FIRST Robotics Competition

WPILib

May 22, 2026

Zero to Robot

1 Introduction 3

2 Step 1: Building your Robot 5

3 Step 2: Installing Software 23

4 Step 3: Preparing Your Robot 59

5 Step 4: Programming your Robot 87

6 Hardware Component Overview 113

7 Software Component Overview 145

8 What is WPILib? 157

9 2026 Overview 159

10 VS Code Overview 177

11 Dashboards 203

12 Telemetry 337

13 FRC LabVIEW Programming 361

14 FRC Python Programming 397

15 Hardware APIs 403

16 CAN Devices 513

17 Basic Programming 529

18 Support Resources 609

19 FRC Glossary 613

20 Driver Station 621

21 RobotBuilder 647
22 Robot Simulation 721
23 OutlineViewer 759
24 roboRIO Team Number Setter 761
25 WPIcal 763
26 Vision Processing 775
27 Command-Based Programming 837
28 Kinematics and Odometry 937
29 NetworkTables 969
30 Path Planning 1021
31 roboRIO 1061
32 Advanced GradleRIO 1075
33 Advanced Controls 1101
34 Convenience Features 1271
35 WPILib Example Projects 1279
36 Third Party Example Projects 1285
37 Hardware - Basics 1287
38 Hardware Tutorials 1371
39 Sensors 1373
40 Getting Started with Romi 1411
41 Getting Started with XRP 1437
42 Networking Introduction 1449
43 Networking Utilities 1475
44 Contributing to frc-docs 1477
45 Developing with allwpilib 1499
46 Privacy Policy 1501
Index 1505

Welcome to the *FIRST*® Robotics Competition Control System Documentation! This site contains everything you need to know for programming a competition robot!

Community translations can be found in a variety of languages in the menu towards the bottom-right of the screen.

Returning Teams

If you are a returning team, please check out the overview of changes from 2025 to 2026, known issues, and quick start guide for updating.

Changelog

Known Issues

Quick Start

New Teams

The Zero-to-Robot tutorial will guide you through preparation, wiring, and programming a basic robot!

Go to Zero-to-Robot

Hardware Overview

An overview of the hardware components available to teams.

Go to Hardware Overview

Software Overview

An overview of the software components and tools available to teams.

Go to Software Overview

Programming Basics

Documentation that is useful throughout a team's programming process.

View articles

Advanced Programming

Documentation that is suited toward veteran teams. This includes content such as Path Planning and Kinematics.

View articles

Hardware

Hardware tutorials and content available for teams.

View articles

Romi and XRP Robots

The Romi and XRP robots are low-cost platforms for practicing WPILib programming.

View Romi articles

View XRP articles

API Documentation

Java, C++, and Python class documentation.

Java

C++

Python

Software Tools

Essential tooling such as FRC Driver Station, Dashboards, roboRIO Imaging Tool, and more.

View articles

Example Projects

This section showcases the available example projects that teams can reference in VS Code.

View articles

Status Light Quick Reference

Quick reference guide for the status lights on a variety of FRC hardware.

View article

3rd Party libraries

Tutorial on adding 3rd party libraries such as CTRE and REV to your robot project.

View article

1 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

1.1.4 Python

- List of various guides to learn Python

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/Python or jump back to the home page using the logo at the top left to explore the rest of the content.

2 Step 1: Building your Robot

An overview of the available control system hardware can be found *here*.

2.1 Introduction to FRC Robot Wiring

Note

This document details the wiring of a basic electronics board for the Kitbot using components provided in recent Kit of Parts. For a more general reference to wiring which includes other components, see *Introduction to FRC Robot Wiring*

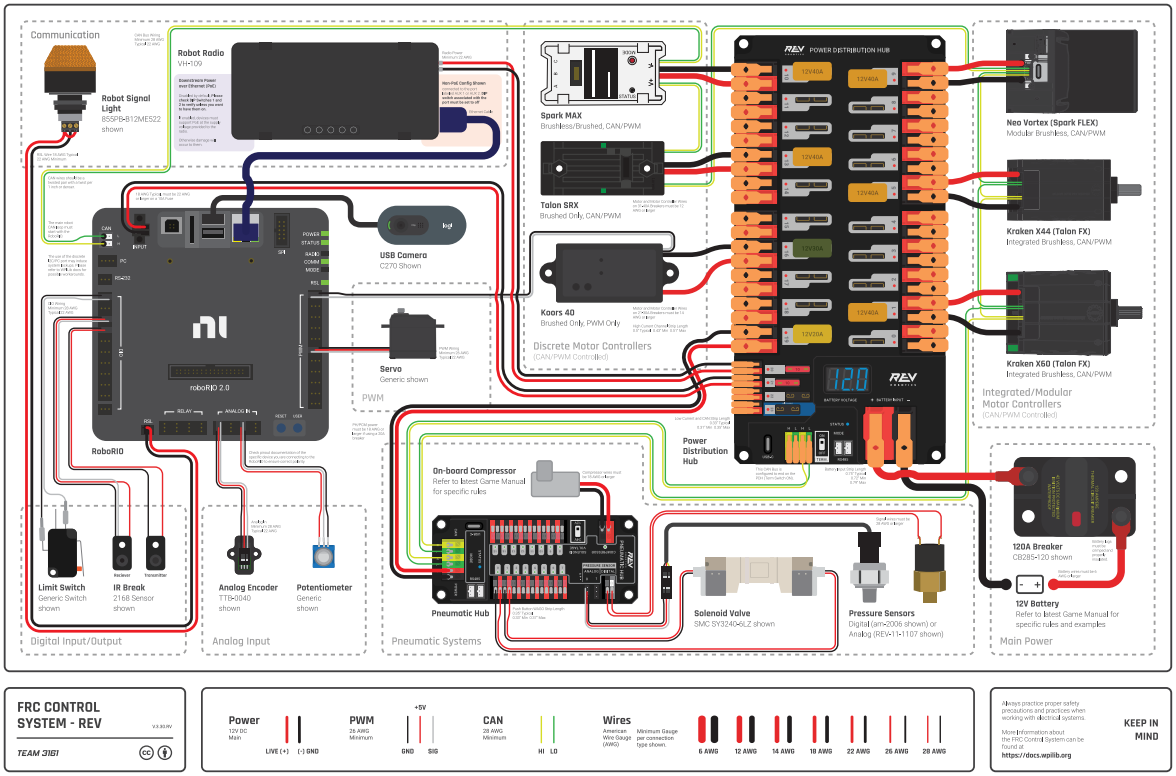
2.1.1 Overview

Diagram courtesy of FRC® Team 3161 and Stefen Acepcion.

2.1.2 Gather Materials

Locate the following control system components and tools

- Kit Materials:
  - Power Distribution Hub (*PDH*)
  - roboRIO
  - Vivid Hosting VH-109 radio (may be already installed on Kitbot)
  - Robot Signal Light (*RSL*) (may be already installed on Kitbot)
  - 6x SPARK MAX
  - 120A Circuit breaker
  - 4x 40A Circuit breaker
  - 1x 30A Circuit breaker
  - 6 AWG (16 mm^2) Red wire

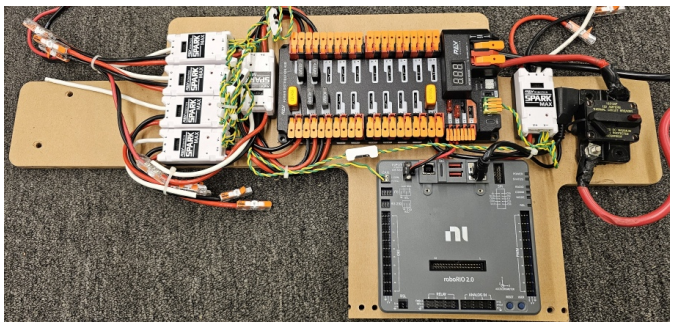


  - 18 AWG (1 mm^2) Red/Black wire
  - Vivid Hosting PoE Injector cable or similar
  - 14 Wago 221 inline splicing connector with lever
  - 2x Anderson SB50 battery connectors
  - 6 AWG (16 mm^2) Terminal lugs
  - 12V Battery
  - Red/Black Electrical tape
  - Dual Lock material or fasteners
  - Zip ties
  - 1/2in (12mm) plywood
- Tools Required:
  - Very small flat head screwdriver (eyeglass repair size)
  - Wire cutters, wire stripper (6, 10, 18, 22 AWGs), and terminal crimper (6AWG terminal) or vice
  - 7/16in (11mm may work if imperial is unavailable) box end wrench or nut driver
  - Additional 7/16in wrench/nut driver or Philips head screw driver
  - Scissors
  - Drill with 1/4in (6mm) bit

2.1.3 Create the Base for the Control System

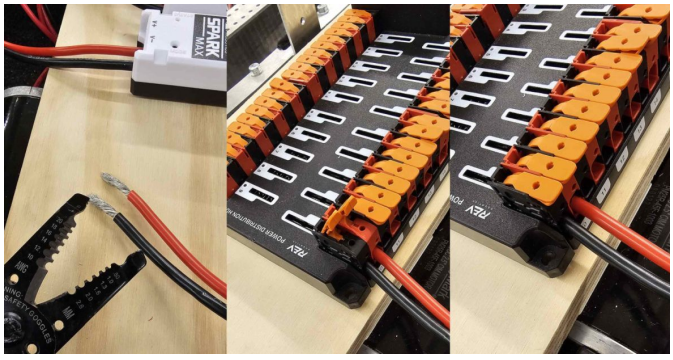
The instructions below heavily rely on the wiring pre-attached to components. The layout provided is designed around using a single board which mounts to the top of the inner rails of the AM14U chassis. Teams are welcome to modify this design and layout but note that additional connectors and wire may be required.

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. The radio and RSL will be mounted directly to the Kitbot superstructure. It is recommended to wire as much as possible before installing the board in the robot, the board is difficult to work on while installed. The board has very little clearance between the PDH and the bottom of the hopper, locate the PDH as far towards the back of the robot as possible while still allowing room for wires between the PDH and roboRIO.

2.1.5 Motor Controller Power



Requires: Wire Stripper

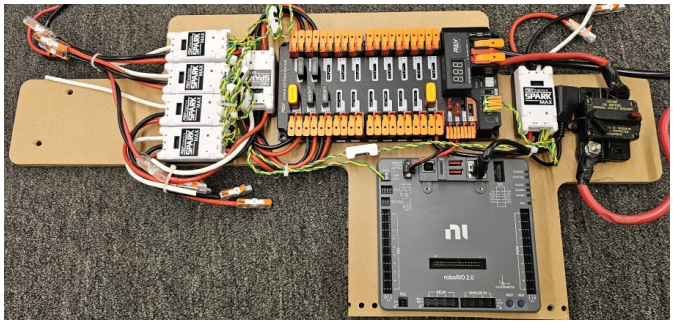
Connect each motor controller's input power (red/black side with no white) to the PDHs high current output channels.

1. Strip approximately 1/2 in. (~12mm) of insulation off the red and black power input wires for one SPARK MAX. While the SPARK MAX wire is 12 AWG, because it is high strand count, the 10 AWG size on your wire strippers may work better.
2. Lift both levers on the PDH for one high current channel until they latch open (the red and black terminals for one channel are indicated by the white label across them below and behind the terminals).

3. Carefully insert the stripped wire all the way into the terminals and press the latches back down to secure the wires.
4. Repeat for each controller. Pay attention to which controllers wire to which channel to ensure all controllers will reach without extension.

> **Tip**
>
> If the wire end has become a bit messy and is hard to insert cleanly, a small twist (~ 1/2 a turn) may help. Do not heavily twist the wire, this prevents the terminal inside the PDH from making solid contact with the wire.

2.1.6 Fasten Components



Requires: Recloseable fastener, scissors, drill with bit

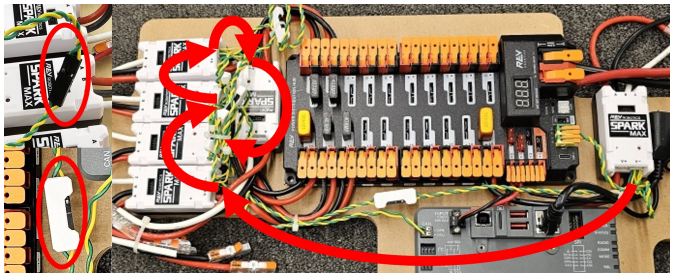
Cable ties plus reclosable fastener make a great pair for securing robot electronics. The cable ties help hold the components down while the recloseable fastener helps hold them in place. The instructions below describe securing your electronics with this method, though you may choose to use hardware (nuts and bolts) to secure some components instead.

1. Secure each item down with recloseable fastener. One easy way to do this is to secure the two sides of the fastener together, peel one backing, secure the fastener(s) to the component, then peel the other backing and carefully press the component into place on the board. To ensure you have enough fastener for both electronics and the battery door, some recommended lengths are provided below:
    - SPARK MAX: 2x 1 in. pieces each
    - roboRIO: 4x 1 in. pieces
    - PDH: 3x 2 in. pieces (or 4x 1 in.)
    - 120A Breaker: 2x 2 in. pieces
2. Mark hole locations for drilling ziptie holes
    - SPARK MAX: 2x holes centered between each SPARK MAX (1 aligned with each ziptie notch) plus 2x just outside each outer controller. 2 cable ties will pass through each hole.
    - PDH: 1 hole directly underneath each device mounting hole (4x total), 1 holes just outside the device footprint near each mounting hole (4x total)
    - roboRIO: 1 hole directly underneath each ziptie notch (4x total), 1 hole just outside the device footprint near each mounting hole (4x total)

- 120A breaker: 1 hole directly underneath each device mounting hole (2x total), 1 hole just outside the device footprint near each mounting hole (2x total)

3. Remove components and drill holes. Using a "rocking" or "peeling" motion can often help remove components from recloseable fastener without removing the fastener adhesive.
4. Place the components back down on the boards. You may wish to wait to actually secure components with cable ties until wiring is complete. Some cable ties may be able to double as wire management.

2.1.7 CAN Bus connections - SPARK MAXes

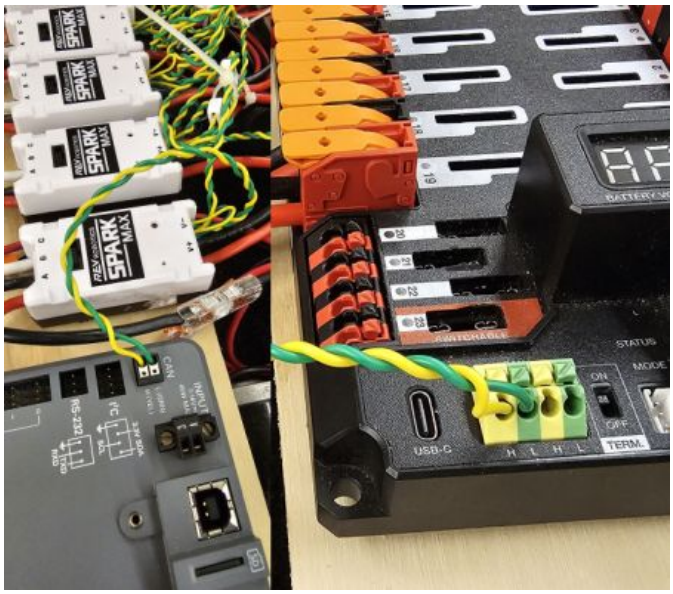


Requires: SPARK MAX CAN Cables and Cable retention clips (found in SPARK MAX boxes)

The SPARK MAX controllers and PDH will communicate with the roboRIO over CAN. Connect the SPARK MAX CAN cables in a chain, skipping the 3rd controller and coming back to it, as shown by the red arrows.

1. Insert a SPARK MAX CAN Cable into each SPARK MAX.
2. Starting with the single SPARK MAX near the roboRIO, take either end of the CAN cable and connect to the opposite end of the CAN cable from the SPARK MAX closest to the battery notch.
3. Secure the connection by placing a Cable retention clip over the joined connectors.
4. Using the remaining end from the SPARK MAX near the battery notch **skipping the controller directly above** connect to the third SPARK MAX in the cluster of four.
5. Then connect from the third SPARK MAX to the fourth in the group, followed by the SPARK MAX closest to the PDH.
6. Finally, connect the last SPARK MAX, that you skipped earlier.
7. You should now have a chain that connects all the SPARK MAX controllers together with a free end on the SPARK MAX closest to the roboRIO and on the second controller in from the battery notch.
8. You may want to use cable ties to organize the extra CAN wire before moving on.

2.1.8 CAN Bus connections - PDH and roboRIO



Requires: Wire cutters, wire stripper (22 AWG)

Connect one free end of the CAN bus chain to the PDH and the other to the roboRIO.

1. Locate the free end of the CAN cable connected to the SPARK MAX in the group of four. Cut off the connector and strip 5/16in (~8mm) of both the yellow and green wire.
2. Insert each wire into the corresponding terminal on the roboRIO by holding down the white button with a fingernail or screw driver while inserting the wire. Give the wire a sharp tug to ensure it has been inserted properly.
3. Locate the remaining free end of the CAN chain (attached to the single isolated SPARK MAX). Cut off the connector and strip 3/8in (~9mm) of both the yellow and green wire.
4. Insert each wire into either matching colored terminal of the PDH by either depressing the button or opening the latch (depending on PDH version).
5. Ensure the PDH CAN termination switch (labeled “TERM” and located next to the CAN terminals) is set to ON. If not, adjust with a small screw driver.

2.1.9 Attach Robot Side Battery Connector



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

Attach the red wire from the battery connector to the main breaker using a terminal lug and attach the black side to the power input of the PDH.

1. Attach terminal lug to positive (red) wire of battery connector. Strip .75in off the black wire.
2. Lift the lever above the black main power input terminal on the PDH until it clicks into place. Insert the wire. Pull the lever down to secure the wire.
3. Using a 7/16in (11mm) box end wrench, remove the nut on the “Batt” side of the main breaker and secure the positive terminal of the battery connector

2.1.10 Wire Breaker to Power Distribution



Requires: 6 AWG (16 mm^2) red wire, 1x 6 AWG (16 mm^2) terminal lugs, 7/16in (11mm) wrench

Connect 6AWG wire between the second terminal of the 120A circuit breaker and the red power input of the PDH

1. Secure one terminal lug to the end of the 6 AWG (16 mm^2) red wire.
2. Using the 7/16in (11mm) wrench, remove the nut from the “AUX” side of the 120A main breaker and place the terminal over the stud.
3. Loosely secure the nut (you may wish to remove it shortly to cut and strip the other end of the wire). Measure out the length of wire required to reach the positive terminal of the PDH.
4. Cut and strip the other end of the red wire.
5. Using the 7/16” (11 mm) wrench, secure the wire to the “AUX” side of the 120A main breaker.
6. Lift the lever on the positive (red) input terminal of the PDH, insert the wire, then close the terminal.

2.1.11 Insulate power connections



Requires: Electrical tape

Using electrical tape, insulate the two connections to the 120A breaker.

2.1.12 roboRIO Power





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

Using 18AWG wire, connect a non-switchable fused channel of the PDH (20-22) with a 10A fuse to the power connector of the roboRIO

1. Insert the 10A fuse into the PDH in one of the non-switchable fused channels (20-22).
2. Strip ~5/16” (~8 mm) on both the red and black 18 AWG (1 mm^2) wire and connect to the corresponding terminals on the PDH channel where the fuse was installed
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 Radio Power



Requires: Vivid Hosting PoE adapter cable

Connect the PoE adapter cable to a non-switchable fused channel (20-22) with a 10A fuse to power the VH-109 radio.

1. Insert the 10A fuse into the PDH in one of the non-switchable fused channels (20-22).
2. Lift the levers of the PDH terminals for that channel, insert the ferrules from the red and black wires on the PoE adapter into the corresponding terminals, then close the levers.
3. Connect the Ethernet plug of the PoE adapter to the roboRIO.

Warning

The Vivid PoE adapter works opposite of the REV PoE adapter. The REV PoE adapter can not be directly substituted for the 2026 KitBot. If the Vivid PoE adapter is not available, wire power directly to the radio 12V port and connect the roboRIO to the AUX2 port if the radio is not the V1.5 hardware revision.

2.1.14 Circuit Breakers



Requires: 6x 40A circuit breakers

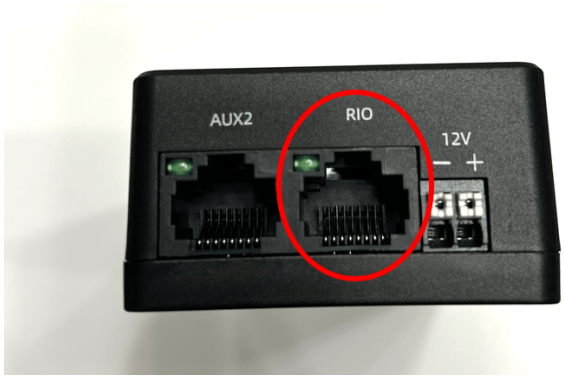
Insert 40-amp Circuit Breakers into the positions on the PDH corresponding with the Wago connectors the motor controllers are connected to. Note that the white graphic indicates which breakers are associated with which terminal pairs.

2.1.15 Pause



If you are preparing the wiring outside the robot, pause at this point. The remaining wiring must be completed after the board is installed on the robot.

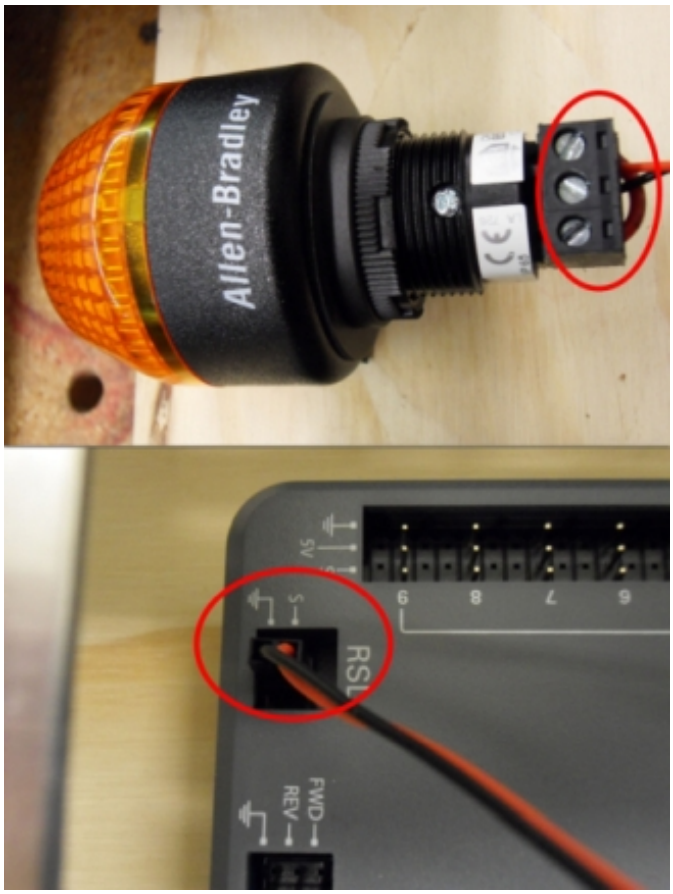
2.1.16 Ethernet Cable



Requires: 1x Ethernet cable

Connect an Ethernet cable from the RJ45 (Ethernet) socket of the PoE adapter to the RIO port on the VH-109 radio.

2.1.17 Robot Signal Light

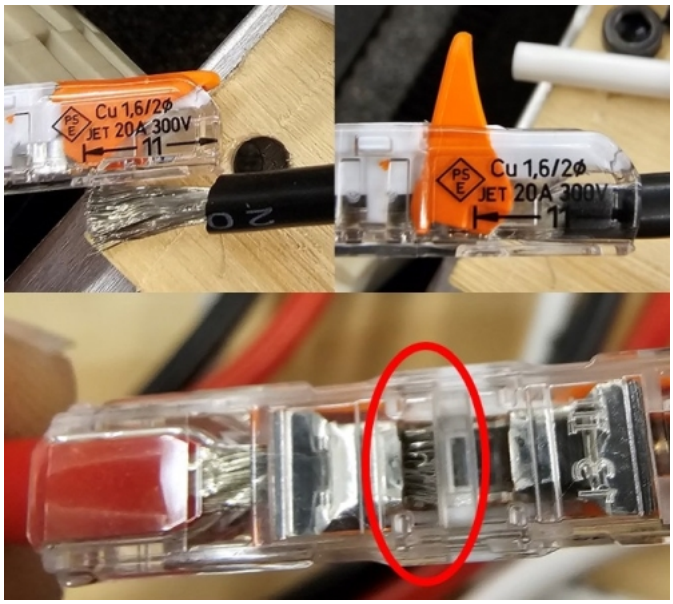


Requires: Wire stripper, 2 pin cable with Molex jacket from Molex box, Robot Signal Light, 18 AWG (1 mm^2) red wire, very small flat screwdriver

Using the 2 Pin Molex cable, connect the roboRIO RSL port to 2 terminals of the RSL (black wire in the middle) and using 18AWG red wire, jumper the outside two ports together.

1. Using a small flat screwdriver, ensure the terminals of the RSL wiring connector are open.
2. Take the loose end of the 2 pin cable (or cut off one connector if both ends have connectors) and 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.

2.1.18 Motor Power



Requires: Wire stripper, 14x Wago 221 terminals, 12AWG wire

For each motor, use the Wago 221 terminals to connect the output of the motor controller to the motor. A recommended arrangement is to use the isolated SPARK MAX for the feeder motor, the other single SPARK MAX for the intake/launcher, and the four grouped SPARK MAXes for the Drivetrain, split into pairs for each side. Note that for the intake/launcher motor, an additional intermediate connection is recommended in order to have the necessary length to route the wire safely along robot elements.

1. Using the print on the side of the 221 connector (upper left image) as a guide, strip 11mm from the end of one wire.
2. Lift the orange lever of the 221 connector until it latches up, then insert the wire fully into the terminal. Flip the lever back down.
3. Using the inspection window on the bottom of the connector, check that the wire is fully inserted by verifying the wire is visible beyond the metal funnel of the terminal.
4. Repeat this process to connect each drivetrain CIM motor and the feeder motor directly to a motor controller output, red-to-red, black-to-black.
5. For the intake/launcher motor, it is recommend to use 12AWG wire as an extension. Attach 12AWG wire to the motor wires, then run the wire along robot components until you reach the motor contoller. Cut and strip the 12AWG extension wire and secure to the motor controller output.

2.1.19 STOP



Danger

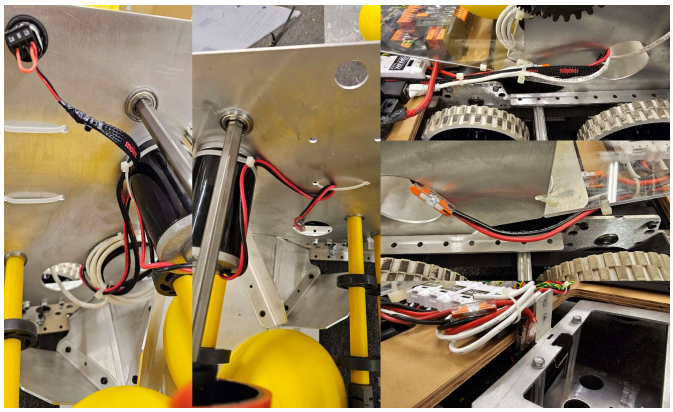
Before plugging in the battery, make sure all connections have been made with the proper polarity. Ideally have someone that did not wire the robot check to make sure all connections are correct.

- 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 V+ terminal on the motor controller (not M+!!!!)
- For each non-motor controller device, verify that the red wire runs from a red terminal on the PD and connects to a red terminal or terminal labeled + or V on the component.

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.

2.1.20 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 and protected. Secure wires to robot components to make them less likely to be caught and pulled by other robots, or when performing robot maintenance. Make sure wires are not pulled too tight (don't put any strain on connection points) and that they do not have enough slack to reach moving components (such as drivetrain belts) while the robot is operating.

> **Warning**
>
> If you are passing wires through any of the openings in the KitBot sheet metal, it is recommended to shield the edge of the metal, the wires passing through, or both, to avoid wire damage. Electrical tape or Gaffers tape work well for this purpose.

2.1.21 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. If you hear any clicking, or see any smoke, power the system off immediately, clicking is likely the sound of circuit breakers tripping.

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 controls 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 continue on to install software and configure your radio and roboRIO, then load up the provided KitBot code.

3 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 you 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

- 2026 FRC Game Tools

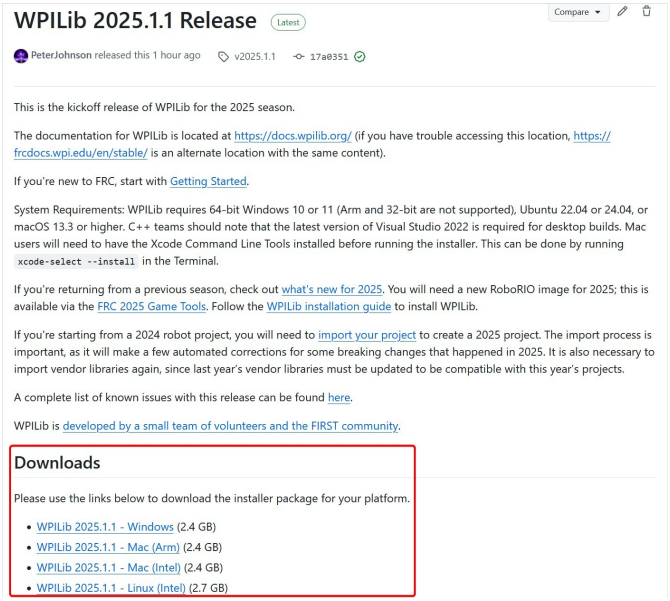
LabVIEW Teams

- LabVIEW Base Installer (Note: Click on link for “Individual Offline Installers”)

Java/C++ Teams

- Java/C++ WPILib Installer

Once on the GitHub releases page, scroll to the download section in the middle of the page.



Then click on the correct binary for your OS and architecture to begin the download.

> **Note**
>
> After downloading the Java/C++ WPILib installer, run it once while connected to the internet and select *Install for this User* then *Create VS Code zip to share with other computers/OSes for offline install* and save the downloaded VS Code archives 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++, Java, and Python 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.

3.2.1 Requirements

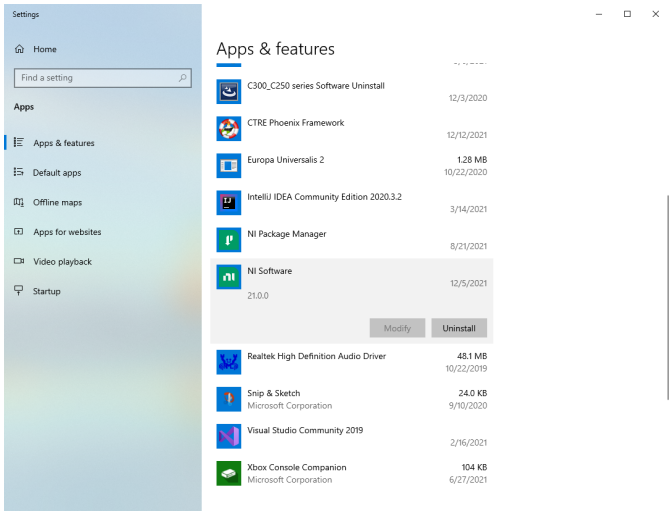
- Windows 10 or higher (Windows 10, 11)

3.2.2 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.

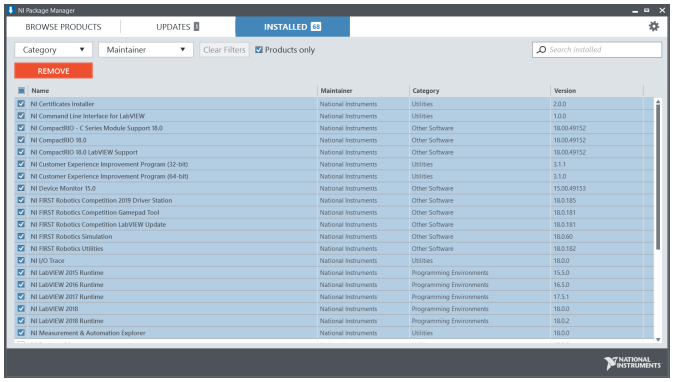
Before installing the new version of LabVIEW it is recommended to remove any old versions. The new version will likely co-exist with the old version, but all testing has been done with FRC 2026 only. Make sure to back up any team code located in the "User\LabVIEW Data" directory before un-installing. Then click Start >> Add or Remove Programs. Locate the entry labeled "NI Software", and select Uninstall.



Select Components to Uninstall

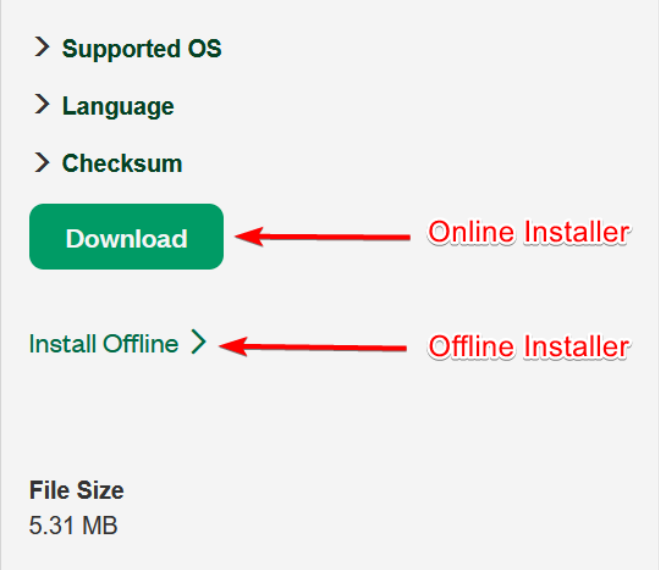
In the dialog box that appears, select all entries. The easiest way to do this is to de-select the “Products Only” check-box and select the check-box to the left of “Name”. Click Remove. Wait for the uninstaller to complete and reboot if prompted.

> **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.



3.2.3 Getting LabVIEW installer

Download the LabVIEW for FRC 2026 installer from NI. Be sure to select the correct version from the drop-down.



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

Note

This is a large download (~11GB). It is recommended to use a fast internet connection and to use the NI Downloader to allow the download to resume if interrupted.

3.2.4 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

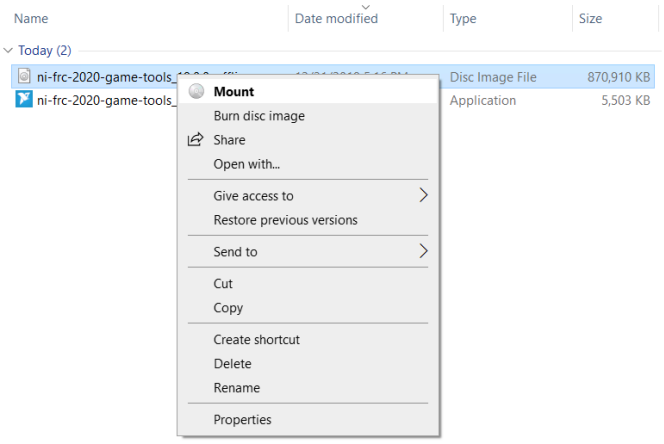
Online Installer

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

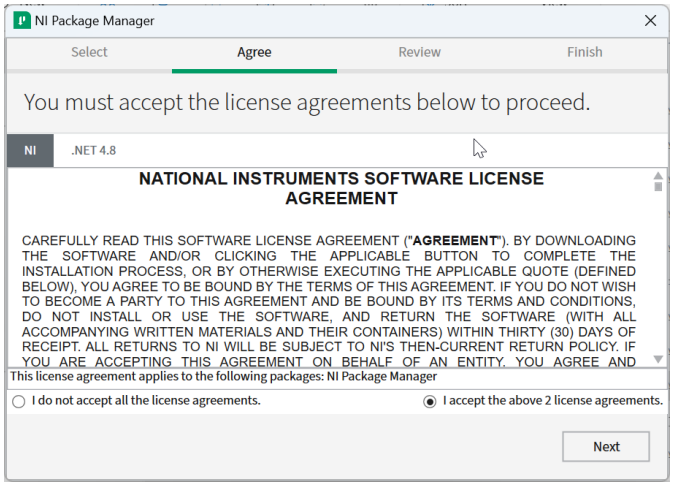
Offline Installer (Windows 10+)

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 install 7-Zip and use that to extract the iso.

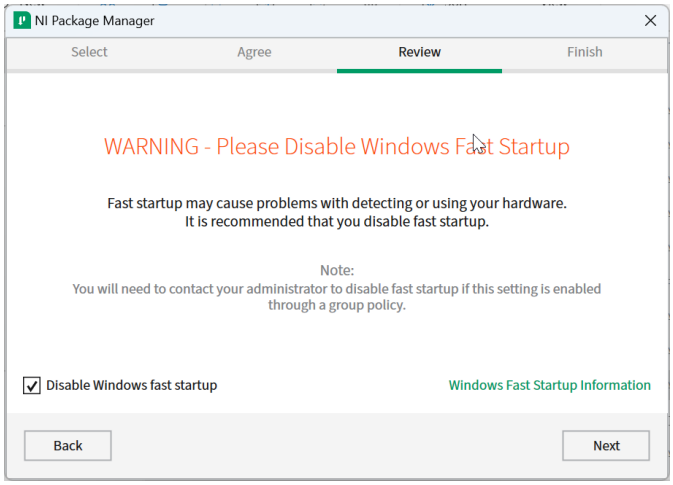


NI Package Manager License



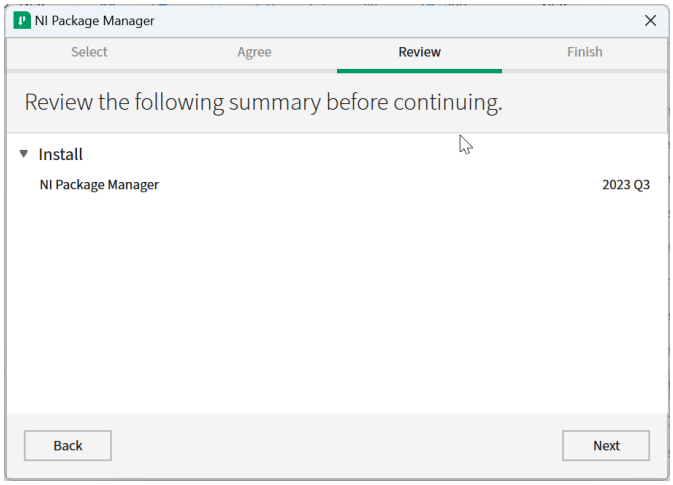
If you see this screen, click *Next*

Disable Windows Fast Startup



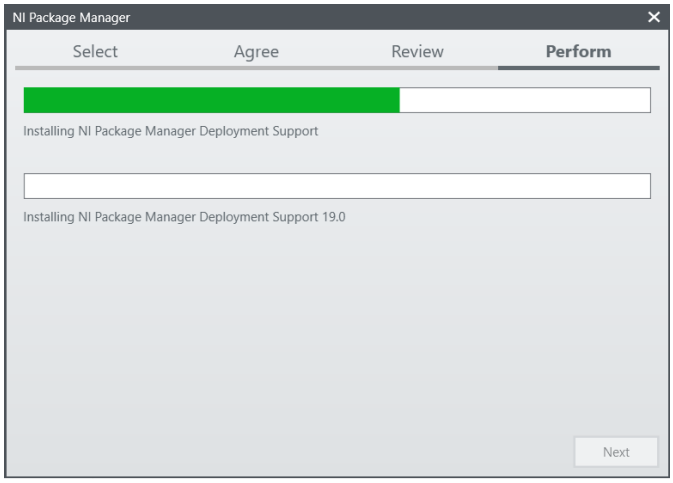
If you see this screen, click *Next*

NI Package Manager Review



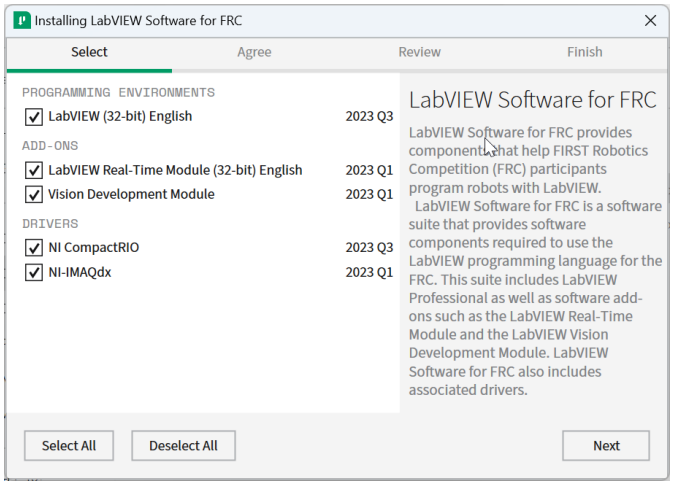
If you see this screen, click *Next*

NI Package Manager Installation



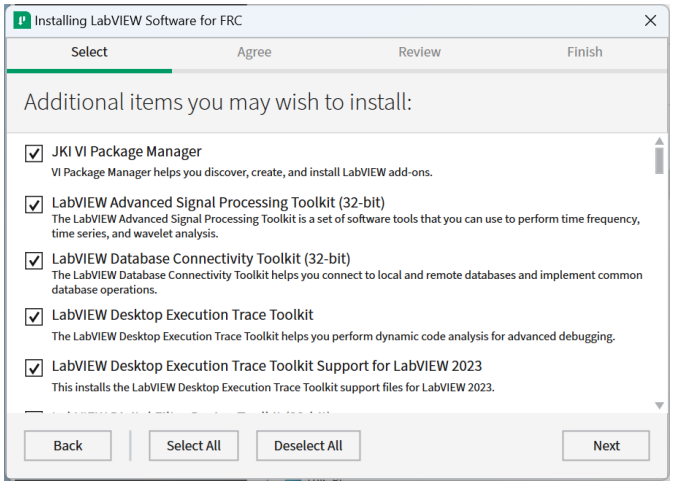
Installation progress of the NI Package Manager will be tracked in this window

Product List



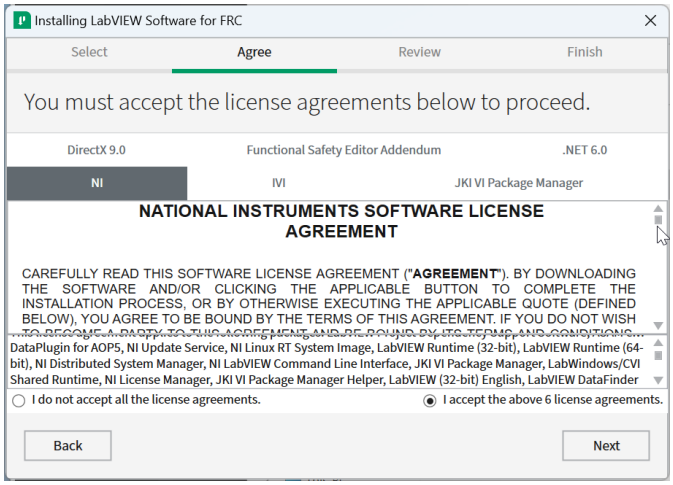
Click *Next*

Additional Packages



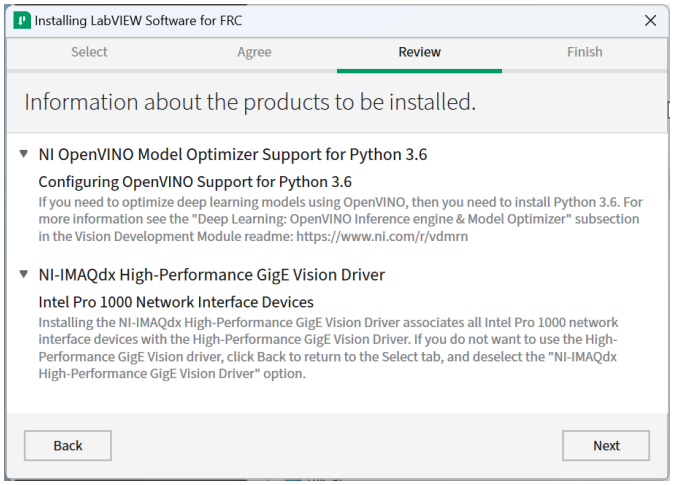
Click *Next*

License agreements



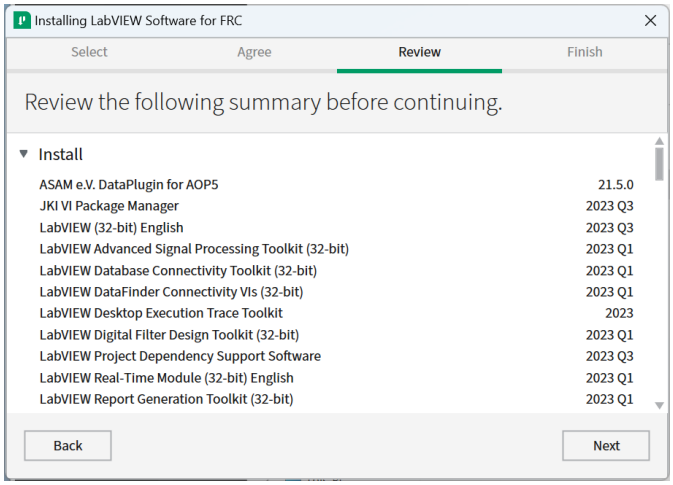
Check “I accept…” then Click *Next*

Product Information



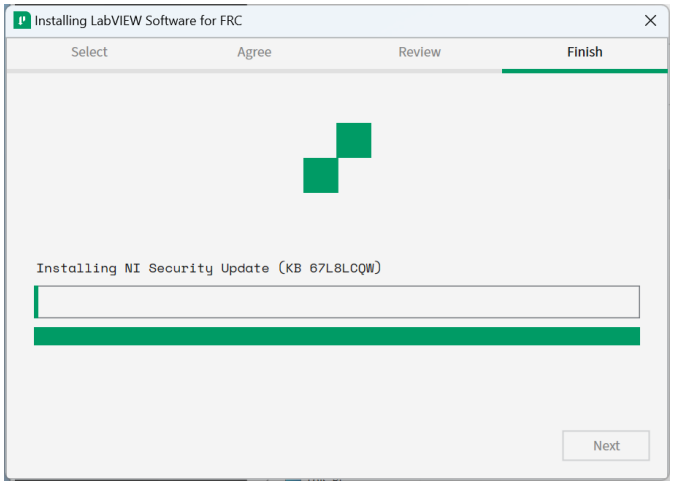
Click *Next*

Start Installation



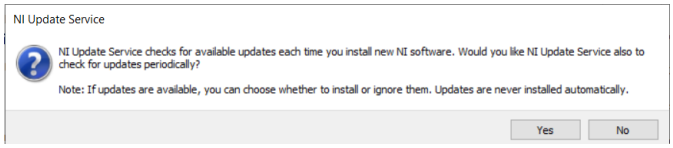
Click *Next*

Overall Progress



Overall installation progress will be tracked in this window

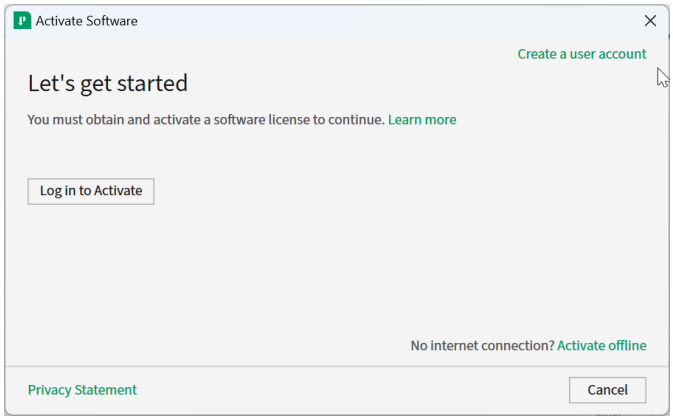
3.2.5 NI Update Service



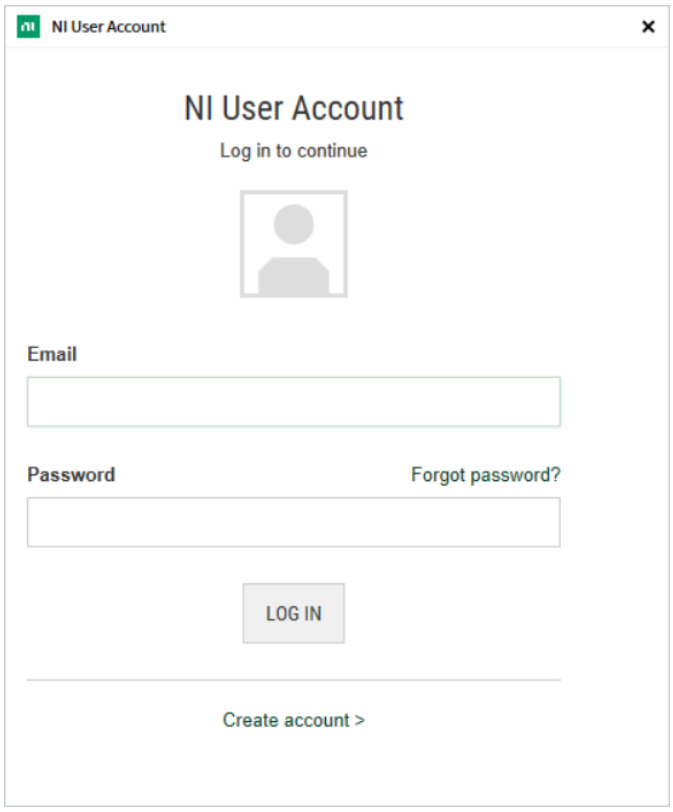
You will be prompted whether to enable the NI update service. You can choose to not enable the update service.

> **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).

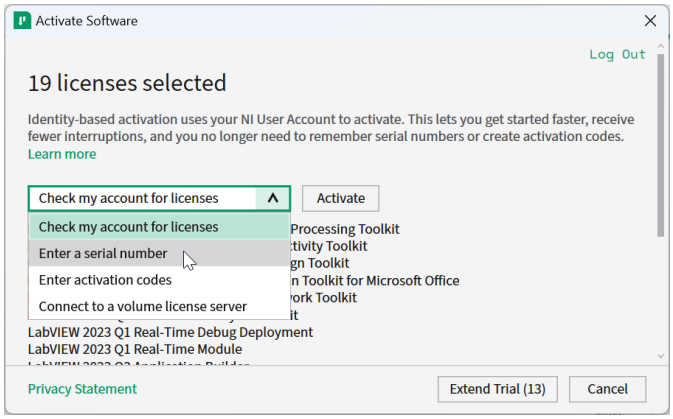
NI Activation Wizard



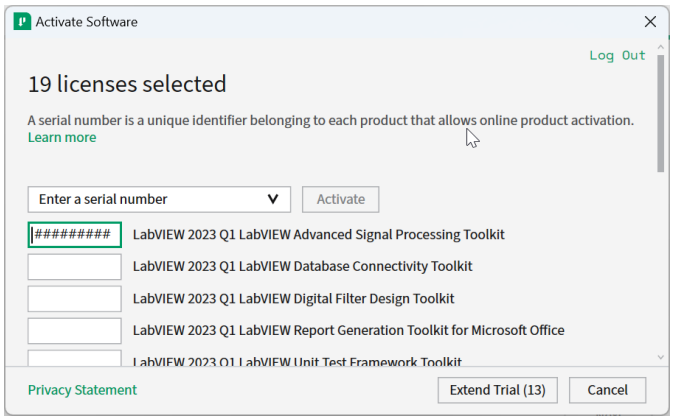
Click the *Log in to Activate* button.



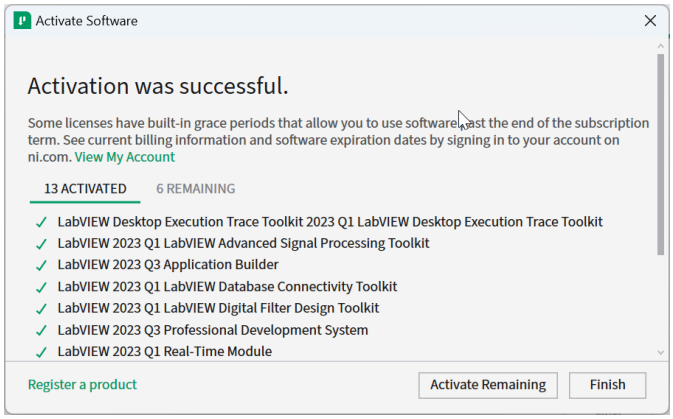
Log into your ni.com account. If you don't have an account, select *Create account* to create a free account.



From the drop-down, select enter a serial number

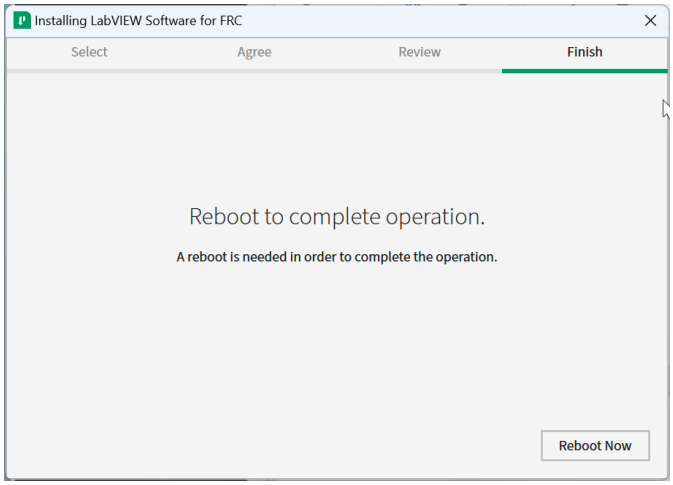


Enter the serial number in all the boxes. Click *Activate*.



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 *Finish*.

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 roboRIO Imaging Tool and Images

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

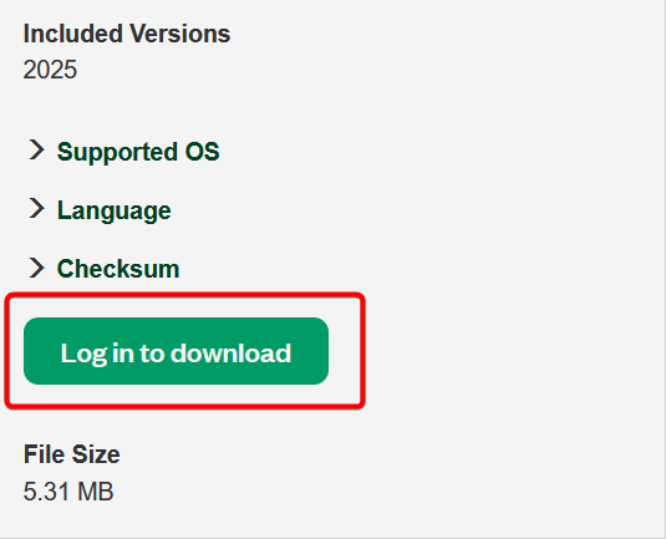
> **Note**
>
> No components from the LabVIEW Software for FRC package are required for running either the Driver Station or Imaging Tool.

3.3.1 Requirements

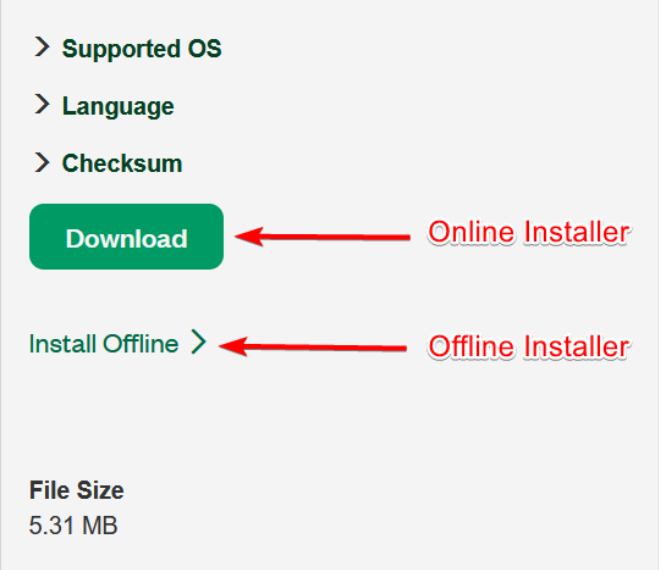
- Windows 10 or higher (Windows 10, 11).

3.3.2 Download Game Tools

Navigate to the FRC Game Tools page from NI.



Log-in or create an National Instruments account.



If you wish to install on other machines offline, click *Install Offline* or else choose *Download* to download the online installer which will download while installing.

3.3.3 Uninstall Old Versions (Recommended)

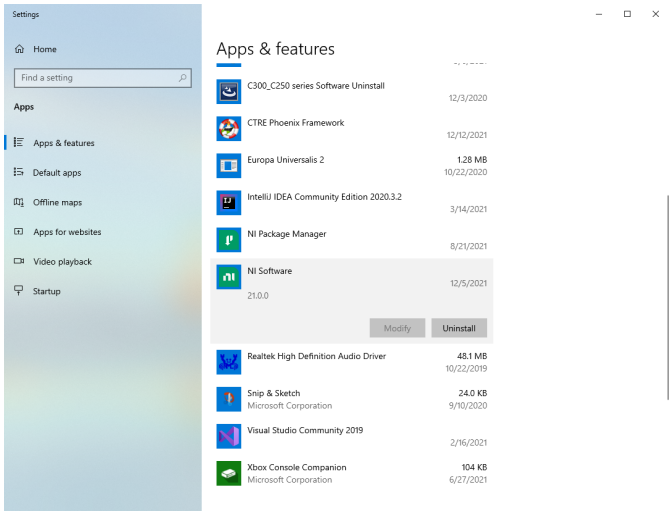
Important

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 2026 only. Then click Start >> Add or Remove Programs. Locate the entry labeled "NI Software", and select *Uninstall*.

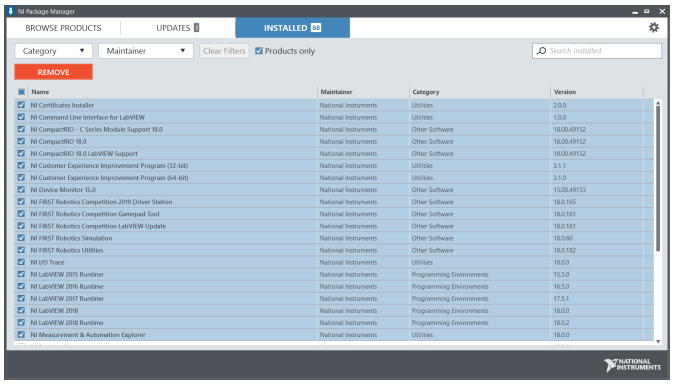
Note

It is only necessary to uninstall previous versions when installing a new year's tools (or when a beta is installed). For example, uninstall the 2021 tools before installing the 2022 tools. It is not necessary to uninstall before upgrading to a new update of the 2022 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.



3.3.4 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.

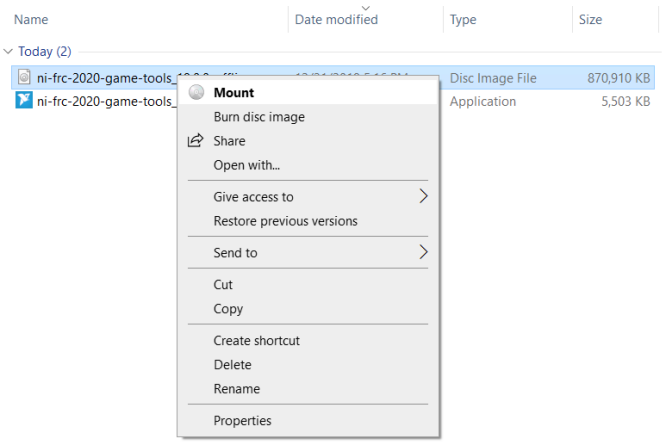
Extraction

Online

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

Offline (Windows 10+)

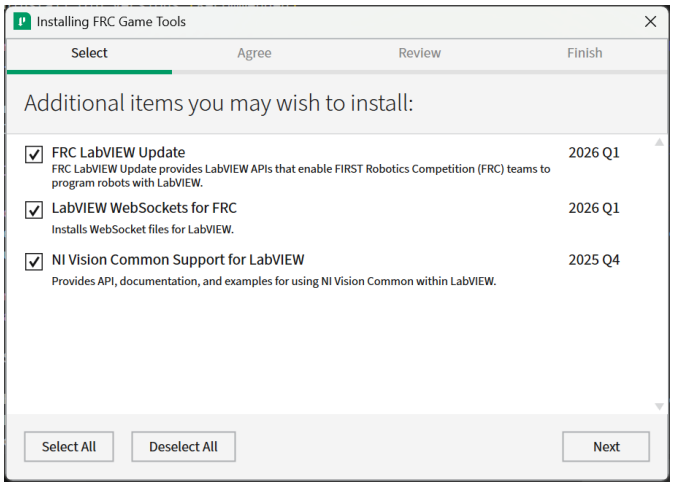
Right click on the downloaded iso file and select *mount*. Run `install.exe` from the mounted iso. Click *Yes* if a Windows Security prompt appears.



Note

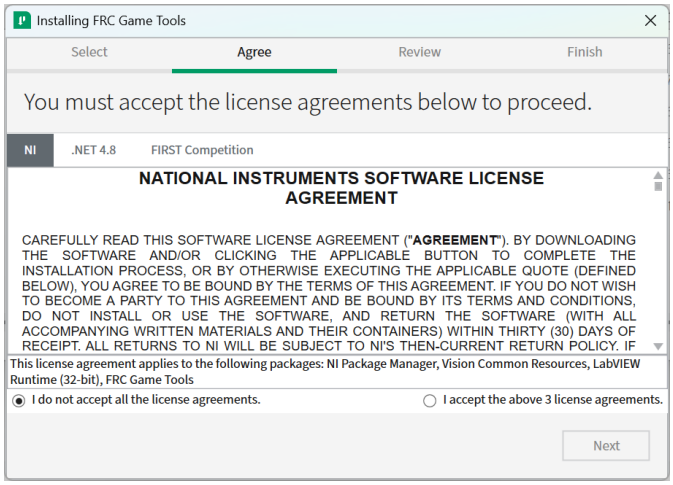
Other installed programs may associate with iso files and the *mount* option may not appear. If that software does not give the option to mount or extract the iso file, then install 7-Zip and use that to extract the iso.

Additional Software



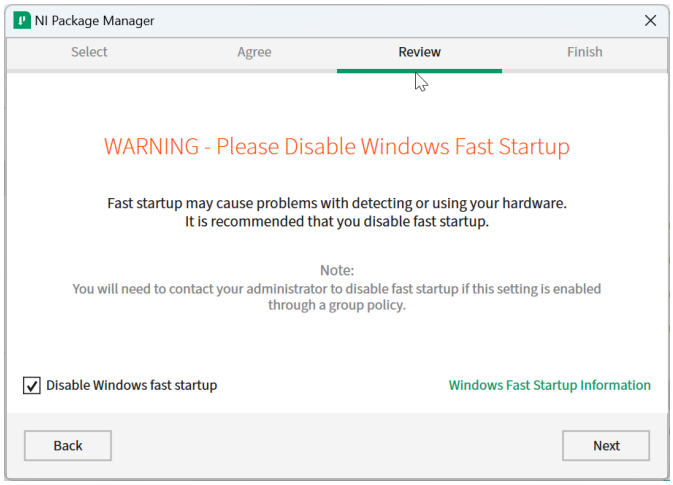
If you see this screen, click *Next*.

License Agreements



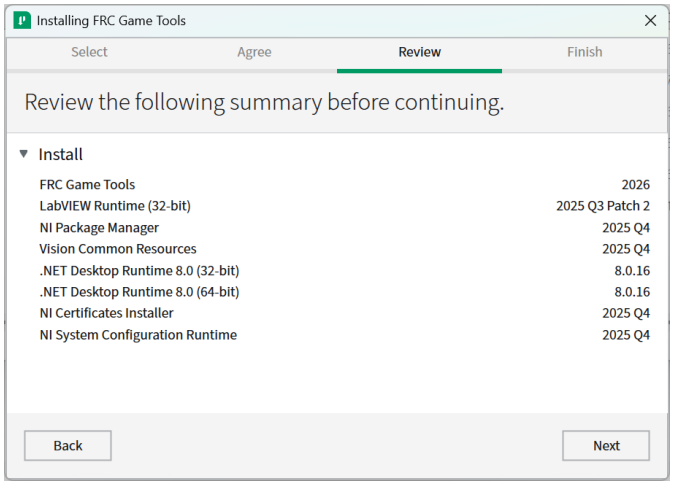
Select *I accept…* then click *Next*

Disable Windows Fast Startup



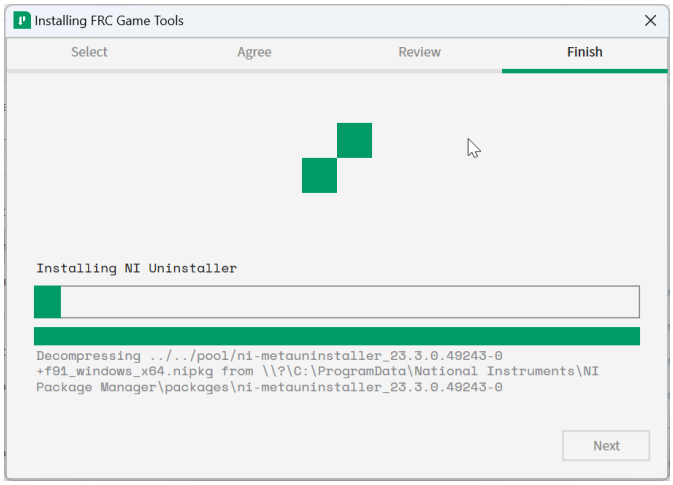
It is recommended to leave this screen as-is, as Windows Fast Startup can cause issues with the NI drivers required to image the roboRIO. Go ahead and click *Next*.

Review Summary



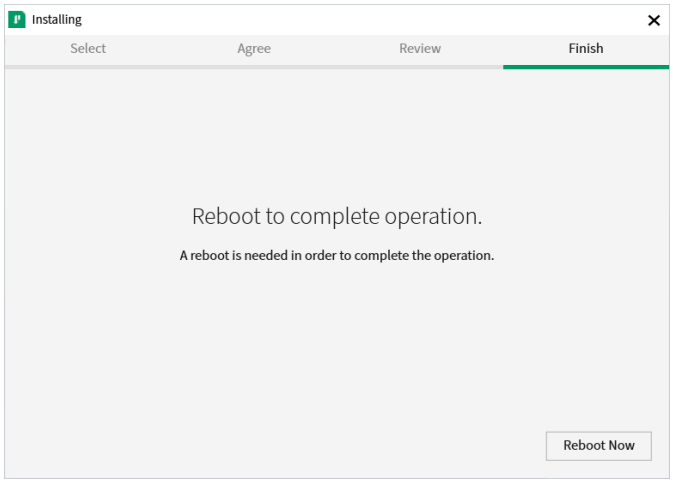
Click *Next*.

Detail Progress



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

3.3.5 Reboot to Complete Installation



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

3.4 WPILib Installation Guide

This guide will walk you through installing the WPILib development environment on your computer. This includes the WPILib tools, Visual Studio Code, and all dependencies needed to develop and deploy code to the roboRIO. WPILib Installation is required for Java and C++ teams. LabVIEW teams can skip to *Installing LabVIEW for FRC (LabVIEW only)*. Python teams can do a full install per these instructions to get a Visual Studio Code environment with some python extensions, or do a tools only installation to install dashboards such as Shuffleboard and Elastic. Python teams additionally need to follow the *Python Installation Guide*.

3.4.1 Prerequisites

Supported Operating Systems and Architectures:

- Windows 10 & 11, 64 bit only. 32 bit and Arm are not supported
- Ubuntu 22.04 & 24.04, 64 bit. Other Linux distributions with glibc >= 2.34 may work, but are unsupported
- macOS 13.3 or higher, both Intel and Arm.

Warning

The following OSes are no longer supported: macOS 12 or earlier, Ubuntu 18.04 & 20.04, Windows 7, Windows 8.1, and any 32-bit Windows.

Note

Windows 10 support from Microsoft ended in October 2025. We intend to continue supporting Windows 10 through the 2026 season, but may have to drop support in 2027. Teams should start planning their upgrade path to Windows 11, or switch to one of the supported Linux distributions listed.

Note

C++ compilation is memory intensive. A minimum of 32 GB of RAM is recommended for C++ teams.

This tutorial shows Windows 11, but the steps are identical for all operating systems. Notes differentiating operating systems will be shown.

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.

3.4.2 Downloading

WPILib Installer

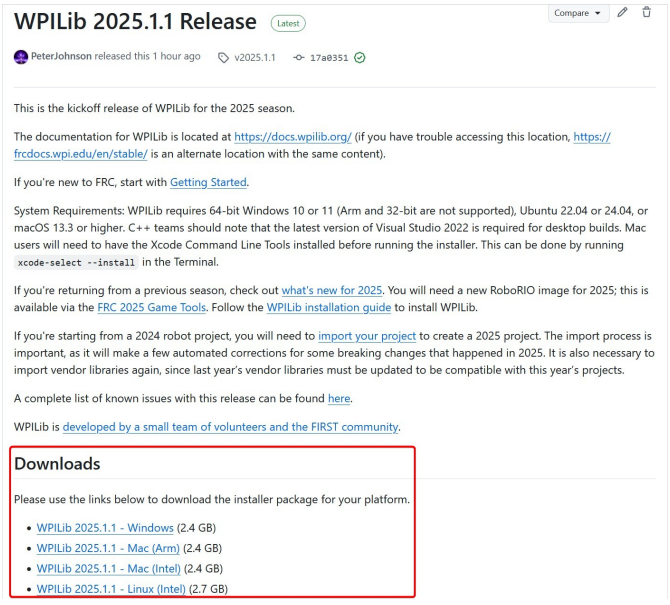
WPILib 2026.2.1 Release - January 16, 2026 Downloads

Downloads for other platforms

Release Notes

You can also download the latest release of the installer from GitHub.

Once on the GitHub releases page, scroll to the Downloads section.



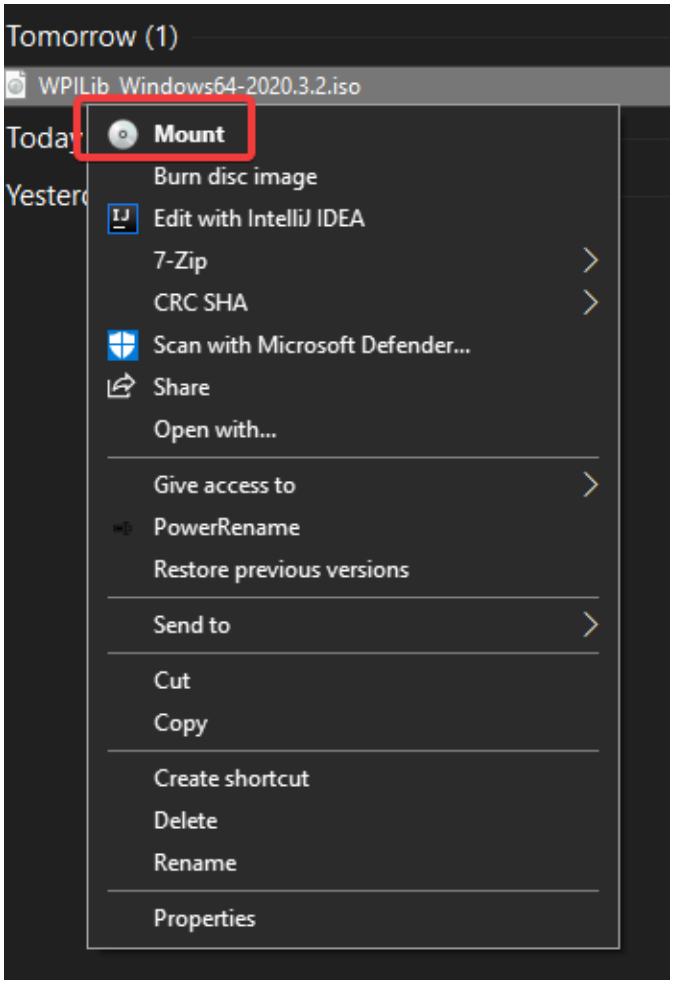
Then click on the correct binary for your OS and architecture to begin the download.

3.4.3 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 10+ users can right click on the downloaded disk image and select *Mount* to open it. Then launch `WPILibInstaller.exe`.

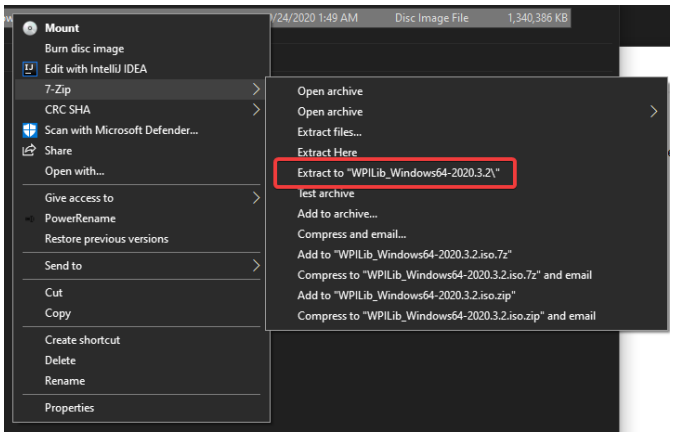


Note

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 below.

You can use 7-zip to extract the disk image by right-clicking, selecting *7-Zip* and selecting *Extract to...*. Windows 11 users may need to select *Show more options* at the bottom of the

context menu.



After opening the `.iso` file, launch the installer by opening `WPILibInstaller.exe`.

Note

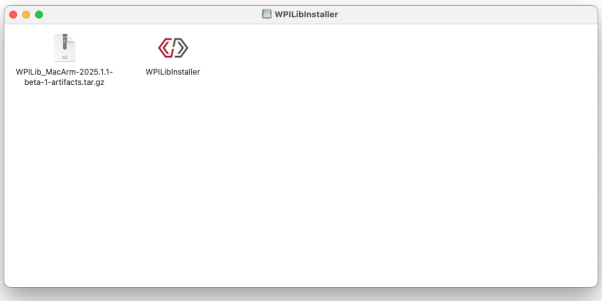
After launching the installer, Windows may display a window titled "Windows protected your PC". Click *More info*, then select *Run anyway* to run the installer.

macOS

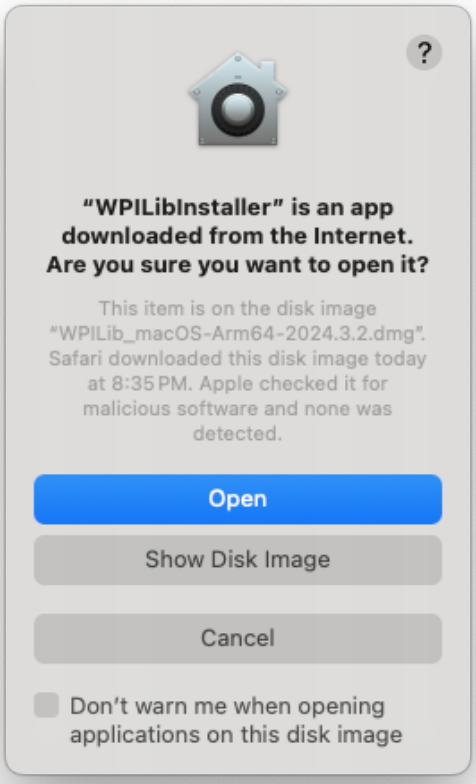
Note

Ensure you've ejected any previous WPILibInstaller images from the desktop before starting installation

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



If a warning is shown about WPILibInstaller being downloaded from the internet, select *Open*.



Linux

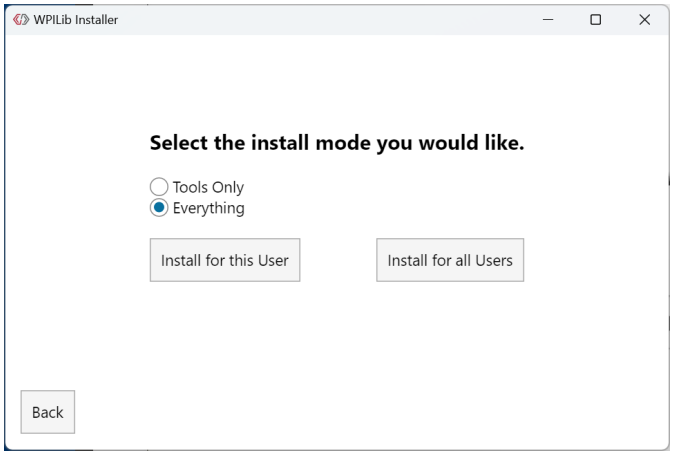
Linux users should extract the downloaded `.tar.gz` and then launch `WPILibInstaller`. Ubuntu treats executables in the file explorer as shared libraries, so double-clicking won't run them. Run the following commands in a terminal instead with `<version>` replaced with the version you're installing.

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

3.4.4 Running the Installer

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





This showcases a list of options included with the WPILib installation.

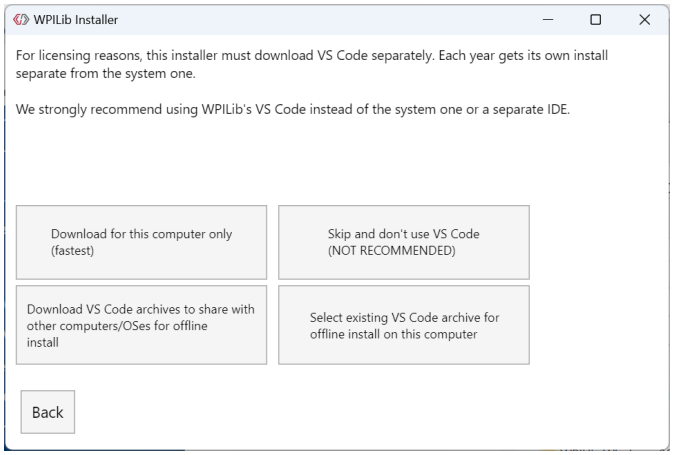
- *Tools Only* installs just the WPILib tools (*SmartDashboard*, *Shuffleboard*, *RobotBuilder*, *OutlineViewer*, *PathWeaver*, *Glass*, *SysId*, *Data Log Tool*, *roboRIO Team Number Setter*, *AdvantageScope*, *Elastic*, *WPIcal*) and JDK.
- *Everything* installs the full development environment (VS Code, extensions, all dependencies, C++ compiler and JDK), WPILib tools, and documentation

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.

> **Note**
>
> If you select Install for all Users, Windows will prompt for administrator access through UAC during installation.

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

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

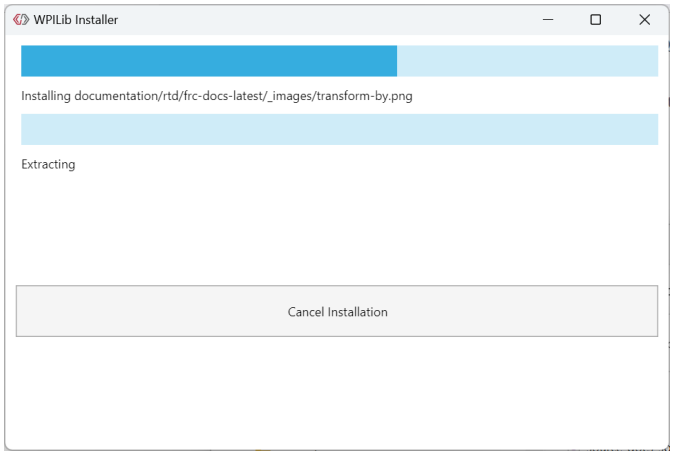


- Download for this computer only
  - This downloads VS Code only for the current platform, which is also the smallest download.
- Skip and don't use VS Code
  - Skips installing VS Code. Useful for advanced installations or configurations. Generally not recommended.
- Select existing VS Code archive for offline install on this computer
  - 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.
- Create VS Code archives to share with other computers/OSes for offline install
  - This option downloads and saves a copy of VS Code for all platforms, which is useful for sharing with the installer for future offline installs.

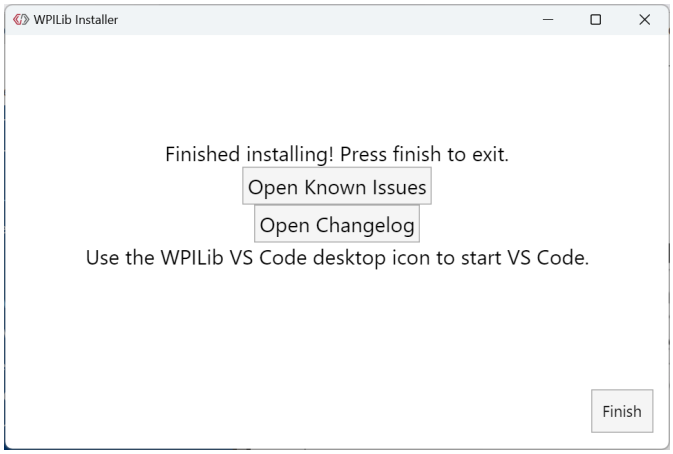
Go ahead and select *Download for this computer only*. This will begin the download process and can take a bit depending on internet connectivity (it's ~150MB). Once the download is done, select *Next*. You should be presented with a screen that looks similar to the one below.

Note

teams upgrading from Beta will already have a version of VS Code installed. However, it's still recommended to select *Download for this computer only* to ensure the latest recommended version of VS Code is installed



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



Important

WPILib installs a separate version of VS Code. It does not use an already existing installation. Each year has it's own copy of the tools appended with the year. IE: `WPILib VS Code 2026`. 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.

3.4.5 Post-Installation

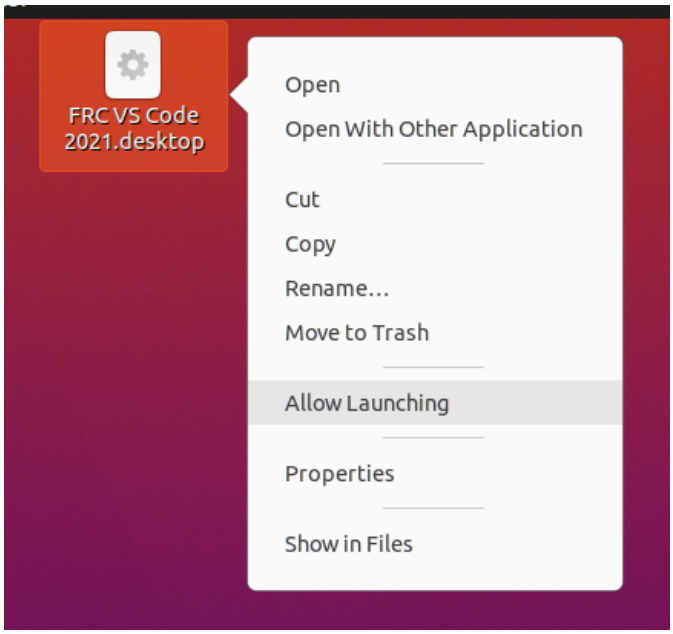
Some operating systems require some final action to complete installation.

macOS

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

Linux

Some versions of Linux (e.g. Ubuntu 22.04 and later) require you to give the desktop shortcut the ability to launch. Right click on the desktop icon and select Allow Launching.



Ubuntu 23.10 and later disable the kernel user namespaces feature for unknown applications. This means that the sandboxing feature won't work on the WPILib VS Code. To enable sandboxing for the WPILib applications, AppArmor profiles are provided, and can be installed using the command below.

```
$ sudo cp ~/wpilib/YEAR/frccode/AppArmor/* /etc/apparmor.d/
$ sudo systemctl reload apparmor.service
```

The above will fix the following error:

```
The SUID sandbox helper binary was found, but is not configured correctly. Rather␣
↪than run without sandboxing I'm aborting now. You need to make sure that ~/wpilib/
↪2026/advantagescope/chrome-sandbox is owned by root and has mode 4755.
```

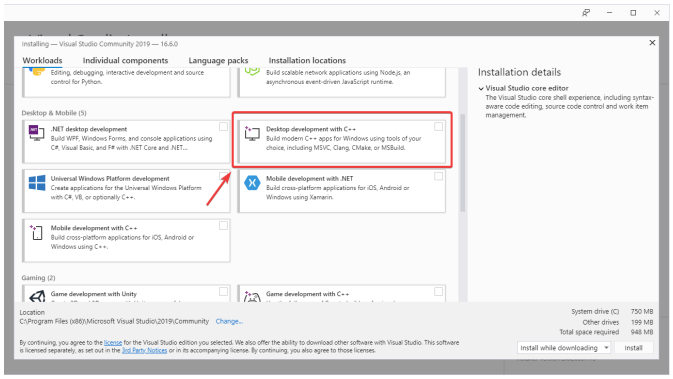
Note

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.

3.4.6 Additional C++ Installation for Simulation

C++ robot simulation requires that a native compiler to be installed. For Windows, this would be Visual Studio 2022 version 17.9 or later (**not** VS Code), macOS requires Xcode 14 or later, 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.



3.4.7 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*, *OutlineViewer*, *PathWeaver*, *Glass*, *SysId*, *Data Log Tool*, *roboRIO Team Number Setter*, *AdvantageScope*, *Elastic*, *WPIcal*
- **WPILib Dependencies** - OpenCV, etc.
- **VS Code Extensions** - WPILib and Java/C++/Python extensions for robot code development in VS Code

- **Documentation** - Offline copies of this frc-docs documentation and Java/C++/Python APIs

> **Note**
>
> It's not recommended to use the VS Code Backup and Settings Sync feature to sync settings between a regular VS Code installation and the FRC VS Code installation. This could lead to either installation being broken.

3.4.8 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

1. Delete the appropriate wpilib folder (`c:\Users\Public\wpilib\YYYY` where `YYYY` is the year to uninstall)
2. Delete the desktop icons at `C:\Users\Public\Public Desktop`

macOS

1. Delete the appropriate wpilib folder (`~/wpilib/YYYY` where `YYYY` is the year to uninstall)

Linux

1. Delete the appropriate wpilib folder (`~/wpilib/YYYY` where `YYYY` is the year to uninstall). eg `rm -rf ~/wpilib/YYYY`

3.4.9 Troubleshooting

In case the installer fails, please open an issue on the installer repository. A link is available here. The installer should give a message on the cause of the error, please include this in the description of your issue.

3.5 Python Installation Guide

This guide is intended for Python teams. Java and C++ teams can skip to *WPILib Installation Guide*. LabVIEW teams can skip to *Installing LabVIEW for FRC (LabVIEW only)*.

3.5.1 Prerequisites

You must install a supported version of Python on a supported operating system. Every year we upgrade RobotPy to the latest available version of Python. In 2026 we support Python 3.10/3.11/3.12/3.13/3.14, but only 3.14 is available for the roboRIO.

Supported Operating Systems and Architectures:

- Windows 10 & 11, 64 bit only. 32 bit and Arm are not supported
- macOS 13.3 or higher
- Ubuntu 22.04 & 24.04, 64 bit. Other Linux distributions with glibc >= 2.35 may work, but are unsupported

On Windows and macOS, we recommend using the official Python installers distributed by python.org.

- Python for Windows
- Python for macOS

3.5.2 Install RobotPy

Once you have installed Python, you can use pip to install RobotPy on your development computer.

Windows

Note

If you previously installed a pre-2024 or 2024 beta version of RobotPy, you should first uninstall RobotPy via `py -m pip uninstall robotpy` before upgrading.

Warning

On Windows, the Visual Studio 2022 redistributable package is required to be installed.

Run the following command from cmd or Powershell to install the core RobotPy packages:

```
py -3 -m pip install robotpy
```

To upgrade, you can run this:

```
py -3 -m pip install --upgrade robotpy
```

If you don't have administrative rights on your computer, either use virtualenv/virtualenvwrapper-win, or or you can install to the user site-packages directory:

```
py -3 -m pip install --user robotpy
```

macOS

> **Note**
>
> If you previously installed a pre-2024 or 2024 beta version of RobotPy, you should first uninstall RobotPy via `python3 -m pip uninstall robotpy` before upgrading.

On a macOS system that has pip installed, just run the following command from the Terminal application (may require admin rights):

```
python3 -m pip install robotpy
```

To upgrade, you can run this:

```
python3 -m pip install --upgrade robotpy
```

If you don't have administrative rights on your computer, either use virtualenv/virtualenvwrapper, or you can install to the user site-packages directory:

```
python3 -m pip install --user robotpy
```

Linux

> **Note**
>
> If you previously installed a pre-2024 or 2024 beta version of RobotPy, you should first uninstall RobotPy via `python3 -m pip uninstall robotpy` before upgrading.

RobotPy distributes manylinux binary wheels on PyPI. However, installing these requires a distro that has glibc 2.35 or newer, and an installer that implements **PEP 600**, such as pip 20.3 or newer. You can check your version of pip with the following command:

```
python3 -m pip --version
```

If you need to upgrade your version of pip, it is highly recommended to use a virtual environment.

If you have a compatible version of pip, you can simply run:

```
python3 -m pip install robotpy
```

To upgrade, you can run this:

```
python3 -m pip install --upgrade robotpy
```

If you manage to install the packages and get the following error or something similar, your system is most likely not compatible with RobotPy:

```
OSError: /usr/lib/x86_64-linux-gnu/libstdc++.so.6: version `GLIBCXX_3.4.22' not found␣
↪(required by /usr/local/lib/python3.7/dist-packages/wpiutil/lib/libwpiutil.so)
```

Linux ARM Coprocessor

We publish prebuilt wheels on artifactory, which can be downloaded by giving the `--extra-index-url` option to pip:

```
python3 -m pip install --extra-index-url=https://wpilib.jfrog.io/artifactory/api/pypi/
↪wpilib-python-release-2026/simple robotpy
```

source install

Alternatively, if you have a C++20 compiler installed, you may be able to use pip to install RobotPy from source.

> **Warning**
>
> It may take a very long time to install!

> **Warning**
>
> Mixing our pre-built wheels with source installs may cause runtime errors. This is due to internal ABI incompatibility between compiler versions.
>
> Our ARM wheels are built for Debian 12 (Bookworm) with GCC 12.

If you need to build with a specific compiler version, you can specify them using the `CC` and `CXX` environment variables:

```
export CC=gcc-12 CXX=g++-12
```

3.5.3 Download RobotPy for roboRIO

After installing the `robotpy` project on your computer, there are a variety of commands available that can be ran from the command line via the `robotpy` module.

> **See also**
>
> *Documentation for robotpy subcommands*

If you already have a RobotPy robot project, you can use that to download the pieces needed to run on the roboRIO. If you don't have a project, running this command in an empty directory will initialize a new robot project:

Windows

```
py -3 -m robotpy init
```

macOS

```
python3 -m robotpy init
```

Linux

```
python3 -m robotpy init
```

This will create a `robot.py` and `pyproject.toml` file. The `robot.py` file contains a skeleton structure to help you get started writing your robot code. The `pyproject.toml` file should be customized and details the requirements needed to run your robot code, among other things.

Important

The `robotpy deploy` command requires that you have written working robot code. The generated `robot.py` is just a starting point - you cannot deploy to the roboRIO until you have implemented your robot-specific code. See the *RobotPy Programming Guide* for information on writing robot code.

See also

The default `pyproject.toml` created for you only contains the version of RobotPy installed on your computer. If you want to enable vendor packages or install other python packages from PyPI, see our *pyproject.toml documentation*

Next run the `robotpy sync` subcommand, which will:

- Download Python compiled for roboRIO
- Download roboRIO compatible python packages as specified by your `pyproject.toml`
- Install the packages specified by your `pyproject.toml` into your local environment

Note

If you aren't using a virtualenv and don't have administrative privileges, the `robotpy sync` command accepts a `--user` argument to install to the user-specific site-packages directory.

Windows

```
py -3 -m robotpy sync
```

macOS

```
python3 -m robotpy sync
```

Linux

```
python3 -m robotpy sync
```

When you deploy your code to the roboRIO, *the deploy subcommand* will automatically install Python (if needed) and your robot project requirements on the roboRIO as part of the deploy process.

3.6 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.

4 Step 3: Preparing Your Robot

4.1 Imaging your roboRIO 2

Note

The imaging instructions for the NI roboRIO 1.0 are *here*.

The NI roboRIO 2.0 boots from a microSD card configured with an appropriate boot image containing the NI Linux Real-Time OS, drivers, and libraries specific to FRC. The microSD card must be imaged with a laptop and an SD burner application per the instructions on this page.

Important

Imaging the roboRIO 2 directly with the roboRIO Imaging Tool is not supported.

4.1.1 microSD Requirements

The NI roboRIO 2.0 supports all microSD cards. It is recommended to use a card with 2GB or more of capacity.

4.1.2 Operation Tips

The NI roboRIO 2.0 requires a fully inserted microSD card containing a valid image in order to boot and operate as intended.

If the microSD card is removed while powered, the roboRIO will hang. Once the microSD card is replaced, the roboRIO will need to be restarted using the reset button, or be power cycled.

No damage will result from microSD card removal or insertion while powered, but best practice is to perform these operations while unpowered.

Warning

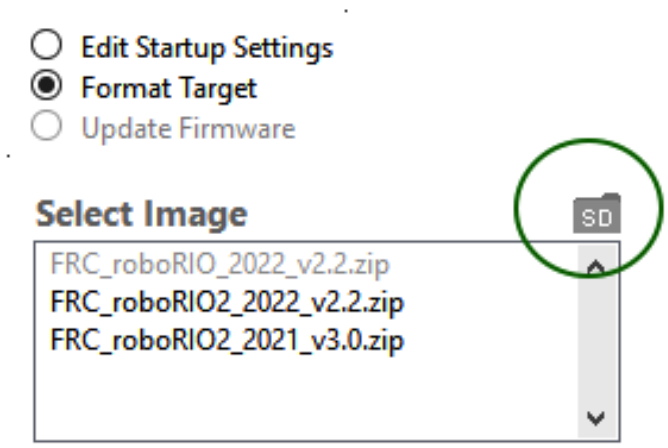
Before imaging your roboRIO, you must have completed installation of the *FRC Game Tools*. You also must have the roboRIO power properly wired to the CTRE Power Distribution Panel or REV Power Distribution Hub. Make sure the power wires to the roboRIO are secure and that the connector is secure firmly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

4.1.3 Imaging Directly to the microSD Card

The image will be transferred to the microSD card using a specialized writing utility, sometimes called a burner. Several utilities are listed below, but most tools that can write arbitrary images for booting a Raspberry Pi or similar dev boards will also produce a bootable SD card for roboRIO 2.0.

Supported image files are named `FRC_roboRIO2_YEAR_VERSION.img.zip`. You can locate them by clicking the SD button in the roboRIO Imaging tool and then navigating to the `SD Images` folder. It is generally best to use the latest version of the image.

If using a non Windows OS you will need to copy this image file to that computer.



A microSD to USB dongle works well for writing to microSD cards.

Note

Raspberry Pi images will not boot on a roboRIO because the OS and drivers are incompatible. Similarly, a roboRIO image is not compatible with Raspberry Pi controller boards.

Writing the image with balenaEtcher

- Download and install balenaEtcher.
- Launch
- *Flash from file* -> locate the image file you want to copy to the microSD card
- *Select target* -> select the destination microSD device
- Press *Flash*

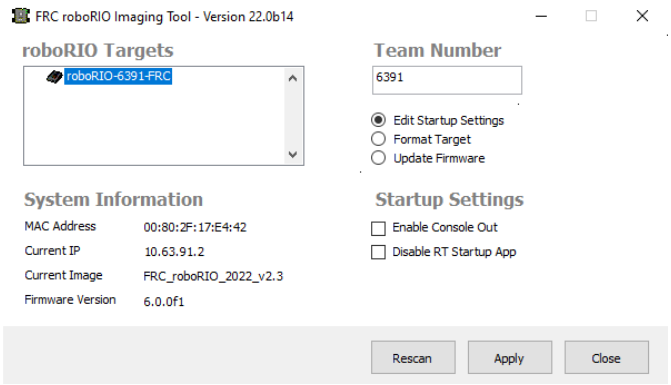
Writing the image with Raspberry Pi Imager

- Download and install from Raspberry Pi Imager.
- Launch
- *Choose OS* -> *Use Custom* -> select the image file you want to copy to the microSD card
- *Choose Storage* -> select the destination microSD device
- Press *Write*

> **Warning**
>
> After writing the image, Windows may prompt to format the drive. Do not reformat, or else you will need to write the image again.

Setting the roboRIO Team Number

The image writing process above does not set a team number. To fix this teams will need to insert the microSD card in the robRIO and connect to the robot. With the roboRIO Imaging Tool go to *Edit Startup Settings*. Next, fill out the *Team Number* box and hit *Apply*.



The team number can also be set using the *RoboRIO Team Number Setter*.

4.2 Imaging your roboRIO 1

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).

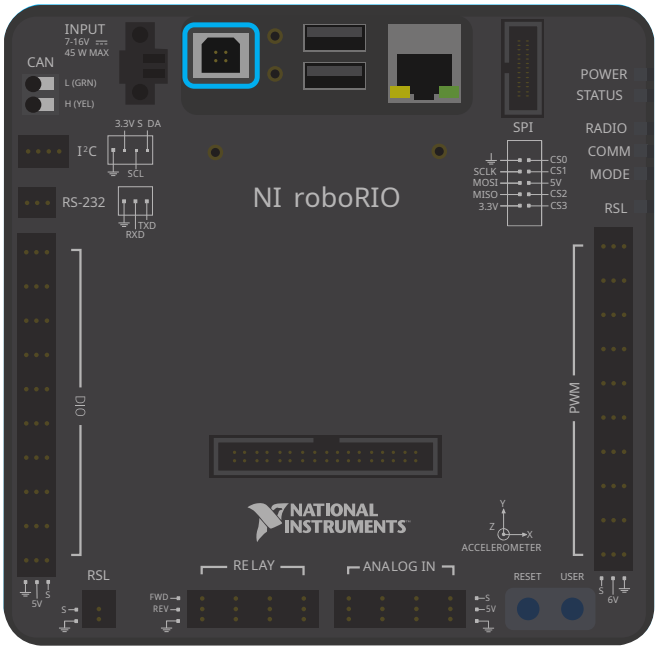
Note

The roboRIO 2 uses different imaging instructions. The imaging instructions for the NI roboRIO 2.0 are *here*.

4.2.1 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.

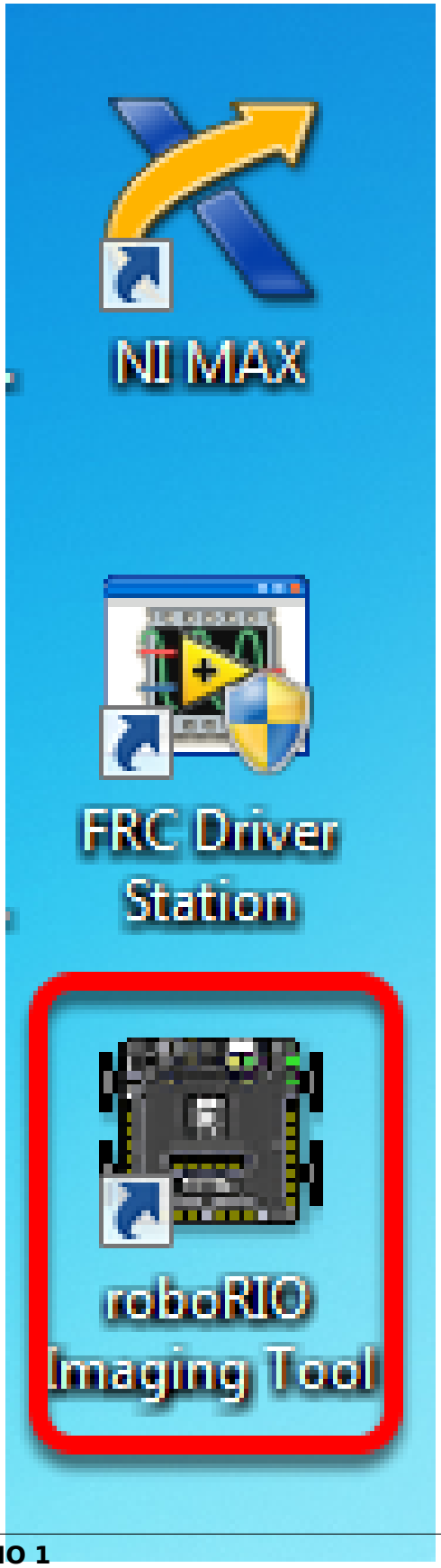
Note

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.

4.2.2 Launching the Imaging Tool

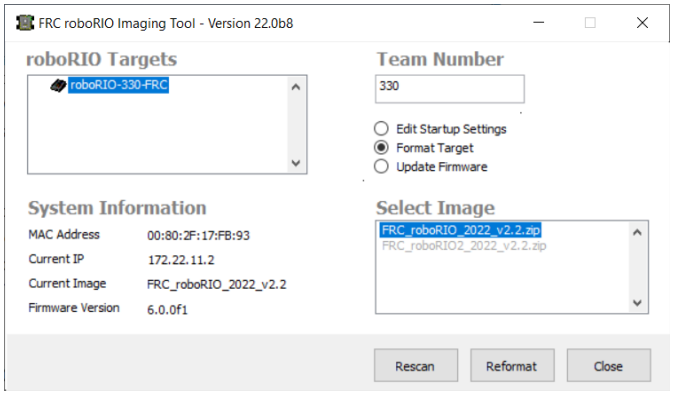


The roboRIO imaging tool and latest image are installed with the NI FRC® Game Tools. Launch the imaging tool by double clicking on the shortcut on the Desktop. If you have difficulties imaging your roboRIO, you may need to try right-clicking on the icon and selecting Run as Administrator instead.

Note

The roboRIO imaging tool is also located at `C:\Program Files (x86)\National Instruments\LabVIEW 2023\project\roboRIO Tool`

4.2.3 roboRIO Imaging Tool



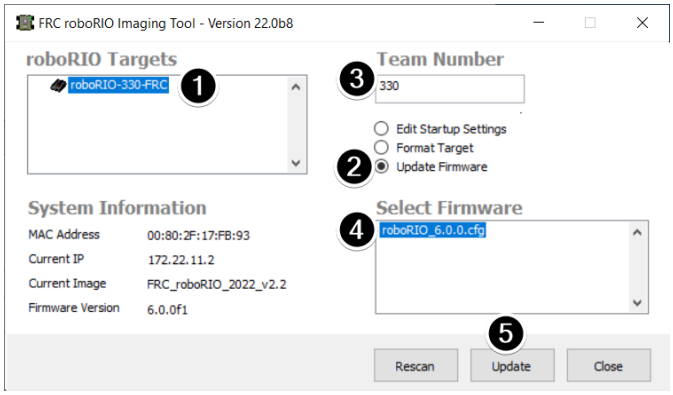
After launching, the roboRIO Imaging Tool will scan for available roboRIOs and indicate any found in the top left box. The bottom left box will show information and settings for the roboRIO currently selected. The right hand pane contains controls for modifying the roboRIO settings:

- **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

Warning

It is only necessary to update the firmware on a brand new roboRIO. It is not recommended to update the firmware unless it doesn't meet the conditions below.



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.

Note

roboRIO firmware has had different version numbering schemes over the years. It isn't necessary to update the firmware if it has version 5, 6, 8, 22.5, 23.5, 25.5 or variations of those version numbers (e.g. 8.8.0f0 is a variation of 8). The firmware is only utilized in *safe mode*, it is not used in normal operations.

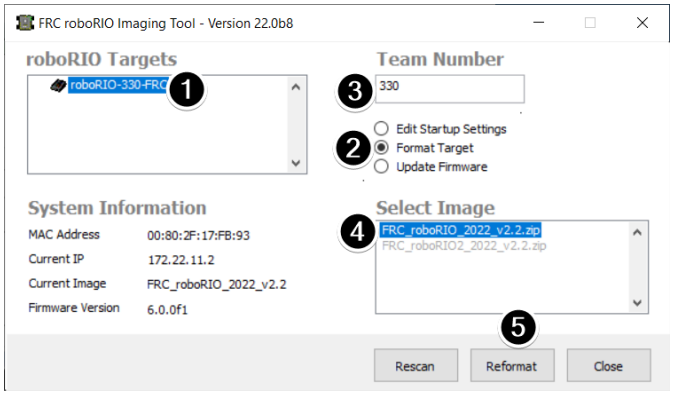
To update roboRIO firmware:

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

4.2.4 Imaging the roboRIO

Warning

The roboRIO image is different then the firmware, and must be updated yearly.

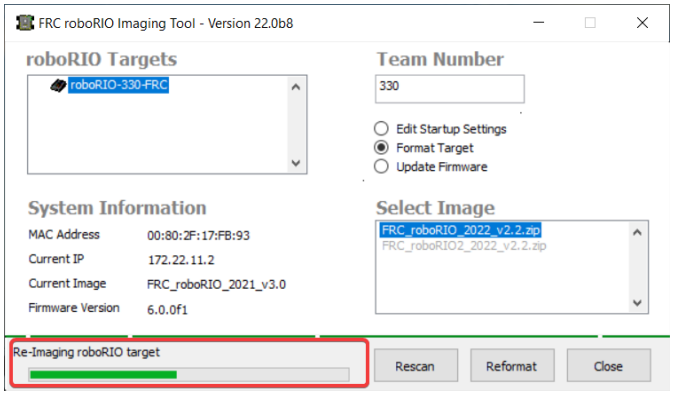


Note

The available image versions will not show until you select *Format Target* per step 2 below.

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

4.2.5 Imaging Progress



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

4.2.6 Imaging Complete



When the imaging completes you should see the dialog above. Click *Ok*, then click the :guilabel`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.

4.2.7 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.
- Use the *RoboRIO Team Number Setter* to ensure the Web Server is enabled.
- *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 special character (e.g. dash -), or number in it, or the name is too long. Try renaming the computer (or using a different PC). On Windows 11, to rename the PC, go to Settings > System > About and click *Rename this PC*
- Try booting the roboRIO into Safe Mode by pressing and holding the reset button for at least 5 seconds.
- Try a different USB Cable
- Try a different PC
- If the status LED is constantly flashing, and imaging in safe mode failed, follow the roboRIO recovery instructions

If the correct roboRIO image version isn't available:

- Ensure you've selected *Format Target*
- If an older version is shown, ensure you've installed the latest *FRC Game Tools*

- If the wrong version still shown after installing Game Tools, *Uninstall Game Tools* and then re-install.

4.3 Programming your Radio

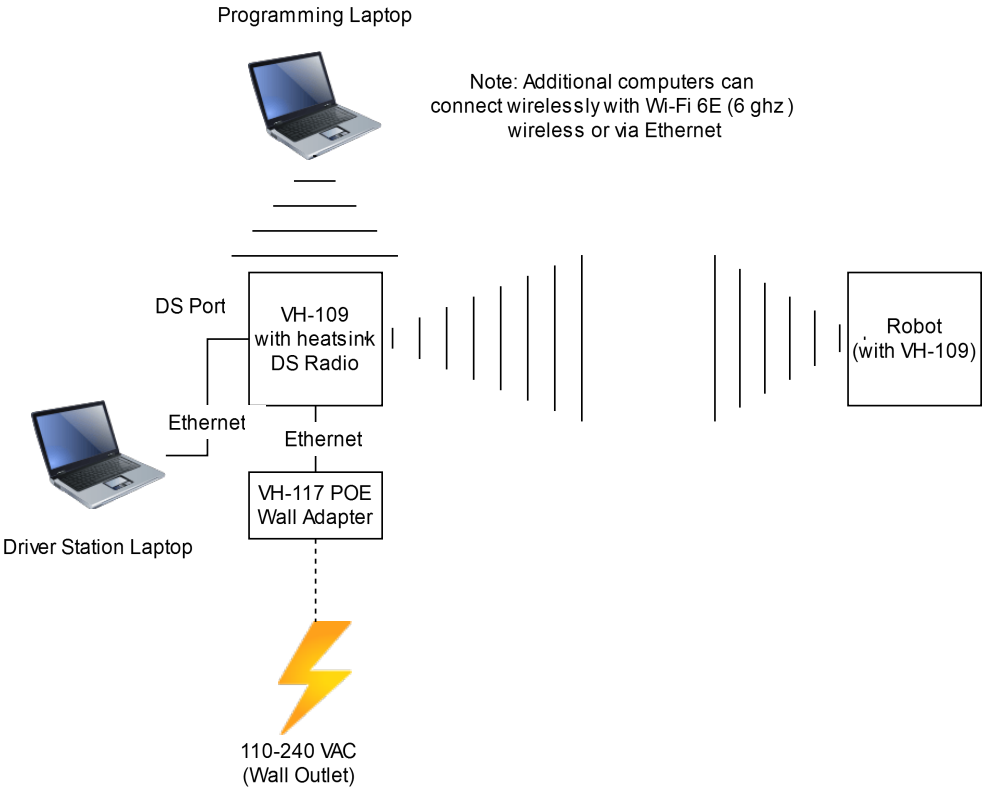
This guide will show you how to do a basic setup for controlling your robot "at home" using the VH-109 FRC® Radio. For complete documentation and information please see the Vivid-Hosting radio site.

> **Note**
>
> If looking for how to use the OpenMesh (OM5P) Radio please see *Using the OpenMesh OM5P Radio*

4.3.1 Prerequisites

The recommended setup requires: 2 VH-109 radios, 1 VH-117 *PoE* Wall Adapter, and 1 Radio Heatsink. Available here.

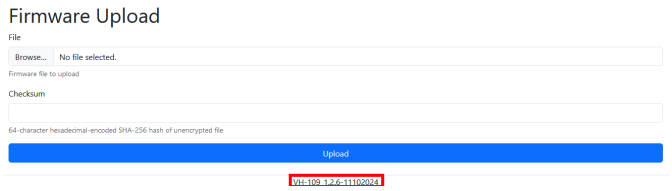
Please see the *Alternative Setup Discussion* if you do not currently have this hardware.



4.3.2 Getting to the Web Configuration Page

1. Connect the radio directly to your computer using an Ethernet cable in the *DS* port.
2. Ensure the radio has power either through the Weidmuller connectors or *PoE*.
3. Open a web browser and navigate to *http://radio.local/*. See *Troubleshooting* if the connection doesn't work.

4.3.3 Radio Firmware Update



Note

Newer versions of the radio firmware will have the version listed at the bottom. If you do not see the version listed you have firmware prior to 1.2.0.

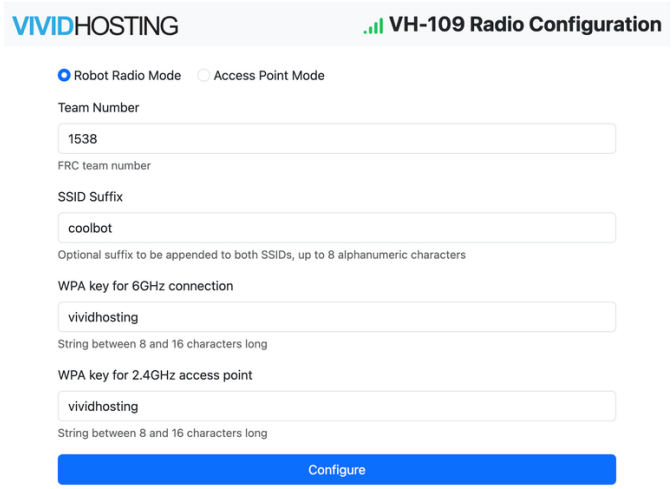
You can skip this section if already at version 1.3.0 or later.

1. On the Vivid-Hosting firmware releases page download the proper firmware for the current firmware version you have. Always choose the *Radio Variant*.
2. Copy the SHA-256 key below the firmware you downloaded.
3. Paste that key into the *Checksum* box of the *Firmware Upload* section at the bottom of the configuration page we navigated to above.
4. Click *Browse...* and select the firmware file you downloaded.
5. Click the *Upload* button.

Warning

The radio will take approximately 2-3 minutes to complete firmware updates. Do not remove power during this process. Damage to the radio can occur. When the PWR light is solid and the SYS light is slowly blinking at 1 Hz, the firmware upgrade process is complete.

4.3.4 Robot Radio Configuration (All Teams)



This section is used for configuring the VH-109 radio outside of competition. At competition, configuration will be done by a provided computer and manual configuration using this page **should not be used**.

1. Select *Robot Radio Mode*
2. Enter the team number
3. Enter the suffix, if desired. This will help identify your robot and distinguish it from other networks.
4. Enter the 6 GHz WPA/SAE key. This key will need to match the key on the Access Point you configure.
5. Enter the 2.4 GHz WPA/SAE key. This is the password team members will type in when connecting to the 2.4 GHz network, if available.

4.3.5 Access Point Radio Configuration

On the Access Point Radio, follow all of the same steps as the robot radio configuration instead choosing *Access Point Mode* at the top of the configuration page. Ensure you use the exact same settings for team number, suffix, and WPA/SAE keys.

4.3.6 Alternative Setup Discussion

Optimal Setup: Two VH-109 Radios

For the best experience and to closely simulate field conditions, it is strongly recommended that your team uses two VH-109 radios during testing and preparation. This dual-radio setup mirrors the competition environment, ensuring your robot operates under realistic network conditions. Additionally, having two radios allows you to fully leverage the high-speed, low-latency communication provided by the 6GHz band, which is crucial for optimal robot performance in high-stakes scenarios.

When mounting the access point radio, ensure it is mounted high where it has a clear line of sight to the robot.

Only 1 VH-109 radio

If your team has access to only one VH-109 radio, there are still viable options to continue testing and preparing for competition. However, these setups require additional considerations:

Use an Old Radio for Testing

If your team still has an older radio from a previous season, it can serve as a temporary substitute for the VH-109 on your robot. In this case, you should:

- Reserve a spot on your robot specifically for the VH-109 radio to ensure seamless integration during competition.
- Provide the older radio regulated power with a REV Radio Power Module or a CTRE Voltage Regulator Module.
- Be prepared to connect devices via a network switch if the older radio does not provide enough Ethernet ports. This may add complexity but ensures all devices are networked properly during testing.

Advantages:

- Connection strength similar to previous years.

Disadvantages:

- Requires additional hardware (e.g., the old radio and maybe a network switch).
- The older radio may not offer the same performance as the VH-109, potentially affecting test results.

Enable 2.4GHz Wifi on the VH-109

The VH-109 radio includes a standalone mode that can be activated using DIP switch 3. In this configuration, the radio hosts its own 2.4GHz network, enabling direct connections without additional hardware.

To enable standalone mode:

1. Locate the DIP switches on the VH-109 radio, removing the sticker if required.
2. Flip DIP switch 3 to the “ON” position.
3. Connect devices directly to the 2.4GHz network hosted by the radio, using the 2.4 GHz WPA/SAE key from configuration.

> **⚠ Warning**
>
> Due to poor performance, Vivid-Hosting does not recommend using this method for practicing at home. You should only use this method in extreme cases. School campuses tend to have noisy 2.4GHz environments, which can significantly degrade wireless performance.

Advantages:

- Simple setup with no need for additional hardware.

- Allows immediate use of the VH-109 without extra configuration.

Disadvantages:

- The 2.4GHz band is more prone to congestion and interference, especially in crowded environments.
- Range and accessibility may be limited compared to the 6GHz band.

No VH-117 PoE Wall Adapter

You can power the access point radio with a 12V wall plug connecting wires to the Weidmuller ports. We recommend trying to find one with a switch to simplify turning on and off the radio. The primary concern will be cord length which will likely not be long enough to run from your wall outlet so you will need to bring the power closer with an extension cord.

No Radio Heatsink

The access point radio will get hot after being on for a longer than a full match. This will cause latency to increase.

4.3.7 Troubleshooting

Cannot Reach the Configuration Page at radio.local

Disconnect other network connections such as Wi-Fi.

Download and use the Network Assistant. See the instructions below the download for how to use.

Disable *firewalls*.

Ensure *Game Tools* or another *mdns responder* is installed.

Set a static ip address with these parameters:

- IP Address: 192.168.69.2
- Subnet Mask: 255.255.255.0
- Gateway: Leave Blank
- DNS: 192.168.69.1 or Leave Blank

Navigate to *http://192.168.69.1/*

Setting Up an Entire Field

See the documentation on using the VH-113 Access Point.

How do I run an Offseason Event?

Use the Vivid-Hosting radio kiosk.

Additional Radio Problems

Contact WCP support at: support@westcoastproducts.com

4.4 Using the OpenMesh OM5P Radio

Prior to 2025 the official radio used in FRC was an OM5P-AN or OM5P-AC Radio. As of the 2026 season the OpenMesh radio is still being used in Taiwan and China.

> **Warning**
>
> This page is either for teams looking to use the OM5P radio to test or teams in China and Taiwan. For the current robot radio please see *Programming your Radio*



Either the OpenMesh OM5P-AN or OpenMesh OM5P-AC wireless radio was used as the robot radio to provide wireless communication functionality to the robot. The device can still 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.

4.4.1 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.

4.4.2 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:

FRC Radio Configuration 25.0.0

FRC Radio Configuration 25.0.0 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.

Note

Teams planning on using the radio tool on a machine without the **2025** WPILib suite installed, will need a copy of Java installed. This can be downloaded here.

Before you begin using the software:

1. *Disabling Network Adapters*
2. Plug directly from your computer into the wireless bridge ethernet port closest to the power jack. Make sure no other devices are connected to your computer via ethernet. If powering the radio via PoE, plug an Ethernet cable from the PC into the socket side of the PoE adapter (where the roboRIO would plug in). If you experience issues configuring through the PoE adapter, you may try connecting the PC to the alternate port on the radio.

Warning

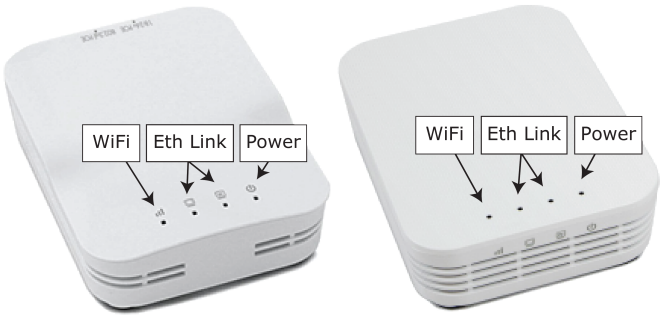
Many barrel connectors look alike, however they are 12V radios. Wire the radio to the 12V 2A terminals on the VRM (center-pin positive).

4.4.3 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, 10 and 11. It may work on other operating systems, but has not been tested.

4.4.4 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.

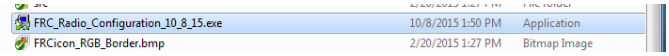
At home only:

- 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.

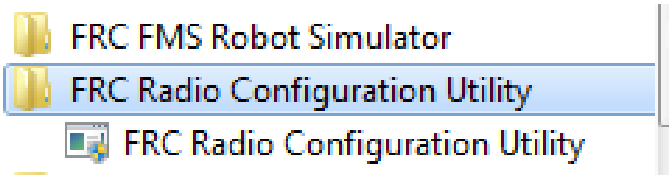
4.4.5 Install the Software



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.

4.4.6 Launch the software

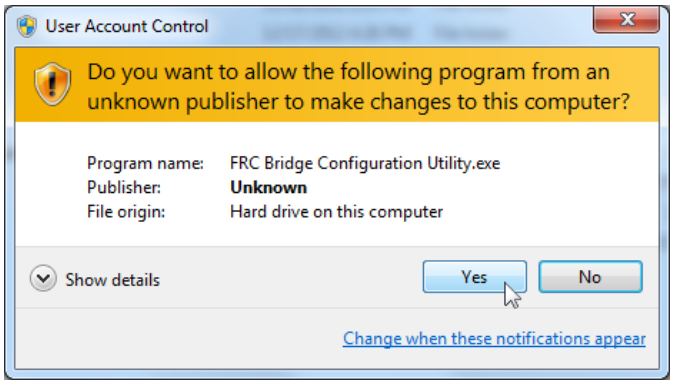


Use the Start menu or desktop shortcut to launch the program.

Note

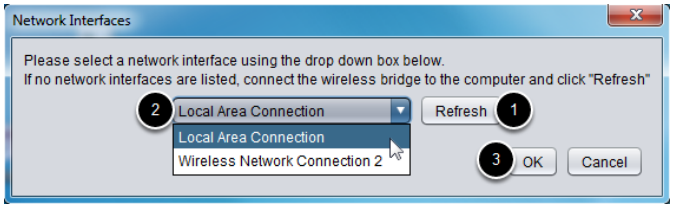
If you need to locate the program, it is installed to `C:\Program Files (x86)\FRC Radio Configuration Utility`. For 32-bit machines the path is `C:\Program Files\FRC Radio Configuration Utility`

4.4.7 Allow the program to make changes, if prompted



A prompt may appear about allowing the configuration utility to make changes to the computer. Click *Yes* if the prompt appears.

4.4.8 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*.

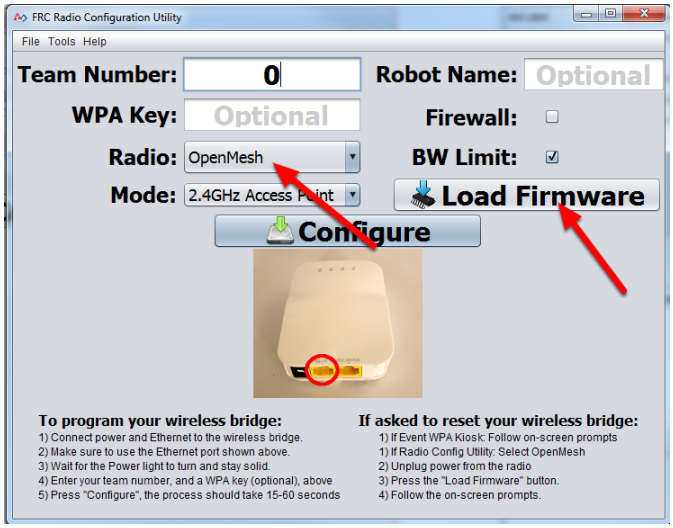
4.4.9 Open Mesh Firmware Note

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.

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

> **Warning**
>
> Radios used in 2019-2024 **do not** need to be updated before configuring, the 2024 tool uses the same 2019 firmware.

4.4.10 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.

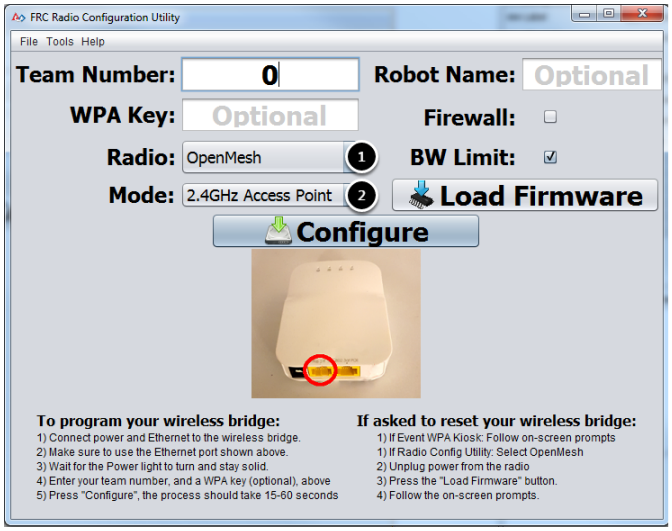
1. Follow the instructions above to install the software, launch the program and select the Ethernet interface.
2. Make sure the Open Mesh radio is selected in the Radio dropdown.
3. Make sure the radio is connected to the PC via Ethernet.
4. Unplug the power from the radio. (If using a PoE cable, this 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.

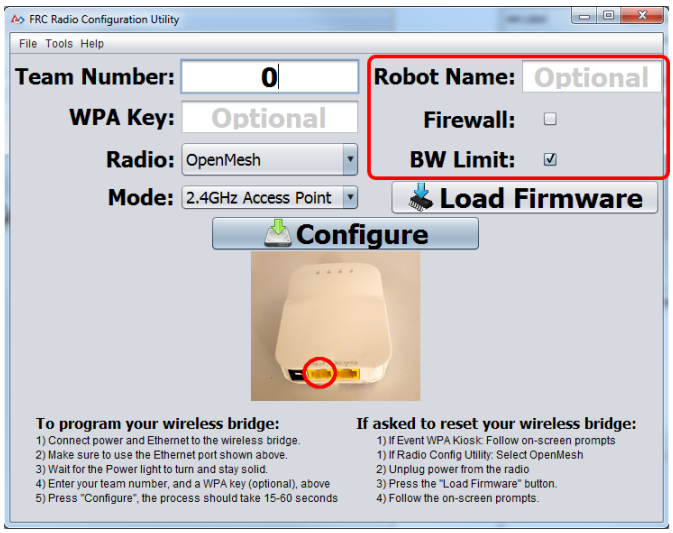
Teams may also see this error with Operating Systems configured for languages other than US English. If you experience issues loading firmware or programming on a foreign language OS, try using an English OS, such as on the KOP provided PC or setting the Locale setting to “en_us” as described on [this page] (https://www.java.com/en/download/help/locale.html).

4.4.11 Select Radio and Operating Mode



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.

4.4.12 Select Options



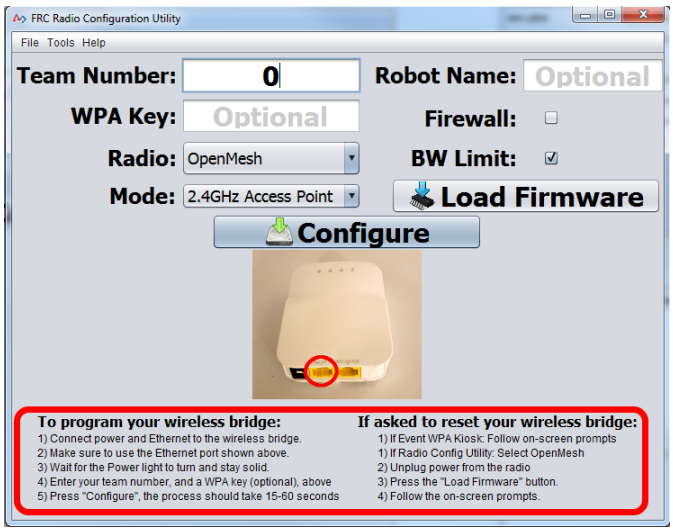
The default values of the options have been selected to match the use case of most teams, however, you may wish to customize these options to your specific scenario:

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 this is a total limit, not per client, so streaming video to multiple clients simultaneously may cause undesired behavior.

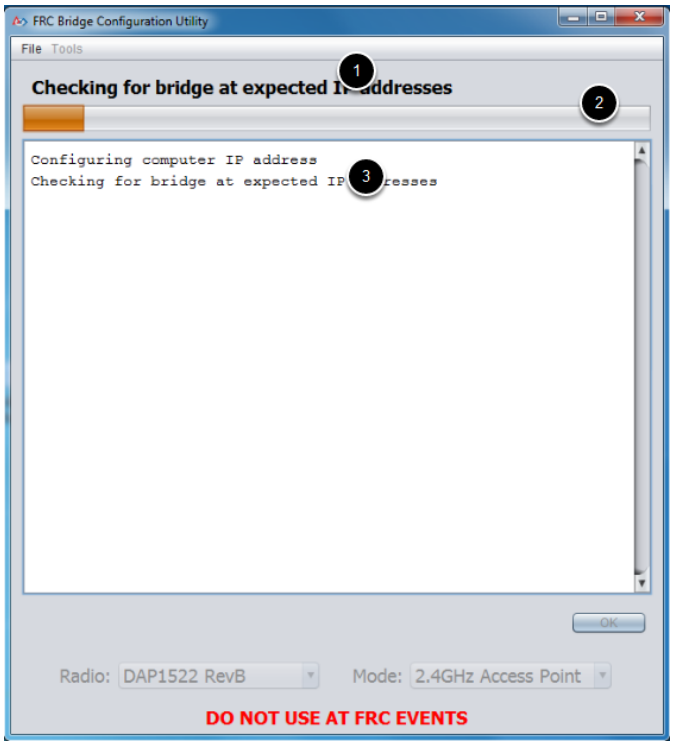
> **Warning**
>
> The "Firewall" option configures the radio to emulate the field firewall. This means that you will not be able to deploy code wirelessly with this option enabled. This is useful for simulating blocked ports that may exist at competitions.

4.4.13 Starting the Configuration Process



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.

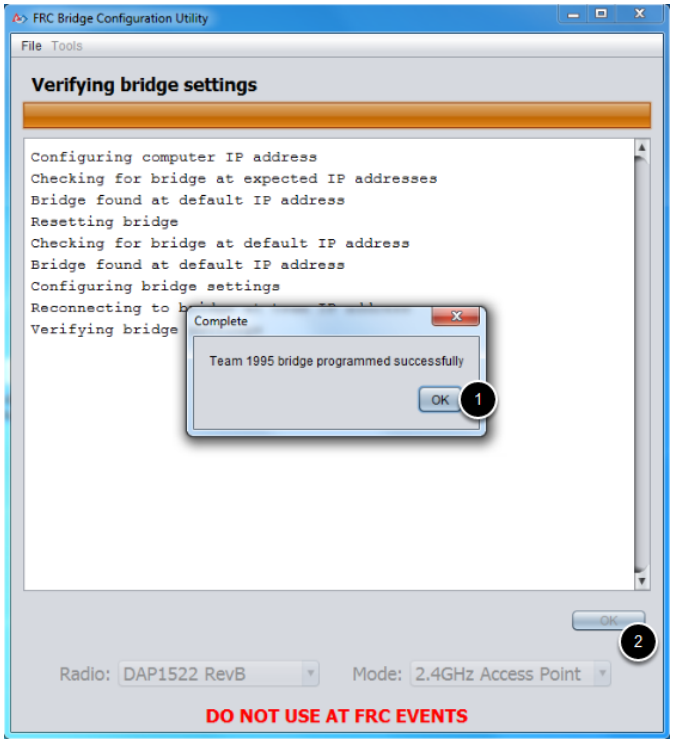
4.4.14 Configuration Progress



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.

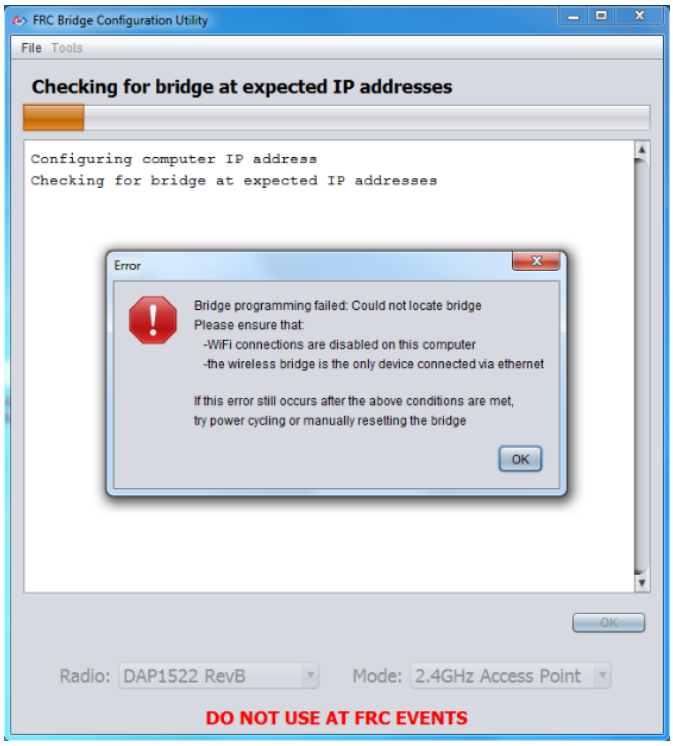
4.4.15 Configuration Completed



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.

4.4.16 Configuration Errors



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

4.4.17 Troubleshooting

- *Disabling 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.
- Make sure your firewall is turned off.
- 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 an Operating System configured for languages other than US English, 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.
- Due to Unicode incompatibilities, non-US Teams may face a configuration failure because of incorrect network interface reading. In that case, change the network adapter name to another name in English and retry.
- Some users have reported success after installing npcap 1.60. If this doesn't resolve the issue, it's recommended to uninstall npcap and the radio tool and then reinstall the radio tool in order to get back to a known configuration.

- If all else fails, try a different computer.

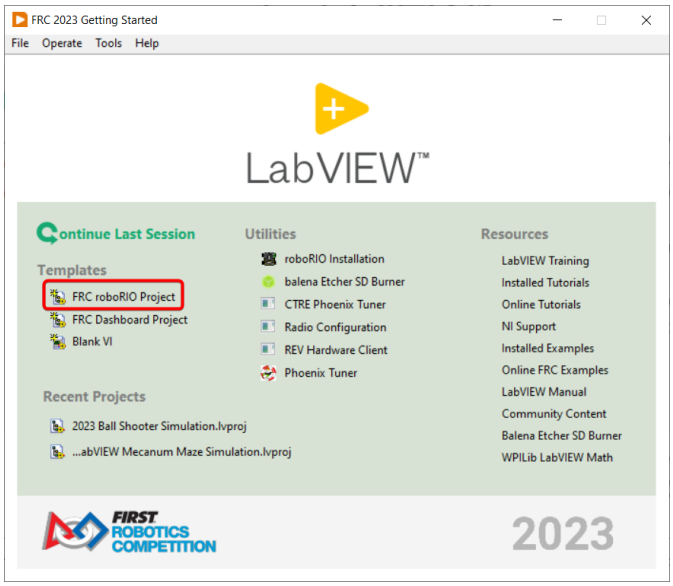
5 Step 4: Programming your Robot

5.1 Creating your Test Drivetrain Program (LabVIEW)

Note

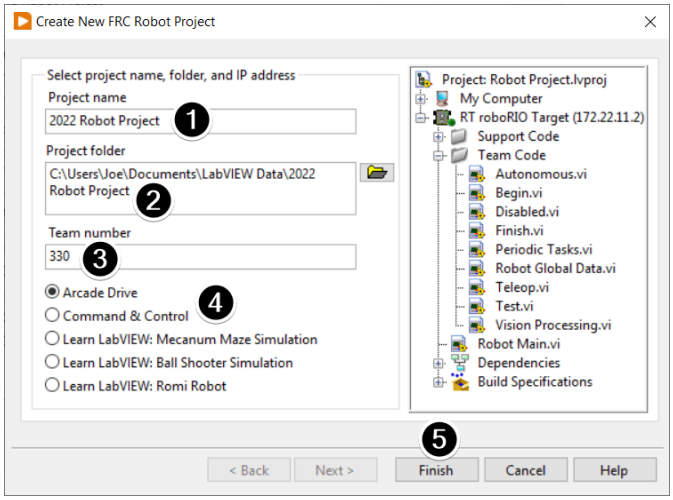
This document covers how to create, build, and load a basic FRC® LabVIEW program for a drivetrain onto a roboRIO. Before beginning, make sure that you have installed LabVIEW for FRC and the FRC Game Tools and that you have configured and imaged your roboRIO as described in the *Zero-to-Robot tutorial*.

5.1.1 Creating a Project



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

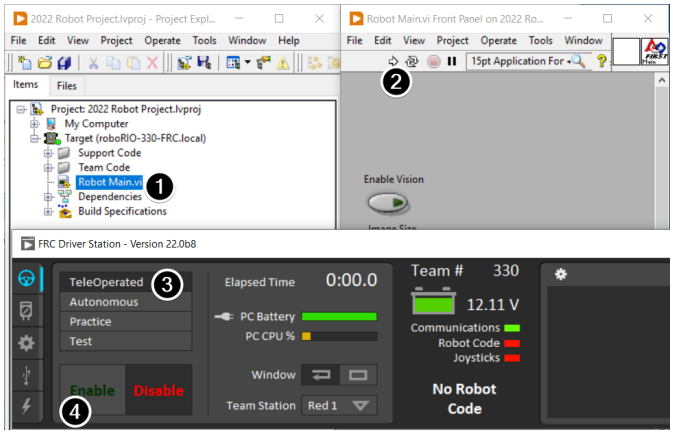
5.1.2 Configuring Project



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*.
5. Click *Finish*

5.1.3 Running the Program

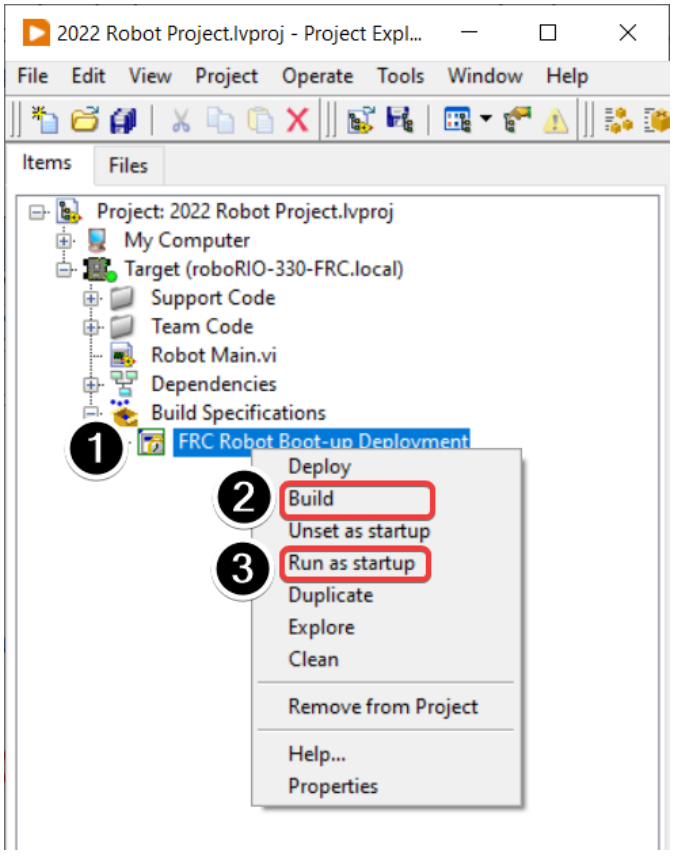


Note

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.

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 Test Drivetrain Program (Java/C++/Python)

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.

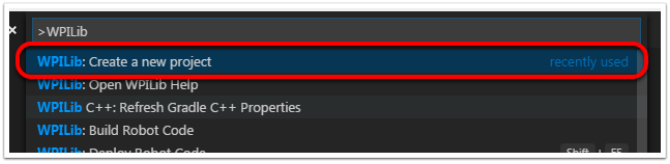
- *Creating a New WPILib Project (Java/C++)*
- *Creating a New WPILib Project (Python)*

Important

This guide includes code examples that involve vendor hardware for the convenience of the user. In this document, *PWM* refers to the motor controller included in the KOP. The CTRE tab references the Talon FX motor controller (Falcon 500 motor), but usage is similar for TalonSRX and VictorSPX. The REV tab references the CAN SPARK MAX controlling a brushless motor, but it's similar for brushed motor. There is an assumption that the user has already installed the required *vendordeps* and configured the device(s) (update firmware, assign CAN IDs, etc.) according to the manufacturer documentation (CTRE / REV).

5.2.1 Creating a New WPILib Project (Java/C++)

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 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. 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.

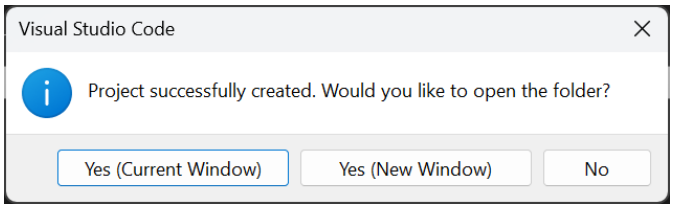
Note

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

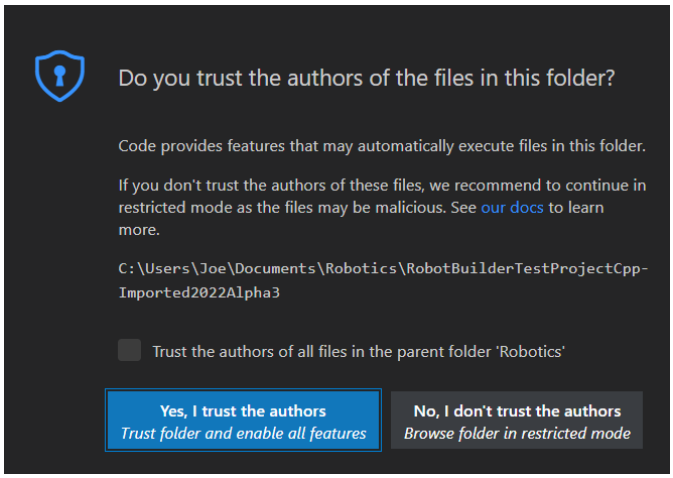
Warning

Creating projects on OneDrive is not supported as OneDrive's caching interferes with the build system. Some Windows installations put the Documents and Desktop folders on OneDrive by default.

5.2.2 Opening The New Project

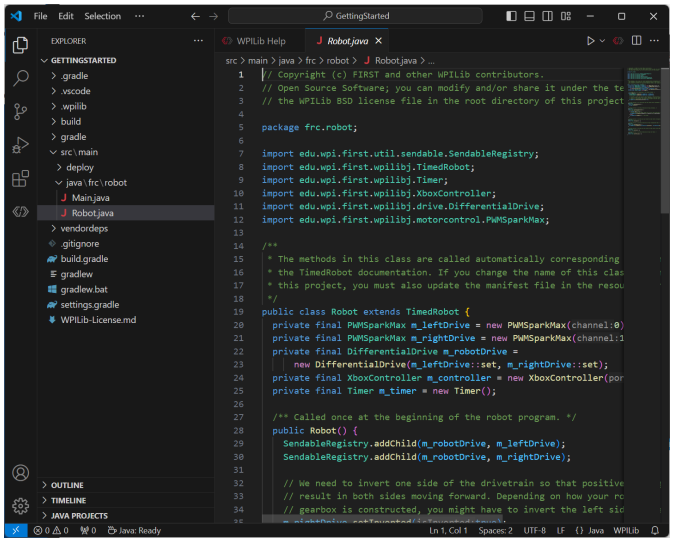


After successfully creating your project, VS Code will give the option of opening the project as shown above. We can choose to do that now or later by typing `Ctrl+K` then `Ctrl+O` (or just `Command+O` on macOS) and select the folder where we saved our project.



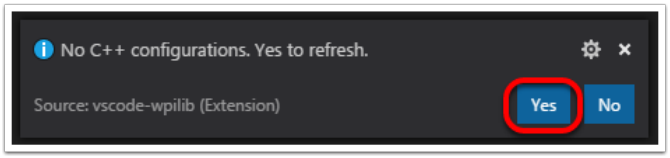
Click *Yes I trust the authors*.

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 Creating a New WPILib Project (Python)

Running the `robotpy init` command will initialize a new robot project:

Windows

```
py -3 -m robotpy init
```

macOS

```
python3 -m robotpy init
```

Linux

```
python3 -m robotpy init
```

This will create a `robot.py` and `pyproject.toml` file, but will not overwrite an existing file.

- The `pyproject.toml` file contains the requirements for your project, which are downloaded and installed via the `robotpy sync` command.
- The `robot.py` file is where you will put the your Robot class.

See also

Download RobotPy for roboRIO

5.2.5 Basic Drivetrain example

First, here is what a simple code can look like for a Drivetrain with PWM controlled motors (such as SparkMax).

Note

the Python example below is from https://github.com/robotpy/examples/tree/main/GettingStarted

JAVA

```
// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

package edu.wpi.first.wpilibj.examples.gettingstarted;

import edu.wpi.first.util.sendable.SendableRegistry;
import edu.wpi.first.wpilibj.TimedRobot;
import edu.wpi.first.wpilibj.Timer;
import edu.wpi.first.wpilibj.XboxController;
import edu.wpi.first.wpilibj.drive.DifferentialDrive;
import edu.wpi.first.wpilibj.motorcontrol.PWMSparkMax;

/**
 * The methods in this class are called automatically corresponding to each mode, as␣
↪described in
 * the TimedRobot documentation. If you change the name of this class or the package␣
↪after creating
 * this project, you must also update the manifest file in the resource directory.
 */
public class Robot extends TimedRobot {
  private final PWMSparkMax m_leftDrive = new PWMSparkMax(0);
  private final PWMSparkMax m_rightDrive = new PWMSparkMax(1);
  private final DifferentialDrive m_robotDrive =
```

(continues on next page)

(continued from previous page)

```
      new DifferentialDrive(m_leftDrive::set, m_rightDrive::set);
  private final XboxController m_controller = new XboxController(0);
  private final Timer m_timer = new Timer();

  /** Called once at the beginning of the robot program. */
  public Robot() {
    SendableRegistry.addChild(m_robotDrive, m_leftDrive);
    SendableRegistry.addChild(m_robotDrive, m_rightDrive);

    // We need to invert one side of the drivetrain so that positive voltages
    // result in both sides moving forward. Depending on how your robot's
    // gearbox is constructed, you might have to invert the left side instead.
    m_rightDrive.setInverted(true);
  }

  /** This function is run once each time the robot enters autonomous mode. */
  @Override
  public void autonomousInit() {
    m_timer.restart();
  }

  /** This function is called periodically during autonomous. */
  @Override
  public void autonomousPeriodic() {
    // Drive for 2 seconds
    if (m_timer.get() < 2.0) {
      // Drive forwards half speed, make sure to turn input squaring off
      m_robotDrive.arcadeDrive(0.5, 0.0, false);
    } else {
      m_robotDrive.stopMotor(); // stop robot
    }
  }

  /** This function is called once each time the robot enters teleoperated mode. */
  @Override
  public void teleopInit() {}

  /** This function is called periodically during teleoperated mode. */
  @Override
  public void teleopPeriodic() {
    m_robotDrive.arcadeDrive(-m_controller.getLeftY(), -m_controller.getRightX());
  }

  /** This function is called once each time the robot enters test mode. */
  @Override
  public void testInit() {}

  /** This function is called periodically during test mode. */
  @Override
  public void testPeriodic() {}
}
```

C++

```
// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.

#include <frc/TimedRobot.h>
#include <frc/Timer.h>
#include <frc/XboxController.h>
#include <frc/drive/DifferentialDrive.h>
#include <frc/motorcontrol/PWMSparkMax.h>

class Robot : public frc::TimedRobot {
 public:
  Robot() {
    wpi::SendableRegistry::AddChild(&m_robotDrive, &m_left);
    wpi::SendableRegistry::AddChild(&m_robotDrive, &m_right);

    // We need to invert one side of the drivetrain so that positive voltages
    // result in both sides moving forward. Depending on how your robot's
    // gearbox is constructed, you might have to invert the left side instead.
    m_right.SetInverted(true);
    m_robotDrive.SetExpiration(100_ms);
    m_timer.Start();
  }

  void AutonomousInit() override { m_timer.Restart(); }

  void AutonomousPeriodic() override {
    // Drive for 2 seconds
    if (m_timer.Get() < 2_s) {
      // Drive forwards half speed, make sure to turn input squaring off
      m_robotDrive.ArcadeDrive(0.5, 0.0, false);
    } else {
      // Stop robot
      m_robotDrive.ArcadeDrive(0.0, 0.0, false);
    }
  }

  void TeleopInit() override {}

  void TeleopPeriodic() override {
    // Drive with arcade style (use right stick to steer)
    m_robotDrive.ArcadeDrive(-m_controller.GetLeftY(),
                             m_controller.GetRightX());
  }

  void TestInit() override {}

  void TestPeriodic() override {}

 private:
  // Robot drive system
  frc::PWMSparkMax m_left{0};
  frc::PWMSparkMax m_right{1};
  frc::DifferentialDrive m_robotDrive{
      [&](double output) { m_left.Set(output); },
```

(continues on next page)

(continued from previous page)

```
      [&](double output) { m_right.Set(output); }};

  frc::XboxController m_controller{0};
  frc::Timer m_timer;
};

#ifndef RUNNING_FRC_TESTS
int main() {
  return frc::StartRobot<Robot>();
}
#endif
```

PYTHON

```
#!/usr/bin/env python3
#
# Copyright (c) FIRST and other WPILib contributors.
# Open Source Software; you can modify and/or share it under the terms of
# the WPILib BSD license file in the root directory of this project.
#

import wpilib
import wpilib.drive


class MyRobot(wpilib.TimedRobot):
    def robotInit(self):
        """
        This function is called upon program startup and
        should be used for any initialization code.
        """
        self.leftDrive = wpilib.PWMSparkMax(0)
        self.rightDrive = wpilib.PWMSparkMax(1)
        self.robotDrive = wpilib.drive.DifferentialDrive(
            self.leftDrive, self.rightDrive
        )
        self.controller = wpilib.XboxController(0)
        self.timer = wpilib.Timer()

        # We need to invert one side of the drivetrain so that positive voltages
        # result in both sides moving forward. Depending on how your robot's
        # gearbox is constructed, you might have to invert the left side instead.
        self.rightDrive.setInverted(True)

    def autonomousInit(self):
        """This function is run once each time the robot enters autonomous mode."""
        self.timer.restart()

    def autonomousPeriodic(self):
        """This function is called periodically during autonomous."""

        # Drive for two seconds
        if self.timer.get() < 2.0:
            # Drive forwards half speed, make sure to turn input squaring off
```

(continues on next page)

(continued from previous page)

```
            self.robotDrive.arcadeDrive(0.5, 0, squareInputs=False)
        else:
            self.robotDrive.stopMotor()  # Stop robot

    def teleopInit(self):
        """This function is called once each time the robot enters teleoperated mode."
↪""

    def teleopPeriodic(self):
        """This function is called periodically during teleoperated mode."""
        self.robotDrive.arcadeDrive(
            -self.controller.getLeftY(), -self.controller.getRightX()
        )

    def testInit(self):
        """This function is called once each time the robot enters test mode."""

    def testPeriodic(self):
        """This function is called periodically during test mode."""


if __name__ == "__main__":
    wpilib.run(MyRobot)
```

Now let's look at various parts of the code.

5.2.6 Imports/Includes

PWM

Java

```
import edu.wpi.first.util.sendable.SendableRegistry;
import edu.wpi.first.wpilibj.TimedRobot;
import edu.wpi.first.wpilibj.Timer;
import edu.wpi.first.wpilibj.XboxController;
import edu.wpi.first.wpilibj.drive.DifferentialDrive;
import edu.wpi.first.wpilibj.motorcontrol.PWMSparkMax;
```

C++

```
#include <frc/TimedRobot.h>
#include <frc/Timer.h>
#include <frc/XboxController.h>
#include <frc/drive/DifferentialDrive.h>
#include <frc/motorcontrol/PWMSparkMax.h>
```

Python

```
import wpilib
import wpilib.drive
```

CTRE-Phoenix6

JAVA

```
import edu.wpi.first.wpilibj.TimedRobot;
import edu.wpi.first.wpilibj.Timer;
import edu.wpi.first.wpilibj.XboxController;
import edu.wpi.first.wpilibj.drive.DifferentialDrive;
import com.ctre.phoenix6.hardware.TalonFX;
```

C++

```
#include <frc/TimedRobot.h>
#include <frc/Timer.h>
#include <frc/XboxController.h>
#include <frc/drive/DifferentialDrive.h>
#include <ctre/phoenix6/TalonFX.hpp>
```

PYTHON

```
import wpilib               # Used to get the joysticks
import wpilib.drive         # Used for the DifferentialDrive class
import phoenix6             # CTRE library
```

REV

JAVA

```
import com.revrobotics.CANSparkMax;
import com.revrobotics.CANSparkLowLevel.MotorType;
import edu.wpi.first.wpilibj.TimedRobot;
import edu.wpi.first.wpilibj.Timer;
import edu.wpi.first.wpilibj.XboxController;
import edu.wpi.first.wpilibj.drive.DifferentialDrive;
```

C++

```
#include <frc/TimedRobot.h>
#include <frc/Timer.h>
#include <frc/XboxController.h>
#include <frc/drive/DifferentialDrive.h>
#include <frc/motorcontrol/PWMSparkMax.h>
#include <rev/CANSparkMax.h>
```

PYTHON

```
import wpilib           # Used to get the joysticks
import wpilib.drive     # Used for the DifferentialDrive class
import rev              # REV library
```

CTRE-Phoenix5

JAVA

```
import edu.wpi.first.wpilibj.TimedRobot;
import edu.wpi.first.wpilibj.Timer;
import edu.wpi.first.wpilibj.XboxController;
import edu.wpi.first.wpilibj.drive.DifferentialDrive;
import com.ctre.phoenix.motorcontrol.can.WPI_TalonSRX;
```

C++

```
#include <frc/TimedRobot.h>
#include <frc/Timer.h>
#include <frc/XboxController.h>
#include <frc/drive/DifferentialDrive.h>
#include <ctre/phoenix/motorcontrol/can/WPI_TalonSRX.h>
```

PYTHON

```
import wpilib           # Used to get the joysticks
import wpilib.drive     # Used for the DifferentialDrive class
import ctre             # CTRE library
```

Our code needs to reference the components of WPILib that are used. In C++ this is accomplished using `#include` statements; in Java and Python it is done with `import` statements. The program references classes for `XBoxController` (for driving), `PWMSparkMax` / `TalonFX` / `CANSparkMax` / `WPI_TalonSRX` (for controlling motors), `TimedRobot` (the base class used for the example), `Timer` (used for autonomous), and `DifferentialDrive` (for connecting the Xbox controller to the motors).

5.2.7 Defining the variables for our sample robot

PWM

Java

```java
public class Robot extends TimedRobot {
  private final PWMSparkMax m_leftDrive = new PWMSparkMax(0);
  private final PWMSparkMax m_rightDrive = new PWMSparkMax(1);
  private final DifferentialDrive m_robotDrive =
      new DifferentialDrive(m_leftDrive::set, m_rightDrive::set);
  private final XboxController m_controller = new XboxController(0);
  private final Timer m_timer = new Timer();
```

C++

```cpp
 private:
  // Robot drive system
  frc::PWMSparkMax m_left{0};
  frc::PWMSparkMax m_right{1};
  frc::DifferentialDrive m_robotDrive{
      [&](double output) { m_left.Set(output); },
      [&](double output) { m_right.Set(output); }};

  frc::XboxController m_controller{0};
  frc::Timer m_timer;
};
```

Python

```python
class MyRobot(wpilib.TimedRobot):
    def robotInit(self):
        """
        This function is called upon program startup and
        should be used for any initialization code.
        """
        self.leftDrive = wpilib.PWMSparkMax(0)
        self.rightDrive = wpilib.PWMSparkMax(1)
        self.robotDrive = wpilib.drive.DifferentialDrive(
            self.leftDrive, self.rightDrive
        )
        self.controller = wpilib.XboxController(0)
        self.timer = wpilib.Timer()

        # We need to invert one side of the drivetrain so that positive voltages
        # result in both sides moving forward. Depending on how your robot's
        # gearbox is constructed, you might have to invert the left side instead.
        self.rightDrive.setInverted(True)
```

CTRE-Phoenix6

Java

```
public class Robot extends TimedRobot {
   private final TalonFX m_leftDrive = new TalonFX(1);
   private final TalonFX m_rightDrive = new TalonFX(2);
   private final DifferentialDrive m_robotDrive =
     new DifferentialDrive(m_leftDrive::set, m_rightDrive::set);
   private final XboxController m_controller = new XboxController(0);
   private final Timer m_timer = new Timer();
```

C++

```
 public:
  Robot() {
```

```
    // We need to invert one side of the drivetrain so that positive voltages
    // result in both sides moving forward. Depending on how your robot's
    // gearbox is constructed, you might have to invert the left side instead.
    m_right.SetInverted(true);
    m_robotDrive.SetExpiration(100_ms);
    m_timer.Start();
  }
```

```
private:
 // Robot drive system
 ctre::phoenix6::hardware::TalonFX m_left{1};
 ctre::phoenix6::hardware::TalonFX m_right{2};
 frc::DifferentialDrive m_robotDrive{
   [&](double output) { m_left.Set(output); },
   [&](double output) { m_right.Set(output); }};
 frc::XboxController m_controller{0};
 frc::Timer m_timer;
```

Python

```
class MyRobot(wpilib.TimedRobot):
  def robotInit(self):
      """
      This function is called upon program startup and
      should be used for any initialization code.
      """
      self.leftDrive = phoenix6.hardware.TalonFX(1)
      self.rightDrive = phoenix6.hardware.TalonFX(2)
      self.robotDrive = wpilib.drive.DifferentialDrive(
          self.leftDrive, self.rightDrive
      )
      self.controller = wpilib.XboxController(0)
      self.timer = wpilib.Timer()
      # We need to invert one side of the drivetrain so that positive voltages
      # result in both sides moving forward. Depending on how your robot's
```

(continues on next page)

(continued from previous page)

```
    # gearbox is constructed, you might have to invert the left side instead.
    self.rightDrive.setInverted(True)
```

REV

Java

```
public class Robot extends TimedRobot {
  private final CANSparkMax m_leftDrive = new CANSparkMax(1, MotorType.kBrushless);
  private final CANSparkMax m_rightDrive = new CANSparkMax(2, MotorType.kBrushless);
  private final DifferentialDrive m_robotDrive =
    new DifferentialDrive(m_leftDrive::set, m_rightDrive::set);
  private final XboxController m_controller = new XboxController(0);
  private final Timer m_timer = new Timer();
```

C++

```
 public:
  Robot() {
```

```
    // We need to invert one side of the drivetrain so that positive voltages
    // result in both sides moving forward. Depending on how your robot's
    // gearbox is constructed, you might have to invert the left side instead.
    m_right.SetInverted(true);
    m_robotDrive.SetExpiration(100_ms);
    m_timer.Start();
  }
```

```
private:
 // Robot drive system
 rev::CANSparkMax m_left{1, rev::CANSparkMax::MotorType::kBrushless};
 rev::CANSparkMax m_right{2, rev::CANSparkMax::MotorType::kBrushless};
 frc::DifferentialDrive m_robotDrive{
   [&](double output) { m_left.Set(output); },
   [&](double output) { m_right.Set(output); }};
 frc::XboxController m_controller{0};
 frc::Timer m_timer;
```

Python

```
class MyRobot(wpilib.TimedRobot):
    def robotInit(self):
        """
        This function is called upon program startup and
        should be used for any initialization code.
        """
        self.leftDrive = rev.CANSparkMax(1, rev.CANSparkMax.MotorType.kBrushless)
        self.rightDrive = rev.CANSparkMax(2, rev.CANSparkMax.MotorType.kBrushless)
        self.robotDrive = wpilib.drive.DifferentialDrive(
```

(continues on next page)

(continued from previous page)

```
            self.leftDrive, self.rightDrive
        )
        self.controller = wpilib.XboxController(0)
        self.timer = wpilib.Timer()

        # We need to invert one side of the drivetrain so that positive voltages
        # result in both sides moving forward. Depending on how your robot's
        # gearbox is constructed, you might have to invert the left side instead.
        self.rightDrive.setInverted(True)
```

CTRE-Phoenix5

Java

```
public class Robot extends TimedRobot {
   private final WPI_TalonSRX m_leftDrive = new WPI_TalonSRX(1);
   private final WPI_TalonSRX m_rightDrive = new WPI_TalonSRX(2);
   private final DifferentialDrive m_robotDrive = new DifferentialDrive(m_
↪leftDrive1::set, m_rightDrive2::set);
   private final XboxController m_controller = new XboxController(0);
   private final Timer m_timer = new Timer();
```

C++

```
 public:
  Robot() {
```

```
    // We need to invert one side of the drivetrain so that positive voltages
    // result in both sides moving forward. Depending on how your robot's
    // gearbox is constructed, you might have to invert the left side instead.
    m_right.SetInverted(true);
    m_robotDrive.SetExpiration(100_ms);
    m_timer.Start();
  }
```

```
private:
 // Robot drive system
 ctre::phoenix::motorcontrol::can::WPI_TalonSRX m_left{1};
 ctre::phoenix::motorcontrol::can::WPI_TalonSRX m_right{2};
 frc::DifferentialDrive m_robotDrive{
   [&](double output) { m_left.Set(output); },
   [&](double output) { m_right.Set(output); }};
 frc::XboxController m_controller{0};
 frc::Timer m_timer;
```

Python

```python
class MyRobot(wpilib.TimedRobot):
    def robotInit(self):
        """
        This function is called upon program startup and
        should be used for any initialization code.
        """
        self.leftDrive = ctre.WPI_TalonFX(1)
        self.rightDrive = ctre.WPI_TalonFX(2)
        self.robotDrive = wpilib.drive.DifferentialDrive(
            self.leftDrive, self.rightDrive
        )
        self.controller = wpilib.XboxController(0)
        self.timer = wpilib.Timer()

        # We need to invert one side of the drivetrain so that positive voltages
        # result in both sides moving forward. Depending on how your robot's
        # gearbox is constructed, you might have to invert the left side instead.
        self.rightDrive.setInverted(True)
```

The sample robot in our examples will have an Xbox Controller on USB port 0 for arcade drive and two motors on PWM ports 0 and 1 (Vendor examples use CAN with IDs 1 and 2). Here we create objects of type `DifferentialDrive` (m_robotDrive), `XboxController` (m_controller) and `Timer` (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.

5.2.8 Robot Initialization

Java

```java
  /** Called once at the beginning of the robot program. */
  public Robot() {
    // We need to invert one side of the drivetrain so that positive voltages
    // result in both sides moving forward. Depending on how your robot's
    // gearbox is constructed, you might have to invert the left side instead.
    m_rightDrive.setInverted(true);
  }
```

C++

```
 public:
  Robot() {
    // We need to invert one side of the drivetrain so that positive voltages
    // result in both sides moving forward. Depending on how your robot's
    // gearbox is constructed, you might have to invert the left side instead.
    m_right.SetInverted(true);
    m_robotDrive.SetExpiration(100_ms);
    m_timer.Start();
  }
```

Python

```
def robotInit(self):
```

The `Robot` constructor for our sample program inverts the right side of the drivetrain. Depending on your drive setup, you might need to invert the left side instead.

5.2.9 Simple Autonomous Example

JAVA

```
  /** This function is run once each time the robot enters autonomous mode. */
  @Override
  public void autonomousInit() {
    m_timer.restart();
  }

  /** This function is called periodically during autonomous. */
  @Override
  public void autonomousPeriodic() {
    // Drive for 2 seconds
    if (m_timer.get() < 2.0) {
      // Drive forwards half speed, make sure to turn input squaring off
      m_robotDrive.arcadeDrive(0.5, 0.0, false);
    } else {
      m_robotDrive.stopMotor(); // stop robot
    }
  }
```

C++

```
  void AutonomousInit() override { m_timer.Restart(); }

  void AutonomousPeriodic() override {
    // Drive for 2 seconds
    if (m_timer.Get() < 2_s) {
      // Drive forwards half speed, make sure to turn input squaring off
      m_robotDrive.ArcadeDrive(0.5, 0.0, false);
```

(continues on next page)

(continued from previous page)

```
    } else {
      // Stop robot
      m_robotDrive.ArcadeDrive(0.0, 0.0, false);
    }
  }
```

PYTHON

```
    def autonomousInit(self):
        """This function is run once each time the robot enters autonomous mode."""
        self.timer.restart()

    def autonomousPeriodic(self):
        """This function is called periodically during autonomous."""

        # Drive for two seconds
        if self.timer.get() < 2.0:
            # Drive forwards half speed, make sure to turn input squaring off
            self.robotDrive.arcadeDrive(0.5, 0, squareInputs=False)
        else:
            self.robotDrive.stopMotor()  # Stop robot
```

The `AutonomousInit` method is run once each time the robot transitions to autonomous from another mode. In this program, we restart the `Timer` 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.

5.2.10 Joystick Control for Teleoperation

JAVA

```
  /** This function is called once each time the robot enters teleoperated mode. */
  @Override
  public void teleopInit() {}

  /** This function is called periodically during teleoperated mode. */
  @Override
  public void teleopPeriodic() {
    m_robotDrive.arcadeDrive(-m_controller.getLeftY(), -m_controller.getRightX());
  }
```

C++

```
  void TeleopInit() override {}

  void TeleopPeriodic() override {
    // Drive with arcade style (use right stick to steer)
    m_robotDrive.ArcadeDrive(-m_controller.GetLeftY(),
                             m_controller.GetRightX());
  }
```

PYTHON

```
    def teleopInit(self):
        """This function is called once each time the robot enters teleoperated mode."
↪""

    def teleopPeriodic(self):
        """This function is called periodically during teleoperated mode."""
        self.robotDrive.arcadeDrive(
            -self.controller.getLeftY(), -self.controller.getRightX()
        )
```

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 left thumbstick of the `XBoxController` to forward/back motion of the drive motors and the X-axis to turning motion.

5.2.11 Test Mode

JAVA

```
  /** This function is called once each time the robot enters test mode. */
  @Override
  public void testInit() {}

  /** This function is called periodically during test mode. */
  @Override
  public void testPeriodic() {}
}
```

C++

```
45  void TestInit() override {}
46
47  void TestPeriodic() override {}
```

PYTHON

```
54      def testInit(self):
55          """This function is called once each time the robot enters test mode."""
56
57      def testPeriodic(self):
58          """This function is called periodically during test mode."""
```

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

5.2.12 Deploying the Project to a Robot

- *Deploy Java/C++ code*
- *Deploy Python code*

5.3 Running your Test Program

5.3.1 Overview

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

C++/Java/Python

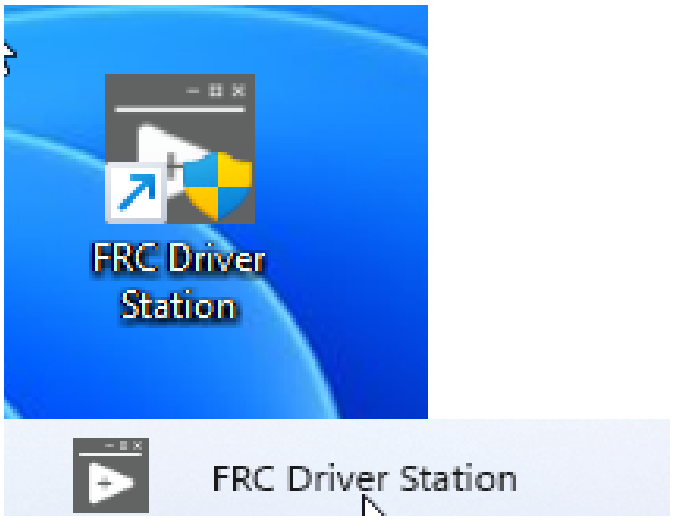
LabVIEW

5.3.2 Tethered Operation

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

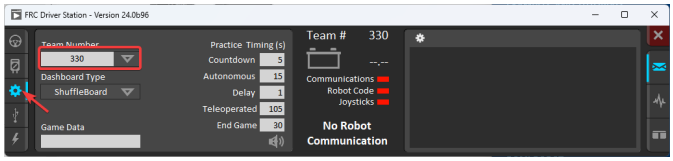
The roboRIO should be powered on and connected to the PC over Ethernet or USB.

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

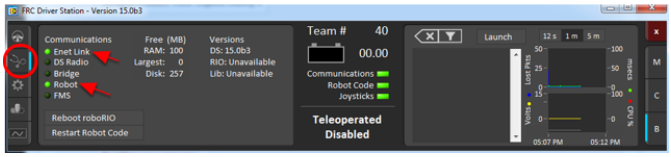


Fig. 1: Tethered

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.

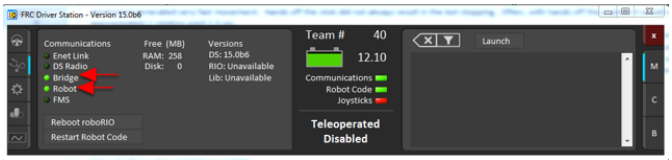
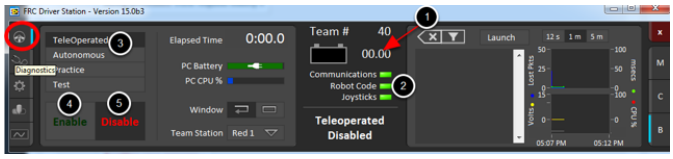


Fig. 2: Wireless

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 test 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 second VH-109 for use as an access point.

After configuring the access point, connect the driver station via Ethernet to the VH-109 AP radio.

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

6 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 wiring instructions/diagrams, please see the *Wiring the FRC Control System* document.

6.1 Overview of Control System

Basic

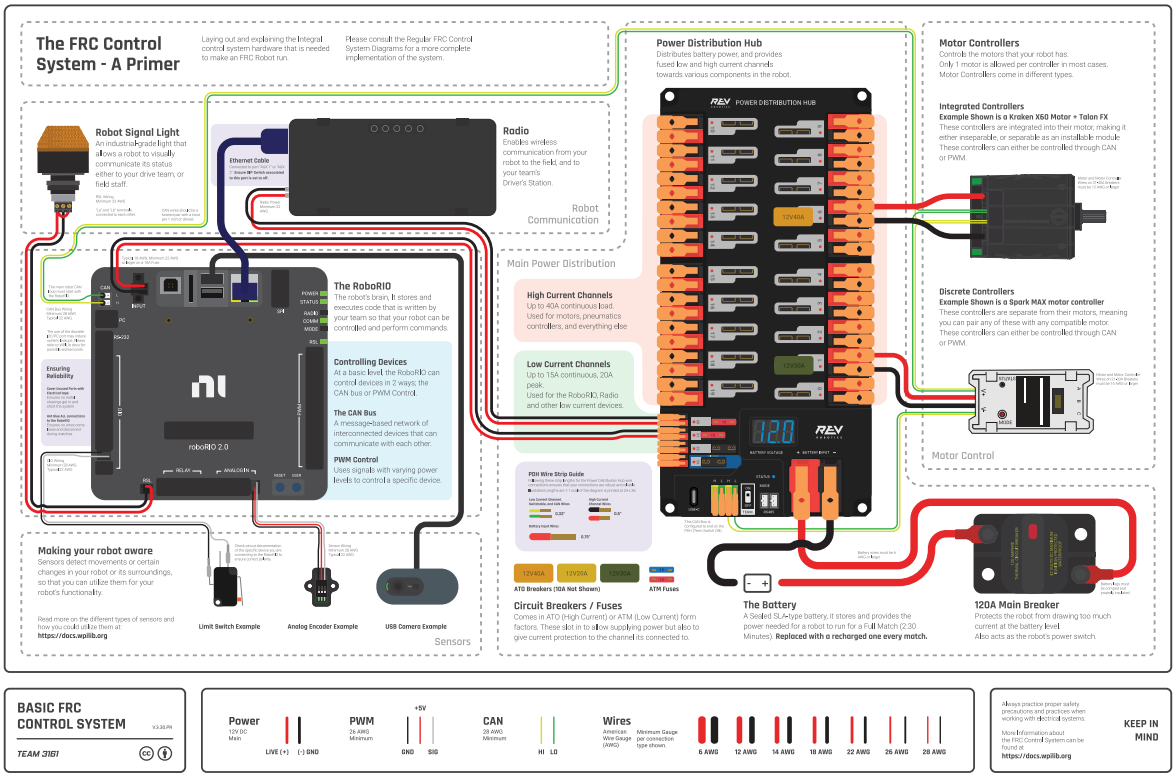
Diagram courtesy of FRC® Team 3161 and Stefen Acepcion.

REV

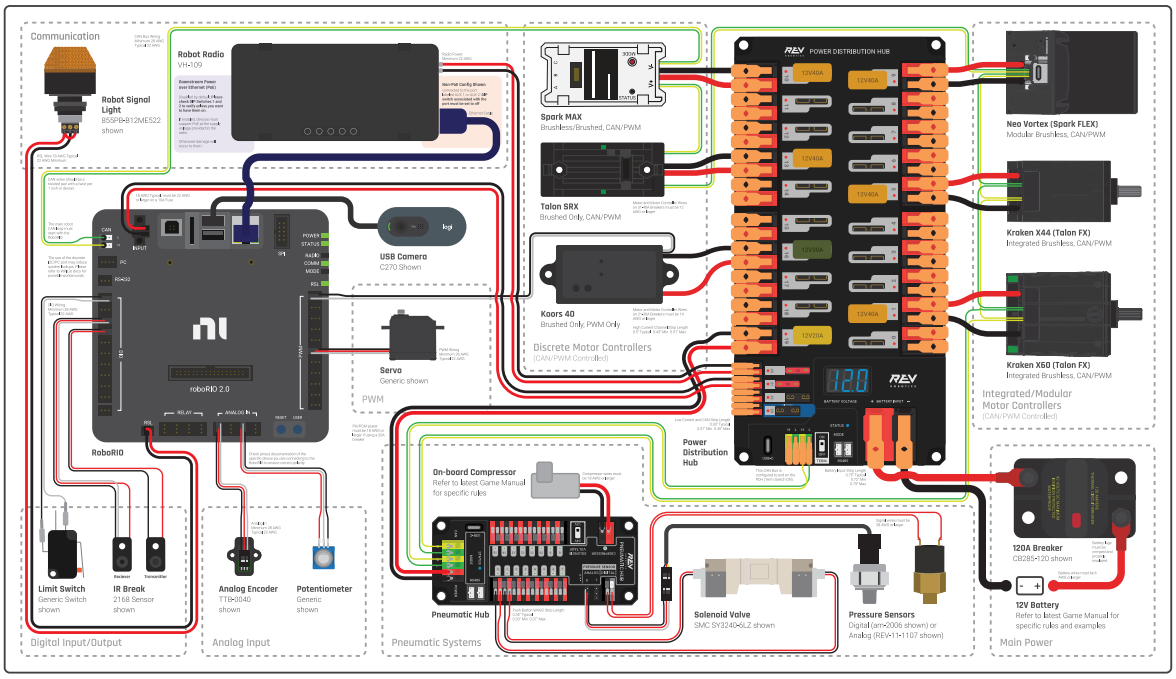
Diagram courtesy of FRC® Team 3161 and Stefen Acepcion.

CTRE

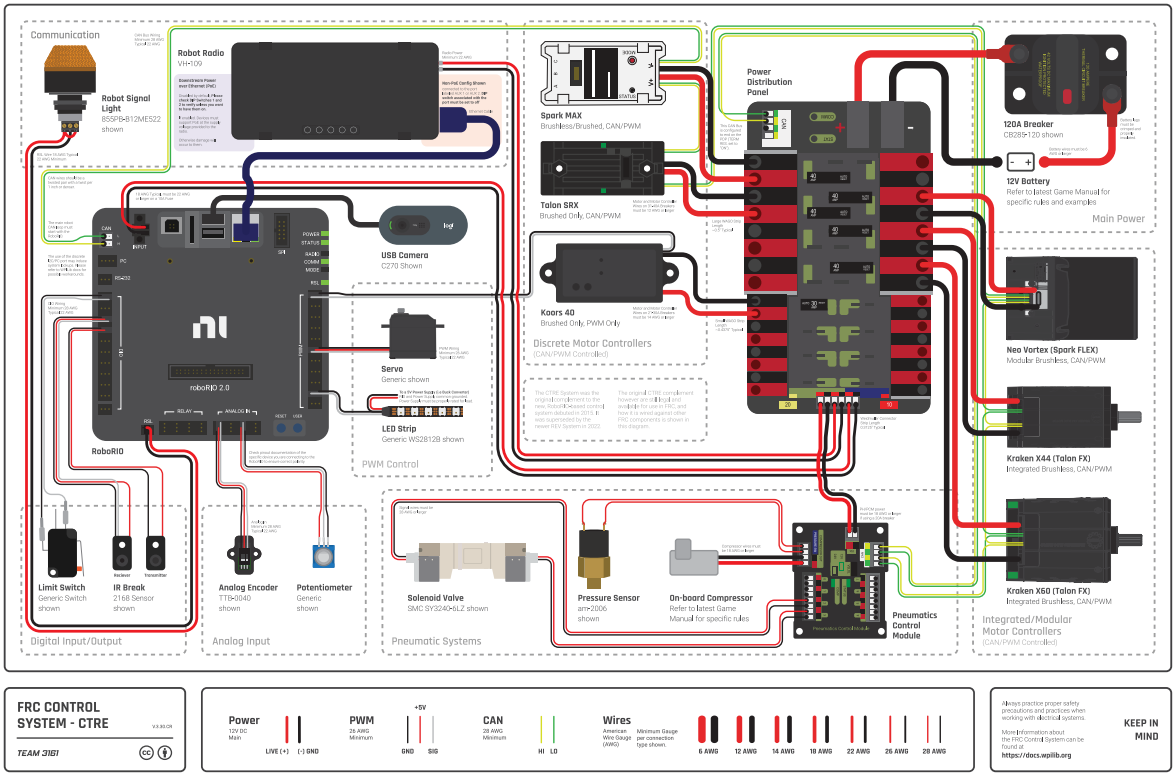
Diagram courtesy of FRC® Team 3161 and Stefen Acepcion.


The FRC Control System - A Primer
Laying out and explaining the Integral control system hardware that is needed to make an FRC Robot run.
Please consult the Regular FRC Control System Diagrams for a more complete implementation of the system.
Robot Signal Light
An industrial-grade light that allows a robot to visually communicate its status either to your drive team, or field staff.
Radio
Enables wireless communication from your robot to the field, and to your team's Driver's Station.
Robot Communication
Power Distribution Hub
Distributes battery power, and provides fused low and high current channels towards various components in the robot.
Motor Controllers
Controls the motors that your robot has. Only 1 motor is allowed per controller in most cases. Motor Controllers come in different types.
Integrated Controllers
Example Shown is a Kraken X60 Motor + Talon FX
These controllers are integrated into their motor, making it either inseparable, or separable as an installable module. These controllers can either be controlled through CAN or PWM.
Discrete Controllers
Example Shown is a Spark MAX motor controller
These controllers are separate from their motors, meaning you can pair any of these with any compatible motor. These controllers can either be controlled through CAN or PWM.
Motor Control
The RoboRIO
The robot's brain. It stores and executes code that is written by your team so that your robot can be controlled and perform commands.
roboRIO 2.0
Main Power Distribution
High Current Channels
Up to 40A continuous load
Used for motors, pneumatics controllers, and everything else
Low Current Channels
Up to 15A continuous, 20A peak.
Used for the RoboRIO, Radio and other low current devices.
Controlling Devices
At a basic level, the RoboRIO can control devices in 2 ways: the CAN bus or PWM Control.
The CAN Bus
A message-based network of interconnected devices that can communicate with each other.
PWM Control
Uses signals with varying power levels to control a specific device.
Ensuring Reliability
Making your robot aware
Sensors detect movements or certain changes in your robot or its surroundings, so that you can utilize them for your robot's functionality.
Read more on the different types of sensors and how you could utilize them at: https://docs.wpilib.org
Limit Switch Example
Analog Encoder Example
USB Camera Example
Sensors
PDH Wire Strip Guide
ATO Breakers (10A Not Shown)
ATM Fuses
Circuit Breakers / Fuses
Comes in ATO (High Current) or ATM (Low Current) form factors. These slot in to allow supplying power but also to give current protection to the channel its connected to.
The Battery
A Sealed SLA-type battery, it stores and provides the power needed for a robot to run for a Full Match (2:30 Minutes). Replaced with a recharged one every match.
120A Main Breaker
Protects the robot from drawing too much current at the battery level. Also acts as the robot's power switch.

