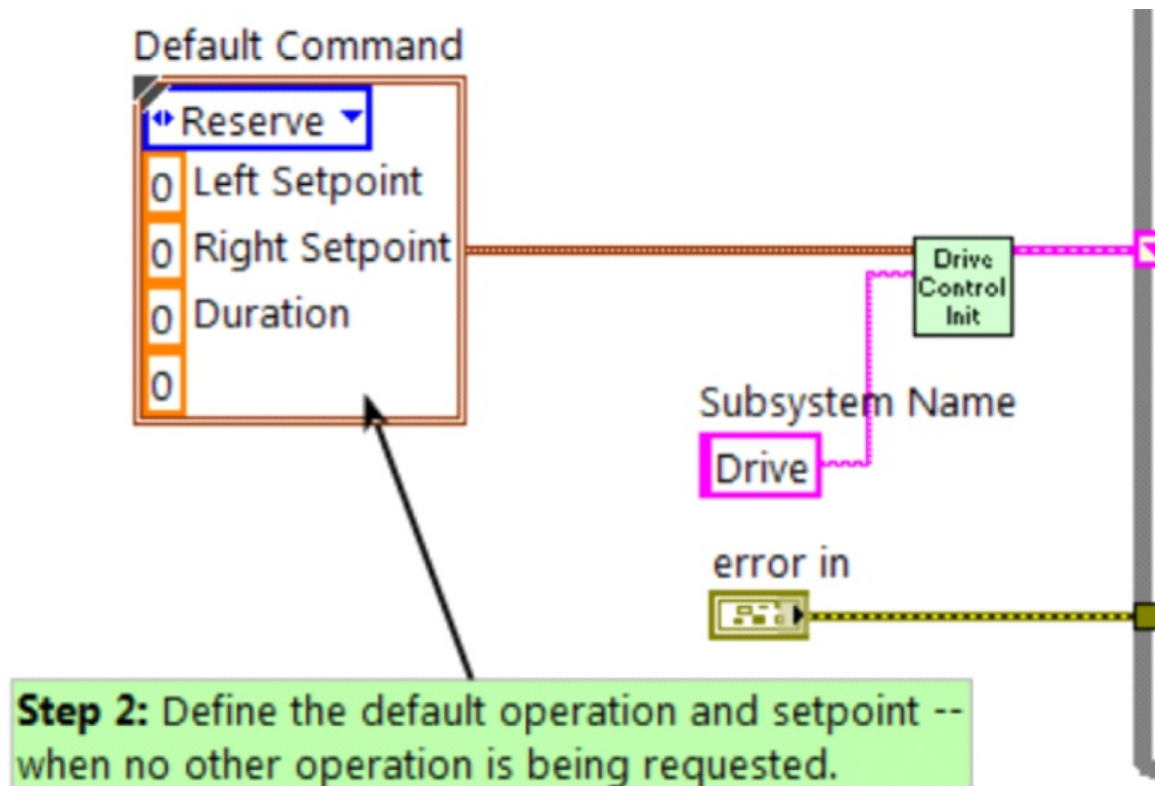
Part 2: Initializing the Drive Subsystem

There are green comments on the controller's block diagram that point out key areas that you will want to know how to edit.

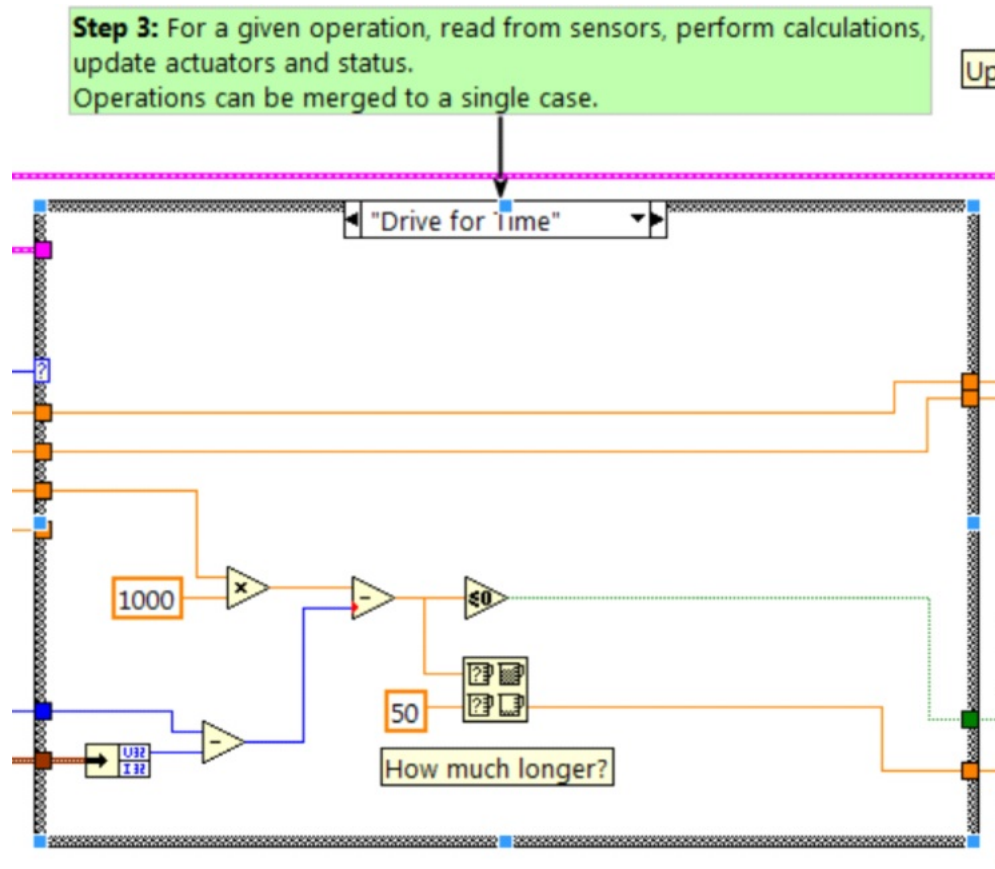
The area to the left of the control loop will execute once when the subsystem starts up. This is where you will typically allocate and initialize all I/O and state data. You may publish the I/O refs, or you may register them for Test Mode Only to keep them private so that other code cannot update motors without using a command.



Note: Initializing the resources for each subsystem in their respective Controller.vi rather than in Begin.vi improves I/O encapsulation, reducing potential resource conflicts and simplifies debugging.

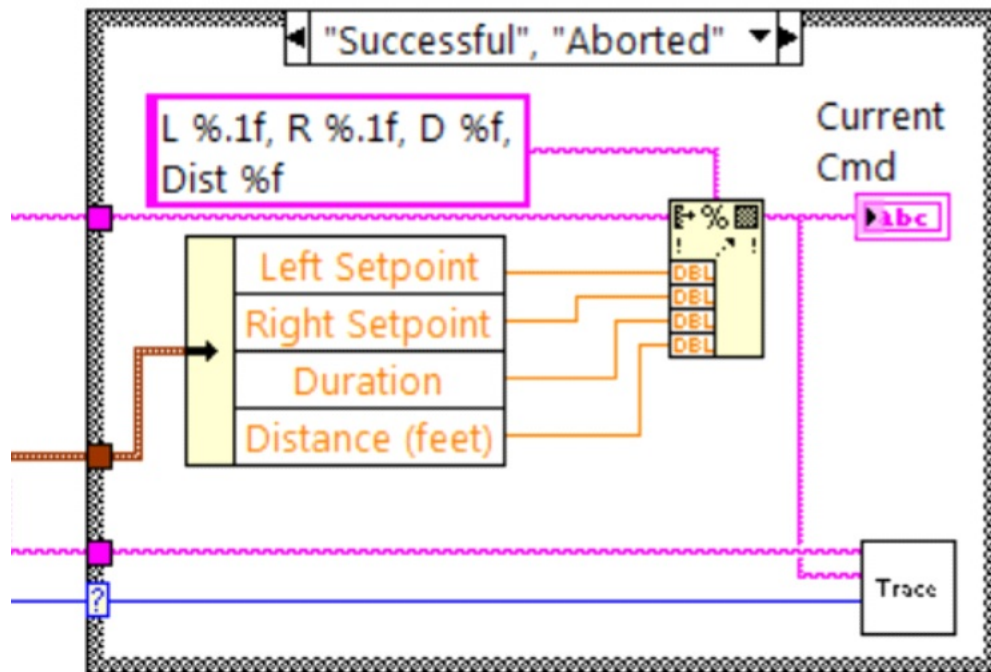


Part of the initialization is to select the default operation and set point values when no other operation is being processed.



Inside the control loop is a case statement where operations are actually implemented. Set point values, iteration delay, iteration count, and sensors can all have influence on how the subsystem operates. This case structure has a value for each operation state of the subsystem.

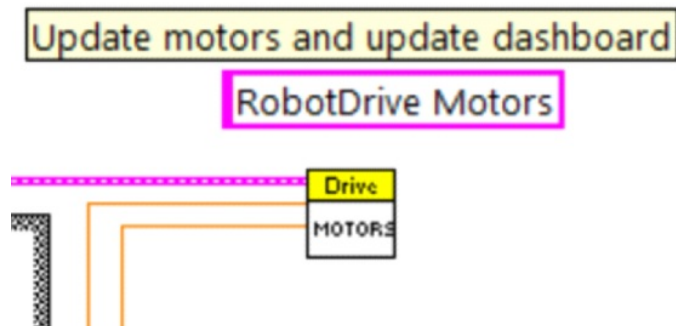
Step 4: Format a string to describe this cmd and how the previous cmd finished



Each iteration of the controller loop will optionally update the Trace VI. The framework already incorporates the subsystem name, operation, and description, and you may find it helpful to format additional set point values into the trace information. Open the Trace VI and click Enable while the robot code is running to current setpoints and commands sent to each subsystem.

The primary goal of the controller is to update actuators for the subsystem. This can occur within the case structure, but many times, it is beneficial to do it downstream of the structure to ensure that values are always updated with the correct value and in only one location in the code.

Step 5: Update your I/O



Part 3: Drive Subsystem Shipped Commands

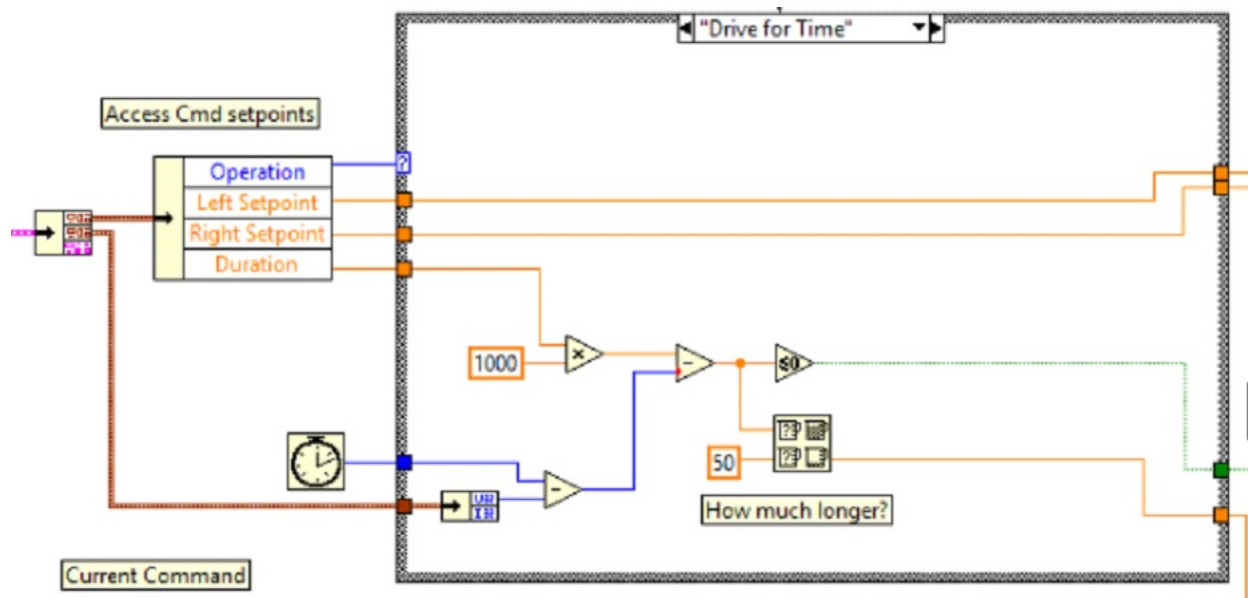
There are 3 shipped example commands for each new subsystem:

Drive For Time.vi



This VI sets the motors to run for a given number of seconds. It optionally synchronizes with the completion of the command.

The Drive for Time case will operate the motors at the set point until the timer elapses or a new command is issued. If the motors have the safety timeout enabled, it is necessary to update the motors at least once every 100ms. This is why the code waits for the smaller of the remaining time and 50ms.

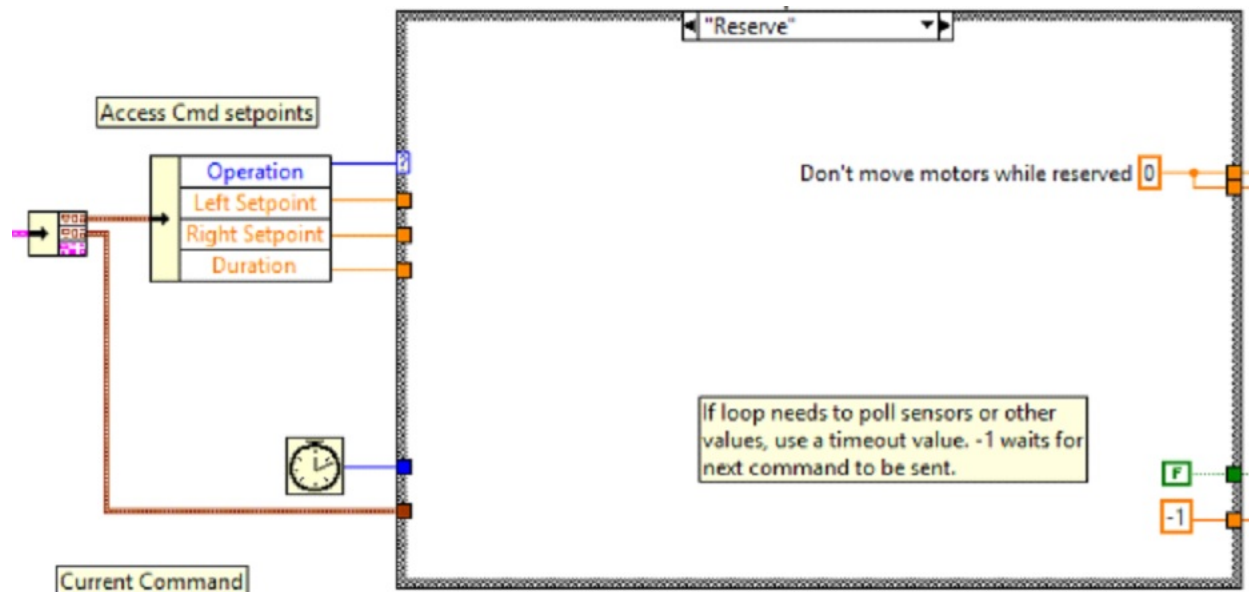


Drive Immediate.vi



Gets the desired left and right speeds for the motors and will set the motors immediately to those set points.

The Immediate case updates the motors to the set point defined by the command. The command is not considered finished since you want the motors to maintain this value until a new command comes in or until a timeout value. The timeout is useful anytime a command includes a dead band. Small values will not be requested if smaller than the dead band, and will result in growling or creeping unless the command times out.

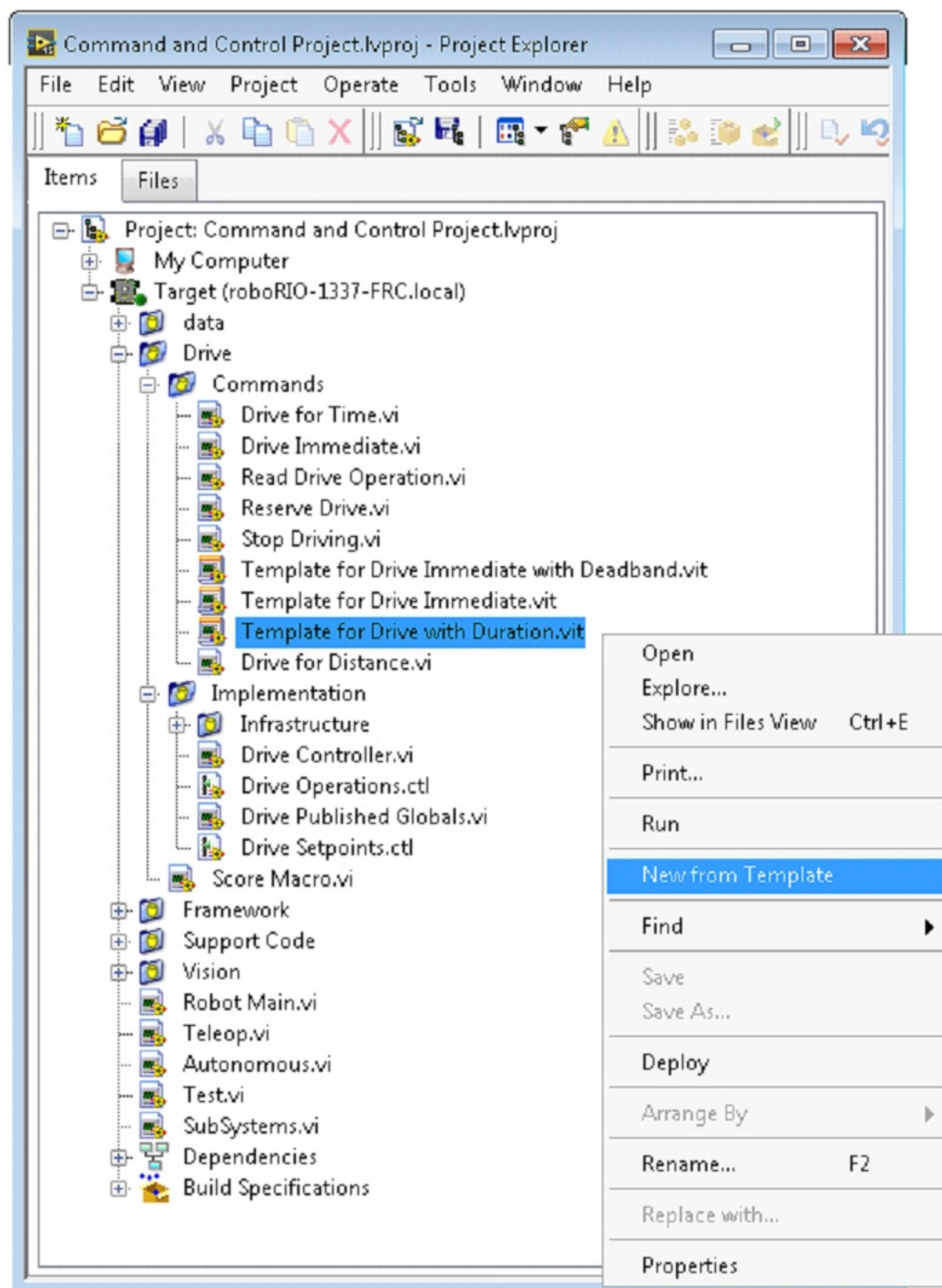


Part 4: Creating New Commands

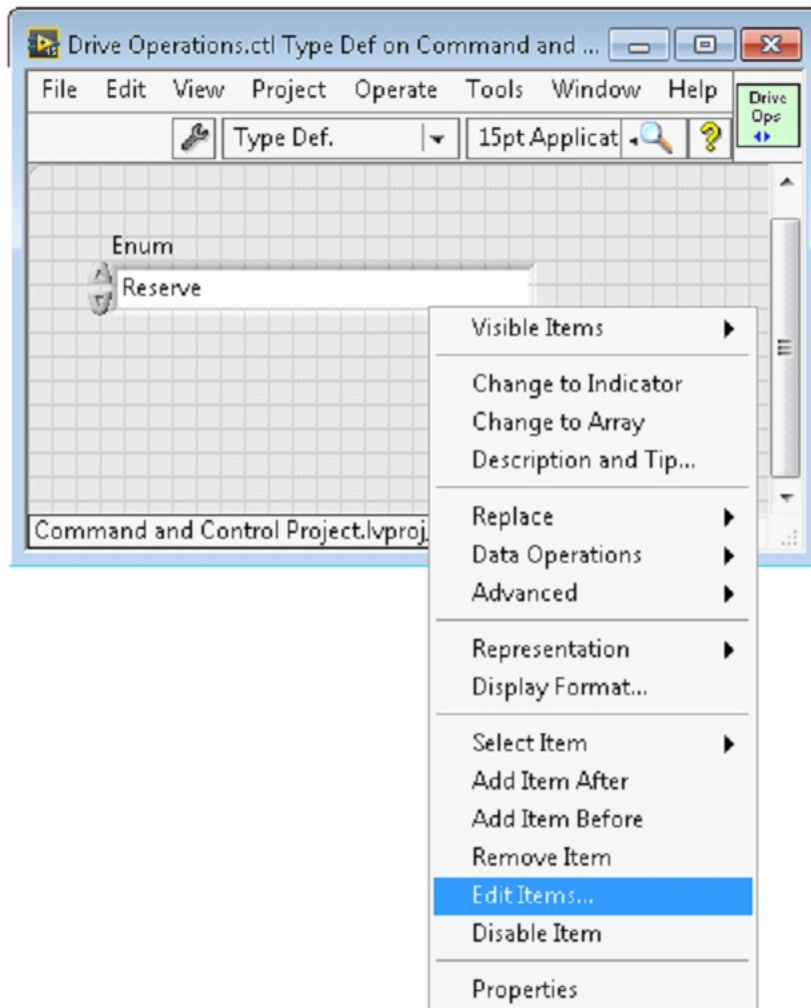
The Command and Control framework allows users to easily create new commands for a subsystem. To Create a new command open the subsystem folder/Commands In the project explorer window, choose one of the VI Templates to use as the starting point of your new command, right click, and select New From Template.

- **Immediate:** This VI notifies the subsystem about the new setpoint.
- **Immediate with deadband:** This VI compares the input value to the deadband and optionally notifies the subsystem about the new setpoint. This is very useful when joystick continuous values are being used.
- **With duration:** This VI notifies the subsystem to perform this command for the given duration, and then return to the default state. Synchronization determines whether this VI Starts the operation and returns immediately, or waits for the operation to complete. The first option is commonly used for TeleOp, and the second for Autonomous sequencing.

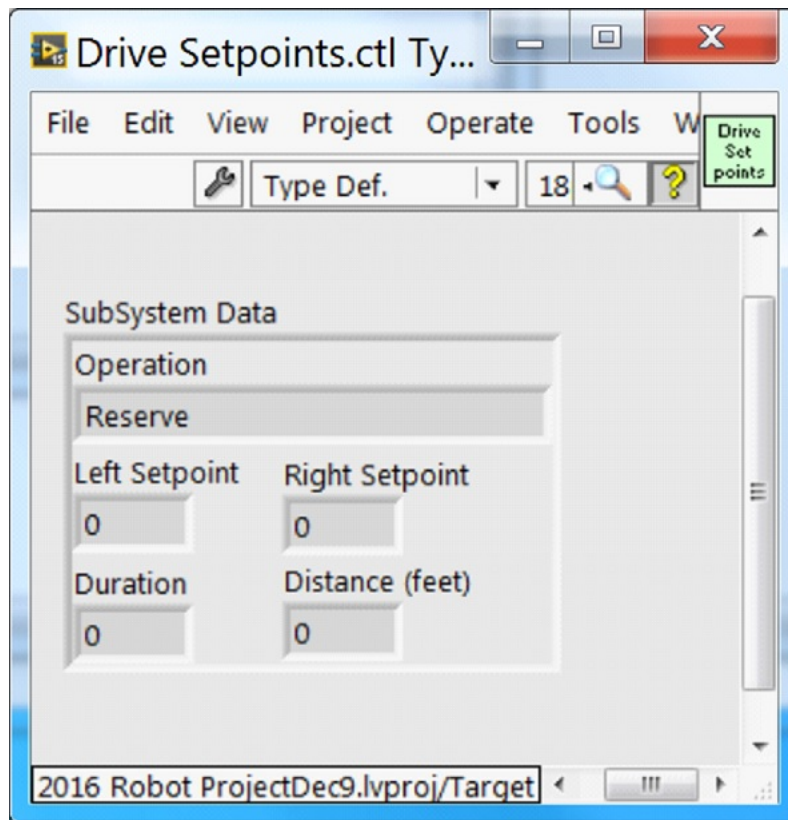
In this example we will add the new command “Drive for Distance”.



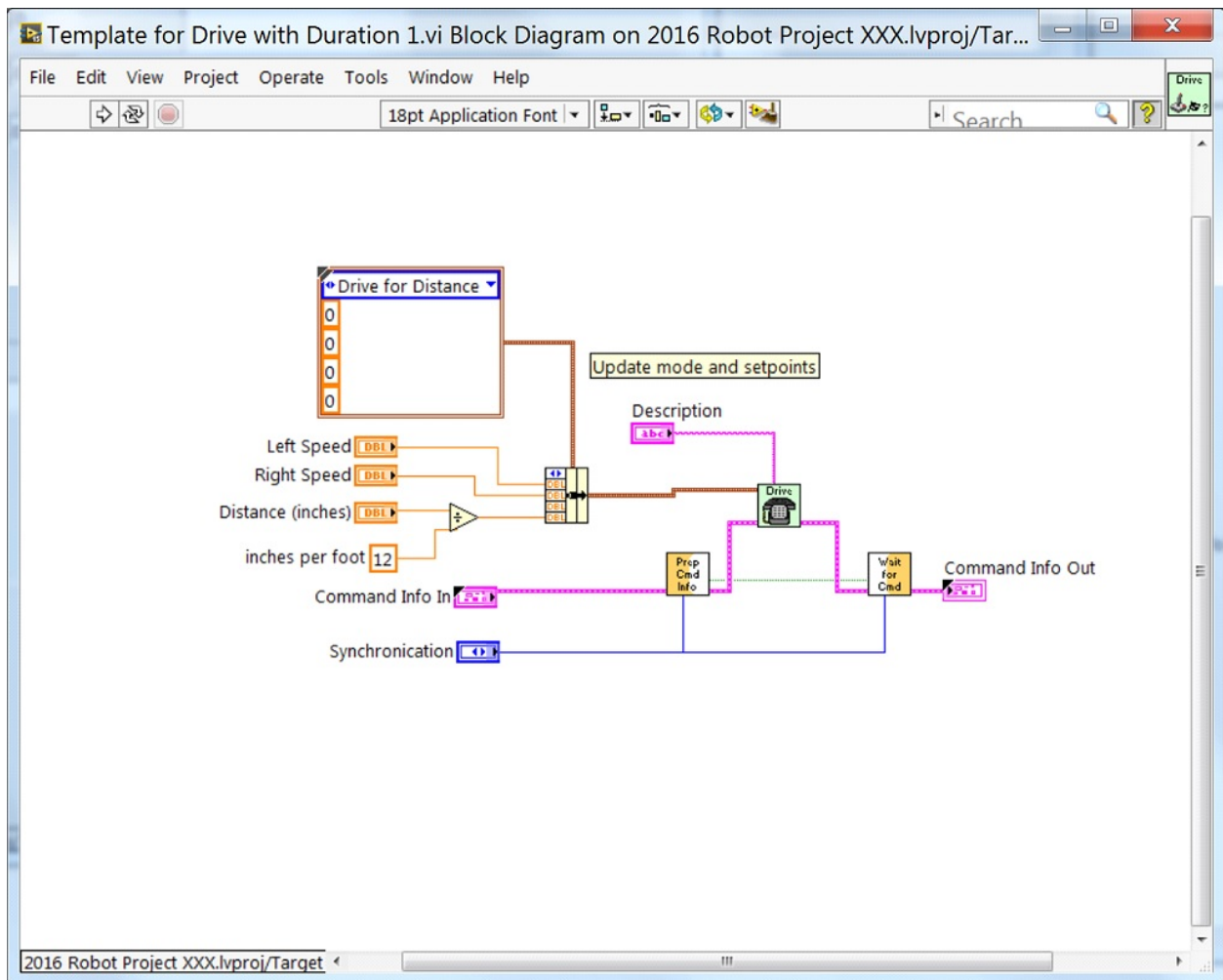
First, save the new VI with a descriptive name such as “Drive for Distance”. Next, determine whether the new command needs a new value added the Drive Operations enum typedef. The initial project code already has an enum value of Drive for Distance, but the following image shows how you would add one if needed.



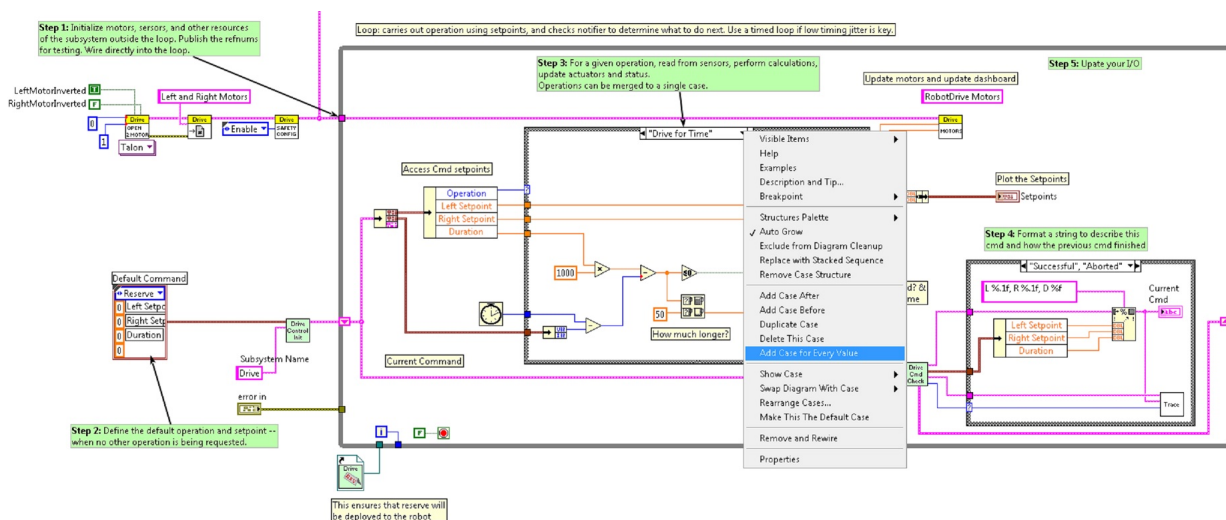
If a command needs additional information to execute, add it to the setpoints control. By default, the Drive subsystem has fields for the Left Setpoint, Right Setpoint, and Duration along with the operation to be executed. The Drive for Distance command could reuse Duration as distance, but let's go ahead and add a numeric control to the Drive Setpoints.ctl called Distance (feet).



Once that we have all of the fields needed to specify our command, we can modify the newly created Drive for Distance.vi. As shown below, select Drive for Distance from the enum's drop down menu and add a VI parameters to specify distance, speeds, etc. If the units do not match, the command VI is a great place to map between units.

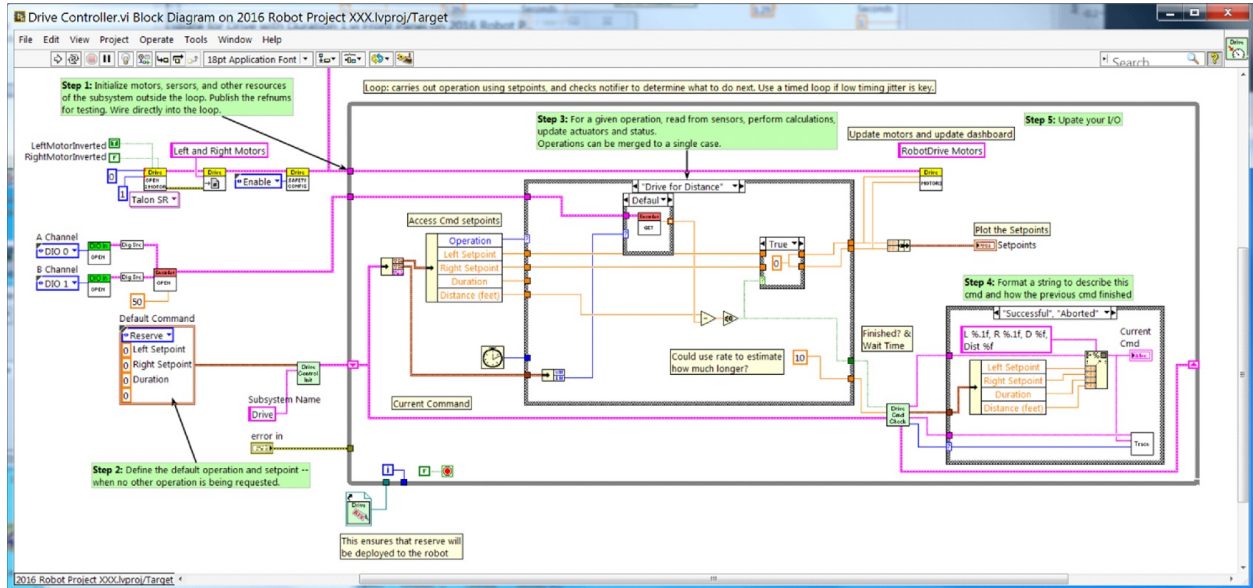


Next, add code to the Drive Controller to define what happens when the Drive for Distance command executes. Right click on the Case Structure and Duplicate or Add Case for Every Value. This will create a new “Drive for Distance” case.



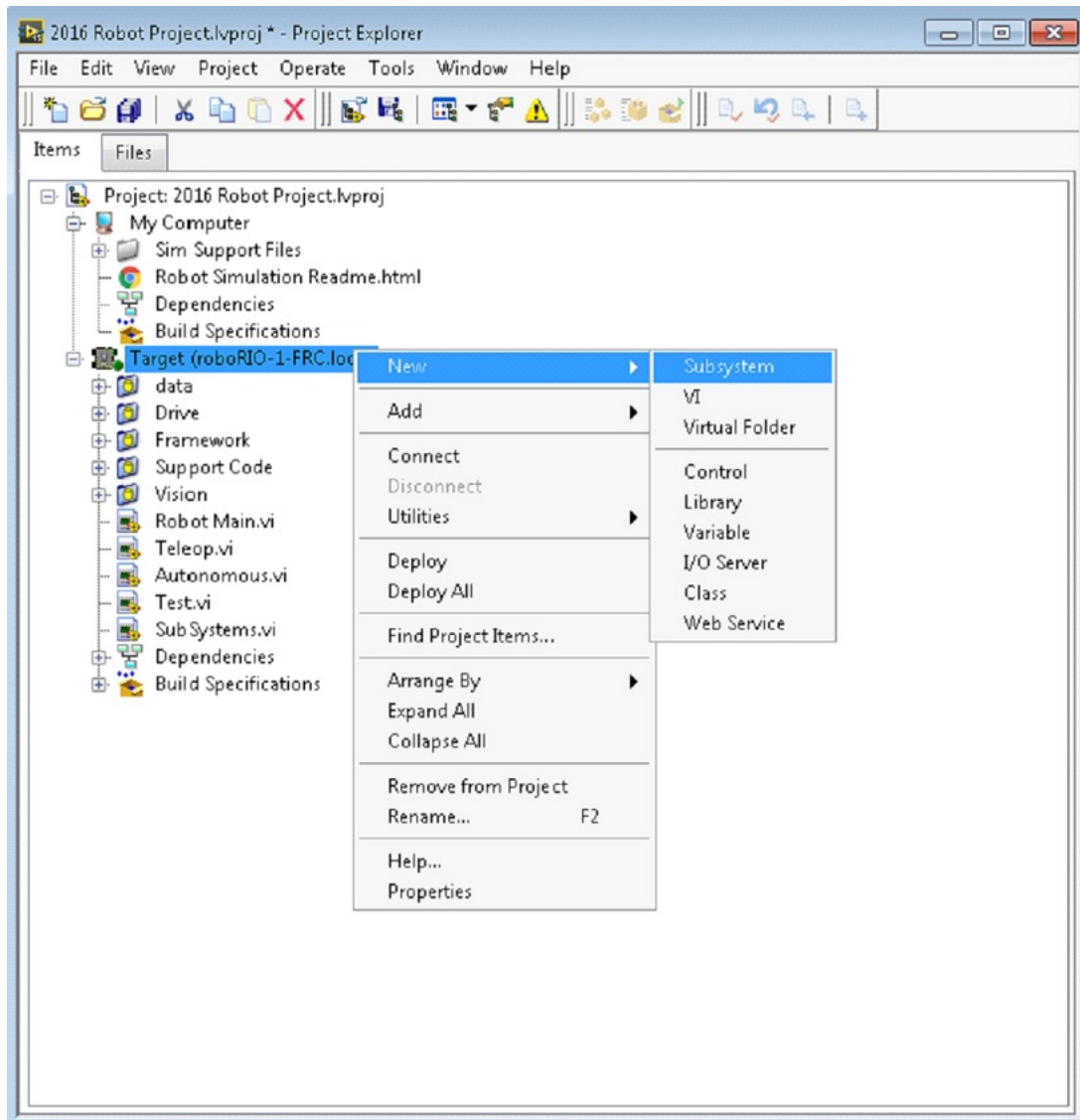
In order to access new setpoint fields, grow the “Access Cmd setpoints” unbundle node. Open

your encoder(s) on the outside, to the left of the loop. In the new diagram of the case structure, we added a call to reset the encoder on the first loop iteration and read it otherwise. There is also some simple code that compares encoder values and updates the motor power. If new controls are added to the setpoints cluster, you should also consider adding them to the Trace. The necessary changes are shown in the image below.

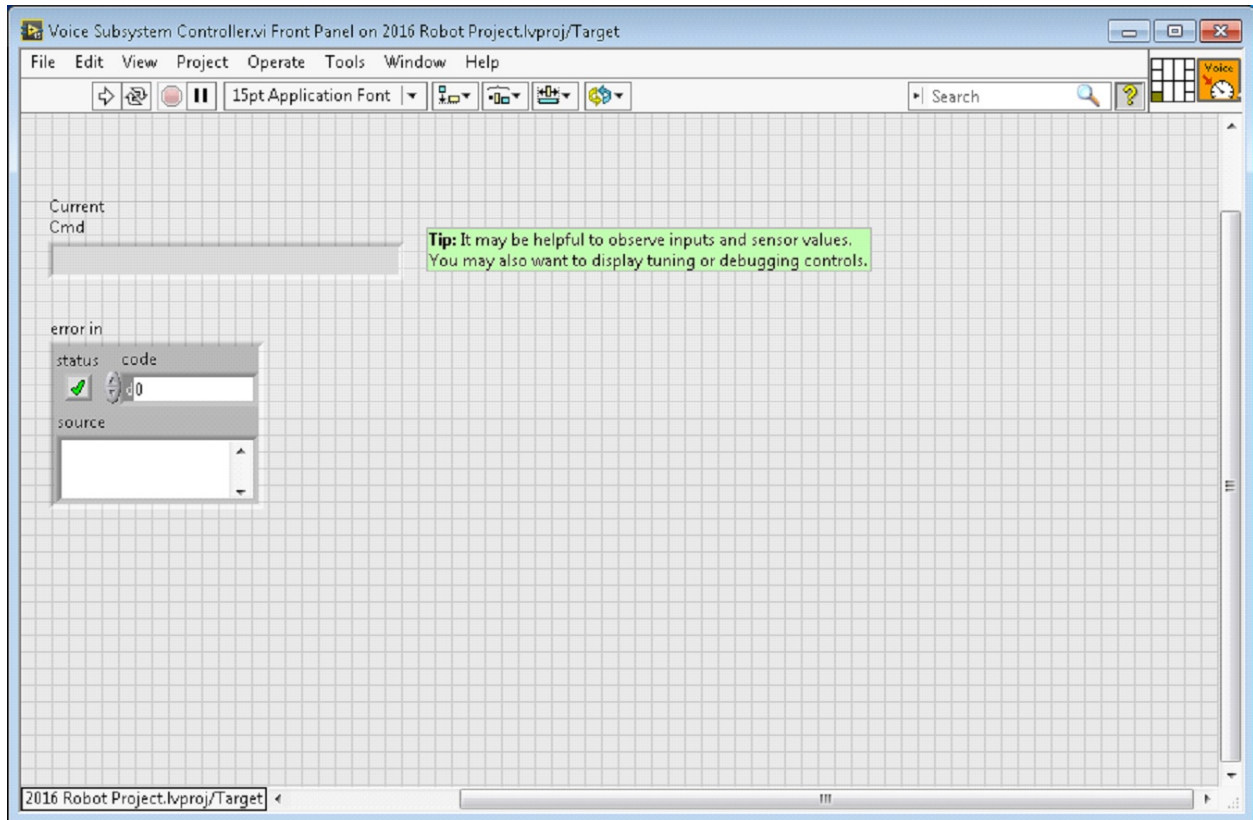


Part 5: Creating a Subsystem

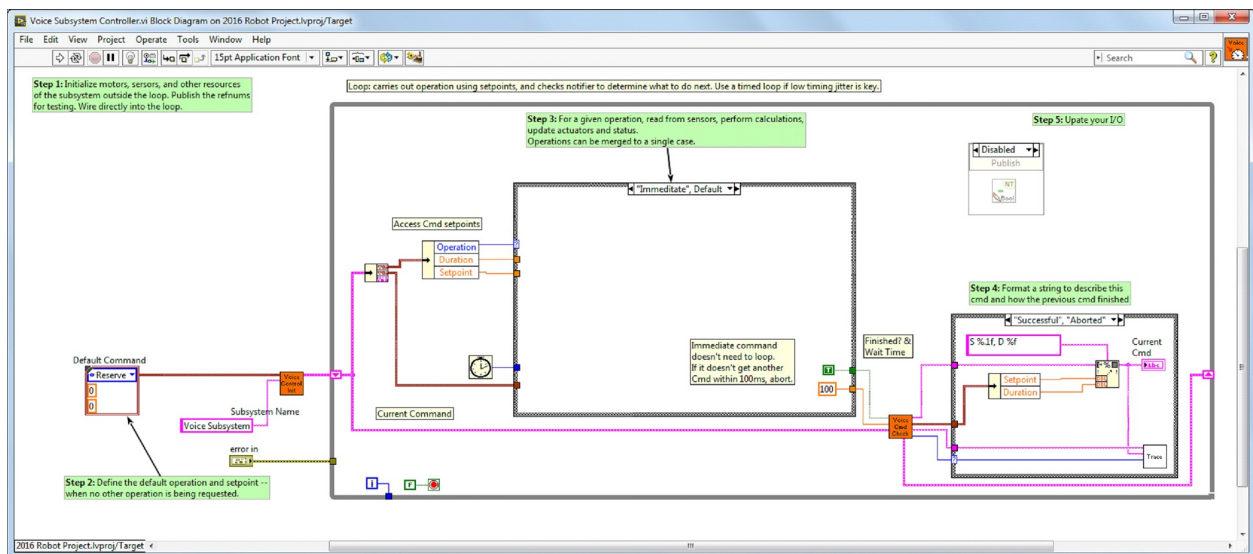
In order to create a new subsystem, right click on the roboRIO target and select New» Subsystem. In the pop up dialog box, enter the name of the subsystem, list the operational modes, and specify the color of the icon.



When you click OK, the subsystem folder will be generated and added to the project disk folder and tree. It will contain a base implementation of the VIs and controls that make up a subsystem. A call to the new controller will be inserted into the Subsystems VI. The controller VI will open, ready for you to add I/O and implement state machine or control code. Generated VI icons will use the color and name provided in the dialog. The generated code will use typedefs for set point fields and operations.



Below is the block diagram of the newly created subsystem. This code will be generated automatically when you create the subsystem.



11.2 LabVIEW Resources

11.2.1 LabVIEW Resources



Note: To learn more about programming in LabVIEW and specifically programming FRC® robots in LabVIEW, check out the following resources.

LabVIEW Basics

NI provides [tutorials on the basics of LabVIEW](#). These tutorials can help you get acquainted with the LabVIEW environment and the basics of the graphical, dataflow programming model used in LabVIEW.

NI FRC Tutorials

NI also hosts many [FRC specific tutorials and presentations ranging from basic to advanced](#). For an in-depth single resource check out the FRC Basic and Advanced Training Classes linked near the bottom of the page.

Installed Tutorials and Examples

There are also tutorials and examples for all sorts of tasks and components provided as part of your LabVIEW installation. To access the tutorials, from the LabVIEW Splash screen (the screen that appears when the program is first launched) click on the Tutorials tab on the left side. Note that the tutorials are all in one document, so once it is open you are free to browse to other tutorials without returning to the splash screen.

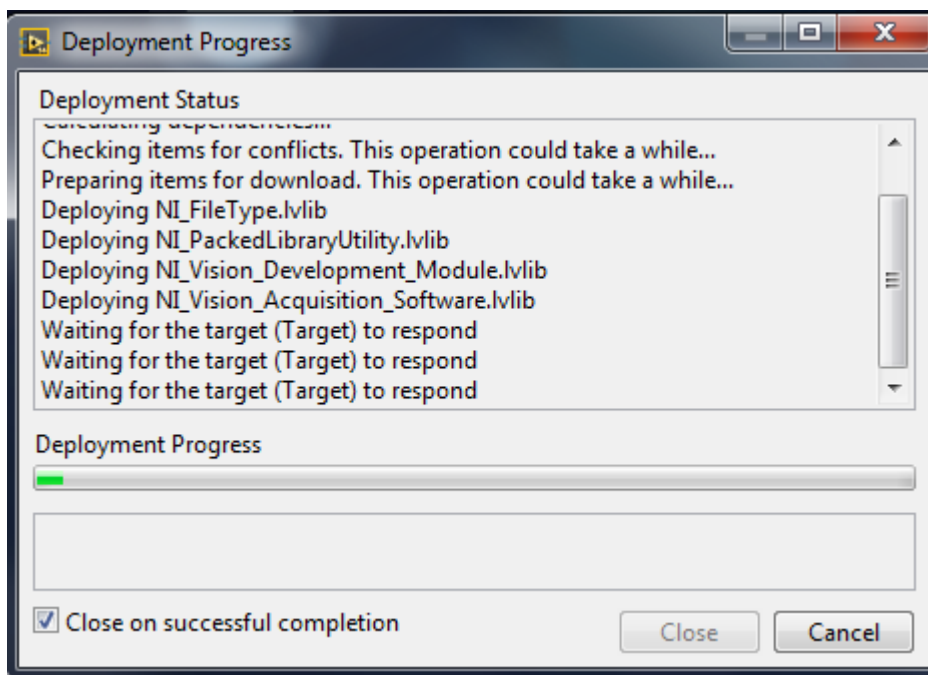
To access the examples either click the Support tab, then Find FRC Examples or anytime you're working on a program open the Help menu, select Find Examples and open the FRC Robotics folder.

11.2.2 Waiting for Target to Respond - Recovering from bad loops



Note: If you download LabVIEW code which contains an unconstrained loop (a loop with no delay) it is possible to get the roboRIO into a state where LabVIEW is unable to connect to download new code. This document explains the process required to load new, fixed, code to recover from this state.[waiting-for-target-to-respond/](#)

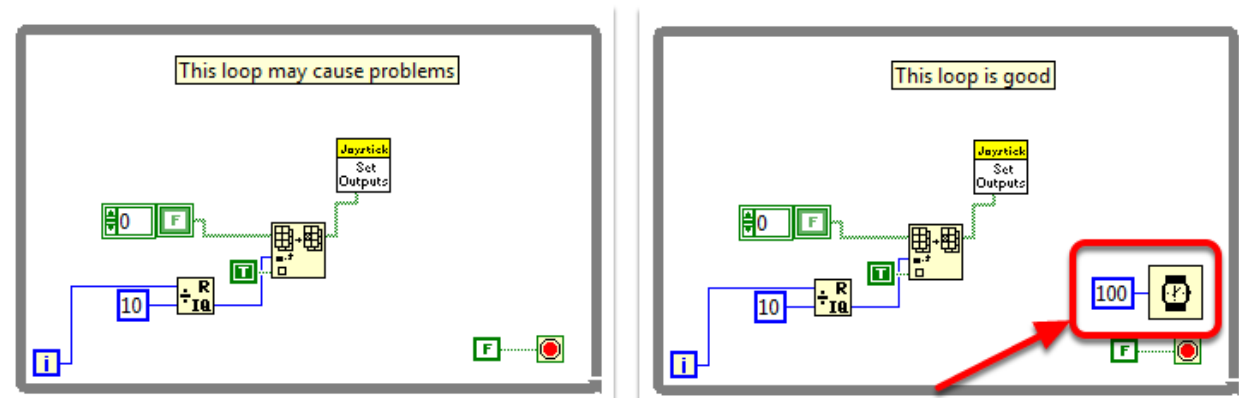
The Symptom



The primary symptom of this issue is attempts to download new robot code hang at the “Waiting for the target (Target) to respond” step as shown above. Note that there are other possible causes of this symptom (such as switching from a C++/Java program to LabVIEW program) but the steps described here should resolve most or all of them.

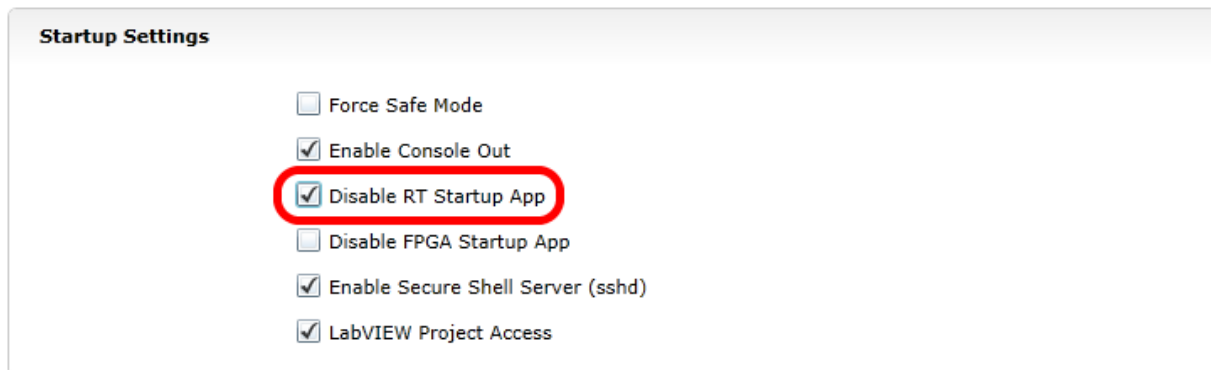
Click Cancel to close the download dialog.

The Problem



One common source of this issue is unconstrained loops in your LabVIEW code. An unconstrained loop is a loop which does not contain any delay element (such as the one on the left). If you are unsure where to begin looking, Disabled.VI, Periodic Tasks.VI and Vision Processing.VI are the common locations for this type of loop. To fix the issue with the code, add a delay element such as the Wait (ms) VI from the Timing palette, found in the right loop.

Set No App

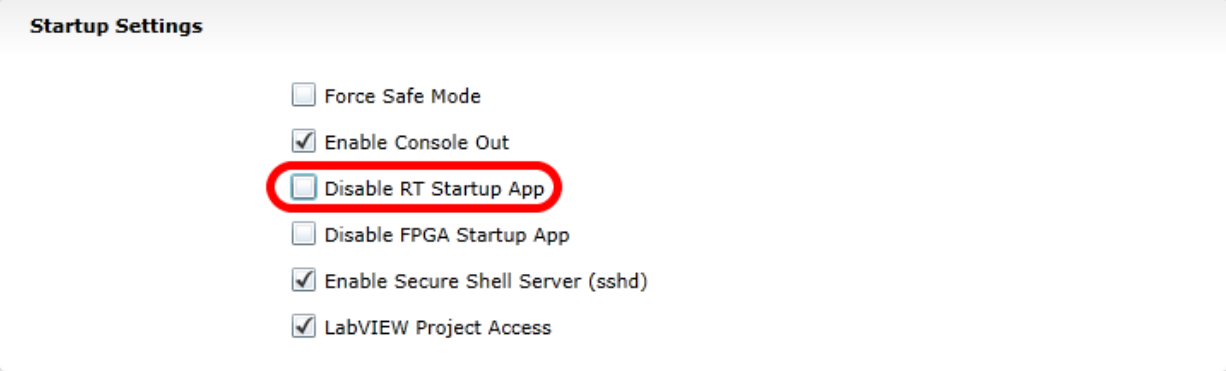


Using the roboRIO webserver (see the article on the [roboRIO Web Dashboard Startup Settings](#) for more details). **Check** the box to “Disable RT Startup App”.

Reboot

Reboot the roboRIO, either using the Reset button on the device or by click Restart in the top right corner of the webpage.

Clear No App



The screenshot shows a web interface titled "Startup Settings". It contains a list of seven checkboxes with their corresponding labels:

- ☐ Force Safe Mode
- ☒ Enable Console Out
- ☐ Disable RT Startup App (This checkbox is circled in red in the original image)
- ☐ Disable FPGA Startup App
- ☒ Enable Secure Shell Server (sshd)
- ☒ LabVIEW Project Access

Using the roboRIO webserver (see the article on the [roboRIO Web Dashboard Startup Settings](#) for more details). **Uncheck** the box to “Disable RT Startup App”.

Load LabVIEW Code

Load LabVIEW code (either using the Run button or Run as Startup). Make sure to set LabVIEW code to Run as Startup before rebooting the roboRIO or you will need to follow the instructions above again.

11.2.3 Talon SRX CAN

The Talon SRX motor controller is a CAN-enabled “smart motor controller” from Cross The Road Electronics/VEX Robotics. The Talon SRX can be controlled over the CAN bus or PWM interface. When using the CAN bus control, this device can take inputs from limit switches and potentiometers, encoders, or similar sensors in order to perform advanced control such as limiting or PID(F) closed loop control on the device.

Extensive documentation about programming the Talon SRX in all three FRC® languages can be found in the [Talon SRX Software Reference Manual on CTRE’s Talon SRX product page](#).

Note: CAN Talon SRX has been removed from WPILib. See [this blog](#) for more info and find the CTRE Toolsuite installer [here](#)

11.2.4 How To Toggle Between Two Camera Modes



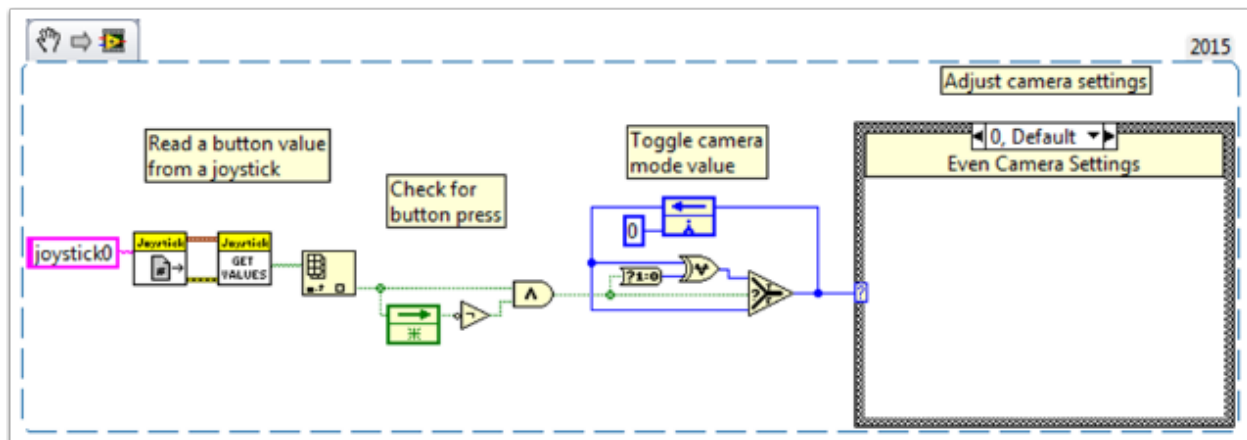
This code shows how to use a button to toggle between two distinct camera modes. The code consists of four stages.

In the first stage, the value of a button on the joystick is read.

Next, the current reading is compared to the previous reading using a **Feedback Node** and some Boolean arithmetic. Together, these ensure that the camera mode is only toggled when the button is initially pressed rather than toggling back and forth multiple times while the button is held down.

After that, the camera mode is toggled by masking the result of the second stage over the current camera mode value. This is called bit masking and by doing it with the **XOR** function the code will toggle the camera mode when the second stage returns true and do nothing otherwise.

Finally, you can insert the code for each camera mode in the case structure at the end. Each time the code is run, this section will run the code for the current camera mode.



11.2.5 LabVIEW Examples and Tutorials



Popular Tutorials

Autonomous Timed Movement Tutorial

- Move your robot autonomously based on different time intervals
- [See more on Autonomous Movement](#)

Basic Motor Control Tutorial

- Setup your roboRIO motor hardware and software
- Learn to setup the FRC® Control System and FRC Robot Project
- [See more on Motor Control](#)

Image Processing Tutorial

- Learn basic Image Processing techniques and how to use NI Vision Assistant
- [See more on Cameras and Image Processing](#)

PID Control Tutorial

- What is PID Control and how can I implement it?

Command and Control Tutorial

- What is Command and Control?
- How do I implement it?

Driver Station Tutorial

- Get to know the FRC Driver Station

Test Mode Tutorial

- Learn to setup and use Test Mode

Looking for more examples and discussions? Search through more documents or post your own discussion, example code, or tutorial by [clicking here!](#) Don't forget to mark your posts with a tag!

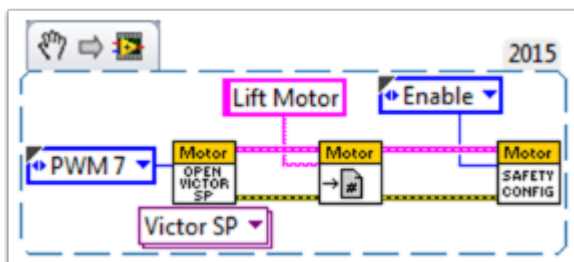
11.2.6 Add an Independent Motor to a Project



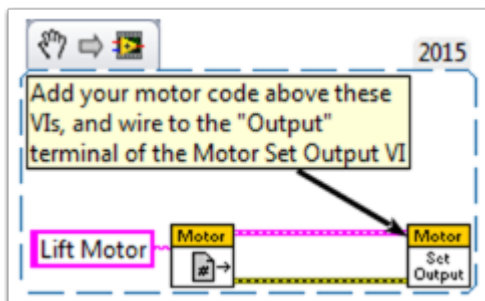
Once your drive that controls the wheels is all set, you might need to add an additional motor to control something completely independent of the wheels, such as an arm. Since this motor will not be part of your tank, arcade, or mecanum drive, you'll definitely want independent control of it.

These VI Snippets show how to set up a single motor in a project that may already contain a multi-motor drive. If you see the HAND>ARROW>LABVIEW symbol, just drag the image into your block diagram, and voila: code! Ok, here's how you do it.

FIRST, create a motor reference in the **Begin.vi**, using the **Motor Control Open VI** and **Motor Control Refnum Registry Set VI**. These can be found by right-clicking in the block diagram and going to **WPI Robotics Library>>RobotDrive>>Motor Control**. Choose your PWM line and name your motor. I named mine "Lift Motor" and connected it to PWM 7. (I also included and enabled the Motor Control Safety Config VI, which automatically turns off the motor if it loses connection.)

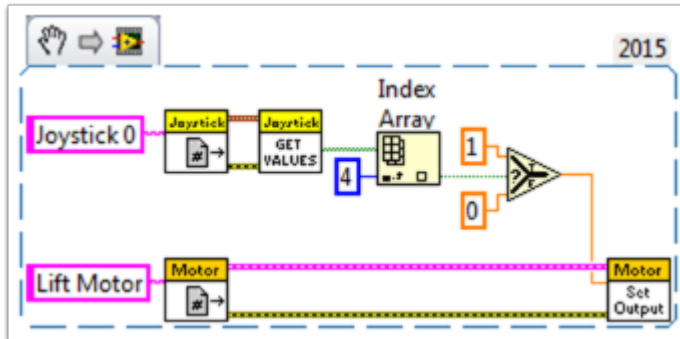


Now, reference your motor (the name has to be exact) in the **Teleop.vi** using the **Motor Control Refnum Registry Get VI** and tell it what to do with the **Motor Control Set Output VI**. These are in the same place as the above VIs.

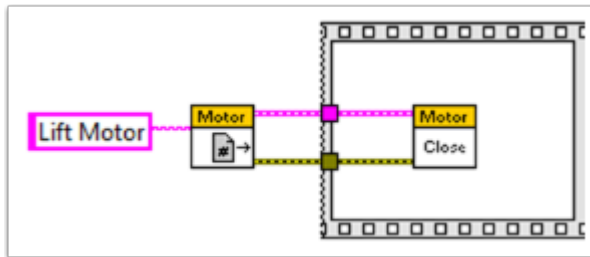


For example, the next snippet tells the Lift Motor to move forward if button 4 is pressed on Joystick 0 and to remain motionless otherwise. For me, button 4 is the left bumper on my

Xbox style controller (“Joystick 0”). For much more in-depth joystick button options, check out *How to Use Joystick Buttons to Control Motors or Solenoids*.



Finally, we need to close the references in the **Finish.vi** (just like we do with the drive and joystick), using the **Motor Control Refnum Registry Get VI** and **Motor Control Close VI**. While this picture shows the Close VI in a flat sequence structure by itself, we really want all of the Close VIs in the same frame. You can just put these two VIs below the other Get VIs and Close VIs (for the joystick and drive).



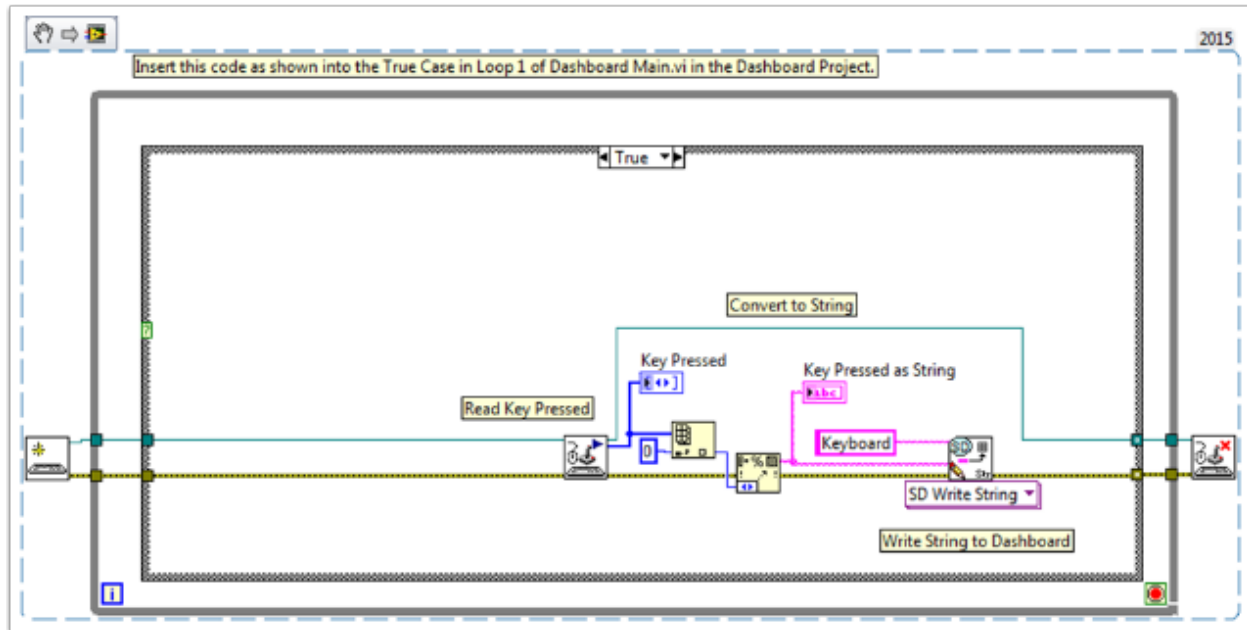
I hope this helps you program the best robot ever! Good luck!

11.2.7 Keyboard Navigation with the roboRIO



This example provides some suggestions for controlling the robot using keyboard navigation in place of a joystick or other controller. In this case, we use the A, W, S, and D keys to control two drive motors in a tank drive configuration.

The first VI Snippet is the code that will need to be included in the Dashboard Main VI. You can insert this code into the True case of Loop 1. The code opens a connection to the keyboard before the loop begins, and on each iteration it reads the pressed key. This information is converted to a string, which is then passed to the Teleop VI in the robot project. When Loop 1 stops running, the connection to the keyboard is closed.



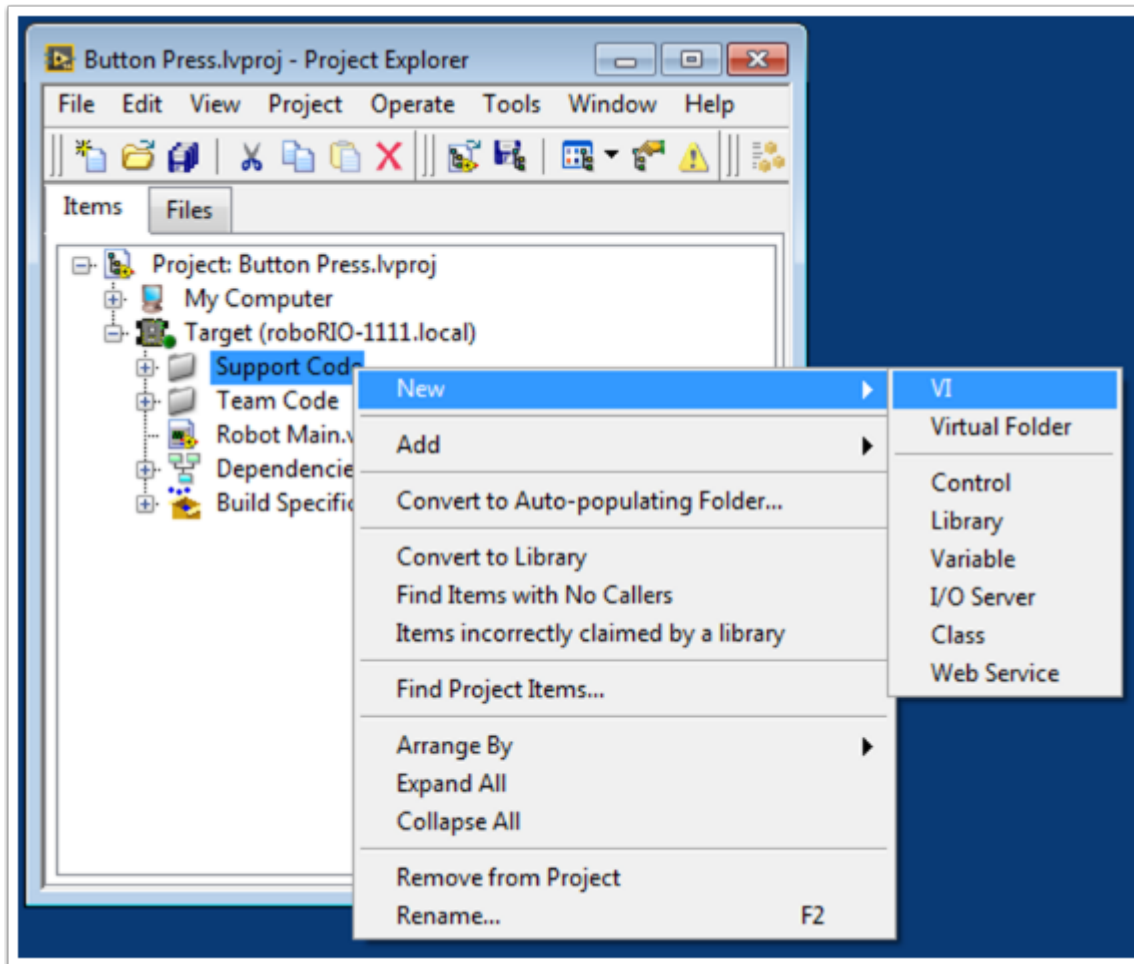
The second VI Snippet is code that should be included in the Teleop VI. This reads the string value from the Dashboard that indicates which key was pressed. A Case Structure then determines which values should be written to the left and right motors, depending on the key. In this case, W is forward, A is left, D is right, and S is reverse. Each case in this example runs the motors at half speed. You can keep this the same in your code, change the values, or add additional code to allow the driver to adjust the speed, so you can drive fast or slow as necessary. Once the motor values are selected, they are written to the drive motors, and motor values are published to the dashboard.

11.2.8 Making a One-Shot Button Press

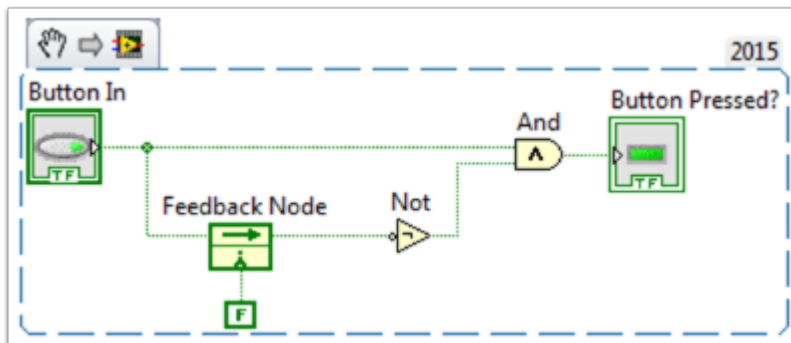


When using the Joystick Get Values function, pushing a joystick button will cause the button to read TRUE until the button is released. This means that you will most likely read multiple TRUE values for each press. What if you want to read only one TRUE value each time the button is pressed? This is often called a “One-Shot Button”. The following tutorial will show you how to create a subVI that you can drop into your Teleop.vi to do this.

First, create a new VI in the Support Code folder of your project.



Now on the block diagram of the new VI, drop in the following code snippet.

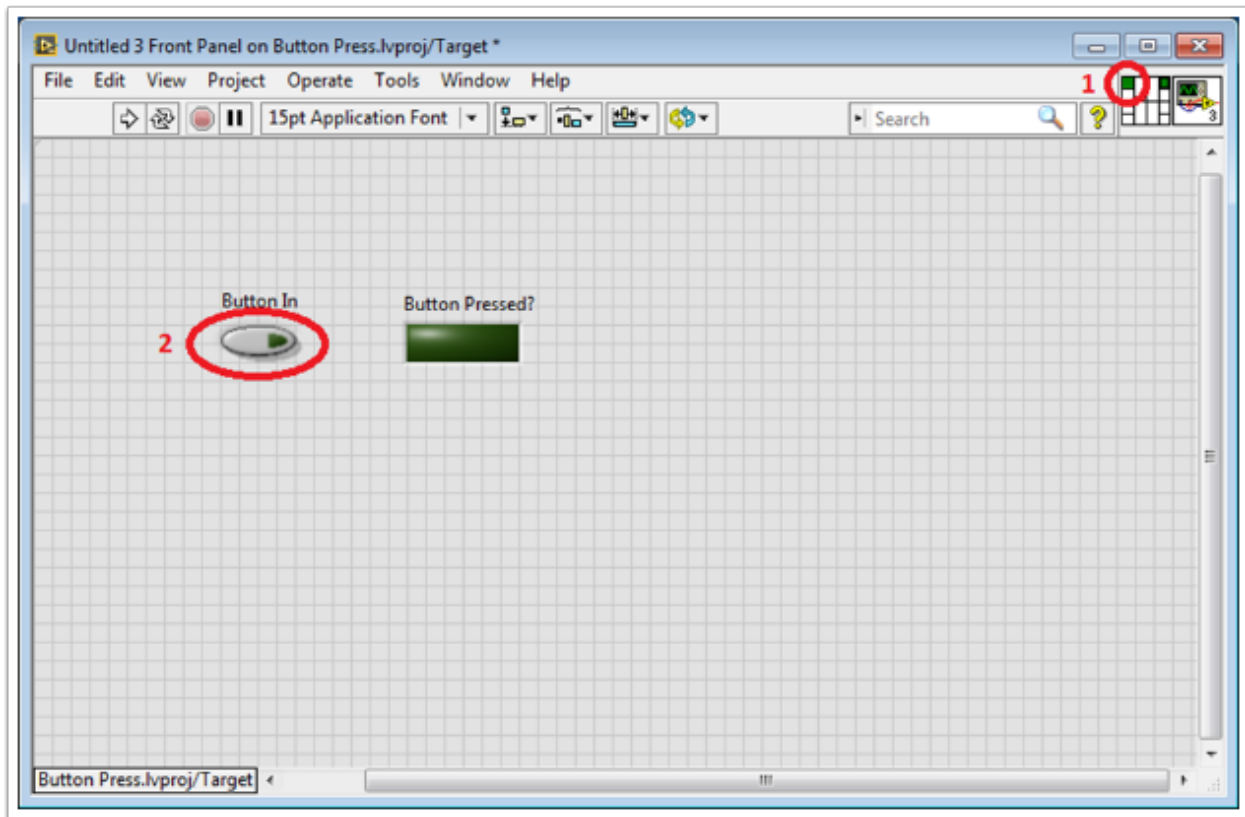


This code uses a function called the Feedback Node. We have wired the current value of the button into the left side of the feedback node. The wire coming out of the arrow of the feedback node represents the previous value of the button. If the arrow on your feedback node is going the opposite direction as shown here, right click to find the option to reverse the direction.

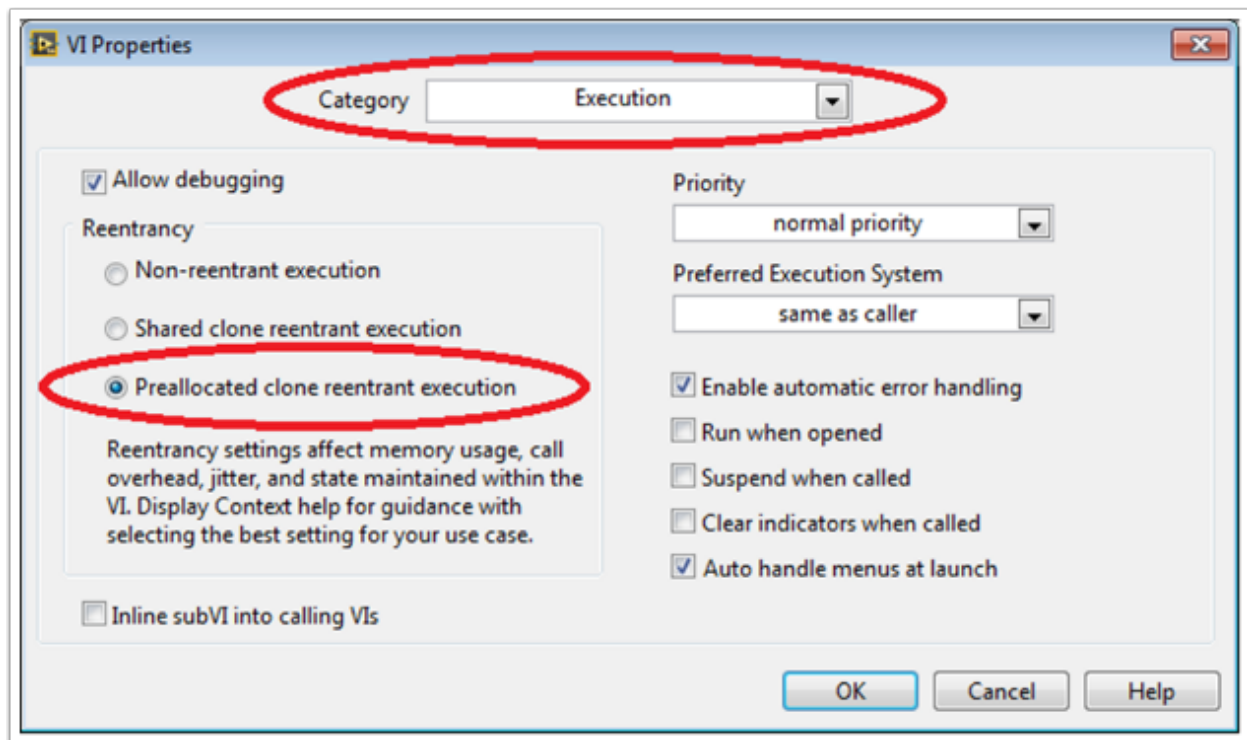
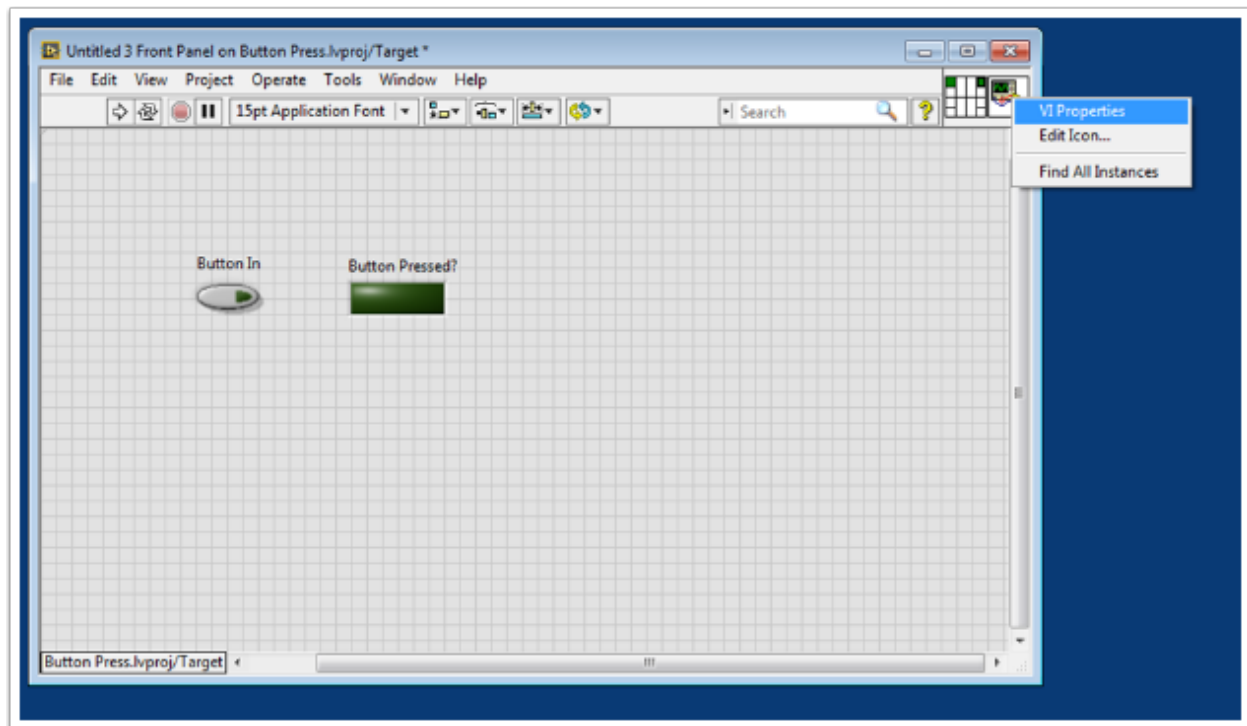
When a button is pressed, the value of the button goes from FALSE to TRUE. We want the output of this VI to be TRUE only when the current value of the button is TRUE, and the previous value of the button is FALSE.

Next we need to connect the boolean control and indicator to the inputs and outputs of the

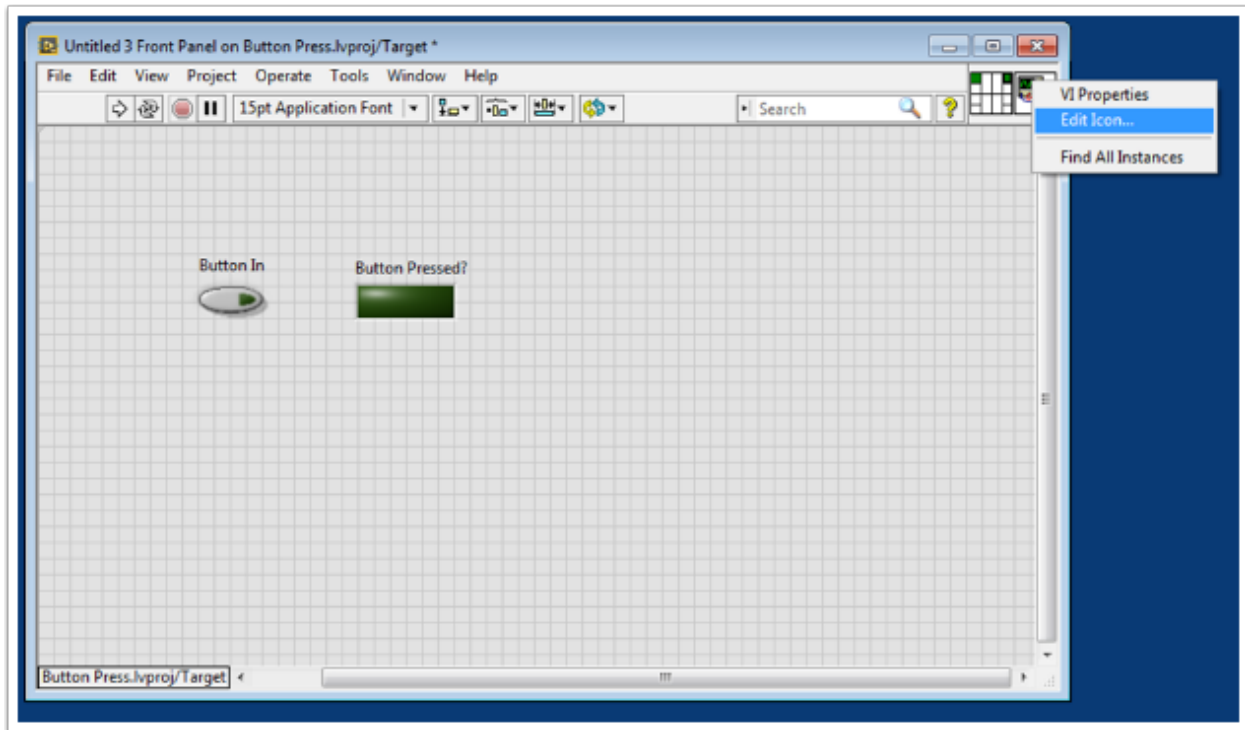
VI. To do this, first click the block on the connector pane, then click the button to connect the two (see the diagram below). Repeat this for the indicator.



Next, we need to change the properties of this VI so that we can use multiples of this VI in our TeleOp.vi. Right click the VI Icon and go to VI Properties. Then select the category "Execution" and select "Preallocated clone reentrant execution".

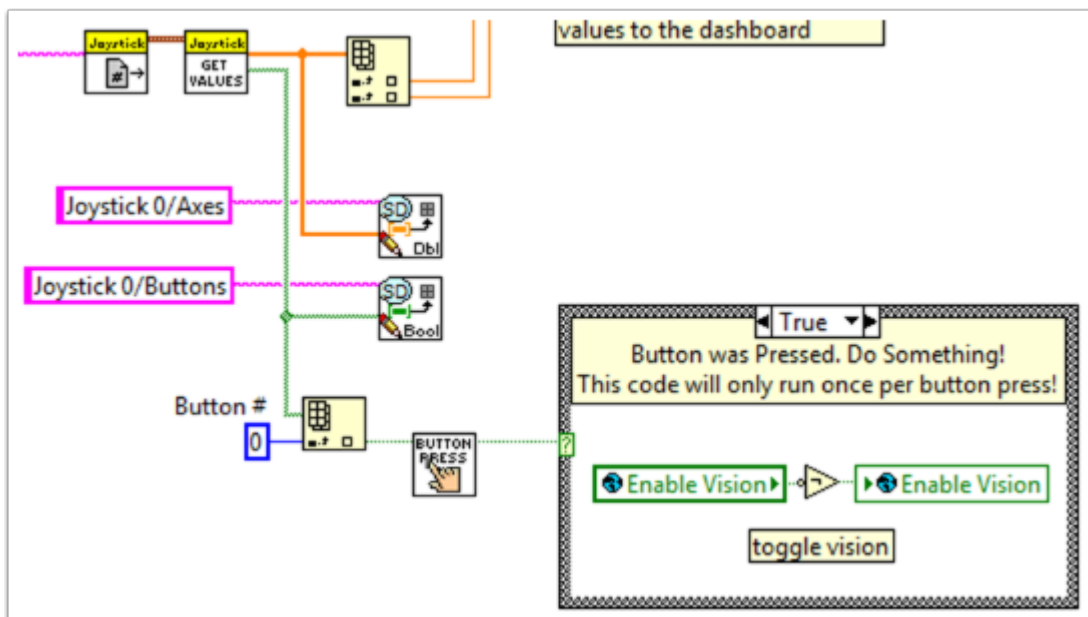


Lastly, we should change the VI Icon to be more descriptive of the VI's function. Right click the Icon and go to Edit Icon. Create a new Icon.



Finally, save the VI with a descriptive name. You can now drag and drop this VI from the Support Files folder into your TeleOp.vi. Here is a copy of the completed VI: Button_Press.vi

Here's an example of how you could use this VI.



11.2.9 Adding Safety Features to Your Robot Code

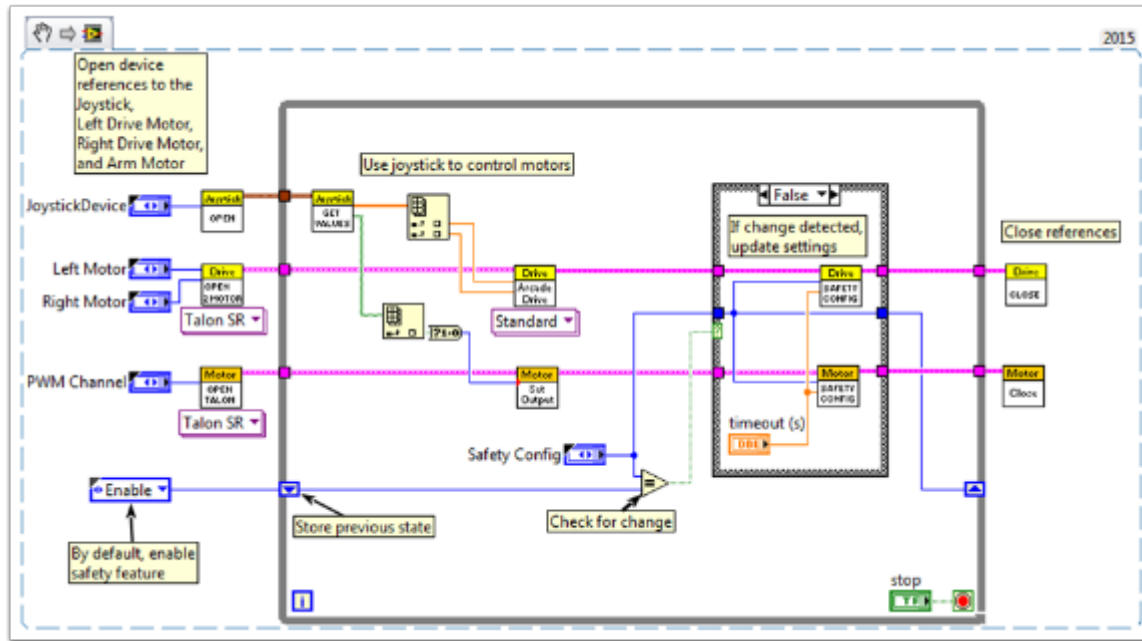


A common problem with complex projects is making sure that all of your code is executing when you expect it to. Problems can arise when tasks with high priority, long execution times, or frequent calls hog processing power on the roboRIO. This leads to what is known as “starvation” for the tasks that are not able to execute due to the processor being busy. In most cases this will simply slow the reaction time to your input from the joysticks and other devices. However, this can also cause the drive motors of your robot to stay on long after you try to stop them. To avoid any robotic catastrophes from this, you can implement safety features that check for task input starvation and automatically shut down potentially harmful operations.

There are built-in functions for the motors that allow easy implementation of safety checks. These functions are:

- Robot Drive Safety Configuration
- Motor Drive Safety Configuration
- Relay Safety Configuration
- PWM Safety Configuration
- Solenoid Safety Configuration
- Robot Drive Delay and Update Safety

In all of the Safety Configuration functions, you can enable and disable the safety checks while your programming is running and configure what timeout you think is appropriate. The functions keep a cache of all devices that have the safety enabled and will check if any of them have exceeded their time limit. If any has, all devices in the cache will be disabled and the robot will come to an immediate stop or have its relay/PWM/solenoid outputs turned off. The code below demonstrates how to use the Drive Safety Configuration functions to set a maximum time limit that the motors will receive no input before being shut off.



To test the safety shut-off, try adding a Wait function to the loop that is longer than your timeout!

The final function that relates to implementing safety checks—Robot Drive Delay and Update Safety—allows you to put the roboRIO in Autonomous Mode without exceeding the time limit. It will maintain the current motor output without making costly calls to the Drive Output functions, and will also make sure that the safety checks are regularly updated so that the motors will not suddenly stop.

Overall, it is highly recommended that some sort of safety check is implemented in your project to make sure that your robot is not unintentionally left in a dangerous state!

11.2.10 How to Use Joystick Buttons to Control Motors or Solenoids

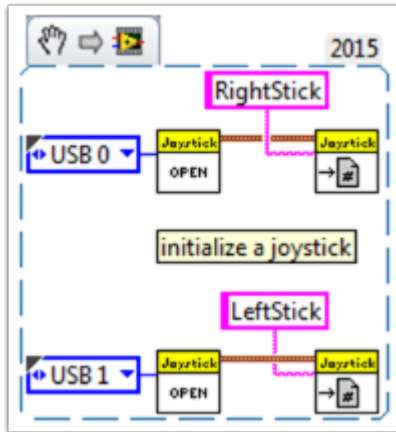


As we all get our drive systems working, we are moving on to connecting our auxiliary devices such as motors and solenoids. With this, we will generally use joystick buttons to control these devices. To get started with this, we'll go through several ways to control devices with joystick buttons.

Did you know that you can click and drag a VI Snippet from a document like this right into your LabVIEW code? Try it with the snippets in this document.

Setup:

No matter what the configuration, you'll need to add one, two, or more (if you're really excited) joysticks to the "Begin.vi". The first example uses 2 joysticks and the others only use one. Give each one a unique name so we can use it in other places, like the snippet below. I named them "LeftStick" and "RightStick" because they are on the left and right sides of my desk. If your joysticks are already configured, great! You can skip this step.

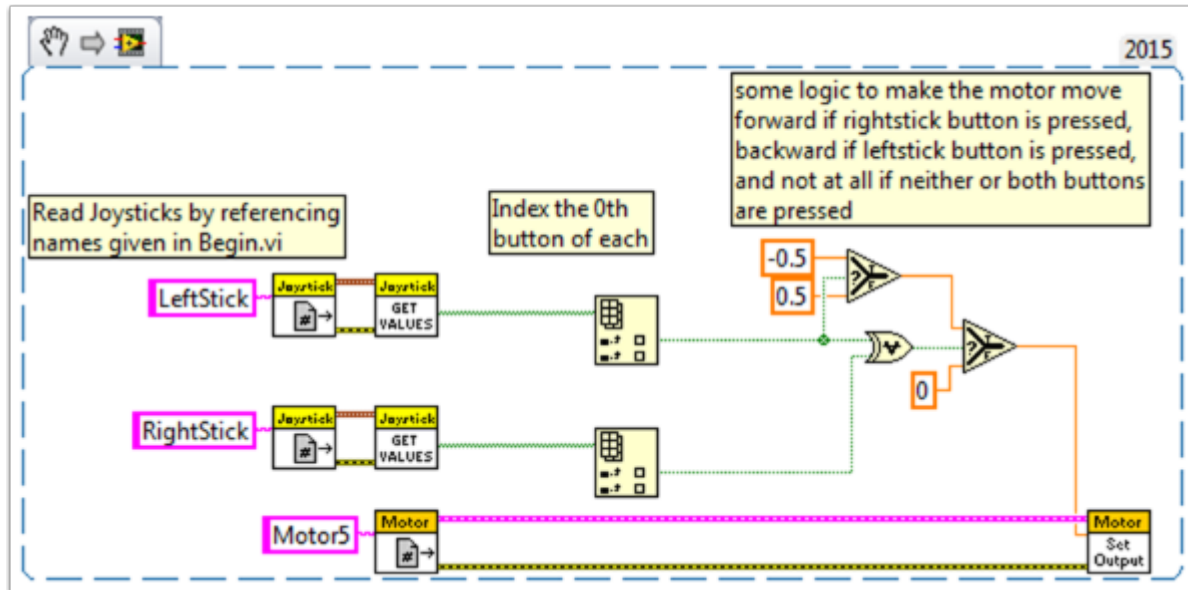


The rest of the code in this document will be placed in the "Teleop.VI" This is where we will be programming our joystick buttons to control different aspects of our motors or solenoids.

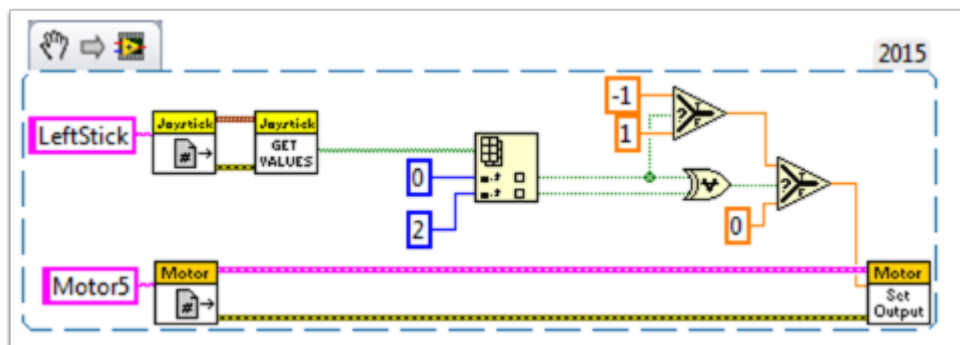
Scenario 1

"I want a motor to move one way when I press one button and the other way when I press a different button."

This code uses button 0 on two different joysticks to control the same motor. If button 0 on LeftStick is pressed, the motor moves backward, and if button 0 on RightStick is pressed, the motor moves forward. If both buttons are pressed or neither button is pressed, the motor doesn't move. Here I named my motor reference "Motor5", but you can name your motor whatever you want in the "Begin.vi"



You may want to use multiple buttons from the same joystick for control. For an example of this, look at the following VI snippet or the VI snippet in Scenario 2.



Here I used joystick buttons 0 and 2, but feel free to use whatever buttons you need.

Scenario 2

"I want different joystick buttons move at various speeds."

This example could be helpful if you need to have one motor do different things based on the buttons you press. For instance, let's say my joystick has a trigger (button 0) and 4 buttons on top (buttons 1 through 4). In this case, the following buttons should have the following functions:

- button 1 - move backward at half speed
- button 2 - move forward at half speed
- button 3 - move backward at 1/4 speed
- button 4 - move forward at 1/4 speed
- trigger - full speed ahead! (forward at full speed)

We would then take the boolean array from the “JoystickGetValues.vi” and wire it to a “Boolean Array to Number” node (Numeric Palette-Conversion Palette). This converts the boolean array to a number that we can use. Wire this numeric to a case structure.

Each case corresponds to a binary representation of the values in the array. In this example, each case corresponds to a one-button combination. We added six cases: 0 (all buttons off), 1 (button 0 on), 2 (button 1 on), 4 (button 2 on), 8 (button 3 on), and 16 (button 4 on). Notice we skipped value 3. 3 would correspond to buttons 0 and 1 pressed at the same time. We did not define this in our requirements so we’ll let the default case handle it.

It might be helpful to review the LabVIEW 2014 Case Structure Help document here:

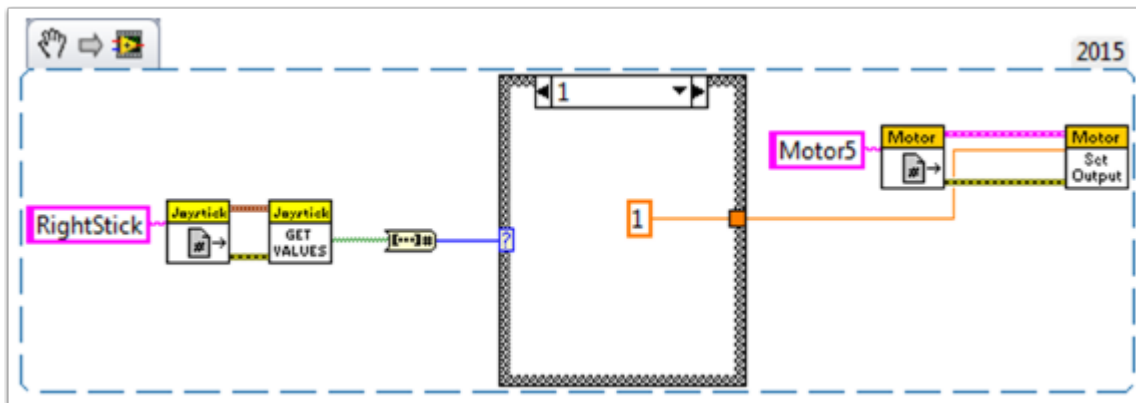
https://zone.ni.com/reference/en-XX/help/371361L-01/glang/case_structure/

There are also 3 Community Tutorials on case structures here:

<https://forums.ni.com/t5/Curriculum-and-Labs-for/Unit-3-Case-Structures-Lesson-1/ta-p/3505945?profile.language=en>

<https://forums.ni.com/t5/Curriculum-and-Labs-for/Unit-3-Case-Structures-Lesson-2/ta-p/3505933?profile.language=en>

<https://forums.ni.com/t5/Curriculum-and-Labs-for/Unit-3-Case-Structures-Lesson-3/ta-p/3505979?profile.language=en>

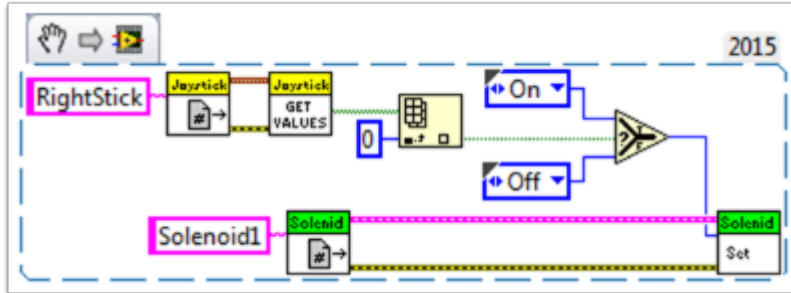


Since our requirements were simple, we only need a single constant in each case. For case 1 (full ahead) we use a 1, for case 2 (half back) we use a -0.5, etc. We can use any constant value between 1 and -1. I left case 0 as the default so if multiple buttons are pressed (any undefined state was reached) the motor will stop. You of course are free to customize these states however you want.

Scenario 3

“I want to control a solenoid with my joystick buttons.”

By now, we are familiar with how the joystick outputs the buttons in an array of booleans. We need to index this array to get the button we are interested in, and wire this boolean to a select node. Since the “Solenoid Set.vi” requires a Enum as an input, the easiest way to get the enum is to right click the “Value” input of the “Solenoid Set.vi” and select “Create Constant”. Duplicate this constant and wire one copy to the True terminal and one to the False terminal of the select node. Then wire the output of the select node to the “Value” input of the solenoid VI.



Happy Roboting!

11.2.11 Local and Global Variables in LabVIEW for FRC



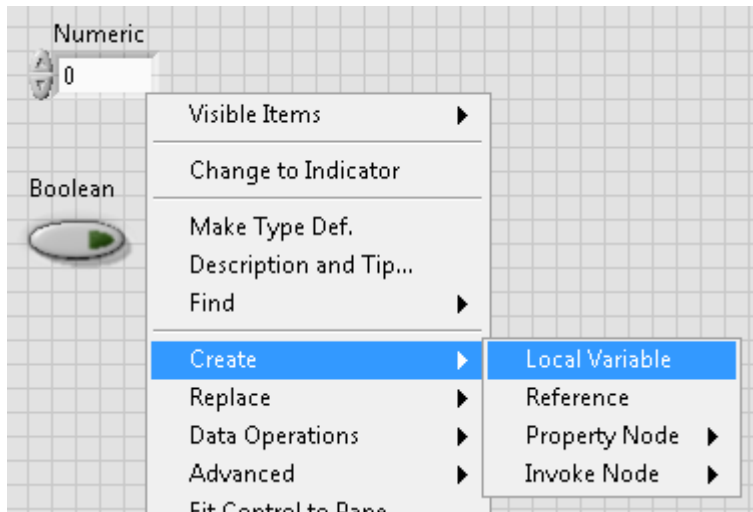
This example serves as an introduction to local and global variables, how they are used in the default LabVIEW for FRC® Robot Project, and how you might want to use them in your project.

Local variables and global variables may be used to transfer data between locations within the same VI (local variables) or within different VI's (global variables), breaking the conventional [Data Flow Paradigm](#) for which LabVIEW is famous. Thus, they may be useful when, for whatever reason, you cannot wire the value directly to the node to another.

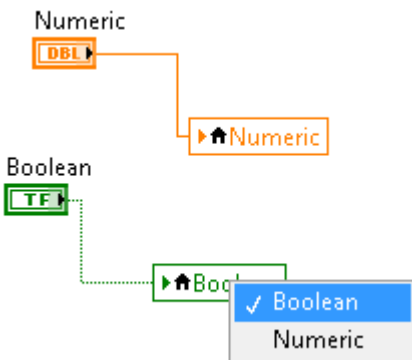
Note: One possible reason may be that you need to pass data between consecutive loop iterations; Miro_T covered this [in this post](#). It should also be noted that the [feedback node](#) in LabVIEW may be used as an equivalent to the shift register, although that may be a topic for another day!

Introduction to Local and Global Variables

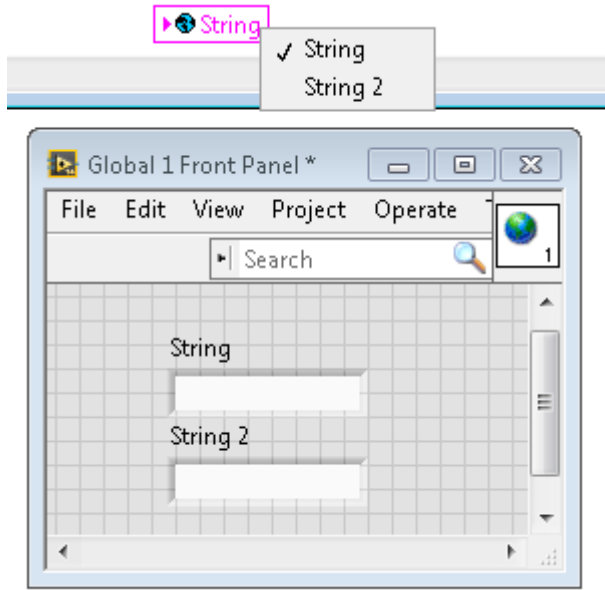
Local variables may be used within the same VI. Create a local variable by right-clicking a control or indicator on your Front Panel:



You may create a local variable from the Structures palette on the block diagram as well. When you have multiple local variables in one VI, you can left-click to choose which variable it is:



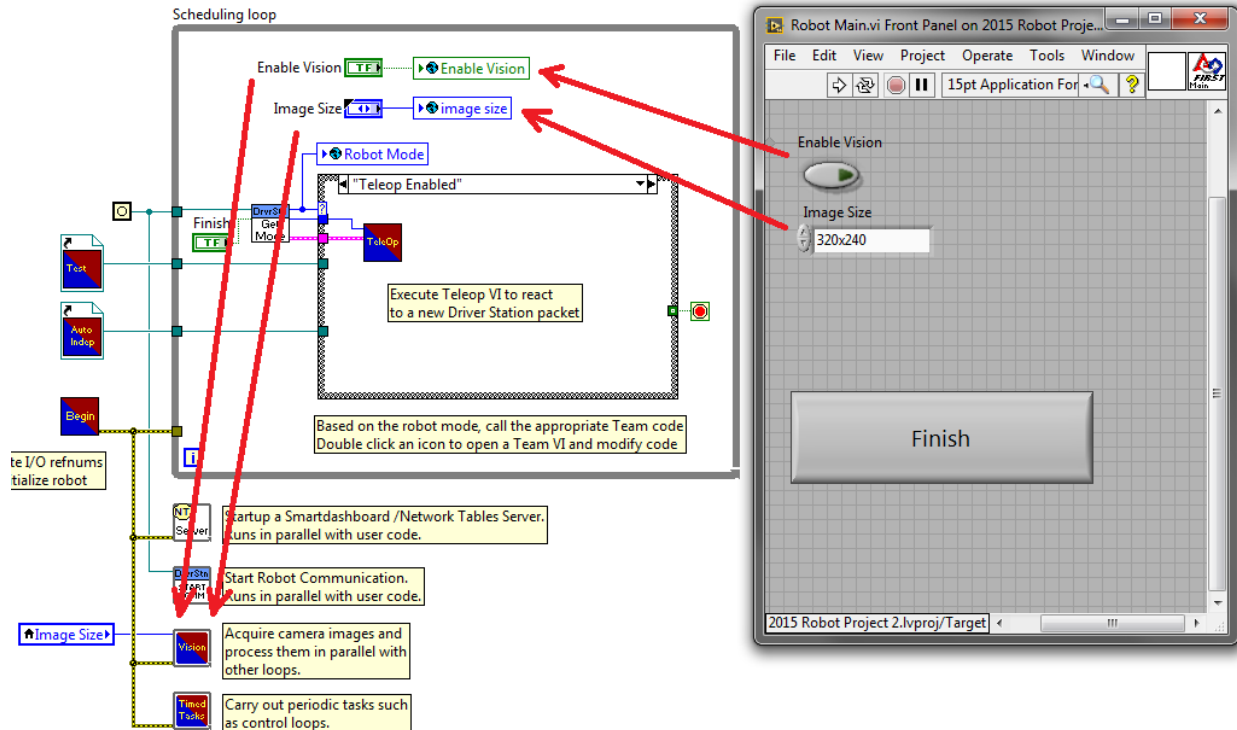
Global variables are created slightly differently. Add one to the block diagram from the Structures palette, and notice that when you double-click it, it opens a separate front panel. This front panel does not have a block diagram, but you add as many entities to the front panel as you wish and save it as a *.vi file:



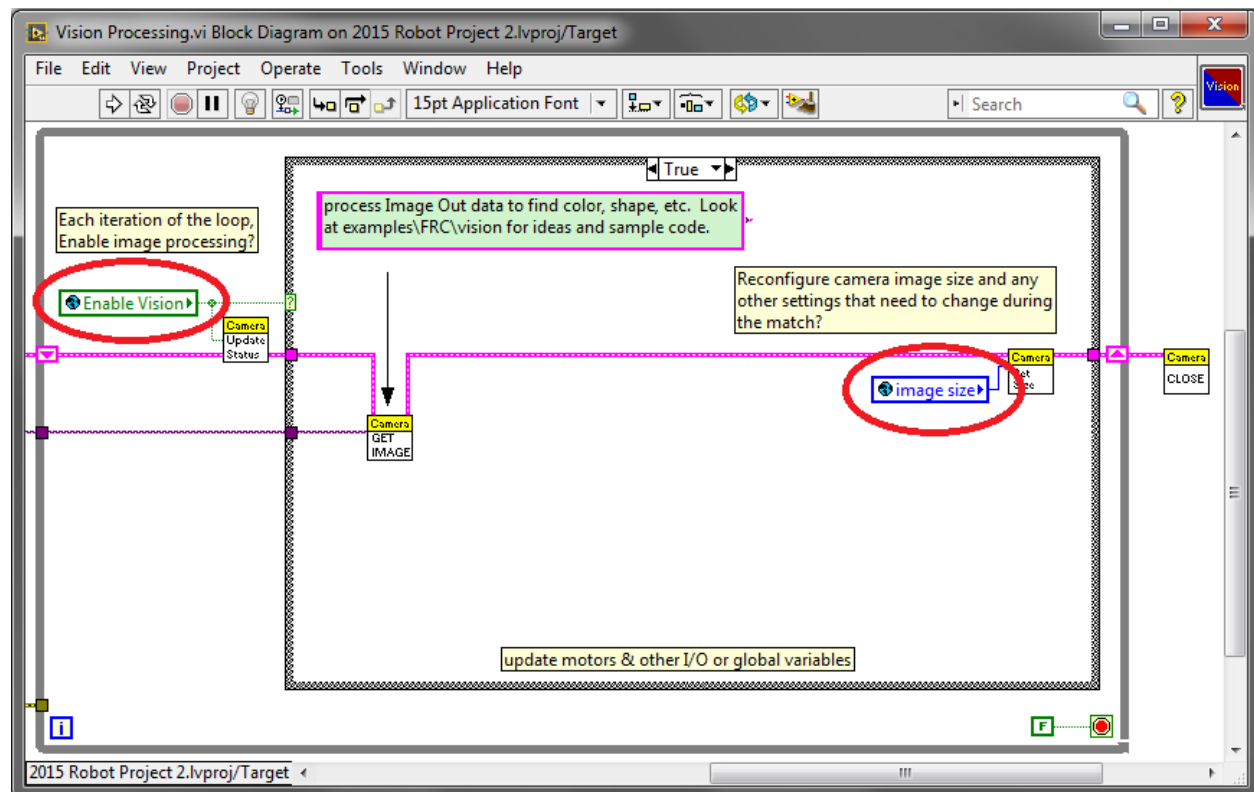
Note: Be very careful to avoid race conditions when using local and global variables! Essentially, make sure that you are not accidentally writing to the same variable in multiple locations without a way to know to which location it was last written. For a more thorough explanation, see [this help document](#)

How They are Used in the Default LabVIEW for FRC Robot Project

Global variables for “Enable Vision” and “Image Size” are written to during each iteration of the Robot Main VI...



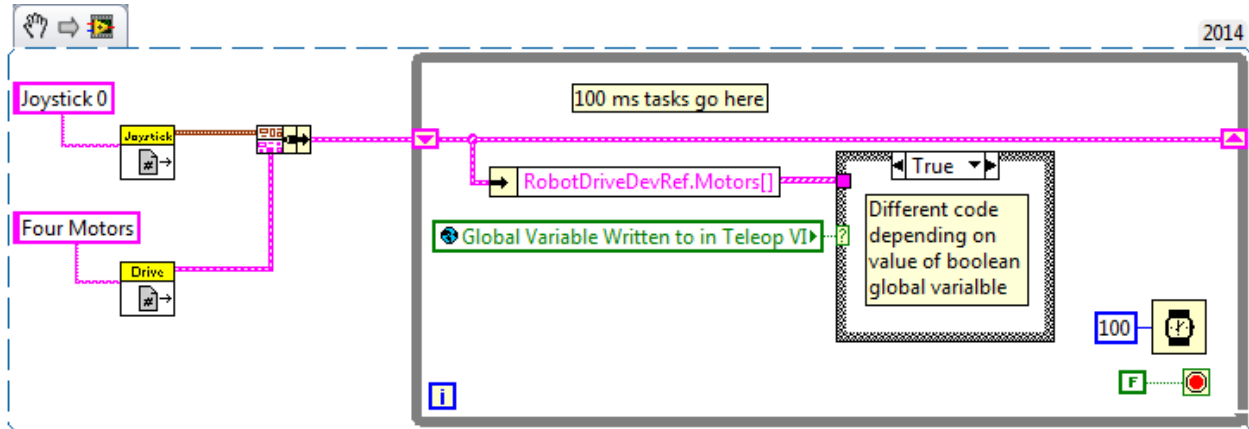
... And then read in each iteration of the Vision Processing VI:



This allows the user, when deploying to Robot Main VI from the LabVIEW Development Environment, to enable/disable vision and change the image size from Robot Main's Front Panel.

How Can You Use Them in Your Project?

Check out the block diagram for the Periodic Tasks VI. Perhaps there is some value, such as a boolean, that may be written to a global variable in the Teleop VI, and then read from in the Periodic Tasks VI. You can then decide what code or values to use in the Periodic Tasks VI, depending on the boolean global variable:



11.2.12 Using the Compressor in LabVIEW



This snippet shows how to set up your roboRIO project to use the Pneumatic Control Module (PCM). The PCM automatically starts and stops the compressor when specific pressures are measured in the tank. In your roboRIO program, you will need to add the following VIs.

For more information, check out the following links:

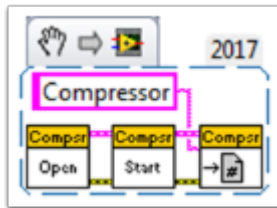
[FRC Pneumatics Manual](#)

[PCM User's Guide](#)

[Pneumatics Step by Step for the roboRIO](#)

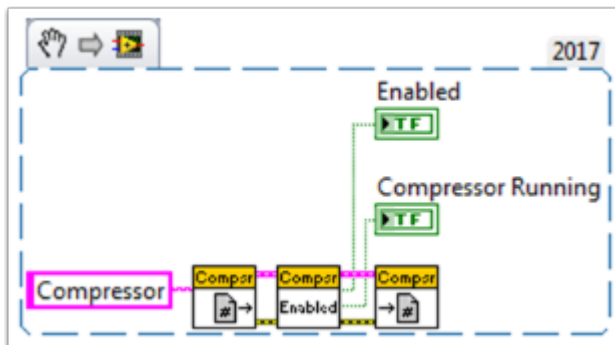
Begin VI

Place this snippet in the Begin.vi.



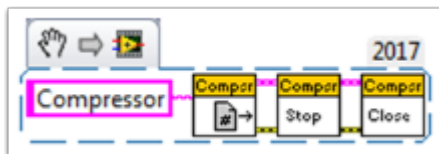
Teleop VI

Place this snippet in the Teleop.vi. This portion is only required if you are using the outputs for other processes.



Finish VI

Place this snippet in Close Refs, save data, etc. frame of the Finish.vi.



12.1 Actuator Overview

This section discusses the control of motors and pneumatics through motor controllers, solenoids and pneumatics, and their interface with C++ and Java WPILib.

12.1.1 Motor Controllers

A motor controller is responsible on your robot for making motors move. For brushed DC motors such as CIMs or 775s, the motor controller regulates the voltage that the motor receives, much like a light bulb. For brushless motor controllers such as the Spark MAX, the controller regulates the power delivered to each “phase” of the motor.

Note: Another name for a motor controller is a speed controller.

Hint: One can make a quick, non-competition-legal motor controller by removing the motor from a cordless BRUSHED drill and attaching PowerPoles or equivalents to the motor’s leads. Make sure that the voltage supplied by the drill will not damage the motor, but note that the 775 is fine at up to 24 volts.

Warning: Connecting a BRUSHLESS motor controller straight to power, such as to a conventional brushed motor controller, will destroy the motor!

FRC Legal Motor Controllers

Motor controllers come in lots of shapes, sizes and feature sets. This is the full list of FRC® Legal motor controllers as of January 2020:

- DMC 60/DMC 60c Motor Controller (P/N: 410-334-1, 410-334-2)
- Jaguar Motor Controller (P/N: MDL-BDC, MDL-BDC24, and 217-3367) connected to PWM only
- Nidec Dynamo BLDC Motor with Controller to control integral actuator only (P/N 840205-000, am-3740)
- SD540 Motor Controller (P/N: SD540x1, SD540x2, SD540x4, SD540Bx1, SD540Bx2, SD540Bx4, SD540C)
- Spark Motor Controller (P/N: REV-11-1200)
- Spark MAX Motor Controller (P/N: REV-11-2158)
- Talon FX Motor Controller (P/N: 217-6515, 19-708850, am-6515, am-6515_Short) for controlling integral Falcon 500 only
- Talon Motor Controller (P/N: CTRE_Talon, CTRE_Talon_SR, and am-2195)
- Talon SRX Motor Controller (P/N: 217-8080, am-2854, 14-838288)
- Victor 884 Motor Controller (P/N: VICTOR-884-12/12)
- Victor 888 Motor Controller (P/N: 217-2769)
- Victor SP Motor Controller (P/N: 217-9090, am-2855, 14-868380)
- Victor SPX Motor Controller (P/N: 217-9191, 17-868388, am-3748)
- Venom Motor with Controller (P/N BDC-10001) for controlling integral motor only

12.1.2 Pneumatics

Pneumatics are a quick and easy way to make something that's in one state or another using compressed air. For information on operating pneumatics, see [Operating pneumatic cylinders](#).

FRC Legal Pneumatics controllers

- Pneumatics Control Module (P/N: am-2858, 217-4243)

12.1.3 Relays

A relay controls power to a motor or custom electronics in an On/Off fashion.

FRC Legal Relay Modules

- Spike H-Bridge Relay (P/N: 217-0220 and SPIKE-RELAY-H)
- Automation Direct Relay (P/N: AD-SSR6M12-DC200D, AD-SSR6M25-DC200D, AD-SSR6M40-DC200D)

12.2 Operating pneumatic cylinders

12.2.1 Using the FRC Control System to control Pneumatics

Note: The Pneumatics Control Module (PCM) is a CAN-based device that provides complete control over the compressor and up to 8 solenoids per module. The PCM is integrated into WPILib through a series of classes that make it simple to use. The closed loop control of the Compressor and Pressure switch is handled by the PCM hardware and the Solenoids are handled by the upgraded Solenoid class that now controls the solenoid channels on the PCM. An additional PCM module can be used where the modules corresponding solenoids are differentiated by the module number in the constructors of the Solenoid and Compressor classes.



The Pneumatics Control Module from CTR Electronics is responsible for regulating the robot's pressure using a pressure switch and a compressor and switching solenoids on and off. The PCM communicates with the roboRIO over CAN. For more information, see *FRC Control System Hardware Overview*

12.2.2 PCM Module Numbers

PCM Modules are identified by their Node ID. The default Node ID for PCMs is 0. If using a single PCM on the bus it is recommended to leave it at the default Node ID.

12.2.3 Generating and Storing Pressure

In FRC®, pressure is created using a pneumatic compressor and stored in pneumatic tanks. The compressor doesn't necessarily have to be on the robot, but must be powered by the robot's PCM(s). The "Closed Loop" mode on the Compressor is enabled by default, and it is *not* recommended that teams change this setting. When closed loop control is enabled the PCM will automatically turn the compressor on when the pressure switch is closed (below the pressure threshold) and turn it off when the pressure switch is open (~120PSI). When closed loop control is disabled the compressor will not be turned on. Using a Compressor, users can query the status of the compressor. The state (currently on or off), pressure switch state, and compressor current can all be queried from the Compressor object.

Note: The Pneumatics Control Module from Cross the Road Electronics allows for integrated closed loop control of a compressor. Creating any instance of a Solenoid or Double Solenoid object will enable the Compressor control on the corresponding PCM. The Compressor object is only needed if you want the ability to turn off the compressor or query compressor status.

Java

C++

```
Compressor c = new Compressor(0);

c.setClosedLoopControl(true);
c.setClosedLoopControl(false);

boolean enabled = c.enabled();
boolean pressureSwitch = c.getPressureSwitchValue();
double current = c.getCompressorCurrent();
```

```
frc::Compressor c{0};

c.SetClosedLoopControl(true);
c.SetClosedLoopControl(false);

bool enabled = c.Enabled();
bool pressureSwitch = c.GetPressureSwitchValue();
double current = c.GetCompressorCurrent();
```

12.2.4 Solenoid control

FRC teams use solenoids to preform a variety of tasks, from shifting gearboxes to operating robot mechanisms. A solenoid is a valve used to electronically switch a pressurized air line “on” or “off”. For more information on solenoids, see [this wikipedia article](#). Solenoids are controlled by a robot’s Pneumatics Control Module, or PCM, which is in turn connected to the robot’s roboRIO via CAN. The easiest way to see a solenoid’s state is via the small red LED (which indicates if the valve is “on” or not), and solenoids can be manually actuated when un-powered with the small button adjacent to the LED.

Single acting solenoids apply or vent pressure from a single output port. They are typically used either when an external force will provide the return action of the cylinder (spring, gravity, separate mechanism) or in pairs to act as a double solenoid. A double solenoid switches air flow between two output ports (many also have a center position where neither output is vented or connected to the input). Double solenoid valves are commonly used when you wish to control both the extend and retract actions of a cylinder using air pressure. Double solenoid valves have two electrical inputs which connect back to two separate channels on the solenoid breakout.

PCM Modules are identified by their CAN Device ID. The default CAN ID for PCMs is 0. If using a single PCM on the bus it is recommended to leave it at the default CAN ID. This ID can be changed with the Phoenix Tuner application, in addition to other debug information. Instructions to download Phoenix Tuner can be found [here](#) and the installer files can be found [here](#). For more information about setting PCM CAN ID see this important [notice](#).

12.2.5 Single Solenoids in WPILib

Single solenoids in WPILib are controlled using the Solenoid class. To construct a Solenoid object, simply pass the desired port number (assumes CAN ID 0) or CAN ID and port number to the constructor. To set the value of the solenoid call `set(true)` to enable or `set(false)` to disable the solenoid output.

Java

C++

```
Solenoid exampleSolenoid = new Solenoid(1);

exampleSolenoid.set(true);
exampleSolenoid.set(false);
```

```
frc::Solenoid exampleSolenoid{1};

exampleSolenoid.Set(true);
exampleSolenoid.Set(false);
```

12.2.6 Double Solenoids in WPILib

Double solenoids are controlled by the DoubleSolenoid class in WPILib. These are constructed similarly to the single solenoid but there are now two port numbers to pass to the constructor, a forward channel (first) and a reverse channel (second). The state of the valve can then be set to kOff (neither output activated), kForward (forward channel enabled) or kReverse (reverse channel enabled). Additionally, the PCM CAN ID can be passed to the DoubleSolenoid if teams have a non-standard PCM CAN ID.

Java

C++

```
// Using "import static an.enum.or.constants.inner.class.;" helps reduce verbosity
// this replaces "DoubleSolenoid.Value.kForward" with just kForward
// further reading is available at https://www.geeksforgeeks.org/static-import-java/
import static edu.wpi.first.wpilibj.DoubleSolenoid.Value.*;
```

```
DoubleSolenoid exampleDouble = new DoubleSolenoid(1, 2);
DoubleSolenoid anotherDoubleSolenoid = new DoubleSolenoid(/* The PCM CAN ID */ 9, 4, 5);
```

```
exampleDouble.set(kOff);
exampleDouble.set(kForward);
exampleDouble.set(kReverse);
```

```
frc::DoubleSolenoid exampleDouble{1, 2};
frc::DoubleSolenoid anotherDoubleSolenoid{/* The PCM CAN ID */ 9, 1, 2};

exampleDouble.Set(frc::DoubleSolenoid::Value::kOff);
exampleDouble.Set(frc::DoubleSolenoid::Value::kForward);
exampleDouble.Set(frc::DoubleSolenoid::Value::kReverse);
```

12.2.7 Toggling Solenoids

Solenoids can be switched from one output to the other (known as toggling) by using the `.toggle()` method.

Note: Since a DoubleSolenoid defaults to off you will have to set it before it can be toggled.

Java

C++

```
Solenoid exampleSingle = new Solenoid(0);
DoubleSolenoid exampleDouble = new DoubleSolenoid(1, 2);

// Initialize the DoubleSolenoid so it knows where to start. Not required for single
↪solenoids.
exampleDouble.set(kReverse);

if (m_controller.getYButtonPressed()) {
    exampleSingle.toggle();
}
```

(continues on next page)

(continued from previous page)

```
exampleDouble.toggle();
}
```

```
frc::Solenoid exampleSingle{0};
frc::DoubleSolenoid exampleDouble{1, 2};

// Initialize the DoubleSolenoid so it knows where to start. Not required for single
↳ solenoids.
exampleDouble.Set(frc::DoubleSolenoid::Value::kReverse);

if (m_controller.GetYButtonPressed()) {
    exampleSingle.Toggle();
    exampleDouble.Toggle();
}
```

12.2.8 Pressure Transducers

One can connect a pressure transducer to measure the pressure stored in a pneumatic system. These transducers connect to the Analog Input ports on the roboRIO, and can be read by the AnalogInput or AnalogPotentiometer classes in WPILib.

Java

C++

```
import edu.wpi.first.wpilibj.AnalogInput;
import edu.wpi.first.wpilibj.AnalogPotentiometer;

// product-specific voltage->pressure conversion, see product manual
// in this case, 250(V/5)-25
// the scale parameter in the AnalogPotentiometer constructor is scaled from 1
↳ instead of 5,
// so if r is the raw AnalogPotentiometer output, the pressure is 250r-25
double scale = 250, offset = -25;
AnalogPotentiometer pressureTransducer = new AnalogPotentiometer(/* the AnalogIn
↳ port*/ 2, scale, offset);

// scaled values in psi units
double psi = pressureTransducer.get();
```

```
// product-specific voltage->pressure conversion, see product manual
// in this case, 250(V/5)-25
// the scale parameter in the AnalogPotentiometer constructor is scaled from 1
↳ instead of 5,
// so if r is the raw AnalogPotentiometer output, the pressure is 250r-25
double scale = 250, offset = -25;
frc::AnalogPotentiometer pressureTransducer{/* the AnalogIn port*/ 2, scale, offset};

// scaled values in psi units
double psi = pressureTransducer.Get();
```

12.3 Using Motor Controllers in Code

Motor controllers come in two main flavors: CAN and PWM. A CAN controller can send more detailed status information back to the roboRIO, whereas a PWM controller can only be set to a value. For information on using these motors with the WPI drivetrain classes, see [Using the WPILib Classes to Drive your Robot](#).

12.3.1 Using PWM Motor Controllers

PWM motor controllers can be controlled in the same way as a CAN motor controller. For a more detailed background on *how* they work, see [PWM Motor Controllers in Depth](#). To use a PWM motor controller, simply use the appropriate motor controller class provided by WPI and supply it the port the motor controller(s) are plugged into on the roboRIO. All approved motor controllers have WPI classes provided for them.

Java

C++

```
Spark spark = new Spark(0); // 0 is the RIO PWM port this is connected to
spark.set(-0.75); // the % output of the motor, between -1 and 1
VictorSP victor = new VictorSP(0); // 0 is the RIO PWM port this is connected to
victor.set(0.6); // the % output of the motor, between -1 and 1
```

```
frc::Spark spark{0}; // 0 is the RIO PWM port this is connected to
spark.Set(-0.75); // the % output of the motor, between -1 and 1
frc::VictorSP victor{0}; // 0 is the RIO PWM port this is connected to
victor.Set(0.6); // the % output of the motor, between -1 and 1
```

12.3.2 CAN Motor Controllers

A handful of CAN motor controllers are available through vendors such as CTR Electronics and REV Robotics.

SPARK MAX

For information regarding the SPARK MAX CAN Motor Controller, which can be used in either CAN or PWM mode, please refer to the SPARK MAX [software resources](#) and [example code](#).

CTRE CAN Motor Controllers

Please refer to the third party CTRE documentation on the Phoenix software for more detailed information. The documentation is available [here](#).

12.4 PWM Motor Controllers in Depth

Hint: WPILib has extensive support for motor control. There are a number of classes that represent different types of motor controllers and servos. There are currently two classes of motor controllers, PWM based motor controllers and CAN based motor controllers. WPILib also contains composite classes (like DifferentialDrive) which allow you to control multiple motors with a single object. This article will cover the details of PWM motor controllers; CAN controllers and composite classes will be covered in separate articles.

12.4.1 PWM Controllers, brief theory of operation

The acronym PWM stands for Pulse Width Modulation. For motor controllers, PWM can refer to both the input signal and the method the controller uses to control motor speed. To control the speed of the motor the controller must vary the perceived input voltage of the motor. To do this the controller switches the full input voltage on and off very quickly, varying the amount of time it is on based on the control signal. Because of the mechanical and electrical time constants of the types of motors used in FRC® this rapid switching produces an effect equivalent to that of applying a fixed lower voltage (50% switching produces the same effect as applying ~6V).

The PWM signal the controllers use for an input is a little bit different. Even at the bounds of the signal range (max forward or max reverse) the signal never approaches a duty cycle of 0% or 100%. Instead the controllers use a signal with a period of either 5ms or 10ms and a midpoint pulse width of 1.5ms. Many of the controllers use the typical hobby RC controller timing of 1ms to 2ms.

12.4.2 Raw vs Scaled output values

In general, all of the motor controller classes in WPILib take a scaled -1.0 to 1.0 value as the output to an actuator. The PWM module in the FPGA on the roboRIO is capable of generating PWM signals with periods of 5, 10, or 20ms and can vary the pulse width in 2000 steps of ~.001ms each around the midpoint (1000 steps in each direction around the midpoint). The raw values sent to this module are in this 0-2000 range with 0 being a special case which holds the signal low (disabled). The class for each motor controller contains information about what the typical bound values (min, max and each side of the deadband) are as well as the typical midpoint. WPILib can then use these values to map the scaled value into the proper range for the motor controller. This allows for the code to switch seamlessly between different types of controllers and abstracts out the details of the specific signaling.

12.4.3 Calibrating Motor Controllers

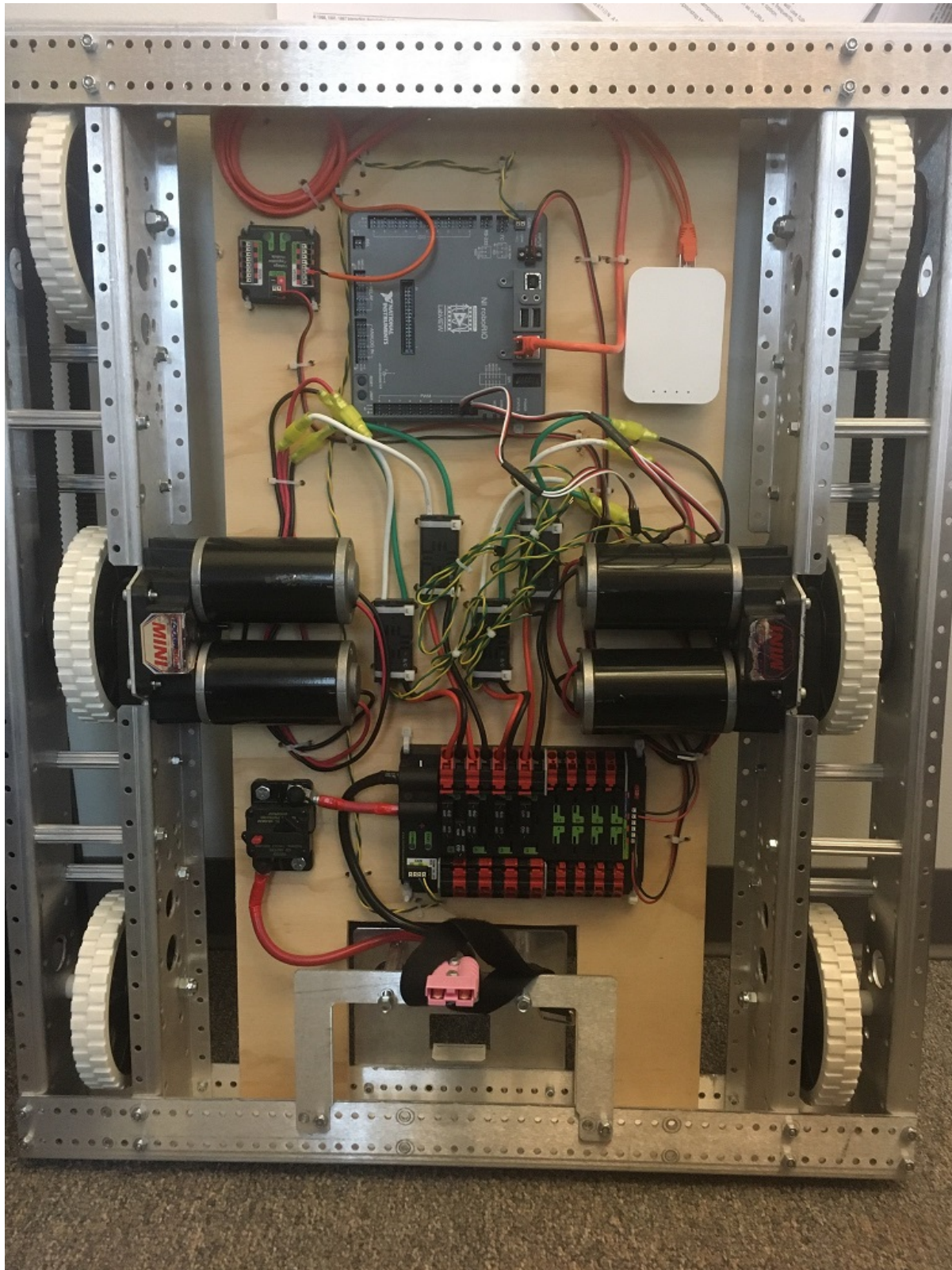
So if WPILib handles all this scaling, why would you ever need to calibrate your motor controller? The values WPILib uses for scaling are approximate based on measurement of a number of samples of each controller type. Due to a variety of factors, the timing of an individual motor controller may vary slightly. In order to definitively eliminate “humming” (midpoint signal interpreted as slight movement in one direction) and drive the controller all the way to each extreme, calibrating the controllers is still recommended. In general, the calibration procedure for each controller involves putting the controller into calibration mode then driving the input signal to each extreme, then back to the midpoint. For examples on how to use these motor controllers in your code, see [Using Motor Controllers in Code/Using PWM Motor Controllers](#)

12.5 Using the WPILib Classes to Drive your Robot

WPILib includes many classes to help make your robot get driving faster.

12.5.1 Standard drivetrains

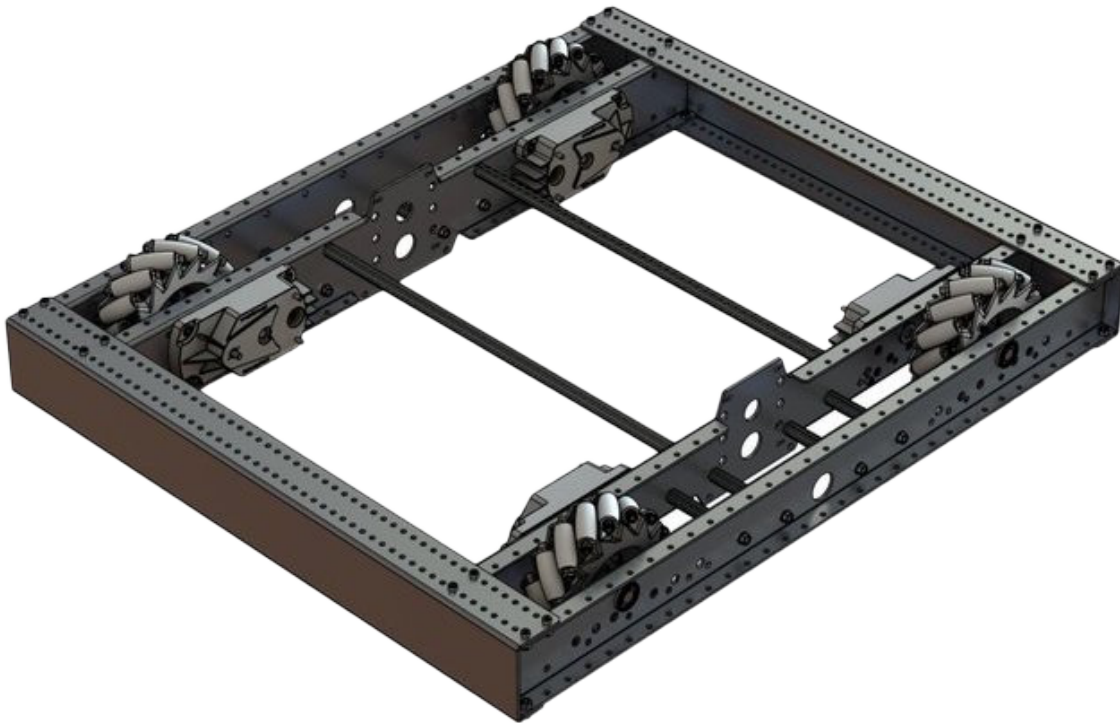
Differential Drive Robots



These drive bases typically have two or more in-line traction or omni wheels per side (e.g.,

6WD or 8WD) and may also be known as “skid-steer”, “tank drive”, or “West Coast Drive”. The Kit of Parts drivetrain is an example of a differential drive. These drivetrains are capable of driving forward/backward and can turn by driving the two sides in opposite directions causing the wheels to skid sideways. These drivetrains are not capable of sideways translational movement.

Mecanum Drive



Mecanum drive is a method of driving using specially designed wheels that allow the robot to drive in any direction without changing the orientation of the robot. A robot with a conventional drivetrain (all wheels pointing in the same direction) must turn in the direction it needs to drive. A mecanum robot can move in any direction without first turning and is called a holonomic drive. The wheels (shown on this robot) have rollers that cause the forces from driving to be applied at a 45 degree angle rather than straight forward as in the case of a conventional drive.

When viewed from the top, the rollers on a mecanum drivetrain should form an ‘X’ pattern. This results in the force vectors (when driving the wheel forward) on the front two wheels pointing forward and inward and the rear two wheels pointing forward and outward. By spinning the wheels in different directions, various components of the force vectors cancel out, resulting in the desired robot movement. A quick chart of different movements has been provided below, drawing out the force vectors for each of these motions may help in understanding how these drivetrains work. By varying the speeds of the wheels in addition to the direction, movements can be combined resulting in translation in any direction and rotation, simultaneously.

12.5.2 Drive Class Conventions

Motor Inversion

By default, the class inverts the motor outputs for the right side of the drivetrain. Generally this will mean that no inversion needs to be done on the individual `SpeedController` objects. To disable this behavior, use the `setRightSideInverted()` method.

Squaring Inputs

When driving robots, it is often desirable to manipulate the joystick inputs such that the robot has finer control at low speeds while still using the full output range. One way to accomplish this is by squaring the joystick input, then reapplying the sign. By default the Differential Drive class will square the inputs. If this is not desired (e.g. if passing values in from a `PIDController`), use one of the drive methods with the `squaredInputs` parameter and set it to false.

Input Deadband

By default, the Differential Drive class applies an input deadband of 0.02. This means that input values with a magnitude below 0.02 (after any squaring as described above) will be set to 0. In most cases these small inputs result from imperfect joystick centering and are not sufficient to cause drivetrain movement, the deadband helps reduce unnecessary motor heating that may result from applying these small values to the drivetrain. To change the deadband, use the `setDeadband()` method.

Maximum Output

Sometimes drivers feel that their drivetrain is driving too fast and want to limit the output. This can be accomplished with the `setMaxOutput()` method. This maximum output is multiplied by result of the previous drive functions like deadband and squared inputs.

Motor Safety

Motor Safety is a mechanism in WPILib that takes the concept of a watchdog and breaks it out into one watchdog (Motor Safety timer) for each individual actuator. Note that this protection mechanism is in addition to the System Watchdog which is controlled by the Network Communications code and the FPGA and will disable all actuator outputs if it does not receive a valid data packet for 125ms.

The purpose of the Motor Safety mechanism is the same as the purpose of a watchdog timer, to disable mechanisms which may cause harm to themselves, people or property if the code locks up and does not properly update the actuator output. Motor Safety breaks this concept out on a per actuator basis so that you can appropriately determine where it is necessary and where it is not. Examples of mechanisms that should have motor safety enabled are systems like drive trains and arms. If these systems get latched on a particular value they could cause damage to their environment or themselves. An example of a mechanism that may not need motor safety is a spinning flywheel for a shooter. If this mechanism gets latched on a particular value it will simply continue spinning until the robot is disabled. By default

Motor Safety is enabled for RobotDrive, DifferentialDrive, KilloughDrive, and MecanumDrive objects and disabled for all other motor controllers and servos.

The Motor Safety feature operates by maintaining a timer that tracks how long it has been since the feed() method has been called for that actuator. Code in the Driver Station class initiates a comparison of these timers to the timeout values for any actuator with safety enabled every 5 received packets (100ms nominal). The set() methods of each motor controller class and the set() and setAngle() methods of the servo class call feed() to indicate that the output of the actuator has been updated.

The Motor Safety interface of motor controllers can be interacted with by the user using the following methods:

Java

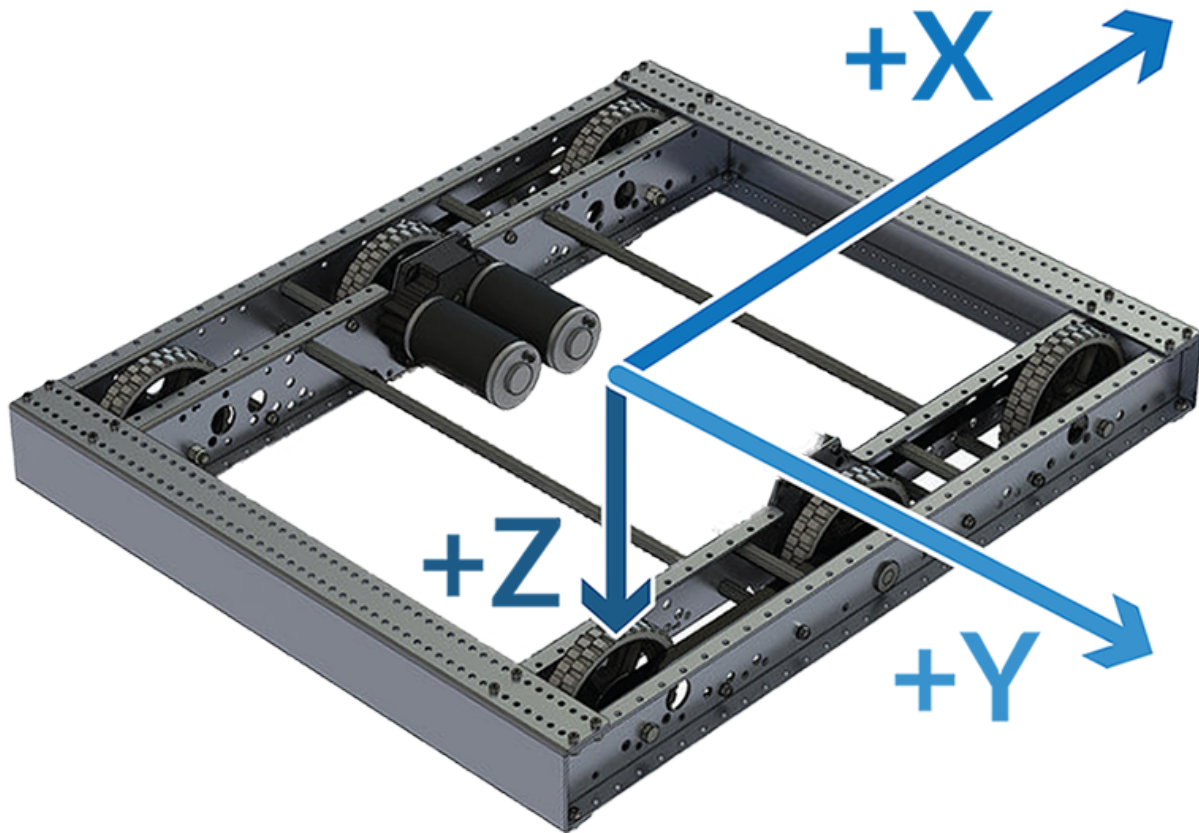
C++

```
exampleJaguar.setSafetyEnabled(true);  
exampleJaguar.setSafetyEnabled(false);  
exampleJaguar.setExpiration(.1);  
exampleJaguar.feed();
```

```
exampleJaguar->SetSafetyEnabled(true);  
exampleJaguar->SetSafetyEnabled(false);  
exampleJaguar->SetExpiration(.1);  
exampleJaguar->Feed();
```

By default all RobotDrive objects enable Motor Safety. Depending on the mechanism and the structure of your program, you may wish to configure the timeout length of the motor safety (in seconds). The timeout length is configured on a per actuator basis and is not a global setting. The default (and minimum useful) value is 100ms.

Axis Conventions



This library uses the NED axes convention (North-East-Down as external reference in the world frame). The positive X axis points ahead, the positive Y axis points right, and the positive Z axis points down. Rotations follow the right-hand rule, so clockwise rotation around the Z axis is positive.

Warning: This convention is different than the convention for joysticks which typically have -Y as Up (commonly mapped to throttle) and +X as Right. Pay close attention to the examples below if you want help with typical Joystick->Drive mapping.

12.5.3 Using the DifferentialDrive class to control Differential Drive robots

Note: WPILib provides separate Robot Drive classes for the most common drive train configurations (differential, mecanum, and Killough). The DifferentialDrive class handles the differential drivetrain configuration. These drive bases typically have two or more in-line traction or omni wheels per side (e.g., 6WD or 8WD) and may also be known as “skid-steer”, “tank drive”, or “West Coast Drive” (WCD). The Kit of Parts drivetrain is an example of a differential drive. There are methods to control the drive with 3 different styles (“Tank”, “Arcade”, or “Curvature”), explained in the article below.

DifferentialDrive is a method provided for the control of “skid-steer” or “West Coast” drivetrains, such as the Kit of Parts chassis. Instantiating a DifferentialDrive is as simple as so:

Java

C++ (Header)

C++ (Source)

```
public class Robot {
    Spark m_left = new Spark(1);
    Spark m_right = new Spark(2);
    DifferentialDrive m_drive = new DifferentialDrive(m_left, m_right);

    public void robotInit() {
        m_left.setInverted(true); // if you want to invert motor outputs, you must do so
        ↪so here
    }
}
```

```
class Robot {
    private:
        frc::Spark m_left{1};
        frc::Spark m_right{2};
        frc::DifferentialDrive m_drive{m_left, m_right};
}
```

```
void Robot::RobotInit() {
    m_left.SetInverted(true); // if you want to invert motor outputs, you must do so
    ↪here
}
```

Multi-Motor DifferentialDrive with SpeedControllerGroups

Many FRC® drivetrains have more than 1 motor on each side. In order to use these with DifferentialDrive, the motors on each side have to be collected into a single SpeedController, using the SpeedControllerGroup class. The examples below show a 4 motor (2 per side) drivetrain. To extend to more motors, simply create the additional controllers and pass them all into the SpeedController group constructor (it takes an arbitrary number of inputs).

Java

C++ (Header)

C++ (Source)

```
public class Robot {
    Spark m_frontLeft = new Spark(1);
    Spark m_rearLeft = new Spark(2);
    SpeedControllerGroup m_left = new SpeedControllerGroup(m_frontLeft, m_rearLeft);

    Spark m_frontRight = new Spark(3);
    Spark m_rearRight = new Spark(4);
    SpeedControllerGroup m_right = new SpeedControllerGroup(m_frontRight, m_
    ↪rearRight);
    DifferentialDrive m_drive = new DifferentialDrive(m_left, m_right);

    public void robotInit() {
}
```

(continues on next page)

(continued from previous page)

```

        m_left.setInverted(true); // if you want to invert the entire side you can do
↪so here
    }

```

```

class Robot {
public:
    frc::Spark m_frontLeft{1};
    frc::Spark m_rearLeft{2};
    frc::SpeedControllerGroup m_left{m_frontLeft, m_rearLeft};

    frc::Spark m_frontRight{3};
    frc::Spark m_rearRight{4};
    frc::SpeedControllerGroup m_right{m_frontRight, m_rearRight};

    frc::DifferentialDrive m_drive{m_left, m_right};

```

```

void Robot::RobotInit() {
    m_left.SetInverted(true); // if you want to invert the entire side you can do so
↪here
}

```

Drive Modes

Note: The `DifferentialDrive` class contains three different default modes of driving your robot's motors.

- Tank Drive, which controls the left and right side independently
- Arcade Drive, which controls a forward and turn speed
- Curvature Drive, a subset of Arcade Drive, which makes your robot handle like a car with constant-curvature turns.

The `DifferentialDrive` class contains three default methods for controlling skid-steer or WCD robots. Note that you can create your own methods of controlling the robot's driving and have them call `tankDrive()` with the derived inputs for left and right motors.

The Tank Drive mode is used to control each side of the drivetrain independently (usually with an individual joystick axis controlling each). This example shows how to use the Y-axis of two separate joysticks to run the drivetrain in Tank mode. Construction of the objects has been omitted, for above for drivetrain construction and here for Joystick construction.

The Arcade Drive mode is used to control the drivetrain using speed/throttle and rotation rate. This is typically used either with two axes from a single joystick, or split across joysticks (often on a single gamepad) with the throttle coming from one stick and the rotation from another. This example shows how to use a single joystick with the Arcade mode. Construction of the objects has been omitted, for above for drivetrain construction and here for Joystick construction.

Like Arcade Drive, the Curvature Drive mode is used to control the drivetrain using speed/throttle and rotation rate. The difference is that the rotation control input controls the radius of curvature instead of rate of heading change, much like the steering wheel of a car. This mode also supports turning in place, which is enabled when the third boolean parameter is true.

Java

C++

```

public void teleopPeriodic() {
    // Tank drive with a given left and right rates
    myDrive.tankDrive(-leftStick.getY(), -rightStick.getY());

    // Arcade drive with a given forward and turn rate
    myDrive.arcadeDrive(-driveStick.getY(), driveStick.getX());

    // Curvature drive with a given forward and turn rate, as well as a quick-turn
    ↪ button
    myDrive.curvatureDrive(-driveStick.getY(), driveStick.getX(), driveStick.
    ↪ getButton(1));
}

```

```

void TeleopPeriodic() override {
    // Tank drive with a given left and right rates
    myDrive.TankDrive(-leftStick.GetY(), -rightStick.GetY());

    // Arcade drive with a given forward and turn rate
    myDrive.ArcadeDrive(-driveStick.GetY(), driveStick.GetX());

    // Curvature drive with a given forward and turn rate, as well as a quick-turn
    ↪ button
    myDrive.CurvatureDrive(-driveStick.GetY(), driveStick.GetX(), driveStick.
    ↪ GetButton(1));
}

```

12.5.4 Using the MecanumDrive class to control Mecanum Drive robots

MecanumDrive is a method provided for the control of holonomic drivetrains with Mecanum wheels, such as the Kit of Parts chassis with the mecanum drive upgrade kit, as shown above. Instantiating a MecanumDrive is as simple as so:

Java

C++

```

31 public void robotInit() {
32     PWMVictorSPX frontLeft = new PWMVictorSPX(kFrontLeftChannel);
33     PWMVictorSPX rearLeft = new PWMVictorSPX(kRearLeftChannel);
34     PWMVictorSPX frontRight = new PWMVictorSPX(kFrontRightChannel);
35     PWMVictorSPX rearRight = new PWMVictorSPX(kRearRightChannel);
36
37     // Invert the left side motors.
38     // You may need to change or remove this to match your robot.
39     frontLeft.setInverted(true);
40     rearLeft.setInverted(true);
41
42     m_robotDrive = new MecanumDrive(frontLeft, rearLeft, frontRight, rearRight);
43
44     m_stick = new Joystick(kJoystickChannel);
45 }

```

```

33 private:
34     static constexpr int kFrontLeftChannel = 0;
35     static constexpr int kRearLeftChannel = 1;
36     static constexpr int kFrontRightChannel = 2;
37     static constexpr int kRearRightChannel = 3;
38
39     static constexpr int kJoystickChannel = 0;
40
41     frc::PWMVictorSPX m_frontLeft{kFrontLeftChannel};
42     frc::PWMVictorSPX m_rearLeft{kRearLeftChannel};
43     frc::PWMVictorSPX m_frontRight{kFrontRightChannel};
44     frc::PWMVictorSPX m_rearRight{kRearRightChannel};
45     frc::MecanumDrive m_robotDrive{m_frontLeft, m_rearLeft, m_frontRight,
46                                     m_rearRight};

```

Mecanum Drive Modes

Note: The drive axis conventions are different from common joystick axis conventions. See the [Axis Conventions](#) above for more information.

The MecanumDrive class contains two different default modes of driving your robot's motors.

- **driveCartesian:** Angles are measured clockwise from the positive X axis. The robot's speed is independent from its angle or rotation rate.
- **drivePolar:** Angles are measured counter-clockwise from straight ahead. The speed at which the robot drives (translation) is independent from its angle or rotation rate.

Java

C++

```

public void teleopPeriodic() {
    m_robotDrive.driveCartesian(m_stick.getX(), -m_stick.getY(), m_stick.getZ());
    m_robotDrive.drivePolar(m_stick.getX(), -m_stick.getY(), m_stick.getZ());
}

```

```

void TeleopPeriodic() override {
    m_robotDrive.driveCartesian(m_stick.GetX(), -m_stick.GetY(), m_stick.GetZ());
    m_robotDrive.drivePolar(m_stick.GetX(), -m_stick.GetY(), m_stick.GetZ());
}

```

Field-Oriented Driving

A 4th parameter can be supplied to the `driveCartesian(double ySpeed, double xSpeed, double zRotation, double gyroAngle)` method, the angle returned from a Gyro sensor. This will adjust the rotation value supplied. This is particularly useful with mecanum drive since, for the purposes of steering, the robot really has no front, back or sides. It can go in any direction. Adding the angle in degrees from a gyro object will cause the robot to move away from the drivers when the joystick is pushed forwards, and towards the drivers when it is pulled towards them, regardless of what direction the robot is facing.

The use of field-oriented driving makes often makes the robot much easier to drive, especially compared to a “robot-oriented” drive system where the controls are reversed when the robot is facing the drivers.

Just remember to get the gyro angle each time `driveCartesian()` is called.

Note: Many teams also like to ramp the joysticks inputs over time to promote a smooth acceleration and reduce jerk. This can be accomplished with a [Slew Rate Limiter](#).

12.6 Repeatable Low Power Movement - Controlling Servos with WPILib

Servo motors are a type of motor which integrates positional feedback into the motor in order to allow a single motor to perform repeatable, controllable movement, taking position as the input signal. WPILib provides the capability to control servos which match the common hobby input specification (PWM signal, 1.0ms-2.0ms pulse width)

12.6.1 Constructing a Servo object

Java

C++

```
Servo exampleServo = new Servo(1);
```

```
frc::Servo exampleServo {1};
```

A servo object is constructed by passing a channel.

12.6.2 Setting Servo Values

Java

C++

```
exampleServo.set(.5);  
exampleServo.setAngle(75);
```

```
exampleServo.Set(.5);  
exampleServo.SetAngle(75);
```

There are two methods of setting servo values in WPILib:

- Scaled Value - Sets the servo position using a scaled 0 to 1.0 value. 0 corresponds to one extreme of the servo and 1.0 corresponds to the other
- Angle - Set the servo position by specifying the angle, in degrees. This method will work for servos with the same range as the Hitec HS-322HD servo (0 to 170 degrees). Any values passed to this method outside the specified range will be coerced to the boundary.

12.7 Addressable LEDs

LED strips have been commonly used by teams for several years for a variety of reasons. They allow teams to debug robot functionality from the audience, provide a visual marker for their robot, and can simply add some visual appeal. WPILib has an API for controlling WS2812 LEDs with their data pin connected via PWM.

12.7.1 Instantiating the AddressableLED Object

You first create an `AddressableLED` object that takes the PWM port as an argument. It *must* be a PWM header on the roboRIO. Then you set the number of LEDs located on your LED strip, with can be done with the `setLength()` function.

Important: It is important to note that setting the length of the LED header is an expensive task and it's **not** recommended to run this periodically.

After the length of the strip has been set, you'll have to create an `AddressableLEDBuffer` object that takes the number of LEDs as an input. You'll then call `myAddressableLed.setData(myAddressableLEDBuffer)` to set the led output data. Finally, you can call `myAddressableLed.start()` to write the output continuously. Below is a full example of the initialization process.

Note: C++ does not have an `AddressableLEDBuffer`, and instead uses an `Array`.

Java

C++

```

21 public void robotInit() {
22     // PWM port 9
23     // Must be a PWM header, not MXP or DIO
24     m_led = new AddressableLED(9);
25
26     // Reuse buffer
27     // Default to a length of 60, start empty output
28     // Length is expensive to set, so only set it once, then just update data
29     m_ledBuffer = new AddressableLEDBuffer(60);
30     m_led.setLength(m_ledBuffer.getLength());
31
32     // Set the data
33     m_led.setData(m_ledBuffer);
34     m_led.start();
35 }
```

```

14 class Robot : public frc::TimedRobot {
15     static constexpr int kLength = 60;
16
17     // PWM port 9
18     // Must be a PWM header, not MXP or DIO
19     frc::AddressableLED m_led{9};
20     std::array<frc::AddressableLED::LEDData, kLength>
```

(continues on next page)

(continued from previous page)

```
21     m_ledBuffer; // Reuse the buffer
22     // Store what the last hue of the first pixel is
23     int firstPixelHue = 0;

41     void RobotInit() override {
42         // Default to a length of 60, start empty output
43         // Length is expensive to set, so only set it once, then just update data
44         m_led.SetLength(kLength);
45         m_led.SetData(m_ledBuffer);
46         m_led.Start();
47     }
```

12.7.2 Setting the Entire Strip to One Color

Color can be set to an individual led on the strip using two methods. `setRGB()` which takes RGB values as an input and `setHSV()` which takes HSV values as an input.

Using RGB Values

RGB stands for Red, Green, and Blue. This is a fairly common color model as it's quite easy to understand. LEDs can be set with the `setRGB` method that takes 4 arguments: index of the LED, amount of red, amount of green, amount of blue. The amount of Red, Green, and Blue are integer values between 0-255.

Java

C++

```
for (var i = 0; i < m_ledBuffer.getLength(); i++) {
    // Sets the specified LED to the RGB values for red
    m_ledBuffer.setRGB(i, 255, 0, 0);
}

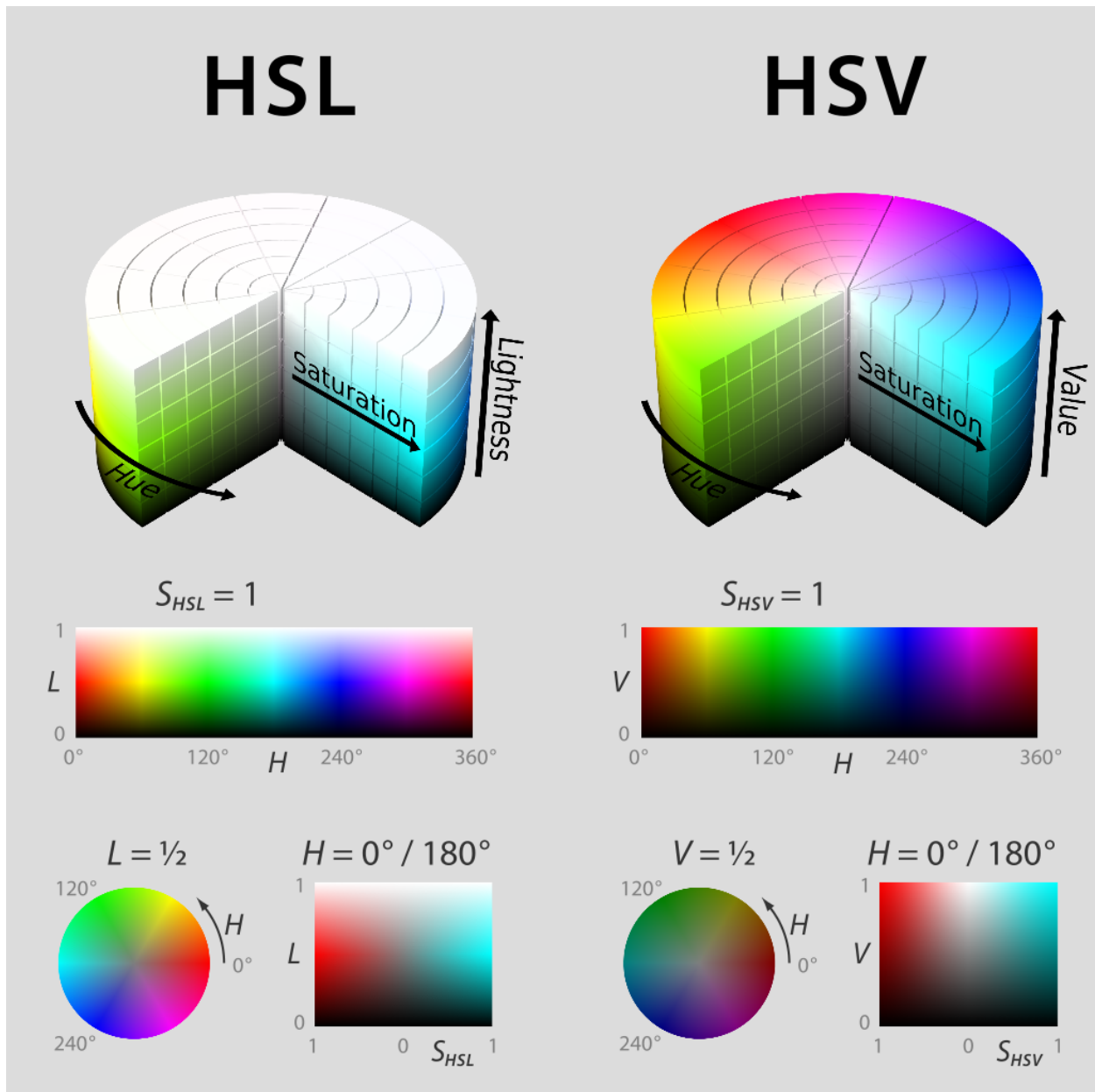
m_led.setData(m_ledBuffer);
```

```
for (int i = 0; i < kLength; i++) {
    m_ledBuffer[i].SetRGB(255, 0, 0);
}

m_led.SetData(m_ledBuffer);
```

Using HSV Values

HSV stands for Hue, Saturation, and Value. Hue describes the color or tint, saturation being the amount of gray, and value being the brightness. In WPILib, Hue is an integer from 0 - 180. Saturation and Value are integers from 0 - 255. If you look at a color picker like [Google's](#), Hue will be 0 - 360 and Saturation and Value are from 0% to 100%. This is the same way that OpenCV handles HSV colors. Make sure the HSV values entered to WPILib are correct, or the color produced might not be the same as was expected.



LEDs can be set with the `setHSV` method that takes 4 arguments: index of the LED, hue, saturation, and value. An example is shown below for setting the color of an LED strip to red (hue of 0).

Java

C++

```
for (var i = 0; i < m_ledBuffer.getLength(); i++) {
    // Sets the specified LED to the HSV values for red
    m_ledBuffer.setHSV(i, 0, 100, 100);
}

m_led.setData(m_ledBuffer);
```

```

for (int i = 0; i < kLength; i++) {
    m_ledBuffer[i].SetHSV(0, 100, 100);
}

m_led.SetData(m_ledBuffer);

```

12.7.3 Creating a Rainbow Effect

The below method does a couple of important things. Inside of the *for* loop, it equally distributes the hue over the entire length of the strand and stores the individual LED hue to a variable called *hue*. Then the *for* loop sets the HSV value of that specified pixel using the *hue* value.

Moving outside of the *for* loop, the *m_rainbowFirstPixelHue* then iterates the pixel that contains the “initial” hue creating the rainbow effect. *m_rainbowFirstPixelHue* then checks to make sure that the *hue* is inside the *hue* boundaries of 180. This is because HSV *hue* is a value from 0-180.

Note: It’s good robot practice to keep the *robotPeriodic()* method as clean as possible, so we’ll create a method for handling setting our LED data. We’ll call this method *rainbow()* and call it from *robotPeriodic()*.

Java

C++

```

45 private void rainbow() {
46     // For every pixel
47     for (var i = 0; i < m_ledBuffer.getLength(); i++) {
48         // Calculate the hue - hue is easier for rainbows because the color
49         // shape is a circle so only one value needs to precess
50         final var hue = (m_rainbowFirstPixelHue + (i * 180 / m_ledBuffer.getLength()))
↪ % 180;
51         // Set the value
52         m_ledBuffer.setHSV(i, hue, 255, 128);
53     }
54     // Increase by to make the rainbow "move"
55     m_rainbowFirstPixelHue += 3;
56     // Check bounds
57     m_rainbowFirstPixelHue %= 180;
58 }

```

```

26 void Rainbow() {
27     // For every pixel
28     for (int i = 0; i < kLength; i++) {
29         // Calculate the hue - hue is easier for rainbows because the color
30         // shape is a circle so only one value needs to precess
31         const auto pixelHue = (firstPixelHue + (i * 180 / kLength)) % 180;
32         // Set the value
33         m_ledBuffer[i].SetHSV(pixelHue, 255, 128);
34     }
35     // Increase by to make the rainbow "move"
36     firstPixelHue += 3;

```

(continues on next page)

(continued from previous page)

```
37 // Check bounds
38 firstPixelHue %= 180;
39 }
```

Now that we have our rainbow method created, we have to actually call the method and set the data of the LED.

Java

C++

```
37 @Override
38 public void robotPeriodic() {
39     // Fill the buffer with a rainbow
40     rainbow();
41     // Set the LEDs
42     m_led.setData(m_ledBuffer);
43 }
```

```
49 void RobotPeriodic() override {
50     // Fill the buffer with a rainbow
51     Rainbow();
52     // Set the LEDs
53     m_led.SetData(m_ledBuffer);
54 }
55 };
```


13.1 Sensor Overview - Software

Note: This section covers using sensors in software. For a guide to sensor hardware, see [Sensor Overview - Hardware](#).

Note: While cameras may definitely be considered “sensors”, vision processing is a sufficiently-complicated subject that it is covered in [its own section](#), rather than here.

In order to be effective, it is often vital for robots to be able to gather information about their surroundings. Devices that provide feedback to the robot on the state of its environment are called “sensors.” WPILib innately supports a large variety of sensors through classes included in the library. This section will provide a guide to both using common sensor types through WPILib, as well as writing code for sensors without official support.

13.1.1 What sensors does WPILIB support?

The roboRIO includes a [FPGA](#) which allows accurate real-time measuring of a variety of sensor input. WPILib, in turn, provides a number of classes for accessing this functionality.

WPILib provides native support for:

- [Accelerometers](#)
- [Gyroscopes](#)
- [Ultrasonic rangefinders](#)
- [Potentiometers](#)
- [Counters](#)
- [Quadrature encoders](#)
- [Limit switches](#)

Additionally, WPILib includes lower-level classes for interfacing directly with the FPGA's digital and analog inputs and outputs.

13.2 Accelerometers - Software

Note: This section covers accelerometers in software. For a hardware guide to accelerometers, see [Accelerometers - Hardware](#).

An accelerometer is a device that measures acceleration.

Accelerometers generally come in two types: single-axis and 3-axis. A single-axis accelerometer measures acceleration along one spatial dimension; a 3-axis accelerometer measures acceleration along all three spatial dimensions at once.

WPILib supports single-axis accelerometers through the [AnalogAccelerometer](#) class.

Three-axis accelerometers often require more complicated communications protocols (such as SPI or I2C) in order to send multi-dimensional data. WPILib has native support for the following 3-axis accelerometers:

- [ADXL345_I2C](#)
- [ADXL345_SPI](#)
- [ADXL362](#)
- [BuiltInAccelerometer](#)

13.2.1 AnalogAccelerometer

The AnalogAccelerometer class (Java, C++) allows users to read values from a single-axis accelerometer that is connected to one of the roboRIO's analog inputs.

Java

C++

```
// Creates an analog accelerometer on analog input 0
AnalogAccelerometer accelerometer = new AnalogAccelerometer(0);

// Sets the sensitivity of the accelerometer to 1 volt per G
accelerometer.setSensitivity(1);

// Sets the zero voltage of the accelerometer to 3 volts
accelerometer.setZero(3);

// Gets the current acceleration
double accel = accelerometer.getAcceleration();
```

```
// Creates an analog accelerometer on analog input 0
frc::AnalogAccelerometer accelerometer{0};

// Sets the sensitivity of the accelerometer to 1 volt per G
accelerometer.SetSensitivity(1);
```

(continues on next page)

(continued from previous page)

```
// Sets the zero voltage of the accelerometer to 3 volts
accelerometer.SetZero(3);

// Gets the current acceleration
double accel = accelerometer.GetAcceleration();
```

If users have a 3-axis analog accelerometer, they can use three instances of this class, one for each axis.

13.2.2 The Accelerometer interface

All 3-axis accelerometers in WPILib implement the Accelerometer interface (Java, C++). This interface defines functionality and settings common to all supported 3-axis accelerometers.

The Accelerometer interface contains getters for the acceleration along each cardinal direction (x, y, and z), as well as a setter for the range of accelerations the accelerometer will measure.

Warning: Not all accelerometers are capable of measuring all ranges.

Java

C++

```
// Sets the accelerometer to measure between -8 and 8 G's
accelerometer.setRange(Accelerometer.Range.k8G);
```

```
// Sets the accelerometer to measure between -8 and 8 G's
accelerometer.SetRange(Accelerometer::Range::kRange_8G);
```

ADXL345_I2C

The ADXL345_I2C class (Java, C++) provides support for the ADXL345 accelerometer over the I2C communications bus.

Java

C++

```
// Creates an ADXL345 accelerometer object on the MXP I2C port
// with a measurement range from -8 to 8 G's
Accelerometer accelerometer = new ADXL345_I2C(I2C.Port.kMXP, Accelerometer.Range.k8G);
```

```
// Creates an ADXL345 accelerometer object on the MXP I2C port
// with a measurement range from -8 to 8 G's
frc::ADXL345_I2C accelerometer{I2C::Port::kMXP, Accelerometer::Range::kRange_8G};
```

ADXL345_SPI

The ADXL345_SPI class (Java, C++) provides support for the ADXL345 accelerometer over the SPI communications bus.

Java

C++

```
// Creates an ADXL345 accelerometer object on the MXP SPI port  
// with a measurement range from -8 to 8 G's  
Accelerometer accelerometer = new ADXL345_SPI(SPI.Port.kMXP, Accelerometer.Range.k8G);
```

```
// Creates an ADXL345 accelerometer object on the MXP SPI port  
// with a measurement range from -8 to 8 G's  
frc::ADXL345_SPI accelerometer{SPI::Port::kMXP, Accelerometer::Range::kRange_8G};
```

ADXL362

The ADXL362 class (Java, C++) provides support for the ADXL362 accelerometer over the SPI communications bus.

Java

C++

```
// Creates an ADXL362 accelerometer object on the MXP SPI port  
// with a measurement range from -8 to 8 G's  
Accelerometer accelerometer = new ADXL362(SPI.Port.kMXP, Accelerometer.Range.k8G);
```

```
// Creates an ADXL362 accelerometer object on the MXP SPI port  
// with a measurement range from -8 to 8 G's  
frc::ADXL362 accelerometer{SPI::Port::kMXP, Accelerometer::Range::kRange_8G};
```

BuiltInAccelerometer

The BuiltInAccelerometer class (Java, C++) provides access to the roboRIO's own built-in accelerometer:

Java

C++

```
// Creates an object for the built-in accelerometer  
// Range defaults to +- 8 G's  
Accelerometer accelerometer = new BuiltInAccelerometer();
```

```
// Creates an object for the built-in accelerometer  
// Range defaults to +- 8 G's  
frc::BuiltInAccelerometer accelerometer{};
```


13.2.3 Third-party accelerometers

While WPILib provides native support for a number of accelerometers that are available in the kit of parts or through FIRST Choice, there are a few popular AHRS (Attitude and Heading Reference System) devices commonly used in FRC that include accelerometers. These are generally controlled through vendor libraries, though if they have a simple analog output they can be used with the *AnalogAccelerometer* class.

13.2.4 Using accelerometers in code

Note: Accelerometers, as their name suggests, measure acceleration. Precise accelerometers can be used to determine position through double-integration (since acceleration is the second derivative of position), much in the way that gyroscopes are used to determine heading. However, the accelerometers available for use in FRC are not nearly high-enough quality to be used this way.

It is recommended to use accelerometers in FRC® for any application which needs a rough measurement of the current acceleration. This can include detecting collisions with other robots or field elements, so that vulnerable mechanisms can be automatically retracted. They may also be used to determine when the robot is passing over rough terrain for an autonomous routine (such as traversing the defenses in FIRST Stronghold).

For detecting collisions, it is often more robust to measure the jerk than the acceleration. The jerk is the derivative (or rate of change) of acceleration, and indicates how rapidly the forces on the robot are changing - the sudden impulse from a collision causes a sharp spike in the jerk. Jerk can be determined by simply taking the difference of subsequent acceleration measurements, and dividing by the time between them:

Java

C++

```
double prevXAccel = 0;
double prevYAccel = 0;

Accelerometer accelerometer = new BuiltInAccelerometer();

@Override
public void robotPeriodic() {
    // Gets the current accelerations in the X and Y directions
    double xAccel = accelerometer.getX();
    double yAccel = accelerometer.getY();

    // Calculates the jerk in the X and Y directions
    // Divides by .02 because default loop timing is 20ms
    double xJerk = (xAccel - prevXAccel)/.02;
    double yJerk = (yAccel - prevYAccel)/.02;

    prevXAccel = xAccel;
    prevYAccel = yAccel;
}
```

```

double prevXAccel = 0;
double prevYAccel = 0;

frc::BuiltInAccelerometer accelerometer{};

void Robot::RobotPeriodic() {
    // Gets the current accelerations in the X and Y directions
    double xAccel = accelerometer.GetX();
    double yAccel = accelerometer.GetY();

    // Calculates the jerk in the X and Y directions
    // Divides by .02 because default loop timing is 20ms
    double xJerk = (xAccel - prevXAccel)/.02;
    double yJerk = (yAccel - prevYAccel)/.02;

    prevXAccel = xAccel;
    prevYAccel = yAccel;
}

```

Most accelerometers legal for FRC use are quite noisy, and it is often a good idea to combine them with the `LinearFilter` class (Java, C++) to reduce the noise:

Java

C++

```

Accelerometer accelerometer = new BuiltInAccelerometer();

// Create a LinearDigitalFilter that will calculate a moving average of the measured
// X acceleration over the past 10 iterations of the main loop

LinearDigitalFilter xAccelFilter = LinearDigitalFilter.movingAverage(10);

@Override
public void robotPeriodic() {
    // Get the filtered X acceleration
    double filteredXAccel = xAccelFilter.calculate(accelerometer.getX());
}

```

```

frc::BuiltInAccelerometer accelerometer;

// Create a LinearDigitalFilter that will calculate a moving average of the measured
// X acceleration over the past 10 iterations of the main loop
auto xAccelFilter = frc::LinearDigitalFilter::MovingAverage(10);

void Robot::RobotPeriodic() {
    // Get the filtered X acceleration
    double filteredXAccel = xAccelFilter.Calculate(accelerometer.GetX());
}

```

13.3 Gyroscopes - Software

Note: This section covers gyros in software. For a hardware guide to gyros, see [Gyroscopes - Hardware](#).

A gyroscope, or “gyro,” is an angular rate sensor typically used in robotics to measure and/or stabilize robot headings. WPILib natively provides specific support for the ADXRS450 gyro available in the kit of parts, as well as more general support for a wider variety of analog gyros through the [AnalogGyro](#) class.

13.3.1 The Gyro interface

All natively-supported gyro objects in WPILib implement the Gyro interface ([Java](#), [C++](#)). This interface provides methods for getting the current angular rate and heading, zeroing the current heading, and calibrating the gyro.

Note: It is crucial that the robot remain stationary while calibrating a gyro.

ADXRS450_Gyro

The ADXRS450_Gyro class ([Java](#), [C++](#)) provides support for the Analog Devices ADXRS450 gyro available in the kit of parts, which connects over the SPI bus.

Note: ADXRS450 Gyro accumulation is handled through special circuitry in the FPGA; accordingly only a single instance of ADXRS450_Gyro may be used.

Java

C++

```
// Creates an ADXRS450_Gyro object on the MXP SPI port
Gyro gyro = new ADXRS450_Gyro(SPI.Port.kMXP);
```

```
// Creates an ADXRS450_Gyro object on the MXP SPI port
ADXRS450_Gyro gyro{SPI::Port::kMXP};
```

AnalogGyro

The AnalogGyro class ([Java](#), [C++](#)) provides support for any single-axis gyro with an analog output.

Note: Gyro accumulation is handled through special circuitry in the FPGA; accordingly, AnalogGyro's may only be used on analog ports 0 and 1.

Java

C++

```
// Creates an AnalogGyro object on port 0
Gyro gyro = new AnalogGyro(0);
```

```
// Creates an AnalogGyro object on port 0
AnalogGyro gyro{0};
```

13.3.2 Third-party gyros

While WPILib provides native support for the ADXRS450 gyro available in the kit of parts and for any analog gyro, there are a few popular AHRS (Attitude and Heading Reference System) devices commonly used in FRC® that include accelerometers and require more complicated communications. These are generally controlled through vendor libraries.

13.3.3 Using gyros in code

Note: As gyros measure rate rather than position, position is inferred by integrating (adding up) the rate signal to get the total change in angle. Thus, gyro angle measurements are always relative to some arbitrary zero angle (determined by the angle of the gyro when either the robot was turned on or a zeroing method was called), and are also subject to accumulated errors (called “drift”) that increase in magnitude the longer the gyro is used. The amount of drift varies with the type of gyro.

Gyros are extremely useful in FRC for both measuring and controlling robot heading. Since FRC matches are generally short, total gyro drift over the course of an FRC match tends to be manageably small (on the order of a couple of degrees for a good-quality gyro). Moreover, not all useful gyro applications require the absolute heading measurement to remain accurate over the course of the entire match.

Displaying the robot heading on the dashboard

Shuffleboard includes a widget for displaying heading data from a Gyro in the form of a compass. This can be helpful for viewing the robot heading when sight lines to the robot are obscured:

Java

C++

```
Gyro gyro = new ADXRS450_Gyro(SPI.Port.kMXP);

public void robotInit() {
    // Places a compass indicator for the gyro heading on the dashboard
    // Explicit down-cast required because Gyro does not extend Sendable
    Shuffleboard.getTab("Example tab").add((Sendable) gyro);
}
```

```
frc::ADXRS450_Gyro gyro{frc::SPI::Port::kMXP};

void Robot::RobotInit() {
    // Places a compass indicator for the gyro heading on the dashboard
    frc::Shuffleboard.GetTab("Example tab").Add(gyro);
}
```

Stabilizing heading while driving

A very common use for a gyro is to stabilize robot heading while driving, so that the robot drives straight. This is especially important for holonomic drives such as mecanum and swerve, but is extremely useful for tank drives as well.

This is typically achieved by closing a PID controller on either the turn rate or the heading, and piping the output of the loop to one's turning control (for a tank drive, this would be a speed differential between the two sides of the drive).

Warning: Like with all control loops, users should be careful to ensure that the sensor direction and the turning direction are consistent. If they are not, the loop will be unstable and the robot will turn wildly.

Example: Tank drive stabilization using turn rate

The following example shows how to stabilize heading using a simple P loop closed on the turn rate. Since a robot that is not turning should have a turn rate of zero, the setpoint for the loop is implicitly zero, making this method very simple.

Java

C++

```
Gyro gyro = new ADXRS450_Gyro(SPI.Port.kMXP);

// The gain for a simple P loop
double kP = 1;

// Initialize motor controllers and drive
Spark left1 = new Spark(0);
Spark left2 = new Spark(1);

Spark right1 = new Spark(2);
Spark right2 = new Spark(3);

SpeedControllerGroup leftMotors = new SpeedControllerGroup(left1, left2);
SpeedControllerGroup rightMotors = new SpeedControllerGroup(right1, right2);

DifferentialDrive drive = new DifferentialDrive(leftMotors, rightMotors);

@Override
public void autonomousPeriodic() {
    // Setpoint is implicitly 0, since we don't want the heading to change
    double error = -gyro.getRate();
```

(continues on next page)

(continued from previous page)

```

    // Drives forward continuously at half speed, using the gyro to stabilize the
    ↪ heading
    drive.tankDrive(.5 + kP * error, .5 - kP * error);
}

```

```

frc::ADXRS450_Gyro gyro{frc::SPI::Port::kMXP};

// The gain for a simple P loop
double kP = 1;

// Initialize motor controllers and drive
frc::Spark left1{0};
frc::Spark left2{1};
frc::Spark right1{2};
frc::Spark right2{3};

frc::SpeedControllerGroup leftMotors{left1, left2};
frc::SpeedControllerGroup rightMotors{right1, right2};

frc::DifferentialDrive drive{leftMotors, rightMotors};

void Robot::AutonomousPeriodic() {
    // Setpoint is implicitly 0, since we don't want the heading to change
    double error = -gyro.GetRate();

    // Drives forward continuously at half speed, using the gyro to stabilize the
    ↪ heading
    drive.TankDrive(.5 + kP * error, .5 - kP * error);
}

```

More-advanced implementations can use a more-complicated control loop. When closing the loop on the turn rate for heading stabilization, PI loops are particularly effective.

Example: Tank drive stabilization using heading

The following example shows how to stabilize heading using a simple P loop closed on the heading. Unlike in the turn rate example, we will need to set the setpoint to the current heading before starting motion, making this method slightly more-complicated.

Java

C++

```

Gyro gyro = new ADXRS450_Gyro(SPI.Port.kMXP);

// The gain for a simple P loop
double kP = 1;

// The heading of the robot when starting the motion
double heading;

// Initialize motor controllers and drive
Spark left1 = new Spark(0);
Spark left2 = new Spark(1);

```

(continues on next page)

(continued from previous page)

```

Spark right1 = new Spark(2);
Spark right2 = new Spark(3);

SpeedControllerGroup leftMotors = new SpeedControllerGroup(left1, left2);
SpeedControllerGroup rightMotors = new SpeedControllerGroup(right1, right2);

DifferentialDrive drive = new DifferentialDrive(leftMotors, rightMotors);

@Override
public void autonomousInit() {
    // Set setpoint to current heading at start of auto
    heading = gyro.getAngle();
}

@Override
public void autonomousPeriodic() {
    double error = heading - gyro.getAngle();

    // Drives forward continuously at half speed, using the gyro to stabilize the
    ↪ heading
    drive.tankDrive(.5 + kP * error, .5 - kP * error);
}

```

```

frc::ADXRS450_Gyro gyro{frc::SPI::Port::kMXP};

// The gain for a simple P loop
double kP = 1;

// The heading of the robot when starting the motion
double heading;

// Initialize motor controllers and drive
frc::Spark left1{0};
frc::Spark left2{1};
frc::Spark right1{2};
frc::Spark right2{3};

frc::SpeedControllerGroup leftMotors{left1, left2};
frc::SpeedControllerGroup rightMotors{right1, right2};

frc::DifferentialDrive drive{leftMotors, rightMotors};

void Robot::AutonomousInit() {
    // Set setpoint to current heading at start of auto
    heading = gyro.GetAngle();
}

void Robot::AutonomousPeriodic() {
    double error = heading - gyro.GetAngle();

    // Drives forward continuously at half speed, using the gyro to stabilize the
    ↪ heading
    drive.TankDrive(.5 + kP * error, .5 - kP * error);
}

```

More-advanced implementations can use a more-complicated control loop. When closing the

loop on the heading for heading stabilization, PD loops are particularly effective.

Turning to a set heading

Another common and highly-useful application for a gyro is turning a robot to face a specified direction. This can be a component of an autonomous driving routine, or can be used during teleoperated control to help align a robot with field elements.

Much like with heading stabilization, this is often accomplished with a PID loop - unlike with stabilization, however, the loop can only be closed on the heading. The following example code will turn the robot to face 90 degrees with a simple P loop:

Java

C++

```
Gyro gyro = new ADXRS450_Gyro(SPI.Port.kMXP);

// The gain for a simple P loop
double kP = 1;

// Initialize motor controllers and drive
Spark left1 = new Spark(0);
Spark left2 = new Spark(1);

Spark right1 = new Spark(2);
Spark right2 = new Spark(3);

SpeedControllerGroup leftMotors = new SpeedControllerGroup(left1, left2);
SpeedControllerGroup rightMotors = new SpeedControllerGroup(right1, right2);

DifferentialDrive drive = new DifferentialDrive(leftMotors, rightMotors);

@Override
public void autonomousPeriodic() {
    // Find the heading error; setpoint is 90
    double error = 90 - gyro.getAngle();

    // Turns the robot to face the desired direction
    drive.tankDrive(kP * error, kP * error);
}
```

```
frc::ADXRS450_Gyro gyro{frc::SPI::Port::kMXP};

// The gain for a simple P loop
double kP = 1;

// Initialize motor controllers and drive
frc::Spark left1{0};
frc::Spark left2{1};
frc::Spark right1{2};
frc::Spark right2{3};

frc::SpeedControllerGroup leftMotors{left1, left2};
frc::SpeedControllerGroup rightMotors{right1, right2};

frc::DifferentialDrive drive{leftMotors, rightMotors};
```

(continues on next page)

(continued from previous page)

```

void Robot::AutonomousPeriodic() {
    // Find the heading error; setpoint is 90
    double error = 90 - gyro.GetAngle();

    // Turns the robot to face the desired direction
    drive.TankDrive(kP * error, kP * error);
}

```

As before, more-advanced implementations can use more-complicated control loops.

Note: Turn-to-angle loops can be tricky to tune correctly due to static friction in the drive-train, especially if a simple P loop is used. There are a number of ways to account for this; one of the most common/effective is to add a “minimum output” to the output of the control loop. Another effective strategy is to cascade to well-tuned velocity controllers on each side of the drive.

13.4 Ultrasonics - Software

Note: This section covers ultrasonics in software. For a hardware guide to ultrasonics, see [Ultrasonics - Hardware](#).

An ultrasonic sensor is commonly used to measure distance to an object using high-frequency sound. Generally, ultrasonics measure the distance to the closest object within their “field of view.”

There are two primary types of ultrasonics supported natively by WPILib:

- *Ping-response ultrasonics*
- *Analog ultrasonics*

13.4.1 Ping-response ultrasonics

The Ultrasonic class (Java, C++) provides support for ping-response ultrasonics. As ping-response ultrasonics (per the: name) require separate pins for both sending the ping and measuring the response, users must specify DIO pin numbers for both output and input when constructing an Ultrasonic instance:

Java

C++

```

// Creates a ping-response Ultrasonic object on DIO 1 and 2.
Ultrasonic ultrasonic = new Ultrasonic(1, 2);

```

```

// Creates a ping-response Ultrasonic object on DIO 1 and 2.
frc::Ultrasonic ultrasonic{1, 2};

```

It is highly recommended to use ping-response ultrasonics in “automatic mode,” as this will allow WPILib to ensure that multiple sensors do not interfere with each other:

Java

C++

```
// Starts the ultrasonic sensor running in automatic mode
ultrasonic.setAutomaticMode(true);
```

```
// Starts the ultrasonic sensor running in automatic mode
ultrasonic.SetAutomaticMode(true);
```

13.4.2 Analog ultrasonics

Some ultrasonic sensors simply return an analog voltage corresponding to the measured distance. These sensors can may simply be used with the [AnalogPotentiometer](#) class.

13.4.3 Third-party ultrasonics

Other ultrasonic sensors offered by third-parties may use more complicated communications protocols (such as I2C or SPI). WPILib does not provide native support for any such ultrasonics; they will typically be controlled with vendor libraries.

13.4.4 Using ultrasonics in code

Ultrasonic sensors are very useful for determining spacing during autonomous routines. For example, the following code will drive the robot forward until the ultrasonic measures a distance of 12 inches (~30 cm) to the nearest object, and then stop:

Java

C++

```
// Creates a ping-response Ultrasonic object on DIO 1 and 2.
Ultrasonic ultrasonic = new Ultrasonic(1, 2);

// Initialize motor controllers and drive
Spark left1 new Spark(0);
Spark left2 = new Spark(1);

Spark right1 = new Spark(2);
Spark right2 = new Spark(3);

SpeedControllerGroup leftMotors = new SpeedControllerGroup(left1, left2);
SpeedControllerGroup rightMotors = new SpeedControllerGroup(right1, right2);

DifferentialDrive drive = new DifferentialDrive(leftMotors, rightMotors);

@Override
public void robotInit() {
    // Start the ultrasonic in automatic mode
    ultrasonic.setAutomaticMode(true);
}
```

(continues on next page)

(continued from previous page)

```

}

@Override
public void autonomousPeriodic() {
    if(ultrasonic.GetRangeInches() > 12) {
        drive.tankDrive(.5, .5);
    }
    else {
        drive.tankDrive(0, 0);
    }
}
}

```

```

// Creates a ping-response Ultrasonic object on DIO 1 and 2.
frc::Ultrasonic ultrasonic{1, 2};

// Initialize motor controllers and drive
frc::Spark left1{0};
frc::Spark left2{1};
frc::Spark right1{2};
frc::Spark right2{3};

frc::SpeedControllerGroup leftMotors{left1, left2};
frc::SpeedControllerGroup rightMotors{right1, right2};

frc::DifferentialDrive drive{leftMotors, rightMotors};

void Robot::RobotInit() {
    // Start the ultrasonic in automatic mode
    ultrasonic.SetAutomaticMode(true);
}

void Robot::AutonomousPeriodic() {
    if(ultrasonic.GetRangeInches() > 12) {
        drive.TankDrive(.5, .5);
    }
    else {
        drive.TankDrive(0, 0);
    }
}
}

```

Additionally, ping-response ultrasonics can be sent to [Shuffleboard](#), where they will be displayed with their own widgets:

Java

C++

```

// Creates a ping-response Ultrasonic object on DIO 1 and 2.
Ultrasonic ultrasonic = new Ultrasonic(1, 2);

public void robotInit() {
    // Places a the ultrasonic on the dashboard
    Shuffleboard.getTab("Example tab").add(ultrasonic);
}

```

```

// Creates a ping-response Ultrasonic object on DIO 1 and 2.

```

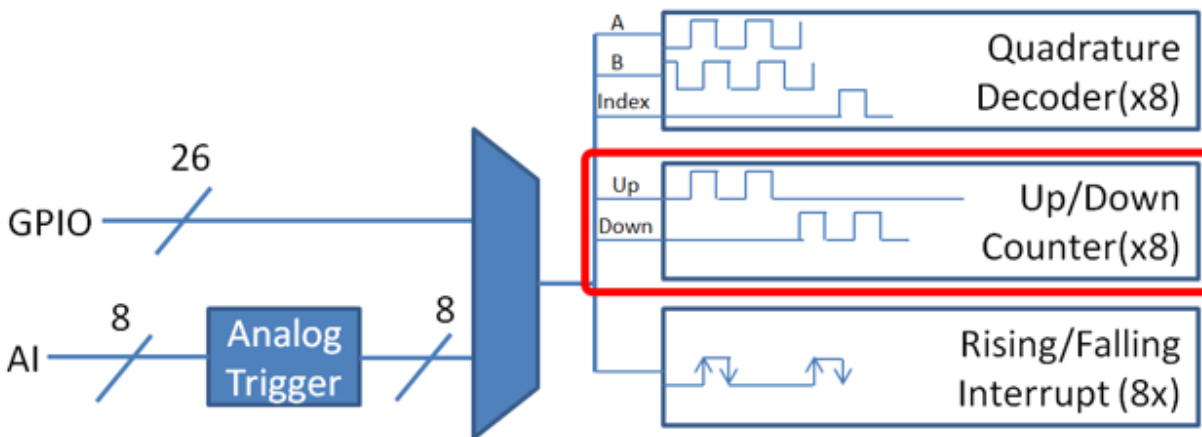
(continues on next page)

(continued from previous page)

```
frc::Ultrasonic ultrasonic{1, 2};

void Robot::RobotInit() {
    // Places the ultrasonic on the dashboard
    frc::Shuffleboard.GetTab("Example tab").Add(ultrasonic);
}
```

13.5 Counters



The Counter class ([Java](#), [C++](#)) is a versatile class that allows the counting of pulse edges on a digital input. Counter is used as a component in several more-complicated WPILib classes (such as [Encoder](#) and [Ultrasonic](#)), but is also quite useful on its own.

Note: There are a total of 8 counter units in the roboRIO FPGA, meaning no more than 8 Counter objects may be instantiated at any one time, including those contained as resources in other WPILib objects. For detailed information on when a Counter may be used by another object, refer to the official API documentation.

13.5.1 Configuring a counter

The Counter class can be configured in a number of ways to provide differing functionalities.

Counter Modes

The Counter object may be configured to operate in one of four different modes:

1. *Two-pulse mode*: Counts up and down based on the edges of two different channels.
2. *Semi-period mode*: Measures the duration of a pulse on a single channel.
3. *Pulse-length mode*: Counts up and down based on the edges of one channel, with the direction determined by the duration of the pulse on that channel.

4. *External direction mode*: Counts up and down based on the edges of one channel, with a separate channel specifying the direction.

Note: In all modes except semi-period mode, the counter can be configured to increment either once per edge (2X decoding), or once per pulse (1X decoding). By default, counters are set to two-pulse mode, if only one channel is specified, the counter will only count up.

Two-pulse mode

In two-pulse mode, the Counter will count up for every edge/pulse on the specified “up channel,” and down for every edge/pulse on the specified “down channel.” A counter can be initialized in two-pulse with the following code:

Java

C++

```
// Create a new Counter object in two-pulse mode
Counter counter = new Counter(Counter.Mode.k2Pulse);

@Override
public void robotInit() {
    // Set up the input channels for the counter
    counter.setUpSource(1);
    counter.setDownSource(2);

    // Set the decoding type to 2X
    counter.setUpSourceEdge(true, true);
    counter.setDownSourceEdge(true, true);
}
```

```
// Create a new Counter object in two-pulse mode
frc::Counter counter{frc::Counter::Mode::k2Pulse};

void Robot::RobotInit() {
    // Set up the input channels for the counter
    counter.SetUpSource(1);
    counter.SetDownSource(2);

    // Set the decoding type to 2X
    counter.SetUpSourceEdge(true, true);
    counter.SetDownSourceEdge(true, true);
}
```

Semi-period mode

In semi-period mode, the Counter will count the duration of the pulses on a channel, either from a rising edge to the next falling edge, or from a falling edge to the next rising edge. A counter can be initialized in semi-period mode with the following code:

Java

C++

```
// Create a new Counter object in two-pulse mode
Counter counter = new Counter(Counter.Mode.kSemiperiod);

@Override
public void robotInit() {
    // Set up the input channel for the counter
    counter.setUpSource(1);

    // Set the encoder to count pulse duration from rising edge to falling edge
    counter.setSemiPeriodMode(true);
}
```

```
// Create a new Counter object in two-pulse mode
frc::Counter counter{frc::Counter::Mode::kSemiperiod};

void Robot() {
    // Set up the input channel for the counter
    counter.SetUpSource(1);

    // Set the encoder to count pulse duration from rising edge to falling edge
    counter.SetSemiPeriodMode(true);
}
```

To get the pulse width, call the `getPeriod()` method:

Java

C++

```
// Return the measured pulse width in seconds
counter.GetPeriod();
```

```
// Return the measured pulse width in seconds
counter.getPeriod();
```

Pulse-length mode

In pulse-length mode, the counter will count either up or down depending on the length of the pulse. A pulse below the specified threshold time will be interpreted as a forward count and a pulse above the threshold is a reverse count. This is useful for some gear tooth sensors which encode direction in this manner. A counter can be initialized in this mode as follows:

Java

C++

```
// Create a new Counter object in two-pulse mode
Counter counter = new Counter(Counter.Mode.kPulseLength);

@Override
public void robotInit() {
    // Set up the input channel for the counter
    counter.setUpSource(1);

    // Set the decoding type to 2X
    counter.setUpSourceEdge(true, true);
}
```

(continues on next page)

(continued from previous page)

```

// Set the counter to count down if the pulses are longer than .05 seconds
counter.SetPulseLengthMode(.05)
}

```

```

// Create a new Counter object in two-pulse mode
frc::Counter counter{frc::Counter::Mode::kPulseLength};

void Robot::RobotInit() {
    // Set up the input channel for the counter
    counter.SetupSource(1);

    // Set the decoding type to 2X
    counter.SetupSourceEdge(true, true);

    // Set the counter to count down if the pulses are longer than .05 seconds
    counter.setPulseLengthMode(.05)
}

```

External direction mode

In external direction mode, the counter counts either up or down depending on the level on the second channel. If the direction source is low, the counter will increase, if the direction source is high, the counter will decrease (to reverse this, see the next section). A counter can be initialized in this mode as follows:

Java

C++

```

// Create a new Counter object in two-pulse mode
Counter counter = new Counter(Counter.Mode.kExternalDirection);

@Override
public void robotInit() {
    // Set up the input channels for the counter
    counter.setUpSource(1);
    counter.setDownSource(2);

    // Set the decoding type to 2X
    counter.setUpSourceEdge(true, true);
}

```

```

// Create a new Counter object in two-pulse mode
frc::Counter counter{frc::Counter::Mode::kExternalDirection};

void RobotInit() {
    // Set up the input channels for the counter
    counter.SetupSource(1);
    counter.SetDownSource(2);

    // Set the decoding type to 2X
    counter.SetupSourceEdge(true, true);
}

```

Configuring counter parameters

Note: The Counter class does not make any assumptions about units of distance; it will return values in whatever units were used to calculate the distance-per-pulse value. Users thus have complete control over the distance units used. However, units of time are *always* in seconds.

Note: The number of pulses used in the distance-per-pulse calculation does *not* depend on the decoding type - each “pulse” should always be considered to be a full cycle (rising and falling).

Apart from the mode-specific configurations, the Counter class offers a number of additional configuration methods:

Java

C++

```
// Configures the counter to return a distance of 4 for every 256 pulses
// Also changes the units of getRate
counter.setDistancePerPulse(4./256.);

// Configures the counter to consider itself stopped after .1 seconds
counter.setMaxPeriod(.1);

// Configures the counter to consider itself stopped when its rate is below 10
counter.setMinRate(10);

// Reverses the direction of the counter
counter.setReverseDirection(true);

// Configures an counter to average its period measurement over 5 samples
// Can be between 1 and 127 samples
counter.setSamplesToAverage(5);
```

```
// Configures the counter to return a distance of 4 for every 256 pulses
// Also changes the units of getRate
counter.SetDistancePerPulse(4./256.);

// Configures the counter to consider itself stopped after .1 seconds
counter.SetMaxPeriod(.1);

// Configures the counter to consider itself stopped when its rate is below 10
counter.SetMinRate(10);

// Reverses the direction of the counter
counter.SetReverseDirection(true);

// Configures an counter to average its period measurement over 5 samples
// Can be between 1 and 127 samples
counter.SetSamplesToAverage(5);
```


13.5.2 Reading information from counters

Regardless of mode, there is some information that the Counter class always exposes to users:

Count

Users can obtain the current count with the `get()` method:

Java

C++

```
// returns the current count
counter.get();
```

```
// returns the current count
counter.Get();
```

Distance

Note: Counters measure *relative* distance, not absolute; the distance value returned will depend on the position of the encoder when the robot was turned on or the encoder value was last *reset*.

If the *distance per pulse* has been configured, users can obtain the total distance traveled by the counted sensor with the `getDistance()` method:

Java

C++

```
// returns the current distance
counter.getDistance();
```

```
// returns the current distance
counter.GetDistance();
```

Rate

Note: Units of time for the Counter class are *always* in seconds.

Users can obtain the current rate of change of the counter with the `getRate()` method:

Java

C++

```
// Gets the current rate of the counter
counter.getRate();
```

```
// Gets the current rate of the counter
counter.GetRate();
```

Stopped

Users can obtain whether the counter is stationary with the `getStopped()` method:

Java

C++

```
// Gets whether the counter is stopped
counter.getStopped();
```

```
// Gets whether the counter is stopped
counter.GetStopped();
```

Direction

Users can obtain the direction in which the counter last moved with the `getDirection()` method:

Java

C++

```
// Gets the last direction in which the counter moved
counter.getDirection();
```

```
// Gets the last direction in which the counter moved
counter.GetDirection();
```

Period

Note: In *semi-period mode*, this method returns the duration of the pulse, not of the period.

Users can obtain the duration (in seconds) of the most-recent period with the `getPeriod()` method:

Java

C++

```
// returns the current period in seconds
counter.getPeriod();
```

```
// returns the current period in seconds
counter.GetPeriod();
```

13.5.3 Resetting a counter

To reset a counter to a distance reading of zero, call the `reset()` method. This is useful for ensuring that the measured distance corresponds to the actual desired physical measurement.

Java

C++

```
// Resets the encoder to read a distance of zero
counter.reset();
```

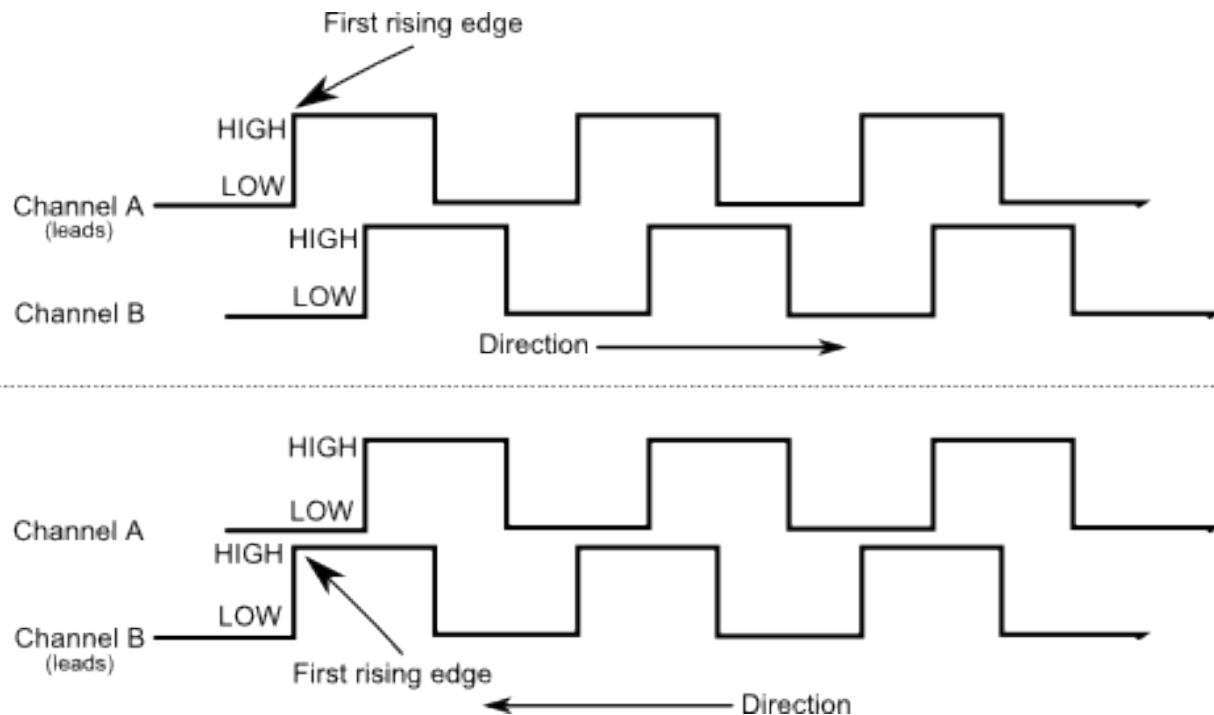
```
// Resets the encoder to read a distance of zero
counter.Reset();
```

13.5.4 Using counters in code

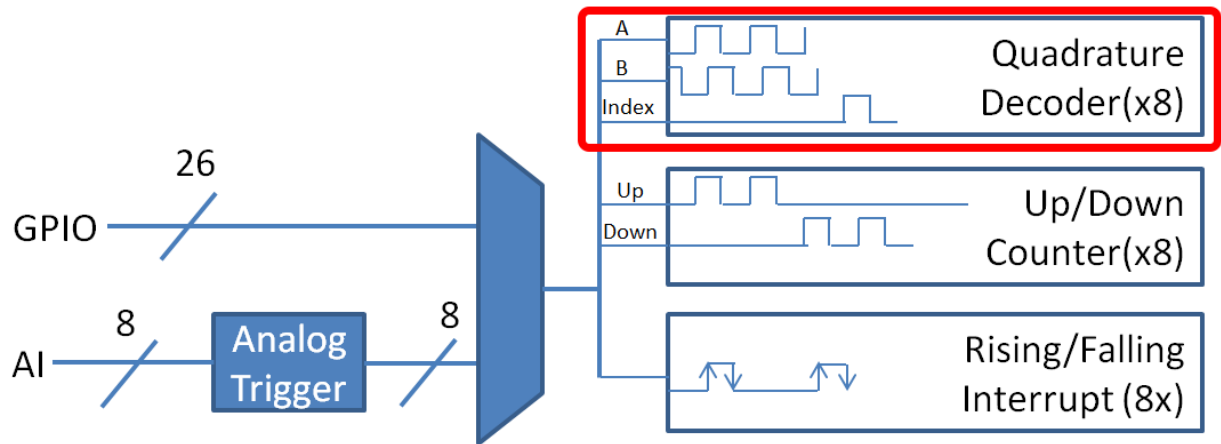
Counters are useful for a wide variety of robot applications - but since the `Counter` class is so varied, it is difficult to provide a good summary of them here. Many of these applications overlap with the `Encoder` class - a simple counter is often a cheaper alternative to a quadrature encoder. For a summary of potential uses for encoders in code, see [Encoders - Software](#).

13.6 Encoders - Software

Note: This section covers encoders in software. For a hardware guide to encoders, see [Encoders - Hardware](#).



Encoders are devices used to measure motion (usually, the rotation of a shaft). The encoders used in FRC® are known as “quadrature encoders.” These encoders produce square-wave signals on two channels that are a quarter-period out-of-phase (hence the term, “quadrature”). The pulses are used to measure the rotation, and the direction of motion can be determined from which channel “leads” the other.



The FPGA handles encoders either through a counter module or an encoder module, depending on the *decoding type* - the choice is handled automatically by WPILib. The FPGA contains 8 encoder modules.

13.6.1 The Encoder class

WPILib provides support for encoders through the Encoder class (Java, C++). This class provides a simple API for configuring and reading data from encoders.

Important: The Encoder class is only used for encoders that are plugged directly into the roboRIO! Please reference the appropriate vendors’ documentation for using encoders plugged into motor controllers.

Initializing an encoder

An encoder can be instantiated as follows:

Java

C++

```
// Initializes an encoder on DIO pins 0 and 1
// Defaults to 4X decoding and non-inverted
Encoder encoder = new Encoder(0, 1);
```

```
// Initializes an encoder on DIO pins 0 and 1
// Defaults to 4X decoding and non-inverted
frc::Encoder encoder{0, 1};
```

Decoding type

The WPILib Encoder class can decode encoder signals in three different modes:

- **1X Decoding:** Increments the distance for every complete period of the encoder signal (once per four edges).
- **2X Decoding:** Increments the distance for every half-period of the encoder signal (once per two edges).
- **4X Decoding:** Increments the distance for every edge of the encoder signal (four times per period).

4X decoding offers the greatest precision, but at the potential cost of increased “jitter” in rate measurements. To use a different decoding type, use the following constructor:

Java

C++

```
// Initializes an encoder on DIO pins 0 and 1
// 2X encoding and non-inverted
Encoder encoder = new Encoder(0, 1, false, Encoder.EncodingType.k2X);
```

```
// Initializes an encoder on DIO pins 0 and 1
// 2X encoding and non-inverted
frc::Encoder encoder{0, 1, false, frc::Encoder::EncodingType::k2X};
```

Configuring encoder parameters

Note: The Encoder class does not make any assumptions about units of distance; it will return values in whatever units were used to calculate the distance-per-pulse value. Users thus have complete control over the distance units used. However, units of time are *always* in seconds.

Note: The number of pulses used in the distance-per-pulse calculation does *not* depend on the *decoding type* - each “pulse” should always be considered to be a full cycle (four edges).

The Encoder class offers a number of configuration methods:

Java

C++

```
// Configures the encoder to return a distance of 4 for every 256 pulses
// Also changes the units of getRate
encoder.setDistancePerPulse(4./256.);

// Configures the encoder to consider itself stopped after .1 seconds
encoder.setMaxPeriod(.1);

// Configures the encoder to consider itself stopped when its rate is below 10
encoder.setMinRate(10);
```

(continues on next page)

(continued from previous page)

```
// Reverses the direction of the encoder
encoder.setReverseDirection(true);

// Configures an encoder to average its period measurement over 5 samples
// Can be between 1 and 127 samples
encoder.setSamplesToAverage(5);

// Configures the encoder to return a distance of 4 for every 256 pulses
// Also changes the units of getRate
encoder.SetDistancePerPulse(4./256.);

// Configures the encoder to consider itself stopped after .1 seconds
encoder.SetMaxPeriod(.1);

// Configures the encoder to consider itself stopped when its rate is below 10
encoder.SetMinRate(10);

// Reverses the direction of the encoder
encoder.SetReverseDirection(true);

// Configures an encoder to average its period measurement over 5 samples
// Can be between 1 and 127 samples
encoder.SetSamplesToAverage(5);
```

Reading information from encoders

The Encoder class provides a wealth of information to the user about the motion of the encoder.

Distance

Note: Quadrature encoders measure *relative* distance, not absolute; the distance value returned will depend on the position of the encoder when the robot was turned on or the encoder value was last *reset*.

Users can obtain the total distance traveled by the encoder with the `getDistance()` method:

Java

C++

```
// Gets the distance traveled
encoder.getDistance();
```

```
// Gets the distance traveled
encoder.GetDistance();
```

Rate

Note: Units of time for the Encoder class are *always* in seconds.

Users can obtain the current rate of change of the encoder with the `getRate()` method:

Java

C++

```
// Gets the current rate of the encoder
encoder.getRate();
```

```
// Gets the current rate of the encoder
encoder.GetRate();
```

Stopped

Users can obtain whether the encoder is stationary with the `getStopped()` method:

Java

C++

```
// Gets whether the encoder is stopped
encoder.getStopped();
```

```
// Gets whether the encoder is stopped
encoder.GetStopped();
```

Direction

Users can obtain the direction in which the encoder last moved with the `getDirection()` method:

Java

C++

```
// Gets the last direction in which the encoder moved
encoder.getDirection();
```

```
// Gets the last direction in which the encoder moved
encoder.GetDirection();
```

Period

Users can obtain the period of the encoder pulses (in seconds) with the `getPeriod()` method:

Java

C++

```
// Gets the current period of the encoder
encoder.getPeriod();
```

```
// Gets the current period of the encoder
encoder.GetPeriod();
```

Resetting an encoder

To reset an encoder to a distance reading of zero, call the `reset()` method. This is useful for ensuring that the measured distance corresponds to the actual desired physical measurement, and is often called during a *homing* routine:

Java

C++

```
// Resets the encoder to read a distance of zero
encoder.reset();
```

```
// Resets the encoder to read a distance of zero
encoder.Reset();
```

13.6.2 Using encoders in code

Encoders are some of the most useful sensors in FRC; they are very nearly a requirement to make a robot capable of nontrivially-automated actuations and movement. The potential applications of encoders in robot code are too numerous to summarize fully here, but a few basic examples are provided below:

Driving to a distance

Encoders can be used on a robot drive to create a simple “drive to distance” routine. This is very useful for robot autonomy:

Java

C++

```
// Creates an encoder on DIO ports 0 and 1
Encoder encoder = new Encoder(0, 1);

// Initialize motor controllers and drive
Spark left1 = new Spark(0);
Spark left2 = new Spark(1);
```

(continues on next page)

(continued from previous page)

```

Spark right1 = new Spark(2);
Spark right2 = new Spark(3);

SpeedControllerGroup leftMotors = new SpeedControllerGroup(left1, left2);
SpeedControllerGroup rightMotors = new SpeedControllerGroup(right1, right2);

DifferentialDrive drive = new DifferentialDrive(leftMotors, rightMotors);

@Override
public void robotInit() {
    // Configures the encoder's distance-per-pulse
    // The robot moves forward 1 foot per encoder rotation
    // There are 256 pulses per encoder rotation
    encoder.setDistancePerPulse(1./256.);
}

@Override
public void autonomousPeriodic() {
    // Drives forward at half speed until the robot has moved 5 feet, then stops:
    if(encoder.getDistance() < 5) {
        drive.tankDrive(.5, .5);
    } else {
        drive.tankDrive(0, 0);
    }
}
}

```

```

// Creates an encoder on DIO ports 0 and 1.
frc::Encoder encoder{0, 1};

// Initialize motor controllers and drive
frc::Spark left1{0};
frc::Spark left2{1};
frc::Spark right1{2};
frc::Spark right2{3};

frc::SpeedControllerGroup leftMotors{left1, left2};
frc::SpeedControllerGroup rightMotors{right1, right2};

frc::DifferentialDrive drive{leftMotors, rightMotors};

void Robot::RobotInit() {
    // Configures the encoder's distance-per-pulse
    // The robot moves forward 1 foot per encoder rotation
    // There are 256 pulses per encoder rotation
    encoder.SetDistancePerPulse(1./256.);
}

void Robot::AutonomousPeriodic() {
    // Drives forward at half speed until the robot has moved 5 feet, then stops:
    if(encoder.GetDistance() < 5) {
        drive.TankDrive(.5, .5);
    } else {
        drive.TankDrive(0, 0);
    }
}
}

```

Stabilizing heading

Warning: Like with all control loops, users should be careful to ensure that the sensor direction and the turning direction are consistent. If they are not, the loop will be unstable and the robot will turn wildly.

Encoders can be used to ensure that a robot drives straight in a manner quite similar to *how it is done with a gyroscope*. A simple implementation with a P loop is given below:

Java

C++

```
// The encoders for the drive
Encoder leftEncoder = new Encoder(0,1);
Encoder rightEncoder = new Encoder(2,3);

// The gain for a simple P loop
double kP = 1;

// Initialize motor controllers and drive
Spark left1 = new Spark(0);
Spark left2 = new Spark(1);

Spark right1 = new Spark(2);
Spark right2 = new Spark(3);

SpeedControllerGroup leftMotors = new SpeedControllerGroup(left1, left2);
SpeedControllerGroup rightMotors = new SpeedControllerGroup(right1, right2);

DifferentialDrive drive = new DifferentialDrive(leftMotors, rightMotors);

@Override
public void autonomousInit() {
    // Configures the encoders' distance-per-pulse
    // The robot moves forward 1 foot per encoder rotation
    // There are 256 pulses per encoder rotation
    leftEncoder.setDistancePerPulse(1./256.);
    rightEncoder.setDistancePerPulse(1./256.);
}

@Override
public void autonomousPeriodic() {
    // Assuming no wheel slip, the difference in encoder distances is proportional to
    // the heading error
    double error = leftEncoder.getDistance() - rightEncoder.getDistance();

    // Drives forward continuously at half speed, using the encoders to stabilize the
    // heading
    drive.tankDrive(.5 + kP * error, .5 - kP * error);
}
```

```
// The encoders for the drive
frc::Encoder leftEncoder{0,1};
frc::Encoder rightEncoder{2,3};
```

(continues on next page)

(continued from previous page)

```

// The gain for a simple P loop
double kP = 1;

// Initialize motor controllers and drive
frc::Spark left1{0};
frc::Spark left2{1};
frc::Spark right1{2};
frc::Spark right2{3};

frc::SpeedControllerGroup leftMotors{left1, left2};
frc::SpeedControllerGroup rightMotors{right1, right2};

frc::DifferentialDrive drive{leftMotors, rightMotors};

void Robot::AutonomousInit() {
    // Configures the encoders' distance-per-pulse
    // The robot moves forward 1 foot per encoder rotation
    // There are 256 pulses per encoder rotation
    leftEncoder.SetDistancePerPulse(1./256.);
    rightEncoder.SetDistancePerPulse(1./256.);
}

void Robot::AutonomousPeriodic() {
    // Assuming no wheel slip, the difference in encoder distances is proportional to
    // the heading error
    double error = leftEncoder.GetDistance() - rightEncoder.GetDistance();

    // Drives forward continuously at half speed, using the encoders to stabilize the
    // heading
    drive.TankDrive(.5 + kP * error, .5 - kP * error);
}

```

More-advanced implementations can use more-complicated control loops. Closing a control loop on the encoder difference is roughly analogous to closing it on the heading error, and so PD loops are particularly effective.

PID Control

Encoders are particularly useful as inputs to PID controllers (the heading stabilization example above is a simple P loop).

Homing a mechanism

Since encoders measure *relative* distance, it is often important to ensure that their “zero-point” is in the right place. A typical way to do this is a “homing routine,” in which a mechanism is moved until it hits a known position (usually accomplished with a limit switch), or “home,” and then the encoder is reset. The following code provides a basic example:

Java

C++

```
Encoder encoder = new Encoder(0, 1);

Spark spark = new Spark(0);

// Limit switch on DIO 2
DigitalInput limit = new DigitalInput(2);

public void autonomousPeriodic() {
    // Runs the motor backwards at half speed until the limit switch is pressed
    // then turn off the motor and reset the encoder
    if(!limit.get()) {
        spark.set(-.5);
    } else {
        spark.set(0);
        encoder.reset();
    }
}
```

```
frc::Encoder encoder{0,1};

frc::Spark spark{0};

// Limit switch on DIO 2
frc::DigitalInput limit{2};

void AutonomousPeriodic() {
    // Runs the motor backwards at half speed until the limit switch is pressed
    // then turn off the motor and reset the encoder
    if(!limit.Get()) {
        spark.Set(-.5);
    } else {
        spark.Set(0);
        encoder.Reset();
    }
}
```

13.7 Analog Inputs - Software

Note: This section covers analog inputs in software. For a hardware guide to analog inputs, see [Analog Inputs - Hardware](#).

The roboRIO's FPGA supports up to 8 analog input channels that can be used to read the value of an analog voltage from a sensor. Analog inputs may be used for any sensor that outputs a simple voltage.

Analog inputs from the FPGA by default return a 12-bit integer proportional to the voltage, from 0 to 5 volts.

13.7.1 The AnalogInput class

Note: It is often more convenient to use the *Analog Potentiometers* wrapper class than to use AnalogInput directly, as it supports scaling to meaningful units.

Support for reading the voltages on the FPGA analog inputs is provided through the AnalogInput class (Java, C++).

Initializing an AnalogInput

An AnalogInput may be initialized as follows:

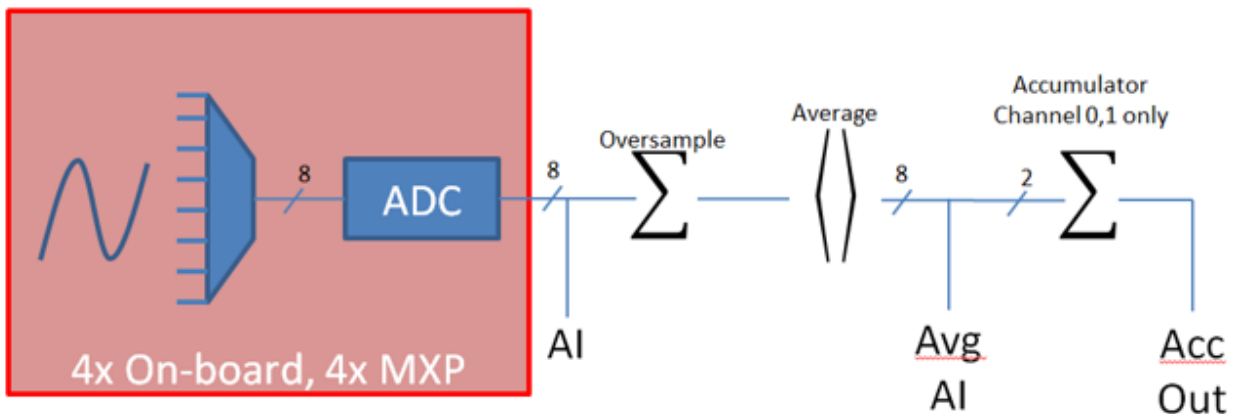
Java

C++

```
// Initializes an AnalogInput on port 0
AnalogInput analog = new AnalogInput(0);
```

```
// Initializes an AnalogInput on port 0
frc::AnalogInput analog{0};
```

Oversampling and Averaging



The FPGA's analog input modules supports both oversampling and averaging. These behaviors are highly similar, but differ in a few important ways. Both may be used at the same time.

Oversampling

When oversampling is enabled, the FPGA will add multiple consecutive samples together, and return the accumulated value. Users may specify the number of *bits* of oversampling - for n bits of oversampling, the number of samples added together is 2^n :

Java

C++

```
// Sets the AnalogInput to 4-bit oversampling. 16 samples will be added together.  
// Thus, the reported values will increase by about a factor of 16, and the update  
// rate will decrease by a similar amount.  
analog.setOversampleBits(4);
```

```
// Sets the AnalogInput to 4-bit oversampling. 16 samples will be added together.  
// Thus, the reported values will increase by about a factor of 16, and the update  
// rate will decrease by a similar amount.  
analog.SetOversampleBits(4);
```

Averaging

Averaging behaves much like oversampling, except the accumulated values are divided by the number of samples so that the scaling of the returned values does not change. This is often more-convenient, but occasionally the additional roundoff error introduced by the rounding is undesirable.

Java

C++

```
// Sets the AnalogInput to 4-bit averaging. 16 samples will be averaged together.  
// The update rate will decrease by a factor of 16.  
analog.setAverageBits(4);
```

```
// Sets the AnalogInput to 4-bit averaging. 16 samples will be averaged together.  
// The update rate will decrease by a factor of 16.  
analog.SetAverageBits(4);
```

Note: When oversampling and averaging are used at the same time, the oversampling is applied *first*, and then the oversampled values are averaged. Thus, 2-bit oversampling and 2-bit averaging used at the same time will increase the scale of the returned values by approximately a factor of 2, and decrease the update rate by approximately a factor of 4.

Reading values from an AnalogInput

Values can be read from an AnalogInput with one of four different methods:

getValue

The `getValue` method returns the raw instantaneous measured value from the analog input, without applying any calibration and ignoring oversampling and averaging settings. The returned value is an integer.

Java

C++

```
analog.getValue();
```

```
analog.GetValue();
```

getVoltage

The `getVoltage` method returns the instantaneous measured voltage from the analog input. Oversampling and averaging settings are ignored, but the value is rescaled to represent a voltage. The returned value is a double.

Java

C++

```
analog.getVoltage();
```

```
analog.GetVoltage();
```

getAverageValue

The `getAverageValue` method returns the averaged value from the analog input. The value is not rescaled, but oversampling and averaging are both applied. The returned value is an integer.

Java

C++

```
analog.getAverageValue();
```

```
analog.GetAverageValue();
```

getAverageVoltage

The `getAverageVoltage` method returns the averaged voltage from the analog input. Rescaling, oversampling, and averaging are all applied. The returned value is a double.

Java

C++

```
analog.getAverageVoltage();
```

```
analog.GetAverageVoltage();
```

Accumulator

Note: The accumulator methods do not currently support returning a value in units of volts - the returned value will always be an integer (specifically, a long).

Analog input channels 0 and 1 additionally support an accumulator, which integrates (adds up) the signal indefinitely, so that the returned value is the sum of all past measured values. Oversampling and averaging are applied prior to accumulation.

Java

C++

```
// Sets the initial value of the accumulator to 0
// This is the "starting point" from which the value will change over time
analog.setAccumulatorInitialValue(0);

// Sets the "center" of the accumulator to 0. This value is subtracted from
// all measured values prior to accumulation.
analog.setAccumulatorCenter(0);

// Returns the number of accumulated samples since the accumulator was last started/
↪ reset
analog.getAccumulatorCount();

// Returns the value of the accumulator. Return type is long.
analog.getAccumulatorValue();

// Resets the accumulator to the initial value
analog.resetAccumulator();
```

```
// Sets the initial value of the accumulator to 0
// This is the "starting point" from which the value will change over time
analog.SetAccumulatorInitialValue(0);

// Sets the "center" of the accumulator to 0. This value is subtracted from
// all measured values prior to accumulation.
analog.SetAccumulatorCenter(0);

// Returns the number of accumulated samples since the accumulator was last started/
↪ reset
```

(continues on next page)

(continued from previous page)

```
analog.GetAccumulatorCount();

// Returns the value of the accumulator. Return type is long.
analog.GetAccumulatorValue();

// Resets the accumulator to the initial value
analog.ResetAccumulator();
```

Obtaining synchronized count and value

Sometimes, it is necessary to obtain matched measurements of the count and the value. This can be done using the `getAccumulatorOutput` method:

Java

C++

```
// Instantiate an AccumulatorResult object to hold the matched measurements
AccumulatorResult result = new AccumulatorResult();

// Fill the AccumulatorResult with the matched measurements
analog.getAccumulatorOutput(result);

// Read the values from the AccumulatorResult
long count = result.count;
long value = result.value;
```

```
// The count and value variables to fill
int_64t count;
int_64t value;

// Fill the count and value variables with the matched measurements
analog.GetAccumulatorOutput(count, value);
```

13.7.2 Using analog inputs in code

The `AnalogInput` class can be used to write code for a wide variety of sensors (including potentiometers, accelerometers, gyroscopes, ultrasonics, and more) that return their data as an analog voltage. However, if possible it is almost always more convenient to use one of the other existing WPILib classes that handles the lower-level code (reading the analog voltages and converting them to meaningful units) for you. Users should only directly use `AnalogInput` as a “last resort.”

Accordingly, for examples of how to effectively use analog sensors in code, users should refer to the other pages of this chapter that deal with more-specific classes.

13.8 Analog Potentiometers - Software

Note: This section covers analog potentiometers in software. For a hardware guide to analog potentiometers, see [Analog Potentiometers - Hardware](#).

Potentiometers are variable resistors that allow information about position to be converted into an analog voltage signal. This signal can be read by the roboRIO to control whatever device is attached to the potentiometer.

While it is possible to read information from a potentiometer directly with an [Analog Inputs - Software](#), WPILib provides an `AnalogPotentiometer` class (Java, C++) that handles re-scaling the values into meaningful units for the user. It is strongly encouraged to use this class.

In fact, the `AnalogPotentiometer` name is something of a misnomer - this class should be used for the vast majority of sensors that return their signal as a simple, linearly-scaled analog voltage.

13.8.1 The `AnalogPotentiometer` class

Note: The “full range” or “scale” parameters in the `AnalogPotentiometer` constructor are scale factors from a range of 0-1 to the actual range, *not* from 0-5. That is, they represent a native fractional scale, rather than a voltage scale.

An `AnalogPotentiometer` can be initialized as follows:

Java

C++

```
// Initializes an AnalogPotentiometer on analog port 0
// The full range of motion (in meaningful external units) is 0-180 (this could be
↪degrees, for instance)
// The "starting point" of the motion, i.e. where the mechanism is located when the
↪potentiometer reads 0v, is 30.
```

```
AnalogPotentiometer pot = new AnalogPotentiometer(0, 180, 30);
```

```
// Initializes an AnalogPotentiometer on analog port 0
// The full range of motion (in meaningful external units) is 0-180 (this could be
↪degrees, for instance)
// The "starting point" of the motion, i.e. where the mechanism is located when the
↪potentiometer reads 0v, is 30.
```

```
frc::AnalogPotentiometer pot{0, 180, 30};
```

Customizing the underlying AnalogInput

Note: If the user changes the scaling of the AnalogInput with oversampling, this must be reflected in the scale setting passed to the AnalogPotentiometer.

If the user would like to apply custom settings to the underlying AnalogInput used by the AnalogPotentiometer, an alternative constructor may be used in which the AnalogInput is injected:

Java

C++

```
// Initializes an AnalogInput on port 0, and enables 2-bit averaging
AnalogInput input = new AnalogInput(0);
input.setAverageBits(2);

// Initializes an AnalogPotentiometer with the given AnalogInput
// The full range of motion (in meaningful external units) is 0-180 (this could be
↪degrees, for instance)
// The "starting point" of the motion, i.e. where the mechanism is located when the
↪potentiometer reads 0v, is 30.

AnalogPotentiometer pot = new AnalogPotentiometer(input, 180, 30);
```

```
// Initializes an AnalogInput on port 0, and enables 2-bit averaging
frc::AnalogInput input{0};
input.SetAverageBits(2);

// Initializes an AnalogPotentiometer with the given AnalogInput
// The full range of motion (in meaningful external units) is 0-180 (this could be
↪degrees, for instance)
// The "starting point" of the motion, i.e. where the mechanism is located when the
↪potentiometer reads 0v, is 30.

frc::AnalogPotentiometer pot{input, 180, 30};
```

Reading values from the AnalogPotentiometer

The scaled value can be read by simply calling the get method:

Java

C++

```
pot.get();
```

```
pot.Get();
```

13.8.2 Using AnalogPotentiometers in code

Analog sensors can be used in code much in the way other sensors that measure the same thing can be. If the analog sensor is a potentiometer measuring an arm angle, it can be used similarly to an *encoder*. If it is an ultrasonic sensor, it can be used similarly to other *ultrasonics*.

It is very important to keep in mind that actual, physical potentiometers generally have a limited range of motion. Safeguards should be present in both the physical mechanism and the code to ensure that the mechanism does not break the sensor by traveling past its maximum throw.

13.9 Digital Inputs - Software

Note: This section covers digital inputs in software. For a hardware guide to digital inputs, see *Digital Inputs - Hardware*.

The roboRIO's FPGA supports up to 26 digital inputs. 10 of these are made available through the built-in DIO ports on the RIO itself, while the other 16 are available through the MXP breakout port.

Digital inputs read one of two states - "high" or "low." By default, the built-in ports on the RIO will read "high" due to internal pull-up resistors (for more information, see *Digital Inputs - Hardware*). Accordingly, digital inputs are most-commonly used with switches of some sort. Support for this usage is provided through the DigitalInput class ([Java](#), [C++](#)).

13.9.1 The DigitalInput class

A DigitalInput can be initialized as follows:

Java

C++

```
// Initializes a DigitalInput on DIO 0
DigitalInput input = new DigitalInput(0);
```

```
// Initializes a DigitalInput on DIO 0
frc::DigitalInput input{0};
```

Reading the value of the DigitalInput

The state of the DigitalInput can be polled with the get method:

Java

C++

```
// Gets the value of the digital input. Returns true if the circuit is open.
input.get();
```

```
// Gets the value of the digital input. Returns true if the circuit is open.
input.Get();
```

13.9.2 Creating a DigitalInput from an AnalogInput

Note: An AnalogTrigger constructed with a port number argument can share that analog port with a separate AnalogInput, but two *AnalogInput* objects may not share the same port.

Sometimes, it is desirable to use an analog input as a digital input. This can be easily achieved using the AnalogTrigger class (Java, C++).

An AnalogTrigger may be initialized as follows. As with AnalogPotentiometer, an AnalogInput may be passed explicitly if the user wishes to customize the sampling settings:

Java

C++

```
// Initializes an AnalogTrigger on port 0
AnalogTrigger trigger0 = new AnalogTrigger(0);

// Initializes an AnalogInput on port 1 and enables 2-bit oversampling
AnalogInput input = new AnalogInput(1);
input.setAverageBits(2);

// Initializes an AnalogTrigger using the above input
AnalogTrigger trigger1 = new AnalogTrigger(input);
```

```
// Initializes an AnalogTrigger on port 0
frc::AnalogTrigger trigger0{0};

// Initializes an AnalogInput on port 1 and enables 2-bit oversampling
frc::AnalogInput input{1};
input.SetAverageBits(2);

// Initializes an AnalogTrigger using the above input
frc::AnalogTrigger trigger1{input};
```

Setting the trigger points

Note: For details on the scaling of “raw” AnalogInput values, see [Analog Inputs - Software](#).

To convert the analog signal to a digital one, it is necessary to specify at what values the trigger will enable and disable. These values may be different to avoid “dithering” around the transition point:

Java

C++

```
// Sets the trigger to enable at a raw value of 3500, and disable at a value of 1000
trigger.setLimitsRaw(1000, 3500);

// Sets the trigger to enable at a voltage of 4 volts, and disable at a value of 1.5
↪volts
trigger.setLimitsVoltage(1.5, 4);
```

```
// Sets the trigger to enable at a raw value of 3500, and disable at a value of 1000
trigger.SetLimitsRaw(1000, 3500);

// Sets the trigger to enable at a voltage of 4 volts, and disable at a value of 1.5
↪volts
trigger.SetLimitsVoltage(1.5, 4);
```

13.9.3 Using DigitalInputs in code

As almost all switches on the robot will be used through a `DigitalInput`, this class is extremely important for effective robot control.

Limiting the motion of a mechanism

Nearly all motorized mechanisms (such as arms and elevators) in FRC® should be given some form of “limit switch” to prevent them from damaging themselves at the end of their range of motions. A short example is given below:

Java

C++

```
Spark spark = new Spark(0);

// Limit switch on DIO 2
DigitalInput limit = new DigitalInput(2);

public void autonomousPeriodic() {
    // Runs the motor forwards at half speed, unless the limit is pressed
    if(!limit.get()) {
        spark.set(.5);
    } else {
        spark.set(0);
    }
}
```

```
// Motor for the mechanism
frc::Spark spark{0};

// Limit switch on DIO 2
frc::DigitalInput limit{2};

void AutonomousPeriodic() {
    // Runs the motor forwards at half speed, unless the limit is pressed
    if(!limit.Get()) {
        spark.Set(.5);
    }
}
```

(continues on next page)

(continued from previous page)

```

    } else {
        spark.Set(0);
    }
}

```

Homing a mechanism

Limit switches are very important for being able to “home” a mechanism with a encoder. For an example of this, see [Homing a mechanism](#).

13.10 Programming Limit Switches

Limit switches are often used to control mechanisms on robots. While limit switches are simple to use, they only can sense a single position of a moving part. This makes them ideal for ensuring that movement doesn't exceed some limit but not so good at controlling the speed of the movement as it approaches the limit. For example, a rotational shoulder joint on a robot arm would best be controlled using a potentiometer or an absolute encoder, the limit switch could make sure that if the potentiometer ever failed, the limit switch would stop the robot from going too far and causing damage.

Limit switches can have “normally open” or “normally closed” outputs. This will control if a high signal means the switch is opened or closed. To learn more about limit switch hardware see this [article](#).

13.10.1 Controlling a Motor with Two Limit Switches

Java

C++

```

DigitalInput toplimitSwitch = new DigitalInput(0);
DigitalInput bottomlimitSwitch = new DigitalInput(1);
PWMVictorSPX motor = new PWMVictorSPX(0);
Joystick joystick = new Joystick(0);

@Override
public void teleopPeriodic() {
    setMotorSpeed(joystick.getRawAxis(2));
}

public void setMotorSpeed(double speed) {
    if (speed > 0) {
        if (toplimitSwitch.get()) {
            // We are going up and top limit is tripped so stop
            motor.set(0);
        } else {
            // We are going up but top limit is not tripped so go at commanded speed
            motor.set(speed);
        }
    } else {

```

(continues on next page)

(continued from previous page)

```
        if (bottomlimitSwitch.Get()) {  
            // We are going down and bottom limit is tripped so stop  
            motor.Set(0);  
        } else {  
            // We are going down but bottom limit is not tripped so go at commanded_  
↪speed  
            motor.Set(speed);  
        }  
    }  
}
```

```
frc::DigitalInput toplimitSwitch {0};  
frc::DigitalInput bottomlimitSwitch {1};  
frc::PWMVictorSPX motor {0};  
frc::Joystick joystick {0};  
  
void TeleopPeriodic() {  
    SetMotorSpeed(joystick.GetRawAxis(2));  
}  
  
void SetMotorSpeed(double speed) {  
    if (speed > 0) {  
        if (toplimitSwitch.Get()) {  
            // We are going up and top limit is tripped so stop  
            motor.Set(0);  
        } else {  
            // We are going up but top limit is not tripped so go at commanded speed  
            motor.Set(speed);  
        }  
    } else {  
        if (bottomlimitSwitch.Get()) {  
            // We are going down and bottom limit is tripped so stop  
            motor.Set(0);  
        } else {  
            // We are going down but bottom limit is not tripped so go at commanded_  
↪speed  
            motor.Set(speed);  
        }  
    }  
}
```


14.1 Using CAN Devices

CAN has many advantages over other methods of connection between the robot controller and peripheral devices.

- CAN connections are daisy-chained from device to device, which often results in much shorter wire runs than having to wire each device to the RIO itself.
- Much more data can be sent over a CAN connection than over a PWM connection - thus, CAN motor controllers are capable of a much more expansive feature-set than are PWM motor controllers.
- CAN is bi-directional, so CAN motor controllers can send data back to the RIO, again facilitating a more expansive feature-set than can be offered by PWM Controllers.

For instructions on wiring CAN devices, see the relevant section of the [robot wiring guide](#).

CAN devices generally have their own WPILib classes. The following sections will describe the use of several of these classes.

14.2 Pneumatics Control Module

The Pneumatics Control Module (PCM) is a CAN-based device that provides complete control over the compressor and up to 8 solenoids per module. The PCM is integrated into WPILib through a series of classes that make it simple to use.

The closed loop control of the Compressor and Pressure switch is handled by the Compressor class ([Java](#), [C++](#)), and the Solenoids are handled by the Solenoid ([Java](#), [C++](#)) and DoubleSolenoid ([Java](#), [C++](#)) classes.

An additional PCM module can be used where the modules corresponding solenoids are differentiated by the module number in the constructors of the Solenoid and Compressor classes.

For more information on controlling the compressor, see [Operating a Compressor for Pneumatics](#).

For more information on controlling solenoids, see [Operating Pneumatic Cylinders](#).

14.3 Power Distribution Panel

The Power Distribution Panel (PDP) can use its CAN connectivity to communicate a wealth of status information regarding the robot's power use to the roboRIO, for use in user code. The PDP has the capability to report its current temperature, the bus voltage, the total robot current draw, the total robot energy use, and the individual current draw of each device power channel. These data can be used for a number of advanced control techniques, such as motor torque limiting and brownout avoidance.

14.3.1 Creating a PDP Object

To use the PDP, create an instance of the `PowerDistributionPanel` class (Java, C++):

Java

C++

```
PowerDistributionPanel examplePDP = new PowerDistributionPanel(0);
```

```
PowerDistributionPanel examplePDP{0};
```

Note: it is not necessary to create a `PowerDistributionPanel` object unless you need to read values from it. The board will work and supply power on all the channels even if the object is never created.

Warning: To enable voltage and current logging in the Driver Station, the CAN ID for the PDP *must* be 0.

14.3.2 Reading the Bus Voltage

Java

C++

```
examplePDP.getVoltage();
```

```
examplePDP.GetVoltage();
```

Monitoring the bus voltage can be useful for (among other things) detecting when the robot is near a brownout, so that action can be taken to avoid brownout in a controlled manner. See the [roboRIO Brownouts document](#) for more information.

14.3.3 Reading the Temperature

Java

C++

```
examplePDP.getTemperature();
```

```
examplePDP.GetTemperature();
```

Monitoring the temperature can be useful for detecting if the robot has been drawing too much power and needs to be shut down for a while, or if there is a short or other wiring problem.

14.3.4 Reading the Total Current and Energy

Java

C++

```
examplePDP.getTotalCurrent();  
examplePDP.getTotalEnergy();
```

```
examplePDP.GetTotalCurrent();  
examplePDP.GetTotalEnergy();
```

Monitoring the total current and total energy (the total energy is simply the total current multiplied by the bus voltage) can be useful for controlling how much power is being drawn from the battery, both for preventing brownouts and ensuring that mechanisms have sufficient power available to perform the actions required.

14.3.5 Reading Individual Channel Currents

The PDP also allows users to monitor the current drawn by the individual device power channels. For example, to read the current on channel 0:

Java

C++

```
examplePDP.getCurrent(0);
```

```
examplePDP.GetCurrent(0);
```

Monitoring individual device current draws can be useful for detecting shorts or stalled motors.

14.4 Third-Party CAN Devices

A number of FRC® vendors offer their own CAN peripherals. As CAN devices offer expansive feature-sets, vendor CAN devices require similarly expansive code libraries to operate. As a result, these libraries are not maintained as an official part of WPILib, but are instead maintained by the vendors themselves. For a guide to installing third-party libraries, see [3rd Party Libraries](#)

A list of common third-party CAN devices from various vendors, along with links to corresponding external documentation, is provided below:

14.4.1 Cross-the-Road Electronics

Cross-the-Road Electronics (CTRE) offers several CAN peripherals with external libraries. General resources for all CTRE devices include:

- [Phoenix Device Software Documentation](#)
- [Phoenix Device Software Examples](#)

CTRE Motor Controllers

- **Talon FX (with Falcon 500 Motor)**
 - [API Documentation \(Java, C++\)](#)
 - [Hardware User's Manual](#)
 - [Other Resources](#)
- **Talon SRX**
 - [API Documentation \(Java, C++\)](#)
 - [Hardware User's Manual](#)
 - [Other Resources](#)
- **Victor SPX**
 - [API Documentation \(Java, C++\)](#)
 - [Hardware User's Manual](#)
 - [Other Resources](#)

CTRE Sensors

- **CANcoder**
 - [API Documentation\(Java, C++\)](#)
 - [Hardware User's Manual](#)
 - [Other Resources](#)
- **Pigeon IMU**
 - [API Documentation\(Java, C++\)](#)

- [Hardware User's Manual](#)
- [Other Resources](#)
- **CANifier**
 - [API Documentation \(Java, C++\)](#)
 - [Hardware User's Manual](#)
 - [Other Resources](#)

14.4.2 REV Robotics

REV Robotics currently offers the SPARK MAX motor controller, which has a similar feature-set to the Talon SRX.

REV Motor Controllers

- **SPARK MAX**
 - [API Documentation \(Java, C++\)](#)
 - [Technical Manual](#)

14.4.3 Playing With Fusion

Playing With Fusion (PWF) offers the Venom integrated motor/controller as well as a Time-of-Flight distance sensor:

PWF Motor Controllers

- **Venom**
 - [API Documentation \(Java, C++\)](#)
 - [Technical Manual](#)

PWF Sensors

- **Time of Flight Sensor**
 - [API Documentation\(Java, C++\)](#)
 - [Technical Manual](#)

14.5 FRC CAN Device Specifications

This document seeks to describe the basic functions of the current FRC® CAN system and the requirements for any new CAN devices seeking to work with the system.

14.5.1 Addressing

FRC CAN nodes assign arbitration IDs based on a pre-defined scheme that breaks the ID into 5 components:

Device Type

This is a 5-bit value describing the type of device being addressed. A table of currently assigned device types can be found below. If you wish to have a new device type assigned from the Reserved pool, please submit a request to FIRST.

Device Types	
Broadcast Messages	0
Robot Controller	1
Motor Controller	2
Relay Controller	3
Gyro Sensor	4
Accelerometer	5
Ultrasonic Sensor	6
Gear Tooth Sensor	7
Power Distribution Module	8
Pneumatics Controller	9
Miscellaneous	10
IO Breakout	11
Reserved	12-30
Firmware Update	31

Manufacturer

This is an 8-bit value indicating the manufacturer of the CAN device. Currently assigned values can be found in the table below. If you wish to have a manufacturer ID assigned from the Reservedpool, please submit a request to FIRST.

Manufacturer	
Broadcast	0
NI	1
Luminary Micro	2
DEKA	3
CTR Electronics	4
REV Robotics	5
Grapple	6
MindSensors	7
Team Use	8
Kauai Labs	9
Copperforge	10
Playing With Fusion	11
Studica	12
Reserved	13-255

API/Message Identifier

The API or Message Identifier is a 10-bit value that identifies a particular command or message type. These identifiers are unique for each Manufacturer + Device Type combination (so an API identifier that may be a “Voltage Set” for a Luminary Micro Motor Controller may be a “Status Get” for a CTR Electronics Motor Controller or Current Get for a CTR Power Distribution Module).

The Message identifier is further broken down into 2 sub-fields: the 6-bit API Class and the 4-bit API Index.

API Class

The API Class is a 6-bit identifier for an API grouping. Similar messages are grouped into a single API Class. An example of the API Classes for the Jaguar Motor Controller is shown in the table below.

API Class	
Voltage Control Mode	0
Speed Control Mode	1
Voltage Compensation Mode	2
Position Control Mode	3
Current Control Mode	4
Status	5
Periodic Status	6
Configuration	7
Ack	8

API Index

The API Index is a 4-bit identifier for a particular message within an API Class. An example of the API Index values for the Jaguar Motor Controller Speed Control API Class is shown in the table below.

API Index	
Enable Control	0
Disable Control	1
Set Setpoint	2
P Constant	3
I Constant	4
D Constant	5
Set Reference	6
Trusted Enable	7
Trusted Set No Ack	8
Trusted Set Setpoint No Ack	10
Set Setpoint No Ack	11

Device Number

Device Number is a 6-bit quantity indicating the number of the device of a particular type. Devices should default to device ID 0 to match other components of the FRC Control System. Device 0x3F may be reserved for device specific broadcast messages.

Example

Speed Control Mode Disable from Luminary Micro Jaguar Speed Controller (dev # 4)

Field	Device Type	Manufacturer Code	API		Device Number
Value	2	2	1	1	4
Bits	0 0 0 1 0	0 0 0 0 0 0 1 0	0 0 0 0 0 1	0 0 0 0 0 0	0 1 0 0
Bit Position	28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10	9 8 7 6 5 4	3 2 1 0

14.5.2 Protected Frames

FRC CAN Nodes which implement actuator control capability (motor controllers, relays, pneumatics controllers, etc.) must implement a way to verify that the robot is enabled and that commands originate with the main robot controller (i.e. the roboRIO).

14.5.3 Broadcast Messages

Broadcast messages are messages sent to all nodes by setting the device type and manufacturer fields to 0. The API Class for broadcast messages is 0. The currently defined broadcast messages are shown in the table below:

Description	
Disable	0
System Halt	1
System Reset	2
Device Assign	3
Device Query	4
Heartbeat	5
Sync	6
Update	7
Firmware Version	8
Enumerate	9
System Resume	10

Devices should disable immediately when receiving the Disable message (arbID 0), implementation of other broadcast messages is optional.

14.5.4 Requirements for FRC CAN Nodes

For CAN Nodes to be accepted for use in the FRC System, they must:

- Communicate using Arbitration IDs which match the prescribed FRC format:
 - A valid, issued CAN Device Type (per Table 1 - CAN Device Types)
 - A valid, issued Manufacturer ID (per Table 2 - CAN Manufacturer Codes)
 - API Class(es) and Index(s) assigned and documented by the device manufacturer
 - A user selectable device number if multiple units of the device type are intended to co-exist on the same network.
- Support the minimum Broadcast message requirements as detailed in the Broadcast Messages section.
- If controlling actuators, utilize a scheme to assure that the robot is issuing commands, is enabled, and is still present
- Provide software library support for LabVIEW, C++, and Java or arrange with *FIRST*® or FIRSTs Control System Partners to provide such interfaces.

15.1 Git Version Control Introduction

Important: A more in-depth guide on Git is available on the [Git website](#).

[Git](#) is a Distributed Version Control System (VCS) created by Linus Torvalds, also known for creating and maintaining the Linux kernel. Version Control is a system for tracking changes of code for developers. The advantages of Git Version Control are:

- Separate testing environments into *branches*
- Ability to navigate to a particular *commit* without removing history
- Ability to manage *commits* in various ways, including combining them
- Various other features, see [here](#)

15.1.1 Prerequisites

Important: This tutorial uses the Windows operating system

You have to download and install Git from the following links:

- [Windows](#)
- [macOS](#)
- [Linux](#)

Note: You may need to add Git to your [path](#)

15.1.2 Git Vocabulary

Git revolves around several core commands:

- **Repository:** the data structure of your code, including a `.git` folder in the root directory
- **Commit:** a particular saved state of the repository, this includes all files and additions
- **Branch:** a means of separating various commits, having a unique history. This is primarily used for separating development and stable branches.
- **Push:** update the remote repository with your local changes
- **Pull:** update your local repository with the remote changes
- **Clone:** retrieving a local copy of a repository to modify
- **Fork:** duplicating a pre-existing repository to modify, and to compare against the original
- **Merge:** combining various changes from different branches/commits/forks into a single history

15.1.3 Repository

A Git repository is a data structure containing the structure, history, and files of a project.

Git repositories usually consist of:

- A `.git` folder. This folder contains the various information about the repository.
- A `.gitignore` file. This file contains the files or directories that you do *not* want included when you commit.
- Files and folders. This is the main content of the repository.

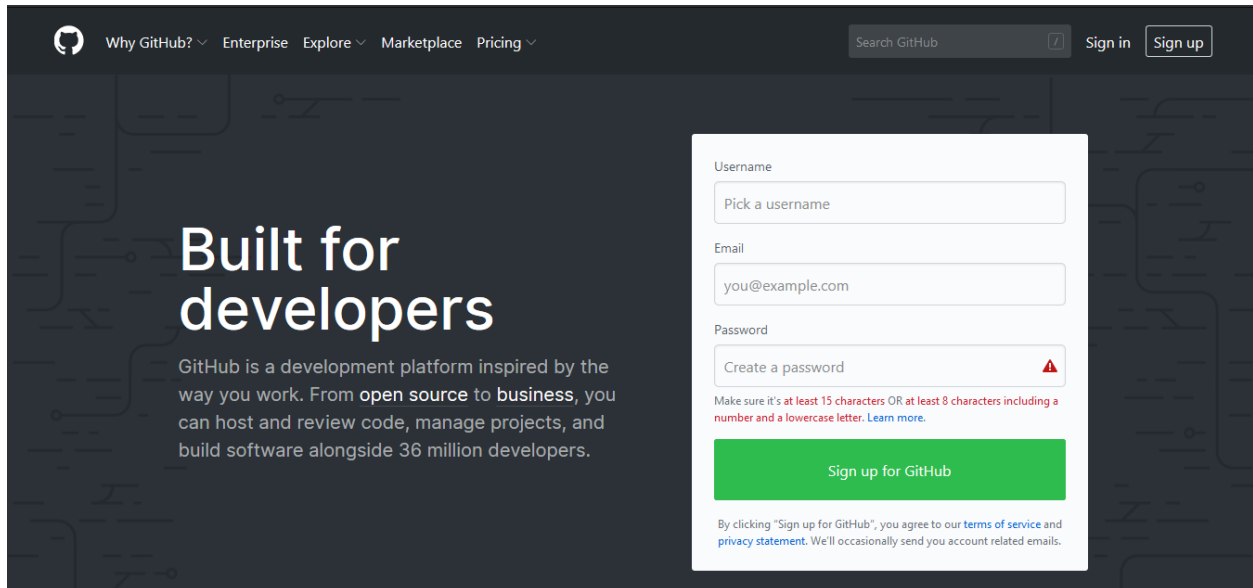
Creating the repository

You can store the repository locally, or through a remote. A remote being the cloud, or possibly another storage medium that hosts your repository. [GitHub](#) is a popular free hosting service. Numerous developers use it, and that's what this tutorial will use.

Note: There are various providers that can host repositories. [Gitlab](#) and [Bitbucket](#) are a few alternatives to Github.

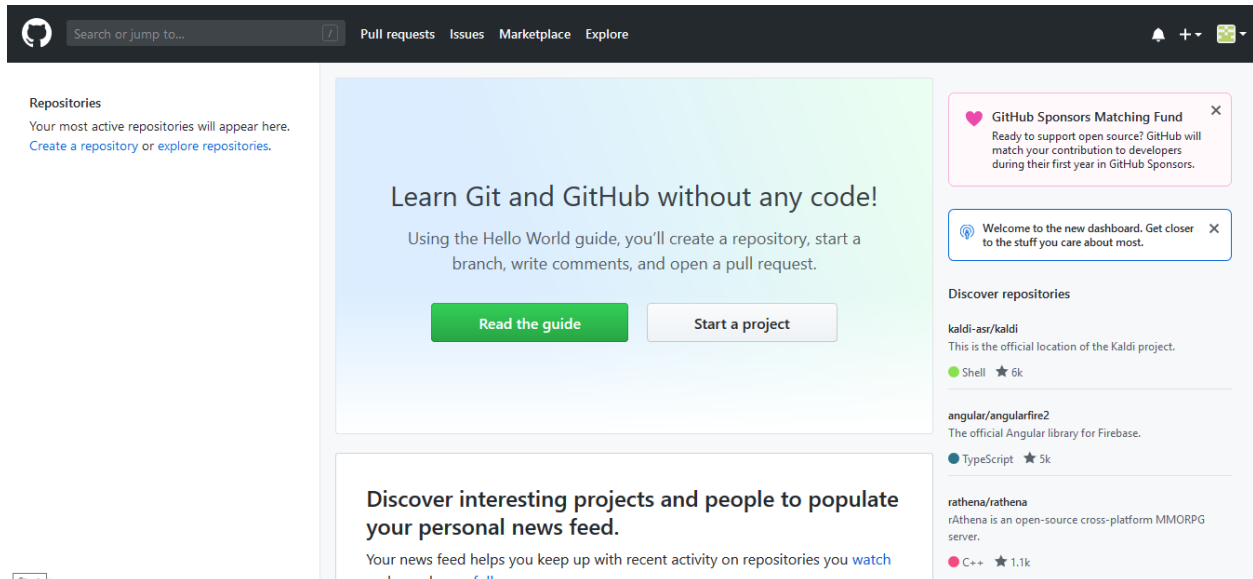
Creating a GitHub Account

Go ahead and create a GitHub account by visiting the [website](#) and following the own screen prompts.

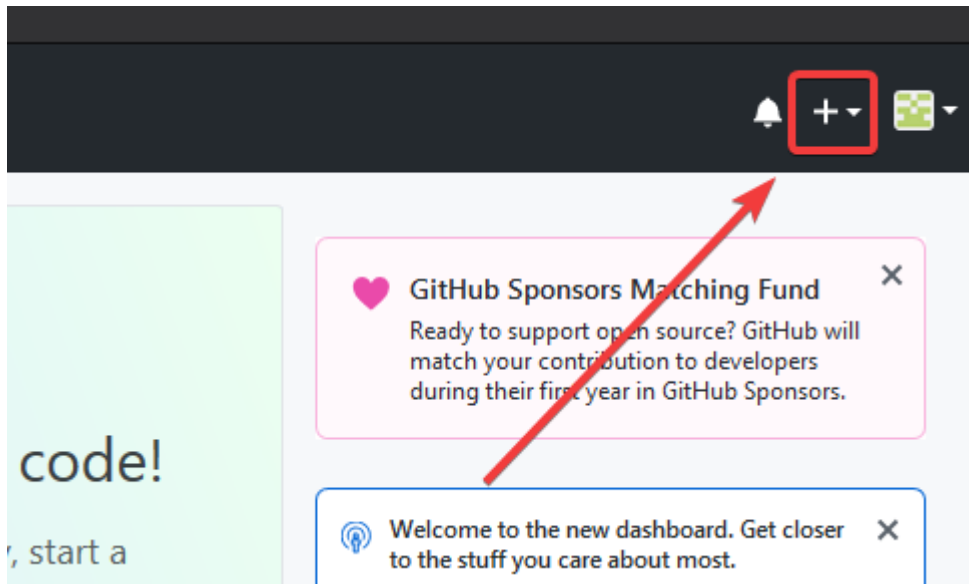


Local Creation

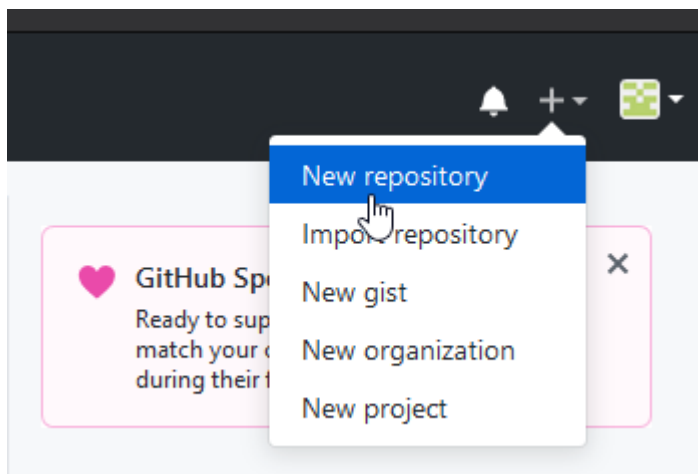
After creating and verifying your account, you'll want to visit the homepage. It'll look similar to the shown image.



Click the plus icon in the top right.



Then click “New Repository”




Fill out the appropriate information, and then click “Create repository”

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)



Owner

Repository name *

 ExampleUser9007 ▾ / ExampleRepo ✓

Great repository names are short and memorable. Need inspiration? How about [reimagined-palm-tree?](#)

Description (optional)

- ☒  **Public**
Anyone can see this repository. You choose who can commit.
- ☐  **Private**
You choose who can see and commit to this repository.

☐ **Initialize this repository with a README**

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None** ▾



Add a license: **None** ▾



Create repository

You should see a screen similar to this


Quick setup — if you've done this kind of thing before

 Set up in Desktop or **HTTPS** **SSH** 

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).


...or create a new repository on the command line

```
echo "# ExampleRepo" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/ExampleUser9007/ExampleRepo.git
git push -u origin master
```




...or push an existing repository from the command line

```
git remote add origin https://github.com/ExampleUser9007/ExampleRepo.git
git push -u origin master
```



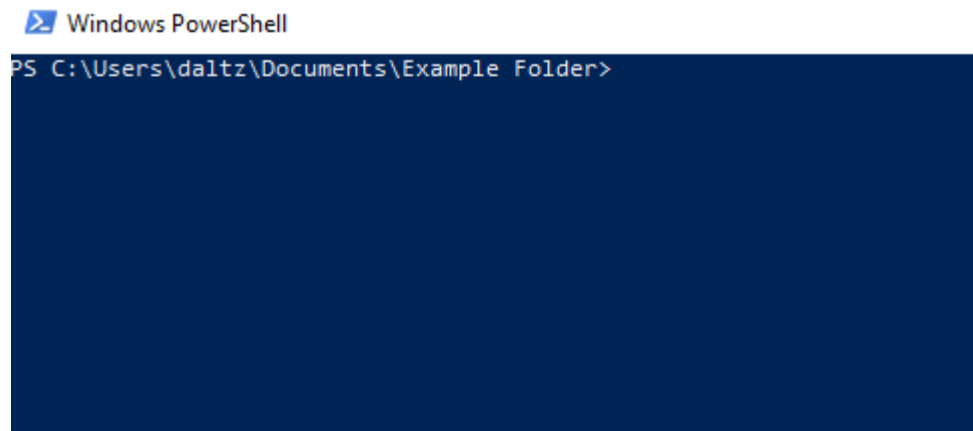
...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

 Import code

Note: The keyboard shortcut `Ctrl+~` can be used to open a terminal in Visual Studio Code.

Now you'll want to open a PowerShell window and navigate to your project directory. An excellent tutorial on PowerShell can be found [here](#). Please consult your search engine on how to open a terminal on alternative operating systems.



If a directory is empty, a file needs to be created in order for git to have something to track. In the below Empty Directory example, we created a file called `README.md` with the contents of `# Example Repo`. For FRC® Robot projects, the below Existing Project commands should be run in the root of a project *created by the VS Code WPILib Project Creator*. More details on the various commands can be found in the subsequent sections.

Note: Replace the filepath "C:\Users\ExampleUser9007\Documents\Example Folder" with the one you want to create the repo in, and replace the remote URL `https://github.com/ExampleUser9007/ExampleRepo.git` with the URL for the repo you created in the previous steps.

Empty Directory

Existing Project

```
> cd "C:\Users\ExampleUser9007\Documents\Example Folder"
> git init
Initialized empty Git repository in C:/Users/ExampleUser9007/Documents/Example Folder/.git/
> echo "# ExampleRepo" >> README.md
> git add README.md
> git commit -m "First commit"
[main (root-commit) fafafa] First commit
1 file changed, 1 insertions(+), 0 deletions(-)
create mode 100644 README.md
> git remote add origin https://github.com/ExampleUser9007/ExampleRepo.git
> git push -u origin main
```

```
> cd "C:\Users\ExampleUser9007\Documents\Example Folder"
> git init
Initialized empty Git repository in C:/Users/ExampleUser9007/Documents/Example Folder/.git/
> git add .
> git commit -m "First commit"
[main (root-commit) fafafa] First commit
1 file changed, 1 insertions(+), 0 deletions(-)
create mode 100644 README.md
> git remote add origin https://github.com/ExampleUser9007/ExampleRepo.git
> git push -u origin main
```

15.1.4 Commits

Repositories are primarily composed of commits. Commits are saved states or *versions* of code.

In the previous example, we created a file called README.md. Open that file in your favorite text editor and edit a few lines. After tinkering with the file for a bit, simply save and close. Navigate to PowerShell and type the following commands.

```
> git add README.md
> git commit -m "Adds a description to the repository"
[main bcbcbc] Adds a description to the repository
1 file changed, 2 insertions(+), 0 deletions(-)
> git push
```

Note: Writing good commit messages is a key part of a maintainable project. A guide on writing commit messages can be found [here](#).

Git Pull

Note: `git fetch` can be used when the user does not wish to automatically merge into the current working branch

This command retrieves the history or commits from the remote repository. When the remote contains work you do not have, it will attempt to automatically merge. See [Merging](#).

Run: `git pull`

Git Add

This command adds a selected file(s) to a commit. To commit every file/folder that isn't excluded via *gitignore*.

Run: `git add FILENAME.txt` where `FILENAME.txt` is the name and extension of the file to add to a commit. Run: `git add .` will add every untracked, unexcluded file when ran in the root of the repository.

Git Commit

This command creates the commit and stores it locally. This saves the state and adds it to the repositories history.

Run: `git commit -m "type message here"`

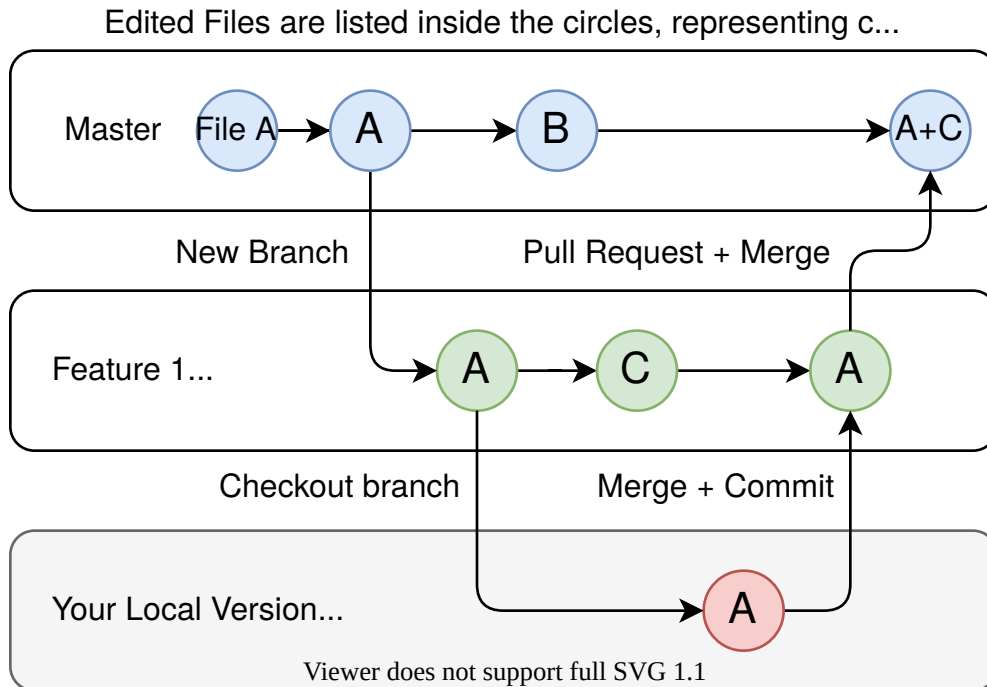
Git Push

Upload (Push) your local changes to the remote (Cloud)

Run: `git push`

15.1.5 Branches

Branches are a similar to parallel worlds to Git. They start off the same, and then they can “branch” out into different varying paths. Consider the Git control flow to look similar to this.



In the above example, main was branched (or duplicated) into the branch Feature 1 and someone checked out the branch, creating a local copy. Then, someone committed (or uploaded) their changes, merging them into the branch Feature 1. You are “merging” the changes from one branch into another.

Creating a Branch

Run: `git branch branch-name` where `branch-name` is the name of the branch to create. The new branch history will be created from the current active branch.

Entering a Branch

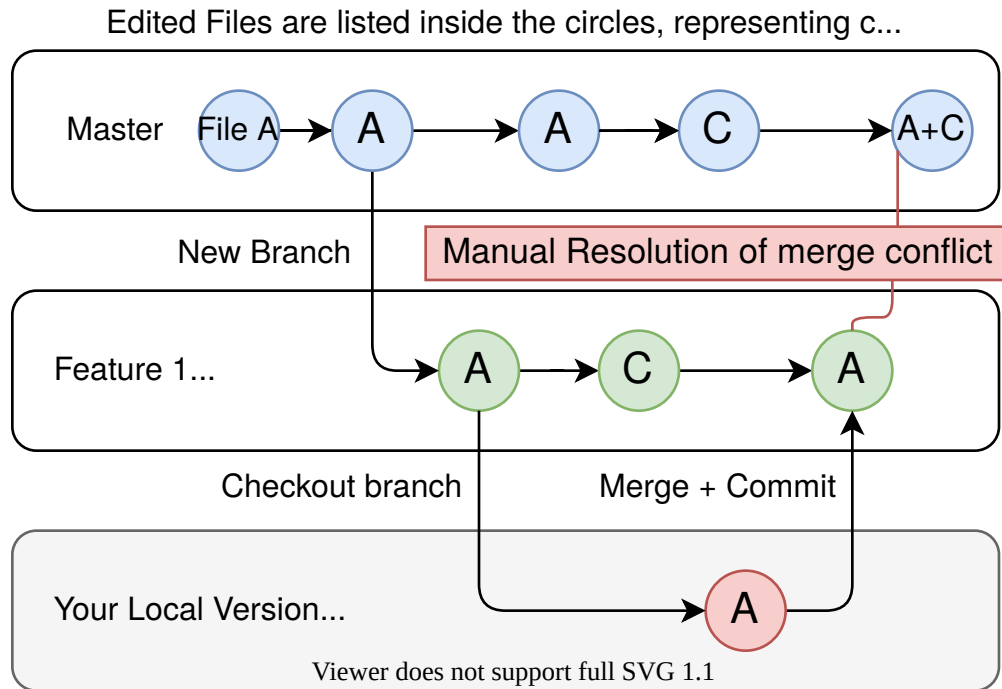
Once a branch is created, you have to then enter the branch.

Run: `git checkout branch-name` where `branch-name` is the branch that was previously created.

15.1.6 Merging

In scenarios where you want to copy one branches history into another, you can merge them. A merge is done by calling `git merge branch-name` with `branch-name` being the name of the branch to merge from. It is automatically merged in the current active branch.

It’s common for a remote repository to contain work (history) that you do not have. Whenever you run `git pull`, it will attempt to automatically merge those commits. That merge may look like the below.



However, in the above example, what if File 1 was modified by both branch FeatureA and FeatureB? This is called a **merge conflict**. A merge conflict will can be resolved by editing the conflicting file. In the example, we would need to edit File 1 to keep the history or changes that we want. After that has been done. Simply re-add, re-commit, and then push your changes.

15.1.7 Resets

Sometimes history needs to be reset, or a commit needs to be undone. This can be done multiple ways.

Reverting the Commit

Note: You cannot revert a merge, as git does not know which branch or origin it should choose.

To revert history leading up to a commit run `git revert commit-id`. The commit IDs can be shown using the `git log` command.

Resetting the Head

Warning: Forcibly resetting the head is a dangerous command. It permanently erases all history past the target. You have been warned!

Run: `git reset --hard commit-id`.

15.1.8 Forks

Forks can be treated similarly to branches. You can merge the upstream (original repository) into the origin (forked repository).

Cloning an Existing Repo

In the situation that a repository is already created and stored on a remote, you can clone it using

```
git clone https://github.com/myrepo.git
```

where `myrepo.git` is replaced with your git repo. If you follow this, you can skip to `:ref:commits <docs/software/basic-programming/git-getting-started:Commits>`.

Updating a Fork

1. Add the upstream: `git remote add upstream https://github.com/ORIGINAL_OWNER/ORIGINAL_REPOSITORY.git`
2. Confirm it was added via: `git remote -v`
3. Pull changes from upstream: `git fetch upstream`
4. Merge the changes into head: `git merge upstream/upstream-branch-name`

15.1.9 Gitignore

Important: It is extremely important that teams **do not** modify the `.gitignore` file that is included with their robot project. This can lead to offline deployment not working.

A `.gitignore` file is commonly used as a list of files to not automatically commit with `git add`. Any files or directory listed in this file will **not** be committed. They will also not show up with `git status`.

Additional Information can be found [here](#)

Hiding a Folder

Simply add a new line containing the folder to hide, with a forward slash at the end

EX: directory-to-exclude/

Hiding a File

Add a new line with the name of the file to hide, including any prepending directory relative to the root of the repository.

EX: directory/file-to-hide.txt

EX: file-to-hide2.txt

15.1.10 Additional Information

A much more in-depth tutorial can be found at the official [git](#) website.

A guide for correcting common mistakes can be found at the [git flight rules](#) repository.

15.2 The C++ Units Library

The 2020 release of WPILib is coupled with a [Units](#) library for C++ teams. This library leverages the C++ [type system](#) to enforce proper dimensionality for method parameters, automatically perform unit conversions, and even allow users to define arbitrary defined unit types. Since the C++ type system is enforced at compile-time, the library has essentially no runtime cost.

15.2.1 Using the Units Library

The units library is a header-only library. You must include the relevant header in your source files for the units you want to use. Here's a list of available units.

```
#include <units/acceleration.h>
#include <units/angle.h>
#include <units/angular_acceleration.h>
#include <units/angular_velocity.h>
#include <units/area.h>
#include <units/capacitance.h>
#include <units/charge.h>
#include <units/concentration.h>
#include <units/conductance.h>
#include <units/current.h>
#include <units/curvature.h>
#include <units/data.h>
#include <units/data_transfer_rate.h>
#include <units/density.h>
#include <units/dimensionless.h>
#include <units/energy.h>
#include <units/force.h>
```

(continues on next page)

(continued from previous page)

```

#include <units/frequency.h>
#include <units/illuminance.h>
#include <units/impedance.h>
#include <units/inductance.h>
#include <units/length.h>
#include <units/luminous_flux.h>
#include <units/luminous_intensity.h>
#include <units/magnetic_field_strength.h>
#include <units/magnetic_flux.h>
#include <units/mass.h>
#include <units/moment_of_inertia.h>
#include <units/power.h>
#include <units/pressure.h>
#include <units/radiation.h>
#include <units/solid_angle.h>
#include <units/substance.h>
#include <units/temperature.h>
#include <units/time.h>
#include <units/torque.h>
#include <units/velocity.h>
#include <units/voltage.h>
#include <units/volume.h>

```

The `units/math.h` header provides unit-aware functions like `units::math::abs()`.

Unit Types and Container Types

The C++ units library is based around two sorts of type definitions: unit types and container types.

Unit Types

Unit types correspond to the abstract concept of a unit, without any actual stored value. Unit types are the fundamental “building block” of the units library - all unit types are defined constructively (using the `compound_unit` template) from a small number of “basic” unit types (such as meters, seconds, etc).

While unit types cannot contain numerical values, their use in building other unit types means that when a type or method uses a [template parameter](#) to specify its dimensionality, that parameter will be a unit type.

Container Types

Container types correspond to an actual quantity dimensioned according to some unit - that is, they are what actually hold the numerical value. Container types are constructed from unit types with the `unit_t` template. Most unit types have a corresponding container type that has the same name suffixed by `_t` - for example, the unit type `units::meter` corresponds to the container type `units::meter_t`.

Whenever a specific quantity of a unit is used (as a variable or a method parameter), it will be an instance of the container type. By default, container types will store the actual value as a `double` - advanced users may change this by calling the `unit_t` template manually.

A full list of unit and container types can be found in the [documentation](#).

Creating Instances of Units

To create an instance of a specific unit, we create an instance of its container type:

```
// The variable speed has a value of 5 meters per second.
units::meter_per_second_t speed{5.0};
```

Alternatively, the units library has [type literals](#) defined for some of the more common container types. These can be used in conjunction with type inference via `auto` to define a unit more succinctly:

```
// The variable speed has a value of 5 meters per second.
auto speed = 5_mps;
```

Units can also be initialized using a value of another container type, as long as the types can be converted between one another. For example, a `meter_t` value can be created from a `foot_t` value.

```
auto feet = 6_ft;
units::meter_t meters{feet};
```

In fact, all container types representing convertible unit types are *implicitly convertible*. Thus, the following is perfectly legal:

```
units::meter_t distance = 6_ft;
```

In short, we can use *any* unit of length in place of *any other* unit of length, anywhere in our code; the units library will automatically perform the correct conversion for us.

Performing Arithmetic with Units

Container types support all of the ordinary arithmetic operations of their underlying data type, with the added condition that the operation must be *dimensionally* sound. Thus, addition must always be performed on two compatible container types:

```
// Add two meter_t values together
auto sum = 5_m + 7_m; // sum is 12_m

// Adds meters to feet; both are length, so this is fine
auto sum = 5_m + 7_ft;

// Tries to add a meter_t to a second_t, will throw a compile-time error
auto sum = 5_m + 7_s;
```

Multiplication may be performed on any pair of container types, and yields the container type of a compound unit:

Note: When a calculation yields a compound unit type, this type will only be checked for validity at the point of operation if the result type is specified explicitly. If `auto` is used, this check will not occur. For example, when we divide distance by time, we may want to

ensure the result is, indeed, a velocity (i.e. `units::meter_per_second_t`). If the return type is declared as `auto`, this check will not be made.

```
// Multiply two meter_t values, result is square_meter_t
auto product = 5_m * 7_m; // product is 35_sq_m
```

```
// Divide a meter_t value by a second_t, result is a meter_per_second_t
units::meter_per_second_t speed = 6_m / 0.5_s; // speed is 12_mps
```

<cmath> Functions

Some `std` functions (such as `clamp`) are templated to accept any type on which the arithmetic operations can be performed. Quantities stored as container types will work with these functions without issue.

However, other `std` functions work only on ordinary numerical types (e.g. `double`). The units library's `units::math` namespace contains wrappers for several of these functions that accept units. Examples of such functions include `sqrt`, `pow`, etc.

```
auto area = 36_sq_m;
units::meter_t sideLength = units::math::sqrt(area);
```

Removing the Unit Wrapper

To convert a container type to a raw numeric value, the `to<...>()` method can be used, where the template argument is the underlying type.

```
units::meter_t distance = 6.5_m;
double distanceMeters = distance.to<double>();
```

15.2.2 Example of the Units Library in WPILib Code

Several arguments for methods in new features of WPILib (ex. *kinematics*) use the units library. Here is an example of *sampling a trajectory*.

```
// Sample the trajectory at 1.2 seconds. This represents where the robot
// should be after 1.2 seconds of traversal.
Trajectory::State point = trajectory.Sample(1.2_s);

// Since units of time are implicitly convertible, this is exactly equivalent to the
// above code
Trajectory::State point = trajectory.Sample(1200_ms);
```

Some WPILib classes represent objects that could naturally work with multiple choices of unit types - for example, a motion profile might operate on either linear distance (e.g. meters) or angular distance (e.g. radians). For such classes, the unit type is required as a template parameter:

```
// Creates a new set of trapezoidal motion profile constraints  
// Max velocity of 10 meters per second  
// Max acceleration of 20 meters per second squared  
frc::TrapezoidProfile<units::meters>::Constraints{10_mps, 20_mps_sq};  
  
// Creates a new set of trapezoidal motion profile constraints  
// Max velocity of 10 radians per second  
// Max acceleration of 20 radians per second squared  
frc::TrapezoidProfile<units::radians>::Constraints{10_rad_per_s, 20__rad_per_s / 1_s};
```

For more detailed documentation, please visit the official [GitHub page](#) for the units library.

Support Resources

In addition to the documentation here, there are a variety of other resources available to FRC® teams to help understand the Control System and software.

16.1 Other Documentation

In addition to this site there are a few other places teams may check for documentation:

- [NI FRC Community Documents Section](#)
- [FIRST Inspires Technical Resources Page](#)
- [CTRE Product Pages](#)

16.2 Forums

Stuck? Have a question not answered by the documentation? Official Support is provided on these forums:

- [NI FRC Community Discussion Section](#) (roboRIO, LabVIEW and Driver Station software questions)
- [FIRST Inspires Control System Forum](#) (wiring, hardware and Driver Station questions)
- [FIRST Inspires Programming Forum](#) (programming questions for C++, Java, or LabVIEW)
- [NI FRC Support Forum](#)

16.3 CTRE Support

Support for Cross The Road Electronics components (Pneumatics Control Module, Power Distribution Panel, Talon SRX, and Voltage Regulator Module) is provided via the email address support@crosstheroadelectronics.com

16.4 REV Robotics Support

Support for REV Robotics components (SPARK MAX, Sensors, upcoming new control system components) is provided via phone at [844-255-2267](tel:844-255-2267) or via the email address support@revrobotics.com

16.5 Bug Reporting

Found a bug? Let us know by reporting it in the Issues section of the appropriate WPILibSuite project on Github: <https://github.com/wpilibsuite>

- accelerometer** A common sensor used to measure acceleration in one or more axis.
- alliance** A cooperative of up to four (4) FIRST® Robotics Competition teams.
- auto** The first phase of each *match* is called Autonomous (auto) and consists of the first fifteen (0:15) seconds.
- COTS** Commercial off the shelf, a standard (i.e. not custom order) part commonly available from a vendor to all teams for purchase.
- C++** One of the three officially supported programming languages.
- deprecated** Software that will be maintained for at least 1 year, but may be removed after that.
- DHCP** Dynamic Host Configuration Protocol, the protocol that allows a central device to assign unique IP addresses to all other devices.
- FMS** Field Management System, the electronics core responsible for sensing and controlling the FIRST Robotics Competition field.
- GradleRIO** The mechanism that powers the deployment of robot code to the roboRIO.
- gyroscope** A device that measures rate of rotation. It can add up the rotation measurements to determine *heading* of the robot. (“gyro”, for short)
- heading** The direction the robot is pointed, usually expressed as an angle in degrees.
- IMU** Inertial Measurement Unit, a sensor that combines both an *accelerometer* and a *gyro-scope* into a single sensor.
- Java** One of the three officially supported programming languages.
- KOP** Kit of Parts, the collection of items listed on the Kickoff Kit checklists, distributed to the team via FIRST Choice, or paid for completely (except shipping) with a Product Donation Voucher (PDV).
- KOP chassis** The *AM14U4* chassis distributed to every team (that did not opt out) as part of the *KOP*.
- LabVIEW** One of the three officially supported programming languages.
- match** A two (2) minute and thirty (30) second period of time in which an *alliance* plays a FRC game.

NetworkTables A way to communicate key / value pairs, of data, between programs.

RP Ranking Point, a way of ordering teams rewarding specific objectives in addition to winning the match.

simulation A way for teams to test their code without having an actual robot available.

teleop The second phase of each match is called the Teleoperated Period (teleop) and consists of drivers controlling their robots.

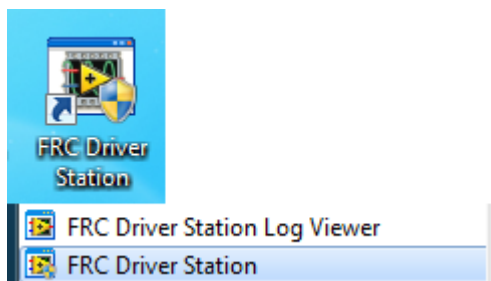
trajectory A trajectory is a smooth curve, with velocities and accelerations at each point along the curve, connecting two endpoints on the field.

18.1 FRC Driver Station Powered by NI LabVIEW

This article describes the use and features of the FRC® Driver Station Powered by NI LabVIEW.

For information on installing the Driver Station software see [this document](#).

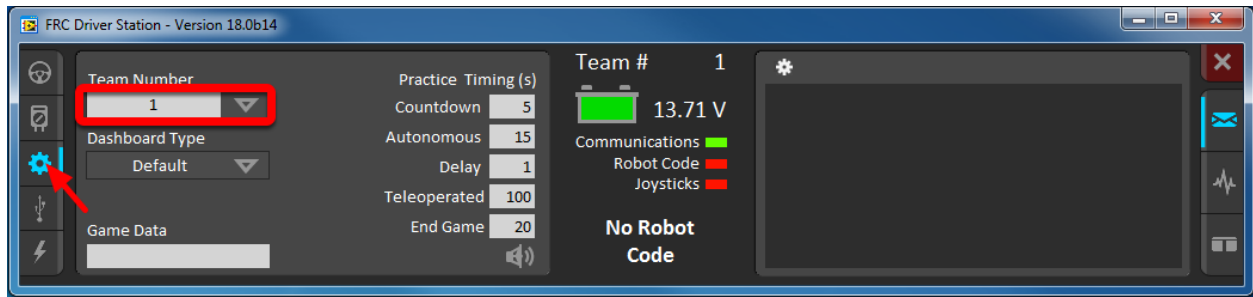
18.1.1 Starting the FRC Driver Station



The FRC Driver Station can be launched by double-clicking the icon on the Desktop or by selecting Start->All Programs->FRC Driver Station.

Note: By default the FRC Driver Station launches the [LabVIEW Dashboard](#). It can also be configured on [Setup Tab](#) to launch the other Dashboards: [SmartDashboard](#) and [Shuffleboard](#).

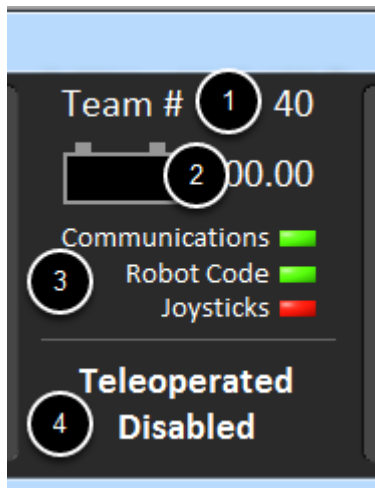
18.1.2 Setting Up the Driver Station



The DS should be set to your team number in order to connect to your robot. In order to do this click the Setup tab then enter your team number in the team number box. Press return or click outside the box for the setting to take effect.

PCs will typically have the correct network settings for the DS to connect to the robot already, but if not, make sure your Network adapter is set to DHCP.

18.1.3 Status Pane



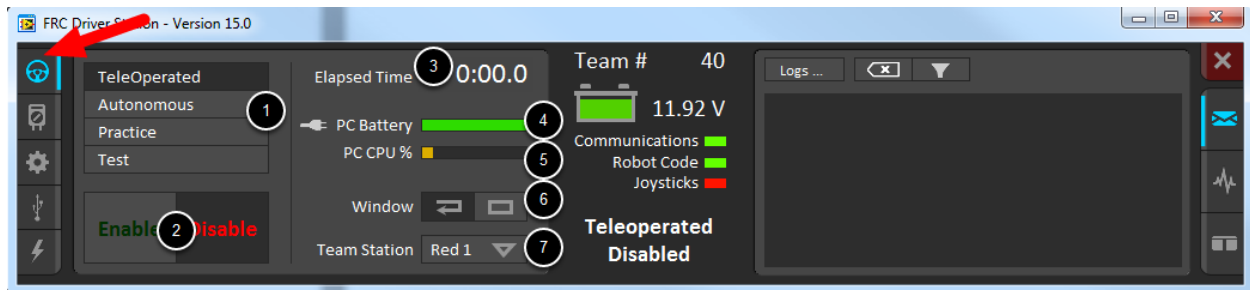
The Status Pane of the Driver Station is located in the center of the display and is always visible regardless of the tab selected. It displays a selection of critical information about the state of the DS and robot:

1. **Team #** - The Team number the DS is currently configured for. This should match your FRC team number. To change the team number see the Setup Tab.
2. **Battery Voltage** - If the DS is connected and communicating with the roboRIO this displays current battery voltage as a number and with a small chart of voltage over time in the battery icon. The background of the numeric indicator will turn red when the roboRIO brownout is triggered. See [roboRIO Brownout and Understanding Current Draw](#) for more information.
3. **Major Status Indicators** - These three indicators display major status items for the DS. The “Communications” indicates whether the DS is currently communicating with the FRC Network Communications Task on the roboRIO (it is split in half for the TCP and UDP communication). The “Robot Code” indicator shows whether the team Robot Code

is currently running (determined by whether or not the Driver Station Task in the robot code is updating the battery voltage), The “Joysticks” indicator shows if at least one joystick is plugged in and recognized by the DS.

4. Status String - The Status String provides an overall status message indicating the state of the robot, some examples are “No Robot Communication”, “No Robot Code”, “Emergency Stopped”, and “Teleoperated Enabled”. When the roboRIO brownout is triggered this will display “Voltage Brownout”.

18.1.4 Operation Tab

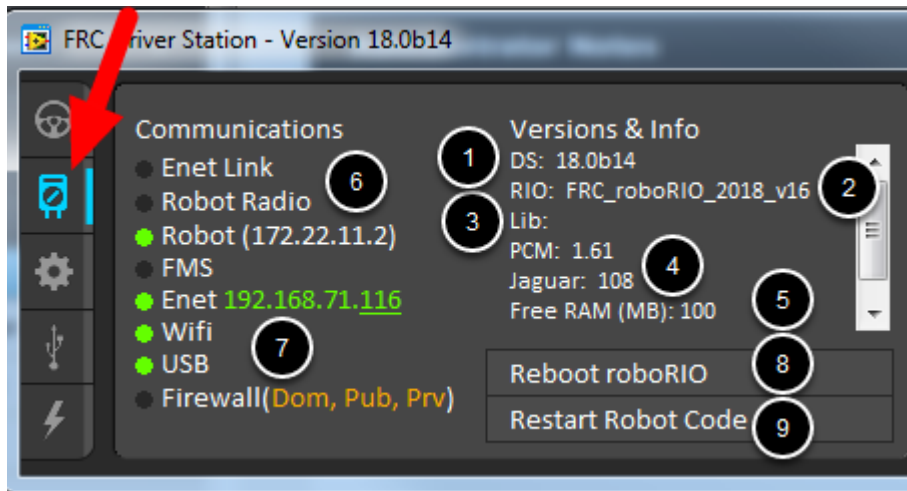


The Operations Tab is used to control the mode of the robot and provide additional key status indicators while the robot is running.

1. Robot Mode - This section controls the Robot Mode. Practice Mode causes the robot to cycle through the same transitions as an FRC match after the Enable button is pressed (timing for practice mode can be found on the setup tab).
2. Enable/Disable - These controls enable and disable the robot. See also [Driver Station Key Shortcuts](#)
3. Elapsed Time - Indicates the amount of time the robot has been enabled
4. PC Battery - Indicates current state of DS PC battery and whether the PC is plugged in
5. PC CPU% - Indicates the CPU Utilization of the DS PC
6. Window Mode - When not on the Driver account on the Classmate allows the user to toggle between floating (arrow) and docked (rectangle)
7. Team Station - When not connected to FMS, sets the team station to transmit to the robot.

Note: When connected to the Field Management System the controls in sections 1, and 2 will be replaced by the words FMS Connected and the control in Section 7 will be greyed out.

18.1.5 Diagnostics Tab

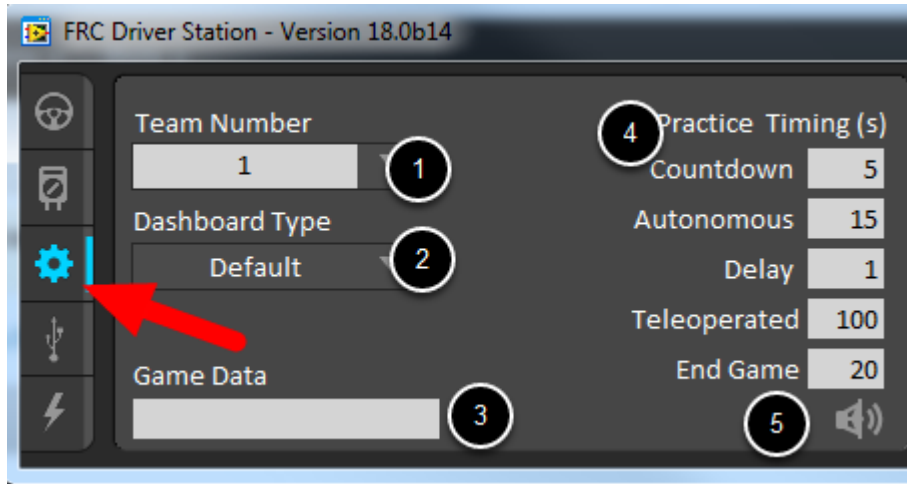


The Diagnostics Tab contains additional status indicators that teams can use to diagnose issues with their robot:

1. DS Version - Indicates the Driver Station Version number
2. roboRIO Image Version - String indicating the version of the roboRIO Image
3. WPILib Version - String indicating the version of WPILib in use
4. CAN Device Versions - String indicating the firmware version of devices connected to the CAN bus. These items may not be present if the CTRE Phoenix Framework has not been loaded
5. Memory Stats - This section shows stats about the roboRIO memory
6. Connection Indicators - The top half of these indicators show connection status to various components.
 - “Enet Link” indicates the computer has something connected to the ethernet port.
 - “Robot Radio” indicates the ping status to the robot wireless bridge at 10.XX.YY.1.
 - “Robot” indicates the ping status to the roboRIO using mDNS (with a fallback of a static 10.TE.AM.2 address).
 - “FMS” indicates if the DS is receiving packets from FMS (this is NOT a ping indicator).
7. Network Indicators - The second section of indicators indicates status of network adapters and firewalls. These are provided for informational purposes, communication may be established with one or more unlit indicators in this section
 - “Enet” indicates the IP address of the detected Ethernet adapter
 - “WiFi” indicates if a wireless adapter has been detected as enabled
 - “USB” indicates if a roboRIO USB connection has been detected
 - “Firewall” indicates if any firewalls are detected as enabled. Enabled firewalls will show in orange (Dom = Domain, Pub = Public, Prv = Private)
8. Reboot roboRIO - This button attempts to perform a remote reboot of the roboRIO (after clicking through a confirmation dialog)

9. Restart Robot Code - This button attempts to restart the code running on the robot (but not restart the OS)

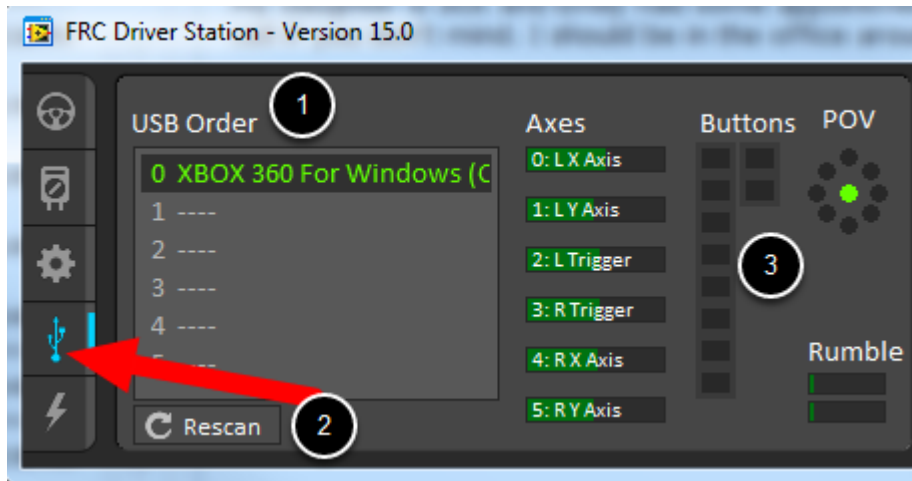
18.1.6 Setup Tab



The Setup Tab contains a number of buttons teams can use to control the operation of the Driver Station:

1. Team Number - Should contain your FRC Team Number. This controls the mDNS name that the DS expects the robot to be at. Shift clicking on the dropdown arrow will show all roboRIO names detected on the network for troubleshooting purposes.
2. Dashboard Type - Controls what Dashboard is launched by the Driver Station. *Default* launches the file pointed to by the "FRC DS Data Storage.ini" (for more information about setting a *custom dashboard*), by default this is Dashboard.exe in the Program Files (x86)\FRC Dashboard folder. *LabVIEW* attempts to launch a dashboard at the default location for a custom built LabVIEW dashboard, but will fall back to the default if no dashboard is found. *SmartDashboard* and *Shuffleboard* launch the respective dashboards included with the C++ and Java WPILib installation. *Remote* launches another dashboard from the "DashboardRemoteIP" field in the "FRC DS Data Storage.ini" file.
3. Game Data - This box can be used for at home testing of the Game Data API. Text entered into this box will appear in the Game Data API on the Robot Side. When connected to FMS, this data will be populated by the field automatically.
4. Practice Mode Timing - These boxes control the timing of each portion of the practice mode sequence. When the robot is enabled in practice mode the DS automatically proceeds through the modes indicated from top to bottom.
5. Audio Control - This button controls whether audio tones are sounded when the Practice Mode is used.

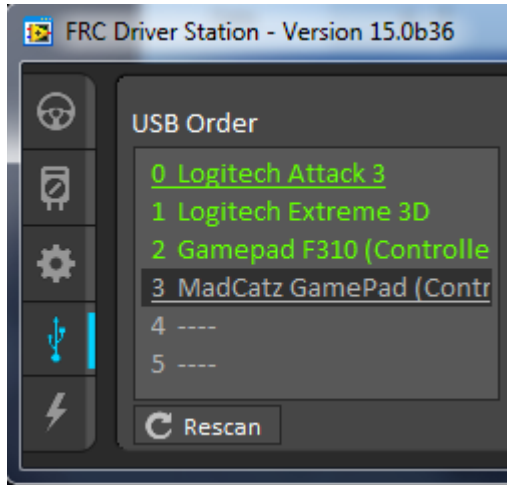
18.1.7 USB Devices Tab



The USB Devices tab includes the information about the USB Devices connected to the DS

1. USB Setup List - This contains a list of all compatible USB devices connected to the DS. Pressing a button on a device will highlight the name in green and put 2 *s before the device name
2. Rescan - This button will force a Rescan of the USB devices. While the robot is disabled, the DS will automatically scan for new devices and add them to the list. To force a complete re-scan or to re-scan while the robot is Enabled (such as when connected to FMS during a match) press F1 or use this button.
3. Device indicators - These indicators show the current status of the Axes, buttons and POV of the joystick.
4. Rumble - For XInput devices (such as X-Box controllers) the Rumble control will appear. This can be used to test the rumble functionality of the device. The top bar is "Right Rumble" and the bottom bar is "Left Rumble". Clicking and holding anywhere along the bar will activate the rumble proportionally (left is no rumble = 0, right is full rumble = 1). This is a control only and will not indicate the Rumble value set in robot code.

Re-Arranging and Locking Devices



The Driver Station has the capability of “locking” a USB device into a specific slot. This is done automatically if the device is dragged to a new position and can also be triggered by double clicking on the device. “Locked” devices will show up with an underline under the device. A locked device will reserve its slot even when the device is not connected to the computer (shown as grayed out and underlined). Devices can be unlocked (and unconnected devices removed) by double clicking on the entry.

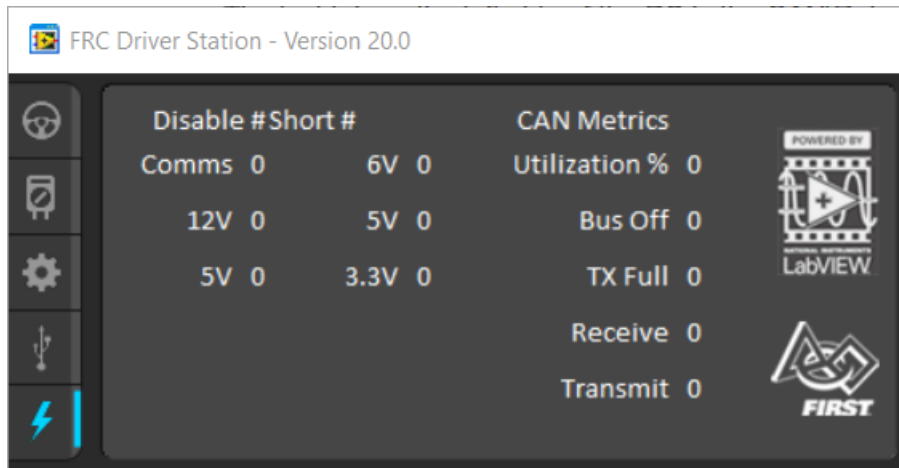
Note: If you have two or more of the same device, they should maintain their position as long as all devices remain plugged into the computer in the same ports they were locked in. If you switch the ports of two identical devices the lock should follow the port, not the device. If you re-arrange the ports (take one device and plug it into a new port instead of swapping) the behavior is not determinate (the devices may swap slots). If you unplug one or more of the set of devices, the positions of the others may move, they should return to the proper locked slots when all devices are reconnected.

Example: The image above shows 4 devices:

- A Locked “Logitech Attack 3” joystick. This device will stay in this position unless dragged somewhere else or unlocked
- An unlocked “Logitech Extreme 3D” joystick
- An unlocked “Gamepad F310 (Controller)” which is a Logitech F310 gamepad
- A Locked, but disconnected “MadCatz GamePad (Controller)” which is a MadCatz Xbox 360 Controller

In this example, unplugging the Logitech Extreme 3D joystick will result in the F310 Gamepad moving up to slot 1. Plugging in the MadCatz Gamepad (even if the devices in Slots 1 and 2 are removed and those slots are empty) will result in it occupying Slot 3.

18.1.8 CAN/Power Tab

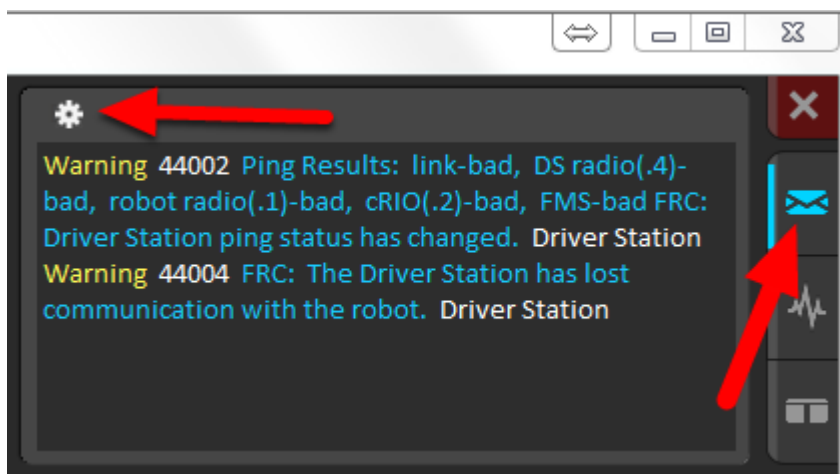


The last tab on the left side of the DS is the CAN/Robot Power Tab. This tab contains information about the power status of the roboRIO and the status of the CAN bus:

1. Comms Faults - Indicates the number of Comms faults that have occurred since the DS has been connected
2. 12V Faults - Indicates the number of input power faults (Brownouts) that have occurred since the DS has been connected
3. 6V/5V/3.3V Faults - Indicates the number of faults (typically cause by short circuits) that have occurred on the User Voltage Rails since the DS has been connected
4. CAN Bus Utilization - Indicates the percentage utilization of the CAN bus
5. CAN faults - Indicates the counts of each of the 4 types of CAN faults since the DS has been connected

If a fault is detected, the indicator for this tab (shown in blue in the image above) will turn red.

18.1.9 Messages Tab



The Messages tab displays diagnostic messages from the DS, WPILib, User Code, and/or the roboRIO. The messages are filtered by severity. By default, only Errors are displayed.

To access settings for the Messages tab, click the Gear icon. This will display a menu that will allow you to select the detail level (Errors, Errors+Warnings or Errors+Warnings+Prints), Clear the box, launch a larger Console window for viewing messages, or launch the DS Log Viewer.

18.1.10 Charts Tab



The Charts tab plots and displays advanced indicators of robot status to help teams diagnose robot issues:

1. The top graph charts trip time in milliseconds in green (against the axis on the right) and lost packets per second in orange (against the axis on the left)
2. The bottom graph plots battery voltage in yellow (against the axis on the left), roboRIO CPU in red (against the axis on the right), DS Requested mode as a continuous line on the bottom of the chart and robot mode as a discontinuous line above it.
3. This key shows the colors used for the DS Requested and Robot Reported modes in the bottom chart.
4. Chart scale - These controls change the time scale of the DS Charts
5. This button launches the *DS Log File Viewer*

The DS Requested mode is the mode that the Driver Station is commanding the robot to be in. The Robot Reported mode is what code is actually running based on reporting methods contained in the coding frameworks for each language.

18.1.11 Both Tab

The last tab on the right side is the Both tab which displays Messages and Charts side by side

18.1.12 Driver Station Key Shortcuts

- *F1* - Force a Joystick refresh.
- *[+] + * - Enable the robot (the 3 keys above Enter on most keyboards)
- *Enter* - Disable the Robot
- *Space* - Emergency Stop the robot. After an emergency stop is triggered the roboRIO will need to be rebooted before the robot can be enabled again.

Note: Space bar will E-Stop the robot regardless of if the Driver Station window has focus or not

Warning: When connected to FMS in a match, teams must press the Team Station E-Stop button to emergency stop their robot as the DS enable/disable and E-Stop key shortcuts are ignored.

18.2 Driver Station Best Practices

This document was created by Steve Peterson, with contributions from Juan Chong, James Cole-Henry, Rick Kosbab, Greg McKaskle, Chris Picone, Chris Roadfeldt, Joe Ross, and Ryan Sjostrand. The original post and follow-up posts can be found [here](#)

Want to ensure the driver station isn't a stopper for your team at the FIRST Robotics Competition (FRC) field? Building and configuring a solid driver station laptop is an easy project for the time between stop build day and your competition. Read on to find lessons learned by many teams over thousands of matches.

18.2.1 Prior To Departing For The Competition

1. Dedicate a laptop to be used solely as a driver station. Many teams do. A dedicated machine allows you manage the configuration for one goal – being ready to compete at the field. Dedicated means no other software except the FRC-provided Driver Station software and associated Dashboard installed or running.
2. Use a business-class laptop for your driver station. Why? They're much more durable than the \$300 Black Friday special at Best Buy. They'll survive being banged around at the competition. Business-class laptops have higher quality device drivers, and the drivers are maintained for a longer period than consumer laptops. This makes your investment last longer. Lenovo ThinkPad T series and Dell Latitude are two popular business-class brands you'll commonly see at competitions. There are thousands for sale every day on eBay. The laptop provided in recent rookie kits is a good entry level machine. Teams often graduate from it to bigger displays as they do more with vision and dashboards.

3. Consider used laptops rather than new. The FRC® Driver Station and dashboard software uses very few system resources, so you don't need to buy a new laptop – instead, buy a cheap 4-5 year old used one. You might even get one donated by a used computer store in your area.
4. Laptop recommended features
 - a. RAM – 2GB of RAM is minimum, if you have a SSD.
 - b. A display size of 13” or greater, with minimum resolution of 1440x1050.
 - c. Ports
 - i. A built-in Ethernet port is highly preferred. Ensure that it's a full-sized port. The hinged Ethernet ports don't hold up to repeated use.
 - ii. Use an Ethernet port saver to make your Ethernet connection. This extends the life of the port on the laptop. This is particularly important if you have a consumer-grade laptop with a hinged Ethernet port.
 - iii. If the Ethernet port on your laptop is dodgy, either replace the laptop (recommended) or buy a USB Ethernet dongle from a reputable brand. Many teams find that USB Ethernet is less reliable than built-in Ethernet, primarily due to cheap hardware and bad drivers. The dongles given to rookies in the KOP have a reputation for working well.
 - iv. 2 USB ports minimum
- d. A keyboard. It's hard to quickly do troubleshooting on touch-only computers at the field.
- e. A solid-state disk (SSD). If the laptop has a rotating disk, spend \$50 and replace it with a SSD.
- f. Updated to the current release of Windows 10. Being the most common OS now seen at competitions, bugs are more likely to be found and fixed for Windows 10 than on older Windows versions.
5. Install all Windows updates a week before the competition. This allows you time to ensure the updates will not interfere with driver station functions. To do so, open the Windows Update settings page and see that you're up-to-date. Install pending updates if not. Reboot and check again to make sure you're up to date.
6. Change “Active Hours” for Windows Updates to prevent updates from installing during competition hours. Navigate to Start -> Settings -> Update & Security -> Windows Update, then select Change active hours. If you're traveling to a competition, take time zone differences into account. This will help ensure your driver station does not reboot or fail due to update installing on the field.
7. Remove any 3rd party antivirus or antimalware software. Instead, use Windows Defender on Windows 10. Since you're only connecting to the internet for Windows and FRC software updating, the risk is low. Only install software on your driver station that's needed for driving. Your goal here is to eliminate variables that might interfere with proper operation. Remove any unneeded preinstalled software (“bloatware”) that came with the machine. Don't use the laptop as your Steam machine for gaming back at the hotel the night before the event. Many teams go as far as having a separate programming laptop.
8. Avoid managed Windows 10 installations from the school's IT department. These deployments are built for the school environment and often come with unwanted software that interferes with your robot's operation.

9. Laptop battery / power
 - a. Turn off Put the computer to sleep in your power plan for both battery and powered operation.
 - b. Turn off USB Selective Suspend:
 - i. Right click on the battery/charging icon in the tray, then select Power Options.
 - ii. Edit the plan settings of your power plan.
 - iii. Click the Change advanced power settings link.
 - iv. Scroll down in the advanced settings and disable the USB selective suspend setting for both Battery and Plugged in.
 - c. Ensure the laptop battery can hold a charge for at least an hour after making the changes above. This allows plenty of time for the robot and drive team to go through the queue and reach the alliance station without mains power.
10. Bring a trusted USB and Ethernet cable for use connecting to the roboRIO.
11. Add retention/strain relief to prevent your joystick/gamepad controllers from falling on the floor and/or yanking on the USB ports. This helps prevent issues with intermittent controller connections.
12. The Windows user account you use to drive must be a member of the Administrator group.

18.2.2 At The Competition

1. Turn off Windows firewall using these instructions.
2. Turn off the Wi-Fi adapter, either using the dedicated hardware Wi-Fi switch or by disabling it in the Adapter Settings control panel.
3. Charge the driver station when it's in the pit.
4. Remove login passwords or ensure everyone on the drive team knows the password. You'd be surprised at how often drivers arrive at the field without knowing the password for the laptop.
5. Ensure your LabView code is deployed permanently and set to "run as startup", using the instructions in the LabView Tutorial. If you must deploy code every time you turn the robot on, you're doing it wrong.
6. Limit web browsing to FRC related web sites. This minimizes the chance of getting malware during the competition.
7. Don't plan on using internet access to do software updates. There likely won't be any in the venue, and hotel Wi-Fi varies widely in quality. If you do need updates, contact a Control System Advisor in the pit.

18.2.3 Before Each Match

1. Make sure the laptop is on and logged in prior to the end of the match before yours.
2. Close programs that aren't needed during the match - e.g., Visual Studio Code or Lab-View - when you are competing.
3. Bring your laptop charger to the field. Power is provided for you in each player station.
4. Fasten your laptop with hook-and-loop tape to the player station shelf. You never know when your alliance partner will have an autonomous programming issue and blast the wall.
5. Ensure joysticks and controllers are assigned to the correct USB ports.
 - a. In the USB tab in the FRC Driver Station software, drag and drop to assign joysticks as needed.
 - b. Use the rescan button (F1) if joysticks / controllers do not appear green
 - c. Use the rescan button (F1) during competition if joystick or controllers become unplugged and then are plugged back in or otherwise turn gray during competition.

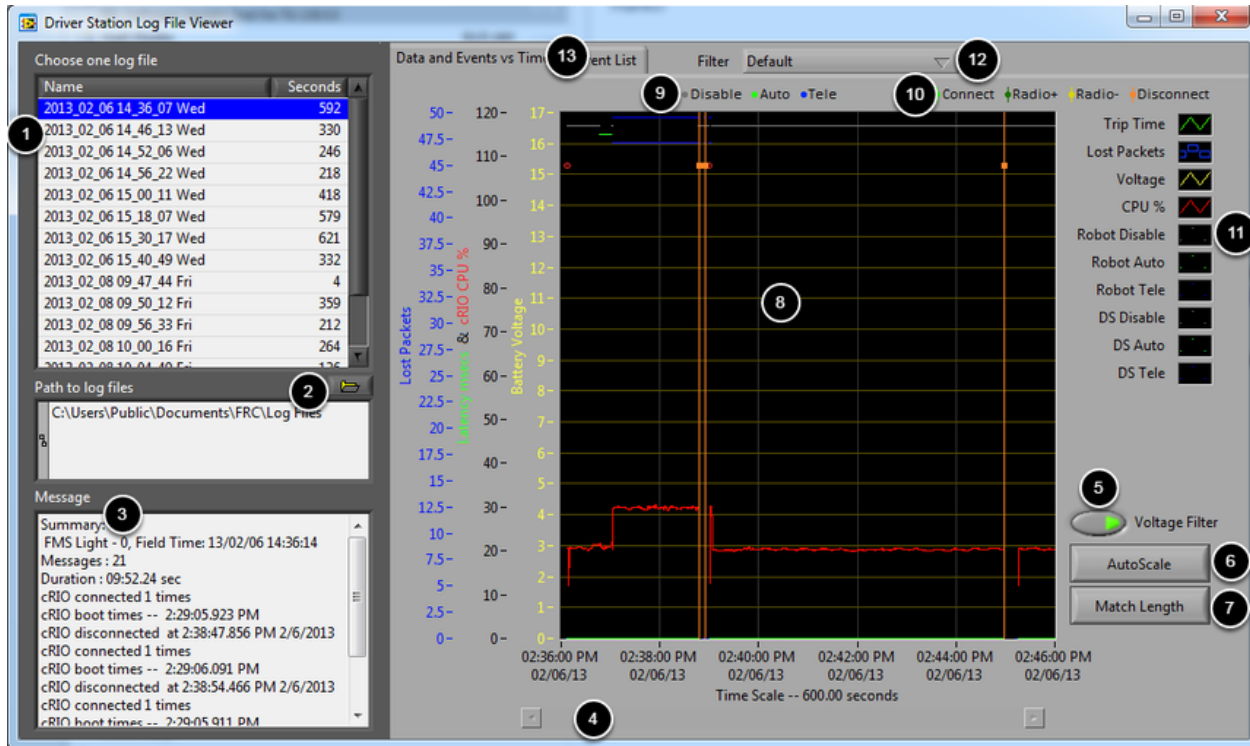
18.3 Driver Station Log File Viewer

In an effort to provide information to aid in debugging, the FRC® Driver Station creates log files of important diagnostic data while running. These logs can be reviewed later using the FRC Driver Station Log Viewer. The Log Viewer can be found via the shortcut installed in the Start menu or in the FRC Driver Station folder in Program Files.

18.3.1 Event Logs

The Driver Station logs all messages sent to the Messages box on the Diagnostics tab (not the User Messages box on the Operation tab) into a new Event Log file. When viewing Log Files with the Driver Station Log File Viewer, the Event Log and DSLog files are overlaid in a single display.

18.3.2 Log Viewer UI



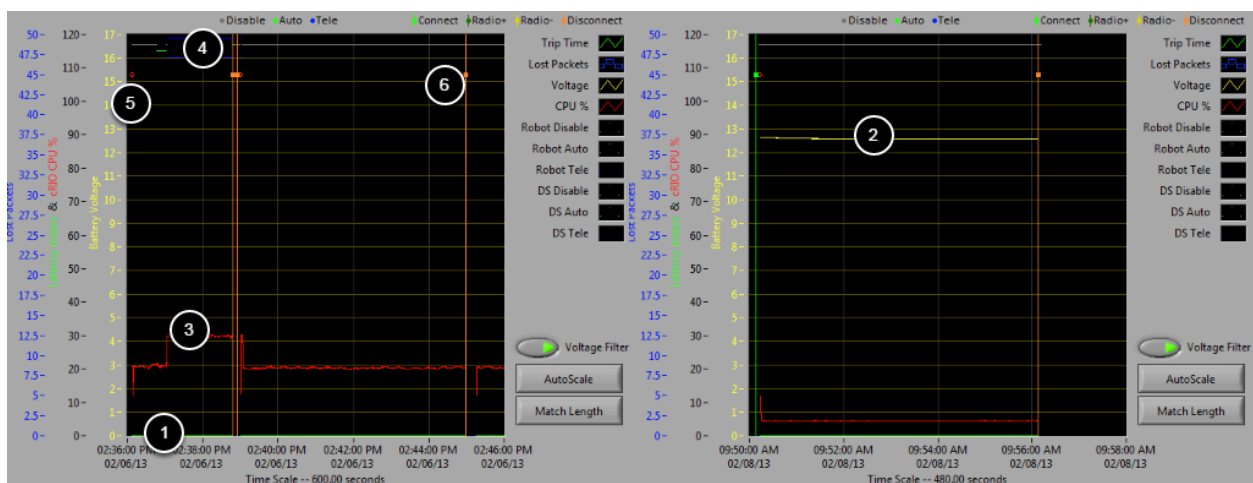
The Log Viewer contains a number of controls and displays to aid in the analysis of the Driver Station log files:

1. File Selection Box - This window displays all available log files in the currently selected folder. Click on a log file in the list to select it.
2. Path to Log Files - This box displays the current folder the viewer is looking in for log files. This defaults to the folder that the Driver Station stores log files in. Click the folder icon to browse to a different location.
3. Message Box - This box displays a summary of all messages from the Event Log. When hovering over an event on the graph this box changes to display the information for that event.
4. Scroll Bar - When the graph is zoomed in, this scroll bar allows for horizontal scrolling of the graph.
5. Voltage Filter - This control turns the Voltage Filter on and off (defaults to on). The Voltage Filter filters out data such as CPU %, robot mode and trip time when no Battery Voltage is received (indicating that the DS is no in communication with the roboRIO).
6. AutoScale - This button zooms the graph out to show all data in the log.
7. Match Length - This button scales the graph to approximately the length of an FRC match (2 minutes and 30 seconds shown). It does not automatically locate the start of the match, you will have to scroll using the scroll bar to locate the beginning of the Autonomous mode.
8. Graph - This display shows graph data from the DS Log file (voltage, trip time, roboRIO CPU%, Lost Packets, and robot mode) as well as overlaid event data (shown as dots on the graph with select events showing as vertical lines across the entire graph). Hovering

over event markers on the graph displays information about the event in the Messages window in the bottom left of the screen.

9. Robot Mode Key - Key for the Robot Mode displayed at the top of the screen
10. Major event key - Key for the major events, displayed as vertical lines on the graph
11. Graph key - Key for the graph data
12. Filter Control - Drop-down to select the filter mode (filter modes explained below)
13. Tab Control - Control to switch between the Graph (Data and Events vs. Time) and Event List displays.

18.3.3 Using the Graph Display



The Graph Display contains the following information:

1. Graphs of Trip Time in ms (green line) and Lost Packets per second (displayed as blue vertical bars). In these example images Trip Time is a flat green line at the bottom of the graph and there are no lost packets
2. Graph of Battery voltage displayed as a yellow line.
3. Graph of roboRIO CPU % as a red line
4. Graph of robot mode and DS mode. The top set of the display shows the mode commanded by the Driver Station. The bottom set shows the mode reported by the robot code. In this example the robot is not reporting it's mode during the disabled and autonomous modes, but is reported during Teleop.
5. Event markers will be displayed on the graph indicating the time the event occurred. Errors will display in red; warnings will display in yellow. Hovering over an event marker will display information about the event in the Messages box at the bottom left of the screen.
6. Major events are shown as vertical lines across the graph display.

To zoom in on a portion of the graph, click and drag around the desired viewing area. You can only zoom the time axis, you cannot zoom vertically.

18.3.4 Event List

DS Time	Event Message Text
2:36:07.288 PM	WARNING <Code> 44007 occurred at FRC_NetworkCommunications <secondsSinceReboot> 421.365 Warning <Code> 44001 occurred at No Change to Network Configuration: "Local Area Connection" <noNIC> FRC: Time since robot boot. Driver Station <time> 2/6/2013 2:36:07 PM<unique#> 3 ERROR <Code> -44009 occurred at Driver Station <time> 2/6/2013 2:36:06 PM<unique#> 2 FRC: A joystick was disconnected while the robot was enabled. Warning <Code> 44006 occurred at Driver Station <time> 2/6/2013 2:36:06 PM<unique#> 1 FRC: Custom I/O is not enabled or is not connected to the driver station.
2:36:07.328 PM	FMS Connected: FMS Light - 0, Field Time: 13/02/06 14:36:14
2:36:10.441 PM	WARNING <Code> 44008 occurred at FRC_NetworkCommunications <radioLostEvents> 173.563 <radioSec> FRC: Robot radio detection times.
2:37:01.461 PM	Watchdog Expiration: System 1, User 0
2:38:47.856 PM	Warning <Code> 44004 occurred at Driver Station <time> 2/6/2013 2:38:47 PM<unique#> 4 FRC: The Driver Station has lost communication with the robot.
2:38:49.356 PM	Warning <Code> 44002 occurred at Ping Results: link-GOOD, DS radio(4)-GOOD, robot radio(1)-GOOD, <time> 2/6/2013 2:38:49 PM<unique#> 5 FRC: Driver Station ping status has changed.
2:38:53.460 PM	WARNING <Code> 44007 occurred at FRC_NetworkCommunications <secondsSinceReboot> 587.369 FRC: Time since robot boot.
2:38:54.466 PM	Warning <Code> 44004 occurred at Driver Station <time> 2/6/2013 2:38:53 PM<unique#> 6 FRC: The Driver Station has lost communication with the robot.
2:38:55.468 PM	Warning <Code> 44002 occurred at Ping Results: link-GOOD, DS radio(4)-GOOD, robot radio(1)-GOOD, <time> 2/6/2013 2:38:55 PM<unique#> 7 FRC: Driver Station ping status has changed.
2:38:59.278 PM	WARNING <Code> 44008 occurred at FRC_NetworkCommunications <radioLostEvents> 339.065 <radioSec> FRC: Robot radio detection times. WARNING <Code> 44007 occurred at FRC_NetworkCommunications <secondsSinceReboot> 593.367

The Event List tab displays a list of events (warnings and errors) recorded by the Driver Station. The events and detail displayed are determined by the currently active filter (images shows "All Events, All Info" filter active).

18.3.5 Filters

Three filters are currently available in the Log Viewer:

1. Default: This filter filters out many of the errors and warnings produced by the Driver Station. This filter is useful for identifying errors thrown by the code on the Robot.
2. All Events and Time: This filter shows all events and the time they occurred
3. All Events, All Info: This filter shows all events and all recorded info. At this time the primary difference between this filter and "All Events and Time" is that this option shows the "unique" designator for the first occurrence of a particular message.

18.3.6 Identifying Logs from Matches

3:19:30.893 PM | FMS Connected: Practice - 1, Field Time: 13/02/06 15:19:37

A common task when working with the Driver Station Logs is to identify which logs came from competition matches. Logs which were taken during a match can now be identified using the FMS Connected event which will display the match type (Practice, Qualification or Elimination), match number, and the current time according to the FMS server. In this example, you can see that the FMS server time and the time of the Driver Station computer are fairly close, approximately 7 seconds apart.

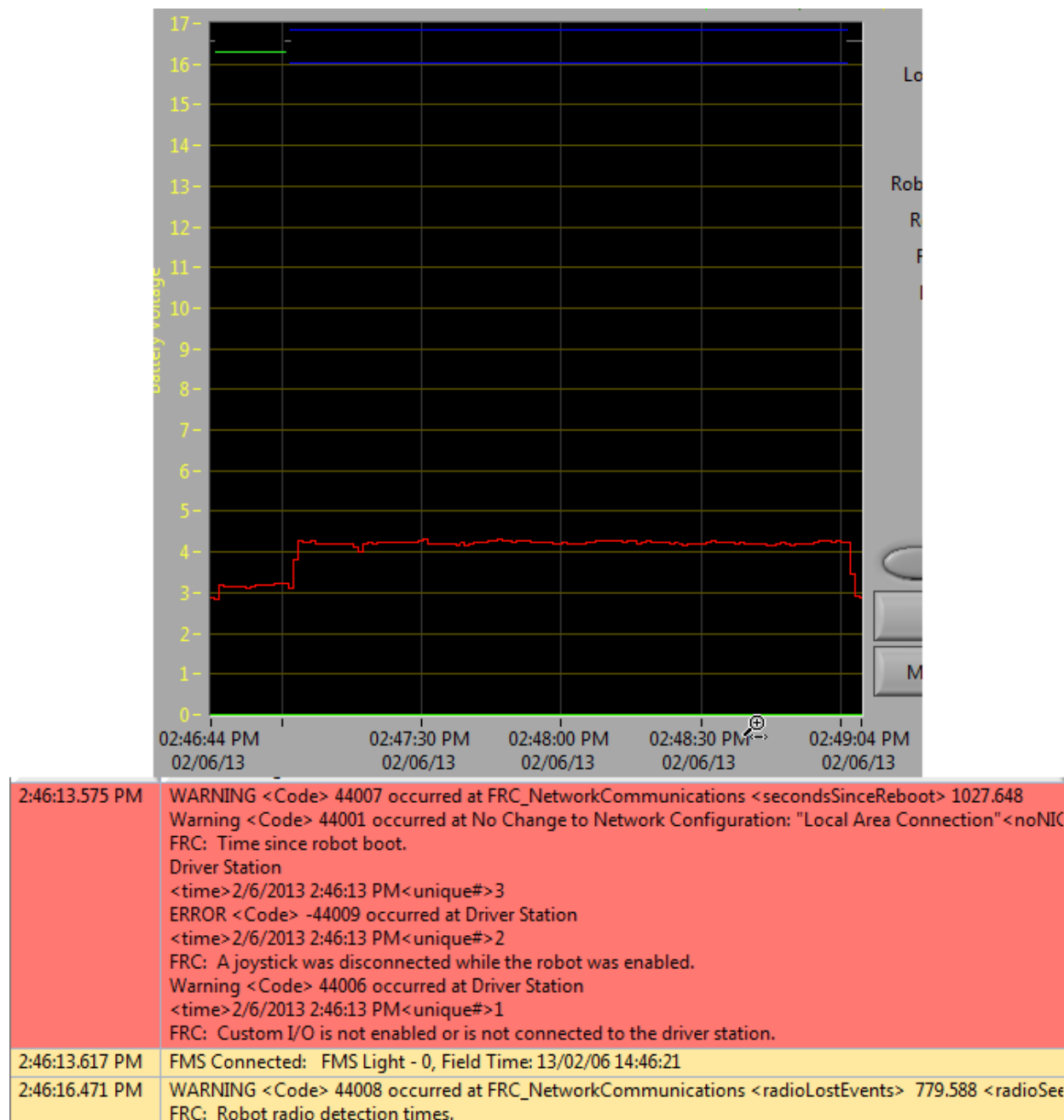
18.3.7 Identifying Common Connection Failures with the Log Viewer

When diagnosing robot issues, there is no substitute for thorough knowledge of the system and a methodical debugging approach. If you need assistance diagnosing a connection problem at your events it is strongly recommended to seek assistance from your FTA and/or CSA. The goal of this section is to familiarize teams with how some common failures can manifest themselves in the DS Log files. Please note that depending on a variety of conditions a particular failure show slightly differently in a log file.

Note: Note that all log files shown in this section have been scaled to match length using the Match Length button and then scrolling to the beginning of the autonomous mode. Also, many of the logs do not contain battery voltage information, the platform used for log capture was not properly wired for reporting the battery voltage.

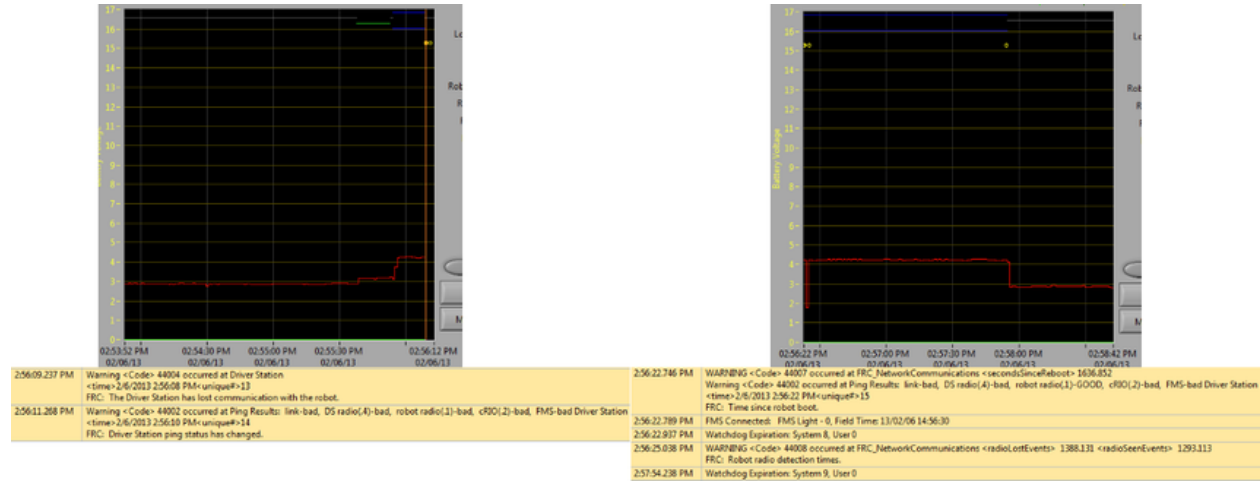
Tip: Some error messages that are found in the Log Viewer are show below and more are detailed in the [Driver Station Errors/Warnings](#) article.

“Normal” Log



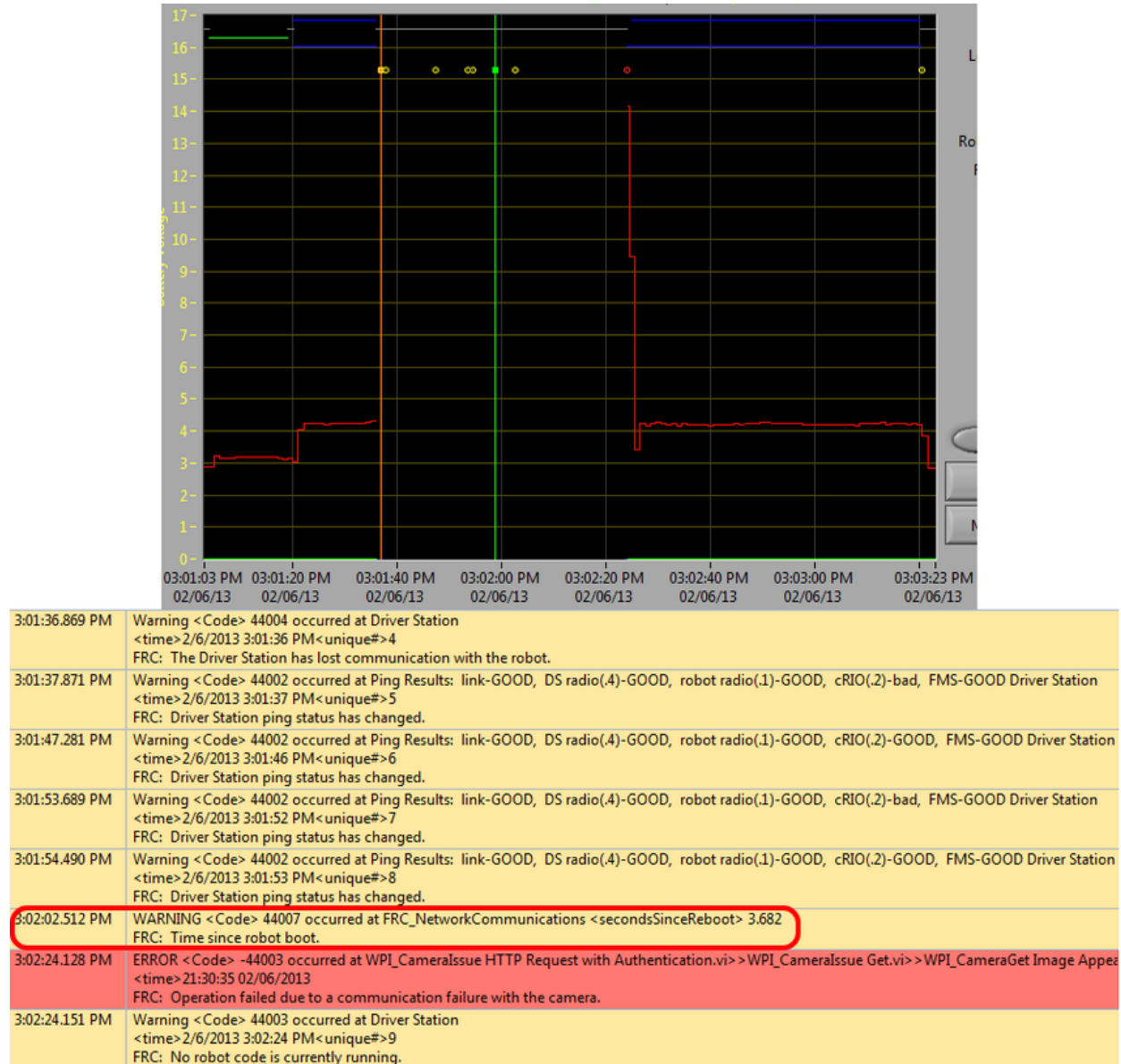
This is an example of a normal match log. The errors and warnings contained in the first box are from when the DS first started and can be ignored. This is confirmed by observing that these events occurred prior to the “FMS Connected:” event. The last event shown can also be ignored, it is also from the robot first connecting to the DS (it occurs 3 seconds after connecting to FMS) and occurs roughly 30 seconds before the match started.