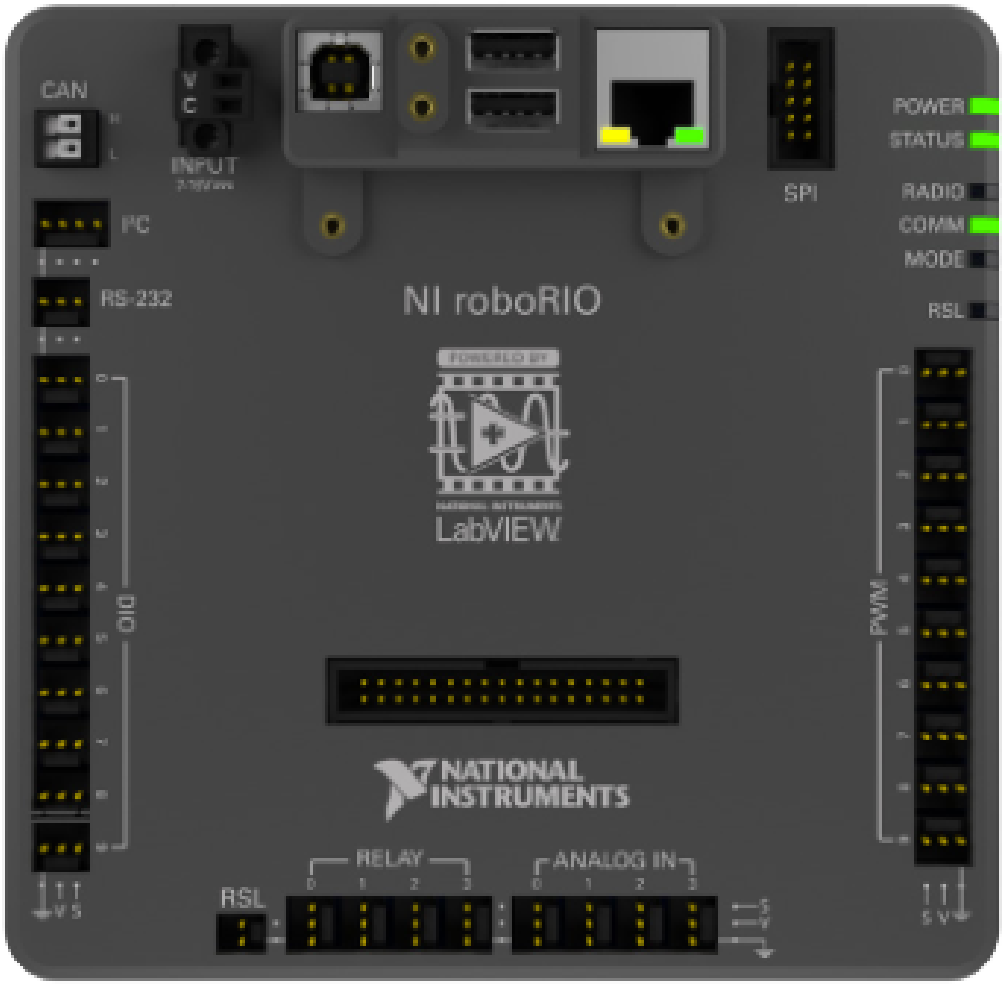
BASIC FRC CONTROL SYSTEM
TEAM 3161
Power
12V DC Main
LIVE (+)
(-) GND
PWM
26 AWG Minimum
+5V
GND
SIG
CAN
28 AWG Minimum
HI
LO
Wires
American Wire Gauge (AWG)
Minimum Gauge per connection type shown.
6 AWG
12 AWG
14 AWG
18 AWG
22 AWG
26 AWG
28 AWG
KEEP IN MIND
More Information about the FRC Control System can be found at https://docs.wpilib.org


Communication
Robot Radio
VH-109
Robot Signal Light
855PB-B12ME522 shown
RoboRIO
roboRIO 2.0
USB Camera
C270 Shown
Servo
Generic shown
PWM
Spark MAX
Brushless/Brushed, CAN/PWM
Talon SRX
Brushed Only, CAN/PWM
Koors 40
Brushed Only, PWM Only
Discrete Motor Controllers
(CAN/PWM Controlled)
Neo Vortex (Spark FLEX)
Modular Brushless, CAN/PWM
Kraken X44 (Talon FX)
Integrated Brushless, CAN/PWM
Kraken X60 (Talon FX)
Integrated Brushless, CAN/PWM
Integrated/Modular Motor Controllers
(CAN/PWM Controlled)
Power Distribution Hub
On-board Compressor
Refer to latest Game Manual for specific rules
Pneumatic Hub
Solenoid Valve
SMC SY3240-5LZ shown
Pressure Sensors
Digital (am-2006 shown) or Analog (REV-11-1107 shown)
Pneumatic Systems
120A Breaker
CB285-120 shown
12V Battery
Refer to latest Game Manual for specific rules and examples
Main Power
Limit Switch
Generic Switch shown
IR Break
2168 Sensor shown
Analog Encoder
TTB-0040 shown
Potentiometer
Generic shown
Digital Input/Output
Analog Input


FRC CONTROL SYSTEM - REV
TEAM 3161
Power
12V DC Main
LIVE (+)
(-) GND
PWM
26 AWG Minimum
+5V
GND
SIG
CAN
28 AWG Minimum
HI
LO
Wires
American Wire Gauge (AWG)
Minimum Gauge per connection type shown.
6 AWG
12 AWG
14 AWG
18 AWG
22 AWG
26 AWG
28 AWG
KEEP IN MIND
More Information about the FRC Control System can be found at https://docs.wpilib.org


Communication
Robot Signal Light
Robot Radio
VH-109
Power Distribution Panel
120A Breaker
12V Battery
Main Power
Spark MAX
Brushless/Brushed, CAN/PWM
Talon SRX
Brushed Only, CAN/PWM
Koors 40
Brushed Only, PWM Only
Discrete Motor Controllers
USB Camera
C270 Shown
roboRIO 2.0
RoboRIO
Servo
Generic shown
LED Strip
Generic WS2812B shown
PWM Control
Neo Vortex (Spark FLEX)
Modular Brushless, CAN/PWM
Kraken X44 (Talon FX)
Integrated Brushless, CAN/PWM
Kraken X60 (Talon FX)
Integrated Brushless, CAN/PWM
Integrated/Modular Motor Controllers
Limit Switch
Generic Switch shown
IR Break
2168 Sensor shown
Digital Input/Output
Analog Encoder
TTB-0040 shown
Potentiometer
Generic shown
Analog Input
Solenoid Valve
SMC SY3240-6LZ shown
Pressure Sensor
am-2006 shown
On-board Compressor
Refer to latest Game Manual for specific rules
Pneumatics Control Module
Pneumatic Systems
FRC CONTROL SYSTEM - CTRE
TEAM 3181
Power
PWM
CAN
Wires
6 AWG
12 AWG
14 AWG
18 AWG
22 AWG
26 AWG
28 AWG
KEEP IN MIND

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 CTRE Power Distribution Panel



The *CTRE 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 CTRE Power Distribution Panel 2.0

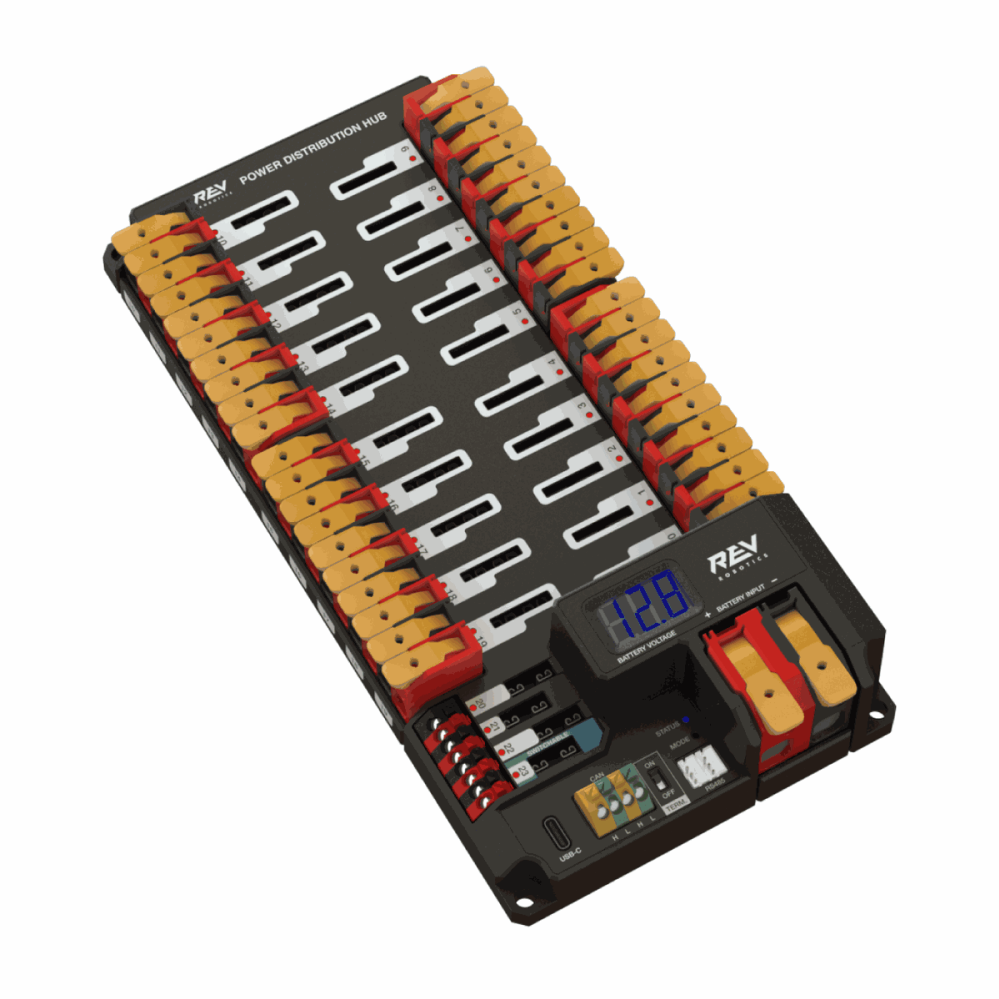


The CTRE Power Distribution Panel 2.0 (PDP 2.0) is designed to distribute power from a 12VDC battery to various robot components. The PDP 2.0 features 24 40A channels and can accoommodate ATO breakers. The PDP 2.0 is designed to maximize efficiency.

Note

The PDP 2.0 does not have buit in current logging and current logs will not be available in the *Driver Station Log Viewer*

6.5 REV Power Distribution Hub



The REV Power Distribution Hub (PDH) is designed to distribute power from a 12VDC battery to various robot components. The PDH features 20 high-current (40A max) channels, 3 low-current (15A max), and 1 switchable low-current channel. The Power Distribution Hub features toolless latching WAGO terminals, an LED voltage display, and the ability to connect over CAN or USB-C to the REV Hardware Client for real-time telemetry.

6.6 AndyMark Power Distribution Board



The AndyMark Power Distribution Board (AMPD) is designed to distribute power from a 12VDC battery to various robot components. The AMPD features 24 fully insulated 40A ports that work with ATO/ATC breakers. With toolless lever connectors and 22.5 degree angled outputs for clean wiring, the AMPD is designed to simplify robot wiring.

Note

The AMPD does not have buit in current logging and current logs will not be available in the *Driver Station Log Viewer*

6.7 CTRE Voltage Regulator Module



The CTRE 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 custom circuits, and IP vision cameras. For more information, see the VRM User Manual.

6.8 Vivid-Hosting VH-109 Radio



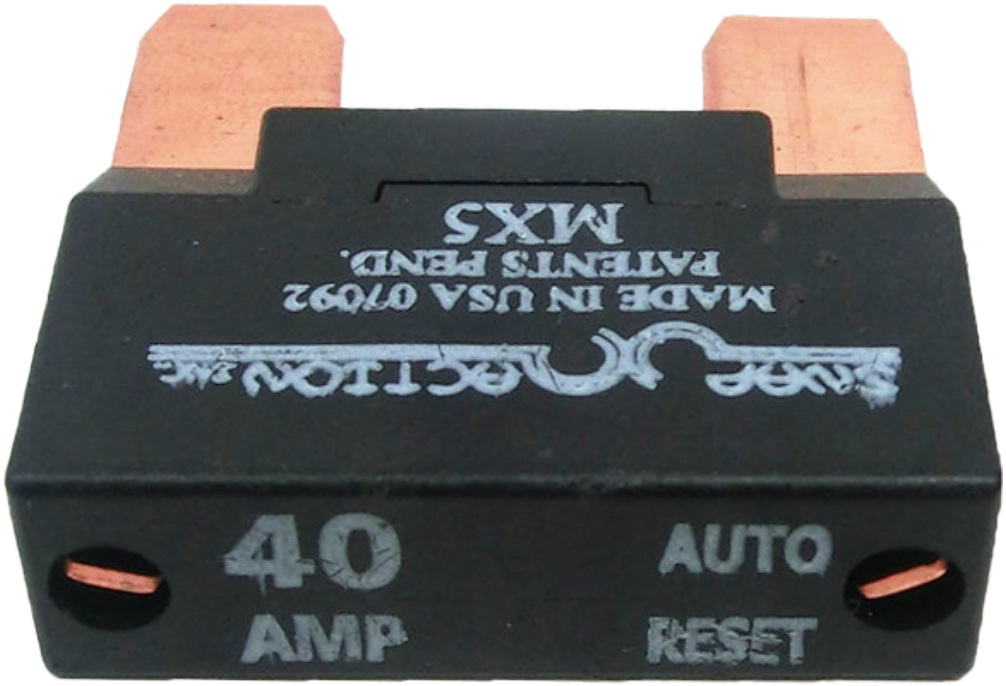
The Vivid-Hosting VH-109 radio is designed specifically for FRC. It uses Wi-Fi 6E to avoid the common congestion problems that plague 2.4 GHz Wi-Fi networks. The ruggedized design allows for unfiltered power input coming directly from a power distribution device. It also has 4 Ethernet ports reducing the need for an additional network switch.

6.9 120A Circuit Breaker



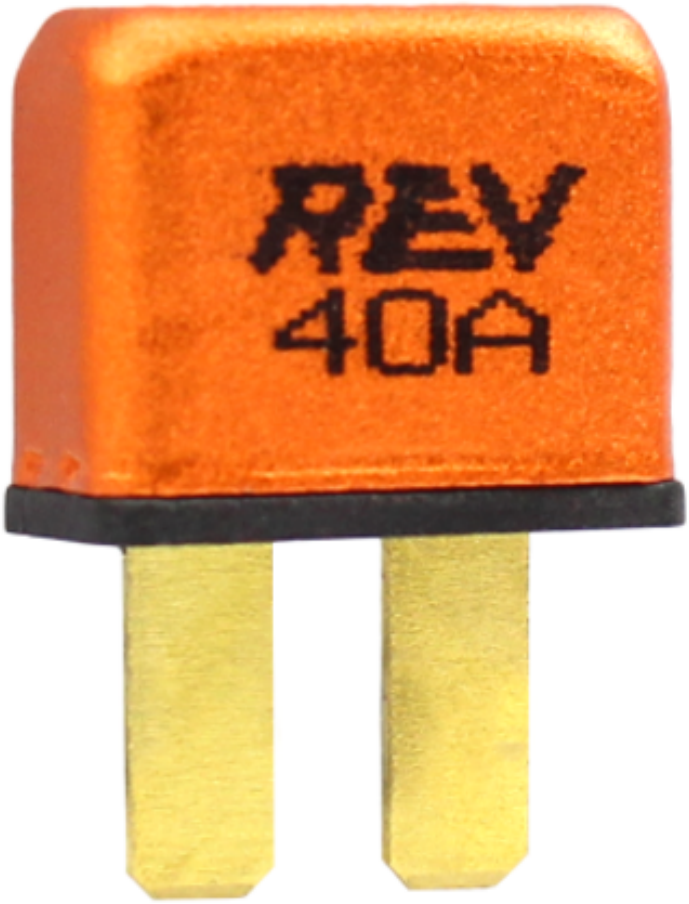
The 120A Main Circuit Breaker serves two roles on the robot: the main robot power switch and a protection device for downstream robot wiring and components. The 120A circuit breaker is wired to the positive terminals of the robot battery and Power Distribution boards. For more information, please see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

6.10 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.

6.11 Rev ATO Circuit Breakers



The Rev ATO Circuit Breakers are used with the Power Distribution Panel, Power Distribution Hub, and Power Distribution Panel 2.0 to limit current to branch circuits. They come in 40A, 30A, 20A, and 10A variants. The ratings on these circuit breakers are for continuous current, temporary peak values can be considerably higher.

6.12 CTRE Power Distribution Panel Breakers



The CTRE Power Distribution Panel Breakers are auto-reset circuit breakers designed for use with CTRE Power Distribution Panels. They come in 40A, 30A, 20A, and 10A variants. These custom ATO-sized breakers automatically reset after tripping, providing consistent performance even after hundreds of trips.

6.13 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.

6.14 Robot Signal Light

Allen-Bradley



Fig. 1: Allen-Bradley 855PB-B12ME522

AndyMark

The Robot Signal Light (RSL) is required to be either Allen-Bradley 855PB-B12ME522 or AndyMark am-3583. It is directly controlled by the roboRIO and will flash when enabled and stay solid while disabled.



Fig. 2: AndyMark am-3583

6.15 CTRE Pneumatics Control Module



The *CTRE 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.

6.16 REV Pneumatic Hub



The REV Pneumatic Hub is a standalone module that is capable of switching both 12V and 24V pneumatic solenoid valves. The Pneumatic Hub features 16 solenoid channels which allow for up to 16 single-acting solenoids, 8 double-acting solenoids, or a combination of the two types. The user selectable output voltage is fully regulated, allowing even 12V solenoids to stay active when the robot battery drops as low as 4.75V.

Digital and analog pressure sensor ports are built into the device, increasing the flexibility and feedback functionality of the pneumatic system. The USB-C connection on the Hub works with the REV Hardware Client, allowing users to test pneumatic systems without a need for an additional robot controller.

6.17 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 placement is estimated for devices where data is unavailable.

> **Note**
>
> 3rd Party CAN control is not supported from WPILib. See this section on *Third-Party CAN Devices* for more information.

6.17.1 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 Documentation.

6.17.2 TalonFX Motor Controller



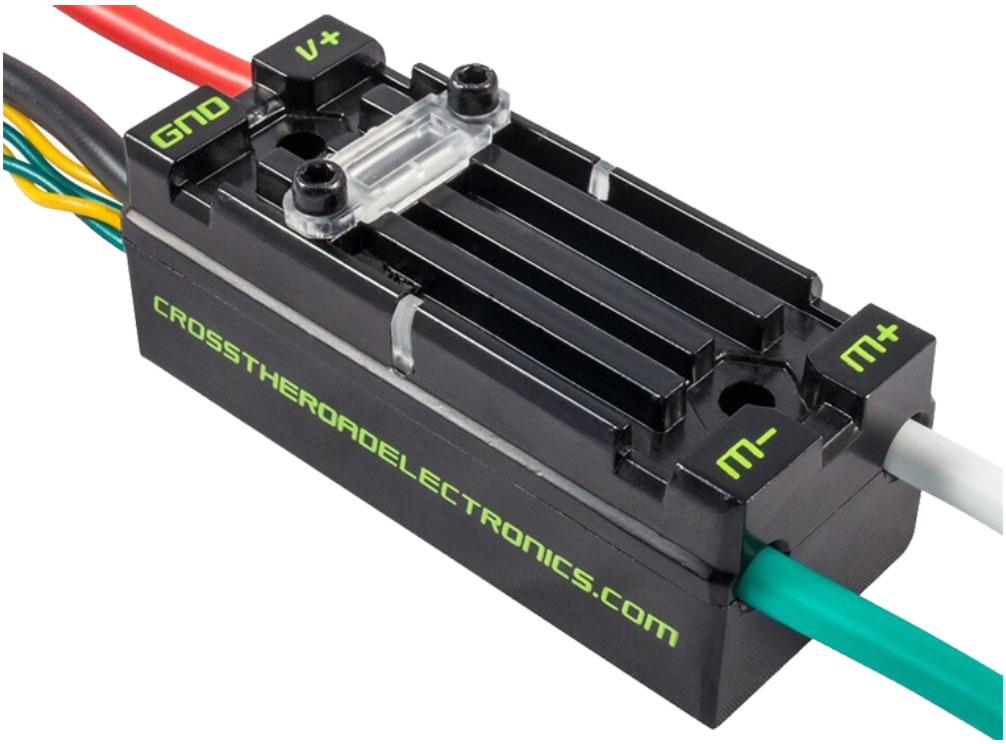
The TalonFX Motor Controller is a brushless motor controller from Cross The Road Electronics which is integrated into the Falcon 500, Kraken X60, and Kraken X44 brushless motors. It features an integrated encoder and all of the smart features of the Talon SRX and more! For more information see the TalonFX Hardware Reference.

6.17.3 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.

6.17.4 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.

6.17.5 SPARK Motor Controller

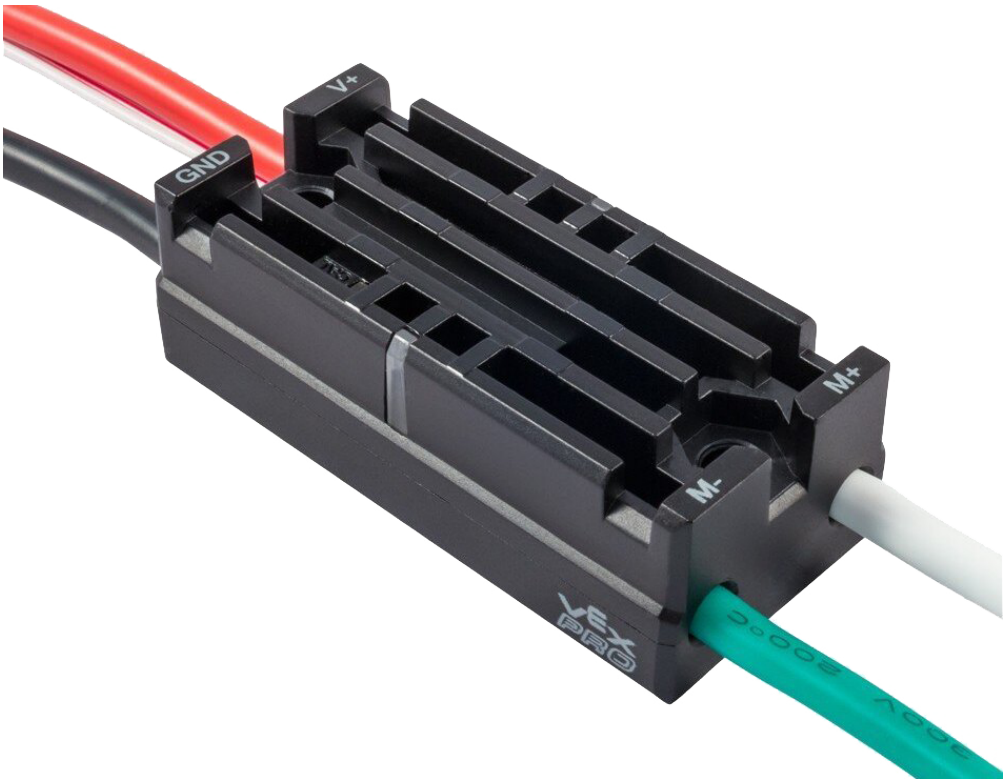


Warning

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.

6.17.6 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.

6.17.7 SPARK Flex Motor Controller



The SPARK Flex Motor Controller is a smart brushed and brushless DC motor controller from Rev Robotics. It is designed for semi-permanent docking to NEO Vortex brushless motors, but can be used with other brushed and brushless motors through the use of an adapter dock. When using CAN bus or USB control, the SPARK Flex uses input from limit switches, encoders, and other sensors, including the integrated encoder of NEO Vortex, Neo 1.0/1.1 and NEO 550 brushless motors, to perform advanced control modes. The SPARK Flex can be controlled over PWM, CAN, or USB (for configuration/testing only). For more information, see the SPARK Flex Documentation.

6.17.8 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.

6.17.9 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.

6.17.10 Thrifty Nova Motor Controller



The Thrifty Nova Motor Controller is a brushless DC motor controller from The Thrifty Bot with CAN and USB connectivity. It can receive sensor inputs from NEO and NEO 550 brush-

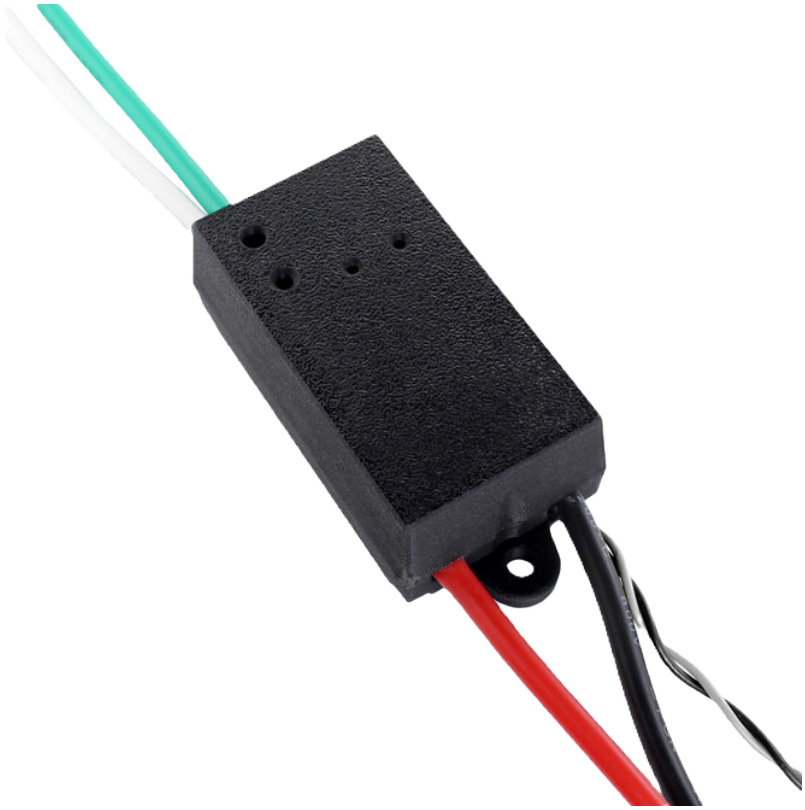
less motors as well as other sensors with the use of expansion boards. For more information, see the Thrifty Nova Documentation.

6.17.11 Talon FXS Motor Controller



The Talon FXS Motor Controller is a smart brushed and brushless DC motor controller from Cross The Road Electronics.

6.17.12 Koors 40 Brushed DC Motor Controller



The Koors 40 Brushed DC Motor Controller is an inexpensive PWM brushed DC motor controller from AndyMark with features such as Ground Loss Protection for PWM, Thermal, and Overcurrent Protection, and Reverse Polarity Protection.

6.18 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.

6.19 Servo Hub



The Servo Hub from Rev Robotics can control up to 6 servos. It can output a maximum of 15A across all channels and. The Servo Hub supports USB, RS485 and CAN control.

6.20 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 webpage.

6.21 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.

6.22 Image Credits

Image of roboRIO courtesy of National Instruments. Images of Talon SRX, Victor SP, Victor SPX, and Spike H-Bridge Relay courtesy of VEX Robotics, Inc. Image of SPARK MAX, SPARK Flex, Servo Hub, ATO Breakers, Power Distribution Hub, and Pneumatic Hub courtesy of REV Robotics. Images of TalonFX, TalonFXS, Power Distribution Panel 2.0, and Power Distribution Panel Breakers courtesy of Cross The Road Electronics, LLC. Image of Thrifty Nova courtesy of The Thrifty Bot. Lifecam, PDP, PCM, SPARK, and VRM photos courtesy of *FIRST*®. Image of the VH-109 radio courtesy of Vivid-Hosting. All other photos courtesy of AndyMark Inc.

7 Software Component Overview

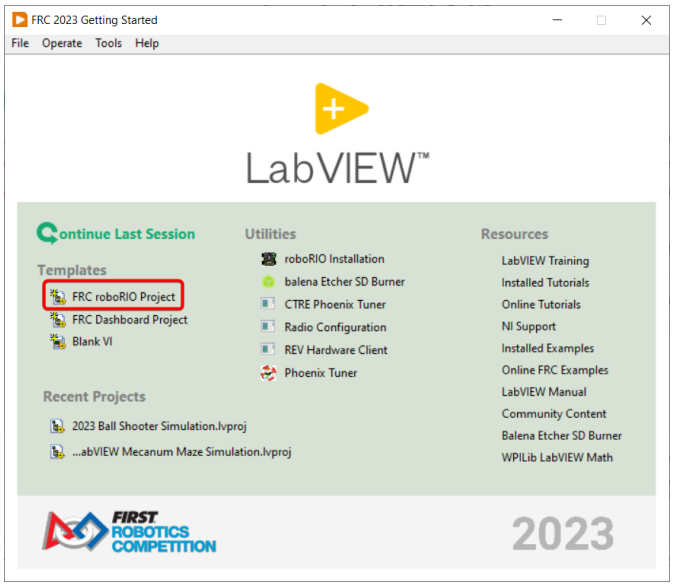
The FRC® software consists of a wide variety of mandatory and optional components. These elements are designed to assist you in the design, development, and debugging of your robot code as well as assist with control robot operation and to provide feedback when troubleshooting. For each software component 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 10 & 11.

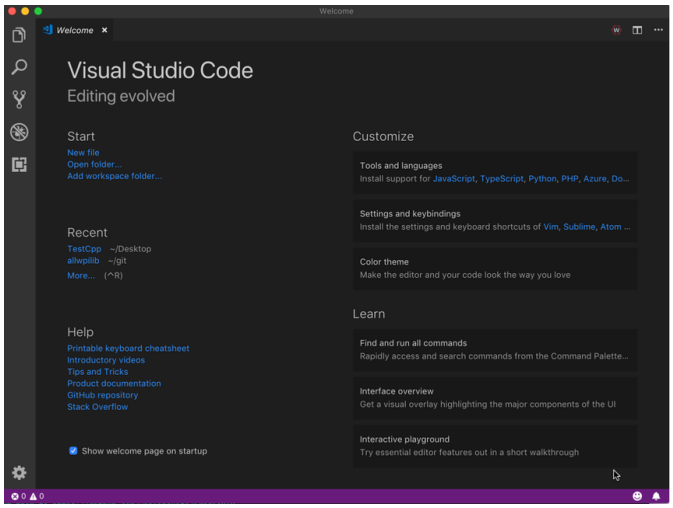
Many of the tools for C++/Java/Python programming are also supported and tested on macOS and Linux. Teams programming in C++/Java/Python should be able to develop using these systems, using a Windows system for the Windows-only operations such as the Driver Station and roboRIO Imaging Tool.

7.2 LabVIEW FRC (Windows Only)



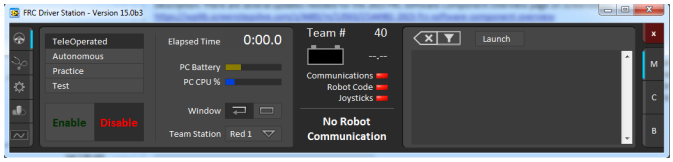
LabVIEW FRC, based on a recent version of LabVIEW Professional, is one of the three officially supported languages for programming an FRC robot. LabVIEW is a graphical, dataflow-driven language. LabVIEW programs consist of a collection of icons, called VIs, wired together with wires which pass data between 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++, Java. A guide to getting started with Java and C++ for FRC, including the installation and configuration of Visual Studio Code can be found *here*.

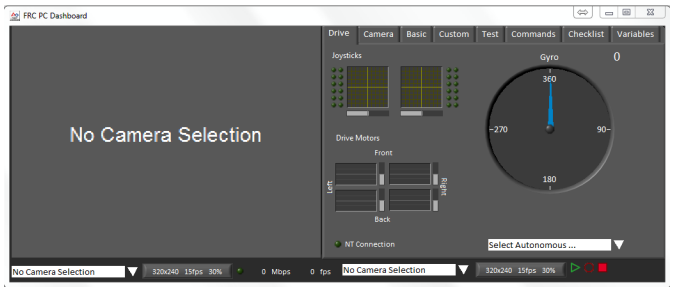
7.4 FRC Driver Station Powered by NI LabVIEW (Windows Only)



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*.

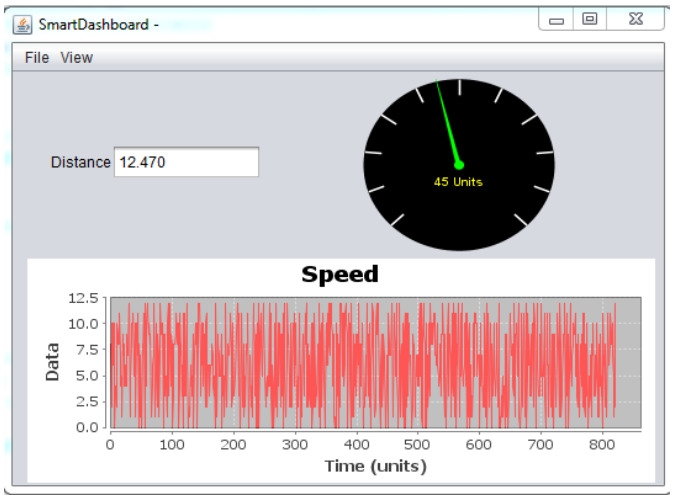
7.5 Dashboard Options

7.5.1 LabVIEW Dashboard (Windows Only)



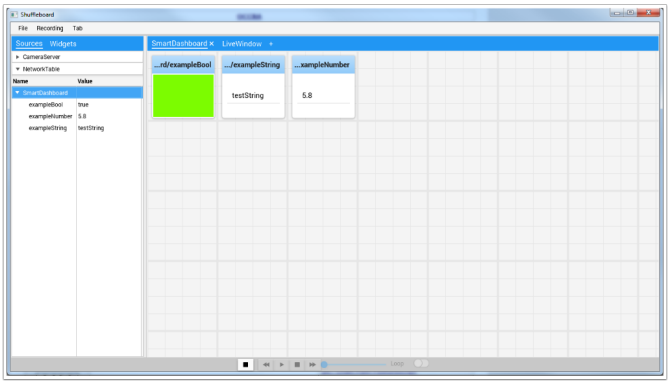
The LabVIEW Dashboard is automatically launched by the FRC Driver Station by default. The purpose of the Dashboard is to provide feedback about the operation of the robot using tabbed display with a variety of built in features. More information about the FRC Default Dashboard software can be found *here*.

7.5.2 SmartDashboard



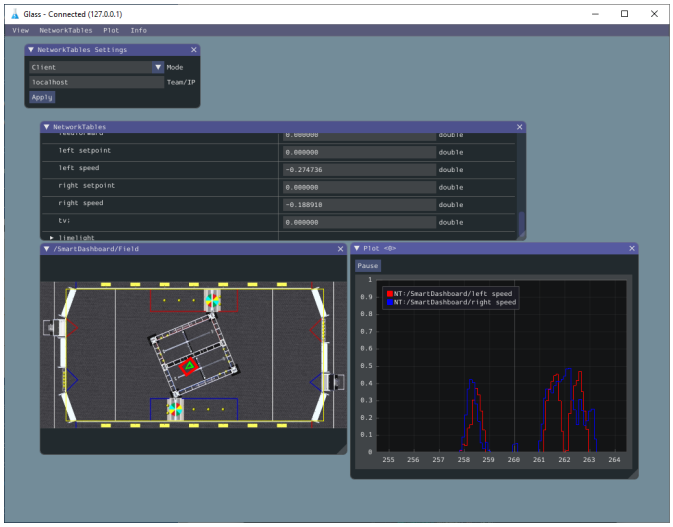
SmartDashboard allows you to view your robot data by automatically creating customizable indicators specifically for each piece of data sent from your robot. Additional documentation on SmartDashboard can be found *here*.

7.5.3 Shuffleboard



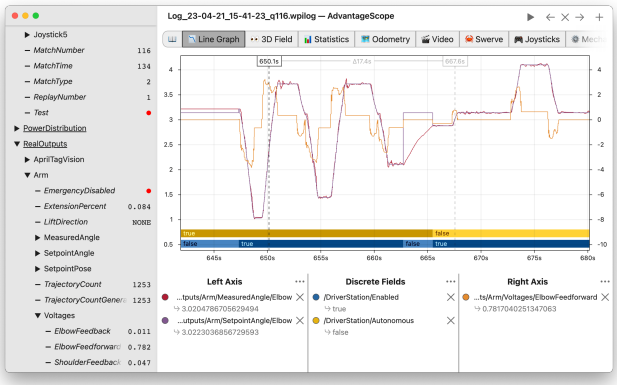
Shuffleboard has the same features as SmartDashboard. It also improves on the setup and visualization of your data with new features and a modern design at the cost of being less resource efficient. Additional documentation on Shuffleboard can be found *here*.

7.5.4 Glass



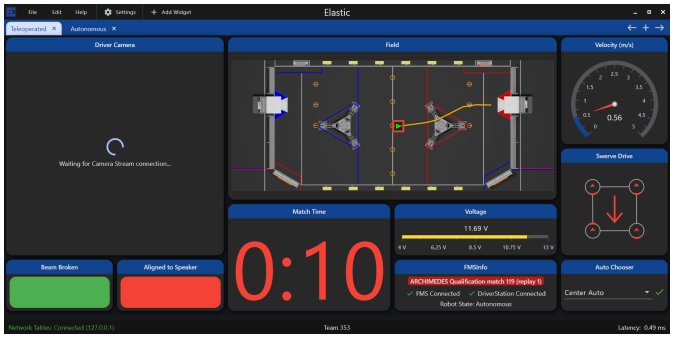
Glass is a Dashboard focused on being a programmer's tool for debugging. The primary advantages are the field view, pose visualization, and advanced signal plotting tools.

7.5.5 AdvantageScope



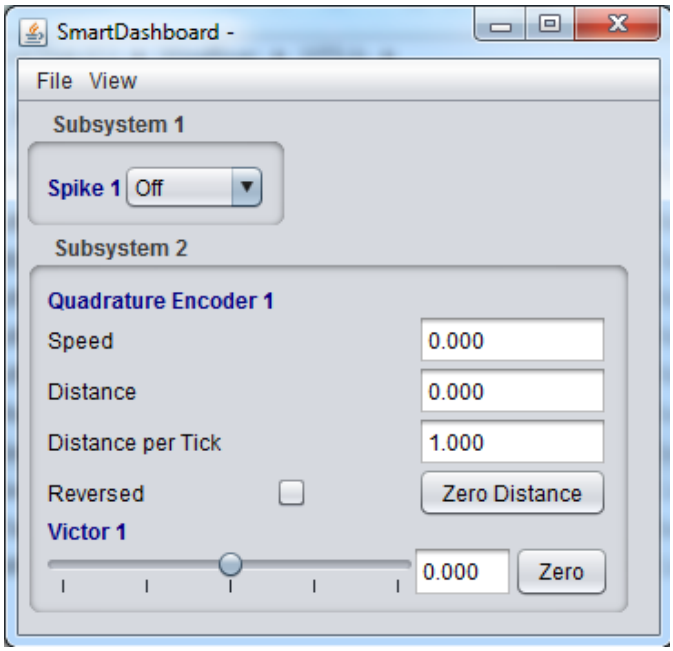
AdvantageScope is a data visualization tool for *NetworkTables*, *WPILib data logs*, and *Driver Station logs*. It is a programmer's tool (rather than a competition dashboard) and can be used to debug real or simulated robot code from a log file or live over the network. For more information see the *AdvantageScope section*.

7.5.6 Elastic



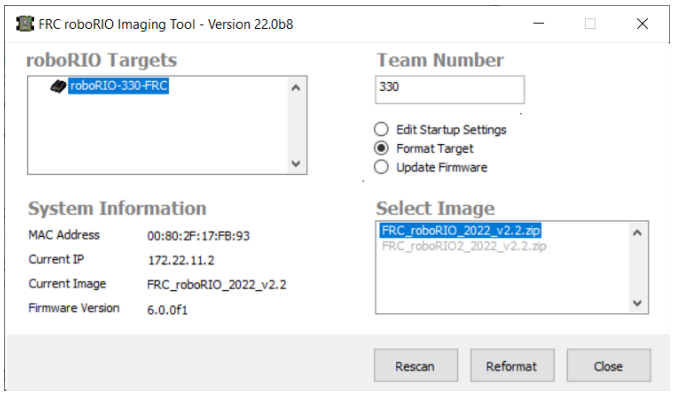
Elastic is a simple and modern dashboard made by Team 353. It is meant to serve as a dashboard for competition but can also be used for testing. It was built with the goal of being a drag & drop dashboard with an easy setup and elegant UI designed for a high pressure competition environment. For more information see the *Elastic section*.

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)



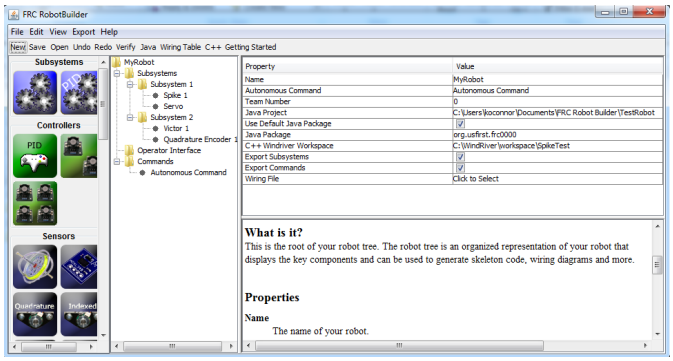
This tool is used to format and setup a roboRIO for use in FRC. Installation instructions can be found *here*. Additional instructions on imaging your roboRIO using this tool can be found *here*.

7.8 FRC Driver Station Log Viewer (Windows Only)



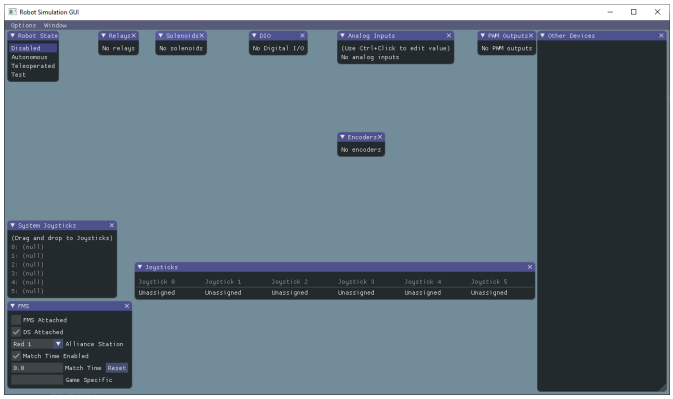
The FRC Driver Station Log Viewer is used to view logs created by the FRC Driver Station. These logs contain a variety of information important for understanding what happened during a practice session or FRC match. More information about the FRC Driver Station Log Viewer and understanding the logs can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

7.9 RobotBuilder



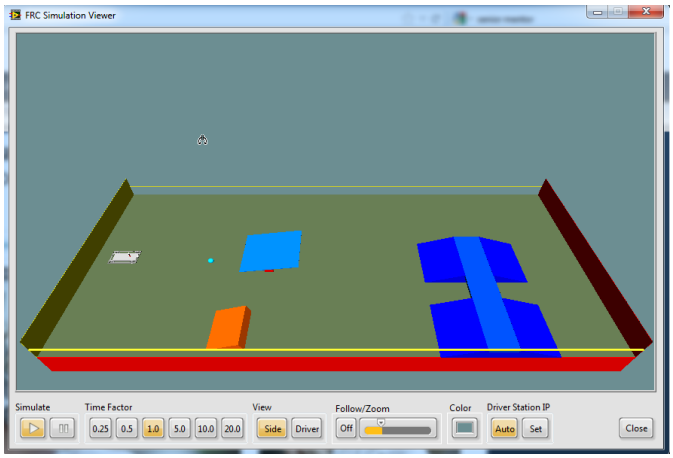
RobotBuilder is a tool designed to aid in setup and structuring of a Command Based robot project for C++ or Java (Python not currently supported). RobotBuilder allows you to enter in the various components of your robot subsystems and operator interface and define what your commands are in a graphical 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*.

7.10 Robot Simulation



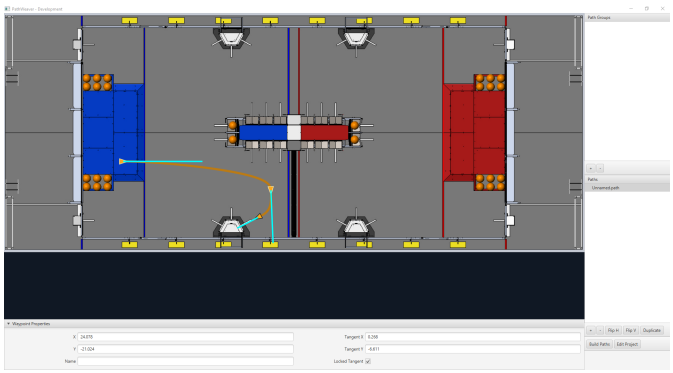
Robot Simulation offers a way for Java, C++, and Python teams to verify their actual robot code is working in a simulated environment. This simulation can be launched directly from VS Code and includes a 2D field that users can visualize their robot's movement on. For more information see the *Robot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

7.11 FRC LabVIEW Robot Simulator (Windows Only)



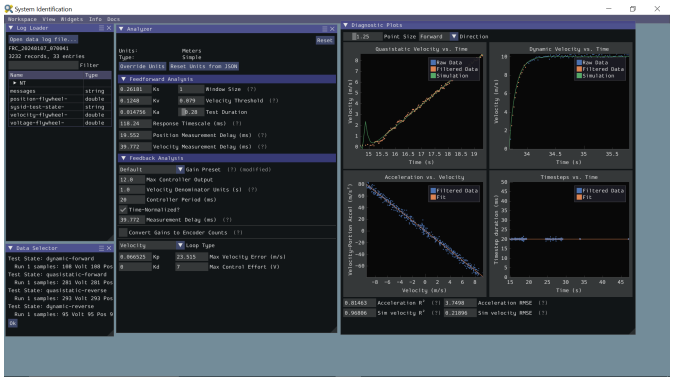
The FRC Robot Simulator is a component of the LabVIEW programming environment that allows you to operate a predefined robot in a simulated environment to test code and/or Driver Station functions. Information on using the FRC Robot Simulator can be found here or by opening the Robot Simulation Readme.html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

7.12 PathWeaver



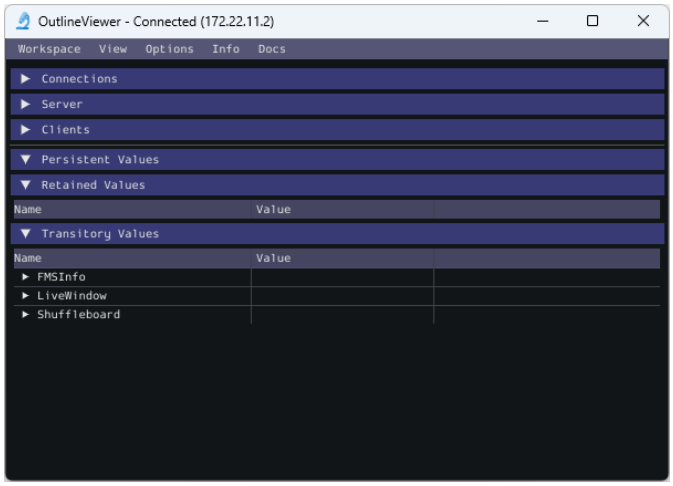
PathWeaver allows teams to quickly generate and configure paths for advanced autonomous routines for differential drivetrains. These paths have smooth curves allowing the team to quickly navigate their robot between points on the field. For more information see the *PathWeaver section*.

7.13 System Identification



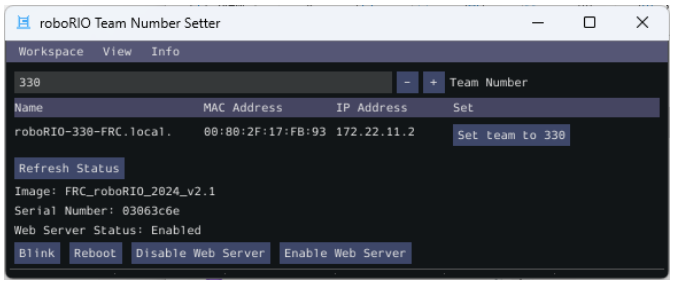
This tool helps teams automatically calculate constants that can be used to describe the physical properties of your robot for use in features like robot simulation, trajectory following, and PID control. For more information see the *System Identification section*.

7.14 OutlineViewer



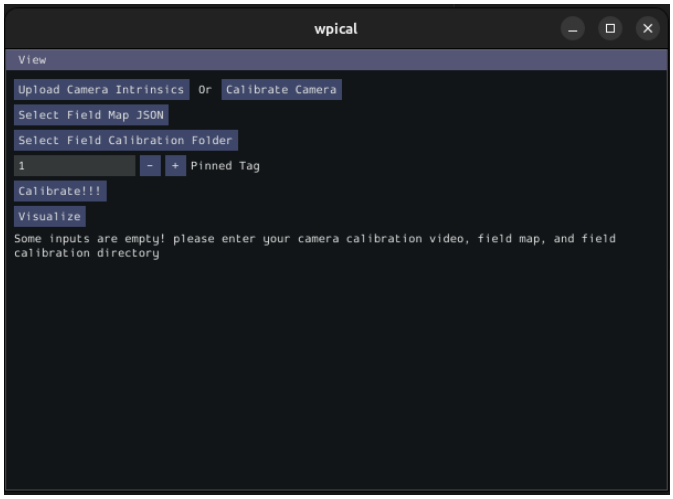
OutlineViewer is a utility used to view, modify, and add to all of the contents of the NetworkTables for debugging purposes. LabVIEW teams can use the Variables tab of the LabVIEW Dashboard to accomplish this functionality. For more information see the *Outline Viewer section*.

7.15 roboRIO Team Number Setter



The roboRIO Team Number Setter is a cross-platform utility that can be used to set the team number on the roboRIO. It is an alternative to the roboRIO imaging tool for setting the team number. For more information see the *roboRIO Team Number Setter section*.

7.16 WPIcal



WPIcal is a cross-platform utility that can be used to calibrate the Apriltags on FRC fields to correct for field setup error. For more information see the *WPIcal section*.

8 What is WPILib?



The WPI Robotics Library (WPILib) is the standard *software library* provided for teams to write code for their FRC® robots. WPILib contains a set of useful classes and subroutines 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 three versions of WPILib, one for each of the three officially-supported text-based languages: WPILibJ for Java, and WPILibC for C++, and RobotPy for Python. A considerable effort is made to maintain feature-parity between these languages - library features are not added unless they can be reasonably supported for both Java and C++ (with the C++ able to be wrapped by pybind for Python), and when possible the class and method names are kept identical or highly-similar. Java, C++, and Python 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 and Python offer lesser performance, but much greater convenience. Python users should take care to test their program to ensure that typos and other issues don't cause robot crashes, as Python is interpreted. New/inexperienced users are encouraged to use Java.

8.2 Source code and documentation

WPILib is an open-source library - the C++ and Java source code is in the allwpilib mono-repo and python source code is in the mostrobotpy mono-repo. The Java and C++ source code can be found in the WPILibJ and WPILibC source directories:

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

- Java source code
- C++ source code
- Python 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 and RobotPy:

- Java documentation
- C++ documentation
- Python documentation

9 2026 Overview

9.1 Known Issues

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

9.1.1 Open Issues

Driver Station Radio Version Error

Issue: The 2026 Driver Station displays "ERROR -44021 Radio Version" when used with VH-109 firmware 2.0.0 or 2.0.1. This is a bug due to the version checking not being updated for the 2026 season and does not affect robot connectivity. Version 2.0.1 is the latest current version and is required by the programming kiosk for use at events.

Unable to deploy on macOS when multiple VS Code versions are installed

Issue: On macOS, having multiple installed copies of Visual Studio Code can cause deploy operations to fail because of macOS local network privacy

Workaround: Open Settings -> Privacy and Security -> Local Network Access and ensure that all VS Code versions are allowed Local Network Access.

If that doesn't work, you can run the following commands in a terminal and then reboot. This allows any program to access the robot's network.

```
sudo defaults write com.apple.network.local-network␣
↪AllowedEthernetLocalNetworkAddresses -array "10.0.0.0/8"
sudo defaults write com.apple.network.local-network AllowedWiFiLocalNetworkAddresses -
↪array "10.0.0.0/8"
```

> **Warning**
>
> This also allows any program to access any network with the address in the range of `10.x.x.x`, which may be undesirable if you connect to other networks in that range beside's

the robot's network. If you only access a robot with a single team number, you can substitute `10.TE.AM.0/24` (*TE.AM IP Notation*). If you only access the robot's network over eithernet or WiFi, you can only run the appropriate command.

Driver Station randomly disabled

Issue: The Driver Station contains tighter safety mechanisms introduced in 2024 to protect against control issues. Some teams have seen this cause the robot to disable.

Workaround: There are multiple potential causes for tripping the safety mechanisms.

Note

The new safety mechanisms will *not* disable the robot when connected to the *FMS*.

The Driver Station software has new tools for control packet delays that could cause this. The control system team requests that teams that experience this issue post screenshots of the *Driver Station Timing window* to https://github.com/wpilibsuite/allwpilib/issues/6174

Some teams have seen this happen only when the robot is operated wirelessly, but not when operated via USB or ethernet tether. Some potential mitigations:

1. Try relocating the robot radio to a better location (high in the robot and away from motors or large amounts of metal). Follow the manufacturer's recommended mounting
2. *Measure your robot's bandwidth* and ensure you have margin under the 7 Mbps bandwidth limit
3. See if the Wi-Fi environment is congested using a tool like WiFi Analyzer.
4. Utilize the recommended *2 radio setup*. 1. Ensure the DS Radio is mounted high, away from interference and humans walking between the DS radio and the robot. 2. Use ethernet to connect the DS computer to the DS Radio
5. Update the Wi-Fi drivers for the DS computer.

Some teams have seen this happen due to software that is running on the driver station (such as Autodesk updater or Discord). Some potential mitigations:

1. Reboot the driver station computer
2. Close software that is running in the background
3. Follow the *Driver Station Best Practices*

While rare, this can be caused by robot code that oversaturates the roboRIO processor or network connection. If all other troubleshooting steps fail, you can try running with one of the WPILib example programs to see if the problem still occurs.

If you identify software that interferes with driver station, please post it to https://github.com/wpilibsuite/allwpilib/issues/6174

Driver Station Reports Less Free RAM then is Available

Issue: The Driver Station diagnostic screen reports free RAM that is misleadingly low. This is due to Linux's use of memory caches. Linux will cache data in memory, but then relinquish when the robot programs requests more memory. The Driver Station only reports memory that isn't used by caches.

Workaround: The true memory available to the robot program is available in the file `/proc/meminfo`. *Use ssh to connect to the robot*, and run `cat /proc/meminfo`.

```
MemTotal:         250152 kB
MemFree:           46484 kB
MemAvailable:     126956 kB
```

The proper value to look is as MemAvailable, rather then MemFree (which is what the driver station is reporting).

Driver Station Reporting No Code

Issue: There is a rare occurrence in the roboRIO 2.0 that causes the roboRIO to not properly start the robot program. This causes the Driver Station to report a successful connection but no code, even though code is deployed on the roboRIO.

Workaround: We are currently investigating the root cause, but FIRST volunteers have been made aware and the recommendation is to reboot the roboRIO when this occurs.

> **Note**
>
> Pressing the physical *User* button on the roboRIO for 5 seconds can also cause the robot code to not start, but a reboot will not start the robot code. If the robot code does not start after rebooting, press the *User* button. Ensure that nothing on the robot is in contact with the *User* button.

Onboard I2C Causing System Lockups

Issue: Use of the onboard I2C port on the roboRIO 1 or 2, in any language, can result in system lockups. The frequency of these lockups appears to be dependent on the specific hardware (i.e. different roboRIOs will behave differently) as well as how the bus is being used.

Workaround: The only surefire mitigation is to use the MXP I2C port or another device to read the I2C data. Accessing the device less frequently and/or using a different roboRIO may significantly reduce the likelihood/frequency of lockups, it will be up to each team to assess their tolerance of the risk of lockup. This lockup can not be definitively identified on the field and a field fault will not be called for a match where this behavior is believed to occur. This lockup is a CPU/kernel hang, the roboRIO will completely stop responding and will not be accessible via the DS, webpage, or SSH. If you can access your roboRIO via any of these methods, you are experiencing a different issue.

Several alternatives exist for accessing the REV color sensor without using the roboRIO I2C port. A similar approach could be used for other I2C sensors.

- Use a Raspberry Pi Pico. Supports up to 2 REV color sensors, sends data to the roboRIO via serial. The Pi Pico is low cost (less than $10) and readily available.

- Use a Raspberry Pi. Supports 1-4 color sensors, sends data to the roboRIO via NetworkTables. Primarily useful for teams already using a Raspberry Pi as a coprocessor.

Updating Properties on roboRIO 2.0 may be slow or hang

Issue: Updating the properties on a roboRIO 2.0 without reformatting using the Imaging Tool (such as setting the team number) may be slow or hang.

Workaround: After a few minutes of the tool waiting the roboRIO should be able to be rebooted and the new properties should be set.

Simulation crashes on Mac after updating WPILib

Issue: On macOS, after updating the project to use a newer version of WPILib, running simulation immediately crashes without the GUI appearing.

Workaround: In VS Code, run WPILib | Run a command in Gradle, `clean`. Alternatively, run `./gradlew clean` in the terminal or delete the build directory.

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:

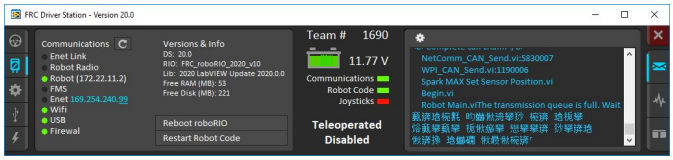
```
Could not apply requested plugin [id: 'edu.wpi.first.GradleRIO', version: '2020.3.2']␣
↪as it does not provide a plugin with id 'edu.wpi.first.GradleRIO'
```

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 then 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 VS Code.
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

Issues with WPILib Dashboards and Simulation on Windows N Editions

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

- Shuffleboard will run, but not load cameras
- Smartdashbard will crash on start-up
- Robot Simulation will crash on start-up

Solution: Install the Media Feature Pack

Python - CameraServer/cscore runs out of memory on roboRIO 1

Issue: When using CameraServer on a roboRIO 1, the image processing program will sometimes exit with a `SIGABRT` or "Error code 6" or a `MemoryError`.

Solution: You may be able to workaround this issue by disabling the NI webserver using the following robotpy-installer command:

```
python -m robotpy installer niweb disable
```

See also

Github issue

9.1.2 Fixed in WPILib 2026.2.1

roboRIO Team Number Setter desktop shortcut broken on Windows

Issue: The *roboRIO Team Number Setter* desktop shortcut is broken in the 2026.1.1 release, but the software is still installed.

Workaround: There are several other ways to start the roboRIO Team Number Setter

1. Use the icon in the Windows start menu
2. Use the Visual Studio Code *Start Tool* feature

9.2 New for 2026

A number of improvements have been made to FRC® Control System software for 2026. This article will describe and provide a brief overview of the new changes and features as well as a more complete changelog for Java/C++ 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.

It's recommended to also review the list of *known issues*.

9.2.1 Importing Projects from Previous Years

Due to internal GradleRIO changes, it is necessary to update projects from previous years. After *Installing WPILib for 2026*, any 2025 projects must be *imported* to be compatible.

9.2.2 Major Changes (Java/C++)

> **Note**
>
> The WPILib team is heavily working on the 2027 Systemcore control system, and thus there are less changes for 2026 then in previous years.

Supported Operating Systems and Architectures:

- Windows 10 & 11, 64 bit only. 32 bit and Arm are not supported
- Ubuntu 22.04 & 24.04, 64 bit. Other Linux distributions with glibc >= 2.34 may work, but are unsupported
- macOS 13.3 or higher, both Intel and Arm.

> **Warning**
>
> The following OSes are no longer supported: macOS 12 or earlier, Ubuntu 18.04 & 20.04, Windows 7, Windows 8.1, and any 32-bit Windows.

Note

Windows 10 support from Microsoft ended in October 2025. We intend to continue supporting Windows 10 through the 2026 season, but may have to drop support in 2027. Teams should start planning their upgrade path to Windows 11.

9.2.3 WPILib

General Library

- Add `Preferences.getNetworkTable()`
- Disambiguate HID Sendable names
- Add X44 and Minion to `DCMotor`
- Add a few method overloads that use units
- Java: Add javac plugin for detecting common error cases at compile time
- 2026.2.1: Add 2026 Apriltag maps and field images

Commands

- Mark CommandPtr class as `[[nodiscard]]`
- Deprecate `Command.schedule()`, use `CommandScheduler.getInstance().schedule(Command...)` instead
- Add `Subsystem.idle()`

NetworkTables

- Check id ranges in control messages

Data Logging

- Add superclass field & method logging to Epilogue
- Use reflection to access non-public superclass fields in Epilogue
- Optimize time and memory usage of epilogue backends
- Support logging of protobuf-serializable types with Epilogue

Hardware interfaces

- Remove Jaguar (and other) motor controllers not legal for 2026

Math

- Upgrade to Eigen 5.0.0
- fix null pointer when calling `TrapezoidProfile.timeLeftUntil`
- Fix Debouncer type-changing behavior
- Fix coordinate frame docs in `HolonomicDriveController`
- Support dynamic matrix sizes in StateSpaceUtil
- Add dynamic size support for numerical jacobian computation
- Add dynamic size support for angle statistics
- Add `nearest()` method to `Pose3d` (mirroring `Pose2d`)
- Add `Translation3d.nearest()`
- Add `copyDirectionPow` to `MathUtil` for joystick input shaping
- Fix `TrapezoidProfile` limiting velocity incorrectly
- Add vector product and squared length operations to Translation2d/3d
- Add structs for TrapezoidProfile.State and ExponentialProfile.State
- Fix units overload resolution
- Add 2D variants of `MathUtil.applyDeadband` and `MathUtil.copySignPow` for circular joystick inputs
- Scale transforms instead of twists in PoseEstimator
- C++: Fix `ElevatorSim::GetCurrentDraw()`
- Correct mil unit
- Add miles per hour conversion methods to `Units.java`
- Fix `ResetTranslation` and `ResetRotation` in `PoseEstimator` and `PoseEstimator3d` causing the robot to teleport
- 2026.2.1: Fix Rotation3d interpolation and document extrinsic vs intrinsic
- 2025.2.1: Add multi tap boolean stream filter and multi tap trigger modifier (double tap detector)

Simulation

Romi/XRP

- XRP: Add support for new Sparkfun RP2350 board

Java units

- Add InchesPerSecondPerSecond unit
- Rename `AngularMomentumUnit.mult` to `per`

CameraServer

- Fix USB video mode handling on macOS
- Add UVC Protocol Support for USB Camera Controls on macOS
- Fix memory leak in usbviewer example
- Resolve macOS camera freeze with specific devices

Util

- Add nested struct schemas before parent schema
- C++: Add reverse iterators to `wpi::circular_buffer` and `wpi::static_circular_buffer`, make other iterators bidirectional

9.2.4 Shuffleboard

Warning

Shuffleboard is deprecated and will be removed for 2027 due to its lack of a maintainer and resource utilization issues. Users can find *additional modern dashboard options here*

- Add deprecated message

9.2.5 SmartDashboard

Important

SmartDashboard is not supported on Apple Silicon (Arm64) Macs.

> **Warning**
>
> SmartDashboard is deprecated and will be removed for 2027 due to its usage of Network Tables v3. Users can find *additional modern dashboard options here*

- Add deprecated messsage

9.2.6 Glass / OutlineViewer / Simulation GUI

- Fix NT int64 value display
- Add GUI context getter hooks

9.2.7 GradleRIO

- Use processstarter for launching tools

9.2.8 WPILib All in One Installer

- Update to VS Code 1.05.1
- VS Code extension updates: cpptools 1.23.2, javaext 1.38
- Ignore settings sync of settings/extensions
- Use processstarter for tools
- Show deprecated message on Windows 10
- Use unique icons for tools
- Fix platform detection for multiple vscode on apple silicon
- 2026.2.1: Catch download failures and show URL
- 2026.2.1: No longer install Python VS Code extensions

9.2.9 Visual Studio Code Extension

- Disable java project manager dependency notification
- Use processstarter for launching tools
- Improve display name for WPILib extension settings
- Defer building code when vendor instructions shown
- Add more context to debugJNI flag in java gradle template
- Add deprecated message to standalone utility
- Fix blank window after installing dependency from local copy
- Import Romi and XRP vendordeps
- Hide the chat sidebar by default

- Hide recommended vscode extensions to install
- Move source backup in jar to backup directory and save more build files
- Check for WPILib project before running most commands
- 2026.2.1: Improve robustness of preferences file detection

9.2.10 RobotBuilder

> **Warning**
>
> RobotBuilder is deprecated and will be removed for 2027 due to its declining usage and burden of updating for the 2027 control system.

- Remove Jaguar (and other) motor controllers not legal for 2026

9.2.11 SysId

- Remove Phoenix5 CANcoder preset
- 2026.2.1: Fix crash on partially empty raw data

9.2.12 PathWeaver

> **Warning**
>
> PathWeaver is deprecated and will be removed for 2027. Users may find *Choreo* or PathPlanner more useful. They both have an intuitive user interface and swerve support.

- No updates other than build updates were made to PathWeaver

9.2.13 AdvantageScope

- Update to 2026 AdvantageScope

9.2.14 Elastic

- Update to 2026.0.0 Elastic. 2026.0.0-beta-1 changelog 2026.0.0 changelog
- 2026.2.1: Upgraded to 2026.1.1 Elastic. 2026.1.1 changelog 2026.1.0 changelog

9.2.15 WPIcal

- No updates other than build updates were made to WPIcal

9.3 Quick Start for Returning Teams

This section serves as a launching point for veteran teams that need to update to the current year's software.

It is advised that **all** teams read through the *changelog* and *known issues* for the season.

1. *Install LabVIEW* (LabVIEW teams only)
2. *Install Game Tools*
3. *Install WPILib* (Java / C++ teams only)
4. *Update third party libraries*
5. Reimage *roboRIO 1* or *roboRIO 2*
6. Update firmware on and *program VH-109 radios*
7. *Import robot project* (Java / C++ teams only)
8. Update software based on the changes described in the *changelog*

9.4 2026 Game Data Details

In the 2026 *FIRST*® Robotics Competition game, the first goal to go inactive is determined by the alliance that scores more Fuel in Auto. The field will transmit the alliance to all 6 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

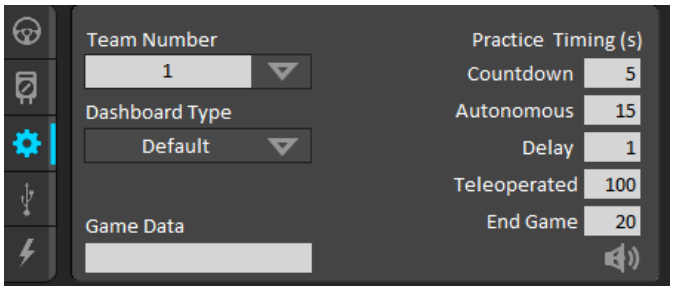
Data is sent to both alliances simultaneously after Fuel scored in Auto is finished being assessed, approximately 3 seconds after the end of Auto. Between the beginning of the match and this point, the Game Data will be an empty string.

Data format

The alliance will be provided as a single character representing the color of the alliance whose goal will go inactive first (i.e. 'R' = red, 'B' = blue). This alliance's goal will be active in Shifts 2 and 4.

9.4.2 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.



Note

For 2026, the match timings have changed and do not match the driver station defaults. To match the 2026 game, change the values to:

- Autonomous: 20 seconds
- Teleoperated: 110 seconds
- End Game: 30 seconds

9.4.3 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.

C++/Java/Python

In C++, Java, and Python 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 selected alliance is sent.

JAVA

```
import edu.wpi.first.wpilibj.DriverStation;
String gameData;
gameData = DriverStation.getGameSpecificMessage();
if(gameData.length() > 0)
{
  switch (gameData.charAt(0))
  {
    case 'B' :
      //Blue case code
      break;
    case 'R' :
      //Red case code
      break;
    default :
      //This is corrupt data
      break;
  }
} else {
  //Code for no data received yet
}
```

C++

```
#include <frc/DriverStation.h>
std::string gameData;
gameData = frc::DriverStation::GetGameSpecificMessage();
if(gameData.length() > 0)
{
  switch (gameData[0])
  {
    case 'B' :
      //Blue case code
      break;
    case 'R' :
```

(continues on next page)

(continued from previous page)

```
      //Red case code
      break;
    default :
      //This is corrupt data
      break;
  }
} else {
  //Code for no data received yet
}
```

PYTHON

```
data = wpilib.DriverStation.getGameSpecificMessage()
if data:
    match data:
        case "B":
            # Blue case code
            ...
        case "R":
            # Red case code
            ...
        case _:
            # This is corrupt data
            ...
else:
    # Code for no data received yet
    ...
```

You can combine the Game Data with the current match time to determine whether your own alliance's hub is currently active. Note however the FMS doesn't send an official match time to robots, only an approximate match time.

For example:

JAVA

```
public boolean isHubActive() {
  Optional<Alliance> alliance = DriverStation.getAlliance();
  // If we have no alliance, we cannot be enabled, therefore no hub.
  if (alliance.isEmpty()) {
    return false;
  }
  // Hub is always enabled in autonomous.
  if (DriverStation.isAutonomousEnabled()) {
    return true;
  }
  // At this point, if we're not teleop enabled, there is no hub.
  if (!DriverStation.isTeleopEnabled()) {
    return false;
  }

  // We're teleop enabled, compute.
  double matchTime = DriverStation.getMatchTime();
```

(continues on next page)

(continued from previous page)

```
  String gameData = DriverStation.getGameSpecificMessage();
  // If we have no game data, we cannot compute, assume hub is active, as its likely␣
↪early in teleop.
  if (gameData.isEmpty()) {
    return true;
  }
  boolean redInactiveFirst = false;
  switch (gameData.charAt(0)) {
    case 'R' -> redInactiveFirst = true;
    case 'B' -> redInactiveFirst = false;
    default -> {
      // If we have invalid game data, assume hub is active.
      return true;
    }
  }

  // Shift was is active for blue if red won auto, or red if blue won auto.
  boolean shift1Active = switch (alliance.get()) {
    case Red -> !redInactiveFirst;
    case Blue -> redInactiveFirst;
  };

  if (matchTime > 130) {
    // Transition shift, hub is active.
    return true;
  } else if (matchTime > 105) {
    // Shift 1
    return shift1Active;
  } else if (matchTime > 80) {
    // Shift 2
    return !shift1Active;
  } else if (matchTime > 55) {
    // Shift 3
    return shift1Active;
  } else if (matchTime > 30) {
    // Shift 4
    return !shift1Active;
  } else {
    // End game, hub always active.
    return true;
  }
}
```

PYTHON

```
from wpilib import DriverStation


def is_hub_active() -> bool:
    alliance = DriverStation.getAlliance()
    # If we have no alliance, we cannot be enabled, therefore no hub.
    if alliance is None:
        return False
```

(continues on next page)

(continued from previous page)

```
    # Hub is always enabled in autonomous.
    if DriverStation.isAutonomousEnabled():
        return True

    # At this point if we're not teleop enabled, there is no hub.
    if not DriverStation.isTeleopEnabled():
        return False

    # We're teleop enabled, compute.
    match_time = DriverStation.getMatchTime()
    game_data = DriverStation.getGameSpecificMessage()

    match game_data:
        case "R":
            red_inactive_first = True
        case "B":
            red_inactive_first = False
        case _:
            # No or invalid game data, assume hub is active.
            return True

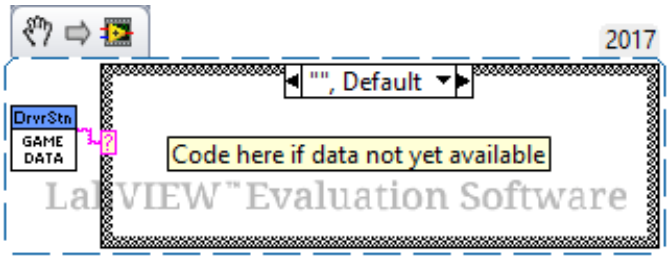
    # Shift 1 is active for blue if red won auto, or red if blue won auto.
    shift1_active = not red_inactive_first if alliance == DriverStation.Alliance.kRed␣
↪else red_inactive_first

    if match_time > 130:
        return True  # Transition shift, hub is active
    elif match_time > 105:
        # Shift 1
        return shift1_active
    elif match_time > 80:
        # Shift 2
        return not shift1_active
    elif match_time > 55:
        # Shift 3
        return shift1_active
    elif match_time > 30:
        # Shift 4
        return not shift1_active
    else:
        return True  # End game, hub always active
```

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 3 possible cases (empty, or either of the 2 letters).

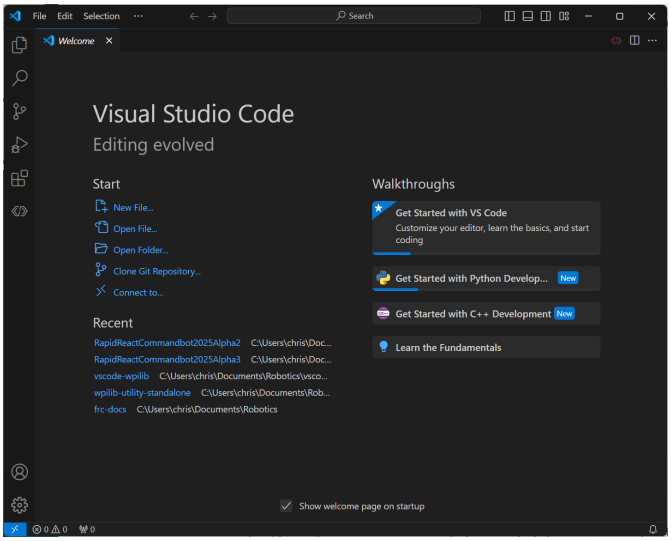

2017
"", Default
DrvrStn
GAME DATA
Code here if data not yet available
LabVIEW™ Evaluation Software

10 VS Code Overview

10.1 Visual Studio Code Basics and the WPILib Extension

Microsoft's Visual Studio Code is the supported IDE for C++ and Java development in FRC. 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

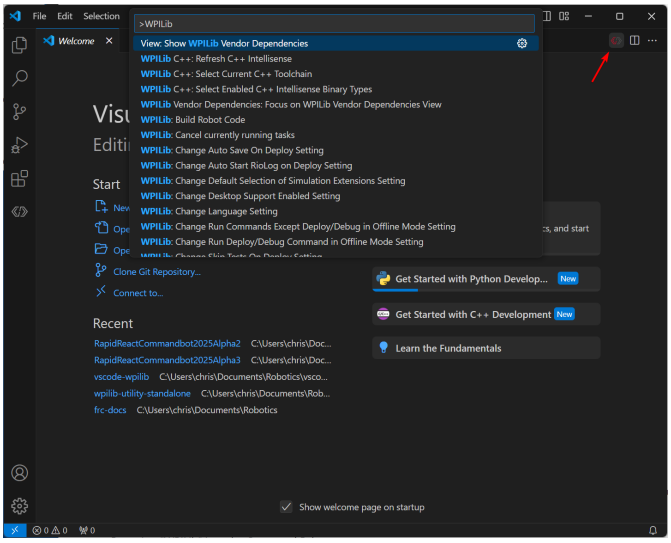
The most important link to take a look at is probably the basic User Interface document. This document describes a lot of the basics of using the UI and provides the majority of the information you should need to get started using Visual Studio Code for FRC.

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 then 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: Change Use WinDbg Preview (From Store) as Windows Debugger Setting** - Change whether to use the VS Code debugger or WinDbg Preview (from Windows Store).
- **WPILib: Check for WPILib Updates** - Check for an update to the WPILib GradleRIO version for the project. This does not update the Visual Studio Code extension, tools, or offline dependencies. Users are strongly recommended to use the *offline wpilib installer*
- **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: Hardware Sim Robot Code** - This builds the current robot code project on your PC and starts it running in simulation using hardware attached to the comupter rather then pure software simulation. Requires vendor support.
- **WPILib: Import a WPILib 2020-202r Gradle Project** - Open a wizard to help you create a new project from a existing VS Code Gradle project from 2020-2022. Further documentation is at *importing gradle 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: run Gradle Clean** - Run Gradle Clean to delete build artifacts
- **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** - 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 various choices available for the base class:

TimedRobot

Documentation: Java - C++

Source: Java - C++

The `TimedRobot` class is the base class recommended for most users. It provides control of the robot program through a collection of `init()`, `periodic()`, and `exit()` methods, which are called by WPILib during specific robot states (e.g. autonomous or teleoperated). During these calls, your code typically polls each input device and acts according to the data it receives. For instance, you would typically determine the position of the joystick and state of the joystick buttons on each call and act accordingly. 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 *TimedRobot*. The example shown below is of *TimedRobot Skeleton*.

JAVA

```java
import edu.wpi.first.wpilibj.TimedRobot;

/**
 * The methods in this class are called automatically corresponding to each mode, as described in
 * the TimedRobot documentation. If you change the name of this class or the package after creating
 * this project, you must also update the Main.java file in the project.
 */
public class Robot extends TimedRobot {
  /**
   * This function is run when the robot is first started up and should be used for any
   * initialization code.
   */
  public Robot() {}

  @Override
  public void robotPeriodic() {}

  @Override
  public void autonomousInit() {}

  @Override
  public void autonomousPeriodic() {}

  @Override
  public void teleopInit() {}

  @Override
  public void teleopPeriodic() {}

  @Override
  public void disabledInit() {}

  @Override
  public void disabledPeriodic() {}

  @Override
  public void testInit() {}

  @Override
  public void testPeriodic() {}

  @Override
  public void simulationInit() {}

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

C++

```
#include "Robot.h"

Robot::Robot() {}
void Robot::RobotPeriodic() {}

void Robot::AutonomousInit() {}
void Robot::AutonomousPeriodic() {}

void Robot::TeleopInit() {}
void Robot::TeleopPeriodic() {}

void Robot::DisabledInit() {}
void Robot::DisabledPeriodic() {}

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

void Robot::SimulationInit() {}
void Robot::SimulationPeriodic() {}

#ifndef RUNNING_FRC_TESTS
int main() {
  return frc::StartRobot<Robot>();
}
#endif
```

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

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

C++

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

Timeslice Robot

Documentation: Java - C++

Source: Java - C++

The `TimesliceRobot` class extends the `TimedRobot` framework to provide more control over the scheduling of periodic functions. It allows users to allocate specific time slices to different robot operations, running them sequentially within a defined period (typically shorter than TimedRobot's default 20ms). This class is recommended for users who need more precise timing control and consistent starting times for their robot's periodic functions.

`TimesliceRobot` provides the same `init()`, `periodic()`, and `exit()` methods as `TimedRobot`, but adds the ability to schedule additional periodic functions with custom allocations. This allows for more efficient use of processing time and can lead to improved performance which can make odometry and estimators more accurate and controller outputs change more consistently.

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 showcases 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

The `Command Robot` framework adds to the basic functionality of a `Timed Robot` by automatically polling inputs and converting the raw input data into events. These events are tied to user code, which is executed when the event is triggered. For instance, when a button is pressed, code tied to the pressing of that button is automatically called and it is not necessary to poll or keep track of the state of that button directly. The `Command Robot` framework makes it easier to write compact easy-to-read code with complex behavior, but requires an additional up-front time investment from a programmer in order to understand how the Command Robot framework works.

Teams using `Command Robot` should see the *Command-Based Programming Tutorial*.

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.

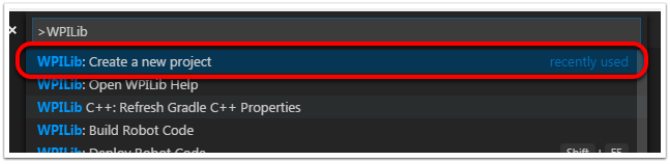
Not Using a Base Class

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 support (see *Introduction to Robot Simulation*).
   - **Java**: This option has no effect and can be left checked or unchecked
   - **C++**: Checking this option enables desktop compilation, which is required for simulation and unit tests
   - **Romi/XRP**: This option has no effect for Romi and XRP templates and can be left checked or unchecked

> **Note**
>
> While WPILib fully supports desktop builds, some third-party vendor libraries may not. If a library doesn't support desktop compilation, your C++ code may not compile

or may crash when running simulation.

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.

Warning

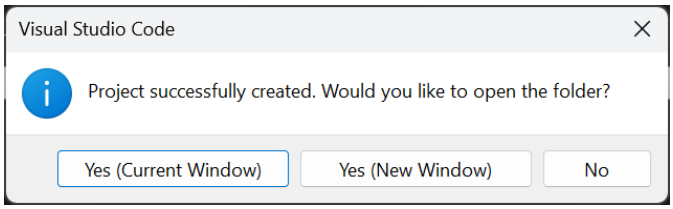
Creating projects on OneDrive is not supported as OneDrive’s caching interferes with the build system. Some Windows installations put the Documents and Desktop folders on OneDrive by default.

An example after all options are selected is shown below.

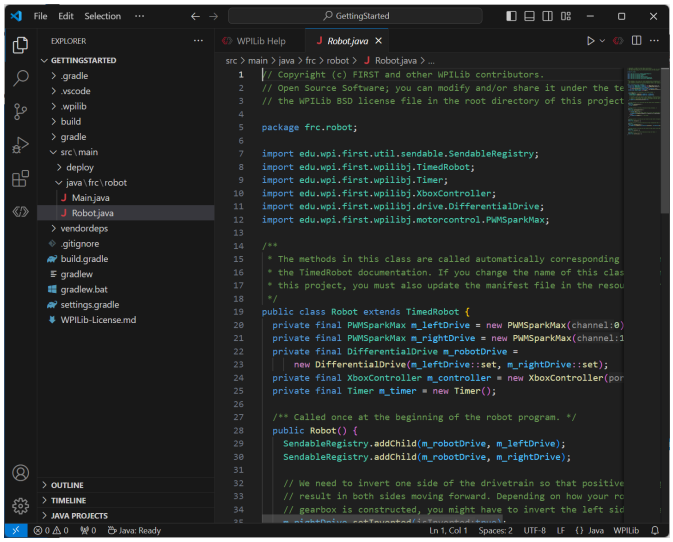


10.3.3 Opening The New Project

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.

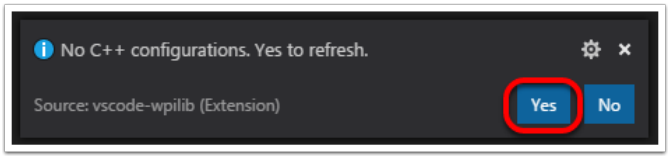


Once opened we will see the project hierarchy on the left. Double clicking on the file will open that file in the editor.



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 using non-*PWM* motor controllers or advanced sensors will most likely need to install external vendor dependencies.

10.4.1 What Are Vendor Dependencies?

A vendor dependency is a way for vendors to add their *software library* to robot projects. This library can interface with motor controllers and other devices. This way, teams can interact with their devices via CAN and have access to more complex and in-depth features than traditional PWM control.

10.4.2 Managing Vendor Dependencies

Vendor dependencies are installed on a per-project basis (so each robot project can have its own set of vendor dependencies). Vendor dependencies can be installed "online" or "offline". The "online" functionality is done by downloading the dependencies over the internet, while offline is typically provided by a vendor-specific installer.

Warning

If installing a vendor dependency via the "online" mode, make sure to reconnect the computer to the internet and rebuild about every 30 days otherwise the cache will clear, completely deleting the downloaded library install.

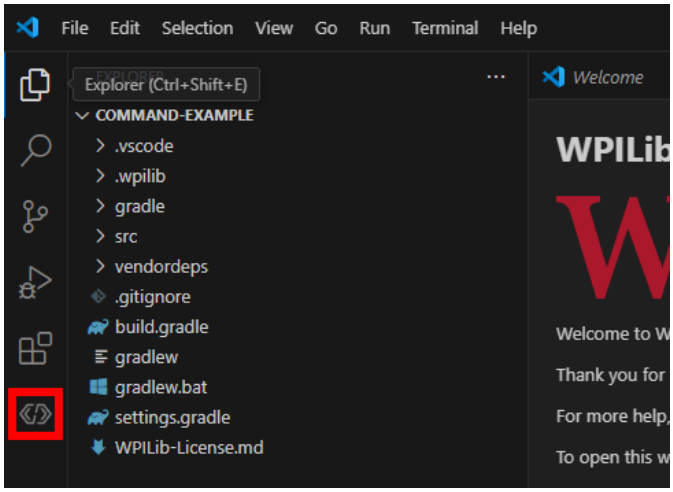
Note

Vendors recommend using their offline installers when available, because the offline installer is typically bundled with additional programs that are extremely useful when working with their devices.

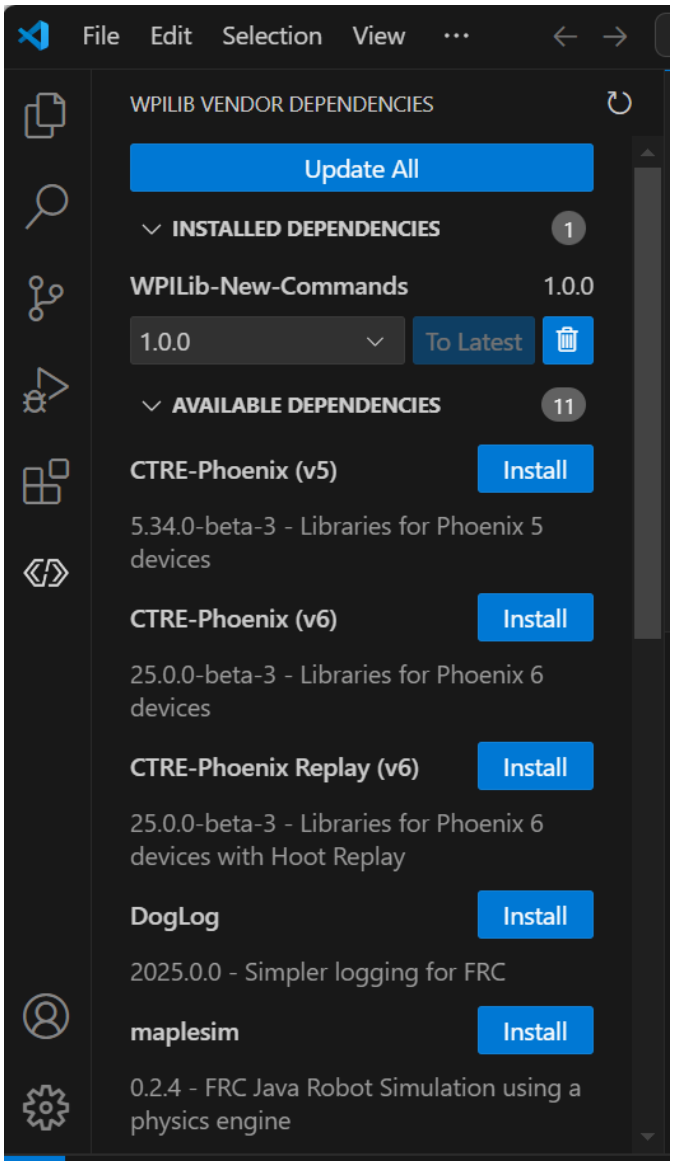
Installing Libraries

Java/C++

VS Code



All vendordep operations can be controlled by the Dependency Manager. Click the WPILib logo in the activity bar as shown above to access the interface.



Select the desired libraries to add to the project by clicking the *Install* button next to each. The JSON file will be copied to the `vendordeps` folder in the project, adding the library as a dependency to the project.

When an update is available for an installed vendordep you will see the *To Latest* button become available. To update you can either press that or the *Update All* to move all vendordeps to the latest version.

The button with the trash icon will uninstall the vendordep. The dropdown shows what version is currently installed but you can change that to a different version to *update* or *downgrade* to the specified version.

Note

The Dependency Manager will automatically build your program when it loses focus. This allows you to use the changed dependencies.

Python

All RobotPy project dependencies are specified in `pyproject.toml`. You can specify optional WPILib components by adding the component name to `components`.

See also

pyproject.toml usage

Optional WPILib components:

	components	requires
Apriltag	apriltag	robotpy-apriltag
Commands	commands2	robotpy-commands-v2
cscore	cscore	robotpy-cscore
Romi	romi	robotpy-romi
XRP	xrp	robotpy-xrp

To add vendor libraries to your project, you must add the PyPI package name to `requires`.

Optional vendor-specific components (not all are available at the beginning of the season):

	Origin	requires
ChoreoLib	Vendor	sleipnirgroup-choreolib
CTRE Phoenix 6	Vendor	phoenix6
CTRE Phoenix 5	Community	robotpy-ctre
PathPlannerLib	Vendor	robotpy-pathplannerlib
PhotonVision	Vendor	photonlibpy
Playing With Fusion	Community	robotpy-playingwithfusion
REVLib	Community	robotpy-rev
Studica	Community	robotpy-navx
URCL	Community	robotpy-urcl

When using `requires`, you should specify a version by appending `==<version>` to the package name, e.g. `robotpy-commands-v2==2024.0.0`. If you do not specify a version, the latest version will be installed.

To check what version of packages are currently installed, run the command `pip list`.

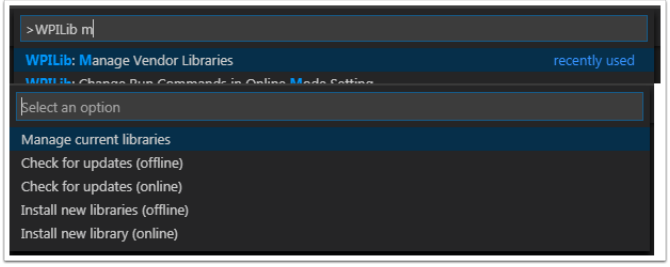
Note that pinning versions may cause issues with incompatibilities between different components. Unpinning all versions, installing, then re-pinning to the latest set is a good practice when updating.

Pinning versions is a good practice to do after most robot code is written and validated, before and during a competition. However, outside of this window, remaining on the latest version of the components is recommended, as it will ensure you have the latest bug fixes and features.

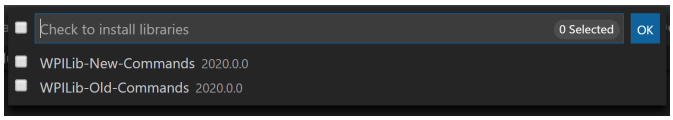
See https://pip.pypa.io/en/stable/topics/repeatable-installs/ for more information on how to specify versions.

Java/C++ (Legacy)

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.

In order to install a vendor library in online mode, 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* and select it in the menu, and then click on *Install new libraries (online)* instead and copy + paste the vendor JSON URL.

Checking for Updates (Offline)

Since dependencies are version managed on a per-project basis, even when installed offline, 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.

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 command-based framework.

How Does It Work?

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 is often done by an offline installer, but may need to be done manually if a `.zip` of the `.json` files is provided. 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 connected to the internet for an initial build to fetch the library from a remote Maven location.

This JSON file allows specification of complex libraries with multiple components (Java, C++, JNI, etc.) and also helps handle some complexities related to simulation.

LabVIEW

For LabVIEW teams, there might be 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 2023\vi.lib\Rock Robotics\WPI\Third Party`

In order to install third party libraries for LabVIEW, download the VIs from the vendor (typically via some sort of installer). Then drag and drop the third party VIs into the respective folder mentioned above just like any other VI.

Python

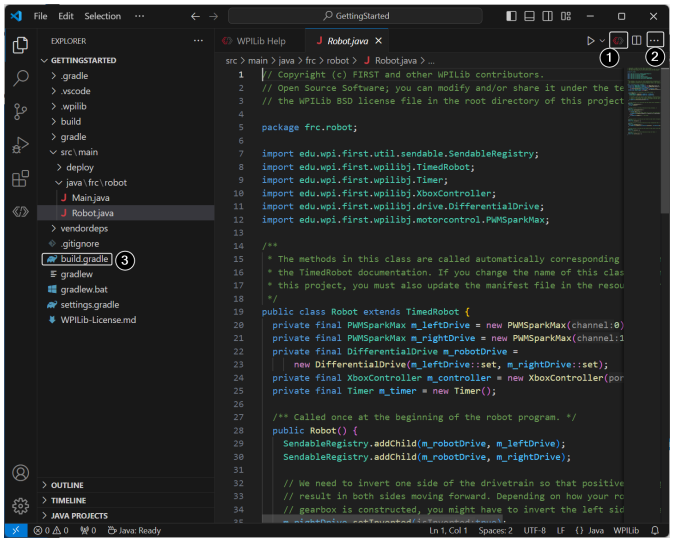
Third party libraries are packaged into Python wheels and uploaded to PyPI (if pure python) and/or WPILib's artifactory. Users can enable them as dependencies either by adding them by adding an explicit dependency for the PyPI package in `requires`. The dependencies are downloaded when `robotpy sync` is executed, and installed on the roboRIO when `robotpy deploy` is executed.

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:

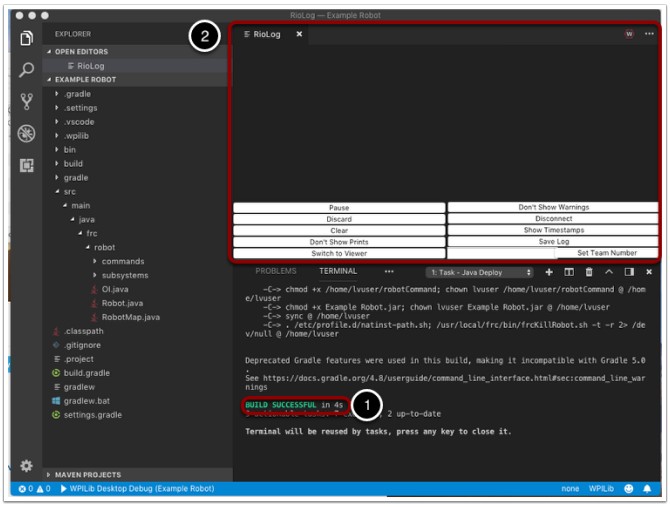
1. Open the Command Palette and enter/select "Build Robot Code"
2. Open the shortcut menu indicated by the ellipses in the top right corner of the VS Code window and select "Build Robot Code"
3. Right-click on the build.gradle file in the project hierarchy and select "Build Robot Code"



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.

> **Warning**
>
> Avoid powering off the robot while deploying robot code. Interrupting the deployment process can corrupt the roboRIO filesystem and prevent your code from working until the roboRIO is *re-imaged*.

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).



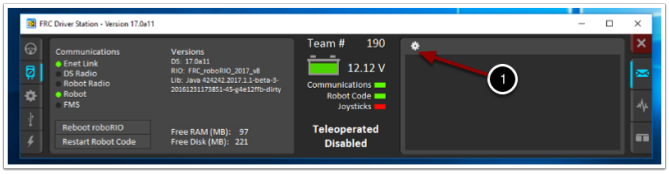
10.6 Viewing Console Output

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.

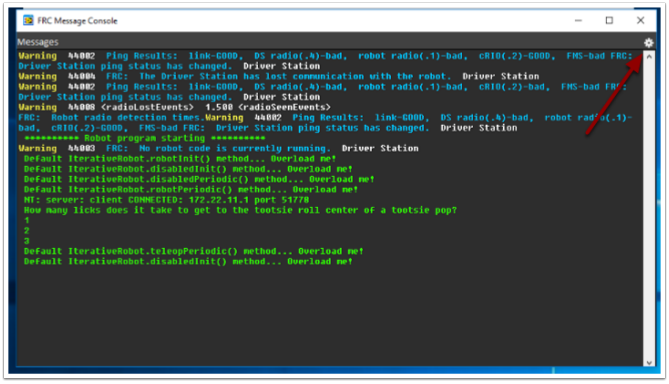
10.6.1 Console Viewer

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

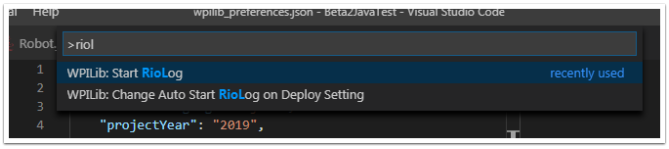


The Console Viewer window displays the output from our robot program in green. The gear in the top right can clear the window and set the level of messages displayed.

10.6.2 Riolog VS Code Plugin

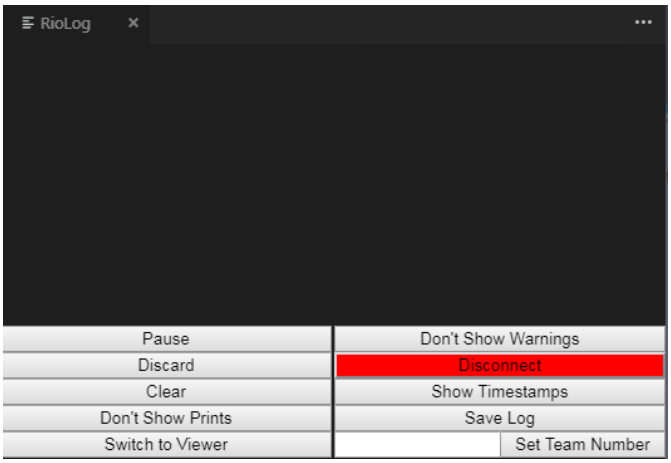
The Riolog plugin is a VS Code view that can be used to view the NetConsole output in VS Code (credit for the original Eclipse version: Manuel Stoeckl, FRC1511).

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

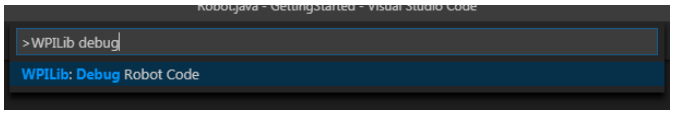
10.7 Debugging a Robot Program

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



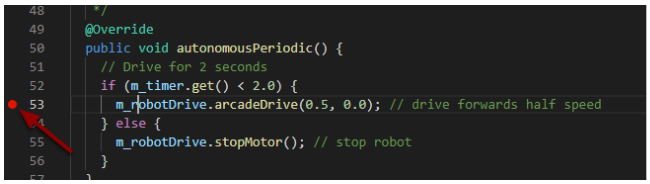
Press `Ctrl+Shift+P` and type `WPILib` or click on the *WPILib Menu Item* to open the Command palette with WPILib pre-populated. Type Debug and select the Debug Robot Code menu item to start debugging. The code will download to the roboRIO and begin debugging.

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

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

C++

```
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 Last Year's Robot Code

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.

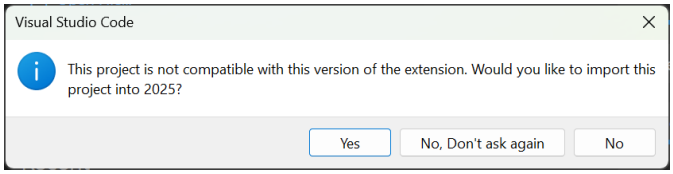
10.8.1 Automatic Import

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.

Important

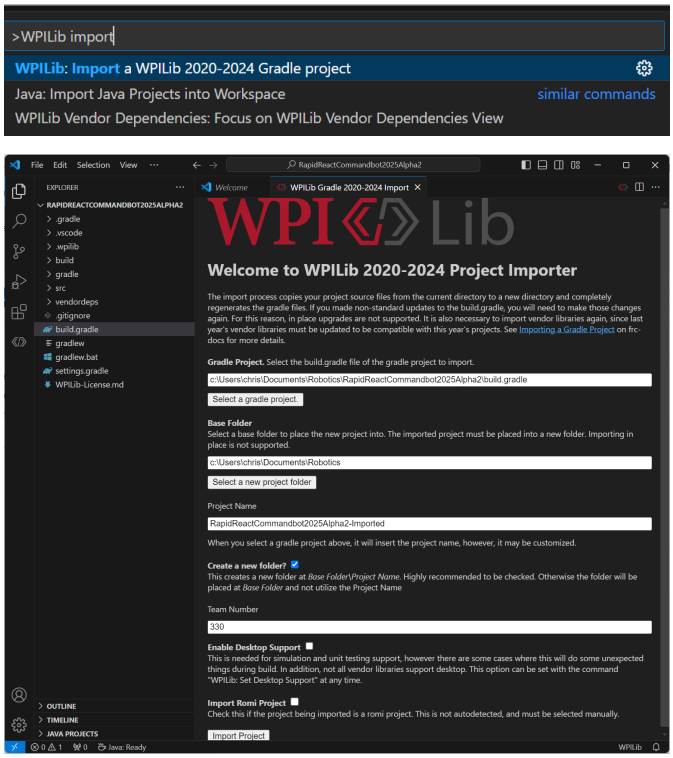
The import process copies your project source files from the current directory to a new directory and completely regenerates the gradle files. If you made non-standard updates to the `build.gradle`, you will need to make those changes again. For this reason, in place upgrades are not supported. It is 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 2020-2025 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.

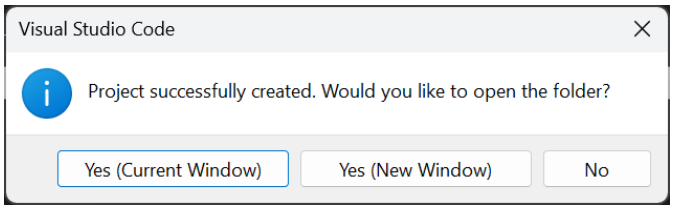
5. **Team Number**: The team number for the project, which will be used for package names within the project and to locate the robot when deploying code.
6. **Enable Desktop Support**: If this is checked, simulation and unit test support is enabled. However, there are some cases where this will do some unexpected things. In addition, all vendor libraries need desktop support which not all libraries do.
7. **Import Romi Project**: If this is checked, the project is imported using the Romi gradle template. This should only be checked for Romi projects.

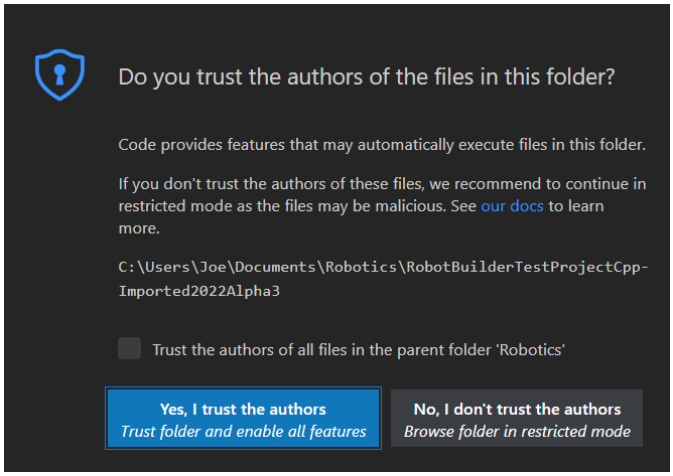
Warning

Creating projects on OneDrive is not supported as OneDrive's caching interferes with the build system. Some Windows installations put the Documents and Desktop folders on OneDrive by default.

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 using the pop-up below or open it later using the `Ctrl+O` (or `Command+O` for macOS) shortcut.

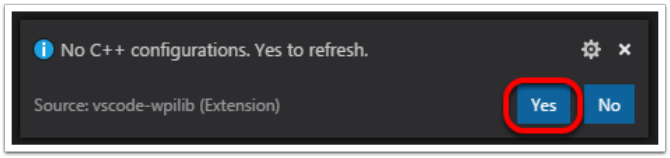




Click *Yes I trust the authors*.

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.

11 Dashboards

11.1 Choosing a Dashboard

A dashboard is a program used to retrieve and display information about the operation of your robot. There are two main types of dashboards that teams may need: driver and programmer dashboards. Some dashboards will try to accommodate both purposes.

11.1.1 Driver Dashboard

During competition the drive team will use this dashboard to get information from the robot. It should focus on conveying key information instantly. This is often best accomplished by using large, colorful, and easy to understand visual elements. Most teams will also use this dashboard to select their autonomous routine.

Take caution to carefully consider what *needs* to be on this dashboard and if there is another better way of communicating that information. Any members of the drive team (especially the driver) looking at the dashboard takes their focus away from the match. Using *LEDs* to indicate the state of your robot is a good example of a way to communicate useful information to the driver without having to take their eyes off the robot.

11.1.2 Programming Dashboard

This dashboard is designed for debugging code and analyzing data from the robot. It supports the monitoring of a wide variety of information simultaneously, prioritizing function and utility over simplicity or ease of use. This functionality often includes complex data visualization and graphing across extended periods. In scenarios where there is an overwhelming amount of data to review, real-time analysis becomes challenging. The capability to examine past data and replay it proves to be extremely beneficial. While some dashboards may log data transmitted to them, *on-robot telemetry* using the `DataLog` class simplifies the process.

11.1.3 Included Dashboards (oldest to newest)

> **Note**
>
> SmartDashboard and Shuffleboard have a long history of aiding FRC teams. However, they do not have a person to maintain them so are not receiving bug fixes or improvements. Notably, Shuffleboard may experience performance issues on some machines under certain scenarios. PRs from external contributors will be reviewed. They are deprecated and will be removed for the 2027 season.

LabVIEW Dashboard (Driver / Programming) - easy to use and provides a lot of features straight out of the box like: camera streams, autonomous selection, and joystick feedback. It can be customized using LabVIEW by creating a new Dashboard project. While it *can be used* by Java or C++ teams, they generally prefer SmartDashboard or Shuffleboard which can be customized in their respective language.

SmartDashboard (Driver) - simple and efficient dashboard that uses relatively few computer resources. It does not have the fancy look or some of the features Shuffleboard has, but it displays network tables data with a variety of widgets without bogging down the driver station computer.

Shuffleboard (Driver / Programming) - straightforward and easily customizable dashboard. It displays network tables data using a variety of widgets that can be positioned and controlled with robot code. It includes many extra features like: tabs, recording / playback, and advanced custom widgets.

Glass (Programming) - 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, with a focus on high performance real time plotting.

AdvantageScope (Programming / Third Party) - robot diagnostics, log review/analysis, and data visualization application. It reads the WPILib Data Log (`.wpilog`) and Driver Station Log (`.dslog` / `.dsevents`) file formats, plus live robot data viewing.

Elastic (Driver / Third Party) - simple and modern Shuffleboard alternative made by Team 353. It is meant to serve as a dashboard for competition but can also be used for testing. It features draggable and resizable card widgets.

11.1.4 External Dashboards (All Third Party)

FRC Web Components (Driver) - A web-based dashboard that can be installed as a standalone application, or as a JavaScript package for custom dashboard solutions.

QDash (Driver) - described as a reliable, high-performance, low-footprint dashboard. QDash has been specifically designed to use as few resources as possible without sacrificing features or aesthetics.

Reflect (Driver) - a modern, web-based FRC dashboard focused on performance and usability with customizable widgets. Made by team 2702.

11.2 Shuffleboard

Shuffleboard is a straightforward and easily customizable driveteam focused dashboard. It displays network tables data using a variety of widgets that can be positioned and controlled with robot code. It includes many extra features like: tabs, recording / playback, and advanced custom widgets.

> **Warning**
>
> Shuffleboard is deprecated and will be removed for 2027 due to its lack of a maintainer and resource utilization issues. Users can find *additional modern dashboard options here*

11.2.1 Shuffleboard - Getting Started

> **Warning**
>
> Shuffleboard is deprecated and will be removed for 2027 due to its lack of a maintainer and resource utilization issues. Users can find *additional modern dashboard options here*

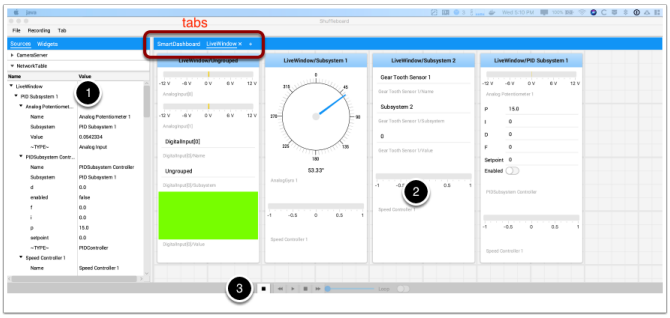
Tour of Shuffleboard

> **Warning**
>
> Shuffleboard is deprecated and will be removed for 2027 due to its lack of a maintainer and resource utilization issues. Users can find *additional modern dashboard options here*

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++, Java, and Python 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:

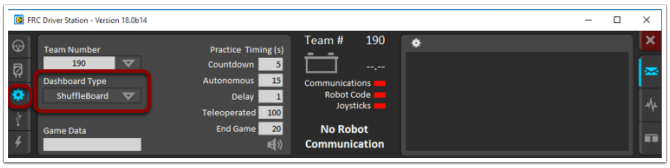
- Graphics is based on **JavaFX**, the Java graphics standard. Each of the components has an associated style sheet so it becomes possible to have different "skins" or "themes" for Shuffleboard. We supply default light and dark themes.
- Shuffleboard supports **multiple sheets for the display of your 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 can be found in *Custom Widgets*.



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 in the *YEAR WPILib tools* folder on the Windows Desktop.
3. You can start from with Visual Studio Code by pressing `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 *Shuffleboard*.
4. You can run it by double-clicking on the shuffleboard.XXX executable (where XXX is `.exe` on Windows and no extension 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.

5. 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 `.exe` (Windows) and no extension (macOS/Linux) executables 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());
```

C++

```
frc::SmartDashboard::PutNumber("Joystick X value", joystick1.getX());
```

PYTHON

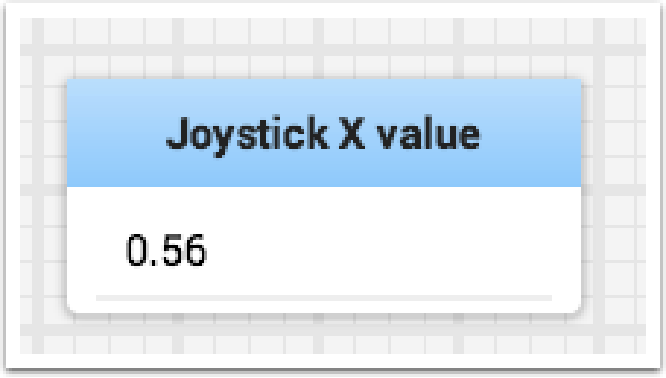
```
from wpilib import SmartDashboard
SmartDashboard.putNumber("Joystick X value", joystick1.getX())
```

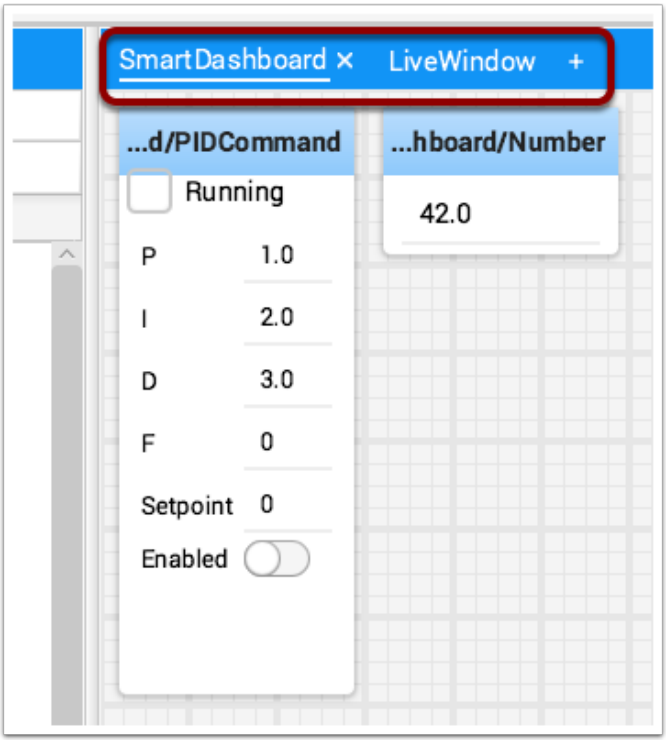
to see a field displayed with the label "Joystick X value" and a value of the X value of the joystick. Each time this line of code is executed, a new joystick value will be sent to Shuffleboard. Remember: you must write the joystick value whenever you want to see an updated value. Executing this line once at the start of the program will only display the value once at the time the line of code was executed.

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


Joystick X value
0.56


SmartDashboard
LiveWindow
...d/PIDCommand
Running
P 1.0
I 2.0
D 3.0
F 0
Setpoint 0
Enabled
...hboard/Number
42.0

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());
}
```

C++

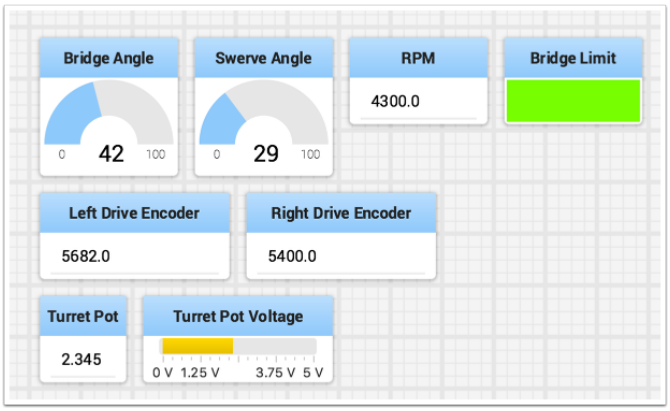
```
frc::SmartDashboard::PutBoolean("Bridge Limit", bridgeTipper.AtBridge());
frc::SmartDashboard::PutNumber("Bridge Angle", bridgeTipper.GetPosition());
frc::SmartDashboard::PutNumber("Swerve Angle", drivetrain.GetSwerveAngle());
frc::SmartDashboard::PutNumber("Left Drive Encoder", drivetrain.GetLeftEncoder());
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());
```

PYTHON

```
from wpilib import SmartDashboard
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())
```

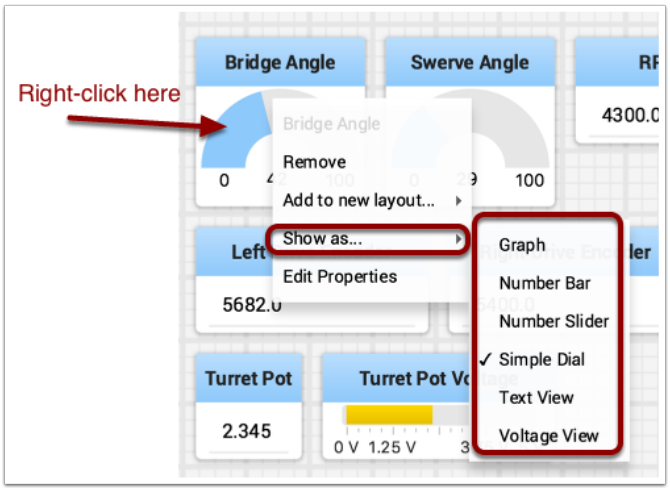
You can write Boolean, Numeric, or String values to Shuffleboard by simply calling the correct method for the type and including the name and the value of the data, no additional code is required.

- 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

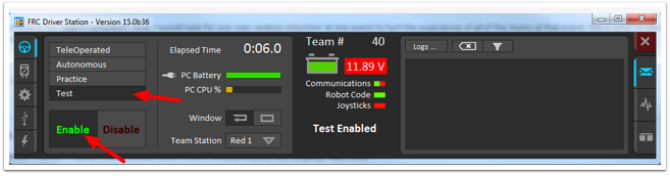
Depending on the data type of the values being sent to Shuffleboard you can often change the display format. In the previous example you can see that number values were displayed as either decimal numbers, a dial to better represent angles, and as a voltage view for the turret potentiometer. To set the 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.

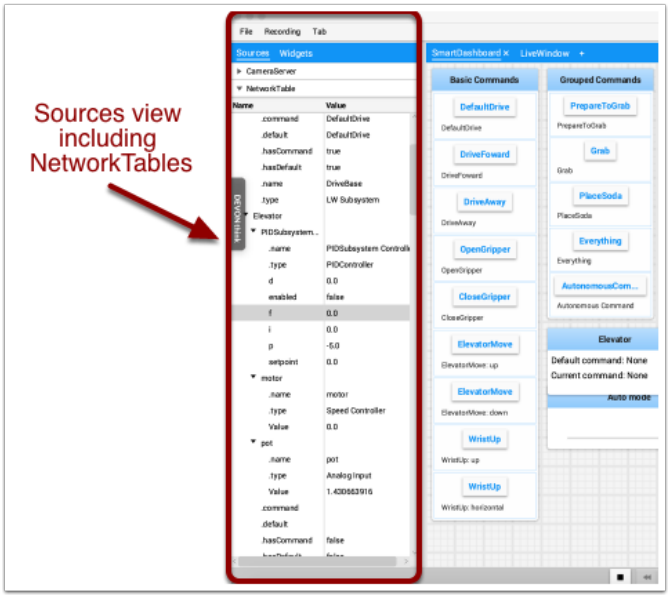
Setting test mode



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.

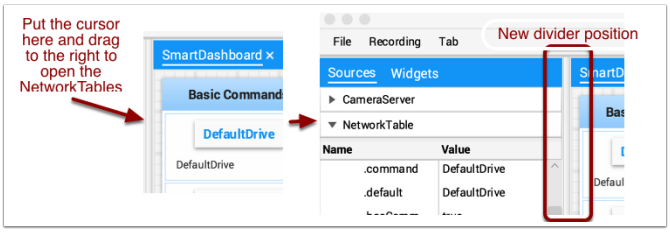
Getting data from the Sources view

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.

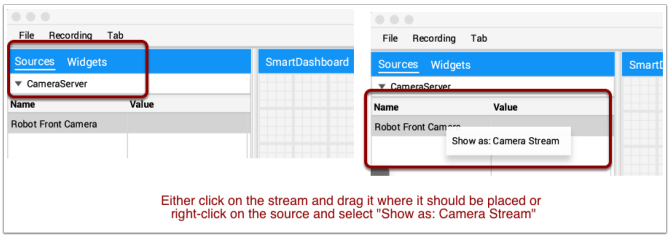


Displaying Camera Streams

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.

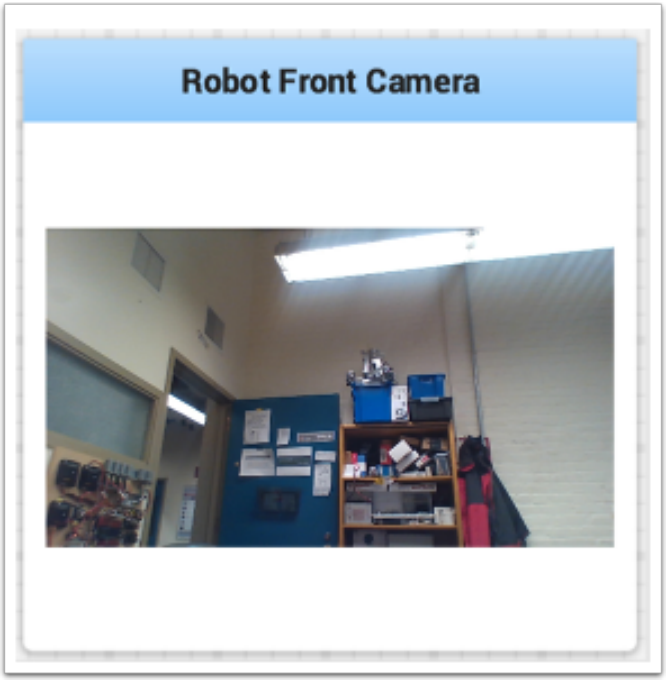
Adding a Camera Stream

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.

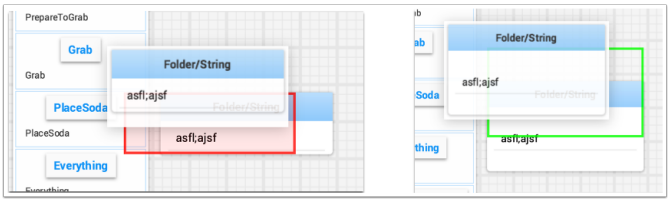


Working with widgets

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.

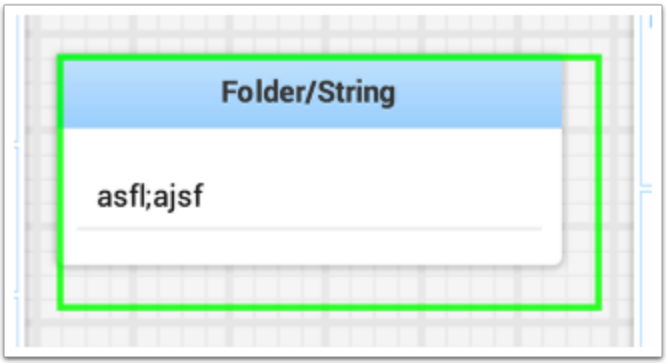
Moving widgets

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

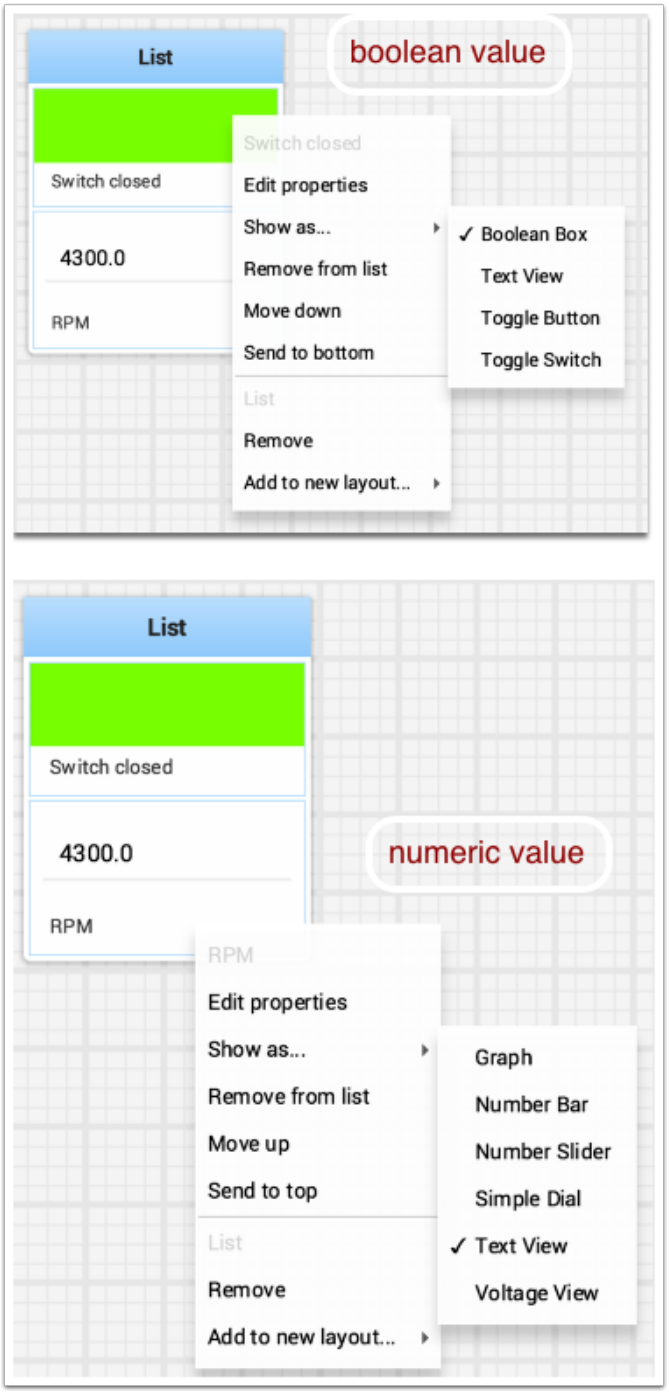
Shuffleboard is very rich in display types depending on the data published from the robot. It will automatically choose a default display type, but you might want to change it depending on the application. To see what the possible displays are for any widget, right-click on the widget and select the “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.

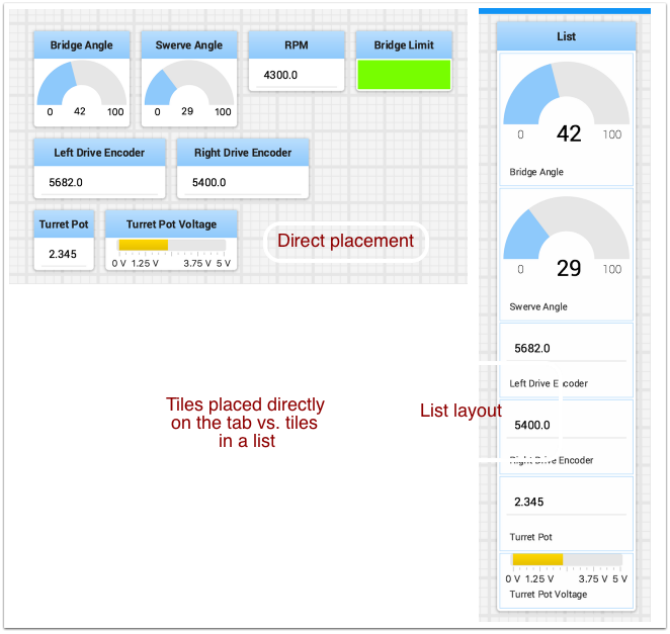

List
boolean value
Switch closed
4300.0
RPM
Switch closed
Edit properties
Show as...
Remove from list
Move down
Send to bottom
List
Remove
Add to new layout...
✓ Boolean Box
Text View
Toggle Button
Toggle Switch
List
Switch closed
4300.0
RPM
numeric value
RPM
Edit properties
Show as...
Remove from list
Move up
Send to top
List
Remove
Add to new layout...
Graph
Number Bar
Number Slider
Simple Dial
✓ Text View
Voltage View



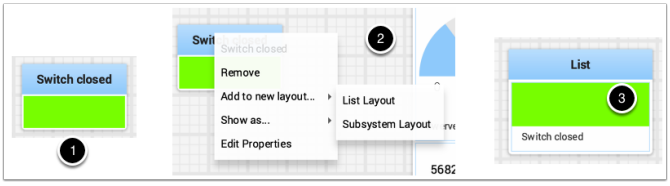
Working with Lists

Lists in Shuffleboard are sets of tiles grouped together in a vertical layout, making it visually obvious that those tiles are related. In addition, tiles in lists take up less screen space than individual tiles:

- Tiles in lists don't have individual header labels; they instead have smaller labels within their list entries.
- Individual tiles placed together create gaps between one another; lists have smaller gaps between tiles.



Creating a list



A list can be created as follows:

1. Right-click on the tile that should be first in the list.
2. Select "Add to new layout…", then "List Layout" from the popup menu.
3. A new list will be created labeled "List", and the tile will be at the top of it.

Note that tiles in lists do not have header labels; their label is at the bottom of their list entry.

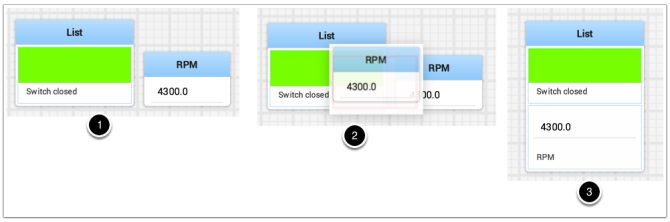
Adding tiles to/removing tiles from a list

A tile can be **added** to an existing list as follows:

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 the current list size is too small to show it, the tile will be added to the list off-screen and a vertical scrollbar will be added if not already present.

A tile can be **removed** from a list by following the process in reverse:

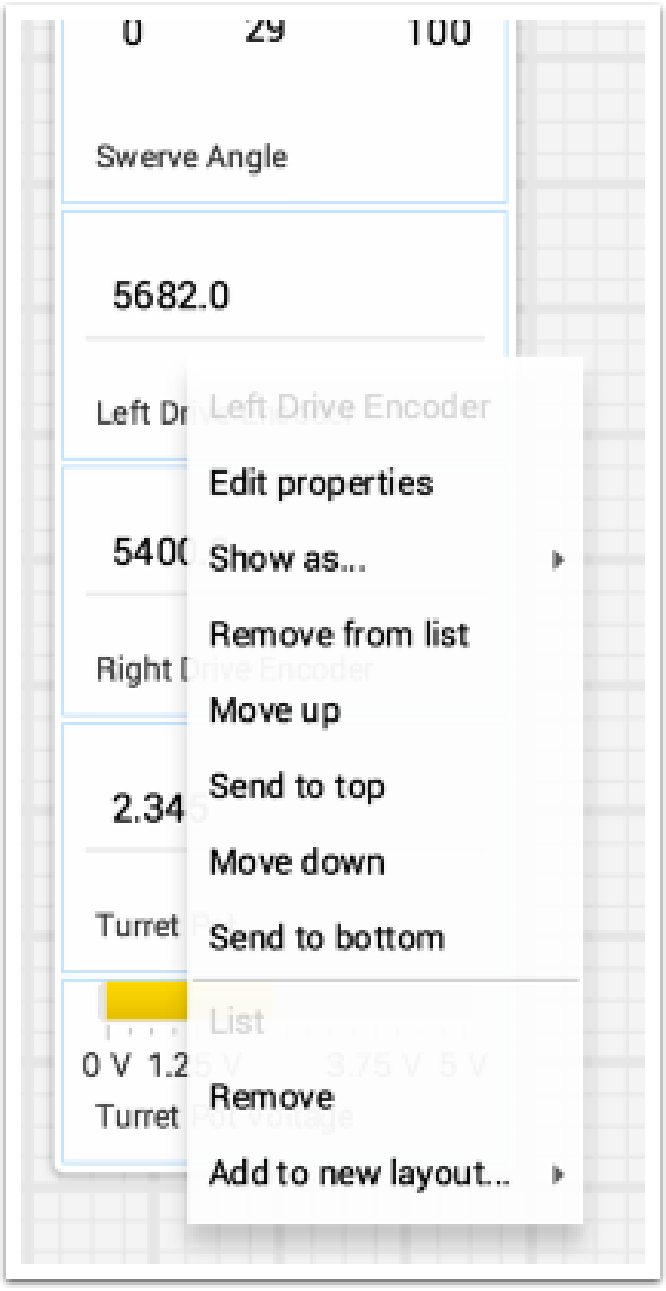
1. Identify the list and the tile within it to be removed.
2. Drag the tile out of the list and place it anywhere with free space.
3. The tile will be removed from the list and placed at that location.



Rearranging tiles in a list

Tiles in a list can be rearranged by right-clicking on the tile and selecting:

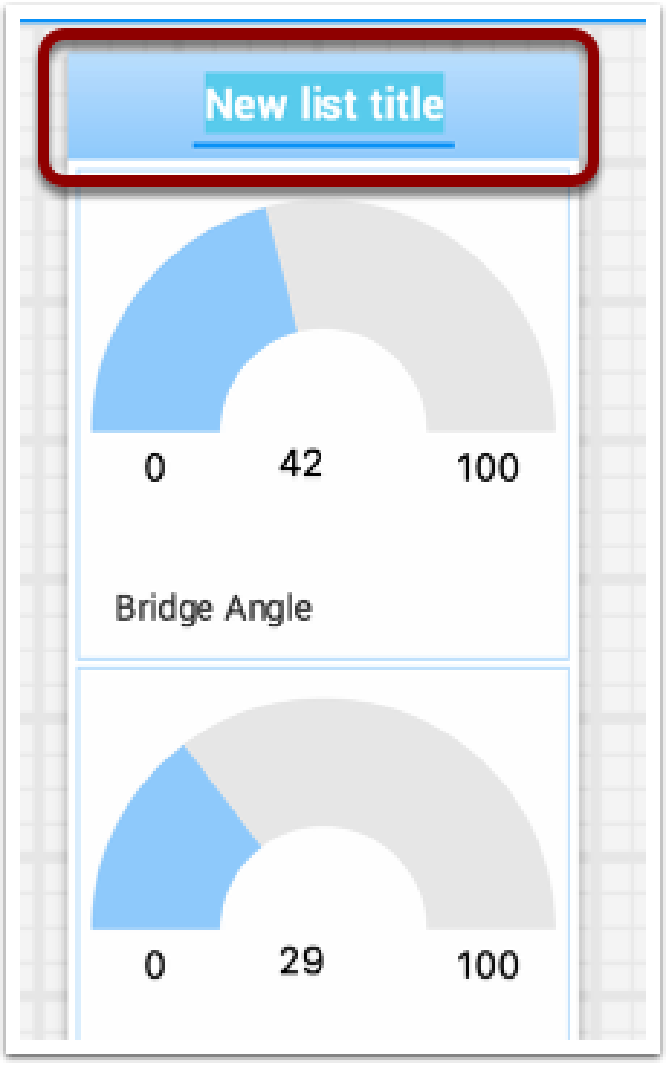
- *Move up* moves the tile **towards** the *top* of the list.
- *Move down* moves the tile **towards** the *bottom* of the list.
- *Send to top* moves the tile **to** the *top* of a list.
- *Send to bottom* moves the tile **to** the *bottom* of a list.


0
29
100
Swerve Angle
5682.0
Left Dr
Left Drive Encoder
Edit properties
540(
Show as...
Remove from list
Right
Move up
Send to top
2.34
Move down
Turret
Send to bottom
List
0 V 1.2
Remove
Turret
Add to new layout...

- **There are two buttons labeled *Remove*, and each button does:**
  - The **top** *Remove* button (above the pinline; section of dropdown with grayed-out *tile* label) **deletes** the *tile* from the Shuffleboard layout.
  - The **bottom** *Remove* button (below the pinline; section of dropdown with grayed-out *list* label) **deletes** the *list and all tiles inside it* from the Shuffleboard layout.
  - If you want to take an entry out of a list without deleting it, see *Adding tiles to/removing tiles from a list*.

Renaming a list

You can rename a list by double-clicking on the list label and changing the name. Click outside the label to save changes.

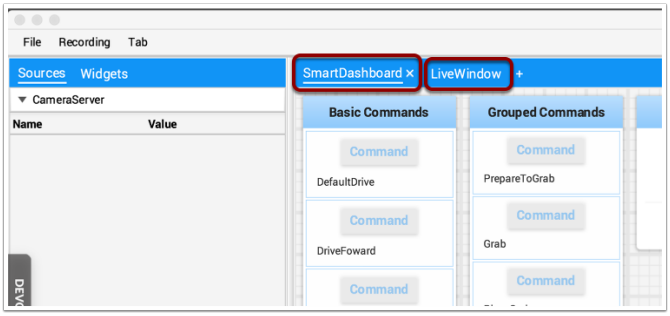


Creating and manipulating tabs

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.



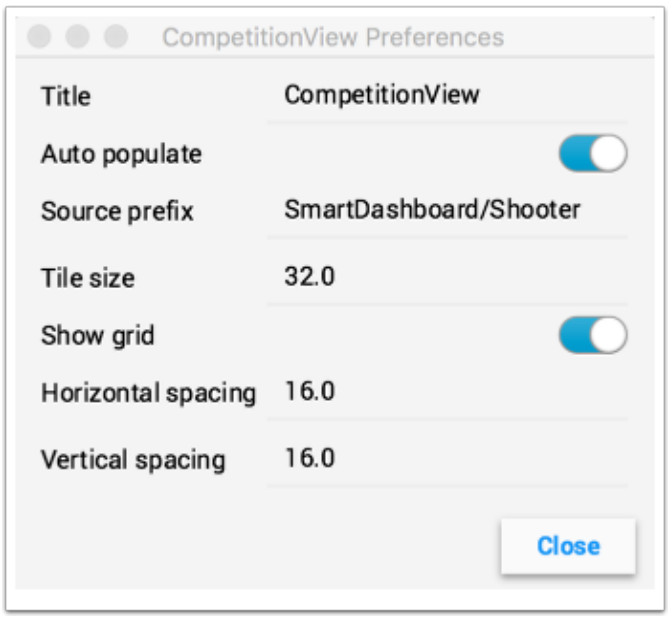
On the SmartDashboard tab all the values that are written using the SmartDashboard.putType() set of methods. On the LiveWindow tab all the autogenerated debugging values are shown.

Switching between tabs

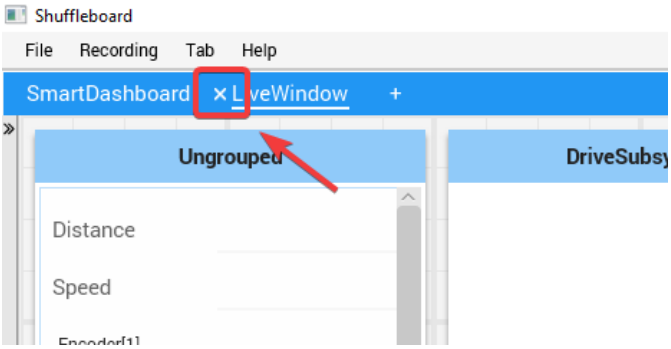
You can switch between tabs clicking on the tab label at the top of the window. In the case above, simply click on SmartDashboard or LiveWindow to see the values that are associated with each tab.

Adding and Hiding 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.



You can hide tabs by clicking the minus(-) symbol to the left of the selected tab name. Since tabs are generated based on the relevant *NetworkTable*, it is not possible to permanently delete them without deleting the table.



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).

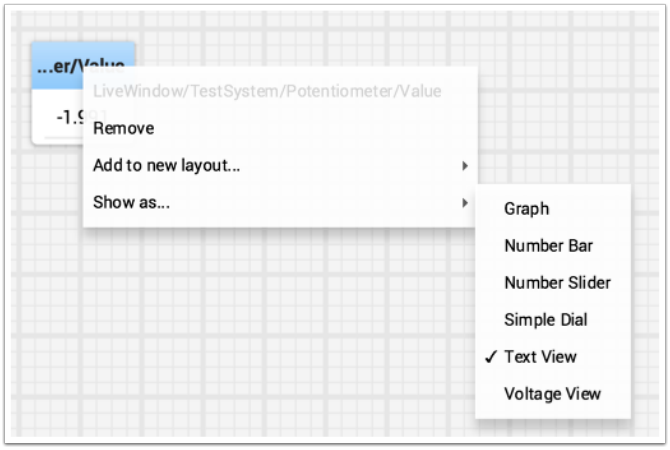
Moving widgets between tabs

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.

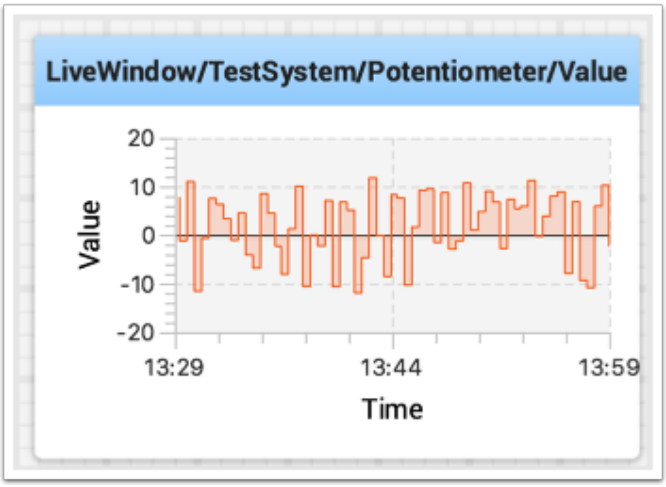
Working with Graphs

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

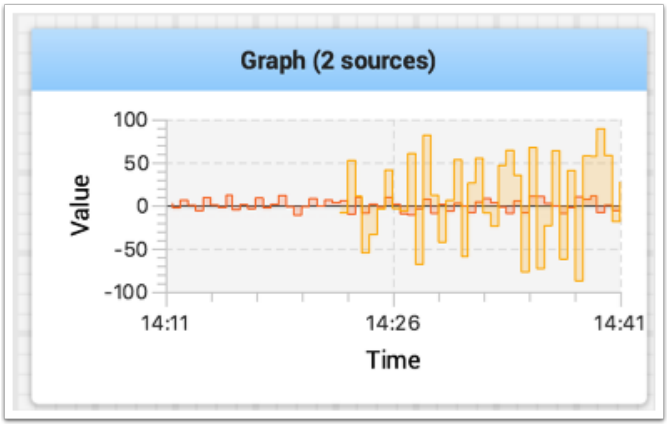


The graph widget shows line plots of the value that you selected. It will automatically set the scale and the default time interval that the graph will show will be 30 seconds. You can change that in the setting for the graph (see below).

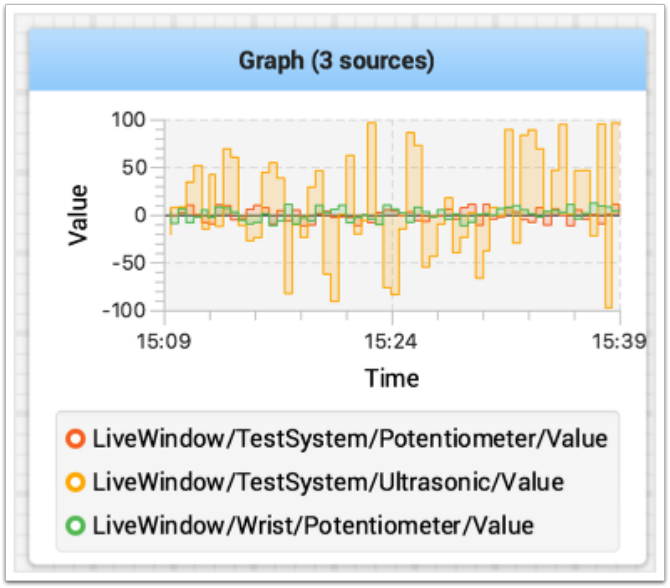


Adding Additional Data Values

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.



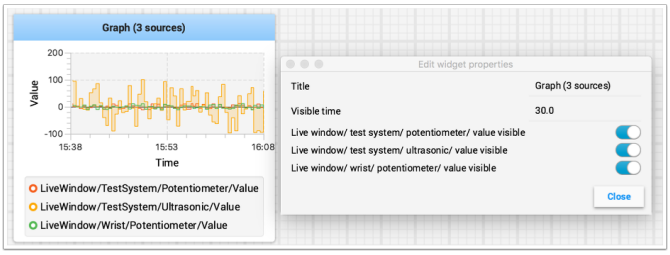
You can resize the graph vertically to view the legend if it is not displayed as shown in the image below. The legend shows all the sources that are used in the plot.



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).

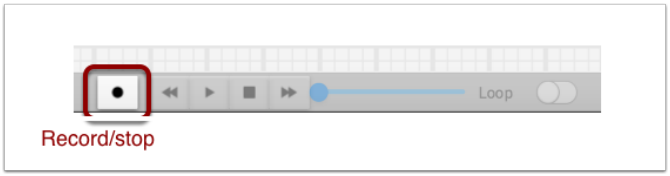


Recording and Playback

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

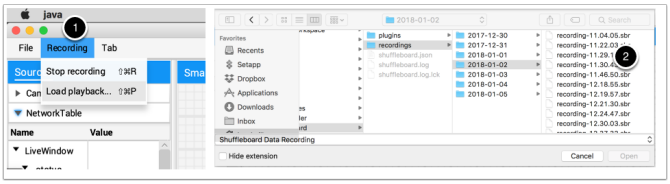
When shuffleboard starts it begins recording to all the NetworkTables values are recorded and continues until stopped by hitting the record/stop button in the recorder controls as shown below. If a new recording is desired, such as when a new piece of code or mechanical system is being tested, stop the 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 recoding is currently running so click it to stop the recording.



Playing a Recording

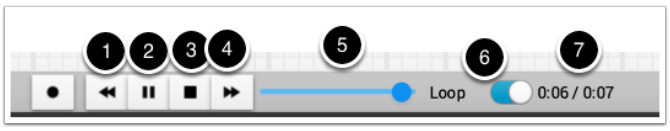
Previous recordings can be played back by:

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

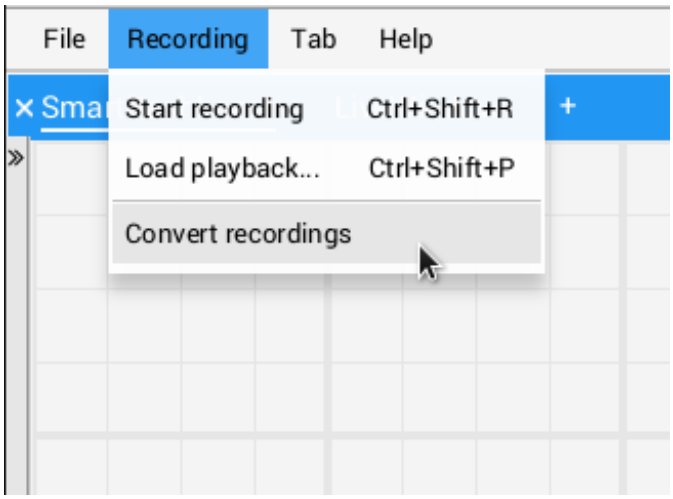
Selecting the recoding file will begin playback of that file. While the recording is playing the recording controls will show the current time within the recording as well as the option to loop the recording while watching it. When the recording is being played back the "transport" controls will allow the playback to be controlled.



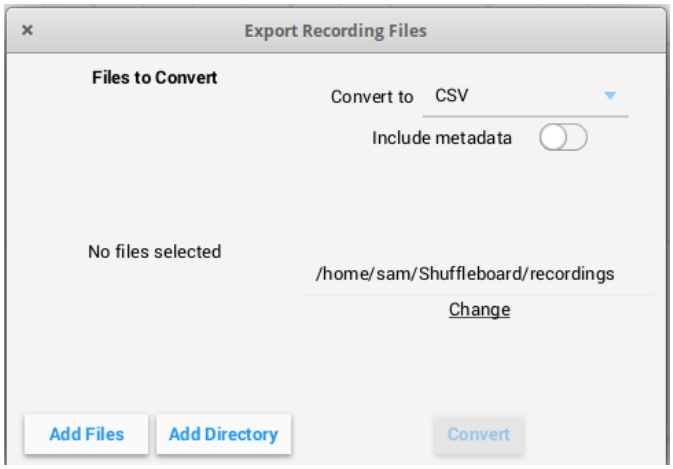
The controls work as follows:

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

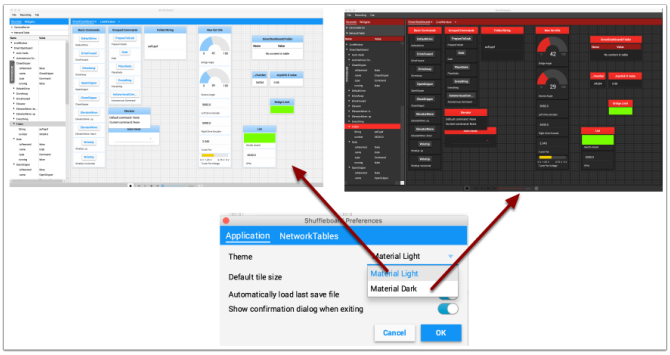
Graphs won't display properly while scrubbing the timeline but if it is playing through where the graph history can be captured by the graph then they will display as in the original run.

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.

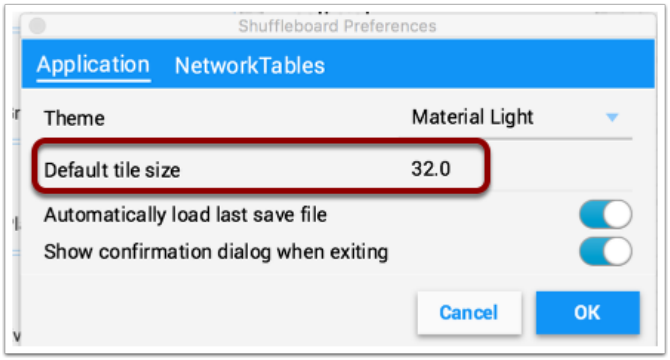
Setting the theme

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.



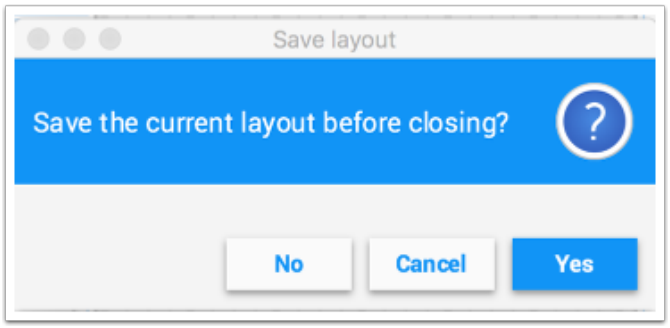
Setting the default tile size

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.

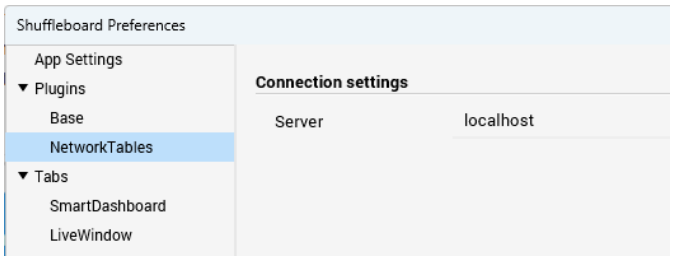


Working with the layout save files

You can save your layout 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 display a "Save layout" window to remind you to save the layout on exit, if the layout has changed. 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 loose 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.

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?

There is no active maintainer of Shuffleboard, but we are accepting pull requests. Bugs, issues, and feature requests can be added on the Shuffleboard GitHub page by creating an issue. 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?

Custom Widgets has a large amount of documentation on extending the program with custom plugins. Sample plugin projects that can be used for additional custom widgets and themes can be found on the Shuffleboard GitHub page.

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	./gradlew :app:run
Building the APIs and utility classes for plugin creation	./gradlew :api:shadowJar
Building the complete application jar file	./gradlew :app:shadowJar

11.2.2 Shuffleboard - Layouts with Code

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

```
ShuffleboardTab tab = Shuffleboard.getTab("Tab Title");
```

C++

```
ShuffleboardTab& tab = Shuffleboard::GetTab("Tab Title");
```

PYTHON

```
from wpilib.shuffleboard import Shuffleboard
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

```
Shuffleboard.selectTab("Tab Title");
```

C++

```
Shuffleboard::SelectTab("Tab Title");
```

PYTHON

```
from wpilib.shuffleboard import Shuffleboard
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

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

```
Shuffleboard.getTab("Numbers")
     .add("Pi", 3.14);
```

C++

```
frc::Shuffleboard::GetTab("Numbers")
     .Add("Pi", 3.14);
```

PYTHON

```
from wpilib.shuffleboard import Shuffleboard
Shuffleboard.getTab("Tab Title").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 GenericEntry distanceEntry =
      tab.add("Distance to target", 0)
         .getEntry();
   public void calculate() {
      double distance = ...;
      distanceEntry.setDouble(distance);
   }
}
```

PYTHON

```
from wpilib.shuffleboard import Shuffleboard
def robotInit(self):
   tab = Shuffleboard.getTab("Vision")
   self.distanceEntry = tab.add("Distance to target", 0).getEntry()
def teleopPeriodic(self):
   distance = self.encoder.getDistance()
   self.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);
```

C++

```
frc::Shuffleboard::GetTab("Drive")
     .AddPersistent("Max Speed", 1.0);
```

PYTHON

```
from wpilib.shuffleboard import Shuffleboard
(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);
```

C++

```
frc::Shuffleboard::GetTab("Tab Title")
     .Add("Sendable Title", mySendable);
```

PYTHON

```
from wpilib.shuffleboard import Shuffleboard
(Shuffleboard.getTab("Tab Title")
      .add("Sendable Title", mySendable))
```

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 GenericEntry maxSpeed =
      tab.add("Max Speed", 1)
         .getEntry();
   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);
   }
}
```

PYTHON

```
import commands2
import wpilib.drive
from wpilib.shuffleboard import Shuffleboard
class DriveSubsystem(commands2.SubsystemBase):
   def __init__(self) -> None:
      super().__init__()
      tab = Shuffleboard.getTab("Drive")
      self.maxSpeed = tab.add("Max Speed", 1).getEntry()
      this.robotDrive = ...
   def drive(self, left: float, right: float):
      # Retrieve the maximum speed from the dashboard
      max = self.maxSpeed.getDouble(1.0)
      self.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.

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();
```

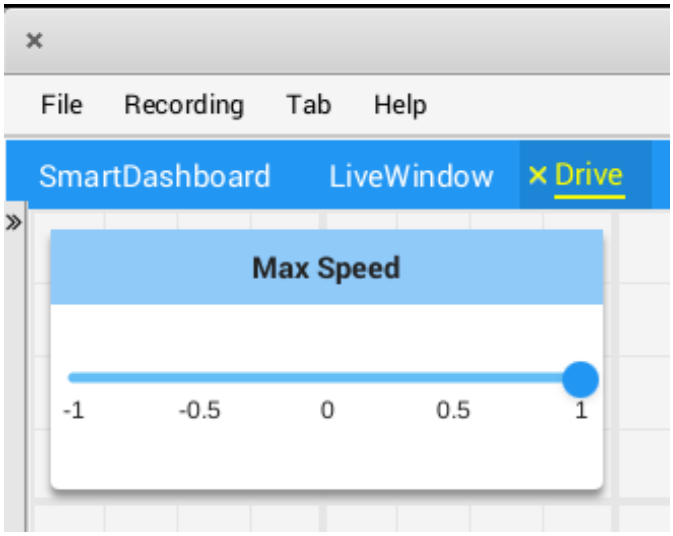
C++

```
frc::Shuffleboard::GetTab("Drive")
   .Add("Max Speed", 1)
   .WithWidget(frc::BuiltInWidgets::kNumberSlider) // specify the widget here
   .GetEntry();
```

PYTHON

```
from wpilib.shuffleboard import Shuffleboard
from wpilib.shuffleboard import BuiltInWidgets
(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

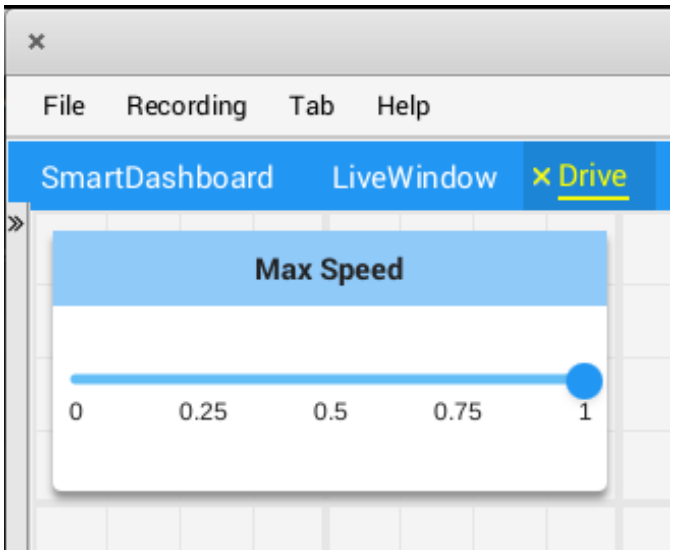
```
Shuffleboard.getTab("Drive")
   .add("Max Speed", 1)
   .withWidget(BuiltInWidgets.kNumberSlider)
   .withProperties(Map.of("min", 0, "max", 1)) // specify widget properties here
   .getEntry();
```

C++

```
frc::Shuffleboard::GetTab("Drive")
   .Add("Max Speed", 1)
   .WithWidget(frc::BuiltInWidgets::kNumberSlider)
   .WithProperties({ // specify widget properties here
      {"min", nt::Value::MakeDouble(0)},
      {"max", nt::Value::MakeDouble(1)}
   })
   .GetEntry();
```

PYTHON

```
from wpilib.shuffleboard import Shuffleboard
from wpilib.shuffleboard import BuiltInWidgets
(Shuffleboard.getTab("Drive")
   .add("Max Speed", 1)
   .withWidget(BuiltInWidgets.kNumberSlider)
   .withProperties(map("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.

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

```
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
```

C++

```
frc::Shuffleboard::GetTab("Pre-round")
  .Add("Auto Mode", autoModeChooser)
  .WithSize(2, 1)
  .WithPosition(0,0);
```

PYTHON

```
from wpilib.shuffleboard import Shuffleboard
(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
```

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

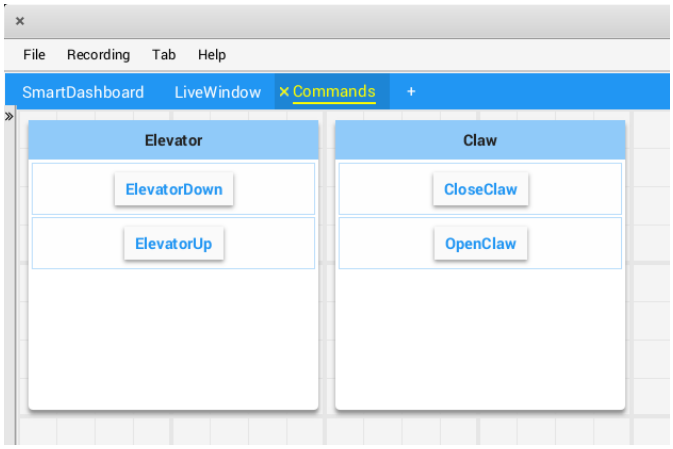
```
ShuffleboardLayout elevatorCommands = Shuffleboard.getTab("Commands")
  .getLayout("Elevator", BuiltInLayouts.kList)
  .withSize(2, 2)
  .withProperties(Map.of("Label position", "HIDDEN")); // hide labels for commands
   elevatorCommands.add(new ElevatorDownCommand());
elevatorCommands.add(new ElevatorUpCommand());
```

C++

```
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);
```

PYTHON

```
from wpilib.shuffleboard import Shuffleboard
from wpilib.shuffleboard import BuiltInLayouts
(elevatorCommands = Shuffleboard.getTab("Commands")
  .getLayout("Elevator", BuiltInLayouts.kList)
  .withSize(2, 2)
  .withProperties(map("Label position", "HIDDEN"))) # hide labels for commands
   elevatorCommands.add(ElevatorDownCommand())
elevatorCommands.add(ElevatorUpCommand())
```



11.2.3 Shuffleboard - Advanced Usage

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

```
SmartDashboard.putData(subsystem-reference);
```

C++

```
SmartDashboard::PutData(subsystem-pointer);
```

PYTHON

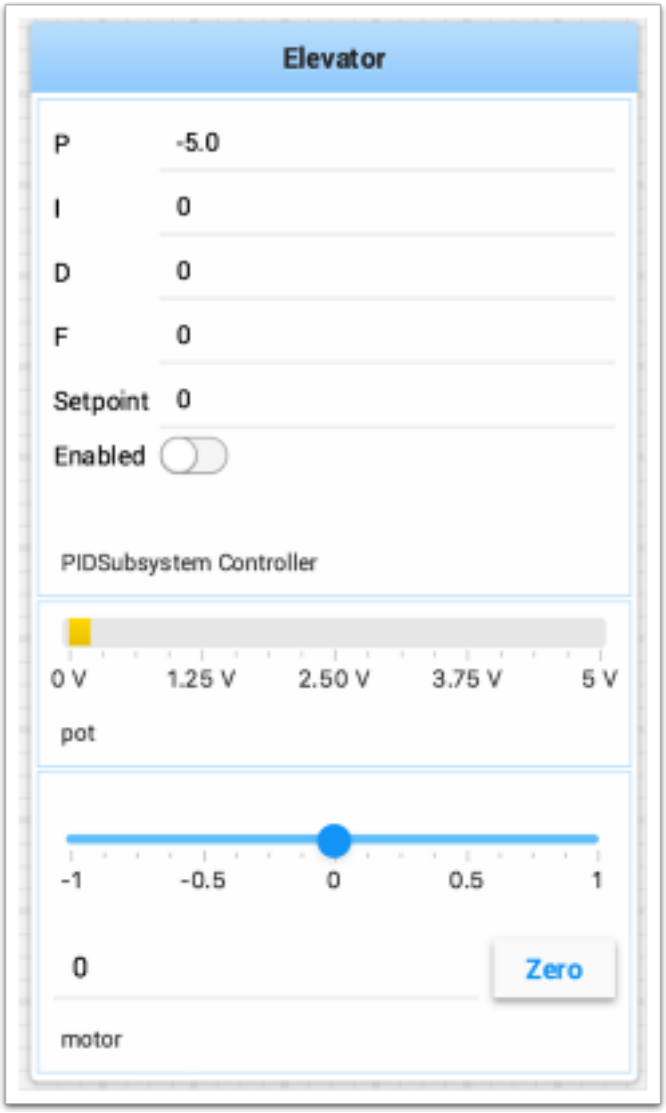
```
from wpilib import SmartDashboard
SmartDashboard.putData(subsystem-reference)
```

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

```
SmartDashboard.putData("ElevatorMove: up", new ElevatorMove(2.7));
```

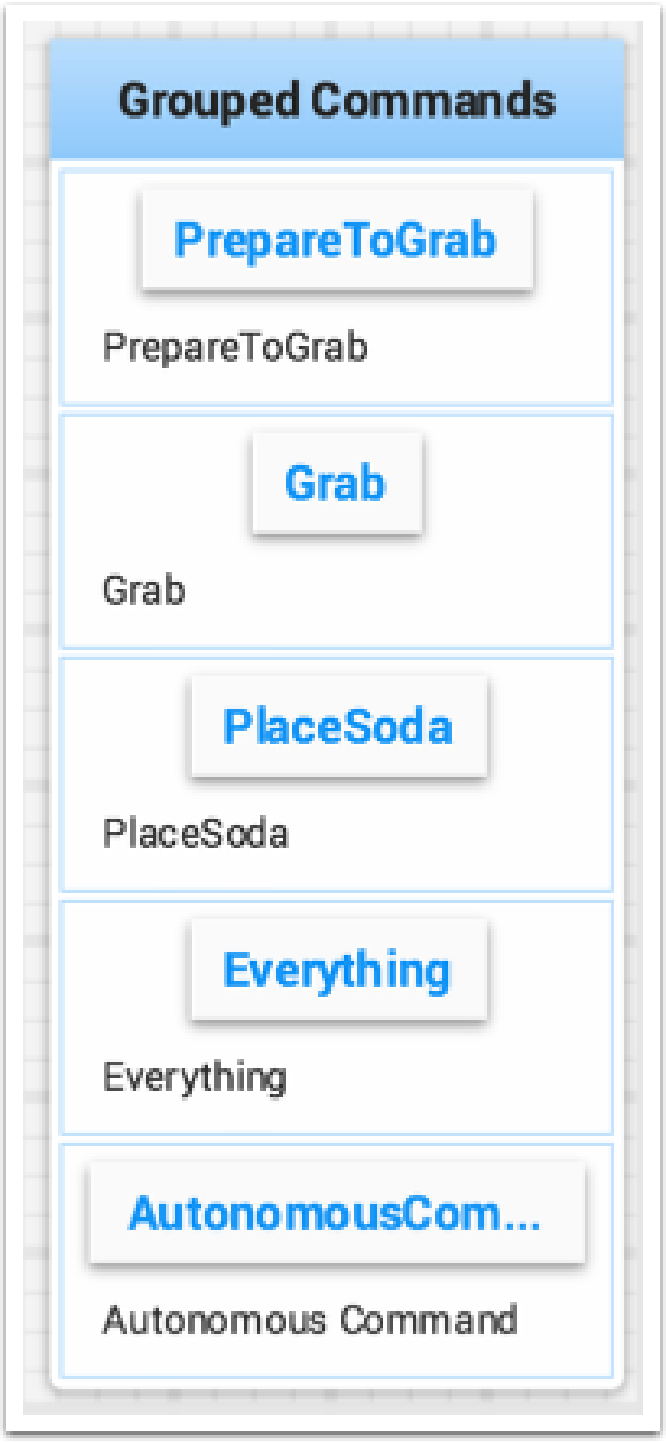
C++

```
SmartDashboard::PutData("ElevatorMove: up", new ElevatorMove(2.7));
```

PYTHON

```
from wpilib import SmartDashboard
SmartDashboard.putData("ElevatorMove: up", 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.


Grouped Commands
PrepareToGrab
PrepareToGrab
Grab
Grab
PlaceSoda
PlaceSoda
Everything
Everything
AutonomousCom...
Autonomous Command

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 that explain the control algorithms described here. The PID algorithm converts sensor values into motor speeds by:

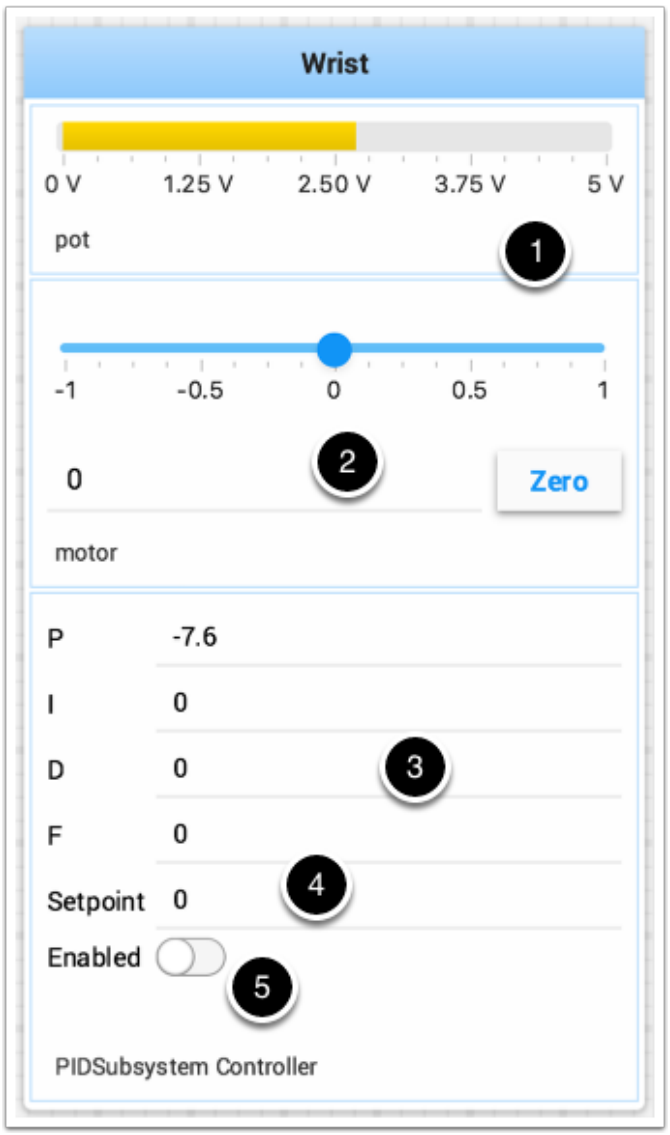
1. Reading sensor values to determine how far the robot or mechanism from the desired setpoint. The setpoint is the sensor value that corresponds to the expected goal. For example, a robot arm with a wrist joint should be able to move to a specified angle very quickly and stop at that angle as indicated 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 Kp, Ki, and Kd 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 (Kp) 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 (Ki) 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 (kd) to scale it to match the rest of the system.

Tuning the PID Controller

Tuning the PID controller consists of adjusting constants for accurate results. 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

```
ShuffleboardTab tab = Shuffleboard.getTab("Shooter");
GenericEntry shooterEnable = tab.add("Shooter Enable", false).getEntry();
// Command Example assumed to be in a PIDSubsystem
new NetworkButton(shooterEnable).onTrue(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));
}
```

C++

```
frc::ShuffleboardTab& tab = frc::Shuffleboard::GetTab("Shooter");
nt::GenericEntry& shooterEnable = *tab.Add("Shooter Enable", false).GetEntry();
// Command-based assumed to be in a PIDSubsystem
frc2::NetworkButton(shooterEnable).OnTrue(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));
}
```

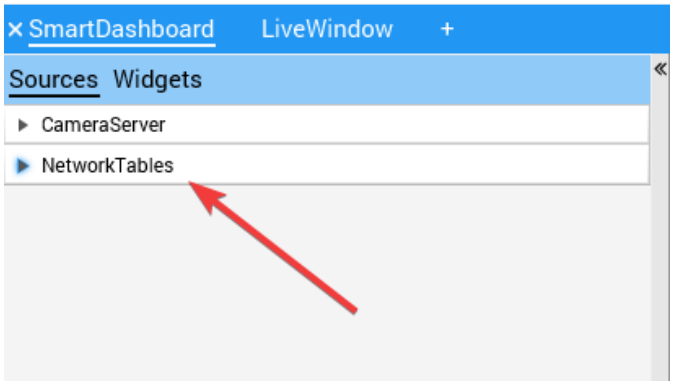
PYTHON

```
from wpilib.shuffleboard import Shuffleboard
tab = Shuffleboard.getTab("Shooter")
shooterEnable = tab.add("Shooter Enable", false).getEntry()
# Command Example assumed to be in a PIDSubsystem
NetworkButton(shooterEnable).onTrue(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))
```

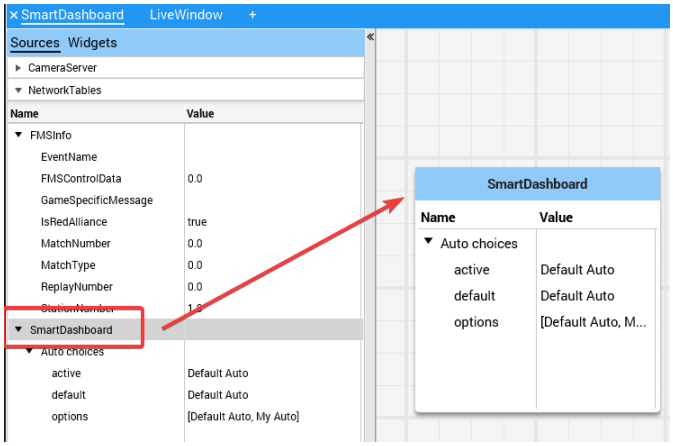
Viewing Hierarchies of Data

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.



11.2.4 Shuffleboard - Custom Widgets

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 ntcore. Since the `LiveWindow`, `SmartDashboard`, and `Shuffleboard` 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.

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

The easiest way to build plugins is to utilize the *example-plugins* folder in the shuffleboard source tree. Clone Shuffleboard with `git clone https://github.com/wpilibsuite/shuffleboard.git`, and checkout the version that corresponds to the WPILib version you have installed (e.g. 2023.2.1). `git checkout v2023.2.1`

Put your plugin in the `example-plugins\PLUGIN-NAME` directory. Copy the `custom-data-and-widget.gradle` from `example-plugins\custom-data-and-widget` and rename to match your plugin name. Edit `settings.gradle` in the shuffleboard root directory to add `include "example-plugins:PLUGIN-NAME"`

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", name = "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`.

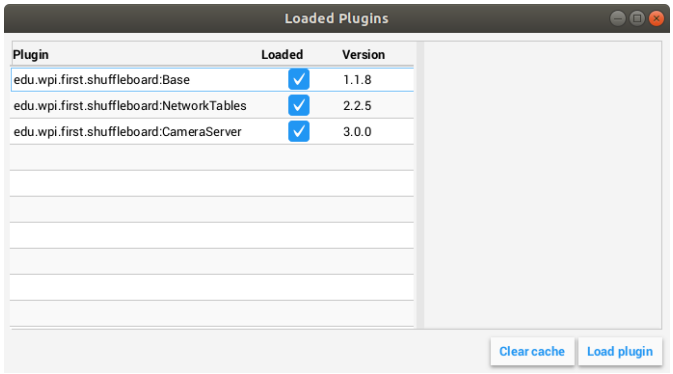
Deploying Plugin To Shuffleboard

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 by running from the shuffleboard root `gradlew :example-plugins:PLUGIN-NAME:installPlugin`

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.

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. The jar file is located in `example-plugins\PLUGIN-NAME\build\libs` after running `gradlew build` in the shuffleboard root 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.

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 -> {
         return new MyPoint2D((double) map.getOrDefault("x", 0.0), (double) map.
↪getOrDefault("y", 0.0));
      };
   }
   @Override
   public MyPoint2D getDefaultValue() {
      // use default values of 0 for X and Y coordinates
      return new MyPoint2D(0, 0);
```

(continues on next page)

(continued from previous page)

```
    }
}
```

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() {
      return List.of(PointDataType.Instance);
   }
}
```

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 (Panes, 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;
```

(continues on next page)

(continued from previous page)

```
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
   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));
}
```

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%);
    -swatch-400: hsb(0, 80%, 68%);
    -swatch-500: hsb(0, 80%, 58%);
}
```

Replacing red with blue (dark)

```
@import "/edu/wpi/first/shuffleboard/app/dark.css"

.root {
    -swatch-100: #BBDEFB;
    -swatch-200: #90CAF9;
    -swatch-300: #64BEF6;
    -swatch-400: #42A5F5;
    -swatch-500: #2196F3;
}
```

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 `getProperties()`, `getSources()`, and `titleProperty()`
`SingleTypeWidget<T>`	Adds properties for widgets that only support a single data type
`AnnotatedWidget`	Adds default implementations for `getName()` and `getDataTypes()` for widgets with a `@Description` annotation
`SingleSourceWidget`	For widgets with only a single source (by default, widgets support multiple sources)
`SimpleAnnotatedWidget<T>`	Combines `SingleTypeWidget<T>`, `AnnotatedWidget`, and `SingleSourceWidget`

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 easy 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 `@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

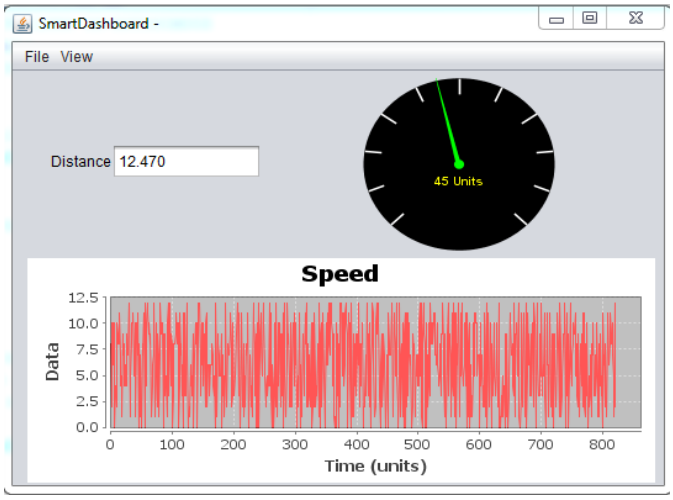
This allows widgets to have their name and supported data types defined by a single annotation, when used alongside *AnnotatedWidget*.

11.3 SmartDashboard

SmartDashboard is a simple and efficient dashboard that uses relatively few computer resources. It does not have the fancy look or some of the features Shuffleboard has, but it displays network tables data with a variety of widgets without bogging down the driver station computer.

> **Warning**
>
> SmartDashboard is deprecated and will be removed for 2027 due to its usage of Network Tables v3. Users can find *additional modern dashboard options here*

11.3.1 SmartDashboard Introduction



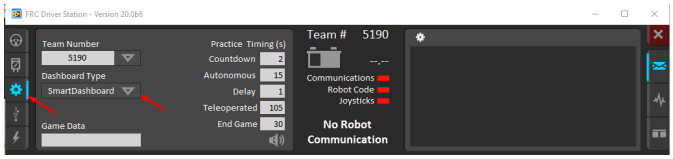
The SmartDashboard is a Java program that will display robot data in real time. The SmartDashboard helps you with these things:

- 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.

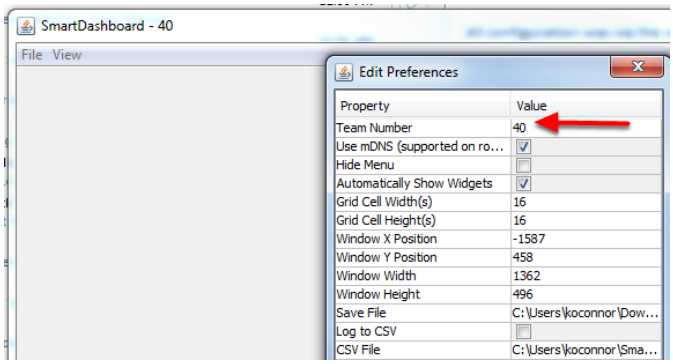
> **Warning**
>
> SmartDashboard is deprecated and will be removed for 2027 due to its usage of Network Tables v3. Users can find *additional modern dashboard options here*

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.

Configuring the Team Number



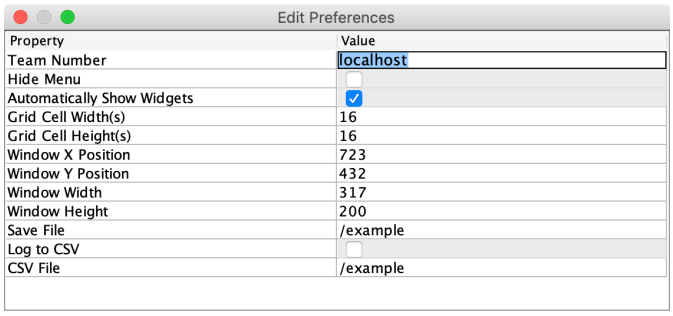
The first time you launch the SmartDashboard you should be prompted for your team number. To change the team number after this: click **File > Preferences** to open the Preferences dialog. Double-click the box to the right of **Team Number** and enter your FRC® Team Number, then click outside the box to save.

> **Note**
>
> SmartDashboard will take a moment to configure itself for the team number, do not be alarmed.

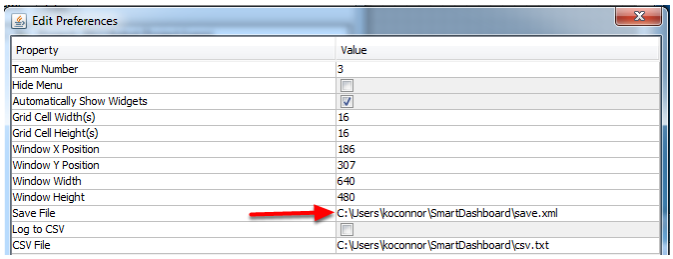
Setting a Custom NetworkTables Server Location

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!

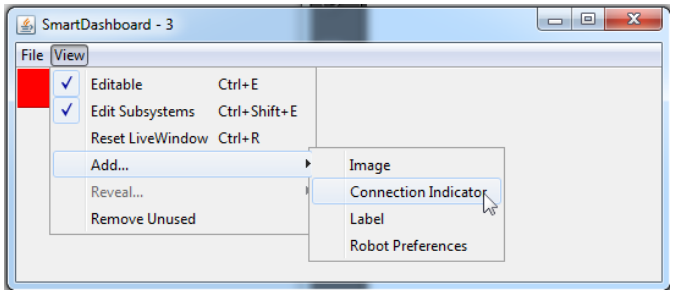


Locating the Save File



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.

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.

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*.

11.3.2 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.

Writing Values to SmartDashboard

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());
}
```

C++

```
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());
}
```

PYTHON

```
from wpilib import SmartDashboard
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 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.

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*.

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.

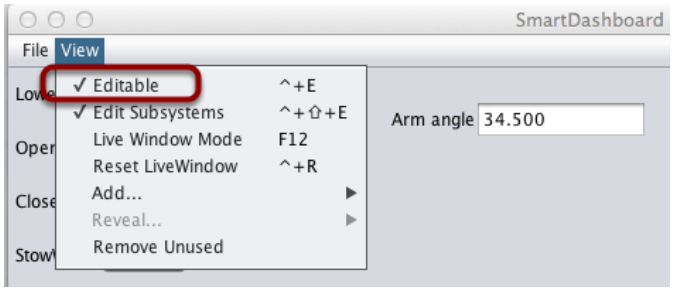
To ensure that the SmartDashboard and OutlineViewer are showing current values, it is necessary to restart the NetworkTables clients and robot at the same time. That way, old values that one is holding won't get replicated to the others.

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.

11.3.3 Changing the display properties of a value

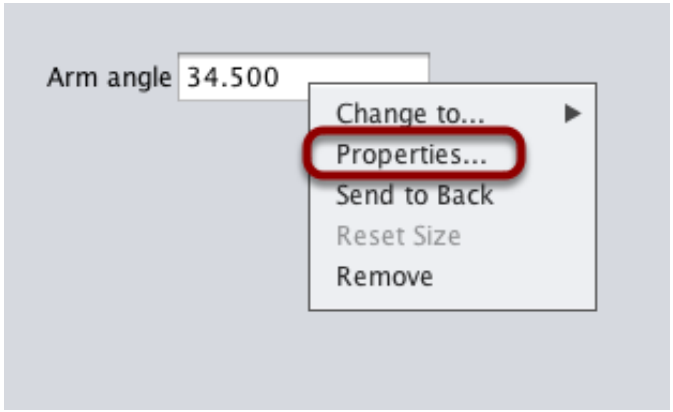
Each value displayed with SmartDashboard has a set of properties that effect the way it's displayed.

Setting the SmartDashboard display into editing mode



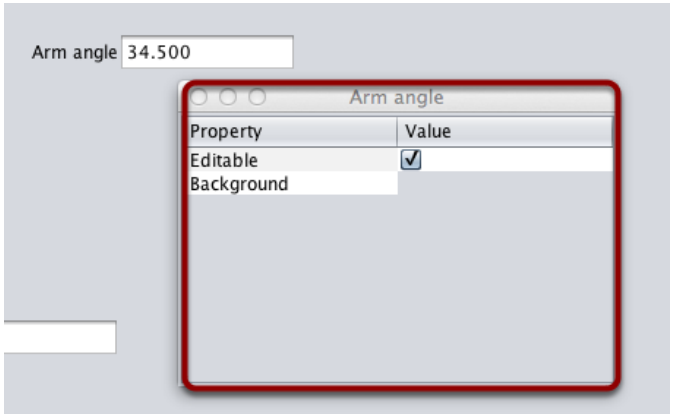
The SmartDashboard has two modes it can operate in, display mode and edit mode. In edit mode you can move around widgets on the screen and edit their properties. To put the SmartDashboard into edit mode, click the "View" menu, then select "Editable" to turn on edit mode.

Getting the properties editor of a widget



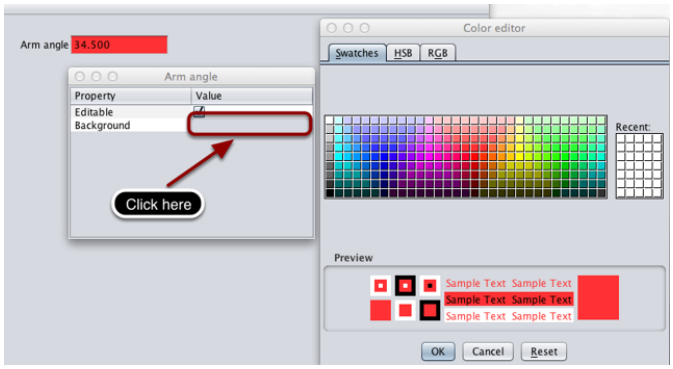
Once in edit mode, you can display and edit the properties for a widget. Right-click on the widget and select "Properties...".

Editing the properties on a field



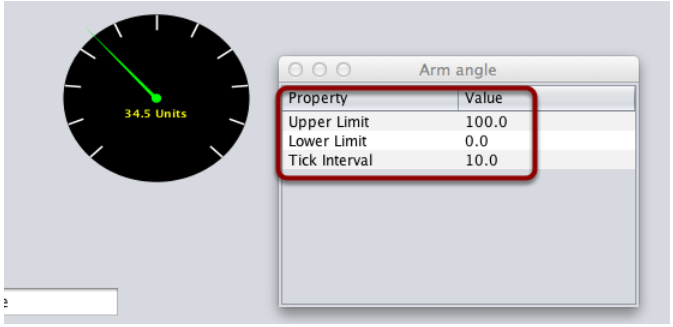
A dialog box will be shown in response to the “Properties…” menu item on the widgets right-click context menu.

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.

Edit properties of other widgets



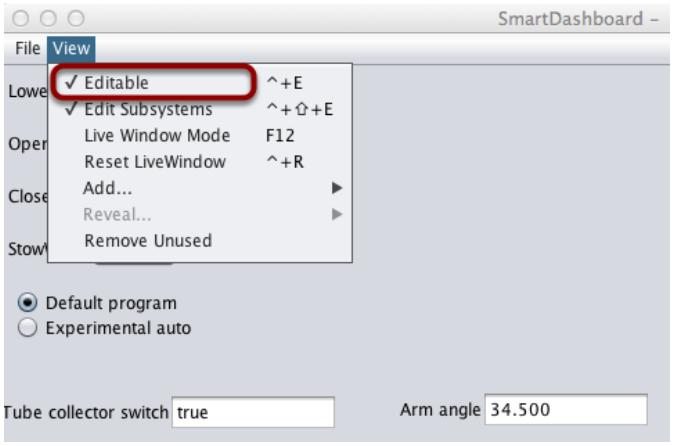
Different widget types have different sets of editable properties to change the appearance. In this example, the upper and lower limits of the dial and the tick interval are changeable

parameters.

11.3.4 Changing the Display Widget Type for a Value

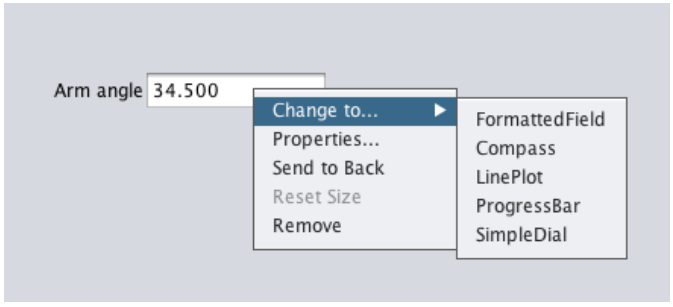
One can change the type of widget that displays values with the SmartDashboard. The allowable widgets depend on the type of the value being displayed.

Setting Edit Mode



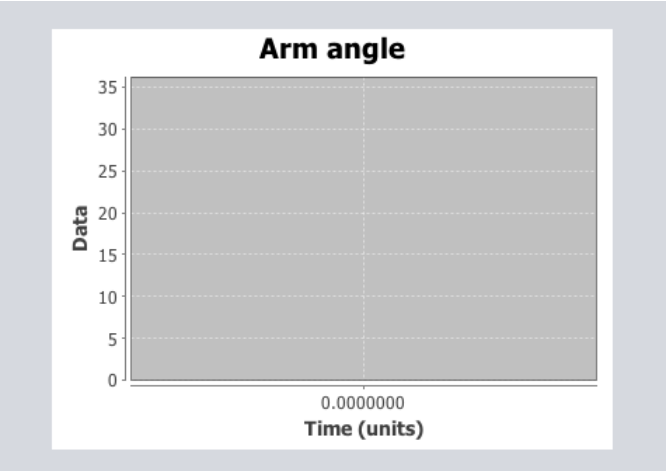
Make sure that the SmartDashboard is in edit mode. This is done by selecting `Editable` from the `View menu`.

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`.

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*.

11.3.5 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, with examples for both TimedRobot and Command-Based Robots.

TimedRobot

> **Note**
>
> The code snippets shown below are part of the TimedRobot template (Java, C++):

Creating SendableChooser Object

In `Robot.java` / `Robot.h`, create a variable to hold a reference to a `SendableChooser` object. Two or more auto modes can be added by creating strings to send to the chooser. Using the `SendableChooser`, one can choose between them. In this example, `Default` and `My Auto` are shown as options. You will also need a variable to store which auto has been chosen, `m_autoSelected`.

Java

```
  private static final String kDefaultAuto = "Default";
  private static final String kCustomAuto = "My Auto";
  private String m_autoSelected;
  private final SendableChooser<String> m_chooser = new SendableChooser<>();
```

C++

```
  frc::SendableChooser<std::string> m_chooser;
  const std::string kAutoNameDefault = "Default";
  const std::string kAutoNameCustom = "My Auto";
  std::string m_autoSelected;
```

Python

```
import wpilib
self.defaultAuto = "Default"
self.customAuto = "My Auto";
self.chooser = wpilib.SendableChooser()
```

Setting Up Options

The chooser allows you to pick from a list of defined elements, in this case the strings we defined above. In the `Robot` constructor, add your options created as strings above using `setDefaultOption` or `addOption`. `setDefaultOption` will be the one selected by default when the dashboard starts. The `putData` function will push it to the dashboard on your driver station computer.

Java

```
  public Robot() {
    m_chooser.setDefaultOption("Default Auto", kDefaultAuto);
    m_chooser.addOption("My Auto", kCustomAuto);
    SmartDashboard.putData("Auto choices", m_chooser);
  }
```

C++

```
Robot::Robot() {
  m_chooser.SetDefaultOption(kAutoNameDefault, kAutoNameDefault);
  m_chooser.AddOption(kAutoNameCustom, kAutoNameCustom);
  frc::SmartDashboard::PutData("Auto Modes", &m_chooser);
}
```

Python

```
from wpilib import SmartDashboard
self.chooser.setDefaultOption("Default Auto", self.defaultAuto)
self.chooser.addOption("My Auto", self.customAuto)
SmartDashboard.putData("Auto choices", self.chooser)
```

Running Autonomous Code

Now, in `autonomousInit` and `autonomousPeriodic`, you can use the `m_autoSelected` variable to read which option was chosen, and change what happens during the autonomous period.

Java

```
  @Override
  public void autonomousInit() {
    m_autoSelected = m_chooser.getSelected();
    System.out.println("Auto selected: " + m_autoSelected);
  }

  /** This function is called periodically during autonomous. */
  @Override
  public void autonomousPeriodic() {
    switch (m_autoSelected) {
      case kCustomAuto:
        // Put custom auto code here
        break;
      case kDefaultAuto:
      default:
        // Put default auto code here
        break;
    }
  }
```

C++

```
void Robot::AutonomousInit() {
  m_autoSelected = m_chooser.GetSelected();
  wpi::print("Auto selected: {}\n", m_autoSelected);

  if (m_autoSelected == kAutoNameCustom) {
    // Custom Auto goes here
  } else {
    // Default Auto goes here
  }
}

void Robot::AutonomousPeriodic() {
  if (m_autoSelected == kAutoNameCustom) {
    // Custom Auto goes here
  } else {
```

(continues on next page)

(continued from previous page)

```
    // Default Auto goes here
  }
}
```

Python

```
def autonomousInit(self):
   self.autoSelected = self.chooser.getSelected()
   print("Auto selected: " + self.autoSelected)
def autonomousPeriodic(self):
   match self.autoSelected:
      case self.customAuto:
         # Put custom auto code here
      case _:
         # Put default auto code here
```

Command-Based

Note

The code snippets shown below are part of the HatchbotTraditional example project (Java, C++, Python):

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

```
  // 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<>();
```

C++ (using raw pointers)

```
  // 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;
```

C++ (using `CommandPtr`)

```
  // The autonomous routines
  frc2::CommandPtr m_simpleAuto = autos::SimpleAuto(&m_drive);
  frc2::CommandPtr m_complexAuto = autos::ComplexAuto(&m_drive, &m_hatch);

  // The chooser for the autonomous routines
  frc::SendableChooser<frc2::Command*> m_chooser;
```

Python

```
        # A simple auto routine that drives forward a specified distance, and then␣
↪stops.
        self.simpleAuto = DriveDistance(
            constants.kAutoDriveDistanceInches, constants.kAutoDriveSpeed, self.drive
        )

        # A complex auto routine that drives forward, drops a hatch, and then drives␣
↪backward.
        self.complexAuto = ComplexAuto(self.drive, self.hatch)

        # Chooser
        self.chooser = wpilib.SendableChooser()
```

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

```
    // Add commands to the autonomous command chooser
    m_chooser.setDefaultOption("Simple Auto", m_simpleAuto);
    m_chooser.addOption("Complex Auto", m_complexAuto);
```

C++ (using raw pointers)

```
  // Add commands to the autonomous command chooser
  m_chooser.SetDefaultOption("Simple Auto", &m_simpleAuto);
  m_chooser.AddOption("Complex Auto", &m_complexAuto);
```

C++ (using `CommandPtr`)

```
  // Add commands to the autonomous command chooser
  // Note that we do *not* move ownership into the chooser
  m_chooser.SetDefaultOption("Simple Auto", m_simpleAuto.get());
  m_chooser.AddOption("Complex Auto", m_complexAuto.get());
```

Python

```
        # Add commands to the autonomous command chooser
        self.chooser.setDefaultOption("Simple Auto", self.simpleAuto)
        self.chooser.addOption("Complex Auto", self.complexAuto)
```

Then, publish the chooser to the dashboard:

Java

```
// Put the chooser on the dashboard
SmartDashboard.putData(m_chooser);
```

C++

```
// Put the chooser on the dashboard
frc::SmartDashboard::PutData(&m_chooser);
```

Python

```
from wpilib import SmartDashboard
# Put the chooser on the dashboard
SmartDashboard.putData(chooser)
```

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

```
  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();
    }
  }
```

C++ (Source)

```
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();
  }
}
```

Python

```
    def getAutonomousCommand(self) -> commands2.Command:
        return self.chooser.getSelected()
```

```
    def autonomousInit(self) -> None:
        """This autonomous runs the autonomous command selected by your␣
↪RobotContainer class."""
        self.autonomousCommand = self.container.getAutonomousCommand()

        if self.autonomousCommand:
            self.autonomousCommand.schedule()
```

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. In Python the scheduler runs automatically when `TimedCommandRobot` is used.

> **Note**
>
> Running the scheduler can occur in the `autonomousPeriodic()` function or `robotPeriodic()`, both will function similarly in autonomous mode.

Java

```
49  @Override
50  public void robotPeriodic() {
51    CommandScheduler.getInstance().run();
52  }
```

C++ (Source)

```
29 void Robot::RobotPeriodic() {
30   frc2::CommandScheduler::GetInstance().Run();
31 }
```

Canceling the Autonomous Command

In `Robot.java`, when the teleop period begins, the autonomous command will be canceled.

Java

```
87   @Override
88   public void teleopInit() {
89     // This makes sure that the autonomous stops running when
90     // teleop starts running. If you want the autonomous to
91     // continue until interrupted by another command, remove
92     // this line or comment it out.
93     if (m_autonomousCommand != null) {
94       m_autonomousCommand.cancel();
95     }
96   }
```

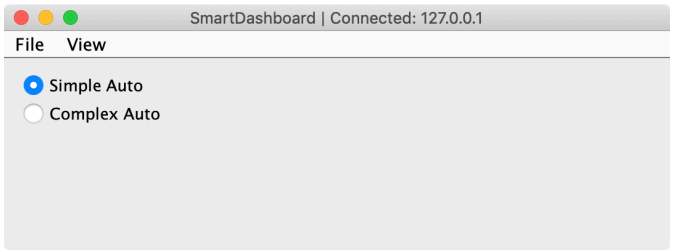
C++ (Source)

```
56 void Robot::TeleopInit() {
57   // This makes sure that the autonomous stops running when
58   // teleop starts running. If you want the autonomous to
59   // continue until interrupted by another command, remove
60   // this line or comment it out.
61   if (m_autonomousCommand != nullptr) {
62     m_autonomousCommand->Cancel();
63     m_autonomousCommand = nullptr;
64   }
65 }
```

Python

```
51     def teleopInit(self) -> None:
52         # This makes sure that the autonomous stops running when
53         # teleop starts running. If you want the autonomous to
54         # continue until interrupted by another command, remove
55         # this line or comment it out.
56         if self.autonomousCommand:
57             self.autonomousCommand.cancel()
```

SmartDashboard Display

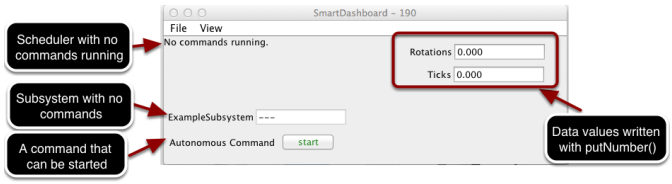


When the SmartDashboard is run, the choices from the `SendableChooser` are automatically displayed. You can simply pick an option before the autonomous period begins and the corresponding command will run.

11.3.6 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.

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.

Displaying the Scheduler Status

JAVA

```
SmartDashboard.putData(CommandScheduler.getInstance());
```

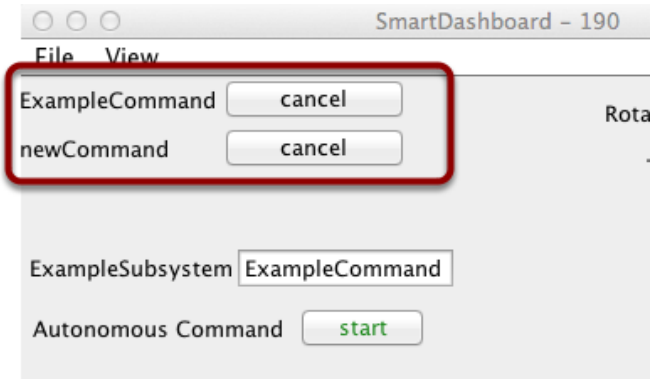
C++

```
frc::SmartDashboard::PutData(frc2::CommandScheduler::GetInstance());
```

PYTHON

```
from wpilib import SmartDashboard
from commands2 import CommandScheduler
SmartDashboard.putData(CommandScheduler.getInstance())
```

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 `Robot` constructor in your Robot program 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.

Displaying Subsystem Status

JAVA

```
SmartDashboard.putData(exampleSubsystem);
```

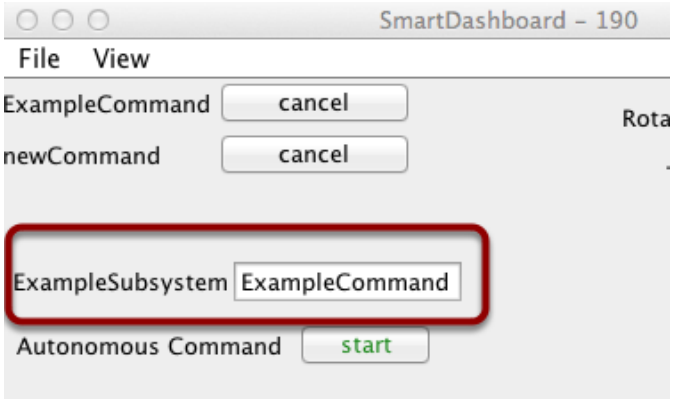
C++

```
frc::SmartDashboard::PutData(&exampleSubsystem);
```

PYTHON

```
from wpilib import SmartDashboard
SmartDashboard.putData(exampleSubsystem)
```

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`.

Activating Commands with a Button

JAVA

```
SmartDashboard.putData("Autonomous Command", exampleCommand);
```

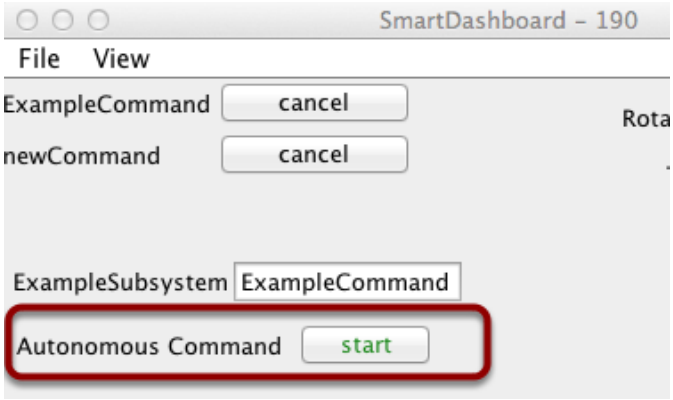
C++

```
frc::SmartDashboard::PutData("Autonomous Command", &exampleCommand);
```

PYTHON

```
from wpilib import SmartDashboard
SmartDashboard.putData("Autonomous Command", exampleCommand)
```

This is the code required to create a button for the command on SmartDashboard. 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.

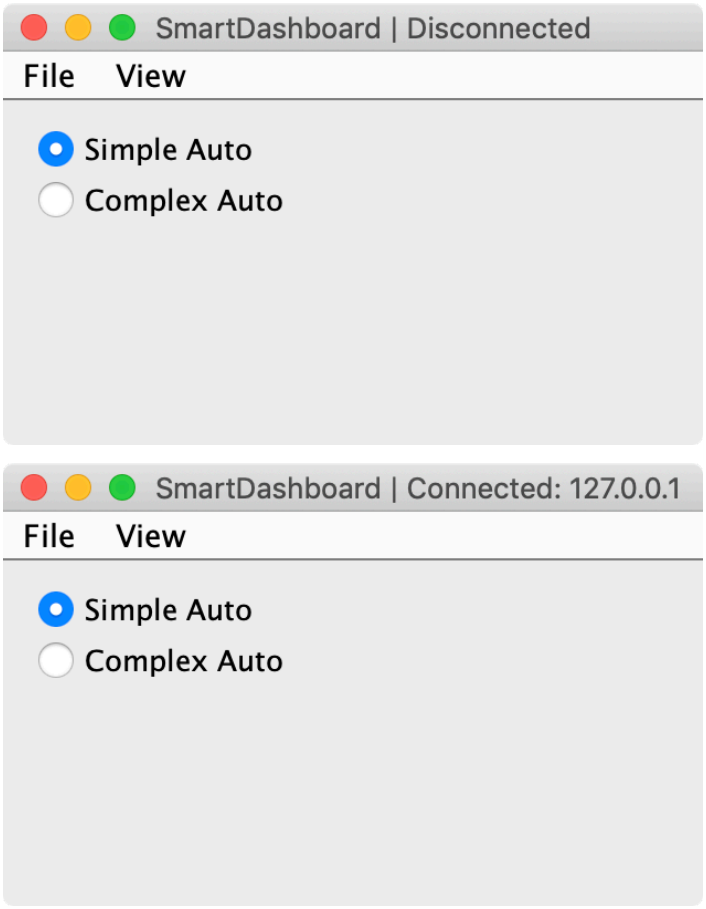


In this example you can see a button labeled `Autonomous Command`. Pressing this button will run the associated command and is an excellent way of testing commands one at a time without having to add throw-away test code to your robot program. Adding buttons for each command makes it simple to test the program, one command at a time.

11.3.7 Verifying SmartDashboard is working

Connection Indicator

SmartDashboard will automatically include the connection status and IP address of the NetworkTables source in the title of the window.



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.

Robot Program Example

JAVA

```
public class Robot extends TimedRobot {
   double counter = 0.0;
   public void teleopPeriodic() {
      SmartDashboard.putNumber("Counter", counter++);
   }
}
```

C++

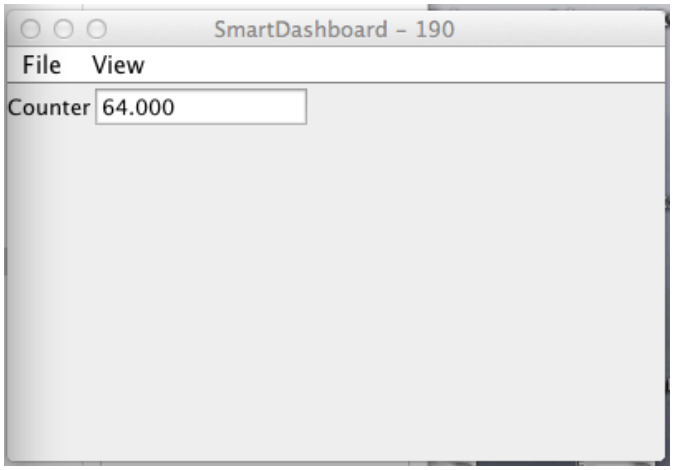
```
#include "Robot.h"
float counter = 0.0;
void Robot::TeleopPeriodic() {
   frc::SmartDashboard::PutNumber("Counter", counter++);
}
```

PYTHON

```
from wpilib import SmartDashboard
self.counter = 0.0
def teleopPeriodic(self) -> None:
   SmartDashboard.putNumber("Counter", self.counter += 1)
```

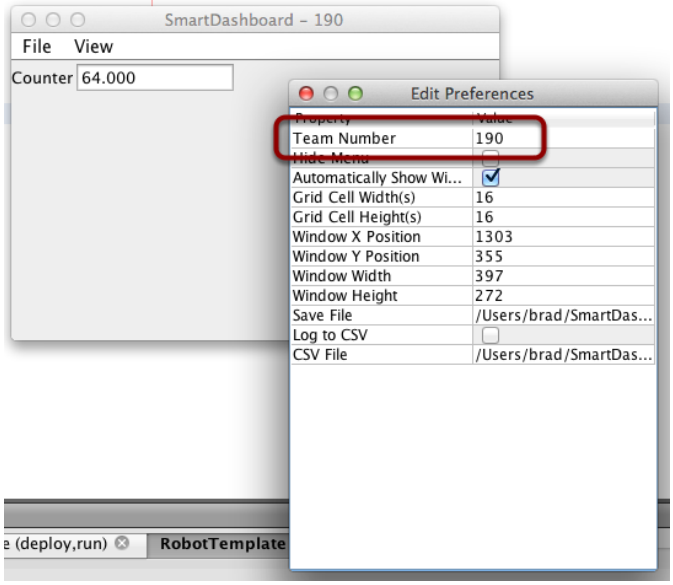
This is a minimal robot program that writes a value to the SmartDashboard. It simply increments a counter 50 times per second to verify that the connection is working. However, to minimize bandwidth usage, NetworkTables by default will throttle the updates to 10 times per second.

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.

Verifying the IP address in SmartDashboard

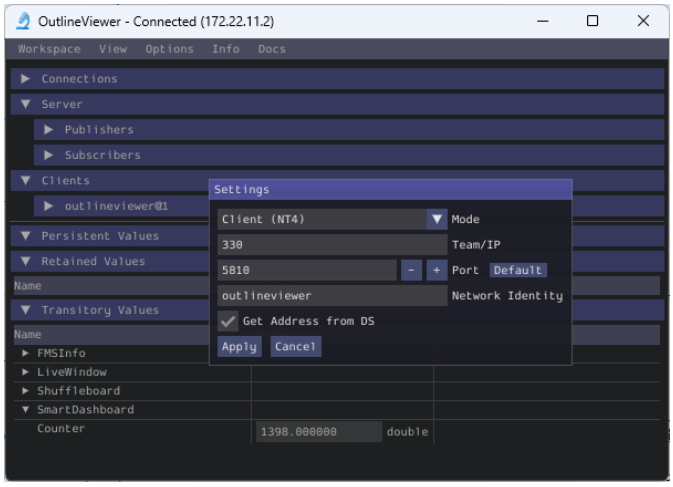


If the display of the value is not appearing, verify that the team number is correctly set as shown in this picture. The preferences dialog can be viewed by selecting `File`, then `Preferences`.

Verifying Program using OutlineViewer

You can verify that the robot program is generating SmartDashboard values by using the *OutlineViewer program*.

Expand the SmartDashboard row. The value `Counter` is the variable written to the SmartDashboard via NetworkTables. As the program runs you should see the value increasing (`1398.0` 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.



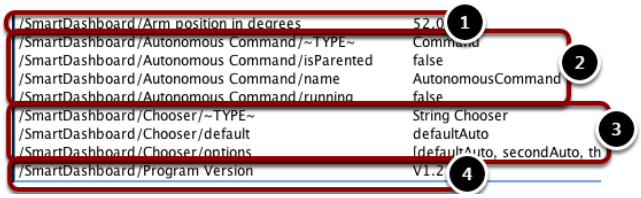
11.3.8 SmartDashboard Namespace

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.

SmartDashboard Data Values

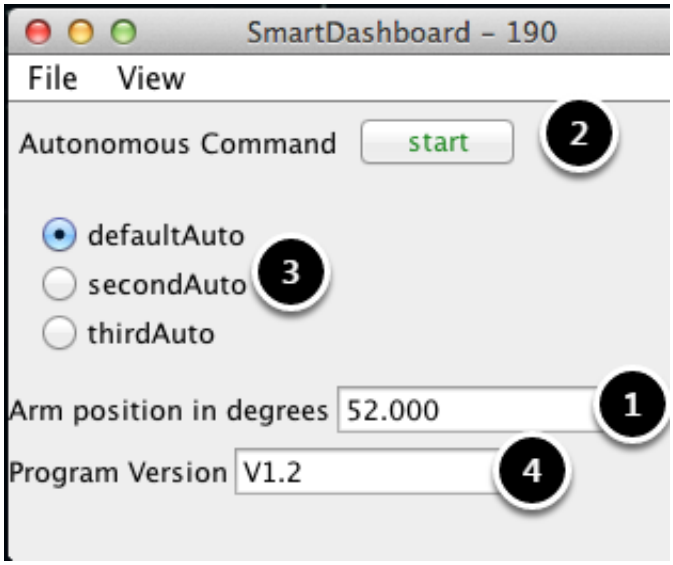


SmartDashboard values are created with key names that begin with `SmartDashboard/`. The above values viewed with OutlineViewer correspond to data put to the SmartDashboard with the following statements:

```
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");
```

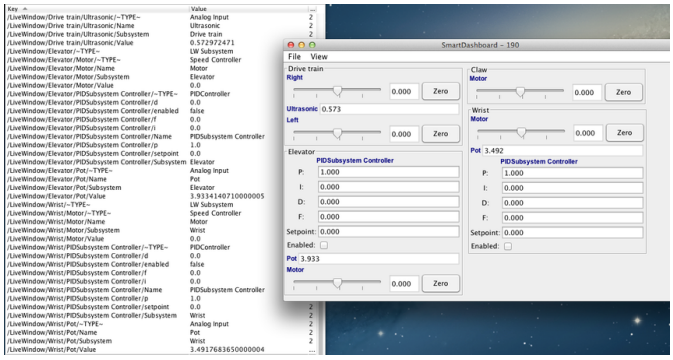
The `Arm position` is created with the `putNumber()` call. The `AutonomousCommand` is written with a `putData("Autonomous Command", command)` that is not shown in the above code fragment. The chooser is created as a `SendableChooser` object and the string value, `Program Version` is created with the `putString()` call.

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.

LiveWindow Data Values



LiveWindow data is automatically grouped by subsystem. The data is viewable in the SmartDashboard when the robot is in Test mode (set on the Driver Station). If you are not writing a command based program, you can still cause sensors and actuators to be grouped for easy viewing by specifying the subsystem name. In the above display you can see the key names and the resultant output in Test mode on the SmartDashboard. All the strings start with `/LiveWindow` then the Subsystem name, then a group of values that are used to display each element. The code that generates this LiveWindow display is shown below:

```
drivetrainLeft = new PWMVictorSPX(1);
drivetrainLeft.setName("Drive train", "Left");
drivetrainRight = new PWMVictorSPX(1);
drivetrainRight.setName("Drive train", "Right";
drivetrainRobotDrive = new DifferentialDrive(drivetrainLeft, drivetrainRight);
drivetrainRobotDrive.setSafetyEnabled(false);
drivetrainRobotDrive.setExpiration(0.1);
drivetrainUltrasonic = new AnalogInput(3);
drivetrainUltrasonic.setName("Drive train", "Ultrasonic");
elevatorMotor = new PWMVictorSPX(6);
elevatorMotor.setName("Elevator", "Motor");
elevatorPot = new AnalogInput(4);
elevatorPot.setName("Elevator", "Pot");
wristPot = new AnalogInput(2);
wristPot.setName("Wrist", "Pot");
wristMotor = new PWMVictorSPX(3);
wristMotor.setName("Wrist", "Motor");
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.

11.3.9 SmartDashboard: Test Mode and Live Window

Displaying LiveWindow Values

LiveWindow will automatically add your sensors and actuators 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. This allows you to customize the names and group them 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

For each sensor or actuator that is created, set the subsystem name and display name by calling `setName` (`SetName` in C++). When the SmartDashboard is put into LiveWindow mode, it will display the sensors and actuators.

JAVA

```
Ultrasonic ultrasonic = new Ultrasonic(1, 2);
SendableRegistry.setName(ultrasonic, "Arm", "Ultrasonic");
Spark elbow = new Spark(1);
SendableRegistry.setName(elbow, "Arm", "Elbow");
VictorSP wrist = new VictorSP(2);
SendableRegistry.setName(wrist, "Arm", "Wrist");
```

C++

```
frc::Ultrasonic ultrasonic{1, 2};
SendableRegistry::SetName(ultrasonic, "Arm", "Ultrasonic");
frc::Spark elbow{1};
SendableRegistry::SetName(elbow, "Arm", "Elbow");
frc::VictorSP wrist{2};
SendableRegistry::SetName(wrist, "Arm", "Wrist");
```

PYTHON

```
from wpilib import Spark, Ultrasonic, VictorSP
from wpiutil import SendableRegistry
    ultrasonic = Ultrasonic(1, 2)
SendableRegistry.setName(ultrasonic, "Arm", "Ultrasonic")
elbow = Spark(1)
SendableRegistry.setName(elbow, "Arm", "Elbow")
wrist = VictorSP(2)
SendableRegistry.setName(wrist, "Arm", "Wrist")
```

If your objects are in a `Subsystem`, this can be simplified using the addChild method of `SubsystemBase`

JAVA

```
Ultrasonic ultrasonic = new Ultrasonic(1, 2);
addChild("Ultrasonic", ultrasonic);
Spark elbow = new Spark(1);
addChild("Elbow", elbow);
VictorSP wrist = new VictorSP(2);
addChild("Wrist", wrist);
```

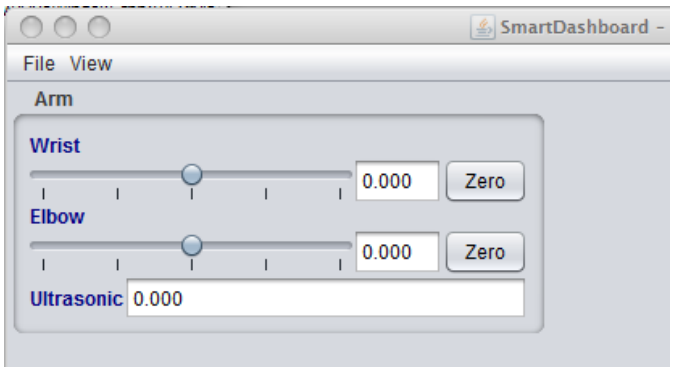
C++

```
frc::Ultrasonic ultrasonic{1, 2};
AddChild("Ultrasonic", ultrasonic);
frc::Spark elbow{1};
AddChild("Elbow", elbow);
frc::VictorSP wrist{2};
AddChild("Wrist", wrist);
```

PYTHON

```
from wpilib import Spark, Ultrasonic, VictorSP
from commands2 import SubsystemBase
    ultrasonic = Ultrasonic(1, 2)
SubsystemBase.addChild("Ultrasonic", ultrasonic)
elbow = Spark(1)
SubsystemBase.addChild("Elbow", elbow)
wrist = VictorSP(2)
SubsystemBase.addChild("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.

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 (see *Enabling Test Mode* for details on how to do this). 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. When the robot is enabled in Test mode, the SmartDashboard display will switch to test mode (LiveWindow) and will display the status of any actuators and sensors used by your program.

Important

Since 2024, LiveWindow is not enabled by default in Test mode!

Enabling LiveWindow in Test Mode

To run LiveWindow in Test Mode, the following code is needed in the `Robot` class:

JAVA

```
public Robot() {
   enableLiveWindowInTest(true);
}
```

C++

```
Robot::Robot() {
   EnableLiveWindowInTest(true);
}
```

PYTHON

```
def robotInit(self) -> None:
   enableLiveWindowInTest(true)
```

Explicitly vs. implicit test mode display

JAVA

```
PWMSparkMax leftDrive;
PWMSparkMax rightDrive;
PWMVictorSPX arm;
BuiltInAccelerometer accel;

public Robot() {
   leftDrive = new PWMSparkMax(0);
   rightDrive = new PWMSparkMax(1);
   arm = new PWMVictorSPX(2);
   accel = new BuiltInAccelerometer();
   SendableRegistry.setName(arm, "SomeSubsystem", "Arm");
   SendableRegistry.setName(accel, "SomeSubsystem", "Accelerometer");
}
```

C++

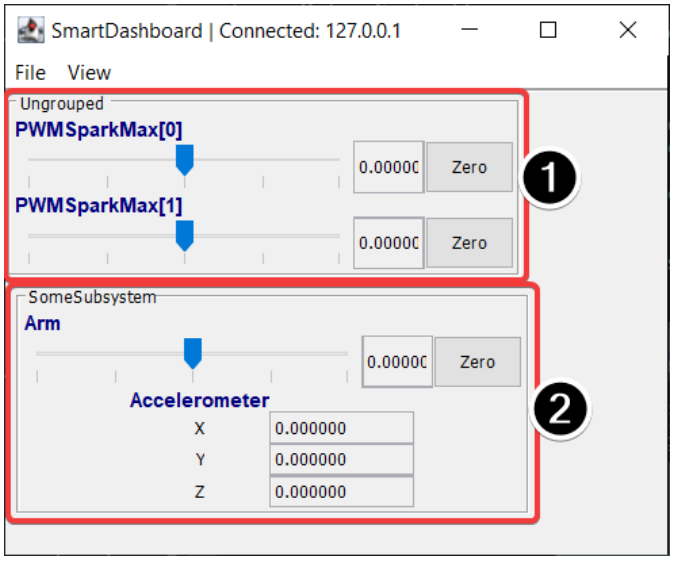
```
frc::PWMSparkMax leftDrive{0};
frc::PWMSparkMax rigthDrive{1};
frc::BuiltInAccelerometer accel{};
frc::PWMVictorSPX arm{3};
Robot::Robot() {
   wpi::SendableRegistry::SetName(&arm, "SomeSubsystem", "Arm");
   wpi::SendableRegistry::SetName(&accel, "SomeSubsystem", "Accelerometer");
}
```

PYTHON

```
from wpilib import BuiltInAccelerometer, PWMSparkMax, PWMVictorSPX
from wpiutil import SendableRegistry
def robotInit(self) -> None:
   leftDrive = PWMSparkMax(0)
   rightDrive = PWMSparkMax(1)
   arm = PWMVictorSPX(2)
   accel = BuiltInAccelerometer()
   SendableRegistry.setName(arm, "SomeSubsystem", "Arm")
   SendableRegistry.setName(accel, "SomeSubsystem", "Accelerometer")
```

All sensors and actuators will automatically be displayed on the SmartDashboard in test mode and will be named using the object type (such as PWMSparkMax, PWMVictorSPX, BuiltInAccelerometer, etc.) with 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.

Understanding what is displayed in Test mode



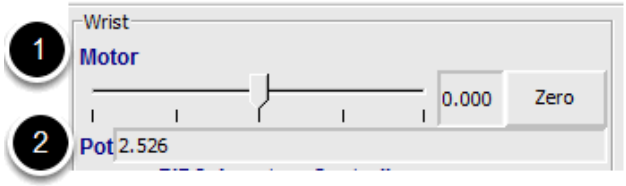
This is the output in the SmartDashboard display when the robot is placed into test mode. In the display shown above the objects listed as Ungrouped were implicitly created by WPILib

when the corresponding objects were created. These objects are contained in a subsystem group called "Ungrouped" **(1)** and are named with the device type (PWMSparkMax in this case), and the 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 `SendableRegistry.setName()`. Explicitly created sensors and actuators will be grouped by the specified subsystem.

PID Tuning with SmartDashboard

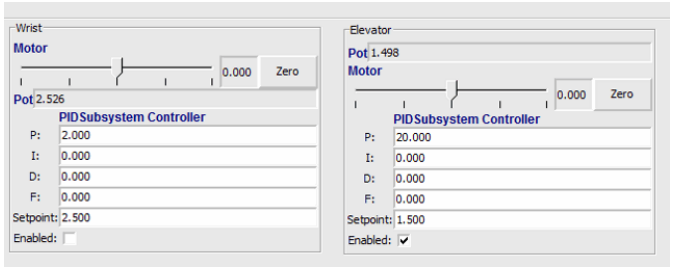
The PID (Proportional, Integral, Differential) is an algorithm for determining the motor speed based on sensor feedback to reach a setpoint as quickly as possible. For example, a robot with an elevator that moves to a predetermined position should move there as fast as possible then stop without excessive 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



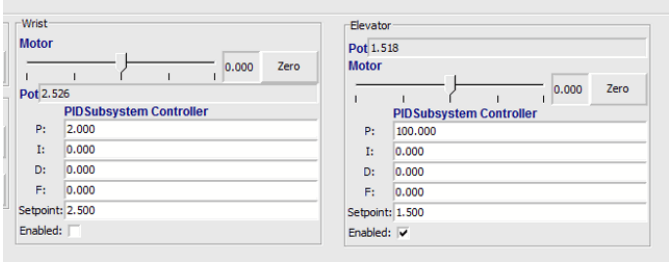
In Test mode, the PID Subsystems display their P, I, and D parameters that are set in the code. The P, I, and D values are the weights applied to the computed error (P), sum of errors over time (I), and the rate of change of errors (D). Each of those terms is multiplied by the weights

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 Flywheel Velocity 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*.

11.4 Glass

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

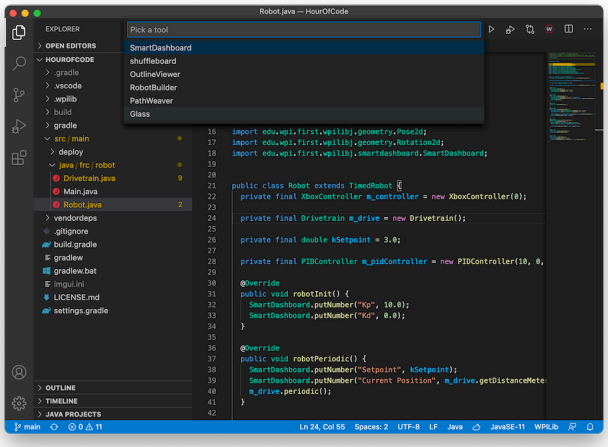
Glass will not be available within the list of dashboards in the NI Driver Station.

11.4.1 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.

Opening Glass

Glass can be launched by selecting the ellipsis menu (...) in VS Code, clicking on *Start Tool* and then choosing *Glass*.



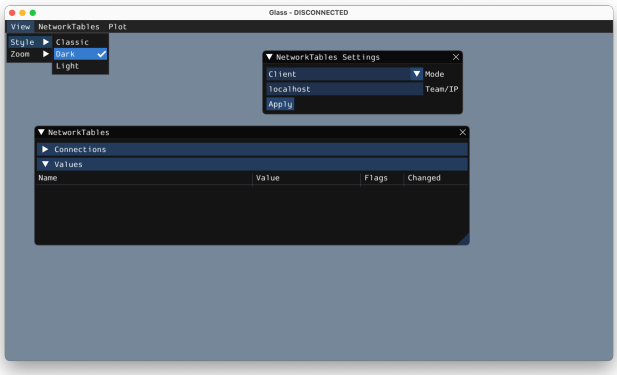
Note

You can also launch Glass directly by navigating to `~/wpilib/YYYY/tools` and running `Glass.py` (Linux and macOS) or by using the shortcut inside the WPILib Tools desktop folder (Windows).

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:



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.

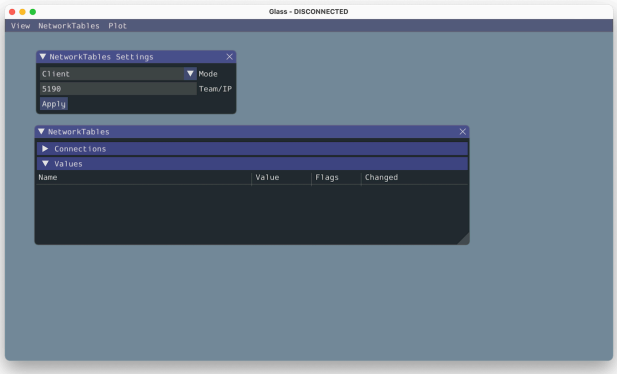
The `glass.ini` configuration file can simply be deleted to restore Glass to a "clean slate".

11.4.2 Establishing NetworkTables Connections

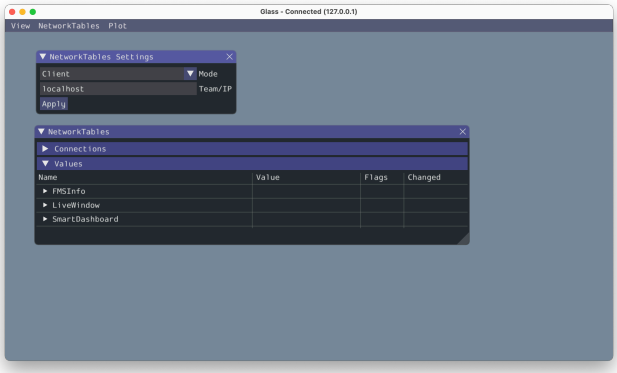
Glass uses the *NetworkTables* protocol to establish a connection with your robot program. It is also used to transmit and receive data to and from the robot.

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.

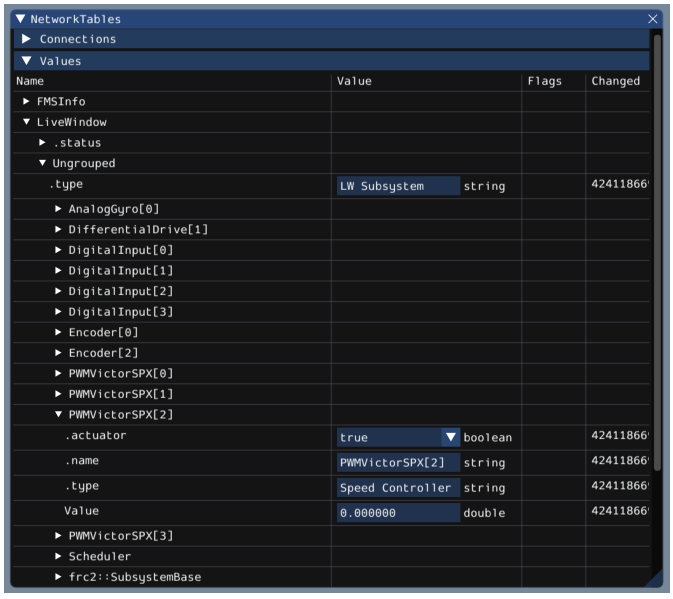


Important

The NetworkTables connection status is always visible on the title bar of the Glass application.

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.



Furthermore, you can view all connected NetworkTables clients under the *Connections* pane of the widget.

11.4.3 Glass Widgets

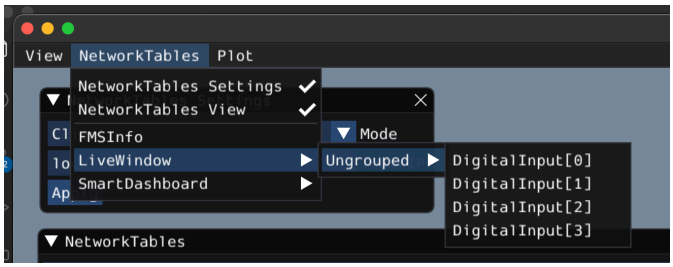
Specialized widgets are available for certain types that exist in robot code. These include objects that are manually sent over NetworkTables such as `SendableChooser` instances, or hardware that is automatically sent over *LiveWindow*.

> **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.

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:



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

```
SmartDashboard.putData("Auto Selector", m_selector);
```

C++

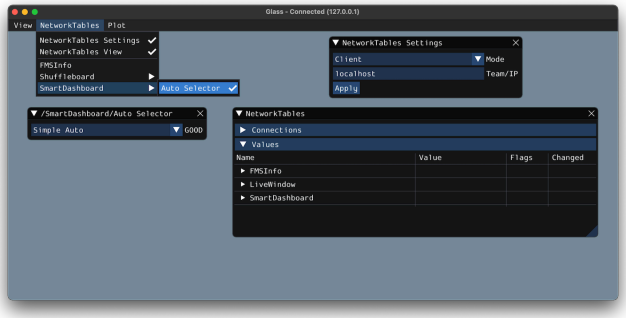
```
frc::SmartDashboard::PutData("Auto Selector", &m_selector);
```

PYTHON

```
from wpilib import SmartDashboard
SmartDashboard.putData("Auto Selector", 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*.



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

```
SmartDashboard.putData("Elevator PID Controller", m_elevatorPIDController);
```

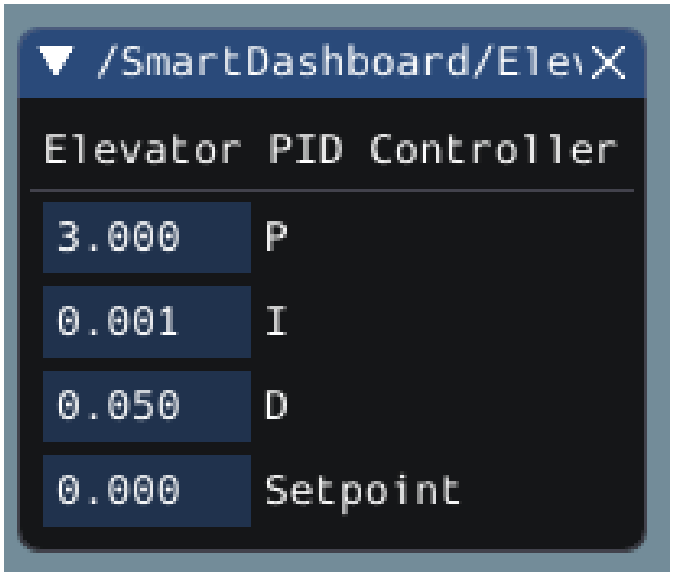
C++

```
frc::SmartDashboard::PutData("Elevator PID Controller", &m_elevatorPIDController);
```

PYTHON

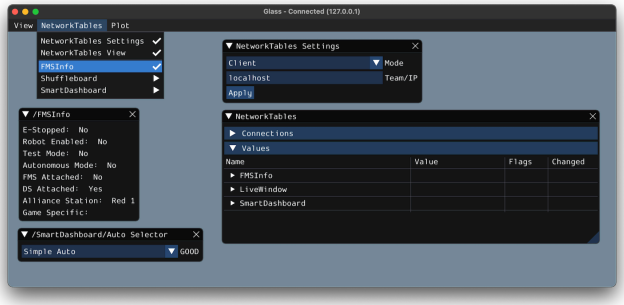
```
from wpilib import SmartDashboard
SmartDashboard.putData("Elevator PID Controller", elevatorPIDController)
```

This allows you to quickly tune P, I, and D values for various setpoints.



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*.



11.4.4 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*.

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

```
MyCommand command = new MyCommand(...);
SmartDashboard.putData("My Command", command);
```

C++

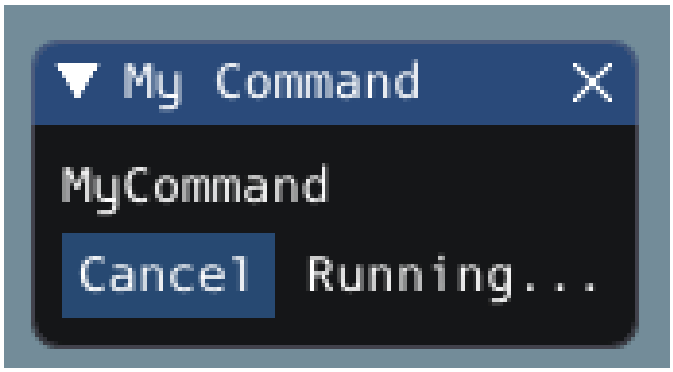
```
#include <frc/smartdashboard/SmartDashboard.h>
...
MyCommand command{...};
frc::SmartDashboard::PutData("My Command", &command);
```

PYTHON

```
from wpilib import SmartDashboard
command = MyCommand(...)
SmartDashboard.putData("My Command", command)
```

Note

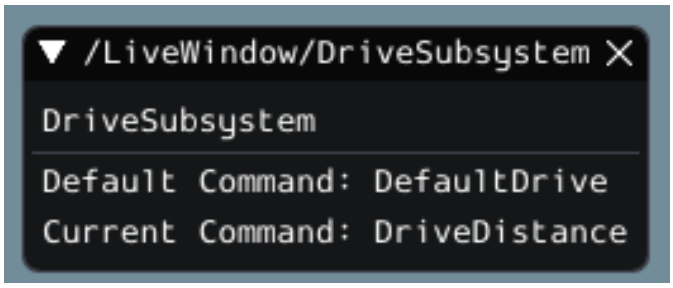
The `MyCommand` instance can also be sent using a lower-level NetworkTables API or using the *Shuffleboard API*. In this case, the `SmartDashboard` API was used, meaning that the *Command Selector* widget will appear under the `SmartDashboard` table name.



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.

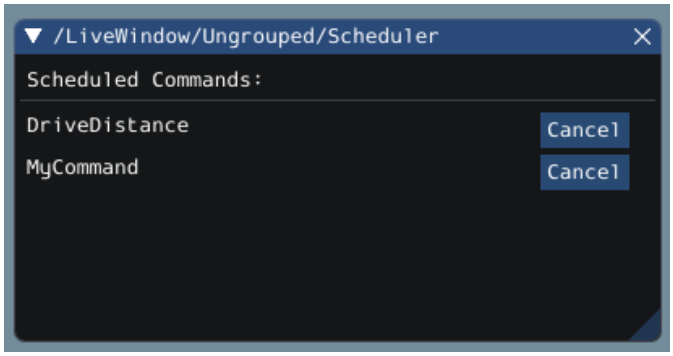
Subsystem Widget

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.



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.

11.4.5 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.

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

```java
private final Field2d m_field = new Field2d();
// Do this in either robot or subsystem init
SmartDashboard.putData("Field", m_field);
// Do this in either robot periodic or subsystem periodic
m_field.setRobotPose(m_odometry.getPoseMeters());
```

C++

```cpp
#include <frc/smartdashboard/Field2d.h>
#include <frc/smartdashboard/SmartDashboard.h>
frc::Field2d m_field;
// Do this in either robot or subsystem init
frc::SmartDashboard::PutData("Field", &m_field);
// Do this in either robot periodic or subsystem periodic
m_field.SetRobotPose(m_odometry.GetPose());
```

PYTHON

```python
from wpilib import SmartDashboard, Field2d
self.field = Field2d()
# Do this in either robot or subsystem init
SmartDashboard.putData("Field", self.field)
# Do this in either robot periodic or subsystem periodic
self.field.setRobotPose(self.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.

Sending Trajectories to Field2d

Visualizing your trajectory is a great debugging step for verifying that your trajectories are created as intended. Trajectories can be easily visualized in *Field2d* using the `setTrajectory()`/`SetTrajectory()` functions.

JAVA

```
  public void robotInit() {
    // Create the trajectory to follow in autonomous. It is best to initialize
    // trajectories here to avoid wasting time in autonomous.
    m_trajectory =
        TrajectoryGenerator.generateTrajectory(
            new Pose2d(0, 0, Rotation2d.fromDegrees(0)),
            List.of(new Translation2d(1, 1), new Translation2d(2, -1)),
            new Pose2d(3, 0, Rotation2d.fromDegrees(0)),
            new TrajectoryConfig(Units.feetToMeters(3.0), Units.feetToMeters(3.0)));

    // Create and push Field2d to SmartDashboard.
    m_field = new Field2d();
    SmartDashboard.putData(m_field);

    // Push the trajectory to Field2d.
    m_field.getObject("traj").setTrajectory(m_trajectory);
  }
```

C++

```
  void AutonomousInit() override {
    // Start the timer.
    m_timer.Start();

    // Send Field2d to SmartDashboard.
    frc::SmartDashboard::PutData(&m_field);

    // Reset the drivetrain's odometry to the starting pose of the trajectory.
    m_drive.ResetOdometry(m_trajectory.InitialPose());

    // Send our generated trajectory to Field2d.
    m_field.GetObject("traj")->SetTrajectory(m_trajectory);
  }
```

PYTHON

```
    def robotInit(self):

        # An example trajectory to follow during the autonomous period.
        self.trajectory = wpimath.trajectory.TrajectoryGenerator.generateTrajectory(
            wpimath.geometry.Pose2d(0, 0, wpimath.geometry.Rotation2d.fromDegrees(0)),
            [
                wpimath.geometry.Translation2d(1, 1),
```

(continues on next page)

(continued from previous page)

```
                wpimath.geometry.Translation2d(2, -1),
            ],
            wpimath.geometry.Pose2d(3, 0, wpimath.geometry.Rotation2d.fromDegrees(0)),
            wpimath.trajectory.TrajectoryConfig(
                wpimath.units.feetToMeters(3.0), wpimath.units.feetToMeters(3.0)
            ),
        )

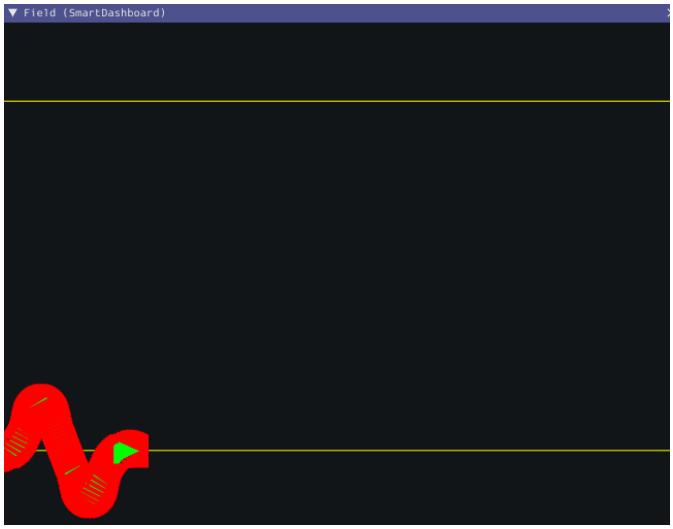
        # Create Field2d for robot and trajectory visualizations.
        self.field = wpilib.Field2d()

        # Create and push Field2d to SmartDashboard.
        wpilib.SmartDashboard.putData(self.field)

        # Push the trajectory to Field2d.
        self.field.getObject("traj").setTrajectory(self.trajectory)
```

Viewing Trajectories with Glass

The sent trajectory can be viewed with *Glass* through the dropdown *NetworkTables -> SmartDashboard -> Field2d*.

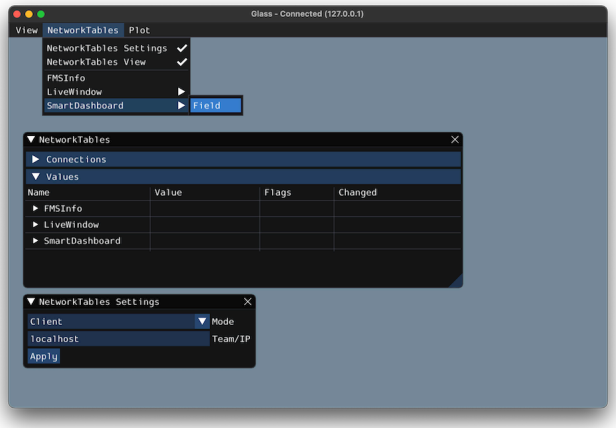


Note

The above example which uses the RamseteController (Java / C++ / Python) will not show the sent trajectory until autonomous is enabled at least once.

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.

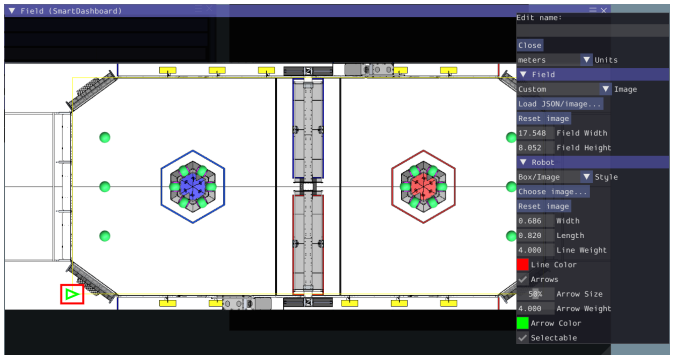


Once the widget appears, you can resize and place it on the Glass workspace as you desire. Right-clicking the top of the widget will allow you to customize the name of the widget, select a custom field image, select a custom robot image, and choose the dimensions of the field and robot.

Selecting Field Image

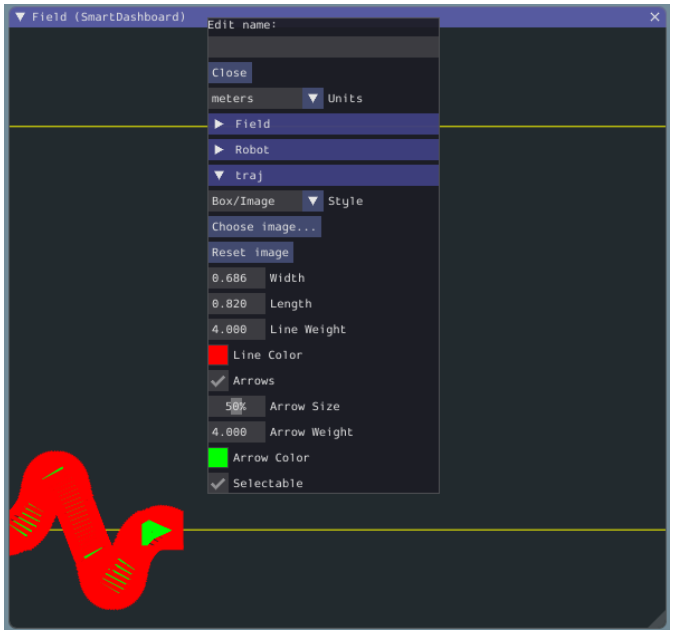
You can choose from an existing field layout using the *Image* drop-down. Or you can select a custom file by setting the *Image* to `Custom` and selecting *Choose JSON/image...*. You can choose to either select an image file or a PathWeaver JSON file as long as the image file is in the same directory. 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*.

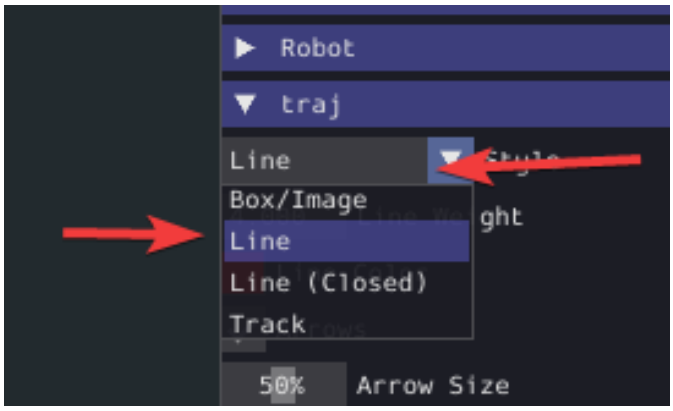


Modifying Pose Style

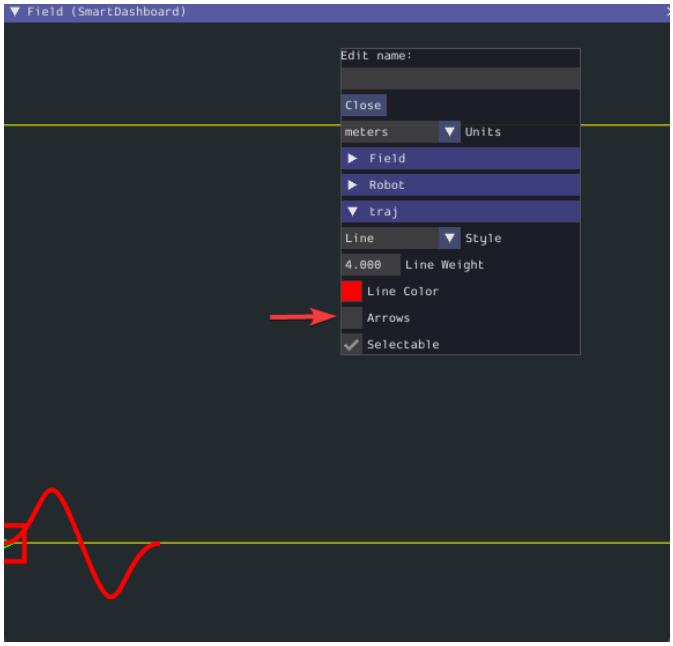
Poses can be customized in a plethora of ways by right clicking on the Field2d menu bar. Examples of customization are: line width, line weight, style, arrow width, arrow weight, color, etc.



One usage of customizing the pose style is converting the previously shown `traj` pose object to a line, rather than a list of poses. Click on the *Style* dropdown box and select *Line*. You should notice an immediate change in how the trajectory looks.

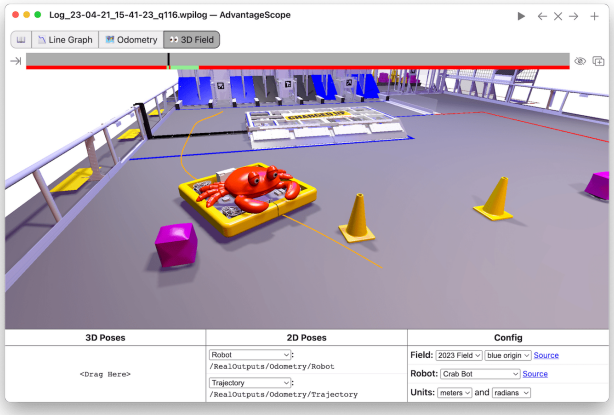


Now, uncheck the *Arrows* checkbox. This will cause our trajectory to look like a nice and fluid line!



Viewing Pose Data with AdvantageScope

AdvantageScope is an alternative option for viewing pose data from a `Field2d` object, including data recorded to a log file using *WPILib data logs*. Both 2D and 3D visualizations are supported. See the documentation for the odometry and 3D field tabs for more details.



11.4.6 The Mechanism2d Widget

Glass supports displaying stick-figure representations of your robot's mechanisms using the *Mechanism2d* widget. It supports combinations of ligaments that can rotate and / or extend or retract, such as arms and elevators and they can be combined for more complicated mechanisms. An instance of the `Mechanism2d` class should be created and populated, sent over NetworkTables, and updated periodically with the latest mechanism states in your robot code. It can also be used with the *Physics Simulation* to visualize and program your robot's mechanisms before the robot is built.

Creating and Configuring the Mechanism2d Instance

The `Mechanism2d` object is the "canvas" where the mechanism is drawn. The root node is where the mechanism is anchored to `Mechanism2d`. For a single jointed arm this would the pivot point. For an elevator, this would be where it's attached to the robot's base. To get a root node (represented by a `MechanismRoot2d` object), call `getRoot(name, x, y)` on the container `Mechanism2d` object. The name is used to name the root within NetworkTables, and should be unique, but otherwise isn't important. The `x / y` coordinate system follows the same orientation as `Field2d` - `(0,0)` is bottom left.

In the examples below, an elevator is drawn, with a rotational wrist on top of the elevator. The full Mechanism2d example is available in Java / C++

JAVA

```
43    // the main mechanism object
44    Mechanism2d mech = new Mechanism2d(3, 3);
45    // the mechanism root node
46    MechanismRoot2d root = mech.getRoot("climber", 2, 0);
```

C++

```
59  // the main mechanism object
60  frc::Mechanism2d m_mech{3, 3};
61  // the mechanism root node
62  frc::MechanismRoot2d* m_root = m_mech.GetRoot("climber", 2, 0);
```

PYTHON

```
32        # the main mechanism object
33        self.mech = wpilib.Mechanism2d(3, 3)
34        # the mechanism root node
35        self.root = self.mech.getRoot("climber", 2, 0)
```

Each `MechanismLigament2d` object represents a stage of the mechanism. It has a three required parameters, a name, an initial length to draw (relative to the size of the `Mechanism2d` object), and an initial angle to draw the ligament in degrees. Ligament angles are relative to the parent ligament, and follow math notation - the same as *Rotation2d* (counterclockwise-positive). A ligament based on the root with an angle of zero will point right. Two optional

parameters let you change the width (also relative to the size of the Mechanism2d object) and the color. Call `append()`/`Append()` on a root node or ligament node to add another node to the figure. In Java, pass a constructed `MechanismLigament2d` object to add it. In C++, pass the construction parameters in order to construct and add a ligament.

JAVA

```
    // MechanismLigament2d objects represent each "section"/"stage" of the mechanism, and are based
    // off the root node or another ligament object
    m_elevator = root.append(new MechanismLigament2d("elevator", kElevatorMinimumLength, 90));
    m_wrist =
        m_elevator.append(
            new MechanismLigament2d("wrist", 0.5, 90, 6, new Color8Bit(Color.kPurple)));
```

C++

```
  // MechanismLigament2d objects represent each "section"/"stage" of the
  // mechanism, and are based off the root node or another ligament object
  frc::MechanismLigament2d* m_elevator =
      m_root->Append<frc::MechanismLigament2d>("elevator", 1, 90_deg);
  frc::MechanismLigament2d* m_wrist =
      m_elevator->Append<frc::MechanismLigament2d>(
          "wrist", 0.5, 90_deg, 6, frc::Color8Bit{frc::Color::kPurple});
```

PYTHON

```
        # MechanismLigament2d objects represent each "section"/"stage" of the mechanism, and are based
        # off the root node or another ligament object
        self.elevator = self.root.appendLigament(
            "elevator", self.kElevatorMinimumLength, 90
        )
        self.wrist = self.elevator.appendLigament(
            "wrist", 0.5, 90, 6, wpilib.Color8Bit(wpilib.Color.kPurple)
        )
```

Then, publish the `Mechanism2d` object to NetworkTables:

JAVA

```
    // post the mechanism to the dashboard
    SmartDashboard.putData("Mech2d", mech);
```

C++

```
    // publish to dashboard
    frc::SmartDashboard::PutData("Mech2d", &m_mech);
```

PYTHON

```
        # post the mechanism to the dashboard
        wpilib.SmartDashboard.putData("Mech2d", self.mech)
```

Note

The `Mechanism2d` 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 *Mechanism2d* widget will appear under the `SmartDashboard` table name.

To manipulate a ligament angle or length, call `setLength()` or `setAngle()` on the `MechanismLigament2d` object. When manipulating ligament length based off of sensor measurements, make sure to add the minimum length to prevent 0-length (and therefore invisible) ligaments.

JAVA

```
  @Override
  public void robotPeriodic() {
    // update the dashboard mechanism's state
    m_elevator.setLength(kElevatorMinimumLength + m_elevatorEncoder.getDistance());
    m_wrist.setAngle(m_wristPot.get());
  }
```

C++

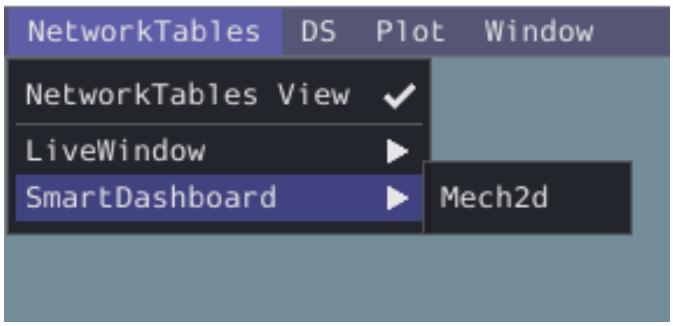
```
  void RobotPeriodic() override {
    // update the dashboard mechanism's state
    m_elevator->SetLength(kElevatorMinimumLength +
                          m_elevatorEncoder.GetDistance());
    m_wrist->SetAngle(units::degree_t{m_wristPotentiometer.Get()});
  }
```

PYTHON

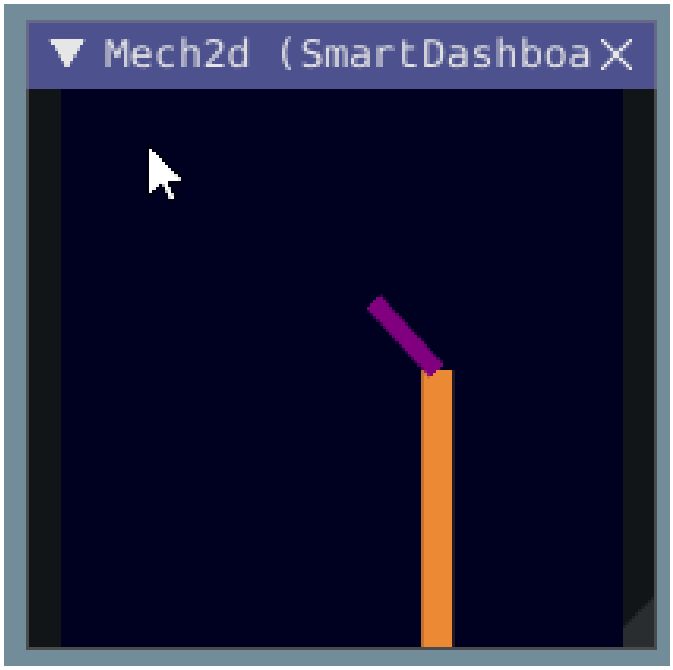
```
49      def robotPeriodic(self):
50          # update the dashboard mechanism's state
51          self.elevator.setLength(
52              self.kElevatorMinimumLength + self.elevatorEncoder.getDistance()
53          )
54          self.wrist.setAngle(self.wristPot.get())
```

Viewing the Mechanism2d in Glass

After sending the `Mechanism2d` instance over NetworkTables, the *Mechanism2d* 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.

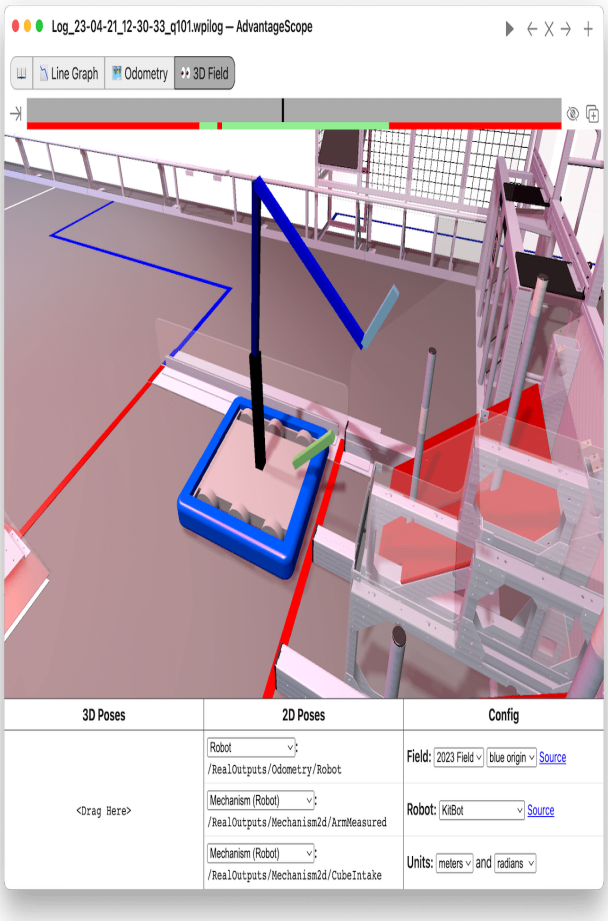


Once the widget appears as shown below, you can resize and place it on the Glass workspace as you desire. Right-clicking the top of the widget will allow you to customize the name of the widget. As the wrist potentiometer and elevator encoder changes, the mechanism will update in the widget.



Viewing the Mechanism2d in AdvantageScope

AdvantageScope is an alternative option for viewing a `Mechanism2d` object, including data recorded to a log file using *WPILib data logs*. Both 2D and 3D visualizations are supported. See the documentation for the mechanism and 3D field tabs for more details.



Next Steps

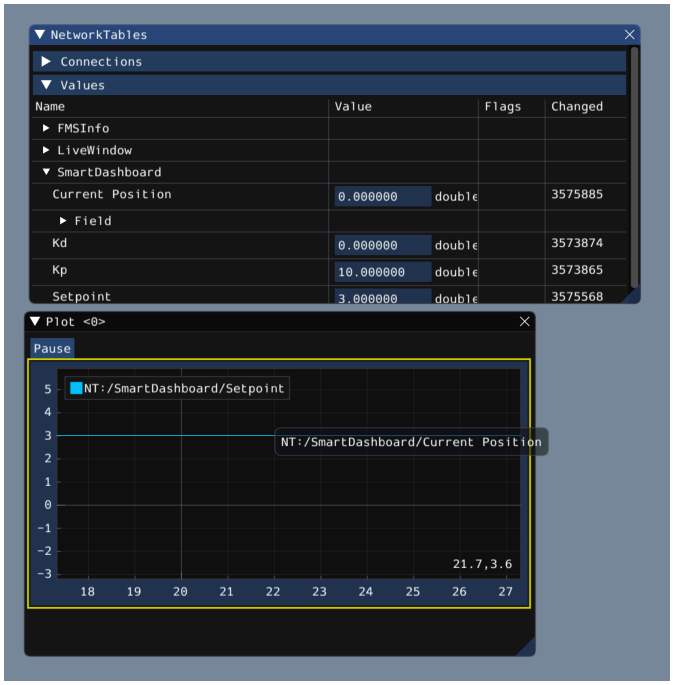
As mentioned above, the Mechanism2d visualization can be combined with *Physics Simulation* to help you program mechanisms before your robot is built. The ArmSimulation (Java / C++ / Python) and ElevatorSimulation (Java / C++ / Python) examples combine physics simulation and Mechanism2d visualization so that you can practice programming a single jointed arm and elevator without a robot.

11.4.7 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.

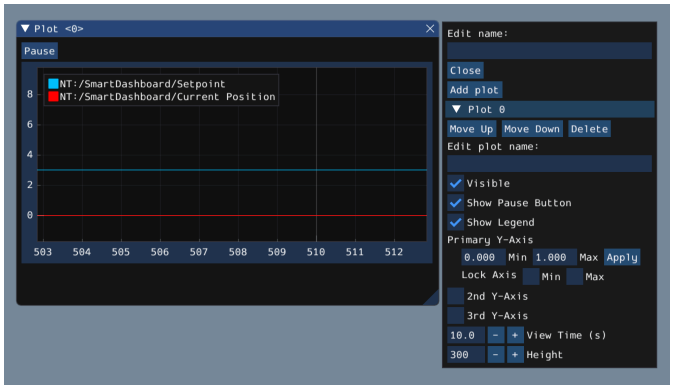
Creating a Plot

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:

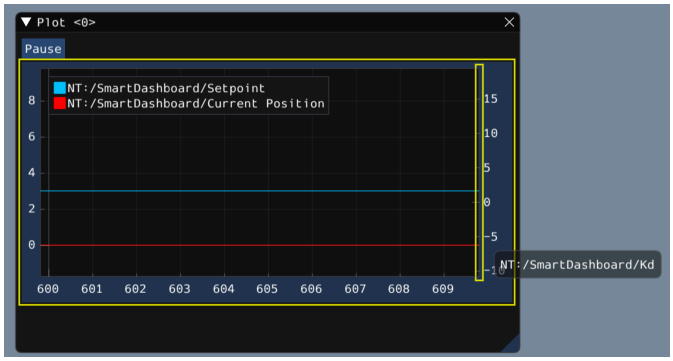


Manipulating Plots

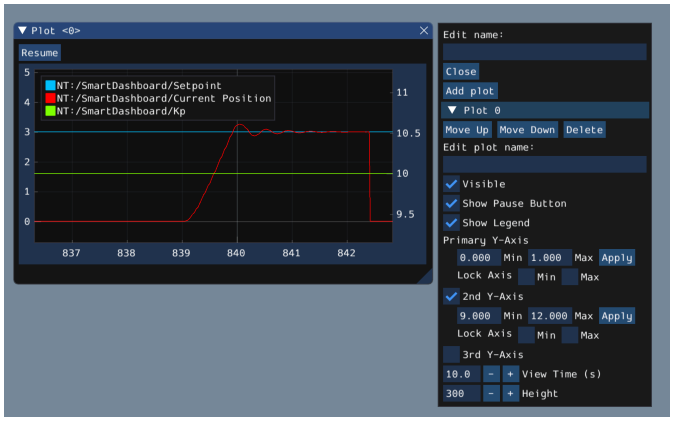
You can click and drag on the plot to move around and scroll on top of the plot to zoom the y axes in and out. Double clicking on the graph will autoscale it so that the zoom and axis limits fit all of the data it is plotting. Furthermore, right-clicking on the plot will present you with a plethora of 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:

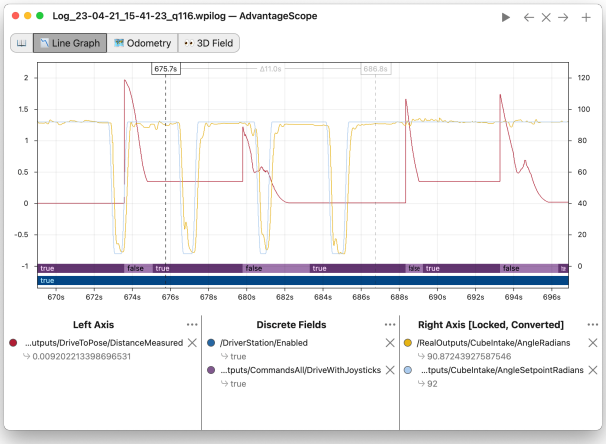


Then, you can lock certain axes so that their range always remains constant, regardless of panning. In this example, the secondary axis range (with the `/SmartDashboard/Kp` entry) was locked between 9 and 12.



Plotting with AdvantageScope

AdvantageScope is an alternative option for creating plots, including from data recorded to a log file using *WPILib data logs*. See the documentation for the line graph tab for more details.



11.5 AdvantageScope

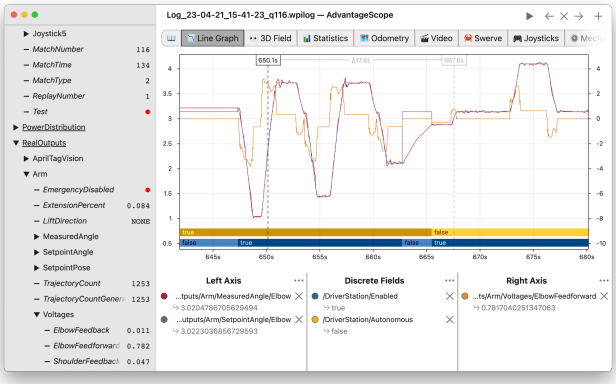
AdvantageScope is a data visualization tool for *NetworkTables*, *WPILib data logs*, and *Driver Station logs*. It is a programmer's tool (rather than a competition dashboard) and can be used to debug real or simulated robot code from a log file or live over the network.

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 *AdvantageScope*. You can also open any supported log file in AdvantageScope using a standard file browser.

> **Note**
>
> Detailed documentation for AdvantageScope can be found here. It is also available offline by clicking the book icon in the tab bar.

The capabilities of AdvantageScope include:

- Display of numeric, textual, and boolean data in graphs and tables
- Visualization of pose and mechanism data in 2D and 3D, including custom 3D robot models
- Automatic synchronization of data sources, including various types of log files and match videos
- Specialized displays for joysticks, swerve module states, and console text
- Analysis of numeric fields using histograms and statistical measures
- Multiple export options, including CSV, and WPILib data logs



11.6 Elastic

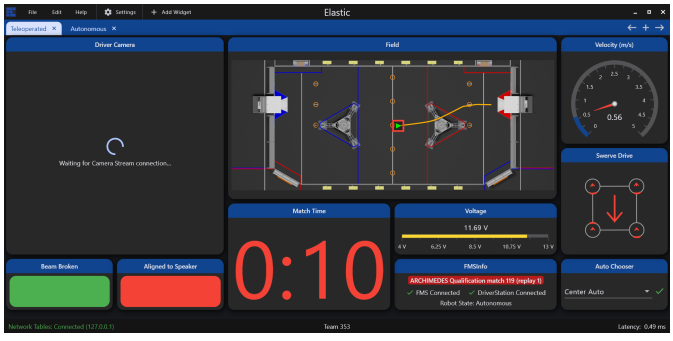
Elastic is a simple and modern dashboard made by Team 353. It is meant to be used behind the glass as a competition driver dashboard, but it can also be used for testing. It was built with the goal of being a drag & drop dashboard with an easy setup and elegant UI designed for a high pressure competition environment.

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 *Elastic*.

> **Note**
>
> Detailed documentation for Elastic can be found here.

The capabilities of Elastic include:

- Viewing data from different NT4 topics and widgets
- Draggable and resizable card widgets
- Customizable color scheme with over 20 variants
- Optimized camera streams which automatically deactivate when not in use
- Automatic IP retrieval from the FRC Driver Station



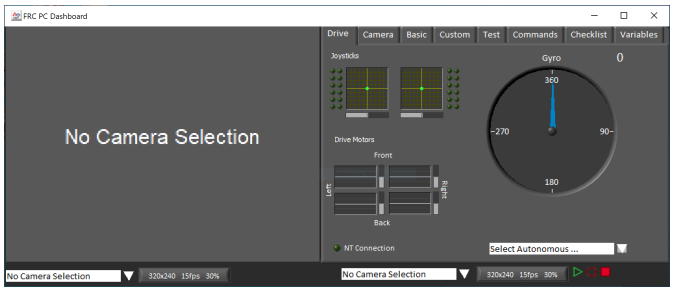
11.7 LabVIEW Dashboard

The LabVIEW Dashboard is easy to use and provides a lot of features straight out of the box like: camera streams, autonomous selection, and joystick feedback. It can be customized using LabVIEW by creating a new Dashboard project. While it *can be used* by Java, C++, or Python teams, they generally prefer SmartDashboard or Shuffleboard which can be customized in their respective language.

11.7.1 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.

LabVIEW Dashboard

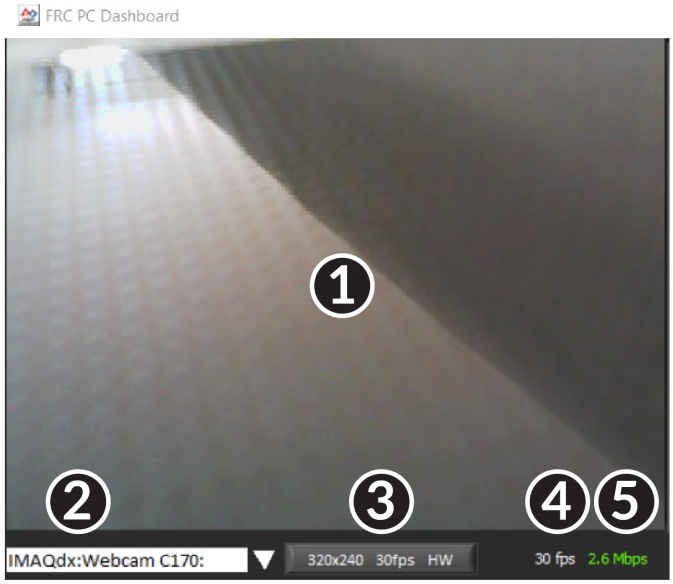


The Dashboard is broken into two main sections. The left pane is for displaying a camera image. The right pane contains:

- 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*.

Camera Image and Controls



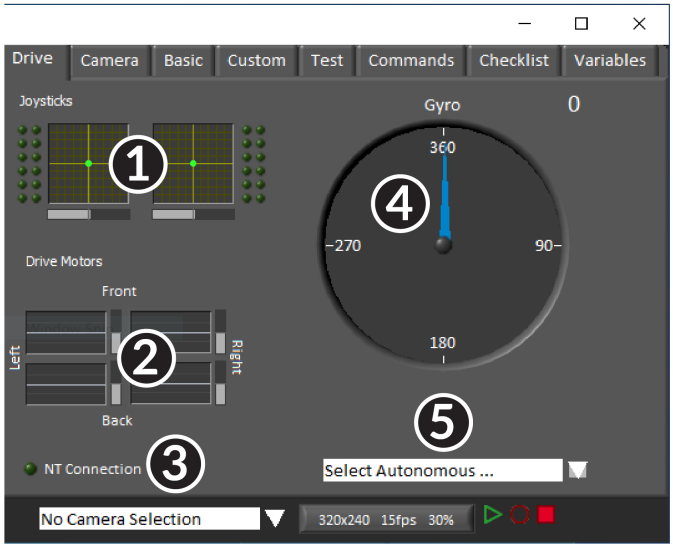
The left pane is used to display a video feed from acamera located on the robot. There are also some controls and indicators related to the camera below the tab area:

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. 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.

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:

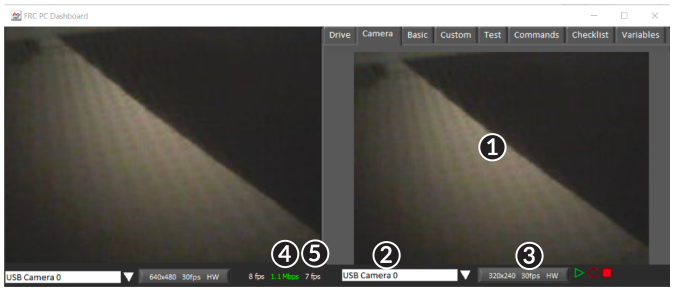
1. Displays X,Y, and Throttle information and button values for up to 2 joysticks when using the LabVIEW framework
2. Displays values being sent to motor controllers when using LabVIEW framework
3. Displays a connection indicator for the NetworkTables data from the robot
4. Displays a Gyro value
5. Displays 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, or Python Code*.

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.



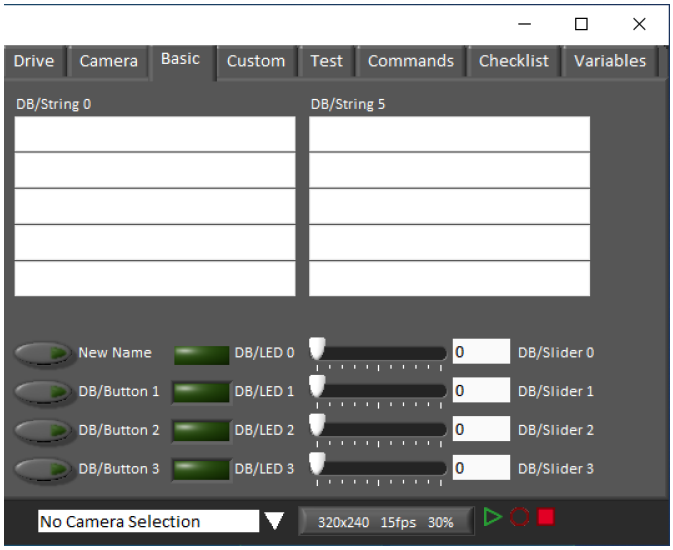
The camera tab is used to display a video feed from a camera located on the robot. There are also some controls and indicators related to the camera below the tab area:

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. 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

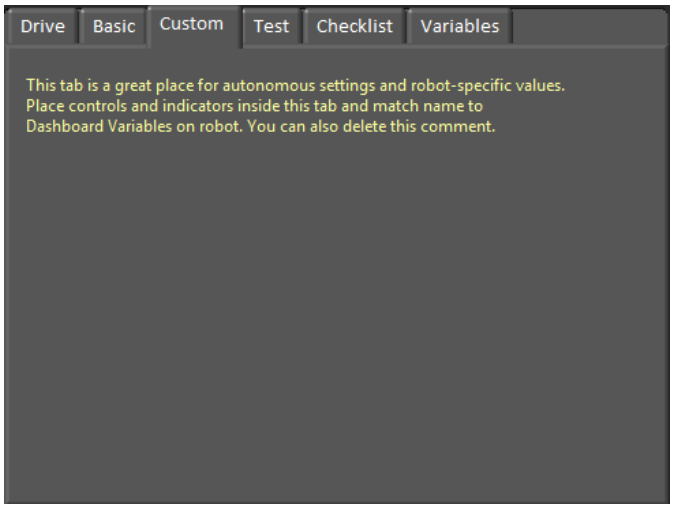
The bandwidth indicator indicates the combined bandwidth for all camera streams open.

Basic



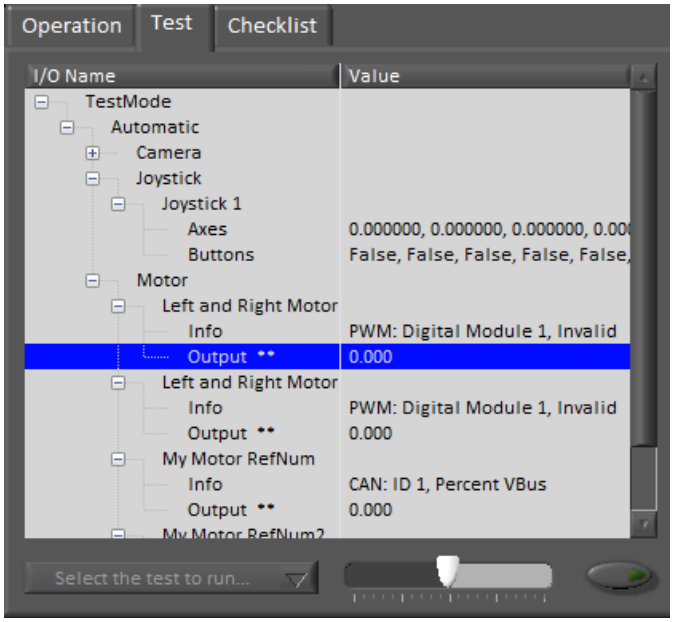
The Basic tab contains a variety of pre-populated bi-directional controls/indicators which can be used to control the robot or display information from the robot. The SmartDashboard key names associated with each item are labeled next to the indicator with the exception of the Strings which follow the 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, or Python Code*.

Custom



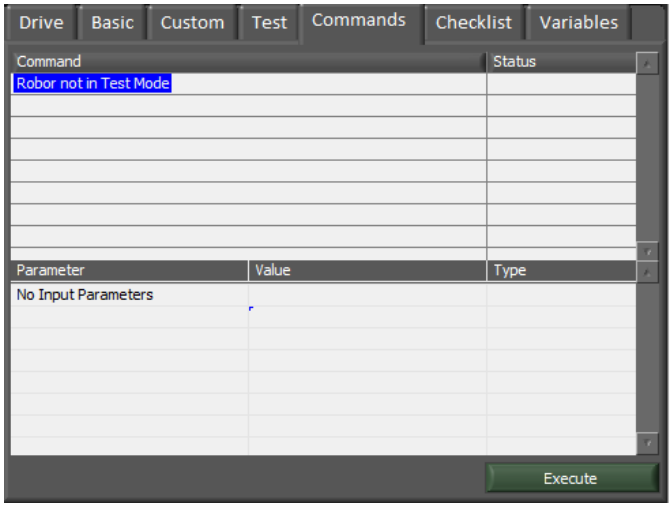
The Custom tab allows you to add additional controls/indicators to the dashboard using LabVIEW without removing any existing functionality. To customize this tab you will need to create a Dashboard project in LabVIEW.

Test



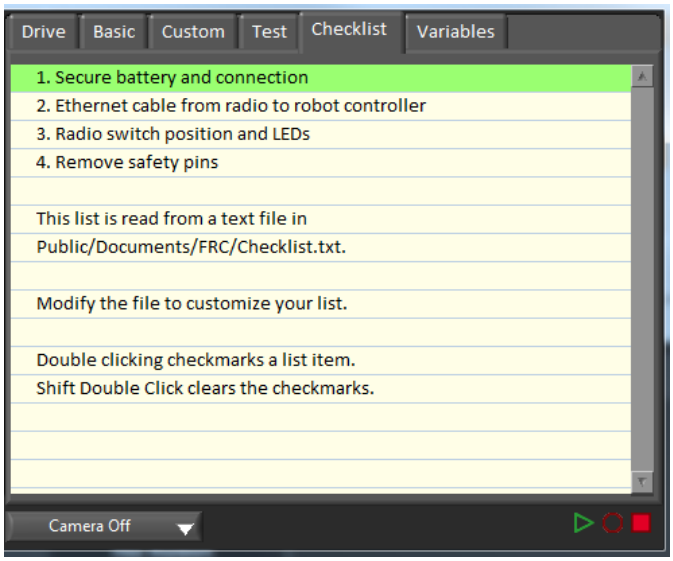
The Test tab is for use with Test mode for teams using LabVIEW (Java and C++ teams should use SmartDashboard or Shuffleboard when using Test Mode). For many items in the libraries, Input/Output info will be populated here automatically. All items which have ** next to them are outputs that can be controlled 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.

Commands



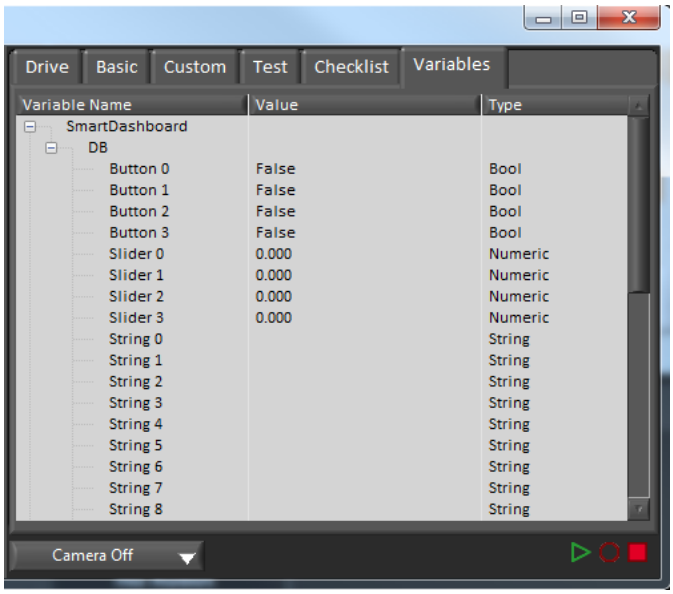
The Commands tab can be used with the Robot in Test mode to see which commands are running and to manually run commands for test purposes.

Checklist



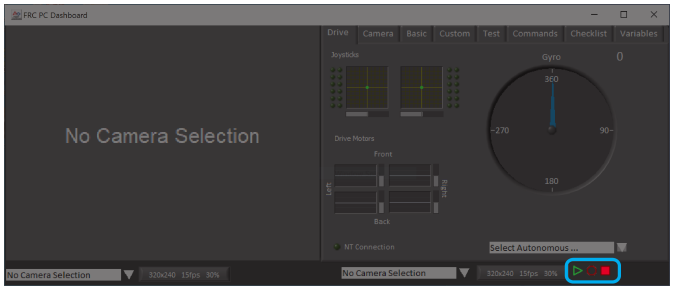
The Checklist tab can be used by teams to create a list of tasks to perform before or between matches. Instructions for using the Checklist tab are pre-populated in the default checklist file.

Variables



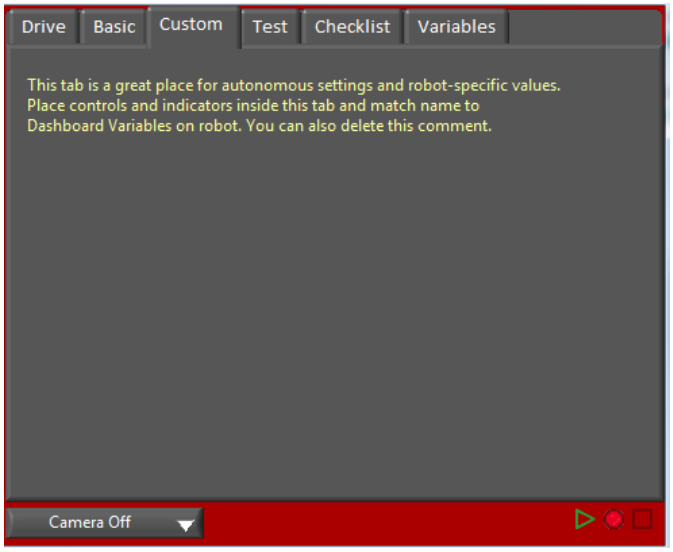
The Variables tab of the left pane shows all NetworkTables variables in a tree display. The Variable Name (Key), Value and data type are shown for each variable. Information about the NetworkTables bandwidth usage is also displayed in this tab. Entries will be shown with black diamonds if they are not currently synced with the robot.

Record/Playback



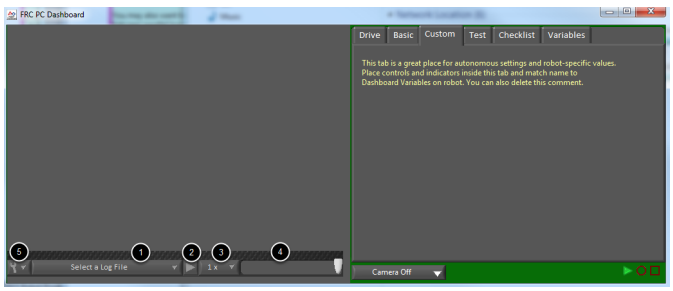
The LabVIEW Dashboard includes a Record/Playback feature that allows you to record video and NetworkTables data (such as the state of your Dashboard indicators) and play it back later.

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



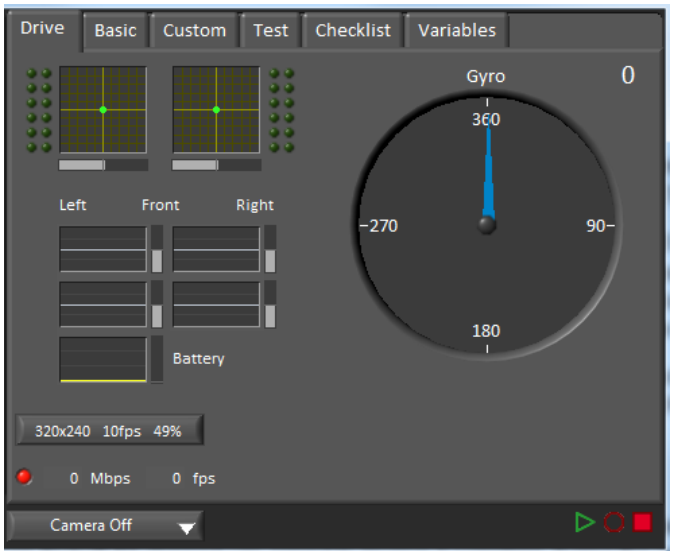
To play a recording back, click the green triangle Play button. The background of the right pane will begin pulsing green and playback controls will appear at the bottom of the camera pane.

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`)

11.7.2 Using the LabVIEW Dashboard with C++, Java, or Python Code

The default LabVIEW Dashboard utilizes *NetworkTables* to pass values and is therefore compatible with C++, Java, and Python robot programs. This article covers the keys and value ranges to use to work with the Dashboard.

Drive Tab



The *Select Autonomous...* dropdown can be used so show the available autonomous routines and choose one to run for the match.

JAVA

```
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
}
```

C++

```
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
}
```

PYTHON

```
from wpilib import SmartDashboard
   SmartDashboard.putStringArray("Auto List", ["Drive Forwards", "Drive Backwards",
↪"Shoot"])
   # At the beginning of auto
autoName = SmartDashboard.getString("Auto Selector", "Drive Forwards") # This would␣
↪make "Drive Forwards the default auto
match 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

```
SmartDashboard.putNumber("Gyro", drivetrain.getHeading());
```

C++

```
frc::SmartDashboard::PutNumber("Gyro", Drivetrain.GetHeading());
```

PYTHON

```
from wpilib import SmartDashboard
SmartDashboard.putNumber("Gyro", self.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

```
SmartDashboard.putNumberArray("RobotDrive Motors", {drivetrain.getLeftFront(),␣
↪drivetrain.getRightFront(), drivetrain.getLeftBack(), drivetrain.getRightBack()});
```

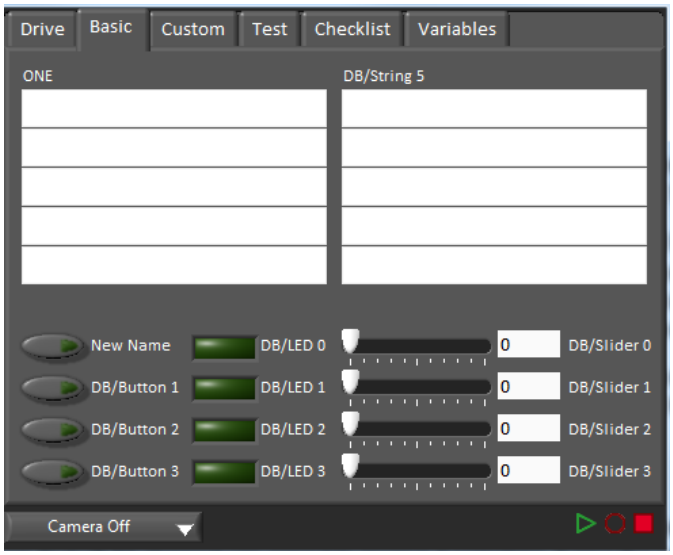
C++

```
frc::SmartDashboard::PutNumberArray("Gyro", {drivetrain.GetLeftFront(), drivetrain.
↪GetRightFront(), drivetrain.GetLeftBack(), drivetrain.GetRightBack()});
```

PYTHON

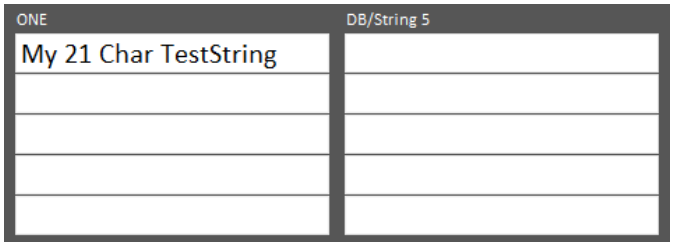
```
from wpilib import SmartDashboard
SmartDashboard.putNumberArray("RobotDrive Motors", [self.drivetrain.getLeftFront(),␣
↪self.drivetrain.getRightFront(), self.drivetrain.getLeftBack(), self.drivetrain.
↪getRightBack()])
```

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

```
SmartDashboard.putString("DB/String 0", "My 21 Char TestString");
```

C++

```
frc::SmartDashboard::PutString("DB/String 0", "My 21 Char TestString");
```

PYTHON

```
from wpilib import SmartDashboard
SmartDashboard.putString("DB/String 0", "My 21 Char TestString")
```

To read string data entered on the Dashboard:

JAVA

```
String dashData = SmartDashboard.getString("DB/String 0", "myDefaultData");
```

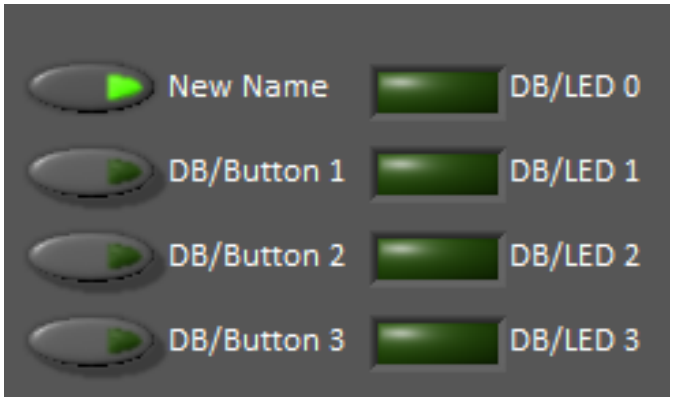
C++

```
std::string dashData = frc::SmartDashboard::GetString("DB/String 0", "myDefaultData");
```

PYTHON

```
from wpilib import SmartDashboard
dashData = 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

```
SmartDashboard.putBoolean("DB/Button 0", true);
```

C++

```
frc::SmartDashboard::PutBoolean("DB/Button 0", true);
```

PYTHON

```
from wpilib import SmartDashboard
SmartDashboard.putBoolean("DB/Button 0", true)
```

To read from the Buttons: (default value is false)

JAVA

```
boolean buttonValue = SmartDashboard.getBoolean("DB/Button 0", false);
```

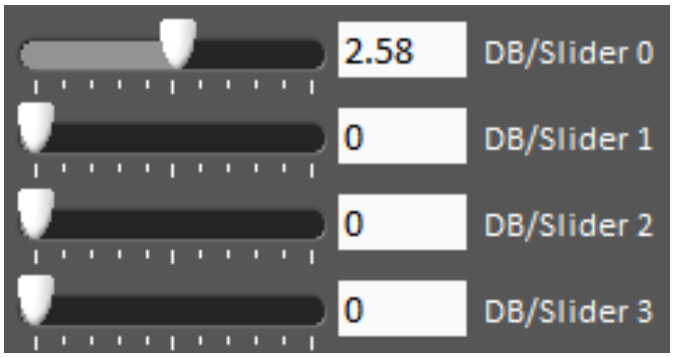
C++

```
bool buttonValue = frc::SmartDashboard::GetBoolean("DB/Button 0", false);
```

PYTHON

```
from wpilib import SmartDashboard
buttonValue = 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

```
SmartDashboard.putNumber("DB/Slider 0", 2.58);
```

C++

```
frc::SmartDashboard::PutNumber("DB/Slider 0", 2.58);
```

PYTHON

```
from wpilib import SmartDashboard
SmartDashboard.putNumber("DB/Slider 0", 2.58)
```

To read values from the Dashboard into the robot program: (default value of 0.0)

JAVA

```
double dashData = SmartDashboard.getNumber("DB/Slider 0", 0.0);
```

C++

```
double dashData = frc::SmartDashboard::GetNumber("DB/Slider 0", 0.0);
```

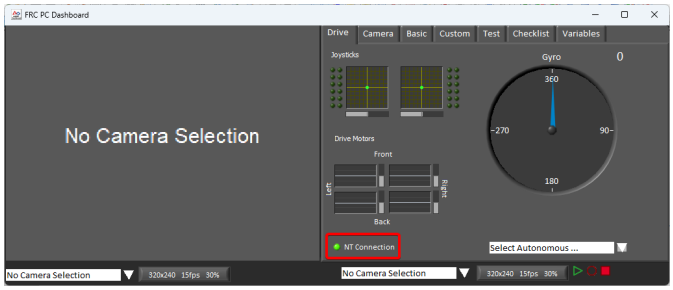
PYTHON

```
from wpilib import SmartDashboard
dashData = SmartDashboard.getNumber("DB/Slider 0", 0.0)
```

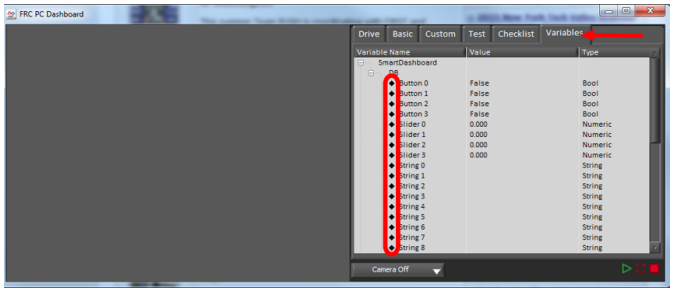
11.7.3 Troubleshooting Dashboard Connectivity

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.

Recognizing LabVIEW Dashboard Connectivity

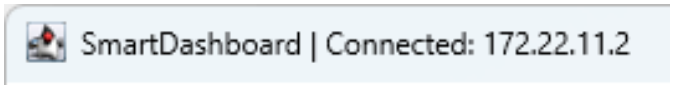


The LabVIEW Dashboard has a NetworkTables Connection indicator on the front panel.

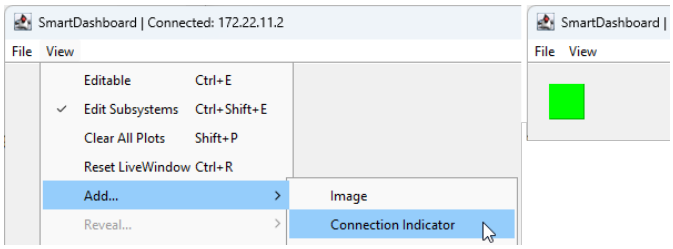


On the Variables tab of the Dashboard, the variables are shown with a black diamond when they are not synced with the robot. Once the Dashboard connects to the robot and these variables are synced, the diamond will disappear.

Recognizing SmartDashboard Connectivity



SmartDashboard indicates if it is connected or not in the title bar. It shows the IP address it is connected to. See *this page* for more on configuring the connection.



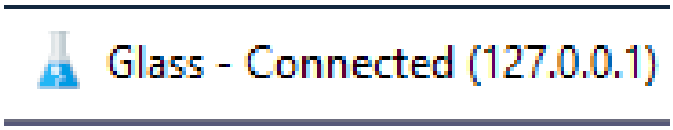
For more visibility, you can also add a Connection Indicator widget. The connection indicator can be moved or re-sized if the Editable checkbox is checked.

Recognizing Shuffleboard Connectivity



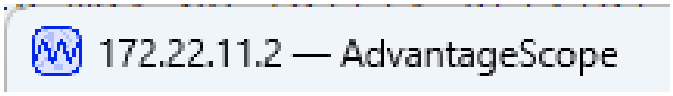
Shuffleboard indicates if it is connected or not in the bottom right corner of the application as shown in the image above. See *page* for more on configuring the connection.

Recognizing Glass Connectivity



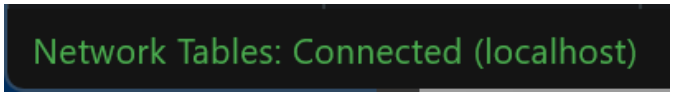
Glass indicates if it is connected or not in the title bar. It shows the IP address it is connected to. See this *page* for more on configuring the connection.

Recognizing AdvantageScope Connectivity



AdvantageScope indicates if it is connected or not in the title bar. It shows the IP address it is connected to, or else the IP address it is attempting to connect to. See the AdvantageScope Documentation for more on configuring the connection.

Recognizing Elastic Connectivity



Elastic indicates if it is connected or not in the lower left corner. It shows what it is connected to. See the Elastic Documentation 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

12 Telemetry

12.1 Telemetry: Recording and Sending Real-Time Data

Recording and viewing *telemetry* data is a crucial part of the engineering process - accurate telemetry data helps you tune your robot to perform optimally, and is indispensable for debugging your robot when it fails to perform as expected.

By default, no telemetry data is recorded (saved) on the robot. However, recording data on the robot can provide benefits over recording on a dashboard, namely that more data can be recorded (there are no bandwidth limitations), and all the recorded data can be very accurately timestamped. WPILib has integrated support for on-robot recording of telemetry data via the `DataLogManager` and `DataLog` classes and provides a tool for downloading data log files and converting them to CSV. The Java library also provides a convenient *annotation* to autogenerate telemetry logging code based on your project.

Note

In addition to on-robot recording of telemetry data, teams can record their telemetry data on their driver station computer with *Shuffleboard recordings*.

12.1.1 Adding Telemetry to Robot Code

WPILib supports several different ways to record and send telemetry data from robot code.

At the most basic level, the *Riolog* provides support for viewing print statements from robot code. This is useful for on-the-fly debugging of problematic code, but does not scale as console interfaces are not suitable for rich data streams.

WPILib supports several *dashboards* that allow users to more easily send rich telemetry data to the driver-station computer. All WPILib dashboards communicate with the *NetworkTables* protocol, and so they are *to some degree* interoperable (telemetry logged with one dashboard will be visible on the others, but the specific widgets/formatting will generally not be compatible). NetworkTables (and thus WPILib all dashboards) currently support the following data types:

- `boolean`

- `boolean[]`
- `double`
- `double[]`
- `string`
- `string[]`
- `byte[]`

Telemetry data can be sent to a WPILib dashboard using an associated WPILib method (for more details, see the documentation for the individual dashboard in question), or by *directly publishing to NetworkTables*.

While NetworkTables does not yet support serialization of complex data types (this is tentatively scheduled for 2024), *mutable* types from user code can be easily extended to interface directly with WPILib dashboards via the `Sendable` interface, whose usage is described in the next article.

12.2 New for 2025: Robot Telemetry with Annotations (Java Only)

At its most basic, telemetry with annotations can be accomplished with only a few lines of code:

1. Add the `@Logged` annotation to your main robot class and to the classes it references, like subsystems
2. Bind the logger to the robot in the constructor to start logging

This is enough to start logging data to NetworkTables for display and capture on a desktop application like a dashboard or AdvantageScope. Optionally, you can start the data log manager to capture the NetworkTables-logged data and save it to a log file on the roboRIO.

JAVA

```
@Logged
public class Robot extends TimedRobot {
  private final Arm arm;
  private final Drivetrain drivetrain;

  public Robot() {
    arm = new Arm();
    drivetrain = new Drivetrain();
    DataLogManager.start(); // Optional to mirror the NetworkTables-logged data to a␣
↪file on disk
    Epilogue.bind(this);
  }
}

@Logged
public class Arm {
  // ...
}
```

(continues on next page)

(continued from previous page)

```
@Logged
public class Drivetrain {
  // ...
}
```

The `@Logged` annotation, when placed on a class, tells the Java compiler to run WPILib's code generator to write custom logging code that will log everything it can in your class. The generated code will exist in the `build/generated/sources/annotationProcessor` folder in your robot project, and your code can reference and use that generated code. The `Epilogue` class is, in fact, one of those generated files - it is responsible for tracking all of the custom class-specific loggers created by the code generator and is the entry point for configuring and starting automatic logging. Placing the `@Logged` annotation on individual fields and methods will let you control how they appear in log files or networktables; placing `@NotLogged` on a field or method will explicitly exclude it from being logged, regardless of how its enclosing class is configured.

Annotation logging will detect all public and private fields and all public methods in an annotated class, as long as those fields and methods return a known loggable type. Placing an `@Logged` annotation on any non-loggable field or method will result in a compilation error. Static fields and methods cannot be logged, nor can non-public methods or methods that accept arguments.

Currently, annotation logging supports the following data types:

- Any type (class, interface, or enum) tagged with the `@Logged` annotation
- Any type (class, interface, or enum) with a custom logger defined with a `@CustomLoggerFor` annotation (intended to be used for logging of third-party classes)
- All primitive types (`byte`, `char`, `short`, `int`, `long`, `float`, `double`, and `boolean`) (note: this excludes wrapper types like `Byte`, `Integer`, and `Boolean`)
- Strings
- Any type that inherits from `StructSerializable`, such as WPILib math types or custom user-defined types
- Any `edu.wpi.first.units.Measure` (logged as a double in terms of the base unit)
- `byte[]`, `int[]`, `long[]`, `float[]`, `double[]`, and `boolean[]` primitive arrays
- `String[]` arrays
- Arrays of anything that inherits from `StructSerializable`, such as `Rotation2d[]` or `SwerveModuleState[]`
- Java collections, like `List` and `Set`, of `String` or `StructSerializable` objects, such as `Set<String>` or `List<SwerveModuleState>`
- Any `Sendable` object, like a `DigitalInput` (note: commands and subsystems are excluded and will not be logged using their sendable implementation) * `Mechanism2D` objects will not be logged as they do not implement the `Sendable` interface in a way that is compatible with annotation logging.
- Any `BooleanSupplier`, `IntSupplier`, `LongSupplier`, or `DoubleSupplier` (note: this includes command triggers!)

12.2.1 Customizing your logging setup

The default logging behavior is to log *every* non-static field (public or private) and public method in a class annotated with `@Logged`, where the associated log entries will use the name of the element. For example, a field declared as `private double m_someValue` will appear in logs under the entry `"m_someValue"`, and a method declared as `public double getSomeValue()` will appear under the entry `"getSomeValue"`.

If you desire to only log certain fields or methods within a class, you can remove the `` `@Logged` `` annotation from the class itself and place it on individual fields/methods within the class. This allows you to opt-in on logging within a class instead of logging every value.

The names of log entries can be changed using the `name` configuration option on fields and methods (on a class-level annotation, it changes the name of the autogenerated logger class for avoiding name conflicts and does not change the log names of any instance of that class). For example, a `private double m_someValue` field with an `@Logged(name = "Motor Temperature")` will appear in logs as `"Motor Temperature"` instead of the default `"m_someValue"` that would be used.

Table 1: `@Logged` Configurations

Attribute	Description
Name	Sets a specific name to use for the annotated element. If placed on a class, this controls the name of the generated logger class for avoiding name conflicts.
Importance	Sets the specific importance level for the annotated element, which can be used by the runtime `minimumImportance` configuration to control what data to log. If placed on a class, this sets the *default* importance level for all contained elements, which can be overridden on an element-by-element basis. Defaults to `DEBUG`.
Logging Period	Sets the amount of time between logging calls.
Logging Period Offset	Sets the offset of logging calls relative to the robot periodic functions.

Original code without logging

```
public class Robot extends RobotBase {
  private final Arm arm;

  public Robot() {
    arm = new Arm();
  }
}

class Arm {
  public final Trigger atLowStop = new Trigger(...);
  public final Trigger atHighStop = new Trigger(...);
```

(continues on next page)

(continued from previous page)

```
  private Rotation2d lastPosition = getPosition();

  public Rotation2d getPosition() {
    // ...
  }

  public Measure<Velocity<Angle>> getSpeed() {
    // ...
  }
}
```

Code with logging (minimal)

```
@Logged
public class Robot extends RobotBase {
  private final Arm arm; // Anything loggable within the arm object will be logged␣
↪under an "arm" entry

  public Robot() {
    arm = new Arm();
    Epilogue.bind(this);
  }
}

@Logged
class Arm {
  public final Trigger atLowStop = new Trigger(...);  // Logged as a boolean in an␣
↪"atLowStop" entry
  public final Trigger atHighStop = new Trigger(...); // Logged as a boolean in an␣
↪"atHighStop" entry
  private Rotation2d lastPosition = getPosition();    // Logged as a Rotation2d␣
↪struct in a "lastPosition" entry

  // Logged as a Rotation2d struct object in a "getPosition" entry
  public Rotation2d getPosition() {
    // ...
  }

  // Logged as a double in terms of radians per second in a "getSpeed" entry
  public Measure<Velocity<Angle>> getSpeed() {
    // ...
  }
}
```

Data will be logged as:

` /Robot/arm/atLowStop /Robot/arm/atHighStop /Robot/arm/lastPosition /Robot/arm/getPosition /Robot/arm/getSpeed `

Code with logging (configured)

```
@Logged
public class Robot extends RobotBase {
  @Logged(name = "Arm")
  private Arm arm;

  public Robot() {
    arm = new Arm();
    DataLogManager.start();
    Epilogue.bind(this);
  }
}

class Arm {
  @Logged(name = "At Low Stop", importance = DEBUG)
  public final Trigger atLowStop = new Trigger(...);

  @Logged // defaults to an importance of DEBUG and the name as "atHighStop"
  public final Trigger atHighStop = new Trigger(...);

  @NotLogged // Redundant because the class strategy is opt-in
  private Rotation2d lastPosition = getPosition(); // No @Logged annotation, not␣
↪logged

  @Logged(name = "Position", importance = CRITICAL)
  public Rotation2d getPosition() {
    // ...
  }

  @Logged
  public Measure<Velocity<Angle>> getSpeed() {
      // ...
  }
}
```

Data will be logged as:

` /Robot/Arm/At Low Stop /Robot/Arm/atHighStop /Robot/Arm/Position /Robot/Arm/getSpeed `

12.2.2 The Epilogue Class

`Epilogue` is a special class that is created by the code generator. Its responsibility is to keep track of all the custom logger files and as the main entry point for configuring and starting automatic data logging.

If your main robot class inherits from `TimedRobot`, the generated `Epilogue` class will have an additional `bind()` method that can be used to add a periodic logging call to the robot. The logging call will run at the same frequency as the main robot loop (typically 50Hz), but offset by half a phase to avoid extra CPU overhead while control loops are running. Keep in mind that, depending on your code structure, this means that logged data may be slightly out of sync with the data that the control loops operate on.

Table 2: Logging Configurations

Configuration	Description	Default Value
Data Logger	The data logging implementation to use to save logged data.	NetworkTables-based logging, which the DataLogManager can read and save to a file on disk
Root Data Entry	The root entry that logged data will appear under. Used for both NetworkTables-based and DataLog-based logging. Can be set to any string.	`"Robot"`
Minimum Importance Level	The minimum importance level of data for it to be logged. Any logged data marked with a lower priority will be excluded from telemetry.	`DEBUG` (logs everything)
Error handler	Catches and responds to errors encountered while logging data. Error handlers are used to prevent any errors encountered during logging from crashing the entire robot program.	Errors are caught and printed to the console

JAVA

```java
@Logged
public class Robot extends TimedRobot {
  public Robot() {
    Epilogue.configure(config -> {
      // Log only to disk, instead of the default NetworkTables logging
      // Note that this means data cannot be analyzed in realtime by a dashboard
      config.backend = new FileBackend(DataLogManager.getLog());

      if (isSimulation()) {
        // If running in simulation, then we'd want to re-throw any errors that
        // occur so we can debug and fix them!
        config.errorHandler = ErrorHandler.crashOnError();
      }

      // Change the root data path
      config.root = "Telemetry";

      // Only log critical information instead of the default DEBUG level.
      // This can be helpful in a pinch to reduce network bandwidth or log file size
      // while still logging important information.
      config.minimumImportance = Logged.Importance.CRITICAL;
    });
    Epilogue.bind(this);
  }
}
```

12.2.3 Logging Third-Party Data

A large part of the Java ecosystem is its access to third-party libraries that provide extra functionality, such as vendor libraries for interacting with custom hardware like motor controllers and sensors. If those vendors do not use logging annotations in their libraries, then you can still log them by writing a custom logger class! These classes are actually what the WPILib annotations will generate; you just write a custom one because third-party code cannot be annotated.

Custom loggers can be declared in any package, and only need to have the `@CustomLoggerFor` annotation present to be able to be detected and used. They must extend from `ClassSpecificLogger` (Javadoc) and must have a public no-argument constructor - failing to declare one will result in a compilation error - and must pass the type they log to the superclass constructor.

> **Note**
>
> Only one custom logger may be defined for a single type. Custom loggers will only be detected and used if they are defined in the robot project; custom loggers defined in third-party libraries cannot be detected.

JAVA

```java
class VendorMotor {
  public int getFaults();
  public void set(double speed);
  public double get();
  public double getAppliedVoltage();
  public double getInputCurrent();
}

@CustomLoggerFor(VendorMotor.class)
public class YourCustomVendorMotorLogger extends ClassSpecificLogger<VendorMotor> {
  public YourCustomVendorMotorLogger() {
    super(VendorMotor.class);
  }

  @Override
  public void update(EpilogueBackend backend, VendorMotor motor) {
    if (Epilogue.shouldLog(Logged.Importance.DEBUG)) {
      backend.log("Faults", motor.getFaults());
    }
    backend.log("Requested Speed (Duty Cycle)", motor.get());
    backend.log("Motor Voltage (V)", motor.getAppliedVoltage());
    backend.log("Input Current (A)", motor.getInputCurrent());
  }
}
```

12.2.4 Caveats and Limitations

Because the logging code generation runs at compile time, information only available at runtime cannot be used. If a field is declared to be of some type `T`, then it will be logged using whatever configuration is defined for `T`, even if at runtime it is set to a concrete subtype `S` that has its own distinct logging configuration. You may encounter this if your code uses IO interfaces; consider annotating the interface with `@Logged` and provide methods for reading data you want to be logged regardless of implementation, such as motor voltages and sensor readings.

Null values are unrepresentable in log data and therefore cannot be logged. If a logged field is set to `null` or is uninitialized, or if a logged method returns `null`, then the logging code will throw a `NullPointerException` to be handled by the configured error handler.

Data logging increases CPU load on the roboRIO and can lead to loop overruns. The CPU load is not caused by the logging itself, but from calling expensive methods to read data. This is most often seen when querying information from devices on a CAN bus. If your code frequently sees loop overruns caused by logging, consider reducing the number of logged fields or restructure your code to periodically read and cache data from connected devices and log the cached data instead of logging methods that needlessly re-query it.

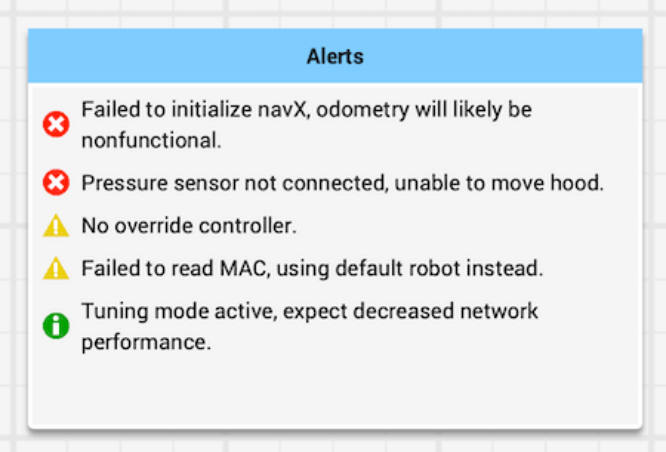
Note

Time spent on logging in each loop will be logged to `/Epilogue/Stats/Last Run` in NetworkTables and can be analyzed in a tool like AdvantageScope.

12.3 Recording Faults with Persistent Alerts

Robots encounter a variety of fault conditions: disconnected sensors/motors, invalid mechanism states, initialization failures, etc. While the FRC Driver Station provides a *console interface* for instantaneous alerts, the risk of missing important messages makes it poorly suited to communicate faults that persist over time.

Instead, the `Alert` class (Java, C++, Python) can be used for managing persistent alerts published via Network Tables. Alerts are assigned a priority (*error*, *warning*, or *info*) and can be *activated* or *deactivated* in robot code. The set of active alerts can be displayed on a dashboard, as shown below in Shuffleboard.



Active alerts are automatically displayed in order based on priority and how recently they were activated, with newer and more critical alerts at the top of the list. This provides a mechanism for drivers (on the field) and programmers (off the field) to quickly assess the state of the robot and determine whether it is in a "match-ready" state. Highly descriptive alerts also allow drive teams to adapt to faults in real time, such as switching strategies in case of a subsystem failure.

12.3.1 API Usage

Alerts are created and managed using the `Alert` class. Typically, an alert should be instantiated once and stored as a class member. The `set` method accepts a boolean for the alert state, which activates or deactivates the alert. Change detection is built-in, so calling `set` multiple times with the same value does affect the order in which alerts are displayed.

Alert states are often expressed most easily as a conditional, such as whether the latest signal from a CAN device is up-to-date. In this case, we recommend calling the `set` method in a periodic loop and relying on the built-in change detection to activate the alert.

JAVA

```
class Robot {
  Alert alert = new Alert("Something went wrong", AlertType.kWarning);

  periodic() {
    alert.set(...);
  }
}
```

C++

```
class Robot {
  frc::Alert alert{"Something went wrong", frc::Alert::AlertType::kWarning};
}

Robot::periodic() {
  alert.Set(...);
}
```

PYTHON

```
self.alert = Alert("Something went wrong", AlertType.kWarning)

def periodic() {
  self.alert.set(...)
}
```

> **Note**
>
> Suggested usage of each alert type (error, warning, or info) is provided in the enum API documentation.

The text description of an alert can be updated after the alert has been created by calling `setText`, allowing additional information to be provided inline. For example, an alert indicating that a motor is overheating could include the motor's current temperature.

Optionally, a group name can also be provided to the `Alert` constructor. Alerts are published to Network Tables using this name, allowing multiple sets of alerts to be displayed on separate widgets (such as "Auto" and "Teleop" alert sets). If the group name is omitted, the alert is published to a group called "Alerts".

12.3.2 Dashboard Usage

Alerts are published to Network Tables using the key `/SmartDashboard/<Group Name>`. Shuffleboard automatically populates data from this table to the "SmartDashboard" tab, or the widget can be manually created by dragging the alert table from the Network Tables tree to the main view as described *here*.

> **Note**
>
> In addition to Shuffleboard, the alerts API is supported by many third-party dashboards. Check the documentation of your preferred dashboard for details.

12.4 Robot Telemetry with Sendable

While the WPILib dashboard APIs allow users to easily send small pieces of data from their robot code to the dashboard, it is often tedious to manually write code for publishing telemetry values from the robot code's operational logic.

A cleaner approach is to leverage the existing object-oriented structure of user code to mark important data fields for telemetry logging in a *declarative programming* style. The WPILib framework can then handle the tedious/tricky part of correctly reading from (and, potentially, *writing to*) those fields for you, greatly reducing the total amount of code the user has to write and improving readability.

WPILib provides this functionality with the `Sendable` interface. Classes that implement `Sendable` are able to register value listeners that automatically send data to the dashboard - and, in some cases, receive values back. These classes can be declaratively sent to any of the WPILib dashboards (as one would an ordinary data field), removing the need for teams to write their own code to send/poll for updates.

12.4.1 What is Sendable?

`Sendable` (Java, C++, `Python`) is an interface provided by WPILib to facilitate robot telemetry. Classes that implement `Sendable` can declaratively send their state to the dashboard - once declared, WPILib will automatically send the telemetry values every robot loop. This removes the need for teams to handle the iteration-to-iteration logic of sending and receiving values from the dashboard, and also allows teams to separate their telemetry code from their robot logic.

Many WPILib classes (such as *Commands*) already implement `Sendable`, and so can be sent to the dashboard without any user modification. Users are also able to easily extend their own classes to implement `Sendable`.

The `Sendable` interface contains only one method: `initSendable`. Implementing classes override this method to perform the binding of in-code data values to structured *JSON* data, which is then automatically sent to the robot dashboard via NetworkTables. Implementation of the `Sendable` interface is discussed in the *next article*.

12.4.2 Sending a Sendable to the Dashboard

> **Note**
>
> Unlike simple data types, Sendables are automatically kept up-to-date on the dashboard by WPILib, without any further user code - "set it and forget it". Accordingly, they should usually be sent to the dashboard in an initialization block or constructor, *not* in a periodic function.

To send a `Sendable` object to the dashboard, simply use the dashboard's `putData` method. For example, an "arm" class that uses a *PID Controller* can automatically log telemetry from the controller by calling the following in its constructor:

JAVA

```
SmartDashboard.putData("Arm PID", armPIDController);
```

C++

```
frc::SmartDashboard::PutData("Arm PID", &armPIDController);
```

PYTHON

```
from wpilib import SmartDashboard
SmartDashboard.putData("Arm PID", armPIDController)
```

Additionally, some `Sendable` classes bind setters to the data values sent *from the dashboard to the robot*, allowing remote tuning of robot parameters.

12.5 On-Robot Telemetry Recording Into Data Logs

By default, no telemetry data is recorded (saved) on the robot. The `DataLogManager` class provides a convenient wrapper around the lower-level `DataLog` class for on-robot recording of telemetry data into data logs. The WPILib data logs are binary for size and speed reasons. In general, the data log facilities provided by WPILib have minimal overhead to robot code, as all file I/O is performed on a separate thread–the log operation consists of mainly a mutex acquisition and copying the data.

12.5.1 Structure of Data Logs

Similar to NetworkTables, data logs have the concept of entries with string identifiers (keys) with a specified data type. Unlike NetworkTables, the data type cannot be changed after the entry is created, and entries also have metadata–an arbitrary (but typically JSON) string that can be used to convey additional information about the entry such as the data source or data schema. Also unlike NetworkTables, data log operation is unidirectional–the `DataLog` class can only write data logs (it does not support read-back of written values) and the `DataLogReader` class can only read data logs (it does not support changing values in the data log).

Data logs consist of a series of timestamped records. Control records allow starting, finishing, or changing the metadata of entries, and data records record data value changes. Timestamps are stored in integer microseconds; when running on the RoboRIO, the FPGA timestamp is used (the same timestamp returned by `Timer.getFPGATimestamp()`).

Note

For more information on the details of the data log file format, see the WPILib Data Log File Format Specification.

12.5.2 Standard Data Logging using DataLogManager

The `DataLogManager` class (Java, C++, `Python`) provides a centralized data log that provides automatic data log file management. It automatically cleans up old files when disk space is low and renames the file based either on current date/time or (if available) competition match number. The data file will be saved to a USB flash drive in a folder called `logs` if one is attached, or to `/home/lvuser/logs` otherwise.

> **Note**
>
> USB flash drives need to be formatted as FAT32 to work with the roboRIO. NTFS or exFAT formatted drives will not work. Flash drives of 32GB or smaller are recommended, as Windows doesn't format drives larger then 32GB with FAT32.

Log files are initially named `FRC_TBD_{random}.wpilog` until the DS connects. After the DS connects, the log file is renamed to `FRC_yyyyMMdd_HHmmss.wpilog` (where the date/time is UTC). If the *FMS* is connected and provides a match number, the log file is renamed to `FRC_yyyyMMdd_HHmmss_{event}_{match}.wpilog`.

On startup, all existing log files where a DS has not been connected will be deleted. If there is less than 50 MB of free space on the target storage, `FRC_` log files are deleted (oldest to newest) until there is 50 MB free OR there are 10 files remaining.

The most basic usage of DataLogManager only requires a single line of code (typically this would be called from `Robot` constructor). This will record all NetworkTables changes to the data log.

JAVA

```java
import edu.wpi.first.wpilibj.DataLogManager;
// Starts recording to data log
DataLogManager.start();
```

C++

```cpp
#include "frc/DataLogManager.h"
// Starts recording to data log
frc::DataLogManager::Start();
```

PYTHON

```python
from wpilib import DataLogManager
DataLogManager.start()
```

DataLogManager provides a convenience function (`DataLogManager.log()`) for logging of text messages to the `messages` entry in the data log. The message is also printed to standard output, so this can be a replacement for `System.out.println()`.

DataLogManager also records the current roboRIO system time (in UTC) to the data log every ~5 seconds to the `systemTime` entry in the data log. This can be used to (roughly) synchronize the data log with other records such as DS logs or match video.

For custom logging, the managed `DataLog` can be accessed via `DataLogManager.getLog()`.

Logging Joystick Data

DataLogManager by default does not record joystick data. The `DriverStation` class provides support for logging of DS control and joystick data via the `startDataLog()` function:

JAVA

```java
import edu.wpi.first.wpilibj.DataLogManager;
import edu.wpi.first.wpilibj.DriverStation;
// Starts recording to data log
DataLogManager.start();
// Record both DS control and joystick data
DriverStation.startDataLog(DataLogManager.getLog());
// (alternatively) Record only DS control data
DriverStation.startDataLog(DataLogManager.getLog(), false);
```

C++

```cpp
#include "frc/DataLogManager.h"
#include "frc/DriverStation.h"
// Starts recording to data log
frc::DataLogManager::Start();
// Record both DS control and joystick data
DriverStation::StartDataLog(DataLogManager::GetLog());
// (alternatively) Record only DS control data
DriverStation::StartDataLog(DataLogManager::GetLog(), false);
```

PYTHON

```python
from wpilib import DataLogManager, DriverStation
# Starts recording to data log
DataLogManager.start()
# Record both DS control and joystick data
DriverStation.startDataLog(DataLogManager.getLog())
# (alternatively) Record only DS control data
DriverStation.startDataLog(DataLogManager.getLog(), False)
```

12.5.3 Custom Data Logging using DataLog

The `DataLog` class (Java, C++, Python) and its associated LogEntry classes (e.g. `BooleanLogEntry`, `DoubleLogEntry`, etc) provides low-level access for writing data logs.

Note

Unlike NetworkTables, there is no change checking performed. **Every** call to a `LogEntry.append()` function will result in a record being written to the data log. Checking for changes and only appending to the log when necessary is the responsibility of the caller.

The LogEntry classes can be used in conjunction with DataLogManager to record values only to a data log and not to NetworkTables:

JAVA

```
import edu.wpi.first.util.datalog.BooleanLogEntry;
import edu.wpi.first.util.datalog.DataLog;
import edu.wpi.first.util.datalog.DoubleLogEntry;
import edu.wpi.first.util.datalog.StringLogEntry;
import edu.wpi.first.wpilibj.DataLogManager;
BooleanLogEntry myBooleanLog;
DoubleLogEntry myDoubleLog;
StringLogEntry myStringLog;

public Robot() {
  // Starts recording to data log
  DataLogManager.start();
  // Set up custom log entries
  DataLog log = DataLogManager.getLog();
  myBooleanLog = new BooleanLogEntry(log, "/my/boolean");
  myDoubleLog = new DoubleLogEntry(log, "/my/double");
  myStringLog = new StringLogEntry(log, "/my/string");
}
public void teleopPeriodic() {
  if (...) {
    // Only log when necessary
    myBooleanLog.append(true);
    myDoubleLog.append(3.5);
    myStringLog.append("wow!");
  }
}
```

C++

```
#include "frc/DataLogManager.h"
#include "wpi/DataLog.h"
wpi::log::BooleanLogEntry myBooleanLog;
wpi::log::DoubleLogEntry myDoubleLog;
wpi::log::StringLogEntry myStringLog;
Robot() {
  // Starts recording to data log
  frc::DataLogManager::Start();
  // Set up custom log entries
  wpi::log::DataLog& log = frc::DataLogManager::GetLog();
  myBooleanLog = wpi::log::BooleanLogEntry(log, "/my/boolean");
  myDoubleLog = wpi::log::DoubleLogEntry(log, "/my/double");
  myStringLog = wpi::log::StringLogEntry(log, "/my/string");
}
void TeleopPeriodic() {
  if (...) {
    // Only log when necessary
    myBooleanLog.Append(true);
    myDoubleLog.Append(3.5);
    myStringLog.Append("wow!");
```

(continues on next page)

(continued from previous page)

```
  }
}
```

PYTHON

```
 from wpilib import DataLogManager, TimedRobot
 from wpiutil.log import (
     DataLog,
     BooleanLogEntry,
     DoubleLogEntry,
     StringLogEntry,
 )
class MyRobot(TimedRobot):
     def robotInit(self):
         # Starts recording to data log
         DataLogManager.start()
         # Set up custom log entries
         log = DataLogManager.getLog()
         self.myBooleanLog = BooleanLogEntry(log, "/my/boolean")
         self.myDoubleLog = DoubleLogEntry(log, "/my/double")
         self.myStringLog = StringLogEntry(log, "/my/string")
     def teleopPeriodic(self):
         if ...:
             # Only log when necessary
             self.myBooleanLog.append(True)
             self.myDoubleLog.append(3.5)
             self.myStringLog.append("wow!")
```

12.6 Downloading & Processing Data Logs

Data logs can be processed and viewed offline by AdvantageScope, the DataLogTool, or custom utilities. If data log files are being stored to the roboRIO integrated flash memory instead of a removable USB flash drive, it's important to periodically download and delete data logs to avoid the storage from filling up.

12.6.1 Managing Data Logs with AdvantageScope

AdvantageScope is an analysis tool that supports downloading, visualizing, and exporting data logs. See the relevant sections of the documentation for more details:

- Downloading log files
- Exporting to new formats

12.6.2 Managing Data Logs with the DataLogTool

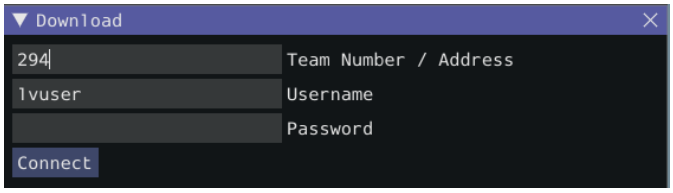
The DataLogTool desktop application integrates a SFTP client for downloading data log files from a network device (e.g. roboRIO or coprocessor) to the local computer.

This process consists of four steps:

1. Connect to roboRIO or coprocessor
2. Navigate to remote directory and select what files to download
3. Select download folder
4. Download files and optionally delete remote files after downloading

Connecting to RoboRIO

> **Note**
>
> The downloader uses SSH, so it will not be able to connect wirelessly if the radio firewall is enabled (e.g. when the robot is on the competition field).



Either a team number, IP address, or hostname can be entered into the *Team Number / Address* field. This field specifies the remote host to connect to. If a team number is entered, `roborio-TEAM-frc.local` is used as the connection address.

The remote username and password are also entered here. For the roboRIO, the username should be `lvuser` with a blank password.

The tool also supports connecting to network devices other than the roboRIO, such as coprocessors, as long as the device supports SFTP password-based authentication.

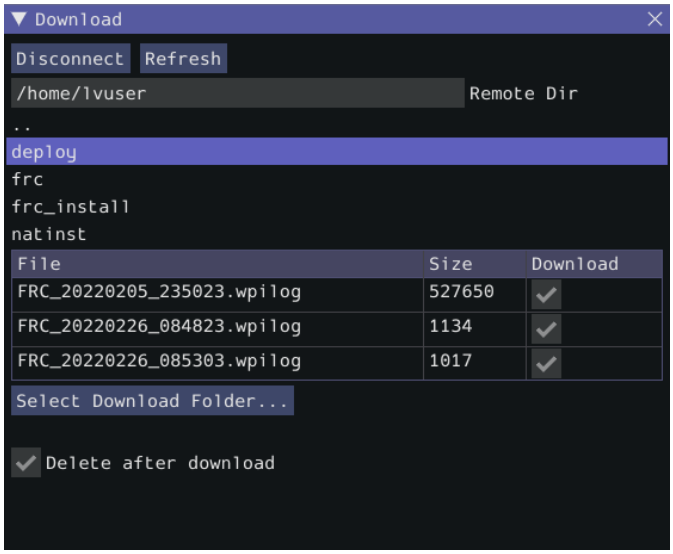
Click *Connect* to connect to the remote device. This will attempt to connect to the device. The connection attempt can be aborted at any time by clicking *Disconnect*. If the application is unable to connect to the remote device, an error will be displayed above the *Team Number / Address* field and a new connection can be attempted.

Downloading Files

After the connection is successfully established, a simplified file browser will be displayed. This is used to navigate the remote filesystem and select which files to download. The first text box shows the current directory. A specific directory can be navigated to by typing it in this text box and pressing Enter. Alternatively, directory navigation can be performed by clicking on one of the directories that are listed below the remote dir textbox. Following the list of directories is a table of files. Only files with a `.wpilog` extension are shown, so the table will be empty if there are no log files in the current directory. The checkbox next to each data log file indicates whether the file should be downloaded.

Note

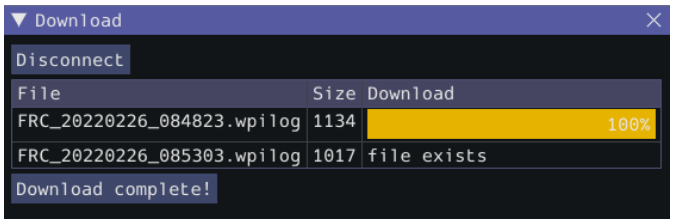
On the roboRIO, log files are typically saved to either `/home/lvuser/logs` or `/u/logs` (USB stick location).



Click *Select Download Folder...* to bring up a file browser for the local computer.

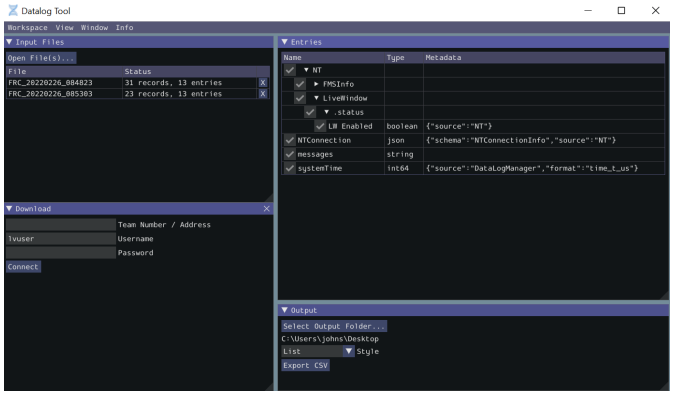
If you want to delete the files from the remote device after they are downloaded, check the *Delete after download* checkbox.

Once a download folder is selected, *Download* will appear. After clicking this button, the display will change to a download progress display. Any errors will be shown next to each file. Click *Download complete!* to return to the file browser.



Converting Data Logs to CSV

As data logs are binary files, the DataLogTool desktop application provides functionality to convert data logs into CSV files for further processing or analysis. Multiple data logs may be simultaneously loaded into the tool for batch processing, and partial data exports can be performed by selecting only the data that is desired to be output.

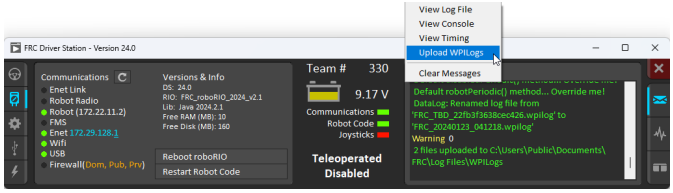


The conversion process is started by opening data log files in the "Input Files" window. Files are opened by clicking *Open File(s)...*. Summary status on each file (e.g. number of records and entries) is displayed. Clicking *X* in the table row closes the file.

After at least one file is loaded, the "Entries" window displays a tree view of the entries (this can be changed to a flat view by right clicking on the "Entries" window title bar and unchecking *Tree View*). Individual entries or entire subtrees can be checked or unchecked to indicate whether they should be included in the export. The data type information and initial metadata for each entry is also shown in the table. As the "Entries" view shows a merged view of all entries across all input files, if more than one input file is open, hovering over an entry's name will highlight what input files contain that entry.

The output window is used to specify the output folder (via *Select Output Folder...*) as well as the output style (list or table). The list output style outputs a CSV file with 3 columns (timestamp, entry name, and value) and a row for every value change (for every exported entry). The table output style outputs a CSV file with a timestamp column and a column for every exported entry; a row is output for every value change (for every exported entry), but the value is placed in the correct column for that entry. Clicking *Export CSV* will create a `.csv` file in the output folder corresponding to each input file.

12.6.3 Managing Data Logs with the Driver Station



The Driver Station software can download WPILogs. Click on the gear icon and select *Upload WPILogs*. The logs in `/home/lvuser/logs` or `/u/logs` will be downloaded automatically to `C:\Users\Public\Documents\FRC\Log Files\WPILogs`

12.6.4 Custom Processing of Data Logs

For more advanced processing of data logs (e.g. for processing of binary values that can't be converted to CSV), WPILib provides a `DataLogReader` class for reading data logs in Java, C++, or Python. There is also a pure python datalog reader (datalog.py). For other languages, the data log format is also documented.

DataLogReader provides a low-level view of a data log, in that it supports iterating over a data log's control and data records and decoding of common data types, but does not provide any higher level abstractions such as a NetworkTables-like map of entries. The printlog example in Java and C++ (and the Python `datalog.py`) demonstrates basic usage.

12.7 Writing Your Own Sendable Classes

Since the `Sendable` interface only has one method, writing your own classes that implement `Sendable` (and thus automatically log values to and/or consume values from the dashboard) is extremely easy: just provide an implementation for the overridable `initSendable` method, in which setters and getters for your class's fields are declaratively bound to key values (their display names on the dashboard).

For example, here is the implementation of `initSendable` from WPILib's `BangBangController`:

JAVA

```
151   @Override
152   public void initSendable(SendableBuilder builder) {
153     builder.setSmartDashboardType("BangBangController");
154     builder.addDoubleProperty("tolerance", this::getTolerance, this::setTolerance);
155     builder.addDoubleProperty("setpoint", this::getSetpoint, this::setSetpoint);
156     builder.addDoubleProperty("measurement", this::getMeasurement, null);
157     builder.addDoubleProperty("error", this::getError, null);
158     builder.addBooleanProperty("atSetpoint", this::atSetpoint, null);
159   }
```

C++

```
55  void BangBangController::InitSendable(wpi::SendableBuilder& builder) {
56    builder.SetSmartDashboardType("BangBangController");
57    builder.AddDoubleProperty(
58        "tolerance", [this] { return GetTolerance(); },
59        [this](double tolerance) { SetTolerance(tolerance); });
60    builder.AddDoubleProperty(
61        "setpoint", [this] { return GetSetpoint(); },
62        [this](double setpoint) { SetSetpoint(setpoint); });
63    builder.AddDoubleProperty(
64        "measurement", [this] { return GetMeasurement(); }, nullptr);
65    builder.AddDoubleProperty(
66        "error", [this] { return GetError(); }, nullptr);
67    builder.AddBooleanProperty(
```

(continues on next page)

(continued from previous page)

```
68        "atSetpoint", [this] { return AtSetpoint(); }, nullptr);
69  }
```

To enable the automatic updating of values by WPILib "in the background", `Sendable` data names are bound to getter and setter functions rather than specific data values. If a field that you wish to log has no defined setters and getters, they can be defined inline with a lambda expression.

12.7.1 The SendableBuilder Class

As seen above, the `initSendable` method takes a single parameter, `builder`, of type `SendableBuilder` (Java, C++, Python). This builder exposes methods that allow binding of getters and setters to dashboard names, as well as methods for safely ensuring that values consumed *from* the dashboard do not cause unsafe robot behavior.

Databinding with addProperty Methods

Like all WPILib dashboard code, `Sendable` fields are ultimately transmitted over *NetworkTables*, and thus the databinding methods provided by `SendableBuilder` match the supported NetworkTables data types:

- `boolean`: `addBooleanProperty`
- `boolean[]`: `addBooleanArrayProperty`
- `double`: `addDoubleProperty`
- `double[]`: `addDoubleArrayProperty`
- `string`: `addStringProperty`
- `string[]`: `addStringArrayProperty`
- `byte[]`: `addRawProperty`

Ensuring Safety with setSafeState and setActuator

Since `Sendable` allows users to consume arbitrary values from the dashboard, it is possible for users to pipe dashboard controls directly to robot actuations. This is extremely unsafe if not done with care; dashboards are not a particularly good interface for controlling robot movement, and users generally do not expect the robot to move in response to a change on the dashboard.

To help users ensure safety when interfacing with dashboard values, `SendableBuilder` exposes a `setSafeState` method, which is called to place any `Sendable` mechanism that actuates based on dashboard input into a safe state. Any potentially hazardous user-written `Sendable` implementation should call `setSafeState` with a suitable safe state implementation. For example, here is the implementation from the WPILib `PWMMotorController` class:

JAVA

```
120   @Override
121   public void initSendable(SendableBuilder builder) {
122     builder.setSmartDashboardType("Motor Controller");
123     builder.setActuator(true);
124     builder.setSafeState(this::disable);
125     builder.addDoubleProperty("Value", this::get, this::set);
126   }
```

C++

```
56  void PWMMotorController::InitSendable(wpi::SendableBuilder& builder) {
57    builder.SetSmartDashboardType("Motor Controller");
58    builder.SetActuator(true);
59    builder.SetSafeState([=, this] { Disable(); });
60    builder.AddDoubleProperty(
61        "Value", [=, this] { return Get(); },
62        [=, this](double value) { Set(value); });
```

Additionally, users may call `builder.setActuator(true)` to mark any mechanism that might move as a result of `Sendable` input as an actuator. Currently, this is used by *Shuffleboard* to disable actuator widgets when not in *LiveWindow* mode.

12.8 Third-Party Telemetry Libraries

> **Tip**
>
> Is your library not listed here when it should be? Open a pull request to add it!

Several third-party logging utilities and frameworks exist that provide functionality beyond what is currently provided by WPILib:

- AdvantageKit (Java only): "Log everything"-based logging framework with hooks for replaying logged data in *simulation*.
- DSLOG: An alternate driver station log viewer.
- DogLog (Java only): A simple logging library for Java.

13 FRC LabVIEW Programming

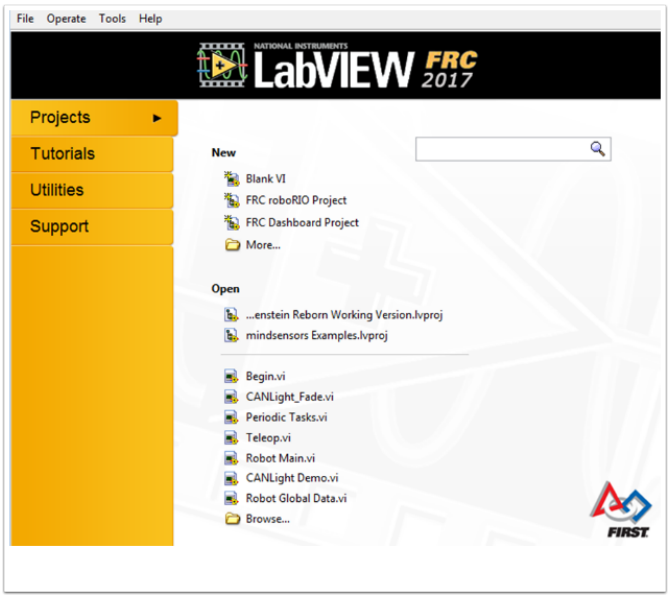
13.1 Creating Robot Programs

13.1.1 Tank Drive Tutorial

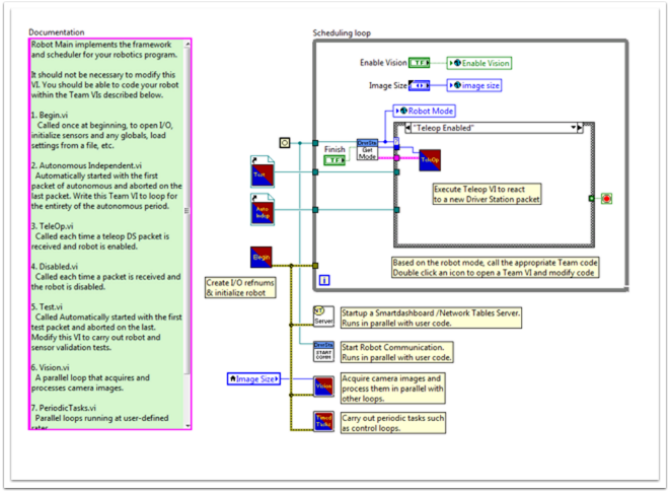
Question: How do I get my robot to drive with two joysticks using tank drive?

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`.



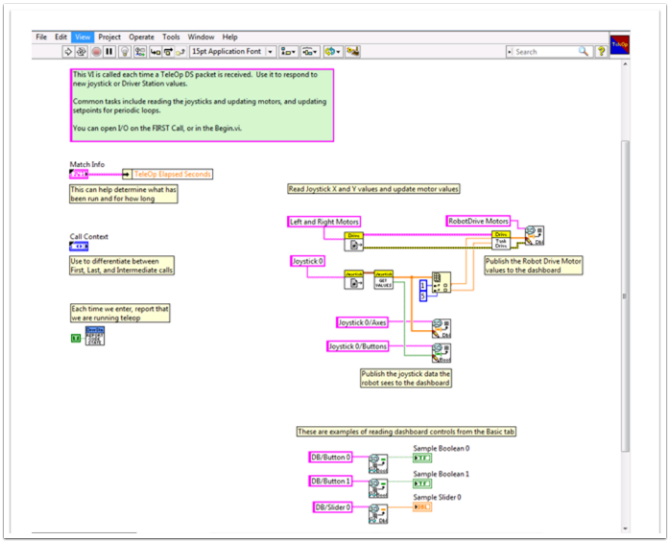
2. Give your project a name, add your team number, and select `Arcade Drive Robot roboRIO`. You can select another option, however, this tutorial will discuss how to setup tank drive for this project.
3. In the Project Explorer window, open up the `Robot Main.vi`.
4. Push `Ctrl+E` to see the block diagram.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

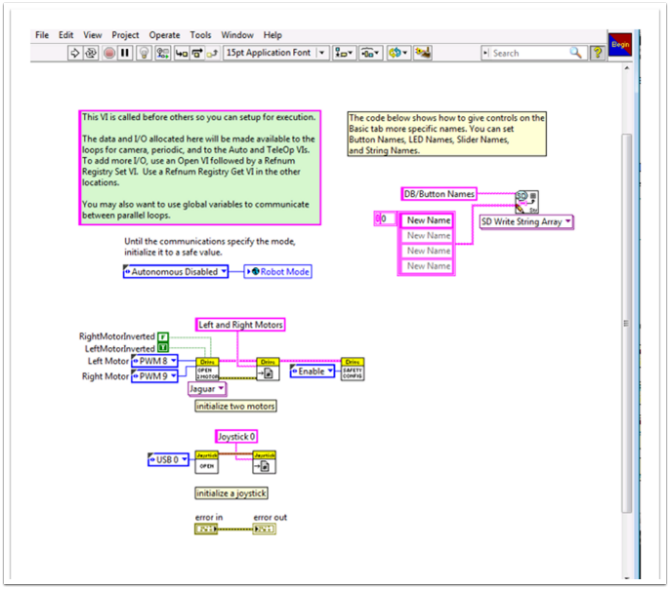
- Replace Arcade Drive with the tank drive.vi. This can be found by right clicking on the block diagram >> WPI Robotics Library >> Robot Drive >> and clicking the Tank Drive VI.
- Find the Index Array function that is after the Get Values.vi. You will need to create two numeric constants and wire each 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, Victor, Spark, etc.).

For example, the image below shows motors wired into PWM 8 and 9:



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

13.1.2 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?

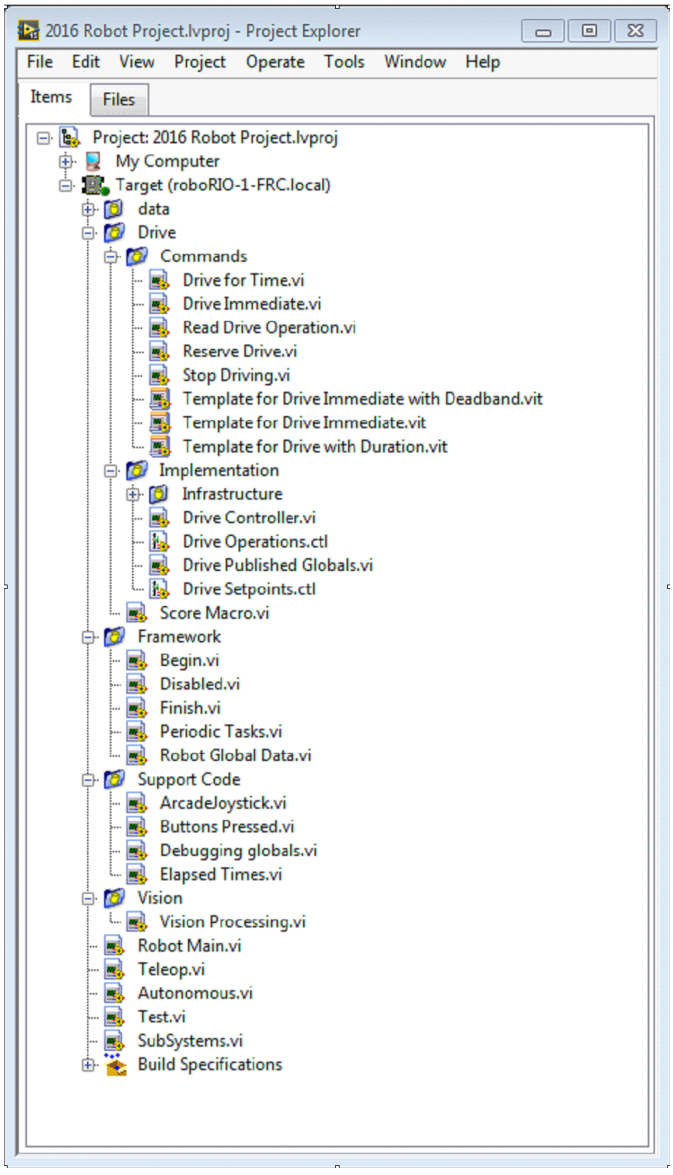
Command and Control adds functionality to the existing LabVIEW project templates, allowing code to scale better with more sophisticated robots and robot code. Subsystems are used to abstract the details of the implementation, and game code is built from sequences of high level command VIs. The commands 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.

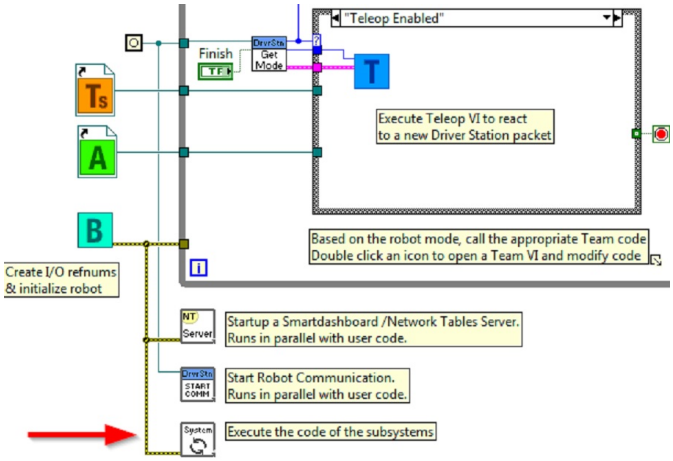
Test.vi
VI that runs when driver station is in test mode.

SubSystems.vi
VI that contains and starts all subsystems.

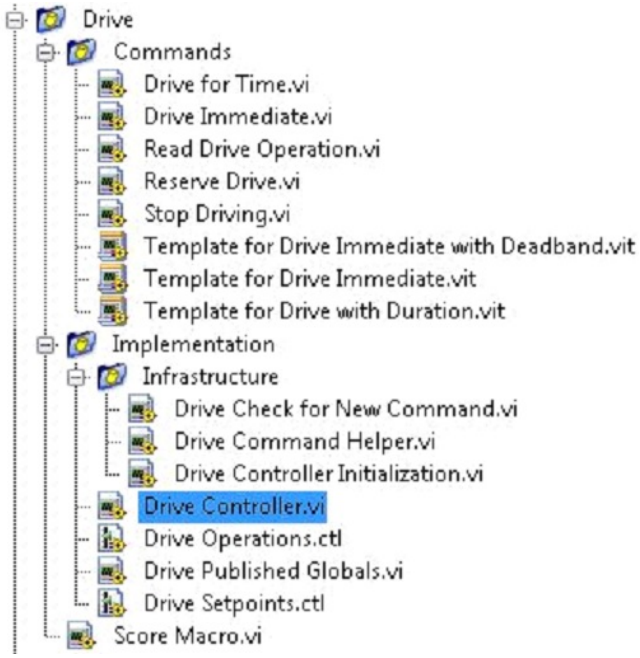
Dependencies
Shows the files used by the robot code.

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

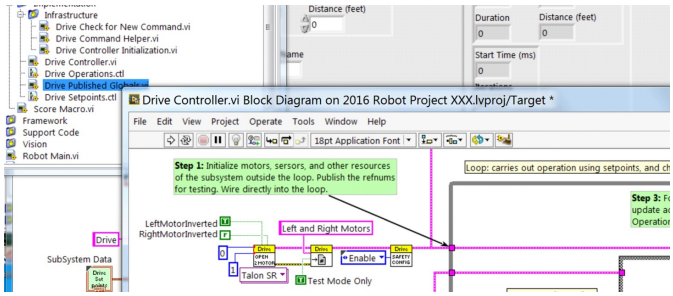
- Drive Check for New Command: It is called each iteration of the controller loop. It checks for new commands, updates timing data, and upon completion notifies a waiting command.
- Drive Command Helper.vi: Commands call this VI to notify the controller that a new command has been issued.
- Drive Controller 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

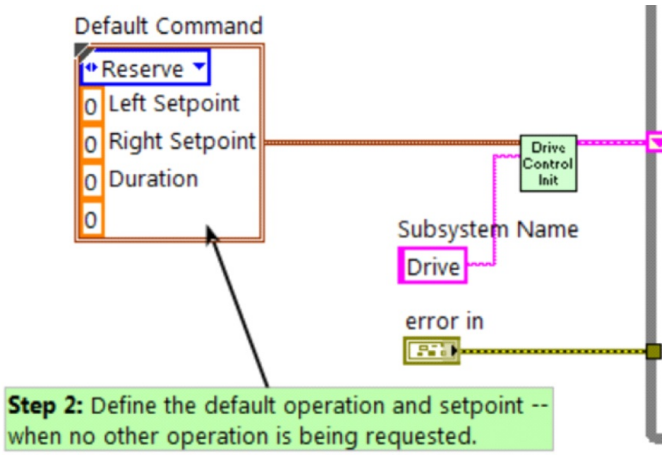
There are green comments on the controller's block diagram that point out key areas that you will want to know how to edit.

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 refnums, or you may register them for Test Mode Only to keep them private so that other code cannot update motors without using a command.

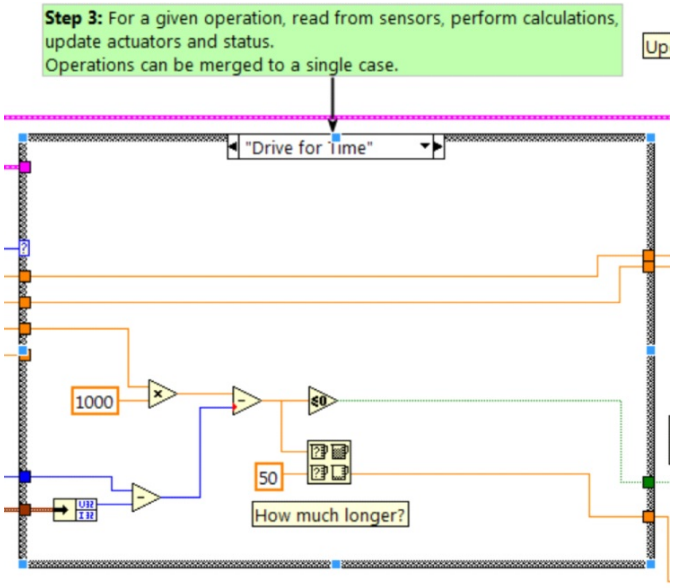


Note

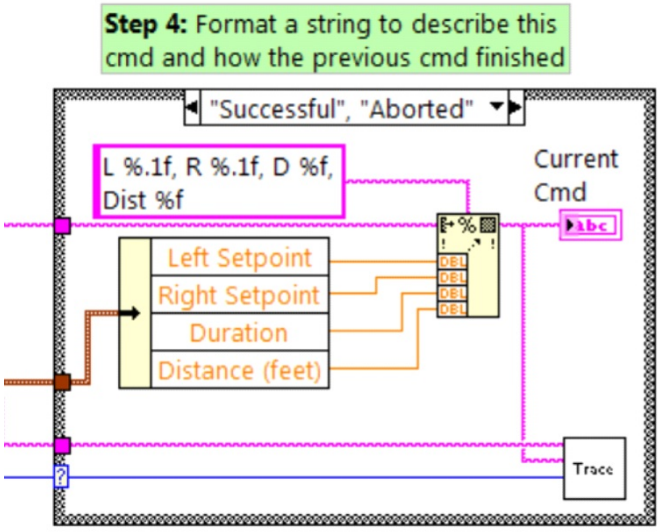
Initializing the resources for each subsystem in their respective Controller.vi rather than in Begin.vi improves I/O encapsulation, reducing potential resource conflicts and simplifies debugging.



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

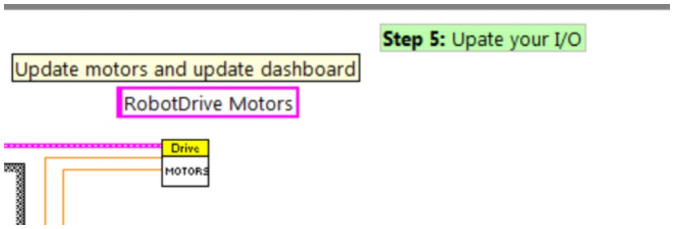


Inside the control loop is a case statement where operations are actually implemented. Set point values, iteration delay, iteration count, and sensors can all have influence on how the subsystem operates. This case structure has a value for each operation state of the subsystem.



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.



Part 3: Drive Subsystem Shipped Commands

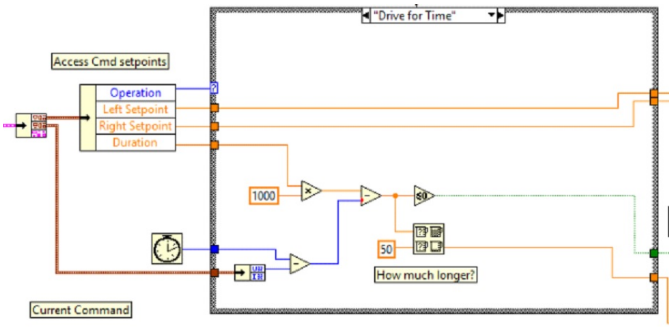
There are 3 shipped example commands for each new subsystem:

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.

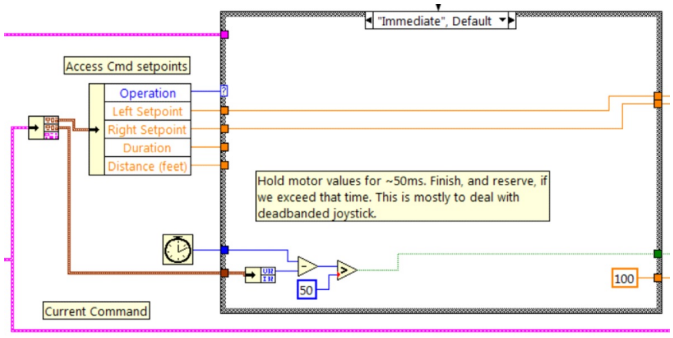


Drive Immediate.vi



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.

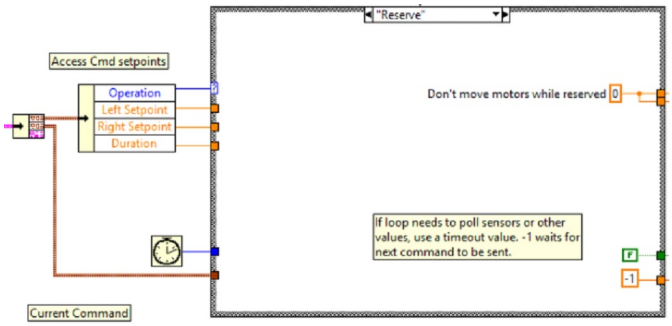


Stop Driving.vi



Zero the drive motors, making the robot stationary.

The Reserve command turns off the motors and waits for a new command. When used with a named command sequence, reserve identifies that the drive subsystem is part of a sequence, even if not currently moving the robot. This helps to arbitrate subsystem resource between simultaneously running commands.



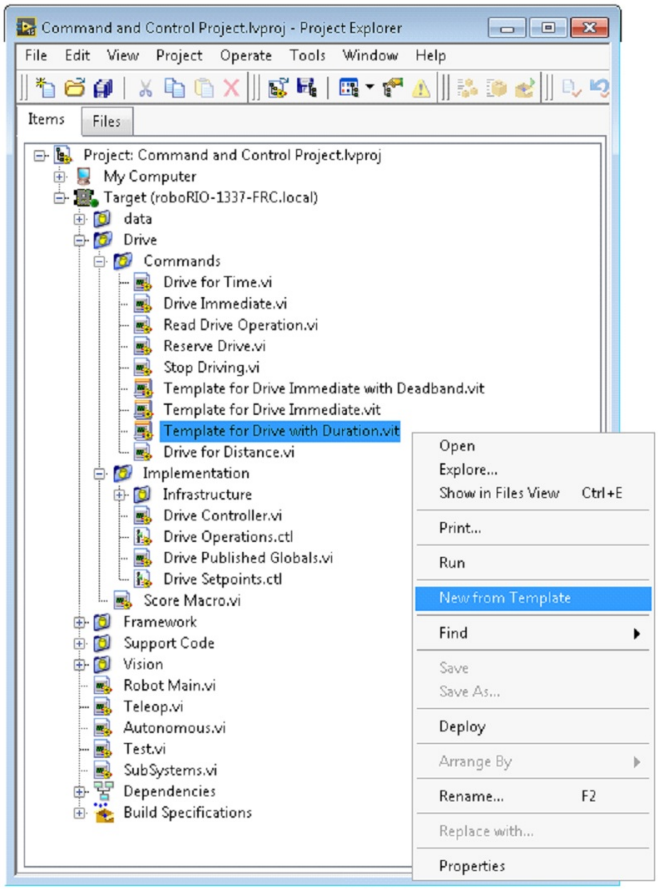
Part 4: Creating New Commands

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.

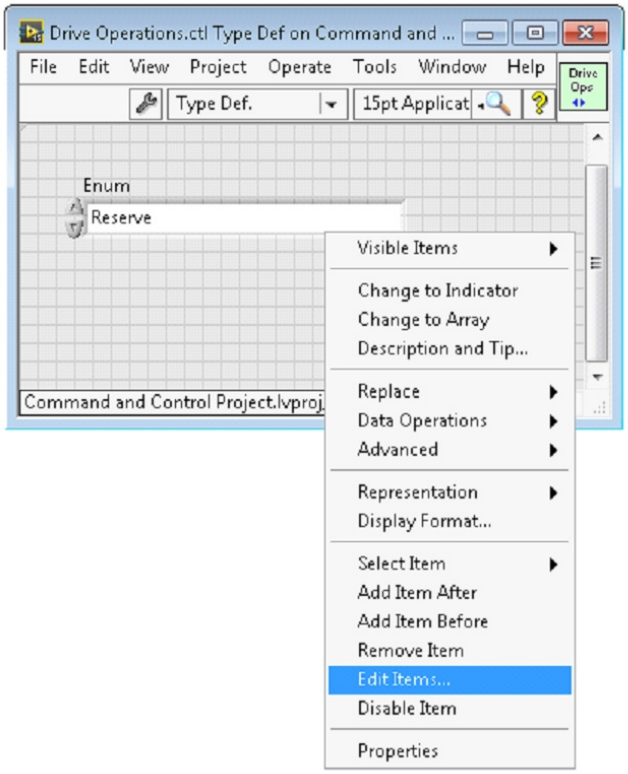
- **Immediate**: This VI notifies the subsystem about the new setpoint.
- **Immediate with deadband**: This VI compares the input value to the deadband and optionally notifies the subsystem about the new setpoint. This is very useful when joystick continuous values are being used.

- **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.

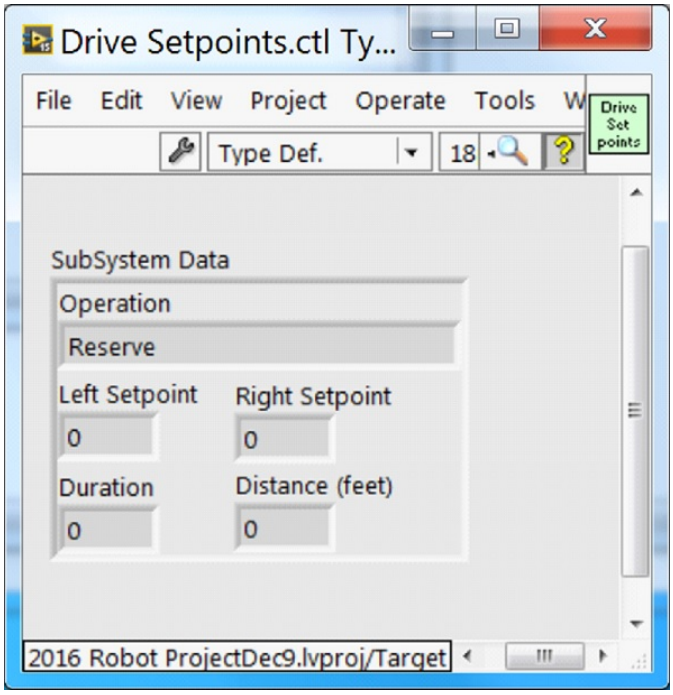
In this example we will add the new command “Drive for Distance”.



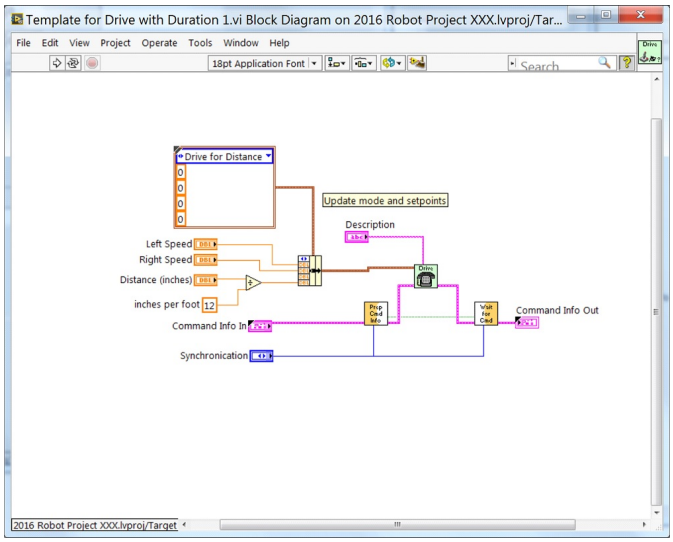
First, save the new VI with a descriptive name such as “Drive for Distance”. Next, determine whether the new command needs a new value added the Drive Operations enum typedef. The initial project code already has an enum value of Drive for Distance, but the following image shows how you would add one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



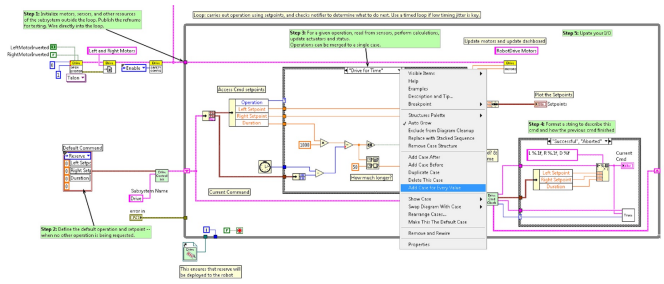
If a command needs additional information to execute, add it to the setpoints control. By default, the Drive subsystem has fields for the Left Setpoint, Right Setpoint, and Duration along with the operation to be executed. The Drive for Distance command could reuse Duration as distance, but let's go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



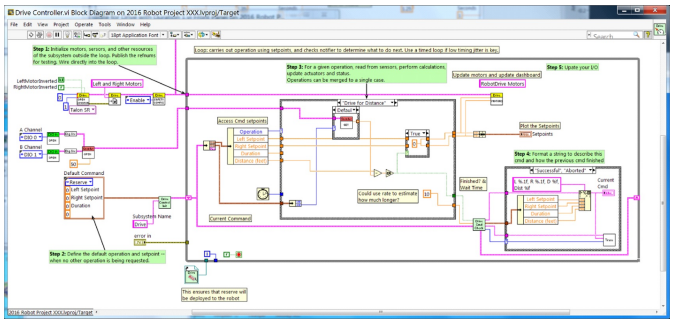
Once that we have all of the fields needed to specify our command, we can modify the newly created Drive for Distance.vi. As shown below, select Drive for Distance from the enum's drop down menu and add a VI parameters to specify distance, speeds, etc. If the units do not match, the command VI is a great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

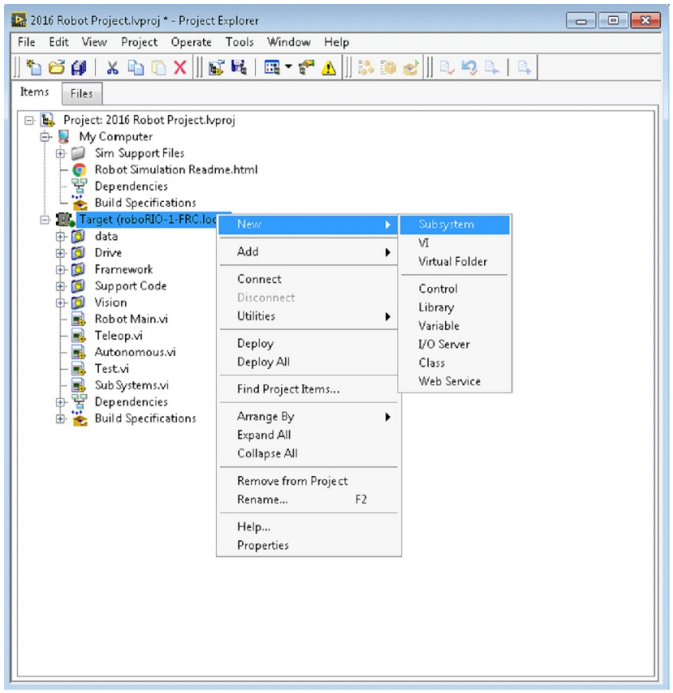


In order to access new setpoint fields, grow the "Access Cmd setpoints" unbundle node. Open your encoder(s) on the outside, to the left of the loop. In the new diagram of the case structure, we added a call to reset the encoder on the first loop iteration and read it otherwise. There is also some simple 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.