Disconnected from FMS



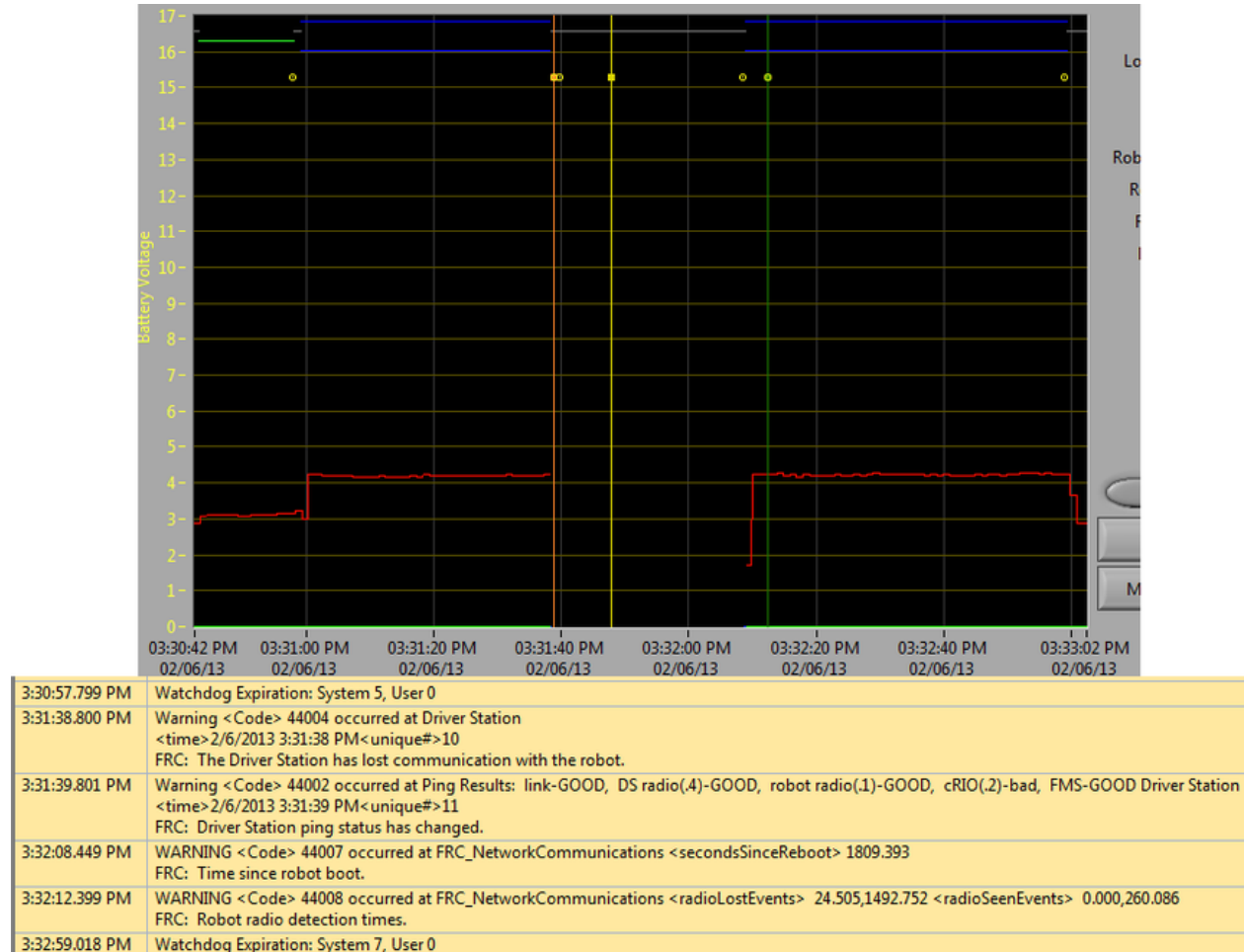
When the DS disconnects from FMS, and therefore the robot, during the match it may segment the log into pieces. The key indicators to this failure are the last event of the first log, indicating that the connection to FMS is now “bad” and the second event from the 2nd log which is a new FMS connected message followed by the DS immediately transitioning into Teleop Enabled. The most common cause of this type of failure is an ethernet cable with no latching tab or a damaged ethernet port on the DS computer.

roboRIO Reboot



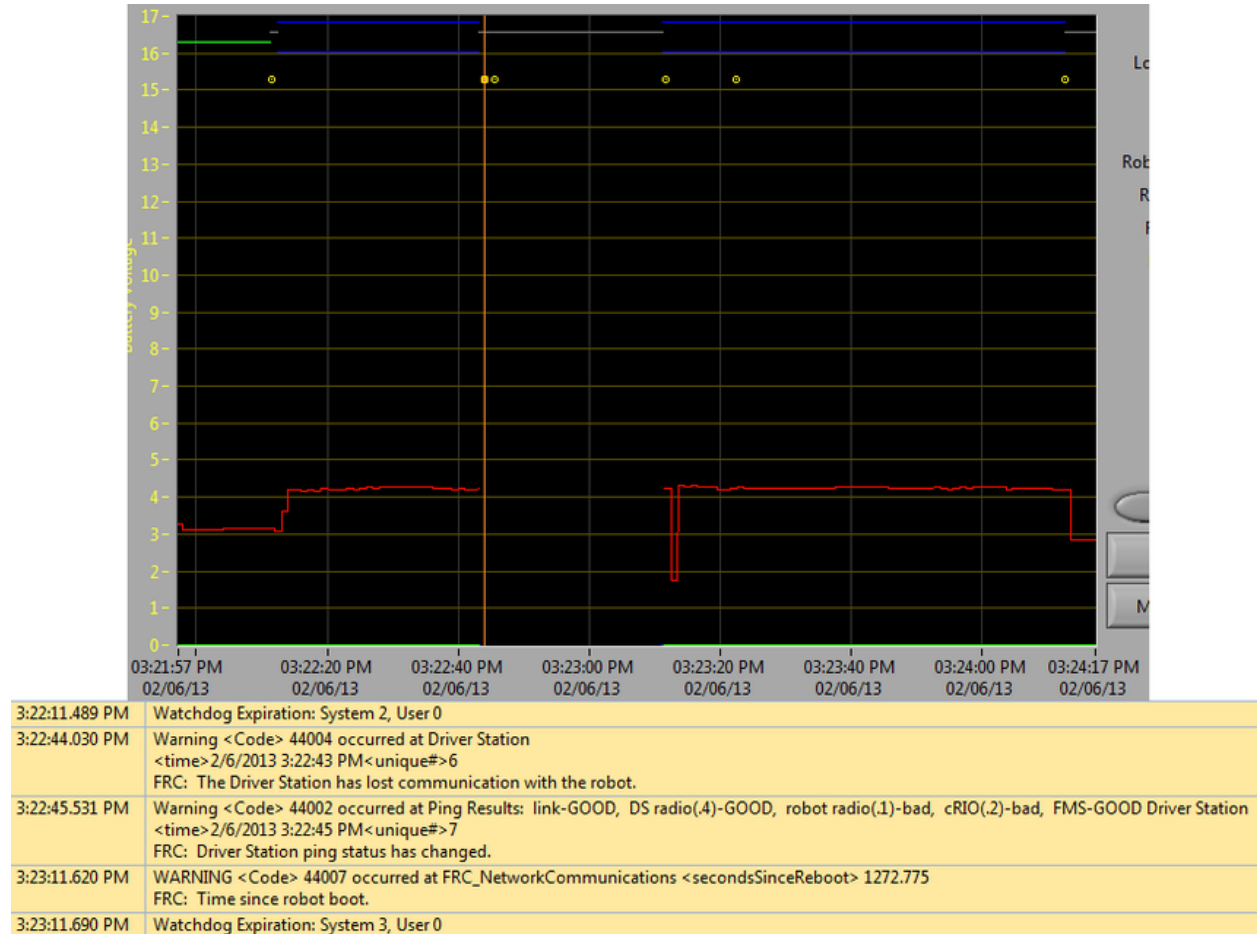
The “Time since robot boot” message is the primary indicator in a connection failure caused by the roboRIO rebooting. In this log the DS loses connection with the roboRIO at 3:01:36 as indicated by the first event. The second event indicates that the ping initiated after the connection failed was successful to all devices other than the roboRIO. At 3:01:47 the roboRIO begins responding to pings again, one additional ping fails at 3:01:52. At 3:02:02 the Driver Station connects to the roboRIO and the roboRIO reports that it has been up for 3.682 seconds. This is a clear indicator that the roboRIO has rebooted. The code continues to load and at 3:02:24 the code reports an error communicating with the camera. A warning is also reported indicating that no robot code is running right before the code finishes starting up.

Ethernet cable issue on robot



An issue with the ethernet cable on the robot is primarily indicated by the ping to the roboRIO going to bad and Radio Lost and Radio Seen events when the roboRIO reconnects. The “Time since robot boot” message when the roboRIO reconnects will also indicate that the roboRIO has not rebooted. In this example, the robot Ethernet cable was disconnected at 3:31:38. The ping status indicates that the radio is still connected. When the robot reconnects at 3:32:08 the “Time since robot boot” is 1809 seconds indicating that the roboRIO clearly did not reboot. At 3:32:12 the robot indicates that it lost the radio 24.505 seconds ago and it returned 0.000 seconds ago. These points are plotted as vertical lines on the graph, yellow for radio lost and green for radio seen. Note that the times are slightly offset from the actual events as shown via the disconnection and connection, but help to provide additional information about what is occurring.

Radio reboot



A reboot of the robot radio is typically characterized by a loss of connection to the radio for ~40-45 seconds. In this example, the radio briefly lost power at 3:22:44, causing it to start rebooting. The event at 3:22:45 indicates that the ping to the radio failed. At 3:23:11, the DS regains communication with the roboRIO and the roboRIO indicates it has been up for 1272.775 seconds, ruling out a roboRIO reboot. Note that the network switch on the radio comes back up very quickly so a momentary power loss may not result in a “radio lost”/“radio seen” event pair. A longer disturbance may result in radio events being logged by the DS. In that case, the distinguishing factor which points towards a radio reboot is the ping status of the radio from the DS. If the radio resets, the radio will be unreachable. If the issue is a cabling or connection issue on the robot, the radio ping should remain “GOOD”.

18.4 Driver Station Errors/Warnings

In an effort to provide both Teams and Volunteers (FTAs/CSAs/etc.) more information to use when diagnosing robot problems, a number of Warning and Error messages have been added to the Driver Station. These messages are displayed in the DS diagnostics tab when they occur and are also included in the DS Log Files that can be viewed with the Log File Viewer. This document discusses the messages produced by the DS (messages produced by WPILib can also appear in this box and the DS Logs).

18.4.1 Joystick Unplugged

```
ERROR<Code>-44009 occurred at Driver Station
<time>2/5/2013 4:43:54 PM <unique#>1
FRC: A joystick was disconnected while the robot was enabled.
```

This error is triggered when a Joystick is unplugged. Contrary to the message text this error will be printed even if the robot is not enabled, or even connected to the DS. You will see a single instance of this message occur each time the Driver Station is started, even if Joysticks are properly connected and functioning.

18.4.2 Lost Communication

```
Warning<Code>44004 occurred at Driver Station
<time>2/6/2013 11:07:53 AM<unique#>2
FRC: The Driver Station has lost communication with the robot.
```

This Warning message is printed whenever the Driver Station loses communication with the robot (Communications indicator changing from green to red). A single instance of this message is printed when the DS starts up, before communication is established.

18.4.3 Ping Status

```
Warning<Code>44002 occurred at Ping Results: link-GOOD, DS radio(.4)-bad, robot_
↪radio(.1)-GOOD, cRIO(.2)-bad, FMS- bad Driver Station
<time>2/6/2013 11:07:59 AM<unique#>5
FRC: Driver Station ping status has changed.
```

A Ping Status warning is generated each time the Ping Status to a device changes while the DS is not in communication with the roboRIO. As communications is being established when the DS starts up, a few of these warnings will appear as the Ethernet link comes up, then the connection to the robot radio, then the roboRIO (with FMS mixed in if applicable). If communications are later lost, the ping status change may help identify at which component the communication chain broke.

18.4.4 Time Since Robot Boot

```
WARNING<Code>44007 occurred at FRC_NetworkCommunications
**<secondsSinceReboot> 3.585**
FRC: Time since robot boot.
```

This message is printed each time the DS begins communicating with the roboRIO. The message indicates the up-time, in seconds, of the roboRIO and can be used to determine if a loss of communication was due to a roboRIO Reboot.

18.4.5 Radio Detection Times

```
WARNING<Code>44008 occurred at FRC_NetworkCommunications
<radioLostEvents> 19.004<radioSeenEvents> 0.000
FRC: Robot radio detection times

WARNING<Code>44008 occurred at FRC_NetworkCommunications
<radioLostEvents> 2.501,422.008<radioSeenEvents> 0.000,147.005
FRC: Robot radio detection times.
```

This message may be printed when the DS begins communicating with the roboRIO and indicates the time, in seconds, since the last time the radio was lost and seen. In the first example image above the message indicates that the roboRIO's connection to the radio was lost 19 seconds before the message was printed and the radio was seen again right when the message was printed. If multiple radioLost or radioSeen events have occurred since the roboRIO booted, up to 2 events of each type will be included, separated by commas.

18.4.6 No Robot Code

```
Warning<Code>44003 occurred at Driver Station
<time>2/8/2013 9:50:13 AM<unique#>8
FRC: No robot code is currently running.
```

This message is printed when the DS begins communicating with the roboRIO, but detects no robot code running. A single instance of this message will be printed if the Driver Station is open and running while the roboRIO is booting as the DS will begin communication with the roboRIO before the robot code finishes loading.

18.5 Programming Radios for FMS Offseason

When using the FMS Offseason software, the typical networking setup is to use a single access point with a single SSID and WPA key. This means that the radios should all be programmed to connect to this network, but with different IPs for each team. The Team version of the FRC® Bridge Configuration Utility has an FMS-Lite mode that can be used to do this configuration.

Before you begin using the software:

1. Disable WiFi connections on your computer, as it may prevent the configuration utility from properly communicating with the bridge

2. Make sure no devices are connected to your computer via ethernet, other than the wireless bridge.

18.5.1 Pre-Requisites

Note: Even though WPILib uses Java 11, the FRC Radio Configuration Utility requires Java 8.

The FRC Radio Configuration Utility requires the Java Runtime Engine (JRE). If you do not have Java installed, you can download the JRE from [here](#).

The FRC Radio Configuration Utility requires Administrator privileges to configure the network settings on your machine. The program should request the necessary privileges automatically (may require a password if run from a non-Administrator account), but if you are having trouble try running it from an Administrator account.

18.5.2 Application Notes

The Radio Kiosk will program the radio to enforce the 4 Mbps bandwidth limit on traffic exiting the radio over the wireless interface. In the home configuration (AP mode) this is a total, not a per client limit. This means that streaming video to multiple clients is not recommended.

The Kiosk has been tested on Windows 7, 8, and 10. It may work on other operating systems, but has not been tested.

Programmed Configuration

The Radio Configuration Utility programs a number of configuration settings into the radio when run. These settings apply to the radio in all modes (including at events). These include:

- Set a static IP of 10.TE.AM.1
- Set an alternate IP on the wired side of 192.168.1.1 for future programming
- Bridge the wired ports so they may be used interchangeably
- The LED configuration noted in the graphic above
- 4Mb/s bandwidth limit on the outbound side of the wireless interface
- QoS rules for internal packet prioritization (affects internal buffer and which packets to discard if bandwidth limit is reached). These rules are Robot Control and Status (UDP 1110, 1115, 1150) >> Robot TCP & *NetworkTables* (TCP 1735, 1740) >> Bulk (All other traffic).

Tip: See the *Status Light Reference* for details on the behavior of the radio status lights when configured.

When programmed with the team version of the Radio Configuration - Utility, the user accounts will be left at (or set to) the firmware - defaults **for the DAPs only**:

- Username: root

- Password: root

Note: It is not recommended to modify the configuration manually

18.5.3 Download the software

The screenshot shows the TeamForge web interface. The top navigation bar includes 'Projects', 'My Workspace', 'Admin', 'History', and 'More'. A search bar contains 'pkg1105' and a 'Jump to ID' button. The user 'koconnor' is logged in. The main navigation menu includes 'Project Home', 'Trackers', 'Source Code', 'File Releases', 'Documents', 'Wiki', 'Discussions', 'Reports', and 'Project Admin'. The breadcrumb trail is 'WPILib / File Releases / FRC Radio Configuration Utility / List Releases'. On the left, a sidebar shows 'File Release' with links for 'Summary', 'Packages', 'Driver Station', 'FRC Radio Configuration Utility' (selected), and 'Simulation'. The main content area has a 'Package Details' section with 'Package Name: FRC Radio Configuration Utility', 'Description: Tool to configure FRC Robot Radios.', and 'Show Download Link in Project List: No'. Below this is a 'Releases' section with a table. The table has columns: 'Release ID: Name', 'Maturity', 'Created On', 'Status', 'Files', 'Downloads', 'Related Tracker Artifacts', and 'Related Planning Folders'. The table is empty, showing 'No results found.' at the bottom. There are buttons for 'Download Selected', 'Monitor', 'Delete', 'Edit', and 'Add'.

Download the latest FRC Radio Configuration Utility Installer from the [WPILib project File Releases](#).

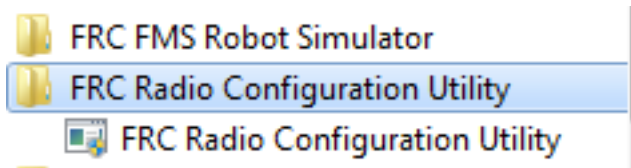
18.5.4 Install the software

FRC_Radio_Configuration_10_8_15.exe	10/8/2015 1:50 PM	Application
FRCicon_RGB_Border.bmp	2/20/2015 1:27 PM	Bitmap Image

Double click on FRC_Radio_Configuration_MM_DD_YY.exe to launch the installer. Follow the prompts to complete the installation.

Part of the installation prompts will include installing WinPCap if it is not already present. The WinPCap installer contains a checkbox (checked by default) to start the WinPCap driver on boot. You should leave this box checked.

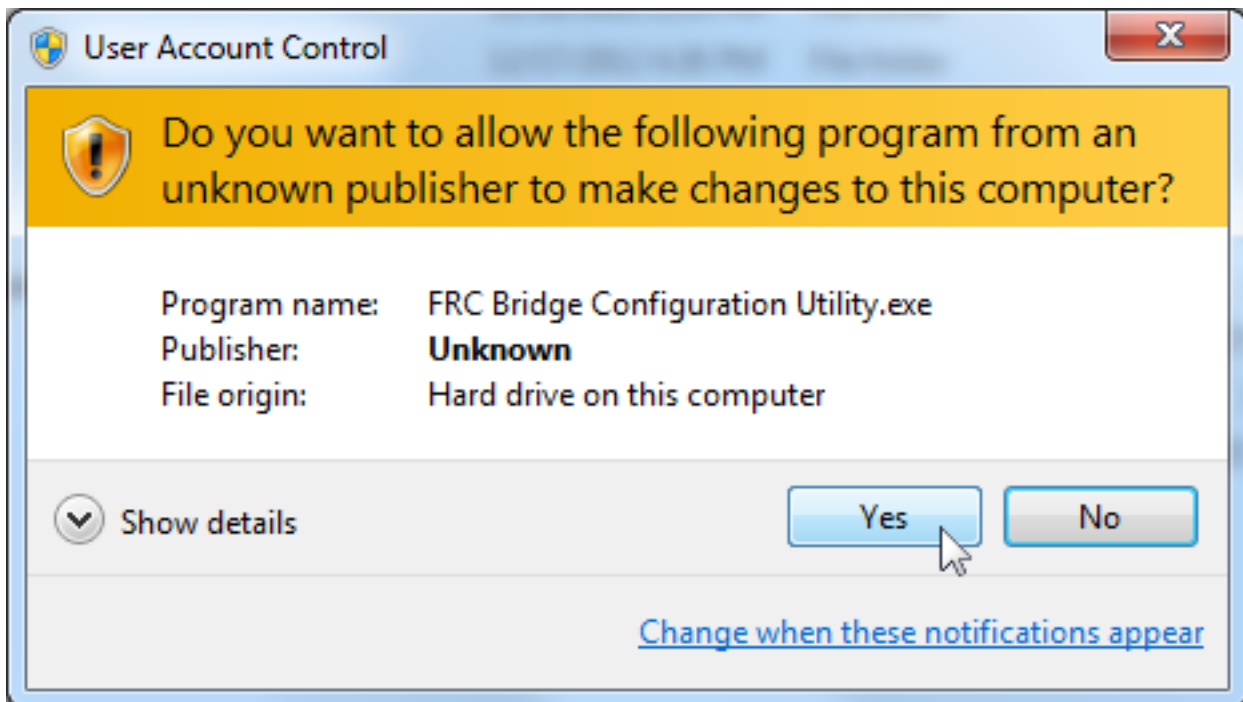
18.5.5 Launch the software



Use the Start menu or desktop shortcut to launch the program.

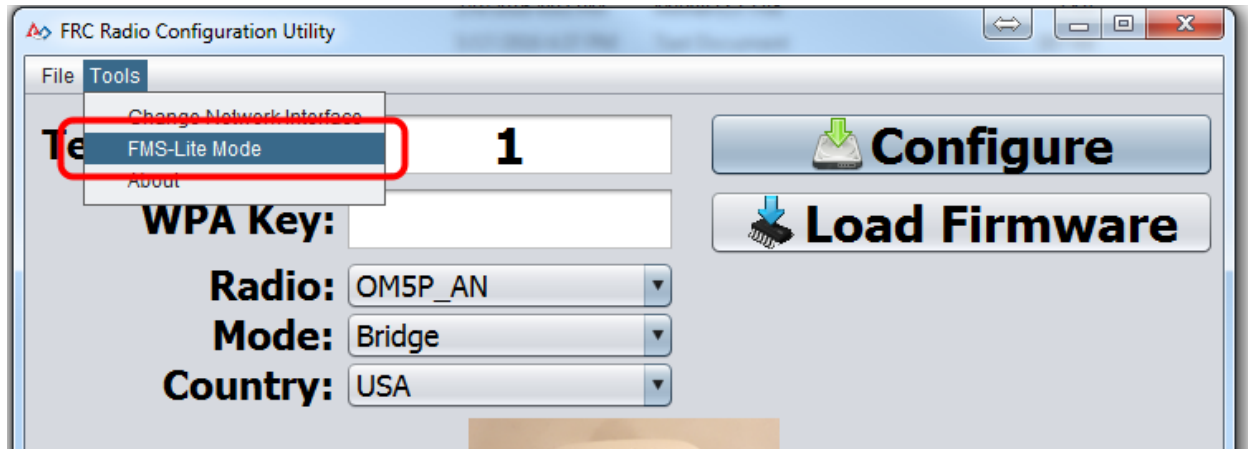
Note: If you need to locate the program it is installed to C:/Program Files (x86)/FRC Radio Configuration Utility. For 32-bit machines the path is C:/Program Files/FRC Radio Configuration Utility/

18.5.6 Allow the program to make changes, if prompted



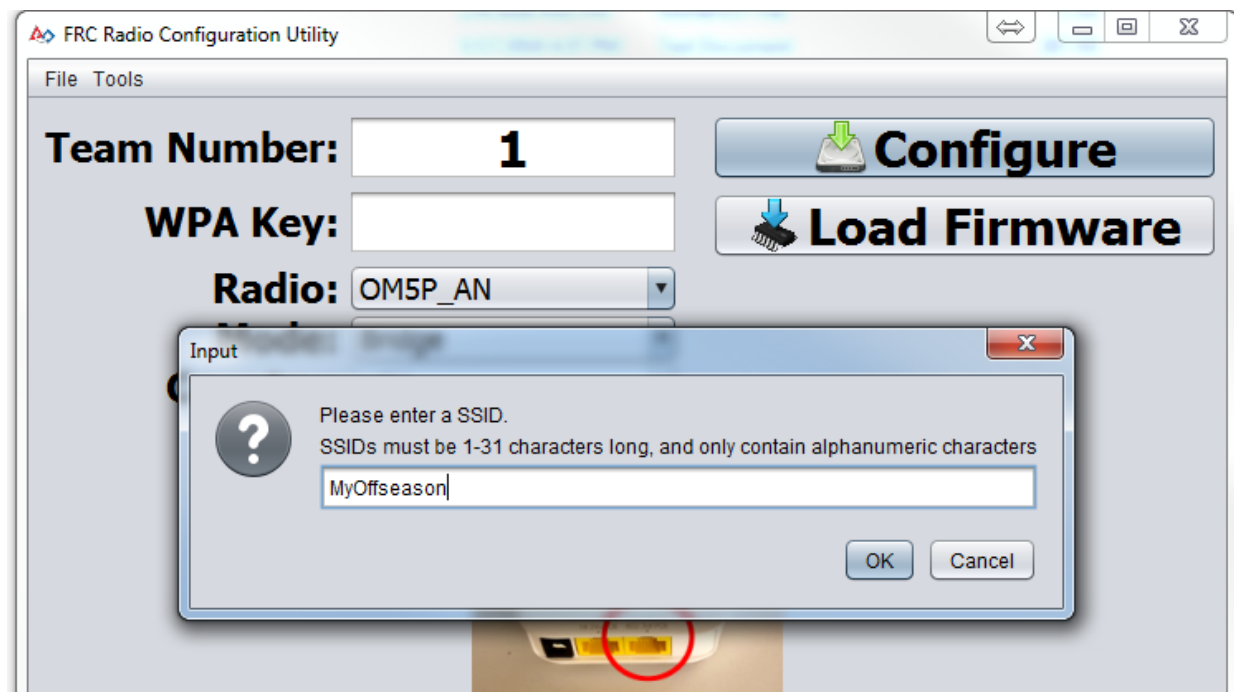
If the your computer is running Windows Vista or Windows 7, a prompt may appear about allowing the configuration utility to make changes to the computer. Click “Yes” if the prompt appears.

18.5.7 Enter FMS-Lite Mode



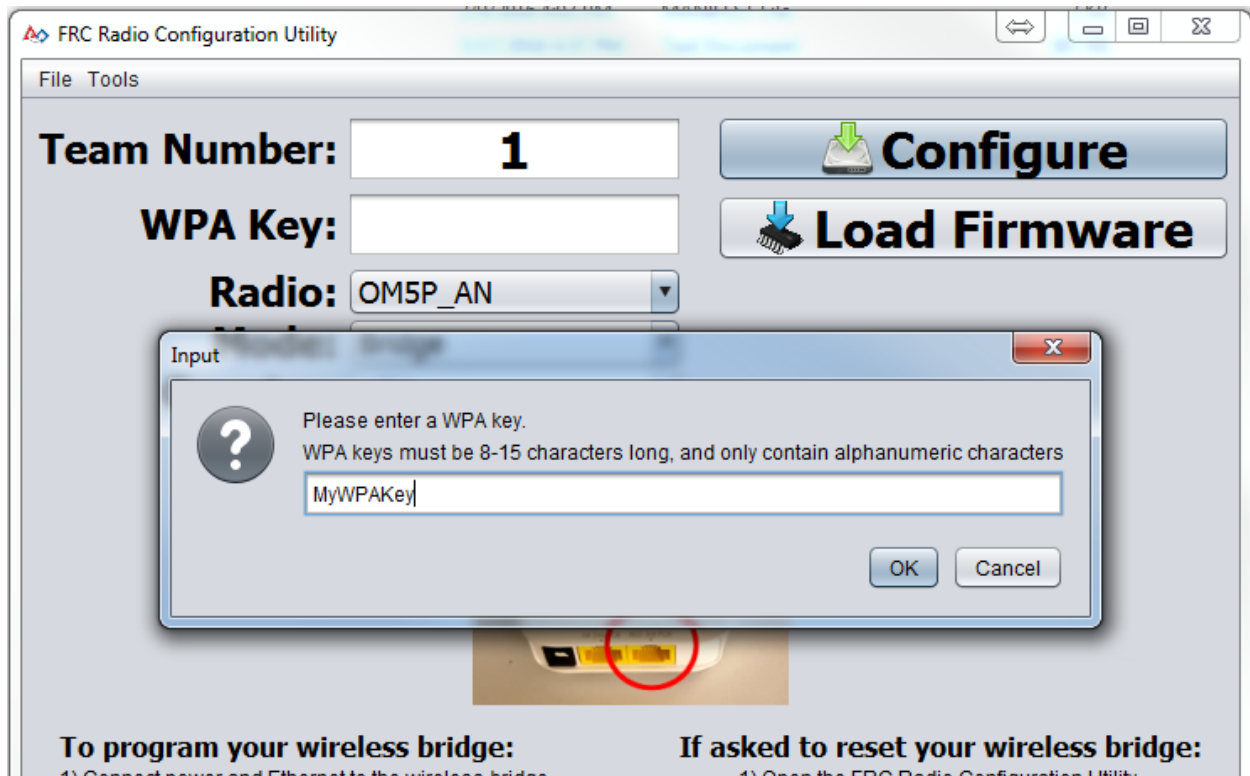
Click Tools -> FMS-Lite Mode to enter FMS-Lite Mode.

18.5.8 Enter SSID



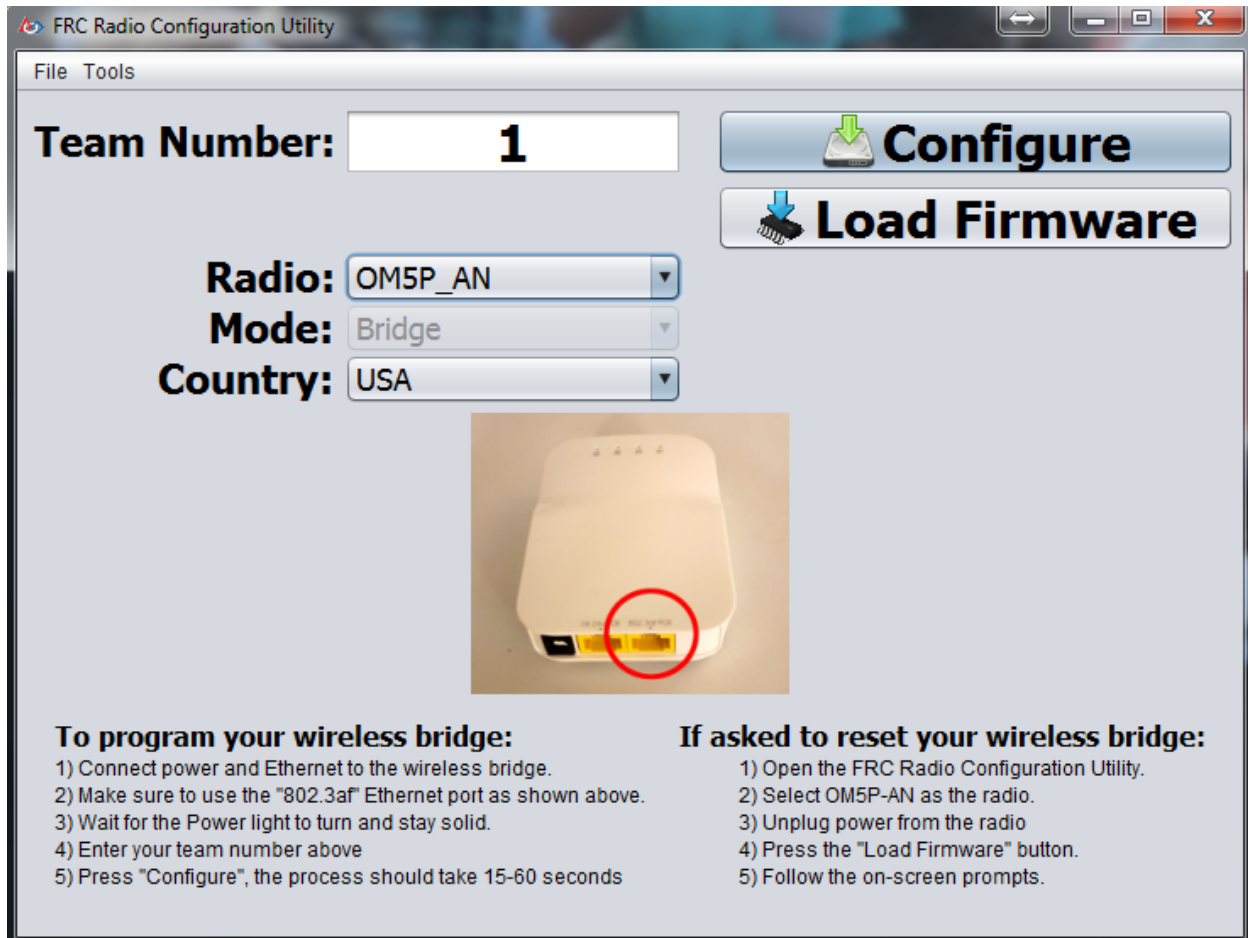
Enter the SSID (name) of your wireless network in the box and click OK.

18.5.9 Enter WPA Key



Enter the WPA key for your network in the box and click OK. Leave the box blank if you are using an unsecured network.

18.5.10 Program Radios



Team Number:


Radio:

Mode:

Country:

Configure

Load Firmware



To program your wireless bridge:

- 1) Connect power and Ethernet to the wireless bridge.
- 2) Make sure to use the "802.3af" Ethernet port as shown above.
- 3) Wait for the Power light to turn and stay solid.
- 4) Enter your team number above
- 5) Press "Configure", the process should take 15-60 seconds

If asked to reset your wireless bridge:

- 1) Open the FRC Radio Configuration Utility.
- 2) Select OM5P-AN as the radio.
- 3) Unplug power from the radio
- 4) Press the "Load Firmware" button.
- 5) Follow the on-screen prompts.

The Kiosk is now ready to program any number of radios to connect to the network entered. To program each radio, connect the radio to the Kiosk, set the Team Number in the box, and click Configure.

The kiosk will program OpenMesh, D-Link Rev A or D-Link Rev B radios to work on an offseason FMS network by selecting the appropriate option from the "Radio" dropdown.

Note: Bandwidth limitations and QoS will not be configured on the D-Link radios in this mode.

18.5.11 Changing SSID or Key

If you enter something incorrectly or need to change the SSID or WPA Key, go to the Tools menu and click FMS-Lite Mode to take the kiosk out of FMS-Lite Mode. When you click again to put the Kiosk back in FMS-Lite Mode, you will be re-prompted for the SSID and Key.

18.6 Imaging your Classmate (Veteran Image Download)

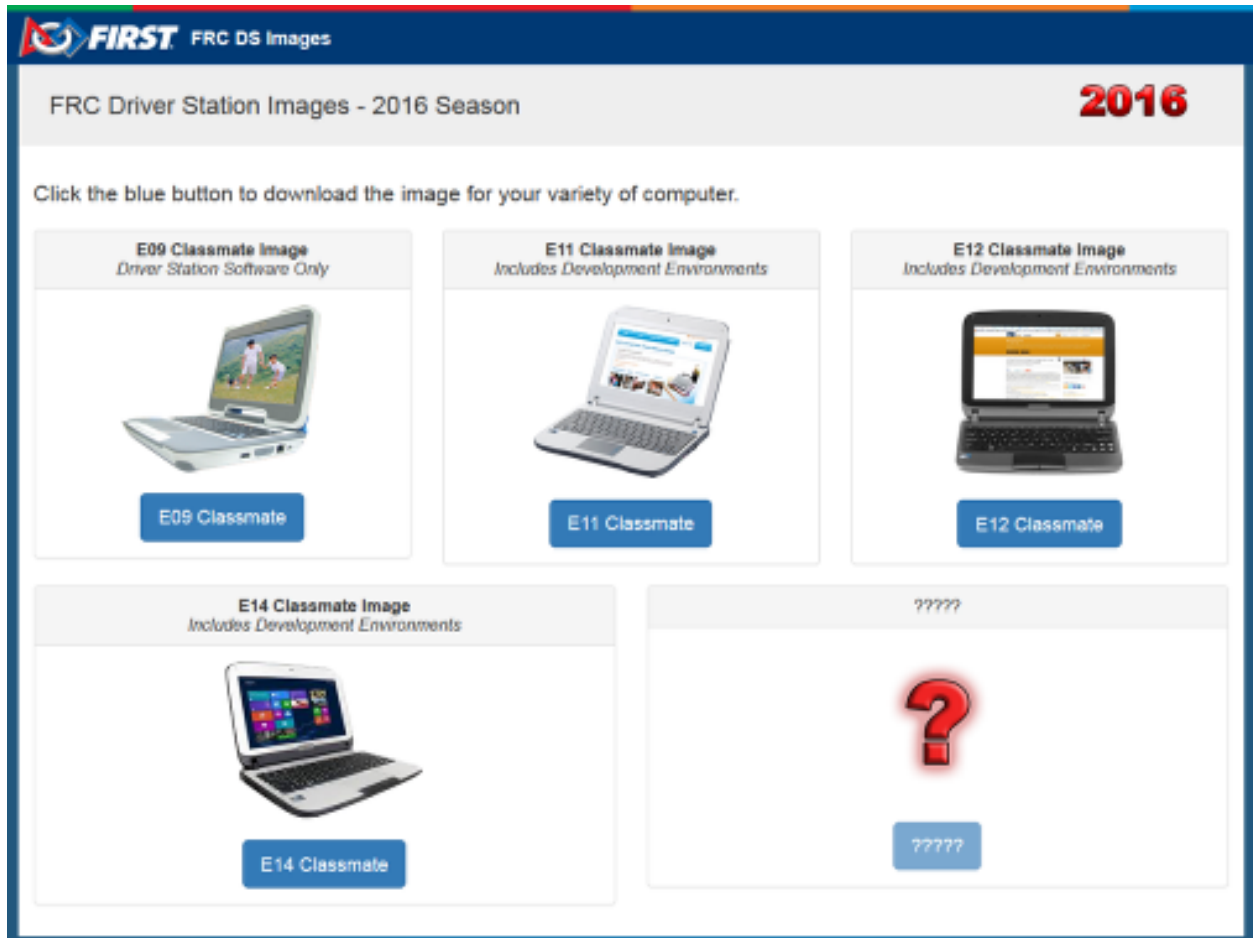
Note: Veteran teams are not required to re-image their classmate

This document describes the procedure for creating a bootable USB drive to restore the FRC® image on a Classmate computer. If you do not wish to re-image your Classmate then you can start with the appropriate document for C++/Java, LabVIEW, or DS only.

18.6.1 Prerequisites

1. E09, E11, E12, or E14 Classmate computer or Acer ES1 computer
2. 16GB or larger USB drive
3. 7-Zip software installed (download [here](#)). As of the writing of this document, the current released version is 19.00 (2019-02-21).
4. RMprepUSB software installed (download [here](#)). Scroll down the page and select the stable (Full) version download link. As of the writing of this document, the current stable version is 2.1.745.

18.6.2 Download the Computer Image



Download the image from the [FIRST FRC Driver Station System Image Portal](#). There are several computer images available, one for each model. On the download site, select the option that matches your computer by clicking the button below the image. Due to the limited size of the hard drive in the E09, it is supported with a DS/Utilities image only and does not have the IDEs for LabVIEW or C++/Java installed. All other images have the LabVIEW base installation already present.

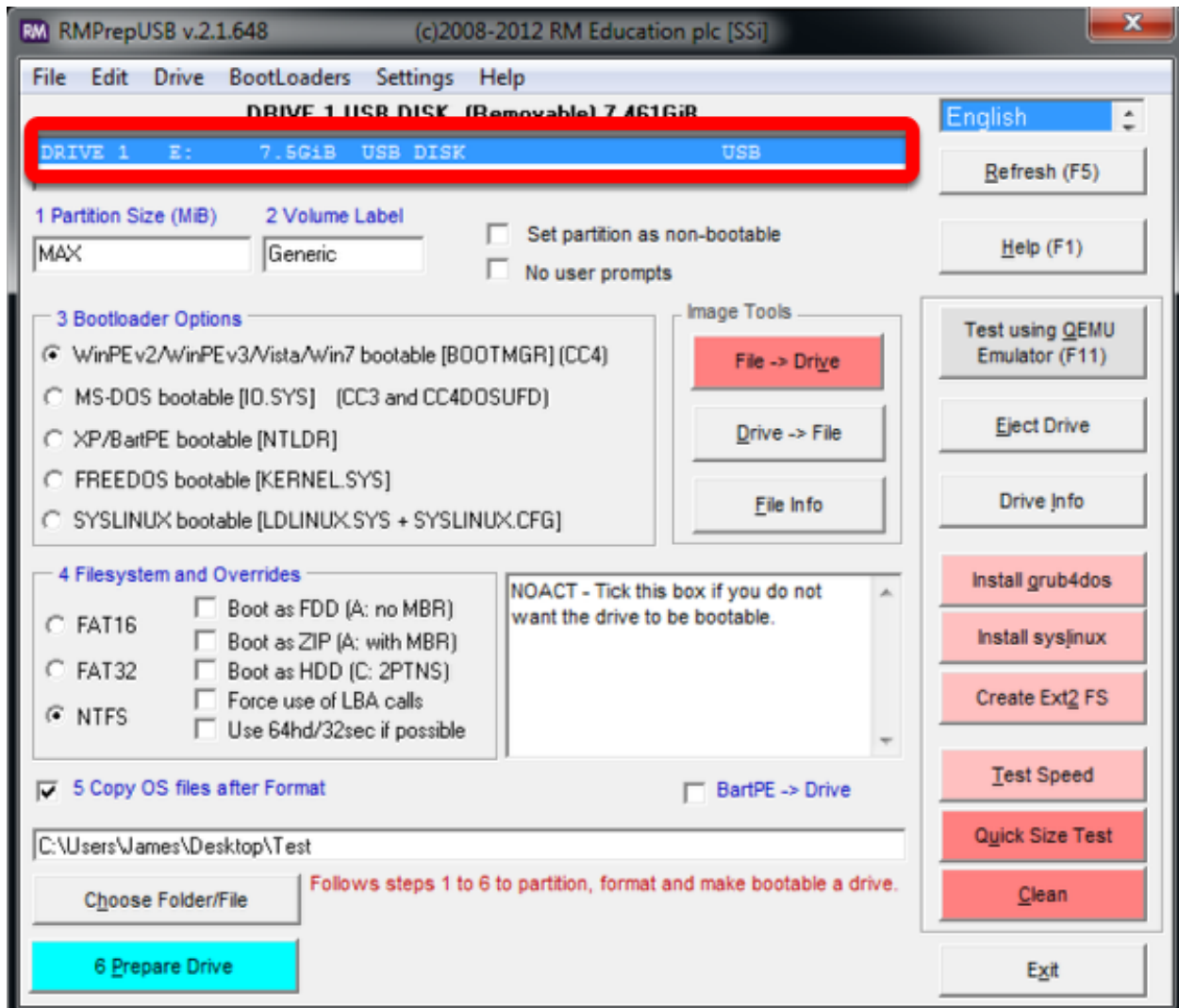
Note: These images only install the prerequisite core FRC software, it is still necessary to install the FRC specific updates. See the Update Software step for more information.

Warning: Due to computer availability, the E14 image provided is the 2018 image. If using this image, teams may need to remove the old IDE (LabVIEW or Eclipse) and install the new IDE.

18.6.3 Preparation

1. Place the image file downloaded from the site to a folder on your root drive (e.g. C:\2016_Image).
2. Connect 16GB or larger USB Flash drive to the PC to use as the new restoration drive.

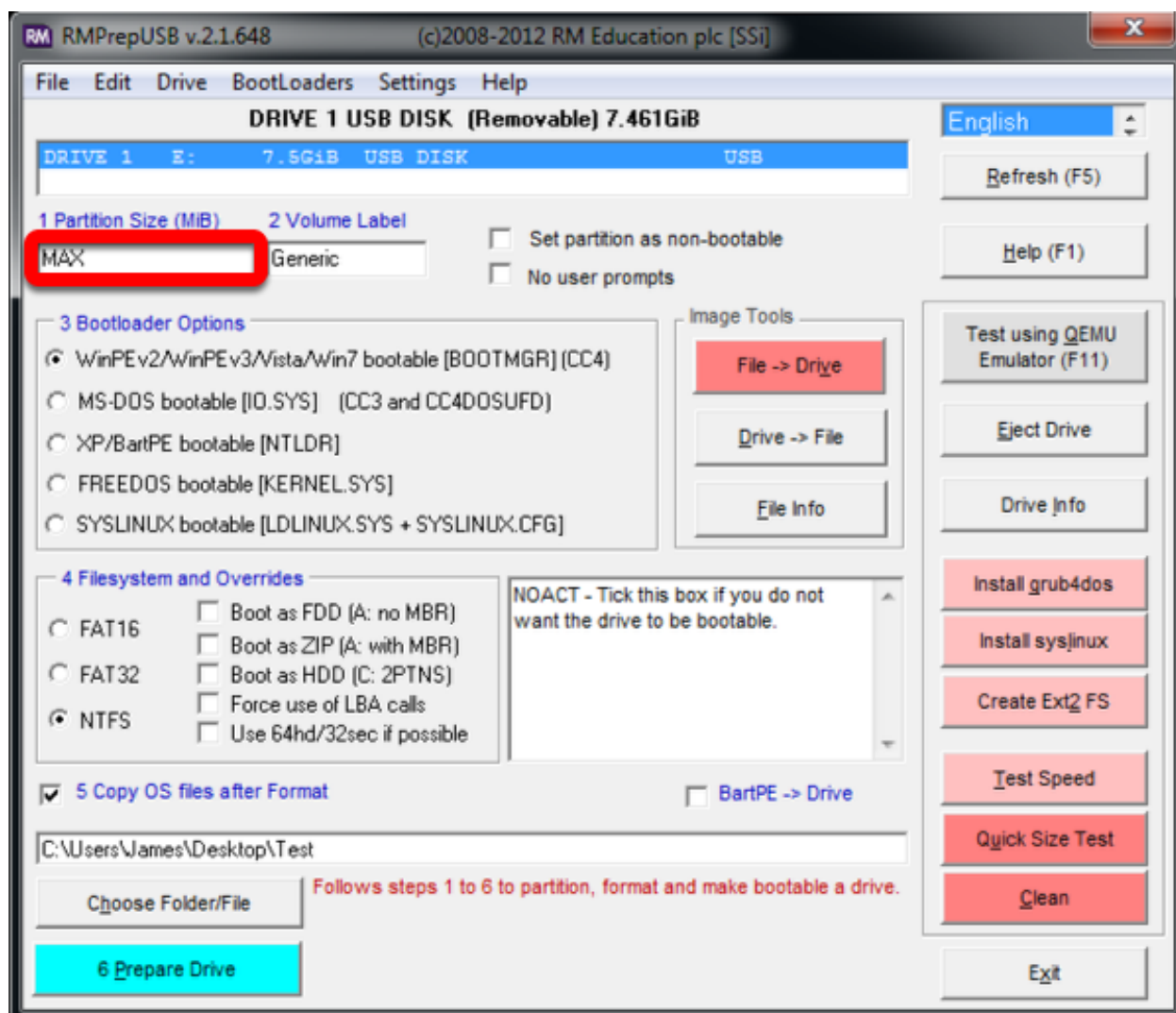
18.6.4 RMPrep



Start/Run RMPrepUSB

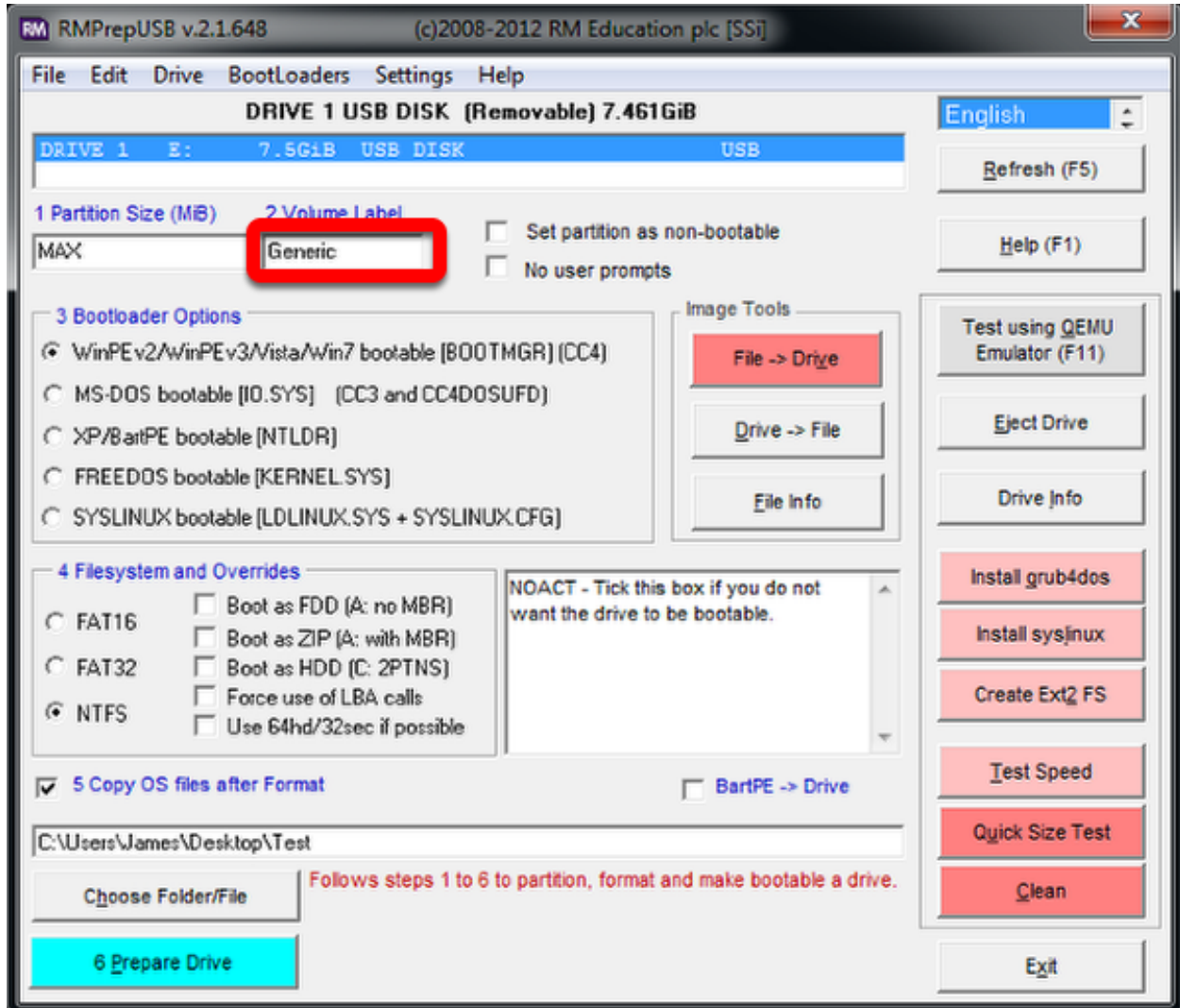
Select USB Drive

Set Partition Size



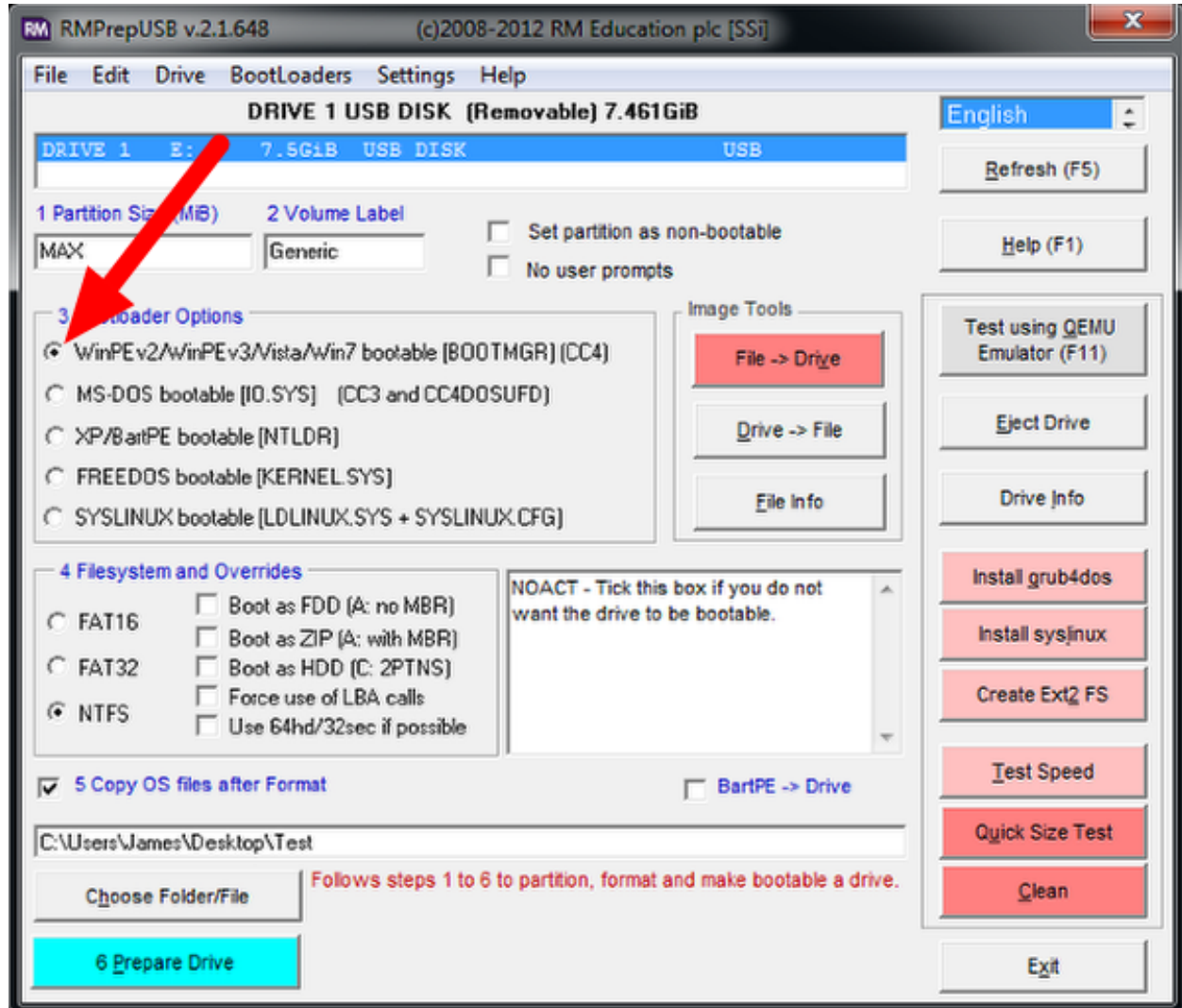
Set Partition Size to MAX

Set Volume Label



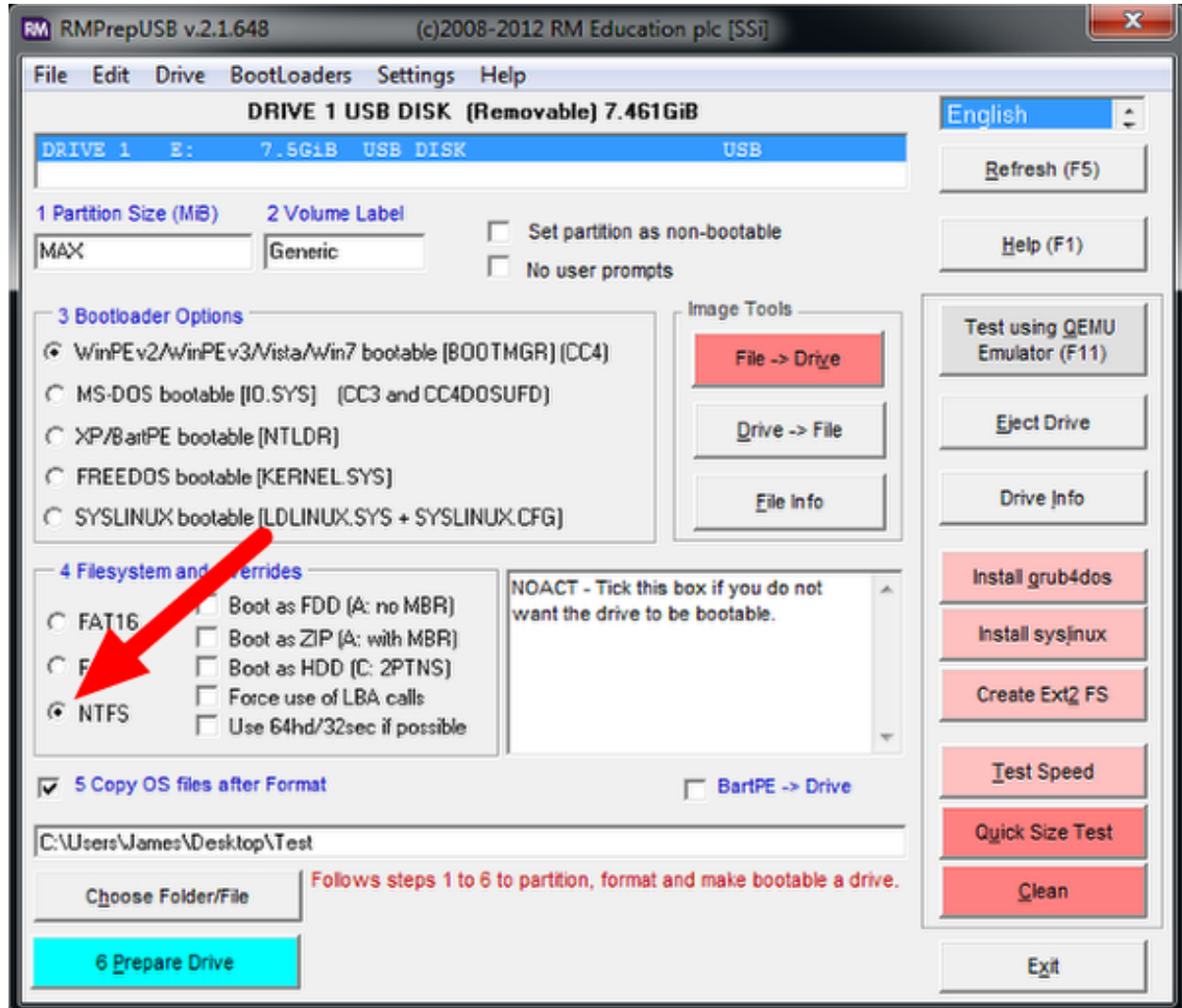
Set Volume Label to Generic

Set Bootloader Option



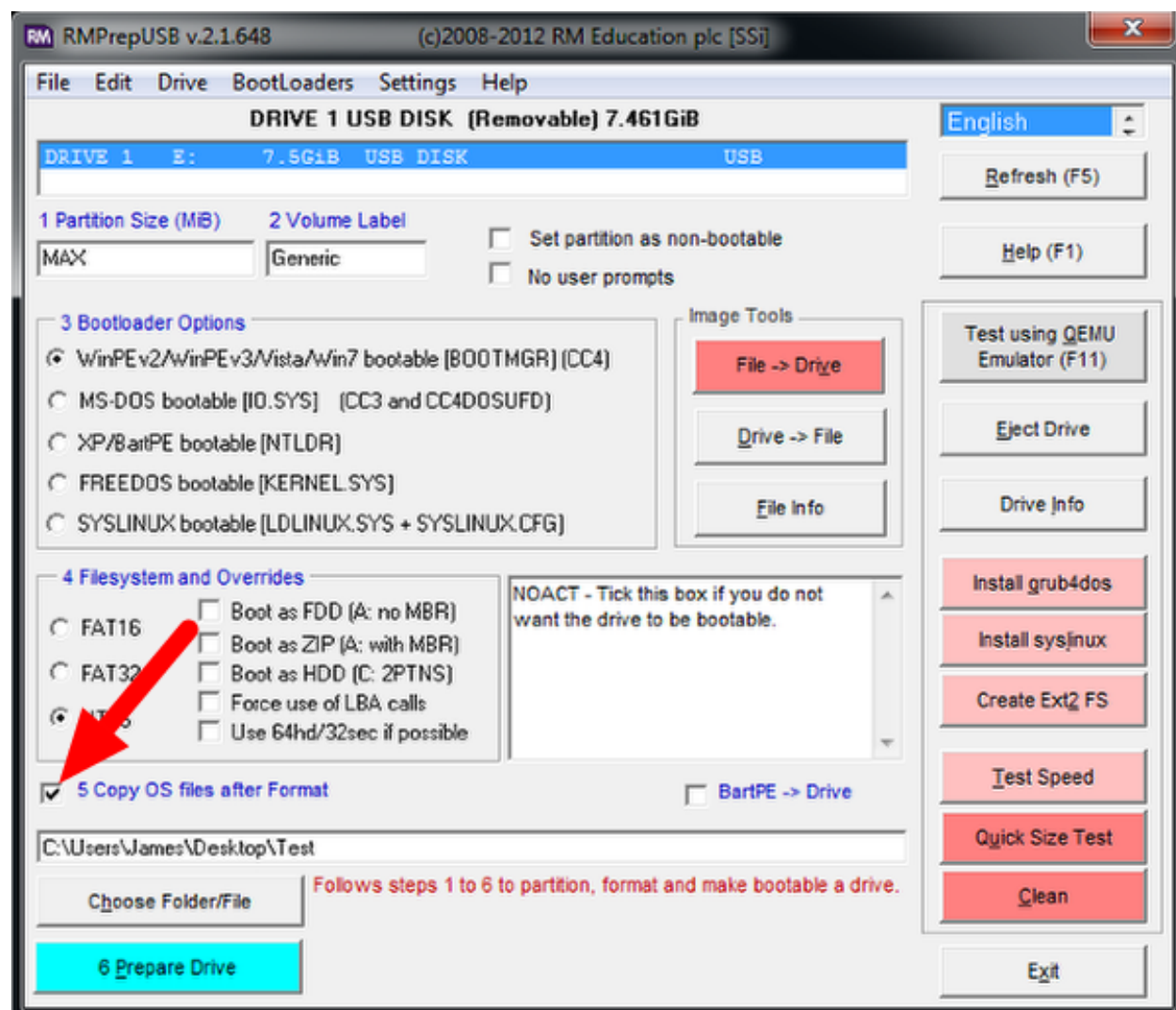
Select Bootloader Option “WinPE v2/WinPE v3/Vista/Win7 bootable”

Select Filesystem



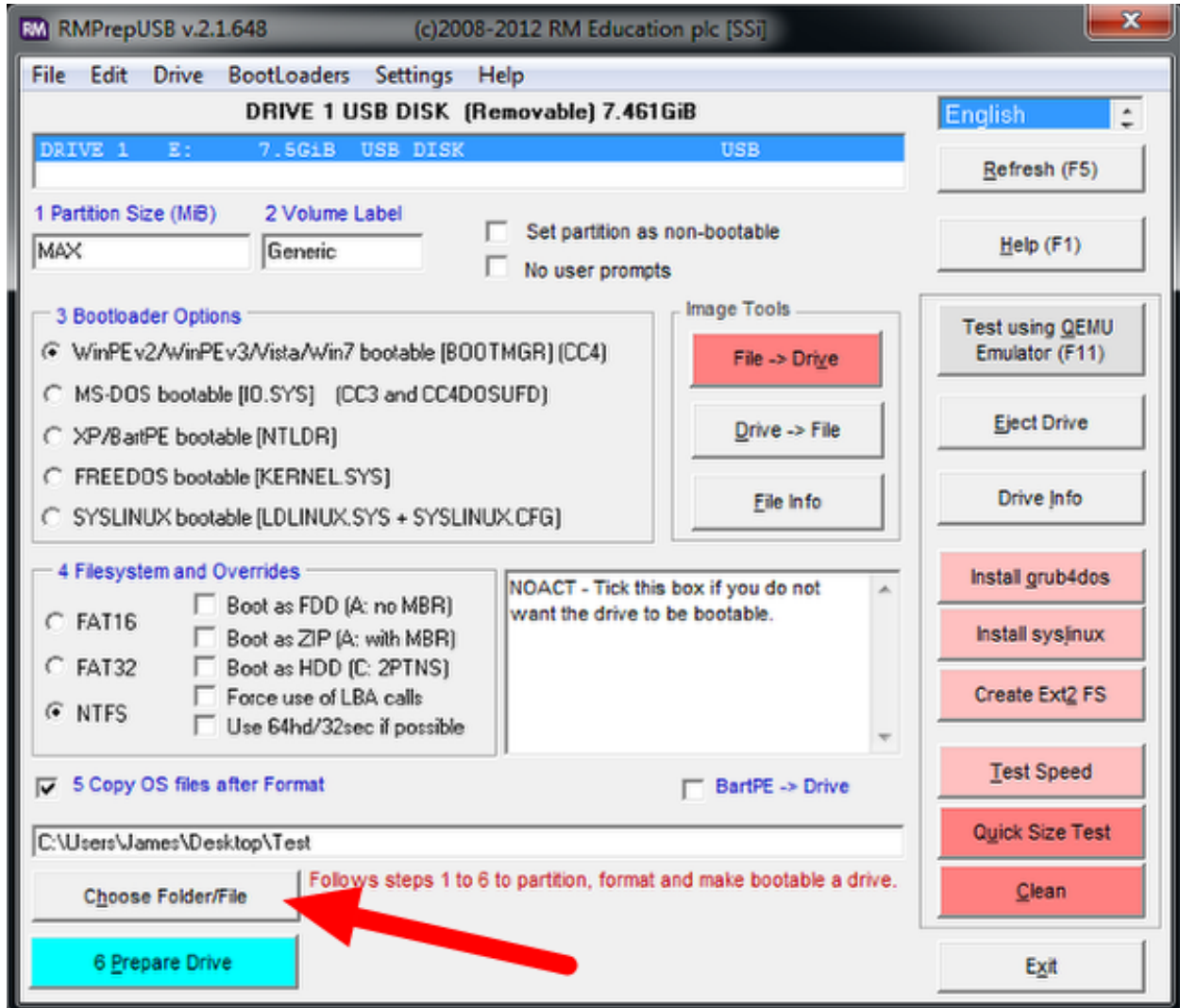
Select NTFS Filesystem

Copy OS Files Option



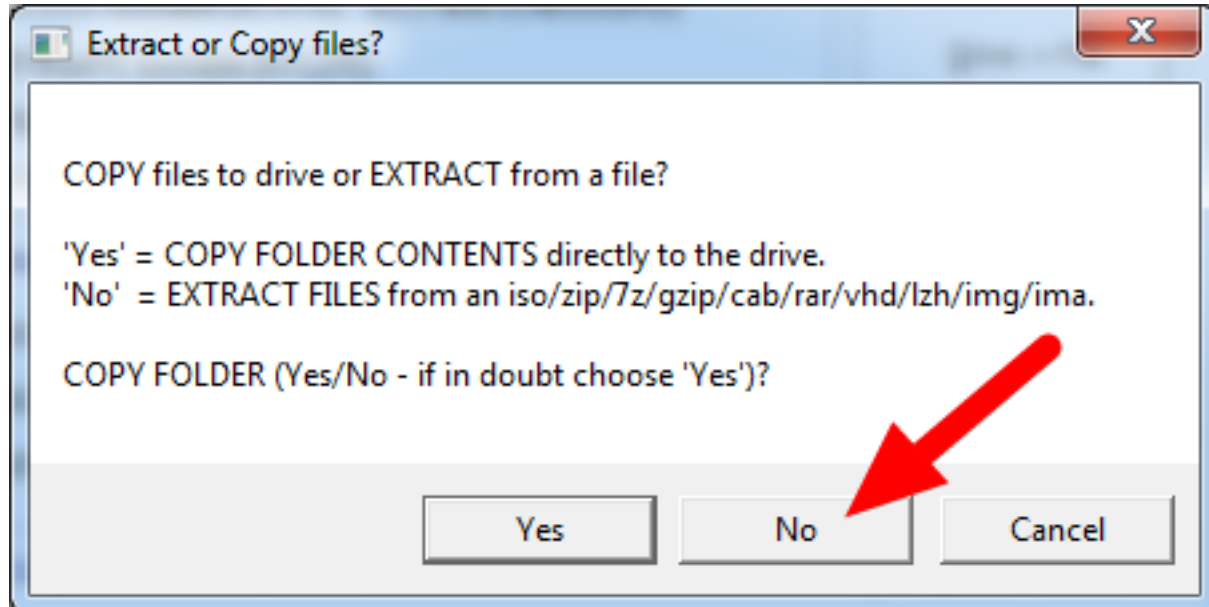
Ensure the “Copy OS files after Format” box is checked

Locate Image



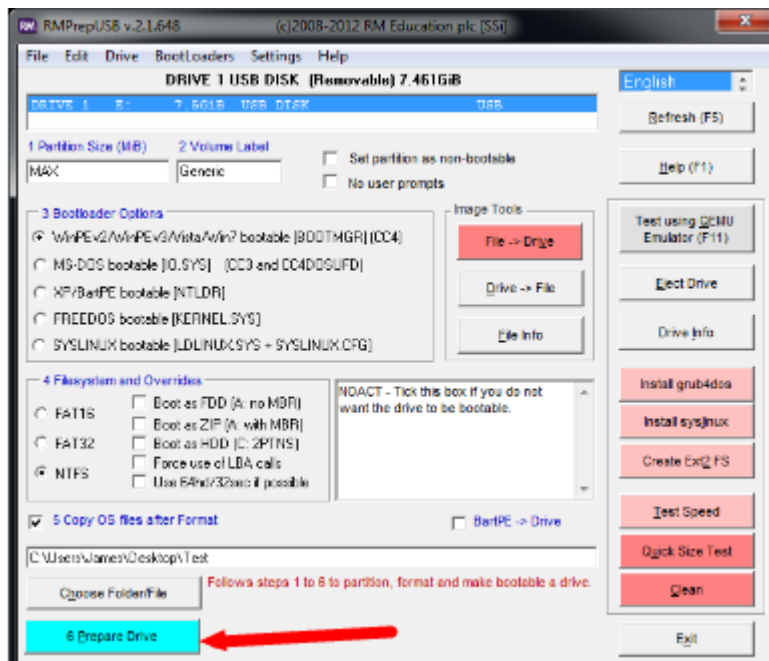
Select the "Choose Folder/File" button

Copy Files Dialog

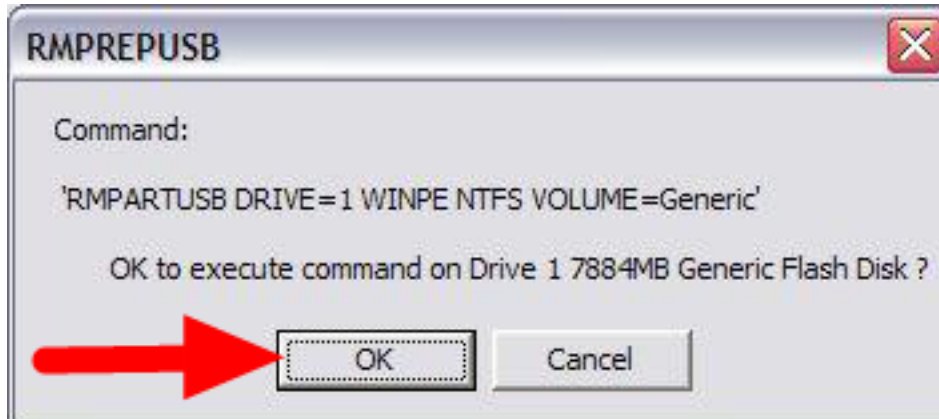


Choose "No" and select your .7z image

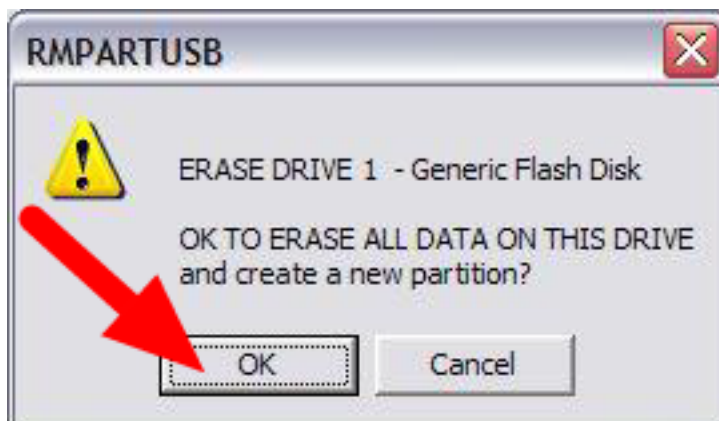
Prepare Drive



All configuration settings are now complete. Select "Prepare Drive" to begin the process

Confirmation Dialog 1

Click "OK" to execute the command on the selected USB Flash drive. A Command Prompt will open showing the progress

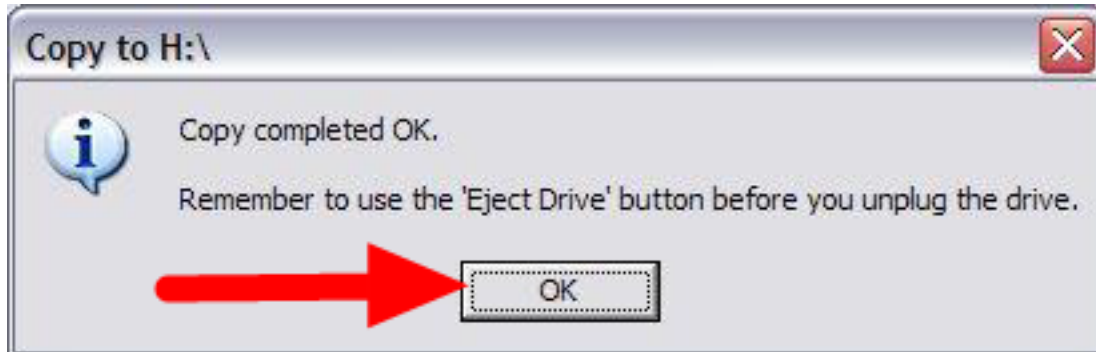
Confirmation Dialog 2

Click "OK" to format the USB drive

Danger: ALL DATA ON THE DRIVE WILL BE ERASED!

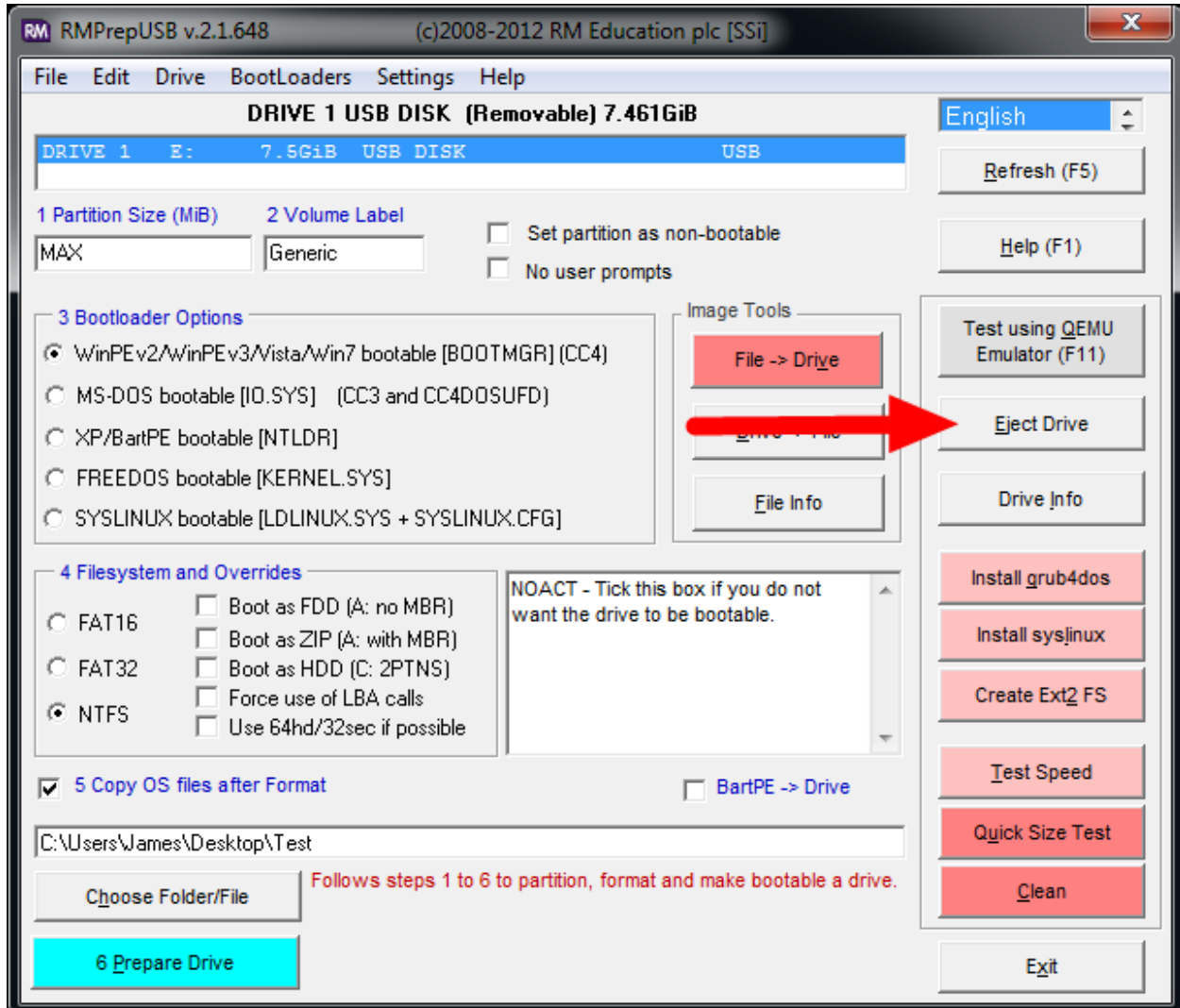
Decryption

Note: If you are using an encrypted version of the image downloaded before kickoff you will be prompted to enter the decryption key found at the end of the Kickoff video.

Copy Complete

Once formatting is complete, the restoration files will be extracted and copied to the USB drive. This process should take ~15 minutes when connected to a USB 2.0 port. When all files have been copied, this message will appear, press OK to continue.

Eject Drive

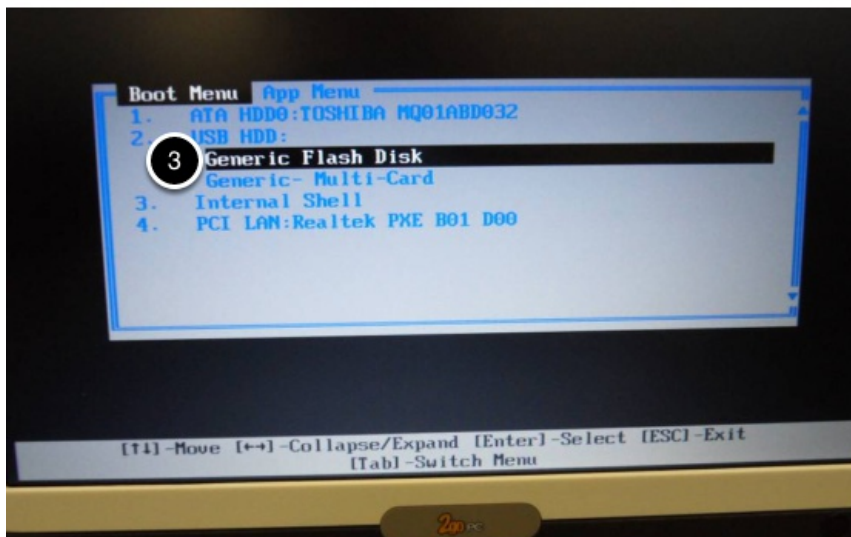
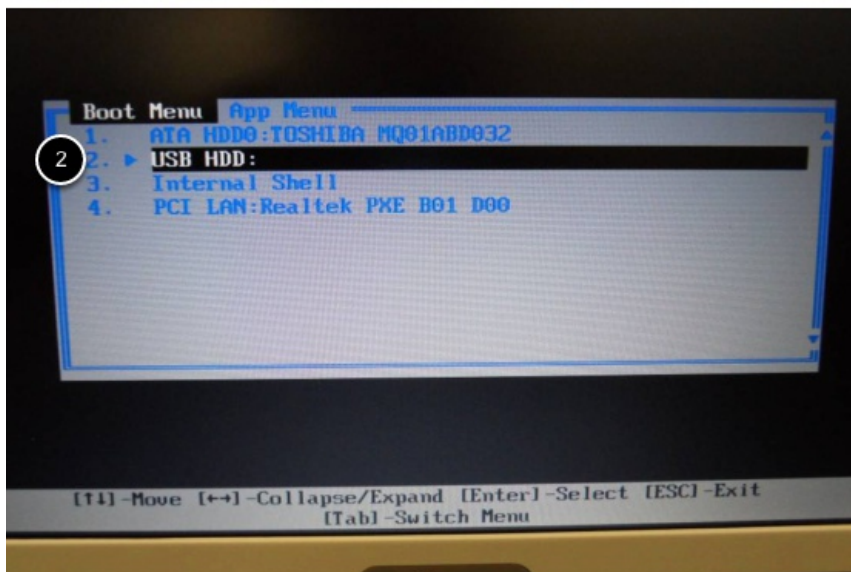
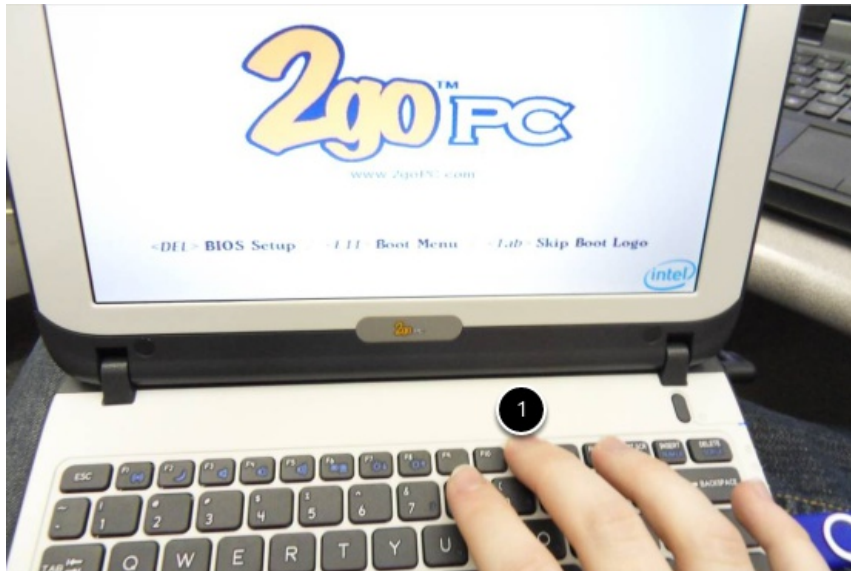


Press the “Eject Drive” button to safely remove the USB drive. The USB drive is now ready to be used to restore the image onto the PC.

18.6.5 Hardware Setup

1. Make sure the computer is turned off, but plugged in.
2. Insert the USB Thumb Drive into a USB port on the Driver Station computer.

Boot to USB



Classmate:

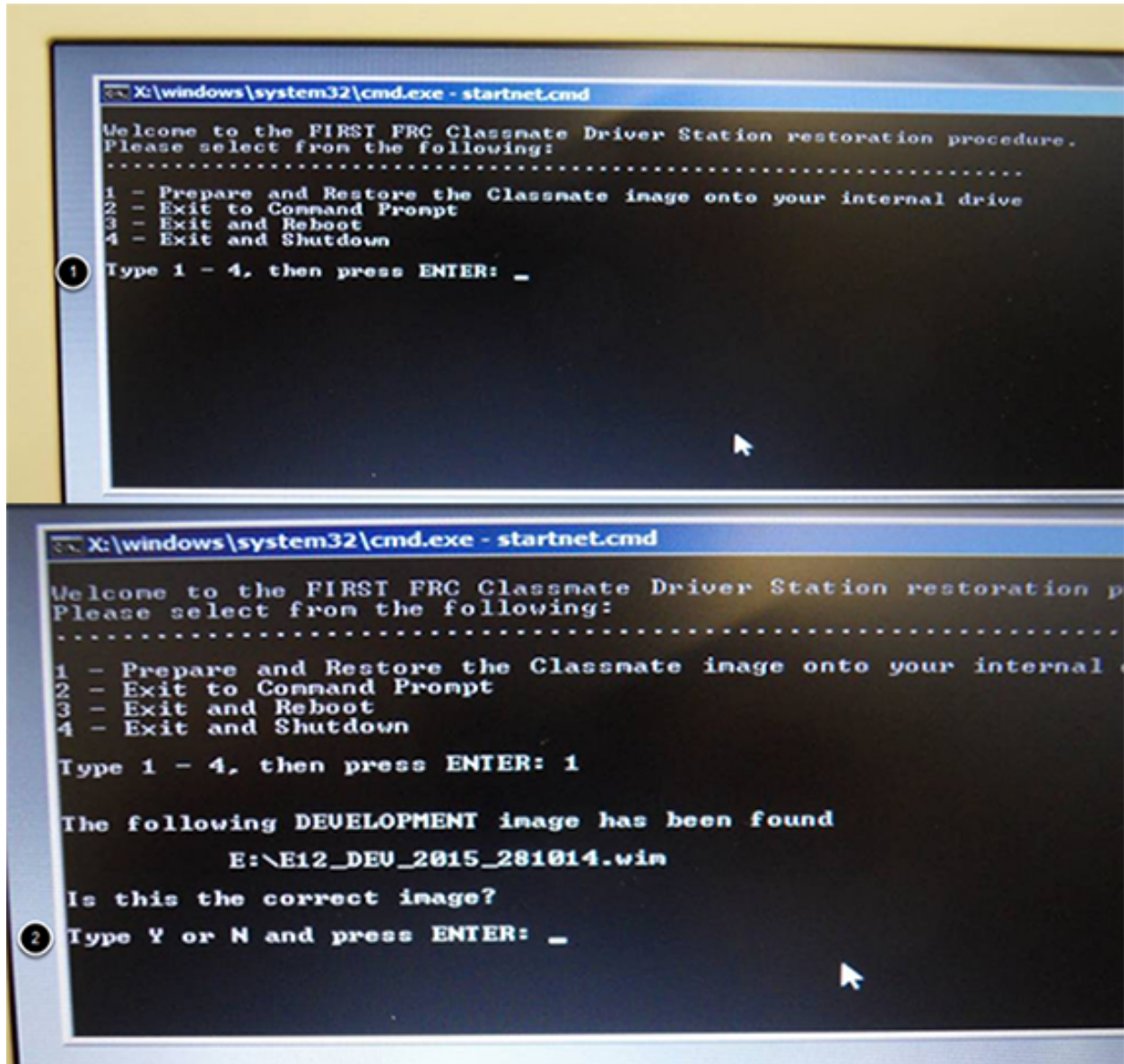
1. Power on the Classmate and tap the F11 key on the keyboard. Tapping the F11 key during boot will bring up the boot menu.
2. Use the up/down keys to select the **USB HDD:** entry on the menu, then press the right arrow to expand the listing
3. Use the up/down arrow keys on the keyboard to select the USB device (it will be called "Generic Flash Disk"). Press the ENTER key when the USB device is highlighted.

Acer ES1:

1. Power on the computer and tap the F12 key on the keyboard. Tapping the F12 key during boot will bring up the boot menu.
2. Use the up/down keys to select the **USB HDD: Generic** entry on the menu, then press the ENTER key when the USB device is highlighted.

Acer ES1: If pressing F12 does not pull up the boot menu or if the USB device is not listed in the boot menu, see "Checking BIOS Settings" at the bottom of this article.

Image the Classmate



1. To confirm that you want to reimage the Classmate, type "1" and press ENTER.
2. Then, type "Y" and press ENTER. The Classmate will begin re-imaging. The installation will take 15-30 minutes.
3. When the installation is complete, remove the USB drive.
4. Restart the Classmate. The Classmate will boot into Windows.

18.6.6 Initial Driver Station Boot

The first time the Classmate is turned on, there are some unique steps, listed below, that you'll need to take. The initial boot may take several minutes; make sure you do not cycle power during the process.

Note: These steps are only required during original startup.

Enter Setup

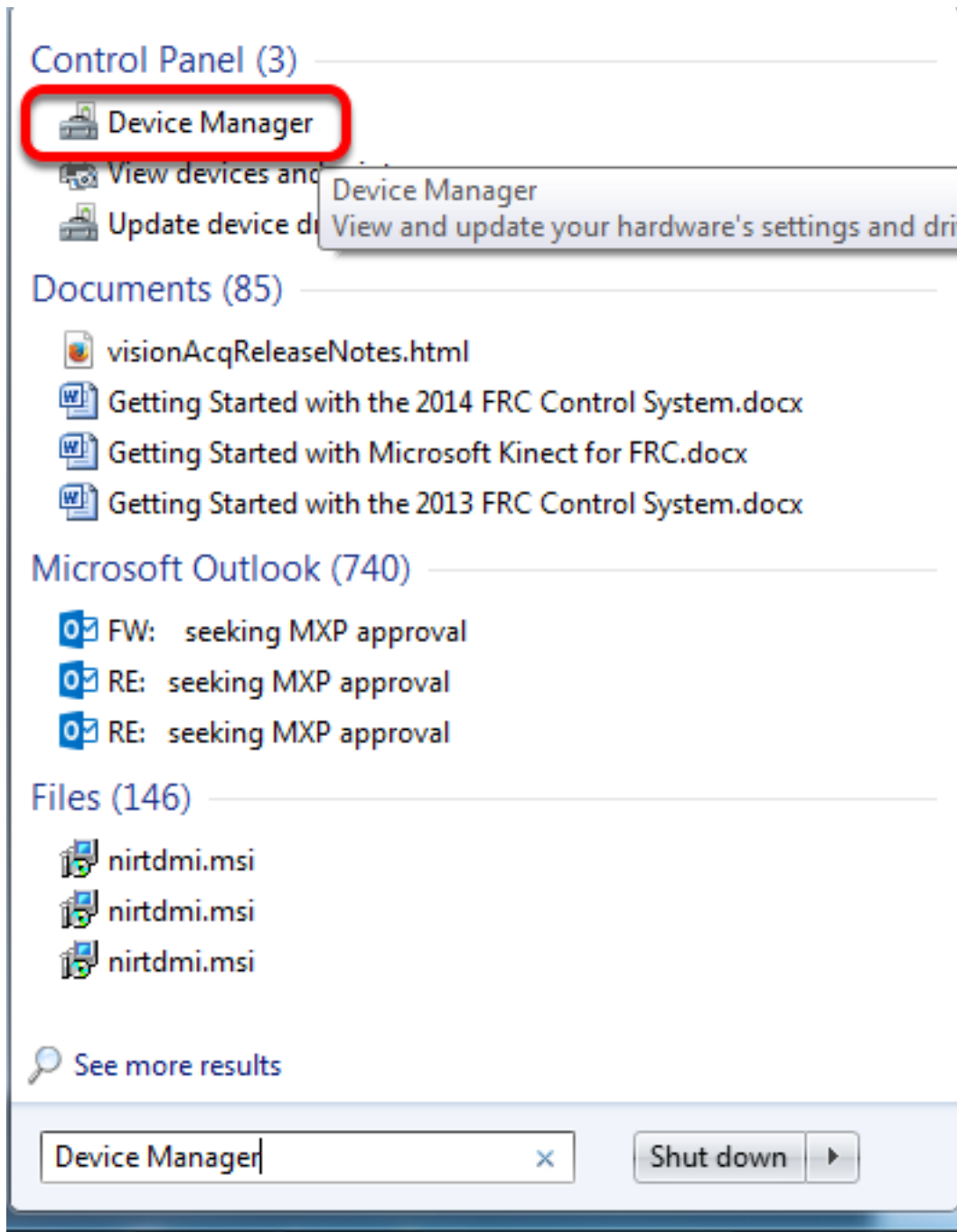
1. Log into the Developer account.
2. Click "Ask me later".
3. Click "OK". The computer now enters a Set Up that may take a few minutes.

Activate Windows

1. Establish an Internet connection.
2. Once you have an Internet connection, click the Start menu, right click "Computer" and click "Properties".
3. Scroll to the bottom section, "Windows activation", and Click "Activate Windows now"
4. Click "Activate Windows online now". The activation may take a few minutes.
5. When the activation is complete, close all of the windows.

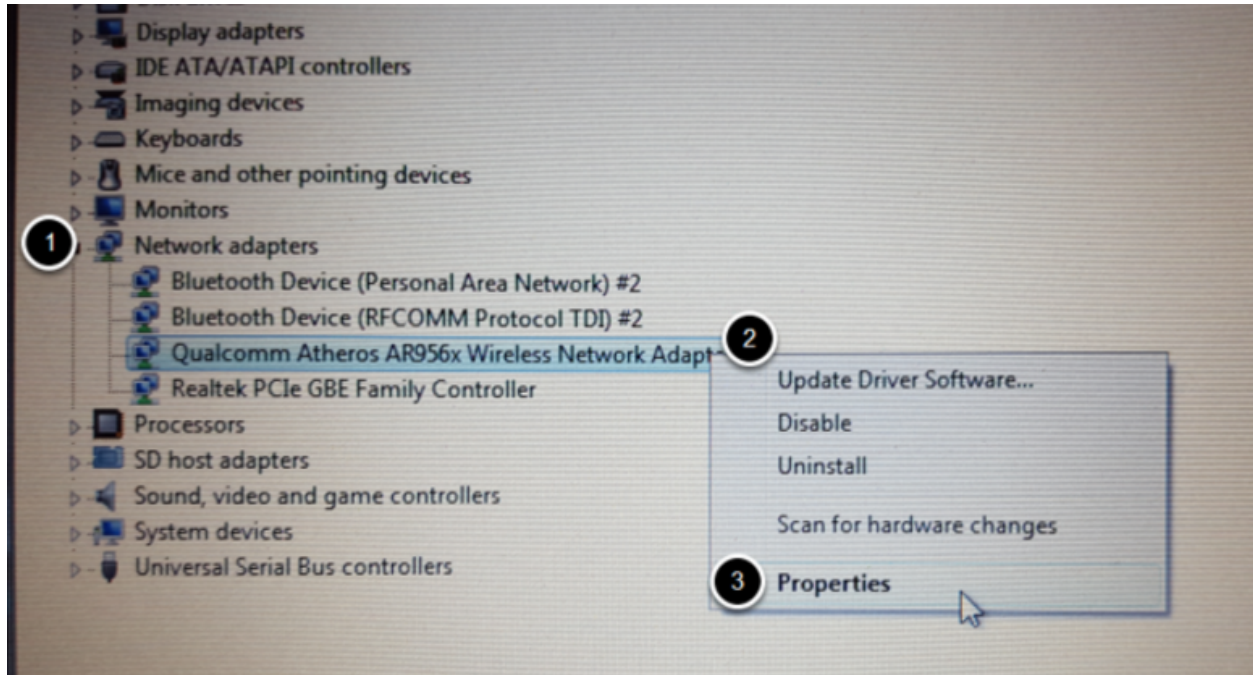
Microsoft Security Essentials

Navigate through the Microsoft Security Essentials Setup Wizard. Once it is complete, close all of the windows.

Acer ES1: Fix Wireless Driver**Acer ES1 PC only!**

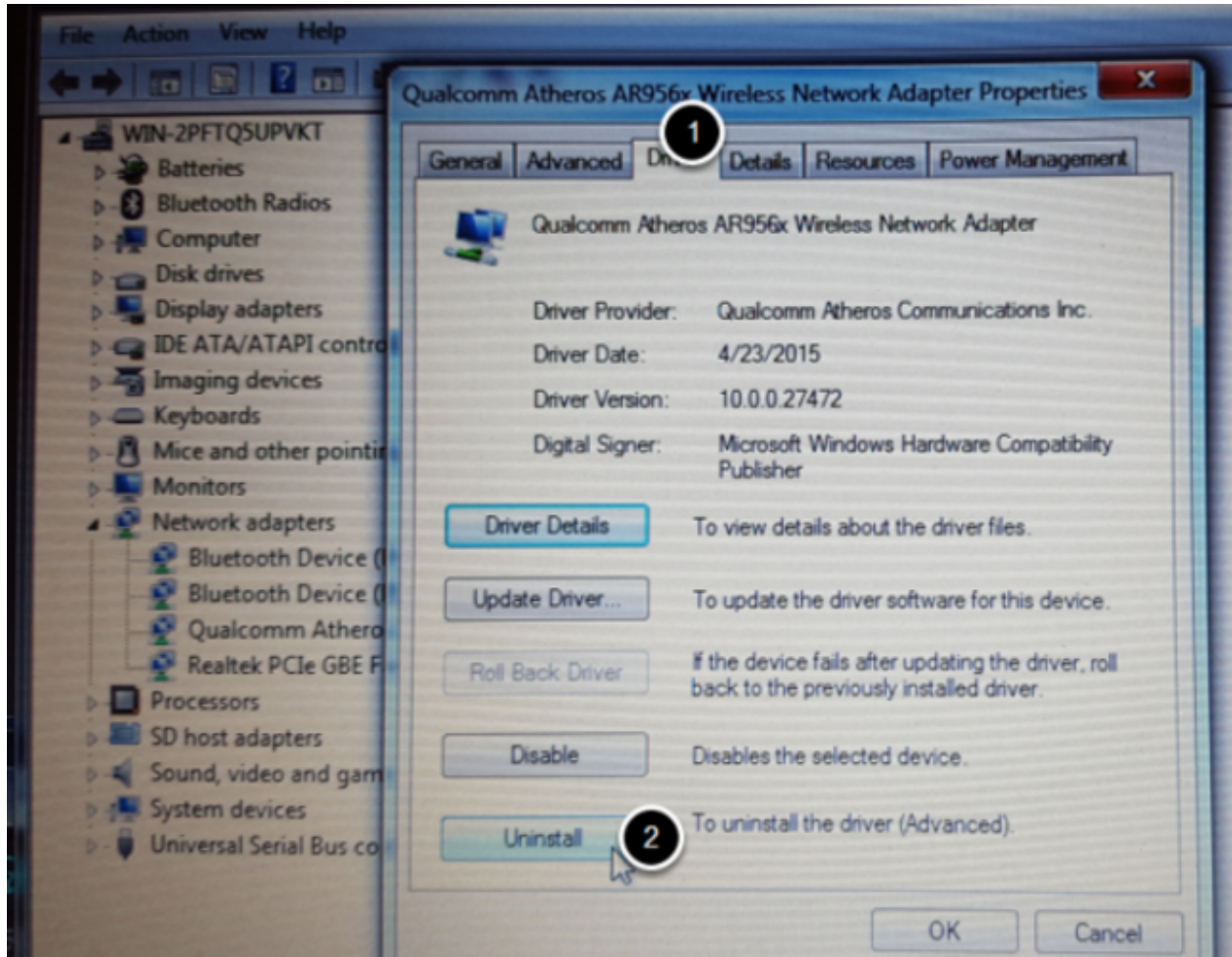
The default wireless driver in the image may have issues with intermittent communication with the robot radio. The correct driver is in the image, but could not be set to load by default. To load the correct driver, open the Device Manager by clicking start, typing "Device Manager" in the box and clicking Device Manager.

Open Wireless Device Properties



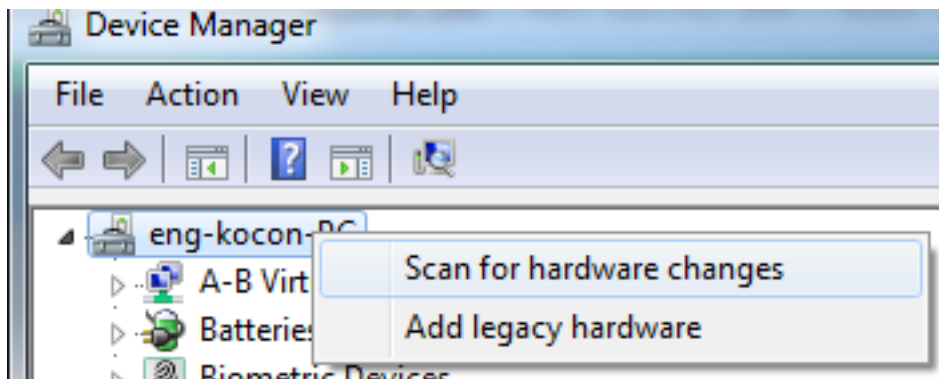
Click on the arrow next to Network Adapters to expand it and locate the Wireless Network Adapter. Right click the adapter and select Properties.

Uninstall-Driver



Click on the Driver tab, then click the Uninstall button. Click Yes at any prompts.

Scan for New Hardware

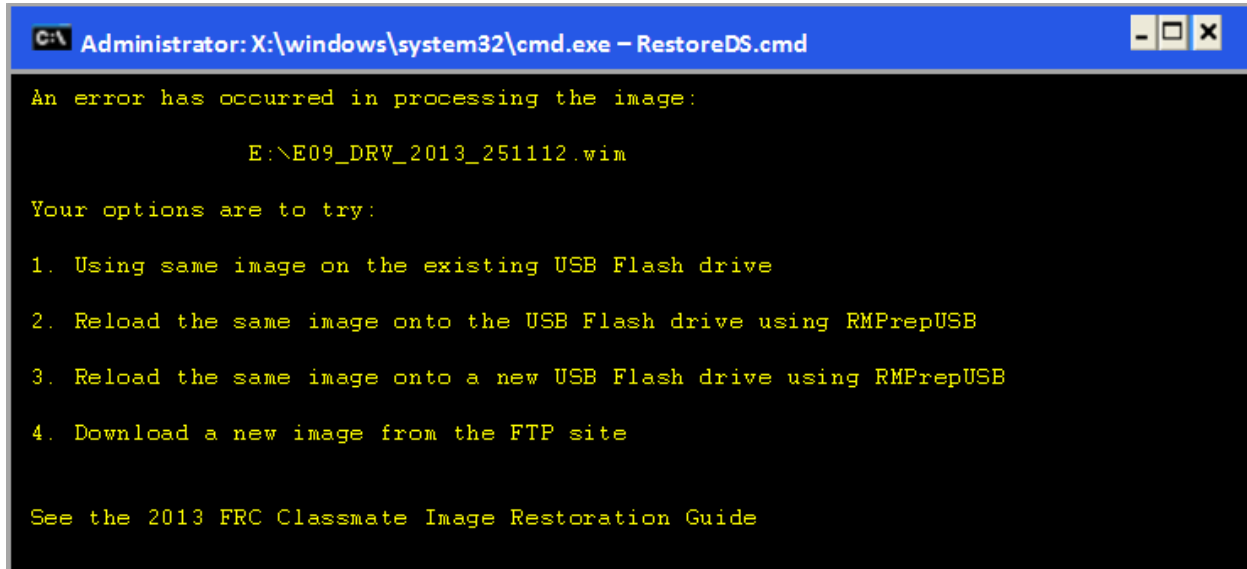


Right click on the top entry of the tree and click "Scan for hardware changes". The wireless adapter should automatically be re-detected and the correct driver should be installed.

18.6.7 Update Software

In order for the Classmate images to be prepared on time, they are created before the final versions of the software were ready. To use the software for FRC some additional components will need to be installed. LabVIEW teams should continue with Installing the FRC Game Tools (All Languages). C++ or Java teams should continue Installing C++ and Java Development Tools for FRC.

18.6.8 Errors during Imaging Process



If an error is detected during the imaging process, the following screen will appear. Note that the screenshot below shows the error screen for the Driver Station-only image for the E09. The specific image filename shown will vary depending on the image being applied.

The typical reason for the appearance of this message is due to an error with the USB device on which the image is stored. Each option is listed below with further details as to the actions you can take in pursuing a solution. Pressing any key once this error message is shown will return the user to the menu screen shown in Image the Classmate.

Option 1

Using same image on the existing USB Flash drive To try this option, press any key to return to the main menu and select #1. This will run the imaging process again.

Option 2

Reload the same image onto the USB Flash drive using RMPrepUSB It's possible the error message was displayed due to an error caused during the creation of the USB Flash drive (e.g. file copy error, data corruption, etc.) Press any key to return to the main menu and select #4 to safely shutdown the Classmate then follow the steps starting with RMPrep to create a new USB Restoration Key using the same USB Flash drive.

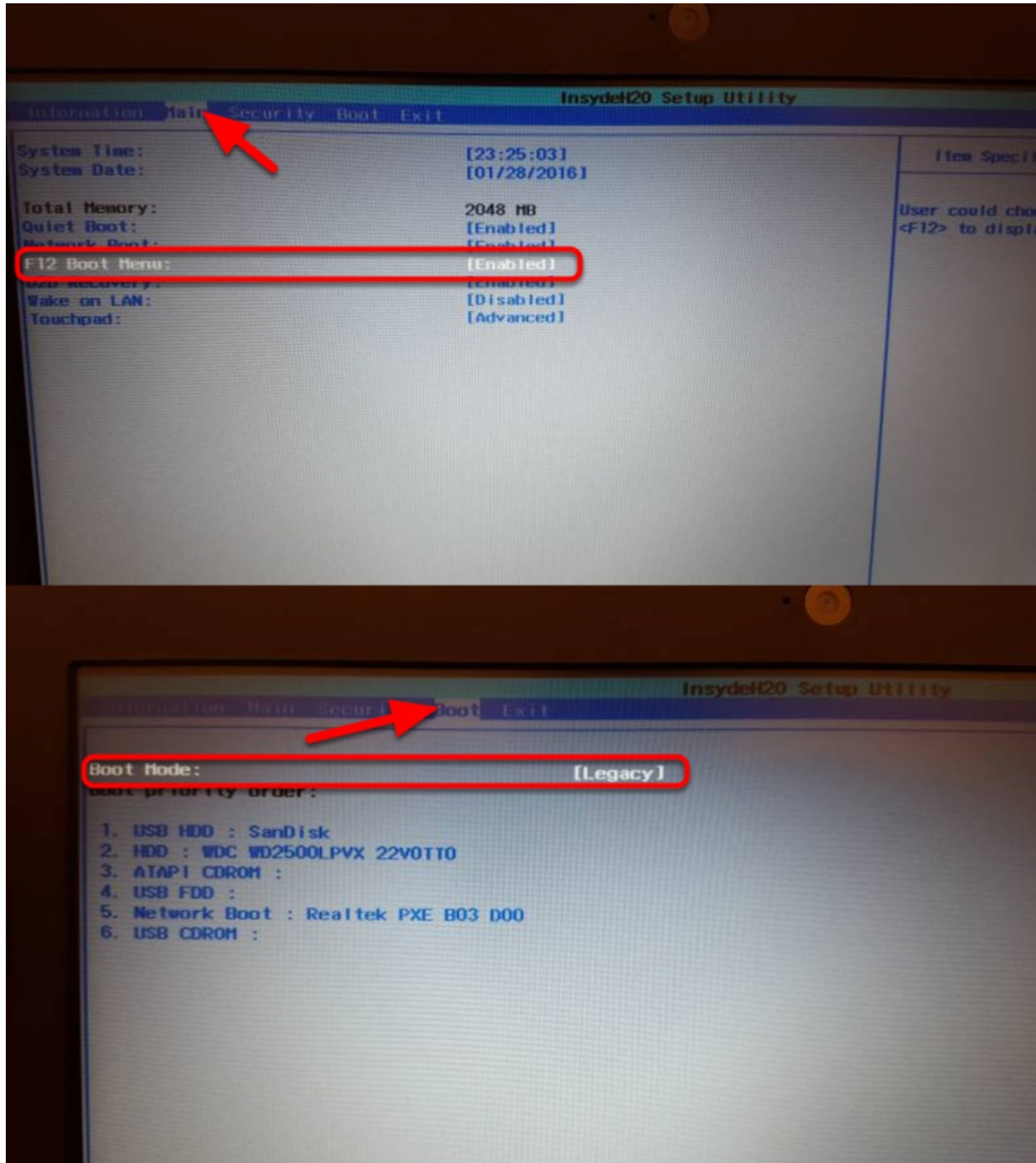
Option 3

Reload the same image onto a new USB Flash drive using RMPrepUSB The error message displayed may also be caused by an error with the USB Flash drive itself. Press any key to return to the main menu and select #4 to safely shutdown the Classmate. Select a new USB Flash drive and follow the steps starting with RMPrep.

Option 4

Download a new image An issue with the downloaded image may also cause an error when imaging. Press any key to return to the main menu and select #4 to safely shutdown the Classmate. Starting with Download the Classmate Image create a new copy of the imaging stick.

Checking BIOS Settings



If you are having difficulty booting to USB, check the BIOS settings to insure they are correct. To do this:

- Repeatedly tap the **F2** key while the computer is booting to enter the BIOS settings
- Once the BIOS settings screen has loaded, use the right and left arrow keys to select the "Main" tab, then check if the line for "F12 Boot Menu" is set to "Enabled". If it is not, use the Up/Down keys to highlight it, press Enter, use Up/Down to select "Enabled" and

press Enter again.

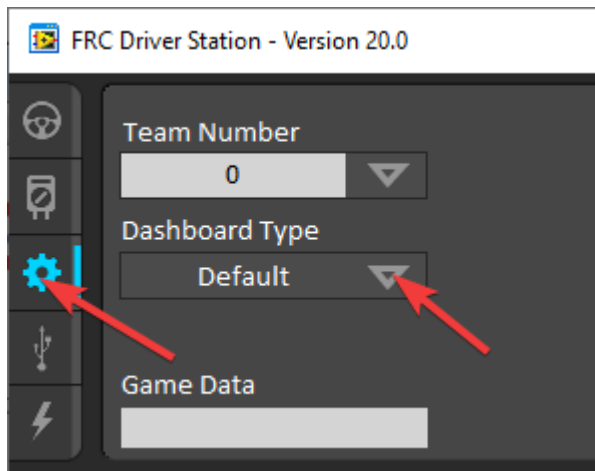
- Next, use the Left/Right keys to select the “Boot” tab. Make sure that the “Boot Mode” is set to “Legacy”. If it is not, highlight it using UpDown, press Enter, highlight “Legacy” and press Enter again. Press Enter to move through any pop-up dialogs you may see.
- Press F10 to save any changes and exit.

18.7 Manually Setting the Driver Station to Start Custom Dashboard

Note: If WPILib is not installed to the default location (such as when files are copied to a PC manually), the dashboard of choice may not launch properly. To have the DS start a custom dashboard when it starts up, you have to manually modify the settings for the default dashboard.

Warning: This is not needed for most installations, try using the appropriate *Dashboard Type setting* for your language first.

18.7.1 Set Driver Station to Default

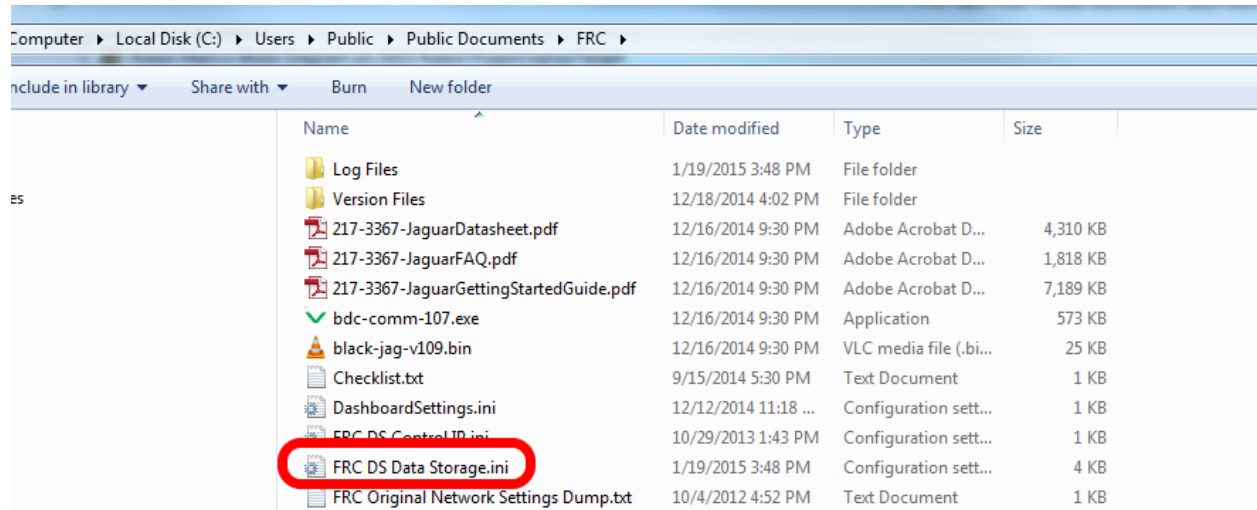


Open the Driver Station software, click on the Setup tab and set the Dashboard setting to Default. **Then close the Driver Station!**

18.7.2 Locate Dashboard JAR file

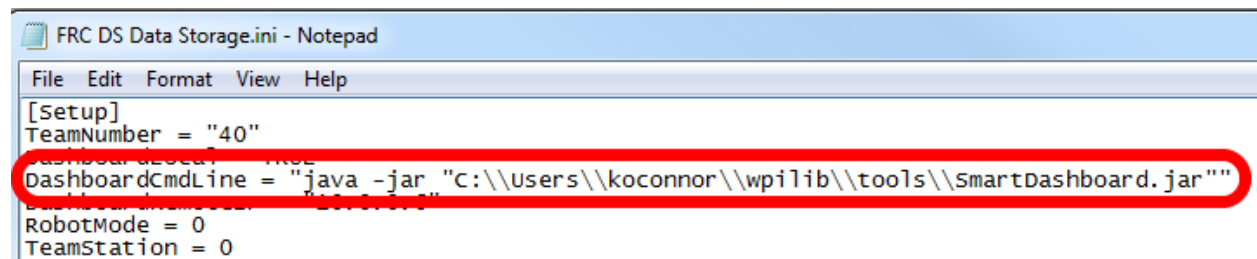
Find the location of the dashboard jar file.

18.7.3 Open DS Data Storage file



Browse to C:\Users\Public\Documents\FRC and double click on FRC DS Data Storage to open it.

18.7.4 DashboardCmdLine



Locate the line beginning with `DashboardCmdLine`. Modify it to point to the dashboard to launch when the driver station starts

LabVIEW Custom Dashboard

Replace the string after `=` with `"C:\\\\PATH\\\\T0\\\\DASHBOARD.exe"` where the path specified is the path to the dashboard exe file. Save the FRC DS Data Storage file.

Java Dashboard

Replace the string after = with `java -jar "C:\\PATH\\TO\\DASHBOARD.jar"` where the path specified is the path to the dashboard jar file. Save the FRC DS Data Storage file.

Tip: Shuffleboard and Smartdashboard require Java 11.

Dashboard from WPILib installer

Replace the string after = with `wscript "C:\\Users\\Public\\wpilib\\YYYY\\tools\\DASHBOARD.vbs"` where YYYY is the year and DASHBOARD.vbs is either Shuffleboard.vbs or Smartdashboard.vbs. Save the FRC DS Data Storage file.

18.7.5 Launch Driver Station

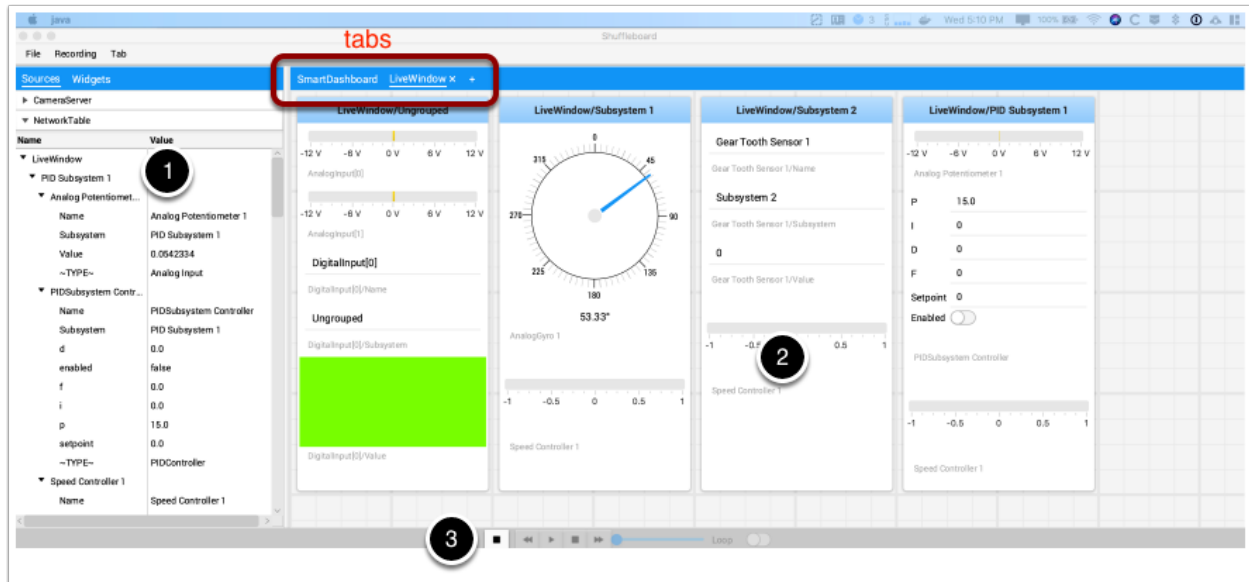
The Driver Station should now launch the dashboard each time it is opened.

19.1 Shuffleboard - Getting Started

19.1.1 Tour of Shuffleboard

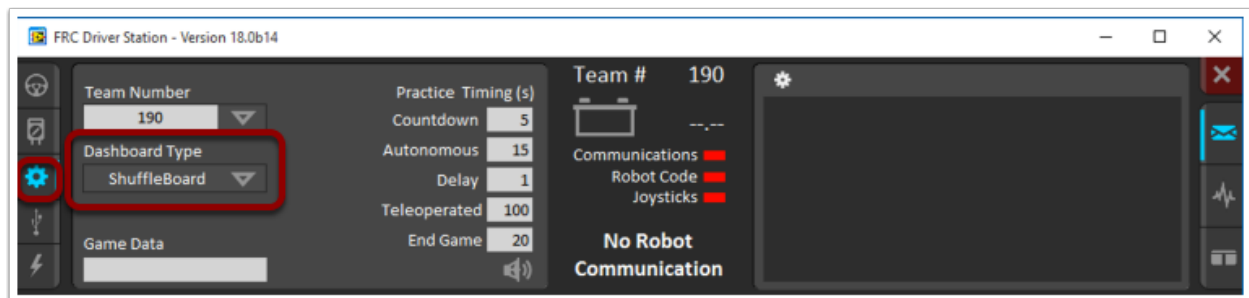
Shuffleboard is a dashboard for FRC® based on newer technologies such as JavaFX that are available to Java programs. It is designed to be used for creating dashboards for C++ and Java programs. If you've used SmartDashboard in the past then you are already familiar with many of the features of Shuffleboard since they fundamentally work the same way. But Shuffleboard has many features that aren't in SmartDashboard. Here are some of the highlights:

- Graphics is based on **JavaFX**, the Java graphics standard. Each of the components has an associated style sheet so it becomes possible to have different “skins” or “themes” for Shuffleboard. We supply default light and dark themes.
- Shuffleboard supports **multiple sheets for the display of your data**. In fact you can create a new sheet (shown as a tab in the Shuffleboard window) and indicate if and which data should be autopopulated on it. By default there is a Test tab and a SmartDashboard tab that are autopopulated as data arrives. Other tabs might be for robot debugging vs. driving.
- Graphical **display elements (widgets) are laid out on a grid** to keep the interface clean and easy to read. You can change the grid size to have more or less resolution in your layouts and visual cues are provided to help you change your layout using drag and drop. Or you can choose to turn off the grid lines although the grid layout is preserved.
- Layouts are saved and the previous layout is instantiated by default when you run shuffleboard again.
- There is a **record and playback** feature that lets you review the data sent by your robot program after it finishes. That way you can carefully review the actions of the robot if something goes wrong.
- **Graph widgets are available for numeric data** and you can drag data onto a graph to see multiple points at the same time and on the same scale.
- You can extend Shuffleboard by writing your own widgets that are specific to your team's requirements. Documentation on extending it are on the [Shuffleboard GitHub Wiki](#).



1. **Sources area:** Here are data sources from which you can choose values from NetworkTables or other sources to display by dragging a value into one of the tabs
2. **Tab panes:** This is where you data is displayed from the robot or other sources. In this example it is Test-mode subsystems that are shown here in the LiveWindow tab. This area can show any number of tabbed windows, and each window has it's own set of properties like grid size and auto-populate.
3. **Record/playback controls:** set of media-like controls where you can playback the current session to see historical data

Starting Shuffleboard



You can start Shuffleboard in one of four ways:

1. You can automatically start it when the Driver Station starts by setting the "Dashboard Type" to Shuffleboard in the settings tab as shown in the picture above.
2. You can run it by double-clicking the Shuffleboard icon on the Windows Desktop.
3. You can run it by double-clicking on the shuffleboard.XXX file (where XXX is .vbs on Windows and .py on Linux or macOS) in ~/WPILib/YYYY/tools/ (where YYYY is the year and ~ is C:\Users\Public on Windows). This is useful on a development system that does not have the Driver Station installed such as a macOS or Linux system.

4. You can start it from the command line by typing the command: `shuffleboard` on Windows or `python shuffleboard.py` on macOS or Linux from `~/WPILib/YYYY/tools` directory (where YYYY is the year and ~ is `C:\Users\Public` on Windows). This is often easiest on a development system that doesn't have the Driver Station installed.

Note: The `.vbs` (Windows) and `.py` (macOS/Linux) scripts help launch the tools using the correct JDK.

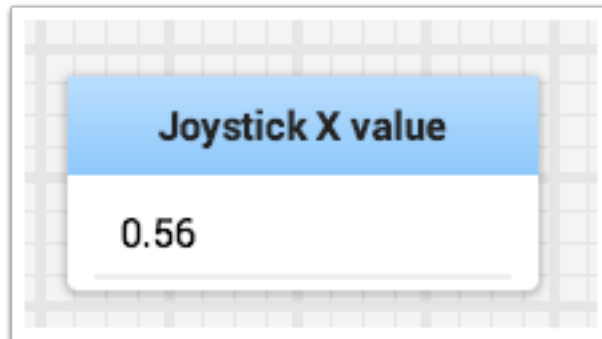
Getting robot data onto the dashboard

The easiest way to get data displayed on the dashboard is simply to use methods in the SmartDashboard class. For example to write a number to Shuffleboard write:

Java

```
SmartDashboard.putNumber("Joystick X value", joystick1.getX());
```

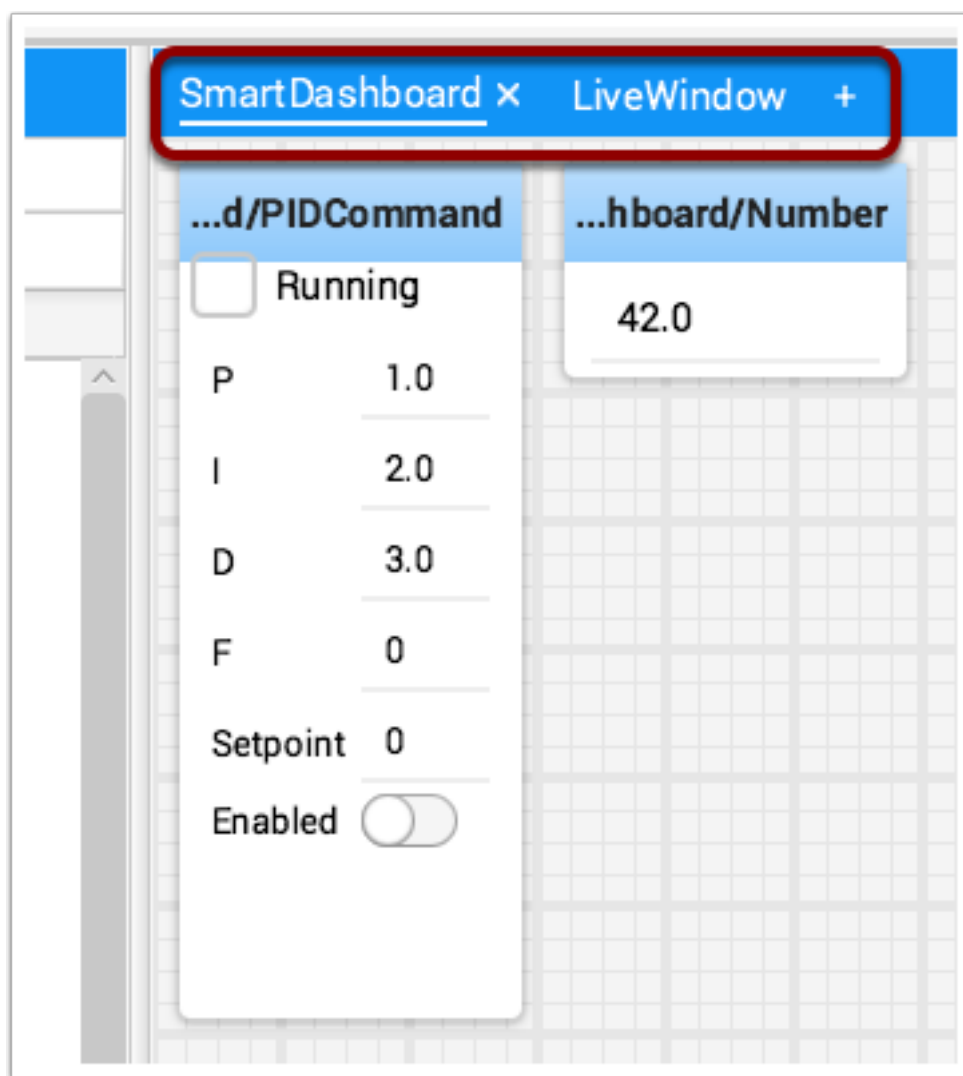
to see a field displayed with the label "Joystick X value" and a value of the X value of the joystick. Each time this line of code is executed, a new joystick value will be sent to Shuffleboard. Remember: you must write the joystick value whenever you want to see an updated value. Executing this line once at the start of the program will only display the value once at the time the line of code was executed.



19.1.2 Displaying data from your robot

Your robot can display data in regular operating modes like Teleop and Autonomous modes but you can also display the status and operate all the robot subsystems when the robot is switched to Test mode. By default you'll see two tabs when you start Shuffleboard, one for Teleop/Autonomous and another for Test mode. The currently selected tab is underlined as can be seen in the picture below.

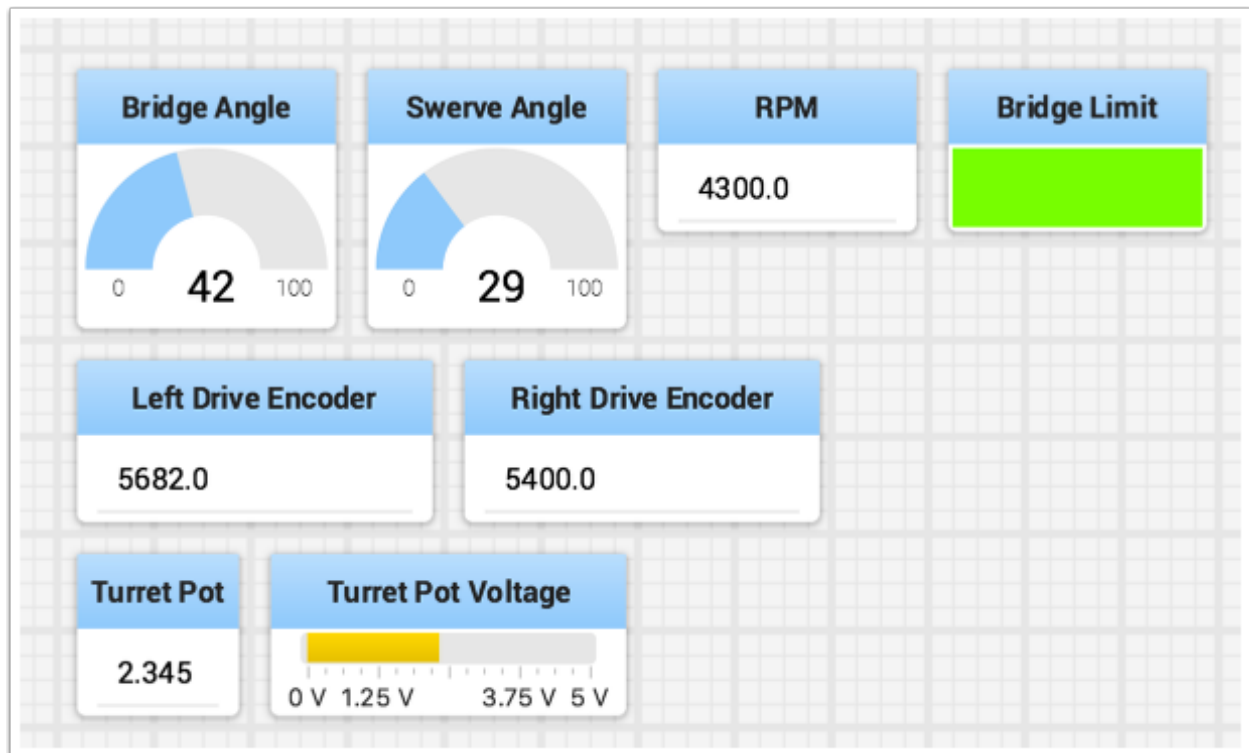
Often debugging or monitoring the status of a robot involves writing a number of values to the console and watching them stream by. With Shuffleboard you can put values to a GUI that is automatically constructed based on your program. As values are updated, the corresponding GUI element changes value - there is no need to try to catch numbers streaming by on the screen.



Displaying values in normal operating mode (autonomous or teleop)

Java

```
protected void execute() {
    SmartDashboard.putBoolean("Bridge Limit", bridgeTipper.atBridge());
    SmartDashboard.putNumber("Bridge Angle", bridgeTipper.getPosition());
    SmartDashboard.putNumber("Swerve Angle", drivetrain.getSwerveAngle());
    SmartDashboard.putNumber("Left Drive Encoder", drivetrain.getLeftEncoder());
    SmartDashboard.putNumber("Right Drive Encoder", drivetrain.getRightEncoder());
    SmartDashboard.putNumber("Turret Pot", turret.getCurrentAngle());
    SmartDashboard.putNumber("Turret Pot Voltage", turret.getAverageVoltage());
    SmartDashboard.putNumber("RPM", shooter.getRPM());
}
```

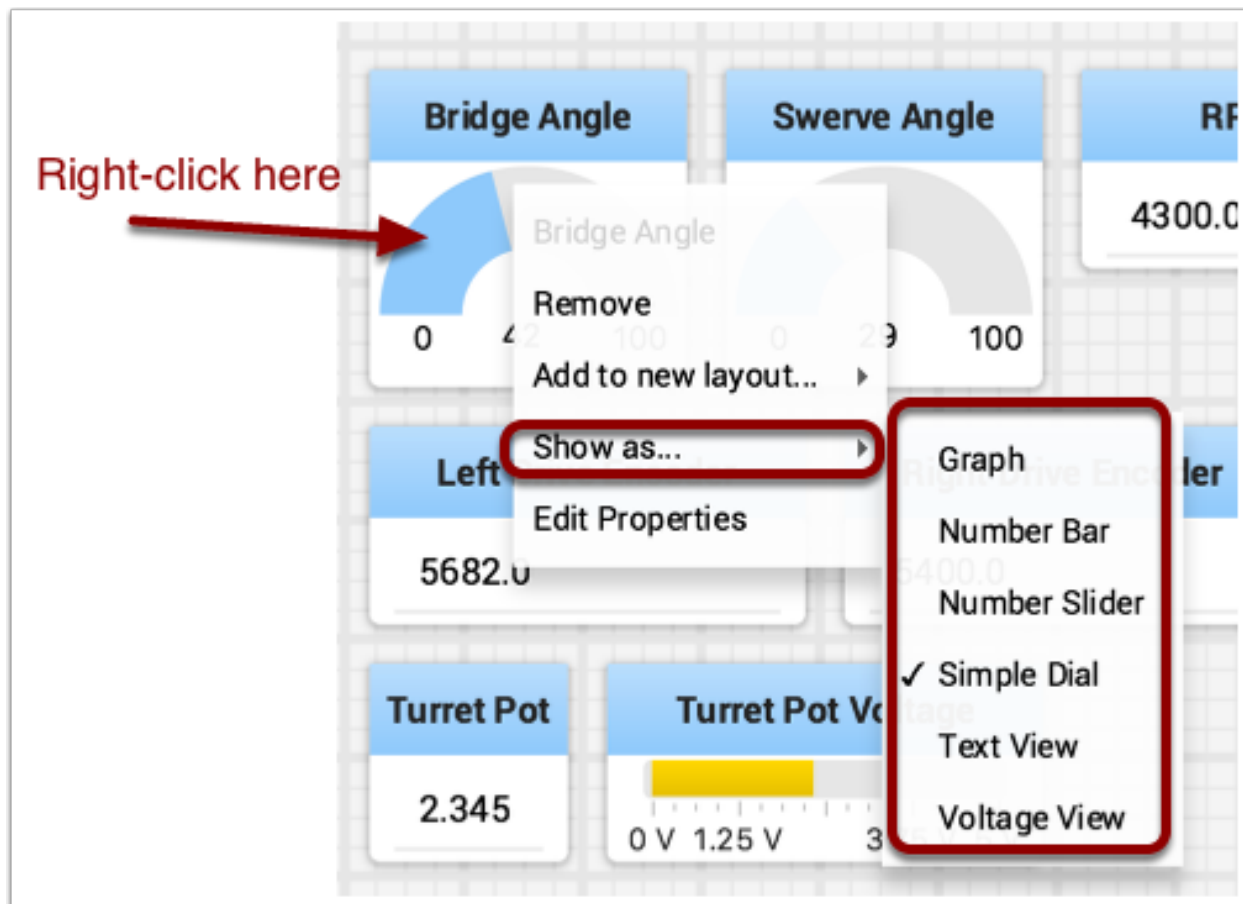


You can write Boolean, Numeric, or String values to Shuffleboard by simply calling the correct method for the type and including the name and the value of the data, no additional code is required.

- Numeric types such as char, int, long, float or double call `SmartDashboard.putNumber("dashboard-name", value)`.
- String types call `SmartDashboard.putString("dashboard-name", value)`
- Boolean types call `SmartDashboard.putBoolean("dashboard-name", value)`

Changing the display type of data

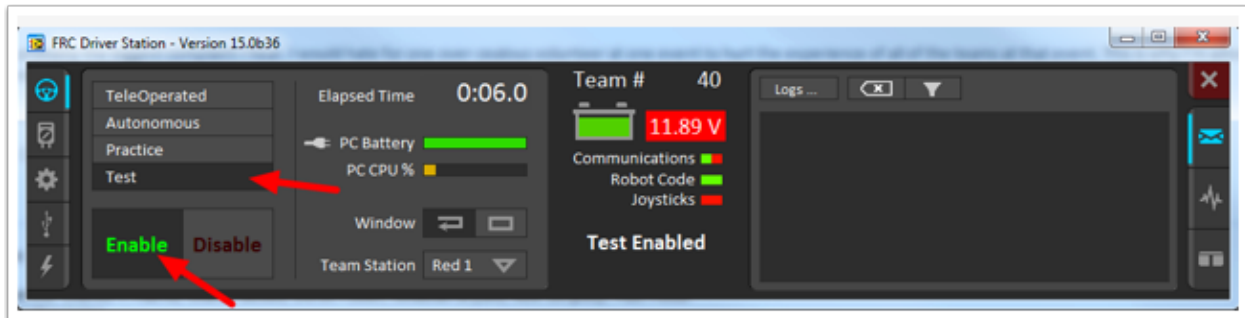
Depending on the data type of the values being sent to Shuffleboard you can often change the display format. In the previous example you can see that number values were displayed as either decimal numbers, a dial to better represent angles, and as a voltage view for the turret potentiometer. To set the display type right-click on the tile and select “Show as...”. You can choose display types from the list in the popup menu.



Displaying data in Test mode

You may add code to your program to display values for your sensors and actuators while the robot is in Test mode. This can be selected from the Driver Station whenever the robot is not on the field. The code to display these values is automatically generated by RobotBuilder or manually added to your program and is described in the next article. Test mode is designed to verify the correct operation of the sensors and actuators on a robot. In addition it can be used for obtaining setpoints from sensors such as potentiometers and for tuning PID loops in your code.

Setting test mode

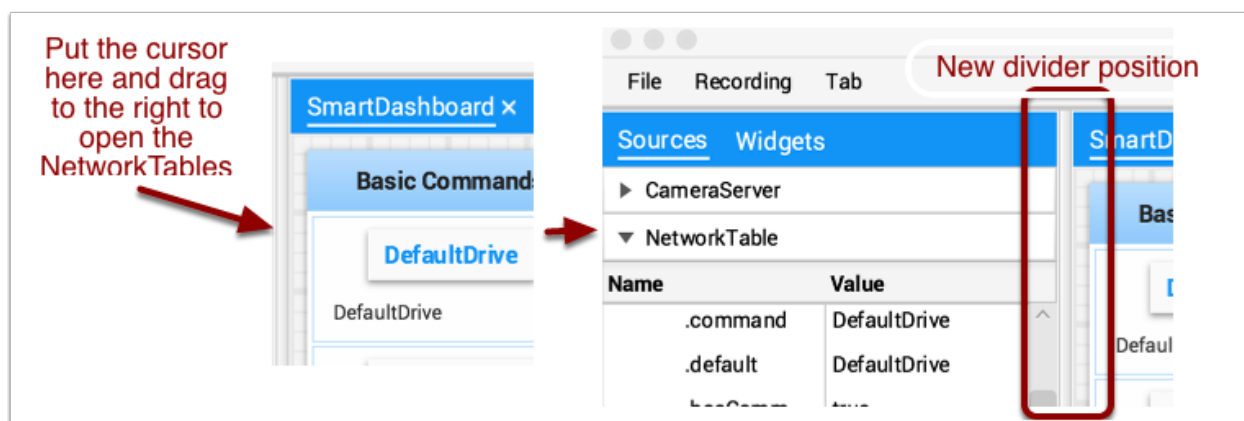
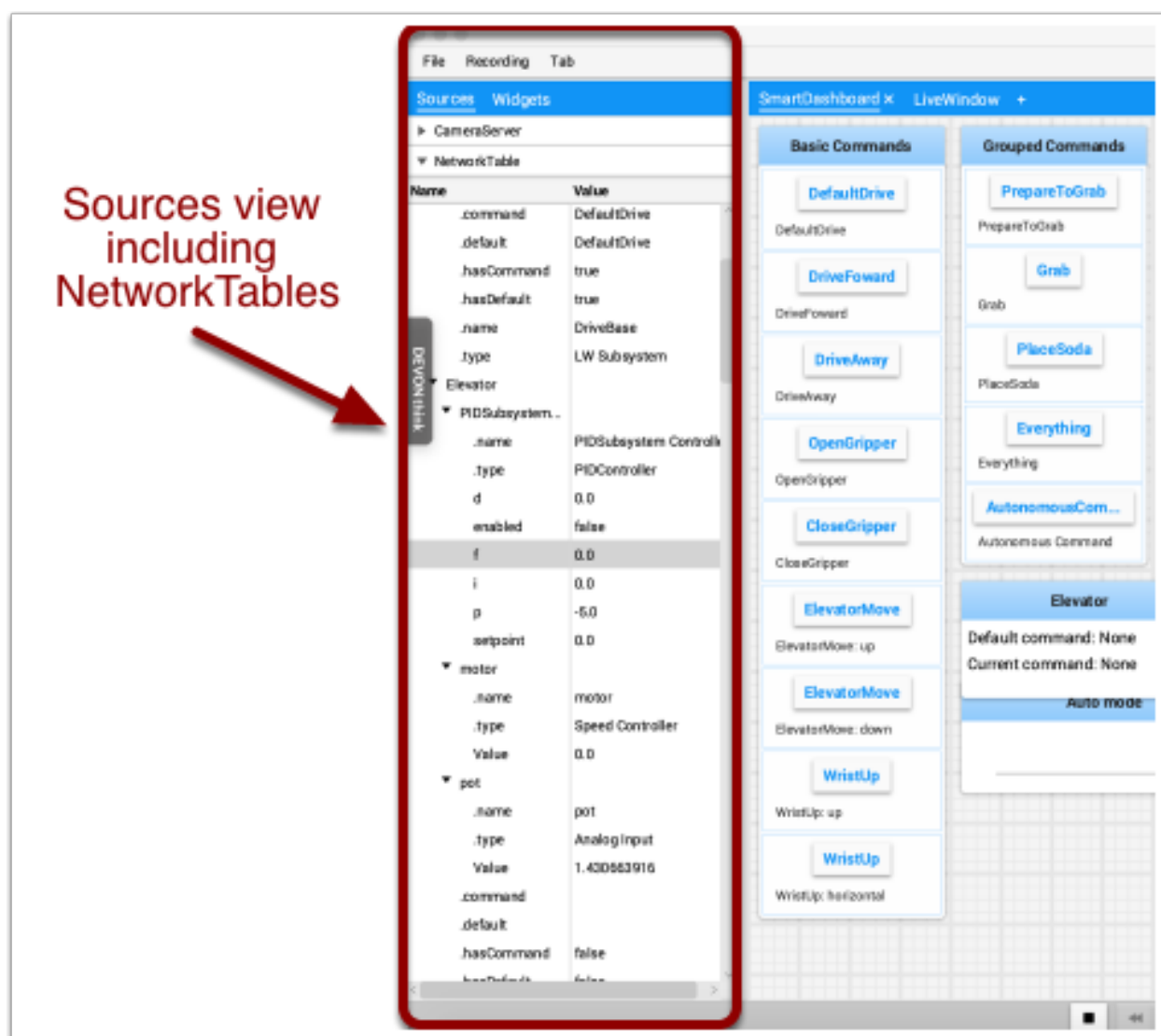


Enable Test Mode in the Driver Station by clicking on the “Test” button and setting “Enable” on the robot. When doing this, Shuffleboard will display the status of any actuators and sensors used by your program organized by subsystem.

Getting data from the Sources view

Normally *NetworkTables* data automatically appears on one of the tabs and you just rearrange and use that data. Sometimes you might want to recover a value that was accidentally deleted from the tab or display a value that is not part of the SmartDashboard / NetworkTables key. For these cases the values can be dragged onto a pane from NetworkTables view under Sources on the left side of the window. Choose the value that you want to display and just drag it to the pane and it will be automatically created with the default type of widget for the data type.

Note: Sometimes the Sources view is not visible on the left - it is possible to drag the divider between the tabbed panes and the Sources so the sources is not visible. If this happens move the cursor over the left edge and look for a divider resizing cursor, then left click and drag out the view. In the two images below you can see where to click and drag, and when finished the divider is as shown in the second image.

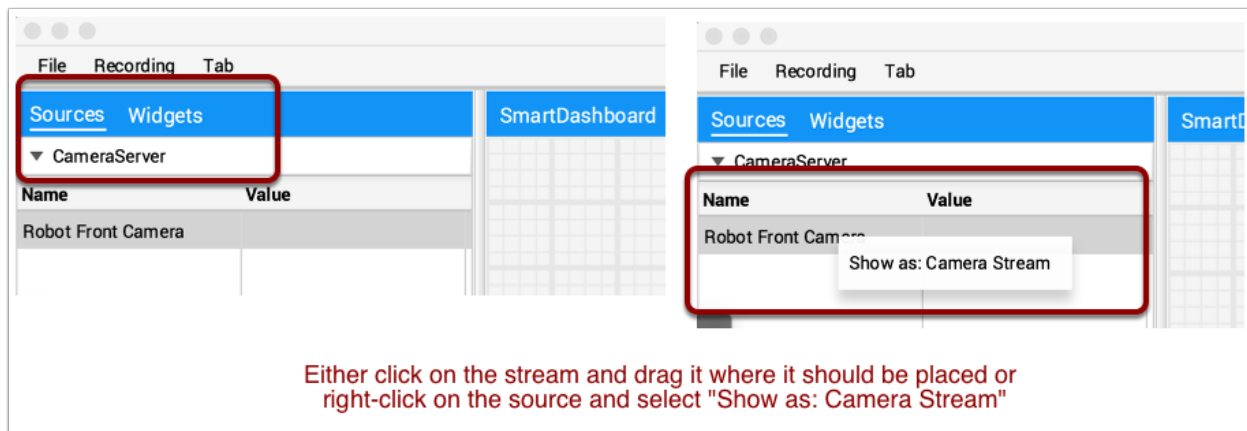


19.1.3 Displaying Camera Streams

Camera streams from the robot can be viewed on a tab in Shuffleboard. This is useful for viewing what the robot is seeing to give a less obstructed view for operators or helping visualize the output from a vision algorithm running on the driver station computer or a coprocessor on the robot. Any stream that is running using the CameraServer API can be viewed in a camera stream widget.

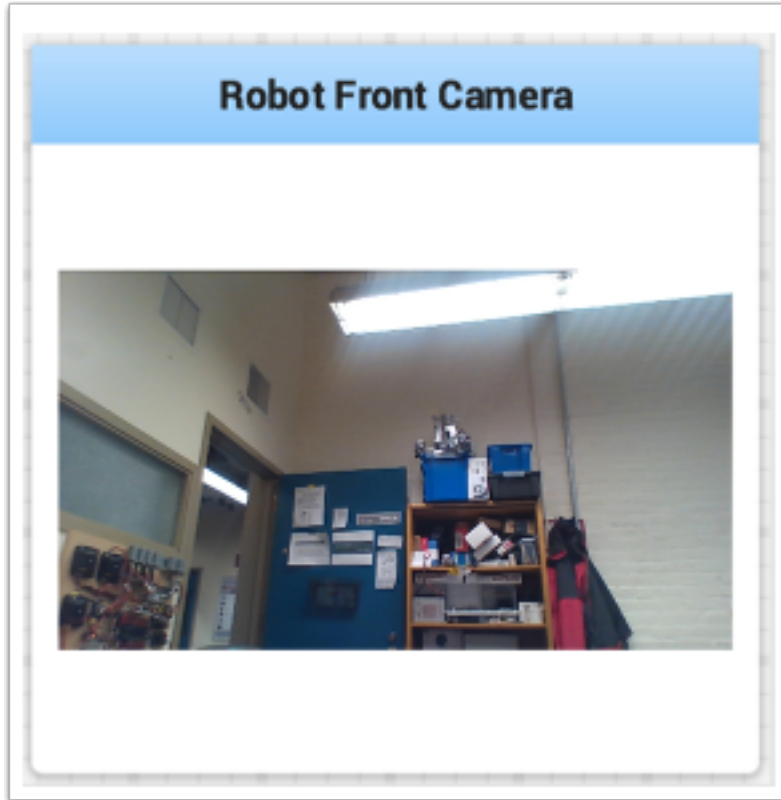
Adding a Camera Stream

To add a camera to your dashboard select “Sources” and view the “CameraServer” source in the left side panel in the Shuffleboard window as shown in the example below. A list of camera streams will be shown, in this case there is only one camera called “Robot Front Camera”. Drag that to the tab where it should be displayed. Alternatively the stream can also be placed on the dashboard by right-clicking on the stream in the Sources list and selecting “Show as: Camera Stream”.



Once the camera stream is added it will be displayed in the window. It can be resized and moved where you would like it.

Note: Be aware that sending too much data from too high a resolution or too high a frame rate will cause high CPU usage on both the roboRIO and the laptop.

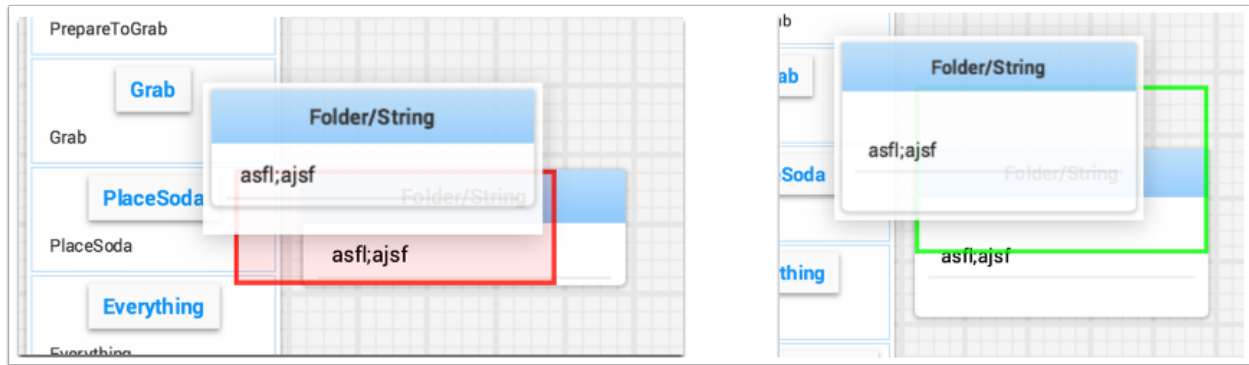


19.1.4 Working with widgets

The visual displays that you manipulate on the screen in Shuffleboard are called widgets. Widgets are generally automatically displayed from values that the robot program publishes with NetworkTables.

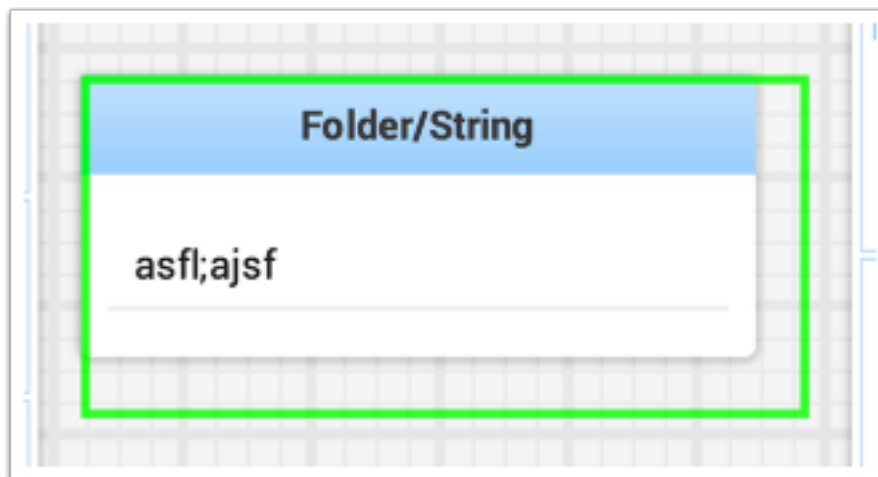
Moving widgets

Widgets can be moved simply with drag and drop. Just move the cursor over the widget, left-click and drag it to the new position. When dragging you can only place widgets on grid squares and the size of the grid will effect the resolution of your display. When dragging a red or green outline will be displayed. Green generally means that there is enough room at the current location to drop the widget and red generally means that it will overlap or be too big to drop. In the example below a widget is being moved to a location where it doesn't fit.



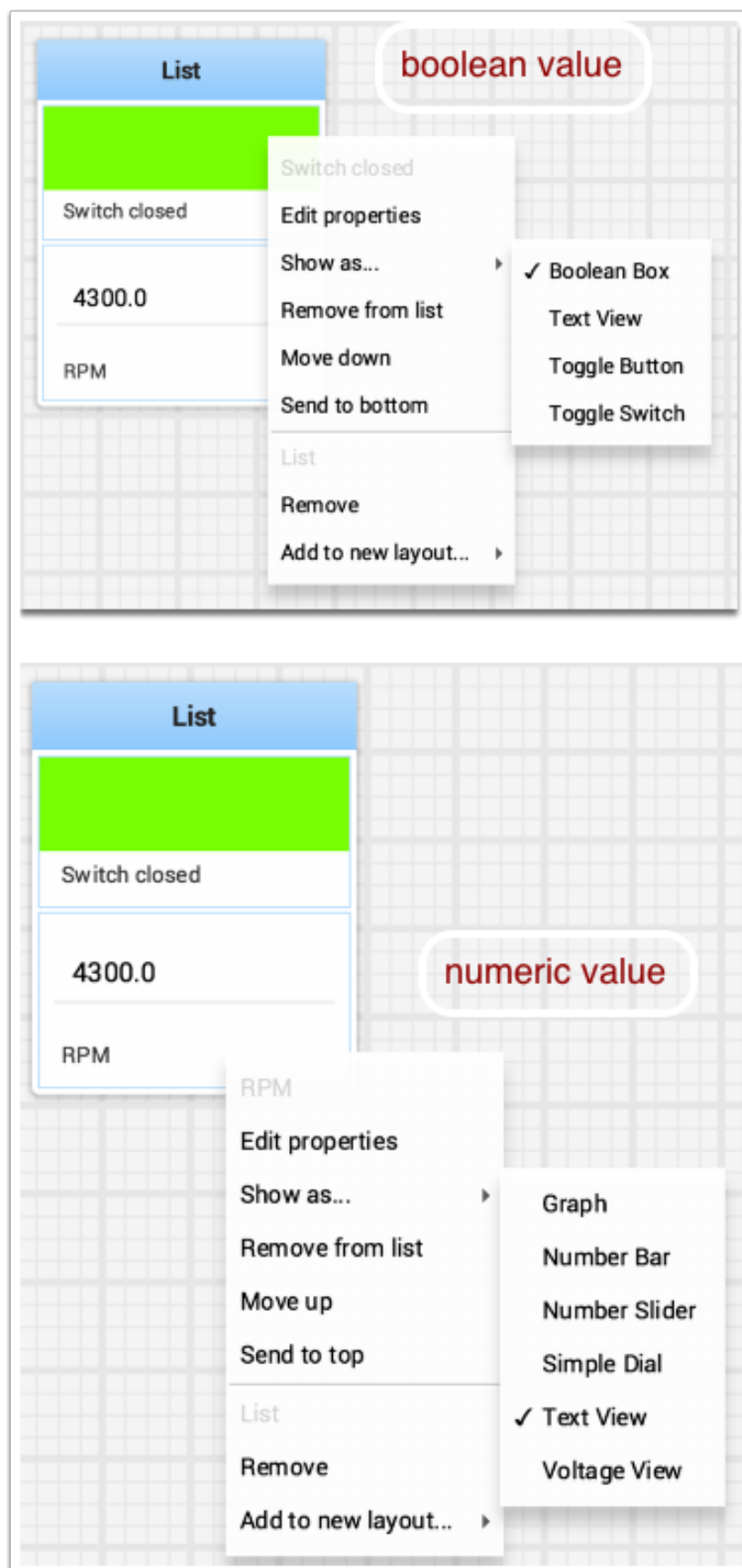
Resizing widgets

Widgets can be resized by clicking and dragging the edge or corner of the widget image. The cursor will change to a resize-cursor when it is in the right position to resize the widget. As with moving widgets, a green or red outline will be drawn indicating that the widget can be resized or not. The example below shows a widget being resized to a larger area with the green outline indicating that there is no overlap with surrounding widgets.



Changing the display type of widgets

Shuffleboard is very rich in display types depending on the data published from the robot. It will automatically choose a default display type, but you might want to change it depending on the application. To see what the possible displays are for any widget, right-click on the widget and select the "Show as..." and from the popup menu, choose the desired type. In the example below are two data values, one a number and the other a boolean. You can see the different types of display options that are available to each. The boolean value has only two possible values (true/false) it can be shown as a boolean box (the red/green color), or text, or a toggle button or toggle switch. The number value can be displayed as a graph, number bar, number slider, dial, text, or a voltage view depending on the context of the value.

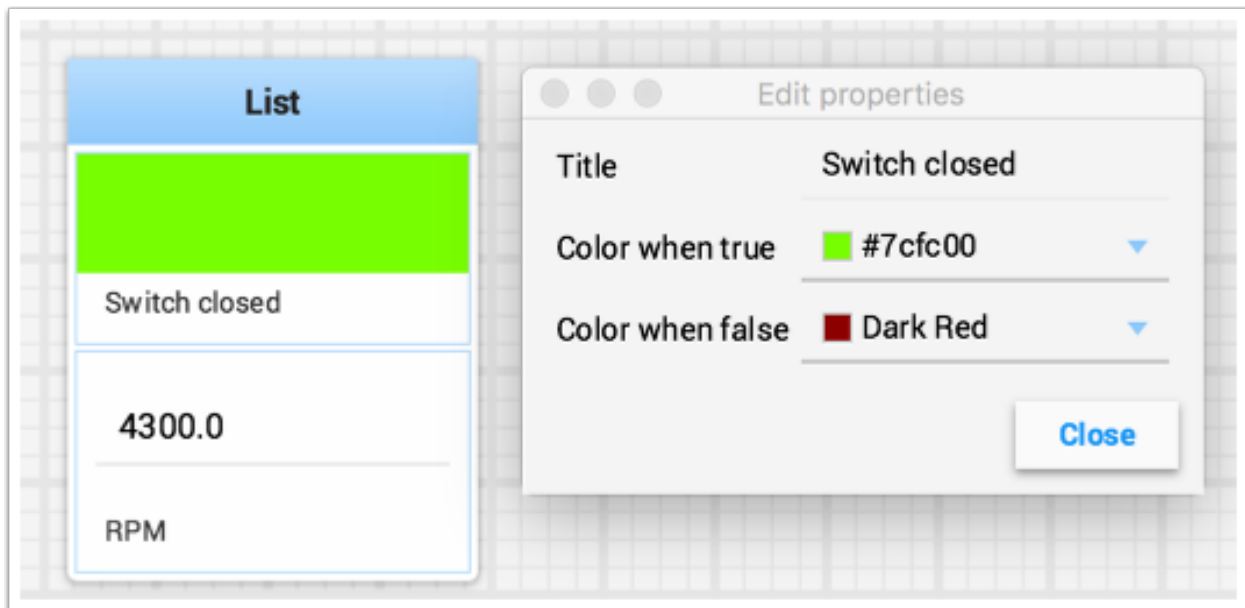


Changing the title of widgets

You can change the title of widgets by double-clicking in their title bar and editing the title to the new value. If a widget is contained in a layout, then right-click on the widget and select the properties. From there you can change the widget title that is displayed.

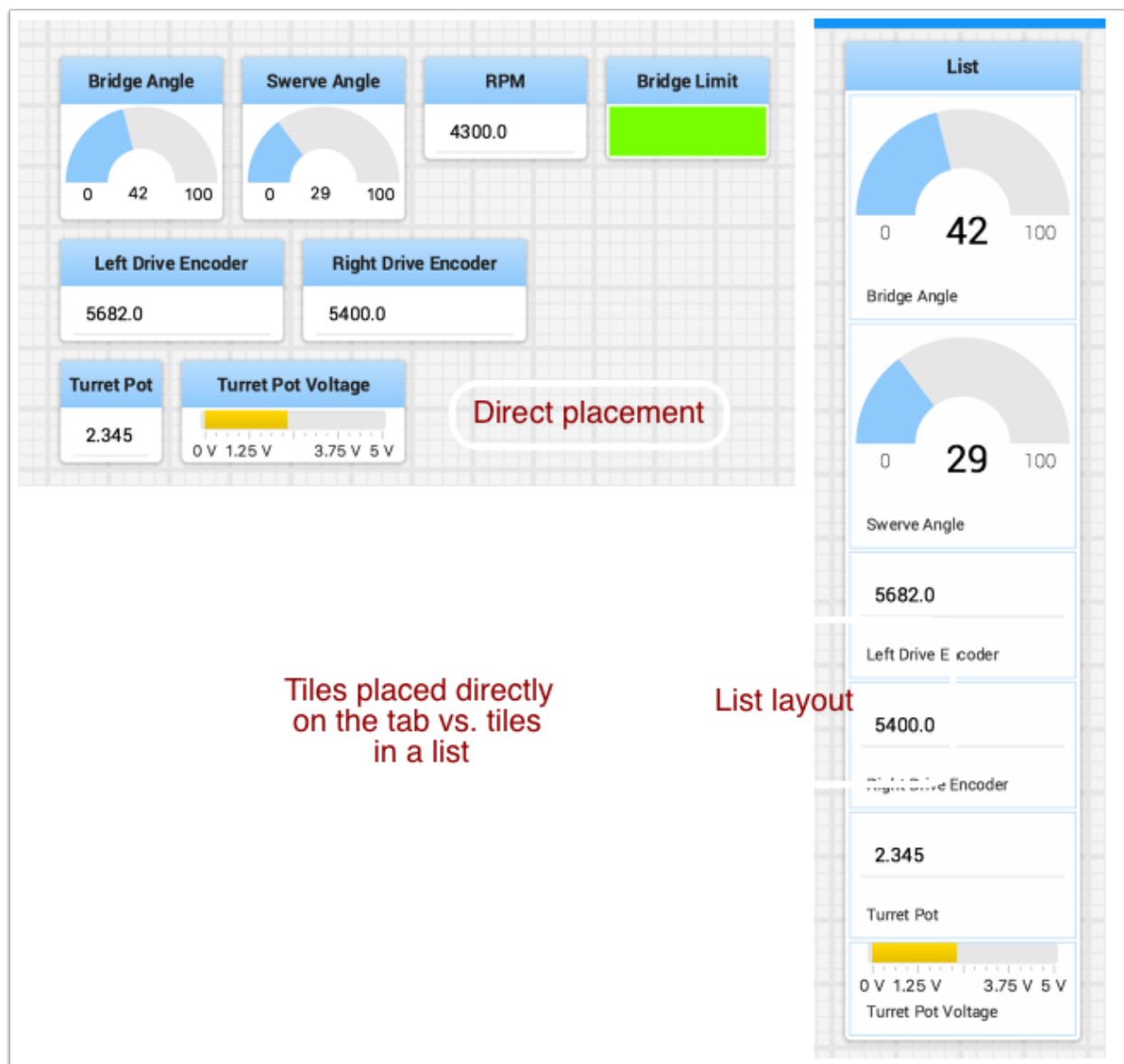
Changing widget properties

You can change the appearance of a widget such as the range of values represented, colors or some other visual element. In cases where this is possible right-click on the widget and select “Edit properties” from the popup menu. In this boolean value widget shown below, the widget title, true color and false color can all be edited.

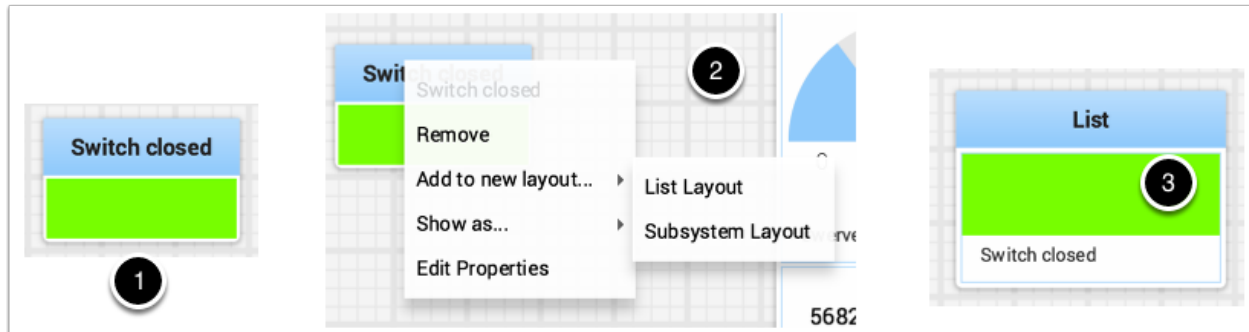


19.1.5 Working with Lists

Lists are sets of tiles that are be logically grouped together in a vertical layout. When tiles are added to a list, it becomes visually obvious that the tiles are related in function. In addition tiles in lists take up less screen space than the same tiles directly on the desktop because tiles in a list don't display the dark window title, but instead display a smaller textual label documenting the contents of the list.



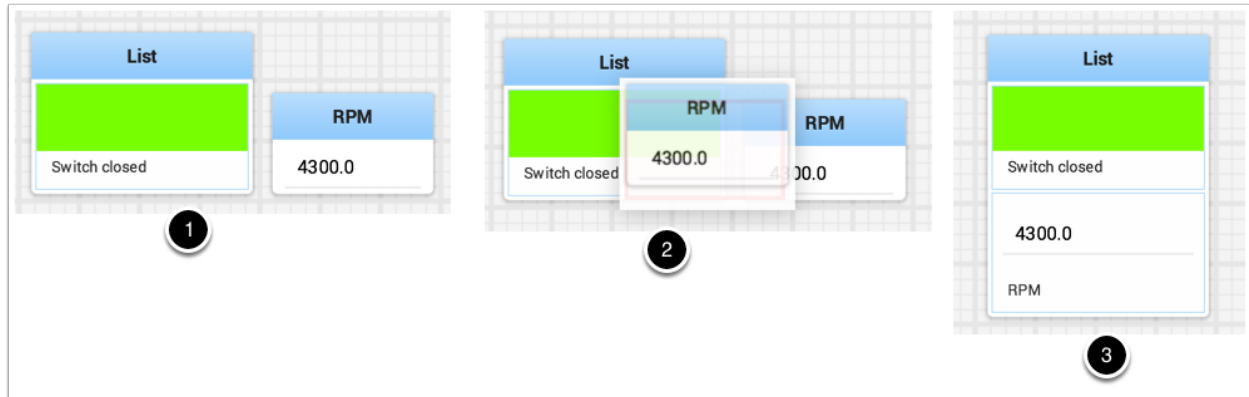
Creating a list



A list can be created by: 1. Right-clicking on the first tile that should go into the list. 2. Select the "Add to new layout..." option from the popup menu. 3. A new list will be created called "List" and the tile will be at the top of it. Notice that tiles in lists do not have the window title at the top, but instead have the text that was in the window title.

Adding tiles to a list

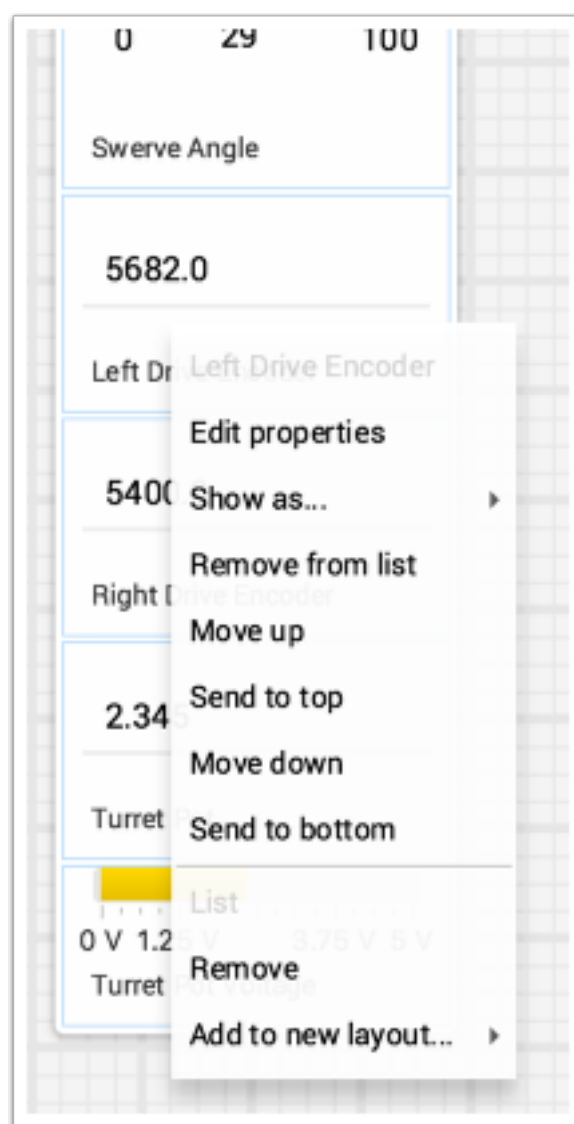
Add tiles to a list layout by: 1. identify the list and the tile to be added. 2. Drag the new tile onto the list. 3. The tile will be added to the list. If there is no room as the list is currently sized, the tile will be added off the end of the list and a vertical scrollbar will be added if it's not already there.



Rearranging tiles in a list

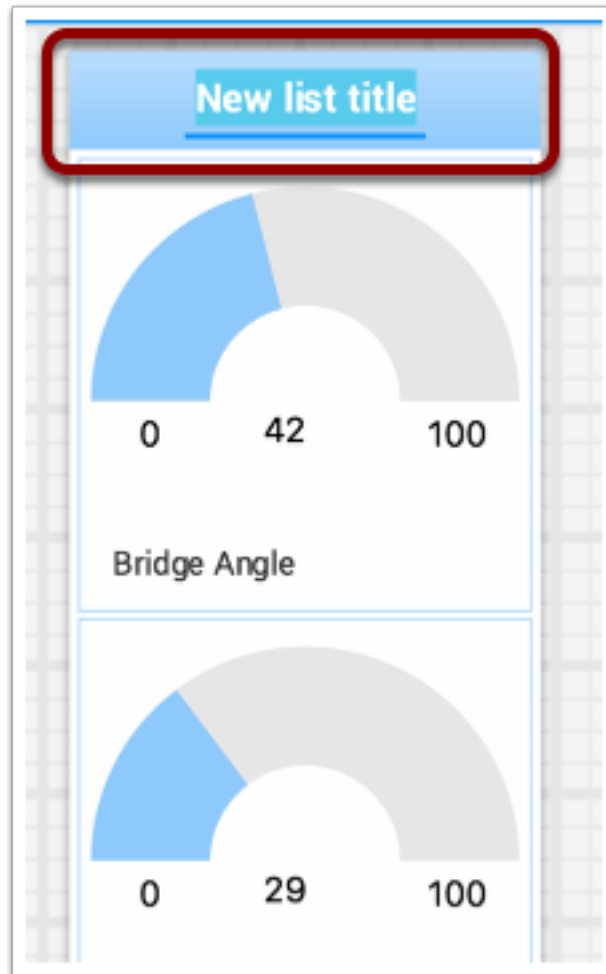
Tiles in a list can be rearranged by right-clicking on the tile to be moved and selecting:

1. "Move up" to move the tile before the previous tile
2. "Move down" to move the tile after the next tile
3. "Send to top" to move the tile to the top of the list
4. "Send to bottom" to move the tile to the bottom of the list
5. "Remove from list" to delete the tile from the list.



Renaming a list

You can rename a list by double-clicking on the list title and changing the name. Click outside the title to save the changes

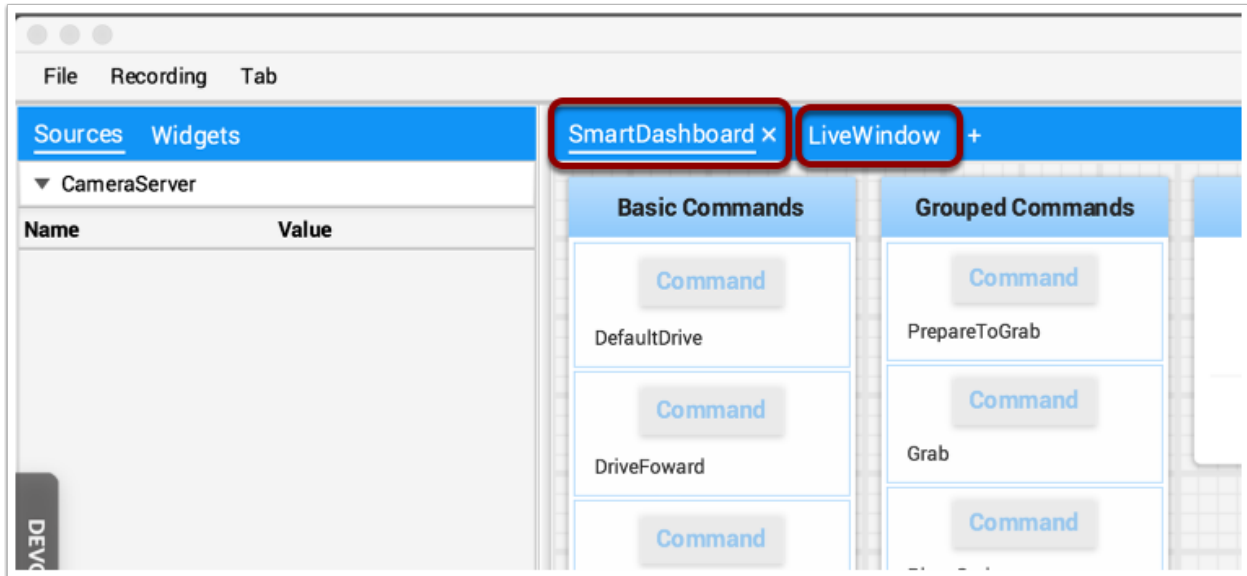


19.1.6 Creating and manipulating tabs

The tabbed layout the Shuffleboard uses help separate different “views” of your robot data and make the displays more useful. You might have a tab the has the display for helping debug the robot program and a different tab for use in competitions. There are a number of options that make tabs very powerful. You can control which data from NetworkTables or other sources appears in each of your tabs using the auto-populate options described later in this article.

Default tabs

When you open Shuffleboard for the first time there are two tabs, labeled SmartDashboard and LiveWindow. These correspond to the two views that SmartDashboard had depending on whether your robot is running in Autonomous/Teleop or Test mode. In shuffleboard both of these views are available any time.



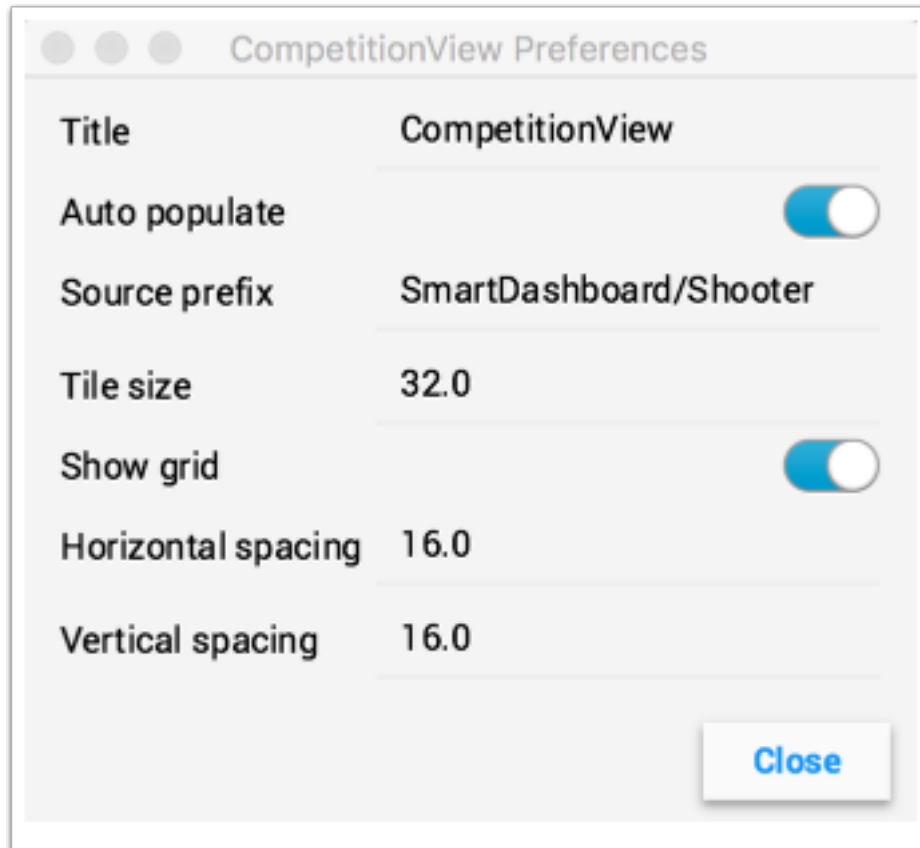
On the SmartDashboard tab all the values that are written using the SmartDashboard.putType() set of methods. On the LiveWindow tab all the autogenerated debugging values are shown.

Switching between tabs

You can switch between tabs clicking on the tab label at the top of the window. In the case above, simply click on SmartDashboard or LiveWindow to see the values that are associated with each tab.

Adding tabs and deleting tabs

You can add additional tabs by clicking on the plus(+) symbol just to the right of the last tab. Once you create a new tab you can set the label by double-clicking on the label in the tab and editing it. You can also right-click on the tab or use the Tab menu to bring up the tab preferences and from that window you can change the name by editing the Title field.



Setting the tab to auto-populate

One of the most powerful features with tabs is to have them automatically populate new values based on a source prefix that is supplied in the tab Preferences pane. In the above example the Preferences pane has a Source prefix of “SmartDashboard/Shooter” and Auto populate is turned on. Any values that are written using the SmartDashboard class that specifies a sub-key of Shooter will automatically appear on that tab. Note: keys that match more than one Source prefix will appear in both tabs. Because those keys also start with SmartDashboard/ and that’s the Source prefix for the default SmartDashboard tab, those widgets will appear in both panes. To only have values appear in one pane, you can use NetworkTables to write labels and values and use a different path that is not under SmartDashboard. Alternatively you could let everything appear in the SmartDashboard tab making it very cluttered, but have specific tabs for your needs that will be better filtered.

Using the tab grid and spacing

Each tab can have it’s own Tile size (number of pixels per large square). So some tabs might have coarser resolution for easier layout and others might have a fine grid. The Tile size in the Tab preferences overrides any global settings in the Shuffleboard preferences. In addition, you can specify the padding between the drawing in the widget and the edge of the of the widget. If you program user interfaces these parameters are usually referred to as horizontal and vertical gap (hgap, vgap).

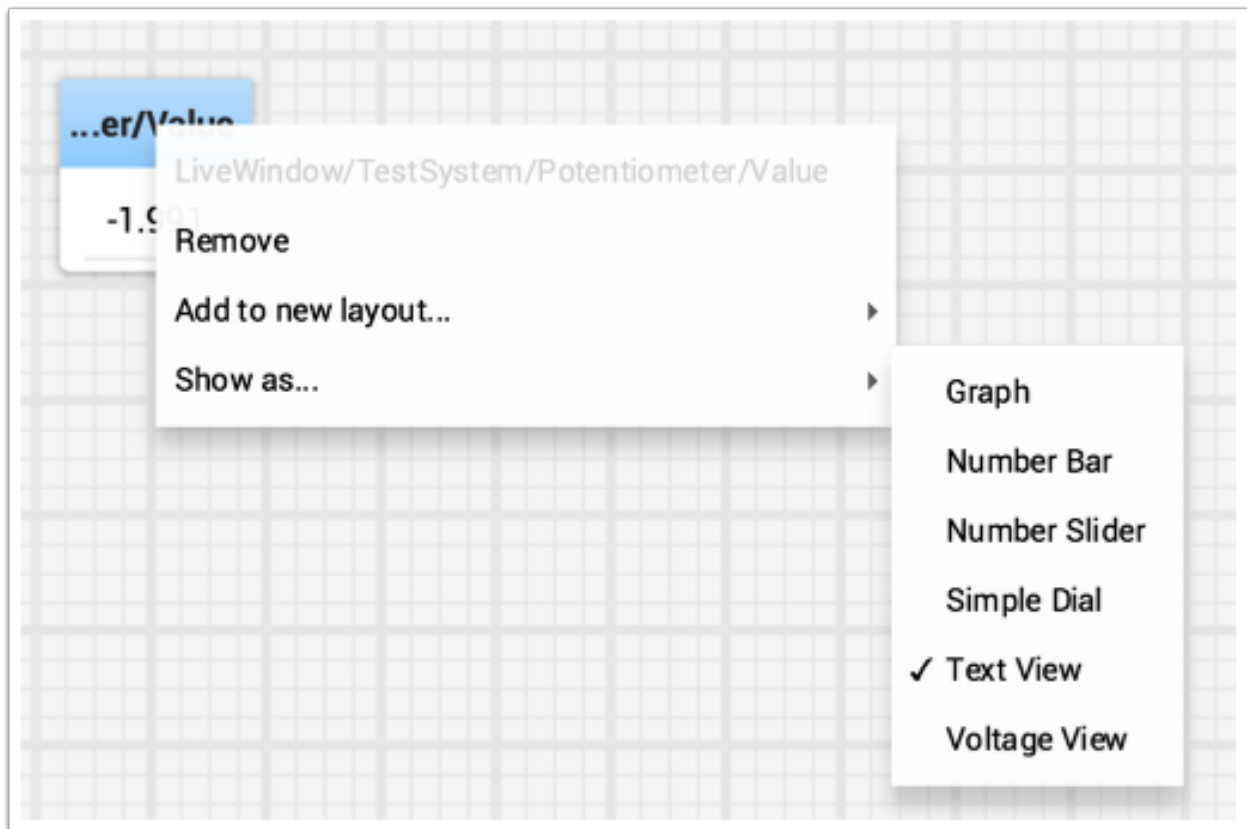
Moving widgets between tabs

Currently there is no way to easily move widgets between tabs without deleting it from one tab and dragging the field from the sources hierarchy on the left into the new pane. We hope to have that capability in a subsequent update soon.

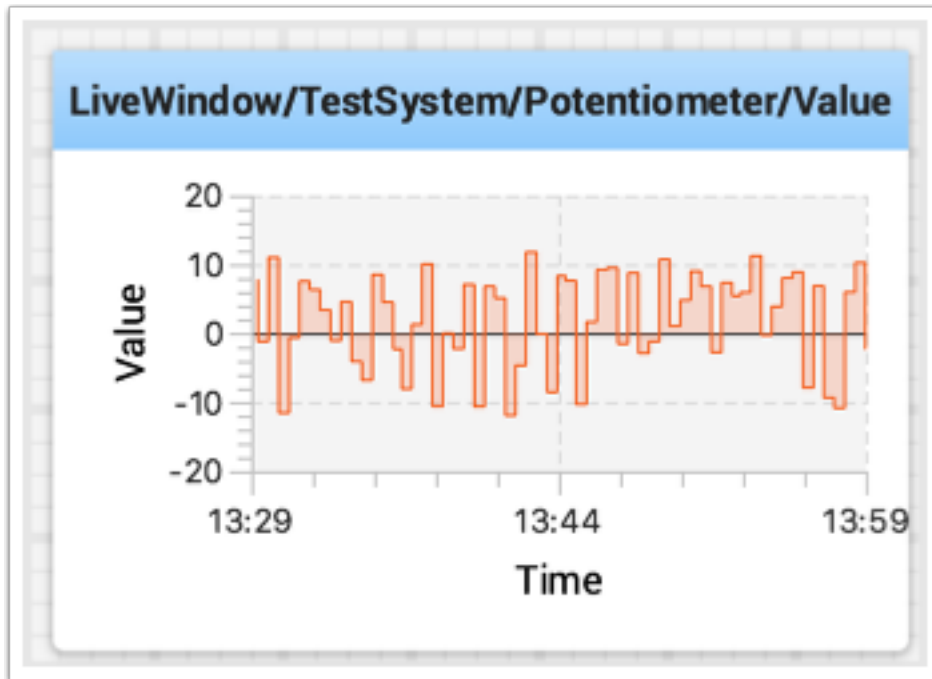
19.1.7 Working with Graphs

With Shuffleboard you can graph numeric values over time. Graphs are very useful to see how sensor or motor values are changing as your robot is operating. For example the sensor value can be graphed in a PID loop to see how it is responding during tuning.

To create a graph, choose a numeric value and right-click in the heading and select “Show as...” and then choose graph

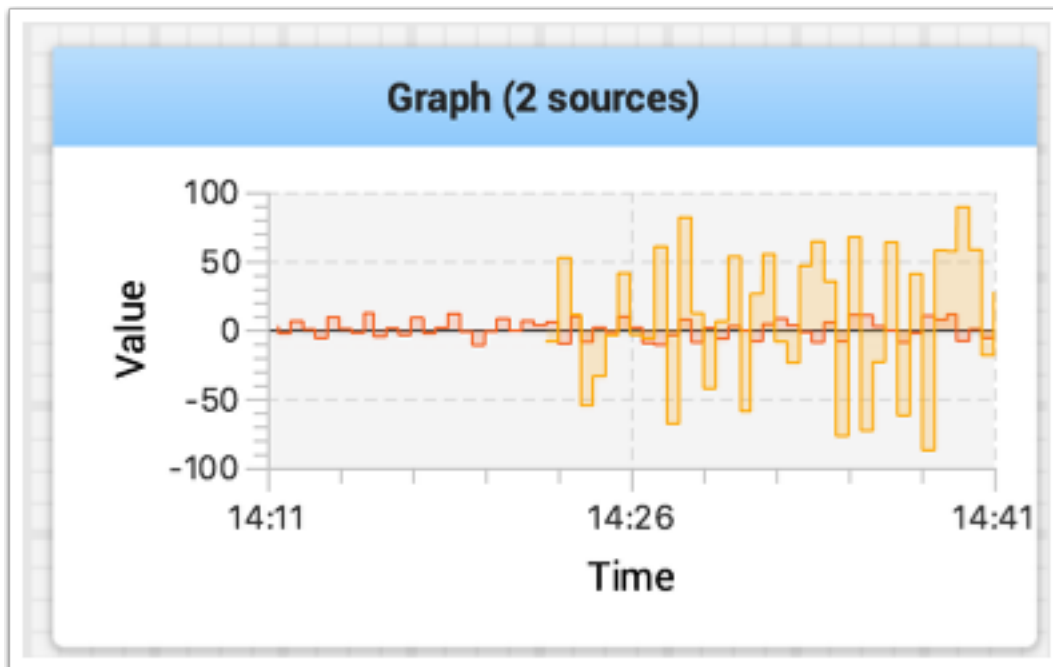


The graph widget shows line plots of the value that you selected. It will automatically set the scale and the default time interval that the graph will show will be 30 seconds. You can change that in the setting for the graph (see below).

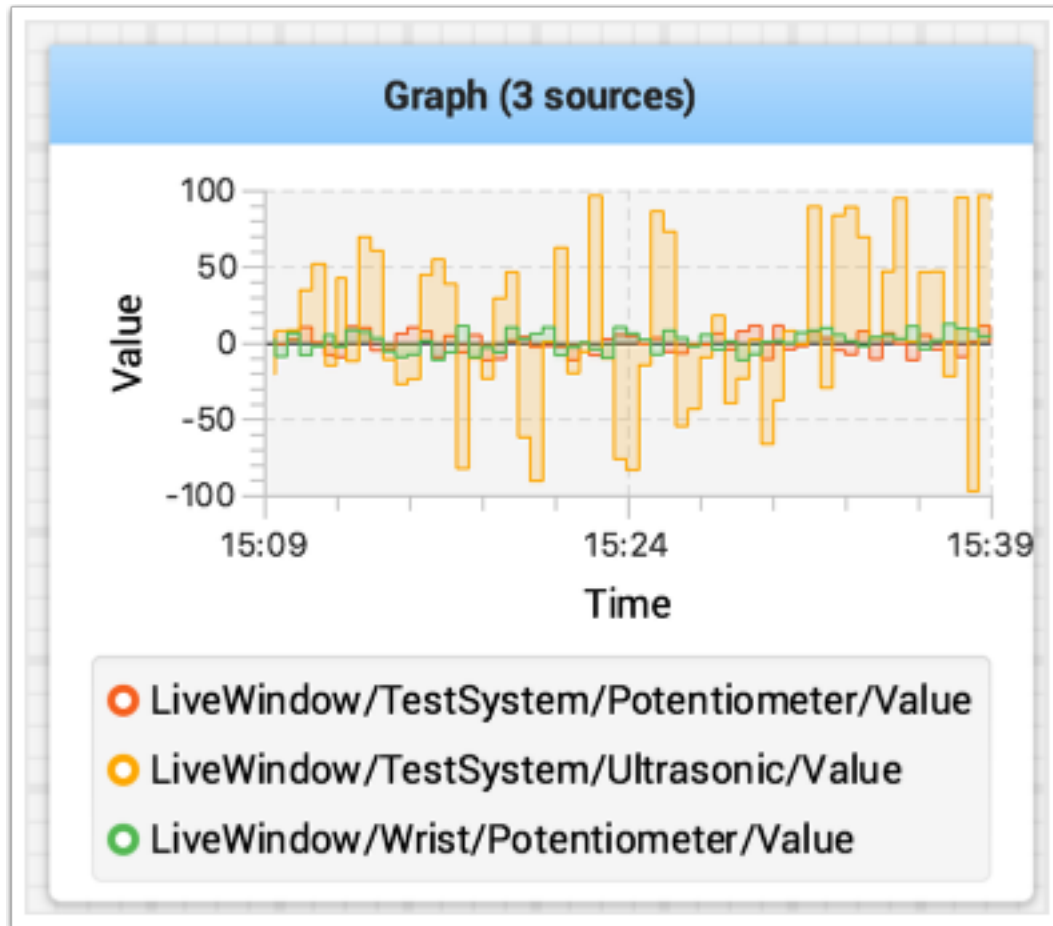


Adding Additional Data Values

For related values it is often desirable to show multiple values on the same graph. To do that, simply drag additional values from the NetworkTables source view (left side of the Shuffleboard window) and drop it onto the graph and that value will be added as shown below. You can continue to drag additional values onto the graph.



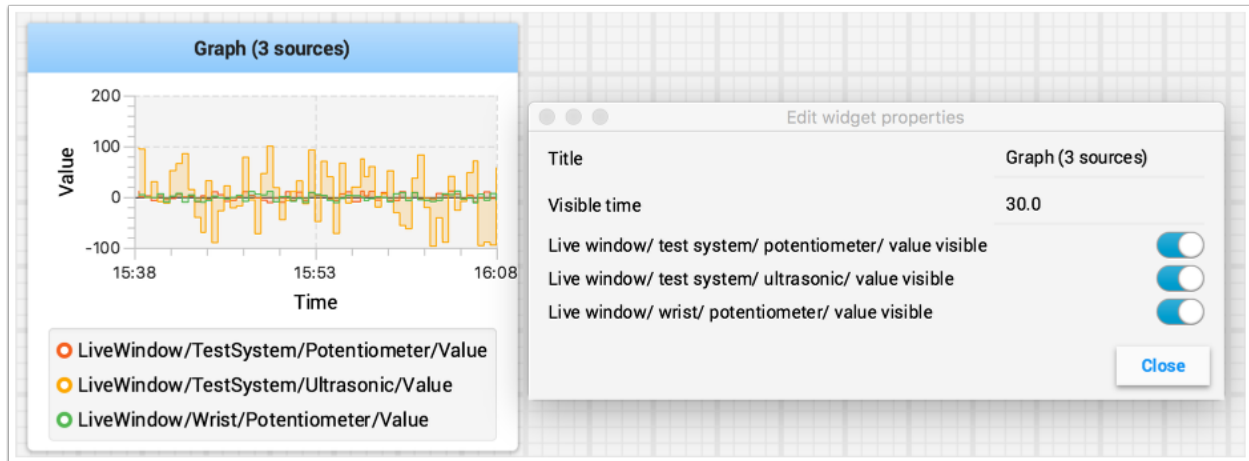
You can resize the graph vertically to view the legend if it is not displayed as shown in the image below. The legend shows all the sources that are used in the plot.



Setting Graph Properties

You can set the number of seconds that are shown in the graph by changing the “Visible time” in the graph widget properties. To access the properties, right-click on the graph and select “Edit properties”.

In addition to setting the visible time the graph can selectively turn sources on and off by turning the switch on and off for each of the sources shown in the properties window (see below).

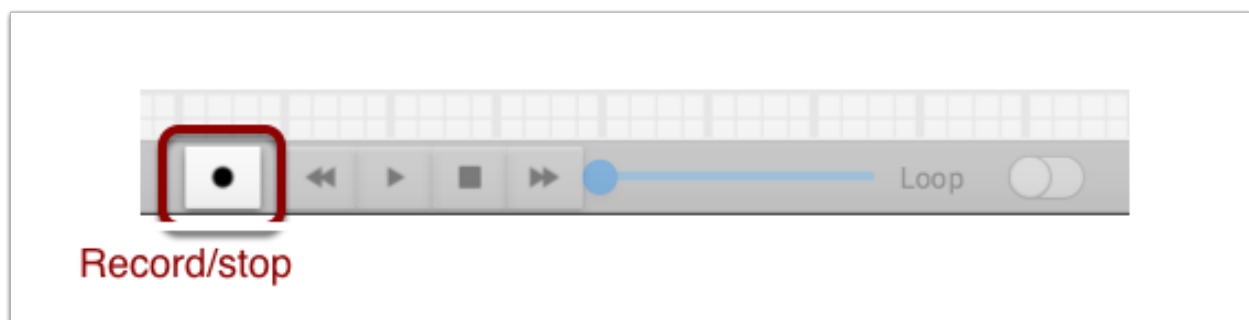


19.1.8 Recording and Playback

Shuffleboard can log all widget updates during a session. Later the log file can be “played back” to see what happened during a match or a practice run. This is especially useful if something doesn’t operate as intended during a match and you want to see what happened. Each recording is captured in a recording file.

Creating a Recording

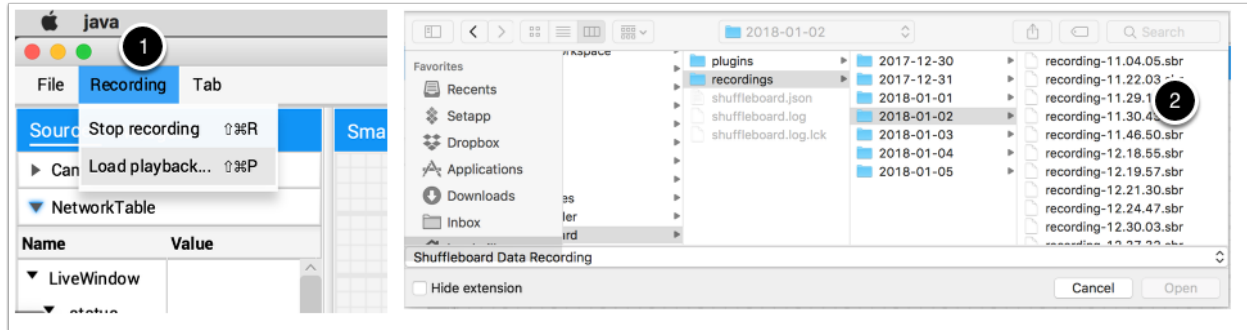
When shuffleboard starts it begins recording to all the NetworkTables values are recorded and continues until stopped by hitting the record/stop button in the recorder controls as shown below. If a new recording is desired, such as when a new piece of code or mechanical system is being tested, stop the current recording if it is running, and click the record button. Click the button again to stop recording and close the recording file. If the button is round (as shown) then click it to start a recording. If the button is a square, then a recording is currently running so click it to stop the recording.



Playing a Recording

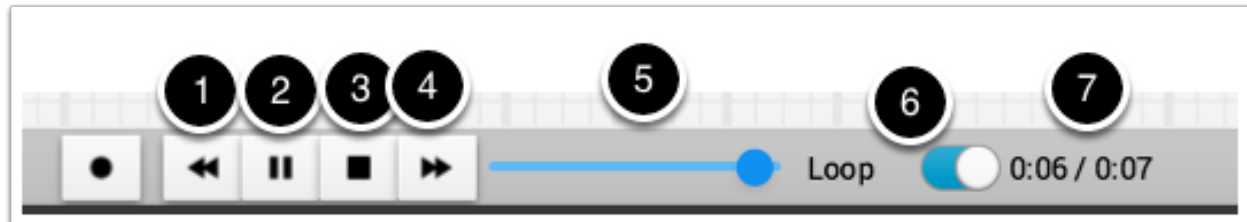
Previous recordings can be played back by:

1. Selecting the “Recording” menu then click “Load playback”.
2. Choose a recording from the from the directory shown. Recordings are grouped by date and the file names are the time the recording was made to help identify the correct one. Select the correct recording from the list.



Controlling the Playback

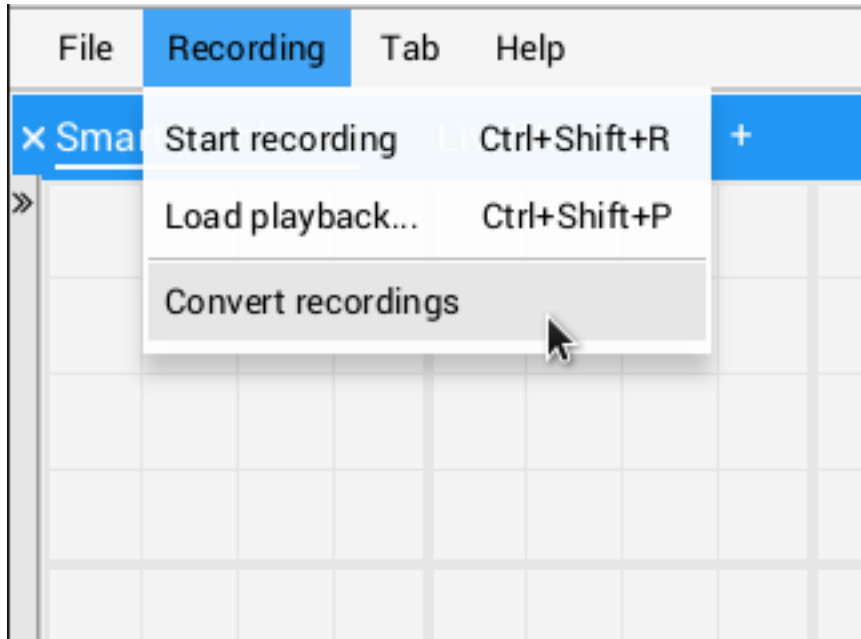
Selecting the recording file will begin playback of that file. While the recording is playing the recording controls will show the current time within the recording as well as the option to loop the recording while watching it. When the recording is being played back the “transport” controls will allow the playback to be controlled.



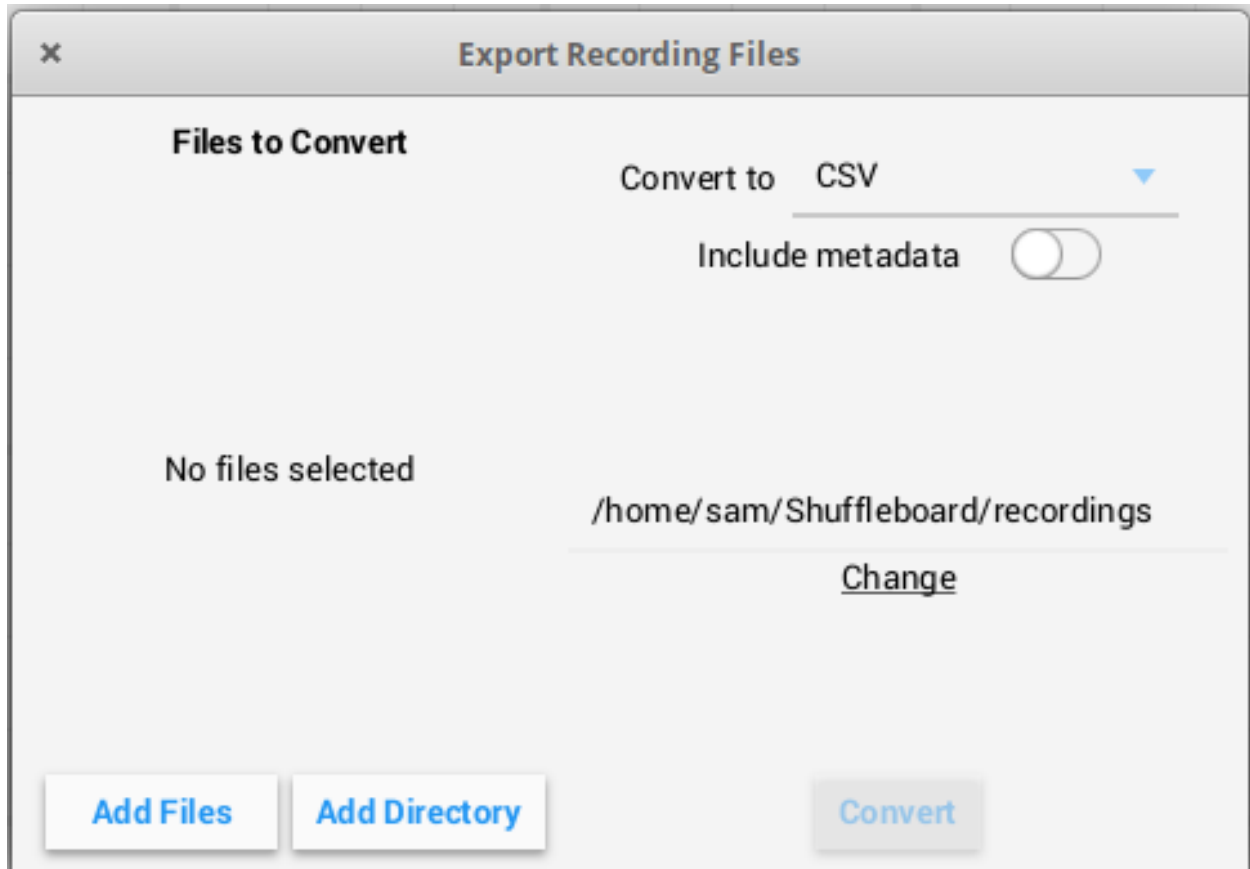
The controls work as follows:

1. The left double-arrow button backs up the playback to the last changed data point
2. The play/pause controls starts and stops the playback
3. The square stop button stops playback and resumes showing current robot values
4. The right double-arrow skips forward to the next changed data value
5. The slider allows for direct positioning to any point in time to view different parts of the recording
6. The loop switch turns on playback looping, that is, the playback will run over and over until stopped
7. The time shows the current point within the recording and the total time of the recording

Converting to Different File Formats



Shuffleboard recordings are in a custom binary format for efficiency. To analyze recorded data without playing it back through the app, Shuffleboard supports data converters to convert the recordings to an arbitrary format. Only a simple CSV converter is shipped with the app, but teams can write custom converters and include them in Shuffleboard plugins.



Multiple recordings can be converted at once. Individual files can be selected with the “Add Files” button, or all recording files in a directory can be selected at once with the “Add Directory” button.

Converted recordings will be generated in the `~/Shuffleboard/recordings` directory, but can be manually selected with the “Change” button.

Different converters can be selected with the dropdown in the top right. By default, only the CSV converter is available. Custom converters from plugins will appear as options in the dropdown.

Additional Notes

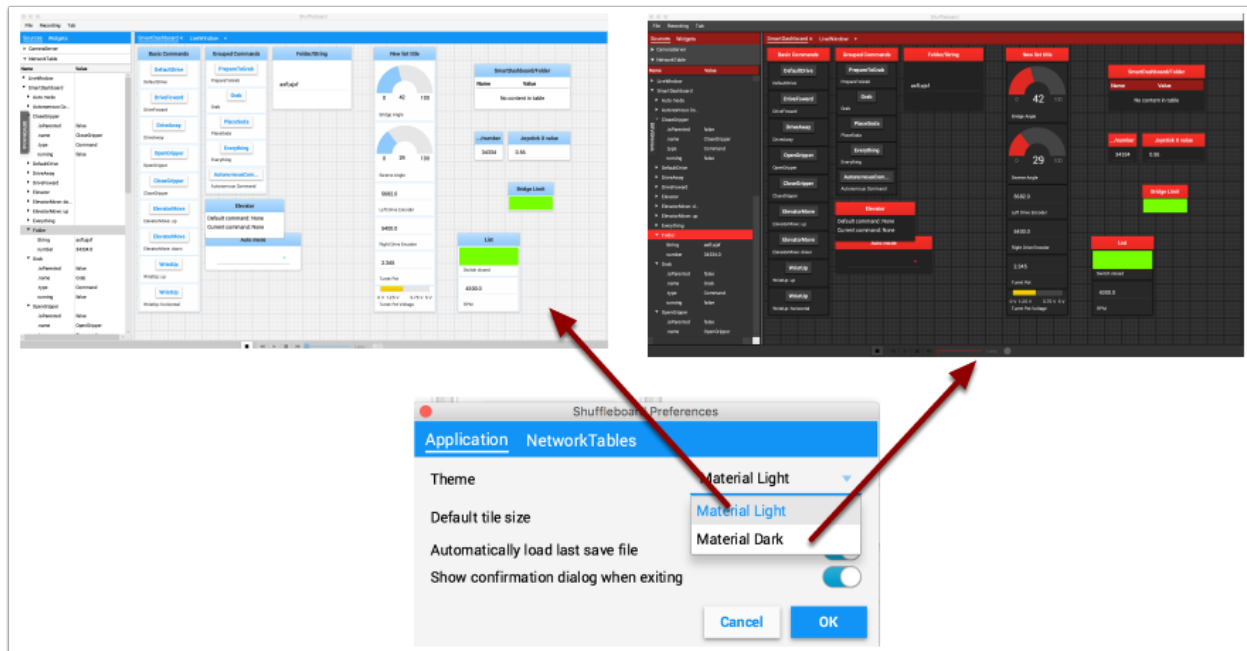
Graphs won’t display properly while scrubbing the timeline but if it is playing through where the graph history can be captured by the graph then they will display as in the original run.

19.1.9 Setting global preferences for Shuffleboard

There are a number of settings that set the way Shuffleboard looks and behaves. Those are on the Shuffleboard Preferences pane that can be accessed from the File menu.

Setting the theme

Shuffleboard supports two themes, Material Dark and Material Light and the setting depends on your preferences. This uses css styles that apply to the entire application and can be changed any time.

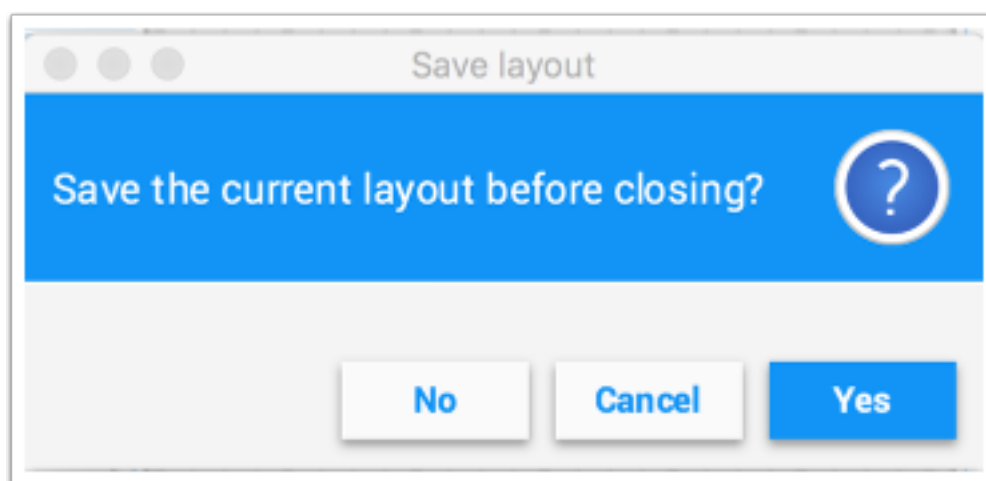
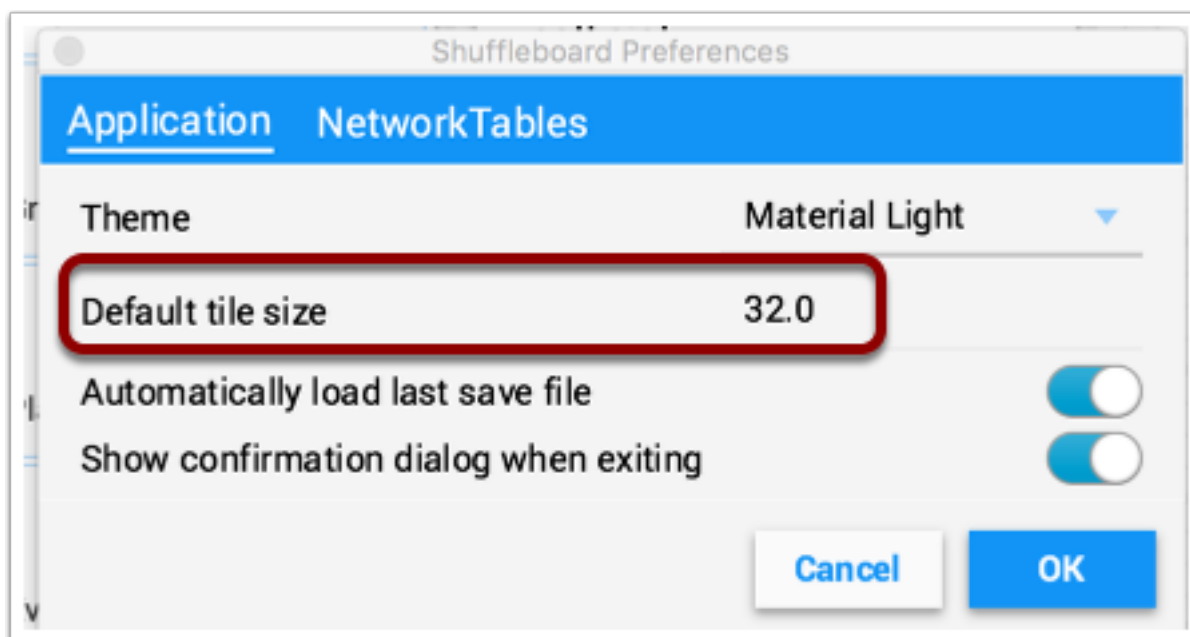


Setting the default tile size

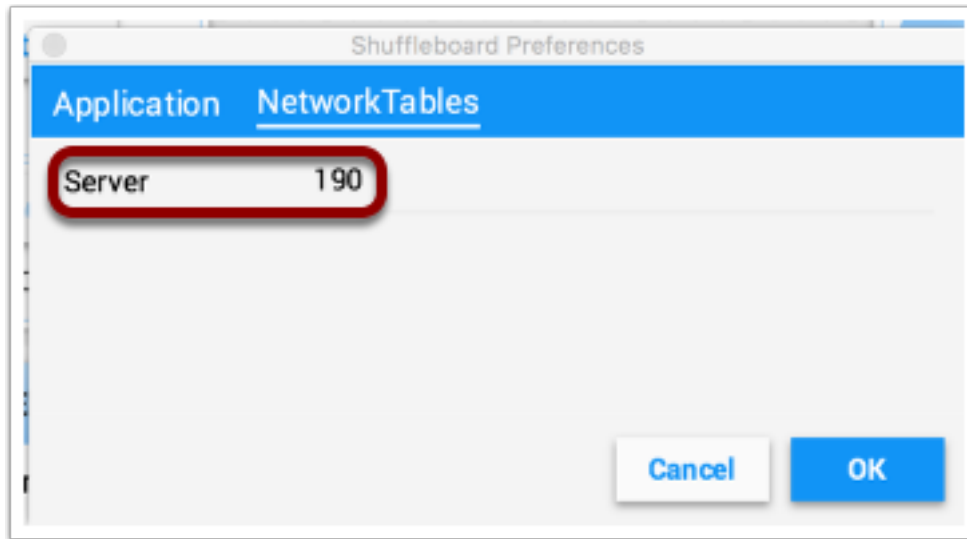
Shuffleboard positions tiles on a grid when you are adding or moving them yourself or when they are auto-populated. You can set the default tile size when for each tab or it can be set globally for all the tabs created after the default setting is changed. Finer resolution in the grid results in finer control over placement of tiles. This can be set in the Shuffleboard Preferences window as shown below.

Working with the layout save files

Shuffleboard will default to saving your layout when you exit the application. You can also save it explicitly using the File / Save and File / Save as... menu options. The preferences window has options to cause the previous layout to be automatically applied when Shuffleboard starts. In addition, Shuffleboard will always display a "Save layout" window to remind you to save the layout. You can choose to turn off the automatic prompt on exit, but be sure to save the layout manually in this case so you don't lose your changes.



Setting the team number



In order for Shuffleboard to be able to find your NetworkTables server on your robot, specify your team number in the “NetworkTables” tab on the Preferences pane. If you’re running Shuffleboard with a running Driver Station, the Server field will be auto-populated with the correct information. If you’re running on a computer without the Driver Station, you can manually enter your team number or the robotRIO network address.

19.1.10 Shuffleboard FAQ, issues, and bugs

Warning: Shuffleboard as well as most of the other control system components were developed with Java 11 and will not work with Java 8. Be sure before reporting problems that your computer has Java 11 installed and is set as the default Java Environment.

Frequently Asked Questions

How do I report issues, bugs or feature requests with Shuffleboard?

Bugs, issues, and feature requests can be added on the Shuffleboard GitHub page by creating an issue. We will try to address them as they are entered into the system. Please try to look at existing issues before creating new ones to make sure you aren’t duplicating something that has already been reported or work that is planned. You can find the issues on the [Shuffleboard GitHub page](#).

How can I add my own widgets or other extensions to Shuffleboard?

The [Shuffleboard wiki](#) has a large amount of documentation on extending the program with custom plugins. In the future we will be posting a sample plugin project that can be used for additional custom widgets, but for now the wiki documentation is complete.

How can I build Shuffleboard from the source code?

You can get the source code by downloading, cloning, or forking the repository on the GitHub site. To build and run Shuffleboard from the source, make sure that the current directory is the top level source code and use one of these commands:

Application	Command (for Windows systems run the gradlew.bat file)
Running Shuffleboard	<code>./gradlew :app:run</code>
Building the APIs and utility classes for plugin creation	<code>./gradlew :api:shadowJar</code>
Building the complete application jar file	<code>./gradlew :app:shadowJar</code>

19.2 Shuffleboard - Layouts with Code

19.2.1 Using tabs

Shuffleboard is a tabbed interface. Each tab organizes widgets in a logical grouping. By default, Shuffleboard has tabs for the legacy SmartDashboard and LiveWindow - but new tabs can now be created in Shuffleboard directly from a robot program for better organization.

Creating a new tab

Java

C++

```
ShuffleboardTab tab = Shuffleboard.getTab("Tab Title");
```

```
ShuffleboardTab& tab = Shuffleboard::GetTab("Tab Title");
```

Creating a new tab is as simple as calling a single method on the Shuffleboard class, which will create a new tab on Shuffleboard and return a handle for adding your data to the tab. Calling `getTab` multiple times with the same tab title will return the same handle each time.

Selecting a tab

Java

C++

```
Shuffleboard.selectTab("Tab Title");
```

```
Shuffleboard::SelectTab("Tab Title");
```

This method lets a tab be selected by title. This is case-sensitive (so “Tab Title” and “Tab title” are two individual tabs), and only works if a tab with that title exists at the time the method is called, so calling `selectTab("Example")` will only have an effect if a tab named “Example” has previously been defined.

This method can be used to select any tab in Shuffleboard, not just ones created by the robot program.

Caveats

Tabs created from a robot program differ in a few important ways from normal tabs created from the dashboard:

- Not saved in the Shuffleboard save file
- No support for autopopulation
- Users are expected to specify the tab contents in their robot program
- Have a special color to differentiate from normal tabs

19.2.2 Sending data

Unlike SmartDashboard, data cannot be sent directly to Shuffleboard without first specifying what tab the data should be placed in.

Sending simple data

Sending simple data (numbers, strings, booleans, and arrays of these) is done by calling `add` on a tab. This method will set the value if not already present, but will not overwrite an existing value.

Java

C++

```
Shuffleboard.getTab("Numbers")
    .add("Pi", 3.14);
```

```
Shuffleboard::GetTab("Numbers")
    .Add("Pi", 3.14);
```

If data needs to be updated (for example, the output of some calculation done on the robot), call `getEntry()` after defining the value, then update it when needed or in a periodic function

Java

```
class VisionCalculator {
    private ShuffleboardTab tab = Shuffleboard.getTab("Vision");
    private NetworkTableEntry distanceEntry =
        tab.add("Distance to target", 0)
            .getEntry();

    public void calculate() {
        double distance = ...;
        distanceEntry.setDouble(distance);
    }
}
```

Making choices persist between reboots

When configuring a robot from the dashboard, some settings may want to persist between robot or driverstation reboots instead of having drivers remember (or forget) to configure the settings before each match.

Simply using *addPersistent* instead of *add* will make the value saved on the roboRIO and loaded when the robot program starts.

Note: This does not apply to sendable data such as choosers or motor controllers.

Java

```
Shuffleboard.getTab("Drive")
    .addPersistent("Max Speed", 1.0);
```

Sending sensors, motors, etc

Analogous to *SmartDashboard.putData*, any *Sendable* object (most sensors, motor controllers, and *SendableChoosers*) can be added to any tab

Java

```
Shuffleboard.getTab("Tab Title")
    .add("Sendable Title", mySendable);
```

19.2.3 Retrieving data

Unlike *SmartDashboard.getNumber* and friends, retrieving data from Shuffleboard is also done through the *NetworkTableEntries*, which we covered in the previous article.

Java

```
class DriveBase extends Subsystem {
    private ShuffleboardTab tab = Shuffleboard.getTab("Drive");
    private NetworkTableEntry maxSpeed =
        tab.add("Max Speed", 1)
            .getEntry();
}
```

(continues on next page)

(continued from previous page)

```
private DifferentialDrive robotDrive = ...;

public void drive(double left, double right) {
    // Retrieve the maximum speed from the dashboard
    double max = maxSpeed.getDouble(1.0);
    robotDrive.tankDrive(left * max, right * max);
}
}
```

This basic example has a glaring flaw: the maximum speed can be set on the dashboard to a value outside [0, 1] - which could cause the inputs to be saturated (always at maximum speed), or even reversed! Fortunately, there is a way to avoid this problem - covered in the next article.

19.2.4 Configuring widgets

Robot programs can specify exactly which widget to use to display a data point, as well as how that widget should be configured. As there are too many widgets to be listed here, consult the docs for details.

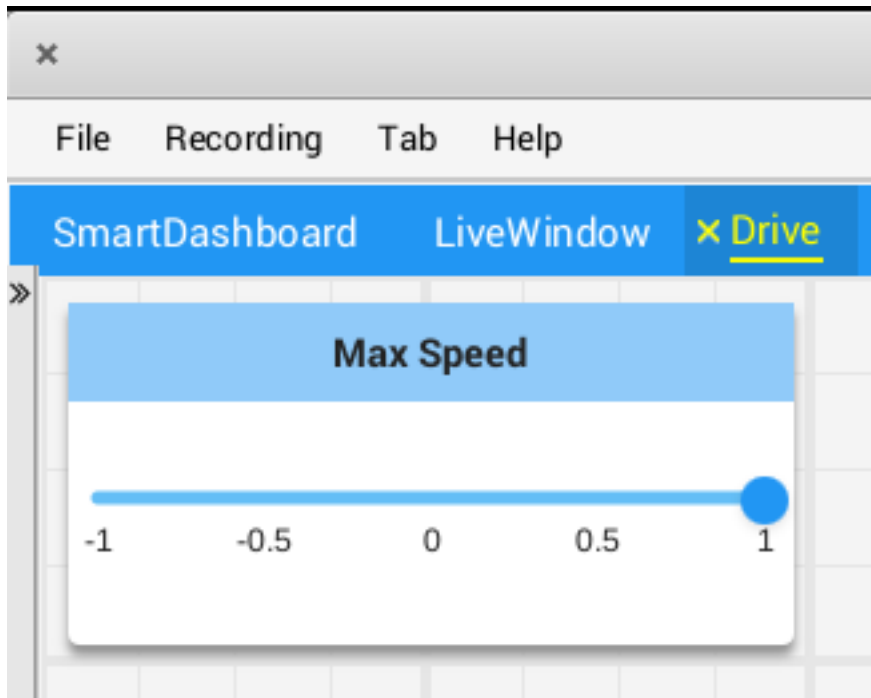
Specifying a widget

Call *withWidget* after *add* in the call chain:

Java

```
Shuffleboard.getTab("Drive")
    .add("Max Speed", 1)
    .withWidget(BuiltInWidgets.kNumberSlider) // specify the widget here
    .getEntry();
```

In this example, we configure the “Max Speed” widget to use a slider to modify the values instead of a basic text field.

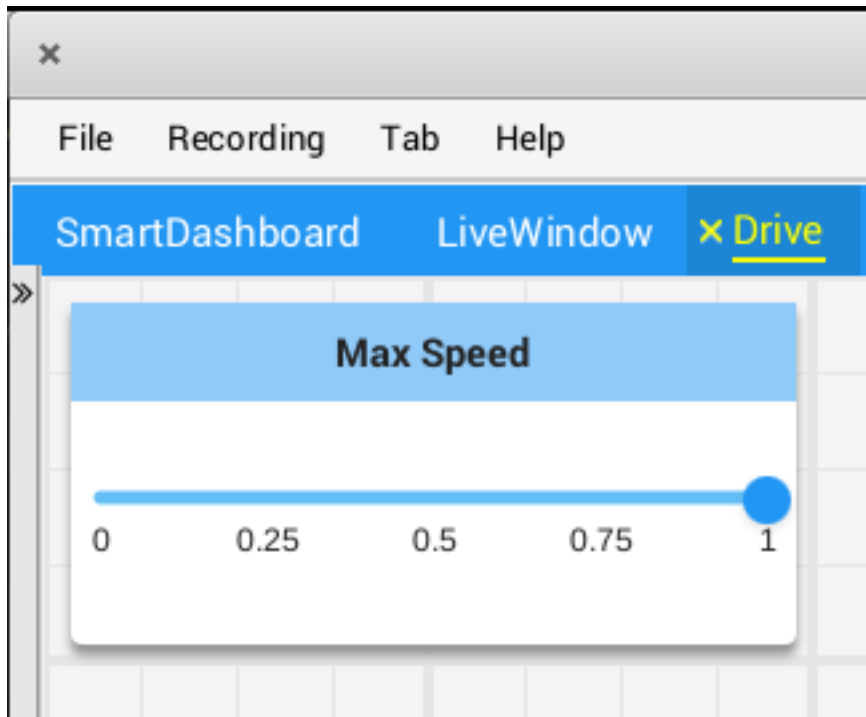


Setting widget properties

Since the maximum speed only makes sense to be a value from 0 to 1 (full stop to full speed), a slider from -1 to 1 can cause problems if the value drops below zero. Fortunately, we can modify that using the `withProperties` method

Java

```
Shuffleboard.getTab("Drive")
    .add("Max Speed", 1)
    .withWidget(BuiltInWidgets.kNumberSlider)
    .withProperties(Map.of("min", 0, "max", 1)) // specify widget properties here
    .getEntry();
```



Notes

Widgets can be specified by name; however, names are case- and whitespace-sensitive (“Number Slider” is different from “Number slider” and “NumberSlider”). For this reason, it is recommended to use the built in widgets class to specify the widget instead of by raw name. However, a custom widget can only be specified by name or by creating a custom `WidgetType` for that widget.

Widget property names are neither case-sensitive nor whitespace-sensitive (“Max” and “max” are the same). Consult the documentation on the widget in the `BuiltInWidgets` class for details on the properties of that widget.

19.2.5 Organizing Widgets

Setting Widget Size and Position

Call `withSize` and `withPosition` to set the size and position of the widget in the tab.

`withSize` sets the number of columns wide and rows high the widget should be. For example, calling `withSize(1, 1)` makes the widget occupy a single cell in the grid. Note that some widgets have a minimum size that may be greater than the specified size, in which case the widget will use the smallest supported size.

`withPosition` sets the row and column of the top-left corner of the widget. Rows and columns are both 0-indexed, so the topmost row is number 0 and the leftmost column is also number 0. If the position of any widget in a tab is specified, every widget should also have its position set to avoid overlapping widgets.

Java

C++

```
Shuffleboard.getTab("Pre-round")
    .add("Auto Mode", autoModeChooser)
    .withSize(2, 1) // make the widget 2x1
    .withPosition(0, 0); // place it in the top-left corner
```

```
frc::Shuffleboard::GetTab("Pre-round")
    .Add("Auto Mode", autoModeChooser)
    .WithSize(2, 1)
    .WithPosition(0,0);
```

Adding Widgets to Layouts

If there are many widgets in a tab with related data, it can be useful to place them into smaller subgroups instead of loose in the tab. Much like how the handle to a tab is retrieved with `Shuffleboard.getTab`, a layout inside a tab (or even in another layout) can be retrieved with `ShuffleboardTab.getLayout`.

Java

C++

```
ShuffleboardLayout elevatorCommands = Shuffleboard.getTab("Commands")
    .getLayout("Elevator", BuiltInLayouts.kList)
    .withSize(2, 2)
    .withProperties(Map.of("Label position", "HIDDEN")); // hide labels for commands

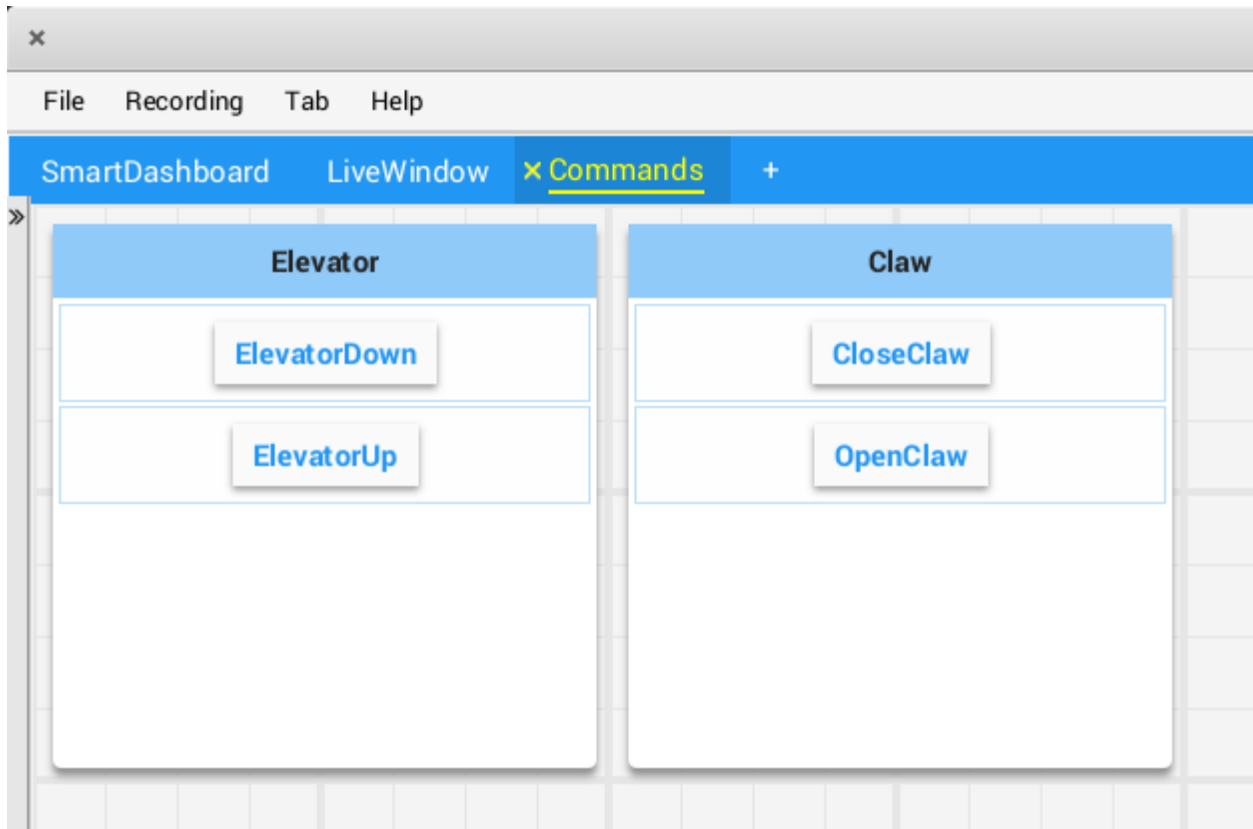
elevatorCommands.add(new ElevatorDownCommand());
elevatorCommands.add(new ElevatorUpCommand());
// Similarly for the claw commands
```

```
wpi::StringMap<std::shared_ptr<nt::Value>> properties{
    std::make_pair("Label position", nt::Value::MakeString("HIDDEN"))
};

frc::ShuffleboardLayout& elevatorCommands = frc::Shuffleboard::GetTab("Commands")
    .GetLayout("Elevator", frc::BuiltInLayouts::kList)
    .WithSize(2, 2)
    .WithProperties(properties);

ElevatorDownCommand* elevatorDown = new ElevatorDownCommand();
ElevatorUpCommand* elevatorUp = new ElevatorUpCommand();

elevatorCommands.Add("Elevator Down", elevatorDown);
elevatorCommands.Add("Elevator Up", elevatorUp);
```



19.3 Shuffleboard - Advanced Usage

19.3.1 Commands and Subsystems

When using the command-based framework Shuffleboard makes it easier to understand what the robot is doing by displaying the state of various commands and subsystems in real-time.

Displaying Subsystems

To see the status of a subsystem while the robot is operating in either autonomous or teleoperated modes, that is what its default command is and what command is currently using that subsystem, send a subsystem instance to Shuffleboard:

Java

C++

```
SmartDashboard.putData(subsystem-reference);
```

```
SmartDashboard::PutData(subsystem-pointer);
```

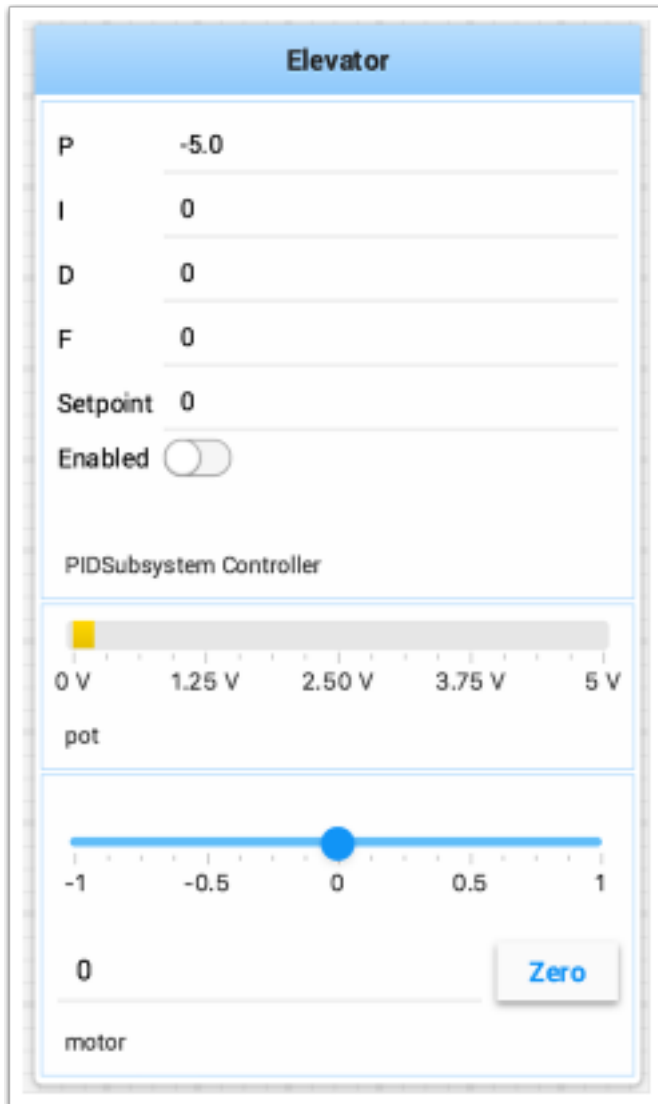
Shuffleboard will display the subsystem name, the default command associated with this subsystem, and the currently running command. In this example the default command for the

Elevator subsystem is called `AutonomousCommand` and it is also the current command that is using the Elevator subsystem.



Subsystems in Test Mode

In Test mode (Test/Enabled in the driver station) subsystems may be displayed in the LiveWindow tab with the sensors and actuators of the subsystem. This is ideal for verifying of sensors are working by seeing the values that they are returning. In addition, actuators can be operated. For example, motors can be operated using sliders to set their commanded speed and direction. For PIDSubsystems the P, I, D, and F constants are displayed along with the setpoint and an enable control. This is useful for tuning PIDSubsystems by adjusting the constants, putting in a setpoint, and enabling the embedded PIDController. Then the mechanism's response can be observed. This cycle (change parameters, enable, and observe) can be repeated until a reasonable set of parameters is found.



More information on tuning PIDSubsystems can be found [here](#). Using RobotBuilder will automatically generate the code to get the subsystem displayed in Test mode. The code that is necessary to have subsystems displayed is shown below where subsystem-name is a string containing the name of the subsystem:

```
setName(subsystem-name);
```

Displaying Commands

Using commands and subsystems makes very modular robot programs that can easily be tested and modified. Part of this is because commands can be written completely independently of other commands and can therefore be easily run from Shuffleboard. To write a command to Shuffleboard use the `SmartDashboard.putData` method as shown here:

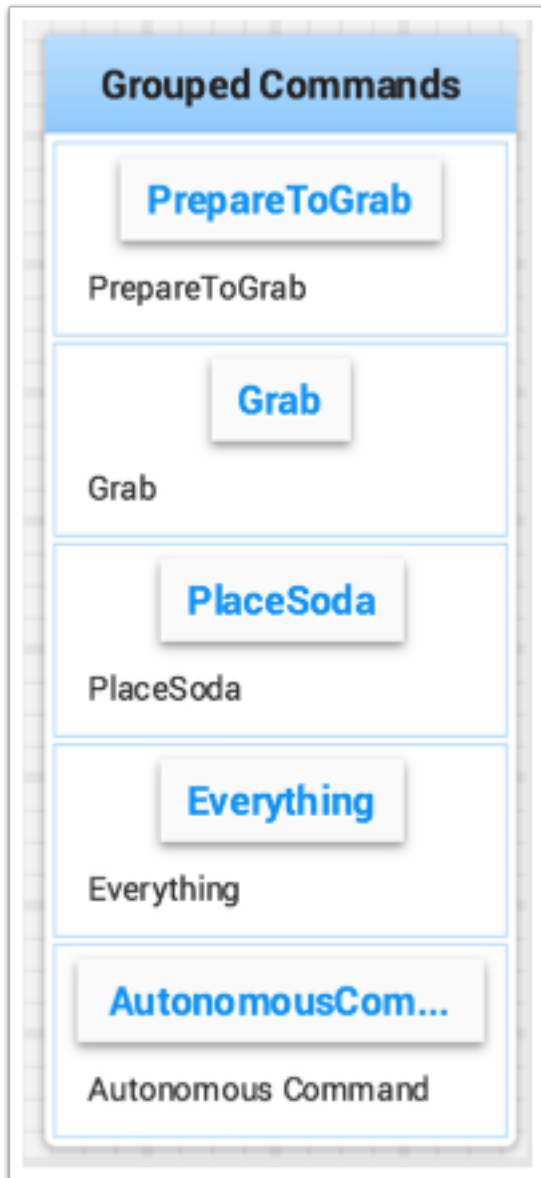
Java

C++

```
SmartDashboard.putData("ElevatorMove: up", new ElevatorMove(2.7));
```

```
SmartDashboard::PutData("ElevatorMove: up", new ElevatorMove(2.7));
```

Shuffleboard will display the command name and a button to execute the command. In this way individual commands and command groups can easily be tested without needing special test code in a robot program. In the image below there are a number of commands contained in a Shuffleboard list. Pressing the button once runs the command and pressing it again stops the command. To use this feature the robot must be enabled in teleop mode.



19.3.2 Testing and Tuning PID Loops

One challenge in using sensors to control mechanisms is to have a good algorithm to drive the motors to the proper position or speed. The most commonly used control algorithm is called PID control. There is a [good set of videos](#) (look for the robot controls playlist) that explain the control algorithms described here. The PID algorithm converts sensor values into motor speeds by:

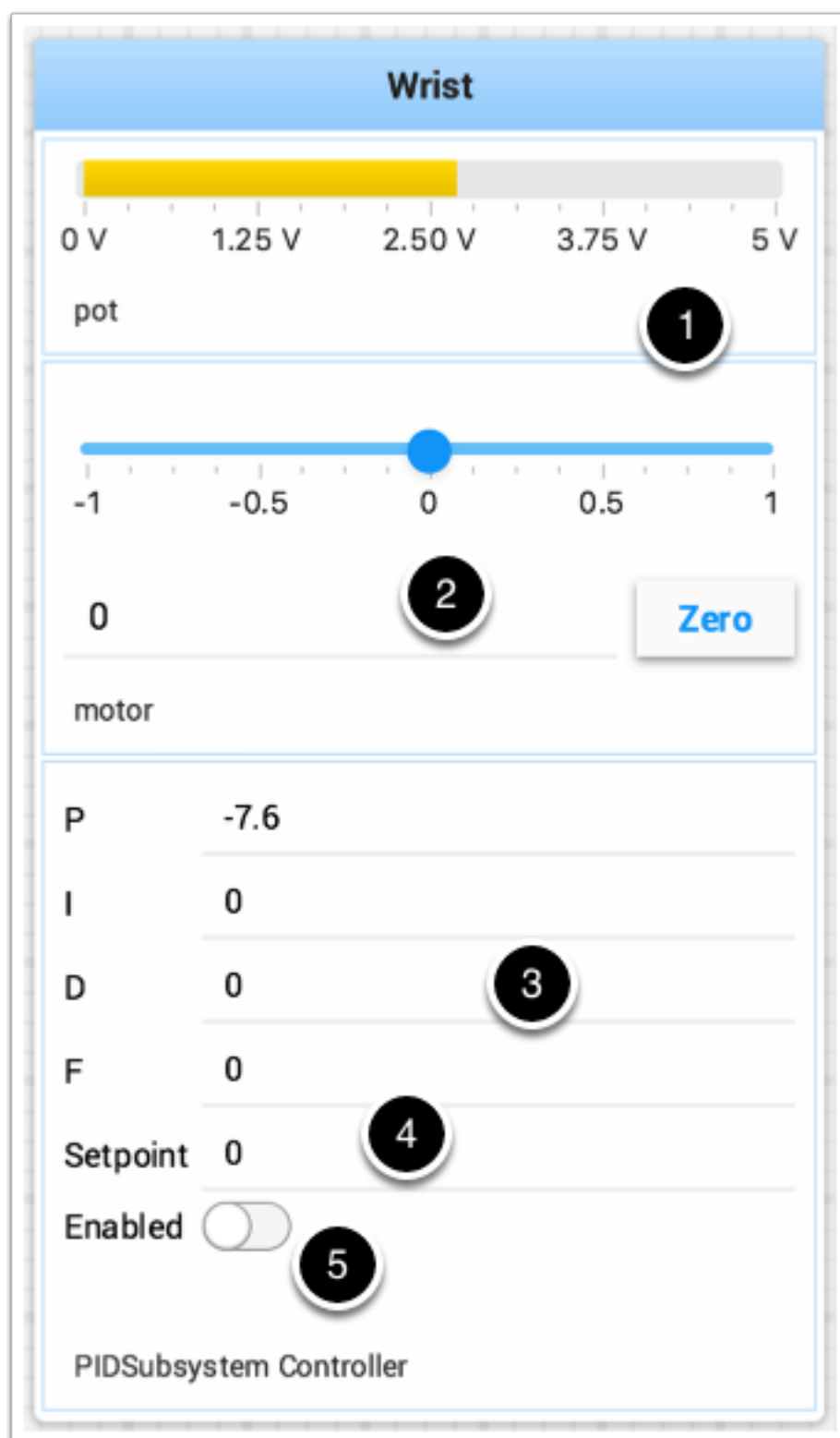
1. Reading sensor values to determine how far the robot or mechanism from the desired setpoint. The setpoint is the sensor value that corresponds to the expected goal. For example, a robot arm with a wrist joint should be able to move to a specified angle very quickly and stop at that angle as indicated by a sensor. A potentiometer is a sensor that can measure rotational angle. By connecting it to an analog input, the program can get a voltage measurement that is directly proportional to the angle.
2. Compute an error (the difference between the sensor value and the desired value). The sign of the error value indicates which side of the setpoint the wrist is on. For example negative values might indicate that the measured wrist angle is larger than the desired wrist angle. The magnitude of the error is how far the measured wrist angle is from the actual wrist angle. If the error is zero, then the measured angle exactly matches the desired angle. The error can be used as an input to the PID algorithm to compute a motor speed.
3. The resultant motor speed is then used to drive the motor in the correct direction and a speed that hopefully will reach the setpoint as quickly as possible without overshooting (moving past the setpoint).

WPILib has a `PIDController` class that implements the PID algorithm and accepts constants that correspond to the K_p , K_i , and K_d values. The PID algorithm has three components that contribute to computing the motor speed from the error.

1. P (proportional) - this is a term that when multiplied by a constant (K_p) will generate a motor speed that will help move the motor in the correct direction and speed.
2. I (integral) - this term is the sum of successive errors. The longer the error exists the larger the integral contribution will be. It is simply a sum of all the errors over time. If the wrist isn't quite getting to the setpoint because of a large load it is trying to move, the integral term will continue to increase (sum of the errors) until it contributes enough to the motor speed to get it to move to the setpoint. The sum of the errors is multiplied by a constant (K_i) to scale the integral term for the system.
3. D (differential) - this value is the rate of change of the errors. It is used to slow down the motor speed if it's moving too fast. It's computed by taking the difference between the current error value and the previous error value. It is also multiplied by a constant (K_d) to scale it to match the rest of the system.

Tuning the PID Controller

Tuning the PID controller consists of picking constants that will give good performance. Shuffleboard helps this process by displaying the details of a PID subsystem with a user interface for setting constant values and testing how well it operates. This is displayed while the robot is operating in test mode (done by setting "Test" in the driver station).



This is the test mode picture of a wrist subsystem that has a potentiometer as the sensor (pot) and a motor controller connected to the motor. It has a number of areas that correspond to the PIDSubsystem.

1. The analog input voltage value from the potentiometer. This is the sensor input value.

2. A slider that moves the wrist motor in either direction with 0 as stopped. The positive and negative values correspond to moving up or down.
3. The PID constants as described above (F is a feedforward value that is used for speed PID loops)
4. The setpoint value that corresponds the to the pot value when the wrist has reached the desired value
5. Enables the PID controller - No longer working, see below.

Try various PID gains to get the desired motor performance. You can look at the video linked to at the beginning of this article or other sources on the internet to get the desired performance.

Important: The enable option does not affect the `PIDController` introduced in 2020, as the controller is updated every robot loop. See the example below on how to retain this functionality.

Enable Functionality in the New PIDController

The following example demonstrates how to create a button on your dashboard that will enable/disable the PIDController.

Java

C++

```

ShuffleboardTab tab = Shuffleboard.getTab("Shooter");
NetworkTableEntry shooterEnable = tab.add("Shooter Enable", false).getEntry();

// Command Example assumed to be in a PIDSubsystem
new NetworkButton(shooterEnable).whenPressed(new InstantCommand(m_shooter::enable));

// Timed Robot Example
if (shooterEnable.getBoolean()) {
    // Calculates the output of the PID algorithm based on the sensor reading
    // and sends it to a motor
    motor.set(pid.calculate(encoder.getDistance(), setpoint));
}

```

```

frc::ShuffleboardTab& tab = frc::Shuffleboard::GetTab("Shooter");
nt::NetworkTableEntry shooterEnable = tab.Add("Shooter Enable", false).GetEntry();

// Command-based assumed to be in a PIDSubsystem
frc2::NetworkButton(shooterEnable).WhenPressed(frc2::InstantCommand([&] { m_shooter.
    ↪Enable(); }));

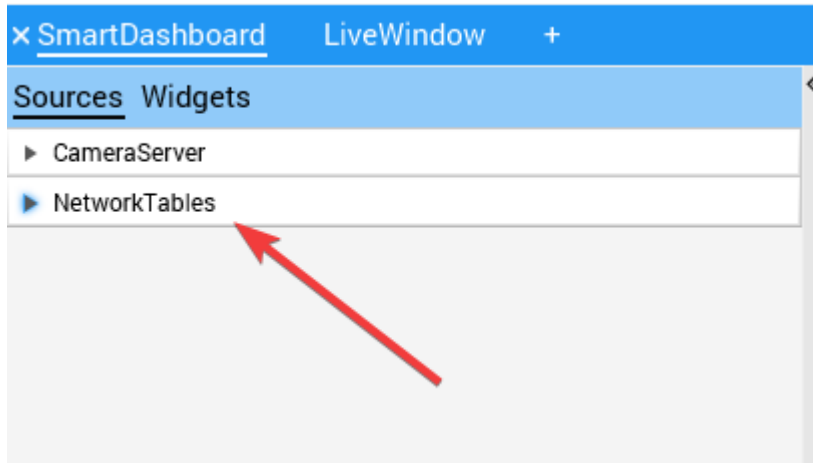
// Timed Robot Example
if (shooterEnable.GetBoolean()) {
    // Calculates the output of the PID algorithm based on the sensor reading
    // and sends it to a motor
    motor.Set(pid.Calculate(encoder.GetDistance(), setpoint));
}

```

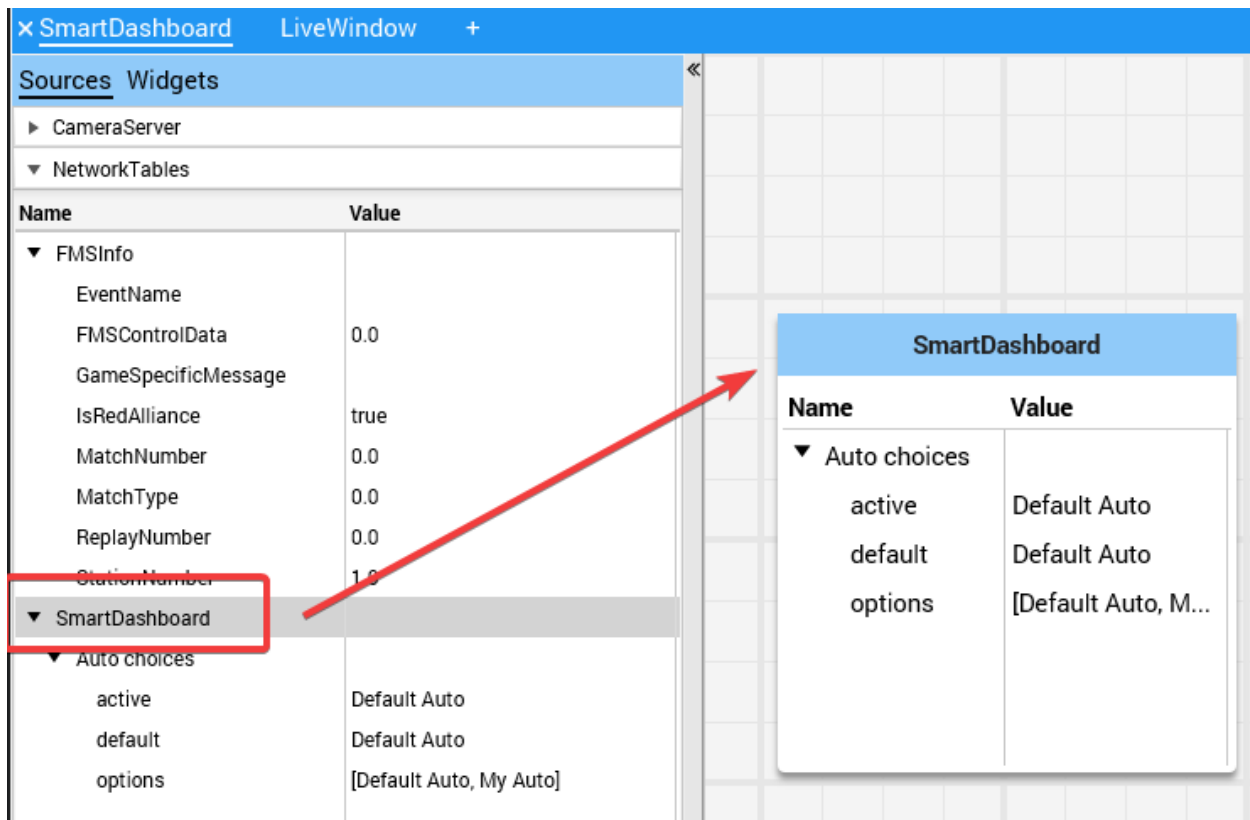
19.3.3 Viewing Hierarchies of Data

Dragging a key with other keys below it (deeper in the hierarchy) displays the hierarchy in a tree, similar to the NetworkTables sources on the left.

Select the data source:



Click and drag the NetworkTables key into the preferred tab.



19.4 Shuffleboard - Custom Widgets

19.4.1 Built-in Plugins

Shuffleboard provides a number of built-in plugins that handle common tasks for FRC® use, such as camera streams, all widgets, and *NetworkTables* connections.

Base Plugin

The base plugin defines all the data types, widgets, and layouts necessary for FRC use. It does *not* define any of the source types, or any special data types or widgets for those source types. Those are handled by the *NetworkTables Plugin* and the *CameraServer Plugin*. This separation of concerns makes it easier for teams to create plugins for custom source types or protocols (eg HTTP, ZeroMQ) for the FRC data types without needing a NetworkTables client.

CameraServer Plugin

The camera server plugin provides sources and widgets for viewing camerastreams from the CameraServer WPILib class.

This plugin depends on the *NetworkTables Plugin* in order to discover the available camera streams.

Stream discovery

CameraServer sources are automatically discovered by looking at the /CameraPublisher NetworkTable.

```
/CameraPublisher
  /<camera name>
    streams=["url1", "url2", ...]
```

For example, a camera named “Camera” with a server at roborio-0000-frc.local would have this table layout:

```
/CameraPublisher
  /Camera
    streams=["mjpeg:http://roborio-0000-frc.local:1181/?action=stream"]
```

This setup will automatically discover all camera streams hosted on a roboRIO by the CameraServer class in WPILib. Any non-WPILib projects that want to have camera streams appear in shuffleboard will have to set the streams entry for the camera server.

NetworkTables Plugin

The NetworkTables plugin provides data sources backed by ntc core. Since the LiveWindow and SmartDashboard classes in WPILib use NetworkTables to send the data to the driver station, this plugin will need to be loaded in order to use those classes.

This plugin handles the connection and reconnection to NetworkTables automatically, users of shuffleboard and writers of custom plugins will not have to worry about the intricacies of the NetworkTables protocol.

19.4.2 Creating a Plugin

Overview

Plugins provide the ability to create custom widgets, layouts, data sources/types, and custom themes. Shuffleboard provides the following *built-in plugins*.

- NetworkTables Plugin: To connect to data published over NetworkTables
- Base Plugin: To display custom FRC® data types in custom widgets
- CameraServer Plugin: To view streams from the CameraServer

Tip: An example custom Shuffleboard plugin which creates a custom data type and a simple widget for displaying it can be found [here](#).

Create a Custom Plugin

In order to define a plugin, the plugin class must be a subclass of `edu.wpi.first.shuffleboard.api.Plugin` or one of its subclasses. An example of a plugin class would be as following.

Java

```
import edu.wpi.first.shuffleboard.api.plugin.Description;
import edu.wpi.first.shuffleboard.api.plugin.Plugin;

@Description(group = "com.example", name = "MyPlugin", version = "1.2.3", summary =
    ↪ "An example plugin")
public class MyPlugin extends Plugin {

}
```

Additional explanations on how these attributes are used, including version numbers can be found [here](#).

Note the `@Description` annotation is needed to tell the plugin loader the properties of the custom plugin class. Plugin classes are permitted to have a default constructor but it cannot take any arguments.

Building plugin

Plugins require the usage of the [Shuffleboard API Library](#). These dependencies can be resolved in the `build.gradle` file or using maven. The dependencies would be as follows:

For Gradle:

```
repositories {
    mavenCentral()
    maven{ url "https://frcmaven.wpi.edu:443/artifactory/release" }
}

dependencies {
    compileOnly("edu.wpi.first.shuffleboard", "api", "2020.+")
    compileOnly("edu.wpi.first.shuffleboard.plugin", "networktables", "2020.+")
}
```

Plugins are allowed to have dependencies on other plugins and libraries, however, they must be included correctly in the maven or gradle build file. When a plugin depends on other plugins, it is good practice to define those dependencies so the plugin does not load when the dependencies do not load as well. This can be done using the `@Requires` annotation as shown below:

```
@Requires(group = "com.example", name = "Good Plugin", minVersion = "1.2.3")
@Requires(group = "edu.wpi.first.shuffleboard", "Base", minVersion = "1.0.0")
@Description(group = "com.example", name = "MyPlugin", version = "1.2.3", summary =
    ↪ "An example plugin")
public class MyPlugin extends Plugin {
}
}
```

The `minVersion` specifies the minimum allowable version of the plugin that can be loaded. For example, if the `minVersion` is 1.4.5, and the plugin with the version 1.4.7 is loaded, it will be allowed to do so. However, if the plugin with the version 1.2.4 is loaded, it will not be allowed to since it is less than the `minVersion`.

Deploying Plugin To Shuffleboard

In order to load a plugin in Shuffleboard, you will need to generate a jar file of the plugin and put it in the `~/Shuffleboard/plugins` folder. This can be done automatically from gradle as noted:

The path to your Shuffleboard plugin folder will most likely be `~/Shuffleboard/plugins`.

The `deployPlugin` task takes four parameters, the `type`: `Copy` parameter makes the task implement the [CopySpec](#) interface specifying what to copy. The `group` and `description` parameters to specify what the Group ID of the plugin is and a short descriptive description to what the Plugin does.

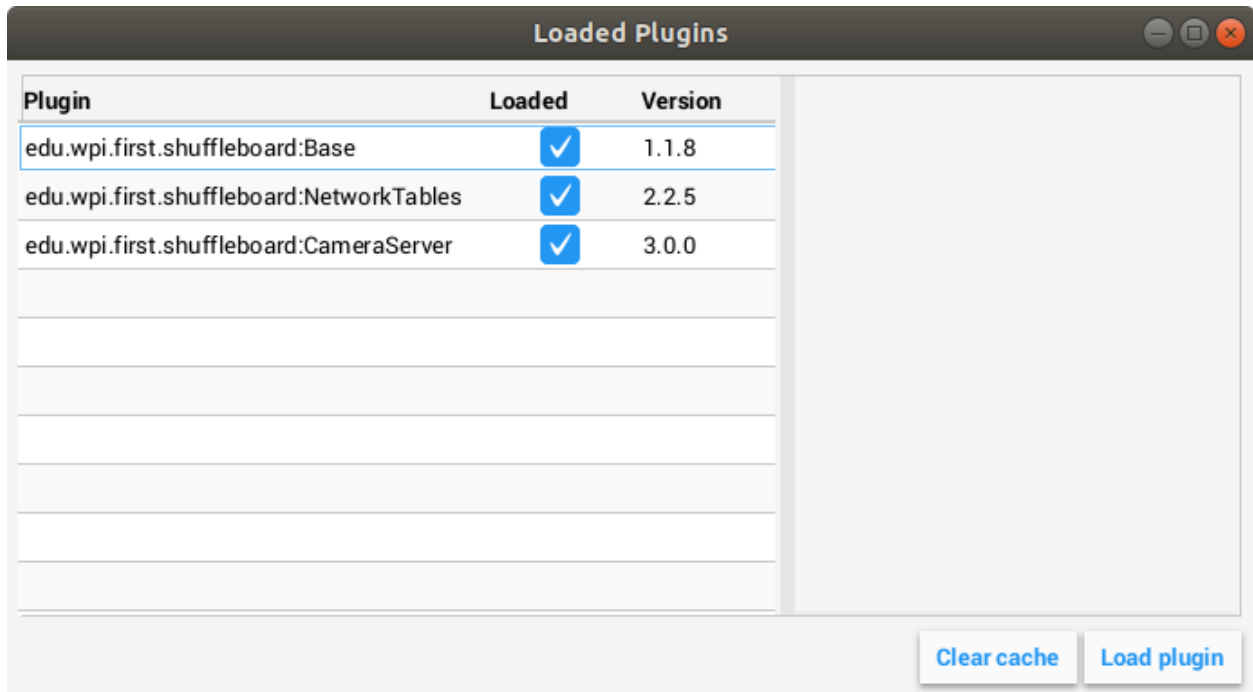
In the body, the `from` field specifies from where the file is to be copied from, followed by the `into` field specifying the destination to where the file needs to be copied to. Finally, the `include` field ensures all files with the `.jar` extension is also copied.

After deploying, Shuffleboard will cache the path of the plugin so it can be automatically loaded the next time Shuffleboard loads. It may be necessary to click on `Clear Cache` under the `plugins` menu to remove a plugin or reload a plugin into Shuffleboard.

By running `gradle deployPlugin` from the command line, the jar file will automatically be placed into the Shuffleboard plugin folder.

Manually Adding Plugin

The other way to add a plugin to Shuffleboard is to compile it to a jar file and add it from Shuffleboard. First, compile your plugin into a `.jar` file using Maven or Gradle. Then, open Shuffleboard, click on the file tab in the top left, and choose Plugins from the drop down menu.



From the plugins window, choose the “Load plugin” button in the bottom right, and select your jar file.

19.4.3 Creating Custom Data Types

Widgets allow us to control and visualize different types of data. This data could be integers and doubles or even Java Objects. In order to display these types of data using widgets, it is helpful to create a container class for them. It is not necessary to create your own Data Class if the widget will handle single fielded data types such as doubles, arrays, or strings.

Creating The Data Class

In this example, we will create a custom data type for a 2D Point and its x and y coordinates. In order to create a custom data type class, it must extend the abstract class `ComplexData`. Your custom data class must also implement the `asMap()` method that returns the represented data as a simple map as noted below with the `@Override` annotation:

```
import edu.wpi.first.shuffleboard.api.data.ComplexData;

import java.util.Map;

public class MyPoint2D extends ComplexData<MyPoint2D> {

    private final double x;
    private final double y;

    //Constructor should take all the different fields needed and assign them their
    ↪corresponding instance variables.
    public MyPoint2D(double x, double y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public Map<String, Object> asMap() {
        return Map.of("x", x, "y", y);
    }
}
```

It is also good practice to override the default equals and hashCode methods to ensure that different objects are considered equivalent when their fields are the same. The `asMap()` method should return the data represented in a simple Map object as it will be mapped to the NetworkTables entry it corresponds to. In this case, we can represent the point as its X and Y coordinates and return a Map containing them.

```
import edu.wpi.first.shuffleboard.api.data.ComplexData;

import java.util.Map;

public final class MyPoint2D extends ComplexData<MyPoint2D> {

    private final double x;
    private final double y;

    // Constructor should take all the different fields needed and assign them to
    ↪their corresponding instance variables.
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public Map<String, Object> asMap() {
        return Map.of("x", this.x, "y", this.y);
    }
}
```

Other methods can be added to retrieve or edit fields and instance variables, however, it is

good practice to make these classes immutable to prevent changing the source data objects. Instead, you can make a new copy object instead of manipulating the existing object. For example, if we wanted to change the y coordinate of our point, we can define the following method:

```
public MyPoint2D withY(double newY) {  
    return new MyPoint2D(this.x, newY);  
}
```

This creates a new `MyPoint2D` object and returns it with the new y-coordinate. Same can be done for changing the x coordinate.

Creating a Data Type

There are two different data types that can be made: Simple data types that have only one field (ie. a single number or string), and Complex data types that have multiple data fields (ie. multiple strings, multiple numbers).

In order to define a simple data type, the class must extend the `SimpleDataType<DataType>` class with the data type needed and implement the `getDefaultValue()` method. In this example, we will use a double as our simple data type.

```
public final class MyDoubleDataType extends SimpleDataType<Double> {  
  
    private static final String NAME = "Double";  
  
    private MyDataType() {  
        super(NAME, Double.class);  
    }  
  
    @Override  
    public Double getDefaultValue() {  
        return 0.0;  
    }  
}
```

The class constructor is set to private to ensure that only a single instance of the data type will exist.

In order to define a complex data type, the class must extend the `ComplexDataType` class and override the `fromMap()` and `getDefaultValue()` methods. We will use our `MyPoint2D` class as an example to see what a complex data type class would look like.

```
public final class PointDataType extends ComplexDataType<MyPoint2D> {  
  
    private static final String NAME = "MyPoint2D";  
    public static final PointDataType Instance = new PointDataType();  
  
    private PointDataType() {  
        super(NAME, MyPoint2D.class);  
    }  
  
    @Override  
    public Function<Map<String, Object>, MyPoint2D> fromMap() {  
        return map -> {  

```

(continues on next page)

(continued from previous page)

```

        return new MyPoint2D((double) map.getDefault("x", 0.0), (double) map.
↪getDefault("y", 0.0));
    };
}

@Override
public MyPoint2D getDefaultValue() {
    // use default values of 0 for X and Y coordinates
    return new MyPoint2D(0, 0);
}
}

```

The following code above works as noted:

The fromMap() method creates a new MyPoint2D using the values in the NetworkTables entry it is bound to. The getOrDefault method will return 0.0 if it cannot get the entry values. The getDefaultValue will return a new MyPoint2D object if no source is present.

Exporting Data Type To Plugin

In order to have the data type be recognized by Shuffleboard, the plugin must export them by overriding the getDataTypes method. For example,

```

public class MyPlugin extends Plugin {

    @Override
    public List<DataType> getDataTypes() {
        List.of(PointDataType.Instance);
    }
}

```

19.4.4 Creating A Widget

Widgets allow us to view, change, and interact with data published through different data sources. The CameraServer, NetworkTables, and Base plugins provide the widgets to control basic data types (including FRC-specific data types). However, custom widgets allow us to control our custom data types we made in the previous sections or Java Objects.

The basic Widget interface inherits from the Component and Sourced interfaces. Component is the most basic building block of components that be displayed in Shuffleboard. Sourced is an interface for things that can handle and interface with data sources to display or modify data. Widgets that don't support data bindings but simply have child nodes would not use the Sourced interface but simply the Component interface. Both are basic building blocks towards making widgets and allows us to modify and display data.

A good widget allows the end-user to customize the widget to suit their needs. An example could be to allow the user to control the range of the number slider, that is, its maximum and minimum or the orientation of the slider itself. The view of the widget or how it looks is defined using FXML. FXML is an XML based language that is useful for defining the static layout of the widget (Panels, Labels and Controls).

More about FXML can be found [here](#).

Defining a Widget's FXML

In this example, we will create two sliders to help us control the X and Y coordinates of our Point2D data type we created in previous sections. It is helpful to place the FXML file in the same package as the Java class.

In order to create an empty, blank window for our widget, we need to create a Pane. A Pane is a parent node that contains other child nodes, in this case, 2 sliders. There are many different types of Pane, they are as noted:

- Stack Pane
 - Stack Panes allow elements to be overlaid. Also, StackPanes by default center child nodes.
- Grid Pane
 - Grid Panes are extremely useful defining child elements using a coordinate system by creating a flexible grid of rows and columns on the pane.
- Flow Pane
 - Flow Panes wrap all child nodes at a boundary set. Child nodes can flow vertically (wrapped at the height boundary for the pane) or horizontally (wrapped at the width boundary of the pane).
- Anchor Pane
 - Anchor Panes allow child elements to be placed in the top, bottom, left side, right side, or center of the pane.

Layout panes are also extremely useful for placing child nodes in one horizontal row using a [HBox](#) or one vertical column using a [VBox](#).

The basic syntax for defining a Pane using FXML would be as the following:

```
<?import javafx.scene.layout.*?>
<StackPane xmlns:fx="http://javafx.com/fxml/1" fx:controller="/path/to/widget/class"
  ↪fx:id="root">
  ...
</StackPane>
```

The `fx:controller` attribute contains the name of the widget class. An instance of this class is created when the FXML file is loaded. For this to work, the controller class must have a no-argument constructor.

Creating A Widget Class

Now that we have a Pane, we can now add child elements to that pane. In this example, we can add two slider objects. Remember to add an `fx:id` to each element so they can be referenced in our Java class we will make later on. We will use a `VBox` to position our slider on top of each other.

```
<?import javafx.scene.layout.*?>
<StackPane xmlns:fx="http://javafx.com/fxml/1" fx:controller="/path/to/widget/class"
  ↪fx:id="root">

  <VBox>
    <Slider fx:id = "xSlider"/>
```

(continues on next page)

(continued from previous page)

```

        <Slider fx:id = "ySlider"/>
    </VBox>

</StackPane>

```

Now that we have finished creating our FXML file, we can now create a widget class. The widget class should include a `@Description` annotation that states the supported data types of the widget and the name of the widget. If a `@Description` annotation is not present, the plugin class must implement the `get()` method to return its widgets.

It also must include a `@ParametrizedController` annotation that points to the FXML file containing the layout of the widget. If the class that only supports one data source it must extend the `SimpleAnnotatedWidget` class. If the class supports multiple data sources, it must extend the `ComplexAnnotatedWidget` class. For more information, see [Widget Types](#).

```

import edu.wpi.first.shuffleboard.api.widget.Description;
import edu.wpi.first.shuffleboard.api.widget.ParametrizedController;
import edu.wpi.first.shuffleboard.api.widget.SimpleAnnotatedWidget;

/*
 * If the FXML file and Java file are in the same package, that is the Java file is
 * in src/main/java and the
 * FXML file is under src/main/resources or your code equivalent package, the
 * relative path will work
 * However, if they are in different packages, an absolute path will be required.
 */

@Description(name = "MyPoint2D", dataTypes = MyPoint2D.class)
@ParametrizedController("Point2DWidget.fxml")
public final class Point2DWidget extends SimpleAnnotatedWidget<MyPoint2D> {

}

```

If you are not using a custom data type, you can reference any Java data type (ie. `Double.class`), or if the widget does not need data binding you can pass `NoneType.class`.

Now that we have created our class we can create fields for the widgets we declared in our FXML file using the `@FXML` annotation. For our two sliders, an example would be:

```

import edu.wpi.first.shuffleboard.api.widget.Description;
import edu.wpi.first.shuffleboard.api.widget.ParametrizedController;
import edu.wpi.first.shuffleboard.api.widget.SimpleAnnotatedWidget;
import javafx.fxml.FXML;

@Description(name = "MyPoint2D", dataTypes = MyPoint2D.class)
@ParametrizedController("Point2DWidget.fxml")
public final class Point2DWidget extends SimpleAnnotatedWidget<MyPoint2D> {

    @FXML
    private Pane root;

    @FXML
    private Slider xSlider;

    @FXML
    private Slider ySlider;
}

```

In order to display our pane on our custom widget we need to override the `getView()` method and return our `StackPane`.

```
import edu.wpi.first.shuffleboard.api.widget.Description;
import edu.wpi.first.shuffleboard.api.widget.ParametrizedController;
import edu.wpi.first.shuffleboard.api.widget.SimpleAnnotatedWidget;
import javafx.fxml.FXML;

@Description(name = "MyPoint2D", dataTypes = MyPoint2D.class)
@ParametrizedController("Point2DWidget.fxml")
public final class Point2DWidget extends SimpleAnnotatedWidget<MyPoint2D> {

    @FXML
    private StackPane root;

    @FXML
    private Slider xSlider;

    @FXML
    private Slider ySlider;

    @Override
    public Pane getView() {
        return root;
    }
}
```

Binding Elements and Adding Listeners

Binding is a mechanism that allows JavaFX widgets to express direct relationships with the data source. For example, changing a widget will change its related `NetworkTableEntry` and vice versa.

An example, in this case, would be changing the X and Y coordinate of our 2D point by changing the values of `xSlider` and `ySlider` respectively.

A good practice is to set bindings in the `initialize()` method tagged with the `@FXML` annotation which is required to call the method from FXML if the method is not public.

```
import edu.wpi.first.shuffleboard.api.widget.Description;
import edu.wpi.first.shuffleboard.api.widget.ParametrizedController;
import edu.wpi.first.shuffleboard.api.widget.SimpleAnnotatedWidget;
import javafx.fxml.FXML;

@Description(name = "MyPoint2D", dataTypes = MyPoint2D.class)
@ParametrizedController("Point2DWidget.fxml")
public final class Point2DWidget extends SimpleAnnotatedWidget<MyPoint2D> {

    @FXML
    private StackPane root;

    @FXML
    private Slider xSlider;

    @FXML
```

(continues on next page)

(continued from previous page)

```

private Slider ySlider;

@FXML
private void initialize() {
    xSlider.valueProperty().bind(dataOrDefault.map(MyPoint2D::getX));
    ySlider.valueProperty().bind(dataOrDefault.map(MyPoint2D::getY));
}

@Override
public Pane getView() {
    return root;
}
}

```

The above `initialize` method binds the slider's value property to the `MyPoint2D` data class' corresponding X and Y value. Meaning, changing the slider will change the coordinate and vice versa. The `dataOrDefault.map()` method will get the data source's value, or, if no source is present, will return the default value.

Using a listener is another way to change values when the slider or data source has changed. For example a listener for our slider would be:

```

xSlider.valueProperty().addListener((observable, oldValue, newValue) -> {
    setData(getData().withX(newValue));
});

```

In this case, the `setData()` method sets the value in the data source of the widget to the `newValue`.

Exploring Custom Components

Widgets are not automatically discovered when loading plugins; the defining plugin must explicitly export it for it to be usable. This approach is taken to allow multiple plugins to be defined in the same JAR.

```

@Override
public List<ComponentType> getComponents() {
    return List.of(WidgetType.forAnnotatedWidget(Point2DWidget.class));
}

```

Set Default Widget For Data type

In order to set your widget as default for your custom data type, you can override the `getDefaultComponents()` in your plugin class that stores a `Map` for all default widgets as noted below:

```

@Override
public Map<DataType, ComponentType> getDefaultComponents() {
    return Map.of(Point2DType.Instance, WidgetType.forAnnotatedWidget(Point2DWidget.class));
}

```

19.4.5 Custom Themes

Since shuffleboard is a JavaFX application, it has support for custom themes via Cascading Stylesheets (**CSS** for short). These are commonly used on webpages for making HTML look nice, but JavaFX also has support, albeit for a different language subset (see [here](#) for documentation on how to use it).

Shuffleboard comes with three themes by default: Material Light, Material Dark, and Midnight. These are color variations on the same material design stylesheet. In addition, they inherit from a `base.css` stylesheet that defines styles for the custom components defined in shuffleboard or libraries that it uses; the base material design stylesheet only applies to the UI components built into JavaFX.

There are two ways to define a custom theme: place the stylesheets in a directory with the name of the theme in `~/Shuffleboard/themes`; for example, a theoretical “Yellow” theme could be placed in

```
~/Shuffleboard/themes/Yellow/yellowtheme.css
```

All the stylesheets in the directory will be treated as part of the theme.

Loading Themes via Plugins

Custom themes can also be defined by plugins. This makes them easier to share and bundle with custom widgets, but are slightly more difficult to define. The theme object will need a reference to a class defined in the plugin so that the plugin loader can determine where the stylesheets are located. If a class is passed that is *not* present in the JAR that the plugin is in, the theme will not be able to be used.

```
@Description(group = "com.example", name = "My Plugin", version = "1.2.3", summary = "
→")
class MyPlugin extends Plugin {

    private static final Theme myTheme = new Theme(MyPlugin.class, "My Theme Name", "/"
→path/to/stylesheet", "/path/to/stylesheet", ...);

    @Override
    public List<Theme> getThemes() {
        return ImmutableList.of(myTheme);
    }
}
```

Modifying or Extending Shuffleboard’s Default Themes

Shuffleboard’s Material Light and Material Dark themes provide a lot of the framework for light and dark themes, respectively, as well as many styles specific to shuffleboard, ControlsFX, and Medusa UI components to fit with the material-style design.

Themes that want to modify these themes need to add import statements for these stylesheets:


```

@import "/edu/wpi/first/shuffleboard/api/material.css"; /* Material design CSS for
↳ JavaFX components */
@import "/edu/wpi/first/shuffleboard/api/base.css"; /* Material design CSS for
↳ shuffleboard components */
@import "/edu/wpi/first/shuffleboard/app/light.css"; /* CSS for the Material Light
↳ theme */
@import "/edu/wpi/first/shuffleboard/app/dark.css"; /* CSS for the Material Dark
↳ theme */
@import "/edu/wpi/first/shuffleboard/app/midnight.css"; /* CSS for the Midnight
↳ theme */

```

Note that `base.css` internally imports `material.css`, and `light.css`, `dark.css`, and `midnight.css` all import `base.css`, so importing `light.css` will implicitly import both `base.css` and `material.css` as well.

Source Code for the CSS Files

- `_material.css`: <https://github.com/wpilibsuite/shuffleboard/blob/main/api/src/main/resources/edu/wpi/first/shuffleboard/api/material.css>
- `_base.css`: <https://github.com/wpilibsuite/shuffleboard/blob/main/api/src/main/resources/edu/wpi/first/shuffleboard/api/base.css>
- `_light.css`: <https://github.com/wpilibsuite/shuffleboard/blob/main/app/src/main/resources/edu/wpi/first/shuffleboard/app/light.css>
- `_dark.css`: <https://github.com/wpilibsuite/shuffleboard/blob/main/app/src/main/resources/edu/wpi/first/shuffleboard/app/dark.css>
- `_midnight.css`: <https://github.com/wpilibsuite/shuffleboard/blob/main/app/src/main/resources/edu/wpi/first/shuffleboard/app/midnight.css>

Material Design Color Swatches

The material design CSS uses color swatch variables for almost everything. These variables can be set from custom CSS files, reducing the amount of custom code needed.

The `-swatch-<100|200|300|400|500>` variables define progressively darker shades of the same primary color. The light theme uses the default shades of blue set in `material.css`, but the dark theme overrides these with shades of red. `-swatch-<|light|dark>-gray` defines three levels of gray to use for various background or text colors.

Overriding the Swatch Colors

Replacing blue with red (light)

```

@import "/edu/wpi/first/shuffleboard/app/light.css"

.root {
  -swatch-100: hsb(0, 80%, 98%);
  -swatch-200: hsb(0, 80%, 88%);
  -swatch-300: hsb(0, 80%, 78%);

```

(continues on next page)

(continued from previous page)

```

-swatches-400: hsb(0, 80%, 68%);
-swatches-500: hsb(0, 80%, 58%);
}

```

Replacing red with blue (dark)

```

@import "/edu/wpi/first/shuffleboard/app/dark.css"

.root {
  -swatches-100: #BBDEFB;
  -swatches-200: #90CAF9;
  -swatches-300: #64B5F6;
  -swatches-400: #42A5F5;
  -swatches-500: #2196F3;
}

```

19.4.6 Widget Types

While Widget is pretty straightforward as far as the interface is concerned, there are several intermediate implementations to make it easier to define the widget.

Class	Description
AbstractWidget	Implements <code>getProperties()</code> , <code>getSources()</code> , and <code>titleProperty()</code>
SingleTypeWidget<T>	Adds properties for widgets that only support a single data type
AnnotatedWidget	Adds default implementations for <code>getName()</code> and <code>getDataTypes()</code> for widgets with a <code>@Description</code> annotation
SingleSourceWidget	For widgets with only a single source (by default, widgets support multiple sources)
SimpleAnnotatedWidget<T>	Combines <code>SingleTypeWidget<T></code> , <code>AnnotatedWidget</code> , and <code>SingleSourceWidget</code>

There are also two annotations to help define widgets:

Name	Description
@ParametrizedController	Allows widgets to be FXML controllers for JavaFX views defined via FXML
@Description	Lets the name and supported data types be defined in a single line

AbstractWidget

This class implements `getProperties()`, `getSources()`, `addSource()`, and `titleProperty()`. It also defines a method `exportProperties(Property<?>...)` method so subclasses can easily add custom widget properties, or properties for the JavaFX components in the widget. Most of the [widgets in the base plugin](#) use this.

SingleTypeWidget

A type of widget that only supports a single data type. This interface is parametrized and has methods for setting or getting the data, as well as a method for getting the (single) data type of the widget.

AnnotatedWidget

This interface implements `getDataTypes()` and `getName()` by looking at the the `@Description` annotation on the implementing class. This *requires* the annotation to be present, or the widget will not be able to be loaded and used.

```
// No @Description annotation!
public class WrongImplementation implements AnnotatedWidget {
    // ...
}
```

```
@Description(name = ..., dataTypes = ...)
public class CorrectImplementation implements AnnotatedWidget {
    // ...
}
```

SingleSourceWidget

A type of widget that only uses a single source.

SimpleAnnotatedWidget

A combination of `SingleTypeWidget<T>`, `AnnotatedWidget`, and `SingleSourceWidget`. Most widgets in the base plugin extend from this class. This also has a protected field called `dataOrDefault` that lets subclasses use a default data value if the widget doesn't have a source, or if the source is providing null.

@ParametrizedController

This annotation can be placed on a widget class to let shuffleboard know that it's an FXML controller for a JavaFX view defined via FXML. The annotation takes a single parameter that defines where the FXML file *in relation to the class on which it is placed*. For example, a widget in the directory `src/main/java/com/acme` that is an FXML controller for a FXML file in `src/main/resources/com/acme` can use the annotation as either

```
@ParametrizedController("MyWidget.fxml")
```

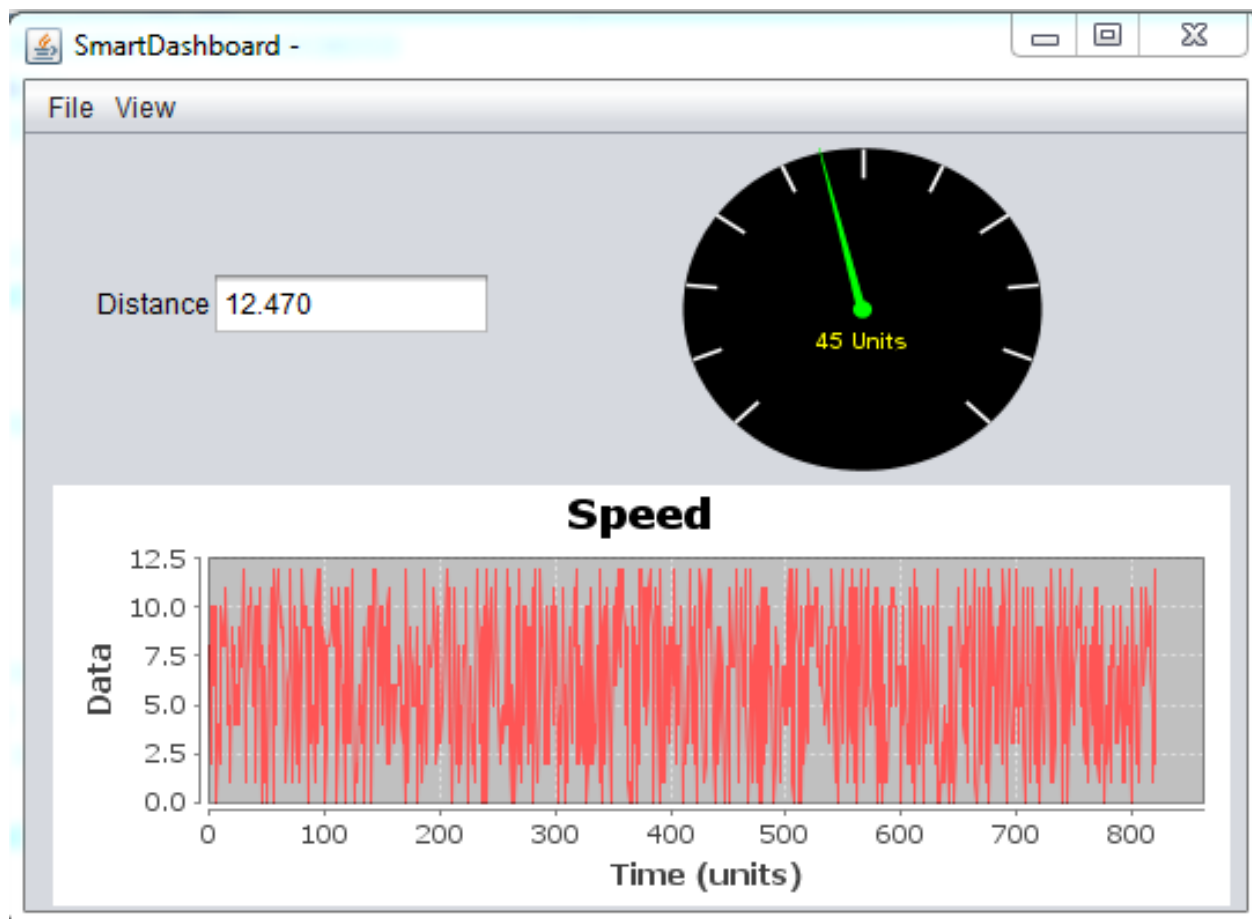
or as

```
@ParametrizedController("/com/acme/MyWidget.fxml")
```

@Description

This allows widgets to have their name and supported data types defined by a single annotation, when used alongside *AnnotatedWidget*.

20.1 SmartDashboard Introduction



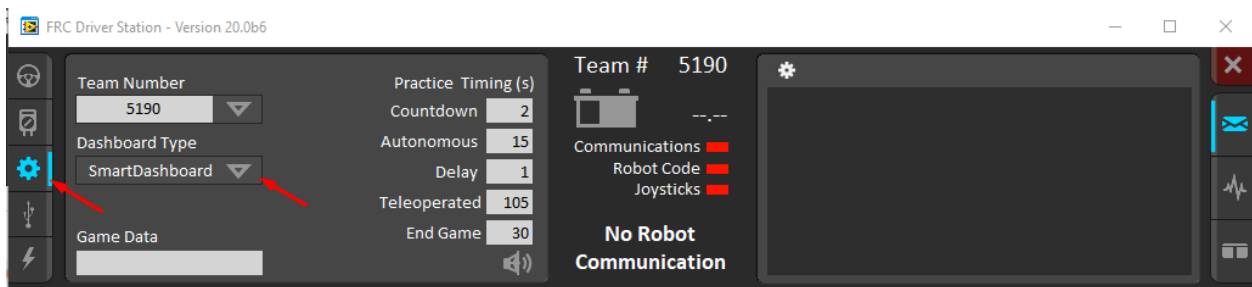
The SmartDashboard is a Java program that will display robot data in real time. The SmartDashboard helps you with these things:

- Displays robot data of your choice while the program is running. It can be displayed as simple text fields or more elaborately in many other display types like graphs, dials, etc.

- Displays the state of the robot program such as the currently executing commands and the status of any subsystems
- Displays buttons that you can press to cause variables to be set on your robot
- Allows you to choose startup options on the dashboard that can be read by the robot program

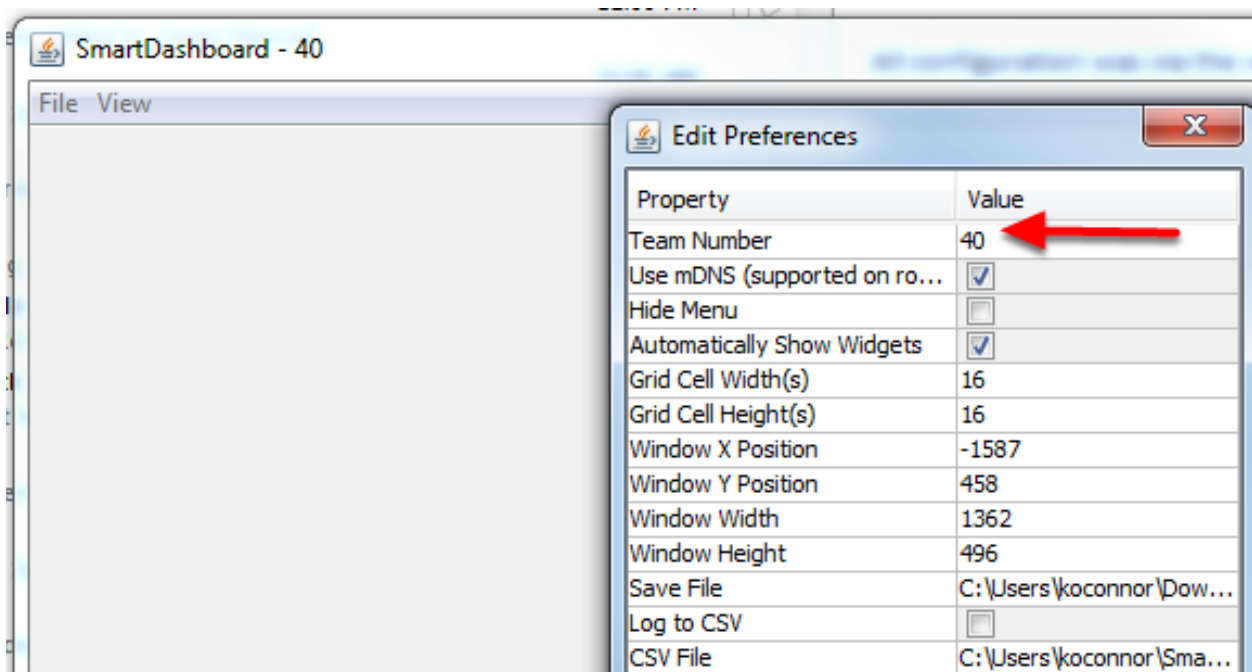
The displayed data is automatically formatted in real-time as the data is sent from the robot, but you can change the format or the display widget types and then save the new screen layouts to be used again later. And with all these options, it is still extremely simple to use. To display some data on the dashboard, simply call one of the SmartDashboard methods with the data and its name and the value will automatically appear on the dashboard screen.

20.1.1 Installing the SmartDashboard



The SmartDashboard is packaged with the WPILib Installer and can be launched directly from the Driver Station by selecting the **SmartDashboard** button on the Setup tab.

20.1.2 Configuring the Team Number



The first time you launch the SmartDashboard you should be prompted for your team number. To change the team number after this: click **File > Preferences** to open the Preferences dialog. Double-click the box to the right of **Team Number** and enter your FRC® Team Number, then click outside the box to save.

Note: SmartDashboard will take a moment to configure itself for the team number, do not be alarmed.

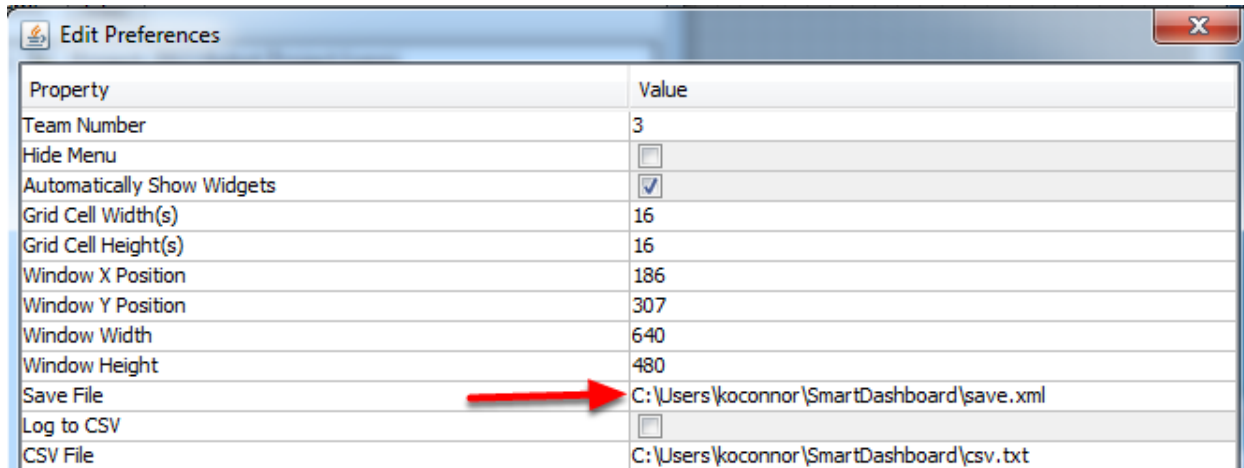
Setting a Custom NetworkTables Server Location

By default, SmartDashboard will look for NetworkTables instances running on a connected RoboRIO, but it's sometimes useful to look for NetworkTables at a different IP address. To connect to SmartDashboard from a host other than the roboRIO, open SmartDashboard preferences under the File menu and in the Team Number field, enter the IP address or hostname of the NetworkTables host.

This option is incredibly useful for using SmartDashboard with *WPILib simulation*. Simply add localhost to the Team Number field and SmartDashboard will detect your locally-hosted robot!

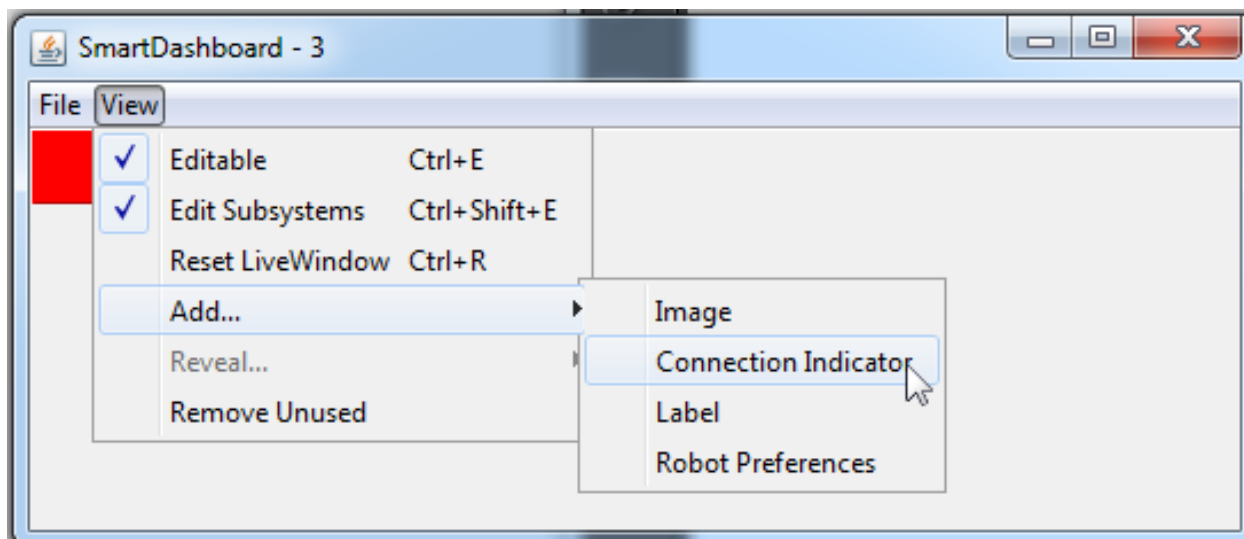
Edit Preferences	
Property	Value
Team Number	localhost
Hide Menu	<input type="checkbox"/>
Automatically Show Widgets	<input checked="" type="checkbox"/>
Grid Cell Width(s)	16
Grid Cell Height(s)	16
Window X Position	723
Window Y Position	432
Window Width	317
Window Height	200
Save File	/example
Log to CSV	<input type="checkbox"/>
CSV File	/example

20.1.3 Locating the Save File



Users may wish to customize the save location of the SmartDashboard. To do this click the box next to **Save File** then browse to the folder where you would like to save the configuration. Files saved in the installation directories for the WPILib components will likely be overwritten on updates to the tools.

20.1.4 Adding a Connection Indicator



It is often helpful to see if the SmartDashboard is connected to the robot. To add a connection indicator, select **View > Add > Connection Indicator**. This indicator will be red when disconnected and green when connected. To move or resize this indicator, select **View > Editable** to toggle the SmartDashboard into editable mode, then drag the center of the indicator to move it or the edges to resize. Select the **Editable** item again to lock it in place.

20.1.5 Adding Widgets to the SmartDashboard

Widgets are automatically added to the SmartDashboard for each “key” sent by the robot code. For instructions on adding robot code to write to the SmartDashboard see [Displaying Expressions from Within the Robot Program](#).

20.2 Displaying Expressions from a Robot Program

Note: Often debugging or monitoring the status of a robot involves writing a number of values to the console and watching them stream by. With SmartDashboard you can put values to a GUI that is automatically constructed based on your program. As values are updated, the corresponding GUI element changes value - there is no need to try to catch numbers streaming by on the screen.

20.2.1 Writing Values to SmartDashboard

Java

C++

```
protected void execute() {
    SmartDashboard.putBoolean("Bridge Limit", bridgeTipper.atBridge());
    SmartDashboard.putNumber("Bridge Angle", bridgeTipper.getPosition());
    SmartDashboard.putNumber("Swerve Angle", drivetrain.getSwerveAngle());
    SmartDashboard.putNumber("Left Drive Encoder", drivetrain.getLeftEncoder());
    SmartDashboard.putNumber("Right Drive Encoder", drivetrain.getRightEncoder());
    SmartDashboard.putNumber("Turret Pot", turret.getCurrentAngle());
    SmartDashboard.putNumber("Turret Pot Voltage", turret.getAverageVoltage());
    SmartDashboard.putNumber("RPM", shooter.getRPM());
}
```

```
void Command::Execute() {
    frc::SmartDashboard::PutBoolean("Bridge Limit", BridgeTipper.AtBridge());
    frc::SmartDashboard::PutNumber("Bridge Angle", BridgeTipper.GetPosition());
    frc::SmartDashboard::PutNumber("Swerve Angle", Drivetrain.GetSwerveAngle());
    frc::SmartDashboard::PutNumber("Left Drive Encoder", Drivetrain.GetLeftEncoder());
    frc::SmartDashboard::PutNumber("Right Drive Encoder", Drivetrain.GetRightEncoder());
    frc::SmartDashboard::PutNumber("Turret Pot", Turret.GetCurrentAngle());
    frc::SmartDashboard::PutNumber("Turret Pot Voltage", Turret.GetAverageVoltage());
    frc::SmartDashboard::PutNumber("RPM", Shooter.GetRPM());
}
```

You can write Boolean, Numeric, or String values to the SmartDashboard by simply calling the correct method for the type and including the name and the value of the data, no additional code is required. Any time in your program that you write another value with the same name, it appears in the same UI element on the screen on the driver station or development computer. As you can imagine this is a great way of debugging and getting status of your robot as it is operating.

20.2.2 Creating Widgets on SmartDashboard

Widgets are populated on the SmartDashboard automatically, no user intervention is required. Note that the widgets are only populated when the value is first written, you may need to enable your robot in a particular mode or trigger a particular code routine for an item to appear. To alter the appearance of the widget, see the next two sections *Changing the Display Properties of a Value* and *Changing the Display Widget Type for a Value*.

20.2.3 Stale Data

SmartDashboard uses *NetworkTables* for communicating values between the robot and the driver station laptop. NetworkTables acts as a distributed table of name and value pairs. If a name/value pair is added to either the client (laptop) or server (robot) it is replicated to the other. If a name/value pair is deleted from, say, the robot but the SmartDashboard or OutlineViewer are still running, then when the robot is restarted, the old values will still appear in the SmartDashboard and OutlineViewer because they never stopped running and continue to have those values in their tables. When the robot restarts, those old values will be replicated to the robot.

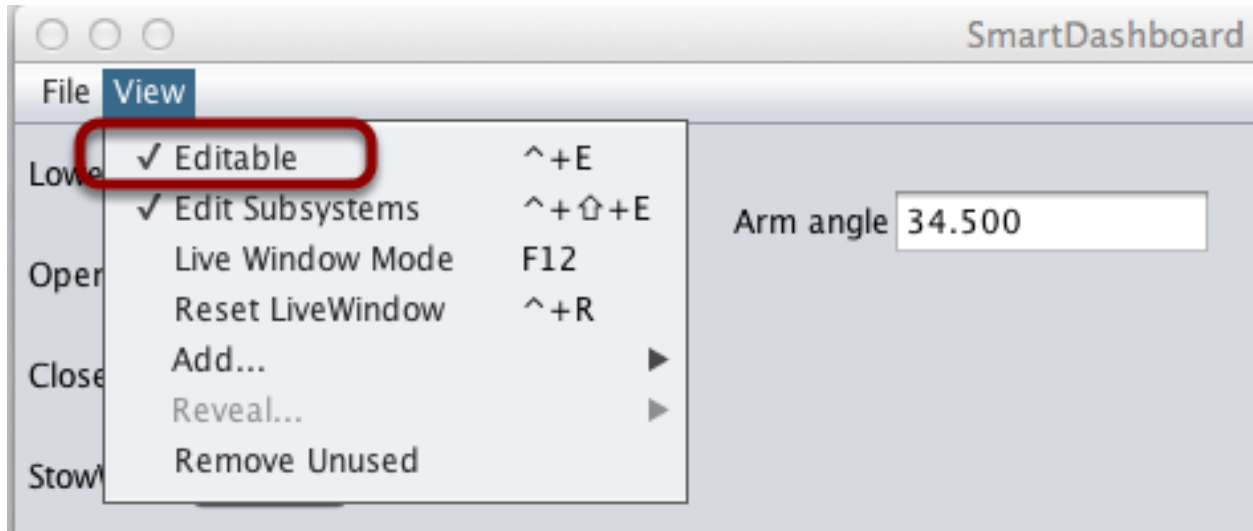
To ensure that the SmartDashboard and OutlineViewer are showing current values, it is necessary to restart the NetworkTables clients and robot at the same time. That way, old values that one is holding won't get replicated to the others.

This usually isn't a problem if the program isn't constantly changing, but if the program is in development and the set of keys being added to NetworkTables is constantly changing, then it might be necessary to do the restart of everything to accurately see what is current.

20.3 Changing the display properties of a value

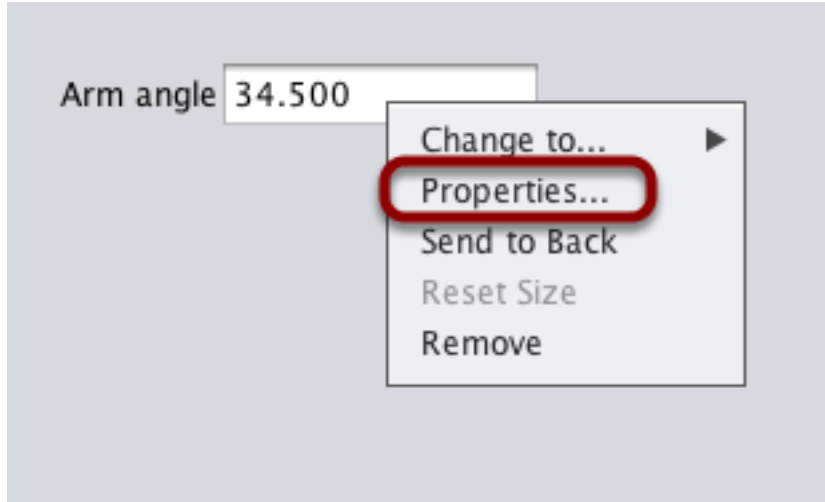
Each value displayed with SmartDashboard has a set of properties that effect the way it's displayed.

20.3.1 Setting the SmartDashboard display into editing mode



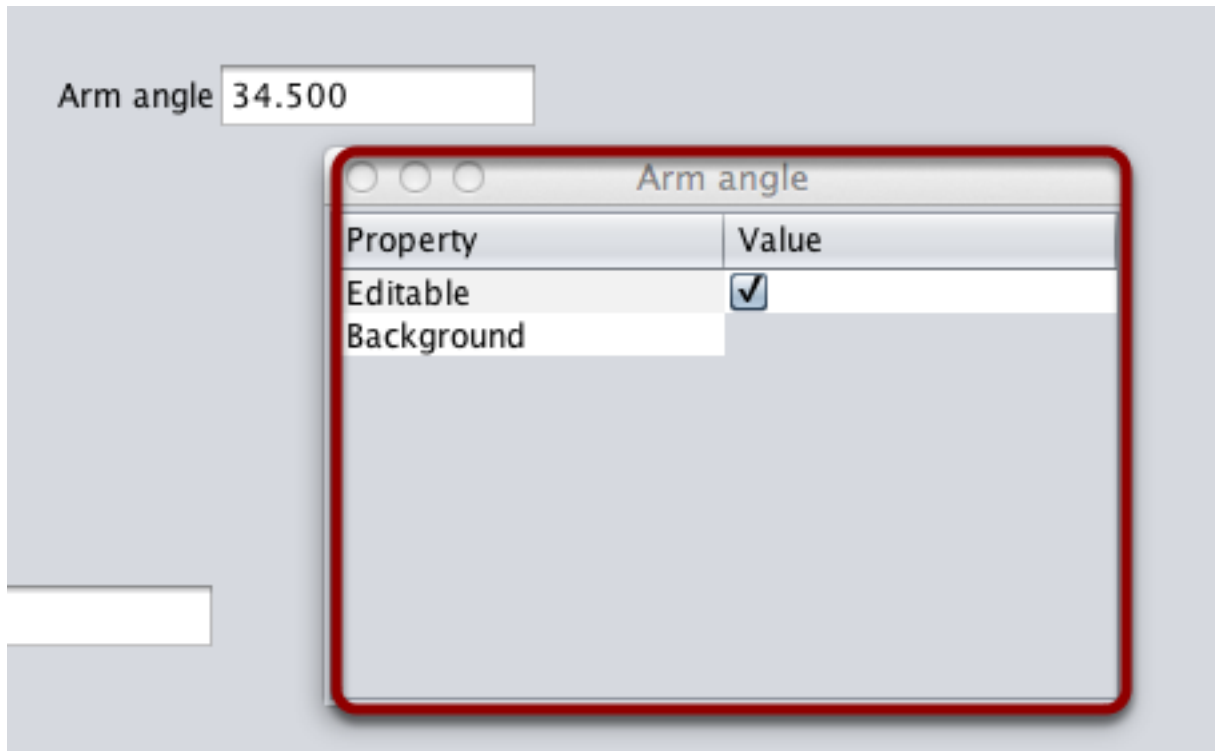
The SmartDashboard has two modes it can operate in, display mode and edit mode. In edit mode you can move around widgets on the screen and edit their properties. To put the SmartDashboard into edit mode, click the “View” menu, then select “Editable” to turn on edit mode.

20.3.2 Getting the properties editor of a widget



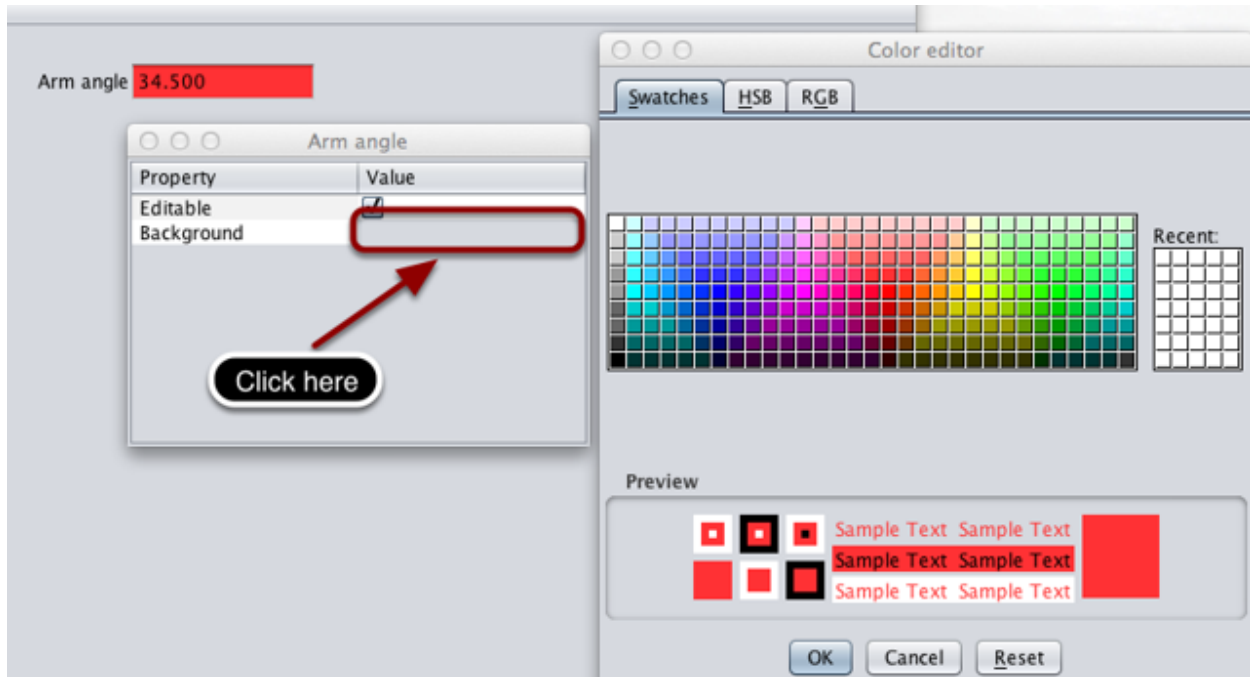
Once in edit mode, you can display and edit the properties for a widget. Right-click on the widget and select “Properties...”.

20.3.3 Editing the properties on a field



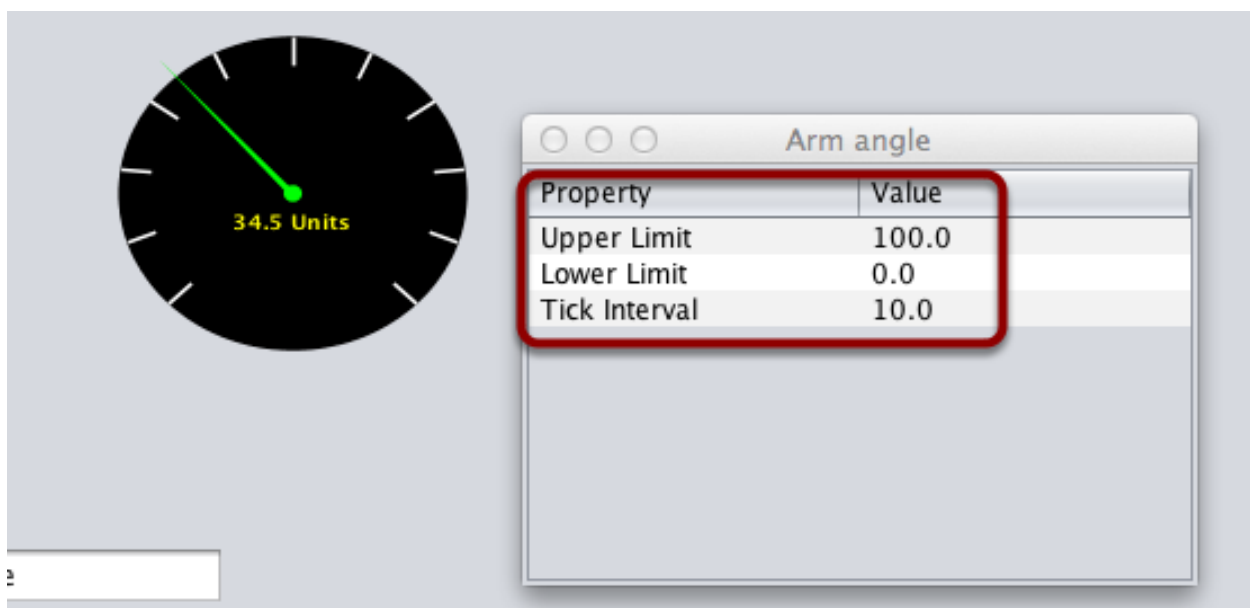
A dialog box will be shown in response to the “Properties...” menu item on the widgets right-click context menu.

20.3.4 Editing the widgets background color



To edit a property value, say, Background color, click the background color shown (in this case grey), and choose a color from the color editor that pops up. This will be used as the widgets background color.

20.3.5 Edit properties of other widgets

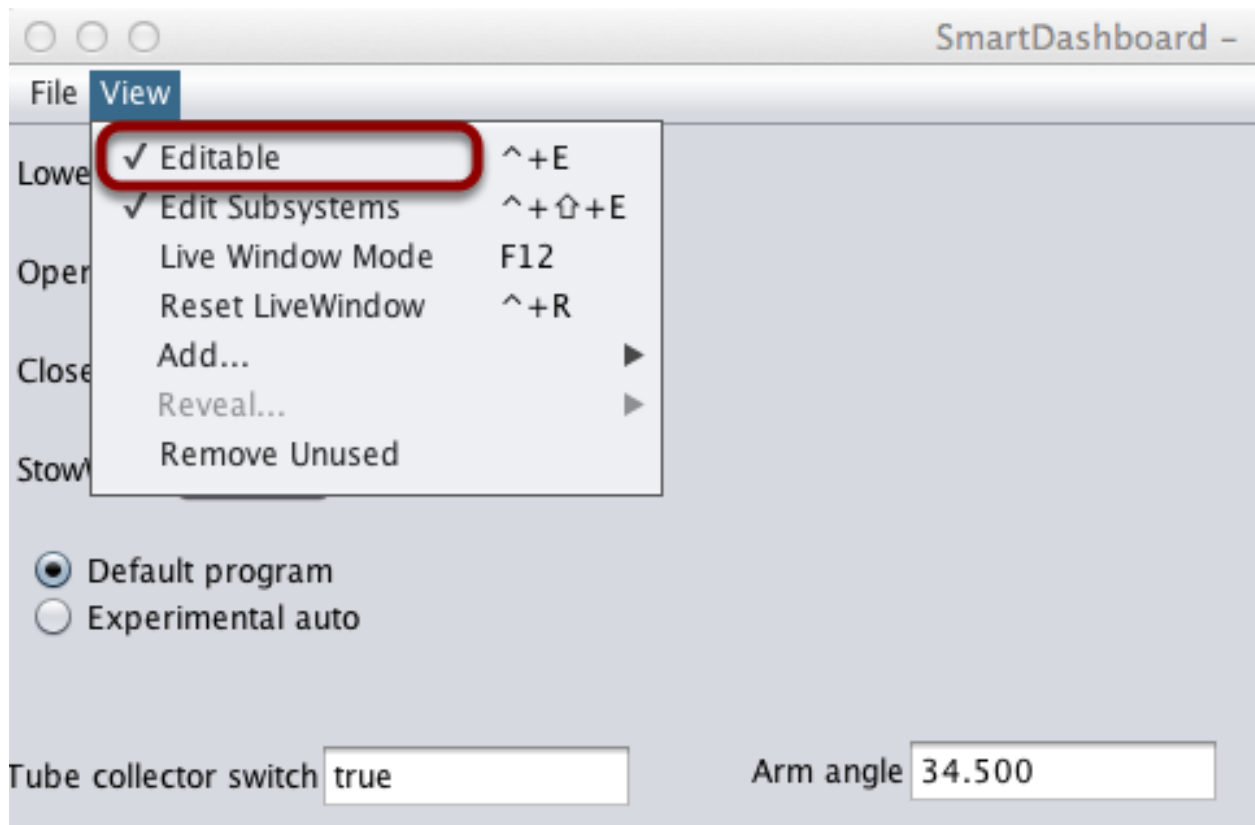


Different widget types have different sets of editable properties to change the appearance. In this example, the upper and lower limits of the dial and the tick interval are changeable parameters.

20.4 Changing the Display Widget Type for a Value

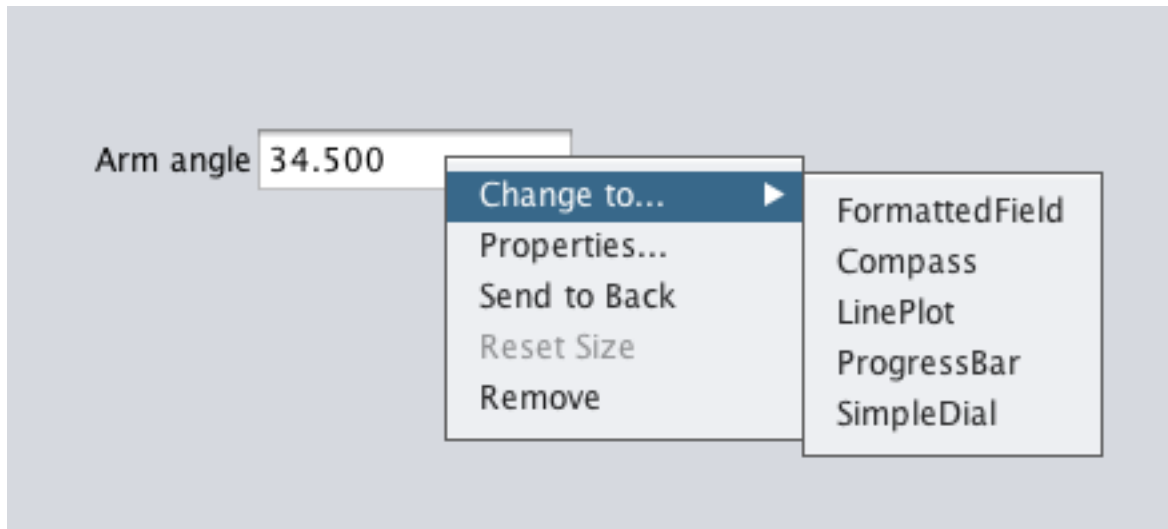
One can change the type of widget that displays values with the SmartDashboard. The allowable widgets depend on the type of the value being displayed.

20.4.1 Setting Edit Mode



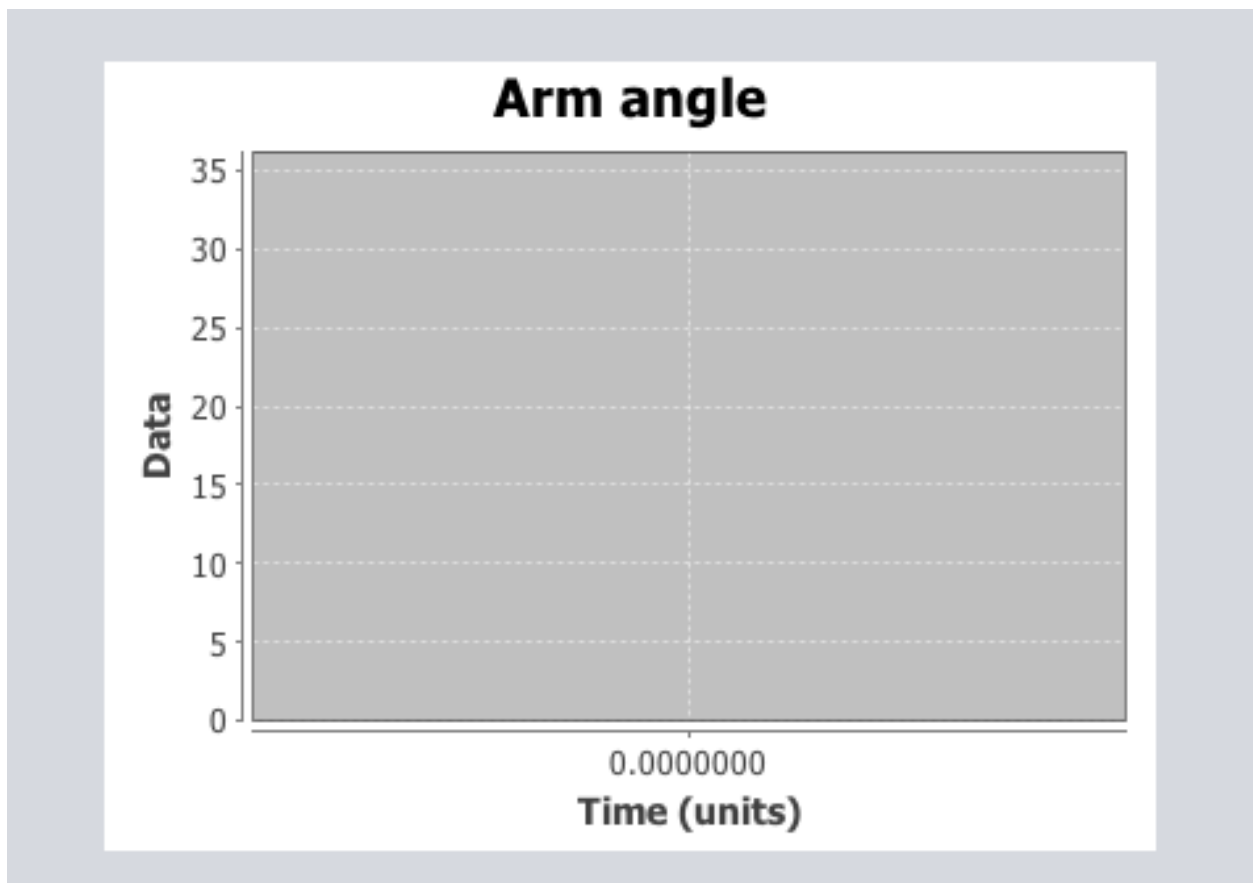
Make sure that the SmartDashboard is in edit mode. This is done by selecting `Editable` from the View menu.

20.4.2 Choosing Widget Type



Right-click on the widget and select **Change to...**. Then, pick the type of widget to use for the particular value. In this case we choose **LinePlot**.

20.4.3 Showing New Widget Type



The new widget type is displayed. In this case, a Line Plot, will show the values of the Arm angle value over time. You can set the properties of the graph to make it better fit your data by right-clicking and selecting Properties.... See: [Changing the display properties of a value](#).

20.5 Choosing an Autonomous Program

Often teams have more than one autonomous program, either for competitive reasons or for testing new software. Programs often vary by adding things like time delays, different strategies, etc. The methods to choose the strategy to run usually involves switches, joystick buttons, knobs or other hardware based inputs.

With the SmartDashboard you can simply display a widget on the screen to choose the autonomous program that you would like to run. And with command based programs, that program is encapsulated in one of several commands. This article shows how to select an autonomous program with only a few lines of code and a nice looking user interface.

Note: The code snippets shown below are part of the HatchbotTraditional example project (Java, C++):

20.5.1 Creating the SendableChooser Object

In RobotContainer, create a variable to hold a reference to a SendableChooser object. Two or more commands can be created and stored in new variables. Using the SendableChooser, one can choose between them. In this example, SimpleAuto and ComplexAuto are shown as options.

Java

C++ (Header)

```
// A simple auto routine that drives forward a specified distance, and then stops.
private final Command m_simpleAuto =
    new DriveDistance(AutoConstants.kAutoDriveDistanceInches, AutoConstants.
↪kAutoDriveSpeed,
                        m_robotDrive);

// A complex auto routine that drives forward, drops a hatch, and then drives ↪
↪backward.
private final Command m_complexAuto = new ComplexAuto(m_robotDrive, m_
↪hatchSubsystem);

// A chooser for autonomous commands
SendableChooser<Command> m_chooser = new SendableChooser<>();
```

```
// The autonomous routines
DriveDistance m_simpleAuto{AutoConstants::kAutoDriveDistanceInches,
                            AutoConstants::kAutoDriveSpeed, &m_drive};
ComplexAuto m_complexAuto{&m_drive, &m_hatch};

// The chooser for the autonomous routines
frc::SendableChooser<frc2::Command*> m_chooser;
```


20.5.2 Setting up SendableChooser

Imagine that you have two autonomous programs to choose between and they are encapsulated in commands SimpleAuto and ComplexAuto. To choose between them:

In RobotContainer, create a SendableChooser object and add instances of the two commands to it. There can be any number of commands, and the one added as a default (setDefaultOption), becomes the one that is initially selected. Notice that each command is included in an setDefaultOption() or addOption() method call on the SendableChooser instance.

Java

C++

```
// Add commands to the autonomous command chooser
m_chooser.setDefaultOption("Simple Auto", m_simpleAuto);
m_chooser.addOption("Complex Auto", m_complexAuto);

// Put the chooser on the dashboard
SmartDashboard.putData(m_chooser);
```

```
// Add commands to the autonomous command chooser
m_chooser.SetDefaultOption("Simple Auto", &m_simpleAuto);
m_chooser.AddOption("Complex Auto", &m_complexAuto);

// Put the chooser on the dashboard
frc::SmartDashboard::PutData(&m_chooser);
```

20.5.3 Starting an Autonomous Command

In Robot.java, when the autonomous period starts, the SendableChooser object is polled to get the selected command and that command must be scheduled.

Java

C++ (Source)

```
public Command getAutonomousCommand() {
    return m_chooser.getSelected();
}
```

```
public void autonomousInit() {
    m_autonomousCommand = m_robotContainer.getAutonomousCommand();
    // schedule the autonomous command (example)
    if (m_autonomousCommand != null) {
        m_autonomousCommand.schedule();
    }
}
```

```
frc2::Command* RobotContainer::GetAutonomousCommand() {
    // Runs the chosen command in autonomous
    return m_chooser.GetSelected();
}
```

```
void Robot::AutonomousInit() {
    m_autonomousCommand = m_container.GetAutonomousCommand();

    if (m_autonomousCommand != nullptr) {
        m_autonomousCommand->Schedule();
    }
}
```

20.5.4 Running the Scheduler during Autonomous

In `Robot.java`, this will run the scheduler every driver station update period (about every 20ms) and cause the selected autonomous command to run.

Note: Running the scheduler can occur in the `autonomousPeriodic()` function or `robotPeriodic()`, both will function similarly in autonomous mode.

Java

C++ (Source)

```
public void robotPeriodic() {
    CommandScheduler.getInstance().run();
}
```

```
void Robot::RobotPeriodic() { frc2::CommandScheduler::GetInstance().Run(); }
```

20.5.5 Canceling the Autonomous Command

In `Robot.java`, when the teleop period begins, the autonomous command will be canceled.

Java

C++ (Source)

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

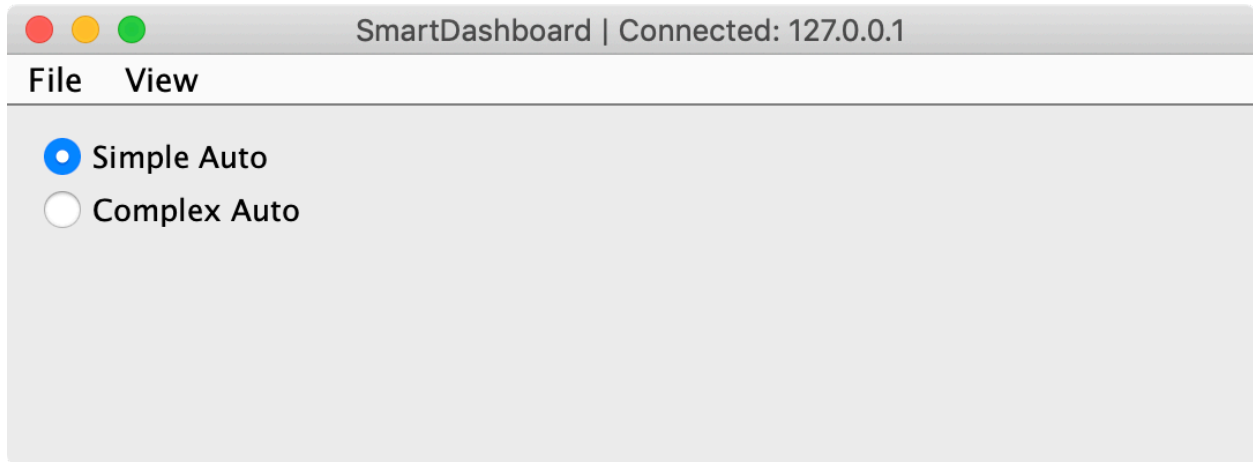
```
void Robot::TeleopInit() {
    // This makes sure that the autonomous stops running when
    // teleop starts running. If you want the autonomous to
    // continue until interrupted by another command, remove
    // this line or comment it out.
    if (m_autonomousCommand != nullptr) {
        m_autonomousCommand->Cancel();
        m_autonomousCommand = nullptr;
    }
}
```

(continues on next page)

(continued from previous page)

```
}
}
```

20.5.6 SmartDashboard Display

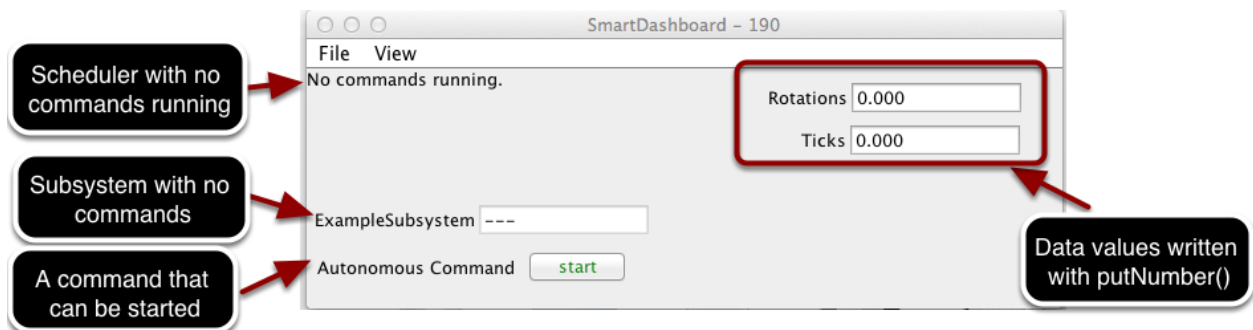


When the SmartDashboard is run, the choices from the `SendableChooser` are automatically displayed. You can simply pick an option before the autonomous period begins and the corresponding command will run.

20.6 Displaying the Status of Commands and Subsystems

If you are using the command-based programming features of WPILib, you will find that they are very well integrated with SmartDashboard. It can help diagnose what the robot is doing at any time and it gives you control and a view of what's currently running.

20.6.1 Overview of Command and Subsystem Displays



With SmartDashboard you can display the status of the commands and subsystems in your robot program in various ways. The outputs should significantly reduce the debugging time for your programs. In this picture you can see a number of displays that are possible. Displayed here are:

- The Scheduler currently with No commands running. In the next example you can see what it looks like with a few commands running showing the status of the robot.
- A subsystem, ExampleSubsystem that indicates that there are currently no commands running that are “requiring” it. When commands are running, it will indicate the name of the commands that are using the subsystem.
- A command written to SmartDashboard that shows a start button that can be pressed to run the command. This is an excellent way of testing your commands one at a time.
- And a few data values written to the dashboard to help debug the code that’s running.

In the following examples, you’ll see what the screen would look like when there are commands running, and the code that produces this display.

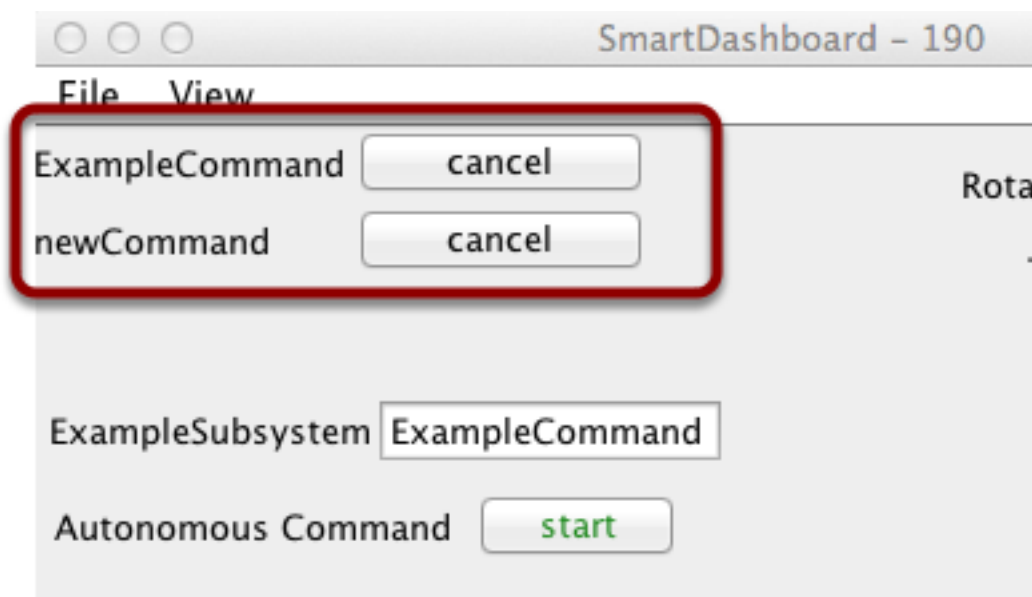
20.6.2 Displaying the Scheduler Status

```

13
14 public class TestScheduler extends IterativeRobot {
15
16     Command autonomousCommand;
17
18     public void robotInit() {
19         autonomousCommand = new ExampleCommand();
20
21         CommandBase.init();
22         SmartDashboard.putData(Scheduler.getInstance());
23     }

```

You can display the status of the Scheduler (the code that schedules your commands to run). This is easily done by adding a single line to the RobotInit method in your RobotProgram as shown here. In this example the Scheduler instance is written using the putData method to SmartDashboard. This line of code produces the display in the previous image.



This is the scheduler status when there are two commands running, ExampleCommand and newCommand. This replaces the No commands running. message from the previous screen image. You can see commands displayed on the dashboard as the program runs and various commands are triggered.

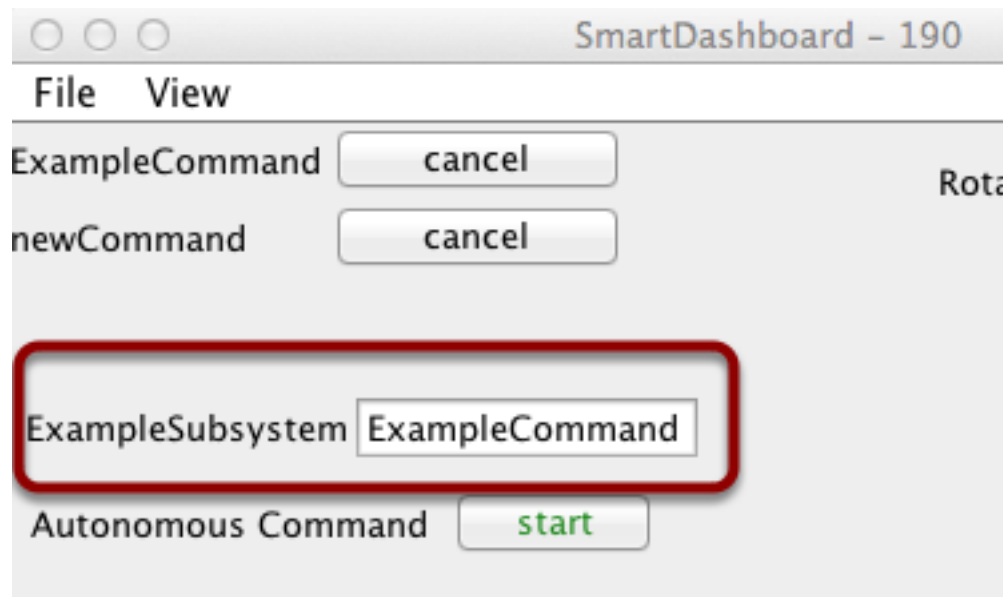
20.6.3 Displaying Subsystem Status

```

7
8
9 public abstract class CommandBase extends Command {
10
11     public static OI oi;
12     // Create a single static instance of all of your subsystems
13     public static ExampleSubsystem exampleSubsystem = new ExampleSubsystem();
14
15     public static void init() {
16         oi = new OI();
17         SmartDashboard.putData(exampleSubsystem);
18     }
19

```

In this example we are writing the command instance, exampleSubsystem and instance of the ExampleSubsystem class to the SmartDashboard. This causes the display shown in the previous image. The text field will either contain a few dashes, - - - indicating that no command is current using this subsystem, or the name of the command currently using this subsystem.

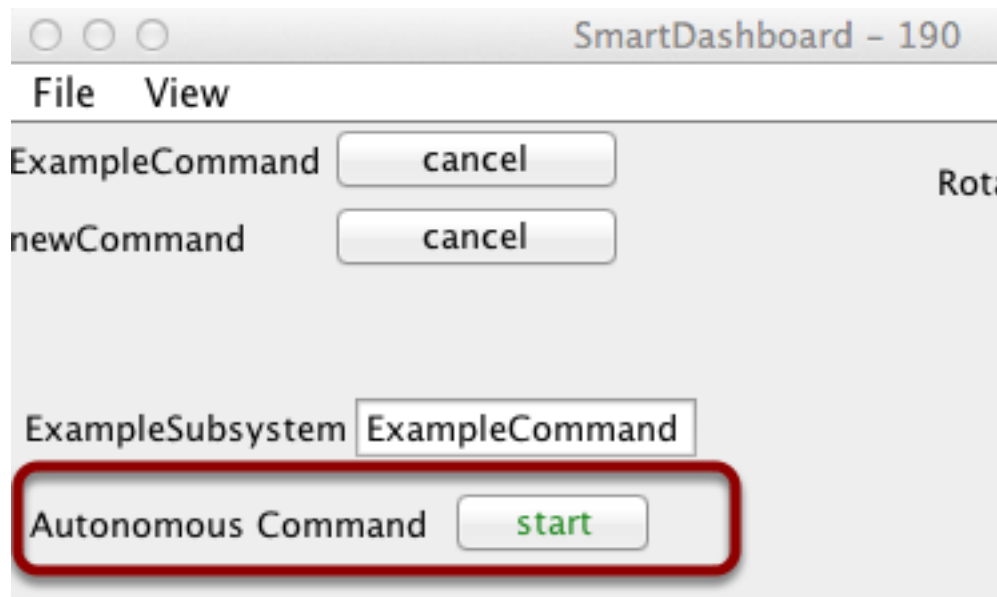


Running commands will “require” subsystems. That is the command is reserving the subsystem for its exclusive use. If you display a subsystem on SmartDashboard, it will display which command is currently using it. In this example, ExampleSubsystem is in use by ExampleCommand.

20.6.4 Activating Commands with a Button

```
17  
18  
19     public void robotInit() {  
20         autonomousCommand = new ExampleCommand();  
21  
22         CommandBase.init();  
23         SmartDashboard.putData(Scheduler.getInstance());  
24         SmartDashboard.putData("Autonomous Command", autonomousCommand);  
25     }
```

This is the code required to create a button for the command on SmartDashboard. Robot-Builder will automatically generate this code for you, but it can easily be done by hand as shown here. Pressing the button will schedule the command. While the command is running, the button label changes from start to cancel and pressing the button will cancel the command.



In this example you can see a button labeled Autonomous Command. Pressing this button will run the associated command and is an excellent way of testing commands one at a time without having to add throw-away test code to your robot program. Adding buttons for each command makes it simple to test the program, one command at a time.