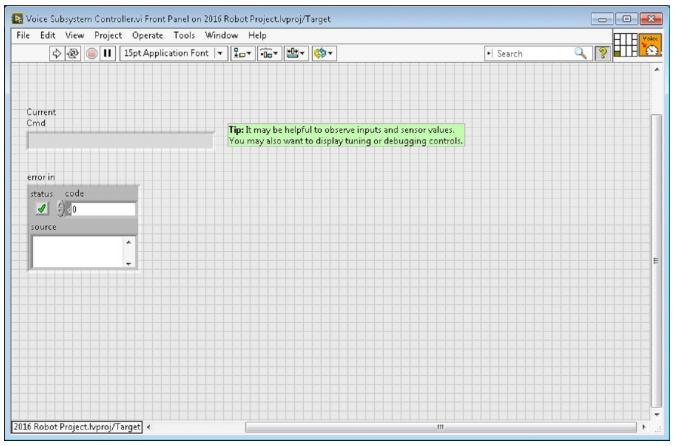


Part 5: Creating a Subsystem

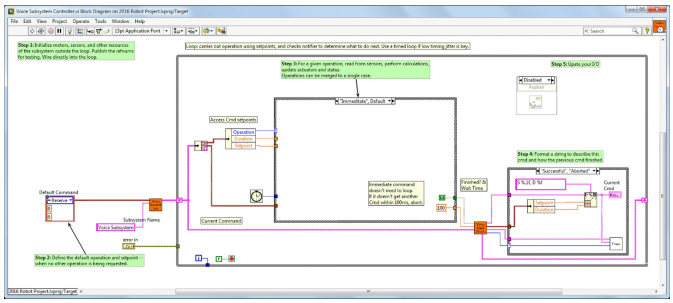
In order to create a new subsystem, right click on the roboRIO target and select New» Subsystem. In the pop up dialog box, enter the name of the subsystem, list the operational modes, and specify the color of the icon.



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.



13.2 LabVIEW Resources

13.2.1 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.

Third Party Resources

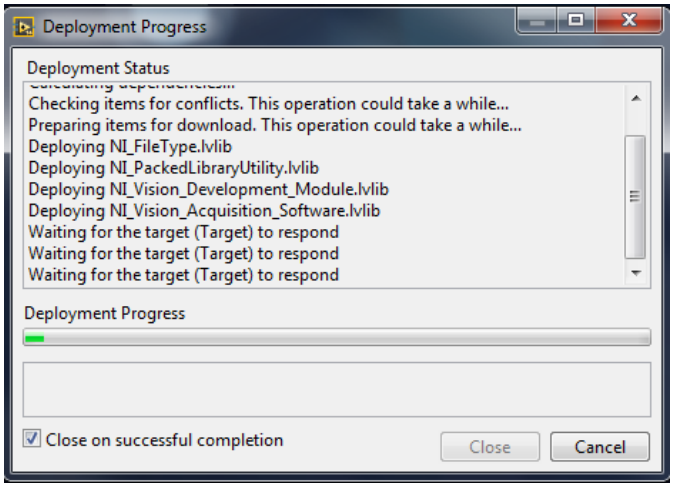
- FRC Control and Trajectory Library
- Secret Book Of FRC LabVIEW 2

13.2.2 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.

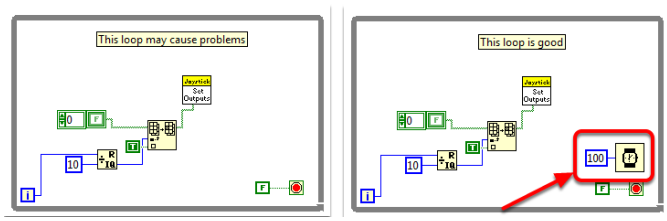
The Symptom



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.

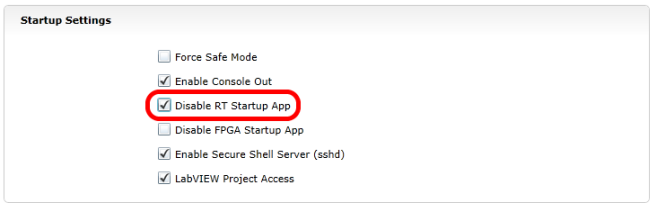
The Problem



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

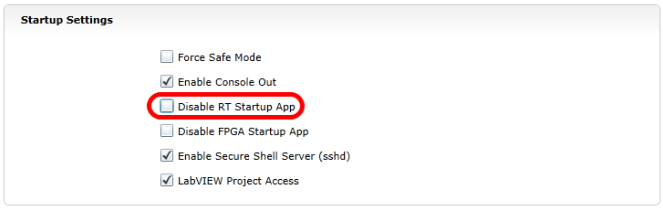


Using the roboRIO webserver (see the article on the *roboRIO Web Dashboard Startup Settings* for more details). **Check** the box to "Disable RT Startup App".

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



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.

13.2.3 How To Toggle Between Two Camera Modes

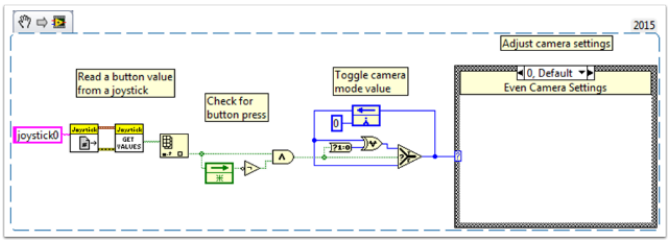
This code shows how to use a button to toggle between two distinct camera modes. The code consists of four stages.

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.



13.2.4 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

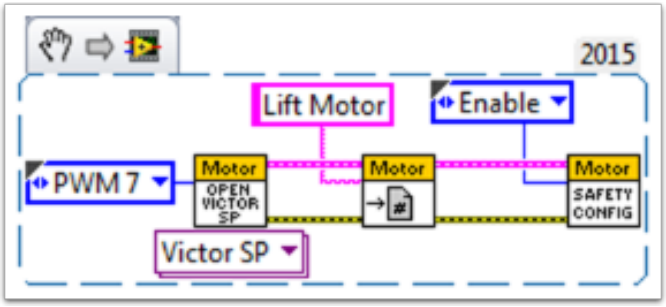
Looking for more examples and discussions? Search through more documents or post your own discussion, example code, or tutorial by clicking here! Don't forget to mark your posts with a tag!

13.2.5 Add an Independent Motor to a Project

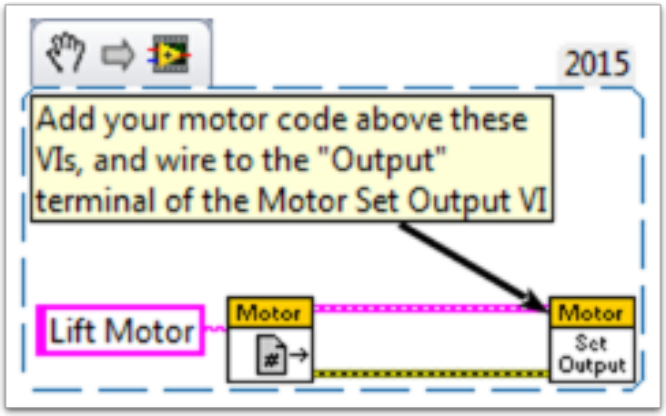
Once your drive that controls the wheels is all set, you might need to add an additional motor to control something completely independent of the wheels, such as an arm. Since this motor will not be part of your tank, arcade, or mecanum drive, you'll definitely want independent control of it.

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.

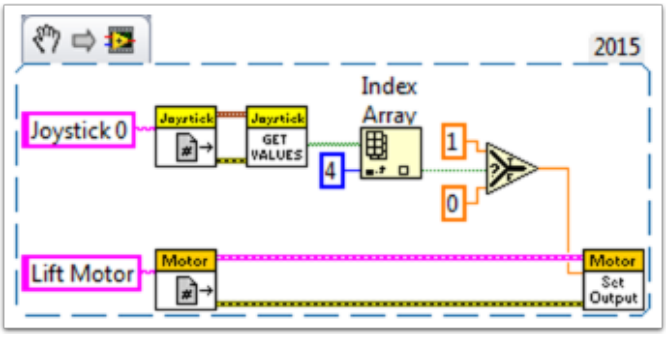
FIRST, create a motor reference in the **Begin.vi**, using the **Motor Control Open VI** and **Motor Control Refnum Registry Set VI**. These can be found by right-clicking in the block diagram and going to **WPI Robotics Library>>RobotDrive>>Motor Control**. Choose your *PWM* line and name your motor. 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.)



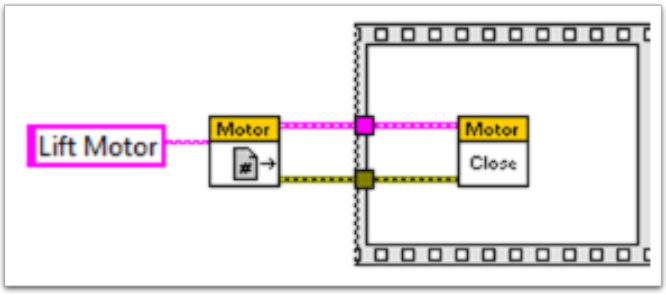
Now, reference your motor (the name has to be exact) in the **Teleop.vi** using the **Motor Control Refnum Registry Get VI** and tell it what to do with the **Motor Control Set Output VI**. These are in the same place as the above VIs.



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).

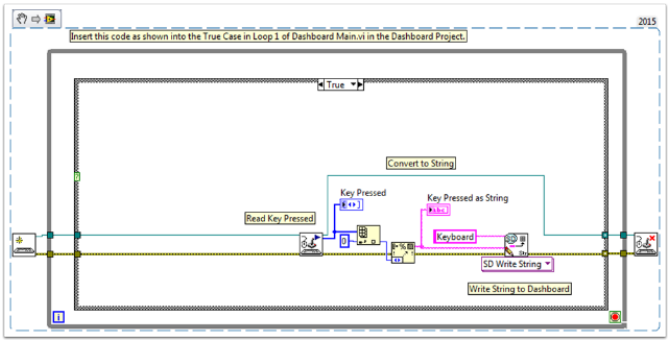


I hope this helps you program the best robot ever! Good luck!

13.2.6 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.

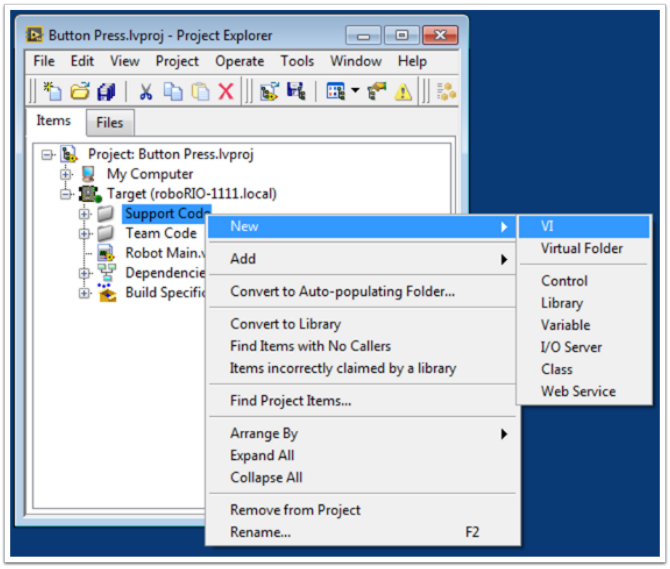


The second VI Snippet is code that should be included in the Teleop VI. This reads the string value from the Dashboard that indicates which key was pressed. A Case Structure then determines which values should be written to the left and right motors, depending on the key. In this case, W is forward, 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.

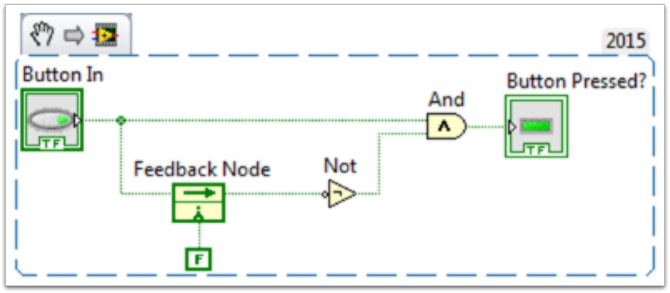
13.2.7 Making a One-Shot Button Press

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.



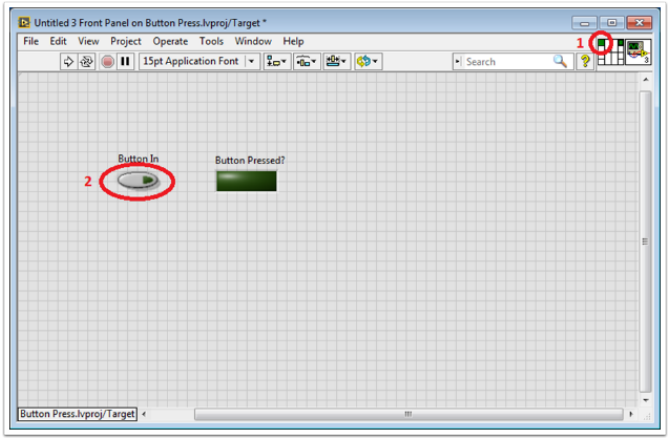
Now on the block diagram of the new VI, drop in the following code snippet.



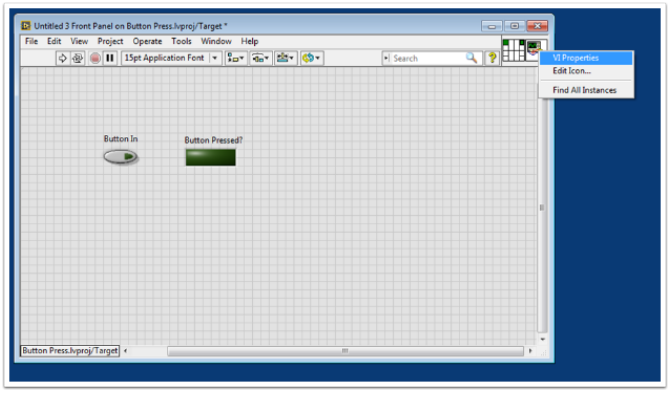
This code uses a function called the Feedback Node. We have wired the current value of the button into the left side of the feedback node. The wire coming out of the arrow of the feedback node represents the previous value of the button. If the arrow on your feedback node is going the opposite direction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

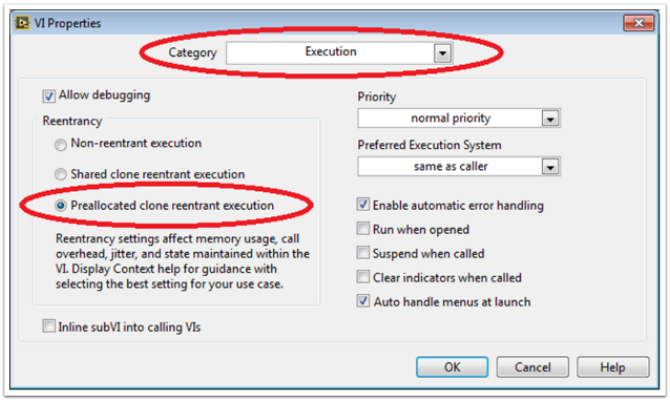
When a button is pressed, the value of the button goes from FALSE to TRUE. We want the output of this VI to be TRUE only when the current value of the button is TRUE, and the previous value of the button is FALSE.

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.

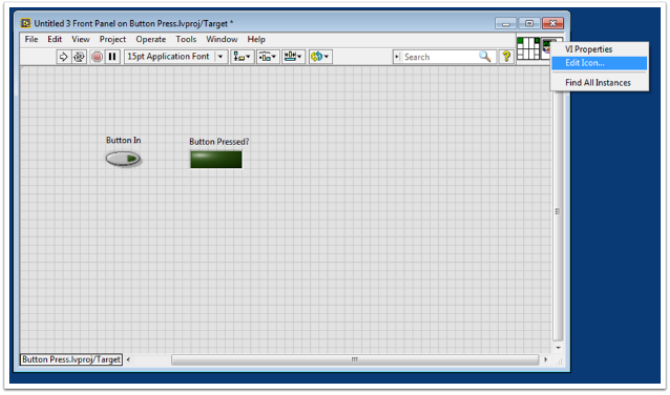


Next, we need to change the properties of this VI so that we can use multiples of this VI in our TeleOp.vi. Right click the VI Icon and go to VI Properties. Then select the category "Execution" and select "Preallocated clone reentrant execution".



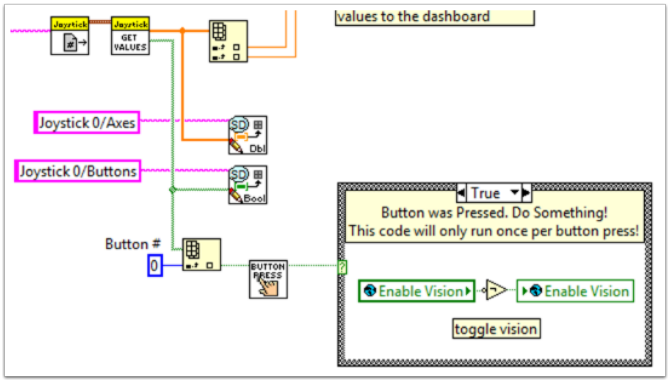


Lastly, we should change the VI Icon to be more descriptive of the VI's function. Right click the Icon and go to Edit Icon. Create a new Icon.



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.



13.2.8 Adding Safety Features to Your Robot Code

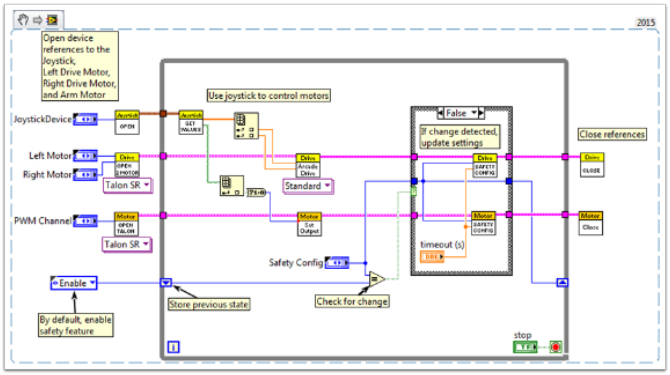
A common problem with complex projects is making sure that all of your code is executing when you expect it to. Problems can arise when tasks with high priority, long execution times, or frequent calls hog processing power on the roboRIO. This leads to what is known as "starvation" for the tasks that 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!

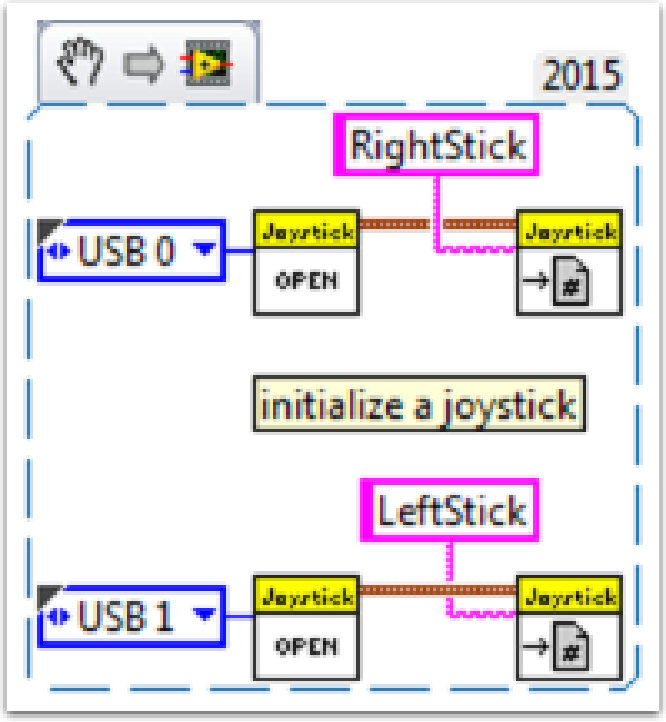
13.2.9 How to Use Joystick Buttons to Control Motors or Solenoids

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.

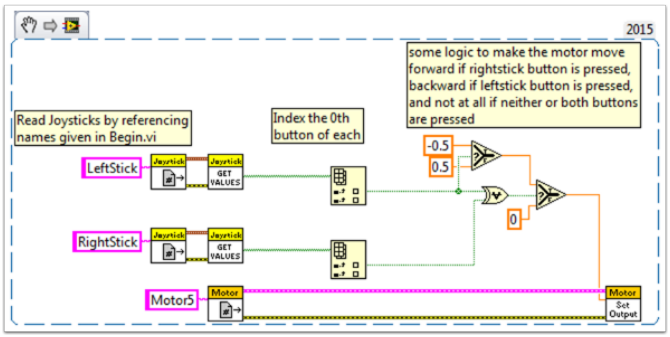


The rest of the code in this document will be placed in the "Teleop.VI" This is where we will be programming our joystick buttons to control different aspects of our motors or solenoids.

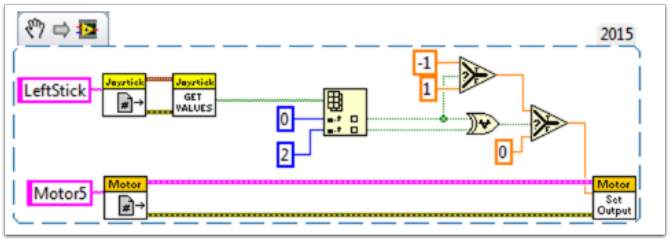
Scenario 1

"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:

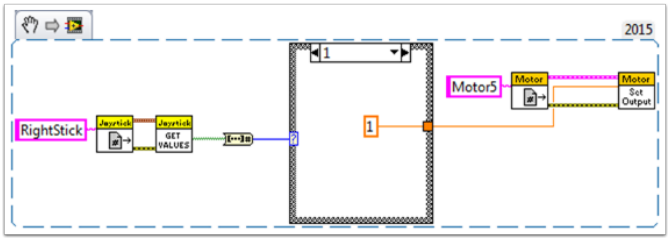
https://zone.ni.com/reference/en-XX/help/371361L-01/glang/case_structure/

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

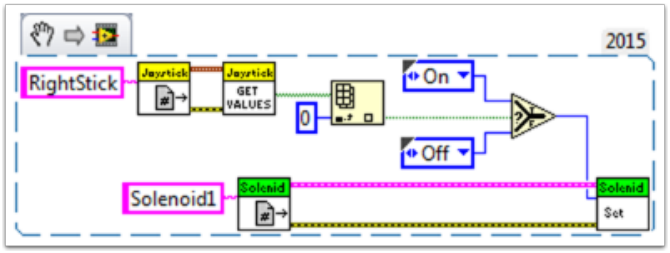


Since our requirements were simple, we only need a single constant in each case. For case 1 (full ahead) we use a 1, for case 2 (half back) we use a -0.5, etc. We can use any constant value between 1 and -1. I left case 0 as the default so if multiple buttons are pressed (any undefined state was reached) 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!

13.2.10 Local and Global Variables in LabVIEW for FRC

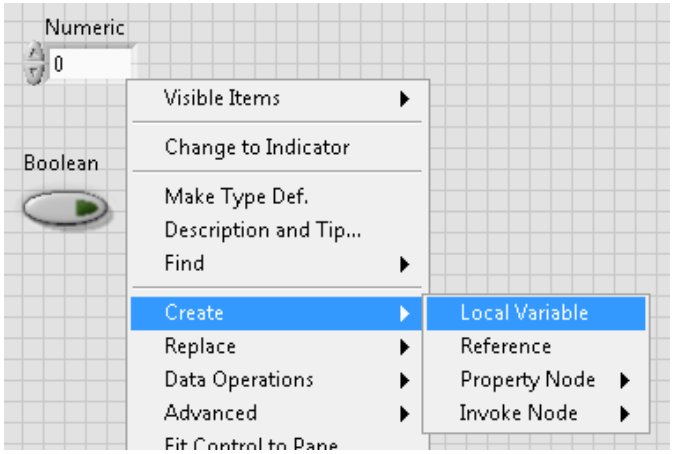
This example serves as an introduction to local and global variables, how they are used in the default LabVIEW for FRC® Robot Project, and how you might want to use them in your project.

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.

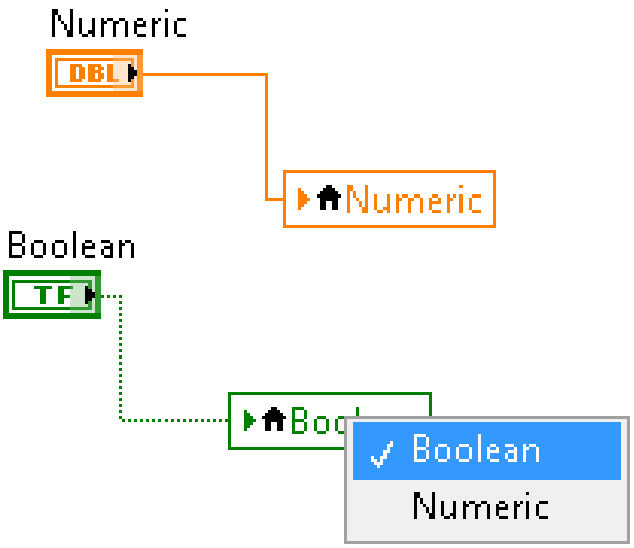
Note: One possible reason may be that you need to pass data between consecutive loop iterations; Miro_T covered this in this post. It should also be noted that the feedback node in LabVIEW may be used as an equivalent to the shift register, although that may be a topic for another day!

Introduction to Local and Global Variables

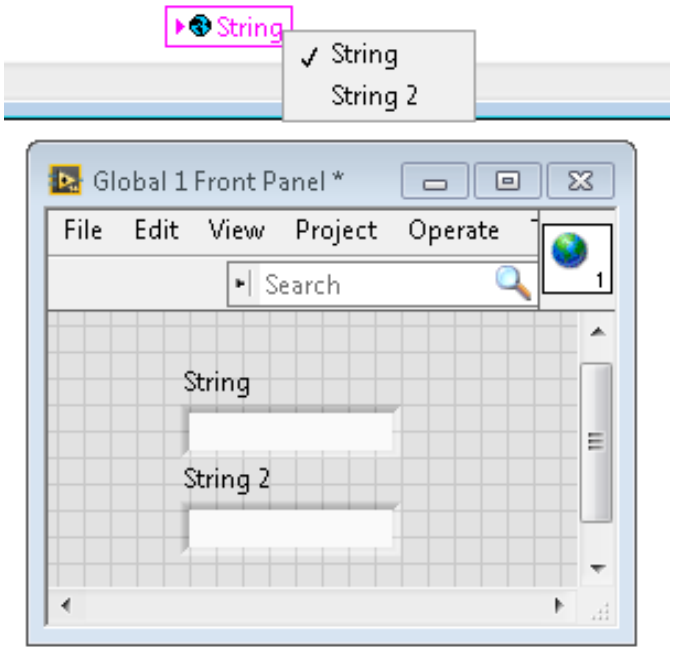
Local variables may be used within the same VI. Create a local variable by right-clicking a control or indicator on your Front Panel:



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:

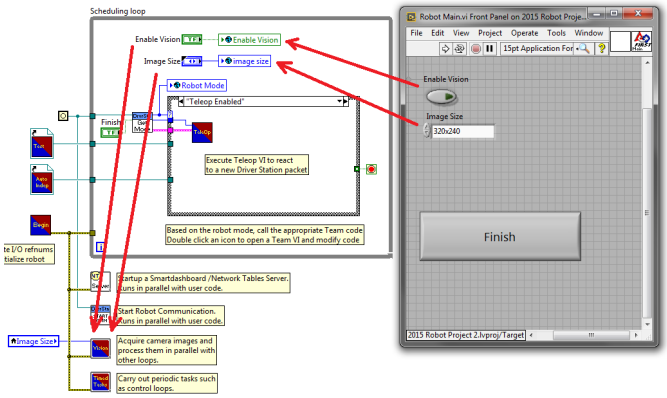


Note

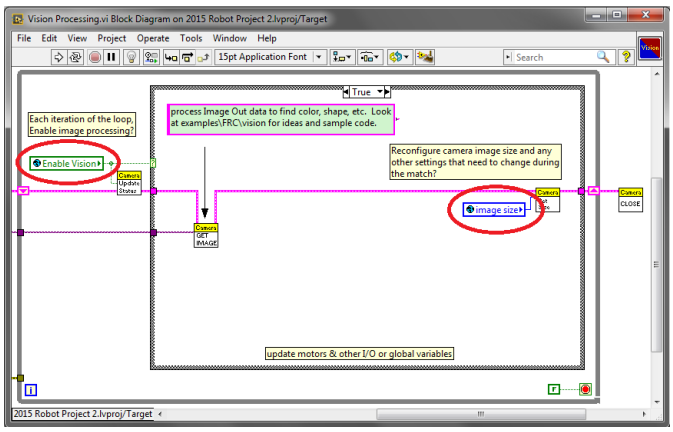
Be very careful to avoid race conditions when using local and global variables! Essentially, make sure that you are not accidentally writing to the same variable in multiple locations without a way to know to which location it was last written. For a more thorough explanation, see this help document

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...



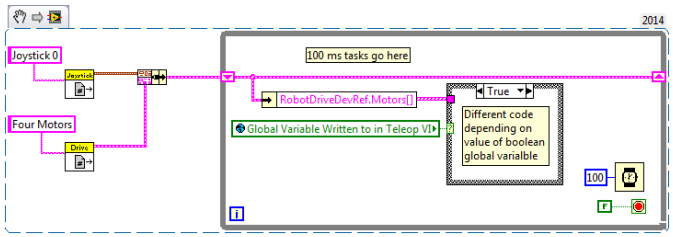
... And then read in each iteration of the Vision Processing VI:



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:



13.2.11 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:

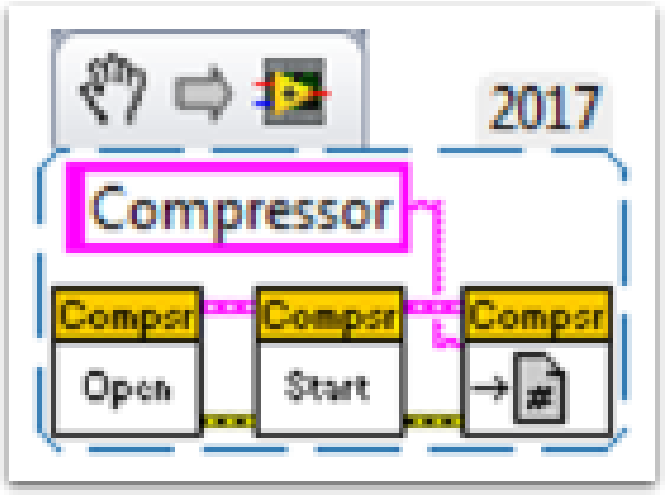
`FRC Pneumatics Manual`

PCM User's Guide

Pneumatics Step by Step for the roboRIO

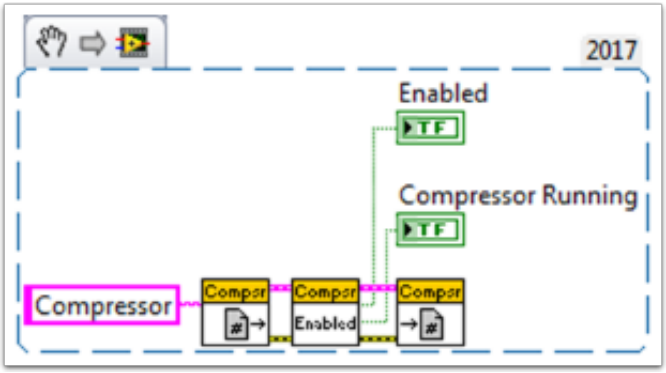
Begin VI

Place this snippet in the Begin.vi.



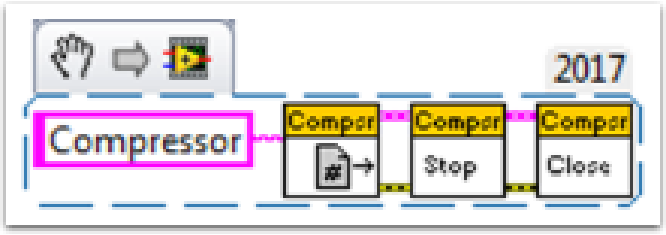
Teleop VI

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.



14 FRC Python Programming

14.1 pyproject.toml usage

> **Note**
>
> RobotPy projects are not required to have a `pyproject.toml`, but when you run `robotpy sync` one will automatically be created for you.

`pyproject.toml` has become a standard way to store build and tooling configuration for Python projects. The `[tool.XXX]` section(s) of the TOML file is a place where tools can store their configuration information.

Currently RobotPy only stores deployment related information in `pyproject.toml`, in the `[tool.robotpy]` section. Users can customize the other sections however they want, and `robotpy` will ignore them.

The `pyproject.toml` file looks something like this:

```
#
# Use this configuration file to control what RobotPy packages are installed
# on your RoboRIO
#
[tool.robotpy]
# Version of robotpy this project depends on
robotpy_version = "2026.1.1"
# Which extra RobotPy components should be installed
# -> equivalent to `pip install robotpy[extra1, ...]
components = [
    # "all",
    # "apriltag",
    # "commands2",
    # "cscore",
    # "romi",
    # "sim",
    # "xrp",
]
# Other pip packages to install
requires = []
```

Each of the following will instruct the deploy process to install packages to the roboRIO:

`robotpy_version` is the version of the `robotpy` PyPI package that this robot code depends on.

`components` defines extra RobotPy components that can be installed, as only the core RobotPy libraries are installed by default.

> **Note**
>
> `components` is equivalent to `robotpy_extras` in prior years

`requires` is a list of strings, and each item is equivalent to a line of a requirements.txt file. You can install any pure python packages on the roboRIO and they will likely work, but any packages that have binary dependencies must be cross-compiled for the roboRIO. For example, if you needed to use `numpy` in your robot code:

```
[tool.robotpy]
...
requires = ["numpy"]
```

Binary packages that can be installed are stored on the WPILib Artifactory server. If you find that you need a package that isn't available on artifactory, consult the roborio-wheels repository.

14.2 RobotPy subcommands

When you install RobotPy in your Python installation, it installs a package called `robotpy-cli`, which provides a `robotpy` command that can be used to perform tasks related to your robot and related code.

If you execute the command from the command line, it will show the various subcommands that are available:

Windows

```
py -3 -m robotpy
```

macOS

```
python3 -m robotpy
```

Linux

```
python3 -m robotpy
```

Note

If you don't see a list of commands but either see a RobotPy logo or an error saying `No module named robotpy.__main__; 'robotpy' is a package and cannot be directly executed`, you should uninstall the `robotpy` module and then reinstall it via pip.

This only affects users who upgraded from pre-2024 or the 2024 beta.

You can pass the `--help` argument to see more information about the subcommand. For example, to see help for the `sim` command you can do the following:

Windows

```
py -3 -m robotpy sim --help
```

macOS

```
python3 -m robotpy sim --help
```

Linux

```
python3 -m robotpy sim --help
```

This page has more detailed documentation for some of the subcommands:

14.2.1 Deploy Python program to roboRIO

Note

Before deploying the code to your robot, you must start by *installing RobotPy on your computer*

In particular, it is expected that you have ran `robotpy sync` to download all of the roboRIO python dependencies.

Windows

```
py -3 -m robotpy deploy
```

macOS

```
python3 -m robotpy deploy
```

Linux

```
python3 -m robotpy deploy
```

When you execute the `robotpy deploy` subcommand, it will do the following:

- Run `pytest` tests on your code (will exit if they fail)
- Install Python on the roboRIO (if not already present)
- Install python packages on the roboRIO as specified by your `pyproject.toml` (if not already present)
- Copy the entire robot project directory to the roboRIO and execute it

Warning

Avoid powering off the robot while deploying robot code. Interrupting the deployment process can corrupt the roboRIO filesystem and prevent your code from working until the roboRIO is *re-imaged*.

When successful, you will see a `SUCCESS: Deploy was successful!` message.

You can watch your robot code's output (and see any problems) with netconsole by using the Driver Station Log Viewer or pynetconsole. You can use netconsole and the normal FRC tools to interact with the running robot code.

See also

Viewing Console Output

Immediate feedback via Netconsole

When deploying the code to the roboRIO, you can have immediate feedback by adding the option *–nc*. This will cause the deploy command to show your program's console output, by launching a netconsole listener.

Windows

```
py -3 -m robotpy deploy --nc
```

macOS

```
python3 -m robotpy deploy --nc
```

Linux

```
python3 -m robotpy deploy --nc
```

Note

Viewing netconsole output requires the driver station software to be connected to your robot

Skipping Tests

In the event that the tests are failing but you want to upload the code anyway, you can skip them by adding the option *–skip-tests*.

Windows

```
py -3 -m robotpy deploy --skip-tests
```

macOS

```
python3 -m robotpy deploy --skip-tests
```

Linux

```
python3 -m robotpy deploy --skip-tests
```

15 Hardware APIs

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

15.1 Motors APIs

Programming your motors are absolutely essential to a moving robot! This section showcases some helpful classes and examples for getting your robot up and moving!

15.1.1 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 WPILib drivetrain classes, see *Using the WPILib Classes to Drive your Robot*.

Using PWM Motor Controllers

For a more detailed background on *how* PWM motor controllers work, see *PWM Motor Controllers in Depth*. To use a PWM motor controller, simply use the appropriate motor controller class provided by WPILib and supply it the port the motor controller(s) are plugged into on the roboRIO. All approved motor controllers have WPILib classes provided for them.

Note

The `Spark` and `VictorSP` classes are used here as an example; other PWM motor controller classes have exactly the same API.

Important

The `set()` method accepts values from **-1.0 to 1.0**, where:

- **-1.0** = full speed reverse
- **0.0** = stopped
- **1.0** = full speed forward

Where to Put This Code

In Command-Based programs:

Motor controllers should be declared as member variables in your subsystem class. Create them in the subsystem constructor, and use them in command methods.

JAVA

```
public class IntakeSubsystem extends SubsystemBase {
  private final Spark m_motor = new Spark(0);

  public void runIntake() {
    m_motor.set(0.8);
  }

  public void stopIntake() {
    m_motor.set(0.0);
  }
}
```

C++

```
class IntakeSubsystem : public frc2::SubsystemBase {
 public:
  IntakeSubsystem() : m_motor{0} {}

  void RunIntake() {
    m_motor.Set(0.8);
  }

  void StopIntake() {
    m_motor.Set(0.0);
  }

 private:
  frc::Spark m_motor;
};
```

PYTHON

```
class IntakeSubsystem(commands2.SubsystemBase):
    def __init__(self):
        super().__init__()
        self.motor = wpilib.Spark(0)

    def run_intake(self):
        self.motor.set(0.8)

    def stop_intake(self):
        self.motor.set(0.0)
```

In Timed Robot programs:

Motor controllers should be declared as member variables in your `Robot` class. Create them in `robotInit()`, and use them in periodic methods or autonomous/teleop methods.

JAVA

```
public class Robot extends TimedRobot {
  private Spark m_intakeMotor;
  private Joystick m_joystick;

  @Override
  public void robotInit() {
    m_intakeMotor = new Spark(0);
    m_joystick = new Joystick(0);
  }

  @Override
  public void teleopPeriodic() {
    // Run intake when button is pressed
    if (m_joystick.getRawButton(1)) {
      m_intakeMotor.set(0.8);
    } else {
      m_intakeMotor.set(0.0);
    }
  }
}
```

C++

```
class Robot : public frc::TimedRobot {
 public:
  Robot() : m_intakeMotor{0}, m_joystick{0} {}

  void TeleopPeriodic() override {
    // Run intake when button is pressed
    if (m_joystick.GetRawButton(1)) {
      m_intakeMotor.Set(0.8);
    } else {
      m_intakeMotor.Set(0.0);
```

(continues on next page)

(continued from previous page)

```
    }
  }

 private:
  frc::Spark m_intakeMotor;
  frc::Joystick m_joystick;
};
```

PYTHON

```
class MyRobot(wpilib.TimedRobot):
    def robotInit(self):
        self.intake_motor = wpilib.Spark(0)
        self.joystick = wpilib.Joystick(0)

    def teleopPeriodic(self):
        # Run intake when button is pressed
        if self.joystick.getRawButton(1):
            self.intake_motor.set(0.8)
        else:
            self.intake_motor.set(0.0)
```

Common Use Cases

Motor controllers are used throughout the robot for many mechanisms:

- **Intakes**: Spin wheels to collect game pieces
- **Shooters**: Spin flywheels to launch game pieces
- **Conveyors**: Move game pieces through the robot
- **Arms/Elevators**: Raise and lower mechanisms (often with position control)
- **Climbers**: Extend or retract climbing mechanisms

For drivetrain usage, see *Using the WPILib Classes to Drive your Robot*. For more complete examples, see *WPILib Example Projects*.

CAN Motor Controllers

CAN motor controllers are available through vendors such as CTR Electronics (Talon FX), REV Robotics (SPARK MAX/FLEX), and others. See *Third-Party CAN Devices*, *3rd Party Libraries*, and *Third Party Example Projects* for more information.

15.1.2 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.

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.

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 4096 steps of 1us each . The raw values sent to this module are in this 0-4096 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.

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

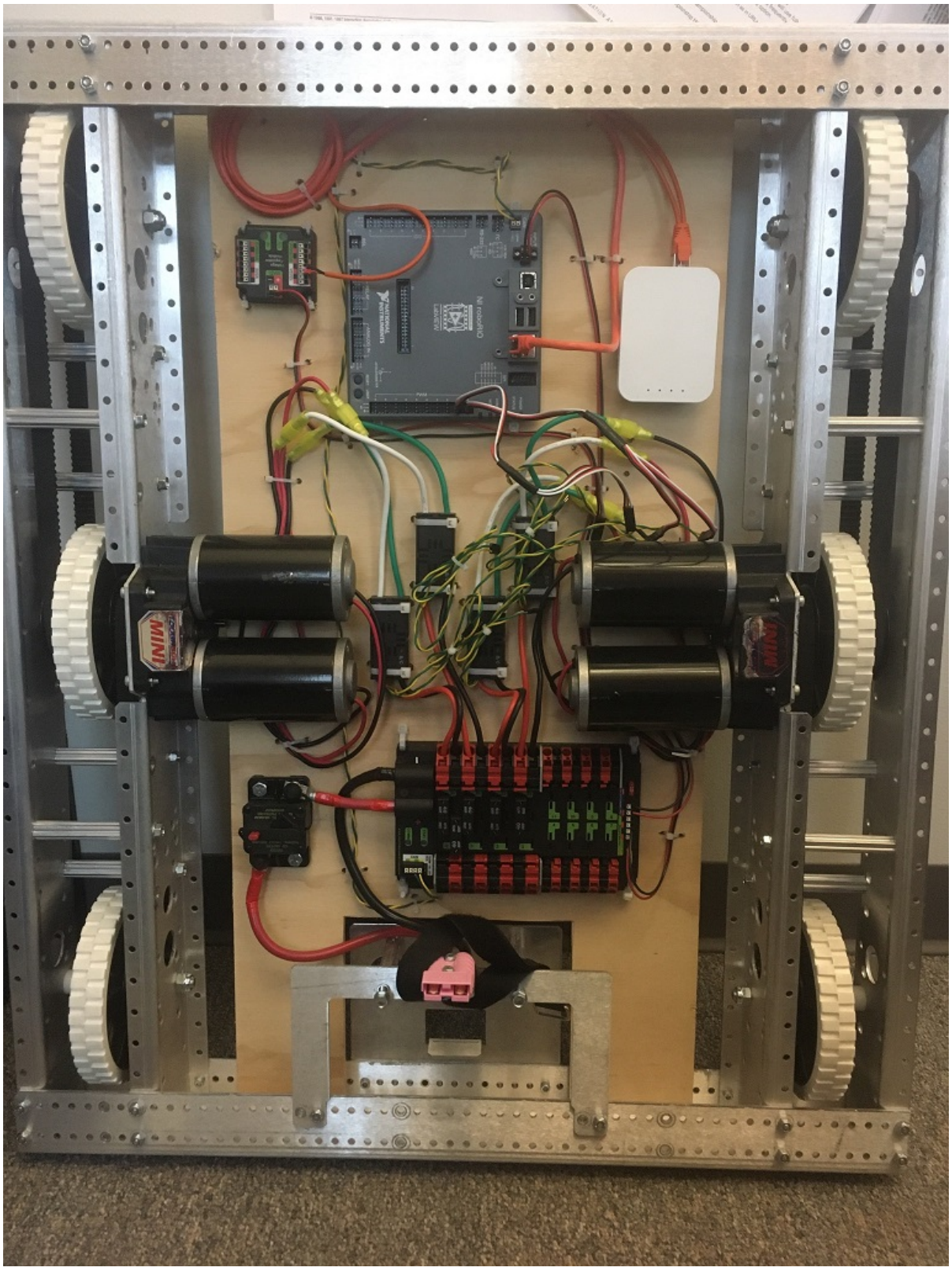
these motor controllers in your code, see *Using Motor Controllers in Code/Using PWM Motor Controllers*

15.1.3 Using the WPILib Classes to Drive your Robot

WPILib includes many classes to help make your robot get driving faster.

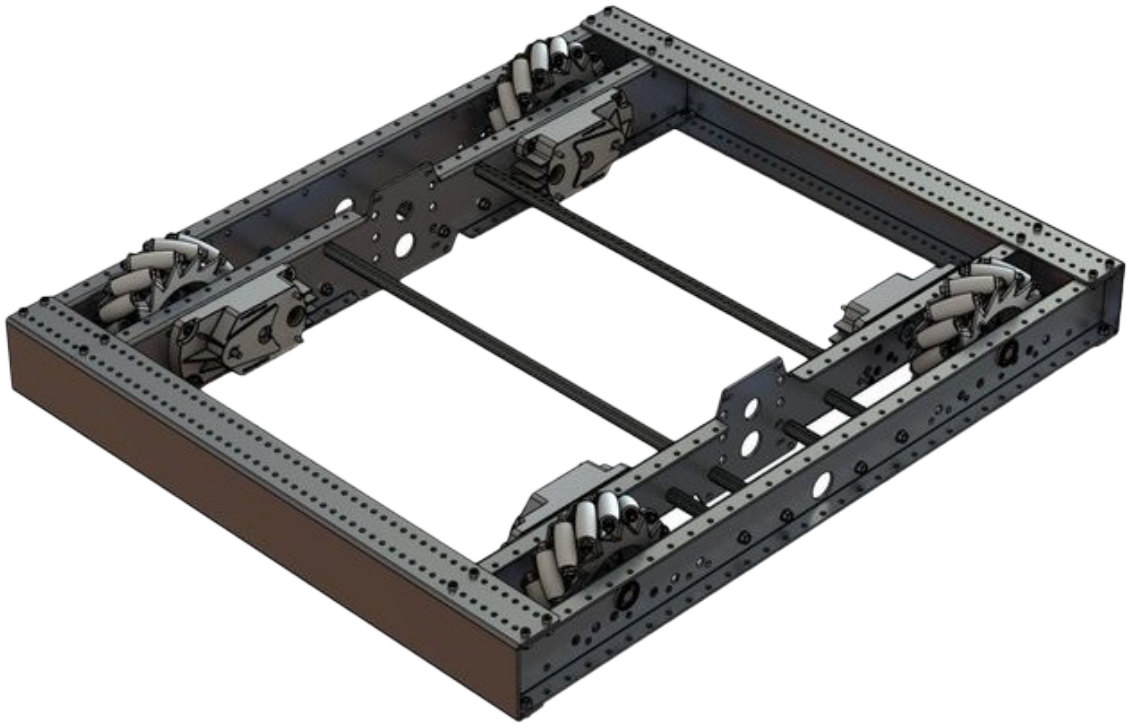
Standard drivetrains

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.

Drive Class Conventions

Motor Inversion

It is the responsibility of the user to manage proper inversions for their drivetrain. Users can invert motors by calling `setInverted()`/`SetInverted()` on their motor objects. Typically for differential drive trains, one side must be inverted to avoid spinning in circles, since the forward direction of each side of the tank drive is opposite of each other.

JAVA

```
PWMSparkMax m_motorRight = new PWMSparkMax(0);

public Robot() {
   m_motorRight.setInverted(true);
}
```

C++

```
frc::PWMSparkMax m_motorLeft{0};
public:
 Robot::Robot() {
   m_motorRight.SetInverted(true);
 }
```

PYTHON

```
def robotInit(self):
    self.motorRight = wpilib.PWMSparkMax(0)
    self.motorRight.setInverted(True)
```

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 DifferentialDrive 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

```
m_motorRight.setSafetyEnabled(true);
m_motorRight.setSafetyEnabled(false);
m_motorRight.setExpiration(.1);
m_motorRight.feed()
```

C++

```
m_motorRight->SetSafetyEnabled(true);
m_motorRight->SetSafetyEnabled(false);
m_motorRight->SetExpiration(.1);
m_motorRight->Feed();
```

PYTHON

```
m_motorRight.setSafetyEnabled(True)
m_motorRight.setSafetyEnabled(False)
m_motorRight.setExpiration(.1)
m_motorRight.feed()
```

By default all Drive 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

The drive classes use the NWU axes convention (North-West-Up as external reference in the world frame). The positive X axis points ahead, the positive Y axis points left, and the positive Z axis points up. We use NWU here because the rest of the library, and math in general, use NWU axes convention.

Joysticks follow NED (North-East-Down) convention, where the positive X axis points ahead, the positive Y axis points right, and the positive Z axis points down. However, it's important to note that axes values are rotations around the respective axes, not translations. When viewed with each axis pointing toward you, CCW is a positive value and CW is a negative value. Pushing forward on the joystick is a CW rotation around the Y axis, so you get a negative value. Pushing to the right is a CCW rotation around the X axis, so you get a positive value.

Note

See the *Coordinate System* section for more detail about the axis conventions and coordinate systems.

Using the DifferentialDrive class to control Differential Drive robots

> **Note**
>
> WPILib provides separate Robot Drive classes for the most common drive train configurations (differential and mecanum). 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

```
public class Robot extends TimedRobot {
  private DifferentialDrive m_robotDrive;
  private final PWMSparkMax m_leftMotor = new PWMSparkMax(0);
  private final PWMSparkMax m_rightMotor = new PWMSparkMax(1);

  @Override
  public void robotInit() {
    // We need to invert one side of the drivetrain so that positive voltages
    // result in both sides moving forward. Depending on how your robot's
    // gearbox is constructed, you might have to invert the left side instead.
    m_rightMotor.setInverted(true);

    m_robotDrive = new DifferentialDrive(m_leftMotor::set, m_rightMotor::set);
  }
```

C++ (Header)

```
  frc::PWMSparkMax m_leftMotor{0};
  frc::PWMSparkMax m_rightMotor{1};
  frc::DifferentialDrive m_robotDrive{
      [&](double output) { m_leftMotor.Set(output); },
      [&](double output) { m_rightMotor.Set(output); }};
```

C++ (Source) :sync: C++ (Source)

```
  void RobotInit() override {
    // We need to invert one side of the drivetrain so that positive voltages
    // result in both sides moving forward. Depending on how your robot's
    // gearbox is constructed, you might have to invert the left side instead.
    m_rightMotor.SetInverted(true);
  }
```

Python

```
    def robotInit(self):
        """Robot initialization function"""

        leftMotor = wpilib.PWMSparkMax(0)
        rightMotor = wpilib.PWMSparkMax(1)
        self.robotDrive = wpilib.drive.DifferentialDrive(leftMotor, rightMotor)
        # We need to invert one side of the drivetrain so that positive voltages
        # result in both sides moving forward. Depending on how your robot's
        # gearbox is constructed, you might have to invert the left side instead.
        rightMotor.setInverted(True)
```

Multi-Motor DifferentialDrive

Many FRC® drivetrains have more than 1 motor on each side. Classes derived from `PWMMotorController` (Java / C++ / Python) have an `addFollower` method so that multiple follower motor controllers can be updated when the leader motor controller is commanded. CAN motor controllers have similar features, review the vendor's documentation to see how to use them. The examples below show a 4 motor (2 per side) drivetrain. To extend to more motors, simply create the additional controllers and use additional `addFollower` calls.

Java

Class variables (e.g. in Robot.java or Subsystem):

```
  // The motors on the left side of the drive.
  private final PWMSparkMax m_leftLeader = new PWMSparkMax(DriveConstants.
↪kLeftMotor1Port);
  private final PWMSparkMax m_leftFollower = new PWMSparkMax(DriveConstants.
↪kLeftMotor2Port);

  // The motors on the right side of the drive.
  private final PWMSparkMax m_rightLeader = new PWMSparkMax(DriveConstants.
↪kRightMotor1Port);
  private final PWMSparkMax m_rightFollower = new PWMSparkMax(DriveConstants.
↪kRightMotor2Port);
```

In Robot or Subsystem constructor:

```
    m_leftLeader.addFollower(m_leftFollower);
    m_rightLeader.addFollower(m_rightFollower);

    // We need to invert one side of the drivetrain so that positive voltages
    // result in both sides moving forward. Depending on how your robot's
    // gearbox is constructed, you might have to invert the left side instead.
    m_rightLeader.setInverted(true);
```

C++ (Header)

```
 private:
  // The motor controllers
  frc::PWMSparkMax m_left1;
  frc::PWMSparkMax m_left2;
  frc::PWMSparkMax m_right1;
  frc::PWMSparkMax m_right2;

  // The robot's drive
  frc::DifferentialDrive m_drive{[&](double output) { m_left1.Set(output); },
                                 [&](double output) { m_right1.Set(output); }};
```

C++ (Source)

In Robot or Subsystem constructor:

```
  m_left1.AddFollower(m_left2);
  m_right1.AddFollower(m_right2);

  // We need to invert one side of the drivetrain so that positive voltages
  // result in both sides moving forward. Depending on how your robot's
  // gearbox is constructed, you might have to invert the left side instead.
  m_right1.SetInverted(true);
```

Python

```
def robotInit(self):
    leftLeader = wpilib.Spark(1)
    leftFollower = wpilib.Spark(2)
    leftLeader.addFollower(leftFollower)
    leftLeader.setInverted(
        True
    )  # if you want to invert the entire side you can do so here
    rightLeader = wpilib.Spark(3)
    rightFollower = wpilib.Spark(4)
    rightLeader.addFollower(rightFollower)
    self.drive = wpilib.drive.DifferentialDrive(leftLeader, rightLeader)
```

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

```
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 button for␣
↪turning in-place.
    myDrive.curvatureDrive(-driveStick.getY(), -driveStick.getX(), driveStick.
↪getButton(1));
}
```

C++

```
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));
}
```

PYTHON

```
def teleopPeriodic(self):
    # Tank drive with a given left and right rates
    self.myDrive.tankDrive(-self.leftStick.getY(), -self.rightStick.getY())
    # Arcade drive with a given forward and turn rate
    self.myDrive.arcadeDrive(-self.driveStick.getY(), -self.driveStick.getX())
    # Curvature drive with a given forward and turn rate, as well as a button for␣
↪turning in-place.
    self.myDrive.curvatureDrive(-self.driveStick.getY(), -self.driveStick.getX(),␣
↪self.driveStick.getButton(1))
```

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

```
  private static final int kFrontLeftChannel = 2;
  private static final int kRearLeftChannel = 3;
  private static final int kFrontRightChannel = 1;
  private static final int kRearRightChannel = 0;

  @Override
  public void robotInit() {
    PWMSparkMax frontLeft = new PWMSparkMax(kFrontLeftChannel);
    PWMSparkMax rearLeft = new PWMSparkMax(kRearLeftChannel);
    PWMSparkMax frontRight = new PWMSparkMax(kFrontRightChannel);
    PWMSparkMax rearRight = new PWMSparkMax(kRearRightChannel);
    // Invert the right side motors.
    // You may need to change or remove this to match your robot.
    frontRight.setInverted(true);
    rearRight.setInverted(true);

    m_robotDrive = new MecanumDrive(frontLeft::set, rearLeft::set, frontRight::set,␣
↪rearRight::set);
  }
```

C++

```
private:
 static constexpr int kFrontLeftChannel = 0;
 static constexpr int kRearLeftChannel = 1;
 static constexpr int kFrontRightChannel = 2;
 static constexpr int kRearRightChannel = 3;

 frc::PWMSparkMax m_frontLeft{kFrontLeftChannel};
 frc::PWMSparkMax m_rearLeft{kRearLeftChannel};
 frc::PWMSparkMax m_frontRight{kFrontRightChannel};
 frc::PWMSparkMax m_rearRight{kRearRightChannel};
 frc::MecanumDrive m_robotDrive{
     [&](double output) { m_frontLeft.Set(output); },
     [&](double output) { m_rearLeft.Set(output); },
     [&](double output) { m_frontRight.Set(output); },
     [&](double output) { m_rearRight.Set(output); }};

 void RobotInit() override {
   // Invert the right side motors. You may need to change or remove this to
   // match your robot.
   m_frontRight.SetInverted(true);
   m_rearRight.SetInverted(true);
 }
```

PYTHON

```
    # Channels on the roboRIO that the motor controllers are plugged in to
    kFrontLeftChannel = 2
    kRearLeftChannel = 3
    kFrontRightChannel = 1
    kRearRightChannel = 0

    def robotInit(self):
        self.frontLeft = wpilib.PWMSparkMax(self.kFrontLeftChannel)
        self.rearLeft = wpilib.PWMSparkMax(self.kRearLeftChannel)
        self.frontRight = wpilib.PWMSparkMax(self.kFrontRightChannel)
        self.rearRight = wpilib.PWMSparkMax(self.kRearRightChannel)

        # invert the right side motors
        # you may need to change or remove this to match your robot
        self.frontRight.setInverted(True)
        self.rearRight.setInverted(True)

        self.robotDrive = wpilib.drive.MecanumDrive(
            self.frontLeft, self.rearLeft, self.frontRight, self.rearRight
        )

        self.stick = wpilib.Joystick(self.kJoystickChannel)
```

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

```
public void teleopPeriodic() {
    // Drive using the X, Y, and Z axes of the joystick.
    m_robotDrive.driveCartesian(-m_stick.getY(), -m_stick.getX(), -m_stick.getZ());
    // Drive at 45 degrees relative to the robot, at the speed given by the Y axis of␣
↪the joystick, with no rotation.
    m_robotDrive.drivePolar(-m_stick.getY(), Rotation2d.fromDegrees(45), 0);
}
```

C++

```
void TeleopPeriodic() override {
    // Drive using the X, Y, and Z axes of the joystick.
    m_robotDrive.driveCartesian(-m_stick.GetY(), -m_stick.GetX(), -m_stick.GetZ());
    // Drive at 45 degrees relative to the robot, at the speed given by the Y axis of␣
↪the joystick, with no rotation.
    m_robotDrive.drivePolar(-m_stick.GetY(), 45_deg, 0);
}
```

PYTHON

```
def teleopPeriodic(self):
    # Drive using the X, Y, and Z axes of the joystick.
    self.robotDrive.driveCartesian(-self.stick.getY(), -self.stick.getX(), -self.
↪stick.getZ())
    # Drive at 45 degrees relative to the robot, at the speed given by the Y axis of␣
↪the joystick, with no rotation.
    self.robotDrive.drivePolar(-self.stick.getY(), Rotation2d.fromDegrees(45), 0)
```

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*.

15.1.4 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 (Pulse Width Modulation (PWM) signal, 0.6 ms - 2.4 ms pulse width)

Constructing a Servo object

JAVA

```
Servo exampleServo = new Servo(1);
```

C++

```
frc::Servo exampleServo {1};
```

PYTHON

```
exampleServo = wpilib.Servo(1)
```

A servo object is constructed by passing a channel.

Setting Servo Values

JAVA

```
exampleServo.set(.5);
exampleServo.setAngle(75);
```

C++

```
exampleServo.Set(.5);
exampleServo.SetAngle(75);
```

PYTHON

```
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 from 0 to 180. This method will work for servos with the same range as the Hitec HS-322HD servo . Any values passed to this method outside the specified range will be coerced to the boundary.

15.2 Pneumatics APIs

15.2.1 Operating Pneumatic Cylinders

FRC teams can use a *solenoid valve* as part of performing a variety of tasks, including shifting gearboxes and moving robot mechanisms. A solenoid valve is used to electronically switch a pressurized air line "on" or "off". Solenoids are controlled by a robot's Pneumatics Control Module, or Pneumatic Hub, which is in turn connected to the robot's roboRIO via *CAN*. The easiest way to see a solenoid's state is via the LEDs on the PCM or PH (which indicates if the valve is "on" or not). When un-powered, solenoids can be manually actuated with the small button on the valve body.

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.

Single Solenoids in WPILib

Single solenoids in WPILib are controlled using the `Solenoid` class (Java / C++). To construct a Solenoid object, simply pass the desired port number (assumes default CAN ID) and pneumatics module type or CAN ID, pneumatics module type, 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

```
30  // Solenoid corresponds to a single solenoid.
31  // In this case, it's connected to channel 0 of a PH with the default CAN ID.
32  private final Solenoid m_solenoid = new Solenoid(PneumaticsModuleType.REVPH, 0);
```

```
72    /*
73     * The output of GetRawButton is true/false depending on whether
74     * the button is pressed; Set takes a boolean for whether
75     * to retract the solenoid (false) or extend it (true).
76     */
77    m_solenoid.set(m_stick.getRawButton(kSolenoidButton));
```

C++ (Header)

```
44  // Solenoid corresponds to a single solenoid.
45  // In this case, it's connected to channel 0 of a PH with the default CAN
46  // ID.
47  frc::Solenoid m_solenoid{frc::PneumaticsModuleType::REVPH, 0};
```

C++ (Source)

```
42  /*
43   * The output of GetRawButton is true/false depending on whether
44   * the button is pressed; Set takes a boolean for whether
45   * to retract the solenoid (false) or extend it (true).
46   */
47  m_solenoid.Set(m_stick.GetRawButton(kSolenoidButton));
```

Double Solenoids in WPILib

Double solenoids are controlled by the `DoubleSolenoid` class in WPILib (Java / C++). 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 CAN ID can be passed to the DoubleSolenoid if teams have a non-default CAN ID.

Java

```
34  // DoubleSolenoid corresponds to a double solenoid.
35  // In this case, it's connected to channels 1 and 2 of a PH with the default CAN ID.
36  private final DoubleSolenoid m_doubleSolenoid =
37      new DoubleSolenoid(PneumaticsModuleType.REVPH, 1, 2);
```

```
84      m_doubleSolenoid.set(DoubleSolenoid.Value.kForward);
85      m_doubleSolenoid.set(DoubleSolenoid.Value.kReverse);
```

C++ (Header)

```
49  // DoubleSolenoid corresponds to a double solenoid.
50  // In this case, it's connected to channels 1 and 2 of a PH with the default
51  // CAN ID.
52  frc::DoubleSolenoid m_doubleSolenoid{frc::PneumaticsModuleType::REVPH, 1, 2};
```

C++ (Source)

```
54    m_doubleSolenoid.Set(frc::DoubleSolenoid::kForward);
55    m_doubleSolenoid.Set(frc::DoubleSolenoid::kReverse);
```

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

```
Solenoid exampleSingle = new Solenoid(PneumaticsModuleType.CTREPCM, 0);
DoubleSolenoid exampleDouble = new DoubleSolenoid(PneumaticsModuleType.CTREPCM, 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();
   exampleDouble.toggle();
}
```

C++

```
frc::Solenoid exampleSingle{frc::PneumaticsModuleType::CTREPCM, 0};
frc::DoubleSolenoid exampleDouble{frc::PneumaticsModuleType::CTREPCM, 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();
}
```

15.2.2 Generating and Storing Pressure

Pressure is created using a pneumatic compressor and stored in pneumatic tanks. The compressor must be on the robot and powered by the robot's pneumatics module. 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 pneumatic module will automatically turn the compressor on when the digital 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 the `Compressor` (Java / C++) class, 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, as shown by the following code from the Solenoid example project (Java, C++):

> **Note**
>
> The Compressor object is only needed if you want the ability to turn off the compressor, change the pressure sensor (PH only), or query compressor status.

Construct a `Compressor` object:

REV Pneumatic Hub (PH)

Java

```
  // Compressor connected to a PH with a default CAN ID (1)
  private final Compressor m_compressor = new Compressor(PneumaticsModuleType.REVPH);
```

C++ (Header)

```
  // Compressor connected to a PH with a default CAN ID
  frc::Compressor m_compressor{frc::PneumaticsModuleType::REVPH};
```

CTRE Pneumatics Control Module (PCM)

Java

```
  // Compressor connected to a PCM with a default CAN ID (0)
  private final Compressor m_compressor = new Compressor(PneumaticsModuleType.
↪CTREPCM);
```

C++ (Header)

```
  // Compressor connected to a PH with a default CAN ID
```

Querying compressor current and state:

Java

```
          // Get compressor current draw.
          return m_compressor.getCurrent();
          // Get whether the compressor is active.
          return m_compressor.isEnabled();
          // Get the digital pressure switch connected to the PCM/PH.
          // The switch is open when the pressure is over ~120 PSI.
          return m_compressor.getPressureSwitchValue();
```

C++ (Source)

```
    // Get compressor current draw.
    units::ampere_t compressorCurrent = m_compressor.GetCurrent();
    return compressorCurrent.value();
    // Get whether the compressor is active.
    return m_compressor.IsEnabled();
    // Get the digital pressure switch connected to the PCM/PH.
    // The switch is open when the pressure is over ~120 PSI.
    return m_compressor.GetPressureSwitchValue();
```

Enable/disable digital closed-loop compressor control (enabled by default):

Java

```
            // Disable closed-loop mode on the compressor.
            m_compressor.disable();
            // Enable closed-loop mode based on the digital pressure switch␣
↪connected to the
            // PCM/PH.
            // The switch is open when the pressure is over ~120 PSI.
            m_compressor.enableDigital();
```

C++ (Source)

```
      // Disable closed-loop mode on the compressor.
      m_compressor.Disable();
      // Enable closed-loop mode based on the digital pressure switch
      // connected to the PCM/PH. The switch is open when the pressure is over
      // ~120 PSI.
      m_compressor.EnableDigital();
```

The Pneumatic Hub also has methods for enabling compressor control using the REV Analog Pressure Sensor:

Java

```
        // Enable closed-loop mode based on the analog pressure sensor connected to␣
↪the PH.
        // The compressor will run while the pressure reported by the sensor is in␣
↪the
        // specified range ([70 PSI, 120 PSI] in this example).
        // Analog mode exists only on the PH! On the PCM, this enables digital␣
↪control.
        m_compressor.enableAnalog(70, 120);
        // Enable closed-loop mode based on both the digital pressure switch AND␣
↪the analog
        // pressure sensor connected to the PH.
        // The compressor will run while the pressure reported by the analog sensor␣
↪is in the
        // specified range ([70 PSI, 120 PSI] in this example) AND the digital␣
↪switch reports
        // that the system is not full.
        // Hybrid mode exists only on the PH! On the PCM, this enables digital␣
↪control.
        m_compressor.enableHybrid(70, 120);
```

C++ (Source)

```
    // Enable closed-loop mode based on the analog pressure sensor connected
    // to the PH. The compressor will run while the pressure reported by the
    // sensor is in the specified range ([70 PSI, 120 PSI] in this example).
    // Analog mode exists only on the PH! On the PCM, this enables digital
    // control.
    m_compressor.EnableAnalog(70_psi, 120_psi);
    // Enable closed-loop mode based on both the digital pressure switch AND the␣
↪analog
    // pressure sensor connected to the PH.
    // The compressor will run while the pressure reported by the analog sensor is␣
↪in the
    // specified range ([70 PSI, 120 PSI] in this example) AND the digital switch␣
↪reports
    // that the system is not full.
    // Hybrid mode exists only on the PH! On the PCM, this enables digital control.
    m_compressor.EnableHybrid(70_psi, 120_psi);
```

Pressure Transducers

A pressure transducer is a sensor where analog voltage is proportial to the measured pressure.

Pneumatic Hub

The Pneumatic Hub has analog inputs that may be used to read a pressure transducer using the Compressor class.

Java

```
// Compressor connected to a PH with a default CAN ID (1)
private final Compressor m_compressor = new Compressor(PneumaticsModuleType.REVPH);
```

```
        // Get the pressure (in PSI) from the analog sensor connected to the PH.
        // This function is supported only on the PH!
        // On a PCM, this function will return 0.
        return m_compressor.getPressure();
```

C++ (Header)

```
// Compressor connected to a PH with a default CAN ID
frc::Compressor m_compressor{frc::PneumaticsModuleType::REVPH};
```

C++ (Source)

```
  // Get the pressure (in PSI) from the analog sensor connected to the PH.
  // This function is supported only on the PH!
  // On a PCM, this function will return 0.
  units::pounds_per_square_inch_t pressure = m_compressor.GetPressure();
  return pressure.value();
```

roboRIO

A pressure transducer can be connected to the Analog Input ports on the roboRIO, and can be read by the `AnalogInput` or `AnalogPotentiometer` classes in WPILib.

Java

```
 // External analog pressure sensor
 // 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
 static final double kScale = 250;
 static final double kOffset = -25;
 private final AnalogPotentiometer m_pressureTransducer =
     new AnalogPotentiometer(/* the AnalogIn port*/ 2, kScale, kOffset);
```

```
  // Get the pressure (in PSI) from an analog pressure sensor connected to the RIO.
  return m_pressureTransducer.get();
```

C++ (Header)

```
 // External analog pressure sensor
 // 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
 static constexpr double kScale = 250;
 static constexpr double kOffset = -25;
 frc::AnalogPotentiometer m_pressureTransducer{/* the AnalogIn port*/ 2,
                                               kScale, kOffset};
```

C++ (Source)

```
// Get the pressure (in PSI) from an analog pressure sensor connected to
// the RIO.
return units::pounds_per_square_inch_t{m_pressureTransducer.Get()};
```

15.2.3 Using the FRC Control System to Control Pneumatics

There are two options for operating solenoids to control pneumatic cylinders, the CTRE Pneumatics Control Module and the REV Robotics Pneumatics Hub.



The CTRE Pneumatics Control Module (PCM) is a CAN-based device that provides control over the compressor and up to 8 solenoids per module.



The REV Pneumatic Hub (PH) is a CAN-based device that provides control over the compressor and up to 16 solenoids per module.

These devices are integrated into WPILib through a series of classes that make them 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 `Solenoid` class that controls the solenoid channels.

These modules are responsible for regulating the robot's pressure using a pressure switch and a compressor and switching solenoids on and off. They communicate with the roboRIO over CAN. For more information, see *Hardware Component Overview*.

15.2.4 Module Numbers

CAN Devices are identified by their CAN ID. The default CAN ID for PCMs is 0. The default CAN ID for PHs is 1. If using a single module on the bus it is recommended to leave it at the default CAN ID. Additional modules can be used where the modules corresponding solenoids are differentiated by the module number in the constructors of the `Solenoid`, `DoubleSolenoid` and `Compressor` classes.

15.3 Sensors

Sensors are an integral way of having your robot hardware and software communicate with each other. This section highlights interfacing with those sensors at a software level.

15.3.1 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.

What sensors does WPILIB support?

The roboRIO includes an *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.

15.3.2 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*

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

```
// 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();
```

C++

```
// 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);
// 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.

There are 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.

JAVA

```
// Sets the accelerometer to measure between -8 and 8 G's
accelerometer.setRange(BuiltInAccelerometer.Range.k8G);
```

C++

```
// Sets the accelerometer to measure between -8 and 8 G's
accelerometer.SetRange(BuiltInAccelerometer::Range::kRange_8G);
```

ADXL345_I2C

The ADXL345_I2C class (Java, C++) provides support for the ADXL345 accelerometer over the I2C communications bus.

JAVA

```
// Creates an ADXL345 accelerometer object on the MXP I2C port
// with a measurement range from -8 to 8 G's
ADXL345_I2C accelerometer = new ADXL345_I2C(I2C.Port.kMXP, ADXL345_I2C.Range.k8G);
```

C++

```
// 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, frc::ADXL345_I2C::Range::kRange_8G};
```

ADXL345_SPI

The ADXL345_SPI class (Java, C++) provides support for the ADXL345 accelerometer over the SPI communications bus.

JAVA

```
// Creates an ADXL345 accelerometer object on the MXP SPI port
// with a measurement range from -8 to 8 G's
ADXL345_SPI accelerometer = new ADXL345_SPI(SPI.Port.kMXP, ADXL345_SPI.Range.k8G);
```

C++

```
// 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, frc::ADXL345_SPI::Range::kRange_8G};
```

ADXL362

The `ADXL362` class (Java, C++) provides support for the ADXL362 accelerometer over the SPI communications bus.

JAVA

```
// Creates an ADXL362 accelerometer object on the MXP SPI port
// with a measurement range from -8 to 8 G's
ADXL362 accelerometer = new ADXL362(SPI.Port.kMXP, ADXL362.Range.k8G);
```

C++

```
// 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, frc::ADXL362::Range::kRange_8G};
```

BuiltInAccelerometer

The `BuiltInAccelerometer` class (Java, C++) provides access to the roboRIO's own built-in accelerometer:

JAVA

```
// Creates an object for the built-in accelerometer
// Range defaults to +- 8 G's
BuiltInAccelerometer accelerometer = new BuiltInAccelerometer();
```

C++

```
// Creates an object for the built-in accelerometer
// Range defaults to +- 8 G's
frc::BuiltInAccelerometer accelerometer;
```

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.

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

```
double prevXAccel = 0.0;
double prevYAccel = 0.0;
BuiltInAccelerometer 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) / 0.02;
```

(continues on next page)

(continued from previous page)

```
    double yJerk = (yAccel - prevYAccel) / 0.02;
    prevXAccel = xAccel;
    prevYAccel = yAccel;
}
```

C++

```
double prevXAccel = 0.0;
double prevYAccel = 0.0;
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) / 0.02;
    double yJerk = (yAccel - prevYAccel) / 0.02;
    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

```
BuiltInAccelerometer accelerometer = new BuiltInAccelerometer();
// Create a LinearFilter that will calculate a moving average of the measured X␣
↪acceleration over the past 10 iterations of the main loop
LinearFilter xAccelFilter = LinearFilter.movingAverage(10);
@Override
public void robotPeriodic() {
    // Get the filtered X acceleration
    double filteredXAccel = xAccelFilter.calculate(accelerometer.getX());
}
```

C++

```
frc::BuiltInAccelerometer accelerometer;
// Create a LinearFilter that will calculate a moving average of the measured X␣
↪acceleration over the past 10 iterations of the main loop
auto xAccelFilter = frc::LinearFilter::MovingAverage(10);
void Robot::RobotPeriodic() {
    // Get the filtered X acceleration
    double filteredXAccel = xAccelFilter.Calculate(accelerometer.GetX());
}
```

15.3.3 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.

There are getters the current angular rate and heading and functions for zeroing the current heading and calibrating the gyro.

> **Note**
>
> It is crucial that the robot remain stationary while calibrating a gyro.

ADIS16448

The ADIS16448 uses the `ADIS16448_IMU` class (Java, C++, `Python`). See the Analog Devices ADIS16448 documentation for additional information and examples.

> **Warning**
>
> The Analog Devices documentation linked above contains outdated instructions for software installation as the ADIS16448 is now built into WPILib.

JAVA

```
// ADIS16448 plugged into the MXP port
ADIS16448_IMU gyro = new ADIS16448_IMU();
```

C++

```
// ADIS16448 plugged into the MXP port
ADIS16448_IMU gyro;
```

PYTHON

```
from wpilib import ADIS16448_IMU
# ADIS16448 plugged into the MXP port
self.gyro = ADIS16448_IMU()
```

ADIS16470

The ADIS16470 uses the `ADIS16470_IMU` class (Java, C++, Python). See the Analog Devices ADIS16470 documentation for additional information and examples.

> **Warning**
>
> The Analog Devices documentation linked above contains outdated instructions for software installation as the ADIS16470 is now built into WPILib.

JAVA

```
// ADIS16470 plugged into the SPI port
ADIS16470_IMU gyro = new ADIS16470_IMU();
```

C++

```
// ADIS16470 plugged into the SPI port
ADIS16470_IMU gyro;
```

PYTHON

```
# ADIS16470 plugged into the SPI port
self.gyro = ADIS16470_IMU()
```

ADXRS450_Gyro

The `ADXRS450_Gyro` class (Java, C++, Python) 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

```
// Creates an ADXRS450_Gyro object on the onboard SPI port
ADXRS450_Gyro gyro = new ADXRS450_Gyro();
```

C++

```
// Creates an ADXRS450_Gyro object on the onboard SPI port
frc::ADXRS450_Gyro gyro;
```

PYTHON

```
# Creates an ADXRS450_Gyro object on the onboard SPI port
self.gyro = ADXRS450_Gyro()
```

AnalogGyro

The `AnalogGyro` class (Java, C++, Python) 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

```
// Creates an AnalogGyro object on port 0
AnalogGyro gyro = new AnalogGyro(0);
```

C++

```
// Creates an AnalogGyro object on port 0
frc::AnalogGyro gyro{0};
```

PYTHON

```
# Creates an AnalogGyro object on port 0
self.gyro = AnalogGyro(0)
```

navX

The navX uses the AHRS class. See the navX documentation for additional connection types.

JAVA

```
// navX MXP using SPI
AHRS gyro = new AHRS(NavXComType.kMXP_SPI);
```

C++

```
// navX MXP using SPI
studica::AHRS navx{studica::AHRS::NavXComType::kMXP_SPI};
```

PYTHON

```
import navx
# navX MXP using SPI
self.gyro = navx.AHRS(NavXComType.kMXP_SPI)
```

Pigeon 2

The Pigeon should use the Pigeon2 class. The Pigeon can either be connected with CAN or by data cable to a TalonSRX. The Pigeon IMU User's Guide contains full details on using the Pigeon.

JAVA

```
private final Pigeon2 pidgey = new Pigeon2(1, "rio"); // Pigeon is on roboRIO CAN Bus␣
↪with device ID 1
```

C++

```
ctre::phoenix6::hardware::Pigeon2 pidgey{1, "rio"}; // Pigeon is on roboRIO CAN Bus␣
↪with device ID 1
```

PYTHON

```
import phoenix5
import ctre.sensors
self.gyro = ctre.WPI_PigeonIMU(0); # Pigeon is on CAN Bus with device ID 0
# OR (choose one or the other based on your connection)
talon = ctre.TalonSRX(0); # TalonSRX is on CAN Bus with device ID 0
self.gyro = ctre.WPI_PigeonIMU(talon) # Pigeon uses the talon created above
```

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

```
// Use gyro declaration from above here
public Robot() {
    // Places a compass indicator for the gyro heading on the dashboard
    Shuffleboard.getTab("Example tab").add(gyro);
}
```

C++

```
// Use gyro declaration from above here
Robot::Robot() {
    // Places a compass indicator for the gyro heading on the dashboard
    frc::Shuffleboard.GetTab("Example tab").Add(gyro);
}
```

PYTHON

```
from wpilib.shuffleboard import Shuffleboard
def robotInit(self):
    # Use gyro declaration from above here
    # Places a compass indicator for the gyro heading on the dashboard
    Shuffleboard.getTab("Example tab").add(self.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

```
// Use gyro declaration from above here
// The gain for a simple P loop
double kP = 1;
// Initialize motor controllers and drive
Spark leftLeader = new Spark(0);
Spark leftFollower = new Spark(1);
Spark rightLeader = new Spark(2);
```

(continues on next page)

(continued from previous page)

```
Spark rightFollower = new Spark(3);
DifferentialDrive drive = new DifferentialDrive(leftLeader::set, rightLeader::set);

public Robot() {
    // 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.);
    // Invert the right side of the drivetrain. You might have to invert the other␣
↪side
    rightLeader.setInverted(true);
    // Configure the followers to follow the leaders
    leftLeader.addFollower(leftFollower);
    rightLeader.addFollower(rightFollower);
}

@Override
public void 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);
}
```

C++

```
// Use gyro declaration from above here
// The gain for a simple P loop
double kP = 1;
// Initialize motor controllers and drive
frc::Spark leftLeader{0};
frc::Spark leftFollower{1};
frc::Spark rightLeader{2};
frc::Spark rightFollower{3};
frc::DifferentialDrive drive{[&](double output) { leftLeader.Set(output); },
                             [&](double output) { rightLeader.Set(output); }};
Robot::Robot() {
    // Invert the right side of the drivetrain. You might have to invert the other␣
↪side
    rightLeader.SetInverted(true);
    // Configure the followers to follow the leaders
    leftLeader.AddFollower(leftFollower);
    rightLeader.AddFollower(rightFollower);
}
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);
}
```

PYTHON

```
from wpilib import Spark
from wpilib.drive import DifferentialDrive


def robotInit(self):
    # Use gyro declaration from above here
    # The gain for a simple P loop
    self.kP = 1
    # Initialize motor controllers and drive
    leftLeader = Spark(0)
    leftFollower = Spark(1)
    rightLeader = Spark(2)
    rightFollower = Spark(3)
    leftLeader.addFollower(leftFollower)
    rightLeader.addFollower(rightFollower)
    self.drive = DifferentialDrive(leftLeader, rightLeader)
    rightLeader.setInverted(True)


def autonomousPeriodic(self):
    # Setpoint is implicitly 0, since we don't want the heading to change
    error = -self.gyro.getRate()
    # Drives forward continuously at half speed, using the gyro to stabilize the␣
↪heading
    self.drive.tankDrive(0.5 + self.kP * error, 0.5 - self.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

```
// Use gyro declaration from above here
// 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);
Spark right1 = new Spark(2);
Spark right2 = new Spark(3);
MotorControllerGroup leftMotors = new MotorControllerGroup(left1, left2);
MotorControllerGroup rightMotors = new MotorControllerGroup(right1, right2);
DifferentialDrive drive = new DifferentialDrive(leftMotors, rightMotors);
```

(continues on next page)

(continued from previous page)

```
public Robot() {
    rightMotors.setInverted(true);
}

@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);
}
```

C++

```
// Use gyro declaration from above here
// 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::MotorControllerGroup leftMotors{left1, left2};
frc::MotorControllerGroup rightMotors{right1, right2};
frc::DifferentialDrive drive{leftMotors, rightMotors};
Robot::Robot() {
  rightMotors.SetInverted(true);
}
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);
}
```

PYTHON

```
from wpilib import Spark
from wpilib.drive import DifferentialDrive


def robotInit(self):
    # Use gyro declaration from above here
    # The gain for a simple P loop
    self.kP = 1
    # Initialize motor controllers and drive
    leftLeader = Spark(0)
    leftFollower = Spark(1)
    rightLeader = Spark(2)
    rightFollower = Spark(3)
    leftLeader.addFollower(leftFollower)
    rightLeader.addFollower(rightFollower)
    self.drive = DifferentialDrive(leftLeader, leftFollower)
    rightLeader.setInverted(True)


def autonomousInit(self):
    # Set setpoint to current heading at start of auto
    self.heading = self.gyro.getAngle()


def autonomousPeriodic(self):
    error = self.heading - self.gyro.getAngle()
    # Drives forward continuously at half speed, using the gyro to stabilize the␣
↪heading
    self.drive.tankDrive(0.5 + self.kP * error, 0.5 - self.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

```java
// Use gyro declaration from above here
// The gain for a simple P loop
double kP = 0.05;
// 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);
MotorControllerGroup leftMotors = new MotorControllerGroup(left1, left2);
MotorControllerGroup rightMotors = new MotorControllerGroup(right1, right2);
DifferentialDrive drive = new DifferentialDrive(leftMotors, rightMotors);

public Robot() {
    rightMotors.setInverted(true);
}

@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);
}
```

C++

```cpp
// Use gyro declaration from above here
// The gain for a simple P loop
double kP = 0.05;
// Initialize motor controllers and drive
frc::Spark left1{0};
frc::Spark left2{1};
frc::Spark right1{2};
frc::Spark right2{3};
frc::MotorControllerGroup leftMotors{left1, left2};
frc::MotorControllerGroup rightMotors{right1, right2};
frc::DifferentialDrive drive{leftMotors, rightMotors};
Robot::Robot() {
  rightMotors.SetInverted(true);
}
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);
}
```

PYTHON

```
from wpilib import Spark
from wpilib.drive import DifferentialDrive


def robotInit(self):
    # Use gyro declaration from above here
    # The gain for a simple P loop
    self.kP = 0.05
    # Initialize motor controllers and drive
    leftLeader = Spark(0)
    leftFollower = Spark(1)
    rightLeader = Spark(2)
    rightFollower = Spark(3)
    leftLeader.addFollower(leftFollower)
    rightLeader.addFollower(rightFollower)
    self.drive = DifferentialDrive(leftLeader, rightLeader)
    rightLeader.setInverted(True)


def autonomousPeriodic(self):
    # Find the heading error; setpoint is 90
    error = 90 - self.gyro.getAngle()
    # Drives forward continuously at half speed, using the gyro to stabilize the␣
↪heading
    self.drive.tankDrive(self.kP * error, -self.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 drivetrain, 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.

15.3.4 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*

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

```
// Creates a ping-response Ultrasonic object on DIO 1 and 2.
Ultrasonic m_rangeFinder = new Ultrasonic(1, 2);
```

C++

```
// Creates a ping-response Ultrasonic object on DIO 1 and 2.
frc::Ultrasonic m_rangeFinder{1, 2};
```

The measurement can then be retrieved in either inches or millimeters in Java; in C++ the *units library* is used to automatically convert to any desired length unit:

Java

```
  // We can read the distance in millimeters
  double distanceMillimeters = m_rangeFinder.getRangeMM();
  // ... or in inches
  double distanceInches = m_rangeFinder.getRangeInches();
```

C++

```
// We can read the distance
units::meter_t distance = m_rangeFinder.GetRange();
// units auto-convert
units::millimeter_t distanceMillimeters = distance;
units::inch_t distanceInches = distance;
```

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.

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.

Using ultrasonics in code

Ultrasonic sensors are very useful for determining spacing during autonomous routines. For example, the following code from the UltrasonicPID example project (Java, C++) will move the robot to 1 meter away from the nearest object the sensor detects:

Java

```
public class Robot extends TimedRobot {
  // distance the robot wants to stay from an object
  // (one meter)
  static final double kHoldDistanceMillimeters = 1.0e3;

  // proportional speed constant
  private static final double kP = 0.001;
  // integral speed constant
  private static final double kI = 0.0;
  // derivative speed constant
  private static final double kD = 0.0;

  static final int kLeftMotorPort = 0;
  static final int kRightMotorPort = 1;

  static final int kUltrasonicPingPort = 0;
  static final int kUltrasonicEchoPort = 1;

  // Ultrasonic sensors tend to be quite noisy and susceptible to sudden outliers,
  // so measurements are filtered with a 5-sample median filter
  private final MedianFilter m_filter = new MedianFilter(5);

  private final Ultrasonic m_ultrasonic = new Ultrasonic(kUltrasonicPingPort,␣
↪kUltrasonicEchoPort);
  private final PWMSparkMax m_leftMotor = new PWMSparkMax(kLeftMotorPort);
  private final PWMSparkMax m_rightMotor = new PWMSparkMax(kRightMotorPort);
  private final DifferentialDrive m_robotDrive =
      new DifferentialDrive(m_leftMotor::set, m_rightMotor::set);
  private final PIDController m_pidController = new PIDController(kP, kI, kD);

  public Robot() {
    SendableRegistry.addChild(m_robotDrive, m_leftMotor);
    SendableRegistry.addChild(m_robotDrive, m_rightMotor);
  }

  @Override
  public void autonomousInit() {
    // Set setpoint of the pid controller
    m_pidController.setSetpoint(kHoldDistanceMillimeters);
  }
```

(continues on next page)

(continued from previous page)

```
  @Override
  public void autonomousPeriodic() {
    double measurement = m_ultrasonic.getRangeMM();
    double filteredMeasurement = m_filter.calculate(measurement);
    double pidOutput = m_pidController.calculate(filteredMeasurement);

    // disable input squaring -- PID output is linear
    m_robotDrive.arcadeDrive(pidOutput, 0, false);
  }

}
```

C++ (Header)

```
class Robot : public frc::TimedRobot {
 public:
  Robot();
  void AutonomousInit() override;
  void AutonomousPeriodic() override;

  // distance the robot wants to stay from an object
  static constexpr units::millimeter_t kHoldDistance = 1_m;

  static constexpr int kLeftMotorPort = 0;
  static constexpr int kRightMotorPort = 1;
  static constexpr int kUltrasonicPingPort = 0;
  static constexpr int kUltrasonicEchoPort = 1;

 private:
  // proportional speed constant
  static constexpr double kP = 0.001;
  // integral speed constant
  static constexpr double kI = 0.0;
  // derivative speed constant
  static constexpr double kD = 0.0;

  // Ultrasonic sensors tend to be quite noisy and susceptible to sudden
  // outliers, so measurements are filtered with a 5-sample median filter
  frc::MedianFilter<units::millimeter_t> m_filter{5};

  frc::Ultrasonic m_ultrasonic{kUltrasonicPingPort, kUltrasonicEchoPort};
  frc::PWMSparkMax m_left{kLeftMotorPort};
  frc::PWMSparkMax m_right{kRightMotorPort};
  frc::DifferentialDrive m_robotDrive{
      [&](double output) { m_left.Set(output); },
      [&](double output) { m_right.Set(output); }};
  frc::PIDController m_pidController{kP, kI, kD};
};
```

C++ (Source)

```
void Robot::AutonomousInit() {
  // Set setpoint of the pid controller
  m_pidController.SetSetpoint(kHoldDistance.value());
}

void Robot::AutonomousPeriodic() {
  units::millimeter_t measurement = m_ultrasonic.GetRange();
  units::millimeter_t filteredMeasurement = m_filter.Calculate(measurement);
  double pidOutput = m_pidController.Calculate(filteredMeasurement.value());

  // disable input squaring -- PID output is linear
  m_robotDrive.ArcadeDrive(pidOutput, 0, false);
}
```

Additionally, ping-response ultrasonics can be sent to *Shuffleboard*, where they will be displayed with their own widgets:

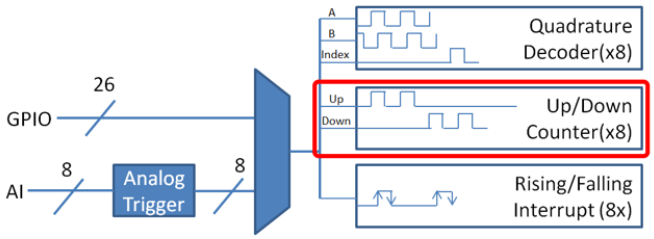
Java

```
    // Add the ultrasonic on the "Sensors" tab of the dashboard
    // Data will update automatically
    Shuffleboard.getTab("Sensors").add(m_rangeFinder);
```

C++

```
  // Add the ultrasonic on the "Sensors" tab of the dashboard
  // Data will update automatically
  frc::Shuffleboard::GetTab("Sensors").Add(m_rangeFinder);
```

15.3.5 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.

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, though 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

```
// Create a new Counter object in two-pulse mode
Counter counter = new Counter(Counter.Mode.k2Pulse);

public Robot() {
    // 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);
}
```

C++

```
// Create a new Counter object in two-pulse mode
frc::Counter counter{frc::Counter::Mode::k2Pulse};
Robot::Robot() {
    // 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

```
// Create a new Counter object in two-pulse mode
Counter counter = new Counter(Counter.Mode.kSemiperiod);

public 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);
}
```

C++

```
// 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

```
// Return the measured pulse width in seconds
counter.getPeriod();
```

C++

```
// 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

```
// Create a new Counter object in two-pulse mode
Counter counter = new Counter(Counter.Mode.kPulseLength);

public Robot() {
    // 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)
}
```

C++

```
// Create a new Counter object in two-pulse mode
frc::Counter counter{frc::Counter::Mode::kPulseLength};
Robot::Robot() {
    // 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

```
// Create a new Counter object in two-pulse mode
Counter counter = new Counter(Counter.Mode.kExternalDirection);

public Robot() {
    // Set up the input channels for the counter
    counter.setUpSource(1);
    counter.setDownSource(2);
    // Set the decoding type to 2X
    counter.setUpSourceEdge(true, true);
}
```

C++

```
// 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

```
// 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);
```

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);
```

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

```
// returns the current count
counter.get();
```

C++

```
// 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

```
// returns the current distance
counter.getDistance();
```

C++

```
// 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

```
// Gets the current rate of the counter
counter.getRate();
```

C++

```
// Gets the current rate of the counter
counter.GetRate();
```

Stopped

Users can obtain whether the counter is stationary with the `getStopped()` method:

JAVA

```
// Gets whether the counter is stopped
counter.getStopped();
```

C++

```
// 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

```
// Gets the last direction in which the counter moved
counter.getDirection();
```

C++

```
// 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

```
// returns the current period in seconds
counter.getPeriod();
```

C++

```
// returns the current period in seconds
counter.GetPeriod();
```

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

```
// Resets the encoder to read a distance of zero
counter.reset();
```

C++

```
// Resets the encoder to read a distance of zero
counter.Reset();
```

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*.

15.3.6 Encoders - Software

Note

This section covers encoders in software. For a hardware guide to encoders, see *Encoders - Hardware*.

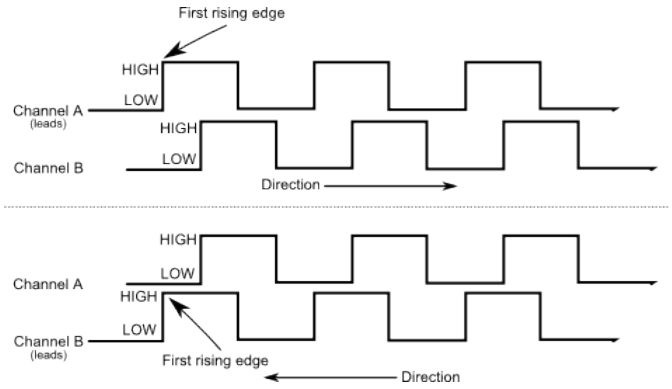
Encoders are devices used to measure motion (usually, the rotation of a shaft).

Important

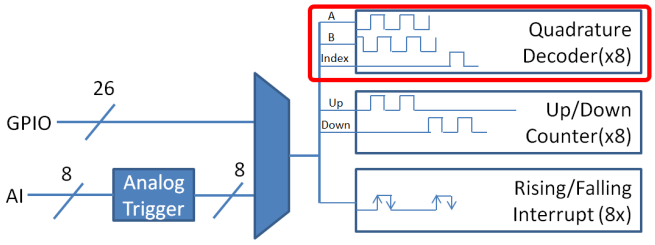
The classes in this document are only used for encoders that are plugged directly into the roboRIO! Please reference the appropriate vendors' documentation for using encoders plugged into motor controllers.

Quadrature Encoders - The `Encoder` Class

WPILib provides support for quadrature encoders through the `Encoder` class (Java, C++). This class provides a simple API for configuring and reading data from 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 quadrature 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.

Examples of quadrature encoders:

- AMT103-V available through FIRST Choice

- CIMcoder
- CTRE Mag Encoder
- Grayhill 63r
- REV Through Bore Encoder
- US Digital E4T

Initializing a Quadrature Encoder

A quadrature encoder can be instantiated as follows:

JAVA

```
// Initializes an encoder on DIO pins 0 and 1
// Defaults to 4X decoding and non-inverted
Encoder m_encoder = new Encoder(0, 1);
```

C++

```
// Initializes an encoder on DIO pins 0 and 1
// Defaults to 4X decoding and non-inverted
frc::Encoder m_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

```
// Initializes an encoder on DIO pins 0 and 1
// 2X encoding and non-inverted
Encoder m_encoder2x = new Encoder(0, 1, false, Encoder.EncodingType.k2X);
```

C++

```
// Initializes an encoder on DIO pins 0 and 1
// 2X encoding and non-inverted
frc::Encoder m_encoder2x{0, 1, false, frc::Encoder::EncodingType::k2X};
```

Configuring Quadrature 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

```
// Configures the encoder to return a distance of 4 for every 256 pulses
// Also changes the units of getRate
m_encoder.setDistancePerPulse(4.0 / 256.0);
// Configures the encoder to consider itself stopped after .1 seconds
m_encoder.setMaxPeriod(0.1);
// Configures the encoder to consider itself stopped when its rate is below 10
m_encoder.setMinRate(10);
// Reverses the direction of the encoder
m_encoder.setReverseDirection(true);
// Configures an encoder to average its period measurement over 5 samples
// Can be between 1 and 127 samples
m_encoder.setSamplesToAverage(5);
```

C++

```
// Configures the encoder to return a distance of 4 for every 256 pulses
// Also changes the units of getRate
m_encoder.SetDistancePerPulse(4.0 / 256.0);
// Configures the encoder to consider itself stopped after .1 seconds
m_encoder.SetMaxPeriod(0.1_s);
// Configures the encoder to consider itself stopped when its rate is below
// 10
m_encoder.SetMinRate(10);
// Reverses the direction of the encoder
m_encoder.SetReverseDirection(true);
// Configures an encoder to average its period measurement over 5 samples
// Can be between 1 and 127 samples
m_encoder.SetSamplesToAverage(5);
```

Reading information from Quadrature 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

```
// Gets the distance traveled
m_encoder.getDistance();
```

C++

```
// Gets the distance traveled
m_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

```
// Gets the current rate of the encoder
m_encoder.getRate();
```

C++

```
// Gets the current rate of the encoder
m_encoder.GetRate();
```

Stopped

Users can obtain whether the encoder is stationary with the `getStopped()` method:

JAVA

```
// Gets whether the encoder is stopped
m_encoder.getStopped();
```

C++

```
// Gets whether the encoder is stopped
m_encoder.GetStopped();
```

Direction

Users can obtain the direction in which the encoder last moved with the `getDirection()` method:

JAVA

```
    // Gets the last direction in which the encoder moved
    m_encoder.getDirection();
```

C++

```
    // Gets the last direction in which the encoder moved
    m_encoder.GetDirection();
```

Period

Users can obtain the period of the encoder pulses (in seconds) with the `getPeriod()` method:

JAVA

```
    // Gets the current period of the encoder
    m_encoder.getPeriod();
```

C++

```
    // Gets the current period of the encoder
    m_encoder.GetPeriod();
```

Resetting a Quadrature Encoder

To reset a quadrature 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

```
    // Resets the encoder to read a distance of zero
    m_encoder.reset();
```

C++

```
  // Resets the encoder to read a distance of zero
  m_encoder.Reset();
```

Duty Cycle Encoders - The `DutyCycleEncoder` class

WPILib provides support for duty cycle (also marketed as *PWM*) encoders through the `DutyCycleEncoder` class (Java, C++). This class provides a simple API for configuring and reading data from duty cycle encoders.

The roboRIO's FPGA handles duty cycle encoders automatically.

> **Warning**
>
> In 2025 the API changed to remove rollover detection as rollover detection did not work. The `get()` method returns the value within a rotation where the maximum value in a rotation is defined in the constructor (default 1).

Examples of duty cycle encoders:

- AndyMark Mag Encoder
- CTRE Mag Encoder
- REV Through Bore Encoder
- Team 221 Lamprey2
- US Digital MA3

Initializing a Duty Cycle Encoder

A duty cycle encoder can be instantiated as follows:

JAVA

```
// Initializes a duty cycle encoder on DIO pins 0
DutyCycleEncoder m_encoder = new DutyCycleEncoder(0);
```

C++

```
// Initializes a duty cycle encoder on DIO pins 0
frc::DutyCycleEncoder m_encoder{0};
```

Configuring Duty Cycle Encoder Range and Zero

> **Note**
>
> The `DutyCycleEncoder` class does not assume specific units of rotation. It returns values in the same units used to calculate the full range of rotation, giving users complete control over the rotation units.

The `DutyCycleEncoder` class provides an alternate constructor that allows control over the full range and the zero position of the encoder.

The zero position is useful for ensuring that the measured rotation corresponds to the desired physical measurement. Unlike quadrature encoders, duty cycle encoders don't need to be homed. The desired rotation can be read and stored to be set when the program starts. The *Preferences class* provides methods to save and retrieve these values on the roboRIO.

JAVA

```
// Initializes a duty cycle encoder on DIO pins 0 to return a value of 4 for
// a full rotation, with the encoder reporting 0 half way through rotation (2
// out of 4)
DutyCycleEncoder m_encoderFR = new DutyCycleEncoder(0, 4.0, 2.0);
```

C++

```
// Initializes a duty cycle encoder on DIO pins 0 to return a value of 4 for
// a full rotation, with the encoder reporting 0 half way through rotation (2
// out of 4)
frc::DutyCycleEncoder m_encoderFR{0, 4.0, 2.0};
```

Reading Rotation from Duty Cycle Encoders

> **Note**
>
> Duty Cycle encoders measure absolute rotation. It does not depend on the starting position of the encoder.

Users can obtain the rotation measured by the encoder with the `get()` method:

JAVA

```
    // Gets the rotation
    m_encoder.get();
```

C++

```
    // Gets the rotation
    m_encoder.Get();
```

Detecting a Duty Cycle Encoder is Connected

As duty cycle encoders output a continuous set of pulses, it is possible to detect that the encoder has been unplugged.

JAVA

```
    // Gets if the encoder is connected
    m_encoder.isConnected();
```

C++

```
    // Gets if the encoder is connected
    m_encoder.IsConnected();
```

Analog Encoders - The `AnalogEncoder` Class

WPILib provides support for analog absolute encoders through the `AnalogEncoder` class (Java, C++). This class provides a simple API for configuring and reading data from analog encoders.

Examples of analog encoders:

- Team 221 Lamprey2
- Thrifty Absolute Magnetic Encoder
- US Digital MA3

Initializing an Analog Encoder

An analog encoder can be instantiated as follows:

JAVA

```
// Initializes an analog encoder on Analog Input pin 0
AnalogEncoder m_encoder = new AnalogEncoder(0);
```

C++

```
// Initializes an analog encoder on Analog Input pin 0
frc::AnalogEncoder m_encoder{0};
```

Configuring Analog Encoder Range and Zero

Note

The `AnalogEncoder` class makes no assumptions about rotation units, returning values in the same units used to calculate the full range. This gives users complete control over the choice of rotation units.

The `AnalogEncoder` class offers an alternate constructor that offers control over the full range of rotation and zero position of the encoder.

The zero position is useful for ensuring that the measured rotation corresponds to the desired physical measurement. Unlike quadrature encoders, analog encoders don't need to be homed. The desired rotation can be read and stored to be set when the program starts. The *Preferences class* provides methods to save and retrieve these values on the roboRIO.

JAVA

```
// Initializes an analog encoder on DIO pins 0 to return a value of 4 for
// a full rotation, with the encoder reporting 0 half way through rotation (2
// out of 4)
AnalogEncoder m_encoderFR = new AnalogEncoder(0, 4.0, 2.0);
```

C++

```
// Initializes an analog encoder on DIO pins 0 to return a value of 4 for
// a full rotation, with the encoder reporting 0 half way through rotation (2
// out of 4)
frc::AnalogEncoder m_encoderFR{0, 4.0, 2.0};
```

Reading Rotation from Analog Encoders

> **Note**
>
> Analog encoders measure absolute rotation. It does not depend on the starting position of the encoder.

Users can obtain the rotation measured by the encoder with the `get()` method:

JAVA

```
  // Gets the rotation
  m_encoder.get();
```

C++

```
  // Gets the rotation
  m_encoder.Get();
```

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 an example is provided below:

Driving to a Distance

Encoders can be used on a robot drive to create a simple "drive to distance" routine. This is useful in autonomous mode, but has the disadvantage that the robot's momentum will cause it to overshoot the intended distance. Better methods include using a *PID Controller* or using *Path Planning*

Note

The following example uses the *Encoder* class, but is similar if other *DutyCycleEncoder* or *AnalogEncoder* is used. However, quadrature encoders are typically better suited for drivetrains since they roll over many times and don't have an absolute position.

JAVA

```
  // Creates an encoder on DIO ports 0 and 1
  Encoder m_encoder = new Encoder(0, 1);
  // Initialize motor controllers and drive
  Spark m_leftLeader = new Spark(0);
  Spark m_leftFollower = new Spark(1);
  Spark m_rightLeader = new Spark(2);
  Spark m_rightFollower = new Spark(3);
  DifferentialDrive m_drive = new DifferentialDrive(m_leftLeader::set, m_
↪rightLeader::set);

  /** Called once at the beginning of the robot program. */
  public Robot() {
    // Configures the encoder's distance-per-pulse
    // The robot moves forward 1 foot per encoder rotation
    // There are 256 pulses per encoder rotation
    m_encoder.setDistancePerPulse(1.0 / 256.0);
    // Invert the right side of the drivetrain. You might have to invert the other␣
↪side
    m_rightLeader.setInverted(true);
    // Configure the followers to follow the leaders
    m_leftLeader.addFollower(m_leftFollower);
    m_rightLeader.addFollower(m_rightFollower);
  }

  /** Drives forward at half speed until the robot has moved 5 feet, then stops. */
  @Override
  public void autonomousPeriodic() {
    if (m_encoder.getDistance() < 5.0) {
      m_drive.tankDrive(0.5, 0.5);
    } else {
      m_drive.tankDrive(0.0, 0.0);
    }
  }
```

C++

```
  // Creates an encoder on DIO ports 0 and 1.
  frc::Encoder m_encoder{0, 1};
  // Initialize motor controllers and drive
  frc::Spark leftLeader{0};
  frc::Spark leftFollower{1};
  frc::Spark rightLeader{2};
  frc::Spark rightFollower{3};
  frc::DifferentialDrive drive{[&](double output) { leftLeader.Set(output); },
```

(continues on next page)

(continued from previous page)

```
                                [&](double output) { rightLeader.Set(output); }};
  Robot() {
    // Configures the encoder's distance-per-pulse
    // The robot moves forward 1 foot per encoder rotation
    // There are 256 pulses per encoder rotation
    m_encoder.SetDistancePerPulse(1.0 / 256.0);
    // Invert the right side of the drivetrain. You might have to invert the
    // other side
    rightLeader.SetInverted(true);
    // Configure the followers to follow the leaders
    leftLeader.AddFollower(leftFollower);
    rightLeader.AddFollower(rightFollower);
  }
  void AutonomousPeriodic() override {
    // Drives forward at half speed until the robot has moved 5 feet, then
    // stops:
    if (m_encoder.GetDistance() < 5) {
      drive.TankDrive(0.5, 0.5);
    } else {
      drive.TankDrive(0, 0);
    }
  }
```

Homing a Mechanism

Since quadrature 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:

Note

Homing is not necessary for absolute encoders like duty cycle encoders and analog encoders.

JAVA

```
  Encoder m_encoder = new Encoder(0, 1);
  Spark m_spark = new Spark(0);
  // Limit switch on DIO 2
  DigitalInput m_limit = new DigitalInput(2);

  /**
   * Runs the motor backwards at half speed until the limit switch is pressed then␣
↪turn off the
   * motor and reset the encoder.
   */
  @Override
  public void autonomousPeriodic() {
    if (!m_limit.get()) {
      m_spark.set(-0.5);
```

(continues on next page)

(continued from previous page)

```
    } else {
      m_spark.set(0.0);
      m_encoder.reset();
    }
  }
```

C++

```
  frc::Encoder m_encoder{0, 1};
  frc::Spark m_spark{0};
  // Limit switch on DIO 2
  frc::DigitalInput m_limit{2};
  void AutonomousPeriodic() override {
    // Runs the motor backwards at half speed until the limit switch is pressed
    // then turn off the motor and reset the encoder
    if (!m_limit.Get()) {
      m_spark.Set(-0.5);
    } else {
      m_spark.Set(0);
      m_encoder.Reset();
    }
  }
```

15.3.7 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.

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:

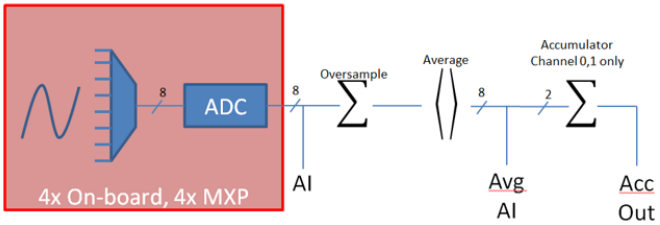
JAVA

```
// Initializes an AnalogInput on port 0
AnalogInput m_analog = new AnalogInput(0);
```

C++

```
// Initializes an AnalogInput on port 0
frc::AnalogInput m_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

```
  // 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.
  m_analog.setOversampleBits(4);
```

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.
m_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

```
// Sets the AnalogInput to 4-bit averaging.  16 samples will be averaged together.
// The update rate will decrease by a factor of 16.
m_analog.setAverageBits(4);
```

C++

```
// Sets the AnalogInput to 4-bit averaging.  16 samples will be averaged
// together. The update rate will decrease by a factor of 16.
m_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

```
    // Gets the raw instantaneous measured value from the analog input, without
    // applying any calibration and ignoring oversampling and averaging
    // settings.
    m_analog.getValue();
```

C++

```
    // Gets the raw instantaneous measured value from the analog input, without
    // applying any calibration and ignoring oversampling and averaging
    // settings.
    m_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

```
    // Gets the instantaneous measured voltage from the analog input.
    // Oversampling and averaging settings are ignored
    m_analog.getVoltage();
```

C++

```
    // Gets the instantaneous measured voltage from the analog input.
    // Oversampling and averaging settings are ignored
    m_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

```
    // Gets the averaged value from the analog input.  The value is not
    // rescaled, but oversampling and averaging are both applied.
    m_analog.getAverageValue();
```

C++

```
    // Gets the averaged value from the analog input.  The value is not
    // rescaled, but oversampling and averaging are both applied.
    m_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

```
    // Gets the averaged value from the analog input.  The value is not
    // rescaled, but oversampling and averaging are both applied.
    m_analog.getAverageValue();
```

C++

```
    // Gets the averaged voltage from the analog input.  Rescaling,
    // oversampling, and averaging are all applied.
    m_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

```
    // Gets the instantaneous measured voltage from the analog input.
    // Oversampling and averaging settings are ignored
    m_analog.getVoltage();
```

C++

```
    // Sets the initial value of the accumulator to 0
    // This is the "starting point" from which the value will change over time
    m_analog.SetAccumulatorInitialValue(0);
    // Sets the "center" of the accumulator to 0.  This value is subtracted from
    // all measured values prior to accumulation.
    m_analog.SetAccumulatorCenter(0);
    // Returns the number of accumulated samples since the accumulator was last
    // started/reset
    m_analog.GetAccumulatorCount();
    // Returns the value of the accumulator.  Return type is long.
    m_analog.GetAccumulatorValue();
    // Resets the accumulator to the initial value
    m_analog.ResetAccumulator();
```

Obtaining synchronized count and value

Sometimes, it is necessarily to obtain matched measurements of the count and the value. This can be done using the `getAccumulatorOutput` method:

JAVA

```
  // Instantiate an AccumulatorResult object to hold the matched measurements
  AccumulatorResult m_result = new AccumulatorResult();
    // Fill the AccumulatorResult with the matched measurements
    m_analog.getAccumulatorOutput(m_result);
    // Read the values from the AccumulatorResult
    long count = m_result.count;
    long value = m_result.value;
```

C++

```
// The count and value variables to fill
int64_t count;
int64_t value;
  // Fill the count and value variables with the matched measurements
  m_analog.GetAccumulatorOutput(count, value);
```

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.

15.3.8 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.

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

```
  // 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 m_pot = new AnalogPotentiometer(0, 180, 30);
```

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.
  frc::AnalogPotentiometer m_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

```
  // Initializes an AnalogInput on port 1
  AnalogInput m_input = new AnalogInput(0);
  // 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 m_pot1 = new AnalogPotentiometer(m_input, 180, 30);
```

C++

```
// Initializes an AnalogInput on port 1
frc::AnalogInput m_input{1};
// 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 m_pot1{&m_input, 180, 30};
```

Reading values from the AnalogPotentiometer

The scaled value can be read by simply calling the `get` method:

JAVA

```
  // Get the value of the potentiometer
  m_pot.get();
```

C++

```
  // Get the value of the potentiometer
  m_pot.Get();
```

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.

15.3.9 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++).

The DigitalInput class

A `DigitalInput` can be initialized as follows:

JAVA

```
// Initializes a DigitalInput on DIO 0
DigitalInput m_input = new DigitalInput(0);
```

C++

```
// Initializes a DigitalInput on DIO 0
frc::DigitalInput m_input{0};
```

Reading the value of the DigitalInput

The state of the `DigitalInput` can be polled with the `get` method:

JAVA

```
// Gets the value of the digital input.  Returns true if the circuit is open.
m_input.get();
```

C++

```
// Gets the value of the digital input.  Returns true if the circuit is
// open.
m_input.Get();
```

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

```
// Initializes an AnalogTrigger on port 0
AnalogTrigger m_trigger0 = new AnalogTrigger(0);
// Initializes an AnalogInput on port 1
AnalogInput m_input = new AnalogInput(1);

// Initializes an AnalogTrigger using the above input
AnalogTrigger m_trigger1 = new AnalogTrigger(m_input);
```

C++

```
// Initializes an AnalogTrigger on port 0
frc::AnalogTrigger trigger0{0};
// Initializes an AnalogInput on port 1
frc::AnalogInput input{1};

// 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

```
   // Enables 2-bit oversampling
   m_input.setAverageBits(2);
   // Sets the trigger to enable at a raw value of 3500, and disable at a value of␣
↪1000
   m_trigger0.setLimitsRaw(1000, 3500);
   // Sets the trigger to enable at a voltage of 4 volts, and disable at a value of␣
↪1.5 volts
   m_trigger0.setLimitsVoltage(1.5, 4);
```

C++

```
    // Enables 2-bit oversampling
    input.SetAverageBits(2);
    // Sets the trigger to enable at a raw value of 3500, and disable at a value
    // of 1000
    trigger0.SetLimitsRaw(1000, 3500);
    // Sets the trigger to enable at a voltage of 4 volts, and disable at a
    // value of 1.5 volts
    trigger0.SetLimitsVoltage(1.5, 4);
```

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. For an example of this, see *Programming Limit Switches*.

Homing a mechanism

Limit switches are very important for being able to "home" a mechanism with an encoder. For an example of this, see *Homing a Mechanism*.

15.3.10 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. A 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*.

Controlling a Motor with Two Limit Switches

JAVA

```
  DigitalInput m_toplimitSwitch = new DigitalInput(0);
  DigitalInput m_bottomlimitSwitch = new DigitalInput(1);
  PWMVictorSPX m_motor = new PWMVictorSPX(0);
  Joystick m_joystick = new Joystick(0);

  @Override
  public void teleopPeriodic() {
    setMotorSpeed(m_joystick.getRawAxis(2));
  }

  /**
   * Sets the motor speed based on joystick input while respecting limit switches.
   *
   * @param speed the desired speed of the motor, positive for up and negative for␣
↪down
   */
  public void setMotorSpeed(double speed) {
    if (speed > 0) {
      if (m_toplimitSwitch.get()) {
        // We are going up and top limit is tripped so stop
        m_motor.set(0);
      } else {
        // We are going up but top limit is not tripped so go at commanded speed
        m_motor.set(speed);
      }
    } else {
      if (m_bottomlimitSwitch.get()) {
        // We are going down and bottom limit is tripped so stop
        m_motor.set(0);
      } else {
        // We are going down but bottom limit is not tripped so go at commanded speed
        m_motor.set(speed);
      }
    }
  }
```

C++

```
  frc::DigitalInput m_toplimitSwitch{0};
  frc::DigitalInput m_bottomlimitSwitch{1};
  frc::PWMVictorSPX m_motor{0};
  frc::Joystick m_joystick{0};
  void TeleopPeriodic() override { SetMotorSpeed(m_joystick.GetRawAxis(2)); }
  void SetMotorSpeed(double speed) {
    if (speed > 0) {
      if (m_toplimitSwitch.Get()) {
        // We are going up and top limit is tripped so stop
        m_motor.Set(0);
      } else {
        // We are going up but top limit is not tripped so go at commanded speed
        m_motor.Set(speed);
```

(continues on next page)

(continued from previous page)

```
    }
  } else {
    if (m_bottomlimitSwitch.Get()) {
      // We are going down and bottom limit is tripped so stop
      m_motor.Set(0);
    } else {
      // We are going down but bottom limit is not tripped so go at commanded
      // speed
      m_motor.Set(speed);
    }
  }
}
```

15.4 Miscellaneous Hardware APIs

This section highlights miscellaneous hardware APIs that are standalone.

15.4.1 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, WS2812B, and WS2815 LEDs with their data pin connected via *PWM*.

Note

LEDs can be controlled through this API while the robot is disabled.

Important

The roboRIO can only control **one** `AddressableLED` object at a time through its PWM ports. Attempting to create multiple `AddressableLED` objects will result in a HAL allocation error. If you need to control multiple physical LED strips, you have several options:

- **Daisy-chain strips in series**: Connect multiple LED strips end-to-end as a single long strip, then use *buffer views* to control different sections independently
- **Use PWM Y-cables**: If you need identical patterns on multiple strips, use PWM Y-cables to send the same signal to multiple strips simultaneously

See also

For detailed information about powering and best practices for addressable LEDs, see the Adafruit NeoPixel Überguide.

> **Warning**
>
> WS2812B LEDs are designed for 5V, but roboRIO PWM/Servo ports output 6V. While the LEDs will function, this may reduce their lifespan. Consider using a voltage regulator or level shifter if longevity is a concern.

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, which can be done with the `setLength()` function.

> **Warning**
>
> 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

```
32  /** Called once at the beginning of the robot program. */
33  public Robot() {
34    // PWM port 9
35    // Must be a PWM header, not MXP or DIO
36    m_led = new AddressableLED(9);
37
38    // Reuse buffer
39    // Default to a length of 60, start empty output
40    // Length is expensive to set, so only set it once, then just update data
41    m_ledBuffer = new AddressableLEDBuffer(60);
42    m_led.setLength(m_ledBuffer.getLength());
43
44    // Set the data
45    m_led.setData(m_ledBuffer);
46    m_led.start();
47  }
```

C++

```
 private:
  static constexpr int kLength = 60;

  // PWM port 9
  // Must be a PWM header, not MXP or DIO
  frc::AddressableLED m_led{9};
  std::array<frc::AddressableLED::LEDData, kLength>
      m_ledBuffer;  // Reuse the buffer

  // Our LED strip has a density of 120 LEDs per meter
```

```
Robot::Robot() {
  // Default to a length of 60, start empty output
  // Length is expensive to set, so only set it once, then just update data
  m_led.SetLength(kLength);
  m_led.SetData(m_ledBuffer);
  m_led.Start();
}
```

Controlling Sections of an LED Strip

The roboRIO can only control a single addressable LED output at a time, but there are often multiple physical LED strips daisy-chained around a robot, or a single flexible LED strip wrapped around structures on a robot. Individual sections can be accessed in Java using `AddressableLEDBufferView`. Buffer views behave like subsections of the larger buffer, and can be accessed using indices in the typical [0, length) range. They can also be reversed, to allow for parallel serpentine sections to be animated in the same physical orientation (i.e. both sections would animate "forward" in the same direction, even if the strips are physically tip-to-tail).

Java

```
// Create the buffer
AddressableLEDBuffer m_buffer = new AddressableLEDBuffer(120);

// Create the view for the section of the strip on the left side of the robot.
// This section spans LEDs from index 0 through index 59, inclusive.
AddressableLEDBufferView m_left = m_buffer.createView(0, 59);

// The section of the strip on the right side of the robot.
// This section spans LEDs from index 60 through index 119, inclusive.
// This view is reversed to cancel out the serpentine arrangement of the
// physical LED strip on the robot.
AddressableLEDBufferView m_right = m_buffer.createView(60, 119).reversed();
```

C++

```
// Create the buffer
std::array<frc::AddressableLED::LEDData, 120> m_buffer;

// Create the view for the section of the strip on the left side of the robot.
// This section spans LEDs from index 0 through index 59, inclusive.
std::view<frc::AddressableLED::LEDData> m_left =
   std::ranges::take_view(m_buffer, 60);

// The section of the strip on the right side of the robot.
// This section spans LEDs from index 60 through index 119, inclusive.
// This view is reversed to cancel out the serpentine arrangement of the
// physical LED strip on the robot.
std::view<frc::AddressableLED::LEDData> m_right =
   std::ranges::reverse_view(
      std::ranges::drop_view(m_buffer, 60));
```

LED Patterns

The `LEDPattern` API simplifies setting LED data. Rather than needing to manually loop over every LED index, you can apply a pattern object to the data buffer directly. LED patterns are stateless, and can safely be applied to multiple buffers or views.

Java

```
// Create an LED pattern that sets the entire strip to solid red
LEDPattern red = LEDPattern.solid(Color.kRed);

// Apply the LED pattern to the data buffer
red.applyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.setData(m_ledBuffer);
```

C++

```
// Create an LED pattern that sets the entire strip to solid red
LEDPattern red = LEDPattern.Solid(Color::kRed);

// Apply the LED pattern to the data buffer
red.ApplyTo(m_ledBuffer);

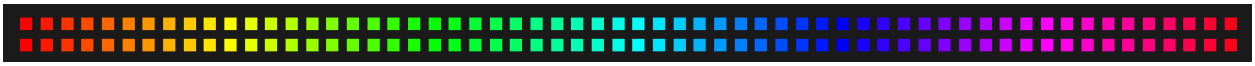
// Write the data to the LED strip
m_led.SetData(m_ledBuffer);
```

Creating a Rainbow Effect

Using the built in `LEDPattern.rainbow` method, we can create a pattern that displays a full rainbow across an entire LED strip. Then, by calling `scrollAtAbsoluteSpeed` we can make it animate and cycle around the strip. `rainbow` accepts two arguments - one for the saturation and one for the value, expressed as a number from 0 to 255.

> **Note**
>
> Animating effects like scrolling use the *Java units library* and the *C++ units library* for speeds and durations.

The base rainbow pattern will look like this:



Java

```
21  // all hues at maximum saturation and half brightness
22  private final LEDPattern m_rainbow = LEDPattern.rainbow(255, 128);
23
24  // Our LED strip has a density of 120 LEDs per meter
25  private static final Distance kLedSpacing = Meters.of(1 / 120.0);
26
27  // Create a new pattern that scrolls the rainbow pattern across the LED strip,␣
    ↪moving at a speed
28  // of 1 meter per second.
29  private final LEDPattern m_scrollingRainbow =
30      m_rainbow.scrollAtAbsoluteSpeed(MetersPerSecond.of(1), kLedSpacing);
```

C++

```
27  // Our LED strip has a density of 120 LEDs per meter
28  units::meter_t kLedSpacing{1 / 120.0};
29
30  // Create an LED pattern that will display a rainbow across
31  // all hues at maximum saturation and half brightness
32  frc::LEDPattern m_rainbow = frc::LEDPattern::Rainbow(255, 128);
33
34  // Create a new pattern that scrolls the rainbow pattern across the LED
35  // strip, moving at a speed of 1 meter per second.
36  frc::LEDPattern m_scrollingRainbow =
37      m_rainbow.ScrollAtAbsoluteSpeed(1_mps, kLedSpacing);
```

Now that the rainbow pattern is defined, we only need to apply it.

Java

```
  public void robotPeriodic() {
    // Update the buffer with the rainbow animation
    m_scrollingRainbow.applyTo(m_ledBuffer);
    // Set the LEDs
    m_led.setData(m_ledBuffer);
  }
}
```

C++

```
void Robot::RobotPeriodic() {
  // Run the rainbow pattern and apply it to the buffer
  m_scrollingRainbow.ApplyTo(m_ledBuffer);
  // Set the LEDs
  m_led.SetData(m_ledBuffer);
}
```



Controlling when patterns are applied

Use commands. The command framework is specifically built for managing when actions run and stop, and prevents multiple actions from running simultaneously.

Java

```
public class LEDSubsystem extends SubsystemBase {
  private static final int kPort = 9;
  private static final int kLength = 120;

  private final AddressableLED m_led;
  private final AddressableLEDBuffer m_buffer;

  public LEDSubsystem() {
    m_led = new AddressableLED(kPort);
    m_buffer = new AddressableLEDBuffer(kLength);
    m_led.setLength(kLength);
    m_led.start();

    // Set the default command to turn the strip off, otherwise the last colors␣
↪written by
    // the last command to run will continue to be displayed.
    // Note: Other default patterns could be used instead!
    setDefaultCommand(runPattern(LEDPattern.solid(Color.kBlack)).withName("Off"));
  }

  @Override
  public void periodic() {
```

(continues on next page)

(continued from previous page)

```
    // Periodically send the latest LED color data to the LED strip for it to display
    m_led.setData(m_buffer);
  }

  /**
   * Creates a command that runs a pattern on the entire LED strip.
   *
   * @param pattern the LED pattern to run
   */
  public Command runPattern(LEDPattern pattern) {
    return run(() -> pattern.applyTo(m_buffer));
  }
}
```

C++

Header:

```
class LEDSubsystem : public SubsystemBase {
 public:
  LEDSubsystem();
  void Periodic() override;

  frc::CommandPtr RunPattern(frc::LEDPattern pattern);

 private:
  static constexpr int kPort = 9;
  static constexpr int kLength = 120;
  frc::AddressableLED m_led{kPort};
  std::array<frc::AddressableLED::LEDData, kLength> m_ledBuffer;
}
```

```
LEDSubsystem::LEDSubsystem() {
  m_led.SetLength(kLength);
  m_led.Start();

  // Set the default command to turn the strip off, otherwise the last colors written␣
↪by
  // the last command to run will continue to be displayed.
  // Note: Other default patterns could be used instead!
  SetDefaultCommand(RunPattern(frc::LEDPattern::Solid(frc::Color::kBlack)).WithName(
↪"Off"));
}

LEDSubsystem::Periodic() {
  // Periodically send the latest LED color data to the LED strip for it to display
  m_led.SetData(m_ledBuffer);
}

frc::CommandPtr LEDSubsystem::RunPattern(frc::LEDPattern pattern) {
  // std::move is necessary for inline pattern declarations to work
  // Otherwise we could have a use-after-free!
  return Run([this, pattern = std::move(pattern)] { pattern.ApplyTo(m_buffer); });
}
```

Basic effects

The basic effects can all be created from the factory methods declared in the `LEDPattern` class

Solid color



The solid color pattern sets the target LED buffer to a single solid color.

Java

```
// Create an LED pattern that sets the entire strip to solid red
LEDPattern red = LEDPattern.solid(Color.kRed);

// Apply the LED pattern to the data buffer
red.applyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.setData(m_ledBuffer);
```

C++

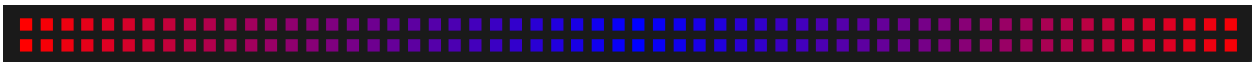
```
// Create an LED pattern that sets the entire strip to solid red
LEDPattern red = LEDPattern.Solid(Color::kRed);

// Apply the LED pattern to the data buffer
red.ApplyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.SetData(m_ledBuffer);
```

Continuous Gradient

The gradient pattern sets the target buffer to display a smooth gradient between the specified colors. The gradient wraps around so scrolling effects can be seamless.



Java

```
// Create an LED pattern that displays a red-to-blue gradient.
// The LED strip will be red at both ends and blue in the center,
// with smooth gradients between
LEDPattern gradient = LEDPattern.gradient(LEDPattern.GradientType.kContinuous, Color.
↪kRed, Color.kBlue);

// Apply the LED pattern to the data buffer
gradient.applyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.setData(m_ledBuffer);
```

C++

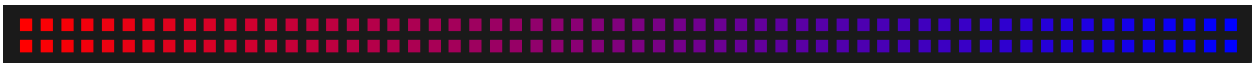
```
// Create an LED pattern that displays a red-to-blue gradient.
// The LED strip will be red at both ends and blue in the center,
// with smooth gradients between
std::array<Color, 2> colors{Color::kRed, Color::kBlue};
LEDPattern gradient = LEDPattern.Gradient(LEDPattern::GradientType::kContinuous,␣
↪colors);

// Apply the LED pattern to the data buffer
gradient.ApplyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.SetData(m_ledBuffer);
```

Discontinuous Gradient

The gradient pattern sets the target buffer to display a smooth gradient between the specified colors. The gradient does not wrap around so it can be used for non-scrolling patterns that don't care about continuity.



Java

```
// Create an LED pattern that displays a red-to-blue gradient.
// The LED strip will be red at one end and blue at the other.
LEDPattern gradient = LEDPattern.gradient(LEDPattern.GradientType.kDiscontinuous,␣
↪Color.kRed, Color.kBlue);

// Apply the LED pattern to the data buffer
gradient.applyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.setData(m_ledBuffer);
```

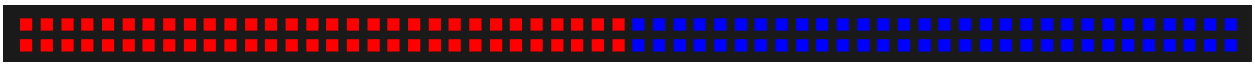
C++

```
// Create an LED pattern that displays a red-to-blue gradient.
// The LED strip will be red at one end and blue at the other.
std::array<Color, 2> colors{Color::kRed, Color::kBlue};
LEDPattern gradient = LEDPattern.Gradient(LEDPattern::GradientType::kDiscontinuous,␣
↪colors);

// Apply the LED pattern to the data buffer
gradient.ApplyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.SetData(m_ledBuffer);
```

Steps



Displays segments of solid colors along the target buffer. This combines well with mask and overlay combination effects.

Steps are specified as a combination of the *starting position* of that color, as a number between 0 (start of the buffer) and 1 (end of the buffer).

Note

If the first step does not start at zero, every LED before that step starts will be set to black - effectively, as if there is a default step of `(0, Color.kBlack)` that can be overwritten.

Java

```
// Create an LED pattern that displays the first half of a strip as solid red,
// and the second half of the strip as solid blue.
LEDPattern steps = LEDPattern.steps(Map.of(0, Color.kRed, 0.5, Color.kBlue));

// Apply the LED pattern to the data buffer
steps.applyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.setData(m_ledBuffer);
```

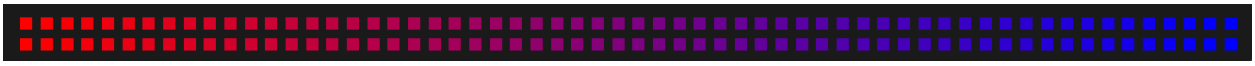
C++

```
// Create an LED pattern that displays the first half of a strip as solid red,
// and the second half of the strip as solid blue.
std::array<std::pair<double, Color>, 2> colorSteps{std::pair{0.0, Color::kRed},
                                                   std::pair{0.5, Color::kBlue}};
LEDPattern steps = LEDPattern.Steps(colorSteps);

// Apply the LED pattern to the data buffer
gradient.ApplyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.SetData(m_ledBuffer);
```

Progress mask



Slightly different from the basic color patterns, the progress mask pattern generates a white-and-black pattern where the white portion is a varying length depending on the value of the value function. This can be combined with another pattern using a *mask* to display a portion of another base pattern depending on the progress of some process - such as the position of a mechanism in its range of motion (eg an elevator's height) or the progress of a PID controller towards its goal.

Java

```
// Create an LED pattern that displays a black-and-white mask that displays the␣
↪current height of an elevator
// mechanism. This can be combined with other patterns to change the displayed color␣
↪to something other than white.
LEDPattern pattern = LEDPattern.progressMaskLayer(() -> m_elevator.getHeight() / m_
↪elevator.getMaxHeight());

// Apply the LED pattern to the data buffer
pattern.applyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.setData(m_ledBuffer);
```

C++

```
// Create an LED pattern that displays a black-and-white mask that displays the␣
↪current height of an elevator
// mechanism. This can be combined with other patterns to change the displayed color␣
↪to something other than white.
LEDPattern pattern = LEDPattern::ProgressMaskLayer([&]() { return m_elevator.
↪GetHeight() / m_elevator.GetMaxHeight(); });

// Apply the LED pattern to the data buffer
```

(continues on next page)

(continued from previous page)

```
pattern.ApplyTo(m_ledBuffer);

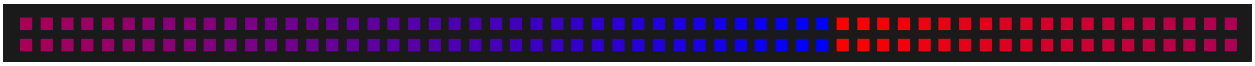
// Write the data to the LED strip
m_led.SetData(m_ledBuffer);
```

Modifying effects

Basic LED patterns can be combined with modifier effects to create new patterns with a combination of effects. Multiple modifiers can be used together to create complex patterns.

> **Note**
>
> The built in animating effects like blinking and scrolling are based on the time returned by `WPIUtilJNI.now()` - in effect, they will play as if they started when the robot booted. Because all built in animation patterns are periodic, this means that the *first* period of a pattern may be truncated at any arbitrary point between 0% and 100%, and every period after that will play normally.

Offset



Offsets can be used to bias patterns forward of backward by a certain number of pixels. Offset patterns will wrap around the end of an LED strip; offset values can be positive (biasing *away* from the start of the strip) or negative (biasing *towards* the start of the strip).

Java

```
// Create an LED pattern that displays a red-to-blue gradient, offset 40 pixels␣
↪forward.
LEDPattern base = LEDPattern.discontinuousGradient(Color.kRed, Color.kBlue);
LEDPattern pattern = base.offsetBy(40);
LEDPattern negative = base.offsetBy(-20); // Equivalent to the above when applied to␣
↪a 60-LED buffer

// Apply the LED pattern to the data buffer
pattern.applyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.setData(m_ledBuffer);
```

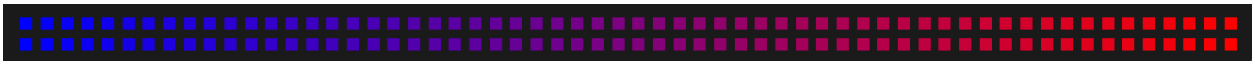
C++

```
// Create an LED pattern that displays a red-to-blue gradient, offset 40 pixels␣
↪forward.
std::array<Color, 2> colors{Color::kRed, Color::kBlue};
LEDPattern base = LEDPattern::DiscontinuousGradient(colors);
LEDPattern pattern = base.OffsetBy(40);
LEDPattern negative = base.OffsetBy(-20); // Equivalent to the above when applied to␣
↪a 60-LED buffer

// Apply the LED pattern to the data buffer
heightDisplay.ApplyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.SetData(m_ledBuffer);
```

Reverse



Patterns and animations can be reversed to flip the direction that patterns are applied in; instead of starting from the lowest-indexed pixel in a buffer or view, a reversed pattern will start from the highest-indexed pixel and move toward the lowest-index pixel. A reversed *scrolling pattern* will scroll in reverse, as if its velocity's sign was flipped.

Java

```
// Create an LED pattern that displays a red-to-blue gradient, then reverse it so it␣
↪displays blue-to-red.
LEDPattern base = LEDPattern.discontinuousGradient(Color.kRed, Color.kBlue);
LEDPattern pattern = base.reversed();

// Apply the LED pattern to the data buffer
pattern.applyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.setData(m_ledBuffer);
```

C++

```
// Create an LED pattern that displays a red-to-blue gradient, then reverse it so it␣
↪displays blue-to-red.
std::array<Color, 2> colors{Color::kRed, Color::kBlue};
LEDPattern base = LEDPattern::DiscontinuousGradient(colors);
LEDPattern pattern = base.Reversed();

// Apply the LED pattern to the data buffer
heightDisplay.ApplyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.SetData(m_ledBuffer);
```

Scroll



Scrolling can be controlled in two different ways: either at a speed as a function of the length of the buffer or view to which it is applied (i.e., the scrolling speed is in terms of percentage per second, or a similar unit), or as a function of the density of the physical LED strips (i.e. scrolling speed is in meters per second, or a similar unit). Relative velocities are particularly useful when a scrolling pattern is applied to different LED strips with different LED spacing (such as one strip with 120 LEDs per meter daisy chained to a second strip with 60 or 144 LEDs per meter), when prototyping before having a particular LED strip in mind (where the density isn't yet known), or when LED strips are quickly changed out. Scrolling at a fixed real-world speed (eg `InchesPerSecond.of(2)`) may be more understandable to readers, but will move faster or slower when applied to an LED strip with a lower or higher pixel density, respectively.

Java

```
// Create an LED pattern that displays a red-to-blue gradient, then scroll at one␣
↪quarter of the LED strip's length per second.
// For a half-meter length of a 120 LED-per-meter strip, this is equivalent to␣
↪scrolling at 12.5 centimeters per second.
Distance ledSpacing = Meters.of(1 / 120.0);
LEDPattern base = LEDPattern.discontinuousGradient(Color.kRed, Color.kBlue);
LEDPattern pattern = base.scrollAtRelativeSpeed(Percent.per(Second).of(25));
LEDPattern absolute = base.scrollAtAbsoluteSpeed(Centimeters.per(Second).of(12.5),␣
↪ledSpacing);

// Apply the LED pattern to the data buffer
pattern.applyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.setData(m_ledBuffer);
```

C++

```
// Create an LED pattern that displays a red-to-blue gradient, then scroll at one␣
↪quarter of the LED strip's length per second.
// For a half-meter length of a 120 LED-per-meter strip, this is equivalent to␣
↪scrolling at 12.5 centimeters per second.
std::array<Color, 2> colors{Color::kRed, Color::kBlue};
LEDPattern base = LEDPattern::DiscontinuousGradient(colors);
LEDPattern pattern = base.ScrollAtRelativeSpeed(units::hertz_t{0.25});
LEDPattern absolute = base.ScrollAtAbsoluteSpeed(0.125_mps, units::meter_t{1/120.0});

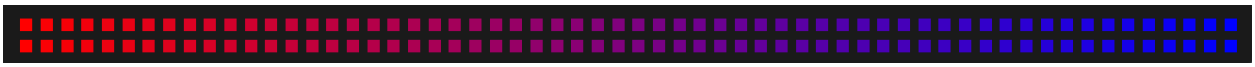
// Apply the LED pattern to the data buffer
heightDisplay.ApplyTo(m_ledBuffer);
```

(continues on next page)

(continued from previous page)

```
// Write the data to the LED strip
m_led.SetData(m_ledBuffer);
```

Breathe



A breathing modifier will make the base pattern brighten and dim in a sinusoidal pattern over the given period of time. Brightness is relative to the original brightness of the base pattern - breathing will only make it dimmer, never brighter than the original.

Java

```
// Create an LED pattern that displays a red-to-blue gradient, breathing at a 2␣
↪second period (0.5 Hz)
LEDPattern base = LEDPattern.discontinuousGradient(Color.kRed, Color.kBlue);
LEDPattern pattern = base.breathe(Seconds.of(2));

// Apply the LED pattern to the data buffer
pattern.applyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.setData(m_ledBuffer);
```

C++

```
// Create an LED pattern that displays a red-to-blue gradient, breathing at a 2␣
↪second period (0.5 Hz)
std::array<Color, 2> colors{Color::kRed, Color::kBlue};
LEDPattern base = LEDPattern::DiscontinuousGradient(colors);
LEDPattern pattern = base.Breathe(2_s);

// Apply the LED pattern to the data buffer
heightDisplay.ApplyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.SetData(m_ledBuffer);
```

Blink



Blinking can be done in one of three ways:

1. Symmetrically, where an equal amount of time is spent in the “on” and “off” states per cycle
2. Asymetrically, where the time spent “on” can be configured independently from the time spent “off”
3. Synchronously, where the time spent on and off is synchronized with an external source (for example, the state of the RSL)

Java

```
// Create an LED pattern that displays a red-to-blue gradient, blinking at various␣
↪rates.
LEDPattern base = LEDPattern.discontinuousGradient(Color.kRed, Color.kBlue);

// 1.5 seconds on, 1.5 seconds off, for a total period of 3 seconds
LEDPattern pattern = base.blink(Seconds.of(1.5));

// 2 seconds on, 1 second off, for a total period of 3 seconds
LEDPattern asymmetric = base.blink(Seconds.of(2), Seconds.of(1));

// Turn the base pattern on when the RSL is on, and off when the RSL is off
LEDPattern sycned = base.synchronizedBlink(RobotController::getRSLState);

// Apply the LED pattern to the data buffer
pattern.applyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.setData(m_ledBuffer);
```

C++

```
// Create an LED pattern that displays a red-to-blue gradient, blinking at various␣
↪rates.
std::array<Color, 2> colors{Color::kRed, Color::kBlue};
LEDPattern base = LEDPattern::DiscontinuousGradient(colors);

// 1.5 seconds on, 1.5 seconds off, for a total period of 3 seconds
LEDPattern pattern = base.Blink(1.5_s);

// 2 seconds on, 1 second off, for a total period of 3 seconds
LEDPattern asymmetric = base.Blink(2_s, 1_s));

// Turn the base pattern on when the RSL is on, and off when the RSL is off
LEDPattern sycned = base.SynchronizedBlink([]() { return RobotController.
↪GetRSLState(); });

// Apply the LED pattern to the data buffer
pattern.ApplyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.SetData(m_ledBuffer);
```

Brightness



Patterns can be brightened and dimmed relative to their original brightness; a brightness value of 100% is identical to the original pattern, a value of 200% is twice as bright, and a value of 0% is completely turned off. This can be useful in a pinch to tone down patterns that are too bright (apologies to the 2024 NE Greater Boston district event staff, who were subjected to a maximimum brightness white flashing pattern with a precursor version of this library before the brightness modifier was added).

> **Note**
>
> For speed, brightness calculations are done naively in the RGB color space instead of HSL/HSV/Lab. This sacrifices accuracy, so large changes in brightness may look undersaturated.

Java

```
// Create an LED pattern that displays a red-to-blue gradient at half brightness
LEDPattern base = LEDPattern.discontinuousGradient(Color.kRed, Color.kBlue);
LEDPattern pattern = base.atBrightness(Percent.of(50));

// Apply the LED pattern to the data buffer
pattern.applyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.setData(m_ledBuffer);
```

C++

```
// Create an LED pattern that displays a red-to-blue gradient at half brightness
std::array<Color, 2> colors{Color::kRed, Color::kBlue};
LEDPattern base = LEDPattern::DiscontinuousGradient(colors);
LEDPattern pattern = base.AtBrightness(0.5);

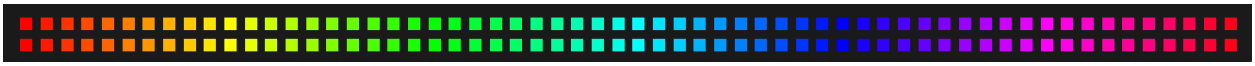
// Apply the LED pattern to the data buffer
pattern.ApplyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.SetData(m_ledBuffer);
```

Combinatory effects

Complex LED patterns are built up from combining simple base patterns (such as solid colors or gradients) with animating effects (such as scrolling or breathing) and combinatory effects (like masks and overlays). Multiple effects can be combined at once, like in the scrolling rainbow effect above that takes a basic base effect - a static rainbow - and then adds a scrolling effect to it.

Mask



Masks work by combining the RGB values of two patterns and keeping only the values that are shared by both. The combination works on the individual bits of each color using a bitwise AND operation - for example, if a pixel's red channel were set to 255 by one pattern (represented as 11111111 in binary), then the output red color would be identical to the red channel of the second pattern. If the first pattern sets it to zero (00000000 in binary), then the output red color would also be zero, regardless of whatever the second pattern sets. For this reason, black (all zeroes) and white (all ones) masks are very useful for selectively enabling and disabling parts of another pattern. Other mask colors can be used as well: masking with solid red would keep only the red channel of the original pattern, while discarding all green and blue values.

Java

```
// Create an LED pattern that displays a red-to-blue gradient at a variable length
// depending on the relative position of the elevator. The blue end of the gradient
// will only be shown when the elevator gets close to its maximum height; otherwise,
// that end will be solid black when the elevator is at lower heights.
LEDPattern base = LEDPattern.discontinuousGradient(Color.kRed, Color.kBlue);
LEDPattern mask = LEDPattern.progressMaskLayer(() -> m_elevator.getHeight() / m_
↪elevator.getMaxHeight());
LEDPattern heightDisplay = base.mask(mask);

// Apply the LED pattern to the data buffer
heightDisplay.applyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.setData(m_ledBuffer);
```

C++

```
// Create an LED pattern that displays a red-to-blue gradient at a variable length
// depending on the relative position of the elevator. The blue end of the gradient
// will only be shown when the elevator gets close to its maximum height; otherwise,
// that end will be solid black when the elevator is at lower heights.
std::array<Color, 2> colors{Color::kRed, Color::kBlue};
LEDPattern base = LEDPattern::DiscontinuousGradient(colors);
LEDPattern mask = LEDPattern::ProgressMaskLayer([&]() { m_elevator.GetHeight() / m_
```

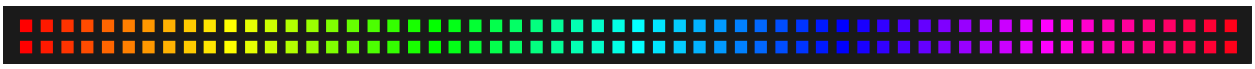
(continues on next page)

(continued from previous page)

```
↪elevator.GetMaxHeight() });
LEDPattern heightDisplay = base.Mask(mask);

// Apply the LED pattern to the data buffer
heightDisplay.ApplyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.SetData(m_ledBuffer);
```



Masks can also be animated (see *progressMask*). Masking a base pattern with a scrolling pattern will result in a panning effect. The animation above was generated by masking a rainbow pattern with a scrolling white/black pattern

Java

```
Map<Double, Color> maskSteps = Map.of(0, Color.kWhite, 0.5, Color.kBlack);
LEDPattern base = LEDPattern.rainbow(255, 255);
LEDPattern mask =
   LEDPattern.steps(maskSteps).scrollAtRelativeSpeed(Percent.per(Second).of(0.25));

LEDPattern pattern = base.mask(mask);

// Apply the LED pattern to the data buffer
pattern.applyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.setData(m_ledBuffer);
```

C++

```
std::array<std::pair<double, Color>, 2> maskSteps{std::pair{0.0, Color::kWhite},
                                                  std::pair{0.5, Color::kBlack}};
LEDPattern base = LEDPattern::Rainbow(255, 255);
LEDPattern mask =
   LEDPattern::Steps(maskSteps).ScrollAtRelativeSpeed(units::hertz_t{0.25});

LEDPattern pattern = base.Mask(mask);

// Apply the LED pattern to the data buffer
pattern.ApplyTo(m_ledBuffer);

// Write the data to the LED strip
m_led.SetData(m_ledBuffer);
```

Overlay

Overlays can be used to "stack" patterns atop each other, where black pixels (set to `Color.kBlack`, RGB value #000000) are treated as transparent and allow a lower layer to be displayed. Upper layers are typically combined with *masks* to set transparent sections; recall that masking a pixel with `Color.kBlack` will *set* that pixel to black, which will then be treated by the overlay as transparent.

Blend

Blends will combine the output colors of patterns together, by averaging out the individual RGB colors for every pixel. Like the *brightness modifier*, this tends to output colors that are more desaturated than its inputs.

Low Level Access

`LEDPattern` is an easy and convenient way of controlling LEDs, but direct access to the LED colors is sometimes needed for custom patterns and animations.

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. Low-level access is typically done with an indexed for-loop that iterates over each LED index of the section to control. This method can be used for both `AddressableLEDBuffer` and `AddressableLEDBufferView` objects in Java, and for `std::span` for C++.

Note

RGB stands for Red, Green, and Blue. This is a fairly common color model as it's quite easy to understand, and it corresponds with a typical LED configuration that's comprised of one red, one green, and one blue sub-LED. 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.

Note

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.

These examples demonstrate setting an entire LED strip to solid red using the RGB and HSV methods:

Java (RGB)

```
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);
```

C++ (RGB)

```
for (int i = 0; i < kLength; i++) {
   m_ledBuffer[i].SetRGB(255, 0, 0);
}
m_led.SetData(m_ledBuffer);
```

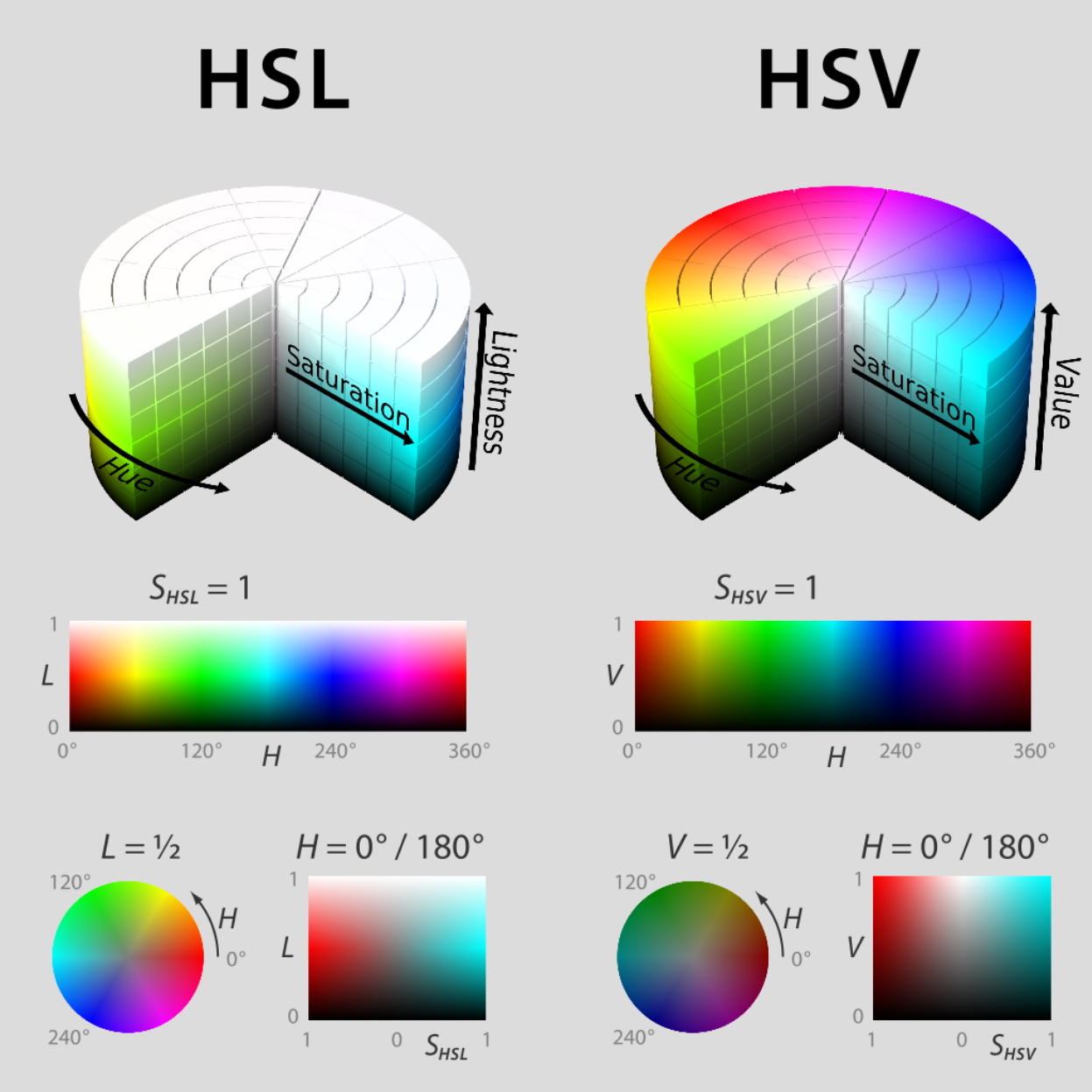
Java (HSV)

```
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);
```

C++ (HSV)

```
for (int i = 0; i < kLength; i++) {
   m_ledBuffer[i].SetHSV(0, 100, 100);
}
m_led.SetData(m_ledBuffer);
```

Using HSV Values



15.5 Motor Controllers

A motor controller is responsible on your robot for making motors move. For brushed DC motors such as the *CIM* or 775, 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!

15.5.1 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 2026:

- Koors40 Motor Controller (P/N am-5600)
- Spark Flex Motor Controller (P/N REV-11-2159, am-5276)
- Spark Motor Controller (P/N: REV-11-1200, am-4260)
- Spark MAX Motor Controller (P/N: REV-11-2158, am-4261)
- Talon FX Motor Controller (P/N 217-6515, 19-708850, am-6515, am-6515_Short, WCP-0940) for controlling integral Falcon 500, Kraken X60, or Kraken X44 only
- Talon FXS Motor Controller (P/N 24-708883, WCP-1692)
- 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)
- Thrifty Nova (P/N TTB-0100)
- Venom Motor with Controller (P/N BDC-10001) for controlling integral motor only
- Victor SP Motor Controller (P/N: 217-9090, am-2855, 14-868380)
- Victor SPX Motor Controller (P/N: 217-9191, 17-868388, am-3748)

15.6 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 *Pneumatics APIs*.

15.6.1 FRC Legal Pneumatics controllers

- Pneumatics Control Module (P/N: am-2858, 217-4243)
- Pneumatic Hub (P/N REV-11-1852)

15.7 Relays

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

15.7.1 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)
- Power Distribution Hub (PDH) switched channel (P/N REV-11-1850)

16 CAN Devices

16.1 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.

16.2 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++, `Python`), and the Solenoids are handled by the `Solenoid` (Java, C++, `Python`) and `DoubleSolenoid` (Java, C++, `Python`) classes.

An additional PCM module can be used where the module's 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*.

16.3 Pneumatic Hub



The Pneumatic Hub (*PH*) is a *CAN*-based device that provides complete control over the compressor and up to 16 solenoids per module. The PH 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++, `Python`), and the Solenoids are handled by the `Solenoid` (Java, C++, `Python`) and `DoubleSolenoid` (Java, C++, `Python`) classes.

An additional PH module can be used where the module's 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*.

16.4 Power Distribution Module

The CTRE Power Distribution Panel (*PDP*) and Rev Power Distribution Hub (*PDH*) can use their *CAN* connectivity to communicate a wealth of status information regarding the robot's power use to the roboRIO, for use in user code. This has the capability to report 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. This data can be used for a number of advanced control techniques, such as motor *torque* limiting and brownout avoidance.

16.4.1 Creating a Power Distribution Object

To use the either Power Distribution module, create an instance of the `PowerDistribution` class (Java, C++, Python). With no arguments, the Power Distribution object will be detected, and must use CAN ID of 0 for CTRE or 1 for REV. If the CAN ID is non-default, additional constructors are available to specify the CAN ID and type.

JAVA

```
PowerDistribution examplePD = new PowerDistribution();
PowerDistribution examplePD = new PowerDistribution(0, ModuleType.kCTRE);
PowerDistribution examplePD = new PowerDistribution(1, ModuleType.kRev);
```

C++

```
PowerDistribution examplePD{};
PowerDistribution examplePD{0, frc::PowerDistribution::ModuleType::kCTRE};
PowerDistribution examplePD{1, frc::PowerDistribution::ModuleType::kRev};
```

PYTHON

```
from wpilib import PowerDistribution
examplePD = PowerDistribution()
examplePD = PowerDistribution(0, PowerDistribution.ModuleType.kCTRE)
examplePD = PowerDistribution(1, PowerDistribution.ModuleType.kRev)
```

Note: it is not necessary to create a PowerDistribution 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.

16.4.2 Reading the Bus Voltage

JAVA

```
    // Get the voltage going into the PDP, in Volts.
    // The PDP returns the voltage in increments of 0.05 Volts.
    double voltage = m_pdp.getVoltage();
    SmartDashboard.putNumber("Voltage", voltage);
```

C++

```
// Get the voltage going into the PDP, in Volts.
// The PDP returns the voltage in increments of 0.05 Volts.
double voltage = m_pdp.GetVoltage();
frc::SmartDashboard::PutNumber("Voltage", voltage);
```

PYTHON

```
        # Get the voltage going into the PDP, in Volts.
        # The PDP returns the voltage in increments of 0.05 Volts.
        voltage = self.pdp.getVoltage()
        wpilib.SmartDashboard.putNumber("Voltage", voltage)
```

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.

16.4.3 Reading the Temperature

JAVA

```
// Retrieves the temperature of the PDP, in degrees Celsius.
double temperatureCelsius = m_pdp.getTemperature();
SmartDashboard.putNumber("Temperature", temperatureCelsius);
```

C++

```
// Retrieves the temperature of the PDP, in degrees Celsius.
double temperatureCelsius = m_pdp.GetTemperature();
frc::SmartDashboard::PutNumber("Temperature", temperatureCelsius);
```

PYTHON

```
        # Retrieves the temperature of the PDP, in degrees Celsius.
        temperatureCelsius = self.pdp.getTemperature()
        wpilib.SmartDashboard.putNumber("Temperature", temperatureCelsius)
```

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.

16.4.4 Reading the Total Current, Power, and Energy

JAVA

```
    // Get the total current of all channels.
    double totalCurrent = m_pdp.getTotalCurrent();
    SmartDashboard.putNumber("Total Current", totalCurrent);

    // Get the total power of all channels.
    // Power is the bus voltage multiplied by the current with the units Watts.
    double totalPower = m_pdp.getTotalPower();
    SmartDashboard.putNumber("Total Power", totalPower);

    // Get the total energy of all channels.
    // Energy is the power summed over time with units Joules.
    double totalEnergy = m_pdp.getTotalEnergy();
    SmartDashboard.putNumber("Total Energy", totalEnergy);
```

C++

```
    // Get the total current of all channels.
    double totalCurrent = m_pdp.GetTotalCurrent();
    frc::SmartDashboard::PutNumber("Total Current", totalCurrent);

    // Get the total power of all channels.
    // Power is the bus voltage multiplied by the current with the units Watts.
    double totalPower = m_pdp.GetTotalPower();
    frc::SmartDashboard::PutNumber("Total Power", totalPower);

    // Get the total energy of all channels.
    // Energy is the power summed over time with units Joules.
    double totalEnergy = m_pdp.GetTotalEnergy();
    frc::SmartDashboard::PutNumber("Total Energy", totalEnergy);
```

PYTHON

```
        # Get the total current of all channels.
        totalCurrent = self.pdp.getTotalCurrent()
        wpilib.SmartDashboard.putNumber("Total Current", totalCurrent)

        # Get the total power of all channels.
        # Power is the bus voltage multiplied by the current with the units Watts.
        totalPower = self.pdp.getTotalPower()
        wpilib.SmartDashboard.putNumber("Total Power", totalPower)

        # Get the total energy of all channels.
        # Energy is the power summed over time with units Joules.
        totalEnergy = self.pdp.getTotalEnergy()
        wpilib.SmartDashboard.putNumber("Total Energy", totalEnergy)
```

Monitoring the total current, power, and energy 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. Power is the bus voltage

multiplied by the current with the units Watts. Energy is the power summed over time with units Joules.

16.4.5 Reading Individual Channel Currents

The PDP/PDH also allows users to monitor the current drawn by the individual device power channels. You can read the current on any of the 16 PDP channels (0-15) or 24 PDH channels (0-23).

JAVA

```
// Get the current going through channel 7, in Amperes.
// The PDP returns the current in increments of 0.125A.
// At low currents the current readings tend to be less accurate.
double current7 = m_pdp.getCurrent(7);
SmartDashboard.putNumber("Current Channel 7", current7);
```

C++

```
// Get the current going through channel 7, in Amperes.
// The PDP returns the current in increments of 0.125A.
// At low currents the current readings tend to be less accurate.
double current7 = m_pdp.GetCurrent(7);
frc::SmartDashboard::PutNumber("Current Channel 7", current7);
```

PYTHON

```
        # Get the current going through channel 7, in Amperes.
        # The PDP returns the current in increments of 0.125A.
        # At low currents the current readings tend to be less accurate.
        current7 = self.pdp.getCurrent(7)
        wpilib.SmartDashboard.putNumber("Current Channel 7", current7)
```

Monitoring individual device current draws can be useful for detecting shorts or stalled motors.

16.4.6 Using the Switchable Channel (PDH)

The REV PDH has one channel that can be switched on or off to control custom circuits.

JAVA

```
examplePD.setSwitchableChannel(true);
examplePD.setSwitchableChannel(false);
```

C++

```
examplePD.SetSwitchableChannel(true);
examplePD.SetSwitchableChannel(false);
```

PYTHON

```
examplePD.setSwitchableChannel(True)
examplePD.setSwitchableChannel(False)
```

16.5 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:

16.5.1 CTR Electronics

CTR Electronics (CTRE) offers several CAN peripherals with external libraries. General resources for all CTRE devices include:

Phoenix Device Software Documentation

CTRE Motor Controllers

- **Talon FX (with Falcon 500 Motor, Kraken x60 motor, and Kraken x44)**
  - API Documentation (v6: Java, C++)
  - Hardware User's Manual (Falcon 500, Kraken x60, Kraken x44)
  - Software Documentation
- **Talon SRX**
  - API Documentation (Java, C++)
  - Hardware User's Manual
  - Software Documentation

- **Victor SPX**
  - API Documentation (Java, C++)
  - Hardware User's Manual
  - Software Documentation

CTRE Sensors

- **CANcoder**
  - API Documentation (v6: Java, C++)
  - Hardware User's Manual
  - Software Documentation
- **Pigeon 2.0**
  - API Documentation (v6: Java, C++)
  - Hardware User's Manual
  - Software Documentation6
- **Pigeon IMU**
  - API Documentation (Java, C++)
  - Hardware User's Manual
  - Software Documentation
- **CANifier**
  - API Documentation (Java, C++)
  - Hardware User's Manual
  - Software Documentation

CTRE Other CAN Devices

- **CANdle LED Controller**
  - API Documentation (Java, C++)
  - Hardware User's Manual
  - Software Documentation

16.5.2 REV Robotics

REV Robotics currently offers the SPARK MAX and SPARK Flex motor controllers which can be used for brushed and REV brushless (NEO, NEO 550, and NEO Vortex) motors.

REV Motor Controllers

- **SPARK MAX**
  - API Documentation (Java, C++)
  - Technical Manual
- **SPARK Flex**
  - API Documentation (Java, C++)
  - Technical Manual

16.5.3 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

16.5.4 Redux Robotics

> **Warning**
>
> Redux Robotics is shutting down. While their devices remain legal for FRC use and software support will be provided for the 2026 season, no new hardware will be available.

Redux Robotics offers the HELIUM Canandmag *CAN* + *PWM* magnetic encoder and the BORON Canandgyro *CAN*-enabled gyro.

Redux Sensors

- **HELIUM Canandmag**
  - API Documentation (Java, C++)
  - Technical Manual
- **BORON Canandgyro**
  - API Documentation (Java, C++)
  - Technical Manual

16.5.5 Grapple Robotics

Grapple Robotics currently offers the LaserCAN *CAN*-enabled range finding sensor

Grapple Sensors

- **LaserCAN**
  - API Documentation (Java, C++)
  - Technical Manual

16.6 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.

16.6.1 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
Distance Sensor	6
Encoder	7
Power Distribution Module	8
Pneumatics Controller	9
Miscellaneous	10
IO Breakout	11
Servo Controller	12
Color Sensor	13
Reserved	14-30
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 `Reserved`pool, 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
The Thrifty Bot	13
Redux Robotics	14
AndyMark	15
Vivid Hosting	16
Vertos Robotics	17
SWYFT Robotics	18
Lumyn Labs	19
Brushland Labs	20
Reserved	21-255

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 a CAN 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

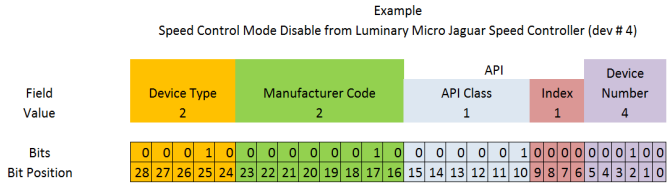
API Index

The API Index is a 4-bit identifier for a particular message within an API Class. An example of the API Index values for a CAN 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.



16.6.2 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).

16.6.3 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.

16.6.4 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 FIRST's Control System Partners to provide such interfaces.

16.6.5 Universal Heartbeat

The roboRIO provides a universal CAN heartbeat that any device on the bus can listen and react to. This heartbeat is sent every 20 ms. The heartbeat has a full CAN ID of `0x01011840` (which is the NI Manufacturer ID, RobotController type, Device ID 0 and API ID `0x061`). It is an 8 byte CAN packet with the following bitfield layout.

Description	Byte	Width (bits)
Match time (seconds)	8	8
Match number	6-7	10
Replay number	6	6
Red alliance	5	1
Enabled	5	1
Autonomous mode	5	1
Test mode	5	1
System watchdog	5	1
Tournament type	5	3
Time of day (year)	4	6
Time of day (month)	3-4	4
Time of day (day)	3	5
Time of day (seconds)	2-3	6
Time of day (minutes)	1-2	6
Time of day (hours)	1	5

```
struct [[gnu::packed]] RobotState {
  uint64_t matchTimeSeconds : 8;
  uint64_t matchNumber : 10;
  uint64_t replayNumber : 6;
  uint64_t redAlliance : 1;
  uint64_t enabled : 1;
```

(continues on next page)

(continued from previous page)

```
  uint64_t autonomous : 1;
  uint64_t testMode : 1;
  uint64_t systemWatchdog : 1;
  uint64_t tournamentType : 3;
  uint64_t timeOfDay_yr : 6;
  uint64_t timeOfDay_month : 4;
  uint64_t timeOfDay_day : 5;
  uint64_t timeOfDay_sec : 6;
  uint64_t timeOfDay_min : 6;
  uint64_t timeOfDay_hr : 5;
};
```

If the `System watchdog` flag is set, motor controllers are enabled. If 100 ms has passed since this packet was received, the robot program can be considered hung, and devices should act as if the robot has been disabled.

Note that all fields except `Enabled`, `Autonomous mode`, `Test mode`, and `System watchdog` will contain invalid values until an arbitrary time after the Driver Station connects.

17

Basic Programming

17.1 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:

- Separation of 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

17.1.1 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

17.1.2 Git Vocabulary

Git revolves around several core data structures and commands:

- **Repository:** the data structure of your code, including a `.git` folder in the root directory
- **Commit:** a particular saved state of the repository, which includes all files and additions
- **Branch:** a means of grouping a set of commits. Each branch has 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:** retrieve a local copy of a repository to modify
- **Fork:** duplicate a pre-existing repository to modify, and to compare against the original
- **Merge:** combine various changes from different branches/commits/forks into a single history

17.1.3 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.

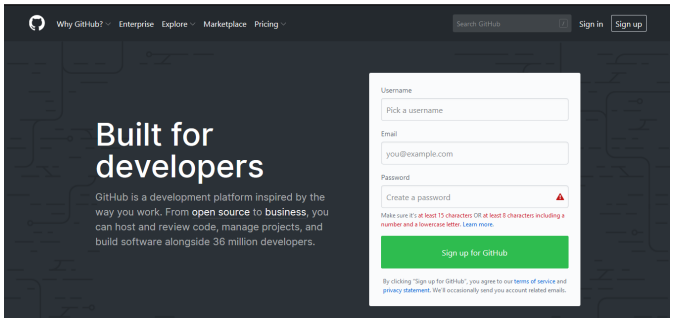
Creating the repository

You can store the repository locally, or through a remote – a remote being the cloud, or possibly another storage medium or server 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.

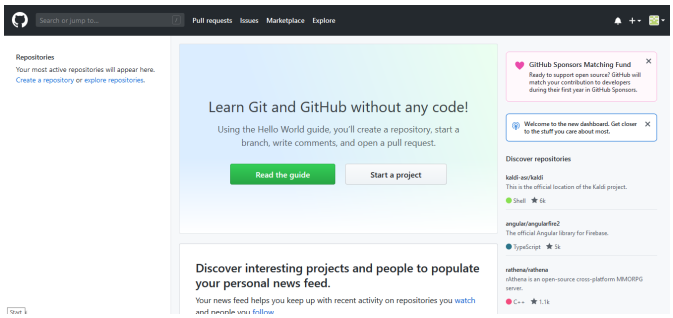
Creating a GitHub Account

Go ahead and create a GitHub account by visiting the website and following the on-screen prompts.

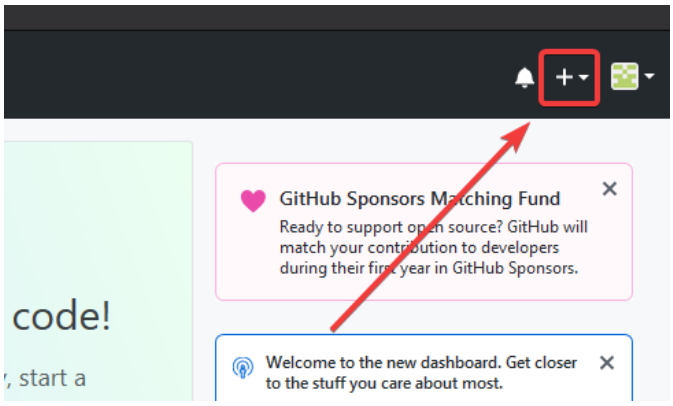


Local Creation

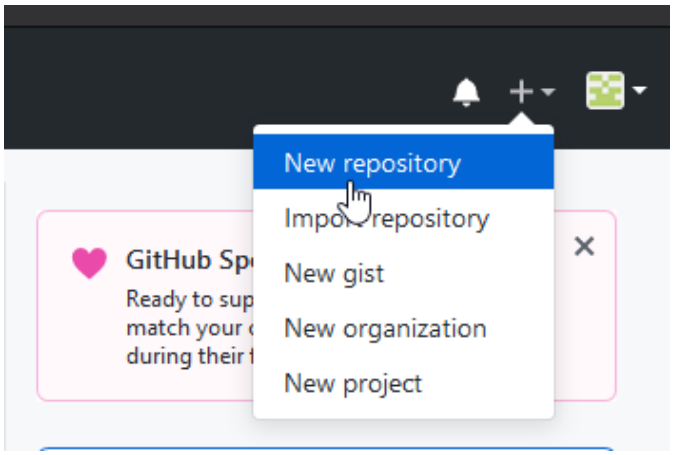
After creating and verifying your account, you'll want to visit the homepage. It'll look similar to the shown image.



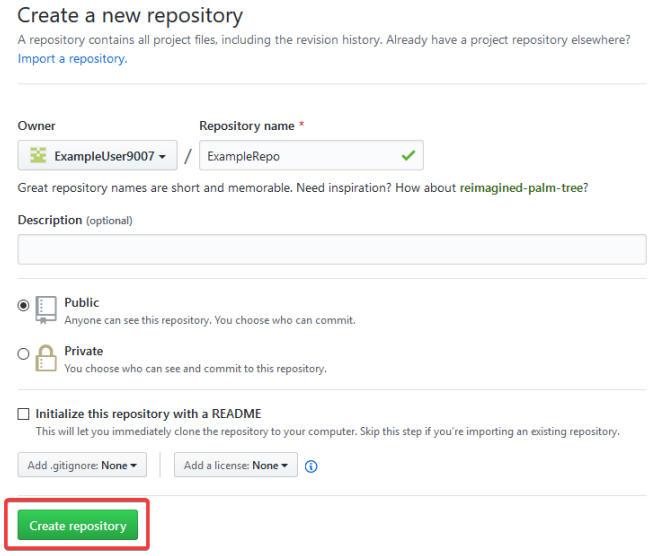
Click the plus icon in the top right.



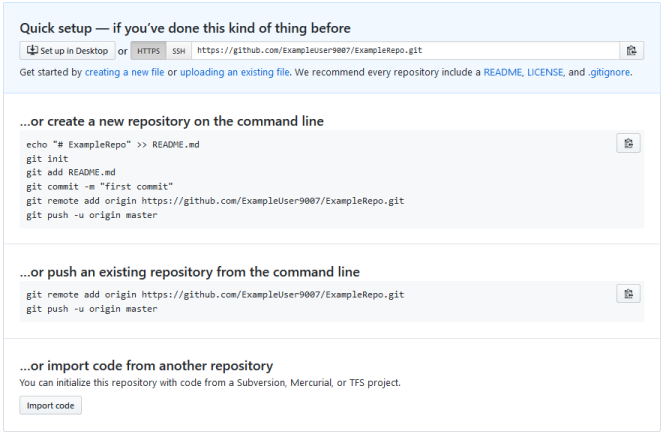
Then click *"New Repository"*



Fill out the appropriate information, and then click *"Create repository"*



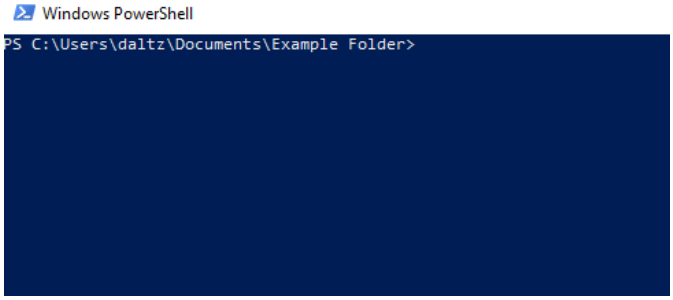
You should see a screen similar to this



Note

The keyboard shortcut `Ctrl+~` can be used to open a terminal in Visual Studio Code for Windows.

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

```
> 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
```

Existing Project

```
> 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
```

17.1.4 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 "stages" the specified file(s) so that they will be included in the next commit.

For a single file, run `git add FILENAME.txt` where FILENAME.txt is the name and extension of the file to add. To add every file/folder that isn't excluded via *gitignore*, run `git add .`. When run in the root of the repository this command will stage every untracked, unexcluded file.

Git Commit

This command creates the commit and stores it locally. This saves the state and adds it to the repository's history. The commit will consist of whatever changes ("diffs") were made to the staged files since the last commit. It is required to specify a "commit message" explaining why you changed this set of files or what the change accomplishes.

Run: `git commit -m "type message here"`

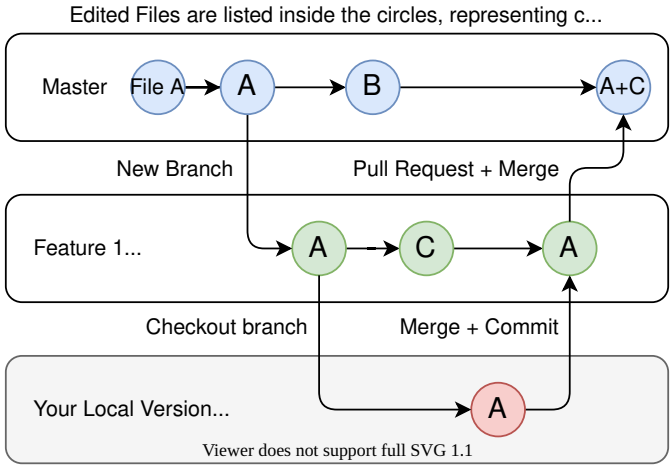
Git Push

Upload (Push) your local changes to the remote (Cloud)

Run: `git push`

17.1.5 Branches

Branches in Git are similar to parallel worlds. 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

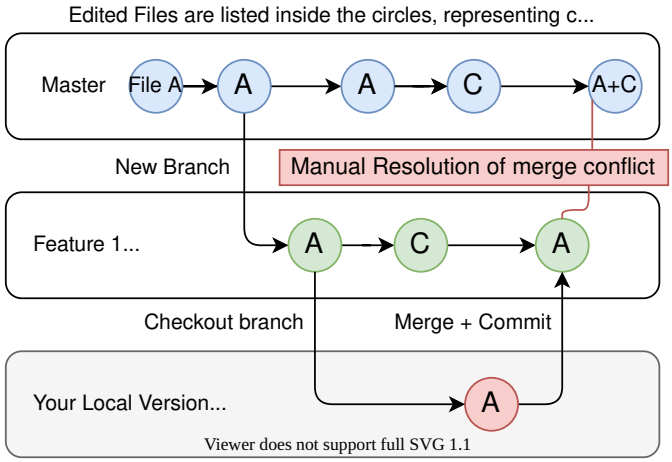
Once a branch is created, you have to then enter the branch.

Run: `git checkout branch-name` where branch-name is the branch that was previously created.

17.1.6 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 into 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 into your local copy. That merge may look like the below.



However, in the above example, what if File A was modified by both branch Feature1 and Feature2? This is called a **merge conflict**. A merge conflict can be resolved by editing the conflicting file. In the example, we would need to edit File A to keep the history or changes that we want. After that has been done, simply re-add, re-commit, and then push your changes.

17.1.7 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`.

17.1.8 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 *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`

17.1.9 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`

17.1.10 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.

17.2 The C++ Units Library

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.

17.2.1 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>
#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::meters_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 an 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::meters_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::meters_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 its underlying value, use the `value()` method. This serves as an escape hatch from the units type system, which should be used only when necessary.

```
units::meter_t distance = 6.5_m;
double distanceMeters = distance.value();
```

17.2.2 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.

17.3 The Java Units Library

> **Note**
>
> New for 2025: The units library has been refactored to have unit-specific measurement classes instead of a single generic `Measure` class. The new measurement classes have clearer names (`Distance` instead of `Measure<Distance>`, or `LinearAcceleration` instead of `Measure<Velocity<Velocity<Distance>>>`), and implement math operations to return the most specific result types possible instead of a wildcard `Measure<?>`.

The units library is a tool that helps programmers avoid mistakes related to units of measurement. It does this by keeping track of the units of measurement, and by ensuring that all operations are performed with the correct units. This can help to prevent errors that can lead to incorrect results, such as adding a distance in inches to a distance in meters.

An added benefit is readability and maintainability, which also reduces bugs. By making the units of measurement explicit in your code, it becomes easier to read and understand what your code is doing. This can also help to make your code more maintainable, as it is easier to identify and fix errors related to units of measurement.

The units library has a number of features:

- A set of predefined units, such as meters, degrees, and seconds.
- The ability to convert between different units.
- Support for performing arithmetic and comparisons on quantities with units.
- Support for displaying quantities with units in a human-readable format.

17.3.1 Terminology

Dimension
: Dimensions represent the nature of a physical quantity, such as length, time, or mass. They are independent of any specific unit system. For example, the dimension of meters is length, regardless of whether the length is expressed in meters, millimeters, or inches.

Unit
: Units are specific realizations of dimensions. They are the way of expressing physical quantities. Each dimension has a base unit, such as the meter for length, the second for time, the kilogram for mass. Derived units are formed by combining base units, such as meters per second for velocity.

Measure
: Measures are the specific magnitude of physical quantities, expressed in a particular unit. For example, 5 meters is a measure of distance.

These concepts are used within the Units Library. For example, the **measure** *10 seconds* has a magnitude of 10, the **dimension** is time, and the **unit** is seconds.

17.3.2 Using the Units Library

The Java units library is available in the `edu.wpi.first.units` package. The most relevant classes are:

- The various classes for predefined dimensions, such as DistanceUnit and TimeUnit
- Units, which contains a set of predefined units. Take a look a the Units javadoc to browse the available units and their types.
- Measure, which is used to tag a value with a unit, and the dimension-specific implementations like Distance and Time

Note

It is recommended to static import `edu.wpi.first.units.Units.*` to get full access to all the predefined units.

Creating Measures

Every dimension has a measurement class with the corresponding name - for example, a `Distance` measures distance, `Time` measures time, and `LinearVelocity` measures linear velocity. To instantiate one of these measurements, call the `Unit.of` method on the appropriate unit object. For example, to create a `Distance` object representing a distance of 6 inches, you would write:

```
Distance wheelDiameter = Inches.of(6);
```

Other measures can also be created using their `Unit.of` method:

```
Mass kArmMass = Kilograms.of(1.423);
Distance kArmLength = Inches.of(32.25);
Angle kMinArmAngle = Degrees.of(5);
Angle kArmMaxTravel = Rotations.of(0.45);
LinearVelocity kMaxSpeed = MetersPerSecond.of(2.5);
```

> **Warning**
>
> Composite units with `PerUnit` and `MultUnit` have special requirements, and the `of` method is not recommended to be used with them

Using Composite Unit Types

Due to requirements of inheritance in Java's type system, `PerUnit` and `MultUnit` cannot return a normal `Per` or `Mult` object from their `of` factory methods. Instead, they need to return a bounded wildcard `Measure<? extends PerUnit<...>>` or `Measure<? extends MultUnit<...>>` to allow subclasses like `LinearVelocity` to return a compatible type. New `ofNative` methods are provided to be able to work with known `Per` and `Mult` objects

```
// Using ofNative:
Per<VoltageUnit, DistanceUnit> kP = VoltsPerMeter.ofNative(1);
kP.in(VoltsPerMeter); // 1.0

Measure<VoltageUnit> output = kP.timesDivisor(Meters.of(1));
output.in(Volts); // 1.0

// Without ofNative
Measure<? extends PerUnit<VoltageUnit, DistanceUnit>> kP = VoltsPerMeter.of(1);
kP.in(VoltsPerMeter); // Compilation error!

Measure<?> output = kP.times(Meters.of(1)); // The compiler can't know what unit this␣
↪is
output.in(Volts); // Compilation error!
```

Performing Calculations

The `Measure` class also supports arithmetic operations, such as addition, subtraction, multiplication, and division. These are done by calling methods on the objects. These operations always ensure that the units are compatible before performing the calculation, and they return a new `Measure` object. For example, you can add two `Distance` objects together, even if they have different units:

```
Distance distance1 = Inches.of(10);
Distance distance2 = Meters.of(0.254);
Distance totalDistance = distance1.plus(distance2);
```

In this code, the units library will automatically convert the measures to the same unit before adding the two distances. The resulting `totalDistance` object will be a new `Distance` object that has a value of 0.508 meters, or 20 inches.

Note

Mathematical operations are type safe. It is impossible to add a distance to a time, or subtract an angle from a voltage. However, multiplication and division operations make a best-effort attempt to return results in the most appropriate unit type; dividing a distance by time results in a `LinearVelocity` measurement, and multiplying it by time returns a `Distance`.

This example combines the wheel diameter and gear ratio to calculate the distance per rotation of the wheel:

```
Distance wheelDiameter = Inches.of(3);
double gearRatio = 10.48;
Distance distancePerRotation = wheelDiameter.times(Math.PI).divide(gearRatio);
```

Warning

By default, arithmetic operations create new `Measure` instances for their results. See *Java Garbage Collection* for discussion on creating a large number of short-lived objects. See also, the *Mutability and Object Creation* section below for a possible workaround.

Converting Units

Unit conversions can be done by calling `Measure.in(Unit)`. The Java type system will prevent units from being converted between incompatible types, such as distances to angles. The returned values will be bare `double` values without unit information - it is up to you, the programmer, to interpret them correctly! It is strongly recommended to only use unit conversions when interacting with APIs that do not support the units library.

```
LinearVelocity kMaxVelocity = FeetPerSecond.of(12.5);
LinearAcceleration kMaxAcceleration = FeetPerSecond.per(Second).of(22.9);
kMaxVelocity.in(MetersPerSecond); // => OK! Returns 3.81
kMaxVelocity.in(RadiansPerSecond); // => Compile error! LinearVelocity cannot be␣
↪converted to AngularVelocity
```

(continues on next page)

(continued from previous page)

```
// The WPILib math libraries use SI metric units, so we have to convert to meters:
TrapezoidProfile.Constraints kDriveConstraints = new TrapezoidProfile.Constraints(
  maxVelocity.in(MetersPerSecond),
  maxAcceleration.in(MetersPerSecondPerSecond)
);
```

Usage Example

Pulling all of the concepts together, we can create an example that calculates the end effector position of an arm mechanism:

```
Distance armLength = Feet.of(3).plus(Inches.of(4.25));
Distance endEffectorX = armLength.times(Math.cos(getArmAngle().in(Radians)));
Distance endEffectorY = armLength.times(Math.sin(getArmAngle().in(Radians)));
```

Human-readable Formatting

The `Measure` class has methods that can be used to get a human-readable representation of the measure. This feature is useful to display a measure on a dashboard or in logs.

- `toString()` and `toShortString()` return a string representation of the measure in a shorthand form. The symbol of the backing unit is used, rather than the full name, and the magnitude is represented in scientific notation. For example, 1.234e+04 V/m
- `toLongString()` returns a string representation of the measure in a longhand form. The name of the backing unit is used, rather than its symbol, and the magnitude is represented in a full string, not scientific notation. For example, 1234 Volt per Meter

17.3.3 Mutability and Object Creation

To reduce the number of object instances you create, and reduce memory usage, a special `MutableMeasure` class is available, with unit-specific subtypes like `MutDistance` and `MutTime`. You may want to consider using mutable objects if you are using the units library repeatedly, such as in the robot's periodic loop. See *Java Garbage Collection* for more discussion on creating a large number of short-lived objects. Mutable measures can be created in a similar way to regular, immutable measures using the `Unit.mutable` method (instead of `Unit.of`).

`MutableMeasure` allows the internal state of the object to be updated, such as with the results of arithmetic operations, to avoid allocating new objects. Special care needs to be taken when mutating a measure because it will change the value every place that instance is referenced. If the object will be exposed as part of a public method, have that method return a regular `Measure` in its signature to prevent the caller from modifying your internal state.

Extra methods are available on `MutableMeasure` for updating the internal value. Note that these methods all begin with the `mut_` prefix - this is to make it obvious that these methods will be mutating the object and are potentially unsafe! For the full list of methods and API documentation, see the MutableMeasure API documentation

`mut_plus(double, Unit)`	Increments the internal value by an amount in another unit. The internal unit will stay the same
`mut_plus(Measure)`	Increments the internal value by another measurement. The internal unit will stay the same
`mut_minus(double, Unit)`	Decrements the internal value by an amount in another unit. The internal unit will stay the same
`mut_minus(Measure)`	Decrements the internal value by another measurement. The internal unit will stay the same
`mut_times(double)`	Multiplies the internal value by a scalar
`mut_divide(double)`	Divides the internal value by a scalar
`mut_replace(double Unit)`	Overrides the internal state and sets it to equal the given value and unit
`mut_replace(Measur`	Overrides the internal state to make it identical to the given measurement
`mut_setMagnitude(d`	Overrides the internal value, keeping the internal unit. Be careful when using this!

```
MutDistance measure = Feet.mutable(0);
measure.mut_plus(10, Inches);    // 0.8333 feet
measure.mut_plus(Inches.of(10)); // 1.6667 feet
measure.mut_minus(5, Inches);    // 1.25 feet
measure.mut_minus(Inches.of(5)); // 0.8333 feet
measure.mut_times(6);            // 0.8333 * 6 = 5 feet
measure.mut_divide(5);           // 5 / 5 = 1 foot
measure.mut_replace(6.2, Meters) // 6.2 meters - note the unit changed!
measure.mut_replace(Millimeters.of(14.2)) // 14.2mm - the unit changed again!
measure.mut_setMagnitude(72)     // 72mm
```

Revisiting the arm example from above, we can use `mut_replace` - and, optionally, `mut_times` - to calculate the end effector position

```
import edu.wpi.first.units.Measure;
import edu.wpi.first.units.MutableMeasure;
import static edu.wpi.first.units.Units.*;
public class Arm {
  // Note the two ephemeral object allocations for the Feet.of and Inches.of calls.
  // Because this is a constant value computed just once, they will easily be garbage␣
↪collected without
  // any problems with memory use or loop timing jitter.
  private static final Distance kArmLength = Feet.of(3).plus(Inches.of(4.25));

  // Angle and X/Y locations will likely be called in the main robot loop, let's␣
↪store them in a MutableMeasure
  // to avoid allocating lots of short-lived objects
  private final MutAngle m_angle = Degrees.mutable(0);
  private final MutDistance m_endEffectorX = Feet.mutable(0);
  private final MutDistance m_endEffectorY = Feet.mutable(0);
  private final Encoder m_encoder = new Encoder(...);

  public Distance getEndEffectorX() {
    return m_endEffectorX.mut_replace(
      Math.cos(getAngle().in(Radians)) * kArmLength.in(Feet), // the new magnitude to␣
↪store
      Feet // the units of the new magnitude
```

(continues on next page)

(continued from previous page)

```
    );
  }

  public Distance getEndEffectorY() {
    // An alternative approach so we don't have to unpack and repack the units
    m_endEffectorY.mut_replace(kArmLength);
    m_endEffectorY.mut_times(Math.sin(getAngle().in(Radians)));
    return m_endEffectorY;
  }

  public Angle getAngle() {
    double rawAngle = m_encoder.getPosition();
    m_angle.mut_replace(rawAngle, Degrees); // NOTE: the encoder must be configured␣
↪with distancePerPulse in terms of degrees!
    return m_angle;
  }
}
```

Warning

`MutableMeasure` objects can - by definition - change their values at any time! It is unsafe to keep a stateful reference to them - prefer to extract a value using the `Measure.in` method, or create a copy with `Measure.copy` that can be safely stored. For the same reason, library authors must also be careful about methods accepting `Measure`.

Can you spot the bug in this code?

```
private Distance m_lastDistance;
public Distance calculateDelta(Distance currentDistance) {
  if (m_lastDistance == null) {
    m_lastDistance = currentDistance;
    return currentDistance;
  } else {
    Distance delta = currentDistance.minus(m_lastDistance);
    m_lastDistance = currentDistance;
    return delta;
  }
}
```

If we run the `calculateDelta` method a few times, we can see a pattern:

```
MutDistance distance = Inches.mutable(0);
distance.mut_plus(10, Inches);
calculateDelta(distance); // expect 10 inches and get 10 - good!
distance.mut_plus(2, Inches);
calculateDelta(distance); // expect 2 inches, but get 0 instead!
distance.mut_plus(8, Inches);
calculateDelta(distance); // expect 8 inches, but get 0 instead!
```

This is because the `m_lastDistance` field is a reference to the *same* `MutDistance` object as the input! Effectively, the delta is calculated as (currentDistance - currentDistance) on every call after the first, which naturally always returns zero. One solution would be to track `m_lastDistance` as a *copy* of the input measure to take a snapshot; however, this approach does incur one extra object allocation for the copy. If you need to be careful about object

allocations, `m_lastDistance` could also be stored as a `MutDistance`.

Immutable Copies

```
private Distance m_lastDistance;
public Distance calculateDelta(Distance currentDistance) {
  if (m_lastDistance == null) {
    m_lastDistance = currentDistance.copy();
    return currentDistance;
  } else {
    var delta = currentDistance.minus(m_lastDistance);
    m_lastDistance = currentDistance.copy();
    return delta;
  }
}
```

Zero-allocation Mutables

```
private final MutDistance m_lastDistance = Meters.mutable(0);
private final MutDistance m_delta = Meters.mutable(0);
public Distance calculateDelta(Distance currentDistance) {
  // m_delta = currentDistance - m_lastDistance
  m_delta.mut_replace(currentDistance);
  m_delta.mut_minus(m_lastDistance);
  m_lastDistance.mut_replace(currentDistance);
  return m_delta;
}
```

17.3.4 Defining New Units

There are four ways to define a new unit that isn't already present in the library:

- Using the `Unit.per` or `Unit.mult` methods to create a composite of two other units;
- Using the `Milli`, `Micro`, and `Kilo` helper methods;
- Using the `derive` method and customizing how the new unit relates to the base unit; and
- Subclassing `Unit` to define a new dimension.

New units can be defined as combinations of existing units using the `Unit.mult` and `Unit.per` methods.

```
PerUnit<VoltageUnit, DistanceUnit> VoltsPerInch = Volts.per(Inch);
VelocityUnit<MassUnit> KgPerSecond = Kilograms.per(Second); // Could also be declared␣
↪as PerUnit<MassUnit, TimeUnit>
DistanceUnit FootMinutesPerSecond = FeetPerSecond.mult(Minutes);
```

Using `mult` and `per` will store the resulting unit. Every call will return the same object to avoid unnecessary allocations and garbage collector pressure.

```
@Override
public void robotPeriodic() {
  // Feet.per(Millisecond) creates a new unit on the first loop,
  // which will be reused on every successive loop
  SmartDashboard.putNumber("Speed", m_drivebase.getSpeed().in(Feet.per(Millisecond)));
}
```

Note

Calling `Unit.per(Time)` will return a `Velocity` unit, which is different from and incompatible with a `Per` unit!

New dimensions can also be created by subclassing `Unit` and implementing the two constructors. Dimension-specific measurement types are recommended, but take considerable effort to implement all the unit-specific math operations.

```
public class ElectricChargeUnit extends Unit {
  public ElectricCharge(double baseUnitEquivalent, String name, String symbol) {
    super(ElectricCharge.class, baseUnitEquivalent, name, symbol);
  }
  // required for derivation with Milli, Kilo, etc.
  public ElectricCharge(UnaryFunction toBaseConverter, UnaryFunction␣
↪fromBaseConverter, String name, String symbol) {
     super(ElectricCharge.class, toBaseConverter, fromBaseConverter, name, symbol);
  }

  @Override
  public ElectricChargeUnit getBaseUnit() {
    // The base method must be overridden in order to return the correct type
    return (ElectricChargeUnit) super.getBaseUnit();
  }

  @Override
  public Measure<ElectricChargeUnit> of(double magnitude) {
    return ImmutableMeasure.ofRelativeUnits(magnitude, this);
  }

  @Override
  public Measure<ElectricChargeUnit> ofBaseUnits(double baseUnitMagnitude) {
    return ImmutableMeasure.ofBaseUnits(baseUnitMagnitude, this);
  }

  @Override
  public Measure<ElectricChargeUnit> zero() {
    return (Measure<ElectricChargeUnit>) super.zero();
  }

  @Override
  public Measure<ElectricChargeUnit> one() {
    return (Measure<ElectricChargeUnit>) super.one();
  }

  @Override
  public MutableMeasure<ElectricChargeUnit> mutable(double magnitude) {
    return new GenericMutableMeasureImpl(magnitude, toBaseUnits(magnitude), this);
```

(continues on next page)

(continued from previous page)

```
  }

  @Override
  public VelocityUnit<ElectricChargeUnit> per(TimeUnit period) {
    // Note: technically, this would return a CurrentUnit, since electric charge per␣
↪time is current (measured in Amperes)
    return VelocityUnit.combine(this, period);
  }

  public double convertFrom(double magnitude, ElectricChargeUnit otherUnit) {
    return fromBaseUnits(otherUnit.toBaseUnits(magnitude));
  }
}

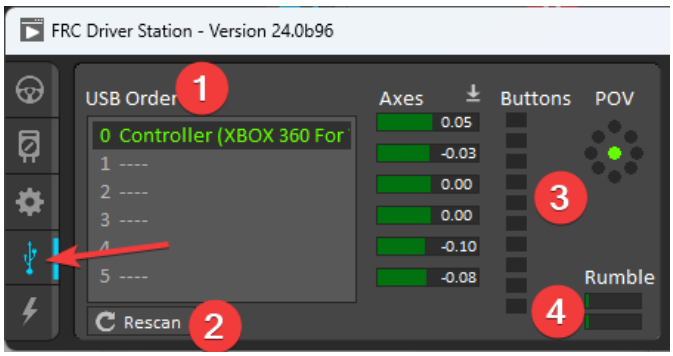
public static final ElectricCharge Coulomb = new ElectricCharge(1, "Coulomb", "C");
public static final ElectricCharge ElectronCharge = new ElectricCharge(1.60217646e-19,
↪ "Electron Charge", "e");
public static final ElectricCharge AmpHour = new ElectricCharge(3600, "Amp Hour", "Ah
↪");
public static final ElectricCharge MilliampHour = Milli(AmpHour);
```

17.4 Joysticks

A joystick can be used with the Driver Station program to control the robot. Almost any "controller" that can be recognized by Windows can be used as a joystick. Joysticks are accessed using the `GenericHID` class. This class has three relevant subclasses for preconfigured joysticks. You may also implement your own for other controllers by extending `GenericHID`. The first is `Joystick` which is useful for standard flight joysticks. The second is `XboxController` which works for the Xbox 360, Xbox One, or Logitech F310 (in XInput mode). Finally, the `PS4Controller` class is ideal for using that controller. Each axis of the controller ranges from -1 to 1.

The command based way to use the these classes is detailed in the section: *Binding Commands to Triggers*.

17.4.1 Driver Station Joysticks



The *USB Devices Tab* of the Driver Station is used to setup and configure the joystick for use with the robot. Pressing a button on a joystick will cause its entry in the table to light up green.

Selecting the joystick will show the values of axes, buttons, and the POV that can be used to determine the mapping between physical joystick features and axis or button numbers.



The USB Devices Tab also assigns a joystick index to each joystick. To reorder the joysticks simply click and drag. The Driver Station software will try to preserve the ordering of devices between runs. It is a good idea to note what order your devices should be in and check each time you start the Driver Station software that they are correct.

When the Driver Station is in disabled mode, it is routinely looking for status changes on the joystick devices. Unplugged devices are removed from the list and new devices are opened and added. When not connected to the FMS, unplugging a joystick will force the Driver Station into disabled mode. To start using the joystick again: plug the joystick in, check that it shows up in the right spot, then re-enable the robot. While the Driver Station is in enabled mode, it will not scan for new devices. This is a time consuming operation and timely update of signals from attached devices takes priority.

Note

For some joysticks the startup routine will read whatever position the joysticks are in as the center position, therefore, when the computer is turned on (or when the joystick is plugged in) the joysticks should be at their center position.

When the robot is connected to the Field Management System at competition, the Driver Station mode is dictated by the *FMS*. This means that you cannot disable your robot and the DS cannot disable itself in order to detect joystick changes. A manual complete refresh of the joysticks can be initiated by pressing the F1 key on the keyboard. Note that this will close and re-open all devices, so all devices should be in their center position as noted above.

17.4.2 `Joystick` Class



JAVA

```
Joystick exampleJoystick = new Joystick(0); // 0 is the USB Port to be used as␣
↪indicated on the Driver Station
```

C++

```
Joystick exampleJoystick{0}; // 0 is the USB Port to be used as indicated on the␣
↪Driver Station
```

PYTHON

```
exampleJoystick = wpilib.Joystick(0) # 0 is the USB Port to be used as indicated on␣
↪the Driver Station
```

The `Joystick` class is designed to make using a flight joystick to operate the robot significantly easier. Depending on the flight joystick, the user may need to set the specific X, Y, Z, and Throttle channels that your flight joystick uses. This class offers special methods for accessing the angle and magnitude of the flight joystick.

Important

Due to differences in coordinate systems, teams usually negate the values when reading joystick axes. See the *Joystick and controller coordinate system* section for more detail.

17.4.3 XboxController Class



JAVA

```
XboxController exampleXbox = new XboxController(0); // 0 is the USB Port to be used␣
↪as indicated on the Driver Station
```

C++

```
XboxController exampleXbox{0}; // 0 is the USB Port to be used as indicated on the␣
↪Driver Station
```

PYTHON

```
exampleXbox = wpilib.XboxController(0) # 0 is the USB Port to be used as indicated on␣
↪the Driver Station
```

The `XboxController` class provides named methods (e.g. `getXButton`, `getXButtonPressed`, `getXButtonReleased`) for each of the buttons, and the indices can be accessed with `XboxController.Button.kX.value`. The rumble feature of the controller can be controlled by using `XboxController.setRumble(GenericHID.RumbleType.kRightRumble, value)`. Many users do a split stick arcade drive that uses the left stick for just forwards / backwards and the right stick for left / right turning.

Important

Due to differences in coordinate systems, teams usually negate the values when reading joystick axes. See the *Joystick and controller coordinate system* section for more detail.

17.4.4 PS4Controller Class



JAVA

```
PS4Controller examplePS4 = new PS4Controller(0); // 0 is the USB Port to be used as␣
↪indicated on the Driver Station
```

C++

```
PS4Controller examplePS4{0}; // 0 is the USB Port to be used as indicated on the␣
↪Driver Station
```

PYTHON

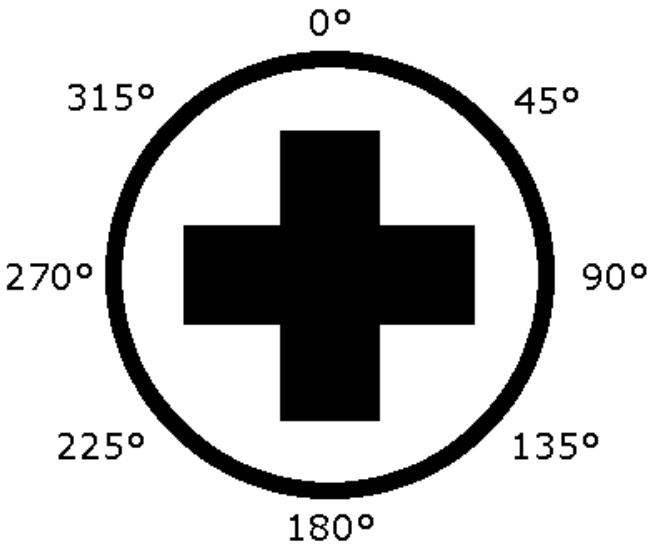
```
examplePS4 = wpilib.PS4Controller(0) # 0 is the USB Port to be used as indicated on␣
↪the Driver Station
```

The `PS4Controller` class provides named methods (e.g. `getSquareButton`, `getSquareButtonPressed`, `getSquareButtonReleased`) for each of the buttons, and the indices can be accessed with `PS4Controller.Button.kSquare.value`. The rumble feature of the controller can be controlled by using `PS4Controller.setRumble(GenericHID.RumbleType.kRightRumble, value)`.

Important

Due to differences in coordinate systems, teams usually negate the values when reading joystick axes. See the *Joystick and controller coordinate system* section for more detail.

17.4.5 POV



On joysticks, the POV is a directional hat that can select one of 8 different angles or read -1 for unpressed. The XboxController/PS4Controller D-pad works the same as a POV. Be careful when using a POV with exact angle requirements as it is hard for the user to ensure they select exactly the angle desired.

17.4.6 GenericHID Usage

An axis can be used with `.getRawAxis(int index)` (if not using any of the classes above) that returns the current value. Zero and one in this example are each the index of an axis as found in the Driver Station mentioned above.

JAVA

```
private final PWMSparkMax m_leftMotor = new PWMSparkMax(Constants.kLeftMotorPort);
private final PWMSparkMax m_rightMotor = new PWMSparkMax(Constants.kRightMotorPort);
private final DifferentialDrive m_robotDrive = new DifferentialDrive(m_leftMotor::set,
↪ m_rightMotor::set);
private final GenericHID m_stick = new GenericHID(Constants.kJoystickPort);
m_robotDrive.arcadeDrive(-m_stick.getRawAxis(0), m_stick.getRawAxis(1));
```

C++

```
frc::PWMVictorSPX m_leftMotor{Constants::kLeftMotorPort};
frc::PWMVictorSPX m_rightMotor{Constants::kRightMotorPort};
frc::DifferentialDrive m_robotDrive{[&](double output) { m_leftMotor.Set(output); },
                                    [&](double output) { m_rightMotor.Set(output); }};
frc::GenericHID m_stick{Constants::kJoystickPort};
m_robotDrive.ArcadeDrive(-m_stick.GetRawAxis(0), m_stick.GetRawAxis(1));
```

PYTHON

```
leftMotor = wpilib.PWMVictorSPX(LEFT_MOTOR_PORT)
rightMotor = wpilib.PWMVictorSPX(RIGHT_MOTOR_PORT)
self.robotDrive = wpilib.drive.DifferentialDrive(leftMotor, rightMotor)
self.stick = wpilib.GenericHID(JOYSTICK_PORT)
self.robotDrive.arcadeDrive(-self.stick.getRawAxis(0), self.stick.getRawAxis(1))
```

17.4.7 Button Usage

Note

Usage such as the following is for code not using the command-based framework. For button usage in the command-based framework, see *Binding Commands to Triggers*.

Unlike an axis, you will usually want to use the `pressed` and `released` methods to respond to button input. These will return true if the button has been activated since the last check. This is helpful for taking an action once when the event occurs but not having to continuously do it while the button is held down.

JAVA

```
if (joystick.getRawButtonPressed(0)) {
   turnIntakeOn(); // When pressed the intake turns on
}
if (joystick.getRawButtonReleased(0)) {
   turnIntakeOff(); // When released the intake turns off
}
// OR
if (joystick.getRawButton(0)) {
   turnIntakeOn();
} else {
   turnIntakeOff();
}
```

C++

```
if (joystick.GetRawButtonPressed(0)) {
   turnIntakeOn(); // When pressed the intake turns on
}
if (joystick.GetRawButtonReleased(0)) {
   turnIntakeOff(); // When released the intake turns off
}
// OR
if (joystick.GetRawButton(0)) {
   turnIntakeOn();
} else {
   turnIntakeOff();
}
```

PYTHON

```
if joystick.getRawButtonPressed(0):
   turnIntakeOn() # When pressed the intake turns on
if joystick.getRawButtonReleased(0):
   turnIntakeOff() # When released the intake turns off
# OR
if joystick.getRawButton(0):
   turnIntakeOn()
else:
   turnIntakeOff()
```

A common request is to toggle something on and off with the press of a button. Toggles should be used with caution, as they require the user to keep track of the robot state.

JAVA

```
boolean toggle = false;
if (joystick.getRawButtonPressed(0)) {
   if (toggle) {
      // Current state is true so turn off
      retractIntake();
      toggle = false;
   } else {
      // Current state is false so turn on
      deployIntake();
      toggle = true;
   }
}
```

C++

```
bool toggle{false};
if (joystick.GetRawButtonPressed(0)) {
   if (toggle) {
      // Current state is true so turn off
      retractIntake();
      toggle = false;
   } else {
      // Current state is false so turn on
      deployIntake();
      toggle = true;
   }
}
```

PYTHON

```
toggle = False
if joystick.getRawButtonPressed(0):
   if toggle:
      # current state is True so turn off
      retractIntake()
      toggle = False
   else:
      # Current state is False so turn on
      deployIntake()
      toggle = True
```

17.5 Coordinate System

Coordinate systems are used in FRC programming in several places. A few of the common places are: robot movement, joystick input, *pose* estimation, AprilTags, and path planning.

It is important to understand the basics of the coordinate system used throughout WPILib and other common tools for programming an FRC robot, such as PathPlanner. Many teams intuitively think of a coordinate system that is different from what is used in WPILib, and this leads to problems that need to be tracked down throughout the season. It is worthwhile to take a few minutes to understand the coordinate system, and come back here as a reference when programming. It's not very difficult to get robot movement with a joystick working without getting the coordinate system right, but it will be much more difficult to build on code using a different coordinate system to add *pose estimation* with *AprilTags* and path planning for autonomous.

17.5.1 WPILib coordinate system

In most cases, WPILib uses the NWU axes convention (North-West-Up as external reference in the world frame.) In the NWU axes convention, where the positive X axis points ahead, the positive Y axis points left, and the positive Z axis points up referenced from the floor. When viewed with each positive axis pointing toward you, counter-clockwise (CCW) is a positive value and clockwise (CW) is a negative value.

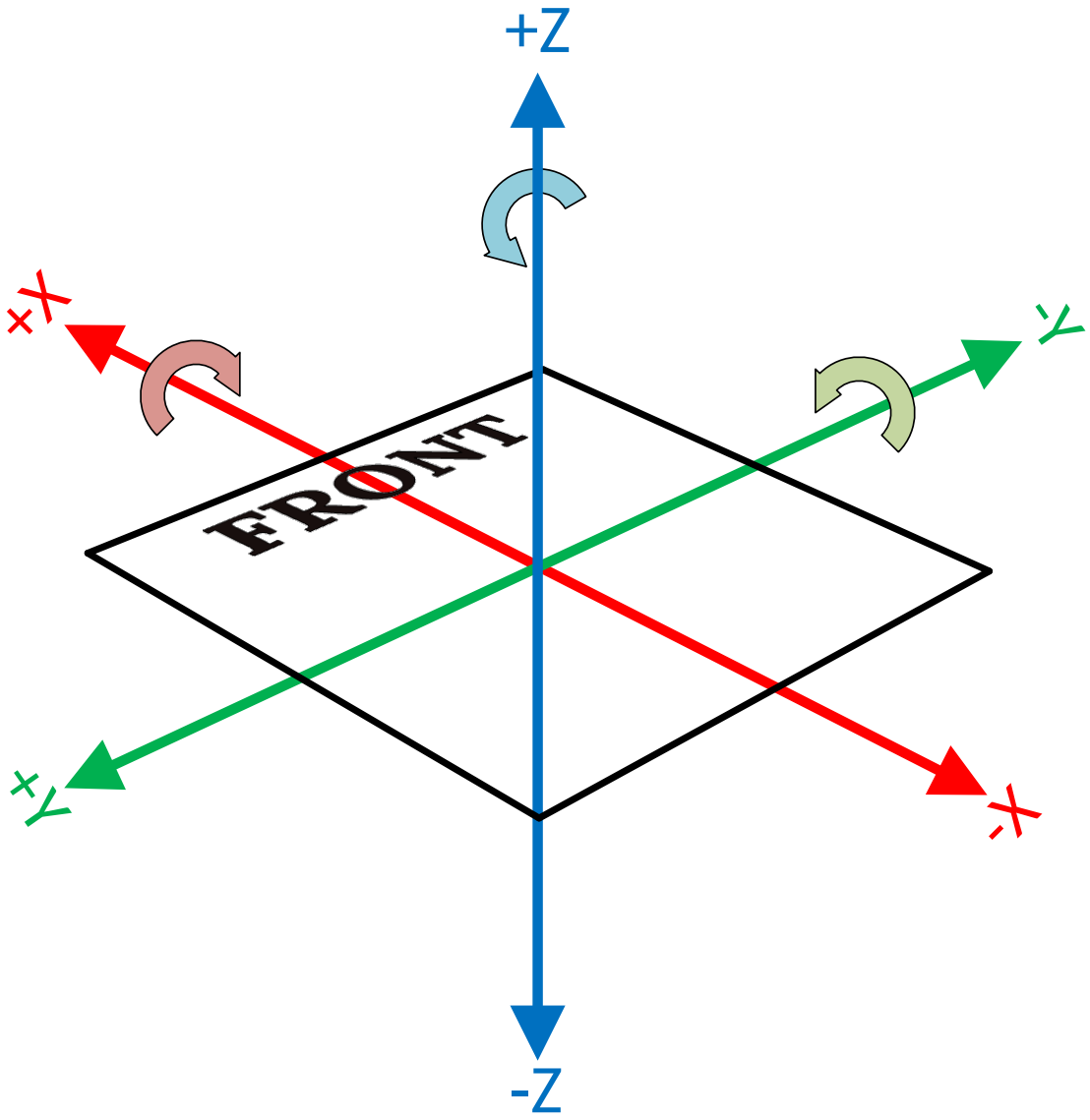


Fig. 1: Robot coordinate system in three dimensions

The figure above shows the coordinate system in relation to an FRC robot. The figure below shows this same coordinate system when viewed from the top (with the Z axis pointing toward you.) This is how you can think of the robot's coordinates in 2D.

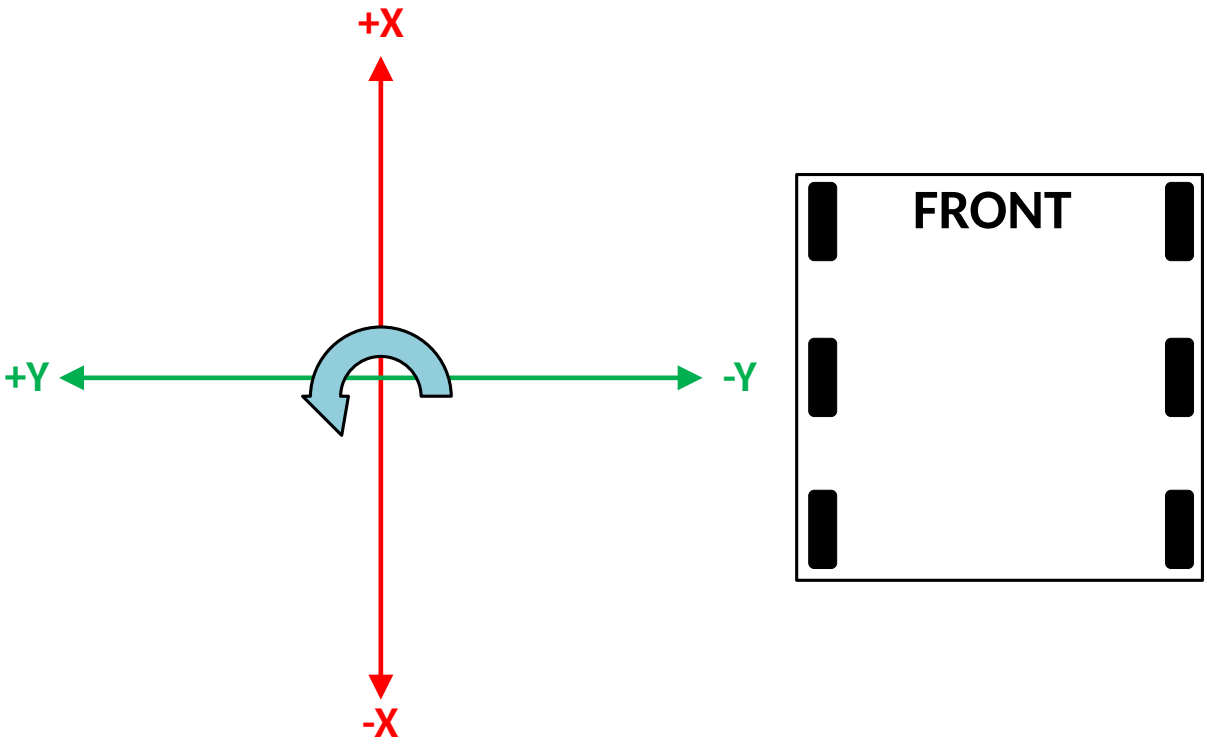


Fig. 2: Robot coordinate system in two dimensions

17.5.2 Rotation conventions

In most cases in WPILib programming, 0° is aligned with the positive X axis, and 180° is aligned with the negative X axis. CCW rotation is positive, so 90° is aligned with the positive Y axis, and -90° is aligned with the negative Y axis.

The figure above shows the unit circle with common angles labeled in degrees (°) and radians (rad). Notice that rotation to the right is negative, and the range for the whole unit circle is -180° to 180° (-Pi radians to Pi radians).

Note

The range is (-180, 180], meaning it is exclusive of -180° and inclusive of 180°.

There are some places you may choose to use a different range, such as 0° to 360° or 0 to 1 rotation, but be aware that many core WPILib classes and FRC tools are built with the unit circle above.

Warning

Some *gyroscope* and *IMU* models use CW positive rotation, such as the NavX IMU. Care must be taken to handle rotation properly, sensor values may need to be inverted. Read the documentation and verify that rotation is CCW positive.

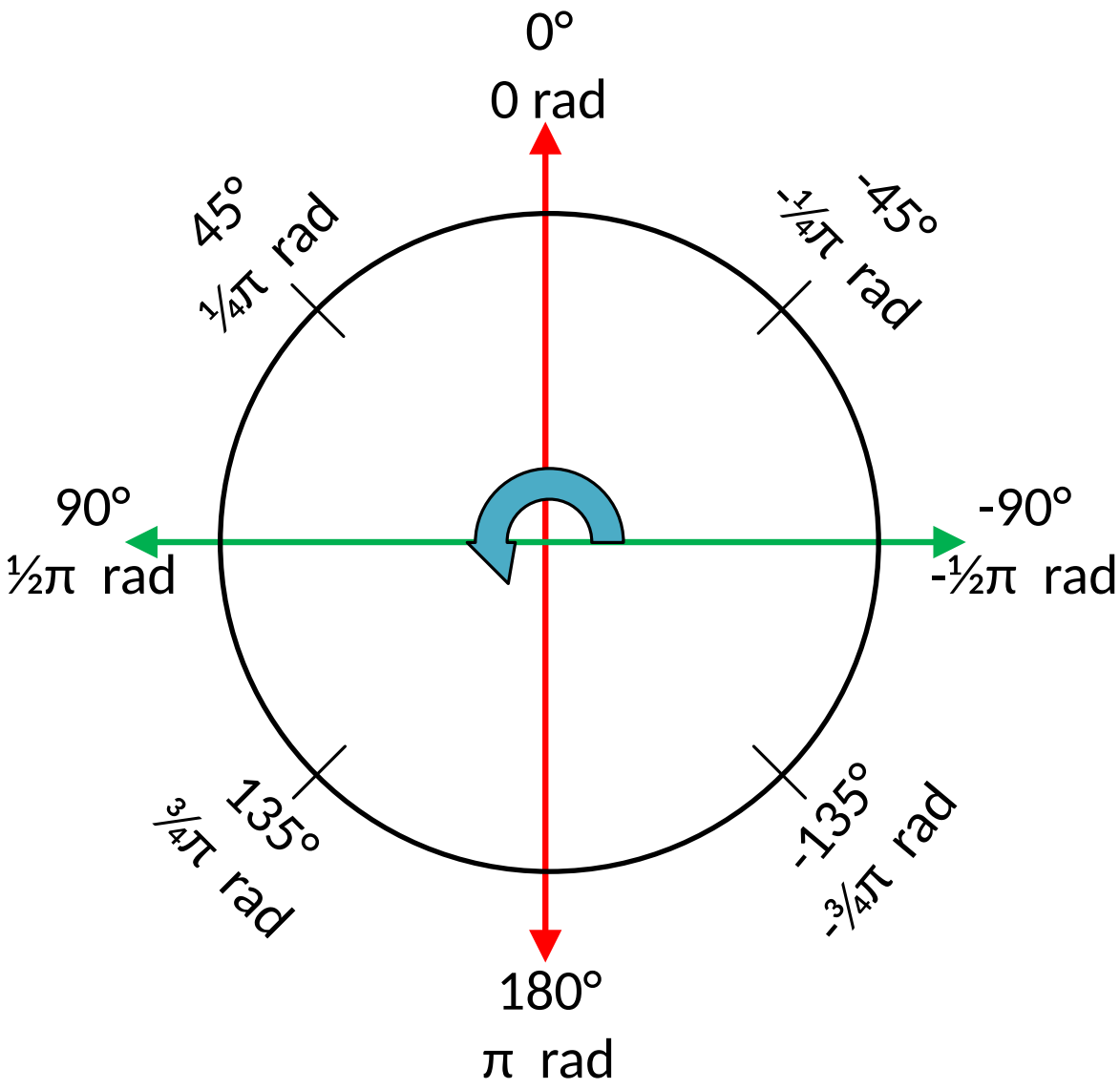


Fig. 3: Unit circle with common angles

> **Warning**
>
> Many sensors that read rotation around an axis, such as encoders and IMU's, read continuously. This means they read more than one rotation, so when rotating past 180° they read 181°, not -179°. Some sensors have configuration settings where you can choose their wrapping behavior and range, while others need to be handled in your code. Careful attention should be paid to make sure sensor readings are consistent and your control loop handles wrapping in the same way as your sensor.

17.5.3 Joystick and controller coordinate system

Joysticks, including the sticks on controllers, don't use the same NWU coordinate system. They use the NED (North-East-Down) convention, where the positive X axis points ahead, the positive Y axis points right, and the positive Z axis points down. When viewed with each positive axis pointing toward you, counter-clockwise (CCW) is a positive value and clockwise (CW) is a negative value.

It's important to note that joystick input values are rotations around an axis, not translations. In practical terms, this means:

- pushing forward on the joystick (toward the positive X axis) is a CW rotation around the Y axis, so you get a negative Y value.
- pushing to the right (toward the positive Y axis) is a CCW rotation around the X axis, so you get a positive X value.
- twisting the joystick CW (toward the positive Y axis) is a CCW rotation around the Z axis, so you get a positive Z value.

17.5.4 Using Joystick and controller input to drive a robot

You may have noticed, the coordinate system used by WPILib for the robot is not the same as the coordinate system used for joysticks and controllers. Care needs to be taken to understand the difference, and properly pass driver input to the drive subsystem.

Differential drivetrain example

Differential drivetrains are non-holonomic, which means the robot drivetrain cannot move side-to-side (strafe). This type of drivetrain can move forward and backward along the X axis, and rotate around the Z axis. Consider a common arcade drive scheme using a single joystick where the driver pushes the joystick forward/backward for forward/backward robot movement, and push the joystick left/right to rotate the robot left/right.

The code snippet below uses the `DifferentialDrive` and `Joystick` classes to drive the robot with the arcade scheme described above. `DifferentialDrive` uses the robot coordinate system defined above, and `Joystick` uses the joystick coordinate system.

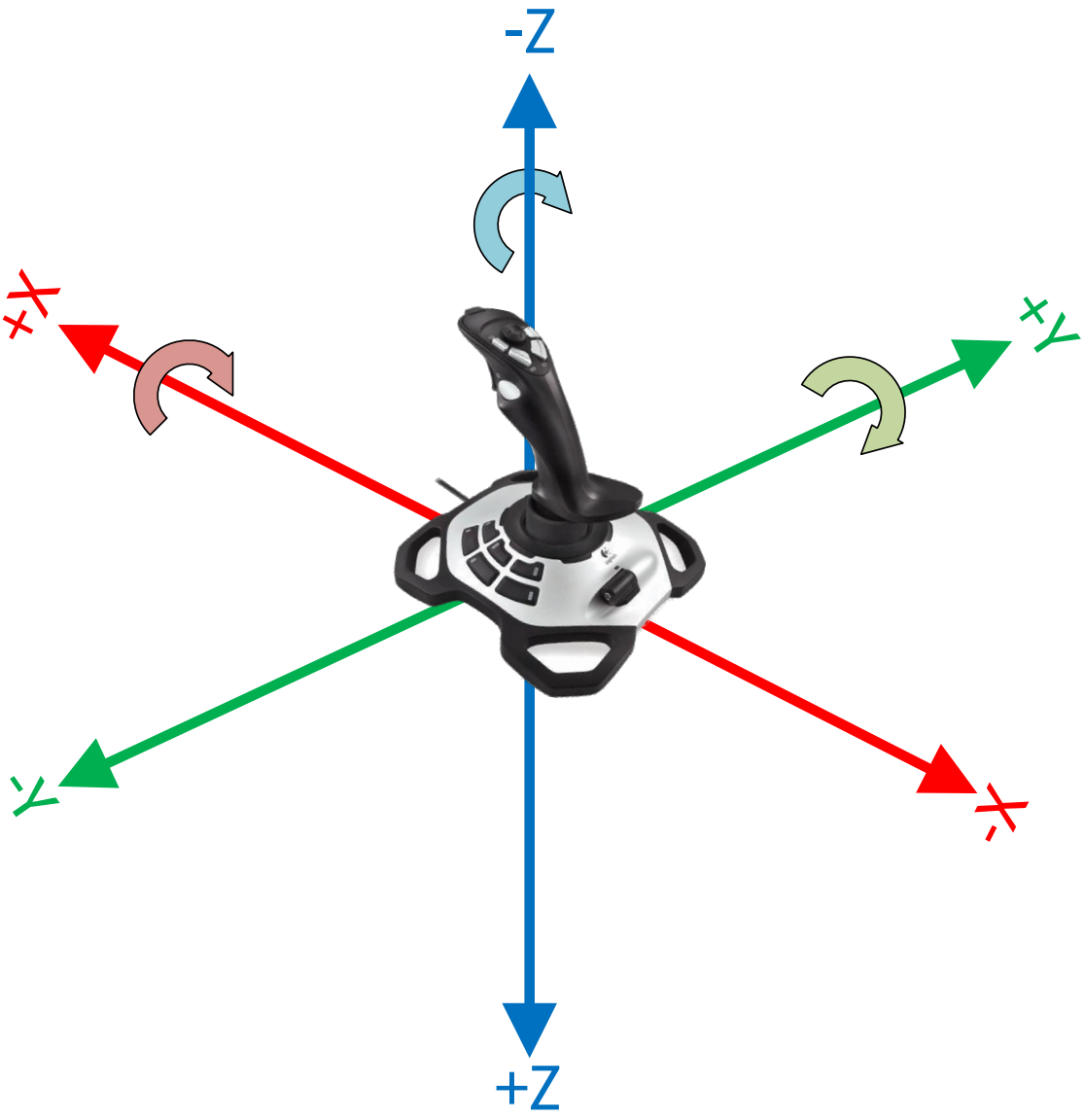


Fig. 4: Joystick coordinate system

JAVA

```java
public void teleopPeriodic() {
    // Arcade drive with a given forward and turn rate
    myDrive.arcadeDrive(-driveStick.getY(), -driveStick.getX());
}
```

C++

```cpp
void TeleopPeriodic() override {
    // Arcade drive with a given forward and turn rate
    myDrive.ArcadeDrive(-driveStick.GetY(), -driveStick.GetX());
}
```

PYTHON

```python
def teleopPeriodic(self):
    # Arcade drive with a given forward and turn rate
    self.myDrive.arcadeDrive(-self.driveStick.getY(), -self.driveStick.getX())
```

The code calls the `DifferentialDrive.arcadeDrive(xSpeed, zRotation)` method, with values it gets from the `Joystick` class:

- **The first argument is `xSpeed`**
  - Robot: `xSpeed` is the speed along the robot's X axis, which is forward/backward.
  - Joystick: The driver sets forward/backward speed by rotating the joystick along its Y axis, which is pushing the joystick forward/backward.
  - Code: Moving the joystick forward is negative Y rotation, whereas moving the robot forward is along the positive X axis. This means the joystick value needs to be inverted by placing a - (minus sign) in front of the value.
- **The second argument is `zRotation`**
  - Robot: `zRotation` is the speed of rotation along the robot's Z axis, which is rotating left/right.
  - Joystick: The driver sets rotation speed by rotating the joystick along its X axis, which is pushing the joystick left/right.
  - Code: Moving the joystick to the right is positive X rotation, whereas robot rotation is CCW positive. This means the joystick value needs to be inverted by placing a - (minus sign) in front of the value.

Mecanum drivetrain example

Mecanum drivetrains are holonomic, meaning they have the ability to move side-to-side. This type of drivetrain can move forward/backward and rotate around the Z axis like differential drivetrains, but it can also move side-to-side along the robot's Y axis. Consider a common arcade drive scheme using a single joystick where the driver pushes the joystick forward/backward for forward/backward robot movement, pushes the joystick left/right to move side-to-side, and twists the joystick to rotate the robot.

JAVA

```
public void teleopPeriodic() {
    // Drive using the X, Y, and Z axes of the joystick.
    m_robotDrive.driveCartesian(-m_stick.getY(), -m_stick.getX(), -m_stick.getZ());
}
```

C++

```
void TeleopPeriodic() override {
    // Drive using the X, Y, and Z axes of the joystick.
    m_robotDrive.driveCartesian(-m_stick.GetY(), -m_stick.GetX(), -m_stick.GetZ());
}
```

PYTHON

```
def teleopPeriodic(self):
    // Drive using the X, Y, and Z axes of the joystick.
    self.robotDrive.driveCartesian(-self.stick.getY(), -self.stick.getX(), -self.
↪stick.getZ())
```

The code calls the `MecanumDrive.driveCartesian(xSpeed, ySpeed, zRotation)` method, with values it gets from the `Joystick` class:

- **The first argument is xSpeed**
  - Robot: `xSpeed` is the speed along the robot's X axis, which is forward/backward.
  - Joystick: The driver sets forward/backward speed by rotating the joystick along its Y axis, which is pushing the joystick forward/backward.
  - Code: Moving the joystick forward is negative Y rotation, whereas robot forward is along the positive X axis. This means the joystick value needs to be inverted by placing a - (minus sign) in front of the value.
- **The second argument is ySpeed**
  - Robot: `ySpeed` is the speed along the robot's Y axis, which is left/right.
  - Joystick: The driver sets left/right speed by rotating the joystick along its X axis, which is pushing the joystick left/right.
  - Code: Moving the joystick to the right is positive X rotation, whereas robot right is along the negative Y axis. This means the joystick value needs to be inverted by placing a - (minus sign) in front of the value.

- **The third argument is `zRotation`**
  - Robot: `zRotation` is the speed of rotation along the robot's Z axis, which is rotating left/right.
  - Joystick: The driver sets rotation speed by twisting the joystick along its Z axis, which is twisting the joystick left/right.
  - Code: Twisting the joystick to the right is positive Z rotation, whereas robot rotation is CCW positive. This means the joystick value needs to be inverted by placing a - (minus sign) in front of the value.

Swerve drivetrain example

Like mecanum drivetrains, swerve drivetrains are holonomic and have the ability to move side-to-side. Joystick control can be handled the same way for all holonomic drivetrains, but WPILib doesn't have a built-in robot drive class for swerve. Swerve coding is described in other sections of this documentation, but an example of using joystick input to set `ChassisSpeeds` values is included below. Consider the same common arcade drive scheme described in the mecanum section above. The scheme uses a single joystick where the driver pushes the joystick forward/backward for forward/backward robot movement, pushes the joystick left/right to move side-to-side, and twists the joystick to rotate the robot.

JAVA

```
// Drive using the X, Y, and Z axes of the joystick.
var speeds = new ChassisSpeeds(-m_stick.getY(), -m_stick.getX(), -m_stick.getZ());
```

C++

```
// Drive using the X, Y, and Z axes of the joystick.
frc::ChassisSpeeds speeds{-m_stick.GetY(), -m_stick.GetX(), -m_stick.GetZ()};
```

PYTHON

```
# Drive using the X, Y, and Z axes of the joystick.
speeds = ChassisSpeeds(-self.stick.getY(), -self.stick.getX(), -self.stick.getZ())
```

The three arguments to the `ChassisSpeeds` constructor are the same as `driveCartesian` in the mecanum section above; `xSpeed`, `ySpeed`, and `zRotation`. See the description of the arguments, and their joystick input in the section above.

17.5.5 Robot drive kinematics

Kinematics is a topic that is covered in a different section, but it's worth discussing here in relation to the coordinate system. It is critically important that kinematics is configured using the coordinate system described above. Kinematics is a common starting point for coordinate system errors that then cascade to basic drivetrain control, field oriented driving, pose estimation, and path planning.

When you construct a `SwerveDriveKinematics` or `MecanumDriveKinematics` object, you specify a translation from the center of your robot to each wheel. These translations use the coordinate system above, with the origin in the center of your robot.

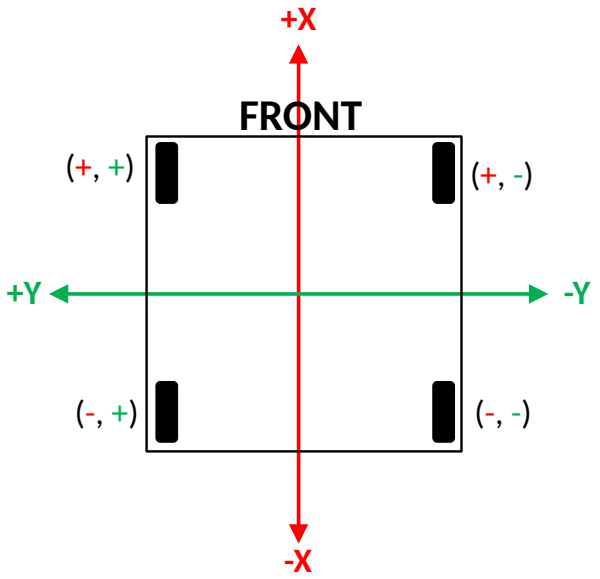


Fig. 5: Kinematics with translation signs

For the robot in the diagram above, let's assume the distance between the front and rear wheels (wheelbase) is 2'. Let's also assume the distance between the left and right wheels (trackwidth) is also 2'. Our translations (x, y) would be like this:

- Front left: (1', 1')
- Front right: (1', -1')
- Rear left: (-1', 1')
- Rear right: (-1', -1')

Warning

A common error is to use an incorrect coordinate system where the positive Y axis points forward on the robot. The correct coordinate system has the positive X axis pointing forward.

17.5.6 Field coordinate systems

The field coordinate system (or global coordinate system) is an absolute coordinate system where a point on the field is designated as the origin. Two common uses of the field coordinate system will be explored in this document:

- Field oriented driving is a drive scheme for holonomic drivetrains, where the driver moves the controls relative to their perspective of the field, and the robot moves in that direction regardless of where the front of the robot is facing. For example, a driver on the red alliance pushes the joystick forward, the robot will move downfield toward the blue alliance wall, even if the robot's front is facing the driver.
- Pose estimation with odometry and/or AprilTags are used to estimate the robot's pose on the field.

Mirrored field vs. rotated field

Historically, FRC has used two types of field layouts in relation to the red and blue alliance.

Games such as Rapid React in 2022 used a rotated layout. A rotated layout means that, from your perspective from behind your alliance wall, your field elements and your opponent's elements are in the same location. Notice in the Rapid React field layout diagram below, whether you are on the red or blue alliance, your human player station is on your right and your hanger is on your left.

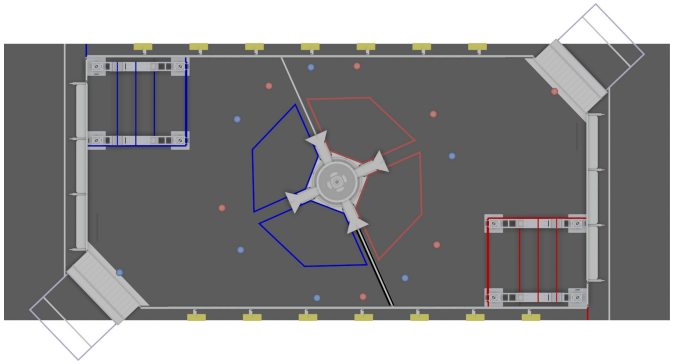


Fig. 6: Rotated field from RAPID REACT in 2022[1]

Games such as CHARGED UP in 2023 and CRESCENDO in 2024 used a mirrored layout. A mirrored layout means that the red and blue alliance layout are mirrored across the center-point of the field. Refer to the CHARGED UP field diagram below. When you are standing behind the blue alliance wall, the charge station is on the right side of the field from your perspective. However, standing behind the red alliance wall, the charge station is on the left side of the field from your perspective.

[1] Rapid React field image from MikLast on Chiefdelphi https://www.chiefdelphi.com/t/2022-top-down-field-renders/399031

[2] CHARGED UP field image from MikLast on Chiefdelphi https://www.chiefdelphi.com/t/2023-top-down-field-renders/421365

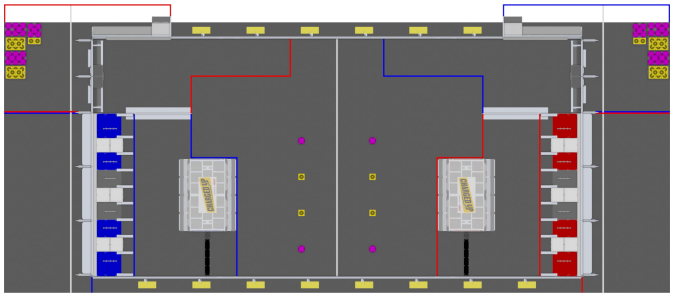


Fig. 7: Mirrored field from CHARGED UP in 2023[Page 569, 2]

Dealing with red or blue alliance

There are two primary ways many teams choose to define the field coordinate system. In both methods, positive rotation (theta) is in the counter-clockwise (CCW) direction.

Warning

There are cases where your alliance may change (or appear to change) after the code is initialized. When you are not connected to the *FMS* at a competition, you can change your alliance station in the Driver Station application at any time. Even when you are at a competition, your robot will usually initialize before connecting to the FMS so you will not have alliance information.

Note

At competition events, the FMS will automatically report your Team Station and alliance color. When you are not connected to an FMS, you can choose your Team Station and alliance color on the Driver Station *Operation Tab*.

Always blue origin

You may choose to define the origin of the field on the blue side, and keep it there regardless of your alliance color. With this solution, positive x-axis points away from the blue alliance wall.

Some advantages to this approach are:

- Pose estimation with AprilTags is simplified. AprilTags throughout the field are unique. If you keep the coordinate system the same regardless of alliance, there is no need for special logic to deal with the location of AprilTags on the field relative to your alliance.
- Many of the tools and libraries used in FRC follow this convention. Some of the tools include: PathPlanner, Choreo, and the ShuffleBoard and Glass Field2d widget.

In order to use this approach for field oriented driving, driver input needs to consider the alliance color. When your alliance is red and the driver is standing behind the red alliance wall, they will want the robot to move downfield toward the blue alliance wall. However, when your alliance is blue, the driver will want the robot to go downfield toward the red alliance wall.

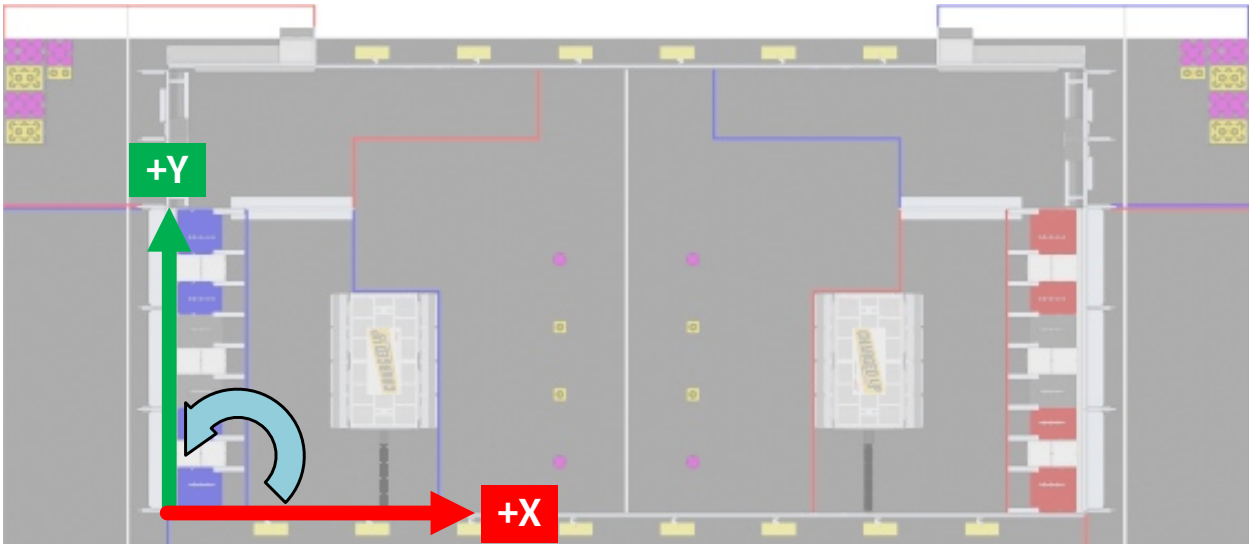


Fig. 8: CHARGED UP with blue origin

A simple way to deal with field oriented driving is to check the alliance color reported by the *DriverStation* class, and invert the driver's controls based on the alliance. As noted above, your alliance color can change so it needs to be checked on every robot iteration.

JAVA

```
// The origin is always blue. When our alliance is red, X and Y need to be inverted
var alliance = DriverStation.getAlliance();
var invert = 1;
if (alliance.isPresent() && alliance.get() == Alliance.Red) {
    invert = -1;
}
// Create field relative ChassisSpeeds for controlling Swerve
var chassisSpeeds = ChassisSpeeds
        .fromFieldRelativeSpeeds(xSpeed * invert, ySpeed * invert, zRotation, imu.
↪getRotation2d());
// Control a mecanum drivetrain
m_robotDrive.driveCartesian(xSpeed * invert, ySpeed * invert, zRotation, imu.
↪getRotation2d());
```

C++

```
// The origin is always blue. When our alliance is red, X and Y need to be inverted
int invert = 1;
if (frc::DriverStation::GetAlliance() == frc::DriverStation::Alliance::kRed) {
    invert = -1;
}
// Create field relative ChassisSpeeds for controlling Swerve
frc::ChassisSpeeds chassisSpeeds =
        frc::ChassisSpeeds::FromFieldRelativeSpeeds(xSpeed * invert, ySpeed * invert,␣
↪zRotation, imu.GetRotation2d());
```

(continues on next page)

(continued from previous page)

```
// Control a mecanum drivetrain
m_robotDrive.driveCartesian(xSpeed * invert, ySpeed * invert, zRotation, imu.
↪GetRotation2d());
```

PYTHON

```
# The origin is always blue. When our alliance is red, X and Y need to be inverted
invert = 1
if wpilib.DriverStation.getAlliance() == wpilib.DriverStation.Alliance.kRed:
    invert = -1
# Create field relative ChassisSpeeds for controlling Swerve
chassis_speeds = wpilib.ChassisSpeeds.FromFieldRelativeSpeeds(
    xSpeed * invert, ySpeed * invert, zRotation, self.imu.GetAngle()
)
# Control a mecanum drivetrain
self.robotDrive.driveCartesian(xSpeed * invert, ySpeed * invert, zRotation, self.imu.
↪GetAngle())
```

Origin follows your alliance

You may choose to define the origin of the field based on the alliance you are one. With this approach, the positive x-axis always points away from your alliance wall.

When you are on the blue alliance, your origin looks like this:

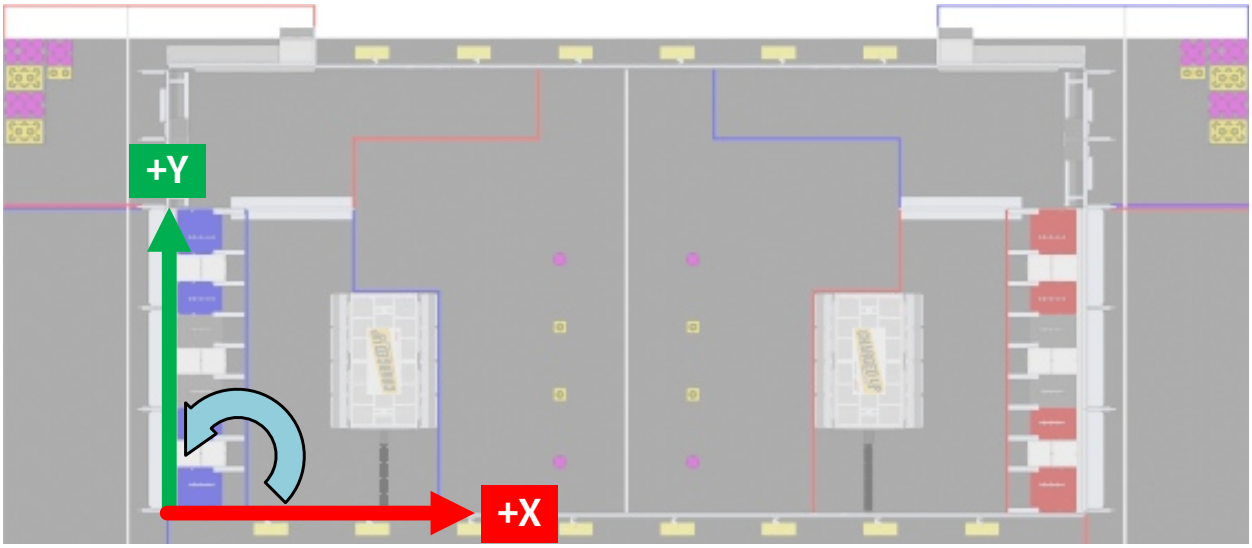


Fig. 9: CHARGED UP field with blue alliance as origin

When you are on the red alliance, your origin looks like this:

This approach has a few more complications than the previous approach, especially in years when the field layout is mirrored between alliances.

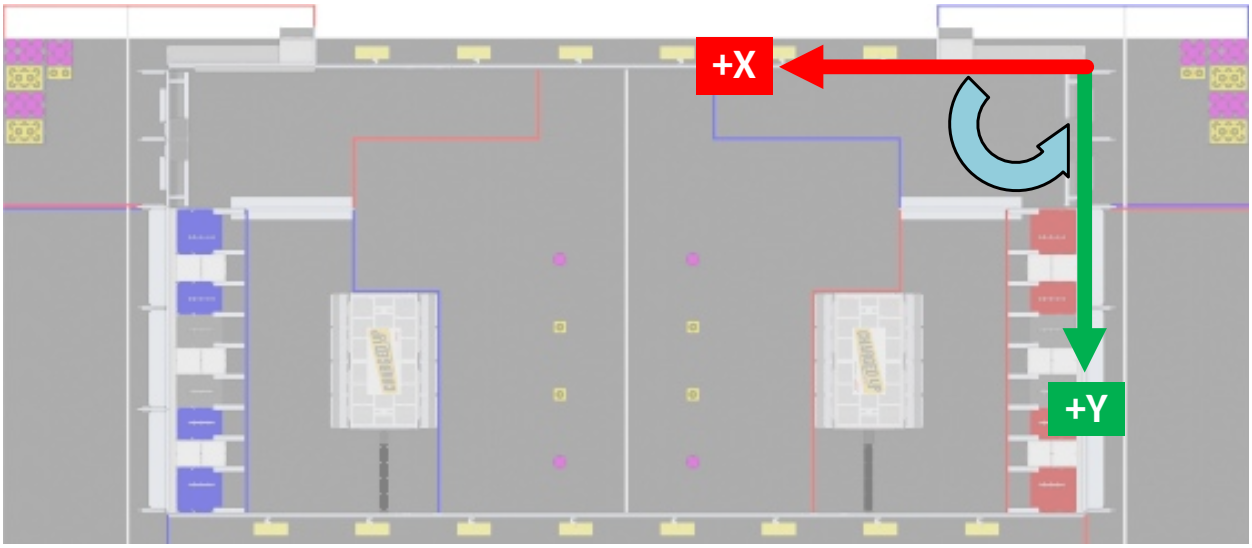


Fig. 10: CHARGED UP field with red alliance as origin

In years when the field layout is rotated, this is a simple approach if you are not using AprilTags for pose estimation or doing other advanced techniques. When the field layout is rotated, the field elements appear at the same coordinates regardless of your alliance.

Some things you need to consider when using this approach are:

- As warned above, your alliance color can change after initialization. If you are not using AprilTags, you may not have anything to adjust when the alliance changes. However, if you are using AprilTags and your robot has seen a tag and used it for pose estimation, you will need to adjust your origin and reset your estimated pose.
- The field image in the ShuffleBoard and Glass Field2d widget follows the *Always blue origin* approach. Special handling is needed to display your robot pose correctly when your alliance is red. You will need to change the origin for your estimated pose to the blue alliance coordinate system before sending it to the dashboard.

17.6 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, setpoints, etc. that you would like to change without having to rebuild the program. The values can be viewed on SmartDashboard or Shuffleboard and read and written by the robot program.

This example shows how to utilize Preferences to change the setpoint of a PID controller and the P constant. The code examples are adapted from the Arm Simulation example (Java, C++). You can run the Arm Simulation example in the Robot Simulator to see how to use the preference class and interact with it using the dashboards without needing a robot.

17.6.1 Initializing Preferences

Java

```
  public static final String kArmPositionKey = "ArmPosition";
  public static final String kArmPKey = "ArmP";

  // The P gain for the PID controller that drives this arm.
  public static final double kDefaultArmKp = 50.0;
  public static final double kDefaultArmSetpointDegrees = 75.0;
```

```
  // The P gain for the PID controller that drives this arm.
  private double m_armKp = Constants.kDefaultArmKp;
  private double m_armSetpointDegrees = Constants.kDefaultArmSetpointDegrees;
  public Arm() {
    m_encoder.setDistancePerPulse(Constants.kArmEncoderDistPerPulse);
    // Set the Arm position setpoint and P constant to Preferences if the keys don't␣
↪already exist
    Preferences.initDouble(Constants.kArmPositionKey, m_armSetpointDegrees);
    Preferences.initDouble(Constants.kArmPKey, m_armKp);
  }
```

C++

```
inline constexpr std::string_view kArmPositionKey = "ArmPosition";
inline constexpr std::string_view kArmPKey = "ArmP";

inline constexpr double kDefaultArmKp = 50.0;
inline constexpr units::degree_t kDefaultArmSetpoint = 75.0_deg;
```

```
Arm::Arm() {
  // Set the Arm position setpoint and P constant to Preferences if the keys
  // don't already exist
  frc::Preferences::InitDouble(kArmPositionKey, m_armSetpoint.value());
  frc::Preferences::InitDouble(kArmPKey, m_armKp);
}
```

Python

```
    kArmPositionKey = "ArmPosition"
    kArmPKey = "ArmP"

    # The P gain for the PID controller that drives this arm.
    kDefaultArmKp = 50.0
    kDefaultArmSetpointDegrees = 75.0
```

```
        # The P gain for the PID controller that drives this arm.
        self.armKp = Constants.kDefaultArmKp
        self.armSetpointDegrees = Constants.kDefaultArmSetpointDegrees

        # Set the Arm position setpoint and P constant to Preferences if the keys don
```

(continues on next page)

(continued from previous page)

```
↪'t already exist
        wpilib.Preferences.initDouble(
            Constants.kArmPositionKey, self.armSetpointDegrees
        )
        wpilib.Preferences.initDouble(Constants.kArmPKey, self.armKp)
```

Preferences are stored using a name, the key. It's helpful to store the key in a constant, like `kArmPositionKey` and `kArmPKey` in the code above to avoid typing it multiple times and avoid typos. We also declare variables, `kArmKp` and `armPositionDeg` to hold the data retrieved from preferences.

In `Arm` constructor, each key is checked to see if it already exists in the Preferences database. The `containsKey` method takes one parameter, the key to check if data for that key already exists in the preferences database. If it doesn't exist, a default value is written. The `setDouble` method takes two parameters, the key to write and the data to write. There are similar methods for other data types like booleans, ints, and strings.

If using the Command Framework, this type of code could be placed in the constructor of a Subsystem or Command.

17.6.2 Reading Preferences

Java

```
  public void loadPreferences() {
    // Read Preferences for Arm setpoint and kP on entering Teleop
    m_armSetpointDegrees = Preferences.getDouble(Constants.kArmPositionKey, m_
↪armSetpointDegrees);
    if (m_armKp != Preferences.getDouble(Constants.kArmPKey, m_armKp)) {
      m_armKp = Preferences.getDouble(Constants.kArmPKey, m_armKp);
      m_controller.setP(m_armKp);
    }
  }
```

C++

```
void Arm::LoadPreferences() {
  // Read Preferences for Arm setpoint and kP on entering Teleop
  m_armSetpoint = units::degree_t{
      frc::Preferences::GetDouble(kArmPositionKey, m_armSetpoint.value())};
  if (m_armKp != frc::Preferences::GetDouble(kArmPKey, m_armKp)) {
    m_armKp = frc::Preferences::GetDouble(kArmPKey, m_armKp);
    m_controller.SetP(m_armKp);
  }
}
```

Python

```
    def loadPreferences(self):
        # Read Preferences for Arm setpoint and kP on entering Teleop
        self.armSetpointDegrees = wpilib.Preferences.getDouble(
            Constants.kArmPositionKey, self.armSetpointDegrees
        )
        if self.armKp != wpilib.Preferences.getDouble(Constants.kArmPKey, self.armKp):
            self.armKp = wpilib.Preferences.getDouble(Constants.kArmPKey, self.armKp)
            self.controller.setP(self.armKp)
```

Reading a preference is easy. The `getDouble` method takes two parameters, the key to read, and a default value to use in case the preference doesn't exist. There are similar methods for other data types like booleans, ints, and strings.

Depending on the data that is stored in preferences, you can use it when you read it, such as the proportional constant above. Or you can store it in a variable and use it later, such as the setpoint, which is used in `telopPeriodic` below.

Java

```
  @Override
  public void teleopPeriodic() {
    if (m_joystick.getTrigger()) {
      // Here, we run PID control like normal.
      m_arm.reachSetpoint();
    } else {
      // Otherwise, we disable the motor.
      m_arm.stop();
    }
  }
```

```
  /** Run the control loop to reach and maintain the setpoint from the preferences. */
  public void reachSetpoint() {
    var pidOutput =
        m_controller.calculate(
            m_encoder.getDistance(), Units.degreesToRadians(m_armSetpointDegrees));
    m_motor.setVoltage(pidOutput);
  }
```

C++

```
void Robot::TeleopPeriodic() {
  if (m_joystick.GetTrigger()) {
    // Here, we run PID control like normal.
    m_arm.ReachSetpoint();
  } else {
    // Otherwise, we disable the motor.
    m_arm.Stop();
  }
}
```

```
void Arm::ReachSetpoint() {
  // Here, we run PID control like normal, with a setpoint read from
  // preferences in degrees.
  double pidOutput = m_controller.Calculate(
      m_encoder.GetDistance(), (units::radian_t{m_armSetpoint}.value()));
  m_motor.SetVoltage(units::volt_t{pidOutput});
}
```

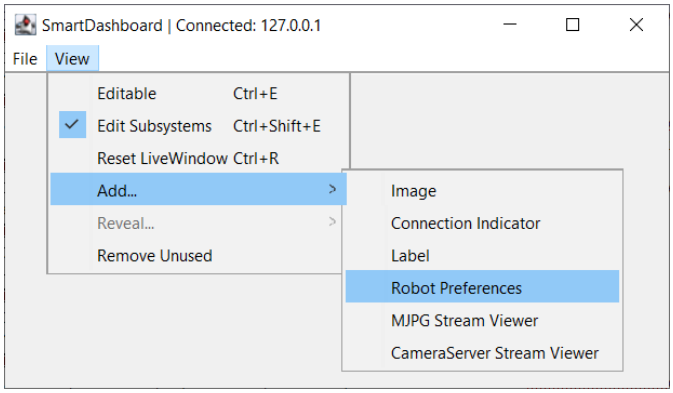
Python

```
    def teleopPeriodic(self):
        if self.joystick.getTrigger():
            # Here, we run PID control like normal.
            self.arm.reachSetpoint()
        else:
            # Otherwise, we disable the motor.
            self.arm.stop()
```

```
    def reachSetpoint(self):
        pidOutput = self.controller.calculate(
            self.encoder.getDistance(),
            units.degreesToRadians(self.armSetpointDegrees),
        )
        self.motor.setVoltage(pidOutput)
```

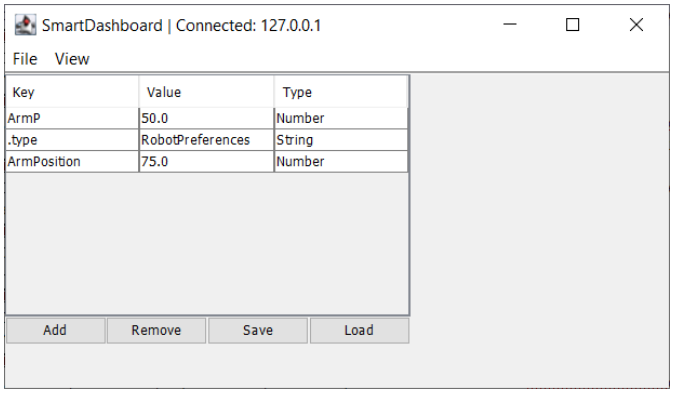
17.6.3 Using Preferences in SmartDashboard

Displaying Preferences in SmartDashboard



In the SmartDashboard, the Preferences display can be added to the display by selecting *View* then *Add...* then *Robot Preferences*. This reveals the contents of the preferences file stored in the roboRIO flash memory.

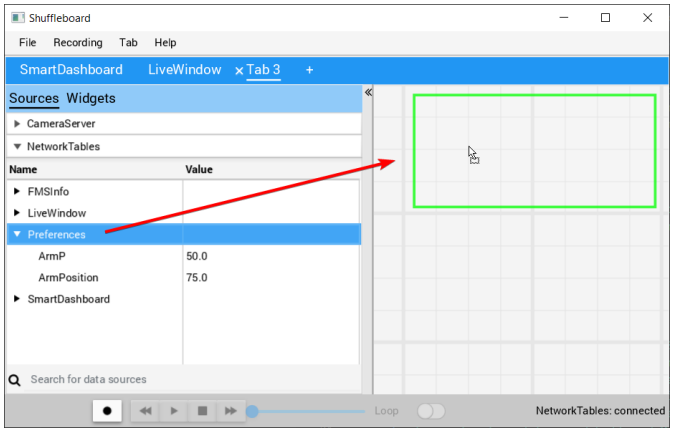
Editing Preferences in SmartDashboard



The values are shown here with the default values from the code. If the values need to be adjusted they can be edited here and saved.

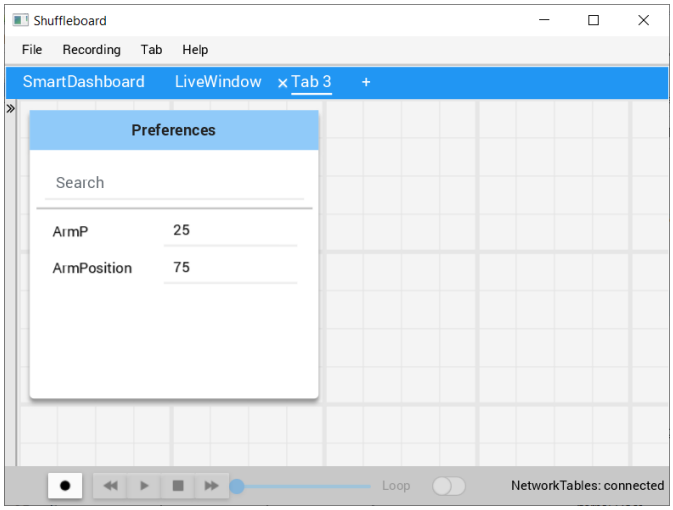
17.6.4 Using Preferences in Shuffleboard

Displaying Preferences in Shuffleboard



In Shuffleboard, the Preferences display can be added to the display by dragging the preferences field from the sources window. This reveals the contents of the preferences file stored in the roboRIO flash memory.

Editing Preferences in Shuffleboard



The values are shown here with the default values from the code. If the values need to be adjusted they can be edited here.

17.7 Robot Project Deploy Directory

The deploy directory is a part of an FRC project, used to store files that are transferred to the roboRIO during the deployment process. This directory allows teams to include additional resources, such as configuration files, scripts, or other assets, that their robot code may need at runtime.

17.7.1 Location of the Deploy Directory

In a standard GradleRIO project, the deploy directory is located under the `src/main` folder. Any files placed in this directory will automatically be copied to the roboRIO when the code is deployed.

17.7.2 How the Deploy Directory Works

When you run the `deploy` task, GradleRIO packages the contents of the deploy directory and transfers them to the roboRIO. By default, these files are placed in the `/home/lvuser/deploy` directory on the roboRIO. This ensures that your robot code can access these files during runtime.

Example Use Cases

Here are some common examples of how teams use the `deploy` directory:

1. **Configuration Files** Teams may include configuration files (e.g., JSON, YAML, or XML) that define robot-specific settings, such as PID constants, swerve configuration, or autonomous routines. The *Preferences class* provides another method to store configuration on the roboRIO.
2. **Path Planning Files** For teams using path planning libraries, trajectory files such as those from *Choreo* and PathPlanner can be stored in the deploy directory and loaded by the robot code.

Accessing Deploy Files in Code

To access files from the deploy directory in your robot code, use the `Filesystem` class (Java, C++). The `getDeployDirectory` method returns the path to the deploy directory. This ensures that your code can locate the files regardless of whether the code is running on the roboRIO or in simulation on your computer.

17.7.3 Deleting Unused Deploy Files

By default the deploy directory in your project is transferred to the roboRIO when code is deployed. It is initiated by this section of the `build.gradle` file.

```
32                // Static files artifact
33                frcStaticFileDeploy(getArtifactTypeClass('FileTreeArtifact')) {
34                    files = project.fileTree('src/main/deploy')
35                    directory = '/home/lvuser/deploy'
36                    deleteOldFiles = false // Change to true to delete files on␣
↪roboRIO that no
37                                           // longer exist in deploy directory of␣
↪this project
38                }
```

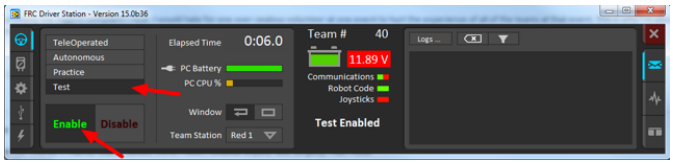
This will overwrite any duplicate files found in the `/home/lvuser/deploy` directory on the RIO and copy over any additional not present there. If `deleteOldFiles` is false it will not remove any files no longer present in the project deploy directory. Changing it to *true* helps prevent programs like *Choreo* and PathPlanner from getting confused by files that were deleted locally but still exist on the roboRIO.

If you want to manage the roboRIO files directly, the *FTP documentation* provides one method to do so.

17.8 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.

17.8.1 Enabling Test Mode



Test mode on the robot can be enabled from the Driver Station just like autonomous or teleop. To enable test mode in the Driver Station, select the "Test" button and enable the robot. The test mode code will then run.

17.8.2 Adding Test mode code to your robot code

When in test mode, the `testInit` method is run once, and the `testPeriodic` method is run once per tick, in addition to `robotPeriodic`, similar to teleop and autonomous control modes.

Adding test mode can be as painless as calling your already written Teleop methods from Test. Or you can write special code to try out a new feature that is only run in Test mode, before integrating it into your teleop or autonomous code. You could even write code to move all motors and check all sensors to help the pit crew!

17.8.3 LiveWindow in Test Mode

Important

Since 2024, LiveWindow in Test Mode is disabled by default! See *Enabling LiveWindow in Test Mode* to enable it.

With LiveWindow, all actuator outputs can be controlled on the Dashboard and all sensor values can be seen. PID Controllers can also be tuned. The sensors and actuators are added automatically, no code is necessary. See *SmartDashboard: Test Mode and Live Window* for more details.

17.9 Reading Stacktraces

`An unexpected error has occurred.`

When your robot code hits an unexpected error, you will see this message show up in some console output (Driver Station or RioLog). You'll probably also notice your robot abruptly stop, or possibly never move. These unexpected errors are called *unhandled exceptions*.

When an unhandled exception occurs, it means that your code has one or more bugs which need to be fixed.

This article will explore some of the tools and techniques involved in finding and fixing those bugs.

17.9.1 What's a "Stack Trace"?

The `unexpected error has occurred` message is a signal that a *stack trace* has been printed out.

In Java and C++, the *call stack* data structure is used to store information about which function or method is currently being executed.

A *stack trace* prints information about what was on this stack when the unhandled exception occurred. This points you to the lines of code which were running just before the problem happened. While it doesn't always point you to the exact *root cause* of your issue, it's usually the best place to start looking.

17.9.2 What's an "Unhandled Exception"?

An unrecoverable error is any condition which arises where the processor cannot continue executing code. It almost always implies that, even though the code compiled and started running, it no longer makes sense for execution to continue.

In almost all cases, the root cause of an unhandled exception is code that isn't correctly implemented. It almost never implies that any hardware has malfunctioned.

17.9.3 So How Do I Fix My Issue?

Read the Stack Trace

To start, search above the `unexpected error has occurred` for the stack trace.

Java

In Java, it should look something like this:

```
Error at frc.robot.Robot.robotInit(Robot.java:24): Unhandled exception: java.lang.
↪NullPointerException
        at frc.robot.Robot.robotInit(Robot.java:24)
        at edu.wpi.first.wpilibj.TimedRobot.startCompetition(TimedRobot.java:94)
        at edu.wpi.first.wpilibj.RobotBase.runRobot(RobotBase.java:335)
        at edu.wpi.first.wpilibj.RobotBase.lambda$startRobot$0(RobotBase.java:387)
        at java.base/java.lang.Thread.run(Thread.java:834)
```

There's a few important things to pick out of here:

- There was an `Error`
- The error was due to an `Unhandled exception`
- The exception was a `java.lang.NullPointerException`
- **The error happened while running line 24 inside of `Robot.java`**
  - `robotInit` was the name of the method executing when the error happened.
- `robotInit` is a function in the `frc.robot.Robot` package (AKA, your team's code)
- `robotInit` was called from a number of functions from the `edu.wpi.first.wpilibj` package (AKA, the WPILib libraries)

The list of indented lines starting with the word `at` represent the state of the *stack* at the time the error happened. Each line represents one method, which was *called by* the method right below it.

For example, If the error happened deep inside your codebase, you might see more entries on the stack:

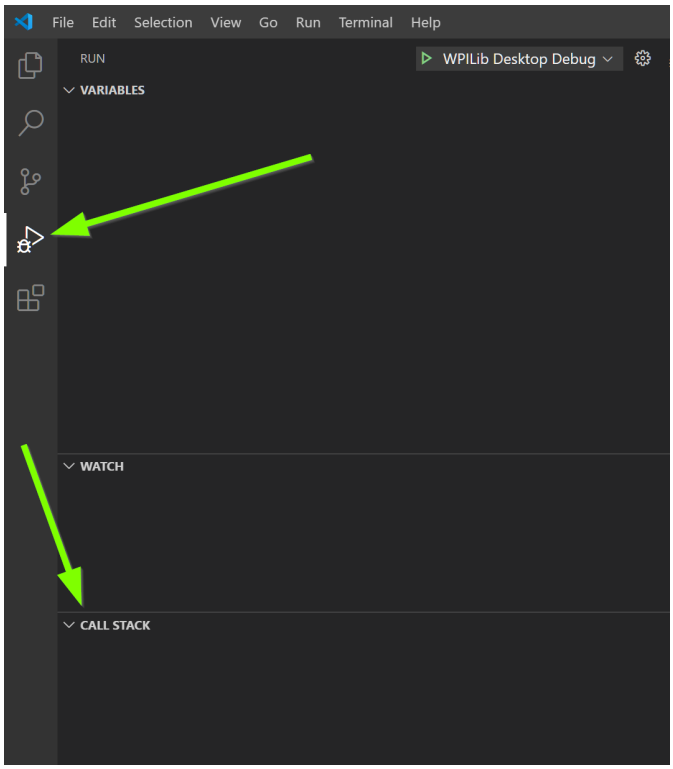
```
Error at frc.robot.Robot.buggyMethod(TooManyBugs.java:1138): Unhandled exception:␣
↪java.lang.NullPointerException
        at frc.robot.Robot.buggyMethod(TooManyBugs.java:1138)
        at frc.robot.Robot.barInit(Bar.java:21)
        at frc.robot.Robot.fooInit(Foo.java:34)
        at frc.robot.Robot.robotInit(Robot.java:24)
        at edu.wpi.first.wpilibj.TimedRobot.startCompetition(TimedRobot.java:94)
        at edu.wpi.first.wpilibj.RobotBase.runRobot(RobotBase.java:335)
        at edu.wpi.first.wpilibj.RobotBase.lambda$startRobot$0(RobotBase.java:387)
        at java.base/java.lang.Thread.run(Thread.java:834)
```

In this case: `robotInit` called `fooInit`, which in turn called `barInit`, which in turn called `buggyMethod`. Then, during the execution of `buggyMethod`, the `NullPointerException` occurred.

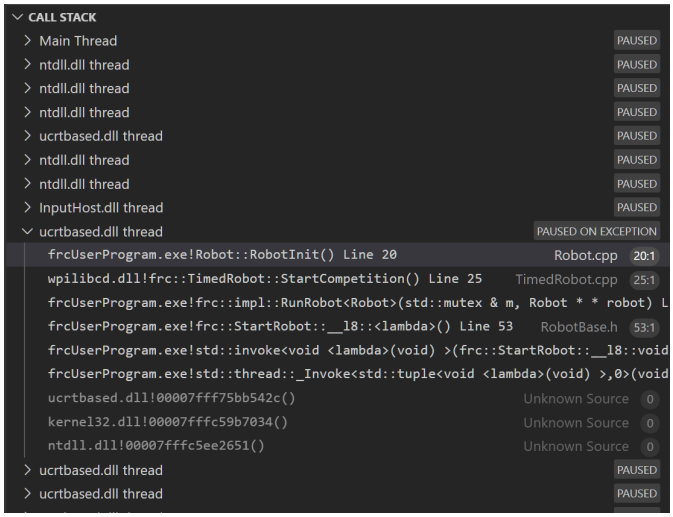
C++

Java will usually produce stack traces automatically when programs run into issues. C++ will require more digging to extract the same info. Usually, a single-step debugger will need to be hooked up to the executing robot program.

Stack traces can be found in the debugger tab of VS Code:



Stack traces in C++ will generally look similar to this:



There's a few important things to pick out of here:

- The code execution is currently paused.
- The reason it paused was one thread having an `exception`

- **The error happened while running line `20` inside of `Robot.cpp`**
  - `RobotInit` was the name of the method executing when the error happened.
- `RobotInit` is a function in the `Robot::` namespace (AKA, your team's code)
- `RobotInit` was called from a number of functions from the `frc::` namespace (AKA, the WPILib libraries)

This "call stack" window represents the state of the *stack* at the time the error happened. Each line represents one method, which was *called by* the method right below it.

The examples in this page assume you are running code examples in simulation, with the debugger connected and watching for unexpected errors. Similar techniques should apply while running on a real robot.

Perform Code Analysis

Once you've found the stack trace, and found the lines of code which are triggering the unhandled exception, you can start the process of determining root cause.

Often, just looking in (or near) the problematic location in code will be fruitful. You may notice things you forgot, or lines which don't match an example you're referencing.

Note

Developers who have lots of experience working with code will often have more luck looking at code than newer folks. That's ok, don't be discouraged! The experience will come with time.

A key strategy for analyzing code is to ask the following questions:

- When was the last time the code "worked" (I.e., didn't have this particular error)?
- What has changed in the code between the last working version, and now?

Frequent testing and careful code changes help make this particular strategy more effective.

Run the Single Step Debugger

Sometimes, just looking at code isn't enough to spot the issue. The *single-step debugger* is a great option in this case - it allows you to inspect the series of events leading up to the unhandled exception.

Search for More Information

Google is a phenomenal resource for understanding the root cause of errors. Searches involving the programming language and the name of the exception will often yield good results on more explanations for what the error means, how it comes about, and potential fixes.

Seeking Outside Help

If all else fails, you can seek out advice and help from others (both in-person and online). When working with folks who aren't familiar with your codebase, it's very important to provide the following information:

- Access to your source code, (EX: *on github.com*)
- The **full text** of the error, including the full stack trace.

17.9.4 Common Examples & Patterns

There are a number of common issues which result in runtime exceptions.

Null Pointers and References

Both C++ and Java have the concept of "null" - they use it to indicate something which has not yet been initialized, and does not refer to anything meaningful.

Manipulating a "null" reference will produce a runtime error.

For example, consider the following code:

JAVA

```
19 PWMSparkMax armMotorCtrl;
20 @Override
21 public void robotInit() {
22    armMotorCtrl.setInverted(true);
23 }
```

C++

```
17 class Robot : public frc::TimedRobot {
18  public:
19   void RobotInit() override {
20      motorRef->SetInverted(false);
21   }
22  private:
23   frc::PWMVictorSPX m_armMotor{0};
24   frc::PWMVictorSPX* motorRef;
25 };
```

When run, you'll see output that looks like this:

Java

```
********** Robot program starting **********
Error at frc.robot.Robot.robotInit(Robot.java:23): Unhandled exception: java.lang.
↪NullPointerException
        at frc.robot.Robot.robotInit(Robot.java:23)
        at edu.wpi.first.wpilibj.TimedRobot.startCompetition(TimedRobot.java:107)
        at edu.wpi.first.wpilibj.RobotBase.runRobot(RobotBase.java:373)
        at edu.wpi.first.wpilibj.RobotBase.startRobot(RobotBase.java:463)
        at frc.robot.Main.main(Main.java:23)
Warning at edu.wpi.first.wpilibj.RobotBase.runRobot(RobotBase.java:388): The robot␣
↪program quit unexpectedly. This is usually due to a code error.
  The above stacktrace can help determine where the error occurred.
  See https://wpilib.org/stacktrace for more information.
Error at edu.wpi.first.wpilibj.RobotBase.runRobot(RobotBase.java:395): The␣
↪startCompetition() method (or methods called by it) should have handled the␣
↪exception above.
```

Reading the stack trace, you can see that the issue happened inside of the `robotInit()` function, on line 23, and the exception involved "Null Pointer".

By going to line 23, you can see there is only one thing which could be null - `armMotorCtrl`. Looking further up, you can see that the `armMotorCtrl` object is declared, but never instantiated.

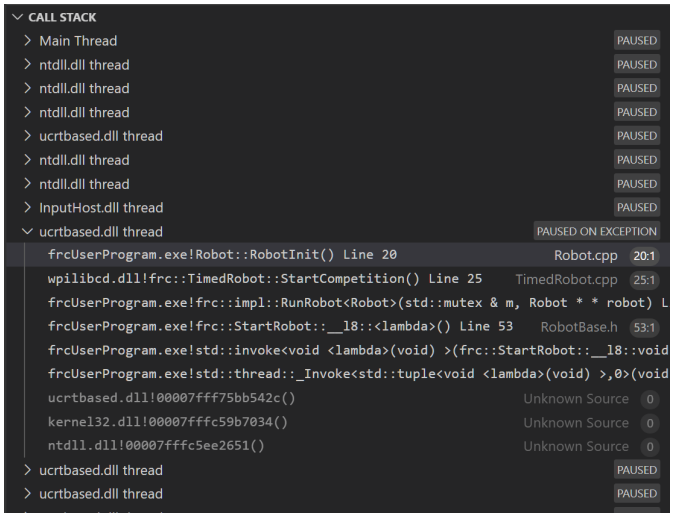
Alternatively, you can step through lines of code with the single step debugger, and stop when you hit line 23. Inspecting the `armMotorCtrl` object at that point would show that it is null.

C++

```
Exception has occurred: W32/0xc0000005
Unhandled exception thrown: read access violation.
this->motorRef was nullptr.
```

In Simulation, this will show up in a debugger window that points to line 20 in the above buggy code.

You can view the full stack trace by clicking the debugger tab in VS Code:



The error is specific - our member variable `motorRef` was declared, but never assigned a value. Therefore, when we attempt to use it to call a method using the `->` operator, the exception occurs.

The exception states its type was `nullptr`.

Fixing Null Object Issues

Generally, you will want to ensure each reference has been initialized before using it. In this case, there is a missing line of code to instantiate the `armMotorCtrl` before calling the `setInverted()` method.

A functional implementation could look like this:

JAVA

```
19 PWMSparkMax armMotorCtrl;
20 @Override
21 public void robotInit() {
22    armMotorCtrl = new PWMSparkMax(0);
23    armMotorCtrl.setInverted(true);
24 }
```

C++

```
17 class Robot : public frc::TimedRobot {
18  public:
19   void RobotInit() override {
20      motorRef = &m_armMotor;
21      motorRef->SetInverted(false);
22   }
23  private:
24   frc::PWMVictorSPX m_armMotor{0};
25   frc::PWMVictorSPX* motorRef;
26 };
```

Divide by Zero

It is not generally possible to divide an integer by zero, and expect reasonable results. Most processors (including the roboRIO) will raise an Unhandled Exception.

For example, consider the following code:

JAVA

```
int armLengthRatio;
int elbowToWrist_in = 39;
int shoulderToElbow_in = 0; //TODO
@Override
public void robotInit() {
   armLengthRatio = elbowToWrist_in / shoulderToElbow_in;
}
```

C++

```
class Robot : public frc::TimedRobot {
 public:
  void RobotInit() override {
     armLengthRatio = elbowToWrist_in / shoulderToElbow_in;
  }
 private:
  int armLengthRatio;
  int elbowToWrist_in = 39;
  int shoulderToElbow_in = 0; //TODO
};
```

When run, you'll see output that looks like this:

Java

```
      ********** Robot program starting **********
Error at frc.robot.Robot.robotInit(Robot.java:24): Unhandled exception: java.lang.
↪ArithmeticException: / by zero
        at frc.robot.Robot.robotInit(Robot.java:24)
        at edu.wpi.first.wpilibj.TimedRobot.startCompetition(TimedRobot.java:107)
        at edu.wpi.first.wpilibj.RobotBase.runRobot(RobotBase.java:373)
        at edu.wpi.first.wpilibj.RobotBase.startRobot(RobotBase.java:463)
        at frc.robot.Main.main(Main.java:23)
Warning at edu.wpi.first.wpilibj.RobotBase.runRobot(RobotBase.java:388): The robot␣
↪program quit unexpectedly. This is usually due to a code error.
  The above stacktrace can help determine where the error occurred.
  See https://wpilib.org/stacktrace for more information.
Error at edu.wpi.first.wpilibj.RobotBase.runRobot(RobotBase.java:395): The␣
↪startCompetition() method (or methods called by it) should have handled the␣
↪exception above.
```

Looking at the stack trace, we can see a `java.lang.ArithmeticException: / by zero` exception has occurred on line 24. If you look at the two variables which are used on the right-hand side of the = operator, you might notice one of them has been initialized to zero. Looks like someone forgot to update it! Furthermore, the zero-value variable is used in the denominator of a division operation. Hence, the divide by zero error happens.

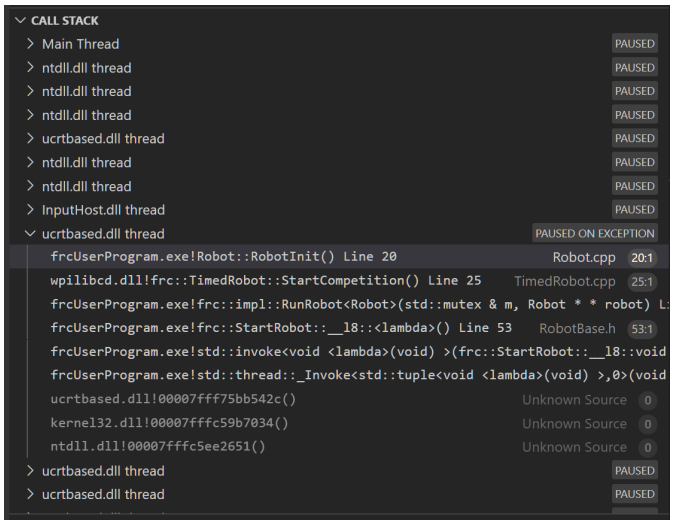
Alternatively, by running the single-step debugger and stopping on line 24, you could inspect the value of all variables to discover `shoulderToElbow_in` has a value of `0`.

C++

```
Exception has occurred: W32/0xc0000094
Unhandled exception at 0x00007FF71B223CD6 in frcUserProgram.exe: 0xC0000094: Integer␣
↪division by zero.
```

In Simulation, this will show up in a debugger window that points to line 20 in the above buggy code.

You can view the full stack trace by clicking the debugger tab in VS Code:



Looking at the message, we see the error is described as `Integer division by zero`. If you look at the two variables which are used on the right-hand side of the = operator on line 20, you might notice one of them has been initialized to zero. Looks like someone forgot to update it! Furthermore, the zero-value variable is used in the denominator of a division operation. Hence, the divide by zero error happens.

Note that the error messages might look slightly different on the roboRIO, or on an operating system other than windows.

Fixing Divide By Zero Issues

Divide By Zero issues can be fixed in a number of ways. It's important to start by thinking about what a zero in the denominator of your calculation *means*. Is it plausible? Why did it happen in the particular case you saw?

Sometimes, you just need to use a different number other than 0.

A functional implementation could look like this:

JAVA

```java
int armLengthRatio;
int elbowToWrist_in = 39;
int shoulderToElbow_in = 3;
@Override
public void robotInit() {
   armLengthRatio = elbowToWrist_in / shoulderToElbow_in;
}
```

C++

```cpp
class Robot : public frc::TimedRobot {
 public:
  void RobotInit() override {
     armLengthRatio = elbowToWrist_in / shoulderToElbow_in;
  }
 private:
  int armLengthRatio;
  int elbowToWrist_in = 39;
  int shoulderToElbow_in = 3
};
```

Alternatively, if zero *is* a valid value, adding `if/else` statements around the calculation can help you define alternate behavior to avoid making the processor perform a division by zero.

Finally, changing variable types to be `float` or `double` can help you get around the issue - floating-point numbers have special values like `NaN` to represent the results of a divide-by-zero operation. However, you may still have to handle this in code which consumes that calculation's value.

HAL Resource Already Allocated

A very common FRC-specific error occurs when the code attempts to put two hardware-related entities on the same HAL resource (usually, roboRIO IO pin).

For example, consider the following code:

JAVA

```java
PWMSparkMax leftFrontMotor;
PWMSparkMax leftRearMotor;
@Override
public void robotInit() {
   leftFrontMotor = new PWMSparkMax(0);
   leftRearMotor = new PWMSparkMax(0);
}
```

C++

```
class Robot : public frc::TimedRobot {
 public:
  void RobotInit() override {
     m_frontLeftMotor.Set(0.5);
     m_rearLeftMotor.Set(0.25);
  }
 private:
  frc::PWMVictorSPX m_frontLeftMotor{0};
  frc::PWMVictorSPX m_rearLeftMotor{0};
};
```

When run, you'll see output that looks like this:

Java

```
********** Robot program starting **********
Error at frc.robot.Robot.robotInit(Robot.java:25): Unhandled exception: edu.wpi.first.
↪hal.util.AllocationException: Code: -1029
PWM or DIO 0 previously allocated.
Location of the previous allocation:
        at frc.robot.Robot.robotInit(Robot.java:24)
        at edu.wpi.first.wpilibj.TimedRobot.startCompetition(TimedRobot.java:107)
        at edu.wpi.first.wpilibj.RobotBase.runRobot(RobotBase.java:373)
        at edu.wpi.first.wpilibj.RobotBase.startRobot(RobotBase.java:463)
        at frc.robot.Main.main(Main.java:23)
Location of the current allocation:
        at edu.wpi.first.hal.PWMJNI.initializePWMPort(Native Method)
        at edu.wpi.first.wpilibj.PWM.<init>(PWM.java:66)
        at edu.wpi.first.wpilibj.motorcontrol.PWMMotorController.<init>
↪(PWMMotorController.java:27)
        at edu.wpi.first.wpilibj.motorcontrol.PWMSparkMax.<init>(PWMSparkMax.java:35)
        at frc.robot.Robot.robotInit(Robot.java:25)
        at edu.wpi.first.wpilibj.TimedRobot.startCompetition(TimedRobot.java:107)
        at edu.wpi.first.wpilibj.RobotBase.runRobot(RobotBase.java:373)
        at edu.wpi.first.wpilibj.RobotBase.startRobot(RobotBase.java:463)
        at frc.robot.Main.main(Main.java:23)
Warning at edu.wpi.first.wpilibj.RobotBase.runRobot(RobotBase.java:388): The robot␣
↪program quit unexpectedly. This is usually due to a code error.
  The above stacktrace can help determine where the error occurred.
  See https://wpilib.org/stacktrace for more information.
Error at edu.wpi.first.wpilibj.RobotBase.runRobot(RobotBase.java:395): The␣
↪startCompetition() method (or methods called by it) should have handled the␣
↪exception above.
```

This stack trace shows that a `edu.wpi.first.hal.util.AllocationException` has occurred. It also gives the helpful message: `PWM or DIO 0 previously allocated.`.

Looking at our stack trace, we see two stack traces. The first stack trace shows that the first allocation occurred in `Robot.java:25`. The second stack trace shows that the error *actually* happened deep within WPILib. However, we should start by looking in our own code. Halfway through the stack trace, you can find a reference to the last line of the team's robot code that called into WPILib: `Robot.java:25`.

Taking a peek at the code, we see line 24 is where the first motor controller is declared

and line 25 is where the second motor controller is declared. We can also note that *both* motor controllers are assigned to PWM output `0`. This doesn't make logical sense, and isn't physically possible. Therefore, WPILib purposely generates a custom error message and exception to alert the software developers of a non-achievable hardware configuration.

C++

In C++, you won't specifically see a stacktrace from this issue. Instead, you'll get messages which look like the following:

```
Error at PWM [C::31]: PWM or DIO 0 previously allocated.
Location of the previous allocation:
        at frc::PWM::PWM(int, bool) + 0x50 [0xb6f01b68]
        at frc::PWMMotorController::PWMMotorController(std::basic_string_view<char,␣
↪std::char_traits<char> >, int) + 0x70 [0xb6ef7d50]
        at frc::PWMVictorSPX::PWMVictorSPX(int) + 0x3c [0xb6e9af1c]
        at void frc::impl::RunRobot<Robot>(wpi::priority_mutex&, Robot**) + 0xa8␣
↪[0x13718]
        at int frc::StartRobot<Robot>() + 0x3d4 [0x13c9c]
        at __libc_start_main + 0x114 [0xb57ec580]
Location of the current allocation:: Channel 0
        at  + 0x5fb5c [0xb6e81b5c]
        at frc::PWM::PWM(int, bool) + 0x334 [0xb6f01e4c]
        at frc::PWMMotorController::PWMMotorController(std::basic_string_view<char,␣
↪std::char_traits<char> >, int) + 0x70 [0xb6ef7d50]
        at frc::PWMVictorSPX::PWMVictorSPX(int) + 0x3c [0xb6e9af1c]
        at void frc::impl::RunRobot<Robot>(wpi::priority_mutex&, Robot**) + 0xb4␣
↪[0x13724]
        at int frc::StartRobot<Robot>() + 0x3d4 [0x13c9c]
        at __libc_start_main + 0x114 [0xb57ec580]
Error at RunRobot: Error: The robot program quit unexpectedly. This is usually due to␣
↪a code error.
  The above stacktrace can help determine where the error occurred.
  See https://wpilib.org/stacktrace for more information.
        at void frc::impl::RunRobot<Robot>(wpi::priority_mutex&, Robot**) + 0x1c8␣
↪[0x13838]
        at int frc::StartRobot<Robot>() + 0x3d4 [0x13c9c]
        at __libc_start_main + 0x114 [0xb57ec580]
terminate called after throwing an instance of 'frc::RuntimeError'
  what():  PWM or DIO 0 previously allocated.
Location of the previous allocation:
        at frc::PWM::PWM(int, bool) + 0x50 [0xb6f01b68]
        at frc::PWMMotorController::PWMMotorController(std::basic_string_view<char,␣
↪std::char_traits<char> >, int) + 0x70 [0xb6ef7d50]
        at frc::PWMVictorSPX::PWMVictorSPX(int) + 0x3c [0xb6e9af1c]
        at void frc::impl::RunRobot<Robot>(wpi::priority_mutex&, Robot**) + 0xa8␣
↪[0x13718]
        at int frc::StartRobot<Robot>() + 0x3d4 [0x13c9c]
        at __libc_start_main + 0x114 [0xb57ec580]
Location of the current allocation:: Channel 0
```

The key thing to notice here is the string, `PWM or DIO 0 previously allocated.`. That string is your primary clue that something in code has incorrectly "doubled up" on pin 0 usage.

The message example above was generated on a roboRIO. If you are running in simulation, it might look different.

Fixing HAL Resource Already Allocated Issues

`HAL: Resource already allocated` are some of the most straightforward errors to fix. Just spend a bit of time looking at the electrical wiring on the robot, and compare that to what's in code.

In the example, the left motor controllers are plugged into *PWM* ports `0` and `1`. Therefore, corrected code would look like this:

JAVA

```
PWMSparkMax leftFrontMotor;
PWMSparkMax leftRearMotor;
@Override
public void robotInit() {
   leftFrontMotor = new PWMSparkMax(0);
   leftRearMotor = new PWMSparkMax(1);
}
```

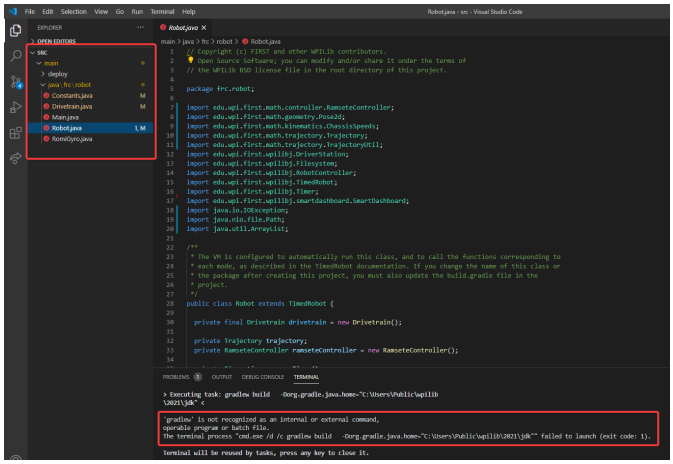
C++

```
class Robot : public frc::TimedRobot {
 public:
  void RobotInit() override {
     m_frontLeftMotor.Set(0.5);
     m_rearLeftMotor.Set(0.25);
  }
  private:
   frc::PWMVictorSPX m_frontLeftMotor{0};
   frc::PWMVictorSPX m_rearLeftMotor{1};
};
```

Another way this error can manifest for teams using the Command framework is if a Subsystem is instantiated twice, which causes the hardware in the subsystem to be instantiated twice.

gradlew is not recognized...

`gradlew is not recognized as an internal or external command` is a common error that can occur when the project or directory that you are currently in does not contain a `gradlew` file. This usually occurs when you open the wrong directory.



In the above screenshot, you can see that the left-hand sidebar does not contain many files. At a minimum, VS Code needs a couple of files to properly build and deploy your project.

- `gradlew`
- `build.gradle`
- `gradlew.bat`

If you do not see any one of the above files in your project directory, then you have two possible causes.

- A corrupt or bad project.
- You are in the wrong directory.

Fixing gradlew is not recognized...

`gradlew is not recognized...` is a fairly easy problem to fix. First identify the problem source:

Are you in the wrong directory? - Verify that the project directory is the correct directory and open this.

Is your project missing essential files? - This issue is more complex to solve. The recommended solution is to *recreate your project* and manually copy necessary code in.

17.10 Treating Functions as Data

Regardless of programming language, one of the first things anyone learns to do when programming a computer is to write a function (also known as a “method” or a “subroutine”). Functions are a fundamental part of organized code - writing functions lets us avoid duplicating the same piece of code over and over again. Instead of writing duplicated sections of code, we call a single function that contains the code we want to execute from multiple places (provided we named the function well, the function name is also easier to read than the code itself!). If the section of code needs some additional information about its surrounding context to run, we pass those to the function as “parameters”, and if it needs to yield something back to the rest of the code once it finishes, we call that a “return value” (together, the parameters and return value are called the function’s “signature”);

Sometimes, we need to pass functions from one part of the code to another part of the code. This might seem like a strange concept, if we're used to thinking of functions as part of a class definition rather than objects in their own right. But at a basic level, functions are just data - in the same way we can store an `integer` or a `double` as a variable and pass it around our program, we can do the same thing with a function. A variable whose value is a function is called a "functional interface" in Java, and a "function pointer" or "functor" in C++.

17.10.1 Why Would We Want to Treat Functions as Data?

Typically, code that calls a function is coupled to (depends on) the definition of the function. While this occurs all the time, it becomes problematic when the code *calling* the function (for example, WPILib) is developed independently and without direct knowledge of the code that *defines* the function (for example, code from an FRC team). Sometimes we solve this challenge through the use of class interfaces, which define collections of data and functions that are meant to be used together. However, often we really only have a dependency on a *single function*, rather than on an *entire class*.

For example, WPILib offers several ways for users to execute certain code whenever a joystick button is pressed - one of the easiest and cleanest ways to do this is to allow the user to *pass a function* to one of the WPILib joystick methods. This way, the user only has to write the code that deals with the interesting and team-specific things (e.g., "move my robot arm") and not the boring, error-prone, and universal thing ("properly read button inputs from a standard joystick").

For another example, the *Command-based framework* is built on `Command` objects that refer to methods defined on various `Subsystem` classes. Many of the included `Command` types (such as `InstantCommand` and `RunCommand`) work with *any* function - not just functions associated with a single `Subsystem`. To support building commands generically, we need to support passing functions from a `Subsystem` (which interacts with the hardware) to a `Command` (which interacts with the scheduler).

In these cases, we want to be able to pass a single function as a piece of data, as if it were a variable - it doesn't make sense to ask the user to provide an entire class, when we really just want them to give us a single appropriately-shaped function.

It's important that *passing* a function is not the same as *calling* a function. When we call a function, we execute the code inside of it and either receive a return value, cause some side-effects elsewhere in the code, or both. When we *pass* a function, nothing in particular happens *immediately*. Instead, by passing the function we are allowing some *other* code to call the function *in the future*. Seeing the name of a function in code does not always mean that the code in the function is being run!

Inside of code that passes a function, we will see some syntax that either refers to the name of an existing function in a special way, or else defines a new function to be passed inside of the call expression. The specific syntax needed (and the rules around it) depends on which programming language we are using.

17.10.2 Treating Functions as Data in Java

Java represents functions-as-data as instances of functional interfaces. A "functional interface" is a special kind of class that has only a single method - since Java was originally designed strictly for object-oriented programming, it has no way of representing a single function detached from a class. Instead, it defines a particular group of classes that *only* represent single functions. Each type of function signature has its own functional interface, which is an interface with a single function definition of that signature.

This might sound complicated, but in the context of WPILib we don't really need to worry much about using the functional interfaces themselves - the code that does that is internal to WPILib. Instead, all we need to know is how to pass a function that we've written to a method that takes a functional interface as a parameter. For a simple example, consider the signature of `Commands.runOnce` (which creates an `InstantCommand` that, when scheduled, runs the given function once and then terminates):

> **Note**
>
> The `requirements` parameter is explained in the *Command-based documentation*, and will not be discussed here.

```
public static Command runOnce(Runnable action, Subsystem... requirements)
```

`runOnce` expects us to give it a `Runnable` parameter (named `action`). A `Runnable` is the Java term for a function that takes no parameters and returns no value. When we call `runOnce`, we need to give it a function with no parameters and no return value. There are two ways to do this: we can refer to some existing function using a "method reference", or we can define the function we want inline using a "lambda expression".

Method References

A method reference lets us pass an already-existing function as our `Runnable`:

```
// Create an InstantCommand that runs the `resetEncoders` method of the `drivetrain`␣
↪object
Command disableCommand = runOnce(drivetrain::resetEncoders, drivetrain);
```

The expression `drivetrain::resetEncoders` is a reference to the `resetEncoders` method of the `drivetrain` object. It is not a method *call* - this line of code does not *itself* reset the encoders of the drivetrain. Instead, it returns a `Command` that will do so *when it is scheduled*.

Remember that in order for this to work, `resetEncoders` must be a `Runnable` - that is, it must take no parameters and return no value. So, its signature must look like this:

```
// void because it returns no parameters, and has an empty parameter list
public void resetEncoders()
```

If the function signature does not match this, Java will not be able to interpret the method reference as a `Runnable` and the code will not compile. Note that all we need to do is make sure that the signature matches the signature of the single method in the `Runnable` functional interface - we don't need to *explicitly* name it as a `Runnable`.

Lambda Expressions in Java

If we do not already have a named function that does what we want, we can define a function "inline" - that means, right inside of the call to `runOnce`! We do this by writing our function with a special syntax that uses an "arrow" symbol to link the argument list to the function body:

```
// Create an InstantCommand that runs the drive forward at half speed
Command driveHalfSpeed = runOnce(() -> { drivetrain.arcadeDrive(0.5, 0.0); },␣
↪drivetrain);
```

Java calls `() -> { drivetrain.arcadeDrive(0.5, 0.0); }` a "lambda expression"; it may be less-confusingly called an "arrow function", "inline function", or "anonymous function" (because it has no name). While this may look a bit funky, it is just another way of writing a function - the parentheses before the arrow are the function's argument list, and the code contained in the brackets is the function body. The "lambda expression" here represents a function that calls `drivetrain.arcadeDrive` with a specific set of parameters - note again that this does not *call* the function, but merely defines it and passes it to the `Command` to be run later when the `Command` is scheduled.

As with method references, we do not need to *explicitly* name the lambda expression as a `Runnable` - Java can infer that our lambda expression is a `Runnable` so long as its signature matches that of the single method in the `Runnable` interface. Accordingly, our lambda takes no arguments and has no return statement - if it did not match the `Runnable` contract, our code would fail to compile.

Capturing State in Java Lambda Expressions

In the above example, our function body references an object that lives outside of the function itself (namely, the `drivetrain` object). This is called a "capture" of a variable from the surrounding code (which is sometimes called the "outer scope" or "enclosing scope"). Usually the captured variables are either local variables from the enclosing method body in which the lambda expression is defined, or else fields of an enclosing class definition in which that method is defined.

In Java capturing state is a fairly safe thing to do in general, with one major caveat: we can only capture state that is "effectively final". That means it is only legal to capture a variable from the enclosing scope if that variable is never reassigned after initialization. Note that this does not mean that the captured state cannot change: Remember that Java objects are references, so the object that the reference *points to* may change after capture - but the reference itself cannot be made to point to another object.

This means we can only capture primitive types (like `int`, `double`, and `boolean`) if they're constants. If we want to capture a state variable that can change, it *must be wrapped in a mutable object*.

Syntactic Sugar for Java Lambda Expressions

The full lambda expression syntax can be needlessly verbose in some cases. To help with this, Java lets us take some shortcuts (called "syntactic sugar") in cases where some of the notation is redundant.

Omitting Function Body Brackets for One-Line Lambdas

If the function body of our lambda expression is only one line, Java lets us omit the brackets around the function body. When omitting function brackets, we also omit trailing semicolons And the *return* keyword.

So, our `Runnable` lambda above could instead be written:

```
// Create an InstantCommand that runs the drive forward at half speed
Command driveHalfSpeed = runOnce(() -> drivetrain.arcadeDrive(0.5, 0.0), drivetrain);
```

Omitting Parentheses around Single Lambda Parameters

If the lambda expression is for a functional interface that takes only a single argument, we can omit the parenthesis around the parameter list:

```
// We can write this lambda with no parenthesis around its single argument
IntConsumer exampleLambda = (a -> System.out.println(a));
```

17.10.3 Treating Functions as Data in C++

C++ has a number of ways to treat functions as data. For the sake of this article, we'll only talk about the parts that are relevant to using WPILibC.

In WPILibC, function types are represented with the `std::function` class (https://en.cppreference.com/w/cpp/utility/functional/function). This standard library class is templated on the function's signature - that means we have to provide it a function type as a template parameter to specify the signature of the function (compare this to *Java* above, where we have a separate interface type for each kind of signature).

This sounds a lot more complicated than it is to use in practice. Let's look at the call signature of `cmd::RunOnce` (which creates an `InstantCommand` that, when scheduled, runs the given function once and then terminates):

> **Note**
>
> The `requirements` parameter is explained in the *Command-based documentation*, and will not be discussed here.

```
CommandPtr RunOnce(
 std::function<void()> action,
 Requirements requirements);
```

`runOnce` expects us to give it a `std::function<void()>` parameter (named `action`). A `std::function<void()>` is the C++ type for a `std::function` that takes no parameters and returns no value (the template parameter, `void()`, is a function type with no parameters and no return value). When we call `runOnce`, we need to give it a function with no parameters and no return value. C++ lacks a clean way to refer to existing class methods in a way that can automatically be converted to a `std::function`, so the typical way to do this is to define a new function inline with a "lambda expression".

Lambda Expressions in C++

To pass a function to `runOnce`, we need to write a short inline function expression using a special syntax that resembles ordinary C++ function declarations, but varies in a few important ways:

```
// Create an InstantCommand that runs the drive forward at half speed
CommandPtr driveHalfSpeed = cmd::RunOnce([this] { drivetrain.ArcadeDrive(0.5, 0.0); },
↪ {drivetrain});
```

C++ calls `[captures] (params) { body; }` a "lambda expression". It has three parts: a *capture list* (square brackets), an optional *parameter list* (parentheses), and a *function body* (curly brackets). It may look a little strange, but the only real difference between a lambda expression and an ordinary function (apart from the lack of a function name) is the addition of the capture list.

Since `RunOnce` wants a function with no parameters and no return value, our lambda expression has no parameter list and no return statement. The "lambda expression" here represents a function that calls `drivetrain.ArcadeDrive` with a specific set of parameters - note again that the above code does not *call* the function, but merely defines it and passes it to the `Command` to be run later when the `Command` is scheduled.

Capturing State in C++ Lambda Expressions

In the above example, our function body references an object that lives outside of the function itself (namely, the `drivetrain` object). This is called a "capture" of a variable from the surrounding code (which is sometimes called the "outer scope" or "enclosing scope"). Usually the captured variables are either local variables from the enclosing method body in which the lambda expression is defined, or else fields of an enclosing class definition in which that method is defined.

C++ has somewhat more-powerful semantics than Java. One cost of this is that we generally need to give the C++ compiler some help to figure out *how exactly* we want it to capture state from the enclosing scope. This is the purpose of the *capture list*. For the purposes of using the WPILibC Command-based framework, it is usually sufficient to use a capture list of `[this]`, which gives access to members of the enclosing class by capturing the enclosing class's `this` pointer by value.

Method locals cannot be captured with the `this` pointer, and must be captured explicitly either by reference or by value by including them in the capture list (or by implicitly by instead specifying a default capture semantics). It is typically safer to capture locals by-value, since a lambda can outlive the lifespan of an object it captures by reference. For more details, consult the C++ standard library documentation on capture semantics.

17.11 Get Alliance Color

The `DriverStation` class (Java, C++, Python) has many useful features for getting data from the Driver Station computer. One of the most important features is `getAlliance` (Java & Python) / `GetAlliance` (C++).

Note that there are three cases: red, blue, and no color yet. It is important that code handles the third case correctly because the alliance color will not be available until the Driver Station connects. In particular, code should not assume that the alliance color will be available during constructor methods or *robotInit*, but it should be available by the time *autoInit* or *teleopInit* is called. FMS will set the alliance color automatically; when not connected to FMS, the alliance color can be set from the Driver Station (see *"Team Station" on the Operation Tab*).

17.11.1 Getting your Alliance Color and Doing an Action

JAVA

```
Optional<Alliance> ally = DriverStation.getAlliance();
if (ally.isPresent()) {
    if (ally.get() == Alliance.Red) {
        <RED ACTION>
    }
    if (ally.get() == Alliance.Blue) {
        <BLUE ACTION>
    }
}
else {
    <NO COLOR YET ACTION>
}
```

C++

```
using frc::DriverStation::Alliance;
if (auto ally = frc::DriverStation::GetAlliance()) {
    if (ally.value() == Alliance::kRed) {
        <RED ACTION>
    }
    if (ally.value() == Alliance::kBlue) {
        <BLUE ACTION>
    }
}
else {
    <NO COLOR YET ACTION>
}
```

PYTHON

```
from wpilib import DriverStation
ally = DriverStation.getAlliance()
if ally is not None:
    if ally == DriverStation.Alliance.kRed:
        <RED ACTION>
    elif ally == DriverStation.Alliance.kBlue:
        <BLUE ACTION>
else:
    <NO COLOR YET ACTION>
```

17.12 Java Garbage Collection

Java garbage collection is the process of automatically managing memory for Java objects. The Java Virtual Machine (JVM) is responsible for creating and destroying objects, and the garbage collector is responsible for identifying and reclaiming unused objects.

Java garbage collection is an automatic process, which means that the programmer does not need to explicitly deallocate memory. The garbage collector keeps track of which objects are in use and which are not, and it periodically reclaims unused objects.

17.12.1 Object Creation

Creating a large number of objects in Java can lead to memory and performance issues. While the Java Garbage Collector (GC) is designed to handle memory management efficiently, creating too many objects can overwhelm the GC and cause performance degradation.

Memory Concerns

When a large number of objects are created, it increases the overall memory footprint of the application. While the overhead for a single object may be insignificant, it can become substantial when multiplied by a large number of objects.

> **Note**
>
> *VisualVM* can be used to see where memory is allocated.

Performance Concerns

The GC's job is to periodically identify and reclaim unused objects in memory. While garbage collection is running on an FRC robot coded in Java, execution of the robot program is paused. When the GC has to collect a large number of objects, it has to pause the application to run more frequently or for longer periods of time. This is because the GC has to perform more work to collect and process each object.

GC-related performance degradation in robot programs can manifest as occasional pauses, freezes, or loop overruns as the GC works to reclaim memory.

Design Considerations

If you anticipate your application creating a large number of short-lived objects, it is important to consider design strategies to mitigate the potential memory and performance issues. Here are some strategies to consider:

- Minimize object creation: Carefully evaluate the need for each object creation. If possible, reuse existing objects or use alternative data structures, such as arrays or primitives, to avoid creating new objects.
- Efficient data structures: Use data structures that are well-suited for the type of data you are working with. For example, if you are dealing with a large number of primitive values, consider using arrays or collections specifically designed for primitives.

17.12.2 Diagnosing Out of Memory Errors with Heap Dumps

All objects in Java are retained in a section of memory called the *heap*. As objects typically consume the greatest amount of memory in a Java program, it is often useful to take a snapshot of the state of the heap—a heap dump—to analyze memory issues. Heap dumps only capture the state of a program's heap at a single point in time, so they are unlikely to be useful if not captured exactly at the time the program is experiencing memory issues.

Since `OutOfMemoryErrors` both crash the program and are a common reason to want a heap dump, the JVM can be configured to automatically take a heap dump the moment an `OutOfMemoryError` is caught by the JVM. To configure these options, locate the `frcJava` code block in your project's `build.gradle`:

```
deploy {
    targets {
        roborio(getTargetTypeClass('RoboRIO')) {
            // Team number is loaded either from the .wpilib/wpilib_preferences.json
            // or from command line. If not found an exception will be thrown.
            // You can use getTeamOrDefault(team) instead of getTeamNumber if you
            // want to store a team number in this file.
            team = project.frc.getTeamNumber()
            debug = project.frc.getDebugOrDefault(false)

            artifacts {
                // First part is artifact name, 2nd is artifact type
                // getTargetTypeClass is a shortcut to get the class type using a string

                frcJava(getArtifactTypeClass('FRCJavaArtifact')) {
                }

                // Static files artifact
                frcStaticFileDeploy(getArtifactTypeClass('FileTreeArtifact')) {
                    files = project.fileTree('src/main/deploy')
                    directory = '/home/lvuser/deploy'
                    deleteOldFiles = false // Change to true to delete files on roboRIO that no
                                           // longer exist in deploy directory of this project
                }
            }
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Add to the code block so that it contains two `jvmArgs` commands, as shown below:

```
frcJava(getArtifactTypeClass('FRCJavaArtifact')) {
    // If you have other configuration here, you do not need to remove it.
    // Enable automatic heap dumps on OutOfMemoryError
    // Note: the heap dump path here is a path on a USB flash drive, see below
    jvmArgs.add("-XX:+HeapDumpOnOutOfMemoryError")
    jvmArgs.add("-XX:HeapDumpPath=/u/frc-usercode.hprof")
}
```

This will cause the JVM to write heap dumps to a file named `frc-usercode.hprof` at the root of a USB flash drive attached to the roboRIO when the code runs out of memory. It is recommended to save these heap dumps to a USB flash drive because heap dumps intrinsically consume the same amount of space on disk as the program heap did in memory when the program crashed, and are likely to be larger than the roboRIO's internal storage has capacity for. Once you have reproduced the `OutOfMemoryError`, redeploy your code without these options enabled, and use the USB flash drive to transfer the heap dump to a computer for analysis in a memory profiler such as *VisualVM*.

Warning

Configuring the JVM this way requires that the flash drive remain connected to the roboRIO while your code is running.

Larger SD cards may provide enough onboard storage to allow the use of these options on the roboRIO 2 without a USB flash drive. To do this, set the `-XX:HeapDumpPath` option to reference a path on the SD card, and use *FTP/SFTP to transfer the heap dump to a computer* before deleting it from the SD card.

Note that the JVM will **not** overwrite heap dumps with the exact path and filename specified by `-XX:HeapDumpPath` if they already exist, nor will it dump the process heap to a file with a different name. If a path to a directory is supplied instead of a path to a file, the JVM will instead write out heap dumps with unique filenames within the specified directory, with the name `java_pidNNNN.hprof`, where `NNNN` is the process ID of the JVM that ran out of memory. Note that this can cause large files to build up on disk if they are not cleaned out, so if you configure the JVM this way, be sure to frequently copy heap dumps to a computer and delete them from the flash drive/SD card afterward.

Caution

Always be vigilant about the amount of available space on the underlying storage medium while you use this feature.

Use of this feature is not recommended during competitive play.

17.12.3 System Memory Tuning

If the JVM cannot allocate memory, the program will be terminated. As an embedded system with only a small amount of memory available (256 MB on the roboRIO 1, 512 MB on the roboRIO 2), the roboRIO is particularly susceptible to running out of memory.

No amount of system tuning can fix out of memory errors caused by out-of-control allocation

If you are running out of memory, always investigate allocations with heap dumps and/or *VisualVM* first.

If you continue to run out of memory even after investigating with VisualVM and taking steps to minimize the number of allocated objects, a few different options are available to make additional memory available to the robot program.

- Disabling the system web server
- Setting sysctls (Linux kernel options)
- Periodically calling the garbage collector
- Setting up swap on a USB flash drive

Implementing most of these options require *connecting with SSH* to the roboRIO and running commands. If run incorrectly, it may require a reimage to recover, so be careful when following the instructions.

Disabling the System Web Server

The built-in NI system web server provides the webpage (the *roboRIO Web Dashboard*) seen when using a web browser to connect to the roboRIO, e.g. to change IP address settings. It also is used by the Driver Station's data log download functionality. However, it consumes several MB of RAM, so disabling it will free up that memory for the robot program to use. There are several ways to disable the web server:

The first and easiest is to use the *RoboRIO Team Number Setter* tool. Versions 2024.2.1 and later of the tool have a button to disable or enable the web server. However, a few teams have reported that this does not work or does not persist between reboots. There are two alternate ways to disable the web server; both require connecting to the roboRIO with SSH and logging in as the `admin` user.

1. Run `/etc/init.d/systemWebServer stop; update-rc.d -f systemWebServer remove; sync`
2. Run `chmod a-x /usr/local/natinst/etc/init.d/systemWebServer; sync`

To revert the alternate ways and re-enable the web server, take the corresponding step:

1. Run `update-rc.d -f systemWebServer defaults; /etc/init.d/systemWebServer start; sync`
2. Run `chmod a+x /usr/local/natinst/etc/init.d/systemWebServer; sync`

Setting sysctls

Several Linux kernel options (called sysctls) can be set to tweak how the kernel allocates memory. Several options have been found to reduce out-of-memory errors:

- Setting `vm.overcommit_memory` to 1 (the default value is 2). This causes the kernel to always pretend there is enough memory for a requested memory allocation at the time of allocation; the default setting always checks to see if there's actually enough memory to back an allocation at the time of allocation, not when the memory is actually used.
- Setting `vm.vfs_cache_pressure` to 1000 (the default value is 100). Increasing this causes the kernel to much more aggressively reclaim file system object caches; it may slightly degrade performance.
- Setting `vm.swappiness` to 100 (the default value is 60). This causes the kernel to more aggressively swap process memory to the swap file. Changing this option has no effect unless you add a swap file.

You can set some or all of these options; the most important one is `vm.overcommit_memory`. Setting these options requires connecting to the roboRIO with SSH and logging in as the `admin` user, then running the following commands:

```
echo "vm.overcommit_memory=1" >> /etc/sysctl.conf
echo "vm.vfs_cache_pressure=1000" >> /etc/sysctl.conf
echo "vm.swappiness=100" >> /etc/sysctl.conf
sync
```

The `/etc/sysctl.conf` file should contain the following lines at the end when done (to check, you can run the command `cat /etc/sysctl.conf`):

```
vm.overcommit_memory=1
vm.vfs_cache_pressure=1000
vm.swappiness=100
```

To revert the change, edit `/etc/sysctl.conf` (this will require the use of the vi editor) and remove these 3 lines.

Periodically Calling the Garbage Collector

Sometimes the garbage collector won't run frequently enough to keep up with the quantity of allocations. As Java provides a way to trigger a garbage collection to occur, running it on a periodic basis may reduce peak memory usage. This can be done by adding a `Timer` and a periodic check:

```
Timer m_gcTimer = new Timer();
public Robot() {
  m_gcTimer.start();
}
public void periodic() {
  // run the garbage collector every 5 seconds
  if (m_gcTimer.advanceIfElapsed(5)) {
    System.gc();
  }
}
```

Setting Up Swap on a USB Flash Drive

A swap file on a Linux system provides disk-backed space that can be used by the system as additional virtual memory to put infrequently used data and programs when they aren't being used, freeing up physical RAM for active use such as the robot program. It is strongly recommended to not use the built-in non-replaceable flash storage on the roboRIO 1 for a swap file, as it has very limited write cycles and may wear out quickly. Instead, however, a FAT32-formatted USB flash drive may be used for this purpose. This does require the USB flash drive to always be plugged into the roboRIO before boot.

Caution

Having a swap file on a USB stick means it's critical the USB stick stay connected to the roboRIO at all times it is powered.

This should be used as a last resort if none of the other steps above help. Generally needing swap is indicative of some other allocation issue, so use VisualVM first to optimize allocations.

A swap file can be set up by plugging the USB flash drive into the roboRIO USB port, connecting to the roboRIO with SSH and logging in as the `admin` user, and running the following commands. Note the vi step requires knowledge of how to edit and save a file in vi.

```
fallocate -l 100M /u/swapfile
mkswap /u/swapfile
swapon /u/swapfile
vi /etc/init.d/addswap.sh
chmod a+x /etc/init.d/addswap.sh
update-rc.d -v addswap.sh defaults
sync
```

The `/etc/init.d/addswap.sh` file contents should look like this:

```
#!/bin/sh
[ -x /sbin/swapon ] && swapon -e /u/swapfile
: exit 0
```

To revert the change, run `update-rc.d -f addswap.sh remove; rm /etc/init.d/addswap.sh; sync; reboot`.

18 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.

18.1 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 Software & Resources Page
- REV Robotics Documentation
- Vivid Hosting

18.2 Forums

Stuck? Have a question not answered by the documentation? Official Support is provided on these forums:

- NI FRC Support Forum (roboRIO, LabVIEW, and Driver Station software questions, as well as roboRIO repairs)
- FIRST Inspires Control System Forum (wiring, hardware, and Driver Station questions)
- FIRST Inspires Programming Forum (programming questions for C++, Java, or LabVIEW)

18.3 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.

18.4 REV Robotics Support

Support for REV Robotics components (SPARK MAX, Sensors, Pneumatic Hub, Power Distribution Hub, Radio Power Module) is provided via phone at 844-255-2267 or via the email address support@revrobotics.com.

18.5 WCP Support

Support for WestCoast Products components (VH-109 radio) is provided via phone at 559-206-2920 or via the email address support@wcproducts.com.

18.6 Other Vendors

Support for vendors outside of the KOP can be found below.

- Copperforge
- Kauai Labs (NavX)
- Limelight
- PhotonVision (Discord)
- Playing with Fusion
- Redux Robotics
- The Thrifty Bot

18.7 Unofficial Support

There are useful forms of support provided by the community through various forums and services. The below links and websites are not endorsed by FIRST® and may be used at your own risk.

- Chief Delphi
- FRC Discord

18.8 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

19 FRC Glossary

accelerometer
A common sensor used to measure acceleration in one or more axis.

AM
AndyMark, Inc - strives to develop innovative products and outstanding service while inspiring our customers and making a positive impact in our community.

AprilTags
Visual tags that provide low overhead, high accuracy localization. AprilTags are useful for helping your robot know where it is at on the field, so it can align itself to some goal position.

auto
The first phase of each match is called Autonomous (auto) and consists of the robot's running pre-programmed instructions.

back-EMF
In electric motors, the force generated by the interaction of spinning magnets in a coil of wire which opposes spinning motion.

boolean
A form of data with only two possible values (true or false), intended to represent the two truth values of logic and Boolean algebra.

call stack
A specially-organized region of memory which helps the program keep track of what function it is in. As each function calls another, the call point is recorded and added to the top of the structure, forming a "stack" of references. Additionally, local variables will also be stored in this stack. See call stack on Wikipedia for more info.

CAD
Computer-Aided Design - software used to design an accurate model of an object. For FRC this is often used to design the robot to get accurate measurements and aid construction.

CAM
Computer-Aided Manufacturing - the use of software to control machine tools in the manufacturing of work pieces.

CAN
Controller Area Network - message-based protocol designed to allow microcontrollers

and devices to communicate with each other's applications without a host computer.

CD
Chief Delphi - FRC team 47 inspired a popular community driven forum that today serves as an unofficial discussion hub for all things FRC.

central limit theorem
A core concept in probability which states that when many independent variables are added up, the result tends to look like a "normal" (or Gaussian) distribution, regardless of whether the independent variables themselves are normally distributed. See Central Limit Theorem on Wikipedia for more info.

CIM
CCL Industrial Motor, Limited - Chiaphua Components Limited is the company that made the commonly used, relatively powerful, brushed motor.

Classical Mechanics
The branch of physics which studies and describes the motion of relatively large, relatively slow objects. See Classical Mechanics on Wikipedia for more info.

COTS
Commercial off the shelf - a standard (i.e. not custom order) part commonly available from a vendor to all teams for purchase.

composition
A formal software term for building (or "composing") software entities out of smaller component entities. See object composition on Wikipedia for more info.

CRTP
Continuously Recurring Template Pattern - A software idiom in which a class $X`$ derives from a class template instantiation using $X`$ itself as a template argument. See CRTP on Wikipedia for more info.

CSA
Control Systems Advisor - FRC volunteer position that assists teams with Robot Control System-related issues.

CTRE
Cross the Road Electronics LLC - is an engineering design, software development, and electronics manufacturer based outside of Detroit in Macomb, MI. They primarily focus on high-performing, high-quality electronics communication, motor control, and control system products for FIRST teams and the EV industry. CTR Electronics was founded in 2006 by two FRC mentors who met through their robotics team: Mike Copioli and Omar Zrien and is staffed largely by FRC alumni, and active volunteers & mentors.

C++
One of the four officially supported programming languages.

declarative programming
A style of software which focuses on describing *what* a program should do, rather than *how* it gets done. See declarative programming on Wikipedia for more info.

dependency injection
A software design pattern where each class receives all objects it depends upon. Sometimes these are passed through the constructor, but not always. See dependency injection on Wikipedia for more info.

deprecated
Software that has been replaced and will no longer receive new features. Deprecated software will be maintained for at least 1 year, but may be removed after that. For

example, if a method is deprecated prior to the 2022 season, it will be usable in the 2022 season, but may be removed prior to the 2023 season. Teams are encouraged to not use deprecated methods in new code. WPILib always deprecates features at least one year prior to removing them from the codebase.

design pattern
A particular, intentionally-chosen style of organizing code. A design pattern intentionally excludes using certain features of a programming language to constrain developers into solutions that are well-suited to a particular problem-space. See design pattern. on Wikipedia for more info.

DHCP
Dynamic Host Configuration Protocol - the protocol that allows a central device to assign unique IP addresses to all other devices.

encapsulation
A software design pattern which uses a class to hide the implementation details of other classes. See encapsulation on Wikipedia for more info.

entry
In *NetworkTables*, a combined *publisher* and *subscriber*. The subscriber is always active, but the publisher is not created until a publish operation is performed (e.g. a value is "set", aka published, on the entry). This may be more convenient than maintaining a separate publisher and subscriber.

enumeration
A list of all elements of a set, typically used to refer to a set of pre-defined values.

EPA
Expected Points Added - builds upon the Elo rating system, but transforms ratings to point units and makes several modifications.

event-driven programming
A style of programming where certain parts of code generate "events" as a result of some input (sensors, user interaction, etc). Then, other parts of code listen for and respond to "handle" these events. See event-based on Wikipedia for more info.

FIRST
For Inspiration and Recognition of Science and Technology - a global nonprofit organization that prepares young people for the future through a suite of life-changing youth robotics programs that build skills, confidence, and resilience.

FLL
FIRST Lego League - Introduces science, technology, engineering, and math (STEM) to children ages 4-16 through fun, exciting hands-on learning.

floating point
A method for approximating real numbers in computer-based arithmetic, using a fixed precision integer scaled by an integer exponent. Typically computer systems support both "single" precision (32-bit storage) and "double" precision (64-bit storage) floating point values, as defined by IEEE 754.

FMS
Field Management System - the electronics core responsible for sensing and controlling the FIRST Robotics Competition field.

FPGA
Field-programmable gate array - a specialized integrated circuit consisting of many digital logic elements, which can be configured to act in different patterns. This allows its

behavior to be changed after manufacturing. In the context of FRC, National Instruments provides a specific configuration for the RIO's FPGA which allows it to process the electrical inputs and outputs at a very high rate. See FPGA on Wikipedia for more info.

FRC
FIRST Robotics Competition - Combining the excitement of sport with the rigors of science and technology. The ultimate Sport for the Mind inspiring High-school students.

FTA
FIRST Technical Advisor - FRC volunteer position that is responsible for ensuring FIRST Robotics Competition events run smoothly, safely, and in accordance with FIRST requirements, and ensuring a high-quality experience for all event participants and teams.

FTC
FIRST Tech Challenge - Grades 7-12 are challenged to design, build, program, and operate robots to compete in a head-to-head challenge in an alliance format.

GDC
Game Design Committee - designs the game for each year and develops the rules and field setup for each competition.

GP
Gracious Professionalism - part of the ethos of FIRST. It's a way of doing things that encourages high-quality work, emphasizes the value of others, and respects individuals and the community.

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.

imperative programming
A style of programming that focuses on *what* the code should be doing, step by step, every loop. See imperative programming on Wikipedia for more info.

IMU
Inertial Measurement Unit - a sensor that combines both an *accelerometer* and a *gyroscope* into a single sensor.

I2C
Inter-Integrated Circuit - a synchronous, multi-master/multi-slave (controller/target), single-ended, serial communication bus.

Java
One of the four officially supported programming languages.

JSON
JavaScript Object Notation - A standardized way of organizing data into named values. The organized data can be easily *serialized*. While the original usage was in Javascript, it can be used and interested by most modern programming languages. See JSON on Wikipedia for more info.

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 KOP contains a drive base (chassis) distributed to every team (that did not opt out) as part of the *KOP*. For the 2026 season, the KOP chassis is the AM14U6.

LabVIEW
One of the four officially supported programming languages.

LED
Light-Emitting Diode - a semiconductor device that emits light when current flows through it. Used on multiple robot parts to convey the status of the device.

mass
the amount of matter in a physical object. Objects with more mass will resist changes in motion more than objects with less mass. See mass on Wikipedia for more info.

moment of inertia
The property of an object that describes both how much mass it has, and how that mass is distributed relative to a certain axis of rotation. Objects with higher moments of inertia resist changes in rotational motion more than objects with lower moments of inertia. Increasing the moment of inertia is accomplished by adding more mass, or moving the mass further away from the axis of rotation. See moment of inertia on Wikipedia for more info.

mutable
An object that can be modified after it is created.

MXP
myRIO Expansion Port - Port in the center of the roboRIO designed to expand the traditional IO count by offering multiple different IO types through one connector.

NetworkTables
A publish-subscribe messaging system to communicate data between programs.

no-op
No-op is a computer instruction which means no operation. When the computer processor encounters a no-op instruction, it simply moves to the next sequential instruction. Read more about no-op on Wikipedia .

odometry
Using sensors on the robot to create an estimate of the pose of the robot on the field.

OPR
Offensive Power Rating - a system to attempt to deduce the average point contribution of a team to an alliance

PCM
Pneumatic Control Module - provides an easy all-in-one interface for pneumatic components.

PDH
REV Power Distribution Hub - latest evolution in power distribution for FRC. With 20 high-current (40A max) channels, 3 low-current (15A max), and 1 switchable low-current channel, the PDH gives teams more flexibility for overall power delivery.

PDP
CTRE Power Distribution Panel - power distribution module with 8 high-current (40A max), 8 lower current (20A / 30A), 1 20A protected channel (for *PCM* and *VRM*), and 1 10A protected channel (for the roboRIO).

permanent-magnet DC motor
: The classification of all legal motors for the FIRST robotics competition. This type of motor takes direct current as input, and uses it to create a magnetic field. In turn, this magnetic field interacts with a physical magnet to create a force that turns the output shaft. Electrical ("brushless") or mechanical ("brushed") means are used to ensure the electrically-generated magnetic field always points in a direction that creates forces when it interacts with the physical magnet, even as the motor's shaft rotates. See permanent-magnet motor on Wikipedia for more info.

persistent
: In *NetworkTables*, a *topic* that is saved to a file by the server and restored at startup.

PH
: Pneumatic Hub - is a standalone module that is capable of switching both 12V and 24V pneumatic solenoid valves. The Pneumatic Hub features 16 solenoid channels which allow for up to 16 single-acting solenoids, 8 double-acting solenoids, or a combination of the two types.

PoE
: Power Over Ethernet - method of powering a device by an Ethernet cord that also carries power. FRC uses passive PoE usually 12-24V that is always being supplied, this can damage a device not expecting the provided voltage. The most common industry standard is active PoE which uses 48V but first verifies that the device is expecting the power.

property
: In *NetworkTables*, named information (metadata) about a *topic* stored and updated separately from the topic's data. A topic may have any number of properties. A property's value can be any data type that can be represented in JSON.

publisher
: In *NetworkTables*, an object that defines a *topic* and creates and sends timestamped data values.

pose
: The collection of position and rotation information that describes how a rigid body is oriented in space, relative to some fixed reference point.

pose estimation
: The process of estimating the robot's pose, commonly with *odometry* and/or *AprilTags*. Also known as *on-field localization*.

PWM
: Pulse-width modulation - a method of controlling the average power or amplitude delivered by an electrical signal. Used in FRC to control the output of motors not using the *CAN* bus.

Python
: One of the four officially supported programming languages.

RAII
: Resource Acquisition Is Initialization - a language behavior (in C++, but not in Java) where holding a resource is tied to object lifetime.

retro-reflection
: The property of reflecting incoming light back at the same angle it came in at, rather than an incident angle (like a mirror), absorbing it, or scattering it. Most FRC vision processing targets are retro-reflective. See retroreflector on Wikipedia for more information.

recursive composition
A type of *composition* in which the composite object may contain components of the same type as itself. For example, a command group may contain one or more command groups. See recursive composition on Wikipedia for more info. See also *recursive composition*.

retained
In *NetworkTables*, a *topic* that is kept alive by the server even after all publishers stop publishing.

REV
REV Robotics - inspires innovation and creativity within the educational robotics community by offering comprehensive product lines, extensive educational resources, world-class customer service, and specialized sponsorship programs. With a global presence spanning over 190 countries, we empower the next generation of STEM professionals by providing cutting-edge solutions and essential tools for success. Founded in 2014 by robotics enthusiasts Greg Needel and David Yanoshak, REV Robotics is driven by the mission to inspire and support students as they explore the exciting world of robotics and unlock their full robotic design potential. A majority of our employees are FIRST Alumni who remain actively involved, serving as volunteers and mentors for the local FIRST Community. This deep engagement reflects our commitment to supporting and inspiring the next generation of STEM enthusiasts.

RPM
Radio Power Module - is designed to keep one of the most critical system components, the OpenMesh OM5P-AC WiFi radio, powered in the toughest moments of the competition. Revolutions Per Minute - a unit of rotational speed often used when describing motors.

RSL
Robot Signal Light - safety light on every FRC robot used to indicate its operational status.

serialized
The property of a data organization scheme that allows the description of the data to be sent in order, byte by byte, over some communication channel. Reading or writing a file on disk is done in this serial fashion (IE, the data is read or written byte by byte, not all at once). Sending data over a *SPI* or *I2C* bus is also done byte by byte, again requiring the data can be serialized.

simulation
A way for teams to test their code without having an actual robot available.

software library
A collection of code that can be imported into and used by other software. See software library on Wikipedia for more info.

solenoid valve
A airflow-controlling valve which is actuated by a small electromagnet. Strictly speaking, the *solenoid* is the coil of wire which forms the electromagnet, and the *valve* is the mechanism which actually redirects airflow. However, the set of solenoid and valve together is often simply called “a solenoid”. See solenoid valve. on Wikipedia for more info.

SPI
Serial Peripheral Interface - protocol for synchronous serial communication, used primarily in embedded systems for short-distance wired communication between integrated circuits.

state machine
A programming construct that divides a problem into many discrete, well-defined,

mutually-exclusive “states”, then defines how the problem is solved by moving between different states. See state machine on Wikipedia for more more info.

subscriber
In *NetworkTables*, an object that receives timestamped data value updates to one or more *topic*s.

TBA
The Blue Alliance - Website for looking up *FRC* team statistics and event information.

telemetry
The process of recording and sending real-time data about the performance of your robot to a real-time readout or log file. For the linguists among us, the word’s roots are “tele” (remote) and “metry” (measurement). See telemetry on Wikipedia for more info.

teleop
The second phase of each match is called the Teleoperated Period (teleop) and consists of drivers controlling their robots.

topic
In *NetworkTables*, a named data channel.

torque
A force applied at a distance from some axis of rotation

trajectory
A trajectory is a smooth curve, with velocities, and accelerations at each point along the curve, connecting two endpoints on the field.

transitory
In *NetworkTables*, a *topic* that will disappear after the last *publisher* stops publishing.

VRM
Voltage Regulator Module - provides access to different constant voltages for custom sensors, cameras, or other unique applications. 12V DC Input Directly fed power from the Power Distribution Panel Designed to work with the roboRIO FRC control system.

WCP
WestCoast Products & Design LLC - was founded in Fall of 2011 by Ranjit Chahal (R.C.) and Harvey Rico. WCP aims to provide FIRST Teams, Hobbyists, and educators top notch quality products and designs for their projects.

WFA
Woodie Flowers Award - This award recognizes an individual who has done an outstanding job of motivation through communication while also challenging the students to be clear and succinct in their communications.

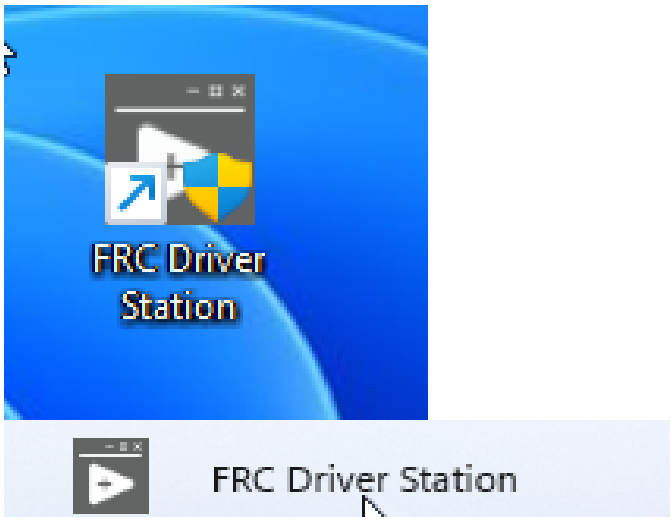
20 Driver Station

20.1 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*.

20.1.1 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 Apps->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*. WPILib must be *installed* to use SmartDashboard and Shuffleboard.

20.1.2 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.
- `Backspace` - "A-Stop" the robot when in Practice Mode - Autonomous Enabled. The robot will be disabled until the Practice Mode reaches Teleop then will be automatically re-enabled. While disabled, the application background will flash orange indicating the robot will re-enable automatically.

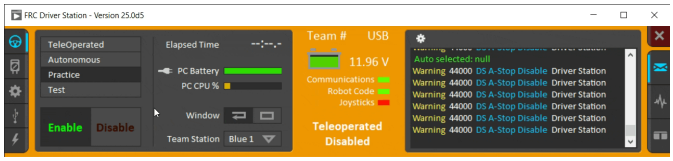


Fig. 1: View this page in the web documentation for an animation of the A-Stop behavior

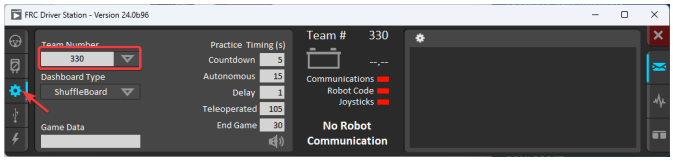
Note

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.

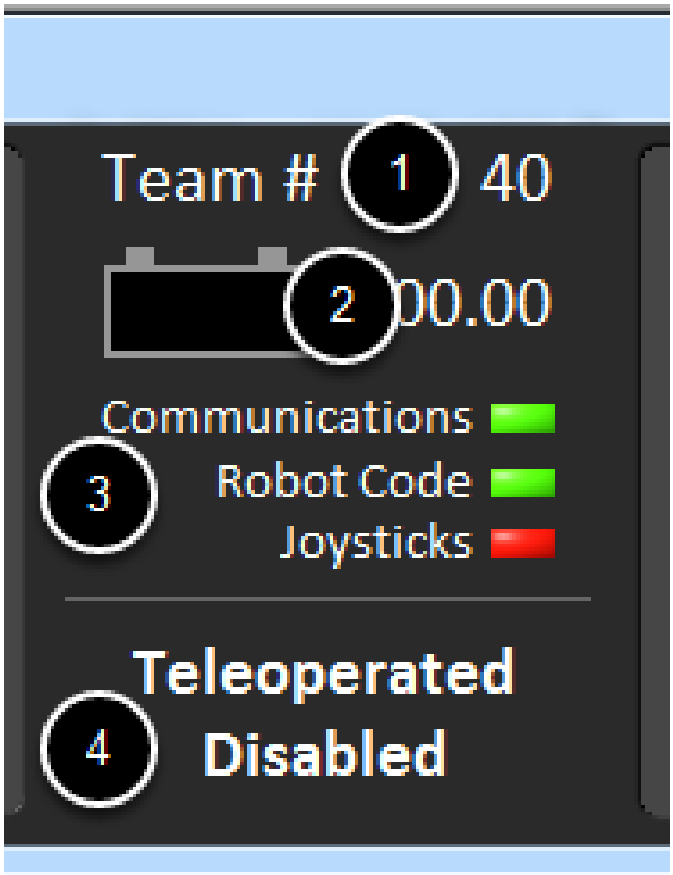
20.1.3 Setting Up the Driver Station



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*.

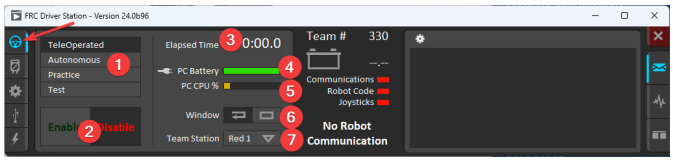
20.1.4 Status Pane



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".

20.1.5 Operation Tab



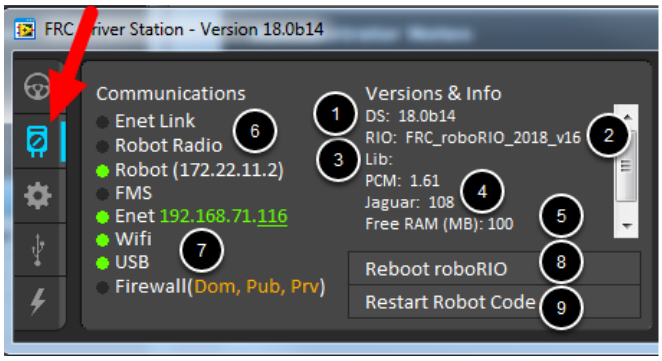
The Operations Tab is used to control the mode of the robot and provide additional key status indicators while the robot is running.

1. Robot Mode - This section controls the Robot Mode.
   - Teleoperated Mode causes the robot to run the code in the Teleoperated portion of the match.
   - Autonomous Mode causes the robot to run the code in the Autonomous portion of the match.
   - 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). When Practice Mode is in use, the DS will flash the background orange to indicate a pending enable (either the start of Autonomous or the start of Teleop after an A-Stop).
   - *Test Mode* is an additional mode where test code that doesn't run in a regular match can be tested.
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.

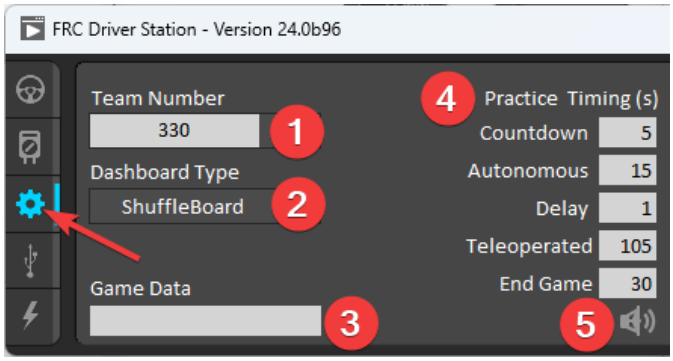
20.1.6 Diagnostics Tab



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 even 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).

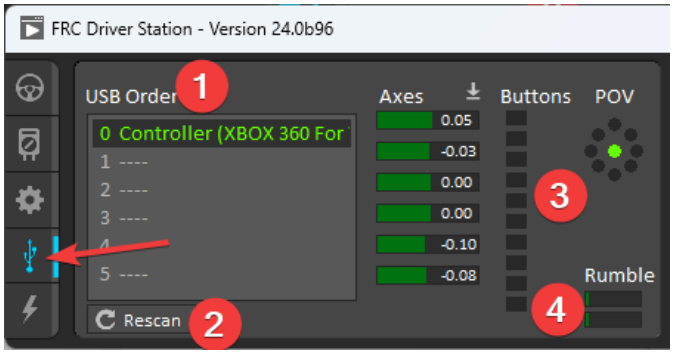
20.1.7 Setup Tab



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* forwards LabVIEW dashboard data to the IP specified in *Dashboard IP* field.
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.

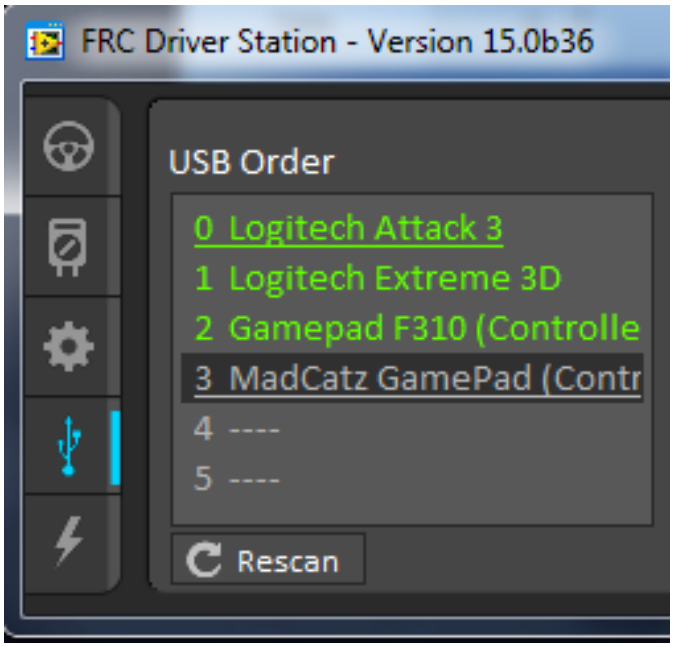
20.1.8 USB Devices Tab



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

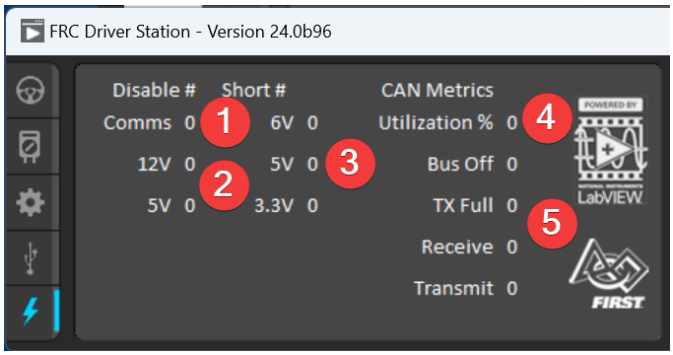
switch the ports of two identical devices the lock should follow the port, not the device. If you re-arrange the ports (take one device and plug it into a new port instead of swapping) the behavior is not determinate (the devices may swap slots). If you unplug one or more of the set of devices, the positions 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.

20.1.9 CAN/Power Tab

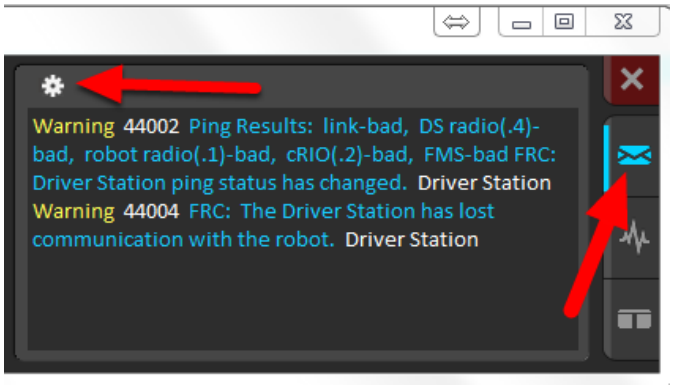


The last tab on the left side of the DS is the CAN/Robot Power Tab. This tab contains information about the power status of the roboRIO and the status of the CAN bus:

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 caused 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.

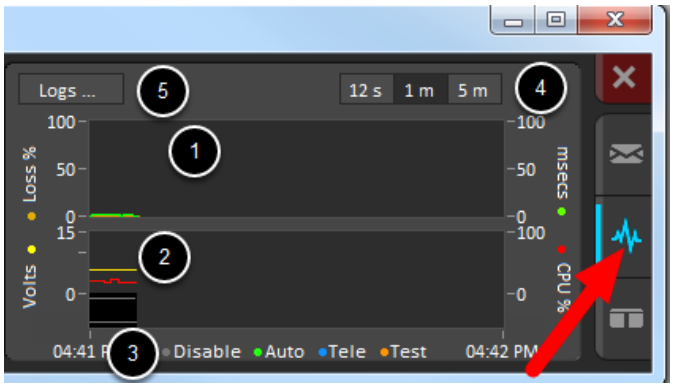
20.1.10 Messages Tab



The Messages tab displays diagnostic messages from the DS, WPILib, User Code, and/or the roboRIO.

To access settings for the Messages tab, click the Gear icon. This will display a menu that will allow you to clear the box, launch a larger Console window for viewing messages, launch the DS Log Viewer, launch a viewer for program timing, or download log files from the robot.

20.1.11 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.

20.1.12 Both Tab

The last tab on the right side is the Both tab which displays Messages and Charts side by side.

20.2 Driver Station (Operator Console) Best Practices

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 and peripherals is an easy project. Read on to find lessons learned by many teams over thousands of matches.

20.2.1 Selecting hardware

Dedicate a laptop to be used solely for driving. 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 Driver Station application and associated Dashboard installed or running.

Use a business-class laptop for your driver station laptop. 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.

Consider used laptops rather than new. The FRC Driver Station application 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. There are thousands for sale every day on eBay. You might even get one donated by a used computer store.

Laptop recommended features

1. 8GB of RAM or greater
2. A display size of 14" or greater, with 1080p minimum resolution
3. Ports
   a. 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.
   b. If the Ethernet port on your laptop is dodgy, either replace the laptop (recommended) or buy a USB Ethernet dongle from a reputable brand. See *Recommended USB Ethernet adapters* for a list. 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.
   c. 2 USB ports minimum. These can be USB-A or USB-C ports, depending on the peripherals you're using.
4. A keyboard. It's hard to quickly do troubleshooting on touch-only computers at the field.

5. A solid-state disk (SSD), 256 GB or larger. If the laptop has a rotating disk, spend $50 and replace it with an SSD.
6. Supports Windows 11. Given that it is at end of life, don't plan on support for Windows 10 beyond the 2026 competition year.
7. AMD or Intel CPU. The FRC Driver Station Application does not work on machines with ARM CPUs (e.g., Snapdragon).
8. A 2.4 and 5 GHz Wi-Fi radio. 6GHz support is optional but not necessary. For the 2026 game, the preferred practice configuration does not require 6GHz, and the 2027 SystemCore has built-in 5 GHz connectivity.

20.2.2 Building your driver station



Fig. 2: Two level driver station with power and USB hub in lower level. USB-C ethernet adapter provides field connectivity.

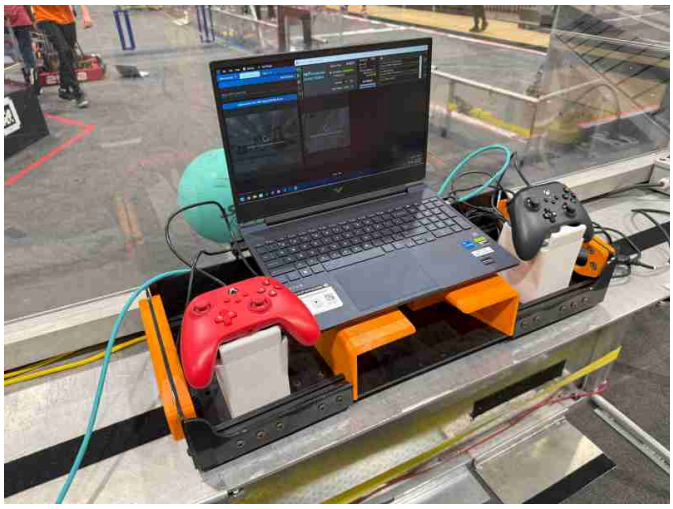


Fig. 3: Driver station with integrated controller holders. Controllers and field network are directly connected to laptop.

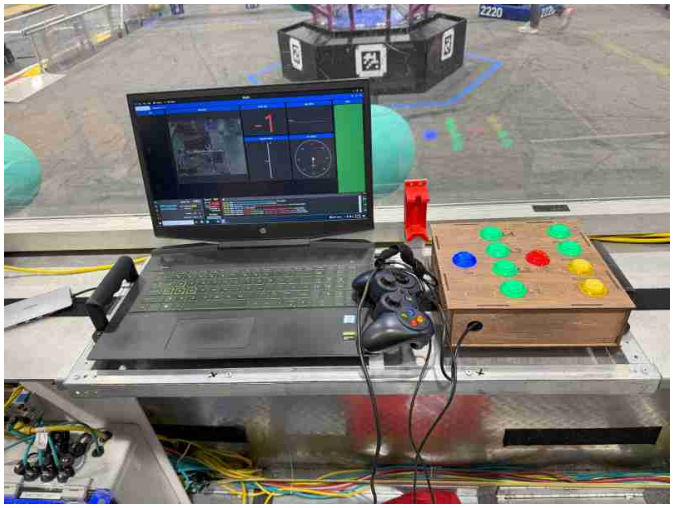


Fig. 4: Single level driver station with external button controls for task automation. USB-C ethernet/hub combo provides field connectivity.

It's best if you combine your driver station laptop and all needed peripherals into a single unit.

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.

Robots often hit the driver station wall with large forces, which can push your driver station off the shelf. Put hook tape on the bottom of the driver station to reduce the chance of that happening. The Game Manual has details on the correct location of the tape. Shorter driver stations have lower center of gravity and are more likely to stay on the shelf.

When fastening down peripherals, it's best to use hook and loop tape. This allows quick changes at the field when needed.

Ensure that the USB ports on the laptop are accessible.

20.2.3 Setting up the laptop

Once you've acquired your laptop, here's a list of things to do.

1. If you're buying used, ensure you're starting from a fresh Windows install.
2. Create one account to use both for software installation and driving. This account must be a member of the Administrator group.
3. Remove any 3rd party antivirus or antimalware software. Instead, use Windows Defender. Since you're only connecting to the internet for Windows and FRC software updating, the risk is low.
4. Only install software on your driver station laptop that's needed for driving. Your goal here is to eliminate variables that might interfere with proper operation. Don't use the driver station laptop for programming. Remove any unneeded preinstalled software ("bloatware") that came with the machine. Don't use the laptop for gaming back at the hotel the night before the event.

5. Avoid managed Windows 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.
6. Make these power settings for both the Plugged in and On battery profiles, by navigating to Start -> Settings -> System -> Power and battery
   a. Set "Make my device sleep after" to Never for both Plugged in and On battery
   b. Turn off Energy Saver
   c. Set Power Mode to Best Performance
   d. 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.
7. If you are running a non-English version of Windows, set the user to English prior to doing any software installation. If your laptop has a non-English Single Language Edition, you should install an edition of Windows that is supported by the FRC Driver Station application and supports English.

20.2.4 Prior to leaving for the competition

Items in this section require internet connectivity and should be done prior to arriving at the event.

1. Install all Windows updates, then test the driver station with your robot. Keep running Windows Update until no updates show as pending.
2. Pause updates. Navigate to Start -> Settings -> Windows Update, then set Pause updates to cover the time of the competition.
3. Change "Active Hours" for Windows Updates to prevent updates from installing during competition hours. This helps if you forget to pause updates. Navigate to Start -> Settings -> Windows Update -> Advanced options, then select Active Hours. Be sure to take timezone differences at your competition location into account.
4. Bring a trusted USB and Ethernet cable for use connecting to the roboRIO.

20.2.5 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 using the Wi-Fi control in Quick Settings.
3. Charge the driver station laptop when it's in the pit.
4. 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.
5. LabVIEW users: 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 at the event to make 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.

20.2.6 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.
3. Bring your laptop charger to the field. Power is provided in each player station. Why? Windows is less aggressive about power management when the laptop is plugged in, which means the FRC Driver Station application behaves more predictably.
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 functioning.
   a. In the USB tab in the FRC Driver Station application, drag and drop to assign joysticks as needed.
   b. Verify that buttons are working and joysticks return to zero position.
   c. Use the rescan button (F1) if joysticks / controllers do not appear green.
   d. 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.
6. Ensure your *Dashboard is connected to the robot* after your driver station laptop connects to the robot.
7. Ensure FRC Driver Station application is fully visible and in focus during the match to avoid Windows depriortizing it, and to ease troubleshooting.

20.2.7 Recommended USB Ethernet adapters

Members of the FRC community report good results with these USB Ethernet adapters.

- Henrety USB C to Ethernet Adapter
- Siig SuperSpeed USB 3.0 LAN Hub
- TP-Link UE330
- uni USB to Ethernet Adapter with hub (both USB-A and USB-C styles)

Don't be confused and use a USB RJ-45 extender. These are reported to fry USB ports.

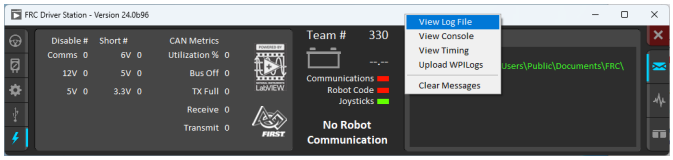


20.2.8 Credits

The source document for this page 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.

20.3 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, in the FRC Driver Station folder in Program Files, or via the Gear icon in the Driver Station.



Note

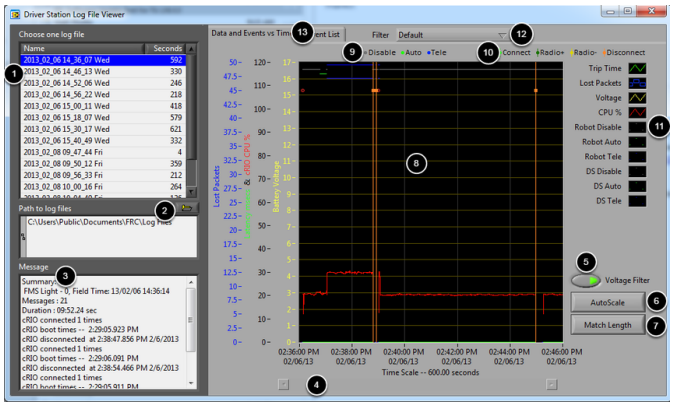
Several alternative tools exist that provide similar functionality to the FRC Driver Station Log Viewer. *AdvantageScope* is an option included in WPILib, and DSLOG Reader is a third-party option. Note that WPILib offers no support for third-party projects.

20.3.1 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 Driver Station Log files are overlaid in a single display.

Log files are stored in `C:\Users\Public\Documents\FRC\Log Files`. Each log has date and timestamp in the file name and has two files with extension `.dslog` and `.dsevents`.

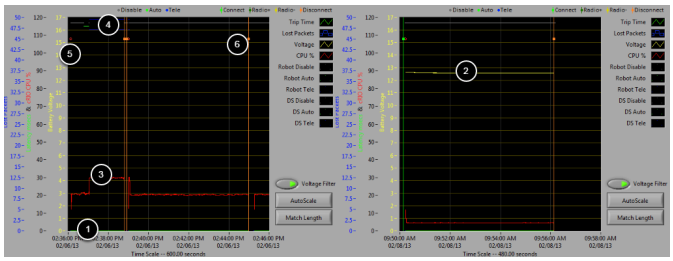
20.3.2 Log Viewer UI



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.

20.3.3 Using the Graph Display



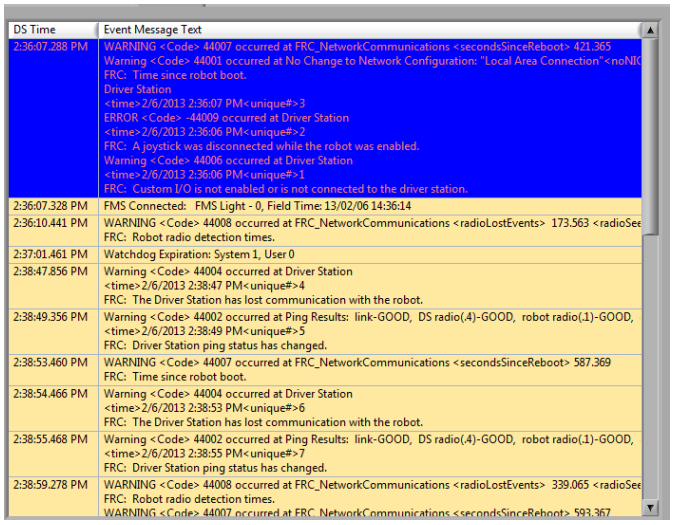
The Graph Display contains the following information:

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.

> **Note**
>
> The CTRE PDP 2.0 and AndyMark AMPD do not have buit in current logging and current logs will not be available in the graph display.

20.3.4 Event List



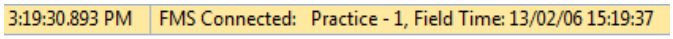
The Event List tab displays a list of events (warnings and errors) recorded by the Driver Station. The events and detail displayed are determined by the currently active filter (images shows “All Events, All Info” filter active).

20.3.5 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.

20.3.6 Identifying Logs from Matches



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.

20.3.7 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.

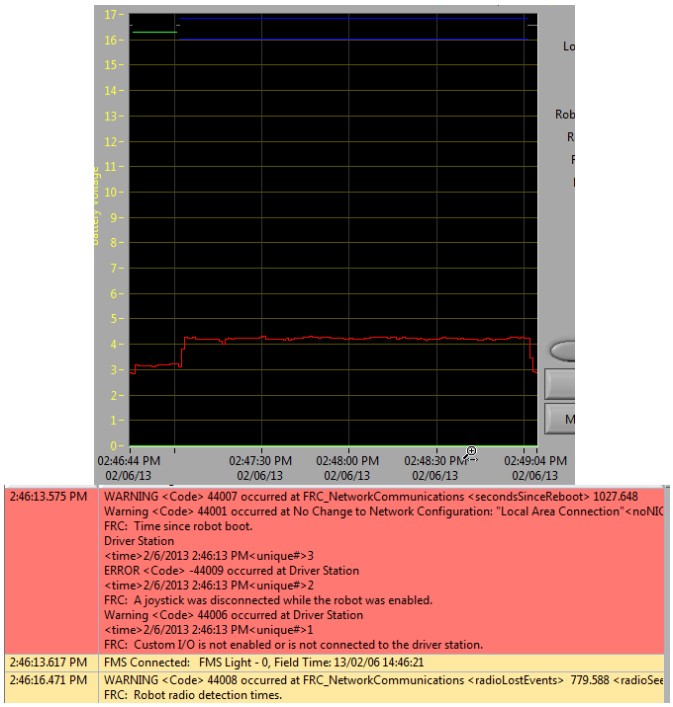
Note

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.

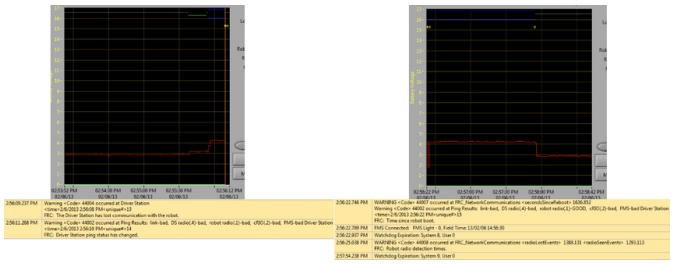
"Normal" Log



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.

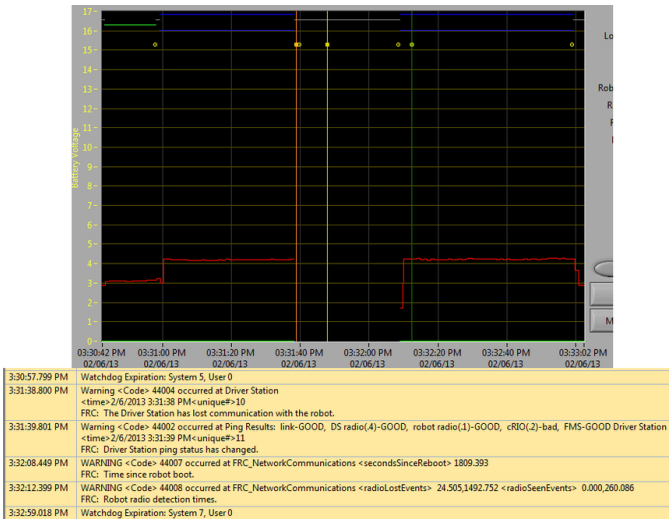
roboRIO Reboot



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

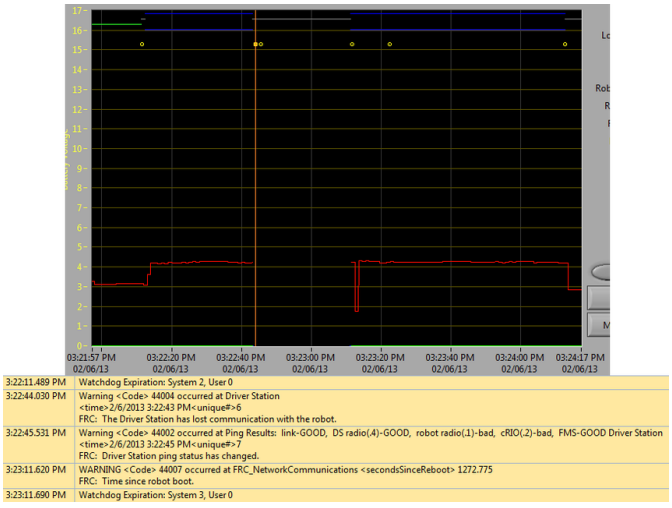
3:02:24 the code reports an error communicating with the camera. A warning is also reported indicating that no robot code is running right before the code finishes starting up.

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 “Tim 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 ~40-50 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".

20.4 Driver Station Errors/Warnings

In an effort to provide both Teams and Volunteers (*FTA* / *CSA* / 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).

20.4.1 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.

Note

Joystick Unplugged warnings can be silenced by calling `DriverStation.silenceJoystickConnectionWarning(true)` (Java, C++)

20.4.2 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.

20.4.3 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.

20.4.4 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.

20.4.5 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.

20.4.6 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.
```

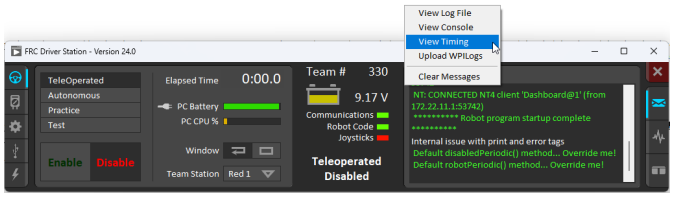
This message is printed when the DS begins communicating with the roboRIO, but detects no robot code running. A single instance of this message will be printed if the Driver Station is open and running while the roboRIO is booting as the DS will begin communication with the roboRIO before the robot code finishes loading.

20.5 Driver Station Timing Viewer

The 2024 Driver Station has a a new window to help diagnose robot control issues.