20.7 Setting Robot Preferences

The Robot Preferences (Java, C++) class is used to store values in the flash memory on the roboRIO. The values might be for remembering preferences on the robot such as calibration settings for potentiometers, PID values, etc. that you would like to change without having to rebuild the program. The values can be viewed on the SmartDashboard and read and written by the robot program.

20.7.1 Reading and Writing Preferences

Java

C++

```
public class Robot extends TimedRobot {

    Preferences prefs;

    double armUpPosition;
    double armDownPosition;

    public void robotInit() {
        prefs = Preferences.getInstance();
        armUpPosition = prefs.getDouble("ArmUpPosition", 1.0);
        armDownPosition = prefs.getDouble("ArmDownPosition", 4.);
    }
}
```

```
class Robot: public TimedRobot {

    frc::Preferences *prefs;

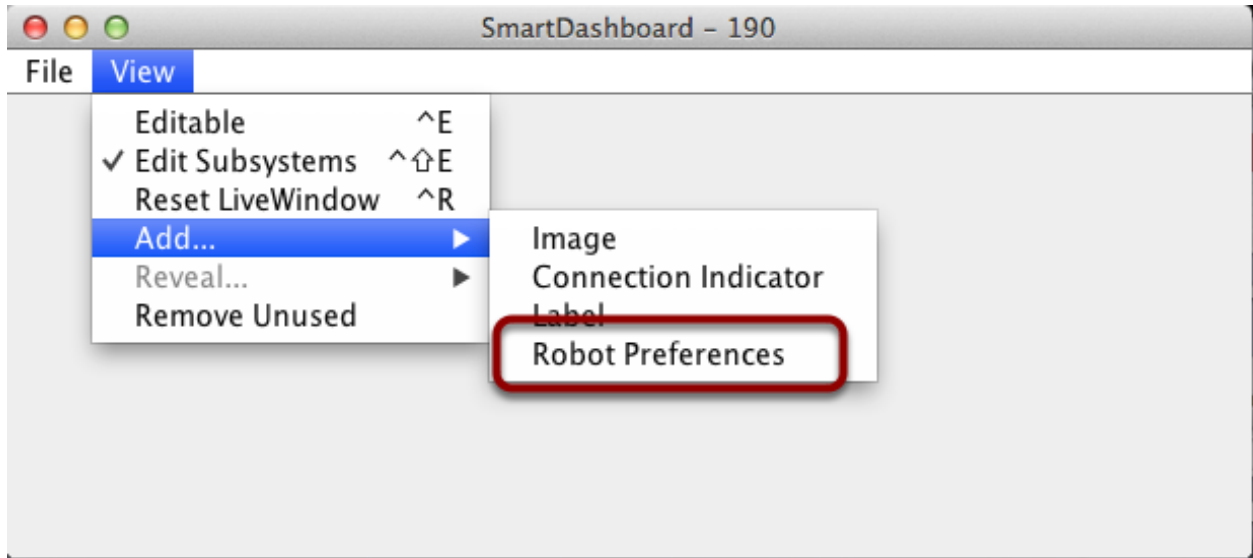
    double armUpPosition;
    double armDownPosition;

    public void RobotInit() {
        prefs = frc::Preferences::GetInstance();
        armUpPosition = prefs->GetDouble("ArmUpPosition", 1.0);
        armDownPosition = prefs->GetDouble("ArmDownPosition", 4.);
    }
}
```

Often potentiometers are used to measure the angle of an arm or the position of some other shaft. In this case, the arm has two positions, ArmUpPosition and ArmDownPosition. Usually programmers create constants in the program that are the two pot values that correspond to the positions of the arm. When the potentiometer needs to be replaced or adjusted then the program needs to be edited and reloaded onto the robot.

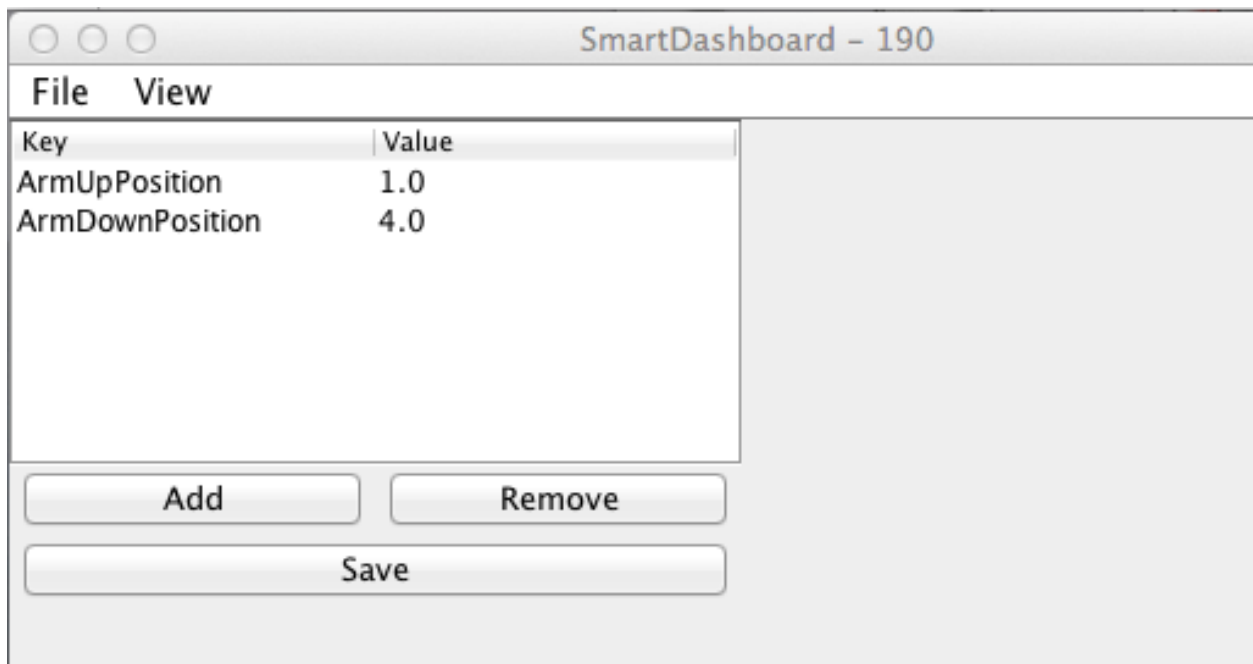
Rather than having “hard-coded” values in the program the potentiometer settings can be stored in the preferences file and read by the program when it starts. In this case the values are read on program startup in the robotInit() method. These values are automatically read from the preferences file stored in the roboRIO flash memory.

20.7.2 Displaying Preferences in SmartDashboard



In the SmartDashboard, the Preferences display can be added to the display revealing the contents of the preferences file stored in the roboRIO flash memory.

20.7.3 Viewing and Editing Preferences

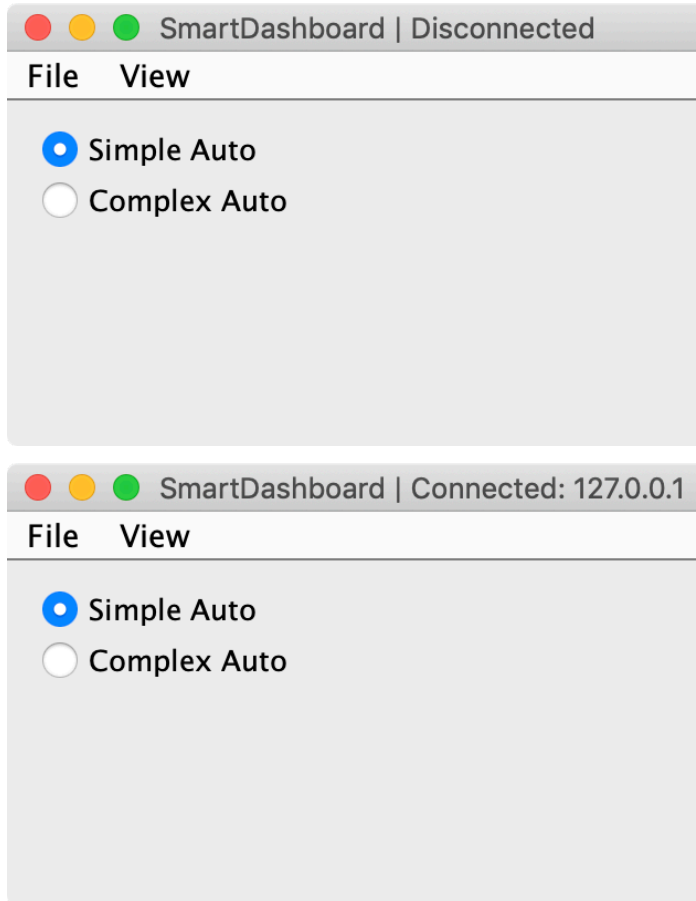


The values are shown here with the default values from the code. This was read from the robot through the NetworkTables interface built into SmartDashboard. If the values need to be adjusted they can be edited here and saved. The next time the robot program starts up the new values will be loaded in the `robotInit()` method. Each subsequent time the robot starts, the new values will be retrieved without having to edit and recompile/reload the robot program.

20.8 Verifying SmartDashboard is working

20.8.1 Connection Indicator

SmartDashboard will automatically include the connection status and IP address of the NetworkTables source in the title of the window.



20.8.2 Connection Indicator Widget

SmartDashboard includes a connection indicator widget which will turn red or green depending on the connection to NetworkTables, usually provided by the roboRIO. For instructions to add this widget, look at [Adding a Connection Indicator](#) in the SmartDashboard Intro.

20.8.3 Robot Program Example

Java

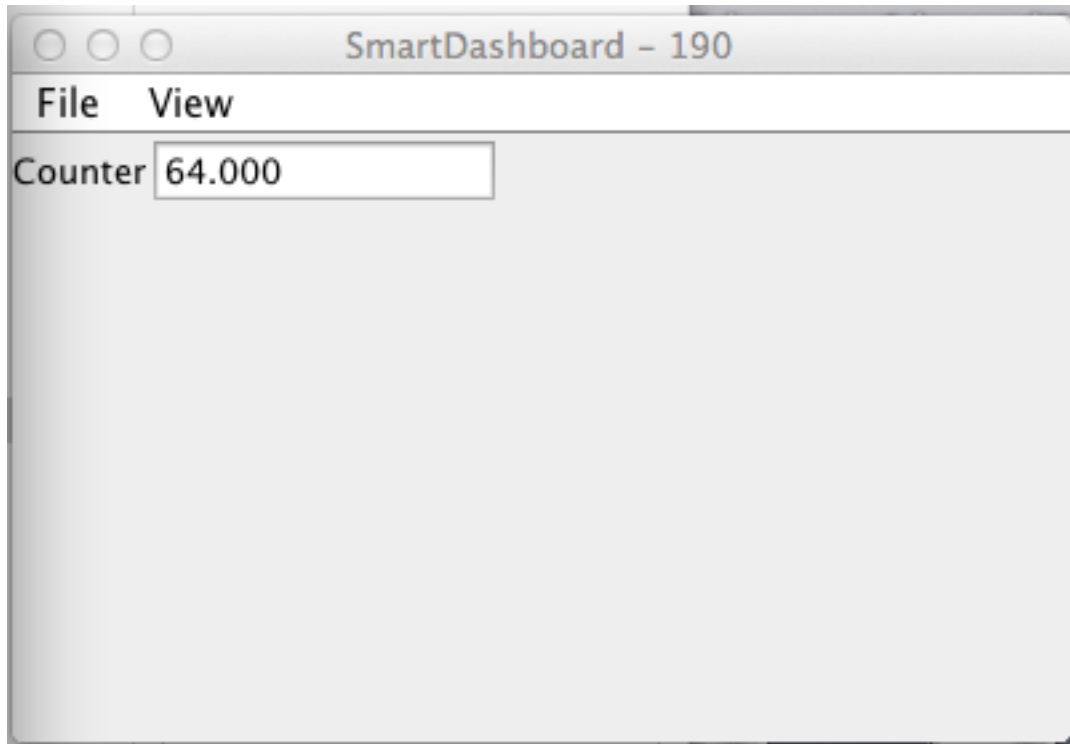
C++

```
public class Robot extends TimedRobot {  
    double counter = 0.0;  
  
    public void teleopPeriodic() {  
        SmartDashboard.putNumber("Counter", counter++);  
    }  
}
```

```
#include "Robot.h"  
float counter = 0.0;  
  
void Robot::TeleopPeriodic() {  
    frc::SmartDashboard::PutNumber("Counter", counter++);  
}
```

This is a minimal robot program that writes a value to the SmartDashboard. It simply increments a counter 50 times per second to verify that the connection is working. However, to minimize bandwidth usage, NetworkTables by default will throttle the updates to 10 times per second.

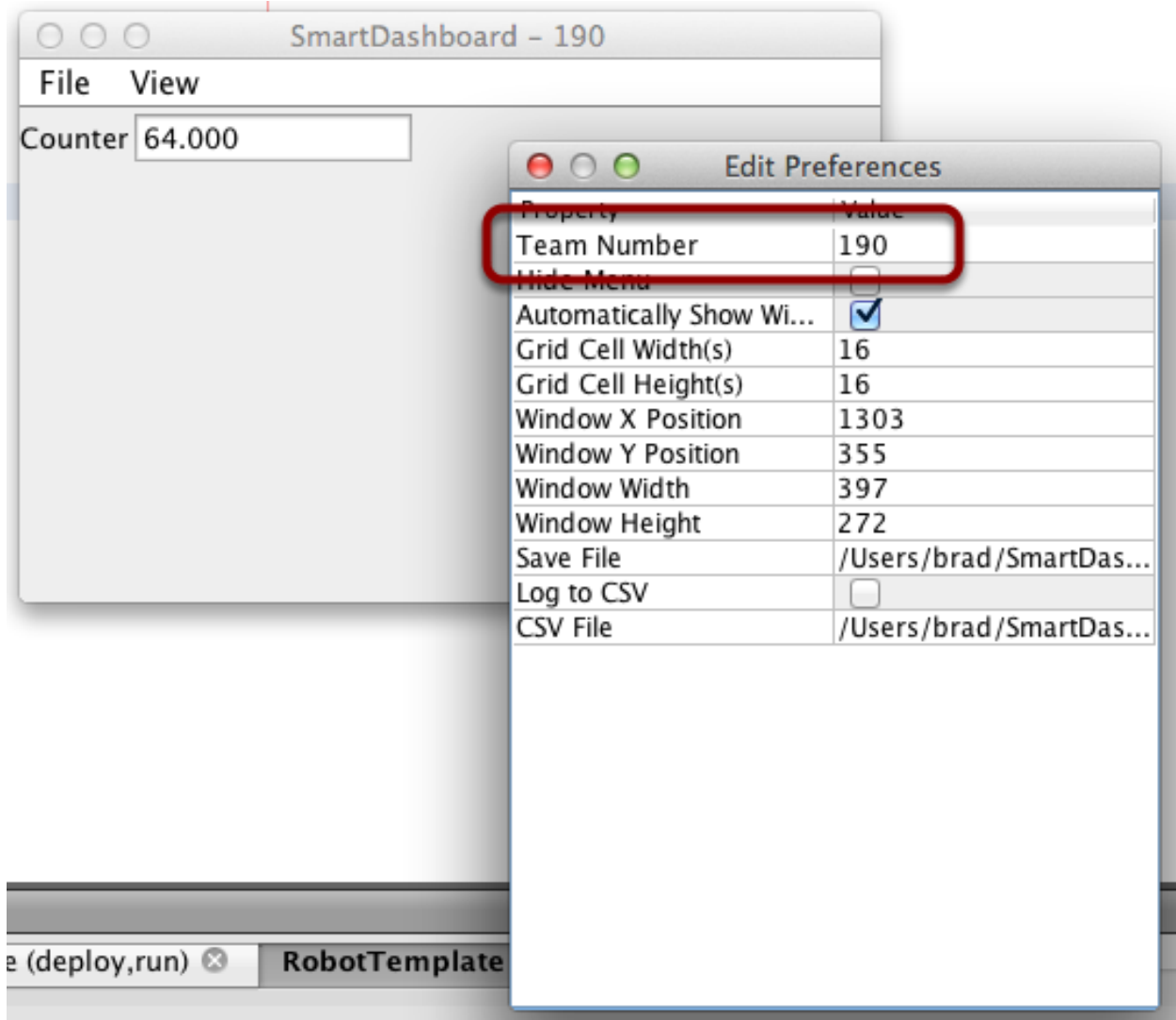
20.8.4 SmartDashboard Output for the Sample Program



The SmartDashboard display should look like this after about 6 seconds of the robot being enabled in Teleop mode. If it doesn't, then you need to check that the connection is correctly

set up.

20.8.5 Verifying the IP address in SmartDashboard



If the display of the value is not appearing, verify that the team number is correctly set as shown in this picture. The preferences dialog can be viewed by selecting File, then Preferences.

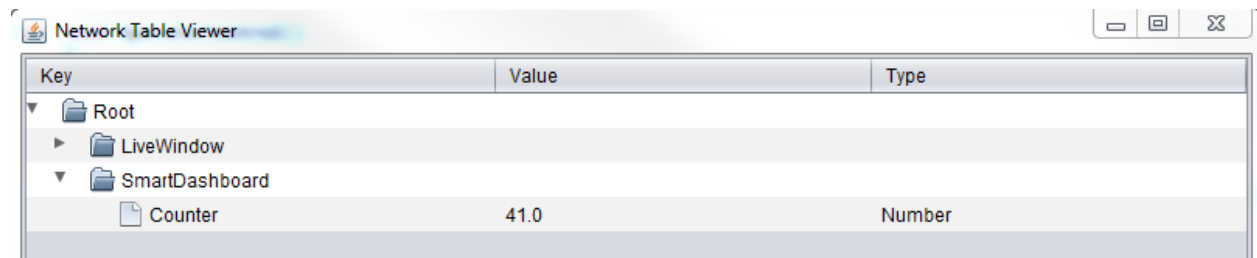
20.8.6 Verifying Program using OutlineViewer

You can verify that the robot program is generating SmartDashboard values by using the OutlineViewer program. This is a Java program, `OutlineViewer.jar`, that is located in `~/wpilib/YYYY/tools` (where YYYY is the year and ~ is `C:\Users\Public` on Windows).

OutlineViewer is downloaded as part of the WPILib Offline Installer. For more information, see the [Windows/macOS/Linux installation guides](#). In Visual Studio Code, press `Ctrl+Shift+P` and type “WPILib” or click the WPILib logo in the top right to launch the WPILib Command Palette. Select *Start Tool*, and then select *OutlineViewer*.

In the “Server Location” box, enter your team number with no leading zeroes. Then, click Start.

Look at the second row in the table, the value `SmartDashboard/Counter` is the variable written to the SmartDashboard via `NetworkTables`. As the program runs you should see the value increasing (41.0 in this case). If you don’t see this variable in the OutlineViewer, look for something wrong with the robot program or the network configuration.



20.9 SmartDashboard Namespace

SmartDashboard uses `NetworkTables` to send data between the robot and the Dashboard (Driver Station) computer. `NetworkTables` sends data as name, value pairs, like a distributed hashtable between the robot and the computer. When a value is changed in one place, its value is automatically updated in the other place. This mechanism and a standard set of name (keys) is how data is displayed on the SmartDashboard.

There is a hierarchical structure in the name space creating a set of tables and subtables. SmartDashboard data is in the `SmartDashboard` subtable and `LiveWindow` data is in the `LiveWindow` subtable as shown below.

For informational purposes, the names and values can be displayed using the OutlineViewer application that is installed in the same location as the SmartDashboard. It will display all the `NetworkTables` keys and values as they are updated.

20.9.1 SmartDashboard Data Values

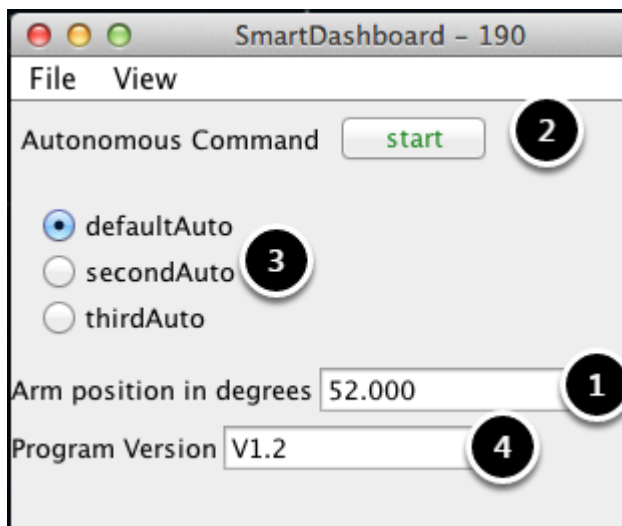
/SmartDashboard/Arm position in degrees	52.0	1
/SmartDashboard/Autonomous Command/~TYPE~	Command	2
/SmartDashboard/Autonomous Command/isParented	false	
/SmartDashboard/Autonomous Command/name	AutonomousCommand	
/SmartDashboard/Autonomous Command/running	false	3
/SmartDashboard/Chooser/~TYPE~	String Chooser	
/SmartDashboard/Chooser/default	defaultAuto	
/SmartDashboard/Chooser/options	[defaultAuto, secondAuto, th	4
/SmartDashboard/Program Version	V1.2	

SmartDashboard values are created with key names that begin with SmartDashboard/. The above values viewed with OutlineViewer correspond to data put to the SmartDashboard with the following statements:

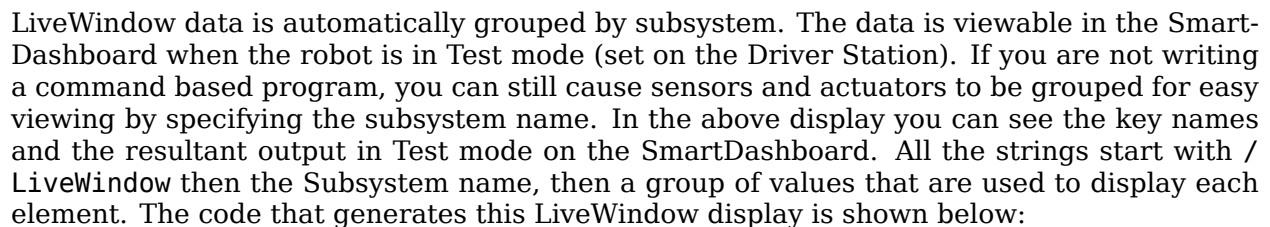
```
chooser = new SendableChooser();
chooser.setDefaultOption("defaultAuto", new AutonomousCommand());
chooser.addOption("secondAuto", new AutonomousCommand());
chooser.addOption("thirdAuto", new AutonomousCommand());
SmartDashboard.putData("Chooser", chooser);
SmartDashboard.putNumber("Arm position in degrees", 52.0);
SmartDashboard.putString("Program Version", "V1.2");
```

The Arm position is created with the putNumber() call. The AutonomousCommand is written with a putData("Autonomous Command", command) that is not shown in the above code fragment. The chooser is created as a SendableChooser object and the string value, Program Version is created with the putString() call.

20.9.2 View of SmartDashboard



The code from the previous step generates the table values as shown and the SmartDashboard display as shown here. The numbers correspond to the NetworkTables variables shown in the previous step.



(continues on next page)

(continued from previous page)

```
clawMotor = new PWMVictorSPX(5);
clawMotor.setName("Claw", "Motor");
```

Values that correspond to actuators are not only displayed, but can be set using sliders created in the SmartDashboard in Test mode.

20.10 SmartDashboard: Test Mode and Live Window

20.10.1 Displaying LiveWindow Values

Note: LiveWindow will automatically add your sensors for you. There is no need to do it manually.

LiveWindow values may also be displayed by writing the code yourself and adding it to your robot program. LiveWindow will display values grouped in subsystems. This is a convenient method of displaying whether they are actual command based program subsystems or just a grouping that you decide to use in your program.

Adding the Necessary Code to your Program

1. First, get a reference (Java) or pointer (C++) to the LiveWindow object.

Java

C++

```
LiveWindow lw = LiveWindow.getInstance();
```

```
frc::LiveWindow lw = frc::LiveWindow::GetInstance();
```

2. Then for each sensor or actuator that is created, add it to the LiveWindow display by either calling AddActuator or AddSensor (addActuator or addSensor in Java). When the SmartDashboard is put into LiveWindow mode, it will display the sensors and actuators.

Java

C++

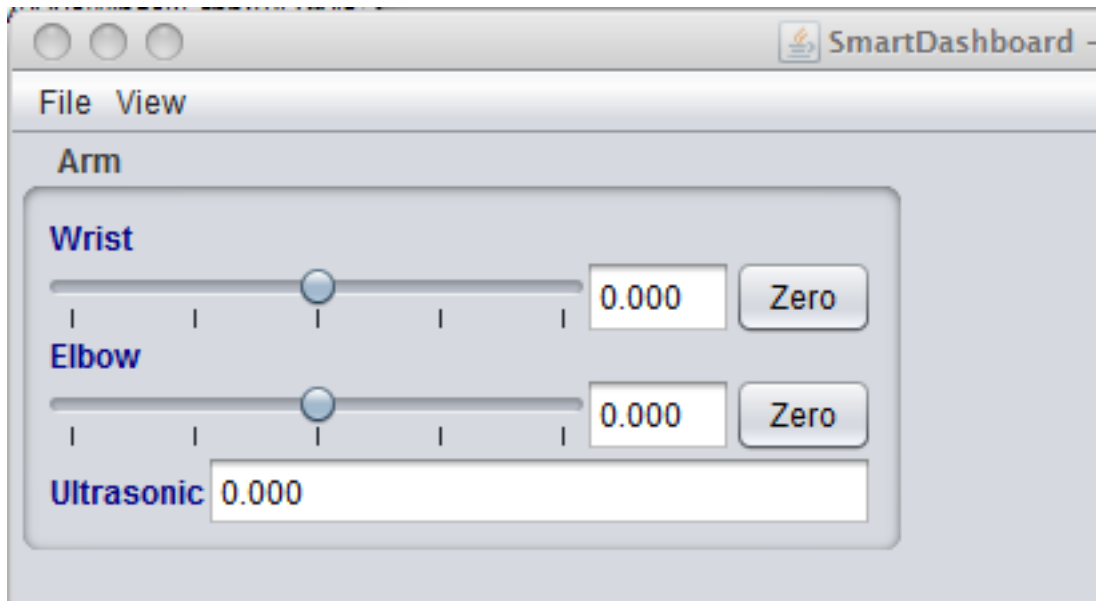
```
Ultrasonic ultrasonic = new Ultrasonic(1, 2);
lw.addSensor("Arm", "Ultrasonic", ultrasonic);
```

```
Jaguar elbow = new Jaguar(1);
lw.addActuator("Arm", "Elbow", elbow);
```

```
Victor wrist = new Victor(2);
lw.addActuator("Arm", "Wrist", wrist);
```

```
frc::Ultrasonic ultrasonic{1, 2};  
lw->AddSensor("Arm", "Ultrasonic", ultrasonic);  
  
frc::Jaguar elbow{1};  
lw->AddActuator("Arm", "Elbow", elbow);  
  
frc::Victor wrist{2};  
lw->AddActuator("Arm", "Wrist", wrist);
```

Viewing the Display in SmartDashboard

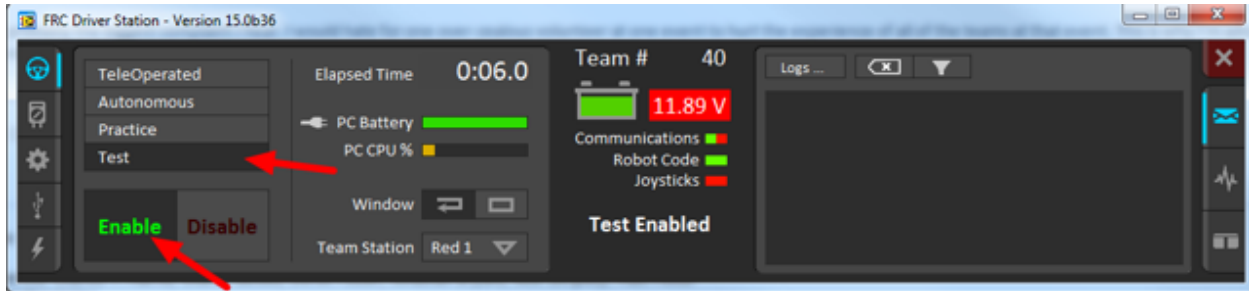


The sensors and actuators added to the LiveWindow will be displayed grouped by subsystem. The subsystem name is just an arbitrary grouping the helping to organize the display of the sensors. Actuators can be operated by operating the slider for the two motor controllers.

20.10.2 Enabling Test mode (LiveWindow)

You may add code to your program to display values for your sensors and actuators while the robot is in Test mode. This can be selected from the Driver Station whenever the robot is not on the field. The code to display these values is automatically generated by RobotBuilder and is described in the next article. Test mode is designed to verify the correct operation of the sensors and actuators on a robot. In addition it can be used for obtaining setpoints from sensors such as potentiometers and for tuning PID loops in your code.

Setting Test mode with the Driver Station



Enable Test Mode in the Driver Station by clicking on the “Test” button and setting “Enable” on the robot. When doing this, the SmartDashboard display will switch to test mode (LiveWindow) and will display the status of any actuators and sensors used by your program.

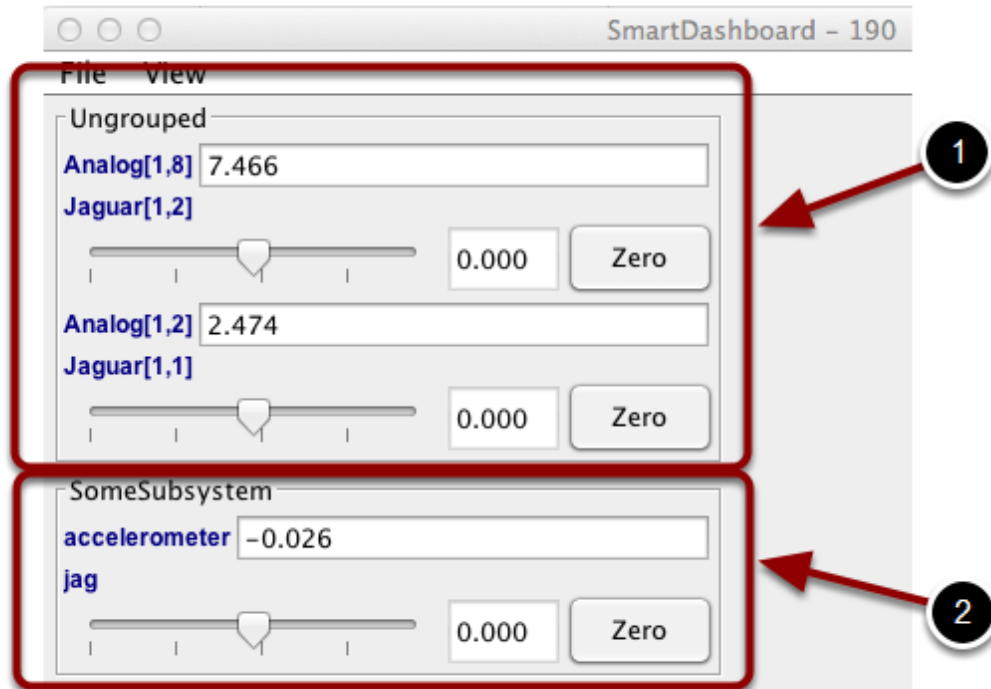
Explicitly vs. implicit test mode display

```
RobotDrive drive = new RobotDrive(1, 2);
Jaguar jag;
Accelerometer accel = new Accelerometer(1, 2);

public void robotInit() {
    jag = new Jaguar(3);
    drive.setSafetyEnabled(false);
    LiveWindow.addActuator("SomeSubsystem", "jag", jag);
    LiveWindow.addSensor("SomeSubsystem", "accelerometer", accel);
    SmartDashboard.putData("TestPID", new PIDController(1, 1, 1, accel, jag));
    SmartDashboard.putNumber("X", 0.0);
}
```

All sensors and actuators will automatically be displayed on the SmartDashboard in test mode and will be named using the object type (such as Jaguar, Analog, Victor, etc.) with the module number and channel number with which the object was created. In addition, the program can explicitly add sensors and actuators to the test mode display, in which case programmer-defined subsystem and object names can be specified making the program clearer. This example illustrates explicitly defining those sensors and actuators in the highlighted code.

Understanding what is displayed in Test mode

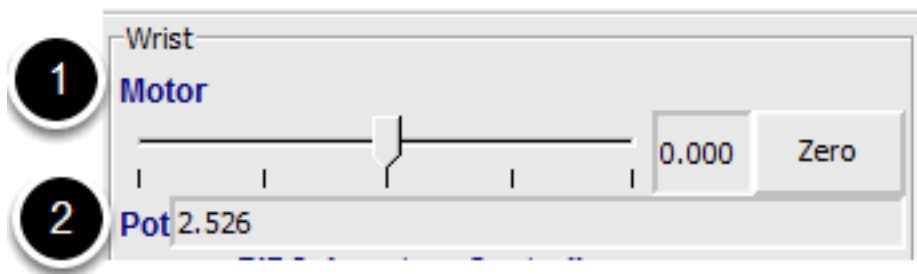


This is the output in the SmartDashboard display when the robot is placed into test mode. In the display shown above the objects listed as Ungrouped were implicitly created by WPILib when the corresponding objects were created. These objects are contained in a subsystem group called “Ungrouped” **(1)** and are named with the device type (Analog, Jaguar in this case), and the module and channel numbers. The objects shown in the “SomeSubsystem” **(2)** group are explicitly created by the programmer from the code example in the previous section. These are named in the calls to `LiveWindow.addActuator()` and `LiveWindow.AddSensor()`. Explicitly created sensors and actuators will be grouped by the specified subsystem.

20.10.3 PID Tuning with SmartDashboard

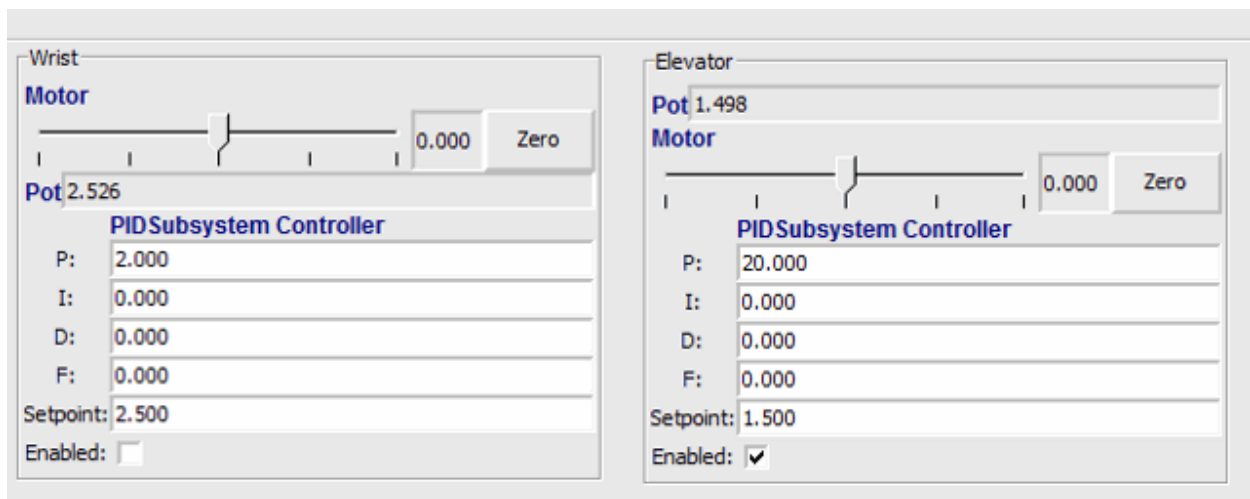
The PID (Proportional, Integral, Differential) is an algorithm for determining the motor speed based on sensor feedback to reach a setpoint as quickly as possible. For example, a robot with an elevator that moves to a predetermined position should move there as fast as possible then stop without excessive overshoot leading to oscillation. Getting the PID controller to behave this way is called “tuning”. The idea is to compute an error value that is the difference between the current value of the mechanism feedback element and the desired (setpoint) value. In the case of the arm, there might be a potentiometer connected to an analog channel that provides a voltage that is proportional to the position of the arm. The desired value is the voltage that is predetermined for the position the arm should move to, and the current value is the voltage for the actual position of the arm.

Finding the setpoint values with LiveWindow



Create a PID Subsystem for each mechanism with feedback. The PID Subsystems contain the actuator (motor) and the feedback sensor (potentiometer in this case). You can use Test mode to display the subsystem sensors and actuators. Using the slider manually adjust the actuator to each desired position. Note the sensor values (2) for each of the desired positions. These will become the setpoints for the PID controller.

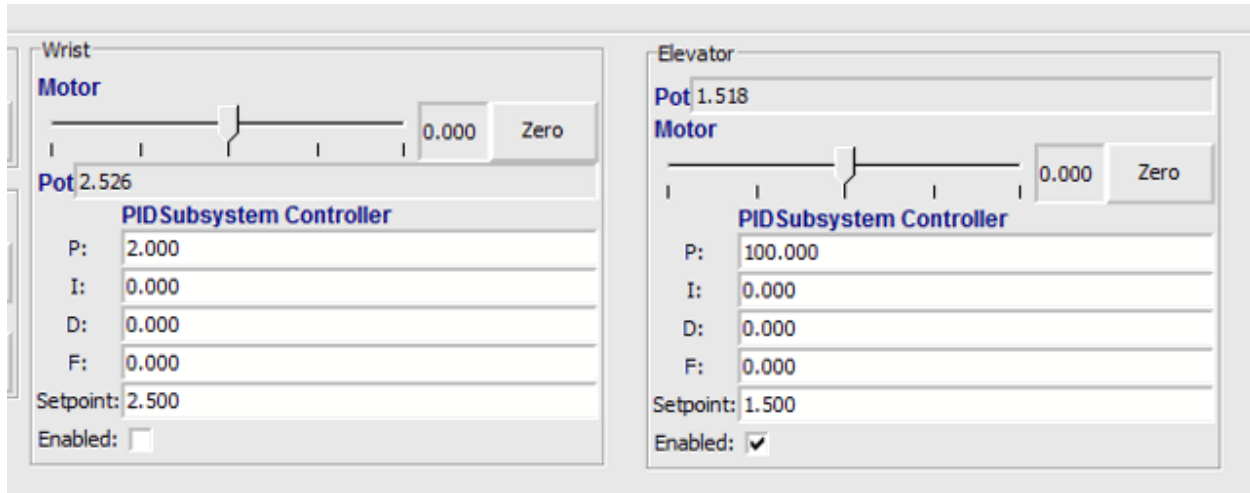
Viewing the PIDController in LiveWindow



In Test mode, the PID Subsystems display their P, I, and D parameters that are set in the code. The P, I, and D values are the weights applied to the computed error (P), sum of errors over time (I), and the rate of change of errors (D). Each of those terms is multiplied by the weights and added together to form the motor value. Choosing the optimal P, I, and D values can be difficult and requires some amount of experimentation. The Test mode on the robot allows the values to be modified, and the mechanism response observed.

Important: The enable option does not affect the [PIDController](#) introduced in 2020, as the controller is updated every robot loop. See the example [here](#) on how to retain this functionality.

Tuning the PIDController



Tuning the PID controller can be difficult and there are many articles that describe techniques that can be used. It is best to start with the P value first. To try different values fill in a low number for P, enter a setpoint determined earlier in this document, and note how fast the mechanism responds. If it responds too slowly, perhaps never reaching the setpoint, increase P. If it responds too quickly, perhaps oscillating, reduce the P value. Repeat this process until you get a response that is as fast as possible without oscillation. It's possible that having a P term is all that's needed to achieve adequate control of your mechanism. Further information is located in the [Tuning a PID Controller](#) document.

Once you have determined P, I, and D values they can be inserted into the program. You'll find them either in the properties for the PIDSubsystem in RobotBuilder or in the constructor for the PID Subsystem in your code.

The F (feedforward) term is used for controlling velocity with a PID controller.

More information can be found at [PID Control in WPILib](#).

Glass is a new dashboard and robot data visualization tool. Its GUI is extremely similar to that of the *Simulation GUI*. In its current state, it is meant to be used as a programmer's tool rather than a proper dashboard in a competition environment.

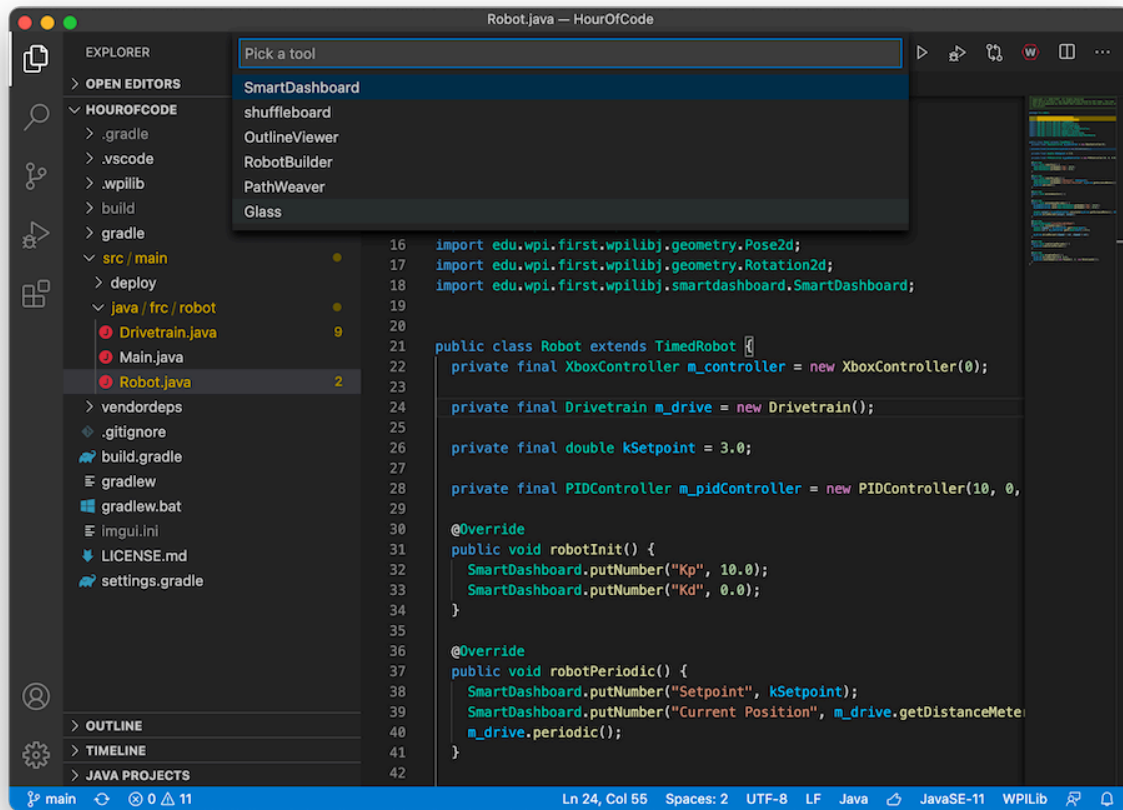
Note: Glass will not be available within the list of dashboards in the NI Driver Station for 2021.

21.1 Introduction to Glass

Glass is a new dashboard and robot data visualization tool. It supports many of the same *widgets* that the Simulation GUI supports, including robot pose visualization and advanced plotting. In its current state, it is meant to be used as a programmer's tool for debugging and not as a dashboard for competition use.

21.1.1 Opening Glass

Glass can be launched by selecting the ellipsis menu (...) in VS Code, clicking on *Start Tool* and then choosing *Glass*.

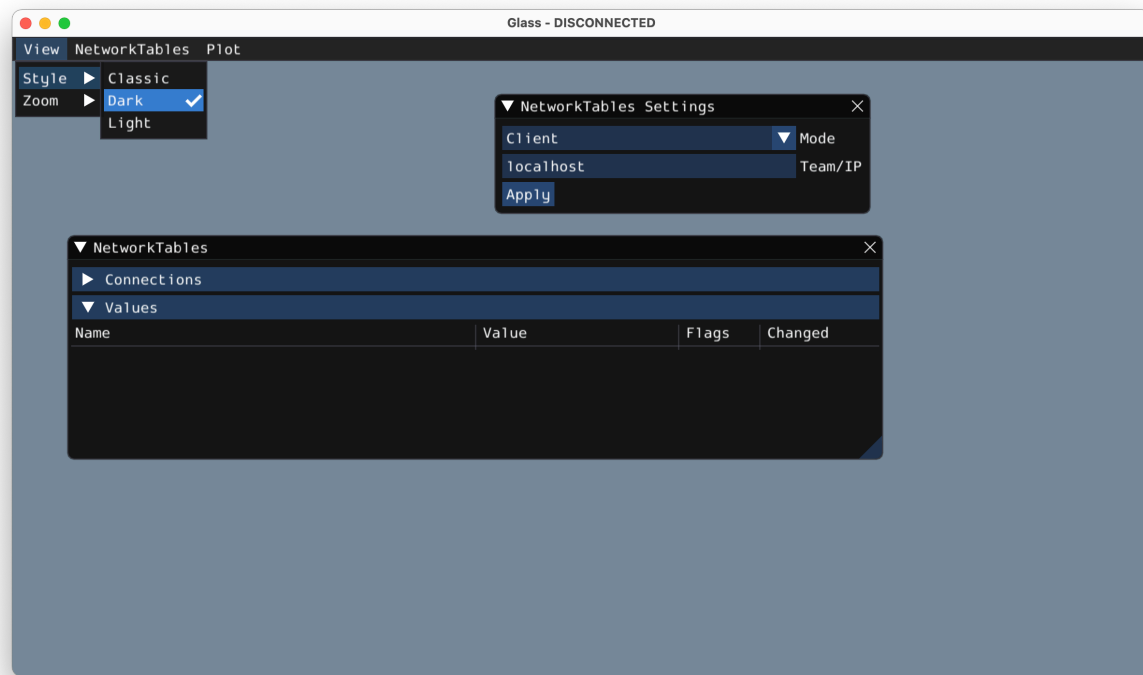


Note: You can also launch Glass directly by navigating to `~/wpilib/YYYY/tools` and running `Glass.py` (Linux and macOS) or by using the desktop shortcut (Windows).

21.1.2 Changing View Settings

The *View* menu item contains *Zoom* and *Style* settings that can be customized. The *Zoom* option dictates the size of the text in the application whereas the *Style* option allows you to select between the Classic, Light, and Dark modes.

An example of the Dark style setting is below:



Note: In Glass v2021.2.1 and below, the default zoom setting of 100% may cause text to appear too big on certain macOS Retina displays. Please reduce the zoom level to 75% or 50% or upgrade to v2021.2.2 or later to mitigate this issue.

21.1.3 Clearing Application Data

Application data for Glass, including widget sizes and positions as well as other custom information for widgets is stored in a `glass.ini` file. The location of this file varies based on your operating system:

- On Windows, the configuration file is located in `%APPDATA%`.
- On macOS, the configuration file is located in `~/Library/Preferences`.
- On Linux, the configuration file is located in `$XDG_CONFIG_HOME` or `~/.config` if the former does not exist.

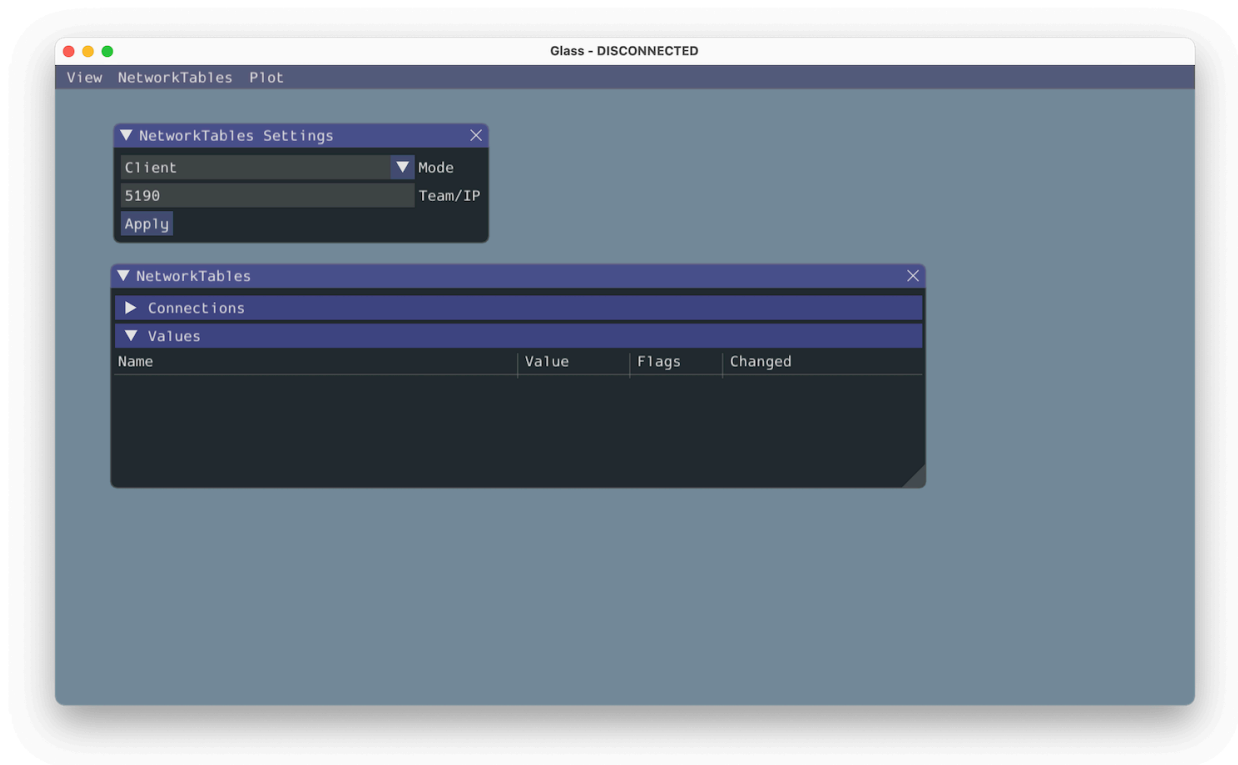
The `glass.ini` configuration file can simply be deleted to restore Glass to a “clean slate”.

21.2 Establishing NetworkTables Connections

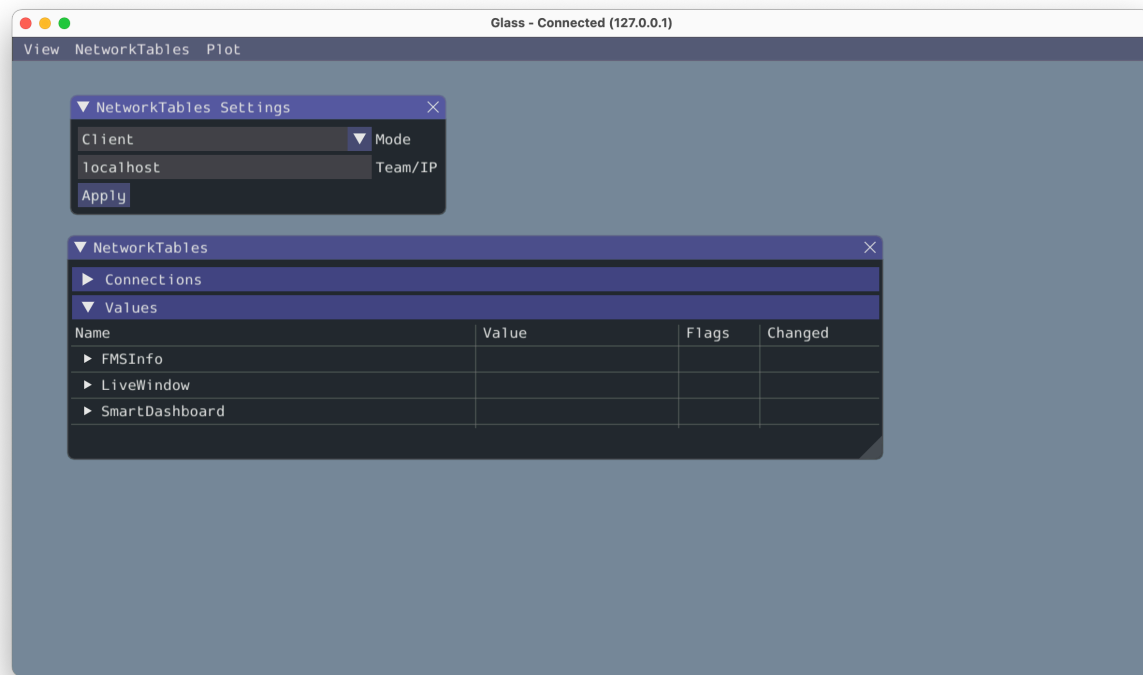
Glass uses the *NetworkTables* protocol to establish a connection with your robot program. It is also used to transmit and receive data to and from the robot.

21.2.1 Connecting to a Robot

When Glass is first launched, you will see two widgets – *NetworkTables Settings* and *NetworkTables*. To connect to a robot, select *Client* under *Mode* in the *NetworkTables Settings* widget, enter your team number and click on *Apply*.



You can also connect to a robot that is running in simulation on your computer (including Romi robots) by typing in `localhost` into the *Team/IP* box.



Important: The NetworkTables connection status is always visible on the title bar of the Glass application.

21.2.2 Viewing NetworkTables Entries

The *NetworkTables* widget can be used to view all entries that are being sent over NetworkTables. These entries are hierarchically arranged by main table, sub-table, and so on.

Name	Value	Flags	Changed
▶ FMSInfo			
▼ LiveWindow			
▶ .status			
▼ Ungrouped			
.type	LW Subsystem	string	42411866'
▶ AnalogGyro[0]			
▶ DifferentialDrive[1]			
▶ DigitalInput[0]			
▶ DigitalInput[1]			
▶ DigitalInput[2]			
▶ DigitalInput[3]			
▶ Encoder[0]			
▶ Encoder[2]			
▶ PWMVictorSPX[0]			
▶ PWMVictorSPX[1]			
▼ PWMVictorSPX[2]			
.actuator	true	boolean	42411866'
.name	PWMVictorSPX[2]	string	42411866'
.type	Speed Controller	string	42411866'
Value	0.000000	double	42411866'
▶ PWMVictorSPX[3]			
▶ Scheduler			
▶ frc2::SubsystemBase			

Furthermore, you can view all connected NetworkTables clients under the *Connections* pane of the widget.

21.3 Glass Widgets

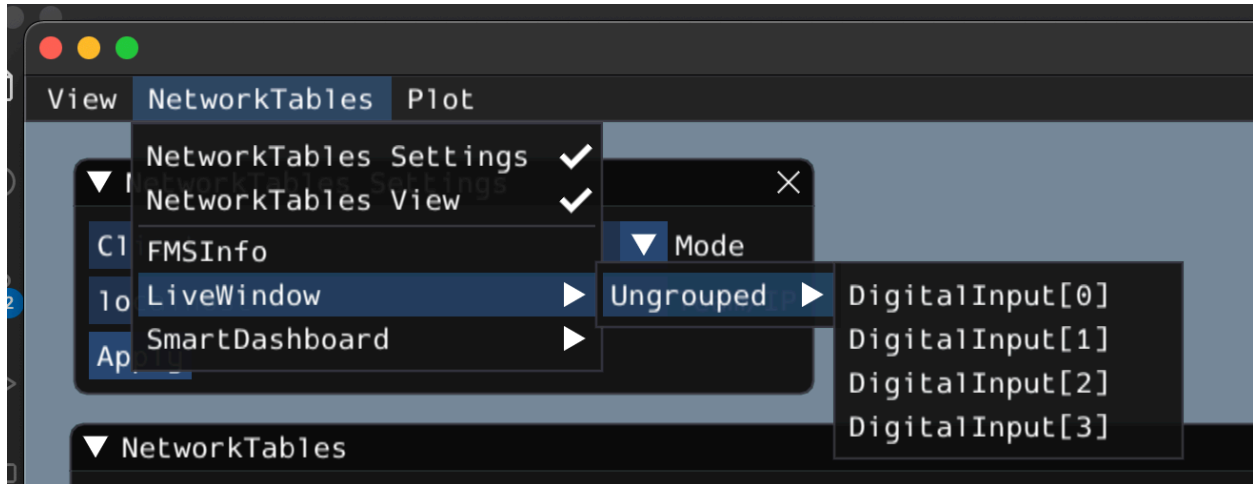
Specialized widgets are available for certain types that exist in robot code. These include objects that are manually sent over NetworkTables such as *SendableChooser* instances, or hardware that is automatically sent over *LiveWindow*.

Note: Widget support in Glass is still in its infancy – therefore, there are only a handful of widgets available. This list will grow as development work continues.

Note: A widget can be renamed by right-clicking on its header and specifying a new name.

21.3.1 Hardware Widgets

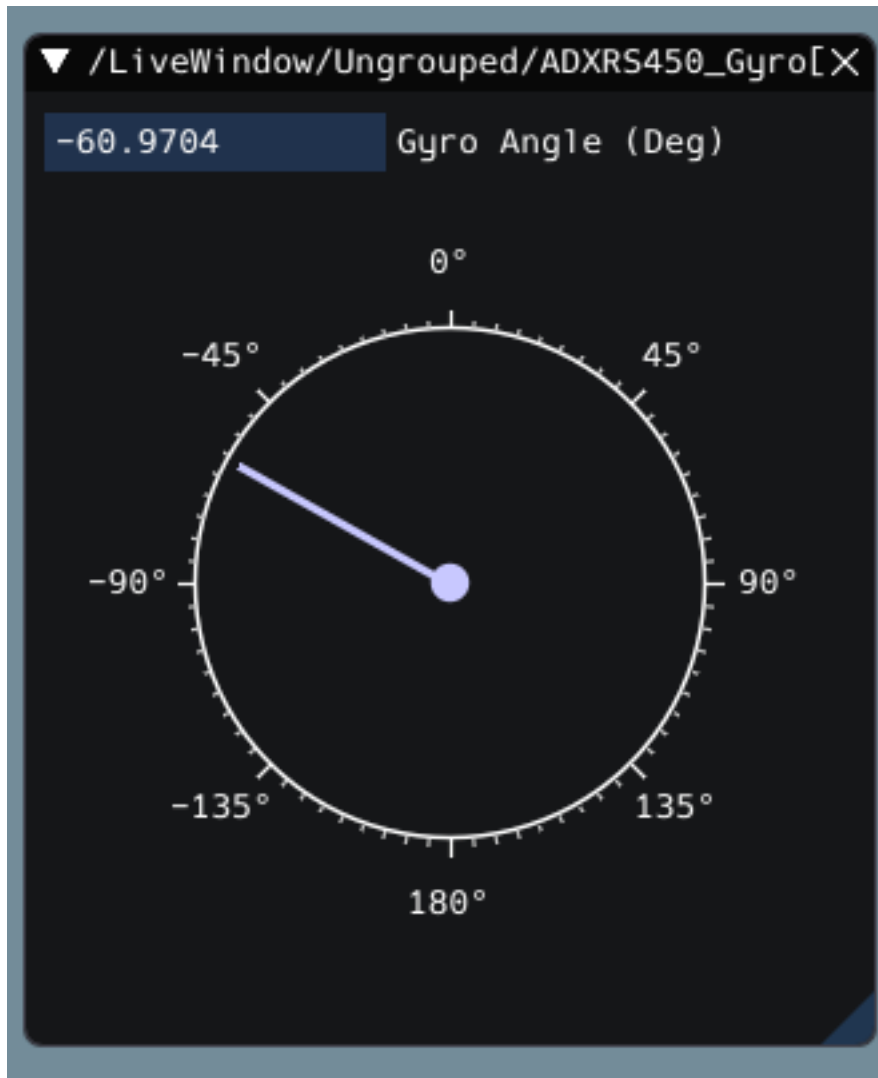
Widgets for specific hardware (such as motor controllers) are usually available via LiveWindow. These can be accessed by selecting the *NetworkTables* menu option, clicking on *LiveWindow* and choosing the desired widget.



The list of hardware (sent over LiveWindow automatically) that has widgets is below:

- DigitalInput
- DigitalOutput
- SpeedController
- Gyro

Here is an example of the widget for gyroscopes:



21.3.2 Sendable Chooser Widget

The *Sendable Chooser* widget represents a `SendableChooser` instance from robot code. It is often used to select autonomous modes. Like other dashboards, your `SendableChooser` instance simply needs to be sent using a `NetworkTables` API. The simplest is to use something like `SmartDashboard`:

Java

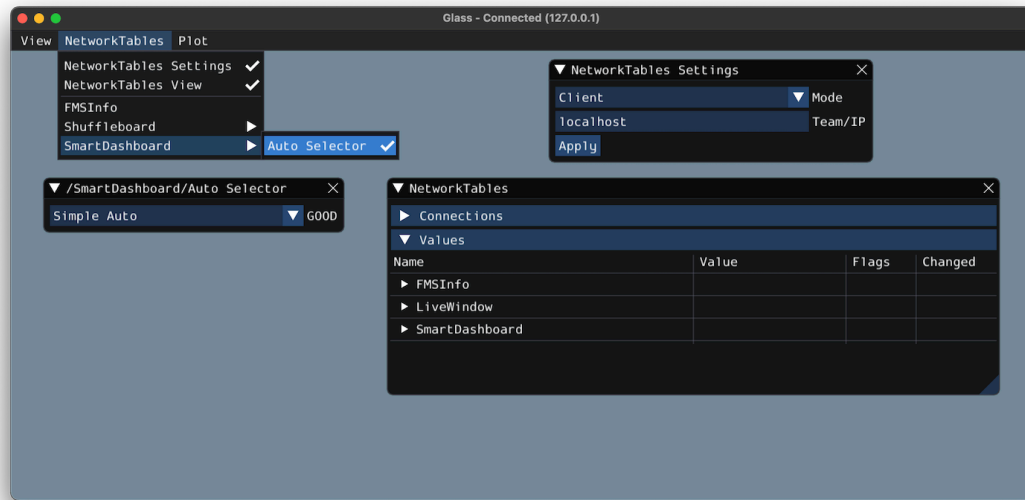
C++

```
SmartDashboard.putData("Auto Selector", m_selector);
```

```
frc::SmartDashboard::PutData("Auto Selector", &m_selector);
```

Note: For more information on creating a `SendableChooser`, please see [this document](#).

The *Sendable Chooser* widget will appear in the *NetworkTables* menu and underneath the main table name that the instance was sent over. From the example above, the main table name would be *SmartDashboard*.



21.3.3 PID Controller Widget

The *PID Controller* widget allows you to quickly tune PID values for a certain controller. A *PIDController* instance must be sent using a *NetworkTables* API. The simplest is to use something like *SmartDashboard*:

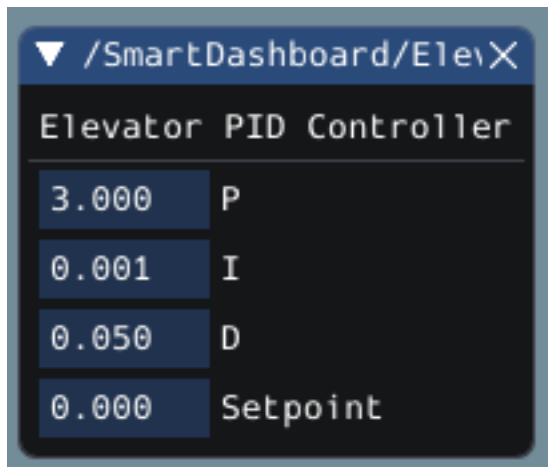
Java

C++

```
SmartDashboard.putData("Elevator PID Controller", m_elevatorPIDController);
```

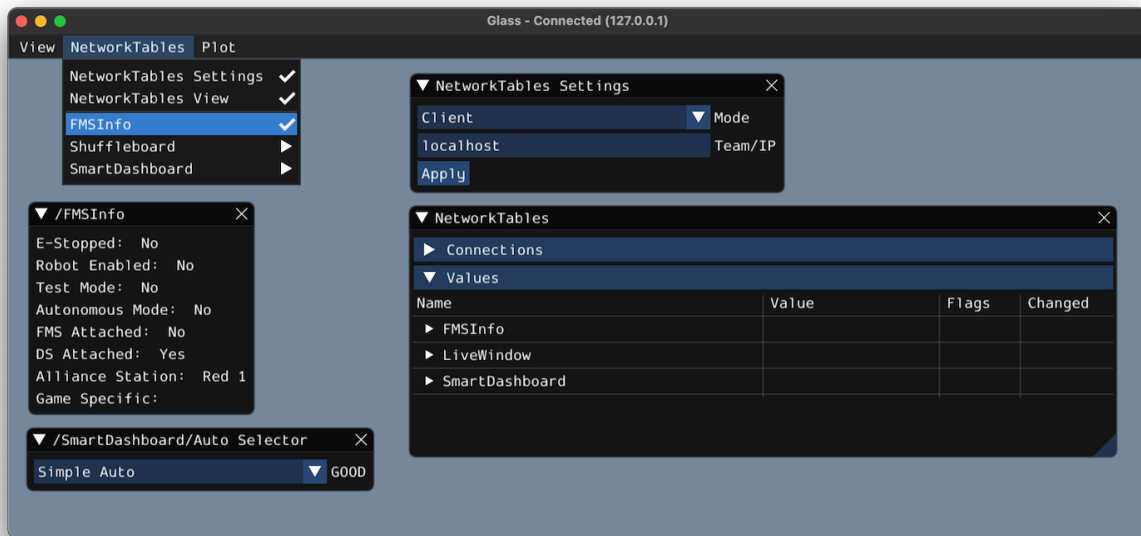
```
frc::SmartDashboard::PutData("Elevator PID Controller", &m_elevatorPIDController);
```

This allows you to quickly tune P, I, and D values for various setpoints.



21.3.4 FMSInfo Widget

The *FMSInfo* widget is created by default when Glass connects to a robot. This widget displays basic information about the robot's enabled state, whether a Driver Station is connected, whether an FMS is connected, the game-specific data, etc. It can be viewed by selecting the *NetworkTables* menu item and clicking on *FMSInfo*.



21.4 Widgets for the Command-Based Framework

Glass also has several widgets that are specific to the *command-based framework*. These include widgets to schedule commands, view actively running commands on a specific subsystem, or view the state of the *command scheduler*.

21.4.1 Command Selector Widget

The *Command Selector* widget allows you to start and cancel a specific instance of a command (sent over NetworkTables) from Glass. For example, you can create an instance of *MyCommand* and send it to SmartDashboard:

Java

C++

```
MyCommand command = new MyCommand(...);
SmartDashboard.putData("My Command", command);
```

```
#include <frc/smartdashboard/SmartDashboard.h>

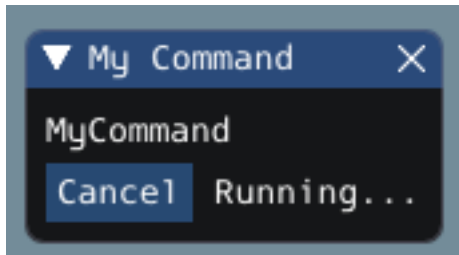
...
```

(continues on next page)

(continued from previous page)

```
MyCommand command{...};
frc::SmartDashboard::PutData("My Command", &command);
```

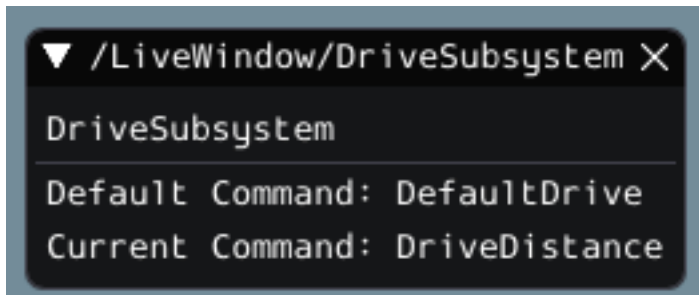
Note: The `MyCommand` instance can also be sent using a lower-level `NetworkTables` API or using the *Shuffleboard API*. In this case, the `SmartDashboard` API was used, meaning that the *Command Selector* widget will appear under the `SmartDashboard` table name.



The widget has two states. When the command is not running, a *Run* button will appear – clicking it will schedule the command. When the command is running, a *Cancel* button, accompanied by *Running...* text, will appear (as shown above). This will cancel the command.

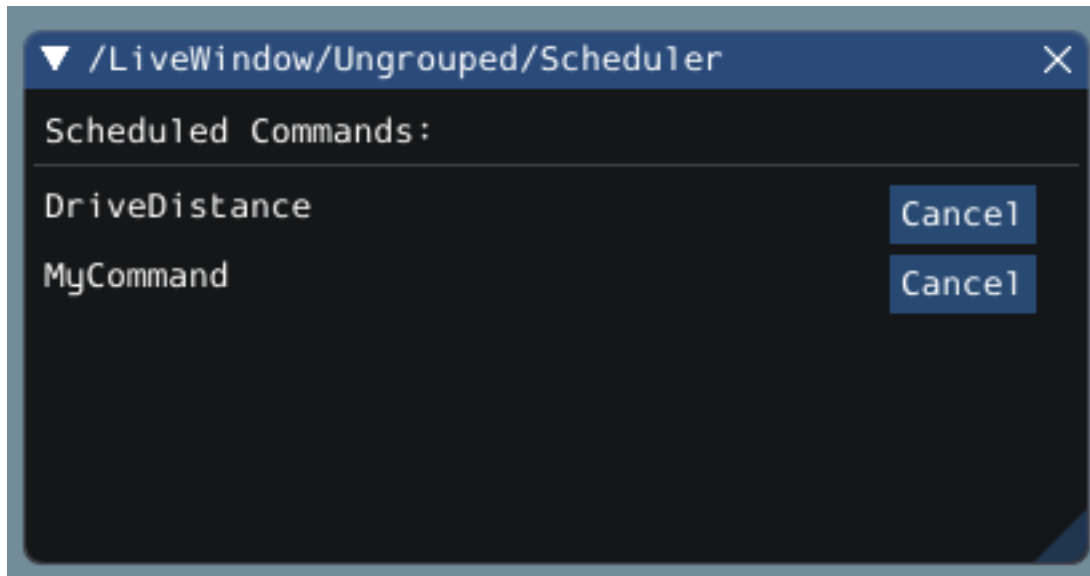
21.4.2 Subsystem Widget

The *Subsystem* widget can be used to see the default command and the currently scheduled command on a specific subsystem. If you are using the `SubsystemBase` base class, your subsystem will be automatically sent to `NetworkTables` over `LiveWindow`. To view this widget, look under the *LiveWindow* main table name in the *NetworkTables* menu.



21.4.3 Command Scheduler Widget

The *Command Scheduler* widget allows you to see all currently scheduled commands. In addition, any of these commands can be canceled from the GUI.



The `CommandScheduler` instance is automatically sent to `NetworkTables` over `LiveWindow`. To view this widget, look under the *LiveWindow* main table name in the *NetworkTables* menu.

21.5 The Field2d Widget

Glass supports displaying your robot's position on the field using the *Field2d* widget. An instance of the `Field2d` class should be created, sent over `NetworkTables`, and updated periodically with the latest robot pose in your robot code.

21.5.1 Sending Robot Pose from User Code

To send your robot's position (usually obtained by *odometry* or a pose estimator), a `Field2d` instance must be created in robot code and sent over `NetworkTables`. The instance must then be updated periodically with the latest robot pose.

Java

C++

```
private final Field2d m_field = new Field2d();

public Drivetrain() {
    ...
    SmartDashboard.putData("Field", m_field);
}

...

public void periodic() {
    ...
    m_field.setRobotPose(m_odometry.getPoseMeters());
}
```



```
#include <frc/smartdashboard/Field2d.h>
#include <frc/smartdashboard/SmartDashboard.h>

frc::Field2d m_field;

Drivetrain() {
    ...
    frc::SmartDashboard::PutData("Field", &m_field);
}

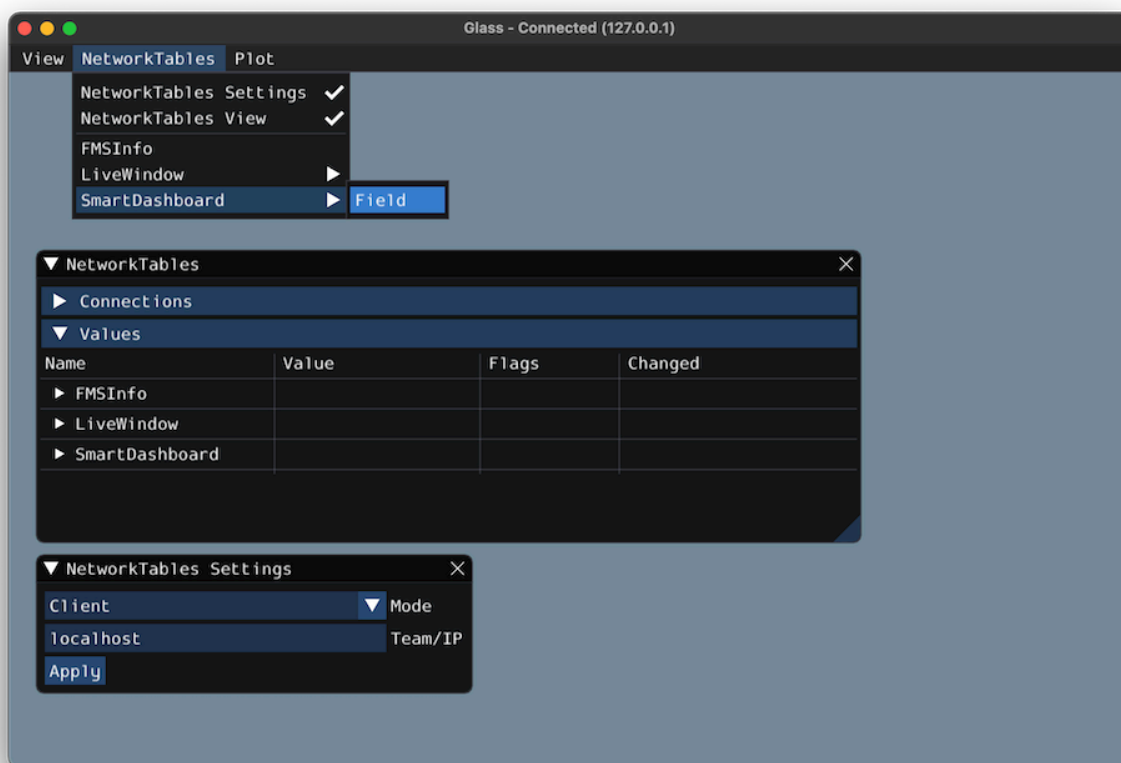
...

void Periodic() {
    ...
    m_field.SetRobotPose(m_odometry.GetPose());
}
```

Note: The `Field2d` instance can also be sent using a lower-level `NetworkTables` API or using the [Shuffleboard API](#). In this case, the `SmartDashboard` API was used, meaning that the *Field2d* widget will appear under the `SmartDashboard` table name.

21.5.2 Viewing the Robot Pose in Glass

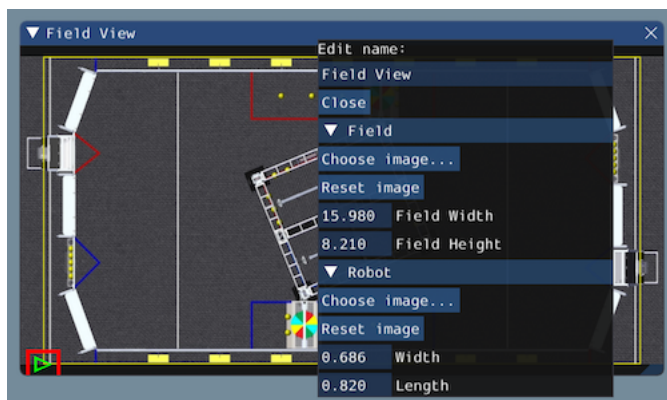
After sending the `Field2d` instance over `NetworkTables`, the *Field2d* widget can be added to Glass by selecting *NetworkTables* in the menu bar, choosing the table name that the instance was sent over, and then clicking on the *Field* button.



Once the widget appears, you can resize and place it on the Glass workspace as you desire. Right-clicking the top of the widget will allow you to customize the name of the widget, select a custom field image, select a custom robot image, and choose the dimensions of the field and robot.

When selecting *Choose image...* you can choose to either select an image file or a PathWeaver JSON file as long as the image file is in the same directory. Choosing the JSON file will automatically import the correct location of the field in the image and the correct size of the field.

Note: You can retrieve the latest field image and JSON files from [here](#). This is the same image and JSON that are used when generating paths using *PathWeaver*.

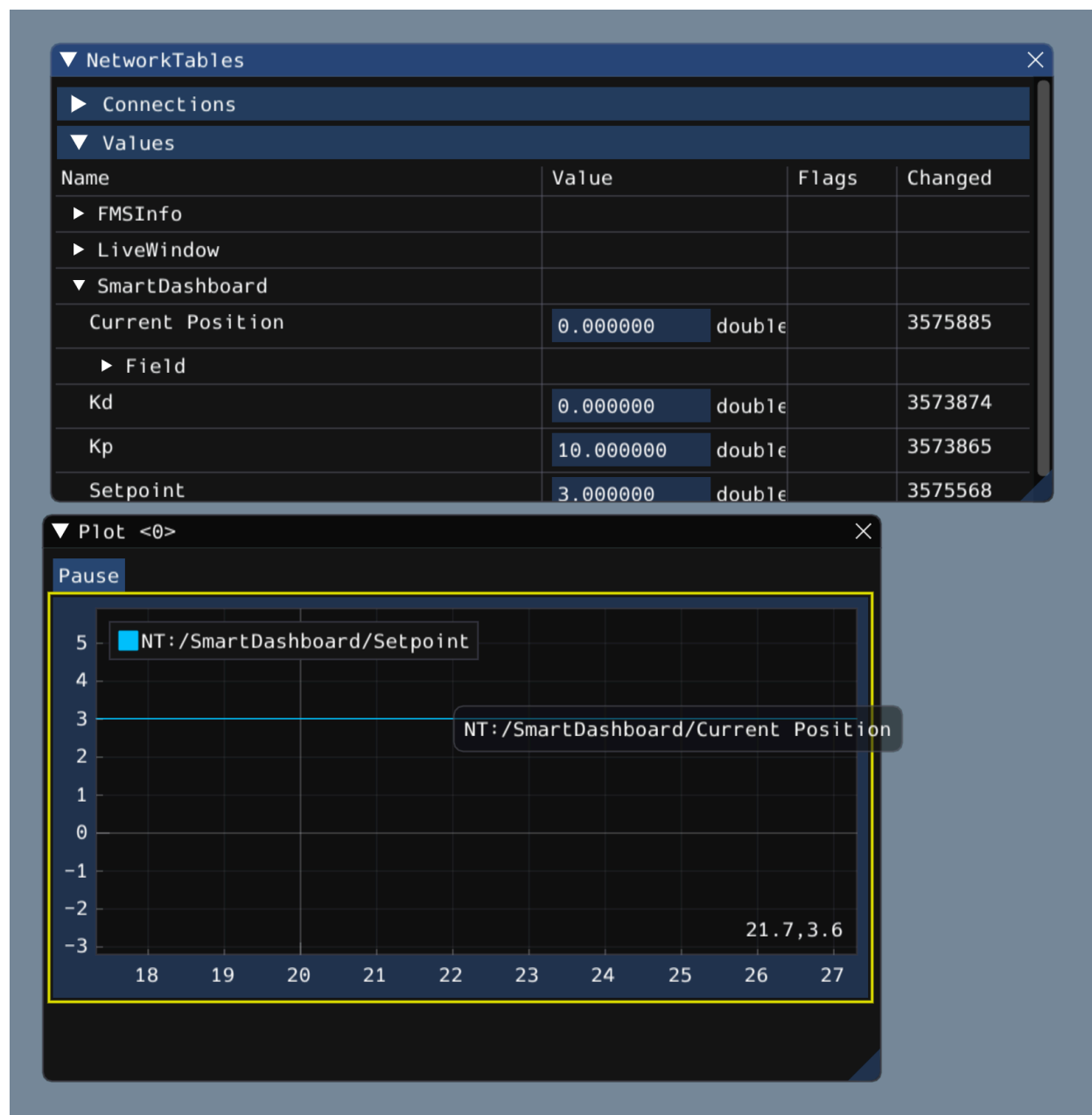


21.6 Plots

Glass excels at high-performance, comprehensive plotting of data from *NetworkTables*. Some features include resizable plots, plots with multiple y axes and the ability to pause, examine, and resume plots.

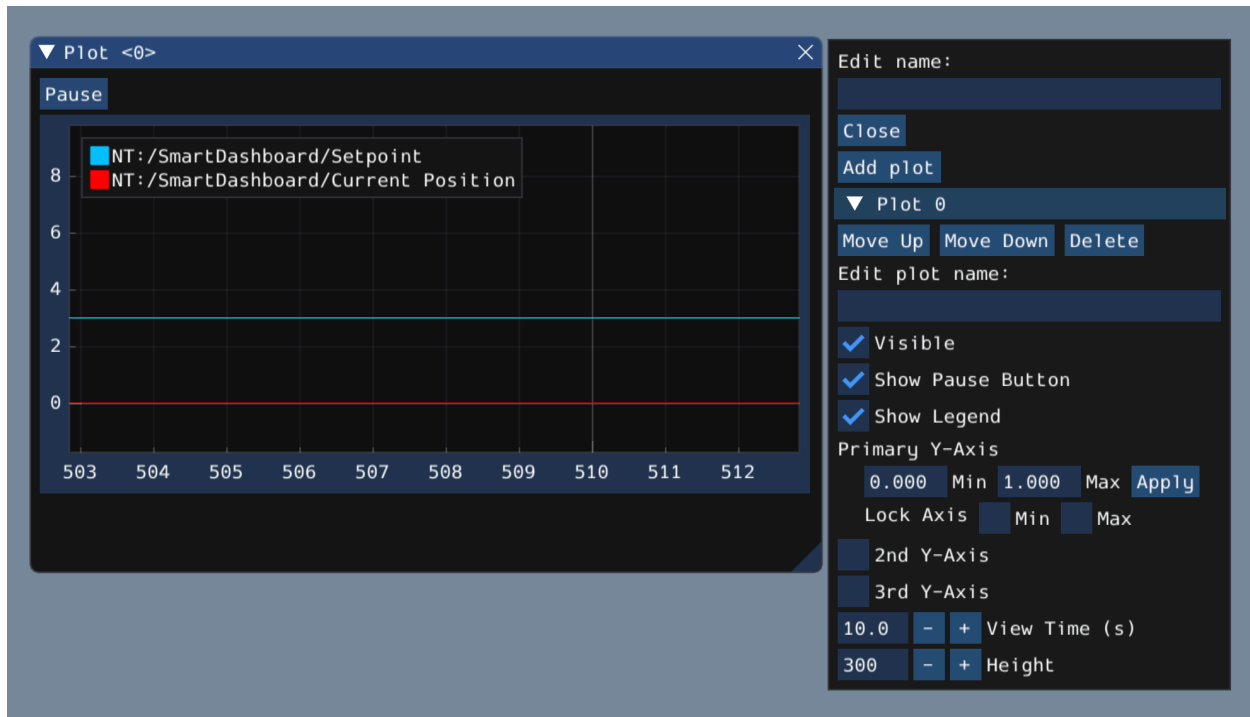
21.6.1 Creating a Plot

A new plot widget can be created by selecting the *Plot* button on the main menu bar and then clicking on *New Plot Window*. Several individual plots can be added to each plot window. To add a plot within a plot window, click the *Add plot* button inside the widget. Then you can drag various sources from the *NetworkTables* widget into the plot:

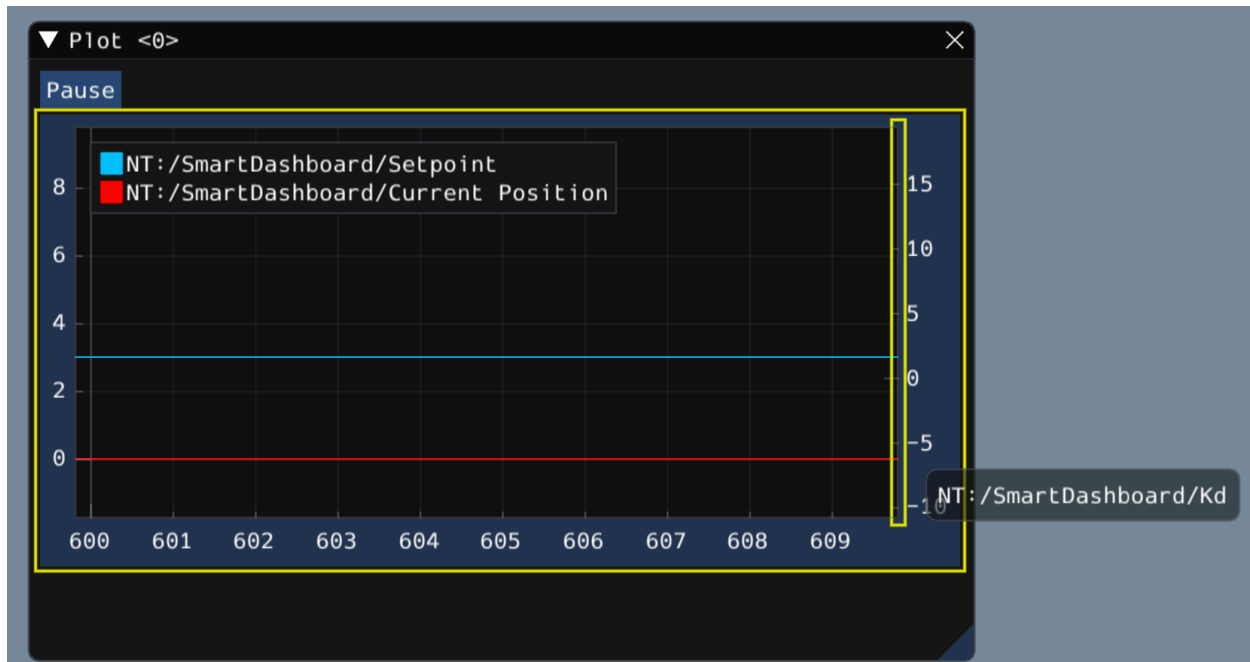


21.6.2 Manipulating Plots

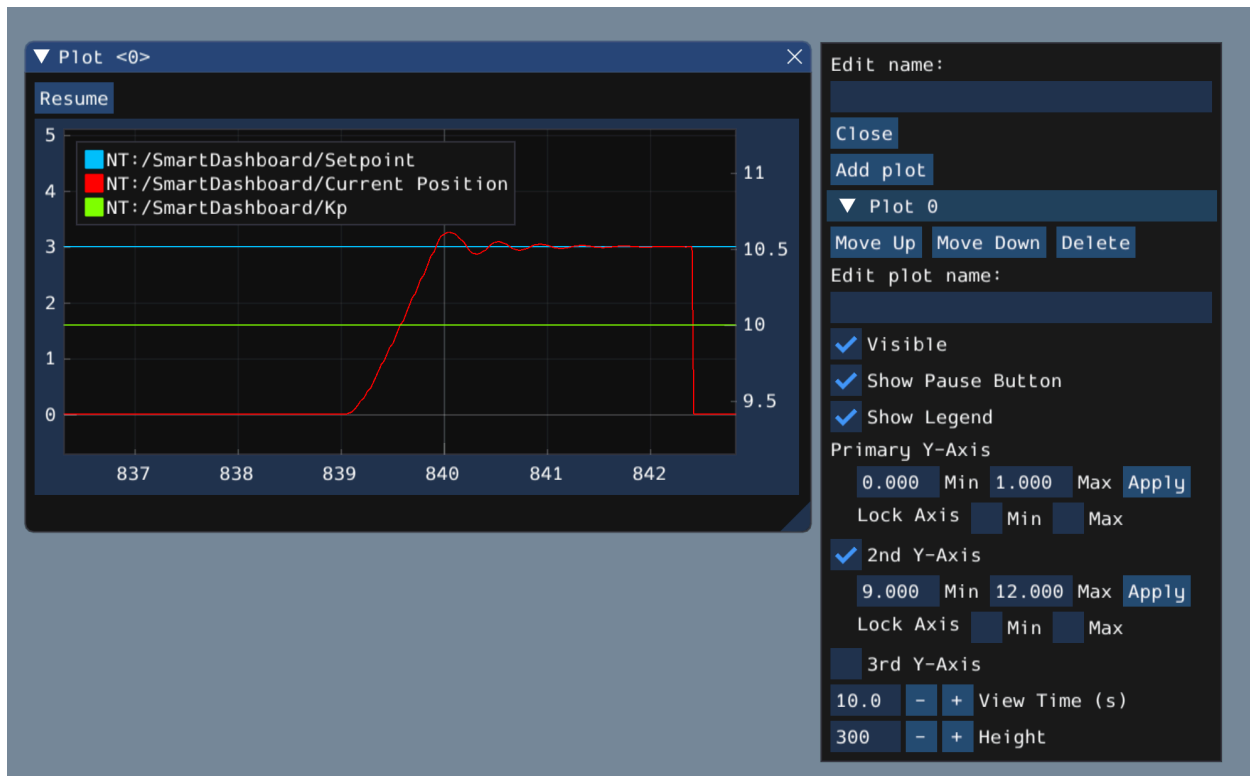
You can click and drag on the plot to move around and scroll on top of the plot to zoom the y axes in and out. Double clicking on the graph will autoscale it so that the zoom and axis limits fit all of the data it is plotting. Furthermore, right-clicking on the plot will present you with a plethora of options, including whether you want to display secondary and tertiary y axes, if you wish to lock certain axes, etc.



If you choose to make secondary and tertiary y axes available, you can drag data sources onto those axes to make their lines correspond with your desired axis:



Then, you can lock certain axes so that their range always remains constant, regardless of panning. In this example, the secondary axis range (with the /SmartDashboard/Kp entry) was locked between 9 and 12.

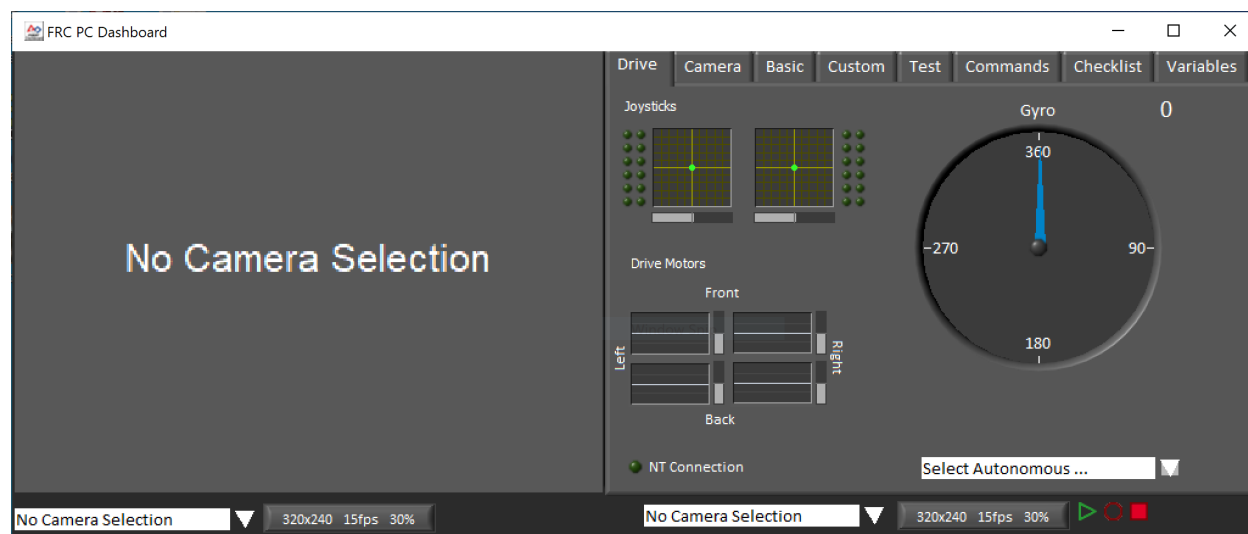


LabVIEW Dashboard

22.1 FRC LabVIEW Dashboard

The Dashboard application installed and launched by the FRC® Driver Station is a LabVIEW program designed to provide teams with basic feedback from their robot, with the ability to expand and customize the information to suit their needs. This Dashboard application uses *NetworkTables* and contains a variety of tools that teams may find useful.

22.1.1 LabVIEW Dashboard



The Dashboard is broken into two main sections. The left pane is for displaying a camera image. The right pane contains:

- Drive tab that contains indicators for joystick and drive motor values (hooked up by default when used with LabVIEW robot code), a gyro indicator, an Autonomous selection text box, a connection indicator and some controls and indicators for the camera
- Basic tab that contains some default controls and indicators

- Camera tab that contains a secondary camera viewer, similar to the viewer in the left pane
- Custom tab for customizing the dashboard using LabVIEW
- Test tab for use with Test Mode in the LabVIEW framework
- Commands tab for use with the new LabVIEW C&C Framework
- Checklist tab that can be used to create task lists to complete before and/or between matches
- Variables tab that displays the raw NetworkTables variables in a tree view format

The LabVIEW Dashboard also includes Record/Playback functionality, located in the bottom right. More detail about this feature is included below under [Record/Playback](#).

22.1.2 Camera Image and Controls



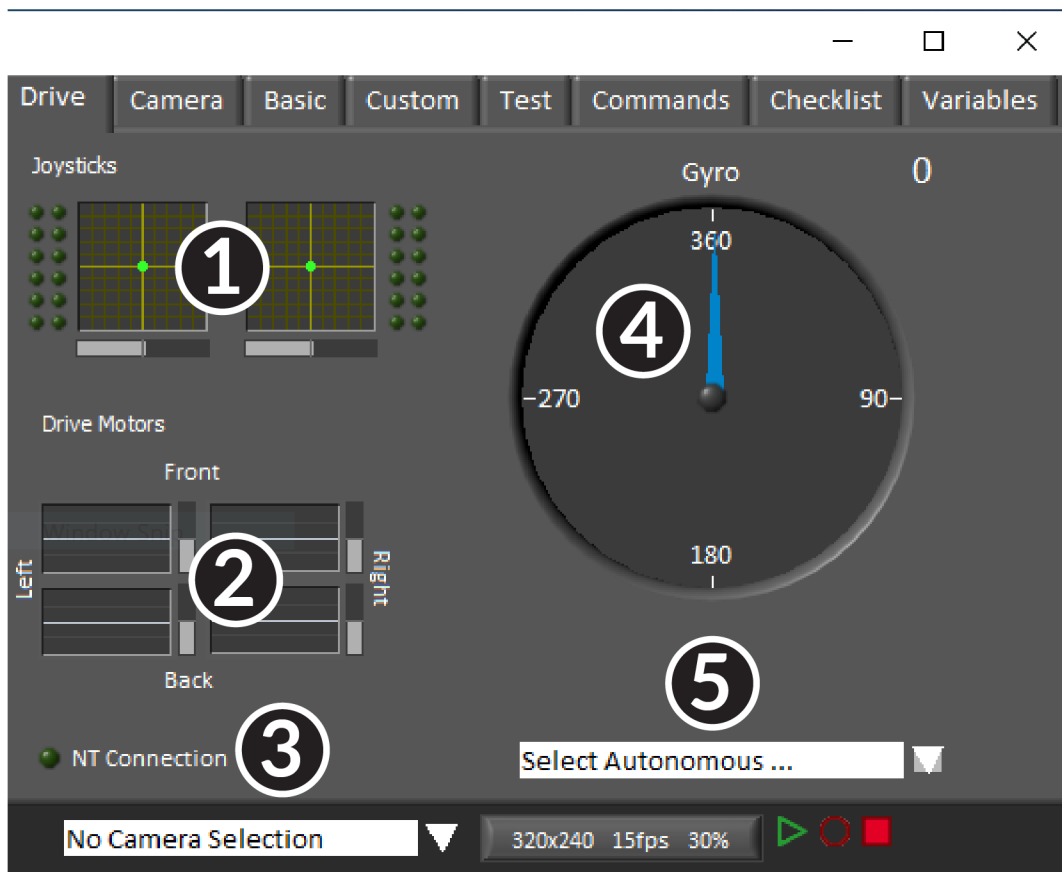
The left pane is used to display a video feed from an Axis camera or USB camera located on the robot. For instructions on setting up the Axis Camera to work with this display [see here](#). There are also some controls and indicators related to the camera below the tab area:

1. Camera Image Display
2. Mode Selector - This drop-down allows you to select the type of camera display to use. The choices are Camera Off, USB Camera SW (software compression), USB Camera HW (hardware compression) and IP Camera (Axis camera). Note that the IP Camera setting will not work when your PC is connected to the roboRIO over USB.

3. Camera Settings - This control allows you to change the resolution, framerate and compression of the image stream to the dashboard, click the control to pop-up the configuration.
4. Bandwidth Indicator - Indicates approximate bandwidth usage of the image stream. The indicator will display green for “safe” bandwidth usage, yellow when teams should use caution and red if the stream bandwidth is beyond levels that will work on the competition field.
5. Framerate - Indicates the approximate received framerate of the image stream.

Tip: The bandwidth indicator indicates the combined bandwidth for all camera streams open.

22.1.3 Drive



The center pane contains a section that provides feedback on the joysticks and drive commands when used with the LabVIEW framework and a section that displays the NetworkTables status and autonomous selector:

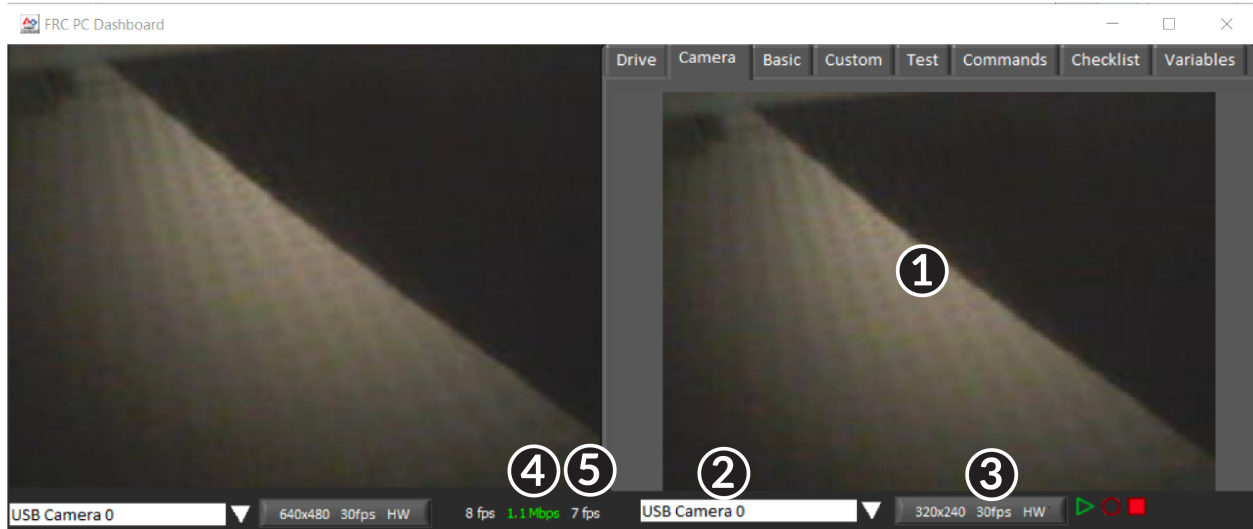
1. Displays X,Y and Throttle information and button values for up to 2 joysticks when using the LabVIEW framework
2. Displays values being sent to motor controllers when using LabVIEW framework

3. Displays a connection indicator for the NetworkTables data from the robot
4. Displays a Gyro value
5. Displays a text box that can be used to select Autonomous modes. Each language's code templates have examples of using this box to select from multiple autonomous programs.

These indicators (other than the Gyro) are hooked up to appropriate values by default when using the LabVIEW framework. For information on using them with C++/Java code see [Using the LabVIEW Dashboard with C++/Java Code](#).

22.1.4 Camera

Tip: The left pane can only display a single camera output, so use the camera tab on the right pane to display a second camera output if needed.

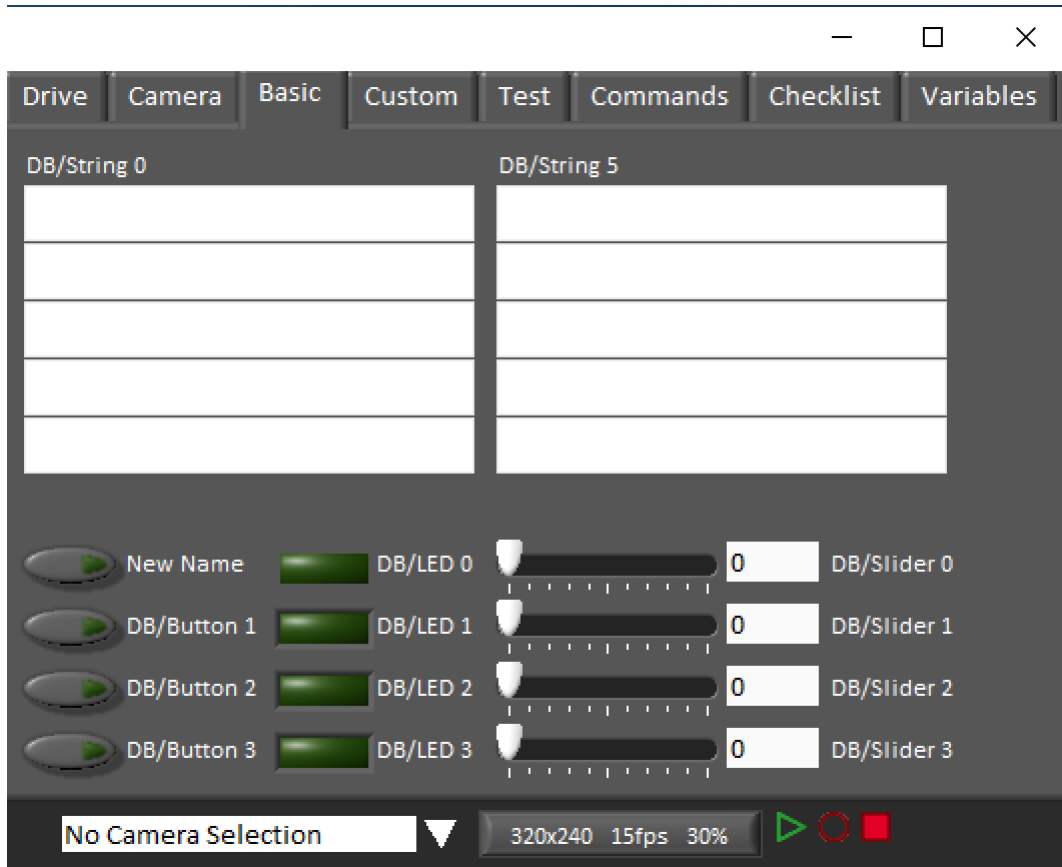


The camera tab is used to display a video feed from an Axis camera or USB camera located on the robot. For instructions on setting up the Axis Camera to work with this display [see here](#). There are also some controls and indicators related to the camera below the tab area:

1. Camera Image Display
2. Mode Selector - This drop-down allows you to select the type of camera display to use. The choices are Camera Off, USB Camera SW (software compression), USB Camera HW (hardware compression) and IP Camera (Axis camera). Note that the IP Camera setting will not work when your PC is connected to the roboRIO over USB.
3. Camera Settings - This control allows you to change the resolution, framerate and compression of the image stream to the dashboard, click the control to pop-up the configuration.
4. Bandwidth Indicator - Indicates approximate bandwidth usage of the image stream. The indicator will display green for “safe” bandwidth usage, yellow when teams should use caution and red if the stream bandwidth is beyond levels that will work on the competition field.
5. Framerate - Indicates the approximate received framerate of the image stream.

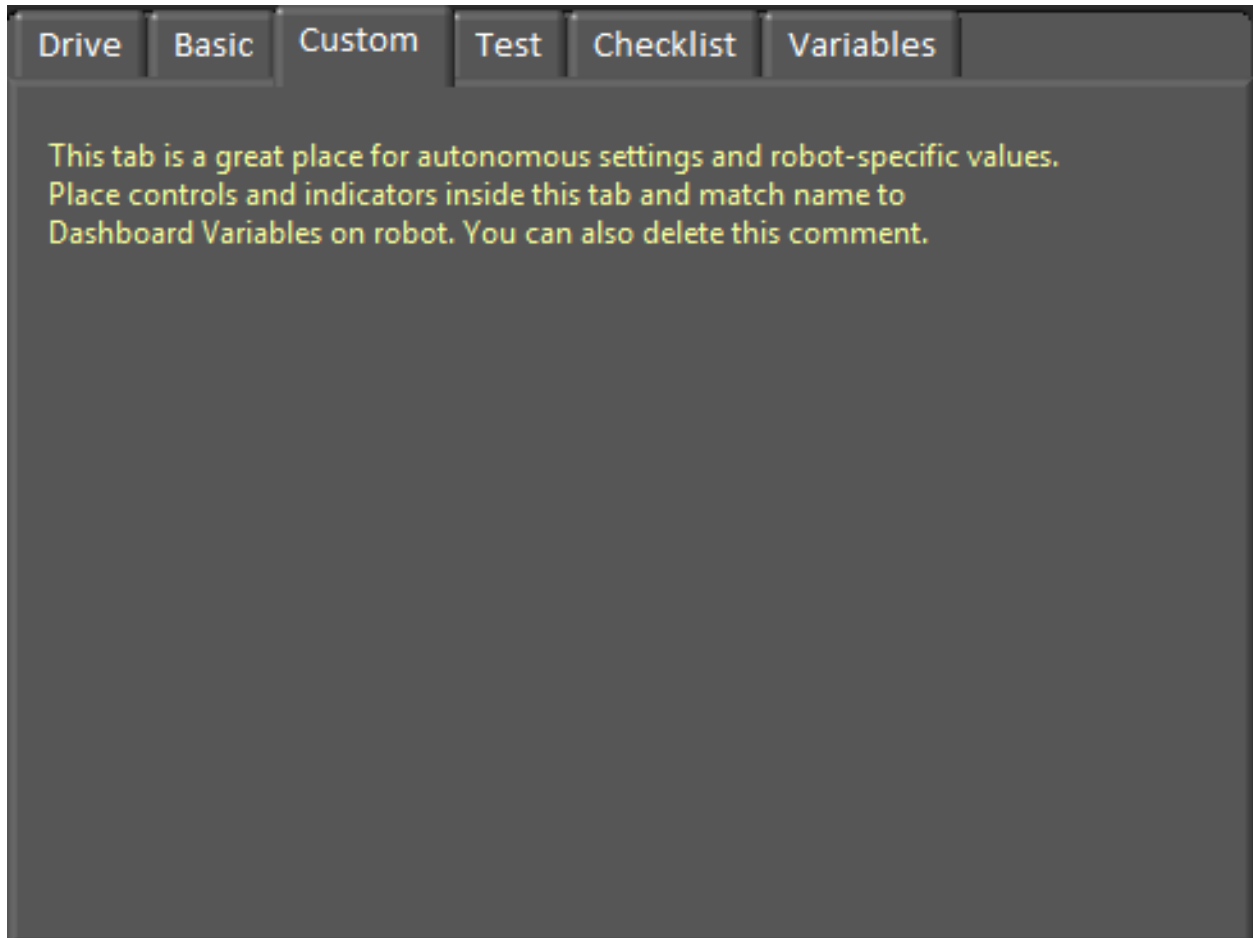
Tip: The bandwidth indicator indicates the combined bandwidth for all camera streams open.

22.1.5 Basic



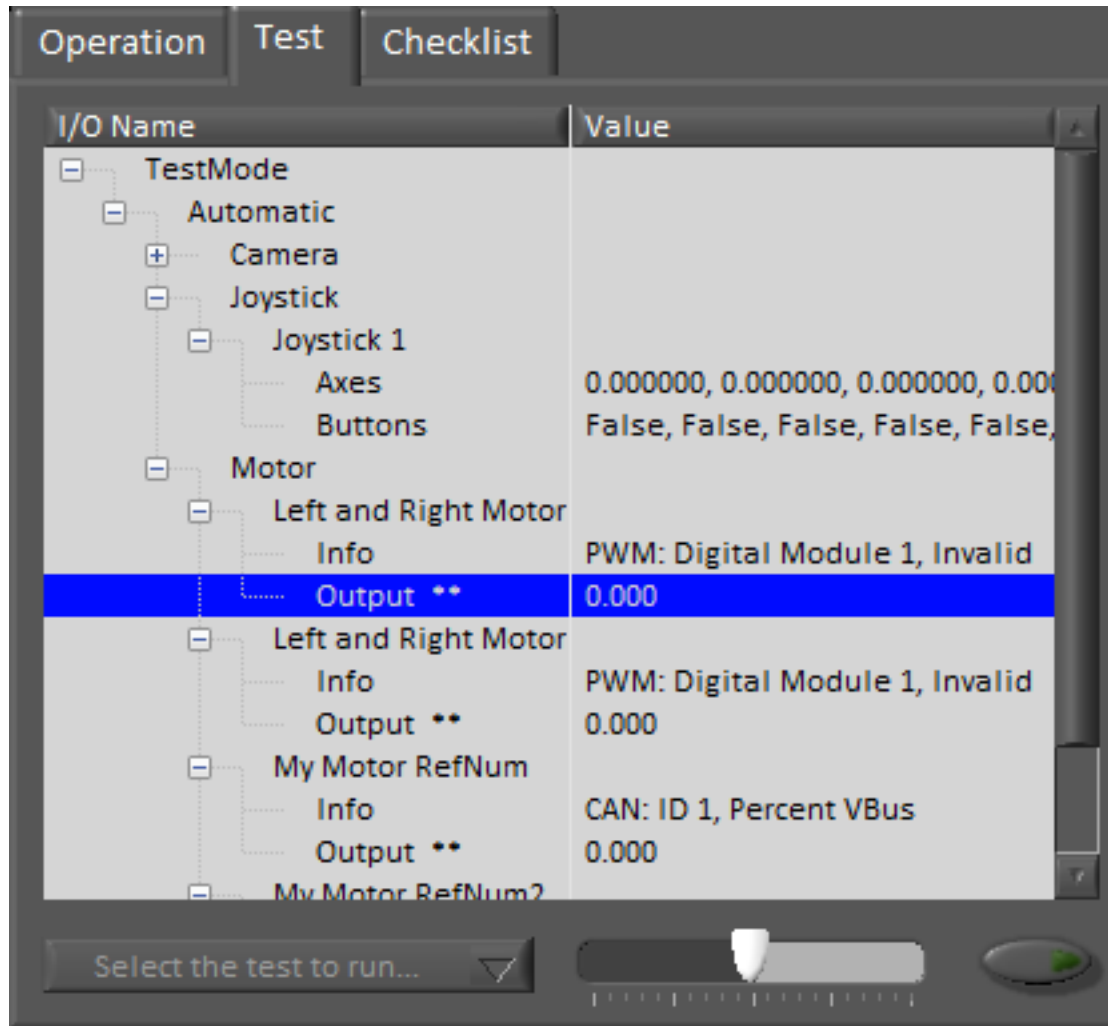
The Basic tab contains a variety of pre-populated bi-directional controls/indicators which can be used to control the robot or display information from the robot. The SmartDashboard key names associated with each item are labeled next to the indicator with the exception of the Strings which follow the same naming pattern and increment from DB/String 0 to DB/String 4 on the left and DB/String 5 to DB/String 9 on the right. The LabVIEW framework contains an example of reading from the Buttons and Sliders in Teleop. It also contains an example of customizing the labels in Begin. For more detail on using this tab with C++/Java code, see [Using the LabVIEW Dashboard with C++/Java Code](#).

22.1.6 Custom



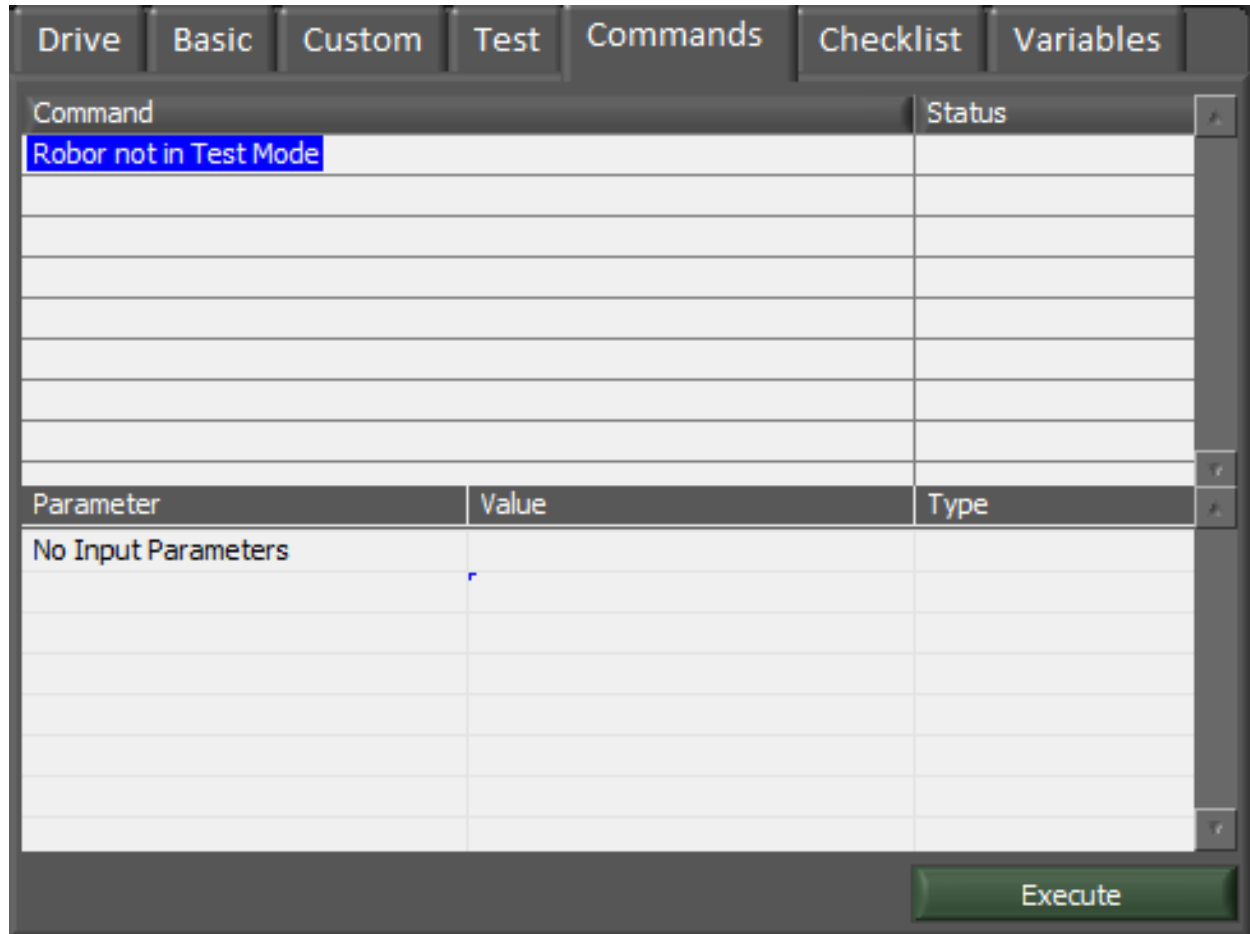
The Custom tab allows you to add additional controls/indicators to the dashboard using LabVIEW without removing any existing functionality. To customize this tab you will need to create a Dashboard project in LabVIEW.

22.1.7 Test



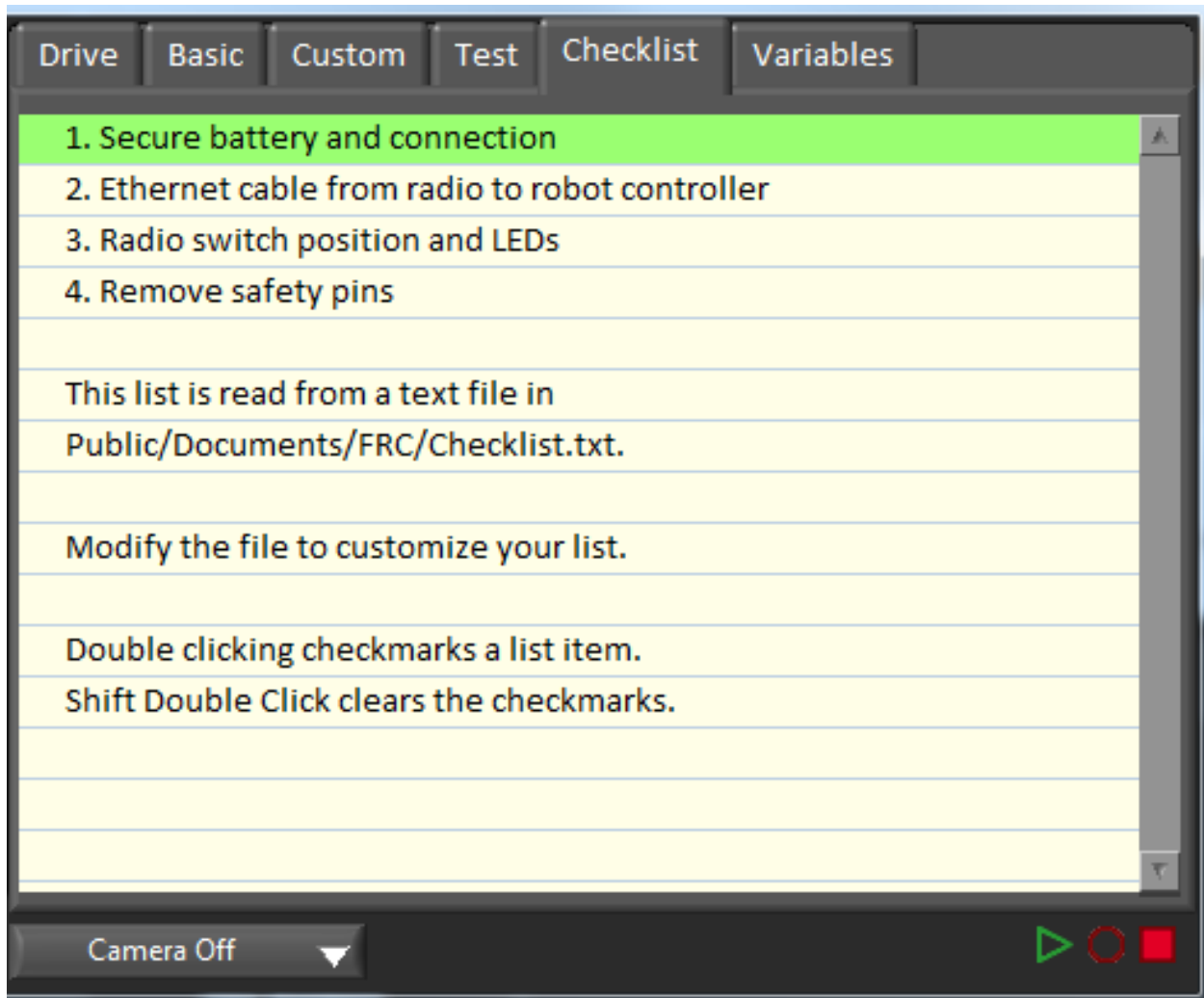
The Test tab is for use with Test mode for teams using LabVIEW (Java and C++ teams should use SmartDashboard or Shuffleboard when using Test Mode). For many items in the libraries, Input/Output info will be populated here automatically. All items which have ** next to them are outputs that can be controlled by the dashboard. To control an output, click on it to select it, drag the slider to set the value then press and hold the green button to enable the output. As soon as the green button is released, the output will be disabled. This tab can also be used to run and monitor tests on the robot. An example test is provided in the LabVIEW framework. Selecting this test from the dropdown box will show the status of the test in place of the slider and enable controls.

22.1.8 Commands



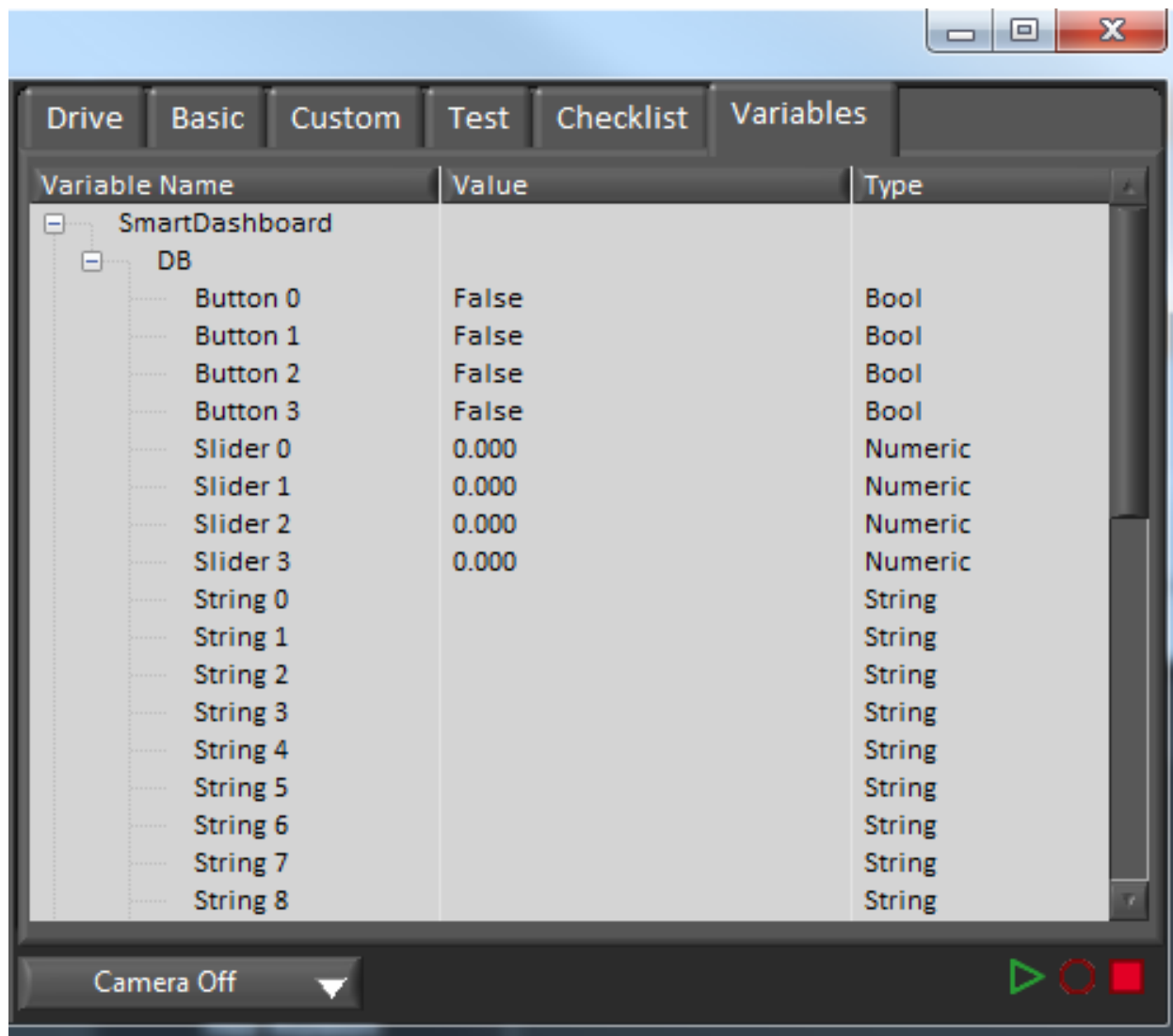
The Commands tab can be used with the Robot in Test mode to see which commands are running and to manually run commands for test purposes.

22.1.9 Checklist



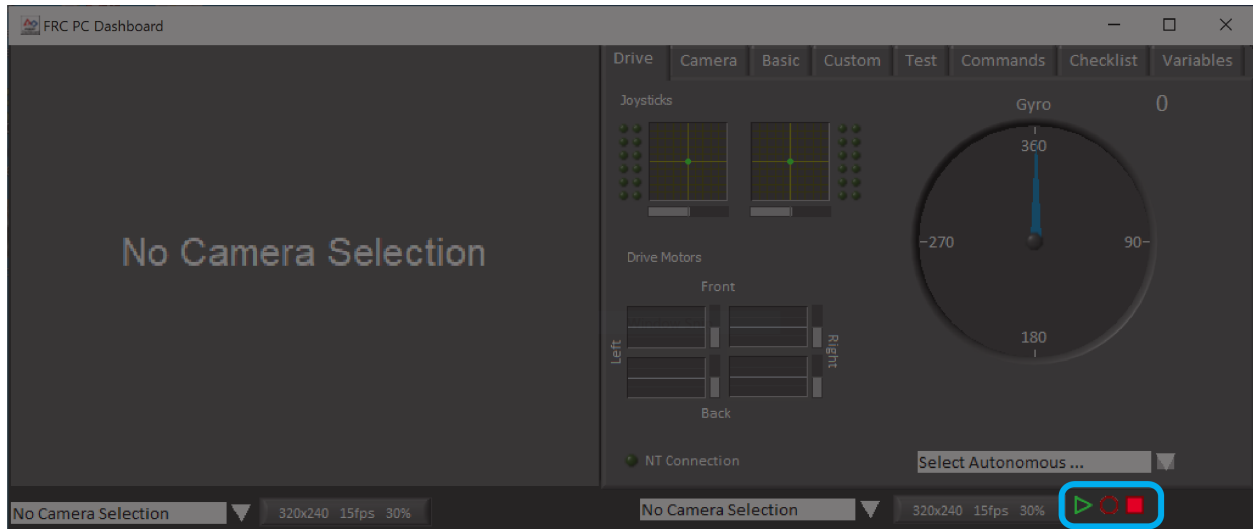
The Checklist tab can be used by teams to create a list of tasks to perform before or between matches. Instructions for using the Checklist tab are pre-populated in the default checklist file.

22.1.10 Variables

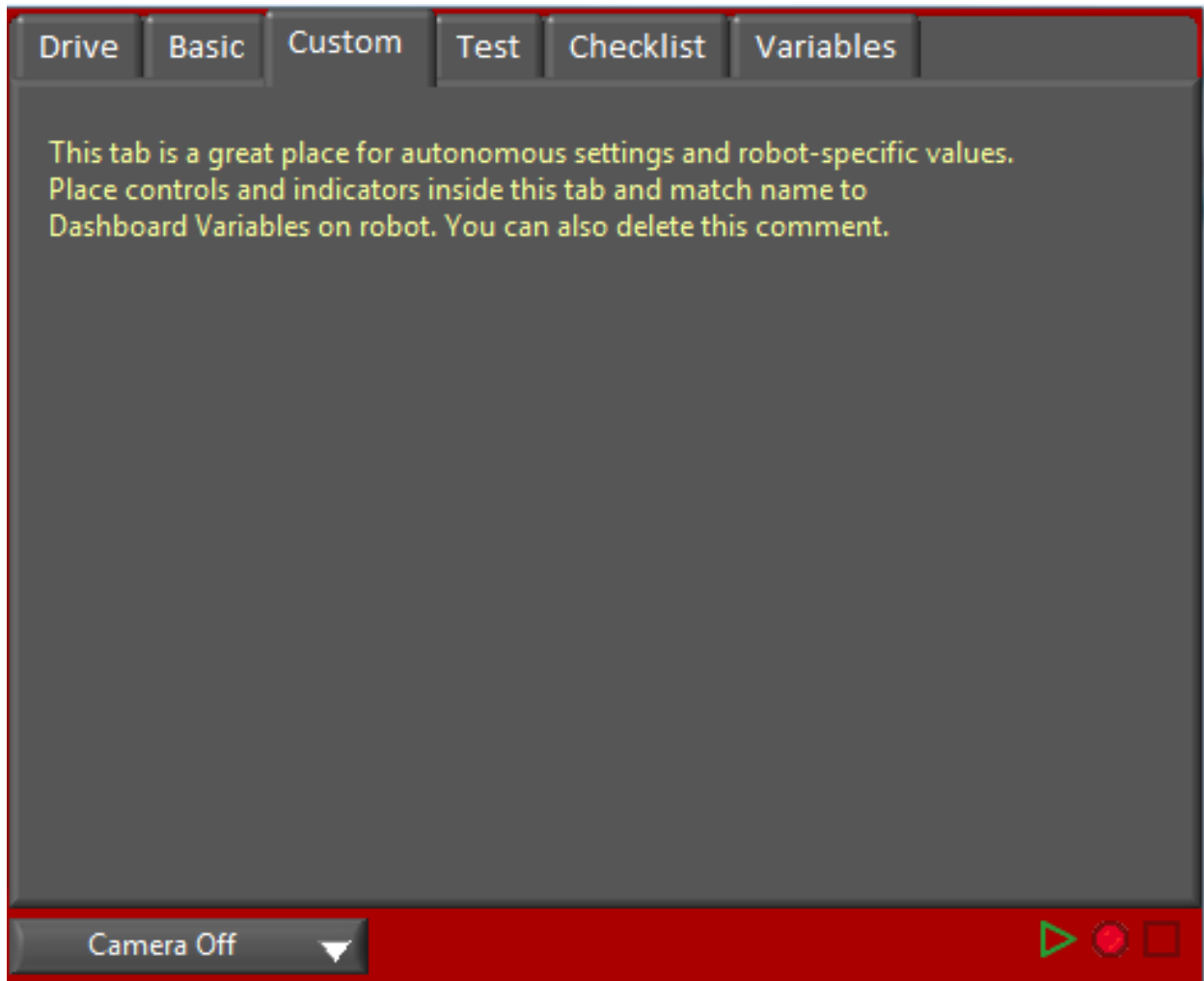


The Variables tab of the left pane shows all NetworkTables variables in a tree display. The Variable Name (Key), Value and data type are shown for each variable. Information about the NetworkTables bandwidth usage is also displayed in this tab. Entries will be shown with black diamonds if they are not currently synced with the robot.

22.1.11 Record/Playback

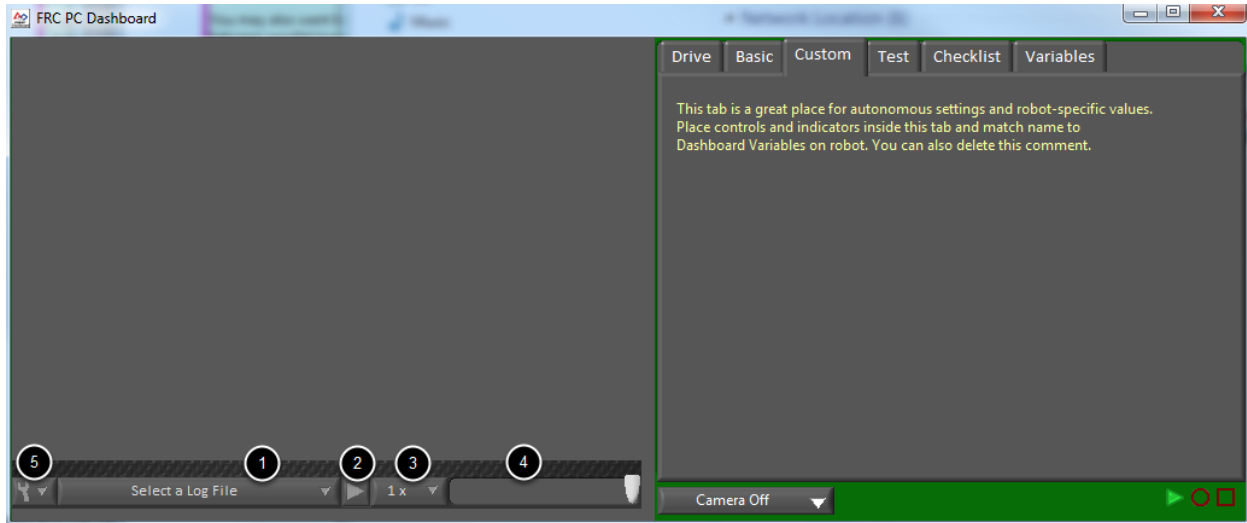


The LabVIEW Dashboard includes a Record/Playback feature that allows you to record video and NetworkTables data (such as the state of your Dashboard indicators) and play it back later.

Recording

To begin recording, click the red circular Record button. The background of the right pane will turn red to indicate you are recording. To stop recording, press the red square Stop button.

Playback



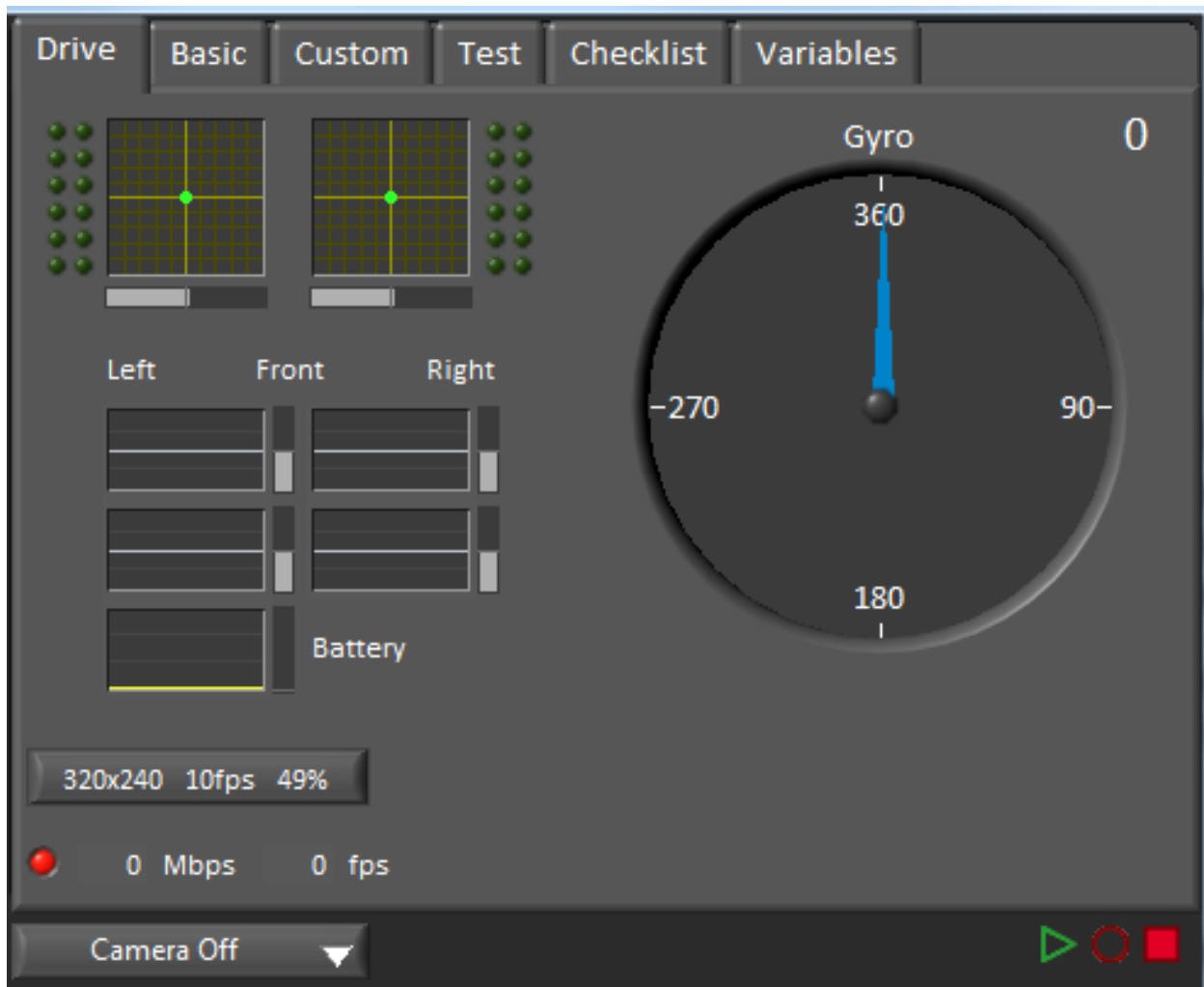
To play a recording back, click the green triangle Play button. The background of the right pane will begin pulsing green and playback controls will appear at the bottom of the camera pane.

1. File Selector - The dropdown allows you to select a log file to play back. The log files are named using the date and time and the dropdown will also indicate the length of the file. Selecting a logfile will immediately begin playing that file.
2. Play/Pause button - This button allows you to pause and resume playback of the log file.
3. Playback Speed - This dropdown allows you to adjust playback speed from 1/10 speed to 10x speed, the default is real-time (1x)
4. Time Control Slider - This slider allows you to fast-forward or rewind through the logfile by clicking on the desired location or dragging the slider.
5. Settings - With a log file selected, this dropdown allows you to rename or delete a file or open the folder containing the logs in Windows Explorer (Typically C:\Users\Public\Documents\FRC\Log Files\Dashboard)

22.2 Using the LabVIEW Dashboard with C++/Java Code

The default LabVIEW Dashboard utilizes *NetworkTables* to pass values and is therefore compatible with C++ and Java robot programs. This article covers the keys and value ranges to use to work with the Dashboard.

22.2.1 Drive Tab



The *Select Autonomous...* dropdown can be used so show the available autonomous routines and choose one to run for the match.

Java

C++

```
SmartDashboard.putStringArray("Auto List", {"Drive Forwards", "Drive Backwards",
↪ "Shoot"});

// At the beginning of auto
```

(continues on next page)

(continued from previous page)

```
String autoName = SmartDashboard.getString("Auto Selector", "Drive Forwards") // This
↳would make "Drive Forwards the default auto
switch(autoName) {
    case "Drive Forwards":
        // auto here
    case "Drive Backwards":
        // auto here
    case "Shoot":
        // auto here
}
```

```
frc::SmartDashboard::PutStringArray("Auto List", {"Drive Forwards", "Drive Backwards",
↳ "Shoot"});

// At the beginning of auto
String autoName = SmartDashboard.GetString("Auto Selector", "Drive Forwards") // This
↳would make "Drive Forwards the default auto
switch(autoName) {
    case "Drive Forwards":
        // auto here
    case "Drive Backwards":
        // auto here
    case "Shoot":
        // auto here
}
```

Sending to the “Gyro” NetworkTables entry will populate the gyro here.

Java

C++

```
SmartDashboard.putNumber("Gyro", drivetrain.getHeading());
```

```
frc::SmartDashboard::PutNumber("Gyro", Drivetrain.GetHeading());
```

There are four outputs that show the motor power to the drivetrain. This is configured for 2 motors per side and a tank style drivetrain. This is done by setting “RobotDrive Motors” like the example below.

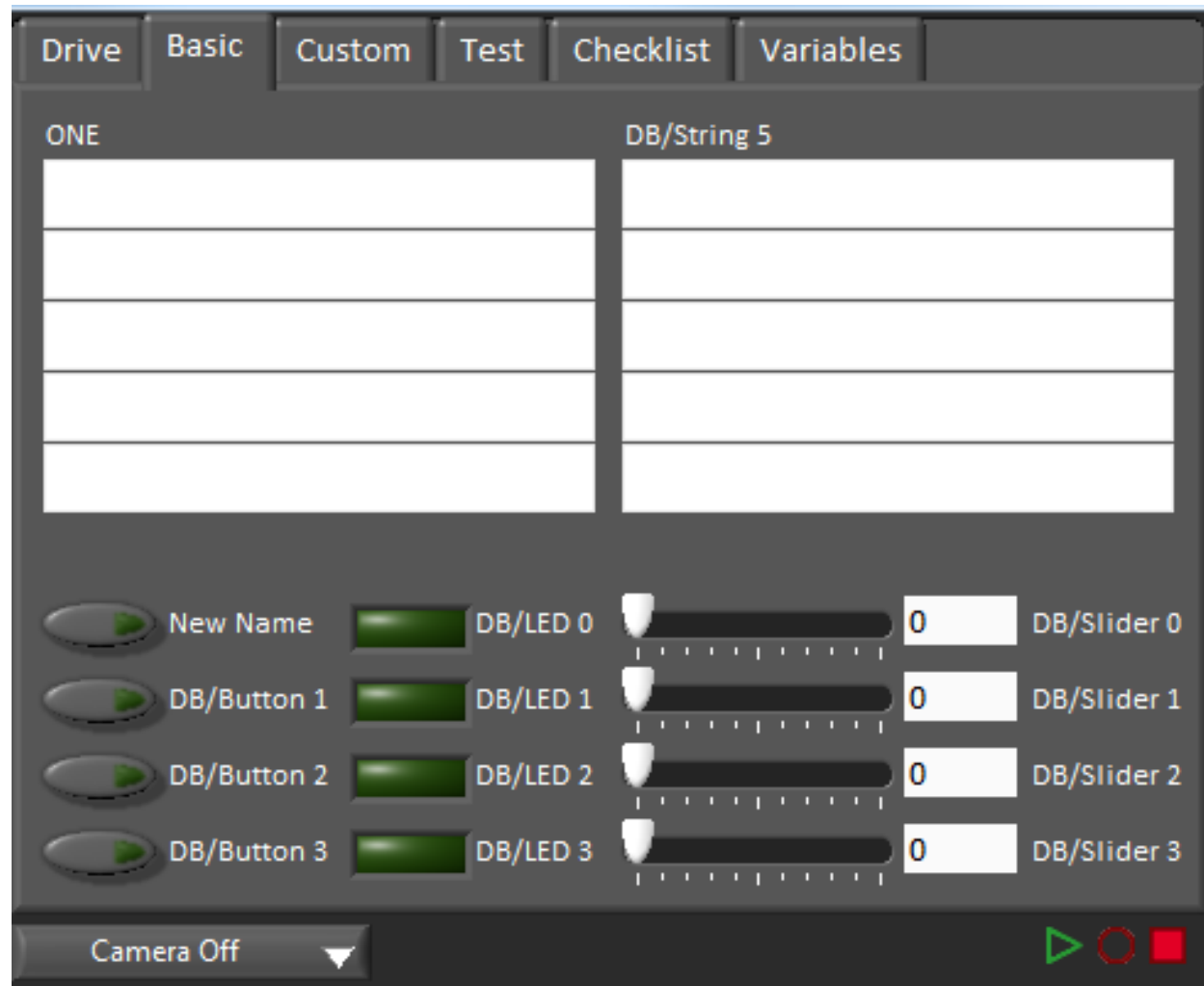
Java

C++

```
SmartDashboard.putNumberArray("RobotDrive Motors", {drivetrain.getLeftFront(),
↳drivetrain.getRightFront(), drivetrain.getLeftBack(), drivetrain.getRightBack()});
```

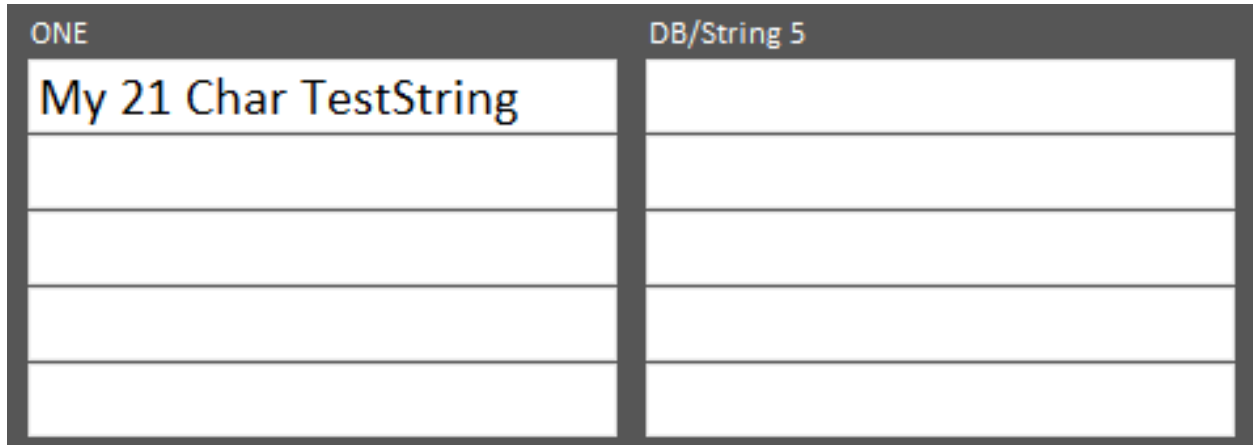
```
frc::SmartDashboard::PutNumberArray("Gyro", {drivetrain.GetLeftFront(), drivetrain.
↳GetRightFront(), drivetrain.GetLeftBack(), drivetrain.GetRightBack()});
```

22.2.2 Basic Tab



The Basic tab uses a number of keys in the a “DB” sub-table to send/receive Dashboard data. The LED’s are output only, the other fields are all bi-directional (send or receive).

Strings



The strings are labeled top-to-bottom, left-to-right from “DB/String 0” to “DB/String 9”. Each String field can display at least 21 characters (exact number depends on what characters). To write to these strings:

Java

C++

```
SmartDashboard.putString("DB/String 0", "My 21 Char TestString");
```

```
frc::SmartDashboard::PutString("DB/String 0", "My 21 Char TestString");
```

To read string data entered on the Dashboard:

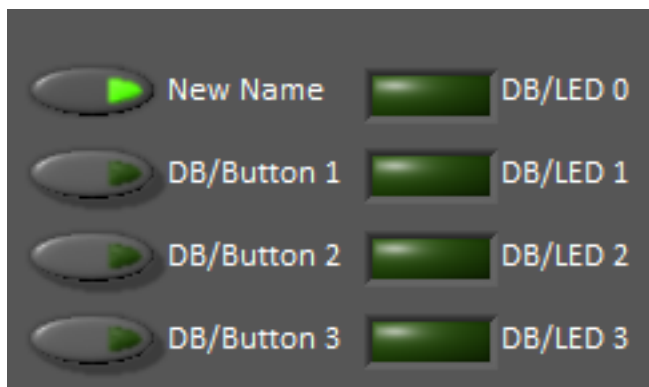
Java

C++

```
String dashData = SmartDashboard.getString("DB/String 0", "myDefaultData");
```

```
std::string dashData = frc::SmartDashboard::GetString("DB/String 0", "myDefaultData");
```

Buttons and LEDs



The Buttons and LEDs are boolean values and are labeled top-to-bottom from “DB/Button 0” to “DB/Button 3” and “DB/LED 0” to “DB/LED 3”. The Buttons are bi-directional, the LEDs are only able to be written from the Robot and read on the Dashboard. To write to the Buttons or LEDs:

Java

C++

```
SmartDashboard.putBoolean("DB/Button 0", true);
```

```
frc::SmartDashboard::PutBoolean("DB/Button 0", true);
```

To read from the Buttons: (default value is false)

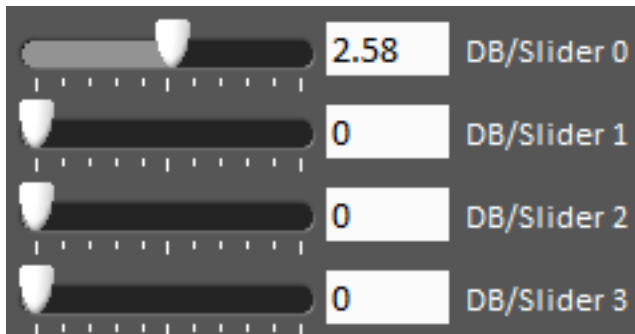
Java

C++

```
boolean buttonValue = SmartDashboard.getBoolean("DB/Button 0", false);
```

```
bool buttonValue = frc::SmartDashboard::GetBoolean("DB/Button 0", false);
```

Sliders



The Sliders are bi-directional analog (double) controls/indicators with a range from 0 to 5. To write to these indicators:

Java

C++

```
SmartDashboard.putNumber("DB/Slider 0", 2.58);
```

```
frc::SmartDashboard::PutNumber("DB/Slider 0", 2.58);
```

To read values from the Dashboard into the robot program: (default value of 0.0)

Java

C++

```
double dashData = SmartDashboard.getNumber("DB/Slider 0", 0.0);
```



```
double dashData = frc::SmartDashboard::GetNumber("DB/Slider 0", 0.0);
```

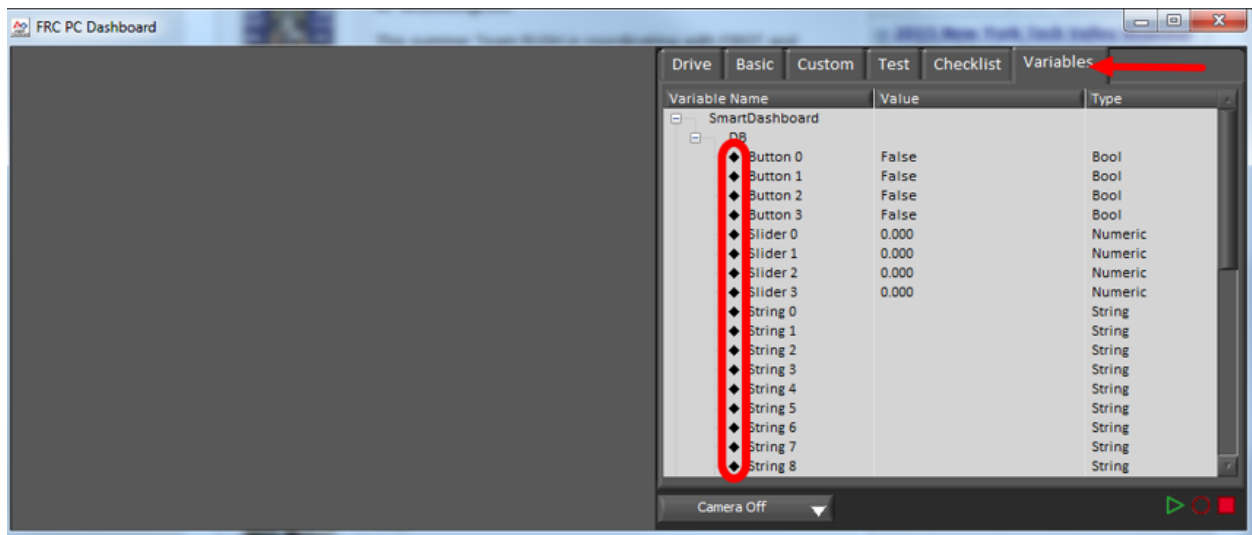
22.3 Troubleshooting Dashboard Connectivity

We have received a number of reports of Dashboard connectivity issues from events. This document will help explain how to recognize if the Dashboard is not connected to your robot, steps to troubleshoot this condition and a code modification you can make.

22.3.1 LabVIEW Dashboard

This section discusses connectivity between the robot and LabVIEW dashboard

Recognizing LabVIEW Dashboard Connectivity



If you have an indicator on your dashboard that you expect to be changing it may be fairly trivial to recognize if the Dashboard is connected. If not, there is a way to check without making any changes to your robot code. On the Variables tab of the Dashboard, the variables are shown with a black diamond when they are not synced with the robot. Once the Dashboard connects to the robot and these variables are synced, the diamond will disappear.

Troubleshooting LabVIEW Dashboard Connectivity

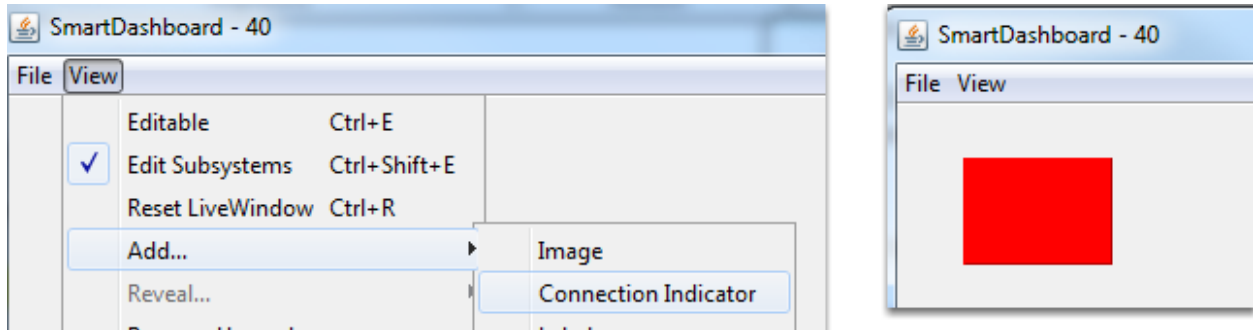
If the Dashboard does not connect to the Robot (after the Driver Station has connected to the robot) the recommended troubleshooting steps are:

1. Close the Driver Station and Dashboard, then re-open the Driver Station (which should launch the Dashboard).
2. If that doesn't work, restart the Robot Code using the Restart Robot Code button on the Diagnostics tab of the Driver Station

22.3.2 Recognizing Connectivity

This section discusses connectivity between the robot and SmartDashboard

Recognizing SmartDashboard Connectivity



The typical way to recognize connectivity with the SmartDashboard is to add a Connection Indicator widget and to make sure your code is writing at least one key during initialization or disabled to trigger the connection indicator. The connection indicator can be moved or re-sized if the Editable checkbox is checked.

Recognizing Shuffleboard Connectivity

NetworkTables: not connected

Shuffleboard indicates if it is connected or not in the bottom right corner of the application as shown in the image above.

Recognizing Glass Connectivity

Glass - Connected (127.0.0.1)

Glass displays if it is connected or not in the bar across the top. See this [page](#) for more on configuring the connection.

Troubleshooting Connectivity

If the Dashboard does not connect to the Robot (after the Driver Station has connected to the robot) the recommended troubleshooting steps are:

1. Restart the Dashboard (there is no need to restart the Driver Station software)
2. If that doesn't work, restart the Robot Code using the Restart Robot Code button on the Diagnostics tab of the Driver Station
3. If it still doesn't connect, verify that the Team Number / Server is set properly in the Dashboard and that your Robot Code writes a value during initialization or disabled

23.1 Introduction to PathWeaver

Autonomous is an important section of the match; it is exciting when robots do impressive things in autonomous. In order to score, the robot usually need to go somewhere. The faster the robot arrives at that location, the sooner it can score points! The traditional method for autonomous is driving in a straight line, turning to a certain angle, and driving in a straight line again. This approach works fine, but the robot spends a non-negligible amount of time stopping and starting again after each straight line and turn.

A more advanced approach to autonomous is called “path planning”. Instead of driving in a straight line and turning once the line is complete, the robot continuously moves, driving with a curve-like motion. This can reduce turning stoppage time.

WPILib contains a trajectory generation suite that can be used by teams to generate and follow trajectories. This series of articles will go over how to generate and visualize trajectories using PathWeaver. For a comprehensive tutorial on following trajectories, please visit the [end-to-end trajectory tutorial](#).

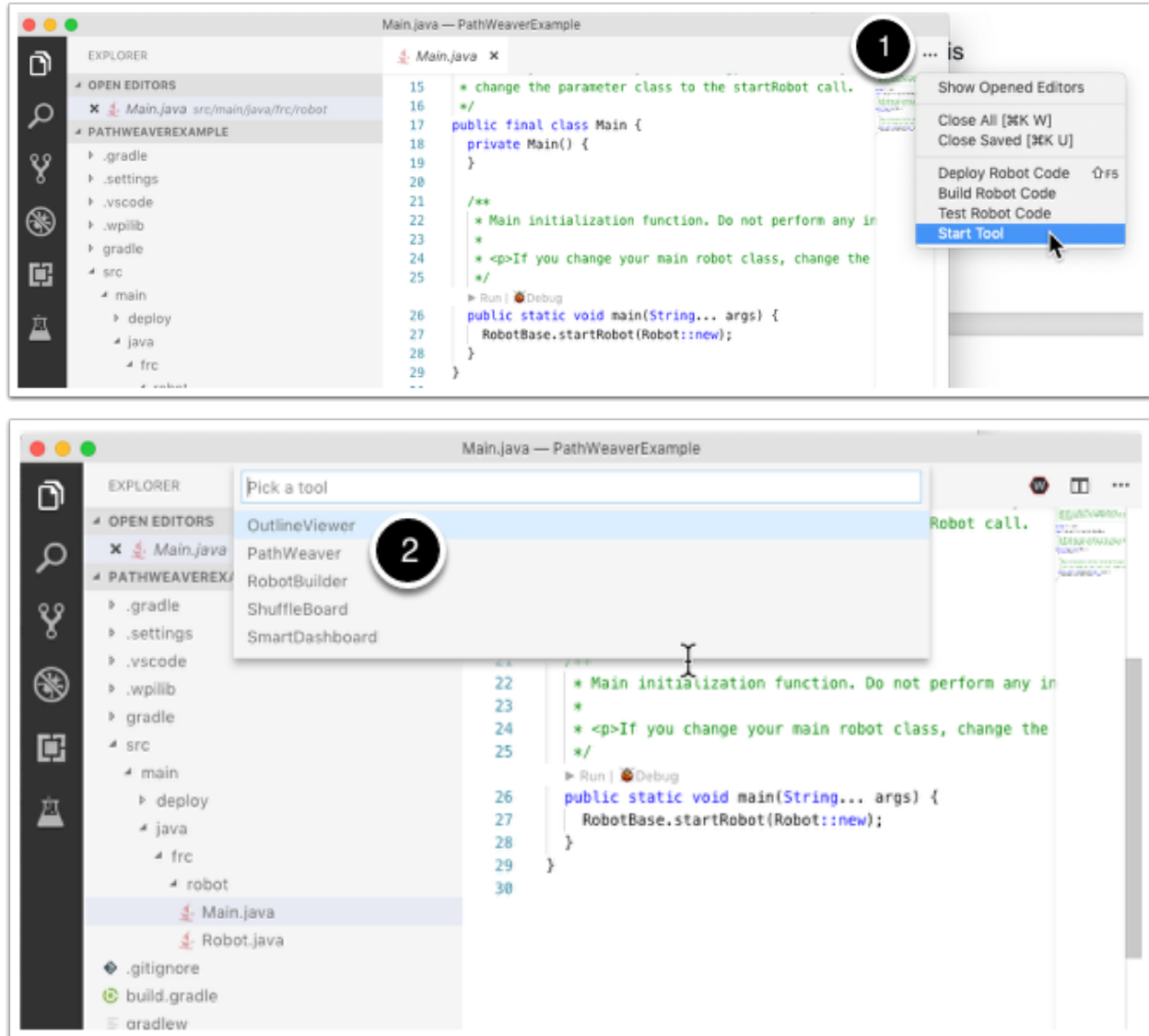
Note: *Trajectory following* code is required to use PathWeaver. We recommend that you start with Trajectory following and get that working with simple paths. From there you can continue on to testing more complicated paths generated by PathWeaver.

23.2 Creating a Pathweaver Project

PathWeaver is the tool used to draw the paths for a robot to follow. The paths for a single program are stored in a PathWeaver project.

23.2.1 Starting PathWeaver

PathWeaver is started by clicking on the ellipsis icon in the top right of the corner of the Visual Studio Code interface. You must select a source file from the WPILib project to see the icon. Then click on “Start tool” and then click on “PathWeaver” as shown below.



23.2.2 Creating the Project

To create a PathWeaver project, click on “Create project” and then fill out the project creation form. Notice that hovering over any of the fields in the form will display more information about what is required.

PathWeaver - 2021.1.2

Create Project...

Project Directory

Output Directory

Game

Length Unit

Export Unit

Max Velocity m/sec

Max Acceleration m/sec²

Wheel Base m

Project Directory: This is normally the top level project directory that contains the build.gradle and src files for your robot program. Choosing this directory is the expected way to use PathWeaver and will cause it to locate all the output files in the correct directories for automatic path deployment to your robot.

Output directory: The directory where the paths are stored for deployment to your robot. If you specified the top level project folder of our robot project in the previous step (as recommended) filling in the output directory is optional.

Game: The game (which FRC® game is being used) will cause the correct field image overlay to be used. You can also create your own field images and the procedure will be described later in this series.

Length Unit: The units to be used in describing your robot and for the field measurements when visualizing trajectories using PathWeaver.

Export Unit: The units to be used when exporting the paths and waypoints. If you are planning to use WPILib Trajectories, then you should choose *Always Meters*. Choosing *Same as Project* will export in the same units as *Length Unit* above.

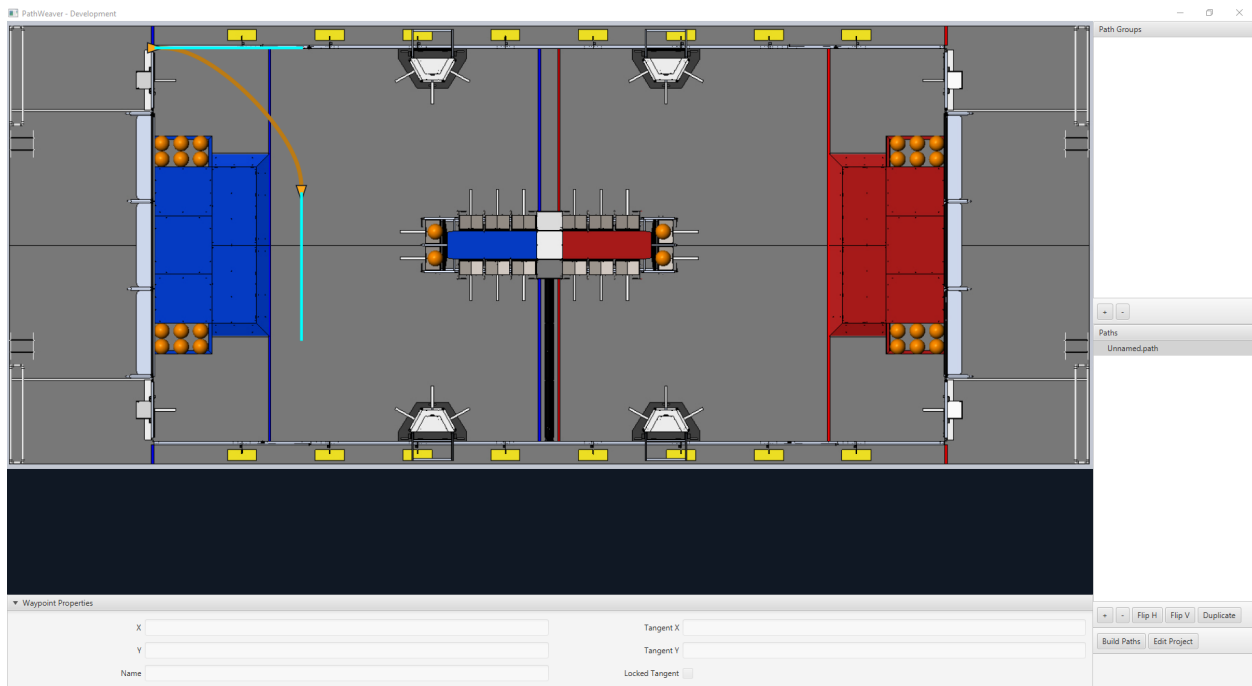
Max Velocity: The max speed of the robot for trajectory tracking. This does not need to be the maximum attainable speed of the robot, but just the max speed that you want to run

trajectories at.

Max Acceleration: The max acceleration of the robot for trajectory tracking. This does not need to be the maximum attainable acceleration of the robot, but just the max acceleration that you want to run trajectories at.

Wheel Base: The distance between the left and right wheels of your robot. This is used to ensure that no wheel on a differential drive will go over the specified max velocity around turns.

23.2.3 PathWeaver User Interface

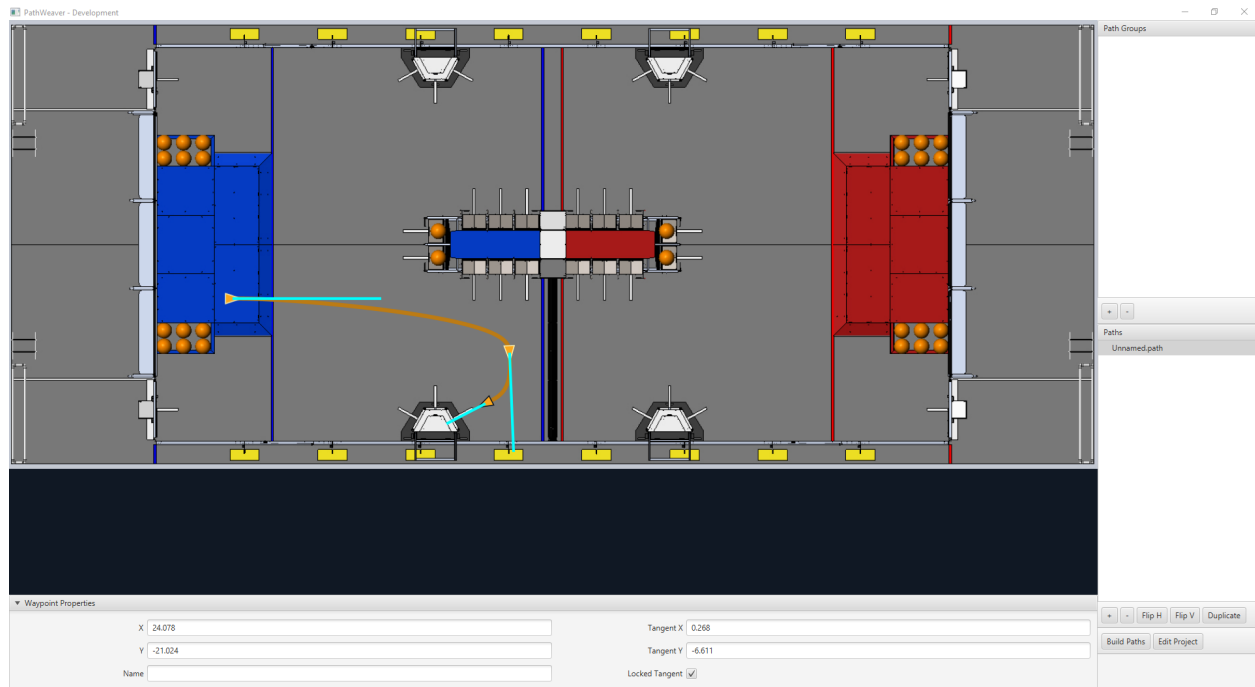


The PathWeaver user interface consists of the following:

1. The field area in the top left corner, which takes up most of the PathWeaver window. Trajectories are drawn on this part of the program.
2. The properties of the currently selected waypoint are displayed in the bottom pane. These properties include the X and Y, along with the tangents at each point.
3. A group of paths (or an “autonomous” mode) is displayed on the upper right side of the window. This is a convenient way of seeing all of the trajectories in a single auto mode.
4. The individual paths that a robot might follow are displayed in the lower right side of the window.
5. The “Build Paths” button will export the trajectories in a JSON format. These JSON files can be used from the robot code to follow the trajectory.
6. The “Edit Project” button allows you to edit the project properties.

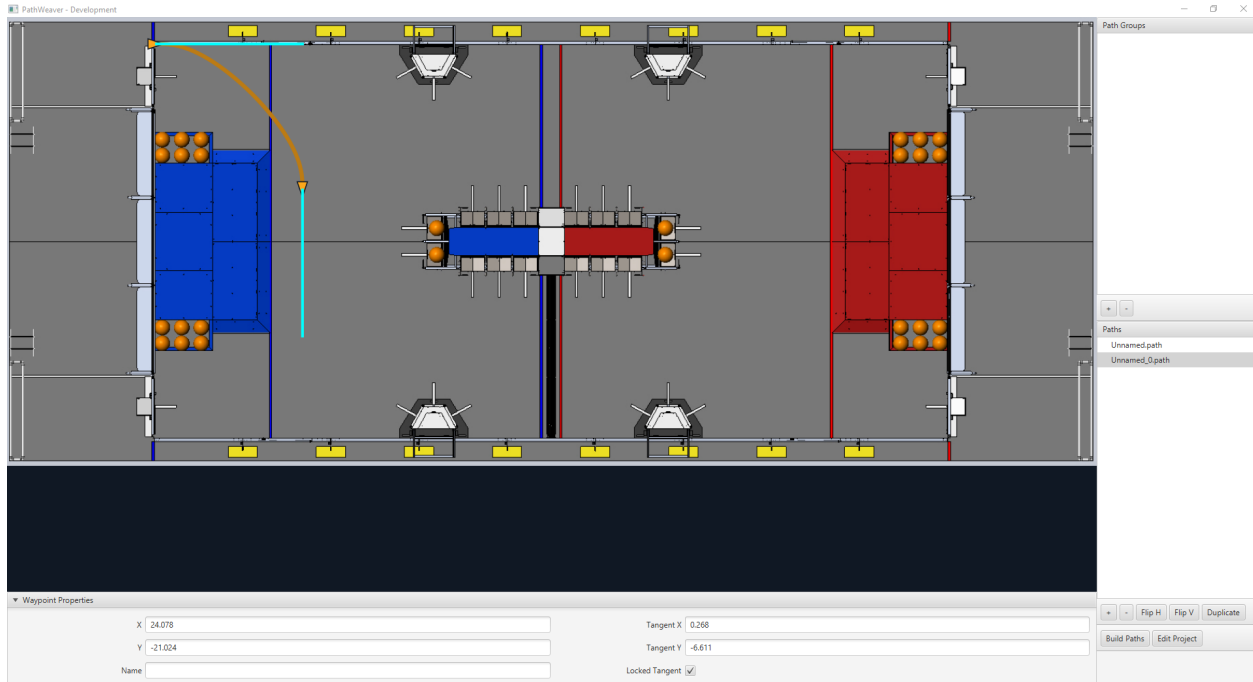
23.3 Visualizing PathWeaver Trajectories

PathWeaver's primary feature is to visualize trajectories. The following images depict a smooth trajectory that represents a trajectory that a robot might take during the autonomous period. Paths can have any number of waypoints that can allow more complex driving to be described. In this case there are 3 waypoints (including the start and stop) depicted with the triangle icons. Each waypoint consists of a X, Y position on the field as well as a robot heading described as the X and Y tangent lines.



23.3.1 Creating the Initial Trajectory

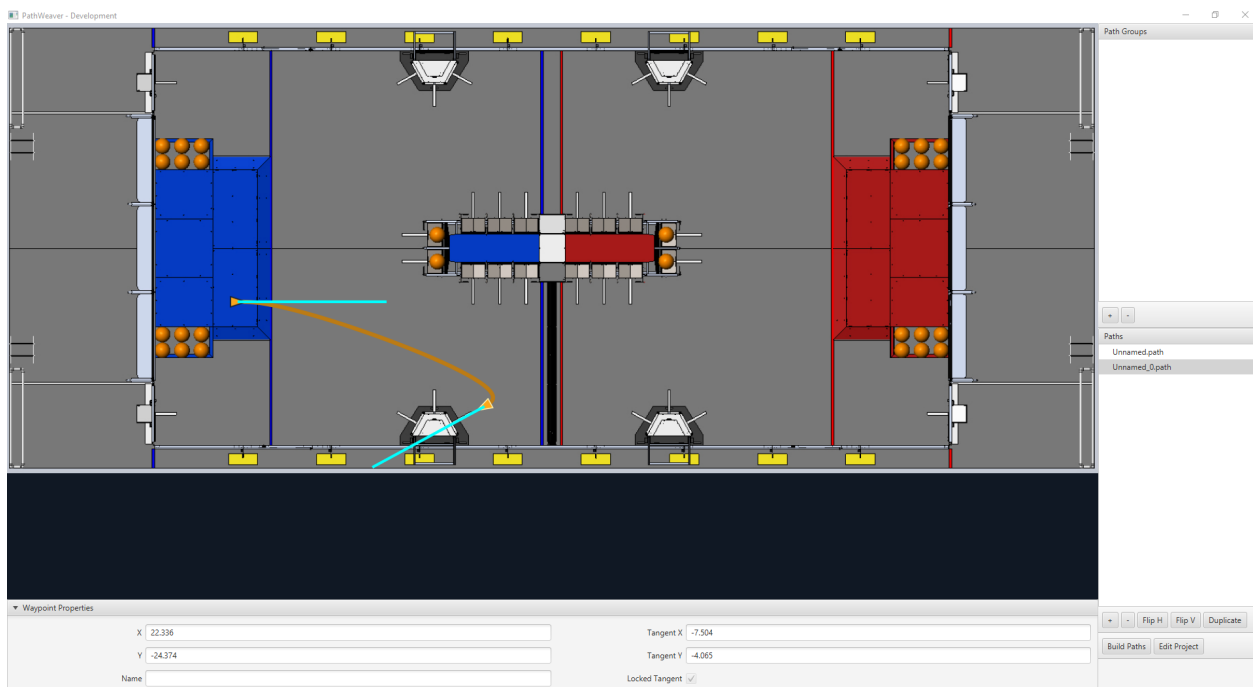
To start creating a trajectory, click the + (plus) button in the path window. A default trajectory will be created that probably does not have the proper start and end points that you desire. The path also shows the tangent vectors (teal lines) for the start and end points. Changing the angle of the tangent vectors changes the shape of the trajectory.



Drag the start and end points of the trajectory to the desired locations. Notice that in this case, the default trajectory does not start in a legal spot for the 2019 game. We can drag the initial waypoint to make the robot start on the HAB.

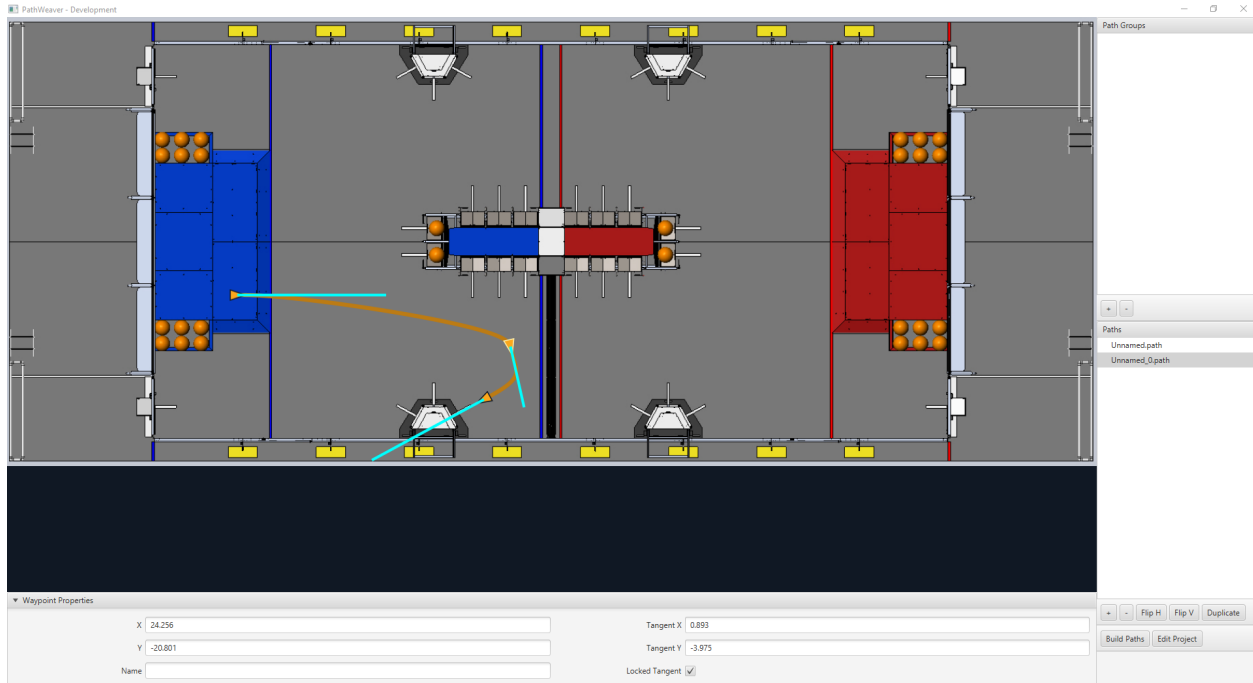
23.3.2 Changing a Waypoint Heading

The robot heading can be changed by dragging the tangent vector (teal) line. Here, the final waypoint was dragged to the desired final pose and was rotated to face the rocket.



23.3.3 Adding Additional Waypoints to Control the Robot Path

Adding additional waypoints and changing their tangent vectors can affect the path that is followed. Additional waypoints can be added by dragging in the middle of the path. In this case, we added another waypoint in the middle of the path.



23.3.4 Locking the Tangent Lines

Locking tangent lines prevents them from changing when the trajectory is being manipulated. The tangent lines will also be locked when the point is moved.

23.3.5 More Precise control of Waypoints

While PathWeaver makes it simple to draw trajectories that the robot should follow, it is sometimes hard to precisely set where the waypoints should be placed. In this case, setting the waypoint locations can be done by entering the X and Y value which might come from an accurate CAD model of the field. The points can be entered in the X and Y fields when a waypoint is selected.

23.4 Creating Path Groups

Path Groups are a way of visualizing where one path ends and the next one starts. An example is when the robot program drives one path, does something after the path has completed, drives to another location to obtain a game piece, then back again to score it. It's important that the start and end points of each path in the group have common end and start points. By adding all the paths to a single path group and selecting the group, all paths in that group will be shown. Then each path can be edited while viewing all the paths.

23.4.1 Creating a Path Group

Press the “plus” button underneath Path Groups. Then drag the Paths from the Paths section into your Path Group.

Each path added to a path group will be drawn in a different color making it easy to figure out what the name is for each path.

If there are multiple paths in a group, selection works as follows:

1. Selecting the group displays all paths in the group making it easy to see the relationship between them. Any waypoint on any of the paths can be edited while the group is selected and it will only change the path containing the waypoint.
2. Selecting on a single path in the group will only display that path, making it easy to precisely see what all the waypoints are doing and preventing clutter in the interface if multiple paths cross over or are close to each other.

23.5 Importing a PathWeaver JSON

The `TrajectoryUtil` class can be used to import a PathWeaver JSON into robot code to follow it. This article will go over importing the trajectory. Please visit the [end-to-end trajectory tutorial](#) for more information on following the trajectory.

The `fromPathweaverJson` (Java) / `FromPathweaverJson` (C++) static methods in `TrajectoryUtil` can be used to create a trajectory from a JSON file stored on the roboRIO file system.

Important: To be compatible with the Field2d view in the simulator GUI, the coordinates for the exported JSON have changed. Previously (before 2021), the range of the y-coordinate was from -27 feet to 0 feet whereas now, the range of the y-coordinate is from 0 feet to 27 feet (with 0 being at the bottom of the screen and 27 feet being at the top). This should not affect teams who are correctly *resetting their odometry to the starting pose of the trajectory* before path following.

Note: PathWeaver places JSON files in `src/main/deploy/paths` which will automatically be placed on the roboRIO file system in `/home/lvuser/deploy/paths` and can be accessed using `getDeployDirectory` as shown below.

Java

C++

```
String trajectoryJSON = "paths/YourPath.wpilib.json";
Trajectory trajectory = new Trajectory();

@Override
public void robotInit() {
    try {
        Path trajectoryPath = Filesystem.getDeployDirectory().toPath().
        ↪ resolve(trajectoryJSON);
        trajectory = TrajectoryUtil.fromPathweaverJson(trajectoryPath);
    } catch (IOException ex) {
        DriverStation.reportError("Unable to open trajectory: " + trajectoryJSON, ex.
        ↪ getStackTrace());
    }
}
```

```
#include <frc/Filesystem.h>
#include <frc/trajectory/TrajectoryUtil.h>
#include <wpi/Path.h>
#include <wpi/SmallString.h>

frc::Trajectory trajectory;

void Robot::RobotInit() {
    wpi::SmallString<64> deployDirectory;
    frc::filesystem::GetDeployDirectory(deployDirectory);
    wpi::sys::path::append(deployDirectory, "paths");
    wpi::sys::path::append(deployDirectory, "YourPath.wpilib.json");

    trajectory = frc::TrajectoryUtil::FromPathweaverJson(deployDirectory);
}
```

In the examples above, YourPath should be replaced with the name of your path.

Warning: Loading a PathWeaver JSON from file in Java can take more than one loop iteration so it is highly recommended that the robot load these paths on startup.

23.6 Adding field images to PathWeaver

Here are instructions for adding your own field image using the 2019 game as an example.

Games are loaded from the ~/PathWeaver/Games on Linux and macOS or %USERPROFILE%/PathWeaver/Games directory on Windows. The files can be in either a game-specific subdirectory, or in a zip file in the Games directory. The ZIP file must follow the same layout as a game directory; the JSON file must be in the root of the ZIP file (cannot be in a subdirectory).

Download the example *FIRST* Destination Deep Space field definition [here](#). Other field definitions are available in the [PathWeaver GitHub repository](#).

23.6.1 File Layout

```
~/PathWeaver
/Games
/Custom Game
  custom-game.json
  field-image.png
OtherGame.zip
```

23.6.2 JSON Format

```
{
  "game": "game name",
  "field-image": "relative/path/to/img.png",
  "field-corners": {
    "top-left": [x, y],
    "bottom-right": [x, y]
  },
  "field-size": [width, length],
  "field-unit": "unit name"
}
```

The path to the field image is relative to the JSON file. For simplicity, the image file should be in the same directory as the JSON file.

The field corners are the X and Y coordinates of the top-left and bottom-right pixels defining the rectangular boundary of the playable area in the field image. Non-rectangular playing areas are not supported.

The field size is the width and length of the playable area of the field in the provided units.

The field units are case-insensitive and can be in meters, cm, mm, inches, feet, yards, or miles. Singular, plural, and abbreviations are supported (e.g. “meter”, “meters”, and “m” are all valid for specifying meters)

Note: When making a new field image, a border (minimum of 20 pixels is recommended) should be left around the outside so that waypoints on the field edge are accessible.

Important: RobotBuilder has been updated to support the new commandbased framework! Unfortunately, this documentation is in the process of being updated for the new command-based framework. Individuals interested in updating this documentation can open a pull request on the [frc-docs](#) repository.

24.1 RobotBuilder - Introduction

24.1.1 RobotBuilder Overview

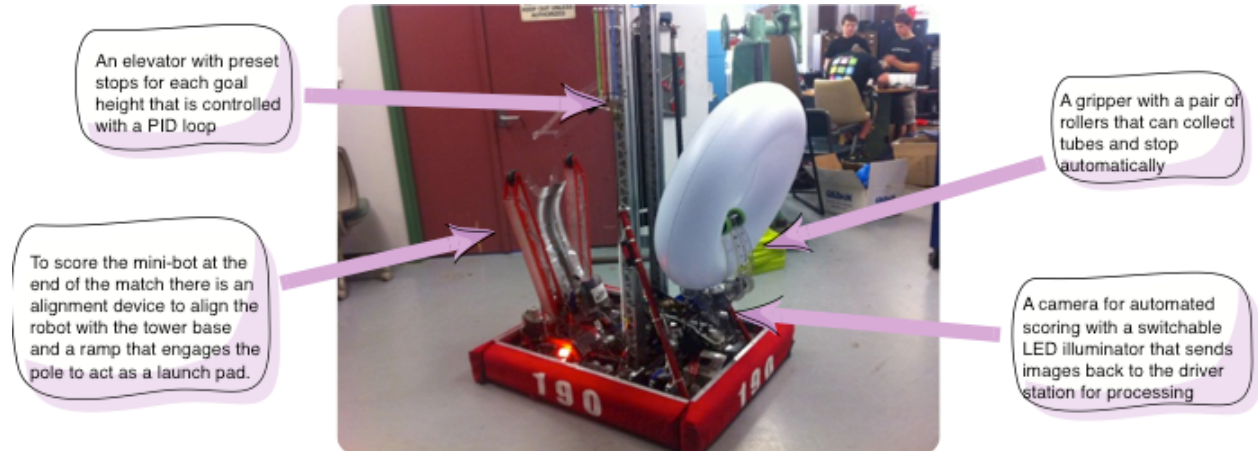
RobotBuilder is an application designed to aid the robot development process. RobotBuilder can help you:

- Generating boilerplate code.
- Organize your robot and figure out what its key subsystems are.
- Check that you have enough channels for all of your sensors and actuators.
- Generate wiring diagrams.
- Easily modify your operator interface.
- More...

Creating a program with RobotBuilder is a very straight forward procedure by following a few steps that are the same for any robot. This lesson describes the steps that you can follow. You can find more details about each of these steps in subsequent sections of the document.

Note: RobotBuilder generates code using the new Command Framework. For more details on the new framework see [Command Based Programming](#).

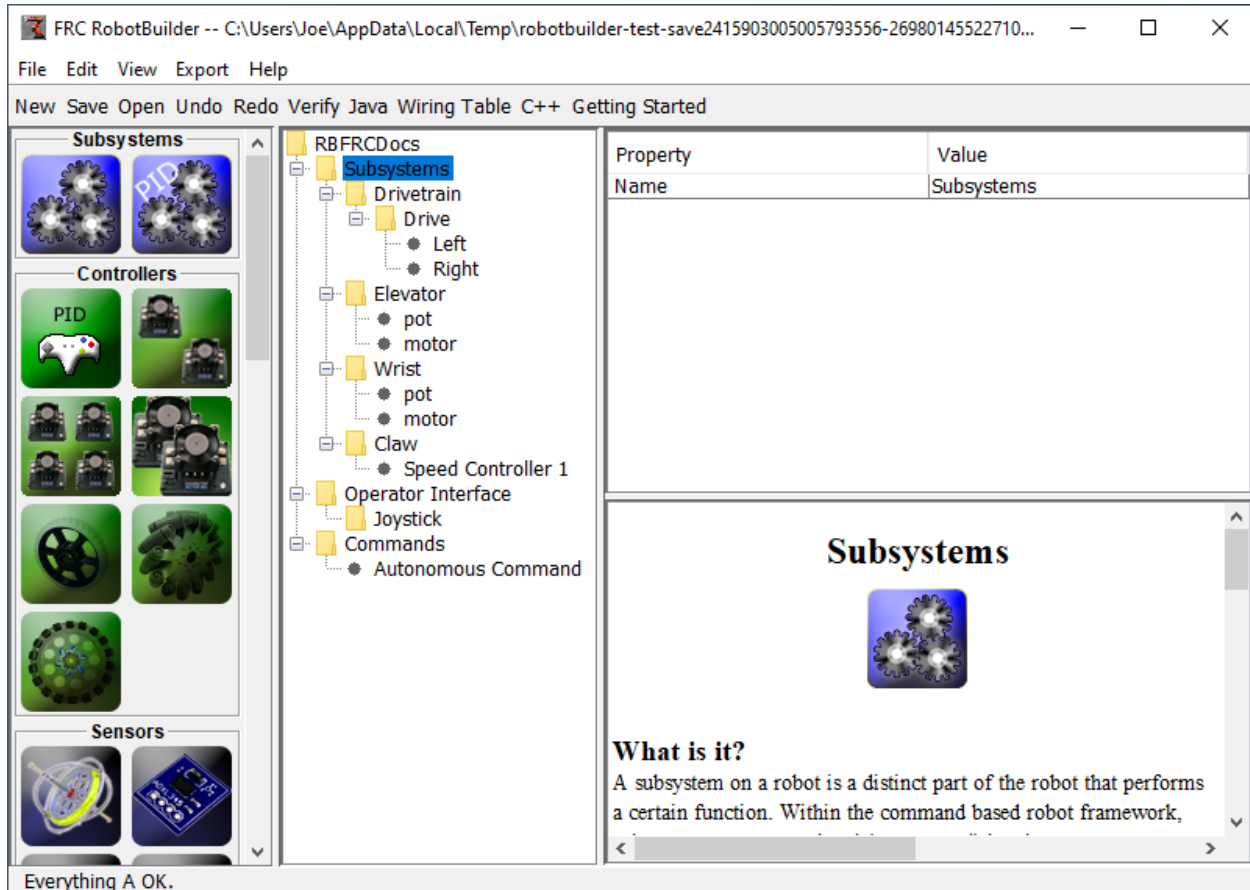
Divide the Robot into Subsystems



Your robot is naturally made up of a number of smaller systems like the drive trains, arms, shooters, collectors, manipulators, wrist joints, etc. You should look at the design of your robot and break it up into smaller, separately operated subsystems. In this particular example there is an elevator, a minibot alignment device, a gripper, and a camera system. In addition one might include the drive base. Each of these parts of the robot are separately controlled and make good candidates for subsystems.

For more information see [Creating a Subsystem](#).

Adding each Subsystem to the Project



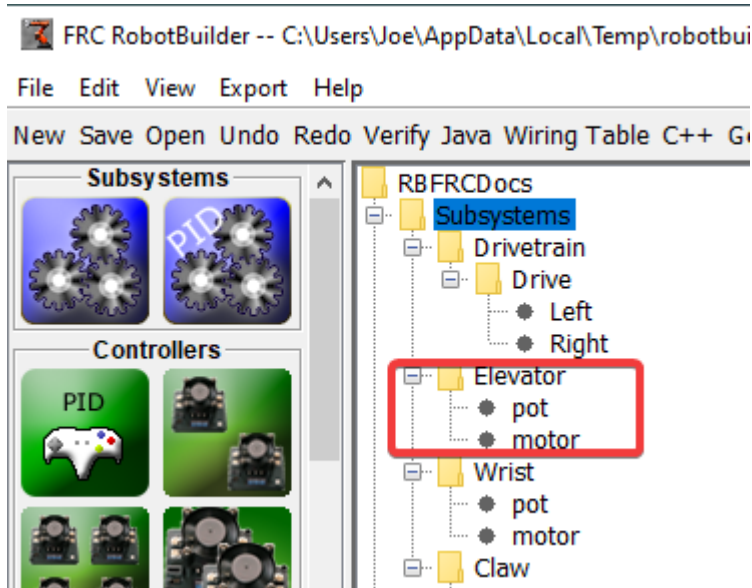
Each subsystem will be added to the “Subsystems” folder in the RobotBuilder and given a meaningful name. For each of the subsystems there are several attributes that get filled in to specify more information about the subsystems. In addition there are two types of subsystems that you might want to create:

1. **PIDSubsystems** - often it is desirable to control a subsystems operation with a PID controller. This is code in your program that makes the subsystem element, for example arm angle, move more quickly to a desired position then stop when reaching it. PIDSubsystems have the PID Controller code built-in and are often more convenient than adding it yourself. PIDSubsystems have a sensor that determines when the device has reached the target position and an actuator (motor controller) that is driven to the setpoint.
2. **Regular subsystem** - these subsystems don't have an integrated PID controller and are used for subsystems without PID control for feedback or for subsystems requiring more complex control than can be handled with the default embedded PID controller.

As you look through more of this documentation the differences between the subsystem types will become more apparent.

For more information see [Creating a Subsystem](#) and [Writing Code for a Subsystem](#).

Adding Components to each of the Subsystems



Each subsystem consists of a number of actuators, sensors and controllers that it uses to perform its operations. These sensors and actuators are added to the subsystem with which they are associated. Each of the sensors and actuators comes from the RobotBuilder palette and is dragged to the appropriate subsystem. For each, there are usually other properties that must be set such as port numbers and other parameters specific to the component.

In this example there is an Elevator subsystem that uses a motor and a potentiometer (motor and pot) that have been dragged to the Elevator subsystem.

Adding Commands That Describe Subsystem Goals

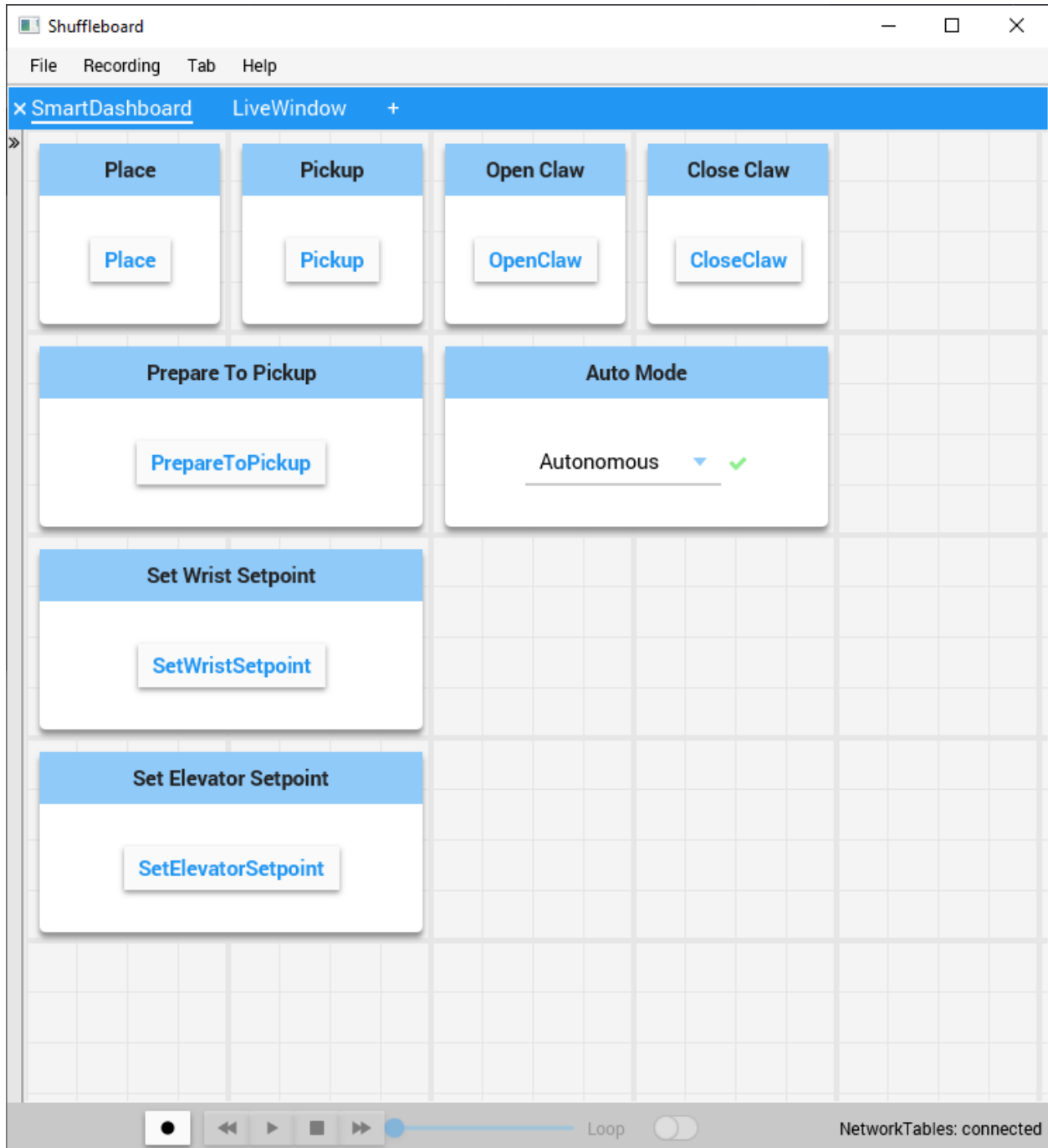
Commands are distinct goals that the robot will perform. These commands are added by dragging the command under the “Commands” folder. When creating a command, there are 7 choices (shown on the palette on the left of the picture):

- Normal commands - these are the most flexible command, you have to write all of the code to perform the desired actions necessary to accomplish the goal.
- Timed commands - these commands are a simplified version of a command that ends after a timeout
- Instant commands - these commands are a simplified version of a command that runs for one iteration and then ends
- Command groups - these commands are a combination of other commands running both in a sequential order and in parallel. Use these to build up more complicated actions after you have a number of basic commands implemented.
- Setpoint commands - setpoint commands move a PID Subsystem to a fixed setpoint, or the desired location.
- PID commands - these commands have a built-in PID controller to be used with a regular subsystem.

- Conditional commands - these commands select one of two commands to run at the time of initialization.

For more information see [Creating a Command](#) and [Writing Command Code](#).

Testing each Command



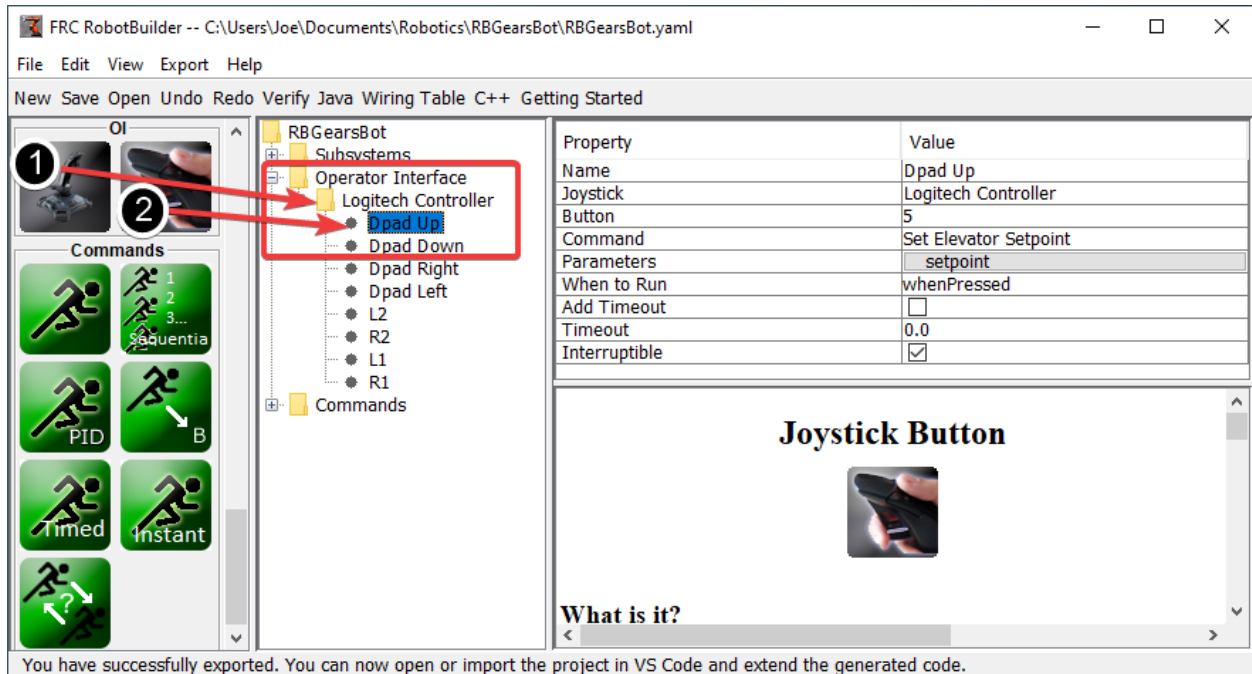
Each command can be run from Shuffleboard or SmartDashboard. This is useful for testing commands before you add them to the operator interface or to a command group. As long as

you leave the “Button on SmartDashboard” property checked, a button will be created on the SmartDashboard. When you press the button, the command will run and you can check that it performs the desired action.

By creating buttons, each command can be tested individually. If all the commands work individually, you can be pretty sure that the robot will work as a whole.

For more information see [Testing with Smartdashboard](#).

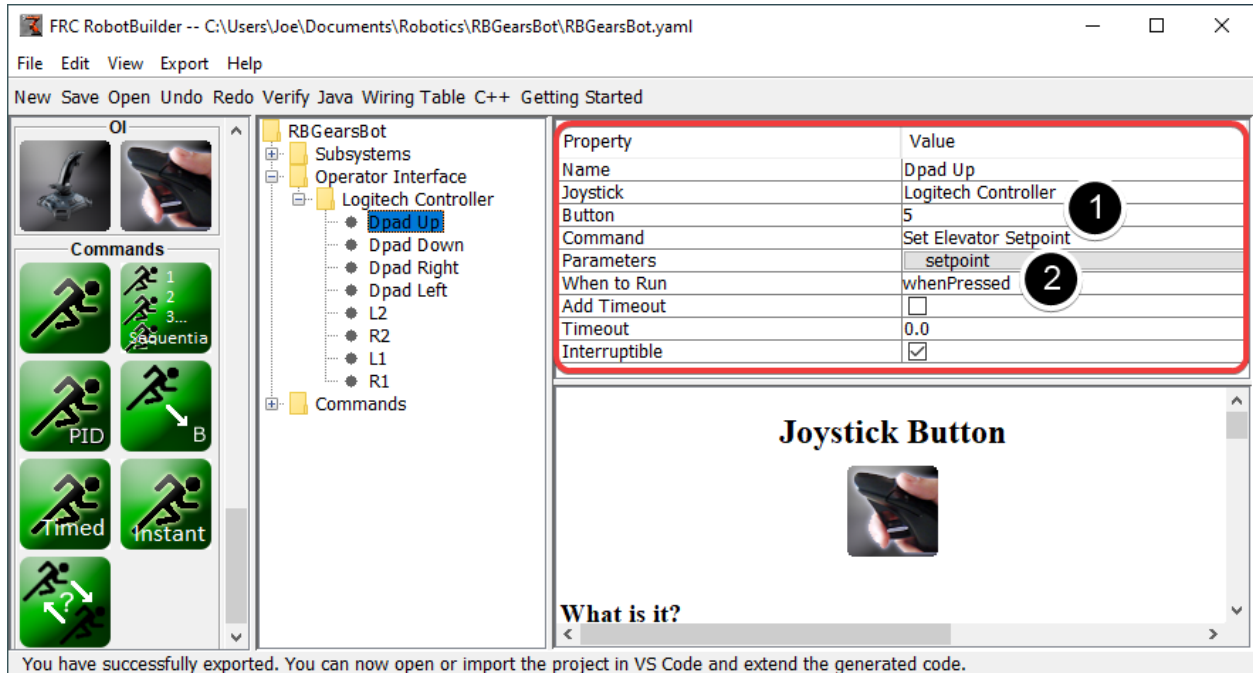
Adding Operator Interface Components



The operator interface consists of joysticks, gamepads and other HID input devices. You can add operator interface components (joysticks, joystick buttons) to your program in RobotBuilder. It will automatically generate code that will initialize all of the components and allow them to be connected to commands.

The operator interface components are dragged from the palette to the “Operator Interface” folder in the RobotBuilder program. First (1) add Joysticks to the program then put buttons under the associated joysticks (2) and give them meaningful names, like ShootButton.

Connecting the Commands to the Operator Interface

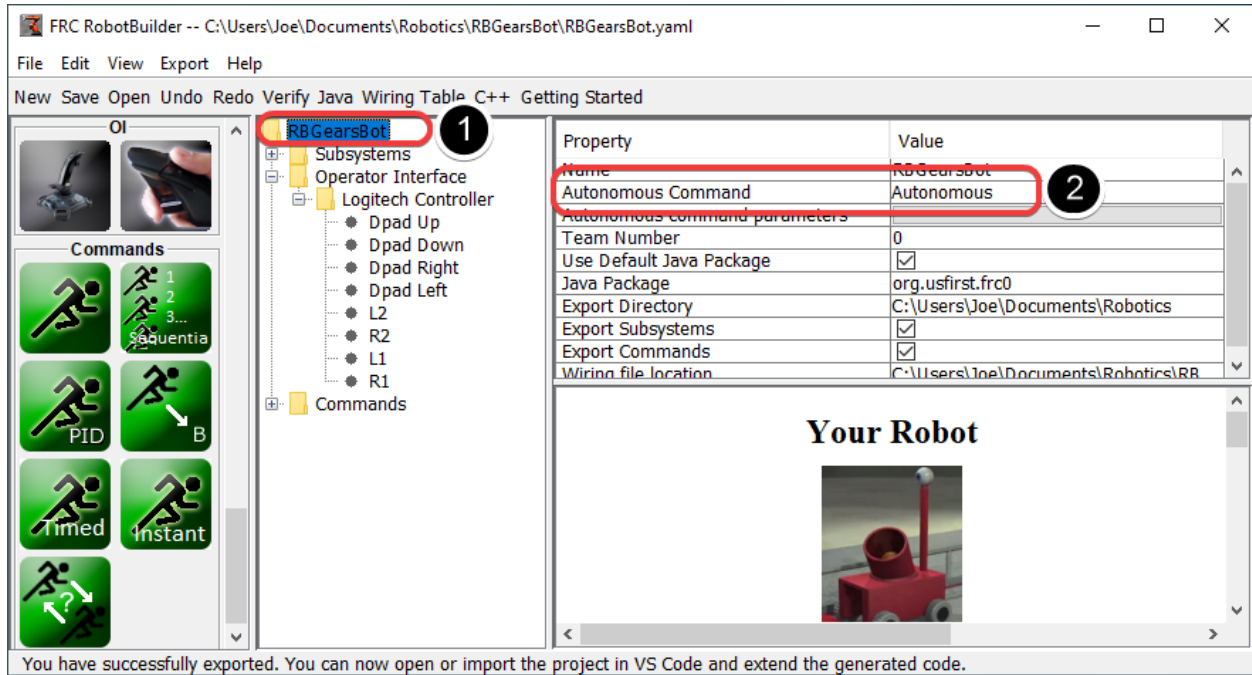


Commands can be associated with buttons so that when a button is pressed the command is scheduled. This should, for the most part, handle most of the tele-operated part of your robot program.

This is simply done by (1) adding the command to the JoystickButton object in the RobotBuilder program, then (2) setting the condition in which the command is scheduled.

For more information see [Connecting the Operator Interface to a Command](#).

Developing Autonomous Commands

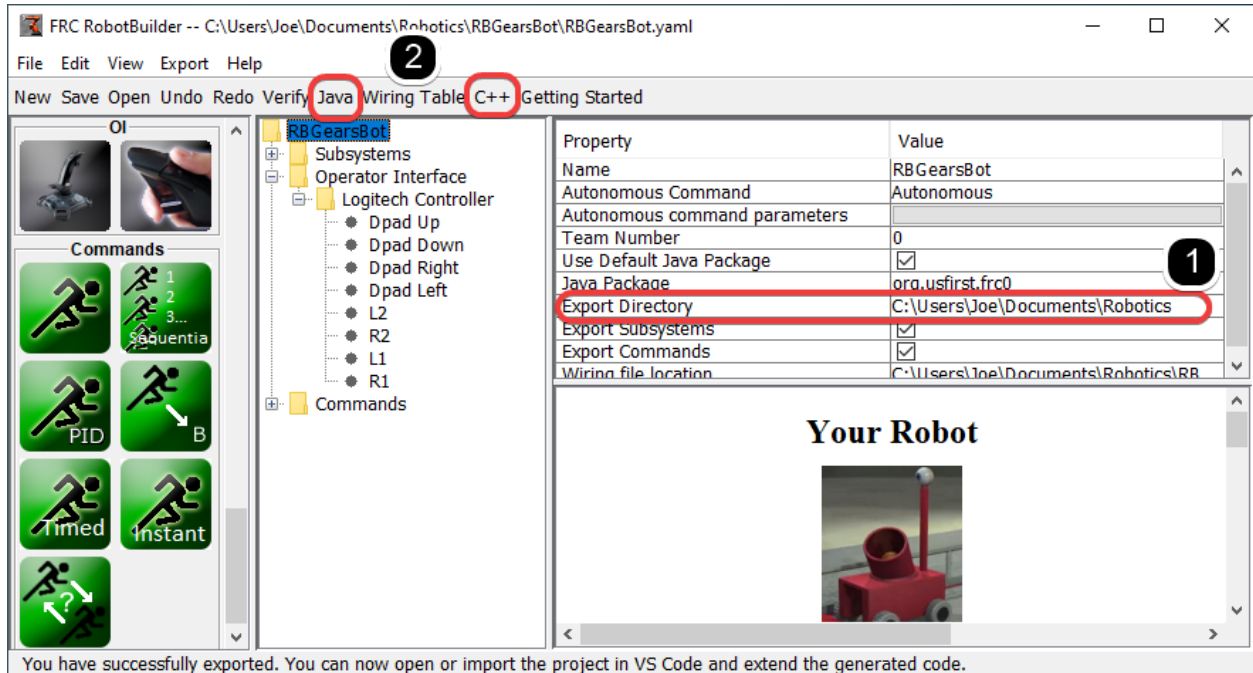


Commands make it simple to develop autonomous programs. You simply specify which command should run when the robot enters the autonomous period and it will automatically be scheduled. If you have tested commands as discussed above, this should simply be a matter of choosing which command should run.

Select the robot at the root of the RobotBuilder project (1), then edit the Autonomous Command property (2) to choose the command to run. It's that simple!

For more information see [Setting the Autonomous Commands](#).

Generating Code



At any point in the process outlined above you can have RobotBuilder generate a C++ or Java program that will represent the project you have created. This is done by specifying the location of the project in the project properties (1), then clicking the appropriate toolbar button to generate the code (2).

For more information see [Generating RobotBuilder Code](#).

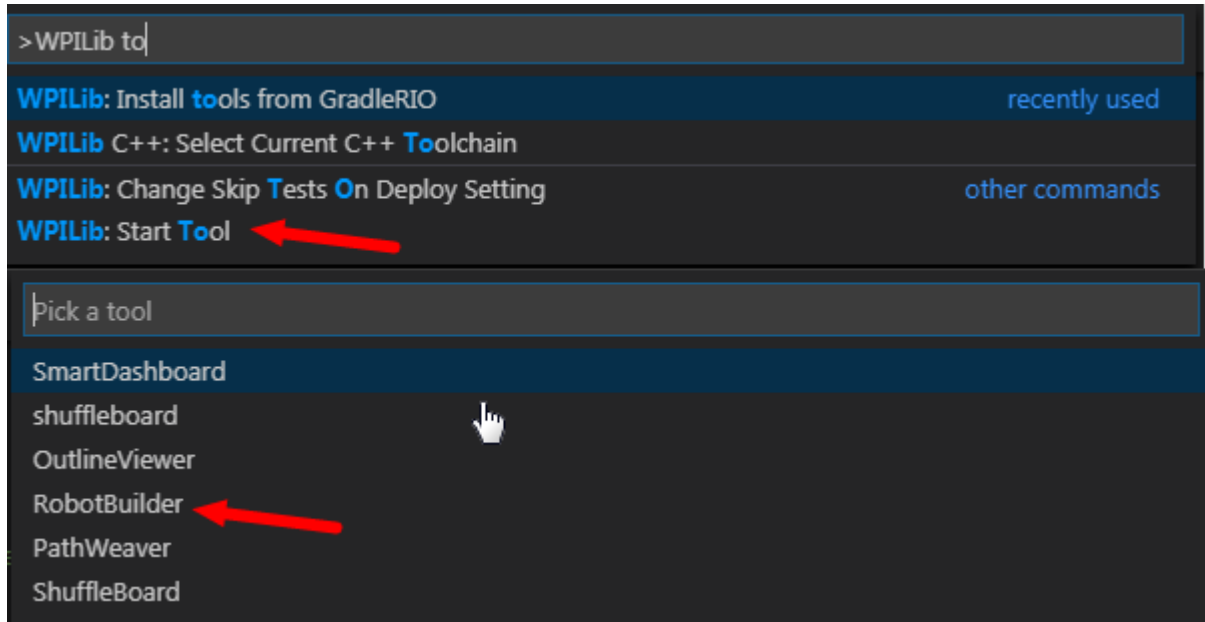
24.1.2 Starting RobotBuilder

Note: RobotBuilder is a Java program and as such should be able to run on any platform that is supported by Java. We have been running RobotBuilder on macOS, Windows, and various versions of Linux successfully.

Getting RobotBuilder

RobotBuilder is downloaded as part of the WPILib Offline Installer. For more information, see the [Windows/macOS/Linux installation guides](#)

Option 1 - Starting from Visual Studio Code

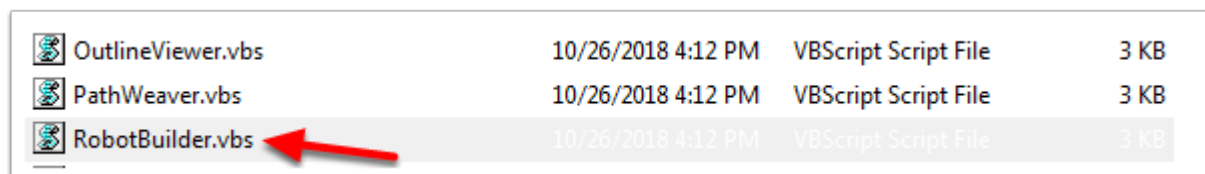


Press `Ctrl+Shift+P` and type “WPILib” or click the WPILib logo in the top right to launch the WPILib Command Palette. Select *Start Tool*, then select *Robot Builder*.

Option 2 - Shortcuts

Shortcuts are installed to the Windows Start Menu and the 2021 WPILib Tools folder on the desktop.

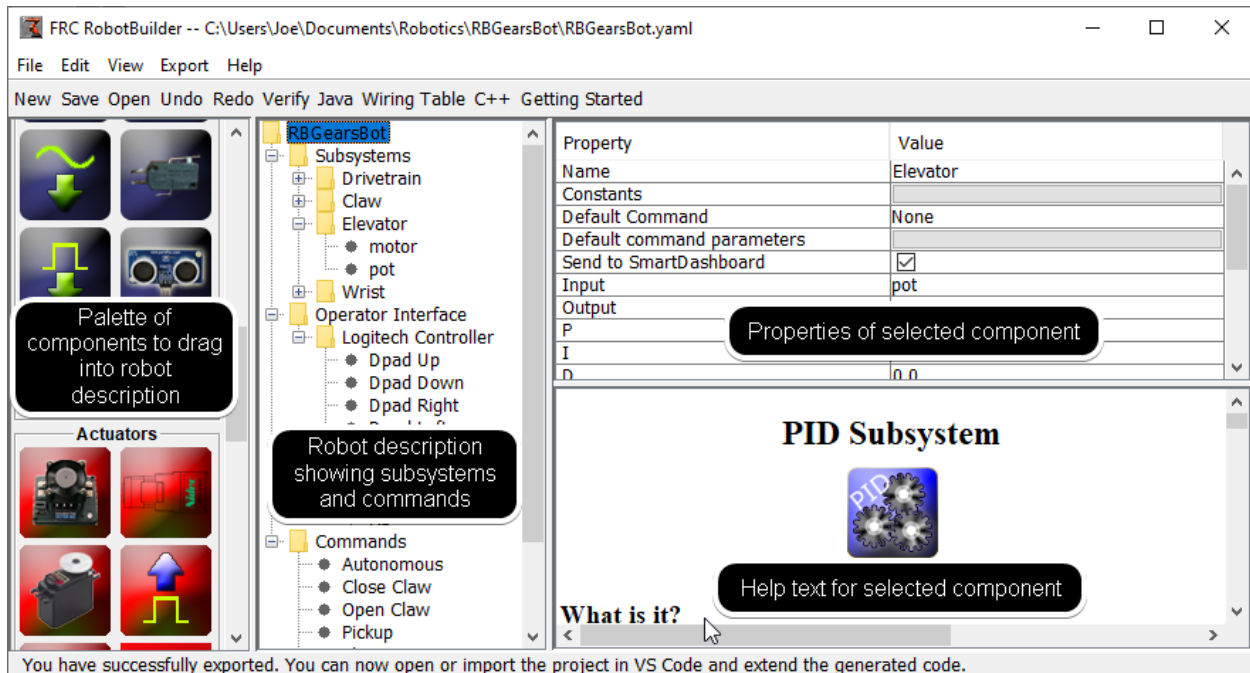
Option 3 - Running from the Script



The install process installs the tools to `~/wpilib/YYYY/tools` (where YYYY is the year and ~ is `C:\Users\Public` on Windows).

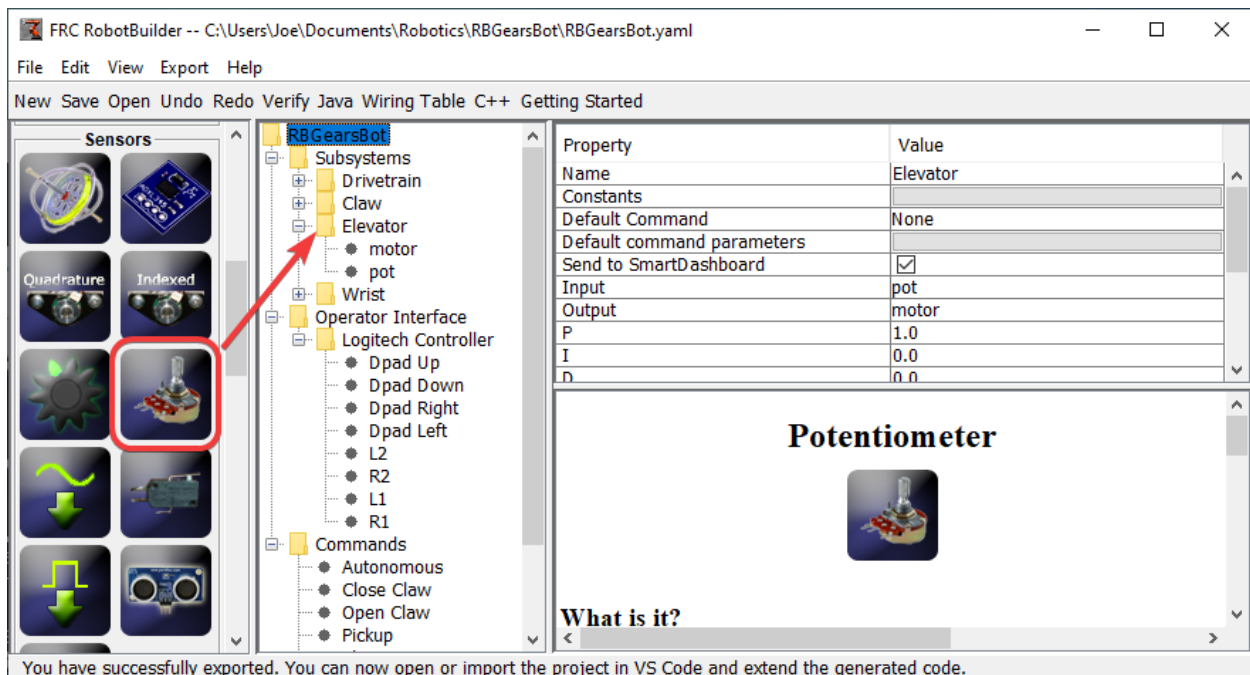
Inside this folder you will find `.vbs` (Windows) and `.py` (macOS/Linux) files that you can use to launch each tool. These scripts help launch the tools using the correct JDK and are what you should use to launch the tools.

24.1.3 RobotBuilder User Interface



RobotBuilder has a user interface designed for rapid development of robot programs. Almost all operations are performed by drag-and-drop or selecting options from drop-down lists.

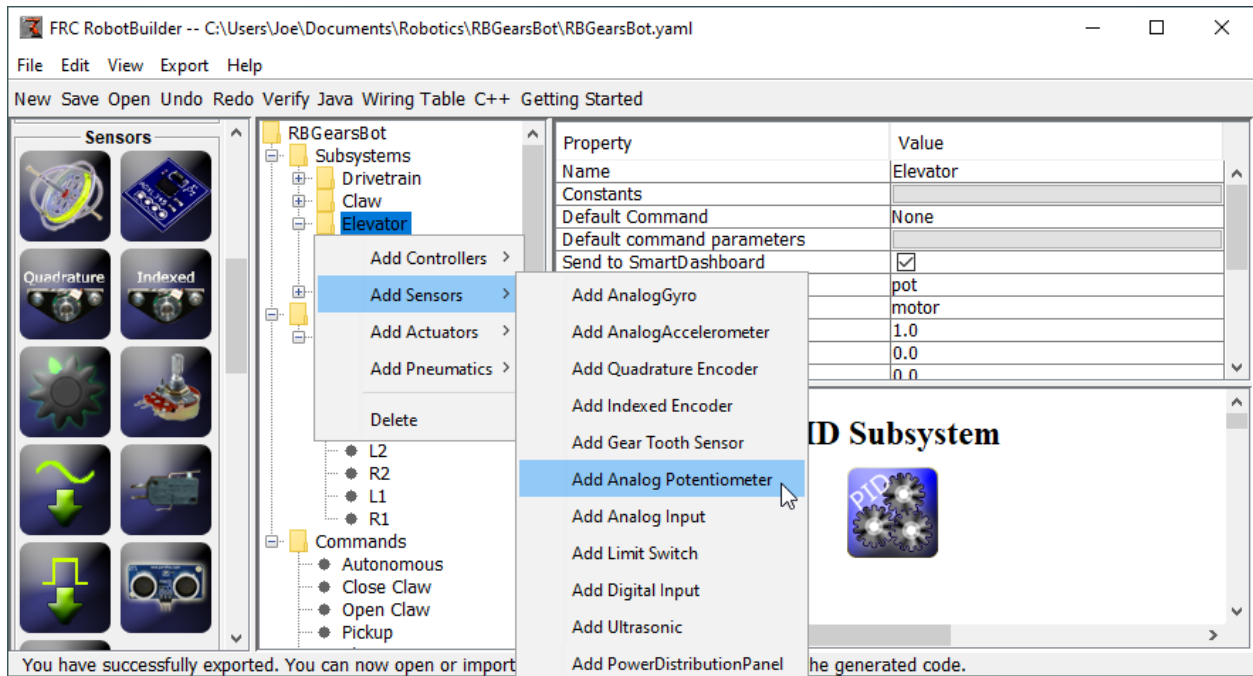
Dragging Items from the Palette to the Robot Description



You can drag items from the palette to the robot description by starting the drag on the palette item and ending on the container where you would like the item to be located. In this example,

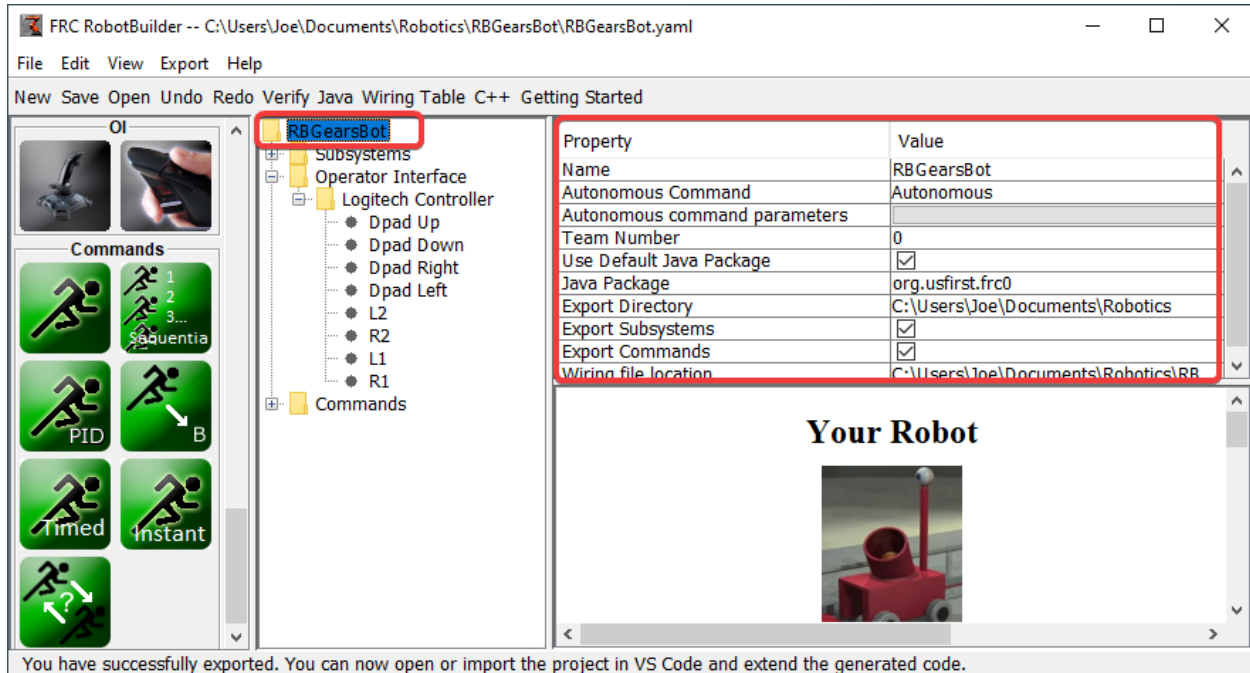
dropping a potentiometer to the Elevator subsystem.

Adding Components using the Right-Click Context Menu



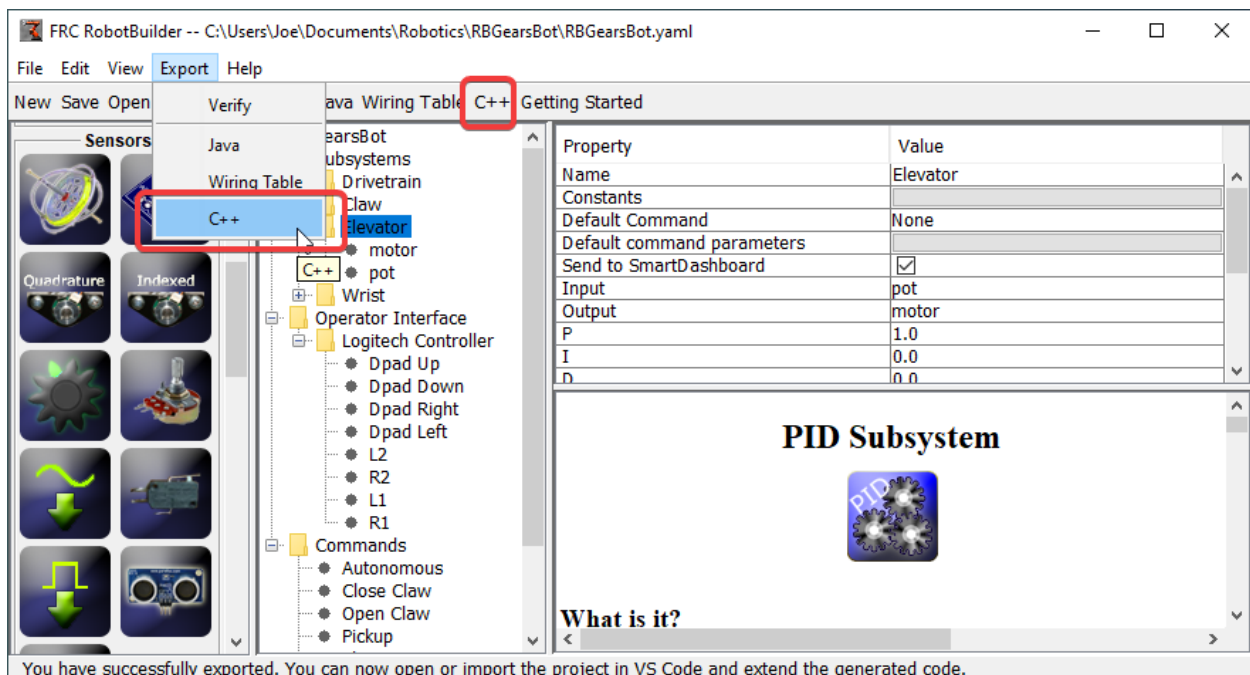
A shortcut method of adding items to the robot description is to right-click on the container object (Elevator) and select the item that should be added (Potentiometer). This is identical to using drag and drop but might be easier for some people.

Editing Properties of Robot Description Items



The properties for a selected item will appear in the properties viewer. The properties can be edited by selecting the value in the right hand column.

Using the Menu System



Operations for RobotBuilder can either be selected through the menu system or the equivalent item (if it is available) from the toolbar.

24.1.4 Setting up the Robot Project

The RobotBuilder program has some default properties that need to be set up so the generated program and other generated files work properly. This setup information is stored in the properties for robot description (the first line).

Robot Project Properties

The properties that describe the robot are:

Name - The name of the robot project that is created

Autonomous Command - the command that will run by default when the program is placed in autonomous mode

Autonomous Command Parameters - Parameters for the Autonomous Command

Team Number - The team number for the project, which will be used to locate the robot when deploying code.

Use Default Java Package - If checked RobotBuilder will use the default package (frc.robot). Otherwise you can specify a custom package name to be used.

Java Package - The name of the generated Java package used when generating the project code

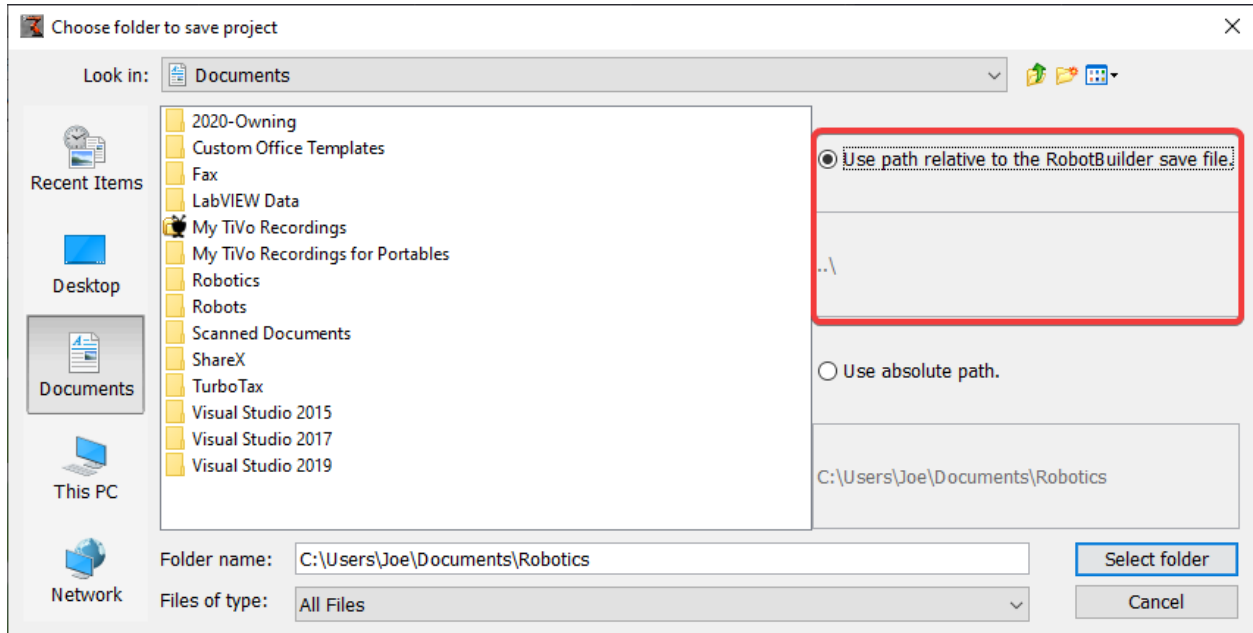
Export Directory - The folder that the project is generated into when Export to Java or C++ is selected

Export Subsystems - Checked if RobotBuilder should export the Subsystem classes from your project

Export Commands - Checked if RobotBuilder should export the Command classes from your project

Wiring File - the location of the html file that contains the wiring diagram for your robot

Using Source Control with the RobotBuilder Project

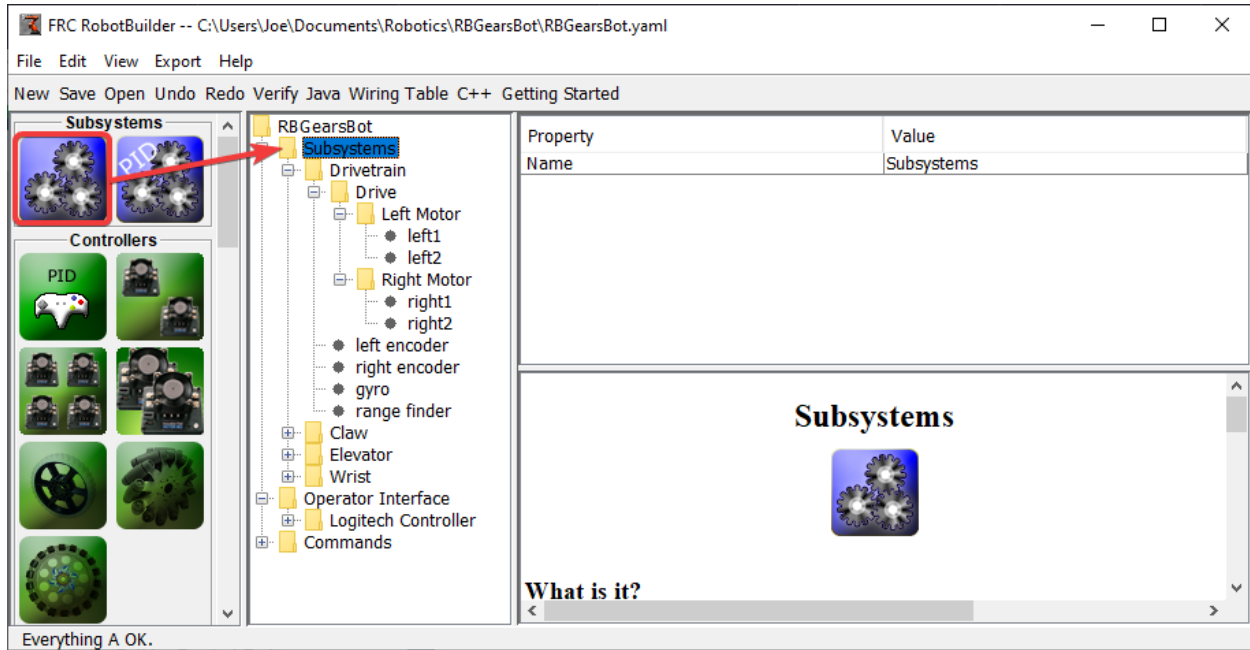


When using source control the project will typically be used on a number of computers and the path to the project directory might be different from one users computer to another. If the RobotBuilder project file is stored using an absolute path, it will typically contain the user name and won't be usable across multiple computers. To make this work, select "relative path" and specify the path as an directory offset from the project files. In the above example, the project file is stored in the folder just above the project files in the file hierarchy. In this case, the user name is not part of the path and it will be portable across all of your computers.

24.1.5 Creating a Subsystem

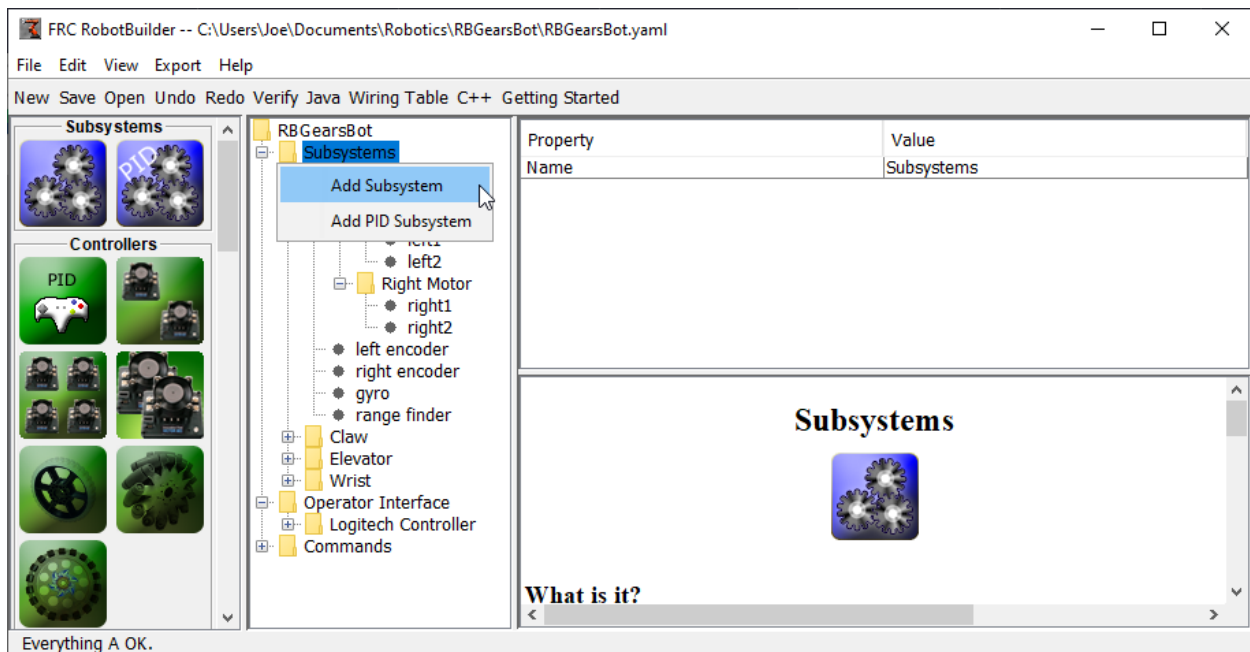
Subsystems are classes that encapsulate (or contain) all the data and code that make a subsystem on your robot operate. The first step in creating a robot program with the RobotBuilder is to identify and create all the subsystems on the robot. Examples of subsystems are grippers, ball collectors, the drive base, elevators, arms, etc. Each subsystem contains all the sensors and actuators that are used to make it work. For example, an elevator might have a Victor SPX speed controller and a potentiometer to provide feedback of the robot position.

Creating a Subsystem using the Palette



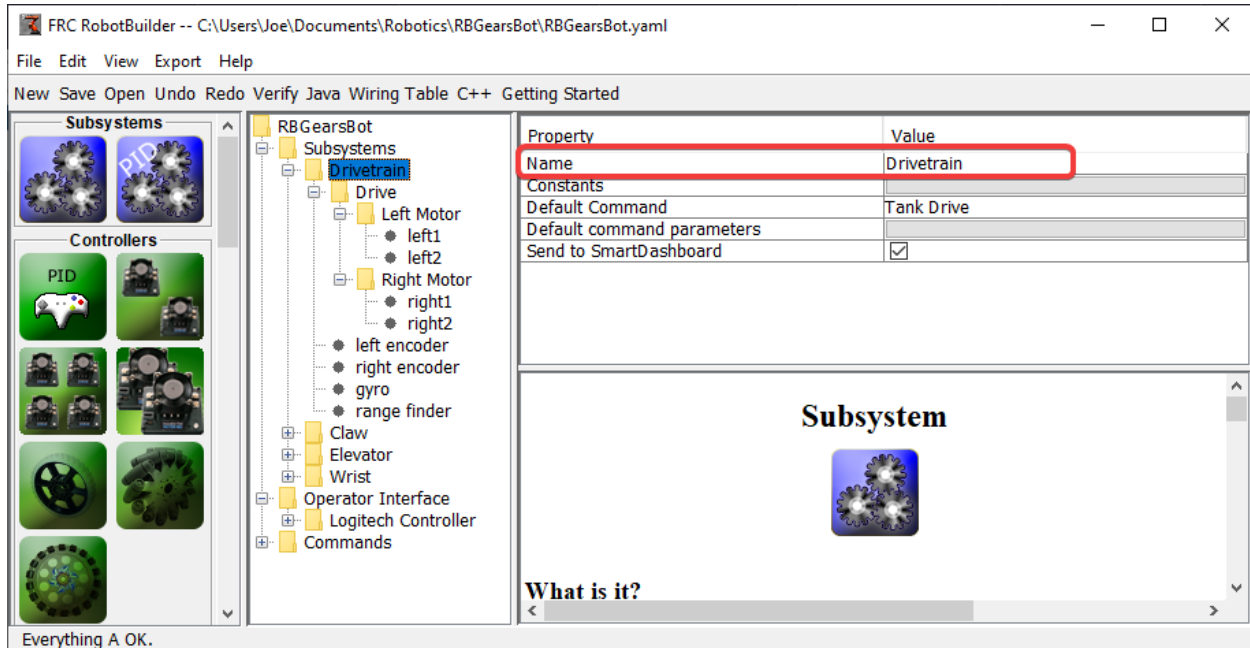
Drag the subsystem icon from the palette to the Subsystems folder in the robot description to create a subsystem class.

Creating a Subsystem using the Context Menu



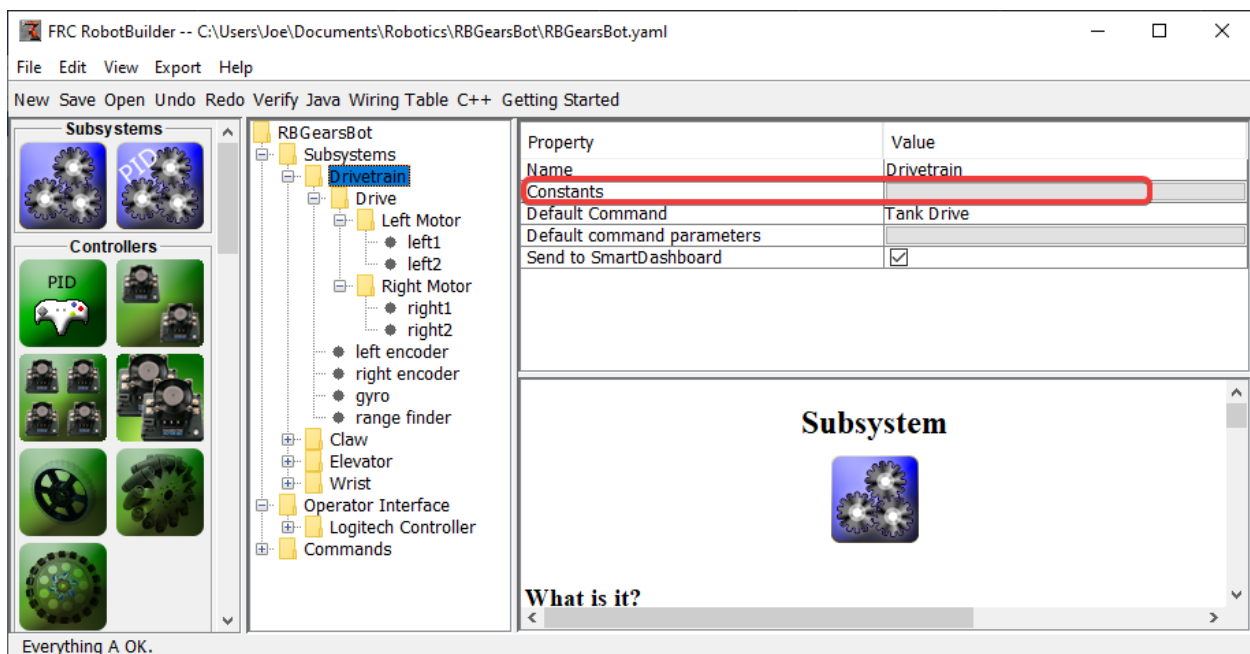
Right-click on the Subsystem folder in the robot description to add a subsystem to that folder.

Name the Subsystem



After creating the subsystem by either dragging or using the context menu as described above, simply type the name you would like to give the subsystem. The name can be multiple words separated by spaces, RobotBuilder will concatenate the words to make a proper Java or C++ class name for you.

Adding Constants

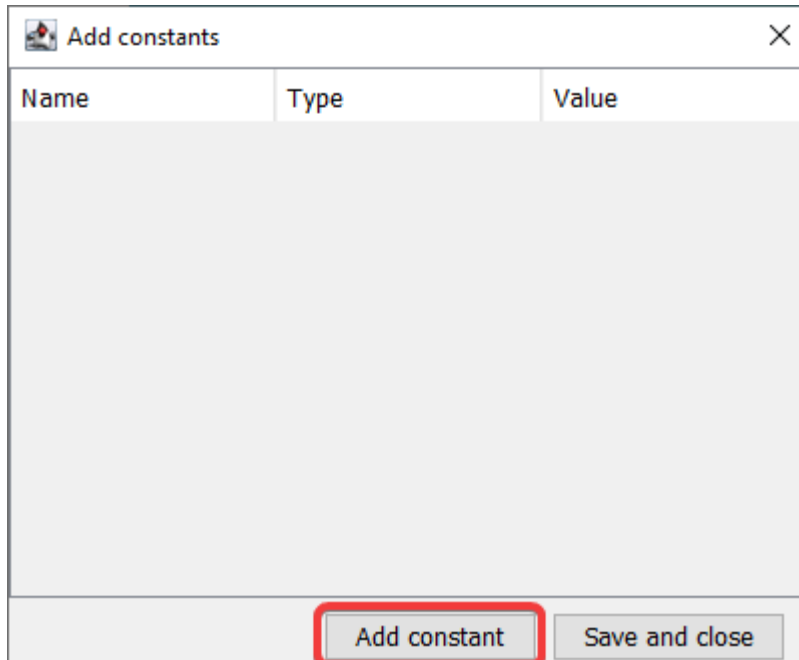


Constants are very useful to reduce the amount of magic numbers in your code. In subsys-

tems, they can be used to keep track of certain values, such as sensor values for specific heights of an elevator, or the speed at which to drive the robot.

By default, there will be no constants in a subsystem. Press the button next to “Constants” to open a dialog to create some.

Creating Constants



The constants table will be empty at first. Press “Add constant” to add one.

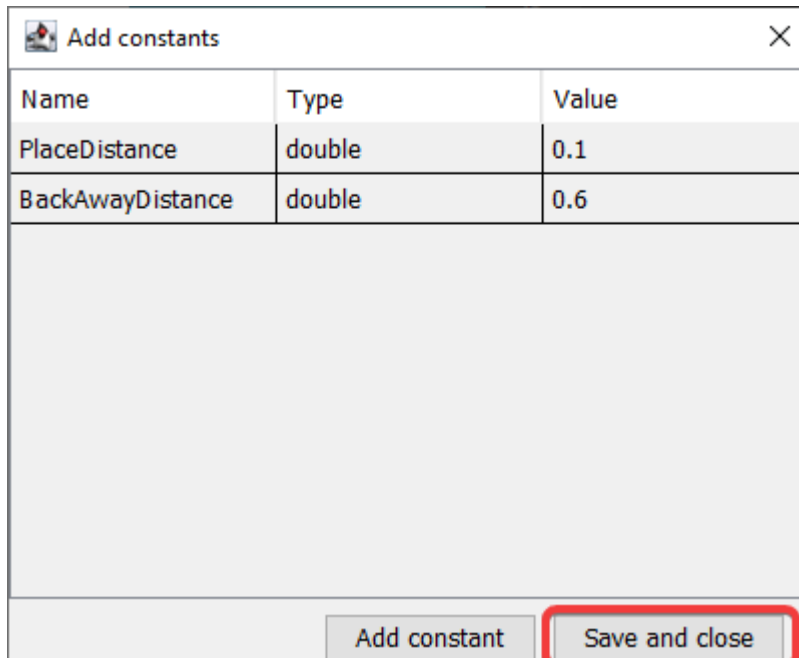
Add Constants

Name	Type	Value
[change me]	String	

Add constant Save and close

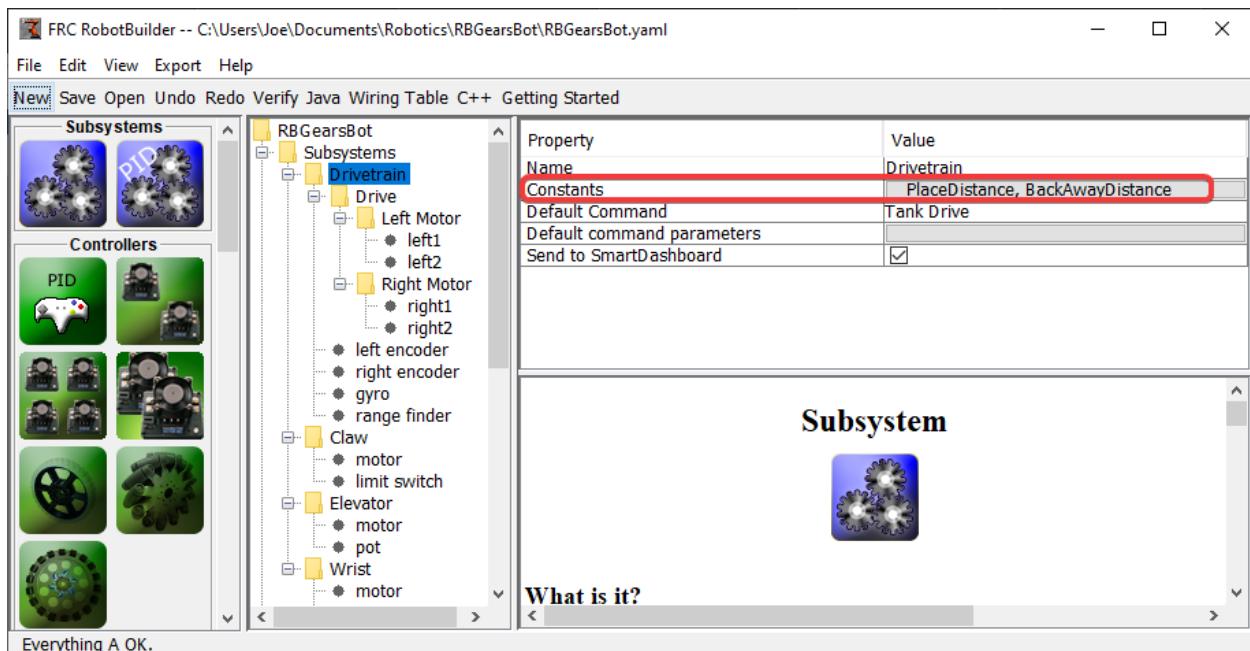
1. The name of the constant. Change this to something descriptive. In this example of a drivetrain some good constants might be "PlaceDistance" and "BackAwayDistance".
2. The type of the constant. This will most likely be a double, but you can choose from one of: String, double, int, long, boolean, or byte.
3. The value of the constant.

Saving Constants



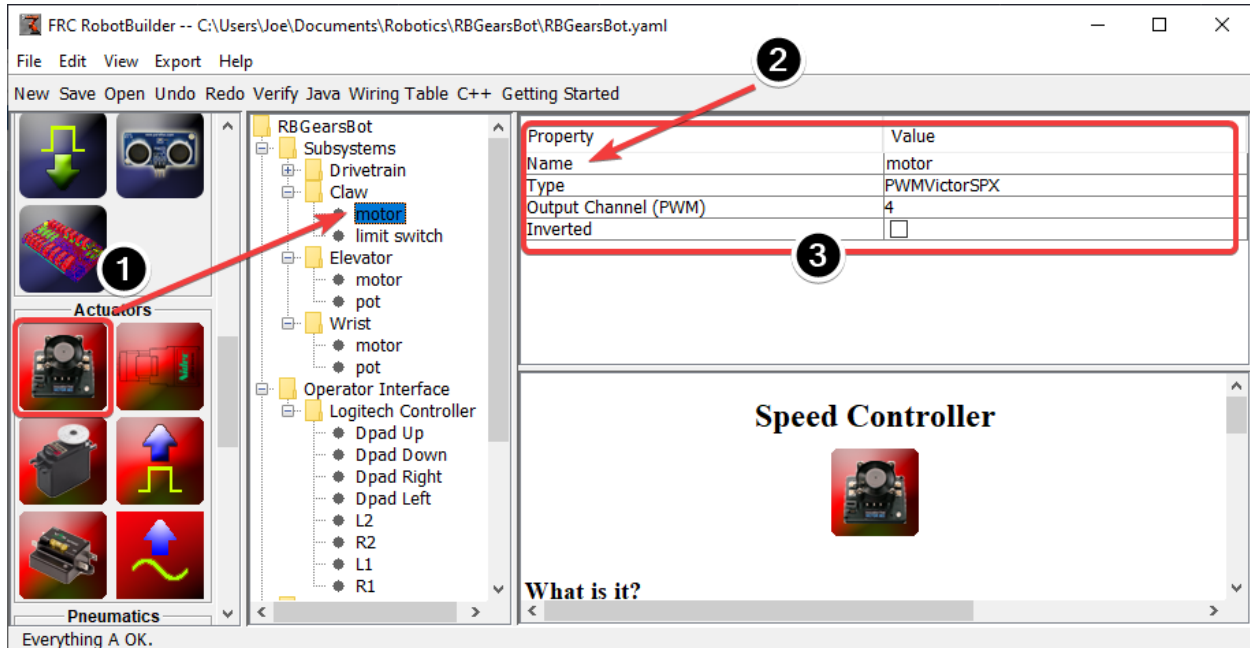
After adding constants and setting their values, just press “Save and close” to save the constants and close the dialog. If you don’t want to save, press the exit button on the top of the window.

After Saving



After saving constants, the names will appear in the “Constants” button in the subsystem properties.

Dragging Actuators/Sensors into the Subsystem



There are three steps to adding components to a subsystem:

1. Drag actuators or sensors from the palette into the subsystem as required.
2. Give the actuator or sensor a meaningful name
3. Edit the properties such as module numbers and channel numbers for each item in the subsystem.

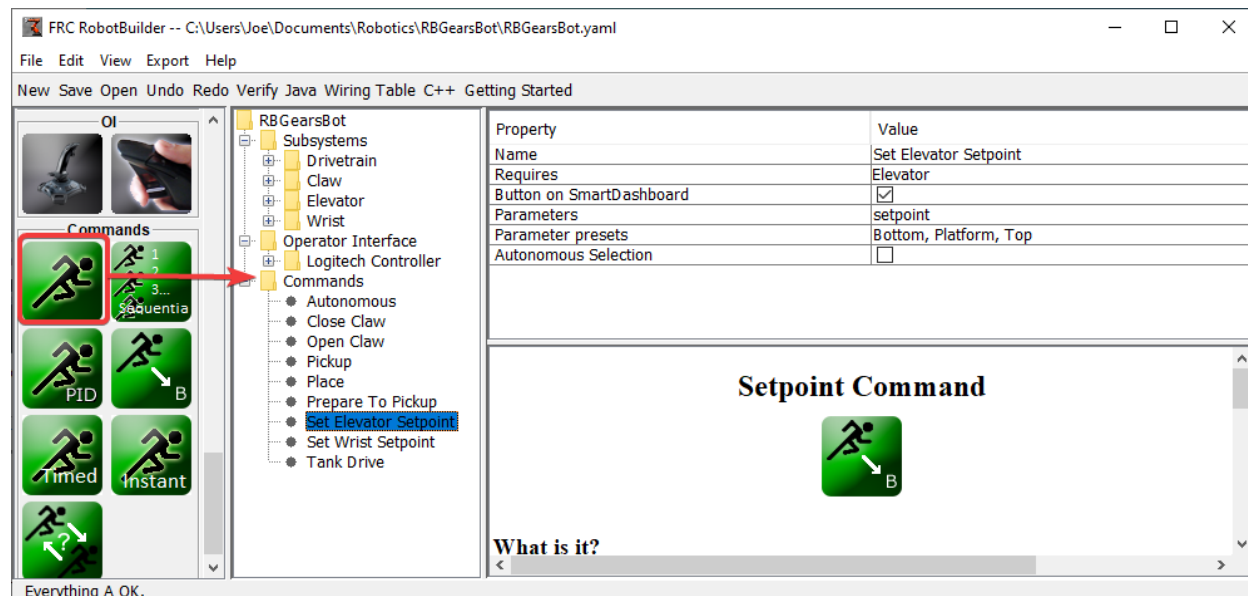
RobotBuilder will automatically use incrementing channel numbers for each module on the robot. If you haven't yet wired the robot you can just let RobotBuilder assign unique channel numbers for each sensor or actuator and wire the robot according to the generating wiring table.

This just creates the subsystem in RobotBuilder, and will subsequently generate skeleton code for the subsystem. To make it actually operate your robot please refer to [Writing Code for a Subsystem](#).

24.1.6 Creating a Command

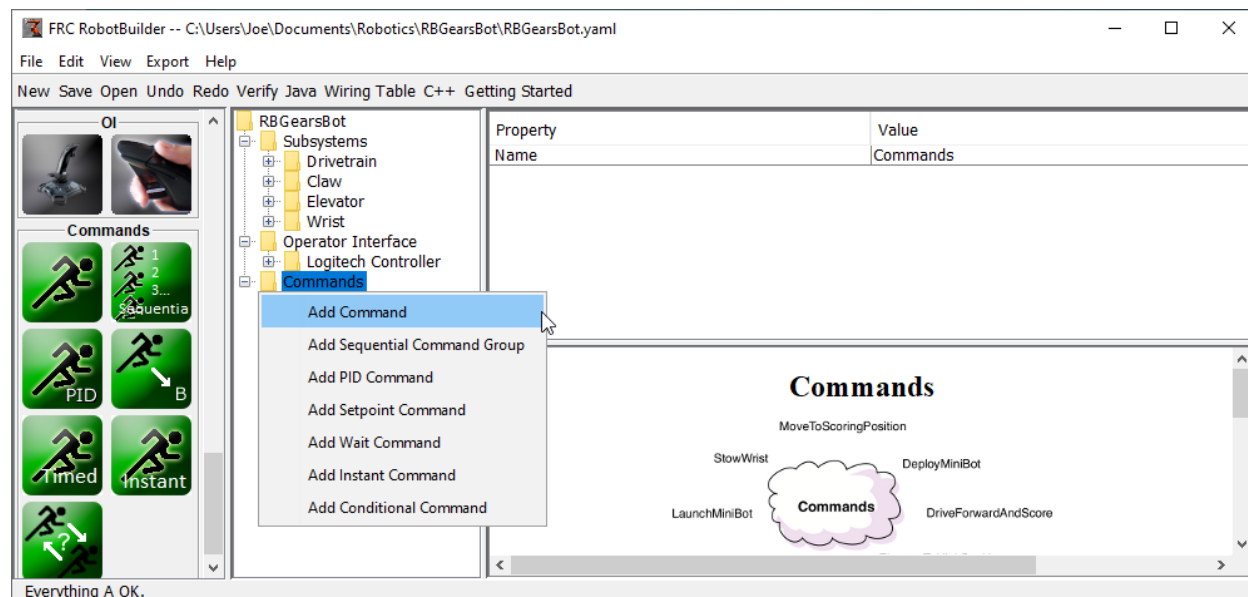
Commands are classes you create that provide behaviors or actions for your subsystems. The subsystem class should set the operation of the subsystem, like `MoveElevator` to start the elevator moving, or `ElevatorToSetPoint` to set the elevator's PID setpoint. The commands initiate the subsystem operation and keep track of when it is finished.

Drag the Command to the Commands Folder



Simple commands can be dragged from the palette to the robot description. The command will be created under the Commands folder.

Creating Commands using the Context Menu



You can also create commands using the right-click context menu on the Command folder in the robot description.

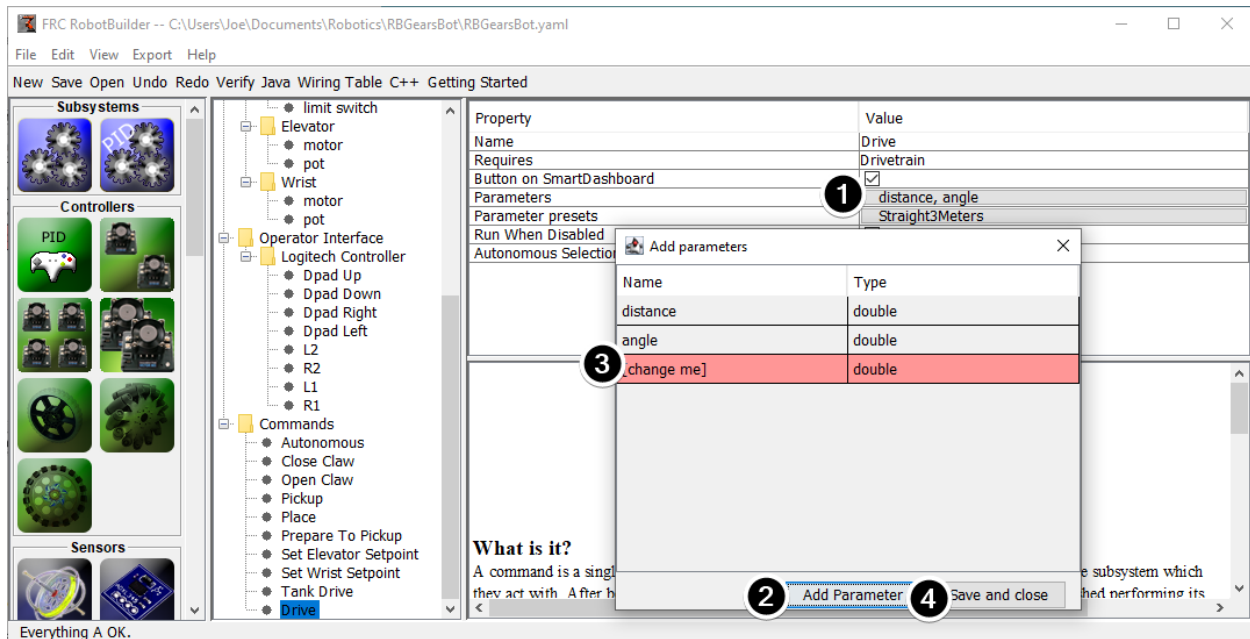
Configuring the Command

Property	Value
Name	Set Elevator Setpoint
Requires	Elevator
Button on SmartDashboard	<input checked="" type="checkbox"/>
Parameters	setpoint
Parameter presets	Bottom, Platform, Top
Autonomous Selection	<input type="checkbox"/>

1. Name the command with something meaningful that describes what the command will do. Commands should be named as if they were in code, although there can be spaces between words.
2. Set the subsystem that is required by this command. When this command is scheduled, it will automatically stop any command currently running that also requires this command. If a command to open the claw is currently running (requiring the claw subsystem) and the close claw command is scheduled, it will immediately stop opening and start closing.
3. Tell RobotBuilder if it should create buttons on the SmartDashboard for the command. A button will be created for each parameter preset.
4. Set the parameters this command takes. A single command with parameters can do the same thing as two or more commands that do not take parameters. For example, “Drive Forward”, “Drive Backward”, and “Drive Distance” commands can be consolidated into a single command that takes values for direction and distance.
5. Set presets for parameters. These can be used elsewhere in RobotBuilder when using the command, such as binding it to a joystick button or setting the default command for a subsystem.
6. *Run When Disabled*. Allows the command to run when the robot is disabled. However, any actuators commanded while disabled will not actuate.
7. *Autonomous Selection*. Whether the command should be added to the Sendable Chooser so that it can be selected for autonomous.

Setpoint commands come with a single parameter (‘setpoint’, of type double); parameters cannot be added, edited, or deleted for setpoint commands.

Adding and Editing Parameters

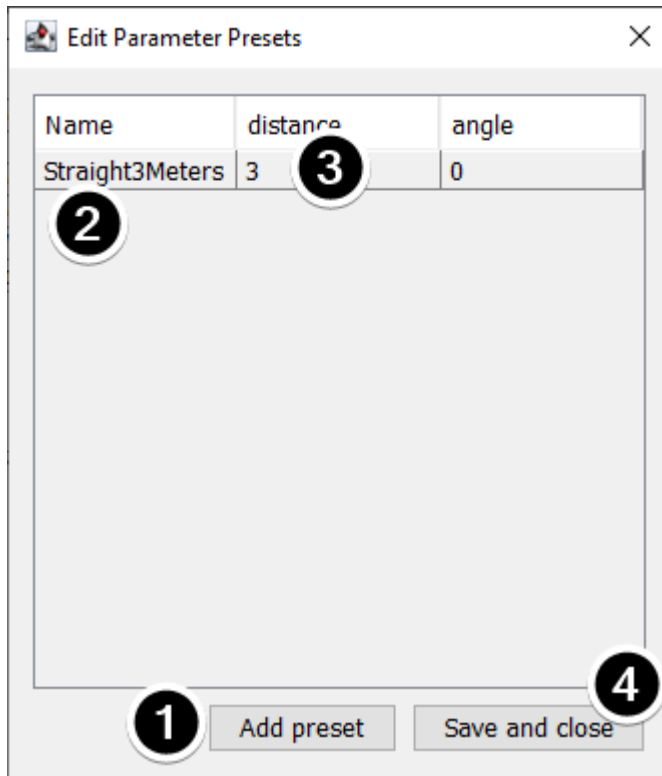


To add or edit parameters:

1. Click the button in the *Value* column of the property table
2. Press the *Add Parameter* button to add a parameter
3. A parameter that has just been added. The name defaults to *[change me]* and the type defaults to String. The default name is invalid, so you will have to change it before exporting. Double click the *Name* cell to start changing the name. Double click the *Type* cell to select the type.
4. Save and close button will save all changes and close the window.

Rows can be reordered simply by dragging, and can be deleted by selecting them and pressing delete or backspace.

Adding and Editing Parameter Presets

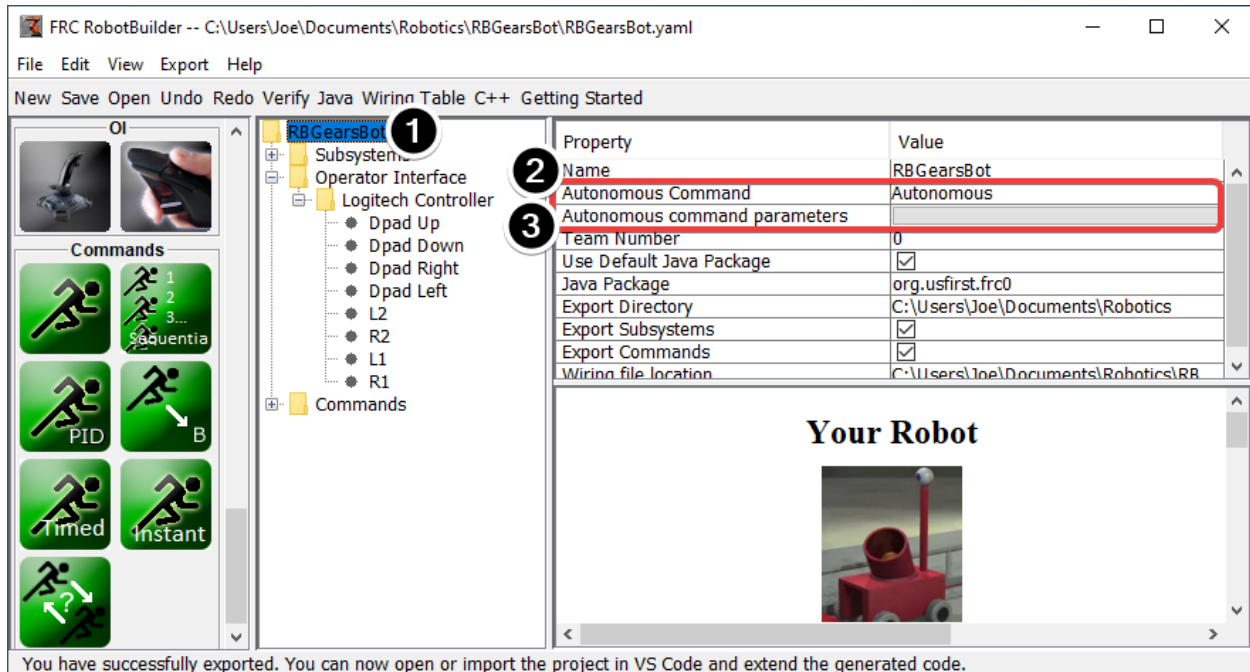


1. Click *Add parameter set* to add a new preset.
2. Change the name of the preset to something descriptive. The presets in this example are for opening and closing the gripper subsystem.
3. Change the value of the parameter(s) for the preset. You can either type a value in (e.g. "3.14") or select from constants defined in the subsystem that the command requires. Note that the type of the constant has to be the same type as the parameter - you can't have an int-type constant be passed to a double-type parameter, for example
4. Click *Save and close* to save changes and exit the dialog; to exit without saving, press the exit button in the top bar of the window.

24.1.7 Setting the Autonomous Commands

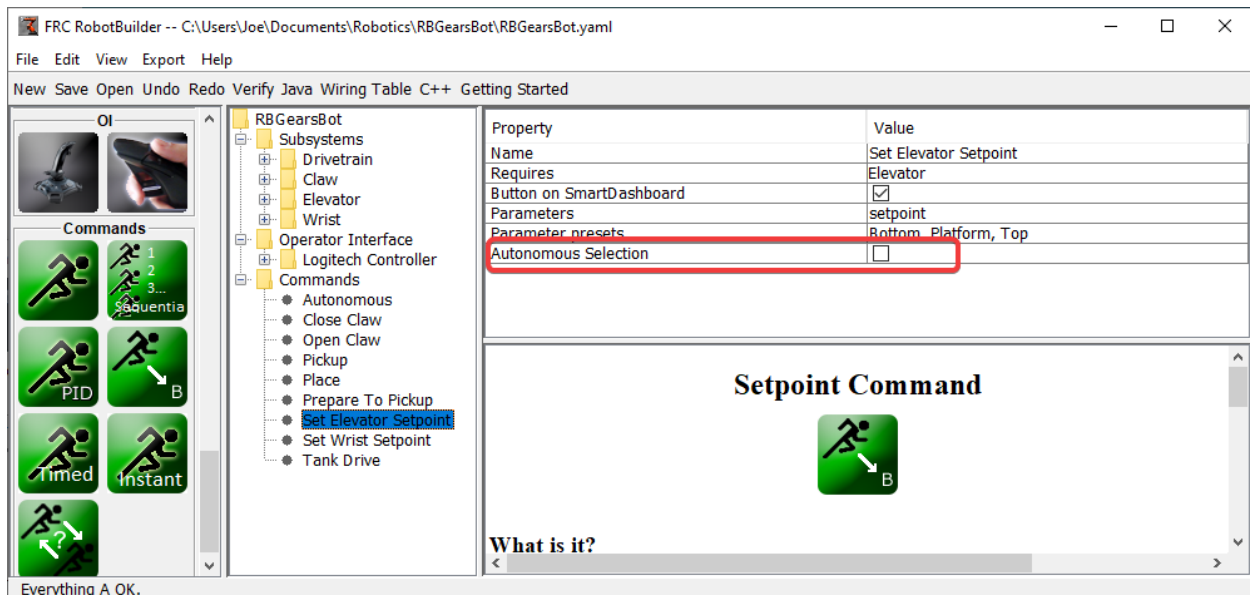
Since a command is simply one or more actions (behaviors) that the robot performs, it makes sense to describe the autonomous operation of a robot as a command. While it could be a single command, it is more likely going to be a command group (a group of commands that happen together).

RobotBuilder generates code for a *Sendable Chooser* which allows the autonomous command to run to be chosen from the dashboard.



To designate the default autonomous command that runs if another command is not selected on the dashboard:

- Select the robot in the robot program description
- Fill in the Autonomous command field with the command that should run when the robot is placed in autonomous mode. This is a drop-down field and will give you the option to select any command that has been defined.
- Set the parameters the command takes, if any.



To select commands to add as options to the Sendable Chooser, select the Autonomous Selection check box.

When the robot is put into autonomous mode, the chosen Autonomous command will be sched-

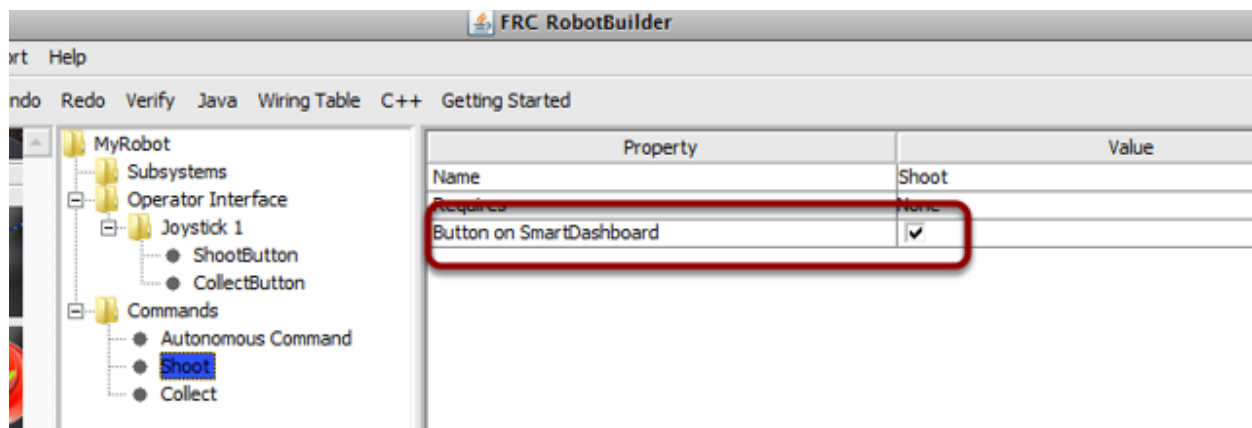
uled.

24.1.8 Using Shuffleboard to Test a Command

Commands are easily tested by adding a button to Shuffleboard/SmartDashboard to trigger the command. In this way, no integration with the rest of the robot program is necessary and commands can easily be independently tested. This is the easiest way to verify commands since with a single line of code in your program, a button can be created on Shuffleboard that will run the command. These buttons can then be left in place to verify subsystems and command operations in the future.

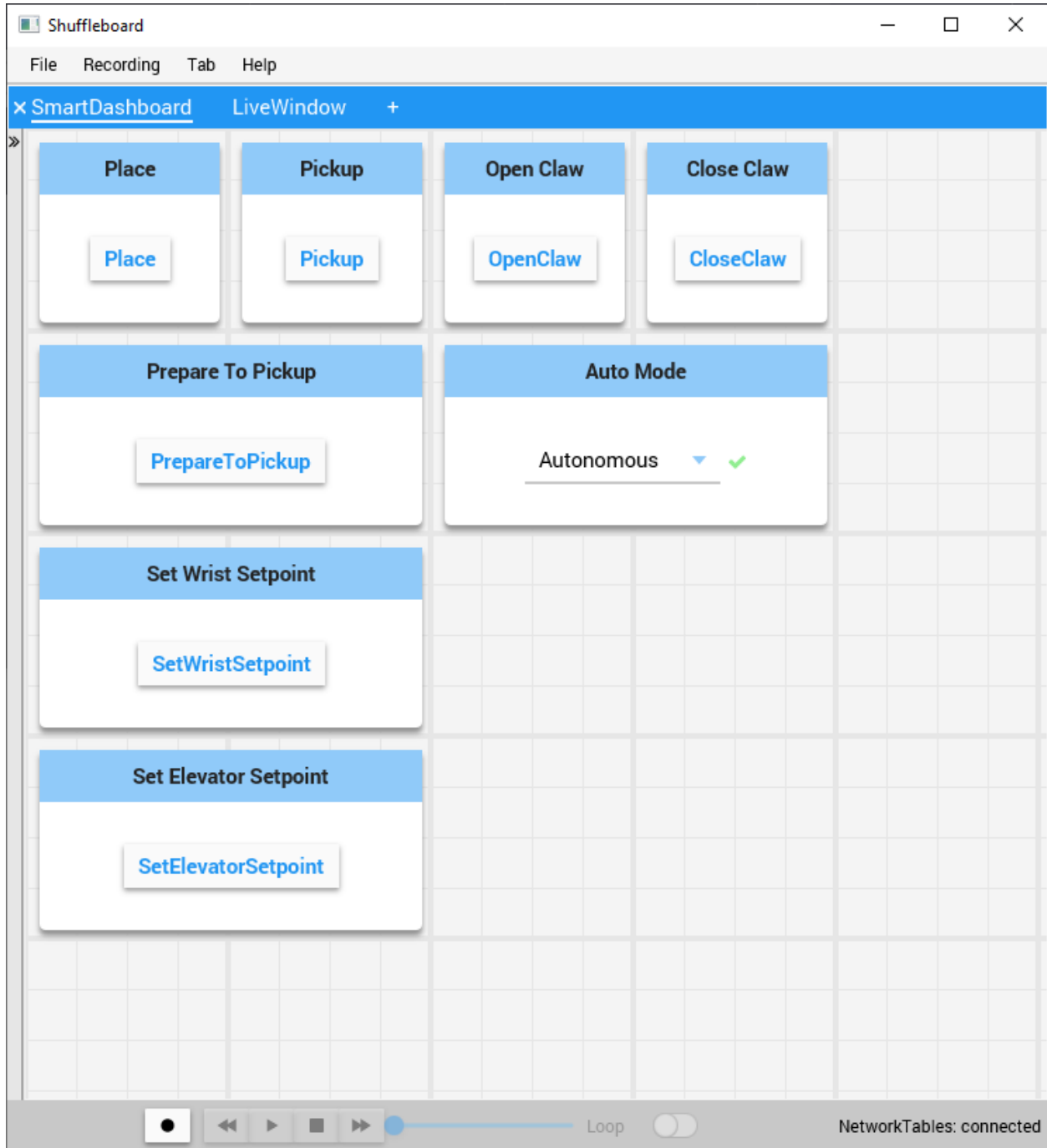
This has the added benefit of accommodating multiple programmers, each writing commands. As the code is checked into the main robot project, the commands can be individually tested.

Creating the Button on Shuffleboard



The button is created on the SmartDashboard by putting an instance of the command from the robot program to the dashboard. This is such a common operation that it has been added to RobotBuilder as a checkbox. When writing your commands, be sure that the box is checked, and buttons will be automatically generated for you.

Operating the Buttons



The buttons will be generated automatically and will appear on the dashboard screen. You can rearrange the buttons on Shuffleboard. In this example there are a number of commands, each with an associated button for testing. Pressing the commands button will run the command. Once it is pressed, pressing again it will interrupt the command causing the `Interrupted()` method to be called.

Adding Commands Manually

Java

C++

```
SmartDashboard.putData("Autonomous Command", new AutonomousCommand());  
SmartDashboard.putData("Open Claw", new OpenClaw(m_claw));  
SmartDashboard.putData("Close Claw", new CloseClaw(m_claw));
```

```
SmartDashboard::PutData("Autonomous Command", new AutonomousCommand());  
SmartDashboard::PutData("Open Claw", new OpenClaw(&m_claw));  
SmartDashboard::PutData("Close Claw", new CloseClaw(&m_claw));
```

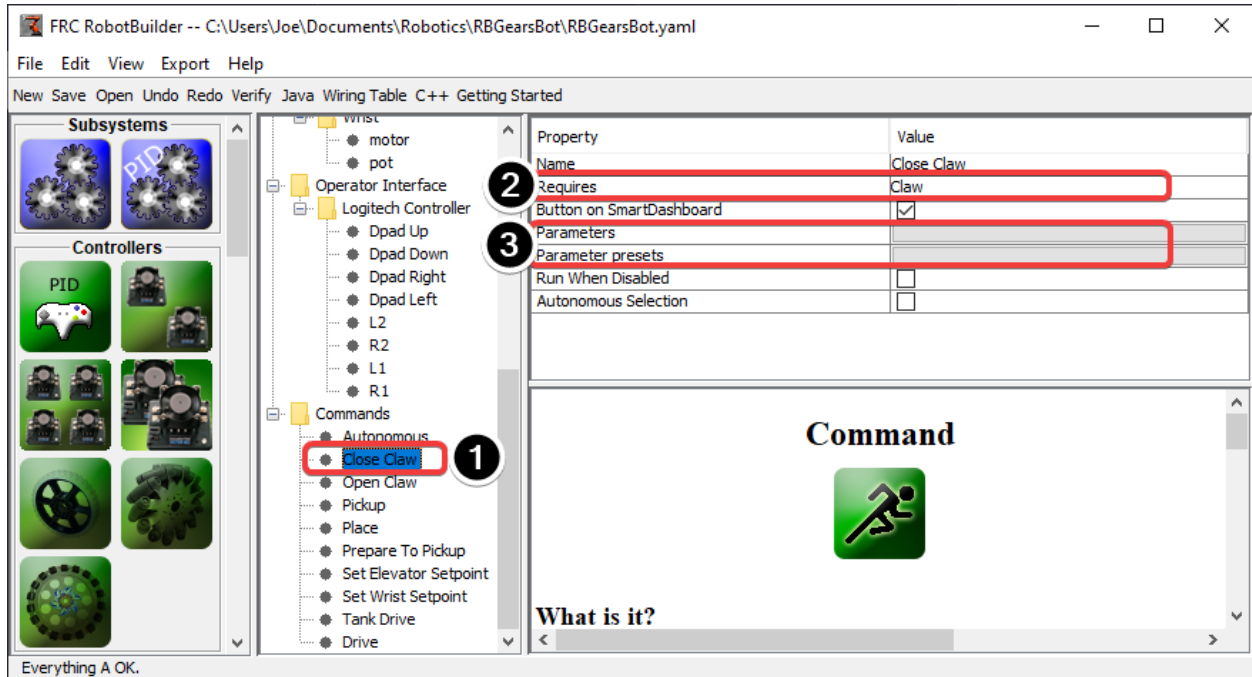
Commands can be added to the Shuffleboard manually by writing the code yourself. This is done by passing instances of the command to the PutData method along with the name that should be associated with the button on the Shuffleboard. These instances are scheduled whenever the button is pressed. The result is exactly the same as RobotBuilder generated code, although clicking the checkbox in RobotBuilder is much easier than writing all the code by hand.

24.1.9 Connecting the Operator Interface to a Command

Commands handle the behaviors for your robot. The command starts a subsystem to some operating mode like raising and elevator and continues running until it reaches some set-point or timeout. The command then handles waiting for the subsystem to finish. That way commands can run in sequence to develop more complex behaviors.

RobotBuilder will also generate code to schedule a command to run whenever a button on your operator interface is pressed. You can also write code to run a command when a particular trigger condition has happened.

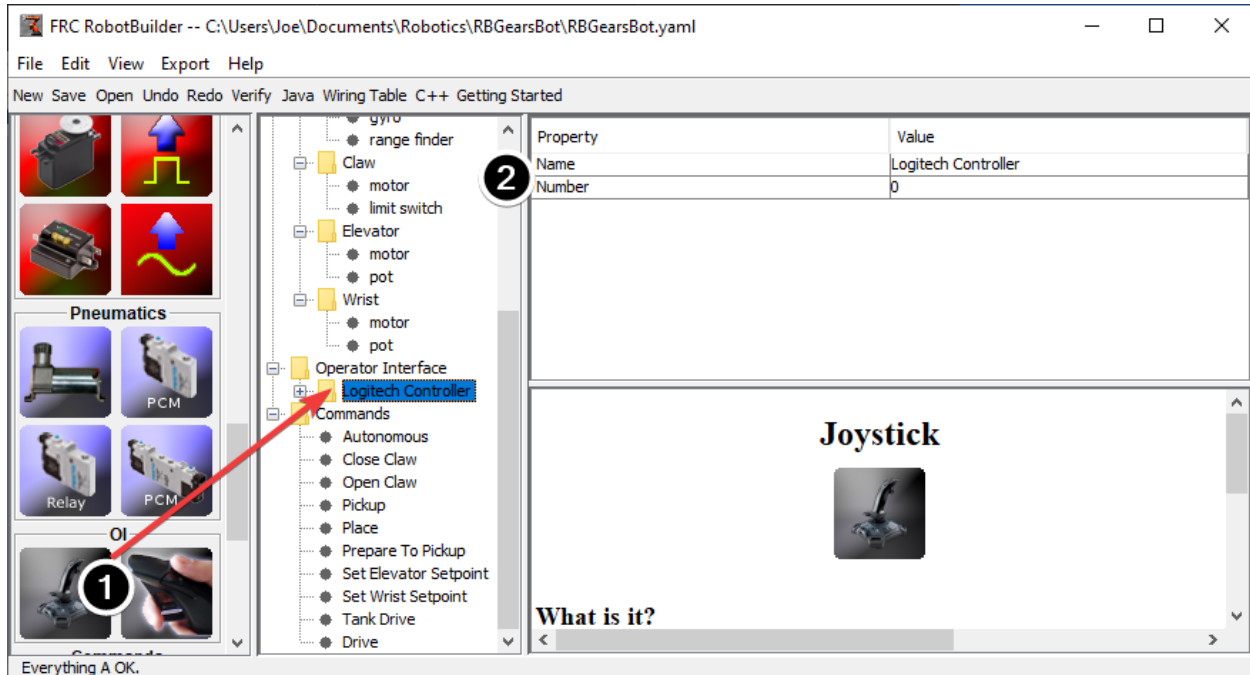
Run a Command with a Button Press



In this example we want to schedule the “Close Claw” command to run whenever the dpad right direction button is pressed on a logitech gamepad (button 6) is pressed.

1. The command to run is called “Close Claw” and its function is to close the claw of the robot
2. Notice that the command requires the Claw subsystem. This will ensure that this command starts running even if there was another operation happening at the same time that used the claw. In this case the previous command would be interrupted.
3. Parameters make it possible for one command to do multiple things; presets let you define values you pass to the command and reuse them

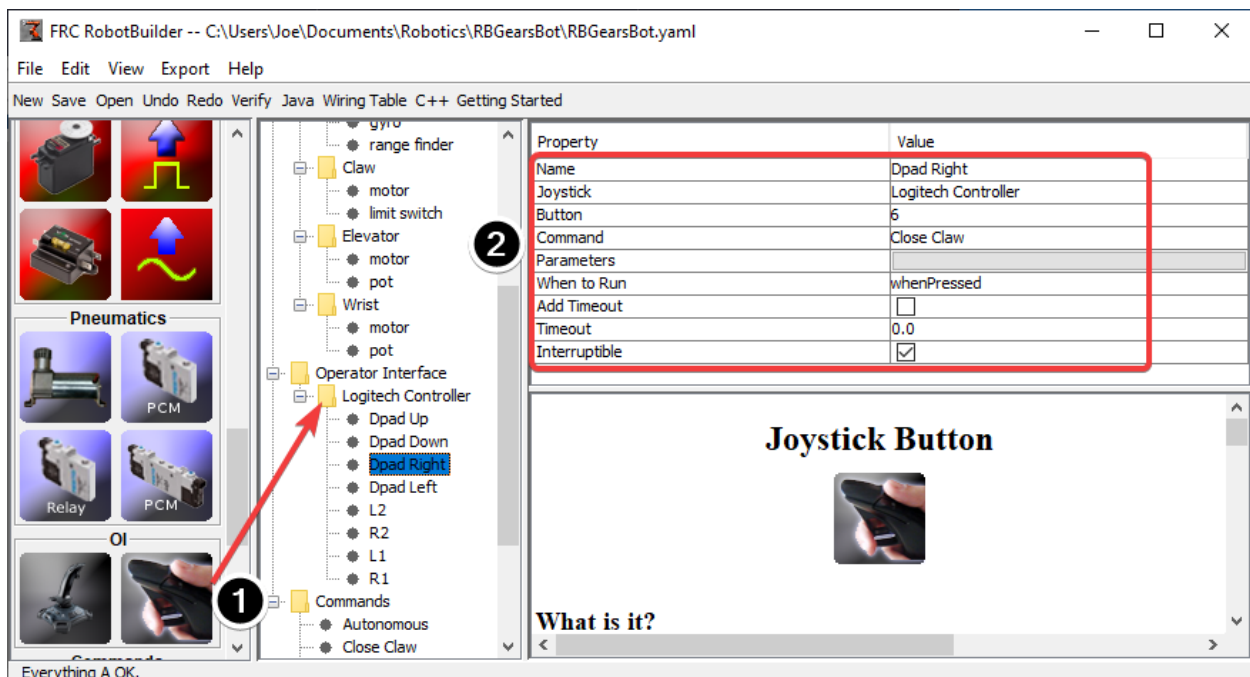
Adding the Joystick to the Robot Program



Add the joystick to the robot program

1. Drag the joystick to the Operator Interface folder in the robot program
2. Name the joystick so that it reflects the use of the joystick and set the USB port number

Linking a Button to the “Move Elevator” Command



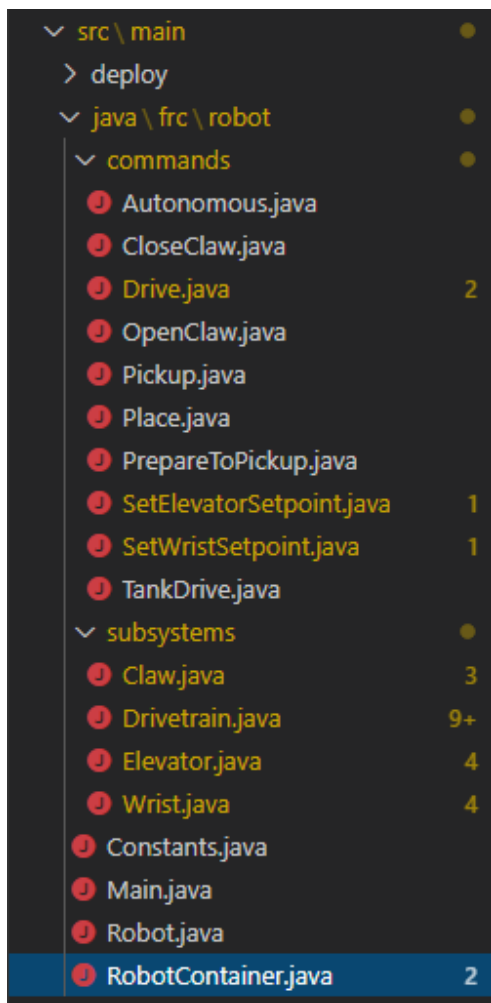
Add the button that should be pressed to the program

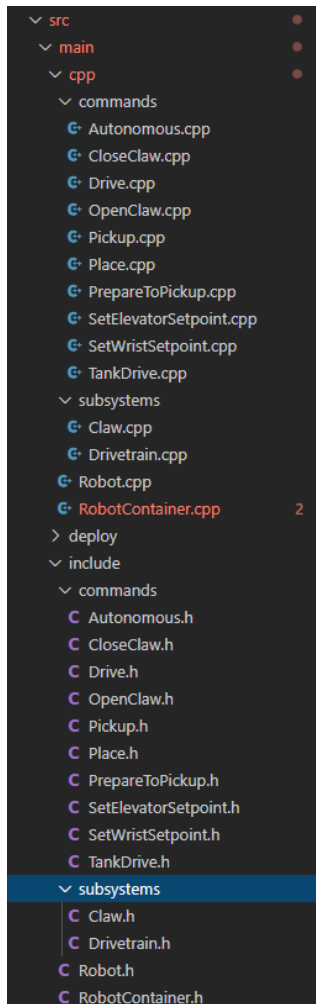
1. Drag the joystick button to the Joystick (Logitech Controller) so that it's under the joystick
2. Set the properties for the button: the button number, the command to run when the button is pressed, parameters the command takes, and the "When to run" property to "whenPressed" to indicate that the command should run whenever the joystick button is pressed.

Note: Joystick buttons must be dragged to (under) a Joystick. You must have a joystick in the Operator Interface folder before adding buttons.

24.1.10 RobotBuilder Created Code

The Layout of a RobotBuilder Generated Project





A RobotBuilder generated project consists of a package (in Java) or a folder (in C++) for Commands and another for Subsystems. Each command or subsystem object is stored under those containers. At the top level of the project you'll find the robot main program (RobotContainer.java/C++).

For more information on the organization of a Command Based robot, see [Structuring a Command-Based Robot Project](#)

Autogenerated Code

Java

C++

```
// BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=AUTONOMOUS
m_chooser.setDefaultOption("Autonomous", new Autonomous());
// END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=AUTONOMOUS

SmartDashboard.putData("Auto Mode", m_chooser);
```

```
// BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=AUTONOMOUS
m_chooser.SetDefaultOption("Autonomous", new Autonomous());
```

(continues on next page)

(continued from previous page)

```
// END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=AUTONOMOUS

frc::SmartDashboard::PutData("Auto Mode", &m_chooser);
```

When the robot description is modified and code is re-exported RobotBuilder is designed to not modify any changes you made to the file, thus preserving your code. This makes RobotBuilder a full-lifecycle tool. To know what code is OK to be modified by RobotBuilder, it generates sections that will potentially have to be rewritten delimited with some special comments. These comments are shown in the example above. Don't add any code within these comment blocks, it will be rewritten next time the project is exported from RobotBuilder.

If code inside one of these blocks must be modified, the comments can be removed, but this will prevent further updates from happening later. In the above example, if the //BEGIN and //END comments were removed, then later another required subsystem was added in RobotBuilder, it would not be generated on that next export.

Main Robot Program

Java

C++ (Header)

C++ (Source)

```
12 package frc.robot;
13
14 import edu.wpi.first.hal.FRCNetComm.tInstances;
15 import edu.wpi.first.hal.FRCNetComm.tResourceType;
16 import edu.wpi.first.hal.HAL;
17 import edu.wpi.first.wpilibj.TimedRobot;
18 import edu.wpi.first.wpilibj2.command.Command;
19 import edu.wpi.first.wpilibj2.command.CommandScheduler;
20
21 /**
22  * The VM is configured to automatically run this class, and to call the
23  * functions corresponding to each mode, as described in the TimedRobot
24  * documentation. If you change the name of this class or the package after
25  * creating this project, you must also update the build.properties file in
26  * the project.
27  */
28 public class Robot extends TimedRobot { // (1)
29
30     private Command m_autonomousCommand;
31
32     private RobotContainer m_robotContainer;
33
34     /**
35      * This function is run when the robot is first started up and should be
36      * used for any initialization code.
37      */
38     @Override
39     public void robotInit() {
40         // Instantiate our RobotContainer. This will perform all our button
41         // bindings, and put our
42         // autonomous chooser on the dashboard.
```

(continues on next page)

(continued from previous page)

```

42     m_robotContainer = new RobotContainer();
43     HAL.report(tResourceType.kResourceType_Framework, tInstances.kFramework_
↳ RobotBuilder);
44 }
45
46 /**
47  * This function is called every robot packet, no matter the mode. Use this for
↳ items like
48  * diagnostics that you want ran during disabled, autonomous, teleoperated and
↳ test.
49  *
50  * <p>This runs after the mode specific periodic functions, but before
51  * LiveWindow and SmartDashboard integrated updating.
52  */
53 @Override
54 public void robotPeriodic() {
55     // Runs the Scheduler. This is responsible for polling buttons, adding
↳ newly-scheduled
56     // commands, running already-scheduled commands, removing finished or
↳ interrupted commands,
57     // and running subsystem periodic() methods. This must be called from the
↳ robot's periodic
58     // block in order for anything in the Command-based framework to work.
59     CommandScheduler.getInstance().run(); // (2)
60 }
61
62
63 /**
64  * This function is called once each time the robot enters Disabled mode.
65  */
66 @Override
67 public void disabledInit() {
68 }
69
70 @Override
71 public void disabledPeriodic() {
72 }
73
74 /**
75  * This autonomous runs the autonomous command selected by your {@link
↳ RobotContainer} class.
76  */
77 @Override
78 public void autonomousInit() {
79     m_autonomousCommand = m_robotContainer.getAutonomousCommand(); // (3)
80
81     // schedule the autonomous command (example)
82     if (m_autonomousCommand != null) {
83         m_autonomousCommand.schedule();
84     }
85 }
86
87 /**
88  * This function is called periodically during autonomous.
89  */
90 @Override

```

(continues on next page)

(continued from previous page)

```

91     public void autonomousPeriodic() {
92     }
93
94     @Override
95     public void teleopInit() {
96         // This makes sure that the autonomous stops running when
97         // teleop starts running. If you want the autonomous to
98         // continue until interrupted by another command, remove
99         // this line or comment it out.
100        if (m_autonomousCommand != null) {
101            m_autonomousCommand.cancel();
102        }
103    }
104
105    /**
106     * This function is called periodically during operator control.
107     */
108    @Override
109    public void teleopPeriodic() {
110    }
111
112    @Override
113    public void testInit() {
114        // Cancels all running commands at the start of test mode.
115        CommandScheduler.getInstance().cancelAll();
116    }
117
118    /**
119     * This function is called periodically during test mode.
120     */
121    @Override
122    public void testPeriodic() {
123    }
124
125    }

```

```

12    #pragma once
13
14    #include <frc/TimedRobot.h>
15    #include <frc2/command/Command.h>
16
17    #include "RobotContainer.h"
18
19    class Robot : public frc::TimedRobot { // (1)
20    public:
21        void RobotInit() override;
22        void RobotPeriodic() override;
23        void DisabledInit() override;
24        void DisabledPeriodic() override;
25        void AutonomousInit() override;
26        void AutonomousPeriodic() override;
27        void TeleopInit() override;
28        void TeleopPeriodic() override;
29        void TestPeriodic() override;
30
31    private:

```

(continues on next page)

(continued from previous page)

```

32 // Have it null by default so that if testing teleop it
33 // doesn't have undefined behavior and potentially crash.
34 frc2::Command* m_autonomousCommand = nullptr;
35
36 RobotContainer m_container;
37 };

```

```

12 #include "Robot.h"
13
14 #include <frc/smartdashboard/SmartDashboard.h>
15 #include <frc2/command/CommandScheduler.h>
16
17 void Robot::RobotInit() {}
18
19 /**
20  * This function is called every robot packet, no matter the mode. Use
21  * this for items like diagnostics that you want to run during disabled,
22  * autonomous, teleoperated and test.
23  *
24  * <p> This runs after the mode specific periodic functions, but before
25  * LiveWindow and SmartDashboard integrated updating.
26  */
27 void Robot::RobotPeriodic() { frc2::CommandScheduler::GetInstance().Run(); } // (2)
28
29 /**
30  * This function is called once each time the robot enters Disabled mode. You
31  * can use it to reset any subsystem information you want to clear when the
32  * robot is disabled.
33  */
34 void Robot::DisabledInit() {}
35
36 void Robot::DisabledPeriodic() {}
37
38 /**
39  * This autonomous runs the autonomous command selected by your {@link
40  * RobotContainer} class.
41  */
42 void Robot::AutonomousInit() {
43     m_autonomousCommand = m_container.GetAutonomousCommand(); // (3)
44
45     if (m_autonomousCommand != nullptr) {
46         m_autonomousCommand->Schedule();
47     }
48 }
49
50 void Robot::AutonomousPeriodic() {}
51
52 void Robot::TeleopInit() {
53     // This makes sure that the autonomous stops running when
54     // teleop starts running. If you want the autonomous to
55     // continue until interrupted by another command, remove
56     // this line or comment it out.
57     if (m_autonomousCommand != nullptr) {
58         m_autonomousCommand->Cancel();
59         m_autonomousCommand = nullptr;
60     }

```

(continues on next page)

(continued from previous page)

```

61 }
62
63 /**
64  * This function is called periodically during operator control.
65  */
66 void Robot::TeleopPeriodic() {}
67
68 /**
69  * This function is called periodically during test mode.
70  */
71 void Robot::TestPeriodic() {}
72
73 #ifndef RUNNING_FRC_TESTS
74 int main() { return frc::StartRobot<Robot>(); }
75 #endif

```

This is the main program generated by RobotBuilder. There are a number of parts to this program (highlighted sections):

1. This class extends TimedRobot. TimedRobot will call your autonomousPeriodic() and teleopPeriodic() methods every 20ms.
2. In the robotPeriodic method which is called every 20ms, make one scheduling pass.
3. The autonomous command provided is scheduled at the start of autonomous in the autonomousInit() method and canceled at the end of the autonomous period in teleopInit().

RobotContainer

Java

C++ (Header)

C++ (Source)

```

12 package frc.robot;
13
14 import frc.robot.commands.*;
15 import frc.robot.subsystems.*;
16 import edu.wpi.first.wpilibj.smartdashboard.SendableChooser;
17 import edu.wpi.first.wpilibj.smartdashboard.SmartDashboard;
18
19 // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=IMPORTS
20 import edu.wpi.first.wpilibj2.command.Command;
21 import edu.wpi.first.wpilibj2.command.InstantCommand;
22 import edu.wpi.first.wpilibj.Joystick;
23 import edu.wpi.first.wpilibj2.command.button.JoystickButton;
24 import frc.robot.subsystems.*;
25
26 // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=IMPORTS
27
28
29 /**
30  * This class is where the bulk of the robot should be declared. Since Command-based
  ↪ is a

```

(continues on next page)

(continued from previous page)

```

31  * "declarative" paradigm, very little robot logic should actually be handled in the
    ↳{@link Robot}
32  * periodic methods (other than the scheduler calls). Instead, the structure of the
    ↳robot
33  * (including subsystems, commands, and button mappings) should be declared here.
34  */
35  public class RobotContainer {
36
37      private static RobotContainer m_robotContainer = new RobotContainer();
38
39      // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
40  // The robot's subsystems
41      private final Wrist m_wrist = new Wrist(); // (1)
42      private final Elevator m_elevator = new Elevator();
43      private final Claw m_claw = new Claw();
44      private final Drivetrain m_drivetrain = new Drivetrain();
45
46  // Joysticks
47  private final Joystick logitechController = new Joystick(0); // (3)
48
49      // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
50
51
52  // A chooser for autonomous commands
53  SendableChooser<Command> m_chooser = new SendableChooser<>();
54
55  /**
56   * The container for the robot. Contains subsystems, OI devices, and commands.
57   */
58  private RobotContainer() {
59      // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=SMARTDASHBOARD
60  // Smartdashboard Subsystems
61      SmartDashboard.putData(m_wrist);
62      SmartDashboard.putData(m_elevator);
63      SmartDashboard.putData(m_claw);
64      SmartDashboard.putData(m_drivetrain);
65
66
67  // SmartDashboard Buttons
68      SmartDashboard.putData("Close Claw", new CloseClaw( m_claw )); // (6)
69      SmartDashboard.putData("Open Claw", new OpenClaw( m_claw ));
70      SmartDashboard.putData("Pickup", new Pickup());
71      SmartDashboard.putData("Place", new Place());
72      SmartDashboard.putData("Prepare To Pickup", new PrepareToPickup());
73      SmartDashboard.putData("Set Elevator Setpoint: Bottom", new SetElevatorSetpoint(0,
    ↳ m_elevator));
74      SmartDashboard.putData("Set Elevator Setpoint: Platform", new
    ↳ SetElevatorSetpoint(0.2, m_elevator));
75      SmartDashboard.putData("Set Elevator Setpoint: Top", new SetElevatorSetpoint(0.3,
    ↳ m_elevator));
76      SmartDashboard.putData("Set Wrist Setpoint: Horizontal", new SetWristSetpoint(0,
    ↳ m_wrist));
77      SmartDashboard.putData("Set Wrist Setpoint: Raise Wrist", new SetWristSetpoint(-
    ↳ 45, m_wrist));
78      SmartDashboard.putData("Drive: Straight3Meters", new Drive(3, 0, m_drivetrain));
79      SmartDashboard.putData("Drive: Place", new Drive(Drivetrain.PlaceDistance,
    ↳ Drivetrain.BackAwayDistance, m_drivetrain));

```

(continues on next page)

(continued from previous page)

```

80
81 // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=SMARTDASHBOARD
82 // Configure the button bindings
83 configureButtonBindings();
84
85 // Configure default commands
86 // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=SUBSYSTEM_DEFAULT_COMMAND
87 m_drivetrain.setDefaultCommand(new TankDrive( m_drivetrain ) ); // (5)
88
89
90 // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=SUBSYSTEM_DEFAULT_COMMAND
91
92 // Configure autonomous sendable chooser
93 // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=AUTONOMOUS
94
95 m_chooser.setDefaultOption("Autonomous", new Autonomous()); // (2)
96
97 // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=AUTONOMOUS
98
99 SmartDashboard.putData("Auto Mode", m_chooser);
100 }
101
102 public static RobotContainer getInstance() {
103     return m_robotContainer;
104 }
105
106 /**
107  * Use this method to define your button->command mappings. Buttons can be created
108  * by
109  * instantiating a {@link GenericHID} or one of its subclasses ({@link
110  * edu.wpi.first.wpilibj.Joystick} or {@link XboxController}), and then passing it
111  * to a
112  * {@link edu.wpi.first.wpilibj2.command.button.JoystickButton}.
113  */
114 private void configureButtonBindings() {
115     // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=BUTTONS
116     // Create some buttons
117     final JoystickButton dpadUp = new JoystickButton(logitechController, 5); // (4)
118     dpadUp.whenPressed(new SetElevatorSetpoint(0.3, m_elevator), true);
119     SmartDashboard.putData("Dpad Up", new SetElevatorSetpoint(0.3, m_elevator) );
120
121     final JoystickButton dpadDown = new JoystickButton(logitechController, 7);
122     dpadDown.whenPressed(new SetElevatorSetpoint(0, m_elevator), true);
123     SmartDashboard.putData("Dpad Down", new SetElevatorSetpoint(0, m_elevator) );
124
125     final JoystickButton dpadRight = new JoystickButton(logitechController, 6);
126     dpadRight.whenPressed(new CloseClaw( m_claw ), true);
127     SmartDashboard.putData("Dpad Right", new CloseClaw( m_claw ) );
128
129     final JoystickButton dpadLeft = new JoystickButton(logitechController, 8);
130     dpadLeft.whenPressed(new OpenClaw( m_claw ), true);
131     SmartDashboard.putData("Dpad Left", new OpenClaw( m_claw ) );
132
133     final JoystickButton l2 = new JoystickButton(logitechController, 9);
134     l2.whenPressed(new PrepareToPickup(), true);
135     SmartDashboard.putData("L2", new PrepareToPickup() );

```

(continues on next page)

(continued from previous page)

```

134
135 final JoystickButton r2 = new JoystickButton(logitechController, 10);
136 r2.whenPressed(new Pickup() ,true);
137     SmartDashboard.putData("R2",new Pickup() );
138
139 final JoystickButton l1 = new JoystickButton(logitechController, 11);
140 l1.whenPressed(new Place() ,true);
141     SmartDashboard.putData("L1",new Place() );
142
143 final JoystickButton r1 = new JoystickButton(logitechController, 12);
144 r1.whenPressed(new Autonomous() ,true);
145     SmartDashboard.putData("R1",new Autonomous() );
146
147
148     // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=BUTTONS
149 }
150
151     // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=FUNCTIONS
152 public Joystick getLogitechController() {
153     return logitechController;
154 }
155
156
157     // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=FUNCTIONS
158
159 /**
160  * Use this to pass the autonomous command to the main {@link Robot} class.
161  *
162  * @return the command to run in autonomous
163  */
164 public Command getAutonomousCommand() {
165     // The selected command will be run in autonomous
166     return m_chooser.getSelected();
167 }
168
169
170
171 }

```

```

12 #pragma once
13
14 // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=INCLUDES
15 #include <frc/smartdashboard/SendableChooser.h>
16 #include <frc2/command/Command.h>
17
18 #include "subsystems/Wrist.h"
19 #include "subsystems/Elevator.h"
20 #include "subsystems/Claw.h"
21 #include "subsystems/Drivetrain.h"
22
23
24 #include "subsystems/Claw.h"
25 #include "subsystems/Drivetrain.h"
26 #include "subsystems/Elevator.h"
27 #include "subsystems/Wrist.h"
28

```

(continues on next page)

(continued from previous page)

```

29 #include "commands/Autonomous.h"
30 #include "commands/CloseClaw.h"
31 #include "commands/Drive.h"
32 #include "commands/OpenClaw.h"
33 #include "commands/Pickup.h"
34 #include "commands/Place.h"
35 #include "commands/PrepareToPickup.h"
36 #include "commands/SetElevatorSetpoint.h"
37 #include "commands/SetWristSetpoint.h"
38 #include "commands/TankDrive.h"
39 #include <frc/Joystick.h>
40 #include <frc2/command/button/JoystickButton.h>
41
42 // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=INCLUDES
43
44 class RobotContainer {
45
46 public:
47
48     frc2::Command* GetAutonomousCommand();
49     static RobotContainer* GetInstance();
50
51     // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=PROTOTYPES
52
53     frc::Joystick* getLogitechController();
54
55     // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=PROTOTYPES
56
57 private:
58
59     RobotContainer();
60
61     // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
62     // The robot's subsystems
63     Drivetrain m_drivetrain; // (1)
64     Claw m_claw;
65     Elevator m_elevator;
66     Wrist m_wrist;
67
68     // Joysticks
69     frc::Joystick m_logitechController{0}; // (3)
70
71     frc::SendableChooser<frc2::Command*> m_chooser;
72
73     // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
74
75     Autonomous m_autonomousCommand;
76     static RobotContainer* m_robotContainer;
77
78     void ConfigureButtonBindings();
79 };

```

```

12 #include "RobotContainer.h"
13 #include <frc2/command/ParallelRaceGroup.h>
14 #include <frc/smartdashboard/SmartDashboard.h>
15

```

(continues on next page)

(continued from previous page)

```

16
17
18 RobotContainer* RobotContainer::m_robotContainer = NULL;
19
20 RobotContainer::RobotContainer() : m_autonomousCommand(
21     // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTOR
22 ){
23
24
25
26     // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTOR
27
28     // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=SMARTDASHBOARD
29     // Smartdashboard Subsystems
30     frc::SmartDashboard::PutData(&m_drivetrain);
31     frc::SmartDashboard::PutData(&m_claw);
32     frc::SmartDashboard::PutData(&m_elevator);
33     frc::SmartDashboard::PutData(&m_wrist);
34
35
36     // SmartDashboard Buttons
37     frc::SmartDashboard::PutData("Drive: Straight3Meters", new Drive(3, 0, &m_
38     drivetrain)); // (6)
39     frc::SmartDashboard::PutData("Drive: Place", new Drive(Drivetrain::PlaceDistance,
40     Drivetrain::BackAwayDistance, &m_drivetrain));
41     frc::SmartDashboard::PutData("Set Wrist Setpoint: Horizontal", new
42     SetWristSetpoint(0, &m_wrist));
43     frc::SmartDashboard::PutData("Set Wrist Setpoint: Raise Wrist", new
44     SetWristSetpoint(-45, &m_wrist));
45     frc::SmartDashboard::PutData("Set Elevator Setpoint: Bottom", new
46     SetElevatorSetpoint(0, &m_elevator));
47     frc::SmartDashboard::PutData("Set Elevator Setpoint: Platform", new
48     SetElevatorSetpoint(0.2, &m_elevator));
49     frc::SmartDashboard::PutData("Set Elevator Setpoint: Top", new
50     SetElevatorSetpoint(0.3, &m_elevator));
51     frc::SmartDashboard::PutData("Prepare To Pickup", new PrepareToPickup());
52     frc::SmartDashboard::PutData("Place", new Place());
53     frc::SmartDashboard::PutData("Pickup", new Pickup());
54     frc::SmartDashboard::PutData("Open Claw", new OpenClaw( &m_claw ));
55     frc::SmartDashboard::PutData("Close Claw", new CloseClaw( &m_claw ));
56
57     // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=SMARTDASHBOARD
58
59     ConfigureButtonBindings();
60
61     // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DEFAULT-COMMANDS
62     m_drivetrain.SetDefaultCommand(TankDrive( &m_drivetrain )); // (5)
63
64     // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DEFAULT-COMMANDS
65
66     // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=AUTONOMOUS
67
68     m_chooser.SetDefaultOption("Autonomous", new Autonomous()); // (2)
69
70     // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=AUTONOMOUS

```

(continues on next page)

(continued from previous page)

```

65
66     frc::SmartDashboard::PutData("Auto Mode", &m_chooser);
67
68 }
69
70 RobotContainer* RobotContainer::GetInstance() {
71     if (m_robotContainer == NULL) {
72         m_robotContainer = new RobotContainer();
73     }
74     return(m_robotContainer);
75 }
76
77 void RobotContainer::ConfigureButtonBindings() {
78     // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=BUTTONS
79
80     frc2::JoystickButton m_r1{&m_logitechController, 12}; // (4)
81     frc2::JoystickButton m_l1{&m_logitechController, 11};
82     frc2::JoystickButton m_r2{&m_logitechController, 10};
83     frc2::JoystickButton m_l2{&m_logitechController, 9};
84     frc2::JoystickButton m_dpadLeft{&m_logitechController, 8};
85     frc2::JoystickButton m_dpadRight{&m_logitechController, 6};
86     frc2::JoystickButton m_dpadDown{&m_logitechController, 7};
87     frc2::JoystickButton m_dpadUp{&m_logitechController, 5};
88
89     m_r1.WhenPressed(Autonomous(), true);
90     m_l1.WhenPressed(Place(), true);
91     m_r2.WhenPressed(Pickup(), true);
92     m_l2.WhenPressed(PrepareToPickup(), true);
93     m_dpadLeft.WhenPressed(OpenClaw( &m_claw ), true);
94     m_dpadRight.WhenPressed(CloseClaw( &m_claw ), true);
95     m_dpadDown.WhenPressed(SetElevatorSetpoint(0, &m_elevator), true);
96     m_dpadUp.WhenPressed(SetElevatorSetpoint(0.3, &m_elevator), true);
97
98     // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=BUTTONS
99 }
100
101 // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=FUNCTIONS
102
103 frc::Joystick* RobotContainer::getLogitechController() {
104     return &m_logitechController;
105 }
106
107 // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=FUNCTIONS
108
109
110 frc2::Command* RobotContainer::GetAutonomousCommand() {
111     // The selected command will be run in autonomous
112     return m_chooser.GetSelected();
113 }

```

This is the RobotContainer generated by RobotBuilder which is where the subsystems and operator interface are defined. There are a number of parts to this program (highlighted sections):

1. Each of the subsystems is declared here. They can be passed as parameters to any commands that require them.
2. If there is an autonomous command provided in RobotBuilder robot properties, it is

added to the Sendable Chooser to be selected on the dashboard.

3. The code for all the operator interface components is generated here.
4. In addition the code to link the OI buttons to commands that should run is also generated here.
5. Commands to be run on a subsystem when no other commands are running are defined here.
6. Commands to be run via a dashboard are defined here.

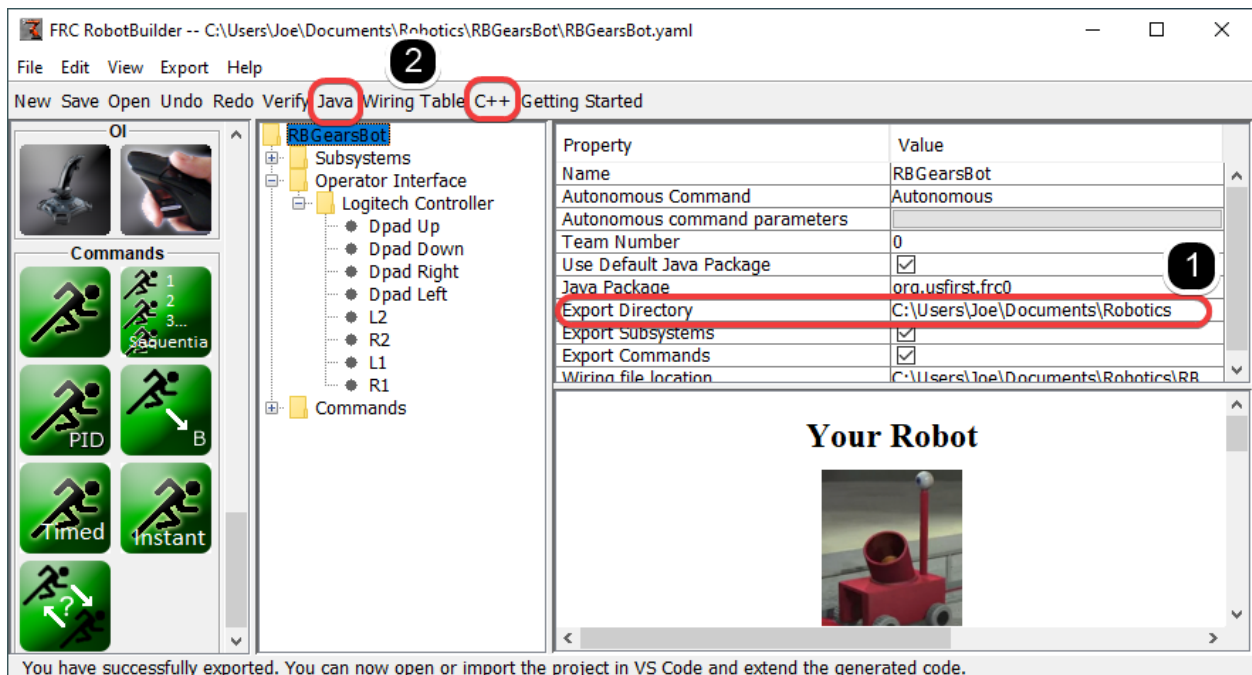
24.2 RobotBuilder - Writing the Code

Important: RobotBuilder has been updated to support the new commandbased framework! Unfortunately, this documentation is outdated and only for the old commandbased framework. Individuals interested in updating this documentation can open a pull request on the [frc-docs](#) repository.

24.2.1 Generating Code for a Project

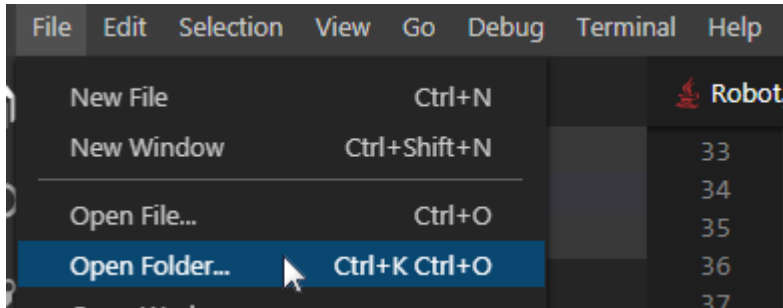
After you've set up your robot framework in RobotBuilder, you'll need to export the code and load it into Visual Studio Code. This article describes the process for doing so.

Generate the Code for the Project



Verify that the Export Directory points to where you want (1) and then click Java or C++ (2) to generate a VS Code project or update code.

Open the Project in Visual Studio Code

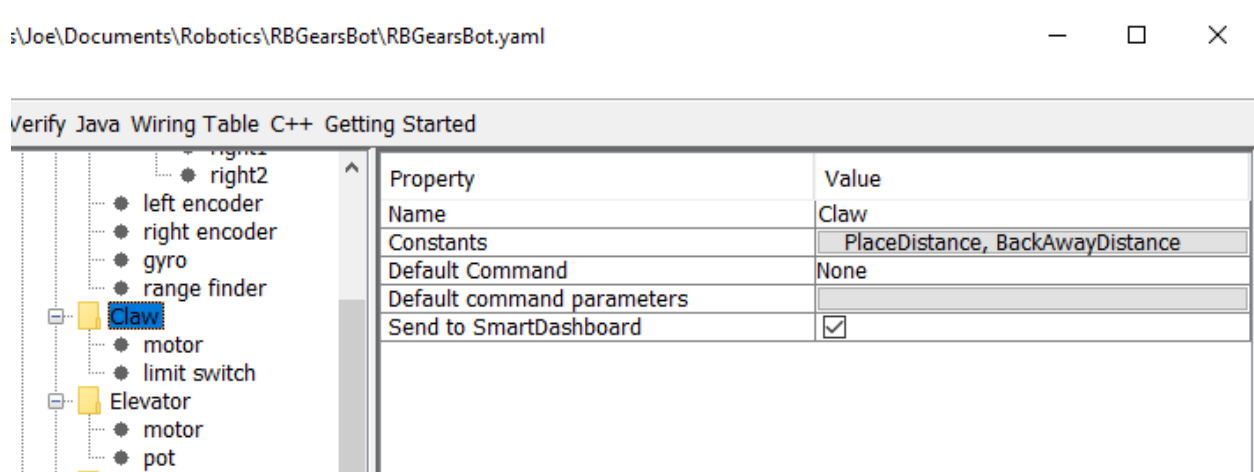


Open VS Code and select **File -> Open Folder**. Navigate to your Export location and click **Select Folder**.

24.2.2 Writing the Code for a Subsystem

Adding code to create an actual working subsystem is very straightforward. For simple subsystems that don't use feedback it turns out to be extremely simple. In this section we will look at an example of a *Claw* subsystem. The *Claw* subsystem also has a limit switch to determine if an object is in the grip.

RobotBuilder Representation of the Claw Subsystem



The claw at the end of a robot arm is a subsystem operated by a single VictorSPX Motor Controller. There are three things we want the motor to do, start opening, start closing, and stop moving. This is the responsibility of the subsystem. The timing for opening and closing will be handled by a command later in this tutorial. We will also define a method to get if the claw is gripping an object.

Adding Subsystem Capabilities

Java

C++

```

12 public class Claw extends SubsystemBase {
13     // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTANTS
14     public static final double PlaceDistance = 0.1;
15     public static final double BackAwayDistance = 0.6;
16
17     // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTANTS
18
19     // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
20     private PWMVictorSPX motor;
21     private DigitalInput limitswitch;
22
23     // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
24
25     public Claw() {
26         // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTORS
27         motor = new PWMVictorSPX(4);
28         addChild("motor",motor);
29         motor.setInverted(false);
30
31         limitswitch = new DigitalInput(4);
32         addChild("limit switch",limitswitch);
33
34
35
36         // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTORS
37     }
38
39     @Override
40     public void periodic() {
41         // This method will be called once per scheduler run
42     }
43
44
45     @Override
46     public void simulationPeriodic() {
47         // This method will be called once per scheduler run when in simulation
48     }
49
50
51     public void open() {
52         motor.set(1.0);
53     }
54
55     public void close() {
56         motor.set(-1.0);
57     }
58
59     public void stop() {
60         motor.set(0.0);
61     }
62
63     public boolean isGripping() {

```

(continues on next page)

(continued from previous page)

```

64     return limitswitch.get();
65 }
66
67 }

```

```

12 Claw::Claw(){
13 SetName("Claw");
14     // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
15
16 AddChild("motor", &m_motor);
17 m_motor.SetInverted(false);
18
19 AddChild("limit switch", &m_limitswitch);
20
21
22     // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
23 }
24
25 void Claw::Periodic() {
26     // Put code here to be run every loop
27 }
28
29
30 void Claw::SimulationPeriodic() {
31     // This method will be called once per scheduler run when in simulation
32 }
33
34
35 // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CMDPIDGETTERS
36
37 // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CMDPIDGETTERS
38
39 void Claw::Open() {
40     m_motor.Set(1.0);
41 }
42
43 void Claw::Close() {
44     m_motor.Set(-1.0);
45 }
46
47 void Claw::Stop() {
48     m_motor.Set(0.0);
49 }
50
51 bool Claw::IsGripping() {
52     return m_limitswitch.Get();
53 }

```

Add methods to the `claw.java` or `claw.cpp` that will open, close, and stop the claw from moving and get the claw limit switch. Those will be used by commands that actually operate the claw.

Note: The comments have been removed from this file to make it easier to see the changes for this document.

Notice that member variable called `motor` and `limitswitch` are created by RobotBuilder so it can be used throughout the subsystem. Each of your dragged-in palette items will have a member variable with the name given in RobotBuilder.

Adding the Method Declarations to the Header File (C++ Only)

C++

```

12 class Claw: public frc2::SubsystemBase {
13 private:
14     // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
15     frc::PWMVictorSPX m_motor{4};
16     frc::DigitalInput m_limitswitch{4};
17
18     // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
19 public:
20     Claw();
21
22     void Periodic() override;
23     void SimulationPeriodic() override;
24     void Open();
25     void Close();
26     void Stop();
27     bool IsGripping();
28     // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CMDPIDGETTERS
29
30     // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CMDPIDGETTERS
31     // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTANTS
32     static constexpr const double PlaceDistance = 0.1;
33     static constexpr const double BackAwayDistance = 0.6;
34
35     // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTANTS
36
37
38 };

```

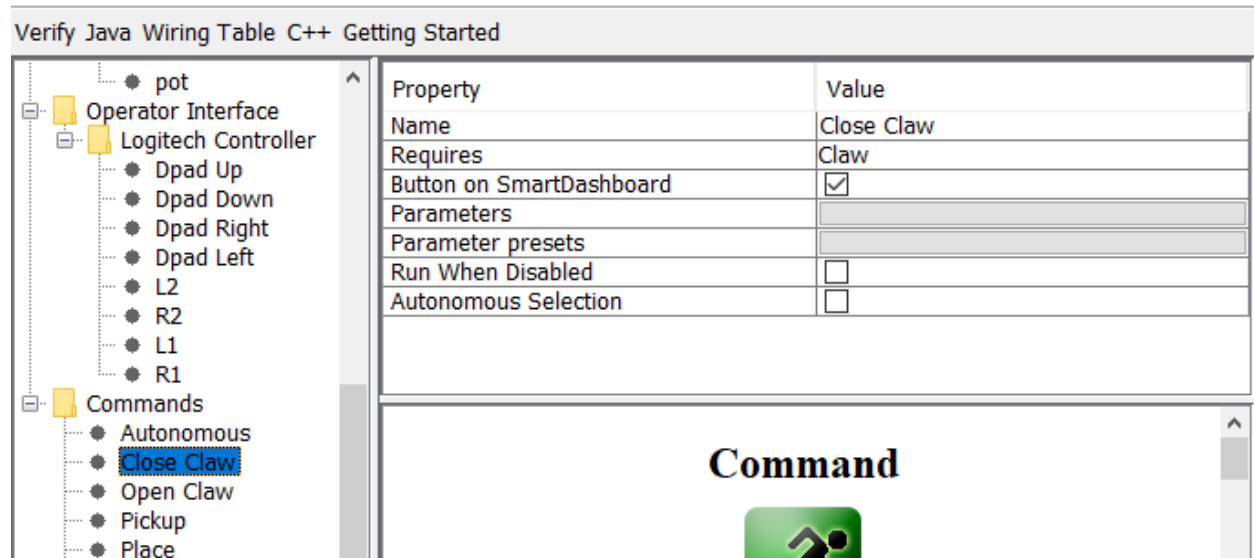
In addition to adding the methods to the class implementation file, `Claw.cpp`, the declarations for the methods need to be added to the header file, `Claw.h`. Those declarations that must be added are shown here.

To add the behavior to the claw subsystem to handle opening and closing you need to *define commands*.

24.2.3 Writing the Code for a Command

Subsystem classes get the mechanisms on your robot moving, but to get it to stop at the right time and sequence through more complex operations you write Commands. Previously in *writing the code for a subsystem* we developed the code for the *Claw* subsystem on a robot to start the claw opening, closing, or to stop moving. Now we will write the code for a command that will actually run the claw motor for the right time to get the claw to open and close. Our claw example is a very simple mechanism where we run the motor for 1 second to open it or until the limit switch is tripped to close it.

Close Claw Command in RobotBuilder



This is the definition of the *CloseClaw* command in RobotBuilder. Notice that it requires the *Claw* subsystem. This is explained in the next step.

Generated CloseClaw Class

Java

C++

```

12 public class CloseClaw extends CommandBase {
13
14     // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=VARIABLE_DECLARATIONS
15     private final Claw m_claw;
16
17     // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=VARIABLE_DECLARATIONS
18
19     // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTORS
20
21
22     public CloseClaw(Claw subsystem) {
23
24
25         // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTORS
26         // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=VARIABLE_SETTING
27
28         // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=VARIABLE_SETTING
29         // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=REQUIRES
30
31         m_claw = subsystem;
32         addRequirements(m_claw);
33
34         // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=REQUIRES
35     }
36

```

(continues on next page)

(continued from previous page)

```

37 // Called when the command is initially scheduled.
38 @Override
39 public void initialize() {
40     m_claw.open(); // (1)
41 }
42
43 // Called every time the scheduler runs while the command is scheduled.
44 @Override
45 public void execute() {
46 }
47
48 // Called once the command ends or is interrupted.
49 @Override
50 public void end(boolean interrupted) {
51     m_claw.stop(); // (3)
52 }
53
54 // Returns true when the command should end.
55 @Override
56 public boolean isFinished() {
57     return m_claw.isGripping(); // (2)
58 }
59
60 @Override
61 public boolean runsWhenDisabled() {
62     // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DISABLED
63     return false;
64
65     // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DISABLED
66 }
67 }

```

```

12 #include "commands/CloseClaw.h"
13
14 CloseClaw::CloseClaw(Claw* m_claw)
15 :m_claw(m_claw){
16
17     // Use AddRequirements() here to declare subsystem dependencies
18     // eg. AddRequirements(Robot::chassis.get());
19     SetName("CloseClaw");
20     AddRequirements(m_claw);
21
22     // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTOR
23 }
24
25
26 // Called just before this Command runs the first time
27 void CloseClaw::Initialize() {
28     m_claw->Close(); // (1)
29 }
30
31
32 // Called repeatedly when this Command is scheduled to run
33 void CloseClaw::Execute() {
34
35 }

```

(continues on next page)

(continued from previous page)

```

36
37 // Make this return true when this Command no longer needs to run execute()
38 bool CloseClaw::IsFinished() {
39     return m_claw->IsGripping(); // (2)
40 }
41
42 // Called once after isFinished returns true
43 void CloseClaw::End(bool interrupted) {
44     m_claw->Stop(); // (3)
45 }
46
47 bool CloseClaw::RunsWhenDisabled() const {
48     // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DISABLED
49     return false;
50
51     // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DISABLED
52 }

```

RobotBuilder will generate the class files for the *CloseClaw* command. The command represents the behavior of the claw, that is the operation over time. To operate this very simple claw mechanism the motor needs to operate in the close direction,. The *Claw* subsystem has methods to start the motor running in the right direction and to stop it. The commands responsibility is to run the motor for the correct time. The lines of code that are shown in the boxes are added to add this behavior.

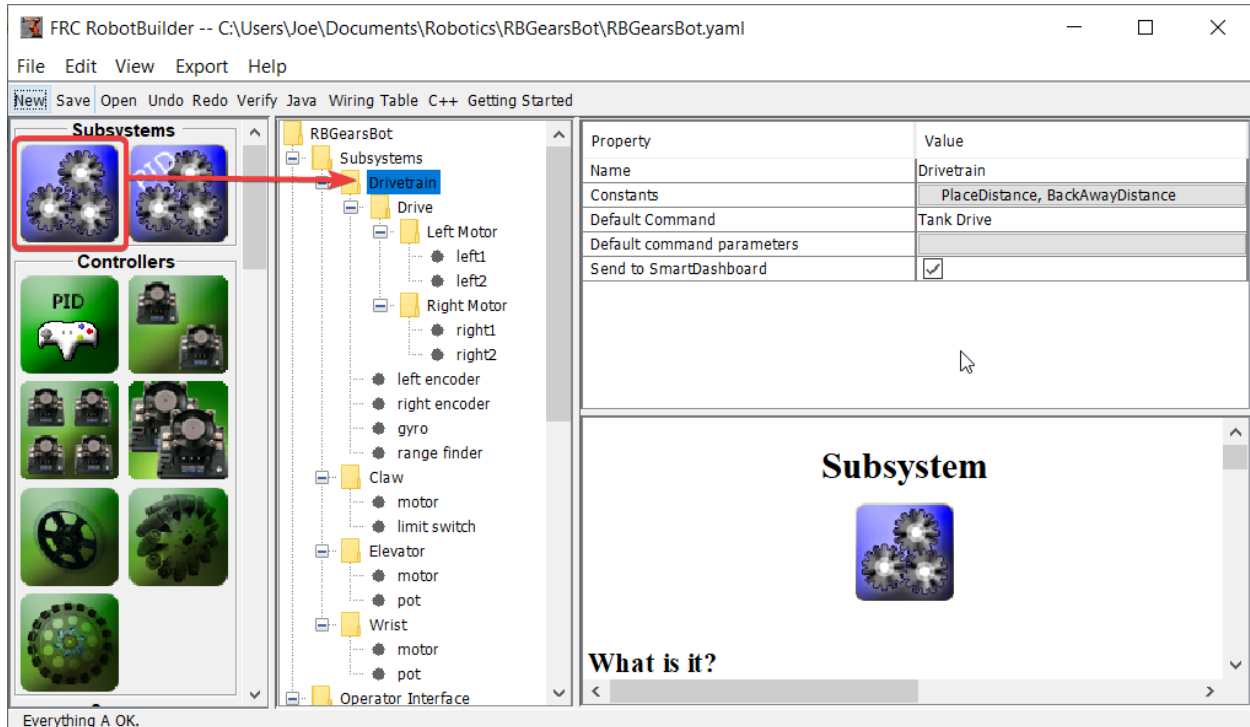
1. Start the claw motor moving in the closing direction by calling the `Close()` method that was added to the *Claw* subsystem in the *CloseClaw* Initialize method.
2. This command is finished when the the limit switch in the *Claw* subsystem is tripped.
3. The `End()` method is called when the command is finished and is a place to clean up. In this case, the motor is stopped since the time has run out.