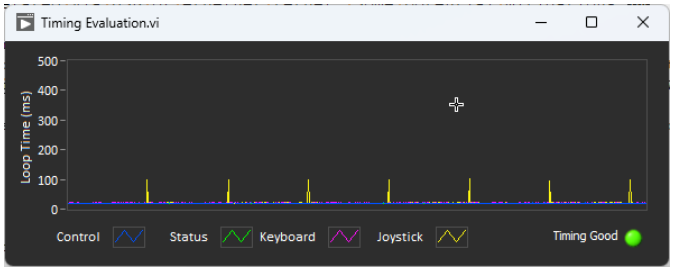
20.5.1 Opening the Timing Viewer

To start the Driver Station Timing Viewer, select the gear icon and then select *View Timing*.

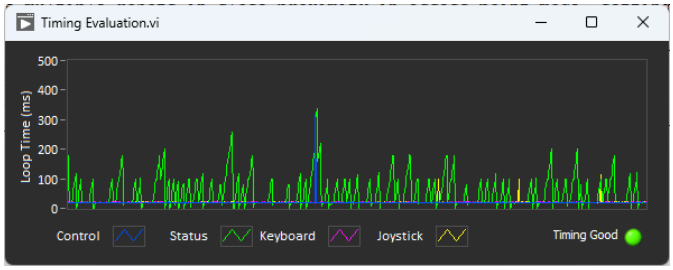


20.5.2 Viewing Timing

The Timing viewer shows the timing of the driver station loops measuring the joysick, keyboard, and control, and status network packets. When timing is good, all values should be close to 0 ms. This can help diagnose what is causing robot control issues.



The next image shows what it looks like when network congestion causes network packets to be delayed and combined. In this example, the communication was so bad that the robot wouldn't stay enabled or connected for more then a second.



20.6 Manually Setting the Driver Station to Start Custom Dashboard

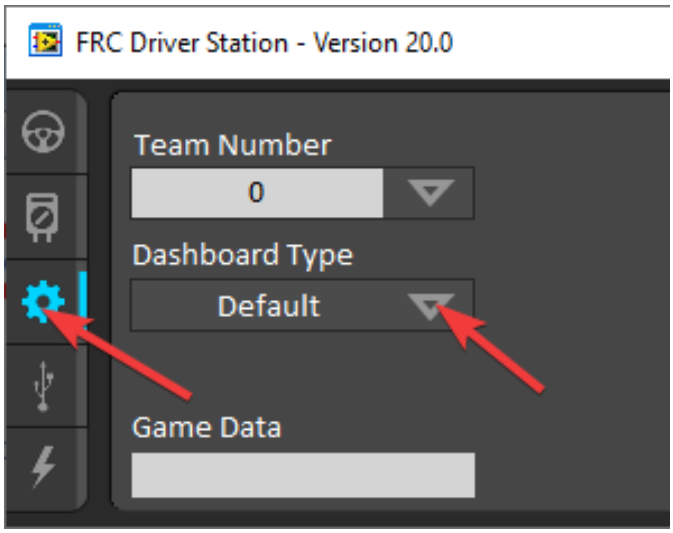
Note

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

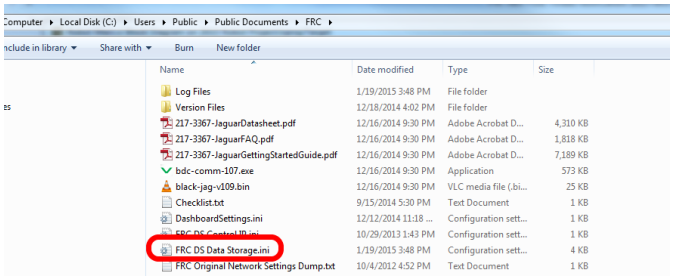
This is not needed for most installations, try using the appropriate *Dashboard Type setting* for your language first.

20.6.1 Set Driver Station to Default



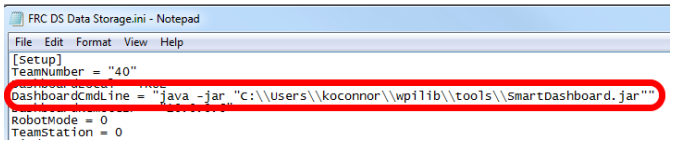
Open the Driver Station software, click on the Setup tab and set the Dashboard setting to Default. **Then close the Driver Station!**

20.6.2 Open DS Data Storage file



Browse to `C:\Users\Public\Documents\FRC` and double click on `FRC DS Data Storage` to open it.

20.6.3 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\\TO\\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 17.

Dashboard from WPILib installer

Replace the string after = with `C:\\Users\\Public\\wpilib\\YYYY\\tools\\DASHBOARD.exe` where `YYYY` is the year and `DASHBOARD.exe` is `Shuffleboard.exe`, `Smartdashboard.exe`, or `Elastic.exe`. For example `DashboardCmdLine = "wscript C:\\Users\\Public\\wpilib\\2026\\tools\\Elastic.exe"`. Save the `FRC DS Data Storage` file.

20.6.4 Launch Driver Station

The Driver Station should now launch the dashboard each time it is opened.

21 RobotBuilder

> **Warning**
>
> RobotBuilder is deprecated and will be removed for 2027 due to its declining usage and burden of updating for the 2027 control system.

21.1 RobotBuilder - Introduction

> **Warning**
>
> RobotBuilder is deprecated and will be removed for 2027 due to its declining usage and burden of updating for the 2027 control system.

21.1.1 RobotBuilder Overview

> **Warning**
>
> RobotBuilder is deprecated and will be removed for 2027 due to its declining usage and burden of updating for the 2027 control system.

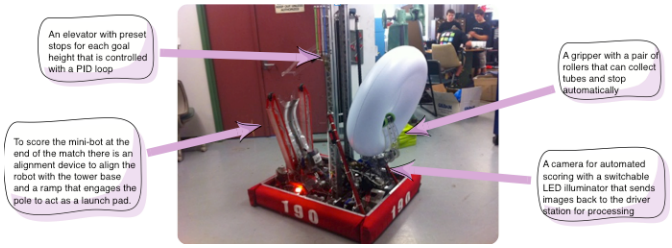
RobotBuilder is an application designed to aid the robot development process. RobotBuilder can help you:

- 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*.

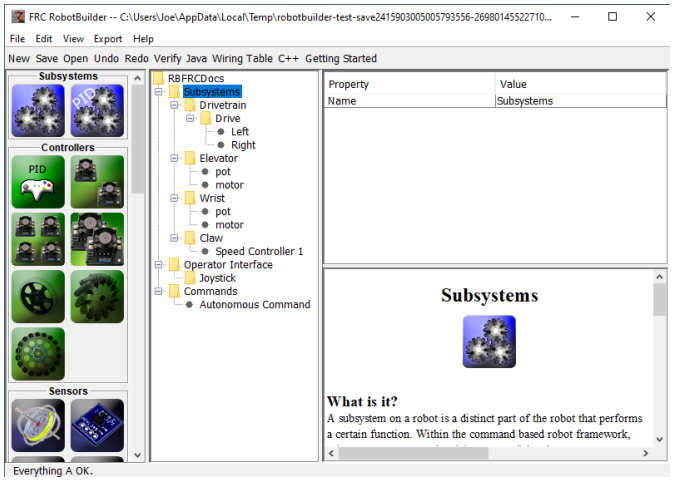
Divide the Robot into Subsystems



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



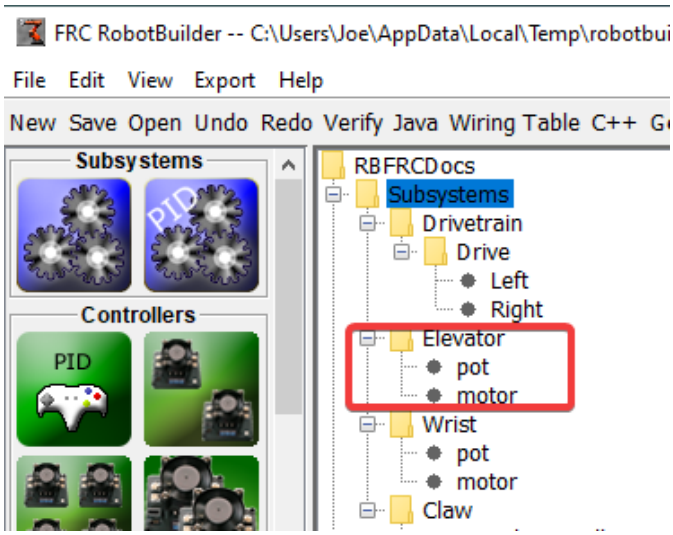
Each subsystem will be added to the “Subsystems” folder in the RobotBuilder and given a meaningful name. For each of the subsystems there are several attributes that get filled in to specify more information about the subsystems. In addition there are two types of subsystems that you might want to create:

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, more quickly to a desired position then stop when reaching it. PIDSubsystems have the PID Controller code built-in and are often more convenient then 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.

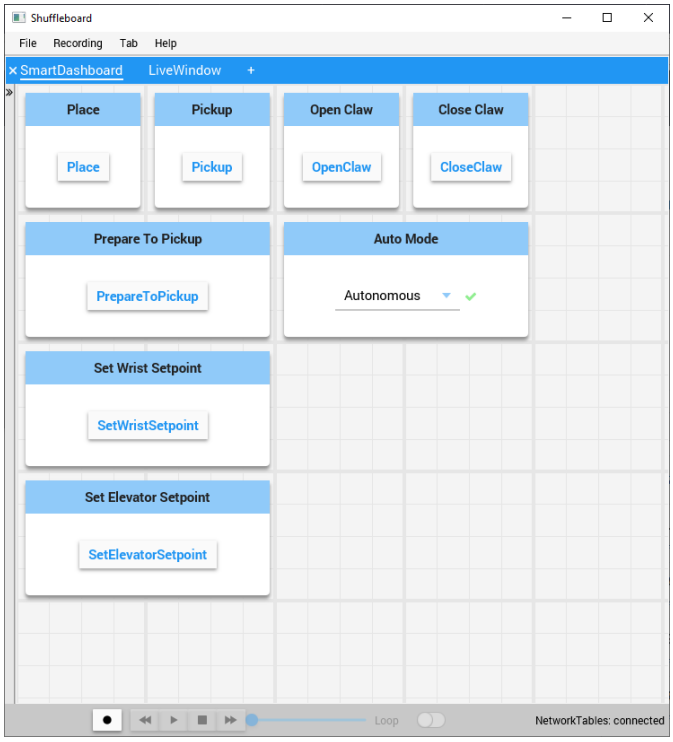
Adding Commands That Describe Subsystem Goals

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

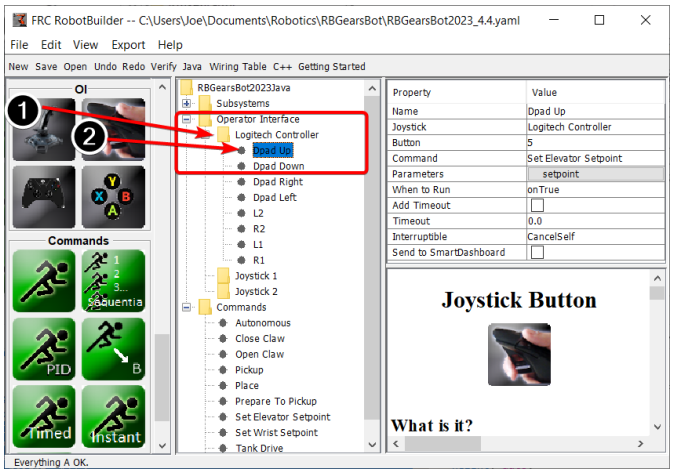


Each command can be run from Shuffleboard or SmartDashboard. This is useful for testing commands before you add them to the operator interface or to a command group. As long as you leave the "Button on SmartDashboard" property checked, a button will be created on the SmartDashboard. When you press the 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*.

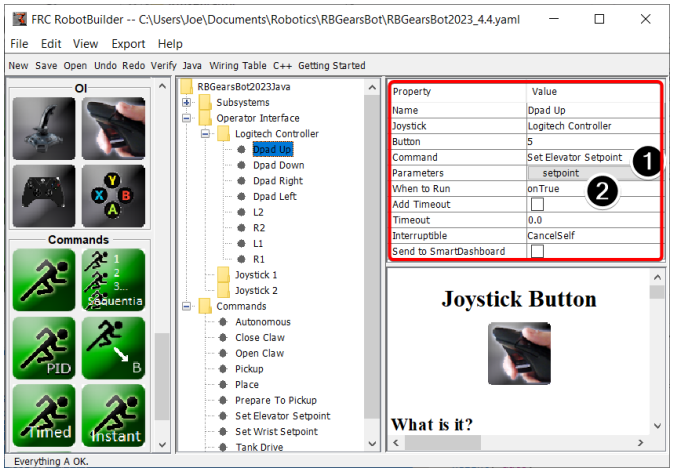
Adding Operator Interface Components



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

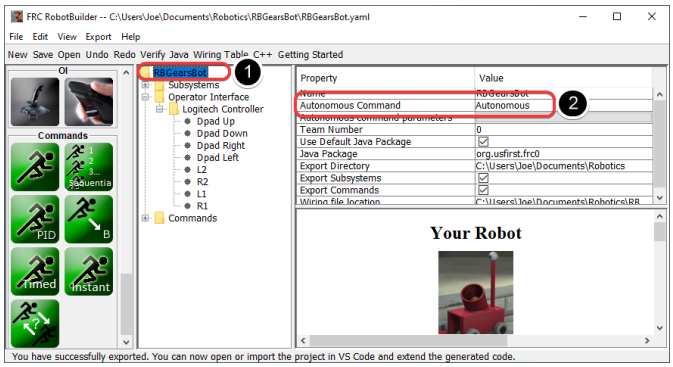


Commands can be associated with buttons so that when a button is pressed the command is scheduled. This should, for the most part, handle most of the tele-operated part of your robot program.

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

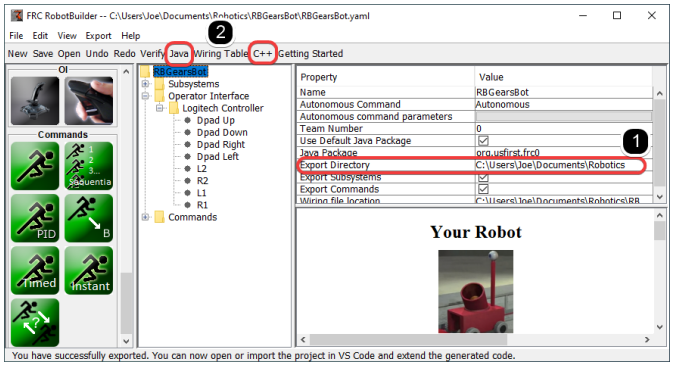


Commands make it simple to develop autonomous programs. You simply specify which command should run when the robot enters the autonomous period and it will automatically be scheduled. If you have tested commands as discussed above, this should simply be a matter of choosing which command should run.

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*.

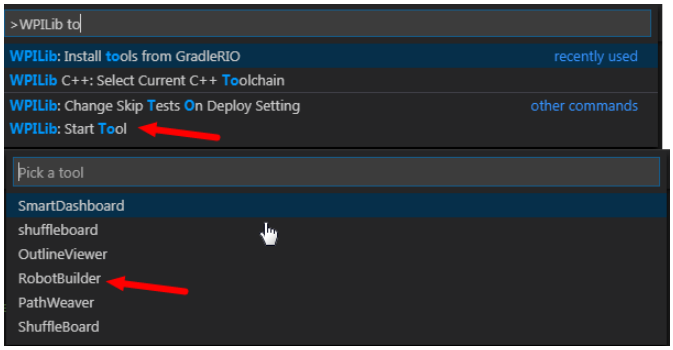
21.1.2 Starting RobotBuilder

> **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

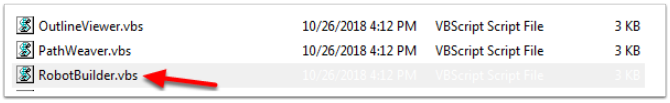


Press `Ctrl+Shift+P` and type “WPILib” or click the WPILib logo in the top right to launch the WPILib Command Palette. Select *Start Tool*, then select *Robot Builder*.

Option 2 - Shortcuts

Shortcuts are installed to the Windows Start Menu and the `2026 WPILib Tools` folder on the desktop.

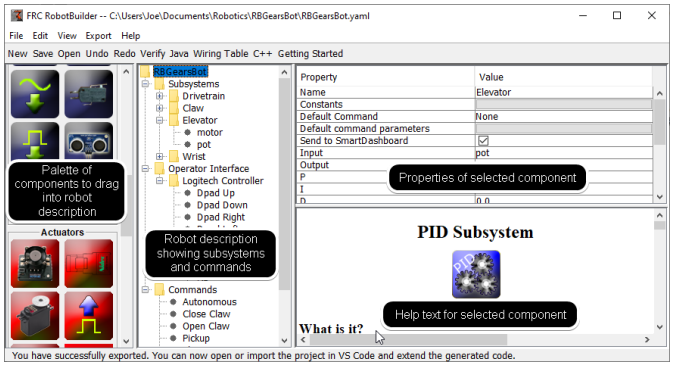
Option 3 - Running from the Script



The install process installs the tools to `~/wpilib/YYYY/tools` (where YYYY is the year and `~` is `C:\Users\Public` on Windows).

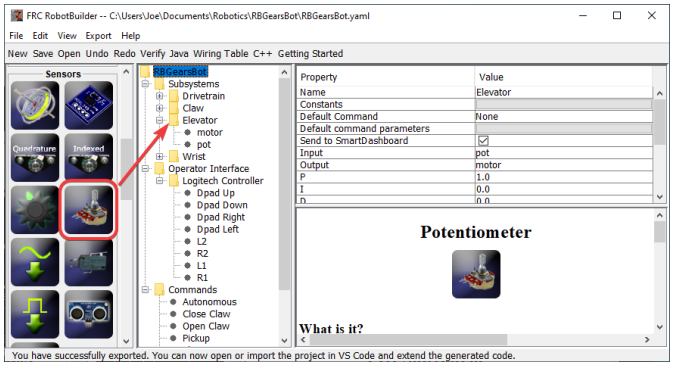
Inside this folder you will find `.exe` (Windows) and no extension (macOS/Linux) executables that you can use to launch each tool. These scripts help launch the tools using the correct JDK and are what you should use to launch the tools.

21.1.3 RobotBuilder User Interface



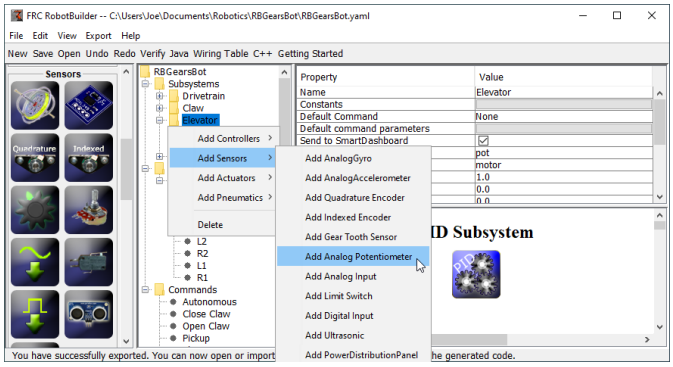
RobotBuilder has a user interface designed for rapid development of robot programs. Almost all operations are performed by drag-and-drop or selecting options from drop-down lists.

Dragging Items from the Palette to the Robot Description



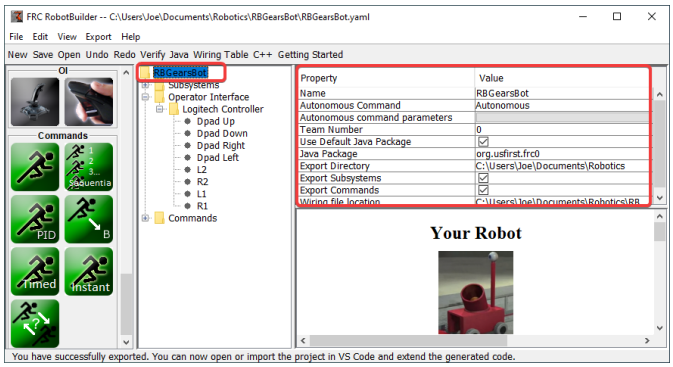
You can drag items from the palette to the robot description by starting the drag on the palette item and ending on the container where you would like the item to be located. In this example, dropping a potentiometer to the Elevator subsystem.

Adding Components using the Right-Click Context Menu



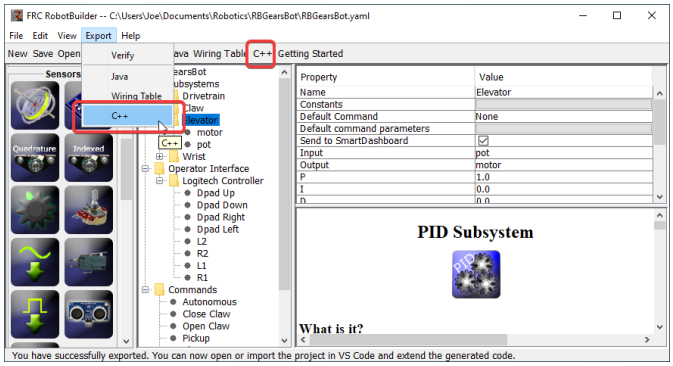
A shortcut method of adding items to the robot description is to right-click on the container object (Elevator) and select the item that should be added (Potentiometer). This is identical to using drag and drop but might be easier for some people.

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.

21.1.4 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).

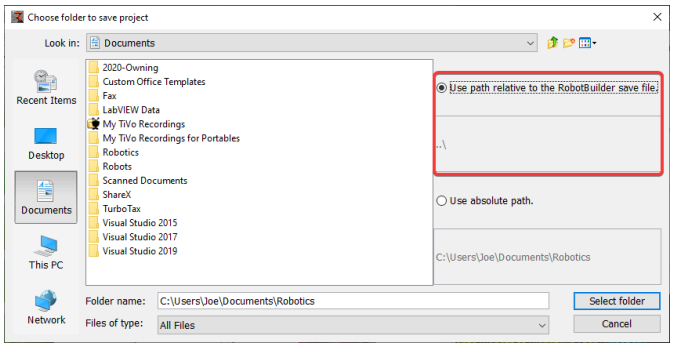
Robot Project Properties

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 location** - the location of the html file to generate that contains the wiring diagram for your robot

- **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.

Using Source Control with the RobotBuilder Project

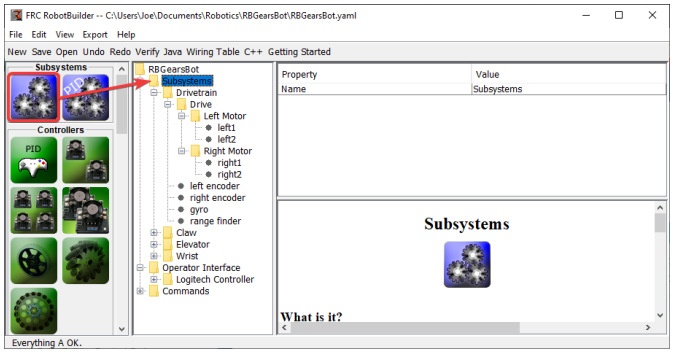


When using source control the project will typically be used on a number of computers and the path to the project directory might be different from one users computer to another. If the RobotBuilder project file is stored using an absolute path, it will typically contain the user name and won't be usable 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.

21.1.5 Creating a Subsystem

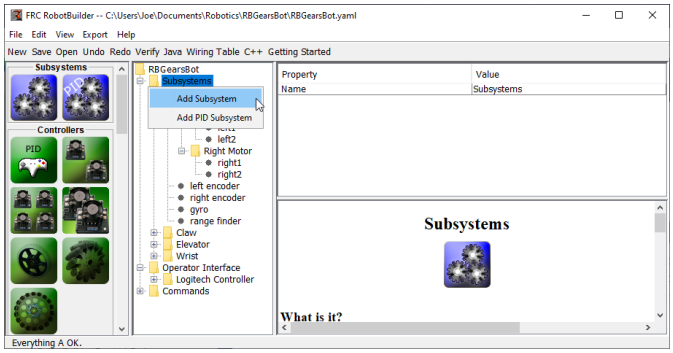
Subsystems are classes that encapsulate (or contain) all the data and code that make a subsystem on your robot operate. The first step in creating a robot program with the RobotBuilder is to identify and create all the subsystems on the robot. Examples of subsystems are grippers, ball collectors, the 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 motor controller and a potentiometer to provide feedback of the robot position.

Creating a Subsystem using the Palette



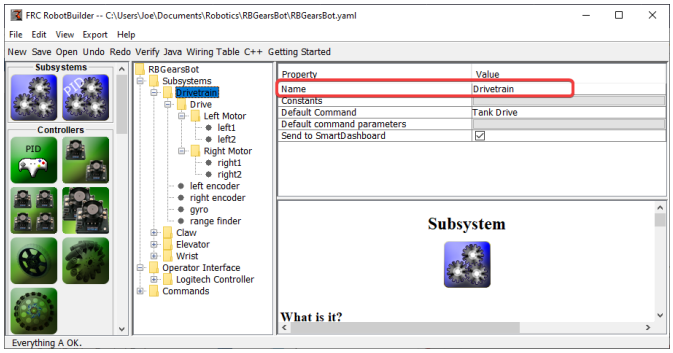
Drag the subsystem icon from the palette to the Subsystems folder in the robot description to create a subsystem class.

Creating a Subsystem using the Context Menu



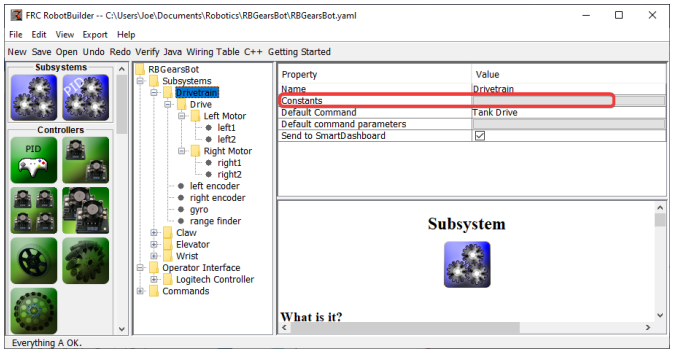
Right-click on the Subsystem folder in the robot description to add a subsystem to that folder.

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.

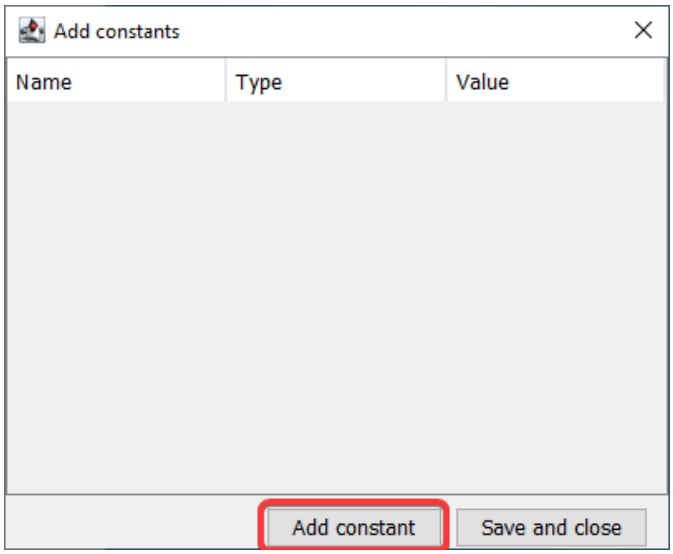
Adding Constants



Constants are very useful to reduce the amount of magic numbers in your code. In subsystems, 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.

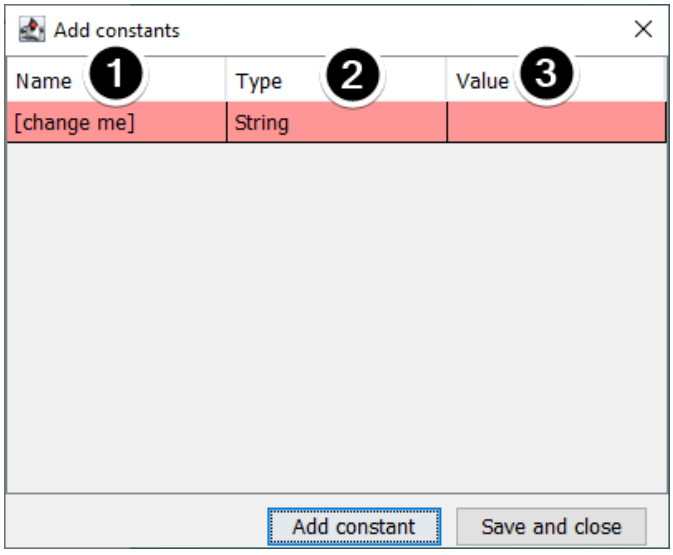
By default, there will be no constants in a subsystem. Press the button next to "Constants" to open a dialog to create some.

Creating Constants



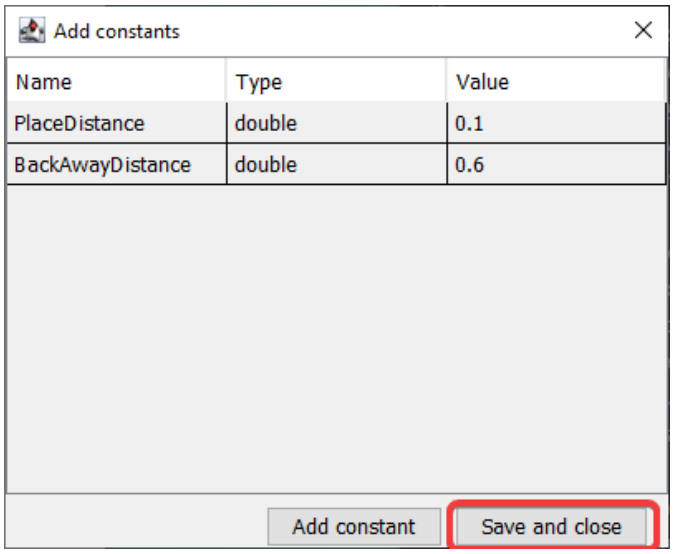
The constants table will be empty at first. Press "Add constant" to add one.

Add Constants



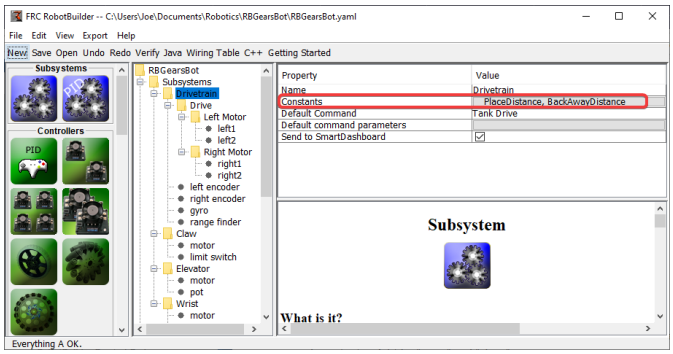
1. The name of the constant. Change this to something descriptive. In this example of a drivetrain some good constants might be "PlaceDistance" and "BackAwayDistance".
2. The type of the constant. This will most likely be a double, but you can choose from one of: String, double, int, long, boolean, or byte.
3. The value of the constant.

Saving Constants



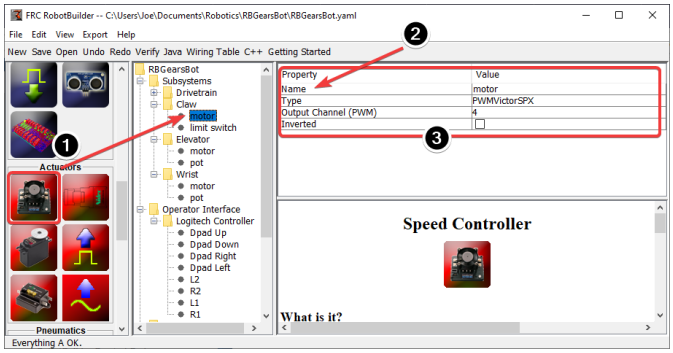
After adding constants and setting their values, just press “Save and close” to save the constants and close the dialog. If you don’t want to save, press the exit button on the top of the window.

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.

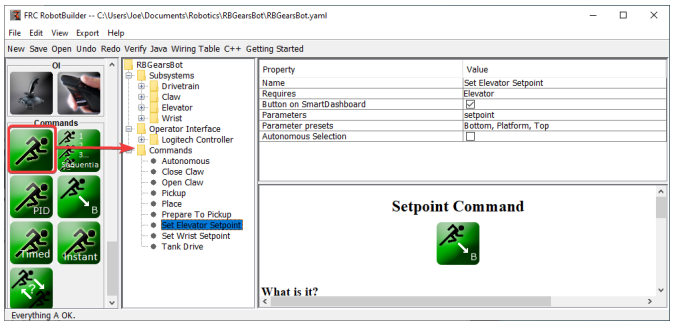
RobotBuilder will automatically use incrementing channel numbers for each module on the robot. If you haven't yet wired the robot you can just let RobotBuilder assign unique channel numbers for each sensor or actuator and wire the robot according to the generating wiring table.

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*.

21.1.6 Creating a Command

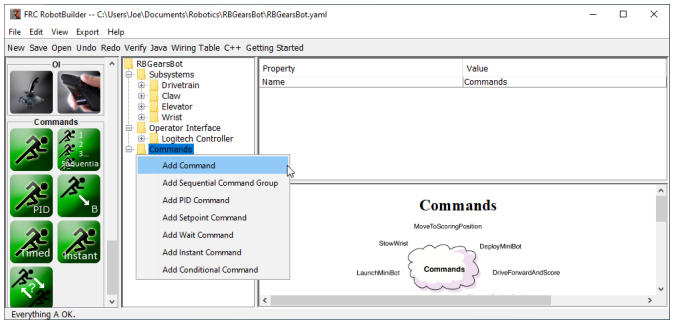
Commands are classes you create that provide behaviors or actions for your subsystems. The subsystem class should set the operation of the subsystem, like MoveElevator to start the elevator moving, or ElevatorToSetPoint to set the elevator's PID setpoint. The commands initiate the subsystem operation 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.

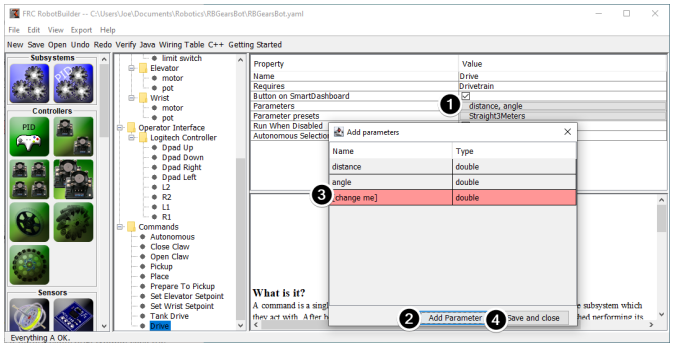
Configuring the Command

Property	Value
Name	Set Elevator Setpoint
Requires	Elevator
Button on SmartDashboard	☑
Parameters	setpoint
Parameter presets	Bottom, Platform, Top
Autonomous Selection	☐

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

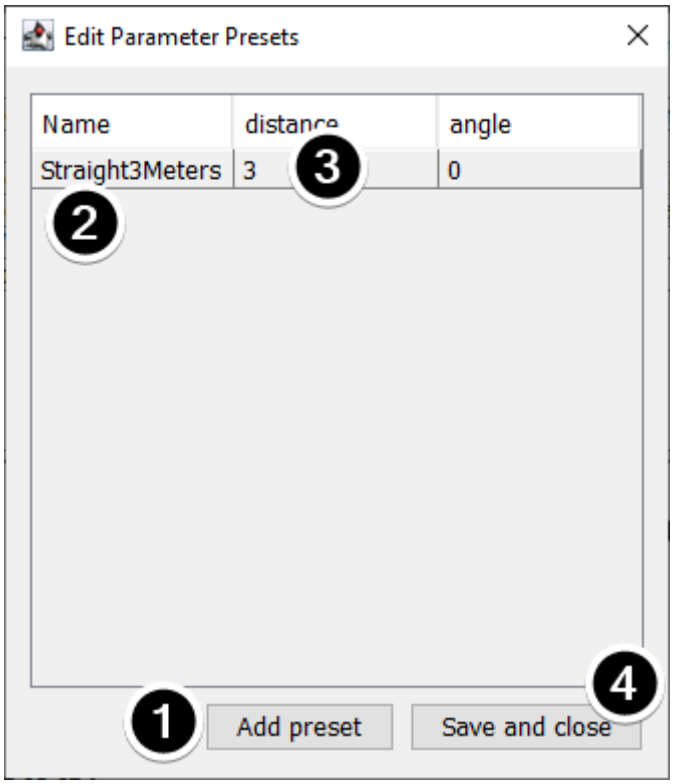


To add or edit parameters:

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

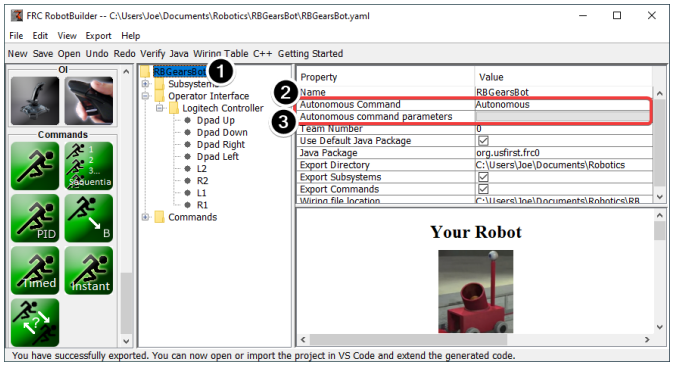


1. Click *Add parameter set* to add a new preset.
2. Change the name of the preset to something descriptive. The presets in this example are for opening and closing the gripper subsystem.
3. Change the value of the parameter(s) for the preset. You can either type a value in (e.g. "3.14") or select from 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.

21.1.7 Setting the Autonomous Commands

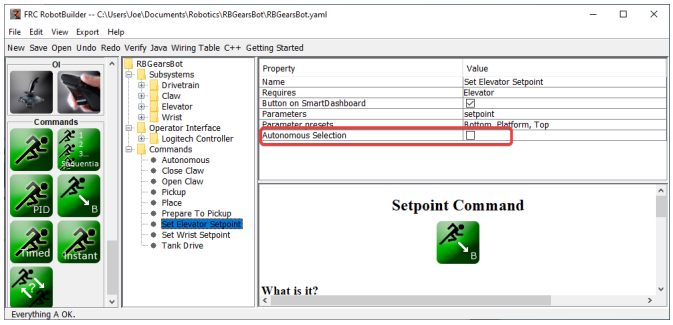
Since a command is simply one or more actions (behaviors) that the robot performs, it makes sense to describe the autonomous operation of a robot as a command. While it could be a single command, it is more likely going to be a command group (a group of commands that happen together).

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.

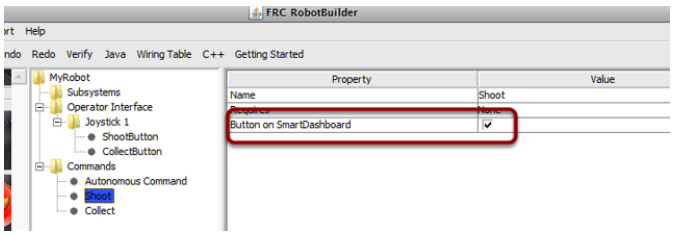
When the robot is put into autonomous mode, the chosen Autonomous command will be scheduled.

21.1.8 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.

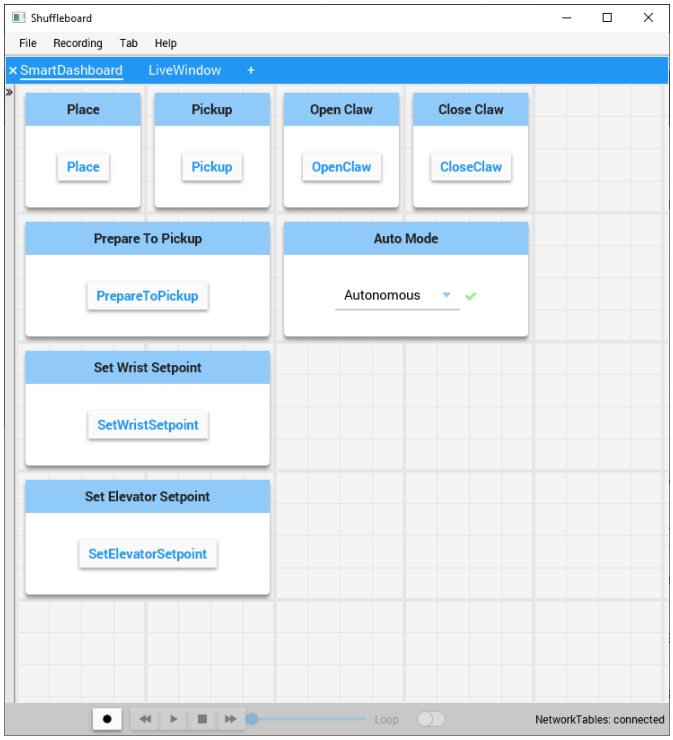
This has the added benefit of accommodating multiple programmers, each writing commands. As the code is checked into the main robot project, the commands can be individually tested.

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

```
SmartDashboard.putData("Autonomous Command", new AutonomousCommand());
SmartDashboard.putData("Open Claw", new OpenClaw(m_claw);
SmartDashboard.putData("Close Claw", new CloseClaw(m_claw));
```

C++

```
SmartDashboard::PutData("Autonomous Command", new AutonomousCommand());
SmartDashboard::PutData("Open Claw", new OpenClaw(&m_claw));
SmartDashboard::PutData("Close Claw", new CloseClaw(&m_claw));
```

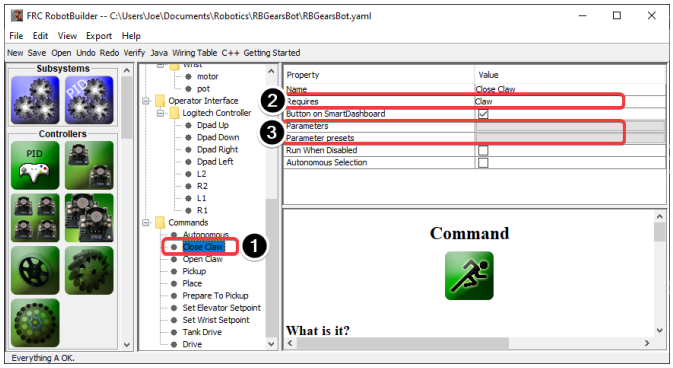
Commands can be added to the Shuffleboard manually by writing the code yourself. This is done by passing instances of the command to the PutData method along with the name that should be associated with the button on the Shuffleboard. These instances are scheduled whenever the button is pressed. The 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.

21.1.9 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

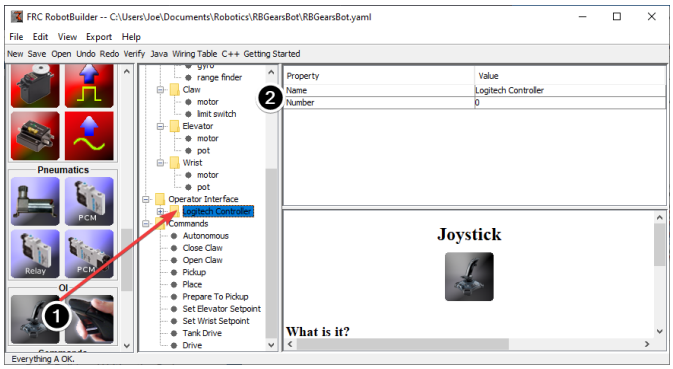


In this example we want to schedule the “Close Claw” command to run whenever the dpad right direction button is pressed on a logitech gamepad (button 6) is pressed.

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

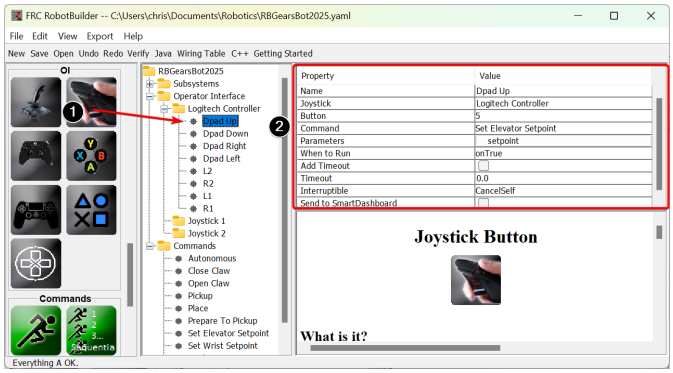
Adding the Joystick to the Robot Program



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

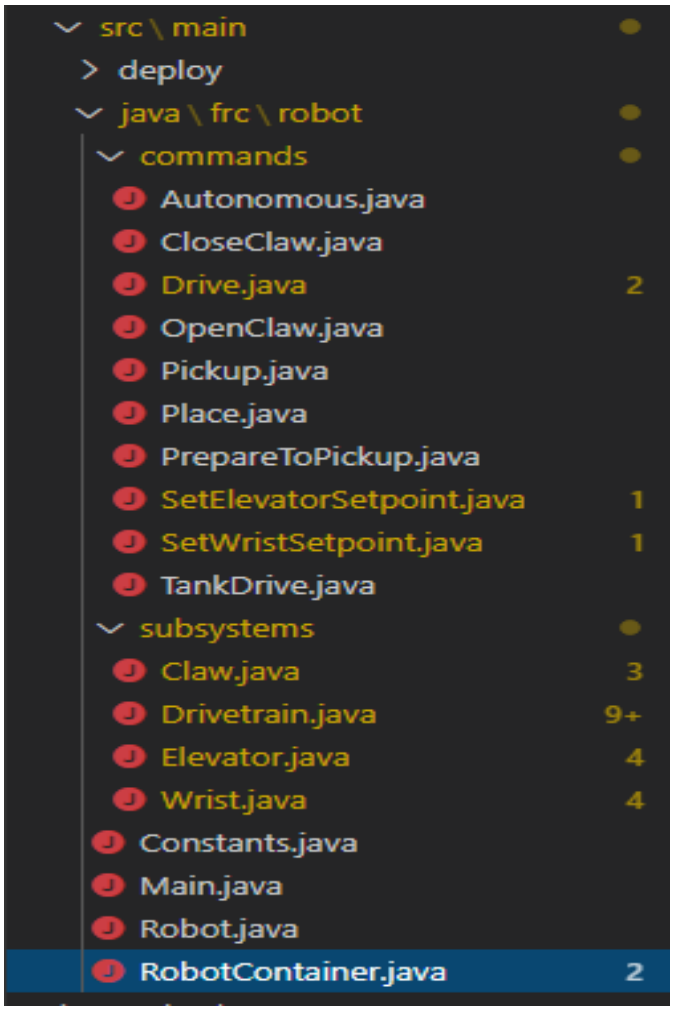
1. Drag the joystick button to the Joystick (Logitech Controller) so that it’s under the joystick
2. Set the properties for the button: the button number, the command to run when the button is pressed, parameters the command takes, and the *When to run* property to *onTrue* to indicate that the command should run whenever the joystick button is pressed.

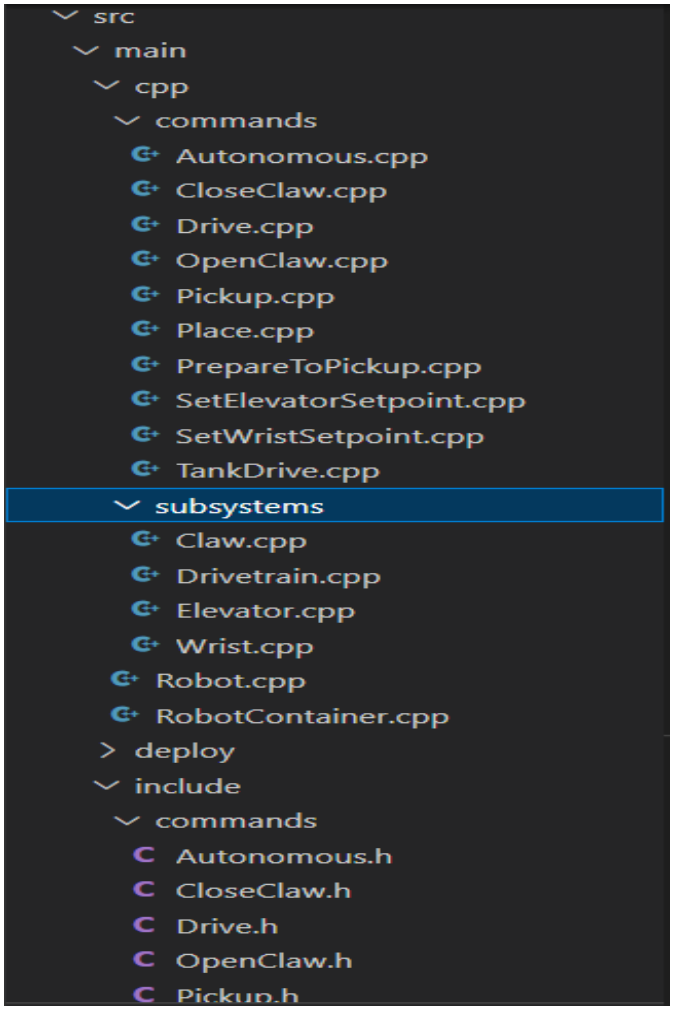
Note

Joystick buttons must be dragged to (under) a Joystick. You must have a joystick in the Operator Interface folder before adding buttons.

21.1.10 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

```
// 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);
```

C++

```
// BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=AUTONOMOUS
m_chooser.SetDefaultOption("Autonomous", new Autonomous());
// 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.

JAVA

```
// ROBOTBUILDER TYPE: Robot.
```

C++

```
// ROBOTBUILDER TYPE: Robot.
```

Additionally, each file has a comment defining the type of file. If this is modified or deleted, RobotBuilder will completely regenerate the file deleting any code added both inside and outside the AUTOGENERATED CODE blocks.

Main Robot Program

Java

```
11 // ROBOTBUILDER TYPE: Robot.
12
13 package frc.robot;
14
15 import edu.wpi.first.hal.FRCNetComm.tInstances;
16 import edu.wpi.first.hal.FRCNetComm.tResourceType;
17 import edu.wpi.first.hal.HAL;
18 import edu.wpi.first.wpilibj.TimedRobot;
19 import edu.wpi.first.wpilibj2.command.Command;
20 import edu.wpi.first.wpilibj2.command.CommandScheduler;
21
22 /**
23  * The methods in this class are called automatically corresponding to each mode, as␣
    ↪described in
```

(continues on next page)

(continued from previous page)

```
 * the TimedRobot documentation. If you change the name of this class or the package after creating
 * this project, you must also update the Main.java file in the project.
 */
public class Robot extends TimedRobot {

    private Command m_autonomousCommand;

    private RobotContainer m_robotContainer;

    /**
     * This function is run when the robot is first started up and should be
     * used for any initialization code.
     */
    public Robot() {
        // Instantiate our RobotContainer.  This will perform all our button bindings, and put our
        // autonomous chooser on the dashboard.
        m_robotContainer = RobotContainer.getInstance();
        HAL.report(tResourceType.kResourceType_Framework, tInstances.kFramework_RobotBuilder);
        enableLiveWindowInTest(true);
    }

    /**
    * This function is called every robot packet, no matter the mode. Use this for items like
    * diagnostics that you want ran during disabled, autonomous, teleoperated and test.
    *
    * <p>This runs after the mode specific periodic functions, but before
    * LiveWindow and SmartDashboard integrated updating.
    */
    @Override
    public void robotPeriodic() {
        // Runs the Scheduler.  This is responsible for polling buttons, adding newly-scheduled
        // commands, running already-scheduled commands, removing finished or interrupted commands,
        // and running subsystem periodic() methods.  This must be called from the robot's periodic
        // block in order for anything in the Command-based framework to work.
        CommandScheduler.getInstance().run();
    }


    /**
    * This function is called once each time the robot enters Disabled mode.
    */
    @Override
    public void disabledInit() {
    }

    @Override
    public void disabledPeriodic() {
    }

```

(continues on next page)

(continued from previous page)

```
    /**
    * This autonomous runs the autonomous command selected by your {@link
↪RobotContainer} class.
    */
    @Override
    public void autonomousInit() {
        m_autonomousCommand = m_robotContainer.getAutonomousCommand();

        // schedule the autonomous command (example)
        if (m_autonomousCommand != null) {
            m_autonomousCommand.schedule();
        }
    }

    /**
    * This function is called periodically during autonomous.
    */
    @Override
    public void autonomousPeriodic() {
    }

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

    /**
     * This function is called periodically during operator control.
     */
    @Override
    public void teleopPeriodic() {
    }

    @Override
    public void testInit() {
        // Cancels all running commands at the start of test mode.
        CommandScheduler.getInstance().cancelAll();
    }

    /**
    * This function is called periodically during test mode.
    */
    @Override
    public void testPeriodic() {
    }

}
```

C++ (Header)

```
// ROBOTBUILDER TYPE: Robot.
#pragma once

#include <frc/TimedRobot.h>
#include <frc2/command/Command.h>

#include "RobotContainer.h"

class Robot : public frc::TimedRobot {
 public:
  Robot();
  void RobotPeriodic() override;
  void DisabledInit() override;
  void DisabledPeriodic() override;
  void AutonomousInit() override;
  void AutonomousPeriodic() override;
  void TeleopInit() override;
  void TeleopPeriodic() override;
  void TestPeriodic() override;

 private:
  // Have it null by default so that if testing teleop it
  // doesn't have undefined behavior and potentially crash.
  frc2::Command* m_autonomousCommand = nullptr;

  RobotContainer* m_container = RobotContainer::GetInstance();
};
```

C++ (Source)

```
// ROBOTBUILDER TYPE: Robot.

#include "Robot.h"

#include <hal/FRCUsageReporting.h>

#include <frc/smartdashboard/SmartDashboard.h>
#include <frc2/command/CommandScheduler.h>

Robot::Robot() {
  EnableLiveWindowInTest(true);
  HAL_Report(HALUsageReporting::kResourceType_Framework,
             HALUsageReporting::kFramework_RobotBuilder);
}

/**
 * This function is called every robot packet, no matter the mode. Use
 * this for items like diagnostics that you want to run during disabled,
 * autonomous, teleoperated and test.
 *
 * <p> This runs after the mode specific periodic functions, but before
 * LiveWindow and SmartDashboard integrated updating.
 */
```

(continues on next page)

(continued from previous page)

```
void Robot::RobotPeriodic() { frc2::CommandScheduler::GetInstance().Run(); }

/**
 * This function is called once each time the robot enters Disabled mode. You
 * can use it to reset any subsystem information you want to clear when the
 * robot is disabled.
 */
void Robot::DisabledInit() {}

void Robot::DisabledPeriodic() {}

/**
 * This autonomous runs the autonomous command selected by your {@link
 * RobotContainer} class.
 */
void Robot::AutonomousInit() {
  m_autonomousCommand = m_container->GetAutonomousCommand();

  if (m_autonomousCommand != nullptr) {
    m_autonomousCommand->Schedule();
  }
}

void Robot::AutonomousPeriodic() {}

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;
  }
}

/**
 * This function is called periodically during operator control.
 */
void Robot::TeleopPeriodic() {}

/**
 * This function is called periodically during test mode.
 */
void Robot::TestPeriodic() {}

#ifndef RUNNING_FRC_TESTS
int main() { return frc::StartRobot<Robot>(); }
#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

```
// ROBOTBUILDER TYPE: RobotContainer.

package frc.robot;

import frc.robot.commands.*;
import frc.robot.subsystems.*;
import edu.wpi.first.wpilibj.smartdashboard.SendableChooser;
import edu.wpi.first.wpilibj.smartdashboard.SmartDashboard;
import edu.wpi.first.wpilibj2.command.Command.InterruptionBehavior;

// BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=IMPORTS
import edu.wpi.first.wpilibj2.command.Command;
import edu.wpi.first.wpilibj2.command.InstantCommand;
import edu.wpi.first.wpilibj.Joystick;
import edu.wpi.first.wpilibj2.command.button.JoystickButton;
import frc.robot.subsystems.*;

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=IMPORTS


/**
 * This class is where the bulk of the robot should be declared.  Since Command-based is a
 * "declarative" paradigm, very little robot logic should actually be handled in the {@link Robot}
 * periodic methods (other than the scheduler calls).  Instead, the structure of the robot
 * (including subsystems, commands, and button mappings) should be declared here.
 */
public class RobotContainer {

  private static RobotContainer m_robotContainer = new RobotContainer();

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
// The robot's subsystems
    public final Wrist m_wrist = new Wrist();
    public final Elevator m_elevator = new Elevator();
    public final Claw m_claw = new Claw();
    public final DriveTrain m_driveTrain = new DriveTrain();

// Joysticks
private final Joystick joystick2 = new Joystick(2);
private final Joystick joystick1 = new Joystick(1);
private final Joystick logitechController = new Joystick(0);

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS


```

(continues on next page)

(continued from previous page)

```
56    // A chooser for autonomous commands
57    SendableChooser<Command> m_chooser = new SendableChooser<>();
58
59    /**
60    * The container for the robot.  Contains subsystems, OI devices, and commands.
61    */
62    private RobotContainer() {
63          // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=SMARTDASHBOARD
64      // Smartdashboard Subsystems
65      SmartDashboard.putData(m_wrist);
66      SmartDashboard.putData(m_elevator);
67      SmartDashboard.putData(m_claw);
68      SmartDashboard.putData(m_driveTrain);
69
70
71      // SmartDashboard Buttons
72      SmartDashboard.putData("Close Claw", new CloseClaw( m_claw ));
73      SmartDashboard.putData("Open Claw: default", new OpenClaw(1));
74      SmartDashboard.putData("Pickup", new Pickup());
75      SmartDashboard.putData("Place", new Place());
76      SmartDashboard.putData("Prepare To Pickup", new PrepareToPickup());
77      SmartDashboard.putData("Set Elevator Setpoint: down", new SetElevatorSetpoint(1,␣
    ↪m_elevator));
78      SmartDashboard.putData("Set Elevator Setpoint: up", new SetElevatorSetpoint(2, m_
    ↪elevator));
79      SmartDashboard.putData("Set Wrist Setpoint: Horizontal", new SetWristSetpoint(0,␣
    ↪m_elevator));
80      SmartDashboard.putData("Set Wrist Setpoint: Raise Wrist", new SetWristSetpoint(-
    ↪45, m_elevator));
81      SmartDashboard.putData("Tank Drive: joystick", new TankDrive(() -> getJoystick1().
    ↪getY(), () -> getJoystick2().getY(), m_driveTrain));
82      SmartDashboard.putData("DriveAuto: default", new DriveAuto(10, 0));
83
84      // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=SMARTDASHBOARD
85      // Configure the button bindings
86      configureButtonBindings();
87
88      // Configure default commands
89          // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=SUBSYSTEM_DEFAULT_COMMAND
90      m_driveTrain.setDefaultCommand(new TankDrive(() -> getJoystick1().getY(), () ->␣
    ↪getJoystick2().getY(), m_driveTrain));
91
92
93      // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=SUBSYSTEM_DEFAULT_COMMAND
94
95      // Configure autonomous sendable chooser
96          // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=AUTONOMOUS
97
98      m_chooser.setDefaultOption("Autonomous", new Autonomous());
99
100     // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=AUTONOMOUS
101
102     SmartDashboard.putData("Auto Mode", m_chooser);
103   }
104
105   public static RobotContainer getInstance() {
106     return m_robotContainer;
```

(continues on next page)

(continued from previous page)

```
  }

  /**
   * Use this method to define your button->command mappings.  Buttons can be created by
   * instantiating a {@link GenericHID} or one of its subclasses ({@link
   * edu.wpi.first.wpilibj.Joystick} or {@link XboxController}), and then passing it to a
   * {@link edu.wpi.first.wpilibj2.command.button.JoystickButton}.
   */
  private void configureButtonBindings() {
        // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=BUTTONS
// Create some buttons
final JoystickButton r1 = new JoystickButton(logitechController, 12);
r1.onTrue(new SetWristSetpoint(0, m_elevator).withInterruptBehavior(InterruptionBehavior.kCancelSelf));

final JoystickButton l1 = new JoystickButton(logitechController, 11);
l1.onTrue(new Place().withInterruptBehavior(InterruptionBehavior.kCancelSelf));

final JoystickButton r2 = new JoystickButton(logitechController, 10);
r2.onTrue(new Pickup().withInterruptBehavior(InterruptionBehavior.kCancelSelf));

final JoystickButton l2 = new JoystickButton(logitechController, 9);
l2.onTrue(new PrepareToPickup().withInterruptBehavior(InterruptionBehavior.kCancelSelf));

final JoystickButton dpadLeft = new JoystickButton(logitechController, 8);
dpadLeft.onTrue(new OpenClaw(0).withInterruptBehavior(InterruptionBehavior.kCancelSelf));

final JoystickButton dpadRight = new JoystickButton(logitechController, 6);
dpadRight.onTrue(new CloseClaw( m_claw ).withInterruptBehavior(InterruptionBehavior.kCancelSelf));

final JoystickButton dpadDown = new JoystickButton(logitechController, 7);
dpadDown.onTrue(new SetElevatorSetpoint(2, m_elevator).withInterruptBehavior(InterruptionBehavior.kCancelSelf));

final JoystickButton dpadUp = new JoystickButton(logitechController, 5);
dpadUp.onTrue(new SetElevatorSetpoint(2, m_elevator).withInterruptBehavior(InterruptionBehavior.kCancelSelf));



    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=BUTTONS
  }

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=FUNCTIONS
public Joystick getLogitechController() {
        return logitechController;
    }

public Joystick getJoystick1() {
        return joystick1;
    }

```

(continues on next page)

(continued from previous page)

```
public Joystick getJoystick2() {
        return joystick2;
    }


    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=FUNCTIONS

  /**
   * Use this to pass the autonomous command to the main {@link Robot} class.
   *
   * @return the command to run in autonomous
   */
  public Command getAutonomousCommand() {
    // The selected command will be run in autonomous
    return m_chooser.getSelected();
  }


}
```

C++ (Header)

```
// ROBOTBUILDER TYPE: RobotContainer.

#pragma once

// BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=INCLUDES
#include <frc/smartdashboard/SendableChooser.h>
#include <frc2/command/Command.h>

#include "subsystems/Claw.h"
#include "subsystems/DriveTrain.h"
#include "subsystems/Elevator.h"
#include "subsystems/Wrist.h"

#include "commands/Autonomous.h"
#include "commands/CloseClaw.h"
#include "commands/DriveAuto.h"
#include "commands/OpenClaw.h"
#include "commands/Pickup.h"
#include "commands/Place.h"
#include "commands/PrepareToPickup.h"
#include "commands/SetElevatorSetpoint.h"
#include "commands/SetWristSetpoint.h"
#include "commands/TankDrive.h"
#include <frc/Joystick.h>
#include <frc2/command/button/JoystickButton.h>

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=INCLUDES

class RobotContainer {

public:

```

(continues on next page)

(continued from previous page)

```
    frc2::Command* GetAutonomousCommand();
    static RobotContainer* GetInstance();

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=PROTOTYPES
// The robot's subsystems
DriveTrain m_driveTrain;
Claw m_claw;
Elevator m_elevator;
Wrist m_wrist;


frc::Joystick* getJoystick2();
frc::Joystick* getJoystick1();
frc::Joystick* getLogitechController();

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=PROTOTYPES

private:

    RobotContainer();

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
// Joysticks
frc::Joystick m_logitechController{0};
frc::Joystick m_joystick1{1};
frc::Joystick m_joystick2{2};

frc::SendableChooser<frc2::Command*> m_chooser;

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS

    static RobotContainer* m_robotContainer;

    void ConfigureButtonBindings();
};
```

C++ (Source)

```
// ROBOTBUILDER TYPE: RobotContainer.

#include "RobotContainer.h"
#include <frc2/command/ParallelRaceGroup.h>
#include <frc/smartdashboard/SmartDashboard.h>



RobotContainer* RobotContainer::m_robotContainer = NULL;

RobotContainer::RobotContainer(){
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTOR


    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTOR

```

(continues on next page)

(continued from previous page)

```
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=SMARTDASHBOARD
    // Smartdashboard Subsystems
    frc::SmartDashboard::PutData(&m_driveTrain);
    frc::SmartDashboard::PutData(&m_claw);
    frc::SmartDashboard::PutData(&m_elevator);
    frc::SmartDashboard::PutData(&m_wrist);


    // SmartDashboard Buttons
    frc::SmartDashboard::PutData("DriveAuto: default", new DriveAuto(10, 0));
    frc::SmartDashboard::PutData("Tank Drive: joystick", new TankDrive([this] {return␣
↪getJoystick1()->GetY();}, [this] {return getJoystick2()->GetY();}, &m_driveTrain));
    frc::SmartDashboard::PutData("Set Wrist Setpoint: Horizontal", new␣
↪SetWristSetpoint(0, &m_elevator));
    frc::SmartDashboard::PutData("Set Wrist Setpoint: Raise Wrist", new␣
↪SetWristSetpoint(-45, &m_elevator));
    frc::SmartDashboard::PutData("Set Elevator Setpoint: down", new␣
↪SetElevatorSetpoint(1, &m_elevator));
    frc::SmartDashboard::PutData("Set Elevator Setpoint: up", new␣
↪SetElevatorSetpoint(2, &m_elevator));
    frc::SmartDashboard::PutData("Prepare To Pickup", new PrepareToPickup());
    frc::SmartDashboard::PutData("Place", new Place());
    frc::SmartDashboard::PutData("Pickup", new Pickup());
    frc::SmartDashboard::PutData("Open Claw: default", new OpenClaw(1_s));
    frc::SmartDashboard::PutData("Close Claw", new CloseClaw( &m_claw ));

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=SMARTDASHBOARD

    ConfigureButtonBindings();

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DEFAULT-COMMANDS
m_driveTrain.SetDefaultCommand(TankDrive([this] {return getJoystick1()->GetY();},␣
↪[this] {return getJoystick2()->GetY();}, &m_driveTrain));

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DEFAULT-COMMANDS

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=AUTONOMOUS

    m_chooser.SetDefaultOption("Autonomous", new Autonomous());

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=AUTONOMOUS

    frc::SmartDashboard::PutData("Auto Mode", &m_chooser);

}

RobotContainer* RobotContainer::GetInstance() {
    if (m_robotContainer == NULL) {
        m_robotContainer = new RobotContainer();
    }
    return(m_robotContainer);
}

void RobotContainer::ConfigureButtonBindings() {
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=BUTTONS

frc2::JoystickButton m_dpadUp{&m_logitechController, 5};
```

(continues on next page)

(continued from previous page)

```
frc2::JoystickButton m_dpadDown{&m_logitechController, 7};
frc2::JoystickButton m_dpadRight{&m_logitechController, 6};
frc2::JoystickButton m_dpadLeft{&m_logitechController, 8};
frc2::JoystickButton m_l2{&m_logitechController, 9};
frc2::JoystickButton m_r2{&m_logitechController, 10};
frc2::JoystickButton m_l1{&m_logitechController, 11};
frc2::JoystickButton m_r1{&m_logitechController, 12};

m_dpadUp.OnTrue(SetElevatorSetpoint(2, &m_elevator).
↪WithInterruptBehavior(frc2::Command::InterruptionBehavior::kCancelSelf));

m_dpadDown.OnTrue(SetElevatorSetpoint(2, &m_elevator).
↪WithInterruptBehavior(frc2::Command::InterruptionBehavior::kCancelSelf));

m_dpadRight.OnTrue(CloseClaw( &m_claw ).
↪WithInterruptBehavior(frc2::Command::InterruptionBehavior::kCancelSelf));

m_dpadLeft.OnTrue(OpenClaw(0_s).
↪WithInterruptBehavior(frc2::Command::InterruptionBehavior::kCancelSelf));

m_l2.OnTrue(PrepareToPickup().
↪WithInterruptBehavior(frc2::Command::InterruptionBehavior::kCancelSelf));

m_r2.OnTrue(Pickup().
↪WithInterruptBehavior(frc2::Command::InterruptionBehavior::kCancelSelf));

m_l1.OnTrue(Place().
↪WithInterruptBehavior(frc2::Command::InterruptionBehavior::kCancelSelf));

m_r1.OnTrue(SetWristSetpoint(0, &m_elevator).
↪WithInterruptBehavior(frc2::Command::InterruptionBehavior::kCancelSelf));


    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=BUTTONS
}

// BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=FUNCTIONS

frc::Joystick* RobotContainer::getLogitechController() {
   return &m_logitechController;
}
frc::Joystick* RobotContainer::getJoystick1() {
   return &m_joystick1;
}
frc::Joystick* RobotContainer::getJoystick2() {
   return &m_joystick2;
}

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=FUNCTIONS


frc2::Command* RobotContainer::GetAutonomousCommand() {
  // The selected command will be run in autonomous
  return m_chooser.GetSelected();
}
```

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):

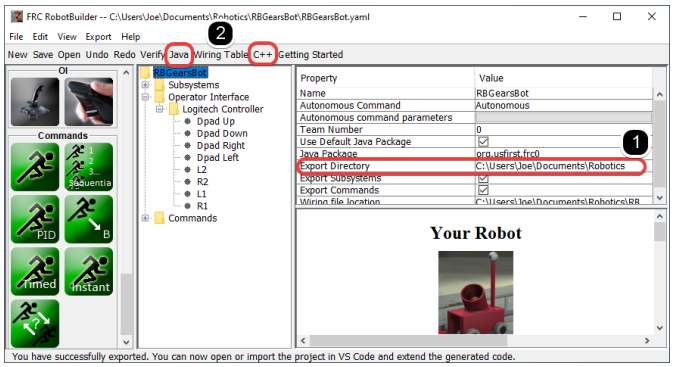
1. Each of the subsystems is declared here. They can be passed as parameters to any commands that require them.
2. If there is an autonomous command provided in RobotBuilder robot properties, it is added to the Sendable Chooser to be selected on the dashboard.
3. The code for all the operator interface 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.

21.2 RobotBuilder - Writing the Code

21.2.1 Generating Code for a Project

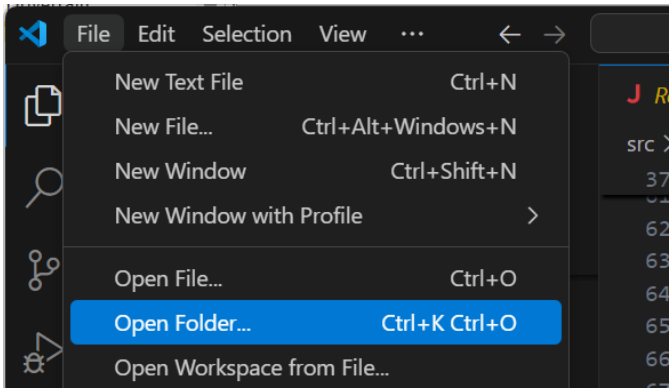
After you've set up your robot framework in RobotBuilder, you'll need to export the code and load it into Visual Studio Code. This article describes the process for doing so.

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

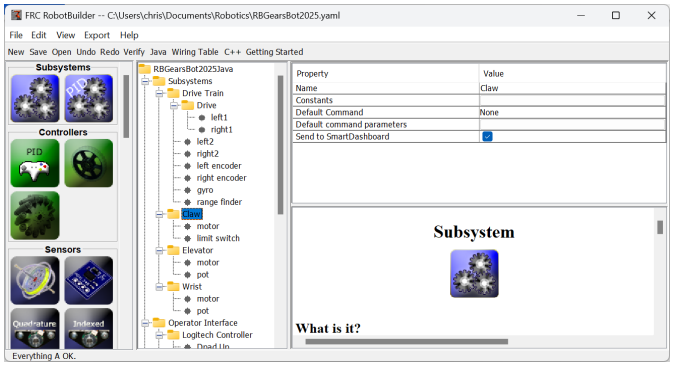


Open VS Code and select **File -> Open Folder**. Navigate to your Export location and click **Select Folder**.

21.2.2 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

```
// ROBOTBUILDER TYPE: Subsystem.

package frc.robot.subsystems;


import frc.robot.commands.*;
import edu.wpi.first.wpilibj.livewindow.LiveWindow;
import edu.wpi.first.wpilibj2.command.SubsystemBase;

// BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=IMPORTS
import edu.wpi.first.wpilibj.DigitalInput;
import edu.wpi.first.wpilibj.motorcontrol.PWMVictorSPX;

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=IMPORTS


/**
 *
 */
public class Claw extends SubsystemBase {
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTANTS

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTANTS

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
private PWMVictorSPX motor;
private DigitalInput limitswitch;

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS

    /**
    *
    */
    public Claw() {
        // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTORS
motor = new PWMVictorSPX(4);
 addChild("motor",motor);
 motor.setInverted(false);

limitswitch = new DigitalInput(4);
 addChild("limit switch", limitswitch);



    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTORS
    }

    @Override
    public void periodic() {
        // This method will be called once per scheduler run

    }

```

(continues on next page)

(continued from previous page)

```
    @Override
    public void simulationPeriodic() {
        // This method will be called once per scheduler run when in simulation

    }

    // Put methods for controlling this subsystem
    // here. Call these from Commands.

    public void open() {
        motor.set(1.0);
    }

    public void close() {
        motor.set(-1.0);
    }

    public void stop() {
        motor.set(0.0);
    }

    public boolean isGripping() {
        return limitswitch.get();
    }

}
```

C++

```
// ROBOTBUILDER TYPE: Subsystem.

// BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=INCLUDES
#include "subsystems/Claw.h"
#include <frc/smartdashboard/SmartDashboard.h>

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=INCLUDES

Claw::Claw(){
    SetName("Claw");
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
    SetSubsystem("Claw");

 AddChild("limit switch", &m_limitswitch);


 AddChild("motor", &m_motor);
 m_motor.SetInverted(false);


    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
}

void Claw::Periodic() {
    // Put code here to be run every loop
```

(continues on next page)

(continued from previous page)

```
36
37  }
38
39  void Claw::SimulationPeriodic() {
40      // This method will be called once per scheduler run when in simulation
41
42  }
43
44  // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CMDPIDGETTERS
45
46      // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CMDPIDGETTERS
47
48
49  // Put methods for controlling this subsystem
50  // here. Call these from Commands.
51
52  void Claw::Open() {
53      m_motor.Set(1.0);
54  }
55
56  void Claw::Close() {
57      m_motor.Set(-1.0);
58  }
59
60  void Claw::Stop() {
61      m_motor.Set(0.0);
62  }
63
64  bool Claw::IsGripping() {
65      return m_limitswitch.Get();
66  }
```

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.

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++

```
11  // ROBOTBUILDER TYPE: Subsystem.
12  #pragma once
13
14  // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=INCLUDES
15  #include <frc2/command/SubsystemBase.h>
16  #include <frc/DigitalInput.h>
17  #include <frc/motorcontrol/PWMVictorSPX.h>
18
19      // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=INCLUDES
20
```

(continues on next page)

(continued from previous page)

```
/**
 *
 *
 * @author ExampleAuthor
 */
class Claw: 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::DigitalInput m_limitswitch{4};
frc::PWMVictorSPX m_motor{4};

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
public:
Claw();

    void Periodic() override;
    void SimulationPeriodic() override;
    void Open();
    void Close();
    void Stop();
    bool IsGripping();

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CMDPIDGETTERS

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CMDPIDGETTERS
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTANTS

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTANTS


};
```

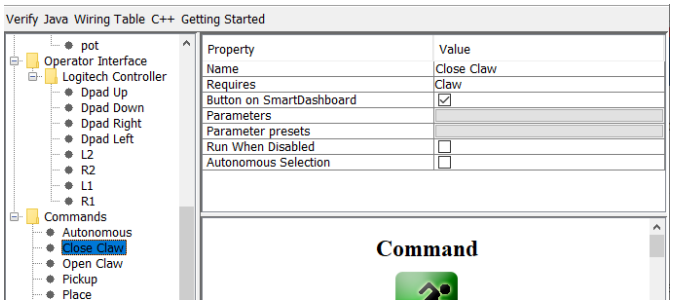
In addition to adding the methods to the class implementation file, `Claw.cpp`, the declarations for the methods need to be added to the header file, `Claw.h`. Those declarations that must be added are shown here.

To add the behavior to the claw subsystem to handle opening and closing you need to *define commands*.

21.2.3 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

```
// ROBOTBUILDER TYPE: Command.

package frc.robot.commands;
import edu.wpi.first.wpilibj2.command.Command;
import java.util.function.DoubleSupplier;

// BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=IMPORTS
import frc.robot.subsystems.Claw;

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=IMPORTS

/**
 *
 */
public class CloseClaw extends Command {

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=VARIABLE_DECLARATIONS
        private final Claw m_claw;

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=VARIABLE_DECLARATIONS

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTORS


    public CloseClaw(Claw subsystem) {


    // 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_claw = subsystem;
        addRequirements(m_claw);
```

(continues on next page)

(continued from previous page)

```

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=REQUIRES
    }

    // Called when the command is initially scheduled.
    @Override
    public void initialize() {
        m_claw.close();
    }

    // Called every time the scheduler runs while the command is scheduled.
    @Override
    public void execute() {
    }

    // Called once the command ends or is interrupted.
    @Override
    public void end(boolean interrupted) {
        m_claw.stop();
    }

    // Returns true when the command should end.
    @Override
    public boolean isFinished() {
        return m_claw.isGripping();
    }

    @Override
    public boolean runsWhenDisabled() {
        // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DISABLED
        return false;

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DISABLED
    }
}
```

C++

```
// ROBOTBUILDER TYPE: Command.

// BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTOR

#include "commands/CloseClaw.h"

CloseClaw::CloseClaw(Claw* m_claw)
:m_claw(m_claw){

    // Use AddRequirements() here to declare subsystem dependencies
    // eg. AddRequirements(m_Subsystem);
    SetName("CloseClaw");
    AddRequirements({m_claw});

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTOR

```

(continues on next page)

(continued from previous page)

```
}

// Called just before this Command runs the first time
void CloseClaw::Initialize() {
    m_claw->Close();
}

// Called repeatedly when this Command is scheduled to run
void CloseClaw::Execute() {

}

// Make this return true when this Command no longer needs to run execute()
bool CloseClaw::IsFinished() {
    return m_claw->IsGripping();
}

// Called once after isFinished returns true
void CloseClaw::End(bool interrupted) {
    m_claw->Stop();
}

bool CloseClaw::RunsWhenDisabled() const {
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DISABLED
    return false;

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DISABLED
}
```

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.

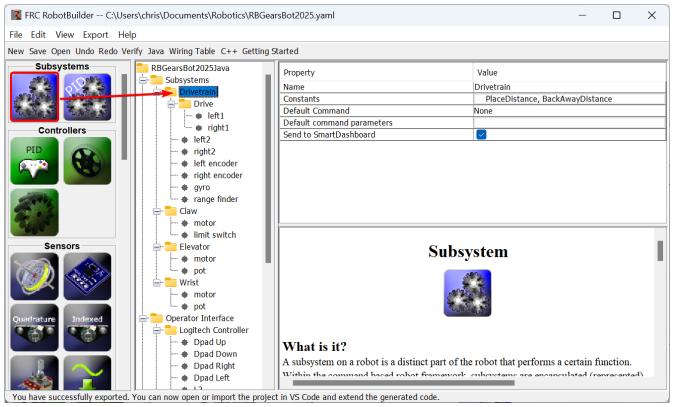
1. Start the claw motor moving in the closing direction by calling the `Close()` method that was added to the *Claw* subsystem in the *CloseClaw* Initialize method.
2. This command is finished when the limit switch in the *Claw* subsystem is tripped.
3. The `End()` method is called when the command is finished and is a place to clean up. In this case, the motor is stopped since the time has run out.

21.2.4 Driving the Robot with Tank Drive and Joysticks

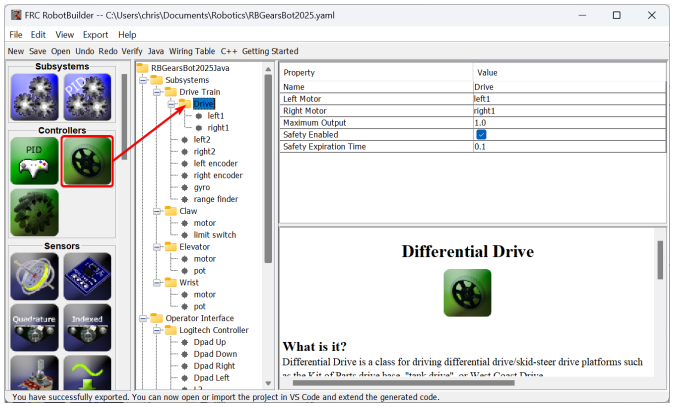
A common use case is to have a joystick that should drive some actuators that are part of a subsystem. The problem is that the joystick is created in the RobotContainer class and the motors to be controlled are in the subsystem. The idea is to create a command that, when scheduled, reads input from the 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.

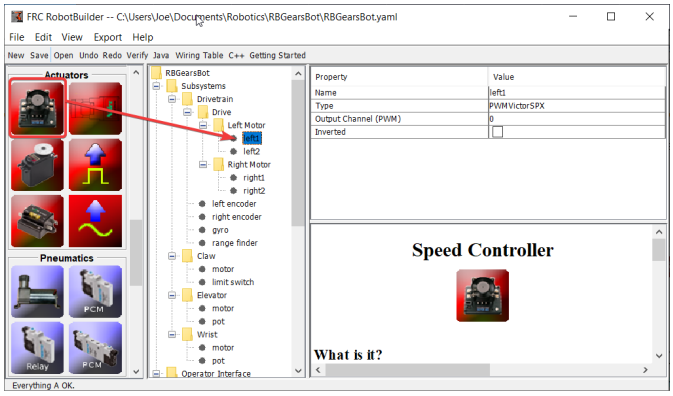
Create a Drive Train Subsystem



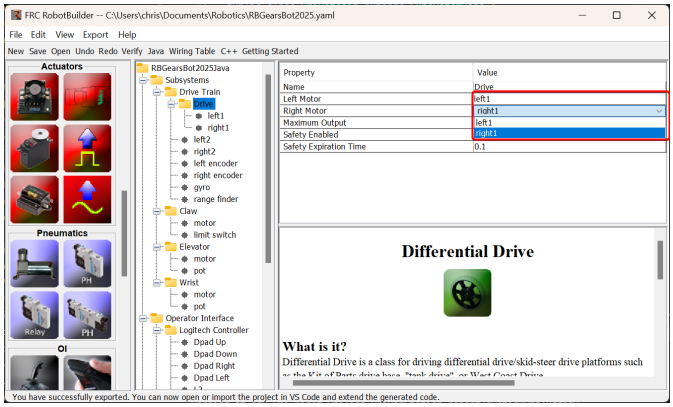
Create a subsystem called Drive Train. Its responsibility will be to handle the driving for the robot base.



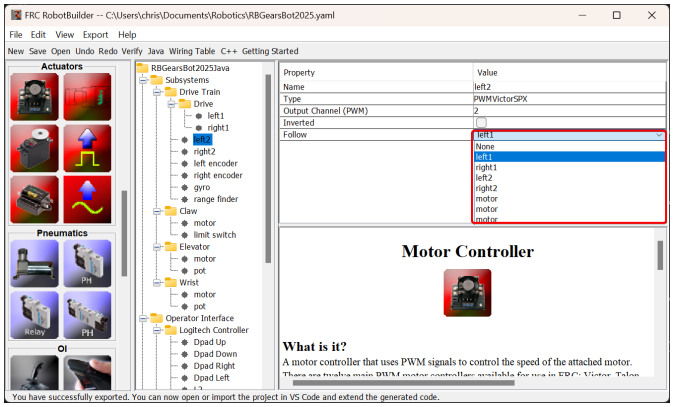
Inside the Drive Train, create a Differential Drive object. The Differential Drive object can contain a left motor controller and a right motor controller.



Create two Motor Controllers in the Differential Drive, and two more outside the Differential Drive, inside the Drive Train subsystem.

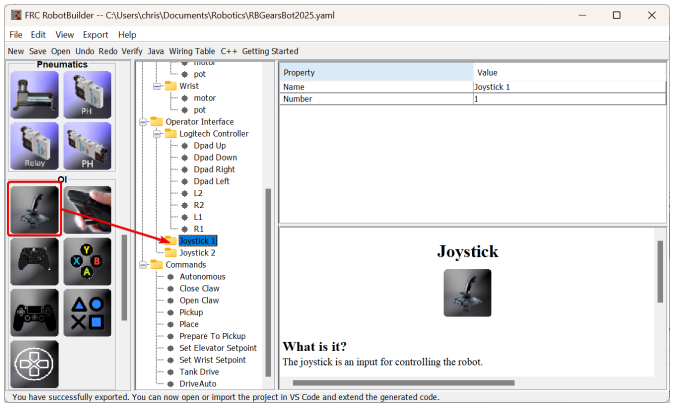


In the Differential Drive, set the left and right motors to the appropriate motor controllers.



Finally, for the two motor controllers that aren't in the differential drive, set them to follow the motor controllers in the differential drive.

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

```
// ROBOTBUILDER TYPE: Subsystem.

package frc.robot.subsystems;


import frc.robot.commands.*;
import edu.wpi.first.wpilibj.livewindow.LiveWindow;
import edu.wpi.first.wpilibj2.command.SubsystemBase;

// BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=IMPORTS
import edu.wpi.first.wpilibj.AnalogGyro;
import edu.wpi.first.wpilibj.AnalogInput;
import edu.wpi.first.wpilibj.CounterBase.EncodingType;
import edu.wpi.first.wpilibj.Encoder;
import edu.wpi.first.wpilibj.drive.DifferentialDrive;
import edu.wpi.first.wpilibj.motorcontrol.PWMVictorSPX;

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=IMPORTS


/**
 *
 */
public class DriveTrain extends SubsystemBase {
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTANTS
public static final double PlaceDistance = 0.1;
public static final double BackAwayDistance = 0.6;

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTANTS

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
private PWMVictorSPX left1;
private PWMVictorSPX right1;
private DifferentialDrive drive;
private PWMVictorSPX left2;
private PWMVictorSPX right2;
private Encoder leftencoder;
private Encoder rightencoder;
private AnalogGyro gyro;
private AnalogInput rangefinder;

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS

    /**
    *
    */
    public DriveTrain() {
        // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTORS
```

(continues on next page)

(continued from previous page)

```
left1 = new PWMVictorSPX(0);
 addChild("left1",left1);
 left1.setInverted(false);

right1 = new PWMVictorSPX(1);
 addChild("right1",right1);
 right1.setInverted(true);

drive = new DifferentialDrive(left1, right1);
 addChild("Drive",drive);
 drive.setSafetyEnabled(true);
drive.setExpiration(0.1);
drive.setMaxOutput(1.0);


left2 = new PWMVictorSPX(2);
 addChild("left2",left2);
 left1.addFollower(left2);
left2.setInverted(false);

right2 = new PWMVictorSPX(3);
 addChild("right2",right2);
 right1.addFollower(right2);
right2.setInverted(false);

leftencoder = new Encoder(0, 1, false, EncodingType.k4X);
 addChild("left encoder",leftencoder);
 leftencoder.setDistancePerPulse(1.0);

rightencoder = new Encoder(2, 3, false, EncodingType.k4X);
 addChild("right encoder",rightencoder);
 rightencoder.setDistancePerPulse(1.0);

gyro = new AnalogGyro(0);
 addChild("gyro",gyro);
 gyro.setSensitivity(0.007);

rangefinder = new AnalogInput(1);
 addChild("range finder", rangefinder);



    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTORS
    }

    @Override
    public void periodic() {
        // This method will be called once per scheduler run

    }

    @Override
    public void simulationPeriodic() {
        // This method will be called once per scheduler run when in simulation

    }

```

(continues on next page)

(continued from previous page)

```
    // Put methods for controlling this subsystem
    // here. Call these from Commands.

    public void drive(double left, double right) {
        drive.tankDrive(left, right);
    }

    public void stop() {
        drive.tankDrive(0.0, 0.0);
    }

}
```

C++ (Header)

```
// ROBOTBUILDER TYPE: Subsystem.
#pragma once

// BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=INCLUDES
#include <frc2/command/SubsystemBase.h>
#include <frc/AnalogGyro.h>
#include <frc/AnalogInput.h>
#include <frc/Encoder.h>
#include <frc/drive/DifferentialDrive.h>
#include <frc/motorcontrol/PWMVictorSPX.h>

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=INCLUDES

/**
 *
 *
 * @author ExampleAuthor
 */
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::AnalogInput m_rangefinder{1};
frc::AnalogGyro m_gyro{0};
frc::Encoder m_rightencoder{2, 3, false, frc::Encoder::k4X};
frc::Encoder m_leftencoder{0, 1, false, frc::Encoder::k4X};
frc::PWMVictorSPX m_right2{3};
frc::PWMVictorSPX m_left2{2};
frc::DifferentialDrive m_drive{m_left1, m_right1};
frc::PWMVictorSPX m_right1{1};
frc::PWMVictorSPX m_left1{0};

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
public:
DriveTrain();

    void Periodic() override;
    void SimulationPeriodic() override;
```

(continues on next page)

(continued from previous page)

```
    void Drive(double left, double right);
    void Stop();
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CMDPIDGETTERS

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CMDPIDGETTERS
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTANTS
static constexpr const double PlaceDistance = 0.1;
static constexpr const double BackAwayDistance = 0.6;

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTANTS


};
```

C++ (Source)

```
// ROBOTBUILDER TYPE: Subsystem.

// BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=INCLUDES
#include "subsystems/DriveTrain.h"
#include <frc/smartdashboard/SmartDashboard.h>

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=INCLUDES

DriveTrain::DriveTrain(){
    SetName("Drive Train");
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
    SetSubsystem("Drive Train");

 AddChild("range finder", &m_rangefinder);


 AddChild("gyro", &m_gyro);
 m_gyro.SetSensitivity(0.007);

 AddChild("right encoder", &m_rightencoder);
 m_rightencoder.SetDistancePerPulse(1.0);

 AddChild("left encoder", &m_leftencoder);
 m_leftencoder.SetDistancePerPulse(1.0);

 AddChild("right2", &m_right2);
 m_right2.SetInverted(false);
m_right1.AddFollower(m_right2);

 AddChild("left2", &m_left2);
 m_left2.SetInverted(false);
m_left1.AddFollower(m_left2);

 AddChild("Drive", &m_drive);
 m_drive.SetSafetyEnabled(true);
m_drive.SetExpiration(0.1_s);
m_drive.SetMaxOutput(1.0);

```

(continues on next page)

(continued from previous page)

```
 AddChild("right1", &m_right1);
 m_right1.SetInverted(true);


 AddChild("left1", &m_left1);
 m_left1.SetInverted(false);


    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
}

void DriveTrain::Periodic() {
    // Put code here to be run every loop

}

void DriveTrain::SimulationPeriodic() {
    // This method will be called once per scheduler run when in simulation

}

// BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CMDPIDGETTERS

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CMDPIDGETTERS

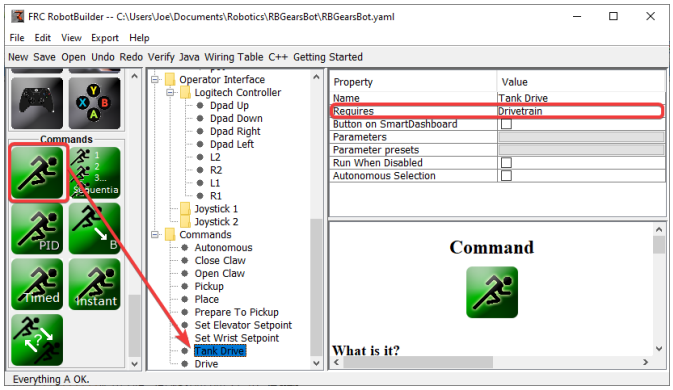

// Put methods for controlling this subsystem
// here. Call these from Commands.

void DriveTrain::Drive(double left, double right) {
    m_drive.TankDrive(left, right);
}

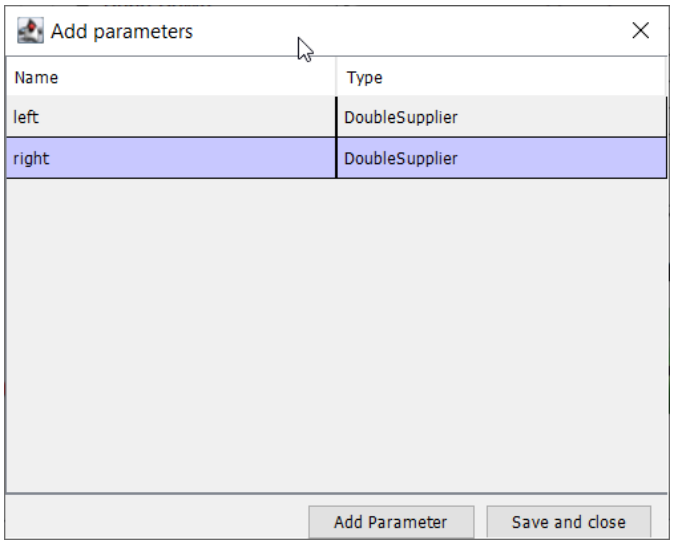
void DriveTrain::Stop() {
    m_drive.TankDrive(0.0, 0.0);
}
```

Create a method that takes the joystick inputs, in this case the left and right driver joystick. The values are passed to the DifferentialDrive 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.

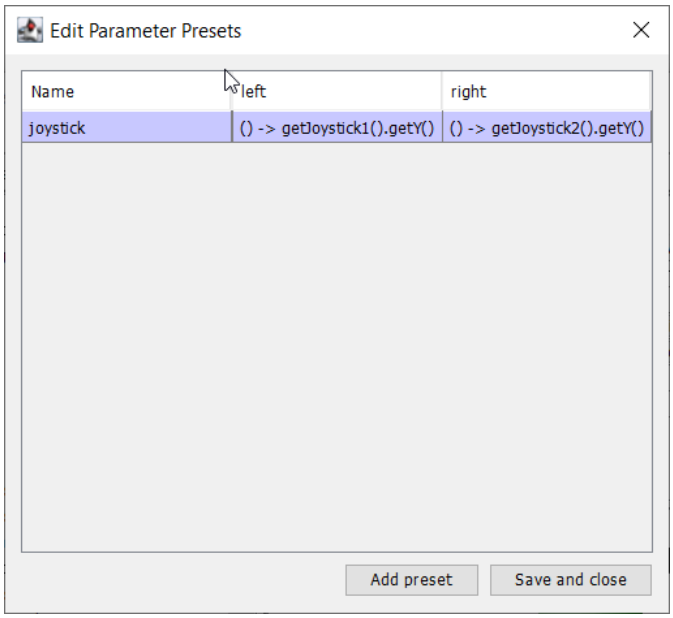
Read Joystick Values and Call the Subsystem Methods



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.



Create two parameters (`DoubleSupplier` for Java or `std::function<double()>` for C++) for the left and right speeds.



Create a parameter preset to retrieve joystick values. Java: For the left parameter enter `() -> getJoystick1().getY()` and for right enter `() -> getJoystick2().getY()`. C++: For the left parameter enter `[this] {return getJoystick1()->GetY();}` and for the right enter `[this] {return getJoystick2()->GetY();}`

> **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

```
// ROBOTBUILDER TYPE: Command.

package frc.robot.commands;
import edu.wpi.first.wpilibj2.command.Command;
import java.util.function.DoubleSupplier;

// BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=IMPORTS
import frc.robot.subsystems.DriveTrain;

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=IMPORTS

/**
 *
 */
public class TankDrive extends Command {

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=VARIABLE_DECLARATIONS
        private final DriveTrain m_driveTrain;
    private DoubleSupplier m_left;
```

(continues on next page)

(continued from previous page)

```
    private DoubleSupplier m_right;

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=VARIABLE_DECLARATIONS

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTORS


    public TankDrive(DoubleSupplier left, DoubleSupplier right, DriveTrain subsystem)
↪{

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTORS
        // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=VARIABLE_SETTING
        m_left = left;
        m_right = right;

    // 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(m_left.getAsDouble(), m_right.getAsDouble());
    }

    // Called once the command ends or is interrupted.
    @Override
    public void end(boolean interrupted) {
        m_driveTrain.stop();
    }

    // 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
    }
}
```

C++ (Header)

```
// ROBOTBUILDER TYPE: Command.

#pragma once

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=INCLUDES

#include <frc2/command/CommandHelper.h>
#include <frc2/command/Command.h>

#include "subsystems/DriveTrain.h"

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=INCLUDES

/**
 *
 *
 * @author ExampleAuthor
 */
class TankDrive: public frc2::CommandHelper<frc2::Command, TankDrive> {
public:
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTOR
    explicit TankDrive(std::function<double()> left, std::function<double()> right,␣
↪DriveTrain* m_drivetrain);

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTOR

void Initialize() override;
void Execute() override;
bool IsFinished() override;
void End(bool interrupted) override;
bool RunsWhenDisabled() const override;


private:
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=VARIABLES

std::function<double()> m_left;
std::function<double()> m_right;

DriveTrain* m_drivetrain;


    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=VARIABLES
};
```

C++ (Source)

```cpp
// ROBOTBUILDER TYPE: Command.

// BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTOR

#include "commands/TankDrive.h"

TankDrive::TankDrive(std::function<double()> left, std::function<double()> right,␣
↪DriveTrain* m_drivetrain) :
    m_left(left),
    m_right(right),
m_drivetrain(m_drivetrain)
{

    // Use AddRequirements() here to declare subsystem dependencies
    // eg. AddRequirements(m_Subsystem);
    SetName("TankDrive");
    AddRequirements({m_drivetrain});

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTRUCTOR

}

// Called just before this Command runs the first time
void TankDrive::Initialize() {

}

// Called repeatedly when this Command is scheduled to run
void TankDrive::Execute() {
    m_drivetrain->Drive(m_left(),m_right());
}

// Make this return true when this Command no longer needs to run execute()
bool TankDrive::IsFinished() {
    return false;
}

// Called once after isFinished returns true
void TankDrive::End(bool interrupted) {
    m_drivetrain->Drive(0,0);
}

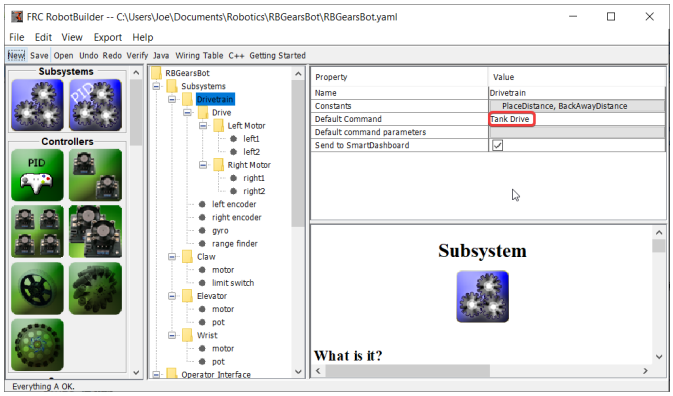
bool TankDrive::RunsWhenDisabled() const {
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DISABLED
    return false;

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DISABLED
}
```

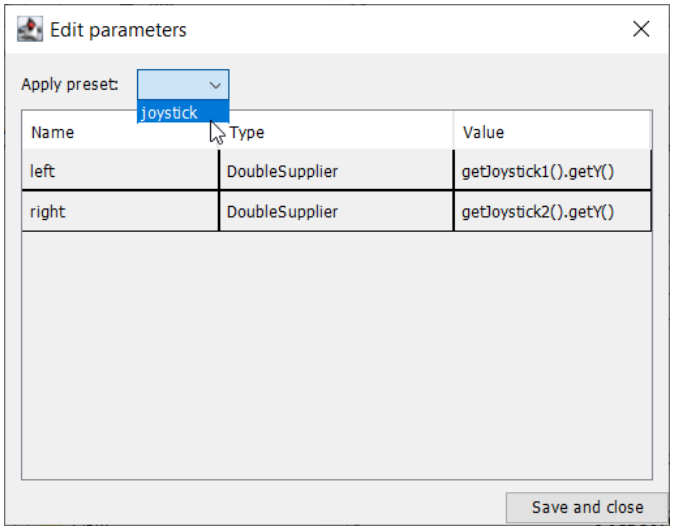
Add code to the execute method to do the actual driving. All that is needed is pass the for the left and right parameters to the Drive Train subsystem. The subsystem just uses them for the tank steering method on its DifferentialDrive object. And we get tank steering.

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.



The final step is to choose the joystick parameter preset previously set up.

Note

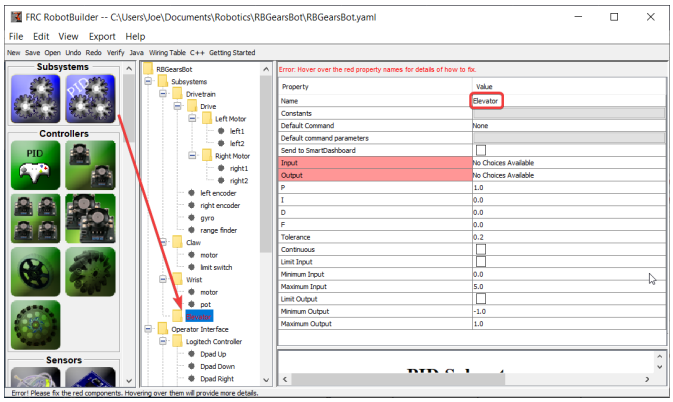
Be sure to export your program to C++ or Java before continuing.

21.3 RobotBuilder - Advanced

21.3.1 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. 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 PIDSubsystem has a built-in PIDController to automatically control the mechanism to the correct setpoints.

Create a PIDSubsystem

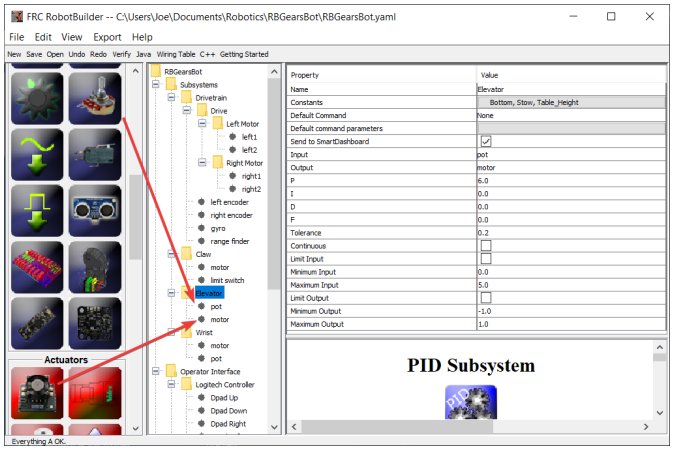


Creating a subsystem that uses feedback to control the position or speed of a mechanism is very easy.

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.

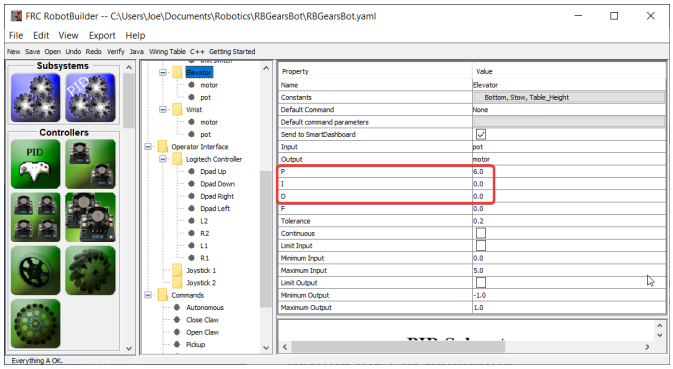
Adding Sensors and Actuators to the PIDSubsystem



Add the missing components for the PIDSubsystem

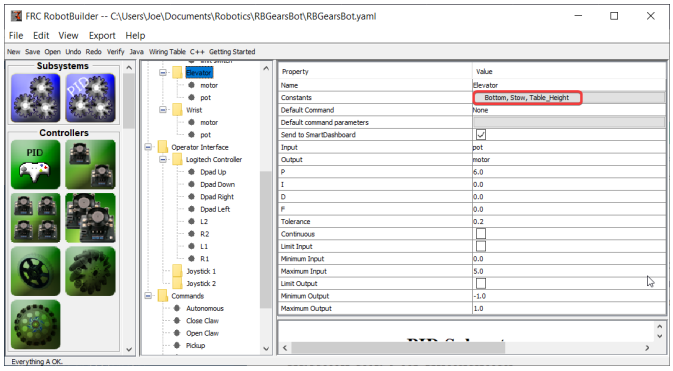
1. Drag in the actuator (a motor controller) to the particular subsystem - in this case the Elevator
2. Drag the sensor that will be used for feedback to the subsystem, in this case the sensor is a potentiometer that might give elevator height feedback.

Fill in the PID Parameters

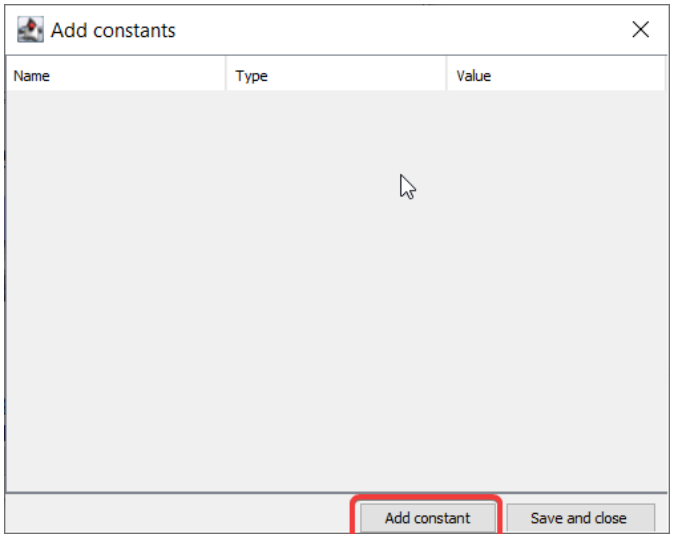


The P, I, and D values need to be filled in to get the desired sensitivity and stability of the component. In the case of our elevator we use a proportional constant of 6.0 and 0 for the I and D terms.

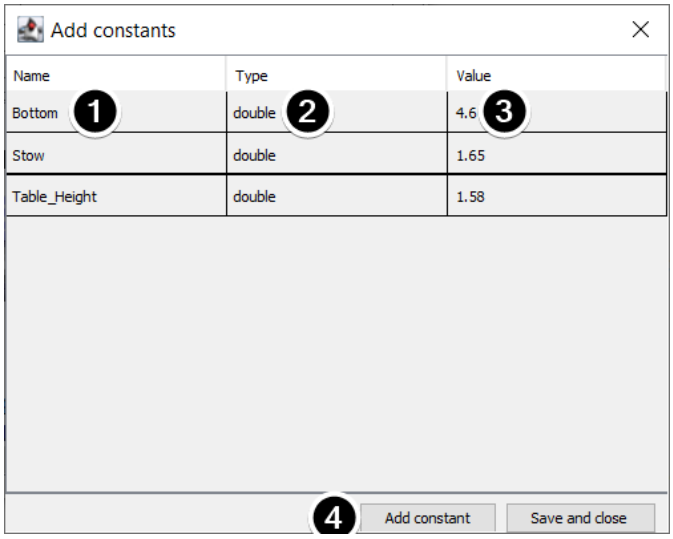
Create Setpoint Constants



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



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
5. After entering all constants, Click *Save and close*

21.3.2 Writing the Code for a PIDSubsystem

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.

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.

RobotBuilder generates the PIDSubsystem methods such that no additional code is needed for basic operation.

Setting the PID Constants

The height constants and PID constants are automatically generated.

JAVA

```
public class Elevator extends PIDSubsystem {

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTANTS
public static final double Bottom = 4.6;
public static final double Stow = 1.65;
public static final double Table_Height = 1.58;
    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTANTS

    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
private AnalogPotentiometer pot;private PWMVictorSPX motor;
    //P I D Variables
    private static final double kP = 6.0;
    private static final double kI = 0.0;
    private static final double kD = 0.0;
    private static final double kF = 0.0;
    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=DECLARATIONS
```

C++

```
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTANTS
static constexpr const double Bottom = 4.6;
static constexpr const double Stow = 1.65;
static constexpr const double Table_Height = 1.58;
    static constexpr const double kP = 6.0;
    static constexpr const double kI = 0.0;
    static constexpr const double kD = 0.0;
    static constexpr const double kF = 0.0;
    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=CONSTANTS
```

Get Potentiometer Measurement

JAVA

```
@Override
public double getMeasurement() {
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=SOURCE
    return pot.get();

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=SOURCE
}
```

C++

```
double Elevator::GetMeasurement() {
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=SOURCE
    return m_pot.Get();

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=SOURCE
}
```

The `getMeasurement()` 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 voltage as returned by the `get()` method.

Calculate PID Output

JAVA

```
    @Override
    public void useOutput(double output, double setpoint) {
        output += setpoint*kF;
        // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=OUTPUT
motor.set(output);

        // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=OUTPUT
    }
```

C++

```
void Elevator::UseOutput(double output, double setpoint) {
    output += setpoint*kF;
    // BEGIN AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=OUTPUT
m_motor.Set(output);

    // END AUTOGENERATED CODE, SOURCE=ROBOTBUILDER ID=OUTPUT
}
```

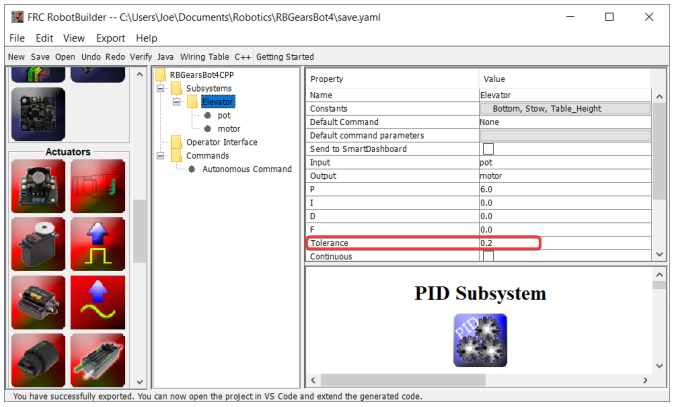
The `useOutput` method writes the calculated PID output directly to the motor.

That's all that is required to create the Elevator PIDSubsystem.

21.3.3 Setpoint Command

A Setpoint Command works in conjunction with a PIDSubsystem 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.

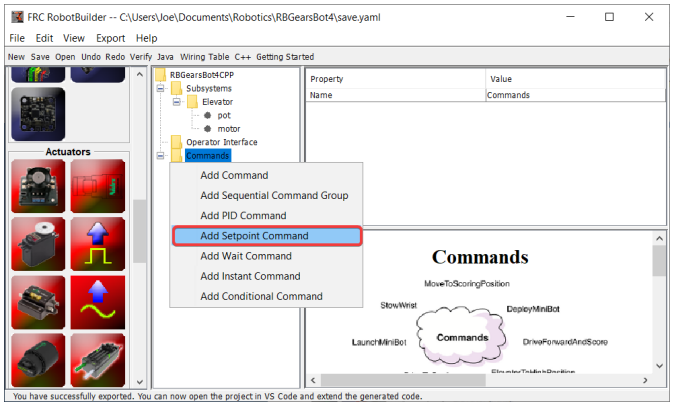
Start with a PIDSubsystem



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.

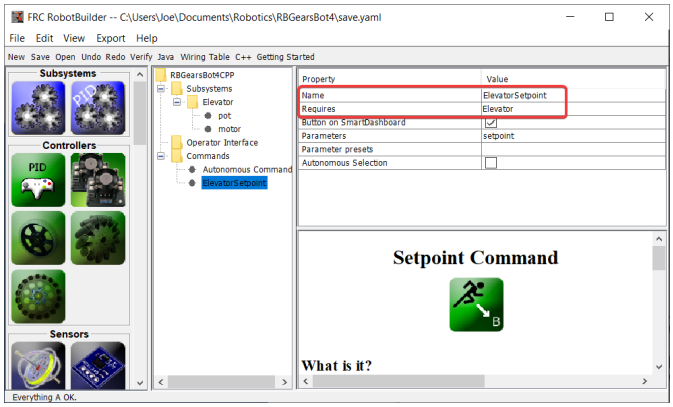
It is important to set the **Tolerance** parameter. This controls how far off the current value can be from the setpoint and be considered on target. This is the criteria that the SetpointCommand uses to move onto the next command.

Creating the Setpoint Command

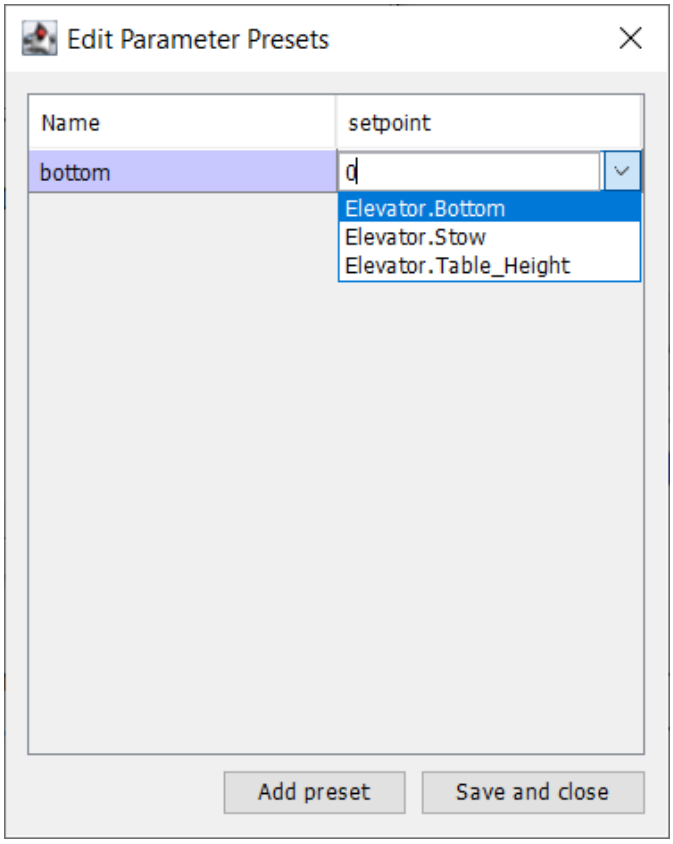


Right-click on the Commands folder in the palette and select “Add Setpoint command”.

Setpoint Command Parameters



Fill in the name of the new command. The Requires field is the PIDSubsystem that is being driven to a setpoint, in this case the Elevator subsystem.



1. Click on the Parameter Presets to set up the setpoints.
2. Select *Add Preset*
3. Enter a preset name (in this case 'bottom'
4. Click the dropdown next to the setpoint entry box
5. Select the Elevator.Bottom constant, that was created in the Elevator subsystem previously
6. Repeat steps 2-5 for the other setpoints.
7. Click *Save and close*

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.

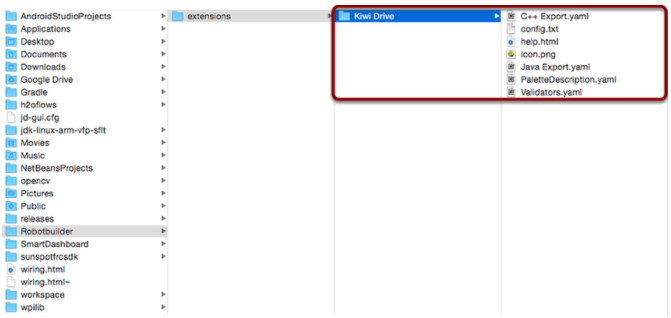
Note

For more information about PID Control, please see the *Advanced Controls Introduction*.

21.3.4 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

PaletteDescription.yaml | Validators.yaml | C++ Export.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.
```

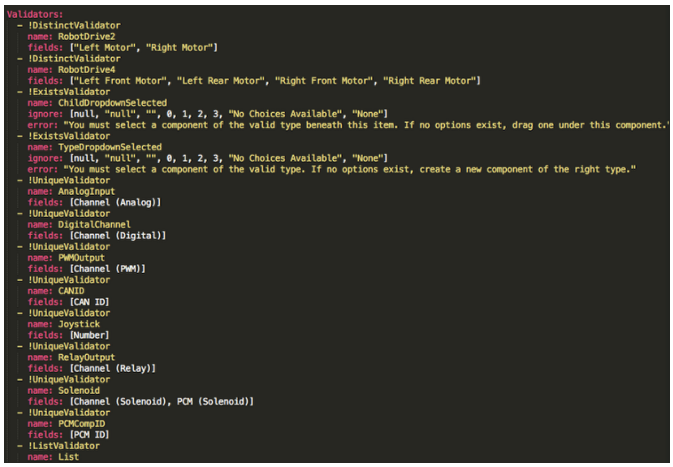
Validators.yaml

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



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



A line-by-line breakdown of the file:

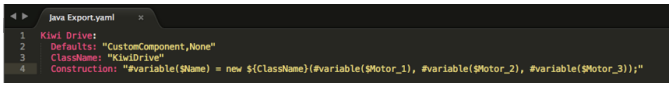
- Kiwi Drive: the name of the component being exported. This is the same as the name set in PaletteDescription.yaml, and the name of the folder containing this file
- Defaults: provides some default values for includes needed by this component, the name of the class, a construction template, and more. The CustomComponent default adds an include for `Custom/${ClassName}.h` to every generated file that uses the component (e.g. `RobotDrive.h` would have `#include "Custom/KiwiDrive.h` the top of the file)
- ClassName: the name of the custom class you're adding.
- Construction: an instruction for how the component should be constructed. Variables will be replaced with their values ("${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



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`).

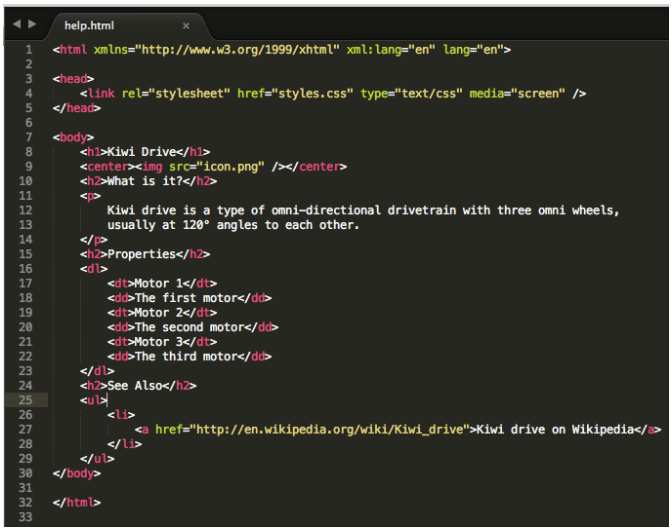
Variables are either component properties (e.g. "Motor 1", "Motor 2", "Motor 3" in the kiwi drive example), or one of the following:

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`)

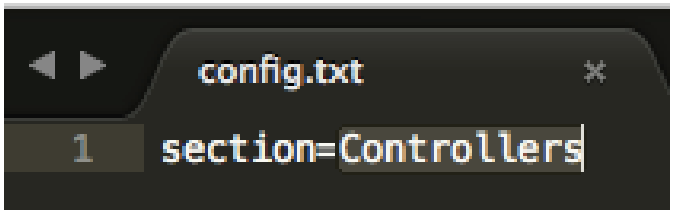
If you have variables with spaces in the name (such as “Motor 1”, “Right Front Motor”, etc.), the spaces need to be replaced with underscores when using them in the export files.

help.html



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



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.

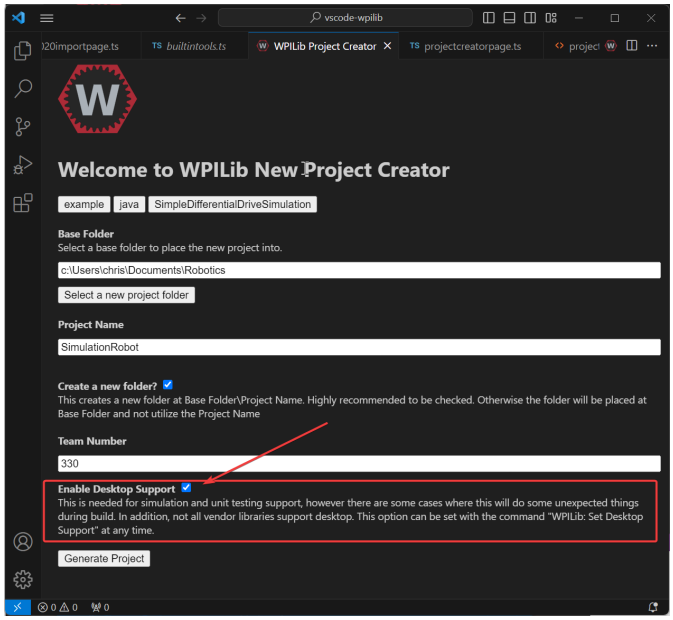
22 Robot Simulation

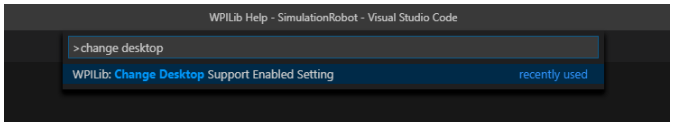
22.1 Introduction to Robot Simulation

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.

Java/C++

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`

Important

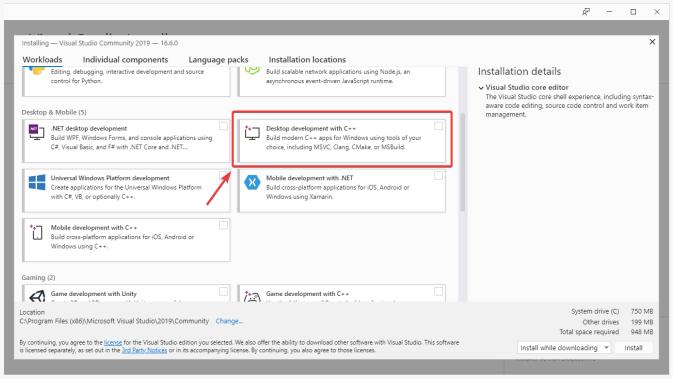
It is important to note that enabling desktop/simulation support can have unintended consequences. Not all vendors will support this option, and code that uses their libraries may even crash when attempting to run simulation!

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 or change `includeDesktopSupport` to false in build.gradle.

Note

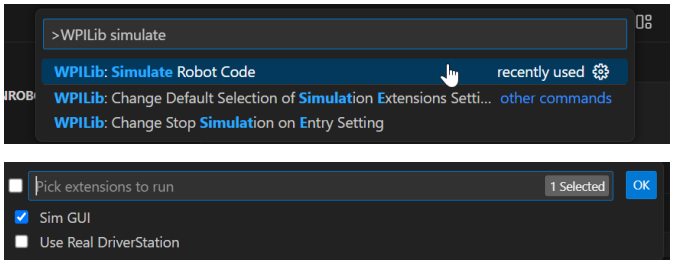
C++ robot simulation requires that a native compiler to be installed. For Windows, this would be Visual Studio 2022 version 17.9 or later (**not** VS Code), macOS requires Xcode 14 or later, and Linux (Ubuntu) requires the `build-essential` package.

Ensure the *Desktop Development with C++* option is checked in the Visual Studio installer for simulation support.



Running Robot Simulation

Basic robot simulation can be run using VS Code. This can be done by using VS Code's command palette *WPILib: Simulate Robot Code* as shown below.



The *Sim GUI* option will be selected by default. This will launch the *Simulation GUI*. You can also launch simulation without a GUI by unchecking *Sim GUI* in which case your console output in Visual Studio Code should look like the below.

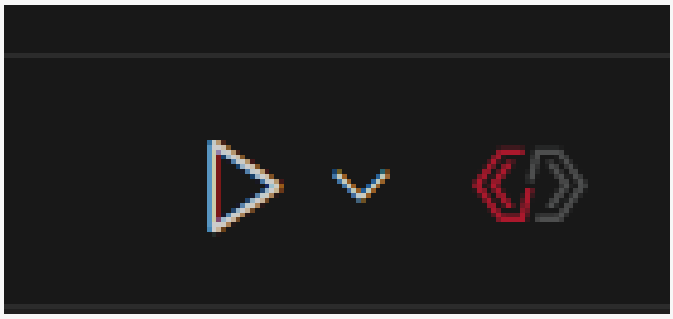
```
********** Robot program starting **********
Default disabledInit() method... Override me!
Default disabledPeriodic() method... Override me!
Default robotPeriodic() method... Override me!
```

If you would would like to prevent the pop up and only use build.gradle to configure your simulation settings you can add the following in your vscode settings.json file.

```
"wpilib.skipSelectSimulateExtension": true
```

Warning

You may see a run button next to the WPILib button. This button does not set up simulation appropriately and should not be used. Instead, the menu item shown above *WPILib: Simulate Robot Code* should be used.



Note

Simulation can also be run outside of VS Code using `./gradlew simulateJava` for Java or `./gradlew simulateNative` for C++.

Note

Some vendors support attaching hardware to your PC and using the hardware in desktop simulation (e.g. CANivore). See *vendor documentation* for more information about the command *WPILib: Hardware Sim Robot Code*.

Python

GUI simulation support is installed by default when you install RobotPy.

There is a `robotpy` subcommand that you can execute to run your code in simulation:

Windows

```
py -3 -m robotpy sim
```

macOS

```
python3 -m robotpy sim
```

Linux

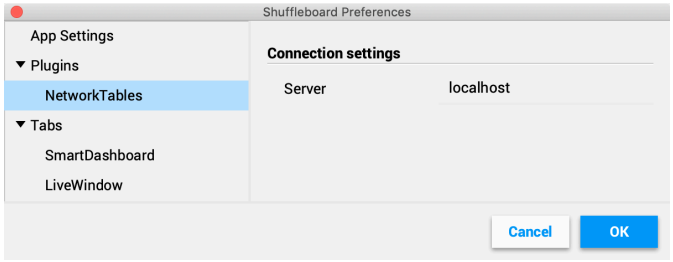
```
python3 -m robotpy sim
```

22.1.1 Running Robot Dashboards

Shuffleboard, SmartDashboard, Glass, and AdvantageScope can be used with WPILib simulation when they are configured to connect to the local computer (i.e. `localhost`).

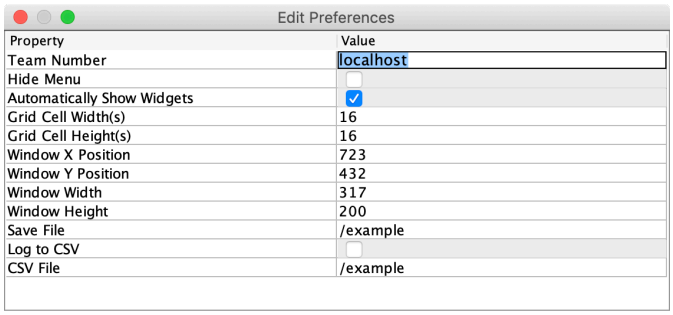
Shuffleboard

Shuffleboard is automatically configured to look for a NetworkTables instance from the robotRIO but **not from other sources**. To connect to a simulation, 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`.



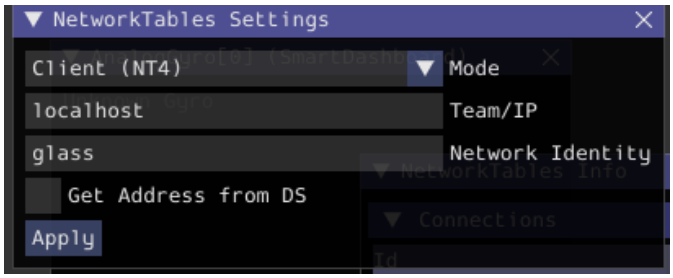
SmartDashboard

SmartDashboard is automatically configured to look for a NetworkTables instance from the roboRIO, but **not from other sources**. To connect to a simulation, 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`.



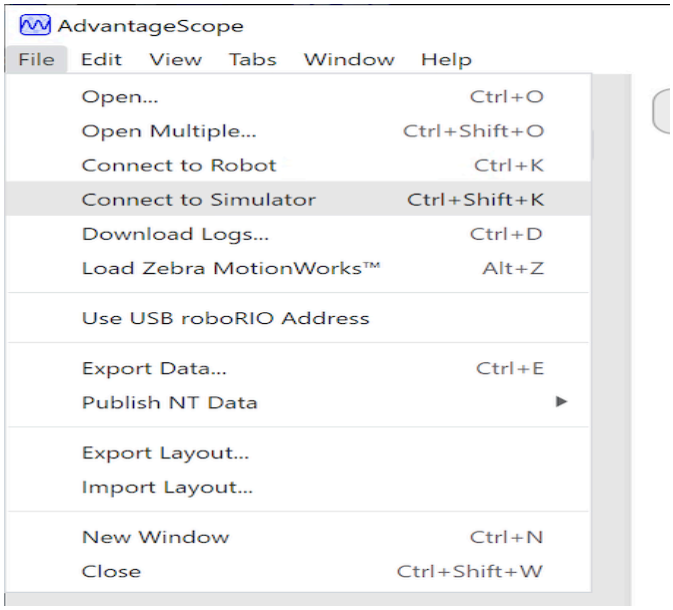
Glass

Glass is automatically configured to look for a NetworkTables instance from the roboRIO, but **not from other sources**. To connect to a simulation, open *NetworkTables Settings* under the *NetworkTables* menu and in the *Team/IP* field, enter the IP address or hostname of the NetworkTables host. For a standard simulation configuration, use `localhost`.



AdvantageScope

No configuration is required to connect to a NetworkTables instance running on the local computer. To connect to a simulation, click *Connect to Simulator* under the *File* menu or press `Ctrl+Shift+K`.

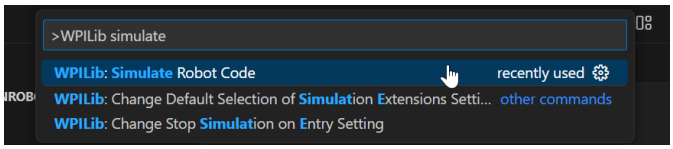


22.2 Simulation Specific User Interface Elements

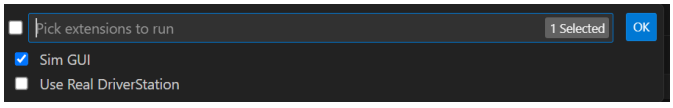
WPILib has extended robot simulation to introduce a graphical user interface (GUI) component. This allows teams to easily visualize their robot's inputs and outputs.

> **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.

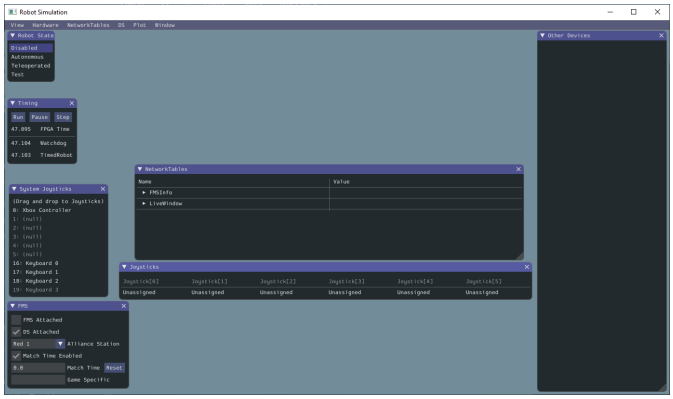
22.2.1 Running the GUI



You can simply launch the GUI via the *Simulate Robot Code* command palette option.

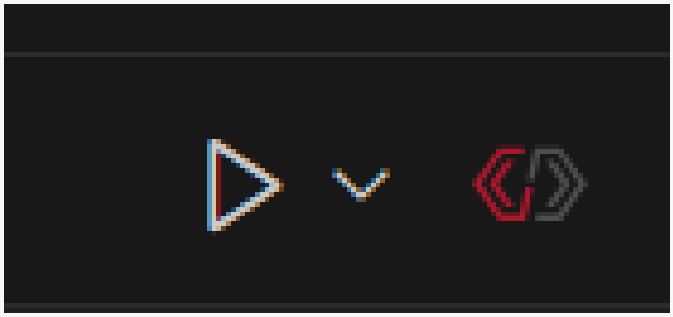


And the *Sim GUI* option should popup in a new dialog and will be selected by default. Press *Ok*. This will now launch the Simulation GUI.



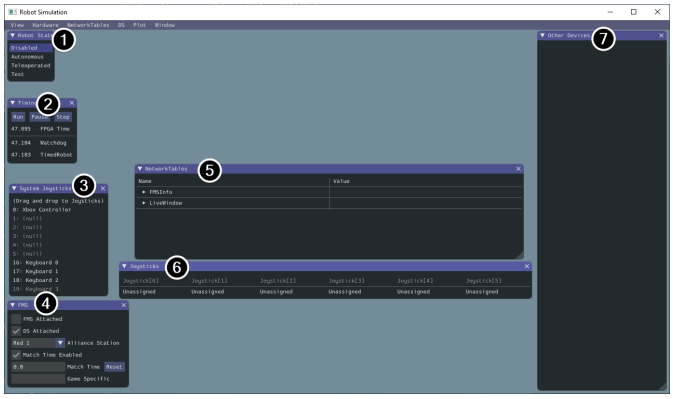
Warning

You may see a run button next to the WPILib button. This button does not set up simulation appropriately and should not be used. Instead, the menu item shown above *WPILib: Simulate Robot Code* should be used.



22.2.2 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.

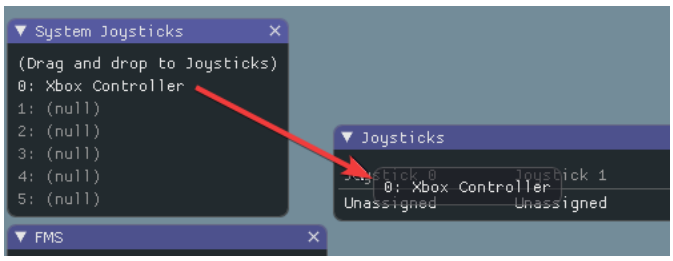
3. **System Joysticks** - This is a list of joysticks connected to your system currently.
4. **FMS** - This is used for simulating many of the common *FMS* systems.
5. **NetworkTables** - This shows the data that has been published to NetworkTables.
6. **Joysticks** - This is joysticks that the robot 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".

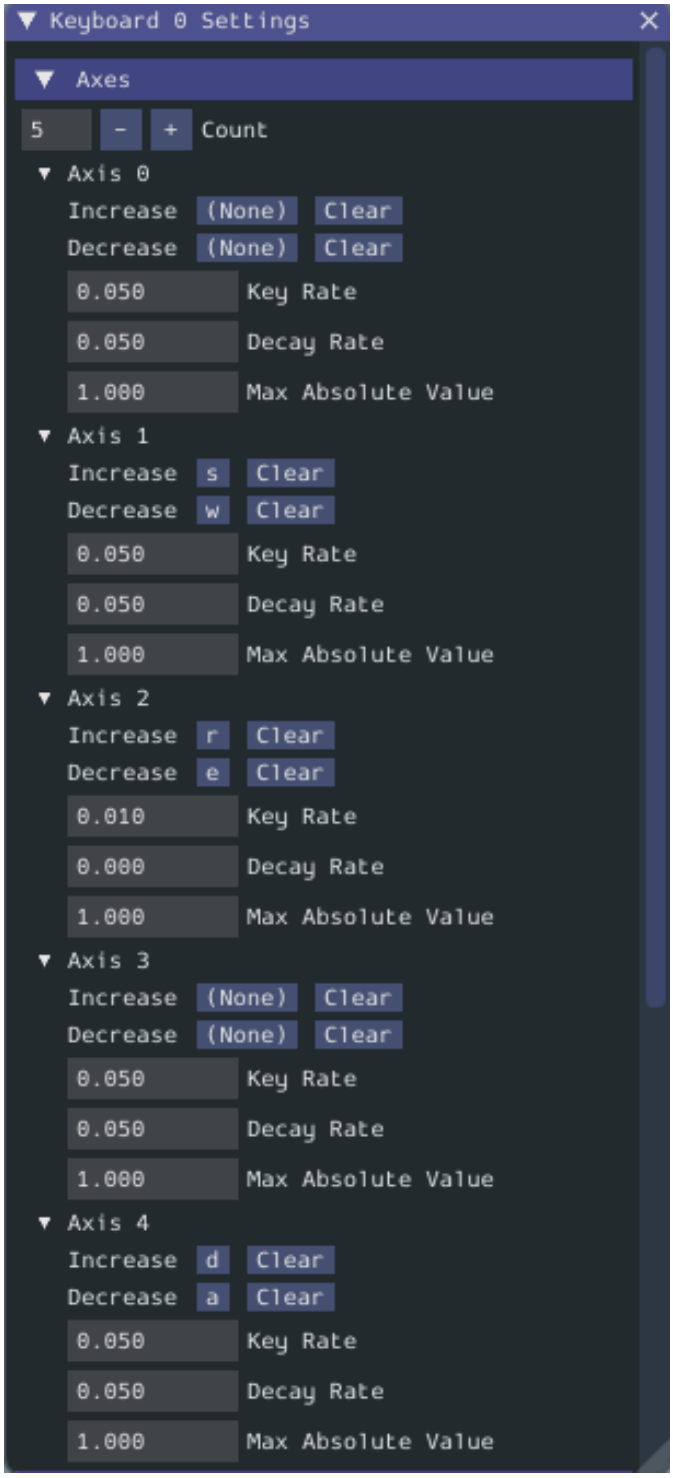


Note

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.

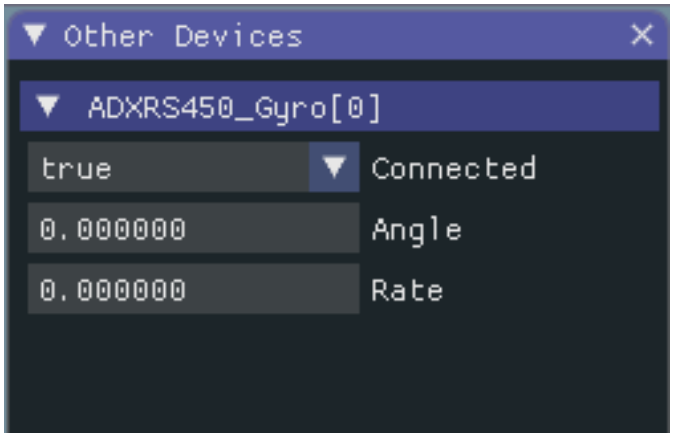
Using the Keyboard as a Joystick

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 and use on-the-fly.



22.2.3 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);
 }
}
```

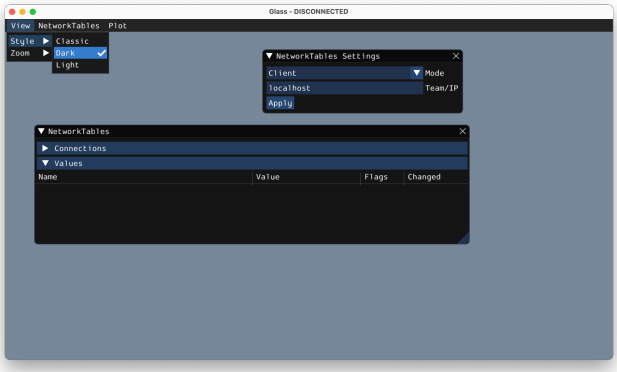
Note

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.

22.2.4 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:



22.2.5 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".

22.3 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

22.3.1 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 in the direction of elevator motion
- SingleJointedArmSim, which models gravity proportional to the arm angle
- 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.

Note

Swerve support for simulation is in the works, but we cannot provide an ETA. For updates on progress, please follow this pull request.

22.3.2 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), the starting height of the elevator, 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

```
  // Simulation classes help us simulate what's going on, including gravity.
  private final ElevatorSim m_elevatorSim =
      new ElevatorSim(
          m_elevatorGearbox,
          Constants.kElevatorGearing,
          Constants.kCarriageMass,
          Constants.kElevatorDrumRadius,
          Constants.kMinElevatorHeightMeters,
          Constants.kMaxElevatorHeightMeters,
          true,
          0,
          0.01,
          0.0);
```

C++

```
  // Simulation classes help us simulate what's going on, including gravity.
  frc::sim::ElevatorSim m_elevatorSim{m_elevatorGearbox,
                                      Constants::kElevatorGearing,
                                      Constants::kCarriageMass,
                                      Constants::kElevatorDrumRadius,
                                      Constants::kMinElevatorHeight,
                                      Constants::kMaxElevatorHeight,
                                      true,
                                      0_m,
                                      {0.01}};
  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

```
  @Override
  public void teleopPeriodic() {
    if (m_joystick.getTrigger()) {
      // Here, we set the constant setpoint of 0.75 meters.
      m_elevator.reachGoal(Constants.kSetpointMeters);
    } else {
      // Otherwise, we update the setpoint to 0.
      m_elevator.reachGoal(0.0);
    }
  }
```

```
  public void reachGoal(double goal) {
    m_controller.setGoal(goal);

    // With the setpoint value we run PID control like normal
    double pidOutput = m_controller.calculate(m_encoder.getDistance());
    double feedforwardOutput = m_feedforward.calculate(m_controller.getSetpoint().
↪velocity);
```

(continues on next page)

(continued from previous page)

```
    m_motor.setVoltage(pidOutput + feedforwardOutput);
  }
```

C++

```
void Robot::TeleopPeriodic() {
  if (m_joystick.GetTrigger()) {
    // Here, we set the constant setpoint of 0.75 meters.
    m_elevator.ReachGoal(Constants::kSetpoint);
  } else {
    // Otherwise, we update the setpoint to 0.
    m_elevator.ReachGoal(0.0_m);
  }
}
```

```
void Elevator::ReachGoal(units::meter_t goal) {
  m_controller.SetGoal(goal);
  // With the setpoint value we run PID control like normal
  double pidOutput =
      m_controller.Calculate(units::meter_t{m_encoder.GetDistance()});
  units::volt_t feedforwardOutput =
      m_feedforward.Calculate(m_controller.GetSetpoint().velocity);
  m_motor.SetVoltage(units::volt_t{pidOutput} + feedforwardOutput);
}
```

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.

> **Note**
>
> Classes inheriting from command-based's `Subsystem` can override the inherited `simulationPeriodic()` method. Other classes will need their simulation update methods called from `Robot`'s `simulationPeriodic`.

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

```
  public void simulationPeriodic() {
    // In this method, we update our simulation of what our elevator is doing
    // First, we set our "inputs" (voltages)
    m_elevatorSim.setInput(m_motorSim.getSpeed() * RobotController.
↪getBatteryVoltage());

    // Next, we update it. The standard loop time is 20ms.
    m_elevatorSim.update(0.020);

```

(continues on next page)

(continued from previous page)

```
    // Finally, we set our simulated encoder's readings and simulated battery voltage
    m_encoderSim.setDistance(m_elevatorSim.getPositionMeters());
    // SimBattery estimates loaded battery voltages
    RoboRioSim.setVInVoltage(
        BatterySim.calculateDefaultBatteryLoadedVoltage(m_elevatorSim.
↪getCurrentDrawAmps()));
  }
```

C++

```
void Elevator::SimulationPeriodic() {
  // In this method, we update our simulation of what our elevator is doing
  // First, we set our "inputs" (voltages)
  m_elevatorSim.SetInput(frc::Vectord<1>{
      m_motorSim.GetSpeed() * frc::RobotController::GetInputVoltage()});

  // Next, we update it. The standard loop time is 20ms.
  m_elevatorSim.Update(20_ms);

  // Finally, we set our simulated encoder's readings and simulated battery
  // voltage
  m_encoderSim.SetDistance(m_elevatorSim.GetPosition().value());
  // SimBattery estimates loaded battery voltages
  frc::sim::RoboRioSim::SetVInVoltage(
      frc::sim::BatterySim::Calculate({m_elevatorSim.GetCurrentDraw()}));
}
```

22.4 Device Simulation

WPILib provides a way to manage simulation device data in the form of the SimDevice API.

22.4.1 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

```
// 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);
```

C++

```
// 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

```
simEncoder.setCount(100);
encoder.getCount(); // 100
simEncoder.getCount(); // 100
```

C++

```
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

```
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
```

C++

```
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
```

22.4.2 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

```
SimDeviceSim device = new SimDeviceSim(deviceKey, index);
```

C++

```
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

```
SimDouble field = device.getDouble(fieldKey);
field.get();
field.set(value);
```

C++

```
hal::SimDouble field = device.GetDouble(fieldKey);
field.Get();
field.Set(value);
```

22.5 Drivetrain Simulation Tutorial

This is a tutorial for implementing a simulation model of your differential drivetrain using the 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.

22.5.1 Drivetrain Simulation Overview

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.



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.

22.5.2 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. See *the Device Simulation article* for more info about these simulation hardware classes and simulation of vendor devices.

> **Note**
>
> Simulation objects associated with a particular subsystem should live in that subsystem. An example of this is in the `StateSpaceDriveSimulation` (Java, C++) example.

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

```
// 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);
```

C++

```
#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};
```

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

```
// 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);
```

C++

```
#include <frc/AnalogGyro.h>
#include <frc/simulation/AnalogGyroSim.h>
...
// Create our gyro objectl ike 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};
```

22.5.3 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.

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

```
// 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));
```

C++

```
#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.
```

(continues on next page)

(continued from previous page)

```
  // 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
  {0.001, 0.001, 0.001, 0.1, 0.1, 0.005, 0.005}};
```

Creating a `DifferentialDrivetrainSim` from SysId Gains

You can also use the gains produced by *System Identification*, 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 identification 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 identification 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

```
// Create our feedforward gain constants (from the identification
// 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 identification 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));
```

C++

```
#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 identification
// 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;
// The track width is 0.7112 meters.
static constexpr auto kTrackwidth = 0.7112_m;
// Create the simulation model of our drivetrain.
frc::sim::DifferentialDrivetrainSim m_driveSim{
  // Create a linear system from our identification gains.
  frc::LinearSystemId::IdentifyDrivetrainSystem(
    KvLinear, KaLinear, KvAngular, KaAngular, kTrackWidth),
  kTrackWidth,
```

(continues on next page)

(continued from previous page)

```
  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
  // l and r position: 0.005 m
  {0.001, 0.001, 0.001, 0.1, 0.1, 0.005, 0.005}};
```

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

```
private DifferentialDrivetrainSim m_driveSim = DifferentialDrivetrainSim.
↪createKitbotSim(
  KitbotMotor.kDualCIMPerSide, // 2 CIMs per side.
  KitbotGearing.k10p71,        // 10.71:1
  KitbotWheelSize.kSixInch,    // 6" diameter wheels.
  null                         // No measurement noise.
);
```

C++

```
#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.

22.5.4 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

```
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());
}
```

C++

```
frc::PWMSparkMax m_leftMotor{0};
frc::PWMSparkMax m_rightMotor{1};
Drivetrain() {
  ...
  m_leftEncoder.SetDistancePerPulse(2 * std::numbers::pi * kWheelRadius /␣
↪kEncoderResolution);
  m_rightEncoder.SetDistancePerPulse(2 * std::numbers::pi * kWheelRadius /␣
↪kEncoderResolution);
}
```

(continues on next page)

(continued from previous page)

```
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().value());
  m_leftEncoderSim.SetRate(m_driveSim.GetLeftVelocity().value());
  m_rightEncoderSim.SetDistance(m_driveSim.GetRightPosition().value());
  m_rightEncoderSim.SetRate(m_driveSim.GetRightVelocity().value());
  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.

22.5.5 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*.

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

```
private Field2d m_field = new Field2d();
```

C++

```
#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

```
public Drivetrain() {
  ...
  SmartDashboard.putData("Field", m_field);
}
```

C++

```
#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

```
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());
}
```

C++

```
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 the *Field2d Widget page* to add the field that your simulated robot will run on to the GUI.

22.6 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 5 available by default, and C++ robot projects have Google Test.

22.6.1 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

```
import edu.wpi.first.wpilibj.DoubleSolenoid;
import edu.wpi.first.wpilibj.PneumaticsModuleType;
import edu.wpi.first.wpilibj.examples.unittest.Constants.IntakeConstants;
import edu.wpi.first.wpilibj.motorcontrol.PWMSparkMax;

public class Intake implements AutoCloseable {
  private final PWMSparkMax m_motor;
  private final DoubleSolenoid m_piston;

  public Intake() {
    m_motor = new PWMSparkMax(IntakeConstants.kMotorPort);
    m_piston =
        new DoubleSolenoid(
            PneumaticsModuleType.CTREPCM,
            IntakeConstants.kPistonFwdChannel,
            IntakeConstants.kPistonRevChannel);
  }

  public void deploy() {
    m_piston.set(DoubleSolenoid.Value.kForward);
  }

  public void retract() {
    m_piston.set(DoubleSolenoid.Value.kReverse);
    m_motor.set(0); // turn off the motor
  }
```

(continues on next page)

(continued from previous page)

```
  public void activate(double speed) {
    if (isDeployed()) {
      m_motor.set(speed);
    } else { // if piston isn't open, do nothing
      m_motor.set(0);
    }
  }

  public boolean isDeployed() {
    return m_piston.get() == DoubleSolenoid.Value.kForward;
  }

  @Override
  public void close() {
    m_piston.close();
    m_motor.close();
  }
}
```

C++ (Header)

```
#include <frc/DoubleSolenoid.h>
#include <frc/motorcontrol/PWMSparkMax.h>

#include "Constants.h"

class Intake {
 public:
  void Deploy();
  void Retract();
  void Activate(double speed);
  bool IsDeployed() const;

 private:
  frc::PWMSparkMax m_motor{IntakeConstants::kMotorPort};
  frc::DoubleSolenoid m_piston{frc::PneumaticsModuleType::CTREPCM,
                               IntakeConstants::kPistonFwdChannel,
                               IntakeConstants::kPistonRevChannel};
};
```

C++ (Source)

```
#include "subsystems/Intake.h"

void Intake::Deploy() {
  m_piston.Set(frc::DoubleSolenoid::Value::kForward);
}

void Intake::Retract() {
  m_piston.Set(frc::DoubleSolenoid::Value::kReverse);
  m_motor.Set(0);  // turn off the motor
}
```

(continues on next page)

(continued from previous page)

```
void Intake::Activate(double speed) {
  if (IsDeployed()) {
    m_motor.Set(speed);
  } else {  // if piston isn't open, do nothing
    m_motor.Set(0);
  }
}

bool Intake::IsDeployed() const {
  return m_piston.Get() == frc::DoubleSolenoid::Value::kForward;
}
```

22.6.2 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.jupiter.api.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.jupiter.api.BeforeEach` and `@org.junit.jupiter.api.AfterEach`. 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; 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 `import static org.junit.jupiter.api.Assertions.*`. 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

```
import static org.junit.jupiter.api.Assertions.assertEquals;

import edu.wpi.first.hal.HAL;
import edu.wpi.first.wpilibj.DoubleSolenoid;
import edu.wpi.first.wpilibj.PneumaticsModuleType;
import edu.wpi.first.wpilibj.examples.unittest.Constants.IntakeConstants;
import edu.wpi.first.wpilibj.simulation.DoubleSolenoidSim;
import edu.wpi.first.wpilibj.simulation.PWMSim;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class IntakeTest {
  static final double DELTA = 1e-2; // acceptable deviation range
  Intake m_intake;
  PWMSim m_simMotor;
  DoubleSolenoidSim m_simPiston;

  @BeforeEach // this method will run before each test
  void setup() {
    assert HAL.initialize(500, 0); // initialize the HAL, crash if failed
    m_intake = new Intake(); // create our intake
    m_simMotor =
        new PWMSim(IntakeConstants.kMotorPort); // create our simulation PWM motor␣
↪controller
    m_simPiston =
        new DoubleSolenoidSim(
            PneumaticsModuleType.CTREPCM,
            IntakeConstants.kPistonFwdChannel,
            IntakeConstants.kPistonRevChannel); // create our simulation solenoid
  }

  @SuppressWarnings("PMD.SignatureDeclareThrowsException")
  @AfterEach // this method will run after each test
  void shutdown() throws Exception {
    m_intake.close(); // destroy our intake object
  }

  @Test // marks this method as a test
  void doesntWorkWhenClosed() {
    m_intake.retract(); // close the intake
    m_intake.activate(0.5); // try to activate the motor
    assertEquals(
        0.0, m_simMotor.getSpeed(), DELTA); // make sure that the value set to the␣
↪motor is 0
  }

  @Test
  void worksWhenOpen() {
    m_intake.deploy();
    m_intake.activate(0.5);
    assertEquals(0.5, m_simMotor.getSpeed(), DELTA);
  }

  @Test
```

(continues on next page)

(continued from previous page)

```
  void retractTest() {
    m_intake.retract();
    assertEquals(DoubleSolenoid.Value.kReverse, m_simPiston.get());
  }

  @Test
  void deployTest() {
    m_intake.deploy();
    assertEquals(DoubleSolenoid.Value.kForward, m_simPiston.get());
  }
}
```

C++

```
#include <frc/DoubleSolenoid.h>
#include <frc/simulation/DoubleSolenoidSim.h>
#include <frc/simulation/PWMSim.h>
#include <gtest/gtest.h>

#include "Constants.h"
#include "subsystems/Intake.h"

class IntakeTest : public testing::Test {
 protected:
  Intake intake;  // create our intake
  frc::sim::PWMSim simMotor{
      IntakeConstants::kMotorPort};  // create our simulation PWM
  frc::sim::DoubleSolenoidSim simPiston{
      frc::PneumaticsModuleType::CTREPCM, IntakeConstants::kPistonFwdChannel,
      IntakeConstants::kPistonRevChannel};  // create our simulation solenoid
};

TEST_F(IntakeTest, DoesntWorkWhenClosed) {
  intake.Retract();      // close the intake
  intake.Activate(0.5);  // try to activate the motor
  EXPECT_DOUBLE_EQ(
      0.0,
      simMotor.GetSpeed());  // make sure that the value set to the motor is 0
}

TEST_F(IntakeTest, WorksWhenOpen) {
  intake.Deploy();
  intake.Activate(0.5);
  EXPECT_DOUBLE_EQ(0.5, simMotor.GetSpeed());
}

TEST_F(IntakeTest, Retract) {
  intake.Retract();
  EXPECT_EQ(frc::DoubleSolenoid::Value::kReverse, simPiston.Get());
}

TEST_F(IntakeTest, Deploy) {
  intake.Deploy();
  EXPECT_EQ(frc::DoubleSolenoid::Value::kForward, simPiston.Get());
}
```

For more advanced usage of JUnit and Google Test, see the framework docs.

22.6.3 Running Tests

> **Note**
>
> Tests will always be run in simulation on your desktop. For prerequisites and more info, see *the simulation introduction*.

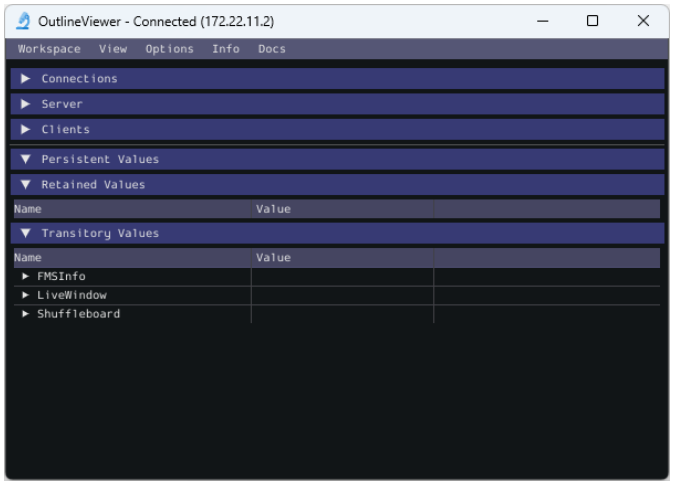
For Java tests to run, make sure that your `build.gradle` file contains the following block:

```
test {
    useJUnitPlatform()
    systemProperty 'junit.jupiter.extensions.autodetection.enabled', 'true'
}
```

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 any failed tests 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.

23 OutlineViewer



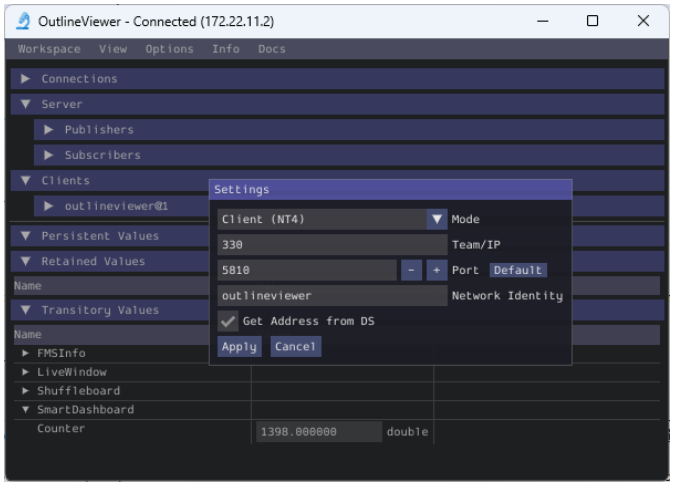
OutlineViewer is a utility used to view, modify, and add to the contents of the NetworkTables for debugging purposes. It displays all key value pairs currently in the NetworkTables and can be used to modify the value of existing keys or add new keys to the table. OutlineViewer is included in the Java and C++ language installations.

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 select *options* then *settings* and set the `Team/IP` to be your team number. After you click *Apply*, OutlineViewer will connect. If you have trouble connecting to OutlineViewer please see the *Dashboard Troubleshooting Steps*.

Note

You can use `localhost` instead of a team number to point OutlineViewer at a simulated robot, Romi, or XRP.

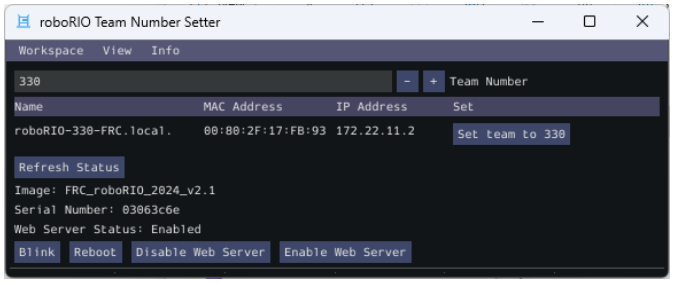


To add additional key/value pairs to NetworkTables, right click on a location and choose the corresponding data type.

Note

LabVIEW teams can use the Variables tab of the LabVIEW Dashboard to accomplish the same functionality as OutlineViewer.

24 roboRIO Team Number Setter



The roboRIO Team Number Setter is a cross-platform utility that can be used to set the team number on the roboRIO. It is an alternative to the roboRIO imaging tool for setting the team number.

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 *roboRIOTeamNumberSetter*.

Note

The desktop shortcut for the roboRIO Team Number Setter doesn't work in WPILib release 2026.1.1. However, the instructions above for starting the tool from VS Code and the start menu shortcut work. See *Known Issues*

Connect to the roboRIO over USB to use the tool, as this is the simplest method when the team number hasn't been set.

24.1 Setting Team Number

Enter your team number in the *Team Number* field and select *Set team to xxxxx*. This will take about a second, then press the *Reboot* button to reboot the roboRIO so the new team number takes effect.

24.2 Enabling/Disabling Webserver

The *roboRIO's webserver* provides some debugging and enables some configuration. However, it also takes memory away from the robot program. You can disable it by clicking on the *Disable Web Server* button. If you'd like to enable it again, you can click *Enable Web Server*.

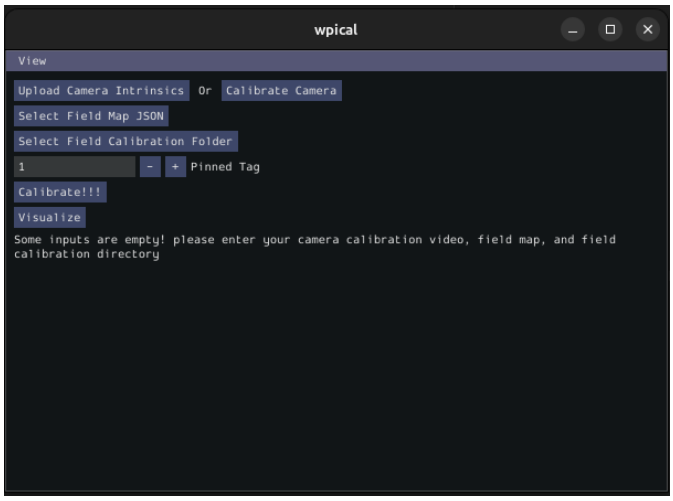
> **Warning**
>
> the Imaging Tool will not be able to connect to the roboRIO when the Web Server is disabled.

24.3 roboRIO Identification

Clicking the *Blink* button will cause the roboRIO's Radio LED to blink a few times to help identify the roboRIO.

25 WPIcal

WPIcal (inspired by cowlibration from FRC 1538) is a cross-platform utility that can be used to empirically measure the position and orientation of Apriltags. The primary use case for WPIcal is to make sure robots can perform as they would on an event field, but on a practice field that may have imperfect Apriltags setup.



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 *WPIcal*.

Warning

Before using this tool, it is important to understand that this tool is not a "silver bullet" of computer vision accuracy. The primary use case for WPIcal is to help detect and correct for incorrect Apriltag placement. While this tool can detect minor errors in precise tag positioning on a competition field, these corrections offer limited benefit and only for teams that have rigorously implemented all other good localization practices. With that said, this tool is exceedingly helpful in correcting for minor errors in practice fields, which are generally not within the normal field tolerance.

25.1 Overview

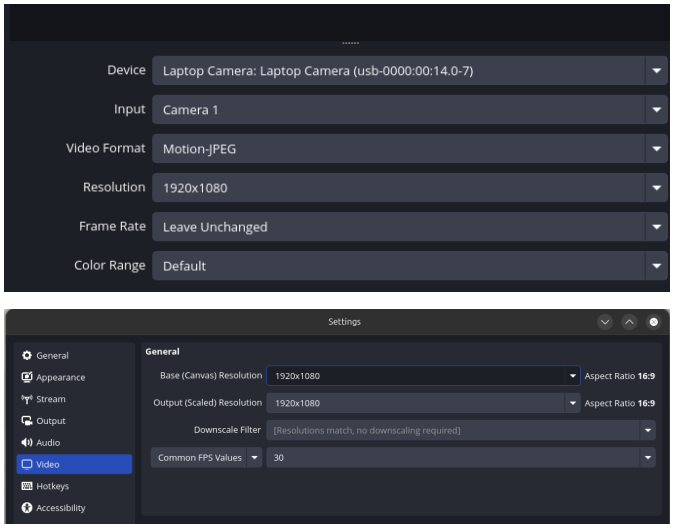
Conceptually, if you know how far tags are away from each other, and you know the field-relative coordinates of one of the tags, (the “pinned” tag) you can figure out the field-relative coordinates of the other tags. WPIcal achieves this by running an optimization on frames from videos that the user takes of the field. WPIcal is field-agnostic, meaning it will work with any FRC apriltag layout, as well as custom apriltag layouts.

25.2 Video Capture

The best way to obtain videos for both camera and field calibration is with a phone/tablet. Users can then transfer these videos to the computer running WPIcal for processing.

Alternatively, users can use other devices to capture videos, such as a laptop cameras. Users who opt to capture videos this way should use OBS Studio.

OBS Users should set the resolution of the video capture and the base video resolution in the OBS settings:



25.3 Camera Calibration

To measure the distance between tags in a video, you need the camera’s intrinsics to account for things like distortion. WPIcal allows users to upload a video file to calibrate the camera in the tool, or upload a JSON file with your camera’s intrinsics.

25.3.1 In-App Calibration

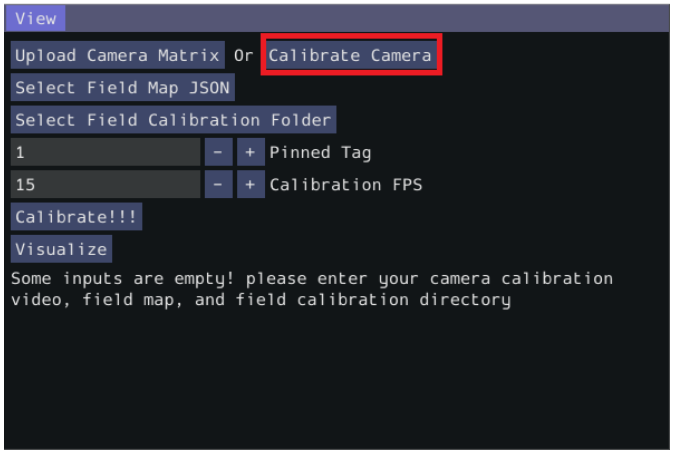
Linux Users Only:

Linux users must transcode all videos to MJPEG codec without an audio stream in an .avi file before uploading to WPIcal. This can be done with an ffmpeg command: `ffmpeg -i <video_file.mp4> -f avi -c:v mjpeg -b:v 20m -an <video_file.avi>`

> **Note**
>
> The `-b:v 20m` flag is optional, but it is recommended to use it to ensure the video quality is high enough for the calibration process. If you find the output .avi file is too grainy, you can increase the value of the flag.

All Users:

To calibrate your camera from a video file, click on *Calibrate Camera*



There are two options for calibrating your camera, *OpenCV* and *MRcal*. It is generally recommended to calibrate with *MRcal*, as it has slightly better accuracy and is more robust against bad calibration data. WPIcal uses a 5x5 aruco dictionary ChArUco board for camera calibration. You can generate a ChArUco board here: calib.io

> **Warning**
>
> If generating a ChArUco board with calib.io, make sure the height of the board is an odd number of squares. There is a bug in OpenCV which effects calib.io boards with an even number of rows.

> **Note**
>
> Regardless of which calibration option you use, make sure your calibration board has not been bent or creased, and is lying on a flat surface. It does not matter if you use the

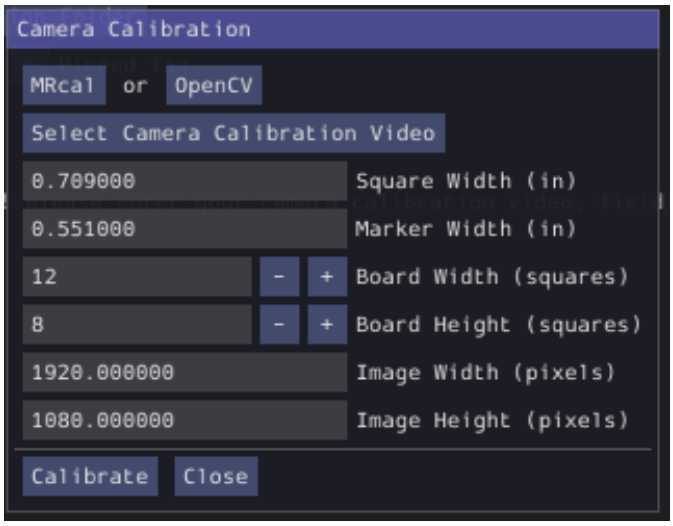
exact same board as shown in the images, as there are fields in each calibration option to customize the calibration for your specific board.

There are a couple of common fields that both *OpenCV* and *MRcal* calibration options use:

- *Select Camera Calibration Video* is the button to select your video file.
- *Square Width* is the width of the squares (in inches) of the board.
- *Marker Width* is the width of the ChArUco markers (in inches) on the board
- *Board Width* is the width (in squares) of the board
- *Board Height* is the height (in squares) of the board

For *MRcal*, there are two more options to fill in:

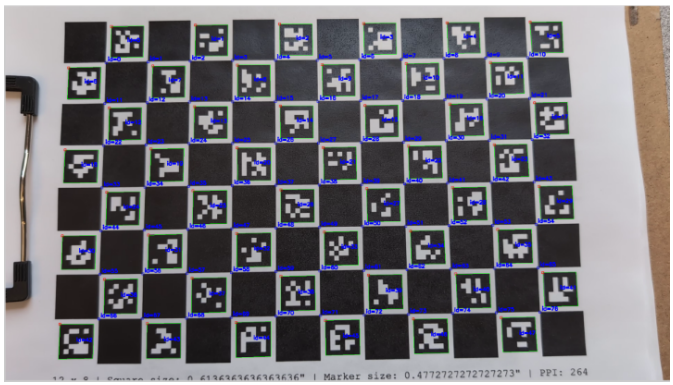
- *Image Width* is the width (in pixels) of the video's resolution
- *Image Height* is the height (in pixels) of the video's resolution



Important

For the calibration to work properly, it is important that you enter the exact width of the squares and markers. It is recommended to use calipers to accurately measure these widths.

When all the calibration fields have been entered, select *Calibrate* to run the calibration. A window will appear showing you the corner detection process (except on Linux, see below):



Warning

The corner detection window will not appear on Linux machines. WPIcal may show a `not responding` warning during the calibration, which is safe to ignore. The warning will disappear after calibration is complete.

Warning

If any frames from the camera calibration look suspect, take a new video of the calibration board and try again.

The camera calibration will automatically load the generated camera intrinsics JSON into WPIcal to continue to field calibration, but will also output the JSON file so it can be used for future calibrations. The calibration JSON will be saved as `cameracalibration.json` in the directory containing the calibration video you provided to WPIcal.

25.3.2 External Camera Calibration

As opposed to calibrating cameras in the tool, WPIcal also allows you to upload a camera intrinsics JSON.

CalibDB

One common method for external camera calibration is CalibDB.net. WPIcal allows you to directly upload the generated CalibDB JSON without any modification. Make sure to download the OpenCV format JSON:



Then:



Important

When uploading a CalibDB JSON, ensure that you download the calibration with the proper resolution selected.

Warning

CalibDB calibrations have been known to fail when bad snapshots are taken. If your calibration data looks suspect, repeat calibration process again.

Custom JSON

There are a three things each calibration JSON needs:

- `avg_reprojection_error`
- `camera_matrix`
- `distortion_coefficients`

Example:

```
{
    "avg_reprojection_error": 0.3989609373420966,
    "camera_matrix": [
        1767.912766105077,          // fx
        0.0,
        989.4346078930205,          // cx
        0.0,
        1763.5161174120185,         // fy
        540.7047848580446,          // cy
        0.0,
        0.0,
        1.0
    ],
    "distortion_coefficients": [
        0.3878712310541597,         // k1
        -2.565991238351721,         // k2
        -0.0011767048240595168,     // p1
        0.0017341455009923185,      // p2
        5.70562560518866,           // k3
        0.03694246896004978,        // k4
        0.11811204683668342,        // k5
        -0.2274481545065017,        // k6
        0.0,                        // s1
        0.0,                        // s2
        0.0,                        // s3
```

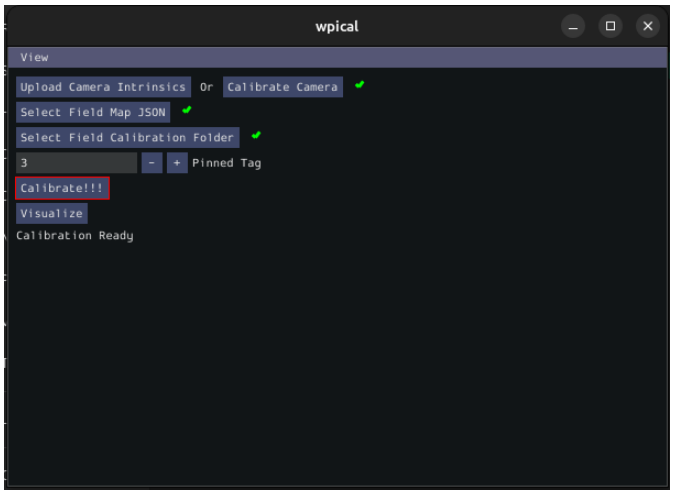
(continues on next page)

(continued from previous page)

```
        0.0,                        // s4
        0.0,                        // tx
        0.0                         // ty
    ]
}
```

25.4 Field Calibration

After calibrating the camera, you can use the camera model to find the relative positions of the Apriltags. The calibration process will generate a WPILib field layout .json file and a .fmap for use on coprocessors and in robot code. WPIcal will prompt the user to specify a location to save the generated .json and .fmap field layouts to when the *Calibrate!!!* button is pressed.



25.4.1 Field Video Tips

In order to get a good field calibration, it is important use follow these tips:

- Make sure to use the same camera as was used in the camera calibration.
- WPIcal can take in a directory of videos to calibrate the field. It is recommended to take videos from multiple angles on the tags of interest to get the best calibration.
- Tags of interest should clearly link back to the pinned tag. For example if tag 1 is pinned:

Video 1	Video 2
tag 1, tag 2	tag 2, tag 3

25.4.2 Upload Ideal Field Map

WPIcal uses an "ideal" field map JSON as an initial guess point for the optimization. It is recommended to upload the json file included with WPILib, which can be found here: Field JSON. If using a custom map, it must follow these rules:

- Translation components (in meters) are measured relative to the blue alliance origin.
- Rotations are represented as Quaternions.

Example:


```
{
    "ID": 1,
    "pose": {
        "translation": {
            "x": 15.079471999999997,
            "y": 0.24587199999999998,
            "z": 1.355852
        },
        "rotation": {
            "quaternion": {
                "W": 0.5000000000000001,
                "X": 0.0,
                "Y": 0.0,
                "Z": 0.8660254037844386
            }
        },
    },
}
```


25.4.3 Select Field Calibration Directory

WPIcal can calibrate a field based on one or more videos. All the calibration videos must be stored in their own directory, separate from any other files.

25.4.4 Pinned Tag

The pinned tag is the tag that other tags are transformed relative to. This tag should be the tag that is the most accurate on the field.

25.5 View Field Calibration

After the calibration is completed, you can view the difference between the reference tags and the calibrated tags to double check that your calibrated values look reasonable.

Differences are shown in meters and degrees. The differences are calculated by taking the difference between the pose of the tag in the ideal map and the pose of the tag in the calibrated map.

References show the pose of the calibrated tag in relation to another calibrated tag to aid in double checking the tag's position relative to others.

Focused Tag is the tag that WPIcal is calculating the position error of.

Reference Tag is the tag that the *Focused Tag* is being compared to.



Important

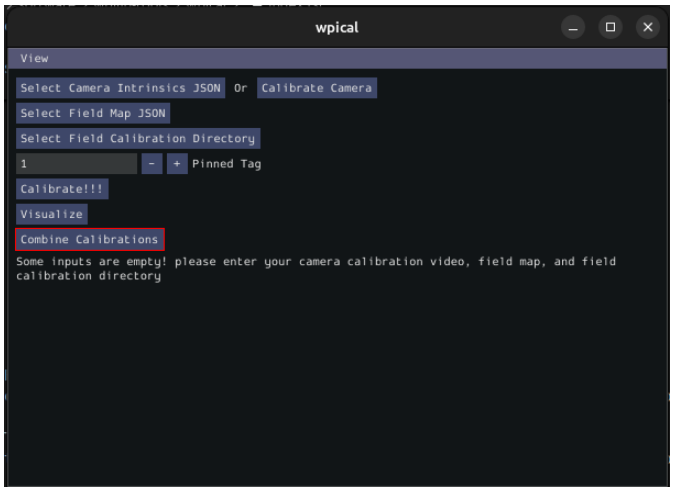
WPIcal is meant to correct for SMALL variations in tag placement. It is still important that you set up your Apriltags in mostly the correct location and orientation, so WPIcal performs the optimal calibration.

Important

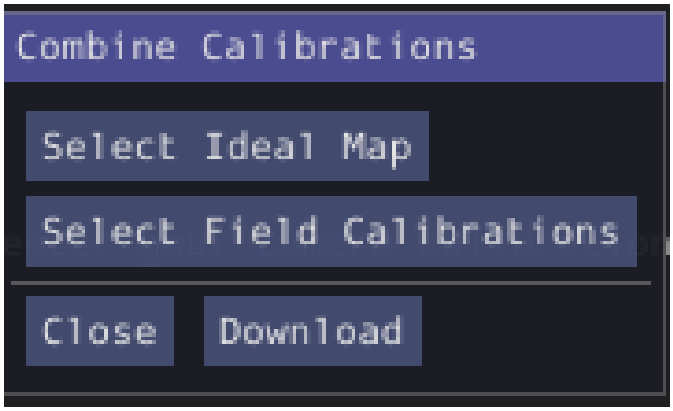
Make sure that you verify the results of each calibration thoroughly to ensure that your calibration matches your field setup accurately.

25.6 Combine Field Calibrations

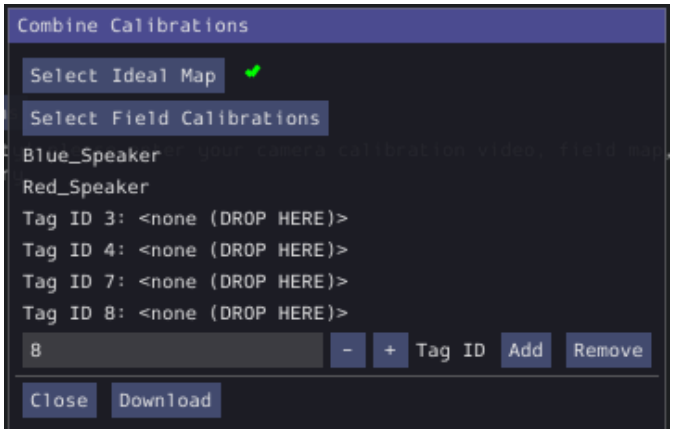
WPIcal can combine multiple field calibrations into a single field calibration. This is useful in cases where fields are split into multiple sections. To combine field calibrations, select the *Combine Calibrations* button.



Load in the ideal field map JSON and the field calibration JSONs you want to combine.



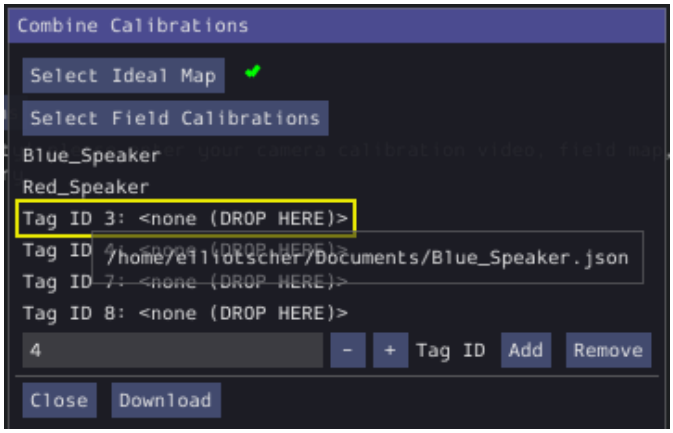
WPIcal will prompt you to select which tags you would like to add to your combined field calibration. Change *Tag ID* to the ID of the tag you would like to add or remove from the combined field calibration. Press *Add* to add the tag to the combined field calibration, or *Remove* to remove the tag from the combined field calibration.



Tip

Any tags that you do not include in the combination will fall back to the ideal field map.

After selecting the tags you would like to add to your combined field calibration, drag and drop the calibrated fields to the tags you wish to associate them with.



After dragging and dropping the field maps, press *Download* to generate the combined field calibration .json and .fmap. If you have not yet selected a download directory from the field calibration process, WPIcal will prompt you to save the combined field calibration to a directory.

26 Vision Processing

26.1 Vision Introduction

26.1.1 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*

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 OpenCV.

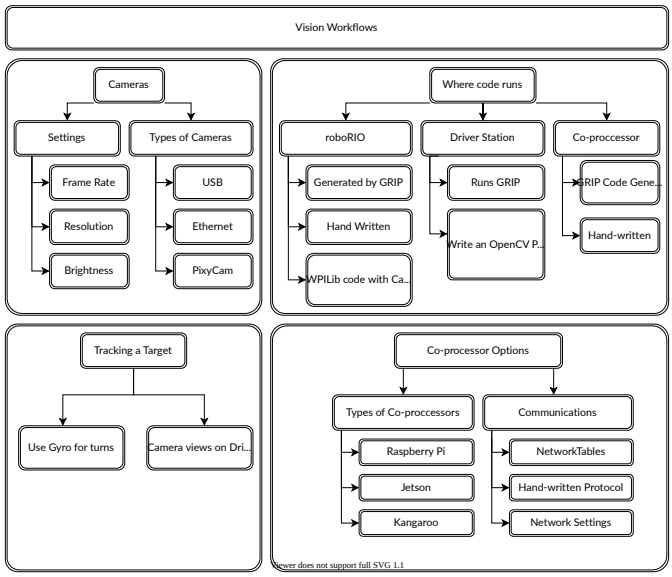
- *Vision Processing with Raspberry Pi*
- *Vision Processing with the roboRIO*

For additional information on the pros and cons of using a coprocessor for vision processing, see the next page, *Strategies for Vision Programming*.

26.1.2 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.

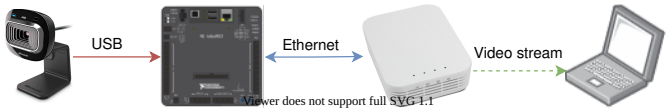
There are many options for choosing the components for vision processing and where the vision program should run. WPILib and associated tools support a number of options and give teams a lot of flexibility to decide what to do. This article will attempt to give you some insight into many of the choices 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

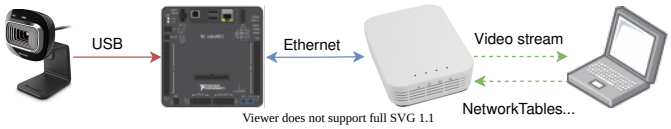


Vision code can be embedded into the main robot program on the roboRIO. Building and running the vision code is straightforward because it is built and deployed along with the robot program. The vision code can be written in C++, Java, or Python. The disadvantage of this approach is that having vision 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 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

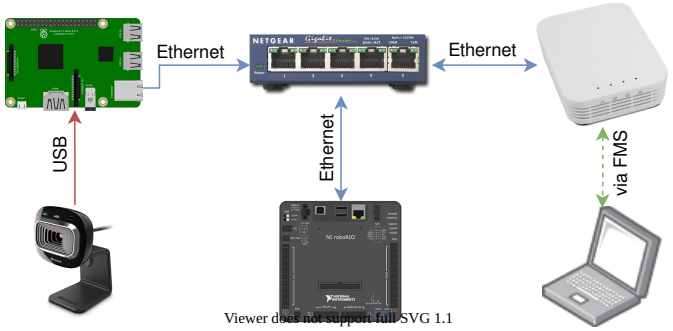


When vision code is running on the DS computer, the video is streamed back to the Driver Station laptop for processing. Even the older Classmate laptops are substantially faster at vision processing than the roboRIO. You can write your own vision program using a language of your choosing. Python makes 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 SmartDashboard.

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 retrieve frames for local processing through a Source object and sends the stream to your driver station for viewing or processing there.

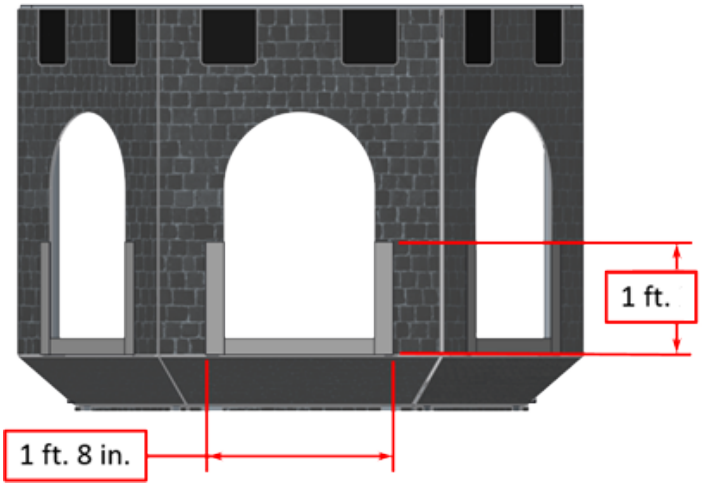
26.1.3 Target Info and Retroreflection

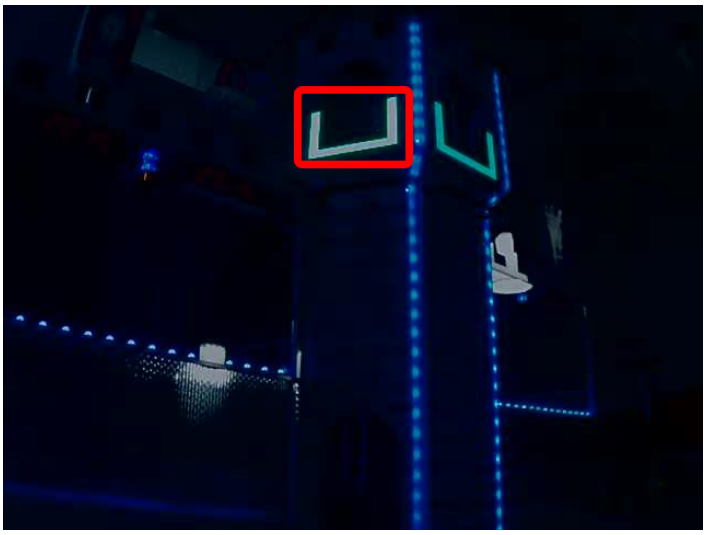
Many FRC® games have retroreflective tape attached to field elements to aid in vision processing. This document describes the Vision Targets from the 2016 FRC game and the visual properties of the material making up the targets.

Note

For official dimensions and drawings of all field components, please see the Official Field Drawings.

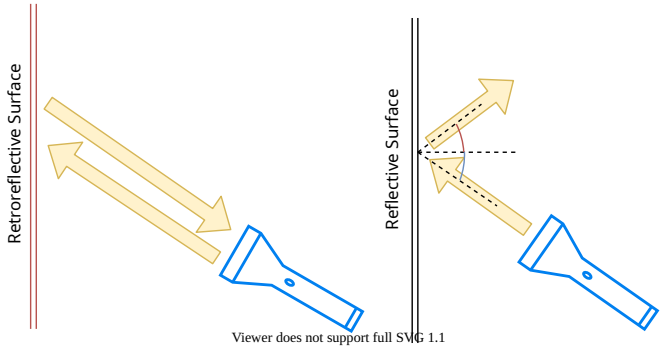
Targets





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 exhibit *retro-reflection*, 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.

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

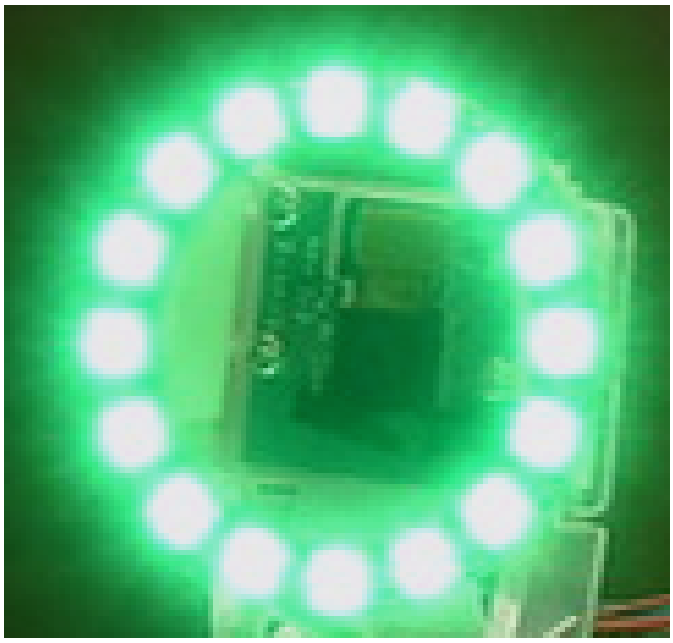
To further explore retro-reflective material properties:

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

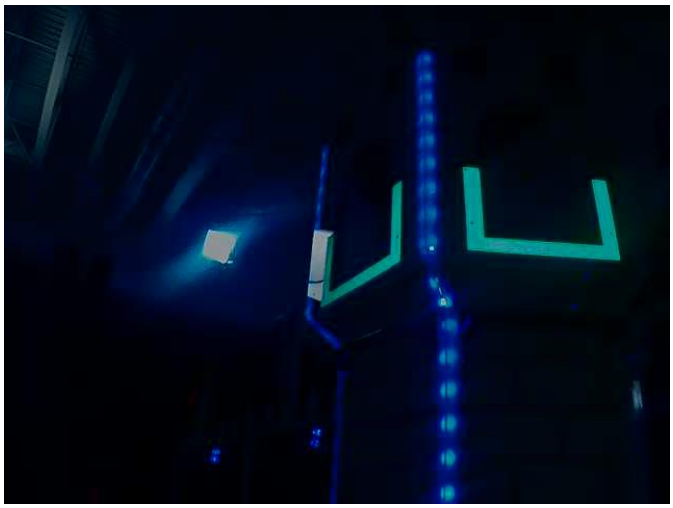
Sample images are located with the code examples for each language (packaged with LabVIEW, and in a separate ZIP in the same location as the C++/Java samples).

26.1.4 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.

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. Additional sample images are provided with the vision code examples.



What is HSL/HSV?

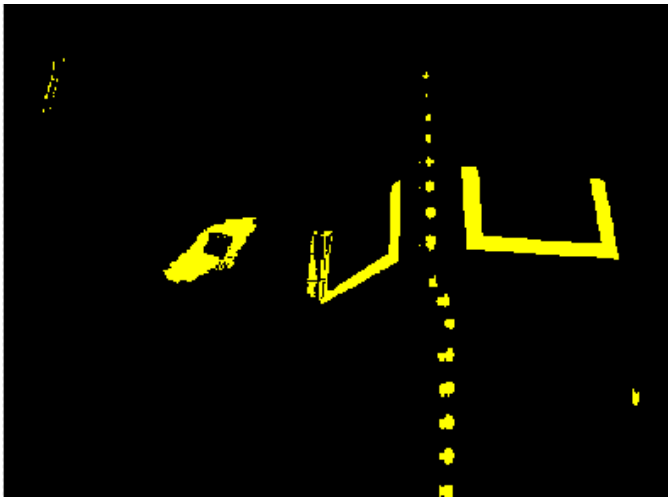
The Hue or tone of the color is commonly seen on the artist's color wheel and contains the colors of the rainbow Red, Orange, Yellow, Green, Blue, Indigo, and Violet. The hue is specified using a radial angle on the wheel, but in imaging the circle typically contains only 256 units, starting with red 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 (~516 cm^2). The area of the rectangle that contains the target is 240 square inches (~0.15 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

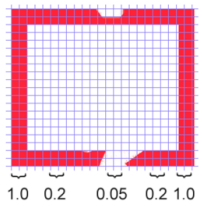
The aspect ratio score is based on (Particle Width / Particle Height). The width and height of the particle are determined using something called the "equivalent rectangle". The equivalent rectangle is the rectangle with side lengths x and y where $2x + 2y$ equals the particle perimeter and $x \cdot y$ 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.

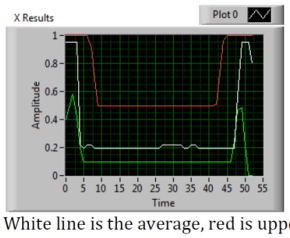
Moment

The "moment" measurement calculates how spread out each pixel is from the center of the blob. This measurement provides a representation of the pixel distribution in the particle. It can be thought of as analogous to a physics *moment of inertia* calculation. The ideal score for this test is ~0.28.

X/Y Profiles



Column averages for a particle rectangle.



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.

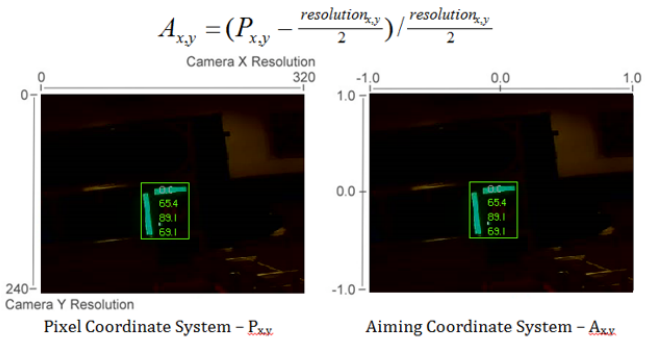
Position

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 Drive inputs.

$$A_{x,y} = \left(P_{x,y} - \frac{resolution_{x,y}}{2}\right) / \frac{resolution_{x,y}}{2}$$



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.

26.1.5 Read and Process Video: CameraServer Class

Concepts

The cameras typically used in FRC® (commodity USB and Ethernet cameras) 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()` functions will automatically name the camera (e.g. "USB Camera 0"), 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.
- `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 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

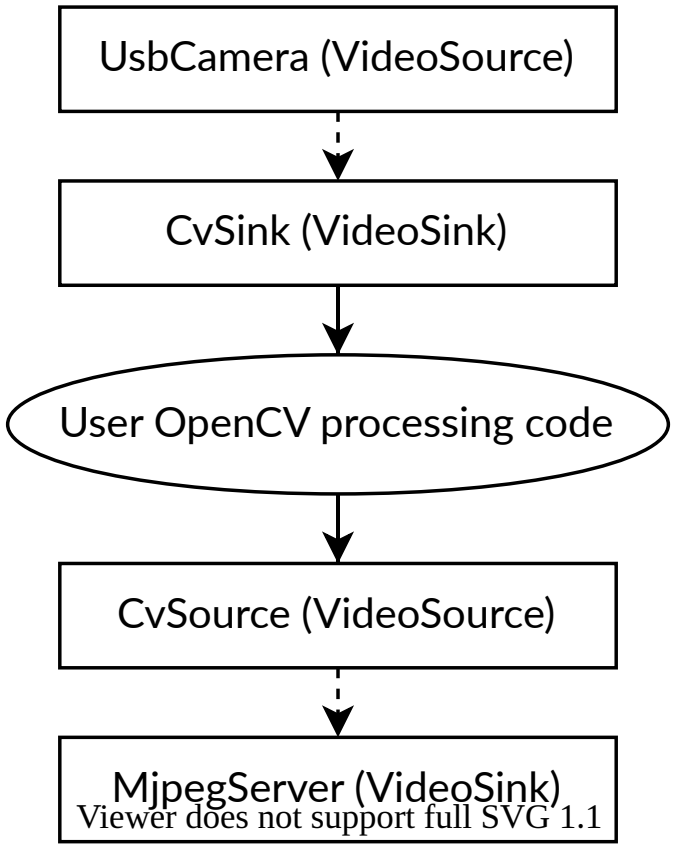
Sources and Sinks

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.

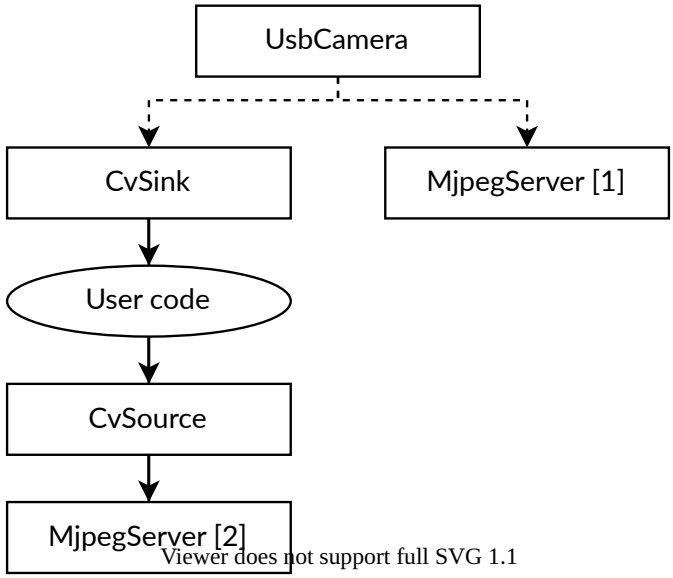
An object that generates images is a source and an object that accepts/consumes images is a sink. The generate/consume is from the perspective of the library. Thus cameras are sources (they generate images). The MJPEG web server is a sink because it accepts images from within the program (even though 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

```
import edu.wpi.first.cameraserver.CameraServer;
import edu.wpi.cscore.CvSink;
import edu.wpi.cscore.CvSource;
// Creates UsbCamera and MjpegServer [1] and connects them
CameraServer.startAutomaticCapture();
// Creates the CvSink and connects it to the UsbCamera
CvSink cvSink = CameraServer.getVideo();
// Creates the CvSource and MjpegServer [2] and connects them
CvSource outputStream = CameraServer.putVideo("Blur", 640, 480);
```

C++

```
#include "cameraserver/CameraServer.h"
// Creates UsbCamera and MjpegServer [1] and connects them
frc::CameraServer::StartAutomaticCapture();
// Creates the CvSink and connects it to the UsbCamera
cs::CvSink cvSink = frc::CameraServer::GetVideo();
// Creates the CvSource and MjpegServer [2] and connects them
cs::CvSource outputStream = frc::CameraServer::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

```
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);
```

C++

```
#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::kMJPEG, 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).

26.1.6 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.

26.2 Vision with WPILibPi

26.2.1 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.

26.2.2 Using a Coprocessor for vision processing

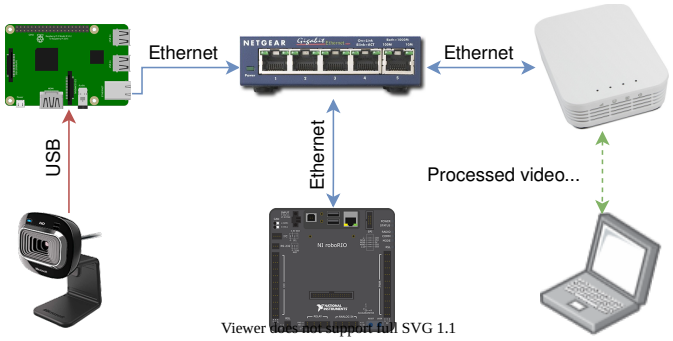
Vision processing using libraries like OpenCV for recognizing field targets or game pieces can often be a CPU intensive process. Often the load isn't too significant and the processing can easily be handled by the roboRIO. In cases where there are more camera streams or the image processing is complex, 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

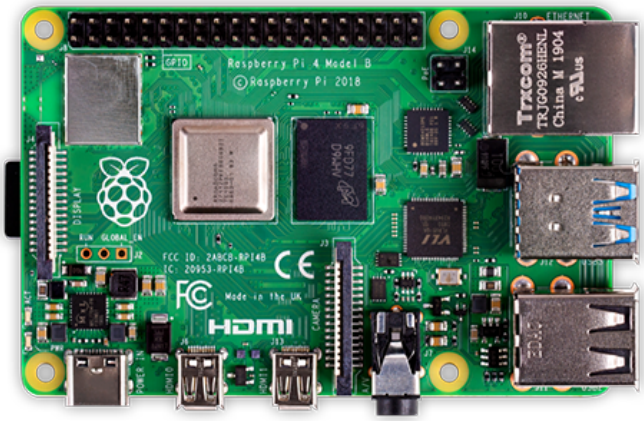
It is often desirable to simply stream the camera data to the dashboard over the robot network. In this case one or more camera connections can be sent to the network and viewed on a dashboard such as Shuffleboard or a web browser. Using Shuffleboard has the advantage of having easy controls to set the 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.

26.2.3 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.

26.2.4 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.

26.2.5 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.

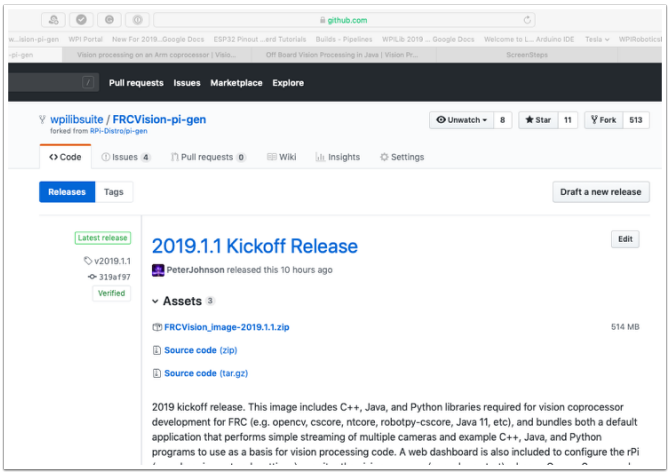
In addition to the instructions on this page, see the documentation on the GitHub web page (below).

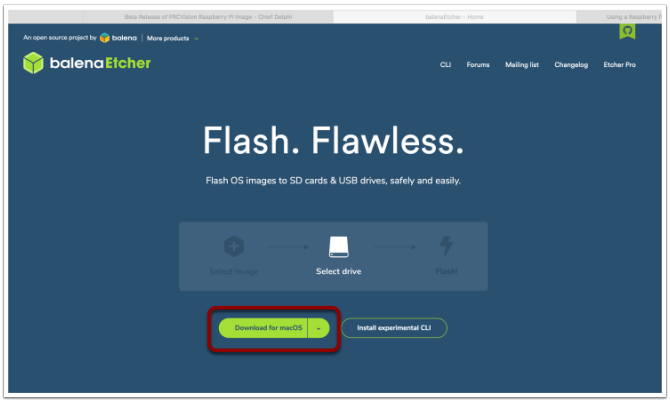
The image is fairly large so have a fast internet connection when downloading it. Always use the most recent release from the top of the list of releases.

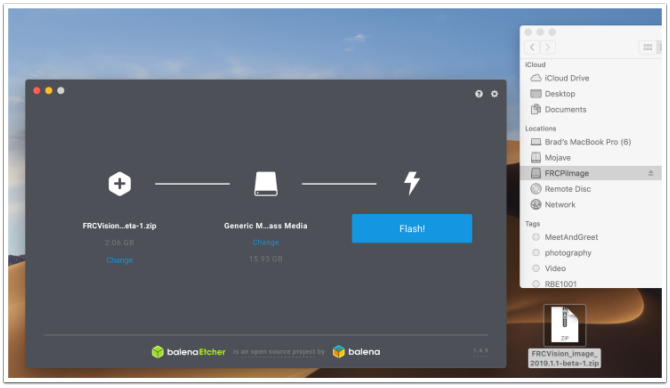
Copy the image to your MicroSD card

Download and install Etcher to image the micro SD card. The micro SD card needs to be at least 8 GB. A micro SD to USB dongle 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.


Pull requests
Issues
Marketplace
Explore
wpilibsuite / FRCVision-pi-gen
forked from RPi-Distro/pi-gen
Unwatch 8
Star 11
Fork 513
Code
Issues 4
Pull requests 0
Wiki
Insights
Settings
Releases
Tags
Draft a new release
Latest release
v2019.1.1
319af97
Verified
2019.1.1 Kickoff Release
Edit
PeterJohnson released this 10 hours ago
Assets 3
FRCVision_image-2019.1.1.zip
514 MB
Source code (zip)
Source code (tar.gz)
2019 kickoff release. This image includes C++, Java, and Python libraries required for vision coprocessor development for FRC (e.g. opencv, cscore, ntcore, robotpy-cscore, Java 11, etc), and bundles both a default application that performs simple streaming of multiple cameras and example C++, Java, and Python programs to use as a basis for vision processing code. A web dashboard is also included to configure the rPi


balenaEtcher
CLI
Forums
Mailing list
Changelog
Etcher Pro
Flash. Flawless.
Flash OS images to SD cards & USB drives, safely and easily.
Select drive
Download for macOS
Install experimental CLI


Flash!
iCloud
iCloud Drive
Desktop
Documents
Locations
Mojave
Remote Disc
Network
Tags
MeetAndGreet
photography
Video
RBE1001

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
```

26.2.6 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:

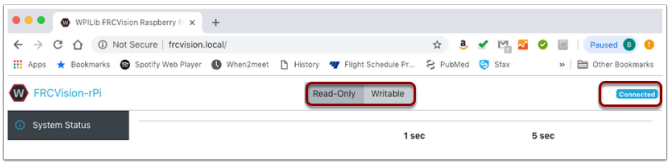
- Look at the Raspberry PI status
- View the status of the background process running the camera
- View or change network settings
- Look at each camera plugged into the rPI and add additional cameras
- Load a new vision program onto the rPI

Setting the rPI to be Read-Only vs. Writable

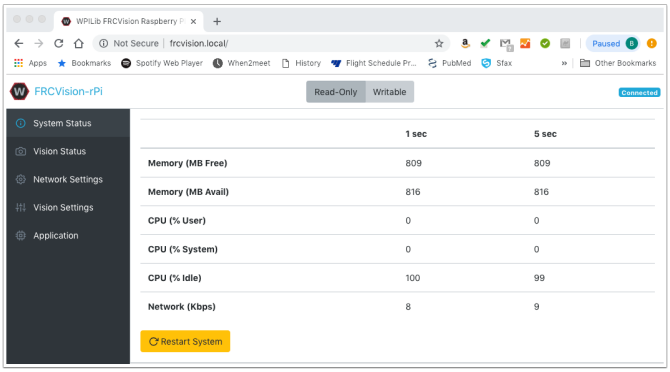
The rPI is normally set to Read-Only which means that the file system cannot be changed. This ensures that if power is removed without first shutting down the rPi the file system isn't corrupted. When settings are changed (following sections), the new settings cannot be saved while the rPI file system 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

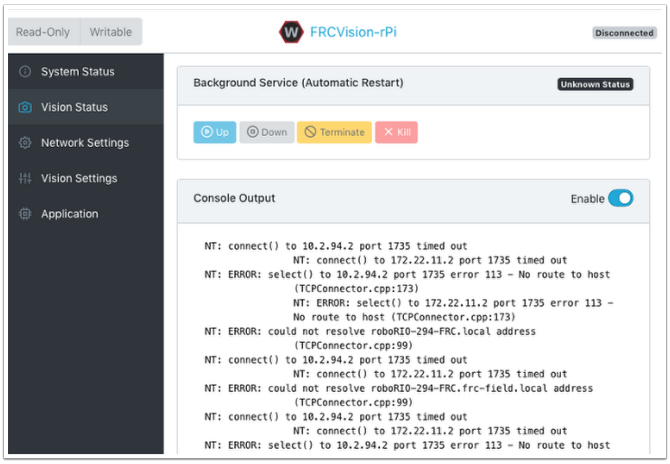


The system status shows what the CPU on the rPI is doing at any time. There are two columns of status values, on being a 1 second average and the other a 5 second average. Shown is:

- Free and available RAM on the PI
- CPU usage for user processes and system processes as well as idle time
- 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

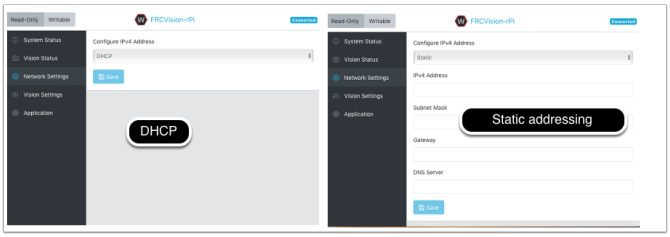
Allows monitoring of the task which is running the camera code in the rPI, either one of the default programs or your own program in Java, C++, or Python. You can also enable and view the console output to see messages coming from the background camera service. In this case there are number of messages 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

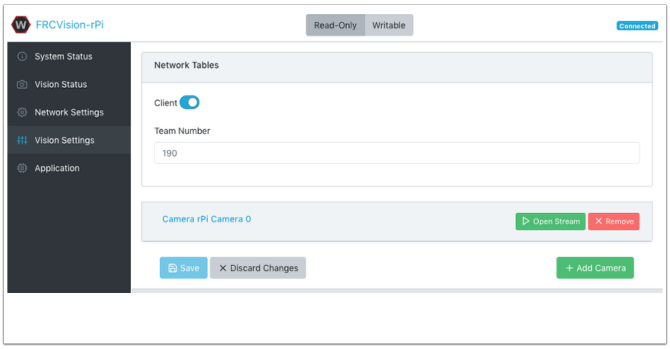
The rPI network settings have options to connect to the PI:

- *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.



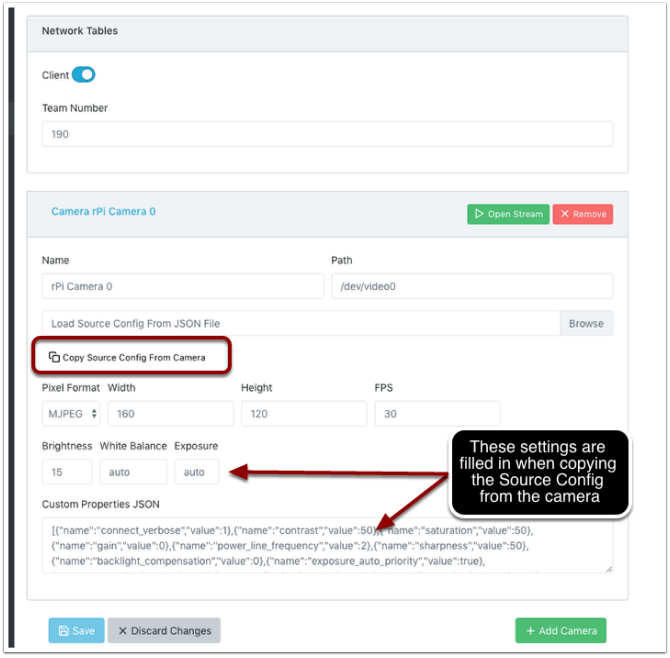
Manipulating the camera settings is reflected in the current camera view. The bottom of the page shows all the possible camera modes (combinations of Width, Height, and frame rates) that are supported by this camera.

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

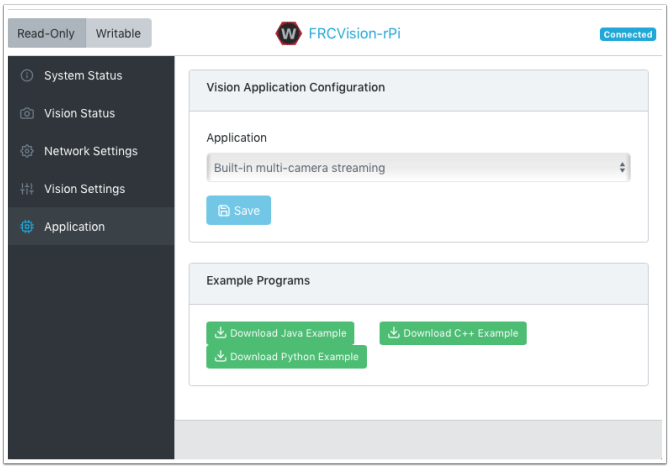
The Application tab shows the application that is currently running on the rPi.

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:

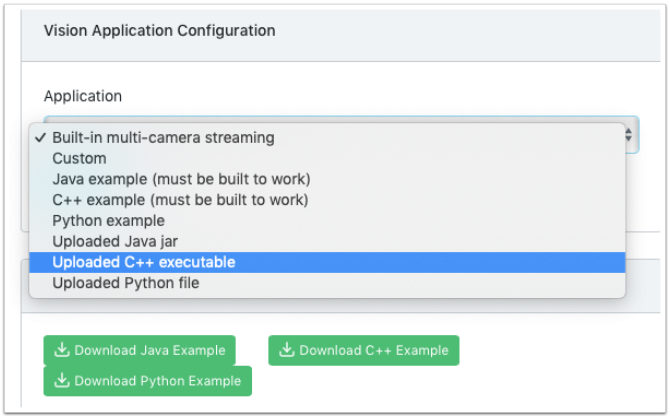
- Stream one or more cameras from the rPi for consumption on the driver station computer and displayed using ShuffleBoard
- Edit and build one of the sample programs (one for each language: Java, C++ or Python) on the rPi using the included toolchains

- 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
- Do everything yourself using completely custom applications and scripts (probably based on one of the samples)



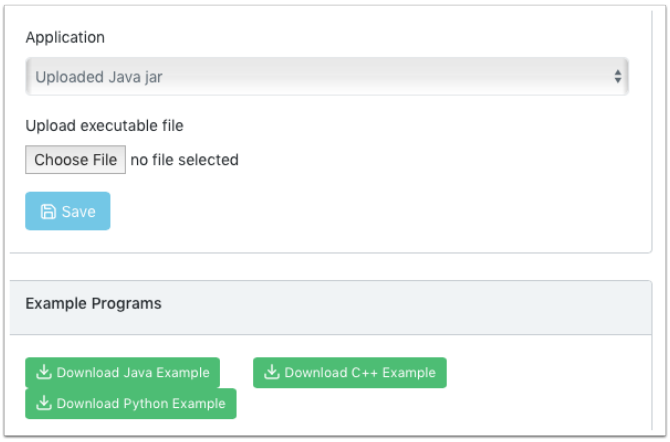
The running application can be changed by selecting one of the choices in the drop-down menu. The choices are:

- 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.



26.2.7 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*.

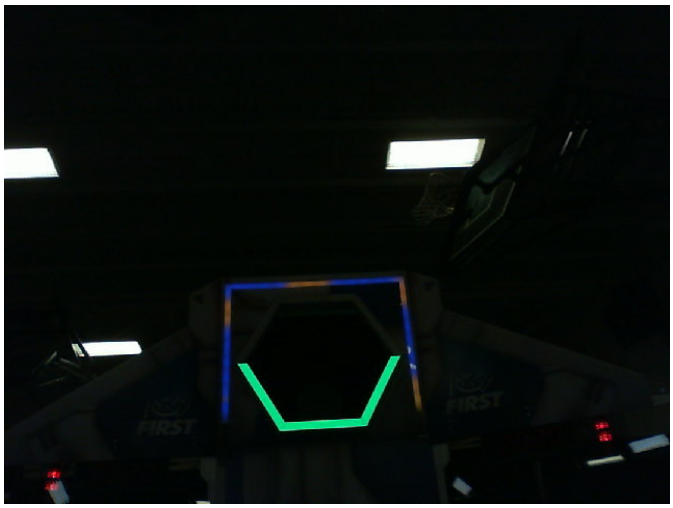
PY

```
from cscore import CameraServer
import cv2
import numpy as np
CameraServer.enableLogging()
camera = CameraServer.startAutomaticCapture()
camera.setResolution(width, height)
sink = CameraServer.getVideo()
input_img = np.zeros(shape=(height, width, 3), dtype=np.uint8)
while True:
   time, input_img = sink.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

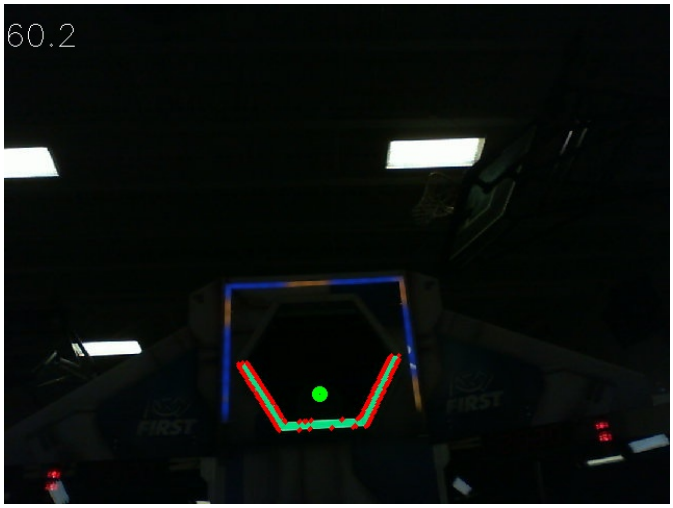
Sometimes, you may want to send processed video frames back to the CameraServer instance for debugging purposes, or viewing in a dashboard application like Shuffleboard.

PY

```
#
# CameraServer initialization code here
#
output = CameraServer.putVideo("Name", width, height)
input_img = np.zeros(shape=(height, width, 3), dtype=np.uint8)
while True:
   time, input_img = sink.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.



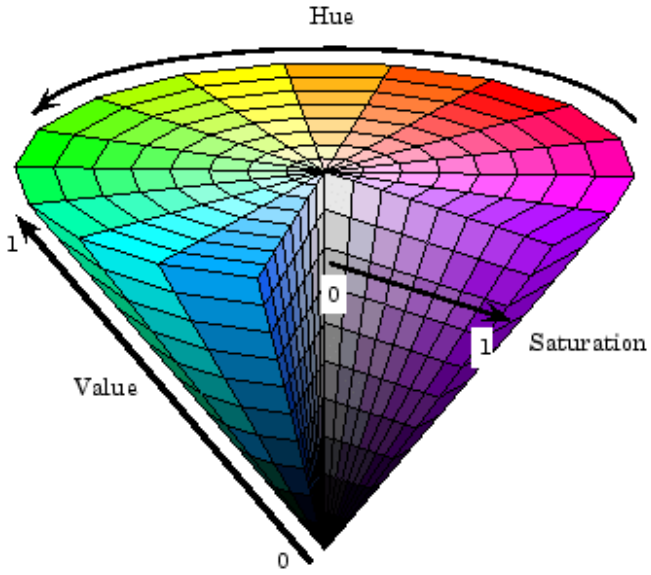
26.2.8 Thresholding an Image

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.

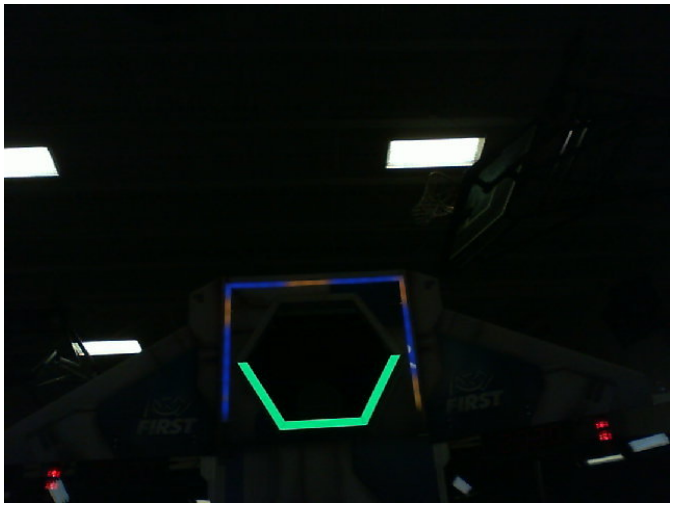
PY

```
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.



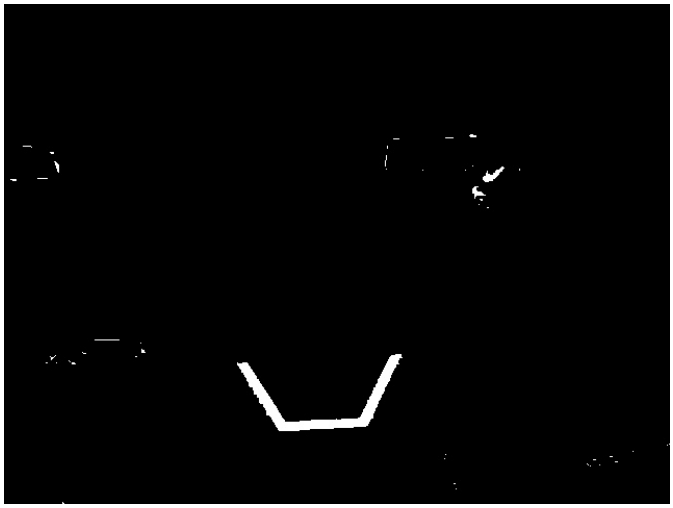
By thresholding the image using HSV, you can separate the image into the vision target (foreground), and the other things that the camera sees (background). The following code example converts a HSV image into a binary image by thresholding with HSV values.

PY

```
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.

26.2.9 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.

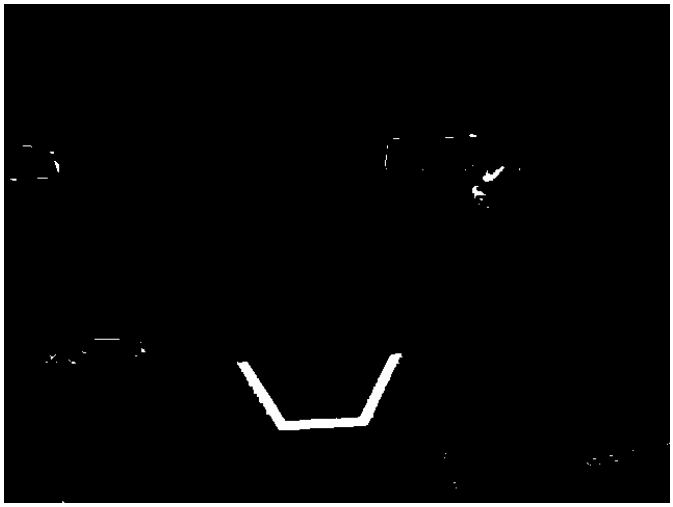
Kernel

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 NxN 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:

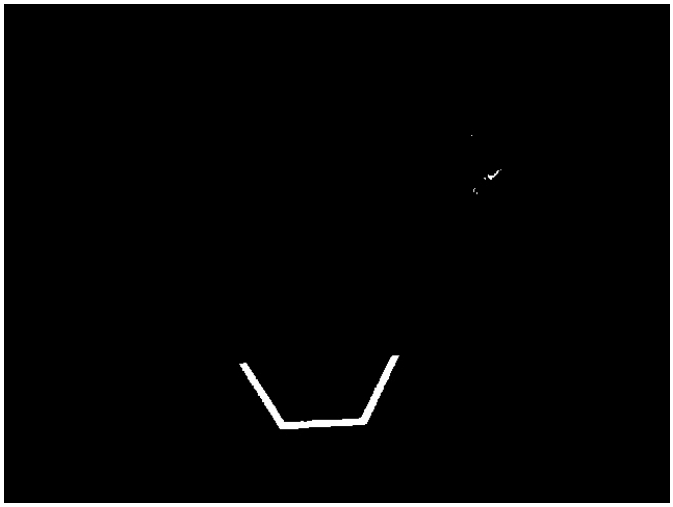


Erosion

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.

PY

```
kernel = np.ones((3, 3), np.uint8)
binary_img = cv2.erode(binary_img, kernel, iterations = 1)
```



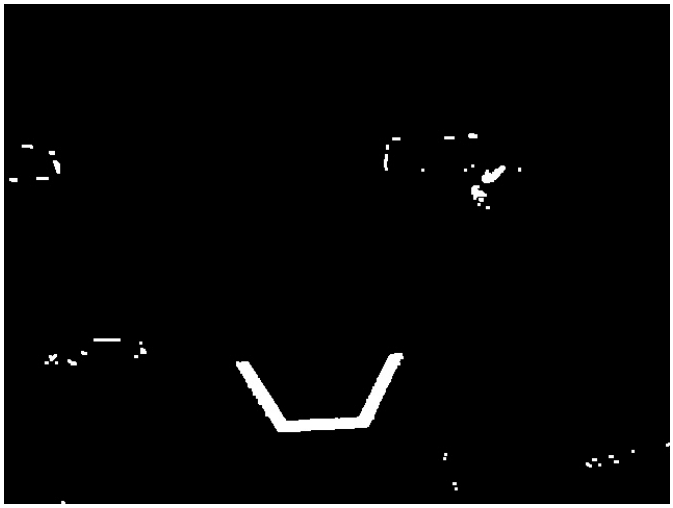
During erosion, if the superimposed kernel's pixels are not contained completely by the binary image's pixels, the pixel that it was superimposed on is deleted.

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.

PY

```
kernel = np.ones((3, 3), np.uint8)
binary_img = cv2.dilate(binary_img, kernel, iterations = 1)
```



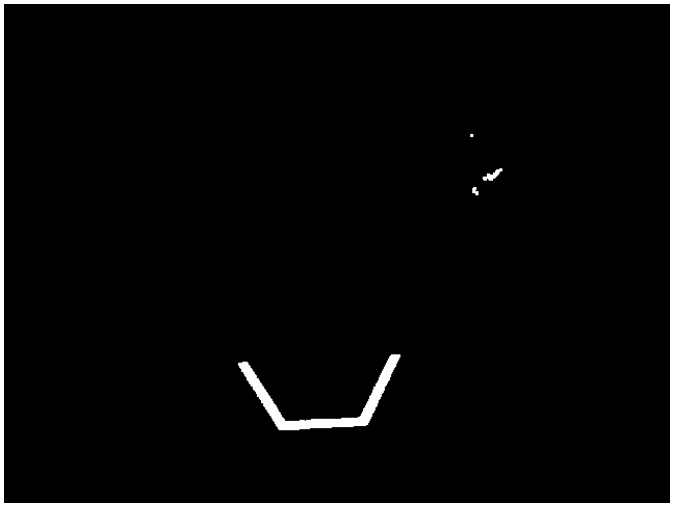
During dilation, every pixel of every superimposed kernel is included in the dilation.

Opening

Opening is erosion followed by dilation. This process removes noise without affecting the shape of larger features.

PY

```
kernel = np.ones((3, 3), np.uint8)
binary_img = cv2.morphologyEx(binary_img, cv2.MORPH_OPEN, kernel)
```



Note

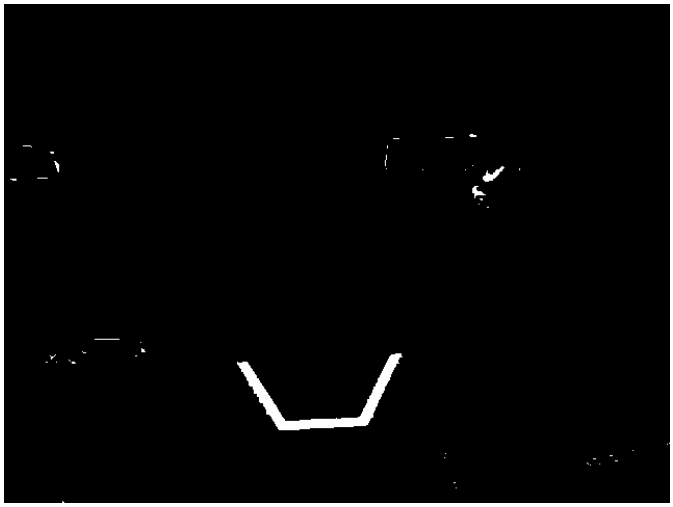
In this specific case, it is appropriate to do more iterations of opening in order to get rid of the pixels in the top right.

Closing

Closing is dilation followed by erosion. This process removes small holes or breaks without affecting the shape of larger features.

PY

```
kernel = np.ones((3, 3), np.uint8)
binary_img = cv2.morphologyEx(binary_img, cv2.MORPH_CLOSE, kernel)
```



26.2.10 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

PY

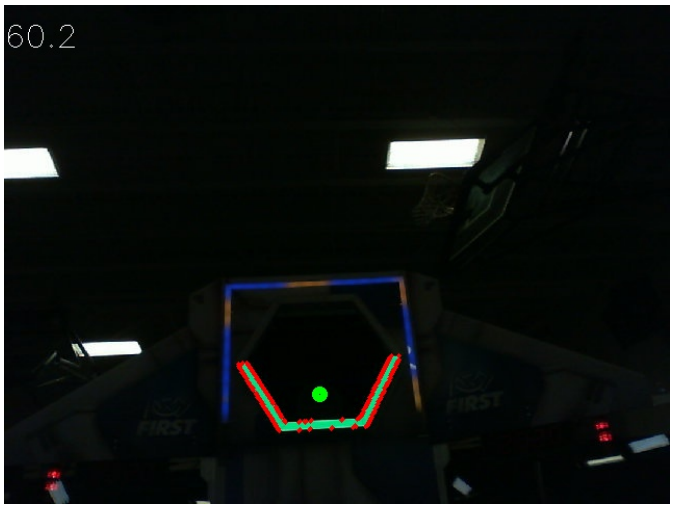
```
_, contours, _ = cv2.findContours(binary_img, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_
↪SIMPLE)
```

In cases where there is only one vision target, you can just take the largest contour and assume that 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.

PY

```
if len(contours) > 0:
   largest = contours[0]
   for contour in contours:
      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

PY

```
rect = cv2.minAreaRect(contour)
center, _, _ = rect
center_x, center_y = center
```

Corners

PY

```
corners = cv2.convexHull(contour)
corners = cv2.approxPolyDP(corners, 0.1 * cv2.arcLength(contour), True)
```

Rotation

PY

```
_, _, 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.

PY

```
import ntcore
nt = ntcore.NetworkTableInstance.getDefault().getTable('vision')
#
# Initialization code here
#
while True:
   #
   # Image processing code here
   #
   nt.putNumber('center_x', center_x)
   nt.putNumber('center_y', center_y)
```

26.2.11 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.

PY

```
from cscore import CameraServer
import ntcore
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']
   nt = ntcore.NetworkTableInstance.getDefault()
   CameraServer.startAutomaticCapture()
   input_stream = CameraServer.getVideo()
   output_stream = CameraServer.putVideo('Processed', width, height)
   # Table for vision output information
   vision_nt = nt.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, (0, 0, 100), (85, 255, 255))
      _, contour_list = cv2.findContours(binary_img, mode=cv2.RETR_EXTERNAL,␣
↪method=cv2.CHAIN_APPROX_SIMPLE)
      x_list = []
      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 = tuple([int(dim) for dim in center])  # Convert to int so we can␣
↪draw
```

(continues on next page)

(continued from previous page)

```
            # Draw rectangle and circle
            cv2.drawContours(output_img, [cv2.boxPoints(rect).astype(int)], -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()
```

26.3 AprilTag Introduction

26.3.1 What Are AprilTags?



AprilTags are a system of visual tags developed by researchers at the University of Michigan to provide low overhead, high accuracy localization for many different applications.

Additional information about the tag system and its creators can be found on their website. This document attempts to summarize the content for FIRST robotics related purposes.

Application to FRC

In the context of FRC, AprilTags are useful for helping your robot know where it is at on the field, so it can align itself to some goal position.

AprilTags have been in development since 2011, and have been refined over the years to increase the robustness and speed of detection.

Starting in 2023, FIRST is providing a number of tags, scattered throughout the field, each at a known *pose*.

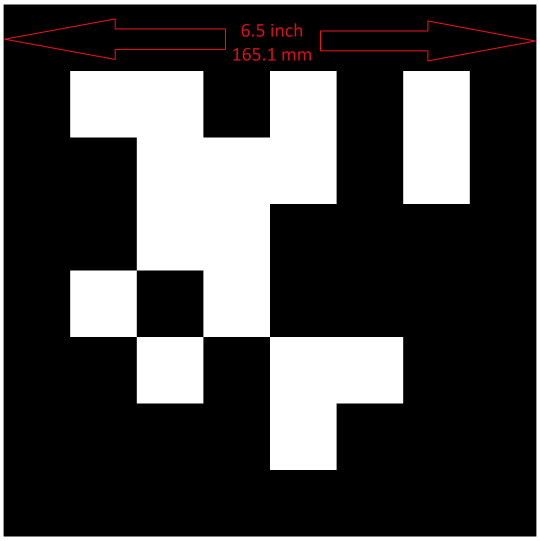
The AprilTag library implementation defines standards on how sets of tags should be designed. Some of the possible tag families are described here.

Starting from 2024, FIRST has chosen the 36h11 family. This family of tags is made of a 6x6 grid of pixels, each representing one bit of information. An additional black and white border must be present around the outside of the bits.

While there are $2^{36} = 68,719,476,736$ theoretical possible tags, only 587 are actually used. These are chosen to:

1. Be robust against some bit flips (IE, issues where a bit has its color incorrectly identified).
2. Not involve "simple" geometric patterns likely to be found on things which are not tags. (IE, squares, stripes, etc.)
3. Ensure the geometric pattern is asymmetric enough that you can always figure out which way is up.

All tags will be printed such that the tag's main "body" is 6.5 inches in length.



For home usage, tag files may be printed off and placed around your practice area. Mount them to a rigid backing material to ensure the tag stays flat, as the processing algorithm assumes the tags are flat.

Software Support

The main repository for the source code that detects and decodes AprilTags is located here.

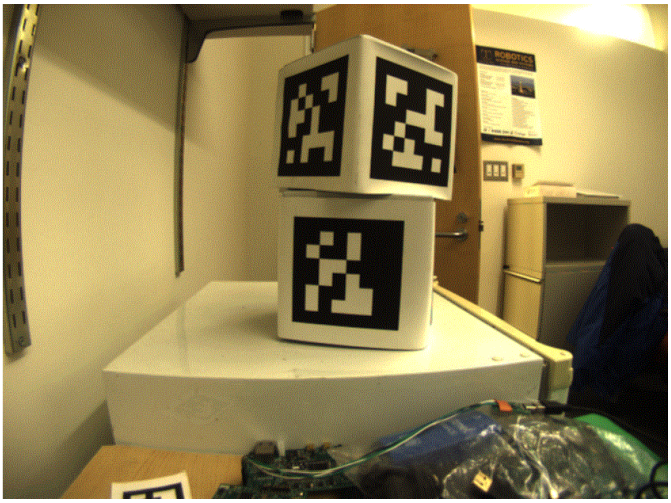
WPILib has forked the repository to add new features for FRC. These include:

1. Building the source code for common FRC targets, including the roboRIO and Raspberry Pi.
2. Adding Java Native Interface (JNI) support to allow invoking its functionality from Java
3. Gradle & Maven publishing support

Processing Technique

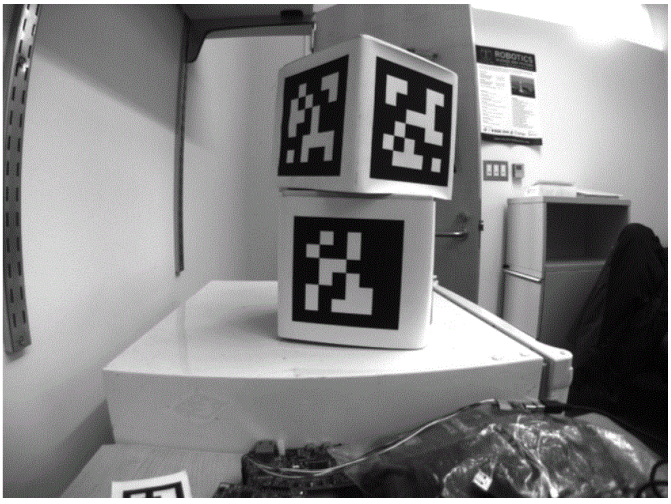
While most FRC teams should not have to implement their own code to identify AprilTags in a camera image, it is useful to know the basics of how the underlying libraries function.

Original Image



An image from a camera is simply an array of values, corresponding to the color and brightness of each pixel.

Remove Colors



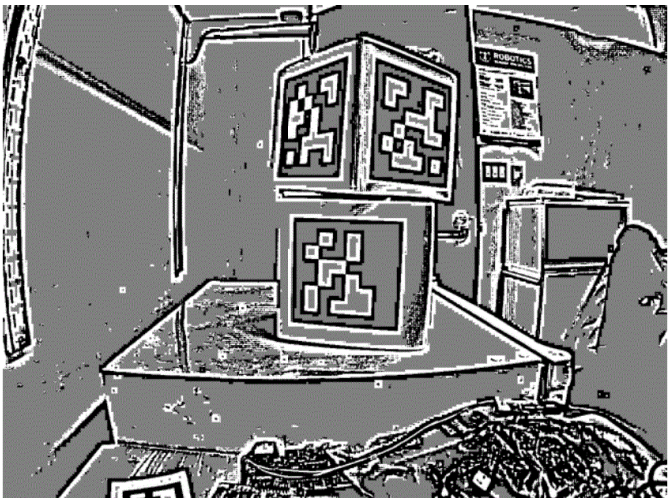
The first step is to convert the image to a grey-scale (brightness-only) image. Color information is not needed to detect the black-and-white tags.

Decimate



The next step is to convert the image to a lower resolution. Working with fewer pixels helps the algorithm work faster. The full-resolution image will be used later to refine early estimates.

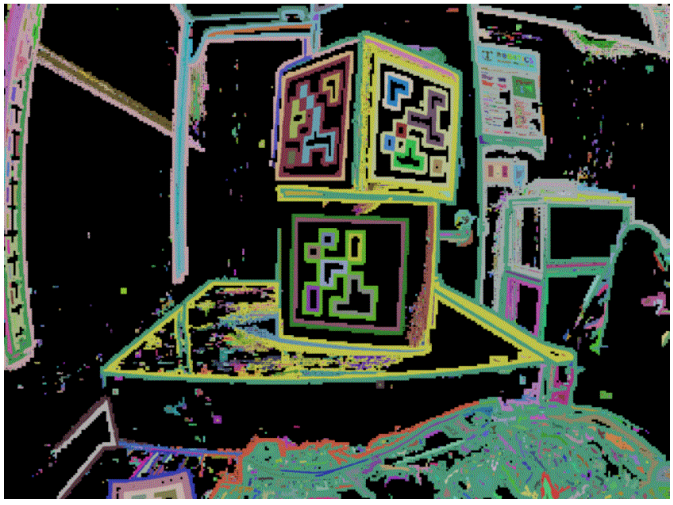
Adaptive Threshold



An adaptive threshold algorithm is run to classify each pixel as “definitely light”, “definitely dark”, or “not sure”.

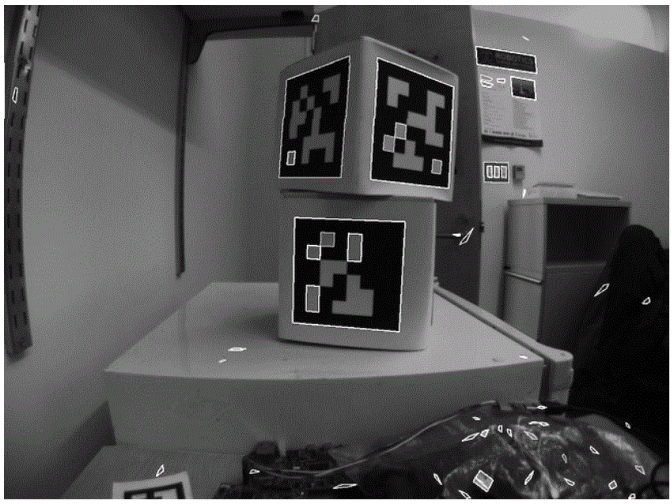
The threshold is calculated by looking at the pixel’s brightness, compared to a small neighborhood of pixels around it.

Segmentation



Next, the known pixels are clumped together. Any clumps which are too small to reasonably be a meaningful part of a tag are discarded.

Quad Detection



An algorithm for fitting a quadrilateral to each clump is now run:

- Identify likely “corner” candidates by pixels which are outliers in both dimensions.
- Iterate through all possible combinations of corners, evaluating the fit each time
- Pick the best-fit quadrilateral

Given the set of all quadralaterals, Identify a subset of quadrilaterals which is likely a tag.

A single large exterior quadrilateral with many interior quadrilateral is likely a good candidate.

If all has gone well so far, we are left with a four-sided region of pixels that is likely a valid tag.

Decode ID

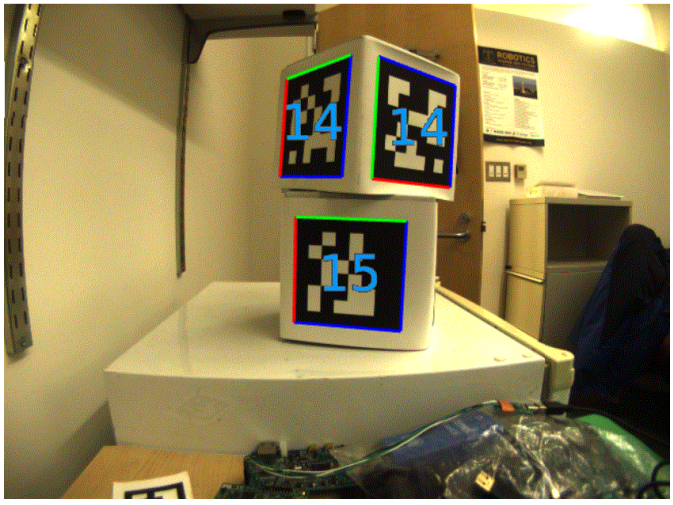


Now that we have one or more regions of pixels which we believe to be a valid AprilTag, we need to identify which tag we are looking at. This is done by "decoding" the pattern of light and dark squares on the inside.

- Calculate the expected interior pixel coordinates where the center of each bit should be
- Mark each location as "1" or "0" by comparing the pixel intensity to a threshold
- Find the tag ID which most closely matches what was seen in the image, allowing for one or two bit errors.

It is possible there is no valid tag ID which matches the suspect tag. In this case, the decoding process stops.

Fit External Quad



Now that we have a tag ID for the region of pixels, we need to do something useful with it.

For most FRC applications, we care about knowing the precise location of the corners of the tag, or its center. In both cases, we expect the resolution-lowering operation we did at the

beginning to have distorted the image, and we want to undo those effects.

The algorithm to do this is:

- Use the detected tag location to define a region of interest in the original-resolution image
- Calculate the *gradient* at pre-defined points in the region of interest to detect where the image most sharply transitions between black to white
- Use these gradient measurements to rapidly re-fit an exterior quadrilateral at full resolution
- Use geometry to calculate the exact center of the re-fit quadrilateral

Note that this step is optional, and can be skipped for faster image processing. However, skipping it can induce significant errors into your robot's behavior, depending on how you are using the tag outputs.

Usage

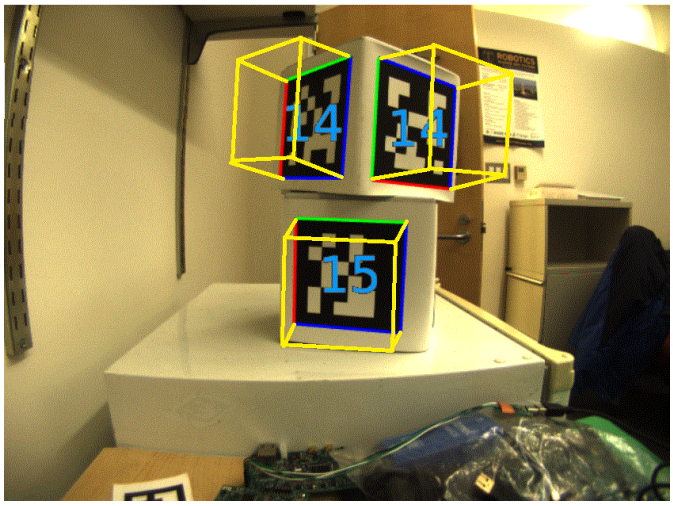
2D Alignment

A simple strategy for using targets is to move the robot until the target is centered in the image. Assuming the field and robot are constructed such that the gamepiece, scoring location, vision target, and camera are all aligned, this method should proved a straightforward method to automatically align the robot to the scoring position.

Using a camera, identify the *centroid* of the AprilTags in view. If the tag's ID is correct, apply drivetrain commands to rotate the robot left or right until the tag is centered in the camera image.

This method does not require calibrating the camera or performing the homography step.

3D Alignment



A more advanced usage of AprilTags is to use their corner locations to help perform *pose estimation*.

Each image is searched for AprilTags using the algorithm described on this page. Using assumptions about how the camera's lense distorts the 3d world onto the 2d array of pixels in the camera, an estimate of the camera's position relative to the tag is calculated. A good camera calibration is required for the assumptions about its lens behavior to be accurate.

The tag's ID is also decoded. from the image. Given each tag's ID, the position of the tag on the field can be looked up.

Knowing the position of the tag on the field, and the position of the camera relative to the tag, the 3D geometry classes can be used to estimate the position of the camera on the field.

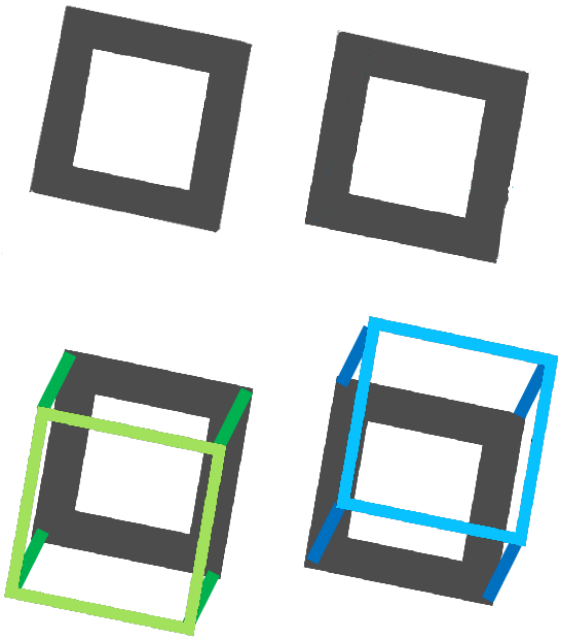
If the camera's position on the robot is known, the robot's position on the field can also be estimated.

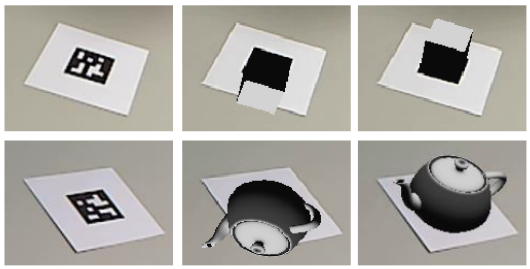
These estimates can be incorporated into the WPILib pose estimation classes.

2D to 3D Ambiguity

The process of translating the four known corners of the target in the image (two-dimensional) into a real-world position relative to the camera (three-dimensional) is inherently ambiguous. That is to say, there are multiple real-world positions that result in the target corners ending up in the same spot in the camera image.

Humans can often use lighting or background clues to understand how objects are oriented in space. However, computers do not have this benefit. They can be tricked by similar-looking targets:





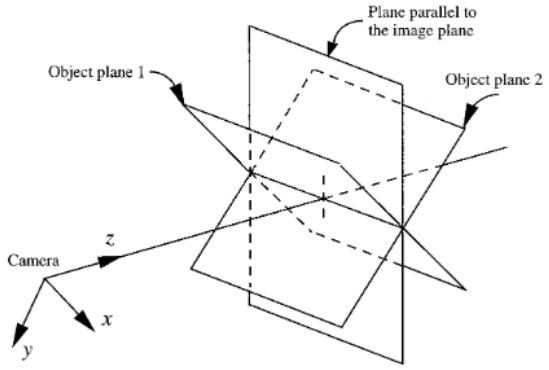


FIG. 4. Two object poses giving the same image under the SOP approximation.

Resolving which position is “correct” can be done in a few different ways:

1. Use your *odometry* history from all sensors to pick the pose closest to where you expect the robot to be.
2. Reject poses which are very unlikely (ex: outside the field perimeter, or up in the air)
3. Ignore pose estimates which are very close together (and hard to differentiate)
4. Use multiple cameras to look at the same target, such that at least one camera can generate a good pose estimate
5. Look at many targets at once, using each to generate multiple pose estimates. Discard the outlying estimates, use the ones which are tightly clustered together.

Adjustable Parameters

`Decimation factor` impacts how much the image is down-sampled before processing. Increasing it will increase detection speed, at the cost of not being able to see tags which are far away.

`Blur` applies smoothing to the input image to decrease noise, which increases speed when fitting quads to pixels, at the cost of precision. For most good cameras, this may be left at zero.

`Threads` changes the number of parallel threads which the algorithm uses to process the image. Certain steps may be sped up by allowing multithreading. In general, you want this to be approximately equal to the number of physical cores in your CPU, minus the number of cores which will be used for other processing tasks.

Detailed information about the tunable parameters can be found here.

Further Learning

The three major versions of AprilTags are described in three academic papers. It's recommended to read them in order, as each builds upon the previous:

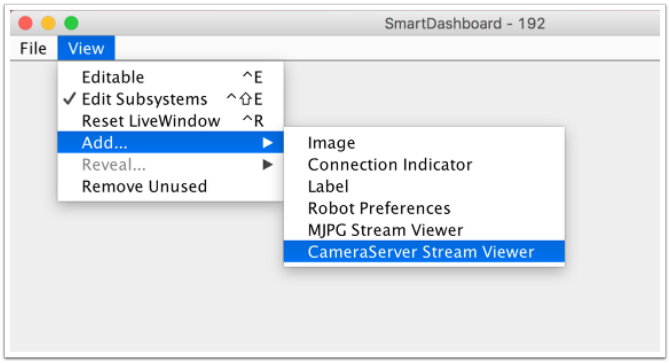
- `AprilTags v1`
- `AprilTags v2`
- `AprilTags v3`
- `Pose Ambiguity`

26.4 Vision on the RoboRIO

26.4.1 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

```
7  import edu.wpi.first.cameraserver.CameraServer;
8  import edu.wpi.first.wpilibj.TimedRobot;
9
10 /**
11  * Uses the CameraServer class to automatically capture video from a USB webcam and␣
   ↪send it to the
12  * FRC dashboard without doing any vision processing. This is the easiest way to get␣
   ↪camera images
13  * to the dashboard. Just add this to the robot class constructor.
14  */
15 public class Robot extends TimedRobot {
16   public Robot() {
17     CameraServer.startAutomaticCapture();
```

(continues on next page)

(continued from previous page)

```
  }
}
```

C++

```
#include <cameraserver/CameraServer.h>
#include <frc/TimedRobot.h>
class Robot : public frc::TimedRobot {
 public:
  Robot() {
    frc::CameraServer::StartAutomaticCapture();
  }
};

#ifndef RUNNING_FRC_TESTS
int main() {
  return frc::StartRobot<Robot>();
}
```

PYTHON

```
import wpilib
from wpilib.cameraserver import CameraServer


class MyRobot(wpilib.TimedRobot):
    """
    Uses the CameraServer class to automatically capture video from a USB webcam and send it to the
    FRC dashboard without doing any vision processing. This is the easiest way to get camera images
    to the dashboard. Just add this to the robotInit() method in your program.
    """

    def robotInit(self):
        CameraServer().launch()
```

Advanced Camera Server Program

In the following example a thread created in `Robot` constructor gets the Camera Server instance. Each frame of the video is individually processed, in this case drawing a rectangle on the image using the OpenCV `rectangle()` method. The resultant images are then passed to the output stream and sent to the dashboard. You can replace the `rectangle` 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

```
import edu.wpi.first.cameraserver.CameraServer;
import edu.wpi.first.cscore.CvSink;
import edu.wpi.first.cscore.CvSource;
import edu.wpi.first.cscore.UsbCamera;
import edu.wpi.first.wpilibj.TimedRobot;
import org.opencv.core.Mat;
import org.opencv.core.Point;
import org.opencv.core.Scalar;
import org.opencv.imgproc.Imgproc;

/**
 * This is a demo program showing the use of OpenCV to do vision processing. The image is acquired
 * from the USB camera, then a rectangle is put on the image and sent to the dashboard. OpenCV has
 * many methods for different types of processing.
 */
public class Robot extends TimedRobot {
  Thread m_visionThread;

  /** Called once at the beginning of the robot program. */
  public Robot() {
    m_visionThread =
        new Thread(
            () -> {
              // Get the UsbCamera from CameraServer
              UsbCamera camera = CameraServer.startAutomaticCapture();
              // Set the resolution
              camera.setResolution(640, 480);

              // Get a CvSink. This will capture Mats from the camera
              CvSink cvSink = CameraServer.getVideo();
              // Setup a CvSource. This will send images back to the Dashboard
              CvSource outputStream = CameraServer.putVideo("Rectangle", 640, 480);

              // Mats are very memory expensive. Lets reuse this Mat.
              Mat mat = new Mat();

              // This cannot be 'true'. The program will never exit if it is. This
              // lets the robot stop this thread when restarting robot code or
              // deploying.
              while (!Thread.interrupted()) {
                // Tell the CvSink to grab a frame from the camera and put it
                // in the source mat.  If there is an error notify the output.
                if (cvSink.grabFrame(mat) == 0) {
                  // Send the output the error.
                  outputStream.notifyError(cvSink.getError());
                  // skip the rest of the current iteration
                  continue;
                }
                // Put a rectangle on the image
                Imgproc.rectangle(
                    mat, new Point(100, 100), new Point(400, 400), new Scalar(255, 255, 255), 5);
                // Give the output stream a new image to display
```

(continues on next page)

(continued from previous page)

```
                outputStream.putFrame(mat);
              }
            });
    m_visionThread.setDaemon(true);
    m_visionThread.start();
  }
}
```

c++

```
#include <cstdio>
#include <thread>

#include <cameraserver/CameraServer.h>
#include <frc/TimedRobot.h>
#include <opencv2/core/core.hpp>
#include <opencv2/core/types.hpp>
#include <opencv2/imgproc/imgproc.hpp>

/**
 * This is a demo program showing the use of OpenCV to do vision processing. The
 * image is acquired from the USB camera, then a rectangle is put on the image
 * and sent to the dashboard. OpenCV has many methods for different types of
 * processing.
 */
class Robot : public frc::TimedRobot {
 public:
  Robot() {
    // We need to run our vision program in a separate thread. If not, our robot
    // program will not run.
    std::thread visionThread(VisionThread);
    visionThread.detach();
  }
 private:
  static void VisionThread() {
    // Get the USB camera from CameraServer
    cs::UsbCamera camera = frc::CameraServer::StartAutomaticCapture();
    // Set the resolution
    camera.SetResolution(640, 480);

    // Get a CvSink. This will capture Mats from the Camera
    cs::CvSink cvSink = frc::CameraServer::GetVideo();
    // Setup a CvSource. This will send images back to the Dashboard
    cs::CvSource outputStream =
        frc::CameraServer::PutVideo("Rectangle", 640, 480);

    // Mats are very memory expensive. Lets reuse this Mat.
    cv::Mat mat;

    while (true) {
      // Tell the CvSink to grab a frame from the camera and
      // put it
      // in the source mat.  If there is an error notify the
      // output.
```

(continues on next page)

(continued from previous page)

```
      if (cvSink.GrabFrame(mat) == 0) {
        // Send the output the error.
        outputStream.NotifyError(cvSink.GetError());
        // skip the rest of the current iteration
        continue;
      }
      // Put a rectangle on the image
      rectangle(mat, cv::Point(100, 100), cv::Point(400, 400),
                cv::Scalar(255, 255, 255), 5);
      // Give the output stream a new image to display
      outputStream.PutFrame(mat);
    }
  }
};

#ifndef RUNNING_FRC_TESTS
int main() {
  return frc::StartRobot<Robot>();
}
#endif
```

PYTHON

Image processing on the roboRIO when using Python is slightly different from C++/Java. Instead of using a separate thread, we need to launch the image processing code in a completely separate process.

> **Warning**
>
> Image processing is a CPU intensive task, and because of the Python Global Interpreter Lock (GIL) **we do NOT recommend using cscore directly in your robot process**. Don't do it. Really.
>
> For more information on the GIL and its effects, you may wish to read the following resources:
>
> - Python Wiki: Global Interpreter Lock
> - Efficiently Exploiting Multiple Cores with Python

This introduces a number of rules that your image processing code must follow to efficiently and safely run on the RoboRIO:

- Your image processing code must be in its own file. It's easiest to just place it next to your `robot.py`
- Never import the `cscore` package from your robot code, it will just waste memory
- Never import the `wpilib` or `hal` packages from your image processing file
- The camera code will be killed when the `robot.py` program exits. If you wish to perform cleanup, you should register an atexit handler.
- `robotpy-cscore` is not installed on the roboRIO by default, you need to update your `pyproject.toml` file to install it

> **Warning**
>
> `wpilib` may not be imported from two programs on the RoboRIO. If this happens, the second program will attempt to kill the first program.

Here's what your `robot.py` needs to contain to launch the image processing process:

```
import wpilib


class MyRobot(wpilib.TimedRobot):
    """
    This is a demo program showing the use of OpenCV to do vision processing. The␣
↪image is acquired
    from the USB camera, then a rectangle is put on the image and sent to the␣
↪dashboard. OpenCV has
    many methods for different types of processing.
    """

    def robotInit(self):
        wpilib.CameraServer.launch("vision.py:main")
```

The `launch("vision.py")` function says to launch `vision.py` and call the `run` function in that file. Here's what is in `vision.py`:

```
# NOTE: This code runs in its own process, so we cannot access the robot here,
#       nor can we create/use/see wpilib objects
#
# To try this code out locally (if you have robotpy-cscore installed), you
# can execute `python3 -m cscore vision.py:main`
#

import cv2
import numpy as np

from cscore import CameraServer as CS


def main():
    CS.enableLogging()

    # Get the UsbCamera from CameraServer
    camera = CS.startAutomaticCapture()
    # Set the resolution
    camera.setResolution(640, 480)

    # Get a CvSink. This will capture images from the camera
    cvSink = CS.getVideo()
    # Setup a CvSource. This will send images back to the Dashboard
    outputStream = CS.putVideo("Rectangle", 640, 480)

    # Allocating new images is very expensive, always try to preallocate
    mat = np.zeros(shape=(480, 640, 3), dtype=np.uint8)

    while True:
        # Tell the CvSink to grab a frame from the camera and put it
```

(continues on next page)

(continued from previous page)

```
        # in the source image.  If there is an error notify the output.
        time, mat = cvSink.grabFrame(mat)
        if time == 0:
            # Send the output the error.
            outputStream.notifyError(cvSink.getError())
            # skip the rest of the current iteration
            continue

        # Put a rectangle on the image
        cv2.rectangle(mat, (100, 100), (400, 400), (255, 255, 255), 5)

        # Give the output stream a new image to display
        outputStream.putFrame(mat)
```

You need to update `pyproject.toml` contents to include cscore in the components list (this only shows the portions you need to update):

```
[tool.robotpy]
...
# Add cscore to the components list
components = ["cscore"]
```

Notice that in these examples, the `PutVideo()` method writes the video to a named stream. To view that stream on SmartDashboard or Shuffleboard, select that named stream. In this case that is "Rectangle".

26.4.2 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

```
UsbCamera camera1;
UsbCamera camera2;
Joystick joy1 = new Joystick(0);
NetworkTableEntry cameraSelection;

public Robot() {
    camera1 = CameraServer.startAutomaticCapture(0);
    camera2 = CameraServer.startAutomaticCapture(1);
    cameraSelection = NetworkTableInstance.getDefault().getTable("").getEntry(
↪"CameraSelection");
}
@Override
public void teleopPeriodic() {
```

(continues on next page)

(continued from previous page)

```
    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());
    }
}
```

C++

```
cs::UsbCamera camera1;
cs::UsbCamera camera2;
frc::Joystick joy1{0};
nt::NetworkTableEntry cameraSelection;
Robot() {
  camera1 = frc::CameraServer::StartAutomaticCapture(0);
  camera2 = frc::CameraServer::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());
  }
}
```

PYTHON

> **Note**
>
> Python requires you to place your image processing code in a separate file from your robot code. You can create `robot.py` and `vision.py` in the same directory.

`robot.py` contents:

```
import wpilib
from ntcore import NetworkTableInstance
class MyRobot(wpilib.TimedRobot):
    def robotInit(self):
        self.joy1 = wpilib.Joystick(0)
        self.cameraSelection = NetworkTableInstance.getDefault().getTable("").
↪getEntry("CameraSelection")
        wpilib.CameraServer.launch("vision.py:main")
    def teleopPeriodic(self):
        if self.joy1.getTriggerPressed():
```

(continues on next page)

(continued from previous page)

```
            print("Setting camera 2")
            self.cameraSelection.setString("USB Camera 1")
        elif self.joy1.getTriggerReleased():
            print("Setting camera 1")
            self.cameraSelection.setString("USB Camera 0")
```

`vision.py` contents:

```
from cscore import CameraServer
def main():
    CameraServer.enableLogging()
    camera1 = CameraServer.startAutomaticCapture(0)
    camera2 = CameraServer.startAutomaticCapture(1)
    CameraServer.waitForever()
```

`pyproject.toml` contents (this only shows the portions you need to update):

```
[tool.robotpy]
...
# Add cscore to the components list
components = ["cscore"]
```

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

```
UsbCamera camera1;
UsbCamera camera2;
VideoSink server;
Joystick joy1 = new Joystick(0);

public Robot() {
    camera1 = CameraServer.startAutomaticCapture(0);
    camera2 = CameraServer.startAutomaticCapture(1);
    server = CameraServer.getServer();
}
@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);
    }
}
```

C++

```
cs::UsbCamera camera1;
cs::UsbCamera camera2;
cs::VideoSink server;
frc::Joystick joy1{0};
bool prevTrigger = false;
Robot() {
  camera1 = frc::CameraServer::StartAutomaticCapture(0);
  camera2 = frc::CameraServer::StartAutomaticCapture(1);
  server = frc::CameraServer::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();
}
```

PYTHON

```
# Setting the source directly via joystick isn't possible in Python, you
# should use NetworkTables as shown above instead
```

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

```
UsbCamera camera1;
UsbCamera camera2;
VideoSink server;
Joystick joy1 = new Joystick(0);

public Robot() {
    camera1 = CameraServer.startAutomaticCapture(0);
    camera2 = CameraServer.startAutomaticCapture(1);
    server = CameraServer.getServer();
    camera1.setConnectionStrategy(ConnectionStrategy.kKeepOpen);
    camera2.setConnectionStrategy(ConnectionStrategy.kKeepOpen);
}
@Override
```

(continues on next page)

(continued from previous page)

```
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);
    }
}
```

C++

```
cs::UsbCamera camera1;
cs::UsbCamera camera2;
cs::VideoSink server;
frc::Joystick joy1{0};
bool prevTrigger = false;
Robot() {
  camera1 = frc::CameraServer::StartAutomaticCapture(0);
  camera2 = frc::CameraServer::StartAutomaticCapture(1);
  server = frc::CameraServer::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();
}
```

PYTHON

> **Note**
>
> Python requires you to place your image processing code in a separate file from your robot code. You can create `robot.py` and `vision.py` in the same directory.

`robot.py` contents:

```
import wpilib
from ntcore import NetworkTableInstance
class MyRobot(wpilib.TimedRobot):
    def robotInit(self):
```

(continues on next page)

(continued from previous page)

```
        self.joy1 = wpilib.Joystick(0)
        self.cameraSelection = NetworkTableInstance.getDefault().getTable("").
↪getEntry("CameraSelection")
        wpilib.CameraServer.launch("vision.py:main")
    def teleopPeriodic(self):
        if self.joy1.getTriggerPressed():
            print("Setting camera 2")
            self.cameraSelection.setString("USB Camera 1")
        elif self.joy1.getTriggerReleased():
            print("Setting camera 1")
            self.cameraSelection.setString("USB Camera 0")
```

`vision.py` contents:

```
from cscore import CameraServer, VideoSource
    def main():
    CameraServer.enableLogging()
    camera1 = CameraServer.startAutomaticCapture(0)
    camera2 = CameraServer.startAutomaticCapture(1)
    camera1.setConnectionStrategy(VideoSource.ConnectionStrategy.kConnectionKeepOpen)
    camera2.setConnectionStrategy(VideoSource.ConnectionStrategy.kConnectionKeepOpen)
        CameraServer.waitForever()
```

`pyproject.toml` contents (this only shows the portions you need to update):

```
[tool.robotpy]
...
# Add cscore to the components list
components = ["cscore"]
```

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
(VIDIOC_STREAMON: No space left on device)
```

If you're using Option 3 it will give you this error during the `Robot` constructor. 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.

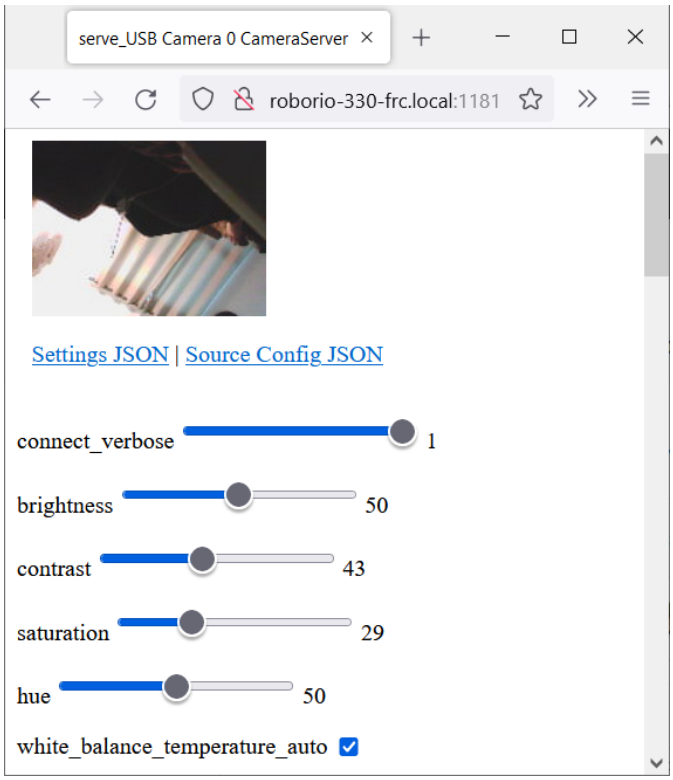
26.4.3 CameraServer Web Interface

When CameraServer opens a camera, it creates a webpage that you can use to view the camera stream and view the effects of various camera settings. To connect to the web interface, use a web browser to navigate to `http://roboRIO-TEAM-frc.local:1181`. There is no additional code needed other than the *Simple CameraServer Program*.

Note

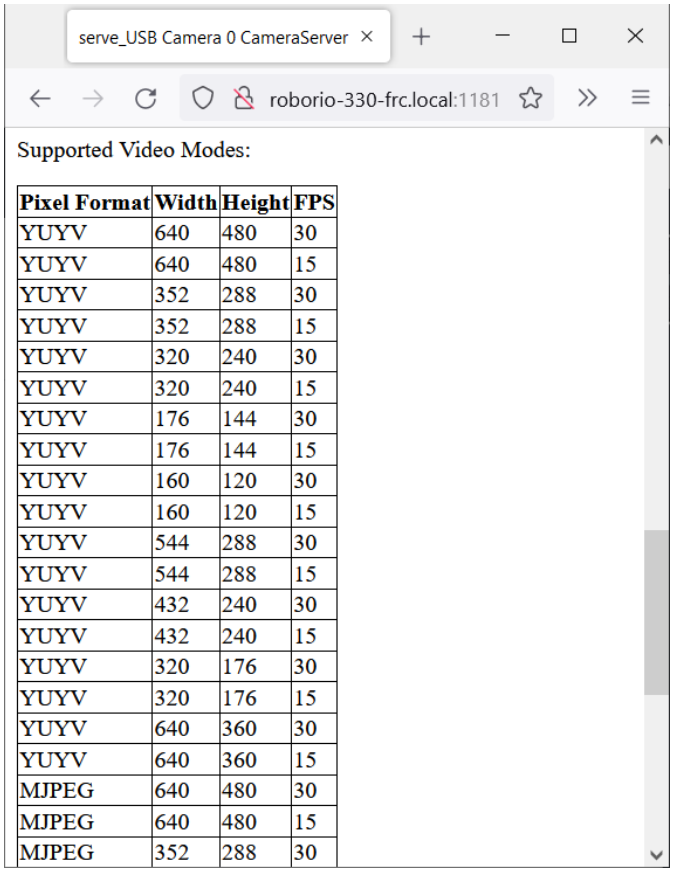
The port 1181 is used for the first camera. The port increments for additional camera, so if you have two cameras, the replace 1181 above with 1182.

Camera Settings



The web server will show a live camera image and has sliders to adjust various camera settings, such as brightness, contrast, sharpness, and many other options. You can adjust the values and see the results live, and then use the `VideoCamera` class to set those in your robot code.

Camera Video Modes



Pixel Format	Width	Height	FPS
YUYV	640	480	30
YUYV	640	480	15
YUYV	352	288	30
YUYV	352	288	15
YUYV	320	240	30
YUYV	320	240	15
YUYV	176	144	30
YUYV	176	144	15
YUYV	160	120	30
YUYV	160	120	15
YUYV	544	288	30
YUYV	544	288	15
YUYV	432	240	30
YUYV	432	240	15
YUYV	320	176	30
YUYV	320	176	15
YUYV	640	360	30
YUYV	640	360	15
MJPEG	640	480	30
MJPEG	640	480	15
MJPEG	352	288	30

One useful feature is the list of supported video modes at the bottom of the web page. This shows all the supported modes that the camera supports to enable you to choose the one that is the best combination of resolution and frame rate for your requirements.

27 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*.

27.1 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 one possible *design pattern* for robot software. 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 reuse from year to year.

The command-based paradigm is also an example of *declarative programming*. The command-based library allow users to define desired robot behaviors while minimizing the amount of iteration-by-iteration robot logic that they must write. For example, in the command-based program, a user can specify that "the robot should perform an action when a condition is true" (note the use of a *lambda*):

JAVA

```
new Trigger(condition::get).onTrue(Commands.runOnce(() -> piston.set(DoubleSolenoid.
↪Value.kForward)));
```

C++

```
Trigger([&condition] { return condition.Get(); }).OnTrue(frc2::cmd::RunOnce([&piston]
↪{ piston.Set(frc::DoubleSolenoid::kForward); }));
```

PYTHON

```
Trigger(condition.get).onTrue(Commands.runOnce(lambda: piston.set(DoubleSolenoid.
↪Value.kForward)))
```

In contrast, without using command-based, the user would need to check the button state every iteration, and perform the appropriate action based on the state of the button.

JAVA

```
if(condition.get()) {
  if(!pressed) {
    piston.set(DoubleSolenoid.Value.kForward);
    pressed = true;
  }
} else {
  pressed = false;
}
```

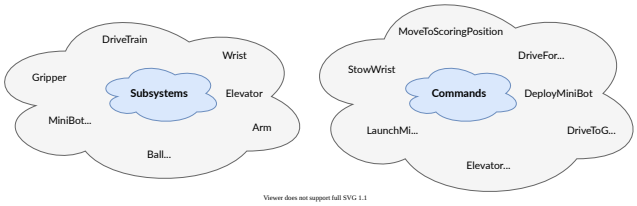
C++

```
if(condition.Get()) {
  if(!pressed) {
    piston.Set(frc::DoubleSolenoid::kForward);
    pressed = true;
  }
} else {
  pressed = false;
}
```

PYTHON

```
if condition.get():
    if not pressed:
        piston.set(DoubleSolenoid.Value.kForward)
        pressed = True
    else:
        pressed = False
```

27.1.1 Subsystems and Commands



The command-based pattern is based around two core abstractions: **commands**, and **subsystems.**

Commands represent actions the robot can take. Commands run when scheduled, until they are interrupted or their end condition is met. Commands are very recursively composable: commands can be composed to accomplish more-complicated tasks. See *Commands* for more info.

Subsystems represent independently-controlled collections of robot hardware (such as motor controllers, sensors, pneumatic actuators, etc.) that operate together. Subsystems back the resource-management system of command-based: only one command can use a given subsystem at the same time. 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's hardware without also changing the rest of the robot code.

27.1.2 How Commands Are Run

> **Note**
>
> For a more detailed explanation, see *The Command Scheduler*.

Commands are run by the `CommandScheduler` (Java, C++, `Python`) singleton, which polls triggers (such as buttons) for commands to schedule, preventing resource conflicts, and executing scheduled commands. The scheduler's `run()` method must be called; 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 specify which subsystems they interact with, and the scheduler will ensure that no more more than one command requiring a given subsystem is scheduled 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.

27.1.3 Command Compositions

It is often desirable to build complex commands from simple pieces. This is achievable by creating a *composition* of commands. The command-based library provides several types of *command compositions* for teams to use, and users may write their own. As command compositions are commands themselves, they may be used in a *recursive composition*. That is to say - one can create a command compositions from multiple command compositions. This provides an extremely powerful way of building complex robot actions from simple components.

27.2 Commands

Commands represent actions the robot can take. Commands run when scheduled, until they are interrupted or their end condition is met. Commands are represented in the command-based library by the `Command` class (Java, C++) or the `Command` class in `commands2` library (`Python`).

27.2.1 The Structure of a Command

Commands 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 finishes naturally, and will run until interrupted).

Initialization

The `initialize()` method (Java, C++, `Python`) marks the command start, and is called exactly once per time a command is scheduled. The `initialize()` method should be used to place the command in a known starting state for execution. Command objects may be reused and scheduled multiple times, so any state or resources needed for the command's functionality should be initialized or opened in `initialize` (which will be called at the start of each use) rather than the constructor (which is invoked only once on object allocation). 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++, `Python`) is called repeatedly while the command is scheduled; this is when 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(bool interrupted)` method (Java, C++, Python) 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. Any state or resources initialized in `initialize()` should be closed in `end()`.

Specifying end conditions

The `isFinished()` method (Java, C++, Python) 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 ends. The `isFinished()` method is called after the `execute()` method, so the command will execute once on the same iteration that it ends.

27.2.2 Command Properties

In addition to the four lifecycle methods described above, each `Command` also has three properties, defined by getter methods that should always return the same value with no side affects.

getRequirements

Each command should declare any subsystems it controls as requirements. This backs the scheduler's resource management mechanism, ensuring that no more than one command requires a given subsystem at the same time. This prevents situations such as two different pieces of code attempting to set the same motor controller to different output values.

Declaring requirements is done by overriding the `getRequirements()` method in the relevant command class, by calling `addRequirements()`, or by using the `requirements` vararg (Java) / `Requirements` struct (C++) parameter / `requirements` argument (Python) at the end of the parameter list of most command constructors and factories in the library:

JAVA

```
Commands.run(intake::activate, intake);
```

C++

```
frc2::cmd::Run([&intake] { intake.Activate(); }, {&intake});
```

PYTHON

```
commands2.cmd.run(intake.activate, intake)
```

As a rule, command compositions require all subsystems their components require.

runsWhenDisabled

The `runsWhenDisabled()` method (Java, C++, Python) returns a `boolean`/`bool` specifying whether the command may run when the robot is disabled. With the default of returning `false`, the command will be canceled when the robot is disabled and attempts to schedule it will do nothing. Returning `true` will allow the command to run and be scheduled when the robot is disabled.

Important

When the robot is disabled, *PWM* outputs are disabled and CAN motor controllers may not apply voltage, regardless of `runsWhenDisabled`!

This property can be set either by overriding the `runsWhenDisabled()` method in the relevant command class, or by using the `ignoringDisable` decorator (Java, C++, Python):

JAVA

```
Command mayRunDuringDisabled = Commands.run(() -> updateTelemetry()).
↪ignoringDisable(true);
```

C++

```
frc2::CommandPtr mayRunDuringDisabled = frc2::cmd::Run([] { UpdateTelemetry(); }).
↪IgnoringDisable(true);
```

PYTHON

```
may_run_during_disabled = commands2.cmd.run(lambda: update_telemetry()).ignoring_
↪disable(True)
```

As a rule, command compositions may run when disabled if all their component commands set `runsWhenDisabled` as `true`.

getInterruptionBehavior

The `getInterruptionBehavior()` method (Java, C++, Python) defines what happens if another command sharing a requirement is scheduled while this one is running. In the default behavior, `kCancelSelf`, the current command will be canceled and the incoming command will be scheduled successfully. If `kCancelIncoming` is returned, the incoming command's scheduling will be aborted and this command will continue running. Note that `getInterruptionBehavior` only affects resolution of requirement conflicts: all commands can be canceled, regardless of `getInterruptionBehavior`.

> **Note**
>
> This was previously controlled by the `interruptible` parameter passed when scheduling a command, and is now a property of the command object.

This property can be set either by overriding the `getInterruptionBehavior` method in the relevant command class, or by using the *withInterruptBehavior()* decorator (Java, C++, Python)

JAVA

```
Command noninteruptible = Commands.run(intake::activate, intake).
↪withInterruptBehavior(Command.InterruptBehavior.kCancelIncoming);
```

C++

```
frc2::CommandPtr noninterruptible = frc2::cmd::Run([&intake] { intake.Activate(); },
↪{&intake}).WithInterruptBehavior(Command::InterruptBehavior::kCancelIncoming);
```

PYTHON

```
non_interruptible = commands2.cmd.run(intake.activate, intake).with_interrupt_
↪behavior(Command.InterruptBehavior.kCancelIncoming)
```

As a rule, command compositions are `kCancelIncoming` if all their components are `kCancelIncoming` as well.

27.2.3 Included Command Types

The command-based library includes many pre-written command types. Through the use of *lambdas*, these commands can cover almost all use cases and teams should rarely need to write custom command classes. Many of these commands are provided via static factory functions in the `Commands` utility class (Java), in the `frc2::cmd` namespace defined in the `Commands.h` header (C++), or in the `commands2.cmd` namespace (Python). In Java and C++, classes inheriting from `Subsystem` also have instance methods that implicitly require `this`.

Running Actions

The most basic commands are actions the robot takes: setting voltage to a motor, changing a solenoid's direction, etc. For these commands, which typically consist of a method call or two, the command-based library offers several factories to be construct commands inline with one or more lambdas to be executed.

The `runOnce` factory, backed by the `InstantCommand` (Java, C++, Python) class, creates a command that calls a lambda once, and then finishes.

Java

```
  /** Grabs the hatch. */
  public Command grabHatchCommand() {
    // implicitly require `this`
    return this.runOnce(() -> m_hatchSolenoid.set(kForward));
  }

  /** Releases the hatch. */
  public Command releaseHatchCommand() {
    // implicitly require `this`
    return this.runOnce(() -> m_hatchSolenoid.set(kReverse));
  }
```

C++ (Header)

```
  /**
   * Grabs the hatch.
   */
  frc2::CommandPtr GrabHatchCommand();

  /**
   * Releases the hatch.
   */
  frc2::CommandPtr ReleaseHatchCommand();
```

C++ (Source)

```
frc2::CommandPtr HatchSubsystem::GrabHatchCommand() {
  // implicitly require `this`
  return this->RunOnce(
      [this] { m_hatchSolenoid.Set(frc::DoubleSolenoid::kForward); });
}

frc2::CommandPtr HatchSubsystem::ReleaseHatchCommand() {
  // implicitly require `this`
  return this->RunOnce(
      [this] { m_hatchSolenoid.Set(frc::DoubleSolenoid::kReverse); });
}
```

Python

```
    def grabHatch(self) -> commands2.Command:
        """Grabs the hatch"""
        return commands2.cmd.runOnce(
            lambda: self.hatchSolenoid.set(wpilib.DoubleSolenoid.Value.kForward), self
        )

    def releaseHatch(self) -> commands2.Command:
        """Releases the hatch"""
        return commands2.cmd.runOnce(
            lambda: self.hatchSolenoid.set(wpilib.DoubleSolenoid.Value.kReverse), self
        )
```

The `run` factory, backed by the `RunCommand` (Java, C++, Python) class, creates a command that calls a lambda repeatedly, until interrupted.

JAVA

```
// 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.getLeftY(),
    driverController.getRightX()),
    m_robotDrive)
```

C++

```
// 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.GetLeftY(),
        m_driverController.GetRightX());
  },
  {&m_drive})
```

PYTHON

```
# A split-stick arcade command, with forward/backward controlled by the left
# hand, and turning controlled by the right.
commands2.cmd.run(lambda: robot_drive.arcade_drive(
    -driver_controller.get_left_y(),
    driver_controller.get_right_x()),
    robot_drive)
```

The `startEnd` factory, backed by the `StartEndCommand` (Java, C++, Python) class, calls one lambda when scheduled, and then a second lambda when interrupted.

JAVA

```
Commands.startEnd(
    // 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
)
```

C++

```
frc2::cmd::StartEnd(
  // 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}
)
```

PYTHON

```
commands2.cmd.start_end(
   # Start a flywheel spinning at 50% power
   lambda: shooter.shooter_speed(0.5),
   # Stop the flywheel at the end of the command
   lambda: shooter.shooter_speed(0.0),
   # Requires the shooter subsystem
   shooter)
```

`FunctionalCommand` (Java, C++, Python) accepts four lambdas that constitute the four command lifecycle methods: a `Runnable/std::function<void()>/Callable` for each of `initialize()` and `execute()`, a `BooleanConsumer/std::function<void(bool)>/Callable[bool,[]]` for `end()`, and a `BooleanSupplier/std::function<bool()>/Callable[[],bool]` for `isFinished()`.

JAVA

```
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
```

(continues on next page)

(continued from previous page)

```
    m_robotDrive
)
```

C++

```
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}
)
```

PYTHON

```
commands2.cmd.functional_command(
  # Reset encoders on command start
  lambda: robot_drive.reset_encoders(),
  # Start driving forward at the start of the command
  lambda: robot_drive.arcade_drive(ac.kAutoDriveSpeed, 0),
  # Stop driving at the end of the command
  lambda interrupted: robot_drive.arcade_drive(0, 0),
  # End the command when the robot's driven distance exceeds the desired value
  lambda: robot_drive.get_average_encoder_distance() >= ac.kAutoDriveDistanceInches,
  # Require the drive subsystem
  robot_drive)
```

To print a string and ending immediately, the library offers the `Commands.print(String)`/`frc2::cmd::Print(std::string_view)`/`commands2.cmd.print(String)` factory, backed by the `PrintCommand` (Java, C++, Python) subclass of `InstantCommand`.

Waiting

Waiting for a certain condition to happen or adding a delay can be useful to synchronize between different commands in a command composition or between other robot actions.

To wait and end after a specified period of time elapses, the library offers the `Commands.waitSeconds(double)`/`frc2::cmd::Wait(units::second_t)`/`commands2.cmd.wait(float)` factory, backed by the `WaitCommand` (Java, C++, Python) class.

JAVA

```
// Ends 5 seconds after being scheduled
new WaitCommand(5.0)
```

C++

```
// Ends 5 seconds after being scheduled
frc2::WaitCommand(5.0_s)
```

PYTHON

```
# Ends 5 seconds after being scheduled
commands2.cmd.wait(5.0)
```

To wait until a certain condition becomes `true`, the library offers the `Commands.waitUntil(BooleanSupplier)`/`frc2::cmd::WaitUntil(std::function<bool()>)` factory, backed by the `WaitUntilCommand` class (Java, C++, Python).

JAVA

```
// Ends after m_limitSwitch.get() returns true
new WaitUntilCommand(m_limitSwitch::get)
```

C++

```
// Ends after m_limitSwitch.Get() returns true
frc2::WaitUntilCommand([&m_limitSwitch] { return m_limitSwitch.Get(); })
```

PYTHON

```
# Ends after limit_switch.get() returns True
commands2.cmd.wait_until(limit_switch.get)
```

Control Algorithm Commands

There are commands for various control setups:

- `TrapezoidProfile` tracks a trapezoid motion profile. For more info, see *Motion Profiling in Command-based*.
- `MecanumControllerCommand` (Java, C++) is useful for controlling mecanum drivetrains. See API docs and the **MecanumControllerCommand** (Java, C++) example project for more info.

- `SwerveControllerCommand` (Java, C++) is useful for controlling swerve drivetrains. See API docs and the **SwerveControllerCommand** (Java, C++) example project for more info.
- `RamseteCommand` (Java, C++) is useful for path following with differential drivetrains ("tank drive"). See API docs and the *Trajectory Tutorial* for more info.

27.2.4 Custom Command Classes

Users may also write custom command classes. As this is significantly more verbose, it's recommended to use the more concise factories mentioned above.

> **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.

To write a custom command class, subclass the abstract `Command` class (Java) or `CommandHelper` (C++), as seen in the command-based template (Java, C++):

JAVA

```
import edu.wpi.first.wpilibj.templates.commandbased.subsystems.ExampleSubsystem;
import edu.wpi.first.wpilibj2.command.Command;

/** An example command that uses an example subsystem. */
public class ExampleCommand extends Command {
  @SuppressWarnings("PMD.UnusedPrivateField")
  private final ExampleSubsystem m_subsystem;

  /**
   * Creates a new ExampleCommand.
   *
   * @param subsystem The subsystem used by this command.
   */
  public ExampleCommand(ExampleSubsystem subsystem) {
    m_subsystem = subsystem;
    // Use addRequirements() here to declare subsystem dependencies.
    addRequirements(subsystem);
  }
```

C++

```
#pragma once

#include <frc2/command/Command.h>
#include <frc2/command/CommandHelper.h>

#include "subsystems/ExampleSubsystem.h"

/**
 * An example command that uses an example subsystem.
 *
 * <p>Note that this extends CommandHelper, rather extending Command
 * directly; this is crucially important, or else the decorator functions in
 * Command will *not* work!
 */
class ExampleCommand
    : public frc2::CommandHelper<frc2::Command, ExampleCommand> {
 public:
  /**
   * Creates a new ExampleCommand.
   *
   * @param subsystem The subsystem used by this command.
   */
  explicit ExampleCommand(ExampleSubsystem* subsystem);

 private:
  ExampleSubsystem* m_subsystem;
};
```

27.2.5 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`:

Java

```
package edu.wpi.first.wpilibj.examples.hatchbottraditional.commands;

import edu.wpi.first.wpilibj.examples.hatchbottraditional.subsystems.HatchSubsystem;
import edu.wpi.first.wpilibj2.command.Command;

/**
 * A simple command that grabs a hatch with the {@link HatchSubsystem}. Written␣
↪explicitly for
 * pedagogical purposes. Actual code should inline a command this simple with {@link
 * edu.wpi.first.wpilibj2.command.InstantCommand}.
 */
public class GrabHatch extends Command {
  // The subsystem the command runs on
  private final HatchSubsystem m_hatchSubsystem;

  public GrabHatch(HatchSubsystem subsystem) {
```

(continues on next page)

(continued from previous page)

```
    m_hatchSubsystem = subsystem;
    addRequirements(m_hatchSubsystem);
  }

  @Override
  public void initialize() {
    m_hatchSubsystem.grabHatch();
  }

  @Override
  public boolean isFinished() {
    return true;
  }
}
```

C++ (Header)

```
#pragma once

#include <frc2/command/Command.h>
#include <frc2/command/CommandHelper.h>

#include "subsystems/HatchSubsystem.h"

/**
 * A simple command that grabs a hatch with the HatchSubsystem.  Written
 * explicitly for pedagogical purposes.  Actual code should inline a command
 * this simple with InstantCommand.
 *
 * @see InstantCommand
 */
class GrabHatch : public frc2::CommandHelper<frc2::Command, GrabHatch> {
 public:
  explicit GrabHatch(HatchSubsystem* subsystem);

  void Initialize() override;

  bool IsFinished() override;

 private:
  HatchSubsystem* m_hatch;
};
```

C++ (Source)

```cpp
#include "commands/GrabHatch.h"

GrabHatch::GrabHatch(HatchSubsystem* subsystem) : m_hatch(subsystem) {
  AddRequirements(subsystem);
}

void GrabHatch::Initialize() {
  m_hatch->GrabHatch();
}

bool GrabHatch::IsFinished() {
  return true;
}
```

Python

```python
import commands2
from subsystems.hatchsubsystem import HatchSubsystem


class GrabHatch(commands2.Command):
    def __init__(self, hatch: HatchSubsystem) -> None:
        super().__init__()
        self.hatch = hatch
        self.addRequirements(hatch)

    def initialize(self) -> None:
        self.hatch.grabHatch()

    def isFinished(self) -> bool:
        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 as such it would be more succinct to write this command using the factories described above.

What about a more complicated case? Below is a drive command, from the same example project:

Java

```java
package edu.wpi.first.wpilibj.examples.hatchbottraditional.commands;

import edu.wpi.first.wpilibj.examples.hatchbottraditional.subsystems.DriveSubsystem;
import edu.wpi.first.wpilibj2.command.Command;
import java.util.function.DoubleSupplier;

/**
 * A command to drive the robot with joystick input (passed in as {@link DoubleSupplier}s). Written
 * explicitly for pedagogical purposes - actual code should inline a command this simple with {@link
 * edu.wpi.first.wpilibj2.command.RunCommand}.
 */
public class DefaultDrive extends Command {
  private final DriveSubsystem m_drive;
  private final DoubleSupplier m_forward;
  private final DoubleSupplier m_rotation;

  /**
   * Creates a new DefaultDrive.
   *
   * @param subsystem The drive subsystem this command wil run on.
   * @param forward The control input for driving forwards/backwards
   * @param rotation The control input for turning
   */
  public DefaultDrive(DriveSubsystem subsystem, DoubleSupplier forward, DoubleSupplier rotation) {
    m_drive = subsystem;
    m_forward = forward;
    m_rotation = rotation;
    addRequirements(m_drive);
  }

  @Override
  public void execute() {
    m_drive.arcadeDrive(m_forward.getAsDouble(), m_rotation.getAsDouble());
  }
}
```

C++ (Header)

```cpp
#pragma once

#include <functional>

#include <frc2/command/Command.h>
#include <frc2/command/CommandHelper.h>

#include "subsystems/DriveSubsystem.h"

/**
 * A command to drive the robot with joystick input passed in through lambdas.
 * Written explicitly for pedagogical purposes - actual code should inline a
```

(continues on next page)

(continued from previous page)

```
 * command this simple with RunCommand.
 *
 * @see RunCommand
 */
class DefaultDrive : public frc2::CommandHelper<frc2::Command, DefaultDrive> {
 public:
  /**
   * Creates a new DefaultDrive.
   *
   * @param subsystem The drive subsystem this command wil run on.
   * @param forward The control input for driving forwards/backwards
   * @param rotation The control input for turning
   */
  DefaultDrive(DriveSubsystem* subsystem, std::function<double()> forward,
               std::function<double()> rotation);

  void Execute() override;

 private:
  DriveSubsystem* m_drive;
  std::function<double()> m_forward;
  std::function<double()> m_rotation;
};
```

C++ (Source)

```
#include "commands/DefaultDrive.h"

#include <utility>

DefaultDrive::DefaultDrive(DriveSubsystem* subsystem,
                           std::function<double()> forward,
                           std::function<double()> rotation)
    : m_drive{subsystem},
      m_forward{std::move(forward)},
      m_rotation{std::move(rotation)} {
  AddRequirements(subsystem);
}

void DefaultDrive::Execute() {
  m_drive->ArcadeDrive(m_forward(), m_rotation());
}
```

Python

```python
import typing
import commands2
from subsystems.drivesubsystem import DriveSubsystem


class DefaultDrive(commands2.Command):
    def __init__(
        self,
        drive: DriveSubsystem,
        forward: typing.Callable[[], float],
        rotation: typing.Callable[[], float],
    ) -> None:
        super().__init__()

        self.drive = drive
        self.forward = forward
        self.rotation = rotation

        self.addRequirements(self.drive)

    def execute(self) -> None:
        self.drive.arcadeDrive(self.forward(), self.rotation())
```

And then usage:

JAVA

```java
    // Configure default commands
    // Set the default drive command to split-stick arcade drive
    m_robotDrive.setDefaultCommand(
        // A split-stick arcade command, with forward/backward controlled by the left
        // hand, and turning controlled by the right.
        new DefaultDrive(
            m_robotDrive,
            () -> -m_driverController.getLeftY(),
            () -> -m_driverController.getRightX()));
```

C++

```cpp
  // Set up default drive command
  m_drive.SetDefaultCommand(DefaultDrive(
      &m_drive, [this] { return -m_driverController.GetLeftY(); },
      [this] { return -m_driverController.GetRightX(); }));
```

PYTHON

```
        # set up default drive command
        self.drive.setDefaultCommand(
            DefaultDrive(
                self.drive,
                lambda: -self.driverController.getY(),
                lambda: self.driverController.getX(),
            )
        )
```

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. Once more, this command is rather simple and calls the subsystem method only from one place, and as such, could be more concisely written using factories:

JAVA

```
    // Configure default commands
    // Set the default drive command to split-stick arcade drive
    m_robotDrive.setDefaultCommand(
        // A split-stick arcade command, with forward/backward controlled by the left
        // hand, and turning controlled by the right.
        Commands.run(
            () ->
                m_robotDrive.arcadeDrive(
                    -m_driverController.getLeftY(), -m_driverController.getRightX()),
            m_robotDrive));
```

C++

```
  // Set up default drive command
  m_drive.SetDefaultCommand(frc2::cmd::Run(
      [this] {
        m_drive.ArcadeDrive(-m_driverController.GetLeftY(),
                            -m_driverController.GetRightX());
      },
      {&m_drive}));
```

PYTHON

```
        # Configure default commands
        # Set the default drive command to split-stick arcade drive
        self.driveSubsystem.setDefaultCommand(
            # A split-stick arcade command, with forward/backward controlled by the␣
↪left
            # hand, and turning controlled by the right.
            commands2.cmd.run(
                lambda: self.driveSubsystem.arcadeDrive(
                    -self.driverController.getLeftY(),
                    -self.driverController.getRightX(),
```

(continues on next page)

(continued from previous page)

```
                ),
                self.driveSubsystem,
            )
        )
```

27.3 Command Compositions

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 composition functionality included in the command-based library.

As the name suggests, a command composition is a *composition* of one or more commands. 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 compositions are themselves commands - they extend the `Command` class. This allows command compositions to be further composed as a *recursive composition* - that is, a command composition may contain other command compositions as components. This allows very powerful and concise inline expressions:

JAVA

```
// Will run fooCommand, and then a race between barCommand and bazCommand
button.onTrue(fooCommand.andThen(barCommand.raceWith(bazCommand)));
```

C++

```
// Will run fooCommand, and then a race between barCommand and bazCommand
button.OnTrue(std::move(fooCommand).AndThen(std::move(barCommand).
↪RaceWith(std::move(bazCommand))));
```

PYTHON

```
# Will run fooCommand, and then a race between barCommand and bazCommand
button.onTrue(fooCommand.andThen(barCommand.raceWith(bazCommand)))
```

As a rule, command compositions require all subsystems their components require, may run when disabled if all their component set `runsWhenDisabled` as `true`, and are `kCancelIncoming` if all their components are `kCancelIncoming` as well.

Command instances that have been passed to a command composition cannot be independently scheduled or passed to a second command composition. Attempting to do so will throw an exception and crash the user program. This is because composition members are run through their encapsulating command composition, and errors could occur if those same

command instances were independently scheduled at the same time as the composition - 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. The C++ command-based library uses `CommandPtr`, a class with move-only semantics, so this type of mistake is easier to avoid.

27.3.1 Composition Types

The command-based library includes various composition types. All of them can be constructed using factories that accept the member commands, and some can also be constructed using decorators: methods that can be called on a command object, which is transformed into a new object that is returned.

Important

After calling a decorator or being passed to a composition, the command object cannot be reused! Use only the command object returned from the decorator.

Repeating

The `repeatedly()` decorator (Java, C++, Python), backed by the `RepeatCommand` class (Java, C++, Python) restarts the command each time it ends, so that it runs until interrupted.

JAVA

```
// Will run forever unless externally interrupted, restarting every time command.
↪isFinished() returns true
Command repeats = command.repeatedly();
```

C++

```
// Will run forever unless externally interrupted, restarting every time command.
↪IsFinished() returns true
frc2::CommandPtr repeats = std::move(command).Repeatedly();
```

PYTHON

```
# Will run forever unless externally interrupted, restarting every time command.
↪IsFinished() returns true
repeats = command.repeatedly()
```

Sequence

The `Sequence` factory (Java, C++, Python), backed by the `SequentialCommandGroup` class (Java, C++, Python), 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!).

The `andThen()` (Java, C++, Python) and `beforeStarting()` (Java, C++, Python) decorators can be used to construct a sequence composition with infix syntax.

JAVA

```
fooCommand.andThen(barCommand)
```

C++

```
std::move(fooCommand).AndThen(std::move(barCommand))
```

PYTHON

```
fooCommand.andThen(barCommand)
```

Repeating Sequence

As it's a fairly common combination, the `RepeatingSequence` factory (Java, C++, Python) creates a *Repeating Sequence* that runs until interrupted, restarting from the first command each time the last command finishes.

Parallel

There are three types of parallel compositions, differing based on when the composition finishes:

- The `Parallel` factory (Java, C++, Python), backed by the `ParallelCommandGroup` class (Java, C++, Python), constructs a parallel composition that finishes when all members finish. The `alongWith` decorator (Java, C++, Python) does the same in infix notation.
- The `Race` factory (Java, C++, Python), backed by the `ParallelRaceGroup` class (Java, C++, Python), constructs a parallel composition that finishes as soon as any member finishes; all other members are interrupted at that point. The `raceWith` decorator (Java, C++, Python) does the same in infix notation.
- The `Deadline` factory (Java, C++, Python), `ParallelDeadlineGroup` (Java, C++, Python) finishes when a specific command (the "deadline") ends; all other members still running at that point are interrupted. The `deadlineWith` decorator (Java, C++, Python) does the same in infix notation; the command the decorator was called on is the deadline.

JAVA

```
// Will be a parallel command composition that ends after three seconds with all␣
↪three commands running their full duration.
button.onTrue(Commands.parallel(twoSecCommand, oneSecCommand, threeSecCommand));
// Will be a parallel race composition that ends after one second with the two and␣
↪three second commands getting interrupted.
button.onTrue(Commands.race(twoSecCommand, oneSecCommand, threeSecCommand));
// Will be a parallel deadline composition that ends after two seconds (the deadline)␣
↪with the three second command getting interrupted (one second command already␣
↪finished).
button.onTrue(Commands.deadline(twoSecCommand, oneSecCommand, threeSecCommand));
```

C++

```
// Will be a parallel command composition that ends after three seconds with all␣
↪three commands running their full duration.
button.OnTrue(frc2::cmd::Parallel(std::move(twoSecCommand), std::move(oneSecCommand),␣
↪std::move(threeSecCommand)));
// Will be a parallel race composition that ends after one second with the two and␣
↪three second commands getting interrupted.
button.OnTrue(frc2::cmd::Race(std::move(twoSecCommand), std::move(oneSecCommand),␣
↪std::move(threeSecCommand)));
// Will be a parallel deadline composition that ends after two seconds (the deadline)␣
↪with the three second command getting interrupted (one second command already␣
↪finished).
button.OnTrue(frc2::cmd::Deadline(std::move(twoSecCommand), std::move(oneSecCommand),␣
↪std::move(threeSecCommand)));
```

PYTHON

```
# Will be a parallel command composition that ends after three seconds with all three␣
↪commands running their full duration.
button.onTrue(commands2.cmd.parallel(twoSecCommand, oneSecCommand, threeSecCommand))
# Will be a parallel race composition that ends after one second with the two and␣
↪three second commands getting interrupted.
button.onTrue(commands2.cmd.race(twoSecCommand, oneSecCommand, threeSecCommand))
# Will be a parallel deadline composition that ends after two seconds (the deadline)␣
↪with the three second command getting interrupted (one second command already␣
↪finished).
button.onTrue(commands2.cmd.deadline(twoSecCommand, oneSecCommand, threeSecCommand))
```

Adding Command End Conditions

The `until()` (Java, C++, Python) decorator composes the command with an additional end condition. Note that the command the decorator was called on will see this end condition as an interruption.

JAVA

```
// Will be interrupted if m_limitSwitch.get() returns true
button.onTrue(command.until(m_limitSwitch::get));
```

C++

```
// Will be interrupted if m_limitSwitch.get() returns true
button.OnTrue(command.Until([&m_limitSwitch] { return m_limitSwitch.Get(); }));
```

PYTHON

```
# Will be interrupted if limitSwitch.get() returns true
button.onTrue(commands2.cmd.until(limitSwitch.get))
```

The `withTimeout()` decorator (Java, C++, Python) is a specialization of `until` that uses a timeout as the additional end condition.

JAVA

```
// Will time out 5 seconds after being scheduled, and be interrupted
button.onTrue(command.withTimeout(5));
```

C++

```
// Will time out 5 seconds after being scheduled, and be interrupted
button.OnTrue(command.WithTimeout(5.0_s));
```

PYTHON

```
# Will time out 5 seconds after being scheduled, and be interrupted
button.onTrue(commands2.cmd.withTimeout(5.0))
```

Adding End Behavior

The `finallyDo()` (Java, C++, Python) decorator composes the command with an a lambda that will be called after the command's `end()` method, with the same boolean parameter indicating whether the command finished or was interrupted.

The `handleInterrupt()` (Java, C++, Python) decorator composes the command with an a lambda that will be called only when the command is interrupted.

Selecting Compositions

Sometimes it's desired to run a command out of a few options based on sensor feedback or other data known only at runtime. This can be useful for determining an auto routine, or running a different command based on whether a game piece is present or not, and so on.

The `Select` factory (Java, C++, Python), backed by the `SelectCommand` class (Java, C++, Python), executes one command from a map, based on a selector function called when scheduled.

Java

```
public class RobotContainer {
  // The enum used as keys for selecting the command to run.
  private enum CommandSelector {
    ONE,
    TWO,
    THREE
  }

  // An example selector method for the selectcommand.  Returns the selector that will select
  // which command to run.  Can base this choice on logical conditions evaluated at runtime.
  private CommandSelector select() {
    return CommandSelector.ONE;
  }

  // An example selectcommand.  Will select from the three commands based on the value returned
  // by the selector method at runtime.  Note that selectcommand works on Object(), so the
  // selector does not have to be an enum; it could be any desired type (string, integer,
  // boolean, double...)
  private final Command m_exampleSelectCommand =
      new SelectCommand<>(
          // Maps selector values to commands
          Map.ofEntries(
              Map.entry(CommandSelector.ONE, new PrintCommand("Command one was selected!")),
              Map.entry(CommandSelector.TWO, new PrintCommand("Command two was selected!")),
              Map.entry(CommandSelector.THREE, new PrintCommand("Command three was selected!"))),
          this::select);
```

C++ (Header)

```
  // The enum used as keys for selecting the command to run.
  enum CommandSelector { ONE, TWO, THREE };

  // An example of how command selector may be used with SendableChooser
  frc::SendableChooser<CommandSelector> m_chooser;

  // The robot's subsystems and commands are defined here...

  // An example selectcommand.  Will select from the three commands based on the
  // value returned by the selector method at runtime.  Note that selectcommand
  // takes a generic type, so the selector does not have to be an enum; it could
  // be any desired type (string, integer, boolean, double...)
  frc2::CommandPtr m_exampleSelectCommand = frc2::cmd::Select<CommandSelector>(
      [this] { return m_chooser.GetSelected(); },
      // Maps selector values to commands
      std::pair{ONE, frc2::cmd::Print("Command one was selected!")},
      std::pair{TWO, frc2::cmd::Print("Command two was selected!")},
      std::pair{THREE, frc2::cmd::Print("Command three was selected!")});
```

The `Either` factory (Java, C++, Python), backed by the `ConditionalCommand` class (Java, C++, Python), is a specialization accepting two commands and a boolean selector function.

JAVA

```
// Runs either commandOnTrue or commandOnFalse depending on the value of m_
↪limitSwitch.get()
new ConditionalCommand(commandOnTrue, commandOnFalse, m_limitSwitch::get)
```

C++

```
// Runs either commandOnTrue or commandOnFalse depending on the value of m_
↪limitSwitch.get()
frc2::ConditionalCommand(commandOnTrue, commandOnFalse, [&m_limitSwitch] { return m_
↪limitSwitch.Get(); })
```

PYTHON

```
# Runs either commandOnTrue or commandOnFalse depending on the value of limitSwitch.
↪get()
ConditionalCommand(commandOnTrue, commandOnFalse, limitSwitch.get)
```

The `unless()` decorator (Java, C++, Python) composes a command with a condition that will prevent it from running.

JAVA

```
// Command will only run if the intake is deployed. If the intake gets deployed while␣
↪the command is running, the command will not stop running
button.onTrue(command.unless(() -> !intake.isDeployed()));
```

C++

```
// Command will only run if the intake is deployed. If the intake gets deployed while␣
↪the command is running, the command will not stop running
button.OnTrue(command.Unless([&intake] { return !intake.IsDeployed(); }));
```

PYTHON

```
# Command will only run if the intake is deployed. If the intake gets deployed while␣
↪the command is running, the command will not stop running
button.onTrue(command.unless(lambda: not intake.isDeployed()))
```

`ProxyCommand` described below also has a constructor overload (Java, C++, `Python`) that calls a command-returning lambda at schedule-time and runs the returned command by proxy.

Scheduling Other Commands

By default, composition members are run through the command composition, and are never themselves seen by the scheduler. Accordingly, their requirements are added to the composition's requirements. While this is usually fine, sometimes it is undesirable for the entire command composition to gain the requirements of a single command. A good solution is to "fork off" from the command composition and schedule that command separately. However, this requires synchronization between the composition and the individually-scheduled command.

`ProxyCommand` (Java, C++, `Python`), also creatable using the `.asProxy()` decorator (Java, C++, `Python`), schedules a command "by proxy": the command is scheduled when the proxy is scheduled, and the proxy finishes when the command finishes. In the case of "forking off" from a command composition, this allows the composition to track the command's progress without it being in the composition.

Command compositions inherit the union of their compoments' requirements and requirements are immutable. Therefore, a `SequentialCommandGroup` (Java, C++, `Python`) that intakes a game piece, indexes it, aims a shooter, and shoots it would reserve all three subsystems (the intake, indexer, and shooter), precluding any of those subsystems from performing other operations in their "downtime". If this is not desired, the subsystems that should only be reserved for the composition while they are actively being used by it should have their commands proxied.

> **⚠ Warning**
>
> Do not use `ProxyCommand` unless you are sure of what you are doing and there is no other way to accomplish your need! Proxying is only intended for use as an escape hatch from command composition requirement unions.

Note

Because proxied commands still require their subsystem, despite not leaking that requirement to the composition, all of the commands that require a given subsystem must be proxied if one of them is. Otherwise, when the proxied command is scheduled its requirement will conflict with that of the composition, canceling the composition.

JAVA

```
// composition requirements are indexer and shooter, intake still reserved during its␣
↪command but not afterwards
Commands.sequence(
   intake.intakeGamePiece().asProxy(), // we want to let the intake intake another␣
↪game piece while we are processing this one
   indexer.processGamePiece(),
   shooter.aimAndShoot()
);
```

C++

```
// composition requirements are indexer and shooter, intake still reserved during its␣
↪command but not afterwards
frc2::cmd::Sequence(
   intake.IntakeGamePiece().AsProxy(), // we want to let the intake intake another␣
↪game piece while we are processing this one
   indexer.ProcessGamePiece(),
   shooter.AimAndShoot()
);
```

PYTHON

```
# composition requirements are indexer and shooter, intake still reserved during its␣
↪command but not afterwards
commands2.cmd.sequence(
   intake.intakeGamePiece().asProxy(), # we want to let the intake intake another␣
↪game piece while we are processing this one
   indexer.processGamePiece(),
   shooter.aimAndShoot()
)
```

For cases that don't need to track the proxied command, `ScheduleCommand` (Java, C++, Python) schedules a specified command and ends instantly.

JAVA

```
// ScheduleCommand ends immediately, so the sequence continues
new ScheduleCommand(Commands.waitSeconds(5.0))
   .andThen(Commands.print("This will be printed immediately!"))
```

C++

```
// ScheduleCommand ends immediately, so the sequence continues
frc2::ScheduleCommand(frc2::cmd::Wait(5.0_s))
   .AndThen(frc2::cmd::Print("This will be printed immediately!"))
```

PYTHON

```
# ScheduleCommand ends immediately, so the sequence continues
ScheduleCommand(commands2.cmd.waitSeconds(5.0))
   .andThen(commands2.cmd.print("This will be printed immediately!"))
```

27.3.2 Subclassing Compositions

Command compositions can also be written as a constructor-only subclass of the most exterior composition type, passing the composition members to the superclass constructor. Consider the following from the Hatch Bot example project (Java, C++):

Java

```
5  package edu.wpi.first.wpilibj.examples.hatchbottraditional.commands;
6
7  import edu.wpi.first.wpilibj.examples.hatchbottraditional.Constants.AutoConstants;
8  import edu.wpi.first.wpilibj.examples.hatchbottraditional.subsystems.DriveSubsystem;
9  import edu.wpi.first.wpilibj.examples.hatchbottraditional.subsystems.HatchSubsystem;
10 import edu.wpi.first.wpilibj2.command.SequentialCommandGroup;
11
12 /** A complex auto command that drives forward, releases a hatch, and then drives␣
   ↪backward. */
13 public class ComplexAuto extends SequentialCommandGroup {
14   /**
15    * Creates a new ComplexAuto.
16    *
17    * @param drive The drive subsystem this command will run on
18    * @param hatch The hatch subsystem this command will run on
19    */
20   public ComplexAuto(DriveSubsystem drive, HatchSubsystem hatch) {
21     addCommands(
22         // Drive forward the specified distance
23         new DriveDistance(
24             AutoConstants.kAutoDriveDistanceInches, AutoConstants.kAutoDriveSpeed,␣
   ↪drive),
25
```

(continues on next page)

(continued from previous page)

```
        // Release the hatch
        new ReleaseHatch(hatch),

        // Drive backward the specified distance
        new DriveDistance(
            AutoConstants.kAutoBackupDistanceInches, -AutoConstants.kAutoDriveSpeed,␣
↪drive));
  }
}
```

C++ (Header)

```
#pragma once

#include <frc2/command/CommandHelper.h>
#include <frc2/command/SequentialCommandGroup.h>

#include "Constants.h"
#include "commands/DriveDistance.h"
#include "commands/ReleaseHatch.h"

/**
 * A complex auto command that drives forward, releases a hatch, and then drives
 * backward.
 */
class ComplexAuto
    : public frc2::CommandHelper<frc2::SequentialCommandGroup, ComplexAuto> {
 public:
  /**
   * Creates a new ComplexAuto.
   *
   * @param drive The drive subsystem this command will run on
   * @param hatch The hatch subsystem this command will run on
   */
  ComplexAuto(DriveSubsystem* drive, HatchSubsystem* hatch);
};
```

C++ (Source)

```
#include "commands/ComplexAuto.h"

using namespace AutoConstants;

ComplexAuto::ComplexAuto(DriveSubsystem* drive, HatchSubsystem* hatch) {
  AddCommands(
      // Drive forward the specified distance
      DriveDistance(kAutoDriveDistanceInches, kAutoDriveSpeed, drive),
      // Release the hatch
      ReleaseHatch(hatch),
      // Drive backward the specified distance
      DriveDistance(kAutoBackupDistanceInches, -kAutoDriveSpeed, drive));
}
```

Python

```
import commands2

import constants

from .drivedistance import DriveDistance
from .releasehatch import ReleaseHatch

from subsystems.drivesubsystem import DriveSubsystem
from subsystems.hatchsubsystem import HatchSubsystem


class ComplexAuto(commands2.SequentialCommandGroup):
    """
    A complex auto command that drives forward, releases a hatch, and then drives backward.
    """

    def __init__(self, drive: DriveSubsystem, hatch: HatchSubsystem):
        super().__init__(
            # Drive forward the specified distance
            DriveDistance(
                constants.kAutoDriveDistanceInches, constants.kAutoDriveSpeed, drive
            ),
            # Release the hatch
            ReleaseHatch(hatch),
            # Drive backward the specified distance
            DriveDistance(
                constants.kAutoBackupDistanceInches, -constants.kAutoDriveSpeed, drive
            ),
        )
```

The advantages and disadvantages of this subclassing approach in comparison to others are discussed in *Subclassing Command Groups*.

27.4 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 form an *encapsulation* for this hardware, "hiding" it from the rest of the robot code 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 or be ignored, based on the running command's *Interruption Behavior*.

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 "background" actions such as controlling the robot drive, keeping an arm held at a setpoint, or stopping motors when the subsystem isn't used. 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. Subsystems are represented in the command-based library by the `Subsystem` interface (Java, C++, Python).

27.4.1 Creating a Subsystem

The recommended method to create a subsystem for most users is to subclass the abstract `SubsystemBase` class in (Java, C++), as seen in the command-based template (Java, C++). In Python, because Python does not have interfaces, the `Subsystem` class is a concrete class that can be subclassed directly (Python). The following example demonstrates how to create a simple subsystem in each of the supported languages:

Java

```
7  import edu.wpi.first.wpilibj2.command.Command;
8  import edu.wpi.first.wpilibj2.command.SubsystemBase;
9
10 public class ExampleSubsystem extends SubsystemBase {
11   /** Creates a new ExampleSubsystem. */
12   public ExampleSubsystem() {}
13
14   /**
15    * Example command factory method.
16    *
17    * @return a command
18    */
19   public Command exampleMethodCommand() {
20     // Inline construction of command goes here.
21     // Subsystem::RunOnce implicitly requires `this` subsystem.
22     return runOnce(
23         () -> {
24           /* one-time action goes here */
25         });
26   }
27
28   /**
29    * An example method querying a boolean state of the subsystem (for example, a␣
   ↪digital sensor).
30    *
31    * @return value of some boolean subsystem state, such as a digital sensor.
32    */
33   public boolean exampleCondition() {
34     // Query some boolean state, such as a digital sensor.
35     return false;
36   }
37
38   @Override
39   public void periodic() {
```

(continues on next page)

(continued from previous page)

```
    // This method will be called once per scheduler run
  }

  @Override
  public void simulationPeriodic() {
    // This method will be called once per scheduler run during simulation
  }
}
```

C++

```
#pragma once

#include <frc2/command/CommandPtr.h>
#include <frc2/command/SubsystemBase.h>

class ExampleSubsystem : public frc2::SubsystemBase {
 public:
  ExampleSubsystem();

  /**
   * Example command factory method.
   */
  frc2::CommandPtr ExampleMethodCommand();

  /**
   * An example method querying a boolean state of the subsystem (for example, a
   * digital sensor).
   *
   * @return value of some boolean subsystem state, such as a digital sensor.
   */
  bool ExampleCondition();

  /**
   * Will be called periodically whenever the CommandScheduler runs.
   */
  void Periodic() override;

  /**
   * Will be called periodically whenever the CommandScheduler runs during
   * simulation.
   */
  void SimulationPeriodic() override;

 private:
  // Components (e.g. motor controllers and sensors) should generally be
  // declared private and exposed only through public methods.
};
```

Python

```
from commands2 import Command
from commands2 import Subsystem
class ExampleSubsystem(Subsystem):
    def __init__(self):
        """Creates a new ExampleSubsystem."""
        super().__init__()
    def exampleMethodCommand()->Command:
        """
        Example command factory method.
         :return a command
        """
        return self.runOnce(
            lambda: # one-time action goes here #
        )
    def exampleCondition(self)->bool:
        """
        An example method querying a boolean state of the subsystem (for example, a␣
↪digital sensor).
        :return value of some boolean subsystem state, such as a digital sensor.
        """
        #Query some boolean state, such as a digital sensor.
        return False
    def periodic(self):
        # This method will be called once per scheduler run
        pass
    def simulationPeriodic(self):
        # This method will be called once per scheduler run during simulation
        pass
```

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.

27.4.2 Simple Subsystem Example

What might a functional subsystem look like in practice? Below is a simple pneumatically-actuated hatch mechanism from the HatchBotTraditional example project (Java, C++, Python):

Java

```java
package edu.wpi.first.wpilibj.examples.hatchbottraditional.subsystems;

import static edu.wpi.first.wpilibj.DoubleSolenoid.Value.kForward;
import static edu.wpi.first.wpilibj.DoubleSolenoid.Value.kReverse;

import edu.wpi.first.util.sendable.SendableBuilder;
import edu.wpi.first.wpilibj.DoubleSolenoid;
import edu.wpi.first.wpilibj.PneumaticsModuleType;
import edu.wpi.first.wpilibj.examples.hatchbottraditional.Constants.HatchConstants;
import edu.wpi.first.wpilibj2.command.SubsystemBase;

/** A hatch mechanism actuated by a single {@link DoubleSolenoid}. */
public class HatchSubsystem extends SubsystemBase {
  private final DoubleSolenoid m_hatchSolenoid =
      new DoubleSolenoid(
          PneumaticsModuleType.CTREPCM,
          HatchConstants.kHatchSolenoidPorts[0],
          HatchConstants.kHatchSolenoidPorts[1]);

  /** Grabs the hatch. */
  public void grabHatch() {
    m_hatchSolenoid.set(kForward);
  }

  /** Releases the hatch. */
  public void releaseHatch() {
    m_hatchSolenoid.set(kReverse);
  }

  @Override
  public void initSendable(SendableBuilder builder) {
    super.initSendable(builder);
    // Publish the solenoid state to telemetry.
    builder.addBooleanProperty("extended", () -> m_hatchSolenoid.get() == kForward,␣
↪null);
  }
}
```

C++ (Header)

```cpp
#pragma once

#include <frc/DoubleSolenoid.h>
#include <frc/PneumaticsControlModule.h>
#include <frc2/command/SubsystemBase.h>

#include "Constants.h"

class HatchSubsystem : public frc2::SubsystemBase {
 public:
  HatchSubsystem();

  // Subsystem methods go here.
```

(continues on next page)

(continued from previous page)

```

  /**
   * Grabs the hatch.
   */
  void GrabHatch();

  /**
   * Releases the hatch.
   */
  void ReleaseHatch();

  void InitSendable(wpi::SendableBuilder& builder) override;

 private:
  // Components (e.g. motor controllers and sensors) should generally be
  // declared private and exposed only through public methods.
  frc::DoubleSolenoid m_hatchSolenoid;
};
```

C++ (Source)

```
#include "subsystems/HatchSubsystem.h"

#include <wpi/sendable/SendableBuilder.h>

using namespace HatchConstants;

HatchSubsystem::HatchSubsystem()
    : m_hatchSolenoid{frc::PneumaticsModuleType::CTREPCM,
                      kHatchSolenoidPorts[0], kHatchSolenoidPorts[1]} {}

void HatchSubsystem::GrabHatch() {
  m_hatchSolenoid.Set(frc::DoubleSolenoid::kForward);
}

void HatchSubsystem::ReleaseHatch() {
  m_hatchSolenoid.Set(frc::DoubleSolenoid::kReverse);
}

void HatchSubsystem::InitSendable(wpi::SendableBuilder& builder) {
  SubsystemBase::InitSendable(builder);

  // Publish the solenoid state to telemetry.
  builder.AddBooleanProperty(
      "extended",
      [this] { return m_hatchSolenoid.Get() == frc::DoubleSolenoid::kForward; },
      nullptr);
}
```

Python

```python
import wpilib
import commands2

import constants


class HatchSubsystem(commands2.Subsystem):
    def __init__(self) -> None:
        super().__init__()

        self.hatchSolenoid = wpilib.DoubleSolenoid(
            constants.kHatchSolenoidModule,
            constants.kHatchSolenoidModuleType,
            *constants.kHatchSolenoidPorts
        )

    def grabHatch(self) -> None:
        """Grabs the hatch"""
        self.hatchSolenoid.set(wpilib.DoubleSolenoid.Value.kForward)

    def releaseHatch(self) -> None:
        """Releases the hatch"""
        self.hatchSolenoid.set(wpilib.DoubleSolenoid.Value.kReverse)
```

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.

Alternatively, instead of writing `void` public methods that are called from commands, we can define the public methods as factories that return a command. Consider the following from the HatchBotInlined example project (Java, C++, Python):

Java

```java
package edu.wpi.first.wpilibj.examples.hatchbotinlined.subsystems;

import static edu.wpi.first.wpilibj.DoubleSolenoid.Value.kForward;
import static edu.wpi.first.wpilibj.DoubleSolenoid.Value.kReverse;

import edu.wpi.first.util.sendable.SendableBuilder;
import edu.wpi.first.wpilibj.DoubleSolenoid;
import edu.wpi.first.wpilibj.PneumaticsModuleType;
import edu.wpi.first.wpilibj.examples.hatchbotinlined.Constants.HatchConstants;
import edu.wpi.first.wpilibj2.command.Command;
import edu.wpi.first.wpilibj2.command.SubsystemBase;

/** A hatch mechanism actuated by a single {@link edu.wpi.first.wpilibj.
↪DoubleSolenoid}. */
public class HatchSubsystem extends SubsystemBase {
```

(continues on next page)

(continued from previous page)

```
  private final DoubleSolenoid m_hatchSolenoid =
      new DoubleSolenoid(
          PneumaticsModuleType.CTREPCM,
          HatchConstants.kHatchSolenoidPorts[0],
          HatchConstants.kHatchSolenoidPorts[1]);

  /** Grabs the hatch. */
  public Command grabHatchCommand() {
    // implicitly require `this`
    return this.runOnce(() -> m_hatchSolenoid.set(kForward));
  }

  /** Releases the hatch. */
  public Command releaseHatchCommand() {
    // implicitly require `this`
    return this.runOnce(() -> m_hatchSolenoid.set(kReverse));
  }

  @Override
  public void initSendable(SendableBuilder builder) {
    super.initSendable(builder);
    // Publish the solenoid state to telemetry.
    builder.addBooleanProperty("extended", () -> m_hatchSolenoid.get() == kForward,␣
↪null);
  }
}
```

C++ (Header)

```
#pragma once

#include <frc/DoubleSolenoid.h>
#include <frc/PneumaticsControlModule.h>
#include <frc2/command/CommandPtr.h>
#include <frc2/command/SubsystemBase.h>

#include "Constants.h"

class HatchSubsystem : public frc2::SubsystemBase {
 public:
  HatchSubsystem();

  // Subsystem methods go here.

  /**
   * Grabs the hatch.
   */
  frc2::CommandPtr GrabHatchCommand();

  /**
   * Releases the hatch.
   */
  frc2::CommandPtr ReleaseHatchCommand();

```

(continues on next page)

(continued from previous page)

```
  void InitSendable(wpi::SendableBuilder& builder) override;

 private:
  // Components (e.g. motor controllers and sensors) should generally be
  // declared private and exposed only through public methods.
  frc::DoubleSolenoid m_hatchSolenoid;
};
```

C++ (Source)

```
#include "subsystems/HatchSubsystem.h"

#include <wpi/sendable/SendableBuilder.h>

using namespace HatchConstants;

HatchSubsystem::HatchSubsystem()
    : m_hatchSolenoid{frc::PneumaticsModuleType::CTREPCM,
                      kHatchSolenoidPorts[0], kHatchSolenoidPorts[1]} {}

frc2::CommandPtr HatchSubsystem::GrabHatchCommand() {
  // implicitly require `this`
  return this->RunOnce(
      [this] { m_hatchSolenoid.Set(frc::DoubleSolenoid::kForward); });
}

frc2::CommandPtr HatchSubsystem::ReleaseHatchCommand() {
  // implicitly require `this`
  return this->RunOnce(
      [this] { m_hatchSolenoid.Set(frc::DoubleSolenoid::kReverse); });
}

void HatchSubsystem::InitSendable(wpi::SendableBuilder& builder) {
  SubsystemBase::InitSendable(builder);

  // Publish the solenoid state to telemetry.
  builder.AddBooleanProperty(
      "extended",
      [this] { return m_hatchSolenoid.Get() == frc::DoubleSolenoid::kForward; },
      nullptr);
}
```

Python

```
import wpilib
import commands2
import commands2.cmd

import constants


class HatchSubsystem(commands2.Subsystem):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self) -> None:
        super().__init__()

        self.hatchSolenoid = wpilib.DoubleSolenoid(
            constants.kHatchSolenoidModule,
            constants.kHatchSolenoidModuleType,
            *constants.kHatchSolenoidPorts
        )

    def grabHatch(self) -> commands2.Command:
        """Grabs the hatch"""
        return commands2.cmd.runOnce(
            lambda: self.hatchSolenoid.set(wpilib.DoubleSolenoid.Value.kForward), self
        )

    def releaseHatch(self) -> commands2.Command:
        """Releases the hatch"""
        return commands2.cmd.runOnce(
            lambda: self.hatchSolenoid.set(wpilib.DoubleSolenoid.Value.kReverse), self
        )
```

Note the qualification of the `RunOnce` factory used here: this isn't the static factory in `Commands`! Subsystems have similar instance factories that return commands requiring `this` (Java/C++) or `self` (Python) subsystem. Here, the `Subsystem.runOnce(Runnable)` factory (Java, C++, Python) is used.

For a comparison between these options, see *Instance Command Factory Methods*.

27.4.3 Periodic

Subsystems have a `periodic` method that is called once every scheduler iteration (usually, once every 20 ms). This method is typically used for telemetry and other periodic actions that do not interfere with whatever command is requiring the subsystem.

Java

```
  @Override
  public void periodic() {
    // Update the odometry in the periodic block
    m_odometry.update(
        Rotation2d.fromDegrees(getHeading()),
        m_leftEncoder.getDistance(),
        m_rightEncoder.getDistance());
    m_fieldSim.setRobotPose(getPose());
  }
```

C++ (Header)

```
30   void Periodic() override;
```

C++ (Source)

```
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   m_fieldSim.SetRobotPose(m_odometry.GetPose());
36 }
```

Python

```
def periodic(self):
    #Update the odometry in the periodic block
    self.odometry.update(
        Rotation2d.fromDegrees(getHeading()),
        self.leftEncoder.getDistance(),
        self.rightEncoder.getDistance())
    self.fieldSim.setRobotPose(getPose())
```

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.

27.4.4 Default Commands

Note

In the C++ command-based library, the CommandScheduler *owns* the default command object.

"Default commands" are commands that run automatically whenever a subsystem is not being used by another command. This can be useful for "background" actions such as controlling the robot drive, or keeping an arm held at a setpoint.

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

```
CommandScheduler.getInstance().setDefaultCommand(exampleSubsystem, exampleCommand);
```

C++

```
CommandScheduler.GetInstance().SetDefaultCommand(exampleSubsystem,␣
↪std::move(exampleCommand));
```

PYTHON

```
CommandScheduler.getInstance().setDefaultCommand(exampleSubsystem, exampleCommand)
```

JAVA

```
exampleSubsystem.setDefaultCommand(exampleCommand);
```

C++

```
exampleSubsystem.SetDefaultCommand(std::move(exampleCommand));
```

PYTHON

```
exampleSubsystem.setDefaultCommand(exampleCommand)
```

Note

A command that is assigned as the default command for a subsystem must require that subsystem.

27.5 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 programming* 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++).

27.5.1 Getting a Trigger Instance

To bind commands to conditions, we need a `Trigger` object. There are three ways to get a Trigger object:

HID Factories

The command-based HID classes contain factory methods returning a `Trigger` for a given button. `CommandGenericHID` has an index-based `button(int)` factory (Java, C++), and its subclasses `CommandXboxController` (Java, C++), `CommandPS4Controller` (Java, C++), and `CommandJoystick` (Java, C++) have named factory methods for each button.

JAVA

```
CommandXboxController exampleCommandController = new CommandXboxController(1); //␣
↪Creates a CommandXboxController on port 1.
Trigger xButton = exampleCommandController.x(); // Creates a new Trigger object for␣
↪the `X` button on exampleCommandController
```

C++

```
frc2::CommandXboxController exampleCommandController{1} // Creates a␣
↪CommandXboxController on port 1
frc2::Trigger xButton = exampleCommandController.X() // Creates a new Trigger object␣
↪for the `X` button on exampleCommandController
```

JoystickButton

Alternatively, the *regular HID classes* can be used and passed to create an instance of `JoystickButton` Java, C++), a constructor-only subclass of `Trigger`:

JAVA

```
XboxController exampleController = new XboxController(2); // Creates an␣
↪XboxController on port 2.
Trigger yButton = new JoystickButton(exampleController, XboxController.Button.kY.
↪value); // Creates a new JoystickButton object for the `Y` button on␣
↪exampleController
```

C++

```
frc::XboxController exampleController{2} // Creates an XboxController on port 2
frc2::JoystickButton yButton(&exampleStick, frc::XboxController::Button::kY); //␣
↪Creates a new JoystickButton object for the `Y` button on exampleController
```

Arbitrary Triggers

While binding to HID buttons is by far the most common use case, users may want to bind commands to arbitrary triggering events. This can be done inline by passing a lambda to the constructor of `Trigger`:

JAVA

```
DigitalInput limitSwitch = new DigitalInput(3); // Limit switch on DIO 3
Trigger exampleTrigger = new Trigger(limitSwitch::get);
```

C++

```
frc::DigitalInput limitSwitch{3}; // Limit switch on DIO 3
frc2::Trigger exampleTrigger([&limitSwitch] { return limitSwitch.Get(); });
```

27.5.2 Trigger Bindings

Note

The C++ command-based library offers two overloads of each button binding method - one that takes an rvalue reference (`CommandPtr&&`), and one that takes a raw pointer (`Command*`). The rvalue overload moves ownership 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 rvalue 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.

`Trigger` objects *do not need to survive past the call to a binding method*, so the binding methods may be simply called on a temp. Remember that button binding is *declarative*: bindings only need to be declared once, ideally some time during robot initialization. The library handles everything else.

Note

The `Button` subclass is deprecated, and usage of its binding methods should be replaced according to the respective deprecation messages in the API docs.

onTrue

This binding schedules a command when a trigger changes from `false` to `true` (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 `false` and then `true` again (or the button is released and then re-pressed).

JAVA

```
65    // Retract the intake with the Y button
66    m_driverController.y().onTrue(m_intake.retractCommand());
```

C++

```
28  // Deploy the intake with the X button
29  m_driverController.X().OnTrue(m_intake.IntakeCommand());
```

The `onFalse` binding is identical, only that it schedules on `false` instead of on `true`.

whileTrue

This binding schedules a command when a trigger changes from `false` to `true` (or, accordingly, when a button is initially pressed) and cancels it when the trigger becomes `false` again (or the button is released). The command will *not* be re-scheduled if it finishes while the trigger is still `true`. For the command to restart if it finishes while the trigger is `true`, wrap the command in a `RepeatCommand`, or use a `RunCommand` instead of an `InstantCommand`.

JAVA

```
49    // Schedule `exampleMethodCommand` when the Xbox controller's B button is pressed,
50    // cancelling on release.
51    m_driverController.b().whileTrue(m_exampleSubsystem.exampleMethodCommand());
```

C++

```
27  // Schedule `ExampleMethodCommand` when the Xbox controller's B button is
28  // pressed, cancelling on release.
29  m_driverController.B().WhileTrue(m_subsystem.ExampleMethodCommand());
```

The `whileFalse` binding is identical, only that it schedules on `false` and cancels on `true`.

toggleOnTrue

This binding toggles a command, scheduling it when a trigger changes from `false` to `true` (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

```
78      // Toggle compressor with the Start button
79      m_driverController.start().toggleOnTrue(m_pneumatics.disableCompressorCommand());
```

C++

```
41    // Toggle compressor with the Start button
42    m_driverController.Start().ToggleOnTrue(
43        m_pneumatics.DisableCompressorCommand());
```

The `toggleOnFalse` binding is identical, only that it toggles on `false` instead of on `true`.

27.5.3 Chaining Calls

It is useful to note that the command binding methods all return the trigger that they were called on, and thus can be chained to bind multiple commands to different states of the same trigger. For example:

JAVA

```
exampleButton
    // Binds a FooCommand to be scheduled when the button is pressed
    .onTrue(new FooCommand())
    // Binds a BarCommand to be scheduled when that same button is released
    .onFalse(new BarCommand());
```

C++

```
exampleButton
    // Binds a FooCommand to be scheduled when the button is pressed
    .OnTrue(FooCommand().ToPtr())
    // Binds a BarCommand to be scheduled when that same button is released
    .OnFalse(BarCommand().ToPtr());
```

27.5.4 Composing Triggers

The `Trigger` class can be composed to create composite triggers through the `and()`, `or()`, and `negate()` methods (or, in C++, the &&, ||, and ! operators). For example:

JAVA

```
// Binds an ExampleCommand to be scheduled when both the 'X' and 'Y' buttons of the␣
↪driver gamepad are pressed
exampleCommandController.x()
    .and(exampleCommandController.y())
    .onTrue(new ExampleCommand());
```

C++

```
// Binds an ExampleCommand to be scheduled when both the 'X' and 'Y' buttons of the␣
↪driver gamepad are pressed
(exampleCommandController.X()
    && exampleCommandController.Y())
    .OnTrue(ExampleCommand().ToPtr());
```

27.5.5 Debouncing Triggers

To avoid rapid repeated activation, triggers (especially those originating from digital inputs) can be debounced with the *WPILib Debouncer class* using the *debounce* method:

JAVA

```
// debounces exampleButton with a 0.1s debounce time, rising edges only
exampleButton.debounce(0.1).onTrue(new ExampleCommand());
// debounces exampleButton with a 0.1s debounce time, both rising and falling edges
exampleButton.debounce(0.1, Debouncer.DebounceType.kBoth).onTrue(new␣
↪ExampleCommand());
```

C++

```
// debounces exampleButton with a 100ms debounce time, rising edges only
exampleButton.Debounce(100_ms).OnTrue(ExampleCommand().ToPtr());
// debounces exampleButton with a 100ms debounce time, both rising and falling edges
exampleButton.Debounce(100_ms, Debouncer::DebounceType::Both).OnTrue(ExampleCommand().
↪ToPtr());
```

27.6 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.

27.6.1 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

```
21   /**
22    * This function is run when the robot is first started up and should be used for␣
↪any
23    * initialization code.
24    */
25   public Robot() {
26     // Instantiate our RobotContainer.  This will perform all our button bindings,␣
↪and put our
27     // autonomous chooser on the dashboard.
28     m_robotContainer = new RobotContainer();
29   }
```

In Java, an instance of `RobotContainer` should be constructed during the `Robot` constructor - 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

```
  /**
   * This function is called every 20 ms, no matter the mode. Use this for items like diagnostics
   * that you want ran during disabled, autonomous, teleoperated and test.
   *
   * <p>This runs after the mode specific periodic functions, but before LiveWindow and
   * SmartDashboard integrated updating.
   */
  @Override
  public void robotPeriodic() {
    // Runs the Scheduler.  This is responsible for polling buttons, adding newly-scheduled
    // commands, running already-scheduled commands, removing finished or interrupted commands,
    // and running subsystem periodic() methods.  This must be called from the robot's periodic
    // block in order for anything in the Command-based framework to work.
    CommandScheduler.getInstance().run();
  }
```

C++ (Source)

```
/**
 * This function is called every 20 ms, no matter the mode. Use
 * this for items like diagnostics that you want to run during disabled,
 * autonomous, teleoperated and test.
 *
 * <p> This runs after the mode specific periodic functions, but before
 * LiveWindow and SmartDashboard integrated updating.
 */
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

```
  /** This autonomous runs the autonomous command selected by your {@link RobotContainer} class. */
  @Override
  public void autonomousInit() {
    m_autonomousCommand = m_robotContainer.getAutonomousCommand();

    // schedule the autonomous command (example)
    if (m_autonomousCommand != null) {
```

(continues on next page)

(continued from previous page)

```
      CommandScheduler.getInstance().schedule(m_autonomousCommand);
    }
  }
```

C++ (Source)

```
/**
 * This autonomous runs the autonomous command selected by your {@link
 * RobotContainer} class.
 */
void Robot::AutonomousInit() {
  m_autonomousCommand = m_container.GetAutonomousCommand();

  if (m_autonomousCommand) {
    frc2::CommandScheduler::GetInstance().Schedule(m_autonomousCommand.value());
  }
}
```

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

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

C++ (Source)

```
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) {
    m_autonomousCommand->Cancel();
  }
}
```

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.

27.6.2 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

```
23   private final ExampleSubsystem m_exampleSubsystem = new ExampleSubsystem();
```

C++ (Header)

```
32   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

```
61     return Autos.exampleAuto(m_exampleSubsystem);
```

C++ (Source)

```
34   return autos::ExampleAuto(&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

```
  /**
   * Use this method to define your trigger->command mappings. Triggers can be created via the
   * {@link Trigger#Trigger(java.util.function.BooleanSupplier)} constructor with an arbitrary
   * predicate, or via the named factories in {@link
   * edu.wpi.first.wpilibj2.command.button.CommandGenericHID}'s subclasses for {@link
   * CommandXboxController Xbox}/{@link edu.wpi.first.wpilibj2.command.button.CommandPS4Controller
   * PS4} controllers or {@link edu.wpi.first.wpilibj2.command.button.CommandJoystick Flight
   * joysticks}.
   */
  private void configureBindings() {
    // Schedule `ExampleCommand` when `exampleCondition` changes to `true`
    new Trigger(m_exampleSubsystem::exampleCondition)
        .onTrue(new ExampleCommand(m_exampleSubsystem));

    // Schedule `exampleMethodCommand` when the Xbox controller's B button is pressed,
    // cancelling on release.
    m_driverController.b().whileTrue(m_exampleSubsystem.exampleMethodCommand());
  }
```

C++ (Source)

```
void RobotContainer::ConfigureBindings() {
  // Configure your trigger bindings here

  // Schedule `ExampleCommand` when `exampleCondition` changes to `true`
  frc2::Trigger([this] {
    return m_subsystem.ExampleCondition();
  }).OnTrue(ExampleCommand(&m_subsystem).ToPtr());

  // Schedule `ExampleMethodCommand` when the Xbox controller's B button is
  // pressed, cancelling on release.
  m_driverController.B().WhileTrue(m_subsystem.ExampleMethodCommand());
}
```

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 `configureBindings()` method included by default) which are called from the constructor.

Java

```
  /**
   * Use this to pass the autonomous command to the main {@link Robot} class.
   *
   * @return the command to run in autonomous
   */
  public Command getAutonomousCommand() {
    // An example command will be run in autonomous
    return Autos.exampleAuto(m_exampleSubsystem);
  }
}
```

C++ (Source)

```
frc2::CommandPtr RobotContainer::GetAutonomousCommand() {
  // An example command will be run in autonomous
  return autos::ExampleAuto(&m_subsystem);
}
```

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).

27.6.3 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:

- Hatchbot (Java, C++)
- RapidReactCommandBot (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

```
import static edu.wpi.first.wpilibj.templates.commandbased.Constants.OIConstants.*;
```

C++

```
using namespace OIConstants;
```

27.6.4 Subsystems

User-defined subsystems should go in this package/directory.

27.6.5 Commands

User-defined commands should go in this package/directory.

27.7 Organizing Command-Based Robot Projects

As robot code becomes more complicated, navigating, understanding, and maintaining the code takes up more and more time and energy. Making changes to the code often becomes more difficult, sometimes for reasons that have very little to do with the actual complexity of the underlying logic. For a simplified example: putting the logic for many unrelated robot functions into a single 1000-line file makes it difficult to find a specific piece of code within that file, particularly under stress at a competition. But spreading out closely related logic across dozens of tiny files is often just as difficult to navigate.

This is not a problem unique to FRC, and in fact, good organization only becomes more and more critical as software projects become bigger and bigger. The “best” organization system is a perennial topic of debate, much like the “best” programming language, but in the end, the choice (in both cases) comes down to the specific task at hand and the programmer (or programmers) implementing said task. Even in the relatively small space of FRC robot programming, there is no right answer. The best choice for a given team will depend on the nature of the specific robot code, team structure, and pure personal preference.

This article discusses various facets of command-based robot program design that advanced FRC programmers may want to be aware of when writing code. It is not a prescriptive tutorial, though it presents some recommended best practices. If this level of choice seems daunting, however, many teams have been highly successful while sticking closely to WPILib’s example code and guidelines. However, this discussion may be of interest to intermediate and advanced programmers who want to make their code not only effective, but flexible, easily changeable, and sometimes even beautiful.

27.7.1 Why Care About Organization?

Good code organization will rarely make or break a team's competitive ability—but it does mean easier debugging, faster modifications, nicer-looking code, and happier programmers. While it's impossible to define "good" organization by way of what the code looks like from the inside, it's easier to define in terms of what the robot's software looks like from the outside.

What Good Organization Looks Like

When code is well-designed and well-organized, the code's internal structure is intuitive and easily comprehensible. Cumbersome boilerplate is minimized, meaning that new robot functionality can often be added with just a few lines of code. When a constant value (such as the speed of the robot's intake) needs to be changed, it only needs to change in one place. If multiple programmers are working together, they can easily understand each others' work. Bugs are rare, since it is difficult to accidentally introduce unintended behavior (such as creating a command that does not require necessary subsystems). Implementing more advanced functions like unit tests is easier, since the code is abstracted away from the physical hardware. Programmers are happy (most of the time).

What Bad Organization Looks Like

Poorly organized code often has internal structure that makes little to no sense, even to whoever wrote it. When functionality has to be added or changed, it often breaks unrelated parts of the robot: adding automatic shooter control might introduce a bug in the climbing sequence for unclear reasons. Alternatively, the organizational framework might be so strict that it's impossible to implement necessary behavior, requiring nasty hacks or workarounds. Many lines of boilerplate code are needed for simple robot logic. Constants are scattered across the codebase, and changing basic behavior often requires making the same change to many different files. Collaboration among multiple programmers is difficult or impossible.

27.7.2 Defining Commands

In larger robot codebases, multiple copies of the same command need to be used in many different places. For instance, a command that runs a robot's intake might be used in teleop, bound to a certain button; as part of a complicated command group for an autonomous routine; and as part of a self-test sequence.

As an example, let's look at some ways to define a simple command that simply runs the robot's intake forward at full power until canceled.

Inline Commands

The easiest and most expressive way to do this is with a `StartEndCommand`:

JAVA

```
Command runIntake = Commands.startEnd(() -> intake.set(1.0), () -> intake.set(0.0),␣
↪intake);
```

C++

```
frc2::CommandPtr runIntake = frc2::cmd::StartEnd([&intake] { intake.Set(1.0); }, [&
↪intake] { intake.Set(0.0); }, {&intake});
```

This is sufficient for commands that are only used once. However, for a command like this that might get used in many different autonomous routines and button bindings, inline commands everywhere means a lot of repetitive code:

JAVA

```
// RobotContainer.java
intakeButton.whileTrue(Commands.startEnd(() -> intake.set(1.0), () -> intake.set(0.0),
↪ intake));
Command intakeAndShoot = Commands.startEnd(() -> intake.set(1.0), () -> intake.set(0.
↪0), intake)
    .alongWith(new RunShooter(shooter));
Command autonomousCommand = Commands.sequence(
    Commands.startEnd(() -> intake.set(1.0), () -> intake.set(0.0), intake).
↪withTimeout(5.0),
    Commands.waitSeconds(3.0),
    Commands.startEnd(() -> intake.set(1.0), () -> intake.set(0.0), intake).
↪withTimeout(5.0)
);
```

C++

```
intakeButton.WhileTrue(frc2::cmd::StartEnd([&intake] { intake.Set(1.0); }, [&intake]
↪{ intake.Set(0.0); }, {&intake}));
frc2::CommandPtr intakeAndShoot = frc2::cmd::StartEnd([&intake] { intake.Set(1.0); },␣
↪[&intake] { intake.Set(0.0); }, {&intake})
    .AlongWith(RunShooter(&shooter).ToPtr());
frc2::CommandPtr autonomousCommand = frc2::cmd::Sequence(
  frc2::cmd::StartEnd([&intake] { intake.Set(1.0); }, [&intake] { intake.Set(0.0); },
↪{&intake}).WithTimeout(5.0_s),
  frc2::cmd::Wait(3.0_s),
  frc2::cmd::StartEnd([&intake] { intake.Set(1.0); }, [&intake] { intake.Set(0.0); },
↪{&intake}).WithTimeout(5.0_s)
);
```

Creating one `StartEndCommand` instance and putting it in a variable won't work here, since once an instance of a command is added to a command group it is effectively "owned" by that command group and cannot be used in any other context.

Instance Command Factory Methods

One way to solve this quandary is using the "factory method" design pattern: a function that returns a new object every invocation, according to some specification. Using *command composition*, a factory method can construct a complex command object with merely a few lines of code.

For example, a command like the intake-running command is conceptually related to exactly one subsystem: the `Intake`. As such, it makes sense to put a `runIntakeCommand` method as an instance method of the `Intake` class:

> **Note**
>
> In this document we will name factory methods as `lowerCamelCaseCommand`, but teams may decide on other conventions. In general, it is recommended to end the method name with `Command` if it might otherwise be confused with an ordinary method (e.g. `intake.run` might be the name of a method that simply turns on the intake).

JAVA

```
public class Intake extends SubsystemBase {
  // [code for motor controllers, configuration, etc.]
  // ...
  public Command runIntakeCommand() {
    // implicitly requires `this`
    return this.startEnd(() -> this.set(1.0), () -> this.set(0.0));
  }
}
```

C++

```
frc2::CommandPtr Intake::RunIntakeCommand() {
  // implicitly requires `this`
  return this->StartEnd([this] { this->Set(1.0); }, [this] { this->Set(0.0); });
}
```

Notice how since we are in the `Intake` class, we no longer refer to `intake`; instead, we use the `this` keyword to refer to the current instance.

Since we are inside the `Intake` class, technically we can access `private` variables and methods directly from within the `runIntakeCommand` method, thus not needing intermediary methods. (For example, the `runIntakeCommand` method can directly interface with the motor controller objects instead of calling `set()`.) On the other hand, these intermediary methods can reduce code duplication and increase encapsulation. Like many other choices outlined in this document, this tradeoff is a matter of personal preference on a case-by-case basis.

Using this new factory method in command groups and button bindings is highly expressive:

JAVA

```
intakeButton.whileTrue(intake.runIntakeCommand());
Command intakeAndShoot = intake.runIntakeCommand().alongWith(new RunShooter(shooter));
Command autonomousCommand = Commands.sequence(
    intake.runIntakeCommand().withTimeout(5.0),
    Commands.waitSeconds(3.0),
    intake.runIntakeCommand().withTimeout(5.0)
);
```

C++

```
intakeButton.WhileTrue(intake.RunIntakeCommand());
frc2::CommandPtr intakeAndShoot = intake.RunIntakeCommand().AlongWith(RunShooter(&
↪shooter).ToPtr());
frc2::CommandPtr autonomousCommand = frc2::cmd::Sequence(
  intake.RunIntakeCommand().WithTimeout(5.0_s),
  frc2::cmd::Wait(3.0_s),
  intake.RunIntakeCommand().WithTimeout(5.0_s)
);
```

Adding a parameter to the `runIntakeCommand` method to provide the exact percentage to run the intake is easy and allows for even more flexibility.

JAVA

```
public Command runIntakeCommand(double percent) {
    return new StartEndCommand(() -> this.set(percent), () -> this.set(0.0), this);
}
```

C++

```
frc2::CommandPtr Intake::RunIntakeCommand(double percent) {
  // implicitly requires `this`
  return this->StartEnd([this, percent] { this->Set(percent); }, [this] { this->Set(0.
↪0); });
}
```

For instance, this code creates a command group that runs the intake forwards for two seconds, waits for two seconds, and then runs the intake backwards for five seconds.

JAVA

```
Command intakeRunSequence = intake.runIntakeCommand(1.0).withTimeout(2.0)
    .andThen(Commands.waitSeconds(2.0))
    .andThen(intake.runIntakeCommand(-1.0).withTimeout(5.0));
```

C++

```
frc2::CommandPtr intakeRunSequence = intake.RunIntakeCommand(1.0).WithTimeout(2.0_s)
    .AndThen(frc2::cmd::Wait(2.0_s))
    .AndThen(intake.RunIntakeCommand(-1.0).WithTimeout(5.0_s));
```

This approach is recommended for commands that are conceptually related to only a single subsystem, and is very concise. However, it doesn't fare well with commands related to more than one subsystem: passing in other subsystem objects is unintuitive and can cause race conditions and circular dependencies, and thus should be avoided. Therefore, this approach is best suited for single-subsystem commands, and should be used only for those cases.

Static Command Factories

Instance factory methods work great for single-subsystem commands. However, complicated robot actions (like the ones often required during the autonomous period) typically need to coordinate multiple subsystems at once. When we want to define an inline command that uses multiple subsystems, it doesn't make sense for the command factory to live in any single one of those subsystems. Instead, it can be cleaner to define the command factory methods statically in some external class:

> **Note**
>
> The `sequence` and `parallel` static factories construct sequential and parallel command groups: this is equivalent to the `andThen` and `alongWith` decorators, but can be more readable. Their use is a matter of personal preference.

JAVA

```
public class AutoRoutines {
    public static Command driveAndIntake(Drivetrain drivetrain, Intake intake) {
        return Commands.sequence(
            Commands.parallel(
                drivetrain.driveCommand(0.5, 0.5),
                intake.runIntakeCommand(1.0)
            ).withTimeout(5.0),
            Commands.parallel(
              drivetrain.stopCommand();
              intake.stopCommand();
            )
        );
    }
}
```

C++

```
// TODO
```

Non-Static Command Factories

If we want to avoid the verbosity of adding required subsystems as parameters to our factory methods, we can instead construct an instance of our `AutoRoutines` class and inject our subsystems through the constructor:

JAVA

```
public class AutoRoutines {
    private Drivetrain drivetrain;
    private Intake intake;
    public AutoRoutines(Drivetrain drivetrain, Intake intake) {
        this.drivetrain = drivetrain;
        this.intake = intake;
    }
    public Command driveAndIntake() {
        return Commands.sequence(
            Commands.parallel(
                drivetrain.driveCommand(0.5, 0.5),
                intake.runIntakeCommand(1.0)
            ).withTimeout(5.0),
            Commands.parallel(
                drivetrain.stopCommand();
                intake.stopCommand();
            )
        );
    }
    public Command driveThenIntake() {
        return Commands.sequence(
            drivetrain.driveCommand(0.5, 0.5).withTimeout(5.0),
            drivetrain.stopCommand(),
            intake.runIntakeCommand(1.0).withTimeout(5.0),
            intake.stopCommand()
        );
    }
}
```

C++

```
// TODO
```

Then, elsewhere in our code, we can instantiate an single instance of this class and use it to produce several commands:

JAVA

```
AutoRoutines autoRoutines = new AutoRoutines(this.drivetrain, this.intake);
Command driveAndIntake = autoRoutines.driveAndIntake();
Command driveThenIntake = autoRoutines.driveThenIntake();
Command drivingAndIntakingSequence = Commands.sequence(
  autoRoutines.driveAndIntake(),
  autoRoutines.driveThenIntake()
);
```

C++

```
// TODO
```

Capturing State in Inline Commands

Inline commands are extremely concise and expressive, but do not offer explicit support for commands that have their own internal state (such as a drivetrain trajectory following command, which may encapsulate an entire controller). This is often accomplished by instead writing a Command class, which will be covered later in this article.

However, it is still possible to ergonomically write a stateful command composition using inline syntax, so long as we are working within a factory method. To do so, we declare the state as a method local and "capture" it in our inline definition. For example, consider the following instance command factory to turn a drivetrain to a specific angle with a PID controller:

> **Note**
>
> The `Subsystem.run` and `Subsystem.runOnce` factory methods sugar the creation of a `RunCommand` and an `InstantCommand` requiring `this` subsystem.

JAVA

```
public Command turnToAngle(double targetDegrees) {
    // Create a controller for the inline command to capture
    PIDController controller = new PIDController(Constants.kTurnToAngleP, 0, 0);
    // We can do whatever configuration we want on the created state before returning␣
↪from the factory
    controller.setPositionTolerance(Constants.kTurnToAngleTolerance);
    // Try to turn at a rate proportional to the heading error until we're at the␣
↪setpoint, then stop
    return run(() -> arcadeDrive(0,-controller.calculate(gyro.getHeading(),␣
↪targetDegrees)))
        .until(controller::atSetpoint)
        .andThen(runOnce(() -> arcadeDrive(0, 0)));
}
```

C++

```
// TODO
```

This pattern works very well in Java so long as the captured state is "effectively final" - i.e., it is never reassigned. This means that we cannot directly define and capture primitive types (e.g. *int*, *double*, *boolean*) - to circumvent this, we need to wrap any state primitives in a mutable container type (the same way *PIDController* wraps its internal *kP*, *kI*, and *kD* values).

Writing Command Classes

Another possible way to define reusable commands is to write a class that represents the command. This is typically done by subclassing either `Command` or one of the `CommandGroup` classes.

Subclassing Command

Returning to our simple intake command from earlier, we could do this by creating a new subclass of `Command` that implements the necessary `initialize` and `end` methods.

JAVA

```
public class RunIntakeCommand extends Command {
    private Intake m_intake;
    public RunIntakeCommand(Intake intake) {
        this.m_intake = intake;
        addRequirements(intake);
    }
    @Override
    public void initialize() {
        m_intake.set(1.0);
    }
    @Override
    public void end(boolean interrupted) {
        m_intake.set(0.0);
    }
    // execute() defaults to do nothing
    // isFinished() defaults to return false
}
```

C++

```
// TODO
```

This, however, is just as cumbersome as the original repetitive code, if not more verbose. The only two lines that really matter in this entire file are the two calls to `intake.set()`, yet there are over 20 lines of boilerplate code! Not to mention, doing this for a lot of robot actions quickly clutters up a robot project with dozens of small files. Nevertheless, this might feel more "natural," particularly for programmers who prefer to stick closely to an object-oriented model.

This approach should be used for commands with internal state (not subsystem state!), as the class can have fields to manage said state. It may also be more intuitive to write commands with complex logic as classes, especially for those less experienced with command composition. As the command is detached from any specific subsystem class and the required subsystem objects are injected through the constructor, this approach deals well with commands involving multiple subsystems.

Subclassing Command Groups

If we wish to write composite commands as their own classes, we may write a constructor-only subclass of the most exterior group type. For example, an intake-then-outtake sequence (with single-subsystem commands defined as instance factory methods) can look like this:

JAVA

```
public class IntakeThenOuttake extends SequentialCommandGroup {
    public IntakeThenOuttake(Intake intake) {
        super(
            intake.runIntakeCommand(1.0).withTimeout(2.0),
            new WaitCommand(2.0),
            intake.runIntakeCommand(-1).withTimeout(5.0)
        );
    }
}
```

C++

```
// TODO
```

This is relatively short and minimizes boilerplate. It is also comfortable to use in a purely object-oriented paradigm and may be more acceptable to novice programmers. However, it has some downsides. For one, it is not immediately clear exactly what type of command group this is from the constructor definition: it is better to define this in a more inline and expressive way, particularly when nested command groups start showing up. Additionally, it requires a new file for every single command group, even when the groups are conceptually related.

As with factory methods, state can be defined and captured within the command group subclass constructor, if necessary.

Summary

Approach	Primary Use Case	Single-subsystem Commands	Multi-subsystem Commands	Stateful Commands	Complex Logic Commands
Instance Factory Methods	Single-subsystem commands	Excels at them	No	Yes, but must obey capture rules	Yes
Subclassing Command	Stateful commands	Very verbose	Relatively verbose	Excels at them	Yes; may be more natural than other approaches
Static and Instance Command Factories	Multi-subsystem commands	Yes	Yes	Yes, but must obey capture rules	Yes
Subclassing Command Groups	Multi-subsystem command groups	Yes	Yes	Yes, but must obey capture rules	Yes

27.8 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`.

27.8.1 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` classes).

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.

27.8.2 The `schedule()` Method

To schedule a command, users call the `schedule()` method (Java, C++). This method takes a command, 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.

This method walks through the following steps:

1. Verifies that the command isn't in a composition.
2. *No-op* if scheduler is disabled, command is already scheduled, or robot is disabled and command doesn't *runsWhenDisabled*.
3. If requirements are in use:
   - If all conflicting commands are interruptible, cancel them.
   - If not, don't schedule the new command.
4. Call `initialize()`.

Java

```
202   private void schedule(Command command) {
203     if (command == null) {
204       DriverStation.reportWarning("Tried to schedule a null command", true);
205       return;
206     }
207     if (m_inRunLoop) {
208       m_toSchedule.add(command);
209       return;
210     }
211
212     requireNotComposed(command);
213
214     // Do nothing if the scheduler is disabled, the robot is disabled and the command␣
↪doesn't
215     // run when disabled, or the command is already scheduled.
216     if (m_disabled
217         || isScheduled(command)
218         || RobotState.isDisabled() && !command.runsWhenDisabled()) {
219       return;
220     }
221
222     Set<Subsystem> requirements = command.getRequirements();
223
224     // Schedule the command if the requirements are not currently in-use.
225     if (Collections.disjoint(m_requirements.keySet(), requirements)) {
226       initCommand(command, requirements);
227     } else {
228       // Else check if the requirements that are in use have all have interruptible␣
↪commands,
229       // and if so, interrupt those commands and schedule the new command.
230       for (Subsystem requirement : requirements) {
231         Command requiring = requiring(requirement);
232         if (requiring != null
233             && requiring.getInterruptionBehavior() == InterruptionBehavior.
```

(continues on next page)

(continued from previous page)

```
↪kCancelIncoming) {
234           return;
235         }
236       }
237       for (Subsystem requirement : requirements) {
238         Command requiring = requiring(requirement);
239         if (requiring != null) {
240           cancel(requiring);
241         }
242       }
243       initCommand(command, requirements);
244     }
245   }
```

```
181   private void initCommand(Command command, Set<Subsystem> requirements) {
182     m_scheduledCommands.add(command);
183     for (Subsystem requirement : requirements) {
184       m_requirements.put(requirement, command);
185     }
186     command.initialize();
187     for (Consumer<Command> action : m_initActions) {
188       action.accept(command);
189     }
190
191     m_watchdog.addEpoch(command.getName() + ".initialize()");
```

C++ (Source)

```
114 void CommandScheduler::Schedule(Command* command) {
115   if (m_impl->inRunLoop) {
116     m_impl->toSchedule.emplace_back(command);
117     return;
118   }
119
120   RequireUngrouped(command);
121
122   if (m_impl->disabled || m_impl->scheduledCommands.contains(command) ||
123       (frc::RobotState::IsDisabled() && !command->RunsWhenDisabled())) {
124     return;
125   }
126
127   const auto& requirements = command->GetRequirements();
128
129   wpi::SmallVector<Command*, 8> intersection;
130
131   bool isDisjoint = true;
132   bool allInterruptible = true;
133   for (auto&& i1 : m_impl->requirements) {
134     if (requirements.find(i1.first) != requirements.end()) {
135       isDisjoint = false;
136       allInterruptible &= (i1.second->GetInterruptionBehavior() ==
137                            Command::InterruptionBehavior::kCancelSelf);
138       intersection.emplace_back(i1.second);
139     }
```

(continues on next page)

(continued from previous page)

```
  }

  if (isDisjoint || allInterruptible) {
    if (allInterruptible) {
      for (auto&& cmdToCancel : intersection) {
        Cancel(cmdToCancel);
      }
    }
    m_impl->scheduledCommands.insert(command);
    for (auto&& requirement : requirements) {
      m_impl->requirements[requirement] = command;
    }
    command->Initialize();
    for (auto&& action : m_impl->initActions) {
      action(*command);
    }
    m_watchdog.AddEpoch(command->GetName() + ".Initialize()");
  }
}
```

27.8.3 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. For the full implementation, see the source code (Java, C++).

Step 1: Run Subsystem Periodic Methods

First, the scheduler runs the `periodic()` method of each registered `Subsystem`. In simulation, each subsystem's `simulationPeriodic()` method is called as well.

Java

```
    // Run the periodic method of all registered subsystems.
    for (Subsystem subsystem : m_subsystems.keySet()) {
      subsystem.periodic();
      if (RobotBase.isSimulation()) {
        subsystem.simulationPeriodic();
      }
      m_watchdog.addEpoch(subsystem.getClass().getSimpleName() + ".periodic()");
    }
```

C++ (Source)

```
  // Run the periodic method of all registered subsystems.
  for (auto&& subsystem : m_impl->subsystems) {
    subsystem.getFirst()->Periodic();
    if constexpr (frc::RobotBase::IsSimulation()) {
      subsystem.getFirst()->SimulationPeriodic();
    }
    m_watchdog.AddEpoch("Subsystem Periodic()");
  }
```

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.

> **Note**
>
> If a newly-scheduled command has requirement conflicts with a currently-running command, the currently-running command is interrupted first. The `end(true)` method of the interrupted command is called **before** the `initialize()` method of the new command.

Java

```
    // Poll buttons for new commands to add.
    loopCache.poll();
    m_watchdog.addEpoch("buttons.run()");
```

C++ (Source)

```
  // Poll buttons for new commands to add.
  loopCache->Poll();
  m_watchdog.AddEpoch("buttons.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.

Java

```
295     // Run scheduled commands, remove finished commands.
296     for (Iterator<Command> iterator = m_scheduledCommands.iterator(); iterator.
    ↪hasNext(); ) {
297       Command command = iterator.next();
298
299       if (!command.runsWhenDisabled() && RobotState.isDisabled()) {
300         command.end(true);
301         for (Consumer<Command> action : m_interruptActions) {
302           action.accept(command);
303         }
304         m_requirements.keySet().removeAll(command.getRequirements());
305         iterator.remove();
306         m_watchdog.addEpoch(command.getName() + ".end(true)");
307         continue;
308       }
309
310       command.execute();
311       for (Consumer<Command> action : m_executeActions) {
312         action.accept(command);
313       }
314       m_watchdog.addEpoch(command.getName() + ".execute()");
315       if (command.isFinished()) {
316         command.end(false);
317         for (Consumer<Command> action : m_finishActions) {
318           action.accept(command);
319         }
320         iterator.remove();
321
322         m_requirements.keySet().removeAll(command.getRequirements());
323         m_watchdog.addEpoch(command.getName() + ".end(false)");
324       }
325     }
```

C++ (Source)

```
201   for (Command* command : m_impl->scheduledCommands) {
202     if (!command->RunsWhenDisabled() && frc::RobotState::IsDisabled()) {
203       Cancel(command);
204       continue;
205     }
206
207     command->Execute();
208     for (auto&& action : m_impl->executeActions) {
209       action(*command);
210     }
211     m_watchdog.AddEpoch(command->GetName() + ".Execute()");
212
213     if (command->IsFinished()) {
214       command->End(false);
215       for (auto&& action : m_impl->finishActions) {
216         action(*command);
217       }
218
219       for (auto&& requirement : command->GetRequirements()) {
220         m_impl->requirements.erase(requirement);
221       }
222
223       m_impl->scheduledCommands.erase(command);
224       m_watchdog.AddEpoch(command->GetName() + ".End(false)");
225     }
226   }
```

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.