24.2.4 Driving the Robot with Tank Drive and Joysticks

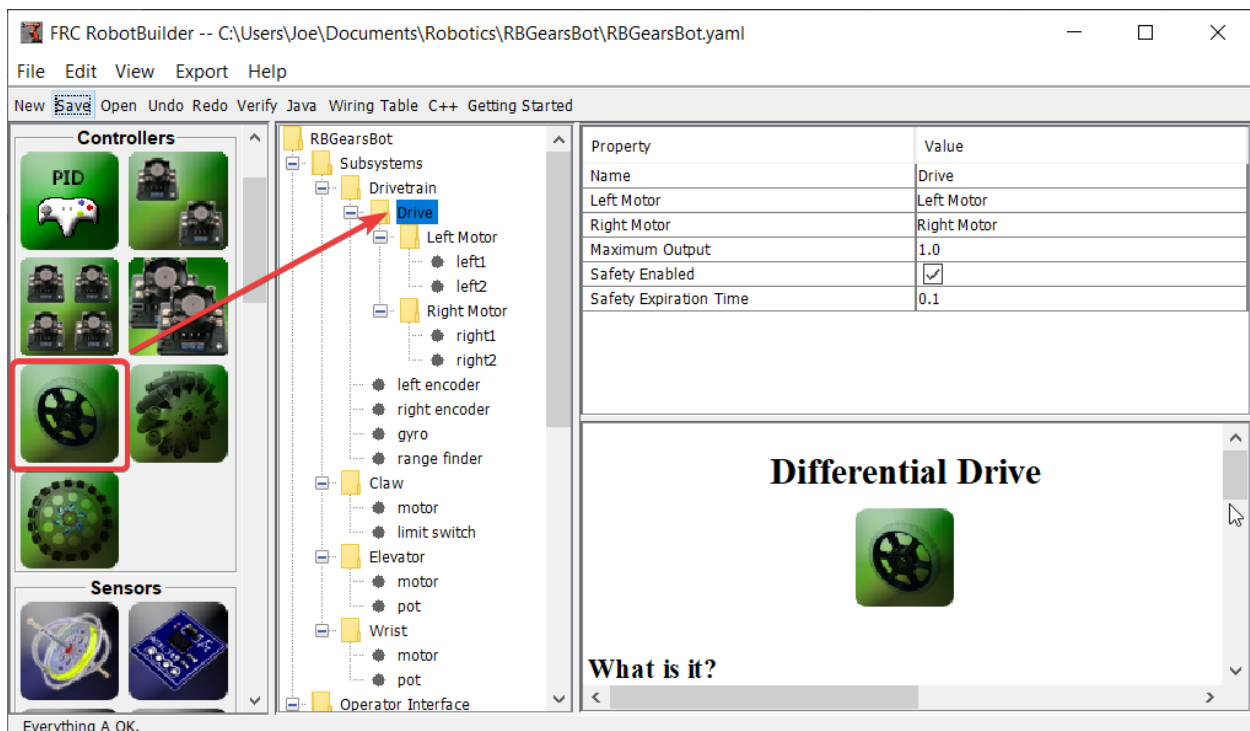
A common use case is to have a joystick that should drive some actuators that are part of a subsystem. The problem is that the joystick is created in the *RobotContainer* class and the motors to be controlled are in the subsystem. The idea is to create a command that, when scheduled, reads input from the joystick and calls a method that is created on the subsystem that drives the motors.

In this example a drive base subsystem is shown that is operated in tank drive using a pair of joysticks.

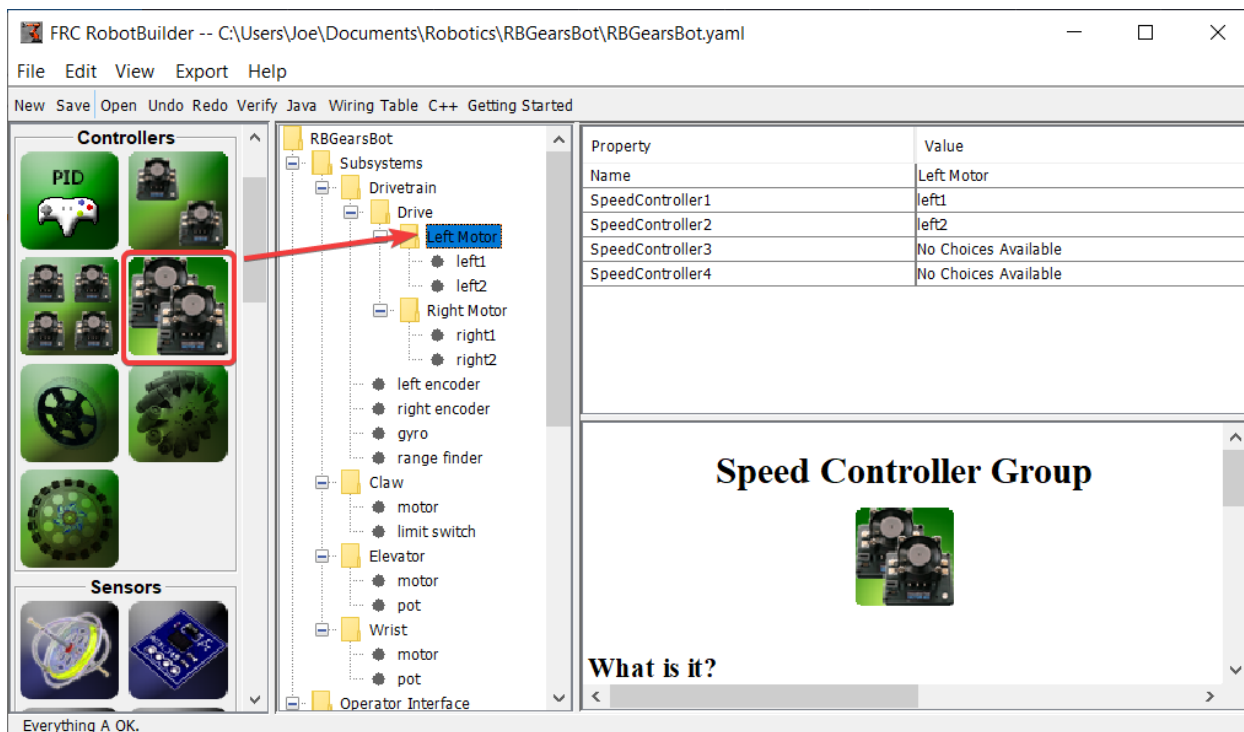
Create a Drive Train Subsystem



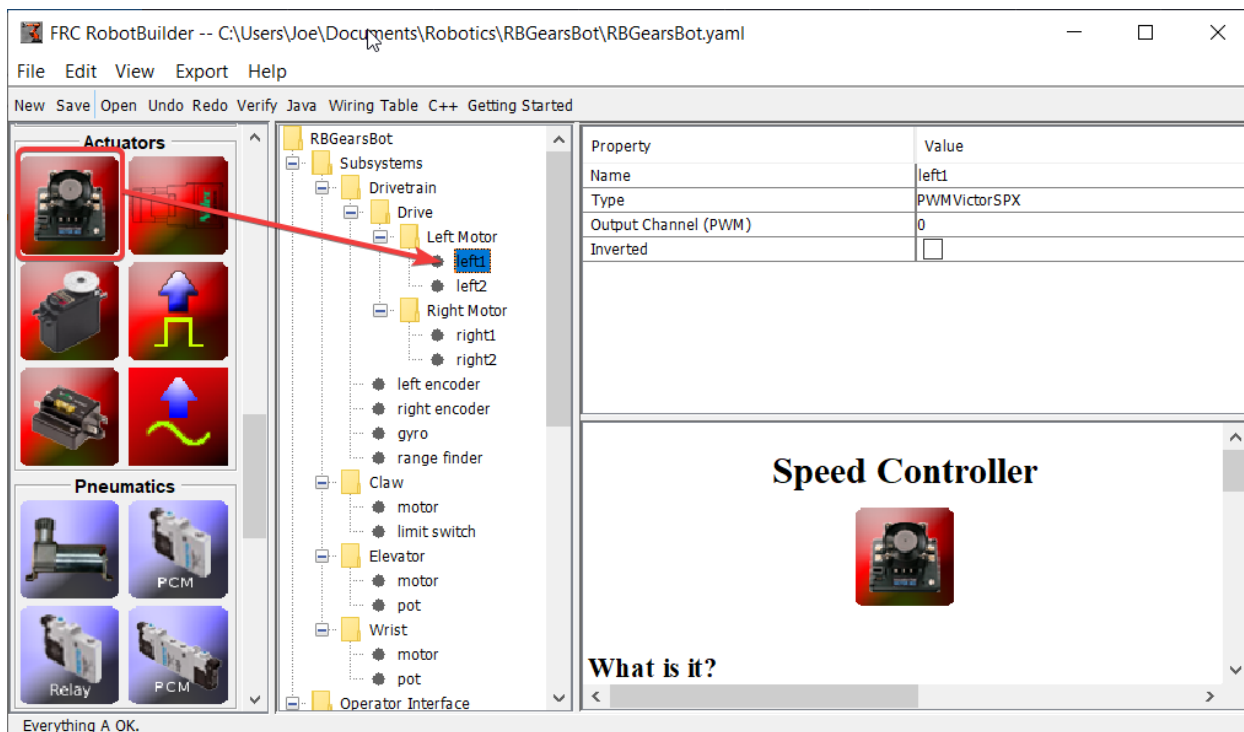
Create a subsystem called Drive Train. Its responsibility will be to handle the driving for the robot base.



Inside the Drive Train create a Differential Drive object for a two motor drive. There is a left motor and right motor as part of the Differential Drive class.

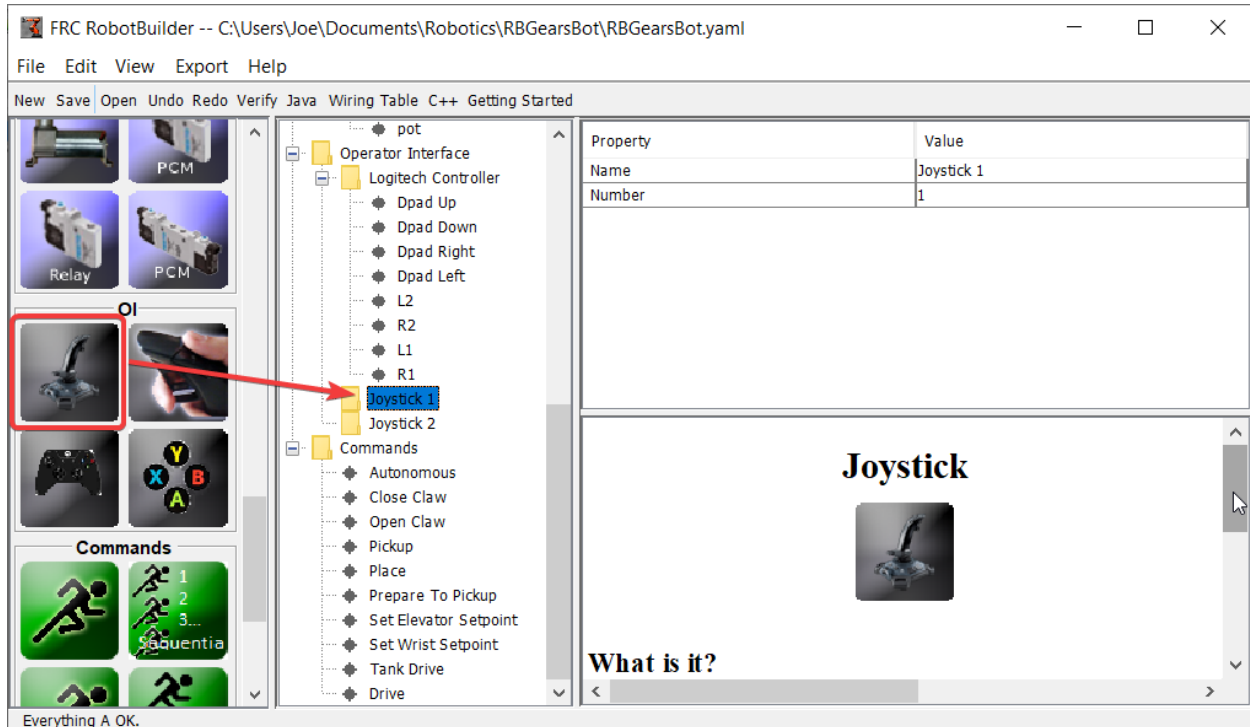


Since we want to use more than two motors to drive the robot, inside the Differential Drive, create two Speed Controller Groups. These will group multiple speed controllers so they can be used with Differential Drive.



Finally, create two Speed Controllers in each Speed Controller Group.

Add the Joysticks to the Operator Interface



Add two joysticks to the Operator Interface, one is the left stick and the other is the right stick. The y-axis on the two joysticks are used to drive the robots left and right sides.

Note: Be sure to export your program to C++ or Java before continuing to the next step.

Create a Method to Write the Motors on the Subsystem

java

C++ (Header)

C++ (Source)

```
public class Drivetrain extends SubsystemBase {

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
private PWMVictorSPX left1;
private PWMVictorSPX left2;
private SpeedControllerGroup leftMotor;
private PWMVictorSPX right1;
private PWMVictorSPX right2;
private SpeedControllerGroup rightMotor;
private DifferentialDrive drive;

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS

    public Drivetrain() {
```

(continues on next page)

(continued from previous page)

```

        // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTORS
left1 = new PWMVictorSPX(0);
addChild("left1",left1);
left1.setInverted(false);

left2 = new PWMVictorSPX(1);
addChild("left2",left2);
left2.setInverted(false);

SpeedControllerGroup leftMotor = new SpeedControllerGroup(left1, left2 );
addChild("Left Motor",leftMotor);

right1 = new PWMVictorSPX(5);
addChild("right1",right1);
right1.setInverted(false);

right2 = new PWMVictorSPX(6);
addChild("right2",right2);
right2.setInverted(false);

SpeedControllerGroup rightMotor = new SpeedControllerGroup(right1, right2 );
addChild("Right Motor",rightMotor);

drive = new DifferentialDrive(leftMotor, left1);
addChild("Drive",drive);
drive.setSafetyEnabled(true);
drive.setExpiration(0.1);
drive.setMaxOutput(1.0);

        // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTORS
    }

    // Put methods for controlling this subsystem
    // here. Call these from Commands.

    public void drive(double left, double right) {
        drive.tankDrive(left, right);
    }
}

```

```

class Drivetrain: public frc2::SubsystemBase {
private:
    // It's desirable that everything possible is private except
    // for methods that implement subsystem capabilities
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
frc::PWMVictorSPX m_left1{0};
frc::PWMVictorSPX m_left2{1};
frc::SpeedControllerGroup m_leftMotor{m_left1, m_left2 };
frc::PWMVictorSPX m_right1{5};
frc::PWMVictorSPX m_right2{6};
frc::SpeedControllerGroup m_rightMotor{m_right1, m_right2 };
frc::DifferentialDrive m_drive{m_leftMotor, m_left1};

        // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS

```

(continues on next page)

(continued from previous page)

```

public:
Drivetrain();

    void Periodic() override;
    void SimulationPeriodic() override;
    void Drive(double left, double right);

};

```

```

Drivetrain::Drivetrain(){
SetName("Drivetrain");
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS

AddChild("left1", &m_left1);
m_left1.SetInverted(false);

AddChild("left2", &m_left2);
m_left2.SetInverted(false);

AddChild("Left Motor", &m_leftMotor);


AddChild("right1", &m_right1);
m_right1.SetInverted(false);

AddChild("right2", &m_right2);
m_right2.SetInverted(false);

AddChild("Right Motor", &m_rightMotor);


AddChild("Drive", &m_drive);
m_drive.SetSafetyEnabled(true);
m_drive.SetExpiration(0.1);
m_drive.SetMaxOutput(1.0);

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
}

// Put methods for controlling this subsystem
// here. Call these from Commands.

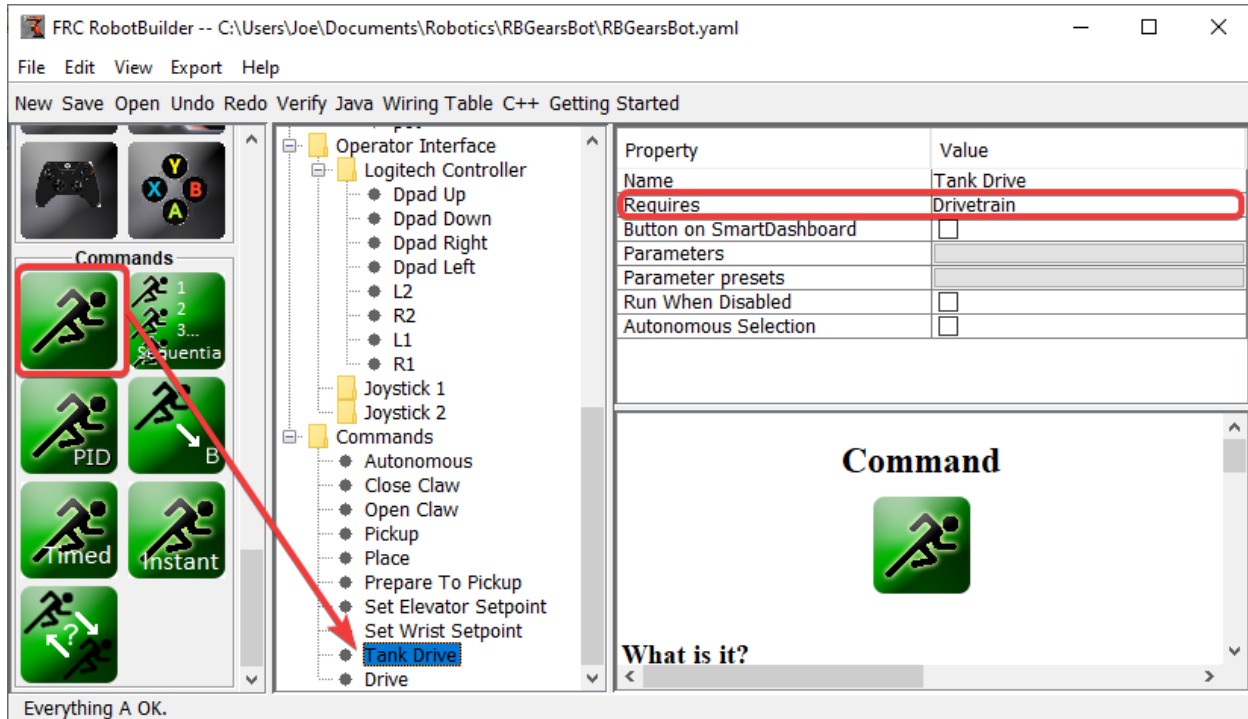
void Drivetrain::Drive(double left, double right) {
    m_drive.TankDrive(left, right);
}

```

Create a method that takes the joystick inputs, in this case the the left and right driver joystick. The values are passed to the RobotDrive object that in turn does tank steering using the joystick values. Also create a method called stop() that stops the robot from driving, this might come in handy later.

Note: Some RobotBuilder output has been removed for this example for clarity

Read Joystick Values and Call the Subsystem Methods



Create a command, in this case called Tank Drive. Its purpose will be to read the joystick values and send them to the Drive Base subsystem. Notice that this command Requires the Drive Train subsystem. This will cause it to stop running whenever anything else tries to use the Drive Train.

Note: Be sure to export your program to C++ or Java before continuing to the next step.

Add the Code to do the Driving

java

```
public class TankDrive extends CommandBase {

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=VARIABLE_DECLARATIONS
    private final Drivetrain m_drivetrain;

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=VARIABLE_DECLARATIONS

    private Joystick leftJoystick = RobotContainer.getInstance().getJoystick1();
    private Joystick rightJoystick = RobotContainer.getInstance().getJoystick2();

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTORS

    public TankDrive(Drivetrain subsystem) {
```

(continues on next page)

(continued from previous page)

```

// END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTORS
// BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=VARIABLE_SETTING

// END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=VARIABLE_SETTING
// BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=REQUIRES

    m_drivetrain = subsystem;
    addRequirements(m_drivetrain);

// END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=REQUIRES
}

// Called when the command is initially scheduled.
@Override
public void initialize() {
}

// Called every time the scheduler runs while the command is scheduled.
@Override
public void execute() {
    m_drivetrain.drive(leftJoystick.getY(), rightJoystick.getY());
}

// Called once the command ends or is interrupted.
@Override
public void end(boolean interrupted) {
    m_drivetrain.drive(0.0, 0.0);
}

// Returns true when the command should end.
@Override
public boolean isFinished() {
    return false;
}

@Override
public boolean runsWhenDisabled() {
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DISABLED
    return false;

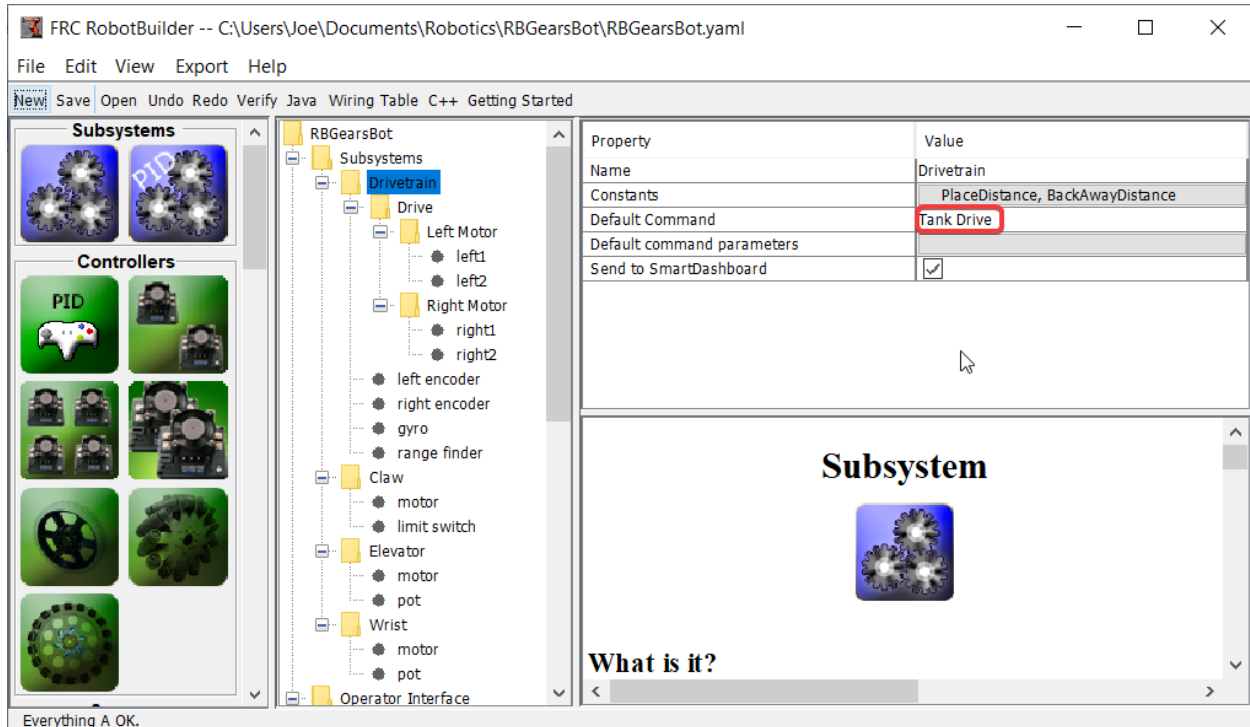
// END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DISABLED
}
}

```

Add code to the execute method to do the actual driving. All that is needed is to get the Joystick objects for the left and right drive joysticks and pass them to the Drive Train subsystem. The subsystem just uses them for the tank steering method on its RobotDrive object. And we get tank steering.

We also filled in the end() method so that when this command is interrupted or stopped, the motors will be stopped as a safety precaution.

Make Default Command



The last step is to make the Tank Drive command be the “Default Command” for the Drive Train subsystem. This means that whenever no other command is using the Drive Train, the Joysticks will be in control. This is probably the desirable behavior. When the autonomous code is running, it will also require the drive train and interrupt the Tank Drive command. When the autonomous code is finished, the DriveWithJoysticks command will restart automatically (because it is the default command), and the operators will be back in control. If you write any code that does teleop automatic driving, those commands should also “require” the DriveTrain so that they too will interrupt the Tank Drive command and have full control.

Note: Be sure to export your program to C++ or Java before continuing.

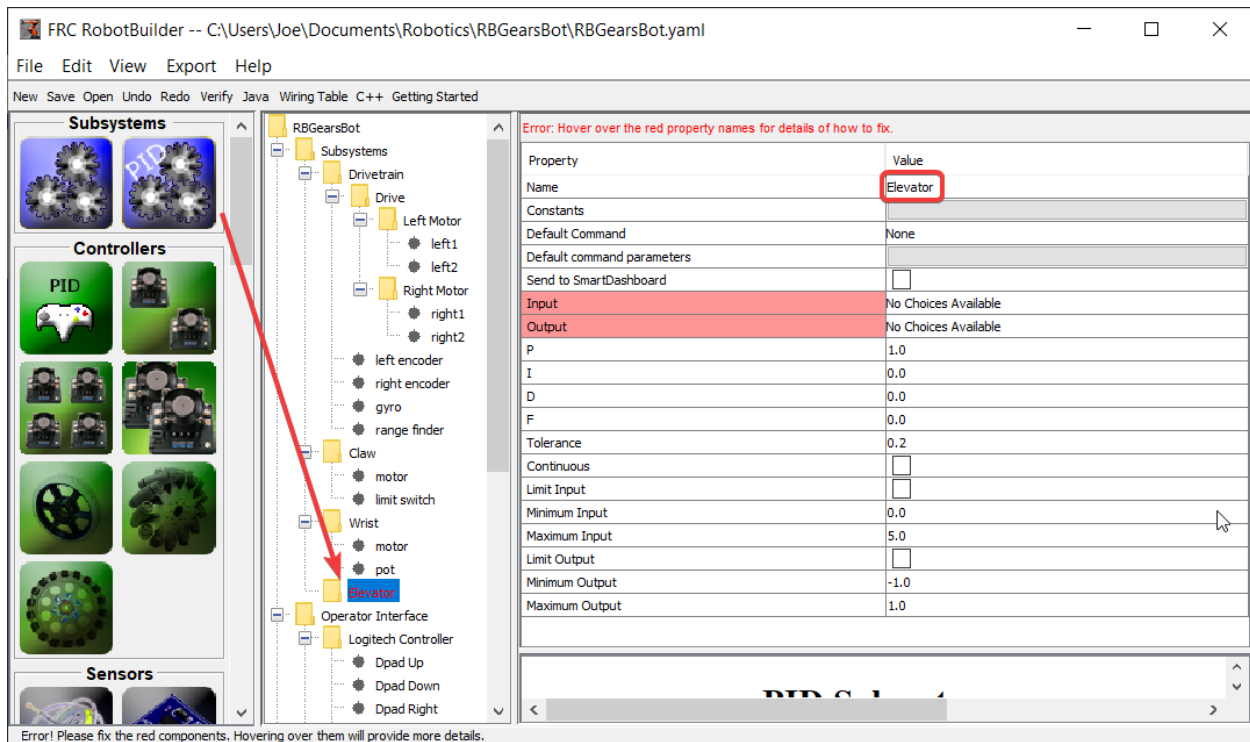
24.3 RobotBuilder - Advanced

Important: RobotBuilder has been updated to support the new commandbased framework! Unfortunately, this documentation is outdated and only for the old commandbased framework. Individuals interested in updating this documentation can open a pull request on the [frc-docs](#) repository.

24.3.1 Using PIDSubsystem to Control Actuators

More advanced subsystems will use sensors for feedback to get guaranteed results for operations like setting elevator heights or wrist angles. The PIDSubsystem has a built-in PIDController to automatically control the mechanism to the correct setpoints.

Create a PIDSubsystem

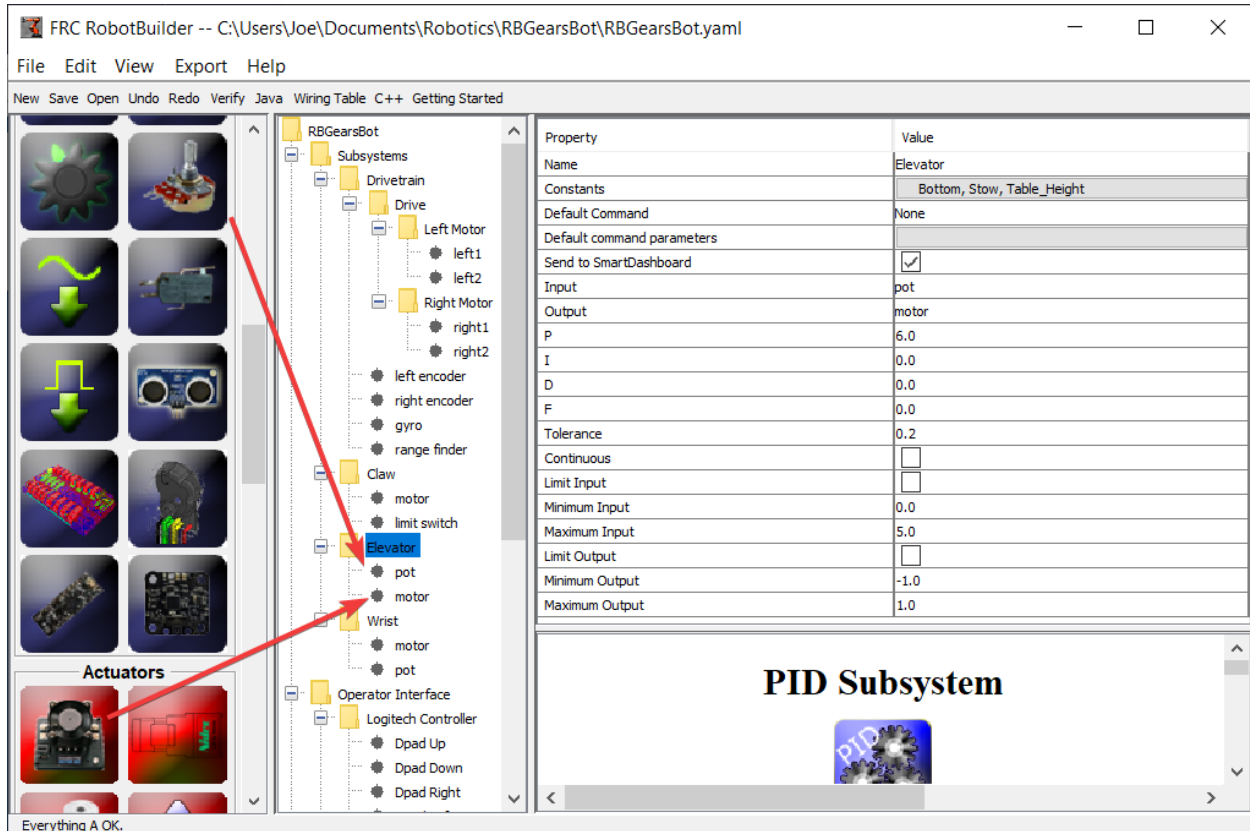


Creating a subsystem that uses feedback to control the position or speed of a mechanism is very easy.

1. Drag a PIDSubsystem from the palette to the Subsystems folder in the robot description
2. Rename the PID Subsystem to a more meaningful name for the subsystem, in this case Elevator

Notice that some of the parts of the robot description have turned red. This indicates that these components (the PIDSubsystem) haven't been completed and need to be filled in. The properties that are either missing or incorrect are shown in red.

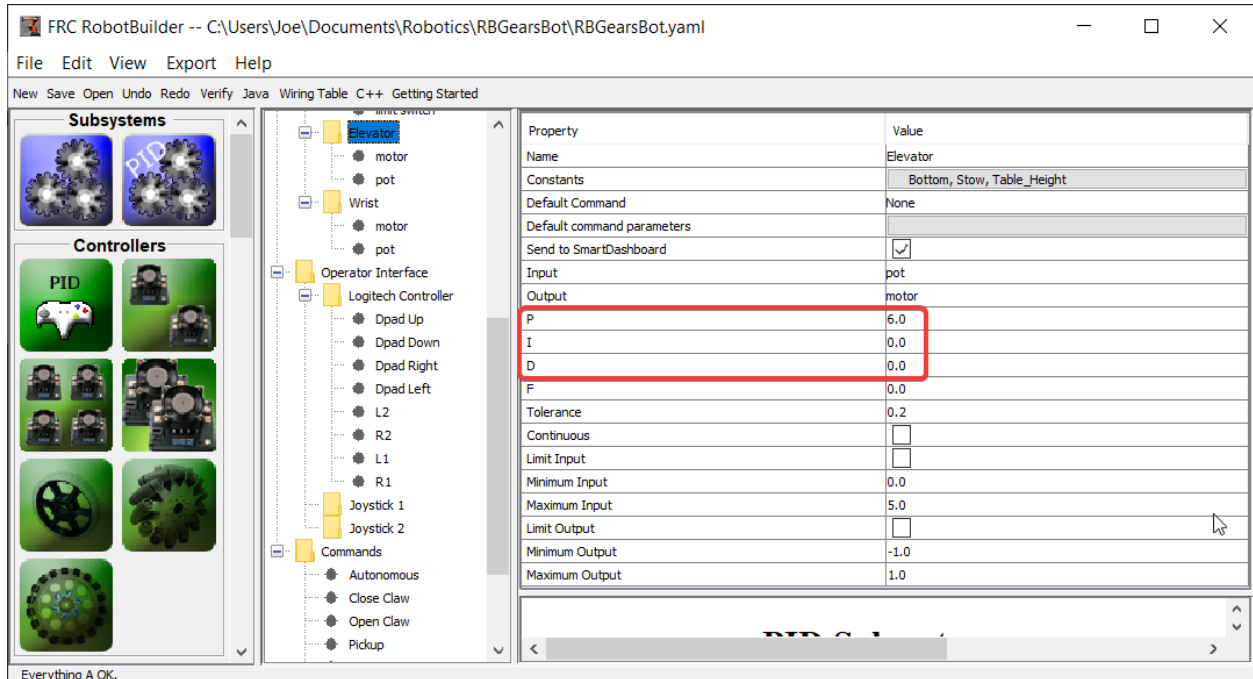
Adding Sensors and Actuators to the PIDSubsystem



Add the missing components for the PIDSubsystem

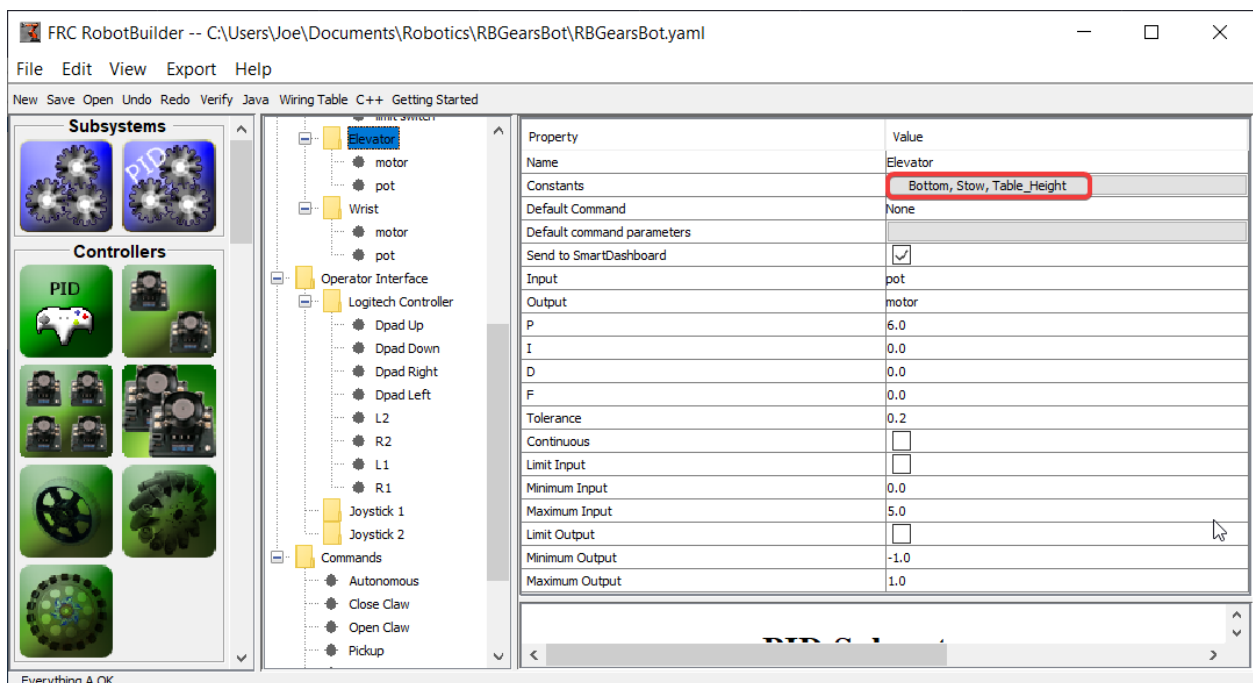
1. Drag in the actuator (a motor controller) to the particular subsystem - in this case the Elevator
2. Drag the sensor that will be used for feedback to the subsystem, in this case the sensor is a potentiometer that might give elevator height feedback.

Fill in the PID Parameters

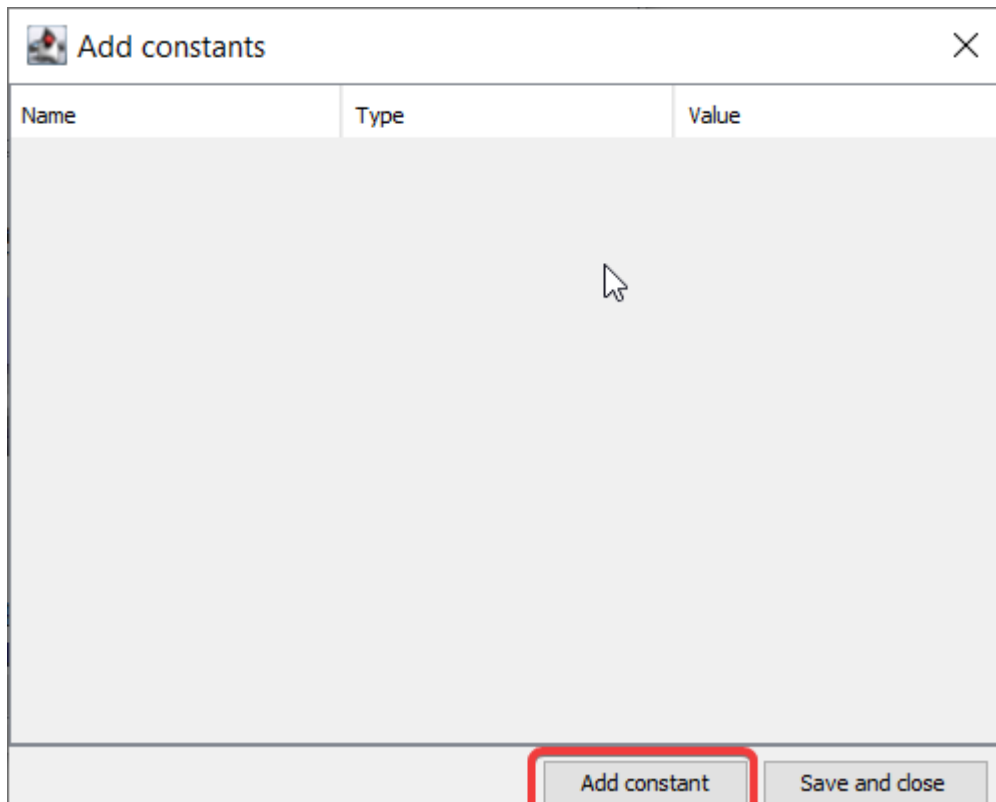


The P, I, and D values need to be filled in to get the desired sensitivity and stability of the component. In the case of our elevator we use a proportional constant of 6.0 and 0 for the I and D terms.

Create Setpoint Constants



In order to make it easier to manage elevator setpoints, we will create constants to manage the setpoints. Click on the constants box to bring up the constants dialog.



Click on the *add constant* button

Name	Type	Value
Bottom	double	4.6
Stow	double	1.65
Table_Height	double	1.58

Buttons: Add constant, Save and close

1. Fill in a name for the constant, in this case: Bottom
2. Select a type for the constant from the drop-down menu, in this case: double
3. Select a value for the constant, in this case: 4.65
4. Click *add constant* to continue adding constants

24.3.2 Writing the Code for a PIDSubsystem

Important: RobotBuilder has been updated to support the new commandbased framework! Unfortunately, this documentation is outdated and only for the old commandbased framework. Individuals interested in updating this documentation can open a pull request on the [frc-docs](#) repository.

PIDSubsystems use feedback to control the actuator and drive it to a particular position. In this example we use an elevator with a 10-turn potentiometer connected to it to give feedback on the height. The skeleton of the PIDSubsystem is generated by the RobotBuilder and we have to fill in the rest of the code to provide the potentiometer value and drive the motor with the output of the embedded PIDController.

Setting the PID Constants

Make sure the Elevator PID subsystem has been created in the RobotBuilder. Once it's all set, generate Java/C++ code for the project using the Export menu or the Java/C++ toolbar menu.

Add Constants for Elevator Preset Positions

Java

C++

```
public class Elevator extends PIDSubsystem {
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATION
    AnalogChannel potentiometer = RobotMap.ELEVATOR_POTENTIOMETER;
    Victor victor = RobotMap.ELEVATOR_VICTOR;
    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATION

    public static final double BOTTOM = 4.6,
                               STOW = 1.65,
                               TABLE_HEIGHT = 1.58;

    public Elevator() {
        // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER, ID=PID
        super("Elevator", 1.0, 0.0, 0.0);
        getPIDController().setContinuous(false);
        // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER, ID=PID

        setSetpoint(STOW);
        enable();
    }

    public void initDefaultCommand() {

    }

    protected double returnPIDInput() {
        // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER, ID=SOURCE
        return potentiometer.pidGet();
        // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER, ID=SOURCE
    }

    protected void usePIDOutput(double output) {

    }
}
```

```
#include "Elevator.h"
#include "../Robotmap.h"
#include "SmartDashboard/SmartDashboard.h"

// BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=PID
Elevator::Elevator() : PIDSubsystem("Elevator", 1.0, 0.0, 0.0) {
    GetPIDController()->SetContinuous(false);

    // END AUTOGENERATED COD, SOURCE=ROBOTBUILDER ID=PID
```

(continues on next page)

(continued from previous page)

```

// BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER, ID=DECLARATIONS

potentiometer = RobotMap::ELEVATOR_POTENTIOMETER;
victor = RobotMap::ELEVATOR_VICTOR;

// END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER, ID=DECLARATIONS
SetSetpoint(STOW);
Enable();
}

```

To make it easier to drive the elevator to preset positions, we added preset positions for the bottom, stow, and table height. Then the elevator is set to the STOW position by setting the PID setpoint and the PID controller is enabled. This will cause the elevator to move to the stowed position when the robot is enabled.

Adding the Constants to the Header File (C++ Only)

C++

```

#pragma once

#include "frc/commands/PIDSubsystem.h"
#include "frc/AnalogInput.h"
#include "frc/Victor.h"

class Elevator: public PIDSubsystem {
public:
    static const double BOTTOM = 4.6;
    static const double STOW = 1.65;
    static const double TABLE_HEIGHT = 1.58;

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
    AnalogInput* potentiometer;
    Victor* victor;
    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS

    Elevator();
    double ReturnPIDInput();
    void UsePIDOutput(double output);
    void InitDefaultCommand();
};

```

Return PID Input Values

Java

C++

```

protected double returnPIDInput() {
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER, ID=SOURCE
    return potentiometer.pidGet();
    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER, ID=SOURCE
}

```

```
double Elevator::ReturnPIDInput() {  
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=SOURCE  
    return potentiometer->PIDGet();  
    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=SOURCE  
}
```

The `returnPIDInput()` method is used to set the value of the sensor that is providing the feedback for the PID controller. In this case, the code is automatically generated and returns the potentiometer raw analog input value (a number that ranges from 0-1023). In our case we would like the PID controller to be based on the average voltage read by the analog input for the potentiometer, not the raw value.

If we just change the line:

`return potentiometer.pidGet();` for Java or `return potentiometer->PIDGet();` for C++

it will be overwritten by RobotBuilder next time we export to Java. You can tell which lines are automatically generated by looking at the “//BEGIN AUTOGENERATED CODE” and “//END AUTOGENERATED CODE” comments. Any code in-between those markers will be overwritten next time RobotBuilder is run. You’re free to change anything outside of those blocks.

Return the Average Voltage

Java

C++

```
protected double returnPIDInput() {  
    return potentiometer.getAverageVoltage();  
}
```

```
double Elevator::ReturnPIDInput() {  
    return potentiometer->GetAverageVoltage();  
}
```

To get around the problem from the last step, the comment blocks can be removed. Then if the line is changed as shown, it will no longer be overwritten by RobotBuilder.

Remember, if we just wanted to add code to a method it could be added safely outside of the comment blocks.

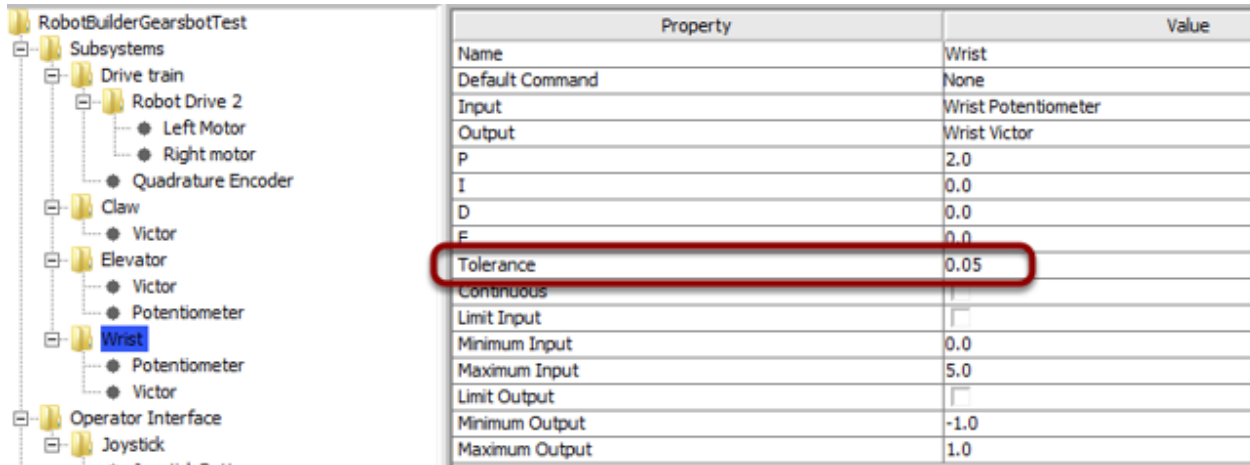
That’s all that is required to create the Elevator PIDSubsystem.

24.3.3 Setpoint Command

Important: RobotBuilder has been updated to support the new commandbased framework! Unfortunately, this documentation is outdated and only for the old commandbased framework. Individuals interested in updating this documentation can open a pull request on the [frc-docs](#) repository.

A common use case in robot programs is to drive an actuator to a particular angle or position that is measured using a potentiometer or encoder. This happens so often that there is a shortcut in RobotBuilder to do this task. It is called the Setpoint command and it’s one of the choices on the palette or the right-click context menu that can be inserted under “Commands”.

Start with a PIDSubsystem

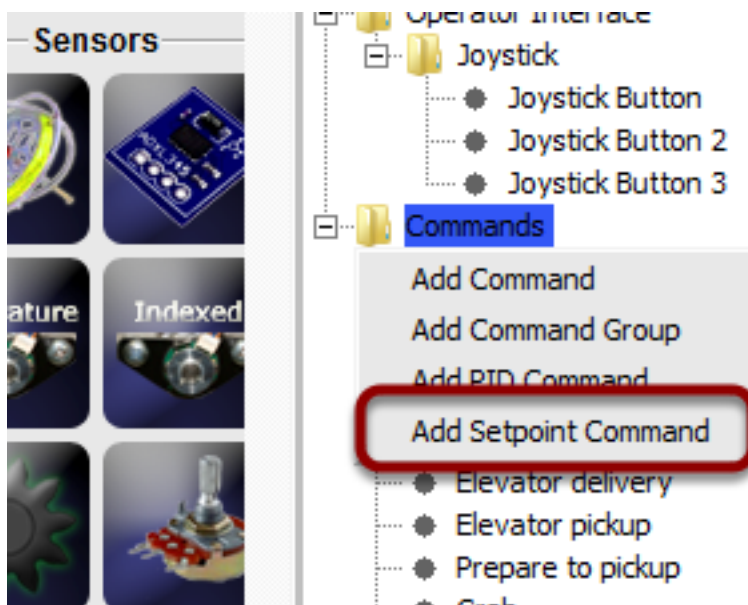


Property	Value
Name	Wrist
Default Command	None
Input	Wrist Potentiometer
Output	Wrist Victor
P	2.0
I	0.0
D	0.0
F	0.0
Tolerance	0.05
Continuous	<input type="checkbox"/>
Limit Input	<input type="checkbox"/>
Minimum Input	0.0
Maximum Input	5.0
Limit Output	<input type="checkbox"/>
Minimum Output	-1.0
Maximum Output	1.0

Suppose in a robot there is a wrist joint with a potentiometer that measures the angle. First *create a PIDSubsystem* that include the motor that moves the wrist joint and the potentiometer that measures the angle. The PIDSubsystem should have all the PID constants filled in and working properly.

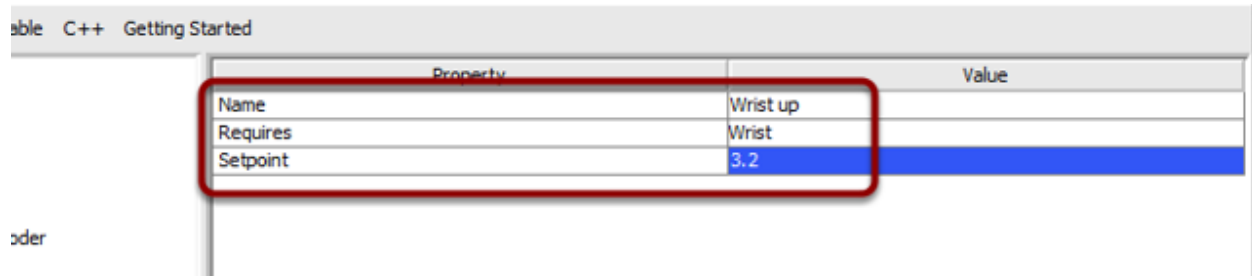
It is important to set the **Tolerance** parameter. This controls how far off the current value can be from the setpoint and be considered on target. This is the criteria that the Setpoint-Command uses to move onto the next command.

Creating the Setpoint Command



Right-click on the Commands folder in the palette and select “Add Setpoint command”.

Setpoint Command Parameters



Fill in the name of the new command. The Requires field is the PIDSubsystem that is being driven to a setpoint and the Setpoint parameter is the setpoint value for the PIDSubsystem. There is no need to fill in any code for this command, it is automatically created by RobotBuilder.

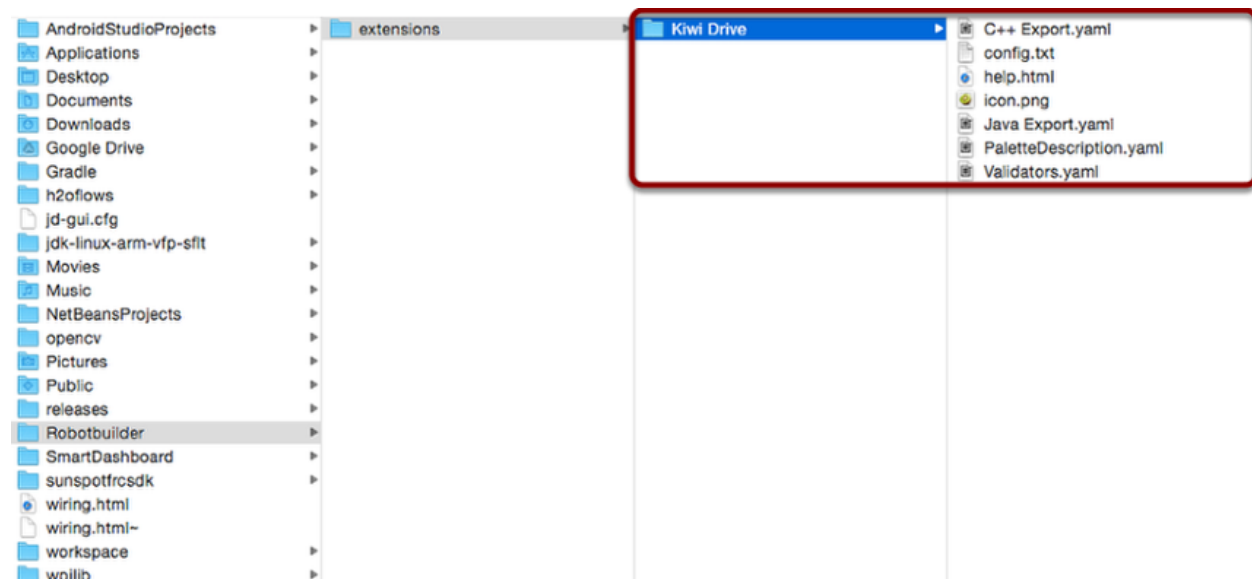
Whenever this command is scheduled, it will automatically drive the subsystem to the specified setpoint. When the setpoint is reached within the tolerance specified in the PIDSubsystem, the command ends and the next command starts. It is important to specify a tolerance in the PIDSubsystem or this command might never end because the tolerance is not achieved.

Note: For more information about PID Control, please see the [Advanced Controls Introduction](#).

24.3.4 Adding Custom Components

RobotBuilder works very well for creating robot programs that just use WPILib for motors, controllers, and sensors. But for teams that use custom classes, RobotBuilder doesn't have any support for those classes, so a few steps need to be taken to use them in RobotBuilder

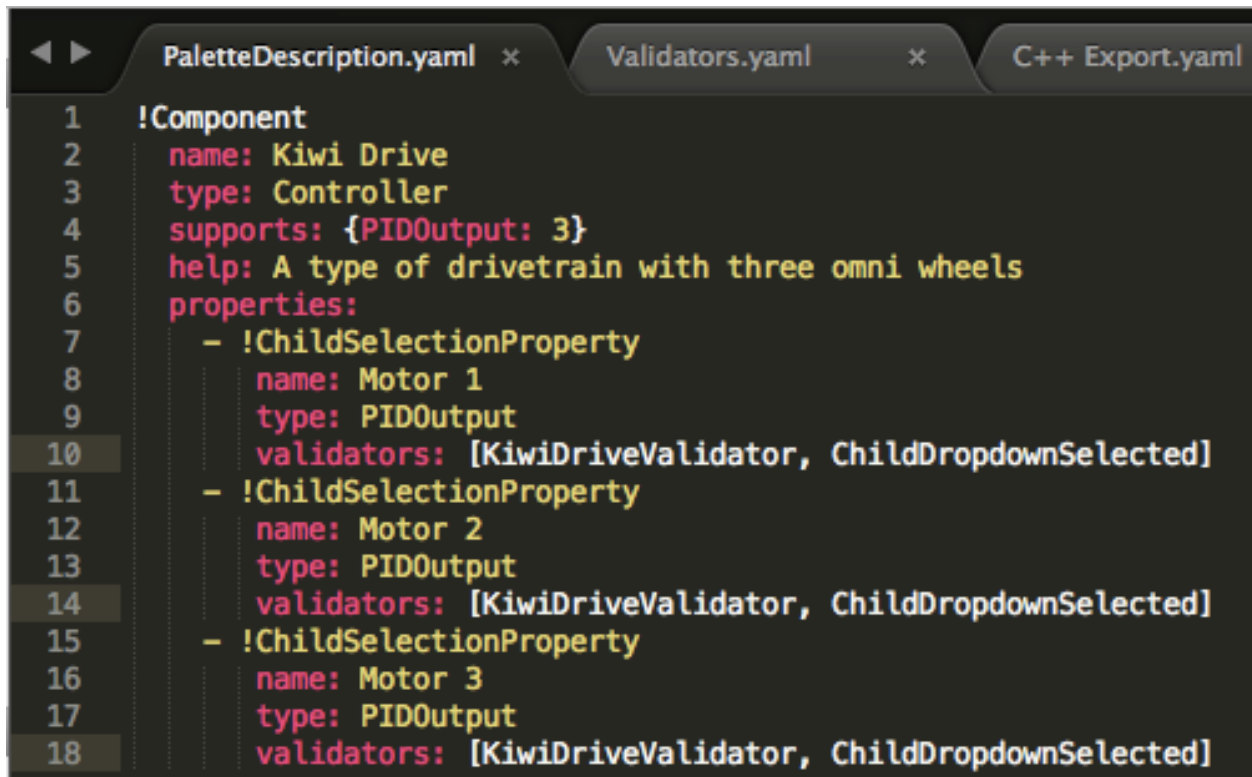
Custom Component Structure



Custom components all go in ~/wpilib/YYYY/Robotbuilder/extensions where ~ is C:\Users\Public on Windows and YYYY is the FRC® year.

There are seven files and one folder that are needed for a custom component. The folder contains the files describing the component and how to export it. It should have the same name as the component (e.g. "Kiwi Drive" for a kiwi drive controller, "Robot Drive 6" for a six-motor drive controller, etc.). The files should have the same names and extensions as the ones shown here. Other files can be in the folder along with these seven, but the seven must be present for RobotBuilder to recognize the custom component.

PaletteDescription.yaml



```

1  !Component
2  name: Kiwi Drive
3  type: Controller
4  supports: {PIDOutput: 3}
5  help: A type of drivetrain with three omni wheels
6  properties:
7    - !ChildSelectionProperty
8      name: Motor 1
9      type: PIDOutput
10     validators: [KiwiDriveValidator, ChildDropdownSelected]
11    - !ChildSelectionProperty
12      name: Motor 2
13      type: PIDOutput
14     validators: [KiwiDriveValidator, ChildDropdownSelected]
15    - !ChildSelectionProperty
16      name: Motor 3
17      type: PIDOutput
18     validators: [KiwiDriveValidator, ChildDropdownSelected]

```

Line-by-line:

- **!Component:** Declares the beginning of a new component
- **name:** The name of the component. This is what will show up in the palette/tree – this should also be the same as the name of the containing folder
- **type:** the type of the component (these will be explained in depth later on)
- **supports:** a map of the amount of each type of component this can support. Motor controllers in RobotBuilder are all PIDOutputs, so a kiwi drive can support three PIDOutputs. If a component doesn't support anything (such as sensors or motor controllers), just leave this line out
- **help:** a short string that gives a helpful message when one of these components is hovered over
- **properties:** a list of the properties of this component. In this kiwi drive example, there are three very similar properties, one for each motor. A ChildSelectionProperty allows

the user to choose a component of the given type from the subcomponents of the one being edited (so here, they would show a dropdown asking for a PIDOutput - i.e. a motor controller - that has been added to the kiwi drive)

The types of component RobotBuilder supports (these are case-sensitive):

- Command
- Subsystem
- PIDOutput (motor controller)
- PIDSource (sensor that implements PIDSource e.g. analog potentiometer, encoder)
- Sensor (sensor that does not implement PIDSource e.g. limit switch)
- Controller (robot drive, PID controller, etc.)
- Actuator (an output that is not a motor, e.g. solenoid, servo)
- Joystick
- Joystick Button

Properties

The properties relevant for a custom component:

- StringProperty: used when a component needs a string e.g. the name of the component
- BooleanProperty: used when a component needs a boolean value e.g. putting a button on the SmartDashboard
- DoubleProperty: used when a component needs a number value e.g. PID constantsChoicesProperty
- ChildSelectionProperty: used when you need to choose a child component e.g. motor controllers in a RobotDrive
- TypeSelectionProperty: used when you need to choose any component of the given type from anywhere in the program e.g. input and output for a PID command

The fields for each property are described below:

A property is one of:

- !StringProperty
name: The name of this property, should be unique within this component
validator: Optional. The validator that should be used to validate this property.
default: The default value when no other is presented.
- !BooleanProperty
name: The name of this property, should be unique within this component
validator: Optional. The validator that should be used to validate this property.
default: The default value when no other is presented.
- !DoubleProperty
name: The name of this property, should be unique within this component
validator: Optional. The validator that should be used to validate this property.
default: The default value when no other is presented.
- !FileProperty
name: The name of this property, should be unique within this component
validator: Optional. The validator that should be used to validate this property.
default: The default value when no other is presented.
extension: The extension at the end of this file without the '.'
folder: Whether or not to select folders instead of files
- !ChoicesProperty
name: The name of this property, should be unique within this component
validator: Optional. The validator that should be used to validate this property.
default: The default value when no other is presented.
choices: List of choices to present to the user.
- !ChildSelectionProperty
name: The name of this property, should be unique within this component
validator: Optional. The validator that should be used to validate this property.
default: The default value when no other is presented.
type: Type of the child to select.
- !TypeSelectionProperty
name: The name of this property, should be unique within this component
validator: Optional. The validator that should be used to validate this property.
default: The default value when no other is presented.
type: Type of component to select.

Validators.yaml

```

1  !DistinctValidator
2      name: KiwiDriveValidator
3      fields: ["Motor 1", "Motor 2", "Motor 3"]

```

You may have noticed “KiwiDriveValidator” in the validators entry of each of the motor properties in PaletteDescription.yaml. It’s not a built-in validator, so it had to be defined in Validators.yaml. This example validator is very simple - it just makes sure that each of the named fields has a different value than the others.

Built-in Validators and Validator Types

```

Validators:
- !DistinctValidator
  name: RobotDrive2
  fields: ["Left Motor", "Right Motor"]
- !DistinctValidator
  name: RobotDrive4
  fields: ["Left Front Motor", "Left Rear Motor", "Right Front Motor", "Right Rear Motor"]
- !ExistsValidator
  name: ChildDropdownSelected
  ignore: [null, "null", "", 0, 1, 2, 3, "No Choices Available", "None"]
  error: "You must select a component of the valid type beneath this item. If no options exist, drag one under this component."
- !ExistsValidator
  name: TypeDropdownSelected
  ignore: [null, "null", "", 0, 1, 2, 3, "No Choices Available", "None"]
  error: "You must select a component of the valid type. If no options exist, create a new component of the right type."
- !UniqueValidator
  name: AnalogInput
  fields: [Channel (Analog)]
- !UniqueValidator
  name: DigitalChannel
  fields: [Channel (Digital)]
- !UniqueValidator
  name: PWMOutput
  fields: [Channel (PWM)]
- !UniqueValidator
  name: CANID
  fields: [CAN ID]
- !UniqueValidator
  name: Joystick
  fields: [Number]
- !UniqueValidator
  name: RelayOutput
  fields: [Channel (Relay)]
- !UniqueValidator
  name: Solenoid
  fields: [Channel (Solenoid), PCM (Solenoid)]
- !UniqueValidator
  name: PCMCompID
  fields: [PCM ID]
- !ListValidator
  name: List

```

The built-in validators are very useful (especially the UniqueValidators for port/channel use), but sometimes a custom validator is needed, like in the previous step

- DistinctValidator: Makes sure the values of each of the given fields are unique
- ExistsValidator: Makes sure that a value has been set for the property using this validator
- UniqueValidator: Makes sure that the value for the property is unique globally for the given fields

- ListValidator: Makes sure that all the values in a list property are valid

C++ Export.yaml



```

1 Kiwi Drive:
2   Defaults: "CustomComponent,None"
3   ClassName: "KiwiDrive"
4   Construction: "#variable($Name).reset(new ${ClassName}({#variable($Motor_1), #variable($Motor_2), #variable($Motor_3)}));"

```

A line-by-line breakdown of the file:

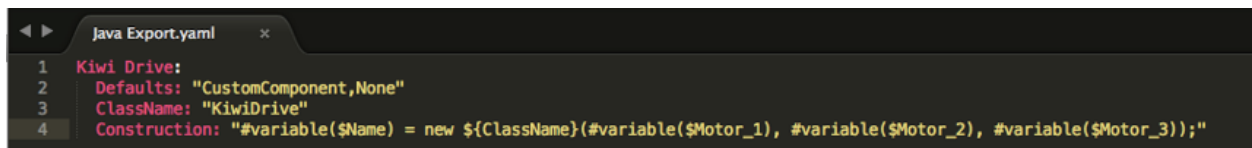
- Kiwi Drive: the name of the component being exported. This is the same as the name set in PaletteDescription.yaml, and the name of the folder containing this file
- Defaults: provides some default values for includes needed by this component, the name of the class, a construction template, and more. The CustomComponent default adds an include for Custom/\${ClassName}.h to every generated file that uses the component (e.g. RobotDrive.h would have `#include "Custom/KiwiDrive.h"` at the top of the file)
- ClassName: the name of the custom class you're adding.
- Construction: an instruction for how the component should be constructed. Variables will be replaced with their values ("`${ClassName}`" will be replaced with "KiwiDrive"), then macros will be evaluated (for example, `#variable($Name)` may be replaced with `drivebaseKiwiDrive`).

This example expects a KiwiDrive class with the constructor

```
KiwiDrive(SpeedController, SpeedController, SpeedController)
```

If your team uses Java, this file can be empty.

Java Export.yaml



```

1 Kiwi Drive:
2   Defaults: "CustomComponent,None"
3   ClassName: "KiwiDrive"
4   Construction: "#variable($Name) = new ${ClassName}({#variable($Motor_1), #variable($Motor_2), #variable($Motor_3)});"

```

Very similar to the C++ export file; the only difference should be the Construction line. This example expects a KiwiDrive class with the constructor

```
KiwiDrive(SpeedController, SpeedController, SpeedController)
```

If your team uses C++, this file can be empty.

Using Macros and Variables

Macros are simple functions that RobotBuilder uses to turn variables into text that will be inserted into generated code. They always start with the “#” symbol, and have a syntax similar to functions: `<macro_name>(arg0, arg1, arg2, ...)`. The only macro you’ll probably need to use is `#variable(component_name)`

`#variable` takes a string, usually the a variable defined somewhere (i.e. “Name” is the name given to the component in RobotBuilder, such as “Arm Motor”), and turns it into the name of a variable defined in the generated code. For example, `#variable("Arm Motor")` results in the string `ArmMotor`

Variables are referenced by placing a dollar sign (“\$”) in front of the variable name, which an optionally be placed inside curly braces to easily distinguish the variable from other text in the file. When the file is parsed, the dollar sign, variable name, and curly braces are replaced with the value of the variable (e.g. `${ClassName}` is replaced with `KiwiDrive`).

Variables are either component properties (e.g. “Motor 1”, “Motor 2”, “Motor 3” in the kiwi drive example), or one of the following:

1. `Short_Name`: the name given to the component in the editor panel in RobotBuilder
2. `Name`: the full name of the component. If the component is in a subsystem, this will be the short name appended to the name of the subsystem
3. `Export`: The name of the file this component should be created in, if any. This should be “RobotMap” for components like actuators, controllers, and sensors; or “OI” for things like gamepads or other custom OI components. Note that the “CustomComponent” default will export to the RobotMap.
4. `Import`: Files that need to be included or imported for this component to be able to be used.
5. `Declaration`: an instruction, similar to `Construction`, for how to declare a variable of this component type. This is taken care of by the default “None”
6. `Construction`: an instruction for how to create a new instance of this component
7. `LiveWindow`: an instruction for how to add this component to the LiveWindow
8. `Extra`: instructions for any extra functions or method calls for this component to behave correctly, such as encoders needing to set the encoding type.
9. `Prototype (C++ only)`: The prototype for a function to be created in the file the component is declared in, typically a getter in the OI class
10. `Function`: A function to be created in the file the component is declared in, typically a getter in the OI class
11. `PID`: An instruction for how to get the PID output of the component, if it has one (e.g. `#variable($Short_Name) ->PIDGet()`)
12. `ClassName`: The name of the class that the component represents (e.g. `KiwiDrive` or `Joystick`)

If you have variables with spaces in the name (such as “Motor 1”, “Right Front Motor”, etc.), the spaces need to be replaced with underscores when using them in the export files.

help.html

```

1  <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
2
3  <head>
4      <link rel="stylesheet" href="styles.css" type="text/css" media="screen" />
5  </head>
6
7  <body>
8      <h1>Kiwi Drive</h1>
9      <center></center>
10     <h2>What is it?</h2>
11     <p>
12         Kiwi drive is a type of omni-directional drivetrain with three omni wheels,
13         usually at 120° angles to each other.
14     </p>
15     <h2>Properties</h2>
16     <dl>
17         <dt>Motor 1</dt>
18         <dd>The first motor</dd>
19         <dt>Motor 2</dt>
20         <dd>The second motor</dd>
21         <dt>Motor 3</dt>
22         <dd>The third motor</dd>
23     </dl>
24     <h2>See Also</h2>
25     <ul>
26         <li>
27             <a href="http://en.wikipedia.org/wiki/Kiwi_drive">Kiwi drive on Wikipedia</a>
28         </li>
29     </ul>
30 </body>
31
32 </html>
33

```

A HTML file giving information on the component. It is better to have this be as detailed as possible, though it certainly isn't necessary if the programmer(s) are familiar enough with the component, or if it's so simple that there's little point in a detailed description.

config.txt

```

1  section=Controllers

```

A configuration file to hold miscellaneous information about the component. Currently, this only has the section of the palette to put the component in.

The sections of the palette (these are case sensitive):

- Subsystems
- Controllers

- Sensors
- Actuators
- Pneumatics
- OI
- Commands

icon.png

The icon that shows up in the palette and the help page. This should be a 64x64 .png file.

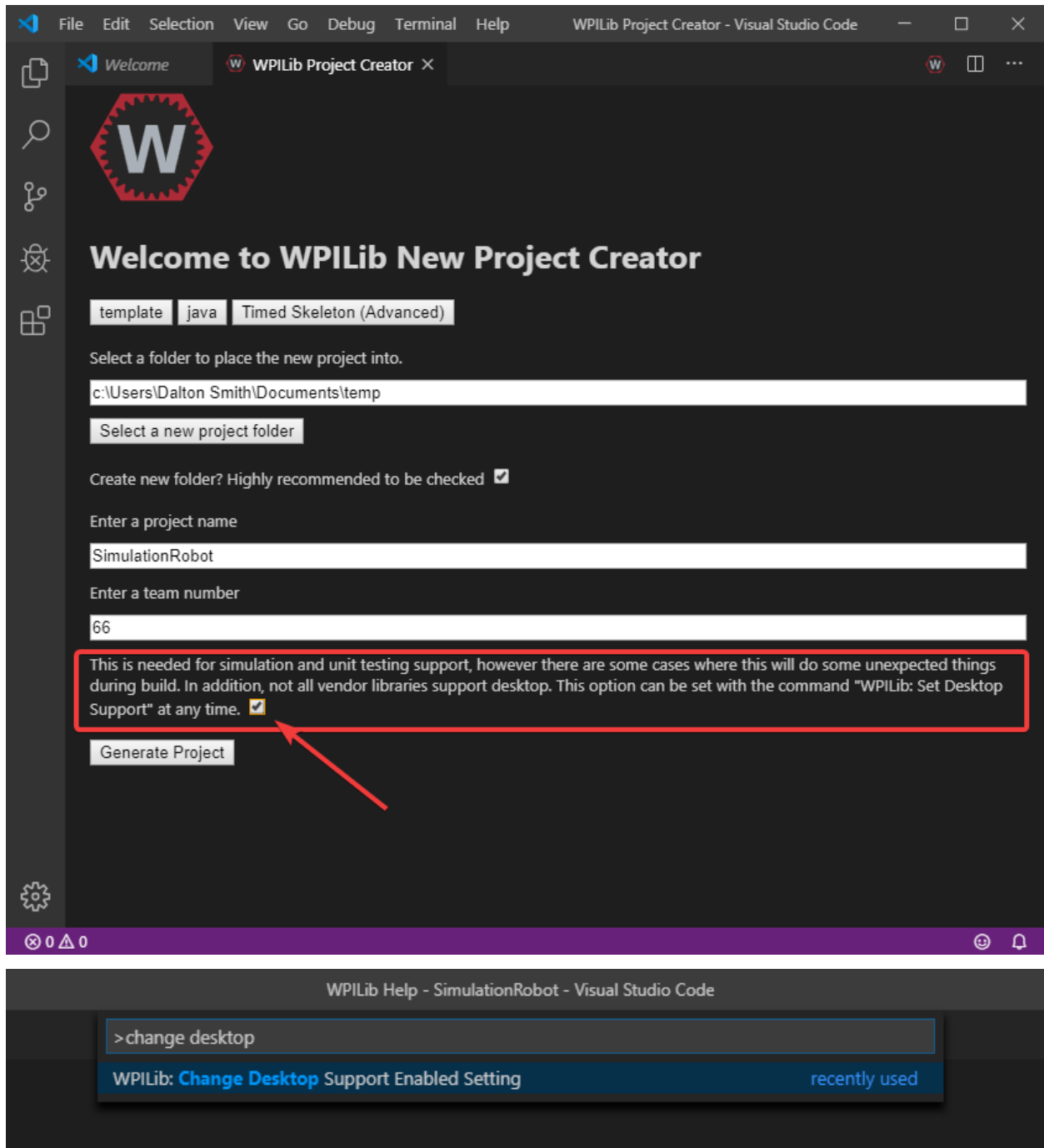
It should use the color scheme and general style of the section it's in to avoid visual clutter, but this is entirely optional. Photoshop .psd files of the icons and backgrounds are in [src/main/icons/icons](#) and png files of the icons and backgrounds are in [src/main/resources/icons](#).

25.1 Introduction to Robot Simulation

Often a team may want to test their code without having an actual robot available. WPILib provides teams with the ability to simulate various robot features using simple gradle commands.

25.1.1 Enabling Desktop Support

Use of the Desktop Simulator requires Desktop Support to be enabled. This can be done by checking the “Enable Desktop Support Checkbox” when creating your robot project or by running “WPILib: Change Desktop Support Enabled Setting” from the Visual Studio Code command palette.



Desktop support can also be enabled by manually editing your `build.gradle` file located at the root of your robot project. Simply change `includeDesktopSupport = false` to `includeDesktopSupport = true`

```
def includeDesktopSupport = true
```

Important: It is important to note that enabling desktop/simulation support can have unintended consequences. Not all vendors will support this option, and code that uses their

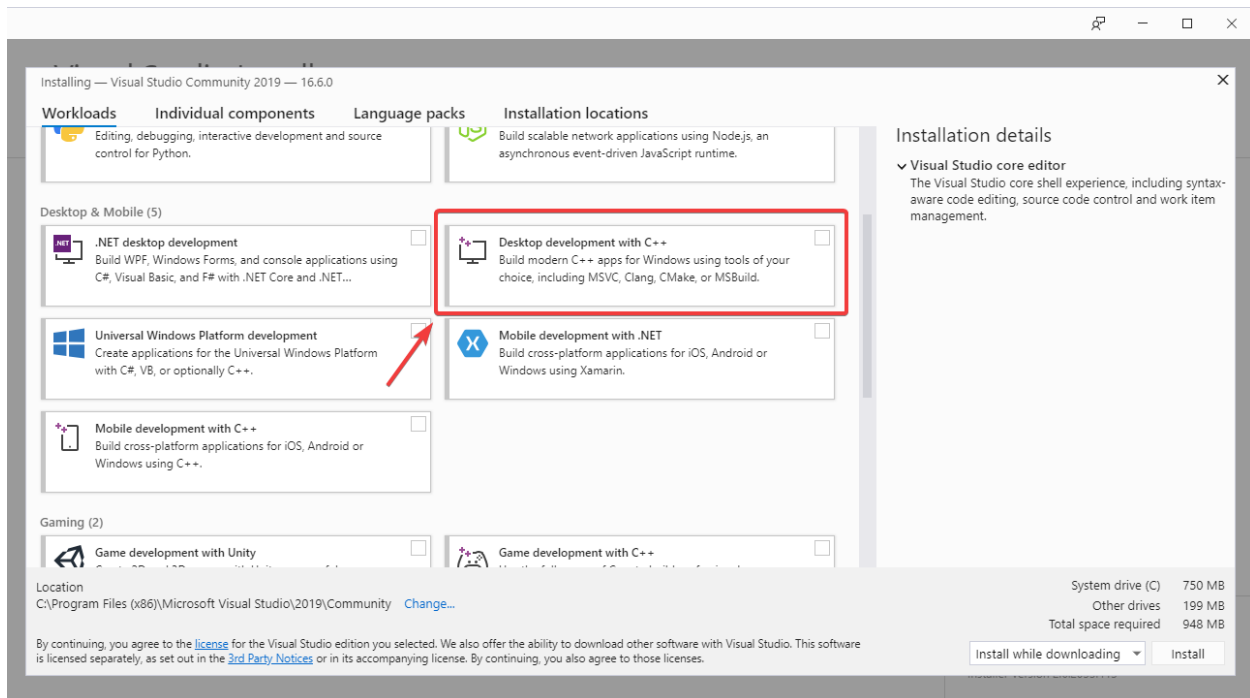
libraries may even crash when attempting to run simulation!

If at any point in time you want to disable Desktop Support, simply re-run the “WPILib: Change Desktop Support Enabled Setting” from the command palette.

Additional C++ Dependency

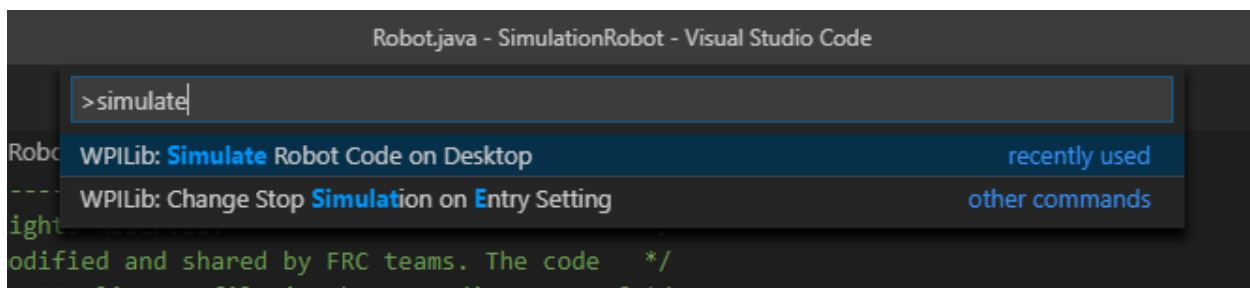
C++ robot simulation requires that a native compiler to be installed. For Windows, this would be [Visual Studio 2019](#) (**not** VS Code), macOS requires [Xcode](#), and Linux (Ubuntu) requires the [build-essential](#) package.

Ensure the Desktop Development with C++ option is checked in the Visual Studio installer for simulation support.



25.1.2 Running Robot Simulation

Basic robot simulation can be run using VS Code. This can be done without using any commands by using VS Code’s command palette.



Your console output in Visual Studio Code should look like the below. However, teams probably will want to actually *test* their code versus just running the simulation. This can be done using *WPILib's Simulation GUI*.

```
***** Robot program starting *****  
Default disabledInit() method... Override me!  
Default disabledPeriodic() method... Override me!  
Default robotPeriodic() method... Override me!
```

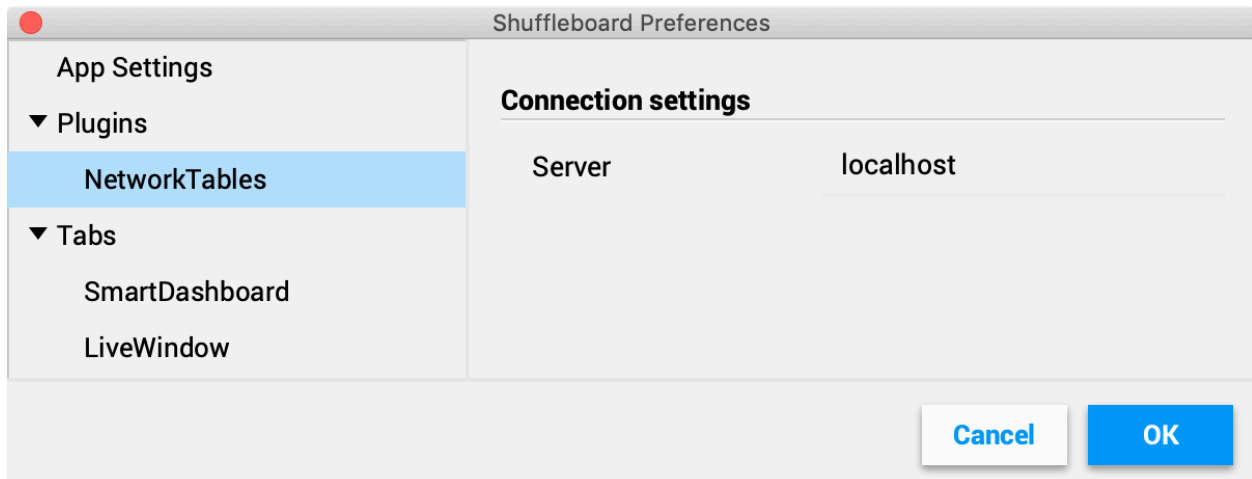
Important: Simulation can also be run outside of VS Code using `./gradlew simulateJava`. It's important to note that C++ simulation is not available through command-line at this time.

25.1.3 Running Robot Dashboards

Both Shuffleboard and SmartDashboard can be used with WPILib simulation.

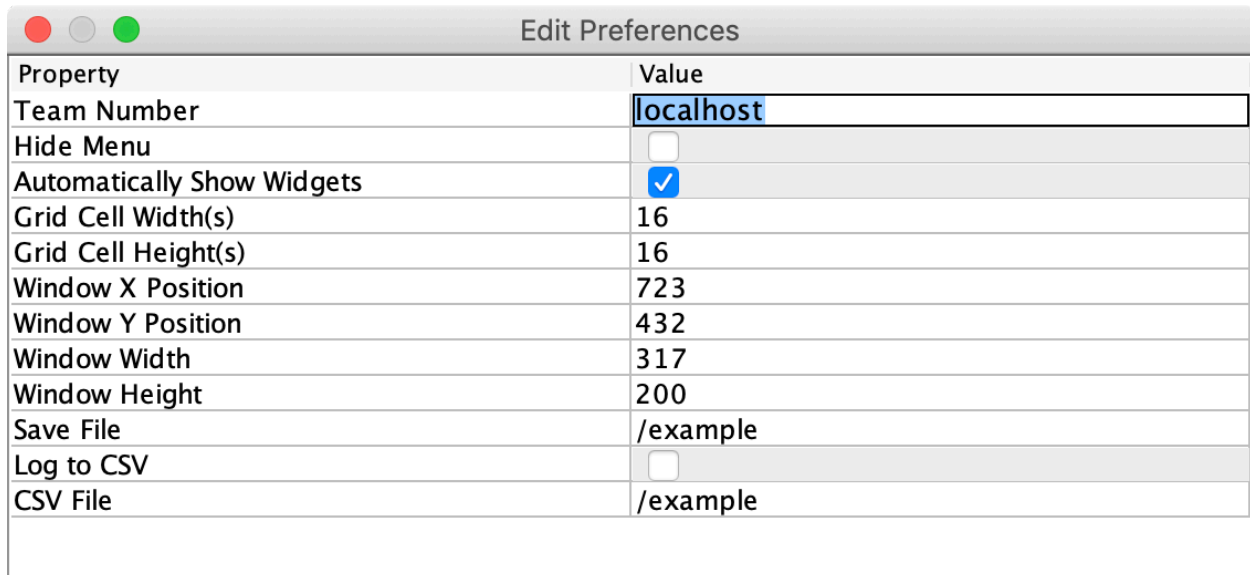
Shuffleboard

Shuffleboard is automatically configured to look for a NetworkTables instance from the robotRIO but **not from other sources**. To connect to Shuffleboard, open Shuffleboard preferences from the File menu and select NetworkTables under Plugins on the left navigation bar. In the Server field, type in the IP address or hostname of the NetworkTables host. For a standard simulation configuration, use `localhost`.



SmartDashboard

SmartDashboard is automatically configured to look for a NetworkTables instance from the roboRIO, but **not from other sources**. To connect to SmartDashboard, open SmartDashboard preferences under the File menu and in the Team Number field, enter the IP address or hostname of the NetworkTables host. For a standard simulation configuration, use localhost.



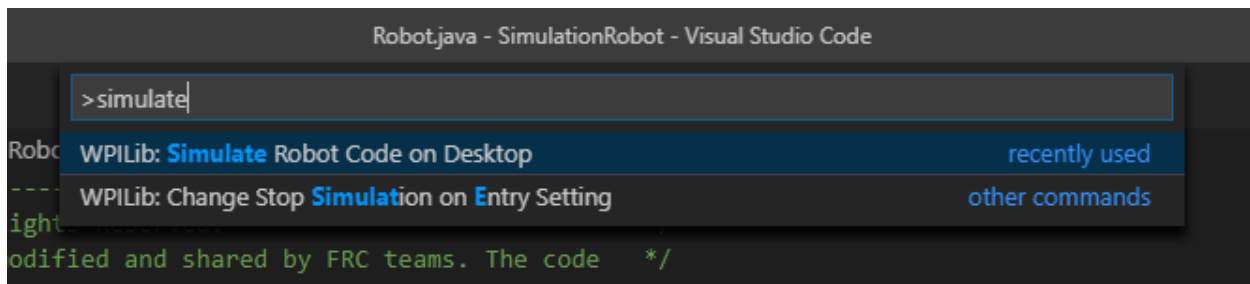
Property	Value
Team Number	localhost
Hide Menu	<input type="checkbox"/>
Automatically Show Widgets	<input checked="" type="checkbox"/>
Grid Cell Width(s)	16
Grid Cell Height(s)	16
Window X Position	723
Window Y Position	432
Window Width	317
Window Height	200
Save File	/example
Log to CSV	<input type="checkbox"/>
CSV File	/example

25.2 Simulation Specific User Interface Elements

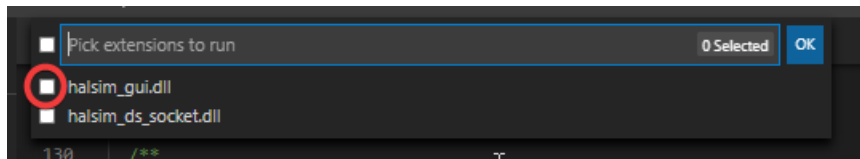
WPILib has extended robot simulation to introduce a graphical user interface (GUI) component. This allows teams to easily visualize their robot's inputs and outputs.

Note: The Simulation GUI is very similar in many ways to *Glass*. Some of the following pages will link to Glass sections that describe elements common to both GUIs.

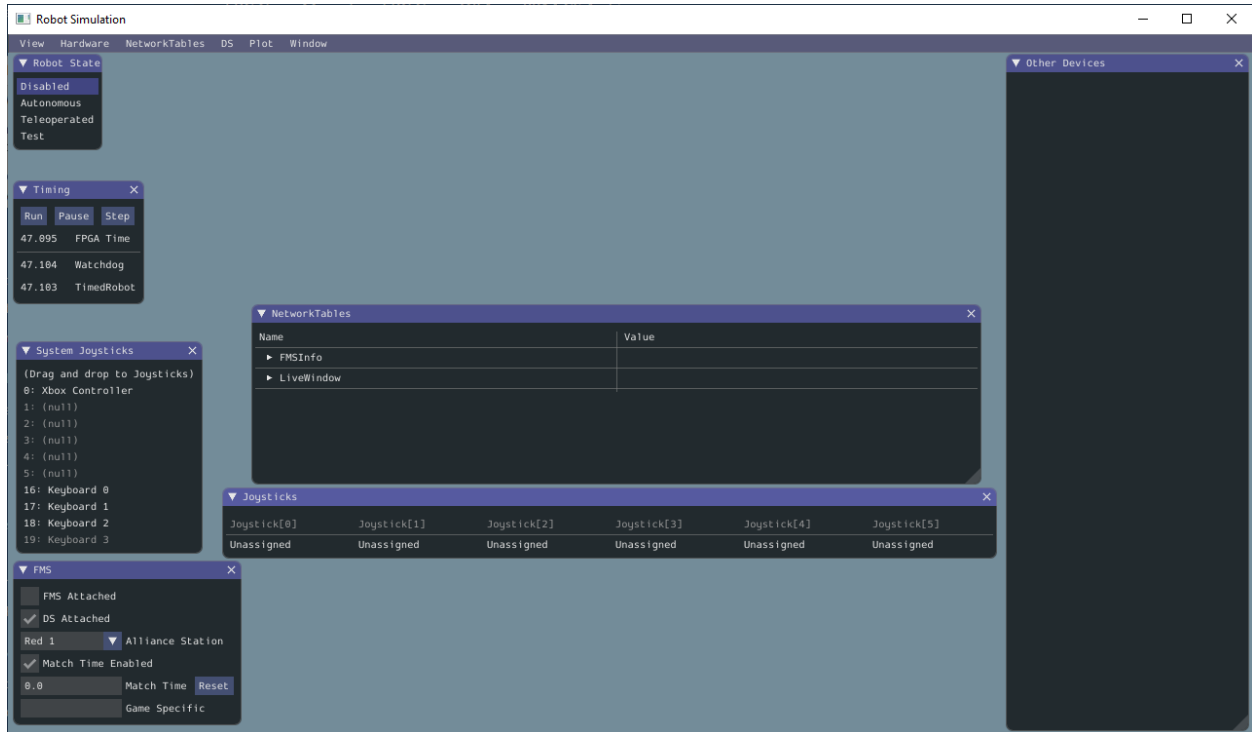
25.2.1 Running the GUI



You can simply launch the GUI via the **Run Simulation** command palette option.

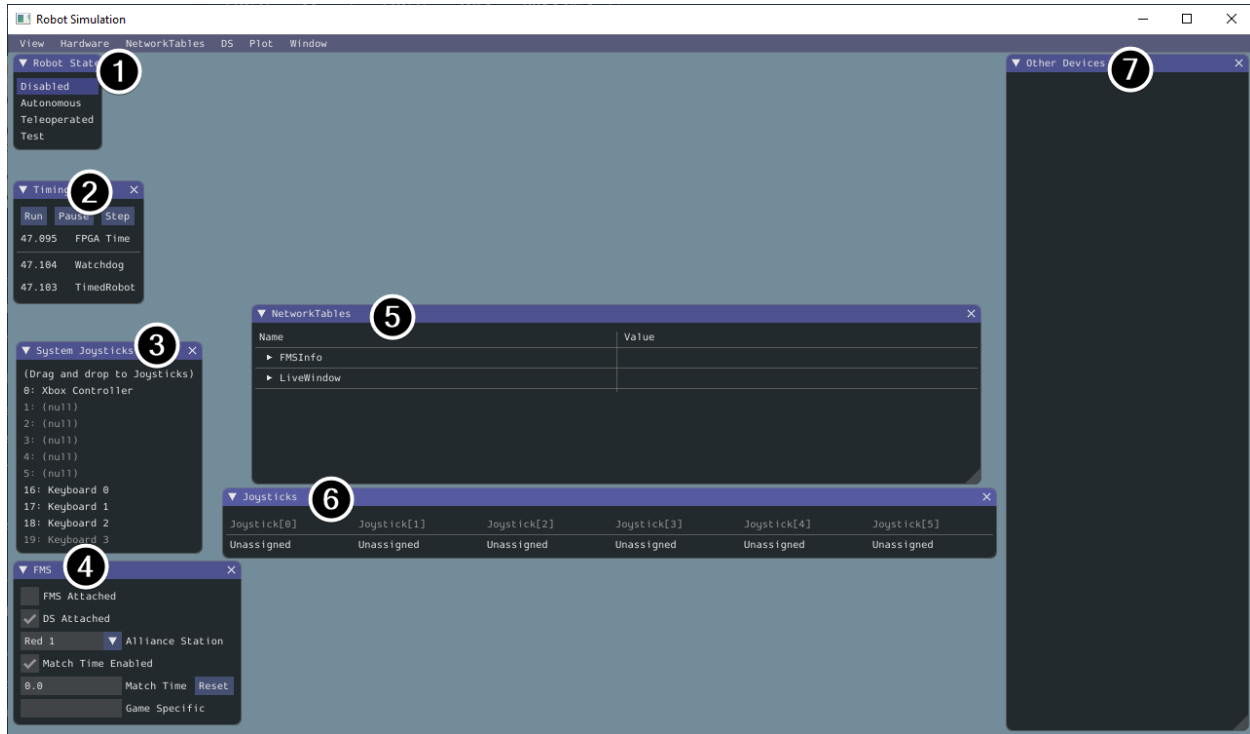


And the `halsim_gui.dll` option should popup in a new dialog (`halsim_gui.so` on Linux and `halsim_gui.dylib` on macOS). Select this and press **Ok**. This will now launch the Simulation GUI!



25.2.2 Using the GUI

Learning the Layout



The following items are shown on the simulation GUI by default:

1. **Robot State** - This is the robot's current state or "mode". You can click on the labels to change mode as you would on the normal Driver Station.
2. **Timing** - Shows the values of the Robot's timers and allows the timing to be manipulated.
3. **System Joysticks** - This is a list of joysticks connected to your system currently.
4. **FMS** - This is used for simulating many of the common FMS systems.
5. **NetworkTables** - This shows the data that has been published to NetworkTables.
6. **Joysticks** - This is joysticks that the robot code can directly pull from.
7. **Other Devices** - This includes devices that do not fall into any of the other categories, such as the ADXRS450 gyro that is included in the Kit of Parts or third party devices that support simulation.

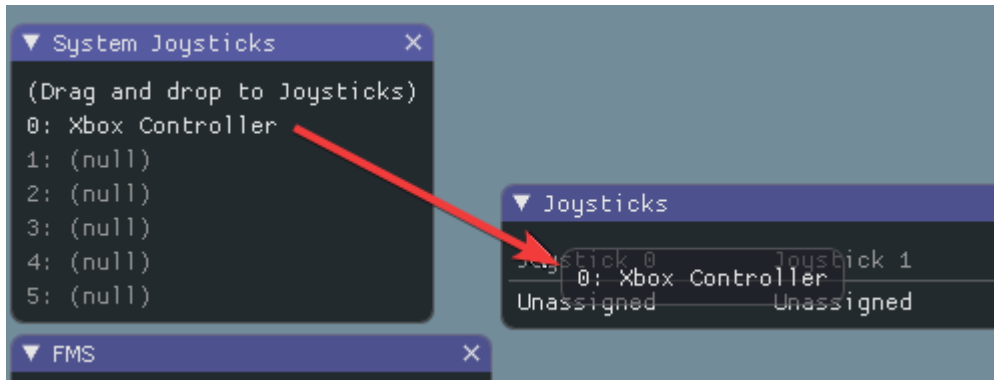
The following items can be added from the Hardware menu, but are not shown by default.

1. **Addressable LEDs** - This shows LEDs controlled by the AddressableLED Class.
2. **Analog Inputs** - This includes any devices that would normally use the **ANALOG IN** connector on the roboRIO, such as any Analog based gyros.
3. **DIO** - (Digital Input Output) This includes any devices that use the **DIO** connector on the roboRIO.
4. **Encoders** - This will show any instantiated devices that extend or use the Encoder class.
5. **PDPs** - This shows the Power Distribution Panel object.
6. **PWM Outputs** - This is a list of instantiated PWM devices. This will appear as many devices as you instantiate in robot code, as well as their outputs.

7. **Relays** - This includes any relay devices. This includes VEX Spike relays.
8. **Solenoids** - This is a list of “connected” solenoids. When you create a solenoid object and push outputs, these are shown here.

Adding a System Joystick to Joysticks

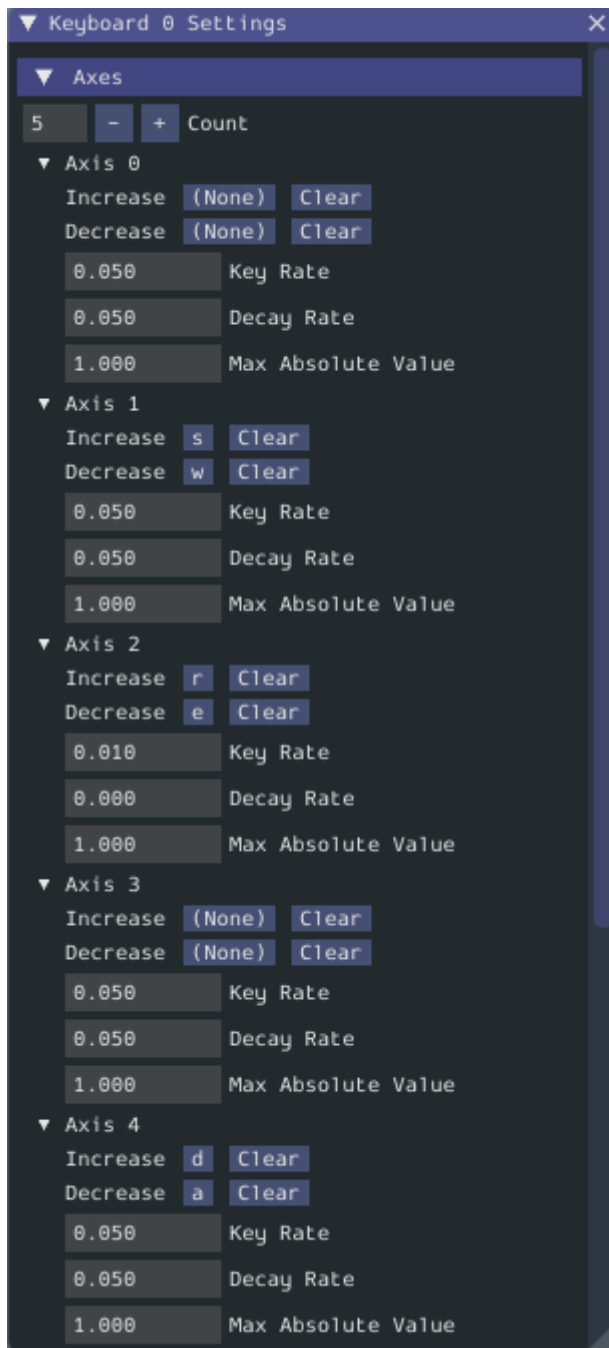
To add a joystick from the list of system joysticks, simply click and drag a shown joystick under the “System Joysticks” menu to the “Joysticks” menu”.



Note: The FRC® Driver Station does special mapping to gamepads connected and the WPILib simulator does not “map” these by default. You can turn on this behavior by pressing the “Map gamepad” toggle underneath the “Joysticks” menu.

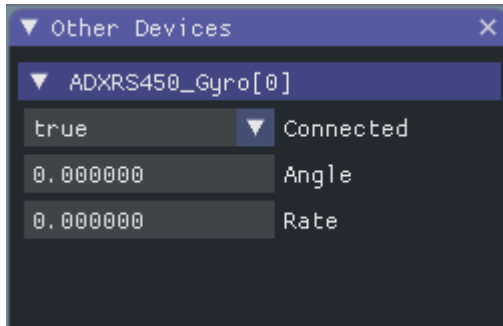
Using the Keyboard as a Joystick

You add a keyboard to the list of system joysticks by clicking and dragging one of the keyboard items (e.g. Keyboard 0) just like a joystick above. To edit the settings of the keyboard go to the *DS* item in the menu bar then choose *Keyboard 0 Settings*. This allows you to control which keyboard buttons control which axis. This is a common example of how to make the keyboard similar to a split sticks arcade drive on an Xbox controller (uses axis 1 & 4):



Modifying ADXRS450 Inputs

Using the ADXRS450 object is a fantastic way to test gyro based outputs. This will show up in the “Other Devices” menu. A drop down menu is then exposed that shows various options such as “Connected”, “Angle”, and “Rate”. All of these values are values that you can change, and that your robot code can use on-the-fly.



25.2.3 Determining Simulation from Robot Code

In cases where vendor libraries do not compile when running the robot simulation, you can wrap their content with `RobotBase.isReal()` which returns a boolean.

Java

```
TalonSRX motorLeft;
TalonSRX motorRight;

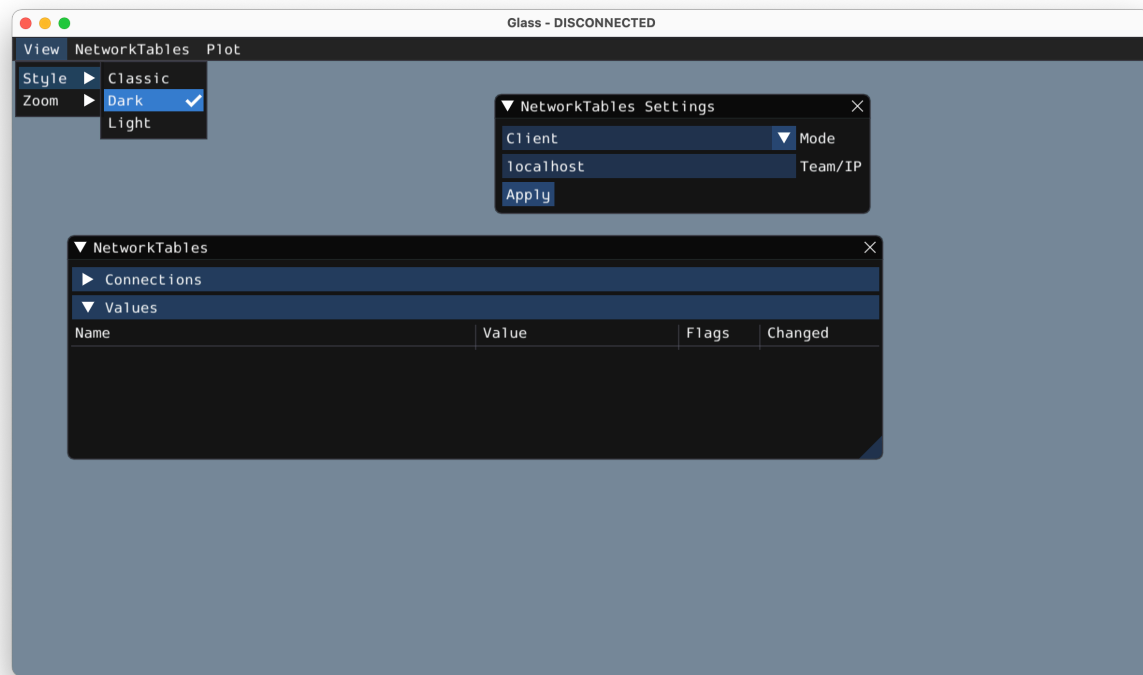
public Robot() {
    if (RobotBase.isReal()) {
        motorLeft = new TalonSRX(0);
        motorRight = new TalonSRX(1);
    }
}
```

Note: Reassigning value types in C++ requires move or copy assignment; vendors classes that both do not support the SIM and lack a move or copy assignment operator cannot be worked around with conditional allocation unless a pointer is used, instead of a value type.

25.2.4 Changing View Settings

The *View* menu item contains *Zoom* and *Style* settings that can be customized. The *Zoom* option dictates the size of the text in the application whereas the *Style* option allows you to select between the Classic, Light, and Dark modes.

An example of the Dark style setting is below:



Note: In Simulation GUI v2021.2.1 and below, the default zoom setting of 100% may cause text to appear too big on certain macOS Retina displays. Please reduce the zoom level to 75% or 50% or upgrade to v2021.2.2 or later to mitigate this issue.

25.2.5 Clearing Application Data

Application data for the Simulation GUI, including widget sizes and positions as well as other custom information for widgets is stored in a `imgui.ini` file. This file is stored in the root of the project directory that the simulation is run from.

The `imgui.ini` configuration file can simply be deleted to restore the Simulation GUI to a “clean slate”.

25.3 Physics Simulation with WPILib

Because *state-space notation* allows us to compactly represent the *dynamics of systems*, we can leverage it to provide a backend for simulating physical systems on robots. The goal of these simulators is to simulate the motion of robot mechanisms without modifying existing non-simulation user code. The basic flow of such simulators is as follows:

- In normal user code:
 - PID or similar control algorithms generate voltage commands from encoder (or other sensor) readings
 - Motor outputs are set

- In simulation periodic code:
 - The simulation's *state* is updated using *inputs*, usually voltages from motors set from a PID loop
 - Simulated encoder (or other sensor) readings are set for user code to use in the next timestep

25.3.1 WPILib's Simulation Classes

The following physics simulation classes are available in WPILib:

- LinearSystemSim, for modeling systems with linear dynamics
- FlywheelSim
- DifferentialDrivetrainSim
- ElevatorSim, which models gravity
- SingleJointedArmSim, which models gravity
- BatterySim, which simply estimates battery voltage sag based on drawn currents

All simulation classes (with the exception of the differential drive simulator) inherit from the LinearSystemSim class. By default, the dynamics are the linear system dynamics $\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k$. Subclasses override the `UpdateX(x, u, dt)` method to provide custom, nonlinear dynamics, such as modeling gravity.

25.3.2 Usage in User Code

The following is available from the WPILib `elevatorsimulation` [example project](#).

In addition to standard objects such as motors and encoders, we instantiate our elevator simulator using known constants such as carriage mass and gearing reduction. We also instantiate an `EncoderSim`, which sets the distance and rate read by our `Encoder`.

In the following example, we simulate an elevator given the mass of the moving carriage (in kilograms), the radius of the drum driving the elevator (in meters), the gearing reduction between motor and drum as output over input (so usually greater than one), the minimum and maximum height of the elevator (in meters), and some random noise to add to our position estimate.

Note: The elevator and arm simulators will prevent the simulated position from exceeding given minimum or maximum heights or angles. If you wish to simulate a mechanism with infinite rotation or motion, `LinearSystemSim` may be a better option.

Java

C++

```
36 private final ElevatorSim m_elevatorSim = new ElevatorSim(m_elevatorGearbox,  
37     kElevatorGearing,  
38     kCarriageMass,  
39     kElevatorDrumRadius,  
40     kMinElevatorHeight,
```

(continues on next page)

(continued from previous page)

```

41     kMaxElevatorHeight,
42     VecBuilder.fill(0.01));
43     private final EncoderSim m_encoderSim = new EncoderSim(m_encoder);

36     frc::sim::ElevatorSim m_elevatorSim{m_elevatorGearbox,
37                                         kElevatorGearing,
38                                         kCarriageMass,
39                                         kElevatorDrumRadius,
40                                         kMinElevatorHeight,
41                                         kMaxElevatorHeight,
42                                         {0.01}};
43     frc::sim::EncoderSim m_encoderSim{m_encoder};

```

Next, teleopPeriodic/TeleopPeriodic (Java/C++) uses a simple PID control loop to drive our elevator to a setpoint 30 inches off the ground.

Java

C++

```

36     @Override
37     public void teleopPeriodic() {
38         if (m_joystick.getTrigger()) {
39             // Here, we run PID control like normal, with a constant setpoint of 30in.
40             double pidOutput = m_controller.calculate(m_encoder.getDistance(), Units.
41             ↪ inchesToMeters(30));
42             m_motor.setVoltage(pidOutput);
43         } else {
44             // Otherwise, we disable the motor.
45             m_motor.set(0.0);
46         }
47     }

```

```

87     void TeleopPeriodic() {
88         if (m_joystick.GetTrigger()) {
89             // Here, we run PID control like normal, with a constant setpoint of 30in.
90             double pidOutput =
91                 m_controller.Calculate(m_encoder.GetDistance(), (30_in).to<double>());
92             m_motor.SetVoltage(units::volt_t(pidOutput));
93         } else {
94             // Otherwise, we disable the motor.
95             m_motor.Set(0.0);
96         }
97     }

```

Next, simulationPeriodic/SimulationPeriodic (Java/C++) uses the voltage applied to the motor to update the simulated position of the elevator. We use SimulationPeriodic because it runs periodically only for simulated robots. This means that our simulation code will not be run on a real robot.

Finally, the simulated encoder's distance reading is set using the simulated elevator's position, and the robot's battery voltage is set using the estimated current drawn by the elevator.

Java

C++

```

68 @Override
69 public void simulationPeriodic() {
70     // In this method, we update our simulation of what our elevator is doing
71     // First, we set our "inputs" (voltages)
72     m_elevatorSim.setInput(m_motor.get() * RobotController.getBatteryVoltage());
73
74     // Next, we update it. The standard loop time is 20ms.
75     m_elevatorSim.update(0.020);
76
77     // Finally, we set our simulated encoder's readings and simulated battery voltage
78     m_encoderSim.setDistance(m_elevatorSim.getPositionMeters());
79     // SimBattery estimates loaded battery voltages
80     RoboRioSim.setVInVoltage(BatterySim.calculateDefaultBatteryLoadedVoltage(m_
81     ↪ elevatorSim.getCurrentDrawAmps()));
82 }

```

```

69 void SimulationPeriodic() {
70     // In this method, we update our simulation of what our elevator is doing
71     // First, we set our "inputs" (voltages)
72     m_elevatorSim.SetInput(frc::MakeMatrix<1, 1>(
73         m_motor.Get() * frc::RobotController::GetInputVoltage()));
74
75     // Next, we update it. The standard loop time is 20ms.
76     m_elevatorSim.Update(20_ms);
77
78     // Finally, we set our simulated encoder's readings and simulated battery
79     // voltage
80     m_encoderSim.SetDistance(m_elevatorSim.GetPosition().to<double>());
81     // SimBattery estimates loaded battery voltages
82     frc::sim::RoboRioSim::SetVInVoltage(
83         frc::sim::BatterySim::Calculate({m_elevatorSim.GetCurrentDraw()}));
84 }

```

25.4 Device Simulation

WPILib provides a way to manage simulation device data in the form of the SimDevice API.

25.4.1 Simulating Core WPILib Device Classes

Core WPILib device classes (i.e Encoder, Ultrasonic, etc.) have simulation classes named EncoderSim, UltrasonicSim, and so on. These classes allow interactions with the device data that wouldn't be possible or valid outside of simulation. Constructing them outside of simulation likely won't interfere with your code, but calling their functions and the like is undefined behavior - in the best case they will do nothing, worse cases might crash your code! Place functional simulation code in simulation-only functions (such as simulationPeriodic()) or wrap them with RobotBase.isReal()/ RobotBase::IsReal() checks (which are constexpr in C++).

Note: This example will use the EncoderSim class as an example. Use of other simulation classes will be almost identical.

Creating Simulation Device objects

Simulation device object can be constructed in two ways:

- a constructor that accepts the regular hardware object.
- a constructor or factory method that accepts the port/index/channel number that the device is connected to. These would be the same number that was used to construct the regular hardware object. This is especially useful for *unit testing*.

Java

C++

```
// create a real encoder object on DIO 2,3
Encoder encoder = new Encoder(2, 3);
// create a sim controller for the encoder
EncoderSim simEncoder = new EncoderSim(encoder);
```

```
// create a real encoder object on DIO 2,3
frc::Encoder encoder{2, 3};
// create a sim controller for the encoder
frc::sim::EncoderSim simEncoder{encoder};
```

Reading and Writing Device Data

Each simulation class has getter (getXxx()/GetXxx()) and setter (setXxx(value)/SetXxx(value)) functions for each field Xxx. The getter functions will return the same as the getter of the regular device class.

Java

C++

```
simEncoder.setCount(100);
encoder.getCount(); // 100
simEncoder.getCount(); // 100
```

```
simEncoder.SetCount(100);
encoder.GetCount(); // 100
simEncoder.GetCount(); // 100
```

Registering Callbacks

In addition to the getters and setters, each field also has a registerXxxCallback() function that registers a callback to be run whenever the field value changes and returns a CallbackStore object. The callbacks accept a string parameter of the name of the field and a HALValue object containing the new value. Before retrieving values from a HALValue, check the type of value contained. Possible types are HALValue.kBoolean/HAL_BOOL, HALValue.kDouble/HAL_DOUBLE, HALValue.kEnum/HAL_ENUM, HALValue.kInt/HAL_INT, HALValue.kLong/HAL_LONG.

In Java, call close() on the CallbackStore object to cancel the callback. Keep a reference to the object so it doesn't get garbage-collected - otherwise the callback will be canceled by GC. To provide arbitrary data to the callback, capture it in the lambda or use a method reference.

In C++, save the CallbackStore object in the right scope - the callback will be canceled when the object goes out of scope and is destroyed. Arbitrary data can be passed to the callbacks via the param parameter.

Warning: Attempting to retrieve a value of a type from a HALValue containing a different type is undefined behavior.

Java

C++

```
NotifyCallback callback = (String name, HALValue value) -> {
    if (value.getType() == HALValue.kInt) {
        System.out.println("Value of " + name + " is " + value.getInt());
    }
}
CallbackStore store = simEncoder.registerCountCallback(callback);

store.close(); // cancel the callback
```

```
HAL_NotifyCallback callback = [](const char* name, void* param, const HALValue*
↪value) {
    if (value->type == HAL_INT) {
        wpi::outs() << "Value of " << name << " is " << value->data.v_int << '\n';
    }
};
frc::sim::CallbackStore store = simEncoder.RegisterCountCallback(callback);
// the callback will be canceled when ``store`` goes out of scope
```

25.4.2 Simulating Other Devices - The SimDeviceSim Class

Note: Vendors might implement their connection to the SimDevice API slightly different than described here. They might also provide a simulation class specific for their device class. See your vendor's documentation for more information as to what they support and how.

The SimDeviceSim (**not** ``SimDevice``!) class is a general device simulation object for devices that aren't core WPILib devices and therefore don't have specific simulation classes - such as vendor devices. These devices will show up in the *Other Devices* tab of the *SimGUI*.

The SimDeviceSim object is created using a string key identical to the key the vendor used to construct the underlying SimDevice in their device class. This key is the one that the device shows up with in the *Other Devices* tab, and is typically of the form Prefix:Device Name[index]. If the key contains ports/index/channel numbers, they can be passed as separate arguments to the SimDeviceSim constructor. The key contains a prefix that is hidden by default in the SimGUI, it can be shown by selecting the *Show prefix* option. Not including this prefix in the key passed to SimDeviceSim will not match the device!

Java

C++

```
SimDeviceSim device = new SimDeviceSim(deviceKey, index);
```

```
frc::sim::SimDeviceSim device{deviceKey, index};
```

Once we have the `SimDeviceSim`, we can get `SimValue` objects representing the device's fields. Type-specific `SimDouble`, `SimInt`, `SimLong`, `SimBoolean`, and `SimEnum` subclasses also exist, and should be used instead of the type-unsafe `SimValue` class. These are constructed from the `SimDeviceSim` using a string key identical to the one the vendor used to define the field. This key is the one the field appears as in the `SimGUI`. Attempting to retrieve a `SimValue` object outside of simulation or when either the device or field keys are unmatched will return `null` - this can cause `NullPointerException` in Java or undefined behavior in C++.

Java

C++

```
SimDouble field = device.getDouble(fieldKey);
field.get();
field.set(value);
```

```
hal::SimDouble field = device.GetDouble(fieldKey);
field.Get();
field.Set(value);
```

25.5 Unit Testing

Unit testing is a method of testing code by dividing the code into the smallest “units” possible and testing each unit. In robot code, this can mean testing the code for each subsystem individually. There are many unit testing frameworks for most languages. Java robot projects have [JUnit 4](#) available by default, and C++ robot projects have [Google Test](#).

25.5.1 Writing Testable Code

Note: This example can be easily adapted to the command-based paradigm by having `Intake` inherit from `SubsystemBase`.

Our subsystem will be an Infinite Recharge intake mechanism containing a piston and a motor: the piston deploys/retracts the intake, and the motor will pull the Power Cells inside. We don't want the motor to run if the intake mechanism isn't deployed because it won't do anything.

To provide a “clean slate” for each test, we need to have a function to destroy the object and free all hardware allocations. In Java, this is done by implementing the `AutoCloseable` interface and its `.close()` method, destroying each member object by calling the member's `.close()` method - an object without a `.close()` method probably doesn't need to be closed. In C++, the default destructor will be called automatically when the object goes out of scope and will call destructors of member objects.

Note: Vendors might not support resource closing identically to the way shown here. See your vendor's documentation for more information as to what they support and how.

Java

C++ (Header)

C++ (Source)

```
import edu.wpi.first.wpilibj.DoubleSolenoid;
import edu.wpi.first.wpilibj.PWMSparkMax;
import frc.robot.Constants.IntakeConstants;

public class Intake implements AutoCloseable {
    private PWMSparkMax motor;
    private DoubleSolenoid piston;

    public Intake() {
        motor = new PWMSparkMax(IntakeConstants.MOTOR_PORT);
        piston = new DoubleSolenoid(IntakeConstants.PISTON_FWD, IntakeConstants.PISTON_
↵REV);
    }

    public void deploy() {
        piston.set(DoubleSolenoid.Value.kForward);
    }

    public void retract() {
        piston.set(DoubleSolenoid.Value.kReverse);
        motor.set(0); // turn off the motor
    }

    public void activate(double speed) {
        if (piston.get() == DoubleSolenoid.Value.kForward) {
            motor.set(speed);
        } else { // if piston isn't open, do nothing
            motor.set(0);
        }
    }

    @Override
    public void close() throws Exception {
        piston.close();
        motor.close();
    }
}
```

```
#include <frc2/command/SubsystemBase.h>
#include <frc/DoubleSolenoid.h>
#include <frc/PWMSparkMax.h>

#include "Constants.h"

class Intake : public frc2::SubsystemBase {
public:
    void Deploy();
    void Retract();
    void Activate(double speed);

private:
    frc::PWMSparkMax motor{Constants::Intake::MOTOR_PORT};
    frc::DoubleSolenoid piston{Constants::Intake::PISTON_FWD, Constants::Intake::PISTON_
↵REV};
}
```

(continues on next page)

(continued from previous page)

```
};

#include "subsystems/Intake.h"

void Intake::Deploy() {
    piston.Set(frc::DoubleSolenoid::Value::kForward);
}

void Intake::Retract() {
    piston.Set(frc::DoubleSolenoid::Value::kReverse);
    motor.Set(0); // turn off the motor
}

void Intake::Activate(double speed) {
    if (piston.Get() == frc::DoubleSolenoid::Value::kForward) {
        motor.Set(speed);
    } else { // if piston isn't open, do nothing
        motor.Set(0);
    }
}
```

25.5.2 Writing Tests

Important: Tests are placed inside the test source set: `/src/test/java/` and `/src/test/cpp/` for Java and C++ tests, respectively. Files outside that source root do not have access to the test framework - this will fail compilation due to unresolved references.

In Java, each test class contains at least one test method marked with `@org.junit.Test`, each method representing a test case. Additional methods for opening resources (such as our Intake object) before each test and closing them after are respectively marked with `@org.junit.Before` and `@org.junit.After`. In C++, test fixture classes inheriting from `testing::Test` contain our subsystem and simulation hardware objects, and test methods are written using the `TEST_F(testfixture, testname)` macro. The `SetUp()` and `TearDown()` methods can be overridden in the test fixture class and will be run respectively before and after each test.

Each test method should contain at least one *assertion* (`assert*()` in Java or `EXPECT_*()` in C++). These assertions verify a condition at runtime and fail the test if the condition isn't met. If there is more than one assertion in a test method, the first failed assertion will crash the test - execution won't reach the later assertions.

Both JUnit and GoogleTest have multiple assertion types, but the most common is equality: `assertEquals(expected, actual)/EXPECT_EQ(expected, actual)`. When comparing numbers, a third parameter - delta, the acceptable error, can be given. In JUnit (Java), these assertions are static methods and can be used without qualification by adding the static star `import static org.junit.Assert.*`. In Google Test (C++), assertions are macros from the `<gtest/gtest.h>` header.

Note: Comparison of floating-point values isn't accurate, so comparing them should be done with an acceptable error parameter (DELTA).

Java

C++

```

import static org.junit.Assert.*;

import edu.wpi.first.hal.HAL;
import edu.wpi.first.wpilibj.DoubleSolenoid;
import edu.wpi.first.wpilibj.simulation.DoubleSolenoidSim;
import edu.wpi.first.wpilibj.simulation.PWMSim;
import frc.robot.Constants.IntakeConstants;
import org.junit.*;

public class IntakeTest {
    public static final double DELTA = 1e-2; // acceptable deviation range
    Intake intake;
    PWMSim simMotor;
    DoubleSolenoidSim simPiston;

    @Before // this method will run before each test
    public void setup() {
        assert HAL.initialize(500, 0); // initialize the HAL, crash if failed
        intake = new Intake(); // create our intake
        simMotor = new PWMSim(IntakeConstants.MOTOR_PORT); // create our simulation PWM
        ↪ motor controller
        simPiston = new DoubleSolenoidSim(IntakeConstants.PISTON_FWD, IntakeConstants.
        ↪ PISTON_REV); // create our simulation solenoid
    }

    @After // this method will run after each test
    public void shutdown() throws Exception {
        intake.close(); // destroy our intake object
    }

    @Test // marks this method as a test
    public void doesntWorkWhenClosed() {
        intake.retract(); // close the intake
        intake.activate(0.5); // try to activate the motor
        assertEquals(0.0, simMotor.getSpeed(), DELTA); // make sure that the value set to
        ↪ the motor is 0
    }

    @Test
    public void worksWhenOpen() {
        intake.deploy();
        intake.activate(0.5);
        assertEquals(0.5, simMotor.getSpeed(), DELTA);
    }

    @Test
    public void retractTest() {
        intake.retract();
        assertEquals(DoubleSolenoid.Value.kReverse, simPiston.get());
    }

    @Test
    public void deployTest() {
        intake.deploy();
    }
}

```

(continues on next page)

(continued from previous page)

```

    assertEquals(DoubleSolenoid.Value.kForward, simPiston.get());
}
}

```

```

#include <gtest/gtest.h>

#include <frc/DoubleSolenoid.h>
#include <frc/simulation/DoubleSolenoidSim.h>
#include <frc/simulation/PWMSim.h>

#include "subsystems/Intake.h"
#include "Constants.h"

constexpr double DELTA = 1e-2; // acceptable deviation range

class IntakeTest : public testing::Test {
protected:
    Intake intake; // create our intake
    frc::sim::PWMSim simMotor{Constants::Intake::MOTOR_PORT}; // create our simulation_
    frc::sim::DoubleSolenoidSim simPiston{Constants::Intake::PISTON_FWD,
    Constants::Intake::PISTON_REV}; // create our simulation solenoid
};

TEST_F(IntakeTest, DoesntWorkWhenClosed) {
    intake.Retract(); // close the intake
    intake.Activate(0.5); // try to activate the motor
    EXPECT_EQ(0.0, simMotor.GetSpeed(), DELTA); // make sure that the value set to the_
    motor is 0
}

TEST_F(IntakeTest, WorksWhenOpen) {
    intake.Deploy();
    intake.Activate(0.5);
    EXPECT_EQ(0.5, simMotor.GetSpeed(), DELTA);
}

TEST_F(IntakeTest, RetractTest) {
    intake.Retract();
    EXPECT_EQ(frc::DoubleSolenoid::Value::kReverse, simPiston.Get());
}

TEST_F(IntakeTest, DeployTest) {
    intake.Deploy();
    EXPECT_EQ(frc::DoubleSolenoid::Value::kForward, simPiston.Get());
}

```

For more advanced usage of JUnit and Google Test, see the framework docs.

25.5.3 Running Tests

Note: Tests will always be run in simulation on your desktop. For prerequisites and more info, see [the simulation introduction](#).

For Java tests to run, make sure that your `build.gradle` file contains the following block:

```
test {  
    useJUnit()  
}
```

Use *Test Robot Code* from the Command Palette to run the tests. Results will be reported in the terminal output, each test will have a **FAILED** or **PASSED/OK** label next to the test name in the output. JUnit (Java only) will generate a HTML document in `build/reports/tests/test/index.html` with a more detailed overview of the results; if there are failed test a link to render the document in your browser will be printed in the terminal output.

By default, Gradle runs the tests whenever robot code is built, including deploys. This will increase deploy time, and failing tests will cause the build and deploy to fail. To prevent this from happening, you can use *Change Skip Tests On Deploy Setting* from the Command Palette to configure whether to run tests when deploying.

Robot Characterization

26.1 Introduction to Robot Characterization

The characterization tools consist of a python application that runs on the user's PC and matching robot code that runs on the user's robot. The PC application will send control signals to the robot over *NetworkTables*, while the robot sends data back to the application. The application then processes the data and determines characterization parameters for the user's robot mechanism, as well as producing diagnostic plots. Data can be saved (in JSON format) for future use, if desired.

26.1.1 What is “Characterization?”

“Characterization” - or, more formally, *system identification* - is the process of determining a mathematical model for the behavior of a system through statistical analysis of its inputs and outputs.

In FRC, the most common system that we're interested in characterizing is the *permanent-magnet DC motor*. In particular, we're interested in figuring out which motor *input* (i.e. voltage from the motor controller) is required to achieve our desired *outputs* (i.e. velocity and acceleration of the motor).

Fortunately, it is not so difficult to do this. A permanent-magnet DC motor (with no load other than friction and inertia) will obey the following “voltage-balance equation” (for more information, see [this paper](#)):

$$V = kS \cdot \text{sgn}(\dot{d}) + kV \cdot \dot{d} + kA \cdot \ddot{d}$$

where V is the applied voltage, d is the displacement (position) of the motor, \dot{d} is its velocity, and \ddot{d} is its acceleration (the “overdot” notation traditionally denotes the *derivative* with respect to time).

Heuristically, we can interpret the coefficients in the above equation as follows:

kS is the voltage needed to overcome the motor's static friction, or in other words to just barely get it moving; it turns out that this static friction (because it's, well, static) has the same effect regardless of velocity or acceleration. That is, no matter what speed you're going or how fast you're accelerating, some constant portion of the voltage you've applied to your motor (depending on the specific mechanism assembly) will be going towards overcoming

the static friction in your gears, bearings, etc; this value is your kS . Note the presence of the **signum function**, because friction force always opposes the direction-of-motion.

kV describes how much voltage is needed to hold (or “cruise”) at a given constant velocity while overcoming the **electromagnetic resistance in the motor** and any additional friction that increases with speed (known as **viscous drag**). The relationship between speed and voltage (at constant acceleration) is almost entirely linear (with FRC® components, anyway) because of how permanent-magnet DC motors work.

kA describes the voltage needed to induce a given acceleration in the motor shaft. As with kV , the relationship between voltage and acceleration (at constant velocity) is almost perfectly linear for FRC components.

Once these coefficients have been determined (here accomplished by a **multiple linear regression**), we can then take a given desired velocity and acceleration for the motor and calculate the voltage that should be applied to achieve it. This is very useful - not only for, say, following motion profiles, but also for making mechanisms more controllable in open-loop control, because your joystick inputs will more closely match the actual mechanism motion.

Some of the tools in this toolsuite introduce additional terms into the above equation to account for known differences from the simple case described above - details for each tool can be found below:

26.1.2 Included Characterization Tools

Note: Many other types of mechanisms can be characterized by simply adapting the existing code in this library.

The robot characterization toolsuite currently supports characterization for:

- Simple Motor Setups
- Drivetrains
- Arms
- Elevators

Simple Motor Characterization

The simple motor characterization tool determines the best-fit parameters for the equation:

$$V = kS \cdot \text{sgn}(\dot{d}) + kV \cdot \dot{d} + kA \cdot \ddot{d}$$

where V is the applied voltage, d is the displacement (position) of the drive, \dot{d} is its velocity, and \ddot{d} is its acceleration. This is the model for a permanent-magnet dc motor with no loading other than friction and inertia, as mentioned above, and is an accurate model for flywheels, turrets, and horizontal linear sliders.

Drivetrain Characterization

The drivetrain characterization tool determines the best-fit parameters for the equation:

$$V = kS \cdot \text{sgn}(\dot{d}) + kV \cdot \dot{d} + kA \cdot \ddot{d}$$

where V is the applied voltage, d is the displacement (position) of the drive, \dot{d} is its velocity, and \ddot{d} is its acceleration. This is the same modeling equation as is used in the simple motor characterization - however, the drivetrain characterizer is specifically set up to run on differential drives, and will characterize each side of the drive independently if desired.

The drivetrain characterizer can also determine the effective trackwidth of your robot using a gyro. More information on how to run the characterization is available in the [track width characterization](#) article.

Arm Characterization

The arm characterization tool determines the best-fit parameters for the equation:

$$V = kS \cdot \text{sgn}(\dot{\theta}) + kCos \cdot \cos(\theta) + kV \cdot \dot{\theta} + kA \cdot \ddot{\theta}$$

where V is the applied voltage, θ is the angular displacement (position) of the arm, $\dot{\theta}$ is its angular velocity, and $\ddot{\theta}$ is its angular acceleration. The cosine term ($kCos$) is added to correctly account for the effect of gravity.

Elevator Characterization

The elevator characterization tool determines the best-fit parameters for the equation:

$$V = kG + kS \cdot \text{sgn}(\dot{d}) + kV \cdot \dot{d} + kA \cdot \ddot{d}$$

where V is the applied voltage, d is the displacement (position) of the drive, \dot{d} is its velocity, and \ddot{d} is its acceleration. The constant term (kG) is added to correctly account for the effect of gravity.

26.1.3 Prerequisites

To use the Robot Characterization Toolsuite, you must have Python 3.7 installed on your computer, as well as the standard WPILib programming toolsuite.

[Python 3.7](#)

Warning: Do not install Python from the Microsoft Store. Please use the link above to download and install Python.

26.1.4 Installing and Launching the Toolsuite

To install the Robot Characterization Toolsuite, open a console and enter the following command

```
pip install frc-characterization
```

The toolsuite, and all of its dependencies, should be automatically downloaded and installed. If you are using a Windows machine and the command `pip` is not recognized, ensure that your python scripts folder [has been added to the PATH](#).

Note: If you are on Ubuntu, you will have to manually install `tkinter` with `sudo apt-get install python3-tk`. You will also have to use the `pip3` command instead of `pip` as `pip` refers to Python 2 on Ubuntu distributions.

If you already have the toolsuite installed, be sure to update it regularly to benefit from bug-fixes and new features additions:

```
pip install --upgrade frc-characterization
```

Note: If you would like to use the beta version of this tool, you must type `pip install --pre --upgrade frc-characterization` instead.

Once the toolsuite has been installed, launch a new drive characterization project to ensure that it works by running the following command from powershell or a terminal window.

```
frc-characterization drive new
```

The new project GUI should open momentarily. To launch other characterization projects, simply replace `drive` with the desired characterization type (`arm`, `elevator`, `simple-motor`).

While the new project GUI has buttons for launching both the logging tool and the analyzer tool, these can also be launched directly from the CLI by replacing `new` with `logger` or `analyzer`.

Important: It is highly recommended that you utilize the new project GUI to launch the logger and analyzer tools for more effective unit conversions rather than launching the logger and analyzer from the CLI.

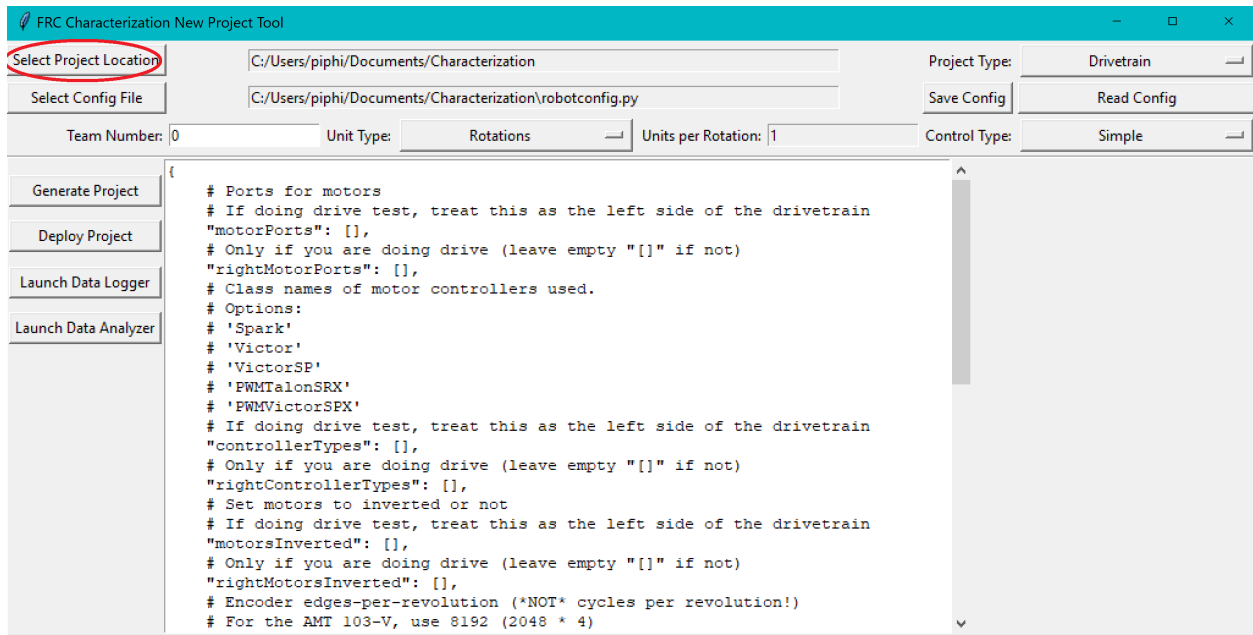
For more information on CLI usage, enter `frc-characterization -h`.

26.2 Generating a Project

To use the toolsuite, we first need to generate a robot project.

26.2.1 Select Project Location

First, select the desired project location on the new project GUI:

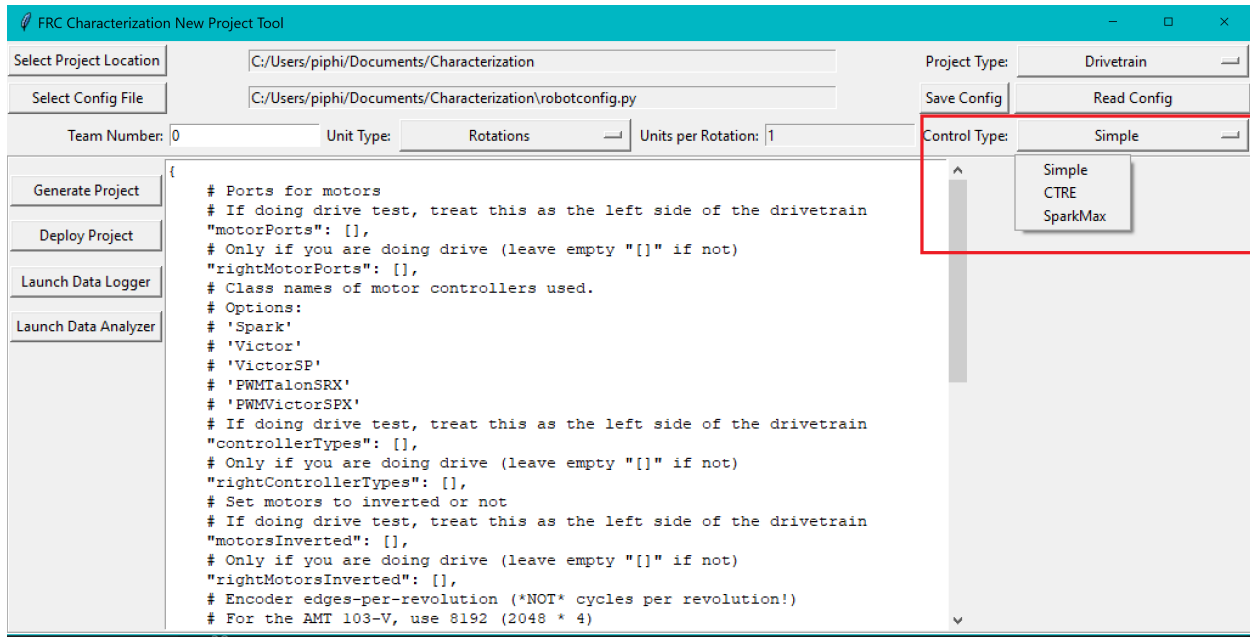


Note: The project type dropdown now chooses between the different types of characterization tests as the previous motor setups have been integrated into the *Control Type* field.

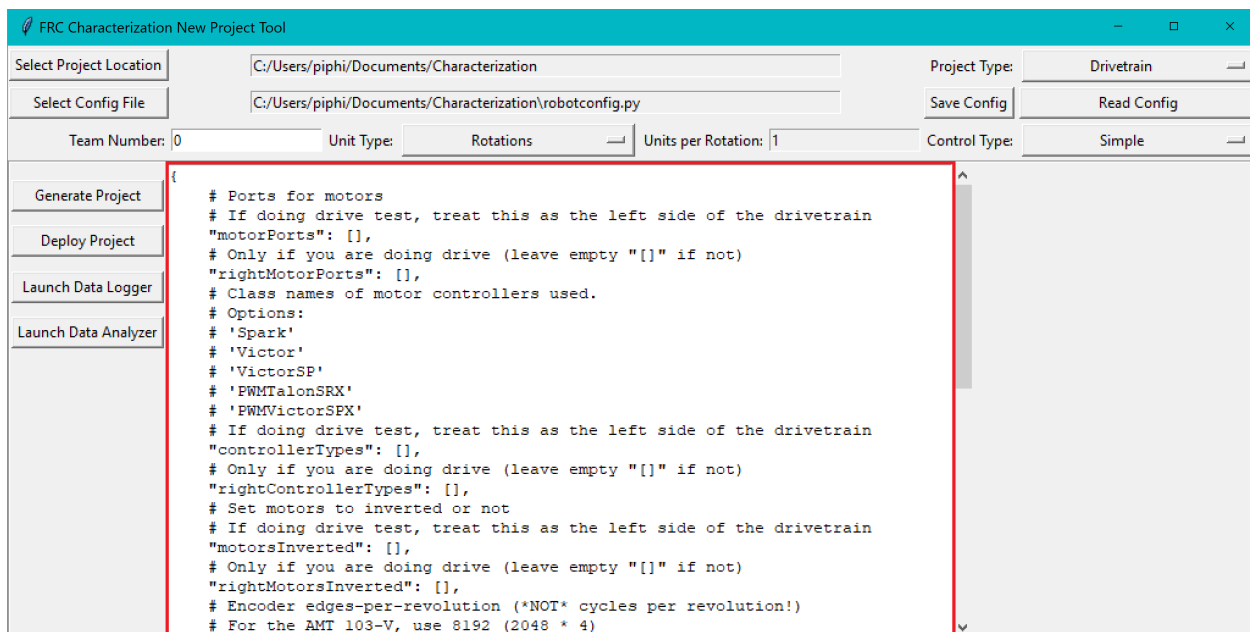
26.2.2 Configure Project Parameters

In order to run on your robot, the tool must know some parameters about how your robot is set up.

First, you need to use the *Control Type* field to select the appropriate project config template. Simple is for PWM Based motor controllers, CTRE is for CAN connected CTRE Motor Controllers (e.g. Talon SRX), and SparkMax is for the Spark Max Motor Controller. This allows you to fill out the parameters specific to the type of controllers you are using.



Project config settings are formatted as a [Python dictionary literal](#). These can be modified via the in-window config editor:

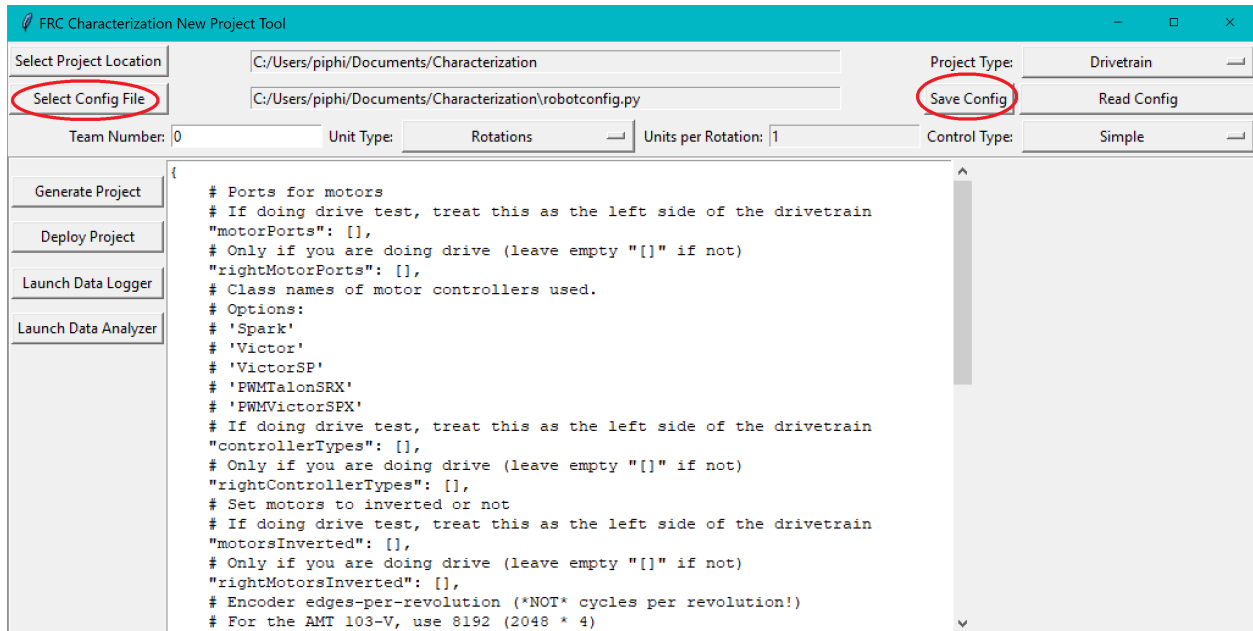


Take care of the following caveats when entering your robot specifications:

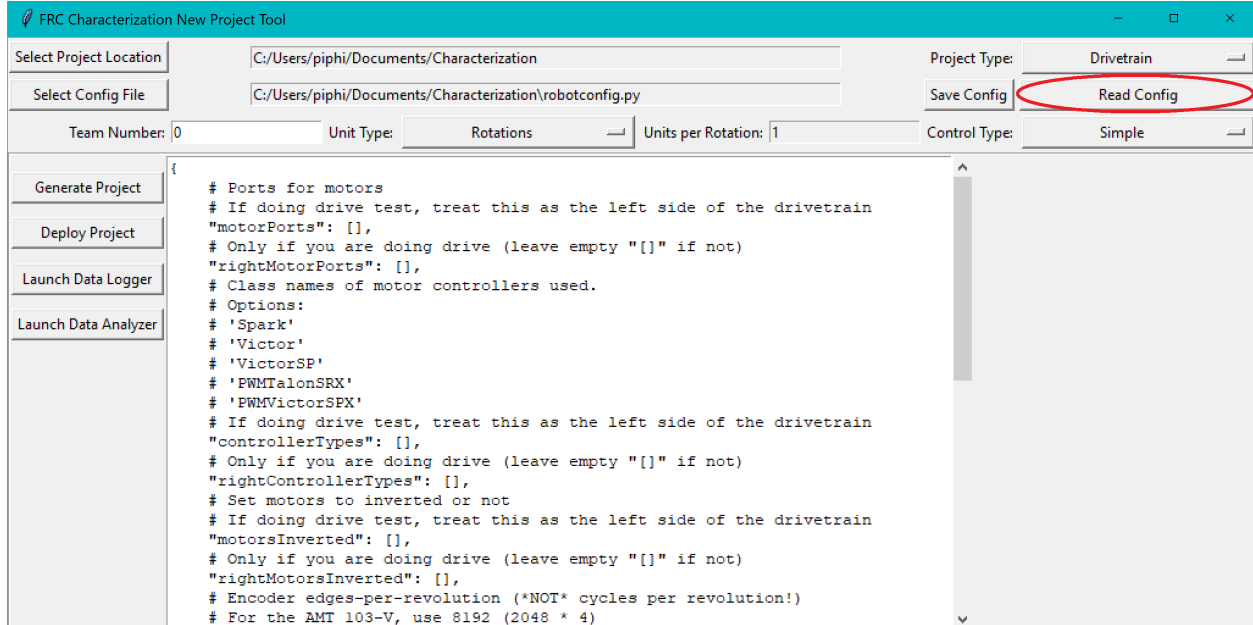
- The key names must *not be changed*, as they are hard-coded for each project type. Only the values (i.e. the things on the right-hand side of the colons) should be modified.
- True and False *must* be capitalized, as they are evaluated as native Python.
- All string values (e.g. controller names and unit types) *must* be wrapped in quotes and *must* correspond exactly to one of the options described.

Important: Read the comments provided in the config file carefully.

Once your robot configuration is set, you may save it to a location/name of your choice:



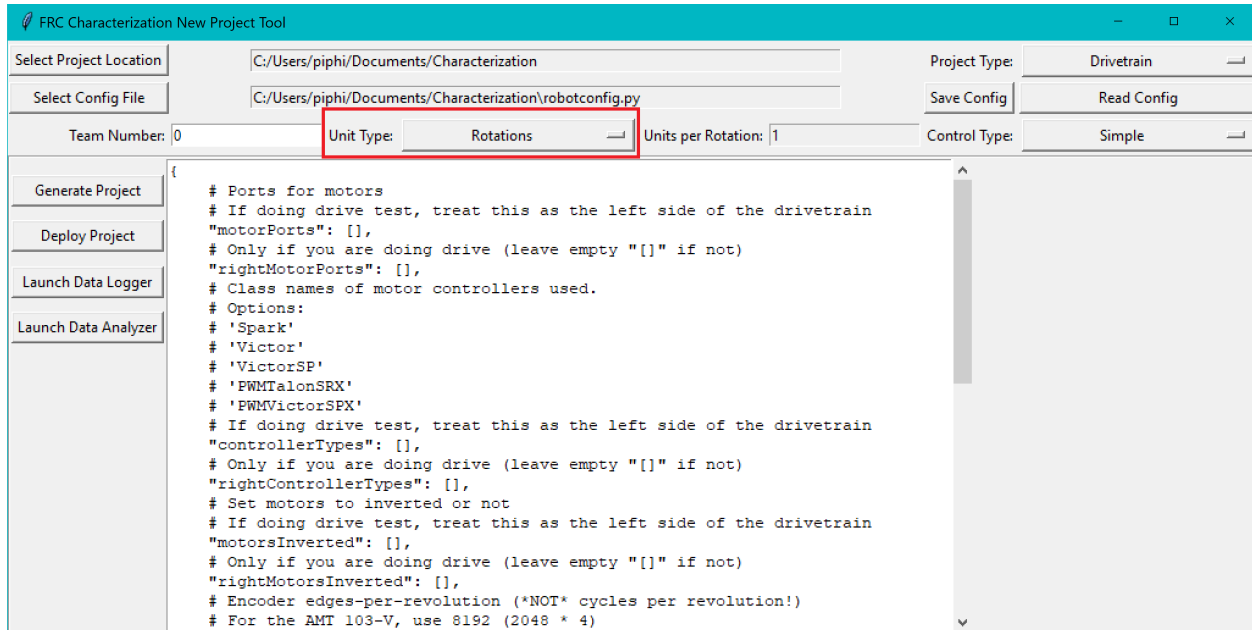
Accordingly, you can also load an existing config file (config files are project-type-specific):



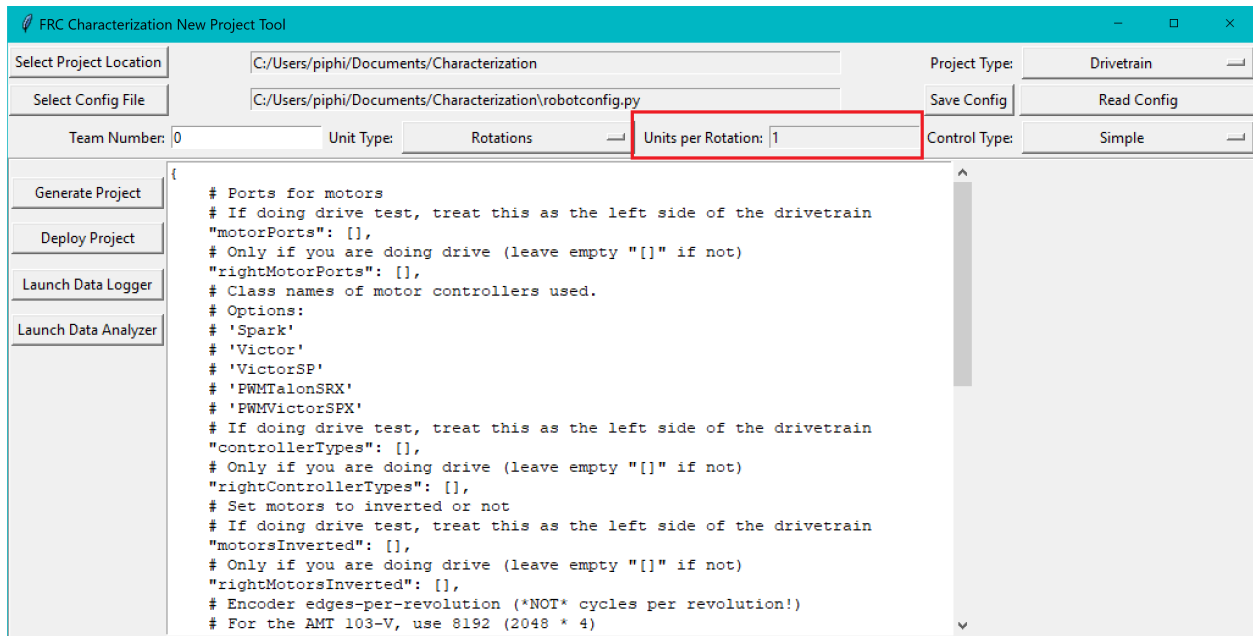
26.2.3 Setting Units

Now is also a good time to set the Team Number box and modify the units and units per rotation if necessary (units per rotation is NOT equivalent to the wheel diameter from last year's tool).

The *Unit Type* field lets you choose between various rotational and translational units (rotations, radians, degrees, feet, meters, and inches). You should choose a unit that facilitates a rough validation of the recorded measurements. For example, you could choose rotations when testing a flywheel to see if the tool's recorded rotations seem reasonable, or you could choose feet when testing a drive base to see if the recorded distance seems reasonable.

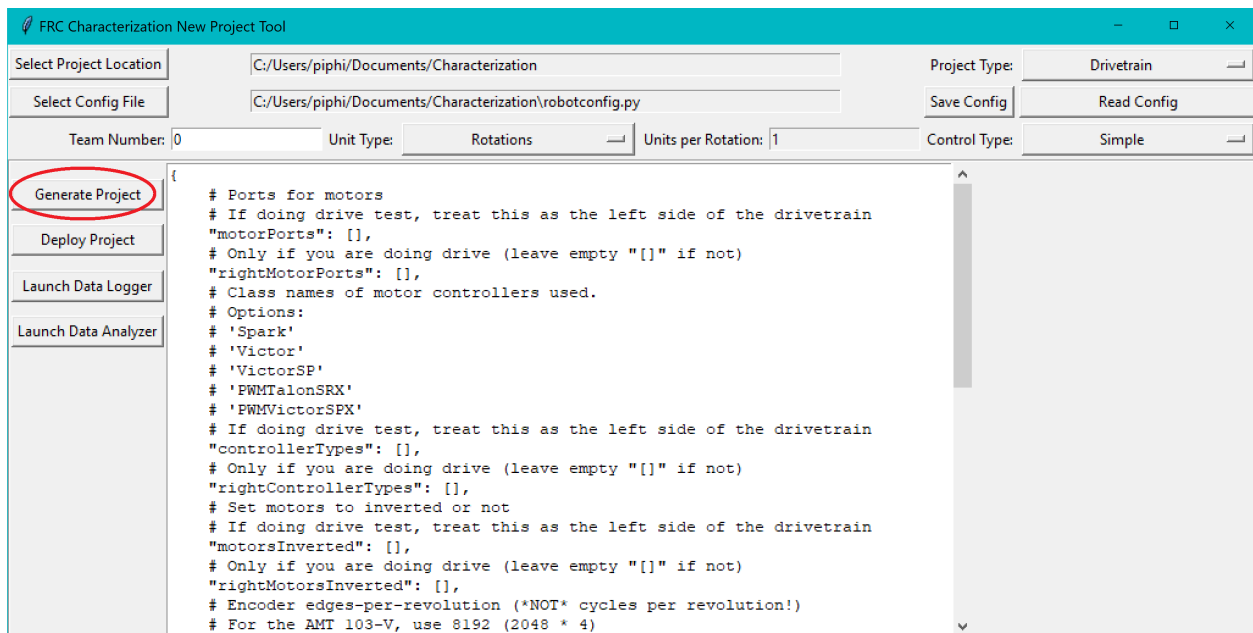


On the other hand, the *Units per Rotation* entry is how many of the previously specified units are recorded per rotation of the shaft. Do note that rotational units (rotations, radians, and degrees) have predefined/unmutable units per rotations. In contrast, translational units (meters, feet, inches) require that you specify the conversion, such as a wheel with a 3-inch diameter can be converted to 9.42 inches per rotation ($\pi * 3$).



26.2.4 Generate Project

Once your project has been configured, it's time to generate a deployable robot project to run the characterization:



A generated robot project will be placed in a subfolder (named characterization-project) of your specified project location.

The generated robot code will be in Java, and will reflect the settings specified in your config file. Advanced users are free to modify the generated code to their liking, if the existing configuration options do not suffice.

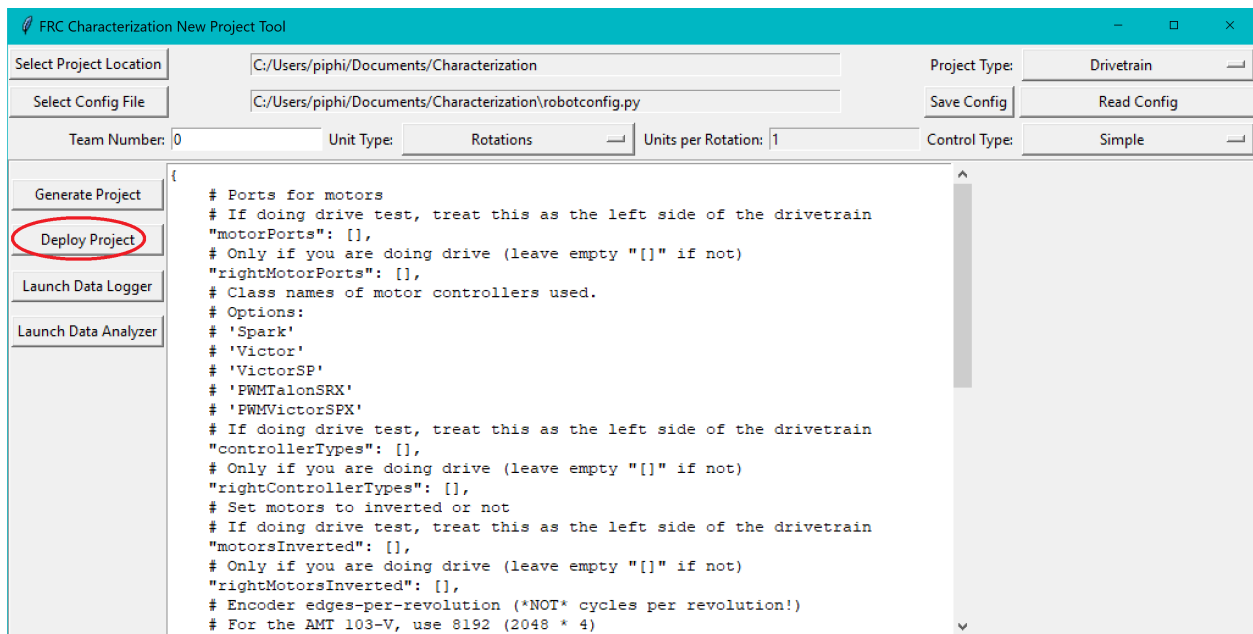
26.3 Deploying a Project

Once a project has been generated, it is time to deploy it to the robot. This can be done in two ways.

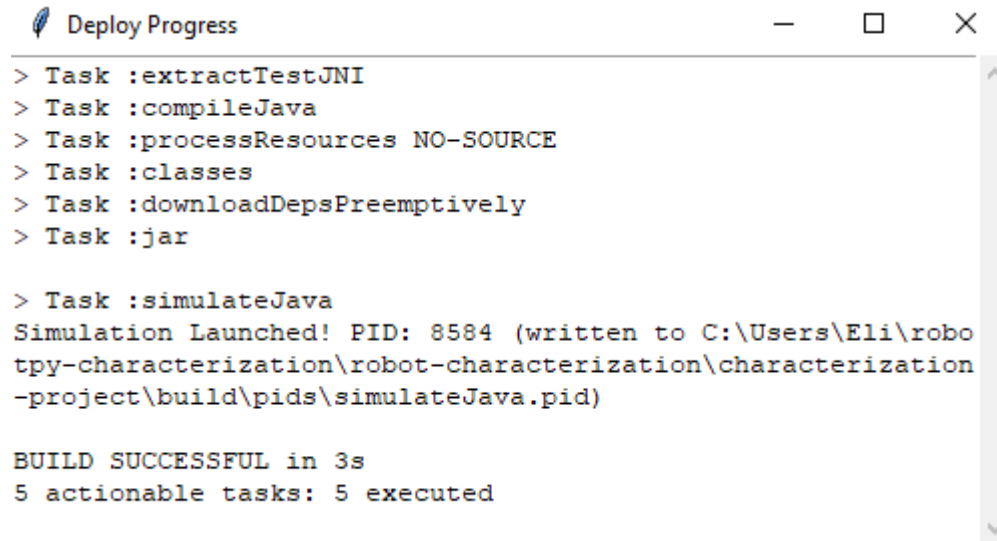
Warning: To ensure the safety of robot hardware, it is highly recommended that you visually inspect the generated `Robot.java` file to make sure it has no potentially harmful errors before deploying the program.

26.3.1 Option 1: Using the Deploy Project Button

Pressing the *Deploy Project* button on the GUI will attempt to use GradleRIO to deploy the project to your robot. The GUI will always *assume* that the project is located in the characterization-project subfolder of the chosen project location.



Assuming a valid robot project is present at that location, a window should pop up displaying the Gradle output as the project builds and deploys.



```

Deploy Progress
> Task :extractTestJNI
> Task :compileJava
> Task :processResources NO-SOURCE
> Task :classes
> Task :downloadDepsPreemptively
> Task :jar

> Task :simulateJava
Simulation Launched! PID: 8584 (written to C:\Users\Eli\robotpy-characterization\robot-characterization\characterization-project\build\pids\simulateJava.pid)

BUILD SUCCESSFUL in 3s
5 actionable tasks: 5 executed

```

26.3.2 Option 2: Deploying Manually

Since the generated project is a standard GradleRIO Java project, it can be deployed like any other. Users may open the generated project in their editor of choice and deploy as they normally would any other robot project. This can be convenient if customization of the generated code is required.

Now that the characterization code has been deployed, you'll need to run the characterization routine.

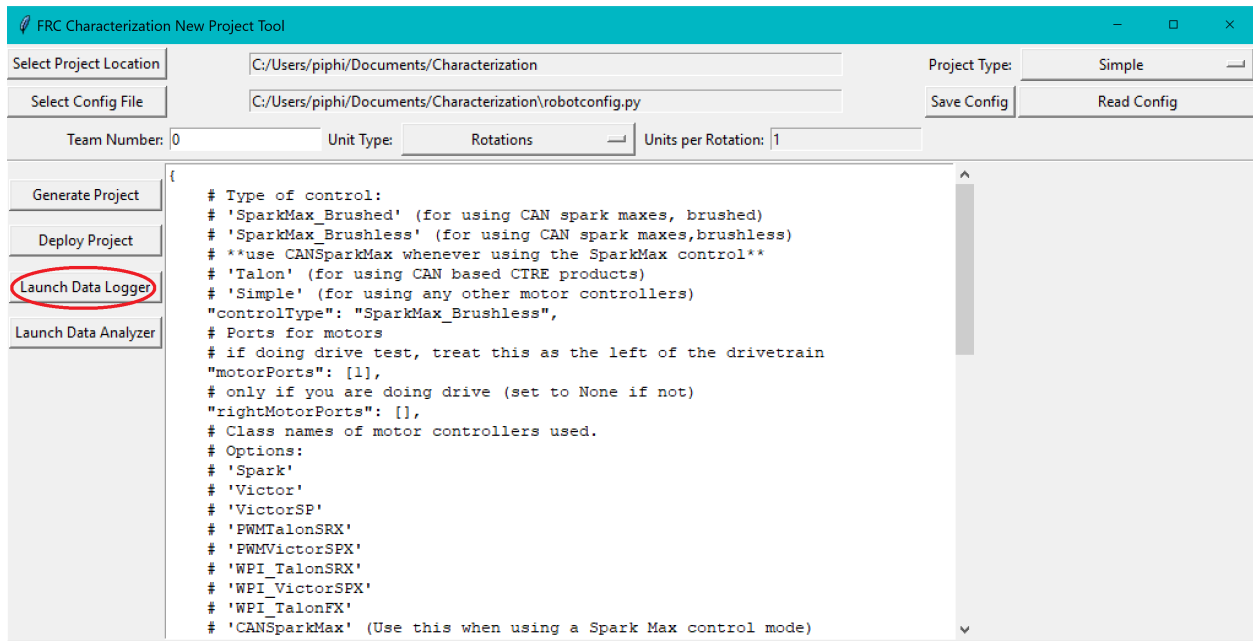
26.4 Running the Characterization Routine

Once the characterization code has been deployed, we can now run the characterization routine, and record the resulting data for analysis.

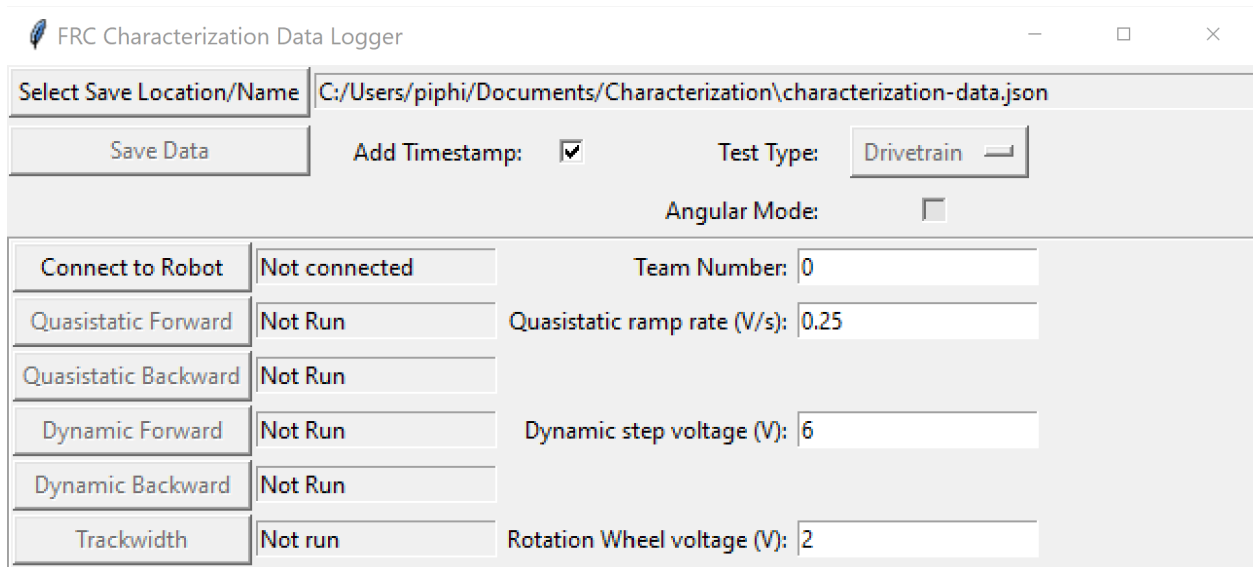
Note: Ensure you have sufficient space around the robot before running any characterization routine! The drive characterization requires at least 10' of space, ideally closer to 20'. The robot drive can not be accurately characterized while on blocks.

26.4.1 Launch the Data Logger

To launch the data logger, press the *Launch Data Logger* button.



This should open the data logger GUI.



Note: The logger allows you to change the previously selected test type through the *Test Type* dropdown. Also, the Angular Mode, Trackwidth, and Rotation Wheel features will do nothing unless you have selected Drivetrain as your Test Type.

26.4.2 Connect to the Robot

Next, we must connect to the robot. Press the *Connect to Robot* button. The status indicated next to the button should change to *Connecting...* while the tool attempts to connect to the robot's NetworkTables server.

The screenshot shows the 'FRC Characterization Data Logger' window. At the top, there is a 'Select Save Location/Name' field with the path 'C:/Users/piphi/Documents/Characterization\characterization-data.json'. Below this is a 'Save Data' button. To the right of 'Save Data' are 'Add Timestamp:' (checked) and 'Test Type:' (set to 'Drivetrain'). Below these is 'Angular Mode:' (unchecked). The main area contains a table of test buttons and their status:

Connect to Robot (circled in red)	Not connected	Team Number:	0
Quasistatic Forward	Not Run	Quasistatic ramp rate (V/s):	0.25
Quasistatic Backward	Not Run		
Dynamic Forward	Not Run	Dynamic step voltage (V):	6
Dynamic Backward	Not Run		
Trackwidth	Not run	Rotation Wheel voltage (V):	2

If the tool does not seem to be successfully connecting, try rebooting the robot. Eventually, the status should change to *Connected*, indicating the tool is successfully communicating with the robot.

The screenshot shows the 'FRC Characterization Data Logger' window after a successful connection. The 'Connect to Robot' button is now circled in red and its status has changed to 'Connected'. The other test buttons remain 'Not Run'. The 'Test Type' is now set to 'Simple'.

Connect to Robot (circled in red)	Connected	Team Number:	0
Quasistatic Forward	Not Run	Quasistatic ramp rate (V/s):	0.25
Quasistatic Backward	Not Run		
Dynamic Forward	Not Run	Dynamic step voltage (V):	6
Dynamic Backward	Not Run		
Trackwidth	Not run	Rotation Wheel voltage (V):	2

26.4.3 Running Tests

A standard motor characterization routine consists of two types of tests:

- **Quasistatic:** In this test, the mechanism is gradually sped-up such that the voltage corresponding to acceleration is negligible (hence, “as if static”).
- **Dynamic:** In this test, a constant ‘step voltage’ is given to the mechanism, so that the behavior while accelerating can be determined.

Each test type is run both forwards and backwards, for four tests in total, corresponding to the four buttons.

The screenshot shows the 'FRC Characterization Data Logger' window. At the top, there's a 'Select Save Location/Name' field with the path 'C:/Users/piphi/Documents/Characterization\characterization-data.json'. Below this are buttons for 'Save Data', 'Add Timestamp' (checked), 'Test Type' (set to 'Simple'), and 'Angular Mode' (unchecked). A table below shows the status of various tests:

Test	Status	Parameter	Value
Connect to Robot	Connected	Team Number	0
Quasistatic Forward	Not Run	Quasistatic ramp rate (V/s)	0.25
Quasistatic Backward	Not Run		
Dynamic Forward	Not Run	Dynamic step voltage (V)	6
Dynamic Backward	Not Run		
Trackwidth	Not run	Rotation Wheel voltage (V)	2

The tests can be run in any order, but running a “backwards” test directly after a “forwards” test is generally advisable (as it will more or less reset the mechanism to its original position).

Follow the instructions in the pop-up windows after pressing each test button.

This screenshot shows the same 'FRC Characterization Data Logger' window, but with a pop-up dialog titled 'Running slow-forward'. The dialog contains an information icon and the following text:

Please enable the robot in autonomous mode, and then disable it before it runs out of space.
 Note: The robot will continue to move until you disable it - It is your responsibility to ensure it does not hit anything!

An 'OK' button is at the bottom right of the dialog. In the background, the test status table is partially visible, showing 'Quasistatic Forward' as 'Running...' and 'Quasistatic Backward' as 'Completed'.

Trackwidth

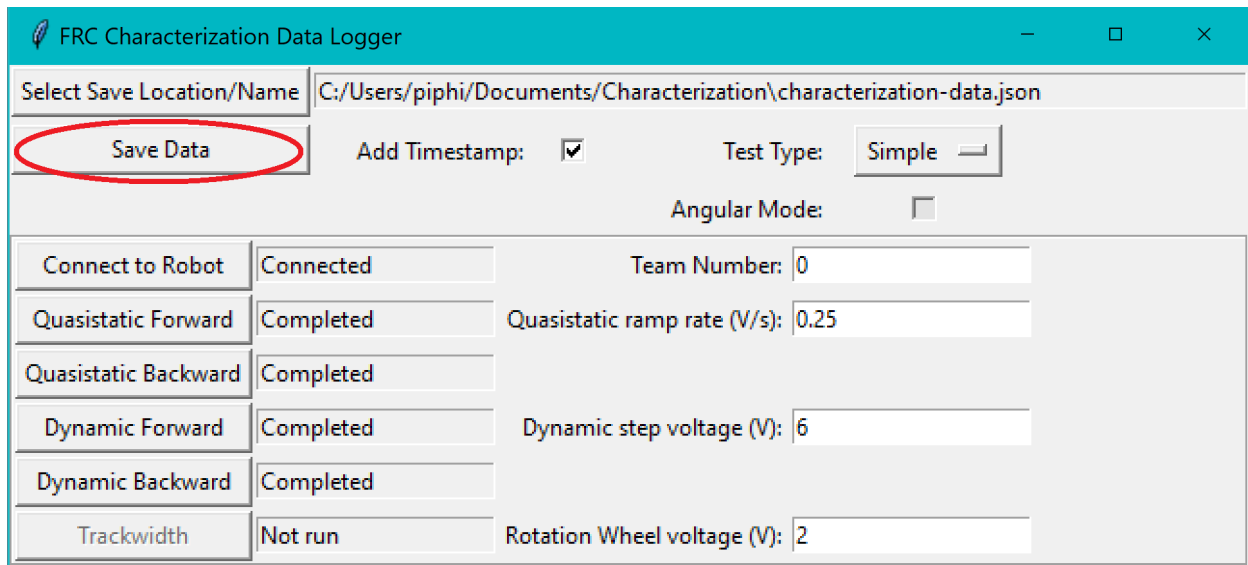
The trackwidth test will spin your robot to determine an empirical trackwidth. It compares how far the wheel encoders drove against the reported rotation from the gyroscope. To get the best results your wheels should maintain contact with the ground.

Note: If your robot is having trouble turning during the Trackwidth test you should increase the *Rotation Wheel voltage (V)*: value until your robot is smoothly turning and run the test again.

Note: For high-friction wheels (like pneumatic tires), the empirical trackwidth calculated by frc-characterization may be significantly different from the real trackwidth (e.g., off by a factor of 2). The empirical value should be preferred over the real one in robot code.

The entire routine should look something like this:

After all four tests have been completed, the *Save Data* button will become activated.



FRC Characterization Data Logger		
Select Save Location/Name	C:/Users/piphi/Documents/Characterization\characterization-data.json	
Save Data	Add Timestamp: <input checked="" type="checkbox"/>	Test Type: Simple
	Angular Mode: <input type="checkbox"/>	
Connect to Robot	Connected	Team Number: 0
Quasistatic Forward	Completed	Quasistatic ramp rate (V/s): 0.25
Quasistatic Backward	Completed	
Dynamic Forward	Completed	Dynamic step voltage (V): 6
Dynamic Backward	Completed	
Trackwidth	Not run	Rotation Wheel voltage (V): 2

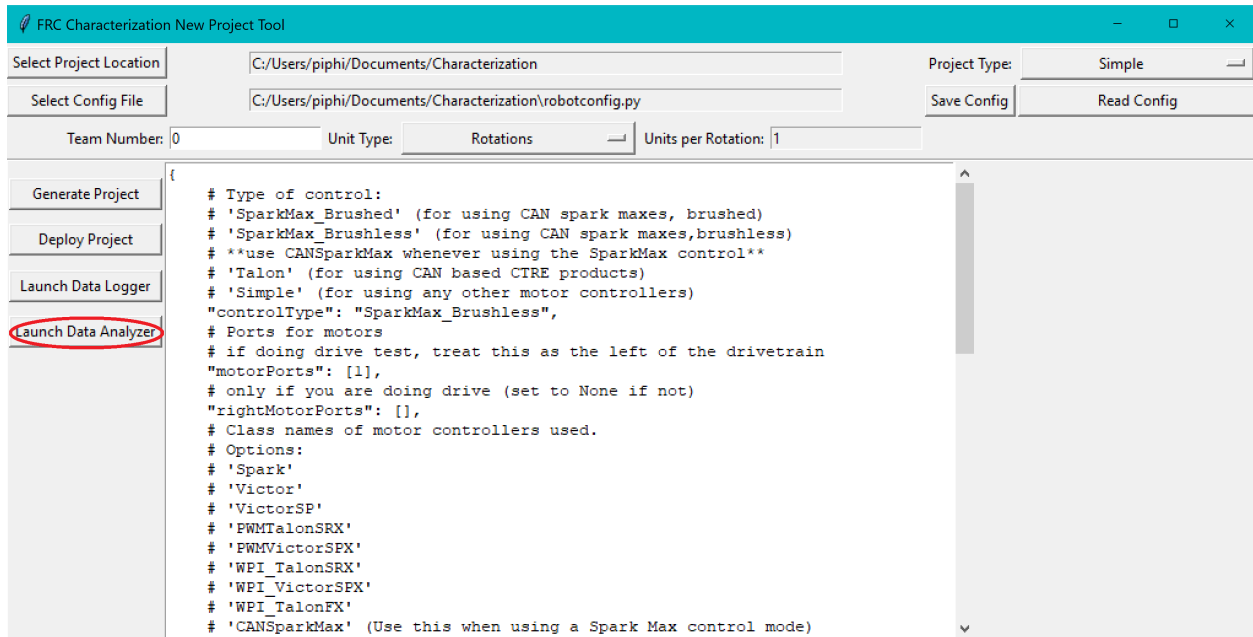
This will save the data as a JSON file with the specified location/name. A timestamp (%Y%m%d-%H%M) will be appended to the chosen filename if the *Add Timestamp* button is checked.

Note: You can run a preliminary check on the quality of the characterization data by enabling prints on Driver Station. After exiting autonomous in each test, the console should output Collected : n in t seconds where n should be $200 * t$ (rounded). More information can be found [here](#)

26.5 Analyzing Data

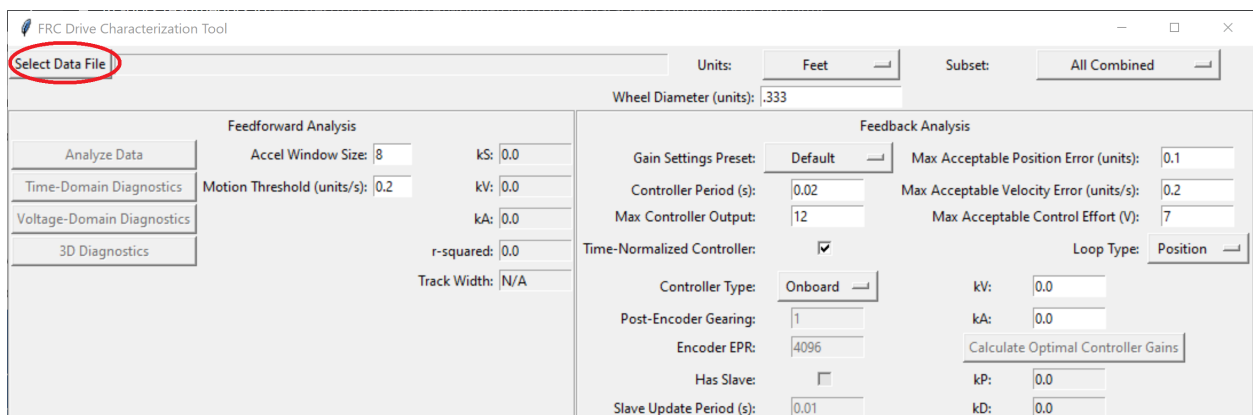
Important: WPILib standardizes on SI units, so it's recommended that the *Units* option is set to **Meters**.

Once we have data from a characterization run, we can analyze it. To launch the data analyzer, click on the *Launch Data Analyzer* button.



26.5.1 Loading your Data File

Now it's time to load the data file we saved from the logger tool. Click on *Select Data File*.



In the resulting file dialog, select the JSON file you want to analyze. If the file appears to be malformed, an error will be shown.

26.5.2 Running Feedforward Analysis

Once a data file has been selected, the *Analyze Data* button, *Units* entry, and *Units per rotation* entry become available in the *Feedforward Analysis* frame. We can now set the units of the analysis to match the units that our program will be using.

Now click the *Analyze Data* button.

Note: If you would like to change units, you will have to press the *Analyze Data* and the *Calculate Optimal Controller Gains* (if you've pressed it) buttons.

The screenshot shows the FRC Drive Characterization Tool interface. The 'Analyze Data' button is circled in red. The 'Units' dropdown is set to 'Rotations' and 'Units per rotation' is 1. The 'Subset' dropdown is set to 'Combined' and 'Test' is 'Simple'. The 'Feedforward Analysis' section shows calculated values: kS: 0.547, kG: 0.0, kCos: 0.0, kV: 0.0693, kA: 0.117, r-squared: 0.999, and Track Width: N/A. The 'Feedback Analysis' section shows various gain settings and a 'Calculate Optimal Controller Gains' button.

By default, the analysis will be run by combining all the data in the test. For a finer-grained analysis, the analysis may be limited to a specific subset of data using the subset dropdown menu.

The screenshot shows the FRC Drive Characterization Tool interface. The 'Combined' option in the 'Subset' dropdown is circled in red. The 'Units' dropdown is set to 'Rotations' and 'Units per rotation' is 1. The 'Test' dropdown is set to 'Simple'. The 'Feedforward Analysis' section shows calculated values: kS: 0.547, kG: 0.0, kCos: 0.0, kV: 0.0693, kA: 0.117, r-squared: 0.999, and Track Width: N/A. The 'Feedback Analysis' section shows various gain settings and a 'Calculate Optimal Controller Gains' button.

The computed coefficients of the mechanism characterization will then be filled in, along with a goodness-of-fit measure (r-squared).

FRC Drive Characterization Tool

Select Data File: C:/Users/piphi/Documents/WpilibProjects/FRCUploader/characterization-data20201022-01: Units: Rotations Subset: Combined

Units per rotation: 1 Test: Simple

Feedforward Analysis		Feedback Analysis	
Analyze Data	Accel Window Size: 8	Gain Settings Preset: Default	Max Acceptable Position Error (units): 1
Time-Domain Diagnostics	Motion Threshold (units/s): 0.2	Controller Period (s): 0.02	Max Acceptable Velocity Error (units/s): 1.5
Voltage-Domain Diagnostics		Max Controller Output: 12	Max Acceptable Control Effort (V): 7
3D Diagnostics		Time-Normalized Controller: <input checked="" type="checkbox"/>	Loop Type: Velocity
	kS: 0.547	Controller Type: Onboard	kV: 0.0693
	kG: 0.0	Measurement delay (ms): 0	kA: 0.117
	kCos: 0.0	Post-Encoder Gearing: 1	Calculate Optimal Controller Gains
	kV: 0.0693	Encoder EPR: 4096	kP: 3.11
	kA: 0.117	Has Follower: <input type="checkbox"/>	kD: 0.0
	r-squared: 0.999	Follower Update Period (s): 0.01	Convert Gains: <input type="checkbox"/>
	Track Width: N/A		

26.6 Viewing Diagnostics

The first diagnostic to look at is the r-squared - it should be somewhere north of $\sim .9$. If it is significantly lower than this, there is likely a problem with your characterization data.

To investigate further, you can generate a number of diagnostic plots with the buttons on the left-hand side:

FRC Drive Characterization Tool

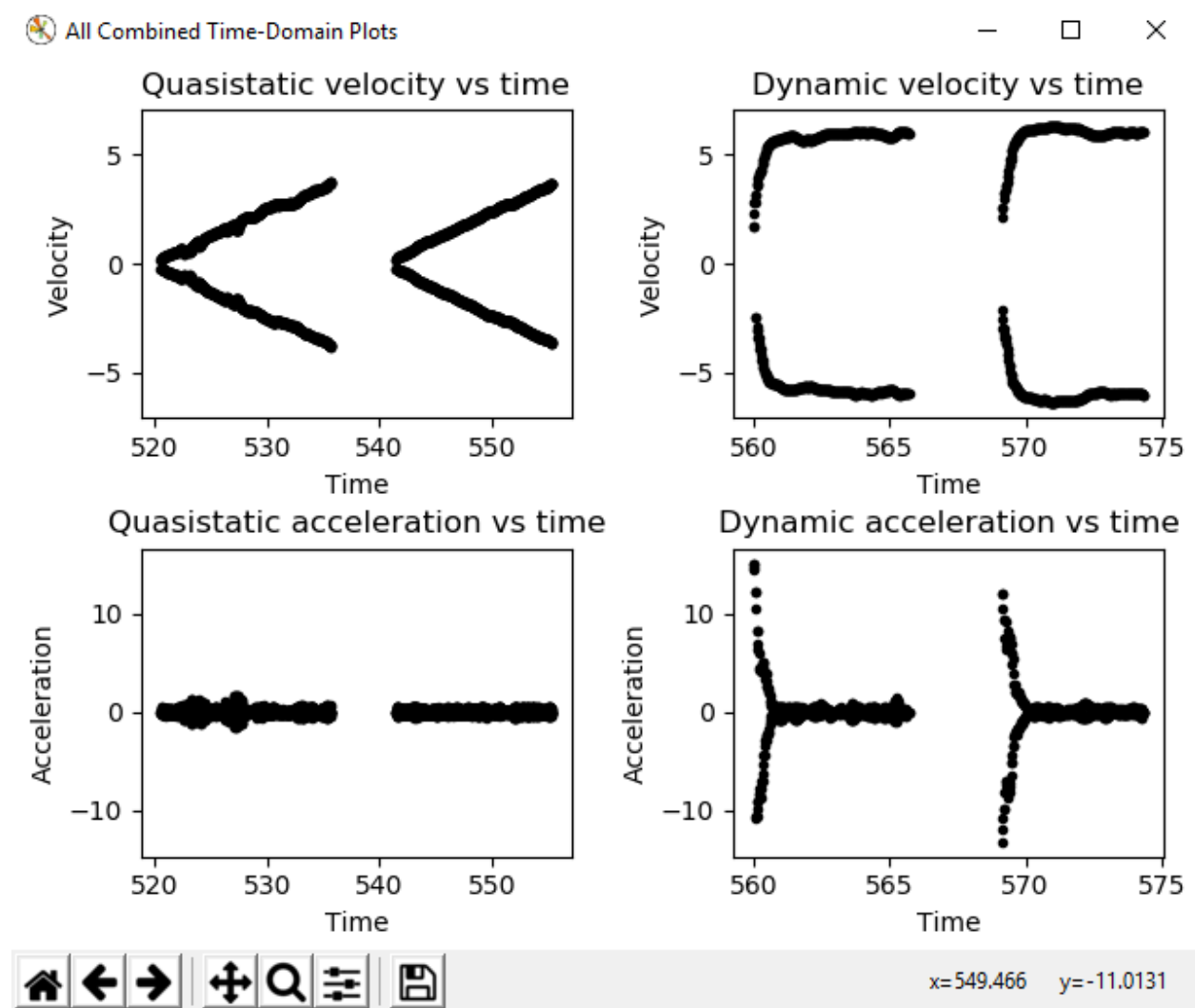
Select Data File: C:/Users/piphi/Documents/WpilibProjects/FRCUploader/characterization-data20201025-09: Units: Feet Subset: Combined

Units per rotation: 1.04 Test: Simple

Feedforward Analysis		Feedback Analysis	
Analyze Data	Accel Window Size: 8	Gain Settings Preset: Default	Max Acceptable Position Error (units): 1
Time-Domain Diagnostics	Motion Threshold (units/s): 0.2	Controller Period (s): 0.02	Max Acceptable Velocity Error (units/s): 1.5
Voltage-Domain Diagnostics		Max Controller Output: 12	Max Acceptable Control Effort (V): 7
3D Diagnostics		Time-Normalized Controller: <input checked="" type="checkbox"/>	Loop Type: Velocity
	kS: 0.696	Controller Type: Onboard	kV: 0.0672
	kG: 0.0	Measurement delay (ms): 0	kA: 0.119
	kCos: 0.0	Post-Encoder Gearing: 1	Calculate Optimal Controller Gains
	kV: 0.0672	Encoder EPR: 4096	kP: 3.13
	kA: 0.119	Has Follower: <input type="checkbox"/>	kD: 0.0
	r-squared: 0.998	Follower Update Period (s): 0.01	Convert Gains: <input type="checkbox"/>
	Track Width: N/A		

26.6.1 Time-Domain Diagnostics

The Time-Domain Diagnostics plots display velocity and acceleration versus time over the course of the analyzed tests. For a typical drive characterization, these should look something like this (other mechanisms will be highly similar):



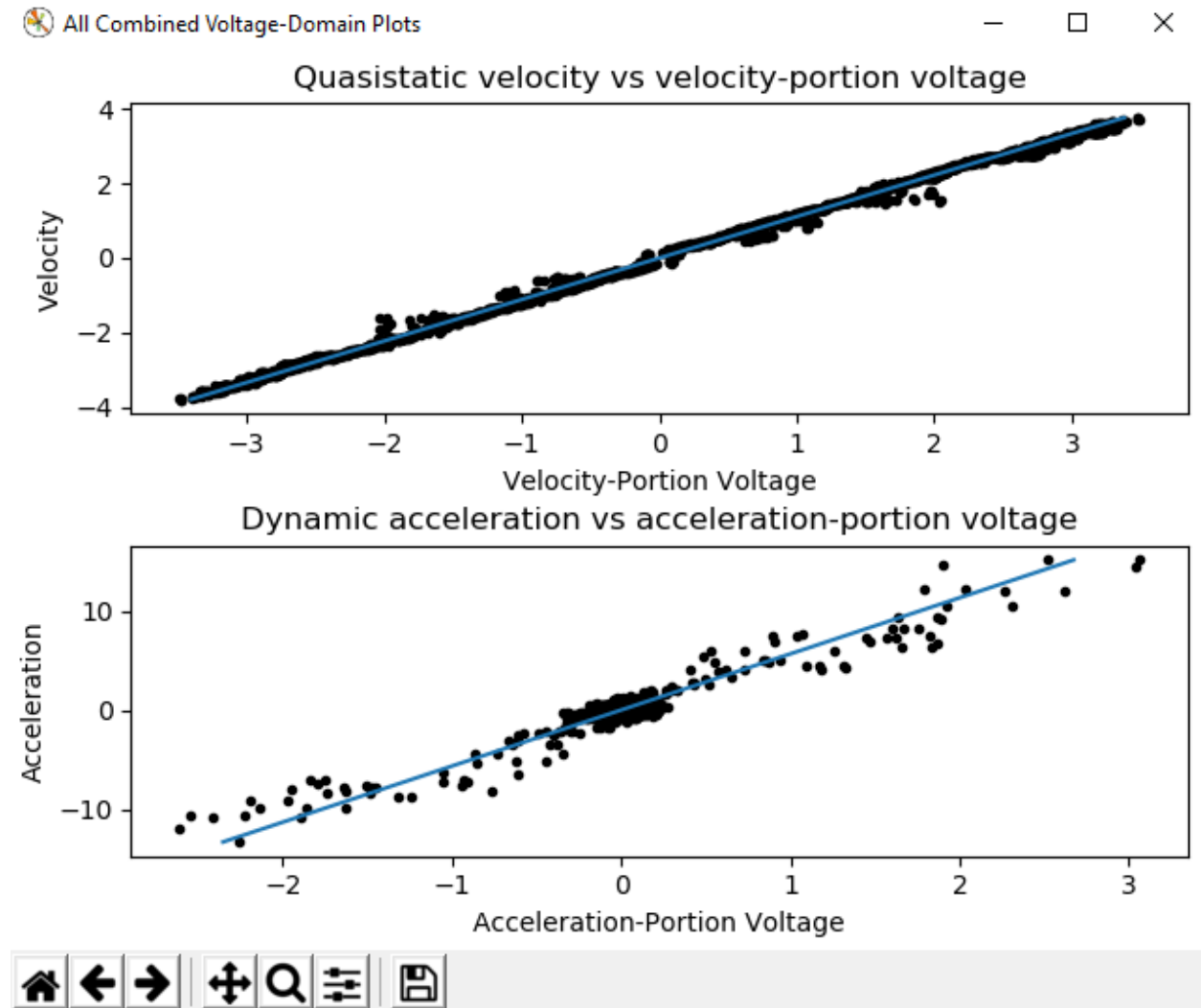
The vertical “mirroring” visible here is normal, and is simply the result of the left- and right-side encoders having different signs - this does not cause any trouble for the characterization tool.

The quasistatic test ought to have nearly linear velocity, and nearly-zero acceleration (hence “quasistatic”). The dynamic test ought to have velocity that asymptotically approaches a steady-state speed (the shape of the curve should be exponential, in fact), and acceleration that, accordingly, rapidly falls to zero (also exponentially, as the derivative of an exponential function is also an exponential function).

Deviation from this behavior is a sign of an [error](#), either in your robot setup, analysis settings, or your test procedure.

26.6.2 Voltage-Domain Diagnostics

The *Voltage-Domain* Diagnostics button plots velocity and acceleration versus voltage. Velocity is plotted for the quasistatic test, and acceleration is plotted for the dynamic test. For a typical drive characterization, the plots should resemble this (again, other mechanisms will be similar)



Both plots should be linear, however the dynamic plot will almost certainly have substantially more noise. The noise on the dynamic plot may be reduced by increasing the Accel Window Size setting.

FRC Drive Characterization Tool

Select Data File: C:/Users/piphi/Documents/WpilibProjects/FRCUploader/characterization-data20201025-09: Units: Feet Subset: Combined

Units per rotation: 1.04 Test: Simple

Feedforward Analysis

Analyze Data

Time-Domain Diagnostics

Voltage-Domain Diagnostics

3D Diagnostics

Accel Window Size: 8

Motion Threshold (units/s): 0.2

kS: 0.696

kG: 0.0

kCos: 0.0

kV: 0.0672

kA: 0.119

r-squared: 0.998

Track Width: N/A

Feedback Analysis

Gain Settings Preset: Default

Controller Period (s): 0.02

Max Controller Output: 12

Time-Normalized Controller: ☒

Controller Type: Onboard

Measurement delay (ms): 0

Post-Encoder Gearing: 1

Encoder EPR: 4096

Has Follower: ☐

Follower Update Period (s): 0.01

Max Acceptable Position Error (units): 1

Max Acceptable Velocity Error (units/s): 1.5

Max Acceptable Control Effort (V): 7

Loop Type: Velocity

kV: 0.0672

kA: 0.119

Calculate Optimal Controller Gains

kP: 3.13

kD: 0.0

Convert Gains: ☐

However, if your robot or mechanism has low mass compared to the motor power, this may “eat” what little meaningful acceleration data you have (however, in these cases kA will tend towards zero and can usually be ignored, anyway).

Note: The x-axis corresponds to velocity-portion voltage and acceleration-portion voltage, respectively - as the governing voltage-balance equations are multi-dimensional, plots against raw voltage are not as useful as one might expect.

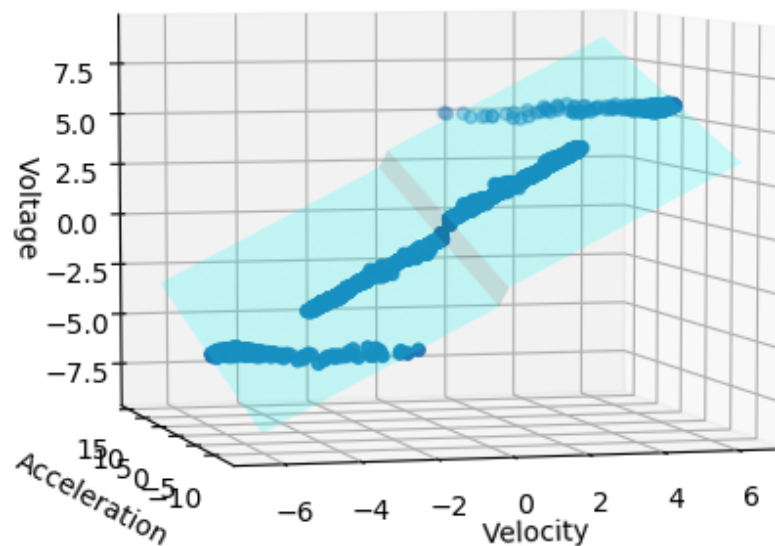
26.6.3 3D Diagnostics

The *3D Diagnostics* button will generate a 3d plot of voltage over the entire velocity-acceleration plane (this may be an adjusted voltage to remove the nonlinearity in mechanisms with nonlinear equations, such as arms).

All Combined 3D Vel-Accel Plane Plot

— □ ×

Voltage vs velocity and acceleration



This plot is interactive, and may be rotated by clicking-and-dragging. The quasistatic and dynamic tests should both be clearly visible as streaks of data, and the best fit-plane should pass through all the data points. The data from both the quasistatic and dynamic tests should appear as straight lines (the reason for this is left as an exercise for the reader).

The discontinuity corresponds to k_S , which always opposes the direction of motion and thus changes direction as the plot crosses the 0 velocity mark.

Common Failure Modes

When something has gone wrong with the characterization, diagnostic plots and console output provide crucial clues as to *what* has gone wrong. This section describes some common failures encountered while running the characterization tool, the identifying features of their diagnostic plots, and the steps that can be taken to fix them.

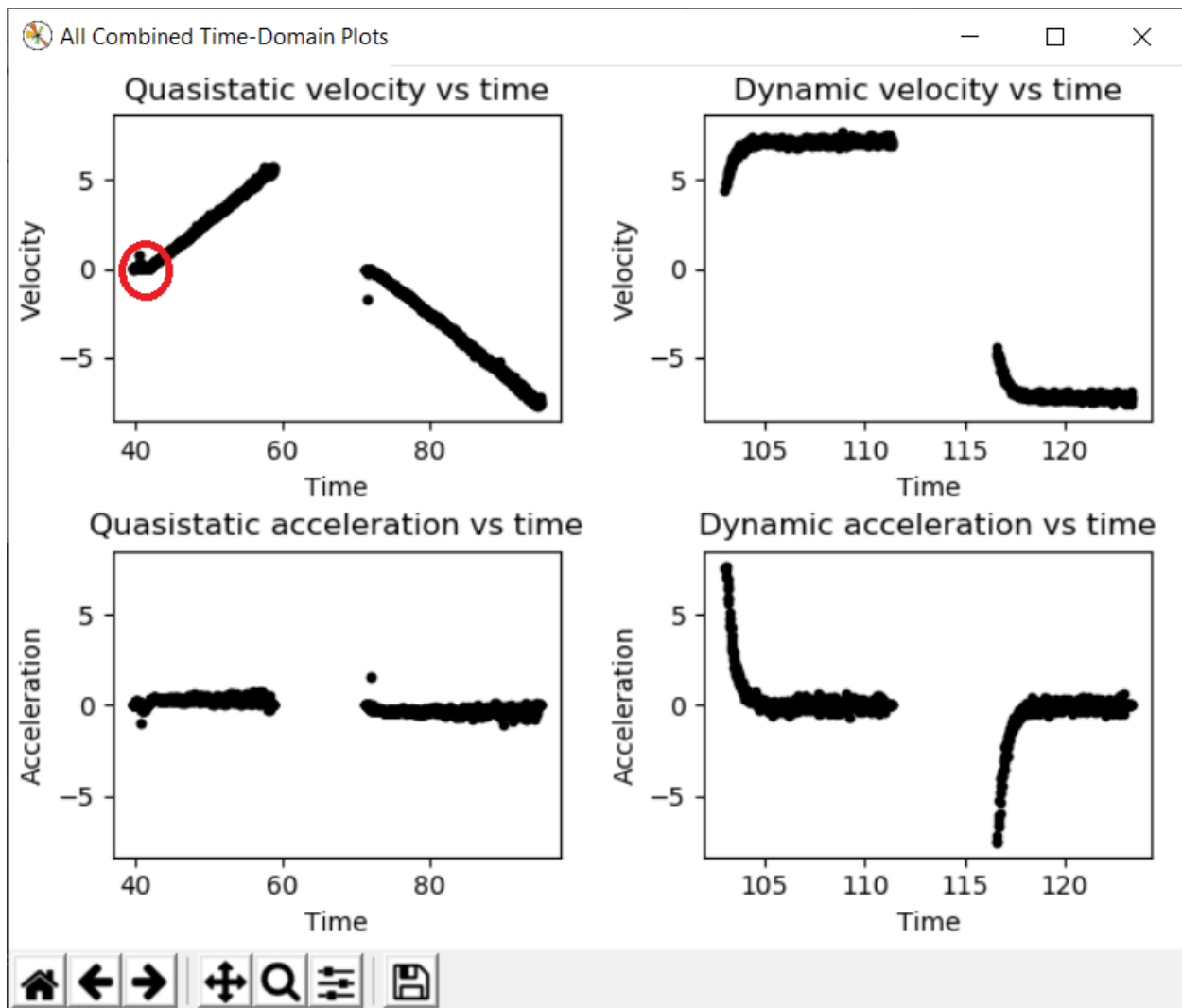
Improperly Set Motion Threshold

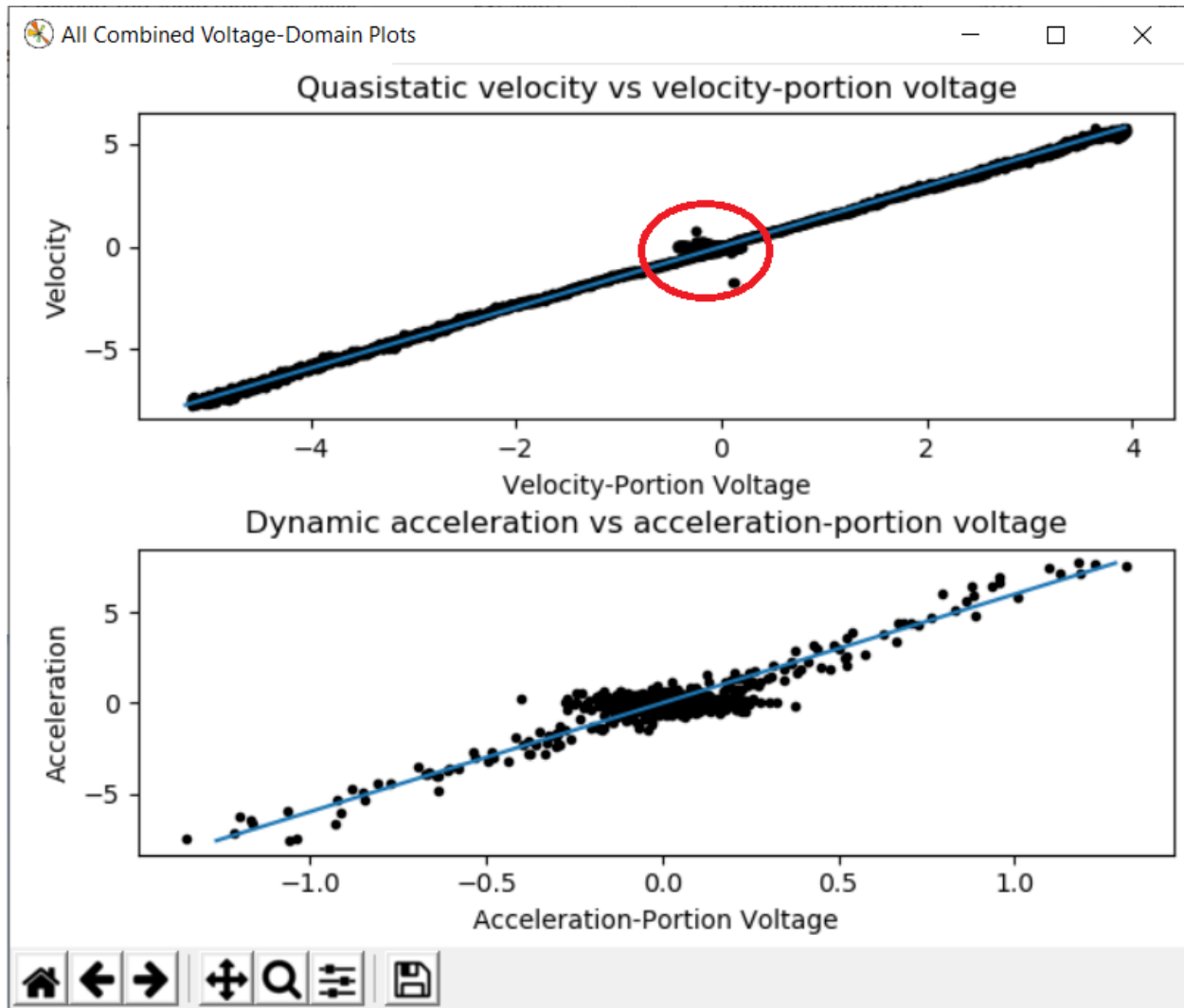
One of the most-common errors is an inappropriate value for the motion threshold.

The screenshot shows the FRC Drive Characterization Tool interface. The 'Motion Threshold (units/s)' is highlighted with a red box and set to 0.2. The interface is divided into two main sections: Feedforward Analysis and Feedback Analysis.

Feedforward Analysis		Feedback Analysis	
Analyze Data	Accel Window Size: 8	kS: 0.696	Gain Settings Preset: Default
Time-Domain Diagnostics	Motion Threshold (units/s): 0.2	kG: 0.0	Controller Period (s): 0.02
Voltage-Domain Diagnostics		kCos: 0.0	Max Controller Output: 12
3D Diagnostics		kV: 0.0672	Time-Normalized Controller: <input checked="" type="checkbox"/>
		kA: 0.119	Controller Type: Onboard
		r-squared: 0.998	Measurement delay (ms): 0
	Track Width: N/A		Post-Encoder Gearing: 1
			Encoder EPR: 4096
			Has Follower: <input type="checkbox"/>
			Follower Update Period (s): 0.01
			Max Acceptable Position Error (units): 1
			Max Acceptable Velocity Error (units/s): 1.5
			Max Acceptable Control Effort (V): 7
			Loop Type: Velocity
			kV: 0.0672
			kA: 0.119
			Calculate Optimal Controller Gains
			kP: 3.13
			kD: 0.0
			Convert Gains: <input type="checkbox"/>

Motion Threshold Too Low

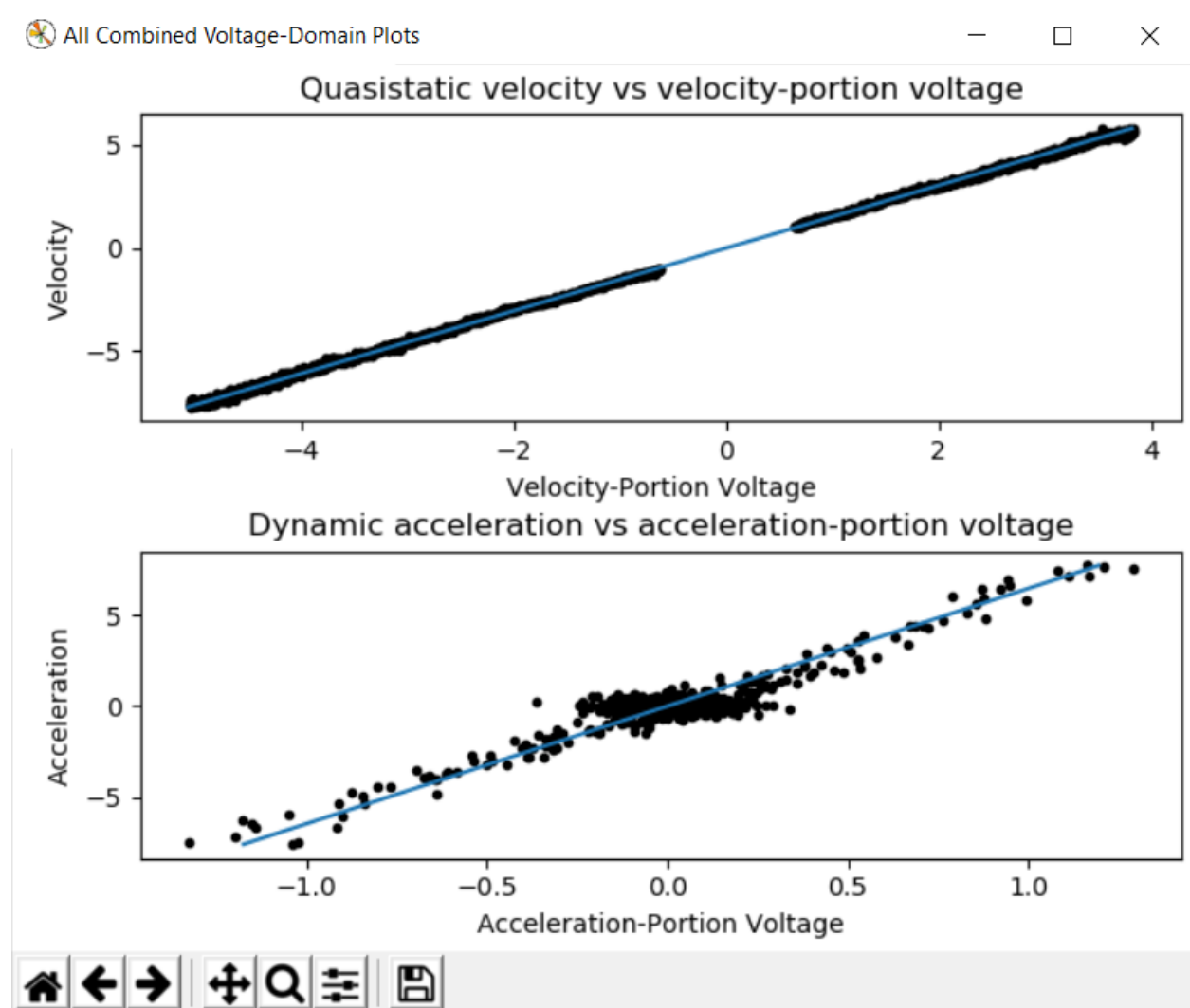




The presence of a “leading tail” (emphasized by added red circle) on the time-domain and voltage-domain plots indicates that the *Motion Threshold* setting is too low, and thus data points from before the robot begins to move are being included.

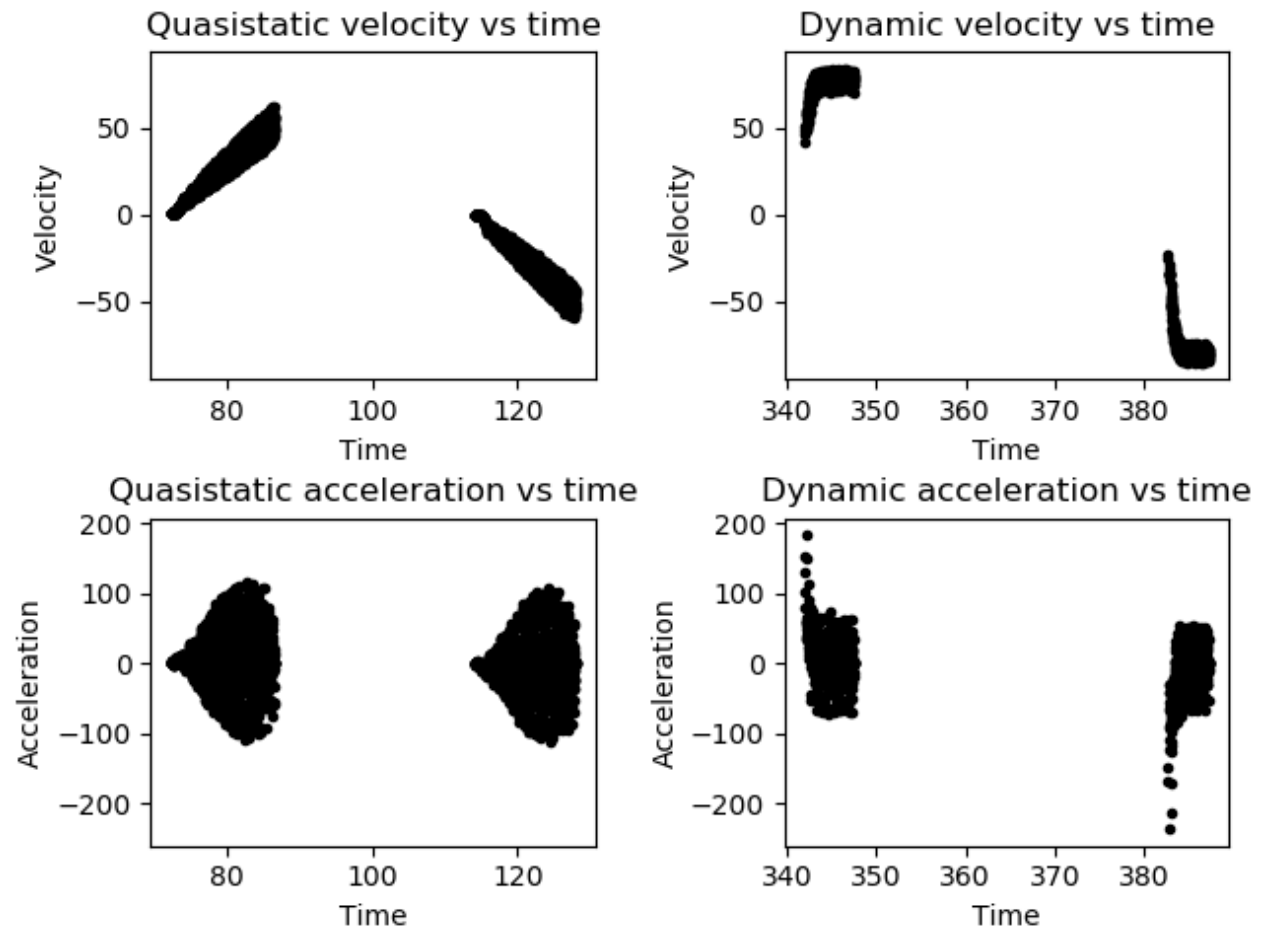
To solve this, increase the setting and re-analyze the data.

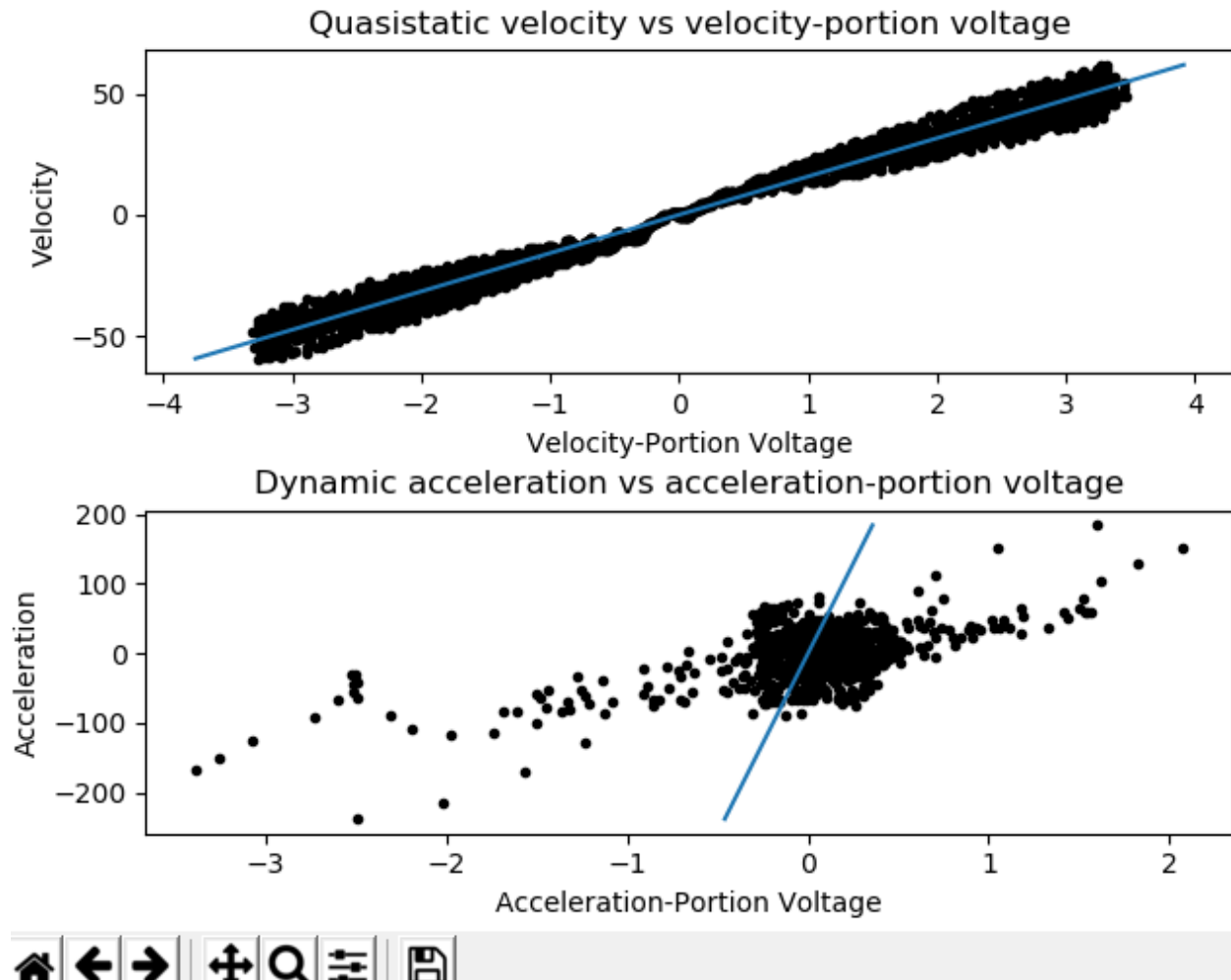
Motion Threshold Too High



While not nearly as problematic as a too-low threshold, a motion threshold that is too high will result in a large “gap” in the voltage domain quasistatic plot.

To solve this, decrease the setting and re-analyze the data.

Magnetic Encoders Velocity Noise



Magnetic encoders such as the [CTRE Mag Encoder](#) and the [AndyMark magnetic encoder](#) are extremely popular in FRC. However, a particular noise pattern has been observed when these encoders are used on robot drives, whose particular cause is not yet known. This noise pattern is uniquely distinguished by significant velocity noise proportional to motor velocity, and is particularly common on the kit-of-parts [toughbox mini](#) gearboxes.

Characterization constants can sometimes be accurately determined even from data polluted this noise by increasing the accel window size setting. However, this sort of encoder noise is problematic for robot code much the same way it is problematic for the characterization tool. As the root cause of the noise is not known, it is recommended to try a different encoder setup if this is observed, either by moving the encoders to a different shaft or replacing them with a different type of encoder.

Template Lag

With the new characterization tool, the logging code might not be able to keep up with its 5 ms refresh rate thus causing faulty data to be collected.

To see if this is the case, enable print statements on the Driver Station whenever running the data logger. When Autonomous mode is exited, the console will output Collected : n in t seconds where n is the number of samples and t is the time elapsed. If the sampling was successful, n should equal $200t$ (rounded).

26.7 Feedback Analysis

Important: These gains are, in effect, “educated guesses” - they are not guaranteed to be perfect, and should be viewed as a “starting point” for further tuning.

Warning: The feedback gain calculation assumes that there is no mechanical backlash, sensor noise, or phase lag in the sensor measurement. While these are reasonable assumptions in many situations, none of them are strictly true in practice. In particular, many “smart motor controllers” (such as the Talon SRX, Talon FX, and SPARK MAX) have default settings that apply substantial *low-pass filtering* to their encoder velocity measurements, which introduces a significant amount of phase lag. This can cause the calculated gains for velocity loops to be unstable. To rectify this, either decrease the amount of filtering through the controller’s API, or reduce the magnitude of the PID gains - it has been found that shrinking gains by about a factor of 10 works well for most default filtering settings.

Once the feedforward coefficients have been computed, the controls on the *Feedback Analysis* pane become available.

The screenshot shows the FRC Drive Characterization Tool interface. The top bar indicates the file path: C:/Users/piphi/Documents/WpilibProjects/FRCUploader/characterization-data20201022-01. The Units are set to Rotations, and the Subset is Combined. The Test is Simple. The Units per rotation is 1. The Feedback Analysis pane is highlighted with a red box. It contains the following settings:

- Gain Settings Preset: Default
- Max Acceptable Position Error (units): 1
- Controller Period (s): 0.02
- Max Acceptable Velocity Error (units/s): 1.5
- Max Controller Output: 12
- Max Acceptable Control Effort (V): 7
- Time-Normalized Controller: ☒
- Loop Type: Velocity
- Controller Type: Onboard
- kV: 0.0693
- kA: 0.117
- Measurement delay (ms): 0
- Post-Encoder Gearing: 1
- Encoder EPR: 4096
- kP: 3.11
- kD: 0.0
- Has Follower: ☐
- Follower Update Period (s): 0.01
- Calculate Optimal Controller Gains:
- Convert Gains: ☐

On the left side of the Feedback Analysis pane, there are buttons for Analyze Data, Time-Domain Diagnostics, Voltage-Domain Diagnostics, and 3D Diagnostics. Below these buttons, the Feedforward Analysis section shows the following values:

- Accel Window Size: 8
- Motion Threshold (units/s): 0.2
- kS: 0.547
- kG: 0.0
- kCos: 0.0
- kV: 0.0693
- kA: 0.117
- r-squared: 0.999
- Track Width: N/A

These can be used to calculate optimal feedback gains for a PD or P controller for your mechanism (via [LQR](#)).

26.7.1 Enter Controller Parameters

Note: The “Spark Max” preset assumes that the user has configured the controller to operate in the units of analysis with the SPARK MAX API’s position/velocity scaling factor feature.

The calculated feedforward gains are *dimensioned quantities*. Unfortunately, not much attention is often paid to the units of PID gains in FRC® controls, and so the various typical options for PID controller implementations differ in their unit conventions (which are often not made clear to the user).

To specify the correct settings for your PID controller, use the following options.

The screenshot shows the 'FRC Drive Characterization Tool' window. The 'Gain Settings Preset' dropdown menu is highlighted with a red box. The interface includes sections for 'Feedforward Analysis' and 'Feedback Analysis'. The 'Gain Settings Preset' dropdown is currently set to 'Default'. Other visible settings include 'Controller Period (s)' at 0.02, 'Max Controller Output' at 12, 'Time-Normalized Controller' checked, 'Controller Type' set to 'Onboard', 'Measurement delay (ms)' at 0, 'Post-Encoder Gearing' at 1, 'Encoder EPR' at 4096, 'Has Follower' unchecked, and 'Follower Update Period (s)' at 0.01. The 'Feedback Analysis' section shows 'Max Acceptable Position Error (units)' at 1, 'Max Acceptable Velocity Error (units/s)' at 1.5, 'Max Acceptable Control Effort (V)' at 7, 'Loop Type' set to 'Velocity', 'kV' at 0.0693, 'kA' at 0.117, 'kP' at 3.11, and 'kD' at 0.0. The 'Calculate Optimal Controller Gains' button is visible, along with a 'Convert Gains' checkbox.

- **Gain Settings Preset:** This drop-down menu will auto-populate the remaining fields with likely settings for one of a number of common FRC controller setups. Note that some settings, such as post-encoder gearing, PPR, and the presence of a follower motor must still be manually specified (as the analyzer has no way of knowing these without user input), and that others may vary from the given defaults depending on user setup.
- **Controller Period:** This is the execution period of the control loop, in seconds. The default RIO loop rate is 50Hz, corresponding to a period of 0.02s. The onboard controllers on most “smart controllers” run at 1Khz, or a period of 0.001s.
- **Max Controller Output:** This is the maximum value of the controller output, with respect to the PID calculation. Most controllers calculate outputs with a maximum value of 1, but Talon controllers have a maximum output of 1023.
- **Time-Normalized Controller:** This specifies whether the PID calculation is normalized to the period of execution, which affects the scaling of the D gain.
- **Controller Type:** This specifies whether the controller is an onboard RIO loop, or is running on a smart motor controller such as a Talon or a SPARK MAX.
- **Post-Encoder Gearing:** This specifies the gearing between the encoder and the mechanism itself. This is necessary for control loops that do not allow user-specified unit scaling in their PID computations (e.g. those running on Talons). This will be disabled if not relevant.
- **Encoder EPR:** This specifies the edges-per-revolution (not cycles per revolution) of the encoder used, which is needed in the same cases as Post-Encoder Gearing.
- **Has Follower:** Whether there is a motor controller following the controller running the control loop, if the control loop is being run on a peripheral device. This changes the

effective loop period.

- **Follower Update Period:** The rate at which the follower (if present) is updated. By default, this is 100Hz (every 0.01s) for the Talon SRX, Talon FX, and the SPARK MAX, but can be changed.

Note: If you select a smart motor controller as the preset (e.g. TalonSRX, SPARK MAX, etc.) the *Convert Gains* checkbox will be automatically checked. This means the tool will convert your gains so that they can be used through the smart motor controller's PID methods. Therefore, if you would like to use WPILib's PID Loops, you must uncheck that box.

26.7.2 Specify Optimality Criteria

Finally, the user must specify what will be considered an “optimal” controller. This takes the form of desired tolerances for the system error and control effort - note that it is *not* guaranteed that the system will obey these tolerances at all times.

As a rule, smaller values for the *Max Acceptable Error* and larger values for the *Max Acceptable Control Effort* will result in larger gains - this will result in larger control efforts, which can grant better setpoint-tracking but may cause more violent behavior and greater wear on components.

The *Max Acceptable Control Effort* should never exceed 12V, as that corresponds to full battery voltage, and ideally should be somewhat lower than this.

26.7.3 Select Loop Type

It is typical to control mechanisms with both position and velocity PIDs, depending on application. Either can be selected using the drop-down *Loop Type* menu.

FRC Drive Characterization Tool

Select Data File: C:/Users/piphi/Documents/WpilibProjects/FRCUploader/characterization-data20201022-01: Units: Rotations Subset: Combined

Units per rotation: 1 Test: Simple

Feedforward Analysis

Analyze Data Accel Window Size: 8 kS: 0.547

Time-Domain Diagnostics Motion Threshold (units/s): 0.2 kG: 0.0

Voltage-Domain Diagnostics kCos: 0.0

3D Diagnostics kV: 0.0693

kA: 0.117

r-squared: 0.999

Track Width: N/A

Feedback Analysis

Gain Settings Preset: Default Max Acceptable Position Error (units): 1

Controller Period (s): 0.02 Max Acceptable Velocity Error (units/s): 1.5

Max Controller Output: 12 Max Acceptable Control Effort (V): 7

Time-Normalized Controller: ☒ Loop Type: Velocity

Controller Type: Onboard kV: 0.0693

Measurement delay (ms): 0 kA: 0.117

Post-Encoder Gearing: 1 Calculate Optimal Controller Gains

Encoder EPR: 4096 kP: 3.11 Convert Gains: ☐

Has Follower: ☐ kD: 0.0

Follower Update Period (s): 0.01

26.7.4 Enter Known Velocity/Acceleration

Note: Sometimes, with an exceptionally light mechanism/robot and/or exceptionally-noisy data, it is possible for the k_A value to be exceedingly small (or even slightly negative). In this case, the user should set k_A to zero. The computed feedback gains in this case may also be zero - this is because such a mechanism should not require feedback to accurately track the setpoint under the assumptions of LQR. These assumptions may not be perfectly accurate, and users may need to add feedback regardless - in this case, the loop must be tuned manually.

If one wishes to use the *Feedback Analysis* pane without running a full analysis on a set of data, or otherwise view the effect of modifying the k_V and k_A values, this can be done here.

FRC Drive Characterization Tool

Select Data File: C:/Users/piphi/Documents/WpilibProjects/FRCUploader/characterization-data20201022-01: Units: Rotations Subset: Combined

Units per rotation: 1 Test: Simple

Feedforward Analysis

Analyze Data Accel Window Size: 8 kS: 0.547

Time-Domain Diagnostics Motion Threshold (units/s): 0.2 kG: 0.0

Voltage-Domain Diagnostics kCos: 0.0

3D Diagnostics kV: 0.0693

kA: 0.117

r-squared: 0.999

Track Width: N/A

Feedback Analysis

Gain Settings Preset: Default Max Acceptable Position Error (units): 1

Controller Period (s): 0.02 Max Acceptable Velocity Error (units/s): 1.5

Max Controller Output: 12 Max Acceptable Control Effort (V): 7

Time-Normalized Controller: ☒ Loop Type: Velocity

Controller Type: Onboard kV: 0.0693

Measurement delay (ms): 0 kA: 0.117

Post-Encoder Gearing: 1 Calculate Optimal Controller Gains

Encoder EPR: 4096 kP: 3.11 Convert Gains: ☐

Has Follower: ☐ kD: 0.0

Follower Update Period (s): 0.01

26.7.5 Calculate Gains

Finally, press the *Calculate Optimal Controller Gains* to determine the feedback gains.

The screenshot shows the FRC Drive Characterization Tool interface. The top bar indicates the file path: C:/Users/piphi/Documents/WpilibProjects/FRCUploader/characterization-data20201022-01. The Units are set to Rotations, Subset to Combined, and Test to Simple. The Units per rotation is 1.

The interface is divided into two main sections: Feedforward Analysis and Feedback Analysis.

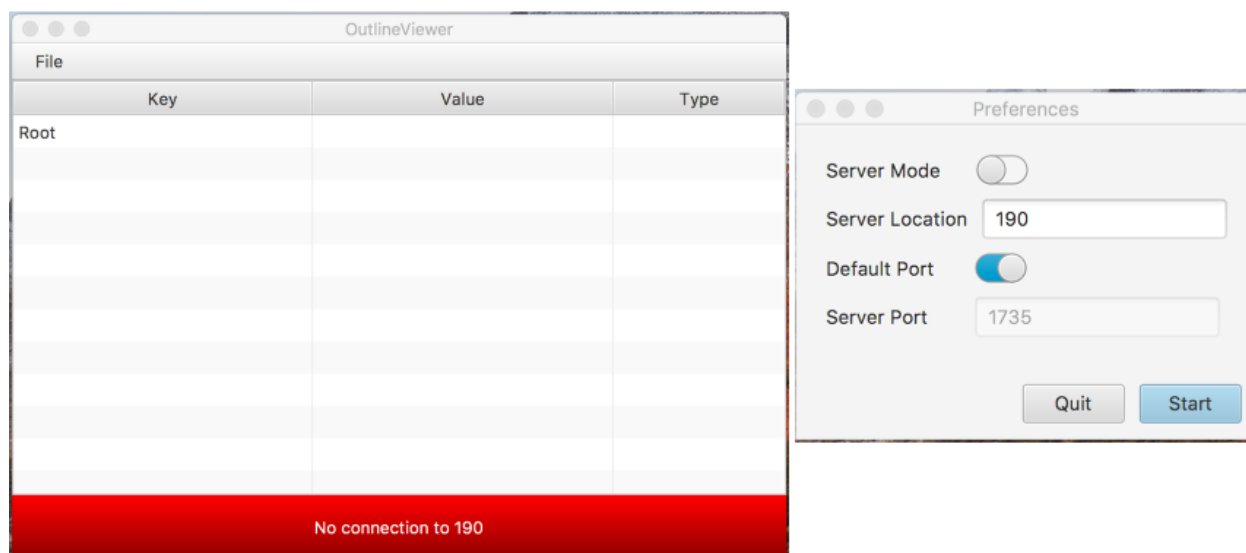
Feedforward Analysis:

- Analyze Data button
- Time-Domain Diagnostics button
- Voltage-Domain Diagnostics button
- 3D Diagnostics button
- Accel Window Size: 8
- Motion Threshold (units/s): 0.2
- kS: 0.547
- kG: 0.0
- kCos: 0.0
- kV: 0.0693
- kA: 0.117
- r-squared: 0.999
- Track Width: N/A

Feedback Analysis:

- Gain Settings Preset: Default
- Controller Period (s): 0.02
- Max Controller Output: 12
- Time-Normalized Controller: ☒
- Controller Type: Onboard
- Measurement delay (ms): 0
- Post-Encoder Gearing: 1
- Encoder EPR: 4096
- Has Follower: ☐
- Follower Update Period (s): 0.01
- Max Acceptable Position Error (units): 1
- Max Acceptable Velocity Error (units/s): 1.5
- Max Acceptable Control Effort (V): 7
- Loop Type: Velocity
- kV: 0.0693
- kA: 0.117
- kP: 3.11
- kD: 0.0
- Convert Gains: ☐
- Calculate Optimal Controller Gains** (button highlighted with a red circle)

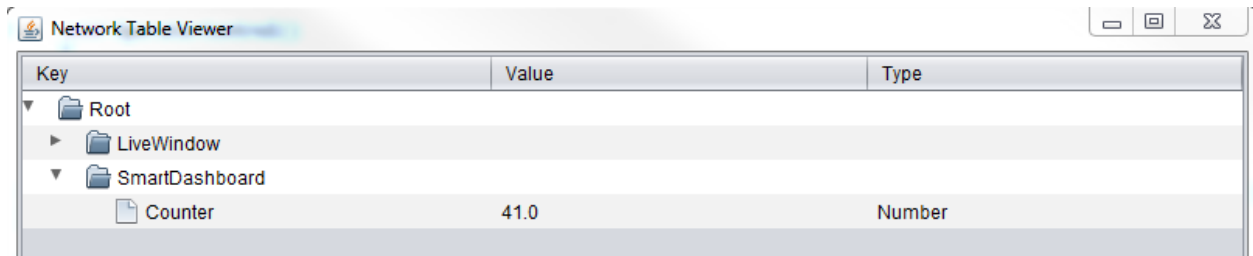
OutlineViewer



OutlineViewer is a utility used to view, modify and add to the contents of the NetworkTables for debugging purposes. It displays all key value pairs currently in the NetworkTables and can be used to modify the value of existing keys or add new keys to the table. OutlineViewer is included in the C++ and Java language updates. Teams may need to install the Java Runtime Environment to use the OutlineViewer on computers not set up for Java programming.

In Visual Studio Code, press `Ctrl+Shift+P` and type “WPILib” or click the WPILib logo in the top right to launch the WPILib Command Palette. Select *Start Tool*, then select *OutlineViewer*.

To connect to your robot, open OutlineViewer and set the “Server Location” to be your team number. After you click start, OutlineViewer will connect. If you have trouble connecting to OutlineViewer please see the [Dashboard Troubleshooting Steps](#).



To add additional key/value pairs to NetworkTables, right click on a location and choose the corresponding data type.

Note: LabVIEW teams can use the Variables tab of the LabVIEW Dashboard to accomplish the same functionality as OutlineViewer.

28.1 Vision Introduction

28.1.1 What is Vision?

Vision in FRC® uses a camera connected to the robot in order to help teams score and drive, during both the autonomous and teleoperated periods.

Vision Methods

There are two main method that most teams use for vision in FRC.

Streaming

This method involves streaming the camera to the Driver Station so that the driver and manipulator can get visual information from the robot's point of view. This method is simple and takes little time to implement, making it a good option if you do not need features of vision processing.

- *Streaming using the roboRIO*
- *Streaming using an Axis Camera*

Processing

Instead of only streaming the camera to the Driver Station, this method involves using the frames captured by the camera to compute information, such as a game piece's or target's angle and distance from the camera. This method requires more technical knowledge and time in order to implement, as well as being more computationally expensive. However, this method can help improve autonomous performance and assist in "auto-scoring" operations during the teleoperated period. This method can be done using the roboRIO or a coprocessor such as the Raspberry Pi using either OpenCV or programs such as GRIP.

- *Vision Processing with Raspberry Pi*

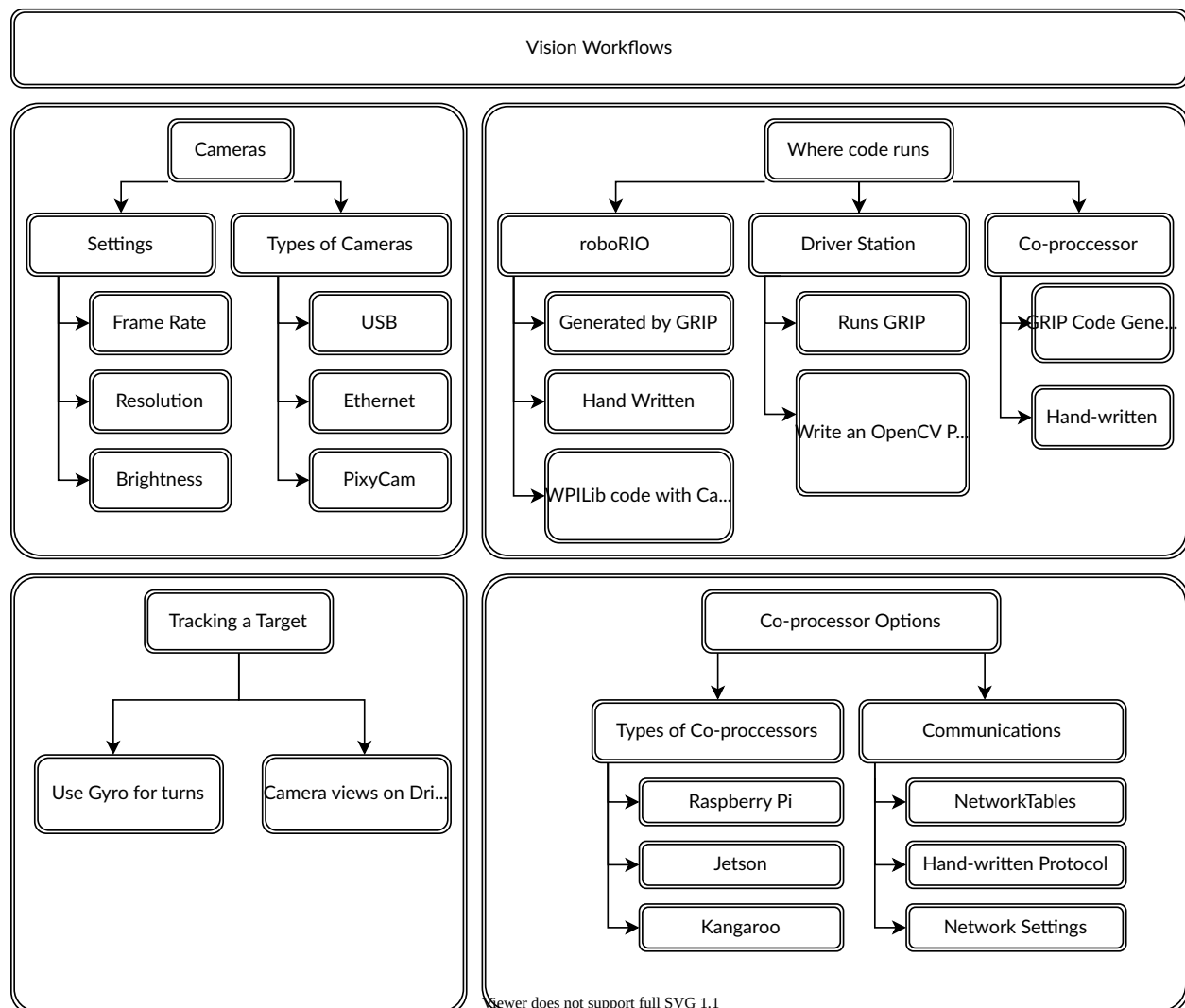
- *Vision Processing with GRIP*
- *Vision Processing with the roboRIO*

For additional information on the pros and cons of using a coprocessor for vision processing, see the next page, *Strategies for Vision Programming*.

28.1.2 Strategies for Vision Programming

Using computer vision is a great way of making your robot be responsive to the elements on the field and make it much more autonomous. Often in FRC® games there are bonus points for autonomously shooting balls or other game pieces into goals or navigating to locations on the field. Computer vision is a great way of solving many of these problems. And if you have autonomous code that can do the challenge, then it can be used during the teleop period as well to help the human drivers.

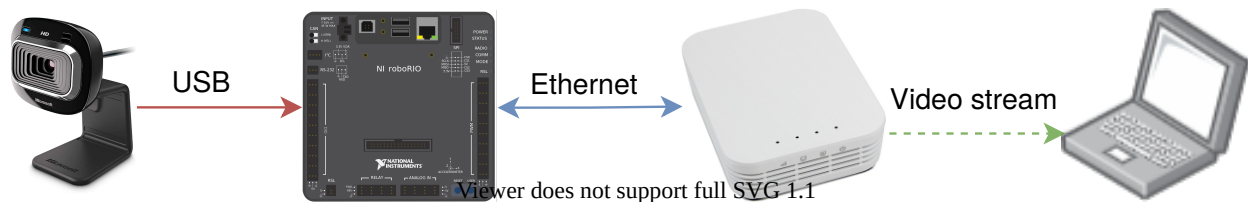
There are many options for choosing the components for vision processing and where the vision program should run. WPILib and associated tools support a number of options and give teams a lot of flexibility to decide what to do. This article will attempt to give you some insight into many of the choices and tradeoffs that are available.



OpenCV Computer Vision Library

OpenCV is an open source computer vision library that is widely used throughout academia and industry. It has support from hardware manufactures providing GPU accelerated processing, it has bindings for a number of languages including C++, Java, and Python. It is also well documented with many web sites, books, videos, and training courses so there are lots of resources available to help learn how to use it. The C++ and Java versions of WPILib include the OpenCV libraries, there is support in the library for capturing, processing and viewing video, and tools to help you create your vision algorithms. For more information about OpenCV see <https://opencv.org>.

Vision Code on roboRIO

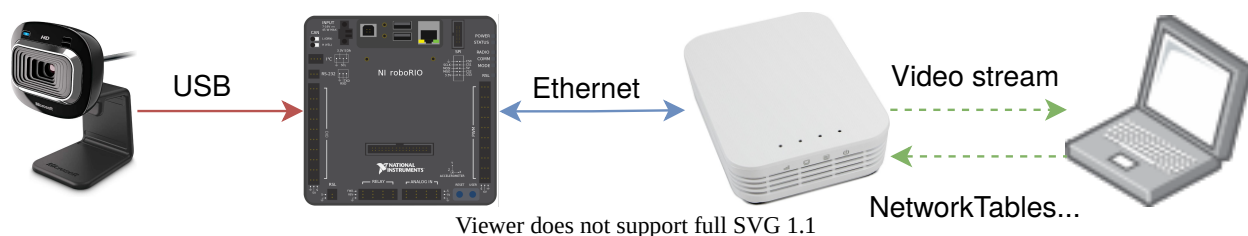


Vision code can be embedded into the main robot program on the roboRIO. Building and running the vision code is straightforward because it is built and deployed along with the robot program. The vision code can be written by hand or generated by GRIP in either C++ or Java. The disadvantage of this approach is that having vision code running on the same processor as the robot program can cause performance issues. This is something you will have to evaluate depending on the requirements for your robot and vision program.

In this approach, the vision code simply produces results that the robot code directly uses. Be careful about synchronization issues when writing robot code that is getting values from a vision thread. The GRIP generated code and the VisionRunner class in WPILib make this easier.

Using functions provided by the CameraServer class, the video stream can be sent to dashboards such as Shuffleboard so operators can see what the camera sees. In addition, annotations can be added to the images using OpenCV commands so targets or other interesting objects can be identified in the dashboard view.

Vision Code on DS Computer



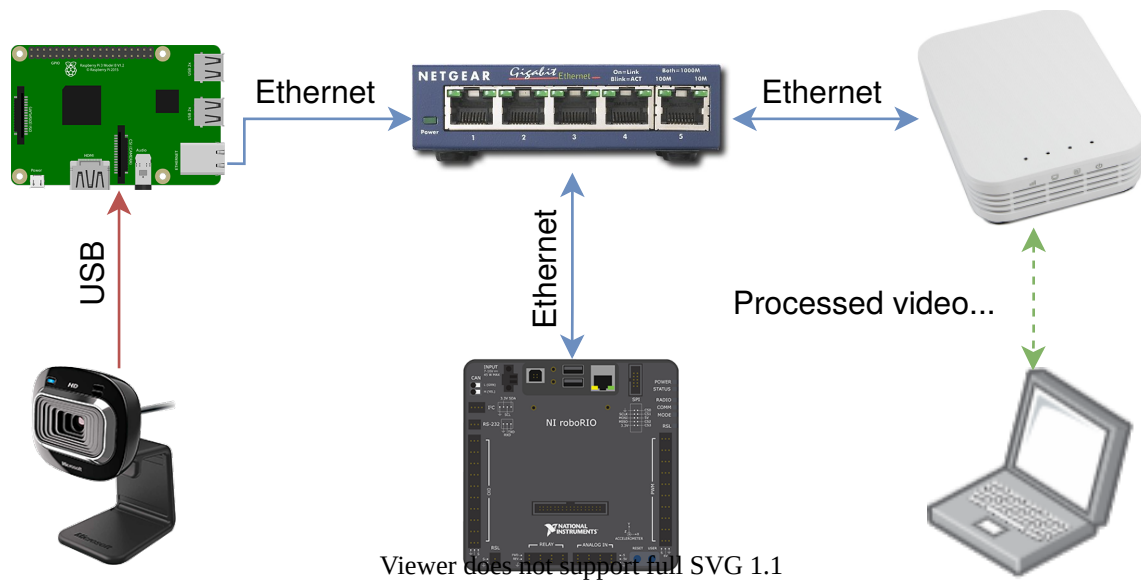
When vision code is running on the DS computer, the video is streamed back to the Driver Station laptop for processing. Even the older Classmate laptops are substantially faster at vision processing than the roboRIO. GRIP can be run on the Driver Station laptop directly with the results sent back to the robot using NetworkTables. Alternatively you can write your

own vision program using a language of your choosing. Python makes a good choice since there is a native NetworkTables implementation and the OpenCV bindings are very good.

After the images are processed, the key values such as the target position, distance or anything else you need can be sent back to the robot with NetworkTables. This approach generally has higher latency, as delay is added due to the images needing to be sent to the laptop. Bandwidth limitations also limit the maximum resolution and FPS of the images used for processing.

The video stream can be displayed on Shuffleboard or in GRIP.

Vision Code on Coprocessor



Coprocessors such as the Raspberry Pi are ideal for supporting vision code (see [Using the Raspberry Pi for FRC](#)). The advantage is that they can run full speed and not interfere with the robot program. In this case, the camera is probably connected to the coprocessor or (in the case of Ethernet cameras) an Ethernet switch on the robot. The program can be written in any language; Python is a good choice because of its simple bindings to OpenCV and NetworkTables. Some teams have used high performance vision coprocessors such as the Nvidia Jetson for fastest speed and highest resolution, although this approach generally requires advanced Linux and programming knowledge.

This approach takes a bit more programming expertise as well as a small amount of additional weight, but otherwise it brings the best of both worlds compared to the other two approaches, as coprocessors are much faster than the roboRIO and the image processing can be performed with minimal latency or bandwidth use.

Data can be sent from the vision program on the coprocessor to the robot using NetworkTables or a private protocol over a network or serial connection.

Camera Options

There are a number of camera options supported by WPILib. Cameras have a number of parameters that affect operation; for example, frame rate and image resolution affect the quality of the received images, but when set too high impact processing time and, if sent to the driver station, may exceed the available bandwidth on the field.

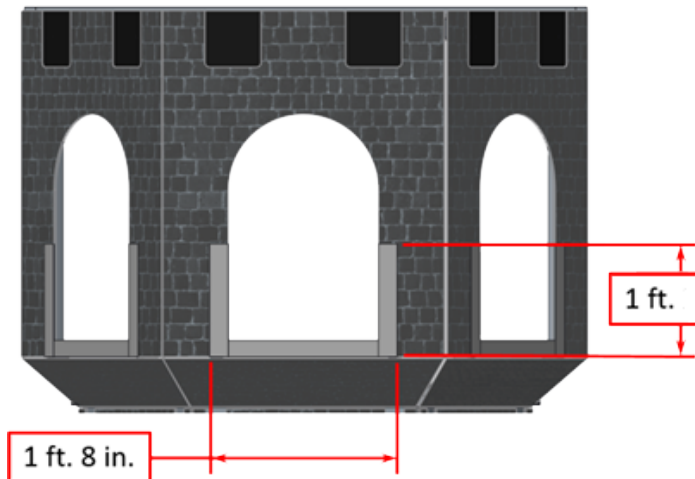
The CameraServer class in C++ and Java is used to interface with cameras connected to the robot. It retrieves frames for local processing through a Source object and sends the stream to your driver station for viewing or processing there.

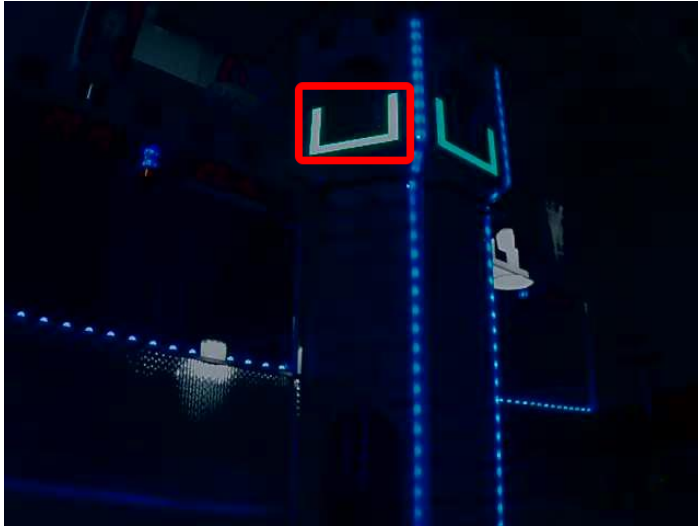
28.1.3 Target Info and Retroreflection

Many FRC® games have retroreflective tape attached to field elements to aid in vision processing. This document describes the Vision Targets from the 2016 FRC game and the visual properties of the material making up the targets.

Note: For official dimensions and drawings of all field components, please see the Official Field Drawings.

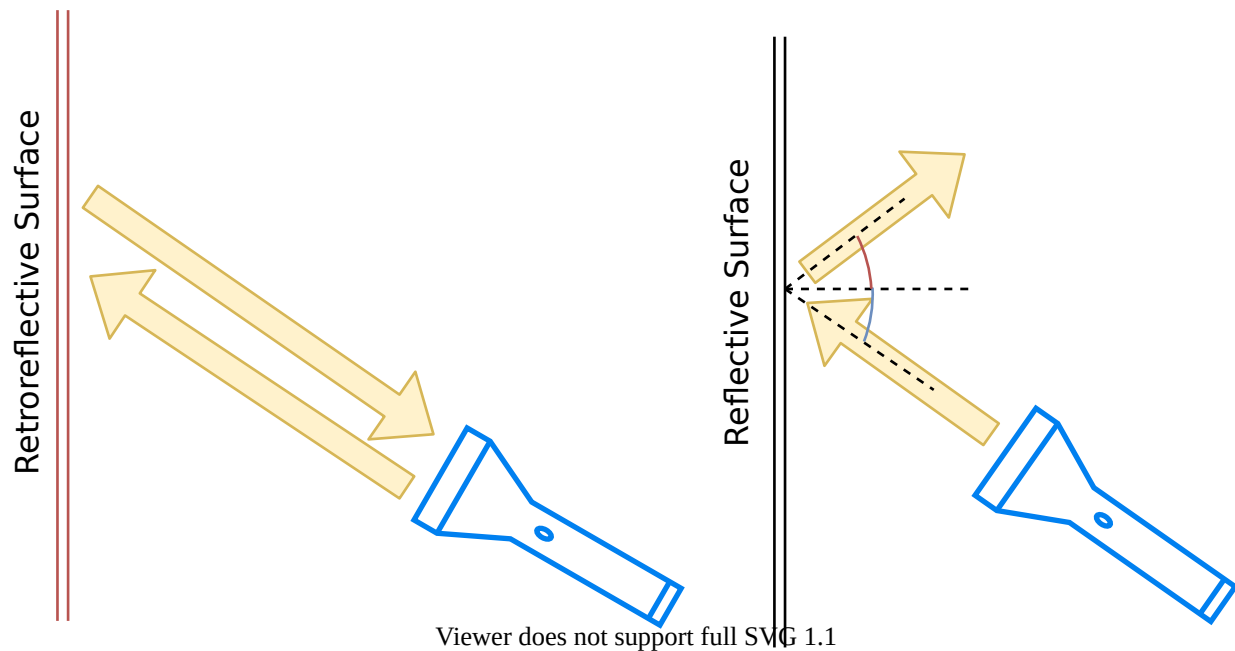
Targets





Each 2016 vision target consists of a 1' 8" wide, 1' tall U-shape made of 2" wide retroreflective material (3M 8830 Silver Marking Film). The targets are located immediately adjacent to the bottom of each high goal. When properly lit, the retroreflective tape produces a bright and/or color-saturated marker.

Retroreflectivity vs. Reflectivity

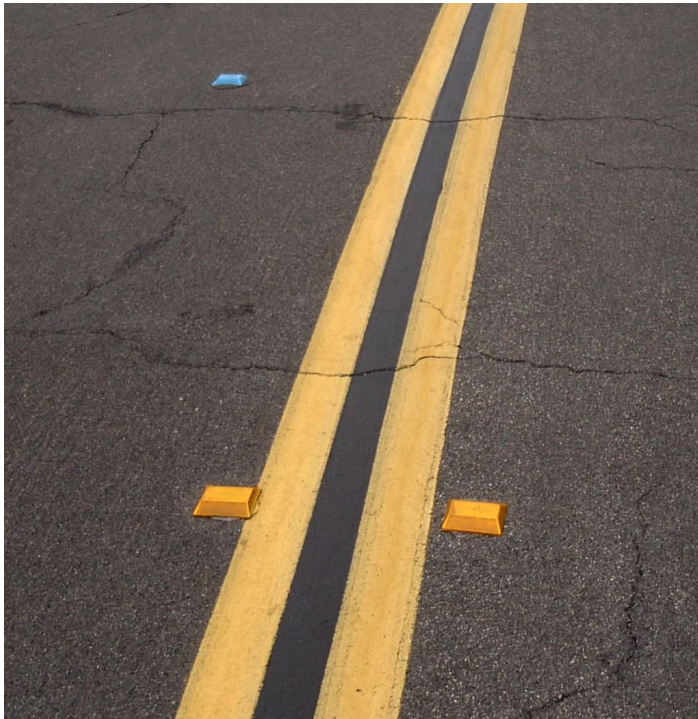


Highly reflective materials are generally mirrored so that light “bounces off” at a supplementary angle. As shown above-left, the blue and red angles sum to 180 degrees. An equivalent explanation is that the light reflects about the surface normal the green line drawn perpendicular to the surface. Notice that a light pointed at the surface will return to the light source only if the blue angle is ~ 90 degrees.

Retro-reflective materials are not mirrored, but it will typically have either shiny facets across the surface, or it will have a pearl-like appearance. Not all faceted or pearl-like materials are

retro-reflective, however. Retro-reflective materials return the majority of light back to the light source, and they do this for a wide range of angles between the surface and the light source, not just the 90 degree case. Retro-reflective materials accomplish this using small prisms, such as found on a bicycle or roadside reflector, or by using small spheres with the appropriate index of refraction that accomplish multiple internal reflections. In nature, the eyes of some animals, including house cats, also exhibit the retro-reflective effect typically referred to as night-shine. The [Wikipedia article on retroreflectors](#) goes into more detail on how retro-reflection is accomplished.

Examples of Retroreflection





This material should be relatively familiar as it is often used to enhance nighttime visibility of road signs, bicycles, and pedestrians.

Initially, retro-reflection may not seem like a useful property for nighttime safety, but when the light and eye are near one another, as shown above, the reflected light returns to the eye, and the material shines brightly even at large distances. Due to the small angle between the driver's eyes and vehicle headlights, retro-reflective materials can greatly increase visibility of distant objects during nighttime driving.

Demonstration

To further explore retro-reflective material properties:

1. Place a piece of the material on a wall or vertical surface
2. Stand 10-20 feet away, and shine a small flashlight at the material.
3. Start with the light held at your belly button, and raise it slowly until it is between your eyes. As the light nears your eyes, the intensity of the returned light will increase rapidly.
4. Alter the angle by moving to other locations in the room and repeating. The bright reflection should occur over a wide range of viewing angles, but the angle from light source to eye is key and must be quite small.

Experiment with different light sources. The material is hundreds of times more reflective than white paint; so dim light sources will work fine. For example, a red bicycle safety light will demonstrate that the color of the light source determines the color of the reflected light. If possible, position several team members at different locations, each with their own light source. This will show that the effects are largely independent, and the material can simultaneously appear different colors to various team members. This also demonstrates that the material is largely immune to environmental lighting. The light returning to the viewer is almost entirely determined by a light source they control or one directly behind them. Using

the flashlight, identify other retro-reflective articles already in your environment: on clothing, backpacks, shoes, etc.

Lighting



We have seen that the retro-reflective tape will not shine unless a light source is directed at it, and the light source must pass very near the camera lens or the observer's eyes. While there are a number of ways to accomplish this, a very useful type of light source to investigate is the ring flash, or ring light, shown above. It places the light source directly on or around the camera lens and provides very even lighting. Because of their bright output and small size, LEDs are particularly useful for constructing this type of device.

As shown above, inexpensive circular arrangements of LEDs are available in a variety of colors and sizes and are easy to attach to cameras, and some can even be powered off of a Raspberry Pi. While not designed for diffuse even lighting, they work quite well for causing retro-reflective tape to shine. A small green LED ring is available through FIRST Choice. Other similar LED rings are available from suppliers such as SuperBrightLEDs.

Sample Images

Sample images are located with the code examples for each language (packaged with LabVIEW, and in a separate ZIP in the same location as the C++/Java samples).

28.1.4 Identifying and Processing the Targets

Once an image is captured, the next step is to identify Vision Target(s) in the image. This document will walk through one approach to identifying the 2016 targets. Note that the images used in this section were taken with the camera intentionally set to underexpose the images, producing very dark images with the exception of the lit targets, see the section on Camera Settings for details.

Additional Options

This document walks through the approach used by the example code provided in LabVIEW (for PC or roboRIO), C++ and Java. In addition to these options teams should be aware of the following alternatives that allow for vision processing on the Driver Station PC or an on-board PC:

1. [RoboRealm](#)
2. SmartDashboard Camera Extension (programmed in Java, works with any robot language)
3. [GRIP](#)

Original Image

The image shown below is the starting image for the example described here. The image was taken using the green ring light available in *FIRST*® Choice combined with an additional ring light of a different size.



What is HSL/HSV?

The Hue or tone of the color is commonly seen on the artist's color wheel and contains the colors of the rainbow Red, Orange, Yellow, Green, Blue, Indigo, and Violet. The hue is specified using a radial angle on the wheel, but in imaging the circle typically contains only 256 units, starting with red at zero, cycling through the rainbow, and wrapping back to red at the upper end. Saturation of a color specifies amount of color, or the ratio of the hue color to a shade of gray. Higher ratio means more colorful, less gray. Zero saturation has no hue and is completely gray. Luminance or Value indicates the shade of gray that the hue is blended with. Black is 0 and white is 255.

The example code uses the HSV color space to specify the color of the target. The primary reason is that it readily allows for using the brightness of the targets relative to the rest of the image as a filtering criteria by using the Value (HSV) or Luminance (HSL) component. Another reason to use the HSV color system is that the thresholding operation used in the example runs more efficiently on the roboRIO when done in the HSV color space.

Masking

In this initial step, pixel values are compared to constant color or brightness values to create a binary mask shown below in yellow. This single step eliminates most of the pixels that are not part of a target's retro-reflective tape. Color based masking works well provided the color is relatively saturated, bright, and consistent. Color inequalities are generally more accurate when specified using the HSL (Hue, Saturation, and Luminance) or HSV (Hue, Saturation, and Value) color space than the RGB (Red, Green, and Blue) space. This is especially true when the color range is quite large in one or more dimension.

Notice that in addition to the target, other bright parts of the image (overhead light and tower lighting) are also caught by the masking step.



Particle Analysis

After the masking operation, a particle report operation is used to examine the area, bounding rectangle, and equivalent rectangle for the particles. These are used to compute several scored terms to help pick the shapes that are most rectangular. Each test described below generates a score (0-100) which is then compared to pre-defined score limits to decide if the particle is a target or not.

Coverage Area

The Area score is calculated by comparing the area of the particle compared to the area of the bounding box drawn around the particle. The area of the retroreflective strips is 80 square inches ($\sim 516 \text{ cm}^2$). The area of the rectangle that contains the target is 240 square inches ($\sim 0.15 \text{ m}^2$). This means that the ideal ratio between area and bounding box area is 1/3. Area ratios close to 1/3 will produce a score near 100, as the ratio diverges from 1/3 the score will approach 0.

Aspect Ratio

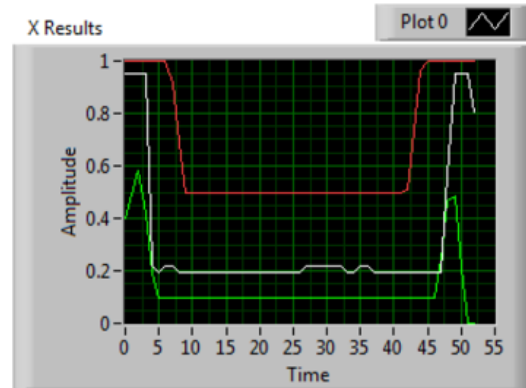
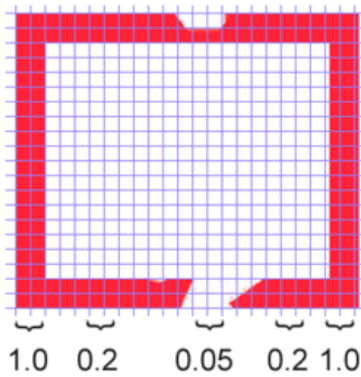
The aspect ratio score is based on (Particle Width / Particle Height). The width and height of the particle are determined using something called the “equivalent rectangle”. The equivalent rectangle is the rectangle with side lengths x and y where $2x + 2y$ equals the particle perimeter and $x \cdot y$ equals the particle area. The equivalent rectangle is used for the aspect ratio calculation as it is less affected by skewing of the rectangle than using the bounding box. When using the bounding box rectangle for aspect ratio, as the rectangle is skewed the height increases and the width decreases.

The target is 20” (508 mm) wide by 12” (304.8 mm) tall, for a ratio of 1.6. The detected aspect ratio is compared to this ideal ratio. The aspect ratio score is normalized to return 100 when the ratio matches the target ratio and drops linearly as the ratio varies below or above.

Moment

The moment measurement calculates the particles moment of inertia about it’s center of mass. This measurement provides a representation of the pixel distribution in the particle. The ideal score for this test is ~ 0.28 . See: [Moment of Inertia](#)

X/Y Profiles



White line is the average, red is upper limit, and green is lower limit..

The edge score describes whether the particle matches the appropriate profile in both the X and Y directions. As shown, it is calculated using the row and column averages across the bounding box extracted from the original image and comparing that to a profile mask. The score ranges from 0 to 100 based on the number of values within the row or column averages that are between the upper and lower limit values.

Measurements

If a particle scores well enough to be considered a target, it makes sense to calculate some real-world measurements such as position and distance. The example code includes these basic measurements, so let's look at the math involved to better understand it.

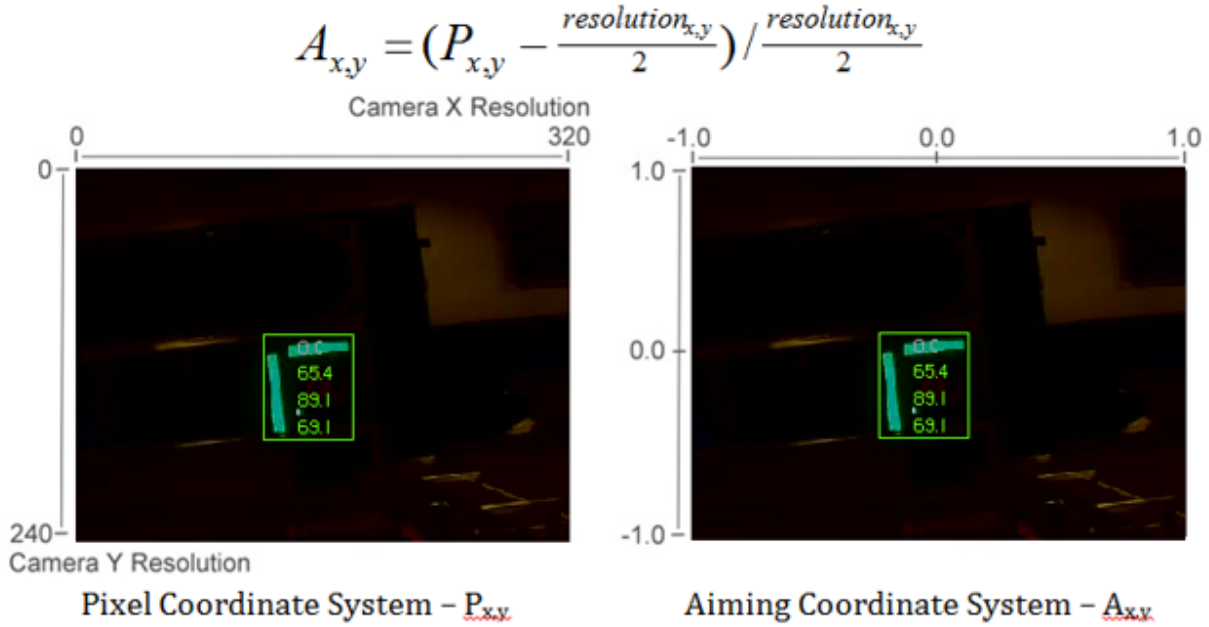
Position

The target position is well described by both the particle and the bounding box, but all coordinates are in pixels with 0,0 being at the top left of the screen and the right and bottom edges determined by the camera resolution. This is a useful system for pixel math, but not nearly as useful for driving a robot; so let's change it to something that may be more useful.

To convert a point from the pixel system to the aiming system, we can use the formula shown below.

The resulting coordinates are close to what you may want, but the Y axis is inverted. This could be corrected by multiplying the point by [1,-1] (Note: this is not done in the sample code). This coordinate system is useful because it has a centered origin and the scale is similar to joystick outputs and RobotDrive inputs.

$$A_{x,y} = \left(P_{x,y} - \frac{\text{resolution}_{x,y}}{2} \right) / \frac{\text{resolution}_{x,y}}{2}$$



Field of View

You can use known constants and the position of the target on the coordinate plane to determine your distance, yaw, and pitch from the target. However, in order to calculate these, you must determine your FOV (field of view). In order to empirically determine vertical field of view, set your camera a set distance away from an flat surface, and measure the distance between the topmost and bottommost row of pixels.

$$\frac{1}{2}FOV_{vertical} = \tan\left(\frac{\frac{1}{2}distance_y}{distance_z}\right)$$

You can find the horizontal FOV using the same method, but using the distance between the first and last column of pixels.

Pitch and Yaw

Finding the pitch and yaw of the target relative to your robot is simple once you know your FOVs and the location of your target in the aiming coordinate system.

$$pitch = \frac{A_y}{2}FOV_{vertical}$$

$$yaw = \frac{A_x}{2}FOV_{horizontal}$$

Distance

If your target is at a significantly different height than your robot, you can use known constants, such as the physical height of the target and your camera, as well as the angle your camera is mounted, to calculate the distance between your camera and the target.

$$distance = \frac{height_{target} - height_{camera}}{\tan(angle_{camera} + pitch)}$$

Another option is to create a lookup table for area to distance, or to estimate the inverse variation constant of area and distance. However, this method is less accurate.

Note: For best results for the above methods of estimating angle and distance, you can calibrate your camera using OpenCV to get rid of any distortions that may be affecting accuracy by reprojecting the pixels of the target using the calibration matrix.

28.1.5 Read and Process Video: CameraServer Class

Concepts

The cameras typically used in FRC® (commodity USB and Ethernet cameras such as the Axis camera) offer relatively limited modes of operation. In general, they provide only a single image output (typically in an RGB compressed format such as JPG) at a single resolution and frame rate. USB cameras are particularly limited as only one application may access the camera at a time.

CameraServer supports multiple cameras. It handles details such as automatically reconnecting when a camera is disconnected, and also makes images from the camera available to multiple “clients” (e.g. both your robot code and the dashboard can connect to the camera simultaneously).

Camera Names

Each camera in CameraServer must be uniquely named. This is also the name that appears for the camera in the Dashboard. Some variants of the CameraServer `startAutomaticCapture()` and `addAxisCamera()` functions will automatically name the camera (e.g. “USB Camera 0” or “Axis Camera”), or you can give the camera a more descriptive name (e.g. “Intake Cam”). The only requirement is that each camera have a unique name.

USB Camera Notes

CPU Usage

The CameraServer is designed to minimize CPU usage by only performing compression and decompression operations when required and automatically disabling streaming when no clients are connected.

To minimize CPU usage, the dashboard resolution should be set to the same resolution as the camera; this allows the CameraServer to not decompress and recompress the image, instead, it can simply forward the JPEG image received from the camera directly to the dashboard. It’s

important to note that changing the resolution on the dashboard does *not* change the camera resolution; changing the camera resolution may be done by calling `setResolution()` on the camera object.

USB Bandwidth

The roboRIO can only transmit and receive so much data at a time over its USB interfaces. Camera images can require a lot of data, and so it is relatively easy to run into this limit. The most common cause of a USB bandwidth error is selecting a non-JPEG video mode or running too high of a resolution, particularly when multiple cameras are connected.

Architecture

The CameraServer consists of two layers, the high level WPILib **CameraServer class** and the low level **cscore library**.

CameraServer Class

The CameraServer class (part of WPILib) provides a high level interface for adding cameras to your robot code. It also is responsible for publishing information about the cameras and camera servers to NetworkTables so that Driver Station dashboards such as the LabVIEW Dashboard and Shuffleboard can list the cameras and determine where their streams are located. It uses a singleton pattern to maintain a database of all created cameras and servers.

Some key functions in CameraServer are:

- `startAutomaticCapture()`: Add a USB camera (e.g. Microsoft LifeCam) and starts a server for it so it can be viewed from the dashboard.
- `addAxisCamera()`: Add an Axis camera. Even if you aren't processing images from the Axis camera in your robot code, you may want to use this function so that the Axis camera appears in the Dashboard's drop down list of cameras. It also starts a server so the Axis stream can still be viewed when your driver station is connected to the roboRIO via USB (useful at competition if you have both the Axis camera and roboRIO connected to the two robot radio Ethernet ports).
- `getVideo()`: Get OpenCV access to a camera. This allows you to get images from the camera for image processing on the roboRIO (in your robot code).
- `putVideo()`: Start a server that you can feed OpenCV images to. This allows you to pass custom processed and/or annotated images to the dashboard.

cscore Library

The cscore library provides the lower level implementation to:

- Get images from USB and HTTP (e.g. Axis) cameras
- Change camera settings (e.g. contrast and brightness)
- Change camera video modes (pixel format, resolution and frame rate)
- Act as a web server and serve images as a standard MJPEG stream
- Convert images to/from OpenCV Mat objects for image processing

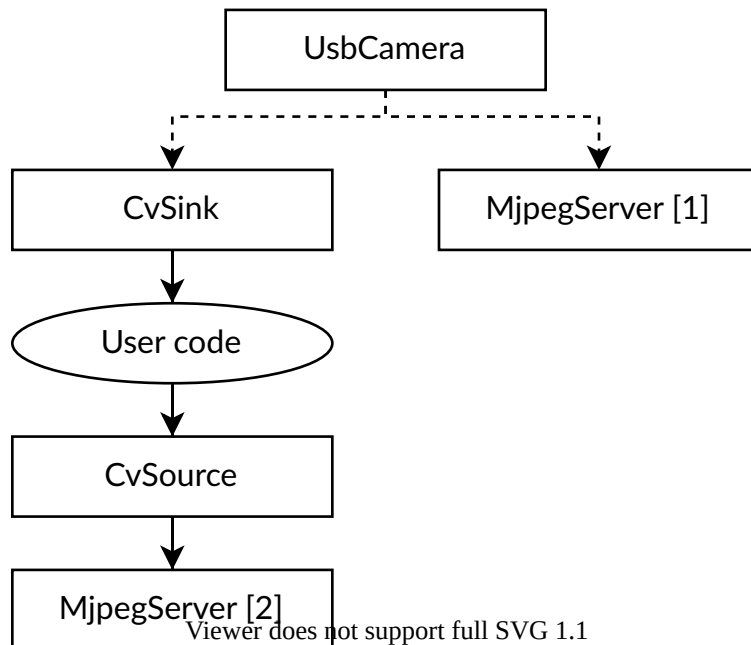
Sources and Sinks

The basic architecture of the cscore library is similar to that of MJPGStreamer, with functionality split between sources and sinks. There can be multiple sources and multiple sinks created and operating simultaneously.

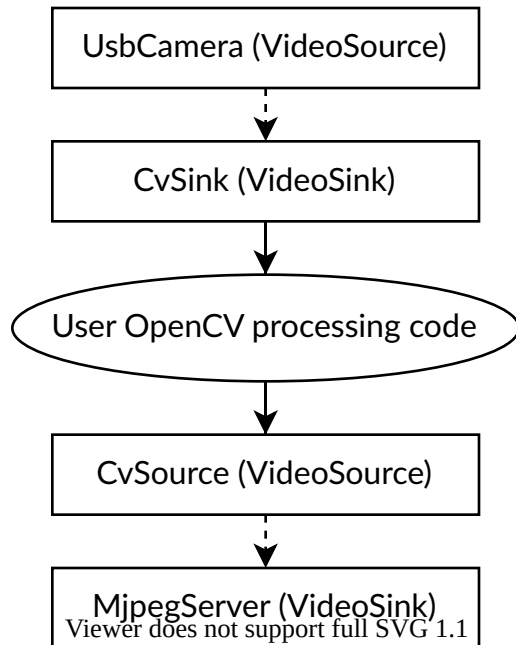
An object that generates images is a source and an object that accepts/consumes images is a sink. The generate/consume is from the perspective of the library. Thus cameras are sources (they generate images). The MJPEG web server is a sink because it accepts images from within the program (even though it may be forwarding those images on to a web browser or dashboard). Sources may be connected to multiple sinks, but sinks can be connected to one and only one source. When a sink is connected to a source, the cscore library takes care of passing each image from the source to the sink.

- **Sources** obtain individual frames (such as provided by a USB camera) and fire an event when a new frame is available. If no sinks are listening to a particular source, the library may pause or disconnect from a source to save processor and I/O resources. The library autonomously handles camera disconnects/reconnects by simply pausing and resuming firing of events (e.g. a disconnect results in no new frames, not an error).
- **Sinks** listen to a particular source's event, grab the latest image, and forward it to its destination in the appropriate format. Similarly to sources, if a particular sink is inactive (e.g. no client is connected to a configured MJPEG over HTTP server), the library may disable parts of its processing to save processor resources.

User code (such as that used in a FRC robot program) can act as either a source (providing processed frames as if it were a camera) or as a sink (receiving a frame for processing) via OpenCV source and sink objects. Thus an image processing pipeline that gets images from a camera and serves the processed images out looks like the below graph:



Because sources can have multiple sinks connected, the pipeline may branch. For example, the original camera image can also be served by connecting the UsbCamera source to a second MjpegServer sink in addition to the CvSink, resulting in the below graph:



When a new image is captured by the camera, both the CvSink and the MjpegServer [1] receive it.

The above graph is what the following CameraServer snippet creates:

Java

C++

```

import edu.wpi.first.cameraserver.CameraServer;
import edu.wpi.first.cscore.CvSink;
import edu.wpi.first.cscore.CvSource;

// Creates UsbCamera and MjpegServer [1] and connects them
CameraServer.getInstance().startAutomaticCapture();

// Creates the CvSink and connects it to the UsbCamera
CvSink cvSink = CameraServer.getInstance().getVideo();

// Creates the CvSource and MjpegServer [2] and connects them
CvSource outputStream = CameraServer.getInstance().putVideo("Blur", 640, 480);

```

```

#include "cameraserver/CameraServer.h"

// Creates UsbCamera and MjpegServer [1] and connects them
frc::CameraServer::GetInstance().StartAutomaticCapture();

// Creates the CvSink and connects it to the UsbCamera
cs::CvSink cvSink = frc::CameraServer::GetInstance().GetVideo();

// Creates the CvSource and MjpegServer [2] and connects them
cs::CvSource outputStream = frc::CameraServer::GetInstance().PutVideo("Blur", 640,
↪480);

```

The CameraServer implementation effectively does the following at the cscore level (for explanation purposes). CameraServer takes care of many of the details such as creating unique

names for all cscore objects and automatically selecting port numbers. CameraServer also keeps a singleton registry of created objects so they aren't destroyed if they go out of scope.

Java

C++

```
import edu.wpi.cscore.CvSink;
import edu.wpi.cscore.CvSource;
import edu.wpi.cscore.MjpegServer;
import edu.wpi.cscore.UsbCamera;

// Creates UsbCamera and MjpegServer [1] and connects them
UsbCamera usbCamera = new UsbCamera("USB Camera 0", 0);
MjpegServer mjpegServer1 = new MjpegServer("serve_USB Camera 0", 1181);
mjpegServer1.setSource(usbCamera);

// Creates the CvSink and connects it to the UsbCamera
CvSink cvSink = new CvSink("opencv_USB Camera 0");
cvSink.setSource(usbCamera);

// Creates the CvSource and MjpegServer [2] and connects them
CvSource outputStream = new CvSource("Blur", PixelFormat.kMJPEG, 640, 480, 30);
MjpegServer mjpegServer2 = new MjpegServer("serve_Blur", 1182);
mjpegServer2.setSource(outputStream);
```

```
#include "cscore_oo.h"

// Creates UsbCamera and MjpegServer [1] and connects them
cs::UsbCamera usbCamera("USB Camera 0", 0);
cs::MjpegServer mjpegServer1("serve_USB Camera 0", 1181);
mjpegServer1.SetSource(usbCamera);

// Creates the CvSink and connects it to the UsbCamera
cs::CvSink cvSink("opencv_USB Camera 0");
cvSink.SetSource(usbCamera);

// Creates the CvSource and MjpegServer [2] and connects them
cs::CvSource outputStream("Blur", cs::PixelFormat::kJPEG, 640, 480, 30);
cs::MjpegServer mjpegServer2("serve_Blur", 1182);
mjpegServer2.SetSource(outputStream);
```

Reference Counting

All cscore objects are internally reference counted. Connecting a sink to a source increments the source's reference count, so it's only strictly necessary to keep the sink in scope. The CameraServer class keeps a registry of all objects created with CameraServer functions, so sources and sinks created in that way effectively never go out of scope (unless explicitly removed).

28.1.6 2017 Vision Examples

LabVIEW

The 2017 LabVIEW Vision Example is included with the other LabVIEW examples. From the Splash screen, click Support->Find FRC® Examples or from any other LabVIEW window, click Help->Find Examples and locate the Vision folder to find the 2017 Vision Example. The example images are bundled with the example.

C++/Java

We have provided a GRIP project and the description below, as well as the example images, bundled into a ZIP that [can be found on TeamForge](#).

See [Using Generated Code in a Robot Program](#) for details about integrating GRIP generated code in your robot program.

The code generated by the included GRIP project will find OpenCV contours for green particles in images like the ones included in the Vision Images folder of this ZIP. From there you may wish to further process these contours to assess if they are the target. To do this:

1. Use the boundingRect method to draw bounding rectangles around the contours
2. The LabVIEW example code calculates 5 separate ratios for the target. Each of these ratios should nominally equal 1.0. To do this, it sorts the contours by size, then starting with the largest, calculates these values for every possible pair of contours that may be the target, and stops if it finds a target or returns the best pair it found.

In the formulas below, each letter refers to a coordinate of the bounding rect (H = Height, L = Left, T = Top, B = Bottom, W = Width) and the numeric subscript refers to the contour number (1 is the largest contour, 2 is the second largest, etc).

- Top height should be 40% of total height (4 in / 10 in):

$$\text{Group Height} = \frac{H_1}{0.4(B_2 - T_1)}$$

- Top of bottom stripe to top of top stripe should be 60% of total height (6 in / 10 in):

$$d\text{Top} = \frac{T_2 - T_1}{0.6(B_2 - T_1)}$$

- The distance between the left edge of contour 1 and the left edge of contour 2 should be small relative to the width of the 1st contour; then we add 1 to make the ratio centered on 1:

$$L\text{Edge} = \frac{L_1 - L_2}{W_1} + 1$$

- The widths of both contours should be about the same:

$$\text{Width ratio} = \frac{W_1}{W_2}$$

- The larger stripe should be twice as tall as the smaller one

$$\text{Height ratio} = \frac{H_1}{2H_2}$$

Each of these ratios is then turned into a 0-100 score by calculating:

$$100 - (100 \cdot \text{abs}(1 - \text{Val}))$$

3. To determine distance, measure pixels from top of top bounding box to bottom of bottom bounding box:

$$distance = \frac{Target\ height\ in\ ft.(10/12) \cdot YRes}{2 \cdot PixelHeight \cdot \tan(viewAngle\ of\ camera)}$$

The LabVIEW example uses height as the edges of the round target are the most prone to noise in detection (as the angle points further from the camera the color looks less green). The downside of this is that the pixel height of the target in the image is affected by perspective distortion from the angle of the camera. Possible fixes include:

- Try using width instead
- Empirically measure height at various distances and create a lookup table or regression function
- Mount the camera to a servo, center the target vertically in the image and use servo angle for distance calculation (you'll have to work out the proper trig yourself or find a math teacher to help!)
- Correct for the perspective distortion using OpenCV. To do this you will need to [calibrate your camera with OpenCV](#). This will result in a distortion matrix and camera matrix. You will take these two matrices and use them with the `undistortPoints` function to map the points you want to measure for the distance calculation to the "correct" coordinates (this is much less CPU intensive than undistorting the whole image)

28.2 Vision with WPILibPi

28.2.1 A Video Walkthrough of using WPILibPi with the Raspberry Pi

Note: The video mentions FRCVision which is the old name of WPILibPi.

At the "RSN Spring Conference, Presented by WPI" in 2020, Peter Johnson from the WPILib team gave a presentation on FRC® Vision with a Raspberry Pi.

The link to the presentation is available [here](#).

28.2.2 Using a Coprocessor for vision processing

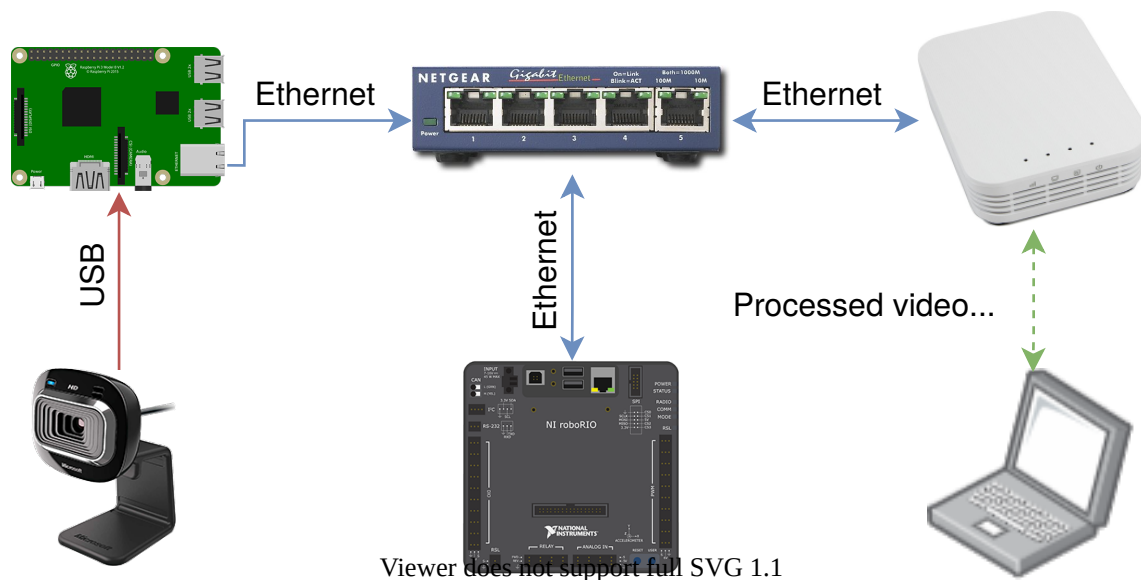
Vision processing using libraries like OpenCV for recognizing field targets or game pieces can often be a CPU intensive process. Often the load isn't too significant and the processing can easily be handled by the roboRIO. In cases where there are more camera streams or the image processing is complex, it is desirable to off-load the roboRIO by putting the code and the camera connection on a different processor. There are a number of choices of processors that are popular in FRC® such as the Raspberry Pi, the intel-based Kangaroo, the LimeLight for the ultimate in simplicity, or for more complex vision code a graphics accelerator such as one of the nVidia Jetson models.

Strategy

Generally the idea is to set up the coprocessor with the required software that generally includes:

- OpenCV - the open source computer vision library
- *NetworkTables* - to commute the results of the image processing to the roboRIO program
- Camera server library - to handle the camera connections and publish streams that can be viewed on a dashboard
- The language library for whatever computer language is used for the vision program
- The actual vision program that does the object detection

The coprocessor is connected to the roboRIO network by plugging it into the extra ethernet port on the network router or, for more connections, adding a small network switch to the robot. The cameras are plugged into the coprocessor, it acquires the images, processes them, and publishes the results, usually target location information, to NetworkTables so it is can be consumed by the robot program for steering and aiming.



Streaming camera data to the dashboard

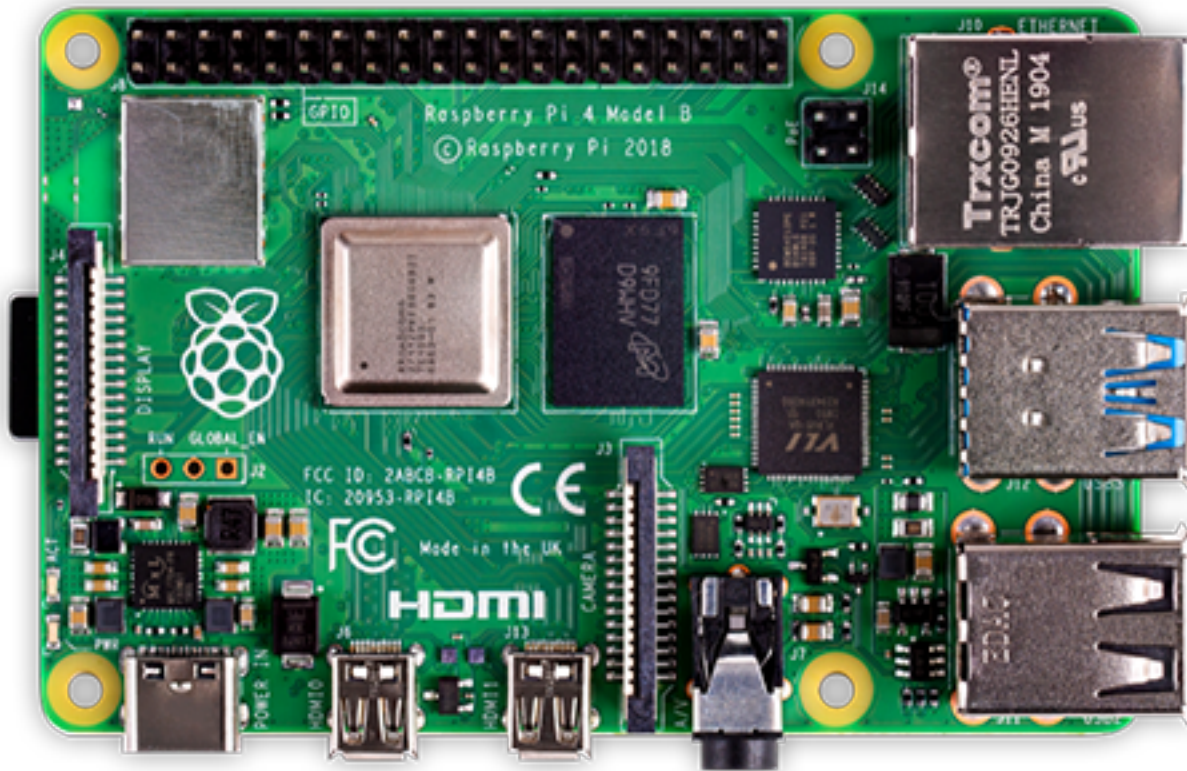
It is often desirable to simply stream the camera data to the dashboard over the robot network. In this case one or more camera connections can be sent to the network and viewed on a dashboard such as Shuffleboard or a web browser. Using Shuffleboard has the advantage of having easy controls to set the camera resolution and bit rate as well as integrating the camera streams with other data sent from the robot.

It is also possible to process images and add annotation to the image, such as target lines or boxes showing what the image processing code has detected then send it forward to the dashboard to make it easier for operators to see a clear picture of what's around the robot.

28.2.3 Using the Raspberry Pi for FRC

One of the most popular coprocessor choices is the Raspberry Pi because:

- Low cost - around \$35
- High availability - it's easy to find Raspberry Pis from a number of suppliers, including Amazon
- Very good performance - the current Raspberry Pi 3b+ has the following specifications:
- Technical Specifications: - Broadcom BCM2837BO 64 bit ARMv8 QUAD Core A53 64bit Processor powered Single Board Computer run at 1.4GHz - 1GB RAM - BCM43143 WiFi on board - Bluetooth Low Energy (BLE) on board - 40 pin extended GPIO - 4 x USB2 ports - 4 pole Stereo output and Composite video port - Full size HDMI - CSI camera port for connecting the Raspberry - Pi camera - DSI display port for connecting the Raspberry - Pi touch screen display - MicroSD port for loading your operating system and storing data - Upgraded switched Micro USB power source (now supports up to 2.5 Amps).



Pre-built Raspberry Pi image

To make using the Raspberry Pi as easy as possible for teams, there is a provided Raspberry Pi image. The image can be copied to a micro SD card, inserted into the Pi, and booted. By default it supports:

- A web interface for configuring it for the most common functions
- Supports an arbitrary number camera streams (defaults to one) that are published on the network interface
- OpenCV, [NetworkTables](#), Camera Server, and language libraries for C++, Java, and Python custom programs

If the only requirement is to stream one or more cameras to the network (and dashboard) then no programming is required and can be completely set up through the web interface.

The next section discusses how to install the image onto a flash card and boot the Pi.

28.2.4 What you need to get the Pi image running

To start using the Raspberry Pi as a video or image coprocessor you need the following:

- A Raspberry Pi 3 B, Raspberry Pi 3 B+, or a Raspberry Pi 4 B
- A micro SD card that is at least 8 GB to hold all the provided software, with a recommended Speed Class of 10 (10MB/s)
- An ethernet cable to connect the Pi to your roboRIO network
- A USB micro power cable to connect to the Voltage Regulator Module (VRM) on your robot. It is recommended to use the VRM connection for power rather than powering it from one of the roboRIO USB ports for higher reliability
- A laptop that can write the MicroSD card, either using a USB dongle (preferred) or a SD to MicroSD adapter that ships with most MicroSD cards



Shown is an inexpensive USB dongle that will write the FRC® image to the MicroSD card.

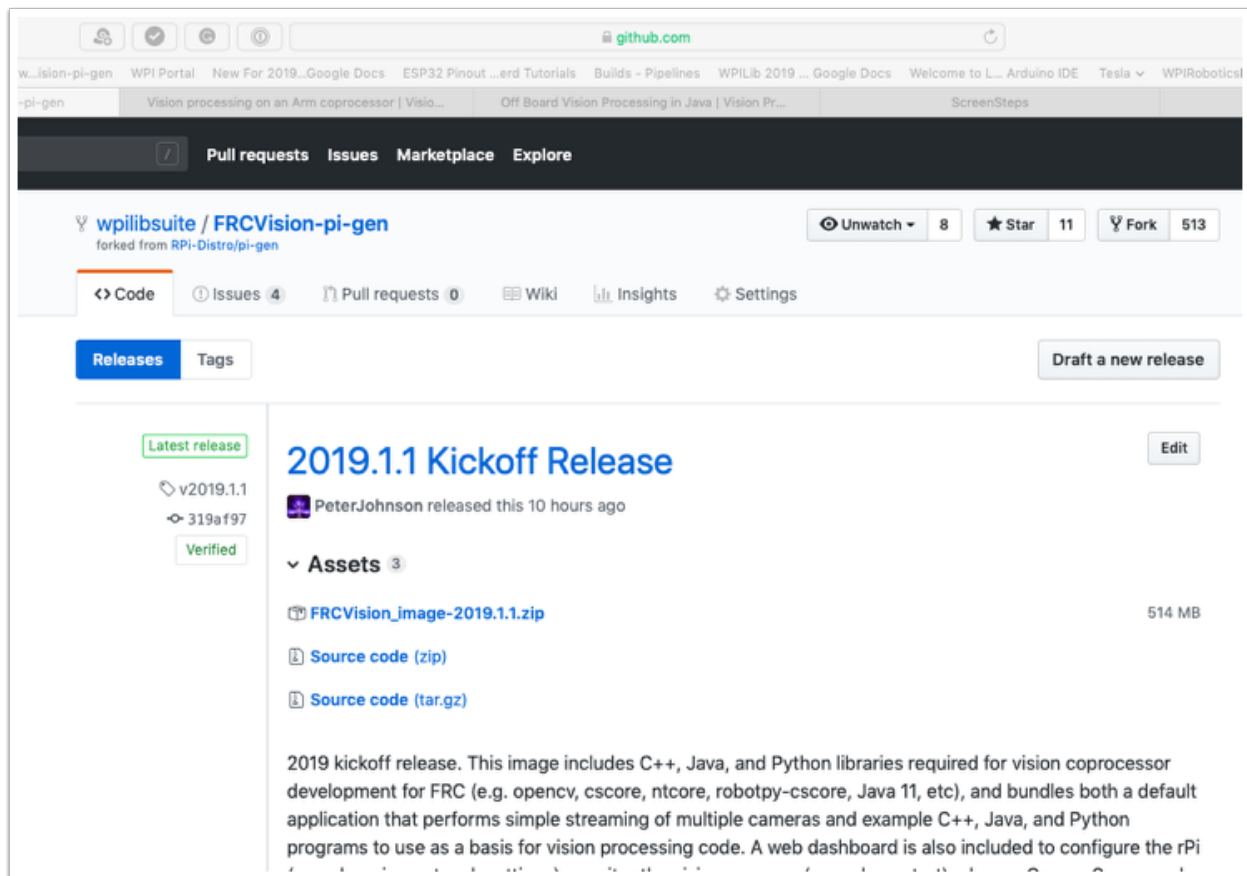
28.2.5 Installing the image to your MicroSD card

Getting the FRC Raspberry PI image

The image is stored on the GitHub release page for the WPILibPi [repository](#).

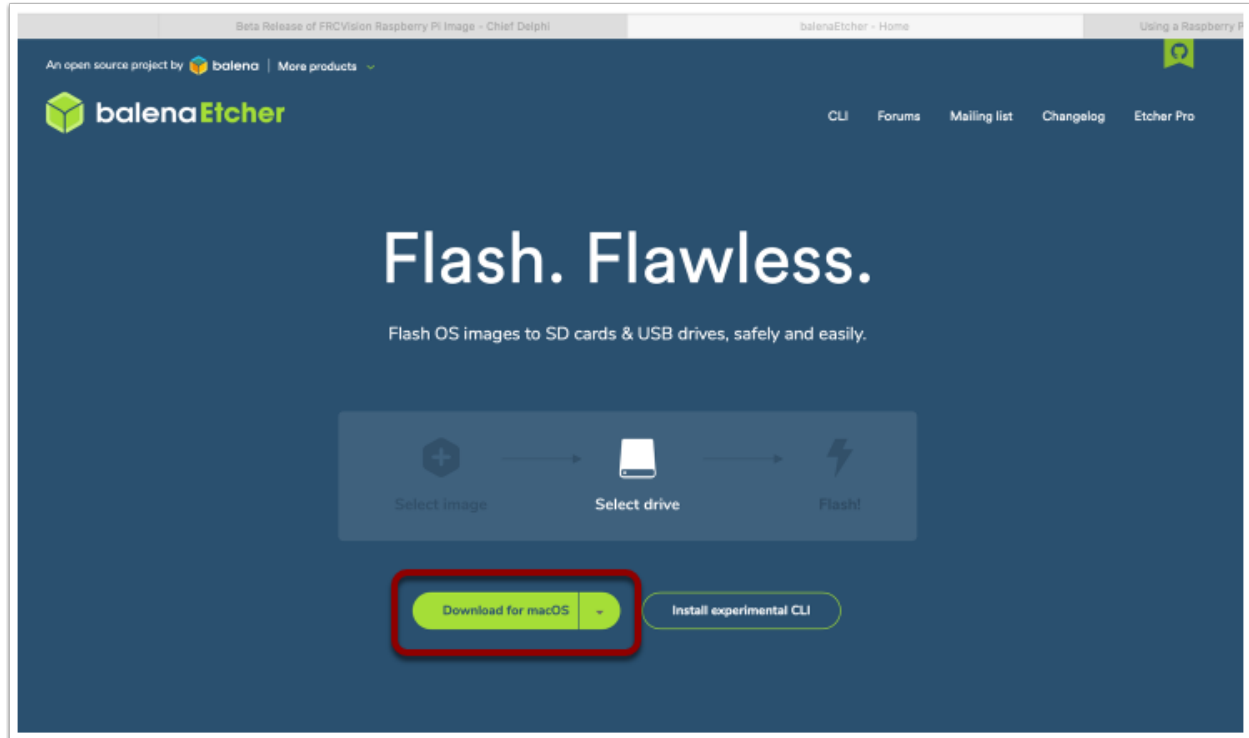
In addition to the instructions on this page, see the documentation on the GitHub web page (below).

The image is fairly large so have a fast internet connection when downloading it. Always use the most recent release from the top of the list of releases.



Copy the image to your MicroSD card

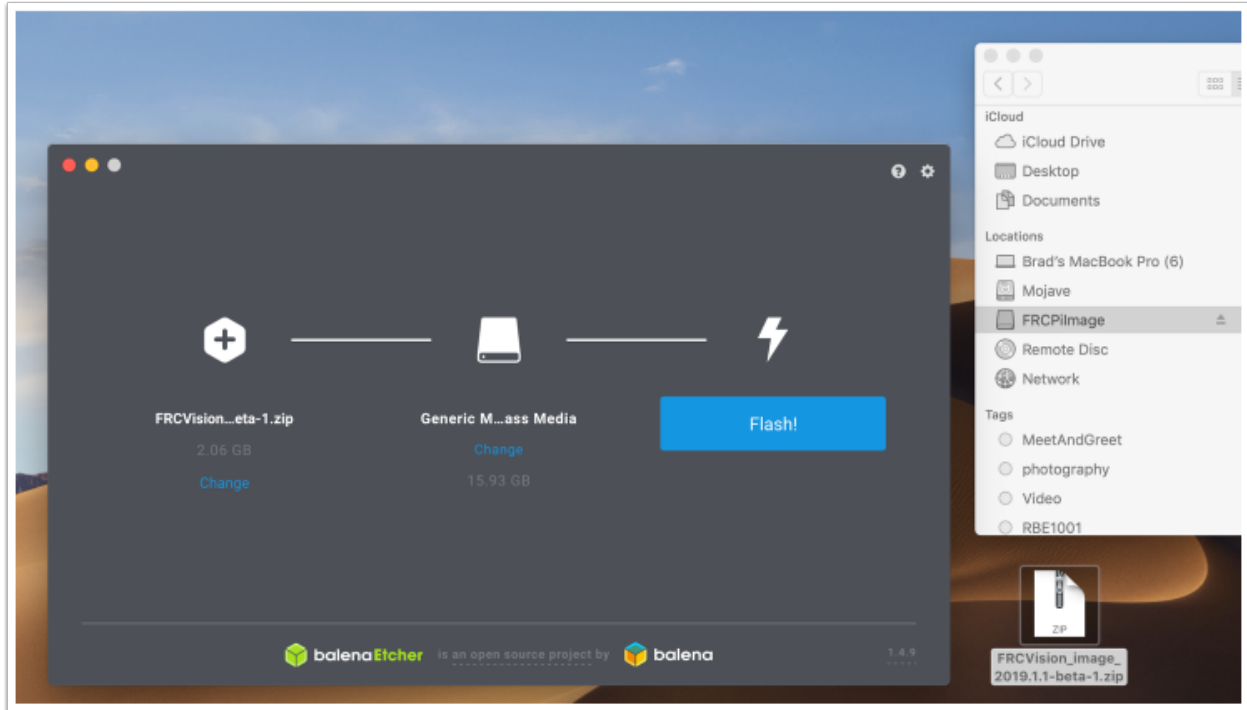
Download and install Etcher (<https://www.balena.io/etcher/>) to image the micro SD card. The micro SD card needs to be at least 4 GB. Note: a micro SD to USB dongle such as <https://www.amazon.com/gp/product/B0779V61XB> works well for writing to micro SD cards.



Flash the MicroSD card with the image using Etcher by selecting the zip file as the source, your SD card as the destination and click “Flash”. Expect the process to take about 3 minutes on a fairly fast laptop.

Testing the Raspberry PI

1. Put the micro SD card in a rPi 3 and apply power.
2. Connect the rPi 3 ethernet to a LAN or PC. Open a web browser and connect to <http://wpilibpi.local/> to open the web dashboard. On the first bootup the filesystem will be writable, but later bootups will default to read only, so it's necessary to click the “writable” button to make changes.



Logging into the Raspberry PI

Most tasks with the rPi can be done from the web console interface. Sometimes for advanced use such as program development on the rPi it is necessary to log in. To log in, use the default Raspberry PI password:

Username: pi
Password: raspberry

28.2.6 The Raspberry PI

FRC Console

The FRC® image for the Raspberry PI includes a console that can be viewed in any web browser that makes it easy to:

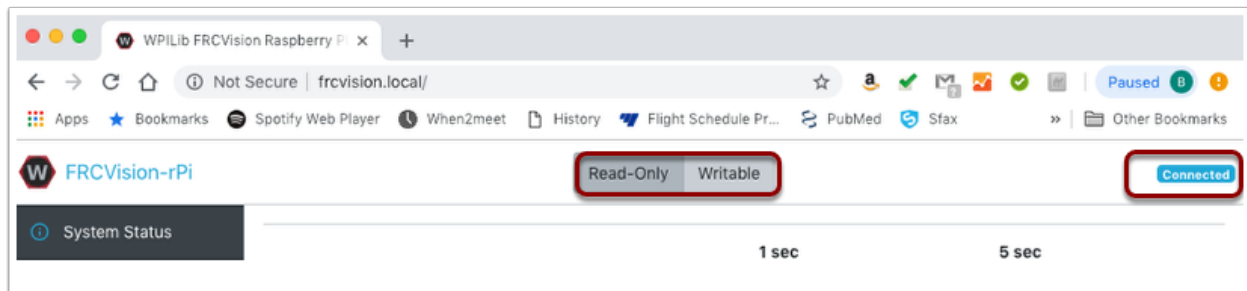
- Look at the Raspberry PI status
- View the status of the background process running the camera
- View or change network settings
- Look at each camera plugged into the rPi and add additional cameras
- Load a new vision program onto the rPi

Setting the rPI to be Read-Only vs. Writable

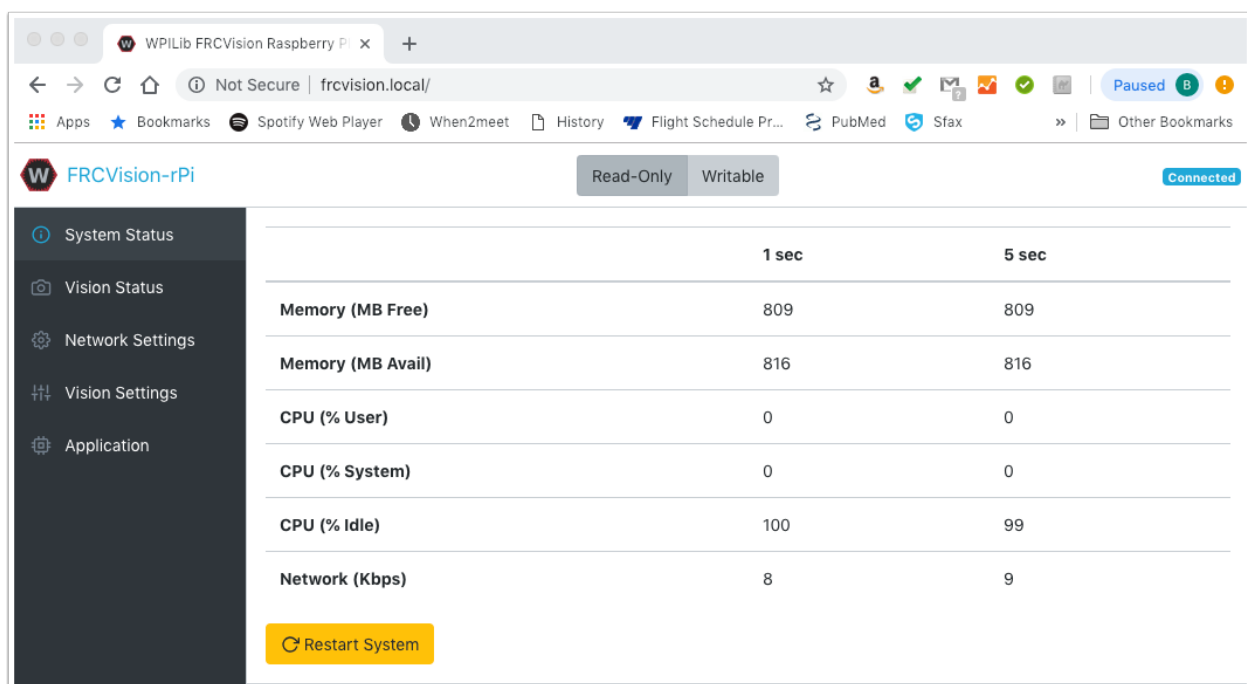
The rPi is normally set to Read-Only which means that the file system cannot be changed. This ensures that if power is removed without first shutting down the rPi the file system isn't corrupted. When settings are changed (following sections), the new settings cannot be saved while the rPi file system is set as Read-Only. Buttons are provided that allow the file system to be changed from Read-Only to Writable and back whenever changes are made. If the other buttons that change information stored on the rPi cannot be press, check the Read-Only status of the system.

Status of the network connection to the rPi

There is a label in the top right corner of the console that indicates if the rPi is currently connected. It will change from Connected to Disconnected if there is no longer a network connection to the rPi.



System status



The system status shows what the CPU on the rPI is doing at any time. There are two columns of status values, one being a 1 second average and the other a 5 second average. Shown is:

- free and available RAM on the PI
- CPU usage for user processes and system processes as well as idle time.
- And network bandwidth - which allows one to determine if the used camera bandwidth is exceeding the maximum bandwidth allowed in the robot rules for any year.

Vision Status

The screenshot shows the FRCVision-rPi web interface. At the top, there are tabs for 'Read-Only' and 'Writable', and a 'Disconnected' status indicator. The left sidebar contains a menu with 'System Status', 'Vision Status' (selected), 'Network Settings', 'Vision Settings', and 'Application'. The main content area has a 'Background Service (Automatic Restart)' section with a status of 'Unknown Status' and buttons for 'Up', 'Down', 'Terminate', and 'Kill'. Below this is a 'Console Output' section with an 'Enable' toggle switch. The console output displays several error messages from the NetworkTables (NT) library, indicating connection timeouts and errors due to no route to host or unresolved addresses.

```

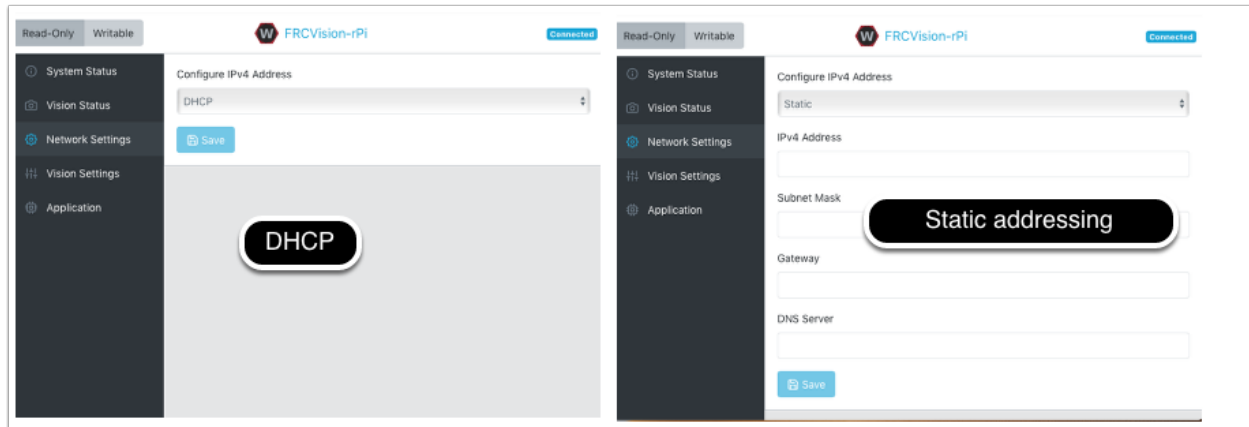
NT: connect() to 10.2.94.2 port 1735 timed out
NT: connect() to 172.22.11.2 port 1735 timed out
NT: ERROR: select() to 10.2.94.2 port 1735 error 113 - No route to host
(TCPConnector.cpp:173)
NT: ERROR: select() to 172.22.11.2 port 1735 error 113 -
No route to host (TCPConnector.cpp:173)
NT: ERROR: could not resolve roboRIO-294-FRC.local address
(TCPConnector.cpp:99)
NT: connect() to 10.2.94.2 port 1735 timed out
NT: connect() to 172.22.11.2 port 1735 timed out
NT: ERROR: could not resolve roboRIO-294-FRC.frc-field.local address
(TCPConnector.cpp:99)
NT: connect() to 10.2.94.2 port 1735 timed out
NT: connect() to 172.22.11.2 port 1735 timed out
NT: ERROR: select() to 10.2.94.2 port 1735 error 113 - No route to host
  
```

Allows monitoring of the task which is running the camera code in the rPI, either one of the default programs or your own program in Java, C++, or Python. You can also enable and view the console output to see messages coming from the background camera service. In this case there are number of messages about being unable to connect to [NetworkTables](#) (NT: connect()) because in this example the rPI is simply connected to a laptop with no NetworkTables server running (usually the roboRIO.)

Network Settings

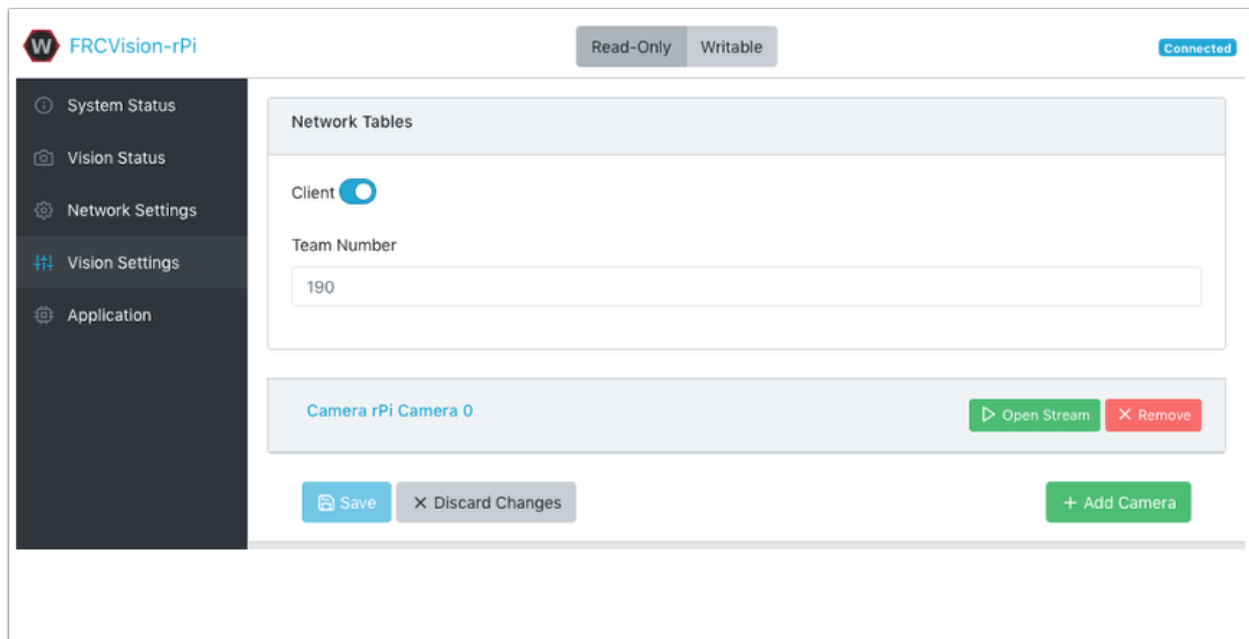
The rPi network settings have options to connect to the PI:

- DHCP - the default name resolution usually used by the roboRIO. The default name is wpilibpi.local.
- Static - where a fixed IP address, network mask, and router settings are filled in explicitly
- DHCP with Static Fallback - DHCP with Static Fallback - the PI will try to get an IP address via DHCP, but if it can't find a DHCP server, it will use the provided static IP address and parameters



The picture above is showing the settings for both DHCP and Static IP Addressing. The mDNS name for the rPi should always work regardless of the options selected above.

Vision Settings



The Vision Settings are to set the parameters for each camera and whether the rPi should be a NetworkTables client or server. There can only be one server on the network and the roboRIO is always a server. Therefore when connected to a roboRIO, the rPi should always be in client mode with the team number filled in. If testing on a desktop setup with no roboRIO or anything acting as a server then it should be set to Server (Client switch is off).

To view and manipulate all the camera settings click on the camera in question. In this case the camera is called “Camera rPi Camera 0” and clicking on the name reveals the current camera view and the associated settings.

Manipulating the camera settings is reflected in the current camera view. The bottom of the page shows all the possible camera modes (combinations of Width, Height, and frame rates) that are supported by this camera.

Note: If the camera image is not visible on the *Open Stream* screen then check the supported video modes at the bottom of the page. Then go back to ‘Vision Settings’ and click on the camera in question and verify that the pixel format, width, height, and FPS are listed in the supported video modes.

Getting the current settings to persist over reboots

The rPi will load all the camera settings on startup. Editing the camera configuration in the above screen is temporary. To make the values persist click on the “Load Source Config From Camera” button and the current settings will be filled in on the camera settings fields. Then click “Save” at the bottom of the page. Note: you must set the file system Writeable in order to save the settings. *The Writeable button is at the top of the page.*

There are some commonly used camera settings values shown in the camera settings (above). These values Brightness, White Balance, and Exposure are loaded into the camera before the user JSON file is applied. So if a user JSON file contains those settings they will overwrite the ones from the text field.

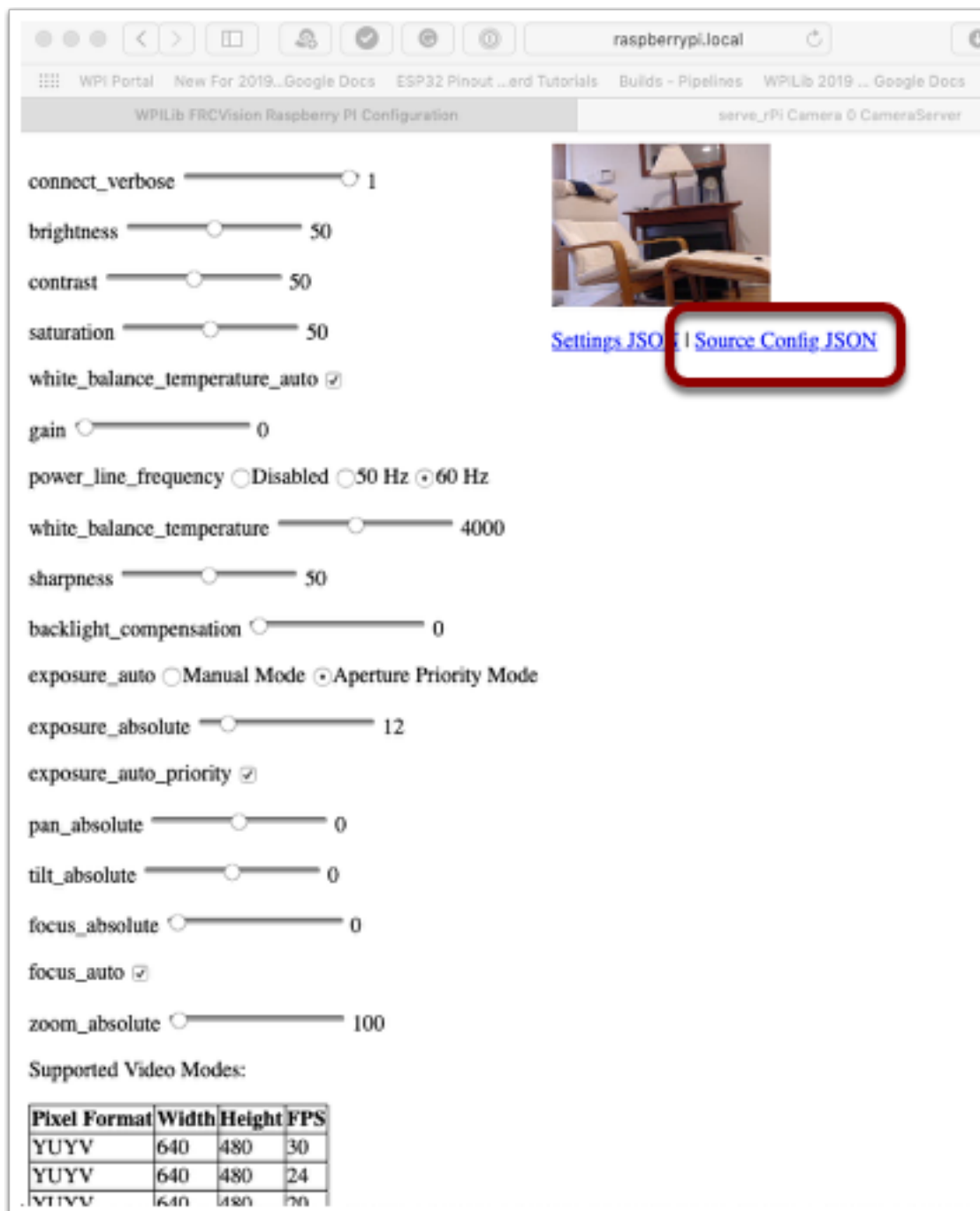
Application

The Application tab shows the application that is currently running on the rPi.

Vision workflows

There is a sample vision program using OpenCV in each of the supported languages, C++, Java, or Python. Each sample program can capture and stream video from the rPi. In addition, the samples have some minimal OpenCV. They are all set up to be extended to replace the provided OpenCV sample code with the code needed for the robot application. The rPi Application tab supports a number of programming workflows:

- Stream one or more cameras from the rPi for consumption on the driver station computer and displayed using ShuffleBoard
- Edit and build one of the sample programs (one for each language: Java, C++ or Python) on the rPi using the included toolchains
- Download a sample program for the chosen language and edit and build it on your development computer. Then upload that built program back to the rPi



connect_verbose ☐ 1

brightness 50

contrast 50

saturation 50

white_balance_temperature_auto ☒

gain 0

power_line_frequency ☐ Disabled ☐ 50 Hz ☒ 60 Hz

white_balance_temperature 4000

sharpness 50

backlight_compensation 0

exposure_auto ☐ Manual Mode ☒ Aperture Priority Mode

exposure_absolute 12

exposure_auto_priority ☒

pan_absolute 0

tilt_absolute 0

focus_absolute 0

focus_auto ☒

zoom_absolute 100

Supported Video Modes:

Pixel Format	Width	Height	FPS
YUYV	640	480	30
YUYV	640	480	24
YUYV	640	480	15

Network Tables

Client ☒

Team Number
190

Camera rPi Camera 0

[▶ Open Stream](#) [✕ Remove](#)

Name


Path

rPi Camera 0

/dev/video0

Load Source Config From JSON File

Browse

 Copy Source Config From Camera

Pixel Format

Width

Height

FPS

MJPEG

160

120

30

Brightness

White Balance

Exposure

15

auto

auto

Custom Properties JSON

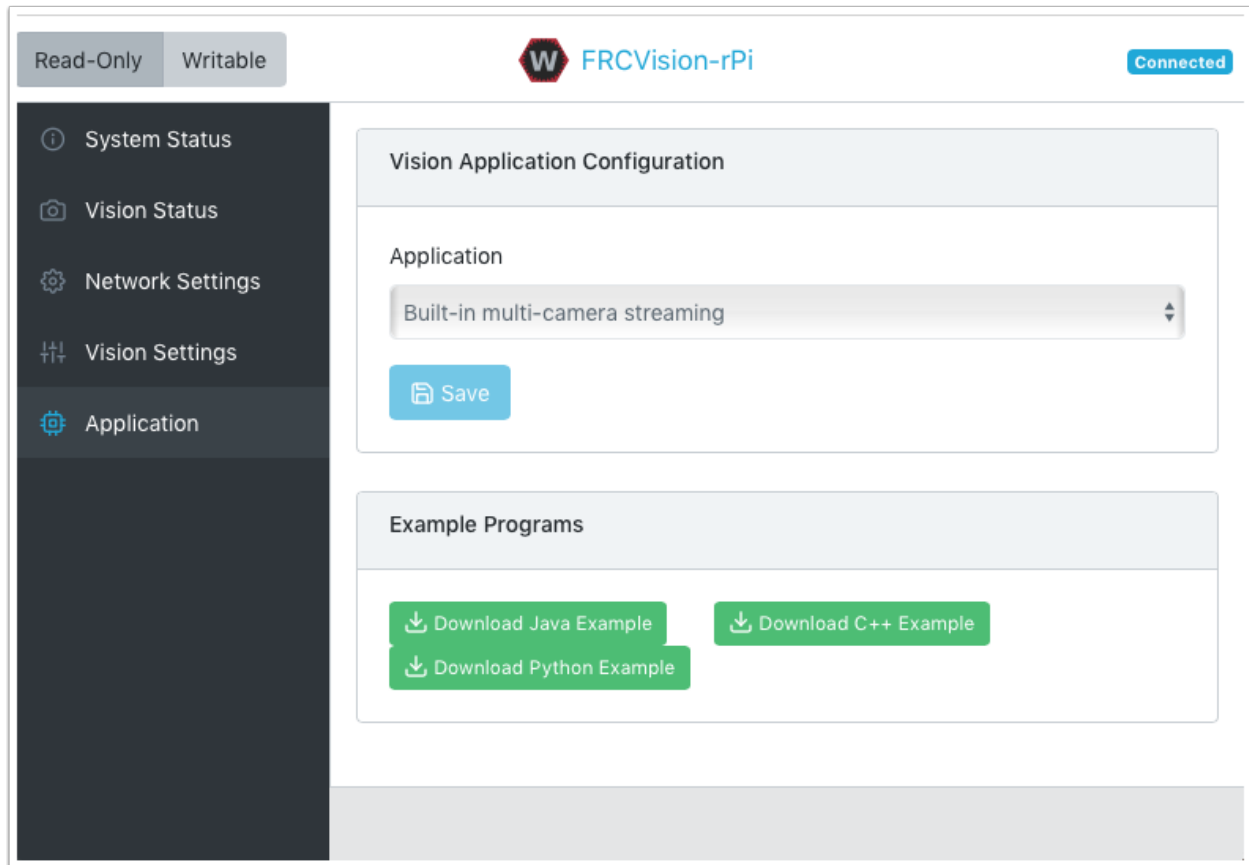
```
[{"name":"connect_verbose","value":1},{"name":"contrast","value":50}, {"name":"saturation","value":50}, {"name":"gain","value":0}, {"name":"power_line_frequency","value":2}, {"name":"sharpness","value":50}, {"name":"backlight_compensation","value":0}, {"name":"exposure_auto_priority","value":true},
```

These settings are filled in when copying the Source Config from the camera

[Save](#) [✕ Discard Changes](#)

[+ Add Camera](#)

- Do everything yourself using completely custom applications and scripts (probably based on one of the samples)



The running application can be changed by selecting one of the choices in the drop-down menu. The choices are:

- Built-in multi camera streaming which streams whatever cameras are plugged into the rPi. The camera configuration including number of cameras can be set on the “Vision Settings” tab.
- Custom application which doesn’t upload anything to the rPi and assumes that the developer wants to have a custom program and script.
- Java, C++ or Python pre-installed sample programs that can be edited into your own application.
- Java, C++, or Python uploaded program. Java programs require a .jar file with the compiled program and C++ programs require an rPi executable to be uploaded to the rPi.

When selecting one of the Upload options, a file chooser is presented where the jar, executable or Python program can be selected and uploaded to the rPi. In the following picture an Uploaded Java jar is chosen and the “Choose File” button will select a file and clicking on the “Save” button will upload the selected file.

Note: in order to Save a new file onto the rPi, the file system has to be set writeable using the “Writable” button at the top left of the web page. After saving the new file, set the file system back to “Read-Only” so that it is protected against accidental changes.

Vision Application Configuration

Application

✓ Built-in multi-camera streaming

Custom

Java example (must be built to work)

C++ example (must be built to work)

Python example

Uploaded Java jar

Uploaded C++ executable

Uploaded Python file

Download Java Example

Download C++ Example

Download Python Example

Application

Uploaded Java jar

Upload executable file

Choose File

no file selected

Save

Example Programs

Download Java Example

Download C++ Example

Download Python Example

28.2.7 Using CameraServer

Grabbing Frames from CameraServer

The WPILibPi image comes with all the necessary libraries to make your own vision processing system. In order to get the current frame from the camera, you can use the CameraServer library. For information about CameraServer, the *Read and Process Video: CameraServer Class*.

Python

```
from cscore import CameraServer
import cv2
import numpy as np

cs = CameraServer.getInstance()
cs.enableLogging()

camera = cs.startAutomaticCapture()
camera.setResolution(width, height)

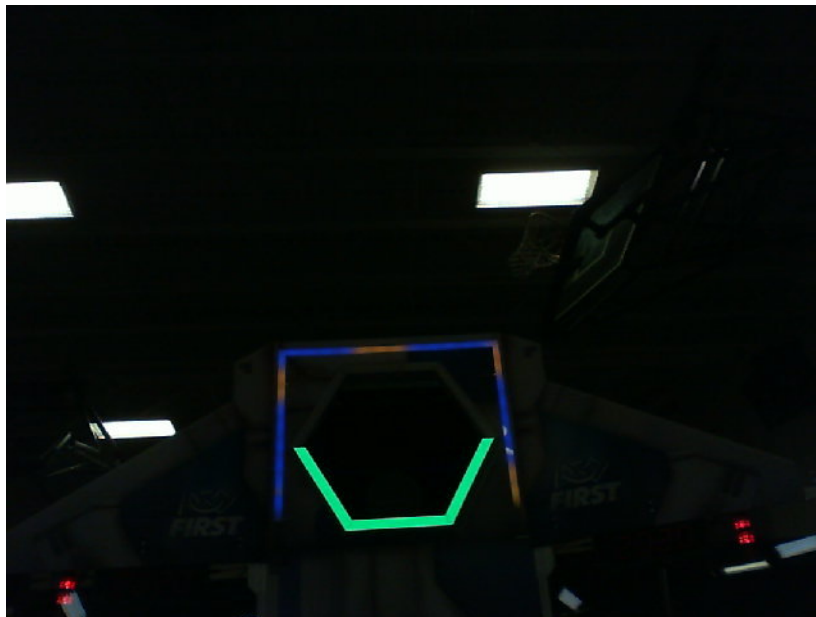
sink = cs.getVideo()

while True:
    time, input_img = cvSink.grabFrame(input_img)

    if time == 0: # There is an error
        continue
```

Note: OpenCV reads in the image as **BGR**, not **RGB** for historical reasons. Use cv2.cvtColor if you want to change it to RGB.

Below is an example of an image that might be grabbed from CameraServer.



Sending frames to CameraServer

Sometimes, you may want to send processed video frames back to the CameraServer instance for debugging purposes, or viewing in a dashboard application like Shuffleboard.

Python

```
#
# CameraServer initialization code here
#

output = cs.putVideo("Name", width, height)

while True:
    time, input_img = cvSink.grabFrame(input_img)

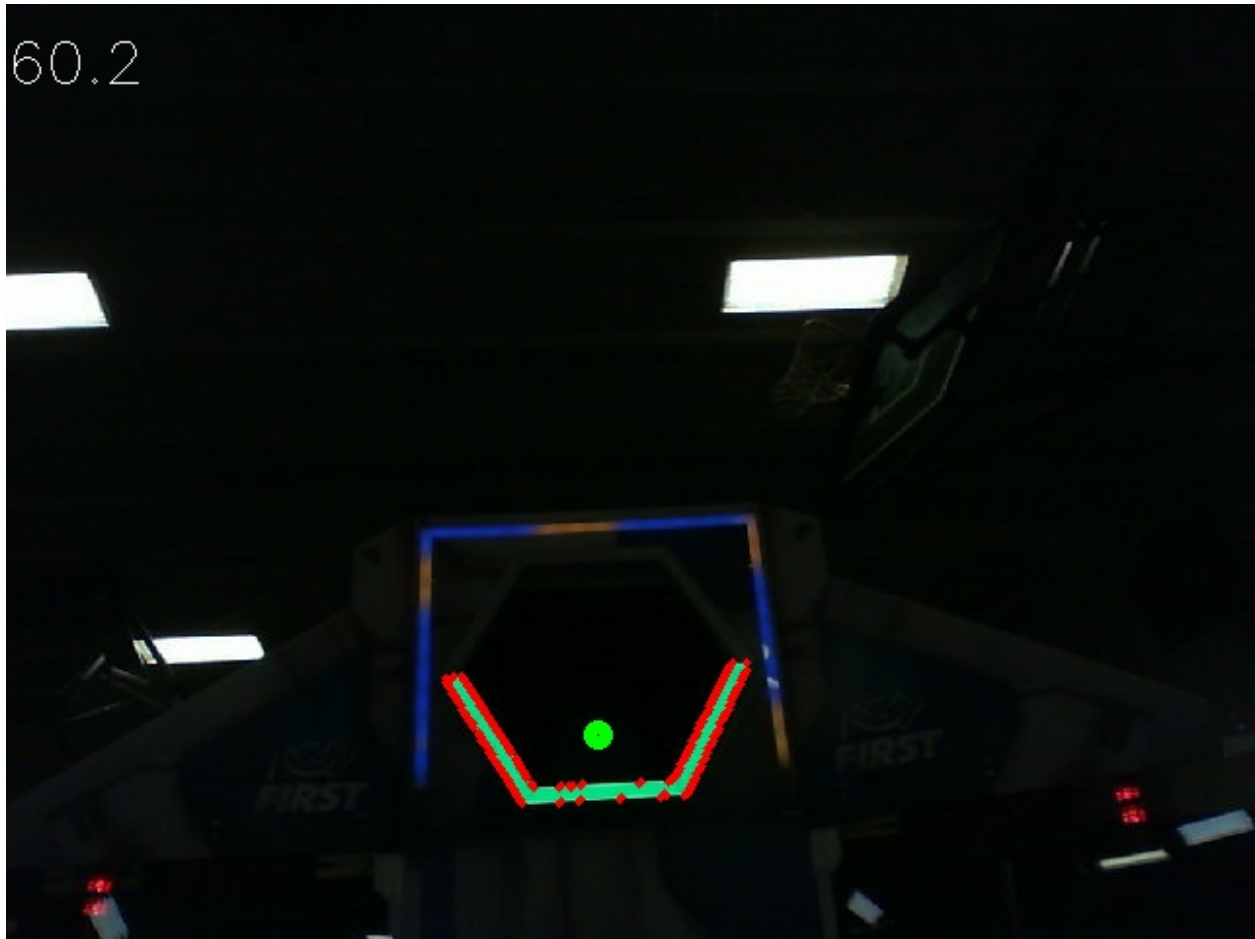
    if time == 0: # There is an error
        output.notifyError(sink.getError())
        continue

    #
    # Insert processing code here
    #

    output.putFrame(processed_img)
```

As an example, the processing code could outline the target in red, and show the corners in yellow for debugging purposes.

Below is an example of a fully processed image that would be sent back to CameraServer and displayed on the Driver Station computer.



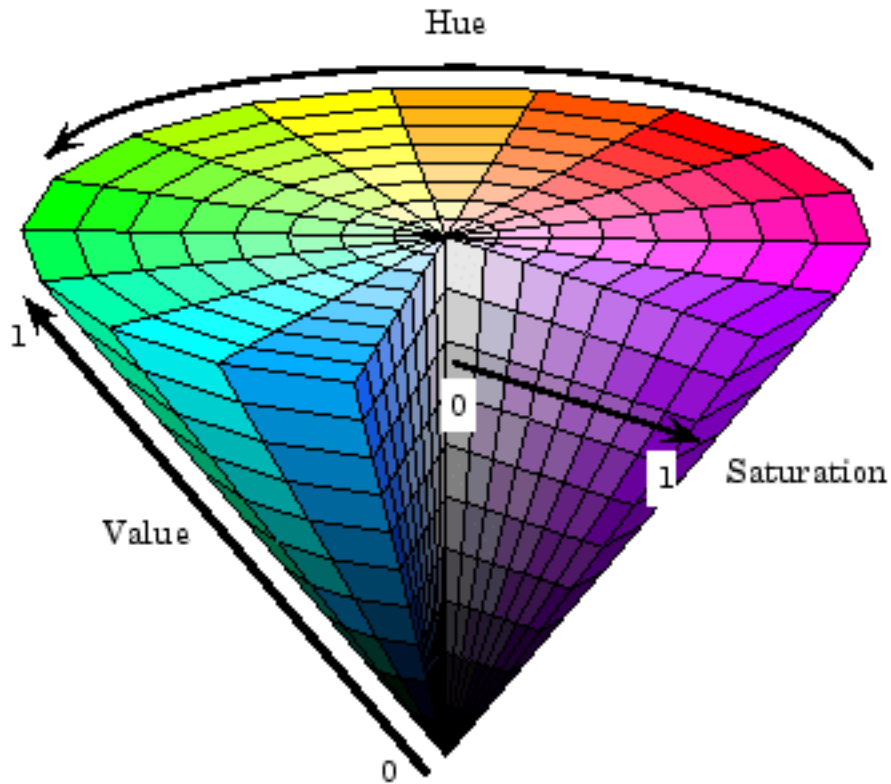
28.2.8 Thresholding an Image

In order to turn a colored image, such as the one captured by your camera, into a binary image, with the target as the “foreground”, we need to threshold the image using the hue, saturation, and value of each pixel.

The HSV Model

Unlike RGB, HSV allows you to not only filter based on the colors of the pixels, but also by the intensity of color and the brightness.

- Hue: Measures the color of the pixel.
- Saturation: Measures the intensity of color of the pixel.
- Value: Measures the brightness of the pixel.



You can use OpenCV to convert a BGR image matrix to HSV.

Python

```
hsv_img = cv2.cvtColor(input_img, cv2.COLOR_BGR2HSV)
```

Note: OpenCV's hue range is from 1° to 180° instead of the common 1° to 360° . In order to convert a common hue value to OpenCV, divide by 2.

Thresholding

We will use this field image as an example for the whole process of image processing.



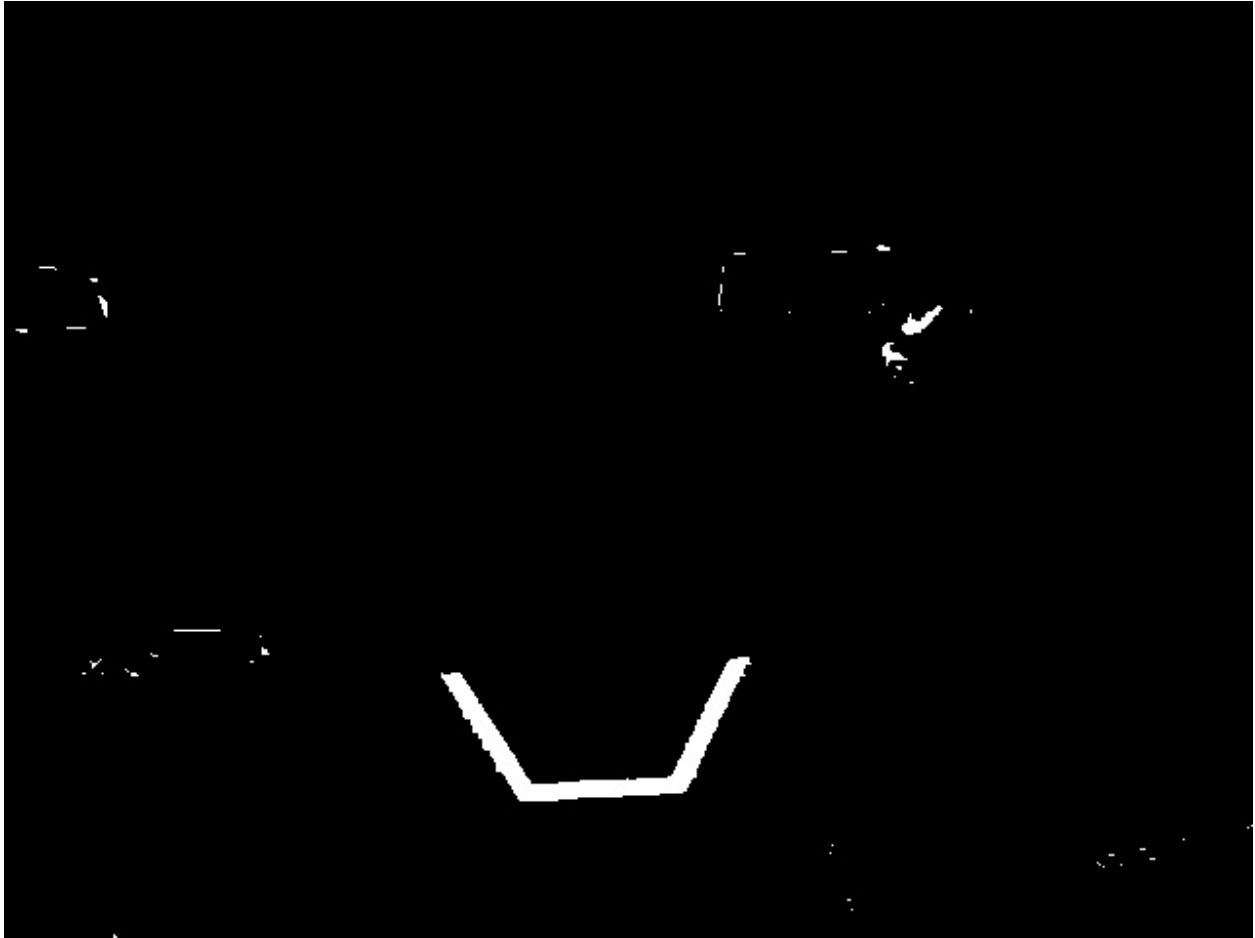
By thresholding the image using HSV, you can separate the image into the vision target (foreground), and the other things that the camera sees (background). The following code example converts a HSV image into a binary image by thresholding with HSV values.

Python

```
binary_img = cv2.inRange(hsv_img, (min_hue, min_sat, min_val), (max_hue, max_sat, max_val))
```

Note: These values may have to be tuned on an per-venue basis, as ambient lighting may differ across venues. It is recommended to allow editing of these values through NetworkTables in order to facilitate on-the-fly editing.

After thresholding, your image should look like this.



As you can see, the thresholding process may not be 100% clean. You can use morphological operations to deal with the noise.

28.2.9 Morphological Operations

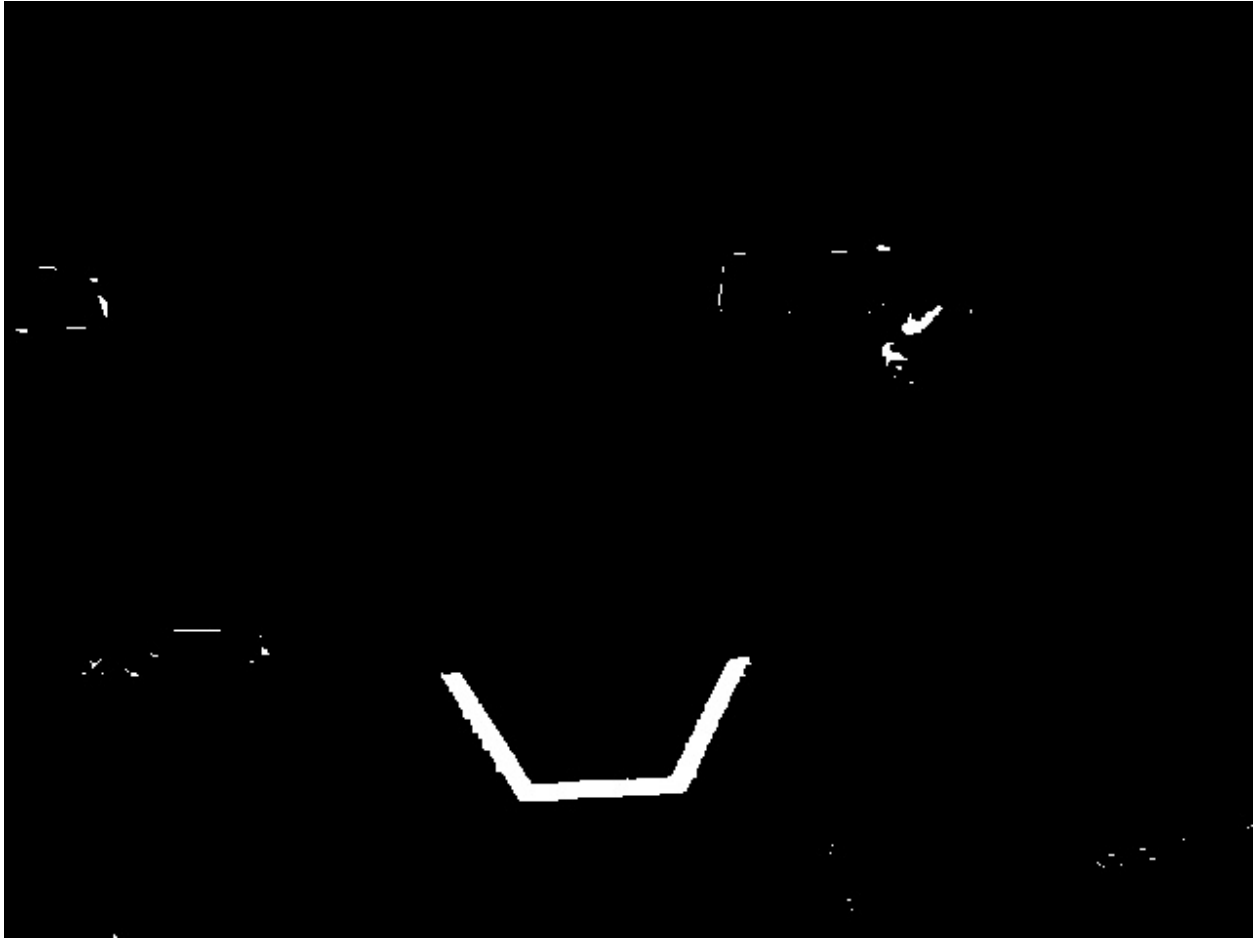
Sometimes, after thresholding your image, you have unwanted noise in your binary image. Morphological operations can help remove that noise from the image.

Kernel

The kernel is a simple shape where the origin is superimposed on each pixel of value 1 of the binary image. OpenCV limits the kernel to a $N \times N$ matrix where N is an odd number. The origin of the kernel is the center. A common kernel is

$$kernel = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Different kernels can affect the image differently, such as only eroding or dilating vertically. For reference, this is our binary image we created:

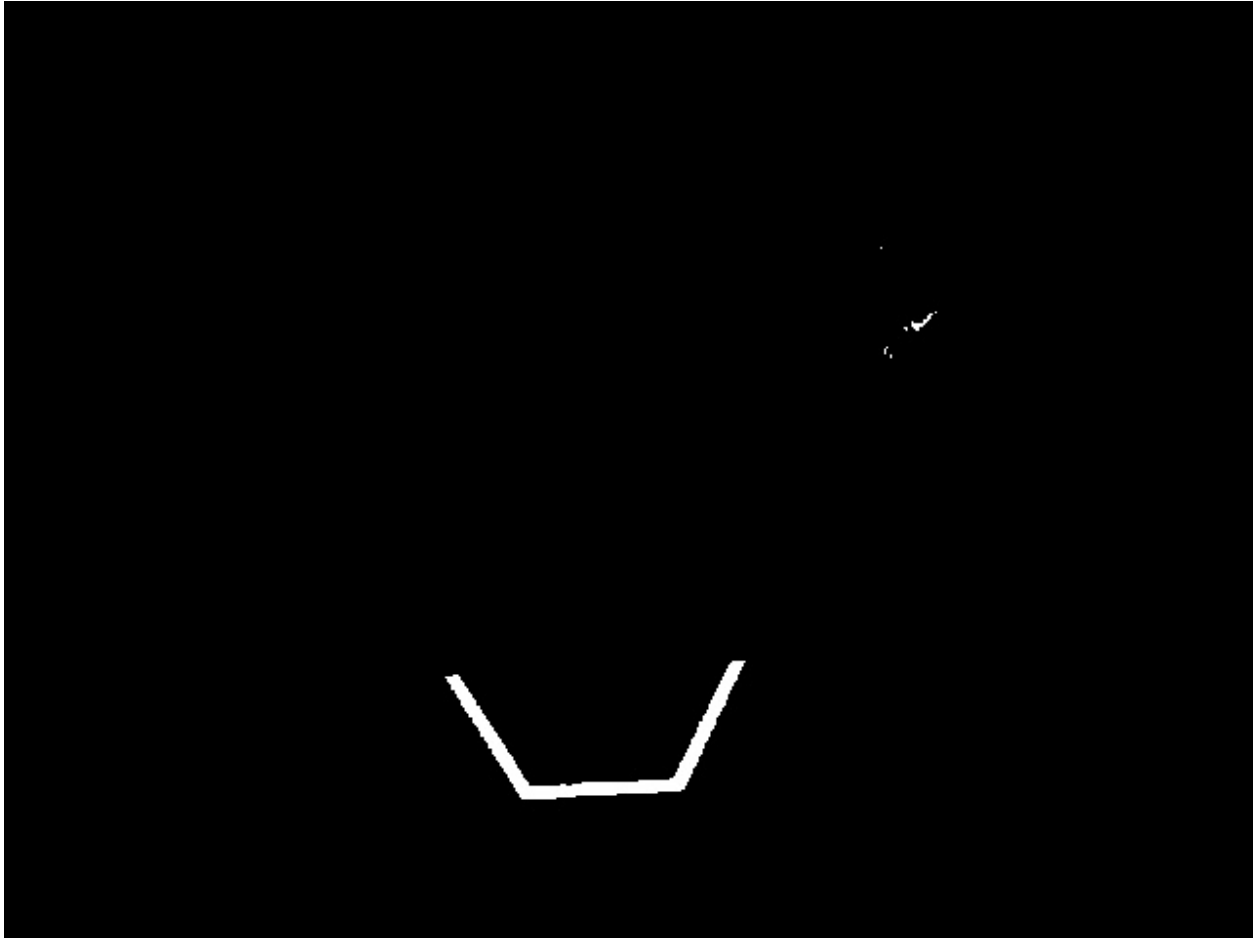


Erosion

Erosion in computer vision is similar to erosion on soil. It takes away from the borders of foreground objects. This process can remove noise from the background.

Python

```
kernel = np.ones((3, 3), np.uint8)
binary_img = cv2.erode(binary_img, kernel, iterations = 1)
```



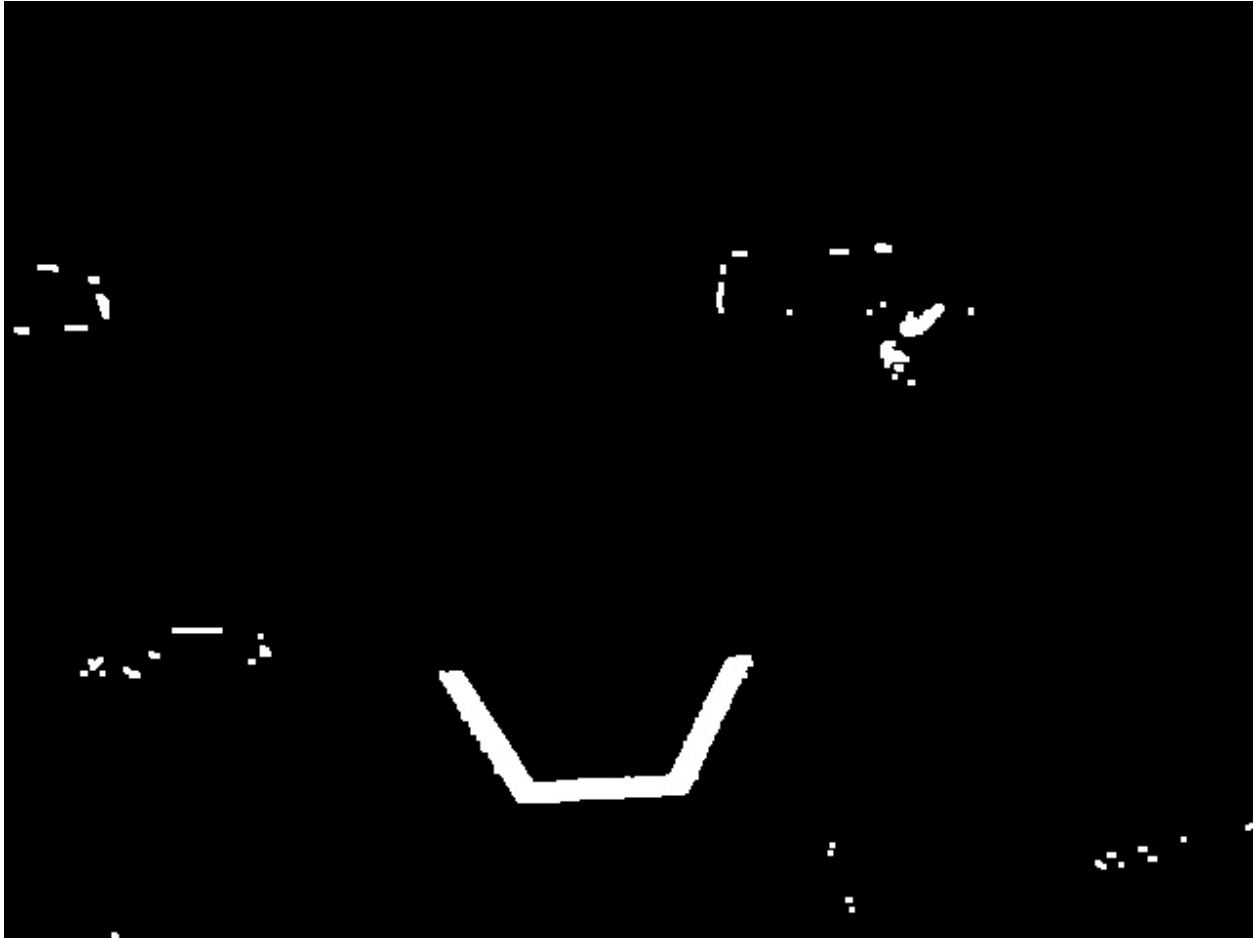
During erosion, if the superimposed kernel's pixels are not contained completely by the binary image's pixels, the pixel that it was superimposed on is deleted.

Dilation

Dilation is opposite of erosion. Instead of taking away from the borders, it adds to them. This process can remove small holes inside a larger region.

Python

```
kernel = np.ones((3, 3), np.uint8)
binary_img = cv2.dilate(binary_img, kernel, iterations = 1)
```



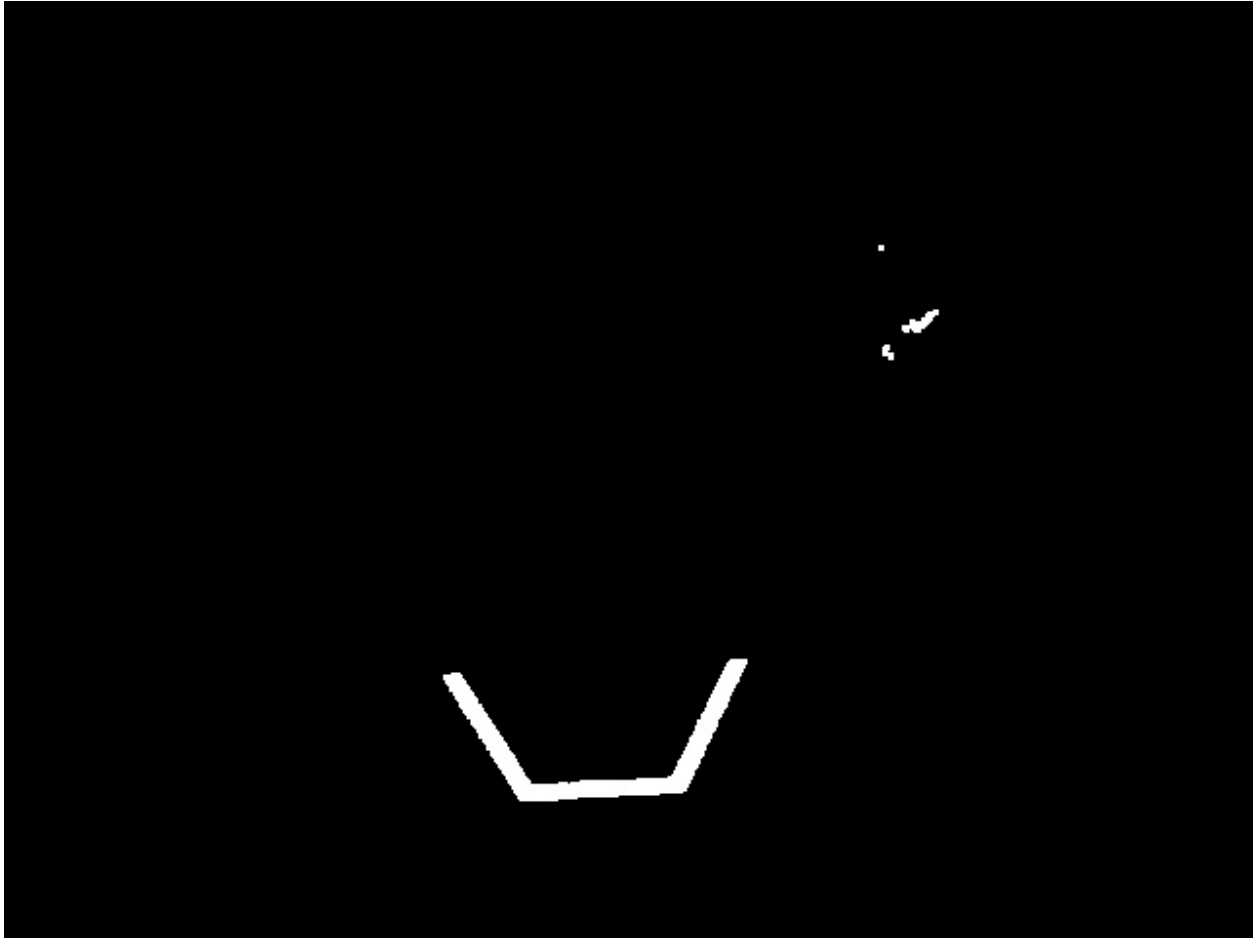
During dilation, every pixel of every superimposed kernel is included in the dilation.

Opening

Opening is erosion followed by dilation. This process removes noise without affecting the shape of larger features.

Python

```
kernel = np.ones((3, 3), np.uint8)
binary_img = cv2.morphologyEx(binary_img, cv2.MORPH_OPEN, kernel)
```

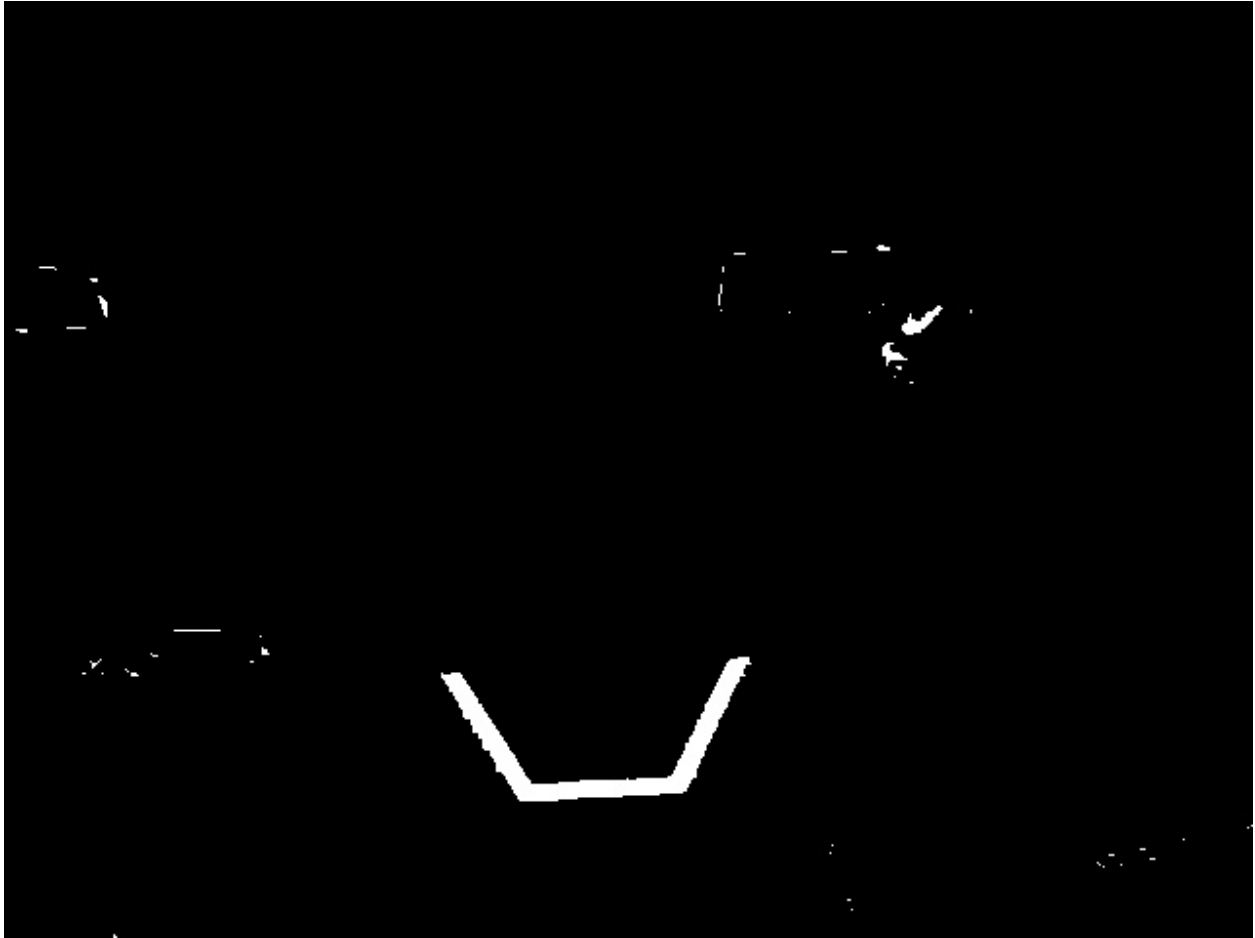
Note: In this specific case, it is appropriate to do more iterations of opening in order to get rid of the pixels in the top right.

Closing

Closing is dilation followed by erosion. This process removes small holes or breaks without affecting the shape of larger features.

Python

```
kernel = np.ones((3, 3), np.uint8)
binary_img = cv2.morphologyEx(binary_img, cv2.MORPH_CLOSE, kernel)
```



28.2.10 Working with Contours

After thresholding and removing noise with morphological operations, you are now ready to use OpenCV's `findContours` method. This method allows you to generate contours based on your binary image.

Finding and Filtering Contours

Python

```
_, contours, _ = cv2.findContours(binary_img, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_  
    ↪SIMPLE)
```

In cases where there is only one vision target, you can just take the largest contour and assume that that is the target you are looking for. When there is more than one vision target, you can use size, shape, fullness, and other properties to filter unwanted contours out.

Python

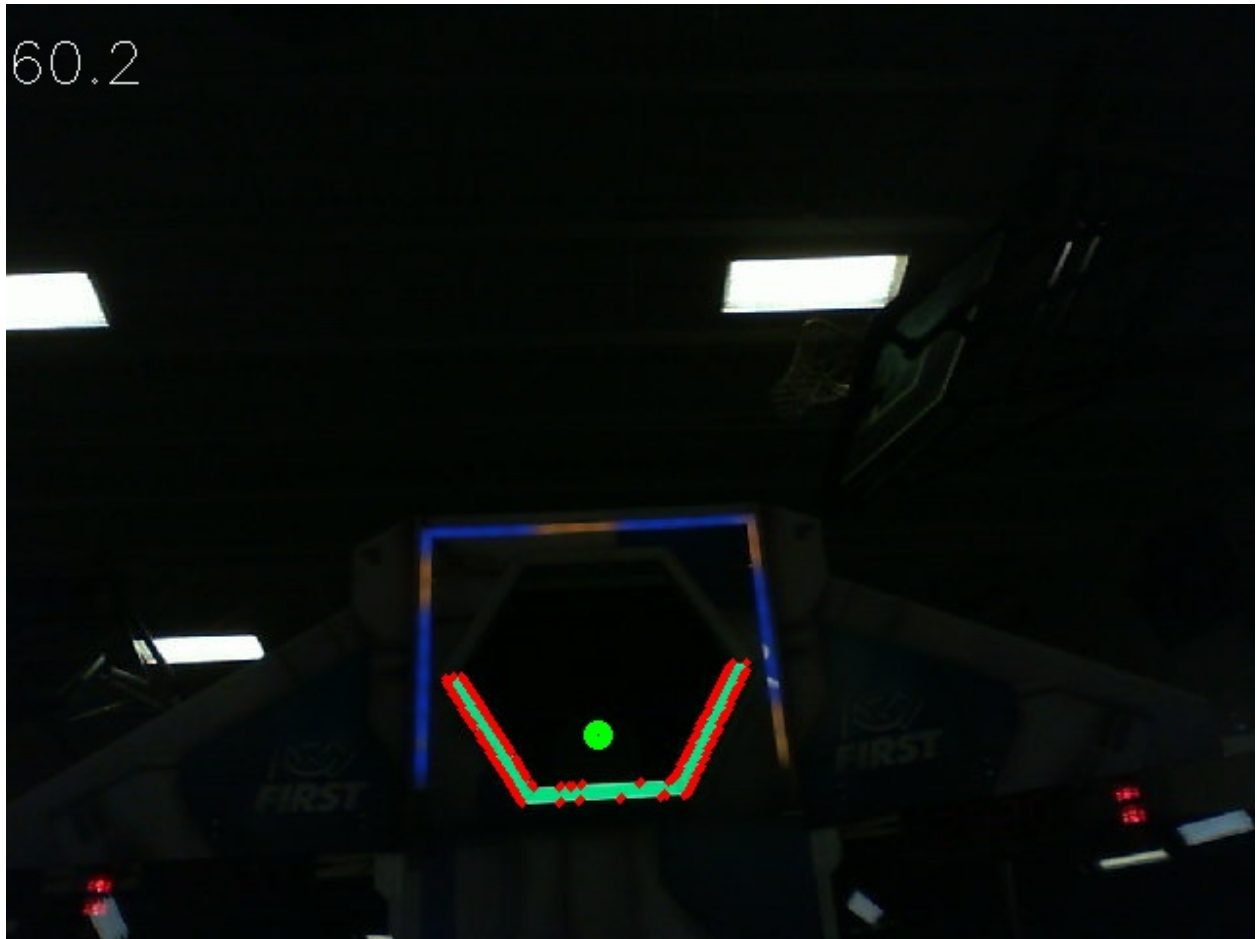
```
if len(contours) > 0:  
    largest = contours[0]  
    for contour in contours:
```

(continues on next page)

(continued from previous page)

```
if cv2.contourArea(contour) > cv2.contourArea(largest):  
    largest = contour  
  
#  
# Contour processing code  
#
```

If you draw the contour you just found, it should look something like this:



Extracting Information from Contours

Now that you've found the contour(s) that you want, you now want to get information about it, such as the center, corners, and rotation.

Center

Python

```
rect = cv2.minAreaRect(contour)
center, _, _ = rect
center_x, center_y = center
```

Corners

Python

```
corners = cv2.convexHull(contour)
corners = cv2.approxPolyDP(corners, 0.1 * cv2.arcLength(contour), True)
```

Rotation

Python

```
_, _, rotation = cv2.fitEllipse(contour)
```

For more information on how you can use these values, see [Measurements](#)

Publishing to NetworkTables

You can use NetworkTables to send these properties to the Driver Station and the RoboRIO. Additional processing could be done on the Raspberry Pi, or the RoboRIO itself.

Python

```
from networktables import NetworkTables

nt = NetworkTables.getTable('vision')

#
# Initialization code here
#

while True:

    #
    # Image processing code here
    #

    nt.putNumber('center_x', center_x)
    nt.putNumber('center_y', center_y)
```

28.2.11 Basic Vision Example

This is an example of a basic vision setup that posts the target's location in the aiming coordinate system described [here](#) to NetworkTables, and uses CameraServer to display a bounding rectangle of the contour detected. This example will display the framerate of the processing code on the images sent to CameraServer.

Python

```
from cscore import CameraServer
from networktables import NetworkTables

import cv2
import json
import numpy as np
import time

def main():
    with open('/boot/frc.json') as f:
        config = json.load(f)
        camera = config['cameras'][0]

    width = camera['width']
    height = camera['height']

    CameraServer.getInstance().startAutomaticCapture()

    input_stream = CameraServer.getInstance().getVideo()
    output_stream = CameraServer.getInstance().putVideo('Processed', width, height)

    # Table for vision output information
    vision_nt = NetworkTables.getTable('Vision')

    # Allocating new images is very expensive, always try to preallocate
    img = np.zeros(shape=(240, 320, 3), dtype=np.uint8)

    # Wait for NetworkTables to start
    time.sleep(0.5)

    while True:
        start_time = time.time()

        frame_time, input_img = input_stream.grabFrame(img)
        output_img = np.copy(input_img)

        # Notify output of error and skip iteration
        if frame_time == 0:
            output_stream.notifyError(input_stream.getError())
            continue

        # Convert to HSV and threshold image
        hsv_img = cv2.cvtColor(input_img, cv2.COLOR_BGR2HSV)
        binary_img = cv2.inRange(hsv_img, (65, 65, 200), (85, 255, 255))

        _, contour_list, _ = cv2.findContours(binary_img, mode=cv2.RETR_EXTERNAL,
        ↪method=cv2.CHAIN_APPROX_SIMPLE)

        x_list = []
```

(continues on next page)

(continued from previous page)

```

y_list = []

for contour in contour_list:

    # Ignore small contours that could be because of noise/bad thresholding
    if cv2.contourArea(contour) < 15:
        continue

    cv2.drawContours(output_img, contour, -1, color = (255, 255, 255), thickness_
    ↪ = -1)

    rect = cv2.minAreaRect(contour)
    center, size, angle = rect
    center = [int(dim) for dim in center] # Convert to int so we can draw

    # Draw rectangle and circle
    cv2.drawContours(output_img, np.int0(cv2.boxPoints(rect)), -1, color = (0, 0,
    ↪ 255), thickness = 2)
    cv2.circle(output_img, center = center, radius = 3, color = (0, 0, 255),
    ↪ thickness = -1)

    x_list.append((center[0] - width / 2) / (width / 2))
    x_list.append((center[1] - width / 2) / (width / 2))

    vision_nt.putNumberArray('target_x', x_list)
    vision_nt.putNumberArray('target_y', y_list)

    processing_time = time.time() - start_time
    fps = 1 / processing_time
    cv2.putText(output_img, str(round(fps, 1)), (0, 40), cv2.FONT_HERSHEY_SIMPLEX,
    ↪ 1, (255, 255, 255))
    output_stream.putFrame(output_img)

main()

```

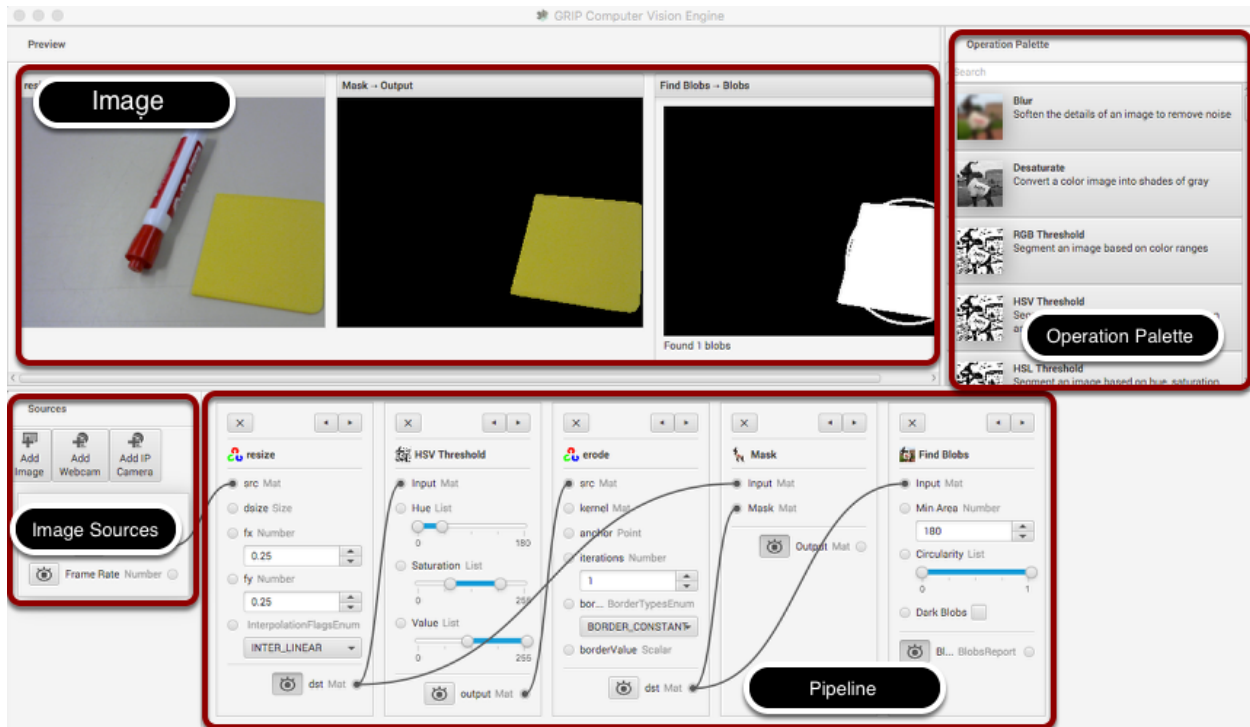
28.3 Vision with GRIP

28.3.1 Introduction to GRIP

GRIP is a tool for developing computer vision algorithms interactively rather than through trial and error coding. After developing your algorithm you may run GRIP in headless mode on your roboRIO, on a Driver Station Laptop, or on a coprocessor connected to your robot network. With Grip you choose vision operations to create a graphical pipeline that represents the sequence of operations that are performed to complete the vision algorithm.

GRIP is based on OpenCV, one of the most popular computer vision software libraries used for research, robotics, and vision algorithm implementations. The operations that are available in GRIP are almost a 1 to 1 match with the operations available if you were hand coding the same algorithm with some text-based programming language.

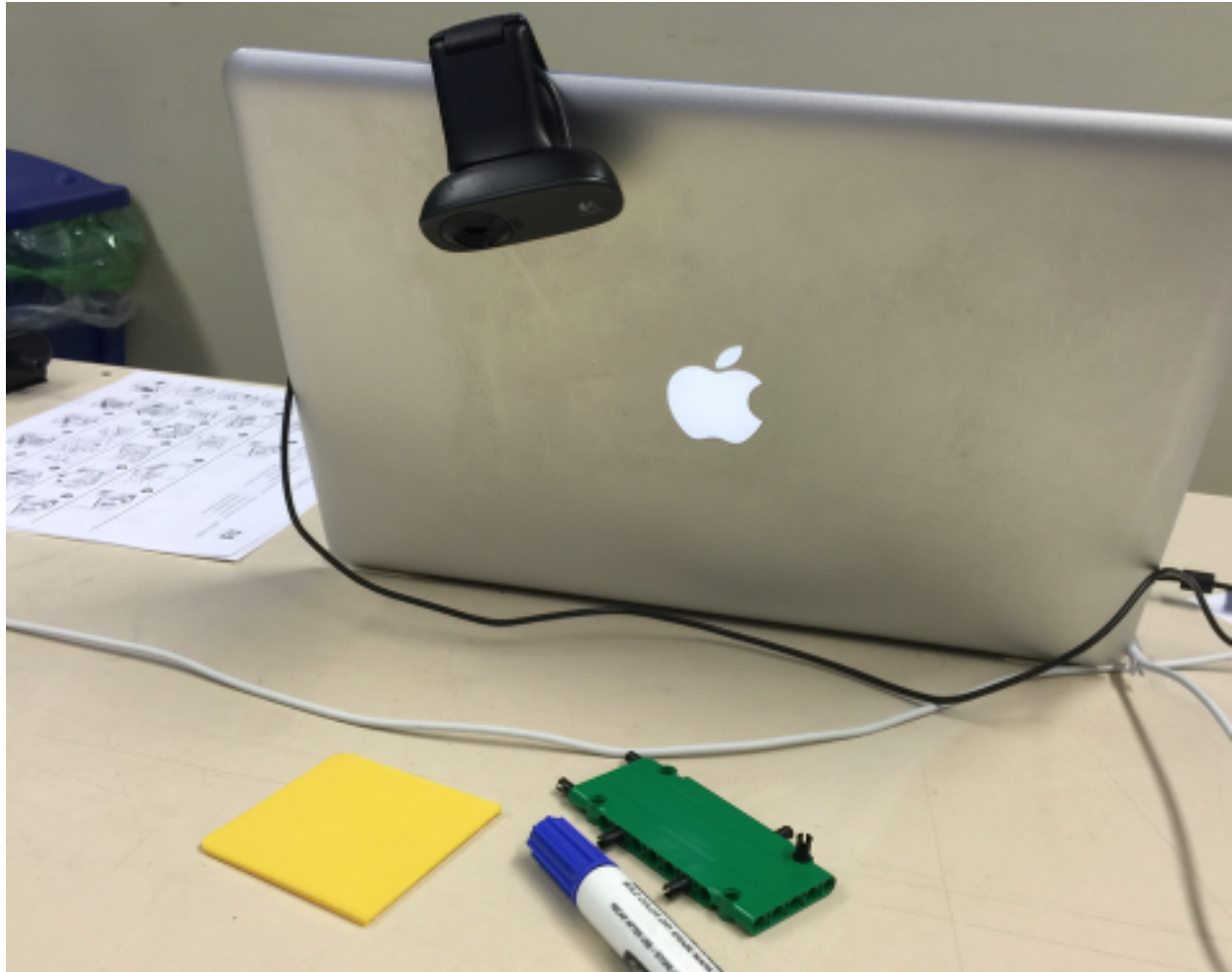
The GRIP user interface



The GRIP user interface consists of 4 parts:

- **Image Sources** are the ways of getting images into the GRIP pipeline. You can provide images through attached cameras or files. Sources are almost always the beginning of the image processing algorithm.
- **Operation Palette** contains the image processing steps from the OpenCV library that you can chain together in the pipeline to form your algorithm. Clicking on an operation in the palette adds it to the end of the pipeline. You can then use the left and right arrows to move the operation within the pipeline.
- **Pipeline** is the sequence of steps that make up the algorithm. Each step (operation) in the pipeline is connected to a previous step from the output of one step to an input to the next step. The data flows from generally from left to right through the connections that you create.
- **Image Preview** are shows previews of the result of each step that has it's preview button pressed. This makes it easy to debug algorithms by being able to preview the outputs of each intermediate step.

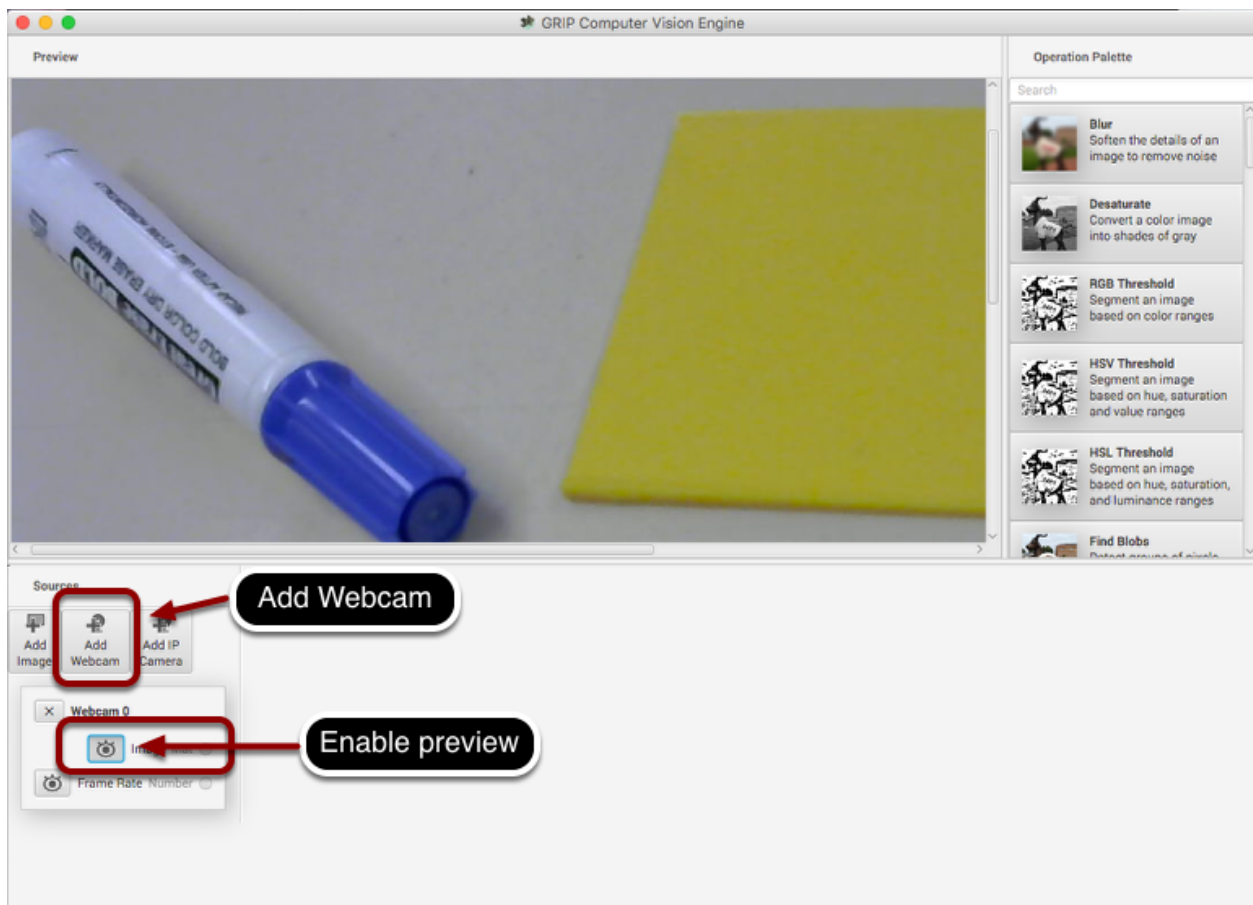
Finding the yellow square



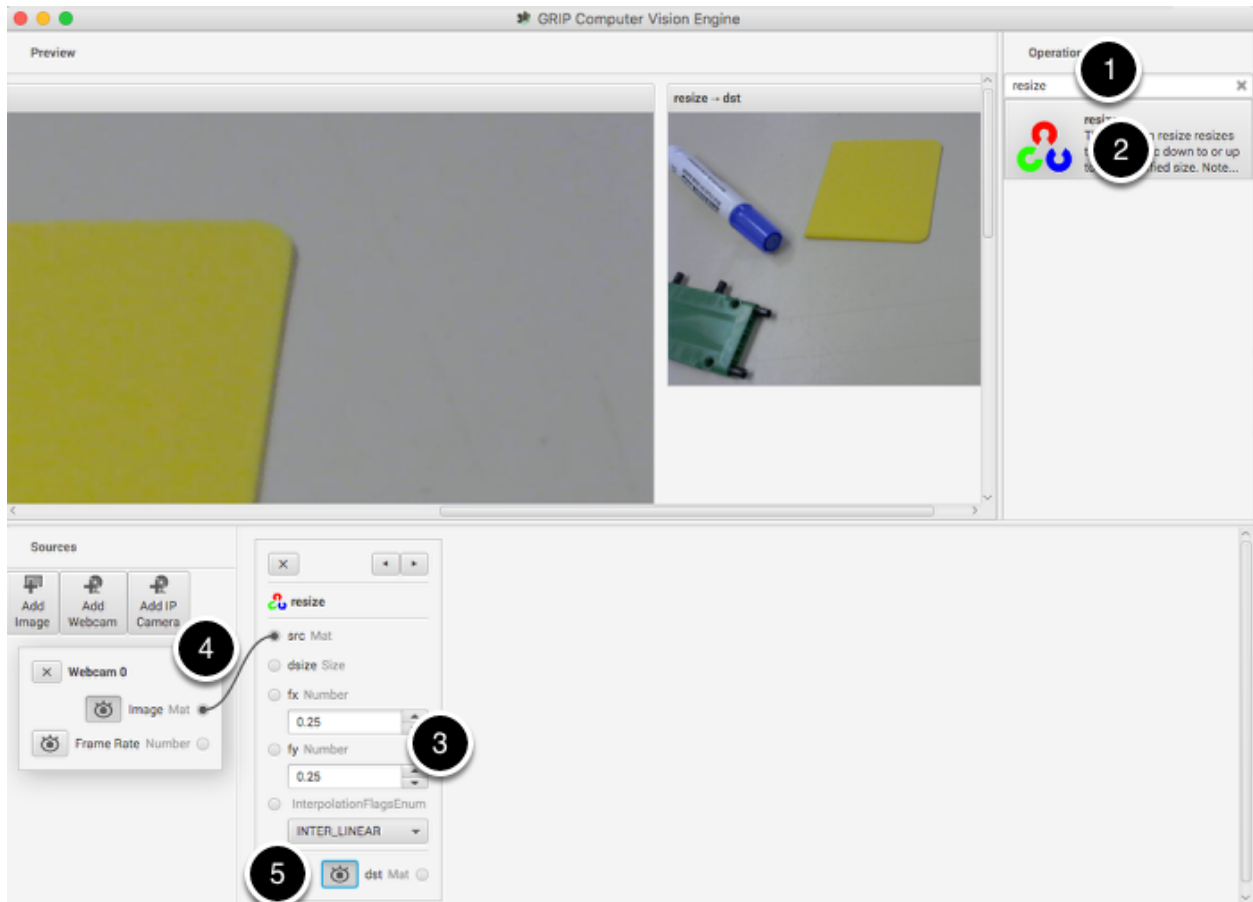
In this application we will try to find the yellow square in the image and display its position. The setup is pretty simple, just a USB web camera connected to the computer looking down at some colorful objects. The yellow plastic square is the thing that we're interested in locating in the image.

Enable the image source

The first step is to acquire an image. To use the source, click on the "Add Webcam" button and select the camera number. In this case the Logitech USB camera that appeared as Webcam 0 and the computer monitor camera was Webcam 1. The web camera is selected in this case to grab the image behind the computer as shown in the setup. Then select the image preview button and the real-time display of the camera stream will be shown in the preview area.



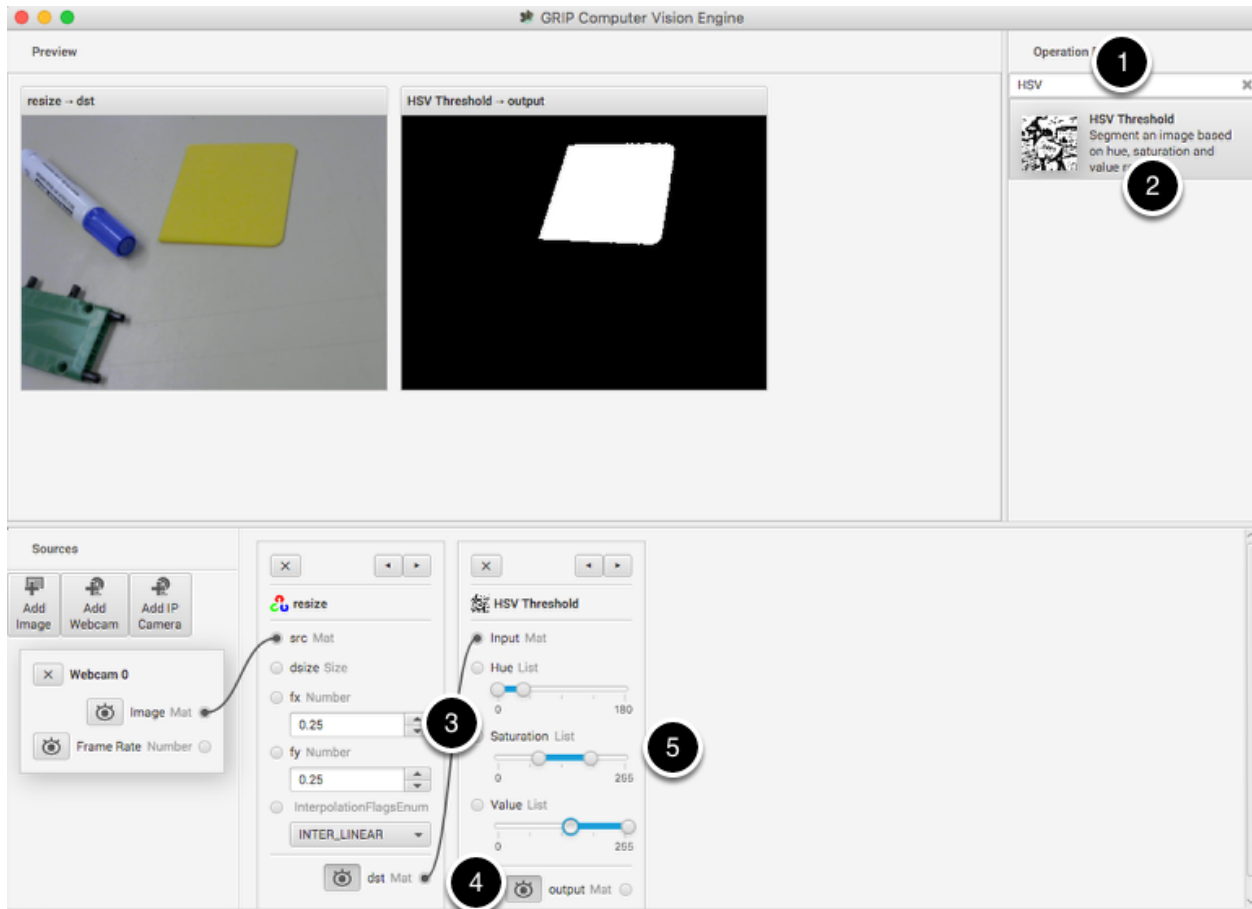
Resize the image



In this case the camera resolution is too high for our purposes, and in fact the entire image cannot even be viewed in the preview window. The “Resize” operation is clicked from the Operation Palette to add it to the end of the pipeline. To help locate the Resize operation, type “Resize” into the search box at the top of the palette. The steps are:

1. Type “Resize” into the search box on the palette
2. Click the Resize operation from the palette. It will appear in the pipeline.
3. Enter the x and y resize scale factor into the resize operation in the pipeline. In this case 0.25 was chosen for both.
4. Drag from the Webcam image output mat socket to the Resize image source mat socket. A connection will be shown indicating that the camera output is being sent to the resize input.
5. Click on the destination preview button on the “Resize” operation in the pipeline. The smaller image will be displayed alongside the larger original image. You might need to scroll horizontally to see both as shown.
6. Lastly, click the Webcam source preview button since there is no reason to look at both the large image and the smaller image at the same time.

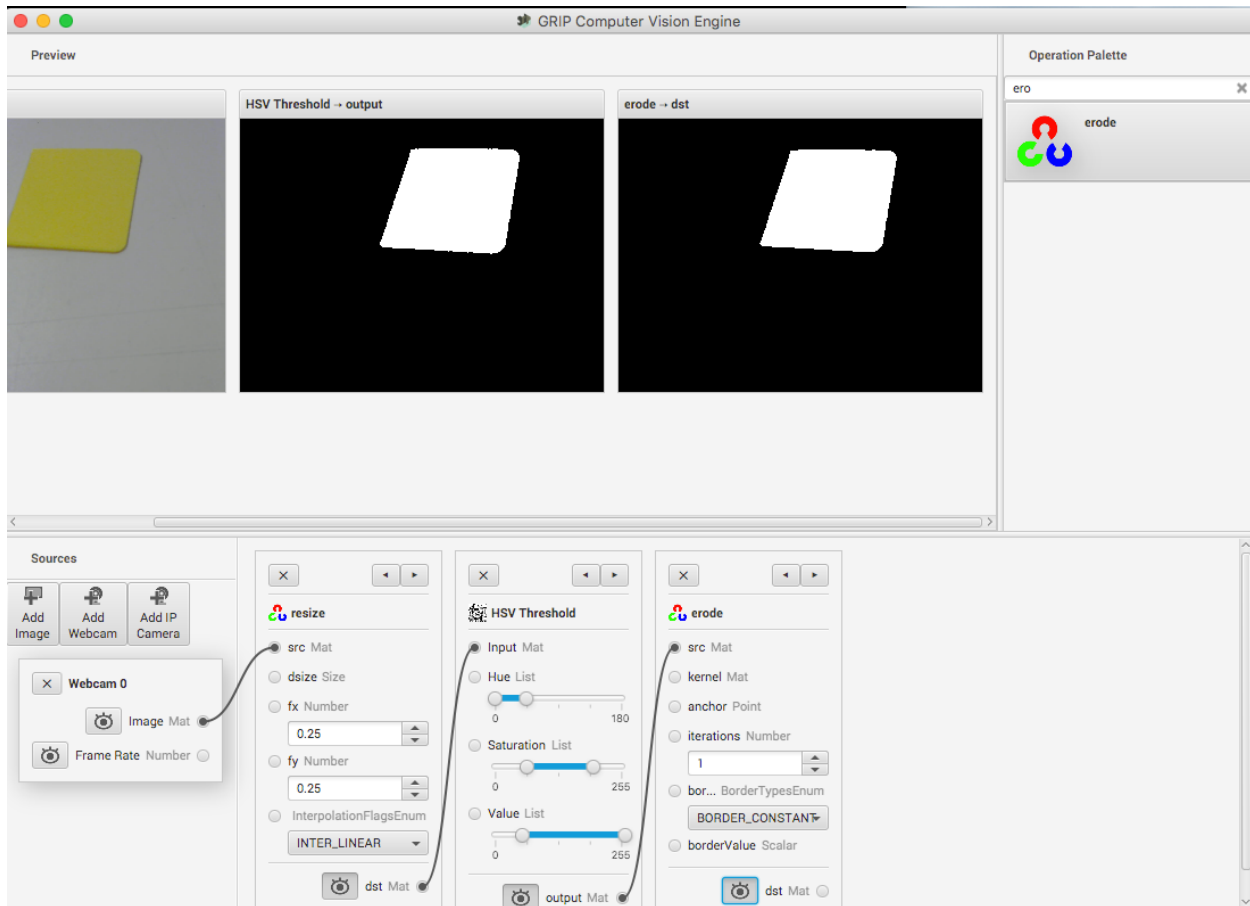
Find only the yellow parts of the image



The next step is to remove everything from the image that doesn't match the yellow color of the piece of plastic that is the object being detected. To do that a HSV Threshold operation is chosen to set upper and lower limits of HSV values to indicate which pixels should be included in the resultant binary image. Notice that the target area is white while everything that wasn't within the threshold values are shown in black. Again, as before:

1. Type HSV into the search box to find the HSV Threshold operation.
2. Click on the operation in the palette and it will appear at the end of the pipeline.
3. Connect the dst (output) socket on the resize operation to the input of the HSV Threshold.
4. Enable the preview of the HSV Threshold operation so the result of the operation is displayed in the preview window.
5. Adjust the Hue, Saturation, and Value parameters only the target object is shown in the preview window.

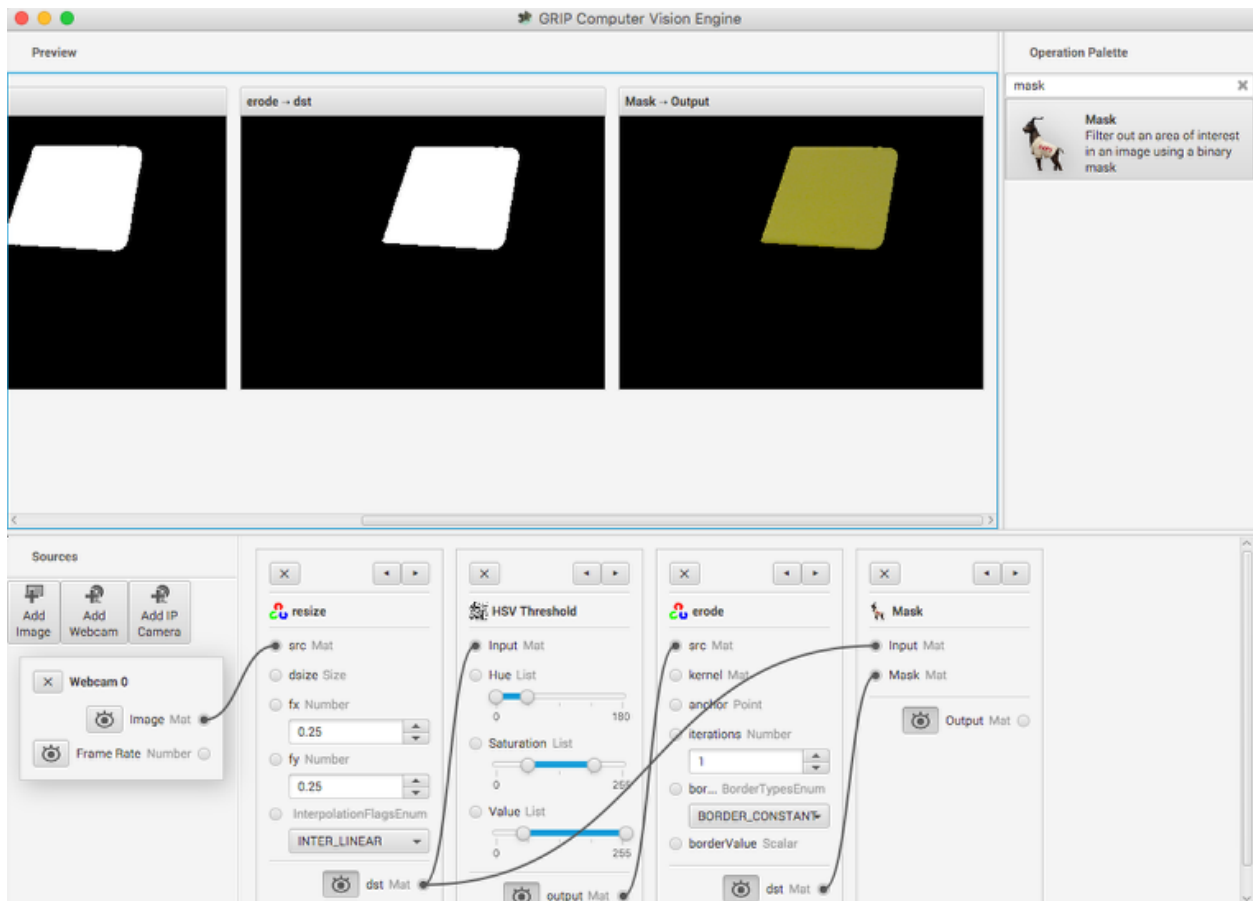
Get rid of the noise and extraneous hits



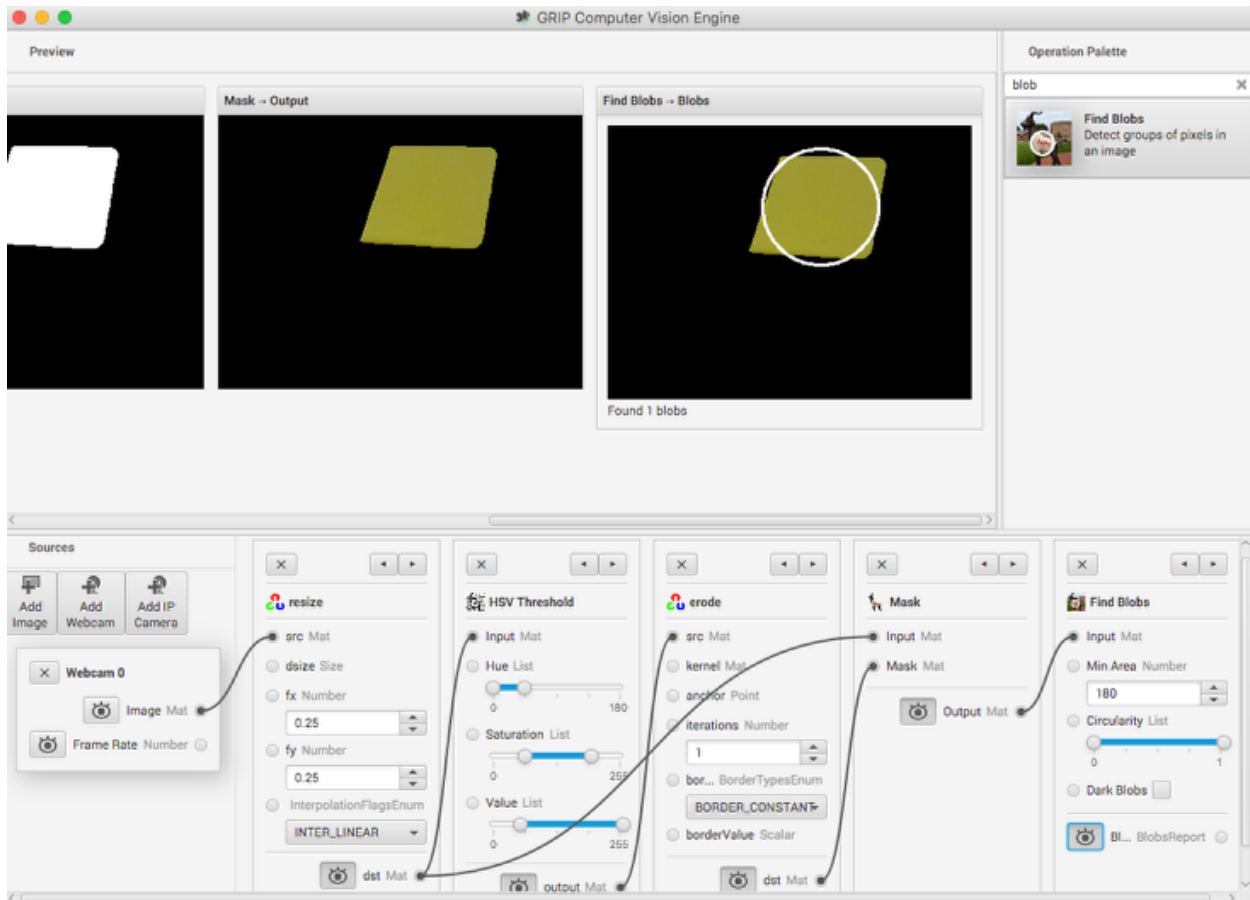
This looks pretty good so far, but sometimes there is noise from other things that couldn't quite be filtered out. To illustrate one possible technique to reduce those occasional pixels that were detected, an Erosion operation is chosen. Erosion will remove small groups of pixels that are not part of the area of interest.

Mask just the yellow area from the original image

Here a new image is generated by taking the original image and masking (and operation) it with the the results of the erosion. This leaves just the yellow card as seen in the original image with nothing else shown. And it makes it easy to visualize exactly what was being found through the series of filters.



Find the yellow area (blob)



The last step is actually detecting the yellow card using a Blob Detector. This operation looks for a grouping of pixels that have some minimum area. In this case, the only non-black pixels are from the yellow card after the filtering is done. You can see that a circle is drawn around the detected portion of the image. In the release version of GRIP (watch for more updates between now and kickoff) you will be able to send parameters about the detected blob to your robot program using *NetworkTables*.

Status of GRIP

As you can see from this example, it is very easy and fast to be able to do simple object recognition using GRIP. While this is a very simple example, it illustrates the basic principles of using GRIP and feature extraction in general. Over the coming weeks the project team will be posting updates to GRIP as more features are added. Currently it supports cameras (Axis ethernet camera and web cameras) and image inputs. There is no provision for output yet although *NetworkTables* and ROS (Robot Operating System) are planned.

You can either download a pre-built release of the code from the GitHub page “Releases” section (<https://github.com/WPIRoboticsProjects/GRIP>) or you can clone the source repository and build it yourself. Directions on building GRIP are on the project page. There is also additional documentation on the project wiki.

So, please play with GRiP and give us feedback here on the forum. If you find bugs, you can either post them here or as a GitHub project issue on the project page.

28.3.2 Generating Code from GRIP

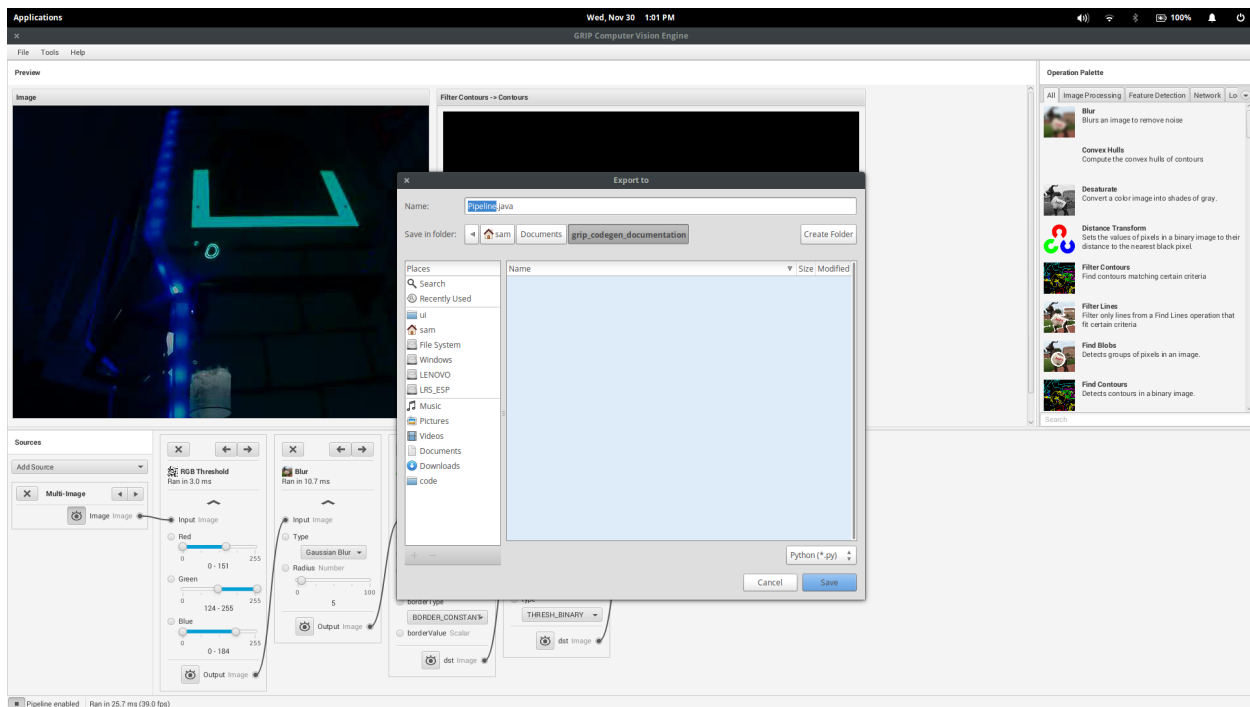
GRIP Code Generation

When running your vision algorithm on a small processor such as a roboRIO or Raspberry PI it is encouraged to run OpenCV directly on the processor without the overhead of GRIP. To facilitate this, GRIP can generate code in C++, Java, and Python for the pipeline that you have created. This generated code can be added to your robot project and called directly from your existing robot code.

Input sources such as cameras or image directories and output steps such as NetworkTables are not generated. Your code must supply images as OpenCV mats. On the roboRIO, the CameraServer class supplies images in that format. For getting results you can just use generated getter methods for retrieving the resultant values such as contour x and y values.

Generating Code

To generate code, go to **Tools > Generate Code**. This will bring up a save dialog that lets you create a C++, Java, or Python class that performs the steps in the GRIP pipeline.



If generating code to be used in a pre-existing project, choose a relevant directory to save the pipeline to.

- **C++ Users:** the pipeline class is split into a header and implementation file
- **Java Users:** the generated class lacks a package declaration, so a declaration should be added to match the directory where the file was saved.

- **Python Users:** the module name will be identical to the class, so the import statement will be something like `from Pipeline import Pipeline`

Structure of the Generated Code

```
Pipeline:
// Process -- this will run the pipeline
process(Mat source)

// Output accessors
getFooOutput()
getBar0Output()
getBar1Output()
...
```

Running the Pipeline

To run the Pipeline, call the process method with the sources (webcams, IP camera, image file, etc) as arguments. This will expose the outputs of every operation in the pipeline with the `getFooOutput` methods.

Getting the Results

Users are able to the outputs of every step in the pipeline. The outputs of these operations would be accessible through their respective accessors. For example:

Operation	Java/C++ getter	Python variable
RGB Threshold	<code>getRgbThresholdOutput</code>	<code>rgb_threshold_output</code>
Blur	<code>getBlurOutput</code>	<code>blur_output</code>
CV Erode	<code>getCvErodeOutput</code>	<code>mcv_erode_output</code>
Find Contours	<code>getFindContoursOutput</code>	<code>find_contours_output</code>
Filter Contours	<code>getFilterContoursOutput</code>	<code>filter_contours_output</code>

If an operation appears multiple times in the pipeline, the accessors for those operations have the number of that operation:

Operation	Which appearance	Accessor
Blur	First	<code>getBlur0Output</code>
Blur	Second	<code>getBlur1Output</code>
Blur	Third	<code>getBlur2Output</code>

28.3.3 Using Generated Code in a Robot Program

GRIP generates a class that can be added to an FRC® program that runs on a roboRIO and without a lot of additional code, drive the robot based on the output.

Included here is a complete sample program that uses a GRIP pipeline that drives a robot towards a piece of retroreflective material.

This program is designed to illustrate how the vision code works and does not necessarily represent the best technique for writing your robot program. When writing your own program be aware of the following considerations:

1. **Using the camera output for steering the robot could be problematic.** The camera code in this example that captures and processes images runs at a much slower rate that is desirable for a control loop for steering the robot. A better, and only slightly more complex solution, is to get headings from the camera and it's processing rate, then have a much faster control loop steering to those headings using a gyro sensor.
2. **Keep the vision code in the class that wraps the pipeline.** A better way of writing object oriented code is to subclass or instantiate the generated pipeline class and process the OpenCV results there rather than in the robot program. In this example, the robot code extracts the direction to drive by manipulating the resultant OpenCV contours. By having the OpenCV code exposed throughout the robot program it makes it difficult to change the vision algorithm should you have a better one.

Iterative program definitions

Java

```
package org.usfirst.frc.team190.robot;

import org.usfirst.frc.team190.grip.MyVisionPipeline;

import org.opencv.core.Rect;
import org.opencv.imgproc.Imgproc;

import edu.wpi.cscore.UsbCamera;
import edu.wpi.first.cameraserver.CameraServer;
import edu.wpi.first.wpilibj.RobotDrive;
import edu.wpi.first.wpilibj.TimedRobot;
import edu.wpi.first.wpilibj.vision.VisionRunner;
import edu.wpi.first.wpilibj.vision.VisionThread;

public class Robot extends TimedRobot {

    private static final int IMG_WIDTH = 320;
    private static final int IMG_HEIGHT = 240;

    private VisionThread visionThread;
    private double centerX = 0.0;
    private RobotDrive drive;

    private final Object imgLock = new Object();
```

In this first part of the program you can see all the import statements for the WPILib classes used for this program.

- The **image width and height** are defined as 320x240 pixels.
- The **VisionThread** is a WPILib class makes it easy to do your camera processing in a separate thread from the rest of the robot program.
- **centerX** value will be the computed center X value of the detected target.
- **RobotDrive** encapsulates the 4 motors on this robot and allows simplified driving.
- **imgLock** is a variable to synchronize access to the data being simultaneously updated with each image acquisition pass and the code that's processing the coordinates and steering the robot.

Java

```
@Override
public void robotInit() {
    UsbCamera camera = CameraServer.getInstance().startAutomaticCapture();
    camera.setResolution(IMG_WIDTH, IMG_HEIGHT);

    visionThread = new VisionThread(camera, new MyVisionPipeline(), pipeline -> {
        if (!pipeline.filterContoursOutput().isEmpty()) {
            Rect r = Imgproc.boundingRect(pipeline.filterContoursOutput().get(0));
            synchronized (imgLock) {
                centerX = r.x + (r.width / 2);
            }
        }
    });
    visionThread.start();

    drive = new RobotDrive(1, 2);
}
```

The **robotInit()** method is called once when the program starts up. It creates a **CameraServer** instance that begins capturing images at the requested resolution (IMG_WIDTH by IMG_HEIGHT).

Next an instance of the class **VisionThread** is created. VisionThread begins capturing images from the camera asynchronously in a separate thread. After processing each image, the pipeline computed **bounding box** around the target is retrieved and it's **center X** value is computed. This centerX value will be the x pixel value of the center of the rectangle in the image.

The VisionThread also takes a **VisionPipeline** instance (here, we have a subclass **MyVisionPipeline** generated by GRIP) as well as a callback that we use to handle the output of the pipeline. In this example, the pipeline outputs a list of contours (outlines of areas in an image) that mark goals or targets of some kind. The callback finds the bounding box of the first contour in order to find its center, then saves that value in the variable centerX. Note the synchronized block around the assignment: this makes sure the main robot thread will always have the most up-to-date value of the variable, as long as it also uses **synchronized** blocks to read the variable.

Java

```
@Override
public void autonomousPeriodic() {
    double centerX;
    synchronized (imgLock) {
        centerX = this.centerX;
    }
}
```

(continues on next page)

(continued from previous page)

```
}  
double turn = centerX - (IMG_WIDTH / 2);  
drive.arcadeDrive(-0.6, turn * 0.005);  
}
```

This, the final part of the program, is called repeatedly during the **autonomous period** of the match. It gets the **centerX** pixel value of the target and **subtracts half the image width** to change it to a value that is **zero when the rectangle is centered** in the image and **positive or negative when the target center is on the left or right side of the frame**. That value is used to steer the robot towards the target.

Note the **synchronized** block at the beginning. This takes a snapshot of the most recent centerX value found by the VisionThread.

28.3.4 Using GRIP with a Kangaroo Computer

A recently available computer called the Kangaroo looks like a great platform for running GRIP on FRC® robots. Some of the specs for this processor include:

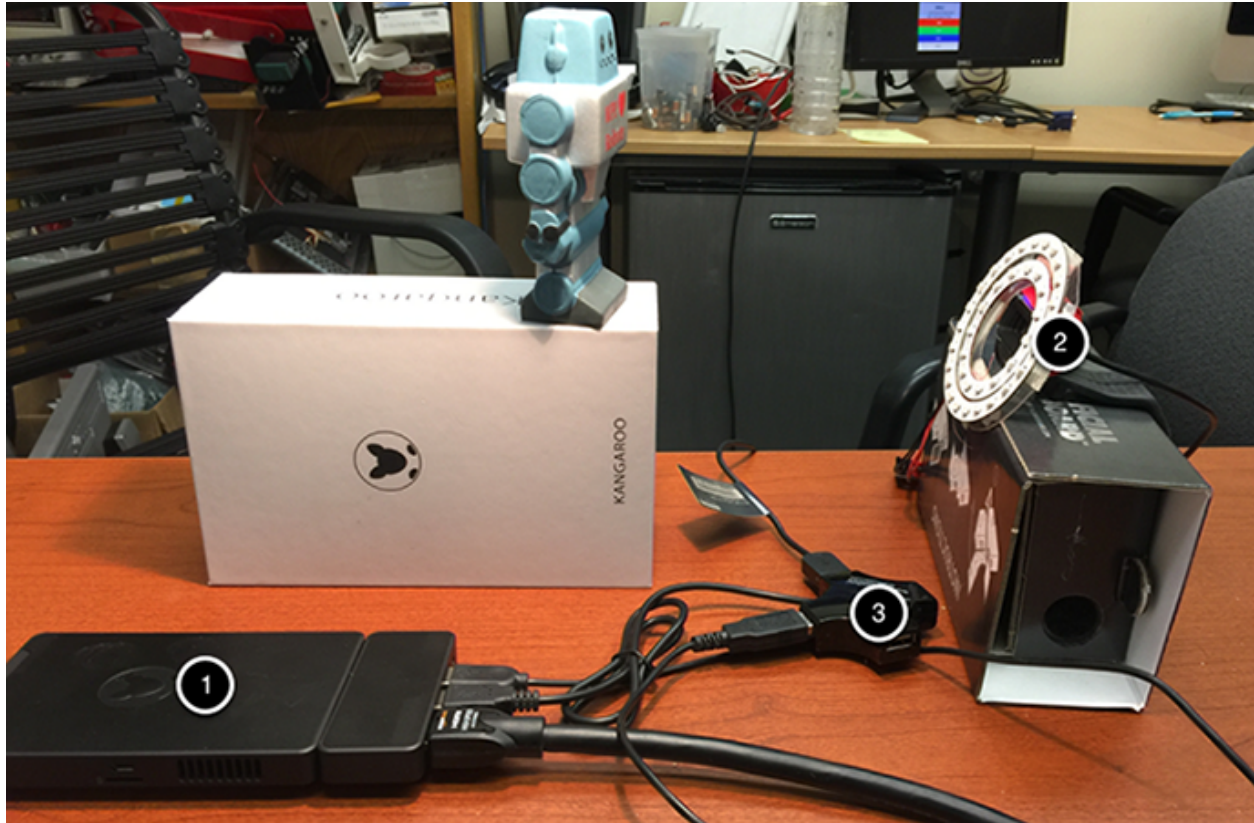
- Quad core 1.4Ghz Atom processor
- HDMI port
- 2 USB ports (1 USB2 and 1 USB3)
- 2GB RAM
- 32GB Flash
- Flash card slot
- WiFi
- Battery with 4 hours running time
- Power supply
- Windows 10
- and a fingerprint reader

The advantage of this setup is that it offloads the roboRIO from doing image processing and it is a normal Windows system so all of our software should work without modification. Be sure to read the caveats at the end of this page before jumping in.

More detailed instructions for using a Kangaroo for running GRIP can be found in the following PDF document created by Scott Taylor and FRC 1735. His explanation goes beyond what is shown here, detailing how to get the GRIP program to auto-start on boot and many other details.

[Grip Plus Kangaroo](#)

Setup

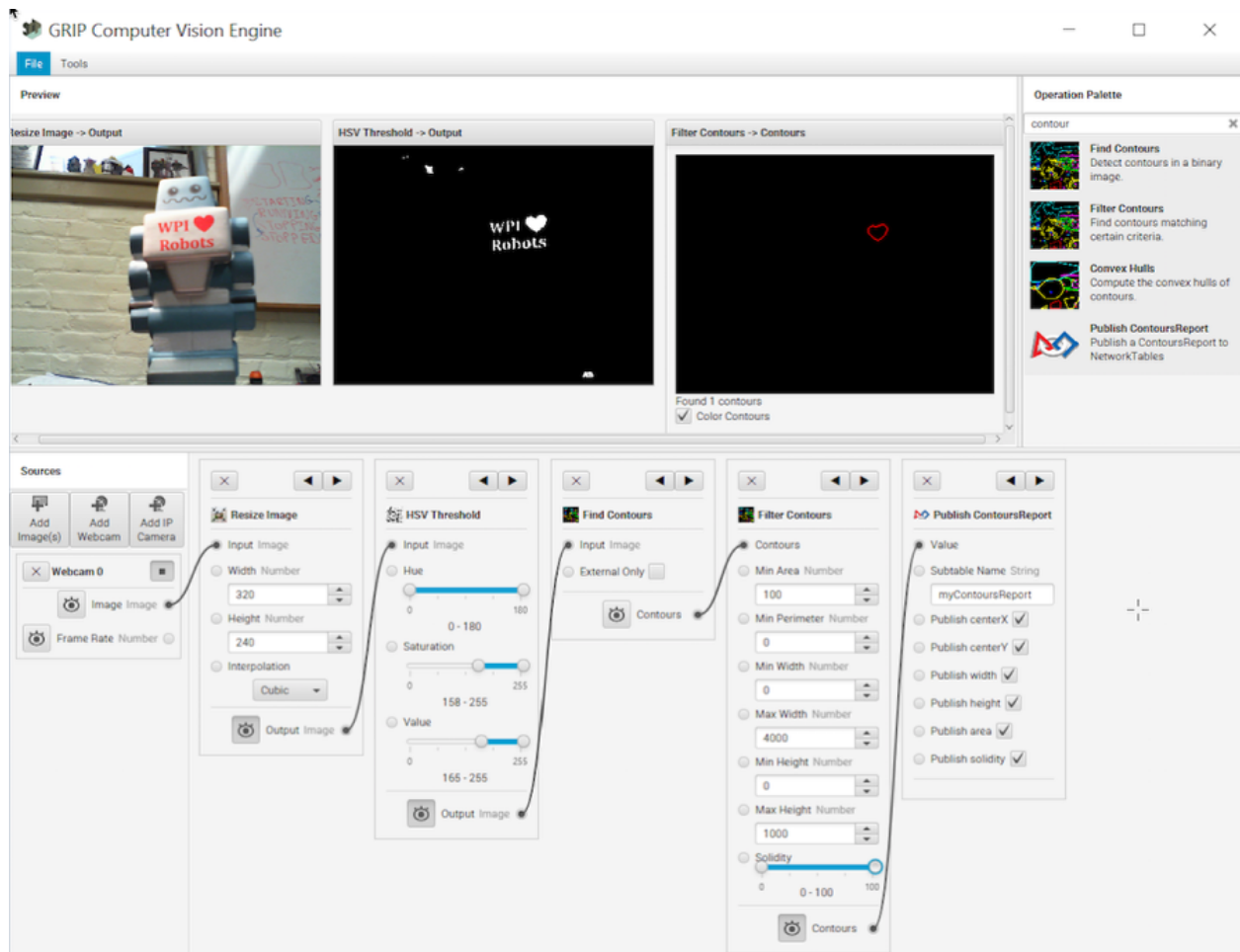


The nice thing about this setup is that you just need to plug in a monitor, keyboard, mouse and (in this case) the Microsoft web camera and you are good to go with programming the GRIP pipeline. When you are finished, disconnect the keyboard, mouse and monitor and put the Kangaroo on your robot. You will need to disable the WiFi on the Kangaroo and connect it to the robot with a USB ethernet dongle to the extra ethernet port on the robot radio.

In this example you can see the Kangaroo computer (1) connected to a USB hub (3), keyboard, and an HDMI monitor for programming. The USB hub is connected to the camera and mouse.

Sample GRIP program

Attached is the sample program running on the Kangaroo detecting the red heart on the little foam robot in the image (left panel). It is doing a HSV threshold to only get that red color then finding contours, and then filtering the contours using the size and solidity. At the end of the pipeline, the values are being published to NetworkTables.



Viewing Contours Report in NetworkTables

Key	Value	Type
Root		
GRIP		
myContoursReport		
centerX	[211.0]	Number[1]
centerY	[80.0]	Number[1]
height	[16.0]	Number[1]
area	[194.0]	Number[1]
width	[20.0]	Number[1]
solidity	[0.9603960396039604]	Number[1]

This is the output from the OutlineViewer (<username>/WPILib/tools/OutlineViewer.jar), running on a different computer as a server (since there is no roboRIO on the network in this example) and the values being reported back for the single contour that the program detected that met the requirements of the Filter Contours operation.

Considerations

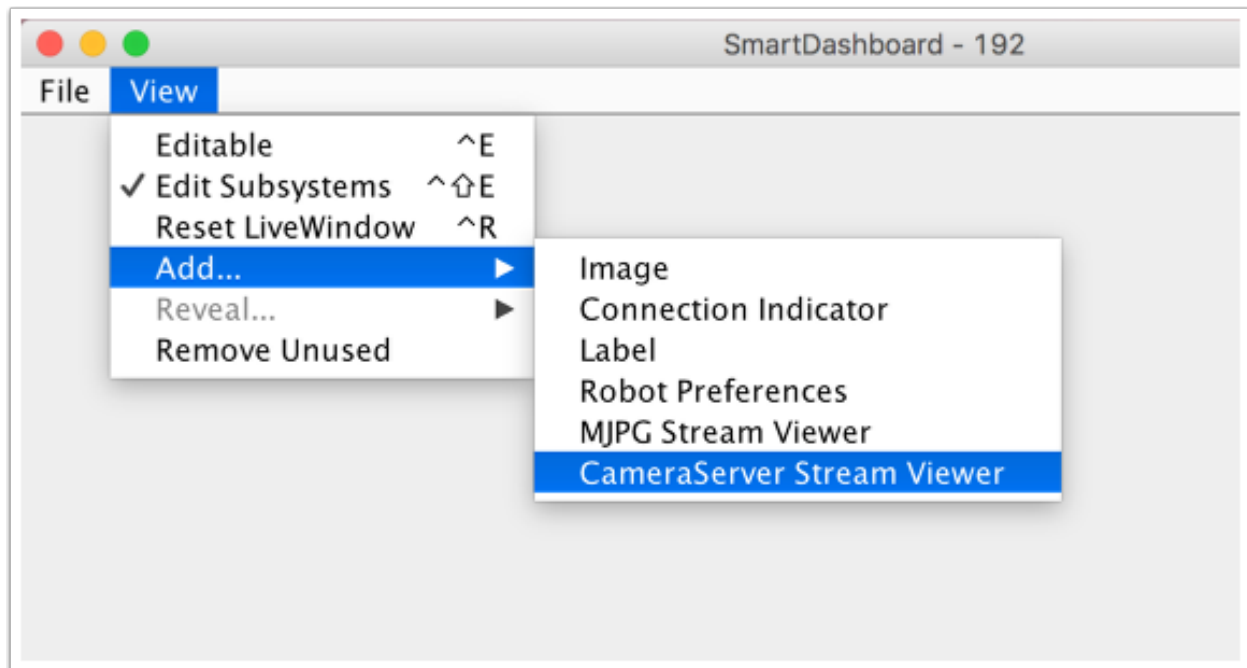
The Kangaroo runs Windows 10, so care must be taken to make sure GRIP will keep running on the robot during a match or testing. For example, it should not try to do a Windows Update, Virus scan refresh, go to sleep, etc. Once configured, it has the advantage of being a normal Intel Architecture and should give predictable performance since it is running only one application.

28.4 Vision on the RoboRIO

28.4.1 Using the CameraServer on the roboRIO

Simple CameraServer Program

The following program starts automatic capture of a USB camera like the Microsoft LifeCam that is connected to the roboRIO. In this mode, the camera will capture frames and send them to the dashboard. To view the images, create a CameraServer Stream Viewer widget using the “View”, then “Add” menu in the dashboard. The images are unprocessed and just forwarded from the camera to the dashboard.



Java

C++

```
package org.usfirst.frc.team190.robot;

import edu.wpi.first.cameraserver.CameraServer;
import edu.wpi.first.wpilibj.IterativeRobot;

public class Robot extends IterativeRobot {

    public void robotInit() {
        CameraServer.getInstance().startAutomaticCapture();
    }
}
```

```
#include "cameraserver/CameraServer.h"
class Robot: public IterativeRobot
{
private:
    void RobotInit()
    {
        CameraServer::GetInstance()->StartAutomaticCapture();
    }
};
START_ROBOT_CLASS(Robot)
```


Advanced Camera Server Program

In the following example a thread created in `robotInit()` gets the Camera Server instance. Each frame of the video is individually processed, in this case converting a color image (BGR) to gray scale using the OpenCV `cvtColor()` method. The resultant images are then passed to the output stream and sent to the dashboard. You can replace the `cvtColor` operation with any image processing code that is necessary for your application. You can even annotate the image using OpenCV methods to write targeting information onto the image being sent to the dashboard.

Java

C++

```
package org.usfirst.frc.team190.robot;

import org.opencv.core.Mat;
import org.opencv.imgproc.Imgproc;

import edu.wpi.cscore.CvSink;
import edu.wpi.cscore.CvSource;
import edu.wpi.cscore.UsbCamera;
import edu.wpi.first.cameraserver.CameraServer;
import edu.wpi.first.wpilibj.IterativeRobot;

public class Robot extends IterativeRobot {

    public void robotInit() {
        new Thread(() -> {
            UsbCamera camera = CameraServer.getInstance().startAutomaticCapture();
            camera.setResolution(640, 480);

            CvSink cvSink = CameraServer.getInstance().getVideo();
            CvSource outputStream = CameraServer.getInstance().putVideo("Blur", 640, 480);

            Mat source = new Mat();
            Mat output = new Mat();

            while(!Thread.interrupted()) {
                if (cvSink.grabFrame(source) == 0) {
                    continue;
                }
                Imgproc.cvtColor(source, output, Imgproc.COLOR_BGR2GRAY);
                outputStream.putFrame(output);
            }
        }).start();
    }
}
```

```
#include "cameraserver/CameraServer.h"
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/core/core.hpp>
class Robot: public IterativeRobot
{
private:
    static void VisionThread()
    {
```

(continues on next page)

(continued from previous page)

```

cs::UsbCamera camera = CameraServer::GetInstance()->StartAutomaticCapture();
camera.SetResolution(640, 480);
cs::CvSink cvSink = CameraServer::GetInstance()->GetVideo();
cs::CvSource outputStreamStd = CameraServer::GetInstance()->PutVideo("Gray", 640,
↪480);
cv::Mat source;
cv::Mat output;
while(true) {
    if (cvSink.GrabFrame(source) == 0) {
        continue;
    }
    cvtColor(source, output, cv::COLOR_BGR2GRAY);
    outputStreamStd.PutFrame(output);
}
}
void RobotInit()
{
    std::thread visionThread(VisionThread);
    visionThread.detach();
}
};
START_ROBOT_CLASS(Robot)

```

Notice that in these examples, the `PutVideo()` method writes the video to a named stream. To view that stream on Shuffleboard, select that named stream. In this case that is “Blur” for the Java program and “Gray” for the C++ sample.

28.4.2 Using Multiple Cameras

Switching the Driver Views

If you’re interested in just switching what the driver sees, and are using SmartDashboard, the SmartDashboard CameraServer Stream Viewer has an option (“Selected Camera Path”) that reads the given [NetworkTables](#) key and changes the “Camera Choice” to that value (displaying that camera). The robot code then just needs to set the [NetworkTables](#) key to the correct camera name. Assuming “Selected Camera Path” is set to “CameraSelection”, the following code uses the joystick 1 trigger button state to show camera1 and camera2.

Java

C++

```

UsbCamera camera1;
UsbCamera camera2;
Joystick joy1 = new Joystick(0);
NetworkTableEntry cameraSelection;

@Override
public void robotInit() {
    camera1 = CameraServer.getInstance().startAutomaticCapture(0);
    camera2 = CameraServer.getInstance().startAutomaticCapture(1);

    cameraSelection = NetworkTableInstance.getDefault().getTable("").getEntry(
↪"CameraSelection");

```

(continues on next page)

(continued from previous page)

```

}

@Override
public void teleopPeriodic() {
    if (joy1.getTriggerPressed()) {
        System.out.println("Setting camera 2");
        cameraSelection.setString(camera2.getName());
    } else if (joy1.getTriggerReleased()) {
        System.out.println("Setting camera 1");
        cameraSelection.setString(camera1.getName());
    }
}
}

```

```

cs::UsbCamera camera1;
cs::UsbCamera camera2;
frc::Joystick joy1{0};

nt::NetworkTableEntry cameraSelection;

void RobotInit() override {
    camera1 = frc::CameraServer::GetInstance()->StartAutomaticCapture(0);
    camera2 = frc::CameraServer::GetInstance()->StartAutomaticCapture(1);

    cameraSelection = nt::NetworkTableInstance::GetDefault().GetTable("")->GetEntry(
    ↪ "CameraSelection");
}

void TeleopPeriodic() override {
    if (joy1.GetTriggerPressed()) {
        std::cout << "Setting Camera 2" << std::endl;
        cameraSelection.SetString(camera2.GetName());
    } else if (joy1.GetTriggerReleased()) {
        std::cout << "Setting Camera 1" << std::endl;
        cameraSelection.SetString(camera1.GetName());
    }
}
}

```

If you're using some other dashboard, you can change the camera used by the camera server dynamically. If you open a stream viewer nominally to camera1, the robot code will change the stream contents to either camera1 or camera2 based on the joystick trigger.

Java

C++

```

UsbCamera camera1;
UsbCamera camera2;
VideoSink server;
Joystick joy1 = new Joystick(0);

@Override
public void robotInit() {
    camera1 = CameraServer.getInstance().startAutomaticCapture(0);
    camera2 = CameraServer.getInstance().startAutomaticCapture(1);
    server = CameraServer.getInstance().getServer();
}

```

(continues on next page)

(continued from previous page)

```
@Override
public void teleopPeriodic() {
    if (joy1.getTriggerPressed()) {
        System.out.println("Setting camera 2");
        server.setSource(camera2);
    } else if (joy1.getTriggerReleased()) {
        System.out.println("Setting camera 1");
        server.setSource(camera1);
    }
}
```

```
cs::UsbCamera camera1;
cs::UsbCamera camera2;
cs::VideoSink server;
frc::Joystick joy1{0};
bool prevTrigger = false;

void RobotInit() override {
    camera1 = frc::CameraServer::GetInstance()->StartAutomaticCapture(0);
    camera2 = frc::CameraServer::GetInstance()->StartAutomaticCapture(1);
    server = frc::CameraServer::GetInstance()->GetServer();
}

void TeleopPeriodic() override {
    if (joy1.GetTrigger() && !prevTrigger) {
        std::cout << "Setting Camera 2" << std::endl;
        server.SetSource(camera2);
    } else if (!joy1.GetTrigger() && prevTrigger) {
        std::cout << "Setting Camera 1" << std::endl;
        server.SetSource(camera1);
    }
    prevTrigger = joy1.GetTrigger();
}
```

Keeping Streams Open

By default, the cscore library is pretty aggressive in turning off cameras not in use. What this means is that when you switch cameras, it may disconnect from the camera not in use, so switching back will have some delay as it reconnects to the camera. To keep both camera connections open, use the `SetConnectionStrategy()` method to tell the library to keep the streams open, even if you aren't using them.

Java

C++

```
UsbCamera camera1;
UsbCamera camera2;
VideoSink server;
Joystick joy1 = new Joystick(0);

@Override
public void robotInit() {
    camera1 = CameraServer.getInstance().startAutomaticCapture(0);
    camera2 = CameraServer.getInstance().startAutomaticCapture(1);
}
```

(continues on next page)

(continued from previous page)

```

server = CameraServer.getInstance().getServer();

camera1.setConnectionStrategy(ConnectionStrategy.kKeepOpen);
camera2.setConnectionStrategy(ConnectionStrategy.kKeepOpen);
}

@Override
public void teleopPeriodic() {
    if (joy1.getTriggerPressed()) {
        System.out.println("Setting camera 2");
        server.setSource(camera2);
    } else if (joy1.getTriggerReleased()) {
        System.out.println("Setting camera 1");
        server.setSource(camera1);
    }
}
}

cs::UsbCamera camera1;
cs::UsbCamera camera2;
cs::VideoSink server;
frc::Joystick joy1{0};
bool prevTrigger = false;
void RobotInit() override {
    camera1 = frc::CameraServer::GetInstance()->StartAutomaticCapture(0);
    camera2 = frc::CameraServer::GetInstance()->StartAutomaticCapture(1);
    server = frc::CameraServer::GetInstance()->GetServer();
    camera1.
    ↪SetConnectionStrategy(cs::VideoSource::ConnectionStrategy::kConnectionKeepOpen);
    camera2.
    ↪SetConnectionStrategy(cs::VideoSource::ConnectionStrategy::kConnectionKeepOpen);
}

void TeleopPeriodic() override {
    if (joy1.GetTrigger() && !prevTrigger) {
        std::cout << "Setting Camera 2" << std::endl;
        server.SetSource(camera2);
    } else if (!joy1.GetTrigger() && prevTrigger) {
        std::cout << "Setting Camera 1" << std::endl;
        server.SetSource(camera1);
    }
    prevTrigger = joy1.GetTrigger();
}
}

```

Note: If both cameras are USB, you may run into USB bandwidth limitations with higher resolutions, as in all of these cases the roboRIO is going to be streaming data from both cameras to the roboRIO simultaneously (for a short period in options 1 and 2, and continuously in option 3). It is theoretically possible for the library to avoid this simultaneity in the option 2 case (only), but this is not currently implemented.

Different cameras report bandwidth usage differently. The library will tell you if you're hitting the limit; you'll get this error message:

```

could not start streaming due to USB bandwidth limitations;
try a lower resolution or a different pixel format
(VIDIIOC_STREAMON: No space left on device)

```

If you're using Option 3 it will give you this error during RobotInit(). Thus you should just try your desired resolution and adjusting as necessary until you both don't get that error and don't exceed the radio bandwidth limitations.

28.5 Vision with an Axis Camera

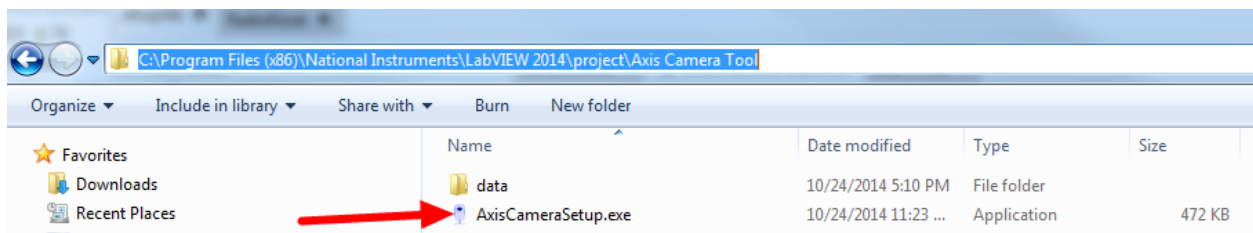
28.5.1 Configuring an Axis Camera

Note: Three different Axis camera models are supported by the FRC® software, the Axis 206, Axis M1011 and Axis M1013. This document provides instructions on how to configure one of these cameras for FRC use. To follow the instructions in this document, you must have installed the NI FRC Game Tools and Configured your radio

Connect the camera

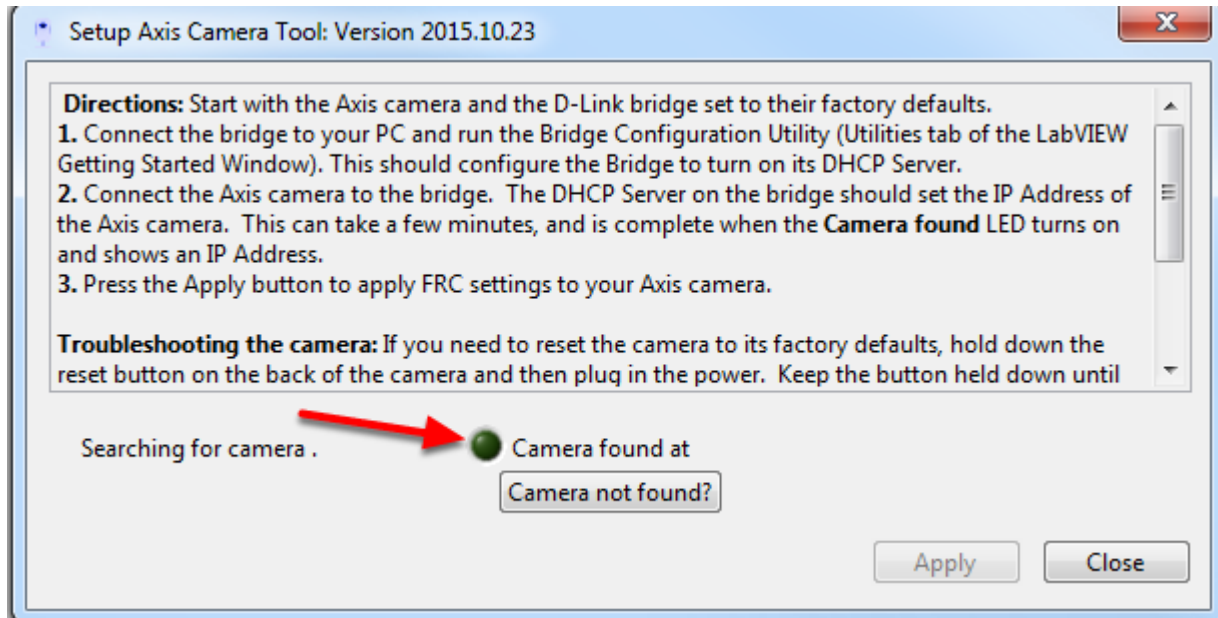
Connect the Axis camera to the radio using an Ethernet cable. Connect your computer to the radio using an Ethernet cable or via a wireless connection.

Axis Camera Setup Tool



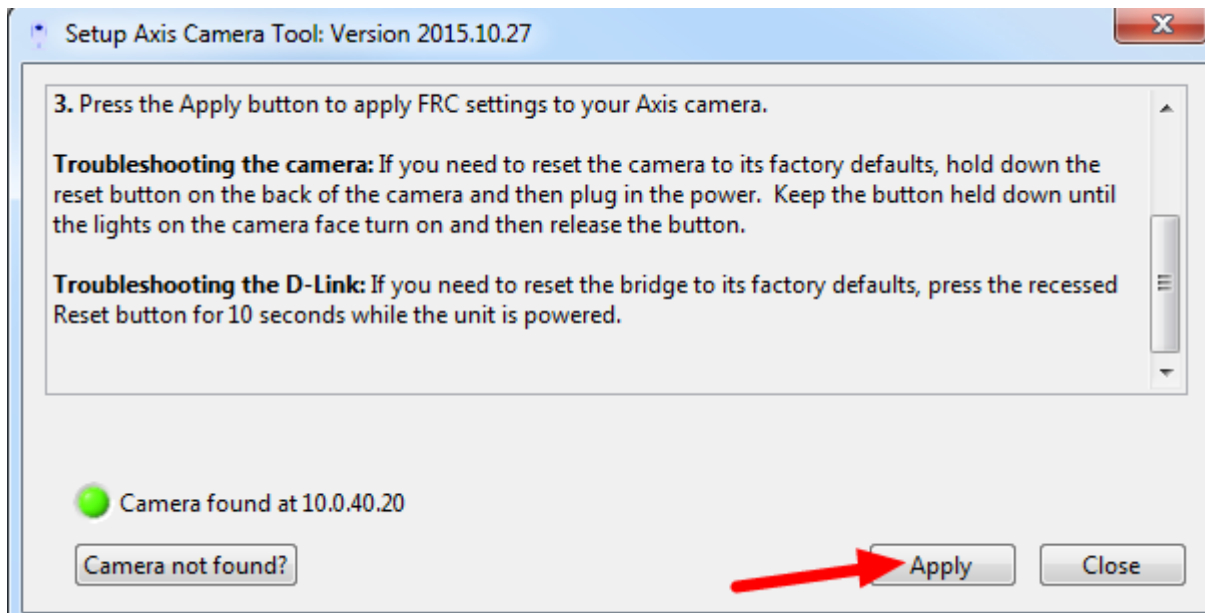
Browse to C:\Program Files (x86)\National Instruments\LabVIEW 2019\project\Axis Camera Tool and double-click on AxisCameraSetup.exe to start the Axis Camera Setup Tool.

Tool Overview



The camera should be automatically detected and the green indicator light should be lit. If it is not, make sure the camera is powered on (the ring on the camera face should be green) and connected to your computer. If the indicator remains off follow the instructions in the tool textbox next to Troubleshooting the camera to reset the camera. You can also use the “Camera not found?” button to check the IP address of your computer; one of the addresses listed should be of the form 10.TE.AM.XX where TEAM is your 4 digit team number.

Setup the Camera



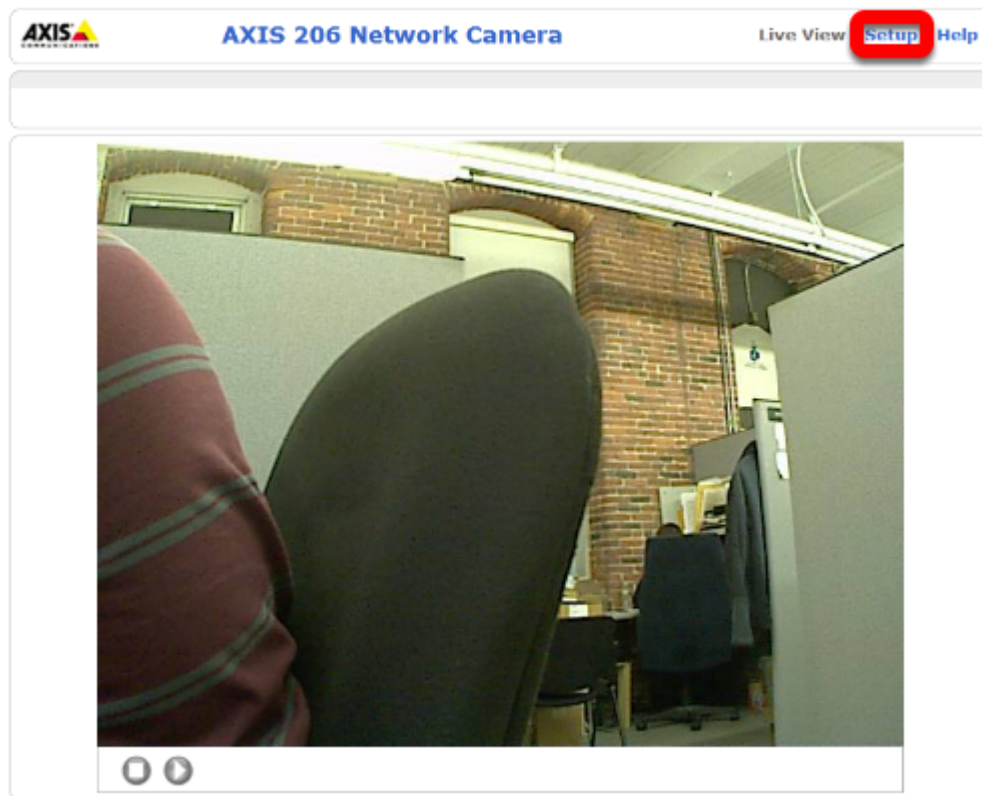
To configure the camera, press Apply. This will configure many of the neces-

sary/recommended settings for using the camera for FRC. Currently the tool does not properly configure the DNS name of the camera in many cases.

Camera Webpage

To set the network settings, open a web browser and enter the address shown next to Camera found at in the tool (in the example above this is 10.0.40.20) in the address bar and press enter. You should see a Configure Root Password page, set this password to whatever you would like, but admin is recommended.

Setup Page



Click Setup to go to the setup page.

Configure Basic Network Settings

AXIS 206 Network Camera Live View | Setup | Help

Basic TCP/IP Settings ?

Network Settings

View current network settings: View

IP Address Configuration

2 ☒ Obtain IP address via DHCP

☐ Use the following IP address:

IP address: Test

Subnet mask:

Default router:

Services

☒ Enable ARP/Ping setting of IP Address

Options for notification of IP address change Settings...

AXIS Internet Dynamic DNS Service Settings...

3 Save Reset

System Options

- Users
- Date & Time
- Network
 - 1 **Basic**
 - Advanced
 - QoS
 - SMTP (email)
 - UPnP
 - Bonjour
 - LED settings
 - Maintenance
- Support
- Advanced

Language

About

To configure the network settings of the camera, click the arrow to expand the System Options pane, then click the arrow to expand Network, then expand TCP/IP and select Basic. Set the camera to obtain an IP address via DHCP by selecting the bubble. Alternately, you may choose to set a static IP in the range 10.TE.AM.3 to 10.TE.AM.19. This is outside the range handed out by the radio (home use) or FMS system (event use) so you will avoid any IP conflicts.

Click Save.

Configure Advanced Network Settings

AXIS 206 Network Camera Live View | Setup | Help

Advanced TCP/IP Settings

Basic Configuration

Video & Image

Live View Config

System Options

- Users
- Date & Time
- Network
 - TCP/IP
 - Basic
 - Advanced**
 - QoS
 - SMTP (email)
 - UPnP
 - Bonjour
 - LED settings
 - Maintenance
 - Support
 - Advanced

Language

About

DNS Configuration

☒ Obtain DNS server address via DHCP View

☐ Use the following DNS server address:

Domain name: (use ; to separate names)

Primary DNS server:

Secondary DNS server:

NTP Configuration

☒ Obtain NTP server address via DHCP View

☐ Use the following NTP server address:

Network address: (host name or IP address)

Host Name Configuration

☒ Obtain host name via DHCP View

☒ Use the host name:

☐ Enable dynamic DNS updates

Register DNS name: (Axisproduct.example.com)

TTL:

Link-Local Address

☒ Auto-Configure Link-Local Address View

HTTP

HTTP port:

NAT traversal (port mapping)

NAT traversal is disabled. Enable

☐ Use manually selected NAT router: (LAN IP address)

Alternative HTTP port: *

* If left blank, a port number will be set automatically upon enable.

FTP

☒ Enable FTP server

Network Traffic

Connection type:

Save Reset

Next click Advanced under TCP/IP. Set the Host Name Configuration to “Use the host name:” and set the value to axis-camera as shown. If you plan to use multiple cameras on your robot, select a unique host name for each. You will need to modify the dashboard and/or robot code to work with the additional cameras and unique host names.

Click Save.

Manual Camera Configuration

It is recommended to use the Setup Axis Camera Tool to configure the Axis Camera. If you need to configure the camera manually, connect the camera directly to the computer, configure your computer to have a static IP of 192.168.0.5, then open a web browser and enter 192.168.0.90 in the address bar and press enter. You should see a Configure Root Password page, set this password to whatever you would like, but admin is recommended.

If you do not see the camera webpage come up, you may need to reset the camera to factory defaults. To do this, remove power from the camera, hold the reset button while applying power to the camera and continue holding it until the lights on the camera face turn on, then release the reset button and wait for the lights to turn green. The camera is now reset to factory settings and should be accessible via the 192.168.0.90 address.

Click Setup to go to the setup page.

Manual - Configure Users

AXIS 206 Network Camera Live View | Setup | Help

Basic Configuration

- 1. Users**
- 2. Password
- 3. Date & Time
- 4. Video & Image

Video & Image

Live View Config

System Options

Language

About

Users

User List

User Name	User Group
root	Administrator

Add... Modify... Remove

User Settings

☐ Enable anonymous viewer login (no user name or password required)

Maximum number of simultaneous viewers limited to: 10 [0..10]

Subsequent viewers will see a blank image.

Save Reset

On the left side click Users to open the users page. Click Add then enter the Username FRC Password FRC and click the Administrator bubble, then click OK. If using the SmartDashboard, check the Enable anonymous viewer login box. Then click Save.

Manual - Configure Image Settings

AXIS 206 Network Camera Live View | Setup | Help

Image Settings ?

Image Appearance

Resolution: 320x240 pixels

Compression: 30 [0..100]

Rotate image: 0 degrees

Color level: 50 [0..100] *

Brightness: 50 [0..100] (Does not affect Test image)

Sharpness: 0 (Does not affect Test image)

* Changes to color level do not affect Test image (exception 0 = B/W)

Overlay Settings

☐ Include date ☐ Include time

☐ Include text: (Does not affect Test image)

Place text/date/time at: top of image

Video Stream

Maximum video stream time:

☒ Unlimited

☐ Limited to [1..] seconds per session

Maximum frame rate:

☒ Unlimited

☐ Limited to [1..30] fps per viewer

Test

Test settings before saving. Test

Save Reset

Click Video & Image on the left side to open the image settings page. Set the Resolution and Compression to the desired values (recommended 320x240, 30). To limit the framerate to under 30 FPS, select the Limited to bubble under Maximum frame rate and enter the desired rate in the box. Color, Brightness and Sharpness may also be set on this screen if desired. Click Save when finished.

Manual - Configure Basic Network Settings

To configure the network settings of the camera, click the arrow to expand the System Options pane, then click the arrow to expand Network, then expand TCP/IP and select Basic. Set the camera to obtain an IP address via DHCP by selecting the bubble. Alternately, you may choose to set a static IP in the range 10.TE.AM.3 to 10.TE.AM.19. This is outside the range handed out by the radio (home use) or FMS system (event use) so you will avoid any IP conflicts.

Click Save.

Manual - Configure Advanced Network Settings

Next click Advanced under TCP/IP. Set the Host Name Configuration to “Use the host name:” and set the value to `axis-camera` as shown. If you plan to use multiple cameras on your robot, select a unique host name for each. You will need to modify the dashboard and/or robot code to work with the additional cameras and unique host names.

Click Save.

28.5.2 Axis M1013 Camera Compatibility

The Axis M1011 camera has been discontinued and superseded by the Axis M1013 camera. This document details the differences between the two cameras when used with WPILib and the provided sample vision programs.

Optical Differences

The Axis M1013 camera has a few major optical differences from the M1011 camera:

1. The M1013 is an adjustable focus camera. Make sure to focus your M1013 camera by turning the grey and black lens housing to make sure you have a clear image at your desired viewing distance.
2. The M1013 has a wider view angle (67°) compared to the M1011 (47°). This means that for a feature of a fixed size, the image of that feature will span a smaller number of pixels

Using the M1013 with WPILib

The M1013 camera has been tested with all of the available WPILib parameters and the following performance exceptions were noted:

1. The M1013 does not support the 160x120 resolution. Requesting a stream of this resolution will result in no images being returned or displayed.
2. The M1013 does not appear to work with the Color Enable parameter exposed by WPILib. Regardless of the setting of this parameter a full color image was returned.

All other WPILib camera parameters worked as expected. If any issues not noted here are discovered, please file a bug report on GitHub.

28.5.3 Camera Settings

It is very difficult to achieve good image processing results without good images. With a light mounted near the camera lens, you should be able to use the provided examples, the LabVIEW Dashboard or Shuffleboard, NI Vision Assistant or a web browser to view camera images and experiment with camera settings.

Changing Camera Settings

- Basic Setup **M1013**
- Video
 - Video Stream**
 - Stream Profiles
 - Camera Settings**
 - Overlay Image
 - Privacy Mask
- Live View Config
- Detectors
- Events
- Recordings
- System Options
- About

Camera Settings

View Area

☐ Enable View Area

Image Appearance

Color level: 50 [0..100]

Brightness: 50 [0..100]

Sharpness: 50 [0..100]

Contrast: 50 [0..100]

White Balance

White balance: Fixed Outdoor 1

Exposure Settings

Exposure value: 50 [0..100]

Enable Backlight compensation: ☐

Exposure priority: Default

View Image Settings

- Basic Config: **206**
M1011
- Video & Image
 - Video & Image**
 - Advanced**
- Live View Config
- System Options
- Language
- About

Image Settings

Image Appearance

Resolution: 640x480 pixels

Compression: 30 [0..100]

Rotate image: 0 degrees

Color level: 50 [0..100] *

Brightness: 50 [0..100] (Does not affect Test image)

Sharpness: 0 (Does not affect Test image)

* Changes to color level do not affect Test image (exception 0 = B/W)

Overlay Settings

☐ Include date ☐ Include time

☐ Include text: (Does not affect Test image)

Place text/date/time at top of image

Video Stream

To change the camera settings on any of the supported Axis cameras (206, M1011, M1013), browse to the camera's webpage by entering its address (usually 10.TE.AM.11) in a web browser. Click Setup near the top right corner of the page. On the M1013, the settings listed below are split between the Video Stream page and the Camera Settings page, both listed under the Video section.

Focus

The Axis M1011 has a fixed-focus lens and no adjustment is needed. The Axis 206 camera has a black bezel around the lens that rotates to move the lens in and out and adjust focus. The Axis M103 has a silver and black bezel assembly around the lens to adjust the focus. Ensure that the images you are processing are relatively sharp and focused for the distances needed on your robot.

Compression



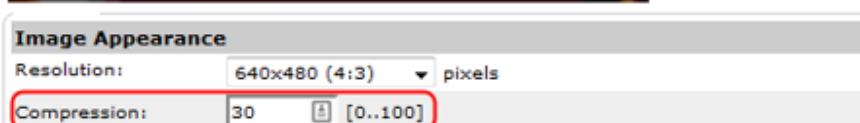
320x240 Color Image:
Compression set to 0. Image file size is 20,715 bytes.
High quality, but large in size. May be slower to compress and decompress - which impacts frame rate.



320x240 Color Image:
Compression set to 30. Image file size is 6,450 bytes. Good quality, relatively small in size. Some image artifacts are present on edges.



320x240 Color Image:
Compression set to 100. Image file size is 2,222 bytes. Poor quality for processing. Notice blocky artifacts and rough edges.



The Axis camera returns images in BMP, JPEG, or MJPEG format. BMP images are quite large and take more time to transmit to the cRIO and laptop. Therefore the WPILib implementations typically use MJPEG (motion JPEG). The compression setting ranges from 0 to 100, with 0 being very high quality images with very little compression, and 100 being very low quality images with very high compression. The camera default is 30, and it is a good compromise, with few artifacts that will degrade image processing.

Note: Teams are advised to consider how the compression setting on the camera affects bandwidth if performing processing on the Driver Station computer, see the FMS Whitepaper for more details.

Resolution

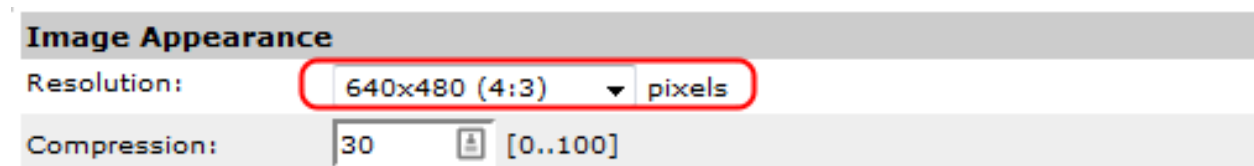


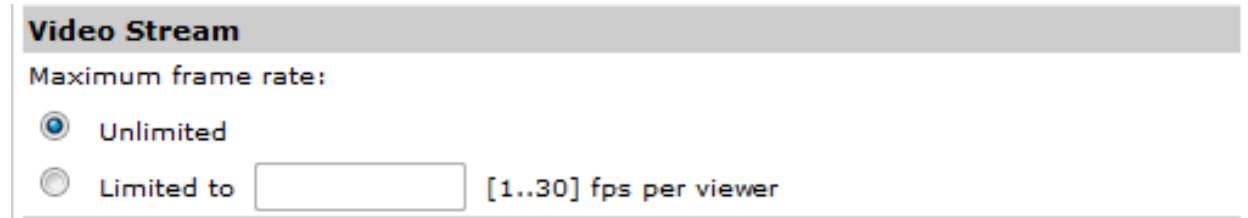
Image sizes shared by the supported cameras are 160x120, 320x240, and 640x480. The M1011 and 1013 have additional sizes, but they aren't built into WPILib. The largest image size has four times as many pixels that are one-fourth the size of the middle size image. The large image has sixteen times as many pixels as the small image.

The tape used on the target is 4 inches (~10 cm) wide, and for good processing, you will want that 4 inch (~10 cm) feature to be at least two pixels wide. Using the distance equations above, we can see that a medium size image should be fine up to the point where the field of view is around 640 inches (~16 m), a little over 53 feet (~16 m), which is nearly double the width of the FRC® field. This occurs at around 60 feet (~18 m) away, longer than the length of the field. The small image size should be usable for processing to a distance of about 30 feet (~9 m) or a little over mid-field.

Image size also impacts the time to decode and to process. Smaller images will be roughly four times faster than the next size up. If the robot or target is moving, it is quite important to minimize image processing time since this will add to the delay between the target location and perceived location. If both robot and target are stationary, processing time is typically less important.

Note: When requesting images using LabVIEW (either the Dashboard or Robot Code), the resolution and Frame Rate settings of the camera will be ignored. The LabVIEW code specifies the framerate and resolution as part of the stream request (this does not change the settings stored in the camera, it overrides that setting for the specific stream). The SmartDashboard and robot code in C++ or Java will use the resolution and framerate stored in the camera.

Frame Rate



Video Stream

Maximum frame rate:

☒ Unlimited

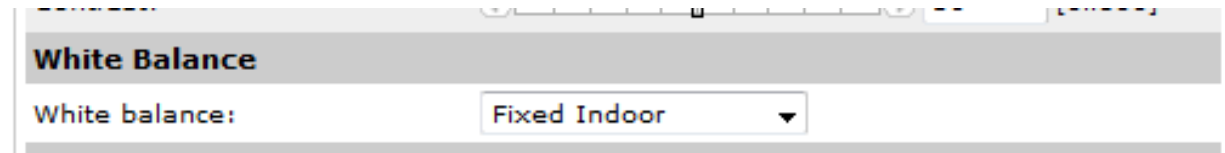
☐ Limited to [1..30] fps per viewer

The Axis Cameras have a max framerate of 30 frames per second. If desired, a limit can be set lower to reduce bandwidth consumption.

Color Enable

The Axis cameras typically return color images, but are capable of disabling color and returning a monochrome or grayscale image. The resulting image is a bit smaller in file size, and considerably quicker to decode. If processing is carried out only on the brightness or luminance of the image, and the color of the ring light is not used, this may be a useful technique for increasing the frame rate or lowering the CPU usage.

White Balance



White Balance

White balance: Fixed Indoor ▼

If the color of the light shine is being used to identify the marker, be sure to control the camera settings that affect the image coloring. The most important setting is white balance. It controls how the camera blends the component colors of the sensor in order to produce an image that matches the color processing of the human brain. The camera has five or six named presets, an auto setting that constantly adapts to the environment, and a hold setting – for custom calibration.

The easiest approach is to use a named preset, one that maintains the saturation of the target and doesn't introduce problems by tinting neutral objects with the color of the light source.

To custom-calibrate the white balance, place a known neutral object in front of the camera. A sheet of white paper is a reasonable object to start with. Set the white balance setting to auto, wait for the camera to update its filters (ten seconds or so), and switch the white balance to hold.

Exposure

Image Appearance **M1013**

Color level: [0..100]

Brightness: [0..100]

Sharpness: [0..100]

Contrast: [0..100]

White Balance

White balance: Fixed Indoor

Exposure Settings

Exposure value: [0..100]

Enable Backlight compensation: ☐

Exposure priority: Motion

Lighting Conditions **206/M1011**

White balance: Automatic

Exposure control: Automatic

Low Light Behavior

Exposure priority: None

The brightness or exposure of the image also has an impact on the colors being reported. The issue is that as overall brightness increases, color saturation will start to drop. Let's look at an example to see how this occurs. A saturated red object placed in front of the camera will return an RGB measurement high in red and low in the other two e.g. (220, 20, 30). As overall white lighting increases, the RGB value increases to (240, 40, 50), then (255, 80, 90), then (255, 120, 130), and then (255, 160, 170). Once the red component is maximized, additional light can only increase the blue and green, and acts to dilute the measured color and lower the saturation. If the point is to identify the red object, it is useful to adjust the exposure to avoid diluting your principal color. The desired image will look somewhat dark except for the colored shine.

There are two approaches to control camera exposure times. One is to allow the camera to compute the exposure settings automatically, based on its sensors, and then adjust the camera's brightness setting to a small number to lower the exposure time. The brightness setting acts similar to the exposure compensation setting on SLR cameras. The other approach is to calibrate the camera to use a custom exposure setting. To do this on a 206 or M1011, change the exposure setting to auto, expose the camera to bright lights so that it computes a short exposure, and then change the exposure setting to hold. Both approaches will result in an overall dark image with bright saturated target colors that stand out from the background and are easier to mask.

The M1013 exposure settings look a little different. The Enable Backlight compensation op-

tion is similar to the Auto exposure settings of the M1011 and 206 and you will usually want to un-check this box. Adjust the Brightness and Exposure value sliders until your image looks as desired. The Exposure Priority should generally be set to Motion. This will prioritize frame rate over image quality. Note that even with these settings the M1013 camera still performs some auto exposure compensation so it is recommended to check calibration frequently to minimize any impact lighting changes may have on image processing. See [Calibration](#) for more details.

28.5.4 Calibration

While many of the numbers for the Vision Processing code can be determined theoretically, there are a few parameters that are typically best to measure empirically then enter back into the code (a process typically known as calibration). This article will show how to perform calibration for the Color (masking), and View Angle (distance) using the NI Vision Assistant. If you are using C++ or Java and have not yet installed the NI Vision Assistant, see the article [Installing NI Vision Assistant](#).

Enable Snapshots

AXIS M1013 Network Camera Live View | **Setup** | Help

Live View Layout

Stream Profile

Stream profile: Motion JPEG

☒ Show stream profile selection

Default Viewer

Windows Internet Explorer: AMC (ActiveX)

Other Browsers: Server push

Note: QuickTime is only used with H.264. Motion JPEG will be shown with AMC in Windows Internet Explorer and with server push in other browsers.

Viewer Settings

☒ Show viewer toolbar

☒ Enable H.264 decoder installation

☒ Show crosshair in PTZ joystick mode*

☐ Use PTZ joystick mode as default*

☐ Enable recording button

* Not applicable to AMC (ActiveX).

Action Buttons

☐ Show manual trigger button

☒ **Show snapshot button**

User Defined Links

☐ Show custom link 1 Use as: ☒ cgi link ☐ web link
 Name: Custom link 1 URL: http://

☐ Show custom link 2 Use as: ☒ cgi link ☐ web link
 Name: Custom link 2 URL: http://

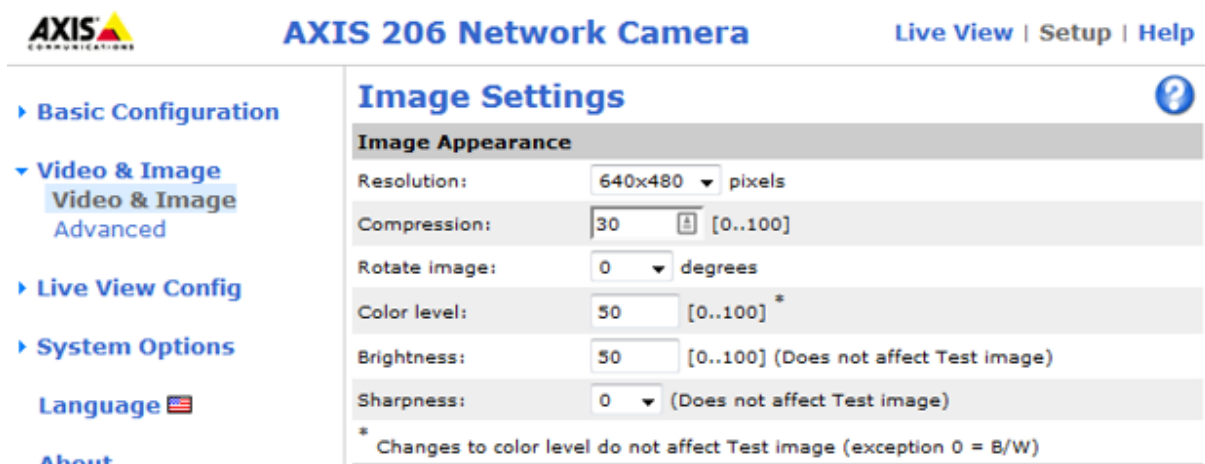
☐ Show custom link 3 Use as: ☒ cgi link ☐ web link
 Name: Custom link 3 URL: http://

☐ Show custom link 4 Use as: ☒ cgi link ☐ web link
 Name: Custom link 4 URL: http://

Save **Reset**

To capture snapshots from the Axis camera, you must first enable the Snapshot button. Open a web-browser and browse to camera's address (10.TE.AM.11), enter the Username/Password combo FRC/FRC if prompted, then click Setup->Live View Config->Layout. Click on the checkbox to Show snapshot button then click Save.

Check Camera Settings



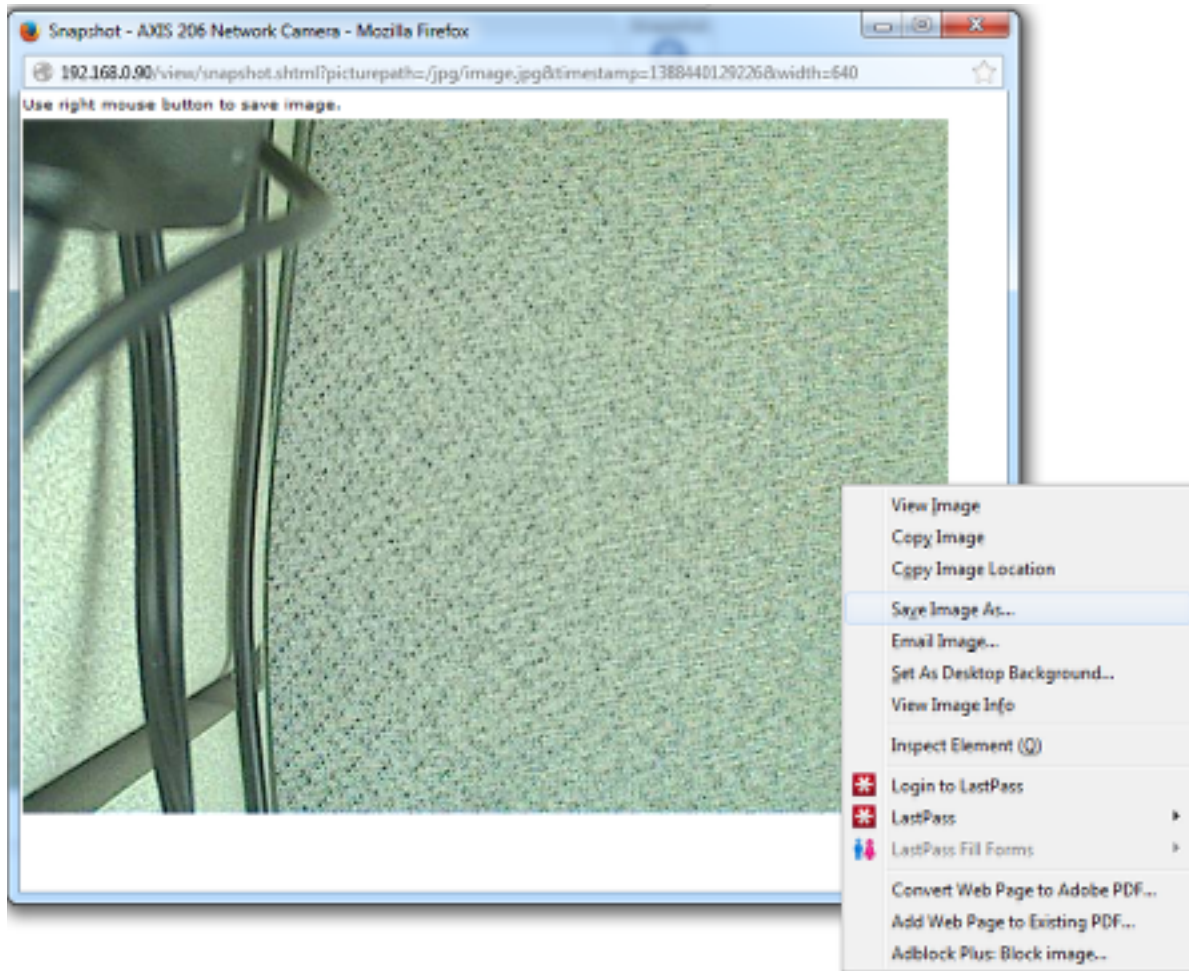
The screenshot shows the web interface for an AXIS 206 Network Camera. The top navigation bar includes the AXIS logo, the camera model name, and links for Live View, Setup, and Help. A left sidebar contains a menu with options: Basic Configuration, Video & Image (selected), Live View Config, System Options, Language, and About. The main content area is titled 'Image Settings' and features a sub-section 'Image Appearance'. This section contains several configuration fields: Resolution (640x480 pixels), Compression (30), Rotate image (0 degrees), Color level (50), Brightness (50), and Sharpness (0). Each field has a text input, a dropdown menu, and a range indicator. A note at the bottom states: '* Changes to color level do not affect Test image (exception 0 = B/W)'.

Image Appearance	
Resolution:	640x480 pixels
Compression:	30 [0..100]
Rotate image:	0 degrees
Color level:	50 [0..100] *
Brightness:	50 [0..100] (Does not affect Test image)
Sharpness:	0 (Does not affect Test image)

* Changes to color level do not affect Test image (exception 0 = B/W)

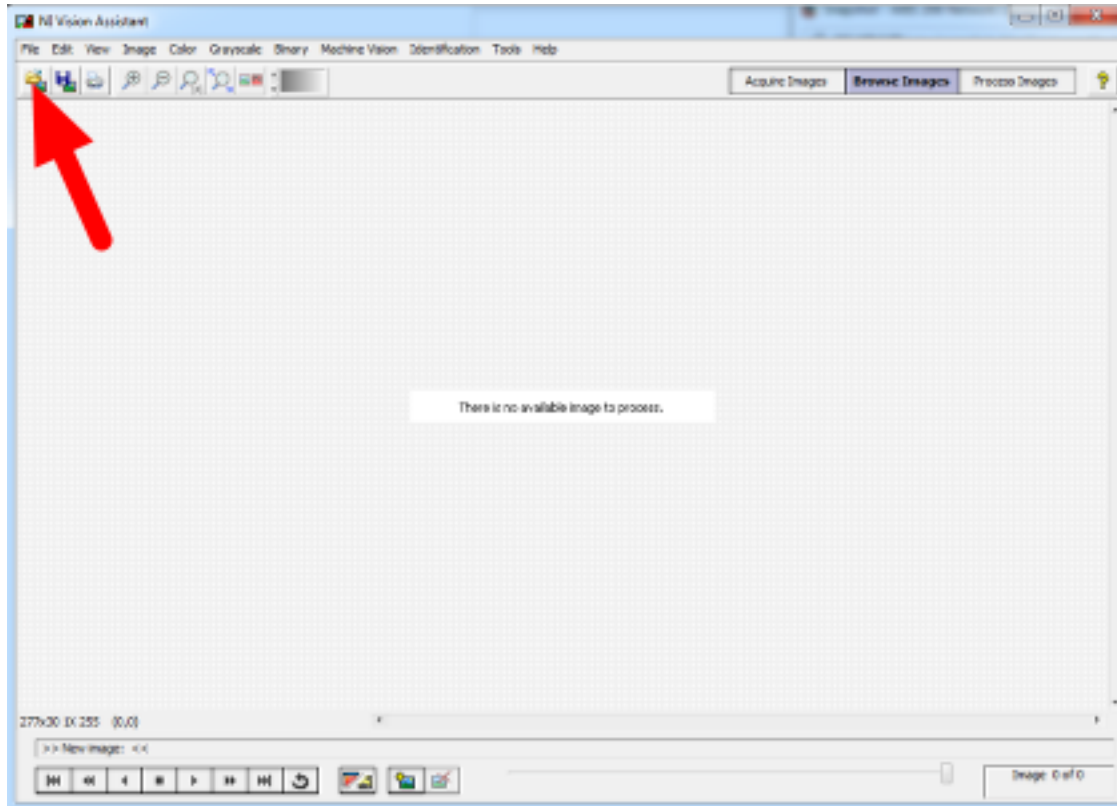
Depending on how you are capturing the image stream in your program, it may be possible to stream a different resolution, framerate and/or compression than what is saved in the camera and used in the Live View. Before performing any calibration it is recommended you verify that the settings in the camera match the settings in your code. To check the settings in the camera, click on the Video and Image header on the left side of the screen, then click Video and Image.

Capture Images



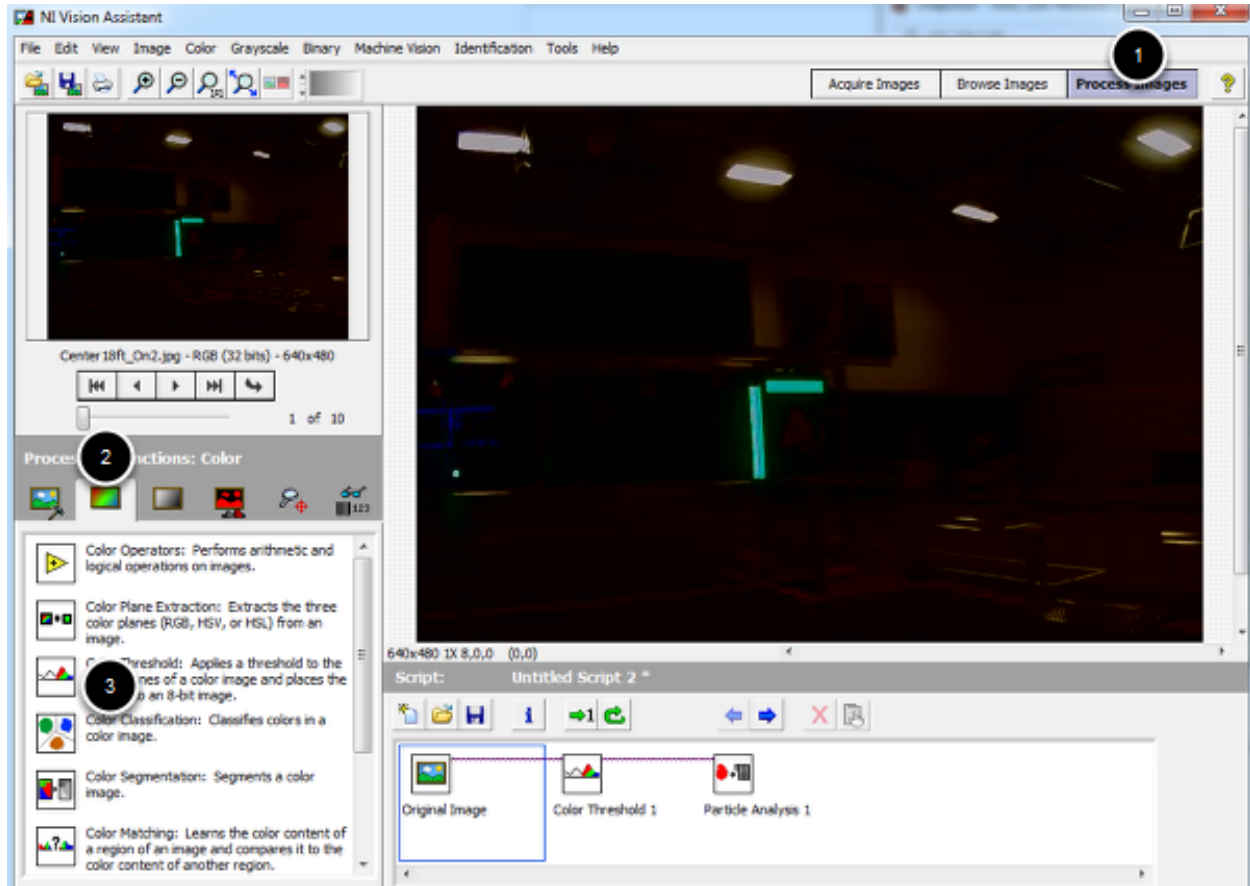
Click the Live View button to return to the Live View page and you should now see a Snapshot button. Clicking this button opens a pop-up window with a static image capture. Right-click on this image, select Save Image as and select your desired location and file name, then save the image.

Load Image(s) in Vision Assistant



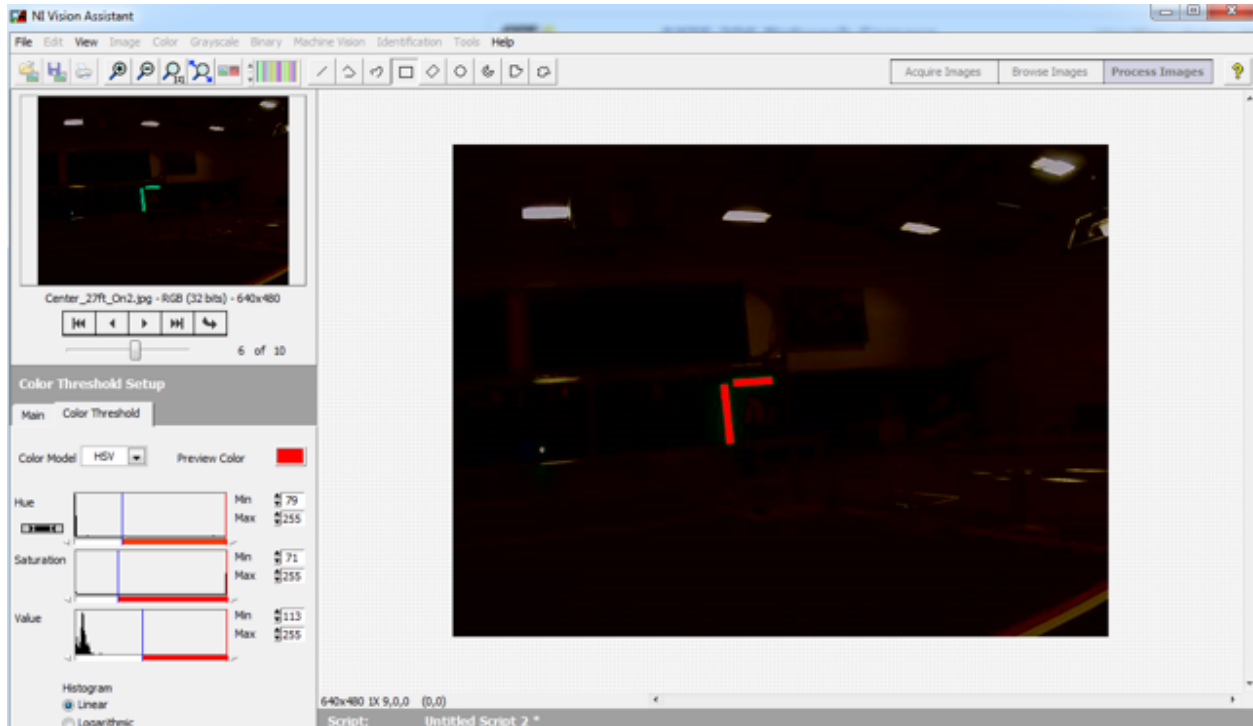
Open the NI Vision Assistant and select the Browse Images option. Select the Open Images icon in the top left of the Toolbar, then locate your images. Repeat as necessary to load all desired images.

Color Threshold



Click Process Images in the top right, then select the color tab on the bottom right and click the Color Threshold icon.

HSV Calibration



Change the Color Model dropdown to HSV. Next tune the window on each of the three values to cover as much of the target as possible while filtering everything else. If using a green light, you may want to use the values in the sample code as a starting point. If you have multiple images you can use the controls in the top left to cycle through them. Use the center two arrow controls or the slider to change the preview image in the top left window, then click the right-most arrow to make it the active image. When you are happy with the values you have selected, note down the ranges for the Hue, Saturation and Value. You will need to enter these into the appropriate place in the vision code. Click OK to finish adding the step to the script.

You may wish to take some new sample images using the time for camera calibration at your event to verify or tweak your ranges slightly based on the venue lighting conditions.

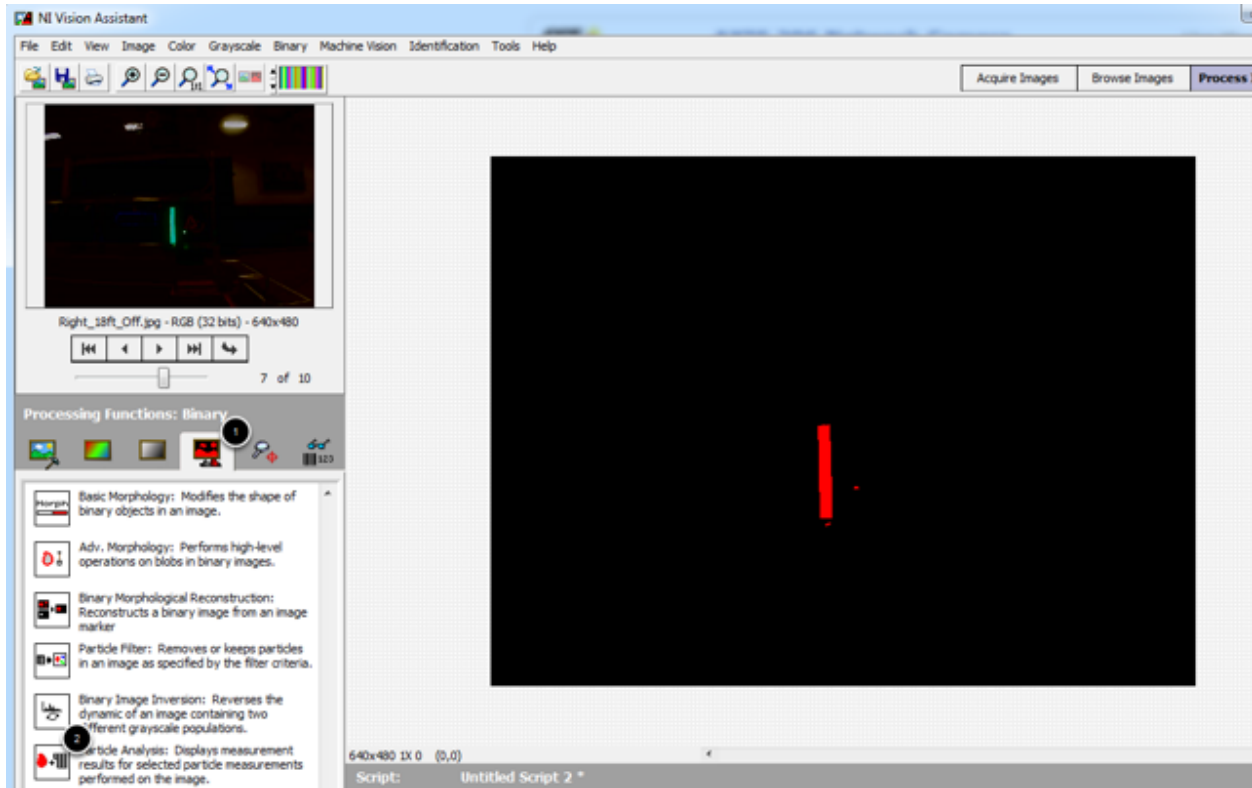
View Angle/Distance Calibration

While a theoretical view angle for each camera model can be found in the datasheet, empirical testing has found that these numbers may be a bit off even for the horizontal view angle. Given that this year's code uses the vertical field-of-view it is best to perform your own calibration for your camera (though empirical values for each camera type are included in the code as a reference). To do this set up an equation where the view angle, θ , is the only unknown. To do this, utilize a target of known size at a known distance, leaving the view angle as the only unknown. Let's take our equation from the previous article, $d = T_{\text{ft}} \cdot \frac{FOV_{\text{pixel}}}{T_{\text{pixel}} \tan \theta}$, and re-arrange it to solve for θ :

$$\tan \theta = T_{\text{ft}} \cdot \frac{FOV_{\text{pixel}}}{T_{\text{pixel}} \cdot d}$$

$$\theta = \arctan \left(T_{ft} \cdot \frac{FOV_{\text{pixel}}}{T_{\text{pixel}} \cdot d} \right)$$

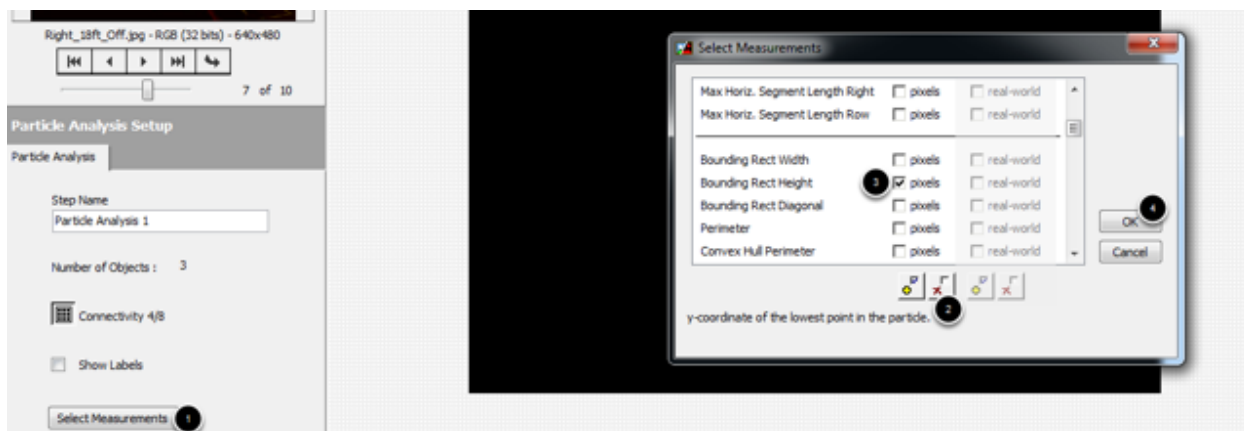
Taking Measurements



One way to take the required measurements is to use the same images of the retro-reflective tape that were used for the color calibration above. We can use Vision Assistant to provide the height of the detected blob in pixels. By measuring the real-world distance between the camera and the target, we now have all of the variables to solve our equation for the view angle.

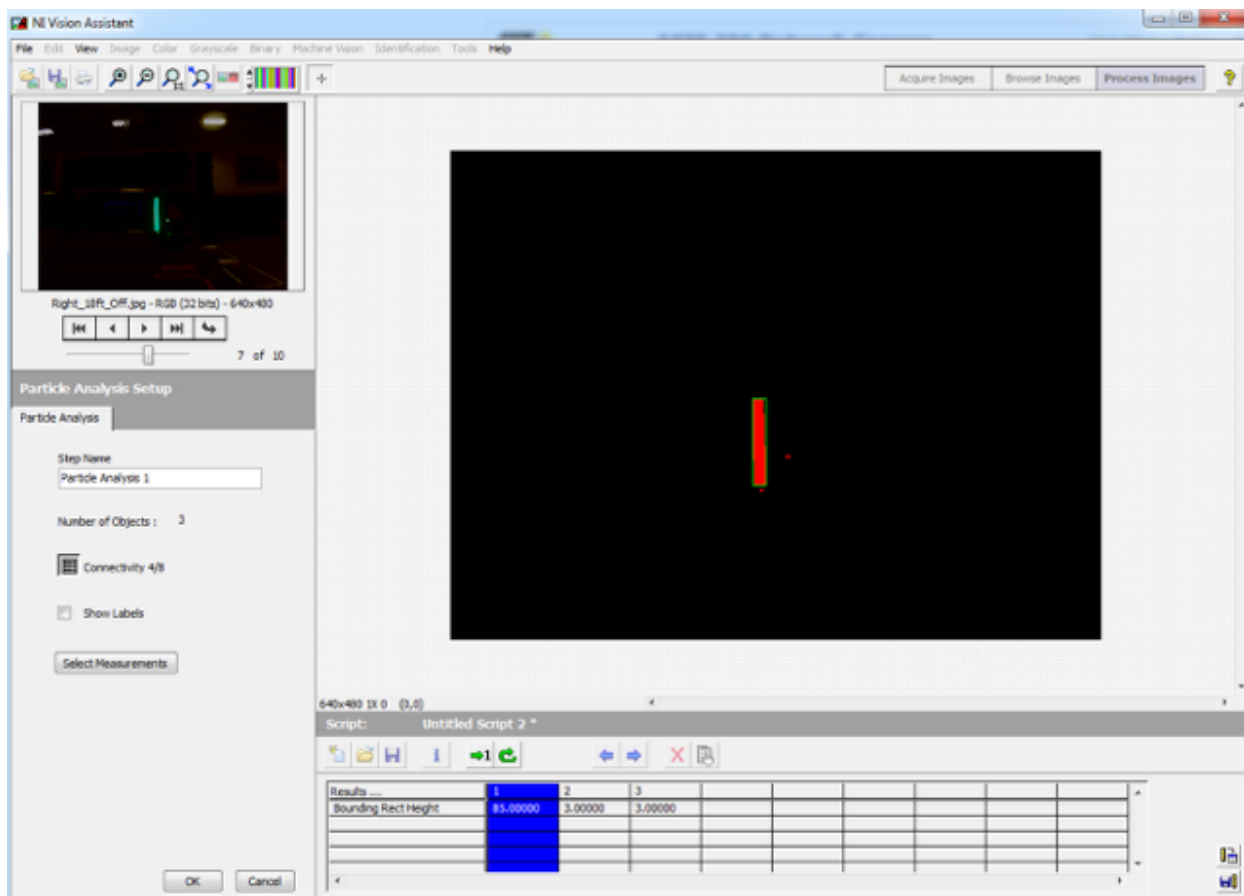
To measure the particles in the image, click the Binary tab, then click the Particle Analysis icon.

Selecting Measurements



Click on the Select Measurements button. In this case, we are only interested in the bounding box height. Click on the button with the X to deselect all measurements, then locate the Bounding Rect Height measurement and check the box. Click OK to save.

Measuring the Particle



The measurements for each particle will now be displayed in the window at the bottom of the

screen. If your image has multiple particles, you can click in each box to have Vision Assistant highlight the particle so you can make sure you have the right one. This article will show the calculation using a single image, but you may wish to perform the calculation on multiple images from multiple distances and use a technique such as averaging or least squares fit to determine the appropriate value for the View angle. You can use the same arrow controls described in the color section above to change the active image.

Calculation

As seen in the previous step, the particle representing the 32in tall vertical target in this example measured 85 pixels tall in a 640x480 image. The image shown was taken from (very roughly) 18 ft. away. Plugging these numbers into the equation from above....

$$\theta = \arctan \left(2.66 \cdot \frac{480}{2 \cdot 85 \cdot 18} \right) = 22.65^\circ$$

Depending on what you use to calculate the arctangent, your answer may be in radians, make sure to convert back to degrees if entering directly into the sample code as the view angle.

Note: The code uses View Angle and we just calculated θ . Make sure to multiply θ by 2 if replacing the constants in the code. Multiplying our result by 2 yields 45.3° . This image is from a M1013 camera, so our value is a bit off from the previously measured 29.1 but given that the 18ft. was a very rough measurement this shows that we are in the ballpark and likely performed the calculation correctly.

28.5.5 Using the Axis Camera at Single Network Events

The convention for using the Axis camera uses mDNS with the camera name set to `axis-camera.local`. At home, this works fine as there is only one camera on the network. At official events, this works fine as each team is on their own VLAN and therefore doesn't have visibility to another team's camera. However, at an off-season event using a single network, this will cause an issue where all teams will connect to whichever team's camera "wins" the mDNS resolution and becomes `axis-camera`. The other cameras will see that the name is taken and use an alternative name. This article describes how to modify the Dashboard and/or robot code to use a different mDNS name to separate the camera streams.

Changing the Camera mDNS Name

To change the mDNS name in the camera, follow the instructions in [Configuring an Axis Camera](#), but substitute the new name such as `axis-cameraTEAM` where TEAM is your team number.

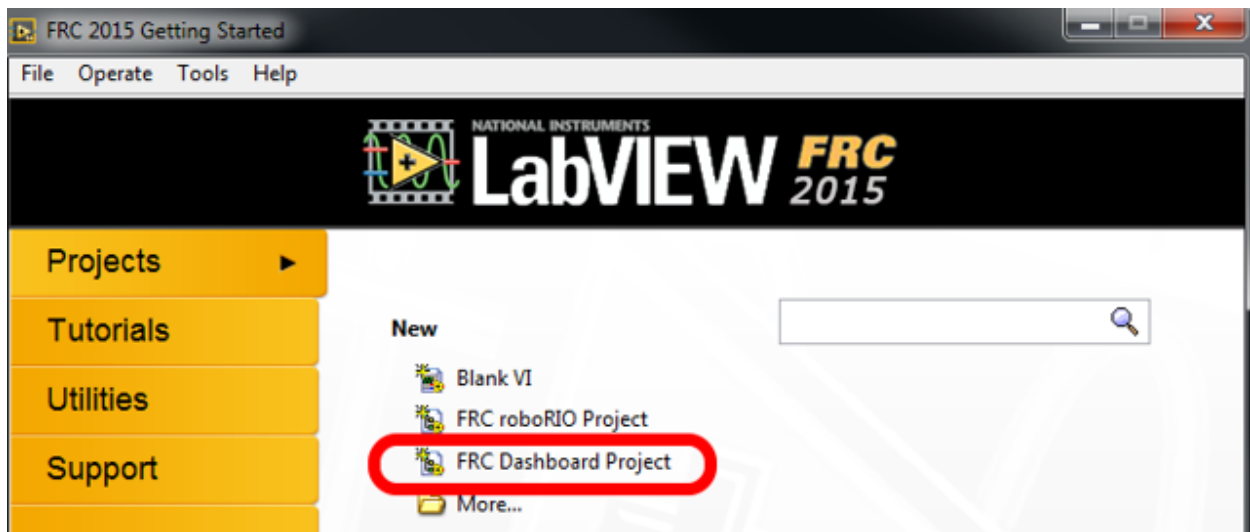
Viewing the Camera on the DS PC - Browser or SmartDashboard

If you are using a web-browser or SmartDashboard (which accepts mDNS names for the Simple Camera Viewer widget), updating to use the new mDNS name is simple. Simply change the URL in the browser or the address in the Simple Camera Viewer widget properties to the new mDNS name and you are all set.

Viewing the Camera on the DS PC - LabVIEW Dashboard

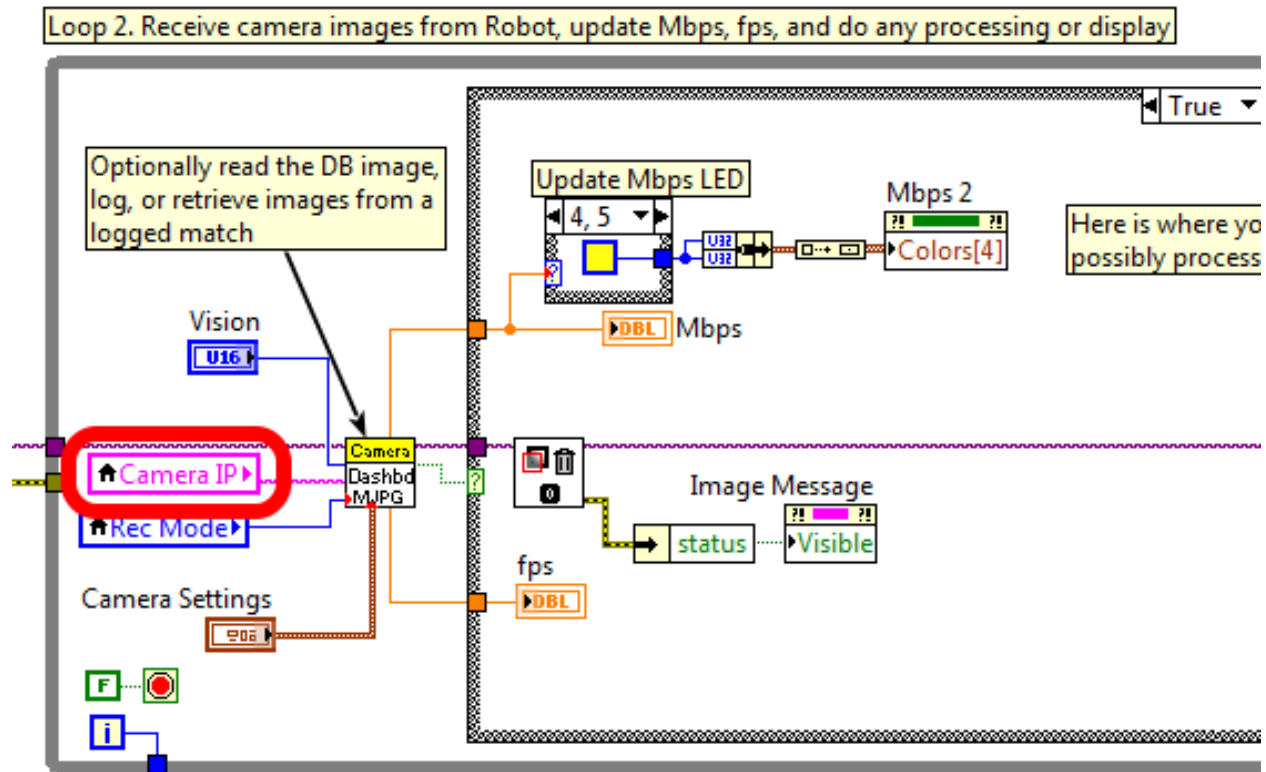
To view the camera stream in the LabVIEW Dashboard, you will need to build a customized version of the Dashboard. Note that this customized version will only work for the Axis camera and will no longer work for a USB camera. Revert to the default Dashboard to use a USB camera.

Creating a Dashboard Project



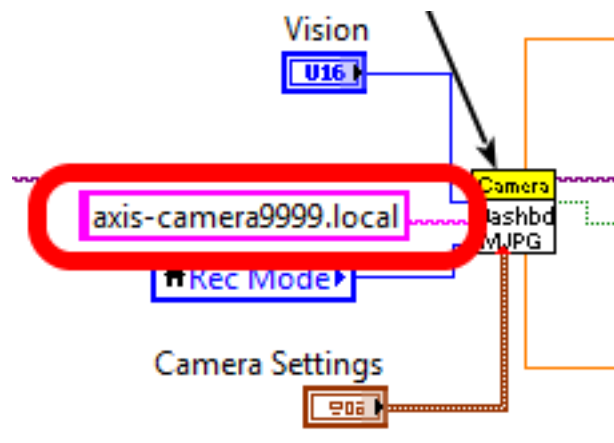
From the LabVIEW Splash screen, select “FRC® Dashboard Project”. Name the project as desired, then click Finish.

Locating Loop 2 - Camera IP



Double click on Dashboard Main.vi in the project explorer to open it and press Ctrl+E to see the block diagram. Scroll down to the loop with the comment that says Loop 2 and locate the "Camera IP" input.

Editing the Camera IP

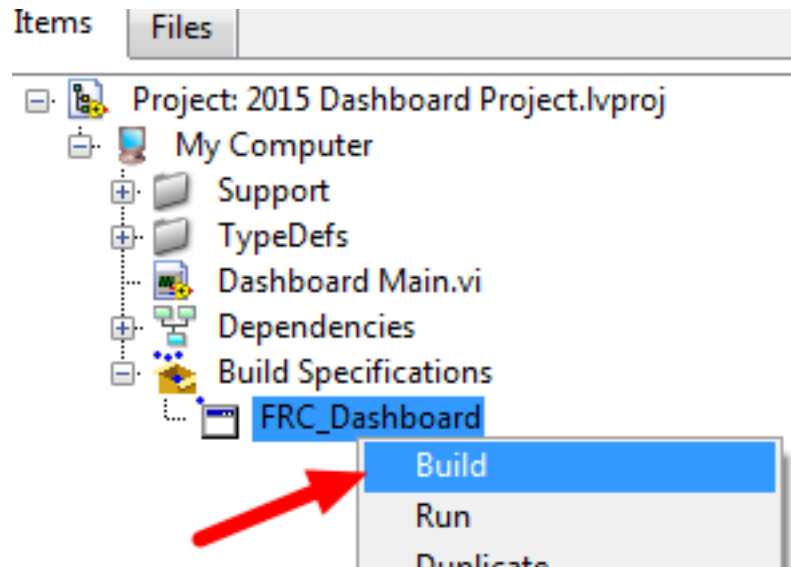


Delete the Camera IP node, right click on the broken wire and click Create Constant (connect the constant to the wire if necessary). In the box, enter the mDNS name of your camera with a .local suffix (e.g. axis-cameraTEAM.local where TEAM is replaced with your team number).

The example above shows a sample name for team 9999. Then click File->Save or Ctrl+S to save the VI.

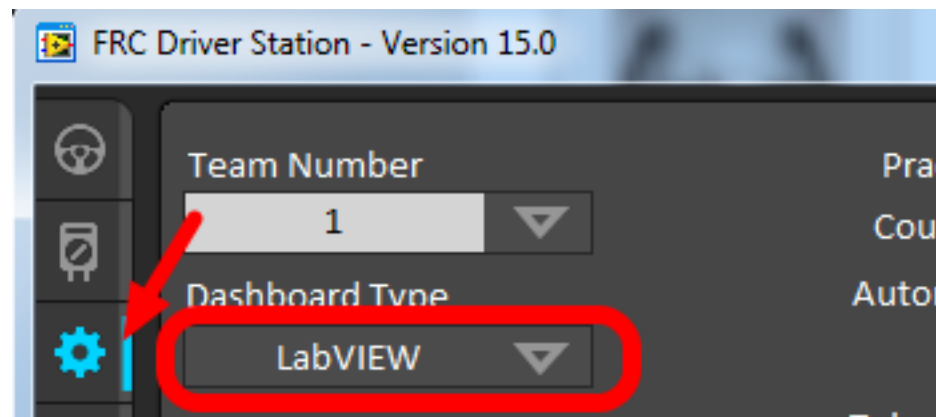
Note: You may also wish to make a minor modification to the Front Panel to verify that you are running the right dashboard later.

Building the Dashboard



To build the new dashboard, expand Build Specifications in the Project Explorer, right click on FRC_Dashboard and select Build.

Setting the Driver Station to Launch the Modified Dashboard



On the Setup tab of the Driver Station, change to dropdown box to LabVIEW to launch your new Dashboard.

Accessing the Camera from Robot Code

If you wish to access the renamed camera from your robot code, you will have to modify it as well. In C++ and Java, just change the String used for the camera host name to match the new name. In LabVIEW follow the step below.

Modifying LabVIEW Robot Code



In the Project Explorer, locate Vision Processing.VI and double click to open it. Then press Ctrl+E to open the Block Diagram. Locate the string axis-camera.local near the left of the image and replace with axis-cameraTEAM.local. Also make sure the constant is set to False to use the Axis camera instead of USB.

Command-Based Programming

This sequence of articles serves as an introduction to and reference for the WPILib command-based framework.

For a collection of example projects using the command-based framework, see [Command-Based Examples](#).

29.1 2020 Command-Based Rewrite: What Changed?

This article provides a summary of changes from the [original command-based framework](#) to the 2020 rewrite. This summary is not necessarily comprehensive - for rigorous documentation, as always, refer to the API docs ([Java](#), [C++](#)).

29.1.1 Package Location

The new command-based framework is located in the `wpilibj2` package for Java, and in the `frc2` namespace for C++. The old command-based framework is still available in the original location. The new framework must be installed using the instructions: [WPILib Command Libraries](#).

29.1.2 Major Architectural Changes

The overall structure of the command-based framework has remained largely the same. However, there are some still a few major architectural changes that users should be aware of:

Commands and Subsystems as Interfaces

Command (Java, C++) and Subsystem (Java, C++) are both now interfaces as opposed to abstract classes, allowing advanced users more potential flexibility. CommandBase and SubsystemBase abstract base classes are still provided for convenience, but are not required. For more information, see [Commands](#) and [Subsystems](#).

Multiple Command Group Classes

The CommandGroup class no longer exists, and has been replaced by a number of narrower classes that can be recursively composed to create more-complicated group structures. For more information see [Command Groups](#).

Inline Command Definitions

Previously, users were required to write a subclass of Command in almost all cases where a command was needed. Many of the new commands are designed to allow inline definition of command functionality, and so can be used without the need for an explicit subclass. For more information, see [Convenience Features](#).

Injection of Command Dependencies

While not an actual change to the coding of the library, the recommended use pattern for the new command-based framework utilizes injection of subsystem dependencies into commands, so that subsystems are not declared as globals. This is a cleaner, more maintainable, and more reusable pattern than the global subsystem pattern promoted previously. For more information, see [Structuring a Command-Based Robot Project](#).

Command Ownership (C++ Only)

The previous command framework required users to use raw pointers for all commands, resulting in nearly-unavoidable memory leaks in all C++ command-based projects, as well as leaving room for common errors such as double-allocating commands within command-groups.

The new command framework offers ownership management for all commands. Default commands and commands bound to buttons are typically owned by the scheduler, and component commands are owned by their encapsulating command groups. As a result, users should generally never heap-allocate a command with new unless there is a very good reason to do so.

Transfer of ownership is done using [perfect forwarding](#), meaning rvalues will be *moved* and lvalues will be *copied* ([rvalue/lvalue explanation](#)).

29.1.3 Changes to the Scheduler

- Scheduler has been renamed to `CommandScheduler` (Java, C++).
- Interruptibility of commands is now the responsibility of the scheduler, not the commands, and can be specified during the call to `schedule`.
- Users can now pass actions to the scheduler which are taken whenever a command is scheduled, interrupted, or ends normally. This is highly useful for cases such as event logging.

29.1.4 Changes to Subsystem

Note: For more information on subsystems, see [Subsystems](#).

- As noted earlier, `Subsystem` is now an interface (Java, C++); the closest equivalent of the old `Subsystem` is the new `SubsystemBase` class. Many of the `Sendable`-related constructor overloads have been removed to reduce clutter; users can call the setters directly from their own constructor, if needed.
- `initDefaultCommand` has been removed; subsystems no longer need to “know about” their default commands, which are instead registered directly with the `CommandScheduler`. The new `setDefaultCommand` method simply wraps the `CommandScheduler` call.
- Subsystems no longer “know about” the commands currently requiring them; this is handled exclusively by the `CommandScheduler`. A convenience wrapper on the `CommandScheduler` method is provided, however.

29.1.5 Changes to Command

Note: For more information on commands, see [Commands](#).

- As noted earlier, `Command` is now an interface (Java, C++); the closest equivalent of the old `Command` is the new `CommandBase` class. Many of the `Sendable`-related constructor overloads have been removed to reduce clutter; users can call the setters directly from their own constructor, if needed.
- Commands no longer handle their own scheduling state; this is now the responsibility of the scheduler.
- The `interrupted()` method has been rolled into the `end()` method, which now takes a parameter specifying whether the command was interrupted (false if it ended normally).
- The `requires()` method has been renamed to `addRequirement()`.
- `void setRunsWhenDisabled(boolean disabled)` has been replaced by an overrideable `runsWhenDisabled()` method. Commands that should run when disabled should override this method to return true.
- `void setInterruptible(boolean interruptible)` has been removed; interruptibility is no longer an innate property of commands, and can be set when the command is scheduled.

- Several “*decorator*” *methods* have been added to allow easy inline modification of commands (e.g. adding a timeout).
- (C++ only) In order to allow the decorators to work with the command ownership model, a *CRTP* is used via the *CommandHelper* *class*. Any user-defined Command subclass Foo *must* extend *CommandHelper*<Foo, Base> where Base is the desired base class.

29.1.6 Changes to PIDSubsystem/PIDCommand

Note: For more information, see *PID Control through PIDSubsystems and PIDCommands*, and *PID Control in WPILib*

- Following the changes to *PIDController*, these classes now run synchronously from the main robot loop.
- The *PIDController* is now injected through the constructor, removing many of the forwarding methods. It can be modified after construction with *getController()*.
- *PIDCommand* is intended largely for inline use, as shown in the *GyroDriveCommands* example (Java, C++).
- If users wish to use *PIDCommand* more “traditionally,” overriding the protected *returnPIDInput()* and *usePIDOutput(double output)* methods has been replaced by modifying the protected *m_measurement* and *m_useOutput* fields. Similarly, rather than calling *setSetpoint*, users can modify the protected *m_setpoint* field.

29.2 What Is “Command-Based” Programming?

WPILib supports a robot programming methodology called “command-based” programming. In general, “command-based” can refer both the general programming paradigm, and to the set of WPILib library resources included to facilitate it.

“Command-based” programming is an example of what is known as a *design pattern*. It is a general way of organizing one’s robot code that is well-suited to a particular problem-space. It is not the only way to write a robot program, but it is a very effective one; command-based robot code tends to be clean, extensible, and (with some tricks) easy to re-use from year to year.

The command-based paradigm is also an example of what is known as *declarative* programming. In declarative programming, the emphasis is placed on *what* the program ought to do, rather than *how* the program ought to do it. Thus, the command-based libraries allow users to define desired robot behaviors while minimizing the amount of iteration-by-iteration robot logic that they must write. For example, in a command-based program, a user can specify that “the robot should perform an action when a button is pressed” (note the use of a *lambda*):

Java

C++

```
aButton.whenPressed(intake::run);
```

```
aButton.WhenPressed([&intake] { intake.Run(); });
```

In contrast, in an ordinary **imperative** program, the user would need to check the button state every iteration, and perform the appropriate action based on the state of the button.

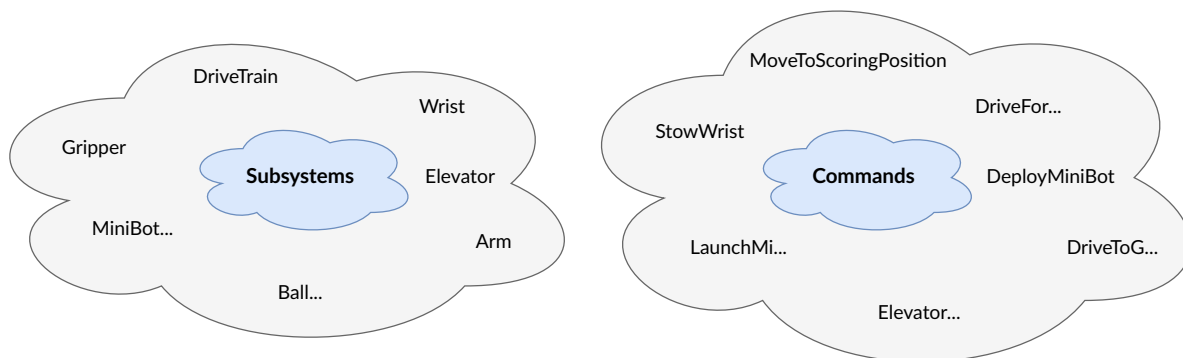
Java

C++

```
if(aButton.get()) {
    if(!pressed) {
        intake.run();
        pressed = true;
    }
} else {
    pressed = false;
}
```

```
if(aButton.Get()) {
    if(!pressed) {
        Intake.Run();
        pressed = true;
    }
} else {
    pressed = false;
}
```

29.2.1 Subsystems and Commands



Viewer does not support full SVG 1.1

The command-based pattern is based around two core abstractions: **commands**, and **subsystems**.

Subsystems are the basic unit of robot organization in the design-based paradigm. Subsystems **encapsulate** lower-level robot hardware (such as motor controllers, sensors, and/or pneumatic actuators), and define the interfaces through which that hardware can be accessed by the rest of the robot code. Subsystems allow users to “hide” the internal complexity of their actual hardware from the rest of their code - this both simplifies the rest of the robot code, and allows changes to the internal details of a subsystem without also changing the rest of the robot code. Subsystems implement the Subsystem interface.

Commands define high-level robot actions or behaviors that utilize the methods defined by the subsystems. A command is a simple **state machine** that is either initializing, executing, ending, or idle. Users write code specifying which action should be taken in each state.

Simple commands can be composed into “command groups” to accomplish more-complicated tasks. Commands, including command groups, implement the Command interface.

29.2.2 How Commands Are Run

Note: For a more detailed explanation, see *The Command Scheduler*.

Commands are run by the `CommandScheduler` (Java, C++), a singleton class that is at the core of the command-based library. The `CommandScheduler` is in charge of polling buttons for new commands to schedule, checking the resources required by those commands to avoid conflicts, executing currently-scheduled commands, and removing commands that have finished or been interrupted. The scheduler’s `run()` method may be called from any place in the user’s code; it is generally recommended to call it from the `robotPeriodic()` method of the `Robot` class, which is run at a default frequency of 50Hz (once every 20ms).

Multiple commands can run concurrently, as long as they do not require the same resources on the robot. Resource management is handled on a per-subsystem basis: commands may specify which subsystems they interact with, and the scheduler will never schedule more than one command requiring a given subsystem at a time. This ensures that, for example, users will not end up with two different pieces of code attempting to set the same motor controller to different output values. If a new command is scheduled that requires a subsystem that is already in use, it will either interrupt the currently-running command that requires that subsystem (if the command has been scheduled as interruptible), or else it will not be scheduled.

Subsystems also can be associated with “default commands” that will be automatically scheduled when no other command is currently using the subsystem. This is useful for continuous “background” actions such as controlling the robot drive, or keeping an arm held at a setpoint.

When a command is scheduled, its `initialize()` method is called once. Its `execute()` method is then called once per call to `CommandScheduler.getInstance().run()`. A command is un-scheduled and has its `end(boolean interrupted)` method called when either its `isFinished()` method returns true, or else it is interrupted (either by another command with which it shares a required subsystem, or by being canceled).

29.2.3 Command Groups

It is often desirable to build complex commands from simple pieces. This is achievable by **composing** commands into “command groups.” A *command group* is a command that contains multiple commands within it, which run either in parallel or in sequence. The command-based library provides several types of command groups for teams to use, and users are encouraged to write their own, if desired. As command groups themselves implement the Command interface, they are **recursively composable** - one can include command groups *within* other command groups. This provides an extremely powerful way of building complex robot actions with a simple library.

29.3 Creating a Robot Project

Creating a project is detailed in [Creating a Robot Program](#). Select “Template” then your programming language then “New Command Robot” to create a basic Command-Based Robot program.

When you create a New Command Robot project, the new command based vendor library is automatically imported. If you imported a 2019 project or created a different type of project, the old command library is imported, and it is necessary to import the new command based vendor library per [3rd Party Libraries](#) and remove the old command library.

29.4 Subsystems

Subsystems are the basic unit of robot organization in the command-based paradigm. A subsystem is an abstraction for a collection of robot hardware that *operates together as a unit*. Subsystems [encapsulate](#) this hardware, “hiding” it from the rest of the robot code (e.g. commands) and restricting access to it except through the subsystem’s public methods. Restricting the access in this way provides a single convenient place for code that might otherwise be duplicated in multiple places (such as scaling motor outputs or checking limit switches) if the subsystem internals were exposed. It also allows changes to the specific details of how the subsystem works (the “implementation”) to be isolated from the rest of robot code, making it far easier to make substantial changes if/when the design constraints change.

Subsystems also serve as the backbone of the CommandScheduler’s resource management system. Commands may declare resource requirements by specifying which subsystems they interact with; the scheduler will never concurrently schedule more than one command that requires a given subsystem. An attempt to schedule a command that requires a subsystem that is already-in-use will either interrupt the currently-running command (if the command has been scheduled as interruptible), or else be ignored.

Subsystems can be associated with “default commands” that will be automatically scheduled when no other command is currently using the subsystem. This is useful for continuous “background” actions such as controlling the robot drive, or keeping an arm held at a setpoint. Similar functionality can be achieved in the subsystem’s `periodic()` method, which is run once per run of the scheduler; teams should try to be consistent within their codebase about which functionality is achieved through either of these methods. There is also a `simulationPeriodic()` method that is similar to `periodic()` except that it is only run during [Simulation](#) and can be used to update the state of the robot. Subsystems are represented in the command-based library by the Subsystem interface ([Java](#), [C++](#)).

29.4.1 Creating a Subsystem

The recommended method to create a subsystem for most users is to subclass the abstract `SubsystemBase` class ([Java](#), [C++](#)), as seen in the command-based template ([Java](#), [C++](#)):

Java

C++

```
10 import edu.wpi.first.wpilibj2.command.SubsystemBase;
11
12 public class ExampleSubsystem extends SubsystemBase {
```

(continues on next page)

(continued from previous page)

```

13  /**
14   * Creates a new ExampleSubsystem.
15   */
16  public ExampleSubsystem() {
17
18  }
19
20  @Override
21  public void periodic() {
22      // This method will be called once per scheduler run
23  }
24  }

```

```

8  #pragma once
9
10 #include <frc2/command/SubsystemBase.h>
11
12 class ExampleSubsystem : public frc2::SubsystemBase {
13 public:
14     ExampleSubsystem();
15
16     /**
17      * Will be called periodically whenever the CommandScheduler runs.
18      */
19     void Periodic() override;
20
21 private:
22     // Components (e.g. motor controllers and sensors) should generally be
23     // declared private and exposed only through public methods.
24 };

```

This class contains a few convenience features on top of the basic Subsystem interface: it automatically calls the `register()` method in its constructor to register the subsystem with the scheduler (this is necessary for the `periodic()` method to be called when the scheduler runs), and also implements the `Sendable` interface so that it can be sent to the dashboard to display/log relevant status information.

Advanced users seeking more flexibility may simply create a class that implements the Subsystem interface.

29.4.2 Simple Subsystem Example

What might a functional subsystem look like in practice? Below is a simple pneumatically-actuated hatch mechanism from the HatchBot example project (Java, C++):

Java

C++ (Header)

C++ (Source)

```

8  package edu.wpi.first.wpilibj.examples.hatchbottraditional.subsystems;
9
10 import edu.wpi.first.wpilibj.DoubleSolenoid;
11 import edu.wpi.first.wpilibj2.command.SubsystemBase;

```

(continues on next page)

(continued from previous page)

```

12 import edu.wpi.first.wpilibj.examples.hatchbottraditional.Constants.HatchConstants;
13
14 import static edu.wpi.first.wpilibj.DoubleSolenoid.Value.kForward;
15 import static edu.wpi.first.wpilibj.DoubleSolenoid.Value.kReverse;
16
17 /**
18  * A hatch mechanism actuated by a single {@link DoubleSolenoid}.
19  */
20 public class HatchSubsystem extends SubsystemBase {
21     private final DoubleSolenoid m_hatchSolenoid =
22         new DoubleSolenoid(HatchConstants.kHatchSolenoidModule, HatchConstants.
23             ↪kHatchSolenoidPorts[0],
24                 HatchConstants.kHatchSolenoidPorts[1]);
25
26     /**
27      * Grabs the hatch.
28      */
29     public void grabHatch() {
30         m_hatchSolenoid.set(kForward);
31     }
32
33     /**
34      * Releases the hatch.
35      */
36     public void releaseHatch() {
37         m_hatchSolenoid.set(kReverse);
38     }
39 }

```

```

8 #pragma once
9
10 #include <frc/DoubleSolenoid.h>
11 #include <frc2/command/SubsystemBase.h>
12
13 #include "Constants.h"
14
15 class HatchSubsystem : public frc2::SubsystemBase {
16 public:
17     HatchSubsystem();
18
19     // Subsystem methods go here.
20
21     /**
22      * Grabs the hatch.
23      */
24     void GrabHatch();
25
26     /**
27      * Releases the hatch.
28      */
29     void ReleaseHatch();
30
31 private:
32     // Components (e.g. motor controllers and sensors) should generally be
33     // declared private and exposed only through public methods.

```

(continues on next page)

(continued from previous page)

```

34   frc::DoubleSolenoid m_hatchSolenoid;
35   };

8   #include "subsystems/HatchSubsystem.h"
9
10  using namespace HatchConstants;
11
12  HatchSubsystem::HatchSubsystem()
13      : m_hatchSolenoid{kHatchSolenoidPorts[0], kHatchSolenoidPorts[1]} {}
14
15  void HatchSubsystem::GrabHatch() {
16      m_hatchSolenoid.Set(frc::DoubleSolenoid::kForward);
17  }
18
19  void HatchSubsystem::ReleaseHatch() {
20      m_hatchSolenoid.Set(frc::DoubleSolenoid::kReverse);
21  }

```

Notice that the subsystem hides the presence of the `DoubleSolenoid` from outside code (it is declared private), and instead publicly exposes two higher-level, descriptive robot actions: `grabHatch()` and `releaseHatch()`. It is extremely important that “implementation details” such as the double solenoid be “hidden” in this manner; this ensures that code outside the subsystem will never cause the solenoid to be in an unexpected state. It also allows the user to change the implementation (for instance, a motor could be used instead of a pneumatic) without any of the code outside of the subsystem having to change with it.

29.4.3 Setting Default Commands

Note: In the C++ command-based library, the `CommandScheduler` owns the default command objects - accordingly, the object passed to the `setDefaultCommand()` method will be either moved or copied, depending on whether it is an rvalue or an lvalue ([rvalue/lvalue explanation](#)). The examples here ensure that move semantics are used by casting to an rvalue with `std::move()`.

“Default commands” are commands that run automatically whenever a subsystem is not being used by another command.

Setting a default command for a subsystem is very easy; one simply calls `CommandScheduler.getInstance().setDefaultCommand()`, or, more simply, the `setDefaultCommand()` method of the `Subsystem` interface:

Java

C++

```
CommandScheduler.getInstance().setDefaultCommand(exampleSubsystem, exampleCommand);
```

```
CommandScheduler.GetInstance().SetDefaultCommand(exampleSubsystem,
↳ std::move(exampleCommand));
```

Java

C++

```
exampleSubsystem.setDefaultCommand(exampleCommand);
```

```
exampleSubsystem.SetDefaultCommand(std::move(exampleCommand));
```

29.5 Commands

Commands are simple state machines that perform high-level robot functions using the methods defined by subsystems. Commands can be either idle, in which they do nothing, or scheduled, in which the scheduler will execute a specific set of the command's code depending on the state of the command. The CommandScheduler recognizes scheduled commands as being in one of three states: initializing, executing, or ending. Commands specify what is done in each of these states through the `initialize()`, `execute()` and `end()` methods. Commands are represented in the command-based library by the Command interface (Java, C++).

29.5.1 Creating Commands

Note: In the C++ API, a `CRTP` is used to allow certain Command methods to work with the object ownership model. Users should *always* extend the `CommandHelper` class when defining their own command classes, as is shown below.

Similarly to subsystems, the recommended method for most users to create a command is to subclass the abstract `CommandBase` class (Java, C++), as seen in the command-based template (Java, C++):

Java

C++

```

10 import edu.wpi.first.wpilibj.templates.commandbased.subsystems.ExampleSubsystem;
11 import edu.wpi.first.wpilibj2.command.CommandBase;
12
13 /**
14  * An example command that uses an example subsystem.
15  */
16 public class ExampleCommand extends CommandBase {
17     @SuppressWarnings({"PMD.UnusedPrivateField", "PMD.SingularField"})
18     private final ExampleSubsystem m_subsystem;
19
20     /**
21      * Creates a new ExampleCommand.
22      *
23      * @param subsystem The subsystem used by this command.
24      */
25     public ExampleCommand(ExampleSubsystem subsystem) {
26         m_subsystem = subsystem;
27         // Use addRequirements() here to declare subsystem dependencies.
28         addRequirements(subsystem);
29     }

```

```

8  #pragma once
9
10 #include <frc2/command/CommandBase.h>
11 #include <frc2/command/CommandHelper.h>
12
13 #include "subsystems/ExampleSubsystem.h"
14
15 /**
16  * An example command that uses an example subsystem.
17  *
18  * <p>Note that this extends CommandHelper, rather extending CommandBase
19  * directly; this is crucially important, or else the decorator functions in
20  * Command will *not* work!
21  */
22 class ExampleCommand
23 : public frc2::CommandHelper<frc2::CommandBase, ExampleCommand> {
24 public:
25     /**
26      * Creates a new ExampleCommand.
27      *
28      * @param subsystem The subsystem used by this command.
29      */
30     explicit ExampleCommand(ExampleSubsystem* subsystem);
31
32 private:
33     ExampleSubsystem* m_subsystem;
34 };

```

As before, this contains several convenience features. It automatically overrides the `getRequirements()` method for users, returning a list of requirements that is empty by default, but can be added to with the `addRequirements()` method. It also implements the `Sendable` interface, and so can be sent to the dashboard - this provides a handy way for scheduling commands for testing (via a button on the dashboard) without needing to bind them to buttons on a controller.

Also as before, advanced users seeking more flexibility are free to simply create their own class implementing the `Command` interface.

29.5.2 The Structure of a Command

While subsystems are fairly freeform, and may generally look like whatever the user wishes them to, commands are quite a bit more constrained. Command code must specify what the command will do in each of its possible states. This is done by overriding the `initialize()`, `execute()`, and `end()` methods. Additionally, a command must be able to tell the scheduler when (if ever) it has finished execution - this is done by overriding the `isFinished()` method. All of these methods are defaulted to reduce clutter in user code: `initialize()`, `execute()`, and `end()` are defaulted to simply do nothing, while `isFinished()` is defaulted to return `false` (resulting in a command that never ends).

Initialization

The `initialize()` method (Java, C++) is run exactly once per time a command is scheduled, as part of the scheduler's `schedule()` method. The scheduler's `run()` method does not need to be called for the `initialize()` method to run. The initialize block should be used to place the command in a known starting state for execution. It is also useful for performing tasks that only need to be performed once per time scheduled, such as setting motors to run at a constant speed or setting the state of a solenoid actuator.

Execution

The `execute()` method (Java, C++) is called repeatedly while the command is scheduled, whenever the scheduler's `run()` method is called (this is generally done in the main robot periodic method, which runs every 20ms by default). The execute block should be used for any task that needs to be done continually while the command is scheduled, such as updating motor outputs to match joystick inputs, or using the output of a control loop.

Ending

The `end()` method (Java, C++) is called once when the command ends, whether it finishes normally (i.e. `isFinished()` returned true) or it was interrupted (either by another command or by being explicitly canceled). The method argument specifies the manner in which the command ended; users can use this to differentiate the behavior of their command end accordingly. The end block should be used to “wrap up” command state in a neat way, such as setting motors back to zero or reverting a solenoid actuator to a “default” state.

Specifying end conditions

The `isFinished()` method (Java, C++) is called repeatedly while the command is scheduled, whenever the scheduler's `run()` method is called. As soon as it returns true, the command's `end()` method is called and it is un-scheduled. The `isFinished()` method is called *after* the `execute()` method, so the command *will* execute once on the same iteration that it is un-scheduled.

29.5.3 Simple Command Example

What might a functional command look like in practice? As before, below is a simple command from the HatchBot example project (Java, C++) that uses the `HatchSubsystem` introduced in the previous section:

Java

C++ (Header)

C++ (Source)

```

8 package edu.wpi.first.wpilibj.examples.hatchbottraditional.commands;
9
10 import edu.wpi.first.wpilibj2.command.CommandBase;
11
12 import edu.wpi.first.wpilibj.examples.hatchbottraditional.subsystems.HatchSubsystem;
```

(continues on next page)

(continued from previous page)

```

13  /**
14  * A simple command that grabs a hatch with the {@link HatchSubsystem}. Written
15  * explicitly for
16  * pedagogical purposes. Actual code should inline a command this simple with {@link
17  * edu.wpi.first.wpilibj2.command.InstantCommand}.
18  */
19  public class GrabHatch extends CommandBase {
20      // The subsystem the command runs on
21      private final HatchSubsystem m_hatchSubsystem;
22
23      public GrabHatch(HatchSubsystem subsystem) {
24          m_hatchSubsystem = subsystem;
25          addRequirements(m_hatchSubsystem);
26      }
27
28      @Override
29      public void initialize() {
30          m_hatchSubsystem.grabHatch();
31      }
32
33      @Override
34      public boolean isFinished() {
35          return true;
36      }
37  }

```

```

8  #pragma once
9
10 #include <frc2/command/CommandBase.h>
11 #include <frc2/command/CommandHelper.h>
12
13 #include "subsystems/HatchSubsystem.h"
14
15 /**
16  * A simple command that grabs a hatch with the HatchSubsystem. Written
17  * explicitly for pedagogical purposes. Actual code should inline a command
18  * this simple with InstantCommand.
19  *
20  * @see InstantCommand
21  */
22 class GrabHatch : public frc2::CommandHelper<frc2::CommandBase, GrabHatch> {
23 public:
24     explicit GrabHatch(HatchSubsystem* subsystem);
25
26     void Initialize() override;
27
28     bool IsFinished() override;
29
30 private:
31     HatchSubsystem* m_hatch;
32 };

```

```

8  #include "commands/GrabHatch.h"
9
10 GrabHatch::GrabHatch(HatchSubsystem* subsystem) : m_hatch(subsystem) {

```

(continues on next page)

(continued from previous page)

```

11     AddRequirements({ subsystem });
12 }
13
14 void GrabHatch::Initialize() { m_hatch->GrabHatch(); }
15
16 bool GrabHatch::IsFinished() { return true; }

```

Notice that the hatch subsystem used by the command is passed into the command through the command's constructor. This is a pattern called [dependency injection](#), and allows users to avoid declaring their subsystems as global variables. This is widely accepted as a best-practice - the reasoning behind this is discussed in a [later section](#).

Notice also that the above command calls the subsystem method once from initialize, and then immediately ends (as `isFinished()` simply returns true). This is typical for commands that toggle the states of subsystems, and in fact the command-based library includes code to make [commands like this](#) even more succinctly.

What about a more complicated case? Below is a drive command, from the same example project:

Java

C++ (Header)

C++ (Source)

```

8  package edu.wpi.first.wpilibj.examples.hatchbottraditional.commands;
9
10 import java.util.function.DoubleSupplier;
11
12 import edu.wpi.first.wpilibj2.command.CommandBase;
13
14 import edu.wpi.first.wpilibj.examples.hatchbottraditional.subsystems.DriveSubsystem;
15
16 /**
17  * A command to drive the robot with joystick input (passed in as {@link
18  * ↳DoubleSupplier}s). Written
19  * explicitly for pedagogical purposes - actual code should inline a command this
20  * ↳simple with {@link
21  * ↳edu.wpi.first.wpilibj2.command.RunCommand}.
22  */
23 public class DefaultDrive extends CommandBase {
24     private final DriveSubsystem m_drive;
25     private final DoubleSupplier m_forward;
26     private final DoubleSupplier m_rotation;
27
28     /**
29      * Creates a new DefaultDrive.
30      *
31      * @param subsystem The drive subsystem this command will run on.
32      * @param forward The control input for driving forwards/backwards
33      * @param rotation The control input for turning
34      */
35     public DefaultDrive(DriveSubsystem subsystem, DoubleSupplier forward,
36     ↳DoubleSupplier rotation) {
37         m_drive = subsystem;
38         m_forward = forward;
39         m_rotation = rotation;

```

(continues on next page)

(continued from previous page)

```

37     addRequirements(m_drive);
38 }
39
40 @Override
41 public void execute() {
42     m_drive.arcadeDrive(m_forward.getAsDouble(), m_rotation.getAsDouble());
43 }
44 }

```

```

8  #pragma once
9
10 #include <frc2/command/CommandBase.h>
11 #include <frc2/command/CommandHelper.h>
12
13 #include "subsystems/DriveSubsystem.h"
14
15 /**
16  * A command to drive the robot with joystick input passed in through lambdas.
17  * Written explicitly for pedagogical purposes - actual code should inline a
18  * command this simple with RunCommand.
19  *
20  * @see RunCommand
21  */
22 class DefaultDrive
23     : public frc2::CommandHelper<frc2::CommandBase, DefaultDrive> {
24 public:
25     /**
26      * Creates a new DefaultDrive.
27      *
28      * @param subsystem The drive subsystem this command wil run on.
29      * @param forward The control input for driving forwards/backwards
30      * @param rotation The control input for turning
31      */
32     DefaultDrive(DriveSubsystem* subsystem, std::function<double()> forward,
33                 std::function<double()> rotation);
34
35     void Execute() override;
36
37 private:
38     DriveSubsystem* m_drive;
39     std::function<double()> m_forward;
40     std::function<double()> m_rotation;
41 };

```

```

8  #include "commands/DefaultDrive.h"
9
10 DefaultDrive::DefaultDrive(DriveSubsystem* subsystem,
11                             std::function<double()> forward,
12                             std::function<double()> rotation)
13     : m_drive{subsystem}, m_forward{forward}, m_rotation{rotation} {
14     AddRequirements({subsystem});
15 }
16
17 void DefaultDrive::Execute() {
18     m_drive->ArcadeDrive(m_forward(), m_rotation());
19 }

```

Notice that this command does not override `isFinished()`, and thus will never end; this is the norm for commands that are intended to be used as default commands (and, as can be guessed, the library includes tools to make *this kind of command* easier to write, too!).

29.6 Command Groups

Individual commands are capable of accomplishing a large variety of robot tasks, but the simple three-state format can quickly become cumbersome when more advanced functionality requiring extended sequences of robot tasks or coordination of multiple robot subsystems is required. In order to accomplish this, users are encouraged to use the powerful command group functionality included in the command-based library.

As the name suggests, command groups are combinations of multiple commands. The act of combining multiple objects (such as commands) into a bigger object is known as *composition*. Command groups *compose* multiple commands into a *composite* command. This allows code to be kept much cleaner and simpler, as the individual *component* commands may be written independently of the code that combines them, greatly reducing the amount of complexity at any given step of the process.

Most importantly, however, command groups *are themselves commands* - they implement the Command interface. This allows command groups to be *recursively composed* - that is, a command group may contain *other command groups* as components.

29.6.1 Types of Command Groups

Note: In the C++ command-based library, command groups *own* their component commands. This means that commands passed to command groups will be either moved or copied depending on whether they are rvalues or lvalues (*rvalue/lvalue explanation*). Due to certain technical concerns, command groups themselves are not copyable, and so recursive composition *must* use move semantics.

The command-based library supports four basic types of command groups: `SequentialCommandGroup`, `ParallelCommandGroup`, `ParallelRaceGroup`, and `ParallelDeadlineGroup`. Each of these command groups combines multiple commands into a composite command - however, they do so in different ways:

SequentialCommandGroup

A `SequentialCommandGroup` (Java, C++) runs a list of commands in sequence - the first command will be executed, then the second, then the third, and so on until the list finishes. The sequential group finishes after the last command in the sequence finishes. It is therefore usually important to ensure that each command in the sequence does actually finish (if a given command does not finish, the next command will never start!).

ParallelCommandGroup

A `ParallelCommandGroup` (Java, C++) runs a set of commands concurrently - all commands will execute at the same time. The parallel group will end when all commands have finished.

ParallelRaceGroup

A `ParallelRaceGroup` (Java, C++) is much like a `ParallelCommandGroup`, in that it runs a set of commands concurrently. However, the race group ends as soon as any command in the group ends - all other commands are interrupted at that point.

ParallelDeadlineGroup

A `ParallelDeadlineGroup` (Java, C++) also runs a set of commands concurrently. However, the deadline group ends when a *specific* command (the “deadline”) ends, interrupting all other commands in the group that are still running at that point.

29.6.2 Creating Command Groups

Users have several options for creating command groups. One way - similar to the previous implementation of the command-based library - is to subclass one of the command group classes. Consider the following from the Hatch Bot example project (Java, C++):

Java

C++ (Header)

C++ (Source)

```
8 package edu.wpi.first.wpilibj.examples.hatchbottraditional.commands;
9
10 import edu.wpi.first.wpilibj2.command.SequentialCommandGroup;
11
12 import edu.wpi.first.wpilibj.examples.hatchbottraditional.Constants.AutoConstants;
13 import edu.wpi.first.wpilibj.examples.hatchbottraditional.subsystems.DriveSubsystem;
14 import edu.wpi.first.wpilibj.examples.hatchbottraditional.subsystems.HatchSubsystem;
15
16 /**
17  * A complex auto command that drives forward, releases a hatch, and then drives
18  * backward.
19  */
20 public class ComplexAuto extends SequentialCommandGroup {
21     /**
22      * Creates a new ComplexAuto.
23      *
24      * @param drive The drive subsystem this command will run on
25      * @param hatch The hatch subsystem this command will run on
26      */
27     public ComplexAuto(DriveSubsystem drive, HatchSubsystem hatch) {
28         addCommands(
29             // Drive forward the specified distance
30             new DriveDistance(AutoConstants.kAutoDriveDistanceInches, AutoConstants.
31                 kAutoDriveSpeed,
```

(continues on next page)

(continued from previous page)

```

30         drive),
31
32         // Release the hatch
33         new ReleaseHatch(hatch),
34
35         // Drive backward the specified distance
36         new DriveDistance(AutoConstants.kAutoBackupDistanceInches, -AutoConstants.
↪ kAutoDriveSpeed,
37         drive));
38     }
39
40 }

```

```

8  #pragma once
9
10 #include <frc2/command/CommandHelper.h>
11 #include <frc2/command/SequentialCommandGroup.h>
12
13 #include "Constants.h"
14 #include "commands/DriveDistance.h"
15 #include "commands/ReleaseHatch.h"
16
17 /**
18  * A complex auto command that drives forward, releases a hatch, and then drives
19  * backward.
20  */
21 class ComplexAuto
22     : public frc2::CommandHelper<frc2::SequentialCommandGroup, ComplexAuto> {
23 public:
24     /**
25      * Creates a new ComplexAuto.
26      *
27      * @param drive The drive subsystem this command will run on
28      * @param hatch The hatch subsystem this command will run on
29      */
30     ComplexAuto(DriveSubsystem* drive, HatchSubsystem* hatch);
31 };

```

```

8  #include "commands/ComplexAuto.h"
9
10 using namespace AutoConstants;
11
12 ComplexAuto::ComplexAuto(DriveSubsystem* drive, HatchSubsystem* hatch) {
13     AddCommands(
14         // Drive forward the specified distance
15         DriveDistance(kAutoDriveDistanceInches, kAutoDriveSpeed, drive),
16         // Release the hatch
17         ReleaseHatch(hatch),
18         // Drive backward the specified distance
19         DriveDistance(kAutoBackupDistanceInches, -kAutoDriveSpeed, drive));
20 }

```

The `addCommands()` method adds commands to the group, and is present in all four types of command group.

29.6.3 Inline Command Groups

Note: Due to the verbosity of Java's new syntax, the Java `CommandGroupBase` object offers a factory method for each of the four command-group types: `sequence`, `parallel`, `race`, and `deadline`.

Command groups can be used without subclassing at all: one can simply pass in the desired commands through the constructor:

Java

C++

```
new SequentialCommandGroup(new FooCommand(), new BarCommand());
```

```
frc2::SequentialCommandGroup{FooCommand(), BarCommand()};
```

This is called an *inline* command definition, and is very handy for circumstances where command groups are not likely to be reused, and writing an entire class for them would be wasteful.

29.6.4 Recursive Composition of Command Groups

As mentioned earlier, command groups are *recursively composable* - since command groups are themselves commands, they may be included as components of other command groups. This is an extremely powerful feature of command groups, and allows users to build very complex robot actions from simple pieces. For example, consider the following code:

Java

C++

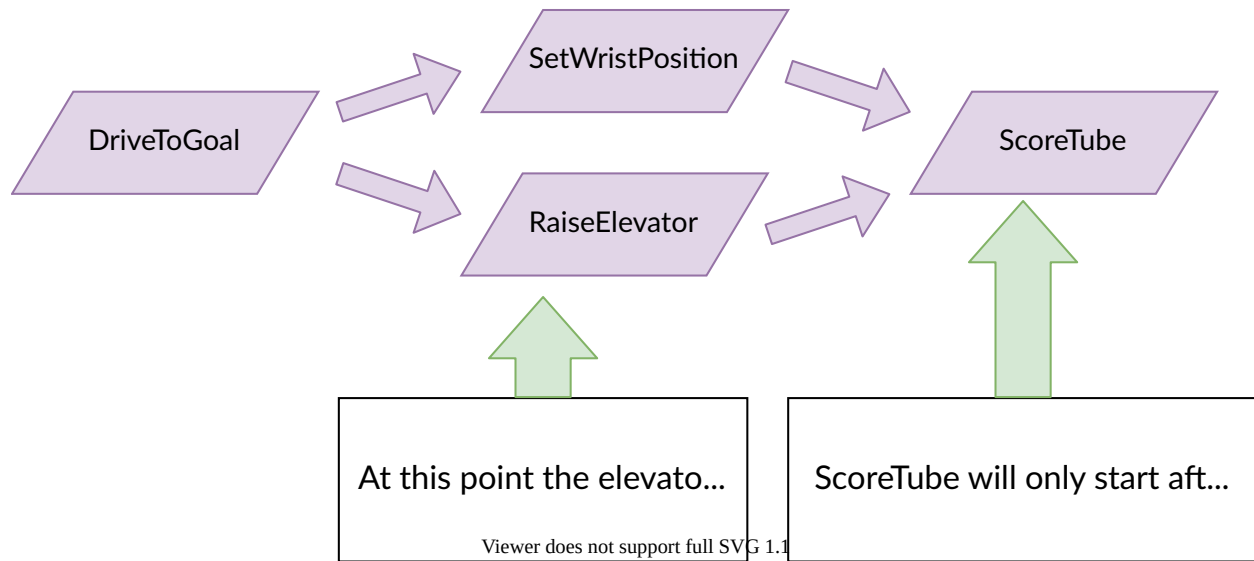
```
new SequentialCommandGroup(
    new DriveToGoal(m_drive),
    new ParallelCommandGroup(
        new RaiseElevator(m_elevator),
        new SetWristPosition(m_wrist)),
    new ScoreTube(m_wrist));
```

```
#include <frc2/command/SequentialCommandGroup.h>
#include <frc2/command/ParallelCommandGroup.h>
```

```
...
```

```
frc2::SequentialCommandGroup{
    DriveToGoal(&m_drive),
    frc2::ParallelCommandGroup{
        RaiseElevator(&m_elevator),
        SetWristPosition(&m_wrist)},
    ScoreTube(&m_wrist)};
```

This creates a sequential command group that *contains* a parallel command group. The resulting control flow looks something like this:



Notice how the recursive composition allows the embedding of a parallel control structure within a sequential one. Notice also that this entire, more-complex structure, could be again embedded in another structure. Composition is an extremely powerful tool, and one that users should be sure to use extensively.

29.6.5 Command Groups and Requirements

As command groups are commands, they also must declare their requirements. However, users are not required to specify requirements manually for command groups - requirements are automatically inferred from the commands included. As a rule, *command groups include the union of all of the subsystems required by their component commands*. Thus, the ComplexAuto shown previously will require both the drive subsystem and the hatch subsystem of the robot.

Additionally, requirements are enforced within all three types of parallel groups - a parallel group may *not* contain multiple commands that require the same subsystem.

Some advanced users may find this overly-restrictive - for said users, the library offers a ScheduleCommand class (Java, C++) that can be used to independently “branch off” from command groups to provide finer granularity in requirement management.

29.6.6 Restrictions on Command Group Components

Note: The following is only relevant for the Java command-based library; the C++ library’s ownership model naturally prevents users from making this category of mistake.

Since command group components are run through their encapsulating command groups, errors could occur if those same command instances were independently scheduled at the same time as the group - the command would be being run from multiple places at once, and thus could end up with inconsistent internal state, causing unexpected and hard-to-diagnose behavior.

For this reason, command instances that have been added to a command group cannot be independently scheduled or added to a second command group. Attempting to do so will throw an exception and crash the user program.

Advanced users who wish to re-use a command instance and are *certain* that it is safe to do so may bypass this restriction with the `clearGroupedCommand()` method in the `CommandGroupBase` class.

29.7 The Command Scheduler

The `CommandScheduler` (Java, C++) is the class responsible for actually running commands. Each iteration (ordinarily once per 20ms), the scheduler polls all registered buttons, schedules commands for execution accordingly, runs the command bodies of all scheduled commands, and ends those commands that have finished or are interrupted.

The `CommandScheduler` also runs the `periodic()` method of each registered Subsystem.

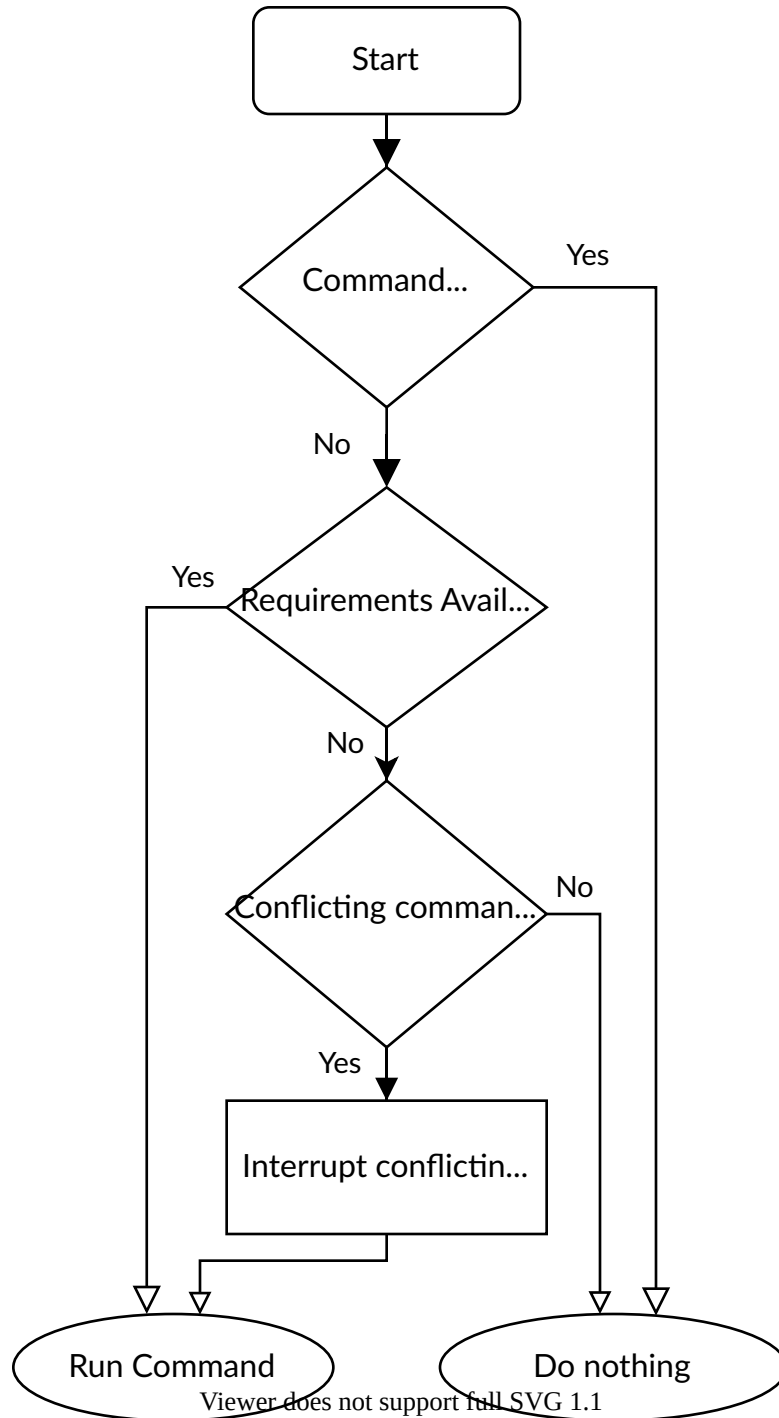
29.7.1 Using the Command Scheduler

The `CommandScheduler` is a *singleton*, meaning that it is a globally-accessible class with only one instance. Accordingly, in order to access the scheduler, users must call the `CommandScheduler.getInstance()` command.

For the most part, users do not have to call scheduler methods directly - almost all important scheduler methods have convenience wrappers elsewhere (e.g. in the `Command` and `Subsystem` interfaces).

However, there is one exception: users *must* call `CommandScheduler.getInstance().run()` from the `robotPeriodic()` method of their `Robot` class. If this is not done, the scheduler will never run, and the command framework will not work. The provided command-based project template has this call already included.

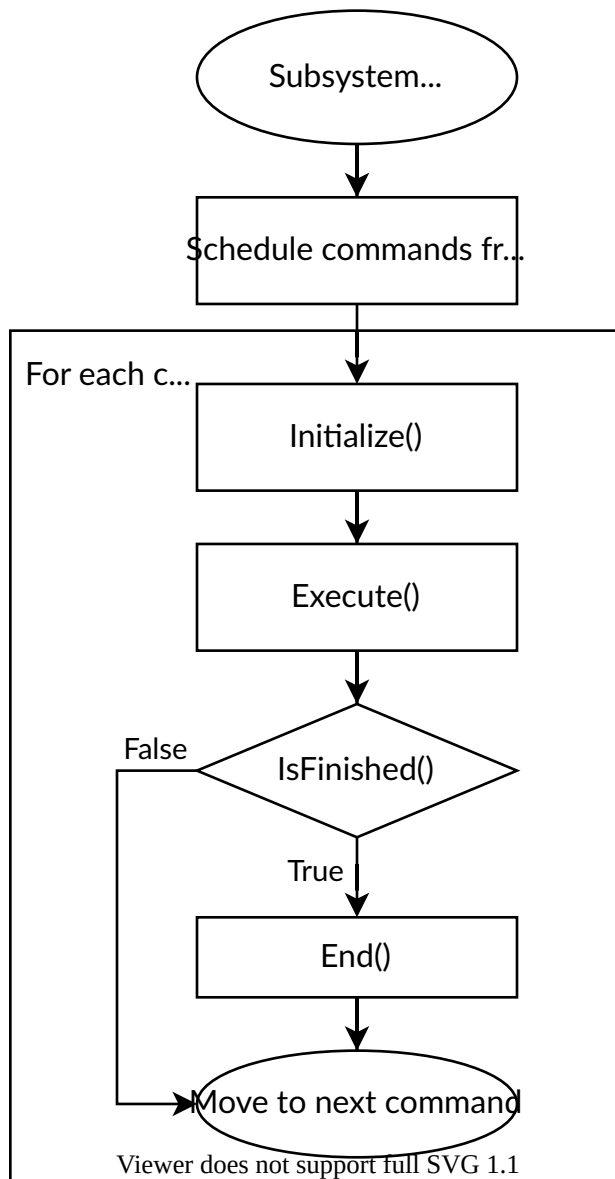
29.7.2 The `schedule()` Method



To schedule a command, users call the `schedule()` method (Java, C++). This method takes a command (and, optionally, a specification as to whether that command is interruptible), and attempts to add it to list of currently-running commands, pending whether it is already running or whether its requirements are available. If it is added, its `initialize()` method is called.

29.7.3 The Scheduler Run Sequence

Note: The `initialize()` method of each `Command` is called when the command is scheduled, which is not necessarily when the scheduler runs (unless that command is bound to a button).



What does a single iteration of the scheduler's `run()` method ([Java](#), [C++](#)) actually do? The following section walks through the logic of a scheduler iteration.

Step 1: Run Subsystem Periodic Methods

First, the scheduler runs the `periodic()` method of each registered Subsystem.

Step 2: Poll Command Scheduling Triggers

Note: For more information on how trigger bindings work, see [Binding Commands to Triggers](#)

Secondly, the scheduler polls the state of all registered triggers to see if any new commands that have been bound to those triggers should be scheduled. If the conditions for scheduling a bound command are met, the command is scheduled and its `Initialize()` method is run.

Step 3: Run/Finish Scheduled Commands

Thirdly, the scheduler calls the `execute()` method of each currently-scheduled command, and then checks whether the command has finished by calling the `isFinished()` method. If the command has finished, the `end()` method is also called, and the command is de-scheduled and its required subsystems are freed.

Note that this sequence of calls is done in order for each command - thus, one command may have its `end()` method called before another has its `execute()` method called. Commands are handled in the order they were scheduled.

Step 4: Schedule Default Commands

Finally, any registered Subsystem has its default command scheduled (if it has one). Note that the `initialize()` method of the default command will be called at this time.

29.7.4 Disabling the Scheduler

The scheduler can be disabled by calling `CommandScheduler.getInstance().disable()`. When disabled, the scheduler's `schedule()` and `run()` commands will not do anything.

The scheduler may be re-enabled by calling `CommandScheduler.getInstance().enable()`.

29.7.5 Command Event Methods

Occasionally, it is desirable to have the scheduler execute a custom action whenever a certain command event (initialization, execution, or ending) occurs. This can be done with the following three methods:

onCommandInitialize

The onCommandInitialize method (Java, C++) runs a specified action whenever a command is initialized.

onCommandExecute

The onCommandExecute method (Java, C++) runs a specified action whenever a command is executed.

onCommandFinish

The onCommandFinish method (Java, C++) runs a specified action whenever a command finishes normally (i.e. the isFinished() method returned true).

onCommandInterrupt

The onCommandInterrupt method (Java, C++) runs a specified action whenever a command is interrupted (i.e. by being explicitly canceled or by another command that shares one of its requirements).

A typical use-case for these methods is adding markers in an event log whenever a command scheduling event takes place, as demonstrated in the SchedulerEventLogging example project (Java, C++):

Java

C++ (Source)

```
50 // Set the scheduler to log Shuffleboard events for command initialize, interrupt,
51 ↪ finish
52 CommandScheduler.getInstance().onCommandInitialize(command -> Shuffleboard.
53 ↪ addEventMarker(
54     "Command initialized", command.getName(), EventImportance.kNormal));
55 CommandScheduler.getInstance().onCommandInterrupt(command -> Shuffleboard.
56 ↪ addEventMarker(
57     "Command interrupted", command.getName(), EventImportance.kNormal));
58 CommandScheduler.getInstance().onCommandFinish(command -> Shuffleboard.
59 ↪ addEventMarker(
60     "Command finished", command.getName(), EventImportance.kNormal));
```

```
22 // Set the scheduler to log Shuffleboard events for command initialize,
23 // interrupt, finish
24 frc2::CommandScheduler::GetInstance().OnCommandInitialize(
25     [](const frc2::Command& command) {
26         frc::Shuffleboard::AddEventMarker(
27             "Command Initialized", command.GetName(),
28             frc::ShuffleboardEventImportance::kNormal);
29     });
30 frc2::CommandScheduler::GetInstance().OnCommandInterrupt(
31     [](const frc2::Command& command) {
32         frc::Shuffleboard::AddEventMarker(
33             "Command Interrupted", command.GetName(),
```

(continues on next page)

(continued from previous page)

```

34         frc::ShuffleboardEventImportance::kNormal);
35     });
36     frc2::CommandScheduler::GetInstance().OnCommandFinish(
37         [](const frc2::Command& command) {
38         frc::Shuffleboard::AddEventMarker(
39             "Command Finished", command.GetName(),
40             frc::ShuffleboardEventImportance::kNormal);
41     });

```

29.8 Binding Commands to Triggers

Apart from autonomous commands, which are scheduled at the start of the autonomous period, and default commands, which are automatically scheduled whenever their subsystem is not currently in-use, the most common way to run a command is by binding it to a triggering event, such as a button being pressed by a human operator. The command-based paradigm makes this extremely easy to do.

As mentioned earlier, command-based is a **declarative** paradigm. Accordingly, binding buttons to commands is done declaratively; the association of a button and a command is “declared” once, during robot initialization. The library then does all the hard work of checking the button state and scheduling (or canceling) the command as needed, behind-the-scenes. Users only need to worry about designing their desired UI setup - not about implementing it!

Command binding is done through the Trigger class (Java, C++) and its various Button subclasses (Java, C++).

29.8.1 Trigger/Button Bindings

Note: The C++ command-based library offers two overloads of each button binding method - one that takes a forwarding reference, and one that takes a raw pointer. The forwarding reference overload transfers ownership (by either moving or copying depending on if the command is an **rvalue** or **lvalue**) to the scheduler, while the raw pointer overload leaves the user responsible for the lifespan of the command object. It is recommended that users preferentially use the forwarding reference overload unless there is a specific need to retain a handle to the command in the calling code.

There are a number of bindings available for the Trigger class. All of these bindings will automatically schedule a command when a certain trigger activation event occurs - however, each binding has different specific behavior. Button and its subclasses have bindings with identical behaviors, but slightly different names that better-match a button rather than an arbitrary triggering event.

whenActive/whenPressed

This binding schedules a command when a trigger changes from inactive to active (or, accordingly, when a button changes is initially pressed). The command will be scheduled on the iteration when the state changes, and will not be scheduled again unless the trigger becomes inactive and then active again (or the button is released and then re-pressed).

whileActiveContinuous/whileHeld

This binding schedules a command repeatedly while a trigger is active (or, accordingly, while a button is held), and cancels it when the trigger becomes inactive (or when the button is released). Note that scheduling an already-running command has no effect; but if the command finishes while the trigger is still active, it will be re-scheduled.

whileActiveOnce/whenHeld

This binding schedules a command when a trigger changes from inactive to active (or, accordingly, when a button is initially pressed) and cancels it when the trigger becomes inactive again (or the button is released). The command will *not* be re-scheduled if it finishes while the trigger is still active.

whenInactive/whenReleased

This binding schedules a command when a trigger changes from active to inactive (or, accordingly, when a button is initially released). The command will be scheduled on the iteration when the state changes, and will not be re-scheduled unless the trigger becomes active and then inactive again (or the button is pressed and then re-released).

toggleWhenActive/toggleWhenPressed

This binding toggles a command, scheduling it when a trigger changes from inactive to active (or a button is initially pressed), and canceling it under the same condition if the command is currently running. Note that while this functionality is supported, toggles are not a highly-recommended option for user control, as they require the driver to keep track of the robot state. The preferred method is to use two buttons; one to turn on and another to turn off. Using a [StartEndCommand](#) or a [ConditionalCommand](#) is a good way to specify the commands that you want to be want to be toggled between.

Java

C++

```
myButton.toggleWhenPressed(new StartEndCommand(mySubsystem::onMethod,  
    mySubsystem::offMethod,  
    mySubsystem));
```

```
myButton.ToggleWhenPressed(StartEndCommand([&] { mySubsystem.OnMethod(); },  
    [&] { mySubsystem.OffMethod(); },  
    {&mySubsystem}));
```

cancelWhenActive/cancelWhenPressed

This binding cancels a command when a trigger changes from inactive to active (or, accordingly, when a button is initially pressed). the command is canceled on the iteration when the state changes, and will not be canceled again unless the trigger becomes inactive and then active again (or the button is released and re-pressed). Note that canceling a command that is not currently running has no effect.

29.8.2 Binding a command to a joystick button

The most-common way to trigger a command is to bind a command to a button on a joystick or other HID (human interface device). To do this, users should use the JoystickButton class.

Creating a JoystickButton

In order to create a JoystickButton, we first need a Joystick. All types of joysticks (including gamepads) are represented in code by the GenericHID class (Java, C++), or one of its subclasses:

Java

C++

```
Joystick exampleStick = new Joystick(1); // Creates a joystick on port 1
XboxController exampleController = new XboxController(2); // Creates an
↳ XboxController on port 2.
```

```
frc::Joystick exampleStick{1}; // Creates a joystick on port 1
frc::XBoxController exampleController{2} // Creates an XboxController on port 2
```

Note: When creating a JoystickButton with an XboxController, it is usually a good idea to use the button enum (Java, C++) to get the button number corresponding to a given button.

After the joystick is instantiated, users can then pass it to a JoystickButton object (Java, C++):

Java

C++

```
JoystickButton exampleButton = new JoystickButton(exampleStick, 1); // Creates a new
↳ JoystickButton object for button 1 on exampleStick
```

```
frc2::JoystickButton exampleButton(&exampleStick, 1); // Creates a new JoystickButton
↳ object for button 1 on exampleStick
```

Binding a Command to a JoystickButton

Note: In the C++ command-based library, button objects *do not need to survive past the call to a binding method*, and so the binding methods may be simply called on a temp.

Putting it all together, it is very simple to bind a button to a JoystickButton:

Java

C++

```
// Binds an ExampleCommand to be scheduled when the trigger of the example joystick
↳is pressed
exampleButton.whenPressed(new ExampleCommand());
```

```
// Binds an ExampleCommand to be scheduled when the trigger of the example joystick
↳is pressed
exampleButton.WhenPressed(ExampleCommand());
```

It is useful to note that the command binding methods all return the trigger/button that they were initially called on, and thus can be chained to bind multiple commands to different states of the same button. For example:

Java

C++

```
exampleButton
    // Binds a FooCommand to be scheduled when the `X` button of the driver gamepad
    ↳is pressed
    .whenPressed(new FooCommand())
    // Binds a BarCommand to be scheduled when that same button is released
    .whenReleased(new BarCommand());
```

```
exampleButton
    // Binds a FooCommand to be scheduled when the `X` button of the driver gamepad
    ↳is pressed
    .WhenPressed(FooCommand())
    // Binds a BarCommand to be scheduled when that same button is released
    .WhenReleased(BarCommand());
```

Remember that button binding is *declarative*: bindings only need to be declared once, ideally some time during robot initialization. The library handles everything else.

29.8.3 Composing Triggers

The Trigger class (including its Button subclasses) can be composed to create composite triggers through the `and()`, `or()`, and `negate()` methods (or, in C++, the `&&`, `||`, and `!` operators). For example:

Java

C++


```
// Binds an ExampleCommand to be scheduled when both the 'X' and 'Y' buttons of the
↳ driver gamepad are pressed
new JoystickButton(exampleController, XboxController.Button.kX.value)
    .and(new JoystickButton(exampleController, XboxController.Button.kY.value))
    .whenActive(new ExampleCommand());
```

```
// Binds an ExampleCommand to be scheduled when both the 'X' and 'Y' buttons of the
↳ driver gamepad are pressed
(frc2::JoystickButton(&exampleController, frc2::XboxController::Button::kX)
    && JoystickButton(&exampleController, frc2::XboxController::Button::kY))
    .WhenActive(new ExampleCommand());
```

Note that these methods return a Trigger, not a Button, so the Trigger binding method names must be used even when buttons are composed.

29.8.4 Creating Your Own Custom Trigger

While binding to HID buttons is by far the most common use case, advanced users may occasionally want to bind commands to arbitrary triggering events. This can be easily done by simply writing your own subclass of trigger or button:

Java

C++

```
public class ExampleTrigger extends Trigger {
    @Override
    public boolean get() {
        // This returns whether the trigger is active
    }
}
```

```
class ExampleTrigger : public frc2::Trigger {
public:
    bool get() override {
        // This returns whether the trigger is active
    }
}
```

Alternatively, this can also be done inline by passing a lambda to the constructor of trigger or button:

Java

C++

```
// Here it is assumed that "condition" is an object with a method "get" that returns
↳ whether the trigger should be active
Trigger exampleTrigger = new Trigger(condition::get);
```

```
// Here it is assumed that "condition" is a boolean that determines whether the
↳ trigger should be active
frc2::Trigger exampleTrigger([&condition] { return condition; });
```

29.9 Structuring a Command-Based Robot Project

While users are free to use the command-based libraries however they like (and advanced users are encouraged to do so), new users may want some guidance on how to structure a basic command-based robot project.

A standard template for a command-based robot project is included in the WPILib examples repository (Java, C++). This section will walk users through the structure of this template.

The root package/directory generally will contain four classes:

Main, which is the main robot application (Java only). New users *should not* touch this class. Robot, which is responsible for the main control flow of the robot code. RobotContainer, which holds robot subsystems and commands, and is where most of the declarative robot setup (e.g. button bindings) is performed. Constants, which holds globally-accessible constants to be used throughout the robot.

The root directory will also contain two sub-packages/sub-directories: Subsystems contains all user-defined subsystem classes. Commands contains all user-defined command classes.

29.9.1 Robot

As Robot (Java, C++ (Header), C++ (Source)) is responsible for the program's control flow, and command-based is an declarative paradigm designed to minimize the amount of attention the user has to pay to explicit program control flow, the Robot class of a command-based project should be mostly empty. However, there are a few important things that must be included

Java

```

25  /**
26   * This function is run when the robot is first started up and should be used for
↳any
27   * initialization code.
28   */
29   @Override
30   public void robotInit() {
31       // Instantiate our RobotContainer. This will perform all our button bindings,
↳and put our
32       // autonomous chooser on the dashboard.
33       m_robotContainer = new RobotContainer();
34   }
```

In Java, an instance of RobotContainer should be constructed during the robotInit() method - this is important, as most of the declarative robot setup will be called from the RobotContainer constructor.

In C++, this is not needed as RobotContainer is a value member and will be constructed during the construction of Robot.

Java

C++ (Source)

```

36  /**
37   * This function is called every robot packet, no matter the mode. Use this for
↳items like
```

(continues on next page)

(continued from previous page)

```

38  * diagnostics that you want ran during disabled, autonomous, teleoperated and test.
39  *
40  * <p>This runs after the mode specific periodic functions, but before
41  * LiveWindow and SmartDashboard integrated updating.
42  */
43  @Override
44  public void robotPeriodic() {
45      // Runs the Scheduler. This is responsible for polling buttons, adding newly-
46      ↪ scheduled
47      ↪ commands,
48      ↪ // commands, running already-scheduled commands, removing finished or interrupted
49      ↪ commands,
50      ↪ // and running subsystem periodic() methods. This must be called from the robot
51      ↪ 's periodic
52      ↪ // block in order for anything in the Command-based framework to work.
53      CommandScheduler.getInstance().run();
54  }

```

```

15  /**
16   * This function is called every robot packet, no matter the mode. Use
17   * this for items like diagnostics that you want to run during disabled,
18   * autonomous, teleoperated and test.
19   *
20   * <p> This runs after the mode specific periodic functions, but before
21   * LiveWindow and SmartDashboard integrated updating.
22   */
23  void Robot::RobotPeriodic() { frc2::CommandScheduler::GetInstance().Run(); }

```

The inclusion of the `CommandScheduler.getInstance().run()` call in the `robotPeriodic()` method is essential; without this call, the scheduler will not execute any scheduled commands. Since `TimedRobot` runs with a default main loop frequency of 50Hz, this is the frequency with which periodic command and subsystem methods will be called. It is not recommended for new users to call this method from anywhere else in their code.

Java

C++ (Source)

```

63  /**
64   * This autonomous runs the autonomous command selected by your {@link
65   ↪ RobotContainer} class.
66   */
67  @Override
68  public void autonomousInit() {
69      m_autonomousCommand = m_robotContainer.getAutonomousCommand();
70
71      // schedule the autonomous command (example)
72      if (m_autonomousCommand != null) {
73          m_autonomousCommand.schedule();
74      }
75  }

```

```

34  /**
35   * This autonomous runs the autonomous command selected by your {@link
36   * RobotContainer} class.
37   */
38  void Robot::AutonomousInit() {

```

(continues on next page)

(continued from previous page)

```

39  m_autonomousCommand = m_container.GetAutonomousCommand();
40
41  if (m_autonomousCommand != nullptr) {
42      m_autonomousCommand->Schedule();
43  }
44  }

```

The `autonomousInit()` method schedules an autonomous command returned by the `RobotContainer` instance. The logic for selecting which autonomous command to run can be handled inside of `RobotContainer`.

Java

C++ (Source)

```

83  @Override
84  public void teleopInit() {
85      // This makes sure that the autonomous stops running when
86      // teleop starts running. If you want the autonomous to
87      // continue until interrupted by another command, remove
88      // this line or comment it out.
89      if (m_autonomousCommand != null) {
90          m_autonomousCommand.cancel();
91      }
92  }

```

```

48  void Robot::TeleopInit() {
49      // This makes sure that the autonomous stops running when
50      // teleop starts running. If you want the autonomous to
51      // continue until interrupted by another command, remove
52      // this line or comment it out.
53      if (m_autonomousCommand != nullptr) {
54          m_autonomousCommand->Cancel();
55          m_autonomousCommand = nullptr;
56      }
57  }

```

The `teleopInit()` method cancels any still-running autonomous commands. This is generally good practice.

Advanced users are free to add additional code to the various init and periodic methods as they see fit; however, it should be noted that including large amounts of imperative robot code in `Robot.java` is contrary to the declarative design philosophy of the command-based paradigm, and can result in confusingly-structured/disorganized code.

29.9.2 RobotContainer

This class (Java, C++ (Header), C++ (Source)) is where most of the setup for your command-based robot will take place. In this class, you will define your robot's subsystems and commands, bind those commands to triggering events (such as buttons), and specify which command you will run in your autonomous routine. There are a few aspects of this class new users may want explanations for:

Java

C++ (Header)

```
24 private final ExampleSubsystem m_exampleSubsystem = new ExampleSubsystem();
```

```
28 private:
```

```
29 // The robot's subsystems and commands are defined here...
```

```
30 ExampleSubsystem m_subsystem;
```

Notice that subsystems are declared as private fields in `RobotContainer`. This is in stark contrast to the previous incarnation of the command-based framework, but is much more aligned with agreed-upon object-oriented best-practices. If subsystems are declared as global variables, it allows the user to access them from anywhere in the code. While this can make certain things easier (for example, there would be no need to pass subsystems to commands in order for those commands to access them), it makes the control flow of the program much harder to keep track of as it is not immediately obvious which parts of the code can change or be changed by which other parts of the code. This also circumvents the ability of the resource-management system to do its job, as ease-of-access makes it easy for users to accidentally make conflicting calls to subsystem methods outside of the resource-managed commands.

Java

C++ (Source)

```
26 private final ExampleCommand m_autoCommand = new ExampleCommand(m_exampleSubsystem);
```

```
10 RobotContainer::RobotContainer() : m_autonomousCommand(&m_subsystem) {
```

Since subsystems are declared as private members, they must be explicitly passed to commands (a pattern called “dependency injection”) in order for those commands to call methods on them. This is done here with `ExampleCommand`, which is passed a pointer to an `ExampleSubsystem`.

Java

C++ (Source)

```
38 /**
39  * Use this method to define your button->command mappings. Buttons can be created
↳ by
40  * instantiating a {@link GenericHID} or one of its subclasses ({@link
41  * edu.wpi.first.wpilibj.Joystick} or {@link XboxController}), and then passing it
↳ to a
42  * {@link edu.wpi.first.wpilibj2.command.button.JoystickButton}.
43  */
44 private void configureButtonBindings() {
45 }
```

```
17 void RobotContainer::ConfigureButtonBindings() {
18 // Configure your button bindings here
19 }
```

As mentioned before, the `RobotContainer()` constructor is where most of the declarative setup for the robot should take place, including button bindings, configuring autonomous selectors, etc. If the constructor gets too “busy,” users are encouraged to migrate code into separate subroutines (such as the `configureButtonBindings()` method included by default) which are called from the constructor.

Java

C++ (Source)

```

48  /**
49   * Use this to pass the autonomous command to the main {@link Robot} class.
50   *
51   * @return the command to run in autonomous
52   */
53  public Command getAutonomousCommand() {
54      // An ExampleCommand will run in autonomous
55      return m_autoCommand;
56  }
57  }

```

```

21  frc2::Command* RobotContainer::GetAutonomousCommand() {
22      // An example command will be run in autonomous
23      return &m_autonomousCommand;
24  }

```

Finally, the `getAutonomousCommand()` method provides a convenient way for users to send their selected autonomous command to the main `Robot` class (which needs access to it to schedule it when autonomous starts).

29.9.3 Constants

The `Constants` class ([Java](#), [C++ \(Header\)](#)) (in C++ this is not a class, but simply a header file in which several namespaces are defined) is where globally-accessible robot constants (such as speeds, unit conversion factors, PID gains, and sensor/motor ports) can be stored. It is recommended that users separate these constants into individual inner classes corresponding to subsystems or robot modes, to keep variable names shorter.

In Java, all constants should be declared `public static final` so that they are globally accessible and cannot be changed. In C++, all constants should be `constexpr`.

For more illustrative examples of what a constants class should look like in practice, see those of the various command-based example projects:

- [FrisbeeBot \(Java, C++\)](#)
- [GyroDriveCommands \(Java, C++\)](#)
- [Hatchbot \(Java, C++\)](#)

In Java, it is recommended that the constants be used from other classes by statically importing the necessary inner class. An `import static` statement imports the static namespace of a class into the class in which you are working, so that any `static` constants can be referenced directly as if they had been defined in that class. In C++, the same effect can be attained with `using namespace`:

Java

C++

```
import static edu.wpi.first.wpilibj.templates.commandbased.Constants.OIConstants.*;
```

```
using namespace OIConstants;
```

29.9.4 Subsystems

User-defined subsystems should go in this package/directory.

29.9.5 Commands

User-defined commands should go in this package/directory.

29.10 Convenience Features

While the previously-described methodologies will work fine for writing command-based robot code, the command-based libraries contain several convenience features for more-advanced users that can greatly reduce the verbosity/complexity of command-based code. It is highly recommended that users familiarize themselves with these features to maximize the value they get out of the command-based libraries.

29.10.1 Inline Command Definitions

While users are able to create commands by explicitly writing command classes (either by subclassing `CommandBase` or implementing `Command`), for many commands (such as those that simply call a single subsystem method) this involves a lot of wasteful boilerplate code. To help alleviate this, many of the prewritten commands included in the command-based library may be *inlined* - that is, the command body can be defined in a single line of code at command construction.

Passing Subroutines As Parameters

In order to inline a command definition, users require some way to specify what code the commands will run as constructor parameters. Fortunately, both Java and C++ offer users the ability to pass subroutines as parameters.

Method References (Java)

In Java, a reference to a subroutine that can be passed as a parameter is called a method reference. The general syntax for a method reference is `object::method`. Note that no method parameters are included, since the method *itself* is the parameter. The method is not being called - it is being passed to another piece of code (in this case, a command) so that *that* code can call it when needed. For further information on method references, see [the official Oracle documentation](#).

Lambda Expressions (Java)

While method references work well for passing a subroutine that has already been written, often it is inconvenient/wasteful to write a subroutine solely for the purpose of sending as a method reference, if that subroutine will never be used elsewhere. To avoid this, Java also supports a feature called “lambda expressions.” A lambda expression is an inline method definition - it allows a subroutine to be defined *inside of a parameter list*. For specifics on how to write Java lambda expressions, see [this tutorial](#).

Lambda Expressions (C++)

C++ lacks a close equivalent to Java method references - pointers to member functions are generally not directly useable as parameters due to the presence of the implicit `this` parameter. However, C++ does offer lambda expressions - in addition, the lambda expressions offered by C++ are in many ways more powerful than those in Java. For specifics on how to write C++ lambda expressions, see [cppreference](#).

Inlined Command Example

So, what does an inlined command definition look like in practice?

The `InstantCommand` class provides an example of a type of command that benefits greatly from inlining. Consider the following from the `HatchBotInlined` example project ([Java](#), [C++](#)):

Java

C++ (Header)

C++ (Source)

```
99     new JoystickButton(m_driverController, Button.kB.value)  
100     .whenPressed(new InstantCommand(m_hatchSubsystem::releaseHatch, m_  
↪ hatchSubsystem));  
101     // While holding the shoulder button, drive at half speed  
102     new JoystickButton(m_driverController, Button.kBumperRight.value)  
103     .whenPressed(() -> m_robotDrive.setMaxOutput(0.5))  
104     .whenReleased(() -> m_robotDrive.setMaxOutput(1));
```

```
63     frc2::InstantCommand m_grabHatch{[this] { m_hatch.GrabHatch(); }, {&m_hatch}};  
64     frc2::InstantCommand m_releaseHatch{[this] { m_hatch.ReleaseHatch(); },  
65                                         {&m_hatch}};
```

```
39     // Grab the hatch when the 'A' button is pressed.  
40     frc2::JoystickButton(&m_driverController, 1).WhenPressed(&m_grabHatch);  
41     // Release the hatch when the 'B' button is pressed.  
42     frc2::JoystickButton(&m_driverController, 2).WhenPressed(&m_releaseHatch);
```

Instead of wastefully writing separate `GrabHatch` and `ReleaseHatch` commands which call only one method before ending, both can be accomplished with a simple inline definition by passing appropriate subsystem method.

29.10.2 Included Command Types

The command-based library includes a variety of pre-written commands for commonly-encountered use cases. Many of these commands are intended to be used “out-of-the-box” via *inlining*, however they may be subclassed, as well. A list of the included pre-made commands can be found below, along with brief examples of each - for more rigorous documentation, see the API docs ([Java](#), [C++](#)).

ConditionalCommand

The ConditionalCommand class ([Java](#), [C++](#)) runs one of two commands when executed, depending on a user-specified true-or-false condition:

Java

C++

```
// Runs either commandOnTrue or commandOnFalse depending on the value of m_
↳ limitSwitch.get()
new ConditionalCommand(commandOnTrue, commandOnFalse, m_limitSwitch::get)
```

```
// Runs either commandOnTrue or commandOnFalse depending on the value of m_
↳ limitSwitch.get()
frc2::ConditionalCommand(commandOnTrue, commandOnFalse, [&m_limitSwitch] { return m_
↳ limitSwitch.Get(); })
```

SelectCommand

Note: While the Java version of SelectCommand simply uses an Object as a key, the C++ version is templated on the key type.

Note: An alternate version of SelectCommand simply takes a method that supplies the command to be run - this can be very succinct, but makes inferring the command’s requirements impossible, and so leaves the user responsible for manually adding the requirements to the SelectCommand.

The SelectCommand class ([Java](#), [C++](#)) is a generalization of the ConditionalCommand class that runs one of a selection of commands based on the value of a user-specified selector. The following example code is taken from the SelectCommand example project ([Java](#), [C++](#)):

Java

C++ (Header)

```
20 public class RobotContainer {
21     // The enum used as keys for selecting the command to run.
22     private enum CommandSelector {
23         ONE,
24         TWO,
25         THREE
26     }
```

(continues on next page)

(continued from previous page)

```

27 // An example selector method for the selectcommand. Returns the selector that
28 // will select
29 // which command to run. Can base this choice on logical conditions evaluated at
30 // runtime.
31 private CommandSelector select() {
32     return CommandSelector.ONE;
33 }
34 // An example selectcommand. Will select from the three commands based on the
35 // value returned
36 // by the selector method at runtime. Note that selectcommand works on Object(),
37 // so the
38 // selector does not have to be an enum; it could be any desired type (string,
39 // integer,
40 // boolean, double...)
41 private final Command m_exampleSelectCommand =
42     new SelectCommand(
43         // Maps selector values to commands
44         Map.ofEntries(
45             Map.entry(CommandSelector.ONE, new PrintCommand("Command one was
46 // selected!")),
47             Map.entry(CommandSelector.TWO, new PrintCommand("Command two was
48 // selected!")),
49             Map.entry(CommandSelector.THREE, new PrintCommand("Command three was
50 // selected!"))),
51         this::select);

```

```

28 // The enum used as keys for selecting the command to run.
29 enum CommandSelector { ONE, TWO, THREE };
30
31 // An example selector method for the selectcommand. Returns the selector
32 // that will select which command to run. Can base this choice on logical
33 // conditions evaluated at runtime.
34 CommandSelector Select() { return ONE; }
35
36 // The robot's subsystems and commands are defined here...
37
38 // An example selectcommand. Will select from the three commands based on the
39 // value returned by the selector method at runtime. Note that selectcommand
40 // takes a generic type, so the selector does not have to be an enum; it could
41 // be any desired type (string, integer, boolean, double...)
42 frc2::SelectCommand<CommandSelector> m_exampleSelectCommand{
43     [this] { return Select(); },
44     // Maps selector values to commands
45     std::pair{ONE, frc2::PrintCommand{"Command one was selected!"}},
46     std::pair{TWO, frc2::PrintCommand{"Command two was selected!"}},
47     std::pair{THREE, frc2::PrintCommand{"Command three was selected!"}}};

```

InstantCommand

The InstantCommand class (Java, C++) executes a single action on initialization, and then ends immediately:

Java

C++

```
// Actuates the hatch subsystem to grab the hatch
new InstantCommand(m_hatchSubsystem::grabHatch, m_hatchSubsystem)
```

```
// Actuates the hatch subsystem to grab the hatch
frc2::InstantCommand([&m_hatchSubsystem] { m_hatchSubsystem.GrabHatch(); }, {&m_
↪ hatchSubsystem})
```

RunCommand

The RunCommand class (Java, C++) runs a specified method repeatedly in its execute() block. It does not have end conditions by default; users can either subclass it, or *decorate* it to add them.

Java

C++

```
// A split-stick arcade command, with forward/backward controlled by the left
// hand, and turning controlled by the right.
new RunCommand(() -> m_robotDrive.arcadeDrive(
    -driverController.getY(GenericHID.Hand.kLeft),
    driverController.getX(GenericHID.Hand.kRight)),
    m_robotDrive)
```

```
// A split-stick arcade command, with forward/backward controlled by the left
// hand, and turning controlled by the right.
frc2::RunCommand(
    [this] {
        m_drive.ArcadeDrive(
            -m_driverController.GetY(frc::GenericHID::kLeftHand),
            m_driverController.GetX(frc::GenericHID::kRightHand));
    },
    {&m_drive}))
```

StartEndCommand

The StartEndCommand class (Java, C++) executes an action when starting, and a second one when ending. It does not have end conditions by default; users can either subclass it, or *decorate* an inlined command to add them.

Java

C++

```
new StartEndCommand(  
    // Start a flywheel spinning at 50% power  
    () -> m_shooter.shooterSpeed(0.5),  
    // Stop the flywheel at the end of the command  
    () -> m_shooter.shooterSpeed(0.0),  
    // Requires the shooter subsystem  
    m_shooter  
)
```

```
frc2::StartEndCommand(  
    // Start a flywheel spinning at 50% power  
    [this] { m_shooter.shooterSpeed(0.5); },  
    // Stop the flywheel at the end of the command  
    [this] { m_shooter.shooterSpeed(0.0); },  
    // Requires the shooter subsystem  
    {&m_shooter}  
)
```

FunctionalCommand

The FunctionalCommand class (Java, C++) allows all four Command methods to be passed in as method references or lambdas:

Java

C++

```
new FunctionalCommand(  
    // Reset encoders on command start  
    m_robotDrive::resetEncoders,  
    // Start driving forward at the start of the command  
    () -> m_robotDrive.arcadeDrive(kAutoDriveSpeed, 0),  
    // Stop driving at the end of the command  
    interrupted -> m_robotDrive.arcadeDrive(0, 0),  
    // End the command when the robot's driven distance exceeds the desired value  
    () -> m_robotDrive.getAverageEncoderDistance() >= kAutoDriveDistanceInches,  
    // Require the drive subsystem  
    m_robotDrive  
)
```

```
frc2::FunctionalCommand(  
    // Reset encoders on command start  
    [this] { m_drive.ResetEncoders(); },  
    // Start driving forward at the start of the command  
    [this] { m_drive.ArcadeDrive(ac::kAutoDriveSpeed, 0); },  
    // Stop driving at the end of the command  
    [this] (bool interrupted) { m_drive.ArcadeDrive(0, 0); },  
    // End the command when the robot's driven distance exceeds the desired value  
    [this] { return m_drive.GetAverageEncoderDistance() >= kAutoDriveDistanceInches; },  
    // Requires the drive subsystem  
    {&m_drive}  
)
```

PrintCommand

The PrintCommand class (Java, C++) prints a given string.

Java

C++

```
new PrintCommand("This message will be printed!")
```

```
frc2::PrintCommand("This message will be printed!")
```

ScheduleCommand

The ScheduleCommand class (Java, C++) schedules a specified command, and ends instantly:

Java

C++

```
// Schedules commandToSchedule when run
new ScheduleCommand(commandToSchedule)
```

```
// Schedules commandToSchedule when run
frc2::ScheduleCommand(&commandToSchedule)
```

This is often useful for “forking off” from command groups: by default, commands in command groups are run *through* the command group, and are never themselves seen by the scheduler. Accordingly, their requirements are added to the group’s requirements. While this is usually fine, sometimes it is undesirable for the entire command group to gain the requirements of a single command - a good solution is to “fork off” from the command group and schedule that command separately.

ProxyScheduleCommand

The ProxyScheduleCommand class (Java, C++) schedules a specified command, and does not end until that command ends:

Java

C++

```
// Schedules commandToSchedule when run, does not end until commandToSchedule is no
↳ longer scheduled
new ProxyScheduleCommand(commandToSchedule)
```

```
// Schedules commandToSchedule when run, does not end until commandToSchedule is no
↳ longer scheduled
frc2::ProxyScheduleCommand(&commandToSchedule)
```

This is often useful for “forking off” from command groups: by default, commands in command groups are run *through* the command group, and are never themselves seen by the scheduler. Accordingly, their requirements are added to the group’s requirements. While this is usually fine, sometimes it is undesirable for the entire command group to gain the requirements of a

single command - a good solution is to “fork off” from the command group and schedule the command separately.

WaitCommand

The `WaitCommand` class (Java, C++) does nothing, and ends after a specified period of time elapses after its initial scheduling:

Java

C++

```
// Ends 5 seconds after being scheduled
new WaitCommand(5)
```

```
// Ends 5 seconds after being scheduled
frc2::WaitCommand(5.0_s)
```

This is often useful as a component of a command group.

`WaitCommand` can also be subclassed to create a more complicated command that runs for a period of time. If `WaitCommand` is used in this method, the user must ensure that the `WaitCommand`’s `Initialize`, `End`, and `IsFinished` methods are still called in order for the `WaitCommand`’s timer to work.

WaitUntilCommand

Warning: The match timer used by `WaitUntilCommand` does *not* provide an official match time! While it is fairly accurate, use of this timer can *not* guarantee the legality of your robot’s actions.

The `WaitUntilCommand` class (Java, C++) does nothing, and ends once a specified condition becomes true, or until a specified match time passes.

Java

C++

```
// Ends after the 60-second mark of the current match
new WaitUntilCommand(60)
```

```
// Ends after m_limitSwitch.get() returns true
new WaitUntilCommand(m_limitSwitch::get)
```

```
// Ends after the 60-second mark of the current match
frc2::WaitUntilCommand(60.0_s)
```

```
// Ends after m_limitSwitch.Get() returns true
frc2::WaitUntilCommand([&m_limitSwitch] { return m_limitSwitch.Get(); })
```

PerpetualCommand

The PerpetualCommand class (Java, C++) runs a given command with its end condition removed, so that it runs forever (unless externally interrupted):

Java

C++

```
// Will run commandToRunForever perpetually, even if its isFinished() method returns true
new PerpetualCommand(commandToRunForever)
```

```
// Will run commandToRunForever perpetually, even if its isFinished() method returns true
frc2::PerpetualCommand(commandToRunForever)
```

29.10.3 Command Decorator Methods

The Command interface contains a number of defaulted “decorator” methods which can be used to add additional functionality to existing commands. A “decorator” method is a method that takes an object (in this case, a command) and returns an object of the same type (i.e. a command) with some additional functionality added to it. A list of the included decorator methods with brief examples is included below - for rigorous documentation, see the API docs (Java, C++).

withTimeout

The withTimeout() decorator (Java, C++) adds a timeout to a command. The decorated command will be interrupted if the timeout expires:

Java

C++

```
// Will time out 5 seconds after being scheduled, and be interrupted
button.whenPressed(command.withTimeout(5));
```

```
// Will time out 5 seconds after being scheduled, and be interrupted
button.WhenPressed(command.WithTimeout(5.0_s));
```

withInterrupt

The withInterrupt() (Java, C++) decorator adds a condition on which the command will be interrupted:

Java

C++

```
// Will be interrupted if m_limitSwitch.get() returns true
button.whenPressed(command.withInterrupt(m_limitSwitch::get));
```

```
// Will be interrupted if m_limitSwitch.get() returns true
button.WhenPressed(command.WithInterrupt([&m_limitSwitch] { return m_limitSwitch.
    ↪Get(); }));
```

andThen

The `andThen()` decorator (Java, C++) adds a method to be executed after the command ends:

Java

C++

```
// Will print "hello" after ending
button.whenPressed(command.andThen(() -> System.out.println("hello")));
```

```
// Will print "hello" after ending
button.WhenPressed(command.AndThen([] { std::cout << "hello"; }));
```

beforeStarting

The `beforeStarting()` decorator (Java, C++) adds a method to be executed before the command starts:

Java

C++

```
// Will print "hello" before starting
button.whenPressed(command.beforeStarting(() -> System.out.println("hello")));
```

```
// Will print "hello" before starting
button.WhenPressed(command.BeforeStarting([] { std::cout << "hello"; }));
```

alongWith (Java only)

Note: This decorator is not supported in C++ due to technical constraints - users should simply construct a parallel command group the ordinary way instead.

The `alongWith()` decorator returns a *parallel command group*. All commands will execute at the same time and each will end independently of each other:

```
// Will be a parallel command group that ends after three seconds with all three
    ↪commands running their full duration.
button.whenPressed(oneSecCommand.alongWith(twoSecCommand, threeSecCommand));
```


raceWith (Java only)

Note: This decorator is not supported in C++ due to technical constraints - users should simply construct a parallel race group the ordinary way instead.

The `raceWith()` decorator returns a *parallel race group* that ends as soon as the first command ends. At this point all others are interrupted. It doesn't matter which command is the calling command:

```
// Will be a parallel race group that ends after one second with the two and three
// second commands getting interrupted.
button.whenPressed(twoSecCommand.raceWith(oneSecCommand, threeSecCommand));
```

deadlineWith (Java only)

Note: This decorator is not supported in C++ due to technical constraints - users should simply construct a parallel deadline group the ordinary way instead.

The `deadlineWith()` decorator returns a *parallel deadline group* with the calling command being the deadline. When this deadline command ends it will interrupt any others that are not finished:

```
// Will be a parallel deadline group that ends after two seconds (the deadline) with
// the three second command getting interrupted (one second command already finished).
button.whenPressed(twoSecCommand.deadlineWith(oneSecCommand, threeSecCommand));
```

withName (Java only)

Note: This decorator is not supported in C++ due to technical constraints - users should set the name of the command inside their command class instead.

The `withName()` decorator adds a name to a command. This name will appear on a dashboard when the command is sent via the *sendable interface*.

```
// This command will be called "My Command".
var command = new PrintCommand("Hello robot!").withName("My Command");
```

perpetually

The `perpetually()` decorator (Java, C++) removes the end condition of a command, so that it runs forever.

Java

C++

```
// Will run forever unless externally interrupted, regardless of command.isFinished()
button.whenPressed(command.perpetually());
```

```
// Will run forever unless externally interrupted, regardless of command.isFinished()
button.WhenPressed(command.Perpetually());
```

Composing Decorators

Remember that decorators, like all command groups, can be composed! This allows very powerful and concise inline expressions:

```
// Will run fooCommand, and then a race between barCommand and bazCommand
button.whenPressed( fooCommand.andThen( barCommand.raceWith( bazCommand) ) );
```

29.10.4 Static Factory Methods for Command Groups (Java only)

Note: These factory methods are not included in the C++ command library, as the reduction in verbosity would be minimal - C++ commands should be stack-allocated, removing the need for the new keyword.

If users do not wish to use the `andThen`, `alongWith`, `raceWith`, and `deadlineWith` decorators for declaring command groups, but still wish to reduce verbosity compared to calling the constructors, the `CommandGroupBase` class contains four static factory methods for declaring command groups: `sequence()`, `parallel()`, `race()`, and `deadline()`. When used from within a command group subclass or in combination with `import static`, these become extremely concise and greatly aid in command composition:

```
public class ExampleSequence extends SequentialCommandGroup {

    // Will run a FooCommand, and then a race between a BarCommand and a BazCommand
    public ExampleSequence() {
        addCommands(
            new FooCommand(),
            race(
                new BarCommand(),
                new BazCommand()
            )
        );
    }
}
```

29.11 PID Control through PIDSubsystems and PIDCommands

Note: For a description of the WPILib PID control features used by these command-based wrappers, see [PID Control in WPILib](#).

Note: Unlike the earlier version of `PIDController`, the 2020 `PIDController` class runs *synchronously*, and is not handled in its own thread. Accordingly, changing its period parameter will *not* change the actual frequency at which it runs in any of these wrapper classes. Users should never modify the period parameter unless they are certain of what they are doing.

One of the most common control algorithms used in FRC® is the [PID controller](#). WPILib offers its own [PIDController](#) class to help teams implement this functionality on their robots. To further help teams integrate PID control into a command-based robot project, the command-based library includes two convenience wrappers for the `PIDController` class: `PIDSubsystem`, which integrates the PID controller into a subsystem, and `PIDCommand`, which integrates the PID controller into a command.

29.11.1 PIDSubsystems

The `PIDSubsystem` class (Java, C++) allows users to conveniently create a subsystem with a built-in `PIDController`. In order to use the `PIDSubsystem` class, users must create a subclass of it.

Creating a PIDSubsystem

When subclassing `PIDSubsystem`, users must override two abstract methods to provide functionality that the class will use in its ordinary operation:

`getMeasurement()`

Java

C++

```
protected abstract double getMeasurement();
```

```
virtual double GetMeasurement() = 0;
```

The `getMeasurement` method returns the current measurement of the process variable. The `PIDSubsystem` will automatically call this method from its `periodic()` block, and pass its value to the control loop.

Users should override this method to return whatever sensor reading they wish to use as their process variable measurement.

useOutput()

Java

C++

```
protected abstract void useOutput(double output, double setpoint);
```

```
virtual void UseOutput(double output, double setpoint) = 0;
```

The `useOutput()` method consumes the output of the PID controller, and the current setpoint (which is often useful for computing a feedforward). The `PIDSubsystem` will automatically call this method from its `periodic()` block, and pass it the computed output of the control loop.

Users should override this method to pass the final computed control output to their subsystem's motors.

Passing In the Controller

Users must also pass in a `PIDController` to the `PIDSubsystem` base class through the superclass constructor call of their subclass. This serves to specify the PID gains, as well as the period (if the user is using a non-standard main robot loop period).

Additional modifications (e.g. enabling continuous input) can be made to the controller in the constructor body by calling `getController()`.

Using a PIDSubsystem

Once an instance of a `PIDSubsystem` subclass has been created, it can be used by commands through the following methods:

setSetpoint()

The `setSetpoint()` method can be used to set the setpoint of the `PIDSubsystem`. The subsystem will automatically track to the setpoint using the defined output:

Java

C++

```
// The subsystem will track to a setpoint of 5.  
examplePIDSubsystem.setSetpoint(5);
```

```
// The subsystem will track to a setpoint of 5.  
examplePIDSubsystem.SetSetpoint(5);
```

enable() and disable()

The `enable()` and `disable()` methods enable and disable the PID control of the `PIDSubsystem`. When the subsystem is enabled, it will automatically run the control loop and track the setpoint. When it is disabled, no control is performed.

Additionally, the `enable()` method resets the internal `PIDController`, and the `disable()` method calls the user-defined `useOutput()` method with both output and setpoint set to 0.

Full PIDSubsystem Example

What does a `PIDSubsystem` look like when used in practice? The following examples are taken from the FrisbeeBot example project (Java, C++):

Java

C++ (Header)

C++ (Source)

```

8  package edu.wpi.first.wpilibj.examples.frisbeebot.subsystems;
9
10 import edu.wpi.first.wpilibj.Encoder;
11 import edu.wpi.first.wpilibj.PWMVictorSPX;
12 import edu.wpi.first.wpilibj.controller.PIDController;
13 import edu.wpi.first.wpilibj.controller.SimpleMotorFeedforward;
14 import edu.wpi.first.wpilibj2.command.PIDSubsystem;
15
16 import edu.wpi.first.wpilibj.examples.frisbeebot.Constants.ShooterConstants;
17
18 public class ShooterSubsystem extends PIDSubsystem {
19     private final PWMVictorSPX m_shooterMotor = new PWMVictorSPX(ShooterConstants.
20 ↪ kShooterMotorPort);
21     private final PWMVictorSPX m_feederMotor = new PWMVictorSPX(ShooterConstants.
22 ↪ kFeederMotorPort);
23     private final Encoder m_shooterEncoder =
24 ↪ new Encoder(ShooterConstants.kEncoderPorts[0], ShooterConstants.
25 ↪ kEncoderPorts[1],
26 ↪ ShooterConstants.kEncoderReversed);
27     private final SimpleMotorFeedforward m_shooterFeedforward =
28 ↪ new SimpleMotorFeedforward(ShooterConstants.kSVolts,
29 ↪ ShooterConstants.kVVoltSecondsPerRotation);
30
31     /**
32     * The shooter subsystem for the robot.
33     */
34     public ShooterSubsystem() {
35 ↪ super(new PIDController(ShooterConstants.kP, ShooterConstants.kI,
36 ↪ ShooterConstants.kD));
37     getController().setTolerance(ShooterConstants.kShooterToleranceRPS);
38     m_shooterEncoder.setDistancePerPulse(ShooterConstants.kEncoderDistancePerPulse);
39     setSetpoint(ShooterConstants.kShooterTargetRPS);
40 }
41
42 @Override
43 public void useOutput(double output, double setpoint) {
44     m_shooterMotor.setVoltage(output + m_shooterFeedforward.calculate(setpoint));

```

(continues on next page)

(continued from previous page)

```

41     }
42
43     @Override
44     public double getMeasurement() {
45         return m_shooterEncoder.getRate();
46     }
47
48     public boolean atSetpoint() {
49         return m_controller.atSetpoint();
50     }
51
52     public void runFeeder() {
53         m_feederMotor.set(ShooterConstants.kFeederSpeed);
54     }
55
56     public void stopFeeder() {
57         m_feederMotor.set(0);
58     }
59 }

```

```

8  #pragma once
9
10 #include <frc/Encoder.h>
11 #include <frc/PWMVictorSPX.h>
12 #include <frc/controller/SimpleMotorFeedforward.h>
13 #include <frc2/command/PIDSubsystem.h>
14 #include <units/angle.h>
15
16 class ShooterSubsystem : public frc2::PIDSubsystem {
17 public:
18     ShooterSubsystem();
19
20     void UseOutput(double output, double setpoint) override;
21
22     double GetMeasurement() override;
23
24     bool AtSetpoint();
25
26     void RunFeeder();
27
28     void StopFeeder();
29
30 private:
31     frc::PWMVictorSPX m_shooterMotor;
32     frc::PWMVictorSPX m_feederMotor;
33     frc::Encoder m_shooterEncoder;
34     frc::SimpleMotorFeedforward<units::turns> m_shooterFeedforward;
35 };

```

```

8  #include "subsystems/ShooterSubsystem.h"
9
10 #include <frc/controller/PIDController.h>
11
12 #include "Constants.h"
13
14 using namespace ShooterConstants;

```

(continues on next page)

(continued from previous page)

```

15 ShooterSubsystem::ShooterSubsystem()
16   : PIDSubsystem(frc2::PIDController(kP, kI, kD)),
17     m_shooterMotor(kShooterMotorPort),
18     m_feederMotor(kFeederMotorPort),
19     m_shooterEncoder(kEncoderPorts[0], kEncoderPorts[1]),
20     m_shooterFeedforward(kS, kV) {
21   m_controller.SetTolerance(kShooterToleranceRPS.to<double>());
22   m_shooterEncoder.SetDistancePerPulse(kEncoderDistancePerPulse);
23   SetSetpoint(kShooterTargetRPS.to<double>());
24 }
25
26
27 void ShooterSubsystem::UseOutput(double output, double setpoint) {
28   m_shooterMotor.SetVoltage(units::volt_t(output) +
29                             m_shooterFeedforward.Calculate(kShooterTargetRPS));
30 }
31
32 bool ShooterSubsystem::AtSetpoint() { return m_controller.AtSetpoint(); }
33
34 double ShooterSubsystem::GetMeasurement() { return m_shooterEncoder.GetRate(); }
35
36 void ShooterSubsystem::RunFeeder() { m_feederMotor.Set(kFeederSpeed); }
37
38 void ShooterSubsystem::StopFeeder() { m_feederMotor.Set(0); }

```

Using a PIDSubsystem with commands can be very simple:

Java

C++ (Header)

C++ (Source)

```

85 // Spin up the shooter when the 'A' button is pressed
86 new JoystickButton(m_driverController, Button.kA.value)
87   .whenPressed(new InstantCommand(m_shooter::enable, m_shooter));
88
89 // Turn off the shooter when the 'B' button is pressed
90 new JoystickButton(m_driverController, Button.kB.value)
91   .whenPressed(new InstantCommand(m_shooter::disable, m_shooter));

```

```

73 frc2::InstantCommand m_spinUpShooter{[this] { m_shooter.Enable(); },
74                                       {&m_shooter}};
75
76 frc2::InstantCommand m_stopShooter{[this] { m_shooter.Disable(); },
77                                       {&m_shooter}};

```

```

32 // Spin up the shooter when the 'A' button is pressed
33 frc2::JoystickButton(&m_driverController, 1).WhenPressed(&m_spinUpShooter);
34
35 // Turn off the shooter when the 'B' button is pressed
36 frc2::JoystickButton(&m_driverController, 2).WhenPressed(&m_stopShooter);

```

29.11.2 PIDCommand

The `PIDCommand` class allows users to easily create commands with a built-in `PIDController`. As with `PIDSubsystem`, users can create a `PIDCommand` by subclassing the `PIDCommand` class. However, as with many of the other command classes in the command-based library, users may want to save code by defining a `PIDCommand` *inline*.

Creating a PIDCommand

A `PIDCommand` can be created two ways - by subclassing the `PIDCommand` class, or by defining the command *inline*. Both methods ultimately extremely similar, and ultimately the choice of which to use comes down to where the user desires that the relevant code be located.

In either case, a `PIDCommand` is created by passing the necessary parameters to its constructor (if defining a subclass, this can be done with a *super()* call):

Java

C++

```

29  /**
30   * Creates a new PIDCommand, which controls the given output with a PIDController.
31   *
32   * @param controller      the controller that controls the output.
33   * @param measurementSource the measurement of the process variable
34   * @param setpointSource   the controller's setpoint
35   * @param useOutput        the controller's output
36   * @param requirements     the subsystems required by this command
37   */
38  public PIDCommand(PIDController controller, DoubleSupplier measurementSource,
39                    DoubleSupplier setpointSource, DoubleConsumer useOutput,
40                    Subsystem... requirements) {
41      requireNonNullParam(controller, "controller", "SynchronousPIDCommand");
42      requireNonNullParam(measurementSource, "measurementSource", "SynchronousPIDCommand
43  ↪");
44      requireNonNullParam(setpointSource, "setpointSource", "SynchronousPIDCommand");
45      requireNonNullParam(useOutput, "useOutput", "SynchronousPIDCommand");
46
47      m_controller = controller;
48      m_useOutput = useOutput;
49      m_measurement = measurementSource;
50      m_setpoint = setpointSource;
51      m_requirements.addAll(Set.of(requirements));
52  }

```

```

29  /**
30   * Creates a new PIDCommand, which controls the given output with a
31   * PIDController.
32   *
33   * @param controller      the controller that controls the output.
34   * @param measurementSource the measurement of the process variable
35   * @param setpointSource   the controller's reference (aka setpoint)
36   * @param useOutput        the controller's output
37   * @param requirements     the subsystems required by this command
38   */
39  PIDCommand(PIDController controller,

```

(continues on next page)

(continued from previous page)

```
40     std::function<double()> measurementSource,  
41     std::function<double()> setpointSource,  
42     std::function<void(double)> useOutput,  
43     std::initializer_list<Subsystem*> requirements);
```

controller

The controller parameter is the PIDController object that will be used by the command. By passing this in, users can specify the PID gains and the period for the controller (if the user is using a nonstandard main robot loop period).

When subclassing PIDCommand, additional modifications (e.g. enabling continuous input) can be made to the controller in the constructor body by calling `getController()`.

measurementSource

The measurementSource parameter is a function (usually passed as a *lambda*) that returns the measurement of the process variable. Passing in the measurementSource function in PIDCommand is functionally analogous to overriding the *getMeasurement()* function in PIDSubsystem.

When subclassing PIDCommand, advanced users may further modify the measurement supplier by modifying the class's `m_measurement` field.

setpointSource

The setpointSource parameter is a function (usually passed as a *lambda*) that returns the current setpoint for the control loop. If only a constant setpoint is needed, an overload exists that takes a constant setpoint rather than a supplier.

When subclassing PIDCommand, advanced users may further modify the setpoint supplier by modifying the class's `m_setpoint` field.

useOutput

The useOutput parameter is a function (usually passed as a *lambda*) that consumes the output and setpoint of the control loop. Passing in the useOutput function in PIDCommand is functionally analogous to overriding the *useOutput()* function in PIDSubsystem.

When subclassing PIDCommand, advanced users may further modify the output consumer by modifying the class's `m_useOutput` field.

requirements

Like all inlineable commands, PIDCommand allows the user to specify its subsystem requirements as a constructor parameter.

Full PIDCommand Example

What does a PIDCommand look like when used in practice? The following examples are from the GyroDriveCommands example project (Java, C++):

Java

C++ (Header)

C++ (Source)

```
8 package edu.wpi.first.wpilibj.examples.gyrodrivecommands.commands;
9
10 import edu.wpi.first.wpilibj.controller.PIDController;
11 import edu.wpi.first.wpilibj2.command.PIDCommand;
12
13 import edu.wpi.first.wpilibj.examples.gyrodrivecommands.Constants.DriveConstants;
14 import edu.wpi.first.wpilibj.examples.gyrodrivecommands.subsystems.DriveSubsystem;
15
16 /**
17  * A command that will turn the robot to the specified angle.
18  */
19 public class TurnToAngle extends PIDCommand {
20     /**
21      * Turns to robot to the specified angle.
22      *
23      * @param targetAngleDegrees The angle to turn to
24      * @param drive               The drive subsystem to use
25      */
26     public TurnToAngle(double targetAngleDegrees, DriveSubsystem drive) {
27         super(
28             new PIDController(DriveConstants.kTurnP, DriveConstants.kTurnI,
29 ↪ DriveConstants.kTurnD),
30             // Close loop on heading
31             drive::getHeading,
32             // Set reference to target
33             targetAngleDegrees,
34             // Pipe output to turn robot
35             output -> drive.arcadeDrive(0, output),
36             // Require the drive
37             drive);
38
39         // Set the controller to be continuous (because it is an angle controller)
40         getController().enableContinuousInput(-180, 180);
41         // Set the controller tolerance - the delta tolerance ensures the robot is
42 ↪ stationary at the
43         // setpoint before it is considered as having reached the reference
44         getController()
45             .setTolerance(DriveConstants.kTurnToleranceDeg, DriveConstants.
46 ↪ kTurnRateToleranceDegPerS);
47     }
48 }
```

(continues on next page)

(continued from previous page)

```

46  @Override
47  public boolean isFinished() {
48      // End when the controller is at the reference.
49      return getController().atSetpoint();
50  }
51  }

```

```

8  #pragma once
9
10 #include <frc2/command/CommandHelper.h>
11 #include <frc2/command/PIDCommand.h>
12
13 #include "subsystems/DriveSubsystem.h"
14
15 /**
16  * A command that will turn the robot to the specified angle.
17  */
18 class TurnToAngle : public frc2::CommandHelper<frc2::PIDCommand, TurnToAngle> {
19 public:
20     /**
21      * Turns to robot to the specified angle.
22      *
23      * @param targetAngleDegrees The angle to turn to
24      * @param drive               The drive subsystem to use
25      */
26     TurnToAngle(units::degree_t target, DriveSubsystem* drive);
27
28     bool IsFinished() override;
29 };

```

```

8  #include "commands/TurnToAngle.h"
9
10 #include <frc/controller/PIDController.h>
11
12 using namespace DriveConstants;
13
14 TurnToAngle::TurnToAngle(units::degree_t target, DriveSubsystem* drive)
15     : CommandHelper(
16         frc2::PIDController(kTurnP, kTurnI, kTurnD),
17         // Close loop on heading
18         [drive] { return drive->GetHeading().to<double>(); },
19         // Set reference to target
20         target.to<double>(),
21         // Pipe output to turn robot
22         [drive](double output) { drive->ArcadeDrive(0, output); },
23         // Require the drive
24         {drive}) {
25     // Set the controller to be continuous (because it is an angle controller)
26     m_controller.EnableContinuousInput(-180, 180);
27     // Set the controller tolerance - the delta tolerance ensures the robot is
28     // stationary at the setpoint before it is considered as having reached the
29     // reference
30     m_controller.SetTolerance(kTurnTolerance.to<double>(),
31                             kTurnRateTolerance.to<double>());
32
33     AddRequirements({drive});

```

(continues on next page)

(continued from previous page)

```

34 }
35
36 bool TurnToAngle::IsFinished() { return GetController().AtSetpoint(); }

```

And, for an *inlined* example:

Java

C++

```

70 // Stabilize robot to drive straight with gyro when left bumper is held
71 new JoystickButton(m_driverController, Button.kBumperLeft.value).whenHeld(new
↳PIDCommand(
72     new PIDController(DriveConstants.kStabilizationP, DriveConstants.
↳kStabilizationI,
73         DriveConstants.kStabilizationD),
74     // Close the loop on the turn rate
75     m_robotDrive::getTurnRate,
76     // Setpoint is 0
77     0,
78     // Pipe the output to the turning controls
79     output -> m_robotDrive.arcadeDrive(m_driverController.getY(GenericHID.Hand.
↳kLeft), output),
80     // Require the robot drive
81     m_robotDrive));

```

```

37 // Stabilize robot to drive straight with gyro when left bumper is held
38 frc2::JoystickButton(&m_driverController, 5)
39 .WhenHeld(frc2::PIDCommand{
40     frc2::PIDController{dc::kStabilizationP, dc::kStabilizationI,
41         dc::kStabilizationD},
42     // Close the loop on the turn rate
43     [this] { return m_drive.GetTurnRate(); },
44     // Setpoint is 0
45     0,
46     // Pipe the output to the turning controls
47     [this](double output) {
48         m_drive.ArcadeDrive(m_driverController.GetY(
49             frc::GenericHID::JoystickHand::kLeftHand),
50             output);
51     },
52     // Require the robot drive
53     {&m_drive});

```

29.12 Motion Profiling through TrapezoidProfileSubsystems and TrapezoidProfileCommands

Note: For a description of the WPILib motion profiling features used by these command-based wrappers, see *Trapezoidal Motion Profiles in WPILib*.

Note: The TrapezoidProfile command wrappers are generally intended for composition

with custom or external controllers. For combining trapezoidal motion profiling with WPILib's PIDController, see [Combining Motion Profiling and PID in Command-Based](#).

When controlling a mechanism, it is often desirable to move it smoothly between two positions, rather than to abruptly change its setpoint. This is called “motion-profiling,” and is supported in WPILib through the TrapezoidProfile class ([Java](#), [C++](#)).

To further help teams integrate motion profiling into their command-based robot projects, WPILib includes two convenience wrappers for the TrapezoidProfile class: TrapezoidProfileSubsystem, which automatically generates and executes motion profiles in its periodic() method, and the TrapezoidProfileCommand, which executes a single user-provided TrapezoidProfile.

29.12.1 TrapezoidProfileSubsystem

Note: In C++, the TrapezoidProfileSubsystem class is templated on the unit type used for distance measurements, which may be angular or linear. The passed-in values *must* have units consistent with the distance units, or a compile-time error will be thrown. For more information on C++ units, see [The C++ Units Library](#).

The TrapezoidProfileSubsystem class ([Java](#), [C++](#)) will automatically create and execute trapezoidal motion profiles to reach the user-provided goal state. To use the TrapezoidProfileSubsystem class, users must create a subclass of it.

Creating a TrapezoidProfileSubsystem

When subclassing TrapezoidProfileSubsystem, users must override a single abstract method to provide functionality that the class will use in its ordinary operation:

useState()

Java

C++

```
protected abstract void useState(TrapezoidProfile.State state);
```

```
virtual void UseState(frc::TrapezoidProfile<Distance>::State state) = 0;
```

The useState() method consumes the current state of the motion profile. The TrapezoidProfileSubsystem will automatically call this method from its periodic() block, and pass it the motion profile state corresponding to the subsystem's current progress through the motion profile.

Users may do whatever they want with this state; a typical use case (as shown in the [Full TrapezoidProfileSubsystem Example](#)) is to use the state to obtain a setpoint and a feedforward for an external “smart” motor controller.

Constructor Parameters

Users must pass in a set of `TrapezoidProfile.Constraints` to the `TrapezoidProfileSubsystem` base class through the superclass constructor call of their subclass. This serves to constrain the automatically-generated profiles to a given maximum velocity and acceleration.

Users must also pass in an initial position for the mechanism.

Advanced users may pass in an alternate value for the loop period, if a non-standard main loop period is being used.

Using a TrapezoidProfileSubsystem

Once an instance of a `TrapezoidProfileSubsystem` subclass has been created, it can be used by commands through the following methods:

setGoal()

Note: If you wish to set the goal to a simple distance with an implicit target velocity of zero, an overload of `setGoal()` exists that takes a single distance value, rather than a full motion profile state.

The `setGoal()` method can be used to set the goal state of the `TrapezoidProfileSubsystem`. The subsystem will automatically execute a profile to the goal, passing the current state at each iteration to the provided `useState()` method.

Java

C++

```
// The subsystem will execute a profile to a position of 5 and a velocity of 3.
examplePIDSubsystem.setGoal(new TrapezoidProfile.Goal(5, 3);
```

```
// The subsystem will execute a profile to a position of 5 meters and a velocity of 3_mps.
examplePIDSubsystem.SetGoal({5_m, 3_mps});
```

Full TrapezoidProfileSubsystem Example

What does a `TrapezoidProfileSubsystem` look like when used in practice? The following examples are taken from the `ArmBotOffboard` example project (Java, C++):

Java

C++ (Header)

C++ (Source)

```
8 package edu.wpi.first.wpilibj.examples.armbotoffboard.subsystems;
9
10 import edu.wpi.first.wpilibj.controller.ArmFeedforward;
11 import edu.wpi.first.wpilibj.trajectory.TrapezoidProfile;
```

(continues on next page)

(continued from previous page)

```

12 import edu.wpi.first.wpilibj2.command.TrapezoidProfileSubsystem;
13
14 import edu.wpi.first.wpilibj.examples.armbotoffboard.Constants.ArmConstants;
15 import edu.wpi.first.wpilibj.examples.armbotoffboard.ExampleSmartMotorController;
16
17 /**
18  * A robot arm subsystem that moves with a motion profile.
19  */
20 public class ArmSubsystem extends TrapezoidProfileSubsystem {
21     private final ExampleSmartMotorController m_motor =
22         new ExampleSmartMotorController(ArmConstants.kMotorPort);
23     private final ArmFeedforward m_feedforward =
24         new ArmFeedforward(ArmConstants.kSVolts, ArmConstants.kCosVolts,
25             ArmConstants.kVVoltSecondPerRad, ArmConstants.
26             ↪kAVoltSecondSquaredPerRad);
27
28     /**
29      * Create a new ArmSubsystem.
30      */
31     public ArmSubsystem() {
32         super(new TrapezoidProfile.Constraints(ArmConstants.kMaxVelocityRadPerSecond,
33             ArmConstants.
34             ↪kMaxAccelerationRadPerSecSquared),
35             ArmConstants.kArmOffsetRads);
36         m_motor.setPID(ArmConstants.kP, 0, 0);
37     }
38
39     @Override
40     public void useState(TrapezoidProfile.State setpoint) {
41         // Calculate the feedforward from the sepoint
42         double feedforward = m_feedforward.calculate(setpoint.position, setpoint.
43         ↪velocity);
44         // Add the feedforward to the PID output to get the motor output
45         m_motor.setSetpoint(ExampleSmartMotorController.PIDMode.kPosition, setpoint.
46         ↪position,
47             feedforward / 12.0);
48     }
49 }

```

```

8 #pragma once
9
10 #include <frc/controller/ArmFeedforward.h>
11 #include <frc2/command/TrapezoidProfileSubsystem.h>
12 #include <units/angle.h>
13
14 #include "ExampleSmartMotorController.h"
15
16 /**
17  * A robot arm subsystem that moves with a motion profile.
18  */
19 class ArmSubsystem : public frc2::TrapezoidProfileSubsystem<units::radians> {
20     using State = frc::TrapezoidProfile<units::radians>::State;
21
22     public:
23         ArmSubsystem();
24

```

(continues on next page)

(continued from previous page)

```

25 void UseState(State setpoint) override;
26
27 private:
28     ExampleSmartMotorController m_motor;
29     frc::ArmFeedforward m_feedforward;
30 };

8  #include "subsystems/ArmSubsystem.h"
9
10 #include "Constants.h"
11
12 using namespace ArmConstants;
13 using State = frc::TrapezoidProfile<units::radians>::State;
14
15 ArmSubsystem::ArmSubsystem()
16     : frc2::TrapezoidProfileSubsystem<units::radians>(
17         {kMaxVelocity, kMaxAcceleration}, kArmOffset),
18         m_motor(kMotorPort),
19         m_feedforward(kS, kCos, kV, kA) {
20     m_motor.SetPID(kP, 0, 0);
21 }
22
23 void ArmSubsystem::UseState(State setpoint) {
24     // Calculate the feedforward from the sepoint
25     units::volt_t feedforward =
26         m_feedforward.Calculate(setpoint.position, setpoint.velocity);
27     // Add the feedforward to the PID output to get the motor output
28     m_motor.SetSetpoint(ExampleSmartMotorController::PIDMode::kPosition,
29         setpoint.position.to<double>(), feedforward / 12_V);
30 }

```

Using a TrapezoidProfileSubsystem with commands can be quite simple:

Java

C++

```

63 // Move the arm to 2 radians above horizontal when the 'A' button is pressed.
64 new JoystickButton(m_driverController, Button.kA.value)
65     .whenPressed(() -> m_robotArm.setGoal(2), m_robotArm);
66
67 // Move the arm to neutral position when the 'B' button is pressed.
68 new JoystickButton(m_driverController, Button.kB.value)
69     .whenPressed(() -> m_robotArm.setGoal(Constants.ArmConstants.kArmOffsetRads),
70     m_robotArm);

```

```

33 // Move the arm to 2 radians above horizontal when the 'A' button is pressed.
34 frc2::JoystickButton(&m_driverController, 1)
35     .WhenPressed([this] { m_arm.SetGoal(2_rad); }, {&m_arm});
36
37 // Move the arm to neutral position when the 'B' button is pressed.
38 frc2::JoystickButton(&m_driverController, 1)
39     .WhenPressed([this] { m_arm.SetGoal(ArmConstants::kArmOffset); },
40     {&m_arm});

```


29.12.2 TrapezoidProfileCommand

Note: In C++, the `TrapezoidProfileCommand` class is templated on the unit type used for distance measurements, which may be angular or linear. The passed-in values *must* have units consistent with the distance units, or a compile-time error will be thrown. For more information on C++ units, see *The C++ Units Library*.

The `TrapezoidProfileCommand` class (Java, C++) allows users to create a command that will execute a single `TrapezoidProfile`, passing its current state at each iteration to a user-defined function.

As with `TrapezoidProfileSubsystem`, users can create a `TrapezoidProfileCommand` by subclassing the `TrapezoidProfileCommand` class. However, as with many of the other command classes in the command-based library, users may want to save code by defining a `TrapezoidProfileCommand` *inline*.

Creating a TrapezoidProfileCommand

A `TrapezoidProfileCommand` can be created two ways - by subclassing the `TrapezoidProfileCommand` class, or by defining the command *inline*. Both methods ultimately extremely similar, and ultimately the choice of which to use comes down to where the user desires that the relevant code be located.

In either case, a `TrapezoidProfileCommand` is created by passing the necessary parameters to its constructor (if defining a subclass, this can be done with a *super()* call):

Java

C++

```

28  /**
29   * Creates a new TrapezoidProfileCommand that will execute the given {@link
    ↪TrapezoidProfile}.
30   * Output will be piped to the provided consumer function.
31   *
32   * @param profile      The motion profile to execute.
33   * @param output       The consumer for the profile output.
34   * @param requirements The subsystems required by this command.
35   */
36  public TrapezoidProfileCommand(TrapezoidProfile profile,
37                                Consumer<State> output,
38                                Subsystem... requirements) {
39      m_profile = requireNonNullParam(profile, "profile", "TrapezoidProfileCommand");
40      m_output = requireNonNullParam(output, "output", "TrapezoidProfileCommand");
41      addRequirements(requirements);
42  }

```

```

36  public:
37  /**
38   * Creates a new TrapezoidProfileCommand that will execute the given
39   * TrapezoidalProfile. Output will be piped to the provided consumer function.
40   *
41   * @param profile The motion profile to execute.
42   * @param output  The consumer for the profile output.

```

(continues on next page)

(continued from previous page)

```

43  */
44  TrapezoidProfileCommand(frc::TrapezoidProfile<Distance> profile,
45                          std::function<void(State)> output,
46                          std::initializer_list<Subsystem*> requirements)
47      : m_profile(profile), m_output(output) {
48      this->AddRequirements(requirements);
49  }

```

profile

The profile parameter is the TrapezoidProfile object that will be executed by the command. By passing this in, users specify the start state, end state, and motion constraints of the profile that the command will use.

output

The output parameter is a function (usually passed as a *lambda*) that consumes the output and setpoint of the control loop. Passing in the useOutput function in PIDCommand is functionally analogous to overriding the *useState()* function in PIDSubsystem.

requirements

Like all inlineable commands, TrapezoidProfileCommand allows the user to specify its subsystem requirements as a constructor parameter.

Full TrapezoidProfileCommand Example

What does a TrapezoidProfileSubsystem look like when used in practice? The following examples are taken from the DriveDistanceOffboard example project ([Java](#), [C++](#)):

Java

C++ (Header)

C++ (Source)

```

8  package edu.wpi.first.wpilibj.examples.drivedistanceoffboard.commands;
9
10 import edu.wpi.first.wpilibj.trajectory.TrapezoidProfile;
11 import edu.wpi.first.wpilibj2.command.TrapezoidProfileCommand;
12
13 import edu.wpi.first.wpilibj.examples.drivedistanceoffboard.Constants.DriveConstants;
14 import edu.wpi.first.wpilibj.examples.drivedistanceoffboard.subsystems.DriveSubsystem;
15
16 /**
17  * Drives a set distance using a motion profile.
18  */
19 public class DriveDistanceProfiled extends TrapezoidProfileCommand {
20     /**
21      * Creates a new DriveDistanceProfiled command.

```

(continues on next page)

(continued from previous page)

```

22  *
23  * @param meters The distance to drive.
24  * @param drive The drive subsystem to use.
25  */
26  public DriveDistanceProfiled(double meters, DriveSubsystem drive) {
27      super(
28          new TrapezoidProfile(
29              // Limit the max acceleration and velocity
30              new TrapezoidProfile.Constraints(DriveConstants.kMaxSpeedMetersPerSecond,
31              ↪kMaxAccelerationMetersPerSecondSquared),
32              // End at desired position in meters; implicitly starts at 0
33              new TrapezoidProfile.State(meters, 0)),
34              // Pipe the profile state to the drive
35              setpointState -> drive.setDriveStates(setpointState, setpointState),
36              // Require the drive
37              drive);
38      // Reset drive encoders since we're starting at 0
39      drive.resetEncoders();
40  }
41  }

```

```

8  #pragma once
9
10 #include <frc2/command/CommandHelper.h>
11 #include <frc2/command/TrapezoidProfileCommand.h>
12
13 #include "subsystems/DriveSubsystem.h"
14
15 class DriveDistanceProfiled
16     : public frc2::CommandHelper<frc2::TrapezoidProfileCommand<units::meters>,
17     DriveDistanceProfiled> {
18 public:
19     DriveDistanceProfiled(units::meter_t distance, DriveSubsystem* drive);
20 };

```

```

8  #include "commands/DriveDistanceProfiled.h"
9
10 #include "Constants.h"
11
12 using namespace DriveConstants;
13
14 DriveDistanceProfiled::DriveDistanceProfiled(units::meter_t distance,
15                                             DriveSubsystem* drive)
16     : CommandHelper(
17         frc::TrapezoidProfile<units::meters>(
18             // Limit the max acceleration and velocity
19             {kMaxSpeed, kMaxAcceleration},
20             // End at desired position in meters; implicitly starts at 0
21             {distance, 0_mps}),
22         // Pipe the profile state to the drive
23         [drive](auto setpointState) {
24             drive->SetDriveStates(setpointState, setpointState);
25         },
26         // Require the drive
27         {drive}) {

```

(continues on next page)

(continued from previous page)

```

28 // Reset drive encoders since we're starting at 0
29 drive->ResetEncoders();
30 }

```

And, for an *inlined* example:

Java

C++

```

68 // Do the same thing as above when the 'B' button is pressed, but defined inline
69 new JoystickButton(m_driverController, Button.kB.value)
70   .whenPressed(
71     new TrapezoidProfileCommand(
72       new TrapezoidProfile(
73         // Limit the max acceleration and velocity
74         new TrapezoidProfile.Constraints(
75           DriveConstants.kMaxSpeedMetersPerSecond,
76           DriveConstants.kMaxAccelerationMetersPerSecondSquared),
77         // End at desired position in meters; implicitly starts at 0
78         new TrapezoidProfile.State(3, 0)),
79       // Pipe the profile state to the drive
80       setpointState -> m_robotDrive.setDriveStates(
81         setpointState,
82         setpointState),
83       // Require the drive
84       m_robotDrive)
85       .beforeStarting(m_robotDrive::resetEncoders)
86       .withTimeout(10));

```

```

44 // Do the same thing as above when the 'B' button is pressed, but defined
45 // inline
46 frc2::JoystickButton(&m_driverController, 2)
47   .WhenPressed(
48     frc2::TrapezoidProfileCommand<units::meters>(
49       frc::TrapezoidProfile<units::meters>(
50         // Limit the max acceleration and velocity
51         {DriveConstants::kMaxSpeed, DriveConstants::kMaxAcceleration},
52         // End at desired position in meters; implicitly starts at 0
53         {3_m, 0_mps}),
54       // Pipe the profile state to the drive
55       [this](auto setpointState) {
56         m_drive.SetDriveStates(setpointState, setpointState);
57       },
58       // Require the drive
59       {&m_drive})
60       .BeforeStarting([this]() { m_drive.ResetEncoders(); })
61       .WithTimeout(10_s));
62 }

```

29.13 Combining Motion Profiling and PID in Command-Based

Note: For a description of the WPILib PID control features used by these command-based wrappers, see [PID Control in WPILib](#).

Note: Unlike the earlier version of `PIDController`, the 2020 `ProfiledPIDController` class runs *synchronously*, and is not handled in its own thread. Accordingly, changing its period parameter will *not* change the actual frequency at which it runs in any of these wrapper classes. Users should never modify the period parameter unless they are certain of what they are doing.

A common FRC® controls solution is to pair a trapezoidal motion profile for setpoint generation with a PID controller for setpoint tracking. To facilitate this, WPILib includes its own [ProfiledPIDController](#) class. To further aid teams in integrating this functionality into their robots, the command-based framework contains two convenience wrappers for the `ProfiledPIDController` class: `ProfiledPIDSubsystem`, which integrates the controller into a subsystem, and `ProfiledPIDCommand`, which integrates the controller into a command.

29.13.1 ProfiledPIDSubsystem

Note: In C++, the `ProfiledPIDSubsystem` class is templated on the unit type used for distance measurements, which may be angular or linear. The passed-in values *must* have units consistent with the distance units, or a compile-time error will be thrown. For more information on C++ units, see [The C++ Units Library](#).

The `ProfiledPIDSubsystem` class (Java, C++) allows users to conveniently create a subsystem with a built-in `PIDController`. In order to use the `ProfiledPIDSubsystem` class, users must create a subclass of it.

Creating a ProfiledPIDSubsystem

When subclassing `ProfiledPIDSubsystem`, users must override two abstract methods to provide functionality that the class will use in its ordinary operation:

getMeasurement()

Java

C++

```
protected abstract double getMeasurement();
```

```
virtual double GetMeasurement() = 0;
```

The `getMeasurement` method returns the current measurement of the process variable. The `PIDSubsystem` will automatically call this method from its `periodic()` block, and pass its value to the control loop.

Users should override this method to return whatever sensor reading they wish to use as their process variable measurement.

`useOutput()`

Java

C++

```
protected abstract void useOutput(double output, TrapezoidProfile.State setpoint);
```

```
virtual void UseOutput(double output, frc::TrapezoidProfile<Distance>::State_u  
↪ setpoint) = 0;
```

The `useOutput()` method consumes the output of the PID controller, and the current setpoint state (which is often useful for computing a feedforward). The `PIDSubsystem` will automatically call this method from its `periodic()` block, and pass it the computed output of the control loop.

Users should override this method to pass the final computed control output to their subsystem's motors.

Passing In the Controller

Users must also pass in a `ProfiledPIDController` to the `ProfiledPIDSubsystem` base class through the superclass constructor call of their subclass. This serves to specify the PID gains, the motion profile constraints, and the period (if the user is using a non-standard main robot loop period).

Additional modifications (e.g. enabling continuous input) can be made to the controller in the constructor body by calling `getController()`.

Using a ProfiledPIDSubsystem

Once an instance of a `PIDSubsystem` subclass has been created, it can be used by commands through the following methods:

`setGoal()`

Note: If you wish to set the goal to a simple distance with an implicit target velocity of zero, an overload of `setGoal()` exists that takes a single distance value, rather than a full motion profile state.

The `setGoal()` method can be used to set the setpoint of the `PIDSubsystem`. The subsystem will automatically track to the setpoint using the defined output:

Java

C++

```
// The subsystem will track to a goal of 5 meters and velocity of 3 meters per second.
examplePIDSubsystem.setGoal(5, 3);
```

```
// The subsystem will track to a goal of 5 meters and velocity of 3 meters per second.
examplePIDSubsystem.SetGoal({5_m, 3_mps});
```

enable() and disable()

The `enable()` and `disable()` methods enable and disable the automatic control of the `ProfiledPIDSubsystem`. When the subsystem is enabled, it will automatically run the motion profile and the control loop and track to the goal. When it is disabled, no control is performed.

Additionally, the `enable()` method resets the internal `ProfiledPIDController`, and the `disable()` method calls the user-defined `useOutput()` method with both output and setpoint set to 0.

Full ProfiledPIDSubsystem Example

What does a `PIDSubsystem` look like when used in practice? The following examples are taken from the `ArmBot` example project (Java, C++):

Java

C++ (Header)

C++ (Source)

```
8 package edu.wpi.first.wpilibj.examples.armbot.subsystems;
9
10 import edu.wpi.first.wpilibj.Encoder;
11 import edu.wpi.first.wpilibj.PWMVictorSPX;
12 import edu.wpi.first.wpilibj.controller.ArmFeedforward;
13 import edu.wpi.first.wpilibj.controller.ProfiledPIDController;
14 import edu.wpi.first.wpilibj.traj.TrapezoidProfile;
15 import edu.wpi.first.wpilibj2.command.ProfiledPIDSubsystem;
16
17 import edu.wpi.first.wpilibj.examples.armbot.Constants.ArmConstants;
18
19 /**
20  * A robot arm subsystem that moves with a motion profile.
21  */
22 public class ArmSubsystem extends ProfiledPIDSubsystem {
23     private final PWMVictorSPX m_motor = new PWMVictorSPX(ArmConstants.kMotorPort);
24     private final Encoder m_encoder =
25         new Encoder(ArmConstants.kEncoderPorts[0], ArmConstants.kEncoderPorts[1]);
26     private final ArmFeedforward m_feedforward =
27         new ArmFeedforward(ArmConstants.kSVolts, ArmConstants.kCosVolts,
28             ArmConstants.kVVoltSecondPerRad, ArmConstants.
29     ↪ kAVoltSecondSquaredPerRad);
```

(continues on next page)

(continued from previous page)

```

30  /**
31   * Create a new ArmSubsystem.
32   */
33  public ArmSubsystem() {
34      super(new ProfiledPIDController(ArmConstants.kP, 0, 0, new TrapezoidProfile.
↳ Constraints(
35          ArmConstants.kMaxVelocityRadPerSecond, ArmConstants.
↳ kMaxAccelerationRadPerSecSquared)), 0);
36      m_encoder.setDistancePerPulse(ArmConstants.kEncoderDistancePerPulse);
37      // Start arm at rest in neutral position
38      setGoal(ArmConstants.kArmOffsetRads);
39  }
40
41  @Override
42  public void useOutput(double output, TrapezoidProfile.State setpoint) {
43      // Calculate the feedforward from the sepoint
44      double feedforward = m_feedforward.calculate(setpoint.position, setpoint.
↳ velocity);
45      // Add the feedforward to the PID output to get the motor output
46      m_motor.setVoltage(output + feedforward);
47  }
48
49  @Override
50  public double getMeasurement() {
51      return m_encoder.getDistance() + ArmConstants.kArmOffsetRads;
52  }
53  }

```

```

8  #pragma once
9
10 #include <frc/Encoder.h>
11 #include <frc/PWMVictorSPX.h>
12 #include <frc/controller/ArmFeedforward.h>
13 #include <frc2/command/ProfiledPIDSubsystem.h>
14 #include <units/angle.h>
15
16 /**
17  * A robot arm subsystem that moves with a motion profile.
18  */
19 class ArmSubsystem : public frc2::ProfiledPIDSubsystem<units::radians> {
20     using State = frc::TrapezoidProfile<units::radians>::State;
21
22     public:
23         ArmSubsystem();
24
25         void UseOutput(double output, State setpoint) override;
26
27         units::radian_t GetMeasurement() override;
28
29     private:
30         frc::PWMVictorSPX m_motor;
31         frc::Encoder m_encoder;
32         frc::ArmFeedforward m_feedforward;
33 };

```



```

8  #include "subsystems/ArmSubsystem.h"
9
10 #include "Constants.h"
11
12 using namespace ArmConstants;
13 using State = frc::TrapezoidProfile<units::radians>::State;
14
15 ArmSubsystem::ArmSubsystem()
16 : frc2::ProfiledPIDSubsystem<units::radians>(
17     frc::ProfiledPIDController<units::radians>(
18         kP, 0, 0, {kMaxVelocity, kMaxAcceleration})),
19     m_motor(kMotorPort),
20     m_encoder(kEncoderPorts[0], kEncoderPorts[1]),
21     m_feedforward(kS, kCos, kV, kA) {
22     m_encoder.SetDistancePerPulse(kEncoderDistancePerPulse.to<double>());
23     // Start arm in neutral position
24     SetGoal(State{kArmOffset, 0_rad_per_s});
25 }
26
27 void ArmSubsystem::UseOutput(double output, State setpoint) {
28     // Calculate the feedforward from the setpoint
29     units::volt_t feedforward =
30         m_feedforward.Calculate(setpoint.position, setpoint.velocity);
31     // Add the feedforward to the PID output to get the motor output
32     m_motor.SetVoltage(units::volt_t(output) + feedforward);
33 }
34
35 units::radian_t ArmSubsystem::GetMeasurement() {
36     return units::radian_t(m_encoder.GetDistance()) + kArmOffset;
37 }

```

Using a ProfiledPIDSubsystem with commands can be very simple:

Java

C++

```

63 // Move the arm to 2 radians above horizontal when the 'A' button is pressed.
64 new JoystickButton(m_driverController, Button.kA.value)
65     .whenPressed(() -> {
66         m_robotArm.setGoal(2);
67         m_robotArm.enable();
68     }, m_robotArm);

```

```

33 // Move the arm to 2 radians above horizontal when the 'A' button is pressed.
34 frc2::JoystickButton(&m_driverController, 1)
35     .WhenPressed(
36         [this] {
37             m_arm.SetGoal(2_rad);
38             m_arm.Enable();
39         },
40         {&m_arm});

```

29.13.2 ProfiledPIDCommand

Note: In C++, the ProfiledPIDCommand class is templated on the unit type used for distance measurements, which may be angular or linear. The passed-in values *must* have units consistent with the distance units, or a compile-time error will be thrown. For more information on C++ units, see [The C++ Units Library](#).

The ProfiledPIDCommand class (Java, C++) allows users to easily create commands with a built-in ProfiledPIDController. As with ProfiledPIDSubsystem, users can create a ProfiledPIDCommand by subclassing the ProfiledPIDCommand class. However, as with many of the other command classes in the command-based library, users may want to save code by defining it *inline*.

Creating a PIDCommand

A ProfiledPIDCommand can be created two ways - by subclassing the ProfiledPIDCommand class, or by defining the command *inline*. Both methods ultimately extremely similar, and ultimately the choice of which to use comes down to where the user desires that the relevant code be located.

In either case, a ProfiledPIDCommand is created by passing the necessary parameters to its constructor (if defining a subclass, this can be done with a *super()* call):

Java

C++

```

32  /**
33   * Creates a new PIDCommand, which controls the given output with a
  ↪ ProfiledPIDController.
34   * Goal velocity is specified.
35   *
36   * @param controller      the controller that controls the output.
37   * @param measurementSource the measurement of the process variable
38   * @param goalSource       the controller's goal
39   * @param useOutput        the controller's output
40   * @param requirements     the subsystems required by this command
41   */
42  public ProfiledPIDCommand(ProfiledPIDController controller, DoubleSupplier
  ↪ measurementSource,
43                           Supplier<State> goalSource, BiConsumer<Double, State>
  ↪ useOutput,
44                           Subsystem... requirements) {
45      requireNonNullParam(controller, "controller", "SynchronousPIDCommand");
46      requireNonNullParam(measurementSource, "measurementSource", "SynchronousPIDCommand
  ↪ ");
47      requireNonNullParam(goalSource, "goalSource", "SynchronousPIDCommand");
48      requireNonNullParam(useOutput, "useOutput", "SynchronousPIDCommand");
49
50      m_controller = controller;
51      m_useOutput = useOutput;
52      m_measurement = measurementSource;
53      m_goal = goalSource;
54      m_requirements.addAll(Set.of(requirements));
55  }

```

```

39 public:
40     /**
41      * Creates a new PIDCommand, which controls the given output with a
42      * ProfiledPIDController.
43      *
44      * @param controller      the controller that controls the output.
45      * @param measurementSource the measurement of the process variable
46      * @param goalSource      the controller's goal
47      * @param useOutput        the controller's output
48      * @param requirements     the subsystems required by this command
49      */
50     ProfiledPIDCommand(frc::ProfiledPIDController<Distance> controller,
51                       std::function<Distance_t()> measurementSource,
52                       std::function<State()> goalSource,
53                       std::function<void(double, State)> useOutput,
54                       std::initializer_list<Subsystem*> requirements)
55     : m_controller{controller},
56       m_measurement{std::move(measurementSource)},
57       m_goal{std::move(goalSource)},
58       m_useOutput{std::move(useOutput)} {
59         this->AddRequirements(requirements);
60     }

```

controller

The controller parameter is the ProfiledPIDController object that will be used by the command. By passing this in, users can specify the PID gains, the motion profile constraints, and the period for the controller (if the user is using a nonstandard main robot loop period).

When subclassing ProfiledPIDCommand, additional modifications (e.g. enabling continuous input) can be made to the controller in the constructor body by calling `getController()`.

measurementSource

The measurementSource parameter is a function (usually passed as a *lambda*) that returns the measurement of the process variable. Passing in the measurementSource function in ProfiledPIDCommand is functionally analogous to overriding the *getMeasurement()* function in ProfiledPIDSubsystem.

When subclassing ProfiledPIDCommand, advanced users may further modify the measurement supplier by modifying the class's `m_measurement` field.

goalSource

The goalSource parameter is a function (usually passed as a *lambda*) that returns the current goal state for the mechanism. If only a constant goal is needed, an overload exists that takes a constant goal rather than a supplier. Additionally, if goal velocities are desired to be zero, overloads exist that take a constant distance rather than a full profile state.

When subclassing ProfiledPIDCommand, advanced users may further modify the setpoint supplier by modifying the class's `m_goal` field.

useOutput

The `useOutput` parameter is a function (usually passed as a *lambda*) that consumes the output and setpoint state of the control loop. Passing in the `useOutput` function in `ProfiledPIDCommand` is functionally analogous to overriding the *useOutput()* function in `ProfiledPIDSubsystem`.

When subclassing `ProfiledPIDCommand`, advanced users may further modify the output consumer by modifying the class's `m_useOutput` field.

requirements

Like all inlineable commands, `ProfiledPIDCommand` allows the user to specify its subsystem requirements as a constructor parameter.

Full ProfiledPIDCommand Example

What does a `ProfiledPIDCommand` look like when used in practice? The following examples are from the `GyroDriveCommands` example project (Java, C++):

Java

C++ (Header)

C++ (Source)

```

8  package edu.wpi.first.wpilibj.examples.gyrodrivecommands.commands;
9
10 import edu.wpi.first.wpilibj.controller.ProfiledPIDController;
11 import edu.wpi.first.wpilibj.trajectory.TrapezoidProfile;
12 import edu.wpi.first.wpilibj2.command.ProfiledPIDCommand;
13
14 import edu.wpi.first.wpilibj.examples.gyrodrivecommands.Constants.DriveConstants;
15 import edu.wpi.first.wpilibj.examples.gyrodrivecommands.subsystems.DriveSubsystem;
16
17 /**
18  * A command that will turn the robot to the specified angle using a motion profile.
19  */
20 public class TurnToAngleProfiled extends ProfiledPIDCommand {
21     /**
22      * Turns to robot to the specified angle using a motion profile.
23      *
24      * @param targetAngleDegrees The angle to turn to
25      * @param drive               The drive subsystem to use
26      */
27     public TurnToAngleProfiled(double targetAngleDegrees, DriveSubsystem drive) {
28         super(
29             new ProfiledPIDController(DriveConstants.kTurnP, DriveConstants.kTurnI,
30                                     DriveConstants.kTurnD, new TrapezoidProfile.
31 ↪ Constraints(
32                 DriveConstants.kMaxTurnRateDegPerS,
33                 DriveConstants.kMaxTurnAccelerationDegPerSSquared)),
34             // Close loop on heading
35             drive::getHeading,
36             // Set reference to target

```

(continues on next page)

(continued from previous page)

```

36     targetAngleDegrees,
37     // Pipe output to turn robot
38     (output, setpoint) -> drive.arcadeDrive(0, output),
39     // Require the drive
40     drive);
41
42     // Set the controller to be continuous (because it is an angle controller)
43     getController().enableContinuousInput(-180, 180);
44     // Set the controller tolerance - the delta tolerance ensures the robot is
45     ↪ stationary at the
46     // setpoint before it is considered as having reached the reference
47     getController()
48     ↪ .setTolerance(DriveConstants.kTurnToleranceDeg, DriveConstants.
49     ↪ kTurnRateToleranceDegPerS);
50 }
51
52 @Override
53 public boolean isFinished() {
54     // End when the controller is at the reference.
55     return getController().atGoal();
56 }
57 }

```

```

8  #pragma once
9
10 #include <frc2/command/CommandHelper.h>
11 #include <frc2/command/ProfiledPIDCommand.h>
12
13 #include "subsystems/DriveSubsystem.h"
14
15 /**
16  * A command that will turn the robot to the specified angle using a motion
17  * profile.
18  */
19 class TurnToAngleProfiled
20     : public frc2::CommandHelper<frc2::ProfiledPIDCommand<units::radians>,
21     TurnToAngleProfiled> {
22 public:
23     /**
24      * Turns to robot to the specified angle using a motion profile.
25      *
26      * @param targetAngleDegrees The angle to turn to
27      * @param drive               The drive subsystem to use
28      */
29     TurnToAngleProfiled(units::degree_t targetAngleDegrees,
30     DriveSubsystem* drive);
31
32     bool IsFinished() override;
33 };

```

```

8  #include "commands/TurnToAngleProfiled.h"
9
10 #include <frc/controller/ProfiledPIDController.h>
11
12 using namespace DriveConstants;
13

```

(continues on next page)

(continued from previous page)

```

14 TurnToAngleProfiled::TurnToAngleProfiled(units::degree_t target,
15                                         DriveSubsystem* drive)
16     : CommandHelper(
17         frc::ProfiledPIDController<units::radians>(
18             kTurnP, kTurnI, kTurnD, {kMaxTurnRate, kMaxTurnAcceleration}),
19         // Close loop on heading
20         [drive] { return drive->GetHeading(); },
21         // Set reference to target
22         target,
23         // Pipe output to turn robot
24         [drive](double output, auto setpointState) {
25             drive->ArcadeDrive(0, output);
26         },
27         // Require the drive
28         {drive}) {
29     // Set the controller to be continuous (because it is an angle controller)
30     GetController().EnableContinuousInput(-180_deg, 180_deg);
31     // Set the controller tolerance - the delta tolerance ensures the robot is
32     // stationary at the setpoint before it is considered as having reached the
33     // reference
34     GetController().SetTolerance(kTurnTolerance, kTurnRateTolerance);
35
36     AddRequirements({drive});
37 }
38
39 bool TurnToAngleProfiled::IsFinished() { return GetController().AtGoal(); }

```

29.14 A Technical Discussion on C++ Commands

Important: This article serves as a technical discussion on some of the design decisions that were made when designing the new command-based framework in C++. You do not need to understand the information within this article to use the command-based framework in your robot code.

Note: This article assumes that you have a fair understanding of advanced C++ concepts, including templates, smart pointers, inheritance, rvalue references, copy semantics, move semantics, and CRTP.

This article will help you understand the reasoning behind some of the decisions made in the new command-based framework (such as the use of `std::unique_ptr`, CRTP in the form of `CommandHelper<Base, Derived>`, the lack of more advanced decorators that are available in Java, etc.)

29.14.1 Ownership Model

The old command-based framework employed the use of raw pointers, meaning that users had to use `new` (resulting in manual heap allocations) in their robot code. Since there was no clear indication on who owned the commands (the scheduler, the command groups, or the user themselves), it was not apparent who was supposed to take care of freeing the memory.

Several examples in the old command-based framework involved code like this:

```
#include "PlaceSoda.h"
#include "Elevator.h"
#include "Wrist.h"

PlaceSoda::PlaceSoda() {
    AddSequential(new SetElevatorSetpoint(Elevator::TABLE_HEIGHT));
    AddSequential(new SetWristSetpoint(Wrist::PICKUP));
    AddSequential(new OpenClaw());
}
```

In the command-group above, the component commands of the command group were being heap allocated and passed into `AddSequential` all in the same line. This meant that that user had no reference to that object in memory and therefore had no means of freeing the allocated memory once the command group ended. The command group itself never freed the memory and neither did the command scheduler. This led to memory leaks in robot programs (i.e. memory was allocated on the heap but never freed).

This glaring problem was one of the reasons for the rewrite of the framework. A comprehensive ownership model was introduced with this rewrite, along with the usage of smart pointers which will automatically free memory when they go out of scope.

Default commands are owned by the command scheduler whereas component commands of command groups are owned by the command group. Other commands are owned by whatever the user decides they should be owned by (e.g. a subsystem instance or a `RobotContainer` instance). This means that the ownership of the memory allocated by any commands or command groups is clearly defined.

`std::unique_ptr` vs. `std::shared_ptr`

Using `std::unique_ptr` allows us to clearly determine who owns the object. Because an `std::unique_ptr` cannot be copied, there will never be more than one instance of a `std::unique_ptr` that points to the same block of memory on the heap. For example, a constructor for `SequentialCommandGroup` takes in a `std::vector<std::unique_ptr<Command>>&&`. This means that it requires an rvalue reference to a vector of `std::unique_ptr<Command>`. Let's go through some example code step-by-step to understand this better:

```
// Let's create a vector to store our commands that we want to run sequentially.
std::vector<std::unique_ptr<Command>> commands;

// Add an instant command that prints to the console.
commands.emplace_back(std::make_unique<InstantCommand>([]{ std::cout << "Hello"; },
↳ requirements));

// Add some other command: this can be something that a user has created.
commands.emplace_back(std::make_unique<MyCommand>(args, needed, for, this, command));
```

(continues on next page)

(continued from previous page)

```
// Now the vector "owns" all of these commands. In its current state, when the vector
↳is destroyed (i.e.
// it goes out of scope), it will destroy all of the commands we just added.

// Let's create a SequentialCommandGroup that will run these two commands
↳sequentially.
auto group = SequentialCommandGroup(std::move(commands));

// Note that we MOVED the vector of commands into the sequential command group,
↳meaning that the
// command group now has ownership of our commands. When we call std::move on the
↳vector, all of its
// contents (i.e. the unique_ptr instances) are moved into the command group.

// Even if the vector were to be destroyed while the command group was running,
↳everything would be OK
// since the vector does not own our commands anymore.
```

With `std::shared_ptr`, there is no clear ownership model because there can be multiple instances of a `std::shared_ptr` that point to the same block of memory. If commands were in `std::shared_ptr` instances, a command group or the command scheduler cannot take ownership and free the memory once the command has finished executing because the user might still unknowingly still have a `std::shared_ptr` instance pointing to that block of memory somewhere in scope.

29.14.2 Use of CRTP

You may have noticed that in order to create a new command, you must extend `CommandHelper`, providing the base class (usually `frc2::Command`) and the class that you just created. Let's take a look at the reasoning behind this:

Command Decorators

The new command-based framework includes a feature known as “command decorators”, which allows the user to do something like this:

```
auto task = MyCommand().AndThen([] { std::cout << "This printed after my command
↳ended."; },
    requirements);
```

When `task` is scheduled, it will first execute `MyCommand()` and once that command has finished executing, it will print the message to the console. The way this is achieved internally is by using a sequential command group.

Recall from the previous section that in order to construct a sequential command group, we need a vector of unique pointers to each command. Creating the unique pointer for the print function is pretty trivial:

```
temp.emplace_back(
    std::make_unique<InstantCommand>(std::move(toRun), requirements));
```


Here `temp` is storing the vector of commands that we need to pass into the `SequentialCommandGroup` constructor. But before we add that `InstantCommand`, we need to add `MyCommand()` to the `SequentialCommandGroup`. How do we do that?

```
temp.emplace_back(std::make_unique<MyCommand>(std::move(*this)));
```

You might think it would be this straightforward, but that is not the case. Because this decorator code is in the `Command` interface, `*this` refers to the `Command` in the subclass that you are calling the decorator from and has the type of `Command`. Effectively, you will be trying to move a `Command` instead of `MyCommand`. We could cast the `this` pointer to a `MyCommand*` and then dereference it but we have no information about the subclass to cast to at compile-time.

Solutions to the Problem

Our initial solution to this was to create a virtual method in `Command` called `TransferOwnership()` that every subclass of `Command` had to override. Such an override would have looked like this:

```
std::unique_ptr<Command> TransferOwnership() && override {
    return std::make_unique<MyCommand>(std::move(*this));
}
```

Because the code would be in the derived subclass, `*this` would actually point to the desired subclass instance and the user has the type info of the derived class to make the unique pointer.

After a few days of deliberation, a CRTP method was proposed. Here, an intermediary derived class of `Command` called `CommandHelper` would exist. `CommandHelper` would have two template arguments, the original base class and the desired derived subclass. Let's take a look at a basic implementation of `CommandHelper` to understand this:

```
// In the real implementation, we use SFINAE to check that Base is actually a
// Command or a subclass of Command.
template<typename Base, typename Derived>
class CommandHelper : public Base {
    // Here, we are just inheriting all of the superclass (base class) constructors.
    using Base::Base;

    // Here, we will override the TransferOwnership() method mentioned above.
    std::unique_ptr<Command> TransferOwnership() && override {
        // Previously, we mentioned that we had no information about the derived class
        // to cast to at compile-time, but because of CRTP we do! It's one of our template
        // arguments!
        return std::make_unique<Derived>(std::move(*static_cast<Derived*>(this)));
    }
};
```

Thus, making your custom commands extend `CommandHelper` instead of `Command` will automatically implement this boilerplate for you and this is the reasoning behind asking teams to use what may seem to be a rather obscure way of doing things.

Going back to our `AndThen()` example, we can now do the following:

```
// Because of how inheritance works, we will call the TransferOwnership()
// of the subclass. We are moving *this because TransferOwnership() can only
```

(continues on next page)

(continued from previous page)

```
// be called on rvalue references.
temp.emplace_back(std::move(*this).TransferOwnership());
```

29.14.3 Lack of Advanced Decorators

Most of the C++ decorators take in `std::function<void()>` instead of actual commands themselves. The idea of taking in actual commands in decorators such as `AndThen()`, `BeforeStarting()`, etc. was considered but then abandoned due to a variety of reasons.

Templating Decorators

Because we need to know the types of the commands that we are adding to a command group at compile-time, we will need to use templates (variadic for multiple commands). However, this might not seem like a big deal. The constructors for command groups do this anyway:

```
template <class... Types,
          typename = std::enable_if_t<std::conjunction_v<
            std::is_base_of<Command, std::remove_reference_t<Types>>...>>>
explicit SequentialCommandGroup(Types&&... commands) {
    AddCommands(std::forward<Types>(commands)...);
}

template <class... Types,
          typename = std::enable_if_t<std::conjunction_v<
            std::is_base_of<Command, std::remove_reference_t<Types>>...>>>
void AddCommands(Types&&... commands) {
    std::vector<std::unique_ptr<Command>> foo;
    ((void)foo.emplace_back(std::make_unique<std::remove_reference_t<Types>>(
        std::forward<Types>(commands))),
    ...);
    AddCommands(std::move(foo));
}
```

Note: This is a secondary constructor for `SequentialCommandGroup` in addition to the vector constructor that we described above.

However, when we make a templated function, its definition must be declared inline. This means that we will need to instantiate the `SequentialCommandGroup` in the `Command.h` header, which poses a problem. `SequentialCommandGroup.h` includes `Command.h`. If we include `SequentialCommandGroup.h` inside of `Command.h`, we have a circular dependency. How do we do it now then?

We use a forward declaration at the top of `Command.h`:

```
class SequentialCommandGroup;

class Command { ... };
```

And then we include `SequentialCommandGroup.h` in `Command.cpp`. If these decorator functions were templated however, we cannot write definitions in the `.cpp` files, resulting in a circular dependency.

Java vs C++ Syntax

These decorators usually save more verbosity in Java (because Java requires raw new calls) than in C++, so in general, it does not make much of a syntactic difference in C++ if you create the command group manually in user code.

[Old] Command Based Programming

Important: This documentation describes the use of the legacy command-based library. While this documentation has been preserved to help teams that have yet to do so, teams are strongly encouraged to migrate to the [new command-based library](#).

30.1 [Old] Command Based - Basics

Important: This documentation describes the use of the legacy command-based library. While this documentation has been preserved to help teams that have yet to do so, teams are strongly encouraged to migrate to the [new command-based library](#).

30.1.1 What is Command-Based Programming?

Important: This documentation describes the use of the legacy command-based library. While this documentation has been preserved to help teams that have yet to do so, teams are strongly encouraged to migrate to the [new command-based library](#).

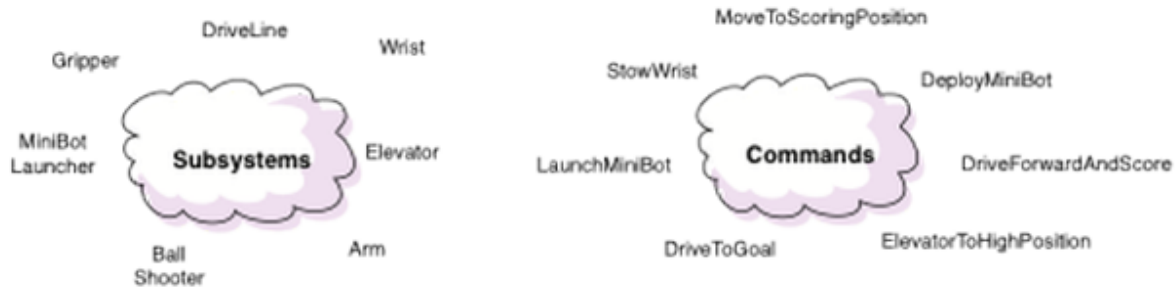
WPILib supports a method of writing programs called “Command-based programming”. Command-based programming is a design pattern to help you organize your robot programs. Some of the characteristics of robot programs that might be different from other desktop programs are:

- Activities happen over time, for example a sequence of steps to shoot a Frisbee or raise an elevator and place a tube on a goal.
- These activities occur concurrently, that is it might be desirable for an elevator, wrist and gripper to all be moving into a pickup position at the same time to increase robot performance.
- It is desirable to test the robot mechanisms and activities each individually to help debug your robot.

- Often the program needs to be augmented with additional autonomous programs at the last minute, perhaps at competitions, so easily extendable code is important.

Command-based programming supports all these goals easily to make the robot program much simpler than using some less structured technique.

Commands and Subsystems

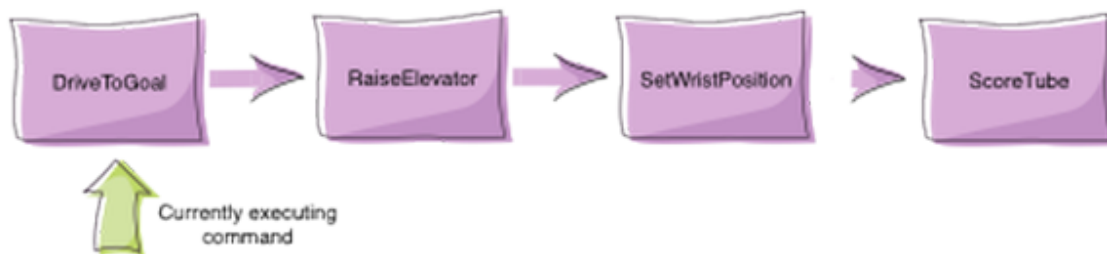


Programs based on the WPILib Command Template are organized around two fundamental concepts: **Subsystems** and **Commands**.

Subsystems - define the capabilities of each part of the robot and are subclasses of `Subsystem`.

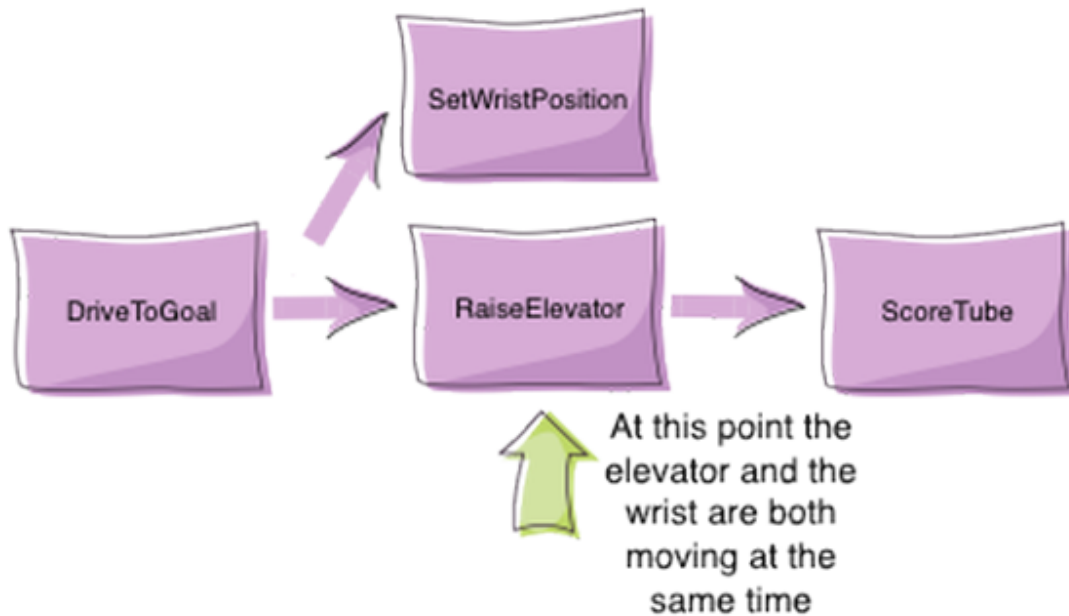
Commands - define the operation of the robot incorporating the capabilities defined in the subsystems. Commands are subclasses of `Command` or `CommandGroup`. Commands run when scheduled or in response to buttons being pressed or virtual buttons from the SmartDashboard.

How Commands Work



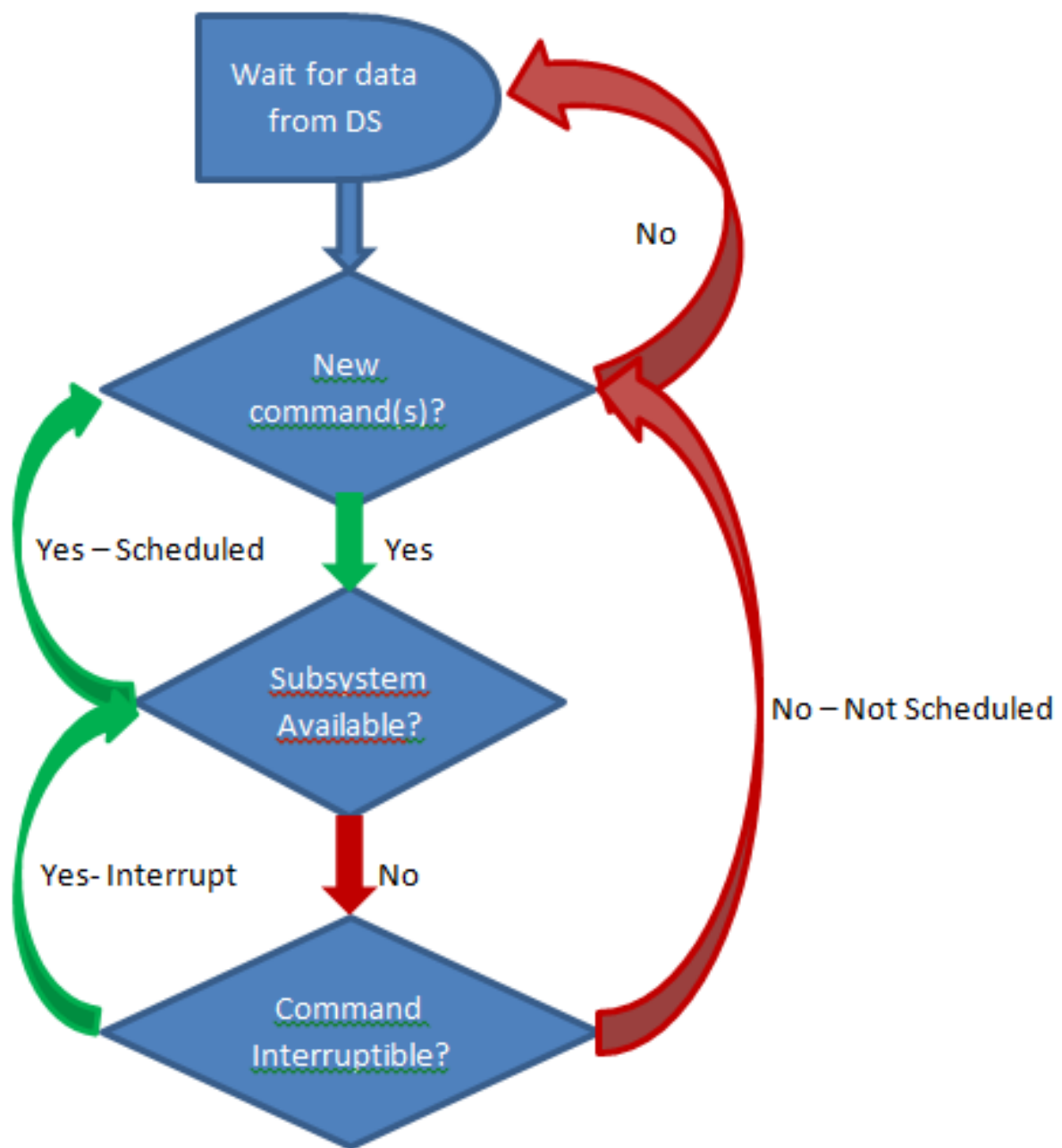
Commands let you break up the tasks of operating the robot into small chunks. Each command has an `execute()` method that does some work and an `isFinished()` method that tells if it is done. This happens on every update from the driver station or about every 20ms. Commands can be grouped together and executed sequentially, starting the next one in the group as the previous one finishes.

Concurrency



Sometimes it is desirable to have several operations happening concurrently. In the previous example you might want to set the wrist position while the elevator is moving up. In this case, a command group can start a parallel command (or command group) running.

How It Works - Scheduling Commands



There are three main ways commands are scheduled:

1. Manually, by calling the `start()` method on the command (*used for autonomous*)
2. Automatically by the scheduler based on *button/trigger actions* specified in the code (typically defined in the OI class but checked by the Scheduler).
3. Automatically when a previous command completes (*default commands* and *command groups*).

Each time the driver station gets new data, the periodic method of your robot program is

called. It runs a Scheduler that checks the trigger conditions to see if any commands need to be scheduled or canceled.

When a command is scheduled, the Scheduler checks to make sure that no other commands are using the same subsystems that the new command requires. If one or more of the subsystems is currently in use, and the current command is interruptible, it will be interrupted and the new command will be scheduled. If the current command is not interruptible, the new command will fail to be scheduled.

How It Works - Running Commands

After checking for new commands, the scheduler proceeds through the list of active commands and calls the `execute()` and `isFinished()` methods on each command. Notice that the apparent concurrent execution is done without the use of threads or tasks which would add complexity to the program. Each command simply has some code to execute (`execute` method) to move it further along towards its goal and a method (`isFinished`) that determines if the command has reached the goal. The `execute` and `isFinished` methods are just called repeatedly.

Command Groups

More complex commands can be built up from simpler commands. For example, shooting a disc may be a long sequence of commands that are executed one after another. Maybe some of these commands in the sequence can be executed concurrently. Command groups are commands, but instead of having an `isFinished` and `execute` method, they have a list of other commands to execute. This allows more complex operations to be built up out of simpler operations, a basic principle in programming. Each of the individual smaller commands can be easily tested first, then the group can be tested. More information on command groups can be found in the [Creating groups of commands article](#).

30.1.2 Creating a Robot Project

Important: This documentation describes the use of the legacy command-based library. While this documentation has been preserved to help teams that have yet to do so, teams are strongly encouraged to migrate to the [new command-based library](#).

Creating a project is detailed in [Creating a Robot Program](#). Select “Template” then your programming language then “Old Command Robot” to create a basic Command-Based Robot program. Alternately you can use RobotBuilder to create the framework of your Command-Based Robot project as detailed in [RobotBuilder](#).

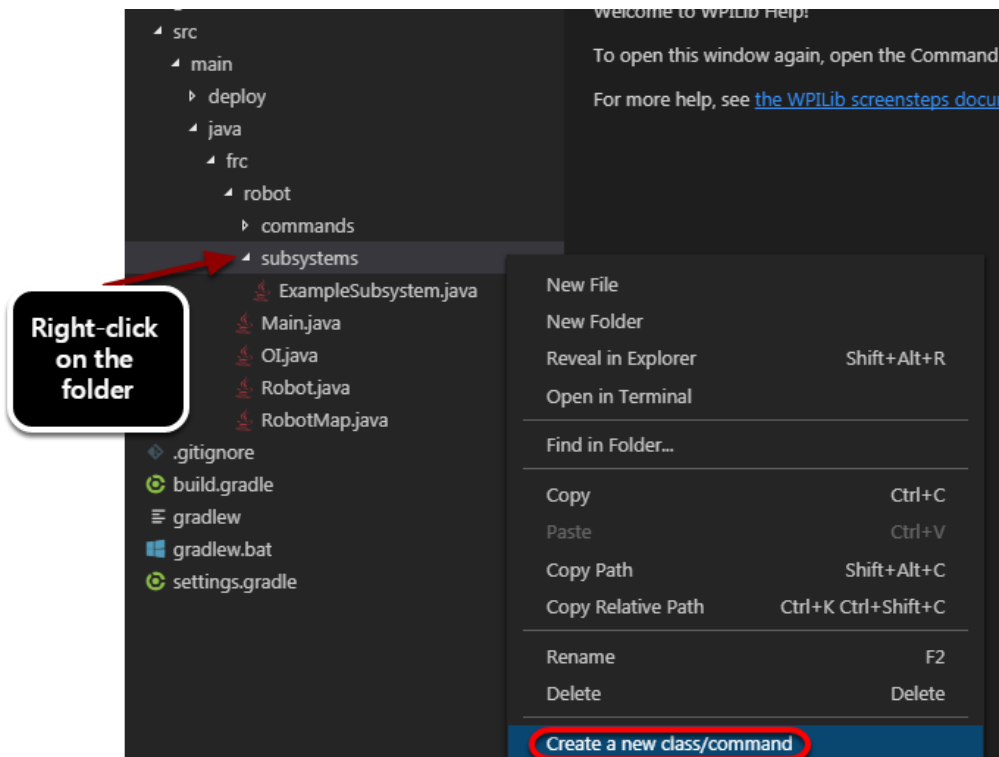
When you create an Old Command Robot project or use RobotBuilder to export a project, the old command based vendor library is automatically imported.

30.1.3 Adding Commands and Subsystems to the Project

Important: This documentation describes the use of the legacy command-based library. While this documentation has been preserved to help teams that have yet to do so, teams are strongly encouraged to migrate to the *new command-based library*.

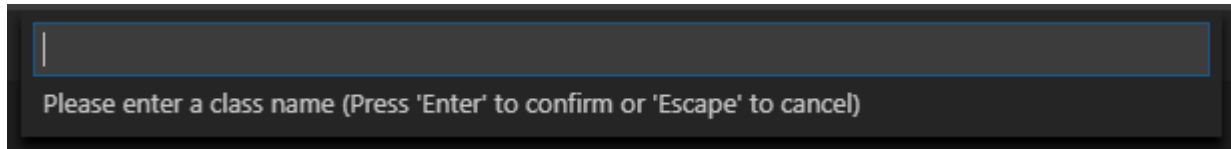
Note: Commands and Subsystems each are created as classes. The plugin has built-in templates for both Commands and Subsystems to make it easier for you to add them to your program.

Adding Subsystems to the Project



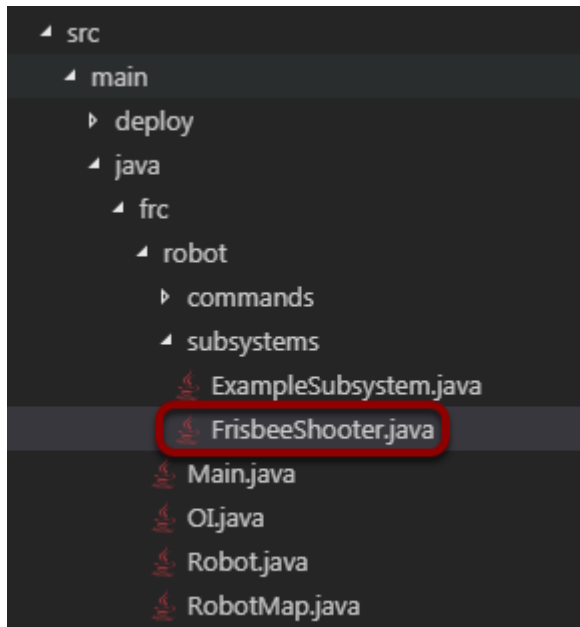
To add a subsystem, right-click on the desired folder and select **Create a new class/command** in the drop down menu. Then select **Subsystem (Old)** or **PID Subsystem (Old)**.

Naming the Subsystem



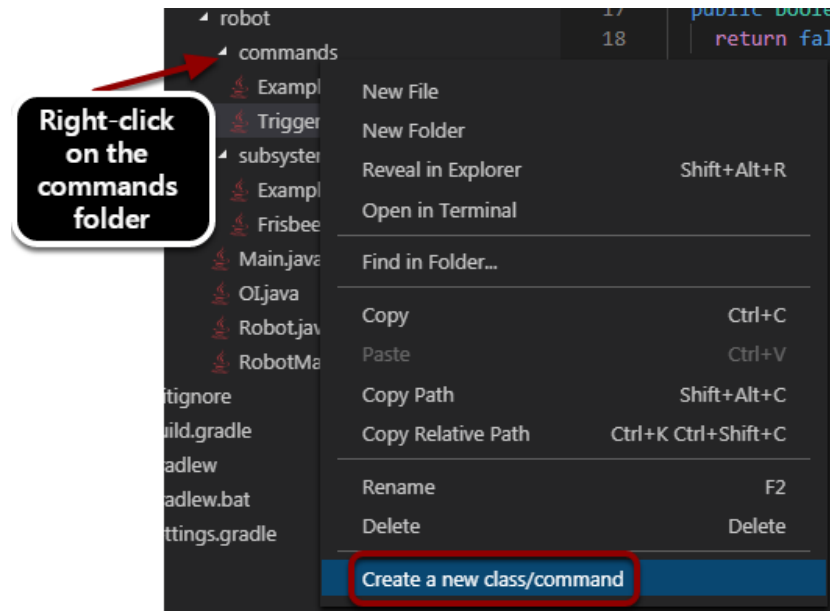
Fill in a name for the subsystem. This will become the resultant class name for the subsystem so the name has to be a valid class name for your language.

Subsystem Created in Project



You can see the new subsystem created in the Subsystems folder in the project. To learn more about creating subsystems, see the [Simple Subsystems](#) article.

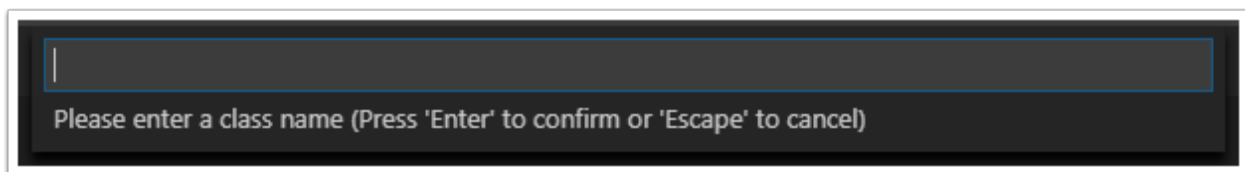
Adding a Command to the Project



A command can be created for the project using steps similar to creating a subsystem. First right-click on the folder name in the project, then select **Create a new class/command** in the drop down menu. Then select **Command (Old)**, **Instant Command (Old)**, **TimedCommand (Old)**, **Command Group (Old)**:

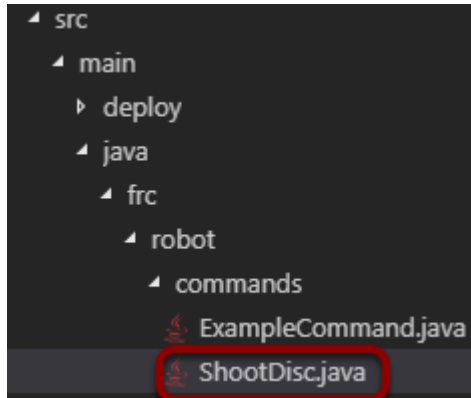
- **Command** - A basic command that operates on a subsystem
- **Instant Command** - A command that runs and completes instantly
- **Timed Command** - A command that runs for a specified time duration
- **Command Group** - A command that runs other commands

Set the Command Name



Enter the Command name into the dialog box. This will be the class name for the Command so it must be a valid class name for your language.

Command Created in the Project



You can see that the Command has been created in the Commands folder in the project in the Project Explorer window. To learn more about creating commands, see the [Creating Simple Commands article](#).

30.1.4 Converting a Simple Autonomous Program to Command-Based

Important: This documentation describes the use of the legacy command-based library. While this documentation has been preserved to help teams that have yet to do so, teams are strongly encouraged to migrate to the [new command-based library](#).

This document describes how to rewrite a simple autonomous into a command based autonomous. Hopefully, going through this process will help those more familiar with the older simple autonomous method understand the command based method better. By re-writing it as a command based program, there are several benefits in terms of testing and reuse. For this example, all of the logic is abstracted out into functions primarily so that the focus of this example can be on the structure.

Initial Autonomous Code

Java

C++

```
// Aim shooter
SetTargetAngle(); // Initialization: prepares for the action to be performed

while (!AtRightAngle()) { // Condition: keeps the loop going while it is satisfied
    CorrectAngle(); // Execution: repeatedly updates the code to try to make the
    ↳condition false
    delay(); // Delay to prevent maxing CPU
}

HoldAngle(); // End: performs any cleanup and final task before moving on to the next
↳action

// Spin up to Speed
```

(continues on next page)

(continued from previous page)

```

SetTargetSpeed(); // Initialization: prepares for the action to be performed

while (!FastEnough()) { // Condition: keeps the loop going while it is satisfied
    SpeedUp(); // Execution: repeatedly updates the code to try to make the condition
    ↪ false
    delay(); // Delay to prevent maxing CPU
}

HoldSpeed();

// Shoot Frisbee
Shoot(); // End: performs any cleanup and final task before moving on to the next
    ↪ action

```

```

// Aim shooter
SetTargetAngle(); // Initialization: prepares for the action to be performed

while (!AtRightAngle()) { // Condition: keeps the loop going while it is satisfied
    CorrectAngle(); // Execution: repeatedly updates the code to try to make the
    ↪ condition false
    delay(); // Delay to prevent maxing CPU
}

HoldAngle(); // End: performs any cleanup and final task before moving on to the next
    ↪ action

// Spin up to Speed
SetTargetSpeed(); // Initialization: prepares for the action to be performed

while (!FastEnough()) { // Condition: keeps the loop going while it is satisfied
    SpeedUp(); // Execution: repeatedly updates the code to try to make the condition
    ↪ false
    delay(); // Delay to prevent maxing CPU
}

HoldSpeed();

// Shoot Frisbee
Shoot(); // End: performs any cleanup and final task before moving on to the next
    ↪ action

```

The code above aims a shooter, then it spins up a wheel and, finally, once the wheel is running at the desired speed, it shoots the frisbee. The code consists of three distinct actions: aim, spin up to speed and shoot the Frisbee. The first two actions follow a command pattern that consists of four parts:

1. Initialization: prepares for the action to be performed.
2. Condition: keeps the loop going while it is satisfied.
3. Execution: repeatedly updates the code to try to make the condition false.
4. End: performs any cleanup and final task before moving on to the next action.

The last action only has an explicit initialize, though depending on how you read it, it can implicitly end under a number of conditions. The most obvious one two in this case are when it's done shooting or when autonomous has ended.

Rewriting it as Commands

Java

C++

```
public class AutonomousCommand extends CommandGroup {

    public AutonomousCommand() {
        addSequential(new Aim());
        addSequential(new SpinUpShooter());
        addSequential(new Shoot());
    }
}
```

```
#include "AutonomousCommand.h"
```

```
AutonomousCommand::AutonomousCommand()
{
    AddSequential(new Aim());
    AddSequential(new SpinUpShooter());
    AddSequential(new Shoot());
}
```

The same code can be rewritten as a `CommandGroup` that groups the three actions, where each action is written as it's own command. First, the command group will be written, then the commands will be written to accomplish the three actions. This code is pretty straightforward. It does the three actions sequentially, that is one after the other. Line 3 aims the robot, then line 4 spins the shooter up and, finally, line 5 actually shoots the frisbee. The `addSequential()` method sets it so that these commands run one after the other.

The Aim Command

Java

C++

```
public class Aim extends Command {

    public Aim() {
        requires(Robot.turret);
    }

    // Called just before this Command runs the first time
    protected void initialize() {
        SetTargetAngle();
    }

    // Called repeatedly when this Command is scheduled to run
    protected void execute() {
        CorrectAngle();
    }

    // Make this return true when this Command no longer needs to run execute()
    protected boolean isFinished() {
        return AtRightAngle();
    }
}
```

(continues on next page)

(continued from previous page)

```
}

// Called once after isFinished returns true
protected void end() {
    HoldAngle();
}

// Called when another command which requires one or more of the same
// subsystems is scheduled to run
protected void interrupted() {
    end();
}
}
```

```
#include "Aim.h"

Aim::Aim()
{
    Requires(Robot::turret);
}

// Called just before this Command runs the first time
void Aim::Initialize()
{
    SetTargetAngle();
}

// Called repeatedly when this Command is scheduled to run
void Aim::Execute()
{
    CorrectAngle();
}

// Make this return true when this Command no longer needs to run execute()
bool Aim::IsFinished()
{
    return AtRightAngle();
}

// Called once after isFinished returns true
void Aim::End()
{
    HoldAngle();
}

// Called when another command which requires one or more of the same
// subsystems is scheduled to run
void Aim::Interrupted()
{
    End();
}
```

As you can see, the command reflects the four parts of the action we discussed earlier. It also has the `interrupted()` method which will be discussed below. The other significant difference is that the condition in the `isFinished()` is the opposite of what you would put as the condition of the while loop, it returns true when you want to stop running the execute method as opposed to false. Initializing, executing and ending are exactly the same, they just

go within their respective method to indicate what they do.

SpinUpShooter Command

Java

C++

```
public class SpinUpShooter extends Command {

    public SpinUpShooter() {
        requires(Robot.shooter);
    }

    // Called just before this Command runs the first time
    protected void initialize() {
        SetTargetSpeed();
    }

    // Called repeatedly when this Command is scheduled to run
    protected void execute() {
        SpeedUp();
    }

    // Make this return true when this Command no longer needs to run execute()
    protected boolean isFinished() {
        return FastEnough();
    }

    // Called once after isFinished returns true
    protected void end() {
        HoldSpeed();
    }

    // Called when another command which requires one or more of the same
    // subsystems is scheduled to run
    protected void interrupted() {
        end();
    }
}
```

```
#include "SpinUpShooter.h"

SpinUpShooter::SpinUpShooter()
{
    Requires(Robot::shooter)
}

// Called just before this Command runs the first time
void SpinUpShooter::Initialize()
{
    SetTargetSpeed();
}

// Called repeatedly when this Command is scheduled to run
void SpinUpShooter::Execute()
```

(continues on next page)

(continued from previous page)

```

{
    SpeedUp();
}

// Make this return true when this Command no longer needs to run execute()
bool SpinUpShooter::IsFinished()
{
    return FastEnough();
}

// Called once after isFinished returns true
void SpinUpShooter::End()
{
    HoldSpeed();
}

// Called when another command which requires one or more of the same
// subsystems is scheduled to run
void SpinUpShooter::Interrupted()
{
    End();
}

```

The spin up shooter command is very similar to the Aim command, it's the same basic idea.

Shoot Command

Java

C++

```

public class Shoot extends Command {

    public Shoot() {
        requires(shooter);
    }

    // Called just before this Command runs the first time
    protected void initialize() {
        Shoot();
    }

    // Called repeatedly when this Command is scheduled to run
    protected void execute() {
    }

    // Make this return true when this Command no longer needs to run execute()
    protected boolean isFinished() {
        return true;
    }

    // Called once after isFinished returns true
    protected void end() {
    }
}

```

(continues on next page)

(continued from previous page)

```

// Called when another command which requires one or more of the same
// subsystems is scheduled to run
protected void interrupted() {
    end();
}
}

#include "Shoot.h"

Shoot::Shoot()
{
    Requires(Robot.shooter);
}

// Called just before this Command runs the first time
void Shoot::Initialize()
{
    Shoot();
}

// Called repeatedly when this Command is scheduled to run
void Shoot::Execute()
{
}

// Make this return true when this Command no longer needs to run execute()
bool Shoot::IsFinished()
{
    return true;
}

// Called once after isFinished returns true
void Shoot::End()
{
}

// Called when another command which requires one or more of the same
// subsystems is scheduled to run
void Shoot::Interrupted()
{
    End();
}

```

The shoot command is the same basic transformation yet again, however it is set to end immediately. In CommandBased programming, it is better to have it's isFinished method return true when the act of shooting is finished, but this is a more direct translation of the original code.

Benefits of Command-Based

Why bother re-writing the code as CommandBased? Writing the code in the CommandBased style offers a number of benefits:

- **Re-Usability** - You can reuse the same command in teleop and multiple autonomous modes. They all reference the same code, so if you need to tweak it to tune it or fix it, you can do it in one place without having to make the same edits in multiple places.
- **Testability** - You can test each part using tools such as the ShuffleBoard to test parts of the autonomous. Once you put them together, you'll have more confidence that each piece works as desired.
- **Parallelization** - If you wanted this code to aim and spin up the shooter at the same time, it's trivial with CommandBased programming. Just use `AddParallel()` instead of `AddSequential()` when adding the Aim command and now aiming and spinning up will happen simultaneously.
- **Interruptibility** - Commands are interruptible, this provides the ability to exit a command early, a task that is much harder in the equivalent while loop based code.

30.2 [Old] Command Based - Subsystems

Important: This documentation describes the use of the legacy command-based library. While this documentation has been preserved to help teams that have yet to do so, teams are strongly encouraged to migrate to the [new command-based library](#).

30.2.1 Simple Subsystems

Important: This documentation describes the use of the legacy command-based library. While this documentation has been preserved to help teams that have yet to do so, teams are strongly encouraged to migrate to the [new command-based library](#).

Subsystems are the parts of your robot that are independently controlled like collectors, shooters, drive bases, elevators, arms, wrists, grippers, etc. Each subsystem is coded as an instance of the Subsystem class. Subsystems should have methods that define the operation of the actuators and sensors but not more complex behavior that happens over time.

Creating a Subsystem

Java

C++

```
import edu.wpi.first.wpilibj.DigitalInput;
import edu.wpi.first.wpilibj.Victor;
import edu.wpi.first.wpilibj.command.Subsystem;
```

(continues on next page)

(continued from previous page)

```

/**
 * The claw subsystem is a simple system with a motor for opening and closing.
 * If using stronger motors, you should probably use a sensor so that the motors
 * don't stall.
 */
public class Claw extends Subsystem {
    private final Victor m_motor = new Victor(7);
    private final DigitalInput m_contact = new DigitalInput(5);

    /**
     * Create a new claw subsystem.
     */
    public Claw() {
        super();
    }

    @Override
    public void initDefaultCommand() {
    }

    /**
     * Set the claw motor to move in the open direction.
     */
    public void open() {
        m_motor.set(-1);
    }

    /**
     * Set the claw motor to move in the close direction.
     */
    @Override
    public void close() {
        m_motor.set(1);
    }

    /**
     * Stops the claw motor from moving.
     */
    public void stop() {
        m_motor.set(0);
    }

    /**
     * Return true when the robot is grabbing an object hard enough to trigger
     * the limit switch.
     */
    public boolean isGrabbing() {
        return m_contact.get();
    }
}

```

```
#include "subsystems/Claw.h"
```

```

Claw::Claw() : frc::Subsystem("Claw") {
    // Let's show everything on the LiveWindow
    AddChild("Motor", m_motor);
}

```

(continues on next page)

(continued from previous page)

```

}

void Claw::InitDefaultCommand() {}

/**
 * Set the claw motor to move in the open direction.
 */
void Claw::Open() { m_motor.Set(-1); }

/**
 * Set the claw motor to move in the close direction.
 */
void Claw::Close() { m_motor.Set(1); }

/**
 * Stops the claw motor from moving.
 */
void Claw::Stop() { m_motor.Set(0); }

/**
 * Return true when the robot is grabbing an object hard enough to trigger
 * the limit switch.
 */
bool Claw::IsGripping() { return m_contact.Get(); }

```

This is an example of a fairly straightforward subsystem that operates a claw on a robot. The claw mechanism has a single motor to open or close the claw and no sensors (not necessarily a good idea in practice, but works for the example). The idea is that the open and close operations are simply timed. There are three methods, `open()`, `close()`, and `stop()` that operate the claw motor. Notice that there is not specific code that actually checks if the claw is opened or closed. The open method gets the claw moving in the open direction and the close method gets the claw moving in the close direction. Use a command to control the timing of this operation to make sure that the claw opens and closes for a specific period of time.

Operating the Claw with a Command

Java

C++

```

package org.usfirst.frc.team1.robot.commands;

import edu.wpi.first.wpilibj.command.Command;
import org.usfirst.frc.team1.robot.Robot;

public class OpenClaw extends Command {

    public OpenClaw() {
        requires(Robot.claw);
        setTimeout(.9);
    }

    protected void initialize() {
        Robot.claw.open()
    }
}

```

(continues on next page)

(continued from previous page)

```

protected void execute() {
}

protected boolean isFinished() {
    return isTimedOut();
}

protected void end() {
    Robot.claw.stop();
}

protected void interrupted() {
    end();
}
}

```

```

#include "commands/OpenClaw.h"

#include "Robot.h"

OpenClaw::OpenClaw() : frc::Command("OpenClaw") {
    Requires(&Robot::claw);
    SetTimeout(1);
}

// Called just before this Command runs the first time
void OpenClaw::Initialize() { Robot::claw.Open(); }

// Make this return true when this Command no longer needs to run execute()
bool OpenClaw::IsFinished() { return IsTimedOut(); }

// Called once after isFinished returns true
void OpenClaw::End() { Robot::claw.Stop(); }

```

Commands provide the timing of the subsystems operations. Each command would do a different operation with the subsystem, the Claw in this case. The commands provides the timing for opening or closing. Here is an example of a simple Command that controls the opening of the claw. Notice that a timeout is set for this command (0.9 seconds) to time the opening of the claw and a check for the time in the `isFinished()` method. You can find more details in the article about [using commands](#).

30.2.2 PIDSubsystems for built-in PID Control

Important: This documentation describes the use of the legacy command-based library. While this documentation has been preserved to help teams that have yet to do so, teams are strongly encouraged to migrate to the [new command-based library](#).

Note: If a mechanism uses a sensor for feedback, then most often a PID controller will be used to control the motor speed or position. Examples of subsystems that might use PID control are: elevators with potentiometers to track the height, shooters with encoders to

measure the speed, wrists with potentiometers to measure the joint angle, etc.

There is a PIDController class built into WPILib, but to simplify its use for command based programs there is a PIDSubsystem. A PIDSubsystem is a normal subsystem with the PIDController built in and exposes the required methods for operation.

Controlling the Angle of a Wrist Joint

In this example you can see the basic elements of a PIDSubsystem for the wrist joint:

Java

C++

```
package org.usfirst.frc.team1.robot.subsystems;
import edu.wpi.first.wpilibj.*;
import edu.wpi.first.wpilibj.command.PIDSubsystem;
import org.usfirst.frc.team1.robot.RobotMap;

public class Wrist extends PIDSubsystem { // This system extends PIDSubsystem

    Victor motor = RobotMap.wristMotor;
    AnalogInput pot = RobotMap.wristPot();

    public Wrist() {
        super("Wrist", 2.0, 0.0, 0.0); // The constructor passes a name for the
        ↪ subsystem and the P, I and D constants that are used when computing the motor output
        setAbsoluteTolerance(0.05);
        getPIDController().setContinuous(false);
    }

    public void initDefaultCommand() {
    }

    protected double returnPIDInput() {
        return pot.getAverageVoltage(); // returns the sensor value that is providing
        ↪ the feedback for the system
    }

    protected void usePIDOutput(double output) {
        motor.pidWrite(output); // this is where the computed output value from the
        ↪ PIDController is applied to the motor
    }
}
```

```
#include "subsystems/Wrist.h"

#include <frc/smartdashboard/SmartDashboard.h>

Wrist::Wrist() : frc::PIDSubsystem("Wrist", kP_real, 0.0, 0.0) {
    #ifdef SIMULATION // Check for simulation and update PID values
        GetPIDController()->SetPID(kP_simulation, 0, 0, 0);
    #endif
    SetAbsoluteTolerance(2.5);
}
```

(continues on next page)

(continued from previous page)

```

void Wrist::InitDefaultCommand() {}

void Wrist::Log() {
    // frc::SmartDashboard::PutData("Wrist Angle", &m_pot);
}

double Wrist::ReturnPIDInput() { return m_pot.Get(); }

void Wrist::UsePIDOutput(double d) { m_motor.Set(d); }

```

30.3 [Old] Command Based - Commands

Important: This documentation describes the use of the legacy command-based library. While this documentation has been preserved to help teams that have yet to do so, teams are strongly encouraged to migrate to the [new command-based library](#).

30.3.1 Creating Simple Commands

Important: This documentation describes the use of the legacy command-based library. While this documentation has been preserved to help teams that have yet to do so, teams are strongly encouraged to migrate to the [new command-based library](#).

This article describes the basic format of a Command and walks through an example of creating a command to drive your robot with Joysticks.

Basic Command Format

To implement a command, a number of methods are overridden from the WPILib Command class. Most of the methods are boiler plate and can often be ignored, but are there for maximum flexibility when you need it. There a number of parts to this basic command class:

Java

C++

```

public class MyCommandName extends Command {

    /*
     * 1. Constructor - Might have parameters for this command such as target
     ↪ positions of devices. Should also set the name of the command for debugging
     ↪ purposes.
     * This will be used if the status is viewed in the dashboard. And the command
     ↪ should require (reserve) any devices is might use.
     */
    public MyCommandName() {

```

(continues on next page)

(continued from previous page)

```

    super("MyCommandName");
    requires(elevator);
}

// initialize() - This method sets up the command and is called immediately
↳ before the command is executed for the first time and every subsequent time it is
↳ started .
// Any initialization code should be here.
protected void initialize() {
}

/*
 * execute() - This method is called periodically (about every 20ms) and does
↳ the work of the command. Sometimes, if there is a position a
 * subsystem is moving to, the command might set the target position for the
↳ subsystem in initialize() and have an empty execute() method.
 */
protected void execute() {
}

// Make this return true when this Command no longer needs to run execute()
protected boolean isFinished() {
    return false;
}
}

```

```

#include "MyCommandName.h"

/*
 * 1. Constructor - Might have parameters for this command such as target
↳ positions of devices. Should also set the name of the command for debugging
↳ purposes.
 * This will be used if the status is viewed in the dashboard. And the command
↳ should require (reserve) any devices it might use.
 */
MyCommandName::MyCommandName() : CommandBase("MyCommandName")
{
    Requires(Elevator);
}

// initialize() - This method sets up the command and is called immediately before
↳ the command is executed for the first time and
// every subsequent time it is started . Any initialization code should be here.
void MyCommandName::Initialize()
{
}

/*
 * execute() - This method is called periodically (about every 20ms) and does the
↳ work of the command. Sometimes, if there is a position a
 * subsystem is moving to, the command might set the target position for the
↳ subsystem in initialize() and have an empty execute() method.
 */
void MyCommandName::Execute()
{
}

```

(continues on next page)

(continued from previous page)

```

bool MyCommandName::IsFinished()
{
    return false;
}

void MyCommandName::End()
{
}

// Make this return true when this Command no longer needs to run execute()
void MyCommandName::Interrupted()
{
}

```

Simple Command Example

This example illustrates a simple command that will drive the robot using tank drive with values provided by the joysticks.

Java

C++

```

public class DriveWithJoysticks extends Command {

    public DriveWithJoysticks() {
        requires(drivetrain); // drivetrain is an instance of our Drivetrain subsystem
    }

    protected void initialize() {
    }

    /*
    * execute() - In our execute method we call a tankDrive method we have created
    ↪ in our subsystem. This method takes two speeds as a parameter which we get from
    ↪ methods in the OI class.
    * These methods abstract the joystick objects so that if we want to change how
    ↪ we get the speed later we can do so without modifying our commands
    * (for example, if we want the joysticks to be less sensitive, we can multiply
    ↪ them by .5 in the getLeftSpeed method and leave our command the same).
    */
    protected void execute() {
        drivetrain.tankDrive(oi.getLeftSpeed(), oi.getRightSpeed());
    }

    /*
    * isFinished - Our isFinished method always returns false meaning this command
    ↪ never completes on it's own. The reason we do this is that this command will be set
    ↪ as the default command for the subsystem. This means that whenever the subsystem is
    ↪ not running another command, it will run this command. If any other command is
    ↪ scheduled it will interrupt this command, then return to this command when the
    ↪ other command completes.
    */
    protected boolean isFinished() {

```

(continues on next page)

(continued from previous page)

```

        return false;
    }

    protected void end() {
    }

    protected void interrupted() {
    }
}

```

```

#include "DriveWithJoysticks.h"
#include "RobotMap.h"

DriveWithJoysticks::DriveWithJoysticks() : CommandBase("DriveWithJoysticks")
{
    Requires(Robot::drivetrain); // Drivetrain is our instance of the drive system
}

// Called just before this Command runs the first time
void DriveWithJoysticks::Initialize()
{
}

    /*
    * execute() - In our execute method we call a tankDrive method we have created
    ↳ in our subsystem. This method takes two speeds as a parameter which we get from
    ↳ methods in the OI class.
    * These methods abstract the joystick objects so that if we want to change how
    ↳ we get the speed later we can do so without modifying our commands
    * (for example, if we want the joysticks to be less sensitive, we can multiply
    ↳ them by .5 in the getLeftSpeed method and leave our command the same).
    */
void DriveWithJoysticks::Execute()
{
    Robot::drivetrain->Drive(Robot::oi->GetLeftSpeed(), Robot::oi->GetRightSpeed());
}

    /*
    * isFinished - Our isFinished method always returns false meaning this command
    ↳ never completes on it's own. The reason we do this is that this command will be set
    ↳ as the default command for the subsystem. This means that whenever the subsystem is
    ↳ not running another command, it will run this command. If any other command is
    ↳ scheduled it will interrupt this command, then return to this command when the
    ↳ other command completes.
    */
bool DriveWithJoysticks::IsFinished()
{
    return false;
}

void DriveWithJoysticks::End()
{
    Robot::drivetrain->Drive(0, 0);
}

// Called when another command which requires one or more of the same

```

(continues on next page)

(continued from previous page)

```
// subsystems is scheduled to run
void DriveWithJoysticks::Interrupted()
{
    End();
}
```

30.3.2 Creating Groups of Commands

Important: This documentation describes the use of the legacy command-based library. While this documentation has been preserved to help teams that have yet to do so, teams are strongly encouraged to migrate to the [new command-based library](#).

Note: Once you have created commands to operate the mechanisms in your robot, they can be grouped together to get more complex operations. These groupings of commands are called CommandGroups and are easily defined as shown in this article.

Complex Operations

Java

C++

```
public class PlaceSoda extends CommandGroup {

    public PlaceSoda() {
        addSequential(new SetElevatorSetpoint(Elevator.TABLE_HEIGHT));
        addSequential(new SetWristSetpoint(Wrist.PICKUP));
        addSequential(new OpenClaw());
    }
}
```

```
#include "PlaceSoda.h"
#include "Elevator.h"
#include "Wrist.h"
```

```
PlaceSoda::PlaceSoda()
{
    AddSequential(new SetElevatorSetpoint(Elevator::TABLE_HEIGHT));
    AddSequential(new SetWristSetpoint(Wrist::PICKUP));
    AddSequential(new OpenClaw());
}
```

This is an example of a command group that places a soda can on a table. To accomplish this, (1) the robot elevator must move to the TABLE_HEIGHT, then (2) set the wrist angle, then (3) open the claw. All of these tasks must run sequentially to make sure that the soda can isn't dropped. The addSequential() method takes a command (or a command group) as a parameter and will execute them one after another when this command is scheduled.

Running Commands in Parallel

Java

C++

```
public class PrepareToGrab extends CommandGroup {  
  
    public PrepareToGrab() {  
        addParallel(new SetWristSetpoint(Wrist.PICKUP));  
        addParallel(new SetElevatorSetpoint(Elevator.BOTTOM));  
        addParallel(new OpenClaw());  
    }  
}
```

```
#include "PrepareToGrab.h"  
#include "Wrist.h"  
#include "Elevator.h"  
  
PrepareToGrab::PrepareToGrab()  
{  
    AddParallel(new SetWristSetpoint(Wrist::PICKUP));  
    AddParallel(new SetElevatorSetpoint(Elevator::BOTTOM));  
    AddParallel(new OpenClaw());  
}
```

To make the program more efficient, often it is desirable to run multiple commands at the same time. In this example, the robot is getting ready to grab a soda can. Since the robot isn't holding anything, all the joints can move at the same time without worrying about dropping anything. Here all the commands are run in parallel so all the motors are running at the same time and each completes whenever the `isFinished()` method is called. The commands may complete out of order. The steps are: (1) move the wrist to the pickup setpoint, then (2) move the elevator to the floor pickup position, and (3) open the claw

Mixing Parallel and Sequential Commands

Java

C++

```
public class Grab extends CommandGroup {  
  
    public Grab() {  
        addSequential(new CloseClaw());  
        addParallel(new SetElevatorSetpoint(Elevator.STOW));  
        addSequential(new SetWristSetpoint(Wrist.STOW));  
    }  
}
```

```
#include "Grab.h"  
#include "Elevator.h"  
#include "Wrist.h"  
  
Grab::Grab()  
{
```

(continues on next page)

(continued from previous page)

```

AddSequential(new CloseClaw());
AddParallel(new SetElevatorSetpoint(Elevator::STOW));
AddSequential(new SetWristSetpoint(Wrist::STOW));
}

```

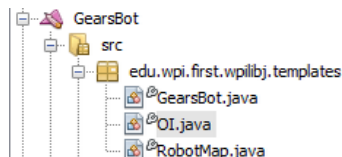
Often there are some parts of a command group that must complete before other parts run. In this example, a soda can is grabbed, then the elevator and wrist can move to their stowed positions. In this case, the wrist and elevator have to wait until the can is grabbed, then they can operate independently. The first command (1) CloseClaw grabs the soda and nothing else runs until it is finished since it is sequential, then the (2) elevator and (3) wrist move at the same time.

30.3.3 Running Commands on Joystick Input

Important: This documentation describes the use of the legacy command-based library. While this documentation has been preserved to help teams that have yet to do so, teams are strongly encouraged to migrate to the *new command-based library*.

Note: You can cause commands to run when joystick buttons are pressed, released, or continuously while the button is held down. This is extremely easy to do only requiring a few lines of code.

The OI Class



```

21 public class OI {
22     // Create the joystick and of the 8 buttons on it
23     Joystick leftJoy = new Joystick(1);
24     Button button1 = new JoystickButton(leftJoy, 1),
25         button2 = new JoystickButton(leftJoy, 2),

```

The command based template contains a class called OI, located in OI.java, where **Operator Interface** behaviors are typically defined.

Create the Joystick object and JoystickButton Objects

Java

C++

```

public class OI {
    // Create the joystick and the 8 buttons on it
    Joystick leftJoy = new Joystick(0);
    Button button1 = new JoystickButton(leftJoy, 1),
        button2 = new JoystickButton(leftJoy, 2),
        button3 = new JoystickButton(leftJoy, 3),
        button4 = new JoystickButton(leftJoy, 4),
        button5 = new JoystickButton(leftJoy, 5),

```

(continues on next page)

(continued from previous page)

```

        button6 = new JoystickButton(leftJoy, 6),
        button7 = new JoystickButton(leftJoy, 7),
        button8 = new JoystickButton(leftJoy, 8);
    }

```

```

OI::OI()
{
    joy = new Joystick(0);

    JoystickButton* button1 = new JoystickButton(joy, 1),
        button2 = new JoystickButton(joy, 2),
        button3 = new JoystickButton(joy, 3),
        button4 = new JoystickButton(joy, 4),
        button5 = new JoystickButton(joy, 5),
        button6 = new JoystickButton(joy, 6),
        button7 = new JoystickButton(joy, 7),
        button8 = new JoystickButton(joy, 8);
}

```

In this example there is a Joystick object connected as Joystick 0. Then 8 buttons are defined on that joystick to control various aspects of the robot. This is especially useful for testing although generating buttons on SmartDashboard is another alternative for testing commands.

Associate the Buttons with Commands

Java

C++

```

public OI() {
    button1.whenPressed(new PrepareToGrab());
    button2.whenPressed(new Grab());
    button3.whenPressed(new DriveToDistance(0.11));
    button4.whenPressed(new PlaceSoda());
    button6.whenPressed(new DriveToDistance(0.2));
    button8.whenPressed(new Stow());

    button7.whenPressed(new SodaDelivery());
}

```

```

button1->WhenPressed(new PrepareToGrab());
button2->WhenPressed(new Grab());
button3->WhenPressed(new DriveToDistance(0.11));
button4->WhenPressed(new PlaceSoda());
button6->WhenPressed(new DriveToDistance(0.2));
button8->WhenPressed(new Stow());

button7->WhenPressed(new SodaDelivery());

```

In this example most of the joystick buttons from the previous code fragment are associated with commands. When the associated button is pressed the command is run. This is an excellent way to create a teleop program that has buttons to do particular actions.

Other Options

In addition to the `whenPressed()` condition showcased above, there are a few other conditions you can use to link buttons to commands:

- Commands can run when a button is released by using `whenReleased()` instead of `whenPressed()`.
- Commands can run continuously while the button is depressed by calling `whileHeld()`.
- Commands can be toggled when a button is pressed using `toggleWhenPressed()`.
- A command can be canceled when a button is pressed using `cancelWhenPressed()`.

Additionally commands can be triggered by arbitrary conditions of your choosing by using the `Trigger` class instead of `Button`. Triggers (and Buttons) are usually polled every 20ms or whenever the scheduler is called.

30.3.4 Running Commands during Autonomous

Important: This documentation describes the use of the legacy command-based library. While this documentation has been preserved to help teams that have yet to do so, teams are strongly encouraged to migrate to the [new command-based library](#).

Note: Once commands are defined they can run in either the teleop or autonomous part of the program. In fact, the power of the command based programming approach is that you can reuse the same commands in either place. If the robot has a command that can shoot Frisbees during autonomous with camera aiming and accurate shooting, there is no reason not to use it to help the drivers during the teleop period of the game.

Java

C++

```
public class SodaDelivery extends CommandGroup {

    public SodaDelivery() {
        addSequential(new PrepareToGrab());
        addSequential(new Grab());
        addSequential(new DriveToDistance(0.11));
        addSequential(new PlaceSoda());
        addSequential(new DriveToDistance(0.2));
        addSequential(new Stow());
    }
}
```

```
#include "PlaceSoda.h"

PlaceSoda::PlaceSoda()
{
    AddSequential(new PrepareToGrab());
    AddSequential(new Grab());
    AddSequential(new DriveToDistance(0.11));
}
```

(continues on next page)

(continued from previous page)

```
AddSequential(new PlaceSoda());
AddSequential(new DriveToDistance(0.2));
AddSequential(new Stow());
}
```

Our robot must do the following tasks during the autonomous period: pick up a soda can off the floor then drive a set distance from a table and deliver the can there. The process consists of:

1. Prepare to grab (move elevator, wrist, and gripper into position)
2. Grab the soda can
3. Drive to a distance from the table indicated by an ultrasonic rangefinder
4. Place the soda
5. Back off to a distance from the rangefinder
6. Re-stow the gripper

To do these tasks there are 6 commands that are executed sequentially as shown in this example (note that some of these, such as PlaceSoda are other command groups).

Setting the Command to Run during Autonomous

Java

C++

```
public class Robot extends TimedRobot {
    Command autonomousCommand;

    /**
     * This function is run when the robot is first started up and should be
     * used for any initialization code.
     */
    public void robotInit() {
        oi = new OI();
        // instantiate the command used for the autonomous period
        autonomousCommand = new SodaDelivery();
    }

    public void autonomousInit() {
        // schedule the autonomous command (example)
        if (autonomousCommand != null) autonomousCommand.start();
    }

    /**
     * This function is called periodically during autonomous
     */
    public void autonomousPeriodic() {
        Scheduler.getInstance().run();
    }
}
```

```
Command* autonomousCommand;
```

(continues on next page)

(continued from previous page)

```

class Robot: public TimedRobot {

    /**
     * This function is run when the robot is first started up and should be
     * used for any initialization code.
     */
    void RobotInit()
    {
        // instantiate the command used for the autonomous period
        autonomousCommand = new SodaDelivery();
        oi = new OI();
    }

    void AutonomousInit()
    {
        // schedule the autonomous command (example)
        if(autonomousCommand != NULL) autonomousCommand->Start();
    }
    /**
     * This function is called periodically during autonomous
     */
    void AutonomousPeriodic()
    {
        Scheduler::GetInstance()->Run();
    }
}

```

To get the SodaDelivery command to run as the Autonomous program,

1. Instantiate it in the RobotInit() method. RobotInit() is called only once when the robot starts so it is a good time to create the command instance.
2. Start it during the AutonomousInit() method. AutonomousInit() is called once at the start of the autonomous period so we schedule the command there.
3. Be sure the scheduler is called repeatedly during the AutonomousPeriodic() method. AutonomousPeriodic() is called (nominally) every 20ms so that is a good time to run the scheduler which makes a pass through all the currently scheduled commands.

30.3.5 Default Commands

Important: This documentation describes the use of the legacy command-based library. While this documentation has been preserved to help teams that have yet to do so, teams are strongly encouraged to migrate to the [new command-based library](#).

In some cases you may have a subsystem which you want to always be running a command no matter what. So what do you do when the command you are currently running ends? That's where default commands come in.

The Default Command

Each subsystem may, but is not required to, have a default command which is scheduled whenever the subsystem is idle (the command currently requiring the system completes). The most common example of a default command is a command for the drivetrain that implements the normal joystick control. This command may be interrupted by other commands for specific maneuvers (“precision mode”, automatic alignment/targeting, etc.) but after any command requiring the drivetrain completes the joystick command would be scheduled again.

Setting the Default Command

Java

C++

```
public class ExampleSubsystem extends Subsystem {  
  
    // Put methods for controlling this subsystem  
    // here. Call these from Commands.  
  
    public void initDefaultCommand() {  
        // Set the default command for a subsystem here.  
        setDefaultCommand(new MyDefaultCommand());  
    }  
}
```

```
#include "ExampleSubsystem.h"  
  
ExampleSubsystem::ExampleSubsystem()  
{  
    // Put methods for controlling this subsystem  
    // here. Call these from Commands.  
}  
  
ExampleSubsystem::InitDefaultCommand()  
{  
    // Set the default command for a subsystem here.  
    SetDefaultCommand(new MyDefaultCommand());  
}
```

All subsystems should contain a method called `initDefaultCommand()` which is where you will set the default command if desired. If you do not wish to have a default command, simply leave this method blank. If you do wish to set a default command, call `setDefaultCommand` from within this method, passing in the command to be set as the default.

30.3.6 Synchronizing Two Commands

Important: This documentation describes the use of the legacy command-based library. While this documentation has been preserved to help teams that have yet to do so, teams are strongly encouraged to migrate to the [new command-based library](#).

Commands can be nested inside of command groups to create more complex commands. The simpler commands can be added to the command groups to either run sequentially (each command finishing before the next starts) or in parallel (the command is scheduled, and the next command is immediately scheduled also). Occasionally there are times where you want to make sure that two parallel command complete before moving onto the next command. This article describes how to do that.

CommandGroup with Sequential and Parallel Commands

Java

C++

```
public class CoopBridgeAutonomous extends CommandGroup {

    public CoopBridgeAutonomous() {
        //SmartDashboard.putDouble("Camera Time", 5.0);
        addSequential(new SetTipperState(BridgeTipper.READY_STATE)); // 1
        addParallel(new SetVirtualSetpoint(SetVirtualSetpoint.HYBRID_LOCATION)); // 2
        addSequential(new DriveToBridge()); // 3
        addParallel(new ContinuousCollect());
        addSequential(new SetTipperState(BridgeTipper.DOWN_STATE));

        // addParallel(new WaitThenShoot());

        addSequential(new TurnToTargetLowPassFilterHybrid(4.0));
        addSequential(new FireSequence());
        addSequential(new MoveBallToShooter(true));
    }
}
```

```
#include "CoopBridgeAutonomous.h"

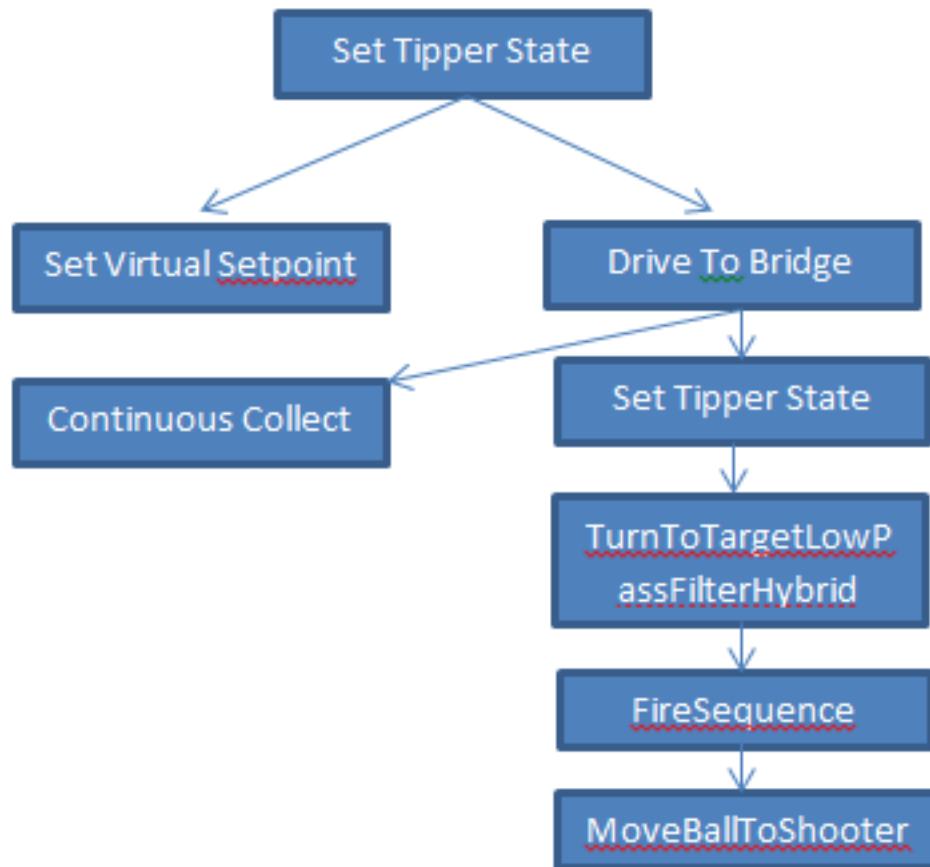
CoopBridgeAutonomous::CoopBridgeAutonomous()
{
    // SmartDashboard->PutDouble("Camera Time", 5.0);
    AddSequential(new SetTipperState(READY_STATE);
    AddParallel(new SetVirtualSetpoint(HYBRID_LOCATION);
    AddSequential(new DriveToBridge());
    AddParallel(new ContinuousCollect());
    AddSequential(new SetTipperState(DOWN_STATE));

    // addParallel(new WaitThenShoot());

    AddSequential(new TurnToTargetLowPassFilterHybrid(4.0));
    AddSequential(new FireSequence());
    AddSequential(new MoveBallToShooter(true));
}
```

In this example some commands are added in parallel and others are added sequentially to the CommandGroup CoopBridgeAutonomous (1). The first command “SetTipperState” is added and completes before the SetVirtualSetpoint command starts (2). Before SetVirtualSetpoint command completes, the DriveToBridge command is immediately scheduled because of the SetVirtualSetpoint is added in parallel (3). This example might give you an idea of how commands are scheduled.

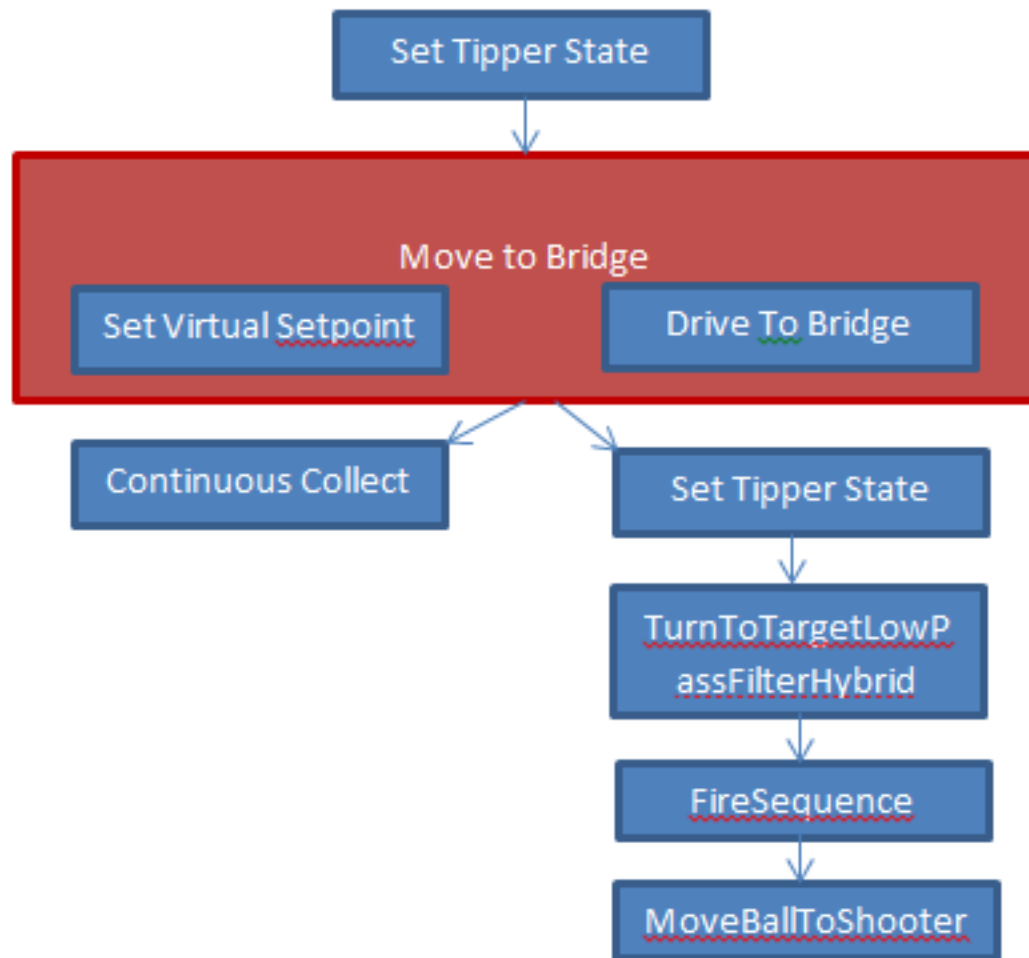
Example Flowchart



Note: There is no dependency coming from the commands scheduled using “Add Parallel” either or both of these commands could still be running when the MoveBallToShooter command is reached.

Here is the code shown above represented as a flowchart. Any command in the main sequence (the sequence on the right here) that requires a subsystem in use by a parallel command will cause the parallel command to be canceled. For example, if the FireSequence required a subsystem in use by SetVirtualSetpoint, the SetVirtualSetpoint command will be canceled when FireSequence is scheduled.

Waiting for a Command



If there are two commands that need to complete before the following commands are scheduled, they can be put into a command group by themselves, adding both in parallel. Then that command group can be scheduled sequentially from an enclosing command group. When a command group is scheduled sequentially, the commands inside it will all finish before the next outer command is scheduled. In this way you can be sure that an action consisting of multiple parallel commands has completed before going on to the next command.

In this example you can see that the addition of a command group “Move to Bridge” containing the **Set Virtual Setpoint** and **Drive to Bridge** commands forces both to complete before the next commands are scheduled.

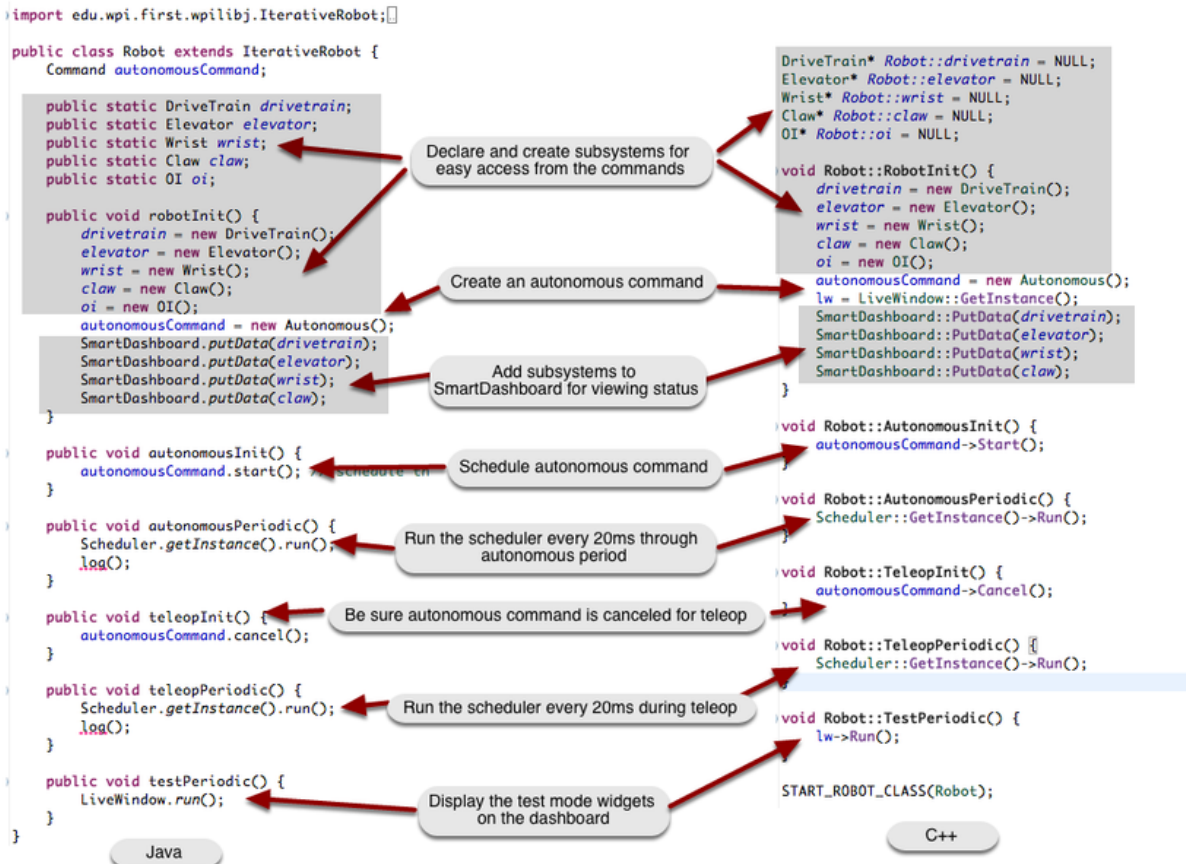
30.3.7 Scheduling Commands

Important: This documentation describes the use of the legacy command-based library. While this documentation has been preserved to help teams that have yet to do so, teams are strongly encouraged to migrate to the *new command-based library*.

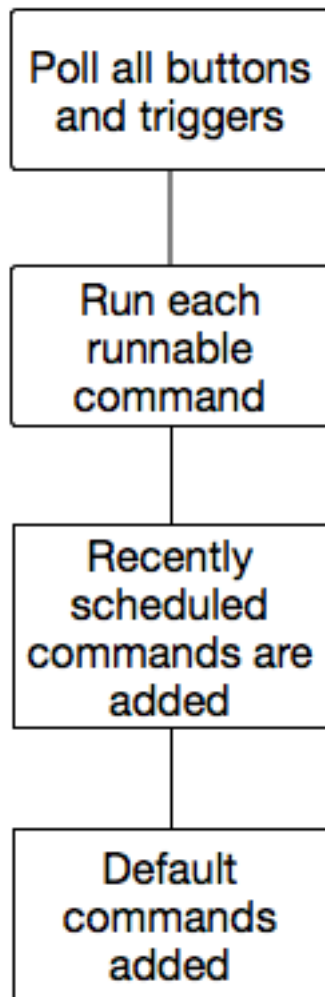
Commands are scheduled to run based on a number of factors such as triggers, default commands when no other running commands require a subsystem, a prior command in a group finishes, button presses, autonomous period starting, etc. Although many commands may be running virtually at the same time, there is only a single thread (the main robot thread). This is to reduce the complexity of synchronization between threads. There are threads that run in the system for systems like PID loops, communications, etc. but those are all self contained with very little interaction requiring complex synchronization. This makes the system much more robust and predictable.

This is accomplished by a class called Scheduler. It has a `run()` method that is called periodically (typically every 20ms in response to a driver station update) that tries to make progress on every command that is currently running. This is done by calling the `execute()` method on the command followed by the `isFinished()` method. If `isFinished()` returns true, the command is marked to be removed from execution on the next pass through the scheduler. So if there are a number of commands all scheduled to run at the same time, then every time the `Scheduler.run()` method is called, each of the active commands `execute()` and `isFinished()` methods are called. This has the same effect as using multiple threads.

Command-Based Program Anatomy



This shows a typical command-based Robot program and all the code needed to ensure that commands are scheduled correctly. The `Scheduler.run` method causes one pass through the scheduler which will let each currently active command run through its `execute()` and `isFinished()` methods. Ignore the `log()` methods in the Java example.

Command Life Cycle**Scheduler.run()**

The work in command-based programs occurs whenever the `Scheduler.Run` (C++) or `Scheduler.run` (Java) method is called. This is typically called on each driver station update which occurs every 20 ms or 50 times per second. The pseudo code illustrates what happens on each call to the run method.

1. Buttons and triggers are polled to see if the associated commands should be scheduled. If the trigger is true, the command is added to a list of commands that should be scheduled.
2. Loop through the list of all the commands that are currently runnable and call their `execute` and `isFinished` methods. Commands where the `isFinished` method returns true are removed from the list of currently running commands.
3. Loop through all the commands that have been scheduled to run in the previous steps. Those commands are added to the list of running commands.

4. Default commands are added for each subsystem that currently has no commands running that require that subsystem.

Optimizing Command Groups

Java

C++

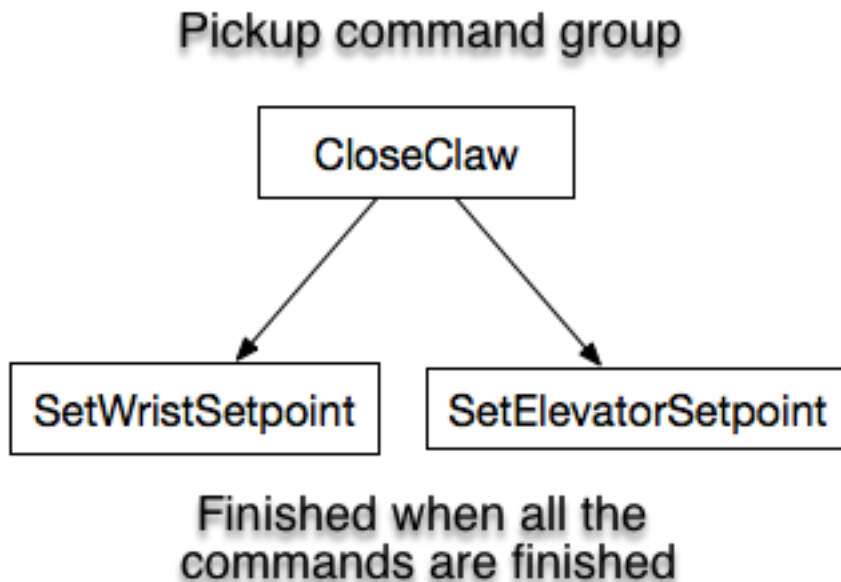
```
public class Pickup extends CommandGroup {
    public Pickup() {
        addSequential(new CloseClaw());
        addParallel(new SetWristSetpoint(-45));
        addSequential(new SetElevatorSetpoint(0.25));
    }
}
```

```
Pickup::Pickup() : CommandGroup("Pickup") {
    AddSequential(new CloseClaw());
    AddParallel(new SetWristSetpoint(-45));
    AddSequential(new SetElevatorSetpoint(0.25));
}
```

Once you have working commands that operate the mechanisms on your robot you can combine those commands into groups to make more complex actions. Commands can be added to command groups to execute sequentially or in parallel. Sequential commands wait until they are finished (isFinished method returns true) before running the next command in the group. Parallel commands start running, then immediately schedule the next command in the group.

It is important to notice that the commands are added to the group in the constructor. The command group is simply a list of command instances that run when scheduled and any parameters that are passed to the commands are evaluated during the constructor for the group.

Imagine that in a robot design, there is a claw, attached to a wrist joint and all of those on an elevator. When picking up something, the claw needs to close first before either the elevator or wrist can move otherwise the object may slip out of the claw. In the example shown above the CloseClaw command will be scheduled first. After it is finished (the claw is closed), the wrist will move to it's setpoint and in parallel, the elevator will move. This gets both the elevator and wrist moving simultaneously optimizing the time required to complete the task.

When do command groups finish?

A command group finishes when all the commands started in that group finish. This is true regardless of the type of commands that are added to the group. For example, if a number of commands are added in parallel and sequentially, the group is finished when all the commands added to the group are finished. As each command is added to a command group, it is put on a list. As those child commands finish, they are taken off the list. The command group is finished when the list of child commands is empty.

In the Pickup command shown in the example above, the command is finished when CloseClaw, SetWristSetpoint, and SetElevatorSetpoint all finish. It doesn't matter that some of the commands are sequential and some parallel.

Schedule a Command within a Running Command

Commands can be scheduled by calling the `start()` method (Java) or `Start()` method (C++) on a command instance. This will cause the command to be added to the currently running set of commands in the scheduler. This is often useful when a program needs to conditionally schedule one command or another. The newly scheduled command will be added to a list of new commands on this pass through the run method of the scheduler and actually will run the first time on the next pass through the run method. Newly created commands are never executed in the same call to the scheduler run method, always queued for the next call which usually occurs 20ms later.

Removing all Commands from the Scheduler

Java

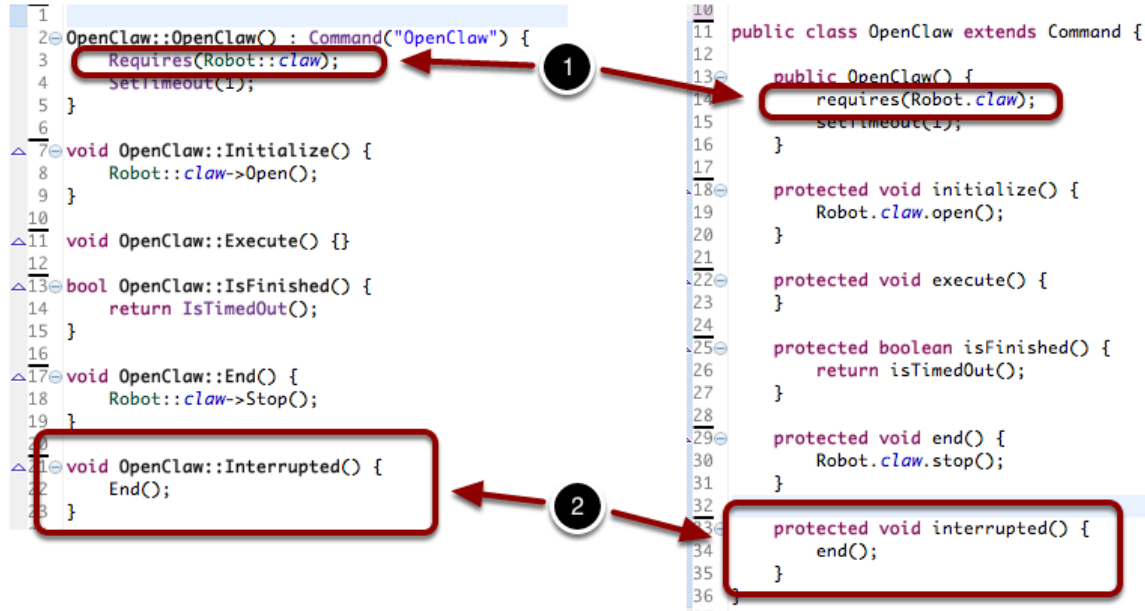
C++

```
Scheduler.getInstance().removeAll();
```

```
Scheduler::RemoveAll();
```

It is occasionally useful to make sure that there are no running commands in the scheduler. To remove all running commands use the `Scheduler.removeAll()` method (Java) or `Scheduler::RemoveAll()` method (C++). This will cause all currently running to have their `interrupted()` method (Java) or `Interrupted()` method (C++) called. Commands that have not yet started will have their `end()` method (Java) or `End()` method (C++) called.

The “requires” method



If you have multiple commands that use the same subsystem it makes sense that they don't run at the same time. For example, if there is a Claw subsystem with `OpenClaw` and `CloseClaw` commands, they can't both run at the same time. Each command that uses the Claw subsystem declares that by 1 calling the `requires()` method (Java) or `Requires()` method (C++). When one of the commands is running, say from a joystick button press, and you try to run another command that also requires the Claw, the second one preempts the first one. Suppose that `OpenClaw` was running, and you press the button to run the `CloseClaw` command. The `OpenClaw` command is interrupted - 2 its interrupted method is called on the next run cycle and the `CloseClaw` command is scheduled. If you think about it, this is almost always the desired behavior. If you pressed a button to start opening the claw and you change your mind and want to close it, it makes sense for the `OpenClaw` command to be stopped and the `CloseClaw` to be started.

A command may require many subsystems, for example a complex autonomous sequence might use a number of subsystems to complete its task.

Command groups automatically require all the subsystems for each of the commands in the group. There is no need to call the requires method for a group.

How are the requirements of a group evaluated?

The subsystems that a command group requires is the union of the set of subsystems that are required for all of the child commands. If 4 commands are added to a group, then the group will require all of the subsystems required by each of the 4 commands in the group. For example, if there are three commands scheduled in a group - the first requires subsystem A, the second requires subsystem B, and the third requires subsystems C and D. The group will require subsystems A, B, C, and D. If another command is started, say from a joystick button, that requires either A, B, C, or D it will interrupt the entire group including any parallel or sequential commands that might be running from that group.

Kinematics and Odometry

31.1 Introduction to Kinematics and The Chassis Speeds Class

31.1.1 What is kinematics?

The brand new kinematics suite contains classes for differential drive, swerve drive, and mecanum drive kinematics and odometry. The kinematics classes help convert between a universal `ChassisSpeeds` object, containing linear and angular velocities for a robot to usable speeds for each individual type of drivetrain i.e. left and right wheel speeds for a differential drive, four wheel speeds for a mecanum drive, or individual module states (speed and angle) for a swerve drive.

31.1.2 What is odometry?

Odometry involves using sensors on the robot to create an estimate of the position of the robot on the field. In FRC, these sensors are typically several encoders (the exact number depends on the drive type) and a gyroscope to measure robot angle. The odometry classes utilize the kinematics classes along with periodic user inputs about speeds (and angles in the case of swerve) to create an estimate of the robot's location on the field.

31.1.3 The Chassis Speeds Class

The `ChassisSpeeds` object is essential to the new WPILib kinematics and odometry suite. The `ChassisSpeeds` object represents the speeds of a robot chassis. This struct has three components:

- `vx`: The velocity of the robot in the x (forward) direction.
- `vy`: The velocity of the robot in the y (sideways) direction. (Positive values mean the robot is moving to the left).
- `omega`: The angular velocity of the robot in radians per second.

Note: A non-holonomic drivetrain (i.e. a drivetrain that cannot move sideways, ex: a differential drive) will have a v_y component of zero because of its inability to move sideways.

31.1.4 Constructing a ChassisSpeeds object

The constructor for the ChassisSpeeds object is very straightforward, accepting three arguments for v_x , v_y , and ω . In Java, v_x and v_y must be in meters per second. In C++, the units library may be used to provide a linear velocity using any linear velocity unit.

Java

C++

```
// The robot is moving at 3 meters per second forward, 2 meters  
// per second to the right, and rotating at half a rotation per  
// second counterclockwise.  
var speeds = new ChassisSpeeds(3.0, -2.0, Math.PI);
```

```
// The robot is moving at 3 meters per second forward, 2 meters  
// per second to the right, and rotating at half a rotation per  
// second counterclockwise.  
frc::ChassisSpeeds speeds{3.0_mps, -2.0_mps,  
    units::radians_per_second_t(wpi::math::pi)};
```

31.1.5 Creating a ChassisSpeeds Object from Field-Relative Speeds

A ChassisSpeeds object can also be created from a set of field-relative speeds when the robot angle is given. This converts a set of desired velocities relative to the field (for example, toward the opposite alliance station and toward the right field boundary) to a ChassisSpeeds object which represents speeds that are relative to the robot frame. This is useful for implementing field-oriented controls for a swerve or mecanum drive robot.

The static ChassisSpeeds.fromFieldRelativeSpeeds (Java) / ChassisSpeeds::FromFieldRelativeSpeeds (C++) method can be used to generate the ChassisSpeeds object from field-relative speeds. This method accepts the v_x (relative to the field), v_y (relative to the field), ω , and the robot angle.

Java

C++

```
// The desired field relative speed here is 2 meters per second  
// toward the opponent's alliance station wall, and 2 meters per  
// second toward the left field boundary. The desired rotation  
// is a quarter of a rotation per second counterclockwise. The current  
// robot angle is 45 degrees.  
ChassisSpeeds speeds = ChassisSpeeds.fromFieldRelativeSpeeds(  
    2.0, 2.0, Math.PI / 2.0, Rotation2d.fromDegrees(45.0));
```

```
// The desired field relative speed here is 2 meters per second  
// toward the opponent's alliance station wall, and 2 meters per  
// second toward the left field boundary. The desired rotation
```

(continues on next page)

(continued from previous page)

```
// is a quarter of a rotation per second counterclockwise. The current
// robot angle is 45 degrees.
frc::ChassisSpeeds speeds = frc::ChassisSpeeds::FromFieldRelativeSpeeds(
    2_mps, 2_mps, units::radians_per_second_t(wpi::math::pi / 2.0), Rotation2d(45_deg));
```

Note: The angular velocity is not explicitly stated to be “relative to the field” because the angular velocity is the same as measured from a field perspective or a robot perspective.

31.2 Differential Drive Kinematics

The `DifferentialDriveKinematics` class is a useful tool that converts between a `ChassisSpeeds` object and a `DifferentialDriveWheelSpeeds` object, which contains velocities for the left and right sides of a differential drive robot.

31.2.1 Constructing the Kinematics Object

The `DifferentialDriveKinematics` object accepts one constructor argument, which is the track width of the robot. This represents the distance between the two sets of wheels on a differential drive.

Note: In Java, the track width must be in meters. In C++, the units library can be used to pass in the track width using any length unit.

31.2.2 Converting Chassis Speeds to Wheel Speeds

The `toWheelSpeeds(ChassisSpeeds speeds)` (Java) / `ToWheelSpeeds(ChassisSpeeds speeds)` (C++) method should be used to convert a `ChassisSpeeds` object to a `DifferentialDriveWheelSpeeds` object. This is useful in situations where you have to convert a linear velocity (v_x) and an angular velocity (ω) to left and right wheel velocities.

Java

C++

```
// Creating my kinematics object: track width of 27 inches
DifferentialDriveKinematics kinematics =
    new DifferentialDriveKinematics(Units.inchesToMeters(27.0));

// Example chassis speeds: 2 meters per second linear velocity,
// 1 radian per second angular velocity.
var chassisSpeeds = new ChassisSpeeds(2.0, 0, 1.0);

// Convert to wheel speeds
DifferentialDriveWheelSpeeds wheelSpeeds = kinematics.toWheelSpeeds(chassisSpeeds);

// Left velocity
```

(continues on next page)

(continued from previous page)

```
double leftVelocity = wheelSpeeds.leftMetersPerSecond;

// Right velocity
double rightVelocity = wheelSpeeds.rightMetersPerSecond;

// Creating my kinematics object: track width of 27 inches
frc::DifferentialDriveKinematics kinematics{27_in};

// Example chassis speeds: 2 meters per second linear velocity,
// 1 radian per second angular velocity.
frc::ChassisSpeeds chassisSpeeds{2_mps, 0_mps, 1_rad_per_s};

// Convert to wheel speeds. Here, we can use C++17's structured bindings
// feature to automatically split the DifferentialDriveWheelSpeeds
// struct into left and right velocities.
auto [left, right] = kinematics.ToWheelSpeeds(chassisSpeeds);
```

31.2.3 Converting Wheel Speeds to Chassis Speeds

One can also use the kinematics object to convert individual wheel speeds (left and right) to a singular ChassisSpeeds object. The `toChassisSpeeds(DifferentialDriveWheelSpeeds speeds)` (Java) / `ToChassisSpeeds(DifferentialDriveWheelSpeeds speeds)` (C++) method should be used to achieve this.

Java

C++

```
// Creating my kinematics object: track width of 27 inches
DifferentialDriveKinematics kinematics =
    new DifferentialDriveKinematics(Units.inchesToMeters(27.0));

// Example differential drive wheel speeds: 2 meters per second
// for the left side, 3 meters per second for the right side.
var wheelSpeeds = new DifferentialDriveWheelSpeeds(2.0, 3.0);

// Convert to chassis speeds.
ChassisSpeeds chassisSpeeds = kinematics.toChassisSpeeds(wheelSpeeds);

// Linear velocity
double linearVelocity = chassisSpeeds.vxMetersPerSecond;

// Angular velocity
double angularVelocity = chassisSpeeds.omegaRadiansPerSecond;
```

```
// Creating my kinematics object: track width of 27 inches
frc::DifferentialDriveKinematics kinematics{27_in};

// Example differential drive wheel speeds: 2 meters per second
// for the left side, 3 meters per second for the right side.
frc::DifferentialDriveWheelSpeeds wheelSpeeds{2_mps, 3_mps};

// Convert to chassis speeds. Here we can use C++17's structured bindings
// feature to automatically split the ChassisSpeeds struct into its 3 components.
```

(continues on next page)

(continued from previous page)

```
// Note that because a differential drive is non-holonomic, the vy variable
// will be equal to zero.
auto [linearVelocity, vy, angularVelocity] = kinematics.ToChassisSpeeds(wheelSpeeds);
```

31.3 Differential Drive Odometry

A user can use the differential drive kinematics classes in order to perform *odometry*. WPILib contains a `DifferentialDriveOdometry` class that can be used to track the position of a differential drive robot on the field.

Note: Because this method only uses encoders and a gyro, the estimate of the robot's position on the field will drift over time, especially as your robot comes into contact with other robots during gameplay. However, odometry is usually very accurate during the autonomous period.

31.3.1 Creating the Odometry Object

The `DifferentialDriveOdometry` class requires one mandatory argument and one optional argument. The mandatory argument is the angle reported by your gyroscope (as a `Rotation2d`). The optional argument is the starting pose of your robot on the field (as a `Pose2d`). By default, the robot will start at $x = 0$, $y = 0$, $\theta = 0$.

Note: 0 degrees / radians represents the robot angle when the robot is facing directly toward your opponent's alliance station. As your robot turns to the left, your gyroscope angle should increase. By default, WPILib gyros exhibit the opposite behavior, so you should negate the gyro angle.

Important: The encoder positions must be reset to zero before constructing the `DifferentialDriveOdometry` class.

Java

C++

```
// Creating my odometry object. Here,
// our starting pose is 5 meters along the long end of the field and in the
// center of the field along the short end, facing forward.
DifferentialDriveOdometry m_odometry = new DifferentialDriveOdometry(
    getGyroHeading(), new Pose2d(5.0, 13.5, new Rotation2d()));
```

```
// Creating my odometry object. Here,
// our starting pose is 5 meters along the long end of the field and in the
// center of the field along the short end, facing forward.
frc::DifferentialDriveOdometry m_odometry{GetGyroHeading(),
    frc::Pose2d{5_m, 13.5_m, 0_rad}};
```

31.3.2 Updating the Robot Pose

The update method can be used to update the robot's position on the field. This method must be called periodically, preferably in the `periodic()` method of a *Subsystem*. The update method returns the new updated pose of the robot. This method takes in the gyro angle of the robot, along with the left encoder distance and right encoder distance.

Note: The encoder distances in Java must be in meters. In C++, the units library can be used to represent the distance using any linear distance unit. If the robot is moving forward in a straight line, **both** distances (left and right) must be positive.

Java

C++

```
@Override
public void periodic() {
    // Get my gyro angle. We are negating the value because gyros return positive
    // values as the robot turns clockwise. This is not standard convention that is
    // used by the WPILib classes.
    var gyroAngle = Rotation2d.fromDegrees(-m_gyro.getAngle());

    // Update the pose
    m_pose = m_odometry.update(gyroAngle, m_leftEncoder.getDistance(), m_rightEncoder.
        ↪getDistance());
}
```

```
void Periodic() override {
    // Get my gyro angle. We are negating the value because gyros return positive
    // values as the robot turns clockwise. This is not standard convention that is
    // used by the WPILib classes.
    frc::Rotation2d gyroAngle{units::degree_t(-m_gyro.GetAngle())};

    // Update the pose
    m_pose = m_odometry.Update(gyroAngle, m_leftEncoder.GetDistance(), m_rightEncoder.
        ↪GetDistance());
}
```

31.3.3 Resetting the Robot Pose

The robot pose can be reset via the `resetPose` method. This method accepts two arguments – the new field-relative pose and the current gyro angle.

Important: If at any time, you decide to reset your gyroscope, the `resetPose` method **MUST** be called with the new gyro angle. Furthermore, the encoders must also be reset to zero when resetting the pose.

Note: A full example of a differential drive robot with odometry is available here: [C++ / Java](#).

In addition, the `GetPose` (C++) / `getPoseMeters` (Java) methods can be used to retrieve the current robot pose without an update.

31.4 Swerve Drive Kinematics

The `SwerveDriveKinematics` class is a useful tool that converts between a `ChassisSpeeds` object and several `SwerveModuleState` objects, which contains velocities and angles for each swerve module of a swerve drive robot.

31.4.1 The swerve module state class

The `SwerveModuleState` class contains information about the velocity and angle of a singular module of a swerve drive. The constructor for a `SwerveModuleState` takes in two arguments, the velocity of the wheel on the module, and the angle of the module.

Note: In Java, the velocity of the wheel must be in meters per second. In C++, the units library can be used to provide the velocity using any linear velocity unit.

Note: An angle of 0 corresponds to the modules facing forward.

31.4.2 Constructing the kinematics object

The `SwerveDriveKinematics` class accepts a variable number of constructor arguments, with each argument being the location of a swerve module relative to the robot center (as a `Translation2d`). The number of constructor arguments corresponds to the number of swerve modules.

Note: A swerve drive must have 2 or more modules.

Note: In C++, the class is templated on the number of modules. Therefore, when constructing a `SwerveDriveKinematics` object as a member variable of a class, the number of modules must be passed in as a template argument. For example, for a typical swerve drive with four modules, the kinematics object must be constructed as follows: `frc::SwerveDriveKinematics<4>m_kinematics{...}`.

The locations for the modules must be relative to the center of the robot. Positive x values represent moving toward the front of the robot whereas positive y values represent moving toward the left of the robot.

Java

C++

```
// Locations for the swerve drive modules relative to the robot center.
Translation2d m_frontLeftLocation = new Translation2d(0.381, 0.381);
Translation2d m_frontRightLocation = new Translation2d(0.381, -0.381);
Translation2d m_backLeftLocation = new Translation2d(-0.381, 0.381);
Translation2d m_backRightLocation = new Translation2d(-0.381, -0.381);
```

(continues on next page)

(continued from previous page)

```
// Creating my kinematics object using the module locations
SwerveDriveKinematics m_kinematics = new SwerveDriveKinematics(
    m_frontLeftLocation, m_frontRightLocation, m_backLeftLocation, m_backRightLocation
);
```

```
// Locations for the swerve drive modules relative to the robot center.
frc::Translation2d m_frontLeftLocation{0.381_m, 0.381_m};
frc::Translation2d m_frontRightLocation{0.381_m, -0.381_m};
frc::Translation2d m_backLeftLocation{-0.381_m, 0.381_m};
frc::Translation2d m_backRightLocation{-0.381_m, -0.381_m};
```

```
// Creating my kinematics object using the module locations.
frc::SwerveDriveKinematics<4> m_kinematics{
    m_frontLeftLocation, m_frontRightLocation, m_backLeftLocation,
    m_backRightLocation};
```

31.4.3 Converting chassis speeds to module states

The `toSwerveModuleStates(ChassisSpeeds speeds)` (Java) / `ToSwerveModuleStates(ChassisSpeeds speeds)` (C++) method should be used to convert a `ChassisSpeeds` object to an array of `SwerveModuleState` objects. This is useful in situations where you have to convert a forward velocity, sideways velocity, and an angular velocity into individual module states.

The elements in the array that is returned by this method are the same order in which the kinematics object was constructed. For example, if the kinematics object was constructed with the front left module location, front right module location, back left module location, and the back right module location in that order, the elements in the array would be the front left module state, front right module state, back left module state, and back right module state in that order.

Java

C++

```
// Example chassis speeds: 1 meter per second forward, 3 meters
// per second to the left, and rotation at 1.5 radians per second
// counterclockwise.
ChassisSpeeds speeds = new ChassisSpeeds(1.0, 3.0, 1.5);

// Convert to module states
SwerveModuleState[] moduleStates = kinematics.toSwerveModuleStates(speeds);

// Front left module state
SwerveModuleState frontLeft = moduleStates[0];

// Front right module state
SwerveModuleState frontRight = moduleStates[1];

// Back left module state
SwerveModuleState backLeft = moduleStates[2];

// Back right module state
SwerveModuleState backRight = moduleStates[3];
```

```
// Example chassis speeds: 1 meter per second forward, 3 meters
// per second to the left, and rotation at 1.5 radians per second
// counterclockwise.
frc::ChassisSpeeds speeds{1_mps, 3_mps, 1.5_rad_per_s};

// Convert to module states. Here, we can use C++17's structured
// bindings feature to automatically split up the array into its
// individual SwerveModuleState components.
auto [fl, fr, bl, br] = kinematics.ToSwerveModuleStates(speeds);
```

Module angle optimization

The `SwerveModuleState` class contains a static `optimize()` (Java) / `Optimize()` (C++) method that is used to “optimize” the speed and angle setpoint of a given `SwerveModuleState` to minimize the change in heading. For example, if the angular setpoint of a certain module from inverse kinematics is 90 degrees, but your current angle is -89 degrees, this method will automatically negate the speed of the module setpoint and make the angular setpoint -90 degrees to reduce the distance the module has to travel.

This method takes two parameters: the desired state (usually from the `toSwerveModuleStates` method) and the current angle. It will return the new optimized state which you can use as the setpoint in your feedback control loop.

Java

C++

```
var frontLeftOptimized = SwerveModuleState.optimize(frontLeft,
    new Rotation2d(m_turningEncoder.getDistance()));
```

```
auto flOptimized = frc::SwerveModuleState::Optimize(fl,
    units::radian_t(m_turningEncoder.GetDistance()));
```

Field-oriented drive

Recall that a `ChassisSpeeds` object can be created from a set of desired field-oriented speeds. This feature can be used to get module states from a set of desired field-oriented speeds.

Java

C++

```
// The desired field relative speed here is 2 meters per second
// toward the opponent's alliance station wall, and 2 meters per
// second toward the left field boundary. The desired rotation
// is a quarter of a rotation per second counterclockwise. The current
// robot angle is 45 degrees.
ChassisSpeeds speeds = ChassisSpeeds.fromFieldRelativeSpeeds(
    2.0, 2.0, Math.PI / 2.0, Rotation2d.fromDegrees(45.0));

// Now use this in our kinematics
SwerveModuleState[] moduleStates = kinematics.toSwerveModuleStates(speeds);
```

```
// The desired field relative speed here is 2 meters per second
// toward the opponent's alliance station wall, and 2 meters per
// second toward the left field boundary. The desired rotation
// is a quarter of a rotation per second counterclockwise. The current
// robot angle is 45 degrees.
frc::ChassisSpeeds speeds = frc::ChassisSpeeds::FromFieldRelativeSpeeds(
    2_mps, 2_mps, units::radians_per_second_t(wpi::math::pi / 2.0), Rotation2d(45_deg));

// Now use this in our kinematics
auto [fl, fr, bl, br] = kinematics.ToSwerveModuleStates(speeds);
```

Using custom centers of rotation

Sometimes, rotating around one specific corner might be desirable for certain evasive maneuvers. This type of behavior is also supported by the WPILib classes. The same `ToSwerveModuleStates()` method accepts a second parameter for the center of rotation (as a `Translation2d`). Just like the wheel locations, the `Translation2d` representing the center of rotation should be relative to the robot center.

Note: Because all robots are a rigid frame, the provided `vx` and `vy` velocities from the `ChassisSpeeds` object will still apply for the entirety of the robot. However, the `omega` from the `ChassisSpeeds` object will be measured from the center of rotation.

For example, one can set the center of rotation on a certain module and if the provided `ChassisSpeeds` object has a `vx` and `vy` of zero and a non-zero `omega`, the robot will appear to rotate around that particular swerve module.

31.4.4 Converting module states to chassis speeds

One can also use the `kinematics` object to convert an array of `SwerveModuleState` objects to a singular `ChassisSpeeds` object. The `toChassisSpeeds(SwerveModuleState... states)` (Java) / `ToChassisSpeeds(SwerveModuleState... states)` (C++) method can be used to achieve this.

Java

C++

```
// Example module states
var frontLeftState = new SwerveModuleState(23.43, Rotation2d.fromDegrees(-140.19));
var frontRightState = new SwerveModuleState(23.43, Rotation2d.fromDegrees(-39.81));
var backLeftState = new SwerveModuleState(54.08, Rotation2d.fromDegrees(-109.44));
var backRightState = new SwerveModuleState(54.08, Rotation2d.fromDegrees(-70.56));

// Convert to chassis speeds
ChassisSpeeds chassisSpeeds = kinematics.toChassisSpeeds(
    frontLeftState, frontRightState, backLeftState, backRightState);

// Getting individual speeds
double forward = chassisSpeeds.vxMetersPerSecond;
double sideways = chassisSpeeds.vyMetersPerSecond;
double angular = chassisSpeeds.omegaRadiansPerSecond;
```



```
// Example module States
frc::SwerveModuleState frontLeftState{23.43_mps, Rotation2d(-140.19_deg)};
frc::SwerveModuleState frontRightState{23.43_mps, Rotation2d(-39.81_deg)};
frc::SwerveModuleState backLeftState{54.08_mps, Rotation2d(-109.44_deg)};
frc::SwerveModuleState backRightState{54.08_mps, Rotation2d(-70.56_deg)};

// Convert to chassis speeds. Here, we can use C++17's structured bindings
// feature to automatically break up the ChassisSpeeds struct into its
// three components.
auto [forward, sideways, angular] = kinematics.ToChassisSpeeds(
    frontLeftState, frontRightState, backLeftState, backRightState);
```

31.5 Swerve Drive Odometry

A user can use the swerve drive kinematics classes in order to perform *odometry*. WPILib contains a `SwerveDriveOdometry` class that can be used to track the position of a swerve drive robot on the field.

Note: Because this method only uses encoders and a gyro, the estimate of the robot's position on the field will drift over time, especially as your robot comes into contact with other robots during gameplay. However, odometry is usually very accurate during the autonomous period.

31.5.1 Creating the odometry object

The `SwerveDriveOdometry<int NumModules>` class requires one template argument (only C++), two mandatory arguments, and one optional argument. The template argument (only C++) is an integer representing the number of swerve modules. The mandatory arguments are the kinematics object that represents your swerve drive (in the form of a `SwerveDriveKinematics` class) and the angle reported by your gyroscope (as a `Rotation2d`). The third optional argument is the starting pose of your robot on the field (as a `Pose2d`). By default, the robot will start at $x = 0$, $y = 0$, $\theta = 0$.

Note: 0 degrees / radians represents the robot angle when the robot is facing directly toward your opponent's alliance station. As your robot turns to the left, your gyroscope angle should increase. By default, WPILib gyros exhibit the opposite behavior, so you should negate the gyro angle.

Java

C++

```
// Locations for the swerve drive modules relative to the robot center.
Translation2d m_frontLeftLocation = new Translation2d(0.381, 0.381);
Translation2d m_frontRightLocation = new Translation2d(0.381, -0.381);
Translation2d m_backLeftLocation = new Translation2d(-0.381, 0.381);
Translation2d m_backRightLocation = new Translation2d(-0.381, -0.381);
```

(continues on next page)

(continued from previous page)

```
// Creating my kinematics object using the module locations
SwerveDriveKinematics m_kinematics = new SwerveDriveKinematics(
    m_frontLeftLocation, m_frontRightLocation, m_backLeftLocation, m_backRightLocation
);
```

```
// Creating my odometry object from the kinematics object. Here,
// our starting pose is 5 meters along the long end of the field and in the
// center of the field along the short end, facing forward.
SwerveDriveOdometry m_odometry = new SwerveDriveOdometry(m_kinematics,
    getGyroHeading(), new Pose2d(5.0, 13.5, new Rotation2d()));
```

```
// Locations for the swerve drive modules relative to the robot center.
frc::Translation2d m_frontLeftLocation{0.381_m, 0.381_m};
frc::Translation2d m_frontRightLocation{0.381_m, -0.381_m};
frc::Translation2d m_backLeftLocation{-0.381_m, 0.381_m};
frc::Translation2d m_backRightLocation{-0.381_m, -0.381_m};
```

```
// Creating my kinematics object using the module locations.
frc::SwerveDriveKinematics<4> m_kinematics{
    m_frontLeftLocation, m_frontRightLocation, m_backLeftLocation,
    m_backRightLocation};

// Creating my odometry object from the kinematics object. Here,
// our starting pose is 5 meters along the long end of the field and in the
// center of the field along the short end, facing forward.
frc::SwerveDriveOdometry<4> m_odometry{m_kinematics, GetGyroHeading(),
    frc::Pose2d{5_m, 13.5_m, 0_rad}};
```

31.5.2 Updating the robot pose

The update method of the odometry class updates the robot position on the field. The update method takes in the gyro angle of the robot, along with a series of module states (speeds and angles) in the form of a `SwerveModuleState` each. It is important that the order in which you pass the `SwerveModuleState` objects is the same as the order in which you created the kinematics object.

This update method must be called periodically, preferably in the `periodic()` method of a *Subsystem*. The update method returns the new updated pose of the robot.

Java

C++

```
@Override
public void periodic() {
    // Get my gyro angle. We are negating the value because gyros return positive
    // values as the robot turns clockwise. This is not standard convention that is
    // used by the WPILib classes.
    var gyroAngle = Rotation2d.fromDegrees(-m_gyro.getAngle());

    // Update the pose
    m_pose = m_odometry.update(gyroAngle, m_frontLeftModule.getState(), m_
    frontRightModule.getState(),
    m_backLeftModule.getState(), m_backRightModule.getState());
}
```

```

void Periodic() override {
    // Get my gyro angle. We are negating the value because gyros return positive
    // values as the robot turns clockwise. This is not standard convention that is
    // used by the WPILib classes.
    frc::Rotation2d gyroAngle{units::degree_t(-m_gyro.GetAngle())};

    // Update the pose
    m_pose = m_odometry.Update(gyroAngle, m_frontLeftModule.GetState(), m_
    ↪ frontRightModule.GetState(),
        m_backLeftModule.GetState(), m_backRightModule.GetState());
}

```

31.5.3 Resetting the Robot Pose

The robot pose can be reset via the `resetPose` method. This method accepts two arguments – the new field-relative pose and the current gyro angle.

Important: If at any time, you decide to reset your gyroscope, the `resetPose` method **MUST** be called with the new gyro angle.

Note: The implementation of `getState()` / `GetState()` above is left to the user. The idea is to get the module state (speed and angle) from each module. For a full example, see here: [C++](#) / [Java](#).

In addition, the `GetPose` (C++) / `getPoseMeters` (Java) methods can be used to retrieve the current robot pose without an update.

31.6 Mecanum Drive Kinematics

The `MecanumDriveKinematics` class is a useful tool that converts between a `ChassisSpeeds` object and a `MecanumDriveWheelSpeeds` object, which contains velocities for each of the four wheels on a mecanum drive.

31.6.1 Constructing the Kinematics Object

The `MecanumDriveKinematics` class accepts four constructor arguments, with each argument being the location of a wheel relative to the robot center (as a `Translation2d`). The order for the arguments is front left, front right, back left, and back right. The locations for the wheels must be relative to the center of the robot. Positive x values represent moving toward the front of the robot whereas positive y values represent moving toward the left of the robot.

Java

C++

```
// Locations of the wheels relative to the robot center.
Translation2d m_frontLeftLocation = new Translation2d(0.381, 0.381);
Translation2d m_frontRightLocation = new Translation2d(0.381, -0.381);
Translation2d m_backLeftLocation = new Translation2d(-0.381, 0.381);
Translation2d m_backRightLocation = new Translation2d(-0.381, -0.381);

// Creating my kinematics object using the wheel locations.
MecanumDriveKinematics m_kinematics = new MecanumDriveKinematics(
    m_frontLeftLocation, m_frontRightLocation, m_backLeftLocation, m_backRightLocation
);
```

```
// Locations of the wheels relative to the robot center.
frc::Translation2d m_frontLeftLocation{0.381_m, 0.381_m};
frc::Translation2d m_frontRightLocation{0.381_m, -0.381_m};
frc::Translation2d m_backLeftLocation{-0.381_m, 0.381_m};
frc::Translation2d m_backRightLocation{-0.381_m, -0.381_m};

// Creating my kinematics object using the wheel locations.
frc::MecanumDriveKinematics m_kinematics{
    m_frontLeftLocation, m_frontRightLocation, m_backLeftLocation,
    m_backRightLocation};
```

31.6.2 Converting Chassis Speeds to Wheel Speeds

The `toWheelSpeeds(ChassisSpeeds speeds)` (Java) / `ToWheelSpeeds(ChassisSpeeds speeds)` (C++) method should be used to convert a `ChassisSpeeds` object to a `MecanumDriveWheelSpeeds` object. This is useful in situations where you have to convert a forward velocity, sideways velocity, and an angular velocity into individual wheel speeds.

Java

C++

```
// Example chassis speeds: 1 meter per second forward, 3 meters
// per second to the left, and rotation at 1.5 radians per second
// counterclockwise.
ChassisSpeeds speeds = new ChassisSpeeds(1.0, 3.0, 1.5);

// Convert to wheel speeds
MecanumDriveWheelSpeeds wheelSpeeds = kinematics.toWheelSpeeds(speeds);

// Get the individual wheel speeds
double frontLeft = wheelSpeeds.frontLeftMetersPerSecond
double frontRight = wheelSpeeds.frontRightMetersPerSecond
double backLeft = wheelSpeeds.rearLeftMetersPerSecond
double backRight = wheelSpeeds.rearRightMetersPerSecond
```

```
// Example chassis speeds: 1 meter per second forward, 3 meters
// per second to the left, and rotation at 1.5 radians per second
// counterclockwise.
frc::ChassisSpeeds speeds{1_mps, 3_mps, 1.5_rad_per_s};

// Convert to wheel speeds. Here, we can use C++17's structured
// bindings feature to automatically split up the MecanumDriveWheelSpeeds
```

(continues on next page)

(continued from previous page)

```
// struct into it's individual components
auto [fl, fr, bl, br] = kinematics.ToWheelSpeeds(speeds);
```

Field-oriented drive

Recall that a `ChassisSpeeds` object can be created from a set of desired field-oriented speeds. This feature can be used to get wheel speeds from a set of desired field-oriented speeds.

Java

C++

```
// The desired field relative speed here is 2 meters per second
// toward the opponent's alliance station wall, and 2 meters per
// second toward the left field boundary. The desired rotation
// is a quarter of a rotation per second counterclockwise. The current
// robot angle is 45 degrees.
ChassisSpeeds speeds = ChassisSpeeds.fromFieldRelativeSpeeds(
    2.0, 2.0, Math.PI / 2.0, Rotation2d.fromDegrees(45.0));

// Now use this in our kinematics
MecanumDriveWheelSpeeds wheelSpeeds = kinematics.toWheelSpeeds(speeds);
```

```
// The desired field relative speed here is 2 meters per second
// toward the opponent's alliance station wall, and 2 meters per
// second toward the left field boundary. The desired rotation
// is a quarter of a rotation per second counterclockwise. The current
// robot angle is 45 degrees.
frc::ChassisSpeeds speeds = frc::ChassisSpeeds::FromFieldRelativeSpeeds(
    2_mps, 2_mps, units::radians_per_second_t(wpi::math::pi / 2.0), Rotation2d(45_deg));

// Now use this in our kinematics
auto [fl, fr, bl, br] = kinematics.ToWheelSpeeds(speeds);
```

Using custom centers of rotation

Sometimes, rotating around one specific corner might be desirable for certain evasive maneuvers. This type of behavior is also supported by the WPILib classes. The same `ToWheelSpeeds()` method accepts a second parameter for the center of rotation (as a `Translation2d`). Just like the wheel locations, the `Translation2d` representing the center of rotation should be relative to the robot center.

Note: Because all robots are a rigid frame, the provided v_x and v_y velocities from the `ChassisSpeeds` object will still apply for the entirety of the robot. However, the ω from the `ChassisSpeeds` object will be measured from the center of rotation.

For example, one can set the center of rotation on a certain wheel and if the provided `ChassisSpeeds` object has a v_x and v_y of zero and a non-zero ω , the robot will appear to rotate around that particular wheel.

31.6.3 Converting wheel speeds to chassis speeds

One can also use the kinematics object to convert a `MecanumDriveWheelSpeeds` object to a singular `ChassisSpeeds` object. The `toChassisSpeeds(MecanumDriveWheelSpeeds speeds)` (Java) / `ToChassisSpeeds(MecanumDriveWheelSpeeds speeds)` (C++) method can be used to achieve this.

Java

C++

```
// Example wheel speeds
var wheelSpeeds = new MecanumDriveWheelSpeeds(-17.67, 20.51, -13.44, 16.26);

// Convert to chassis speeds
ChassisSpeeds chassisSpeeds = kinematics.toChassisSpeeds(wheelSpeeds);

// Getting individual speeds
double forward = chassisSpeeds.vxMetersPerSecond;
double sideways = chassisSpeeds.vyMetersPerSecond;
double angular = chassisSpeeds.omegaRadiansPerSecond;
```

```
// Example wheel speeds
frc::MecanumDriveWheelSpeeds wheelSpeeds{-17.67_mps, 20.51_mps, -13.44_mps, 16.26_mps}
→;

// Convert to chassis speeds. Here, we can use C++17's structured bindings
// feature to automatically break up the ChassisSpeeds struct into its
// three components.
auto [forward, sideways, angular] = kinematics.ToChassisSpeeds(wheelSpeeds);
```

31.7 Mecanum Drive Odometry

A user can use the mecanum drive kinematics classes in order to perform *odometry*. WPILib contains a `MecanumDriveOdometry` class that can be used to track the position of a mecanum drive robot on the field.

Note: Because this method only uses encoders and a gyro, the estimate of the robot's position on the field will drift over time, especially as your robot comes into contact with other robots during gameplay. However, odometry is usually very accurate during the autonomous period.

31.7.1 Creating the odometry object

The `MecanumDriveOdometry` class requires two mandatory arguments and one optional argument. The mandatory arguments are the kinematics object that represents your mecanum drive (in the form of a `MecanumDriveKinematics` class) and the angle reported by your gyroscope (as a `Rotation2d`). The third optional argument is the starting pose of your robot on the field (as a `Pose2d`). By default, the robot will start at $x = 0$, $y = 0$, $\theta = 0$.

Note: 0 degrees / radians represents the robot angle when the robot is facing directly toward your opponent's alliance station. As your robot turns to the left, your gyroscope angle should increase. By default, WPILib gyros exhibit the opposite behavior, so you should negate the gyro angle.

Java

C++

```
// Locations of the wheels relative to the robot center.
Translation2d m_frontLeftLocation = new Translation2d(0.381, 0.381);
Translation2d m_frontRightLocation = new Translation2d(0.381, -0.381);
Translation2d m_backLeftLocation = new Translation2d(-0.381, 0.381);
Translation2d m_backRightLocation = new Translation2d(-0.381, -0.381);

// Creating my kinematics object using the wheel locations.
MecanumDriveKinematics m_kinematics = new MecanumDriveKinematics(
    m_frontLeftLocation, m_frontRightLocation, m_backLeftLocation, m_backRightLocation
);

// Creating my odometry object from the kinematics object. Here,
// our starting pose is 5 meters along the long end of the field and in the
// center of the field along the short end, facing forward.
MecanumDriveOdometry m_odometry = new MecanumDriveOdometry(m_kinematics,
    getGyroHeading(), new Pose2d(5.0, 13.5, new Rotation2d()));
```

```
// Locations of the wheels relative to the robot center.
frc::Translation2d m_frontLeftLocation{0.381_m, 0.381_m};
frc::Translation2d m_frontRightLocation{0.381_m, -0.381_m};
frc::Translation2d m_backLeftLocation{-0.381_m, 0.381_m};
frc::Translation2d m_backRightLocation{-0.381_m, -0.381_m};

// Creating my kinematics object using the wheel locations.
frc::MecanumDriveKinematics m_kinematics{
    m_frontLeftLocation, m_frontRightLocation, m_backLeftLocation,
    m_backRightLocation};

// Creating my odometry object from the kinematics object. Here,
// our starting pose is 5 meters along the long end of the field and in the
// center of the field along the short end, facing forward.
frc::MecanumDriveOdometry m_odometry{m_kinematics, GetGyroHeading(),
    frc::Pose2d{5_m, 13.5_m, 0_rad}};
```

31.7.2 Updating the robot pose

The update method of the odometry class updates the robot position on the field. The update method takes in the gyro angle of the robot, along with a `MecanumDriveWheelSpeeds` object representing the speed of each of the 4 wheels on the robot. This update method must be called periodically, preferably in the `periodic()` method of a *Subsystem*. The update method returns the new updated pose of the robot.

Note: The `MecanumDriveWheelSpeeds` class in Java must be constructed with each wheel speed in meters per second. In C++, the units library must be used to represent your wheel speeds.

Java

C++

```
@Override
public void periodic() {
    // Get my wheel speeds
    var wheelSpeeds = new MecanumDriveWheelSpeeds(
        m_frontLeftEncoder.getRate(), m_frontRightEncoder.getRate(),
        m_backLeftEncoder.getRate(), m_backRightEncoder.getRate());

    // Get my gyro angle. We are negating the value because gyros return positive
    // values as the robot turns clockwise. This is not standard convention that is
    // used by the WPILib classes.
    var gyroAngle = Rotation2d.fromDegrees(-m_gyro.getAngle());

    // Update the pose
    m_pose = m_odometry.update(gyroAngle, wheelSpeeds);
}
```

```
void Periodic() override {
    // Get my wheel speeds
    frc::MecanumDriveWheelSpeeds wheelSpeeds{
        units::meters_per_second_t(m_frontLeftEncoder.GetRate()),
        units::meters_per_second_t(m_frontRightEncoder.GetRate()),
        units::meters_per_second_t(m_backLeftEncoder.GetRate()),
        units::meters_per_second_t(m_backRightEncoder.GetRate())};

    // Get my gyro angle. We are negating the value because gyros return positive
    // values as the robot turns clockwise. This is not standard convention that is
    // used by the WPILib classes.
    frc::Rotation2d gyroAngle{units::degree_t(-m_gyro.GetAngle())};

    // Update the pose
    m_pose = m_odometry.Update(gyroAngle, wheelSpeeds);
}
```


31.7.3 Resetting the Robot Pose

The robot pose can be reset via the `resetPose` method. This method accepts two arguments – the new field-relative pose and the current gyro angle.

Important: If at any time, you decide to reset your gyroscope, the `resetPose` method **MUST** be called with the new gyro angle.

Note: A full example of a mecanum drive robot with odometry is available here: [C++ / Java](#).

In addition, the `GetPose` (C++) / `getPoseMeters` (Java) methods can be used to retrieve the current robot pose without an update.

32.1 What is NetworkTables

NetworkTables is an implementation of a distributed “dictionary”. That is named values are created either on the robot, driver station, or potentially an attached coprocessor, and the values are automatically distributed to all the other participants. For example, a driver station laptop might receive camera images over the network, perform some vision processing algorithm, and come up with some values to sent back to the robot. The values might be an X, Y, and Distance. By writing these results to NetworkTables values called “X”, “Y”, and “Distance” they can be read by the robot shortly after being written. Then the robot can act upon them.

NetworkTables can be used by programs on the robot in either C++, Java or LabVIEW and is built into each version of WPILib.

32.1.1 NetworkTables organization

Data is organized in NetworkTables in a hierarchy much like a directory on disk with folders and files. For any instance of NetworkTables there can be multiple values and subtables that may be nested in whatever way fits the data organization desired. Subtables actually are represented as a long key with slashes (/) separating the nested subtable and value key names. Each value has a key associated with it that is used to retrieve the value. For example, you might have a table called **datatable** as shown in the following examples. Within a **datatable** there are two keys, X and Y and their associated values. The OutlineViewer is a good utility for exploring the values stored in NetworkTables while a program is running.

Data types for NetworkTables are either boolean, numeric, or string. Numeric values are written as double precision values. In addition you can have arrays of any of those types to ensure that multiple data items are delivered consistently. There is also the option of storing raw data which can be used for representing structured data.

There are some default tables that are created automatically when the program starts up:

Table name	Use
/Smart-Dash-board	Used to store values written to the SmartDashboard or Shuffleboard using the SmartDashboard.put() set of methods.
/LiveWin-dow	Used to store Test mode (Test on the Driver Station) values. Typically these are Subsystems and the associated sensors and actuators.
/FMSInfo	Information about the currently running match that comes from the Driver Station and the Field Management System

32.1.2 Writing a simple NetworkTables program

The NetworkTables classes are instantiated automatically when your program starts. To access the instance of NetworkTables do call methods to read and write the getDefault() method can be used to get the default instance.

Java

C++

```
package edu.wpi.first.wpilibj.templates;

import edu.wpi.first.wpilibj.TimedRobot;
import edu.wpi.first.networktables.NetworkTable;
import edu.wpi.first.networktables.NetworkTableEntry;
import edu.wpi.first.networktables.NetworkTableInstance;

public class EasyNetworkTableExample extends TimedRobot {
    NetworkTableEntry xEntry;
    NetworkTableEntry yEntry;

    public void robotInit() {
        //Get the default instance of NetworkTables that was created automatically
        //when your program starts
        NetworkTableInstance inst = NetworkTableInstance.getDefault();

        //Get the table within that instance that contains the data. There can
        //be as many tables as you like and exist to make it easier to organize
        //your data. In this case, it's a table called datatable.
        NetworkTable table = inst.getTable("datatable");

        //Get the entries within that table that correspond to the X and Y values
        //for some operation in your program.
        xEntry = table.getEntry("X");
        yEntry = table.getEntry("Y");
    }

    double x = 0;
    double y = 0;

    public void teleopPeriodic() {
        //Using the entry objects, set the value to a double that is constantly
        //increasing. The keys are actually "/datatable/X" and "/datatable/Y".
        //If they don't already exist, the key/value pair is added.
        xEntry.setDouble(x);
    }
}
```

(continues on next page)

(continued from previous page)

```

        yEntry.setDouble(y);
        x += 0.05;
        y += 1.0;
    }
}

```

```

#include "TimedRobot.h"
#include "networktables/NetworkTable.h"
#include "networktables/NetworkTableEntry.h"
#include "networktables/NetworkTableInstance.h"

class EasyNetworkExample : public frc::TimedRobot {
public:
    nt::NetworkTableEntry xEntry;
    nt::NetworkTableEntry yEntry;

    void RobotInit() {
        auto inst = nt::NetworkTableInstance::GetDefault();
        auto table = inst.GetTable("datatable");
        xEntry = table->GetEntry("X");
        yEntry = table->GetEntry("Y");
    }

    double x = 0;
    double y = 0;

    void TeleopPeriodic() {
        xEntry.SetDouble(x);
        yEntry.SetDouble(y);
        x += 0.05;
        y += 0.05;
    }
}

START_ROBOT_CLASS(EasyNetworkExample)

```

The values for X and Y can be easily viewed using the OutlineViewer program that shows the NetworkTables hierarchy and all the values associated with each key.

Note: Actually NetworkTables has a flat namespace for the keys. Having tables and subtables is an abstraction to make it easier to organize your data. So for a table called "SmartDashboard" and a key named "xValue", it is really a single key called "/SmartDashboard/xValue". The hierarchy is not actually represented in the distributed data, only keys with prefixes that are the contained table.

32.2 Listening for value changes

A common use case for *NetworkTables* is where a program on the Driver Station generates values that need to be sent to the robot. For example, imagine that some image processing code running on the Driver Station computes the heading and distance to a goal and sends those values to the robot. In this case it might be desirable for the robot program to be notified when new values arrive from the Driver Station. NetworkTables provides facilities to do that.

32.2.1 Using a NetworkTable EntryListener

One of the more common cases for NetworkTables is waiting for a value to change on the robot. This is called an EntryListener because it notifies the robot program when a NetworkTables entry (single value) changes. The following code shows how to do this.

Java

```
package networktablesdesktopclient;

import edu.wpi.first.networktables.EntryListenerFlags;
import edu.wpi.first.networktables.NetworkTable;
import edu.wpi.first.networktables.NetworkTableEntry;
import edu.wpi.first.networktables.NetworkTableInstance;

public class TableEntryListenerExample {

    public static void main(String[] args) {
        new TableEntryListenerExample().run();
    }

    public void run() {
        //get the default instance of NetworkTables
        NetworkTableInstance inst = NetworkTableInstance.getDefault();

        //get a reference to the subtable called "datatable"
        NetworkTable table = inst.getTable("datatable");

        //get a reference to key in "datatable" called "Y"
        NetworkTableEntry yEntry = table.getEntry("Y");
        inst.startClientTeam(190);

        //add an entry listener for changed values of "X", the lambda ("->" operator)
        //defines the code that should run when "X" changes
        table.addEntryListener("X", (table, key, entry, value, flags) -> {
            System.out.println("X changed value: " + value.getValue());
        }, EntryListenerFlags.kNew | EntryListenerFlags.kUpdate);

        //add an entry listener for changed values of "Y", the lambda ("->" operator)
        //defines the code that should run when "Y" changes
        yEntry.addListener(event -> {
            System.out.println("Y changed value: " + value.getValue());
        }, EntryListenerFlags.kNew | EntryListenerFlags.kUpdate);

        try {
            Thread.sleep(10000);
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

    } catch (InterruptedException ex) {
        System.out.println("Interrupted");
        Thread.currentThread().interrupt();
        return;
    }
}
}

```

32.3 Creating a client-side program

If all you need to do is have your robot program communicate with GRIP or a dashboard running on the Driver Station laptop, then the previous examples of writing robot programs are sufficient. But if you would like to write some custom client code that would run on the drivers station or on a coprocessor then you need to know how to build *NetworkTables* programs for those (non-roboRIO) platforms.

A basic client program looks like the following example.

Java

```

package networktablesdesktopclient;

import edu.wpi.first.networktables.NetworkTable;
import edu.wpi.first.networktables.NetworkTableEntry;
import edu.wpi.first.networktables.NetworkTableInstance;

public class NetworkTablesDesktopClient {
    public static void main(String[] args) {
        new NetworkTablesDesktopClient().run();
    }

    public void run() {
        NetworkTableInstance inst = NetworkTableInstance.getDefault();
        NetworkTable table = inst.getTable("datatable");
        NetworkTableEntry xEntry = table.getEntry("x");
        NetworkTableEntry yEntry = table.getEntry("y");
        inst.startClientTeam(TEAM); // where TEAM=190, 294, etc, or use inst.startClient(
↳ "hostname") or similar
        inst.startDSClient(); // recommended if running on DS computer; this gets the
↳ robot IP from the DS
        while (true) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                System.out.println("interrupted");
                return;
            }
            double x = xEntry.getDouble(0.0);
            double y = yEntry.getDouble(0.0);
            System.out.println("X: " + x + " Y: " + y);
        }
    }
}

```

In this example an instance of `NetworkTables` is created and a `NetworkTableEntry` is created to reference the values of “x” and “y” from a table called “datatable”.

Then this instance is started as a `NetworkTables` client with the team number (the roboRIO is always the server). Additionally, if the program is running on the Driver Station computer, by using the `startDSClient()` method, `NetworkTables` will get the robot IP address from the Driver Station.

Then this sample program simply loops once a second and gets the values for x and y and prints them on the console. In a more realistic program, the client might be processing or generating values for the robot to consume.

32.3.1 Building the program

When building and running the program you will need some additional libraries to include with your client-side program. These are:

<https://frcmaven.wpi.edu/artifactory/release/edu/wpi/first/ntcore/ntcore-java/> (ntcore Java files)

<https://frcmaven.wpi.edu/artifactory/release/edu/wpi/first/ntcore/ntcore-jni/> (ntcore native libs for all desktop platforms)

<https://frcmaven.wpi.edu/artifactory/release/edu/wpi/first/wpiutil/wpiutil-java/> (wpiutil Java files)

Note: The desktop platform jar is for Windows, macOS, and Linux.

Building using Gradle

The dependencies above can be added to the dependencies block in a `build.gradle` file. The `ntcore-java` and `wpiutil-java` libraries are required at compile-time and the JNI dependencies are required at runtime. The JNI dependencies for all supported platforms should be added to the `build.gradle` if cross-platform support for the application is desired.

First, the FRC® Maven repository should be added to the repositories block. Note that this is not required if you are using the GradleRIO plugin with your application.

```
repositories {  
    maven { url "https://frcmaven.wpi.edu/artifactory/release/" }  
}
```

Then, the dependencies can be added to the dependencies block. Here, `VERSION` should be replaced with the latest version number of the following dependencies. This usually corresponds to the version number of the latest WPILib release.

```
dependencies {  
    // Add ntcore-java  
    compile "edu.wpi.first.ntcore:ntcore-java:VERSION"  
  
    // Add wpiutil-java  
    compile "edu.wpi.first.wpiutil:wpiutil-java:VERSION"  
  
    // Add ntcore-jni for runtime. We are adding all supported platforms
```

(continues on next page)

(continued from previous page)

```
// so that our application will work on all supported platforms.
runtime "edu.wpi.first.ntcore:ntcore-jni:VERSION:windowsx86"
runtime "edu.wpi.first.ntcore:ntcore-jni:VERSION:windowsx86-64"
runtime "edu.wpi.first.ntcore:ntcore-jni:VERSION:linuxx86-64"
runtime "edu.wpi.first.ntcore:ntcore-jni:VERSION:linuxraspbian"
runtime "edu.wpi.first.ntcore:ntcore-jni:VERSION:osxx86-64"
}
```

32.4 Creating multiple instances of NetworkTables

This feature is mainly useful for coprocessors and unit testing. It allows a single program to be a member of two completely independent *NetworkTables* “networks” that contain completely different (and unrelated) sets of tables. For most general usage, you should use tables within the single instance, as all current dashboard programs can only connect to a single NetworkTables server at a time.

Normally the “default” instance is set up on the robot as a server, and used for communication with the dashboard program running on your driver station computer. This is what the SmartDashboard and LiveWindow classes use.

If you had a coprocessor and wanted to have a set of tables that’s shared only between the coprocessor and the robot, you could set up a separate instance in the robot code that acts as a client (or a server) and connect the coprocessor to it, and those tables will NOT be sent to the dashboard.

Similarly, if you wanted to do unit testing of your robot program’s NetworkTables communications, you could set up your unit tests such that they create a separate client instance (still within the same program) and have it connect to the server instance that the main robot code is running.

Another example might be having two completely separate dashboard programs. You could set up two NetworkTables server instances in your robot program (on different TCP ports of course), set up different tables on each one, and have each dashboard connect to a different server instance. Each dashboard would only see the tables on its instance.

32.5 Listening for subtable creation

Java

```
package networktablesdesktopclient;

import edu.wpi.first.networktables.NetworkTable;
import edu.wpi.first.networktables.NetworkTableInstance;

public class SubTableListenerExample {

    public static void main(String[] args) {
        new SubTableListenerExample().run();
    }

    public void run() {
```

(continues on next page)

(continued from previous page)

```
NetworkTableInstance inst = NetworkTableInstance.getDefault();
NetworkTable table = inst.getTable("datatable");
inst.startClientTeam(190);

table.addSubTableListener((parent, name, table) -> {
    System.out.println("Parent: " + parent + " Name: " + name);
}, false);

try {
    Thread.sleep(100000);
} catch (InterruptedException ex) {
    System.out.println("Interrupted!");
}
}
```

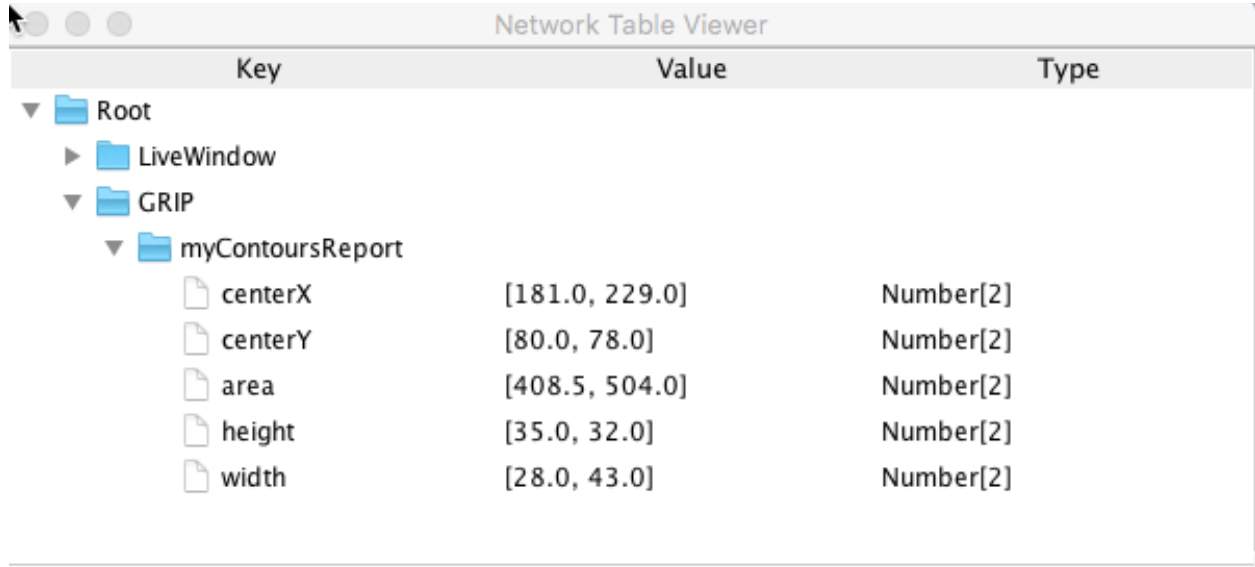
In this example a listener is defined that listens for creation of subtables one level down from the listener. That is subtables with the key `"/datatable/<any-table-name>"`. This will fire the callback for each existing subtable, then continue to listen for new subtables created immediately below `"datatable"` in this case.

32.6 Reading array values published by NetworkTables

This article describes how to read values published by *NetworkTables* using a program running on the robot. This is useful when using computer vision where the images are processed on your driver station laptop and the results stored into NetworkTables possibly using a separate vision processor like a raspberry pi, or a tool on the robot like GRIP, or a python program to do the image processing.

Very often the values are for one or more areas of interest such as goals or game pieces and multiple instances are returned. In the example below, several x, y, width, height, and areas are returned by the image processor and the robot program can sort out which of the returned values are interesting through further processing.

32.6.1 Verify the NetworkTables keys being published



Key	Value	Type
Root		
LiveWindow		
GRIP		
myContoursReport		
centerX	[181.0, 229.0]	Number[2]
centerY	[80.0, 78.0]	Number[2]
area	[408.5, 504.0]	Number[2]
height	[35.0, 32.0]	Number[2]
width	[28.0, 43.0]	Number[2]

You can verify the names of the NetworkTables keys used for publishing the values by using the Outline Viewer application. It is a Java program in your user directory in the wpilib/<YEAR>/tools folder. The application is started by selecting the “WPILib” menu in Visual Studio Code then Start Tool then “OutlineViewer”. In this example, with the image processing program running (GRIP) you can see the values being put into NetworkTables.

In this case the values are stored in a table called GRIP and a sub-table called myContoursReport. You can see that the values are in brackets and there are 2 values in this case for each key. The NetworkTables key names are centerX, centerY, area, height and width.

Both of the following examples are extremely simplified programs that just illustrate the use of NetworkTables. All the code is in the robotInit() method so it’s only run when the program starts up. In your programs, you would more likely get the values in code that is evaluating which direction to aim the robot in a command or a control loop during the autonomous or teleop periods.

32.6.2 Writing a program to access the keys

Java

C++

```
NetworkTable table;
double[] areas;
double[] defaultValue = new double[0];

@Override
public void robotInit() {
    table = NetworkTableInstance.getDefault().getTable("GRIP/mycontoursReport");
}

@Override
public void teleopPeriodic() {
    double[] areas = table.getEntry("area").getDoubleArray(defaultValue);
}
```

(continues on next page)

(continued from previous page)

```
System.out.print("areas: " );

for (double area : areas) {
    System.out.print(area + " ");
}

System.out.println();
}
```

```
std::shared_ptr<NetworkTable> table;

void Robot::RobotInit() override {
    table = NetworkTable::GetTable("GRIP/myContoursReport");
}

void Robot::TeleopPeriodic() override {
    std::cout << "Areas: ";

    std::vector<double> arr = table->GetEntry("area").GetDoubleArray(std::vector<double>
↪());

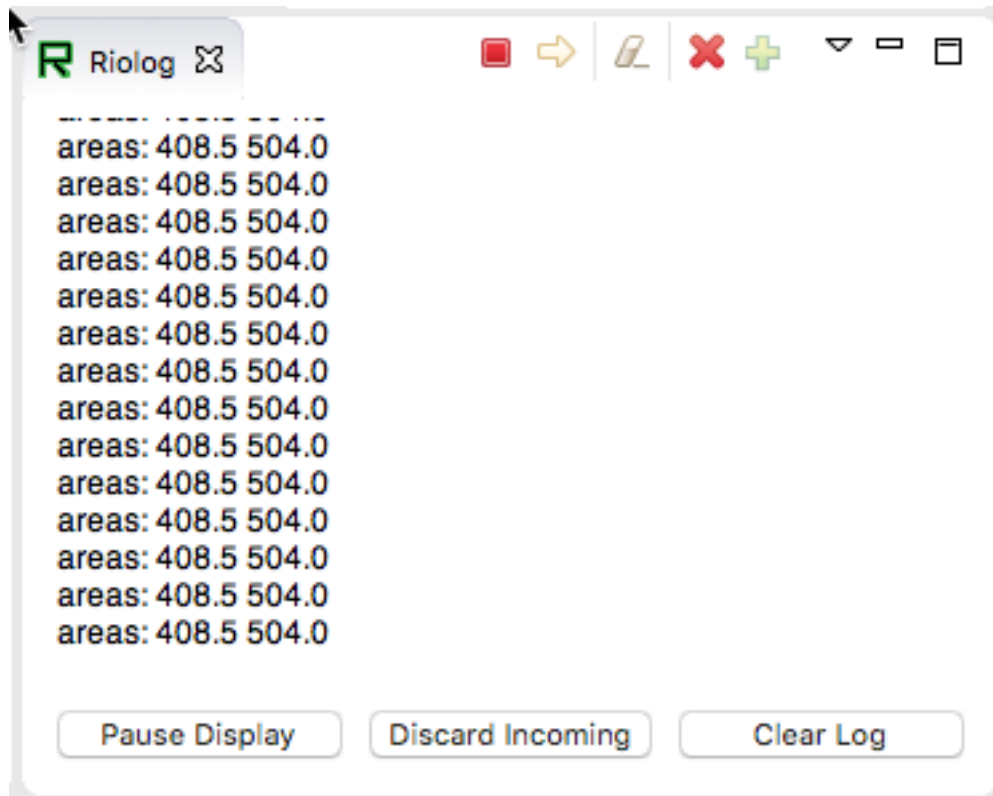
    for (double val : arr) std::cout << val << " ";

    std::cout << std::endl;
}
```

The steps to getting the values and, in this program, printing them are:

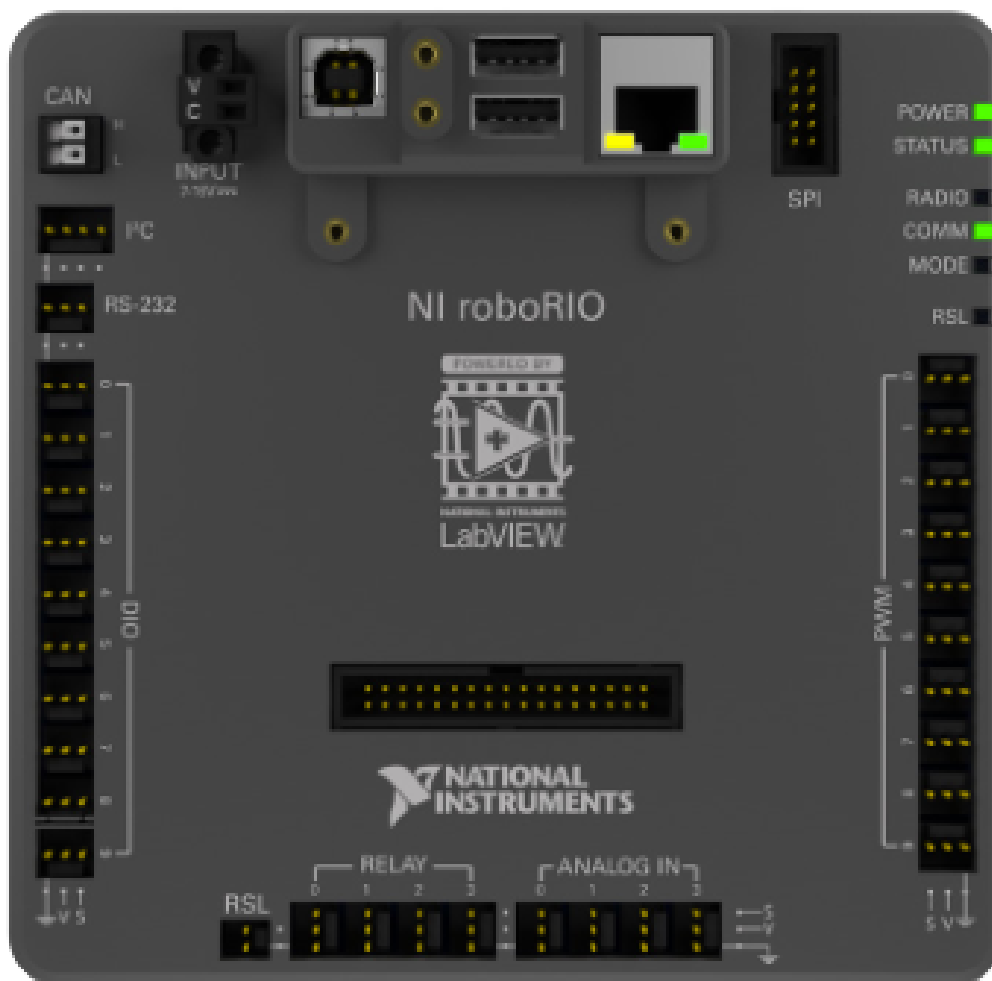
1. Declare the table variable that will hold the instance of the subtable that have the values.
2. Initialize the subtable instance so that it can be used later for retrieving the values.
3. Read the array of values from NetworkTables. In the case of a communicating programs, it's possible that the program producing the output being read here might not yet be available when the robot program starts up. To avoid issues of the data not being ready, a default array of values is supplied. This default value will be returned if the NetworkTables key hasn't yet been published. This code will loop over the value of areas every 20ms.

32.6.3 Program output



In this case the program is only looking at the array of areas, but in a real example all the values would more likely be used. Using the Riolog in VS Code or the Driver Station log you can see the values as they are retrieved. This program is using a sample static image so they areas don't change, but you can imagine with a camera on your robot, the values would be changing constantly.

33.1 roboRIO Introduction



The roboRIO is designed specifically with FIRST in mind. The roboRIO has a basic architecture of a Real-Time processors + FPGA (field programmable gate array) but is more powerful,

lighter, and smaller than some similar systems used in industry.

The roboRIO is a reconfigurable robotics controller that includes built-in ports for inter-integrated circuits (I2C), serial peripheral interfaces (SPI), RS232, USB, Ethernet, pulse width modulation (PWM), and relays to quickly connect the common sensors and actuators used in robotics. The controller features LEDs, buttons, an onboard accelerometer, and a custom electronics port. It has an onboard dual-core ARM real-time Cortex-A9 processor and customizable Xilinx FPGA.

Detailed information on the roboRIO can be found in the [roboRIO User Manual](#) and in the [roboRIO technical specifications](#).

Before deploying programs to your roboRIO, you must first *image the roboRIO* with the FRC roboRIO Imaging Tool shown in the next article.

33.2 roboRIO Web Dashboard

The roboRIO web dashboard is a webpage built into the roboRIO that can be used for checking status and updating settings of the roboRIO.

Unlike the 2015-2018 roboRIO web dashboard, the 2019 web dashboard does not use SilverLight. Users may encounter issues using IE (compatibility) or Edge (mDNS site access). Alternate browsers such as Google Chrome or Mozilla Firefox are recommended for the best experience.

Note: The roboRIO web dashboard was been re-written for 2019. All CAN configuration functionality has been removed. Configuration of CAN devices should be done with software provided by the device vendor. For CTRE devices previously serviced using the webdashboard, the appropriate software is [CTRE Phoenix Tuner](#).

33.2.1 Opening the WebDash



To open the web dashboard, open a web browser and enter the address of the roboRIO into the address bar (172.22.11.2 for USB, or “roboRIO-####-FRC.local where #### is your team number, with no leading zeroes, for either interface). See this document for more details about mDNS and roboRIO networking: [IP Configurations](#)

33.2.2 System Configuration Tab

The home screen of the web dashboard is the System Configuration tab which has 5 main sections:

1. Navigation Bar - This section allows you to navigate to different sections of the web dashboard. The different pages accessible through this navigation bar are discussed below.
2. System Settings - This section contains information about the System Settings. The Hostname field should not be modified manually, instead use the roboRIO Imaging tool to set the Hostname based on your team number. This section contains information such as the device IP, firmware version and image version.

172.22.11.2: System Configuration



Save

Settings

Hostname	roboRIO-1-FRC
IP Address	0.0.0.0 (Ethernet) 172.22.11.2 (Ethernet)
DNS Name	
Vendor	National Instruments
Model	roboRIO
Serial Number	030498A9
Firmware Version	6.0.0f1
Operating System	NI Linux Real-Time ARMv7-A 4.9.47-rt37-ni-6.0.0f1
Status	Running
System Start Time	Mon Nov 19 2018 14:16:34 GMT-0500 (Eastern Standard Time)
Image Title	roboRIO Image
Image Version	FRC_roboRIO_2019_v12
Comments	<div></div>
Locale	English
VISA Resource Name	system

Update Firmware

Startup Settings

☐ Force Safe Mode

☒ Enable Console Out

☐ Disable RT Startup App



☐ Disable FPGA Startup App

☒ Enable Secure Shell Server (sshd)

☒ LabVIEW Project Access

172.22.11.2: System Configuration

1



Save

Settings 2

Hostname	roboRIO-1-FRC
IP Address	0.0.0.0 (Ethernet) 172.22.11.2 (Ethernet)
DNS Name	
Vendor	National Instruments
Model	roboRIO
Serial Number	030498A9
Firmware Version	6.0.0f1
Operating System	NI Linux Real-Time ARMv7-A 4.9.47-rt37-ni-6.0.0f1
Status	Running
System Start Time	Mon Nov 19 2018 14:16:34 GMT-0500 (Eastern Standard Time)
Image Title	roboRIO Image
Image Version	FRC_roboRIO_2019_v12
Comments	<div></div>
Locale	English
VISA Resource Name	system

Update Firmware

Startup Settings 3

- ☐ Force Safe Mode
- ☒ Enable Console Out
- ☐ Disable RT Startup App
- ☐ Disable FPGA Startup App
- ☒ Enable Secure Shell Server (ssh)
- ☒ LabVIEW Project Access

3. Startup Settings - This section contains Startup settings for the roboRIO. These are described in the sub-step below
4. System Resources (not pictured) - This section provides a snapshot of system resources such as memory and CPU load.

Startup Settings

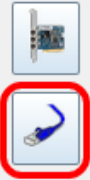
Startup Settings

- ☐ Force Safe Mode
- ☒ Enable Console Out
- ☐ Disable RT Startup App
- ☐ Disable FPGA Startup App
- ☒ Enable Secure Shell Server (sshd)
- ☒ LabVIEW Project Access

- Force Safe Mode - Forces the controller into Safe Mode. This can be used with troubleshooting imaging issues, but it is recommended to use the Reset button on the roboRIO to put the device into Safe Mode instead (with power already applied, hold the reset button for 5 seconds). **Default is unchecked.**
- Enable Console Out - This enables the on-board RS232 port to be used as a Console output. It is recommended to leave this enabled unless you are using this port to talk to a serial device (note that this port uses RS232 levels and should not be connected to many microcontrollers which use TTL levels). **Default is checked.**
- Disable RT Startup App - Checking this box disables code from running at startup. This may be used for troubleshooting if you find the roboRIO is unresponsive to new program download. Default is unchecked
- Disable FPGA Startup App - **This box should not be checked.**
- Enable Secure Shell Server (sshd) - **It is recommended to leave this box checked.** This setting enables SSH which is a way to remotely access a console on the roboRIO. Unchecking this box will prevent C++ and Java teams from loading code onto the roboRIO.
- LabVIEW Project Access -** It is recommended to leave this box checked.** This setting allows LabVIEW projects to access the roboRIO.

33.2.3 Network Configuration

172.22.11.2: Network Configuration



Save

Ethernet Adapter eth0

Settings

MAC Address	00:80:2F:30:49:8A
Configure IPv4 Address	DHCP or Link Local
IPv4 Address	0.0.0.0
Subnet Mask	0.0.0.0
Gateway	0.0.0.0
DNS Server	0.0.0.0
Current Link Speed	10Mbit/Half Duplex
Preferred Link Speed	Autonegotiate

Ethernet Adapter usb0

Settings

MAC Address	00:80:2F:40:49:8A
Configure IPv4 Address	DHCP Only
IPv4 Address	172.22.11.2
Subnet Mask	255.255.255.248
Gateway	0.0.0.0
DNS Server	0.0.0.0
Current Link Speed	Autonegotiate
Preferred Link Speed	Autonegotiate

This page shows the configuration of the roboRIO's network adapters. **It is not recommended to change any settings on this page.** For more information on roboRIO networking see this article: [IP Configurations](#)

33.3 roboRIO FTP

Note: The roboRIO has both SFTP and anonymous FTP enabled. This article describes how to use each to access the roboRIO file system.

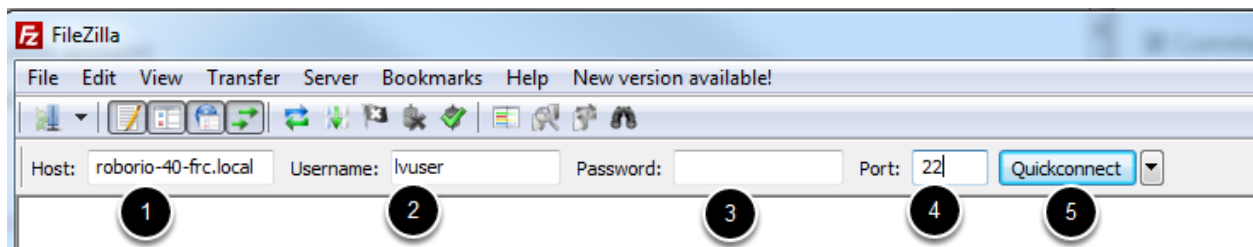
33.3.1 SFTP

SFTP is the recommended way to access the roboRIO file system. Because you will be using the same account that your program will run under, files copied over should always have permissions compatible with your code.

Software

There are a number of freely available programs for SFTP. This article will discuss using FileZilla. You can either download and install [FileZilla](#) before proceeding or extrapolate the directions below to your SFTP client of choice.

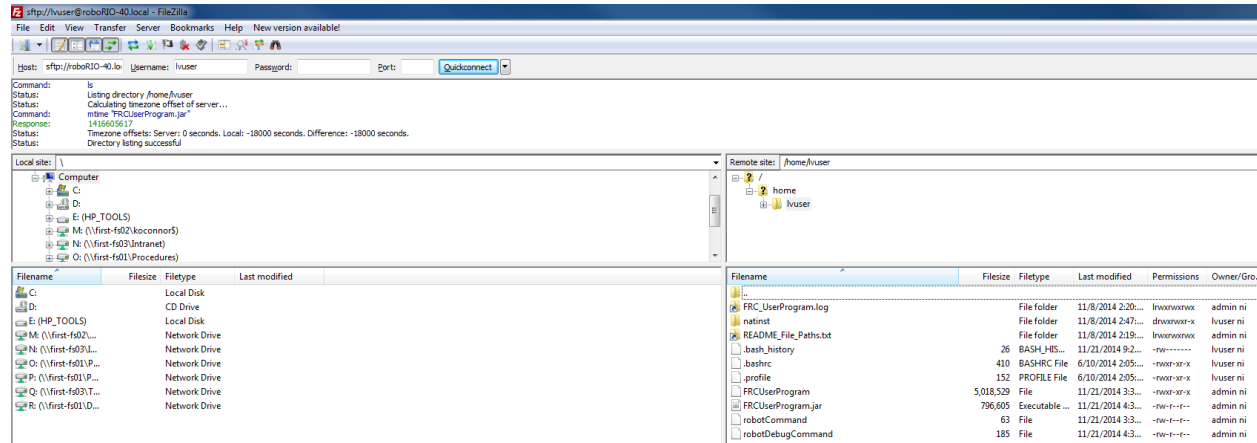
Connecting to the roboRIO



To connect to your roboRIO:

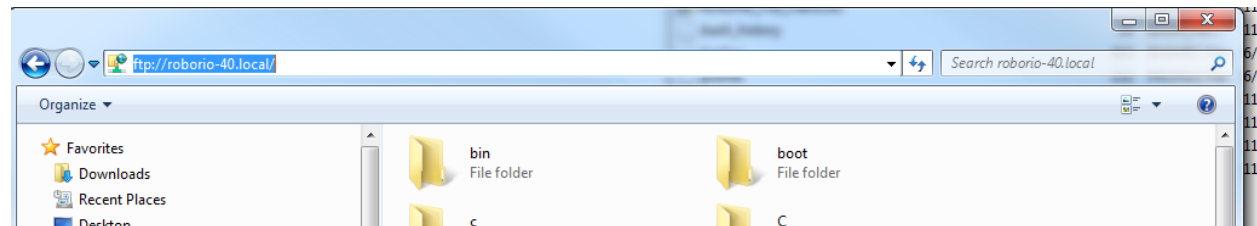
1. Enter the mDNS name (roboRIO-TEAM-frc.local) in the "Host" box
2. Enter "lvuser" in the Username box (this is the account your program runs under)
3. Leave the Password box blank
4. Enter "22" in the port box (the SFTP default port)
5. Click Quickconnect

Browsing the roboRIO filesystem



After connecting to the roboRIO, Filezilla will open to the `\home\lvuser` directory. The right pane is the remote system (the roboRIO), the left pane is the local system (your computer). The top section of each pane shows you the hierarchy to the current directory you are browsing, the bottom pane shows contents of the directory. To transfer files, simply click and drag from one side to the other. To create directories on the roboRIO, right click and select “Create Directory”.

33.3.2 FTP



The roboRIO also has anonymous FTP enabled. It is recommended to use SFTP as described above, but depending on what you need FTP may work in a pinch with no additional software required. To FTP to the roboRIO, open a Windows Explorer window (on Windows 7, you can click Start->My Computer). In the address bar, type `ftp://roboRIO-TEAM-frc.local` and press enter. You can now browse the roboRIO file system just like you would browse files on your computer.

33.4 roboRIO User Accounts and SSH

Note: This document contains advanced topics not required for typical FRC® programming

The roboRIO image contains a number of accounts, this article will highlight the two used for FRC and provide some detail about their purpose. It will also describe how to connect to the roboRIO over SSH.

33.4.1 roboRIO User Accounts

The roboRIO image contains a number of user accounts, but there are two of primary interest for FRC.

Admin

The “admin” account has root access to the system and can be used to manipulate OS files or settings. Teams should take caution when using this account as it allows for the modification of settings and files that may corrupt the operating system of the roboRIO. The credentials for this account are:

Username: admin

Password:

Note: The password is intentionally blank.

Lvuser

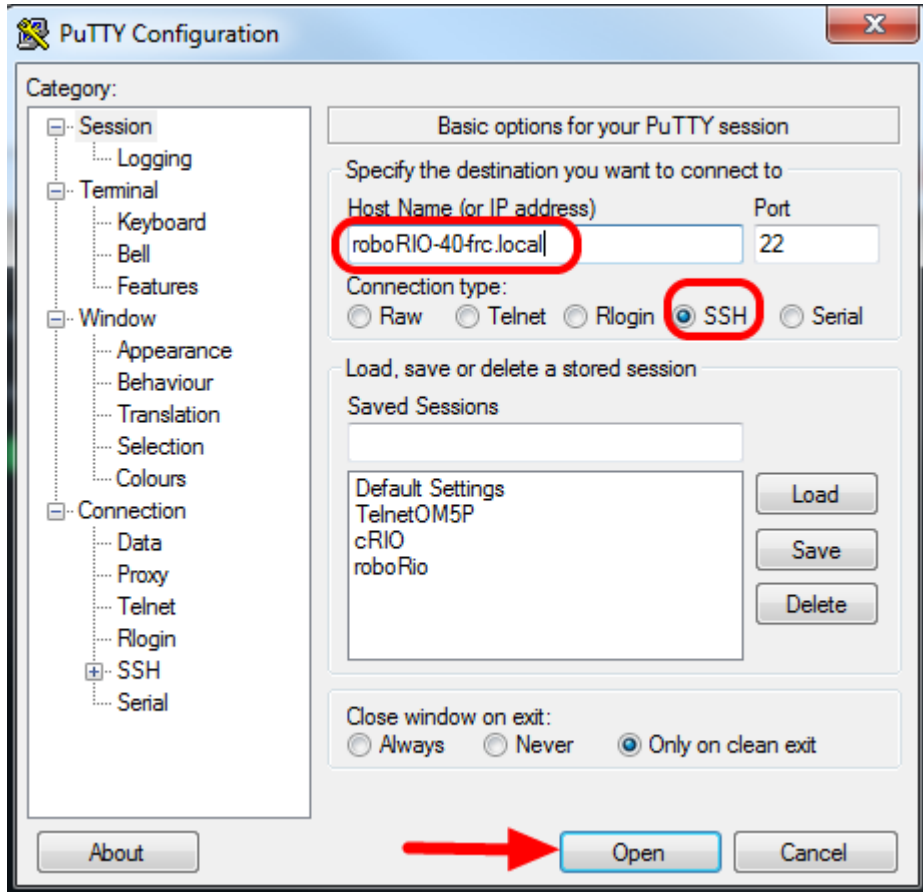
The “lvuser” account is the account used to run user code for all three languages. The credentials for this account should not be changed. Teams may wish to use this account (via ssh or sftp) when working with the roboRIO to ensure that any files or settings changes are being made on the same account as their code will run under.

Danger: Changing the default ssh passwords for either “lvuser” or “admin” will prevent C++ and Java teams from uploading code.

33.4.2 SSH

SSH (Secure SHell) is a protocol used for secure data communication. When broadly referred to regarding a Linux system (such as the one running on the roboRIO) it generally refers to accessing the command line console using the SSH protocol. This can be used to execute commands on the remote system. A free client which can be used for SSH is PuTTY: <https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html>

Open Putty



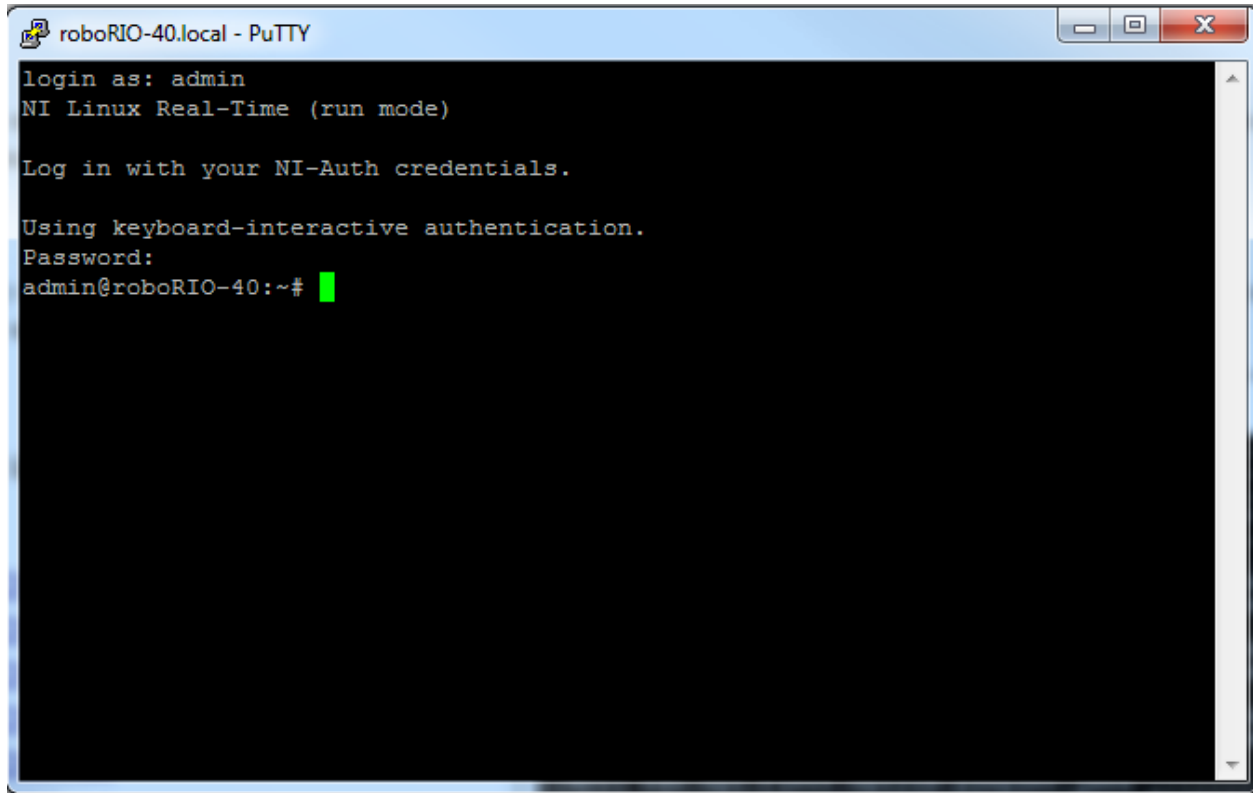
Open Putty (clicking OK at any security prompt). Then set the following settings:

1. Host Name: roboRIO-TEAM-frc.local (where TEAM is your team number, example shows team 40)
2. Connection Type: SSH

Other settings can be left at defaults. Click Open to open the connection. If you see a prompt about SSH keys, click OK.

If you are connected over USB you can use 172.22.11.2 as the hostname. If your roboRIO is set to a static IP you can use that IP as the hostname if connected over Ethernet/wireless.

Log In



```
roboRIO-40.local - PuTTY
login as: admin
NI Linux Real-Time (run mode)

Log in with your NI-Auth credentials.

Using keyboard-interactive authentication.
Password:
admin@roboRIO-40:~#
```

When you see the prompt, enter the desired username (see above for description) then press enter. At the password prompt press enter (password for both accounts is blank).

33.5 roboRIO Brownout and Understanding Current Draw

In order to help maintain battery voltage to preserve itself and other control system components such as the radio during high current draw events, the roboRIO contains a staged brownout protection scheme. This article describes this scheme, provides information about proactively planning for system current draw, and describes how to use the new functionality of the PDP as well as the DS Log File Viewer to understand brownout events if they do happen on your robot.

33.5.1 roboRIO Brownout Protection

The roboRIO uses a staged brownout protection scheme to attempt to preserve the input voltage to itself and other control system components in order to prevent device resets in the event of large current draws pulling the battery voltage dangerously low.

Stage 1 - 6v output drop

Voltage Trigger - 6.8V

When the voltage drops below 6.8V, the 6V output on the PWM pins will start to drop.

Stage 2 - Output Disable

Voltage Trigger - 6.3V

When the voltage drops below 6.3V, the controller will enter the brownout protection state. The following indicators will show that this condition has occurred:

- Power LED on the roboRIO will turn Amber
- Background of the voltage display on the Driver Station will turn red
- Mode display on the Driver Station will change to Voltage Brownout
- The CAN/Power tab of the DS will increment the 12V fault counter by 1.
- The DS will record a brownout event in the DS log.

The controller will take the following steps to attempt to preserve the battery voltage:

- PWM outputs will be disabled. For PWM outputs which have set their neutral value (all motor controllers in WPILib) a single neutral pulse will be sent before the output is disabled.
- 6V, 5V, 3.3V User Rails disabled (This includes the 6V outputs on the PWM pins, the 5V pins in the DIO connector bank, the 5V pins in the Analog bank, the 3.3V pins in the SPI and I2C bank and the 5V and 3.3V pins in the MXP bank)
- GPIO configured as outputs go to High-Z
- Relay Outputs are disabled (driven low)
- CAN-based motor controllers are sent an explicit disable command

The controller will remain in this state until the voltage rises to greater than 7.5V or drops below the trigger for the next stage of the brownout

Stage 3 - Device Blackout

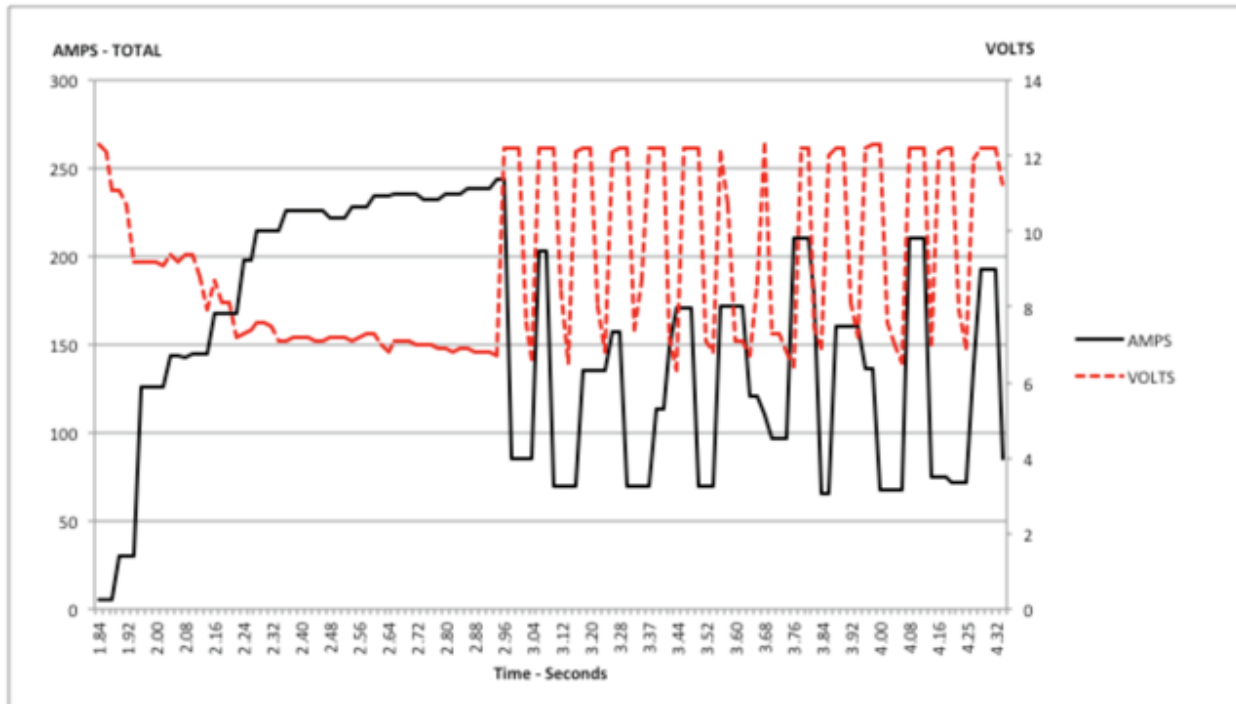
Voltage Trigger - 4.5V

Below 4.5V the device may blackout. The exact voltage may be lower than this and depends on the load on the device.

The controller will remain in this state until the voltage rises above 4.65V when the device will begin the normal boot sequence.

33.5.2 Avoiding Brownout - Proactive Current Draw Planning

PLOT 1 – AMPS and VOLTS v. Time – 2.5 Second Window



The key to avoiding a brownout condition is to proactively plan for the current draw of your robot. The best way to do this is to create some form of power budget. This can be a complex document that attempts to quantify both estimated current draw and time in an effort to most completely understand power usage and therefore battery state at the end of a match, or it can be a simple inventory of current usage. To do this:

1. Establish the max “sustained” current draw (with sustained being loosely defined here as not momentary). This is probably the most difficult part of creating the power budget. The exact current draw a battery can sustain while maintaining a voltage of 7+ volts is dependent on a variety of factors such as battery health (see [this](#) article for measuring battery health) and state of charge. As shown in the [NP18-12 data sheet](#), the terminal voltage chart gets very steep as state of charge decreases, especially as current draw increases. This datasheet shows that at 3CA continuous load (54A) a brand new battery can be continuously run for over 6 minutes while maintaining a terminal voltage of over 7V. As shown in the image above (used with permission from [Team 234s Drive System Testing document](#)), even with a fresh battery, drawing 240A for more than a second or two is likely to cause an issue. This gives us some bounds on setting our sustained current draw. For the purposes of this exercise, we’ll set our limit at 180A.
2. List out the different functions of your robot such as drivetrain, manipulator, main game mechanism, etc.
3. Start assigning your available current to these functions. You will likely find that you run out pretty quickly. Many teams gear their drivetrain to have enough torque to slip their wheels at 40-50A of current draw per motor. If we have 4 motors on the drivetrain, that eats up most, or even exceeds, our power budget! This means that we may need to put together a few scenarios and understand what functions can (and need to be) be used at the same time. In many cases, this will mean that you really need to limit the current

draw of the other functions if/while your robot is maxing out the drivetrain (such as trying to push something). Benchmarking the “driving” current requirements of a drivetrain for some of these alternative scenarios is a little more complex, as it depends on many factors such as number of motors, robot weight, gearing, and efficiency. Current numbers for other functions can be done by calculating the power required to complete the function and estimating efficiency (if the mechanism has not been designed) or by determining the torque load on the motor and using the torque-current curve to determine the current draw of the motors.

4. If you have determined mutually exclusive functions in your analysis, consider enforcing the exclusion in software. You may also use the current monitoring of the PDP (covered in more detail below) in your robot program to provide output limits or exclusions dynamically (such as don’t run a mechanism motor when the drivetrain current is over X or only let the motor run up to half output when the drivetrain current is over Y).

33.5.3 Measuring Current Draw using the PDP

The FRC® Driver Station works in conjunction with the roboRIO and PDP to extract logged data from the PDP and log it on your DS PC. A viewer for this data is still under development.

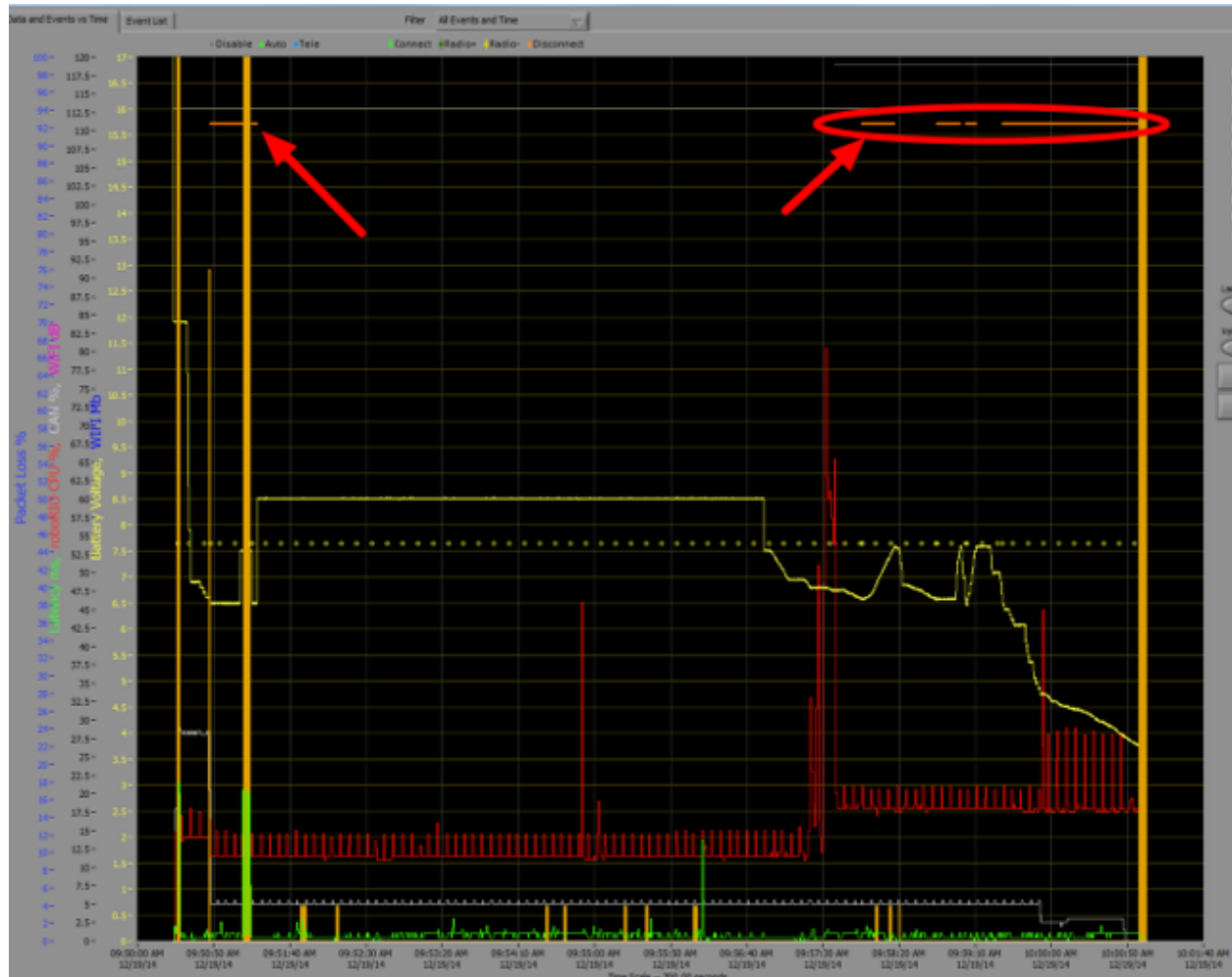
In the meantime, teams can use their robot code and manual logging, a LabVIEW front panel or the SmartDashboard to visualize current draw on their robot as mechanisms are developed. In LabVIEW, you can read the current on a PDP channel using the PDP Channel Current VI found on the Power pallet. For C++ and Java teams, use the `PowerDistributionPanel` class as described in the Power Distribution Panel article. Plotting this information over time (easiest with a LV Front Panel or with the SmartDashboard by using a Graph indicator can provide information to compare against and update your power budget or can locate mechanisms which do not seem to be performing as expected (due to incorrect load calculation, incorrect efficiency assumptions, or mechanism issues such as binding).

33.5.4 Identifying Brownouts

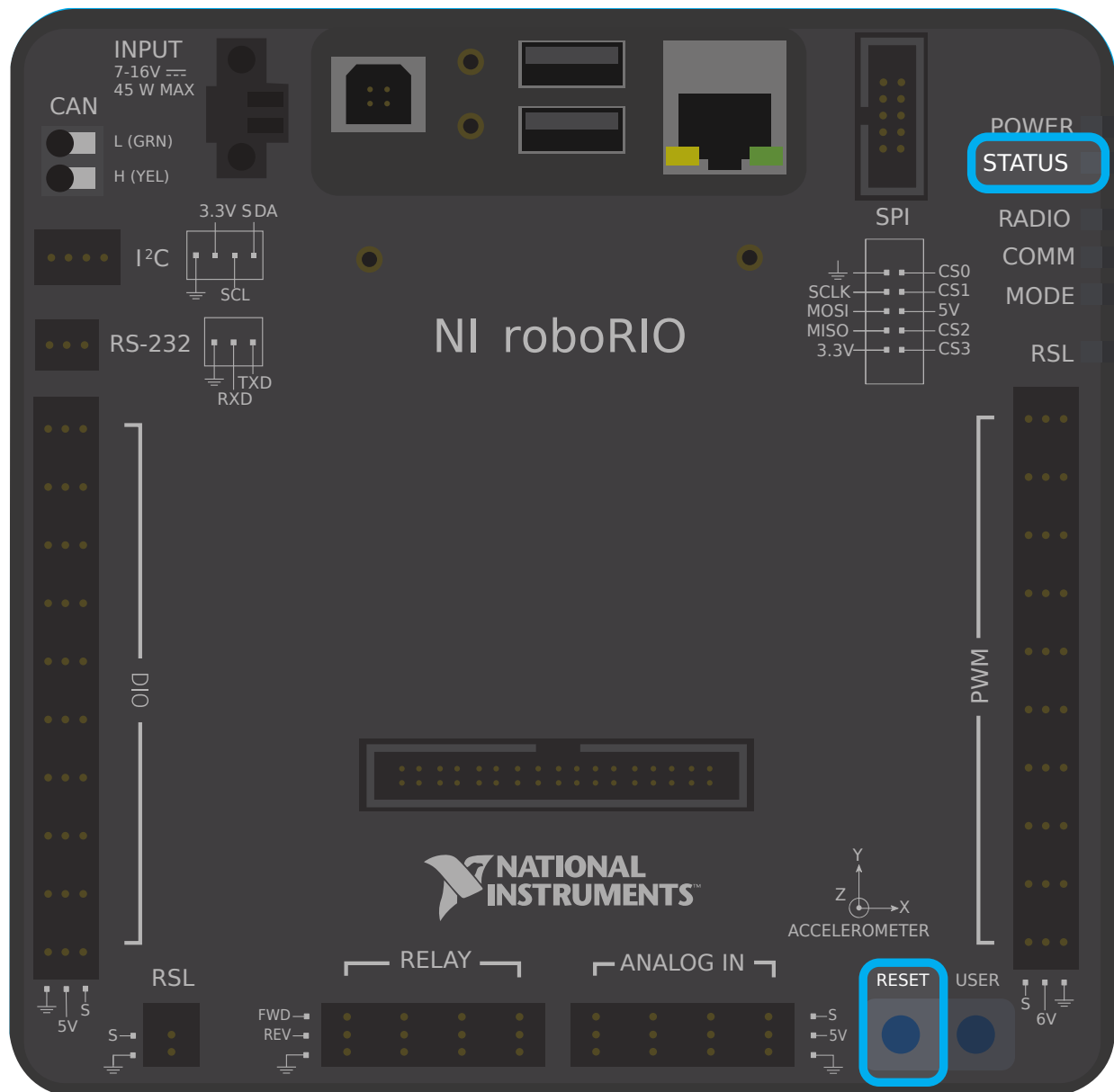
The easiest way to identify a brownout is by clicking on the CAN\Power tab of the DS and checking the 12V fault count. Alternately, you can review the Driver Station Log after the fact using the Driver Station Log Viewer. The log will identify brownouts with a bright orange line, such as in the image above (note that these brownouts were induced with a benchtop supply and may not reflect the duration and behavior of brownouts on a typical FRC robot).

33.6 Recovering a roboRIO using Safe Mode

Occasionally a roboRIO may become corrupted to the point that it cannot be recovered using the normal boot and imaging process. Booting the roboRIO into Safe Mode may allow the device to be successfully re-imaged.



33.6.1 Booting into Safe Mode



To boot the roboRIO into Safe Mode:

1. Apply power to the roboRIO
2. Press and hold the Reset button until the Status LED lights up (~5 seconds) then release the Reset button
3. The roboRIO will boot in Safe Mode (indicated by the Status LED flashing in groups of 3)

33.6.2 Recovering the roboRIO

The roboRIO can now be imaged by using the roboRIO Imaging Tool as described in *Imaging your roboRIO*.

33.6.3 About Safe Mode

In Safe Mode, the roboRIO boots a separate copy of the operating system into a RAM Disk. This allows you to recover the roboRIO even if the normal copy of the OS is corrupted. While in Safe Mode, any changes made to the OS (such as changes made by accessing the device via SSH or Serial) will not persist to the normal copy of the OS stored on disk.

GradleRIO is the mechanism that powers the deployment of robot code to the roboRIO. GradleRIO is built on the popular Gradle dependency and build management system. This section highlights **advanced** configurations that teams can use to enhance their workflow.

34.1 Using External Libraries with Robot Code

Warning: Using external libraries may have unintended behavior with your robot code! It is not recommended unless you are aware of what you are doing!

Often a team might want to add external Java or C++ libraries for usage with their robot code. This article highlights adding Java libraries to your Gradle dependencies, or the options that C++ teams have.

34.1.1 Java

Note: Any external dependencies that rely on native libraries (JNI) are likely not going to work.

Java is quite simple to add external dependencies. You simply add the required repositories and dependencies.

Robot projects by default do not have a `repositories {}` block in the `build.gradle` file. You will have to add this yourself. Above the `dependencies {}` block, please add the following:

```
repositories {  
    mavenCentral()  
    ...  
}
```

`mavenCentral()` can be replaced with whatever repository the library you want to import is using. Now you have to add the dependency on the library itself. This is done by adding the

necessary line to your dependencies `{}` block. The below example showcases adding Apache Commons to your Gradle project.

```
dependencies {  
    implementation 'org.apache.commons:commons-lang3:3.6'  
    ...  
}
```

Now you run a build and ensure the dependencies are downloaded. Intellisense may not work properly until a build is ran!

34.1.2 C++

Adding C++ dependencies to your robot project is non-trivial due to needing to compile for the roboRIO. You have a couple of options.

1. Copy the source code of the wanted library into your robot project.
2. Use the [vendordep template](#) as an example and create a vendordep.

Copying Source Code

Simply copy the necessary source and/or headers into your robot project. You can then configure any necessary platform args like below:

```
nativeUtils.platformConfigs.named("linuxx86-64").configure {  
    it.linker.args.add('-lstdc++fs') // links in C++ filesystem library  
}
```

Creating a Vendordep

Please follow the instructions in the [vendordep repository](#).

34.2 Setting up CI for Robot Code using GitHub Actions

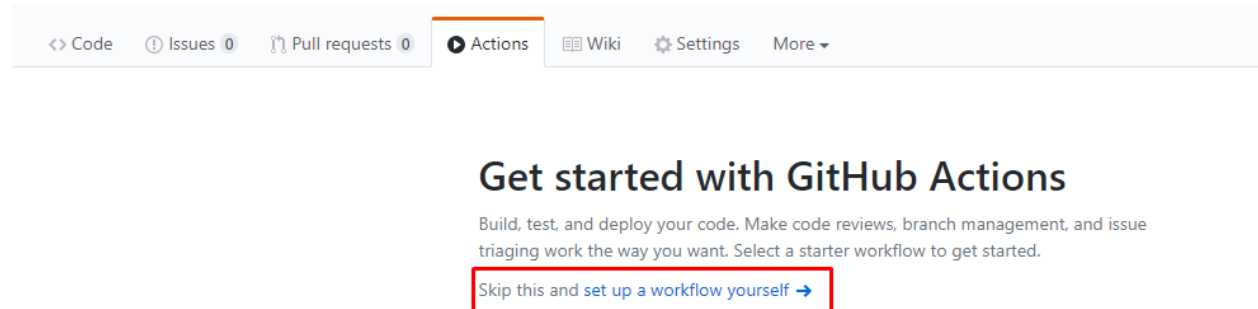
An important aspect of working in a team environment is being able to test code that is pushed to a central repository such as GitHub. For example, a project manager or lead developer might want to run a set of unit tests before merging a pull request or might want to ensure that all code on the main branch of a repository is in working order.

[GitHub Actions](#) is a service that allows for teams and individuals to build and run unit tests on code on various branches and on pull requests. These types of services are more commonly known as “Continuous Integration” services. This tutorial will show you how to setup GitHub Actions on robot code projects.

Note: This tutorial assumes that your team’s robot code is being hosted on GitHub. For an introduction to Git and GitHub, please see this [introduction guide](#).

34.2.1 Creating the Action

The instructions for carrying out the CI process are stored in a YAML file. To create this, click on the “Actions” tab at the top of your repository. Then click on the “set up a workflow yourself” hyperlink.



You will now be greeted with a text editor. Replace all the default text with the following:

```
# This is a basic workflow to build robot code.

name: CI

# Controls when the action will run. Triggers the workflow on push or pull request
# events but only for the main branch.
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

# A workflow run is made up of one or more jobs that can run sequentially or in parallel
jobs:
  # This workflow contains a single job called "build"
  build:
    # The type of runner that the job will run on
    runs-on: ubuntu-latest

    # This grabs the WPILib docker container
    container: wpilib/roborio-cross-ubuntu:2021-18.04

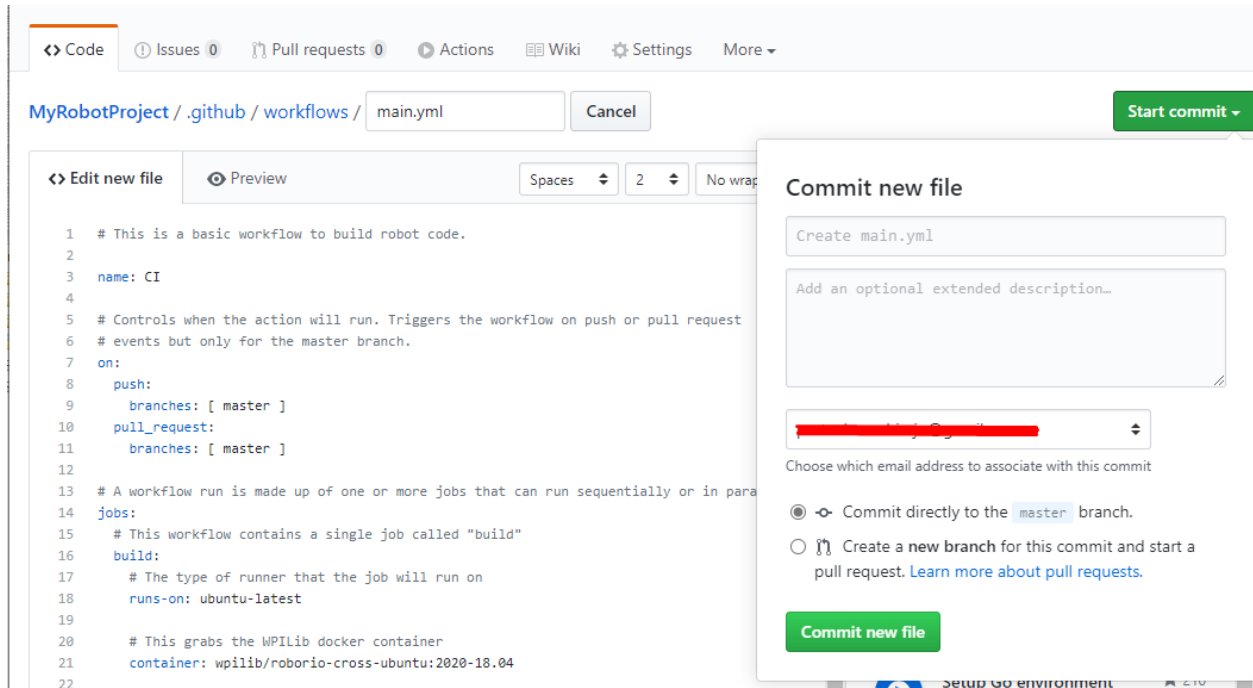
    # Steps represent a sequence of tasks that will be executed as part of the job
    steps:
      # Checks-out your repository under $GITHUB_WORKSPACE, so your job can access it
      - uses: actions/checkout@v2

      # Grant execute permission for gradlew
      - name: Grant execute permission for gradlew
        run: chmod +x gradlew

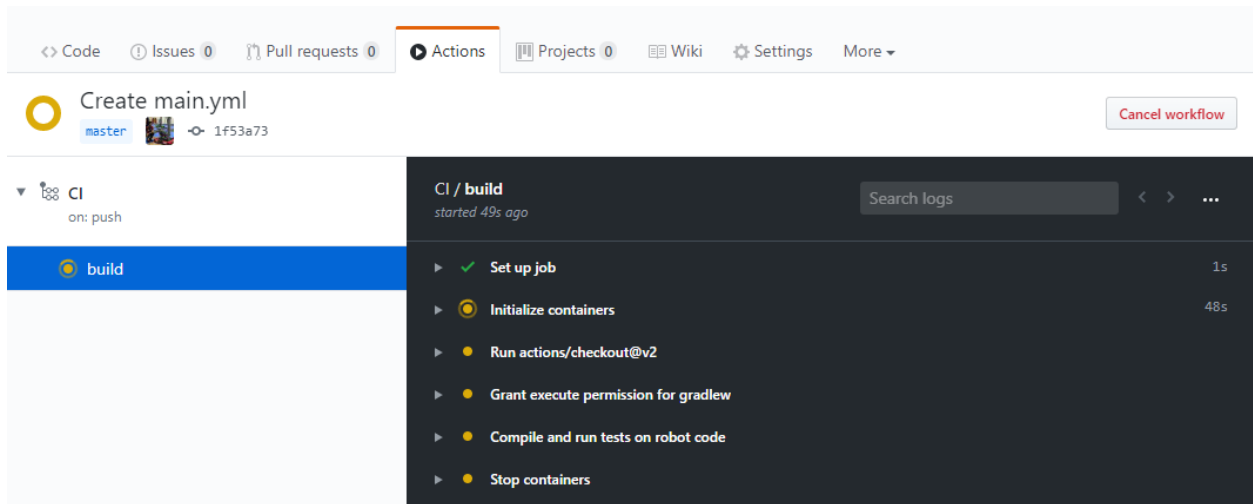
      # Runs a single command using the runners shell
      - name: Compile and run tests on robot code
        run: ./gradlew build
```

Then, save changes by clicking the “Start commit” button on the top-right corner of the

screen. You can amend the default commit message if you wish to do so. Then, click the green “Commit new file” button.



GitHub will now automatically run a build whenever a commit is pushed to main or a pull request is opened. To monitor the status of any build, you can click on the “Actions” tab on the top of the screen.



34.2.2 A Breakdown of the Actions YAML File

Here is a breakdown of the YAML file above. Although a strict understanding of each line is not required, some level of understanding will help you add more features and debug potential issues that may arise.

```
# Controls when the action will run. Triggers the workflow on push or pull request
# events but only for the main branch.
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
```

This block of code dictates when the Action will run. Currently, the action will run when commits are pushed to main or when pull requests are opened against main.

```
# A workflow run is made up of one or more jobs that can run sequentially or in
→parallel
jobs:
  # This workflow contains a single job called "build"
  build:
    # The type of runner that the job will run on
    runs-on: ubuntu-latest

    # This grabs the WPILib docker container
    container: wpilib/roborio-cross-ubuntu:2021-18.04
```

Each Action workflow is made of a one or more jobs that run either sequentially (one after another) or in parallel (at the same time). In our workflow, there is only one “build” job.

We specify that we want the job to run on an Ubuntu virtual machine and in a virtualized Docker container that contains the JDK, C++ compiler and roboRIO toolchains.

```
# Steps represent a sequence of tasks that will be executed as part of the job
steps:
# Checks-out your repository under $GITHUB_WORKSPACE, so your job can access it
- uses: actions/checkout@v2

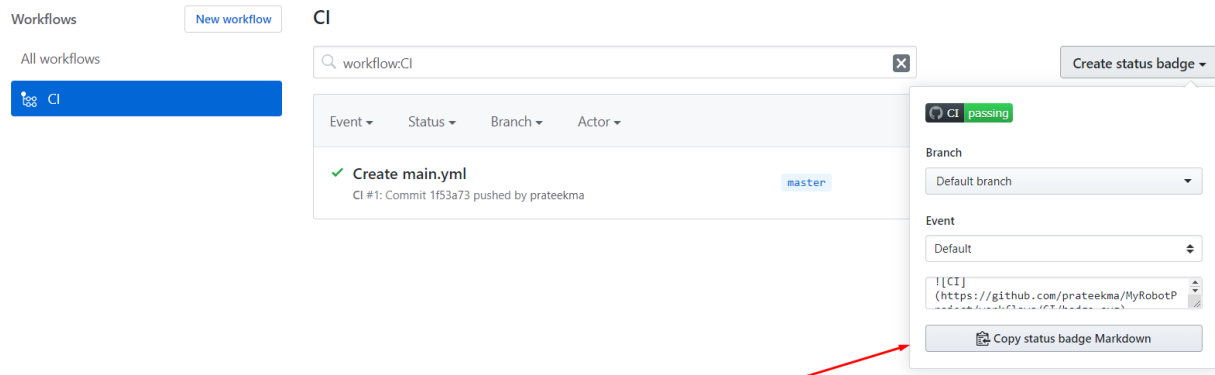
# Grant execute permission for gradlew
- name: Grant execute permission for gradlew
  run: chmod +x gradlew

# Runs a single command using the runners shell
- name: Compile and run tests on robot code
  run: ./gradlew build
```

Each job has certain steps that will be executed. This job has three steps. The first step involves checking out the repository to access the robot code. The second step involves giving the virtual machine permission to execute gradle tasks using `./gradlew`. The final step runs `./gradlew build` to compile robot code and run any unit tests.

34.2.3 Adding a Build Status Badge to a README.md File

It is helpful to add a CI status badge to the top of your repository's README file to quickly check the status of the latest build on main. To do this, click on the "Actions" tab at the top of the screen and select the "CI" tab on the left side of the screen. Then, click on the "Create status badge" button on the top right and copy the status badge Markdown code.



Finally, paste the Markdown code you copied at the top of your README file, commit, and push your changes. Now, you should see the GitHub Actions status badge on your main repository page.

Branch: master		Create new file	Upload files	Find file	Clone or download
prateekma Create README.md		Latest commit 931960c now			
.github/workflows	Create main.yml	18 minutes ago			
.vscode	Initial commit	38 minutes ago			
.wpilib	Initial commit	38 minutes ago			
gradle/wrapper	Initial commit	38 minutes ago			
src	Initial commit	38 minutes ago			
vendordeps	Initial commit	38 minutes ago			
.gitignore	Initial commit	38 minutes ago			
README.md	Create README.md	now			
build.gradle	Initial commit	38 minutes ago			
gradlew	Initial commit	38 minutes ago			
gradlew.bat	Initial commit	38 minutes ago			
settings.gradle	Initial commit	38 minutes ago			

MyRobotProject



A simple example FRC robot code project for setting up Continuous Integration with Azure Pipelines.

34.3 Using a Code Formatter

Code formatters exist to ensure that the style of code written is consistent throughout the entire codebase. This is used in many major projects; from Android to OpenCV. Teams may wish to add a formatter throughout their robot code to ensure that the codebase maintains readability and consistency throughout.

For this article, we will highlight using [Spotless](#) for Java teams and [wpiformat](#) for C++ teams.

34.3.1 Spotless

Configuration

Necessary build.gradle changes are required to get Spotless functional. In the `plugins {}` block of your build.gradle, add the Spotless plugin so it appears similar to the below.

```
plugins {
    id "java"
    id "edu.wpi.first.GradleRIO" version "2021.2.1"
    id 'com.diffplug.spotless' version '5.5.0'
}
```

Then ensure you add a required `spotless {}` block to correctly configure spotless. This can just get placed at the end of your build.gradle.

```
spotless {
    java {
        target fileTree('.') {
            include '**/*.java'
            exclude '**/build/**', '**/build-*/**'
        }
        toggleOffOn()
        googleJavaFormat()
        removeUnusedImports()
        trimTrailingWhitespace()
        endWithNewline()
    }
    groovyGradle {
        target fileTree('.') {
            include '**/*.gradle'
            exclude '**/build/**', '**/build-*/**'
        }
        greclipse()
        indentWithSpaces(4)
        trimTrailingWhitespace()
        endWithNewline()
    }
    format 'xml', {
        target fileTree('.') {
            include '**/*.xml'
            exclude '**/build/**', '**/build-*/**'
        }
        eclipseWtp('xml')
        trimTrailingWhitespace()
    }
}
```

(continues on next page)

(continued from previous page)

```
        indentWithSpaces(2)
        endWithNewline()
    }
    format 'misc', {
        target fileTree('.') {
            include '**/*.md', '**/.gitignore'
            exclude '**/build/**', '**/build-*/**'
        }
        trimTrailingWhitespace()
        indentWithSpaces(2)
        endWithNewline()
    }
}
```

Running Spotless

Spotless can be ran using `./gradlew spotlessApply` which will apply all formatting options. You can also specify a specific task by just adding the name of formatter. An example is `./gradlew spotlessmiscApply`.

Spotless can also be used as a *CI check*. The check is ran with `./gradlew spotlessCheck`.

Explanation of Options

Each format section highlights formatting of custom files in the project. The java and groovyGradle are natively supported by spotless, so they are defined differently.

Breaking this down, we can split this into multiple parts.

- Formatting Java
- Formatting Gradle files
- Formatting XML files
- Formatting Miscellaneous files

They are all similar, except for some small differences that will be explained. The below example will highlight the java `{}` block.

```
java {
    target fileTree('.') {
        include '**/*.java'
        exclude '**/build/**', '**/build-*/**'
    }
    toggleOffOn()
    googleJavaFormat()
    removeUnusedImports()
    trimTrailingWhitespace()
    endWithNewline()
}
```

Let's explain what each of the options mean.


```
target fileTree('.') {
    include '**/*.java'
    exclude '**/build/**', '**/build-*/**'
}
```

The above example tells spotless where our Java classes are and to exclude the build directory. The rest of the options are fairly self-explanatory.

- `toggleOffOn()` adds the ability to have spotless ignore specific portions of a project. The usage looks like the following

```
// format:off

public void myWeirdFunction() {

}

// format:on
```

- `googleJavaFormat()` tells spotless to format according to the [Google Style Guide](#)
- `removeUnusedImports()` will remove any unused imports from any of your java classes
- `trimTrailingWhitespace()` will remove any extra whitespace at the end of your lines
- `endWithNewline()` will add a newline character to the end of your classes

In the `groovyGradle` block, there is a `greclipse` option. This is the formatter that spotless uses to format gradle files.

Additionally, there is a `eclipseWtp` option in the `xml` block. This stands for “Gradle Web Tools Platform” and is the formatter to format xml files. Teams not using any XML files may wish to not include this configuration.

Note: A full list of configurations is available on the [Spotless README](#)

Issues with Line Endings

Spotless will attempt to apply line endings per-OS, which means Git diffs will be constantly changing if two users are on different OSes (Unix vs Windows). It’s recommended that teams who contribute to the same repository from multiple OSes utilize a `.gitattributes` file. The following should suffice for handling line endings.

```
*.gradle text eol=lf
*.java text eol=lf
*.md text eol=lf
*.xml text eol=lf
```

34.3.2 wpiformat

Requirements

- [Python 3.6 or higher](#)
- clang-format (included with [LLVM](#))

Important: Windows is not currently supported at this time! Installing LLVM with Clang **will** break normal robot builds if installed on Windows.

You can install [wpiformat](#) by typing `pip3 install wpiformat` into a terminal or command prompt.

Usage

`wpiformat` can be ran by typing `wpiformat` in a console. This will format with `clang-format`. Three configuration files are required (`.clang-format`, `.styleguide`, `.styleguide-license`). These must exist in the project root.

- `.clang-format`: [Download](#)
- `.styleguide-license`: [Download](#)

An example styleguide is shown below:

```
cppHeaderFileInclude {
  \.h$
  \.hpp$
  \.inc$
  \.inl$
}

cppSrcFileInclude {
  \.cpp$
}

modifiableFileExclude {
  gradle/
}
```

Note: Teams can adapt `.styleguide` and `.styleguide-license` however they wish. It's important that these are not deleted, as they are required to run `wpiformat`!

You can turn this into a [CI check](#) by running `git --no-pager diff --exit-code HEAD`. It can be configured with a `.clang-format` configuration file. An example configuration file is provided below.

Below is an example GitHub Actions check that uses `wpiformat`

```
wpiformat:
  name: "wpiformat"
  runs-on: ubuntu-latest
```

(continues on next page)

(continued from previous page)

```
steps:
- uses: actions/checkout@v2
- name: Fetch all history and metadata
  run: |
    git fetch --prune --unshallow
    git checkout -b pr
    git branch -f main origin/main
- name: Set up Python 3.8
  uses: actions/setup-python@v2
  with:
    python-version: 3.8
- name: Install clang-format
  run: sudo apt-get update -q && sudo apt-get install -y clang-format-10
- name: Install wpiformat
  run: pip3 install wpiformat
- name: Run
  run: wpiformat -clang 10
- name: Check Output
  run: git --no-pager diff --exit-code HEAD
```


This section covers advanced control features in WPILib, such as various feedback/feedforward control algorithms and trajectory following.

35.1 A Video Walkthrough of Model Based Validation of Autonomous in FRC

At the “RSN Spring Conference, Presented by WPI” in 2020, Tyler Veness from the WPILib team gave a presentation on Model Based Validation of Autonomous in FRC®.

The link to the presentation is available [here](#).

35.2 Advanced Controls Introduction

35.2.1 Control System Basics

Note: This article is taken out of [Controls Engineering in FRC](#) by Tyler Veness with permission.

Control systems are all around us and we interact with them daily. A small list of ones you may have seen includes heaters and air conditioners with thermostats, cruise control and the anti-lock braking system (ABS) on automobiles, and fan speed modulation on modern laptops. Control systems monitor or control the behavior of systems like these and may consist of humans controlling them directly (manual control), or of only machines (automatic control).

How can we prove closed-loop controllers on an autonomous car, for example, will behave safely and meet the desired performance specifications in the presence of uncertainty? Control theory is an application of algebra and geometry used to analyze and predict the behavior of systems, make them respond how we want them to, and make them robust to disturbances and uncertainty.

Controls engineering is, put simply, the engineering process applied to control theory. As such, it's more than just applied math. While control theory has some beautiful math behind it, controls engineering is an engineering discipline like any other that is filled with trade-offs. The solutions control theory gives should always be sanity checked and informed by our performance specifications. We don't need to be perfect; we just need to be good enough to meet our specifications.

Nomenclature

Most resources for advanced engineering topics assume a level of knowledge well above that which is necessary. Part of the problem is the use of jargon. While it efficiently communicates ideas to those within the field, new people who aren't familiar with it are lost.

The system or collection of actuators being controlled by a control system is called the *plant*. A controller is used to drive the plant from its current state to some desired state (the reference). Controllers which don't include information measured from the plant's output are called open-loop controllers.

Controllers which incorporate information fed back from the plant's output are called closed-loop controllers or feedback controllers.

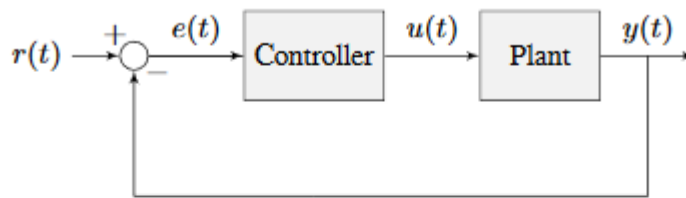


Figure 1.1: Control system nomenclature for a closed-loop system

$r(t)$	reference	$u(t)$	control input
$e(t)$	error	$y(t)$	output

Note: The input and output of a system are defined from the plant's point of view. The negative feedback controller shown is driving the difference between the reference and output, also known as the error, to zero.

What is Gain?

Gain is a proportional value that shows the relationship between the magnitude of an input signal to the magnitude of an output signal at steady-state. Many systems contain a method by which the gain can be altered, providing more or less “power” to the system.

The figure below shows a system with a hypothetical input and output. Since the output is twice the amplitude of the input, the system has a gain of two.

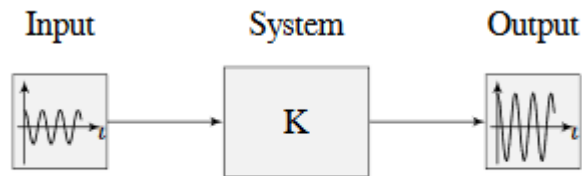


Figure 1.2: Demonstration of system with a gain of $K = 2$

Block Diagrams

When designing or analyzing a control system, it is useful to model it graphically. Block diagrams are used for this purpose. They can be manipulated and simplified systematically.

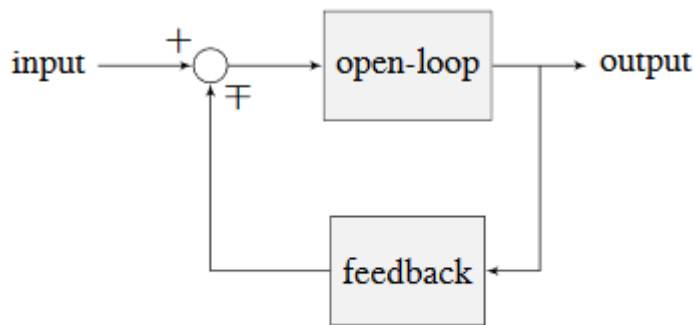


Figure 1.3: Block diagram with nomenclature

The open-loop gain is the total gain from the sum node at the input (the circle) to the output branch. This would be the system's gain if the feedback loop was disconnected. The feedback gain is the total gain from the output back to the input sum node. A sum node's output is the sum of its inputs.

The below figure is a block diagram with more formal notation in a feedback configuration.

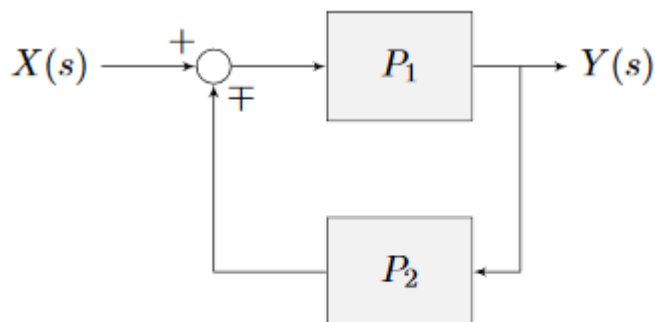


Figure 1.4: Feedback block diagram

\mp means “minus or plus” where a minus represents negative feedback.

Why Feedback Control?

Let's say we are controlling a DC brushed motor. With just a mathematical model and knowledge of all the current states of the system (i.e., angular velocity), we can predict all future states given the future voltage inputs. Why then do we need feedback control? If the system is disturbed in any way that isn't modeled by our equations, like a load was applied, or voltage sag in the rest of the circuit caused the commanded voltage to not match the applied voltage, the angular velocity of the motor will deviate from the model over time.

To combat this, we can take measurements of the system and the environment to detect this deviation and account for it. For example, we could measure the current position and estimate an angular velocity from it. We can then give the motor corrective commands as well as steer our model back to reality. This feedback allows us to account for uncertainty and be robust to it.

35.2.2 PID Introduction Video by WPI

Have you ever had trouble designing a robot system to move quickly and then stop at exactly a desired position? Challenges like this can arise when driving fixed distances or speeds, operating an arm or elevator, or any other motor controlled system that requires specific motion. In this video, WPI Professor Dmitry Berenson talks about robot controls and how PID controls work.

35.2.3 Introduction to PID

The PID controller is a commonly used feedback controller consisting of proportional, integral, and derivative terms, hence the name. This article will build up the definition of a PID controller term by term while trying to provide some intuition for how each of them behaves.

First, we'll get some nomenclature for PID controllers out of the way. The *reference* is called the setpoint (the desired position) and the *output* is called the *process variable* (the measured position). Below are some common variable naming conventions for relevant quantities.

$r(t)$	<i>setpoint</i>	$u(t)$	<i>control input</i>
$e(t)$	<i>error</i>	$y(t)$	<i>output</i>

The *error* $e(t)$ is $r(t) - y(t)$.

For those already familiar with PID control, this book's interpretation won't be consistent with the classical intuition of "past", "present", and "future" error. We will be approaching it from the viewpoint of modern control theory with proportional controllers applied to different physical quantities we care about. This will provide a more complete explanation of the derivative term's behavior for constant and moving *setpoints*.

The proportional term drives the position error to zero, the derivative term drives the velocity error to zero, and the integral term accumulates the area between the *setpoint* and *output* plots over time (the integral of position *error*) and adds the current total to the *control input*. We'll go into more detail on each of these.

Proportional Term

The *Proportional* term drives the position error to zero.

$$u(t) = K_p e(t)$$

where K_p is the proportional gain and $e(t)$ is the error at the current time t .

The below figure shows a block diagram for a *system* controlled by a P controller.

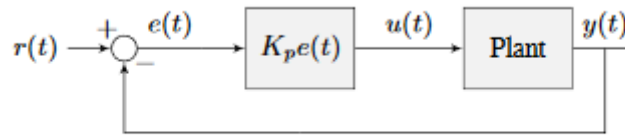


Figure 2.1: P controller block diagram

Proportional gains act like a “software-defined springs” that pull the *system* toward the desired position. Recall from physics that we model springs as $F = -kx$ where F is the force applied, k is a proportional constant, and x is the displacement from the equilibrium point. This can be written another way as $F = k(0 - x)$ where 0 is the equilibrium point. If we let the equilibrium point be our feedback controller’s *setpoint*, the equations have a one to one correspondence.

$$F = k(r - x)$$

$$u(t) = K_p e(t) = K_p (r(t) - y(t))$$

so the “force” with which the proportional controller pulls the *system’s output* toward the *setpoint* is proportional to the *error*, just like a spring.

Derivative Term

The *Derivative* term drives the velocity error to zero.

$$u(t) = K_p e(t) + K_d \frac{de}{dt}$$

where K_p is the proportional gain, K_d is the derivative gain, and $e(t)$ is the error at the current time t .

The below figure shows a block diagram for a *system* controlled by a PD controller.

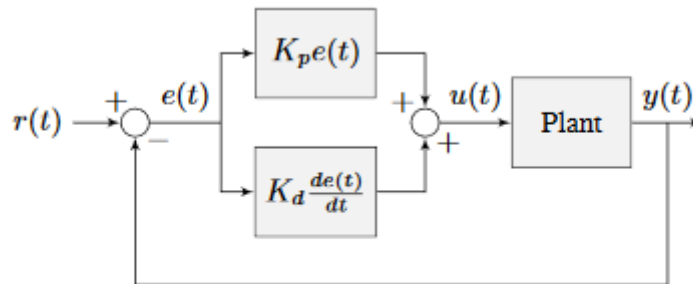


Figure 2.2: PD controller block diagram

A PD controller has a proportional controller for position (K_p) and a proportional controller for velocity (K_d). The velocity *setpoint* is implicitly provided by how the position *setpoint* changes over time. To prove this, we will rearrange the equation for a PD controller.

$$u_k = K_p e_k + K_d \frac{e_k - e_{k-1}}{dt}$$

where u_k is the *control input* at timestep k and e_k is the *error* at timestep k . e_k is defined as $e_k = r_k - x_k$ where r_k is the *setpoint* and x_k is the current *state* at timestep k .

$$\begin{aligned} u_k &= K_p(r_k - x_k) + K_d \frac{(r_k - x_k) - (r_{k-1} - x_{k-1})}{dt} \\ u_k &= K_p(r_k - x_k) + K_d \frac{r_k - x_k - r_{k-1} + x_{k-1}}{dt} \\ u_k &= K_p(r_k - x_k) + K_d \frac{r_k - r_{k-1} - x_k + x_{k-1}}{dt} \\ u_k &= K_p(r_k - x_k) + K_d \frac{(r_k - r_{k-1}) - (x_k - x_{k-1})}{dt} \\ u_k &= K_p(r_k - x_k) + K_d \left(\frac{r_k - r_{k-1}}{dt} - \frac{x_k - x_{k-1}}{dt} \right) \end{aligned}$$

Notice how $\frac{r_k - r_{k-1}}{dt}$ is the velocity of the *setpoint*. By the same reason, $\frac{x_k - x_{k-1}}{dt}$ is the *system's* velocity at a given timestep. That means the K_d term of the PD controller is driving the estimated velocity to the *setpoint* velocity.

If the *setpoint* is constant, the implicit velocity *setpoint* is zero, so the K_d term slows the *system* down if it's moving. This acts like a “software-defined damper”. These are commonly seen on door closers, and their damping force increases linearly with velocity.

Integral Term

Important: Integral gain is generally not recommended for FRC® use. There are better approaches to fix *steady-state error* like using feedforwards or constraining when the integral control acts using other knowledge of the *system*.

The *Integral* term accumulates the area between the *setpoint* and *output* plots over time (i.e., the integral of position *error*) and adds the current total to the *control input*. Accumulating the area between two curves is called integration.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau$$

where K_p is the proportional gain, K_i is the integral gain, $e(t)$ is the error at the current time t , and τ is the integration variable.

The Integral integrates from time 0 to the current time t . we use τ for the integration because we need a variable to take on multiple values throughout the integral, but we can't use t because we already defined that as the current time.

The below figure shows a block diagram for a *system* controlled by a PI controller.

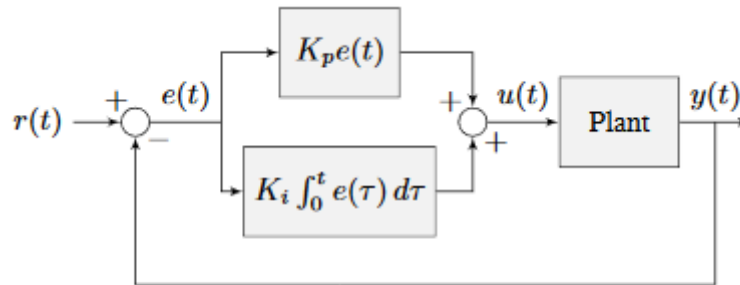


Figure 2.3: PI controller block diagram

When the *system* is close the *setpoint* in steady-state, the proportional term may be too small to pull the *output* all the way to the *setpoint*, and the derivative term is zero. This can result in *steady-state error* as shown in figure 2.4

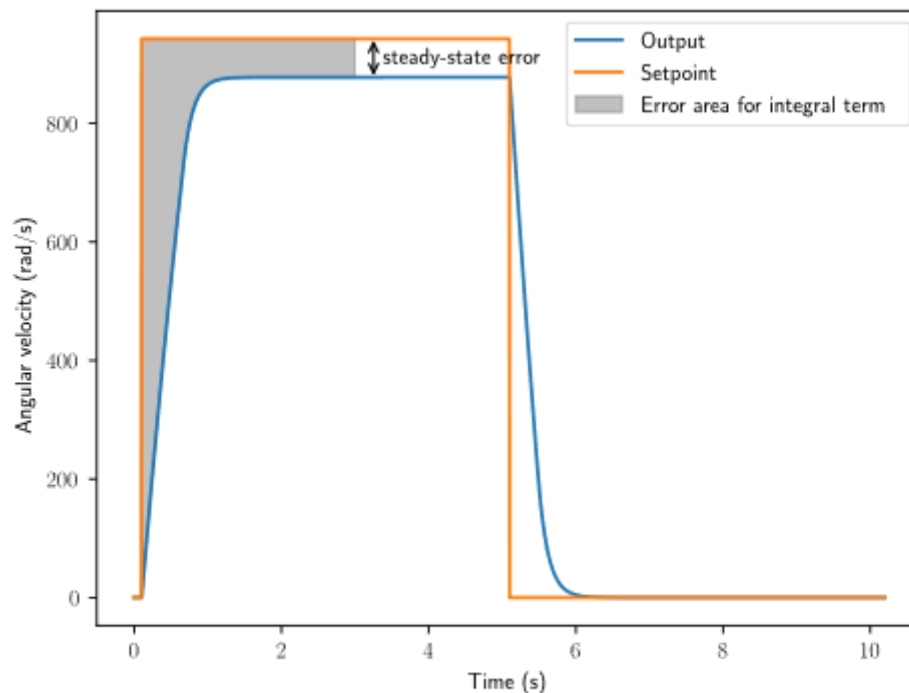


Figure 2.4: P controller with steady-state error

A common way of eliminating *steady-state error* is to integrate the *error* and add it to the *control input*. This increases the *control effort* until the *system* converges. Figure 2.4 shows an example of *steady-state error* for a flywheel, and figure 2.5 shows how an integrator added to the flywheel controller eliminates it. However, too high of an integral gain can lead to overshoot, as shown in figure 2.6.

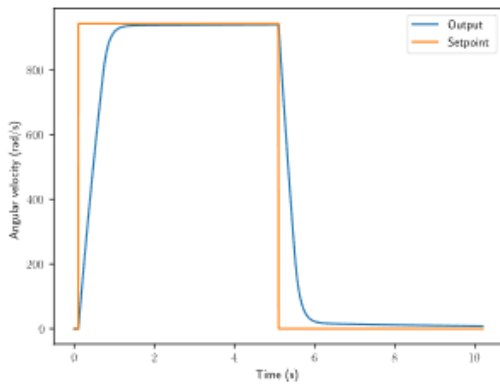


Figure 2.5: PI controller without steady-state error

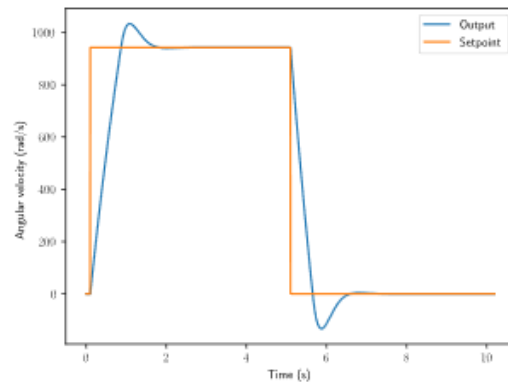


Figure 2.6: PI controller with overshoot from large K_i gain

PID Controller Definition

Note: For information on using the WPILib provided PIDController, see the [relevant article](#).

When these terms are combined, one gets the typical definition for a PID controller.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de}{dt}$$

where K_p is the proportional gain, K_i is the integral gain, K_d is the derivative gain, $e(t)$ is the error at the current time t , and τ is the integration variable.

The below figure shows a block diagram for a PID controller.

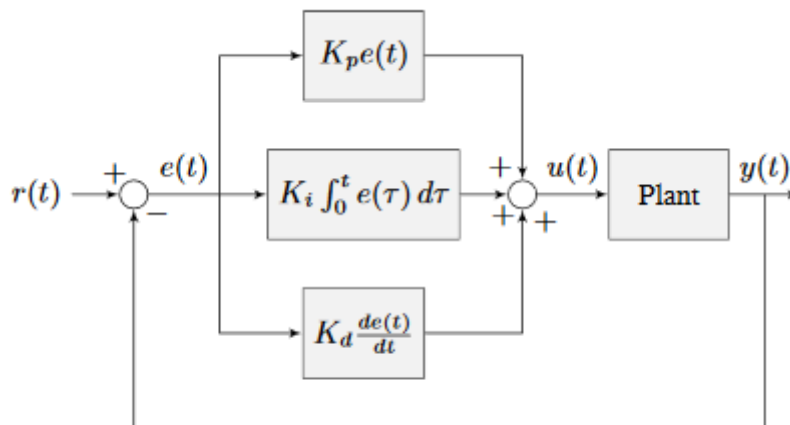


Figure 2.7: PID controller block diagram

Response Types

A *system* driven by a PID controller generally has three types of responses: underdamped, over-damped, and critically damped. These are shown in figure 2.8.

For the *step responses* in figure 2.7, *rise time* is the time the *system* takes to initially reach the reference after applying the *step input*. *Settling time* is the time the *system* takes to settle at the *reference* after the *step input* is applied.

An *underdamped* response oscillates around the *reference* before settling. An *overdamped* response

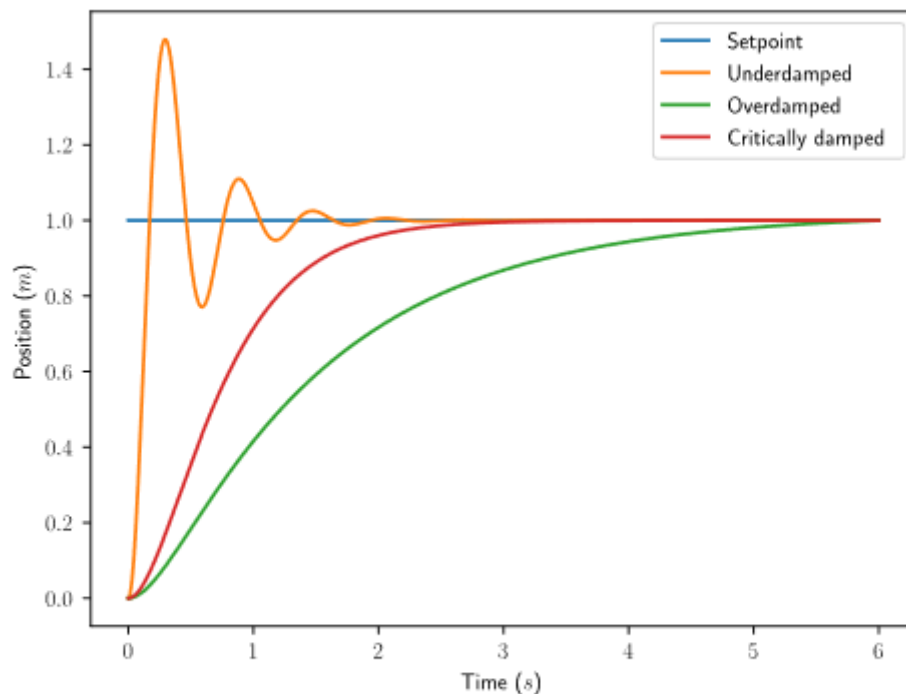


Figure 2.8: PID controller response types

is slow to rise and does not overshoot the *reference*. A *critically damped* response has the fastest *rise time* without overshooting the *reference*.

35.2.4 Tuning a PID Controller

These steps apply to position PID controllers. Velocity PID controllers typically don't need K_d .

1. Set K_p , K_i , and K_d to zero.
2. Increase K_p until the *output* starts to oscillate around the *setpoint*.
3. Increase K_d as much as possible without introducing jittering in the *system response*.

Plot the position *setpoint*, velocity *setpoint*, measured position, and measured velocity. The velocity *setpoint* can be obtained via numerical differentiation of the position *setpoint* (i.e.,

$v_{desired,k} = \frac{r_k - r_{k-1}}{\Delta t}$). Increase K_p until the position tracks well, then increase K_d until the velocity tracks well.

If the *controller* settles at an *output* above or below the *setpoint*, one can increase K_i such that the *controller* reaches the *setpoint* in a reasonable amount of time. However, a steady-state feedforward is strongly preferred over integral control (especially for PID control).

Important: Adding an integral gain to the *controller* is an incorrect way to eliminate *steady-state error*. A better approach would be to tune it with an integrator added to the *plant*, but this requires a *model*. Since we are doing output-based rather than model-based control, our only option is to add an integrator to the *controller*.

Beware that if K_i is too large, integral windup can occur. Following a large change in *setpoint*, the integral term can accumulate an error larger than the maximal *control input*. As a result, the system overshoots and continues to increase until this accumulated error is unwound.

Note: The *frd-characterization toolsuite* can be used to model your system and give accurate Proportional and Derivative values. This is preferred over tuning the controller yourself.

Actuator Saturation

A controller calculates its output based on the error between the *reference* and the current *state*. *Plant* in the real world don't have unlimited control authority available for the controller to apply. When the actuator limits are reached, the controller acts as if the gain has been temporarily reduced.

We'll try to explain this through a bit of math. Let's say we have a controller $u = k(r - x)$ where u is the *control effort*, k is the gain, r is the *reference*, and x is the current state. Let u_{max} be the limit of the actuator's output which is less than the uncapped value of u and k_{max} be the associated maximum gain. We will now compare the capped and uncapped controllers for the same *reference* and current *state*.

$$\begin{aligned}u_{max} &< u \\k_{max}(r - x) &< k(r - x) \\k_{max} &< k\end{aligned}$$

For the inequality to hold, k_{max} must be less than the original value for k . This reduced gain is evident in a *system response* when there is a linear change in state instead of an exponential one as it approaches the *reference*. This is due to the *control effort* no longer following a decaying exponential plot. Once the *system* is closer to the *reference*, the controller will stop saturating and produce realistic controller values again.

35.3 Filters

Note: The data used to generate the various demonstration plots in this section can be found [here](#).

This section describes a number of filters included with WPILib that are useful for noise reduction and/or input smoothing.

35.3.1 Introduction to Filters

Filters are some of the most common tools used in modern technology, and find numerous applications in robotics in both signal processing and controls. Understanding the notion of a filter is crucial to understanding the utility of the various types of filters provided by WPILib.

What Is a Filter?

Note: For the sake of this article, we will assume all data are single-dimensional time-series data. Obviously, the concepts involved are more general than this - but a full/rigorous discussion of signals and filtering is out of the scope of this documentation.

So, what exactly *is* a filter, then? Simply put, a filter is a mapping from a stream of inputs to a stream of outputs. That is to say, the value output by a filter (in principle) can depend not only on the *current* value of the input, but on *the entire set of past and future values* (of course, in practice, the filters provided by WPILib are implementable in real-time on streaming data; accordingly, they can only depend on the *past* values of the input, and not on future values). This is an important concept, because generally we use filters to remove/mitigate unwanted *dynamics* from a signal. When we filter a signal, we're interested in modifying *how the signal changes over time*.

Effects of Using a Filter

Noise Reduction

One of the most typical uses of a filter is for noise reduction. A filter that reduces noise is called a *low-pass* filter (because it allows low frequencies to “pass through,” while blocking high-frequencies). Most of the filters currently included in WPILib are effectively low-pass filters.

Rate Limiting

Filters are also commonly used to reduce the rate at which a signal can change. This is closely related to noise reduction, and filters that reduce noise also tend to limit the rate of change of their output.

Edge Detection

The counterpart to the low-pass filter is the high-pass filter, which only permits high frequencies to pass through to the output. High-pass filters can be somewhat tricky to build intuition for, but a common usage for a high-pass filter is edge-detection - since high-pass filters will reflect sudden changes in the input while ignoring slower changes, they are useful for determining the location of sharp discontinuities in the signal.

Phase Lag

An unavoidable negative effect of a real-time low-pass filter is the introduction of “phase lag.” Since, as mentioned earlier, a real-time filter can only depend on past values of the signal (we cannot time-travel to obtain the future values), the filtered value takes some time to “catch up” when the input starts changing. The greater the noise-reduction, the greater the introduced delay. This is, in many ways, *the* fundamental trade-off of real-time filtering, and should be the primary driving factor of your filter design.

Interestingly, high-pass filters introduce a phase *lead*, as opposed to a phase lag, as they exacerbate local changes to the value of the input.

35.3.2 Linear Filters

The first (and most commonly-employed) sort of filter that WPILib supports is a *linear filter* - or, more specifically, a linear time-invariant (LTI) filter.

An LTI filter is, put simply, a weighted moving average - the value of the output stream at any given time is a localized, weighted average of the inputs near that time. The difference between different types of LTI filters is thus reducible to the difference in the choice of the weighting function (also known as a “window function” or an “impulse response”) used. Mathematically, this kind of moving average is known as a [convolution](#).

There are two broad “sorts” of impulse responses: infinite impulse responses (IIR), and finite impulse responses (FIR).

Infinite impulse responses have infinite “support” - that is, they are nonzero over an infinitely-large region. This means, broadly, that they also have infinite “memory” - once a value appears in the input stream, it will influence *all subsequent outputs, forever*. This is typically undesirable from a strict signal-processing perspective, however filters with infinite impulse responses tend to be very easy to compute as they can be expressed by simple recursion relations.

Finite impulse responses have finite “support” - that is, they are nonzero on a bounded region. The “archetypical” FIR filter is a flat moving average - that is, simply setting the output equal to the average of the past n inputs. FIR filters tend to have more-desirable properties than IIR filters, but are more costly to compute.

Linear filters are supported in WPILib through the `LinearFilter` class ([Java](#), [C++](#)).

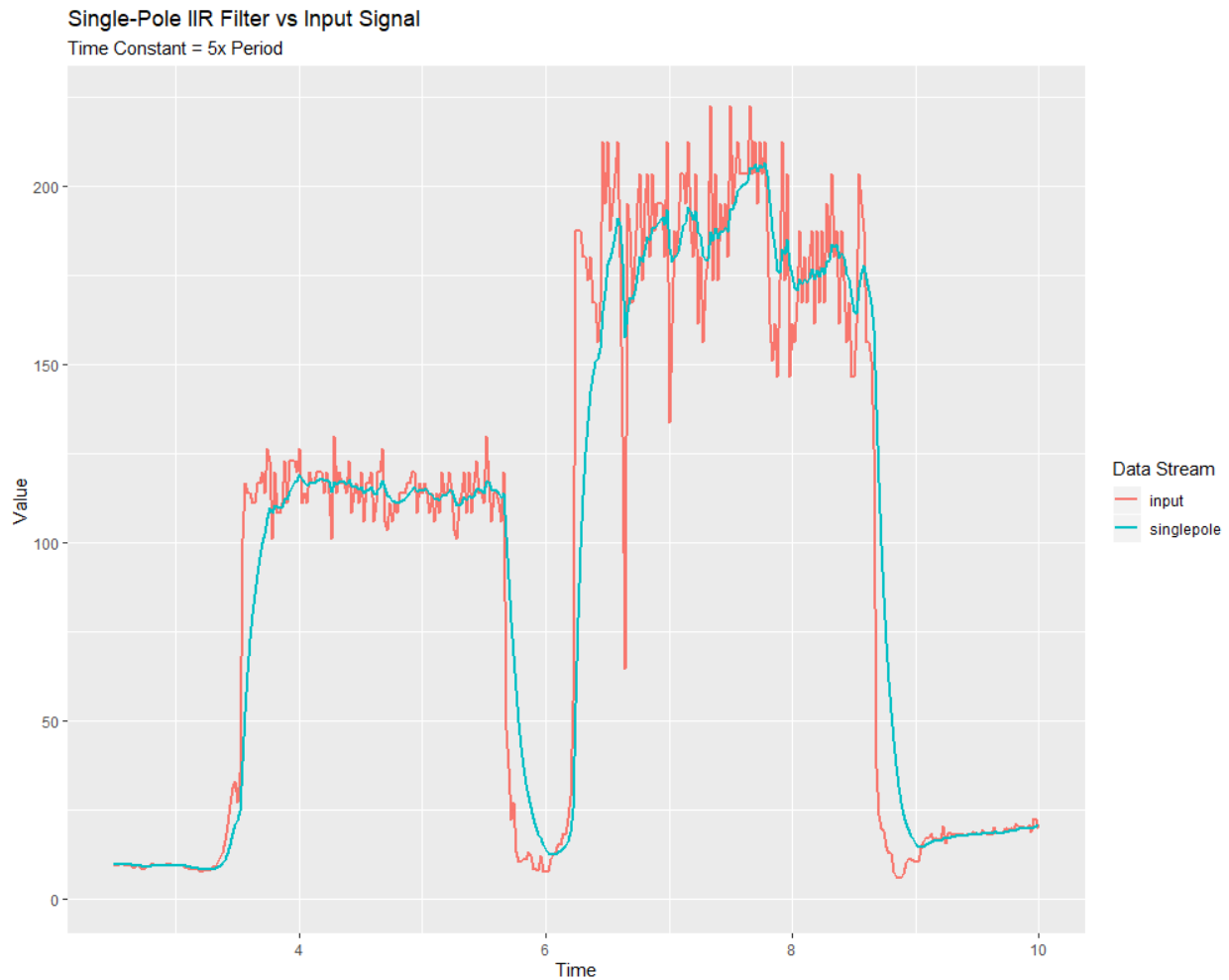
Creating a LinearFilter

Note: The C++ LinearFilter class is templated on the data type used for the input.

Note: Because filters have “memory”, each input stream requires its own filter object. Do *not* attempt to use the same filter object for multiple input streams.

While it is possible to directly instantiate LinearFilter class to build a custom filter, it is far more convenient (and common) to use one of the supplied factory methods, instead:

singlePoleIIR



The singlePoleIIR() factory method creates a single-pole infinite impulse response filter (also known as [exponential smoothing](#), on account of having an exponential impulse response). This is the “go-to,” “first-try” low-pass filter in most applications; it is computationally trivial and works in most cases.

Java

C++

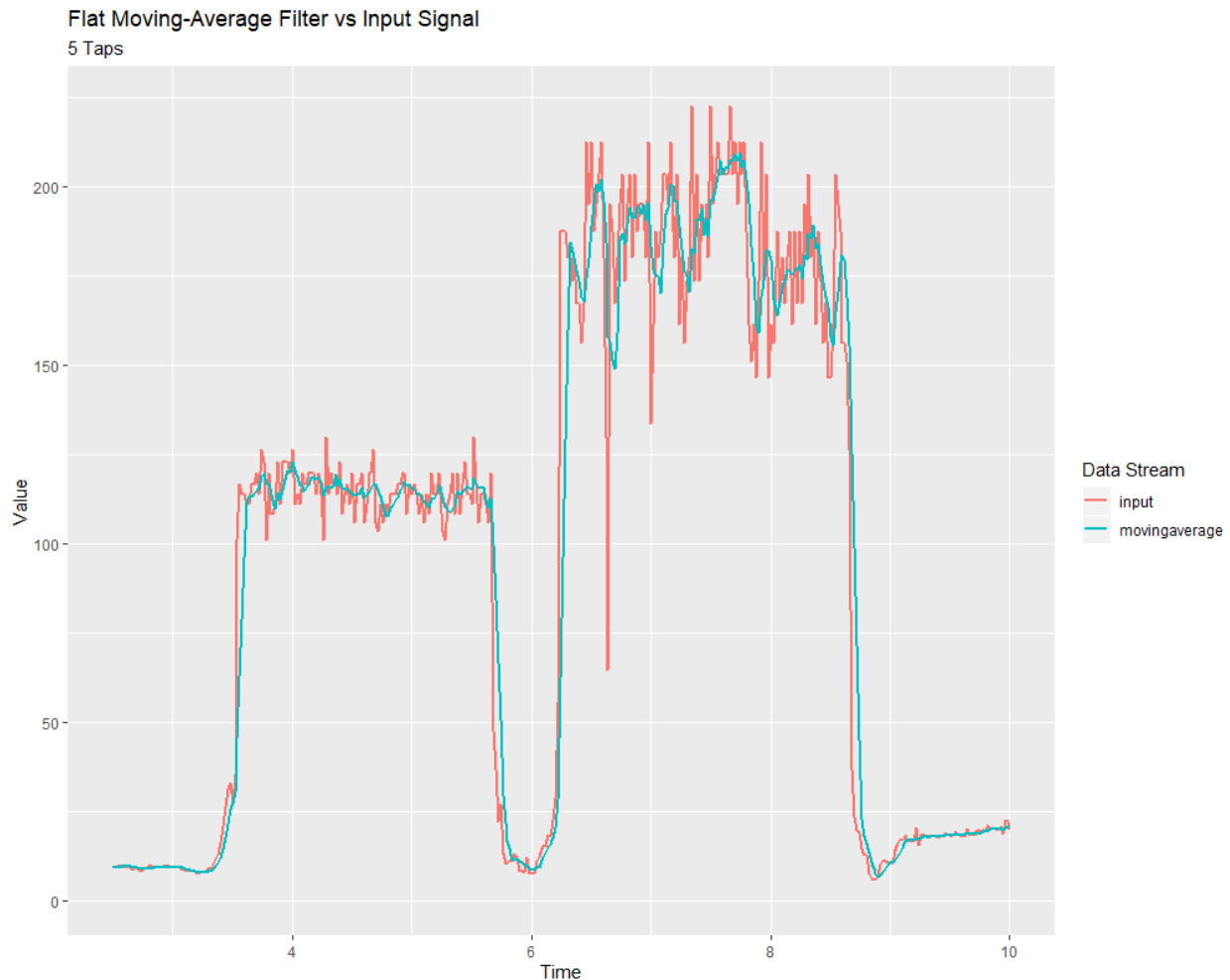
```
// Creates a new Single-Pole IIR filter
// Time constant is 0.1 seconds
// Period is 0.02 seconds - this is the standard FRC main loop period
LinearFilter filter = LinearFilter.singlePoleIIR(0.1, 0.02);
```

```
// Creates a new Single-Pole IIR filter
// Time constant is 0.1 seconds
// Period is 0.02 seconds - this is the standard FRC main loop period
frc::LinearFilter<double> filter = frc::LinearFilter<double>::SinglePoleIIR(0.1_s, 0.
↪02_s);
```

The “time constant” parameter determines the “characteristic timescale” of the filter’s impulse response; the filter will cancel out any signal dynamics that occur on timescales significantly shorter than this. Relatedly, it is also the approximate timescale of the introduced *phase lag*. The reciprocal of this timescale, multiplied by 2 pi, is the “cutoff frequency” of the filter.

The “period” parameter is the period at which the filter’s `calculate()` method will be called. For the vast majority of implementations, this will be the standard main robot loop period of 0.02 seconds.

movingAverage



The `movingAverage` factory method creates a simple flat moving average filter. This is the simplest possible low-pass FIR filter, and is useful in many of the same contexts as the single-pole IIR filter. It is somewhat more costly to compute, but generally behaves in a somewhat nicer manner.

Java

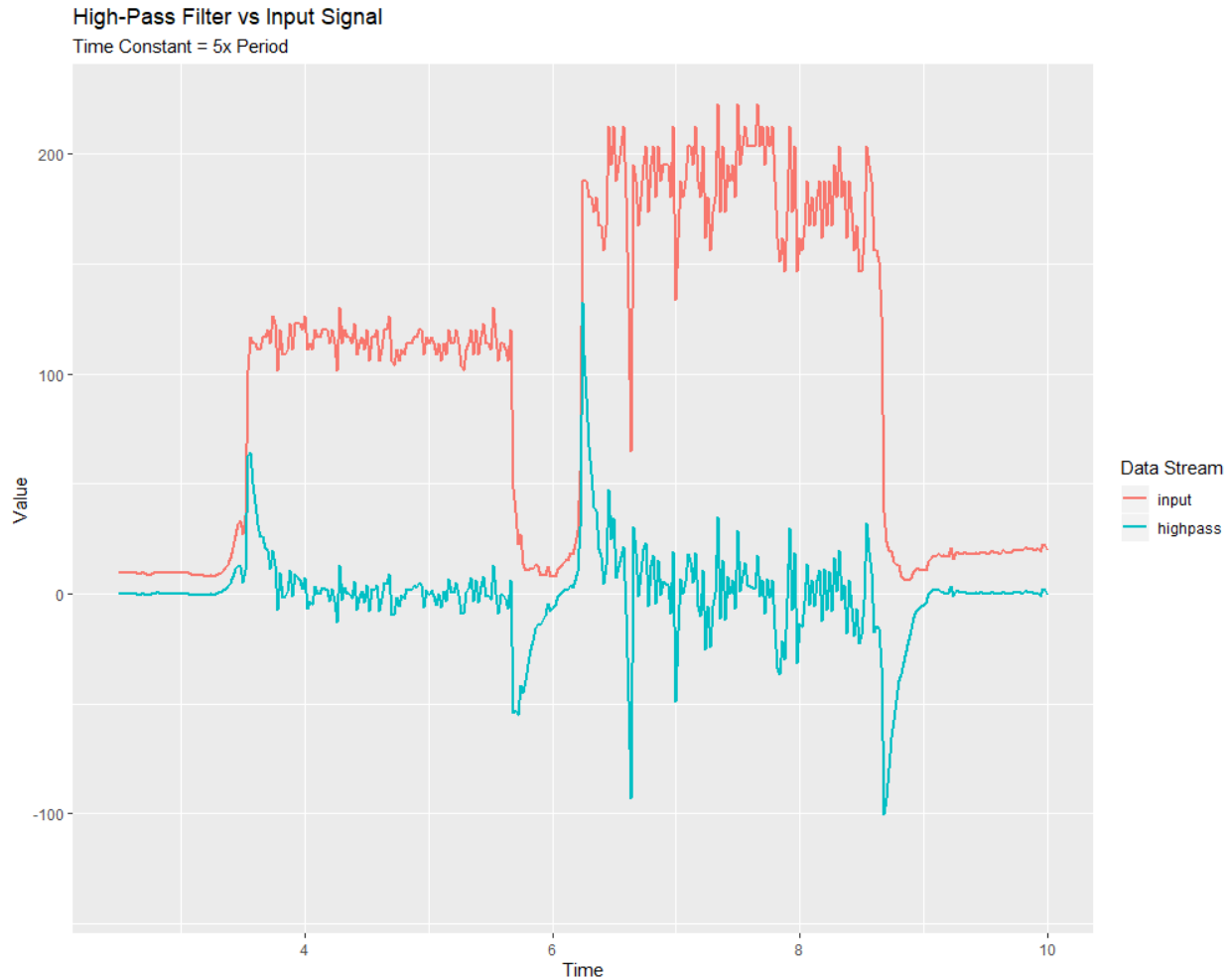
C++

```
// Creates a new flat moving average filter
// Average will be taken over the last 5 samples
LinearFilter filter = LinearFilter.movingAverage(5);
```

```
// Creates a new flat moving average filter
// Average will be taken over the last 5 samples
frc::LinearFilter<double> filter = frc::LinearFilter<double>::MovingAverage(5);
```

The “taps” parameter is the number of samples that will be included in the flat moving average. This behaves similarly to the “time constant” above - the effective time constant is the number of taps times the period at which `calculate()` is called.

highPass



The `highPass` factory method creates a simple first-order infinite impulse response high-pass filter. This is the “counterpart” to the [singlePoleIIR](#).

Java

C++

```
// Creates a new high-pass IIR filter
// Time constant is 0.1 seconds
// Period is 0.02 seconds - this is the standard FRC main loop period
LinearFilter filter = LinearFilter.highPass(0.1, 0.02);
```

```
// Creates a new high-pass IIR filter
// Time constant is 0.1 seconds
// Period is 0.02 seconds - this is the standard FRC main loop period
frc::LinearFilter<double> filter = frc::LinearFilter<double>::HighPass(0.1_s, 0.02_s);
```

The “time constant” parameter determines the “characteristic timescale” of the filter’s impulse response; the filter will cancel out any signal dynamics that occur on timescales significantly longer than this. Relatedly, it is also the approximate timescale of the introduced [phase lead](#). The reciprocal of this timescale, multiplied by 2 pi, is the “cutoff frequency” of

the filter.

The “period” parameter is the period at which the filter’s `calculate()` method will be called. For the vast majority of implementations, this will be the standard main robot loop period of 0.02 seconds.

Using a LinearFilter

Note: In order for the created filter to obey the specified timescale parameter, its `calculate()` function *must* be called regularly at the specified period. If, for some reason, a significant lapse in `calculate()` calls must occur, the filter’s `reset()` method should be called before further use.

Once your filter has been created, using it is easy - simply call the `calculate()` method with the most recent input to obtain the filtered output:

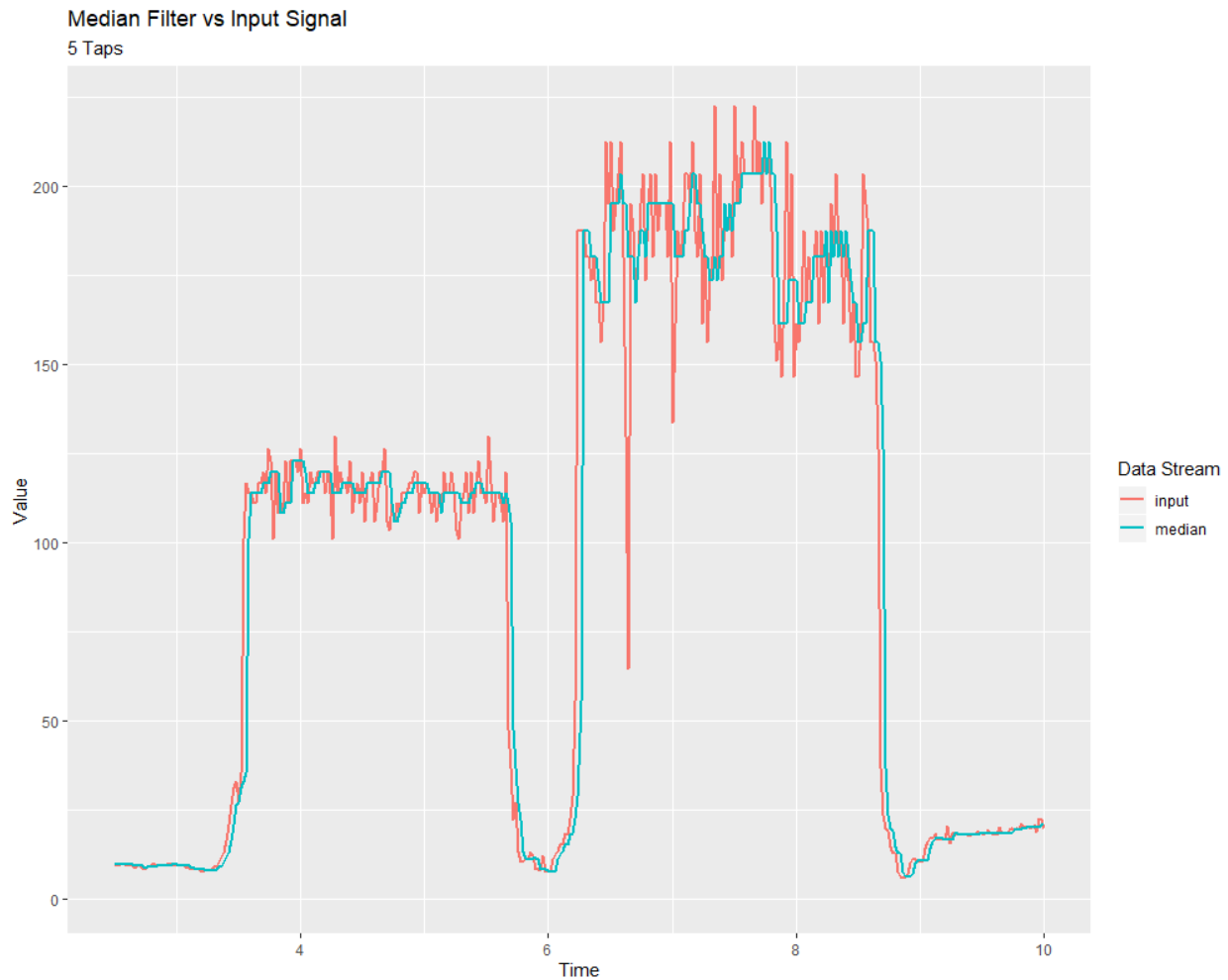
Java

C++

```
// Calculates the next value of the output
filter.calculate(input);
```

```
// Calculates the next value of the output
filter.Calculate(input);
```

35.3.3 Median Filter



A **robust** alternative to the *moving-average filter* is the *median filter*. Where a moving average filter takes the arithmetic *mean* of the input over a moving sample window, a median filter (per the name) takes a median instead.

The median filter is most-useful for removing occasional outliers from an input stream. This makes it particularly well-suited to filtering inputs from distance sensors, which are prone to occasional interference. Unlike a moving average, the median filter will remain completely unaffected by small numbers of outliers, no matter how extreme.

The median filter is supported in WPILib through the `MedianFilter` class ([Java](#), [C++](#)).

Creating a MedianFilter

Note: The C++ MedianFilter class is templated on the data type used for the input.

Note: Because filters have “memory”, each input stream requires its own filter object. Do *not* attempt to use the same filter object for multiple input streams.

Creating a MedianFilter is simple:

Java

C++

```
// Creates a MedianFilter with a window size of 5 samples
MedianFilter filter = new MedianFilter(5);
```

```
// Creates a MedianFilter with a window size of 5 samples
frc::MedianFilter<double> filter(5);
```

Using a MedianFilter

Once your filter has been created, using it is easy - simply call the `calculate()` method with the most recent input to obtain the filtered output:

Java

C++

```
// Calculates the next value of the output
filter.calculate(input);
```

```
// Calculates the next value of the output
filter.Calculate(input);
```

35.3.4 Slew Rate Limiter

A common use for filters in FRC® is to soften the behavior of control inputs (for example, the joystick inputs from your driver controls). Unfortunately, a simple low-pass filter is poorly-suited for this job; while a low-pass filter will soften the response of an input stream to sudden changes, it will also wash out fine control detail and introduce phase lag. A better solution is to limit the rate-of-change of the control input directly. This is performed with a *slew rate limiter* - a filter that caps the maximum rate-of-change of the signal.

A slew rate limiter can be thought of as a sort of primitive motion profile. In fact, the slew rate limiter is the first-order equivalent of the *Trapezoidal Motion Profile* supported by WPILib - it is precisely the limiting case of trapezoidal motion when the acceleration constraint is allowed to tend to infinity. Accordingly, the slew rate limiter is a good choice for applying a de-facto motion profile to a stream of velocity setpoints (or voltages, which are usually approximately proportional to velocity). For input streams that control positions, it is usually better to use a proper trapezoidal profile.

Slew rate limiting is supported in WPILib through the `SlewRateLimiter` class ([Java](#), [C++](#)).

Creating a `SlewRateLimiter`

Note: The C++ `SlewRateLimiter` class is templated on the unit type of the input. For more information on C++ units, see *The C++ Units Library*.

Note: Because filters have “memory”, each input stream requires its own filter object. Do *not* attempt to use the same filter object for multiple input streams.

Creating a `SlewRateLimiter` is simple:

Java

C++

```
// Creates a SlewRateLimiter that limits the rate of change of the signal to 0.5_
↳units per second
SlewRateLimiter filter = new SlewRateLimiter(0.5);
```

```
// Creates a SlewRateLimiter that limits the rate of change of the signal to 0.5_
↳volts per second
frc::SlewRateLimiter<units::volts> filter{0.5_v / 1_s};
```

Using a `SlewRateLimiter`

Once your filter has been created, using it is easy - simply call the `calculate()` method with the most recent input to obtain the filtered output:

Java

C++

```
// Calculates the next value of the output
filter.calculate(input);
```

```
// Calculates the next value of the output
filter.Calculate(input);
```

35.4 Geometry Classes

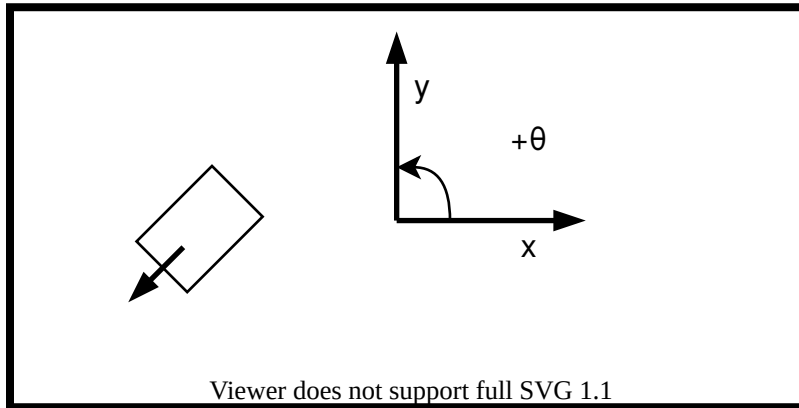
This section covers the geometry classes of WPILib.

35.4.1 Coordinate Systems

In FRC®, there are two main coordinate systems that we use for representing objects' positions.

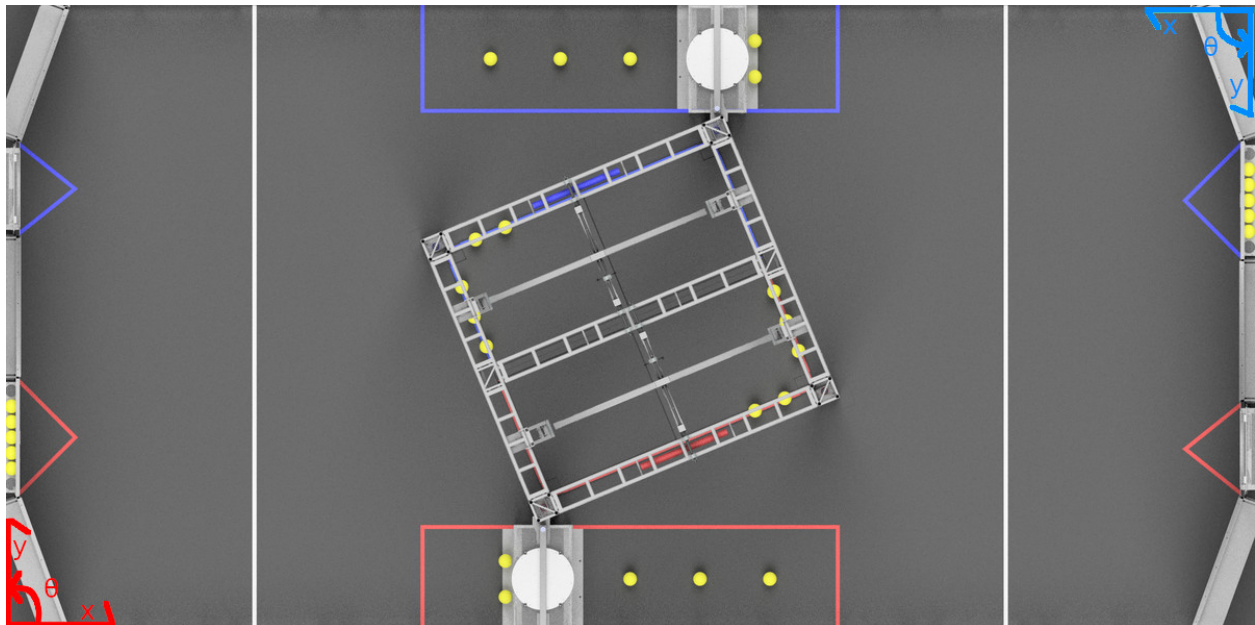
Field Coordinate System

The field coordinate system (or global coordinate system) is an absolute coordinate system where a point on the field is designated as the origin. Positive θ (theta) is in the counter-clockwise direction, and the positive x-axis points away from your alliance's driver station wall, and the positive y-axis is perpendicular and to the left of the positive x-axis.



Note: The axes are shown at the middle of the field for visibility. The origins of the coordinate system for each alliance are shown below.

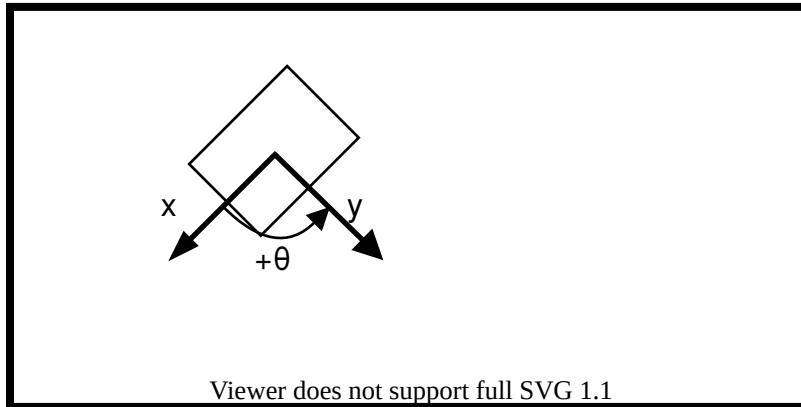
Below is an example of a field coordinate system overlaid on the 2020 FRC field. The red axes shown are for the red alliance, and the blue axes shown are for the blue alliance.



Robot Coordinate System

The robot coordinate system (or local coordinate system) is a relative coordinate system where the robot is the origin. The direction the robot is facing is the positive x axis, and the positive y axis is perpendicular, to the left of the robot. Positive θ is counter-clockwise.

Note: WPILib's Gyro class is clockwise-positive, so you have to invert the reading in order to get the rotation with either coordinate system.



35.4.2 Translation, Rotation, and Pose

Translation

Translation in 2 dimensions is represented by WPILib's Translation2d class ([Java](#), [C++](#)). This class has an x and y component, representing the point (x, y) or the vector $\begin{bmatrix} x \\ y \end{bmatrix}$ on a 2-dimensional coordinate system.

You can get the distance to another Translation2d object by using the `getDistance(Translation2d other)`, which returns the distance to another Translation2d by using the Pythagorean theorem.

Note: Translation2d uses the C++ Units library. If you're planning on using other WPILib classes that use Translation2d in Java, such as the trajectory generator, make sure to use meters.

Rotation

Rotation in 2 dimensions is represented by WPILib's `Rotation2d` class (Java, C++). This class has an angle component, which represents the robot's rotation relative to an axis on a 2-dimensional coordinate system. Positive rotations are counterclockwise.

Note: `Rotation2d` uses the C++ Units library. The constructor in Java accepts either the angle in radians, or the sine and cosine of the angle, but the `fromDegrees` method will construct a `Rotation2d` object from degrees.

Pose

Pose is a combination of both translation and rotation and is represented by the `Pose2d` class (Java, C++). It can be used to describe the pose of your robot in the field coordinate system, or the pose of objects, such as vision targets, relative to your robot in the robot coordinate

system. `Pose2d` can also represent the vector $\begin{bmatrix} x \\ y \\ \theta \end{bmatrix}$.

35.4.3 Transformations

Translation2d

Operations on a `Translation2d` perform operations to the vector represented by the `Translation2d`.

- **Addition:** Addition between two `Translation2d` `a` and `b` can be performed using `plus` in Java, or the `+` operator in C++. Addition adds the two vectors.
- **Subtraction:** Subtraction between two `Translation2d` can be performed using `minus` in Java, or the binary `-` operator in C++. Subtraction subtracts the two vectors.
- **Multiplication:** Multiplication of a `Translation2d` and a scalar can be performed using `times` in Java, or the `*` operator in C++. This multiplies the vector by the scalar.
- **Division:** Division of a `Translation2d` and a scalar can be performed using `div` in Java, or the `/` operator in C++. This divides the vector by the scalar.
- **Rotation:** Rotation of a `Translation2d` by a counter-clockwise rotation θ about the origin can be performed by using `rotateBy`. This is equivalent to multiplying the vector by the matrix $\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$
- Additionally, you can rotate a `Translation2d` by 180 degrees by using `unaryMinus` in Java, or the unary `-` operator in C++.

Rotation2d

Transformations for Rotation2d are just arithmetic operations on the angle measure represented by the Rotation2d.

- `plus` (Java) or `+` (C++): Adds the rotation component of `other` to this Rotation2d's rotation component
- `minus` (Java) or binary `-` (C++): Subtracts the rotation component of `other` to this Rotation2d's rotation component
- `unaryMinus` (Java) or unary `-` (C++): Multiplies the rotation component by a scalar of -1.
- `times` (Java) or `*` (C++) : Multiplies the rotation component by a scalar.
- `div` (Java) or `/`: Divides the rotation component by a scalar.

Transform2d and Twist2d

WPIlib provides 2 classes, Transform2d (Java, C++), which represents a transformation to a pose, and Twist2d (Java, C++) which represents a movement along an arc. Transform2d and Twist2d all have x , y and θ components.

Transform2d represents a **relative** transformation. It has an translation and a rotation component. Transforming a Pose2d by a Transform2d rotates the translation component of the transform by the rotation of the pose, and then adds the rotated translation component and the rotation component to the pose. In other words, `Pose2d.plus(Transform2d)` returns

$$\begin{bmatrix} x_p \\ y_p \\ \theta_p \end{bmatrix} + \begin{bmatrix} \cos\theta_p & -\sin\theta_p & 0 \\ \sin\theta_p & \cos\theta_p & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix}$$

Twist2d represents a change in distance along an arc. Usually, this class is used to represent the movement of a drivetrain, where the x component is the forward distance driven, the y component is the distance driven to the side (left positive), and the θ component is the change in heading. The underlying math behind finding the pose exponential (new pose after moving the pose forward along the curvature of the twist) can be found [here](#) in chapter 10.

Note: For nonholonomic drivetrains, the y component of a Twist2d should always be 0.

Both classes can be used to estimate robot location. Twist2d is used in WPIlib's odometry classes to update the robot's pose based on movement, while Transform2d can be used to estimate the robot's global position from vision data.

35.5 Controllers

This section describes various WPIlib feedback and feedforward controller classes that are useful for controlling the motion of FRC® mechanisms, as well as motion-profiling classes that can automatically generate setpoints for use with these controllers.

35.5.1 PID Control in WPILib

Note: This article covers the in-code implementation of PID Control with WPILib's provided library classes. Documentation describing the involved concepts in more detail is forthcoming.

Note: For a guide on implementing PID control through the *command-based framework*, see *PID Control through PIDSubsystems and PIDCommands*.

WPILib supports PID control of mechanisms through the `PIDController` class (Java, C++). This class handles the feedback loop calculation for the user, as well as offering methods for returning the error, setting tolerances, and checking if the control loop has reached its setpoint within the specified tolerances.

Using the `PIDController` Class

Note: The `PIDController` class in the `frc` namespace is deprecated - C++ teams should use the one in the `frc2` namespace, instead. Likewise, Java teams should use the class in the `wpilibj.controller` package.

Constructing a `PIDController`

Note: While `PIDController` may be used asynchronously, it does *not* provide any thread safety features - ensuring threadsafe operation is left entirely to the user, and thus asynchronous usage is recommended only for advanced teams.

In order to use WPILib's PID control functionality, users must first construct a `PIDController` object with the desired gains:

Java

C++

```
// Creates a PIDController with gains kP, kI, and kD
PIDController pid = new PIDController(kP, kI, kD);
```

```
// Creates a PIDController with gains kP, kI, and kD
frc2::PIDController pid{kP, kI, kD};
```

An optional fourth parameter can be provided to the constructor, specifying the period at which the controller will be run. The `PIDController` object is intended primarily for synchronous use from the main robot loop, and so this value is defaulted to 20ms.

Using the Feedback Loop Output

Note: The `PIDController` assumes that the `calculate()` method is being called regularly at an interval consistent with the configured period. Failure to do this will result in unintended loop behavior.

Warning: Unlike the old `PIDController`, the new `PIDController` does not automatically control an output from its own thread - users are required to call `calculate()` and use the resulting output in their own code.

Using the constructed `PIDController` is simple: simply call the `calculate()` method from the robot's main loop (e.g. the robot's `autonomousPeriodic()` method):

Java

C++

```
// Calculates the output of the PID algorithm based on the sensor reading
// and sends it to a motor
motor.set(pid.calculate(encoder.getDistance(), setpoint));
```

```
// Calculates the output of the PID algorithm based on the sensor reading
// and sends it to a motor
motor.Set(pid.Calculate(encoder.GetDistance(), setpoint));
```

Checking Errors

Note: `getPositionError()` and `getVelocityError()` are named assuming that the loop is controlling a position - for a loop that is controlling a velocity, these return the velocity error and the acceleration error, respectively.

The current error of the measured process variable is returned by the `getPositionError()` function, while its derivative is returned by the `getVelocityError()` function:

Specifying and Checking Tolerances

Note: If only a position tolerance is specified, the velocity tolerance defaults to infinity.

Note: As above, “position” refers to the process variable measurement, and “velocity” to its derivative - thus, for a velocity loop, these are actually velocity and acceleration, respectively.

Occasionally, it is useful to know if a controller has tracked the setpoint to within a given tolerance - for example, to determine if a command should be ended, or (while following a motion profile) if motion is being impeded and needs to be re-planned.

To do this, we first must specify the tolerances with the `setTolerance()` method; then, we can check it with the `atSetpoint()` method.

Java

C++

```
// Sets the error tolerance to 5, and the error derivative tolerance to 10 per second
pid.setTolerance(5, 10);

// Returns true if the error is less than 5 units, and the
// error derivative is less than 10 units
pid.atSetpoint();
```

```
// Sets the error tolerance to 5, and the error derivative tolerance to 10 per second
pid.SetTolerance(5, 10);

// Returns true if the error is less than 5 units, and the
// error derivative is less than 10 units
pid.AtSetpoint();
```

Resetting the Controller

It is sometimes desirable to clear the internal state (most importantly, the integral accumulator) of a `PIDController`, as it may be no longer valid (e.g. when the `PIDController` has been disabled and then re-enabled). This can be accomplished by calling the `reset()` method.

Setting a Max Integrator Value

Note: Integrators introduce instability and hysteresis into feedback loop systems. It is strongly recommended that teams avoid using integral gain unless absolutely no other solution will do - very often, problems that can be solved with an integrator can be better solved through use of a more-accurate *feedforward*.

A typical problem encountered when using integral feedback is excessive “wind-up” causing the system to wildly overshoot the setpoint. This can be alleviated in a number of ways - the WPILib `PIDController` offers an integrator range limiter to help teams overcome this issue.

Enabling this setting with the `setIntegratorRange()` method will prevent the total output contribution from the integral gain from exceeding the user-specified bounds.

Java

C++

```
// The integral gain term will never add or subtract more than 0.5 from
// the total loop output
pid.setIntegratorRange(-0.5, 0.5);
```

```
// The integral gain term will never add or subtract more than 0.5 from
// the total loop output
pid.SetIntegratorRange(-0.5, 0.5);
```

Setting Continuous Input

Warning: If your mechanism is not capable of fully continuous rotational motion (e.g. a turret without a slip ring, whose wires twist as it rotates), *do not* enable continuous input unless you have implemented an additional safety feature to prevent the mechanism from moving past its limit!

Warning: The continuous input function does *not* automatically wrap your input values - be sure that your input values, when using this feature, are never outside of the specified range!

Some process variables (such as the angle of a turret) are measured on a circular scale, rather than a linear one - that is, each “end” of the process variable range corresponds to the same point in reality (e.g. 360 degrees and 0 degrees). In such a configuration, there are two possible values for any given error, corresponding to which way around the circle the error is measured. It is usually best to use the smaller of these errors.

To configure a `PIDController` to automatically do this, use the `enableContinuousInput()` method:

Java

C++

```
// Enables continuous input on a range from -180 to 180
pid.enableContinuousInput(-180, 180);
```

```
// Enables continuous input on a range from -180 to 180
pid.EnableContinuousInput(-180, 180);
```

Clamping Controller Output

Unlike the old `PIDController`, the new controller does not offer any output clamping features, as the user is expected to use the loop output themselves. Output clamping can be easily achieved by composing the controller with WPI’s `clamp()` function (or `std::clamp` in c++):

Java

C++

```
// Clamps the controller output to between -0.5 and 0.5
MathUtil.clamp(pid.calculate(encoder.getDistance(), setpoint), -0.5, 0.5);
```

```
// Clamps the controller output to between -0.5 and 0.5
std::clamp(pid.Calculate(encoder.GetDistance(), setpoint), -0.5, 0.5);
```


35.5.2 Feedforward Control in WPILib

So far, we've used feedback control for reference tracking (making a system's output follow a desired reference signal). While this is effective, it's a reactionary measure; the system won't start applying control effort until the system is already behind. If we could tell the controller about the desired movement and required input beforehand, the system could react quicker and the feedback controller could do less work. A controller that feeds information forward into the plant like this is called a feedforward controller.

A feedforward controller injects information about the system's dynamics (like a mathematical model does) or the intended movement. Feedforward handles parts of the control actions we already know must be applied to make a system track a reference, then feedback compensates for what we do not or cannot know about the system's behavior at runtime.

There are two types of feedforwards: model-based feedforward and feedforward for unmodeled dynamics. The first solves a mathematical model of the system for the inputs required to meet desired velocities and accelerations. The second compensates for unmodeled forces or behaviors directly so the feedback controller doesn't have to. Both types can facilitate simpler feedback controllers. We'll cover several examples below.

Note: The WPILib feedforward classes closely match the available mechanism characterization tools available in the [frc-characterization toolsuite](#) - the characterization toolsuite can be used to quickly and effectively determine the correct gains for each type of feedforward. The toolsuite will indicate the appropriate units for each of the gains.

WPILib provides a number of classes to help users implement accurate feedforward control for their mechanisms. In many ways, an accurate feedforward is more important than feedback to effective control of a mechanism. Since most FRC® mechanisms closely obey well-understood system equations, starting with an accurate feedforward is both easy and hugely beneficial to accurate and robust mechanism control.

WPILib currently provides the following three helper classes for feedforward control:

- [SimpleMotorFeedforward](#) (Java, C++)
- [ArmFeedforward](#) (Java, C++)
- [ElevatorFeedforward](#) (Java, C++)

SimpleMotorFeedforward

Note: In C++, the SimpleMotorFeedforward class is templated on the unit type used for distance measurements, which may be angular or linear. The passed-in gains *must* have units consistent with the distance units, or a compile-time error will be thrown. kS should have units of volts, kV should have units of volts * seconds / distance, and kA should have units of volts * seconds^2 / distance. For more information on C++ units, see [The C++ Units Library](#).

Note: The Java feedforward components will calculate outputs in units determined by the units of the user-provided feedforward gains. Users *must* take care to keep units consistent, as WPILibJ does not have a type-safe unit system.

The SimpleMotorFeedforward class calculates feedforwards for mechanisms that consist of permanent-magnet DC motors with no external loading other than friction and inertia, such as flywheels and robot drives.

To create a SimpleMotorFeedforward, simply construct it with the required gains:

Note: The kA gain can be omitted, and if it is, will default to a value of zero. For many mechanisms, especially those with little inertia, it is not necessary.

Java

C++

```
// Create a new SimpleMotorFeedforward with gains kS, kV, and kA
SimpleMotorFeedforward feedforward = new SimpleMotorFeedforward(kS, kV, kA);
```

```
// Create a new SimpleMotorFeedforward with gains kS, kV, and kA
// Distance is measured in meters
frc::SimpleMotorFeedforward<units::meters> feedforward(kS, kV, kA);
```

To calculate the feedforward, simply call the calculate() method with the desired motor velocity and acceleration:

Note: The acceleration argument may be omitted from the calculate() call, and if it is, will default to a value of zero. This should be done whenever there is not a clearly-defined acceleration setpoint.

Java

C++

```
// Calculates the feedforward for a velocity of 10 units/second and an acceleration
// of 20 units/second^2
// Units are determined by the units of the gains passed in at construction.
feedforward.calculate(10, 20);
```

```
// Calculates the feedforward for a velocity of 10 meters/second and an acceleration
// of 20 meters/second^2
// Output is in volts
feedforward.Calculate(10_mps, 20_mps_sq);
```

ArmFeedforward

Note: In C++, the ArmFeedforward class assumes distances are angular, not linear. The passed-in gains *must* have units consistent with the angular unit, or a compile-time error will be thrown. kS and kCos should have units of volts, kV should have units of volts * seconds / radians, and kA should have units of volts * seconds^2 / radians. For more information on C++ units, see [The C++ Units Library](#).

Note: The Java feedforward components will calculate outputs in units determined by the

units of the user-provided feedforward gains. Users *must* take care to keep units consistent, as WPILibJ does not have a type-safe unit system.

The ArmFeedforward class calculates feedforwards for arms that are controlled directly by a permanent-magnet DC motor, with external loading of friction, inertia, and mass of the arm. This is an accurate model of most arms in FRC.

To create an ArmFeedforward, simply construct it with the required gains:

Note: The kA gain can be omitted, and if it is, will default to a value of zero. For many mechanisms, especially those with little inertia, it is not necessary.

Java

C++

```
// Create a new ArmFeedforward with gains kS, kCos, kV, and kA
ArmFeedforward feedforward = new ArmFeedforward(kS, kCos, kV, kA);
```

```
// Create a new ArmFeedforward with gains kS, kCos, kV, and kA
frc::ArmFeedforward feedforward(kS, kCos, kV, kA);
```

To calculate the feedforward, simply call the calculate() method with the desired arm position, velocity, and acceleration:

Note: The acceleration argument may be omitted from the calculate() call, and if it is, will default to a value of zero. This should be done whenever there is not a clearly-defined acceleration setpoint.

Java

C++

```
// Calculates the feedforward for a position of 1 units, a velocity of 2 units/second,
↪ and
// an acceleration of 3 units/second^2
// Units are determined by the units of the gains passed in at construction.
feedforward.calculate(1, 2, 3);
```

```
// Calculates the feedforward for a position of 1 radians, a velocity of 2 radians/
↪ second, and
// an acceleration of 3 radians/second^2
// Output is in volts
feedforward.Calculate(1_rad, 2_rad_per_s, 3_rad/(1_s * 1_s));
```

ElevatorFeedforward

Note: In C++, the ElevatorFeedforward class is templated on the unit type used for distance measurements, which may be angular or linear. The passed-in gains *must* have units consistent with the distance units, or a compile-time error will be thrown. kS and kG should have units of volts, kV should have units of volts * seconds / distance, and kA should have units of volts * seconds² / distance. For more information on C++ units, see [The C++ Units Library](#).

Note: The Java feedforward components will calculate outputs in units determined by the units of the user-provided feedforward gains. Users *must* take care to keep units consistent, as WPILibJ does not have a type-safe unit system.

The ElevatorFeedforward class calculates feedforwards for elevators that consist of permanent-magnet DC motors loaded by friction, inertia, and the mass of the elevator. This is an accurate model of most elevators in FRC.

To create a ElevatorFeedforward, simply construct it with the required gains:

Note: The kA gain can be omitted, and if it is, will default to a value of zero. For many mechanisms, especially those with little inertia, it is not necessary.

Java

C++

```
// Create a new ElevatorFeedforward with gains kS, kG, kV, and kA
ElevatorFeedforward feedforward = new ElevatorFeedforward(kS, kG, kV, kA);
```

```
// Create a new ElevatorFeedforward with gains kS, kV, and kA
// Distance is measured in meters
frc::ElevatorFeedforward<units::meters> feedforward(kS, kG, kV, kA);
```

To calculate the feedforward, simply call the calculate() method with the desired motor velocity and acceleration:

Note: The acceleration argument may be omitted from the calculate() call, and if it is, will default to a value of zero. This should be done whenever there is not a clearly-defined acceleration setpoint.

Java

C++

```
// Calculates the feedforward for a position of 10 units, velocity of 20 units/second,
// and an acceleration of 30 units/second^2
// Units are determined by the units of the gains passed in at construction.
feedforward.calculate(10, 20, 30);
```

```
// Calculates the feedforward for a position of 10 meters, velocity of 20 meters/
↪second,
// and an acceleration of 30 meters/second^2
// Output is in volts
feedforward.Calculate(10_m, 20_mps, 30_mps_sq);
```

Using Feedforward to Control Mechanisms

Note: Since feedforward voltages are physically meaningful, it is best to use the `setVoltage()` (Java, C++) method when applying them to motors to compensate for “voltage sag” from the battery.

Feedforward control can be used entirely on its own, without a feedback controller. This is known as “open-loop” control, and for many mechanisms (especially robot drives) can be perfectly satisfactory. A `SimpleMotorFeedforward` might be employed to control a robot drive as follows:

Java

C++

```
public void tankDriveWithFeedforward(double leftVelocity, double rightVelocity) {
    leftMotor.setVoltage(feedforward.calculate(leftVelocity));
    rightMotor.setVoltage(feedforward.calculate(rightVelocity));
}
```

```
void TankDriveWithFeedforward(units::meters_per_second_t leftVelocity,
                               units::meters_per_second_t rightVelocity) {
    leftMotor.SetVoltage(feedforward.Calculate(leftVelocity));
    rightMotor.SetVoltage(feedforward.Calculate(rightVelocity));
}
```

35.5.3 Combining Feedforward and PID Control

Note: This article covers the in-code implementation of combined feedforward/PID control with WPILib’s provided library classes. Documentation describing the involved concepts in more detail is forthcoming.

Feedforward and feedback controllers can each be used in isolation, but are most effective when combined together. Thankfully, combining these two control methods is *exceedingly* straightforward - one simply adds their outputs together.

Using Feedforward with a PIDController

Users familiar with the old `PIDController` class may notice the lack of any feedforward gain in the new controller. As users are expected to use the controller output themselves, there is no longer any need for the `PIDController` to implement feedforward - users may simply add any feedforward they like to the output of the controller before sending it to their motors:

Java

C++

```
// Adds a feedforward to the loop output before sending it to the motor
motor.setVoltage(pid.calculate(encoder.getDistance(), setpoint) + feedforward);
```

```
// Adds a feedforward to the loop output before sending it to the motor
motor.SetVoltage(pid.Calculate(encoder.GetDistance(), setpoint) + feedforward);
```

Moreover, feedforward is a separate feature entirely from feedback, and thus has no reason to be handled in the same controller object, as this violates separation of concerns. WPILib comes with several helper classes to compute accurate feedforward voltages for common FRC® mechanisms - for more information, see [Feedforward Control in WPILib](#).

Using Feedforward Components with PID

Note: Since feedforward voltages are physically meaningful, it is best to use the `setVoltage()` (Java, C++) method when applying them to motors to compensate for “voltage sag” from the battery.

What might a more complete example of combined feedforward/PID control look like? Consider the [drive example](#) from the feedforward page. We can easily modify this to include feedback control (with a `SimpleMotorFeedforward` component):

Java

C++

```
public void tankDriveWithFeedforwardPID(double leftVelocitySetpoint, double_
↪rightVelocitySetpoint) {
    leftMotor.setVoltage(feedforward.calculate(leftVelocitySetpoint)
        + leftPID.calculate(leftEncoder.getRate(), leftVelocitySetpoint);
    rightMotor.setVoltage(feedforward.calculate(rightVelocitySetpoint)
        + rightPID.calculate(rightEncoder.getRate(), rightVelocitySetpoint);
}
```

```
void TankDriveWithFeedforwardPID(units::meters_per_second_t leftVelocitySetpoint,
                                units::meters_per_second_t rightVelocitySetpoint) {
    leftMotor.SetVoltage(feedforward.Calculate(leftVelocitySetpoint)
        + leftPID.Calculate(leftEncoder.getRate(), leftVelocitySetpoint.to<double>());
    rightMotor.SetVoltage(feedforward.Calculate(rightVelocitySetpoint)
        + rightPID.Calculate(rightEncoder.getRate(), rightVelocitySetpoint.to<double>
↪());
}
```

Other mechanism types can be handled similarly.

35.5.4 Trapezoidal Motion Profiles in WPILib

Note: This article covers the in-code generation of trapezoidal motion profiles. Documentation describing the involved concepts in more detail is forthcoming.

Note: For a guide on implementing the `TrapezoidProfile` class in the *command-based framework* framework, see *Motion Profiling through TrapezoidProfileSubsystems and TrapezoidProfileCommands*.

Note: The `TrapezoidProfile` class, used on its own, is most useful when composed with a custom controller (such as a “smart” motor controller with a built-in PID functionality). To integrate it with a WPILib `PIDController`, see *Combining Motion Profiling and PID Control with ProfiledPIDController*.

While feedforward and feedback control offer convenient ways to achieve a given setpoint, we are often still faced with the problem of generating setpoints for our mechanisms. While the naive approach of immediately commanding a mechanism to its desired state may work, it is often suboptimal. To improve the handling of our mechanisms, we often wish to command mechanisms to a *sequence* of setpoints that smoothly interpolate between its current state, and its desired goal state.

To help users do this, WPILib provides a `TrapezoidProfile` class ([Java](#), [C++](#)).

Creating a TrapezoidProfile

Note: In C++, the `TrapezoidProfile` class is templated on the unit type used for distance measurements, which may be angular or linear. The passed-in values *must* have units consistent with the distance units, or a compile-time error will be thrown. For more information on C++ units, see *The C++ Units Library*.

Constraints

Note: The various *feedforward helper classes* provide methods for calculating the maximum simultaneously-achievable velocity and acceleration of a mechanism. These can be very useful for calculating appropriate motion constraints for your `TrapezoidProfile`.

In order to create a trapezoidal motion profile, we must first impose some constraints on the desired motion. Namely, we must specify a maximum velocity and acceleration that the mechanism will be expected to achieve during the motion. To do this, we create an instance of the `TrapezoidProfile.Constraints` class ([Java](#), [C++](#)):

Java

C++

```
// Creates a new set of trapezoidal motion profile constraints
// Max velocity of 10 meters per second
// Max acceleration of 20 meters per second squared
new TrapezoidProfile.Constraints(10, 20);
```

```
// Creates a new set of trapezoidal motion profile constraints
// Max velocity of 10 meters per second
// Max acceleration of 20 meters per second squared
frc::TrapezoidProfile<units::meters>::Constraints{10_mps, 20_mps_sq};
```

Start and End States

Next, we must specify the desired starting and ending states for our mechanisms using the `TrapezoidProfile.State` class (Java, C++). Each state has a position and a velocity:

Java

C++

```
// Creates a new state with a position of 5 meters
// and a velocity of 0 meters per second
new TrapezoidProfile.State(5, 0);
```

```
// Creates a new state with a position of 5 meters
// and a velocity of 0 meters per second
frc::TrapezoidProfile<units::meters>::State{5_m, 0_mps};
```

Putting It All Together

Note: C++ is often able to infer the type of the inner classes, and thus a simple initializer list (without the class name) can be sent as a parameter. The full class names are included in the example below for clarity.

Now that we know how to create a set of constraints and the desired start/end states, we are ready to create our motion profile. The `TrapezoidProfile` constructor takes 3 parameters, in order: the constraints, the goal state, and the initial state.

Java

C++

```
// Creates a new TrapezoidProfile
// Profile will have a max vel of 5 meters per second
// Profile will have a max acceleration of 10 meters per second squared
// Profile will end stationary at 5 meters
// Profile will start stationary at zero position
TrapezoidProfile profile = new TrapezoidProfile(new TrapezoidProfile.Constraints(5,
↪10),
                                         new TrapezoidProfile.State(5, 0),
                                         new TrapezoidProfile.State(0, 0));
```



```
// Creates a new TrapezoidProfile
// Profile will have a max vel of 5 meters per second
// Profile will have a max acceleration of 10 meters per second squared
// Profile will end stationary at 5 meters
// Profile will start stationary at zero position
frc::TrapezoidProfile<units::meters> profile(
    frc::TrapezoidProfile<units::meters>::Constraints{5_mps, 10_mps_sq},
    frc::TrapezoidProfile<units::meters>::State{5_m, 0_mps},
    frc::TrapezoidProfile<units::meters>::State{0_m, 0_mps});
```

Using a TrapezoidProfile

Sampling the Profile

Once we've created a TrapezoidProfile, using it is very simple: to get the profile state at the given time after the profile has started, call the `calculate()` method:

Java

C++

```
// Returns the motion profile state after 5 seconds of motion
profile.calculate(5);
```

```
// Returns the motion profile state after 5 seconds of motion
profile.Calculate(5_s);
```

Using the State

The `calculate` method returns a `TrapezoidProfile.State` class (the same one that was used to specify the initial/end states when constructing the profile). To use this for actual control, simply pass the contained position and velocity values to whatever controller you wish (for example, a `PIDController`):

Java

C++

```
var setpoint = profile.calculate(elapsedTime);
controller.calculate(encoder.getDistance(), setpoint.position);
```

```
auto setpoint = profile.Calculate(elapsedTime);
controller.Calculate(encoder.GetDistance(), setpoint.position.to<double>());
```

Complete Usage Example

Note: In this example, the profile is re-computed every timestep. This is a somewhat different usage technique than is detailed above, but works according to the same principles - the profile is sampled at a time corresponding to the loop period to get the setpoint for the next loop iteration.

A more complete example of TrapezoidProfile usage is provided in the ElevatorTrapezoidProfile example project (Java, C++):

Java

C++

```

8 package edu.wpi.first.wpilibj.examples.elevatortrapezoidprofile;
9
10 import edu.wpi.first.wpilibj.Joystick;
11 import edu.wpi.first.wpilibj.TimedRobot;
12 import edu.wpi.first.wpilibj.controller.SimpleMotorFeedforward;
13 import edu.wpi.first.wpilibj.trajectory.TrapezoidProfile;
14
15 public class Robot extends TimedRobot {
16     private static double kDt = 0.02;
17
18     private final Joystick m_joystick = new Joystick(1);
19     private final ExampleSmartMotorController m_motor = new
20 ↪ ExampleSmartMotorController(1);
21     // Note: These gains are fake, and will have to be tuned for your robot.
22     private final SimpleMotorFeedforward m_feedforward = new SimpleMotorFeedforward(1,
23 ↪ 1.5);
24
25     private final TrapezoidProfile.Constraints m_constraints =
26     new TrapezoidProfile.Constraints(1.75, 0.75);
27     private TrapezoidProfile.State m_goal = new TrapezoidProfile.State();
28     private TrapezoidProfile.State m_setpoint = new TrapezoidProfile.State();
29
30     @Override
31     public void robotInit() {
32         // Note: These gains are fake, and will have to be tuned for your robot.
33         m_motor.setPID(1.3, 0.0, 0.7);
34     }
35
36     @Override
37     public void teleopPeriodic() {
38         if (m_joystick.getRawButtonPressed(2)) {
39             m_goal = new TrapezoidProfile.State(5, 0);
40         } else if (m_joystick.getRawButtonPressed(3)) {
41             m_goal = new TrapezoidProfile.State(0, 0);
42         }
43
44         // Create a motion profile with the given maximum velocity and maximum
45         // acceleration constraints for the next setpoint, the desired goal, and the
46         // current setpoint.
47         var profile = new TrapezoidProfile(m_constraints, m_goal, m_setpoint);
48
49         // Retrieve the profiled setpoint for the next timestep. This setpoint moves

```

(continues on next page)

(continued from previous page)

```

48 // toward the goal while obeying the constraints.
49 m_setpoint = profile.calculate(kDt);
50
51 // Send setpoint to offboard controller PID
52 m_motor.setSetpoint(ExampleSmartMotorController.PIDMode.kPosition, m_setpoint.
↪position,
53                      m_feedforward.calculate(m_setpoint.velocity) / 12.0);
54 }
55 }

```

```

8 #include <frc/Joystick.h>
9 #include <frc/TimedRobot.h>
10 #include <frc/controller/SimpleMotorFeedforward.h>
11 #include <frc/trajectory/TrapezoidProfile.h>
12 #include <units/acceleration.h>
13 #include <units/length.h>
14 #include <units/time.h>
15 #include <units/velocity.h>
16 #include <units/voltage.h>
17 #include <wpi/math>
18
19 #include "ExampleSmartMotorController.h"
20
21 class Robot : public frc::TimedRobot {
22 public:
23     static constexpr units::second_t kDt = 20_ms;
24
25     Robot() {
26         // Note: These gains are fake, and will have to be tuned for your robot.
27         m_motor.SetPID(1.3, 0.0, 0.7);
28     }
29
30     void TeleopPeriodic() override {
31         if (m_joystick.GetRawButtonPressed(2)) {
32             m_goal = {5_m, 0_mps};
33         } else if (m_joystick.GetRawButtonPressed(3)) {
34             m_goal = {0_m, 0_mps};
35         }
36
37         // Create a motion profile with the given maximum velocity and maximum
38         // acceleration constraints for the next setpoint, the desired goal, and the
39         // current setpoint.
40         frc::TrapezoidProfile<units::meters> profile{m_constraints, m_goal,
41                                                     m_setpoint};
42
43         // Retrieve the profiled setpoint for the next timestep. This setpoint moves
44         // toward the goal while obeying the constraints.
45         m_setpoint = profile.Calculate(kDt);
46
47         // Send setpoint to offboard controller PID
48         m_motor.SetSetpoint(ExampleSmartMotorController::PIDMode::kPosition,
49                             m_setpoint.position.to<double>(),
50                             m_feedforward.Calculate(m_setpoint.velocity) / 12_V);
51     }
52
53 private:

```

(continues on next page)

(continued from previous page)

```

54 frc::Joystick m_joystick{1};
55 ExampleSmartMotorController m_motor{1};
56 frc::SimpleMotorFeedforward<units::meters> m_feedforward{
57     // Note: These gains are fake, and will have to be tuned for your robot.
58     1_V, 1.5_V * 1_s / 1_m};
59
60 frc::TrapezoidProfile<units::meters>::Constraints m_constraints{1.75_mps,
61                                                                0.75_mps_sq};
62 frc::TrapezoidProfile<units::meters>::State m_goal;
63 frc::TrapezoidProfile<units::meters>::State m_setpoint;
64 };
65
66 #ifndef RUNNING_FRC_TESTS
67 int main() { return frc::StartRobot<Robot>(); }
68 #endif

```

35.5.5 Combining Motion Profiling and PID Control with ProfiledPID-Controller

Note: For a guide on implementing the ProfiledPIDController class in the *command-based framework* framework, see *Combining Motion Profiling and PID in Command-Based*.

In the previous article, we saw how to use the TrapezoidProfile class to create and use a trapezoidal motion profile. The example code from that article demonstrates manually composing the TrapezoidProfile class with the external PID control feature of a “smart” motor controller.

This combination of functionality (a motion profile for generating setpoints combined with a PID controller for following them) is extremely common. To facilitate this, WPILib comes with a ProfiledPIDController class (Java, C++) that does most of the work of combining these two functionalities. The API of the ProfiledPIDController is very similar to that of the PIDController, allowing users to add motion profiling to a PID-controlled mechanism with very few changes to their code.

Using the ProfiledPIDController class

Note: In C++, the ProfiledPIDController class is templated on the unit type used for distance measurements, which may be angular or linear. The passed-in values *must* have units consistent with the distance units, or a compile-time error will be thrown. For more information on C++ units, see *The C++ Units Library*.

Note: Much of the functionality of ProfiledPIDController is effectively identical to that of PIDController. Accordingly, this article will only cover features that are substantially-changed to accommodate the motion profiling functionality. For information on standard PIDController features, see *PID Control in WPILib*.

Constructing a ProfiledPIDController

Note: C++ is often able to infer the type of the inner classes, and thus a simple initializer list (without the class name) can be sent as a parameter. The full class name is included in the example below for clarity.

Creating a ProfiledPIDController is nearly identical to *creating a PIDController*. The only difference is the need to supply a set of *trapezoidal profile constraints*, which will be automatically forwarded to the internally-generated TrapezoidProfile instances:

Java

C++

```
// Creates a ProfiledPIDController
// Max velocity is 5 meters per second
// Max acceleration is 10 meters per second
ProfiledPIDController controller = new ProfiledPIDController(
    kP, kI, kD,
    new TrapezoidProfile.Constraints(5, 10));
```

```
// Creates a ProfiledPIDController
// Max velocity is 5 meters per second
// Max acceleration is 10 meters per second
frc::ProfiledPIDController<units::meters> controller(
    kP, kI, kD,
    frc::TrapezoidProfile<units::meters>::Constraints{5_mps, 10_mps_sq});
```

Goal vs Setpoint

A major difference between a standard PIDController and a ProfiledPIDController is that the actual *setpoint* of the control loop is not directly specified by the user. Rather, the user specifies a *goal* position or state, and the setpoint for the controller is computed automatically from the generated motion profile between the current state and the goal. So, while the user-side call looks mostly identical:

Java

C++

```
// Calculates the output of the PID algorithm based on the sensor reading
// and sends it to a motor
motor.set(controller.calculate(encoder.getDistance(), goal));
```

```
// Calculates the output of the PID algorithm based on the sensor reading
// and sends it to a motor
motor.Set(controller.Calculate(encoder.GetDistance(), goal));
```

The specified goal value (which can be either a position value or a TrapezoidProfile.State, if nonzero velocity is desired) is *not* necessarily the *current* setpoint of the loop - rather, it is the *eventual* setpoint once the generated profile terminates.

Getting/Using the Setpoint

Since the `ProfiledPIDController` goal differs from the setpoint, it is often desirable to poll the current setpoint of the controller (for instance, to get values to use with *feedforward*). This can be done with the `getSetpoint()` method.

The returned setpoint might then be used as in the following example:

Java

C++

```
double lastSpeed = 0;
double lastTime = Timer.getFPGATimestamp();

// Controls a simple motor's position using a SimpleMotorFeedforward
// and a ProfiledPIDController
public void goToPosition(double goalPosition) {
    double acceleration = (controller.getSetpoint().velocity - lastSpeed) / (Timer.
    ↪getFPGATimestamp() - lastTime);
    motor.setVoltage(
        controller.calculate(encoder.getDistance(), goalPosition)
        + feedforward.calculate(controller.getSetpoint().velocity, acceleration));
    lastSpeed = controller.getSetpoint().velocity;
    lastTime = Timer.getFPGATimestamp();
}
```

```
units::meters_per_second_t lastSpeed = 0_mps;
units::second_t lastTime = frc2::Timer::GetFPGATimestamp();

// Controls a simple motor's position using a SimpleMotorFeedforward
// and a ProfiledPIDController
void GoToPosition(units::meter_t goalPosition) {
    auto acceleration = (controller.GetSetpoint().velocity - lastSpeed) /
        (frc2::Timer::GetFPGATimestamp() - lastTime);
    motor.SetVoltage(
        controller.Calculate(units::meter_t{encoder.GetDistance()}, goalPosition) +
        feedforward.Calculate(controller.GetSetpoint().velocity, acceleration));
    lastSpeed = controller.GetSetpoint().velocity;
    lastTime = frc2::Timer::GetFPGATimestamp();
}
```

Complete Usage Example

A more complete example of `ProfiledPIDController` usage is provided in the `ElevatorProfilePID` example project (Java, C++):

Java

C++

```
8 package edu.wpi.first.wpilibj.examples.elevatorprofiledpid;
9
10 import edu.wpi.first.wpilibj.Encoder;
11 import edu.wpi.first.wpilibj.Joystick;
12 import edu.wpi.first.wpilibj.PWMVictorSPX;
13 import edu.wpi.first.wpilibj.SpeedController;
```

(continues on next page)

(continued from previous page)

```

14 import edu.wpi.first.wpilibj.TimedRobot;
15 import edu.wpi.first.wpilibj.controller.ProfiledPIDController;
16 import edu.wpi.first.wpilibj.trajectory.TrapezoidProfile;
17
18 public class Robot extends TimedRobot {
19     private static double kDt = 0.02;
20
21     private final Joystick m_joystick = new Joystick(1);
22     private final Encoder m_encoder = new Encoder(1, 2);
23     private final SpeedController m_motor = new PWMVictorSPX(1);
24
25     // Create a PID controller whose setpoint's change is subject to maximum
26     // velocity and acceleration constraints.
27     private final TrapezoidProfile.Constraints m_constraints =
28         new TrapezoidProfile.Constraints(1.75, 0.75);
29     private final ProfiledPIDController m_controller =
30         new ProfiledPIDController(1.3, 0.0, 0.7, m_constraints, kDt);
31
32     @Override
33     public void robotInit() {
34         m_encoder.setDistancePerPulse(1.0 / 360.0 * 2.0 * Math.PI * 1.5);
35     }
36
37     @Override
38     public void teleopPeriodic() {
39         if (m_joystick.getRawButtonPressed(2)) {
40             m_controller.setGoal(5);
41         } else if (m_joystick.getRawButtonPressed(3)) {
42             m_controller.setGoal(0);
43         }
44
45         // Run controller and update motor output
46         m_motor.set(m_controller.calculate(m_encoder.getDistance()));
47     }
48 }

```

```

8  #include <frc/Encoder.h>
9  #include <frc/Joystick.h>
10 #include <frc/PWMVictorSPX.h>
11 #include <frc/TimedRobot.h>
12 #include <frc/controller/ProfiledPIDController.h>
13 #include <frc/trajectory/TrapezoidProfile.h>
14 #include <units/acceleration.h>
15 #include <units/length.h>
16 #include <units/time.h>
17 #include <units/velocity.h>
18 #include <wpi/math>
19
20 class Robot : public frc::TimedRobot {
21 public:
22     static constexpr units::second_t kDt = 20_ms;
23
24     Robot() {
25         m_encoder.SetDistancePerPulse(1.0 / 360.0 * 2.0 * wpi::math::pi * 1.5);
26     }
27

```

(continues on next page)

(continued from previous page)

```

28 void TeleopPeriodic() override {
29     if (m_joystick.GetRawButtonPressed(2)) {
30         m_controller.SetGoal(5_m);
31     } else if (m_joystick.GetRawButtonPressed(3)) {
32         m_controller.SetGoal(0_m);
33     }
34
35     // Run controller and update motor output
36     m_motor.Set(
37         m_controller.Calculate(units::meter_t(m_encoder.GetDistance())));
38 }
39
40 private:
41     frc::Joystick m_joystick{1};
42     frc::Encoder m_encoder{1, 2};
43     frc::PWMVictorSPX m_motor{1};
44
45     // Create a PID controller whose setpoint's change is subject to maximum
46     // velocity and acceleration constraints.
47     frc::TrapezoidProfile<units::meters>::Constraints m_constraints{1.75_mps,
48                                                                    0.75_mps_sq};
49     frc::ProfiledPIDController<units::meters> m_controller{1.3, 0.0, 0.7,
50                                                            m_constraints, kDt};
51 };
52
53 #ifndef RUNNING_FRC_TESTS
54 int main() { return frc::StartRobot<Robot>(); }
55 #endif

```

35.6 Trajectory Generation and Following with WPILib

This section describes WPILib support for generating parameterized spline trajectories and following those trajectories with typical FRC® robot drives.

35.6.1 Trajectory Generation

The 2020 release of WPILib contains classes that help generating trajectories. A trajectory is a smooth curve, with velocities and accelerations at each point along the curve, connecting two endpoints on the field. Generation and following of trajectories is incredibly useful for performing autonomous tasks. Instead of a simple autonomous routine – which involves moving forward, stopping, turning 90 degrees to the right, then moving forward – using trajectories allows for motion along a smooth curve. This has the advantage of speeding up autonomous routines, creating more time for other tasks; and when implemented well, makes autonomous navigation more accurate and precise.

This article goes over how to generate a trajectory. The next few articles in this series will go over how to actually follow the generated trajectory. There are a few things that your robot must have before you dive into the world of trajectories:

- A way to measure the position and velocity of each side of the robot. An encoder is the best way to do this; however, other options may include optical flow sensors, etc.

- A way to measure the angle or angular rate of the robot chassis. A gyroscope is the best way to do this. Although the angular rate can be calculated using encoder velocities, this method is NOT recommended because of wheel scrubbing.

If you are looking for a simpler way to perform autonomous navigation, see [the section on driving to a distance](#).

Splines

A spline refers to a set of curves that interpolate between points. Think of it as connecting dots, except with curves. WPILib supports two types of splines: hermite clamped cubic and hermite quintic.

- Hermite clamped cubic: This is the recommended option for most users. Generation of trajectories using these splines involves specifying the (x, y) coordinates of all points, and the headings at the start and end waypoints. The headings at the interior waypoints are automatically determined to ensure continuous curvature (rate of change of the heading) throughout.
- Hermite quintic: This is a more advanced option which requires the user to specify (x, y) coordinates and headings for *all* waypoints. This should be used if you are unhappy with the trajectories that are being generated by the clamped cubic splines or if you want finer control of headings at the interior points.

Splines are used as a tool to generate trajectories; however, the spline itself does not have any information about velocities and accelerations. Therefore, it is not recommended that you use the spline classes directly. In order to generate a smooth path with velocities and accelerations, a *trajectory* must be generated.

Creating the trajectory config

A configuration must be created in order to generate a trajectory. The config contains information about special constraints, the max velocity, the max acceleration in addition to the start velocity and end velocity. The config also contains information about whether the trajectory should be reversed (robot travels backward along the waypoints). The `TrajectoryConfig` class should be used to construct a config. The constructor for this class takes two arguments, the max velocity and max acceleration. The other fields (`startVelocity`, `endVelocity`, `reversed`, `constraints`) are defaulted to reasonable values (0, 0, false, {}) when the object is created. If you wish to modify the values of any of these fields, you can call the following methods:

- `setStartVelocity(double startVelocityMetersPerSecond)` (Java) / `SetStartVelocity(units::meters_per_second_t startVelocity)` (C++)
- `setEndVelocity(double endVelocityMetersPerSecond)` (Java) / `SetEndVelocity(units::meters_per_second_t endVelocity)` (C++)
- `setReversed(boolean reversed)` (Java) / `SetReversed(bool reversed)` (C++)
- `addConstraint(TrajectoryConstraint constraint)` (Java) / `AddConstraint(TrajectoryConstraint constraint)` (C++)

Note: The `reversed` property simply represents whether the robot is traveling backward. If you specify four waypoints, a, b, c, and d, the robot will still travel in the same order through the waypoints when the `reversed` flag is set to true. This also means that you must account

for the direction of the robot when providing the waypoints. For example, if your robot is facing your alliance station wall and travels backwards to some field element, the starting waypoint should have a rotation of 180 degrees.

Generating the trajectory

The method used to generate a trajectory is `generateTrajectory(...)`. There are four overloads for this method. Two that use clamped cubic splines and the two others that use quintic splines. For each type of spline, there are two ways to construct a trajectory. The easiest methods are the overloads that accept `Pose2d` objects.

For clamped cubic splines, this method accepts two `Pose2d` objects, one for the starting waypoint and one for the ending waypoint. The method takes in a vector of `Translation2d` objects which represent the interior waypoints. The headings at these interior waypoints are determined automatically to ensure continuous curvature. For quintic splines, the method simply takes in a list of `Pose2d` objects, with each `Pose2d` representing a point and heading on the field.

The more complex overload accepts “control vectors” for splines. This method is used when generating trajectories with Pathweaver, where you are able to control the magnitude of the tangent vector at each point. The `ControlVector` class consists of two double arrays. Each array represents one dimension (x or y), and its elements represent the derivatives at that point. For example, the value at element 0 of the x array represents the x coordinate (0th derivative), the value at element 1 represents the 1st derivative in the x dimension and so on.

When using clamped cubic splines, the length of the array must be 2 (0th and 1st derivatives), whereas when using quintic splines, the length of the array should be 3 (0th, 1st, and 2nd derivative). Unless you know exactly what you are doing, the first and simpler method is HIGHLY recommended for manually generating trajectories. (i.e. when not using Pathweaver JSON files).

Here is an example of generating a trajectory using clamped cubic splines for the 2018 game, FIRST Power Up:

Java

C++

```
class ExampleTrajectory {
    public void generateTrajectory() {

        // 2018 cross scale auto waypoints.
        var sideStart = new Pose2d(Units.feetToMeters(1.54), Units.feetToMeters(23.23),
            Rotation2d.fromDegrees(-180));
        var crossScale = new Pose2d(Units.feetToMeters(23.7), Units.feetToMeters(6.8),
            Rotation2d.fromDegrees(-160));

        var interiorWaypoints = new ArrayList<Translation2d>();
        interiorWaypoints.add(new Translation2d(Units.feetToMeters(14.54), Units.
↪ feetToMeters(23.23)));
        interiorWaypoints.add(new Translation2d(Units.feetToMeters(21.04), Units.
↪ feetToMeters(18.23)));

        TrajectoryConfig config = new TrajectoryConfig(Units.feetToMeters(12), Units.
↪ feetToMeters(12));
        config.setReversed(true);
    }
}
```

(continues on next page)

(continued from previous page)

```

    var trajectory = TrajectoryGenerator.generateTrajectory(
        sideStart,
        interiorWaypoints,
        crossScale,
        config);
}

```

```

void GenerateTrajectory() {
    // 2018 cross scale auto waypoints
    const frc::Pose2d sideStart{1.54_ft, 23.23_ft, frc::Rotation2d(180_deg)};
    const frc::Pose2d crossScale{23.7_ft, 6.8_ft, frc::Rotation2d(-160_deg)};

    std::vector<frc::Translation2d> interiorWaypoints{
        frc::Translation2d{14.54_ft, 23.23_ft},
        frc::Translation2d{21.04_ft, 18.23_ft}};

    frc::TrajectoryConfig config{12_fps, 12_fps_sq};
    config.SetReversed(true);

    auto trajectory = frc::TrajectoryGenerator::GenerateTrajectory(
        sideStart, interiorWaypoints, crossScale, config);
}

```

Note: The Java code utilizes the [Units](#) utility, which was added to WPILib in 2020 for easy unit conversions.

Note: Even though this trajectory generation is orders of magnitude faster than Pathfinder, it is highly recommended to generate all trajectories on startup (`robotInit`) as the generation time is still not negligible. Generation time often ranges from 10 ms to 25 ms for each trajectory.

35.6.2 Trajectory Constraints

In the [previous article](#), you might have noticed that no custom constraints were added when generating the trajectories. Custom constraints allow users to impose more restrictions on the velocity and acceleration at points along the trajectory based on location and curvature.

For example, a custom constraint can keep the velocity of the trajectory under a certain threshold in a certain region or slow down the robot near turns for stability purposes.

WPILib-Provided Constraints

WPILib includes a set of predefined constraints that users can utilize when generating trajectories. The list of WPILib-provided constraints is as follows:

- **CentripetalAccelerationConstraint:** Limits the centripetal acceleration of the robot as it traverses along the trajectory. This can help slow down the robot around tight turns.
- **DifferentialDriveKinematicsConstraint:** Limits the velocity of the robot around turns such that no wheel of a differential-drive robot goes over a specified maximum velocity.
- **DifferentialDriveVoltageConstraint:** Limits the acceleration of a differential drive robot such that no commanded voltage goes over a specified maximum.
- **EllipticalRegionConstraint:** Imposes a constraint only in an elliptical region on the field.
- **MaxVelocityConstraint:** Imposes a max velocity constraint. This can be composed with the **EllipticalRegionConstraint** or **RectangularRegionConstraint** to limit the velocity of the robot only in a specific region.
- **MecanumDriveKinematicsConstraint:** Limits the velocity of the robot around turns such that no wheel of a mecanum-drive robot goes over a specified maximum velocity.
- **RectangularRegionConstraint:** Imposes a constraint only in a rectangular region on the field.
- **SwerveDriveKinematicsConstraint:** Limits the velocity of the robot around turns such that no wheel of a swerve-drive robot goes over a specified maximum velocity.

Note: The **DifferentialDriveVoltageConstraint** only ensures that theoretical voltage commands do not go over the specified maximum using a *feedforward model*. If the robot were to deviate from the reference while tracking, the commanded voltage may be higher than the specified maximum.

Creating a Custom Constraint

Users can create their own constraint by implementing the **TrajectoryConstraint** interface.

Java

C++

```
@Override
public double getMaxVelocityMetersPerSecond(Pose2d poseMeters, double_
↪curvatureRadPerMeter,
                                     double velocityMetersPerSecond) {
    // code here
}

@Override
public MinMax getMinMaxAccelerationMetersPerSecondSq(Pose2d poseMeters,
                                                       double curvatureRadPerMeter,
                                                       double velocityMetersPerSecond) {
    // code here
}
```

```

units::meters_per_second_t MaxVelocity(
  const Pose2d& pose, units::curvature_t curvature,
  units::meters_per_second_t velocity) override {
  // code here
}

MinMax MinMaxAcceleration(const Pose2d& pose, units::curvature_t curvature,
                          units::meters_per_second_t speed) override {
  // code here
}

```

The `MaxVelocity` method should return the maximum allowed velocity for the given pose, curvature, and original velocity of the trajectory without any constraints. The `MinMaxAcceleration` method should return the minimum and maximum allowed acceleration for the given pose, curvature, and constrained velocity.

See the source code (Java, C++) for the WPILib-provided constraints for more examples on how to write your own custom trajectory constraints.

35.6.3 Manipulating Trajectories

Once a trajectory has been generated, you can retrieve information from it using certain methods. These methods will be useful when writing code to follow these trajectories.

Getting the total duration of the trajectory

Because all trajectories have timestamps at each point, the amount of time it should take for a robot to traverse the entire trajectory is pre-determined. The `TotalTime()` (C++) / `getTotalTimeSeconds()` (Java) method can be used to determine the time it takes to traverse the trajectory.

Java

C++

```

// Get the total time of the trajectory in seconds
double duration = trajectory.getTotalTimeSeconds();

```

```

// Get the total time of the trajectory
units::second_t duration = trajectory.TotalTime();

```

Sampling the trajectory

The trajectory can be sampled at various timesteps to get the pose, velocity, and acceleration at that point. The `Sample(units::second_t time)` (C++) / `sample(double timeSeconds)` (Java) method can be used to sample the trajectory at any timestep. The parameter refers to the amount of time passed since 0 seconds (the starting point of the trajectory). This method returns a `Trajectory::Sample` with information about that sample point.

Java

C++

```
// Sample the trajectory at 1.2 seconds. This represents where the robot
// should be after 1.2 seconds of traversal.
Trajectory.Sample point = trajectory.sample(1.2);
```

```
// Sample the trajectory at 1.2 seconds. This represents where the robot
// should be after 1.2 seconds of traversal.
Trajectory::State point = trajectory.Sample(1.2_s);
```

The `Trajectory::Sample` struct has several pieces of information about the sample point:

- `t`: The time elapsed from the beginning of the trajectory up to the sample point.
- `velocity`: The velocity at the sample point.
- `acceleration`: The acceleration at the sample point.
- `pose`: The pose (x, y, heading) at the sample point.
- `curvature`: The curvature (rate of change of heading with respect to distance along the trajectory) at the sample point.

Note: The angular velocity at the sample point can be calculated by multiplying the velocity by the curvature.

Getting all states of the trajectory (advanced)

A more advanced user can get a list of all states of the trajectory by calling the `States()` (C++)/`getStates()` (Java) method. Each state represents a point on the trajectory. *When the trajectory is created* using the `TrajectoryGenerator::GenerateTrajectory(...)` method, a list of trajectory points / states are created. When the user samples the trajectory at a particular timestep, a new sample point is interpolated between two existing points / states in the list.

35.6.4 Transforming Trajectories

Trajectories can be transformed from one coordinate system to another and moved within a coordinate system using the `relativeTo` and the `transformBy` methods. These methods are useful for moving trajectories within space, or redefining an already existing trajectory in another frame of reference.

Note: Neither of these methods changes the shape of the original trajectory.

The `relativeTo` Method

The `relativeTo` method is used to redefine an already existing trajectory in another frame of reference. This method takes one argument: a pose, (via a `Pose2d` object) that is defined with respect to the current coordinate system, that represents the origin of the new coordinate system.

For example, a trajectory defined in coordinate system A can be redefined in coordinate system B, whose origin is at (3, 3, 30 degrees) in coordinate system A, using the `relativeTo` method.

Java

C++

```
Pose2d bOrigin = new Pose2d(3, 3, Rotation2d.fromDegrees(30));
Trajectory bTrajectory = aTrajectory.relativeTo(bOrigin);
```

```
frc::Pose2d bOrigin{3_m, 3_m, frc::Rotation2d(30_deg)};
frc::Trajectory bTrajectory = aTrajectory.RelativeTo(bOrigin);
```



In the diagram above, the original trajectory (`aTrajectory` in the code above) has been defined in coordinate system A, represented by the black axes. The red axes, located at (3, 3) and 30° with respect to the original coordinate system, represent coordinate system B. Calling `relativeTo` on `aTrajectory` will redefine all poses in the trajectory to be relative to coordinate system B (red axes).

The transformBy Method

The transformBy method can be used to move (i.e. translate and rotate) a trajectory within a coordinate system. This method takes one argument: a transform (via a Transform2d object) that maps the current initial position of the trajectory to a desired initial position of the same trajectory.

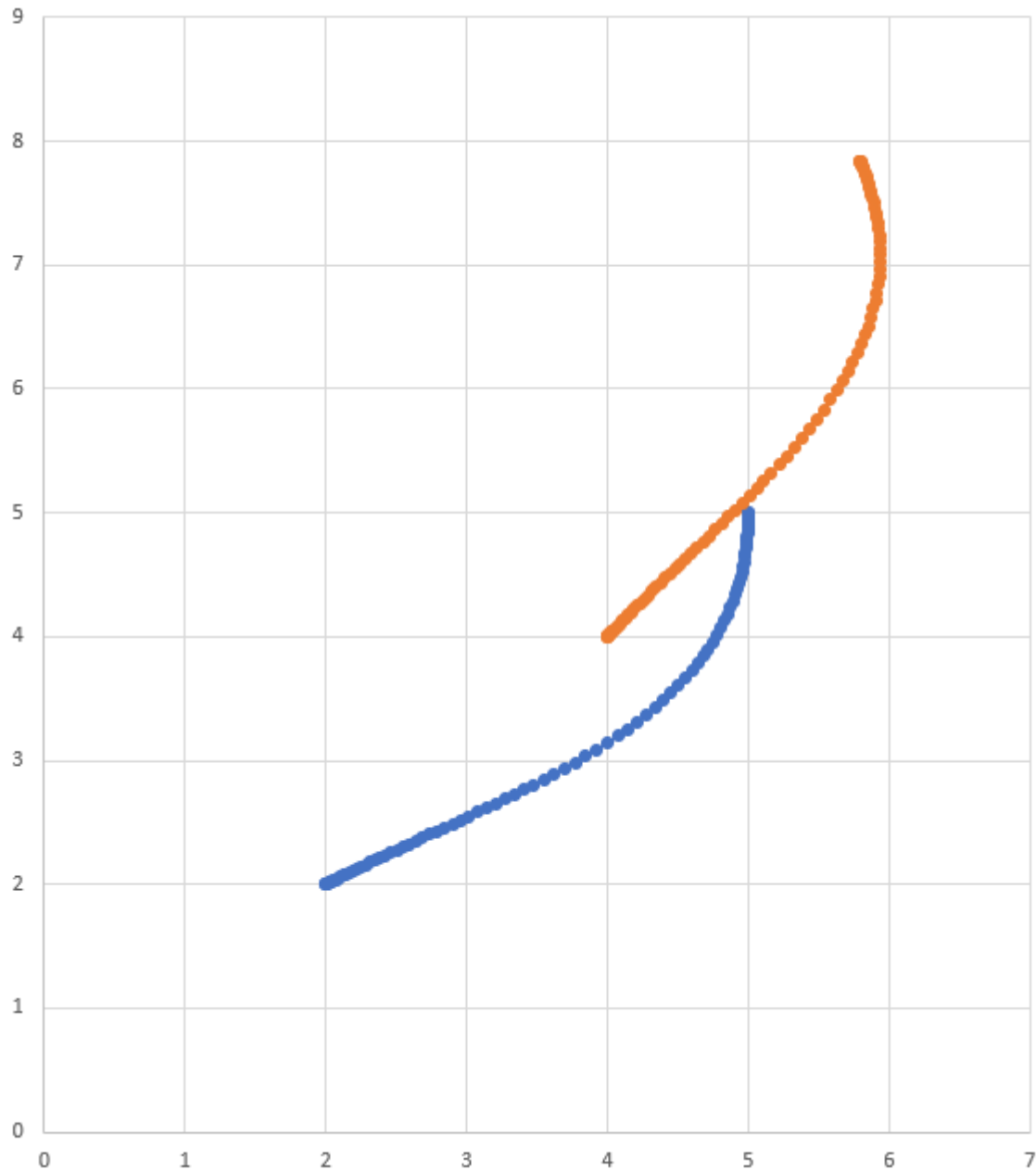
For example, one may want to transform a trajectory that begins at (2, 2, 30 degrees) to make it begin at (4, 4, 50 degrees) using the transformBy method.

Java

C++

```
Transform2d transform = new Pose2d(4, 4, Rotation2d.fromDegrees(50)).minus(trajecory.  
↪getInitialPose());  
Trajectory newTrajectory = trajectory.transformBy(transform);
```

```
frc::Transform2d transform = Pose2d(4_m, 4_m, Rotation2d(50_deg)) - trajectory.  
↪InitialPose();  
frc::Trajectory newTrajectory = trajectory.TransformBy(transform);
```

In the diagram above, the original trajectory, which starts at (2, 2) and at 30° is visible in blue. After applying the transform above, the resultant trajectory's starting location is changed to (4, 4) at 50° . The resultant trajectory is visible in orange.

35.6.5 Ramsete Controller

The Ramsete Controller is a trajectory tracker that is built in to WPILib. This tracker can be used to accurately track trajectories with correction for minor disturbances.

Constructing the Ramsete Controller Object

The Ramsete controller should be initialized with two gains, namely b and $zeta$. Larger values of b make convergence more aggressive like a proportional term whereas larger values of $zeta$ provide more damping in the response. These controller gains only dictate how the controller will output adjusted velocities. It does NOT affect the actual velocity tracking of the robot. This means that these controller gains are generally robot-agnostic.

Note: Gains of 2.0 and 0.7 for b and $zeta$ have been tested repeatedly to produce desirable results when all units were in meters. As such, a zero-argument constructor for `RamseteController` exists with gains defaulted to these values.

Java

C++

```
// Using the default constructor of RamseteController. Here
// the gains are initialized to 2.0 and 0.7.
RamseteController controller1 = new RamseteController();

// Using the secondary constructor of RamseteController where
// the user can choose any other gains.
RamseteController controller2 = new RamseteController(2.1, 0.8);
```

```
// Using the default constructor of RamseteController. Here
// the gains are initialized to 2.0 and 0.7.
frc::RamseteController controller1;

// Using the secondary constructor of RamseteController where
// the user can choose any other gains.
frc::RamseteController controller2{2.1, 0.8};
```

Getting Adjusted Velocities

The Ramsete controller returns “adjusted velocities” so that the when the robot tracks these velocities, it accurately reaches the goal point. The controller should be updated periodically with the new goal. The goal comprises of a desired pose, desired linear velocity, and desired angular velocity. Furthermore, the current position of the robot should also be updated periodically. The controller uses these four arguments to return the adjusted linear and angular velocity. Users should command their robot to these linear and angular velocities to achieve optimal trajectory tracking.

Note: The “goal pose” represents the position that the robot should be at at a particular timestep when tracking the trajectory. It does NOT represent the final endpoint of the trajectory.

The controller can be updated using the Calculate (C++) / calculate (Java) method. There are two overloads for this method. Both of these overloads accept the current robot position as the first parameter. For the other parameters, one of these overloads takes in the goal as three separate parameters (pose, linear velocity, and angular velocity) whereas the other overload accepts a Trajectory.State object, which contains information about the goal pose. For its ease, users should use the latter method when tracking trajectories.

Java

C++

```
Trajectory.State goal = trajectory.sample(3.4); // sample the trajectory at 3.4_s
↳seconds from the beginning
ChassisSpeeds adjustedSpeeds = controller.calculate(currentRobotPose, goal);
```

```
const Trajectory::State goal = trajectory.Sample(3.4_s); // sample the trajectory at
↳3.4 seconds from the beginning
ChassisSpeeds adjustedSpeeds = controller.Calculate(currentRobotPose, goal);
```

These calculations should be performed at every loop iteration, with an updated robot position and goal.

Using the Adjusted Velocities

The adjusted velocities are of type ChassisSpeeds, which contains a vx (linear velocity in the forward direction), a vy (linear velocity in the sideways direction), and an omega (angular velocity around the center of the robot frame). Because the Ramsete controller is a controller for non-holonomic robots (robots which cannot move sideways), the adjusted speeds object has a vy of zero.

The returned adjusted speeds can be converted to usable speeds using the kinematics classes for your drivetrain type. For example, the adjusted velocities can be converted to left and right velocities for a differential drive using a DifferentialDriveKinematics object.

Java

C++

```
ChassisSpeeds adjustedSpeeds = controller.calculate(currentRobotPose, goal);
DifferentialDriveWheelSpeeds wheelSpeeds = kinematics.toWheelSpeeds(adjustedSpeeds);
double left = wheelSpeeds.leftMetersPerSecond;
double right = wheelSpeeds.rightMetersPerSecond;
```

```
ChassisSpeeds adjustedSpeeds = controller.Calculate(currentRobotPose, goal);
DifferentialDriveWheelSpeeds wheelSpeeds = kinematics.ToWheelSpeeds(adjustedSpeeds);
auto [left, right] = kinematics.ToWheelSpeeds(adjustedSpeeds);
```

Because these new left and right velocities are still speeds and not voltages, two PID Controllers, one for each side may be used to track these velocities. Either the WPILib PIDController (C++, Java) can be used, or the Velocity PID feature on smart motor controllers such as the TalonSRX and the SPARK MAX can be used.

Ramsete in the Command-Based Framework

For the sake of ease for users, a `RamseteCommand` class is built in to WPILib. For a full tutorial on implementing a path-following autonomous using `RamseteCommand`, see [Trajectory Tutorial](#).

35.6.6 Holonomic Drive Controller

The holonomic drive controller is a trajectory tracker for robots with holonomic drivetrains (e.g. swerve, mecanum, etc.). This can be used to accurately track trajectories with correction for minor disturbances.

Constructing a Holonomic Drive Controller

The holonomic drive controller should be instantiated with 2 PID controllers and 1 profiled PID controller.

Note: For more information on PID control, see [PID Control in WPILib](#).

The 2 PID controllers are controllers that should correct for error in the field-relative x and y directions respectively. For example, if the first 2 arguments are `PIDController(1, 0, 0)` and `PIDController(1.2, 0, 0)` respectively, the holonomic drive controller will add an additional meter per second in the x direction for every meter of error in the x direction and will add an additional 1.2 meters per second in the y direction for every meter of error in the y direction.

The final parameter is a `ProfiledPIDController` for the rotation of the robot. Because the rotation dynamics of a holonomic drivetrain are decoupled from movement in the x and y directions, users can set custom heading references while following a trajectory. These heading references are profiled according to the parameters set in the `ProfiledPIDController`.

Java

C++

```
var controller = new HolonomicDriveController(
    new PIDController(1, 0, 0), new PIDController(1, 0, 0),
    new ProfiledPIDController(1, 0, 0,
        new TrapezoidProfile.Constraints(6.28, 3.14)));
// Here, our rotation profile constraints were a max velocity
// of 1 rotation per second and a max acceleration of 180 degrees
// per second squared.
```

```
frc::HolonomicDriveController controller{
    frc2::PIDController{1, 0, 0}, frc2::PIDController{1, 0, 0},
    frc::ProfiledPIDController<units::radian>{
        1, 0, 0, frc::TrapezoidProfile<units::radian>::Constraints{
            6.28_rad_per_s, 3.14_rad_per_s / 1_s}}};
// Here, our rotation profile constraints were a max velocity
// of 1 rotation per second and a max acceleration of 180 degrees
// per second squared.
```

Getting Adjusted Velocities

The holonomic drive controller returns “adjusted velocities” such that when the robot tracks these velocities, it accurately reaches the goal point. The controller should be updated periodically with the new goal. The goal is comprised of a desired pose, linear velocity, and heading.

Note: The “goal pose” represents the position that the robot should be at at a particular timestamp when tracking the trajectory. It does NOT represent the trajectory’s endpoint.

The controller can be updated using the `Calculate (C++) / calculate (Java)` method. There are two overloads for this method. Both of these overloads accept the current robot position as the first parameter and the desired heading as the last parameter. For the middle parameters, one overload accepts the desired pose and the linear velocity reference while the other accepts a `Trajectory.State` object, which contains information about the goal pose. The latter method is preferred for tracking trajectories.

Java

C++

```
// Sample the trajectory at 3.4 seconds from the beginning.
Trajectory.State goal = trajectory.sample(3.4);

// Get the adjusted speeds. Here, we want the robot to be facing
// 70 degrees (in the field-relative coordinate system).
ChassisSpeeds adjustedSpeeds = controller.calculate(
    currentRobotPose, goal, Rotation2d.fromDegrees(70.0));
```

```
// Sample the trajectory at 3.4 seconds from the beginning.
const auto goal = trajectory.Sample(3.4_s);

// Get the adjusted speeds. Here, we want the robot to be facing
// 70 degrees (in the field-relative coordinate system).
const auto adjustedSpeeds = controller.Calculate(
    currentRobotPose, goal, 70_deg);
```

Using the Adjusted Velocities

The adjusted velocities are of type `ChassisSpeeds`, which contains a `vx` (linear velocity in the forward direction), a `vy` (linear velocity in the sideways direction), and an `omega` (angular velocity around the center of the robot frame).

The returned adjusted speeds can be converted into usable speeds using the kinematics classes for your drivetrain type. In the example code below, we will assume a swerve drive robot; however, the kinematics code is exactly the same for a mecanum drive robot except using `MecanumDriveKinematics`.

Java

C++

```
SwerveModuleState[] moduleStates = kinematics.toSwerveModuleStates(adjustedSpeeds);
```

(continues on next page)

(continued from previous page)

```
SwerveModuleState frontLeft = moduleStates[0];  
SwerveModuleState frontRight = moduleStates[1];  
SwerveModuleState backLeft = moduleStates[2];  
SwerveModuleState backRight = moduleStates[3];
```

```
auto [fl, fr, bl, br] = kinematics.ToSwerveModuleStates(adjustedSpeeds);
```

Because these swerve module states are still speeds and angles, you will need to use PID controllers to set these speeds and angles.

35.6.7 Troubleshooting

Troubleshooting Complete Failures

There are a number of things that can cause your robot to do completely the wrong thing. The below checklist covers some common mistakes.

- My robot doesn't move.
 - Are you actually outputting to your motors?
 - Is a `MalformedSplineException` getting printed to the driver station? If yes, go to the `MalformedSplineException` section below.
 - Is your trajectory very short or in the wrong units?
- My robot swings around to drive the trajectory facing the other direction.
 - Are the start and end headings of your trajectory wrong?
 - Is your robot's gyro getting reset to the wrong heading?
 - *Do you have the reverse flag set incorrectly?*
 - Are your gyro angles clockwise positive? If so, you should negate them.
- My robot just drives in a straight line even though it should turn.
 - Is your gyro set up correctly and returning good data?
 - Are you passing your gyro heading to your odometry object with the correct units?
 - Is your track width correct? Is it in the correct units?
- I get a `MalformedSplineException` printout on the driver station and the robot doesn't move.
 - *Do you have the reverse flag set incorrectly?*
 - Do you have two waypoints very close together with approximately opposite headings?
 - Do you have two waypoints with the same (or nearly the same) coordinates?
- My robot drives way too far.
 - Are your encoder unit conversions set up correctly?
 - Are your encoders connected?
- My robot mostly does the right thing, but it's a little inaccurate.

- Go to the next section.

Troubleshooting Poor Performance

Note: This section is mostly concerned with troubleshooting poor trajectory tracking performance like a meter of error, not catastrophic failures like compilation errors, robots turning around and going in the wrong direction, or `MalformedSplineExceptions`.

Note: This section is designed for differential drive robots, but most of the ideas can be adapted to swerve drive or mecanum.

Poor trajectory tracking performance can be difficult to troubleshoot. Although the trajectory generator and follower are intended to be easy-to-use and performant out of the box, there are situations where your robot doesn't quite end up where it should. The trajectory generator and followers have many knobs to tune and many moving parts, so it can be difficult to know where to start, especially because it is difficult to locate the source of trajectory problems from the robot's general behavior.

Because it can be so hard to locate the layer of the trajectory generator and followers that is misbehaving, a systematic, layer-by-layer approach is recommended for general poor tracking performance (e.g. the robot is off by few feet or more than twenty degrees). The below steps are listed in the order that you should do them in; it is important to follow this order so that you can isolate the effects of different steps from each other.

Note: The below examples put diagnostic values onto *NetworkTables*. The easiest way to graph these values is to *use Shuffleboard's graphing capabilities*.

Verify Odometry

If your odometry is bad, then your Ramsete controller may misbehave, because it modifies your robot's target velocities based on where your odometry thinks the robot is.

1. Set up your code to record your robot's position after each odometry update:

Java

C++

```
NetworkTableEntry m_xEntry = NetworkTableInstance.getDefault().getTable(
    ↪ "troubleshooting").getEntry("X");
NetworkTableEntry m_yEntry = NetworkTableInstance.getDefault().getTable(
    ↪ "troubleshooting").getEntry("Y");

@Override
public void periodic() {
    // Update the odometry in the periodic block
    m_odometry.update(Rotation2d.fromDegrees(getHeading()), m_leftEncoder.
    ↪ getDistance(),
        m_rightEncoder.getDistance());
```

(continues on next page)

(continued from previous page)

```

var translation = m_odometry.getPoseMeters().getTranslation();
m_xEntry.setNumber(translation.getX());
m_yEntry.setNumber(translation.getY());
}

```

```

NetworkTableEntry m_xEntry = nt::NetworkTableInstance::GetDefault().GetTable(
    ↪ "troubleshooting")->GetEntry("X");
NetworkTableEntry m_yEntry = nt::NetworkTableInstance::GetDefault().GetTable(
    ↪ "troubleshooting")->GetEntry("Y");

void DriveSubsystem::Periodic() {
    // Implementation of subsystem periodic method goes here.
    m_odometry.Update(frc::Rotation2d(units::degree_t(GetHeading()),
        units::meter_t(m_leftEncoder.GetDistance()),
        units::meter_t(m_rightEncoder.GetDistance())));

    auto translation = m_odometry.GetPose().Translation();
    m_xEntry.SetDouble(translation.X().to<double>());
    m_yEntry.SetDouble(translation.Y().to<double>());
}

```

2. Lay out a tape measure parallel to your robot and push your robot out about one meter along the tape measure. Lay out a tape measure along the Y axis and start over, pushing your robot one meter along the X axis and one meter along the Y axis in a rough arc.
3. Compare X and Y reported by the robot to actual X and Y. If X is off by more than 5 centimeters in the first test then you should check that you measured your wheel diameter correctly, and that your wheels are not worn down. If the second test is off by more than 5 centimeters in either X or Y then your track width (distance from the center of the left wheel to the center of the right wheel) may be incorrect; if you're sure that you measured the track width correctly with a tape measure then your robot's wheels may be slipping in a way that is not accounted for by track width—if this is the case then you should [run the track width characterization](#) and use that track width instead of the one from your tape measure.

FRC Characterization Data Logger		
Select Save Location/Name	/home/declan/characterization-data.json	
Save Data	Add Timestamp: <input checked="" type="checkbox"/>	Angular Mode: <input type="checkbox"/>
Connect to Robot	Not connected	Team Number: 4915
Quasistatic Forward	Not Run	Quasistatic ramp rate (V/s): 0.25
Quasistatic Backward	Not Run	
Dynamic Forward	Not Run	Dynamic step voltage (V): 6
Dynamic Backward	Not Run	
Track Width	Not run	Rotation Wheel voltage (V): 5

Verify Feedforward

If your feedforwards are bad then the P controllers for each side of the robot will not track as well, and your `DifferentialDriveVoltageConstraint` will not limit your robot's acceleration accurately. We mostly want to turn off the wheel P controllers so that we can isolate and test the feedforwards.

1. First, we must set disable the P controller for each wheel. Set the P gain to 0 for every controller. In the `RamseteCommand` example, you would set `kPDriveVel` to 0:

Java

C++

```
136 new PIDController(DriveConstants.kPDriveVel, 0, 0),
137 new PIDController(DriveConstants.kPDriveVel, 0, 0),
```

```
80 frc2::PIDController(DriveConstants::kPDriveVel, 0, 0),
81 frc2::PIDController(DriveConstants::kPDriveVel, 0, 0),
```

2. (Java only) Next, we want to disable the Ramsete controller to make it easier to isolate our problematic behavior. This is a bit more involved, because we can't just set the gains (b and zeta) to 0. Pass the following into your `RamseteCommand`:

Java

```
// Paste this variable in
RamseteController disabledRamsete = new RamseteController() {
    @Override
    public ChassisSpeeds calculate(Pose2d currentPose, Pose2d poseRef, double_
    ↪ linearVelocityRefMeters,
        double angularVelocityRefRadiansPerSecond) {
        return new ChassisSpeeds(linearVelocityRefMeters, 0.0,
    ↪ angularVelocityRefRadiansPerSecond);
    }
};

// Be sure to pass your new disabledRamsete variable
RamseteCommand ramseteCommand = new RamseteCommand(
    exampleTrajectory,
    m_robotDrive::getPose,
    disabledRamsete,
    ...
);
```

3. Finally, we need to log desired wheel velocity and actual wheel velocity (you should put actual and desired velocities on the same graph if you're using Shuffleboard, or if your graphing software has that capability):

Java

C++

```
var table = NetworkTableInstance.getDefault().getTable("troubleshooting");
var leftReference = table.getEntry("left_reference");
var leftMeasurement = table.getEntry("left_measurement");
var rightReference = table.getEntry("right_reference");
var rightMeasurement = table.getEntry("right_measurement");
```

(continues on next page)

(continued from previous page)

```

var leftController = new PIDController(kPDriveVel, 0, 0);
var rightController = new PIDController(kPDriveVel, 0, 0);
RamseteCommand ramseteCommand = new RamseteCommand(
    exampleTrajectory,
    m_robotDrive::getPose,
    disabledRamsete, // Pass in disabledRamsete here
    new SimpleMotorFeedforward(ksVolts, kvVoltSecondsPerMeter,
↪ kaVoltSecondsSquaredPerMeter),
    kDriveKinematics,
    m_robotDrive::getWheelSpeeds,
    leftController,
    rightController,
    // RamseteCommand passes volts to the callback
    (leftVolts, rightVolts) -> {
        m_robotDrive.tankDriveVolts(leftVolts, rightVolts);

        leftMeasurement.setNumber(m_robotDrive.getWheelSpeeds().leftMetersPerSecond);
        leftReference.setNumber(leftController.getSetpoint());

        rightMeasurement.setNumber(m_robotDrive.getWheelSpeeds().
↪ rightMetersPerSecond);
        rightReference.setNumber(rightController.getSetpoint());
    },
    m_robotDrive
);

```

```

auto table =
    nt::NetworkTableInstance::GetDefault().GetTable("troubleshooting");
auto leftRef = table->GetEntry("left_reference");
auto leftMeas = table->GetEntry("left_measurement");
auto rightRef = table->GetEntry("right_reference");
auto rightMeas = table->GetEntry("right_measurement");

frc2::PIDController leftController(DriveConstants::kPDriveVel, 0, 0);
frc2::PIDController rightController(DriveConstants::kPDriveVel, 0, 0);
frc2::RamseteCommand ramseteCommand(
    exampleTrajectory, [this]() { return m_drive.GetPose(); },
    frc::RamseteController(AutoConstants::kRamseteB,
        AutoConstants::kRamseteZeta),
    frc::SimpleMotorFeedforward<units::meters>(
        DriveConstants::ks, DriveConstants::kv, DriveConstants::ka),
    DriveConstants::kDriveKinematics,
    [this] { return m_drive.GetWheelSpeeds(); }, leftController,
    rightController,
    [=](auto left, auto right) {
        auto leftReference = leftRef;
        auto leftMeasurement = leftMeas;
        auto rightReference = rightRef;
        auto rightMeasurement = rightMeas;

        m_drive.TankDriveVolts(left, right);

        leftMeasurement.SetDouble(m_drive.GetWheelSpeeds().left.to<double>());
        leftReference.SetDouble(leftController.GetSetpoint());
    }
);

```

(continues on next page)

(continued from previous page)

```

    rightMeasurement.SetDouble(m_drive.GetWheelSpeeds().right.to<double>());
    rightReference.SetDouble(rightController.GetSetpoint());
},
{&m_drive});

```

4. Run the robot on a variety of trajectories (curved and straight line), and check to see if the actual velocity tracks the desired velocity by looking at graphs from NetworkTables.
5. If the desired and actual are off by *a lot* then you should check if the wheel diameter and encoderEPR you used for characterization were correct. If you've verified that your units and conversions are correct, then you should try recharacterizing on the same floor that you're testing on to see if you can get better data.

Verify P Gain

If you completed the previous step and the problem went away then your problem can probably be found in one of the next steps. In this step we're going to verify that your wheel P controllers are well-tuned. If you're using Java then we want to turn off Ramsete so that we can just view our PF controllers on their own.

1. You must re-use all the code from the previous step that logs actual vs. desired velocity (and the code that disables Ramsete, if you're using Java), except that **the P gain must be set back to its previous nonzero value.**
2. Run the robot again on a variety of trajectories, and check that your actual vs. desired graphs look good.
3. If the graphs do not look good (i.e. the actual velocity is very different from the desired) then you should try tuning your P gain and rerunning your test trajectories.

Check Constraints

Note: Make sure that your P gain is nonzero for this step and that you still have the logging code added in the previous steps. If you're using Java then you should remove the code to disable Ramsete.

If your accuracy issue persisted through all of the previous steps then you might have an issue with your constraints. Below are a list of symptoms that the different available constraints will exhibit when poorly tuned.

Test one constraint at a time! Remove the other constraints, tune your one remaining constraint, and repeat that process for each constraint you want to use. The below checklist assumes that you only use one constraint at a time.

- DifferentialDriveVoltageConstraint:
 - If your robot accelerates very slowly then it's possible that the max voltage for this constraint is too low.
 - If your robot doesn't reach the end of the path then your characterization data may be problematic.
- DifferentialDriveKinematicsConstraint:

- If your robot ends up at the wrong heading then it's possible that the max drivetrain side speed is too low, or that it's too high. The only way to tell is to tune the max speed and to see what happens.
- CentripetalAccelerationConstraint:
 - If your robot ends up at the wrong heading then this could be the culprit. If your robot doesn't seem to turn enough then you should increase the max centripetal acceleration, but if it seems to go around tight turns too quickly then you should decrease the maximum centripetal acceleration.

Check Trajectory Waypoints

It is possible that your trajectory itself is not very driveable. Try moving waypoints (and headings at the waypoints, if applicable) to reduce sharp turns.

35.7 State-Space and Model Based Control with WPILib

This section provides an introduction to and describes WPILib support for state-space control.

35.7.1 Introduction to State-Space Control

Note: This article is from [Controls Engineering in FRC](#) by Tyler Veness with permission.

From PID to Model-Based Control

When tuning PID controllers, we focus on fiddling with controller parameters relating to the current, past, and future *error* (P, I and D terms) rather than the underlying system states. While this approach works in a lot of situations, it is an incomplete view of the world.

Model-based control focuses on developing an accurate model of the *system* (mechanism) we are trying to control. These models help inform *gains* picked for feedback controllers based on the physical responses of the system, rather than an arbitrary proportional *gain* derived through testing. This allows us not only to predict ahead of time how a system will react, but also test our controllers without a physical robot and save time debugging simple bugs.

Note: State-space control makes extensive use of linear algebra. More on linear algebra in modern control theory, including an introduction to linear algebra and resources, can be found in Chapter 4 of [Controls Engineering in FRC](#).

If you've used WPILib's feedforward classes for `SimpleMotorFeedforward` or its sister classes, or used FRC-Characterization to pick PID *gains* for you, you're already familiar with model-based control! The *kv* and *ka* *gains* can be used to describe how a motor (or arm, or drivetrain) will react to voltage. We can put these constants into standard state-space notation using WPILib's `LinearSystem`, something we will do in a later article.

Vocabulary

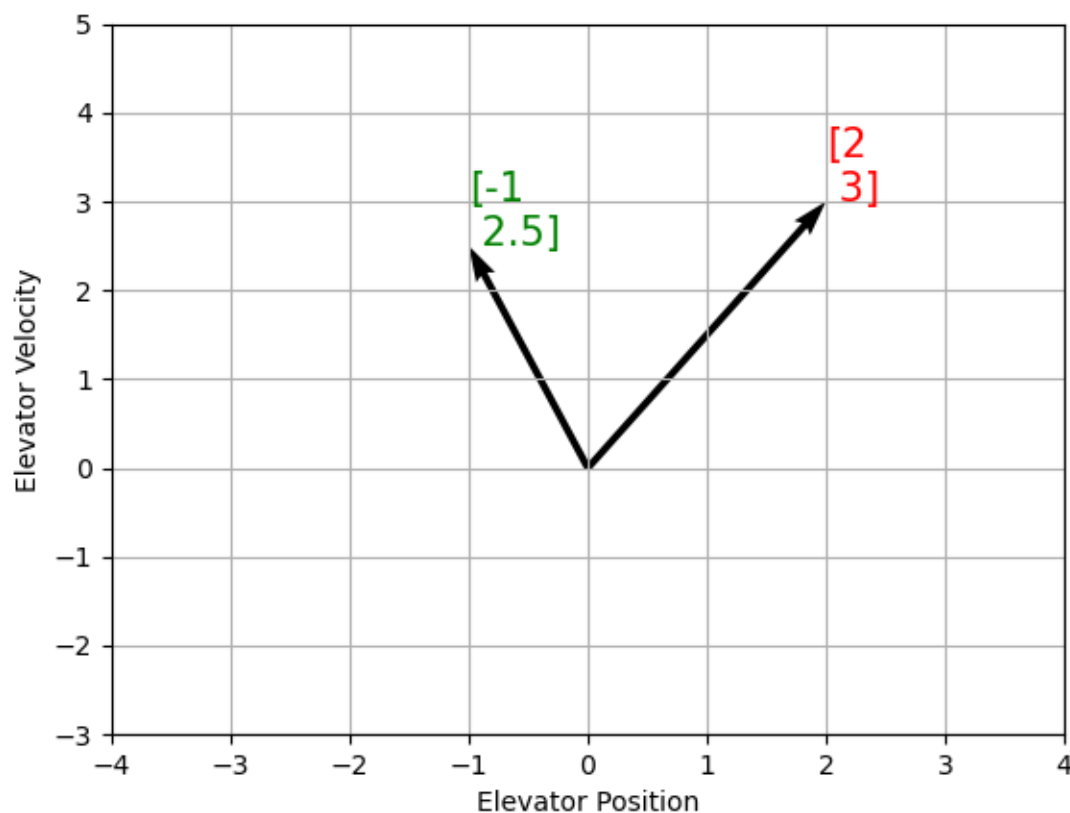
For the background vocabulary that will be used throughout this article, see the [Glossary](#).

Introduction to Linear Algebra

For a short and intuitive introduction to the core concepts of Linear Algebra, we recommend chapters 1 through 4 of [3Blue1Brown's Essence of linear algebra series](#) (Vectors, what even are they?, Linear combinations, span, and basis vectors, Linear transformations and matrices, and Matrix multiplication as composition).

What is State-Space?

Recall that 2D space has two axes: x and y . We represent locations within this space as a pair of numbers packaged in a vector, and each coordinate is a measure of how far to move along the corresponding axis. State-space is a [Cartesian coordinate system](#) with an axis for each state variable, and we represent locations within it the same way we do for 2D space: with a list of numbers in a vector. Each element in the vector corresponds to a state of the system. This example shows two example state vectors in the state-space of an elevator model with the states `[position, velocity]`:



In this image, the vectors representing states in state-space are arrows. From now on these vectors will be represented simply by a point at the vector's tip, but remember that the rest of the vector is still there.

In addition to the *state*, *inputs* and *outputs* are represented as vectors. Since the mapping from the current states and inputs to the change in state is a system of equations, it's natural to write it in matrix form. This matrix equation can be written in state-space notation.

What is State-Space Notation?

State-space notation is a set of matrix equations which describe how a system will evolve over time. These equations relate the change in state $\dot{\mathbf{x}}$, and the *output* \mathbf{y} , to linear combinations of the current state vector \mathbf{x} and *input* vector \mathbf{u} .

State-space control can deal with continuous-time and discrete-time systems. In the continuous-time case, the rate of change of the system's state \mathbf{x} is expressed as a linear combination of the current state \mathbf{x} and input \mathbf{u} .

In contrast, discrete-time systems expresses the state of the system at our next timestep \mathbf{x}_{k+1} based on the current state \mathbf{x}_k and input \mathbf{u}_k , where k is the current timestep and $k + 1$ is the next timestep.

In both the continuous- and discrete-time forms, the *output* vector \mathbf{y} is expressed as a linear combination of the current *state* and *input*. In many cases, the output is a subset of the system's state, and has no contribution from the current input.

When modeling systems, we first derive the continuous-time representation because the equations of motion are naturally written as the rate of change of a system's state as a linear combination of its current state and inputs. We convert this representation to discrete-time on the robot because we update the system in discrete timesteps there instead of continuously.

The following two sets of equations are the standard form of continuous-time and discrete-time state-space notation:

$$\text{Continuous: } \dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$$

$$\mathbf{y} = \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{u}$$

$$\text{Discrete: } \mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k$$

$$\mathbf{y}_k = \mathbf{C}\mathbf{x}_k + \mathbf{D}\mathbf{u}_k$$

A	system matrix	x	state vector
B	input matrix	u	input vector
C	output matrix	y	output vector
D	feedthrough matrix		

A continuous-time state-space system can be converted into a discrete-time system through a process called discretization.

Note: In the discrete-time form, the system's state is held constant between updates. This means that we can only react to disturbances as quickly as our state estimate is updated. Updating our estimate more quickly can help improve performance, up to a point. WPILib's Notifier class can be used if updates faster than the main robot loop are desired.

Note: While a system's continuous-time and discrete-time matrices A, B, C, and D have the same names, they are not equivalent. The continuous-time matrices describes the rate of

change of the state, \mathbf{x} , while the discrete-time matrices describe the system's state at the next timestep as a function of the current state and input.

Important: WPILib's LinearSystem takes continuous-time system matrices, and converts them internally to the discrete-time form where necessary.

State-space Notation Example: Flywheel from kV and kA

Recall that we can model the motion of a flywheel connected to a brushed DC motor with the equation $V = kV \cdot v + kA \cdot a$, where V is voltage output, v is the flywheel's angular velocity and a is its angular acceleration. This equation can be rewritten as $a = \frac{V - kV \cdot v}{kA}$, or $a = \frac{-kV}{kA} \cdot v + \frac{1}{kA} \cdot V$. Notice anything familiar? This equation relates the angular acceleration of the flywheel to its angular velocity and the voltage applied.

We can convert this equation to state-space notation. We can create a system with one state (velocity), one *input* (voltage), and one *output* (velocity). Recalling that the first derivative of velocity is acceleration, we can write our equation as follows, replacing velocity with \mathbf{x} , acceleration with $\dot{\mathbf{x}}$, and voltage V with \mathbf{u} :

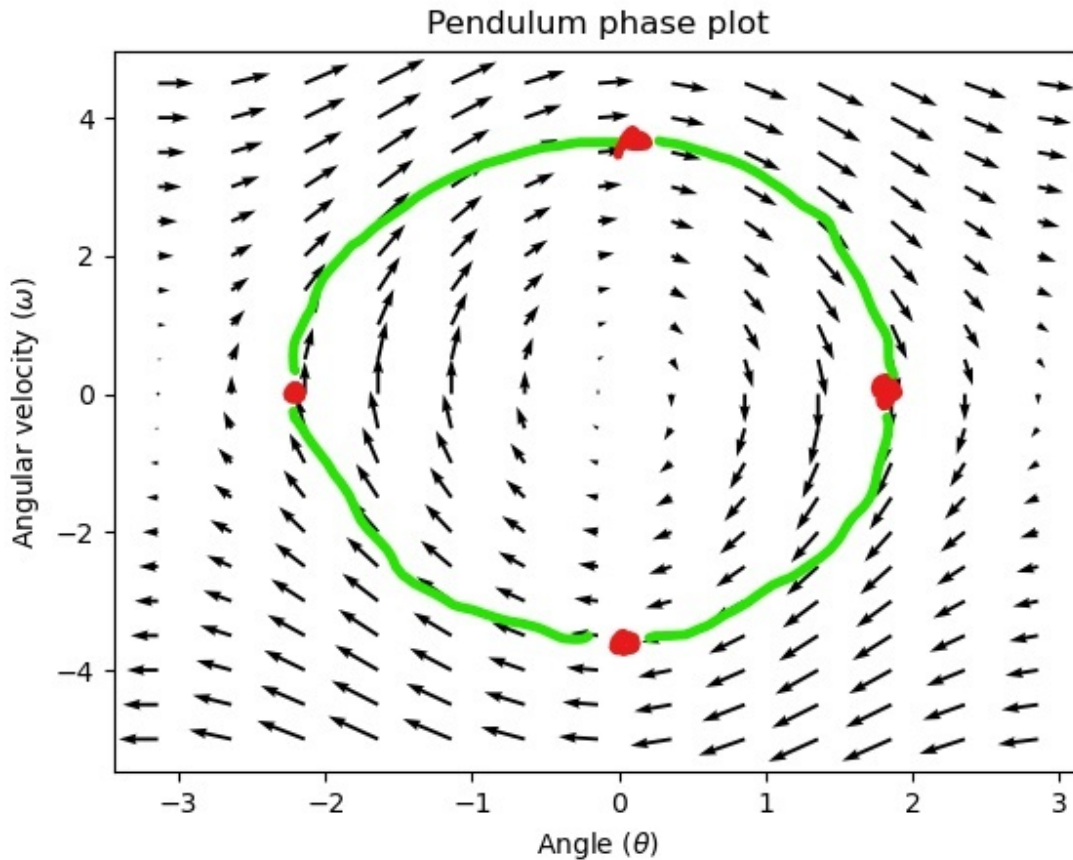
$$\dot{\mathbf{x}} = \left[\frac{-kV}{kA} \right] \mathbf{x} + \left[\frac{1}{kA} \right] \mathbf{u}$$

That's it! That's the state-space model of a system for which we have the kV and kA constants. This same math is use in FRC-Characterization to model flywheels and drivetrain velocity systems.

Visualizing State-Space Responses: Phase Portrait

A *phase portrait* can help give a visual intuition for the response of a system in state-space. The vectors on the graph have their roots at some point \mathbf{x} in state-space, and point in the direction of $\dot{\mathbf{x}}$, the direction that the system will evolve over time. This example shows a model of a pendulum with the states of angle and angular velocity.

To trace a potential trajectory that a system could take through state-space, choose a point to start at and follow the arrows around. In this example, we might start at $[-2, 0]$. From there, the velocity increases as we swing through vertical and starts to decrease until we reach the opposite extreme of the swing. This cycle of spinning about the origin repeats indefinitely.



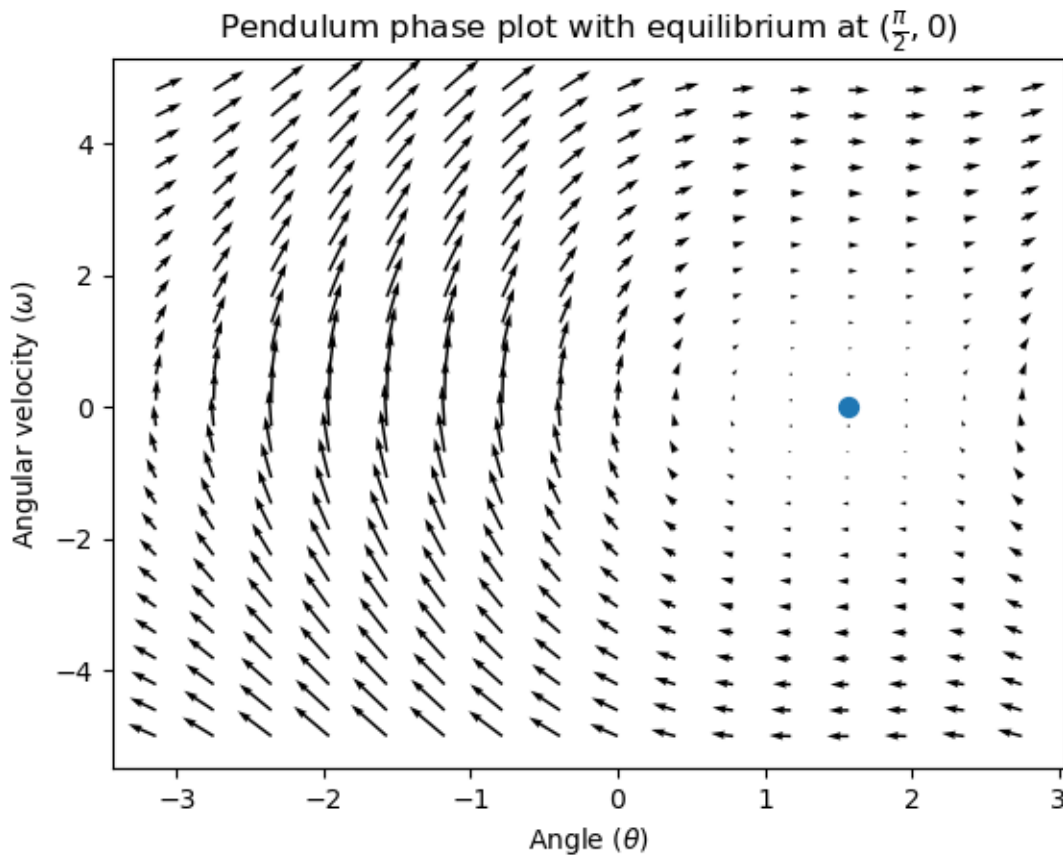
Note that near the edges of the phase portrait, the X axis wraps around as a rotation of π radians counter clockwise and a rotation of π radians clockwise will end at the same point.

For more on differential equations and phase portraits, see [3Blue1Brown's Differential Equations video](#) - they do a great job of animating the pendulum phase space at around 15:30.

Visualizing Feedforward

This phase portrait shows the “open loop” responses of the system - that is, how it will react if we were to let the state evolve naturally. If we want to, say, balance the pendulum horizontal (at $(\frac{\pi}{2}, 0)$ in state space), we would need to somehow apply a control *input* to counteract the open loop tendency of the pendulum to swing downward. This is what feedforward is trying to do - make it so that our phase portrait will have an equilibrium at the *reference* position (or setpoint) in state-space.

Looking at our phase portrait from before, we can see that at $(\frac{\pi}{2}, 0)$ in state space, gravity is pulling the pendulum down with some torque T , and producing some downward angular acceleration with magnitude $\frac{T}{I}$, where I is angular *moment of inertia* of the pendulum. If we want to create an equilibrium at our *reference* of $(\frac{\pi}{2}, 0)$, we would need to apply an *input* can counteract the system's natural tendency to swing downward. The goal here is to solve the equation $\mathbf{0} = \mathbf{Ax} + \mathbf{Bu}$ for \mathbf{u} . Below is shown a phase portrait where we apply a constant *input* that opposes the force of gravity at $(\frac{\pi}{2}, 0)$:



Feedback Control

In the case of a DC motor, with just a mathematical model and knowledge of all current states of the system (i.e., angular velocity), we can predict all future states given the future voltage inputs. But if the system is disturbed in any way that isn't modeled by our equations, like a load or unexpected friction, the angular velocity of the motor will deviate from the model over time. To combat this, we can give the motor corrective commands using a feedback controller.

A PID controller is a form of feedback control. State-space control often uses the following *control law*, where \mathbf{K} is some controller *gain* matrix, \mathbf{r} is the *reference* state, and \mathbf{x} is the current state in state-space. The difference between these two vectors, $\mathbf{r} - \mathbf{x}$, is the *error*.

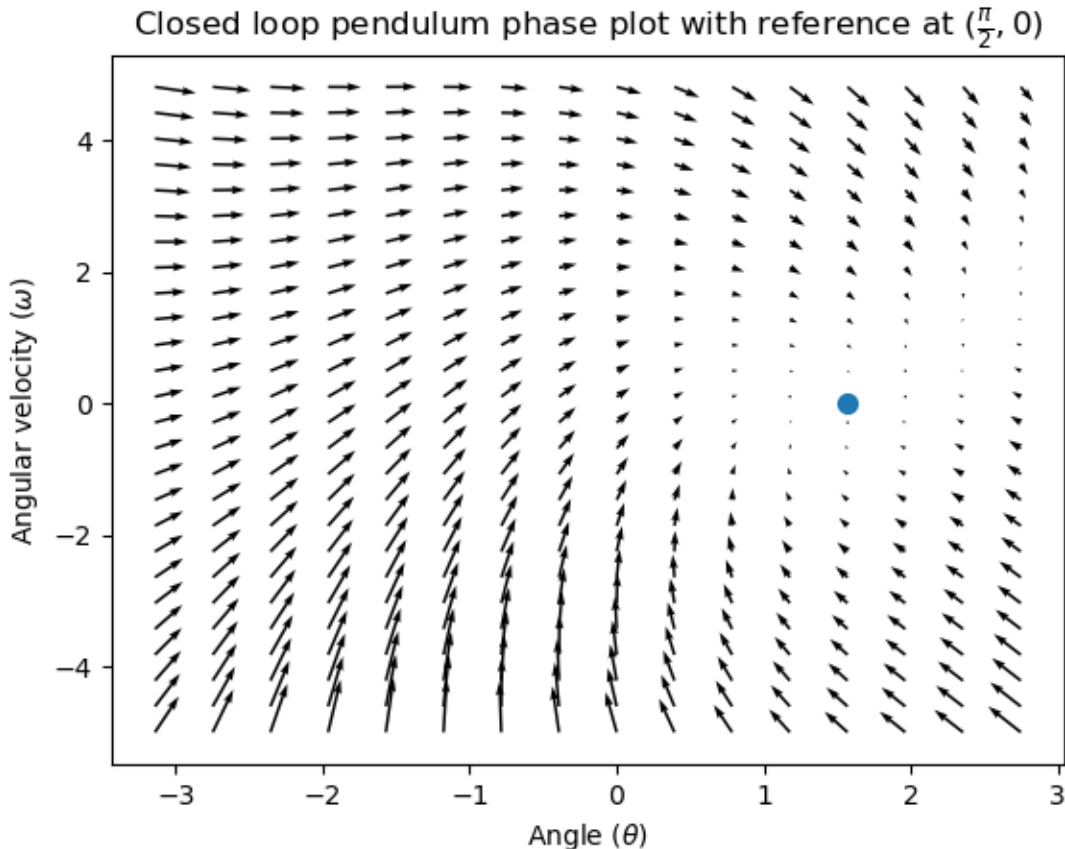
$$\mathbf{u} = \mathbf{K}(\mathbf{r} - \mathbf{x})$$

This *control law* is a proportional controller for each state of our system. Proportional controllers create software-defined springs that pull our system's state toward our reference state in state-space. In the case that the system being controlled has position and velocity states, the *control law* above will behave as a PD controller, which also tries to drive position and velocity error to zero.

Let's show an example of this control law in action. We'll use the pendulum system from above, where the swinging pendulum circled the origin in state-space. The case where \mathbf{K}

is the zero matrix (a matrix with all zeros) would be like picking P and D gains of zero – no control *input* would be applied, and the phase portrait would look identical to the one above.

To add some feedback, we arbitrarily pick a \mathbf{K} of $[2, 2]$, where our *input* to the pendulum is angular acceleration. This \mathbf{K} would mean that for every radian of position *error*, the angular acceleration would be 2 radians per second squared; similarly, we accelerate by 2 radians per second squared for every radian per second of *error*. Try following an arrow from somewhere in state-space inwards – no matter the initial conditions, the state will settle at the *reference* rather than circle endlessly with pure feedforward.



But how can we choose an optimal *gain* matrix \mathbf{K} for our system? While we can manually choose *gains* and simulate the system response or tune it on-robot like a PID controller, modern control theory has a better answer: the Linear-Quadratic Regulator (LQR).

The Linear-Quadratic Regulator

Because model-based control means that we can predict the future states of a system given an initial condition and future control inputs, we can pick a mathematically optimal *gain* matrix \mathbf{K} . To do this, we first have to define what a “good” or “bad” \mathbf{K} would look like. We do this by summing the square of error and control input over time, which gives us a number representing how “bad” our control law will be. If we minimize this sum, we will have arrived at the optimal control law.

LQR: Definition

Linear-Quadratic Regulators work by finding a *control law* that minimizes the following cost function, which weights the sum of *error* and *control effort* over time, subject to the linear *system* dynamics $\mathbf{x} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$.

$$J = \int_0^{\infty} (\mathbf{x}^T \mathbf{Q} \mathbf{x} + \mathbf{u}^T \mathbf{R} \mathbf{u}) dt$$

The *control law* that minimizes J can be written as $\mathbf{u} = \mathbf{K}(\mathbf{r} - \mathbf{x})$, where $r - x$ is the *error*.

Note: LQR design's \mathbf{Q} and \mathbf{R} matrices don't need discretization, but the \mathbf{K} calculated for continuous-time and discrete time *systems* will be different.

LQR: tuning

Like PID controllers can be tuned by adjusting their gains, we also want to change how our control law balances our error and input. For example, a spaceship might want to minimize the fuel it expends to reach a given reference, while a high-speed robotic arm might need to react quickly to disturbances.

We can weight error and control effort in our LQR with \mathbf{Q} and \mathbf{R} matrices. In our cost function (which describes how “bad” our control law will perform), \mathbf{Q} and \mathbf{R} weight our error and control input relative to each other. In the spaceship example from above, we might use a \mathbf{Q} with relatively small numbers to show that we don't want to highly penalize error, while our \mathbf{R} might be large to show that expending fuel is undesirable.

With WPILib, the LQR class takes a vector of desired maximum state excursions and control efforts and converts them internally to full \mathbf{Q} and \mathbf{R} matrices with Bryson's rule. We often use lowercase \mathbf{q} and \mathbf{r} to refer to these vectors, and \mathbf{Q} and \mathbf{R} to refer to the matrices.

Increasing the \mathbf{q} elements would make the LQR less heavily weight large errors, and the resulting *control law* will behave more conservatively. This has a similar effect to penalizing *control effort* more heavily by decreasing \mathbf{q} 's elements.

Similarly, decreasing the \mathbf{q} elements would make the LQR penalize large errors more heavily, and the resulting *control law* will behave more aggressively. This has a similar effect to penalizing *control effort* less heavily by increasing \mathbf{q} elements.

For example, we might use the following \mathbf{Q} and \mathbf{R} for an elevator system with position and velocity states.

Java

C++

```
// Example system -- must be changed to match your robot.
LinearSystem<N2, N1, N1> elevatorSystem = LinearSystemId.identifyPositionSystem(5, 0.
↪5);
LinearQuadraticRegulator<N2, N1, N1> controller = new
↪LinearQuadraticRegulator(elevatorSystem,
    // q's elements
    VecBuilder.fill(0.02, 0.4),
    // r's elements
```

(continues on next page)

(continued from previous page)

```
VecBuilder.fill(12.0),
// our dt
0.020);
```

```
// Example system -- must be changed to match your robot.
LinearSystem<2, 1, 1> elevatorSystem = frc::LinearSystemId::IdentifyVelocitySystem(5,
→ 0.5);
LinearQuadraticRegulator<2, 1> controller{
    elevatorSystem,
    // q's elements
    {0.02, 0.4},
    // r's elements
    {12.0},
    // our dt
    0.020_s};
```

LQR: example application

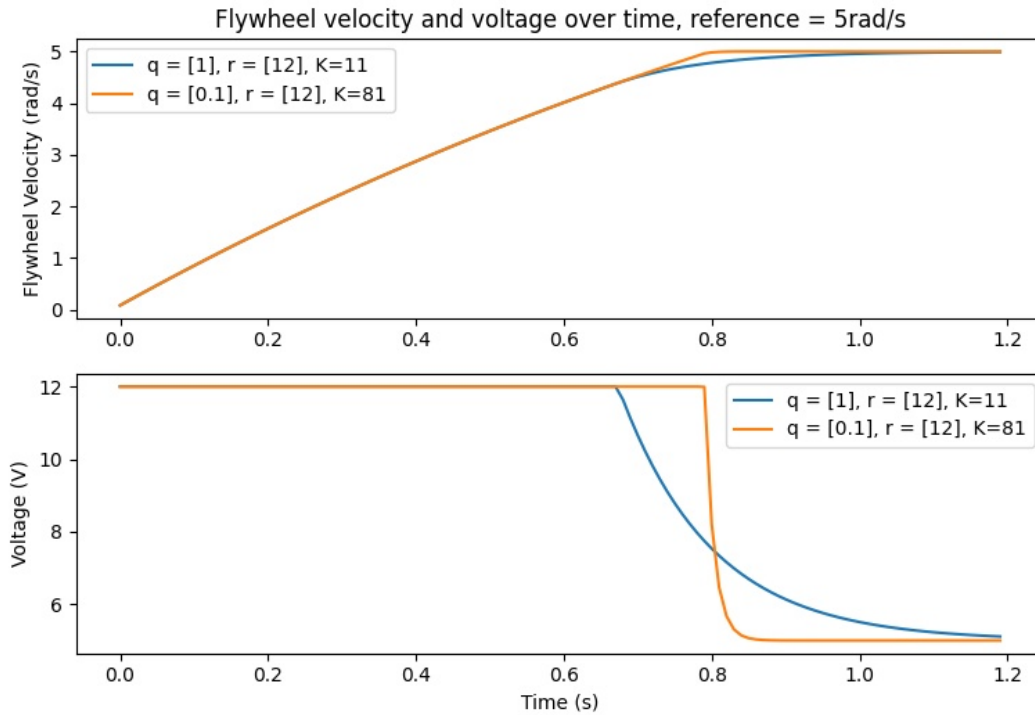
Let's apply a Linear-Quadratic Regulator to a real-world example. Say we have a flywheel velocity system determined through system identification to have $kV = 1 \frac{\text{volts}}{\text{radian per second}}$ and $kA = 1.5 \frac{\text{volts}}{\text{radian per second squared}}$. Using the flywheel example above, we have the following linear system:

$$\mathbf{x} = \begin{bmatrix} -kV \\ kA \end{bmatrix} v + \begin{bmatrix} 1 \\ 0 \end{bmatrix} V$$

We arbitrarily choose a desired state excursion (maximum error) of $q = [0.1 \text{ rad/sec}]$, and an \mathbf{r} of $[12 \text{ volts}]$. After discretization with a timestep of 20ms, we find a *gain* of $\mathbf{K} = 81$. This *gain* acts as the proportional component of a PID loop on flywheel's velocity.

Let's adjust \mathbf{q} and \mathbf{r} . We know that increasing the \mathbf{q} elements or decreasing the \mathbf{r} elements we use to create \mathbf{Q} and \mathbf{R} would make our controller more heavily penalize *control effort*, analogous to trying to driving a car more conservatively to improve fuel economy. In fact, if we increase our *error* tolerance q from 0.1 to 1.0, our *gain* matrix \mathbf{K} drops from ~ 81 to ~ 11 . Similarly, decreasing our maximum voltage r to 1.2 from 12.0 produces the same resultant \mathbf{K} .

The following graph shows the flywheel's angular velocity and applied voltage over time with two different *gains*. We can see how a higher *gain* will make the system reach the reference more quickly (at $t = 0.8$ seconds), while keeping our motor saturated at 12V for longer. This is exactly the same as increasing the P gain of a PID controller by a factor of $\sim 8x$.



LQR and Measurement Latency Compensation

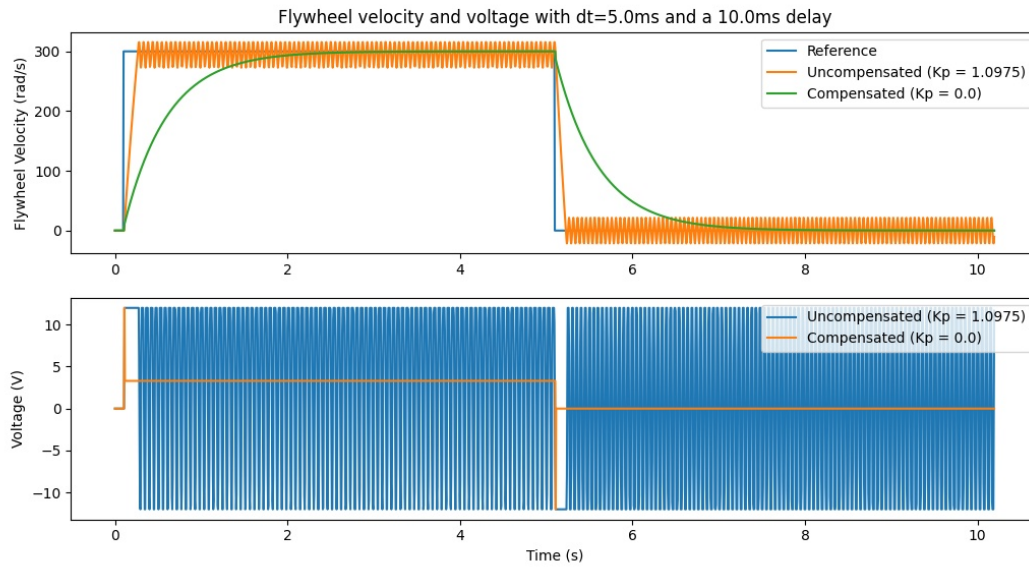
Oftentimes, our sensors have a delay associated with their measurements. For example the SPARK MAX motor controller over CAN can have up to 30ms of delay associated with velocity measurements.

This lag means that our feedback controller will be generating voltage commands based on state estimates from the past. This often has the effect of introducing instability and oscillations into our system, as shown in the graph below.

However, we can model our controller to control where the system's *state* is delayed into the future. This will reduce the LQR's *gain* matrix \mathbf{K} , trading off controller performance for stability. The below formula, which adjusts the *gain* matrix to account for delay, is also used in frc-characterization.

$$\mathbf{K}_{\text{compensated}} = \mathbf{K} \cdot (\mathbf{A} - \mathbf{BK})^{\text{delay}/dt}$$

Multiplying \mathbf{K} by $\mathbf{A} - \mathbf{BK}$ essentially advances the gains by one timestep. In this case, we multiply by $(\mathbf{A} - \mathbf{BK})^{\text{delay}/dt}$ to advance the gains by measurement's delay.



Note: This can have the effect of reducing K to zero, effectively disabling feedback control.

Note: The SPARK MAX motor controller uses a 40-tap FIR filter with a delay of 19.5ms , and status frames are by default sent every 20ms .

The code below shows how to adjust the LQR controller's K gain for sensor input delays:

Java

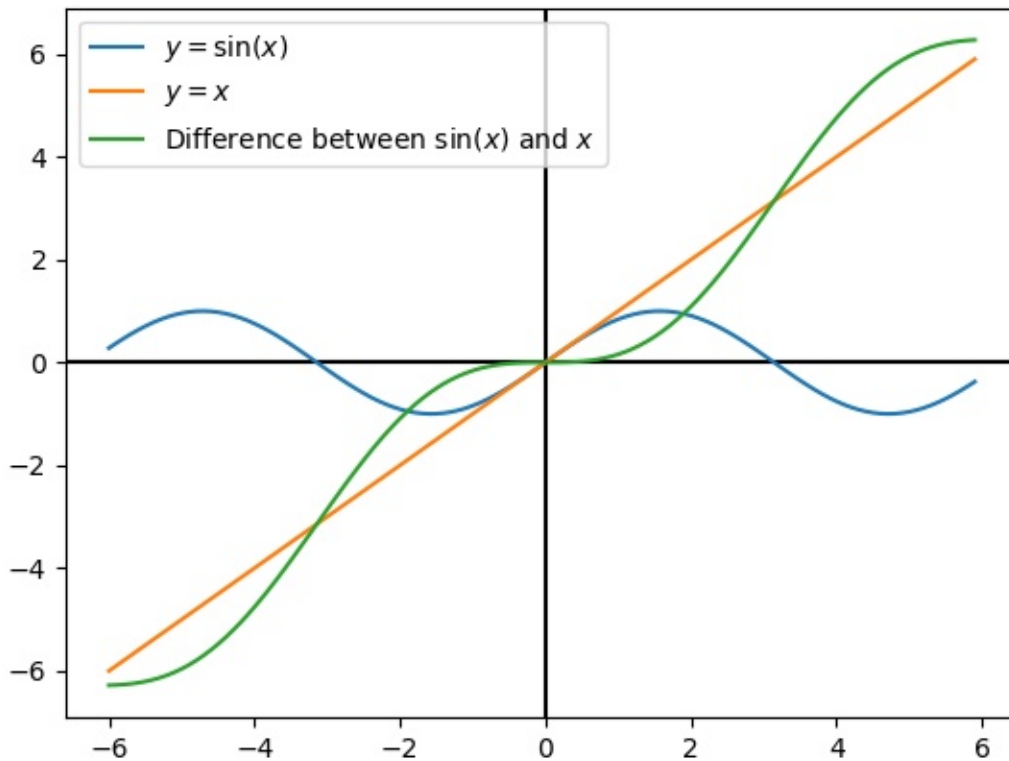
C++

```
// Adjust our LQR's controller for 25 ms of sensor input delay. We
// provide the linear system, discretization timestep, and the sensor
// input delay as arguments.
controller.latencyCompensate(elevatorSystem, 0.02, 0.025);
```

```
// Adjust our LQR's controller for 25 ms of sensor input delay. We
// provide the linear system, discretization timestep, and the sensor
// input delay as arguments.
controller.LatencyCompensate(elevatorSystem, 20_ms, 25_ms);
```

Linearization

Linearization is a tool used to approximate nonlinear functions and state-space systems using linear ones. In two-dimensional space, linear functions are straight lines while nonlinear functions curve. A common example of a nonlinear function and its corresponding linear approximation is $y = \sin x$. This function can be approximated by $y = x$ near zero. This approximation is accurate while near $x = 0$, but loses accuracy as we stray further from the linearization point. For example, the approximation $\sin x \approx x$ is accurate to within 0.02 within 0.5 radians of $y = 0$, but quickly loses accuracy past that. In the following picture, we see $y = \sin x$, $y = x$ and the difference between the approximation and the true value of $\sin x$ at x .



We can also linearize state-space systems with nonlinear *dynamics*. We do this by picking a point \mathbf{x} in state-space and using this as the input to our nonlinear functions. Like in the above example, this works well for states near the point about which the system was linearized, but can quickly diverge further from that state.

35.7.2 State-Space Controller Walkthrough

Note: Before following this tutorial, readers are recommended to have read an *Introduction to State-Space Control*.

The goal of this tutorial is to provide “end-to-end” instructions on implementing a state-space controller for a flywheel. By following this tutorial, readers will learn how to:

1. Create an accurate state-space model of a flywheel using *system identification* or CAD software.
2. Implement a Kalman Filter to filter encoder velocity measurements without lag.
3. Implement a *LQR* feedback controller which, when combined with model-based feedforward, will generate voltage *inputs* to drive the flywheel to a *reference*.

This tutorial is intended to be approachable for teams without a great deal of programming expertise. While the WPILib library offers significant flexibility in the manner in which its state-space control features are implemented, closely following the implementation outlined in this tutorial should provide teams with a basic structure which can be reused for a variety of state-space systems.

The full example is available in the state-space flywheel (Java/C++) and state-space flywheel system identification (Java/C++) example projects.

Why Use State-Space Control?

Because state-space control focuses on creating an accurate model of our system, we can accurately predict how our *model* will respond to control *inputs*. This allows us to simulate our mechanisms without access to a physical robot, as well as easily choose *gains* that we know will work well. Having a model also allows us to create lagless filters, such as Kalman Filters, to optimally filter sensor readings.

Modeling Our Flywheel

Recall that continuous state-space systems are modeled using the following system of equations:

$$\begin{aligned}\dot{\mathbf{x}} &= \mathbf{Ax} + \mathbf{Bu} \\ \mathbf{y} &= \mathbf{Cx} + \mathbf{Du}\end{aligned}$$

Where *x-dot* is the rate of change of the *system's state*, *x* is the system's current state, *u* is the *input* to the system, and *y* is the system's *output*.

Let's use this system of equations to model our flywheel in two different ways. We'll first model it using *system identification* using the frc-characterization toolsuite, and then model it based on the motor and flywheel's *moment of inertia*.

The first step of building up our state-space system is picking our system's states. We can pick anything we want as a state - we could pick completely unrelated states if we wanted - but it helps to pick states that are important. We can include *hidden states* in our state (such as elevator velocity if we were only able to measure its position) and let our Kalman Filter estimate their values. Remember that the states we choose will be driven towards their

respective *references* by the feedback controller (typically the *Linear-Quadratic Regulator* since it's optimal).

For our flywheel, we care only about one state: its velocity. While we could choose to also model its acceleration, the inclusion of this state isn't necessary for our system.

Next, we identify the *inputs* to our system. Inputs can be thought of as things we can put "into" our system to change its state. In the case of the flywheel (and many other single-jointed mechanisms in FRC®), we have just one input: voltage applied to the motor. By choosing voltage as our input (over something like motor duty cycle), we can compensate for battery voltage sag as battery load increases.

A continuous-time state-space system writes *x-dot*, or the instantaneous rate of change of the system's *system*'s state, as proportional to the current *state* and *inputs*. Because our state is angular velocity, *x* will be the flywheel's angular acceleration.

Next, we will model our flywheel as a continuous-time state-space system. WPILib's *LinearSystem* will convert this to discrete-time internally. Review *state-space notation* for more on continuous-time and discrete-time systems.

Modeling with System Identification

To rewrite this in state-space notation using *system identification*, we recall from the flywheel *state-space notation example*, where we rewrote the following equation in terms of *a*.

$$V = kV \cdot \mathbf{v} + kA \cdot \mathbf{a}$$

$$\mathbf{a} = \mathbf{v} = \begin{bmatrix} -kV \\ kA \end{bmatrix} v + \begin{bmatrix} 1 \\ kA \end{bmatrix} V$$

Where *v* is flywheel velocity, *a* and *v* are flywheel acceleration, and *V* is voltage. Rewriting this with the standard convention of *x* for the state vector and *u* for the input vector, we find:

$$\dot{\mathbf{x}} = \begin{bmatrix} -kV \\ kA \end{bmatrix} \mathbf{x} + \begin{bmatrix} 1 \\ kA \end{bmatrix} \mathbf{u}$$

The second part of state-space notation relates the system's current *state* and *inputs* to the *output*. In the case of a flywheel, our output vector *y* (or things that our sensors can measure) is our flywheel's velocity, which also happens to be an element of our *state* vector *x*. Therefore, our output matrix is *C* = [1], and our system feedthrough matrix is *D* = [0]. Writing this out in continuous-time state-space notation yields the following.

$$\dot{\mathbf{x}} = \begin{bmatrix} -kV \\ kA \end{bmatrix} \mathbf{x} + \begin{bmatrix} 1 \\ kA \end{bmatrix} \mathbf{u}$$

$$\mathbf{y} = [1] \mathbf{x} + [0] \mathbf{u}$$

The *LinearSystem* class contains methods for easily creating state-space systems identified using *system identification*. This example shows a flywheel model with a kV of 0.023 and a kA of 0.001:

Java

C++

```

36 // Volts per (radian per second)
37 private static final double kFlywheelKv = 0.023;
38
39 // Volts per (radian per second squared)
40 private static final double kFlywheelKa = 0.001;
41

```

(continues on next page)

(continued from previous page)

```

42 // The plant holds a state-space model of our flywheel. This system has the
    following properties:
43 //
44 // States: [velocity], in radians per second.
45 // Inputs (what we can "put in"): [voltage], in volts.
46 // Outputs (what we can measure): [velocity], in radians per second.
47 //
48 // The Kv and Ka constants are found using the FRC Characterization toolsuite.
49 private final LinearSystem<N1, N1, N1> m_flywheelPlant = LinearSystemId.
    identifyVelocitySystem(
50         kFlywheelKv, kFlywheelKa);

```

```

21 #include <frc/system/plant/LinearSystemId.h>

```

```

36 // Volts per (radian per second)
37 static constexpr auto kFlywheelKv = 0.023_V / 1_rad_per_s;
38
39 // Volts per (radian per second squared)
40 static constexpr auto kFlywheelKa = 0.001_V / 1_rad_per_s_sq;
41
42 // The plant holds a state-space model of our flywheel. This system has the
43 // following properties:
44 //
45 // States: [velocity], in radians per second.
46 // Inputs (what we can "put in"): [voltage], in volts.
47 // Outputs (what we can measure): [velocity], in radians per second.
48 //
49 // The Kv and Ka constants are found using the FRC Characterization toolsuite.
50 frc::LinearSystem<1, 1, 1> m_flywheelPlant =
51     frc::LinearSystemId::IdentifyVelocitySystem<units::radian>(kFlywheelKv,

```

Modeling Using Flywheel Moment of Inertia and Gearing

A flywheel can also be modeled without access to a physical robot, using information about the motors, gearing and flywheel's *moment of inertia*. A full derivation of this model is presented in Section 8.2.1 of [Controls Engineering in FRC](#).

The `LinearSystem` class contains methods to easily create a model of a flywheel from the flywheel's motors, gearing and *moment of inertia*. The moment of inertia can be calculated using CAD software or using physics. The examples used here are detailed in the flywheel example project (Java/C++).

Note: For WPILib's state-space classes, gearing is written as output over input – that is, if the flywheel spins slower than the motors, this number should be greater than one.

Note: The C++ `LinearSystem` class uses *the C++ Units Library* to prevent unit mixups and assert dimensionality.

Java

C++

```

37 private static final double kFlywheelMomentOfInertia = 0.00032; // kg * m^2
38
39 // Reduction between motors and encoder, as output over input. If the flywheel
↳ spins slower than
40 // the motors, this number should be greater than one.
41 private static final double kFlywheelGearing = 1.0;
42
43 // The plant holds a state-space model of our flywheel. This system has the
↳ following properties:
44 //
45 // States: [velocity], in radians per second.
46 // Inputs (what we can "put in"): [voltage], in volts.
47 // Outputs (what we can measure): [velocity], in radians per second.
48 private final LinearSystem<N1, N1, N1> m_flywheelPlant = LinearSystemId.
↳ createFlywheelSystem(
49     DCMotor.getNEO(2),
50     kFlywheelMomentOfInertia,
51     kFlywheelGearing);

```

```

21 #include <frc/system/plant/LinearSystemId.h>

```

```

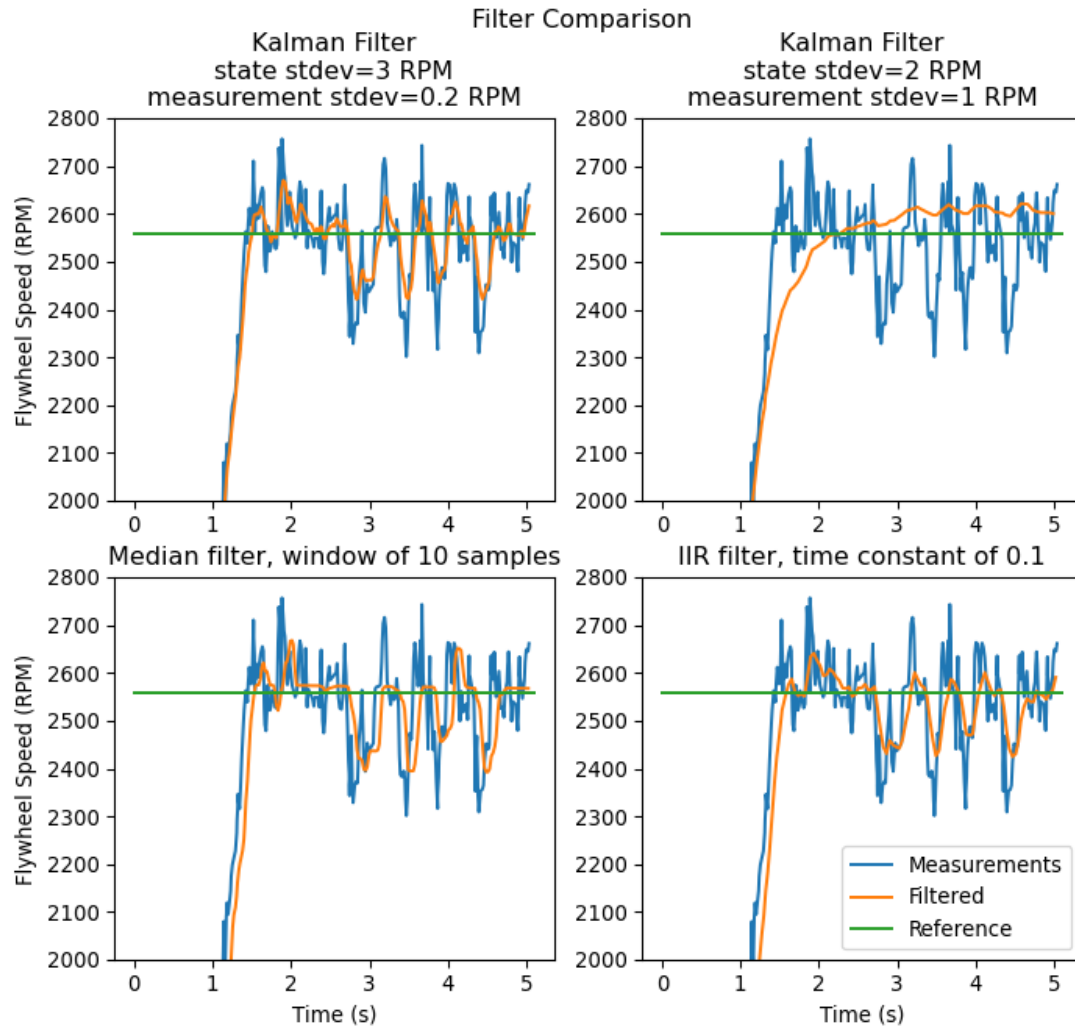
35 static constexpr units::kilogram_square_meter_t kFlywheelMomentOfInertia =
36     0.00032_kg_sq_m;
37
38 // Reduction between motors and encoder, as output over input. If the flywheel
39 // spins slower than the motors, this number should be greater than one.
40 static constexpr double kFlywheelGearing = 1.0;
41
42 // The plant holds a state-space model of our flywheel. This system has the
43 // following properties:
44 //
45 // States: [velocity], in radians per second.
46 // Inputs (what we can "put in"): [voltage], in volts.
47 // Outputs (what we can measure): [velocity], in radians per second.
48 frc::LinearSystem<1, 1, 1> m_flywheelPlant =
49     frc::LinearSystemId::FlywheelSystem(
50         frc::DCMotor::NEO(2), kFlywheelMomentOfInertia, kFlywheelGearing);

```

Kalman Filters: Observing Flywheel State

Kalman filters are used to filter our velocity measurements using our state-space model to generate a state estimate \hat{x} . As our flywheel model is linear, we can use a Kalman filter to estimate the flywheel's velocity. WPILib's Kalman filter takes a `LinearSystem` (which we found above), along with standard deviations of model and sensor measurements. We can adjust how "smooth" our state estimate is by adjusting these weights. Larger state standard deviations will cause the filter to "distrust" our state estimate and favor new measurements more highly, while larger measurement standard deviations will do the opposite.

In the case of a flywheel we start with a state standard deviation of 3 rad/s and a measurement standard deviation of 0.01 rad/s. These values are up to the user to choose – these weights produced a filter that was tolerant to some noise but whose state estimate quickly reacted to external disturbances for a flywheel – and should be tuned to create a filter that behaves well for your specific flywheel. Graphing states, measurements, inputs, references, and outputs over time is a great visual way to tune Kalman filters.



The above graph shows two differently tuned Kalman filters, as well as a *single-pole IIR filter* and a *Median Filter*. This data was collected with a shooter over ~5 seconds, and four balls were run through the shooter (as seen in the four dips in velocity). While there are no hard rules on choosing good state and measurement standard deviations, they should in general be tuned to trust the model enough to reject noise while reacting quickly to external disturbances.

Because the feedback controller computes error using the *x-hat* estimated by the Kalman filter, the controller will react to disturbances only as quickly the filter's state estimate changes. In the above chart, the upper left plot (with a state standard deviation of 3.0 and measurement standard deviation of 0.2) produced a filter that reacted quickly to disturbances while rejecting noise, while the upper right plot shows a filter that was barely affected by the velocity dips.

Java

C++

```
private final KalmanFilter<N1, N1, N1> m_observer = new KalmanFilter<>(
    Nat.N1(), Nat.N1(),
```

(continues on next page)

(continued from previous page)

```

56     m_flywheelPlant,
57     VecBuilder.fill(3.0), // How accurate we think our model is
58     VecBuilder.fill(0.01), // How accurate we think our encoder
59     // data is
60     0.020);

```

```

18 #include <frc/estimator/KalmanFilter.h>

```

```

52 // The observer fuses our encoder data and voltage inputs to reject noise.
53 frc::KalmanFilter<1, 1, 1> m_observer{
54     m_flywheelPlant,
55     {3.0}, // How accurate we think our model is
56     {0.01}, // How accurate we think our encoder data is
57     20_ms};

```

Because Kalman filters use our state-space model in the *Predict step*, it is important that our model is as accurate as possible. One way to verify this is to record a flywheel's input voltage and velocity over time, and replay this data by calling only *predict* on the Kalman filter. Then, the kV and kA gains (or moment of inertia and other constants) can be adjusted until the model closely matches the recorded data.

Linear-Quadratic Regulators and Plant Inversion Feedforward

The *Linear-Quadratic Regulator* finds a feedback controller to drive our flywheel *system* to its *reference*. Because our flywheel has just one state, the control law picked by our LQR will be in the form $\mathbf{u} = \mathbf{K}(\mathbf{r} - \mathbf{x})$ where \mathbf{K} is a 1x1 matrix; in other words, the control law picked by LQR is simply a proportional controller, or a PID controller with only a P gain. This gain is chosen by our LQR based on the state excursion and control efforts we pass it. More on tuning LQR controllers can be found in the *LQR application example*.

Much like SimpleMotorFeedforward can be used to generate feedforward voltage inputs given kS, kV, and kA constants, the Plant Inversion Feedforward class generate *feedforward* voltage inputs given a state-space system. The voltage commands generated by the LinearSystemLoop class are the sum of the feedforward and feedback inputs.

Java

C++

```

63 private final LinearQuadraticRegulator<N1, N1, N1> m_controller
64     = new LinearQuadraticRegulator<>(m_flywheelPlant,
65     VecBuilder.fill(8.0), // qelms. Velocity error tolerance, in radians per
↪second. Decrease
66     // this to more heavily penalize state excursion, or make the controller
↪behave more
67     // aggressively.
68     VecBuilder.fill(12.0), // relms. Control effort (voltage) tolerance. Decrease
↪this to more
69     // heavily penalize control effort, or make the controller less aggressive.
↪12 is a good
70     // starting point because that is the (approximate) maximum voltage of a
↪battery.
71     0.020); // Nominal time between loops. 0.020 for TimedRobot, but can be
72     // lower if using notifiers.

```

```

15 #include <frc/controller/LinearPlantInversionFeedforward.h>
16 #include <frc/controller/LinearQuadraticRegulator.h>

59 // A LQR uses feedback to create voltage commands.
60 frc::LinearQuadraticRegulator<1, 1> m_controller{
61     m_flywheelPlant,
62     // qelms. Velocity error tolerance, in radians per second. Decrease this
63     // to more heavily penalize state excursion, or make the controller behave
64     // more aggressively.
65     {8.0},
66     // relms. Control effort (voltage) tolerance. Decrease this to more
67     // heavily penalize control effort, or make the controller less
68     // aggressive. 12 is a good starting point because that is the
69     // (approximate) maximum voltage of a battery.
70     {12.0},
71     // Nominal time between loops. 20ms for TimedRobot, but can be lower if
72     // using notifiers.
73     20_ms};
74
75 // The state-space loop combines a controller, observer, feedforward and plant
76 // for easy control.
77 frc::LinearSystemLoop<1, 1, 1> m_loop{m_flywheelPlant, m_controller,
78                                         m_observer, 12_V, 20_ms};

```

Bringing it All Together: LinearSystemLoop

LinearSystemLoop combines our system, controller, and observer that we created earlier. The constructor shown will also instantiate a PlantInversionFeedforward.

Java

C++

```

73 // The state-space loop combines a controller, observer, feedforward and plant for
74 // easy control.
75 private final LinearSystemLoop<N1, N1, N1> m_loop = new LinearSystemLoop<>{
76     m_flywheelPlant,
77     m_controller,
78     m_observer,
79     12.0,

```

```

19 #include <frc/system/LinearSystemLoop.h>

```

```

75 // The state-space loop combines a controller, observer, feedforward and plant
76 // for easy control.
77 frc::LinearSystemLoop<1, 1, 1> m_loop{m_flywheelPlant, m_controller,
78                                         m_observer, 12_V, 20_ms};

```

Once we have our LinearSystemLoop, the only thing left to do is actually run it. To do that, we'll periodically update our Kalman filter with our new encoder velocity measurements and apply new voltage commands to it. To do that, we first set the *reference*, then correct with the current flywheel speed, predict the Kalman filter into the next timestep, and apply the inputs generated using getU.

Java

C++

```

101  @Override
102  public void teleopPeriodic() {
103      // Sets the target speed of our flywheel. This is similar to setting the setpoint
104      // of a PID controller.
105      if (m_joystick.getTriggerPressed()) {
106          // We just pressed the trigger, so let's set our next reference
107          m_loop.setNextR(VecBuilder.fill(kSpinupRadPerSec));
108      } else if (m_joystick.getTriggerReleased()) {
109          // We just released the trigger, so let's spin down
110          m_loop.setNextR(VecBuilder.fill(0.0));
111      }
112
113      // Correct our Kalman filter's state vector estimate with encoder data.
114      m_loop.correct(VecBuilder.fill(m_encoder.getRate()));
115
116      // Update our LQR to generate new voltage commands and use the voltages to
117      // predict the next state with out Kalman filter.
118      m_loop.predict(0.020);
119
120      // Send the new calculated voltage to the motors.
121      // voltage = duty cycle * battery voltage, so
122      // duty cycle = voltage / battery voltage
123      double nextVoltage = m_loop.getU(0);
124      m_motor.setVoltage(nextVoltage);
125  }
126  }

```

```

21  #include <frc/DriverStation.h>
22  #include <frc/Encoder.h>
23  #include <frc/GenericHID.h>
24  #include <frc/PWMPWMVictorSPX.h>
25  #include <frc/StateSpaceUtil.h>
26  #include <frc/TimedRobot.h>
27  #include <frc/XboxController.h>
28  #include <frc/controller/LinearPlantInversionFeedforward.h>
29  #include <frc/controller/LinearQuadraticRegulator.h>
30  #include <frc/drive/DifferentialDrive.h>
31  #include <frc/estimator/KalmanFilter.h>
32  #include <frc/system/LinearSystemLoop.h>
33  #include <frc/system/plant/DCMotor.h>
34  #include <frc/system/plant/LinearSystemId.h>
35  #include <wpi/math>

```

```

96  void TeleopPeriodic() {
97      // Sets the target speed of our flywheel. This is similar to setting the
98      // setpoint of a PID controller.
99      if (m_joystick.GetBumper(frc::GenericHID::kRightHand)) {
100          // We pressed the bumper, so let's set our next reference
101          m_loop.SetNextR(frc::MakeMatrix<1, 1>(kSpinup.to<double>()));
102      } else {
103          // We released the bumper, so let's spin down
104          m_loop.SetNextR(frc::MakeMatrix<1, 1>(0.0));
105      }

```

(continues on next page)

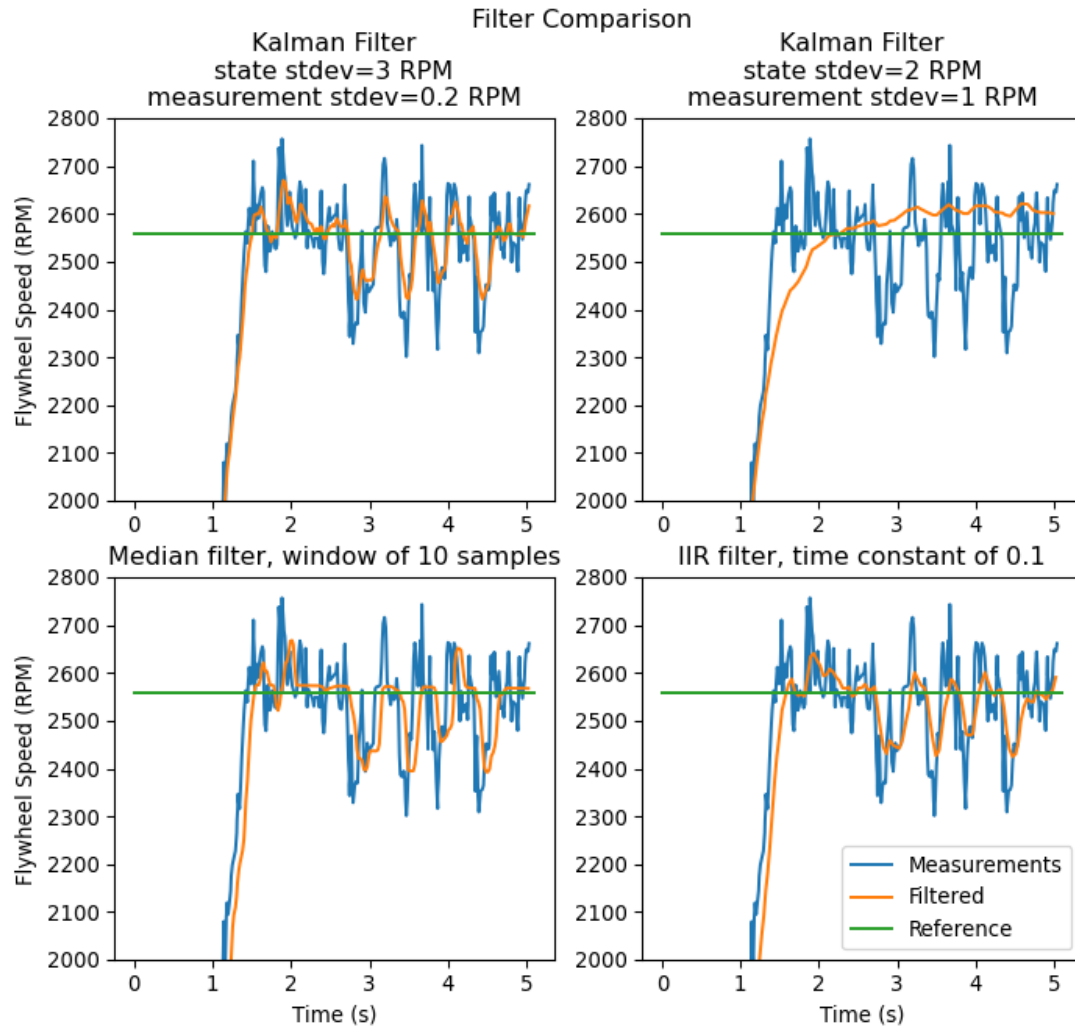
(continued from previous page)

```
106 // Correct our Kalman filter's state vector estimate with encoder data.
107 m_loop.Correct(frc::MakeMatrix<1, 1>(m_encoder.GetRate()));
108
109 // Update our LQR to generate new voltage commands and use the voltages to
110 // predict the next state with out Kalman filter.
111 m_loop.Predict(20_ms);
112
113 // Send the new calculated voltage to the motors.
114 // voltage = duty cycle * battery voltage, so
115 // duty cycle = voltage / battery voltage
116 m_motor.SetVoltage(units::volt_t(m_loop.U(0)));
117 }
118 };
119
```

35.7.3 State Observers and Kalman Filters

State observers combine information about a system's behavior and external measurements to estimate the true *state* of the system. A common observer used for linear systems is the Kalman Filter. Kalman filters are advantageous over other *filters* as they fuse measurements from one or more sensors with a state-space model of the system to optimally estimate a system's state.

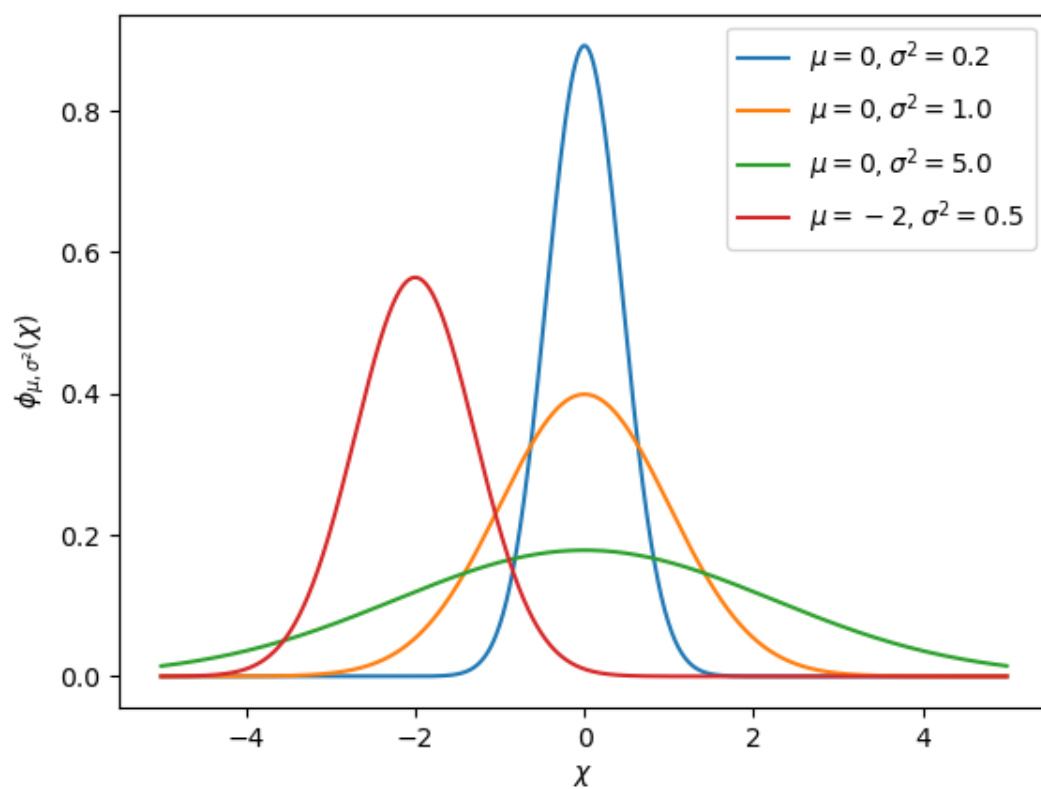
This image shows flywheel velocity measurements over time, run through a variety of different filters. Note that a well-tuned Kalman filter shows no measurement lag during flywheel spinup while still rejecting noisy data and reacting quickly to disturbances as balls pass through it. More on filters can be found in the *filters section*.



Gaussian Functions

Kalman filters utilize **Gaussian distributions** (or bell curves) to model the noise in a process. The graph of a Gaussian function is a “bell curve” shape. This function is described by its mean (the location of the “peak” of the bell curve) and variance (a measure of how “spread out” the bell curve is). In the case of a Kalman filter, the estimated *state* of the system is the mean, while the variance is a measure of how certain (or uncertain) the filter is about the true *state*.

The idea of variance and covariance is central to the function of a Kalman filter. Covariance is a measurement of how two random variables are correlated. In a system with a single state, the covariance matrix is simply $\text{cov}(\mathbf{x}_1, \mathbf{x}_1)$, or a matrix containing the variance $\text{var}(\mathbf{x}_1)$ of the state x_1 . The magnitude of this variance is the square of the standard deviation of the Gaussian function describing the current state estimate. Relatively large values for covariance might indicate noisy data, while small covariances might indicate that the filter is more confident about its estimate. Remember that “large” and “small” values for variance or covariance are relative to the base unit being used – for example, if \mathbf{x}_1 was measured in meters, $\text{cov}(\mathbf{x}_1, \mathbf{x}_1)$ would be in meters squared.



Covariance matrices are written in the following form:

$$\Sigma = \begin{bmatrix} \text{COV}(x_1, x_1) & \text{COV}(x_1, x_2) & \dots & \text{COV}(x_1, x_n) \\ \text{COV}(x_2, x_1) & \text{COV}(x_2, x_2) & \dots & \text{COV}(x_2, x_n) \\ \vdots & \vdots & \ddots & \vdots \\ \text{COV}(x_n, x_1) & \text{COV}(x_n, x_2) & \dots & \text{COV}(x_n, x_n) \end{bmatrix}$$

Kalman Filters

Important: It is important to develop an intuition for what a Kalman filter is actually doing. The book [Kalman and Bayesian Filters in Python by Roger Labbe](#) provides a great visual and interactive introduction to Bayesian filters. The Kalman filters in WPILib use linear algebra to gentrify the math, but the ideas are similar to the single-dimensional case. We suggest reading through Chapter 4 to gain an intuition for what these filters are doing.

To summarize, Kalman filters (and all Bayesian filters) have two parts: prediction and correction. Prediction projects our state estimate forward in time according to our system's dynamics, and correct steers the estimated state towards the measured state. While filters often perform both in the same timestep, it's not strictly necessary - For example, WPILib's pose estimators call predict frequently, and correct only when new measurement data is available (for example, from a low-framerate vision system).

The following shows the equations of a discrete-time Kalman filter:

Predict step

$$\hat{\mathbf{x}}_{k+1}^- = \mathbf{A}\hat{\mathbf{x}}_k + \mathbf{B}\mathbf{u}_k$$

$$\mathbf{P}_{k+1}^- = \mathbf{A}\mathbf{P}_k^- \mathbf{A}^T + \Sigma \mathbf{Q} \Sigma^T$$

Update step

$$\mathbf{K}_{k+1} = \mathbf{P}_{k+1}^- \mathbf{C}^T (\mathbf{C} \mathbf{P}_{k+1}^- \mathbf{C}^T + \mathbf{R})^{-1}$$

$$\hat{\mathbf{x}}_{k+1}^+ = \hat{\mathbf{x}}_{k+1}^- + \mathbf{K}_{k+1} (\mathbf{y}_{k+1} - \mathbf{C} \hat{\mathbf{x}}_{k+1}^- - \mathbf{D} \mathbf{u}_{k+1})$$

$$\mathbf{P}_{k+1}^+ = (\mathbf{I} - \mathbf{K}_{k+1} \mathbf{C}) \mathbf{P}_{k+1}^-$$

A	system matrix	$\hat{\mathbf{x}}$	state estimate vector
B	input matrix	\mathbf{u}	input vector
C	output matrix	\mathbf{y}	output vector
D	feedthrough matrix	Σ	process noise intensity vector
P	error covariance matrix	Q	process noise covariance matrix
K	Kalman gain matrix	R	measurement noise covariance matrix

The state estimate \mathbf{x} , together with \mathbf{P} , describe the mean and covariance of the Gaussian function that describes our filter's estimate of the system's true state.

Process and Measurement Noise Covariance Matrices

The process and measurement noise covariance matrices **Q** and **R** describe the variance of each of our states and measurements. Remember that for a Gaussian function, variance is the square of the function's standard deviation. In a WPILib, **Q** and **R** are diagonal matrices whose diagonals contain their respective variances. For example, a Kalman filter with states $\begin{bmatrix} \text{position} \\ \text{velocity} \end{bmatrix}$ and measurements $\begin{bmatrix} \text{position} \end{bmatrix}$ with state standard deviations $\begin{bmatrix} 0.1 \\ 1.0 \end{bmatrix}$ and measurement standard deviation $[0.01]$ would have the following **Q** and **R** matrices:

$$Q = \begin{bmatrix} 0.01 & 0 \\ 0 & 1.0 \end{bmatrix}, R = [0.0001]$$

Error Covariance Matrix

The error covariance matrix **P** describes the covariance of the state estimate $\hat{\mathbf{x}}$. Informally, **P** describes our certainty about the estimated *state*. If **P** is large, our uncertainty about the true state is large. Conversely, a **P** with smaller elements would imply less uncertainty about our true state.

As we project the model forward, **P** increases as our certainty about the system's true state decreases.

Predict step

In prediction, our state estimate is updated according to the linear system dynamics $\mathbf{x} = \mathbf{Ax} + \mathbf{Bu}$. Furthermore, our error covariance **P** increases by the process noise covariance matrix **Q**. Larger values of **Q** will make our error covariance **P** grow more quickly. This **P** is used in the correction step to weight the model and measurements.

Correct step

In the correct step, our state estimate is updated to include new measurement information. This new information is weighted against the state estimate $\hat{\mathbf{x}}$ by the Kalman gain **K**. Large values of **K** more highly weight incoming measurements, while smaller values of **K** more highly weight our state prediction. Because **K** is related to **P**, larger values of **P** will increase **K** and more heavily weight measurements. If, for example, a filter is predicted for a long duration, the large **P** would heavily weight the new information.

Finally, the error covariance **P** decreases to increase our confidence in the state estimate.

Tuning Kalman Filters

WPILib's Kalman Filter classes' constructors take a linear system, a vector of process noise standard deviations and measurement noise standard deviations. These are converted to **Q** and **R** matrices by filling the diagonals with the square of the standard deviations, or variances, of each state or measurement. By decreasing a state's standard deviation (and therefore its corresponding entry in **Q**), the filter will distrust incoming measurements more. Similarly, increasing a state's standard deviation will trust incoming measurements more. The same holds for the measurement standard deviations - decreasing an entry will make the filter more highly trust the incoming measurement for the corresponding state, while increasing it will decrease trust in the measurement.

Java

C++

```

54 private final KalmanFilter<N1, N1, N1> m_observer = new KalmanFilter<>(
55     Nat.N1(), Nat.N1(),
56     m_flywheelPlant,
57     VecBuilder.fill(3.0), // How accurate we think our model is
58     VecBuilder.fill(0.01), // How accurate we think our encoder
59     // data is
60     0.020);

```

```

8  #include <frc/DriverStation.h>
9  #include <frc/Encoder.h>
10 #include <frc/GenericHID.h>
11 #include <frc/PWMMotorSPX.h>
12 #include <frc/StateSpaceUtil.h>
13 #include <frc/TimedRobot.h>
14 #include <frc/XboxController.h>
15 #include <frc/controller/LinearQuadraticRegulator.h>
16 #include <frc/drive/DifferentialDrive.h>
17 #include <frc/estimator/KalmanFilter.h>
18 #include <frc/system/LinearSystemLoop.h>
19 #include <frc/system/plant/DCMotor.h>
20 #include <frc/system/plant/LinearSystemId.h>
21 #include <units/angular_velocity.h>
22 #include <wpi/math>

```

```

52 // The observer fuses our encoder data and voltage inputs to reject noise.
53 frc::KalmanFilter<1, 1, 1> m_observer{
54     m_flywheelPlant,
55     {3.0}, // How accurate we think our model is
56     {0.01}, // How accurate we think our encoder data is
57     20_ms};

```

35.7.4 WPILib Pose Estimators

WPILib includes pose estimators for differential, swerve and mecanum drivetrains. These estimators are designed to be drop-in replacements for the existing *odometry* classes, with added features that utilize an Unscented *Kalman Filter* to fuse latency-compensated robot pose estimates with encoder and gyro measurements. These estimators can account for encoder drift and noisy vision data. These estimators can behave identically to their corresponding odometry classes if only `update` is called on these estimators.

Pose estimators estimate robot position using a state-space system with the states $[x \ y \ \theta]^T$, which can represent robot position as a `Pose2d`. WPILib includes `DifferentialDrivePoseEstimator`, `SwerveDrivePoseEstimator` and `MecanumDrivePoseEstimator` to estimate robot position. In these, users call `update` periodically with encoder and gyro measurements (same as the odometry classes) to update the robot's estimated position. When the robot receives measurements of its field-relative position (encoded as a `Pose2d`) from sensors such as computer vision or V-SLAM, the pose estimator latency-compensates the measurement to accurately estimate robot position.

The pose estimators perform latency compensation by storing a list of past observer states, including estimated state $\hat{\mathbf{x}}$, error covariance \mathbf{P} , inputs and local measurements. When new measurements are applied, the state of the estimator is first rolled back to the measurement's timestamp. Then, the filter corrects its state estimate with the new measurement and applies the inputs between the measurement timestamp and the present time to incorporate the new measurement. This allows for vision solutions with framerates which might otherwise make them unusable be a viable solution for robot localization.

The following example shows the use of the `DifferentialDrivePoseEstimator`:

Java

C++

```
var estimator = new DifferentialDrivePoseEstimator(new Rotation2d(), new Pose2d(),
    new MatBuilder<>(Nat.N5(), Nat.N1()).fill(0.02, 0.02, 0.01, 0.02, 0.02), //
    ↳ State measurement standard deviations. X, Y, theta.
    new MatBuilder<>(Nat.N3(), Nat.N1()).fill(0.02, 0.02, 0.01), // Local
    ↳ measurement standard deviations. Left encoder, right encoder, gyro.
    new MatBuilder<>(Nat.N3(), Nat.N1()).fill(0.1, 0.1, 0.01)); // Global
    ↳ measurement standard deviations. X, Y, and theta.
```

```
#include "frc/estimator/DifferentialDrivePoseEstimator.h"
#include "frc/StateSpaceUtil.h"

frc::DifferentialDrivePoseEstimator estimator{
    frc::Rotation2d(), frc::Pose2d(),
    frc::MakeMatrix<5, 1>(0.01, 0.01, 0.01, 0.01, 0.01),
    frc::MakeMatrix<3, 1>(0.1, 0.1, 0.1),
    frc::MakeMatrix<3, 1>(0.1, 0.1, 0.1)};
```

Tuning Pose Estimators

All pose estimators offer user-customizable standard deviations for model and measurements. These standard deviations determine how much the filter “trusts” each of these states. For example, increasing the standard deviation for measurements (as one might do for a noisy signal) would lead to the estimator trusting its state estimate more than the incoming measurements. On the field, this might mean that the filter can reject noisy vision data well, at the cost of being slow to correct for model deviations. While these values can be estimated beforehand, they very much depend on the unique setup of each robot and global measurement method.

35.7.5 Debugging State-Space Models and Controllers

Checking Signs

One of the most common causes of bugs with state-space controllers is signs being flipped. For example, models included in WPILib expect positive voltage to result in a positive acceleration, and vice versa. If applying a positive voltage does not make the mechanism accelerate forwards, or if moving “forwards” makes encoder (or other sensor readings) decrease, they should be inverted so that positive voltage input results in a positive encoder reading. For example, if I apply an *input* of $[12, 12]^T$ (full forwards for the left and right motors) to my differential drivetrain, my wheels should propel my robot “forwards” (along the +X axis locally), and for my encoders to read a positive velocity.

Important: The WPILib DifferentialDrive by default inverts the right motors. This behavior can be changed by calling `setRightSideInverted(false)/SetRightSideInverted(false)` (Java/C++) on the DifferentialDrive object.

The Importance of Graphs

Reliable data of the *system’s states*, *inputs* and *outputs* over time is important when debugging state-space controllers and observers. One common approach is to send this data over NetworkTables and use tools such as *Shuffleboard*, which allow us to both graph the data in real-time as well as save it to a CSV file for plotting later with tools such as Google Sheets, Excel or Python.

Note: By default, NetworkTables is limited to a 10hz update rate. For testing, this can be bypassed with the following code snippet to submit data at up to 100hz. This code should be run periodically to forcibly publish new data.

Danger: This will send extra data (at up to 100hz) over NetworkTables, which can cause lag with both user code and robot dashboards. This will also increase network utilization. It is often a good idea to disable this during competitions.

Java

C++

```
@Override
public void robotPeriodic() {
    NetworkTableInstance.getDefault().flush();
}
```

```
void RobotPeriodic() {
    NetworkTableInstance::GetDefault().Flush();
}
```

Compensating for Input Lag

Often times, some sensor input data (i.e. velocity readings) may be delayed due to onboard filtering that smart motor controllers tend to perform. By default, LQR's K gain assumes no input delay, so introducing significant delay on the order of tens of milliseconds can cause instability. To combat this, the LQR's K gain can be reduced, trading off performance for stability. A code example for how to compensate for this latency in a mathematically rigorous manner is available [here](#).

35.8 Controls Glossary

control effort A term describing how much force, pressure, etc. an actuator is exerting.

control input The input of a *plant* used for the purpose of controlling it

control law A mathematical formula that generates *inputs* to drive a *system* to a desired *state*, given the current *state*. A common example is the control law $\mathbf{u} = \mathbf{K}(\mathbf{r} - \mathbf{x})$

controller Used in position or negative feedback with a *plant* to bring about a desired *system state* by driving the difference between a *reference* signal and the *output* to zero.

dynamics A branch of physics concerned with the motion of bodies under the action of forces. In modern control, systems evolve according to their dynamics.

error *Reference* minus an *output* or *state*.

gain A proportional value that relates the magnitude of an input signal to the magnitude of an output signal. In the signal-dimensional case, gain can be thought of as the proportional term of a PID controller. A gain greater than one would amplify an input signal, while a gain less than one would dampen an input signal. A negative gain would negate the input signal.

hidden state A *state* that cannot be directly measured, but whose *dynamics* can be related to other states.

input An input to the *plant* (hence the name) that can be used to change the *plant's state*.

- Ex. A flywheel will have 1 input: the voltage of the motor driving it.
- Ex. A drivetrain might have 2 inputs: the voltages of the left and right motors.

Inputs are often represented by the variable \mathbf{u} , a column vector with one entry per *input* to the *system*.

measurement Measurements are *outputs* that are measured from a *plant*, or physical system, using sensors.

model A set of mathematical equations that reflects some aspect of a physical *system's* behavior.

moment of inertia A measurement of a rotating body's resistance to angular acceleration or deceleration. Angular moment of inertia can be thought of as angular mass. See also: [Moment of inertia](#).

observer In control theory, a system that provides an estimate of the internal *state* of a given real *system* from measurements of the *input* and *output* of the real *system*. WPILib includes a Kalman Filter class for observing linear systems, and ExtendedKalmanFilter and UnscentedKalmanFilter classes for nonlinear systems.

output Measurements from sensors. There can be more measurements than states. These outputs are used in the "correct" step of Kalman Filters.

- Ex. A flywheel might have 1 *output* from an encoder that measures its velocity.
- Ex. A drivetrain might use solvePNP and V-SLAM to find its x/y/heading position on the field. It's fine that there are 6 measurements (solvePNP x/y/heading and V-SLAM x/y/heading) and 3 states (robot x/y/heading).

Outputs of a *system* are often represented using the variable \mathbf{y} , a column vector with one entry per *output* (or thing we can measure). For example, if our *system* had states for velocity and acceleration but our sensor could only measure velocity, our *output* vector would only include the *system's* velocity.

plant The *system* or collection of actuators being controlled.

process variable The term used to describe the output of a *plant* in the context of PID control.

reference The desired state. This value is used as the reference point for a controller's error calculation.

rise time The time a *system* takes to initially reach the *reference* after applying a *step input*.

setpoint The term used to describe the *reference* of a PID controller.

settling time The time a *system* takes to settle at the *reference* after a *step input* is applied.

state A characteristic of a *system* (e.g., velocity) that can be used to determine the *system's* future behavior. In state-space notation, the state of a system is written as a column vector describing its position in state-space.

- Ex. A drivetrain system might have the states $\begin{bmatrix} x \\ y \\ \theta \end{bmatrix}$ to describe its position on the field.
- Ex. An elevator system might have the states $\begin{bmatrix} \text{position} \\ \text{velocity} \end{bmatrix}$ to describe its current height and velocity.

A *system's* state is often represented by the variable \mathbf{x} , a column vector with one entry per *state*.

steady-state error *Error* after *system* reaches equilibrium.

step input A *system input* that is 0 for $t < 0$ and a constant greater than 0 for $t \geq 0$. A step input that is 1 for $t \geq 0$ is called a unit step input.

step response The response of a *system* to a *step input*.

system A term encompassing a *plant* and its interaction with a *controller* and *observer*, which is treated as a single entity. Mathematically speaking, a *system* maps *inputs* to *outputs* through a linear combination of *states*.

system identification The process of capturing a *systems dynamics* in a mathematical model using measured data. The characterization toolsuite uses system identification to find kS, kV and kA terms.

system response The behavior of a *system* over time for a given *input*.

x-dot $\dot{\mathbf{x}}$, or x-dot: the derivative of the *state* vector \mathbf{x} . If the *system* had just a velocity *state*, then $\dot{\mathbf{x}}$ would represent the *system*'s acceleration.

x-hat $\hat{\mathbf{x}}$, or x-hat: the estimated *state* of a system, as estimated by an *observer*.

Convenience Features

This section covers some general convenience features that be used with other advanced programming features.

36.1 Scheduling Functions at Custom Frequencies

TimedRobot's `addPeriodic()` method allows one to run custom methods at a rate faster than the default TimedRobot periodic update rate (20 ms). Previously, teams had to make a `Notifier` to run feedback controllers more often than the TimedRobot loop period of 20 ms (running TimedRobot more often than this is not advised). Now, users can run feedback controllers more often than the main robot loop, but synchronously with the TimedRobot periodic functions, eliminating potential thread safety issues.

The `addPeriodic()` (Java) / `AddPeriodic()` (C++) method takes in a lambda (the function to run), along with the requested period and an optional offset from the common starting time. The optional third argument is useful for scheduling a function in a different timeslot relative to the other TimedRobot periodic methods.

Note: The units for the period and offset are seconds in Java. In C++, the *units library* can be used to specify a period and offset in any time unit.

Java

C++ (Header)

C++ (Source)

```
public class Robot extends TimedRobot {
    private Joystick m_joystick = new Joystick(0);
    private Encoder m_encoder = new Encoder(1, 2);
    private Spark m_motor = new Spark(1);
    private PIDController m_controller = new PIDController(1.0, 0.0, 0.5, 0.01);

    public Robot() {
        addPeriodic(() -> {
            m_motor.set(m_controller.calculate(m_encoder.getRate()));
        }, 0.02, 0.0);
    }
}
```

(continues on next page)

(continued from previous page)

```

    }, 0.01, 0.005);
}

@Override
public teleopPeriodic() {
    if (m_joystick.getRawButtonPressed(1)) {
        if (m_controller.getSetpoint() == 0.0) {
            m_controller.setSetpoint(30.0);
        } else {
            m_controller.setSetpoint(0.0);
        }
    }
}
}

```

```

class Robot : public frc::TimedRobot {
private:
    frc::Joystick m_joystick{0};
    frc::Encoder m_encoder{1, 2};
    frc::Spark m_motor{1};
    frc2::PIDController m_controller{1.0, 0.0, 0.5, 10_ms};

    Robot();

    void TeleopPeriodic() override;
};

```

```

void Robot::Robot() {
    AddPeriodic([&] {
        m_motor.Set(m_controller.Calculate(m_encoder.GetRate()));
    }, 10_ms, 5_ms);
}

void Robot::TeleopPeriodic() {
    if (m_joystick.GetRawButtonPressed(1)) {
        if (m_controller.GetSetpoint() == 0.0) {
            m_controller.SetSetpoint(30.0);
        } else {
            m_controller.SetSetpoint(0.0);
        }
    }
}
}

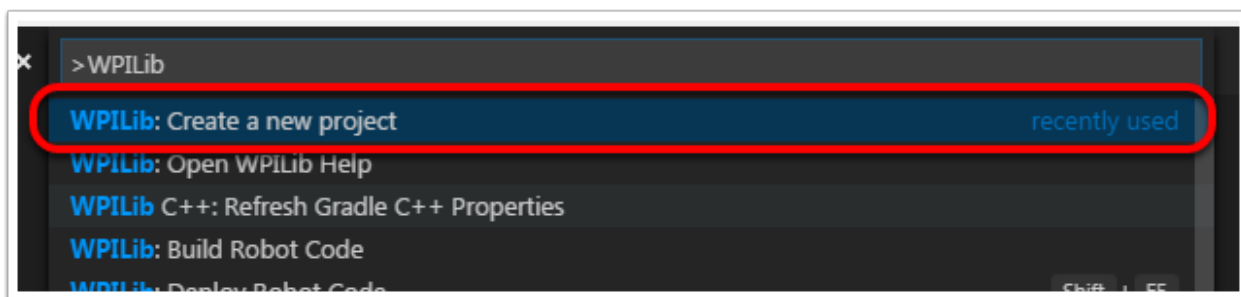
```

The `teleopPeriodic()` method in this example runs every 20 ms, and the controller update is run every 10 ms with an offset of 5 ms from when `teleopPeriodic()` runs so that their time-slots don't conflict (e.g., `teleopPeriodic()` runs at 0 ms, 20 ms, 40 ms, etc.; the controller runs at 5 ms, 15 ms, 25 ms, etc.).

WPILib Example Projects

Warning: While every attempt is made to keep WPILib examples functional, they are *not* intended to be used “as-is.” At the very least, robot-specific constants will need to be changed for the code to work on a user robot. Many empirical constants have their values “faked” for demonstration purposes. Users are strongly encouraged to write their own code (from scratch or from an existing template) rather than copy example code.

WPILib example projects demonstrate a large number of library features and use patterns. Projects range from simple demonstrations of a single functionality to complete, competition-capable robot programs. All of these examples are available in VS Code by entering Ctrl+Shift+P, then selecting *WPILib: Create a new project* and choosing example.



37.1 Basic Examples

These examples demonstrate basic/minimal robot functionality. They are useful for beginning teams who are gaining initial familiarity with robot programming, but are highly limited in functionality.

- **Arcade Drive (Java, C++):** Demonstrates a simple differential drive implementation using “arcade”-style controls through the `DifferentialDrive` class.
- **Arcade Drive Xbox Controller (Java, C++):** Demonstrates the same functionality seen in the previous example, except using an `XboxController` instead of an ordinary joystick.

- **Getting Started (Java, C++)**: Demonstrates a simple autonomous routine that drives forwards for two seconds at half speed.
- **Mecanum Drive (Java, C++)**: Demonstrates a simple mecanum drive implementation using the `MecanumDrive` class.
- **Motor Controller (Java, C++)**: Demonstrates how to control the output of a motor with a joystick.
- **Motor Control With Encoder (Java, C++)**: Identical to the above example, except with the addition of an encoder to read the motor position.
- **Simple Vision (Java, C++)**: Demonstrates how to stream video from a USB camera to the dashboard.
- **Relay (Java, C++)**: Demonstrates the use of the `Relay` class to control a relay output with a set of joystick buttons.
- **Solenoids (Java, C++)**: Demonstrates the use of the `Solenoid` and `DoubleSolenoid` classes to control solenoid outputs with a set of joystick buttons.
- **TankDrive (Java)**: Demonstrates a simple differential drive implementation using “tank”-style controls through the `DifferentialDrive` class.
- **Tank Drive Xbox Controller (Java, C++)**: Demonstrates the same functionality seen in the previous example, except using an `XboxController` instead of an ordinary joystick.

37.2 Control Examples

These examples demonstrate WPILib implementations of common robot controls. Sensors may be present, but are not the emphasized concept of these examples.

- **DifferentialDriveBot (Java, C++)**: Demonstrates an advanced differential drive implementation, including encoder-and-gyro odometry through the `DifferentialDriveOdometry` class, and composition with PID velocity control through the `DifferentialDriveKinematics` and `PIDController` classes.
- **Elevator with profiled PID controller (Java, C++)**: Demonstrates the use of the `ProfiledPIDController` class to control the position of an elevator mechanism.
- **Elevator with trapezoid profiled PID (Java, C++)**: Demonstrates the use of the `TrapezoidProfile` class in conjunction with a “smart motor controller” to control the position of an elevator mechanism.
- **Gyro Mecanum (Java, C++)**: Demonstrates field-oriented control of a mecanum robot through the `MecanumDrive` class in conjunction with a gyro.
- **MecanumBot (Java, C++)**: Demonstrates an advanced mecanum drive implementation, including encoder-and-gyro odometry through the `MecanumDriveOdometry` class, and composition with PID velocity control through the `MecanumDriveKinematics` and `PIDController` classes.
- **PotentiometerPID (Java, C++)**: Demonstrates the use of the `PIDController` class and a potentiometer to control the position of an elevator mechanism.
- **RamseteController (Java, C++)**: Demonstrates the use of the `RamseteController` class to follow a trajectory during the autonomous period.

- **SwerveBot (Java, C++)**: Demonstrates an advanced swerve drive implementation, including encoder-and-gyro odometry through the `SwerveDriveOdometry` class, and composition with PID position and velocity control through the `SwerveDriveKinematics` and `PIDController` classes.
- **UltrasonicPID (Java, C++)**: Demonstrates the use of the `PIDController` class in conjunction with an ultrasonic sensor to drive to a set distance from an object.

37.3 Sensor Examples

These examples demonstrate sensor reading and data processing using WPILib. Mechanisms control may be present, but is not the emphasized concept of these examples.

- **Axis Camera Sample (Java, C++)**: Demonstrates the use of OpenCV and an Axis Net-cam to overlay a rectangle on a captured video feed and stream it to the dashboard.
- **PDP CAN Monitoring (Java, C++)**: Demonstrates obtaining sensor information from the PDP over CAN using the `PowerDistributionPanel` class.
- **Duty Cycle Encoder (Java, C++)**: Demonstrates the use of the `DutyCycleEncoder` class to read values from a PWM-type absolute encoder.
- **DutyCycleInput (Java, C++)**: Demonstrates the use of the `DutyCycleInput` class to read the frequency and fractional duty cycle of a PWM input.
- **Encoder (Java, C++)**: Demonstrates the use of the `Encoder` class to read values from a quadrature encoder.
- **Gyro (Java, C++)**: Demonstrates the use of the `AnalogGyro` class to measure robot heading and stabilize driving.
- **Intermediate Vision (Java, C++)**: Demonstrates the use of OpenCV and a USB camera to overlay a rectangle on a captured video feed and stream it to the dashboard.
- **Ultrasonic (Java, C++)**: Demonstrates the use of the `Ultrasonic` class to read data from an ultrasonic sensor in conjunction with the `MedianFilter` class to reduce signal noise.

37.4 Command-Based Examples

These examples demonstrate the use of the *Command-Based framework*.

- **ArmBot (Java, C++)**: Demonstrates the use of a `ProfiledPIDSubsystem` to control a robot arm.
- **ArmBotOffboard (Java, C++)**: Demonstrates the use of a `TrapezoidProfileSubsystem` in conjunction with a “smart motor controller” to control a robot arm.
- **DriveDistanceOffboard (Java, C++)**: Demonstrates the use of a `TrapezoidProfileCommand` in conjunction with a “smart motor controller” to drive forward by a set distance with a trapezoidal motion profile.
- **FrisbeeBot (Java, C++)**: A complete set of robot code for a simple frisbee-shooting robot typical of the 2013 FRC® game *Ultimate Ascent*. Demonstrates simple PID control through the `PIDSubsystem` class.

- **Gears Bot (Java, C++)**: A complete set of robot code for the WPI demonstration robot, GearsBot.
- **Gyro Drive Commands (Java, C++)**: Demonstrates the use of `PIDCommand` and `ProfiledPIDCommand` in conjunction with a gyro to turn a robot to face a specified heading and to stabilize heading while driving.
- **Inlined Hatchbot (Java, C++)**: A complete set of robot code for a simple hatch-delivery bot typical of the 2019 FRC game *Destination: Deep Space*. Commands are written in an “inline” style, in which explicit subclassing of `Command` is avoided.
- **Traditional Hatchbot (Java, C++)**: A complete set of robot code for a simple hatch-delivery bot typical of the 2019 FRC game *Destination: Deep Space*. Commands are written in a “traditional” style, in which subclasses of `Command` are written for each robot action.
- **MecanumControllerCommand (Java, C++)**: Demonstrates trajectory generation and following with a mecanum drive using the `TrajectoryGenerator` and `MecanumControllerCommand` classes.
- **RamseteCommand (Java, C++)**: Demonstrates trajectory generation and following with a differential drive using the `TrajectoryGenerator` and `RamseteCommand` classes. A matching step-by-step tutorial can be found [here](#).
- **Scheduler Event Logging (Java, C++)**: Demonstrates the use of scheduler event actions to log dashboard event markers whenever a command starts, ends, or is interrupted.
- **Select Command Example (Java, C++)**: Demonstrates the use of the `SelectCommand` class to run one of a selection of commands depending on a runtime-evaluated condition.
- **SwerveControllerCommand (Java, C++)**: Demonstrates trajectory generation and following with a swerve drive using the `TrajectoryGenerator` and `SwerveControllerCommand` classes.

37.5 State-Space Examples

These examples demonstrate the use of the *State-Space Control*.

- **StateSpaceFlywheel (Java, C++)**: Demonstrates state-space control of a flywheel.
- **StateSpaceFlywheelSysId (Java, C++)**: Demonstrates state-space control using FRC Characterization’s System Identification for controlling a flywheel.
- **StateSpaceElevator (Java, C++)**: Demonstrates state-space control of an elevator.
- **StateSpaceArm (Java, C++)**: Demonstrates state-space control of an Arm.
- **StateSpaceDriveSimulation (Java, C++)**: Demonstrates state-space control of a differential drivetrain in combination with a RAMSETE path following controller and `Field2d` class.

37.6 Simulation Physics Examples

These examples demonstrate the use of the physics simulation.

- **ElevatorSimulation** (Java, C++): Demonstrates the use of physics simulation with a simple elevator.
- **ArmSimulation** (Java, C++): Demonstrates the use of physics simulation with a simple single-jointed arm.
- **StateSpaceDriveSimulation** (Java, C++): Demonstrates state-space control of a differential drivetrain in combination with a RAMSETE path following controller and Field2d class.
- **SimpleDifferentialDriveSimulation** (Java, C++): A barebones example of a basic drivetrain that can be used in simulation.

37.7 Miscellaneous Examples

These examples demonstrate miscellaneous WPILib functionality that does not fit into any of the above categories.

- **Addressable LED** (Java, C++): Demonstrates the use of the AddressableLED class to control RGB LEDs for robot decoration and/or driver feedback.
- **DMA** (C++): Demonstrates the use of DMA (Direct Memory Access) to read from sensors without using the RoboRIO's CPU (C++ only).
- **HAL** (C++): Demonstrates the use of HAL (Hardware Abstraction Layer) without the use of the rest of WPILib. This example is for advanced users (C++ only).
- **HID Rumble** (Java, C++): Demonstrates the use of the "rumble" functionality for tactile feedback on supported HID's (such as XboxControllers).
- **PacGoat** (Java, C++): A full command-based robot project from FRC Team 190's 2014 robot. Uses the legacy version of the command framework; categorized as miscellaneous to avoid confusion.
- **Shuffleboard** (Java, C++): Demonstrates configuring tab/widget layouts on the "Shuffleboard" dashboard from robot code through the Shuffleboard class's fluent builder API.
- **RomiReference** (Java, C++): A command based example of how to run the *Romi robot*.

Trajectory Tutorial

This is full tutorial for implementing trajectory generation and following on a differential-drive robot. The full code used in this tutorial can be found in the RamseteCommand example project ([Java](#), [C++](#)).

38.1 Trajectory Tutorial Overview

Note: Before following this tutorial, it is helpful (but not strictly necessary) to have a baseline familiarity with WPILib’s *PID control*, *feedforward*, and *trajectory* features.

Note: The robot code in this tutorial uses the *command-based* framework. The command-based framework is strongly recommended for beginning and intermediate teams.

The goal of this tutorial is to provide “end-to-end” instruction on implementing a trajectory-following autonomous routine for a differential-drive robot. By following this tutorial, readers will learn how to:

1. Accurately characterize their robot’s drivetrain to obtain accurate feedforward calculations and approximate feedback gains.
2. Configure a drive subsystem to track the robot’s pose using WPILib’s odometry library.
3. Generate a simple trajectory through a set of waypoints using WPILib’s TrajectoryGenerator class.
4. Follow the generated trajectory in an autonomous routine using WPILib’s RamseteCommand class with the calculated feedforward/feedback gains and pose.

This tutorial is intended to be approachable for teams without a great deal of programming expertise. While the WPILib library offers significant flexibility in the manner in which its trajectory-following features are implemented, closely following the implementation outlined in this tutorial should provide teams with a relatively-simple, clean, and repeatable solution for autonomous movement.

The full robot code for this tutorial can be found in the RamseteCommand Example Project ([Java](#), [C++](#)).

38.1.1 Why Trajectory Following?

FRC® games often feature autonomous tasks that require a robot to effectively and accurately move from a known starting location to a known scoring location. Historically, the most common solution for this sort of task in FRC has been a “drive-turn-drive” approach - that is, drive forward by a known distance, turn by a known angle, and drive forward by another known distance.

While the “drive-turn-drive” approach is certainly functional, in recent years teams have begun tracking smooth trajectories which require the robot to drive and turn at the same time. While this is a fundamentally more-complicated technical task, it offers significant benefits: in particular, since the robot no longer has to stop to change directions, the paths can be driven much faster, allowing a robot to score more game pieces during the autonomous period.

Beginning in 2020, WPILib now supplies teams with working, advanced code solutions for trajectory generation and tracking, significantly lowering the “barrier-to-entry” for this kind of advanced and effective autonomous motion.

38.1.2 Required Equipment

To follow this tutorial, you will need ready access to the following materials:

1. A differential-drive robot (such as the [AndyMark AM14U4](#)), equipped with:
 - Quadrature encoders for measuring the wheel rotation of each side of the drive.
 - A gyroscope for measuring robot heading.
2. A driver-station computer configured with:
 - *FRC Driver Station*.
 - *WPILib*.
 - *The FRC-Characterization Toolsuite*.

38.2 Step 1: Characterizing Your Robot Drive

Note: For detailed instructions on using the FRC-Characterization tool, see its [dedicated documentation](#).

Note: The drive characterization process requires ample space for the robot to drive. Be sure to have *at least* a 10' stretch (ideally closer to 20') in which the robot can drive during the characterization routine.

Note: The characterization data for this tutorial has been generously provided by Team 5190, who generated it as part of a demonstration of this functionality at the 2019 North Carolina State University P2P Workshop.

Before accurately following a path with a robot, it is important to have an accurate model for how the robot moves in response to its control inputs. Determining such a model is a process called “system identification.” WPILib’s FRC-Characterization is a tool for drive system identification.

38.2.1 Gathering the Data

We begin by gathering our drive characterization data.

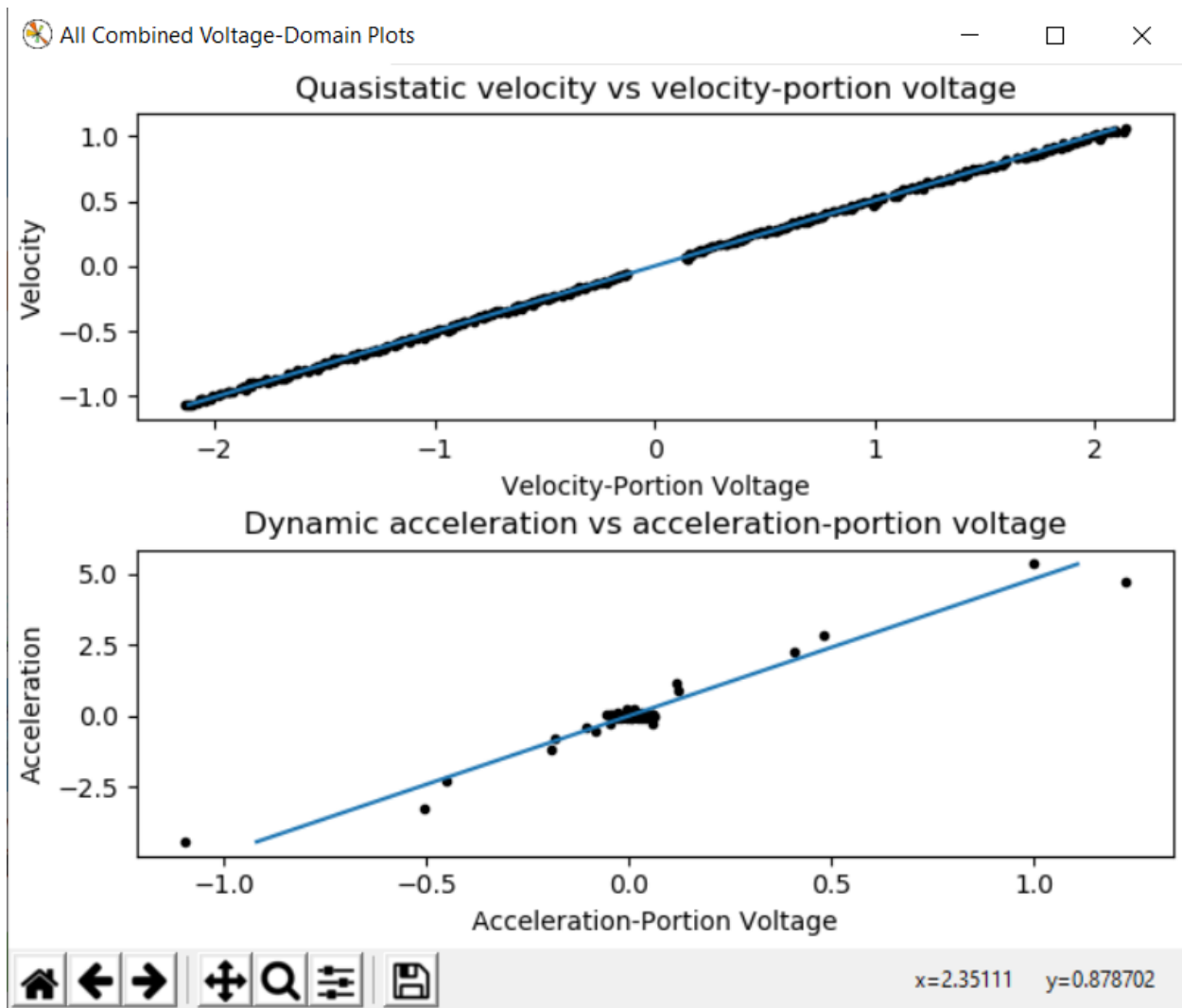
1. *Generate a robot project.*
 - Our example project uses the “simple” project type. Other project types (e.g. Talon and NEO) can be used without much difference; however, be *sure* to specify the required robot parameters correctly, regardless of project type!
 - WPILib’s trajectory library assumes units of meters are used universally for distance - accordingly, be certain that your robot’s wheel diameter is specified in meters!
2. *Deploy the robot project.*
3. *Run the Characterization Routine.*

38.2.2 Analyzing the Data

Once the characterization routine has been run and the data file has been saved, it is time to *open it in the analysis pane*.

Checking Diagnostics

Per the *FRC-Characterization guide*, we first view the diagnostics to ensure that our data look reasonable:



As our data look reasonably linear and our threshold seems to be set correctly, we may move on to the next step.

Record Feedforward Gains

Note: Feedforward gains do *not*, in general, transfer across robots. Do *not* use the gains from this tutorial for your own robot.

We now record the feedforward gains calculated by the tool:

FRC Drive Characterization Tool

Select Data File: C:/development/projects/characterization-ozzy/characterization-data20200113-1802.json Units: Meters Subset: All Combined

Wheel Diameter (units): .15

Feedforward Analysis		Feedback Analysis	
Analyze Data	Accel Window Size: 8	kS: 0.814	Gain Settings Preset: WPILib (2020-)
Time-Domain Diagnostics	Motion Threshold (units/s): 0.05	kV: 3.06	Controller Period (s): 0.02
Voltage-Domain Diagnostics		kA: 0.677	Max Controller Output: 12
3D Diagnostics		r-squared: 0.998	Time-Normalized Controller: <input checked="" type="checkbox"/>
	Track Width: 0.56341078		Loop Type: Position
		Controller Type: Onboard	kV: 3.06
		Post-Encoder Gearing: 1	kA: 0.677
		Encoder EPR: 4096	Calculate Optimal Controller Gains
		Has Slave: <input type="checkbox"/>	kP: 43.6
		Slave Update Period (s): 0.01	kD: 20.2

Since our wheel diameter was specified in meters, our feedforward gains are in the following units:

- kS: Volts
- kV: Volts * Seconds / Meters
- kA: Volts * Seconds² / Meters

If you have specified your units correctly, your feedforward gains will likely be within an order of magnitude of the ones reported here (a possible exception exists for kA, which may be vanishingly small if your robot is light). If they are not, it is possible you specified one of your drive parameters incorrectly when generating your robot project. A good test for this is to calculate the “theoretical” value of kV, which is 12 volts divided by the theoretical free speed of your drivetrain (which is, in turn, the free speed of the motor times the wheel circumference divided by the gear reduction). This value should agree very closely with the kV measured by the tool - if it does not, you have likely made an error somewhere.

Calculate Feedback Gains

Note: Feedback gains do *not*, in general, transfer across robots. Do *not* use the gains from this tutorial for your own robot.

We now *calculate the feedback gains* for the PID control that we will use to follow the path. Trajectory following with WPILib’s RAMSETE controller uses velocity closed-loop control, so we first select Velocity mode in the characterization tool:

FRC Drive Characterization Tool

Select Data File: C:/development/projects/characterization-ozzy/characterization-data20200113-1802.json Units: Meters Subset: All Combined

Wheel Diameter (units): .15

Feedforward Analysis		Feedback Analysis	
Analyze Data	Accel Window Size: 8	kS: 0.814	Gain Settings Preset: WPILib (2020-)
Time-Domain Diagnostics	Motion Threshold (units/s): 0.05	kV: 3.06	Controller Period (s): 0.02
Voltage-Domain Diagnostics		kA: 0.677	Max Controller Output: 12
3D Diagnostics		r-squared: 0.998	Time-Normalized Controller: <input checked="" type="checkbox"/>
	Track Width: 0.56341078		Loop Type: Velocity
		Controller Type: Onboard	kV: 3.06
		Post-Encoder Gearing: 1	kA: 0.677
		Encoder EPR: 4096	Calculate Optimal Controller Gains
		Has Slave: <input type="checkbox"/>	kP: 19.3
		Slave Update Period (s): 0.01	kD: 0.0

Since we will be using the WPILib PIDController for our velocity loop, we furthermore select the WPILib (2020-) option from the drop-down “presets” menu. This is *very* important, as the feedback gains will not be in the correct units if we do not select the correct preset:

The screenshot shows the FRC Drive Characterization Tool interface. The 'Feedback Analysis' tab is active. The 'Gain Settings Preset' dropdown is set to 'WPILib (2020-)' and is highlighted with a red box. Other settings include: Wheel Diameter (units): 0.15, Units: Meters, Subset: All Combined, Max Acceptable Position Error (units): 0.1, Max Acceptable Velocity Error (units/s): 0.2, Max Acceptable Control Effort (V): 7, Loop Type: Velocity, Controller Type: Onboard, Post-Encoder Gearing: 1, Encoder EPR: 4096, Has Slave: unchecked, Slave Update Period (s): 0.01, kV: 3.06, kA: 0.677, kP: 19.3, and kD: 0.0. The 'Calculate Optimal Controller Gains' button is visible.

Finally, we calculate and record the feedback gains for our control loop. Since it is a velocity controller, only a P gain is required:

The screenshot shows the FRC Drive Characterization Tool interface. The 'Feedback Analysis' tab is active. The 'Gain Settings Preset' dropdown is set to 'WPILib (2020-)' and is highlighted with a red box. Other settings include: Wheel Diameter (units): 0.15, Units: Meters, Subset: All Combined, Max Acceptable Position Error (units): 0.1, Max Acceptable Velocity Error (units/s): 0.2, Max Acceptable Control Effort (V): 7, Loop Type: Position, Controller Type: Onboard, Post-Encoder Gearing: 1, Encoder EPR: 4096, Has Slave: unchecked, Slave Update Period (s): 0.01, kV: 3.06, kA: 0.677, kP: 43.6, and kD: 20.2. The 'Calculate Optimal Controller Gains' button is visible.

Assuming we have done everything correctly, our proportional gain will be in units of Volts * Seconds / Meters. Thus, our calculated gain means that, for each meter per second of velocity error, the controller will output an additional 8.5 volts.

38.3 Step 2: Entering the Calculated Constants

Note: In C++, it is important that the feedforward constants be entered as the correct unit type. For more information on C++ units, see [The C++ Units Library](#).

Now that we have our characterization constants, it is time to place them in our code. The recommended place for this is the Constants file of the [standard command-based project structure](#).

The relevant parts of the constants file from the RamseteCommand Example Project (Java, C++) can be seen below.

38.3.1 Feedforward/Feedback Gains

Firstly, we must enter the feedforward and feedback gains which we obtained from the characterization tool.

Note: Feedforward and feedback gains do *not*, in general, transfer across robots. Do *not* use the gains from this tutorial for your own robot.

Java

C++ (Header)

```

44 // These are example values only - DO NOT USE THESE FOR YOUR OWN ROBOT!
45 // These characterization values MUST be determined either experimentally or
↳ theoretically
46 // for *your* robot's drive.
47 // The Robot Characterization Toolsuite provides a convenient tool for obtaining
↳ these
48 // values for your robot.
49 public static final double ksVolts = 0.22;
50 public static final double kvVoltSecondsPerMeter = 1.98;
51 public static final double kaVoltSecondsSquaredPerMeter = 0.2;
52
53 // Example value only - as above, this must be tuned for your drive!
54 public static final double kPDriveVel = 8.5;

```

```

46 // These are example values only - DO NOT USE THESE FOR YOUR OWN ROBOT!
47 // These characterization values MUST be determined either experimentally or
48 // theoretically for *your* robot's drive. The Robot Characterization
49 // Toolsuite provides a convenient tool for obtaining these values for your
50 // robot.
51 constexpr auto ks = 0.22_V;
52 constexpr auto kv = 1.98 * 1_V * 1_s / 1_m;
53 constexpr auto ka = 0.2 * 1_V * 1_s * 1_s / 1_m;
54
55 // Example value only - as above, this must be tuned for your drive!
56 constexpr double kPDriveVel = 8.5;

```

38.3.2 DifferentialDriveKinematics

Additionally, we must create an instance of the DifferentialDriveKinematics class, which allows us to use the trackwidth (i.e. horizontal distance between the wheels) of the robot to convert from chassis speeds to wheel speeds. As elsewhere, we keep our units in meters.

Java

C++ (Header)

```

32 public static final double kTrackwidthMeters = 0.69;
33 public static final DifferentialDriveKinematics kDriveKinematics =
34     new DifferentialDriveKinematics(kTrackwidthMeters);

```

```

35 constexpr auto kTrackwidth = 0.69_m;
36 extern const frc::DifferentialDriveKinematics kDriveKinematics;

```

38.3.3 Max Trajectory Velocity/Acceleration

We must also decide on a nominal max acceleration and max velocity for the robot during path-following. The maximum velocity value should be set somewhat below the nominal free-speed of the robot. Due to the later use of the `DifferentialDriveVoltageConstraint`, the maximum acceleration value is not extremely crucial.

Java

C++ (Header)

```
62 public static final double kMaxSpeedMetersPerSecond = 3;
63 public static final double kMaxAccelerationMetersPerSecondSquared = 3;
```

```
60 constexpr auto kMaxSpeed = 3_mps;
61 constexpr auto kMaxAcceleration = 3_mps_sq;
```

38.3.4 Ramsete Parameters

Finally, we must include a pair of parameters for the RAMSETE controller. The values shown below should work well for most robots, provided distances have been correctly measured in meters - for more information on tuning these values (if it is required), see [Constructing the Ramsete Controller Object](#).

Java

C++ (Header)

```
65 // Reasonable baseline values for a RAMSETE follower in units of meters and
↪seconds
66 public static final double kRamseteB = 2;
67 public static final double kRamseteZeta = 0.7;
```

```
63 // Reasonable baseline values for a RAMSETE follower in units of meters and
64 // seconds
65 constexpr double kRamseteB = 2;
66 constexpr double kRamseteZeta = 0.7;
```

38.4 Step 3: Creating a Drive Subsystem

Now that our drive is characterized, it is time to start writing our robot code *proper*. As mentioned before, we will use the *command-based* framework for our robot code. Accordingly, our first step is to write a suitable drive *subsystem* class.

The full drive class from the RamseteCommand Example Project (Java, C++) can be seen below. The rest of the article will describe the steps involved in writing this class.

Java

C++ (Header)

C++ (Source)

```

8 package edu.wpi.first.wpilibj.examples.ramsetecommand.subsystems;
9
10 import edu.wpi.first.wpilibj.ADXRS450_Gyro;
11 import edu.wpi.first.wpilibj.Encoder;
12 import edu.wpi.first.wpilibj.PWMVictorSPX;
13 import edu.wpi.first.wpilibj.SpeedControllerGroup;
14 import edu.wpi.first.wpilibj.drive.DifferentialDrive;
15 import edu.wpi.first.wpilibj.geometry.Pose2d;
16 import edu.wpi.first.wpilibj.interfaces.Gyro;
17 import edu.wpi.first.wpilibj.kinematics.DifferentialDriveOdometry;
18 import edu.wpi.first.wpilibj.kinematics.DifferentialDriveWheelSpeeds;
19 import edu.wpi.first.wpilibj2.command.SubsystemBase;
20
21 import edu.wpi.first.wpilibj.examples.ramsetecommand.Constants.DriveConstants;
22
23 public class DriveSubsystem extends SubsystemBase {
24     // The motors on the left side of the drive.
25     private final SpeedControllerGroup m_leftMotors =
26         new SpeedControllerGroup(new PWMVictorSPX(DriveConstants.kLeftMotor1Port),
27                                 new PWMVictorSPX(DriveConstants.kLeftMotor2Port));
28
29     // The motors on the right side of the drive.
30     private final SpeedControllerGroup m_rightMotors =
31         new SpeedControllerGroup(new PWMVictorSPX(DriveConstants.kRightMotor1Port),
32                                 new PWMVictorSPX(DriveConstants.kRightMotor2Port));
33
34     // The robot's drive
35     private final DifferentialDrive m_drive = new DifferentialDrive(m_leftMotors, m_
36 ↪ rightMotors);
37
38     // The left-side drive encoder
39     private final Encoder m_leftEncoder =
40 ↪ new Encoder(DriveConstants.kLeftEncoderPorts[0], DriveConstants.
41 ↪ kLeftEncoderPorts[1],
42 ↪ DriveConstants.kLeftEncoderReversed);
43
44     // The right-side drive encoder
45     private final Encoder m_rightEncoder =
46 ↪ new Encoder(DriveConstants.kRightEncoderPorts[0], DriveConstants.
47 ↪ kRightEncoderPorts[1],
48 ↪ DriveConstants.kRightEncoderReversed);
49
50     // The gyro sensor
51     private final Gyro m_gyro = new ADXRS450_Gyro();
52
53     // Odometry class for tracking robot pose
54     private final DifferentialDriveOdometry m_odometry;
55
56     /**
57     * Creates a new DriveSubsystem.
58     */
59     public DriveSubsystem() {
60         // Sets the distance per pulse for the encoders
61         m_leftEncoder.setDistancePerPulse(DriveConstants.kEncoderDistancePerPulse);
62         m_rightEncoder.setDistancePerPulse(DriveConstants.kEncoderDistancePerPulse);
63
64         resetEncoders();
65     }

```

(continues on next page)

(continued from previous page)

```

62     m_odometry = new DifferentialDriveOdometry(m_gyro.getRotation2d());
63 }
64
65 @Override
66 public void periodic() {
67     // Update the odometry in the periodic block
68     m_odometry.update(m_gyro.getRotation2d(), m_leftEncoder.getDistance(),
69                     m_rightEncoder.getDistance());
70 }
71
72 /**
73  * Returns the currently-estimated pose of the robot.
74  *
75  * @return The pose.
76  */
77 public Pose2d getPose() {
78     return m_odometry.getPoseMeters();
79 }
80
81 /**
82  * Returns the current wheel speeds of the robot.
83  *
84  * @return The current wheel speeds.
85  */
86 public DifferentialDriveWheelSpeeds getWheelSpeeds() {
87     return new DifferentialDriveWheelSpeeds(m_leftEncoder.getRate(), m_rightEncoder.
88 ↪ getRate());
89 }
90
91 /**
92  * Resets the odometry to the specified pose.
93  *
94  * @param pose The pose to which to set the odometry.
95  */
96 public void resetOdometry(Pose2d pose) {
97     resetEncoders();
98     m_odometry.resetPosition(pose, m_gyro.getRotation2d());
99 }
100
101 /**
102  * Drives the robot using arcade controls.
103  *
104  * @param fwd the commanded forward movement
105  * @param rot the commanded rotation
106  */
107 public void arcadeDrive(double fwd, double rot) {
108     m_drive.arcadeDrive(fwd, rot);
109 }
110
111 /**
112  * Controls the left and right sides of the drive directly with voltages.
113  *
114  * @param leftVolts the commanded left output
115  * @param rightVolts the commanded right output
116  */
117 public void tankDriveVolts(double leftVolts, double rightVolts) {

```

(continues on next page)

(continued from previous page)

```

117     m_leftMotors.setVoltage(leftVolts);
118     m_rightMotors.setVoltage(-rightVolts);
119     m_drive.feed();
120 }
121
122 /**
123  * Resets the drive encoders to currently read a position of 0.
124  */
125 public void resetEncoders() {
126     m_leftEncoder.reset();
127     m_rightEncoder.reset();
128 }
129
130 /**
131  * Gets the average distance of the two encoders.
132  *
133  * @return the average of the two encoder readings
134  */
135 public double getAverageEncoderDistance() {
136     return (m_leftEncoder.getDistance() + m_rightEncoder.getDistance()) / 2.0;
137 }
138
139 /**
140  * Gets the left drive encoder.
141  *
142  * @return the left drive encoder
143  */
144 public Encoder getLeftEncoder() {
145     return m_leftEncoder;
146 }
147
148 /**
149  * Gets the right drive encoder.
150  *
151  * @return the right drive encoder
152  */
153 public Encoder getRightEncoder() {
154     return m_rightEncoder;
155 }
156
157 /**
158  * Sets the max output of the drive. Useful for scaling the drive to drive more
159  * slowly.
160  *
161  * @param maxOutput the maximum output to which the drive will be constrained
162  */
163 public void setMaxOutput(double maxOutput) {
164     m_drive.setMaxOutput(maxOutput);
165 }
166
167 /**
168  * Zeroes the heading of the robot.
169  */
170 public void zeroHeading() {
171     m_gyro.reset();
172 }

```

(continues on next page)

(continued from previous page)

```

172
173 /**
174  * Returns the heading of the robot.
175  *
176  * @return the robot's heading in degrees, from -180 to 180
177  */
178 public double getHeading() {
179     return m_gyro.getRotation2d().getDegrees();
180 }
181
182 /**
183  * Returns the turn rate of the robot.
184  *
185  * @return The turn rate of the robot, in degrees per second
186  */
187 public double getTurnRate() {
188     return -m_gyro.getRate();
189 }
190 }

```

```

8  #pragma once
9
10 #include <frc/ADXRS450_Gyro.h>
11 #include <frc/Encoder.h>
12 #include <frc/PWMMotorSPX.h>
13 #include <frc/SpeedControllerGroup.h>
14 #include <frc/drive/DifferentialDrive.h>
15 #include <frc/geometry/Pose2d.h>
16 #include <frc/kinematics/DifferentialDriveOdometry.h>
17 #include <frc2/command/SubsystemBase.h>
18 #include <units/voltage.h>
19
20 #include "Constants.h"
21
22 class DriveSubsystem : public frc2::SubsystemBase {
23 public:
24     DriveSubsystem();
25
26     /**
27      * Will be called periodically whenever the CommandScheduler runs.
28      */
29     void Periodic() override;
30
31     // Subsystem methods go here.
32
33     /**
34      * Drives the robot using arcade controls.
35      *
36      * @param fwd the commanded forward movement
37      * @param rot the commanded rotation
38      */
39     void ArcadeDrive(double fwd, double rot);
40
41     /**
42      * Controls each side of the drive directly with a voltage.
43      */

```

(continues on next page)

(continued from previous page)

```

44  * @param left the commanded left output
45  * @param right the commanded right output
46  */
47  void TankDriveVolts(units::volt_t left, units::volt_t right);
48
49  /**
50   * Resets the drive encoders to currently read a position of 0.
51   */
52  void ResetEncoders();
53
54  /**
55   * Gets the average distance of the TWO encoders.
56   *
57   * @return the average of the TWO encoder readings
58   */
59  double GetAverageEncoderDistance();
60
61  /**
62   * Gets the left drive encoder.
63   *
64   * @return the left drive encoder
65   */
66  frc::Encoder& GetLeftEncoder();
67
68  /**
69   * Gets the right drive encoder.
70   *
71   * @return the right drive encoder
72   */
73  frc::Encoder& GetRightEncoder();
74
75  /**
76   * Sets the max output of the drive. Useful for scaling the drive to drive
77   * more slowly.
78   *
79   * @param maxOutput the maximum output to which the drive will be constrained
80   */
81  void SetMaxOutput(double maxOutput);
82
83  /**
84   * Returns the heading of the robot.
85   *
86   * @return the robot's heading in degrees, from -180 to 180
87   */
88  units::degree_t GetHeading() const;
89
90  /**
91   * Returns the turn rate of the robot.
92   *
93   * @return The turn rate of the robot, in degrees per second
94   */
95  double GetTurnRate();
96
97  /**
98   * Returns the currently-estimated pose of the robot.
99   *

```

(continues on next page)

(continued from previous page)

```

100     * @return The pose.
101     */
102     frc::Pose2d GetPose();
103
104     /**
105     * Returns the current wheel speeds of the robot.
106     *
107     * @return The current wheel speeds.
108     */
109     frc::DifferentialDriveWheelSpeeds GetWheelSpeeds();
110
111     /**
112     * Resets the odometry to the specified pose.
113     *
114     * @param pose The pose to which to set the odometry.
115     */
116     void ResetOdometry(frc::Pose2d pose);
117
118 private:
119     // Components (e.g. motor controllers and sensors) should generally be
120     // declared private and exposed only through public methods.
121
122     // The motor controllers
123     frc::PWMVictorSPX m_left1;
124     frc::PWMVictorSPX m_left2;
125     frc::PWMVictorSPX m_right1;
126     frc::PWMVictorSPX m_right2;
127
128     // The motors on the left side of the drive
129     frc::SpeedControllerGroup m_leftMotors{m_left1, m_left2};
130
131     // The motors on the right side of the drive
132     frc::SpeedControllerGroup m_rightMotors{m_right1, m_right2};
133
134     // The robot's drive
135     frc::DifferentialDrive m_drive{m_leftMotors, m_rightMotors};
136
137     // The left-side drive encoder
138     frc::Encoder m_leftEncoder;
139
140     // The right-side drive encoder
141     frc::Encoder m_rightEncoder;
142
143     // The gyro sensor
144     frc::ADXRS450_Gyro m_gyro;
145
146     // Odometry class for tracking robot pose
147     frc::DifferentialDriveOdometry m_odometry;
148 };

```

```

8  #include "subsystems/DriveSubsystem.h"
9
10 #include <frc/geometry/Rotation2d.h>
11 #include <frc/kinematics/DifferentialDriveWheelSpeeds.h>
12
13 using namespace DriveConstants;

```

(continues on next page)

(continued from previous page)

```

14 DriveSubsystem::DriveSubsystem()
15 : m_left1{kLeftMotor1Port},
16   m_left2{kLeftMotor2Port},
17   m_right1{kRightMotor1Port},
18   m_right2{kRightMotor2Port},
19   m_leftEncoder{kLeftEncoderPorts[0], kLeftEncoderPorts[1]},
20   m_rightEncoder{kRightEncoderPorts[0], kRightEncoderPorts[1]},
21   m_odometry{m_gyro.GetRotation2d()} {
22     // Set the distance per pulse for the encoders
23     m_leftEncoder.SetDistancePerPulse(kEncoderDistancePerPulse);
24     m_rightEncoder.SetDistancePerPulse(kEncoderDistancePerPulse);
25
26     ResetEncoders();
27 }
28
29 void DriveSubsystem::Periodic() {
30     // Implementation of subsystem periodic method goes here.
31     m_odometry.Update(m_gyro.GetRotation2d(),
32                      units::meter_t(m_leftEncoder.GetDistance()),
33                      units::meter_t(m_rightEncoder.GetDistance()));
34 }
35
36 void DriveSubsystem::ArcadeDrive(double fwd, double rot) {
37     m_drive.ArcadeDrive(fwd, rot);
38 }
39
40 void DriveSubsystem::TankDriveVolts(units::volt_t left, units::volt_t right) {
41     m_leftMotors.SetVoltage(left);
42     m_rightMotors.SetVoltage(-right);
43     m_drive.Feed();
44 }
45
46 void DriveSubsystem::ResetEncoders() {
47     m_leftEncoder.Reset();
48     m_rightEncoder.Reset();
49 }
50
51 double DriveSubsystem::GetAverageEncoderDistance() {
52     return (m_leftEncoder.GetDistance() + m_rightEncoder.GetDistance()) / 2.0;
53 }
54
55 frc::Encoder& DriveSubsystem::GetLeftEncoder() { return m_leftEncoder; }
56
57 frc::Encoder& DriveSubsystem::GetRightEncoder() { return m_rightEncoder; }
58
59 void DriveSubsystem::SetMaxOutput(double maxOutput) {
60     m_drive.SetMaxOutput(maxOutput);
61 }
62
63 units::degree_t DriveSubsystem::GetHeading() const {
64     return m_gyro.GetRotation2d().Degrees();
65 }
66
67 double DriveSubsystem::GetTurnRate() { return -m_gyro.GetRate(); }
68
69

```

(continues on next page)

(continued from previous page)

```

70 frc::Pose2d DriveSubsystem::GetPose() { return m_odometry.GetPose(); }
71
72 frc::DifferentialDriveWheelSpeeds DriveSubsystem::GetWheelSpeeds() {
73     return {units::meters_per_second_t(m_leftEncoder.GetRate()),
74            units::meters_per_second_t(m_rightEncoder.GetRate())};
75 }
76
77 void DriveSubsystem::ResetOdometry(frc::Pose2d pose) {
78     ResetEncoders();
79     m_odometry.ResetPosition(pose, m_gyro.GetRotation2d());
80 }

```

38.4.1 Configuring the Drive Encoders

The drive encoders measure the rotation of the wheels on each side of the drive. To properly configure the encoders, we need to specify two things: the ports the encoders are plugged into, and the distance per encoder pulse. Then, we need to write methods allowing access to the encoder values from code that uses the subsystem.

Encoder Ports

The encoder ports are specified in the encoder's constructor, like so:

Java

C++ (Source)

```

38 private final Encoder m_leftEncoder =
39     new Encoder(DriveConstants.kLeftEncoderPorts[0], DriveConstants.
40     ↪ kLeftEncoderPorts[1],
41               DriveConstants.kLeftEncoderReversed);
42
43 // The right-side drive encoder
44 private final Encoder m_rightEncoder =
45     new Encoder(DriveConstants.kRightEncoderPorts[0], DriveConstants.
46     ↪ kRightEncoderPorts[1],
47               DriveConstants.kRightEncoderReversed);

```

```

20 m_leftEncoder{kLeftEncoderPorts[0], kLeftEncoderPorts[1]},
21 m_rightEncoder{kRightEncoderPorts[0], kRightEncoderPorts[1]},

```

Encoder Distance per Pulse

The distance per pulse is specified by calling the encoder's `setDistancePerPulse` method. Note that for the WPILib Encoder class, "pulse" refers to a full encoder cycle (i.e. four edges), and thus will be 1/4 the value that was specified in the FRC-Characterization config. Remember, as well, that the distance should be measured in meters!

Java

C++ (Source)

```

58 m_leftEncoder.setDistancePerPulse(DriveConstants.kEncoderDistancePerPulse);
59 m_rightEncoder.setDistancePerPulse(DriveConstants.kEncoderDistancePerPulse);

```

```

23 // Set the distance per pulse for the encoders
24 m_leftEncoder.SetDistancePerPulse(kEncoderDistancePerPulse);
25 m_rightEncoder.SetDistancePerPulse(kEncoderDistancePerPulse);

```

Encoder Accessor Method

To access the values measured by the encoders, we include the following method:

Important: The returned velocities **must** be in meters! Because we configured the distance per pulse on the encoders above, calling `getRate()` will automatically apply the conversion factor from encoder units to meters. If you are not using WPILib's Encoder class, you must perform this conversion either through the respective vendor's API or by manually multiplying by a conversion factor.

Java

C++ (Source)

```

82 /**
83  * Returns the current wheel speeds of the robot.
84  *
85  * @return The current wheel speeds.
86  */
87 public DifferentialDriveWheelSpeeds getWheelSpeeds() {
88     return new DifferentialDriveWheelSpeeds(m_leftEncoder.getRate(), m_rightEncoder.
89     ↪ getRate());
89 }

```

```

72 frc::DifferentialDriveWheelSpeeds DriveSubsystem::GetWheelSpeeds() {
73     return {units::meters_per_second_t(m_leftEncoder.GetRate()),
74             units::meters_per_second_t(m_rightEncoder.GetRate())};
75 }

```

We wrap the measured encoder values in a `DifferentialDriveWheelSpeeds` object for easier integration with the `RamseteCommand` class later on.

38.4.2 Configuring the Gyroscope

The gyroscope measures the rate of change of the robot's heading (which can then be integrated to provide a measurement of the robot's heading relative to when it first turned on). In our example, we use the [Analog Devices ADXRS450 FRC Gyro Board](#), which has been included in the kit of parts for several years:

Java

C++ (Header)

```

48 private final Gyro m_gyro = new ADXRS450_Gyro();

```

```
143 // The gyro sensor
144 frc::ADXRS450_Gyro m_gyro;
```

Gyroscope Accessor Method

To access the current heading measured by the gyroscope, we include the following method:

Java

C++ (Source)

```
173 /**
174  * Returns the heading of the robot.
175  *
176  * @return the robot's heading in degrees, from -180 to 180
177  */
178 public double getHeading() {
179     return m_gyro.getRotation2d().getDegrees();
180 }
```

```
64 units::degree_t DriveSubsystem::GetHeading() const {
65     return m_gyro.GetRotation2d().Degrees();
66 }
```

38.4.3 Configuring the Odometry

Now that we have our encoders and gyroscope configured, it is time to set up our drive subsystem to automatically compute its position from the encoder and gyroscope readings.

First, we create a member instance of the `DifferentialDriveOdometry` class:

Java

C++ (Header)

```
51 private final DifferentialDriveOdometry m_odometry;
```

```
146 // Odometry class for tracking robot pose
147 frc::DifferentialDriveOdometry m_odometry;
```

Updating the Odometry

The odometry class must be regularly updated to incorporate new readings from the encoder and gyroscope. We accomplish this inside the subsystem's periodic method, which is automatically called once per main loop iteration:

Java

C++ (Source)

```

66 public void periodic() {
67     // Update the odometry in the periodic block
68     m_odometry.update(m_gyro.getRotation2d(), m_leftEncoder.getDistance(),
69                     m_rightEncoder.getDistance());
70 }

```

```

30 void DriveSubsystem::Periodic() {
31     // Implementation of subsystem periodic method goes here.
32     m_odometry.Update(m_gyro.GetRotation2d(),
33                     units::meter_t(m_leftEncoder.GetDistance()),
34                     units::meter_t(m_rightEncoder.GetDistance()));
35 }

```

Odometry Accessor Method

To access the robot's current computed pose, we include the following method:

Java

C++ (Source)

```

72 /**
73  * Returns the currently-estimated pose of the robot.
74  *
75  * @return The pose.
76  */
77 public Pose2d getPose() {
78     return m_odometry.getPoseMeters();
79 }

```

```

70 frc::Pose2d DriveSubsystem::GetPose() { return m_odometry.GetPose(); }

```

38.4.4 Voltage-Based Drive Method

Finally, we must include one additional method - a method that allows us to set the voltage to each side of the drive using the `setVoltage()` method of the `SpeedController` interface. The default WPILib drive class does not include this functionality, so we must write it ourselves:

Java

C++ (Source)

```

110 /**
111  * Controls the left and right sides of the drive directly with voltages.
112  *
113  * @param leftVolts the commanded left output
114  * @param rightVolts the commanded right output
115  */
116 public void tankDriveVolts(double leftVolts, double rightVolts) {
117     m_leftMotors.setVoltage(leftVolts);
118     m_rightMotors.setVoltage(-rightVolts);
119     m_drive.feed();
120 }

```

```

41 void DriveSubsystem::TankDriveVolts(units::volt_t left, units::volt_t right) {
42     m_leftMotors.SetVoltage(left);
43     m_rightMotors.SetVoltage(-right);
44     m_drive.Feed();

```

It is very important to use the `setVoltage()` method rather than the ordinary `set()` method, as this will automatically compensate for battery “voltage sag” during operation. Since our feedforward voltages are physically-meaningful (as they are based on measured characterization data), this is essential to ensuring their accuracy.

38.5 Step 4: Creating and Following a Trajectory

With our drive subsystem written, it is now time to generate a trajectory and write an autonomous command to follow it.

As per the [standard command-based project structure](#), we will do this in the `getAutonomousCommand` method of the `RobotContainer` class. The full method from the `RamseteCommand Example Project (Java, C++)` can be seen below. The rest of the article will break down the different parts of the method in more detail.

Java

C++ (Source)

```

82  /**
83   * Use this to pass the autonomous command to the main {@link Robot} class.
84   *
85   * @return the command to run in autonomous
86   */
87  public Command getAutonomousCommand() {
88
89      // Create a voltage constraint to ensure we don't accelerate too fast
90      var autoVoltageConstraint =
91          new DifferentialDriveVoltageConstraint(
92              new SimpleMotorFeedforward(DriveConstants.kvVolts,
93                                          DriveConstants.kvVoltSecondsPerMeter,
94                                          DriveConstants.kaVoltSecondsSquaredPerMeter),
95              DriveConstants.kDriveKinematics,
96              10);
97
98      // Create config for trajectory
99      TrajectoryConfig config =
100          new TrajectoryConfig(AutoConstants.kMaxSpeedMetersPerSecond,
101                               AutoConstants.kMaxAccelerationMetersPerSecondSquared)
102          // Add kinematics to ensure max speed is actually obeyed
103          .setKinematics(DriveConstants.kDriveKinematics)
104          // Apply the voltage constraint
105          .addConstraint(autoVoltageConstraint);
106
107      // An example trajectory to follow. All units in meters.
108      Trajectory exampleTrajectory = TrajectoryGenerator.generateTrajectory(
109          // Start at the origin facing the +X direction
110          new Pose2d(0, 0, new Rotation2d(0)),
111          // Pass through these two interior waypoints, making an 's' curve path
112          List.of(

```

(continues on next page)

(continued from previous page)

```

113         new Translation2d(1, 1),
114         new Translation2d(2, -1)
115     ),
116     // End 3 meters straight ahead of where we started, facing forward
117     new Pose2d(3, 0, new Rotation2d(0)),
118     // Pass config
119     config
120 );
121
122 RamseteCommand ramseteCommand = new RamseteCommand(
123     exampleTrajectory,
124     m_robotDrive::getPose,
125     new RamseteController(AutoConstants.kRamseteB, AutoConstants.kRamseteZeta),
126     new SimpleMotorFeedforward(DriveConstants.kSVolts,
127                               DriveConstants.kVVoltsSecondsPerMeter,
128                               DriveConstants.kAVoltsSecondsSquaredPerMeter),
129     DriveConstants.kDriveKinematics,
130     m_robotDrive::getWheelSpeeds,
131     new PIDController(DriveConstants.kPDriveVel, 0, 0),
132     new PIDController(DriveConstants.kPDriveVel, 0, 0),
133     // RamseteCommand passes volts to the callback
134     m_robotDrive::tankDriveVolts,
135     m_robotDrive
136 );
137
138 // Reset odometry to the starting pose of the trajectory.
139 m_robotDrive.resetOdometry(exampleTrajectory.getInitialPose());
140
141 // Run path following command, then stop at the end.
142 return ramseteCommand.andThen(() -> m_robotDrive.tankDriveVolts(0, 0));
143 }

```

```

48 frc2::Command* RobotContainer::GetAutonomousCommand() {
49     // Create a voltage constraint to ensure we don't accelerate too fast
50     frc::DifferentialDriveVoltageConstraint autoVoltageConstraint(
51         frc::SimpleMotorFeedforward<units::meters>(
52             DriveConstants::ks, DriveConstants::kv, DriveConstants::ka),
53         DriveConstants::kDriveKinematics, 10_V);
54
55     // Set up config for trajectory
56     frc::TrajectoryConfig config(AutoConstants::kMaxSpeed,
57                                AutoConstants::kMaxAcceleration);
58     // Add kinematics to ensure max speed is actually obeyed
59     config.SetKinematics(DriveConstants::kDriveKinematics);
60     // Apply the voltage constraint
61     config.AddConstraint(autoVoltageConstraint);
62
63     // An example trajectory to follow. All units in meters.
64     auto exampleTrajectory = frc::TrajectoryGenerator::GenerateTrajectory(
65         // Start at the origin facing the +X direction
66         frc::Pose2d(0_m, 0_m, frc::Rotation2d(0_deg)),
67         // Pass through these two interior waypoints, making an 's' curve path
68         {frc::Translation2d(1_m, 1_m), frc::Translation2d(2_m, -1_m)},
69         // End 3 meters straight ahead of where we started, facing forward
70         frc::Pose2d(3_m, 0_m, frc::Rotation2d(0_deg)),
71         // Pass the config

```

(continues on next page)

(continued from previous page)

```

72     config);
73
74     frc2::RamseteCommand ramseteCommand(
75         exampleTrajectory, [this]() { return m_drive.GetPose(); },
76         frc::RamseteController(AutoConstants::kRamseteB,
77                               AutoConstants::kRamseteZeta),
78         frc::SimpleMotorFeedforward<units::meters>(
79             DriveConstants::ks, DriveConstants::kv, DriveConstants::ka),
80         DriveConstants::kDriveKinematics,
81         [this] { return m_drive.GetWheelSpeeds(); },
82         frc2::PIDController(DriveConstants::kPDriveVel, 0, 0),
83         frc2::PIDController(DriveConstants::kPDriveVel, 0, 0),
84         [this](auto left, auto right) { m_drive.TankDriveVolts(left, right); },
85         {&m_drive});
86
87     // Reset odometry to the starting pose of the trajectory.
88     m_drive.ResetOdometry(exampleTrajectory.InitialPose());
89
90     // no auto
91     return new frc2::SequentialCommandGroup(
92         std::move(ramseteCommand),
93         frc2::InstantCommand([this] { m_drive.TankDriveVolts(0_V, 0_V); }, {}));

```

38.5.1 Configuring the Trajectory Constraints

First, we must set some configuration parameters for the trajectory which will ensure that the generated trajectory is followable.

Creating a Voltage Constraint

The first piece of configuration we will need is a voltage constraint. This will ensure that the generated trajectory never commands the robot to go faster than it is capable of achieving with the given voltage supply:

Java

C++ (Source)

```

89     // Create a voltage constraint to ensure we don't accelerate too fast
90     var autoVoltageConstraint =
91         new DifferentialDriveVoltageConstraint(
92             new SimpleMotorFeedforward(DriveConstants.ksVolts,
93                                         DriveConstants.kvVoltSecondsPerMeter,
94                                         DriveConstants.kaVoltSecondsSquaredPerMeter),
95             DriveConstants.kDriveKinematics,
96             10);

```

```

49     // Create a voltage constraint to ensure we don't accelerate too fast
50     frc::DifferentialDriveVoltageConstraint autoVoltageConstraint(
51         frc::SimpleMotorFeedforward<units::meters>(
52             DriveConstants::ks, DriveConstants::kv, DriveConstants::ka),
53         DriveConstants::kDriveKinematics, 10_V);

```


Notice that we set the maximum voltage to 10V, rather than the nominal battery voltage of 12V. This gives us some “headroom” to deal with “voltage sag” during operation.

Creating the Configuration

Now that we have our voltage constraint, we can create our `TrajectoryConfig` instance, which wraps together all of our path constraints:

Java

C++ (Source)

```

98 // Create config for trajectory
99 TrajectoryConfig config =
100     new TrajectoryConfig(AutoConstants.kMaxSpeedMetersPerSecond,
101                          AutoConstants.kMaxAccelerationMetersPerSecondSquared)
102     // Add kinematics to ensure max speed is actually obeyed
103     .setKinematics(DriveConstants.kDriveKinematics)
104     // Apply the voltage constraint
105     .addConstraint(autoVoltageConstraint);

```

```

55 // Set up config for trajectory
56 frc::TrajectoryConfig config(AutoConstants::kMaxSpeed,
57                              AutoConstants::kMaxAcceleration);
58 // Add kinematics to ensure max speed is actually obeyed
59 config.SetKinematics(DriveConstants::kDriveKinematics);
60 // Apply the voltage constraint
61 config.AddConstraint(autoVoltageConstraint);

```

38.5.2 Generating the Trajectory

With our trajectory configuration in hand, we are now ready to generate our trajectory. For this example, we will be generating a “clamped cubic” trajectory - this means we will specify full robot poses at the endpoints, and positions only for interior waypoints (also known as “knot points”). As elsewhere, all distances are in meters.

Java

C++ (Source)

```

108 Trajectory exampleTrajectory = TrajectoryGenerator.generateTrajectory(
109     // Start at the origin facing the +X direction
110     new Pose2d(0, 0, new Rotation2d(0)),
111     // Pass through these two interior waypoints, making an 's' curve path
112     List.of(
113         new Translation2d(1, 1),
114         new Translation2d(2, -1)
115     ),
116     // End 3 meters straight ahead of where we started, facing forward
117     new Pose2d(3, 0, new Rotation2d(0)),
118     // Pass config
119     config
120 );

```

```

63 // An example trajectory to follow. All units in meters.
64 auto exampleTrajectory = frc::TrajectoryGenerator::GenerateTrajectory(
65     // Start at the origin facing the +X direction
66     frc::Pose2d(0_m, 0_m, frc::Rotation2d(0_deg)),
67     // Pass through these two interior waypoints, making an 's' curve path
68     {frc::Translation2d(1_m, 1_m), frc::Translation2d(2_m, -1_m)},
69     // End 3 meters straight ahead of where we started, facing forward
70     frc::Pose2d(3_m, 0_m, frc::Rotation2d(0_deg)),
71     // Pass the config
72     config);

```

Note: Instead of generating the trajectory on the roboRIO as outlined above, one can also *import a PathWeaver JSON*.

38.5.3 Creating the RamseteCommand

We will first reset our robot's pose to the starting pose of the trajectory. This ensures that the robot's location on the coordinate system and the trajectory's starting position are the same.

Java

C++ (Source)

```

137 // Reset odometry to the starting pose of the trajectory.
138 m_robotDrive.resetOdometry(exampleTrajectory.getInitialPose());

```

```

87 // Reset odometry to the starting pose of the trajectory.
88 m_drive.ResetOdometry(exampleTrajectory.InitialPose());

```

It is very important that the initial robot pose match the first pose in the trajectory. For the purposes of our example, the robot will be reliably starting at a position of $(0,0)$ with a heading of 0 . In actual use, however, it is probably not desirable to base your coordinate system on the robot position, and so the starting position for both the robot and the trajectory should be set to some other value. If you wish to use a trajectory that has been defined in robot-centric coordinates in such a situation, you can transform it to be relative to the robot's current pose using the `transformBy` method (Java, C++). For more information about transforming trajectories, see [Transforming Trajectories](#).

Now that we have a trajectory, we can create a command that, when executed, will follow that trajectory. To do this, we use the `RamseteCommand` class (Java, C++)

Java

C++ (Source)

```

122 RamseteCommand ramseteCommand = new RamseteCommand(
123     exampleTrajectory,
124     m_robotDrive::getPose,
125     new RamseteController(AutoConstants.kRamseteB, AutoConstants.kRamseteZeta),
126     new SimpleMotorFeedforward(DriveConstants.kSVolts,
127                                 DriveConstants.kvVoltSecondsPerMeter,
128                                 DriveConstants.kaVoltSecondsSquaredPerMeter),
129     DriveConstants.kDriveKinematics,
130     m_robotDrive::getWheelSpeeds,

```

(continues on next page)

(continued from previous page)

```

131     new PIDController(DriveConstants.kPDriveVel, 0, 0),
132     new PIDController(DriveConstants.kPDriveVel, 0, 0),
133     // RamseteCommand passes volts to the callback
134     m_robotDrive::tankDriveVolts,
135     m_robotDrive
136 );

74 frc2::RamseteCommand ramseteCommand(
75     exampleTrajectory, [this]() { return m_drive.GetPose(); },
76     frc::RamseteController(AutoConstants::kRamseteB,
77                           AutoConstants::kRamseteZeta),
78     frc::SimpleMotorFeedforward<units::meters>(
79         DriveConstants::ks, DriveConstants::kv, DriveConstants::ka),
80     DriveConstants::kDriveKinematics,
81     [this] { return m_drive.GetWheelSpeeds(); },
82     frc2::PIDController(DriveConstants::kPDriveVel, 0, 0),
83     frc2::PIDController(DriveConstants::kPDriveVel, 0, 0),
84     [this](auto left, auto right) { m_drive.TankDriveVolts(left, right); },
85     {&m_drive});

```

This declaration is fairly substantial, so we'll go through it argument-by-argument:

1. The trajectory: This is the trajectory to be followed; accordingly, we pass the command the trajectory we just constructed in our earlier steps.
2. The pose supplier: This is a method reference (or lambda) to the *drive subsystem method that returns the pose*. The RAMSETE controller needs the current pose measurement to determine the required wheel outputs.
3. The RAMSETE controller: This is the RamseteController object (Java, C++) that will perform the path-following computation that translates the current measured pose and trajectory state into a chassis speed setpoint.
4. The drive feedforward: This is a SimpleMotorFeedforward object (Java, C++) that will automatically perform the correct feedforward calculation with the feedforward gains (kS, kV, and kA) that we obtained from the drive characterization tool.
5. The drive kinematics: This is the DifferentialDriveKinematics object (Java, C++) that we constructed earlier in our constants file, and will be used to convert chassis speeds to wheel speeds.
6. The wheel speed supplier: This is a method reference (or lambda) to the *drive subsystem method that returns the wheel speeds*.
7. The left-side PIDController: This is the PIDController object (Java, C++) that will track the left-side wheel speed setpoint, using the P gain that we obtained from the drive characterization tool.
8. The right-side PIDController: This is the PIDController object (Java, C++) that will track the right-side wheel speed setpoint, using the P gain that we obtained from the drive characterization tool.
9. The output consumer: This is a method reference (or lambda) to the *drive subsystem method that passes the voltage outputs to the drive motors*.
10. The robot drive: This is the drive subsystem itself, included to ensure the command does not operate on the drive at the same time as any other command that uses the drive.

Finally, note that we append a final “stop” command in sequence after the path-following command, to ensure that the robot stops moving at the end of the trajectory.

38.5.4 Video

If all has gone well, your robot’s autonomous routine should look something like this:

Drivetrain Simulation Tutorial

This is a tutorial for implementing a simulation model of your differential drivetrain using the new 2021 simulation classes. Although the code that we will cover in this tutorial is framework-agnostic, there are two full examples available - one for each framework.

- `StateSpaceDifferentialDriveSimulation` (Java, C++) uses the command-based framework.
- `SimpleDifferentialDriveSimulation` (Java, C++) uses a more traditional approach to data flow.

Both of these examples are also available in the VS Code *New Project* window.

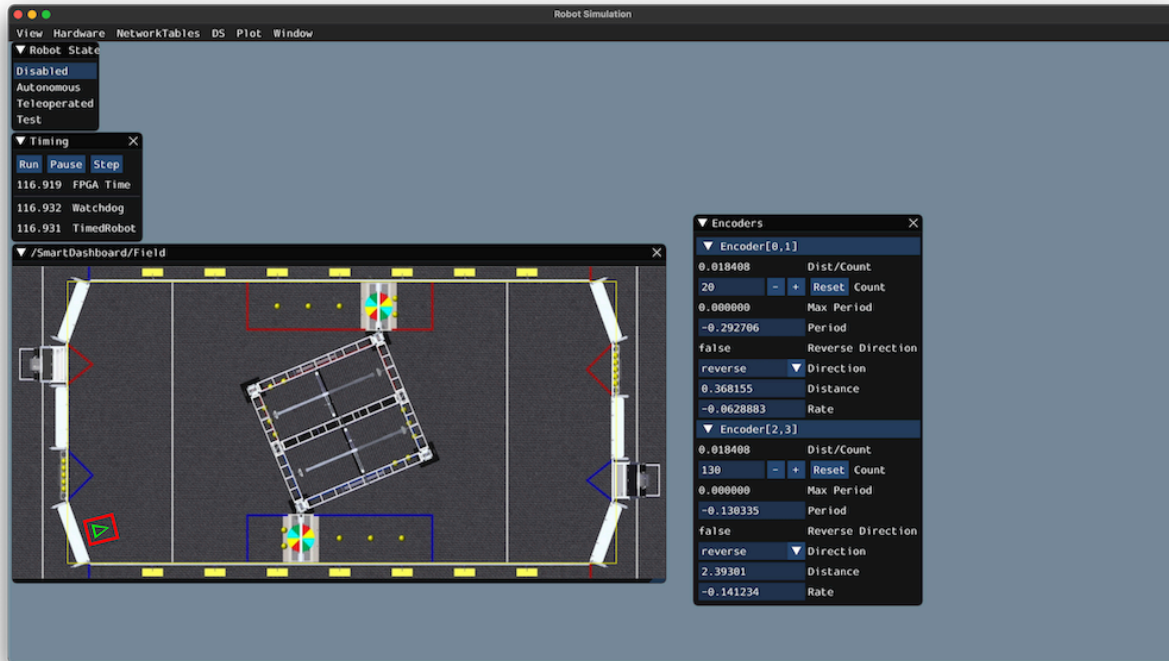
39.1 Drivetrain Simulation Overview

Note: WPILib only supports simulating differential drivetrains for the 2021 season. Support for similar simulation capabilities for teams using swerve and mecanum drivetrains is in development.

Note: The code in this tutorial does not use any specific framework (i.e. command-based vs. simple data flow); however, guidance will be provided in certain areas for how to best implement certain pieces of code in specific framework types.

The goal of this tutorial is to provide guidance on implementing simulation capabilities for a differential-drivetrain robot. By the end of this tutorial, you should be able to:

1. Understand the basic underlying concepts behind the WPILib simulation framework.
2. Create a drivetrain simulation model using your robot's physical parameters.
3. Use the simulation model to predict how your real robot will move given specific voltage inputs.
4. Tune feedback constants and squash common bugs (e.g. motor inversion) before having access to physical hardware.
5. Use the Simulation GUI to visualize robot movement on a virtual field.



39.1.1 Why Simulate a Drivetrain?

The drivetrain of a robot is one of the most important mechanisms on the robot – therefore, it is important to ensure that the software powering your drivetrain is as robust as possible. By being able to simulate how a physical drivetrain responds, you can get a head start on writing quality software before you have access to the physical hardware. With the simulation framework, you can verify not only basic functionality, like making sure that the inversions on motors and encoders are correct, but also advanced capabilities such as verifying accuracy of path following.

39.2 Step 1: Creating Simulated Instances of Hardware

The WPILib simulation framework contains several XXXSim classes, where XXX represents physical hardware such as encoders or gyroscopes. These simulation classes can be used to set positions and velocities (for encoders) and angles (for gyroscopes) from a model of your drivetrain.

39.2.1 Simulating Encoders

The `EncoderSim` class allows users to set encoder positions and velocities on a given `Encoder` object. When running on real hardware, the `Encoder` class interacts with real sensors to count revolutions (and convert them to distance units automatically if configured to do so); however, in simulation there are no such measurements to make. The `EncoderSim` class can accept these simulated readings from a model of your drivetrain.

Note: It is not possible to simulate encoders that are directly connected to CAN motor controllers using WPILib classes. For more information about your specific motor controller, please read your vendor's documentation.

Java

C++

```
// These represent our regular encoder objects, which we would
// create to use on a real robot.
private Encoder m_leftEncoder = new Encoder(0, 1);
private Encoder m_rightEncoder = new Encoder(2, 3);

// These are our EncoderSim objects, which we will only use in
// simulation. However, you do not need to comment out these
// declarations when you are deploying code to the roboRIO.
private EncoderSim m_leftEncoderSim = new EncoderSim(m_leftEncoder);
private EncoderSim m_rightEncoderSim = new EncoderSim(m_rightEncoder);
```

```
#include <frc/Encoder.h>
#include <frc/simulation/EncoderSim.h>

...

// These represent our regular encoder objects, which we would
// create to use on a real robot.
frc::Encoder m_leftEncoder{0, 1};
frc::Encoder m_rightEncoder{2, 3};

// These are our EncoderSim objects, which we will only use in
// simulation. However, you do not need to comment out these
// declarations when you are deploying code to the roboRIO.
frc::sim::EncoderSim m_leftEncoderSim{m_leftEncoder};
frc::sim::EncoderSim m_rightEncoderSim{m_rightEncoder};
```

39.2.2 Simulating Gyroscopes

Similar to the `EncoderSim` class, simulated gyroscope classes also exist for commonly used WPILib gyros - `AnalogGyroSim` and `ADXRS450_GyroSim`. These are also constructed in the same manner.

Note: It is not possible to simulate certain vendor gyros (i.e. Pigeon IMU and NavX) using WPILib classes. Please read the respective vendors' documentation for information on their simulation support.

Java

C++

```
// Create our gyro object like we would on a real robot.
private AnalogGyro m_gyro = new AnalogGyro(1);

// Create the simulated gyro object, used for setting the gyro
// angle. Like EncoderSim, this does not need to be commented out
// when deploying code to the roboRIO.
private AnalogGyroSim m_gyroSim = new AnalogGyroSim(m_gyro);
```

```
#include <frc/AnalogGyro.h>
#include <frc/simulation/AnalogGyroSim.h>

...

// Create our gyro object like we would on a real robot.
frc::AnalogGyro m_gyro{1};

// Create the simulated gyro object, used for setting the gyro
// angle. Like EncoderSim, this does not need to be commented out
// when deploying code to the roboRIO.
frc::sim::AnalogGyroSim m_gyroSim{m_gyro};
```

39.3 Step 2: Creating a Drivetrain Model

In order to accurately determine how your physical drivetrain will respond to given motor voltage inputs, an accurate model of your drivetrain must be created. This model is usually created by measuring various physical parameters of your real robot. In WPILib, this drivetrain simulation model is represented by the `DifferentialDrivetrainSim` class.

39.3.1 Creating a `DifferentialDrivetrainSim` from Physical Measurements

One way to creating a `DifferentialDrivetrainSim` instance is by using physical measurements of the drivetrain and robot – either obtained through CAD software or real-world measurements (the latter will usually yield better results as it will more closely match reality). This constructor takes the following parameters:

- The type and number of motors on one side of the drivetrain.
- The gear ratio between the motors and the wheels as output torque over input torque (this number is usually greater than 1 for drivetrains).
- The moment of inertia of the drivetrain (this can be obtained from a CAD model of your drivetrain. Usually, this is between 3 and 8 kgm^2).
- The mass of the drivetrain (it is recommended to use the mass of the entire robot itself, as it will more accurately model the acceleration characteristics of your robot for trajectory tracking).
- The radius of the drive wheels.
- The track width (distance between left and right wheels).

- Standard deviations of measurement noise: this represents how much measurement noise you expect from your real sensors. The measurement noise is an array with 7 elements, with each element representing the standard deviation of measurement noise in x, y, heading, left velocity, right velocity, left position, and right position respectively. This option can be omitted in C++ or set to null in Java if measurement noise is not desirable.

You can calculate the measurement noise of your sensors by taking multiple data points of the state you are trying to measure and calculating the standard deviation using a tool like Python. For example, to calculate the standard deviation in your encoders' velocity estimate, you can move your robot at a constant velocity, take multiple measurements, and calculate their standard deviation from the known mean. If this process is too tedious, the values used in the example below should be a good representation of average noise from encoders.

Note: The standard deviation of the noise for a measurement has the same units as that measurement. For example, the standard deviation of the velocity noise has units of m/s.

Note: It is very important to use SI units (i.e. meters and radians) when passing parameters in Java. In C++, the *units library* can be used to specify any unit type.

Java

C++

```
// Create the simulation model of our drivetrain.
DifferentialDrivetrainSim m_driveSim = new DifferentialDrivetrainSim(
    DCMotor.getNEO(2),      // 2 NEO motors on each side of the drivetrain.
    7.29,                  // 7.29:1 gearing reduction.
    7.5,                   // MOI of 7.5 kg m^2 (from CAD model).
    60.0,                  // The mass of the robot is 60 kg.
    Units.inchesToMeters(3), // The robot uses 3" radius wheels.
    0.7112,                // The track width is 0.7112 meters.

    // The standard deviations for measurement noise:
    // x and y:          0.001 m
    // heading:          0.001 rad
    // l and r velocity: 0.1 m/s
    // l and r position: 0.005 m
    VecBuilder.fill(0.001, 0.001, 0.001, 0.1, 0.1, 0.005, 0.005));
```

```
#include <frc/simulation/DifferentialDrivetrainSim.h>

...

// Create the simulation model of our drivetrain.
frc::sim::DifferentialDrivetrainSim m_driveSim{
    frc::DCMotor::GetNEO(2), // 2 NEO motors on each side of the drivetrain.
    7.29,                   // 7.29:1 gearing reduction.
    7.5_kg_sq_m,            // MOI of 7.5 kg m^2 (from CAD model).
    60_kg,                  // The mass of the robot is 60 kg.
    3_in,                   // The robot uses 3" radius wheels.
    0.7112_m,               // The track width is 0.7112 meters.

    // The standard deviations for measurement noise:
```

(continues on next page)

(continued from previous page)

```
// x and y:          0.001 m
// heading:          0.001 rad
// l and r velocity: 0.1  m/s
// l and r position: 0.005 m
{0.001, 0.001, 0.001, 0.1, 0.1, 0.005, 0.005}};
```

39.3.2 Creating a DifferentialDrivetrainSim from Characterization Gains

You can also use the gains produced by *robot characterization*, which you may have performed as part of setting up the trajectory tracking workflow outlined *here* to create a simulation model of your drivetrain and often yield results closer to real-world behavior than the method above.

Important: You must need two sets of Kv and Ka gains from the characterization tool - one from straight-line motion and the other from rotating in place. We will refer to these two sets of gains as linear and angular gains respectively.

This constructor takes the following parameters:

- A linear system representing the drivetrain - this can be created using the characterization gains.
- The track width (distance between the left and right wheels).
- The type and number of motors on one side of the drivetrain.
- The gear ratio between the motors and the wheels as output torque over input torque (this number is usually greater than 1 for drivetrains).
- The radius of the drive wheels.
- Standard deviations of measurement noise: this represents how much measurement noise you expect from your real sensors. The measurement noise is an array with 7 elements, with each element representing the standard deviation of measurement noise in x, y, heading, left velocity, right velocity, left position, and right position respectively. This option can be omitted in C++ or set to null in Java if measurement noise is not desirable.

You can calculate the measurement noise of your sensors by taking multiple data points of the state you are trying to measure and calculating the standard deviation using a tool like Python. For example, to calculate the standard deviation in your encoders' velocity estimate, you can move your robot at a constant velocity, take multiple measurements, and calculate their standard deviation from the known mean. If this process is too tedious, the values used in the example below should be a good representation of average noise from encoders.

Note: The standard deviation of the noise for a measurement has the same units as that measurement. For example, the standard deviation of the velocity noise has units of m/s.

Note: It is very important to use SI units (i.e. meters and radians) when passing parameters in Java. In C++, the *units library* can be used to specify any unit type.

Java

C++

```
// Create our feedforward gain constants (from the characterization
// tool)
static final double KvLinear = 1.98;
static final double KaLinear = 0.2;
static final double KvAngular = 1.5;
static final double KaAngular = 0.3;

// Create the simulation model of our drivetrain.
private DifferentialDrivetrainSim m_driveSim = new DifferentialDrivetrainSim(
    // Create a linear system from our characterization gains.
    LinearSystemId.identifyDrivetrainSystem(KvLinear, KaLinear, KvAngular, KaAngular),
    DCMotor.getNEO(2),           // 2 NEO motors on each side of the drivetrain.
    7.29,                       // 7.29:1 gearing reduction.
    0.7112,                     // The track width is 0.7112 meters.
    Units.inchesToMeters(3),    // The robot uses 3" radius wheels.

    // The standard deviations for measurement noise:
    // x and y:           0.001 m
    // heading:           0.001 rad
    // l and r velocity: 0.1 m/s
    // l and r position: 0.005 m
    VecBuilder.fill(0.001, 0.001, 0.001, 0.1, 0.1, 0.005, 0.005));
```

```
#include <frc/simulation/DifferentialDrivetrainSim.h>
#include <frc/system/plant/LinearSystemId.h>
#include <units/acceleration.h>
#include <units/angular_acceleration.h>
#include <units/angular_velocity.h>
#include <units/voltage.h>
#include <units/velocity.h>

...

// Create our feedforward gain constants (from the characterization
// tool). Note that these need to have correct units.
static constexpr auto KvLinear = 1.98_V / 1_mps;
static constexpr auto KaLinear = 0.2_V / 1_mps_sq;
static constexpr auto KvAngular = 1.5_V / 1_rad_per_s;
static constexpr auto KaAngular = 0.3_V / 1_rad_per_s_sq;

// Create the simulation model of our drivetrain.
frc::sim::DifferentialDrivetrainSim m_driveSim{
    // Create a linear system from our characterization gains.
    frc::LinearSystemId::IdentifyDrivetrainSystem(
        KvLinear, KaLinear, KvAngular, KaAngular),
    0.7112_m,           // The track width is 0.7112 meters.
    frc::DCMotor::GetNEO(2), // 2 NEO motors on each side of the drivetrain.
    7.29,              // 7.29:1 gearing reduction.
    3_in,              // The robot uses 3" radius wheels.

    // The standard deviations for measurement noise:
    // x and y:           0.001 m
    // heading:           0.001 rad
    // l and r velocity: 0.1 m/s
```

(continues on next page)

(continued from previous page)

```
// l and r position: 0.005 m
{0.001, 0.001, 0.001, 0.1, 0.1, 0.005, 0.005}};
```

39.3.3 Creating a DifferentialDrivetrainSim of the KoP Chassis

The `DifferentialDrivetrainSim` class also has a static `createKitbotSim()` (Java) / `CreateKitbotSim()` (C++) method that can create an instance of the `DifferentialDrivetrainSim` using the standard Kit of Parts Chassis parameters. This method takes 5 arguments, two of which are optional:

- The type and number of motors on one side of the drivetrain.
- The gear ratio between the motors and the wheels as output torque over input torque (this number is usually greater than 1 for drivetrains).
- The diameter of the wheels installed on the drivetrain.
- The moment of inertia of the drive base (optional).
- Standard deviations of measurement noise: this represents how much measurement noise you expect from your real sensors. The measurement noise is an array with 7 elements, with each element representing the standard deviation of measurement noise in x, y, heading, left velocity, right velocity, left position, and right position respectively. This option can be omitted in C++ or set to null in Java if measurement noise is not desirable.

You can calculate the measurement noise of your sensors by taking multiple data points of the state you are trying to measure and calculating the standard deviation using a tool like Python. For example, to calculate the standard deviation in your encoders' velocity estimate, you can move your robot at a constant velocity, take multiple measurements, and calculate their standard deviation from the known mean. If this process is too tedious, the values used in the example below should be a good representation of average noise from encoders.

Note: The standard deviation of the noise for a measurement has the same units as that measurement. For example, the standard deviation of the velocity noise has units of m/s.

Note: It is very important to use SI units (i.e. meters and radians) when passing parameters in Java. In C++, the [units library](#) can be used to specify any unit type.

Java

C++

```
private DifferentialDrivetrainSim m_driveSim = DifferentialDrivetrainSim.  
createKitbotSim(  
    KitbotMotor.kDualCIMPerSide, // 2 CIMs per side.  
    KitbotGearing.k10p71,        // 10.71:1  
    KitbotWheelSize.SixInch,     // 6" diameter wheels.  
    null                          // No measurement noise.  
);
```

```
#include <frc/simulation/DifferentialDrivetrainSim.h>

...

frc::sim::DifferentialDrivetrainSim m_driveSim =
    frc::sim::DifferentialDrivetrainSim::CreateKitbotSim(
        frc::sim::DifferentialDrivetrainSim::KitbotMotor::DualCIMPerSide, // 2 CIMs per
↪side.
        frc::sim::DifferentialDrivetrainSim::KitbotGearing::k10p71,         // 10.71:1
        frc::sim::DifferentialDrivetrainSim::KitbotWheelSize::kSixInch      // 6" diameter
↪wheels.
    );
```

Note: You can use the `KitbotMotor`, `KitbotGearing`, and `KitbotWheelSize` enum (Java) / struct (C++) to get commonly used configurations of the Kit of Parts Chassis.

Important: Constructing your `DifferentialDrivetrainSim` instance in this way is just an approximation and is intended to get teams quickly up and running with simulation. Using empirical values measured from your physical robot will always yield more accurate results.

39.4 Step 3: Updating the Drivetrain Model

Now that the drivetrain model has been made, it needs to be updated periodically with the latest motor voltage commands. It is recommended to do this step in a separate `simulationPeriodic()` / `SimulationPeriodic()` method inside your subsystem and only call this method in simulation.

Note: If you are using the command-based framework, every subsystem that extends `SubsystemBase` has a `simulationPeriodic()` / `SimulationPeriodic()` which can be overridden. This method is automatically run only during simulation. If you are not using the command-based library, make sure you call your simulation method inside the overridden `simulationPeriodic()` / `SimulationPeriodic()` of the main `Robot` class. These periodic methods are also automatically called only during simulation.

There are three main steps to updating the model:

1. Set the *input* of the drivetrain model. These are the motor voltages from the two sides of the drivetrain.
2. Advance the model forward in time by the nominal periodic timestep (Usually 20 ms). This updates all of the drivetrain's states (i.e. pose, encoder positions and velocities) as if 20 ms had passed.
3. Update simulated sensors with new positions, velocities, and angles to use in other places.

Java

C++

```

private PWMSparkMax m_leftMotor = new PWMSparkMax(0);
private PWMSparkMax m_rightMotor = new PWMSparkMax(1);

public Drivetrain() {
    ...
    m_leftEncoder.setDistancePerPulse(2 * Math.PI * kWheelRadius / kEncoderResolution);
    m_rightEncoder.setDistancePerPulse(2 * Math.PI * kWheelRadius / kEncoderResolution);
}

public void simulationPeriodic() {
    // Set the inputs to the system. Note that we need to convert
    // the [-1, 1] PWM signal to voltage by multiplying it by the
    // robot controller voltage.
    m_driveSim.setInputs(m_leftMotor.get() * RobotController.getInputVoltage(),
                        m_rightMotor.get() * RobotController.getInputVoltage());

    // Advance the model by 20 ms. Note that if you are running this
    // subsystem in a separate thread or have changed the nominal timestep
    // of TimedRobot, this value needs to match it.
    m_driveSim.update(0.02);

    // Update all of our sensors.
    m_leftEncoderSim.setDistance(m_driveSim.getLeftPositionMeters());
    m_leftEncoderSim.setRate(m_driveSim.getLeftVelocityMetersPerSecond());
    m_rightEncoderSim.setDistance(m_driveSim.getRightPositionMeters());
    m_rightEncoderSim.setRate(m_driveSim.getRightVelocityMetersPerSecond());
    m_gyroSim.setAngle(-m_driveSim.getHeading().getDegrees());
}

```

```

frc::PWMSparkMax m_leftMotor{0};
frc::PWMSparkMax m_rightMotor{1};

Drivetrain() {
    ...
    m_leftEncoder.SetDistancePerPulse(2 * wpi::math::pi * kWheelRadius /
    ↪kEncoderResolution);
    m_rightEncoder.SetDistancePerPulse(2 * wpi::math::pi * kWheelRadius /
    ↪kEncoderResolution);
}

void SimulationPeriodic() {
    // Set the inputs to the system. Note that we need to convert
    // the [-1, 1] PWM signal to voltage by multiplying it by the
    // robot controller voltage.
    m_driveSim.SetInputs(
        m_leftMotor.get() * units::volt_t(frc::RobotController::GetInputVoltage()),
        m_rightMotor.get() * units::volt_t(frc::RobotController::GetInputVoltage()));

    // Advance the model by 20 ms. Note that if you are running this
    // subsystem in a separate thread or have changed the nominal timestep
    // of TimedRobot, this value needs to match it.
    m_driveSim.Update(20_ms);

    // Update all of our sensors.
    m_leftEncoderSim.SetDistance(m_driveSim.GetLeftPosition().to<double>());
    m_leftEncoderSim.SetRate(m_driveSim.GetLeftVelocity().to<double>());
    m_rightEncoderSim.SetDistance(m_driveSim.GetRightPosition().to<double>());
}

```

(continues on next page)

(continued from previous page)

```

m_rightEncoderSim.SetRate(m_driveSim.GetRightVelocity().to<double>());
m_gyroSim.SetAngle(-m_driveSim.GetHeading().Degrees());
}

```

Important: If the right side of your drivetrain is inverted, you MUST negate the right voltage in the `SetInputs()` call to ensure that positive voltages correspond to forward movement.

Important: Because the drivetrain simulator model returns positions and velocities in meters and m/s respectively, these must be converted to encoder ticks and ticks/s when calling `SetDistance()` and `SetRate()`. Alternatively, you can configure `SetDistancePerPulse` on the encoders to have the Encoder object take care of this automatically – this is the approach that is taken in the example above.

Now that the simulated encoder positions, velocities, and gyroscope angles have been set, you can call `m_leftEncoder.GetDistance()`, etc. in your robot code as normal and it will behave exactly like it would on a real robot. This involves performing odometry calculations, running velocity PID feedback loops for trajectory tracking, etc.

39.5 Step 4: Updating Odometry and Visualizing Robot Position

Now that the simulated encoder positions, velocities, and gyro angles are being updated with accurate information periodically, this data can be used to update the pose of the robot in a periodic loop (such as the `periodic()` method in a Subsystem). In simulation, the periodic loop will use simulated encoder and gyro readings to update odometry whereas on the real robot, the same code will use real readings from physical hardware.

Note: For more information on using odometry, see [this document](#).

39.5.1 Robot Pose Visualization

The robot pose can be visualized on the Simulator GUI (during simulation) or on a dashboard such as Glass (on a real robot) by sending the odometry pose over a `Field2d` object. A `Field2d` can be trivially constructed without any constructor arguments:

Java

C++

```
private Field2d m_field = new Field2d();
```

```
#include <frc/smartdashboard/Field2d.h>
```

```
..
```

```
frc::Field2d m_field;
```

This Field2d instance must then be sent over NetworkTables. The best place to do this is in the constructor of your subsystem.

Java

C++

```
public Drivetrain() {  
    ...  
    SmartDashboard.putData("Field", m_field);  
}
```

```
#include <frc/smartdashboard/SmartDashboard.h>  
  
Drivetrain() {  
    ...  
    frc::SmartDashboard::PutData("Field", &m_field);  
}
```

Note: The Field2d instance can also be sent using a lower-level NetworkTables API or using the [Shuffleboard API](#).

Finally, the pose from your odometry must be updated periodically into the Field2d object. Remember that this should be in a general periodic() method i.e. one that runs both during simulation and during real robot operation.

Java

C++

```
public void periodic() {  
    ...  
    // This will get the simulated sensor readings that we set  
    // in the previous article while in simulation, but will use  
    // real values on the robot itself.  
    m_odometry.update(m_gyro.getRotation2d(),  
                     m_leftEncoder.getDistance(),  
                     m_rightEncoder.getDistance());  
    m_field.setRobotPose(m_odometry.getPoseMeters());  
}
```

```
void Periodic() {  
    ...  
    // This will get the simulated sensor readings that we set  
    // in the previous article while in simulation, but will use  
    // real values on the robot itself.  
    m_odometry.Update(m_gyro.GetRotation2d(),  
                     units::meter_t(m_leftEncoder.GetDistance()),  
                     units::meter_t(m_rightEncoder.GetDistance()));  
    m_field.SetRobotPose(m_odometry.GetPose());  
}
```

Important: It is important that this code is placed in a regular periodic() method - one that is called periodically regardless of mode of operation. If you are using the command-based library, this method already exists. If not, you are responsible for calling this method

periodically from the main Robot class.

Note: At this point we have covered all of the code changes required to run your code. You should head to the [Simulation User Interface page](#) for more info on how to run the simulation and add the field that your simulated robot will run on to the GUI.

40.1 Wiring Best Practices

Tip: The article [Wiring the FRC Control System](#) walks through the details of what connects where to wire up the FRC Control System and this article provides some additional “Best Practices” that may increase reliability and make maintenance easier. Take a look at [Preemptive Troubleshooting](#) for more tips and tricks.

40.1.1 Vibration/Shock

An FRC® Robot is an incredibly rough environment when it comes to vibrations and shock loads. While many of the FRC specific electronics are extensively tested for mechanical robustness in these conditions, a few components, such as the radio, are not specifically designed for use on a mobile platform. Taking steps to reduce the shock and vibration these components are exposed to may help reduce failures. Some suggestions that may reduce mechanical failures:

- **Vibration Isolation** - Make sure to isolate any components which create excessive vibration, such as compressors, using “vibration isolators”. This will help reduce vibration on the robot which can loosen fasteners and cause premature fatigue failure on some electronic components.
- **Bumpers** - Use Bumpers to cover as much of the robot as possible for your design. While the rules require specific bumper coverage around the corners of your robot, maximizing the use of bumpers increases the likelihood that all collisions will be damped by your bumpers. Bumpers significantly reduce the g-forces experienced in a collision compared to hitting directly on a hard robot surface, reducing the shock experienced by the electronics and decreasing the chance of a shock related failure.
- **Shock Mounting** - You may choose to shock mount some or all of your electronic components to further reduce the forces they see in robot collisions. This is especially helpful for the robot radio and other electronics such as co-processors, which may not be designed for use on mobile platforms. Vibration isolators, springs, foams, or mounting to flexible materials all may reduce the shock forces seen by these components.

40.1.2 Redundancy

Unfortunately there are few places in the FRC Control System where redundancy is feasible. Taking advantage of opportunities for redundancy can increase reliability. The primary example of this is wiring the barrel connector to the radio in addition to the provided PoE connection. This ensures that if one of the cables becomes damaged or dislodged, the other will maintain power to the radio. Keep an eye out for other potential areas to provide redundancy when wiring and programming your robot.

40.1.3 Port Savers

For any connections on the Robot or Driver station that may be frequently plugged and unplugged (such as DS joysticks, DS Ethernet, roboRIO USB tether, and Ethernet tether) using a “Port Saver” or “pigtail” can substantially reduce the potential for damaging the port. This type of device can serve double duty, both reducing the number of cycles that the port on the electronic device sees, as well as relocating the connection to a more convenient location. Make sure to secure the port saver (see the next item) to avoid port damage.

40.1.4 Wire Management and Strain Relief

One of the most critical components to robot reliability and maintenance is good wire management and strain relief. Good wire management is comprised of a few components:

- Make sure cables are the correct length. Any excess wire length is just more to manage. If you must have extra wire due to additional length on COTS cabling, secure the extra into a small bundle using separate cable ties before securing the rest of the wire.
- Ensure that cables are secured close to connection points, with enough slack to avoid putting strain on connectors. This is called strain relief, and is critical to minimizing the likelihood that a cable comes unplugged or a wire breaks off at a connection point (these are generally stress concentrators).
- Secure cables near any moving components. Make sure that all wire runs are secure and protected from moving components, even if the moving components were to bend or over-travel.
- Secure cables at additional points as necessary to keep wiring neat and clean. Take care to not over secure wires; if wires are secured in too many locations, it may actually make troubleshooting and maintenance more difficult.

40.1.5 Documentation

A great way to make maintenance easier is to create documentation describing what is connected where on the robot. There are a number of ways of creating this type of documentation which range from complete wiring diagrams to excel charts to a quick list of what functions are attached to which channels. Many teams also integrate these lists with labeling (see the next bullet).

When a wire is accidentally cut, or a mechanism is malfunctioning, or a component burns out, it will be much easier to repair if you have some documentation to tell you what is connected where without having to trace the wiring all the way through (even if your wiring is neat!)

40.1.6 Labeling

Labeling is a great way to supplement the wiring documentation described above. There are many different strategies to labeling wiring and electronics, all with their own pros and cons. Labels for electronics and flags for wires can be made by hand, or using a label maker (some can also do heat-shrink labels), or you can use different colors of electrical tape or labeling flags to indicate different things. Whatever system you choose, make sure you understand how it complements your documentation and make sure everyone on your team is familiar with it.

40.1.7 Check all wiring and connections

After all wiring on the robot is complete, make sure to check each connection, pulling on each, to ensure that everything is secure. Additionally, ensure that no stray wire “whiskers” are sticking out of any connection point and that no uninsulated connections are exposed. If any connections come loose while testing, or any “whiskers” are discovered, re-make the connection and make sure to have a second person check it when complete.

A common source of poor connections is screw-type or nut-and-bolt fasteners. For any connections of this type on the robot (e.g. battery connections, main breaker, PDP, roboRIO), make sure the fasteners are tight. For nut-and-bolt style connections, ensure that the wire/terminal cannot be rotated by hand; if you can rotate your battery wire or main breaker connection by grasping the terminal and twisting, the connection is not tight enough.

Another common source of failures is the fuses at the end of the PDP. Ensure these fuses are completely seated; you may need to apply more force than you expect to seat them completely. If the fuses are seated properly they will likely be difficult or impossible to remove by hand.

Snap-in connections such as the SB-50 connector should be secured using clips or cable ties to ensure they do not pop loose during impacts.

40.1.8 Re-Check Early and Often

Re-check the entire electrical system as thoroughly as possible after playing the first match or two (or doing very vigorous testing). The first few impacts the robot sees may loosen fasteners or expose issues.

Create a checklist for re-checking electrical connections on a regular basis. As a very rough starting point, rotational fasteners such as battery and PDP connections should be checked every 1-3 matches. Spring type connections such as the WAGO and Weidmuller connectors likely only need to be checked once per event. Ensure that the team knows who is responsible for completing the checklist and how they will document that it has been done.

40.1.9 Battery Maintenance

Take good care of your batteries! A bad battery can easily cause a robot to function poorly, or not at all, during a match. Label all of your batteries to help keep track of usage during the event. Many teams also include information such as the age of the battery on this label.

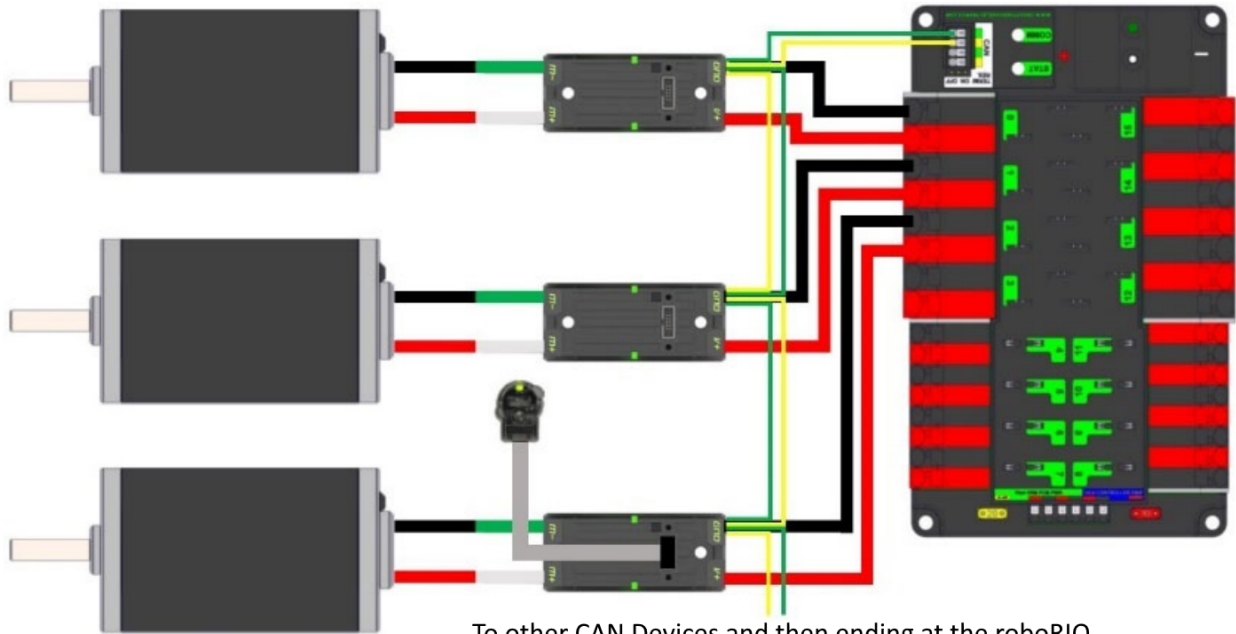
- Never lift or carry batteries by the wires! Carrying batteries by the wires has the potential to damage the internal connection between the terminals and the plates, dramatically increasing internal resistance and degrading performance.
- Mark any dropped battery bad until a complete test can be conducted. In addition to the mentioned terminal connections, dropping a battery also has the potential to damage individual cells. This damage may not register on a simple voltage test, instead hiding until the battery is placed under load.
- Rotate batteries evenly. This helps ensure that batteries have the most time to charge and rest and that they wear evenly (equal number of charge/discharge cycles)
- Load test batteries if possible to monitor health. There are a number of commercially available products teams use to load test batteries, including at least one designed specifically for FRC. A load test can provide an indicator of battery health by measuring internal resistance. This measurement is much more meaningful when it comes to match performance than a simple no-load voltage number provided by a multimeter.

40.1.10 Check DS Logs

After each match, review the DS logs to see what the battery voltage and current usage looks like. Once you have established what the normal range of these items is for your robot, you may be able to spot potential issues (bad batteries, failing motors, mechanical binding) before they become critical failures.

40.2 CAN Wiring Basics

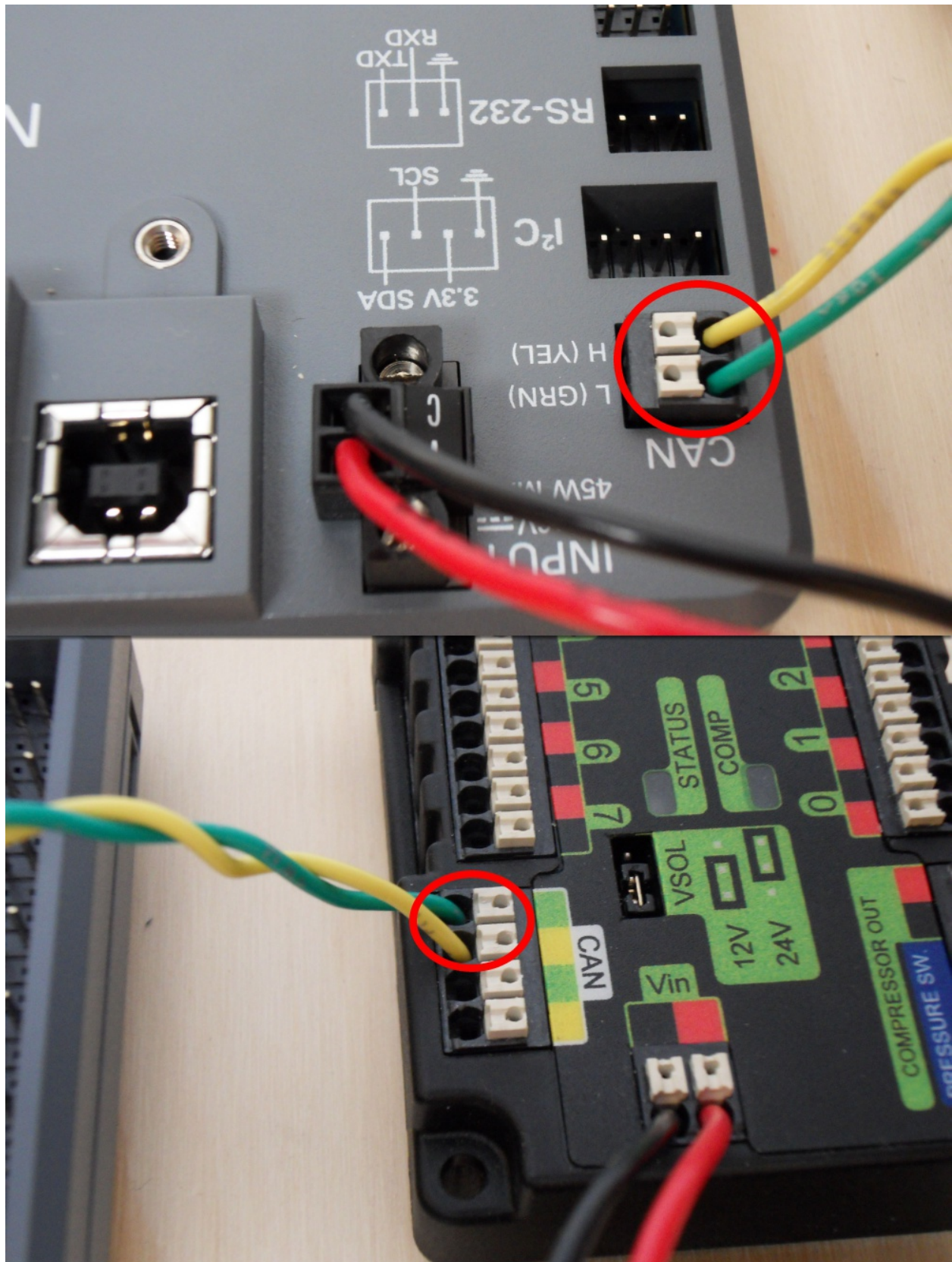
CAN is a two wire network that is designed to facilitate communication between multiple devices on your robot. It is recommended that CAN on your robot follow a “daisy-chain” topology. This means that the CAN wiring should usually start at your roboRIO and go into and out of each device successively until finally ending at the PDP.



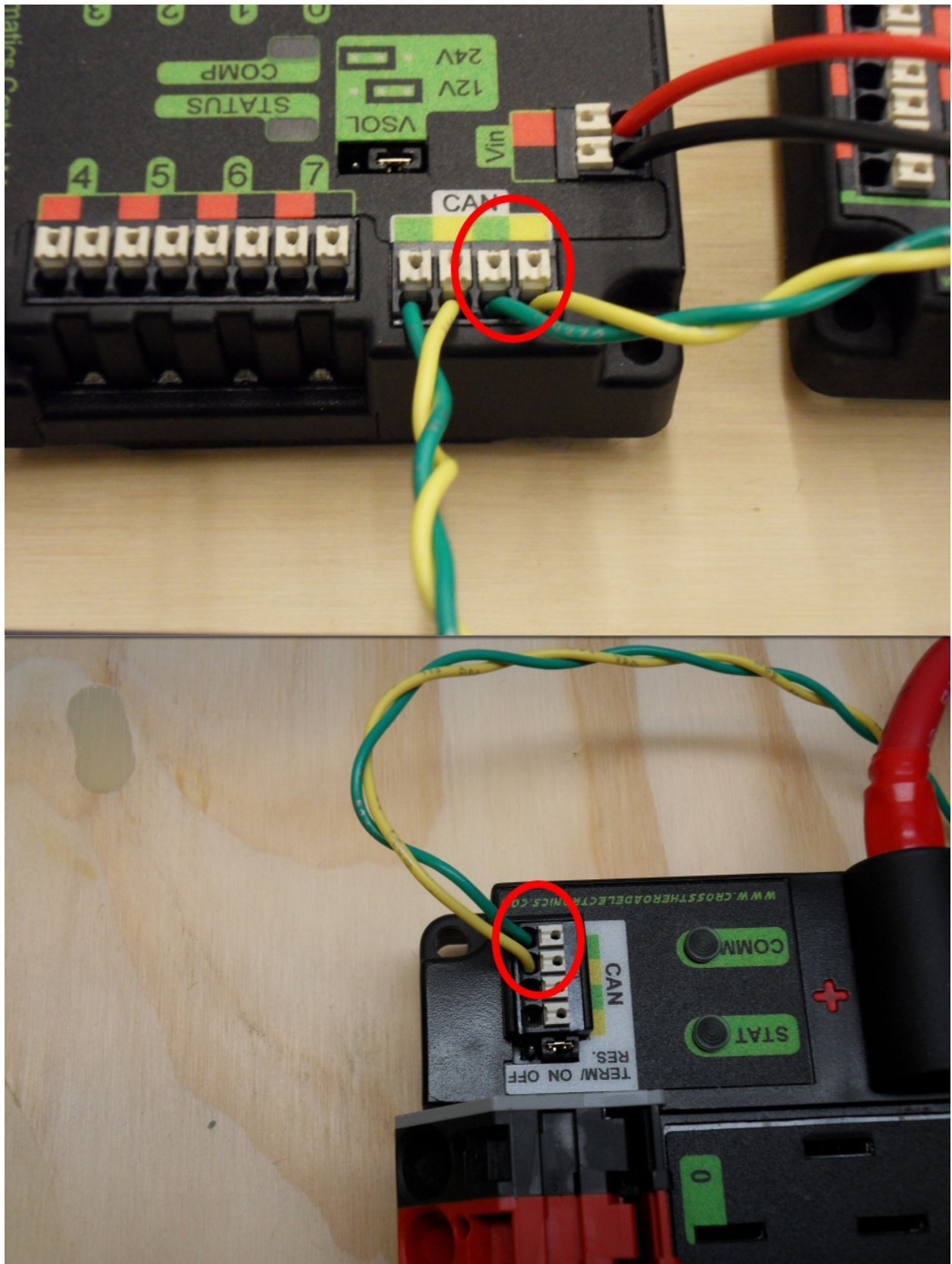
40.2.1 Standard Wiring

CAN is generally wired with yellow and green wire with yellow acting as the CAN-High and green as the CAN-Low signals. Many devices show this yellow and green color scheme to indicate how the wires should be plugged in.

CAN wiring from the roboRIO to the PCM.



CAN wiring from the PCM to the PDP.



40.2.2 Termination

It is recommended that the wiring starts at the roboRIO and ends at the PDP because the CAN network is required to be terminated by 120 Ω resistors and these are built into these two devices. The PDP ships with the CAN bus terminating resistor jumper in the “ON” position. It is recommended to leave the jumper in this position and place any additional CAN nodes between the roboRIO and the PDP (leaving the PDP as the end of the bus). If you wish to place the PDP in the middle of the bus (utilizing both pairs of PDP CAN terminals) move the jumper to the “OFF” position and place your own 120 Ω terminating resistor at the end of your CAN bus chain.

40.3 Wiring Pneumatics

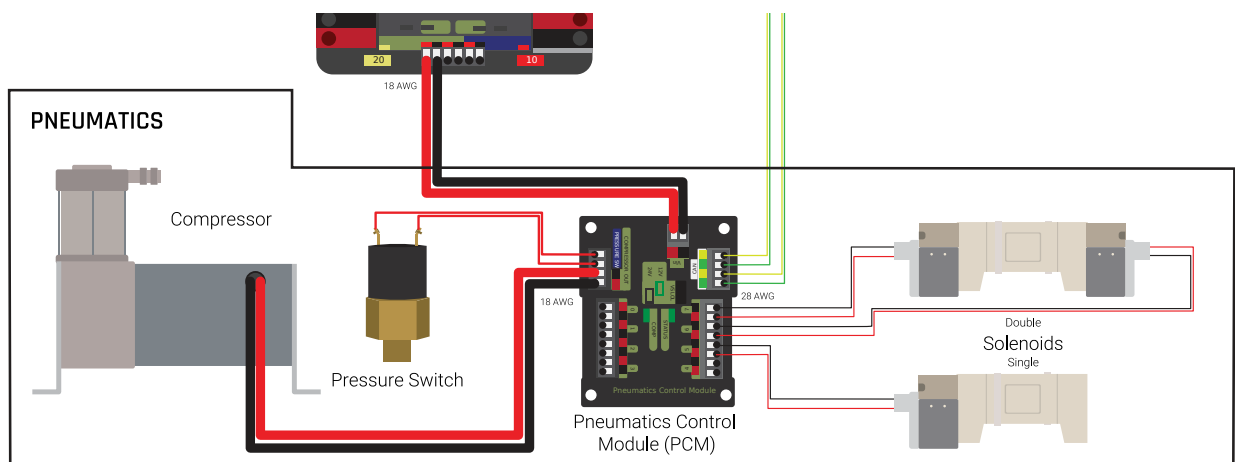
Hint: For pneumatics safety & mechanical requirements, consult this year’s Robot Construction rules. For mechanical design guidelines, the FIRST Pneumatics Manual is located [here](#) (last updated 2017)

40.3.1 Wiring Overview

A single PCM will support most pneumatics applications, providing an output for the compressor, input for the pressure switch, and outputs for up to 8 solenoid channels (12V or 24V selectable). The module is connected to the roboRIO over the CAN bus and powered via 12V from the PDP.

For complicated robot designs requiring more channels or multiple solenoid voltages, additional PCMs can be added to the control system.

40.3.2 PCM Power and Control Wiring



The first PCM on your robot can be wired from the PDP VRM/PCM connectors on the end of the PDP. The PCM is connected to the roboRIO via CAN and can be placed anywhere in the

middle of the CAN chain (or on the end with a custom terminator). For more details on wiring a single PCM, see *Pneumatics Control Module Power (Optional)*

Additional PCMs can be wired to a standard WAGO connector on the side of the PDP and protected with a 20A or smaller circuit breaker. Additional PCMs should also be placed anywhere in the middle of the CAN chain.

40.3.3 The Compressor

The compressor can be wired directly to the Compressor Out connectors on the PCM. If additional length is required, make sure to use 18 AWG wire or larger for the extension.

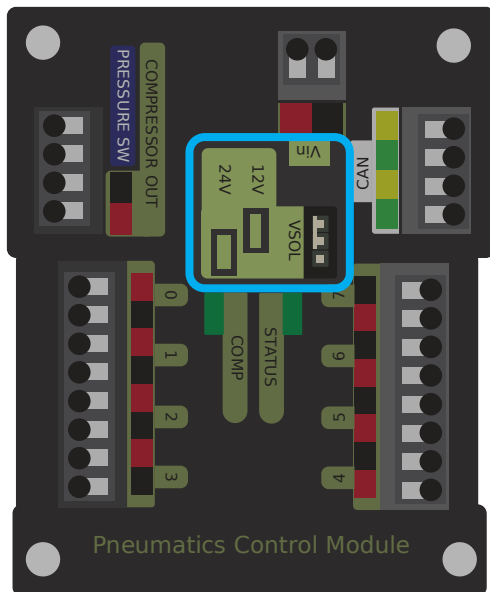
40.3.4 The Pressure Switch

The pressure switch should be connected directly to the pressure switch input terminals on the PCM. There is no polarity on the input terminals or on the pressure switch itself, either terminal on the PCM can be connected to either terminal on the switch. Ring or spade terminals are recommended for the connection to the switch screws (note that the screws are slightly larger than #6, but can be threaded through a ring terminal with a hole for a #6 screw such as the terminals shown in the image).

40.3.5 Solenoids

Each solenoid channel should be wired directly to a numbered pair of terminals on the PCM. A single acting solenoid will use one numbered terminal pair. A double acting solenoid will use two pairs. If your solenoid does not come with color coded wiring, check the datasheet to make sure to wire with the proper polarity.

40.3.6 Solenoid Voltage Jumper



The PCM is capable of powering either 12V or 24V solenoids, but all solenoids connected to a single PCM must be the same voltage. The PCM ships with the jumper in the 12V position as shown in the image. To use 24V solenoids move the jumper from the left two pins (as shown in the image) to the right two pins. The overlay on the PCM also indicates which position corresponds to which voltage. You may need to use a tool such as a small screwdriver, small pair of pliers, or a pair of tweezers to remove the jumper.

40.4 Status Light Quick Reference

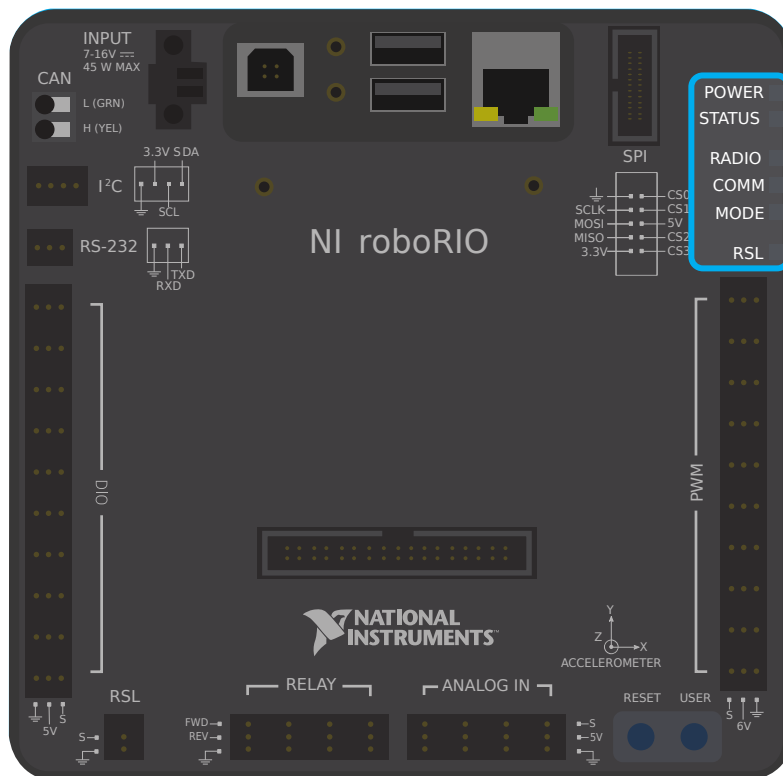
Many of the components of the FRC® Control System have indicator lights that can be used to quickly diagnose problems with your robot. This guide shows each of the hardware components and describes the meaning of the indicators. Photos and information from Innovation FIRST and Cross the Road Electronics.

40.4.1 Robot Signal Light (RSL)



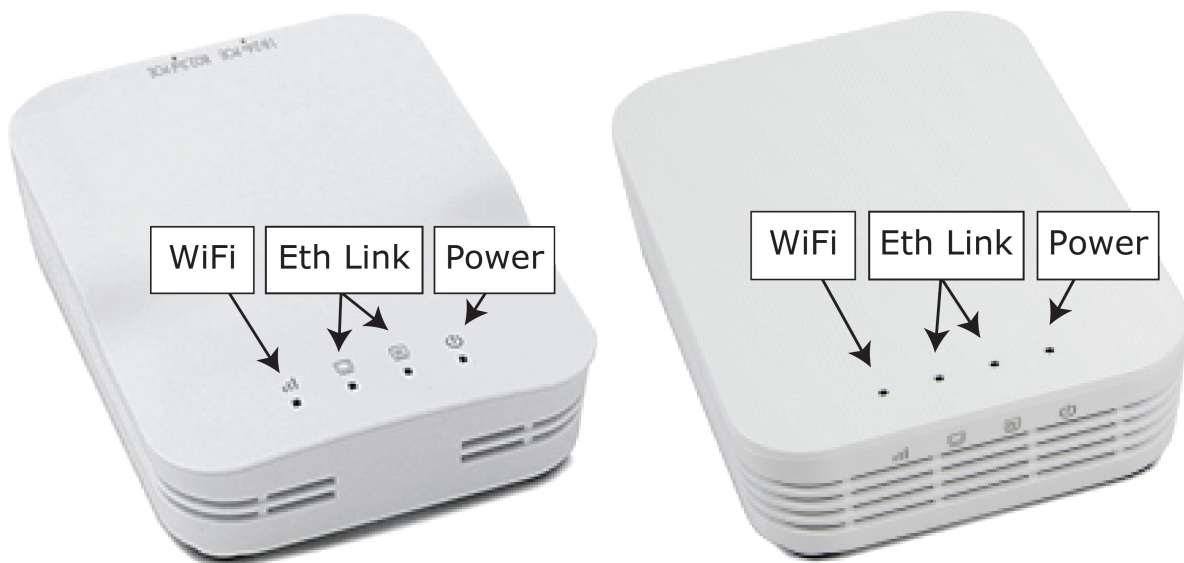
Solid ON	Robot On and Disabled
Blinking	Robot On and Enabled
Off	Robot Off, roboRIO not powered or RSL not wired properly

40.4.2 roboRIO



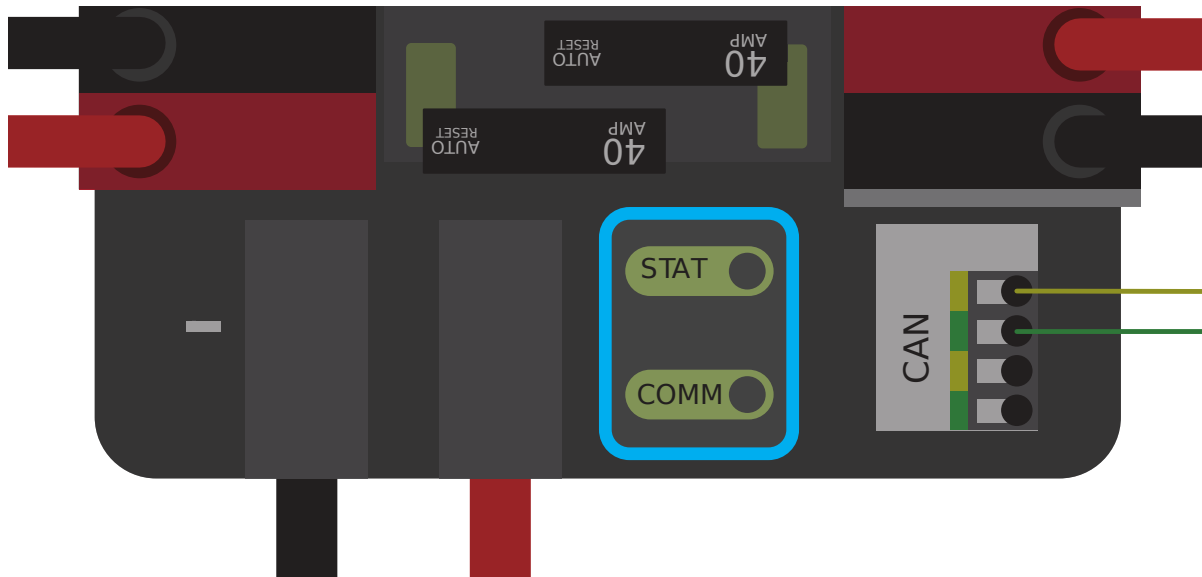
Power	Green	Power is good
	Amber	Brownout protection tripped, outputs disabled
	Red	Power fault, check user rails for short circuit
Status	On while the controller is booting, then should turn off	
	2 blinks	Software error, reimage roboRIO
	3 blinks	Safe Mode, restart roboRIO, reimage if not resolved
	4 blinks	Software crashed twice without rebooting, reboot roboRIO, reimage if not resolved
	Constant flash or stays solid on	Unrecoverable error
Radio	Not currently implemented	
Comm	Off	No Communication
	Red Solid	Communication with DS, but no user code running
	Red Blinking	E-stop triggered
	Green Solid	Good communications with DS
Mode	Off	Outputs disabled (robot in Disabled, brown-out, etc.)
	Orange	Autonomous Enabled
	Green	Teleop Enabled
	Red	Test Enabled
RSL	<i>See above</i>	

40.4.3 OpenMesh Radio



Power	Blue	On or Powering up
	Blue Blinking	Powering Up
Eth Link	Blue	Link up
	Blue Blinking	Traffic Present
WiFi	Off	Bridge mode, Unlinked or non-FRC firmware
	Red	AP, Unlinked
	Yellow/Orange	AP, Linked
	Green	Bridge mode, Linked

40.4.4 Power Distribution Panel



PDP Status/Comm LEDs

LED	Strobe	Slow
Green	No Fault - Robot Enabled	No Fault - Robot Disabled
Orange	NA	Sticky Fault
Red	NA	No CAN Comm

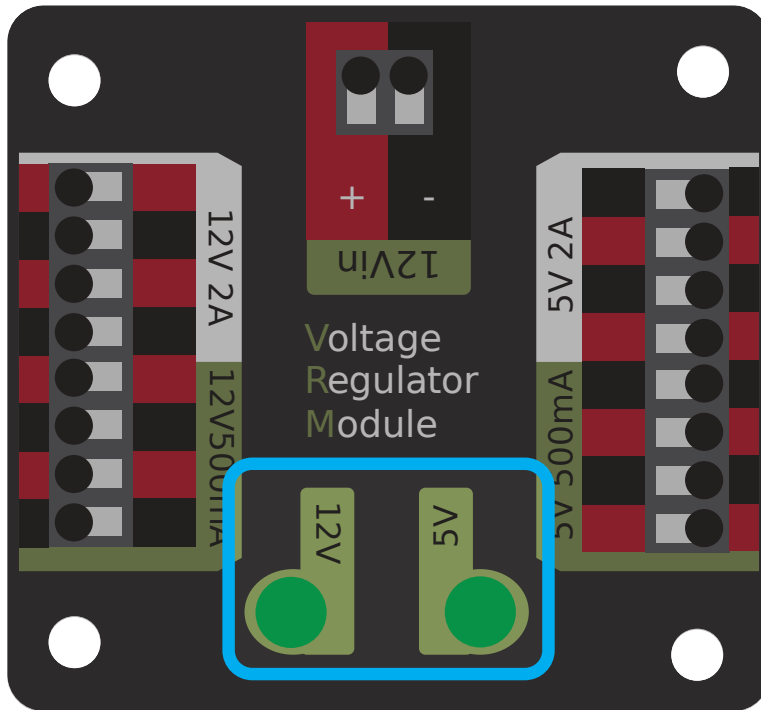
Tip: If a PDP LED is showing more than one color, see the PDP LED special states table below. For more information on resolving PDP faults see the PDP User Manual.

Note: Note that the No CAN Comm fault will occur if the PDP cannot communicate with the roboRIO via CAN Bus.

PDP Special States

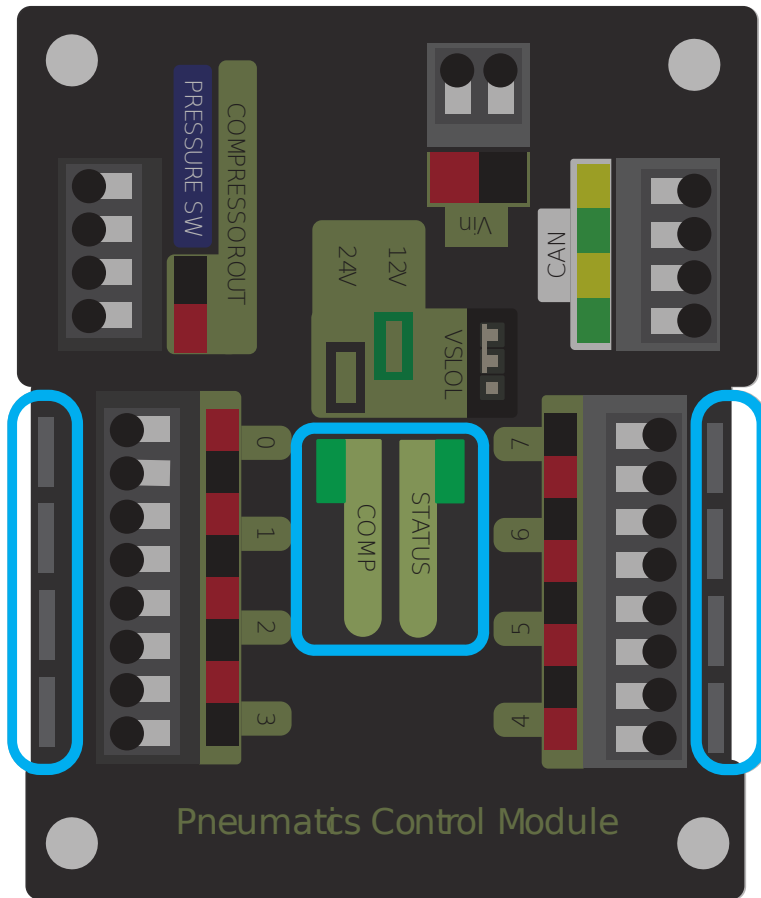
LED Colors	Problem
Red/Orange	Damaged Hardware
Green/Orange	In Bootloader
No LED	No Power/ Incorrect Polarity

40.4.5 Voltage Regulator Module



The status LEDs on the VRM indicate the state of the two power supplies. If the supply is functioning properly the LED should be lit bright green. If the LED is not lit or is dim, the output may be shorted or drawing too much current.

40.4.6 Pneumatics Control Module (PCM)



PCM Status LED

LED	Strobe	Slow	Long
Green	No Fault Robot Enabled	Sticky Fault	NA
Orange	NA	Sticky Fault	NA
Red	NA	No CAN Comm or Solenoid Fault (Blinks Solenoid Index)	Compressor Fault

Tip: If a PCM LED is showing more than one color, see the PCM LED special states table below. For more information on resolving PCM faults see the PCM User Manual.

Note: Note that the No CAN Comm fault will not occur only if the device cannot communicate with any other device, if the PCM and PDP can communicate with each other, but not the

roboRIO.

PCM LED Special States Table

LED	Problems
Red/Orange	Damaged Hardware
Green/Orange	In Bootloader
No LED	No Power/Incorrect Polarity

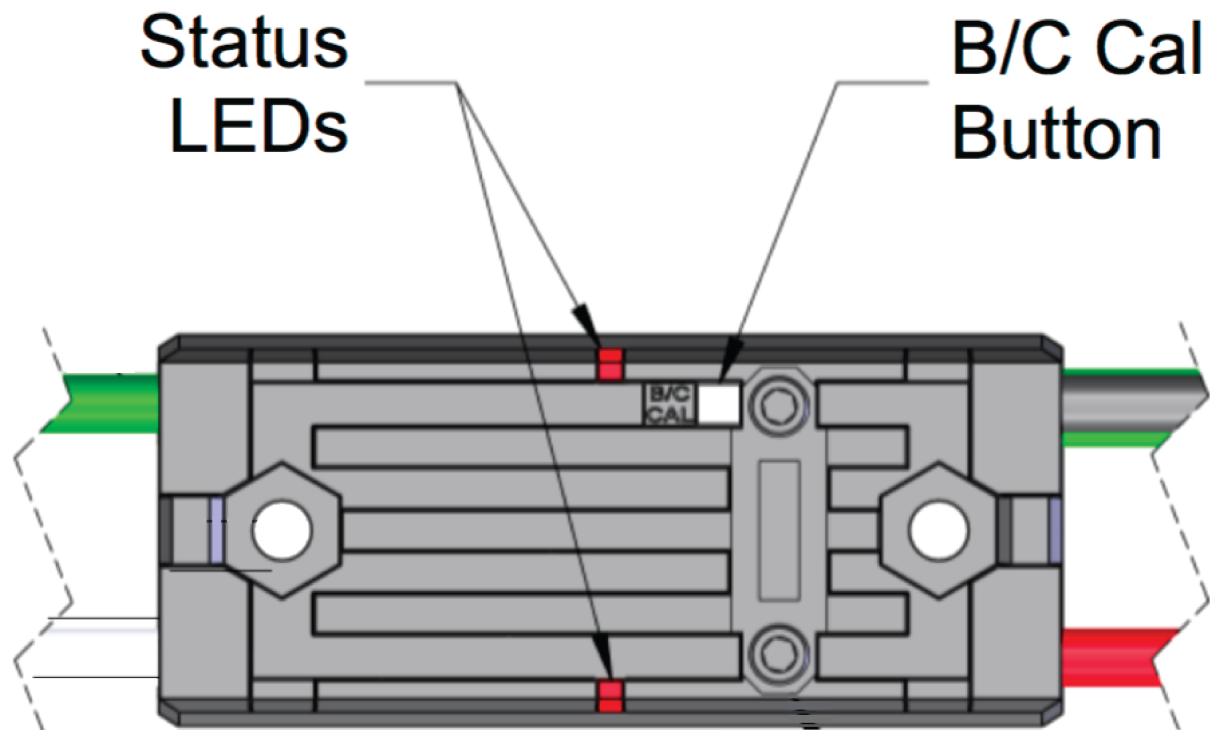
PCM Comp LED

This is the Compressor LED. This LED is green when the compressor output is active (compressor is currently on) and off when the compressor output is not active.

PCM Solenoid Channel LEDs

These LEDs are lit red if the Solenoid channel is enabled and not lit if it is disabled.

40.4.7 Talon SRX & Victor SPX & Talon FX Motor Controllers



Status LEDs During Normal Operation

LEDs	Colors	Talon SRX State
Both	Blinking Green	Forward throttle is applied. Blink rate is proportional to Duty Cycle.
Both	Blinking Red	Reverse throttle is applied. Blink rate is proportional to Duty Cycle.
None	None	No power is being applied to Talon SRX
LEDs Alternate	Off/Orange	CAN bus detected, robot disabled
LEDs Alternate	Off/Slow Red	CAN bus/PWM is not detected
LEDs Alternate	Off/Fast Red	Fault Detected
LEDs Alternate	Red/Orange	Damaged Hardware
LEDs Strobe towards (M-)	Off/Red	Forward Limit Switch or Forward Soft Limit
LEDs Strobe towards (M+)	Off/Red	Reverse Limit Switch or Reverse Soft Limit
LED1 Only (closest to M+/V+)	Green/Orange	In Boot-loader
LEDs Strobe towards (M+)	Off/Orange	Thermal Fault / Shutoff (Talon FX Only)

















Status LEDs During Calibration

Status LEDs Blink Code	Talon SRX State
Flashing Red/Green	Calibration Mode
Blinking Green	Successful Calibration
Blinking Red	Failed Calibration

B/C CAL Blink Codes

B/C CAL Button Color	Talon SRX State
Solid Red	Brake Mode
Off	Coast Mode

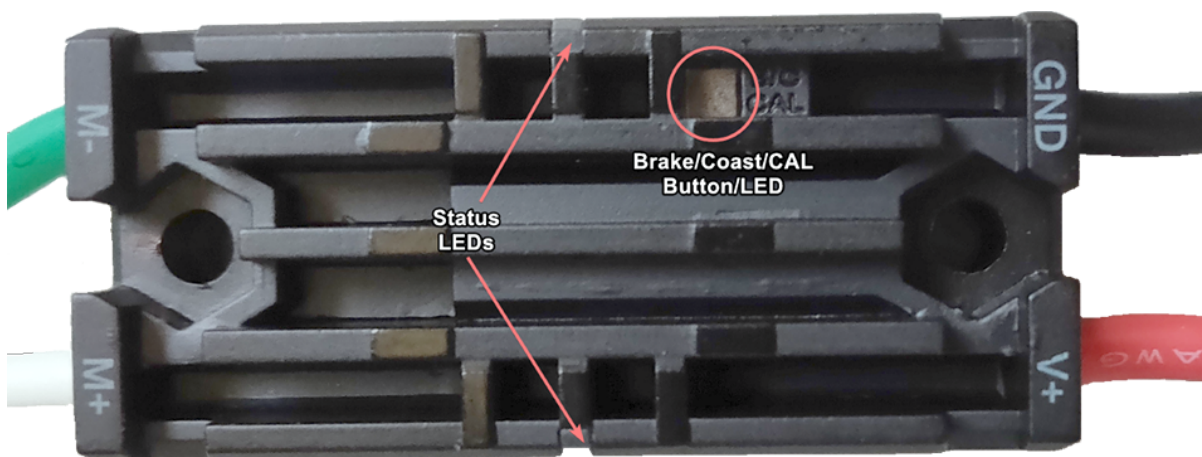
40.4.8 SPARK-MAX Motor Controller

Operating Mode	Idle Mode	State	Color/Pattern	
Brushed	Brake	No Signal	Blue Blink	
		Valid Signal	Blue Solid	
	Coast	No Signal	Yellow Blink	
		Valid Signal	Yellow Solid	
Brushless	Brake	No Signal	Cyan Blink	
		Valid Signal	Cyan Solid	
	Coast	No Signal	Magenta Blink	
		Valid Signal	Magenta Solid	
Partial Forward	-	-	Green Blink	
Full Forward	-	-	Green Solid	
Partial Reverse	-	-	Red Blink	
Full Reverse	-	-	Red Solid	
Forward Limit	-	-	Green/White Blink	
Reverse Limit	-	-	Red/White Blink	
Firmware Update Mode	-	-	Dark (LED off)	
Fault Conditions				
12V Missing	-	-	Orange/Blue Slow Blink	
Brushless Encoder Error	-	-	Orange/Magenta Slow Blink	
Gate Driver Fault	-	-	Orange/Cyan Slow Blink	
CAN Fault	-	-	Orange/Yellow Slow Blink	
10.4 Status Light Quick Reference				

40.4.9 REV Robotics SPARK

		LED Status Code	
Time Scale		1 second	1 second
State		Normal Operation	
No Signal	Brake		
	Coast		
Full Forward			
Proportional Forward			
Neutral	Brake		
	Coast		
Proportional Reverse			
Full Reverse			
Forward Limit Tripped			
Reverse Limit Tripped			
		Calibration	
Calibration Mode			
Successful Calibration			
Failed Calibration			
		Factory Reset	
		Mode button held during power up	Mode button released
Reset to Factory Defaults			

40.4.10 Victor-SP Motor Controller



Brake/Coast/Cal Button/LED - Red if the controller is in brake mode, off if the controller is in coast mode

Status

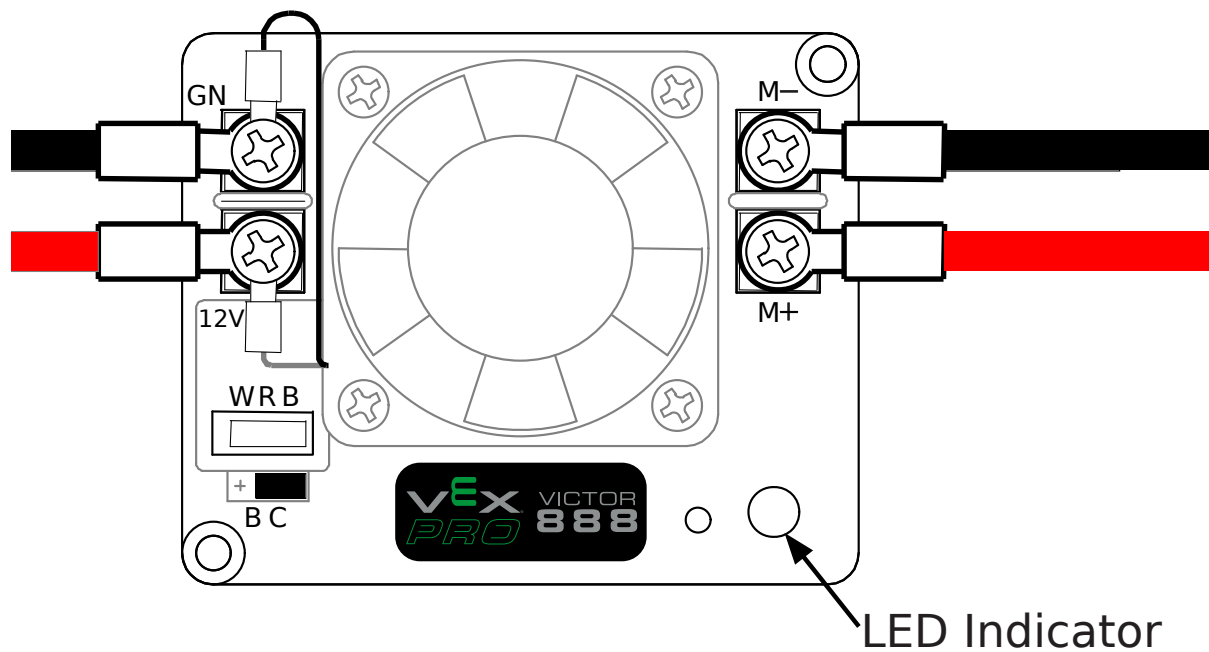
Green	Solid	Full forward output
	Blinking	Proportional to forward output voltage
Red	Solid	Full reverse output
	Blinking	Proportional to forward output voltage
Orange	Solid	FRC robot disabled, PWM signal lost, or signal in deadband range (+/- 4% output)
Red/Green	Blinking	Ready for calibration. Several green flashes indicates successful calibration, and red several times indicates unsuccessful calibration.

40.4.11 Talon Motor Controller



Green	Solid	Full forward output
	Blinking	Proportional to forward output voltage
Red	Solid	Full reverse output
	Blinking	Proportional to reverse output voltage
Orange	Solid	No CAN devices are connected
	Blinking	Disabled state, PWM signal lost, FRC robot disabled, or signal in deadband range (+/- 4% output)
Off		No input power to Talon
Red/Green	Flashing	Ready for calibration. Several green flashes indicates successful calibration, and red several times indicates unsuccessful calibration.

40.4.12 Victor888 Motor Controller



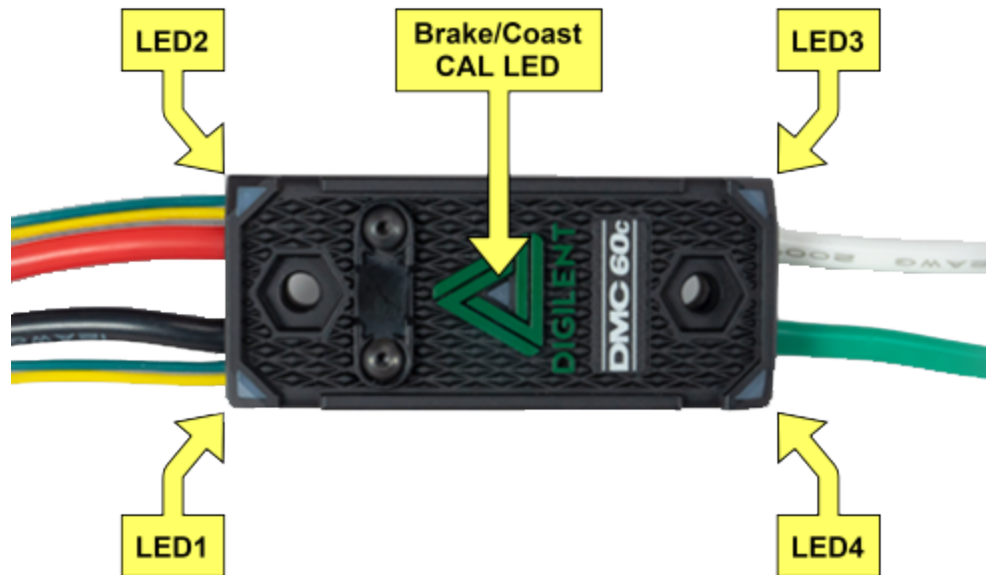
Green	Solid	Full forward output
	Blinking	Successful calibration
Red	Solid	Full reverse output
	Blinking	Unsuccessful calibration
Orange	Solid	Neutral/brake
Red/Green	Blinking	Calibration mode

40.4.13 Jaguar Motor Controller



LED State	Module Status
Normal Operating Conditions	
Solid Yellow	Neutral (speed set to 0)
Fast Flashing Green	Forward
Fast Flashing Red	Reverse
Solid Green	Full-speed forward
Solid Red	Full-speed reverse
Fault Conditions	
Slow Flashing Yellow	Loss of servo or Network link
Fast Flashing Yellow	Invalid CAN ID
Slow Flashing Red	Voltage, Temperature, or Limit Switch fault condition
Slow Flashing Red and Yellow	Current fault condition
Calibration or CAN Conditions	
Flashing Red and Green	Calibration mode active
Flashing Red and Yellow	Calibration mode failure
Flashing Green and Yellow	Calibration mode success
Slow Flashing Green	CAN ID assignment mode
Fast Flashing Yellow	Current CAN ID (count flashes to determine ID)
Flashing Yellow	CAN ID invalid (that is, Set to 0) awaiting valid ID assignment

40.4.14 Digilent DMC-60



The DMC60C contains four RGB (Red, Green, and Blue) LEDs and one Brake/Coast CAL LED. The four RGB LEDs are located in the corners and are used to indicate status during normal operation, as well as when a fault occurs. The Brake/Coast CAL LED is located in the center of the triangle, which is located at the center of the housing, and is used to indicate the current Brake/Coast setting. When the center LED is off, the device is operating in coast mode. When the center LED is illuminated, the device is operating in brake mode. The Brake/Coast mode can be toggled by pressing down on the center of the triangle, and then releasing the button.

At power-on, the RGB LEDs illuminate Blue, continually getting brighter. This lasts for approximately five seconds. During this time, the motor controller will not respond to an input signal, nor will the output drivers be enabled. After the initial power-on has completed, the device begins normal operation and what gets displayed on the RGB LEDs is a function of the input signal being applied, as well as the current fault state. Assuming that no faults have occurred, the RGB LEDs function as follows:

PWM Signal Applied	LED State
No Input Signal or Invalid Input Pulse Width	Alternate between top (LED1 and LED2) and bottom (LED3 and LED4) LEDs being illuminated Red and Off.
Neutral Input Pulse Width	All 4 LEDs illuminated Orange.
Positive Input Pulse Width	LEDs blink Green in a clockwise circular pattern (LED1 → LED2 → LED3 → LED4 → LED1). The LED update rate is proportional to the duty cycle of the output and increases with increased duty cycle. At 100% duty cycle, all 4 LEDs are illuminated Green.
Negative Input Pulse Width	LEDs blink Red in a counter-clockwise circular pattern (LED1 → LED4 → LED3 → LED2 → LED1). The LED update rate is proportional to the duty cycle of the output and increases with increased duty cycle. At 100% duty cycle, all 4 LEDs are illuminated Red.

CAN Bus Control State	LED State
No Input Signal or CAN bus error detected	Alternate between top (LED1 and LED2) and bottom (LED3 and LED4) LEDs being illuminated Red and Off.
No CAN Control Frame received within the last 100ms or the last control frame specified modeN-oDrive (Output Disabled)	Alternate between top (LED1 and LED2) and bottom (LED3 and LED4) LEDs being illuminated Orange and Off.
Valid CAN Control Frame received within the last 100ms. The specified control mode resulted in a Neutral Duty Cycle being applied to Motor Output	All 4 LEDs illuminated solid Orange.
Valid CAN Control Frame received within the last 100ms. The specified control mode resulted in a Positive Duty Cycle being Motor Output	LEDs blink Green in a clockwise circular pattern (LED1 → LED2 → LED3 → LED4 → LED1). The LED update rate is proportional to the duty cycle of the output and increases with increased duty cycle. At 100% duty cycle, all 4 LEDs are illuminated Green.
Valid CAN Control Frame received within the last 100ms. The specified control mode resulted in a Negative Duty Cycle being Motor Output	LEDs blink Red in a counter-clockwise circular pattern (LED1 → LED4 → LED3 → LED2 → LED1). The LED update rate is proportional to the duty cycle of the output and increases with increased duty cycle. At 100% duty cycle, all 4 LEDs are illuminated Red.

Fault Color Indicators
















When a fault condition is detected, the output duty cycle is reduced to 0% and a fault is signaled. The output then remains disabled for 3 seconds. During this time the onboard LEDs (LED1-4) are used to indicate the fault condition. The fault condition is indicated by toggling between the top (LED1 and LED2) and bottom (LED3 and LED4) LEDs being illuminated Red and off. The color of the bottom LEDs depends on which faults are presently active. The table below describes how the color of the bottom LEDs maps to the presently active faults.

Color	Over Temperature	Under Voltage
Green	On	Off
Blue	Off	On
Cyan / Aqua	On	On

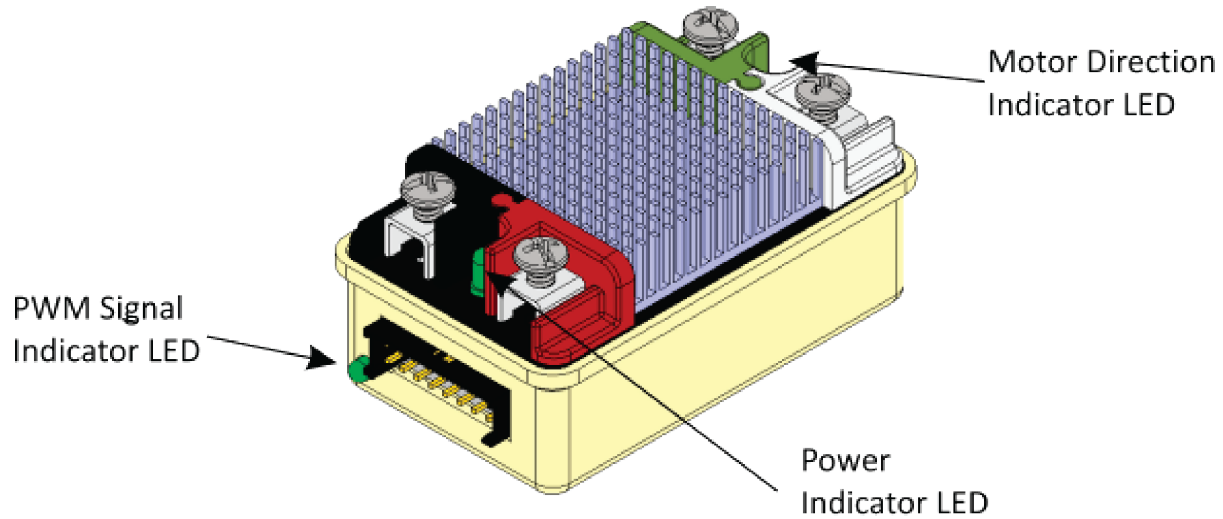
Break/Coast Mode

When the center LED is off the device is operating in coast mode. When the center LED is illuminated the device is operating in brake mode. The Brake/Coast mode can be toggled by pressing down on the center of the triangle and then releasing the button.

40.4.15 Venom Motor Controller

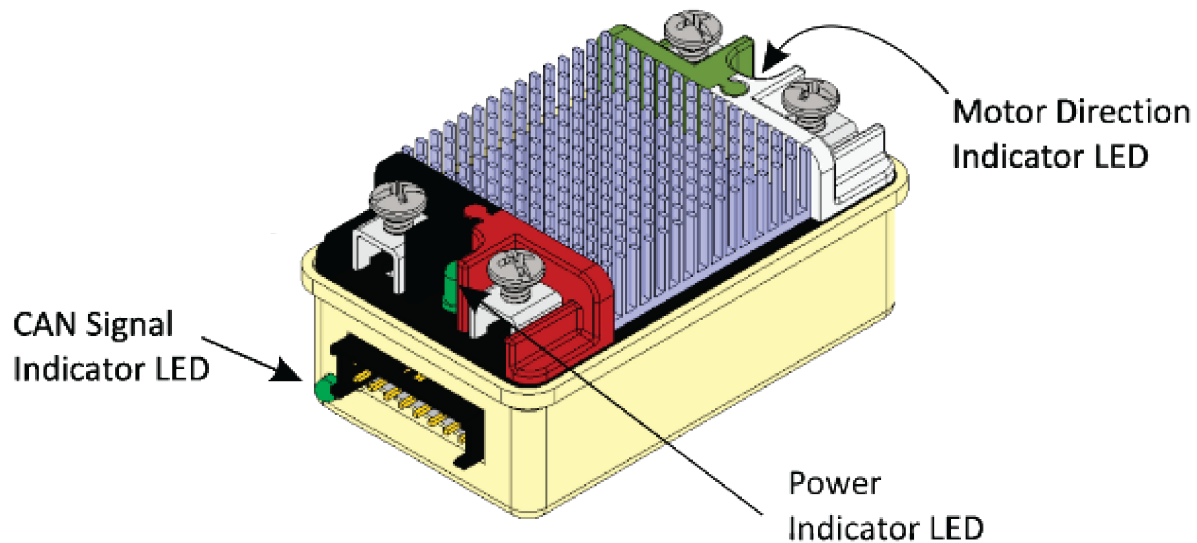
LED Pattern	Description
	Venom is initializing. This state should last less than 40ms after power up.
	The 'Identify Device' feature is active. This pattern is used to locate a particular Playing With Fusion device when multiple are installed on a robot. See the Motor Configuration section for more information
	Venom is initialized and in PWM mode. Waiting for a valid 1.0 to 2.0 ms PWM pulse.
	Venom successfully entered a valid CAN or PWM control mode. No Faults are active and motion may be commanded
	Venom is initialized and detected a valid CAN bus
	CAN communication fault. Check harness connections and bus termination
	Missing heartbeat in CAN control mode. Ensure device ID matches device ID used by CANVenom class. See the Motor Configuration section for more information and instructions to change/verify the device ID
	Lead motor heartbeat is missing while in Follow The Leader mode.
	The lead motor ID is same as the motor ID. One Venom cannot follow itself. Ensure the leader and follower have different IDs
	An invalid control mode was specified by the roboRIO. This should not occur when using PlayingWithFusionDriver. Contact PWF Technical support.
	Another Venom with the same device ID was detected on the CAN bus. All Venom device IDs must be unique
	The forward limit switch is enabled and is active
	The reverse limit switch is enabled and is active
	Motor temperature is too high
	Average motor current is too high

40.4.16 Mindsensors SD540B (PWM)



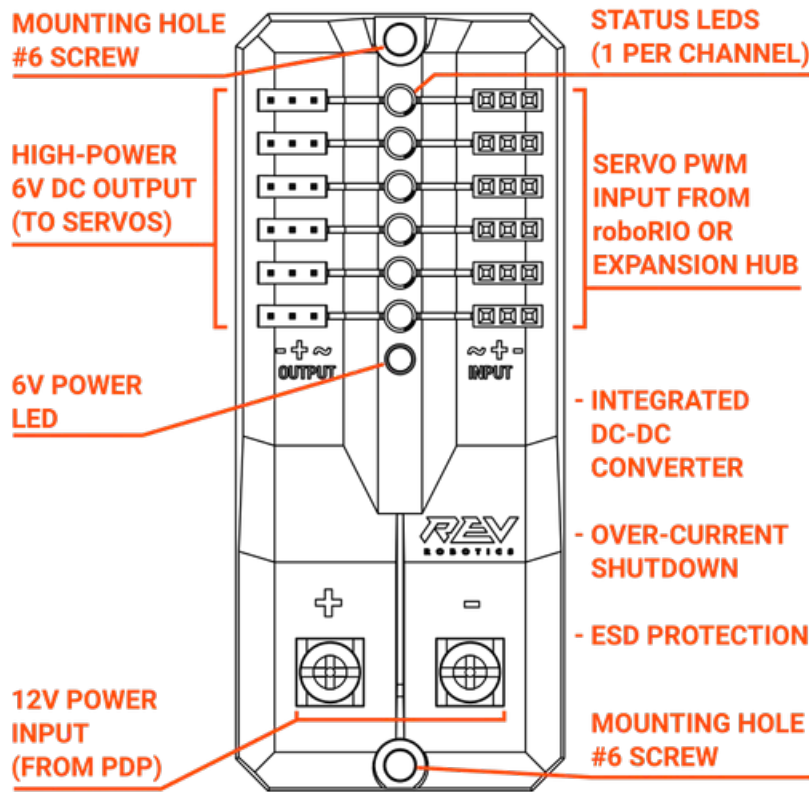
Power LED	Off	Power is not supplied
	Red	Power is supplied
Motor LED	Red	Forward direction
	Green	Reverse direction
PWM Signal LED	Red	No valid PWM signal is detected
	Green	Valid PWM signal is detected

40.4.17 Mindsensors SD540C (CAN Bus)



Power LED	Off	Power is not supplied
	Red	Power is supplied
Motor LED	Red	Forward direction
	Green	Reverse direction
CAN Signal LED	Blinking quickly	No CAN devices are connected
	Off	Connected to the roboRIO and the driver station is open

40.4.18 REV Robotics Servo Power Module



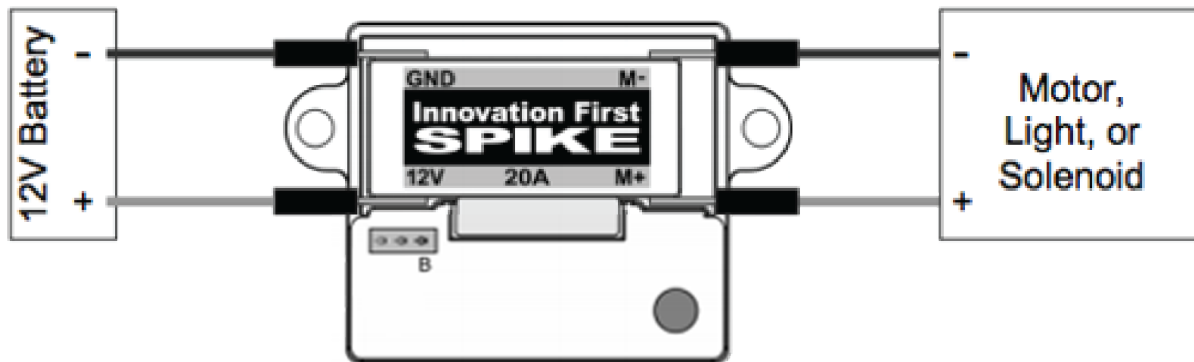
Status LEDs

Each channel has a corresponding status LED that will indicate the sensed state of the connected PWM signal. The table below describes each state's corresponding LED pattern.

State	Pattern
No Signal	Blinking Amber
Left/Reverse Signal	Solid Red
Center/Neutral Signal	Solid Amber
Right/Forward Signal	Solid Green

- 6V Power LED off, dim or flickering with power applied = Over-current shutdown

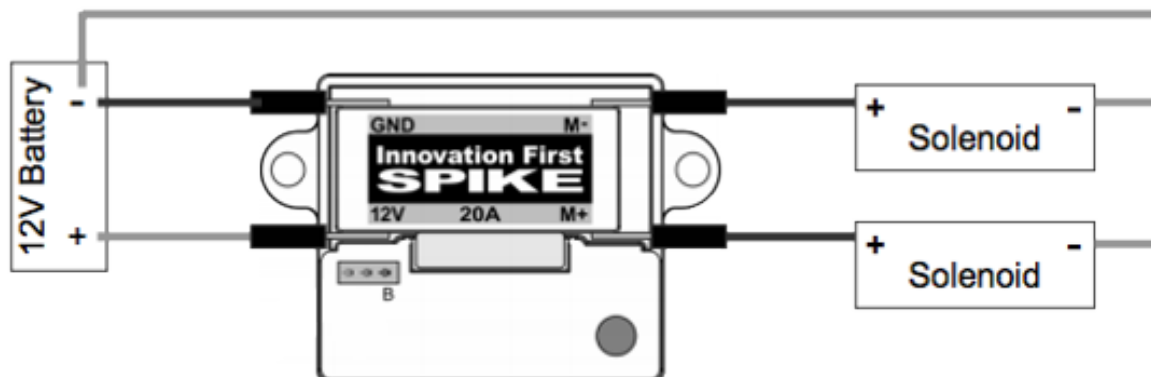
40.4.19 Spike relay configured as a motor, light, or solenoid switch



Inputs		Outputs		Indicator	Motor Function
Forward (White)	Reverse (Red)	M+	M-		
Off	Off	GND	GND	Orange	Off/Brake Condition (default)
On	Off	+12v	GND	Green	Motor rotates in one direction
Off	On	GND	+12v	Red	Motor rotates in opposite direction
On	On	+12v	+12v	Off	Off/Brake Condition

Note: 'Brake Condition' refers to the dynamic stopping of the motor due to the shorting of the motor inputs. This condition is not optional when going to an off state.

40.4.20 Spike relay configured as for one or two solenoids

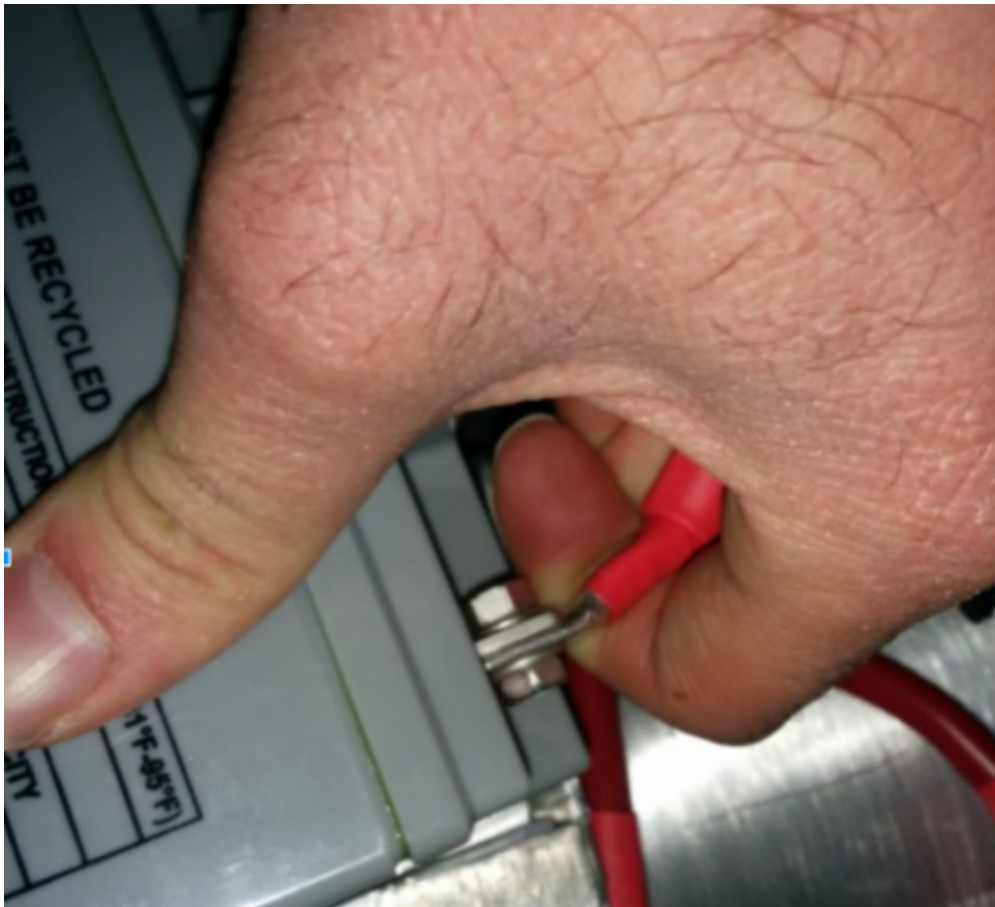


Inputs		Outputs		Indicator	Motor Function
Forward (White)	Reverse (Red)	M+	M-		
Off	Off	GND	GND	Orange	Both Solenoids Off (default)
On	Off	+12v	GND	Green	Solenoid connected to M+ is ON
Off	On	GND	+12v	Red	Solenoid connected to M- is ON
On	On	+12v	+12v	Off	Both Solenoids ON

40.5 Robot Preemptive Troubleshooting

Note: In *FIRST*® Robotics Competition, robots take a lot of stress while driving around the field. It is important to make sure that connections are tight, parts are bolted securely in place and that everything is mounted so that a robot bouncing around the field does not break.

40.5.1 Check Battery Connections

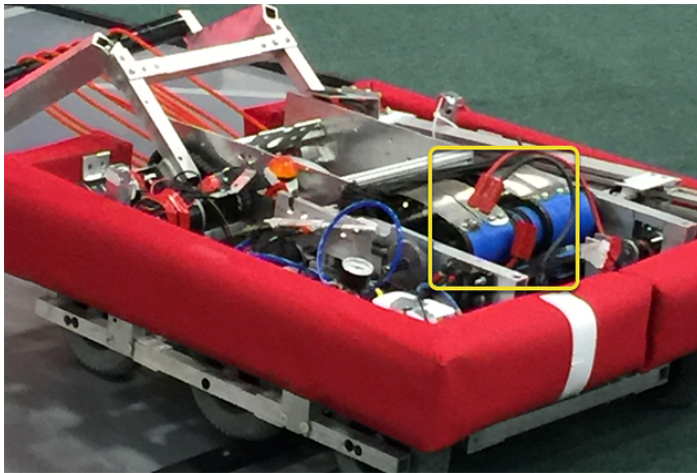


The tape that should be covering the battery connection in these examples has been removed to illustrate what is going on. On your robots, the connections should be covered.

Wiggle battery harness connector. Often these are loose because the screws loosen, or sometimes the crimp is not completely closed. You will only catch the really bad ones though because often the electrical tape stiffens the connection to a point where it feels stiff. Using a voltmeter or Battery Beak will help with this.

Apply considerable force onto the battery cable at 90 degrees to try to move the direction of the cable leaving the battery, if successful the connection was not tight enough to begin with and it should be redone. This [article](#) has more detailed battery information.

40.5.2 Securing the Battery to the Robot



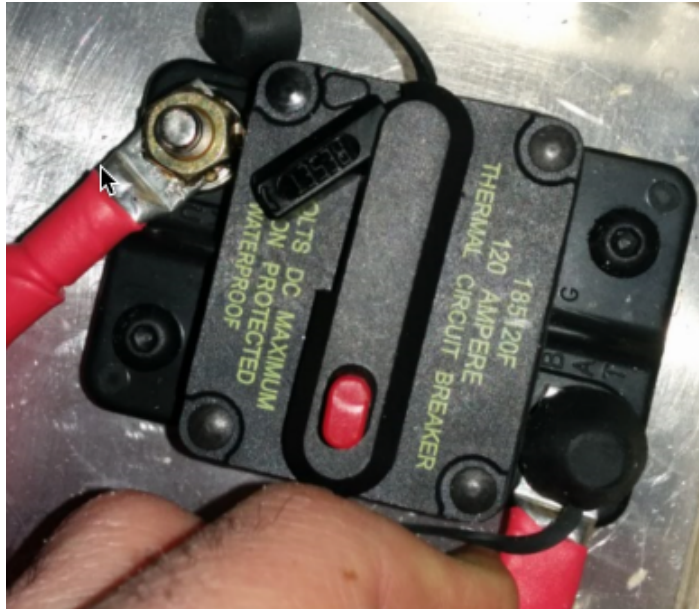
In almost every event we see at least one robot where a not properly secured battery connector (the large Anderson) comes apart and disconnects power from the robot. This has happened in championship matches on the Einstein and everywhere else. It's an easy to ensure that this doesn't happen to you by securing the two connectors by wrapping a tie wrap around the connection. 10 or 12 tie wraps for the piece of mind during an event is not a high price to pay to guarantee that you will not have the problem of this robot from an actual event after a bumpy ride over a defense. Also, secure your battery to the chassis with hook and loop tape or another method, especially in games with rough defense, obstacles or climbing.

40.5.3 Securing the Battery Connector & Main Power Leads

A loose robot-side battery connector (the large Anderson SB) can allow the main power leads to be tugged when the battery is replaced. If the main power leads are loose, that "tug" can get all the way back to the crimp lugs attached to the 120 Amp Circuit Breaker or Power Distribution Panel (PDP), bend the lug, and over time cause the lug end to break from fatigue. Putting a couple tie wraps attaching the main power leads to the chassis and bolting down the robot-side battery connector can prevent this, as well as make it easier to connect the battery.

40.5.4 Main Breaker (120 Amp Circuit Breaker)

Note: Ensure nuts are tightened firmly and the breaker is attached to a rigid element.



Apply a strong twisting force to try to rotate the crimped lug. If the lug rotates then the nut is not tight enough. After tightening the nut, retest by once again trying to rotate the lug.

The original nut has a star locking feature, which can wear out over time: these may require checking every few matches, especially if your robot-side battery connector is not attached to the chassis.

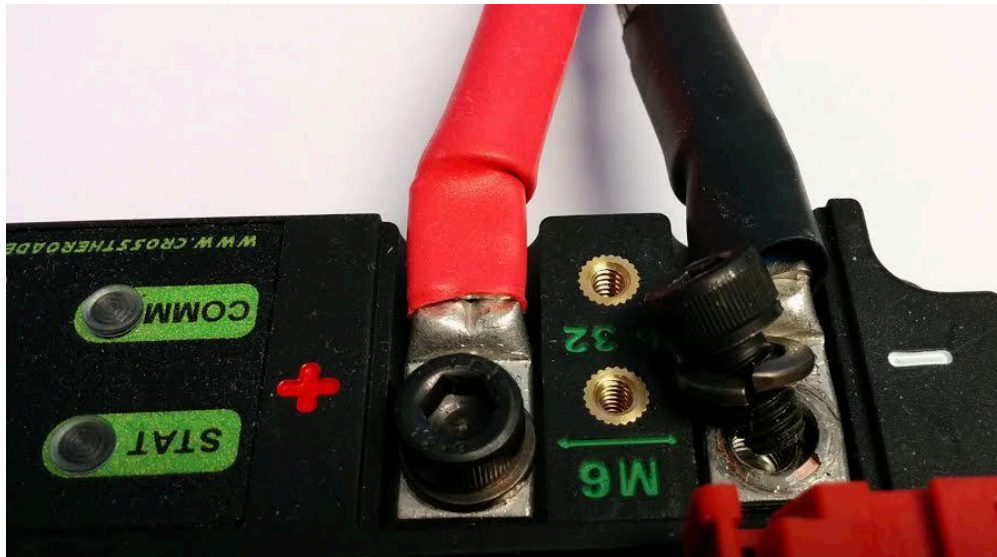
The nut is normally a relatively uncommon 1/4-28 thread: ensure this is correct if the nut is replaced.

Because the metal stud is just molded into the case, every once in awhile you may break off

the stud. Don't stress, just replace the assembly.

When subjected to multiple competition seasons, the Main Breaker is susceptible to fatigue damage from vibration and use, and can start opening under impact. Each time the thermal fuse function is triggered, it can become progressively easier to trip. Many veteran teams start each season with a fresh main breaker, and carry spares.

40.5.5 Power Distribution Panel (PDP)



Make sure that split washers were placed under the PDP screws, but it is not easy to visually confirm, and sometimes you can't. You can check by removing the case. Also if you squeeze the red and black wires together, sometimes you can catch the really loose connections.

40.5.6 Tug Testing





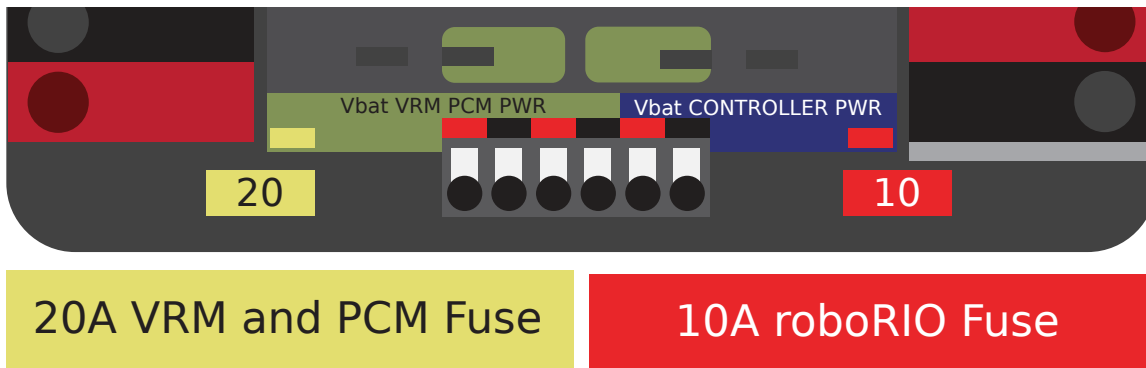
The Weidmuller contacts for power, compressor output, roboRIO power connector, and radio power are important to verify by tugging on the connections as shown. Make sure that none of the connections pull out.

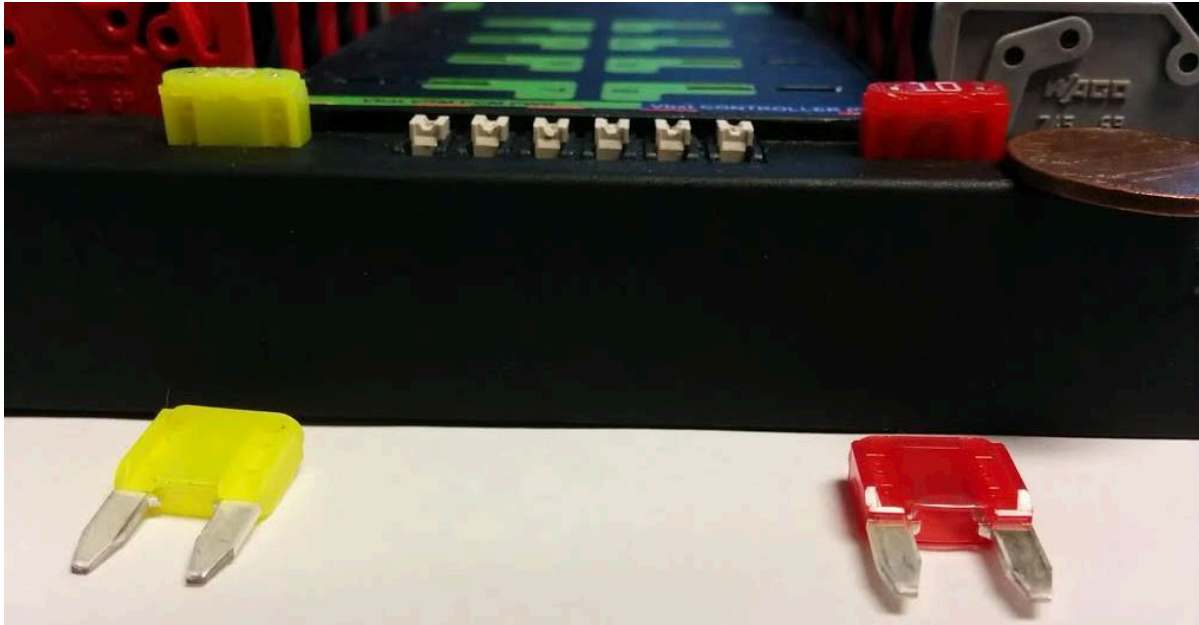
Look for possible or impending shorts with Weidmuller connections that are close to each other, and have too-long wire-lead lengths (wires that are stripped extra long).

Spade connectors can also fail due to improper crimps, so tug-test those as well.

40.5.7 Blade Fuses

Be sure to place the 20A fuse (yellow) on the left and the 10A fuse (red) on the right.





Warning: Take care to ensure fuses are fully seated into the fuse holders. The fuses should descend at least as far as the figure below (different brand fuses have different lead lengths). It should be nearly impossible to remove the fuse with bare hands (without the use of pliers). If this is not properly done, the robot/radio may exhibit intermittent connectivity issues.

If you can remove the blade fuses by hand then they are not in completely. Make sure that they are completely seated in the PDP so that they don't pop out during robot operation.

40.5.8 roboRIO swarf

Swarf is fine chips or filings of stone, metal, or other material produced by a machining operation. Often modifications must be made to a robot while the control system parts are in place. The circuit board for the roboRIO is conformally coated, but that doesn't absolutely guarantee that metal chips won't short out traces or components inside the case. In this case, you must exercise care in making sure that none of the chips end up in the roboRIO or any of the other components. In particular, the exposed 3 pin headers are a place where chips can enter the case. A quick sweep through each of the four sides with a flashlight is usually sufficient to find the really bad areas of infiltration.

40.5.9 Radio Barrel Jack

Make sure the correct barrel jack is used, not one that is too small and falls out for no reason. This isn't common, but ask an FTA and every once in awhile a team will use some random barrel jack that is not sized correctly, and it falls out in a match on first contact.

40.5.10 Ethernet Cable

If the RIO to radio ethernet cable is missing the clip that locks the connector in, get another cable. This is a common problem that will happen several times in every competition. Make sure that your cables are secure. The clip often breaks off, especially when pulling it through a tight path, it snags on something then breaks.

40.5.11 Loose Cables

Cables must be tightened down, particularly the radio power and ethernet cable. The radio power cables don't have a lot of friction force and will fall out (even if it is the correct barrel) if the weight of the cable-slack is allowed to swing freely.

Ethernet cable is also pretty heavy, if it's allowed to swing freely, the plastic clip may not be enough to hold the ethernet pin connectors in circuit.

40.5.12 Reproducing Problems in the Pit

Beyond the normal shaking of cables whilst the robot is powered and tethered, it is suggested that one side of the robot be picked up and dropped. Driving on the field, especially against defenders, will often be very violent, and this helps make sure nothing falls out. It is better for the robot to fail in the pits rather than in the middle of a match.

When doing this test it's important to be ethernet tethered and not USB tethered, otherwise you are not testing all of the critical paths.

40.5.13 Check Firmware and Versions

Robot inspectors do this, but you should do it as well, it helps robot inspectors out and they appreciate it. And it guarantees that you are running with the most recent, bug fixed code. You wouldn't want to lose a match because of an out of date piece of control system software on your robot.

40.5.14 Driver Station Checks

We often see problems with the Drivers Station. You should:

- ALWAYS bring the laptop power cable to the field, it doesn't matter how good the battery is, you are allowed to plug in at the field.
- Check the power and sleep settings, turn off sleep and hibernate, screen savers, etc.
- Turn off power management for USB devices (dev manager)
- Turn off power management for ethernet ports (dev manager)

- Turn off windows defender
- Turn off firewall
- Close all apps except for DS/Dashboard when out on the field.
- Verify that there is nothing unnecessary running in the application tray in the start menu (bottom right side)

40.5.15 Handy Tools



There never seems to be enough light inside robots, at least not enough to scrutinize the critical connection points, so consider using a handheld LED flashlight to inspect the connections on your robot. They're available from home depot or any hardware/automotive store.

A WAGO tool is nice tool for redoing Weidmuller connections with stranded wires. Often I'll do one to show the team, and then have them do the rest using the WAGO tool to press down the white-plunger while they insert the stranded wire. The angle of the WAGO tool makes this particularly helpful.

40.6 Robot Battery Basics

The power supply for an FRC® robot is a single 12V 18Ah SLA (Sealed Lead Acid) non-spillable battery, capable of briefly supplying over 180A and arcing over 500A when fully charged. The Robot Battery assembly includes the COTS battery, lead cables with contacts, and Anderson SB connector. Teams are encouraged to have multiple Robot Batteries.

40.6.1 COTS Battery

The Robot Rules in the Game Manual specify a Commercial Off The Shelf [COTS] non-spillable sealed lead acid battery meeting specific criteria, and gives examples of legal part numbers from a variety of vendors.

40.6.2 Battery Safety & Handling

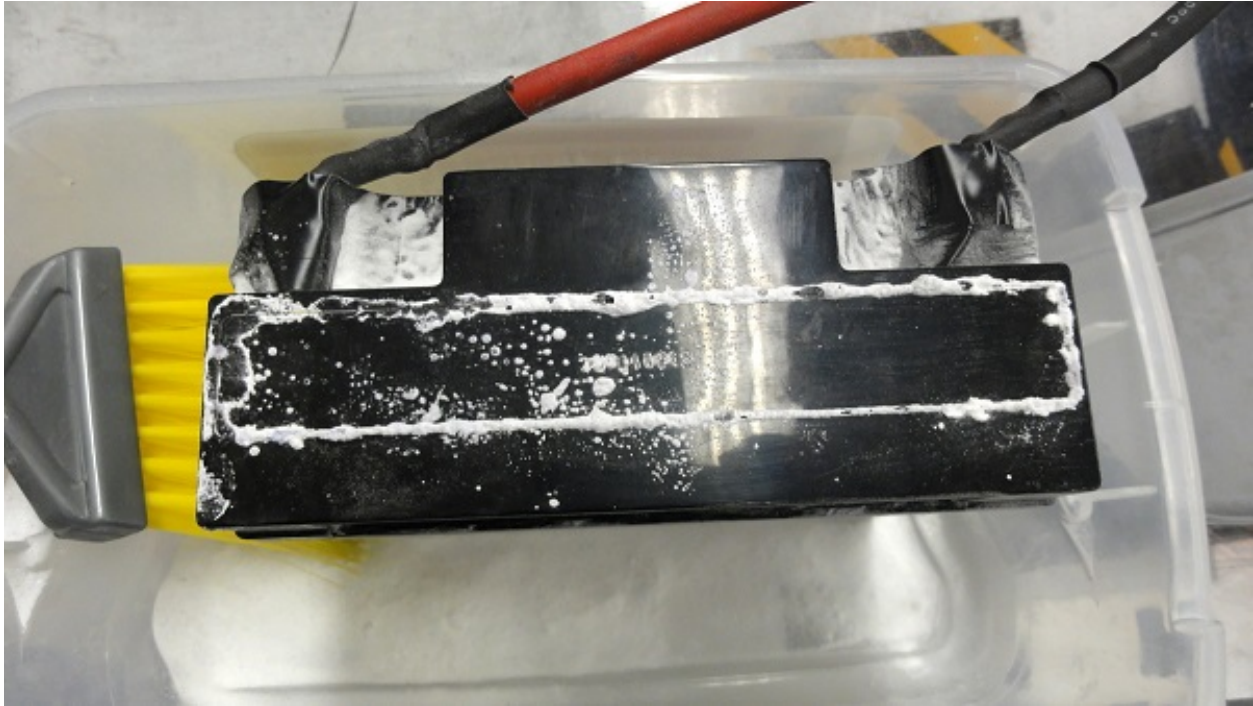
A healthy battery is **always** “On” and the terminals are **always** energized. If the polarities short together - for example, a wrench or aerosol can falls and bridges the gap between two bare terminals - all the stored energy will be released in a dangerous arc. This risk drives a wide range of best practices, such as covering terminals in storage, only uncovering and working on one terminal or polarity at a time, keeping SB contacts fully inserted in connectors, etc.

Do *NOT* carry a battery assembly by the cables, and always avoid pulling by them. Pulling on batteries by the cables will begin to damage the lugs, tabs, and the internal connection of the tab. Over time, fatigue damage can add up until the the entire tab tears out of the housing! Even if it isn't clearly broken, internal fatigue damage can increase the battery internal resistance, prematurely wearing out the battery. The battery will not be able to provide the same amount of current with increased internal resistance or if the *connectors are loose*.



Dropping the batteries can bend the internal plates and cause performance issues, create bulges, or even crack the battery case open. While most FRC batteries use Absorbent Glass Mat [AGM] or Gel technology for safety and performance, when a cell is punctured it may still leak a small amount of battery acid. This is one of the reasons FIRST recommends teams have a battery spill kit available.

Finally, certain older battery chargers without “maintenance mode” features can *overcharge* the battery, resulting in boiling off some of the battery acid.



Damaged batteries should be safely disposed of as soon as possible. All retailers that sell large SLA batteries, like car batteries, should be able to dispose of it for you. They may charge a small fee, or provide a small “core charge refund”, depending on your state law.

Danger: DO NOT attempt to “repair” damaged or non-functional batteries.

40.6.3 Battery Construction & Tools

Battery Leads

Battery leads must be copper, minimum size (cross section) 6 AWG (16mm², 7 SWG) and maximum length 12”, color coded for polarity, with an Anderson SB connector. Standard 6AWG copper leads with Pink/Red SB50 battery leads often come in the Kit of Parts and are sold by FRC vendors.

Lead Cables

Tinned, annealed, or coated copper is allowed. Do not use CCA (copper clad aluminum), aluminum, or other non-copper base metal. The conductor metal is normally printed on the outside of the insulation with the other cable ratings.

Wire size 6AWG is sufficient for almost all robots and fits standard SB50 contacts. A small number of teams adopt larger wire sizes for marginal performance benefits.

Higher strand count wire (sometimes sold as “Flex” or “welding wire”) has a smaller bend radius, which makes it easier to route, and a higher fatigue limit. There is no strand count requirement, but 84/25 (84 strand “flex” hookup wire) and 259/30 (259 strand “welding wire”) will both be *much* easier to work with than 19/0.0372 (19 strand hookup wire).

The insulation must be color-coded per the Game Manual: as of 2021, the +12Vdc wire must be red, white, brown, yellow, or black w/stripe and the ground wire (return wire) must be black or blue. There is no explicit insulation temperature rating requirement, but any blackened or damaged insulation means the wire needs to be replaced: off hand, 105C is plenty and lower will work for almost all robots. There is no insulation voltage rating requirement, lower is better for thinner insulation.

SB Connector

The Anderson SB Connector may be the standard Pink/Red SB50, or another Anderson SB connector. Teams are *STRONGLY* recommended to use the Pink/Red SB50 for interoperability: the other colors and sizes of housings will not intermate, and you will be unable to borrow batteries or chargers.

Follow manufacturer's instructions to crimp contacts and assemble the leads into Anderson SB connectors. A small flathead screwdriver can help to insert the contacts (push on the contact, not on the wire insulation), or it can help to disengage the internal latch if the contact is in the wrong slot or upside down.

Battery Lugs

Compression lugs ("crimp lugs") for #10 bolt (or M5) battery tabs (~0.2" or ~5mm hole diameter) are available online and through electrical supply houses, sold by the accepted wire sizes in AWG (or mm²) and post diameter ("bolt size", "hole diameter"). Higher end vendors will also distinguish between Standard (~19) and Flex (>80) strand counts in their lug catalogs. Some vendors also offer right angle lugs, in addition to more common straight styles. Follow manufacturer's instructions to crimp the lugs.

Screw terminal lugs are legal, but not recommended. If using screw terminal lugs, use the correct tip size screwdriver to tighten the terminal. Check the terminal tightness frequently because they may loosen over time.

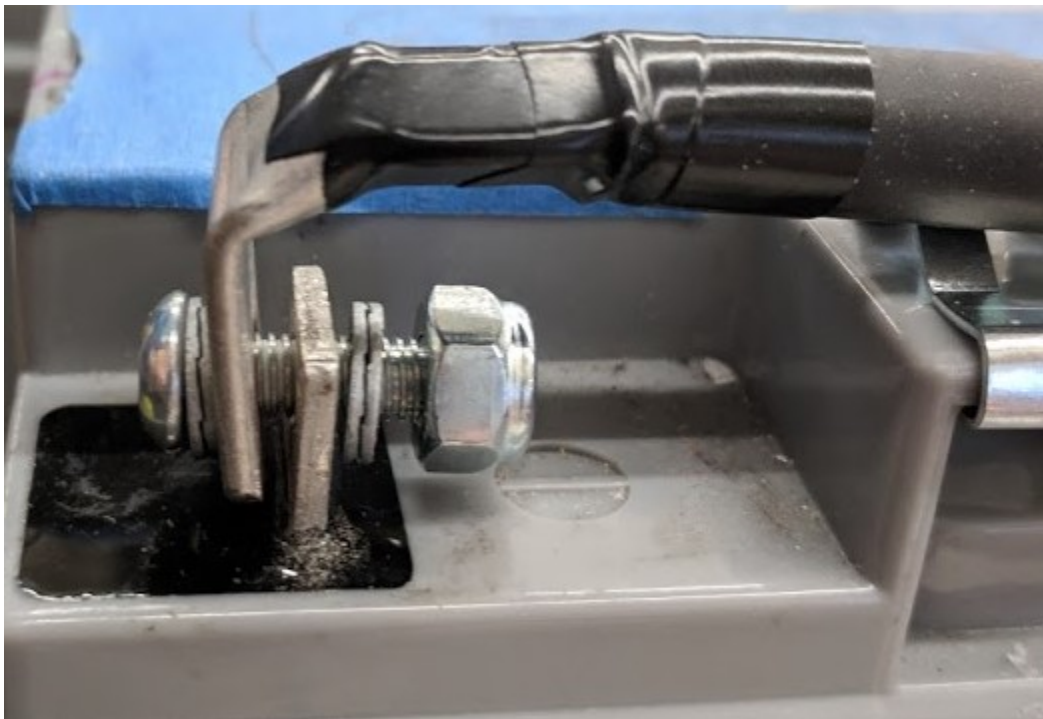
Battery Lead Lug To Post Connection

A #10 or M5 nut & bolt connect the battery lead lug to the battery tab.

Warning: The lug and tab must directly contact, copper to copper: do not put a washer of any kind separating them.



Some batteries come with tab bolts in the package: they may be used, or replaced with stronger alloy steel bolts. It is a good idea to add a functional lock washer, such as a #10 star washer or a nordlock washer system, in addition to a nylon locking (“nylock”) nut. Only use one style of lock washer in each connection. Even if the manufacturer provides split ring lock washers in the package, you are not required to use them.



These connections must be very tight for reliability. Any movement of the lug while in operation may interrupt robot power, resulting in robot reboots and field disconnections lasting

30 seconds or more.

This connection must also be completely covered for electrical safety; electrical tape will work, but heatshrink that fits over the entire connection is recommended. High shrink ratios (minimum 3:1, recommend 4:1) will make it easier to apply the heatshrink. Adhesive lined heat shrink is allowed. Be sure *all* the copper is covered! Heat shrink must be “touched up” with electrical tape if some copper shows.



Battery Chargers

There are many good COTS “smart” battery chargers designed for 12V SLA batteries, rated for 6A or less per battery, with ‘maintenance mode’ features. Chargers rated over 6A are not allowed in FRC pits.

Chargers used at competition are required to use Anderson SB connectors. Attaching a COTS SB connector battery lead to the charger leads using appropriately sized wire nuts or screw terminals is fast and simple (be sure to cover any exposed copper with heat shrink or electrical tape). SB Connector Contacts are also available for smaller wire sizes, if the team has crimping capability.

Warning: After attaching the SB, double check the charger polarities with a multimeter before plugging in the first battery.

Some FRC vendors sell chargers with red SB50 connectors pre-attached.

Battery Evaluation Tools

Battery Charger

If your battery charger has Maintenance Mode indicator, such as a GREEN LED, you can use that indicator to tell you whether you are READY. Some chargers will cycle between “CHARGING” and “READY” periodically. This is a “maintenance” behavior, sometimes associated with the battery cooling off and being able to accept more charge.

Driver Station Display and Log

When the robot is plugged in and connected to the driver station laptop, the battery voltage is displayed on the NI Driver Station software.

After you finish a driving session, you can [review the battery voltage in the Log Viewer](#).

Hand-held Voltmeter or Multimeter

A voltage reading from probes on the SB connector of a disconnected battery will give you a snapshot of what the Voc (Voltage open circuit, or “float voltage”) is in the “Unloaded” state. In general the Voc is not a recommended method for understanding battery health: the open circuit voltage is not as useful as the combination of internal resistance and voltages at specific loads provided by a Load Tester (or Battery Analyzer).

Load Tester

A battery load tester can be used as a quick way to determine the detailed readiness of a battery. It may provide information like: open-load voltage, voltage under load, internal resistance, and state of charge. These metrics can be used to quickly confirm that a battery is ready for a match and even help to identify some long term problems with the battery.

```
Status: Good
Charge: 130%
V0: 13.456 @ 0 Amps
V1: 13.443 @ 1 Amps
V2: 13.153 @ 18 Amps
Rint: 0.017 Ohms
```

Ideal internal resistance should be less than 0.015 Ohms. The manufacturer specification for most batteries is 0.011 Ohms. If a battery gets higher than 0.020 Ohms it is a good idea to consider not using that battery for competition matches.

If a battery shows significantly lower voltages at the higher test current loads, it may not be done charging, or it may need to be retired.

40.6.4 Understanding Battery Voltages

A “12V battery” is anything but 12.0V.

Fully charged, a battery can be anywhere from 12.7 to 13.5 volts open circuit (Voc). Open circuit voltage is measured with *nothing* connected.

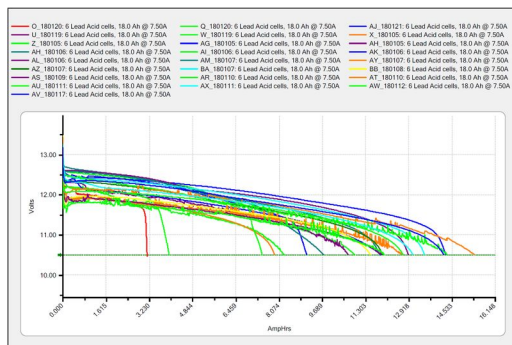
Once a load (like a robot) is connected, and any amount of current is flowing, the battery voltage will drop. So if you check a battery with a Voltmeter, and it reads 13.2, and then connect it to your robot and power on, it will read lower, maybe 12.9 on the Driver Station display. Those numbers will vary with every battery and specific robot, see Characterization below. Once your robot starts running, it will pull more current, and the voltage will drop further.

Batteries reading 12.5V on an idle robot should be swapped and charged before a match. Always swap the batteries before the robot starts reaching brownout safety thresholds (dwelling at low voltages on the Driver Station display), as frequently entering low voltage ranges risks permanent battery damage; this behavior can happen at a variety of Voc states depending on battery health, battery manufacturer, and robot design. The battery State of Charge should be kept over 50% for battery longevity.

Battery voltage and current also depends on temperature: cool batteries are happy batteries.

Battery Characterization

A battery analyzer can be used to give a detailed inspection and comparison of battery performance.



It will provide graphs of battery performance over time. This test takes significant time (roughly two hours) so it is less suited to testing during competition. It is recommended to run this test on each battery every year to monitor and track its performance. This will determine how it should be used: matches, practice, testing, or disposed of.

At the standard 7.5 amps test load, competition batteries should have at least a 11.5 amp hour rating. Anything less than that should only be used for practice or other less demanding use cases.

Battery Longevity

A battery is rated for about 1200 normal charge/recharge cycles. The high currents required for an FRC match reduce that lifespan to about 400 cycles. These cycles are intended to be relatively low discharge, from around 13.5 down to 12 or 12.5 volts. Deep cycling the battery (running it all the way down) will damage it.

Batteries last the longest if they are kept fully charged when not in use, either by charging regularly or by use of a maintenance charger. Batteries drop roughly 0.1V every month of non-use.

Batteries need to be kept away from both extreme heat and cold. This generally means storing the batteries in a climate controlled area: a classroom closet is usually fine, a parking lot shipping container is more risky.

40.6.5 Battery Best Practices

- Only use a charged battery for competition matches. If you are in a situation where you have run out of charged batteries, please ask a veteran team for help! Nobody wants to see a robot dead on the field (*brownout*) due to a bad or uncharged battery.
- Teams are strongly recommended to use properly rated tools and stringent quality control practices for crimping processes (ask local veteran teams or a commercial electrician for help), or use vendor-made Battery Leads.
- Wait for batteries to cool after the match before recharging: the case should not be warm to the touch, fifteen minutes is usually plenty.
- Teams should consider purchasing several new batteries each year to help keep their batteries fresh. Elimination matches can require many batteries and there may not be enough time to recharge.



- A multi bank battery charger allows you to charge more than one battery at a time. Many

teams build a robot cart for their batteries and chargers allowing for easy transport and storage.

- It is a good idea to permanently identify each battery with at least: team number, year, and a unique identifier.
- Teams may also want to use something removeable (stickers, labeling machine etc.) to identify what that battery should be used for based on its performance data and when the last analyzer test was run.



- Using battery flags (a piece of plastic placed in the battery connector) is a common way to indicate that a battery has been charged. Battery flags can also be easily 3D printed.
- Handles for SB50 contacts can be purchased or 3D printed to help avoid pulling on the leads while connecting or disconnecting batteries. Do not use these handles to carry the weight of the battery.



- Some teams sew battery carrying straps from old seatbelts or other flat nylon that fit around the battery to help prevent carrying by leads.



- Cable tie edge clips can be used with 90 degree crimp lugs to strain relieve battery leads.



Note: The pages in this section of the documentation contain media which is only viewable from the web version of the documentation

41.1 Motors for Robotics Applications

One of the most important design decisions that teams have to deal with is selecting and designing the motor driven systems on their robot. So often the incorrect motor is chosen for a particular design yielding reduced performance and, sometimes even worse, motors failing from excessive current draw. In this series of videos, WPI Professor Ken Stafford walks through how motors work, how to design systems to operate at maximum performance, and a sample design for a robot system.

41.2 Sensing and Sensors

Without sensors and sensing robots are really radio controlled vehicles. Sensors allow the robots to understand the internal operation of the robots mechanical systems as well as the ability to interact with the environment around the robot. In these videos, WPI Professor Craig Putnam describes a number of classes of sensors, how they are used, and provides guidance on what sensors are best for your applications.

41.3 Pneumatics

Pneumatics is an often underused actuation device that can be used on robots. There are many advantages to pneumatics over using motors. In this video Professor Ken Stafford describes the characteristics of pneumatics, applications with robots, and calculating the right sized system for an application.

41.4 Power Transmission

Hand in hand with choosing the correct motors for an application is transmitting that motor power to the place it's needed. Using gears or chains and sprockets are two effective ways of matching the motor power to the application being driven. In this video, WPI Robotics Engineering PhD student Michael Delph talks about power transmission, including choosing correct gear or chain and sprocket ratios to get the the maximum performance from your robot design.

42.1 Sensor Overview - Hardware

Note: This section covers sensor hardware, not the use of sensors in code. For a software sensor guide, see [Sensor Overview - Software](#).

In order to be effective, it is often vital for robots to be able to gather information about their surroundings. Devices that provide feedback to the robot on the state of its environment are called “sensors.” There are a large variety of sensors available to FRC® teams, for measuring everything from on-field positioning to robot orientation to motor/mechanism positioning. Making use of sensors is an absolutely crucial skill for on-field success; while most FRC games do have tasks that can be accomplished by a “blind” robot, the best robots rely heavily on sensors to accomplish game tasks as quickly and reliably as possible.

Additionally, sensors can be extremely important for robot safety - many robot mechanisms are capable of breaking themselves if used incorrectly. Sensors provide a safeguard against this, allowing robots to, for example, disable a motor if a mechanism is against a hard-stop.

42.1.1 Types of Sensors

Sensors used in FRC can be generally categorized in two different ways: by function, and by communication protocol. The former categorization is relevant for robot design; the latter for wiring and programming.

Sensors by Function

Sensors can provide feedback on a variety of different aspects of the robot's state. Sensor functions common to FRC include:

- *Proximity switches*
 - Mechanical proximity switches ("limit switches")
 - Magnetic proximity switches
 - Inductive proximity switches
 - Photoelectric proximity switches
- Distance sensors
 - *Ultrasonic sensors*
 - *Triangulating rangefinders*
 - *LIDAR*
- Shaft rotation sensors
 - *Encoders*
 - *Potentiometers*
- *Accelerometers*
- *Gyroscopes*

Sensors by Communication Protocol

In order for a sensor to be useful, it must be able to "talk" to the roboRIO. There are several main methods by which sensors can communicate their readings to the roboRIO:

- *Analog input*
- *Digital input*
- *Serial bus*

In general, support for sensors that communicate via analog and digital inputs is straightforward, while communication over serial bus can be more complicated.

42.2 Analog Inputs - Hardware

Note: This section covers analog input hardware. For a software guide to analog inputs, see *Analog Inputs - Software*.

An **analog signal** is a signal whose value can lie anywhere in a continuous interval. This lies in stark contrast to a **digital signal**, which can take only one of several discrete values. The roboRIO's analog input ports allow the measurement of analog signals with values from 0V to 5V.

In practice, there is no way to measure a “true” analog signal with a digital device such as a computer (like the roboRIO). Accordingly, the analog inputs are actually measured as a 12-bit digital signal - however, this is quite a high resolution¹.

Analog inputs are typically (but not always!) used for sensors whose measurements vary continuously over a range, such as *ultrasonic rangefinders* and *potentiometers*, as they can communicate by outputting a voltage proportional to their measurements.

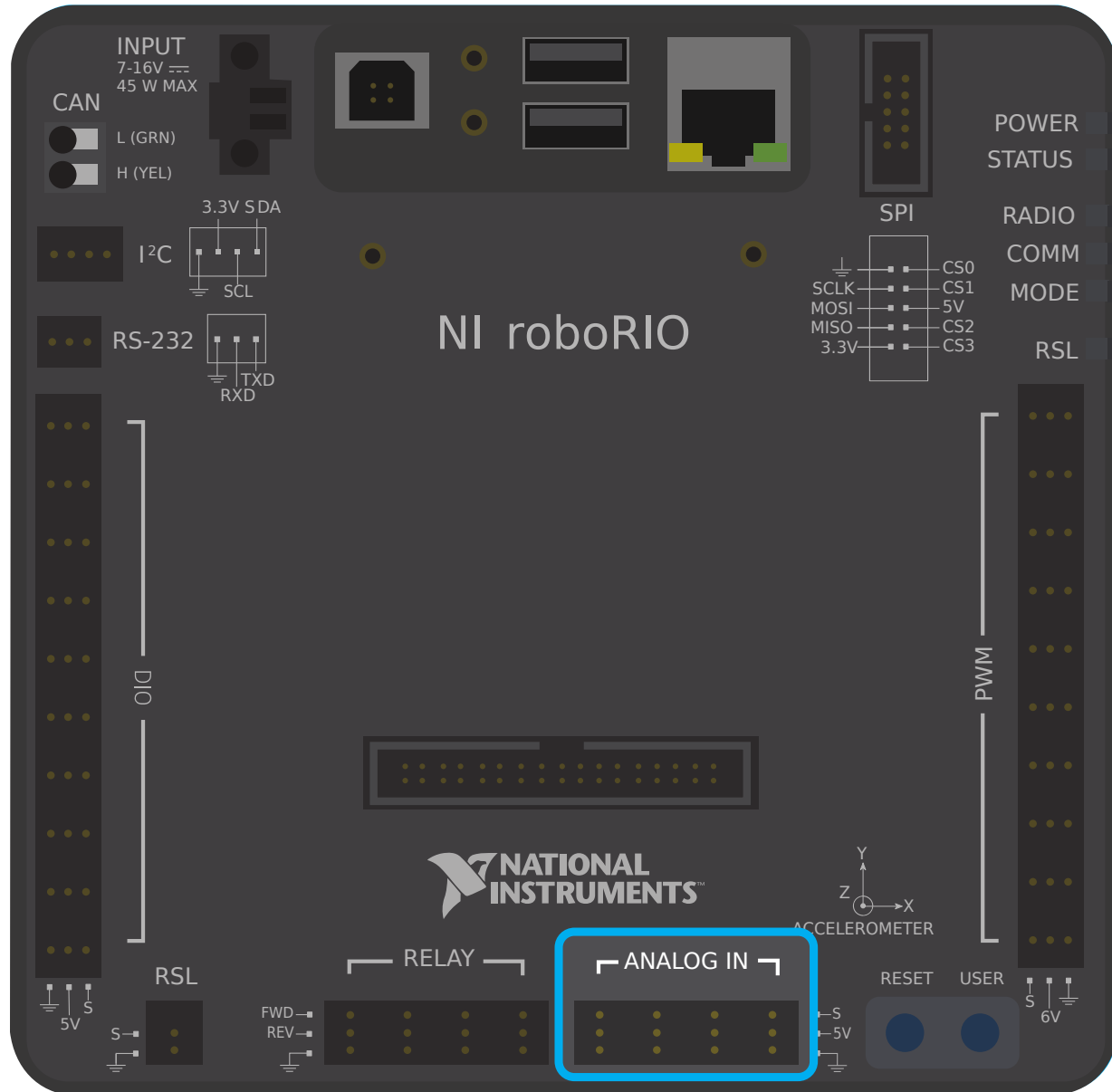
42.2.1 Connecting to roboRIO analog input ports

Note: An additional four analog inputs are available via the “MXP” expansion port. To use these, a breakout board of some sort that connects to the MXP is needed.

Warning: Always consult the technical specifications of the sensor you are using *before* wiring the sensor, to ensure that the correct wire is being connected to each pin. Failure to do so can result in damage to the sensor or the RIO.

Warning: **Never** directly connect the power pin to the ground pin on any port on the roboRIO! This will trigger protection features on the roboRIO and may result in unexpected behavior.

¹ A 12-bit resolution yields 2^{12} , or 4096 different values. For a 5V range, that’s an effective resolution of approximately 1.2 mV, or .0012V. The actual accuracy specification is plus-or-minus 50mV, so the discretization is not the limiting factor in the measurement accuracy.



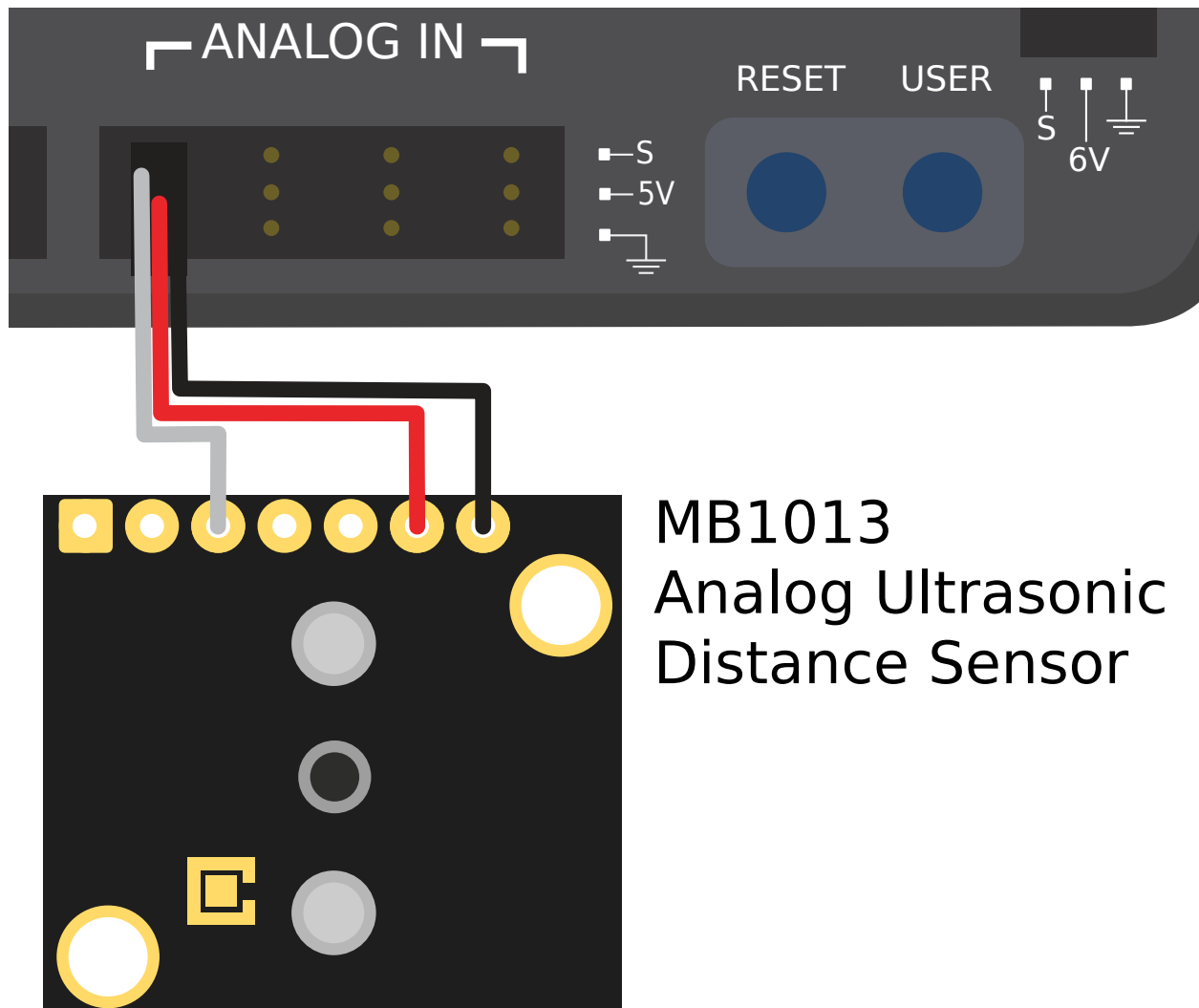
The roboRIO has 4 built-in analog input ports (numbered 0-3), as seen in the image above. Each port has three pins - signal ("S"), power ("V"), and ground ("G"). The "power" and "ground" pins are used to power the peripheral sensors that connect to the analog input ports - there is a constant 5V potential difference between the "power" and the "ground" pins². The signal pin is the pin on which the signal is actually measured.

² All power pins are actually connected to a single rail, as are all ground pins - there is no need to use the power/ground pins corresponding to a given signal pin.

Connecting a sensor to a single analog input port

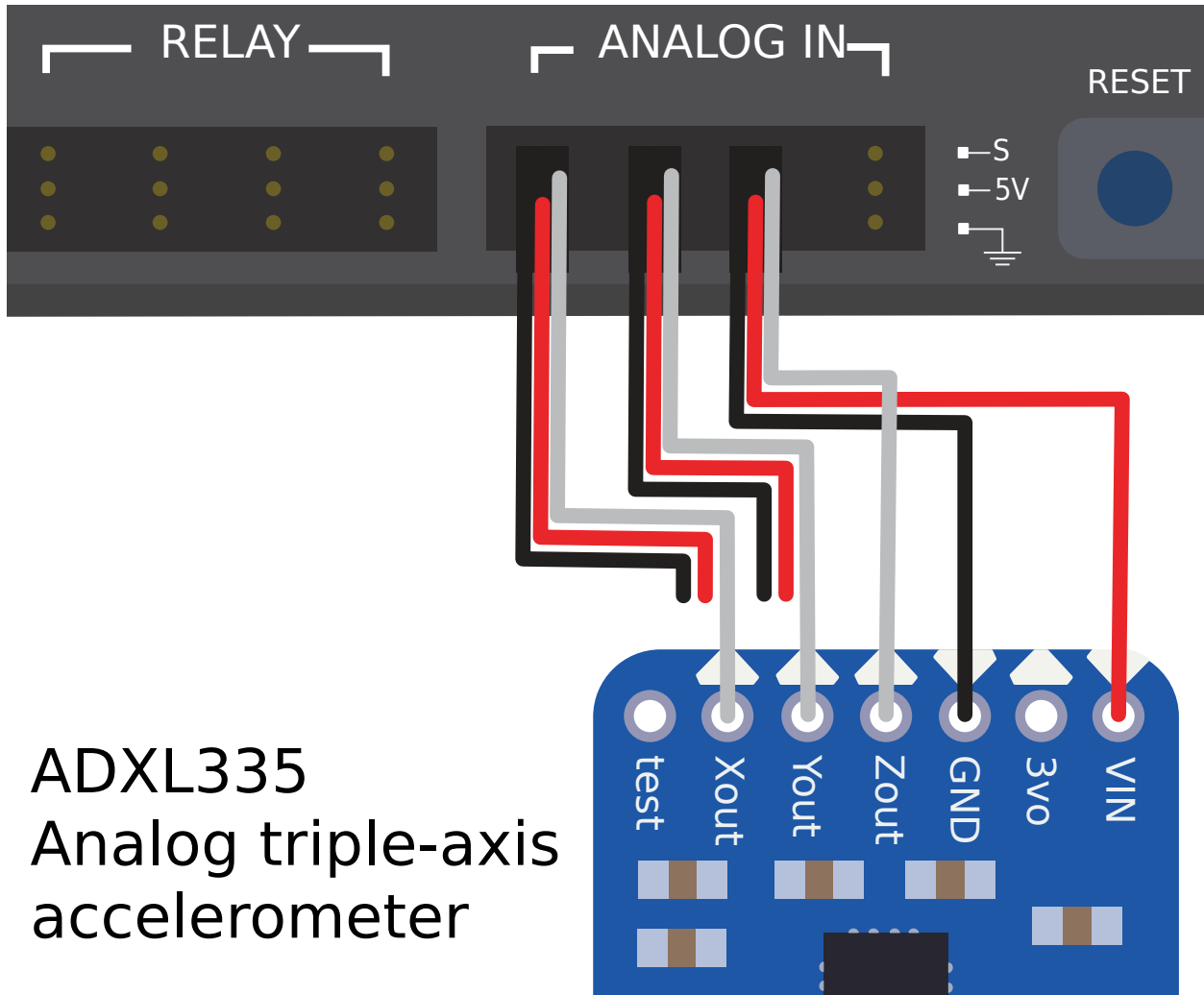
Note: Some sensors (such as *potentiometers*) may have interchangeable power and ground connections.

Most sensors that connect to analog input ports will have three wires - signal, power, and ground - corresponding precisely to the three pins of the analog input ports. They should be connected accordingly.



Connecting a sensor to multiple analog input ports

Some sensors may need to connect to multiple analog input ports in order to function. In general, these sensors will only ever require a single power and a single ground pin - only the signal pin of the additional port(s) will be needed. The image below shows an analog accelerometer that requires three digital input ports, but similar wiring can be used for analog sensors requiring two analog input ports.



42.2.2 Footnotes

42.3 Analog Potentiometers - Hardware

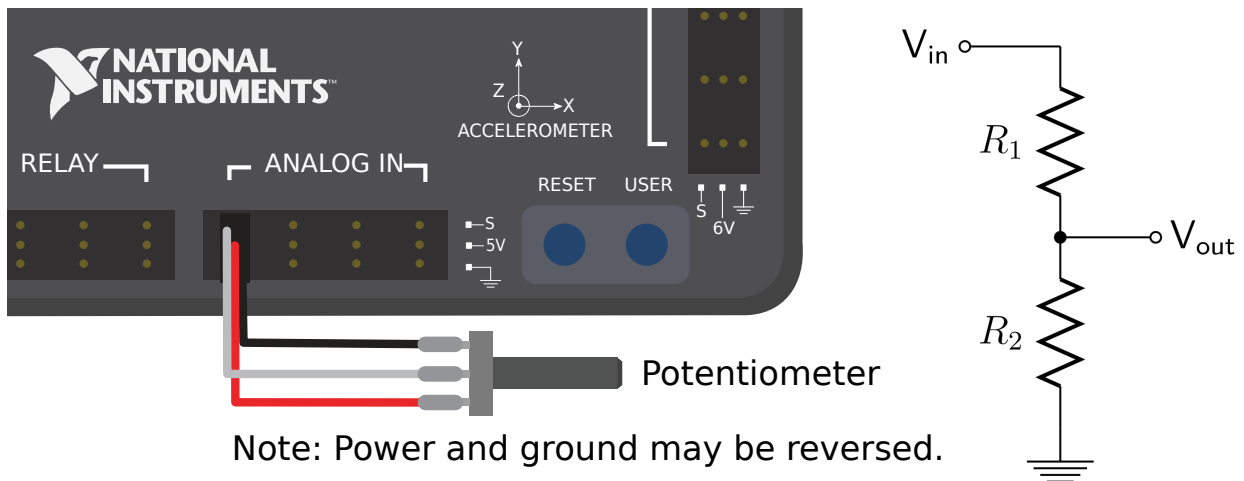
Note: This section covers analog potentiometer hardware. For a software guide to analog potentiometers, see [Analog Potentiometers - Software](#).

Warning: Potentiometers generally have a mechanically-limited travel range. Users should be careful that their mechanisms do not turn their potentiometers past their maximum travel, as this will damage or destroy the potentiometer.

Apart from *quadrature encoders*, another common way of measuring rotation on FRC® robots is with analog potentiometers. A potentiometer is simply a variable resistor - as the shaft of the potentiometer turns, the resistance changes (usually linearly). Placing this resistor in a *voltage divider* allows the user to easily measure the resistance by measuring the voltage across the potentiometer, which can then be used to calculate the rotational position of the shaft.

42.3.1 Wiring an analog potentiometer

As suggested by the names, analog potentiometers connect to the roboRIO's *analog input* ports. To understand how exactly to wire potentiometers, however, it is important to understand their internal circuitry.



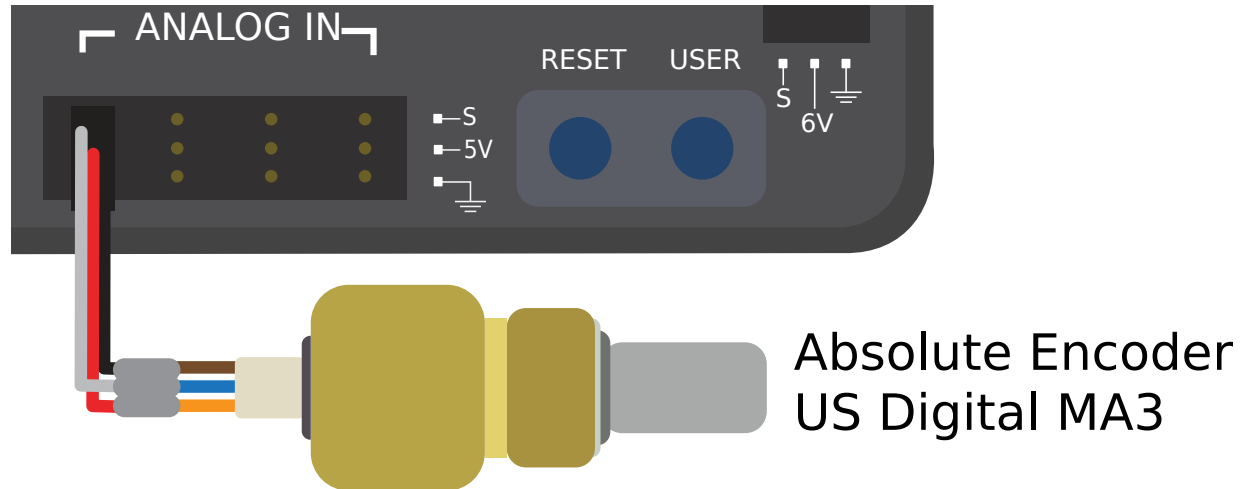
The picture above on the left shows a typical potentiometer. There are three pins, just as there are on the RIO's analog inputs. The middle pin is the signal pin, while the outer pins can both be *either* power or ground.

As mentioned before, a potentiometer is a voltage divider, as shown in the circuit diagram on the right. As the potentiometer shaft turns, the resistances R_1 and R_2 change; however, their sum remains constant¹. Thus, the voltage across the entire potentiometer remains constant (for the roboRIO, this would be 5 volts), but the voltage between the signal pin and either the voltage or ground pin varies linearly as the shaft turns.

Since the circuit is symmetric, it is reversible - this allows the user to choose at which end of the travel the measured voltage is zero, and at which end it is 5 volts. To reverse the directionality of the sensor, it can simply be wired backwards! Be sure to check the directionality of your potentiometer with a multimeter to be sure it is in the desired orientation before soldering your wires to the contacts.

¹ The way this actually works is generally by having the shaft control the position of a contact along a resistive "wiper" of fixed length along which the current flows - the resistance is proportional to the length of wiper between the contact and the end of the wiper.

42.3.2 Absolute encoders



An “absolute encoder” is an encoder that measures the absolute position of the encoder shaft, rather than the incremental movement (as a *quadrature encoder*) does. In this respect, absolute encoders are more similar to potentiometers than to incremental encoders. Many absolute encoders offer a simple analog output - these can be used exactly in the same way as a potentiometer, except their wiring is not generally reversible. Absolute encoders have the advantage of lacking a hard travel limit - the signal will simply reset when the shaft crosses the zero point. The analog potentiometer pictured above can be found at [AndyMark](#).

Absolute encoders that do not offer a simple analog output require *more complicated communications with the RIO*.

42.3.3 Footnotes

42.4 Digital Inputs - Hardware

Note: This section covers digital input hardware. For a software guide to digital inputs, see *Digital Inputs - Software*.

A *digital signal* is a signal that can be in one of several discrete states. In the vast majority of cases, the signal is the voltage in a wire, and there are only two states for a digital signal - high, or low (also denoted 1 and 0, or true and false, respectively).

The roboRIO’s built-in digital input-output ports (or “DIO”) ports function on 5V, so “high” corresponds to a signal of 5V, and “low” to a signal of 0V¹².

¹ More-precisely, the signal reads “high” when it rises above 2.0V, and reads “low” when it falls back below 0.8V - behavior between these two thresholds is not guaranteed to be consistent.

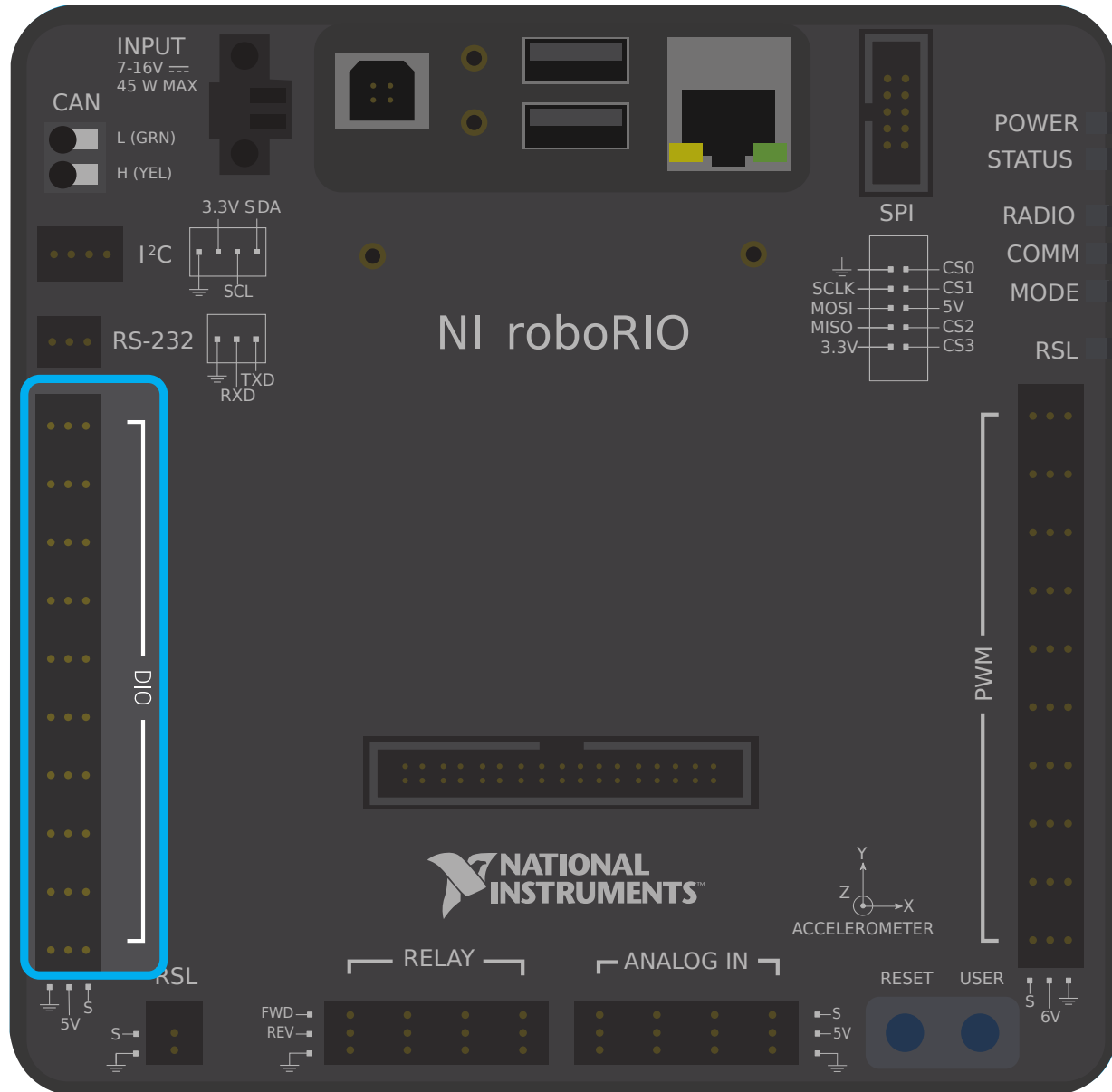
² The roboRIO also offers 3.3V logic via the “MXP” expansion port; however, use of this is far less-common than the 5V.

42.4.1 Connecting to the roboRIO DIO ports

Note: Additional DIO ports are available through the “MXP” expansion port. To use these, a breakout board of some sort that connects to the MXP is needed.

Warning: Always consult the technical specifications of the sensor you are using *before* wiring the sensor, to ensure that the correct wire is being connected to each pin. Failure to do so can result in damage to the device.

Warning: **Never** directly connect the power pin to the ground pin on any port on the roboRIO! This will trigger protection features on the roboRIO and may result in unexpected behavior.



The roboRIO has 10 built-in DIO ports (numbered 0-9), as seen in the image above. Each port has three pins - signal ("S"), power ("V"), and ground ("⏏"). The "power" and "ground" pins are used to power the peripheral sensors that connect to the DIO ports - there is a constant 5V potential difference between the "power" and the "ground" pins³ - the "power" pin corresponds to the "high" state (5V), and the "ground" to "low" (0V). The signal pin is the pin on which the signal is actually measured (or, when used as an output, the pin that sends the signal).

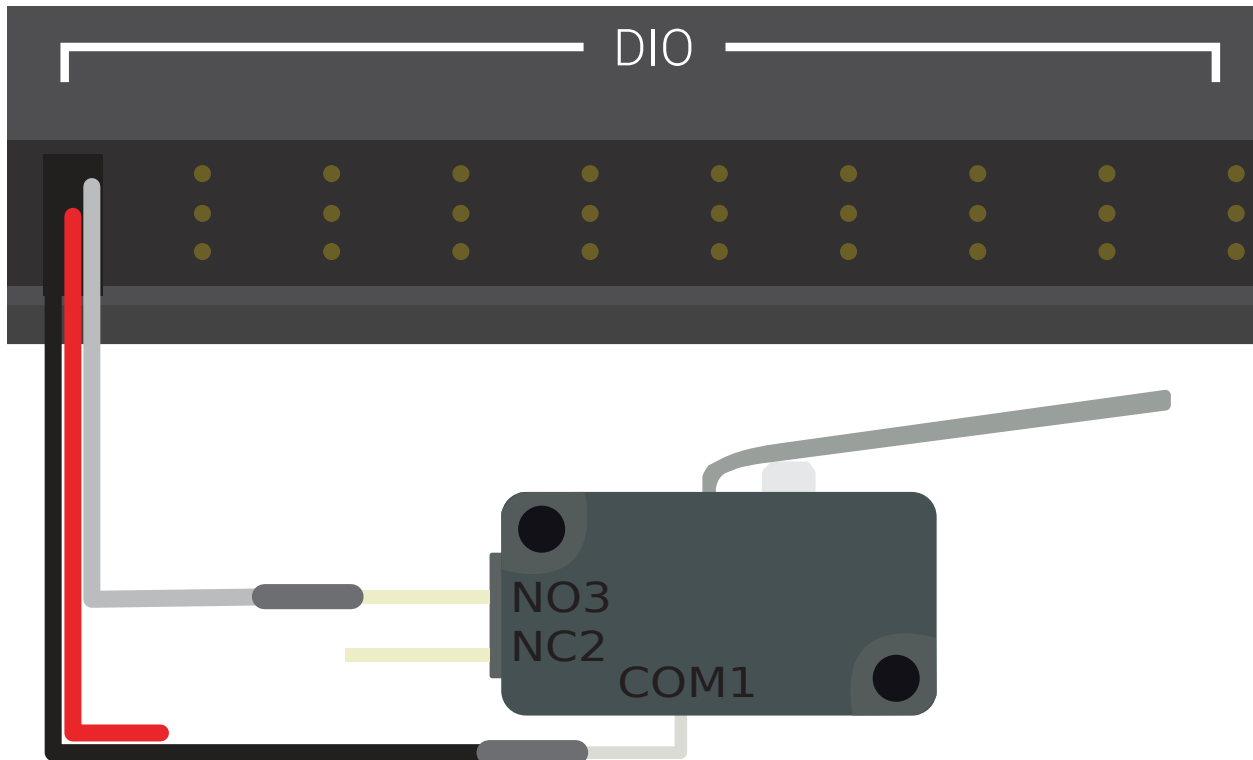
All DIO ports have built in "pull-up" resistors between the power pins and the signal pins - these ensure that when the signal pin is "floating" (i.e. is not connected to any circuit), they consistently remain in a "high" state.

³ All power pins are actually connected to a single rail, as are all ground pins - there is no need to use the power/ground pins corresponding to a given signal pin.

Connecting a simple switch to a DIO port

The simplest device that can be connected to a DIO port is a switch (such as a *limit switch*). When a switch is connected correctly to a DIO port, the port will read “high” when the circuit is open, and “low” when the circuit is closed.

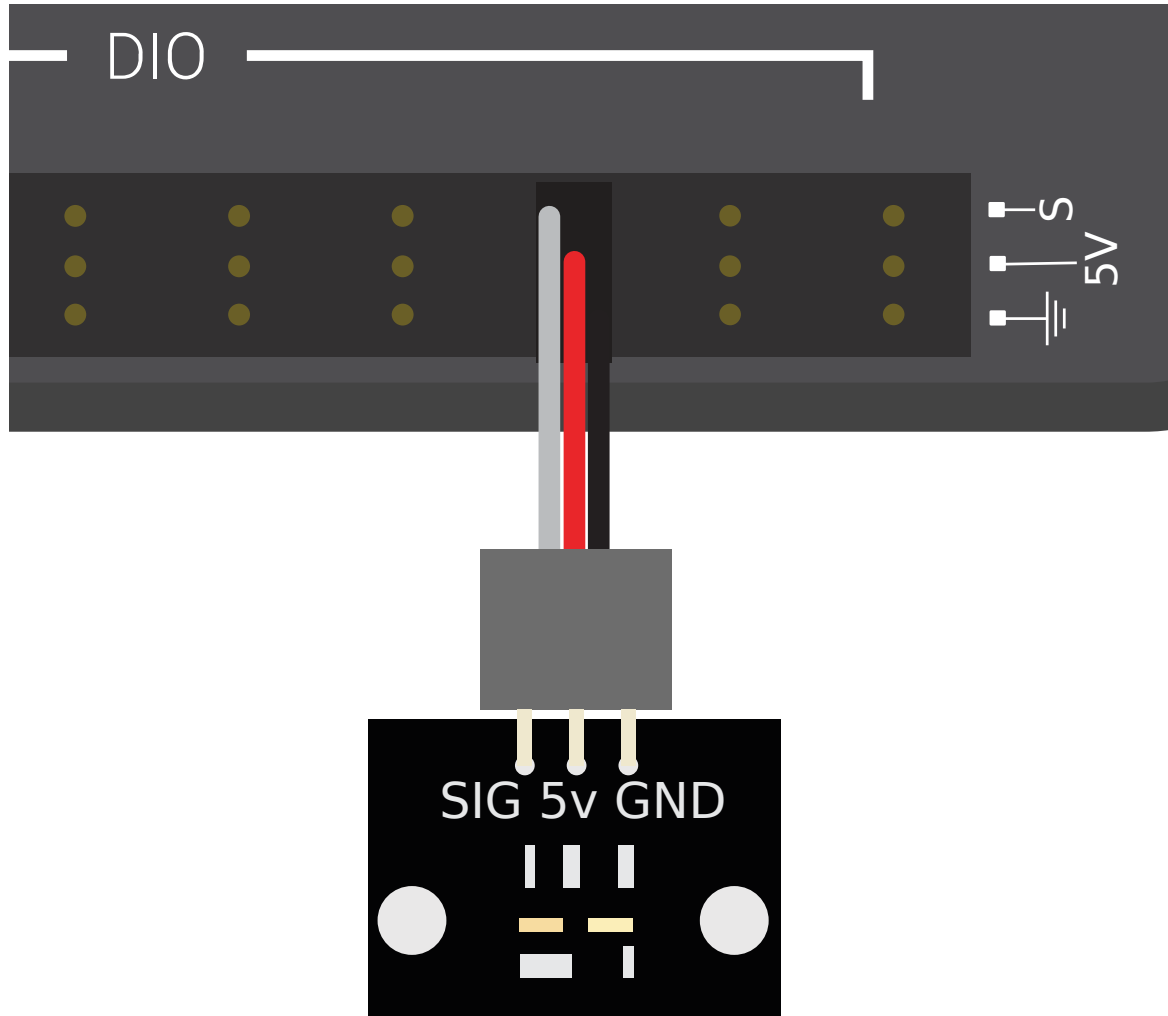
A simple switch does not need to be powered, and thus only has two wires. Switches should be wired between the *signal* and the *ground* pins of the DIO port. When the switch circuit is open, the signal pin will float, and the pull-up resistor will ensure that it reads “high.” When the switch circuit is closed, it will connect directly to the ground rail, and thus read “low.”



Limit Switch or Micro Switch

Connecting a powered sensor to a DIO port

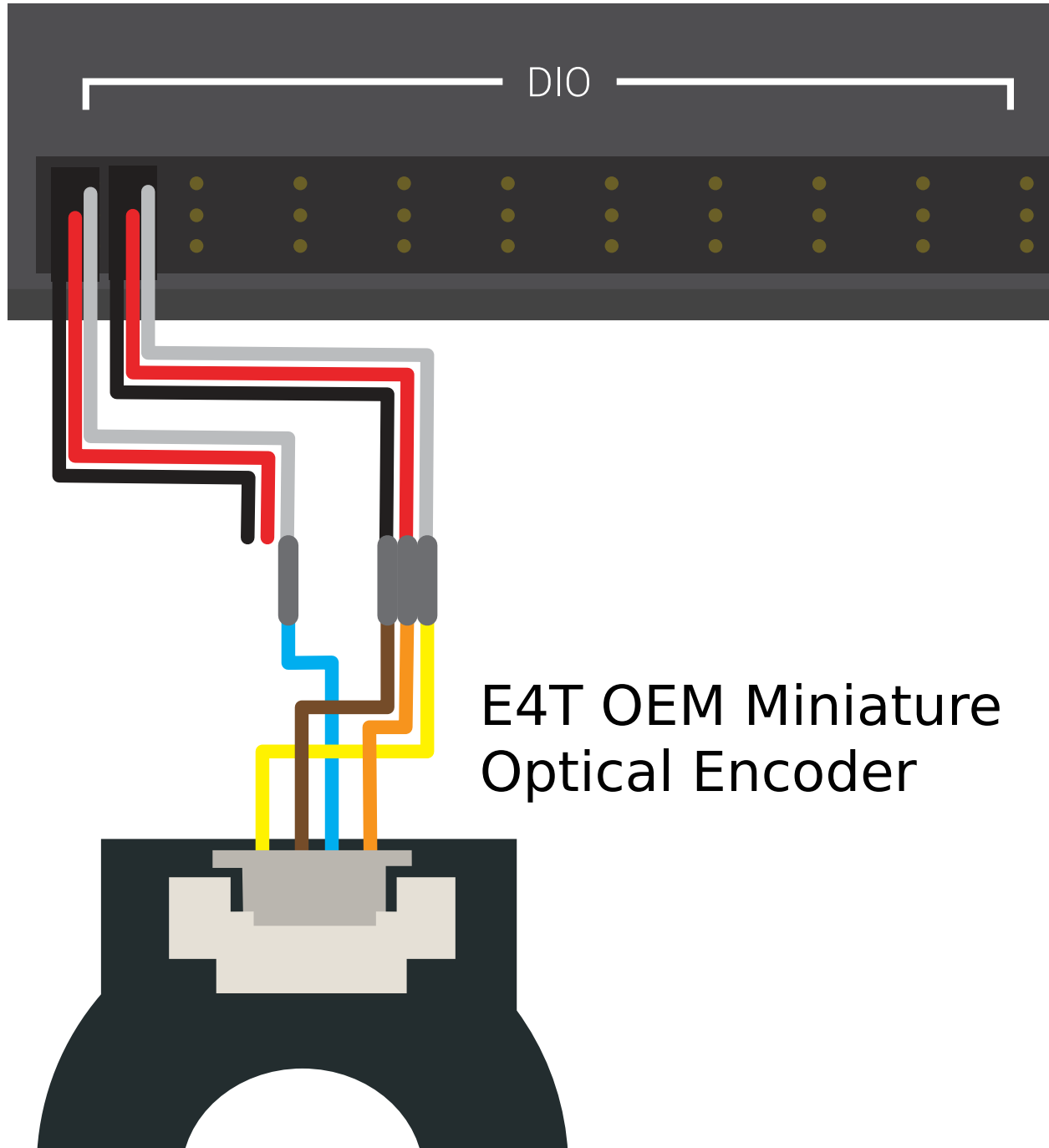
Many digital sensors (such as most no-contact proximity switches) require power in order to work. A powered sensor will generally have three wires - signal, power, and ground. These should be connected to the corresponding pins of the DIO port.



WCP Hall Effect Sensor

Connecting a sensor that uses multiple DIO ports

Some sensors (such as *quadrature encoders*) may need to connect to multiple DIO ports in order to function. In general, these sensors will only ever require a single power and a single ground pin - only the signal pin of the additional port(s) will be needed.



42.4.2 Footnotes

42.5 Proximity Switches - Hardware

Note: This section covers proximity switch hardware. For a guide to using proximity switches in software, see *Digital Inputs - Software*.

One of the most common sensing tasks on a robot is detecting when an object (be it a mechanism, game piece, or field element) is within a certain distance of a known point on the robot. This type of sensing is accomplished by a “proximity switch.”

42.5.1 Proximity switch operation

Proximity switches are switches - they operate a circuit between an “open” state (in which there *is not* connectivity across the circuit) and a “closed” one (in which there *is*). Thus, proximity switches generate a digital signal, and accordingly, they are almost always connected to the roboRIO’s *digital input* ports.

Proximity switches can be either “normally-open,” in which activating the switch closes the circuit, or “normally closed,” in which activating the switch opens the circuit. Some switches offer *both* a NO and a NC circuit connected to the same switch. In practice, the effective difference between a NO and a NC switch is the behavior of the system in the case that the wiring to the switch fails, as a wiring failure will almost always result in an open circuit. NC switches are often “safer,” in that a wiring failure causes the system to behave as if the switch were pressed - as switches are often used to prevent a mechanism from damaging itself, this mitigates the chance of damage to the mechanism in the case of a wiring fault.

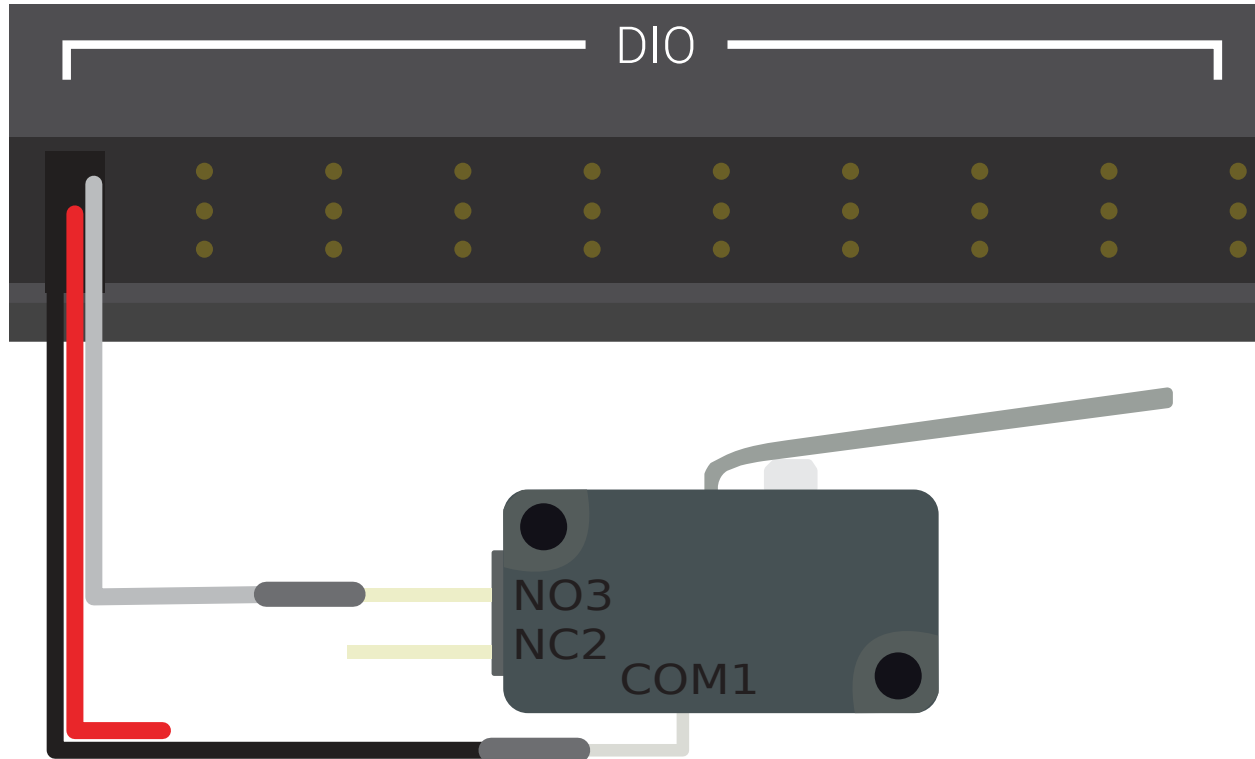
The digital inputs on the roboRIO have pull-up resistors that will make the input be high (1 value) when the switch is open, but when the switch closes the value goes to 0 since the input is now connected to ground.

42.5.2 Types of Proximity Switches

There are several types of proximity switches that are commonly-used in FRC®:

- *Mechanical Proximity Switches (“limit switches”)*
- *Magnetic Proximity Switches*
- *Inductive Proximity Switches*
- *Photoelectric Proximity Switches*
- *Time-of-flight Proximity Switches*

Mechanical Proximity Switches (“limit switches”)



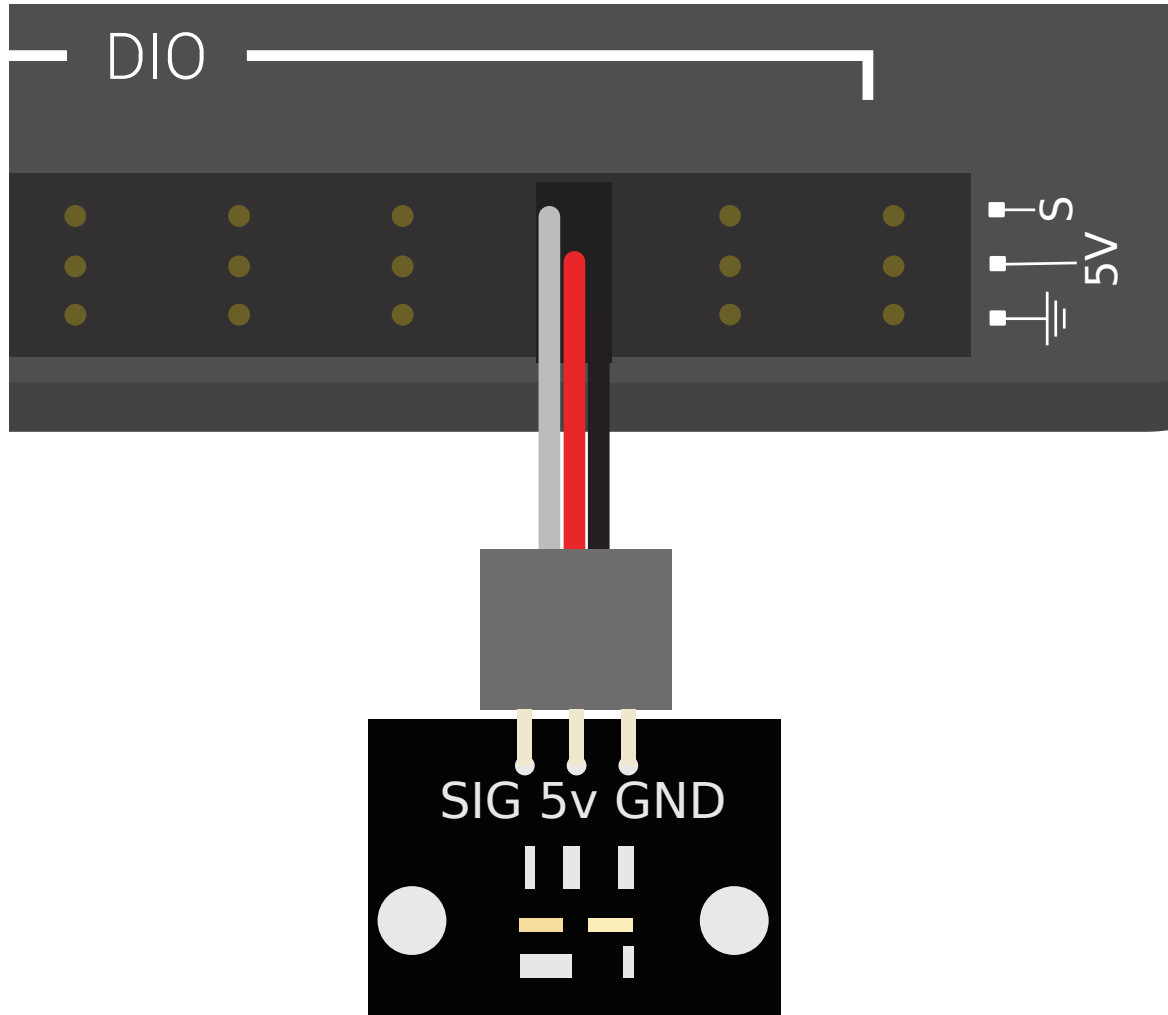
Limit Switch or Micro Switch

Mechanical proximity switches (more commonly known as “limit switches”) are probably the most-commonly used proximity switch in FRC, due to their simplicity, ease-of-use, and low cost. A limit switch is quite simply a switch attached to a mechanical arm, usually at the limits of travel. The switch is activated when an object pushes against the switch arm, actuating the switch.

Limit switches vary in size, the geometry of the switch-arm, and in the amount of “throw” required to activate the switch. While limit switches are quite cheap, their mechanical actuation is sometimes less-reliable than no-contact alternatives. However, they are also extremely versatile, as they can be triggered by any physical object capable of moving the switch arm.

See this [article](#) for writing the software for Limit Switches.

Magnetic Proximity Switches



WCP Hall Effect Sensor

Magnetic proximity switches are activated when a magnet comes within a certain range of the sensor. Accordingly, they are “no-contact” switches - they do not require contact with the object being sensed.

There are two major types of magnetic proximity switches - reed switches and hall-effect sensors. In a reed switch, the magnetic field causes a pair of flexible metal contacts (the “reeds”) to touch each other, closing the circuit. A hall-effect sensor, on the other hand, detects the induced voltage transversely across a current-carrying conductor. Hall-effect sensors are generally the cheaper and more-reliable of the two. Pictured above is the [Hall effect sensor from West Coast Products](#).

Magnetic proximity switches may be either “unipolar,” “bipolar,” or “omnipolar.” A unipolar switch activates and deactivates depending on the presence of a given pole of the magnet (either north or south, depending on the switch). A bipolar switch activates from the proximity of one pole, and deactivates from the proximity of the opposite pole. An omnipolar switch will

activate in the presence of either pole, and deactivates when no magnet is present.

While magnetic proximity switches are often more reliable than their mechanical counterparts, they require the user to mount a magnet on the object to be sensed - thus, they are mostly used for sensing mechanism location.

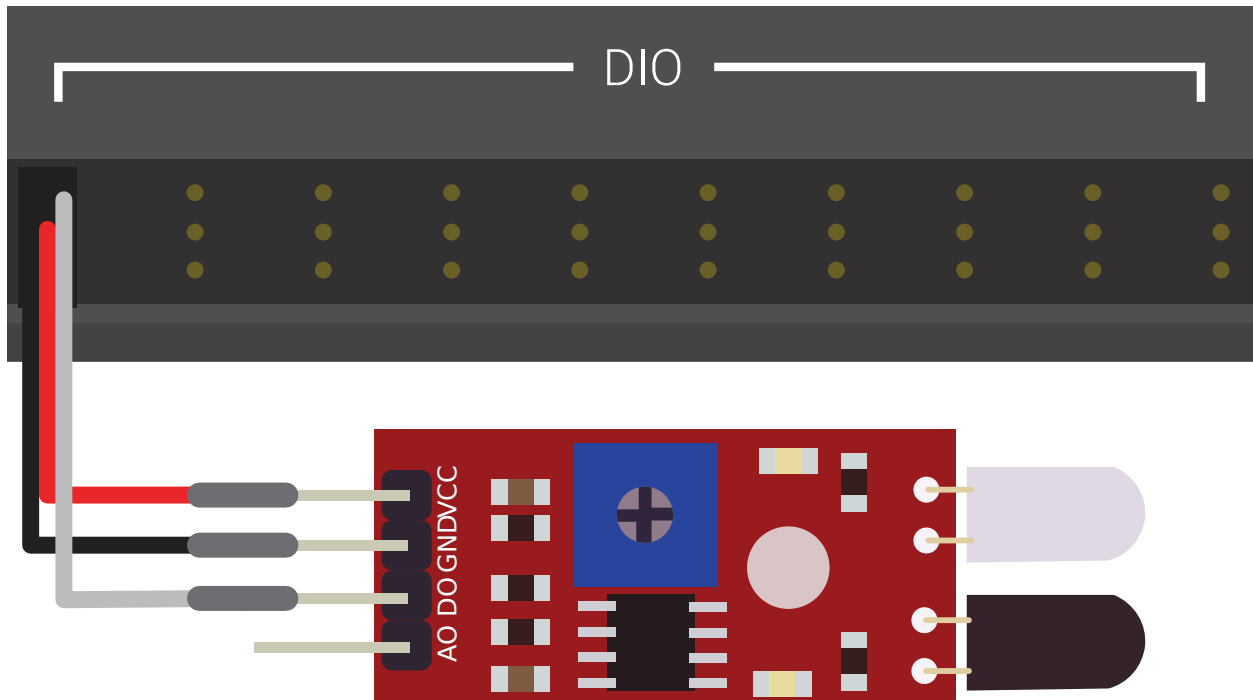
Inductive Proximity Switches



Inductive proximity switches are activated when a conductor of any sort comes within a certain range of the sensor. Like magnetic proximity switches, they are “no-contact” switches.

Inductive proximity switches are used for many of the same purposes as magnetic proximity switches. Their more-general nature (activating in the presence of any conductor, rather than just a magnet) can be either a help or a hindrance, depending on the nature of the application.

Photoelectric Proximity Switches

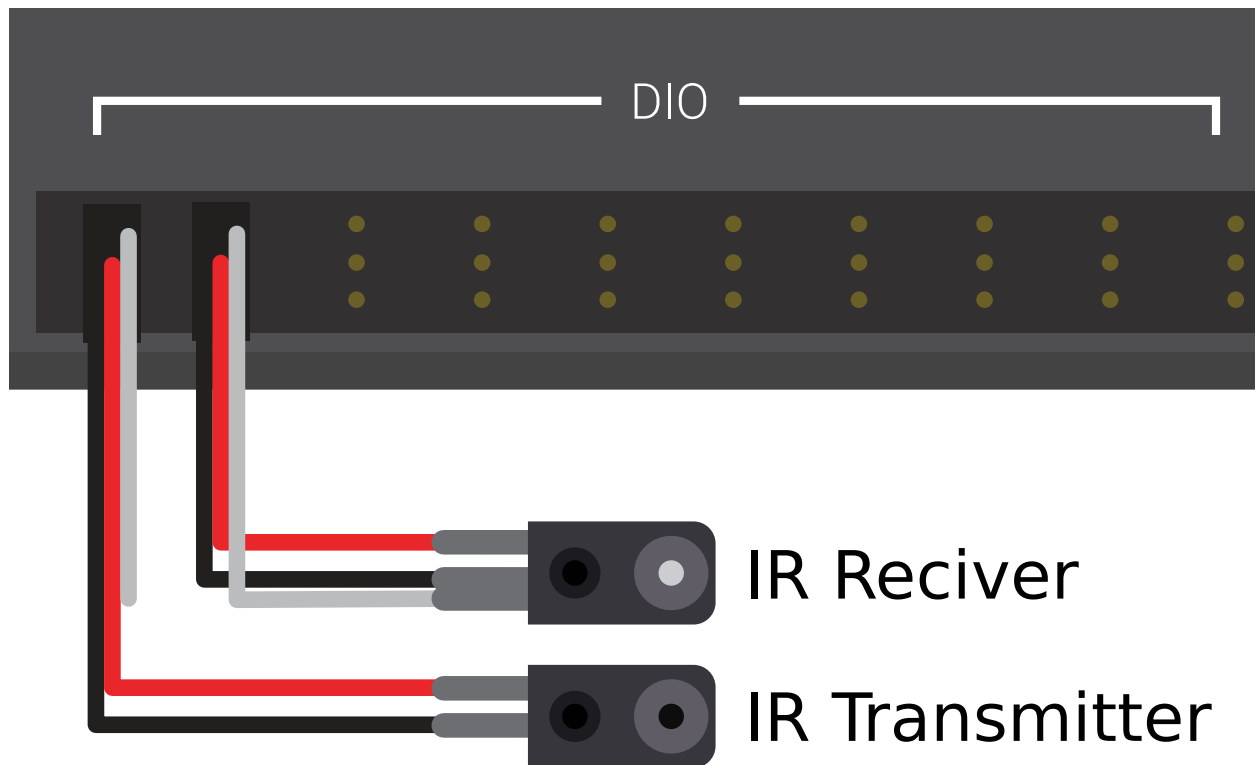


IR Digital Obstacle Avoidance Sensor

Photoelectric proximity switches are another type of no-contact proximity switch in widespread use in FRC. Photoelectric proximity switches contain a light source (usually an IR laser) and a photoelectric sensor that activates the switch when the detected light (which bounces off of the sensor target) exceeds a given threshold. One such sensor is the [IR Obstacle Avoidance Module](#) pictured below.

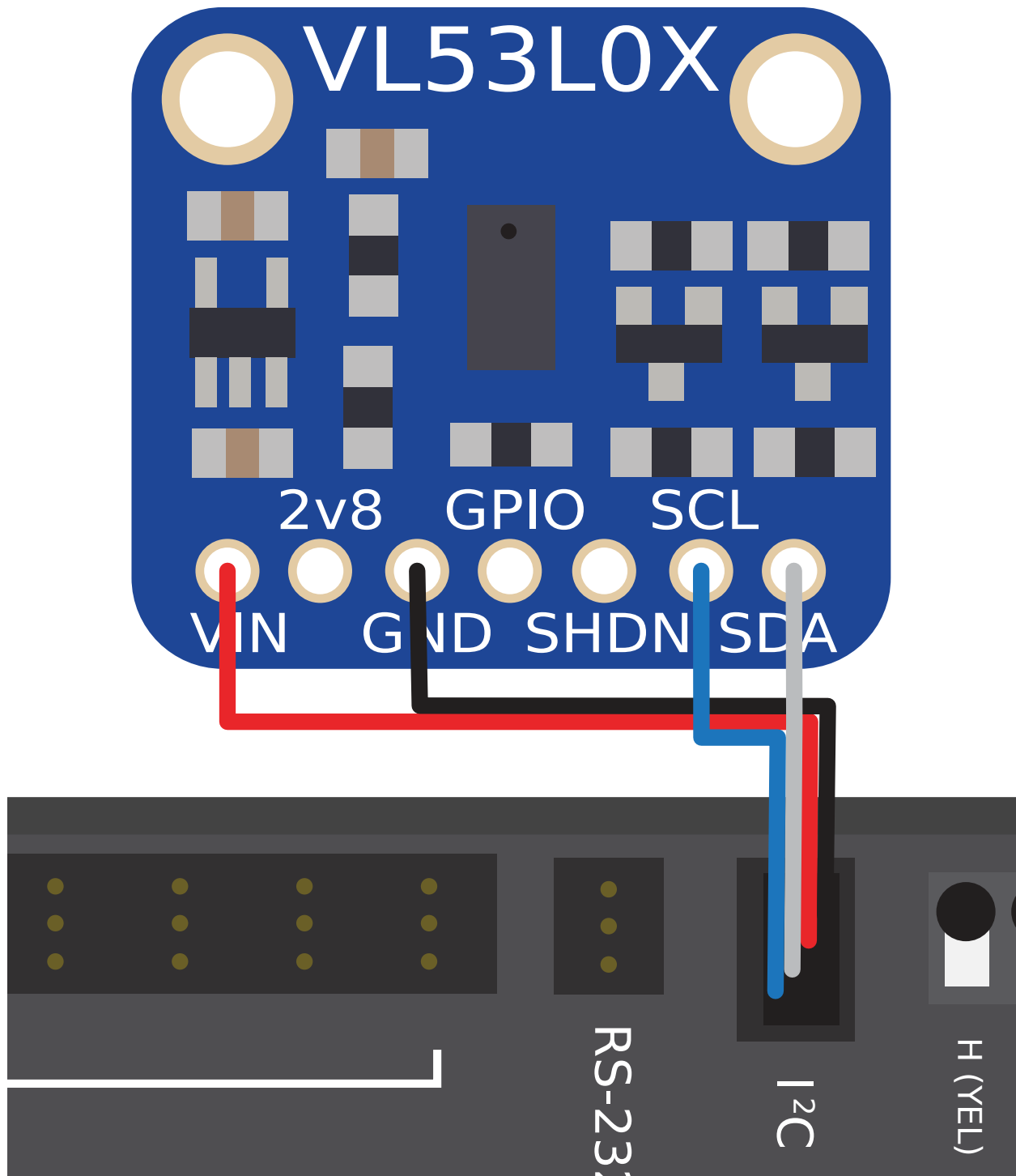
Since photoelectric proximity switches rely on measuring the amount of reflected light, they are often inconsistent in their triggering range between different materials - accordingly, most photoelectric sensors have an adjustable activation point (typically controlled by turning a screw somewhere on the sensor body). On the other hand, photoelectric sensors are also extremely versatile, as they can detect a greater variety of objects than the other types of no-contact switches.

Photoelectric sensors are also often used in a “beam break” configuration, in which the emitter is separate from the sensor. These typically activate when an object is interposed between the emitter and the sensor. Pictured above is a [beam break sensor with an IR LED transmitter and IR receiver](#).



Time-of-flight Proximity Switches

Time of Flight Distance Sensor



Time-of-flight Proximity Switches are newer to the market and are not commonly found in

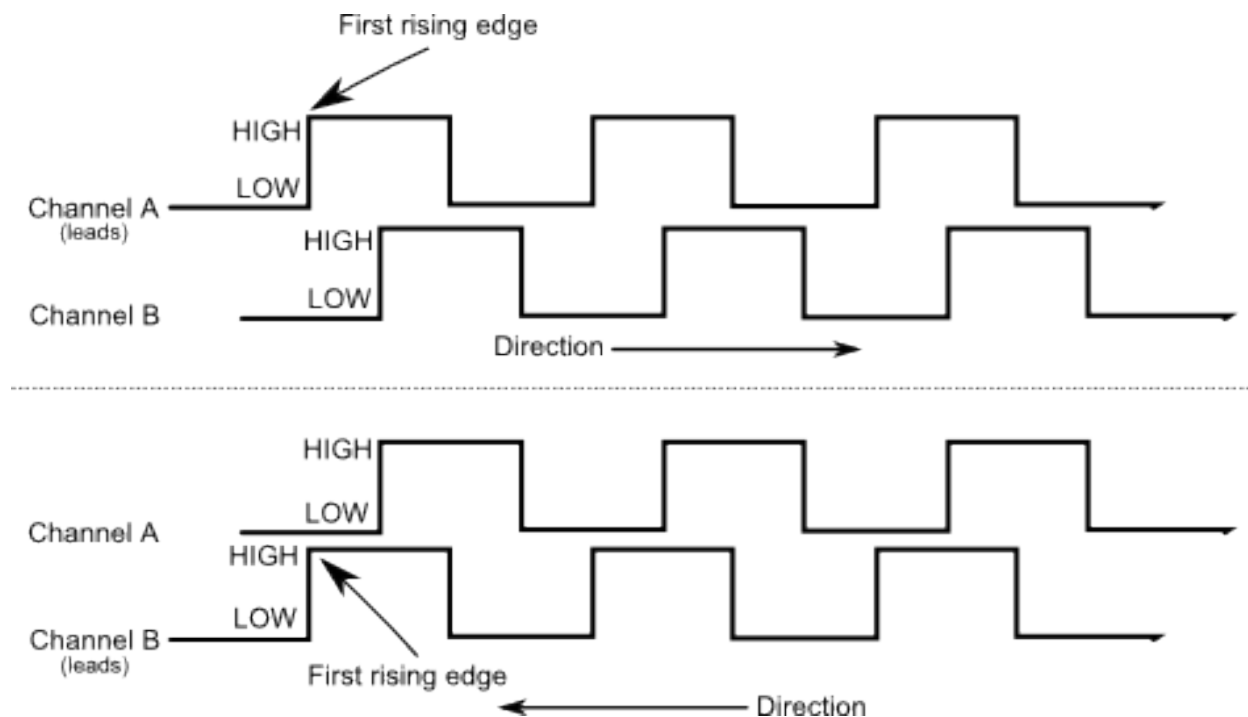
FRC. They use a concentrated light source, such as a small laser, and measure the time between the emission of light and when the receiver detects it. Using the speed of light, it can produce a very accurate distance measurement for a very small target area. Range on this type of sensor can range greatly, between 30mm to around 1000mm for the [VL53L0X sensor](#) pictured above. There are also longer range version available. More information about time of flight sensors can be found in [this article](#) and more about the circuitry can be found in [this article](#).

42.6 Encoders - Hardware

Note: This section covers encoder hardware. For a software guide to encoders, see [Encoders - Software](#).

Quadrature encoders are by far the most common method for measuring rotational motion in FRC®, and for good reason - they are cheap, easy-to-use, and reliable. As they produce digital signals, they are less-prone to noise and interference than analog devices (such as [potentiometers](#)).

The term “quadrature” refers to the method by which the motion is measured/encoded. A quadrature encoder produces two square-wave pulses that are 90-degrees out-of-phase from each other, as seen in the picture below:



Thus, across both channels, there are four total “edges” per period (hence “quad”). The use of two out-of-phase pulses allows the direction of motion to be unambiguously determined from which pulse “leads” the other.

As each square wave pulse is a digital signal, quadrature encoders connect to the [digital input](#) ports on the RIO.

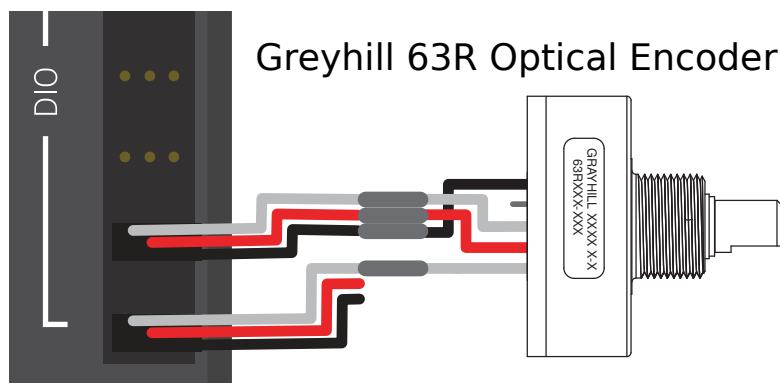
42.6.1 Types of Encoders

There are three types of quadrature encoders typically used in FRC:

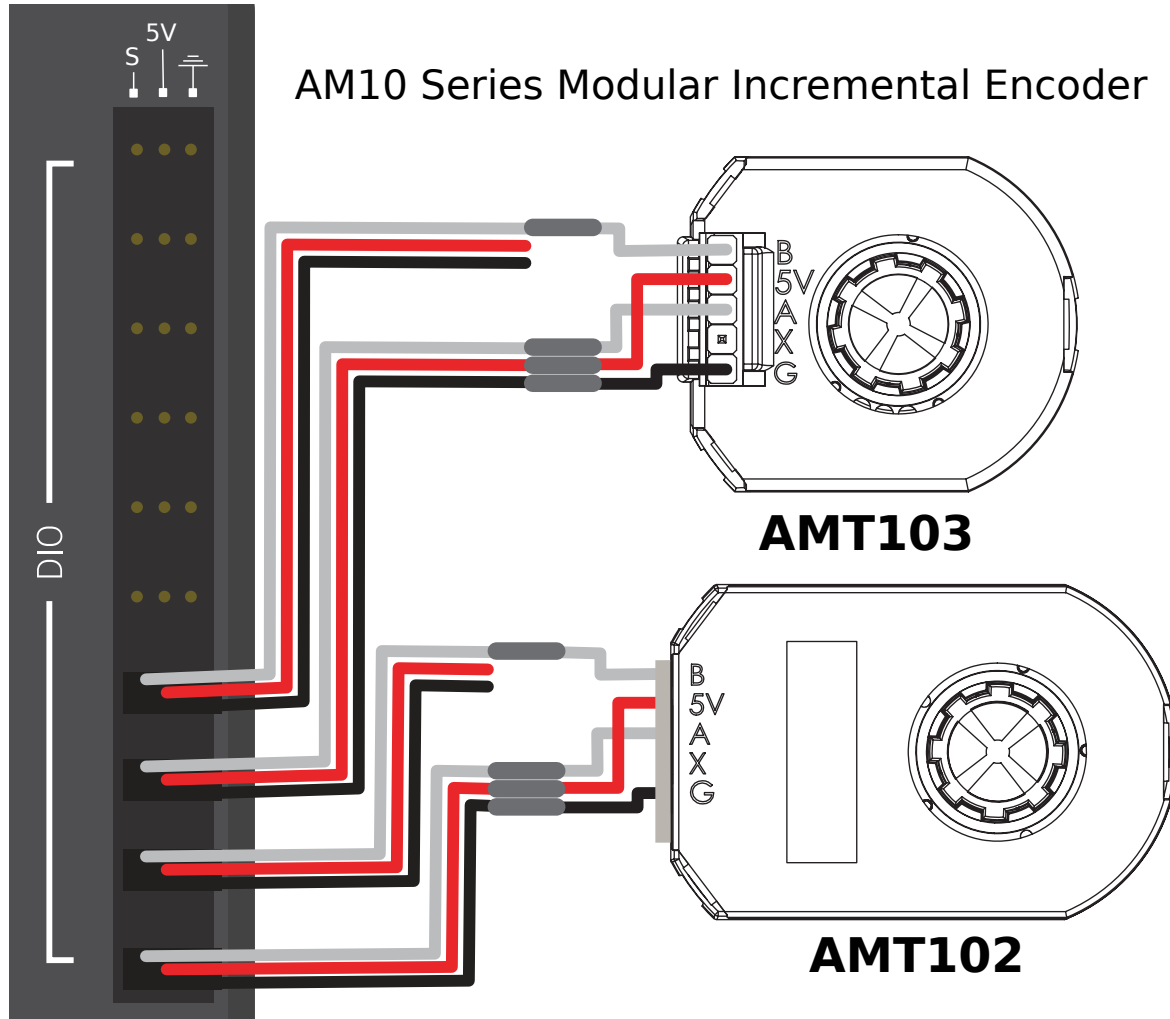
- *Shafted encoders*
- *On-shaft encoders*
- *Magnetic encoders*

These encoders vary in how they are mounted to the mechanism in question. In addition to these types of encoders, many FRC mechanisms come with quadrature encoders integrated into their design.

Shafted encoders

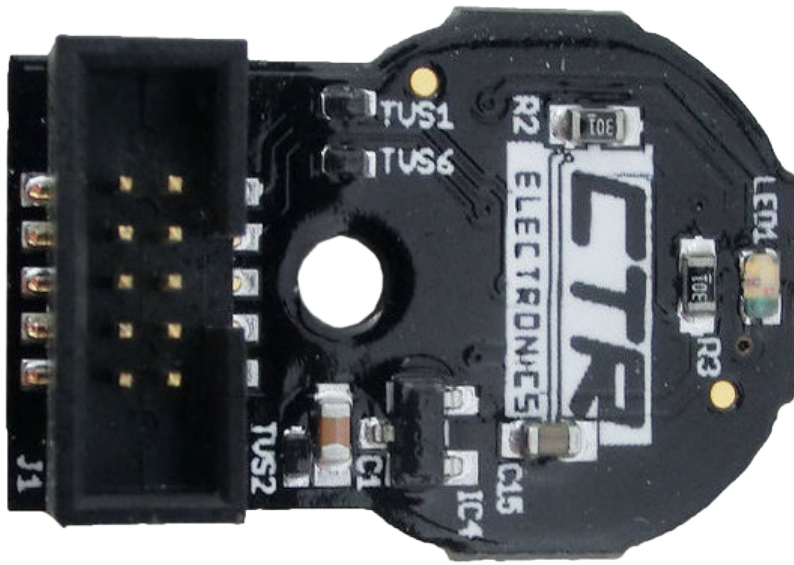


Shafted encoders have a sealed body with a shaft protruding out of it that must be coupled rotationally to a mechanism. This is often done with a helical beam coupling, or, more cheaply, with a length of flexible tubing (such as surgical tubing or pneumatic tubing), fastened with cable ties and/or adhesive at either end. Many commercial off-the-shelf FRC gearboxes have purpose-built mounting points for shafted encoders, such as the popular [Grayhill 63r](#), pictured above.

On-shaft encoders

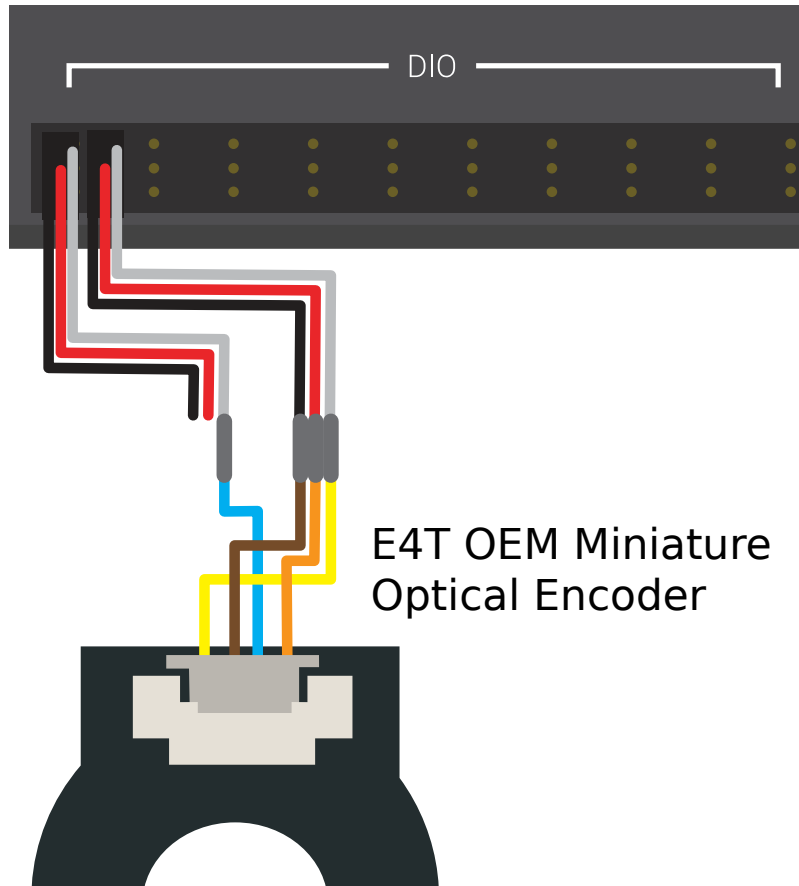
On-shaft encoders (such as the [AMT103-V](#) available through FIRST Choice) couple to a shaft by fitting *around* it, forming a friction coupling between the shaft and a rotating hub inside the encoder.

Magnetic encoders



Magnetic encoders require no mechanical coupling to the shaft at all; rather, they track the orientation of a magnet fixed to the shaft. The [CTRE Mag Encoder](#) is a popular option, with many FRC products offering built-in mounting options for it. While the no-contact nature of magnetic encoders can be handy, they often require precise construction in order to ensure that the magnet is positioned correctly with respect to the encoder.

42.6.2 Encoder Wiring



Encoders that need two digital inputs, such as the [E4T OEM Miniature Optical Encoder](#), can be wired to two digital input ports. Other encoders, such as the on-shaft ones shown above, often need [an analog input port](#). CTRE Magnetic encoders shown above can be wired to a [TalonSRX data port](#) with a ribbon cable.

42.6.3 Encoder Resolution

Warning: The acronyms “CPR” and “PPR” are *both* used by varying sources to denote both edges per revolution *and* cycles per revolution, so the acronym alone is not enough to tell which is of the two is meant when by a given value. When in doubt, consult the technical manual of your specific encoder.

As encoders measure rotation with digital pulses, the accuracy of the measurement is limited by the number of pulses per given amount of rotational movement. This is known as the “resolution” of the encoder, and is traditionally measured in one of two different ways: edges per revolution, or cycles per revolution.

Edges per revolution refers to the total number of transitions from high-to-low or low-to-high across both channels per revolution of the encoder shaft. A full period contains *four* edges.

Cycles per revolution refers to the total number of *complete periods* of both channels per revolution of the encoder shaft. A full period is *one cycle*.

Thus, a resolution stated in edges per revolution has a value four times that of the same resolution stated in cycles per revolution.

In general, the resolution of your encoder in edges-per-revolution should be somewhat finer than your smallest acceptable error in positioning. Thus, if you want to know the mechanism plus-or-minus one degree, you should have an encoder with a resolution somewhat higher than 360 edges per revolution.

42.7 Gyroscopes - Hardware

Note: This section covers gyro hardware. For a software guide to gyros, see [Gyroscopes - Software](#).

Gyroscopes (or “gyros”, for short) are devices that measure rate-of-rotation. These are particularly useful for stabilizing robot driving, or for measuring heading or tilt by integrating (adding-up) the rate measurements to get a measurement of total angular displacement.

Several popular FRC® devices known as *IMUs* (Inertial Measurement Units) combine 3-axis gyros, accelerometers and other position sensors into one device. Some popular examples are:

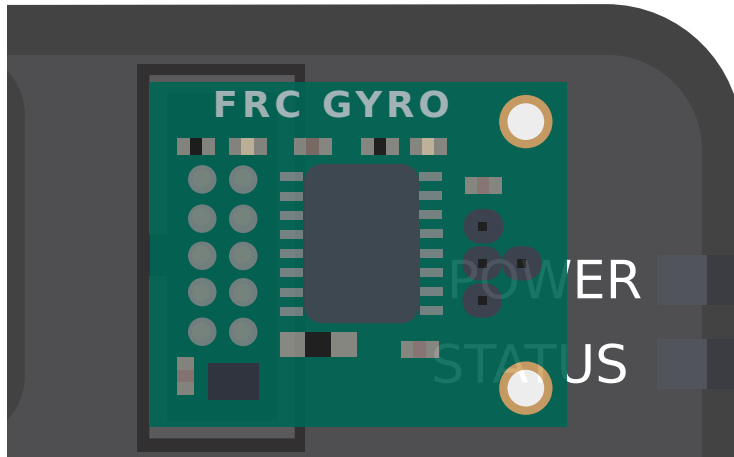
- [Analog Devices ADIS16448 and ADIS 16470 IMUs](#)
- [CTRE Pigeon IMU](#)
- [Kauai Labs NavX](#)

42.7.1 Types of Gyros

There are two types of Gyros commonly-used in FRC: single-axis gyros, three-axis gyros and IMUs, which often include a 3-axis gyro.

Single-axis Gyros

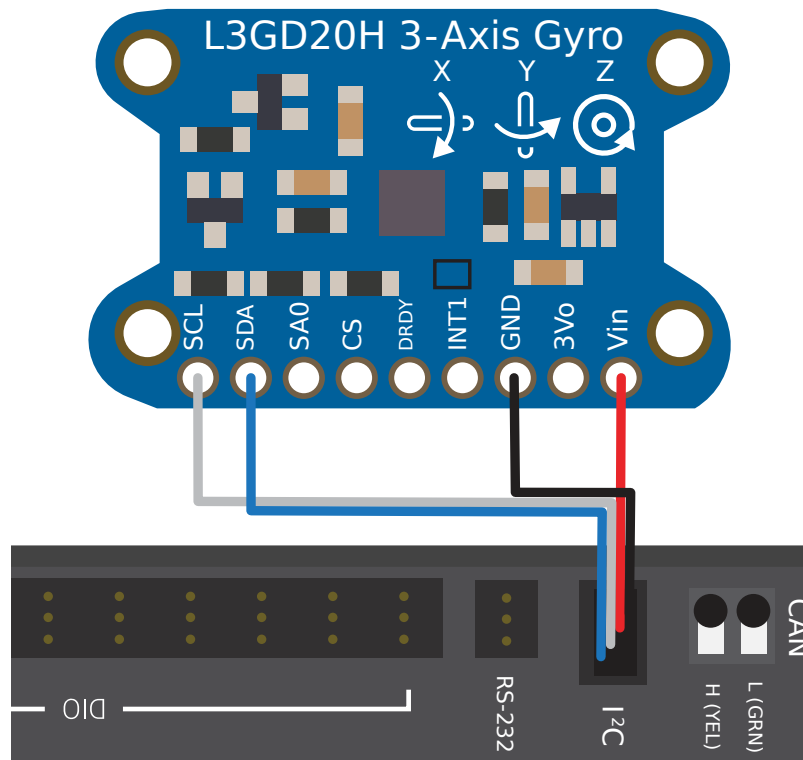
Analog Devices 1-axis SPI Gyro



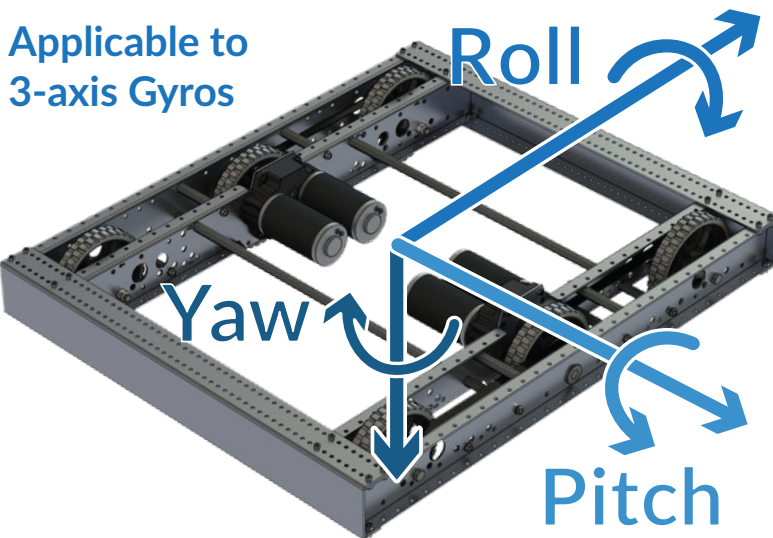
As per their name, single-axis gyros measure rotation rate around a single axis. This axis is generally specified on the physical device, and mounting the device in the proper orientation so that the desired axis is measured is highly important. Some single-axis gyros can output an analog voltage corresponding to the measured rate of rotation, and so connect to the roboRIO's *analog input* ports. Other single-axis gyros, such as as the [ADXRS450](#) pictured above, use the *SPI port* on the roboRIO instead.

The [Analog Devices ADXRS450 FRC Gyro Board](#) that has been in FIRST Choice in recent years is a commonly used single axis gyro.

Three-axis Gyros



Three-axis gyros measure rotation rate around all three spacial axes (typically labeled x, y, and z). The motion around these axis is called pitch, yaw, and roll.



Note: The coordinate system shown above is often used for three axis gyros, as it is a convention in avionics. Note that other coordinate systems are used in mathematics and referenced throughout WPILib. Please refer to the [Drive class axis diagram](#) for axis referenced

in software.

Peripheral three-axis gyros may simply output three analog voltages (and thus connect to the *analog input ports*, or (more commonly) they may communicate with one of the roboRIO's *serial buses*.

42.8 Ultrasonics - Hardware

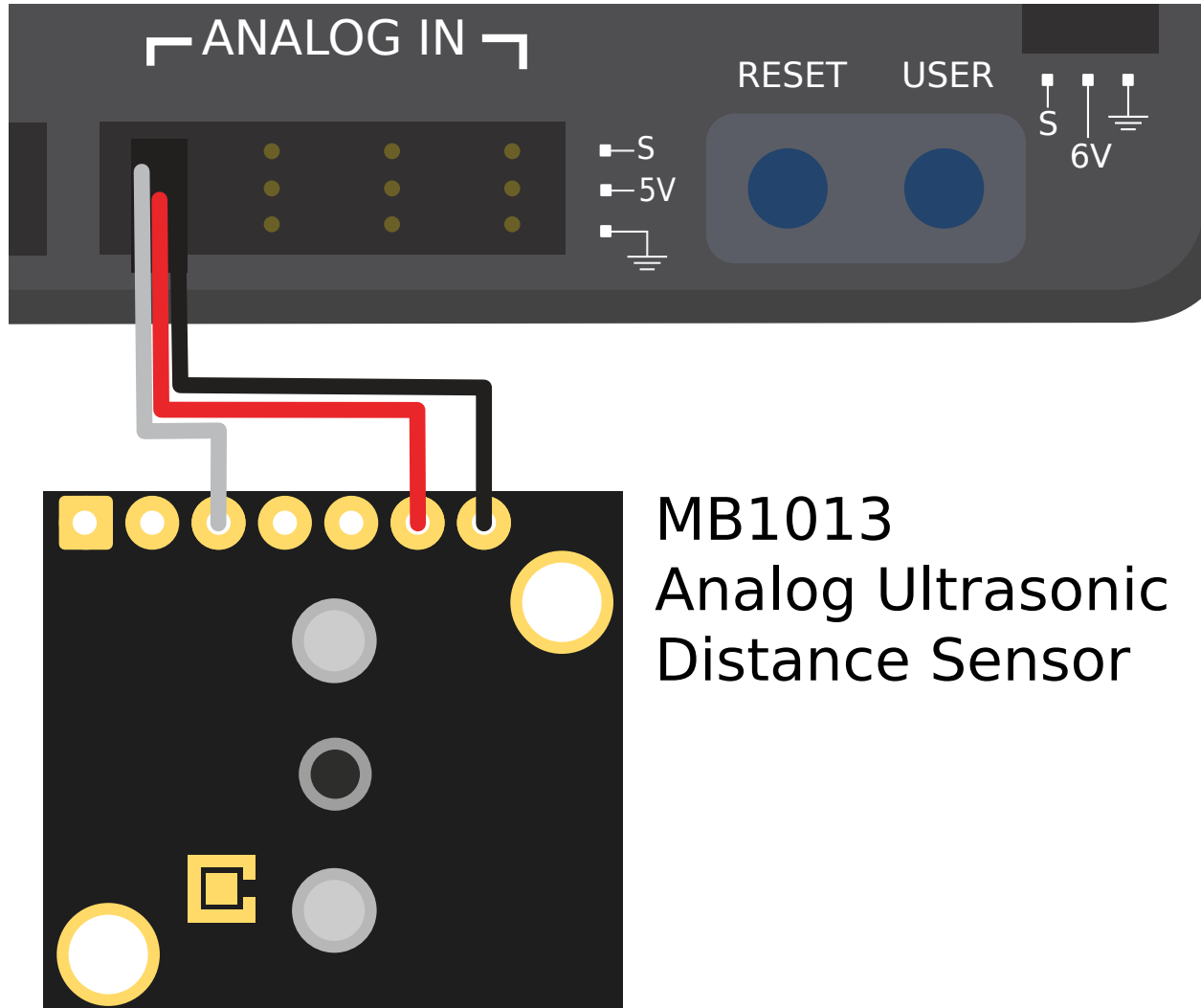
Note: This section covers ultrasonic sensor hardware. For a software guide to ultrasonics, see *Ultrasonics - Software*.

Ultrasonic rangefinders are some of the most common rangefinders used in FRC®. They are cheap, easy-to-use, and fairly reliable. Ultrasonic rangefinders work by emitting a pulse of high-frequency sound, and then measuring how long it takes the echo to reach the sensor after bouncing off the target. From the measured time and the speed of sound in air, it is possible to calculate the distance to the target.

42.8.1 Types of ultrasonics

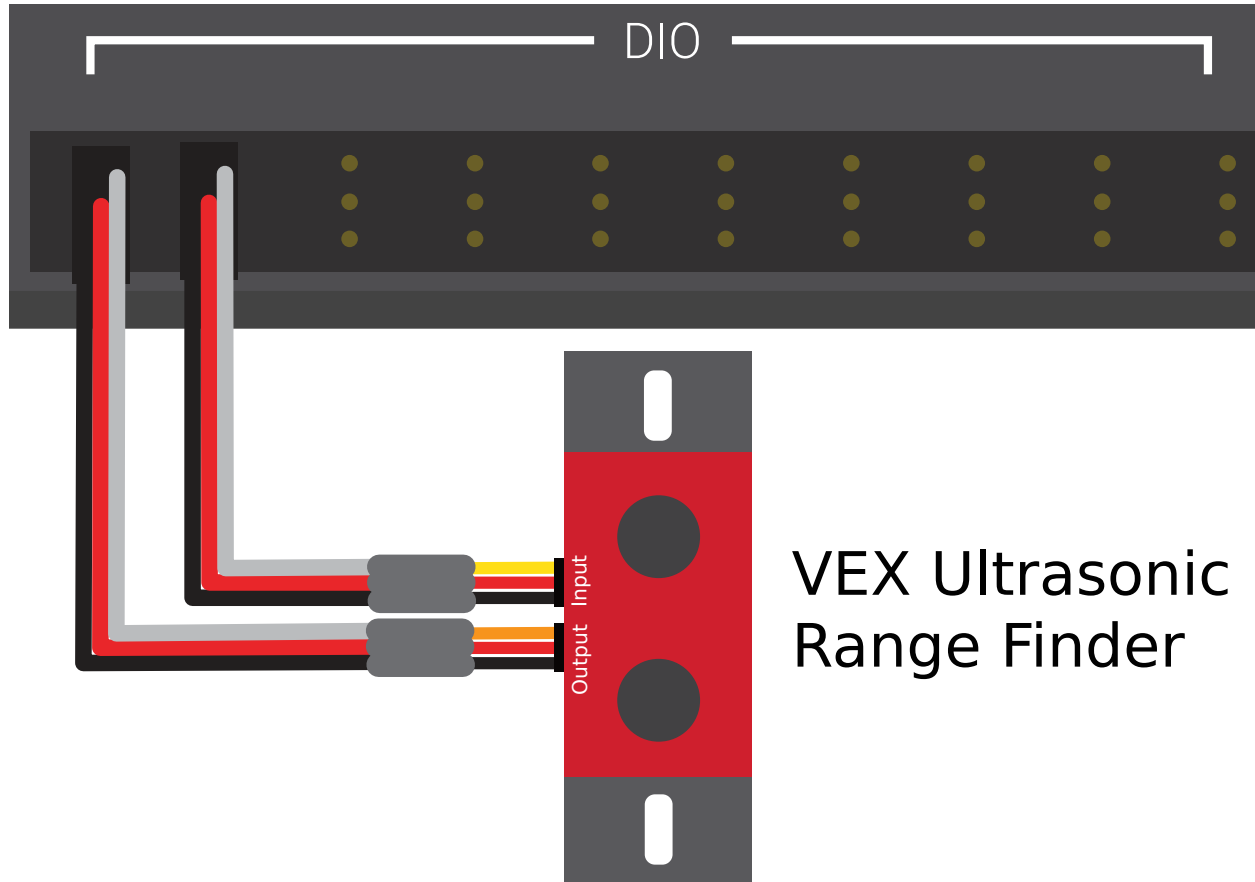
While all ultrasonic rangefinders operate on the “ping-response” principle outlined above, they may vary in the way they communicate with the roboRIO.

Analog ultrasonics



Analog ultrasonics output a simple analog voltage corresponding to the distance to the target, and thus connect to an *analog input* port. The user will need to calibrate the voltage-to-distance conversion in *software*.

Ping-response ultrasonics



VEX Ultrasonic
Range Finder

While, as mentioned, all ultrasonics are functionally ping-response devices, a “ping response” ultrasonic is one configured to connect to *both a digital input and a digital output*. The digital output is used to send the ping, while the input is used to read the response.

Serial ultrasonics



Some more-complicated ultrasonic sensors may communicate with the RIO over one of the *serial buses*, such as RS-232.

42.8.2 Caveats

Ultrasonic sensors are generally quite easy to use, however there are a few caveats. As ultrasonics work by measuring the time between the pulse and its echo, they generally measure distance only to the *closest* target in their range. Thus, it is extremely important to pick the right sensor for the job. The documentation for ultrasonic sensors will generally include a picture of the “beam pattern” that shows the shape of the “window” in which the ultrasonic will detect a target - pay close attention to this when selecting your sensor.

Ultrasonic sensors are also susceptible to interference from other ultrasonic sensors. In order to minimize this, the roboRIO can run ping-response ultrasonics in a “round-robin” fashion - however, in competition, there is no sure way to ensure that interference from sensors mounted on other robots does not occur.

Finally, ultrasonics may not be able to detect objects that absorb sound waves, or that redirect them in strange ways. Thus, they work best for detecting hard, flat objects.

42.9 Accelerometers - Hardware

Accelerometers are common sensors used to measure acceleration.

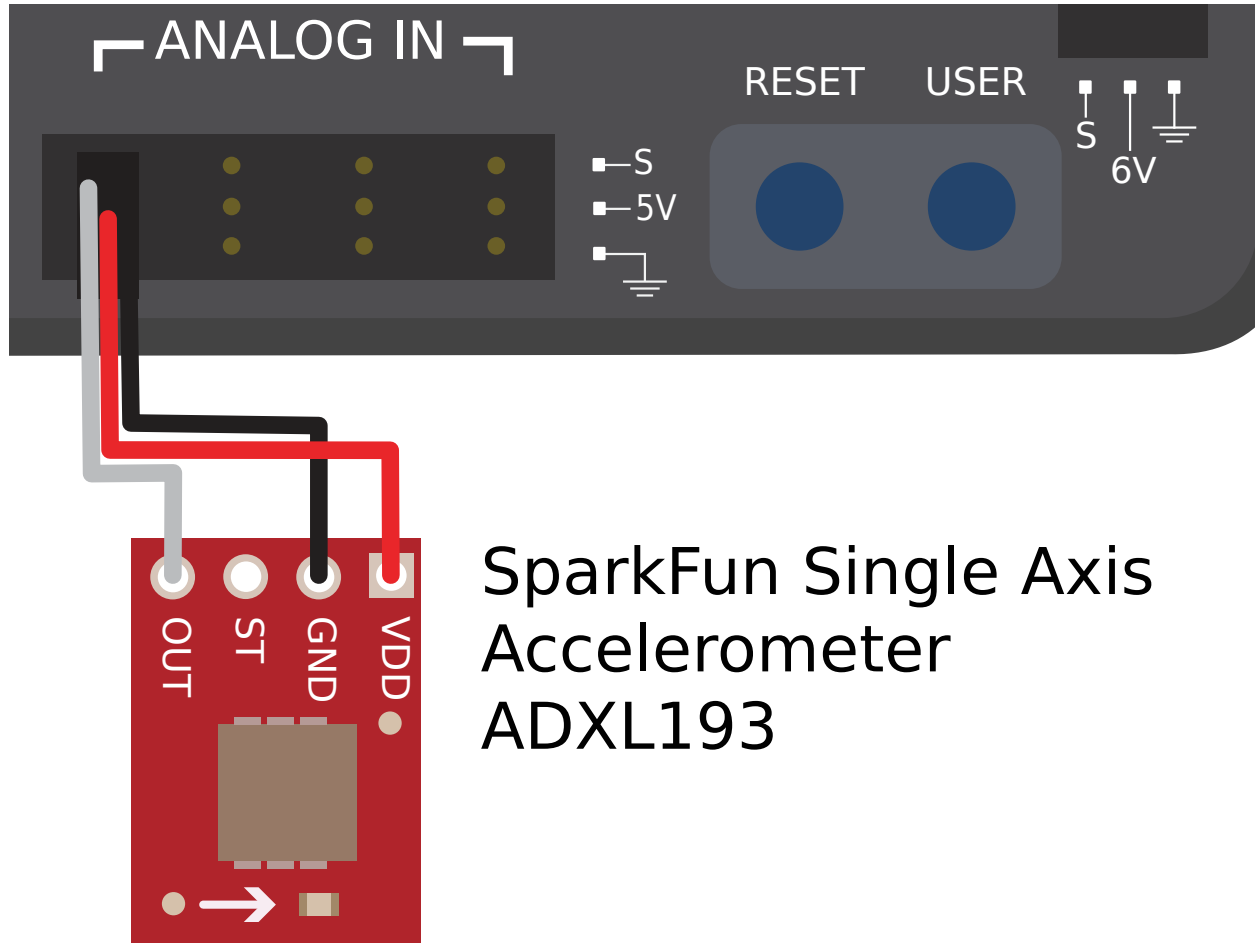
In principle, precise measurements of acceleration can be double-integrated and used to track position (similarly to how the measurement of turn rate from a gyroscope can be integrated to determine heading) - however, in practice, accelerometers that are available within the legal FRC® price range are not nearly accurate for this use. However, accelerometers are still useful for a number of tasks in FRC.

The roboRIO comes with a *built-in three-axis accelerometer* that all teams can use, however teams seeking more-precise measurements may purchase and use a peripheral accelerometer, as well.

42.9.1 Types of accelerometers

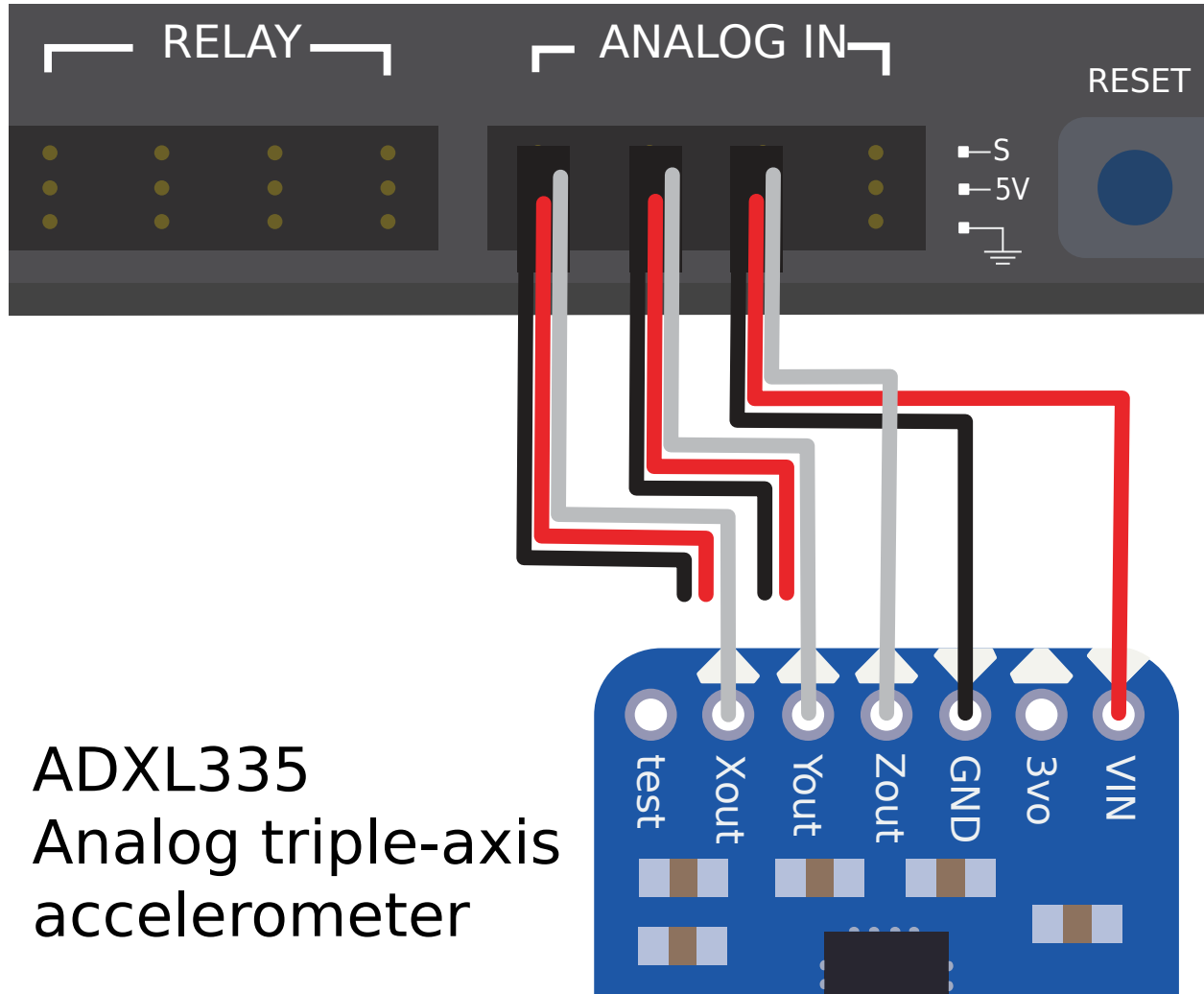
There are three types of accelerometers commonly-used in FRC: single-axis accelerometers, multi-axis accelerometers, and IMUs.

Single-axis accelerometers



As per their name, single-axis accelerometers measure acceleration along a single axis. This axis is generally specified on the physical device, and mounting the device in the proper orientation so that the desired axis is measured is highly important. Single-axis accelerometers generally output an analog voltage corresponding to the measured acceleration, and so connect to the roboRIO's *analog input* ports.

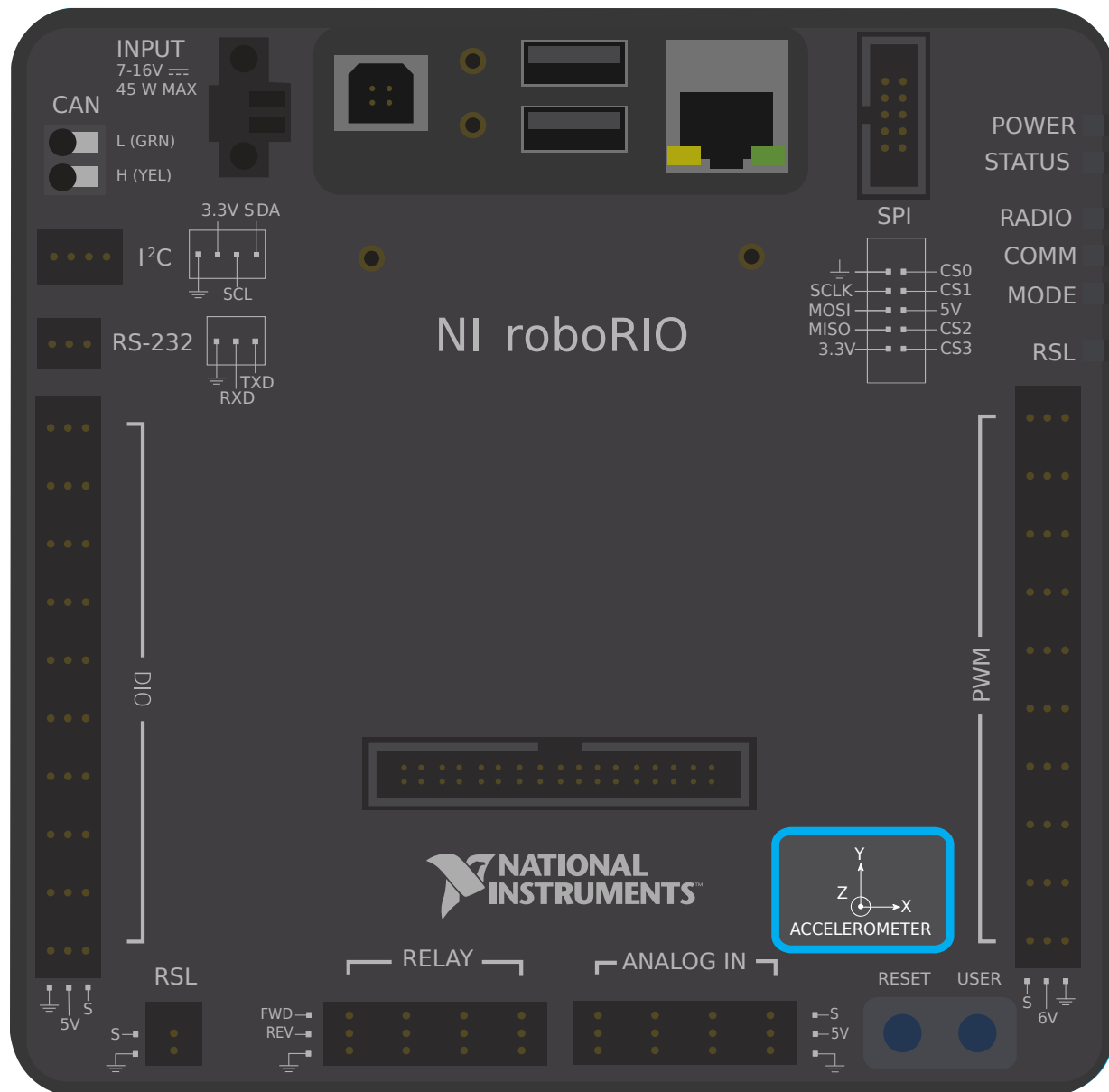
Multi-axis accelerometers



Multi-axis accelerometers measure acceleration along multiple spacial axes. The roboRIO's built-in accelerometer is a three-axis accelerometer.

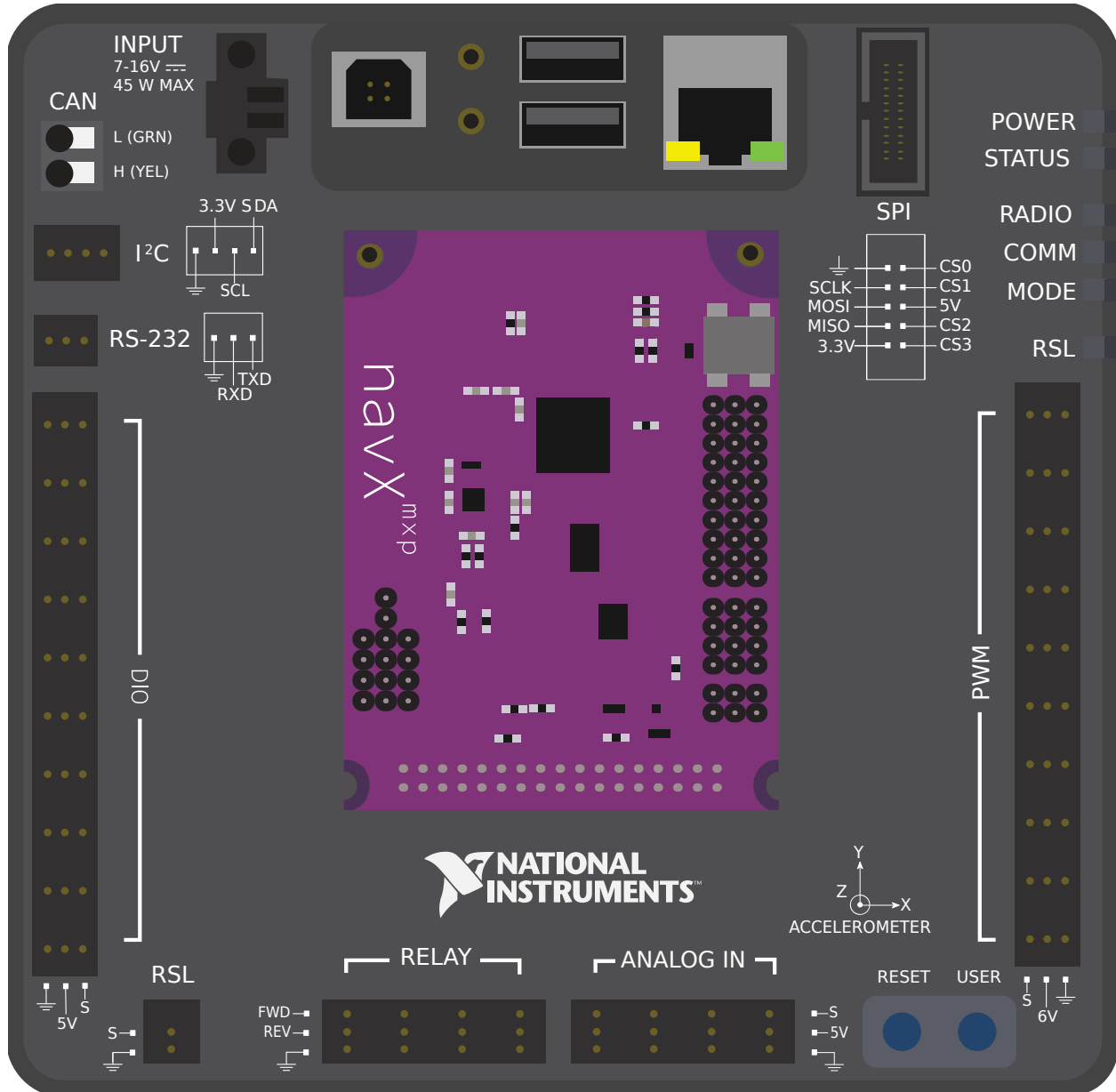
Peripheral multi-axis accelerometers may simply output multiple analog voltages (and thus connect to the *analog input ports*, or (more commonly) they may communicate with one of the roboRIO's *serial buses*.

roboRIO built-in accelerometer



The roboRIO has a built-in accelerometer, which does not need any external connections. You can find more details about how to use it in the [Built-in Accelerometer](#) section of the software documentation.

IMUs (Inertial Measurement Units)



Several popular FRC devices (known as “inertial measurement units,” or “IMUs”) combine both an accelerometer and a gyroscope. Popular FRC examples include:

- Analog Devices ADIS16448 and ADIS 16470 IMUs
- CTRE Pigeon IMU
- Kauai Labs NavX

42.10 LIDAR - Hardware

LIDAR (light detection and ranging) sensors are a variety of rangefinder seeing increasing use in FRC®.

LIDAR sensors work quite similarly to *ultrasonics*, but use light instead of sound. A laser is pulsed, and the sensor measures the time until the pulse bounces back.

42.10.1 Types of LIDAR

There are two types of LIDAR sensors commonly used in current FRC: 1-dimensional LIDAR, and 2-dimensional LIDAR.

1-Dimensional LIDAR



A 1-dimensional (1D) LIDAR sensor works much like an ultrasonic sensor - it measures the distance to the nearest object more or less along a line in front of it. 1D LIDAR sensors can often be more-reliable than ultrasonics, as they have narrower “beam profiles” and are less susceptible to interference. Pictured above is the [Garmin LIDAR-Lite Optical Distance Sensor](#).

1D LIDAR sensors generally output an analog voltage proportional to the measured distance, and thus connect to the roboRIO’s *analog input* ports or to one of the *roboRIO’s serial buses*.

2-Dimensional LIDAR



A 2-dimensional (2D) LIDAR sensor measures distance in all directions in a plane. Generally, this is accomplished (more-or-less) by simply placing a 1D LIDAR sensor on a turntable that spins at a constant rate.

Since, by nature, 2D LIDAR sensors need to send a large amount of data back to the roboRIO, they almost always connect to one of the roboRIO's *serial buses*.

42.10.2 Caveats

LIDAR sensors do suffer from a few common drawbacks:

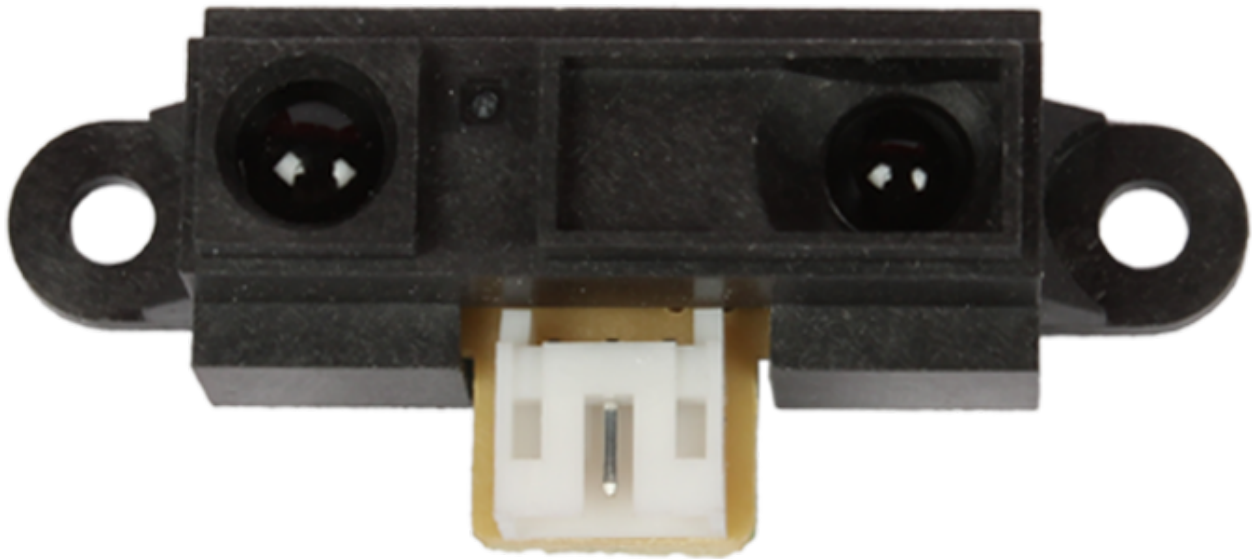
Like ultrasonics, LIDAR relies on the reflection of the emitted pulse back to the sensor. Thus, LIDAR critically depends on the reflectivity of the material in the wavelength of the laser. The FRC field wall is made of polycarbonate, which tends to be transparent in the infrared wavelength (which is what is generally legal for FRC use). Thus, LIDAR tends to struggle to detect the field barrier.

2D LIDAR sensors (at the price range legal for FRC use) tend to be quite noisy, and processing their measured data (known as a “point cloud”) can involve a lot of complex software. Additionally, there are very few 2D LIDAR sensors made specifically for FRC, so software support tends to be scarce.

As 2D LIDAR sensors rely on a turntable to work, their update rate is limited by the rate at which the turntable spins. For sensors in the price range legal for FRC, this often means that they do not update their values particularly quickly, which can be a limitation when the robot (or the targets) are moving.

Additionally, as 2D LIDAR sensors are limited in *angular* resolution, the *spatial* resolution of the point cloud is worse when targets are further away.

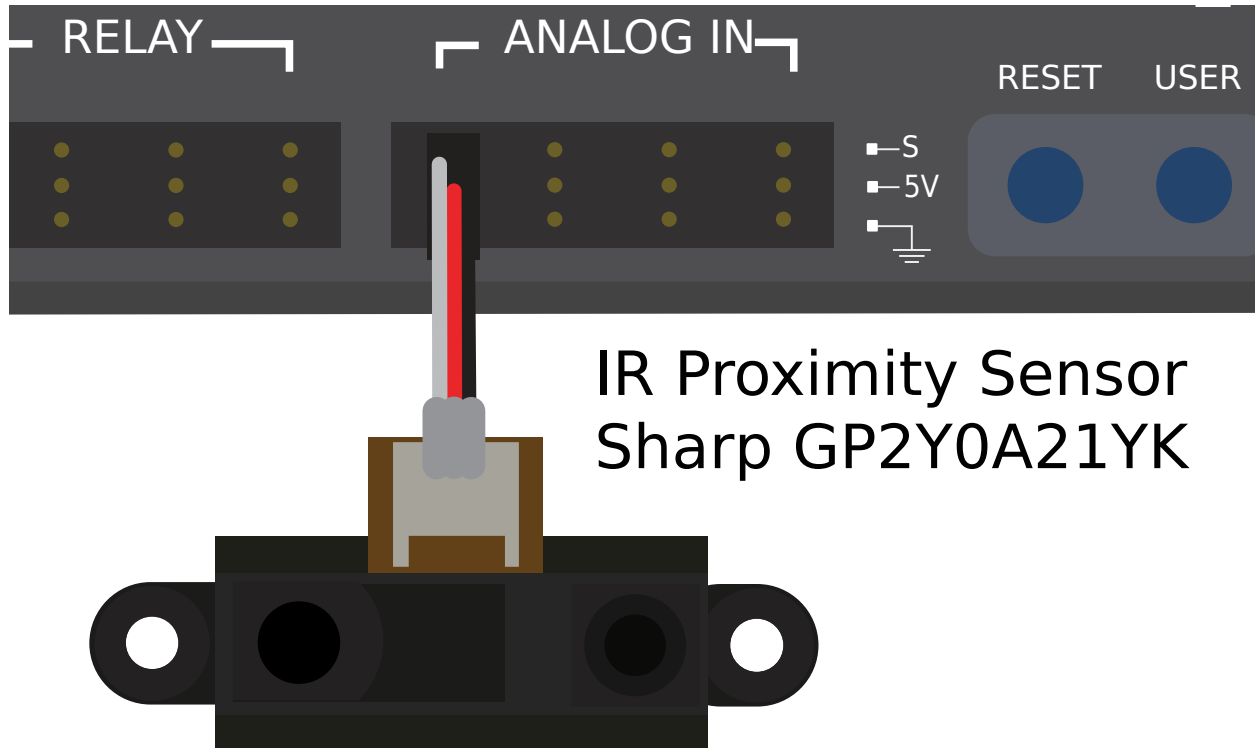
42.11 Triangulating Rangefinders



Triangulating rangefinders (often called “IR rangefinders,” as they commonly function in the infrared wavelength band) are another common type of rangefinder used in FRC®. The sensor shown above is a [common Sharp-brand sensor](#)

Unlike [LIDAR](#), triangulating rangefinders do not measure the time between the emission of a pulse and the receiving of a reflection. Rather, most IR rangefinders work by emitting a constant beam at a slight angle, and measuring the position of the reflected beam. The closer the point of contact of the reflected beam to the emitter, the closer the object to the sensor.

42.11.1 Using IR rangefinders



IR Rangefinders generally output an analog voltage proportional to the distance to the target, and thus connect to the *analog input* ports on the RIO.

42.11.2 Caveats

IR rangefinders suffer similar drawbacks to 1D LIDAR sensors - they are very sensitive to the reflectivity of the target in the wavelength of the emitted laser.

Additionally, while IR rangefinders tend to offer better resolution than LIDAR sensors when measuring at short distances, they are also usually more sensitive to differences in orientation of the target, *especially* if the target is highly-reflective (such as a mirror).

42.12 Serial Buses

In addition to the *digital* and *analog* inputs, the roboRIO also offers several methods of serial communication with peripheral devices.

Both the digital and analog inputs are highly limited in the amount of data that can be send over them. Serial buses allow users to make use of far more-robust and higher-bandwidth communications protocols with sensors that collect large amounts of data, such as inertial measurement units (IMUs) or 2D LIDAR sensors.

42.12.1 Types of supported serial buses

The roboRIO supports many basic types of serial communications:

- *I2C*
- *SPI*
- *RS-232*
- *USB Host*
- *CAN Bus*

Additionally, the roboRIO supports communications with peripheral devices over the CAN bus. However, as the FRC® CAN protocol is quite idiosyncratic, relatively few peripheral sensors support it (though it is heavily used for motor controllers).

42.12.2 I2C



I²C Port

Figure 6 and Table 5 describe the I²C port pins and signals.

Figure 6. I²C Port Pinout

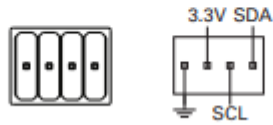


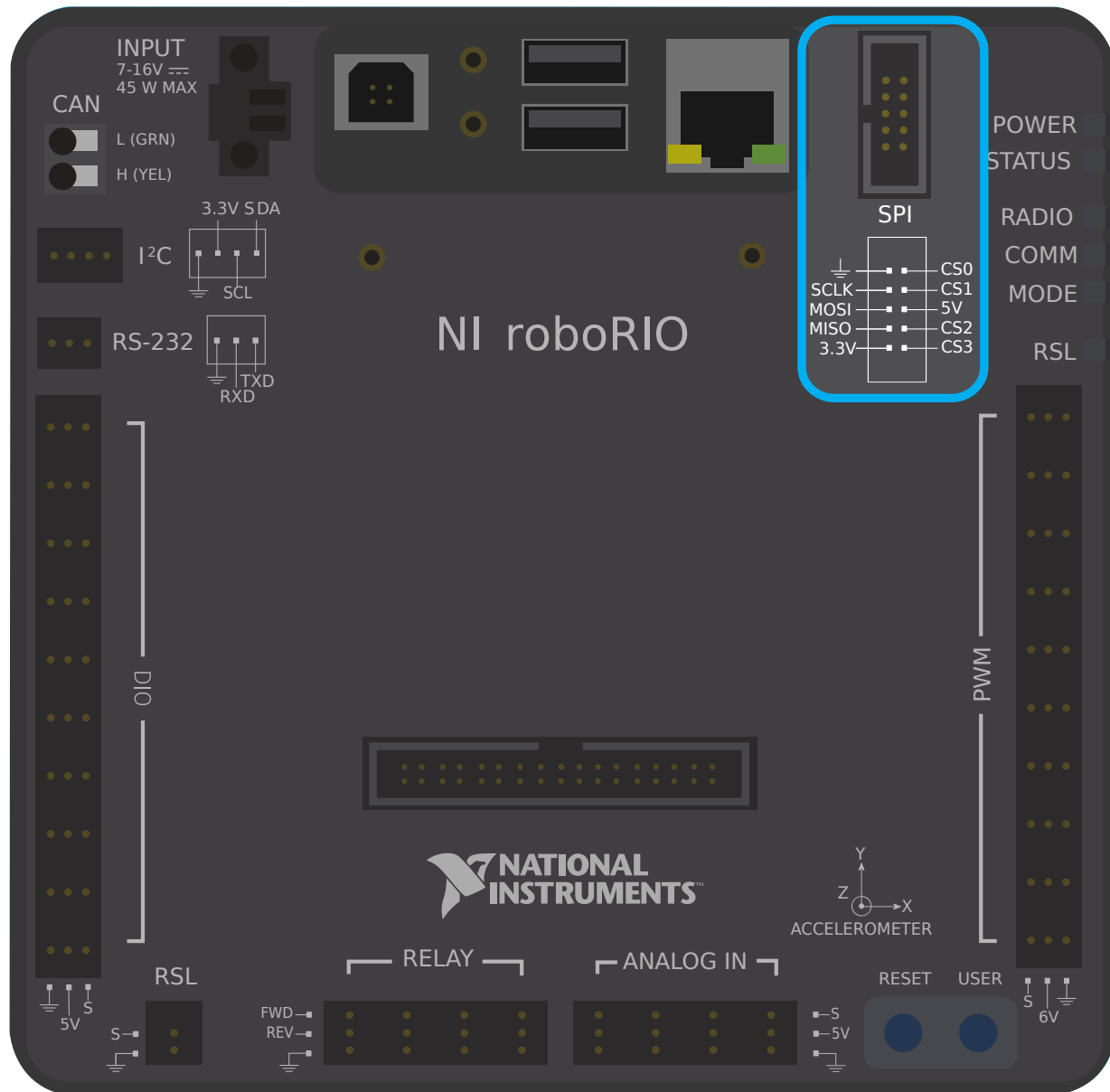
Table 5. I²C Port Signal Descriptions

Signal Name	Direction	Description
GND	—	Reference for digital lines and +3.3 V power output.
3.3V	Output	+3.3 V power output.
SCL	Input or Output	I ² C lines with 3.3 V output, 3.3 V/5 V-compatible input. Refer to the PC Lines section for more information.
SDA	Input or Output	

To communicate to peripheral devices over I²C, each pin should be wired to its corresponding pin on the device. I²C allows users to wire a “chain” of slave devices to a single port, so long as those devices have separate IDs set.

The I²C bus can also be used through the [MXP expansion port](#). The I²C bus on the MXP is independent. For example, a device on the main bus can have the same ID as a device on the MXP bus.

42.12.3 SPI



SPI Port

Figure 13 and Table 12 describe the SPI port pins and signals.

Figure 13. SPI Port Pinout

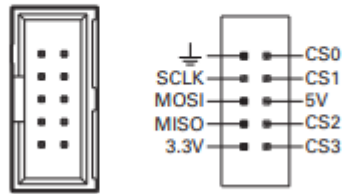


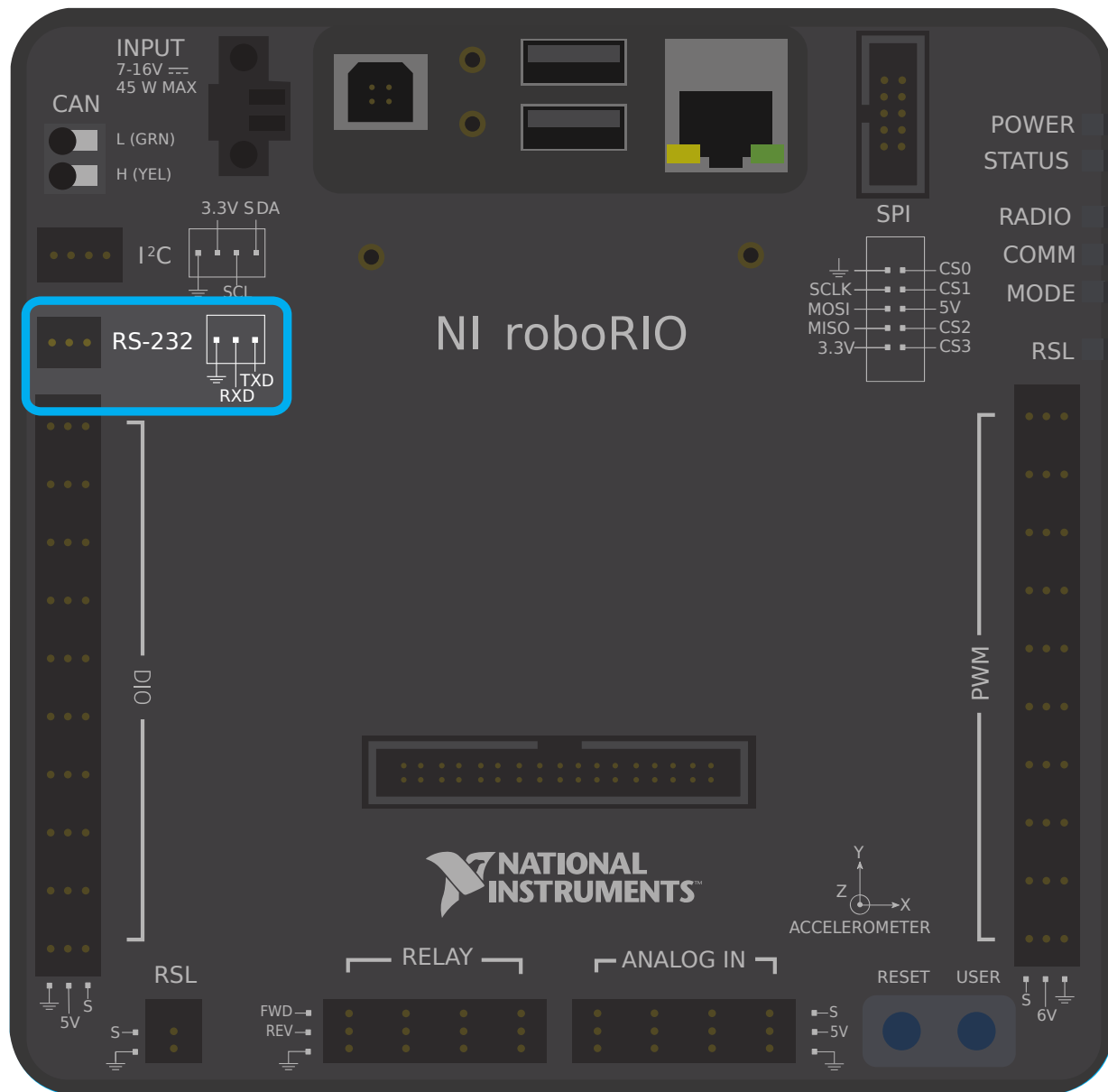
Table 12. SPI Port Signal Descriptions

Signal Name	Direction	Description
3.3V	Output	+3.3 V power output.
5V	Output	+5 V power output.
CS <0..3>	Output	SPI with 3.3 V output, 3.3 V/5 V-compatible input. Refer to the SPI Lines section for more information.
SCLK	Output	
MOSI	Output	
MISO	Input	
GND	—	Reference for digital lines and +3.3 V and +5.5 V power output.

To communicate to peripheral devices over SPI, each pin should be wired to its corresponding pin on the device. The SPI port supports communications to up to four devices (corresponding to the Chip Select (CS) 0-3 pins on the diagram above).

The SPI bus can also be used through the [MXP expansion port](#). The MXP port provides independent clock, and input/output lines and an additional CS.

42.12.4 RS-232



RS-232 Port

Figure 7 and Table 6 describe the RS-232 port pins and signals.

Figure 7. RS-232 Serial Port Pinout

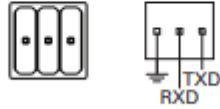


Table 6. RS-232 Serial Port Signal Descriptions

Signal Name	Direction	Description
TXD	Output	Serial transmit output with ± 5 V to ± 15 V signal levels. Refer to the UART and RS-232 Lines section for more information.
RXD	Input	Serial receive input with ± 15 V input voltage range. Refer to the UART and RS-232 Lines section for more information.
GND	—	Reference for digital lines.

To communicate to peripheral devices over RS-232, each pin should be wired to its corresponding pin on the device.

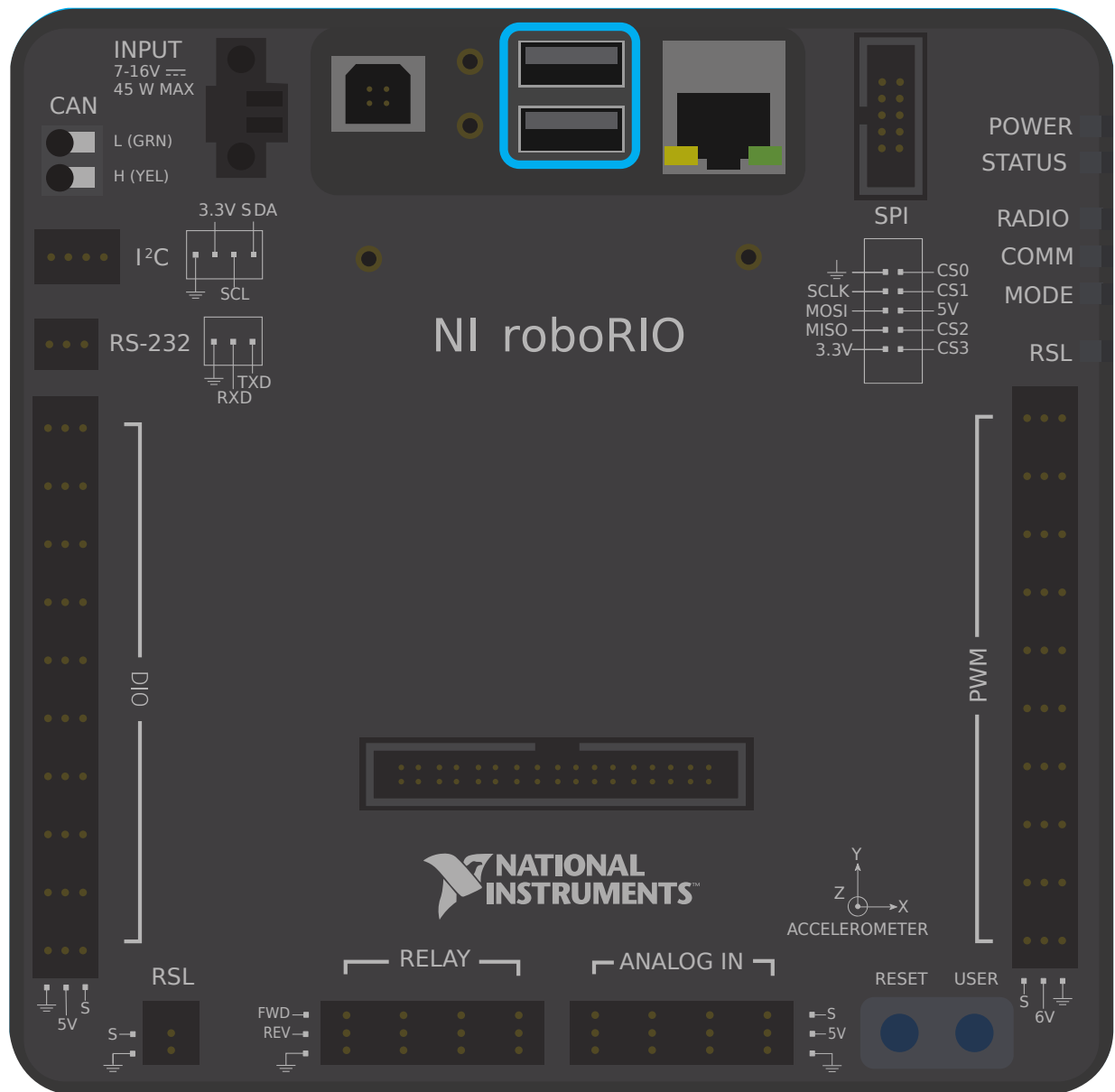
The RS-232 bus can also be used through the [MXP expansion port](#).

The RoboRIO RS-232 serial port uses RS-232 signaling levels (± 15 v). The MXP serial port uses CMOS signaling levels (± 3.3 v).

Note: By default, the onboard RS-232 port is utilized by the roboRIO's serial console. In order to use it for an external device, the serial console must be disabled using the [Imaging Tool](#) or [roboRIO Web Dashboard](#).

One of the USB ports on the roboRIO is a USB-B, or USB client port. This can be connected to devices, such as a Driver Station computer, with a standard USB cable.

42.12.5 USB Host



Two of the USB ports on the roboRIO is a USB-A, or USB host port. These can be connected to devices, such as cameras or sensors, with a standard USB cable.

42.12.6 MXP Expansion Port

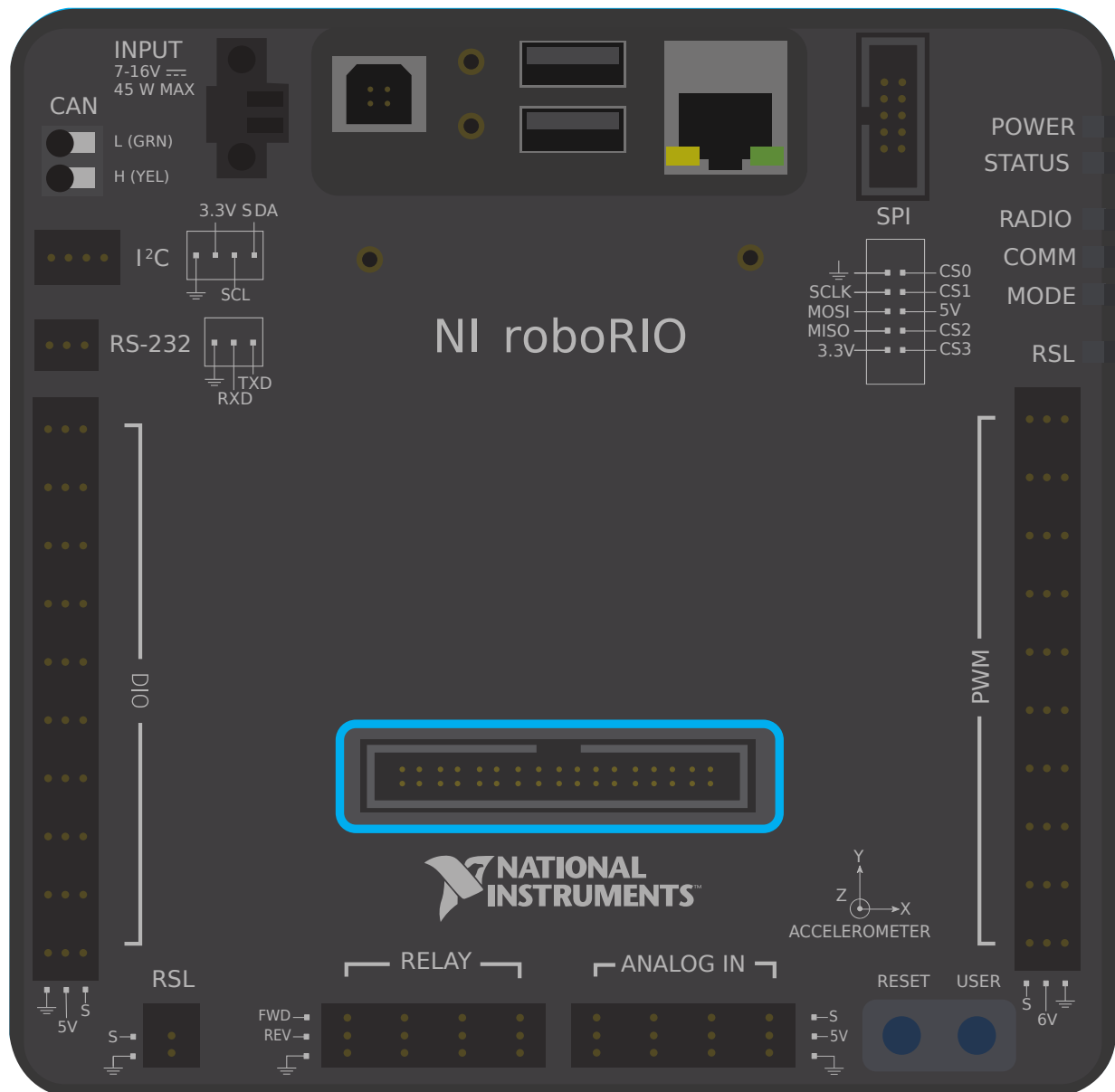
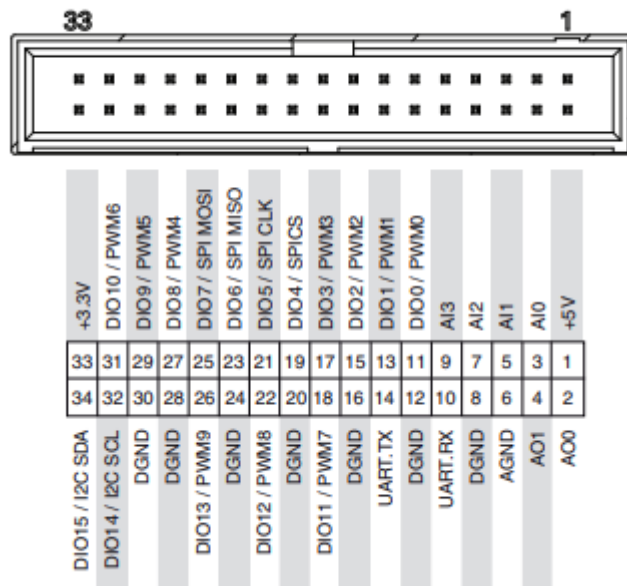


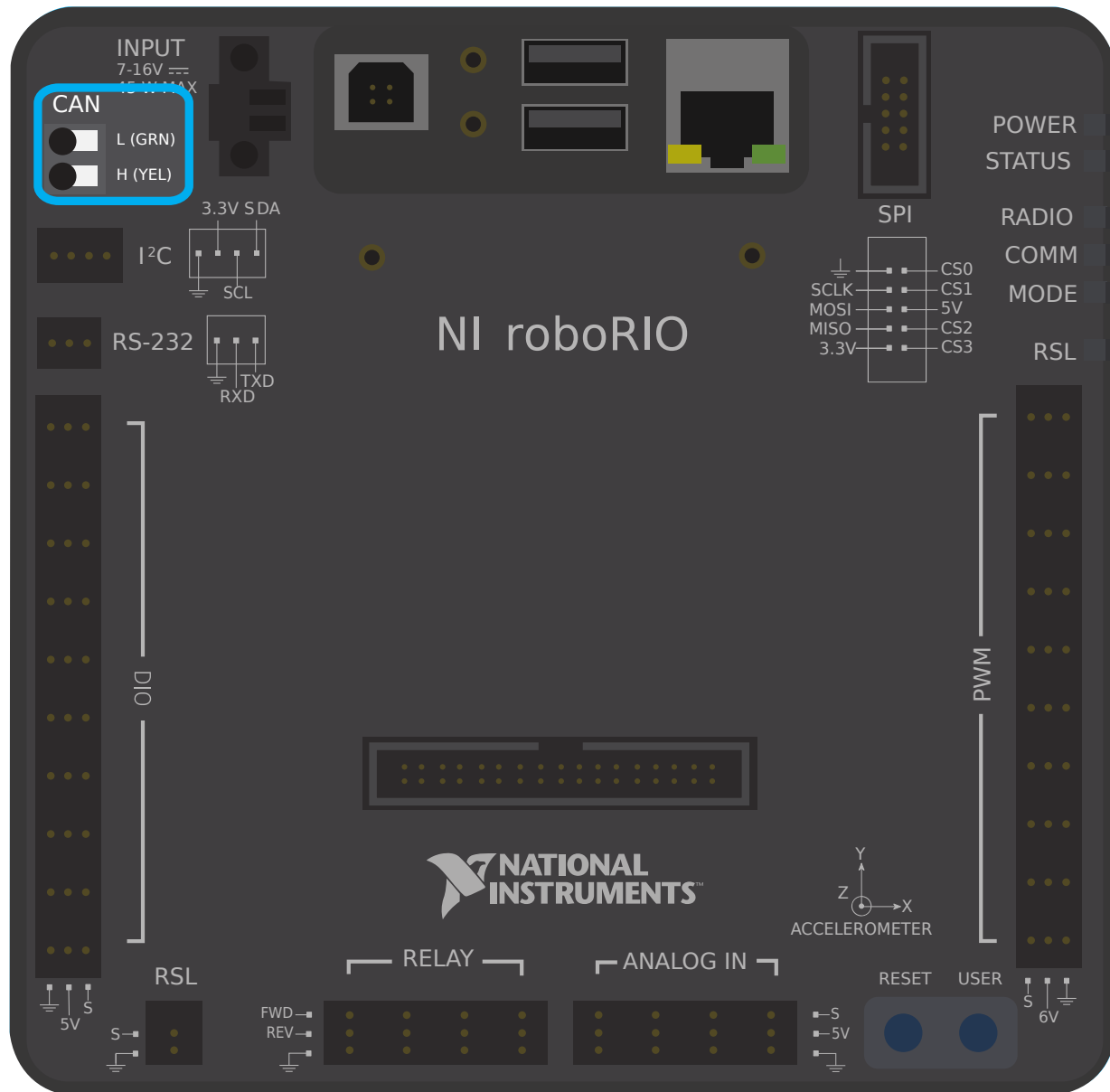
Figure 4. MXP Pinout



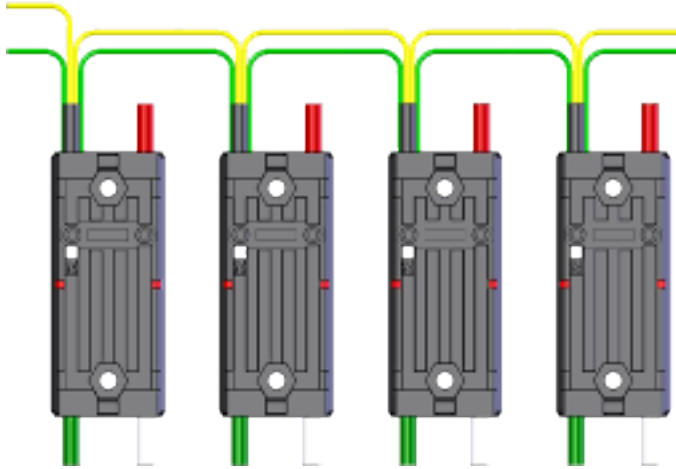
Several of the serial buses are also available for use through the roboRIO's MXP Expansion Port. This port allows users to make use of many additional *digital* and *analog* inputs, as well as the various serial buses.

Many peripheral devices attach directly to the MXP port for convenience, requiring no wiring on the part of the user.

42.12.7 CAN Bus



Additionally, the roboRIO supports communications with peripheral devices over the CAN bus. However, as the FRC CAN protocol is quite idiosyncratic, relatively few peripheral sensors support it (though it is heavily used for motor controllers). One of the advantages of using the CAN bus protocol is that devices can be daisy-chained, as shown below. If power is removed from any device in the chain, data signals will still be able to reach all devices in the chain.



Several sensors primarily use the CAN bus. Some examples include:

- [CAN Based Time-of-Flight Range/Distance Sensor](#) from [playingwithfusion.com](#)
- TalonSRX-based sensors, such as the [Gadgeteer Pigeon IMU](#) and the [SRX MAG Encoder](#)
- [CANifier](#)
- Power monitoring sensors built into the *Power Distribution Panel (PDP)*

More information about using devices connected to the CAN bus can be found in the article about [using can devices](#).

Getting Started with Romi



The Romi is a small and inexpensive robot designed for learning about programming FRC robots. All the same tools used for programming full-sized FRC robots can be used to program the Romi. The Romi comes with two drive motors with integrated wheel encoders. It also includes an IMU sensor that can be used for measuring headings and accelerations. Using it is as simple as writing a robot program, and running it on your computer. It will command the Romi to follow the steps in the program.

43.1 Romi Hardware & Assembly

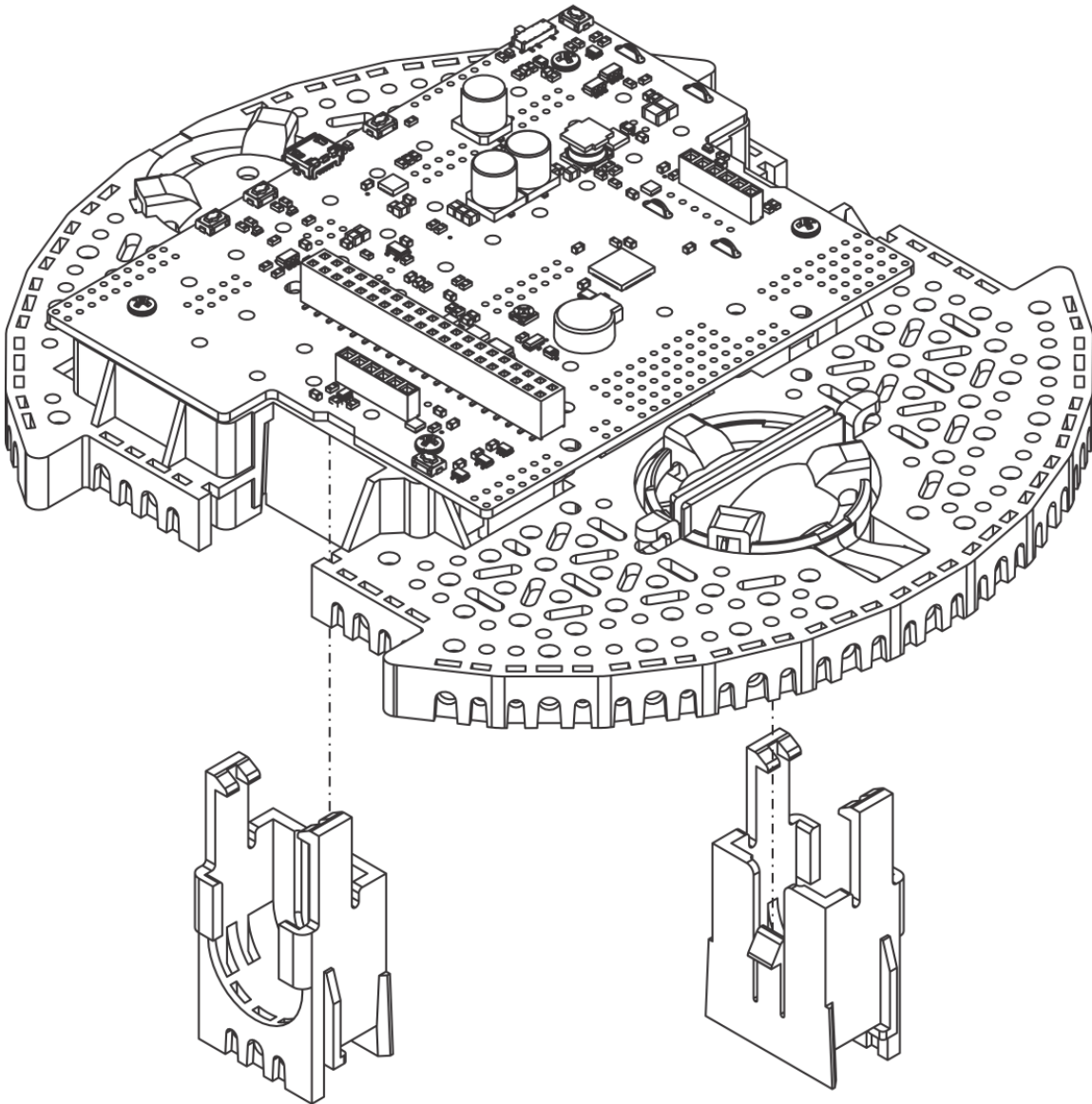
To get started with the Romi, you will need to have the necessary hardware.

1. [Romi Kit from Pololu](#) - Order qualifies for free shipping
2. [Raspberry Pi](#) - 3B+ or 4
3. [8GB \(or larger\) Micro SD card](#)
4. [Micro SD card reader](#) - if you don't already have one
5. [6 AA batteries](#) - Rechargeable is best (don't forget the charger)

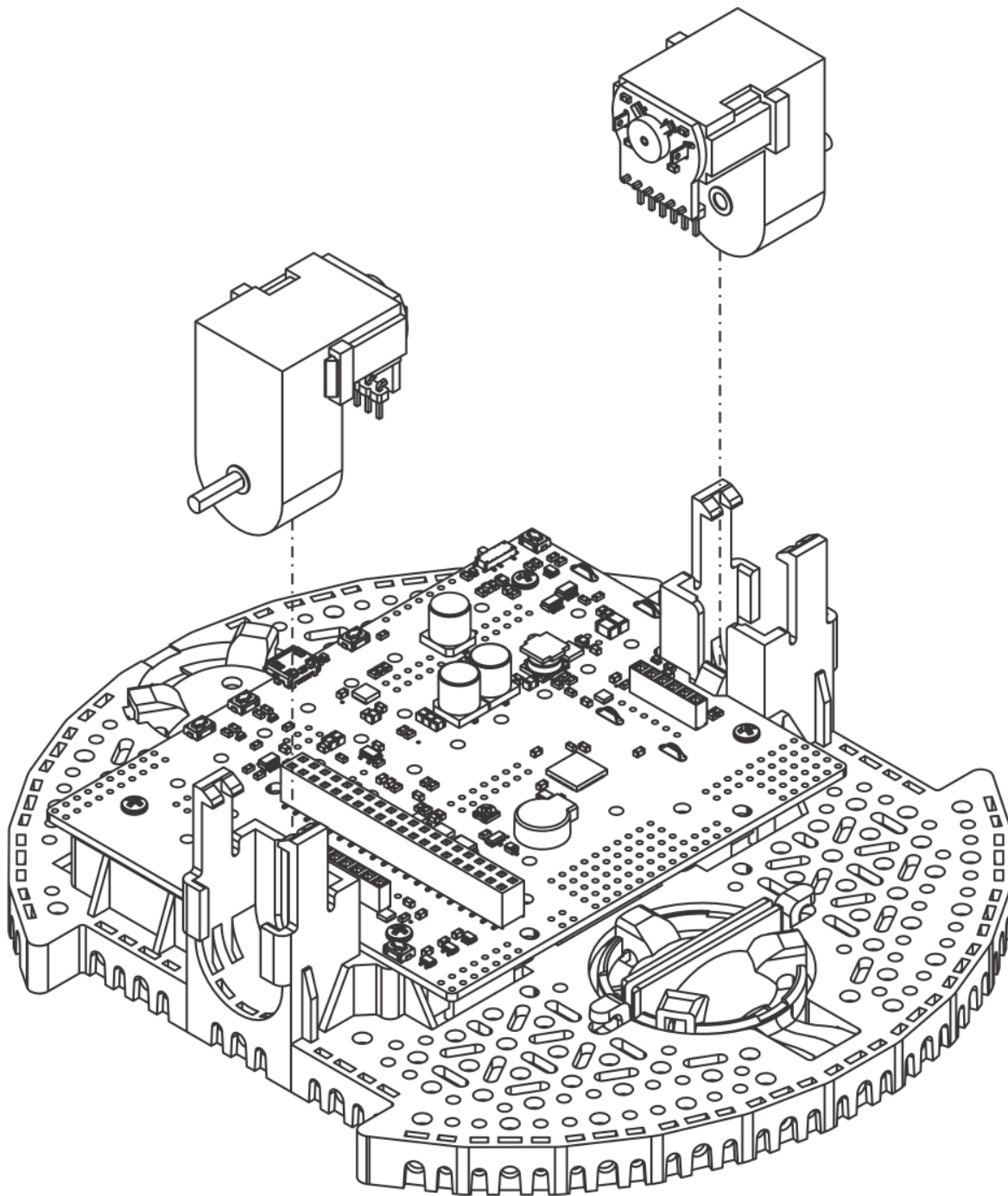
43.1.1 Assembly

The Romi Robot Kit for FIRST comes pre-soldered and only has to be put together before it can be used. Once you have gathered all the materials you can begin assembly:

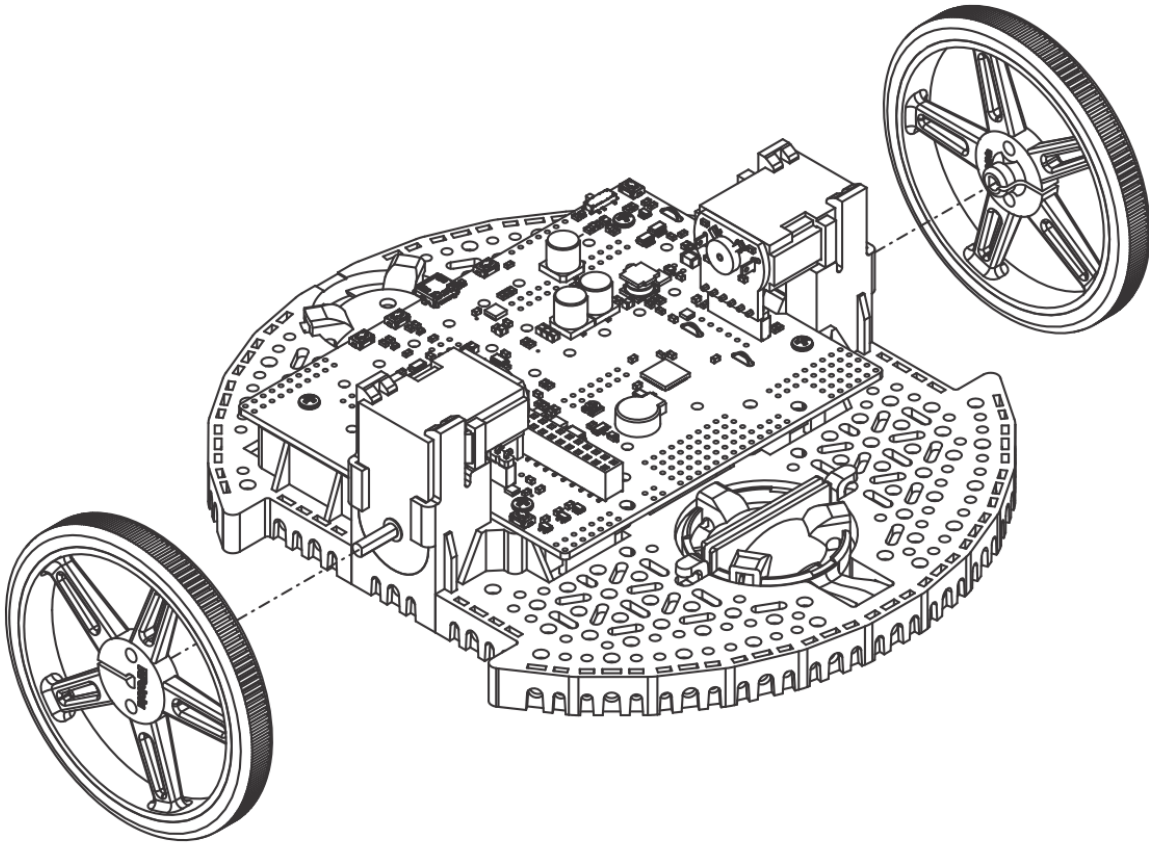
1. Align the motor clips with the chassis as indicated and press them firmly into the chassis until the bottom of the clips are even with the bottom of the chassis (you may hear several clicks).



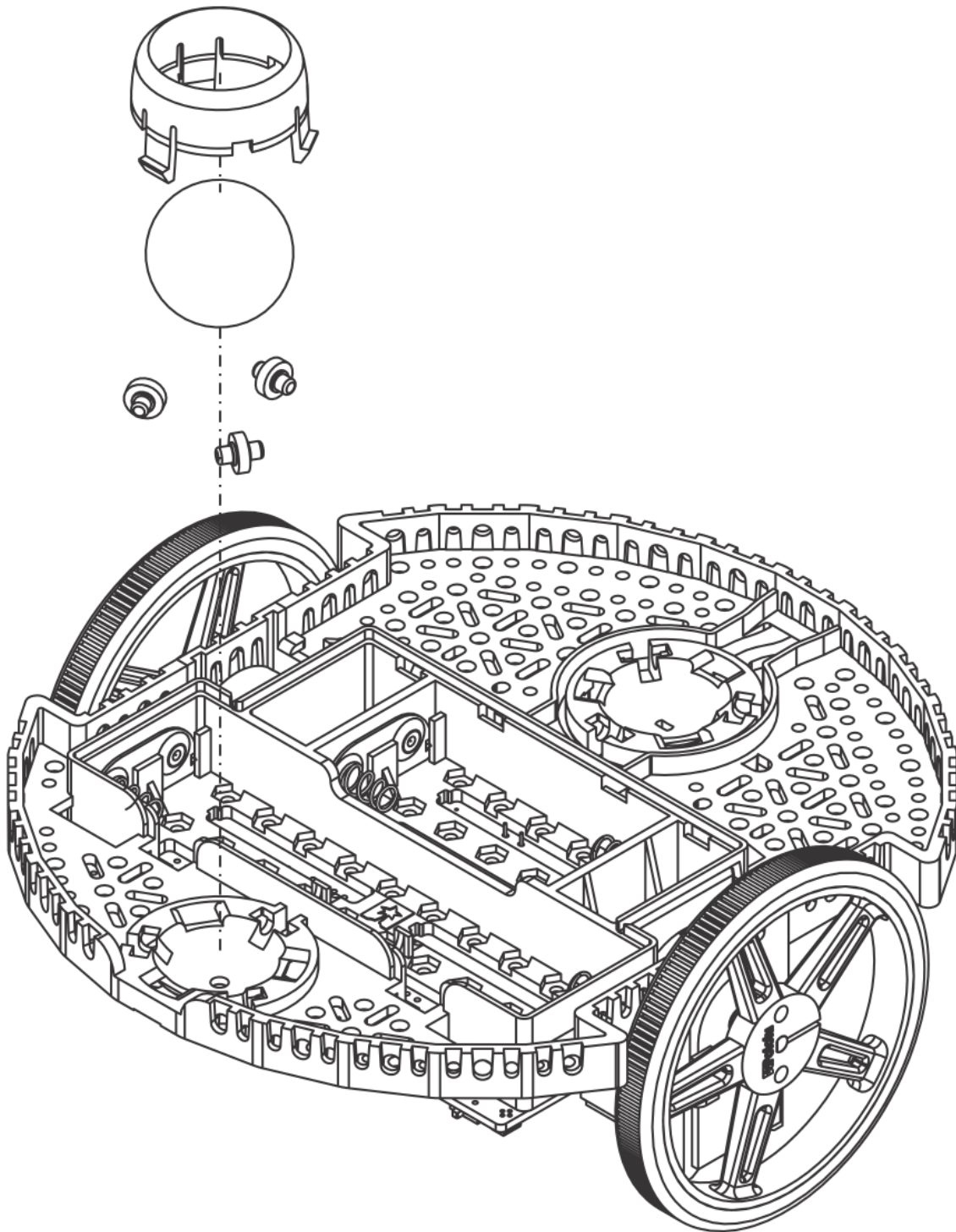
2. Push the Mini Plastic Gearmotors into the motor clips until they snap into place. Note that the motor blocks the clip release, so if you need to remove a motor bracket later, you will first need to remove the motor. The Mini Plastic Gearmotors that come with the kit have extended motor shafts to enable quadrature encoders for position feedback.



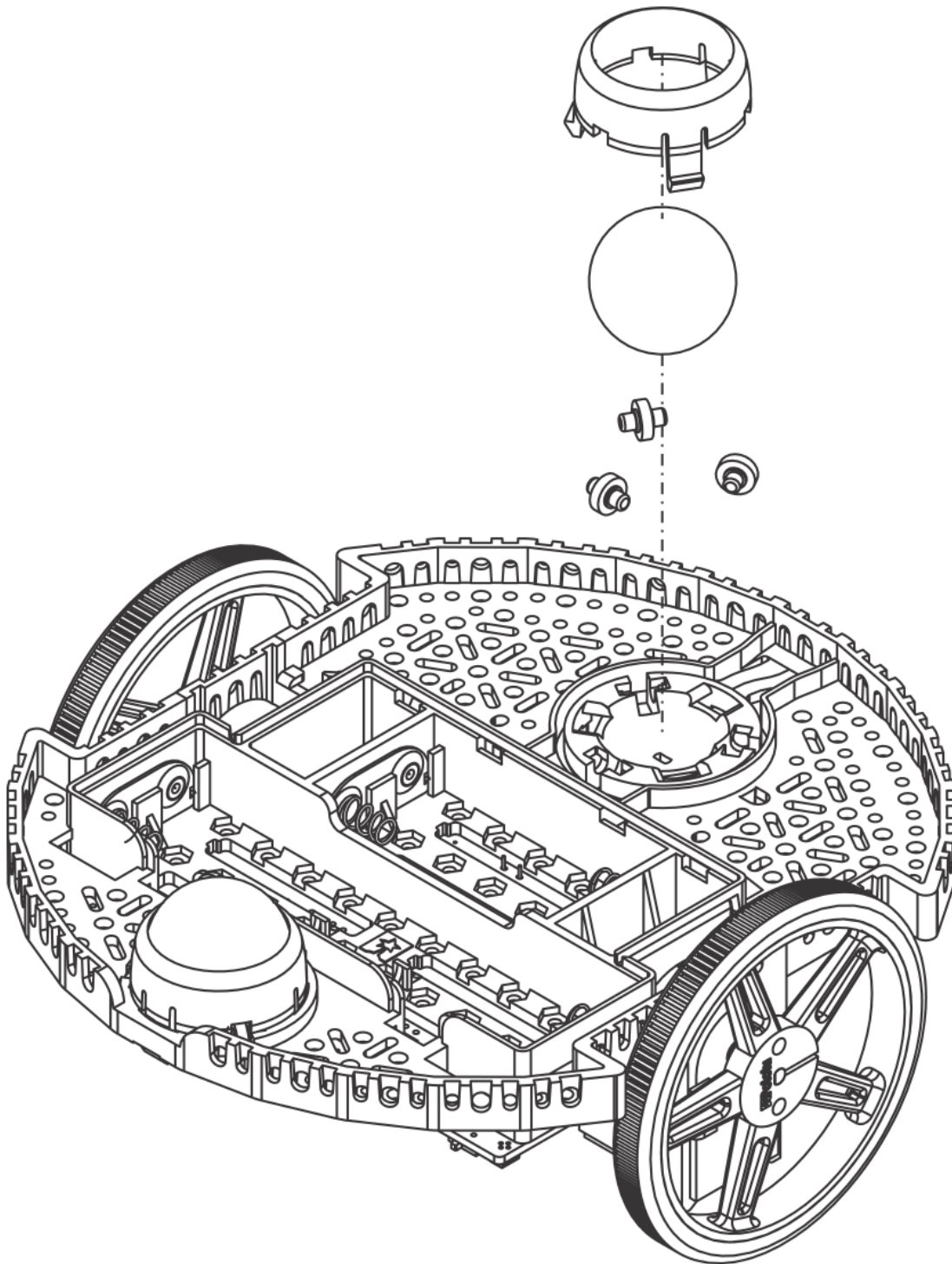
3. Press the wheels onto the output shafts of the motors until the motor shaft is flush with the outer face of the wheel. One way to do this is to set the wheel on a flat surface and line the chassis up with it so that the flat part of the motor's D-shaft lines up correctly with the wheel. Then, lower the chassis, pressing the motor shaft into the wheel until it contacts the surface.



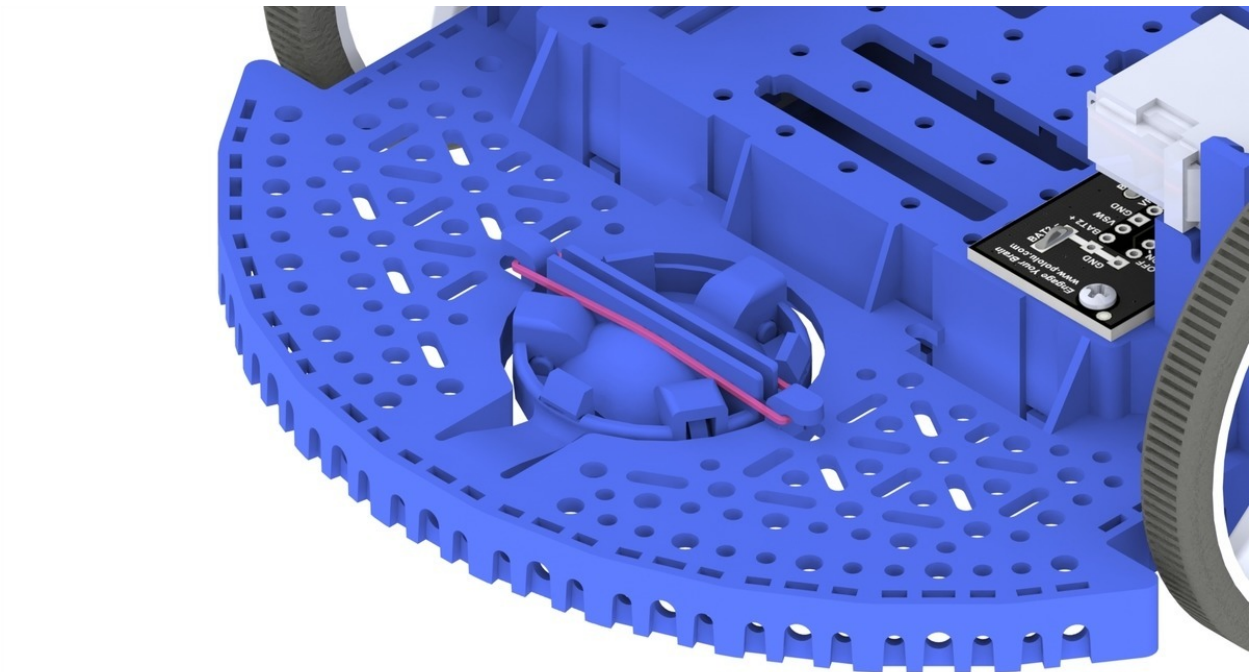
4. Flip the chassis upside down and place the three rollers for the rear ball caster into the cutouts in the chassis. Place the 1" plastic ball on top of the three rollers. Then push the ball caster retention clip over the ball and into the chassis so the three legs snap into their respective holes.



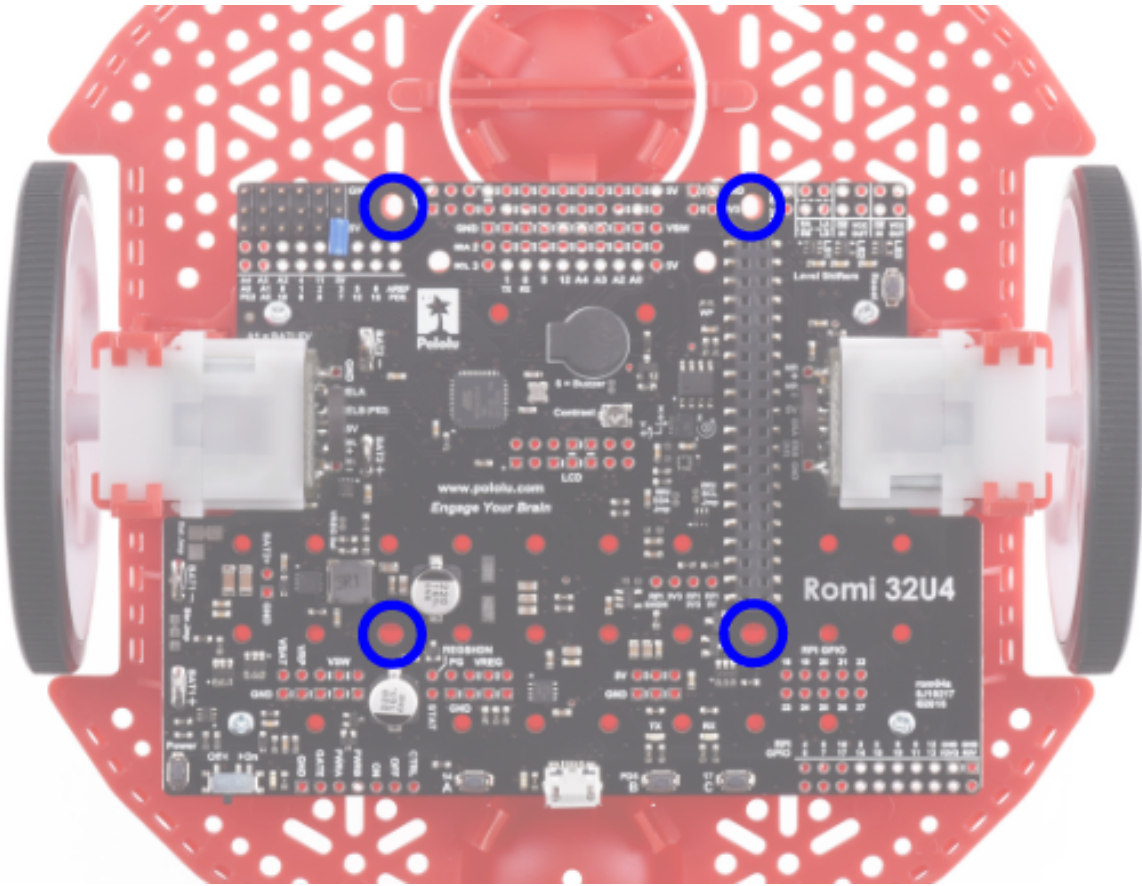
5. Repeat for the front ball caster so there is a caster on the front and the back of the robot.



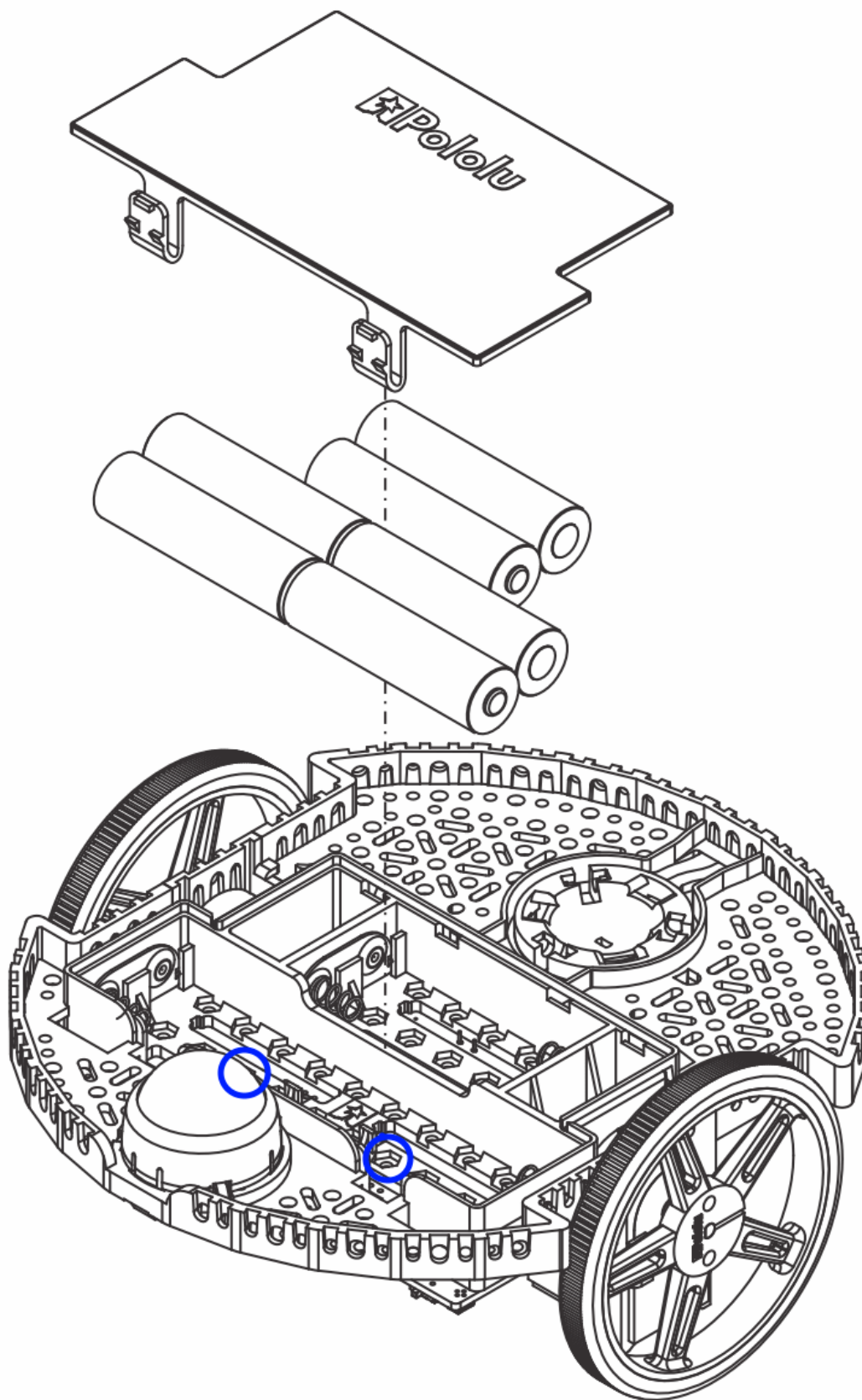
6. Optional: The front ball caster is supported by a flexible arm that acts as a suspension system. If you want to make it stiffer, you can wrap a rubber band around the two hooks located on either side of the ball caster on the top side of the chassis.



7. Install the standoffs to support the Raspberry Pi board. Two standoffs (thread side down) mount in the holes on the side of the Romi board closest to the “Romi 32U4” label as shown in the picture. The nuts for these standoffs are inside the battery compartment. The other two standoffs go into the holes on the opposite side of the board. To attach them, you will need a needle-nose pliers to hold the nut while you screw in the standoffs. The circled holes in the image below show where the standoffs should go.

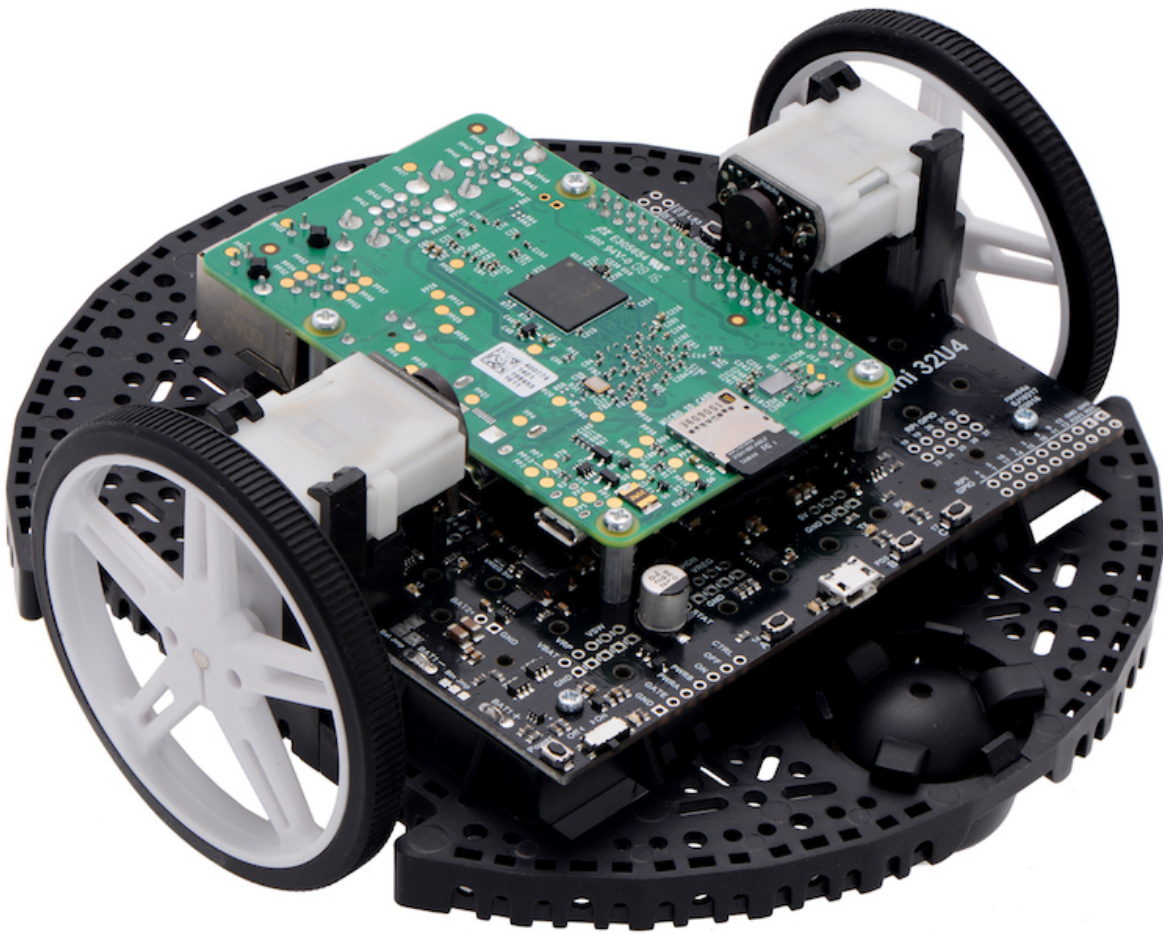


8. The chassis works with four or six AA batteries (we recommend using rechargeable AA NiMH cells). The correct orientation for the batteries is indicated by the battery-shaped holes in the Romi chassis as well as the + and - indicators in the chassis itself.



9. Attach the Raspberry Pi board upside down, carefully aligning the 2x20 pin connector on the Pi with the 2x20 pin socket on the Romi. Push with even pressure taking care to not bend any of the pins. Once inserted, use the supplied screws to fasten the Raspberry Pi board to the standoffs that were installed in a previous step.

Note: Two of the screws will require placing a nut in a hexagonal hole inside the battery compartment. The locations are shown by the blue circles in the image above.



The assembly of your Romi chassis is now complete!

43.2 Imaging your Romi

The Romi has 2 microprocessor boards:

1. A **Raspberry Pi** that handles high level communication with the robot program running on the desktop and
2. A **Romi 32U4 Control Board** that handles low level motor and sensor operation.

Both boards need to have firmware installed so that the robot operate properly.

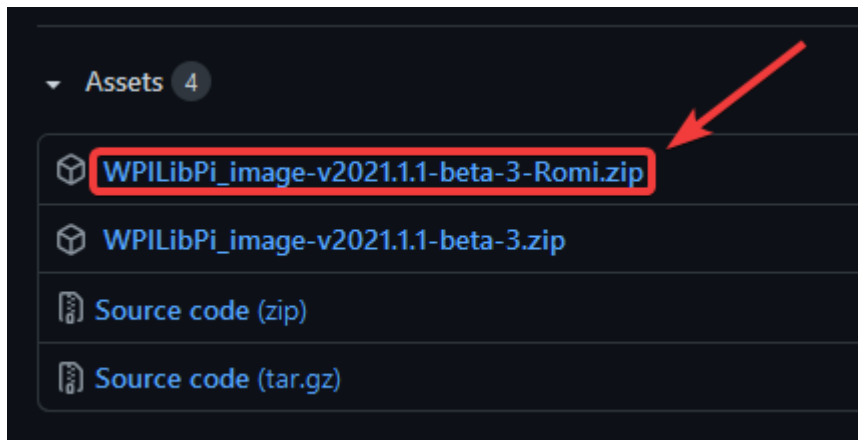
43.2.1 Raspberry Pi

Download

The Raspberry Pi firmware is based on WPILibPi (formerly FRCVision) and must be downloaded and written to the Raspberry Pi micro SD card. Click on Assets at the bottom of the description to see the available image files:

Romi WPILibPi

Be sure to download the Romi version and not the standard release of WPILibPi. The Romi version is suffixed with -Romi. See the below image for an example.



Imaging

The procedure for installing the image is described here: [WPILibPi Installation](#).

Wireless Network Setup

Perform the following steps to get your Raspberry Pi ready to use with the Romi:

1. Turn the Romi on by sliding the power switch on the Romi 32U4 board to the on position. The first time it is started with a new image it will take approximately 2-3 minutes to boot while it resizes the file system and reboots. Subsequent times it will boot in less than a minute.
2. Using your computer, connect to the Romi WiFi network using the SSID WPILibPi-
<number> (where <number> is based on the Raspberry Pi serial number) with the WPA2
passphrase WPILib2021!.

Note: If powering on the Raspberry Pi in an environment with multiple WPILibPi-running Raspberry Pis, the SSID for a particular Raspberry Pi is also announced audibly through the headphone port. The default SSID is also written to the /boot/default-ssid.txt file, which can be read by inserting the SD card (via a reader) into a computer and opening the boot partition.

3. Open a web browser and connect to the Raspberry Pi dashboard at either <http://10.0.0.2/> or <http://wpilibpi.local/>.

Note: The image boots up read-only by default, so it is necessary to click the Writable button to make changes. Once done making changes, click the Read-Only button to prevent memory corruption.

4. Select *Writable* at the top of the dashboard web page.
5. Change the default password for your Romi by setting a new password in the WPA2
Passphrase field.
6. Press the *Save* button at the bottom of the page to save changes.
7. Change the network SSID to a unique name if you plan on operating your Romi on wire-
less network with other Romis.
8. Reconnect to the Romi's WiFi network with the new password you set.

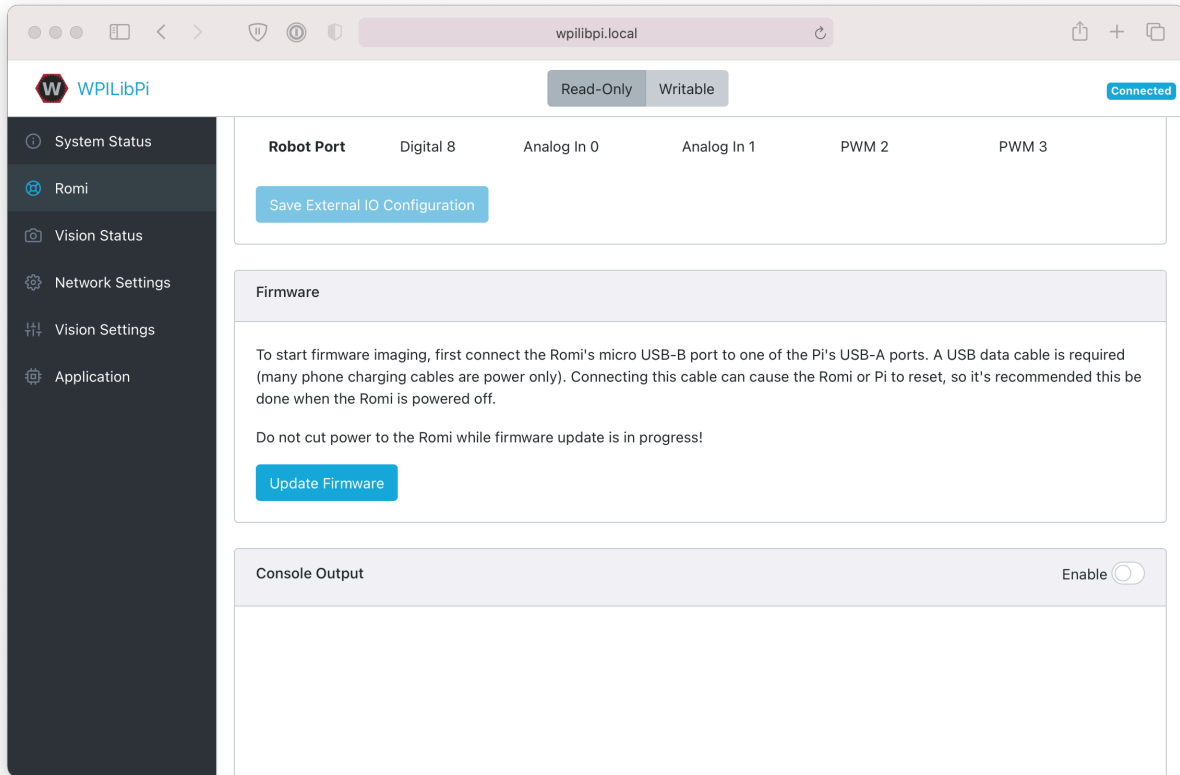
Be sure to set the Dashboard to Read-only when all the changes have been completed.

The screenshot shows a web browser window with the address bar displaying `wpiibpi.local`. The page title is "WPILibPi". There are two tabs at the top: "Read-Only" and "Writable", with "Writable" being the active tab. A "Connected" status indicator is in the top right corner. On the left is a dark sidebar with a menu containing: "System Status", "Romi", "Vision Status", "Network Settings" (highlighted with a gear icon), "Vision Settings", and "Application". The main content area displays network configuration options with labels and input fields or dropdowns: "Ethernet Address" (set to "DHCP"), "WiFi Mode" (set to "Access Point"), "WiFi Channel" (set to "7"), "SSID" (set to "WPILibPi"), "WPA2 Passphrase" (set to "WPILib2021!"), "WiFi Address" (set to "Static"), "IPv4 Address" (set to "10.0.0.2"), "Subnet Mask" (set to "255.255.255.0"), "Gateway" (set to "0.0.0.0"), and "DNS Server" (empty). A blue "Save" button is at the bottom left of the form. The footer contains "WPILibPi © 2020 FIRST" on the left and "v2021.1.1-beta-3-3-gd9fd278" on the right.

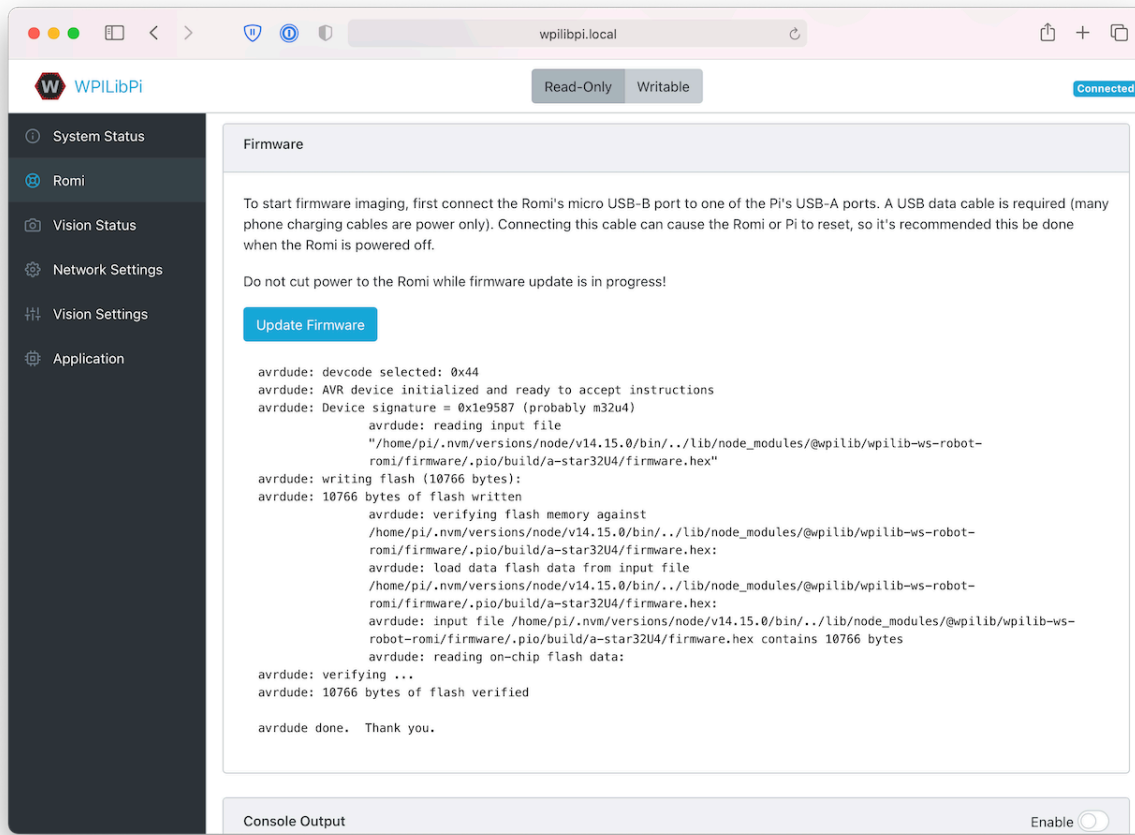
43.2.2 32U4 Control Board

The Raspberry Pi can now be used to write the firmware image to the 32U4 Control Board.

1. Turn off the Romi
2. Connect a USB A to micro-B cable from one of the Raspberry Pi's USB ports to the micro USB port on the 32U4 Control Board.
3. Turn on the Romi and connect to its Wifi network and connect to the web dashboard as in the previous steps.
4. On the Romi configuration page, press the *Update Firmware* button.



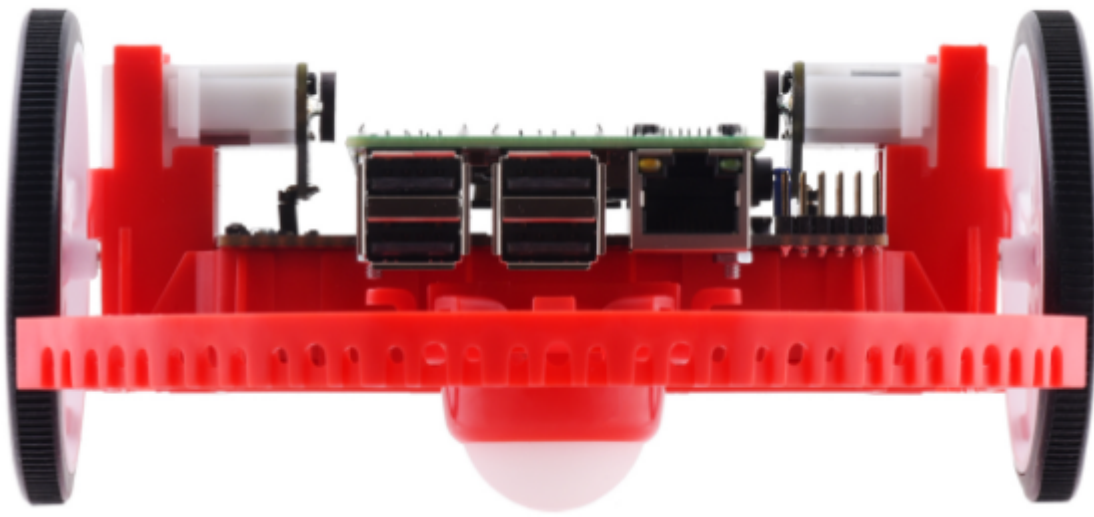
A console will appear showing a log of the firmware deploy process. Once the firmware has been deployed to the 32U4 Control Board, the message `avrdude done. Thank you.` will appear.



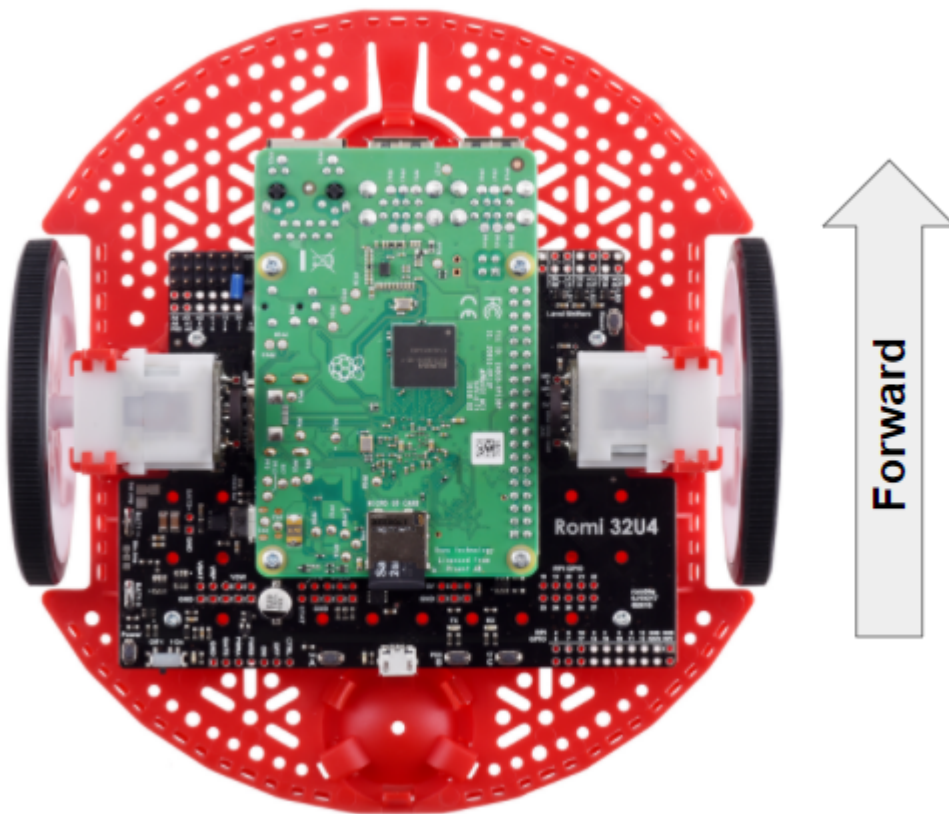
43.3 Getting to know your Romi

43.3.1 Directional Conventions

The front of the Romi is where the Raspberry Pi USB ports, GPIO pins and suspended caster wheel are.



In all Romi documentation, references to driving forward use the above definition of “front”.



43.3.2 Hardware, Sensors and GPIO

The Romi has the following built-in hardware/peripherals:

- 2x geared motors with encoders
- 1x Inertial Measurement Unit (IMU)
- 3x LEDs (green, yellow, red)
- 3x pushbuttons (marked A, B, and C)
- 5x configurable GPIO channels

Motors, Wheels and Encoders

The motors used on the Romi have a 120:1 gear reduction, and a no-load output speed of 150 RPM at 4.5V. The free current is 0.13 amps and the stall current is 1.25 amps. Stall torque is 25 oz-in (0.1765 N-m) but the built-in safety clutch might start slipping at lower torques.

The wheels have a diameter of 70mm (2.75"). They have a trackwidth of 141mm (5.55").

The encoders are connected directly to the motor output shaft and have 12 Counts Per Revolution (CPR). With the provided gear ratio, this nets 1440 counts per wheel revolution.

Note: By default, the encoders count up when the Romi moves forward.

Inertial Measurement Unit

The Romi includes an STMicroelectronics LSM6DS33 Inertial Measurement Unit (IMU) which contains a 3-axis gyro and a 3-axis accelerometer.

The accelerometer has selectable sensitivity of 2G, 4G, 8G, and 16G. The gyro has selectable sensitivity of 125 Degrees Per Second (DPS), 250 DPS, 500 DPS, 1000 DPS and 2000 DPS.

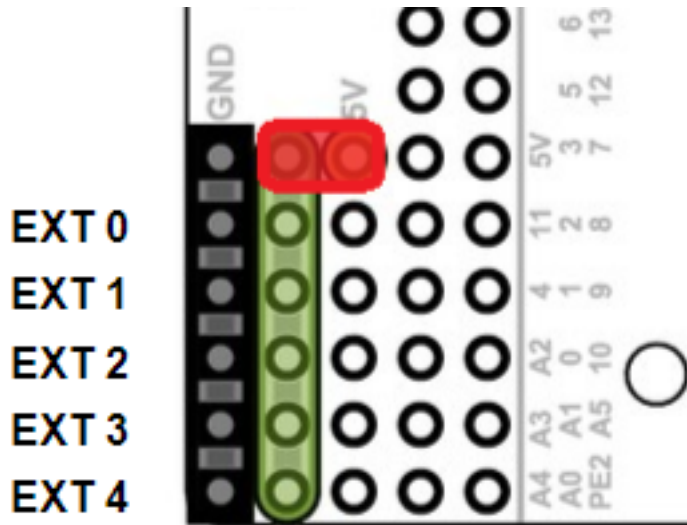
The Romi Web UI also provides a means to calibrate the gyro and measure its zero-offsets before use with robot code.

LEDs and Push Buttons

The Romi 32U4 control board has 3 push buttons and 3 LEDs that are exposed as Digital IO channels to robot code. The section on GPIO Mapping has more information on how to use the built in LEDs and buttons in robot code.

Configurable GPIO Channels

The control board has 5 configurable GPIO channels (named EXT0 through EXT4) that allow a user to connect external sensors and actuators to the Romi.



All 5 channels support the following modes: Digital IO, Analog In, PWM (with the exception of EXT 0, which only supports Digital IO and PWM).

The GPIO channels are exposed via a 3-pin, servo style interface, with connections for Ground, Power and Signal (with the Ground connection being closest to the edge of the board, and the signal being closest to the inside of the board).

The power connections for the GPIO channels are initially left unconnected, but can be hooked into the Romi's on-board 5V supply by using a jumper to connect the 5V pin to the power bus (as seen in the image above). Additionally, if more power than the Romi can provide is needed, the user can provide their own 5V power supply and connect it directly to power bus and ground pins.

43.3.3 GPIO Mapping

To support all the built in peripherals on the Romi, we have provided a hard-coded mapping of WPILib channels/devices to the Romi hardware.

Digital IO

DIO Channel	Romi Hardware Component
DIO 0	Button A (input only)
DIO 1	Button B (input), Green LED (output)
DIO 2	Button C (input), Red LED (output)
DIO 3	Yellow LED (output only)
DIO 4	Left Encoder Quadrature Channel A
DIO 5	Left Encoder Quadrature Channel B
DIO 6	Right Encoder Quadrature Channel A
DIO 7	Right Encoder Quadrature Channel B

PWM Output

PWM Channel	Romi Hardware Component
PWM 0	Left Motor
PWM 1	Right Motor

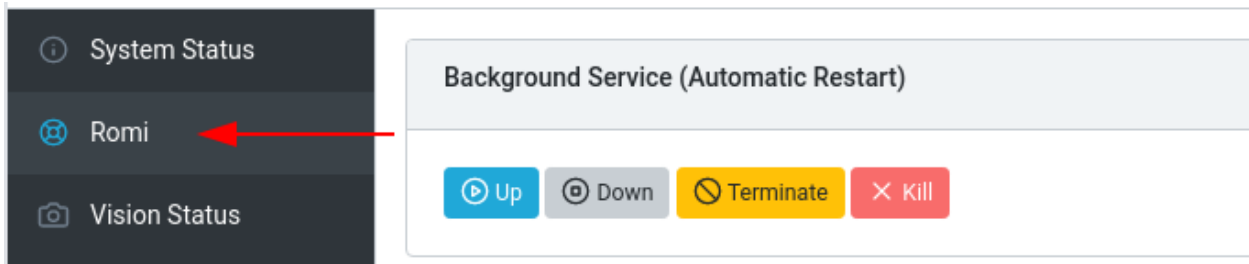
Note: The right motor is hardwired to spin in a forward direction when forward stick movement is applied on a joystick. Thus there is no need to invert the corresponding motor controller in robot code.

GPIO Channels

The Romi Web UI allows the user to customize the functions of the 5 configurable GPIO channels. The UI will also provide the appropriate WPILib channel/device mappings on screen once the IO configuration is complete. More information on this can be found in the next section.

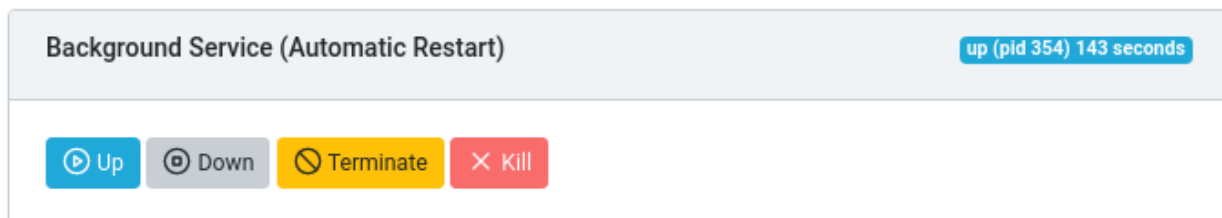
43.4 The Romi Web UI

The Romi Web UI comes installed as part of the WPILibPi Raspberry Pi image. It is accessible by clicking on the Romi tab in the navigation bar of the main WPILibPi Web UI.



The rest of this section will walk through the various parts of the Romi Web UI and describe the relevant functionality.

43.4.1 Background Service Status



This section of the Romi Web UI provides information about the currently running Romi Web Service (which is what allows WPILib to talk to the Romi). The UI provides controls to bring the service up/down as well as shows the current uptime of the web service.

Note: Users will not need to use the functionality in this section often, but it can be useful for troubleshooting.

43.4.2 Romi Status

Romi Status	
	Value
Romi Service Version	0.0.12
Firmware Compatible	Yes
Battery Voltage	7.65

This section provides information about the Romi, including the service version, battery voltage, and whether or not the currently installed firmware on the Romi 32U4 board is compatible with the current version of the web service.

Note: If the firmware is not compatible, see the section on [Imaging your Romi](#)

43.4.3 Web Service Update


Web Service Update

To perform an offline update of the Romi webservice, obtain an appropriate version from the [GitHub release page](#), and upload the .tgz file here.

Upload Romi Webservice Package

Choose File

No file chosen

 Save

Note: The Raspberry Pi must be in **Writable** mode for this section to work.

The Romi WPILibPi image ships with the latest (at publication time) version of the Romi web service. To support upgrading to newer versions of the Romi web service, this section allows users to upload a pre-built bundle that can be obtained via the Romi web service [GitHub releases page](#).

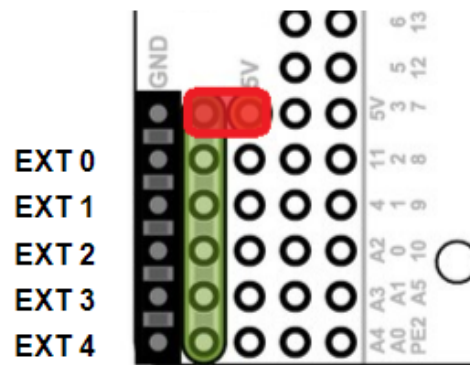
To perform an upgrade, download the appropriate .tgz file from the GitHub Releases page. Next, select the downloaded .tgz file and click *Save*. The updated web service bundle will be uploaded to the Raspberry Pi, and be installed. After a short moment, the Romi Status section should update itself with the latest version information.

43.4.4 External IO Configuration

External IO Configuration

Each of the 5 external pins can be configured to perform one of three functions: DIO, Analog In or PWM (EXT 0 can only be set to DIO or PWM).

After saving the IO configuration, the *Robot Port* section will update with the appropriate channels to use in robot code.



Romi Pin	EXT 0	EXT 1	EXT 2	EXT 3	EXT 4
Setting	<input type="text" value="DIO"/>	<input type="text" value="Analog"/>	<input type="text" value="Analog"/>	<input type="text" value="PWM"/>	<input type="text" value="PWM"/>
Robot Port	Digital 8	Analog In 0	Analog In 1	PWM 2	PWM 3

[Save External IO Configuration](#)

This section allows users to configure the 5 external GPIO channels on the Romi.

Note: The Raspberry Pi must be in **Writable** mode for this section to work.

To change the configuration of a GPIO channel, select an appropriate option from the drop-down lists. All channels (with the exception of EXT 0) support Digital IO, Analog In and PWM as channel types. Once the appropriate selections are made, click on *Save External IO Configuration*. The web service will then restart and pick up the new IO configuration.

The “Robot Port” row provides the appropriate WPILib mapping for each configured GPIO channel. For example, EXT 0 is configured as a Digital IO channel, and will be accessible in WPILib as a DigitalInput (or DigitalOutput) channel 8.

43.4.5 IMU Calibration

IMU Calibration

Most gyros will have some sort of zero offset. In order to get more accurate rate-of-turn readings, the gyro can be calibrated to calculate an appropriate zero offset.

To calibrate the gyro, place the Romi on a flat surface and click the "Calibrate Gyro" button. While the calibration is running, please do not touch the Romi.

Current Gyro Offsets

X Offset	Y Offset	Z Offset
0.683	-4.305	-2.817

 Calibrate Gyro

Note: The Raspberry Pi must be in **Writable** mode for this section to work.

This section allows users to calibrate the gyro on the Romi. Gyros usually have some sort of zero-offset error, and calibration allows the Romi to calculate the offset and use it in calculations.

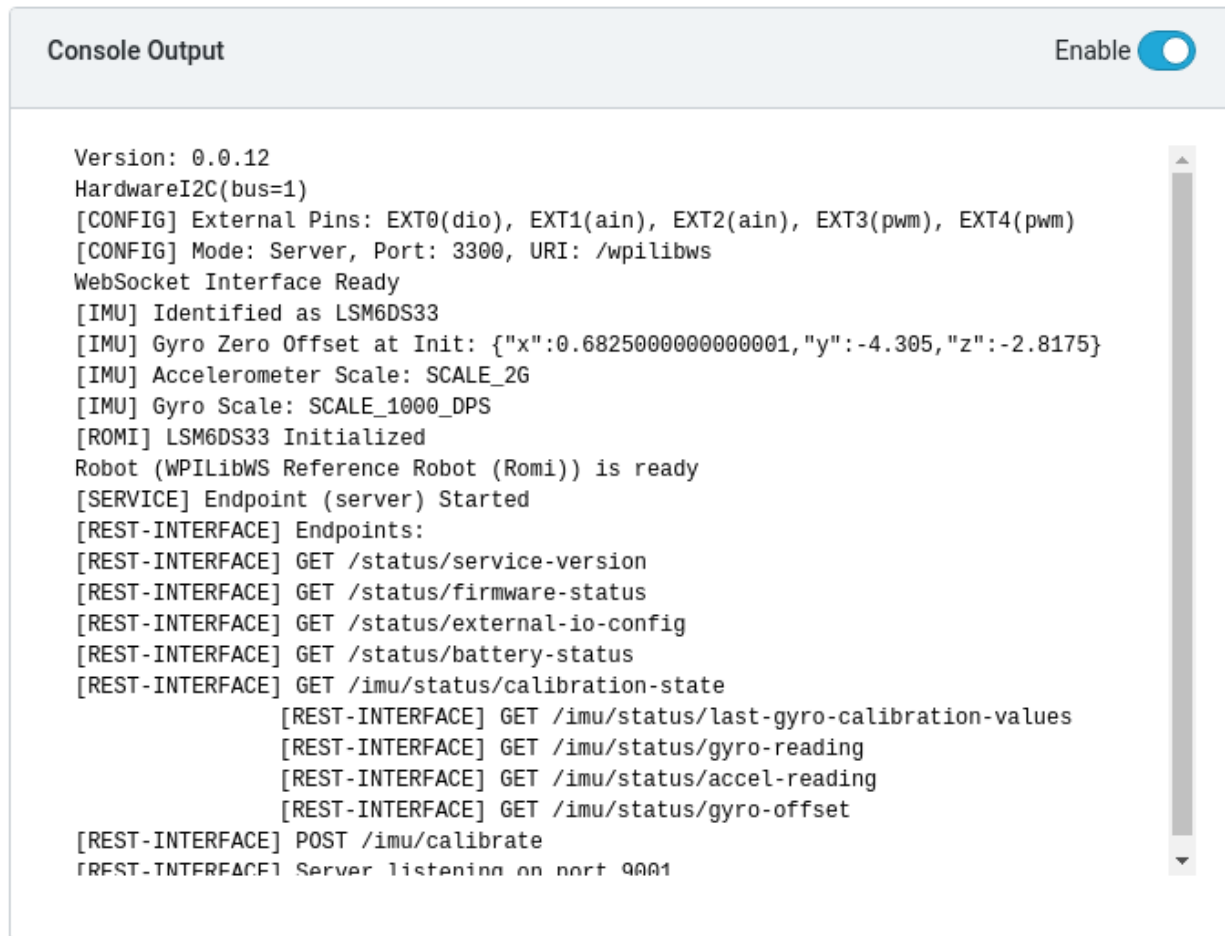
To begin calibration, place the Romi on a flat, stable surface. Then, click the *Calibrate Gyro* button. A progress bar will appear, showing the current calibration process. Once calibration is complete, the latest offset values will be displayed on screen and registered with the Romi web service.

These offset values are saved to disk and persist between reboots.

43.4.6 Firmware

Note: See the section on *Imaging your Romi*

43.4.7 Console Output



When enabled, this section allows users to view the raw console output that the Romi web service provides. This is useful for troubleshooting issues with the Romi, or just to find out more about what goes on behind the scenes.

43.4.8 Bridge Mode

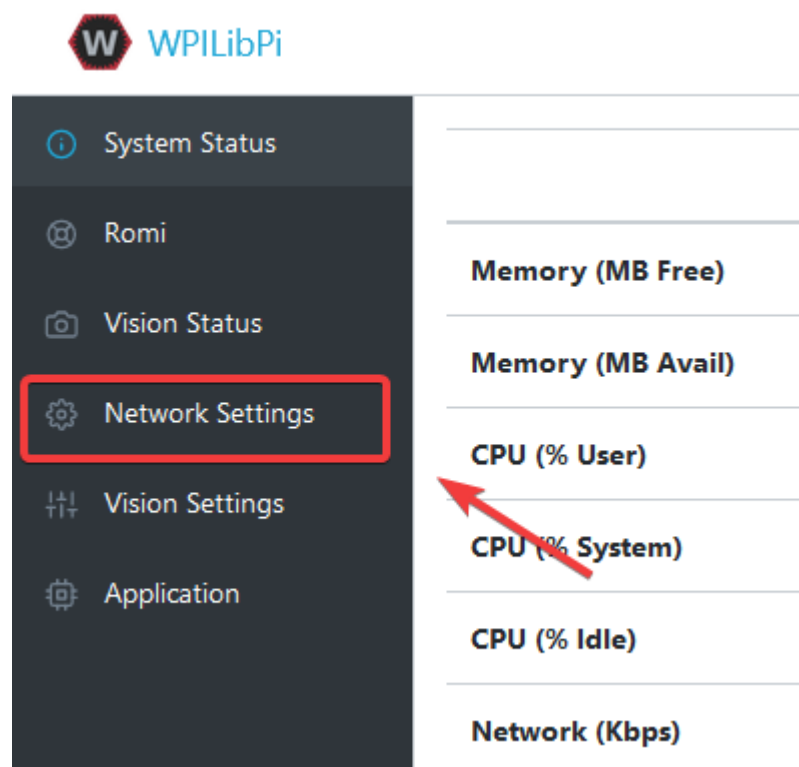
Bridge mode allows your Romi robot to connect to a WiFi network instead of acting as an Access Point (AP). This is especially useful in remote learning environments, as you can use the internet while using the Romi without extra hardware.

Note: Bridge mode is not likely to work properly in restricted network environments (Educational Institutions).


1. Enable *Writable* in the top menu.



2. Click on *Network Settings*.



3. The following network settings must be applied:


WPILibPi

Read-Only
Writable

- ⓘ System Status
- ⊞ Romi
- 📷 Vision Status
- ⚙️ **Network Settings**
- ⚙️ Vision Settings
- ⚙️ Application

Ethernet Address

DHCP

WiFi Mode

Bridge

SSID

MyNetwork

WPA2 Passphrase

MyPassword

WiFi Address

DHCP

💾 Save

- **Ethernet:** DHCP
- **WiFi Mode:** Bridge
- **SSID:** SSID (name) of your network
- **WPA2 Passphrase:** Password of your wifi network
- **WiFi Address:** DHCP

Once the settings are applied, please reboot the Romi. You should now be able to navigate to `wpilibpi.local` in your web browser while connected to your specified network.

Unable to Access Romi

If the Romi has the correct bridge settings and you are unable to access it, we have a few workarounds.

- Ethernet into the Romi
- Reimage the Romi

Some restricted networks can interfere with the hostname of the Romi resolving, you can workaround this by using [Angry IP Scanner](#) to find the IP address.

Warning: Angry IP Scanner is flagged by some antivirus as spyware as it pings devices on your network! It is a safe application!

43.5 Programming the Romi

Writing a program for the Romi is very similar to writing a program for a regular FRC robot. In fact, all the same tools (Visual Studio Code, Driver Station, SmartDashboard, etc) can be used with the Romi.

43.5.1 Creating a Romi Program

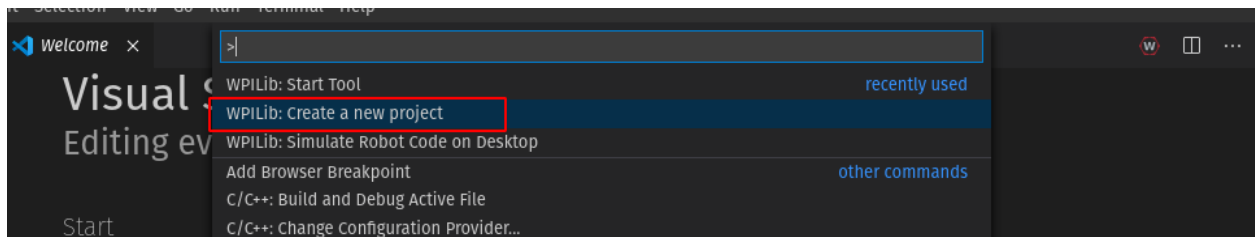
Creating a new program for a Romi is no different than creating a normal FRC program, similar to the *Zero To Robot* programming steps.

WPILib comes with several templates for Romi projects, including ones based on TimedRobot, and a Command-Based project template. Additionally, an example project is also provided which showcases some of the built-in functionality of the Romi. This article will walk through creating a project from this example.

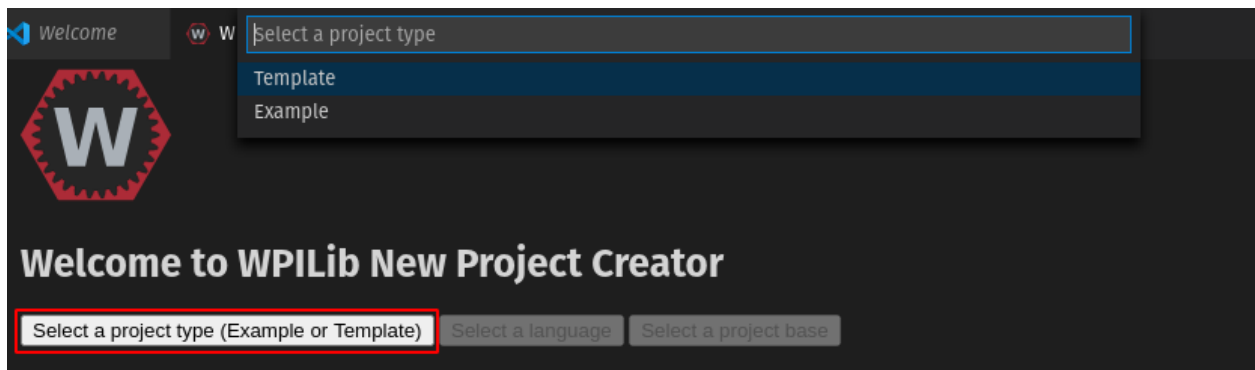
Note: In order to program the Romi using C++, a compatible C++ desktop compiler must be installed. See *Robot Simulation - Additional C++ Dependency*.

Creating a New WPILib Romi Project

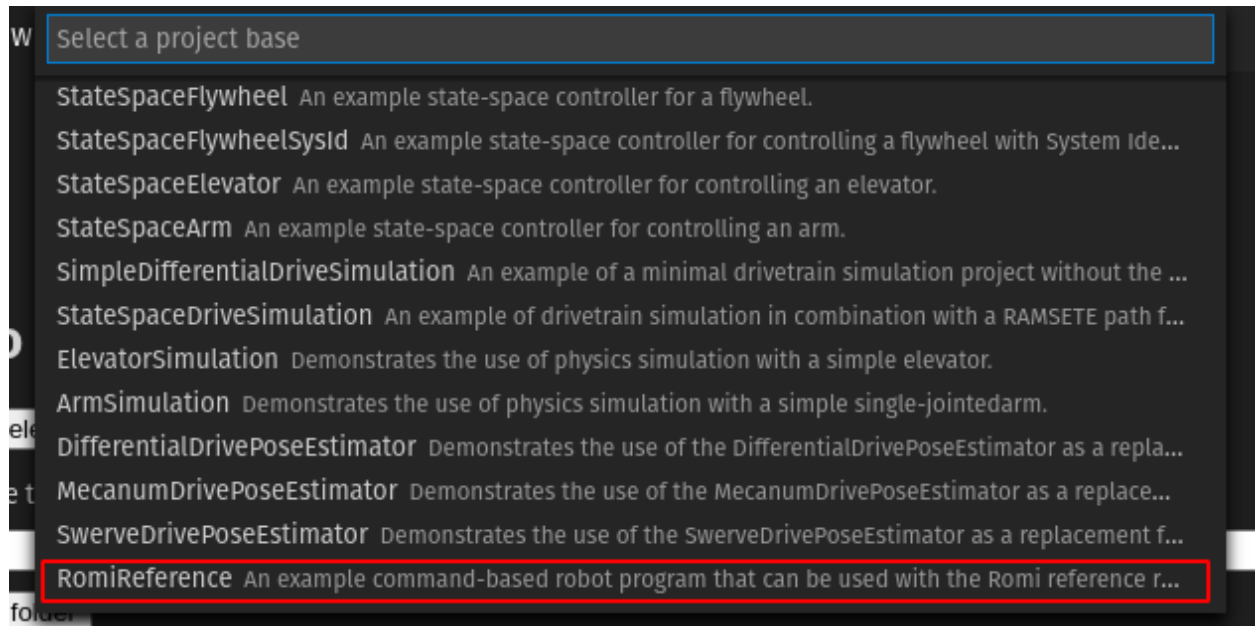
Bring up the Visual Studio Code command palette with Ctrl+Shift+P, and type “New project” into the prompt. Select the “Create a new project” command:



This will bring up the “New Project Creator Window”. From here, click on “Select a project type (Example or Template)”, and pick “Example” from the prompt that appears:



Next, a list of examples will appear. Scroll through the list to find the “RomiReference” example:



Fill out the rest of the fields in the “New Project Creator” and click “Generate Project” to create the new robot project.

Running a Romi Program

Once the robot project is generated, it is essentially ready to run. The project has a pre-built Drivetrain class and associated default command that lets you drive the Romi around using a gamepad.

One aspect where a Romi project differs from a regular FRC robot project is that the code is not deployed directly to the Romi. Instead, a Romi project runs on your development computer, and leverages the WPILib simulation framework to communicate with the Romi robot.

To run a Romi program, first, ensure that your Romi is powered on. Once you connect to the WPILibPi-<number> network broadcast by the Romi, press F5 to start running the Romi program on your computer.

If you changed the Romi network settings (for example, to connect it to your own WiFi network), you can change the IP address that your program uses to connect to the Romi. To do this, open the “build.gradle” file and update the envVar line to the appropriate IP address.

```

26 // Enable simulation gui support. Must check the box in vscode to enable support
27 // upon debugging
28 simulation wpi.deps.sim.gui(wpi.platforms.desktop, false)
29 simulation wpi.deps.sim.driverstation(wpi.platforms.desktop, false)
30
31 // Websocket extensions require additional configuration.
32 // simulation wpi.deps.sim.ws_server(wpi.platforms.desktop, false)
33 simulation wpi.deps.sim.ws_client(wpi.platforms.desktop, false)
34 }
35
36 // Set the websocket remote host (the Romi IP address).
37 sim {
38   envVar "HALSIMWS_HOST", "10.0.0.2"
39 }

```

If all goes well, you should see a line in the console output that reads “HALSimWS: WebSocket Connected”:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

Will attempt to connect to ws://192.168.1.23:3300/wpilibws
HALSim WS Client Extension Initialized
HAL Extensions: Successfully loaded extension
Connection Attempt 1
***** Robot program starting *****
HALSimWS: WebSocket Connected
Default simulationInit() method... Override me!
```

Your Romi code is now running!

Note: By default, the Romi templates and examples are set up so that positive drive values correspond to the Romi moving forward. Since forward movement on a Joystick produces negative values, you will need to invert the values in appropriate places (e.g. the xSpeed parameter to arcadeDrive on a DifferentialDrive object).

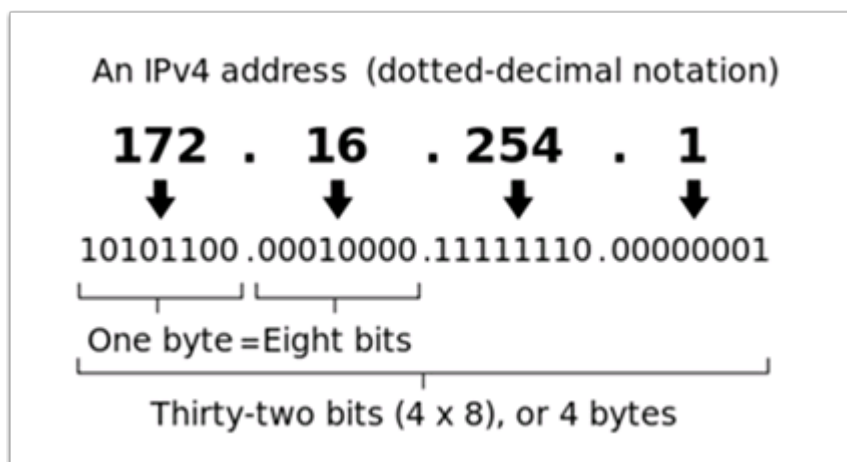
Networking Introduction

This section outlines basic robot configuration and usage relating to communication between the driver station and roboRIO.

44.1 Networking Basics

44.1.1 What is an IP Address?

An IP address is a unique string of numbers, separated by periods that identifies each device on a network. Each IP address is divided up into 4 sections (octets) ranging from 0-255.



As shown above, this means that each IP address is a 32-bit address meaning there are 2^{32} addresses, or nearly 4,300,000,000 addresses possible. However, most of these are used publicly for things like web servers.

This brings up our **first key point** of IP Addressing: Each device on the network must have a unique IP address. No two devices can have the same IP address, otherwise collisions will occur.

Since there are only 4-billion addresses, and there are more than 4-billion computers connected to the internet, we need to be as efficient as possible with giving out IP addresses.

This brings us to public vs. private addresses.

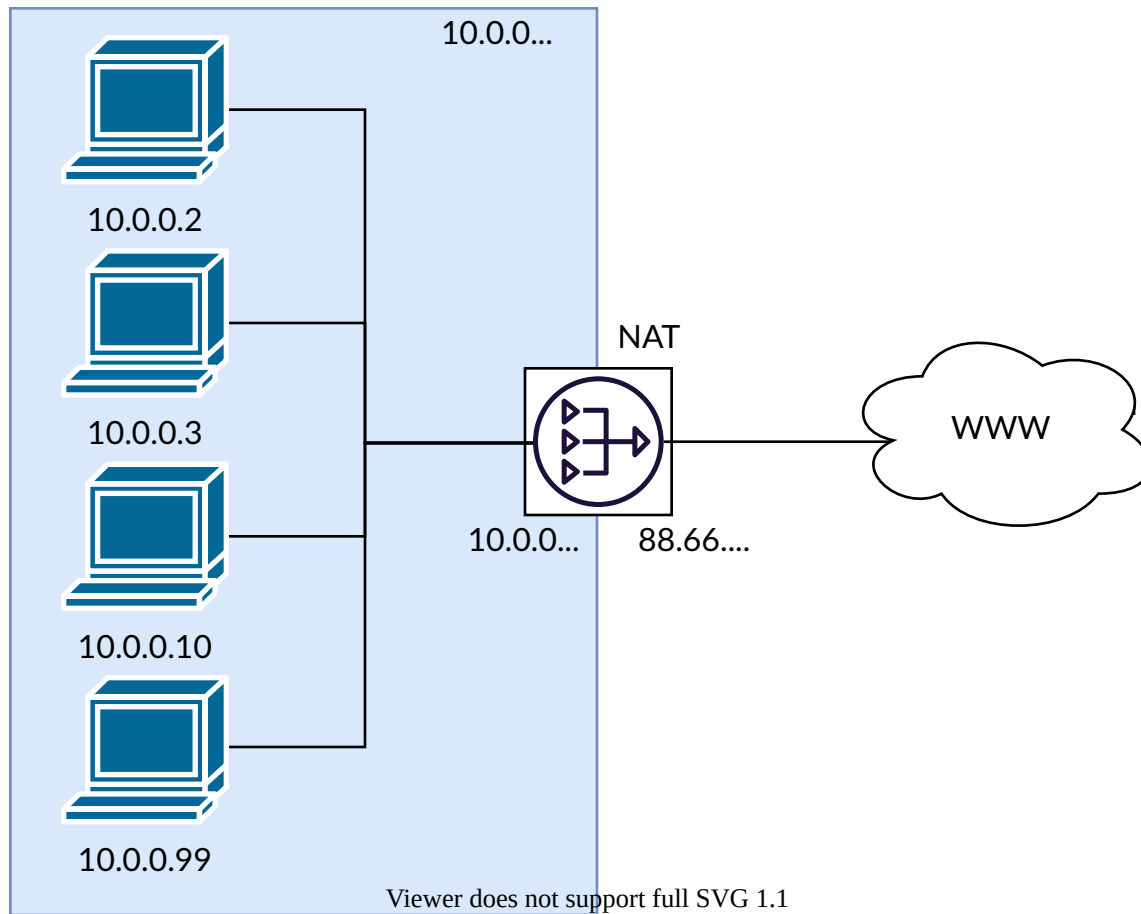
44.1.2 Public vs Private IP Addresses

To be efficient with using IP Addresses, the idea of “Reserved IP Ranges” was implemented. In short, this means that there are ranges of IP Addresses that will never be assigned to web servers, and will only be used for local networks, such as those in your house.

Key point #2: Unless you are directly connecting to your internet provider’s basic modem (no router function), your device will have an IP Address in one of these ranges. This means that at any local network, such as: your school, work office, home, etc., your device will 99% of the time have an IP address in a range listed below:

Class	Bits	Start Address	End Address	Number of Addresses
A	24	10.0.0.0	10.255.255.255	16,777,216
B	20	172.16.0.0	172.31.255.255	1,048,576
C	16	192.168.0.0	192.168.255.255	65,536

These reserved ranges let us assign one “unreserved IP Address” to an entire house, and then use multiple addresses in a reserved range to connect more than one computer to the internet. A process on the home’s internet router known as **NAT** (Network Address Translation), handles the process of keeping track which private IP is requesting data, using the public IP to request that data from the internet, and then passing the returned data back to the private IP that requested it. This allows us to use the same reserved IP addresses for many local networks, without causing any conflicts. An image of this process is presented below.



Note: For the FRC® networks, we will use the 10.0.0.0 range. This range allows us to use the 10.TE.AM.xx format for IP addresses, whereas using the Class B or C networks would only allow a subset of teams to follow the format. An example of this formatting would be 10.17.50.1 for FRC Team 1750.

44.1.3 How are these addresses assigned?

We've covered the basics of what IP addresses are, and which IP addresses we will use for the FRC competition, so now we need to discuss how these addresses will get assigned to the devices on our network. We already stated above that we can't have two devices on the same network with the same IP Address, so we need a way to be sure that every device receives an address without overlapping. This can be done Dynamically (automatic), or Statically (manual).

Dynamically

Dynamically assigning IP addresses means that we are letting a device on the network manage the IP address assignments. This is done through the Dynamic Host Configuration Protocol (DHCP). DHCP has many components to it, but for the scope of this document, we will think of it as a service that automatically manages the network. Whenever you plug in a new device to the network, the DHCP service sees the new device, then provides it with an available IP address and the other network settings required for the device to communicate. This can mean that there are times we do not know the exact IP address of each device.

What is a DHCP server?

A DHCP server is a device that runs the DHCP service to monitor the network for new devices to configure. In larger businesses, this could be a dedicated computer running the DHCP service and that computer would be the DHCP server. For home networks, FRC networks, and other smaller networks, the DHCP service is usually running on the router; in this case, the router is the DHCP server.

This means that if you ever run into a situation where you need to have a DHCP server assigning IP addresses to your network devices, it's as simple as finding the closest home router, and plugging it in.

Statically

Statically assigning IP addresses means that we are manually telling each device on the network which IP address we want it to have. This configuration happens through a setting on each device. By disabling DHCP on the network and assigning the addresses manually, we get the benefit of knowing the exact IP address of each device on the network, but because we set each one manually and there is no service keeping track of the used IP addresses, we have to keep track of this ourselves. While statically setting IP addresses, we must be careful not to assign duplicate addresses, and must be sure we are setting the other network settings (such as subnet mask and default gateway) correctly on each device.

44.1.4 What is link-local?

If a device does not have an IP address, then it cannot communicate on a network. This can become an issue if we have a device that is set to dynamically acquire its address from a DHCP server, but there is no DHCP server on the network. An example of this would be when you have a laptop directly connected to a roboRIO and both are set to dynamically acquire an IP address. Neither device is a DHCP server, and since they are the only two devices on the network, they will not be assigned IP addresses automatically.

Link-local addresses give us a standard set of addresses that we can “fall-back” to if a device set to acquire dynamically is not able to acquire an address. If this happens, the device will assign itself an IP address in the 169.254.xx.yy address range; this is a link-local address. In our roboRIO and computer example above, both devices will realize they haven't been assigned an IP address and assign themselves a link-local address. Once they are both assigned addresses in the 169.254.xx.yy range, they will be in the same network and will be able to communicate, even though they were set to dynamic and a DHCP server did not assign addresses.

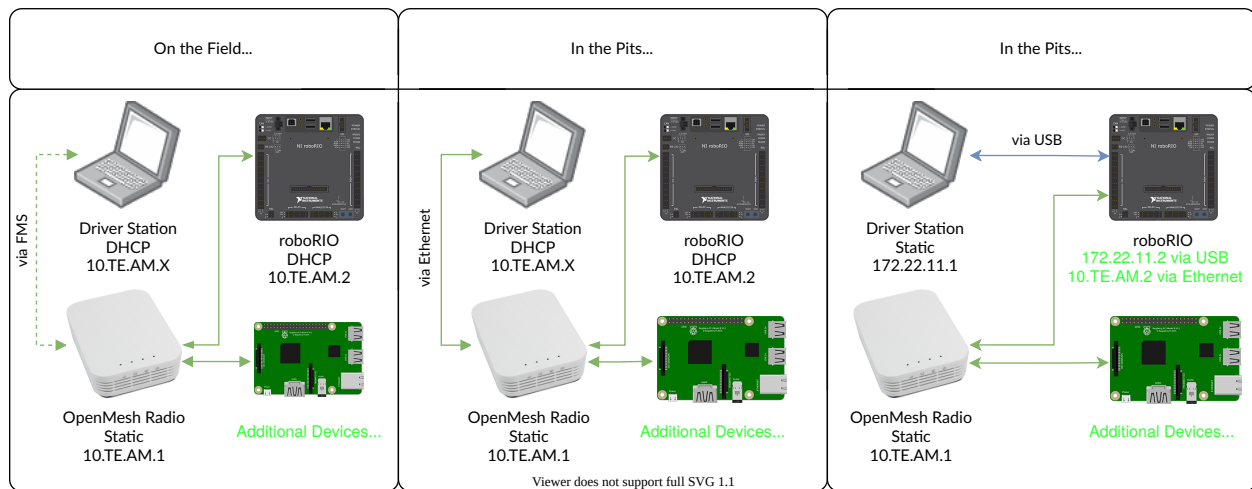
44.1.5 IP Addressing for FRC

See the [IP Networking Article](#) for more information.

Mixing Dynamic and Static Configurations

While on the field, the team should not notice any issues with having devices set statically in the 10.TE.AM.xx range, and having the field assign DHCP addresses as long as there are no IP address conflicts as referred to in the section above.

In the pits, a team may encounter issues with mixing Static and DHCP devices for the following reason. As mentioned above, DHCP devices will fall back to a link-local address (169.254.xx.yy) if a server isn't present. For static devices, the IP address will always be the same. If the DHCP server is not present and the roboRIO, driver station, and laptop fall back to link-local addresses, the statically set devices in the 10.TE.AM.xx range will be in a different network and not visible to those with link-local addresses. A visual description of this is provided below:



Warning: When connected via USB to the roboRIO, a *Port Forwarding* configuration is required to access devices connected to the OpenMesh radio (on the green network shown above).

44.1.6 mDNS

mDNS, or multicast Domain Name System is a protocol that allows us to benefit from the features of DNS, without having a DNS server on the network. To make this clearer, let's take a step back and talk about what DNS is.

What is DNS?

DNS (Domain Name System) can become a complex topic, but for the scope of this paper, we are going to just look at the high level overview of DNS. In the most basic explanation, DNS is what allows us to relate human-friendly names for network devices to IP Addresses, and keep track of those IP addresses if they change.

Example 1: Let's look at the site `www.google.com`. The IP address for this site is `172.217.164.132`, however that is not very user friendly to remember!

Whenever a user types `www.google.com` into their computer, the computer contacts the DNS server (a setting provided by DHCP!) and asks what is the IP address on file for `www.google.com`. The DNS server returns the IP address and then the computer is able to use that to connect to the Google web site.

Example 2: On your home network, you have a server named `MYCOMPUTER` that you want to connect to from your laptop. Your network uses DHCP so you don't know the IP Address of `MYCOMPUTER`, but DNS allows you to connect just by using the `MYCOMPUTER` name. Additionally, whenever the DHCP assignments refresh, `MYCOMPUTER` may end up with a different address, but because you're connecting by using the `MYCOMPUTER` name instead of a specific IP address, the DNS record was updated and you're still able to connect.

This is the second benefit to DNS, and the most relevant for FRC. With DNS, if we reference devices by their friendly name instead of IP Address, we don't have to change anything in our program if the IP Address changes. DNS will keep track of the changes and return the new address if it ever changes.

DNS for FRC

On the field and in the pits, there is no DNS server that allows us to perform the lookups like we do for the Google web site, but we'd still like to have the benefits of not remembering every IP Address, and not having to guess at every device's address if DHCP assigns a different address than we expect. This is where mDNS comes into the picture.

mDNS provides us the same benefits as traditional DNS, but is just implemented in a way that does not require a server. Whenever a user asks to connect to a device using a friendly name, mDNS sends out a message asking the device with that name to identify itself. The device with the name then sends a return message including its IP address so all devices on the network can update their information. mDNS is what allows us to refer to our roboRIO as `roboRIO-TEAM-FRC.local` and have it connect on a DHCP network.

Note: If a device used for FRC does not support mDNS, then it will be assigned an IP Address in the `10.TE.AM.20 - 10.TE.AM.255` range, but we won't know the exact address to connect and we won't be able to use the friendly name like before. In this case, the device would need to have a static IP Address.

mDNS - Principles

Multicast Domain Name System (mDNS) is a system which allows for resolution of host names to IP addresses on small networks with no dedicated name server. To resolve a host name a device sends out a multicast message to the network querying for the device. The device then responds with a multicast message containing its IP. Devices on the network can store this information in a cache so subsequent requests for this address can be resolved from the cache without repeating the network query.

mDNS - Providers

To use mDNS, an mDNS implementation is required to be installed on your PC. Here are some common mDNS implementations for each major platform:

Windows:

- **NI mDNS Responder:** Installed with the NI FRC Game Tools
- **Apple Bonjour:** Installed with iTunes

OSX:

- **Apple Bonjour:** Installed by default

Linux:

- **nss-mDNS/Avahi/Zeroconf:** Installed and enabled by default on some Linux variants (such as Ubuntu or Mint). May need to be installed or enabled on others (such as Arch)

mDNS - Firewalls

Note: Depending on your PC configuration, no changes may be required, this section is provided to assist with troubleshooting.

To work properly mDNS must be allowed to pass through your firewall. Because the network traffic comes from the mDNS implementation and not directly from the Driver Station or IDE, allowing those applications through may not be sufficient. There are two main ways to resolve mDNS firewall issues:

- Add an application/service exception for the mDNS implementation (NI mDNS Responder is C:\Program Files\National Instruments\Shared\mDNS Responder\nimdnsResponder.exe)
- Add a port exception for traffic to/from UDP 5353. IP Ranges:
 - 10.0.0.0 - 10.255.255.255
 - 172.16.0.0 - 172.31.255.255
 - 192.168.0.0 - 192.168.255.255
 - 169.254.0.0 - 169.254.255.255
 - 224.0.0.251

mDNS - Browser support

Most web-browsers should be able to utilize the mDNS address to access the roboRIO web server as long as an mDNS provider is installed. These browsers include Microsoft Edge, Firefox, and Google Chrome.

44.1.7 USB

If using the USB interface, no network setup is required (you do need the *Installing the FRC Game Tools* installed to provide the roboRIO USB Driver). The roboRIO driver will automatically configure the IP address of the host (your computer) and roboRIO and the software listed above should be able to locate and utilize your roboRIO.

44.1.8 Ethernet/Wireless

The *Programming your Radio* will enable the DHCP server on the OpenMesh radio in the home use case (AP mode), if you are putting the OpenMesh in bridge mode and using a router, you can enable DHCP addressing on the router. The bridge is set to the same team based IP address as before (10.TE.AM.1) and will hand out DHCP address from 10.TE.AM.20 to 10.TE.AM.199. When connected to the field, FMS will also hand out addresses in the same IP range.

44.1.9 Summary

IP Addresses are what allow us to communicate with devices on a network. For FRC, these addresses are going to be in the 10.TE.AM.xx range if we are connected to a DHCP server or if they are assigned statically, or in the link-local 169.254.xx.yy range if the devices are set to DHCP, but there is no server present. For more information on how IP Addresses work, see [this](#) article by Microsoft.

If all devices on the network support mDNS, then all devices can be set to DHCP and referred to using their friendly names (ex. roboRIO-TEAM-FRC.local). If some devices do not support mDNS, they will need to be set to use static addresses.

If all devices are set to use DHCP or Static IP assignments (with correct static settings), the communication should work in both the pit and on the field without any changes needed. If there are a mix of some Static and some DHCP devices, then the Static devices will connect on the field, but will not connect in the pit. This can be resolved by either setting all devices to static settings, or leaving the current settings and providing a DHCP server in the pit.

44.2 IP Configurations

Note: This document describes the IP configuration used at events, both on the fields and in the pits, potential issues and workaround configurations.

44.2.1 TE.AM IP Notation

The notation TE.AM is used as part of IPs in numerous places in this document. This notation refers to splitting your four digit team number into two digit pairs for the IP address octets.

Example: 10.TE.AM.2

Team 12 - 10.0.12.2

Team 122 - 10.1.22.2

Team 1212 - 10.12.12.2

Team 3456 - 10.34.56.2

44.2.2 On the Field

This section describes networking when connected to the Field Network for match play

On the Field DHCP Configuration

The Field Network runs a DHCP server with pools for each team that will hand out addresses in the range of 10.TE.AM.20 and up with subnet masks

- OpenMesh OM5P-AN or OM5P-AC radio - Static 10.TE.AM.1 programmed by Kiosk
- roboRIO - DHCP 10.TE.AM.2 assigned by the Robot Radio
- Driver Station - DHCP (“Obtain an IP address automatically”) 10.TE.AM.X assigned by field
- IP camera (if used) - DHCP 10.TE.AM.Y assigned by Robot Radio
- Other devices (if used) - DHCP 10.TE.AM.Z assigned by Robot Radio

On the Field Static Configuration

It is also possible to configure static IPs on your devices to accommodate devices or software which do not support mDNS. When doing so you want to make sure to avoid addresses that will be in use when the robot is on the field network. These addresses are 10.TE.AM.1 and 10.TE.AM.4 for the OpenMesh radio and the field access point and anything 10.TE.AM.20 and up which may be assigned to a device still configured for DHCP. The roboRIO network configuration can be set from the webdashboard.

- OpenMesh radio - Static 10.TE.AM.1 programmed by Kiosk
- roboRIO - Static 10.TE.AM.2 would be a reasonable choice, subnet mask of 255.255.255.0 (default)
- Driver Station - Static 10.TE.AM.5 would be a reasonable choice, subnet mask **must** be 255.0.0.0
- IP Camera (if used) - Static 10.TE.AM.11 would be a reasonable choice, subnet 255.255.255.0 should be fine
- Other devices - Static 10.TE.AM.6 - .10 or .12 - .19 (.11 if camera not present) subnet 255.255.255.0

44.2.3 In the Pits

Note: New for 2018: There is now a DHCP server running on the wired side of the Robot Radio in the event configuration.

In the Pits DHCP Configuration

- OpenMesh radio - Static 10.TE.AM.1 programmed by Kiosk.
- roboRIO - 10.TE.AM.2, assigned by Robot Radio
- Driver Station - DHCP (“Obtain an IP address automatically”), 10.TE.AM.X, assigned by Robot Radio
- IP camera (if used) - DHCP, 10.TE.AM.Y, assigned by Robot Radio
- Other devices (if used) - DHCP, 10.TE.AM.Z, assigned by Robot Radio

In the Pits Static Configuration

It is also possible to configure static IPs on your devices to accommodate devices or software which do not support mDNS. When doing so you want to make sure to avoid addresses that will be in use when the robot is on the field network. These addresses are 10.TE.AM.1 and 10.TE.AM.4 for the OpenMesh radio

44.3 roboRIO Network Troubleshooting

The roboRIO and FRC® tools use dynamic IP addresses (DHCP) for network connectivity. This article describes steps for troubleshooting networking connectivity between your PC and your roboRIO

44.3.1 Ping the roboRIO using mDNS

The first step to identifying roboRIO networking issues is to isolate if it is an application issue or a general network issue. To do this, click **Start -> type cmd -> press Enter** to open the command prompt. Type `ping roboRIO-####-FRC.local` where #### is your team number (with no leading zeroes) and press enter. If the ping succeeds, the issue is likely with the specific application, verify your team number configuration in the application, and check your firewall configuration.

44.3.2 Ping the roboRIO IP Address

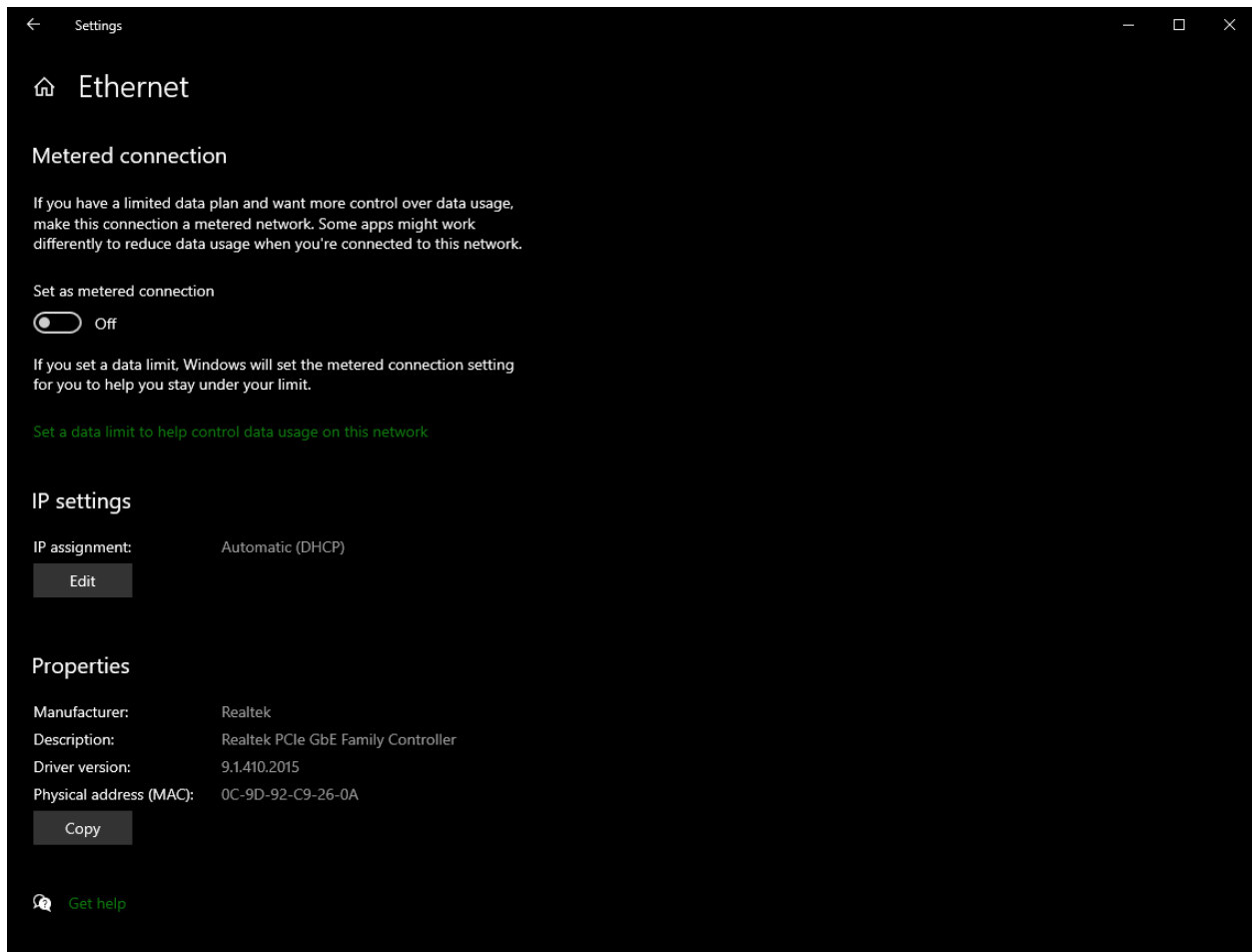
If there is no response, try pinging `10.TE.AM.2` (*TEAM IP Notation*) using the command prompt as described above. If this works, you have an issue resolving the mDNS address on your PC. The two most common causes are not having an mDNS resolver installed on the system and a DNS server on the network that is trying to resolve the `.local` address using regular DNS.

- Verify that you have an mDNS resolver installed on your system. On Windows, this is typically fulfilled by the NI FRC Game Tools. For more information on mDNS resolvers, see the [roboRIO Networking article](#).
- Disconnect your computer from any other networks and make sure you have the OM5P-AN configured as an access point, using the [FRC Radio Configuration Utility](#). Removing any other routers from the system will help verify that there is not a DNS server causing the issue.

44.3.3 Ping Fails

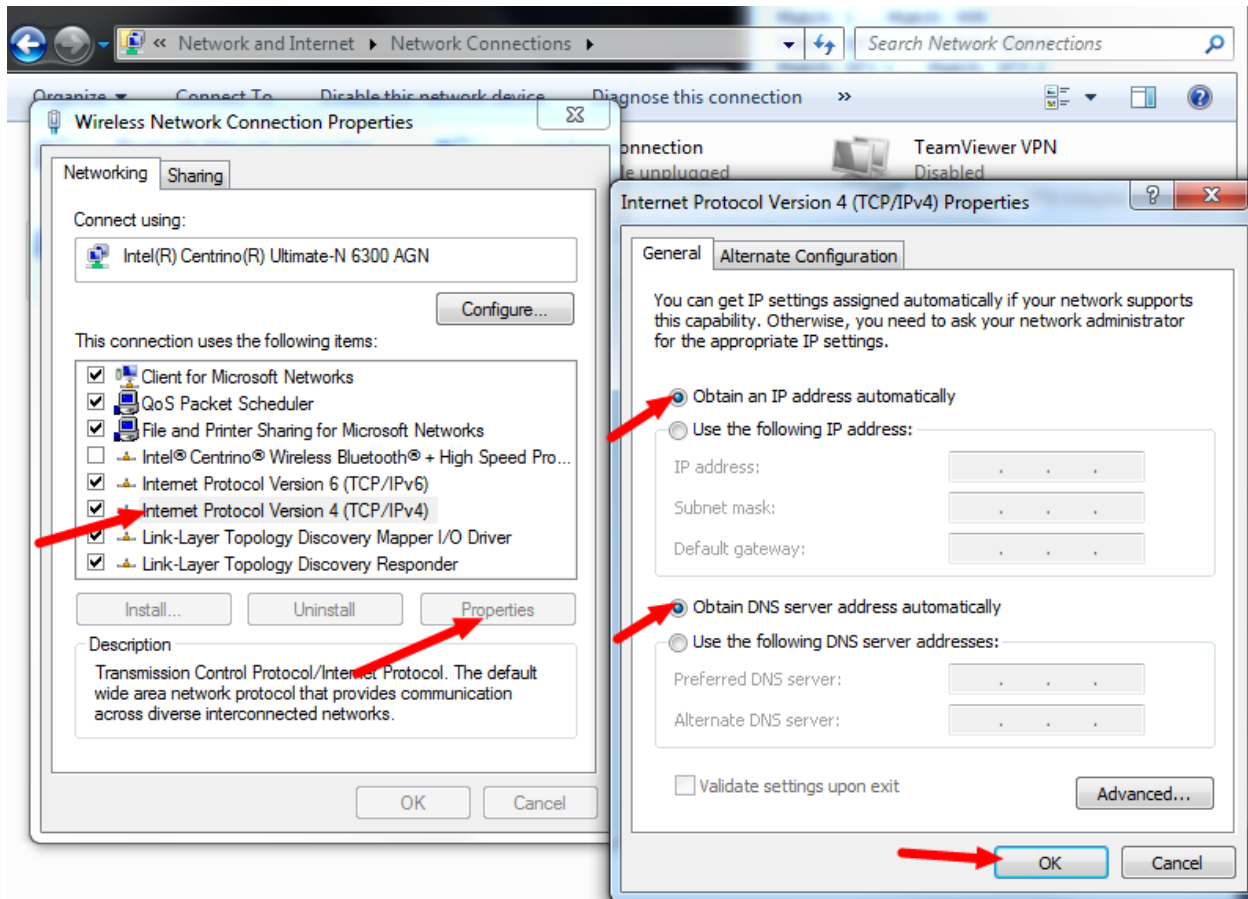
Windows 10

Windows 7



If pinging the IP address directly fails, you may have an issue with the network configuration

of the PC. The PC should be configured to **Automatic**. To check this, click *Start -> Settings -> Network & Internet*. Depending on your network, select *Wifi* or *Ethernet*. Then click on your connected network. Scroll down to **IP settings** and click *Edit* and ensure the *Automatic (DHCP)* option is selected.



If pinging the IP address directly fails, you may have an issue with the network configuration of the PC. The PC should be configured to **Obtain an Address Automatically** (also known as DHCP). To check this, click *Start -> Control Panel -> Network Connections -> Change adapter settings*, then right click on the appropriate interface (usually Local Area Connection for Ethernet or Wireless Network Connection for wireless) and select *Properties*. Click *Internet Protocol Version 4*, then click *Properties*. Make sure both radio buttons are set to *Obtain automatically*.

44.3.4 USB Connection Troubleshooting

If you are attempting to troubleshoot the USB connection, try pinging the roboRIO's IP address. As long as there is only one roboRIO connected to the PC, it should be configured as 172.22.11.2. If this ping fails, make sure you have the roboRIO connected and powered, and that you have installed the NI FRC Game Tools. The game tools installs the roboRIO drivers needed for the USB connection.

If this ping succeeds, but the .local ping fails, it is likely that either the roboRIO hostname is configured incorrectly, or you are connected to a DNS server which is attempting to resolve the .local address.

- Verify that your roboRIO has been *imaged for your team number*. This sets the hostname used by mDNS.
- *Disable all other network adapters*

44.3.5 Ethernet Connection

roboRIO-40 : System Configuration

Search [] Save Refresh

roboRIO
Name: roboRIO-40

CAN Interface
Name: can0

PCM
Name: PCM (1st device found)

PDP
Name: PDP (2nd device found)

NI roboRIO
Name: RIO0

ASRL1::INSTR
Name: ASRL1::INSTR

ASRL2::INSTR
Name: ASRL2::INSTR

System Settings

Hostname: roboRIO-40

IP Address: 10.0.40.2 (Ethernet)
172.22.11.2 (Ethernet)

DNS Name

Vendor: National Instruments

Model: roboRIO

Serial Number: 030498A9

Firmware Revision: 2.0.0f1

Operating System: NI Linux Real-Time ARMv7-A 3.2.35-rt52-2.0.0f0

Status: Running

System Start Time: 10/1/2014 2:15:56 PM

Image Title: roboRIO Image

Image Version: FRC_roboRIO_2015_v14

Comments

Locale: English

Update Firmware

Startup Settings

☐ Force Safe Mode

☒ Enable Console Out

☐ Disable RT Startup App

☐ Disable FPGA Startup App

☒ Enable Secure Shell Server (sshd)

☒ LabVIEW Project Access

System Resources

Total Physical Memory: 232 MB

Free Physical Memory: 103 MB

Total Virtual Memory: 232 MB

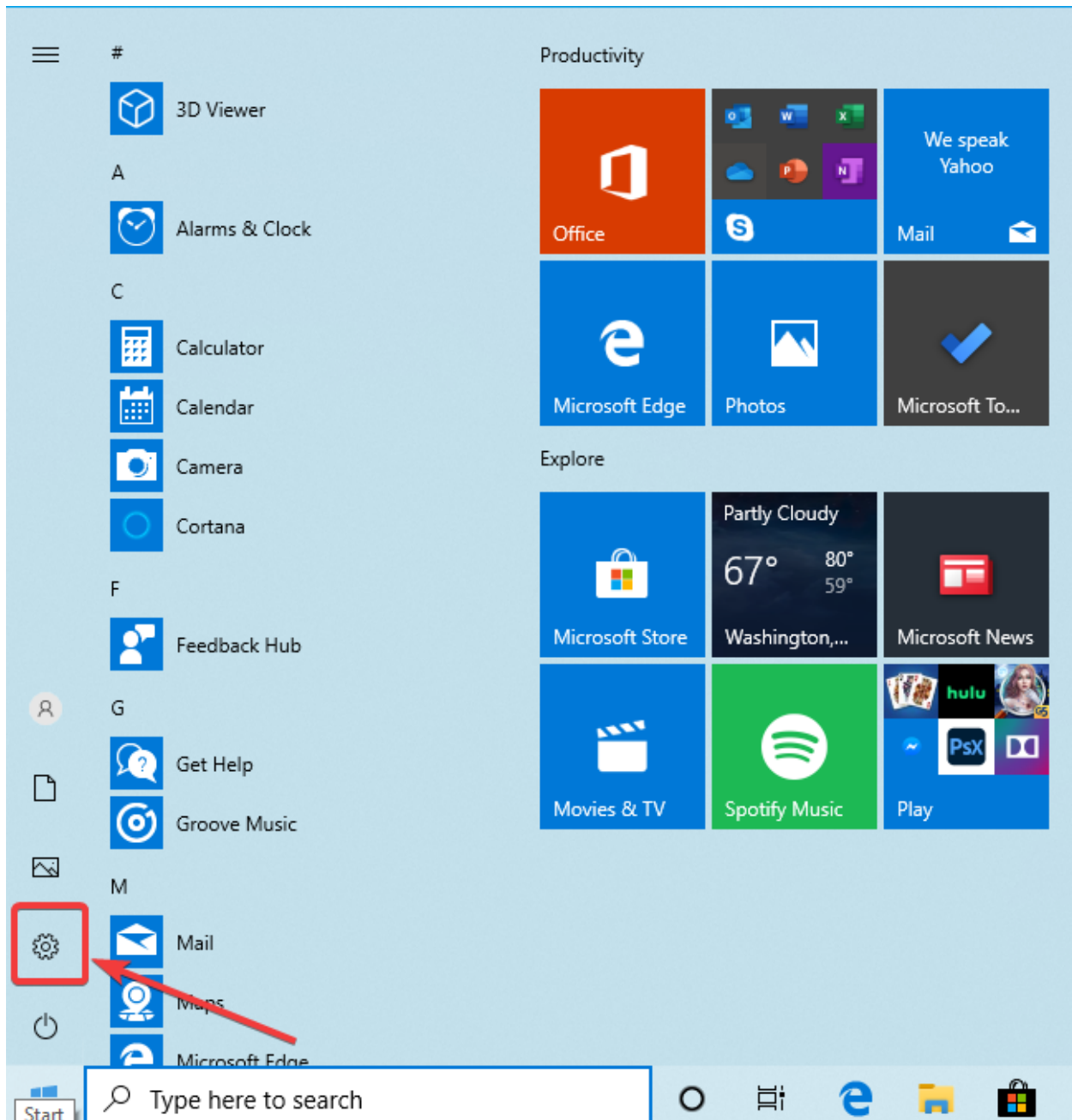
If you are troubleshooting an Ethernet connection, it may be helpful to first make sure that you can connect to the roboRIO using the USB connection. Using the USB connection, open the [roboRIO webdashboard](#) and verify that the roboRIO has an IP address on the ethernet interface. If you are tethering to the roboRIO directly this should be a self-assigned 169.*.*.* address, if you are connected to the OM5P-AN radio, it should be an address of the form 10.TE.AM.XX where TEAM is your four digit FRC team number. If the only IP address here is the USB address, verify the physical roboRIO ethernet connection.

44.3.6 Disabling Network Adapters

This is not always the same as turning the adapters off with a physical button or putting the PC into airplane mode. The following steps provide more detail on how to disable adapters.

Windows 10

Windows 7



Open the Settings application by clicking on the settings icon.

Settings

— □ ×

Windows Settings

Find a setting



System

Display, sound, notifications,
power

Devices

Bluetooth, printers, mouse



Phone

Link your Android, iPhone



Network & Internet

Wi-Fi, airplane mode, VPN



Personalization

Background, lock screen, colors



Apps

Uninstall, defaults, optional
features

Accounts

Your accounts, email, sync,
work, family

Time & Language

Speech, region, date



Gaming

Xbox Game Bar, captures, Game
Mode

Ease of Access

Narrator, magnifier, high
contrast

Search

Find my files, permissions



Privacy

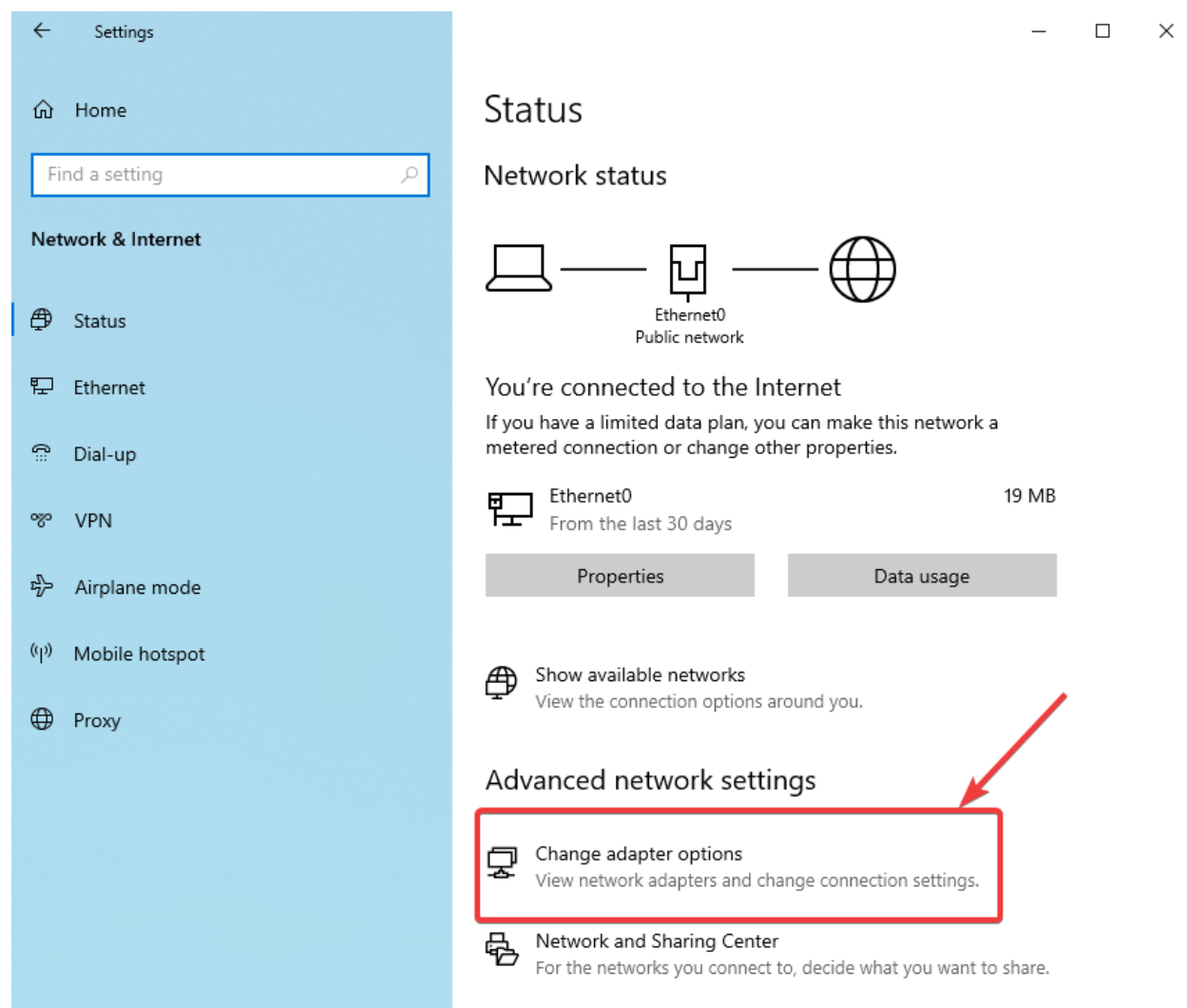
Location, camera, microphone



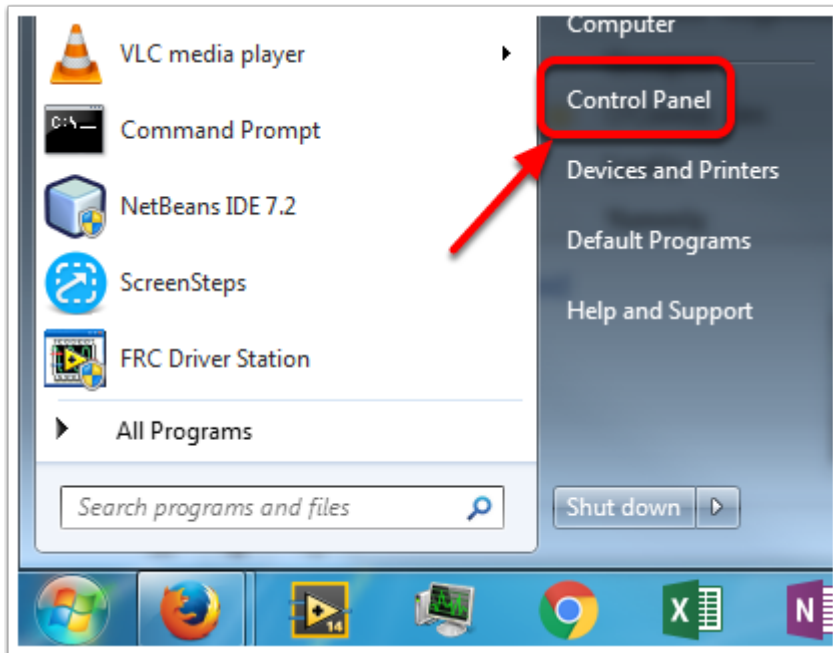
Update & Security

Windows Update, recovery,
backup

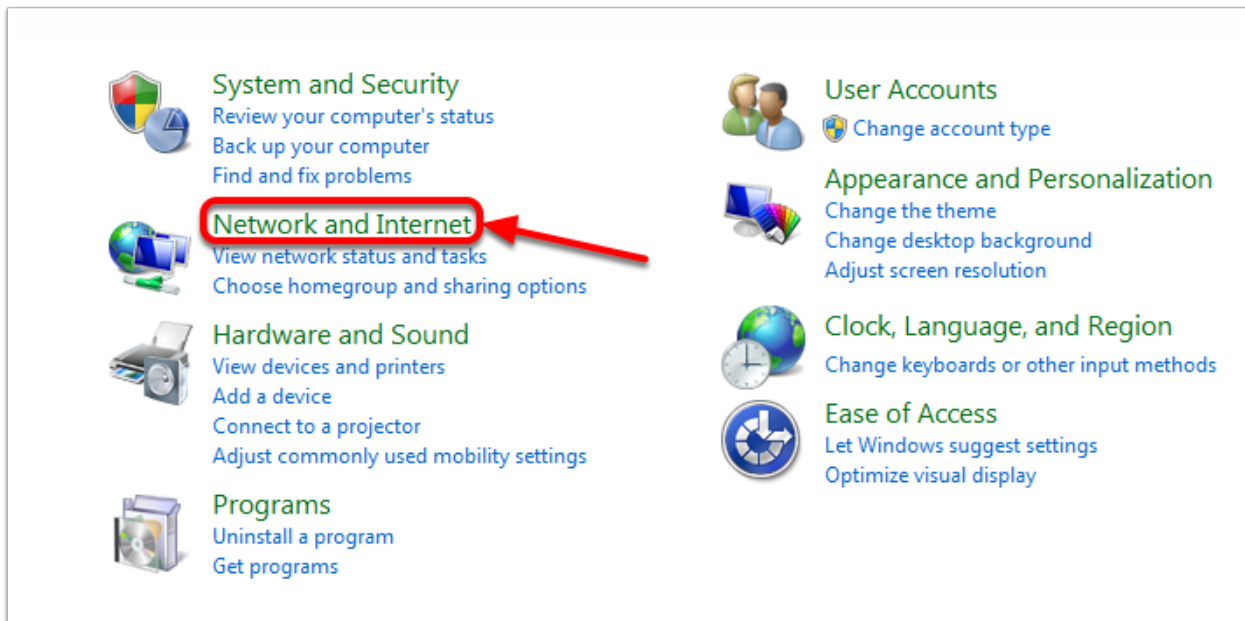
Choose the *Network & Internet* category.



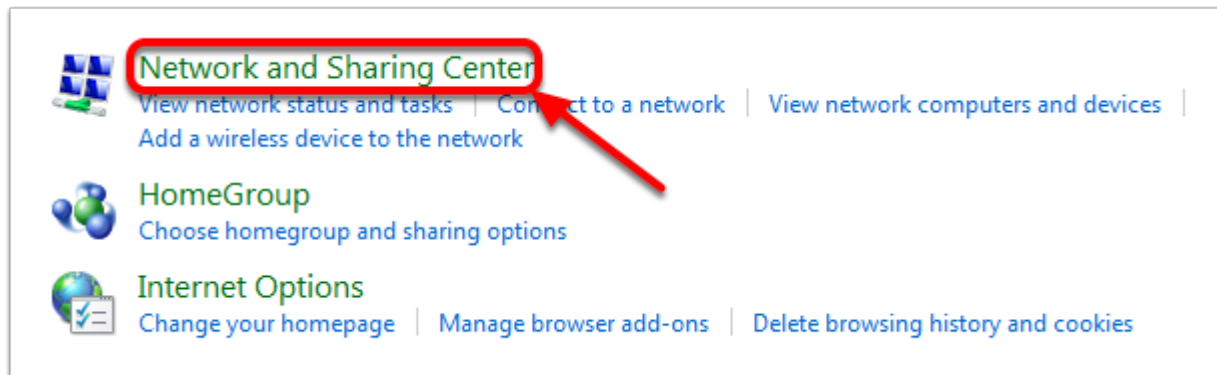
Click on *Change adapter options*.



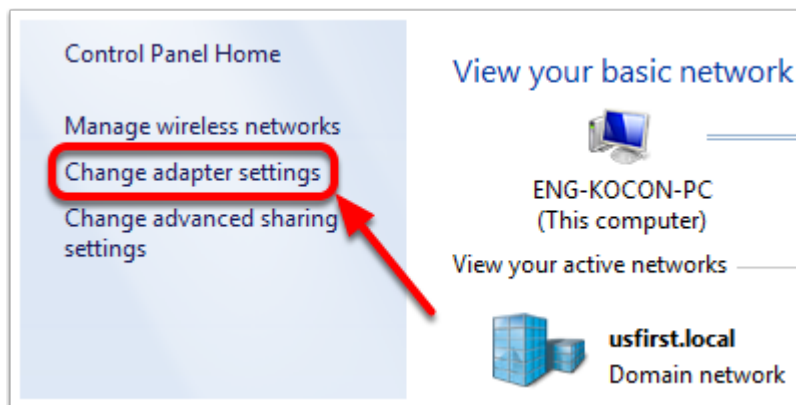
Open the Control Panel by going to *Start -> Control Panel*



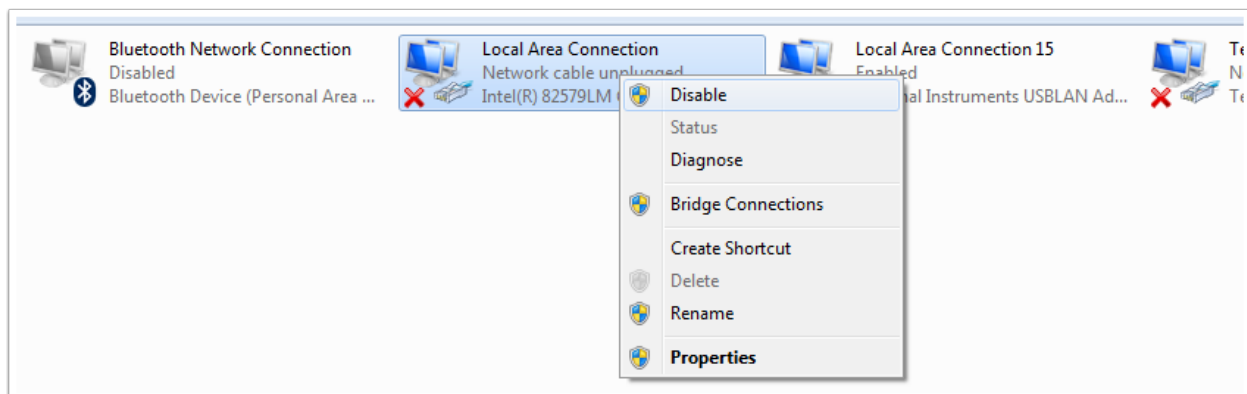
Choose the *Network and Internet* category.



Click *Network and Sharing Center*



On the left pane, click *Change Adapter Settings*.



For each adapter other than the one connected to the radio, right click on the adapter and select *Disable* from the menu.

44.3.7 Proxies

- Proxies. Having a proxy enabled may cause issues with the roboRIO networking.

44.4 Windows Firewall Configuration

Many of the programming tools used in FRC® need network access for various reasons. Depending on the exact configuration, the Windows Firewall may potentially interfere with this access for one or more of these programs.

44.4.1 Disabling Windows Firewall

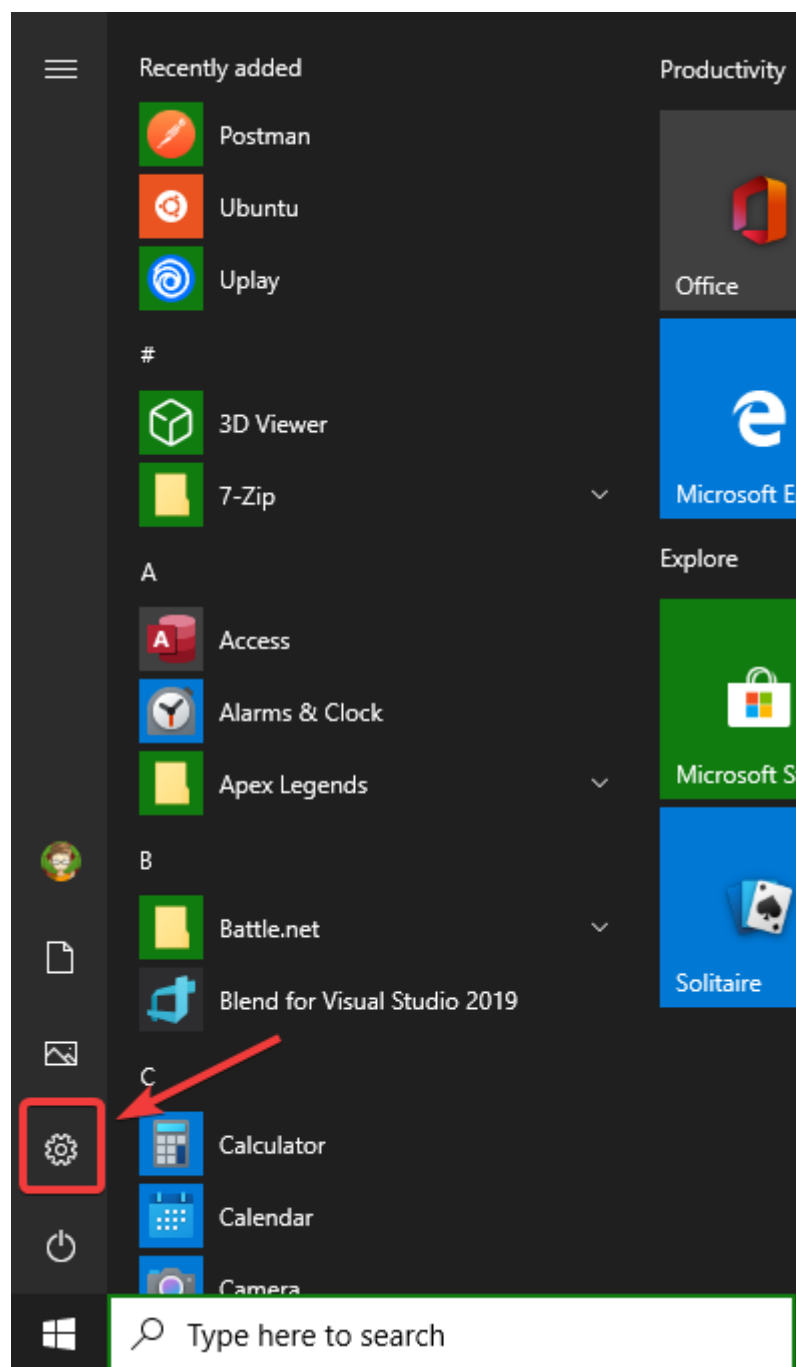
Important: Disabling your firewall requires administrator privileges to the PC. Additionally note that disabling the firewall is not recommended for computers that connect to the internet.

The easiest solution is to disable the Windows Firewall. Teams should beware that this does make the PC potentially more vulnerable to malware attacks if connecting to the internet.

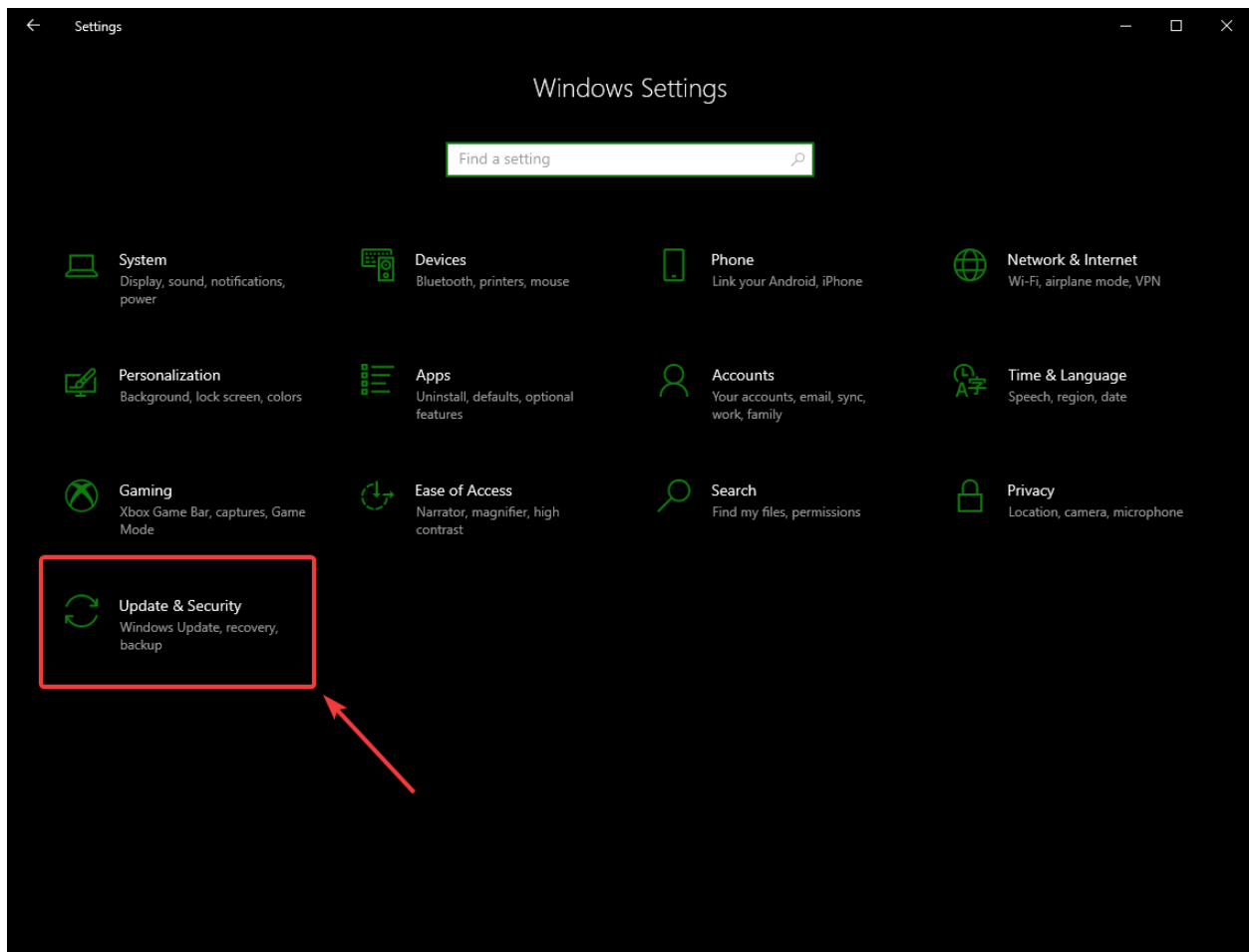
Windows 10

Windows 7

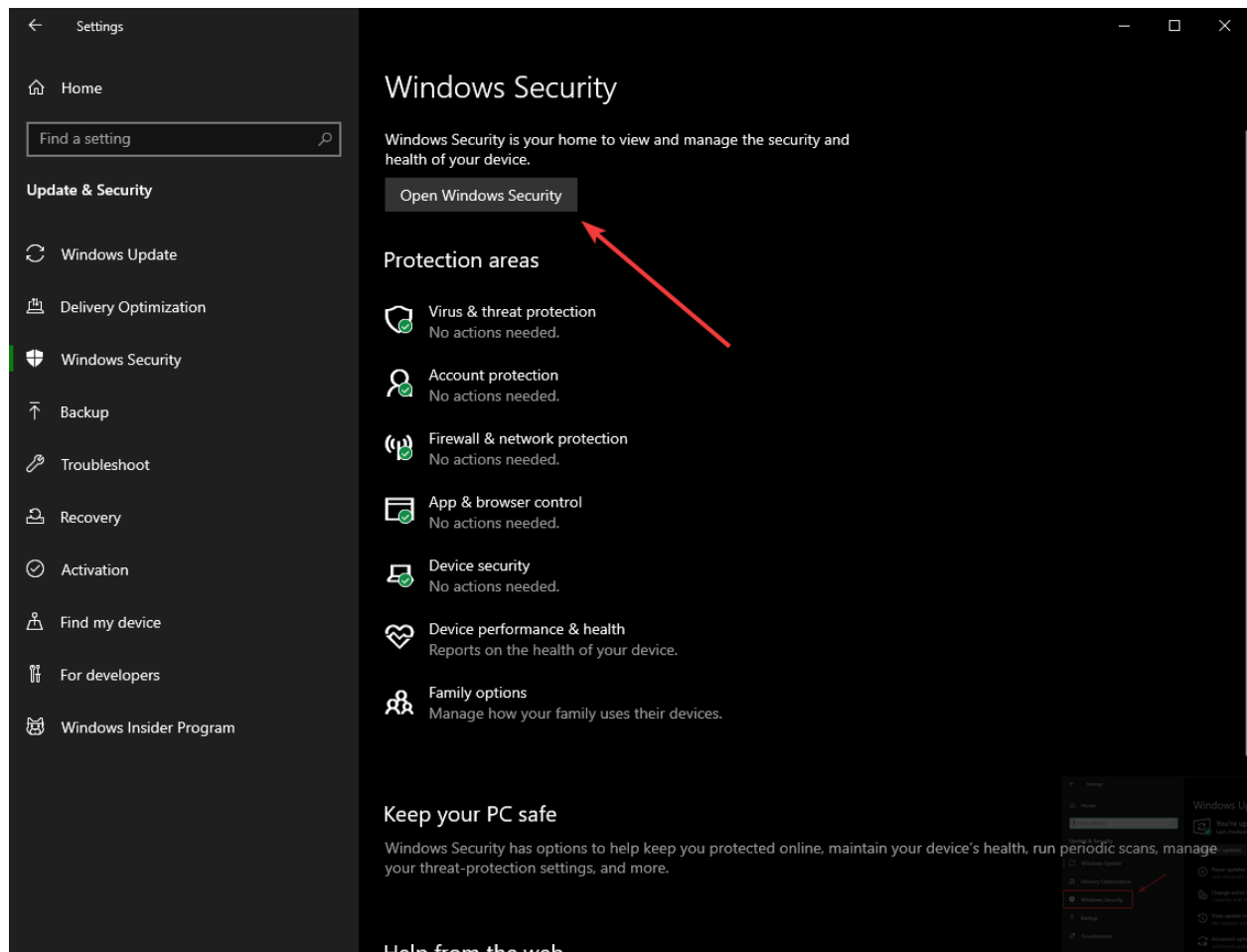
Click *Start* -> *Settings*



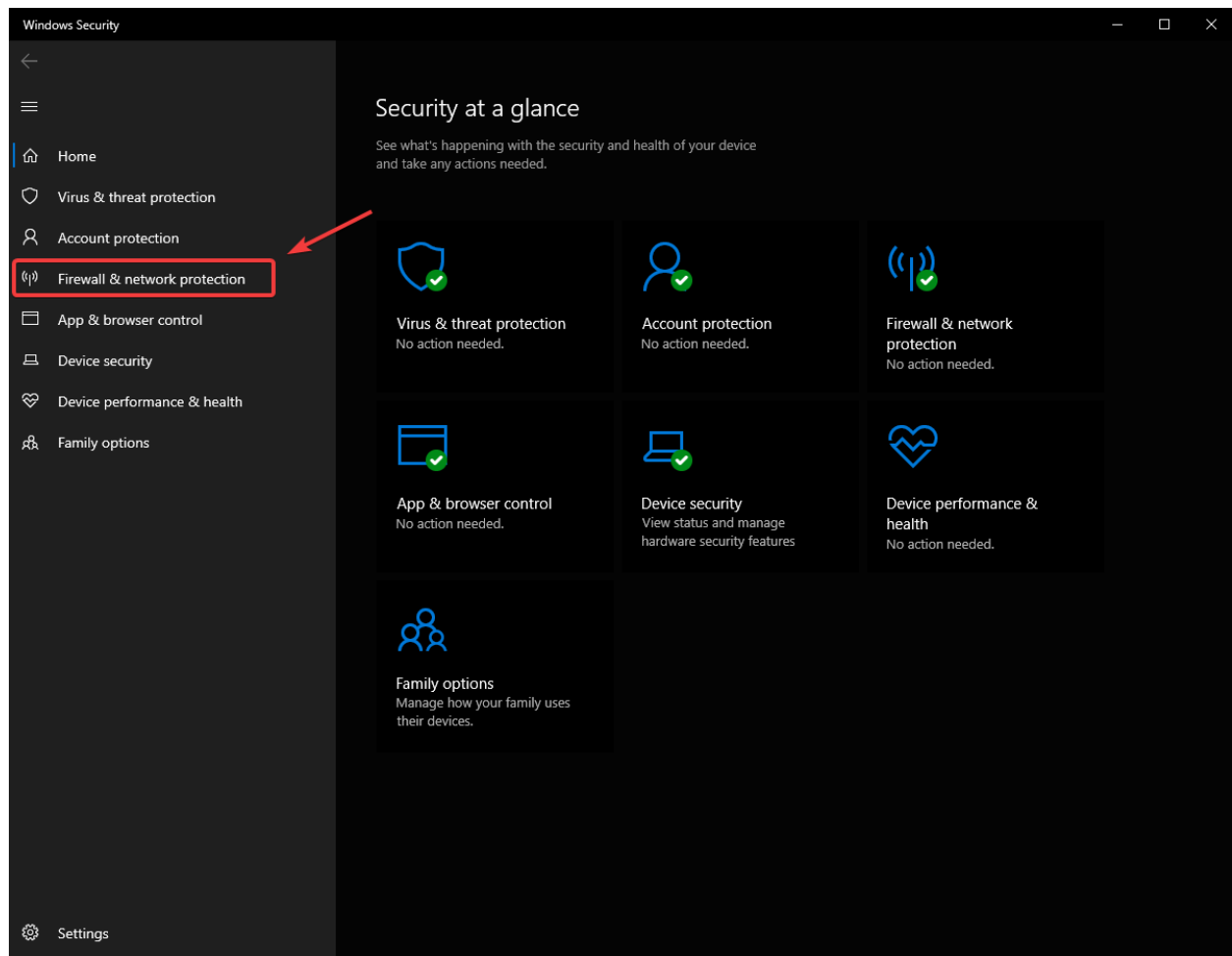
Click *Update & Security*



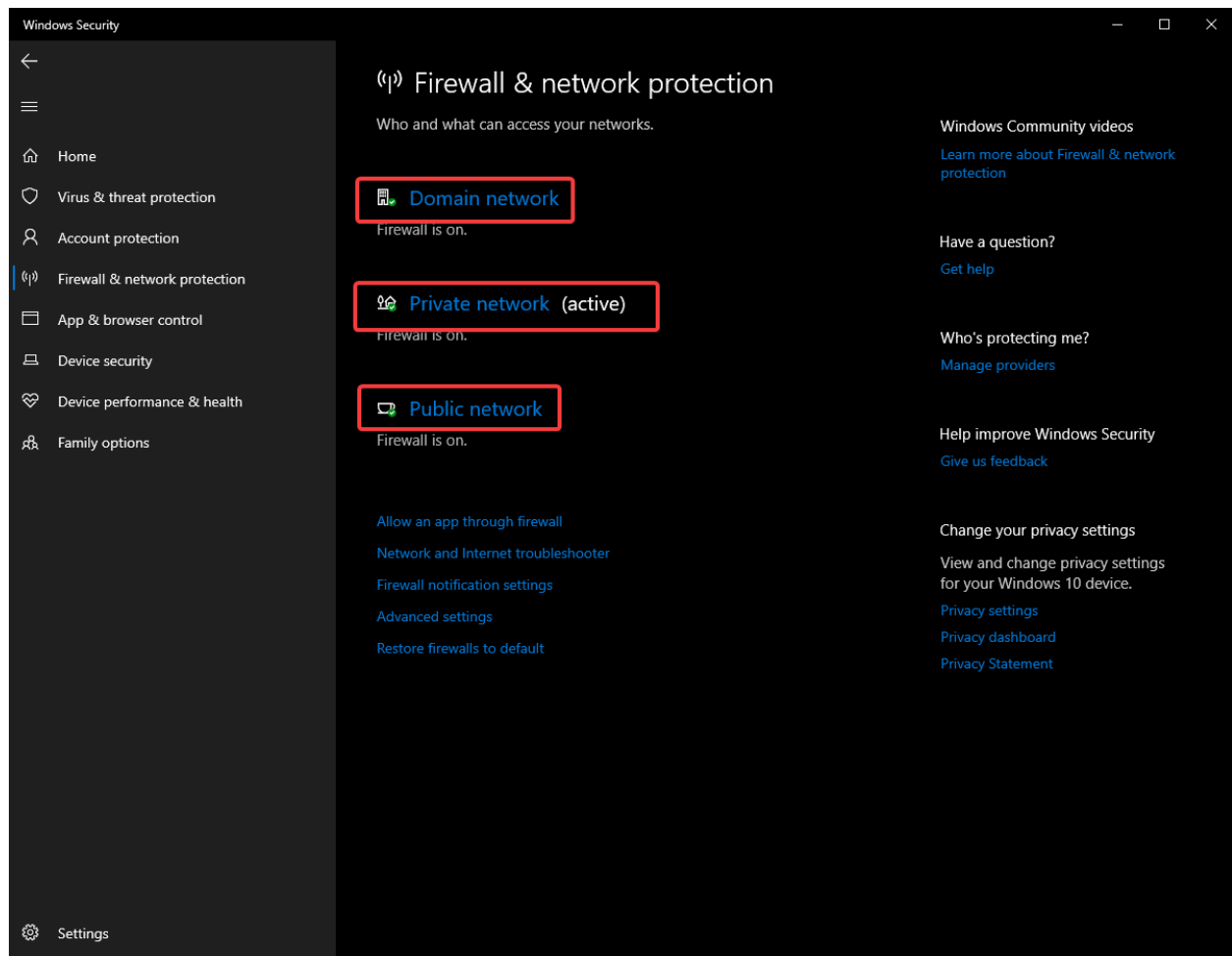
In the right pane, select *Open Windows Security*



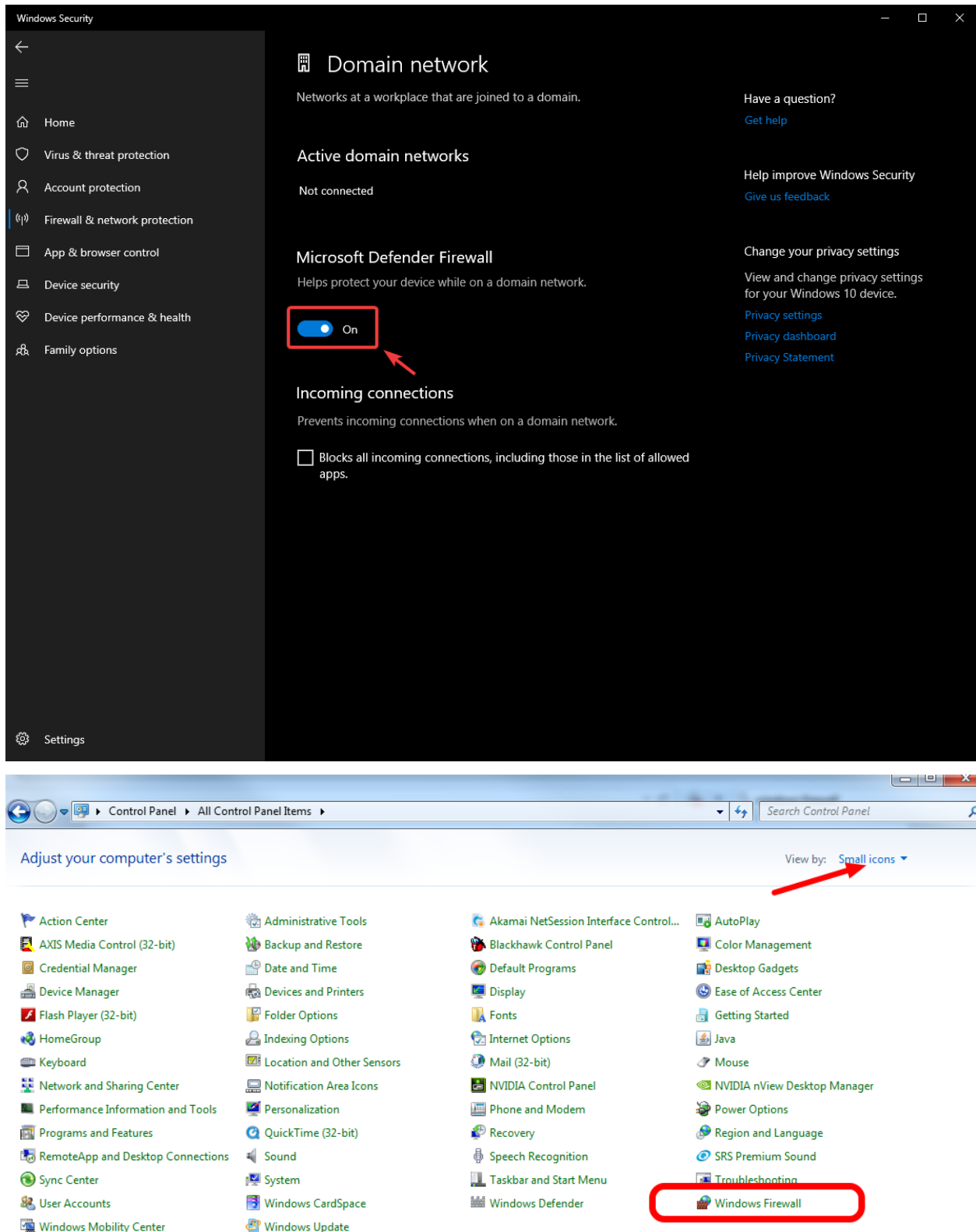
In the left pane, select *Firewall and network protection*



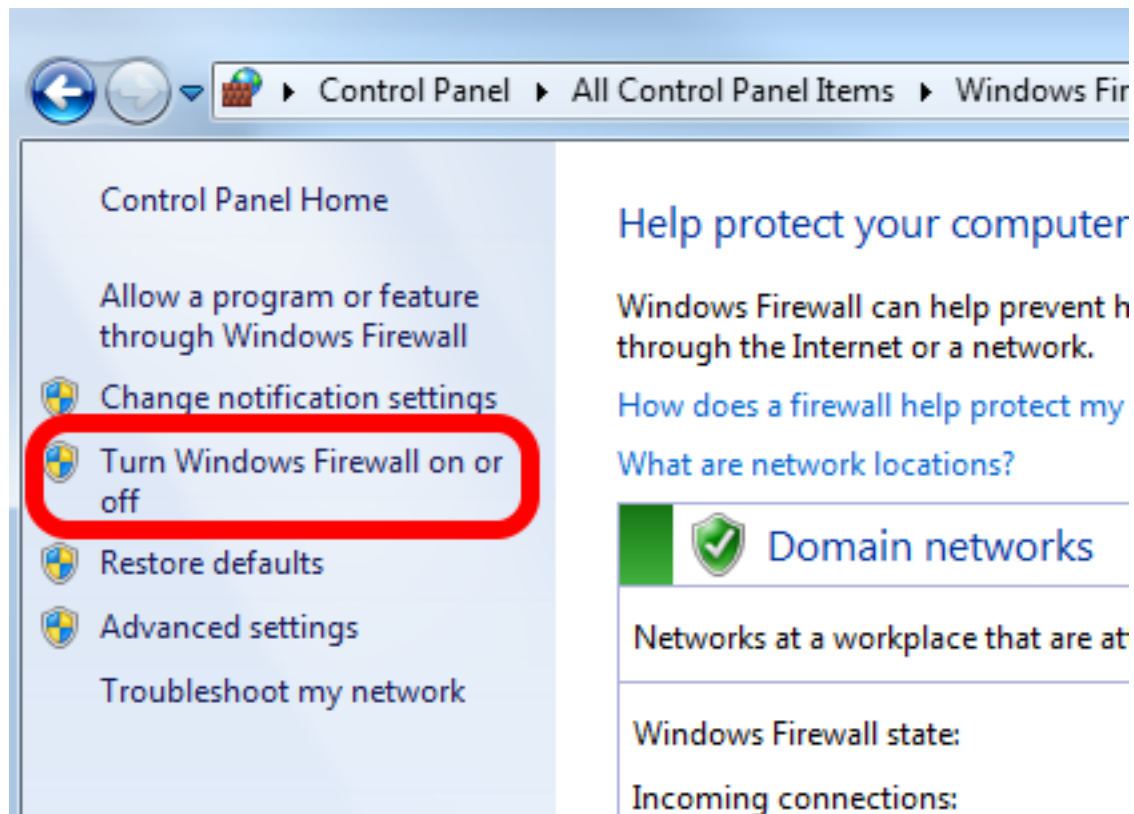
Click on **each** of the highlighted options



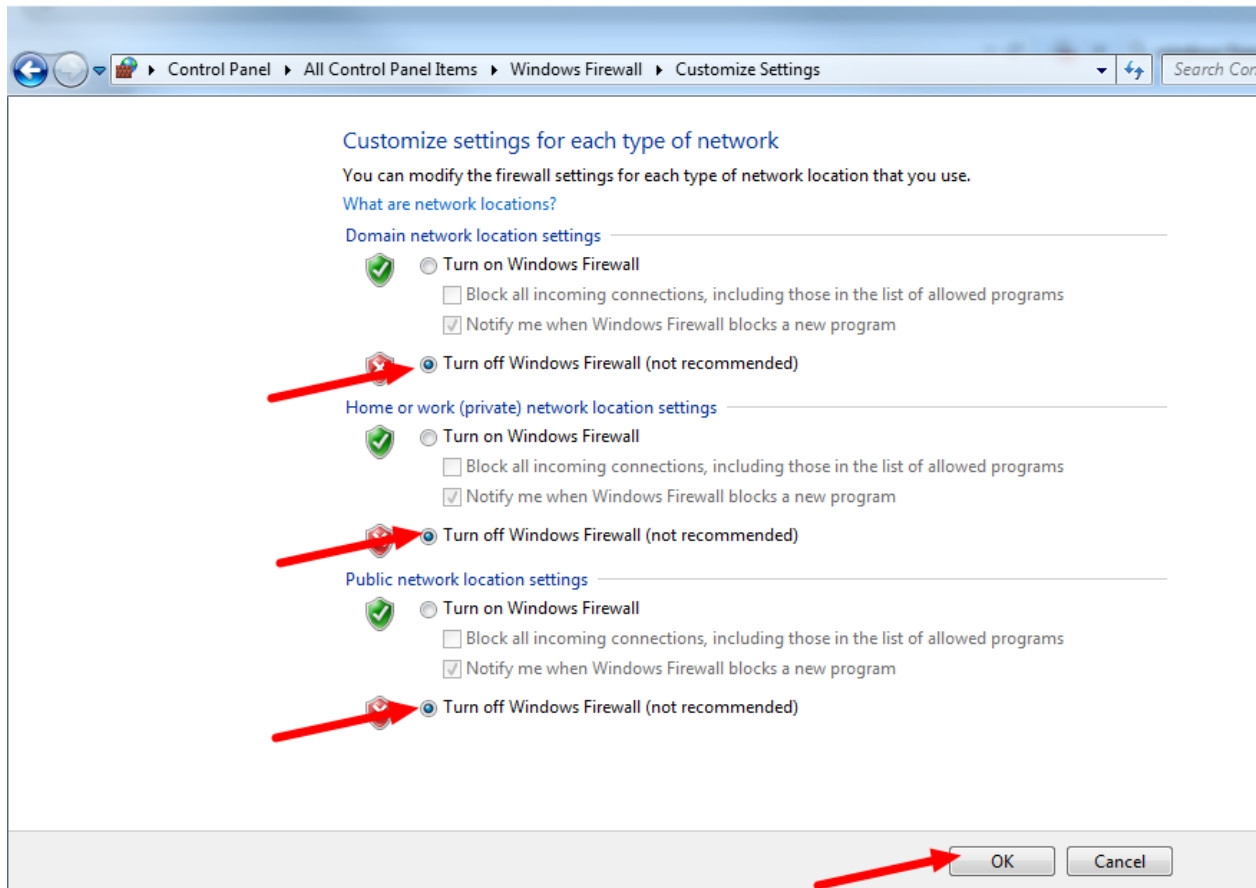
Then click on the **On** toggle to turn it off.



Click *Start* -> *Control Panel* to open the Control Panel. Click the dropdown next to *View by:* and select *Small icons* then click *Windows Firewall*.



In the left pane, click *Turn Windows Defender Firewall on or off*, and click yes. Enter your Administrator password if a dialog appears.



For each category, select the radio button to *Turn off Windows Defender Firewall*. Then click OK.

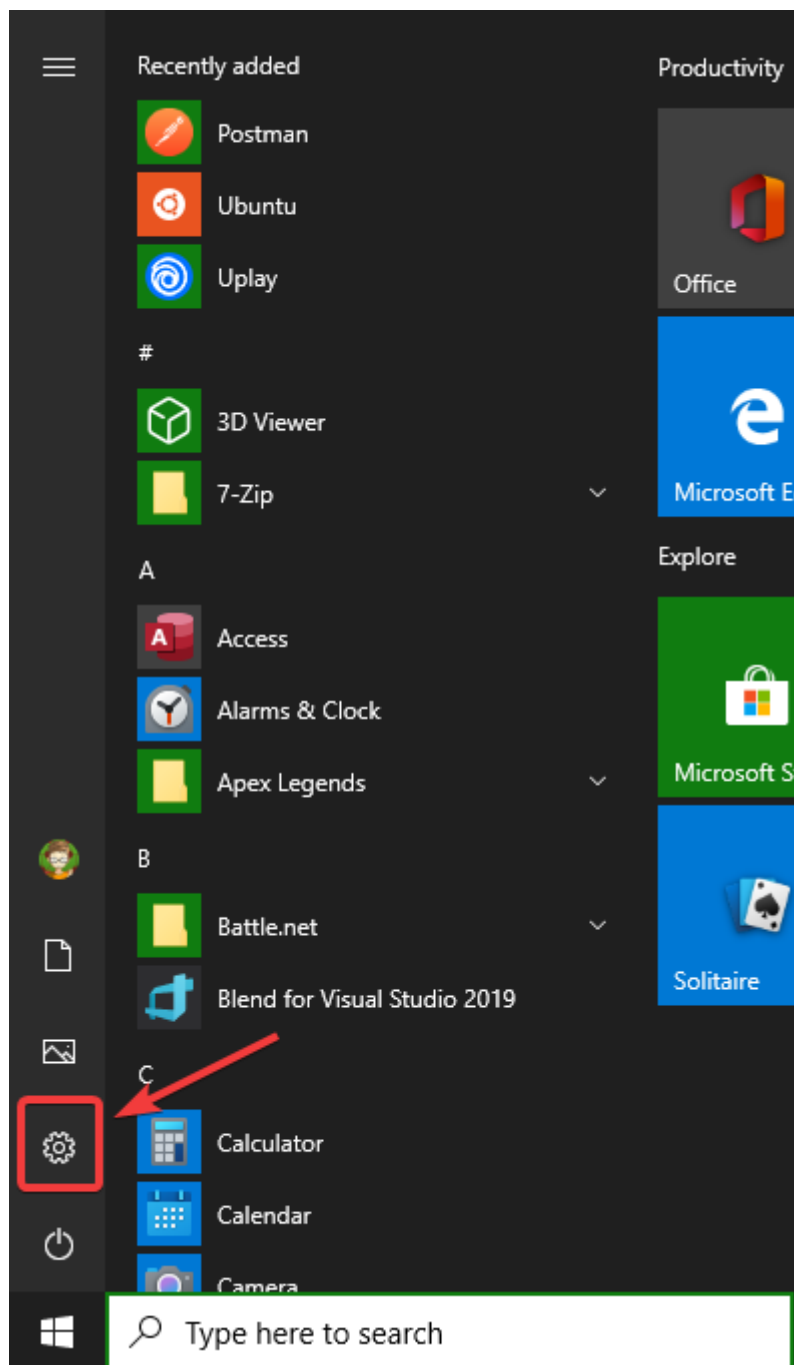
44.4.2 Whitelisting Apps

Alternatively, you can add exceptions to the Firewall for any FRC programs you are having issues with.

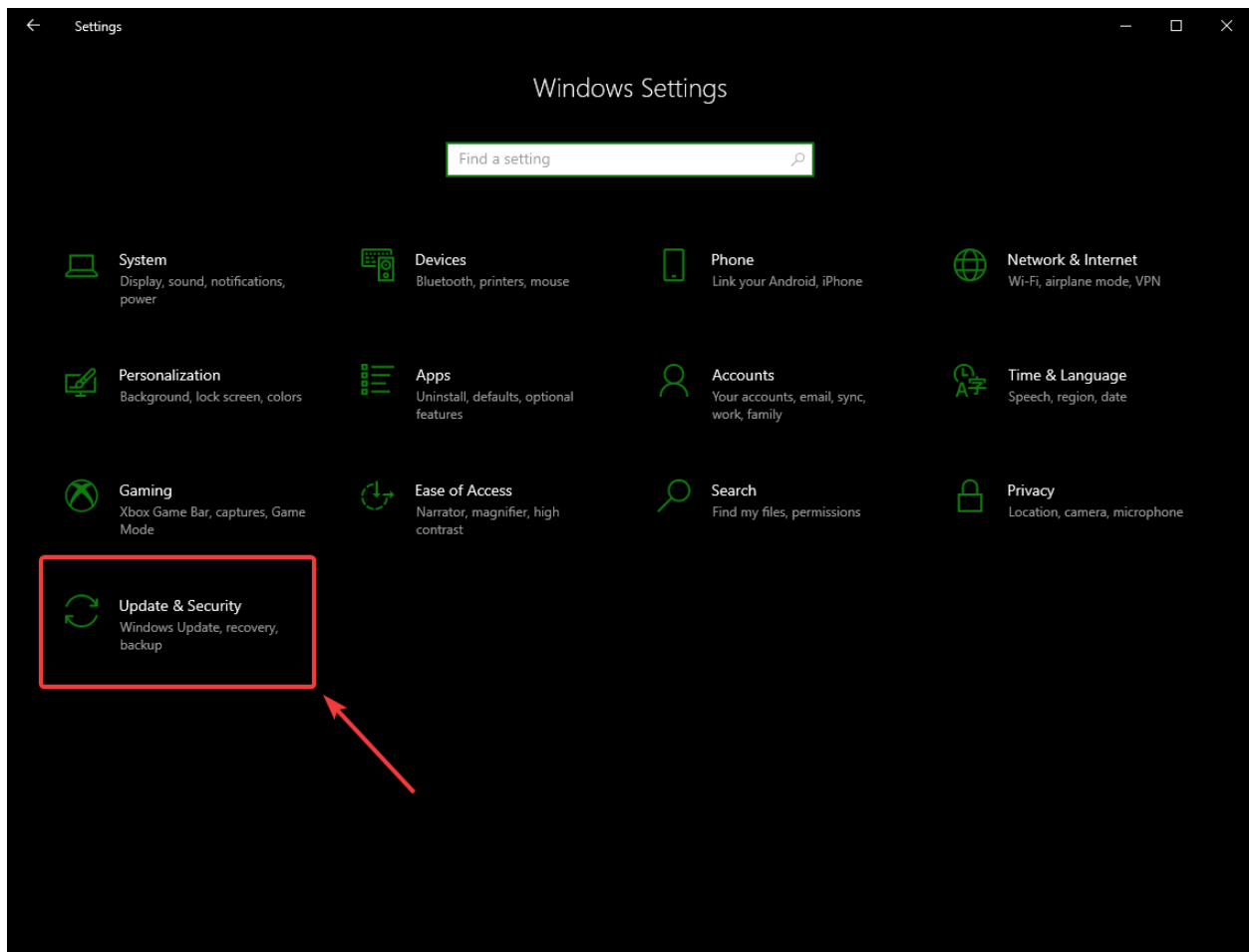
Windows 10

Windows 7

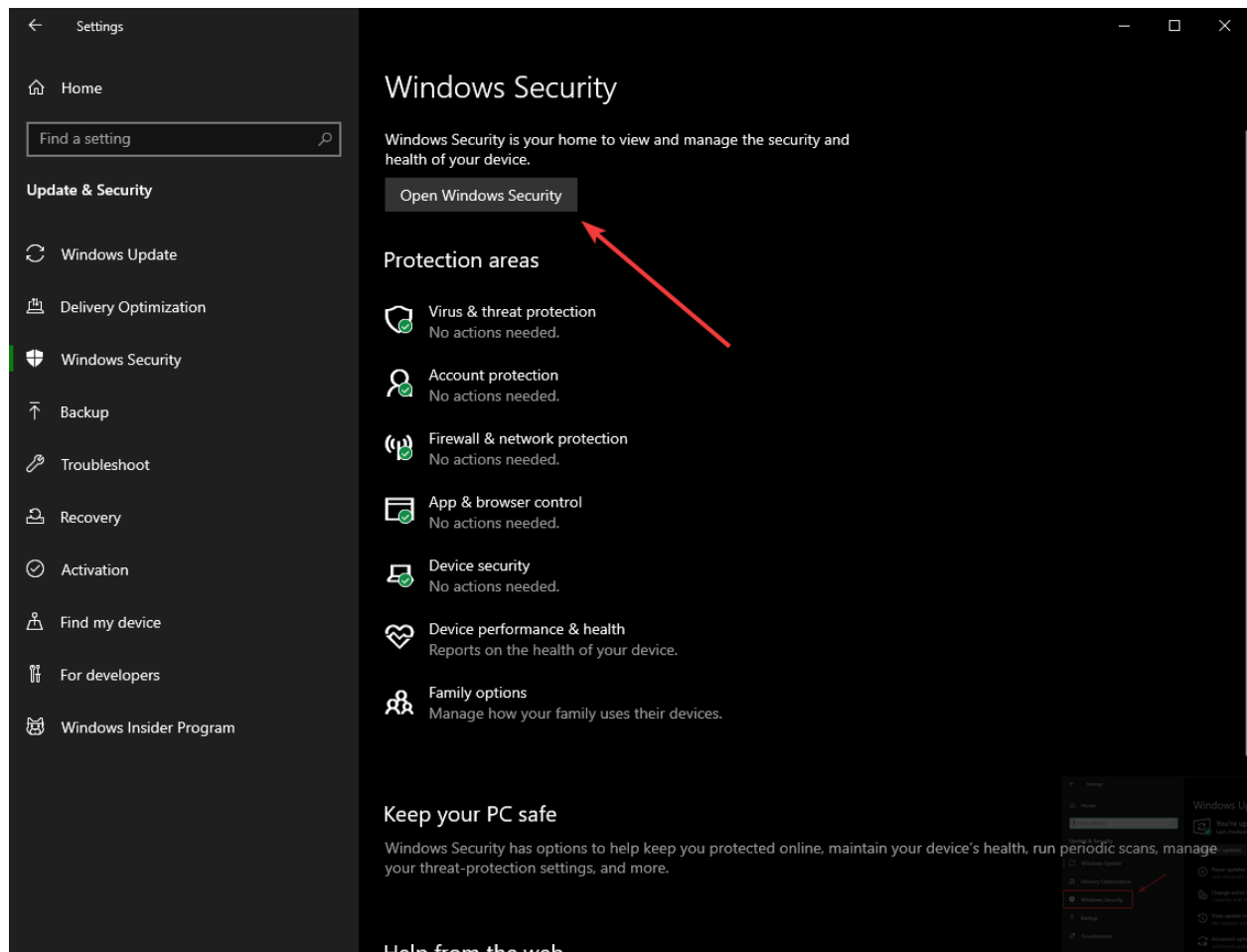
Click *Start -> Settings*



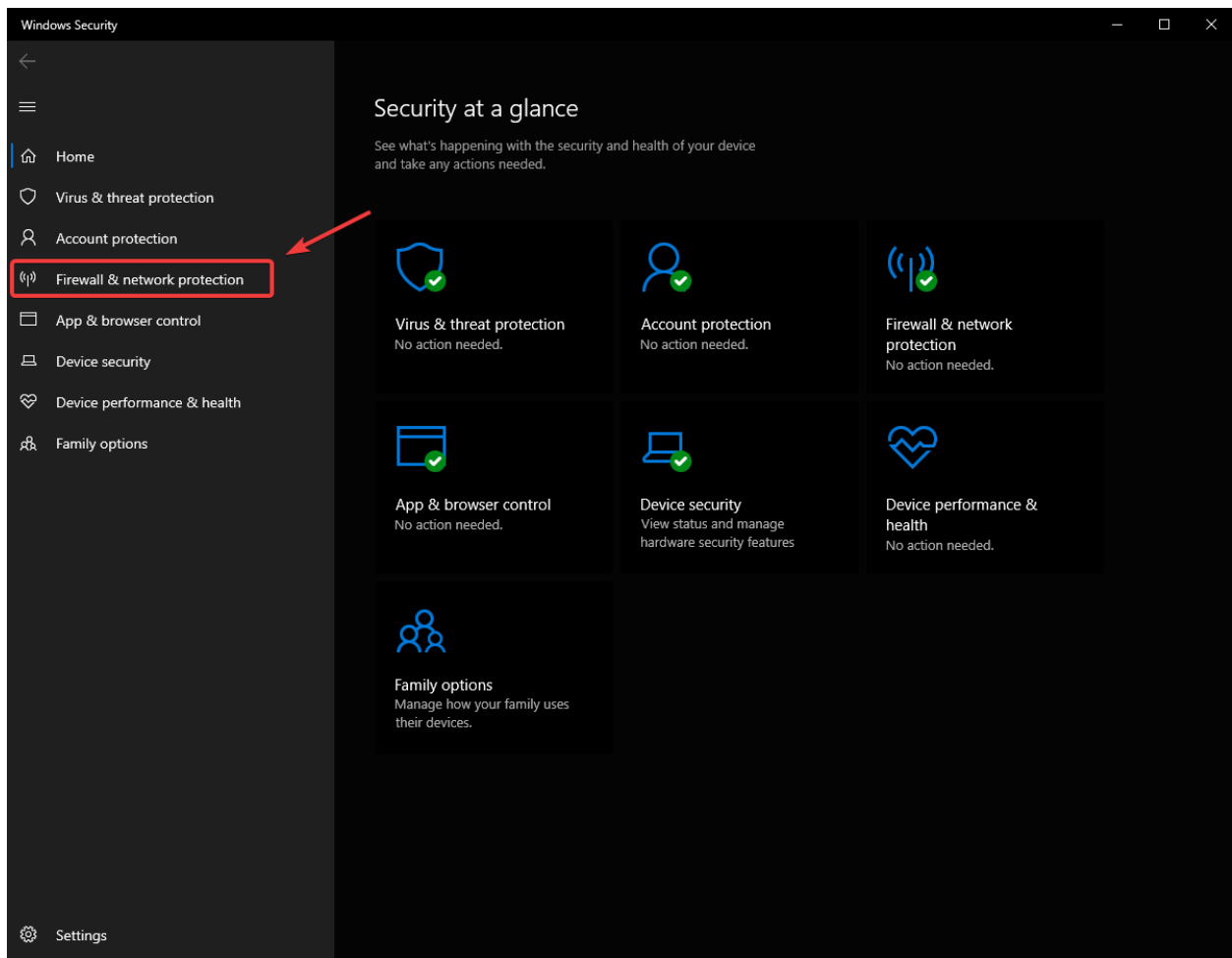
Click *Update & Security*



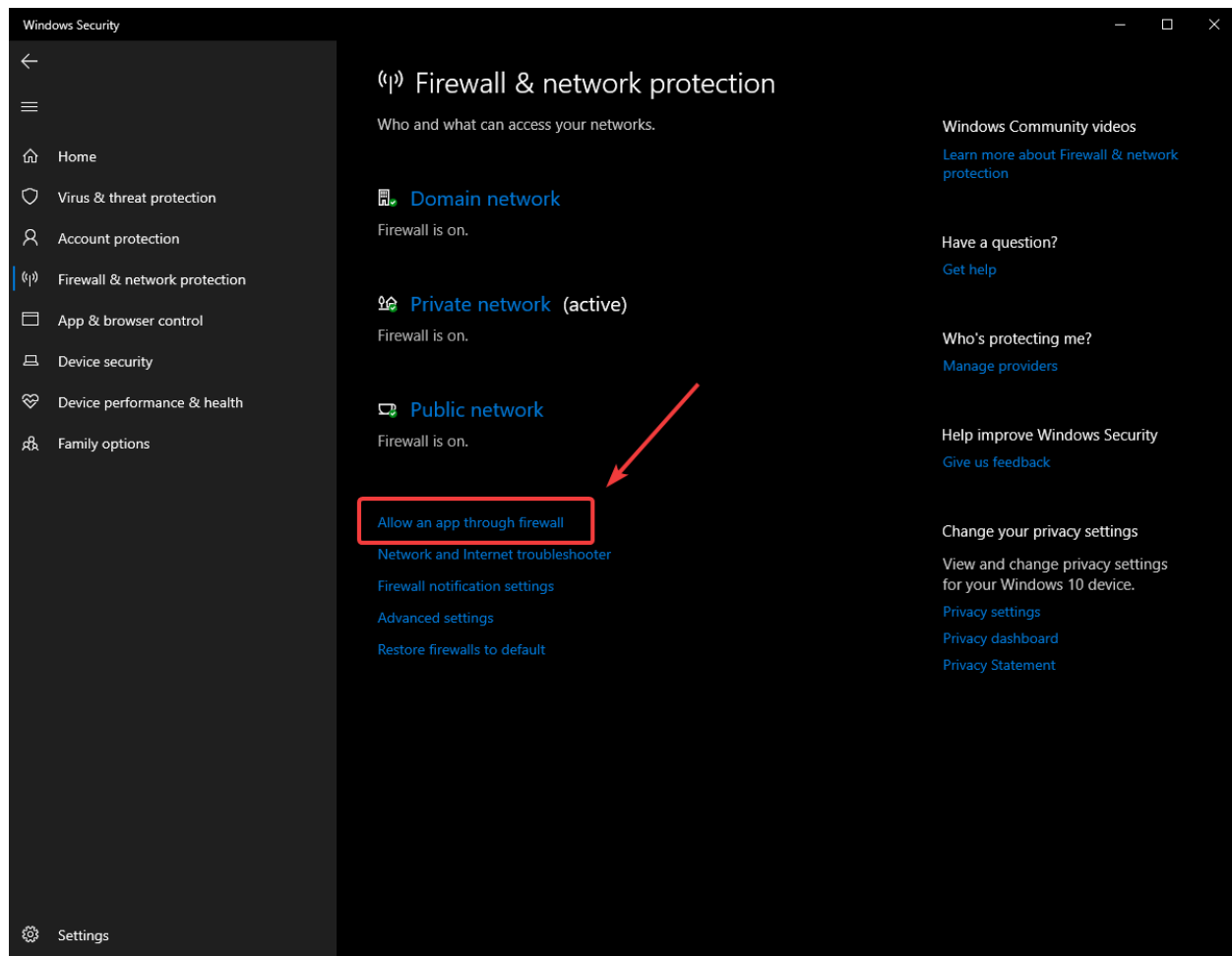
In the right pane, select *Open Windows Security*



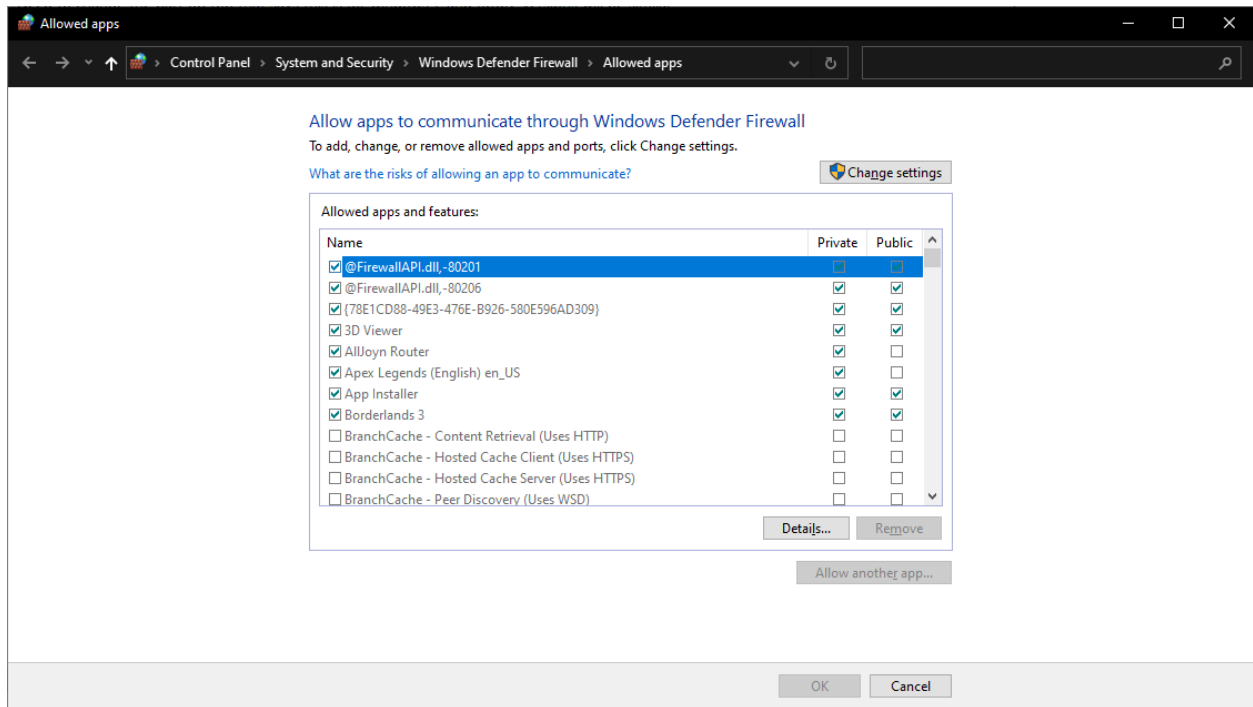
In the left pane, select *Firewall and network protection*



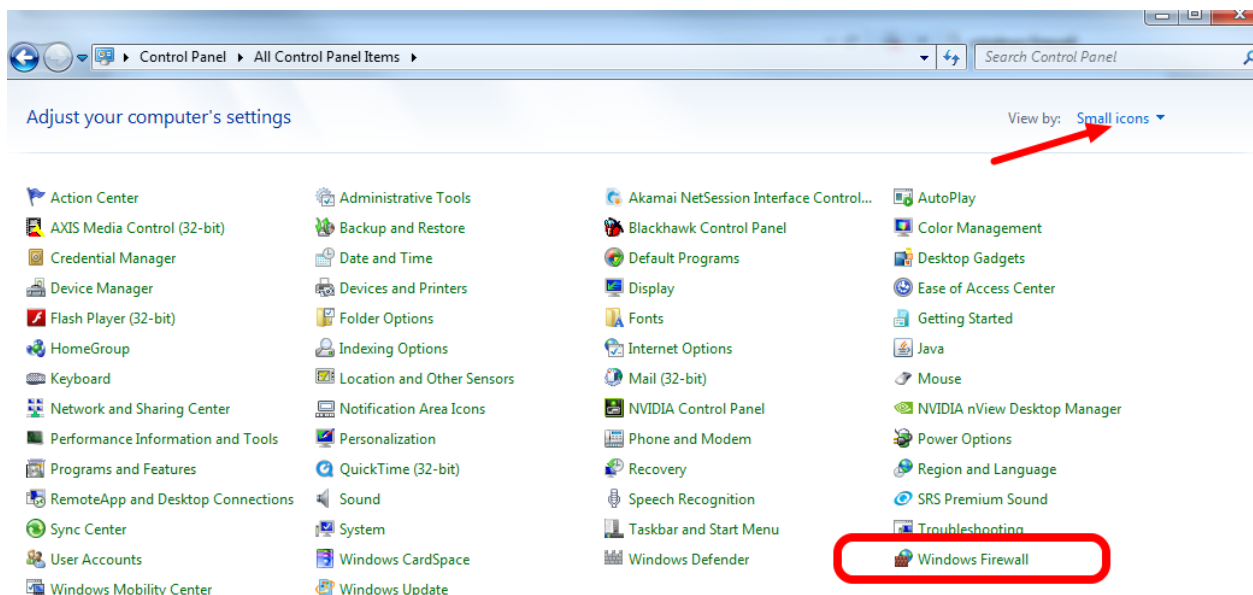
At the bottom of the window, select *Allow an app through firewall*



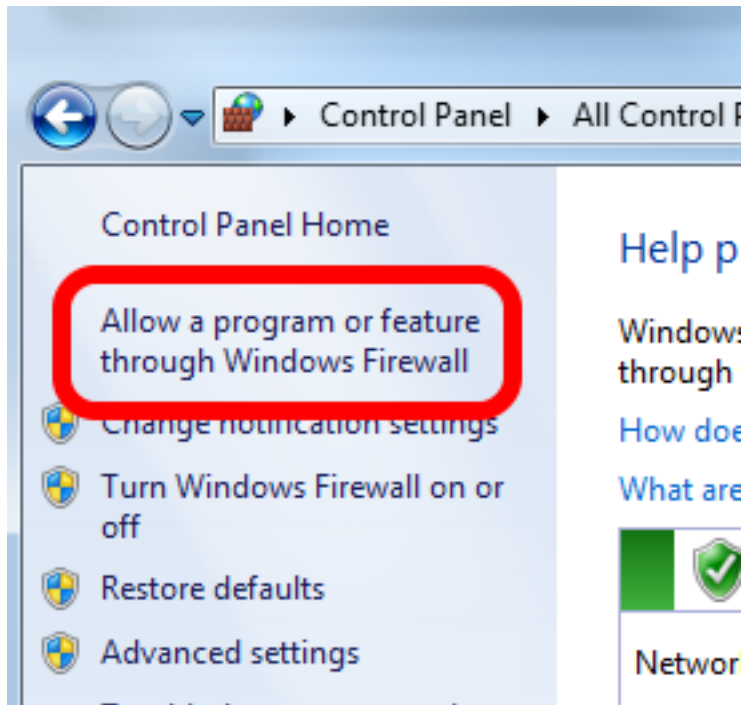
For each FRC program you are having an issue with, make sure that it appears in the list and that it has a check in each of the 3 columns. If you need to change a setting, you made need to click the *Change settings* button in the top right before changing the settings. If the program is not in the list at all, click the *Allow another program...* button and browse to the location of the program to add it.



Click *Start* -> *Control Panel* to open the Control Panel. Click the dropdown next to *View by:* and select *Small icons* then click *Windows Defender Firewall*.



In the left pane, click *Allow a program or feature through Windows Defender Firewall*



For each FRC program you are having an issue with, make sure that it appears in the list and that it has a check in each of the 3 columns. If you need to change a setting, you made need to click the *Change settings* button in the top right before changing the settings. If the program is not in the list at all, click the *Allow another program...* button and browse to the location of the program to add it.

Control Panel ▸ All Control Panel Items ▸ Windows Firewall ▸ Allowed Programs

Allow programs to communicate through Windows Firewall

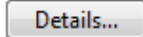
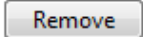
To add, change, or remove allowed programs and ports, click Change settings.

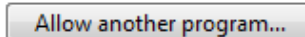
What are the risks of allowing a program to communicate?

 Change settings

Allowed programs and features:

Name	Domain	Home/Work (Private)	Public
<input checked="" type="checkbox"/> FRC Driver Station	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> FRC PC Dashboard	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> FTCounterMonitor.exe	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> FTCounterMonitor.exe	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/> FTCounterMonitor.exe	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/> FTSPVStudio.exe	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> FTSPVStudio.exe	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/> FTSPVStudio.exe	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/> FTSysDiagSvcHost.exe	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/> FTSysDiagSvcHost.exe	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/> FTSysDiagSvcHost.exe	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/> Google Chrome	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

 Details...  Remove

 Allow another program...

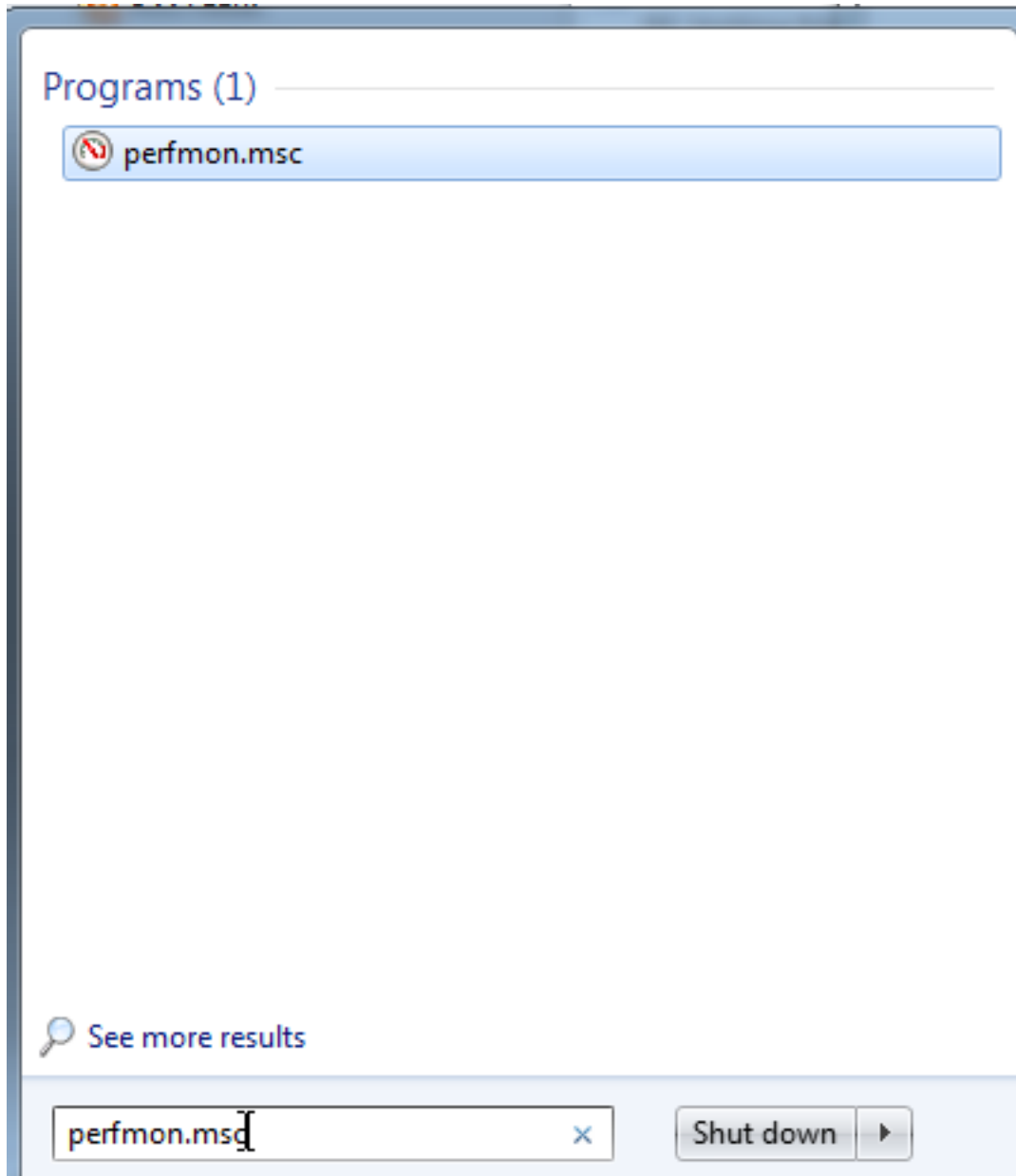
44.5 Measuring Bandwidth Usage

On the FRC® Field (and at home when the radio is configured using the FRC Bridge Configuration Utility) each team is limited to 4Mb/s of network traffic (see the [FMS Whitepaper](#) for more details). The FMS Whitepaper provides information on determining the bandwidth usage of the Axis camera, but some teams may wish to measure their overall bandwidth consumption. This document details how to make that measurement.

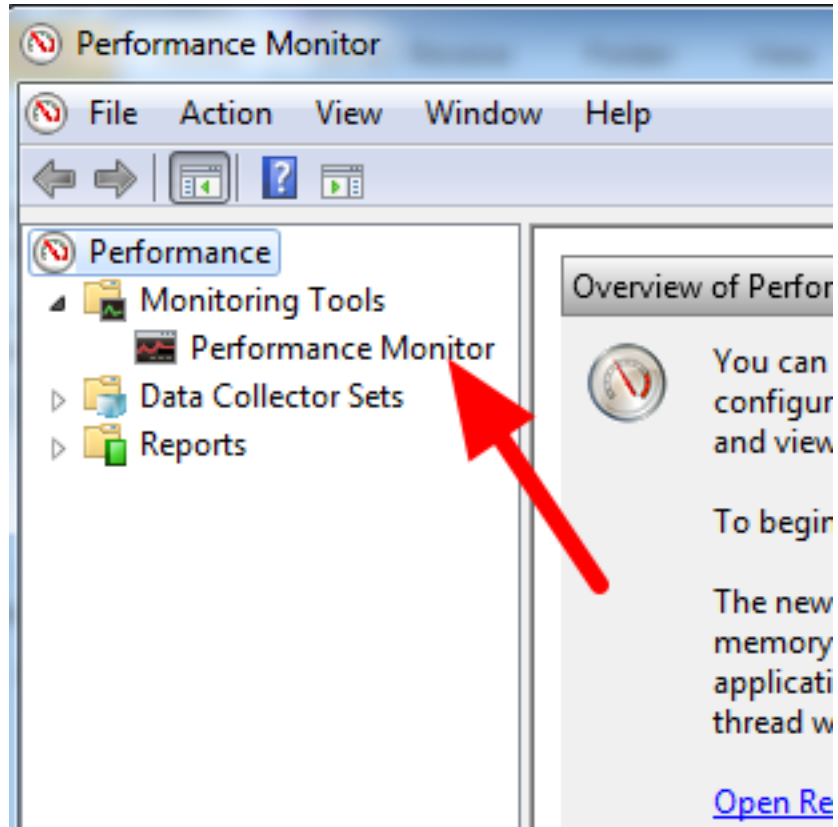
44.5.1 Measuring Bandwidth Using the Performance Monitor (Win 7 only)

Windows 7 contains a built-in tool called the Performance Monitor that can be used to monitor the bandwidth usage over a network interface.

Launching the Performance Monitor

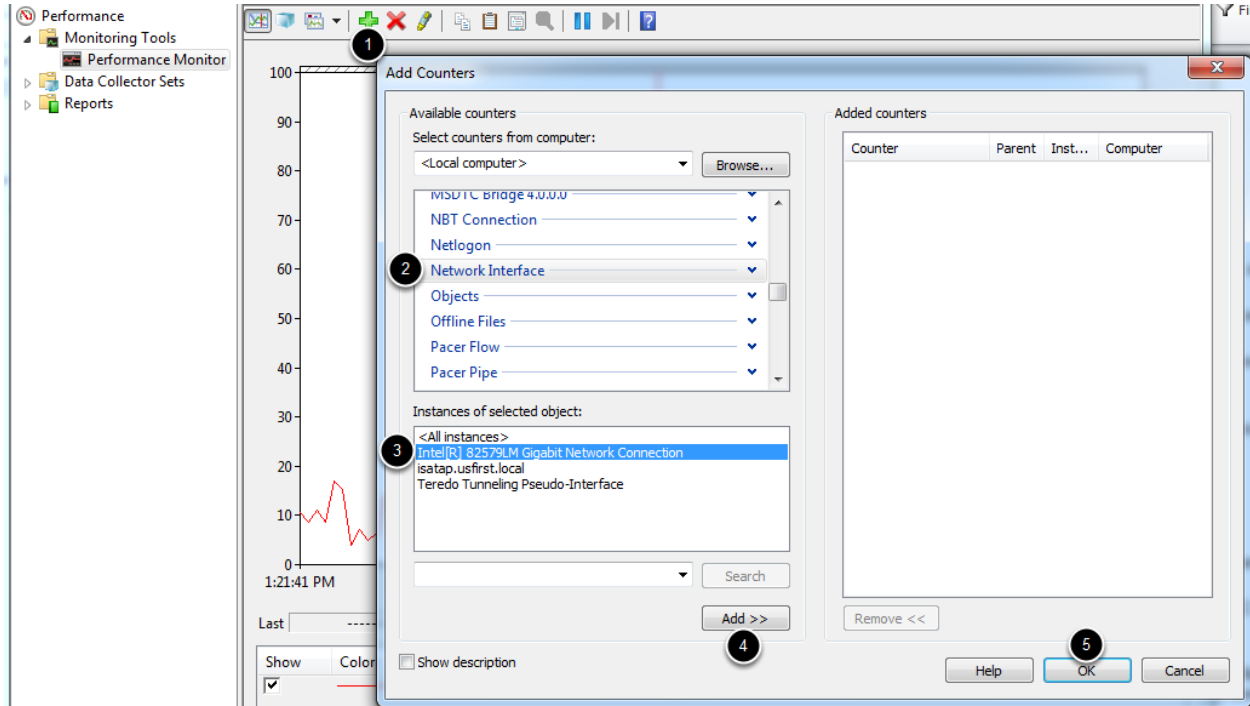


Click Start and in the search box, type `perfmon.msc` and press Enter.

Open Real-Time Monitor

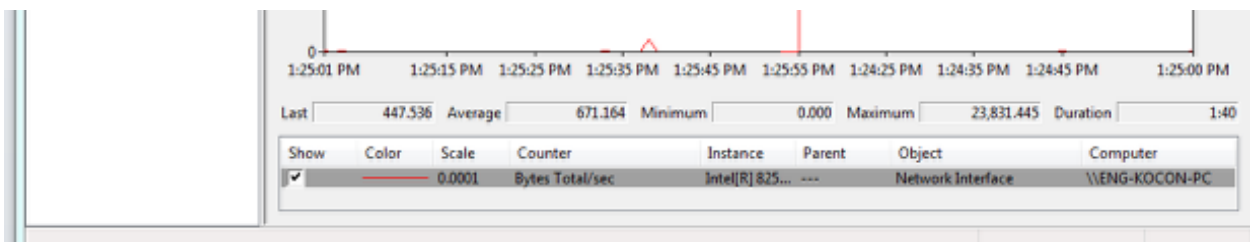
In the left pane, click Performance Monitor to display the real-time monitor.

Add Network Counter

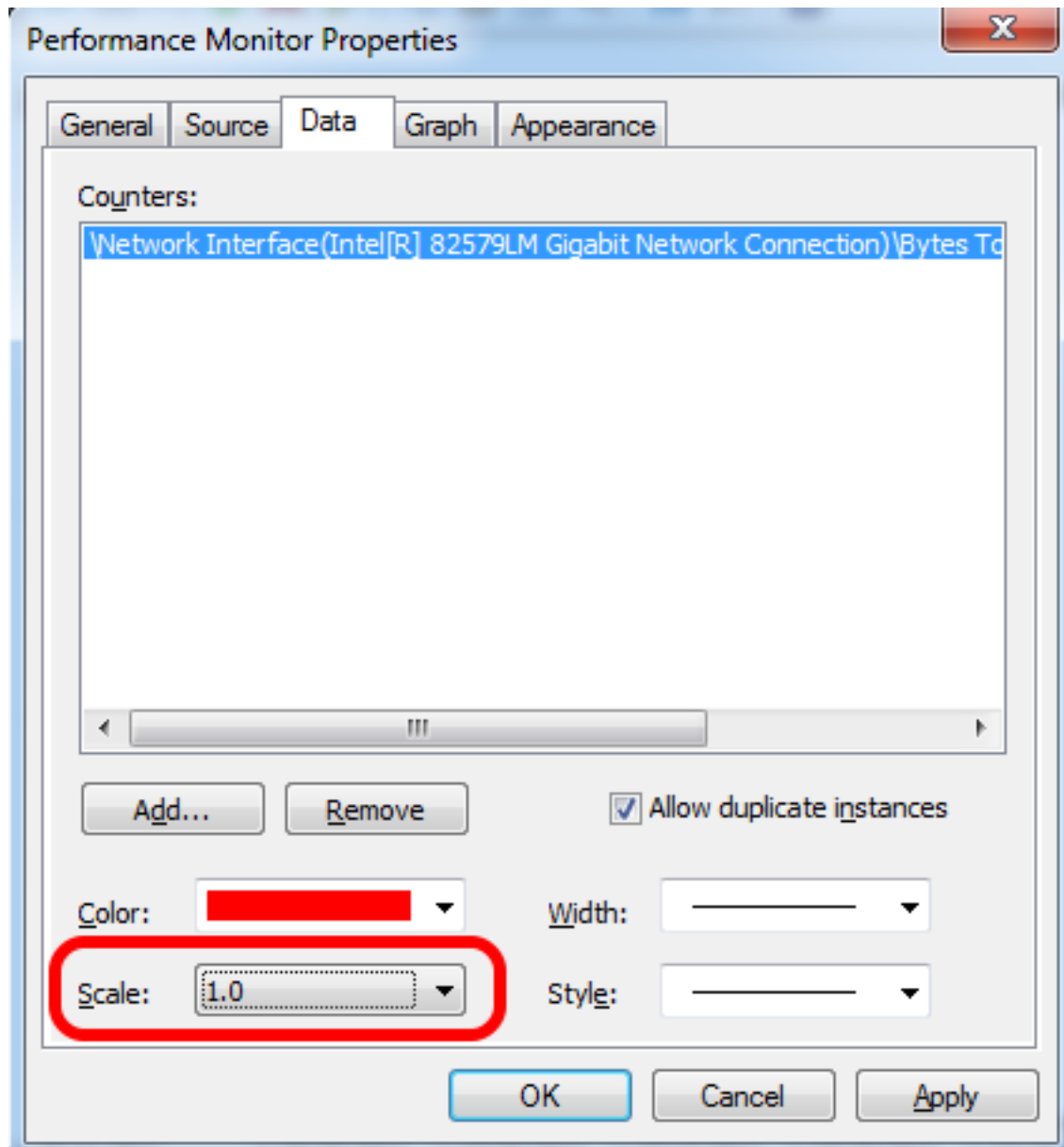


1. Click the green plus near the top of the screen to add a counter
2. In the top left pane, locate and click on Network Interface to select it
3. In the bottom left pane, locate the desired network interface (or use All instances to monitor all interfaces)
4. Click Add>> to add the counter to the right pane.
5. Click OK to add the counters to the graph.

Remove Extra Counters

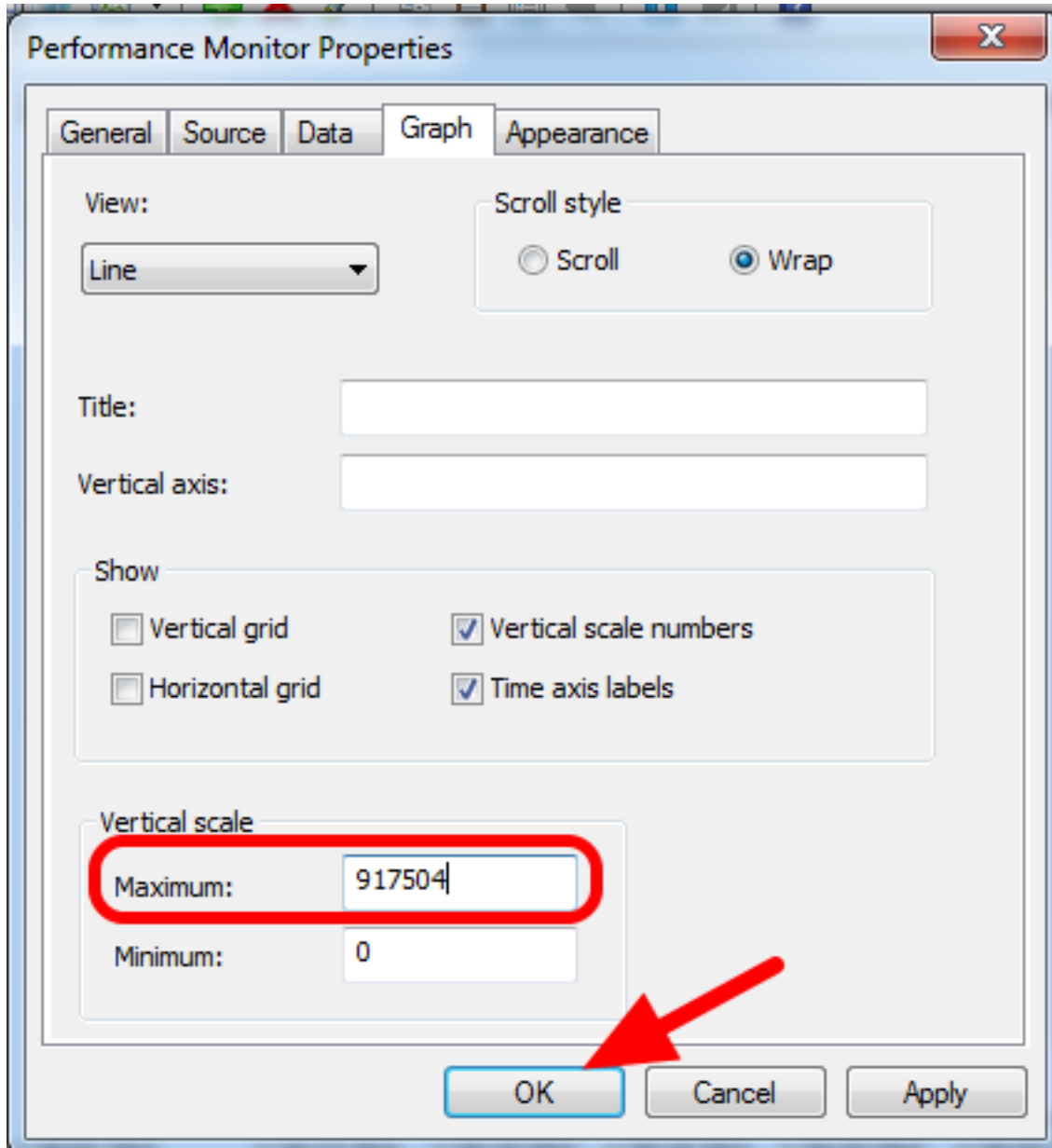


In the bottom pane, select each counter other than Bytes Total/sec and press the Delete key. The Bytes Total/sec entry should be the only entry remaining in the pane.

Configure Data Properties

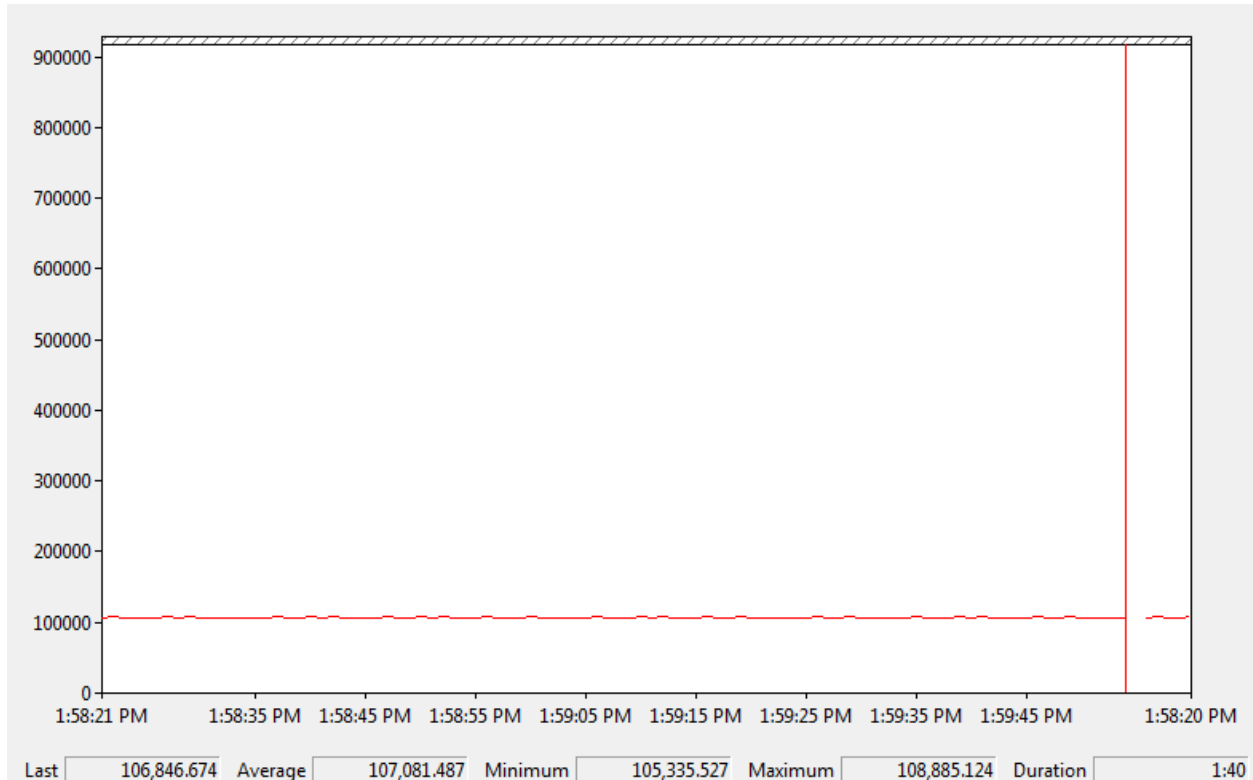
Press Ctrl+Q to bring up the Properties window. Click on the dropdown next to Scale and select 1.0. Then click on the Graph tab.

Configure Graph Properties



In the Maximum Box under Vertical Scale enter 917504 (this is 7 Megabits converted to Bytes). If desired, turn on the horizontal grid by checking the box. Then click OK to close the dialog.

Viewing Bandwidth Usage

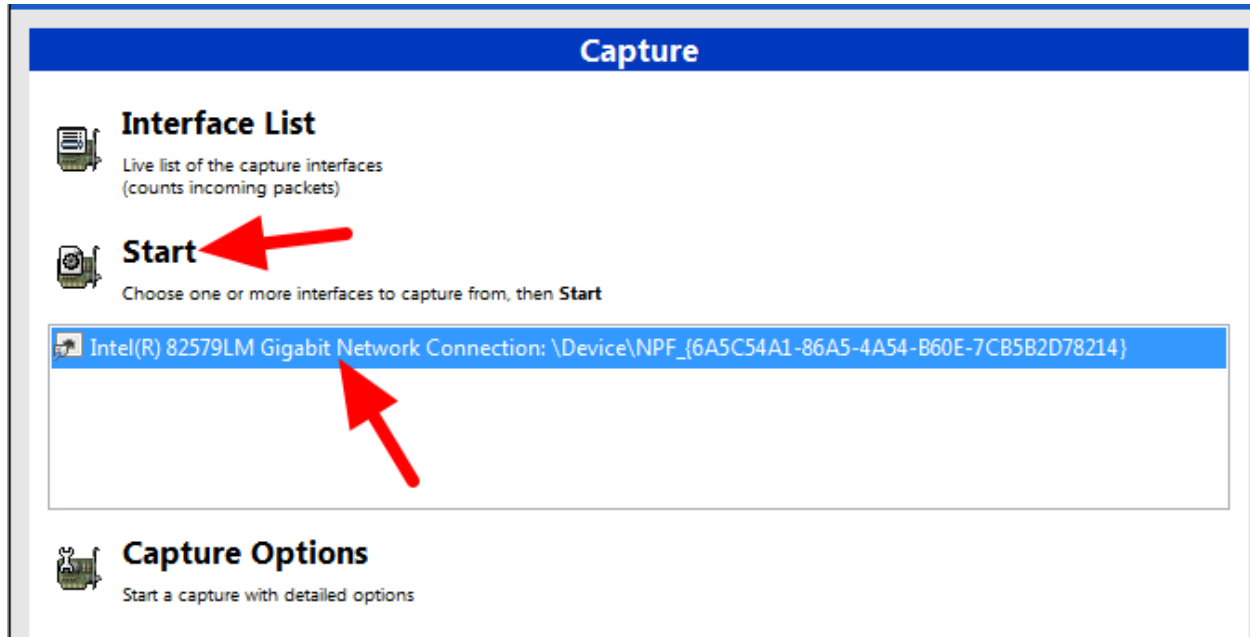


You may now connect to your robot as normal over the selected interface (if you haven't done so already). The graph will show the total bandwidth usage of the connection, with the bandwidth cap at the top of the graph. The Last, Average, Min and Max values are also displayed at the bottom of the graph. Note that these values are in Bytes/Second meaning the cap is 917,504. With just the Driver Station open you should see a flat line at ~100000 Bytes/Second.

44.5.2 Measuring Bandwidth Usage using Wireshark

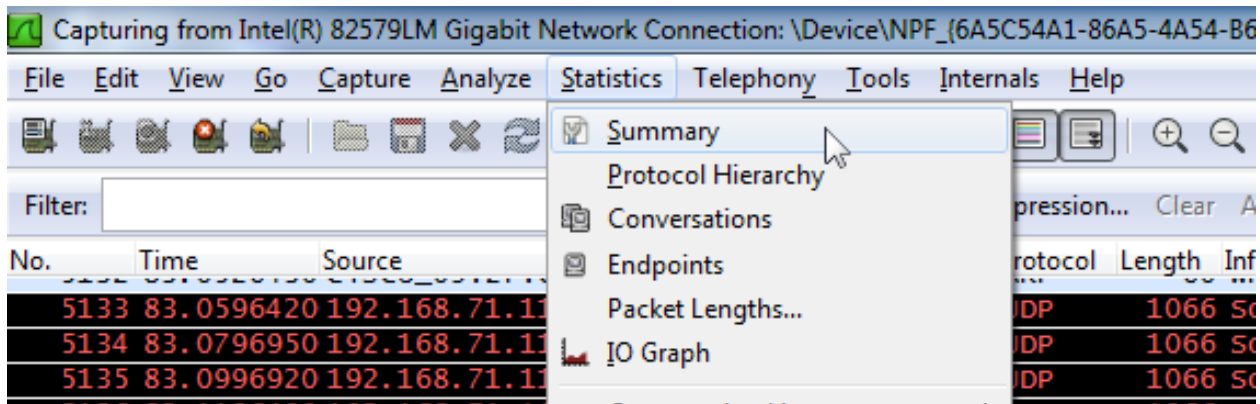
If you are not using Windows 7, you will need to install a 3rd party program to monitor bandwidth usage. One program that can be used for this purpose is Wireshark. Download and install the latest version of Wireshark for your version of Windows. After installation is complete, locate and open Wireshark. Connect your computer to your robot, open the Driver Station and any Dashboard or custom programs you may be using.

Select the interface and Start capture



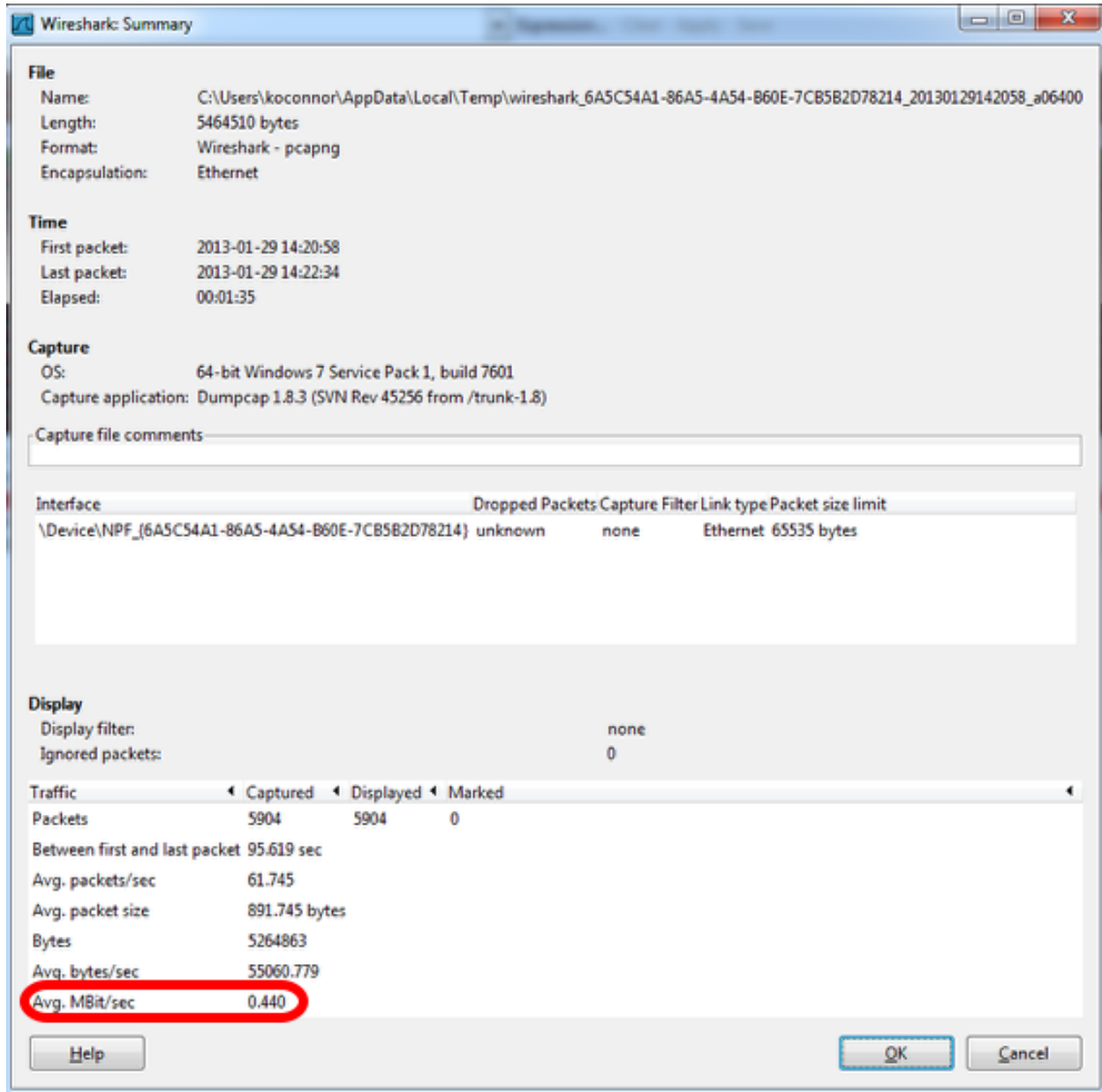
In the Wireshark program on the left side, select the interface you are using to connect to the robot and click Start.

Open Statistics Summary



Let the capture run for at least 1 minute, then click Statistics then Summary.

View Bandwidth Usage



Average bandwidth usage, in Megabits/Second is displayed near the bottom of the summary window.

44.6 OM5P-AC Radio Modification

The intended use case for the OM5P-AC radio does not subject it to the same shocks and forces as it sees in the FRC® environment. If the radio is subjected to significant pressure on the bottom of the case, it is possible to cause a radio reboot by shorting a metal shield at the bottom of the radio to some exposed metal leads on the bottom of the board. This article details a modification to the radio to prevent this scenario.

Warning: It takes significant pressure applied to the bottom of the case to cause a reboot in this manner. Most FRC radio reboot issues can be traced to the power path in some form. We recommend mitigating this risk via strategic mounting of the radio rather than opening and modifying the radio (and risk damaging delicate internal components):

- Avoid using the “mounting tab” features on the bottom of the radio.
- You may wish to mount the radio to allow for some shock absorption. A little can go a long way, mounting the radio using hook and loop fastener or to a robot surface with a small amount of flex (plastic or sheet metal sheet, etc.) can significantly reduce the forces experienced by the radio.

44.6.1 Opening the Radio

Note: The OpenMesh OM5P-AC is not designed to be a user serviceable device. Users perform this modification at their own risk. Make sure to work slowly and carefully to avoid damaging internal components such as radio antenna cables.

Case Screws





Locate the two rubber feet on the front side of the radio then pry them off the radio using fingernails, small flat screwdriver, etc. Using a small Phillips screwdriver, remove the two screws under the feet.

Side Latches



There is a small latch on the lid of the radio near the middle of each long edge (you can see these latches more clearly in the next picture). Using a fingernail or very thin tool, slide along the gap between the lid and case from front to back towards the middle of the radio, you should hear a small pop as you near the middle of radio. Repeat on the other side (note: it's not hard to accidentally re-latch the first side while doing this, make sure both sides are unlatched before proceeding). The radio lid should now be slightly open on the front side as shown in the image above.

Remove Lid

Warning: The board may stick to the lid as you remove it due to the heatsink pads. Look through the vents of the radio as you remove the lid to see if the board is coming with it, if it is you may need to insert a small tool to hold the board down to separate it from the lid. We recommend a small screwdriver or similar tool that fits through the vents, applied through the front corner on the barrel jack side, right above the screw hole. You can scroll down to the picture with the lid removed to see what the board looks like in this area.

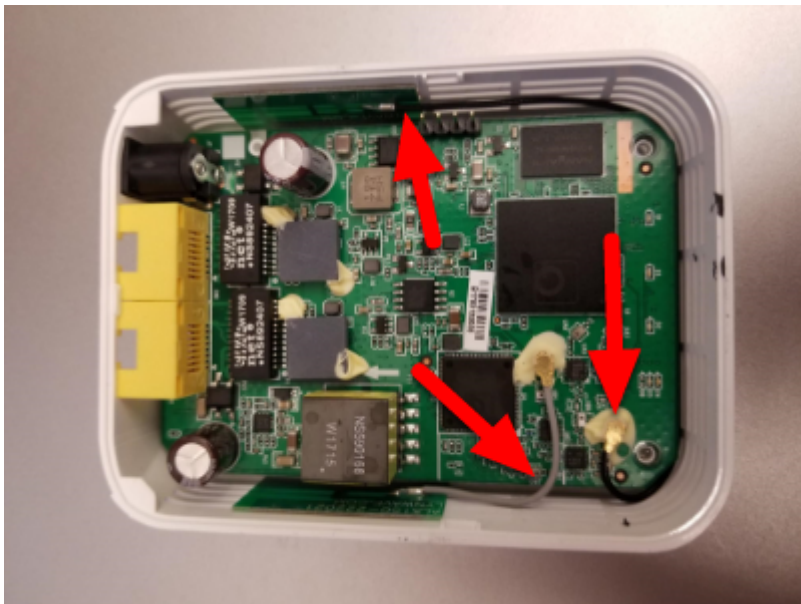


To begin removing the lid, slide it forward (lifting slightly) until the screw holders hit the case front (you may need to apply pressure on the latch areas while doing this).

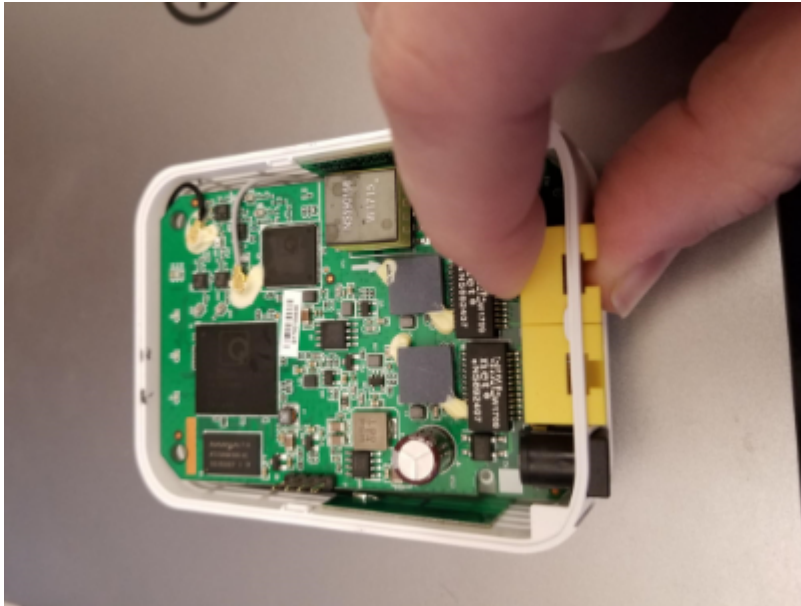


Next, begin rotating the lid slightly away from the barrel jack side, as shown while continuing to lift. This will unhook the lid from the small triangle visible in the top right corner. Continue to rotate slightly in this direction while pushing the top left corner towards the barrel jack (don't try to lift further in this step) to unhook a similar feature in the top left corner. Then lift the lid completely away from the body.

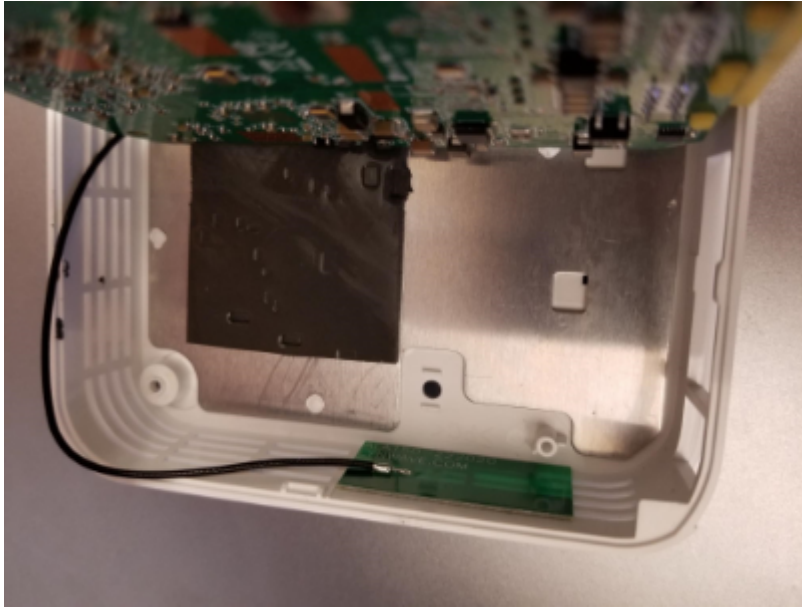
Remove Board



Warning: Note the antenna wires shown in the image above. These wires, and their connectors, are fragile, take care not to damage them while performing the next steps.



To remove the board, we recommend grasping one or both network ports with your fingers (as shown) and pushing inward (toward the front of the radio) and upward until the network ports and barrel jack are free from the case.



Tilt the board up (towards the short grey antenna cable) to expose the metal shield underneath.

Note: When you perform this step, you may notice that there is a small reset button on the underside of the board that is larger than the hole in the case. Note that pressing the reset button with the FRC firmware installed has no effect and that drilling the case of the radio is not a permitted modification.

44.6.2 Apply Tape



Apply a piece of electrical tape to the metal shield in the area just inside of the network port/barrel jack openings. This will prevent the exposed leads on the underside of the board

from short circuiting on this plate.

44.6.3 Re-assemble Radio

Re-assemble the radio by reversing the instructions to open it:

- Lay the board back down, making sure it aligns with the screw holes near the front and seats securely
- Slide the lid onto the back left retaining feature by moving it in from right to left. Take care of the capacitor in this area
- Rotate the lid, press downwards and slide the back right retaining feature in
- Press down firmly on the front/middle of the lid to seat the latches
- Replace 2 screws in front feet
- Replace front feet

45.1 Port Forwarding

This class provides an easy way to forward local ports to another host/port. This is useful to provide a way to access Ethernet-connected devices from a computer tethered to the roboRIO USB port. This class acts as a raw TCP port forwarder, this means you can forward connections such as SSH.

45.1.1 Forwarding a Remote Port

Often teams may wish to connect directly to the roboRIO for controlling their robot. The PortForwarding class (Java, C++) can be used to forward the Raspberry Pi connection for usage during these times. The PortForwarding class establishes a bridge between the remote and the client. To forward a port in Java, simply do `PortForwarder.add(int port, String remoteName, int remotePort)`.

Java

C++

```
@Override
public void robotInit() {
    PortForwarder.add(8888, "wpilibpi.local", 80);
}
```

```
void Robot::RobotInit {
    wpi::PortForwarder::GetInstance().Add(8888, "wpilibpi.local", 80);
}
```

Important: You **can not** use a port less than 1024 as your local forwarded port. It is also important to note that you **can not** use full URLs (`http://wpilibpi.local`) and should only use IP Addresses or DNS names.

45.1.2 Removing a Forwarded Port

To stop forwarding on a specified port, simply call `remove(int port)` with `port` being the port number. If you call `remove()` on a port that is not being forwarded, nothing will happen.

Java

C++

```
@Override
public void robotInit() {
    PortForwarder.remove(8888);
}
```

```
void Robot::RobotInit {
    wpi::PortForwarder::GetInstance().Remove(8888);
}
```

Contributing to frc-docs

46.1 Contribution Guidelines

Welcome to the contribution guidelines for the frc®-docs project. If you are unfamiliar to writing in the reStructuredText format, please read up on it [here](#).

Important: *FIRST*® retains all rights to documentation and images provided. Credit for articles/updates will be in the [GitHub commit history](#).

46.1.1 Mission Statement

The WPILib Mission is to enable *FIRST* Robotics teams to focus on writing game-specific software rather than focusing on hardware details - “raise the floor, don’t lower the ceiling”. We work to enable teams with limited programming knowledge and/or mentor experience to be as successful as possible, while not hampering the abilities of teams with more advanced programming capabilities. We support Kit of Parts control system components directly in the library. We also strive to keep parity between major features of each language (Java, C++, and NI’s LabVIEW), so that teams aren’t at a disadvantage for choosing a specific programming language.

These docs serve to provide a learning ground for all *FIRST* Robotics Competition teams. Contributions to the project must follow these core principles.

- Community-led documentation. Documentation sources are hosted publicly and the community are able to make contributions
- Structured, well-formatted, clean documentation. Documentation should be clean and easy to read, from both a source and release standpoint
- Relevant. Documentation should be focused on the *FIRST* Robotics Competition.

Please see the [Style Guide](#) for information on styling your documentation.

46.1.2 Release Process

frc-docs uses a special release process for handling the main site `/stable/` and the development site `/latest/`. This flow is detailed below.

During Season: - Commit made to main branch

- Updates `/stable/` and `/latest/` on the website

End of Season: - Repository is tagged with year, for archival purposes

Off-Season: - `stable` branch is locked to the last on-season commit - Commit made to main branch

- Only updates `/latest/` on the documentation site

46.1.3 Creating a PR

PRs should be made to the [frc-docs](#) repo on GitHub. They should point to the `main` branch and *not* `stable`.

46.1.4 Creating New Content

Thanks for contributing to the [frc-docs](#) project! There are a couple things you should know before getting started!

Where to place articles?

The location for new articles can be a pretty opinionated subject. Standalone articles that fall well into an already subject category should be placed into mentioned subject category (documentation on something about simulation should be placed into the simulation section). However, things can get pretty complicated when an article combines or references two separate existing sections. In this situation, we advise the author to open an issue on the repository to get discussion going before opening the PR.

Note: All new articles will undergo a review process before being merged into the repository. This review process will be done by members of the WPILib team. New Articles must be on official *FIRST* supported Software and Hardware. Documentation on unofficial libraries or sensors *will not* be accepted. This process may take some time to review, please be patient.

Where to place sections?

Sections are quite tricky, as they contain a large amount of content. We advise the author to open an [issue](#) to gather discussion before opening up a PR.

Linking Other Articles

In the instance that the article references content that is described in another article, the author should make best effort to link to that article upon the first reference.

Imagine we have the following content in a drivetrain tutorial:

```
Teams may often need to test their robot code outside of a competition.
↪:ref:`Simulation <link-to-simulation:simulation>` is a means to achieve this.
↪Simulation offers teams a way to unit test and test their robot code without ever
↪needing a robot.
```

Notice how only the first instance of Simulation is linked. This is the structure the author should follow. There are times where a linked article has different topics of content. If you reference the different types of content in the article, you should link to each new reference once (except in situations where the author has deemed it appropriate otherwise).

46.2 Style Guide

This document contains the various RST/Sphinx specific guidelines for the frc-docs project. For guidelines related to the various WPILib code projects, see [the WPILib GitHub](#)

46.2.1 Filenames

Use only lowercase alphanumeric characters and - (minus) symbol.

For documents that will have an identical software/hardware name, append “Hardware” or “Software” to the end of the document name. IE, ultrasonics-hardware.rst

Suffix filenames with the .rst extension.

Note: If you are having issues editing files with the .rst extension, the recommended text editor is [Notepad++](#). Please make sure that [tabs are replaced with spaces](#), and the space indentation is set to 3.

46.2.2 Page References

Pages references will be auto-generated based on the page filename and section title.

For example, given the following file contributing.rst and a section called Page References, you would reference this by doing :ref:`contributing:Page References`

Note: Please note that document structure is preserved in references, with the root being the location of the conf.py file. To access documents in sub-folders, simply prepend the folder path before the filename. IE, :ref:`docs/software/sensors/ultrasonics-sensors:Ultrasonics - Sensors`

46.2.3 Text

All text content should be on the same line, if you need readability, use the word-wrap function of your editor.

Use the following case for these terms:

- roboRIO (not RoboRIO, roboRio, or RoboRio)
- LabVIEW (not labview or LabView)
- Visual Studio Code (VS Code) (not vscode, VScode, vs code, etc)
- macOS (not Mac OS, Mac OSX, Mac OS X, Mac, Mac OS, etc.)
- GitHub (not github, Github, etc)
- PowerShell (not powershell, Powershell, etc)
- Linux (not linux)
- Java (not java)

Use the ASCII character set for English text. For special characters (e.g. Greek symbols) use the [standard character entity sets](#).

Use `.. math::` for standalone equations and `:math:` for inline equations. A useful LaTeX equation cheat sheet can be found [here](#).

Use literals for filenames, function, and variable names.

Use of the registered trademarks *FIRST*® and FRC® should follow the Policy from [this page](#). Specifically, where possible (i.e. not nested inside other markup or in a document title), the first use of the trademarks should have the ® symbol and all instances of *FIRST* should be italicized. The ® symbol can be added by using `.. include:: <isonum.txt>` at the top of the document and then using `*FIRST*\ |reg|` or `FRC\ |reg|`.

Commonly used terms should be added to the [FRC Glossary](#). You can reference items in the glossary by using `:term:`deprecated``.

46.2.4 Whitespace

Indentation

Indentation should *always* match the previous level of indentation *unless* you are creating a new content block.

Indentation of content directives as new line `.. toctree::` should be 3 spaces.

Blank Lines

There should be 1 blank lines separating basic text blocks and section titles. There *should* be 1 blank line separating text blocks *and* content directives.

Interior Whitespace

Use one space between sentences.

46.2.5 Headings

Headings should be in the following structure. Heading underlines should match the same number of characters as the heading itself.

1. = for document titles. *Do not* use this more than *once* per article.
2. - for document sections
3. ^ for document sub-sections
4. ~ for document sub-sub-sections
5. If you need to use any lower levels of structure, you're doing things wrong.

Use title case for headings.

46.2.6 Lists

Lists should have a new line in between each indent level. The highest indent should have 0 indentation, and subsequent sublists should have an indentation starting at the first character of the previous indentation.

- Block one
- Block two
- Block three

- Sub 1
- Sub 2

- Block four

46.2.7 Code blocks

All code blocks should have a language specified.

1. Exception: Content where formatting must be preserved and has no language. Instead use text.

Follow the [WPILib style guide](#) for C++ and Java example code. For example, use two spaces for indentation in C++ and Java.

46.2.8 Admonitions

Admonitions (list [here](#)) should have their text on the same line as the admonition itself. There are exceptions to this rule however, when having multiple sections of content inside of admonition. Generally having multiple sections of content inside of an admonition is not recommended.

Use

```
.. warning:: This is a warning!
```

NOT

```
.. warning::  
   This is a warning!
```

46.2.9 Links

It is preferred to format links as anonymous hyperlinks. The important thing to note is the **two** underscores appending the text. In the situation that only one underscore is used, issues may arise when compiling the document.

```
Hi there, `this is a link <https://example.com>`__ and it's pretty cool!
```

However, in some cases where the same link must be referenced multiple times, the syntax below is accepted.

```
Hi there, `this is a link`_ and it's pretty cool!  
  
.. _this is a link: https://example.com
```

46.2.10 Images

Images should be created with 1 new line separating content and directive.

All images (including vectors) should be less than 500 kilobytes in size. Please make use of a smaller resolution and more efficient compression algorithms.

```
.. image:: images/my-article/my-image.png  
   :alt: Always add alt text here describing the image.
```

Image Files

Image files should be stored in the document directory, sub-directory of document-name/images

They should follow the naming scheme of short-description.png, where the name of the image is a short description of what the image shows. This should be less than 24 characters.

They should be of the .png or .jpg image extension. .gif is unacceptable due to storage and accessibility concerns.

Note: Accessibility is important! Images should be marked with a `:alt:` directive.

```
.. image:: images/my-document/my-image.png
   :alt: An example image
```

Vector Images

SVG files are supported through the `svg2pdfconverter` Sphinx extension.

Simply use them as you would with any other image.

Note: Ensure that any embedded images in the vector do not bloat the vector to exceed the 500KB limit.

```
.. image:: images/my-document/my-image.svg
   :alt: Always add alt text here describing the image.
```

Draw.io Diagrams

Draw.io (also known as diagrams.net) diagrams are supported through `svg` files with embedded `.drawio` metadata, allowing the `svg` file to act as a source file of the diagrams, and to be rendered like a normal vector graphics file.

Simply use them like you would any other vector image, or any other image.

```
.. image:: diagrams/my-document/diagram-1.drawio.svg
   :alt: Always add alt text here describing the image.
```

Draw.io Files

Draw.io files follow almost the same naming scheme as normal images. To keep track of files that have the embedded `.drawio` metadata, append a `.drawio` to the end of the file name, before the extension, meaning the name of the file should be `document-title-1.drawio.svg` and so on. Additionally, diagrams should be stored in the document directory in a sub-folder named `diagrams`.

For the specifics of saving a diagram as a `.svg` with metadata, take a look at [Draw.io Saving Instructions](#).

Warning: Make sure you don't modify any file that is in a `diagrams` folder, or ends in `.drawio.svg` in any program other than draw.io, otherwise you might risk breaking the metadata of the file, making it uneditable.

46.2.11 File Extensions

File extensions should use code formatting. For example, use:

```
``.png``
```

instead of:

```
.png  
".png"  
"``.png``"
```

46.2.12 Table of Contents (TOC)

Each category should contain an `index.rst`. This index file should contain a maxdepth of 1. Sub-categories are acceptable, with a maxdepth of 1.

The category `index.rst` file can then be added to the root index file located at `source/index.rst`.

46.2.13 Examples

```
Title  
====  
This is an example article  
  
.. code-block:: java  
    System.out.println("Hello World");  
  
Section  
-----  
This is a section!
```

46.2.14 Important Note!

This list is not exhaustive and administrators reserve the right to make changes. Changes will be reflected in this document.

46.3 Build Instructions

This document contains information on how to build the HTML, PDF, and EPUB versions of the frc-docs site. frc-docs uses Sphinx as the documentation generator. This document also assumes you have basic knowledge of [Git](#) and console commands.

46.3.1 Prerequisites

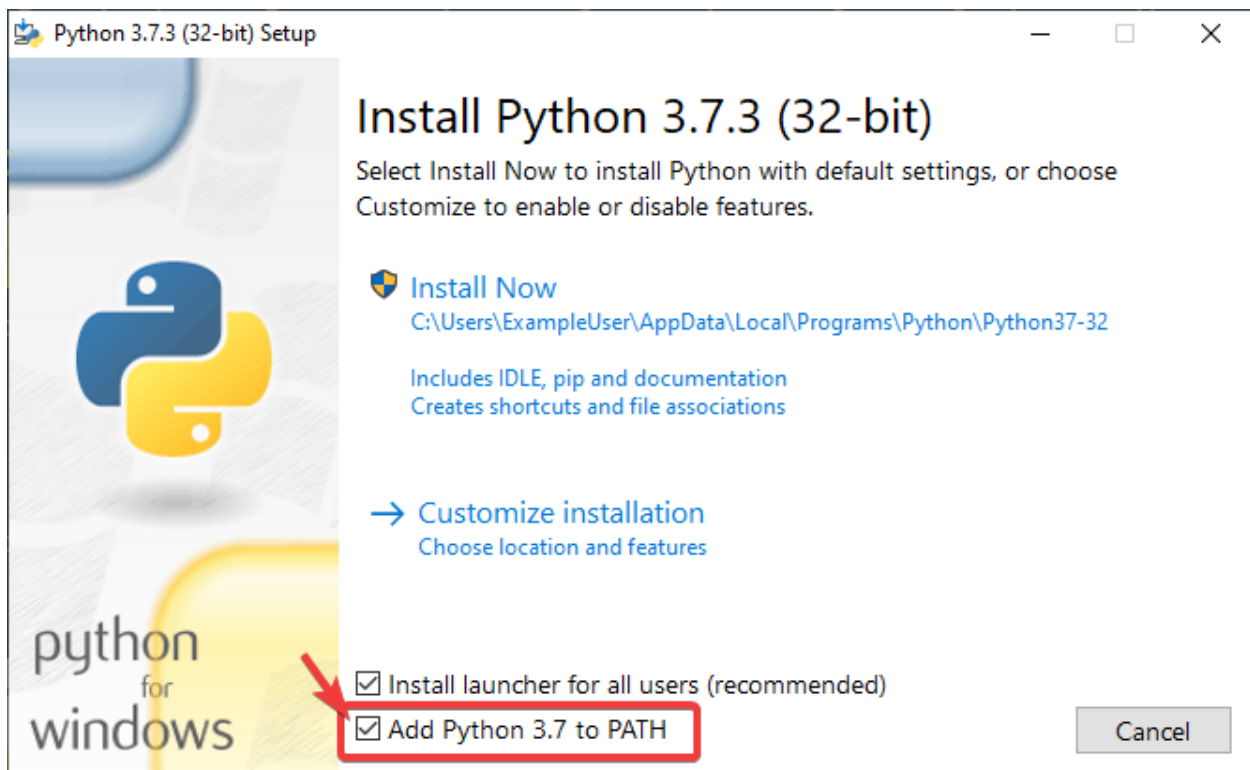
Ensure that [Git](#) is installed and that the frc-docs repository is cloned by using `git clone https://github.com/wpilibsuite/frc-docs.git`.

Windows

Note: The majority of MikTeX packages are not required for building HTML, additional packages may be required for building PDF and EPUB.

- Python 3.6 or greater
- MiKTeX
- Perl

Ensure that Python is in your Path by selecting the **Add Python to PATH** toggle when installing Python.



Once Python is installed, open up Powershell. Then navigate to the frc-docs directory. Run the following command: `pip install -r source/requirements.txt`

Install the missing MikTeX packages by navigating to the frc-docs directory, then running the following command from Powershell: `mpm --verbose --require=@miktex-packages.txt`

Lastly, you need to install rsvg-convert by running the `scripts/install-rsvg-convert.ps1` script in powershell. This requires administrator access.

Linux (Ubuntu)

```
$ sudo apt update
$ sudo apt install python3 python3-pip
$ python3 -m pip install -U pip setuptools wheel
$ python3 -m pip install -r source/requirements.txt
$ sudo apt install -y texlive-latex-recommended texlive-fonts-recommended texlive-
↳ latex-extra latexmk texlive-lang-greek texlive-luatex texlive-xetex texlive-fonts-
↳ extra dvipng librsvg2-bin
```

46.3.2 Building

Open up a Powershell Window or terminal and navigate to the frc-docs directory that was cloned.

```
PS > cd "%USERPROFILE%\Documents"
PS C:\Users\Example\Documents> git clone https://github.com/wpilibsuite/frc-docs.git
Cloning into 'frc-docs'...
remote: Enumerating objects: 217, done.
remote: Counting objects: 100% (217/217), done.
remote: Compressing objects: 100% (196/196), done.
remote: Total 2587 (delta 50), reused 68 (delta 21), pack-reused 2370
Receiving objects: 100% (2587/2587), 42.68MiB | 20.32 MiB/s, done.
Receiving deltas: 100% (1138/1138), done/
PS C:\Users\Example\Documents> cd frc-docs
PS C:\Users\Example\Documents\frc-docs>
```

Lint Check

Note: Lint Check will not check line endings on Windows due to a bug with line endings. See [this issue](#) for more information.

It's encouraged to check any changes you make with the linter. This **will** fail the buildbot if it does not pass. To check, run `.\make lint`

Link Check

The link checker makes sure that all links in the documentation resolve. This **will** fail the buildbot if it does not pass. To check, run `.\make linkcheck`

Image Size Check

Please run `.\make sizecheck` to verify that all images are below 500KB. This check *will* fail CI if it fails. Exclusions are allowed on a case by case basis and are added to the `IMAGE_SIZE_EXCLUSIONS` list in the configuration file.

Redirect Check

Files that have been moved or renamed must have their new location (or replaced with 404) in the `redirects.txt` file in source.

The redirect writer will automatically add renamed/moved files to the redirects file. Run `.\make rediraffewritediff`.

Note: if a file is both moved and substantially changed, the redirect writer will not add it to the `redirects.txt` file, and the `redirects.txt` file will need to be manually updated.

The redirect checker makes sure that there are valid redirects for all files. This **will** fail the buildbot if it does not pass. To check, run `.\make rediraffecheckdiff` to verify all files are redirected. Additionally, an HTML build may need to be ran to ensure that all files redirect properly.

Building HTML

Type the command `.\make html` to generate HTML content. The content is located in the `build/html` directory at the root of the repository.

46.3.3 Building PDF

Warning: Please note that PDF build on Windows may result in distorted images for SVG content. This is due to a lack of `librsvg2-bin` support on Windows.

Type the command `.\make latexpdf` to generate PDF content. The PDF is located in the `build/latex` directory at the root of the repository.

46.3.4 Building EPUB

Type the command `.\make epub` to generate EPUB content. The EPUB is located in the `build/epub` directory at the root of the repository.

46.3.5 Adding Python Third-Party libraries

Important: After modifying frc-docs dependencies in any way, `requirements.txt` must be regenerated by running `poetry export -f requirements.txt --output source/requirements.txt --without-hashes` from the root of the repo.

frc-docs uses [Poetry](#) to manage its dependencies to make sure builds are reproducible.

Note: Poetry is **not** required to build and contribute to frc-docs content. It is *only* used for dependency management.

Installing Poetry

Ensure that Poetry is installed. Run the following command: `pip install poetry`

Adding a Dependency

Add the dependency to the `[tool.poetry.dependencies]` section of `pyproject.toml`. Make sure to specify an exact version. Then, run the following command: `poetry lock --no-update`

Updating a Top-Level Dependency

Update the dependency's version in the `[tool.poetry.dependencies]` section of `pyproject.toml`. Then, run the following command: `poetry lock --no-update`

Updating Hidden Dependencies

Run the following command: `poetry lock`

46.4 Draw.io Saving Instructions

Warning: Make sure you don't modify any file that is in a `diagrams` folder, or ends in `.drawio.svg` in any program other than draw.io, otherwise you might risk breaking the metadata of the file, making it uneditable.

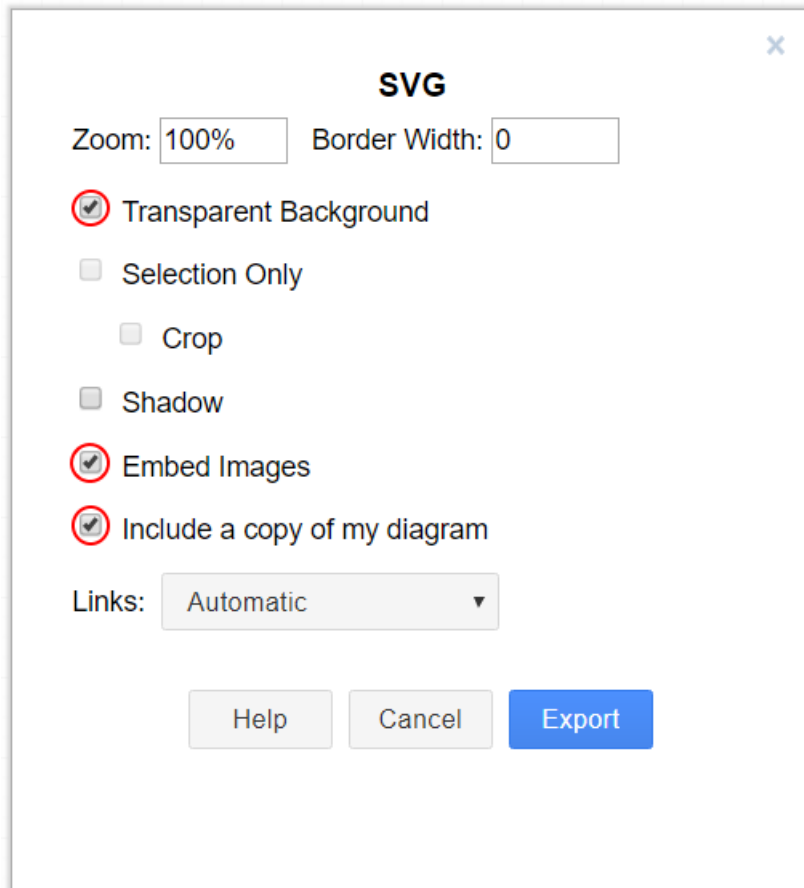
Draw.io (also known as [diagrams.net](#)) are supported when saved as `svg` files, with embedded XML metadata for the draw.io source file (normally stored as `.drawio`). This allows these images to both act as source files for the diagrams that can be edited in the future, and be rendered as normal `svg` files.

There are a few methods to save a diagram with the embedded metadata, but using the export menu is preferred because it allows us to embed any images in the diagram, otherwise they might not render properly on the docs.

This method is applicable to both draw.io desktop and the web version at diagrams.net.

To export go to File - Export as - SVG.... Make sure Include a copy of my diagram is enabled to embed the diagram metadata, and that Embed Images is enabled so image files in the diagram are embedded so they render in the docs. Additionally, mark the Transparent Background option to make sure the background is displayed correctly.

The export menu should look something like this:



Then just click Export then select where you would like to save the file and save it.

Note: When saving, make sure you follow the style-guide at [Draw.io Files](#)

46.5 Translations

frc-docs supports translations using the web-based [Transifex](#) utility. frc-docs has been translated into Spanish - Mexico (es_MX), French - Canada (fr_CA) and Turkish - Turkey (tr_TR). Chinese - China (zh_CN), Hebrew - Israel (he_IL), and Portuguese - Brazil (pt_BR) have translations in progress. Translators that are fluent in *both* English and one of the specified languages would be greatly appreciated to contribute to the translations. Even once a translation is complete, it needs to be updated to keep up with changes in frc-docs.

46.5.1 Workflow

Here are some steps to follow for translating frc-docs.

1. Sign up for [Transifex](#) and ask to join the [frc-docs project](#), and request access to the language you'd like to contribute to.
2. Join GitHub [discussions](#)! This is a direct means of communication with the WPILib team. You can use this to ask us questions in a fast and streamlined fashion.
3. You may be contacted and asked questions involving contributing languages before being granted access to the frc-docs translation project.
4. Translate your language!

46.5.2 Links

Links must be preserved in their original syntax. To translate a link, you can replace the TRANSLATE ME text (this will be replaced with the English title) with the appropriate translation.

An example of the original text may be

```
For complete wiring instructions/diagrams, please see the :doc:`Wiring the FRC
↪Control System Document <Wiring the FRC Control System document>`.
```

where the Wiring the FRC Control System Document then gets translated.

```
For complete wiring instructions/diagrams, please see the :doc:`TRANSLATED TEXT
↪<Wiring the FRC Control System document>`.
```

Another example is below

```
For complete wiring instructions/diagrams, please see the :ref:`TRANSLATED TEXT <docs/
↪zero-to-robot/step-1/how-to-wire-a-robot:How to Wire an FRC Robot>`
```

46.5.3 Publishing Translations

Translations are pulled from Transifex and published automatically each day.

46.5.4 Accuracy

Translations should be accurate to the original text. If improvements to the English text can be made, open a PR or issue on the [frc-docs](#) repository. These can then get translated on merge.

46.6 Top Contributors

- [Daltz333](#) - 246 contributions
- [sciencewhiz](#) - 140 contributions
- [Oblarg](#) - 99 contributions
- [prateekma](#) - 82 contributions
- [jasondaming](#) - 63 contributions
- [yellowjaguar5](#) - 46 contributions
- [AustinShalit](#) - 44 contributions
- [Kevin-OConnor](#) - 23 contributions
- [Starlight220](#) - 15 contributions
- [calcmogul](#) - 15 contributions

46.7 Top Translators

46.7.1 Chinese

- -T.K.-
- 8192 Dhc
- Atlus Zhang
- Emma Yuan
- Jiangshan Gong
- Keseterg
- Michael Zhao
- Ningxi Huang
- Team 5308
- Tianrui Wu
- Tianshuang Zhang

- Xilei
- Xun Sun
- Yitong Zhao
- Yuhao Li
- 孙 宇
- 孙 宇
- 孙 宇
- 孙 宇
- 孙 宇

46.7.2 French

- Alexandra Schneider
- Andre Theberge
- Andy Chang
- Austin Shalit
- Dalton Smith
- Daniel Renaud
- Étienne Beaulac
- Félix Giffard
- Kaitlyn Kenwell
- Laura Luna Bedard
- Martin Regimbald
- Regis Bekale
- Sami G.-D.
- Youdlain Marcellus

46.7.3 Portuguese

- Amanda Carolina Wilmsen
- Bruno Toso
- Gabriel Silveira
- Gabrielatomaz
- Günther Steinmeier
- Matheus Heitor Timm Chanan
- Nadjia Dias
- Pedro Henrique Dias Pellicioli

- Tales Dias De Almeida Silva
- Vinícius Castro

46.7.4 Spanish

- Austin Shalit
- Christopher Marley
- Daniel Núñez
- Diana Ramos
- Emiliano Vargas
- Fernanda Reveles
- Fernando Soltero
- Gibrán Verástegui
- Heber Sepúlveda
- Heriberto Gutierrez
- Hugo Espino
- Karina Torres
- Lian Eng
- Miguel Angel De León Adame
- Paulina Maynez
- Pierre Cote
- Ranferi Lozano
- Rodrigo Acosta
- Sofia Fernandez
- Zara Moreno

46.7.5 Turkish

- Hasan Bilgin
- Müfit Alkaya
- Demet T
- Esra Özemre
- Müfit Alkaya_3390
- Ceren Oktemer
- Melis Aldeniz
- Duru Ünlü
- Demet Tumkaya

- Arhan Ünay
- Duru Hatipoğlu
- Mufit Alkaya
- Ada Zagyapan
- Nesrin Serra Köşkeroğlu
- Kaan Çakıl
- Yaren K
- Ege Feyzioglu
- Eray Özсарay
- Deniz Ornek
- Ece Ekincikli

46.7.6 Hebrew

- Aric Radzin
- Dalton Smith
- Itay Ziv
- Ofek Ashery
- Starlight220
- Yotam Shlomi

Developing with allwpilib

Important: This document contains information for developers *of* WPILib. This is not for programming FRC® robots.

This is a list of links to the various documentation for the [allwpilib](#) repository.

47.1 Core Repository

47.2 NetworkTables

A

accelerometer, [379](#)
alliance, [379](#)
auto, [379](#)

C

C++, [379](#)
control effort, [1094](#)
control input, [1094](#)
control law, [1094](#)
controller, [1094](#)
COTS, [379](#)

D

deprecated, [379](#)
DHCP, [379](#)
dynamics, [1094](#)

E

error, [1094](#)

F

FMS, [379](#)

G

gain, [1094](#)
GradleRIO, [379](#)
gyroscope, [379](#)

H

heading, [379](#)
hidden state, [1094](#)

I

IMU, [379](#)
input, [1094](#)

J

Java, [379](#)

K

KOP, [379](#)
KOP chassis, [379](#)

L

LabVIEW, [379](#)

M

match, [379](#)
measurement, [1094](#)
model, [1095](#)
moment of inertia, [1095](#)

N

NetworkTables, [380](#)

O

observer, [1095](#)
output, [1095](#)

P

plant, [1095](#)
process variable, [1095](#)

R

reference, [1095](#)
rise time, [1095](#)
RP, [380](#)

S

setpoint, [1095](#)
settling time, [1095](#)
simulation, [380](#)
state, [1095](#)
steady-state error, [1095](#)
step input, [1095](#)
step response, [1095](#)
system, [1096](#)
system identification, [1096](#)
system response, [1096](#)

T

teleop, [380](#)

trajectory, [380](#)

X

x-dot, [1096](#)

x-hat, [1096](#)