Java

```
340     // Add default commands for un-required registered subsystems.
341     for (Map.Entry<Subsystem, Command> subsystemCommand : m_subsystems.entrySet()) {
342       if (!m_requirements.containsKey(subsystemCommand.getKey())
343           && subsystemCommand.getValue() != null) {
344         schedule(subsystemCommand.getValue());
345       }
346     }
```

C++ (Source)

```
240   // Add default commands for un-required registered subsystems.
241   for (auto&& subsystem : m_impl->subsystems) {
242     auto s = m_impl->requirements.find(subsystem.getFirst());
243     if (s == m_impl->requirements.end() && subsystem.getSecond()) {
244       Schedule({subsystem.getSecond().get()});
245     }
246   }
```

27.8.4 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()`.

27.8.5 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 methods:

- `onCommandInitialize` (Java, C++) runs a specified action whenever a command is initialized.
- `onCommandExecute` (Java, C++) runs a specified action whenever a command is executed.
- `onCommandFinish` (Java, C++) runs a specified action whenever a command finishes normally (i.e. the `isFinished()` method returned true).
- `onCommandInterrupt` (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 following code from the HatchbotInlined example project (Java, C++):

Java

```
73     // Set the scheduler to log Shuffleboard events for command initialize, interrupt,
↪ finish
74     CommandScheduler.getInstance()
75         .onCommandInitialize(
76             command ->
77                 Shuffleboard.addEventMarker(
78                     "Command initialized", command.getName(), EventImportance.
↪kNormal));
79     CommandScheduler.getInstance()
80         .onCommandInterrupt(
81             command ->
82                 Shuffleboard.addEventMarker(
```

(continues on next page)

(continued from previous page)

```
                    "Command interrupted", command.getName(), EventImportance.
↪kNormal));
    CommandScheduler.getInstance()
        .onCommandFinish(
            command ->
                Shuffleboard.addEventMarker(
                    "Command finished", command.getName(), EventImportance.kNormal));
```

C++ (Source)

```
// Log Shuffleboard events for command initialize, execute, finish, interrupt
frc2::CommandScheduler::GetInstance().OnCommandInitialize(
    [](const frc2::Command& command) {
      frc::Shuffleboard::AddEventMarker(
          "Command initialized", command.GetName(),
          frc::ShuffleboardEventImportance::kNormal);
    });
frc2::CommandScheduler::GetInstance().OnCommandExecute(
    [](const frc2::Command& command) {
      frc::Shuffleboard::AddEventMarker(
          "Command executed", command.GetName(),
          frc::ShuffleboardEventImportance::kNormal);
    });
frc2::CommandScheduler::GetInstance().OnCommandFinish(
    [](const frc2::Command& command) {
      frc::Shuffleboard::AddEventMarker(
          "Command finished", command.GetName(),
          frc::ShuffleboardEventImportance::kNormal);
    });
frc2::CommandScheduler::GetInstance().OnCommandInterrupt(
    [](const frc2::Command& command) {
      frc::Shuffleboard::AddEventMarker(
          "Command interrupted", command.GetName(),
          frc::ShuffleboardEventImportance::kNormal);
    });
```

27.9 A Technical Discussion on C++ Commands

> **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. You do not need to understand the information within this article to use the command-based framework in your robot code.

This article will help you understand the reasoning behind some of the decisions made in the 2020 command-based framework (such as the use of `std::unique_ptr`, CRTP in the form of `CommandHelper<Base, Derived>`, etc.). You do not need to understand the information within this article to use the command-based framework in your robot code.

> **Note**
>
> The model was further changed in 2023, as described *below*.

27.9.1 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 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 compositions are owned by the command composition. 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 compositions 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"; },␣
```

(continues on next page)

(continued from previous page)

```
↪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));
// 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.

27.9.2 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 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` class, `*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
// be called on rvalue references.
temp.emplace_back(std::move(*this).TransferOwnership());
```

27.9.3 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 syntanctic difference in C++ if you create the command group manually in user code.

27.9.4 2023 Updates

After a few years in the new command-based framework, the recommended way to create commands increasingly shifted towards inline commands, decorators, and factory methods. With this paradigm shift, it became evident that the C++ commands model introduced in 2020 and described above has some pain points when used according to the new recommendations.

A significant root cause of most pain points was commands being passed by value in a non-polymorphic way. This made object slicing mistakes rather easy, and changes in composition structure could propagate type changes throughout the codebase: for example, if a `ParallelRaceGroup` were changed to a `ParallelDeadlineGroup`, those type changes would propagate through the codebase. Passing around the object as a `Command` (as done in Java) would result in object slicing.

Additionally, various decorators weren't supported in C++ due to reasons described *above*. As long as decorators were rarely used and were mainly to reduce verbosity (where Java was more verbose than C++), this was less of a problem. Once heavy usage of decorators was recommended, this became more of an issue.

`CommandPtr`

Let's recall the mention of `std::unique_ptr` far above: a value type with only move semantics. This is the ownership model we want!

However, plainly using `std::unique_ptr<Command>` had some drawbacks. Primarily, implementing decorators would be impossible: `unique_ptr` is defined in the standard library so we can't define methods on it, and any methods defined on `Command` wouldn't have access to the owning `unique_ptr`.

The solution is `CommandPtr`: a move-only value class wrapping `unique_ptr`, that we can define methods on.

Commands should be passed around as `CommandPtr`, using `std::move`. All decorators, including those not supported in C++ before, are defined on `CommandPtr` with rvalue-this. The use of rvalues, move-only semantics, and clear ownership makes it very easy to avoid mistakes such as adding the same command instance to more than one *command composition*.

In addition to decorators, `CommandPtr` instances also define utility methods such as `Schedule()`, `IsScheduled()`. `CommandPtr` instances can be used in nearly almost every way command objects can be used in Java: they can be moved into trigger bindings, default commands, and so on. For the few things that require a `Command*` (such as non-owning trigger bindings), a raw pointer to the owned command can be retrieved using `get()`.

There are multiple ways to get a `CommandPtr` instance:

- `CommandPtr`-returning factories are present in the `frc2::cmd` namespace in the `Commands.h` header for almost all command types. For multi-command compositions, there is a vector-taking overload as well as a variadic-templated overload for multiple `CommandPtr` instances.

- All decorators, including those defined on `Command`, return `CommandPtr`. This has allowed defining almost all decorators on `Command`, so a decorator chain can start from a `Command`.
- A `ToPtr()` method has been added to the CRTP, akin to `TransferOwnership`. This is useful especially for user-defined command classes, as well as other command classes that don't have factories.

For instance, consider the following from the HatchbotInlined example project:

```
frc2::CommandPtr autos::ComplexAuto(DriveSubsystem* drive,
                                    HatchSubsystem* hatch) {
  return frc2::cmd::Sequence(
      // Drive forward the specified distance
      frc2::FunctionalCommand(
          // Reset encoders on command start
          [drive] { drive->ResetEncoders(); },
          // Drive forward while the command is executing
          [drive] { drive->ArcadeDrive(kAutoDriveSpeed, 0); },
          // Stop driving at the end of the command
          [drive](bool interrupted) { drive->ArcadeDrive(0, 0); },
          // End the command when the robot's driven distance exceeds the
          // desired value
          [drive] {
            return drive->GetAverageEncoderDistance() >=
                   kAutoDriveDistanceInches;
          },
          // Requires the drive subsystem
          {drive})
          .ToPtr(),
      // Release the hatch
      hatch->ReleaseHatchCommand(),
      // Drive backward the specified distance
      // Drive forward the specified distance
      frc2::FunctionalCommand(
          // Reset encoders on command start
          [drive] { drive->ResetEncoders(); },
          // Drive backward while the command is executing
          [drive] { drive->ArcadeDrive(-kAutoDriveSpeed, 0); },
          // Stop driving at the end of the command
          [drive](bool interrupted) { drive->ArcadeDrive(0, 0); },
          // End the command when the robot's driven distance exceeds the
          // desired value
          [drive] {
            return drive->GetAverageEncoderDistance() <=
                   kAutoBackupDistanceInches;
          },
          // Requires the drive subsystem
          {drive})
          .ToPtr());
}
```

To avoid breakage, command compositions still use `unique_ptr<Command>`, so `CommandPtr` instances can be destructured into a `unique_ptr<Command>` using the `Unwrap()` rvalue-this method. For vectors, the static `CommandPtr::UnwrapVector(vector<CommandPtr>)` function exists.

27.10 PID Control in Command-based

> **Note**
>
> For a description of the WPILib PID control features used by these command-based wrappers, see *PID Control in WPILib*.

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. The following example is from the RapidReactCommandBot example project (Java, C++) and shows how PIDControllers can be used within the command-based framework:

Java

```
5  package edu.wpi.first.wpilibj.examples.rapidreactcommandbot.subsystems;
6
7  import static edu.wpi.first.wpilibj2.command.Commands.parallel;
8  import static edu.wpi.first.wpilibj2.command.Commands.waitUntil;
9
10 import edu.wpi.first.epilogue.Logged;
11 import edu.wpi.first.math.controller.PIDController;
12 import edu.wpi.first.math.controller.SimpleMotorFeedforward;
13 import edu.wpi.first.wpilibj.Encoder;
14 import edu.wpi.first.wpilibj.examples.rapidreactcommandbot.Constants.ShooterConstants;
15 import edu.wpi.first.wpilibj.motorcontrol.PWMSparkMax;
16 import edu.wpi.first.wpilibj2.command.Command;
17 import edu.wpi.first.wpilibj2.command.SubsystemBase;
18
19 @Logged
20 public class Shooter extends SubsystemBase {
21   private final PWMSparkMax m_shooterMotor = new PWMSparkMax(ShooterConstants.
   ↪kShooterMotorPort);
22   private final PWMSparkMax m_feederMotor = new PWMSparkMax(ShooterConstants.
   ↪kFeederMotorPort);
23   private final Encoder m_shooterEncoder =
24       new Encoder(
25           ShooterConstants.kEncoderPorts[0],
26           ShooterConstants.kEncoderPorts[1],
27           ShooterConstants.kEncoderReversed);
28   private final SimpleMotorFeedforward m_shooterFeedforward =
29       new SimpleMotorFeedforward(
30           ShooterConstants.kSVolts, ShooterConstants.kVVoltSecondsPerRotation);
31   private final PIDController m_shooterFeedback = new PIDController(ShooterConstants.
   ↪kP, 0.0, 0.0);
32
33   /** The shooter subsystem for the robot. */
34   public Shooter() {
35     m_shooterFeedback.setTolerance(ShooterConstants.kShooterToleranceRPS);
36     m_shooterEncoder.setDistancePerPulse(ShooterConstants.kEncoderDistancePerPulse);
37
38     // Set default command to turn off both the shooter and feeder motors, and then␣
   ↪idle
39     setDefaultCommand(
```

(continues on next page)

(continued from previous page)

```
        runOnce(
                () -> {
                  m_shooterMotor.disable();
                  m_feederMotor.disable();
                })
            .andThen(run(() -> {}))
            .withName("Idle"));
  }

  /**
   * Returns a command to shoot the balls currently stored in the robot. Spins the shooter flywheel
   * up to the specified setpoint, and then runs the feeder motor.
   *
   * @param setpointRotationsPerSecond The desired shooter velocity
   */
  public Command shootCommand(double setpointRotationsPerSecond) {
    return parallel(
            // Run the shooter flywheel at the desired setpoint using feedforward and feedback
            run(
                () -> {
                  m_shooterMotor.set(
                      m_shooterFeedforward.calculate(setpointRotationsPerSecond)
                          + m_shooterFeedback.calculate(
                              m_shooterEncoder.getRate(), setpointRotationsPerSecond));
                }),

            // Wait until the shooter has reached the setpoint, and then run the feeder
            waitUntil(m_shooterFeedback::atSetpoint).andThen(() -> m_feederMotor.set(1)))
        .withName("Shoot");
  }
}
```

C++

```
#pragma once

#include <functional>

#include <frc/Encoder.h>
#include <frc/controller/PIDController.h>
#include <frc/controller/SimpleMotorFeedforward.h>
#include <frc/motorcontrol/PWMSparkMax.h>
#include <frc2/command/CommandPtr.h>
#include <frc2/command/SubsystemBase.h>
#include <units/angle.h>
#include <units/angular_velocity.h>

#include "Constants.h"

```

(continues on next page)

(continued from previous page)

```
class Shooter : public frc2::SubsystemBase {
 public:
  Shooter();

  /**
   * Returns a command to shoot the balls currently stored in the robot. Spins
   * the shooter flywheel up to the specified setpoint, and then runs the feeder
   * motor.
   *
   * @param setpointRotationsPerSecond The desired shooter velocity
   */
  frc2::CommandPtr ShootCommand(units::turns_per_second_t setpoint);

 private:
  frc::PWMSparkMax m_shooterMotor{ShooterConstants::kShooterMotorPort};
  frc::PWMSparkMax m_feederMotor{ShooterConstants::kFeederMotorPort};

  frc::Encoder m_shooterEncoder{ShooterConstants::kEncoderPorts[0],
                                ShooterConstants::kEncoderPorts[1],
                                ShooterConstants::kEncoderReversed};
  frc::SimpleMotorFeedforward<units::radians> m_shooterFeedforward{
      ShooterConstants::kS, ShooterConstants::kV};
  frc::PIDController m_shooterFeedback{ShooterConstants::kP, 0.0, 0.0};
};
```

C++ (Source)

```
#include "subsystems/Shooter.h"

#include <frc2/command/Commands.h>

Shooter::Shooter() {
  m_shooterFeedback.SetTolerance(ShooterConstants::kShooterTolerance.value());
  m_shooterEncoder.SetDistancePerPulse(
      ShooterConstants::kEncoderDistancePerPulse);

  SetDefaultCommand(RunOnce([this] {
                      m_shooterMotor.Disable();
                      m_feederMotor.Disable();
                    })
                        .AndThen(Run([] {}))
                        .WithName("Idle"));
}

frc2::CommandPtr Shooter::ShootCommand(units::turns_per_second_t setpoint) {
  return frc2::cmd::Parallel(
             // Run the shooter flywheel at the desired setpoint using
             // feedforward and feedback
             Run([this, setpoint] {
               m_shooterMotor.SetVoltage(
                   m_shooterFeedforward.Calculate(setpoint) +
                   units::volt_t(m_shooterFeedback.Calculate(
                       m_shooterEncoder.GetRate(), setpoint.value())));
             }),
```

(continues on next page)

(continued from previous page)

```
            // Wait until the shooter has reached the setpoint, and then
            // run the feeder
            frc2::cmd::WaitUntil([this] {
              return m_shooterFeedback.AtSetpoint();
            }).AndThen([this] { m_feederMotor.Set(1.0); }))
      .WithName("Shoot");
}
```

A `PIDController` is declared inside the `Shooter` subsystem. It is used by `ShootCommand` alongside a feedforward to spin the shooter flywheel to the specified velocity. Once the `PIDController` reaches the specified velocity, the `ShootCommand` runs the feeder.

27.11 Motion Profiling in Command-based

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` class, used on its own, is most useful when composed with external controllers, such as a "smart" motor controller with a built-in PID functionality. For combining trapezoidal motion profiling with WPILib's `PIDController`, see *Combining Motion Profiling and PID in Command-Based*.

When controlling a mechanism, 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++).

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*.

The following examples are taken from the DriveDistanceOffboard example project (Java, C++):

Java

```
package edu.wpi.first.wpilibj.examples.drivedistanceoffboard.subsystems;

import edu.wpi.first.math.controller.SimpleMotorFeedforward;
import edu.wpi.first.math.trajectory.TrapezoidProfile;
import edu.wpi.first.math.trajectory.TrapezoidProfile.State;
import edu.wpi.first.util.sendable.SendableRegistry;
import edu.wpi.first.wpilibj.RobotController;
import edu.wpi.first.wpilibj.Timer;
import edu.wpi.first.wpilibj.drive.DifferentialDrive;
import edu.wpi.first.wpilibj.examples.drivedistanceoffboard.Constants.DriveConstants;
import edu.wpi.first.wpilibj.examples.drivedistanceoffboard.
↪ExampleSmartMotorController;
import edu.wpi.first.wpilibj2.command.Command;
import edu.wpi.first.wpilibj2.command.SubsystemBase;

public class DriveSubsystem extends SubsystemBase {
  // The motors on the left side of the drive.
  private final ExampleSmartMotorController m_leftLeader =
      new ExampleSmartMotorController(DriveConstants.kLeftMotor1Port);

  private final ExampleSmartMotorController m_leftFollower =
      new ExampleSmartMotorController(DriveConstants.kLeftMotor2Port);

  // The motors on the right side of the drive.
  private final ExampleSmartMotorController m_rightLeader =
      new ExampleSmartMotorController(DriveConstants.kRightMotor1Port);

  private final ExampleSmartMotorController m_rightFollower =
      new ExampleSmartMotorController(DriveConstants.kRightMotor2Port);

  // The feedforward controller.
  private final SimpleMotorFeedforward m_feedforward =
      new SimpleMotorFeedforward(
          DriveConstants.ksVolts,
          DriveConstants.kvVoltSecondsPerMeter,
          DriveConstants.kaVoltSecondsSquaredPerMeter);

  // The robot's drive
  private final DifferentialDrive m_drive =
      new DifferentialDrive(m_leftLeader::set, m_rightLeader::set);

  // The trapezoid profile
  private final TrapezoidProfile m_profile =
      new TrapezoidProfile(
          new TrapezoidProfile.Constraints(
              DriveConstants.kMaxSpeedMetersPerSecond,
              DriveConstants.kMaxAccelerationMetersPerSecondSquared));

  // The timer
  private final Timer m_timer = new Timer();

  /** Creates a new DriveSubsystem. */
  public DriveSubsystem() {
    SendableRegistry.addChild(m_drive, m_leftLeader);
    SendableRegistry.addChild(m_drive, m_rightLeader);
```

(continues on next page)

(continued from previous page)

```

    // We need to invert one side of the drivetrain so that positive voltages
    // result in both sides moving forward. Depending on how your robot's
    // gearbox is constructed, you might have to invert the left side instead.
    m_rightLeader.setInverted(true);

    m_leftFollower.follow(m_leftLeader);
    m_rightFollower.follow(m_rightLeader);

    m_leftLeader.setPID(DriveConstants.kp, 0, 0);
    m_rightLeader.setPID(DriveConstants.kp, 0, 0);
  }

  /**
   * Drives the robot using arcade controls.
   *
   * @param fwd the commanded forward movement
   * @param rot the commanded rotation
   */
  public void arcadeDrive(double fwd, double rot) {
    m_drive.arcadeDrive(fwd, rot);
  }

  /**
   * Attempts to follow the given drive states using offboard PID.
   *
   * @param currentLeft The current left wheel state.
   * @param currentRight The current right wheel state.
   * @param nextLeft The next left wheel state.
   * @param nextRight The next right wheel state.
   */
  public void setDriveStates(
      TrapezoidProfile.State currentLeft,
      TrapezoidProfile.State currentRight,
      TrapezoidProfile.State nextLeft,
      TrapezoidProfile.State nextRight) {
    // Feedforward is divided by battery voltage to normalize it to [-1, 1]
    m_leftLeader.setSetpoint(
        ExampleSmartMotorController.PIDMode.kPosition,
        currentLeft.position,
        m_feedforward.calculateWithVelocities(currentLeft.velocity, nextLeft.velocity)
            / RobotController.getBatteryVoltage());
    m_rightLeader.setSetpoint(
        ExampleSmartMotorController.PIDMode.kPosition,
        currentRight.position,
        m_feedforward.calculateWithVelocities(currentLeft.velocity, nextLeft.velocity)
            / RobotController.getBatteryVoltage());
  }

  /**
   * Returns the left encoder distance.
   *
   * @return the left encoder distance
   */
  public double getLeftEncoderDistance() {
    return m_leftLeader.getEncoderDistance();
  }
```

(continues on next page)

(continued from previous page)

```
116
117   /**
118    * Returns the right encoder distance.
119    *
120    * @return the right encoder distance
121    */
122   public double getRightEncoderDistance() {
123     return m_rightLeader.getEncoderDistance();
124   }
125
126   /** Resets the drive encoders. */
127   public void resetEncoders() {
128     m_leftLeader.resetEncoder();
129     m_rightLeader.resetEncoder();
130   }
131
132   /**
133    * Sets the max output of the drive. Useful for scaling the drive to drive more␣
↪slowly.
134    *
135    * @param maxOutput the maximum output to which the drive will be constrained
136    */
137   public void setMaxOutput(double maxOutput) {
138     m_drive.setMaxOutput(maxOutput);
139   }
140
141   /**
142    * Creates a command to drive forward a specified distance using a motion profile.
143    *
144    * @param distance The distance to drive forward.
145    * @return A command.
146    */
147   public Command profiledDriveDistance(double distance) {
148     return startRun(
149             () -> {
150               // Restart timer so profile setpoints start at the beginning
151               m_timer.restart();
152               resetEncoders();
153             },
154             () -> {
155               // Current state never changes, so we need to use a timer to get the␣
↪setpoints we need
156               // to be at
157               var currentTime = m_timer.get();
158               var currentSetpoint =
159                   m_profile.calculate(currentTime, new State(), new State(distance,␣
↪0));
160               var nextSetpoint =
161                   m_profile.calculate(
162                       currentTime + DriveConstants.kDt, new State(), new␣
↪State(distance, 0));
163               setDriveStates(currentSetpoint, currentSetpoint, nextSetpoint,␣
↪nextSetpoint);
164             })
165         .until(() -> m_profile.isFinished(0));
166   }
167
```

(continues on next page)

(continued from previous page)

```
  private double m_initialLeftDistance;
  private double m_initialRightDistance;

  /**
   * Creates a command to drive forward a specified distance using a motion profile without
   * resetting the encoders.
   *
   * @param distance The distance to drive forward.
   * @return A command.
   */
  public Command dynamicProfiledDriveDistance(double distance) {
    return startRun(
            () -> {
              // Restart timer so profile setpoints start at the beginning
              m_timer.restart();
              // Store distance so we know the target distance for each encoder
              m_initialLeftDistance = getLeftEncoderDistance();
              m_initialRightDistance = getRightEncoderDistance();
            },
            () -> {
              // Current state never changes for the duration of the command, so we need to use a
              // timer to get the setpoints we need to be at
              var currentTime = m_timer.get();
              var currentLeftSetpoint =
                  m_profile.calculate(
                      currentTime,
                      new State(m_initialLeftDistance, 0),
                      new State(m_initialLeftDistance + distance, 0));
              var currentRightSetpoint =
                  m_profile.calculate(
                      currentTime,
                      new State(m_initialRightDistance, 0),
                      new State(m_initialRightDistance + distance, 0));
              var nextLeftSetpoint =
                  m_profile.calculate(
                      currentTime + DriveConstants.kDt,
                      new State(m_initialLeftDistance, 0),
                      new State(m_initialLeftDistance + distance, 0));
              var nextRightSetpoint =
                  m_profile.calculate(
                      currentTime + DriveConstants.kDt,
                      new State(m_initialRightDistance, 0),
                      new State(m_initialRightDistance + distance, 0));
              setDriveStates(
                  currentLeftSetpoint, currentRightSetpoint, nextLeftSetpoint, nextRightSetpoint);
            })
        .until(() -> m_profile.isFinished(0));
  }
}
```

C++ (Header)

```
#pragma once

#include <frc/Encoder.h>
#include <frc/Timer.h>
#include <frc/controller/SimpleMotorFeedforward.h>
#include <frc/drive/DifferentialDrive.h>
#include <frc/trajectory/TrapezoidProfile.h>
#include <frc2/command/CommandPtr.h>
#include <frc2/command/SubsystemBase.h>
#include <units/length.h>

#include "Constants.h"
#include "ExampleSmartMotorController.h"

class DriveSubsystem : public frc2::SubsystemBase {
 public:
  DriveSubsystem();

  /**
   * Will be called periodically whenever the CommandScheduler runs.
   */
  void Periodic() override;

  // Subsystem methods go here.

  /**
   * Attempts to follow the given drive states using offboard PID.
   *
   * @param currentLeft The current left wheel state.
   * @param currentRight The current right wheel state.
   * @param nextLeft The next left wheel state.
   * @param nextRight The next right wheel state.
   */
  void SetDriveStates(frc::TrapezoidProfile<units::meters>::State currentLeft,
                      frc::TrapezoidProfile<units::meters>::State currentRight,
                      frc::TrapezoidProfile<units::meters>::State nextLeft,
                      frc::TrapezoidProfile<units::meters>::State nextRight);

  /**
   * Drives the robot using arcade controls.
   *
   * @param fwd the commanded forward movement
   * @param rot the commanded rotation
   */
  void ArcadeDrive(double fwd, double rot);

  /**
   * Resets the drive encoders to currently read a position of 0.
   */
  void ResetEncoders();

  /**
   * Gets the distance of the left encoder.
   *
   * @return the average of the TWO encoder readings
```

(continues on next page)

(continued from previous page)

```
   */
  units::meter_t GetLeftEncoderDistance();

  /**
   * Gets the distance of the right encoder.
   *
   * @return the average of the TWO encoder readings
   */
  units::meter_t GetRightEncoderDistance();

  /**
   * Sets the max output of the drive.  Useful for scaling the drive to drive
   * more slowly.
   *
   * @param maxOutput the maximum output to which the drive will be constrained
   */
  void SetMaxOutput(double maxOutput);

  /**
   * Creates a command to drive forward a specified distance using a motion
   * profile.
   *
   * @param distance The distance to drive forward.
   * @return A command.
   */
  frc2::CommandPtr ProfiledDriveDistance(units::meter_t distance);

  /**
   * Creates a command to drive forward a specified distance using a motion
   * profile without resetting the encoders.
   *
   * @param distance The distance to drive forward.
   * @return A command.
   */
  frc2::CommandPtr DynamicProfiledDriveDistance(units::meter_t distance);

 private:
  frc::TrapezoidProfile<units::meters> m_profile{
      {DriveConstants::kMaxSpeed, DriveConstants::kMaxAcceleration}};
  frc::Timer m_timer;
  units::meter_t m_initialLeftDistance;
  units::meter_t m_initialRightDistance;
  // Components (e.g. motor controllers and sensors) should generally be
  // declared private and exposed only through public methods.

  // The motor controllers
  ExampleSmartMotorController m_leftLeader;
  ExampleSmartMotorController m_leftFollower;
  ExampleSmartMotorController m_rightLeader;
  ExampleSmartMotorController m_rightFollower;

  // A feedforward component for the drive
  frc::SimpleMotorFeedforward<units::meters> m_feedforward;

  // The robot's drive
  frc::DifferentialDrive m_drive{
      [&](double output) { m_leftLeader.Set(output); },
```

(continues on next page)

(continued from previous page)

```
      [&](double output) { m_rightLeader.Set(output); }};
};
```

C++ (Source)

```
#include "subsystems/DriveSubsystem.h"

#include <frc/RobotController.h>

using namespace DriveConstants;

DriveSubsystem::DriveSubsystem()
    : m_leftLeader{kLeftMotor1Port},
      m_leftFollower{kLeftMotor2Port},
      m_rightLeader{kRightMotor1Port},
      m_rightFollower{kRightMotor2Port},
      m_feedforward{ks, kv, ka} {
  wpi::SendableRegistry::AddChild(&m_drive, &m_leftLeader);
  wpi::SendableRegistry::AddChild(&m_drive, &m_rightLeader);

  // We need to invert one side of the drivetrain so that positive voltages
  // result in both sides moving forward. Depending on how your robot's
  // gearbox is constructed, you might have to invert the left side instead.
  m_rightLeader.SetInverted(true);

  m_leftFollower.Follow(m_leftLeader);
  m_rightFollower.Follow(m_rightLeader);

  m_leftLeader.SetPID(kp, 0, 0);
  m_rightLeader.SetPID(kp, 0, 0);
}

void DriveSubsystem::Periodic() {
  // Implementation of subsystem periodic method goes here.
}

void DriveSubsystem::SetDriveStates(
    frc::TrapezoidProfile<units::meters>::State currentLeft,
    frc::TrapezoidProfile<units::meters>::State currentRight,
    frc::TrapezoidProfile<units::meters>::State nextLeft,
    frc::TrapezoidProfile<units::meters>::State nextRight) {
  // Feedforward is divided by battery voltage to normalize it to [-1, 1]
  m_leftLeader.SetSetpoint(
      ExampleSmartMotorController::PIDMode::kPosition,
      currentLeft.position.value(),
      m_feedforward.Calculate(currentLeft.velocity, nextLeft.velocity) /
          frc::RobotController::GetBatteryVoltage());
  m_rightLeader.SetSetpoint(
      ExampleSmartMotorController::PIDMode::kPosition,
      currentRight.position.value(),
      m_feedforward.Calculate(currentRight.velocity, nextRight.velocity) /
          frc::RobotController::GetBatteryVoltage());
}

```

(continues on next page)

(continued from previous page)

```
void DriveSubsystem::ArcadeDrive(double fwd, double rot) {
  m_drive.ArcadeDrive(fwd, rot);
}

void DriveSubsystem::ResetEncoders() {
  m_leftLeader.ResetEncoder();
  m_rightLeader.ResetEncoder();
}

units::meter_t DriveSubsystem::GetLeftEncoderDistance() {
  return units::meter_t{m_leftLeader.GetEncoderDistance()};
}

units::meter_t DriveSubsystem::GetRightEncoderDistance() {
  return units::meter_t{m_rightLeader.GetEncoderDistance()};
}

void DriveSubsystem::SetMaxOutput(double maxOutput) {
  m_drive.SetMaxOutput(maxOutput);
}

frc2::CommandPtr DriveSubsystem::ProfiledDriveDistance(
    units::meter_t distance) {
  return StartRun(
             [&] {
               // Restart timer so profile setpoints start at the beginning
               m_timer.Restart();
               ResetEncoders();
             },
             [&] {
               // Current state never changes, so we need to use a timer to get
               // the setpoints we need to be at
               auto currentTime = m_timer.Get();
               auto currentSetpoint =
                   m_profile.Calculate(currentTime, {}, {distance, 0_mps});
               auto nextSetpoint = m_profile.Calculate(currentTime + kDt, {},
                                                       {distance, 0_mps});
               SetDriveStates(currentSetpoint, currentSetpoint, nextSetpoint,
                              nextSetpoint);
             })
      .Until([&] { return m_profile.IsFinished(0_s); });
}

frc2::CommandPtr DriveSubsystem::DynamicProfiledDriveDistance(
    units::meter_t distance) {
  return StartRun(
             [&] {
               // Restart timer so profile setpoints start at the beginning
               m_timer.Restart();
               // Store distance so we know the target distance for each encoder
               m_initialLeftDistance = GetLeftEncoderDistance();
               m_initialRightDistance = GetRightEncoderDistance();
             },
             [&] {
               // Current state never changes for the duration of the command,
               // so we need to use a timer to get the setpoints we need to be
               // at
```

(continues on next page)

(continued from previous page)

```
                auto currentTime = m_timer.Get();

                auto currentLeftSetpoint = m_profile.Calculate(
                    currentTime, {m_initialLeftDistance, 0_mps},
                    {m_initialLeftDistance + distance, 0_mps});
                auto currentRightSetpoint = m_profile.Calculate(
                    currentTime, {m_initialRightDistance, 0_mps},
                    {m_initialRightDistance + distance, 0_mps});

                auto nextLeftSetpoint = m_profile.Calculate(
                    currentTime + kDt, {m_initialLeftDistance, 0_mps},
                    {m_initialLeftDistance + distance, 0_mps});
                auto nextRightSetpoint = m_profile.Calculate(
                    currentTime + kDt, {m_initialRightDistance, 0_mps},
                    {m_initialRightDistance + distance, 0_mps});
                SetDriveStates(currentLeftSetpoint, currentRightSetpoint,
                               nextLeftSetpoint, nextRightSetpoint);
            })
      .Until([&] { return m_profile.IsFinished(0_s); });
}
```

There are two commands in this example. They function very similarly, with the main difference being that one resets encoders, and the other doesn't, which allows encoder data to be preserved.

The subsystem contains a `TrapezoidProfile` with a `Timer`. The timer is used along with a `kDt` constant of 0.02 seconds to calculate the current and next states from the `TrapezoidProfile`. The current state is fed to the "smart" motor controller for PID control, while the current and next state are used to calculate feedforward outputs. Both commands end when `isFinished(0)` returns true, which means that the profile has reached the goal state.

27.12 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*.

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. The following example is from the RapidReactCommandBot example project (Java, C++) and shows how `ProfiledPIDController` can be used within the command-based framework to turn a drivetrain to a specified angle:

Java

```
package edu.wpi.first.wpilibj.examples.rapidreactcommandbot.subsystems;

import edu.wpi.first.epilogue.Logged;
import edu.wpi.first.epilogue.NotLogged;
import edu.wpi.first.math.controller.ProfiledPIDController;
import edu.wpi.first.math.controller.SimpleMotorFeedforward;
import edu.wpi.first.math.trajectory.TrapezoidProfile;
import edu.wpi.first.util.sendable.SendableRegistry;
import edu.wpi.first.wpilibj.ADXRS450_Gyro;
import edu.wpi.first.wpilibj.Encoder;
import edu.wpi.first.wpilibj.RobotController;
import edu.wpi.first.wpilibj.drive.DifferentialDrive;
import edu.wpi.first.wpilibj.examples.rapidreactcommandbot.Constants.DriveConstants;
import edu.wpi.first.wpilibj.motorcontrol.PWMSparkMax;
import edu.wpi.first.wpilibj2.command.Command;
import edu.wpi.first.wpilibj2.command.SubsystemBase;
import java.util.function.DoubleSupplier;

@Logged
public class Drive extends SubsystemBase {
  // The motors on the left side of the drive.
  private final PWMSparkMax m_leftLeader = new PWMSparkMax(DriveConstants.
↪kLeftMotor1Port);
  private final PWMSparkMax m_leftFollower = new PWMSparkMax(DriveConstants.
↪kLeftMotor2Port);

  // The motors on the right side of the drive.
  private final PWMSparkMax m_rightLeader = new PWMSparkMax(DriveConstants.
↪kRightMotor1Port);
  private final PWMSparkMax m_rightFollower = new PWMSparkMax(DriveConstants.
↪kRightMotor2Port);

  // The robot's drive
  @NotLogged // Would duplicate motor data, there's no point sending it twice
  private final DifferentialDrive m_drive =
      new DifferentialDrive(m_leftLeader::set, m_rightLeader::set);

  // The left-side drive encoder
  private final Encoder m_leftEncoder =
      new Encoder(
          DriveConstants.kLeftEncoderPorts[0],
          DriveConstants.kLeftEncoderPorts[1],
          DriveConstants.kLeftEncoderReversed);

  // The right-side drive encoder
  private final Encoder m_rightEncoder =
      new Encoder(
          DriveConstants.kRightEncoderPorts[0],
          DriveConstants.kRightEncoderPorts[1],
          DriveConstants.kRightEncoderReversed);

  private final ADXRS450_Gyro m_gyro = new ADXRS450_Gyro();
  private final ProfiledPIDController m_controller =
      new ProfiledPIDController(
          DriveConstants.kTurnP,
```

(continues on next page)

(continued from previous page)

```
56            DriveConstants.kTurnI,
57            DriveConstants.kTurnD,
58            new TrapezoidProfile.Constraints(
59                DriveConstants.kMaxTurnRateDegPerS,
60                DriveConstants.kMaxTurnAccelerationDegPerSSquared));
61    private final SimpleMotorFeedforward m_feedforward =
62        new SimpleMotorFeedforward(
63            DriveConstants.ksVolts,
64            DriveConstants.kvVoltSecondsPerDegree,
65            DriveConstants.kaVoltSecondsSquaredPerDegree);
66
67    /** Creates a new Drive subsystem. */
68    public Drive() {
69      SendableRegistry.addChild(m_drive, m_leftLeader);
70      SendableRegistry.addChild(m_drive, m_rightLeader);
71
72      m_leftLeader.addFollower(m_leftFollower);
73      m_rightLeader.addFollower(m_rightFollower);
74
75      // We need to invert one side of the drivetrain so that positive voltages
76      // result in both sides moving forward. Depending on how your robot's
77      // gearbox is constructed, you might have to invert the left side instead.
78      m_rightLeader.setInverted(true);
79
80      // Sets the distance per pulse for the encoders
81      m_leftEncoder.setDistancePerPulse(DriveConstants.kEncoderDistancePerPulse);
82      m_rightEncoder.setDistancePerPulse(DriveConstants.kEncoderDistancePerPulse);
83
84      // Set the controller to be continuous (because it is an angle controller)
85      m_controller.enableContinuousInput(-180, 180);
86      // Set the controller tolerance - the delta tolerance ensures the robot is␣
    ↪stationary at the
87      // setpoint before it is considered as having reached the reference
88      m_controller.setTolerance(
89          DriveConstants.kTurnToleranceDeg, DriveConstants.kTurnRateToleranceDegPerS);
90    }
91
92    /**
93     * Returns a command that drives the robot with arcade controls.
94     *
95     * @param fwd the commanded forward movement
96     * @param rot the commanded rotation
97     */
98    public Command arcadeDriveCommand(DoubleSupplier fwd, DoubleSupplier rot) {
99      // A split-stick arcade command, with forward/backward controlled by the left
100     // hand, and turning controlled by the right.
101     return run(() -> m_drive.arcadeDrive(fwd.getAsDouble(), rot.getAsDouble()))
102         .withName("arcadeDrive");
103   }
104
105   /**
106    * Returns a command that drives the robot forward a specified distance at a␣
    ↪specified speed.
107    *
108    * @param distanceMeters The distance to drive forward in meters
109    * @param speed The fraction of max speed at which to drive
110    */
```

(continues on next page)

(continued from previous page)

```
  public Command driveDistanceCommand(double distanceMeters, double speed) {
    return runOnce(
            () -> {
              // Reset encoders at the start of the command
              m_leftEncoder.reset();
              m_rightEncoder.reset();
            })
        // Drive forward at specified speed
        .andThen(run(() -> m_drive.arcadeDrive(speed, 0)))
        // End command when we've traveled the specified distance
        .until(
            () ->
                Math.max(m_leftEncoder.getDistance(), m_rightEncoder.getDistance())
                    >= distanceMeters)
        // Stop the drive when the command ends
        .finallyDo(interrupted -> m_drive.stopMotor());
  }

  /**
   * Returns a command that turns to robot to the specified angle using a motion␣
↪profile and PID
   * controller.
   *
   * @param angleDeg The angle to turn to
   */
  public Command turnToAngleCommand(double angleDeg) {
    return startRun(
            () -> m_controller.reset(m_gyro.getRotation2d().getDegrees()),
            () ->
                m_drive.arcadeDrive(
                    0,
                    m_controller.calculate(m_gyro.getRotation2d().getDegrees(),␣
↪angleDeg)
                        // Divide feedforward voltage by battery voltage to normalize␣
↪it to [-1, 1]
                        + m_feedforward.calculate(m_controller.getSetpoint().velocity)
                            / RobotController.getBatteryVoltage()))
        .until(m_controller::atGoal)
        .finallyDo(() -> m_drive.arcadeDrive(0, 0));
  }
}
```

C++ (Header)

```
#pragma once

#include <functional>

#include <frc/ADXRS450_Gyro.h>
#include <frc/Encoder.h>
#include <frc/controller/ProfiledPIDController.h>
#include <frc/controller/SimpleMotorFeedforward.h>
#include <frc/drive/DifferentialDrive.h>
#include <frc/motorcontrol/PWMSparkMax.h>
```

(continues on next page)

(continued from previous page)

```
#include <frc2/command/CommandPtr.h>
#include <frc2/command/SubsystemBase.h>
#include <units/angle.h>
#include <units/length.h>

#include "Constants.h"

class Drive : public frc2::SubsystemBase {
 public:
  Drive();
  /**
   * Returns a command that drives the robot with arcade controls.
   *
   * @param fwd the commanded forward movement
   * @param rot the commanded rotation
   */
  frc2::CommandPtr ArcadeDriveCommand(std::function<double()> fwd,
                                      std::function<double()> rot);

  /**
   * Returns a command that drives the robot forward a specified distance at a
   * specified speed.
   *
   * @param distance The distance to drive forward in meters
   * @param speed The fraction of max speed at which to drive
   */
  frc2::CommandPtr DriveDistanceCommand(units::meter_t distance, double speed);

  /**
   * Returns a command that turns to robot to the specified angle using a motion
   * profile and PID controller.
   *
   * @param angle The angle to turn to
   */
  frc2::CommandPtr TurnToAngleCommand(units::degree_t angle);

 private:
  frc::PWMSparkMax m_leftLeader{DriveConstants::kLeftMotor1Port};
  frc::PWMSparkMax m_leftFollower{DriveConstants::kLeftMotor2Port};
  frc::PWMSparkMax m_rightLeader{DriveConstants::kRightMotor1Port};
  frc::PWMSparkMax m_rightFollower{DriveConstants::kRightMotor2Port};

  frc::DifferentialDrive m_drive{
      [&](double output) { m_leftLeader.Set(output); },
      [&](double output) { m_rightLeader.Set(output); }};

  frc::Encoder m_leftEncoder{DriveConstants::kLeftEncoderPorts[0],
                             DriveConstants::kLeftEncoderPorts[1],
                             DriveConstants::kLeftEncoderReversed};
  frc::Encoder m_rightEncoder{DriveConstants::kRightEncoderPorts[0],
                              DriveConstants::kRightEncoderPorts[1],
                              DriveConstants::kRightEncoderReversed};

  frc::ADXRS450_Gyro m_gyro;

  frc::ProfiledPIDController<units::radians> m_controller{
      DriveConstants::kTurnP,
```

(continues on next page)

(continued from previous page)

```
      DriveConstants::kTurnI,
      DriveConstants::kTurnD,
      {DriveConstants::kMaxTurnRate, DriveConstants::kMaxTurnAcceleration}};
  frc::SimpleMotorFeedforward<units::radians> m_feedforward{
      DriveConstants::ks, DriveConstants::kv, DriveConstants::ka};
};
```

C++ (Source)

```
#include "subsystems/Drive.h"

#include <utility>

#include <frc/RobotController.h>
#include <frc2/command/Commands.h>

Drive::Drive() {
  wpi::SendableRegistry::AddChild(&m_drive, &m_leftLeader);
  wpi::SendableRegistry::AddChild(&m_drive, &m_rightLeader);

  m_leftLeader.AddFollower(m_leftFollower);
  m_rightLeader.AddFollower(m_rightFollower);

  // We need to invert one side of the drivetrain so that positive voltages
  // result in both sides moving forward. Depending on how your robot's
  // gearbox is constructed, you might have to invert the left side instead.
  m_rightLeader.SetInverted(true);

  // Sets the distance per pulse for the encoders
  m_leftEncoder.SetDistancePerPulse(DriveConstants::kEncoderDistancePerPulse);
  m_rightEncoder.SetDistancePerPulse(DriveConstants::kEncoderDistancePerPulse);

  // Set the controller to be continuous (because it is an angle controller)
  m_controller.EnableContinuousInput(-180_deg, 180_deg);
  // Set the controller tolerance - the delta tolerance ensures the robot is
  // stationary at the setpoint before it is considered as having reached the
  // reference
  m_controller.SetTolerance(DriveConstants::kTurnTolerance,
                            DriveConstants::kTurnRateTolerance);
}

frc2::CommandPtr Drive::ArcadeDriveCommand(std::function<double()> fwd,
                                           std::function<double()> rot) {
  return Run([this, fwd = std::move(fwd), rot = std::move(rot)] {
           m_drive.ArcadeDrive(fwd(), rot());
         })
      .WithName("ArcadeDrive");
}

frc2::CommandPtr Drive::DriveDistanceCommand(units::meter_t distance,
                                             double speed) {
  return RunOnce([this] {
           // Reset encoders at the start of the command
           m_leftEncoder.Reset();
```

(continues on next page)

(continued from previous page)

```
          m_rightEncoder.Reset();
        })
      // Drive forward at specified speed
      .AndThen(Run([this, speed] { m_drive.ArcadeDrive(speed, 0.0); }))
      .Until([this, distance] {
        return units::math::max(units::meter_t(m_leftEncoder.GetDistance()),
                                units::meter_t(m_rightEncoder.GetDistance())) >=
               distance;
      })
      // Stop the drive when the command ends
      .FinallyDo([this](bool interrupted) { m_drive.StopMotor(); });
}

frc2::CommandPtr Drive::TurnToAngleCommand(units::degree_t angle) {
  return StartRun(
             [this] { m_controller.Reset(m_gyro.GetRotation2d().Degrees()); },
             [this, angle] {
               m_drive.ArcadeDrive(
                   0, m_controller.Calculate(m_gyro.GetRotation2d().Degrees(),
                                             angle) +
                          // Divide feedforward voltage by battery voltage to
                          // normalize it to [-1, 1]
                          m_feedforward.Calculate(
                              m_controller.GetSetpoint().velocity) /
                              frc::RobotController::GetBatteryVoltage());
             })
      .Until([this] { return m_controller.AtGoal(); })
      .FinallyDo([this] { m_drive.ArcadeDrive(0, 0); });
}
```

`turnToAngleCommand` uses a ProfiledPIDController to smoothly turn the drivetrain. The `startRun` command factory is used to reset the `ProfiledPIDController` when the command is scheduled to avoid unwanted behavior, and to calculate PID and feedforward outputs to pass into the `arcadeDrive` method in order to drive the robot. The command is decorated using the `until` decorator to end the command when the ProfiledPIDController is finished with the profile. To ensure the drivetrain stops when the command ends, the `finallyDo` decorator is used to stop the drivetrain by setting the speed to zero.

27.13 Passing Functions As Parameters

In order to provide a concise inline syntax, the command-based library often accepts functions as parameters of constructors, factories, and decorators. Fortunately, both Java and C++ offer users the ability to *pass functions as objects*:

27.13.1 Method References (Java)

In Java, a reference to a function that can be passed as a parameter is called a method reference. The general syntax for a method reference is `object::method` or `Class::staticMethod`. Note that no method parameters are included, since the method *itself* is passed. 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 *Method References*.

27.13.2 Lambda Expressions (Java)

While method references work well for passing a function that has already been written, often it is inconvenient/wasteful to write a function solely for the purpose of sending as a method reference, if that function 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 function to be defined *inside of a parameter list*. For specifics on how to write Java lambda expressions, see *Lambda Expressions in Java*.

27.13.3 Lambda Expressions (C++)

Warning

Due to complications in C++ semantics, capturing `this` in a C++ lambda can cause a null pointer exception if done from a component command of a command composition. Whenever possible, C++ users should capture relevant command members explicitly and by value. For more details, see here.

C++ lacks a close equivalent to Java method references - pointers to member functions are generally not directly usable 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 *Lambda Expressions in C++*.

28 Kinematics and Odometry

28.1 Introduction to Kinematics and The ChassisSpeeds Class

Note

Kinematics and odometry uses a common coordinate system. You may wish to reference the *Coordinate System* section for details.

28.1.1 What is kinematics?

The kinematics suite contains classes for differential drive, swerve drive, and mecanum drive kinematics and odometry. The kinematics classes help convert between a universal `ChassisSpeeds` (Java, C++, Python)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.

28.1.2 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.

28.1.3 The ChassisSpeeds 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 `vy` component of zero because of its inability to move sideways.

28.1.4 Constructing a ChassisSpeeds object

The constructor for the `ChassisSpeeds` object is very straightforward, accepting three arguments for `vx`, `vy`, and `omega`. In Java and Python, `vx` and `vy` 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

```
// 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);
```

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.
frc::ChassisSpeeds speeds{3.0_mps, -2.0_mps,
  units::radians_per_second_t(std::numbers::pi)};
```

PYTHON

```python
import math
from wpimath.kinematics import ChassisSpeeds
# 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.
speeds = ChassisSpeeds(3.0, -2.0, math.pi)
```

28.1.5 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 / Python) / `ChassisSpeeds::FromFieldRelativeSpeeds` (C++) method can be used to generate the `ChassisSpeeds` object from field-relative speeds. This method accepts the `vx` (relative to the field), `vy` (relative to the field), `omega`, and the robot angle.

JAVA

```java
// 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));
```

C++

```cpp
// 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(std::numbers::pi / 2.0), Rotation2d(45_
↪deg));
```

PYTHON

```python
import math
from wpimath.kinematics import ChassisSpeeds
from wpimath.geometry  import Rotation2d
# 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.
speeds = ChassisSpeeds.fromFieldRelativeSpeeds(
  2.0, 2.0, math.pi / 2.0, Rotation2d.fromDegrees(45.0))
```

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.

28.2 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.

28.2.1 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 and Python, 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.

28.2.2 Converting Chassis Speeds to Wheel Speeds

The `toWheelSpeeds(ChassisSpeeds speeds)` (Java / Python) / `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 (`vx`) and an angular velocity (`omega`) to left and right wheel velocities.

JAVA

```java
// 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
double leftVelocity = wheelSpeeds.leftMetersPerSecond;
// Right velocity
double rightVelocity = wheelSpeeds.rightMetersPerSecond;
```

C++

```cpp
// 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);
```

PYTHON

```python
from wpimath.kinematics import DifferentialDriveKinematics
from wpimath.kinematics import ChassisSpeeds
from wpimath.units import inchesToMeters
# Creating my kinematics object: track width of 27 inches
kinematics = DifferentialDriveKinematics(Units.inchesToMeters(27.0))
# Example chassis speeds: 2 meters per second linear velocity,
# 1 radian per second angular velocity.
chassisSpeeds = ChassisSpeeds(2.0, 0, 1.0)
# Convert to wheel speeds
wheelSpeeds = kinematics.toWheelSpeeds(chassisSpeeds)
# Left velocity
leftVelocity = wheelSpeeds.left
# Right velocity
rightVelocity = wheelSpeeds.right
```

28.2.3 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 / Python) / `ToChassisSpeeds(DifferentialDriveWheelSpeeds speeds)` (C++) method should be used to achieve this.

JAVA

```
// 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;
```

C++

```
// 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.
// Note that because a differential drive is non-holonomic, the vy variable
// will be equal to zero.
auto [linearVelocity, vy, angularVelocity] = kinematics.ToChassisSpeeds(wheelSpeeds);
```

PYTHON

```
from wpimath.kinematics import DifferentialDriveKinematics
from wpimath.kinematics import DifferentialDriveWheelSpeeds
from wpimath.units import inchesToMeters
# Creating my kinematics object: track width of 27 inches
kinematics = DifferentialDriveKinematics(inchesToMeters(27.0))
# Example differential drive wheel speeds: 2 meters per second
# for the left side, 3 meters per second for the right side.
wheelSpeeds = DifferentialDriveWheelSpeeds(2.0, 3.0)
# Convert to chassis speeds.
chassisSpeeds = kinematics.toChassisSpeeds(wheelSpeeds)
# Linear velocity
linearVelocity = chassisSpeeds.vx
# Angular velocity
angularVelocity = chassisSpeeds.omega
```

28.3 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.

28.3.1 Creating the Odometry Object

The `DifferentialDriveOdometry` class constructor requires three mandatory arguments and one optional argument. The mandatory arguments are:

- The angle reported by your gyroscope (as a `Rotation2d`)
- The initial left and right encoder readings. In Java / Python, these are a number that represents the distance traveled by each side in meters. In C++, the *units library* must be used to represent your wheel positions.

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. The `Gyro` interface supplies `getRotation2d`/`GetRotation2d` that you can use for this purpose. See *Coordinate System* for more information about the coordinate system.

JAVA

```
// 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(
  m_gyro.getRotation2d(),
  m_leftEncoder.getDistance(), m_rightEncoder.getDistance(),
  new Pose2d(5.0, 13.5, new Rotation2d()));
```

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.
frc::DifferentialDriveOdometry m_odometry{
  m_gyro.GetRotation2d(),
  units::meter_t{m_leftEncoder.GetDistance()},
  units::meter_t{m_rightEncoder.GetDistance()},
  frc::Pose2d{5_m, 13.5_m, 0_rad}};
```

PYTHON

```
from wpimath.kinematics import DifferentialDriveOdometry
from wpimath.geometry import Pose2d
from wpimath.geometry import 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.
m_odometry = DifferentialDriveOdometry(
  m_gyro.getRotation2d(),
  m_leftEncoder.getDistance(), m_rightEncoder.getDistance(),
  Pose2d(5.0, 13.5, Rotation2d()))
```

28.3.2 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**
>
> If the robot is moving forward in a straight line, **both** distances (left and right) must be increasing positively – the rate of change must be positive.

JAVA

```
@Override
public void periodic() {
  // Get the rotation of the robot from the gyro.
  var gyroAngle = m_gyro.getRotation2d();
  // Update the pose
  m_pose = m_odometry.update(gyroAngle,
    m_leftEncoder.getDistance(),
    m_rightEncoder.getDistance());
}
```

C++

```
void Periodic() override {
  // Get the rotation of the robot from the gyro.
  frc::Rotation2d gyroAngle = m_gyro.GetRotation2d();
  // Update the pose
  m_pose = m_odometry.Update(gyroAngle,
    units::meter_t{m_leftEncoder.GetDistance()},
    units::meter_t{m_rightEncoder.GetDistance()});
}
```

PYTHON

```
def periodic(self):
  # Get the rotation of the robot from the gyro.
  gyroAngle = m_gyro.getRotation2d()
  # Update the pose
  m_pose = m_odometry.update(gyroAngle,
    m_leftEncoder.getDistance(),
    m_rightEncoder.getDistance())
```

28.3.3 Resetting the Robot Pose

The robot pose can be reset via the `resetPosition` method. This method accepts four arguments: the current gyro angle, the left and right wheel positions, and the new field-relative pose.

Important

If at any time, you decide to reset your gyroscope or encoders, the `resetPosition` method MUST be called with the new gyro angle and wheel distances.

Note

A full example of a differential drive robot with odometry is available here: C++ / Java / Python

In addition, the `GetPose` (C++) / `getPoseMeters` (Java / Python) methods can be used to retrieve the current robot pose without an update.

28.4 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.

> **Note**
>
> Swerve drive kinematics uses a common coordinate system. You may wish to reference the *Coordinate System* section for details.

28.4.1 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 / Python, 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.

28.4.2 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

```
// 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);
// Creating my kinematics object using the module locations
SwerveDriveKinematics m_kinematics = new SwerveDriveKinematics(
  m_frontLeftLocation, m_frontRightLocation, m_backLeftLocation, m_backRightLocation
);
```

C++

```
// 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};
```

PYTHON

```
# Python requires using the right class for the number of modules you have
from wpimath.geometry import Translation2d
from wpimath.kinematics import SwerveDrive4Kinematics
# Locations for the swerve drive modules relative to the robot center.
frontLeftLocation = Translation2d(0.381, 0.381)
frontRightLocation = Translation2d(0.381, -0.381)
backLeftLocation = Translation2d(-0.381, 0.381)
backRightLocation = Translation2d(-0.381, -0.381)
# Creating my kinematics object using the module locations
self.kinematics = SwerveDrive4Kinematics(
  frontLeftLocation, frontRightLocation, backLeftLocation, backRightLocation
)
```

28.4.3 Converting chassis speeds to module states

The `toSwerveModuleStates(ChassisSpeeds speeds)` (Java / Python) / `ToSwerveModuleStates(ChassisSpeeds speeds)` (C++) method should be used to convert a `ChassisSpeeds` object to a 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

```
// 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];
```

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.
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);
```

PYTHON

```
from wpimath.kinematics import ChassisSpeeds
# Example chassis speeds: 1 meter per second forward, 3 meters
# per second to the left, and rotation at 1.5 radians per second
# counterclockwise.
speeds = ChassisSpeeds(1.0, 3.0, 1.5)
# Convert to module states
frontLeft, frontRight, backLeft, backRight = self.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

```
var frontLeftOptimized = SwerveModuleState.optimize(frontLeft,
   new Rotation2d(m_turningEncoder.getDistance()));
```

C++

```
auto flOptimized = frc::SwerveModuleState::Optimize(fl,
   units::radian_t(m_turningEncoder.GetDistance()));
```

PYTHON

```
from wpimath.kinematics import SwerveModuleState
from wpimath.geometry import Rotation2d
frontLeftOptimized = SwerveModuleState.optimize(frontLeft,
   Rotation2d(self.m_turningEncoder.getDistance()))
```

Cosine compensation

Cosine compensation is a technique that reduces the speed of a module when it is not pointing in the desired direction. This is done by multiplying the desired speed of the module by the cosine of the angle error.

- If the wheel is pointing straight in the desired direction, then the speed remains unchanged as $\cos(0°) = 1$.
- If the wheel is perpendicular to the desired direction of motion, then the speed is reduced to 0 as $\cos(90°) = 0$.
- Everything in between follows the cosine curve.

Cosine compensation has been shown to reduce the amount of "skew" a swerve drive experiences when changing direction.

JAVA

```
var currentAngle = new Rotation2d.fromRadians(m_turningEncoder.getDistance());
var frontLeftOptimized = SwerveModuleState.optimize(frontLeft, currentAngle);
frontLeftOptimized.speedMetersPerSecond *= frontLeftOptimized.angle.
↪minus(currentAngle).getCos();
```

C++

```
Rotation2d currentAngle(m_turningEncoder.GetDistance());
auto flOptimized = frc::SwerveModuleState::Optimize(fl, currentAngle);
flOptimized.speed *= (flOptimized.angle - currentAngle).Cos();
```

PYTHON

```
from wpimath.kinematics import SwerveModuleState
from wpimath.geometry import Rotation2d
currentAngle = Rotation2d(self.m_turningEncoder.getDistance())
frontLeftOptimized = SwerveModuleState.optimize(frontLeft, currentAngle)
frontLeftOptimized.speed *= (frontLeftOptimized.angle - currentAngle).cos()
```

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

```
// 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);
```

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.
frc::ChassisSpeeds speeds = frc::ChassisSpeeds::FromFieldRelativeSpeeds(
  2_mps, 2_mps, units::radians_per_second_t(std::numbers::pi / 2.0), Rotation2d(45_
↪deg));
// Now use this in our kinematics
auto [fl, fr, bl, br] = kinematics.ToSwerveModuleStates(speeds);
```

PYTHON

```
from wpimath.kinematics import ChassisSpeeds
import math
from wpimath.geometry import Rotation2d
# 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.
speeds = ChassisSpeeds.fromFieldRelativeSpeeds(
  2.0, 2.0, math.pi / 2.0, Rotation2d.fromDegrees(45.0))
# Now use this in our kinematics
self.moduleStates = self.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.

28.4.4 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 / Python) / `ToChassisSpeeds(SwerveModuleState... states)` (C++) method can be used to achieve this.

JAVA

```
// 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;
```

C++

```
// 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);
```

PYTHON

```python
from wpimath.kinematics import SwerveModuleState
from wpimath.geometry import Rotation2d
# Example module states
frontLeftState = SwerveModuleState(23.43, Rotation2d.fromDegrees(-140.19))
frontRightState = SwerveModuleState(23.43, Rotation2d.fromDegrees(-39.81))
backLeftState = SwerveModuleState(54.08, Rotation2d.fromDegrees(-109.44))
backRightState = SwerveModuleState(54.08, Rotation2d.fromDegrees(-70.56))
# Convert to chassis speeds
chassisSpeeds = self.kinematics.toChassisSpeeds(
  frontLeftState, frontRightState, backLeftState, backRightState)
# Getting individual speeds
forward = chassisSpeeds.vx
sideways = chassisSpeeds.vy
angular = chassisSpeeds.omega
```

28.4.5 Module state visualization with AdvantageScope

By recording a set of swerve module states using *NetworkTables* or *WPILib data logs*, *AdvantageScope* can be used to visualize the state of a swerve drive. The code below shows how a set of `SwerveModuleState` objects can be published to NetworkTables.

JAVA

```java
public class Example {
  private final StructArrayPublisher<SwerveModuleState> publisher;
  public Example() {
    // Start publishing an array of module states with the "/SwerveStates" key
    publisher = NetworkTableInstance.getDefault()
      .getStructArrayTopic("/SwerveStates", SwerveModuleState.struct).publish();
  }
  public void periodic() {
    // Periodically send a set of module states
    publisher.set(new SwerveModuleState[] {
      frontLeftState,
      frontRightState,
      backLeftState,
      backRightState
    });
  }
}
```

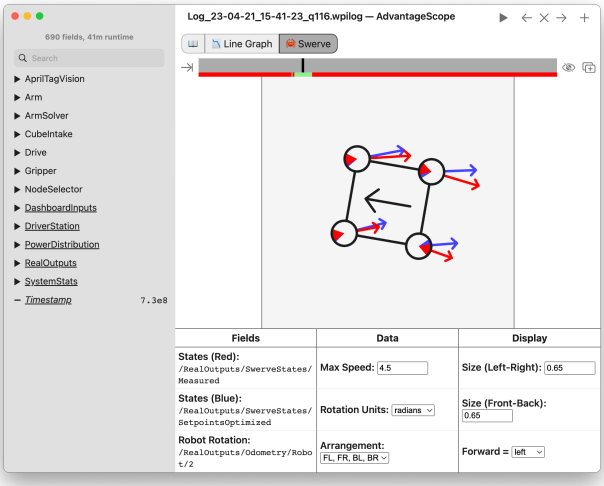
C++

```
class Example {
  nt::StructArrayPublisher<frc::SwerveModuleState> publisher
 public:
  Example() {
    // Start publishing an array of module states with the "/SwerveStates" key
    publisher = nt::NetworkTableInstance::GetDefault()
      .GetStructArrayTopic<frc::SwerveModuleState>("/SwerveStates").Publish();
  }
  void Periodic() {
    // Periodically send a set of module states
    swervePublisher.Set(
      std::vector{
        frontLeftState,
        frontRightState,
        backLeftState,
        backRightState
      }
    );
  }
};
```

PYTHON

```
import ntcore
from wpimath.kinematics import SwerveModuleState
# get the default instance of NetworkTables
nt = ntcore.NetworkTableInstance.getDefault()
# Start publishing an array of module states with the "/SwerveStates" key
topic = nt.getStructArrayTopic("/SwerveStates", SwerveModuleState)
self.pub = topic.publish()
def periodic(self):
  # Periodically send a set of module states
  self.pub.set([frontLeftState,frontRightState,backLeftState,backRightState])
```

See the documentation for the swerve tab for more details on visualizing this data using AdvantageScope.



28.5 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.

28.5.1 Creating the odometry object

The `SwerveDriveOdometry<int NumModules>` class constructor requires one template argument (only C++), three 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 (as a `SwerveDriveKinematics` instance)
- The angle reported by your gyroscope (as a `Rotation2d`)
- The initial positions of the swerve modules (as an array of `SwerveModulePosition`). In Java, this must be constructed with each wheel position in meters. In C++, the *units library* must be used to represent your wheel positions. It is important that the order in which you pass the `SwerveModulePosition` objects is the same as the order in which you created the kinematics object.

The fourth 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. The `Gyro` interface supplies `getRotation2d`/`GetRotation2d` that you can use for this purpose. See *Coordinate System* for more information about the coordinate system.

JAVA

```
// 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);
// 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 and the initial wheel␣
↪positions.
// 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 the opposing alliance wall.
SwerveDriveOdometry m_odometry = new SwerveDriveOdometry(
  m_kinematics, m_gyro.getRotation2d(),
  new SwerveModulePosition[] {
    m_frontLeftModule.getPosition(),
    m_frontRightModule.getPosition(),
    m_backLeftModule.getPosition(),
    m_backRightModule.getPosition()
  }, new Pose2d(5.0, 13.5, new Rotation2d()));
```

C++

```
// 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, m_gyro.GetRotation2d(),
  {m_frontLeft.GetPosition(), m_frontRight.GetPosition(),
  m_backLeft.GetPosition(), m_backRight.GetPosition()},
  frc::Pose2d{5_m, 13.5_m, 0_rad}};
```

PYTHON

```
# Python requires using the right class for the number of modules you have
# For both the Kinematics and Odometry classes
from wpimath.geometry import Translation2d
from wpimath.kinematics import SwerveDrive4Kinematics
from wpimath.kinematics import SwerveDrive4Odometry
from wpimath.geometry import Pose2d
from wpimath.geometry import Rotation2d
class MyRobot:
  def robotInit(self):
    # Locations for the swerve drive modules relative to the robot center.
    frontLeftLocation = Translation2d(0.381, 0.381)
    frontRightLocation = Translation2d(0.381, -0.381)
    backLeftLocation = Translation2d(-0.381, 0.381)
    backRightLocation = Translation2d(-0.381, -0.381)
    # Creating my kinematics object using the module locations
    self.kinematics = SwerveDrive4Kinematics(
      frontLeftLocation, frontRightLocation, backLeftLocation, backRightLocation
    )
    # Creating my odometry object from the kinematics object and the initial wheel␣
↪positions.
    # 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 the opposing alliance wall.
    self.odometry = SwerveDrive4Odometry(
      self.kinematics, self.gyro.getRotation2d(),
      (
        self.frontLeftModule.getPosition(),
        self.frontRightModule.getPosition(),
        self.backLeftModule.getPosition(),
        self.backRightModule.getPosition()
      ),
      Pose2d(5.0, 13.5, Rotation2d()))
```

28.5.2 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 an array of `SwerveModulePosition` objects. It is important that the order in which you pass the `SwerveModulePosition` 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

```
@Override
public void periodic() {
  // Get the rotation of the robot from the gyro.
  var gyroAngle = m_gyro.getRotation2d();
  // Update the pose
  m_pose = m_odometry.update(gyroAngle,
    new SwerveModulePosition[] {
```

(continues on next page)

(continued from previous page)

```
      m_frontLeftModule.getPosition(), m_frontRightModule.getPosition(),
      m_backLeftModule.getPosition(), m_backRightModule.getPosition()
    });
}
```

C++

```
void Periodic() override {
  // Get the rotation of the robot from the gyro.
  frc::Rotation2d gyroAngle = m_gyro.GetRotation2d();
  // Update the pose
  m_pose = m_odometry.Update(gyroAngle,
    {
      m_frontLeftModule.GetPosition(), m_frontRightModule.GetPosition(),
      m_backLeftModule.GetPosition(), m_backRightModule.GetPosition()
    };
  )
}
```

PYTHON

```
def periodic(self):
  # Get the rotation of the robot from the gyro.
  self.gyroAngle = self.gyro.getRotation2d()
  # Update the pose
  self.pose = self.odometry.update(self.gyroAngle,
      self.frontLeftModule.getPosition(), self.frontRightModule.getPosition(),
      self.backLeftModule.getPosition(), self.backRightModule.getPosition()
  )
```

28.5.3 Resetting the Robot Pose

The robot pose can be reset via the `resetPosition` method. This method accepts three arguments: the current gyro angle, an array of the current module positions (as in the constructor and update method), and the new field-relative pose.

Important

If at any time, you decide to reset your gyroscope or wheel encoders, the `resetPosition` method MUST be called with the new gyro angle and wheel encoder positions.

Note

The implementation of `getPosition() / GetPosition()` above is left to the user. The idea is to get the module position (distance and angle) from each module. For a full example, see here: C++ / Java / Python

In addition, the `GetPose` (C++) / `getPoseMeters` (Java / Python) methods can be used to retrieve the current robot pose without an update.

28.6 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.

> **Note**
>
> Mecanum kinematics uses a common coordinate system. You may wish to reference the *Coordinate System* section for details.

28.6.1 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

```
// 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
);
```

C++

```
// 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};
```

PYTHON

```
from wpimath.geometry import Translation2d
from wpimath.kinematics import MecanumDriveKinematics
# Locations of the wheels relative to the robot center.
frontLeftLocation = Translation2d(0.381, 0.381)
frontRightLocation = Translation2d(0.381, -0.381)
backLeftLocation = Translation2d(-0.381, 0.381)
backRightLocation = Translation2d(-0.381, -0.381)
# Creating my kinematics object using the wheel locations.
self.kinematics = MecanumDriveKinematics(
  frontLeftLocation, frontRightLocation, backLeftLocation, backRightLocation
)
```

28.6.2 Converting Chassis Speeds to Wheel Speeds

The `toWheelSpeeds(ChassisSpeeds speeds)` (Java / Python) / `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

```
// 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
```

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.
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
// struct into it's individual components
auto [fl, fr, bl, br] = kinematics.ToWheelSpeeds(speeds);
```

PYTHON

```
from wpimath.kinematics import ChassisSpeeds
# Example chassis speeds: 1 meter per second forward, 3 meters
# per second to the left, and rotation at 1.5 radians per second
# counterclockwise.
speeds = ChassisSpeeds(1.0, 3.0, 1.5)
# Convert to wheel speeds
frontLeft, frontRight, backLeft, backRight = self.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

```
// 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);
```

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.
frc::ChassisSpeeds speeds = frc::ChassisSpeeds::FromFieldRelativeSpeeds(
  2_mps, 2_mps, units::radians_per_second_t(std::numbers::pi / 2.0), Rotation2d(45_
↪deg));
// Now use this in our kinematics
auto [fl, fr, bl, br] = kinematics.ToWheelSpeeds(speeds);
```

PYTHON

```
from wpimath.kinematics import ChassisSpeeds
import math
from wpimath.geometry import Rotation2d
# 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
```

(continues on next page)

(continued from previous page)

```
# robot angle is 45 degrees.
speeds = ChassisSpeeds.fromFieldRelativeSpeeds(
  2.0, 2.0, math.pi / 2.0, Rotation2d.fromDegrees(45.0))
# Now use this in our kinematics
wheelSpeeds = self.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 `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 wheel 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 wheel.

28.6.3 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 / Python) / `ToChassisSpeeds(MecanumDriveWheelSpeeds speeds)` (C++) method can be used to achieve this.

JAVA

```
// 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;
```

C++

```
// 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);
```

PYTHON

```
from wpimath.kinematics import MecanumDriveWheelSpeeds
# Example wheel speeds
wheelSpeeds = MecanumDriveWheelSpeeds(-17.67, 20.51, -13.44, 16.26)
# Convert to chassis speeds
chassisSpeeds = self.kinematics.toChassisSpeeds(wheelSpeeds)
# Getting individual speeds
forward = chassisSpeeds.vx
sideways = chassisSpeeds.vy
angular = chassisSpeeds.omega
```

28.7 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.

28.7.1 Creating the odometry object

The `MecanumDriveOdometry` class constructor requires three mandatory arguments and one optional argument.

The mandatory arguments are:

- The kinematics object that represents your mecanum drive (as a `MecanumDriveKinematics` instance)
- The angle reported by your gyroscope (as a `Rotation2d`)
- The initial positions of the wheels (as `MecanumDriveWheelPositions`). In Java / Python, this must be constructed with each wheel position in meters. In C++, the *units library* must be used to represent your wheel positions.

The fourth 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. The `Gyro` interface supplies `getRotation2d`/`GetRotation2d` that you can use for this purpose. See *Coordinate System* for more information about the coordinate system.

JAVA

```
// 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 and the initial wheel␣
↪positions.
// 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 the opposing alliance wall.
MecanumDriveOdometry m_odometry = new MecanumDriveOdometry(
  m_kinematics,
  m_gyro.getRotation2d(),
  new MecanumDriveWheelPositions(
    m_frontLeftEncoder.getDistance(), m_frontRightEncoder.getDistance(),
    m_backLeftEncoder.getDistance(), m_backRightEncoder.getDistance()
  ),
  new Pose2d(5.0, 13.5, new Rotation2d())
);
```

C++

```
// 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{
```

(continues on next page)

(continued from previous page)

```
  m_kinematics,
  m_gyro.GetRotation2d(),
  frc::MecanumDriveWheelPositions{
    units::meter_t{m_frontLeftEncoder.GetDistance()},
    units::meter_t{m_frontRightEncoder.GetDistance()},
    units::meter_t{m_backLeftEncoder.GetDistance()},
    units::meter_t{m_backRightEncoder.GetDistance()}
  },
  frc::Pose2d{5_m, 13.5_m, 0_rad}};
```

PYTHON

```
from wpimath.geometry import Translation2d
from wpimath.kinematics import MecanumDriveKinematics
from wpimath.kinematics import MecanumDriveOdometry
from wpimath.kinematics import MecanumDriveWheelPositions
from wpimath.geometry import Pose2d
from wpimath.geometry import Rotation2d
# Locations of the wheels relative to the robot center.
frontLeftLocation = Translation2d(0.381, 0.381)
frontRightLocation = Translation2d(0.381, -0.381)
backLeftLocation = Translation2d(-0.381, 0.381)
backRightLocation = Translation2d(-0.381, -0.381)
# Creating my kinematics object using the wheel locations.
self.kinematics = MecanumDriveKinematics(
  frontLeftLocation, frontRightLocation, backLeftLocation, backRightLocation
)
# Creating my odometry object from the kinematics object and the initial wheel␣
↪positions.
# 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 the opposing alliance wall.
self.odometry = MecanumDriveOdometry(
  self.kinematics,
  self.gyro.getRotation2d(),
  MecanumDriveWheelPositions(
    self.frontLeftEncoder.getDistance(), self.frontRightEncoder.getDistance(),
    self.backLeftEncoder.getDistance(), self.backRightEncoder.getDistance()
  ),
  Pose2d(5.0, 13.5, Rotation2d())
)
```

28.7.2 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 `MecanumDriveWheelPositions` object representing the position 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.

JAVA

```
@Override
public void periodic() {
  // Get my wheel positions
  var wheelPositions = new MecanumDriveWheelPositions(
    m_frontLeftEncoder.getDistance(), m_frontRightEncoder.getDistance(),
    m_backLeftEncoder.getDistance(), m_backRightEncoder.getDistance());
  // Get the rotation of the robot from the gyro.
  var gyroAngle = m_gyro.getRotation2d();
  // Update the pose
  m_pose = m_odometry.update(gyroAngle, wheelPositions);
}
```

C++

```
void Periodic() override {
  // Get my wheel positions
  frc::MecanumDriveWheelPositions wheelPositions{
    units::meter_t{m_frontLeftEncoder.GetDistance()},
    units::meter_t{m_frontRightEncoder.GetDistance()},
    units::meter_t{m_backLeftEncoder.GetDistance()},
    units::meter_t{m_backRightEncoder.GetDistance()}};
  // Get the rotation of the robot from the gyro.
  frc::Rotation2d gyroAngle = m_gyro.GetRotation2d();
  // Update the pose
  m_pose = m_odometry.Update(gyroAngle, wheelPositions);
}
```

PYTHON

```
from wpimath.kinematics import MecanumDriveWheelPositions
   def periodic(self):
  # Get my wheel positions
  wheelPositions = MecanumDriveWheelPositions(
    self.frontLeftEncoder.getDistance(), self.frontRightEncoder.getDistance(),
    self.backLeftEncoder.getDistance(), self.backRightEncoder.getDistance())
  # Get the rotation of the robot from the gyro.
  gyroAngle = gyro.getRotation2d()
  # Update the pose
  self.pose = odometry.update(gyroAngle, wheelPositions)
```

28.7.3 Resetting the Robot Pose

The robot pose can be reset via the `resetPosition` method. This method accepts three arguments: the current gyro angle, the current wheel positions, and the new field-relative pose.

Important

If at any time, you decide to reset your gyroscope or encoders, the `resetPosition` method MUST be called with the new gyro angle and wheel positions.

Note

A full example of a mecanum drive robot with odometry is available here: C++ / Java / Python

In addition, the `GetPose` (C++) / `getPoseMeters` (Java / Python) methods can be used to retrieve the current robot pose without an update.

29 NetworkTables

This section outlines the details of using the NetworkTables (v4) API to communicate information across the robot network.

Important

The code examples in this section are not intended for the user to copy-paste. Ensure that the following documentation is thoroughly read and the API (Java, C++, Python) is consulted when necessary.

29.1 What is NetworkTables

NetworkTables is an implementation of a publish-subscribe messaging system. Values are published to named "topics" either on the robot, driver station, or potentially an attached coprocessor, and the values are automatically distributed to all subscribers to the topic. 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 topics called "X", "Y", and "Distance" they can be read by the robot shortly after being written. Then the robot can act upon them. Similarly, the robot program can write sensor values to topics and those can be read and plotted in real time on a dashboard application.

NetworkTables can be used by programs on the robot in Java, C++, or LabVIEW, and is built into each version of WPILib.

Note

NetworkTables has changed substantially in 2023. For more information on migrating pre-2023 code to use the new features, see *Migrating from NetworkTables 3.0 to NetworkTables 4.0*.

29.1.1 NetworkTables Concepts

First, let's define some terms:

- **Topic**: a named data channel. Topics have a fixed data type (for the lifetime of the topic) and *mutable* properties.
- **Publisher**: defines the topic and creates and sends timestamped data values.
- **Subscriber**: receives timestamped data value updates to one or more topics.
- **Entry**: a combined publisher and subscriber. The subscriber is always active, but the publisher is not created until a publish operation is performed (e.g. a value is "set", aka published, on the entry). This may be more convenient than maintaining a separate publisher and subscriber.
- **Property**: named information (metadata) about a topic stored and updated separately from the topic's data. A topic may have any number of properties. A property's value can be any data type that can be represented in JSON.

NetworkTables supports a range of data types, including *boolean*, numeric, string, and arrays of those types. Supported numeric data types are single or double precision *floating point*, or 64-bit integer. There is also the option of storing raw data (an array of bytes), which can be used for representing binary encoded structured data. Types are represented as strings for efficiency reasons. There is also an *enumeration* for the most common types in the NetworkTables API.

Topics are created when the first publisher announces the topic and are removed when the last publisher stops publishing. It's possible to subscribe to a topic that has not yet been created/published.

Topics have properties. Properties are initially set by the first publisher, but may be changed at any time. Similarly to values, property changes to a topic are propagated to all subscribers to that topic. Properties are structured data (JSON), but at the top level are simply a key/value store (a JSON map). Some properties have defined behavior, but arbitrary ones can be set by the application.

Publishers specify the topic's data type; while there can be multiple publishers to a single topic, they must all be publishing the same data type. This is enforced by the NetworkTables server (the first publisher "wins"). Typically single-topic subscribers also specify what data type they're expecting to receive on a topic and thus won't receive value updates of other types.

The Network Tables Protocol Specification contains detailed documentation on the current wire protocol.

29.1.2 Retained and Persistent Topics

While by default topics are *transitory* and disappear after the last publisher stops publishing, topics can be marked as *retained* (via setting the "retained" property to true) to prevent them from disappearing. For retained topics, the server acts as an implicit publisher of the last value, and will keep doing so as long as the server is running. This is primarily useful for configuration values; e.g. an autonomous mode selection published by a dashboard should set the topic as retained so its value is preserved in case the dashboard disconnects.

Additionally, topics can be marked as *persistent* via setting the "persistent" property to true. These operate similarly to retained topics, but in addition, persistent topic values are auto-

matically saved to a file on the server and when the server starts up again, the topic is created and its last value is published by the server.

29.1.3 Value Propagation

The server keeps a copy of the last published value for every topic. When a subscriber initially subscribes to a topic, the server sends the last published value. After that initial value, new value updates are communicated to subscribers each time the publisher sends a new value.

NetworkTables is a client/server system; clients do not talk directly to each other, but rather communicate via the server. Typically, the robot program is the server, and other pieces of software on other computers (e.g. the driver station or a coprocessor) are clients that connect to it. Thus, when a coprocessor (client) publishes a value, the value is sent first from the coprocessor (client) to the robot program (server), and then the robot program distributes that value to any subscribers (e.g. the robot program local program, or other clients such as dashboards).

The server does not send topic changes or value updates to clients that have not subscribed to the topic.

By default, NetworkTables sends value updates periodically, batching the data to help limit the number of small packets being sent over the network. Also, by default, only the most recent value is transmitted; any intermediate value changes made between network transmissions are discarded. This behavior can be changed via publish/subscribe options–publishers and subscribers can indicate that all value updates should be preserved and communicated via the "send all" option. In addition, it is possible to force NetworkTables to "flush" all current updates to the network; this is useful for minimizing latency.

29.1.4 Timestamps

All NetworkTable value updates are timestamped at the time they are published. Timestamps in NetworkTables are measured in integer microseconds.

NetworkTables automatically synchronizes time between the server and clients. Each client maintains an offset between the client local time and the server time, so when a client publishes a value, it stores a timestamp in local time and calculates the equivalent server timestamp. The server timestamp is what is communicated over the network to any subscribers. This makes it possible e.g. for a robot program to get a reasonable estimation of the time when a value was published on a coprocessor relative to the current time.

Because of this, two timestamps are visible through the API: a server timestamp indicating the time (estimated) on the server, and a local timestamp indicating the time on the client. When the RoboRIO is the NetworkTables server, the server timestamp is the same as the FPGA timestamp returned by `Timer.getFPGATimestamp()` (except the units are different: NetworkTables uses microseconds, while `getFPGATimestamp()` returns seconds).

29.1.5 NetworkTables Organization

Data is organized in NetworkTables in a hierarchy much like a filesystem's folders and files. There can be multiple subtables (folders) and topics (files) that may be nested in whatever way fits the data organization desired. At the top level (`NetworkTableInstance`: Java, C++, Python), topic names are handled similar to absolute paths in a filesystem: subtables are represented as a long topic name with slashes ("/") separating the nested subtable and value names. A `NetworkTable` (Java, C++, Python) object represents a single subtable (folder), so topic names are relative to the NetworkTable's base path: e.g. for a root table called "SmartDashboard" with a topic named "xValue", the same topic can be accessed via `NetworkTableInstance` as a topic named "/SmartDashboard/xValue". However, unlike a filesystem, subtables don't really exist in the same way folders do, as there is no way to represent an empty subtable on the network–a subtable "appears" only as long as there are topics published within it.

OutlineViewer is a utility for exploring the values stored in NetworkTables, and can show either a flat view (topics with absolute paths) or a nested view (subtables and topics).

There are some default tables that are created automatically when a robot program starts up:

Table name	Use
/SmartDashboard	Used to store values written to the SmartDashboard or Shuffleboard using the `SmartDashboard.put()` set of methods.
/LiveWindow	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

29.1.6 NetworkTables API Variants

There are two major variants of the NetworkTables API. The object-oriented API (C++ and Java) is recommended for robot code and general team use, and provides classes that help ensure correct use of the API. For advanced use cases such as writing object-oriented wrappers for other programming languages, there's also a C/C++ handle-based API.

29.1.7 Lifetime Management

Publishers, subscribers, and entries only exist as long as the objects exist.

In Java, a common bug is to create a subscriber or publisher and not properly release it by calling `close()`, as this will result in the object lingering around for an unknown period of time and not releasing resources properly. This is less common of an issue in robot programs, as long as the publisher or subscriber object is stored in an instance variable that persists for the life of the program.

In C++, publishers, subscribers, and entries are *RAII*, which means they are automatically destroyed when they go out of scope. `NetworkTableInstance` is an exception to this; it is designed to be explicitly destroyed, so it's not necessary to maintain a global instance of it.

Python is similar to Java, except that subscribers or publishers are released when they are garbage collected.

29.2 NetworkTables Tables and Topics

29.2.1 Using the NetworkTable Class

The `NetworkTable` (Java, C++, Python) class is an API abstraction that represents a single "folder" (or "table") of topics as described in *NetworkTables Organization*. The NetworkTable class stores the base path to the table and provides functions to get topics within the table, automatically prepending the table path.

29.2.2 Getting a Topic

A `Topic` (Java, C++, Python) object (or `NT_Topic` handle) represents a *topic*. This has a 1:1 correspondence with the topic's name, and will not change as long as the instance exists. Unlike publishers and subscribers, it is not necessary to store a Topic object.

Having a Topic object or handle does not mean the topic exists or is of the correct type. For convenience when creating publishers and subscribers, there are type-specific Topic classes (e.g. `BooleanTopic`: Java, C++, Python), but there is no check at the Topic level to ensure that the topic's type actually matches. The preferred method to get a type-specific topic to call the appropriate type-specific getter, but it's also possible to directly convert a generic Topic into a type-specific Topic class. Note: the handle-based API does not have a concept of type-specific classes.

Java

```
NetworkTableInstance inst = NetworkTableInstance.getDefault();
NetworkTable table = inst.getTable("datatable");
// get a topic from a NetworkTableInstance
// the topic name in this case is the full name
DoubleTopic dblTopic = inst.getDoubleTopic("/datatable/X");
// get a topic from a NetworkTable
// the topic name in this case is the name within the table;
// this line and the one above reference the same topic
DoubleTopic dblTopic = table.getDoubleTopic("X");
// get a type-specific topic from a generic Topic
Topic genericTopic = inst.getTopic("/datatable/X");
DoubleTopic dblTopic = new DoubleTopic(genericTopic);
```

C++

```
nt::NetworkTableInstance inst = nt::NetworkTableInstance::GetDefault();
std::shared_ptr<nt::NetworkTable> table = inst.GetTable("datatable");
// get a topic from a NetworkTableInstance
// the topic name in this case is the full name
nt::DoubleTopic dblTopic = inst.GetDoubleTopic("/datatable/X");
// get a topic from a NetworkTable
// the topic name in this case is the name within the table;
// this line and the one above reference the same topic
nt::DoubleTopic dblTopic = table->GetDoubleTopic("X");
// get a type-specific topic from a generic Topic
nt::Topic genericTopic = inst.GetTopic("/datatable/X");
nt::DoubleTopic dblTopic{genericTopic};
```

C++ (Handle-based)

```
NT_Instance inst = nt::GetDefaultInstance();
// get a topic from a NetworkTableInstance
NT_Topic topic = nt::GetTopic(inst, "/datatable/X");
```

C

```
NT_Instance inst = NT_GetDefaultInstance();
// get a topic from a NetworkTableInstance
NT_Topic topic = NT_GetTopic(inst, "/datatable/X");
```

Python

```
import ntcore
inst = ntcore.NetworkTableInstance.getDefault()
table = inst.getTable("datatable")
# get a topic from a NetworkTableInstance
# the topic name in this case is the full name
dblTopic = inst.getDoubleTopic("/datatable/X")
# get a topic from a NetworkTable
# the topic name in this case is the name within the table;
# this line and the one above reference the same topic
dblTopic = table.getDoubleTopic("X")
# get a type-specific topic from a generic Topic
genericTopic = inst.getTopic("/datatable/X")
dblTopic = ntcore.DoubleTopic(genericTopic)
```

29.3 Publishing and Subscribing to a Topic

29.3.1 Publishing to a Topic

In order to create a *topic* and publish values to it, it's necessary to create a *publisher*.

NetworkTable publishers are represented as type-specific Publisher objects (e.g. `BooleanPublisher`: Java, C++, Python). Publishers are only active as long as the Publisher object exists. Typically you want to keep publishing longer than the local scope of a function, so it's necessary to store the Publisher object somewhere longer term, e.g. in an instance variable. In Java, the `close()` method needs be called to stop publishing; in C++ this is handled by the destructor. C++ publishers are moveable and non-copyable. In Python the `close()` method should be called to stop publishing, but it will also be closed when the object is garbage collected.

In the handle-based APIs, there is only the non-type-specific `NT_Publisher` handle; the user is responsible for keeping track of the type of the publisher and using the correct type-specific set methods.

Publishing values is done via a `set()` operation. By default, this operation uses the current time, but a timestamp may optionally be specified. Specifying a timestamp can be useful when multiple values should have the same update timestamp. The timestamp units are integer microseconds (see example code for how to get a current timestamp that is consistent with the library).

Java

```
public class Example {
  // the publisher is an instance variable so its lifetime matches that of the class
  final DoublePublisher dblPub;
  public Example(DoubleTopic dblTopic) {
    // start publishing; the return value must be retained (in this case, via
    // an instance variable)
    dblPub = dblTopic.publish();
    // publish options may be specified using PubSubOption
    dblPub = dblTopic.publish(PubSubOption.keepDuplicates(true));
    // publishEx provides additional options such as setting initial
    // properties and using a custom type string. Using a custom type string for
    // types other than raw and string is not recommended. The properties string
    // must be a JSON map.
    dblPub = dblTopic.publishEx("double", "{\"myprop\": 5}");
  }
  public void periodic() {
    // publish a default value
    dblPub.setDefault(0.0);
    // publish a value with current timestamp
    dblPub.set(1.0);
    dblPub.set(2.0, 0);  // 0 = use current time
    // publish a value with a specific timestamp; NetworkTablesJNI.now() can
    // be used to get the current time. On the roboRIO, this is the same as
    // the FPGA timestamp (e.g. RobotController.getFPGATime())
    long time = NetworkTablesJNI.now();
    dblPub.set(3.0, time);
    // publishers also implement the appropriate Consumer functional interface;
```

(continues on next page)

(continued from previous page)

```
    // this example assumes void myFunc(DoubleConsumer func) exists
    myFunc(dblPub);
  }
  // often not required in robot code, unless this class doesn't exist for
  // the lifetime of the entire robot program, in which case close() needs to be
  // called to stop publishing
  public void close() {
    // stop publishing
    dblPub.close();
  }
}
```

C++

```
class Example {
  // the publisher is an instance variable so its lifetime matches that of the class
  // publishing is automatically stopped when dblPub is destroyed by the class␣
↪destructor
  nt::DoublePublisher dblPub;
 public:
  explicit Example(nt::DoubleTopic dblTopic) {
    // start publishing; the return value must be retained (in this case, via
    // an instance variable)
    dblPub = dblTopic.Publish();
    // publish options may be specified using PubSubOptions
    dblPub = dblTopic.Publish({.keepDuplicates = true});
    // PublishEx provides additional options such as setting initial
    // properties and using a custom type string. Using a custom type string for
    // types other than raw and string is not recommended. The properties must
    // be a JSON map.
    dblPub = dblTopic.PublishEx("double", {{"myprop", 5}});
  }
  void Periodic() {
    // publish a default value
    dblPub.SetDefault(0.0);
    // publish a value with current timestamp
    dblPub.Set(1.0);
    dblPub.Set(2.0, 0);  // 0 = use current time
    // publish a value with a specific timestamp; nt::Now() can
    // be used to get the current time.
    int64_t time = nt::Now();
    dblPub.Set(3.0, time);
  }
};
```

C++ (Handle-based)

```
class Example {
  // the publisher is an instance variable, but since it's a handle, it's
  // not automatically released, so we need a destructor
  NT_Publisher dblPub;
 public:
  explicit Example(NT_Topic dblTopic) {
    // start publishing. It's recommended that the type string be standard
    // for all types except string and raw.
    dblPub = nt::Publish(dblTopic, NT_DOUBLE, "double");
    // publish options may be specified using PubSubOptions
    dblPub = nt::Publish(dblTopic, NT_DOUBLE, "double",
        {.keepDuplicates = true});
    // PublishEx allows setting initial properties. The
    // properties must be a JSON map.
    dblPub = nt::PublishEx(dblTopic, NT_DOUBLE, "double", {{"myprop", 5}});
  }
  void Periodic() {
    // publish a default value
    nt::SetDefaultDouble(dblPub, 0.0);
    // publish a value with current timestamp
    nt::SetDouble(dblPub, 1.0);
    nt::SetDouble(dblPub, 2.0, 0);  // 0 = use current time
    // publish a value with a specific timestamp; nt::Now() can
    // be used to get the current time.
    int64_t time = nt::Now();
    nt::SetDouble(dblPub, 3.0, time);
  }
  ~Example() {
    // stop publishing
    nt::Unpublish(dblPub);
  }
};
```

C

```
// This code assumes that a NT_Topic dblTopic variable already exists
// start publishing. It's recommended that the type string be standard
// for all types except string and raw.
NT_Publisher dblPub = NT_Publish(dblTopic, NT_DOUBLE, "double", NULL, 0);
// publish options may be specified
struct NT_PubSubOptions options;
memset(&options, 0, sizeof(options));
options.structSize = sizeof(options);
options.keepDuplicates = 1;  // true
NT_Publisher dblPub = NT_Publish(dblTopic, NT_DOUBLE, "double", &options);
// PublishEx allows setting initial properties. The properties string must
// be a JSON map.
NT_Publisher dblPub =
    NT_PublishEx(dblTopic, NT_DOUBLE, "double", "{\"myprop\", 5}", NULL, 0);
// publish a default value
NT_SetDefaultDouble(dblPub, 0.0);
// publish a value with current timestamp
NT_SetDouble(dblPub, 1.0);
```

(continues on next page)

(continued from previous page)

```
NT_SetDouble(dblPub, 2.0, 0);  // 0 = use current time
// publish a value with a specific timestamp; NT_Now() can
// be used to get the current time.
int64_t time = NT_Now();
NT_SetDouble(dblPub, 3.0, time);
// stop publishing
NT_Unpublish(dblPub);
```

Python

```
class Example:
    def __init__(self, dblTopic: ntcore.DoubleTopic):
        # start publishing; the return value must be retained (in this case, via
        # an instance variable)
        self.dblPub = dblTopic.publish()
        # publish options may be specified using PubSubOption
        self.dblPub = dblTopic.publish(ntcore.PubSubOptions(keepDuplicates=True))
        # publishEx provides additional options such as setting initial
        # properties and using a custom type string. Using a custom type string for
        # types other than raw and string is not recommended. The properties
        # must be a JSON map.
        self.dblPub = dblTopic.publishEx("double", {"myprop": 5})
    def periodic(self):
        # publish a default value
        self.dblPub.setDefault(0.0)
        # publish a value with current timestamp
        self.dblPub.set(1.0)
        self.dblPub.set(2.0, 0)  # 0 = use current time
        # publish a value with a specific timestamp with microsecond resolution.
        # On the roboRIO, this is the same as the FPGA timestamp (e.g.
        # RobotController.getFPGATime())
        self.dblPub.set(3.0, ntcore._now())
    # often not required in robot code, unless this class doesn't exist for
    # the lifetime of the entire robot program, in which case close() needs to be
    # called to stop publishing
    def close(self):
        # stop publishing
        self.dblPub.close()
```

29.3.2 Subscribing to a Topic

A *subscriber* receives value updates made to a topic. Similar to publishers, NetworkTable subscribers are represented as type-specific Subscriber classes (e.g. `BooleanSubscriber`: Java, C++, Python) that must be stored somewhere to continue subscribing.

Subscribers have a range of different ways to read received values. It's possible to just read the most recent value using `get()`, read the most recent value, along with its timestamp, using `getAtomic()`, or get an array of all value changes since the last call using `readQueue()` or `readQueueValues()`.

Java

```
public class Example {
  // the subscriber is an instance variable so its lifetime matches that of the class
  final DoubleSubscriber dblSub;
  public Example(DoubleTopic dblTopic) {
    // start subscribing; the return value must be retained.
    // the parameter is the default value if no value is available when get() is␣
↪called
    dblSub = dblTopic.subscribe(0.0);
    // subscribe options may be specified using PubSubOption
    dblSub =
        dblTopic.subscribe(0.0, PubSubOption.keepDuplicates(true), PubSubOption.
↪pollStorage(10));
    // subscribeEx provides the options of using a custom type string.
    // Using a custom type string for types other than raw and string is not␣
↪recommended.
    dblSub = dblTopic.subscribeEx("double", 0.0);
  }
  public void periodic() {
    // simple get of most recent value; if no value has been published,
    // returns the default value passed to the subscribe() function
    double val = dblSub.get();
    // get the most recent value; if no value has been published, returns
    // the passed-in default value
    double val = dblSub.get(-1.0);
    // subscribers also implement the appropriate Supplier interface, e.g.␣
↪DoubleSupplier
    double val = dblSub.getAsDouble();
    // get the most recent value, along with its timestamp
    TimestampedDouble tsVal = dblSub.getAtomic();
    // read all value changes since the last call to readQueue/readQueueValues
    // readQueue() returns timestamps; readQueueValues() does not.
    TimestampedDouble[] tsUpdates = dblSub.readQueue();
    double[] valUpdates = dblSub.readQueueValues();
  }
  // often not required in robot code, unless this class doesn't exist for
  // the lifetime of the entire robot program, in which case close() needs to be
  // called to stop subscribing
  public void close() {
    // stop subscribing
    dblSub.close();
  }
}
```

C++

```
class Example {
  // the subscriber is an instance variable so its lifetime matches that of the class
  // subscribing is automatically stopped when dblSub is destroyed by the class␣
↪destructor
  nt::DoubleSubscriber dblSub;
 public:
  explicit Example(nt::DoubleTopic dblTopic) {
    // start subscribing; the return value must be retained.
```

(continues on next page)

(continued from previous page)

```
    // the parameter is the default value if no value is available when get() is␣
↪called
    dblSub = dblTopic.Subscribe(0.0);
    // subscribe options may be specified using PubSubOptions
    dblSub =
        dblTopic.subscribe(0.0,
        {.pollStorage = 10, .keepDuplicates = true});
    // SubscribeEx provides the options of using a custom type string.
    // Using a custom type string for types other than raw and string is not␣
↪recommended.
    dblSub = dblTopic.SubscribeEx("double", 0.0);
  }
  void Periodic() {
    // simple get of most recent value; if no value has been published,
    // returns the default value passed to the Subscribe() function
    double val = dblSub.Get();
    // get the most recent value; if no value has been published, returns
    // the passed-in default value
    double val = dblSub.Get(-1.0);
    // get the most recent value, along with its timestamp
    nt::TimestampedDouble tsVal = dblSub.GetAtomic();
    // read all value changes since the last call to ReadQueue/ReadQueueValues
    // ReadQueue() returns timestamps; ReadQueueValues() does not.
    std::vector<nt::TimestampedDouble> tsUpdates = dblSub.ReadQueue();
    std::vector<double> valUpdates = dblSub.ReadQueueValues();
  }
};
```

C++ (Handle-based)

```
class Example {
  // the subscriber is an instance variable, but since it's a handle, it's
  // not automatically released, so we need a destructor
  NT_Subscriber dblSub;
 public:
  explicit Example(NT_Topic dblTopic) {
    // start subscribing
    // Using a custom type string for types other than raw and string is not␣
↪recommended.
    dblSub = nt::Subscribe(dblTopic, NT_DOUBLE, "double");
    // subscribe options may be specified using PubSubOptions
    dblSub =
        nt::Subscribe(dblTopic, NT_DOUBLE, "double",
        {.pollStorage = 10, .keepDuplicates = true});
  }
  void Periodic() {
    // get the most recent value; if no value has been published, returns
    // the passed-in default value
    double val = nt::GetDouble(dblSub, 0.0);
    // get the most recent value, along with its timestamp
    nt::TimestampedDouble tsVal = nt::GetAtomic(dblSub, 0.0);
    // read all value changes since the last call to ReadQueue/ReadQueueValues
    // ReadQueue() returns timestamps; ReadQueueValues() does not.
    std::vector<nt::TimestampedDouble> tsUpdates = nt::ReadQueueDouble(dblSub);
```

(continues on next page)

(continued from previous page)

```
    std::vector<double> valUpdates = nt::ReadQueueValuesDouble(dblSub);
  }
  ~Example() {
    // stop subscribing
    nt::Unsubscribe(dblSub);
  }
```

C

```
// This code assumes that a NT_Topic dblTopic variable already exists
// start subscribing
// Using a custom type string for types other than raw and string is not recommended.
NT_Subscriber dblSub = NT_Subscribe(dblTopic, NT_DOUBLE, "double", NULL, 0);
// subscribe options may be specified using NT_PubSubOptions
struct NT_PubSubOptions options;
memset(&options, 0, sizeof(options));
options.structSize = sizeof(options);
options.keepDuplicates = 1;  // true
options.pollStorage = 10;
NT_Subscriber dblSub = NT_Subscribe(dblTopic, NT_DOUBLE, "double", &options);
// get the most recent value; if no value has been published, returns
// the passed-in default value
double val = NT_GetDouble(dblSub, 0.0);
// get the most recent value, along with its timestamp
struct NT_TimestampedDouble tsVal;
NT_GetAtomic(dblSub, 0.0, &tsVal);
NT_DisposeTimestamped(&tsVal);
// read all value changes since the last call to ReadQueue/ReadQueueValues
// ReadQueue() returns timestamps; ReadQueueValues() does not.
size_t tsUpdatesLen;
struct NT_TimestampedDouble* tsUpdates = NT_ReadQueueDouble(dblSub, &tsUpdatesLen);
NT_FreeQueueDouble(tsUpdates, tsUpdatesLen);
size_t valUpdatesLen;
double* valUpdates = NT_ReadQueueValuesDouble(dblSub, &valUpdatesLen);
NT_FreeDoubleArray(valUpdates, valUpdatesLen);
// stop subscribing
NT_Unsubscribe(dblSub);
```

Python

```
class Example:
    def __init__(self, dblTopic: ntcore.DoubleTopic):
        # start subscribing; the return value must be retained.
        # the parameter is the default value if no value is available when get() is␣
↪called
        self.dblSub = dblTopic.subscribe(0.0)
        # subscribe options may be specified using PubSubOption
        self.dblSub = dblTopic.subscribe(
            0.0, ntcore.PubSubOptions(keepDuplicates=True, pollStorage=10)
        )
        # subscribeEx provides the options of using a custom type string.
        # Using a custom type string for types other than raw and string is not␣
```

(continues on next page)

(continued from previous page)

```
↪recommended.
        dblSub = dblTopic.subscribeEx("double", 0.0)
    def periodic(self):
        # simple get of most recent value; if no value has been published,
        # returns the default value passed to the subscribe() function
        val = self.dblSub.get()
        # get the most recent value; if no value has been published, returns
        # the passed-in default value
        val = self.dblSub.get(-1.0)
        # get the most recent value, along with its timestamp
        tsVal = self.dblSub.getAtomic()
        # read all value changes since the last call to readQueue
        # readQueue() returns timestamps
        tsUpdates = self.dblSub.readQueue()
        # often not required in robot code, unless this class doesn't exist for
    # the lifetime of the entire robot program, in which case close() needs to be
    # called to stop subscribing
    def close(self):
        # stop subscribing
        self.dblSub.close()
```

29.3.3 Using Entry to Both Subscribe and Publish

An *entry* is a combined publisher and subscriber. The subscriber is always active, but the publisher is not created until a publish operation is performed (e.g. a value is "set", aka published, on the entry). This may be more convenient than maintaining a separate publisher and subscriber. Similar to publishers and subscribers, NetworkTable entries are represented as type-specific Entry classes (e.g. `BooleanEntry`: Java, C++, Python) that must be retained to continue subscribing (and publishing).

Java

```
public class Example {
  // the entry is an instance variable so its lifetime matches that of the class
  final DoubleEntry dblEntry;
  public Example(DoubleTopic dblTopic) {
    // start subscribing; the return value must be retained.
    // the parameter is the default value if no value is available when get() is␣
↪called
    dblEntry = dblTopic.getEntry(0.0);
    // publish and subscribe options may be specified using PubSubOption
    dblEntry =
        dblTopic.getEntry(0.0, PubSubOption.keepDuplicates(true), PubSubOption.
↪pollStorage(10));
    // getEntryEx provides the options of using a custom type string.
    // Using a custom type string for types other than raw and string is not␣
↪recommended.
    dblEntry = dblTopic.getEntryEx("double", 0.0);
  }
  public void periodic() {
    // entries support all the same methods as subscribers:
    double val = dblEntry.get();
```

(continues on next page)

(continued from previous page)

```
    double val = dblEntry.get(-1.0);
    double val = dblEntry.getAsDouble();
    TimestampedDouble tsVal = dblEntry.getAtomic();
    TimestampedDouble[] tsUpdates = dblEntry.readQueue();
    double[] valUpdates = dblEntry.readQueueValues();
    // entries also support all the same methods as publishers; the first time
    // one of these is called, an internal publisher is automatically created
    dblEntry.setDefault(0.0);
    dblEntry.set(1.0);
    dblEntry.set(2.0, 0);  // 0 = use current time
    long time = NetworkTablesJNI.now();
    dblEntry.set(3.0, time);
    myFunc(dblEntry);
  }
  public void unpublish() {
    // you can stop publishing while keeping the subscriber alive
    dblEntry.unpublish();
  }
  // often not required in robot code, unless this class doesn't exist for
  // the lifetime of the entire robot program, in which case close() needs to be
  // called to stop subscribing
  public void close() {
    // stop subscribing/publishing
    dblEntry.close();
  }
}
```

C++

```
class Example {
  // the entry is an instance variable so its lifetime matches that of the class
  // subscribing/publishing is automatically stopped when dblEntry is destroyed by
  // the class destructor
  nt::DoubleEntry dblEntry;
 public:
  explicit Example(nt::DoubleTopic dblTopic) {
    // start subscribing; the return value must be retained.
    // the parameter is the default value if no value is available when get() is␣
↪called
    dblEntry = dblTopic.GetEntry(0.0);
    // publish and subscribe options may be specified using PubSubOptions
    dblEntry =
        dblTopic.GetEntry(0.0,
        {.pollStorage = 10, .keepDuplicates = true});
    // GetEntryEx provides the options of using a custom type string.
    // Using a custom type string for types other than raw and string is not␣
↪recommended.
    dblEntry = dblTopic.GetEntryEx("double", 0.0);
  }
  void Periodic() {
    // entries support all the same methods as subscribers:
    double val = dblEntry.Get();
    double val = dblEntry.Get(-1.0);
    nt::TimestampedDouble tsVal = dblEntry.GetAtomic();
```

(continues on next page)

(continued from previous page)

```
    std::vector<nt::TimestampedDouble> tsUpdates = dblEntry.ReadQueue();
    std::vector<double> valUpdates = dblEntry.ReadQueueValues();
    // entries also support all the same methods as publishers; the first time
    // one of these is called, an internal publisher is automatically created
    dblEntry.SetDefault(0.0);
    dblEntry.Set(1.0);
    dblEntry.Set(2.0, 0);  // 0 = use current time
    int64_t time = nt::Now();
    dblEntry.Set(3.0, time);
  }
  void Unpublish() {
    // you can stop publishing while keeping the subscriber alive
    dblEntry.Unpublish();
  }
};
```

C++ (Handle-based)

```
class Example {
  // the entry is an instance variable, but since it's a handle, it's
  // not automatically released, so we need a destructor
  NT_Entry dblEntry;
 public:
  explicit Example(NT_Topic dblTopic) {
    // start subscribing
    // Using a custom type string for types other than raw and string is not␣
↪recommended.
    dblEntry = nt::GetEntry(dblTopic, NT_DOUBLE, "double");
    // publish and subscribe options may be specified using PubSubOptions
    dblEntry =
        nt::GetEntry(dblTopic, NT_DOUBLE, "double",
        {.pollStorage = 10, .keepDuplicates = true});
  }
  void Periodic() {
    // entries support all the same methods as subscribers:
    double val = nt::GetDouble(dblEntry, 0.0);
    nt::TimestampedDouble tsVal = nt::GetAtomic(dblEntry, 0.0);
    std::vector<nt::TimestampedDouble> tsUpdates = nt::ReadQueueDouble(dblEntry);
    std::vector<double> valUpdates = nt::ReadQueueValuesDouble(dblEntry);
    // entries also support all the same methods as publishers; the first time
    // one of these is called, an internal publisher is automatically created
    nt::SetDefaultDouble(dblPub, 0.0);
    nt::SetDouble(dblPub, 1.0);
    nt::SetDouble(dblPub, 2.0, 0);  // 0 = use current time
    int64_t time = nt::Now();
    nt::SetDouble(dblPub, 3.0, time);
  }
  void Unpublish() {
    // you can stop publishing while keeping the subscriber alive
    nt::Unpublish(dblEntry);
  }
  ~Example() {
    // stop publishing and subscribing
    nt::ReleaseEntry(dblEntry);
  }
```

C

```c
// This code assumes that a NT_Topic dblTopic variable already exists
// start subscribing
// Using a custom type string for types other than raw and string is not recommended.
NT_Entry dblEntry = NT_GetEntryEx(dblTopic, NT_DOUBLE, "double", NULL, 0);
// publish and subscribe options may be specified using NT_PubSubOptions
struct NT_PubSubOptions options;
memset(&options, 0, sizeof(options));
options.structSize = sizeof(options);
options.keepDuplicates = 1;  // true
options.pollStorage = 10;
NT_Entry dblEntry = NT_GetEntryEx(dblTopic, NT_DOUBLE, "double", &options);
// entries support all the same methods as subscribers:
double val = NT_GetDouble(dblEntry, 0.0);
struct NT_TimestampedDouble tsVal;
NT_GetAtomic(dblEntry, 0.0, &tsVal);
NT_DisposeTimestamped(&tsVal);
size_t tsUpdatesLen;
struct NT_TimestampedDouble* tsUpdates = NT_ReadQueueDouble(dblEntry, &tsUpdatesLen);
NT_FreeQueueDouble(tsUpdates, tsUpdatesLen);
size_t valUpdatesLen;
double* valUpdates = NT_ReadQueueValuesDouble(dblEntry, &valUpdatesLen);
NT_FreeDoubleArray(valUpdates, valUpdatesLen);
// entries also support all the same methods as publishers; the first time
// one of these is called, an internal publisher is automatically created
NT_SetDefaultDouble(dblPub, 0.0);
NT_SetDouble(dblPub, 1.0);
NT_SetDouble(dblPub, 2.0, 0);  // 0 = use current time
int64_t time = NT_Now();
NT_SetDouble(dblPub, 3.0, time);
// you can stop publishing while keeping the subscriber alive
// it's not necessary to call this before NT_ReleaseEntry()
NT_Unpublish(dblEntry);
// stop subscribing
NT_ReleaseEntry(dblEntry);
```

Python

```python
class Example:
    def __init__(self, dblTopic: ntcore.DoubleTopic):
        # start subscribing; the return value must be retained.
        # the parameter is the default value if no value is available when get() is␣
↪called
        self.dblEntry = dblTopic.getEntry(0.0)
        # publish and subscribe options may be specified using PubSubOption
        self.dblEntry = dblTopic.getEntry(
            0.0, ntcore.PubSubOptions(keepDuplicates=True, pollStorage=10)
        )
        # getEntryEx provides the options of using a custom type string.
        # Using a custom type string for types other than raw and string is not␣
↪recommended.
        self.dblEntry = dblTopic.getEntryEx("double", 0.0)
    def periodic(self):
        # entries support all the same methods as subscribers:
```

(continues on next page)

(continued from previous page)

```
        val = self.dblEntry.get()
        val = self.dblEntry.get(-1.0)
        val = self.dblEntry.getAsDouble()
        tsVal = self.dblEntry.getAtomic()
        tsUpdates = self.dblEntry.readQueue()
        # entries also support all the same methods as publishers; the first time
        # one of these is called, an internal publisher is automatically created
        self.dblEntry.setDefault(0.0)
        self.dblEntry.set(1.0)
        self.dblEntry.set(2.0, 0)  # 0 = use current time
        time = ntcore._now()
        self.dblEntry.set(3.0, time)
    def unpublish(self):
        # you can stop publishing while keeping the subscriber alive
        self.dblEntry.unpublish()
    # often not required in robot code, unless this class doesn't exist for
    # the lifetime of the entire robot program, in which case close() needs to be
    # called to stop subscribing
    def close(self):
        # stop subscribing/publishing
        self.dblEntry.close()
```

29.3.4 Using GenericEntry, GenericPublisher, and GenericSubscriber

For the most robust code, using the type-specific Publisher, Subscriber, and Entry classes is recommended, but in some cases it may be easier to write code that uses type-specific get and set function calls instead of having the NetworkTables type be exposed via the class (object) type. The `GenericPublisher` (Java, C++, Python), `GenericSubscriber` (Java, C++, Python), and `GenericEntry` (Java, C++, Python) classes enable this approach.

Java

```
public class Example {
  // the entry is an instance variable so its lifetime matches that of the class
  final GenericPublisher pub;
  final GenericSubscriber sub;
  final GenericEntry entry;
  public Example(Topic topic) {
    // start subscribing; the return value must be retained.
    // when publishing, a type string must be provided
    pub = topic.genericPublish("double");
    // subscribing can optionally include a type string
    // unlike type-specific subscribers, no default value is provided
    sub = topic.genericSubscribe();
    sub = topic.genericSubscribe("double");
    // when getting an entry, the type string is also optional; if not provided
    // the publisher data type will be determined by the first publisher-creating call
    entry = topic.getGenericEntry();
    entry = topic.getGenericEntry("double");
    // publish and subscribe options may be specified using PubSubOption
    pub = topic.genericPublish("double",
        PubSubOption.keepDuplicates(true), PubSubOption.pollStorage(10));
```

(continues on next page)

(continued from previous page)

```
    sub =
        topic.genericSubscribe(PubSubOption.keepDuplicates(true), PubSubOption.
↪pollStorage(10));
    entry =
        topic.getGenericEntry(PubSubOption.keepDuplicates(true), PubSubOption.
↪pollStorage(10));
    // genericPublishEx provides the option of setting initial properties.
    pub = topic.genericPublishEx("double", "{\"retained\": true}",
        PubSubOption.keepDuplicates(true), PubSubOption.pollStorage(10));
  }
  public void periodic() {
    // generic subscribers and entries have typed get operations; a default must be␣
↪provided
    double val = sub.getDouble(-1.0);
    double val = entry.getDouble(-1.0);
    // they also support an untyped get (also meets Supplier<NetworkTableValue>␣
↪interface)
    NetworkTableValue val = sub.get();
    NetworkTableValue val = entry.get();
    // they also support readQueue
    NetworkTableValue[] updates = sub.readQueue();
    NetworkTableValue[] updates = entry.readQueue();
    // publishers and entries have typed set operations; these return false if the
    // topic already exists with a mismatched type
    boolean success = pub.setDefaultDouble(1.0);
    boolean success = pub.setBoolean(true);
    // they also implement a generic set and Consumer<NetworkTableValue> interface
    boolean success = entry.set(NetworkTableValue.makeDouble(...));
    boolean success = entry.accept(NetworkTableValue.makeDouble(...));
  }
  public void unpublish() {
    // you can stop publishing an entry while keeping the subscriber alive
    entry.unpublish();
  }
  // often not required in robot code, unless this class doesn't exist for
  // the lifetime of the entire robot program, in which case close() needs to be
  // called to stop subscribing/publishing
  public void close() {
    pub.close();
    sub.close();
    entry.close();
  }
}
```

C++

```
class Example {
  // the entry is an instance variable so its lifetime matches that of the class
  // subscribing/publishing is automatically stopped when dblEntry is destroyed by
  // the class destructor
  nt::GenericPublisher pub;
  nt::GenericSubscriber sub;
  nt::GenericEntry entry;
 public:
```

(continues on next page)

(continued from previous page)

```
  Example(nt::Topic topic) {
    // start subscribing; the return value must be retained.
    // when publishing, a type string must be provided
    pub = topic.GenericPublish("double");
    // subscribing can optionally include a type string
    // unlike type-specific subscribers, no default value is provided
    sub = topic.GenericSubscribe();
    sub = topic.GenericSubscribe("double");
    // when getting an entry, the type string is also optional; if not provided
    // the publisher data type will be determined by the first publisher-creating call
    entry = topic.GetEntry();
    entry = topic.GetEntry("double");
    // publish and subscribe options may be specified using PubSubOptions
    pub = topic.GenericPublish("double",
        {.pollStorage = 10, .keepDuplicates = true});
    sub = topic.GenericSubscribe(
        {.pollStorage = 10, .keepDuplicates = true});
    entry = topic.GetGenericEntry(
        {.pollStorage = 10, .keepDuplicates = true});
    // genericPublishEx provides the option of setting initial properties.
    pub = topic.genericPublishEx("double", {{"myprop", 5}},
        {.pollStorage = 10, .keepDuplicates = true});
  }
  void Periodic() {
    // generic subscribers and entries have typed get operations; a default must be␣
↪provided
    double val = sub.GetDouble(-1.0);
    double val = entry.GetDouble(-1.0);
    // they also support an untyped get
    nt::NetworkTableValue val = sub.Get();
    nt::NetworkTableValue val = entry.Get();
    // they also support readQueue
    std::vector<nt::NetworkTableValue> updates = sub.ReadQueue();
    std::vector<nt::NetworkTableValue> updates = entry.ReadQueue();
    // publishers and entries have typed set operations; these return false if the
    // topic already exists with a mismatched type
    bool success = pub.SetDefaultDouble(1.0);
    bool success = pub.SetBoolean(true);
    // they also implement a generic set and Consumer<NetworkTableValue> interface
    bool success = entry.Set(nt::NetworkTableValue::MakeDouble(...));
  }
  void Unpublish() {
    // you can stop publishing an entry while keeping the subscriber alive
    entry.Unpublish();
  }
};
```

Python

```
class Example:
    def __init__(self, topic: ntcore.Topic):
        # start subscribing; the return value must be retained.
        # when publishing, a type string must be provided
        self.pub = topic.genericPublish("double")
        # subscribing can optionally include a type string
        # unlike type-specific subscribers, no default value is provided
        self.sub = topic.genericSubscribe()
        self.sub = topic.genericSubscribe("double")
        # when getting an entry, the type string is also optional; if not provided
        # the publisher data type will be determined by the first publisher-creating␣
↪call
        self.entry = topic.getGenericEntry()
        self.entry = topic.getGenericEntry("double")
        # publish and subscribe options may be specified using PubSubOption
        self.pub = topic.genericPublish(
            "double", ntcore.PubSubOptions(keepDuplicates=True, pollStorage=10)
        )
        self.sub = topic.genericSubscribe(
            ntcore.PubSubOptions(keepDuplicates=True, pollStorage=10)
        )
        self.entry = topic.getGenericEntry(
            ntcore.PubSubOptions(keepDuplicates=True, pollStorage=10)
        )
        # genericPublishEx provides the option of setting initial properties.
        self.pub = topic.genericPublishEx(
            "double",
            {"retained": true},
            ntcore.PubSubOptions(keepDuplicates=True, pollStorage=10),
        )
    def periodic(self):
        # generic subscribers and entries have typed get operations; a default must␣
↪be provided
        val = self.sub.getDouble(-1.0)
        val = self.entry.getDouble(-1.0)
        # they also support an untyped get (also meets Supplier<NetworkTableValue>␣
↪interface)
        val = self.sub.get()
        val = self.entry.get()
        # they also support readQueue
        updates = self.sub.readQueue()
        updates = self.entry.readQueue()
        # publishers and entries have typed set operations; these return false if the
        # topic already exists with a mismatched type
        success = self.pub.setDefaultDouble(1.0)
        success = self.pub.setBoolean(True)
        # they also implement a generic set
        success = self.entry.set(ntcore.Value.makeDouble(...))
    def unpublish(self):
        # you can stop publishing an entry while keeping the subscriber alive
        self.entry.unpublish()
    # often not required in robot code, unless this class doesn't exist for
    # the lifetime of the entire robot program, in which case close() needs to be
    # called to stop subscribing/publishing
    def close(self):
```

(continues on next page)

(continued from previous page)

```
        self.pub.close()
        self.sub.close()
        self.entry.close()
```

29.3.5 Subscribing to Multiple Topics

While in most cases it's only necessary to subscribe to individual topics, it is sometimes useful (e.g. in dashboard applications) to subscribe and get value updates for changes to multiple topics. Listeners (see *Listening for Changes*) can be used directly, but creating a `MultiSubscriber` (Java, C++) allows specifying subscription options and reusing the same subscriber for multiple listeners.

Java

```
public class Example {
  // the subscriber is an instance variable so its lifetime matches that of the class
  final MultiSubscriber multiSub;
  final NetworkTableListenerPoller poller;
  public Example(NetworkTableInstance inst) {
    // start subscribing; the return value must be retained.
    // provide an array of topic name prefixes
    multiSub = new MultiSubscriber(inst, new String[] {"/table1/", "/table2/"});
    // subscribe options may be specified using PubSubOption
    multiSub = new MultiSubscriber(inst, new String[] {"/table1/", "/table2/"},
        PubSubOption.keepDuplicates(true));
    // to get value updates from a MultiSubscriber, it's necessary to create a␣
↪listener
    // (see the listener documentation for more details)
    poller = new NetworkTableListenerPoller(inst);
    poller.addListener(multiSub, EnumSet.of(NetworkTableEvent.Kind.kValueAll));
  }
  public void periodic() {
    // read value events
    NetworkTableEvent[] events = poller.readQueue();
    for (NetworkTableEvent event : events) {
      NetworkTableValue value = event.valueData.value;
    }
  }
  // often not required in robot code, unless this class doesn't exist for
  // the lifetime of the entire robot program, in which case close() needs to be
  // called to stop subscribing
  public void close() {
    // close listener
    poller.close();
    // stop subscribing
    multiSub.close();
  }
}
```

C++

```
class Example {
  // the subscriber is an instance variable so its lifetime matches that of the class
  // subscribing is automatically stopped when multiSub is destroyed by the class␣
↪destructor
  nt::MultiSubscriber multiSub;
  nt::NetworkTableListenerPoller poller;
 public:
  explicit Example(nt::NetworkTableInstance inst) {
    // start subscribing; the return value must be retained.
    // provide an array of topic name prefixes
    multiSub = nt::MultiSubscriber{inst, {{"/table1/", "/table2/"}}};
    // subscribe options may be specified using PubSubOption
    multiSub = nt::MultiSubscriber{inst, {{"/table1/", "/table2/"}},
        {.keepDuplicates = true}};
    // to get value updates from a MultiSubscriber, it's necessary to create a␣
↪listener
    // (see the listener documentation for more details)
    poller = nt::NetworkTableListenerPoller{inst};
    poller.AddListener(multiSub, nt::EventFlags::kValueAll);
  }
  void Periodic() {
    // read value events
    std::vector<nt::Event> events = poller.ReadQueue();
    for (auto&& event : events) {
      nt::NetworkTableValue value = event.GetValueEventData()->value;
    }
  }
};
```

C++ (Handle-based)

```
class Example {
  // the subscriber is an instance variable, but since it's a handle, it's
  // not automatically released, so we need a destructor
  NT_MultiSubscriber multiSub;
  NT_ListenerPoller poller;
 public:
  explicit Example(NT_Inst inst) {
    // start subscribing; the return value must be retained.
    // provide an array of topic name prefixes
    multiSub = nt::SubscribeMultiple(inst, {{"/table1/", "/table2/"}});
    // subscribe options may be specified using PubSubOption
    multiSub = nt::SubscribeMultiple(inst, {{"/table1/", "/table2/"}},
        {.keepDuplicates = true});
    // to get value updates from a MultiSubscriber, it's necessary to create a␣
↪listener
    // (see the listener documentation for more details)
    poller = nt::CreateListenerPoller(inst);
    nt::AddPolledListener(poller, multiSub, nt::EventFlags::kValueAll);
  }
  void Periodic() {
    // read value events
    std::vector<nt::Event> events = nt::ReadListenerQueue(poller);
```

(continues on next page)

(continued from previous page)

```
    for (auto&& event : events) {
      nt::NetworkTableValue value = event.GetValueEventData()->value;
    }
  }
  ~Example() {
    // close listener
    nt::DestroyListenerPoller(poller);
    // stop subscribing
    nt::UnsubscribeMultiple(multiSub);
  }
```

C

```
// This code assumes that a NT_Inst inst variable already exists
// start subscribing
// provide an array of topic name prefixes
struct NT_String prefixes[2];
prefixes[0].str = "/table1/";
prefixes[0].len = 8;
prefixes[1].str = "/table2/";
prefixes[1].len = 8;
NT_MultiSubscriber multiSub = NT_SubscribeMultiple(inst, prefixes, 2, NULL, 0);
// subscribe options may be specified using NT_PubSubOptions
struct NT_PubSubOptions options;
memset(&options, 0, sizeof(options));
options.structSize = sizeof(options);
options.keepDuplicates = 1;  // true
NT_MultiSubscriber multiSub = NT_SubscribeMultiple(inst, prefixes, 2, &options);
// to get value updates from a MultiSubscriber, it's necessary to create a listener
// (see the listener documentation for more details)
NT_ListenerPoller poller = NT_CreateListenerPoller(inst);
NT_AddPolledListener(poller, multiSub, NT_EVENT_VALUE_ALL);
// read value events
size_t eventsLen;
struct NT_Event* events = NT_ReadListenerQueue(poller, &eventsLen);
for (size_t i = 0; i < eventsLen; i++) {
  NT_Value* value = &events[i].data.valueData.value;
}
NT_DisposeEventArray(events, eventsLen);
// close listener
NT_DestroyListenerPoller(poller);
// stop subscribing
NT_UnsubscribeMultiple(multiSub);
```

Python

```
class Example:
    def __init__(self, inst: ntcore.NetworkTableInstance):
        # start subscribing; the return value must be retained.
        # provide an array of topic name prefixes
        self.multiSub = ntcore.MultiSubscriber(inst, ["/table1/", "/table2/"])
        # subscribe options may be specified using PubSubOption
        self.multiSub = ntcore.MultiSubscriber(
            inst, ["/table1/", "/table2/"], ntcore.PubSubOptions(keepDuplicates=True)
        )
        # to get value updates from a MultiSubscriber, it's necessary to create a␣
↪listener
        # (see the listener documentation for more details)
        self.poller = ntcore.NetworkTableListenerPoller(inst)
        self.poller.addListener(self.multiSub, ntcore.EventFlags.kValueAlls)
    def periodic(self):
        # read value events
        events = self.poller.readQueue()
        for event in events:
            value: ntcore.Value = event.data.value
    # often not required in robot code, unless this class doesn't exist for
    # the lifetime of the entire robot program, in which case close() needs to be
    # called to stop subscribing
    def close(self):
        # close listener
        self.poller.close()
        # stop subscribing
        self.multiSub.close()
```

29.3.6 Publish/Subscribe Options

Publishers and subscribers have various options that affect their behavior. Options can only be set at the creation of the publisher, subscriber, or entry. Options set on an entry affect both the publisher and subscriber portions of the entry. The above examples show how options can be set when creating a publisher or subscriber.

Subscriber options:

- `pollStorage`: Polling storage size for a subscription. Specifies the maximum number of updates NetworkTables should store between calls to the subscriber's `readQueue()` function. If zero, defaults to 1 if sendAll is false, 20 if sendAll is true.
- `topicsOnly`: Don't send value changes, only topic announcements. Defaults to false. As a client doesn't get topic announcements for topics it is not subscribed to, this option may be used with `MultiSubscriber` to get topic announcements for a particular topic name prefix, without also getting all value changes.
- `excludePublisher`: Used to exclude a single publisher's updates from being queued to the subscriber's `readQueue()` function. This is primarily useful in scenarios where you don't want local value updates to be "echoed back" to a local subscriber. Regardless of this setting, the topic value is updated–this only affects `readQueue()` on this subscriber.
- `disableRemote`: If true, remote value updates are not queued for `readQueue()`. Defaults to false. Regardless of this setting, the topic value is updated–this only affects `readQueue()` on this subscriber.

- `disableLocal`: If true, local value updates are not queued for `readQueue()`. Defaults to false. Regardless of this setting, the topic value is updated–this only affects `readQueue()` on this subscriber.

Subscriber and publisher options:

- `periodic`: How frequently changes will be sent over the network, in seconds. NetworkTables may send more frequently than this (e.g. use a combined minimum period for all values) or apply a restricted range to this value. The default is 0.1 seconds. For publishers, it specifies how frequently local changes should be sent over the network; for subscribers, it is a request to the server to send server changes at the requested rate. Note that regardless of the setting of this option, only value changes are sent, unless the `keepDuplicates` option is set.
- `sendAll`: If true, send all value changes over the network. Defaults to false. As with `periodic`, this is a request to the server for subscribers and a behavior change for publishers.
- `keepDuplicates`: If true, preserves duplicate value changes (rather than ignoring them). Defaults to false. As with `periodic`, this is a request to the server for subscribers and a behavior change for publishers.

Entry options:

- `excludeSelf`: Provides the same behavior as `excludePublisher` for the entry's internal publisher. Defaults to false.

29.3.7 NetworkTableEntry

`NetworkTableEntry` (Java, C++, Python) is a class that exists for backwards compatibility. New code should prefer using type-specific Publisher and Subscriber classes, or GenericEntry if non-type-specific access is needed.

It is similar to `GenericEntry` in that it supports both publishing and subscribing in a single object. However, unlike `GenericEntry`, `NetworkTableEntry` is not released (e.g. unsubscribes/unpublishes) if `close()` is called (in Java) or the object is destroyed (in C++); instead, it operates similar to `Topic`, in that only a single `NetworkTableEntry` exists for each topic and it lasts for the lifetime of the instance.

29.4 NetworkTables Instances

The NetworkTables implementation supports simultaneous operation of multiple "instances." Each instance has a completely independent set of topics, publishers, subscribers, and client/server state. This feature is mainly useful for unit testing. It allows a single program to be a member of two *NetworkTables* "networks" that contain different (and unrelated) sets of topics, or running both client and server instances in a single program.

For most general usage, you should use the "default" 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.

However, 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.

The `NetworkTableInstance` (Java, C++, Python) class provides the API abstraction for instances. The number of instances that can be simultaneously created is limited to 16 (including the default instance), so when using multiple instances in cases such as unit testing code, it's important to destroy instances that are no longer needed.

Destroying a NetworkTableInstance frees all resources related to the instance. All classes or handles that reference the instance (e.g. Topics, Publishers, and Subscribers) are invalidated and may result in unexpected behavior if used after the instance is destroyed–in particular, instance handles are reused so it's possible for a handle "left over" from a previously destroyed instance to refer to an unexpected resource in a newly created instance.

Java

```
// get the default NetworkTable instance
NetworkTableInstance defaultInst = NetworkTableInstance.getDefault();
// create a NetworkTable instance
NetworkTableInstance inst = NetworkTableInstance.create();
// destroy a NetworkTable instance
inst.close();
```

C++

```
// get the default NetworkTable instance
nt::NetworkTableInstance defaultInst = nt::NetworkTableInstance::GetDefault();
// create a NetworkTable instance
nt::NetworkTableInstance inst = nt::NetworkTableInstance::Create();
// destroy a NetworkTable instance; NetworkTableInstance objects are not RAII
nt::NetworkTableInstance::Destroy(inst);
```

C++ (Handle-based)

```
// get the default NetworkTable instance
NT_Instance defaultInst = nt::GetDefaultInstance();
// create a NetworkTable instance
NT_Instance inst = nt::CreateInstance();
// destroy a NetworkTable instance
nt::DestroyInstance(inst);
```

C

```
// get the default NetworkTable instance
NT_Instance defaultInst = NT_GetDefaultInstance();
// create a NetworkTable instance
NT_Instance inst = NT_CreateInstance();
// destroy a NetworkTable instance
NT_DestroyInstance(inst);
```

Python

```
import ntcore
# get the default NetworkTable instance
defaultInst = ntcore.NetworkTableInstance.getDefault()
# create a NetworkTable instance
inst = ntcore.NetworkTableInstance.create()
# destroy a NetworkTable instance
ntcore.NetworkTableInstance.destroy(inst)
```

29.5 NetworkTables Networking

The advantage of the robot program being the server is that it's at a known network name (and typically at a known address) that is based on the team number. This is why it's possible in both the NetworkTables client API and in most dashboards to simply provide the team number, rather than a server address. As the robot program is the server, note this means the NetworkTables server is running on the local computer when running in simulation.

29.5.1 Starting a NetworkTables Server

Java

```
NetworkTableInstance inst = NetworkTableInstance.getDefault();
inst.startServer();
```

C++

```
nt::NetworkTableInstance inst = nt::NetworkTableInstance::GetDefault();
inst.StartServer();
```

C++ (Handle-based)

```
NT_Inst inst = nt::GetDefaultInstance();
nt::StartServer(inst, "networktables.json", "", NT_DEFAULT_PORT3, NT_DEFAULT_PORT4);
```

C

```
NT_Inst inst = NT_GetDefaultInstance();
NT_StartServer(inst, "networktables.json", "", NT_DEFAULT_PORT3, NT_DEFAULT_PORT4);
```

Python

```
import ntcore
inst = ntcore.NetworkTableInstance.getDefault()
inst.startServer()
```

29.5.2 Starting a NetworkTables Client

Java

```
NetworkTableInstance inst = NetworkTableInstance.getDefault();
// start a NT4 client
inst.startClient4("example client");
// connect to a roboRIO with team number TEAM
inst.setServerTeam(TEAM);
// starting a DS client will try to get the roboRIO address from the DS application
inst.startDSClient();
// connect to a specific host/port
inst.setServer("host", NetworkTableInstance.kDefaultPort4)
```

C++

```
nt::NetworkTableInstance inst = nt::NetworkTableInstance::GetDefault();
// start a NT4 client
inst.StartClient4("example client");
// connect to a roboRIO with team number TEAM
inst.SetServerTeam(TEAM);
// starting a DS client will try to get the roboRIO address from the DS application
inst.StartDSClient();
// connect to a specific host/port
inst.SetServer("host", NT_DEFAULT_PORT4)
```

C++ (Handle-based)

```
NT_Inst inst = nt::GetDefaultInstance();
// start a NT4 client
nt::StartClient4(inst, "example client");
// connect to a roboRIO with team number TEAM
nt::SetServerTeam(inst, TEAM);
// starting a DS client will try to get the roboRIO address from the DS application
nt::StartDSClient(inst);
// connect to a specific host/port
nt::SetServer(inst, "host", NT_DEFAULT_PORT4)
```

C

```
NT_Inst inst = NT_GetDefaultInstance();
// start a NT4 client
NT_StartClient4(inst, "example client");
// connect to a roboRIO with team number TEAM
NT_SetServerTeam(inst, TEAM);
// starting a DS client will try to get the roboRIO address from the DS application
NT_StartDSClient(inst);
// connect to a specific host/port
NT_SetServer(inst, "host", NT_DEFAULT_PORT4)
```

Python

```
import ntcore
inst = ntcore.NetworkTableInstance.getDefault()
# start a NT4 client
inst.startClient4("example client")
# connect to a roboRIO with team number TEAM
inst.setServerTeam(TEAM)
# starting a DS client will try to get the roboRIO address from the DS application
inst.startDSClient()
# connect to a specific host/port
inst.setServer("host", ntcore.NetworkTableInstance.kDefaultPort4)
```

29.6 Listening for Changes

A common use case for *NetworkTables* is where a coprocessor generates values that need to be sent to the robot. For example, imagine that some image processing code running on a coprocessor 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.

There are a few different ways to detect that a topic's value has changed; the easiest way is to periodically call a subscriber's `get()`, `readQueue()`, or `readQueueValues()` function from the robot's periodic loop, as shown below:

Java

```
public class Example {
  final DoubleSubscriber ySub;
  double prev;
  public Example() {
    // get the default instance of NetworkTables
    NetworkTableInstance inst = NetworkTableInstance.getDefault();
    // get the subtable called "datatable"
    NetworkTable datatable = inst.getTable("datatable");
    // subscribe to the topic in "datatable" called "Y"
    ySub = datatable.getDoubleTopic("Y").subscribe(0.0);
  }
  public void periodic() {
    // get() can be used with simple change detection to the previous value
    double value = ySub.get();
    if (value != prev) {
      prev = value;  // save previous value
      System.out.println("X changed value: " + value);
    }
    // readQueueValues() provides all value changes since the last call;
    // this way it's not possible to miss a change by polling too slowly
    for (double iterVal : ySub.readQueueValues()) {
      System.out.println("X changed value: " + iterVal);
    }
    // readQueue() is similar to readQueueValues(), but provides timestamps
    // for each change as well
    for (TimestampedDouble tsValue : ySub.readQueue()) {
      System.out.println("X changed value: " + tsValue.value + " at local time " +␣
↪tsValue.timestamp);
    }
  }
  // may not be necessary for robot programs if this class lives for
  // the length of the program
  public void close() {
    ySub.close();
  }
}
```

C++

```
class Example {
  nt::DoubleSubscriber ySub;
  double prev = 0;
 public:
  Example() {
    // get the default instance of NetworkTables
    nt::NetworkTableInstance inst = nt::NetworkTableInstance::GetDefault();
    // get the subtable called "datatable"
    auto datatable = inst.GetTable("datatable");
    // subscribe to the topic in "datatable" called "Y"
    ySub = datatable->GetDoubleTopic("Y").Subscribe(0.0);
  }
  void Periodic() {
    // Get() can be used with simple change detection to the previous value
```

(continues on next page)

(continued from previous page)

```
    double value = ySub.Get();
    if (value != prev) {
      prev = value;  // save previous value
      fmt::print("X changed value: {}\n", value);
    }
    // ReadQueueValues() provides all value changes since the last call;
    // this way it's not possible to miss a change by polling too slowly
    for (double iterVal : ySub.ReadQueueValues()) {
      fmt::print("X changed value: {}\n", iterVal);
    }
    // ReadQueue() is similar to ReadQueueValues(), but provides timestamps
    // for each change as well
    for (nt::TimestampedDouble tsValue : ySub.ReadQueue()) {
      fmt::print("X changed value: {} at local time {}\n", tsValue.value, tsValue.
↪timestamp);
    }
  }
};
```

C++ (Handle-based)

```
class Example {
  NT_Subscriber ySub;
  double prev = 0;
 public:
  Example() {
    // get the default instance of NetworkTables
    NT_Inst inst = nt::GetDefaultInstance();
    // subscribe to the topic in "datatable" called "Y"
    ySub = nt::Subscribe(nt::GetTopic(inst, "/datatable/Y"), NT_DOUBLE, "double");
  }
  void Periodic() {
    // Get() can be used with simple change detection to the previous value
    double value = nt::GetDouble(ySub, 0.0);
    if (value != prev) {
      prev = value;  // save previous value
      fmt::print("X changed value: {}\n", value);
    }
    // ReadQueue() provides all value changes since the last call;
    // this way it's not possible to miss a change by polling too slowly
    for (nt::TimestampedDouble value : nt::ReadQueueDouble(ySub)) {
      fmt::print("X changed value: {} at local time {}\n", tsValue.value, tsValue.
↪timestamp);
    }
  }
};
```

Python

```
class Example:
    def __init__(self) -> None:
        # get the default instance of NetworkTables
        inst = ntcore.NetworkTableInstance.getDefault()
        # get the subtable called "datatable"
        datatable = inst.getTable("datatable")
        # subscribe to the topic in "datatable" called "Y"
        self.ySub = datatable.getDoubleTopic("Y").subscribe(0.0)
        self.prev = 0
    def periodic(self):
        # get() can be used with simple change detection to the previous value
        value = self.ySub.get()
        if value != self.prev:
            self.prev = value
            # save previous value
            print("X changed value: " + value)
        # readQueue() provides all value changes since the last call;
        # this way it's not possible to miss a change by polling too slowly
        for tsValue in self.ySub.readQueue():
            print(f"X changed value: {tsValue.value} at local time {tsValue.time}")
            # may not be necessary for robot programs if this class lives for
    # the length of the program
    def close(self):
        self.ySub.close()
```

With a command-based robot, it's also possible to use `NetworkBooleanEvent` to link boolean topic changes to callback actions (e.g. running commands).

While these functions suffice for value changes on a single topic, they do not provide insight into changes to topics (when a topic is published or unpublished, or when a topic's properties change) or network connection changes (e.g. when a client connects or disconnects). They also don't provide a way to get in-order updates for value changes across multiple topics. For these needs, NetworkTables provides an event listener facility.

The easiest way to use listeners is via `NetworkTableInstance`. For more automatic control over listener lifetime (particularly in C++), and to operate without a background thread, NetworkTables also provides separate classes for both polled listeners (`NetworkTableListenerPoller`), which store events into an internal queue that must be periodically read to get the queued events, and threaded listeners (`NetworkTableListener`), which call a callback function from a background thread.

29.6.1 NetworkTableEvent

All listener callbacks take a single `NetworkTableEvent` parameter, and similarly, reading a listener poller returns an array of `NetworkTableEvent`. The event contains information including what kind of event it is (e.g. a value update, a new topic, a network disconnect), the handle of the listener that caused the event to be generated, and more detailed information that depends on the type of the event (connection information for connection events, topic information for topic-related events, value data for value updates, and the log message for log message events).

29.6.2 Using NetworkTableInstance to Listen for Changes

The below example listens to various kinds of events using `NetworkTableInstance`. The listener callback provided to any of the addListener functions will be called asynchronously from a background thread when a matching event occurs.

> **Warning**
>
> Because the listener callback is called from a separate background thread, it's important to use thread-safe synchronization approaches such as mutexes or atomics to pass data to/from the main code and the listener callback function.

The `addListener` functions in NetworkTableInstance return a listener handle. This can be used to remove the listener later.

Java

```
public class Example {
  final DoubleSubscriber ySub;
  // use an AtomicReference to make updating the value thread-safe
  final AtomicReference<Double> yValue = new AtomicReference<Double>();
  // retain listener handles for later removal
  int connListenerHandle;
  int valueListenerHandle;
  int topicListenerHandle;
  public Example() {
    // get the default instance of NetworkTables
    NetworkTableInstance inst = NetworkTableInstance.getDefault();
    // add a connection listener; the first parameter will cause the
    // callback to be called immediately for any current connections
    connListenerHandle = inst.addConnectionListener(true, event -> {
      if (event.is(NetworkTableEvent.Kind.kConnected)) {
        System.out.println("Connected to " + event.connInfo.remote_id);
      } else if (event.is(NetworkTableEvent.Kind.kDisconnected)) {
        System.out.println("Disconnected from " + event.connInfo.remote_id);
      }
    });
    // get the subtable called "datatable"
    NetworkTable datatable = inst.getTable("datatable");
    // subscribe to the topic in "datatable" called "Y"
    ySub = datatable.getDoubleTopic("Y").subscribe(0.0);
    // add a listener to only value changes on the Y subscriber
    valueListenerHandle = inst.addListener(
        ySub,
        EnumSet.of(NetworkTableEvent.Kind.kValueAll),
        event -> {
          // can only get doubles because it's a DoubleSubscriber, but
          // could check value.isDouble() here too
          yValue.set(event.valueData.value.getDouble());
        });
    // add a listener to see when new topics are published within datatable
    // the string array is an array of topic name prefixes.
    topicListenerHandle = inst.addListener(
        new String[] { datatable.getPath() + "/" },
```

(continues on next page)

(continued from previous page)

```
        EnumSet.of(NetworkTableEvent.Kind.kTopic),
        event -> {
          if (event.is(NetworkTableEvent.Kind.kPublish)) {
            // topicInfo.name is the full topic name, e.g. "/datatable/X"
            System.out.println("newly published " + event.topicInfo.name);
          }
        });
  }
  public void periodic() {
    // get the latest value by reading the AtomicReference; set it to null
    // when we read to ensure we only get value changes
    Double value = yValue.getAndSet(null);
    if (value != null) {
      System.out.println("got new value " + value);
    }
  }
  // may not be needed for robot programs if this class exists for the
  // lifetime of the program
  public void close() {
    NetworkTableInstance inst = NetworkTableInstance.getDefault();
    inst.removeListener(topicListenerHandle);
    inst.removeListener(valueListenerHandle);
    inst.removeListener(connListenerHandle);
    ySub.close();
  }
}
```

C++

```
class Example {
  nt::DoubleSubscriber ySub;
  // use a mutex to make updating the value and flag thread-safe
  wpi::mutex mutex;
  double yValue;
  bool yValueUpdated = false;
  // retain listener handles for later removal
  NT_Listener connListenerHandle;
  NT_Listener valueListenerHandle;
  NT_Listener topicListenerHandle;
 public:
  Example() {
    // get the default instance of NetworkTables
    nt::NetworkTableInstance inst = nt::NetworkTableInstance::GetDefault();
    // add a connection listener; the first parameter will cause the
    // callback to be called immediately for any current connections
    connListenerHandle = inst.AddConnectionListener(true, [] (const nt::Event& event)
↪{
      if (event.Is(nt::EventFlags::kConnected)) {
        fmt::print("Connected to {}\n", event.GetConnectionInfo()->remote_id);
      } else if (event.Is(nt::EventFlags::kDisconnected)) {
        fmt::print("Disconnected from {}\n", event.GetConnectionInfo()->remote_id);
      }
    });
    // get the subtable called "datatable"
```

(continues on next page)

(continued from previous page)

```
    auto datatable = inst.GetTable("datatable");
    // subscribe to the topic in "datatable" called "Y"
    ySub = datatable.GetDoubleTopic("Y").Subscribe(0.0);
    // add a listener to only value changes on the Y subscriber
    valueListenerHandle = inst.AddListener(
        ySub,
        nt::EventFlags::kValueAll,
        [this] (const nt::Event& event) {
          // can only get doubles because it's a DoubleSubscriber, but
          // could check value.IsDouble() here too
          std::scoped_lock lock{mutex};
          yValue = event.GetValueData()->value.GetDouble();
          yValueUpdated = true;
        });
    // add a listener to see when new topics are published within datatable
    // the string array is an array of topic name prefixes.
    topicListenerHandle = inst.AddListener(
        {{fmt::format("{}/", datatable->GetPath())}},
        nt::EventFlags::kTopic,
        [] (const nt::Event& event) {
          if (event.Is(nt::EventFlags::kPublish)) {
            // name is the full topic name, e.g. "/datatable/X"
            fmt::print("newly published {}\n", event.GetTopicInfo()->name);
          }
        });
  }
  void Periodic() {
    // get the latest value by reading the value; set it to false
    // when we read to ensure we only get value changes
    wpi::scoped_lock lock{mutex};
    if (yValueUpdated) {
      yValueUpdated = false;
      fmt::print("got new value {}\n", yValue);
    }
  }
  ~Example() {
    nt::NetworkTableInstance inst = nt::NetworkTableInstance::GetDefault();
    inst.RemoveListener(connListenerHandle);
    inst.RemoveListener(valueListenerHandle);
    inst.RemoveListener(topicListenerHandle);
  }
};
```

Python

```
import ntcore
import threading
class Example:
    def __init__(self) -> None:
        # get the default instance of NetworkTables
        inst = ntcore.NetworkTableInstance.getDefault()
        # Use a mutex to ensure thread safety
        self.lock = threading.Lock()
        self.yValue = None
```

(continues on next page)

(continued from previous page)

```
        # add a connection listener; the first parameter will cause the
        # callback to be called immediately for any current connections
        def _connect_cb(event: ntcore.Event):
            if event.is_(ntcore.EventFlags.kConnected):
                print("Connected to", event.data.remote_id)
            elif event.is_(ntcore.EventFlags.kDisconnected):
                print("Disconnected from", event.data.remote_id)
                self.connListenerHandle = inst.addConnectionListener(True, _connect_
↪cb)
        # get the subtable called "datatable"
        datatable = inst.getTable("datatable")
        # subscribe to the topic in "datatable" called "Y"
        self.ySub = datatable.getDoubleTopic("Y").subscribe(0.0)
        # add a listener to only value changes on the Y subscriber
        def _on_ysub(event: ntcore.Event):
            # can only get doubles because it's a DoubleSubscriber, but
            # could check value.isDouble() here too
            with self.lock:
                self.yValue = event.data.value.getDouble()
                self.valueListenerHandle = inst.addListener(
            self.ySub, ntcore.EventFlags.kValueAll, _on_ysub
        )
        # add a listener to see when new topics are published within datatable
        # the string array is an array of topic name prefixes.
        def _on_pub(event: ntcore.Event):
            if event.is_(ntcore.EventFlags.kPublish):
                # topicInfo.name is the full topic name, e.g. "/datatable/X"
                print("newly published", event.data.name)
                self.topicListenerHandle = inst.addListener(
            [datatable.getPath() + "/"], ntcore.EventFlags.kTopic, _on_pub
        )
            def periodic(self):
        # get the latest value by reading the value; set it to null
        # when we read to ensure we only get value changes
        with self.lock:
            value, self.yValue = self.yValue, None
                if value is not None:
            print("got new value", value)
            # may not be needed for robot programs if this class exists for the
    # lifetime of the program
    def close(self):
        inst = ntcore.NetworkTableInstance.getDefault()
        inst.removeListener(self.topicListenerHandle)
        inst.removeListener(self.valueListenerHandle)
        inst.removeListener(self.connListenerHandle)
        self.ySub.close()
```

29.7 Writing a Simple NetworkTables Robot Program

In a robot program, a NetworkTables server is automatically started on the default instance. So it's only necessary to get the default instance to start publishing or subscribing and have it visible over the network.

The example robot program below publishes incrementing X and Y values to a table named `datatable`. 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 topic.

JAVA

```
package edu.wpi.first.wpilibj.templates;
import edu.wpi.first.wpilibj.TimedRobot;
import edu.wpi.first.networktables.DoublePublisher;
import edu.wpi.first.networktables.NetworkTable;
import edu.wpi.first.networktables.NetworkTableInstance;
public class EasyNetworkTableExample extends TimedRobot {
  DoublePublisher xPub;
  DoublePublisher yPub;

  public Robot() {
    // Get the default instance of NetworkTables that was created automatically
    // when the robot 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");
    // Start publishing topics within that table that correspond to the X and Y values
    // for some operation in your program.
    // The topic names are actually "/datatable/x" and "/datatable/y".
    xPub = table.getDoubleTopic("x").publish();
    yPub = table.getDoubleTopic("y").publish();
  }

  double x = 0;
  double y = 0;
  public void teleopPeriodic() {
    // Publish values that are constantly increasing.
    xPub.set(x);
    yPub.set(y);
    x += 0.05;
    y += 1.0;
  }
}
```

C++

```
#include <frc/TimedRobot.h>
#include <networktables/DoubleTopic.h>
#include <networktables/NetworkTable.h>
#include <networktables/NetworkTableInstance.h>
class EasyNetworkExample : public frc::TimedRobot {
 public:
  nt::DoublePublisher xPub;
  nt::DoublePublisher yPub;
  Robot() {
    // Get the default instance of NetworkTables that was created automatically
    // when the robot program starts
    auto inst = nt::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.
    auto table = inst.GetTable("datatable");
    // Start publishing topics within that table that correspond to the X and Y values
    // for some operation in your program.
    // The topic names are actually "/datatable/x" and "/datatable/y".
    xPub = table->GetDoubleTopic("x").Publish();
    yPub = table->GetDoubleTopic("y").Publish();
  }
  double x = 0;
  double y = 0;
  void TeleopPeriodic() {
    // Publish values that are constantly increasing.
    xPub.Set(x);
    yPub.Set(y);
    x += 0.05;
    y += 0.05;
  }
}
START_ROBOT_CLASS(EasyNetworkExample)
```

PYTHON

```
import ntcore
import wpilib
class EasyNetworkTableExample(wpilib.TimedRobot):
    def robotInit(self) -> None:
        # Get the default instance of NetworkTables that was created automatically
        # when the robot program starts
        inst = ntcore.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.
        table = inst.getTable("datatable")
        # Start publishing topics within that table that correspond to the X and Y␣
↪values
        # for some operation in your program.
        # The topic names are actually "/datatable/x" and "/datatable/y".
        self.xPub = table.getDoubleTopic("x").publish()
        self.yPub = table.getDoubleTopic("y").publish()
```

(continues on next page)

(continued from previous page)

```
        self.x = 0
        self.y = 0
    def teleopPeriodic(self) -> None:
        # Publish values that are constantly increasing.
        self.xPub.set(self.x)
        self.yPub.set(self.y)
        self.x += 0.05
        self.y += 1.0
```

29.8 Creating a Client-side Program

If all you need to do is have your robot program communicate with a *COTS* coprocessor 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

```
import edu.wpi.first.networktables.DoubleSubscriber;
import edu.wpi.first.networktables.NetworkTable;
import edu.wpi.first.networktables.NetworkTableInstance;
import edu.wpi.first.networktables.NetworkTablesJNI;
import edu.wpi.first.util.CombinedRuntimeLoader;
import java.io.IOException;
import edu.wpi.first.cscore.CameraServerJNI;
import edu.wpi.first.math.WPIMathJNI;
import edu.wpi.first.util.WPIUtilJNI;
public class Program {
    public static void main(String[] args) throws IOException {
        NetworkTablesJNI.Helper.setExtractOnStaticLoad(false);
        WPIUtilJNI.Helper.setExtractOnStaticLoad(false);
        WPIMathJNI.Helper.setExtractOnStaticLoad(false);
        CameraServerJNI.Helper.setExtractOnStaticLoad(false);
        CombinedRuntimeLoader.loadLibraries(Program.class, "wpiutiljni", "wpimathjni",
↪ "ntcorejni",
            Core.NATIVE_LIBRARY_NAME, "cscorejni");
        new Program().run();
    }
    public void run() {
        NetworkTableInstance inst = NetworkTableInstance.getDefault();
        NetworkTable table = inst.getTable("datatable");
        DoubleSubscriber xSub = table.getDoubleTopic("x").subscribe(0.0);
        DoubleSubscriber ySub = table.getDoubleTopic("y").subscribe(0.0);
        inst.startClient4("example client");
        inst.setServerTeam(TEAM); // where TEAM=190, 294, etc, or use inst.setServer(
↪"hostname") or similar
        inst.startDSClient(); // recommended if running on DS computer; this gets the␣
↪robot IP from the DS
        while (true) {
```

(continues on next page)

(continued from previous page)

```
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                System.out.println("interrupted");
                return;
            }
            double x = xSub.get();
            double y = ySub.get();
            System.out.println("X: " + x + " Y: " + y);
        }
    }
}
```

C++

```
#include <chrono>
#include <thread>
#include <fmt/format.h>
#include <networktables/NetworkTableInstance.h>
#include <networktables/NetworkTable.h>
#include <networktables/DoubleTopic.h>
int main() {
  auto inst = nt::NetworkTableInstance::GetDefault();
  auto table = inst.GetTable("datatable");
  auto xSub = table->GetDoubleTopic("x").Subscribe(0.0);
  auto ySub = table->GetDoubleTopic("y").Subscribe(0.0);
  inst.StartClient4("example client");
  inst.SetServerTeam(TEAM);  // where TEAM=190, 294, etc, or use inst.setServer(
↪"hostname") or similar
  inst.StartDSClient();  // recommended if running on DS computer; this gets the␣
↪robot IP from the DS
  while (true) {
    using namespace std::chrono_literals;
    std::this_thread::sleep_for(1s);
    double x = xSub.Get();
    double y = ySub.Get();
    fmt::print("X: {} Y: {}\n", x, y);
  }
}
```

C++ (Handle-based)

```
#include <chrono>
#include <thread>
#include <fmt/format.h>
#include <ntcore_cpp.h>
int main() {
  NT_Inst inst = nt::GetDefaultInstance();
  NT_Subscriber xSub =
      nt::Subscribe(nt::GetTopic(inst, "/datatable/x"), NT_DOUBLE, "double");
  NT_Subscriber ySub =
      nt::Subscribe(nt::GetTopic(inst, "/datatable/y"), NT_DOUBLE, "double");
```

(continues on next page)

(continued from previous page)

```
  nt::StartClient4(inst, "example client");
  nt::SetServerTeam(inst, TEAM, 0);  // where TEAM=190, 294, etc, or use inst.
↪setServer("hostname") or similar
  nt::StartDSClient(inst, 0);  // recommended if running on DS computer; this gets␣
↪the robot IP from the DS
  while (true) {
    using namespace std::chrono_literals;
    std::this_thread::sleep_for(1s);
    double x = nt::GetDouble(xSub, 0.0);
    double y = nt::GetDouble(ySub, 0.0);
    fmt::print("X: {} Y: {}\n", x, y);
  }
}
```

C

```
#include <stdio.h>
#include <threads.h>
#include <time.h>
#include <networktables/ntcore.h>
int main() {
  NT_Instance inst = NT_GetDefaultInstance();
  NT_Subscriber xSub =
      NT_Subscribe(NT_GetTopic(inst, "/datatable/x"), NT_DOUBLE, "double", NULL, 0);
  NT_Subscriber ySub =
      NT_Subscribe(NT_GetTopic(inst, "/datatable/y"), NT_DOUBLE, "double", NULL, 0);
  NT_StartClient4(inst, "example client");
  NT_SetServerTeam(inst, TEAM);  // where TEAM=190, 294, etc, or use inst.setServer(
↪"hostname") or similar
  NT_StartDSClient(inst);  // recommended if running on DS computer; this gets the␣
↪robot IP from the DS
  while (true) {
    thrd_sleep(&(struct timespec){.tv_sec=1}, NULL);
    double x = NT_GetDouble(xSub, 0.0);
    double y = NT_GetDouble(ySub, 0.0);
    printf("X: %f Y: %f\n", x, y);
  }
}
```

Python

```
#!/usr/bin/env python3
import ntcore
import time
if __name__ == "__main__":
    inst = ntcore.NetworkTableInstance.getDefault()
    table = inst.getTable("datatable")
    xSub = table.getDoubleTopic("x").subscribe(0)
    ySub = table.getDoubleTopic("y").subscribe(0)
    inst.startClient4("example client")
    inst.setServerTeam(TEAM) # where TEAM=190, 294, etc, or use inst.setServer(
↪"hostname") or similar
```

(continues on next page)

(continued from previous page)

```
    inst.startDSClient() # recommended if running on DS computer; this gets the robot␣
↪IP from the DS
    while True:
        time.sleep(1)
        x = xSub.get()
        y = ySub.get()
        print(f"X: {x} Y: {y}")
```

In this example an instance of NetworkTables is created and subscribers are 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.

29.8.1 Building using Gradle

Example build.gradle files are provided in the StandaloneAppSamples Repository Update the GradleRIO version to correspond to the desired WPILib version.

Java

```
plugins {
    id "java"
    id 'application'
    id 'com.gradleup.shadow' version '8.3.5'
    id "edu.wpi.first.GradleRIO" version "2025.1.1"
    id 'edu.wpi.first.WpilibTools' version '2.1.0'
}

application {
    mainClass = 'Program'
}

wpilibTools.deps.wpilibVersion = wpi.versions.wpilibVersion.get()

def nativeConfigName = 'wpilibNatives'
def nativeConfig = configurations.create(nativeConfigName)

def nativeTasks = wpilibTools.createExtractionTasks {
    configurationName = nativeConfigName
}

nativeTasks.addToSourceSetResources(sourceSets.main)
nativeConfig.dependencies.add wpilibTools.deps.wpilib("wpimath")
nativeConfig.dependencies.add wpilibTools.deps.wpilib("wpinet")
nativeConfig.dependencies.add wpilibTools.deps.wpilib("wpiutil")
nativeConfig.dependencies.add wpilibTools.deps.wpilib("ntcore")
```

(continues on next page)

(continued from previous page)

```
nativeConfig.dependencies.add wpilibTools.deps.wpilib("cscore")
nativeConfig.dependencies.add wpilibTools.deps.wpilibOpenCv("frc" + wpi.frcYear.get(), wpi.versions.opencvVersion.get())

dependencies {
    implementation wpilibTools.deps.wpilibJava("wpiutil")
    implementation wpilibTools.deps.wpilibJava("wpimath")
    implementation wpilibTools.deps.wpilibJava("wpinet")
    implementation wpilibTools.deps.wpilibJava("ntcore")
    implementation wpilibTools.deps.wpilibJava("cscore")
    implementation wpilibTools.deps.wpilibJava("cameraserver")
    implementation wpilibTools.deps.wpilibOpenCvJava("frc" + wpi.frcYear.get(), wpi.versions.opencvVersion.get())

    implementation group: "com.fasterxml.jackson.core", name: "jackson-annotations", version: wpi.versions.jacksonVersion.get()
    implementation group: "com.fasterxml.jackson.core", name: "jackson-core", version: wpi.versions.jacksonVersion.get()
    implementation group: "com.fasterxml.jackson.core", name: "jackson-databind", version: wpi.versions.jacksonVersion.get()

    implementation group: "org.ejml", name: "ejml-simple", version: wpi.versions.ejmlVersion.get()
    implementation group: "us.hebi.quickbuf", name: "quickbuf-runtime", version: wpi.versions.quickbufVersion.get();
}

shadowJar {
    archiveBaseName = "TestApplication"
    archiveVersion = ""
    exclude("module-info.class")
    archiveClassifier.set(wpilibTools.currentPlatform.platformName)
}

wrapper {
    gradleVersion = '8.11'
}
```

C++

Uncomment the appropriate platform as highlighted.

```
plugins {
    id "cpp"
    id "edu.wpi.first.GradleRIO" version "2025.1.1"
}

// Disable local cache, as it won't have the cross artifact necessary
wpi.maven.useLocal = false

// Set to true to run simulation in debug mode
wpi.cpp.debugSimulation = false

def appName = "TestApplication"

```

(continues on next page)

(continued from previous page)

```
//nativeUtils.withCrossLinuxArm64() // Uncomment to build for Linux Arm 64. targetPlatform below also needs to be fixed
//nativeUtils.withCrossLinuxArm32() // Uncomment to build for Linux Arm 32. targetPlatform below also needs to be fixed

model {
    components {
        "${appName}"(NativeExecutableSpec) {
            targetPlatform wpi.platforms.desktop // Uncomment to build on whatever the native platform currently is
            //targetPlatform wpi.platforms.linuxarm64 // Uncomment to build for Linux Arm 64
            //targetPlatform wpi.platforms.linuxarm32 // Uncomment to build for Linux Arm 32

            sources.cpp {
                source {
                    srcDir 'src/main/cpp'
                    include '**/*.cpp', '**/*.cc'
                }
                exportedHeaders {
                    srcDir 'src/main/include'
                }
            }

            // Enable run tasks for this component
            wpi.cpp.enableExternalTasks(it)

            wpi.cpp.deps.wpilibStatic(it)
        }
    }
}

wrapper {
    gradleVersion = '8.11'
}
```

29.8.2 Building Python

For Python, refer to the *RobotPy install documentation*.

29.9 Migrating from NetworkTables 3.0 to NetworkTables 4.0

NetworkTables 4.0 (new for 2023) has a number of significant API breaking changes from NetworkTables 3.0, the version of NetworkTables used from 2016-2022.

29.9.1 NetworkTableEntry

While `NetworkTableEntry` can still be used (for backwards compatibility), users are encouraged to migrate to use of type-specific Publisher/Subscriber/Entry classes as appropriate, or if necessary, `GenericEntry` (see *Publishing and Subscribing to a Topic*. It's important to note that unlike `NetworkTableEntry`, these classes need to have appropriate lifetime management. Some functionality (e.g. persistent settings) has also moved to `Topic` properties (see *NetworkTables Tables and Topics*).

NT3 code (was):

JAVA

```
public class Example {
  final NetworkTableEntry yEntry;
  final NetworkTableEntry outEntry;
  public Example() {
    NetworkTableInstance inst = NetworkTableInstance.getDefault();
    // get the subtable called "datatable"
    NetworkTable datatable = inst.getTable("datatable");
    // get the entry in "datatable" called "Y"
    yEntry = datatable.getEntry("Y");
    // get the entry in "datatable" called "Out"
    outEntry = datatable.getEntry("Out");
  }
  public void periodic() {
    // read a double value from Y, and set Out to that value multiplied by 2
    double value = yEntry.getDouble(0.0);  // default to 0
    outEntry.setDouble(value * 2);
  }
}
```

C++

```
class Example {
  nt::NetworkTableEntry yEntry;
  nt::NetworkTableEntry outEntry;
 public:
  Example() {
    nt::NetworkTableInstance inst = nt::NetworkTableInstance::GetDefault();
    // get the subtable called "datatable"
    auto datatable = inst.GetTable("datatable");
    // get the entry in "datatable" called "Y"
    yEntry = datatable->GetEntry("Y");
    // get the entry in "datatable" called "Out"
    outEntry = datatable->GetEntry("Out");
  }
  void Periodic() {
    // read a double value from Y, and set Out to that value multiplied by 2
    double value = yEntry.GetDouble(0.0);  // default to 0
    outEntry.SetDouble(value * 2);
  }
};
```

PYTHON

```
class Example:
    def __init__(self):
        inst = ntcore.NetworkTableInstance.getDefault()
        # get the subtable called "datatable"
        datatable = inst.getTable("datatable")
        # get the entry in "datatable" called "Y"
        self.yEntry = datatable.getEntry("Y")
        # get the entry in "datatable" called "Out"
        self.outEntry = datatable.getEntry("Out")
    def periodic(self):
        # read a double value from Y, and set Out to that value multiplied by 2
        value = self.yEntry.getDouble(0.0)  # default to 0
        self.outEntry.setDouble(value * 2)
```

Recommended NT4 equivalent (should be):

JAVA

```
public class Example {
  final DoubleSubscriber ySub;
  final DoublePublisher outPub;
  public Example() {
    NetworkTableInstance inst = NetworkTableInstance.getDefault();
    // get the subtable called "datatable"
    NetworkTable datatable = inst.getTable("datatable");
    // subscribe to the topic in "datatable" called "Y"
    // default value is 0
    ySub = datatable.getDoubleTopic("Y").subscribe(0.0);
    // publish to the topic in "datatable" called "Out"
    outPub = datatable.getDoubleTopic("Out").publish();
  }
  public void periodic() {
    // read a double value from Y, and set Out to that value multiplied by 2
    double value = ySub.get();
    outPub.set(value * 2);
  }
  // often not required in robot code, unless this class doesn't exist for
  // the lifetime of the entire robot program, in which case close() needs to be
  // called to stop subscribing
  public void close() {
    ySub.close();
    outPub.close();
  }
}
```

C++

```
class Example {
  nt::DoubleSubscriber ySub;
  nt::DoublePublisher outPub;
 public:
  Example() {
    nt::NetworkTableInstance inst = nt::NetworkTableInstance::GetDefault();
    // get the subtable called "datatable"
    auto datatable = inst.GetTable("datatable");
    // subscribe to the topic in "datatable" called "Y"
    // default value is 0
    ySub = datatable->GetDoubleTopic("Y").Subscribe(0.0);
    // publish to the topic in "datatable" called "Out"
    outPub = datatable->GetDoubleTopic("Out").Publish();
  }
  void Periodic() {
    // read a double value from Y, and set Out to that value multiplied by 2
    double value = ySub.Get();
    outPub.Set(value * 2);
  }
};
```

PYTHON

```
class Example:
    def __init__(self) -> None:
        inst = ntcore.NetworkTableInstance.getDefault()
        # get the subtable called "datatable"
        datatable = inst.getTable("datatable")
        # subscribe to the topic in "datatable" called "Y"
        # default value is 0
        self.ySub = datatable.getDoubleTopic("Y").subscribe(0.0)
        # publish to the topic in "datatable" called "Out"
        self.outPub = datatable.getDoubleTopic("Out").publish()
    def periodic(self):
        # read a double value from Y, and set Out to that value multiplied by 2
        value = self.ySub.get()
        self.outPub.set(value * 2)
    # often not required in robot code, unless this class doesn't exist for
    # the lifetime of the entire robot program, in which case close() needs to be
    # called to stop subscribing
    def close(self):
        self.ySub.close()
        self.outPub.close()
```

29.9.2 Shuffleboard

In WPILib's Shuffleboard classes, usage of `NetworkTableEntry` has been replaced with use of `GenericEntry`. In C++, since `GenericEntry` is non-copyable, return values now return a reference rather than a value.

29.9.3 Force Set Operations

Force set operations have been removed, as it's no longer possible to change a topic's type once it's been published. In most cases calls to `forceSet` can simply be replaced with `set`, but more complex scenarios may require a different design approach (e.g. splitting into different topics).

29.9.4 Listeners

The separate connection, value, and log listeners/events have been unified into a single listener/event. The NetworkTable-level listeners have also been removed. Listeners in many cases can be replaced with subscriber `readQueue()` calls, but if listeners are still required, they can be used via `NetworkTableInstance` (see *Listening for Changes* for more information).

While NetworkTable-level listeners were removed, table-specific listening functionality can be achieved by using `NetworkTableInstance.addListener()` with a topic name prefix filter. By passing an array of topic name prefixes (such as `["/datatable/"]`), the listener will only receive events for topics within that table. This is demonstrated in the *Using NetworkTableInstance to Listen for Changes* section.

29.9.5 Client/Server Operations

Starting a NetworkTable server now requires specifying both the NT3 port and the NT4 port. For a NT4-only server, the NT3 port can be specified as 0.

A NetworkTable client can only operate in NT3 mode or NT4 mode, not both (there is no provision for automatic fallback). As such, the `startClient()` call has been replaced by `startClient3()` and `startClient4()`. The client must also specify a unique name for itself–the server will reject connection attempts with duplicate names.

29.9.6 C++ Changes

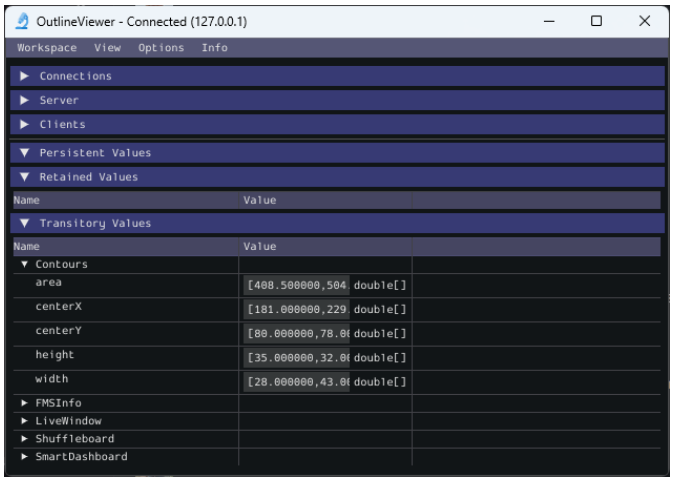
C++ values are now returned/used as value objects (plain `nt::Value`) instead of shared pointers to them (`std::shared_ptr<nt::Value>`).

29.10 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 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.

29.10.1 Verify the NetworkTables Topics Being Published



You can verify the names of the NetworkTables topics used for publishing the values by using the Outline Viewer application. It is a C++ 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 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 topic. The NetworkTables topic 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.

29.10.2 Writing a Program to Access the Topics

JAVA

```
DoubleArraySubscriber areasSub;

public Robot() {
  NetworkTable table = NetworkTableInstance.getDefault().getTable("GRIP/
↪mycontoursReport");
  areasSub = table.getDoubleArrayTopic("area").subscribe(new double[] {});
}

@Override
public void teleopPeriodic() {
  double[] areas = areasSub.get();
  System.out.print("areas: " );
  for (double area : areas) {
    System.out.print(area + " ");
  }
  System.out.println();
}
```

C++

```
nt::DoubleArraySubscriber areasSub;
Robot::Robot() {
  auto table = nt::NetworkTableInstance::GetDefault().GetTable("GRIP/myContoursReport
↪");
  areasSub = table->GetDoubleArrayTopic("area").Subscribe({});
}
void Robot::TeleopPeriodic() override {
  std::cout << "Areas: ";
  std::vector<double> arr = areasSub.Get();
  for (double val : arr) {
    std::cout << val << " ";
  }
  std::cout << std::endl;
}
```

PYTHON

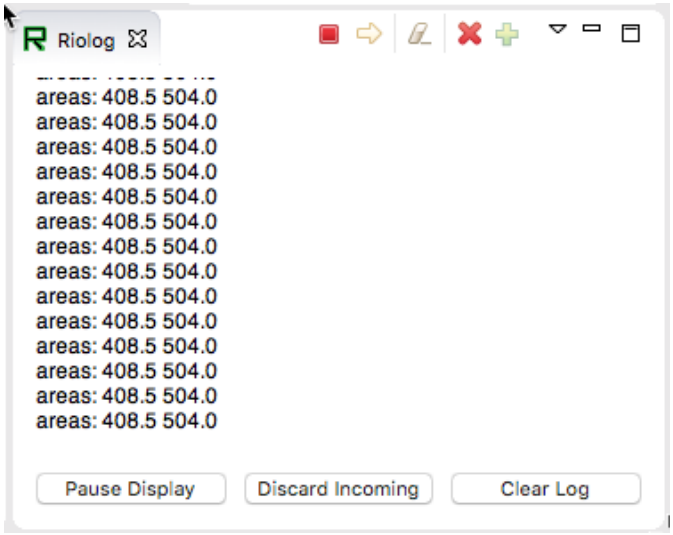
```
def robotInit(self):
    table = ntcore.NetworkTableInstance.getDefault().getTable("GRIP/mycontoursReport")
    self.areasSub = table.getDoubleArrayTopic("area").subscribe([])
def teleopPeriodic(self):
    areas = self.areasSub.get()
    print("Areas:", areas)
```

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 topic hasn't yet been published. This code will loop over the value of areas every 20ms.

29.10.3 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.

30 Path Planning

Path Planning is the process of creating and following trajectories. These paths use the WPILib trajectory APIs for generation and a *LTV Unicycle Controller* for following. This section highlights the process of characterizing your robot for system identification, trajectory following, and usage of PathWeaver. Users may also want to read the *generic trajectory following documents* for additional information about the API and non-commandbased usage.

30.1 Notice on Swerve Support

Swerve support in path following has a couple of limitations that teams need to be aware of:

- WPILib currently does not support swerve in simulation, please see this pull request.
- Pathweaver and Trajectory following currently do not incorporate independent heading. Path following using the WPILib trajectory framework on swerve will be the same as a DifferentialDrive robot. Both *Choreo* and PathPlanner have swerve support.

We are sorry for the inconvenience.

30.1.1 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++).

Trajectory Tutorial Overview

Warning

This tutorial uses the Ramsete Controller which has been *deprecated*. The replacement is *LTV Unicycle Controller* which has more intuitive tuning. We are looking for volunteers to update this tutorial.

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++).

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.

Required Equipment

To follow this tutorial, you will need ready access to the following materials:

1. A differential-drive robot (such as the AndyMark AM14U5), 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 System Identification Toolsuite*.

Step 1: Characterizing Your Robot Drive

Note

For detailed instructions on using the System Identification tool, see its *dedicated documentation*.

Note

The drive identification 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 identification routine.

Note

The identification 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 System Identification tool can accurately determine such a model.

Gathering the Data

We begin by gathering our drive identification data.

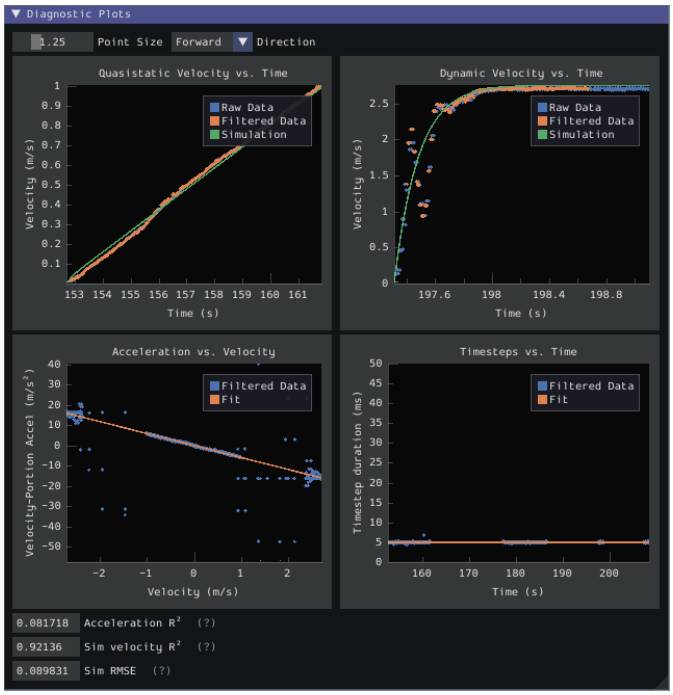
1. *Configure and Deploy your robot project*.
2. *Run the identification Routine*.

Analyzing the Data

Once the identification 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 *system identification guide*, we first view the diagnostics to ensure that our data look reasonable:

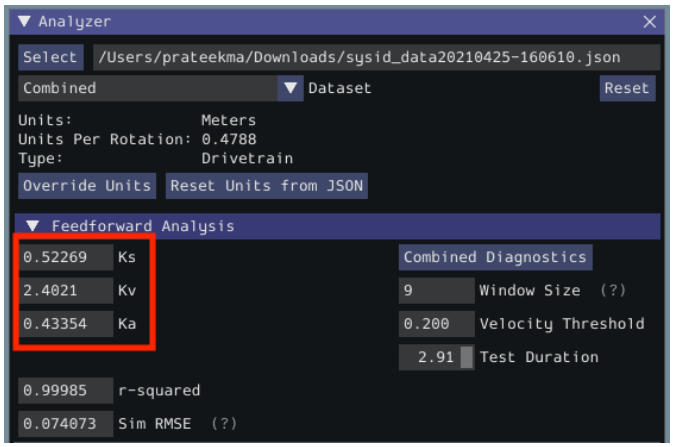


As our data look reasonably linear, and the fit metrics are within acceptable parameters, we proceed to the next step.

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:



Since our wheel diameter was specified in meters, our feedforward gains are in the following units:

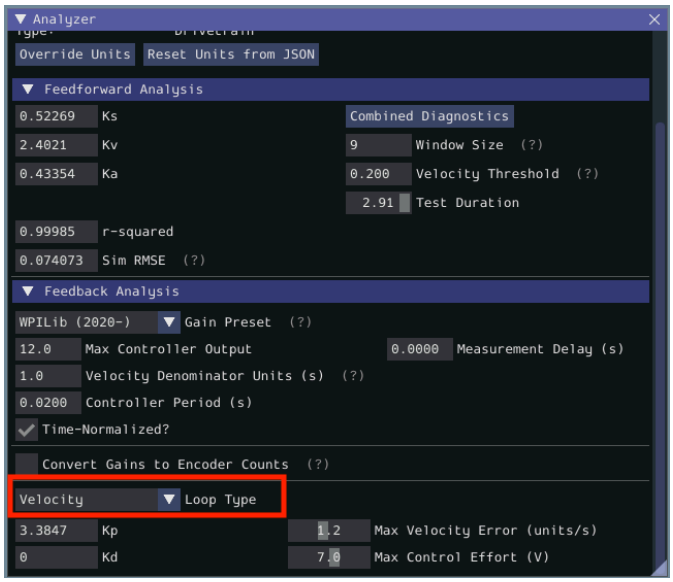
- kS: Volts
- kV: Volts * Seconds / Meters
- kA: Volts * Seconds^2 / Meters

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.

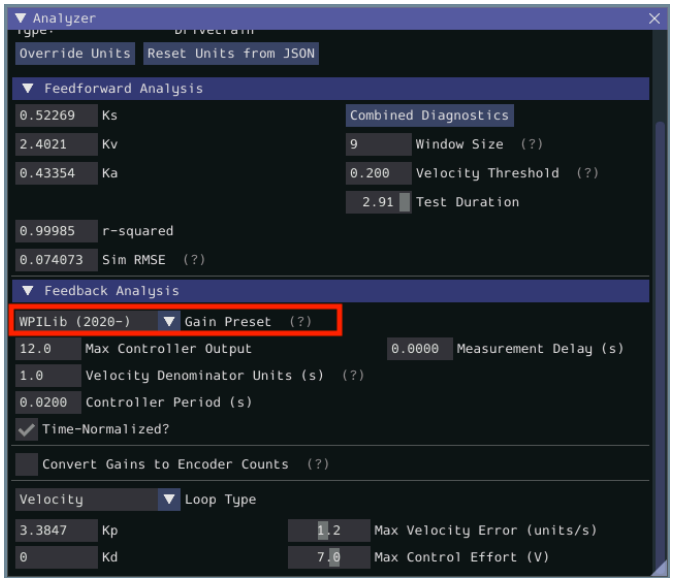
Calculate Feedback Gains

> **Note**
>
> Feedback gains do *not*, in general, transfer across robots. Do *not* use the gains from this tutorial for your own robot.

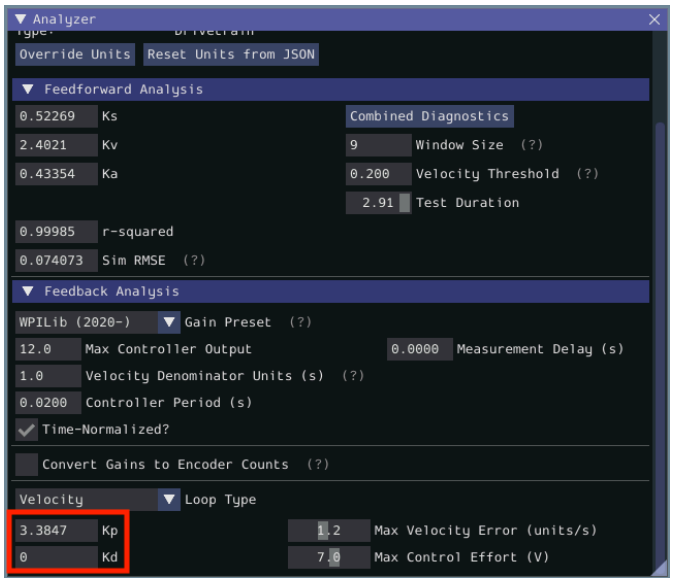
We now *calculate the feedback gains* for the PID control that we will use to follow the path. Trajectory following with WPILib's RAMSETE controller uses velocity closed-loop control, so we first select `Velocity` mode in the identification tool:



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:



Finally, we calculate and record the feedback gains for our control loop. Since it is a velocity controller, only a P gain is required:



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 3.38 volts.

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 system 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.

Feedforward/Feedback Gains

Firstly, we must enter the feedforward and feedback gains which we obtained from the identification 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

```
    // These are example values only - DO NOT USE THESE FOR YOUR OWN ROBOT!
    // These characterization values MUST be determined either experimentally or theoretically
    // for *your* robot's drive.
    // The Robot Characterization Toolsuite provides a convenient tool for obtaining these
    // values for your robot.
    public static final double ksVolts = 0.22;
    public static final double kvVoltSecondsPerMeter = 1.98;
    public static final double kaVoltSecondsSquaredPerMeter = 0.2;

    // Example value only - as above, this must be tuned for your drive!
    public static final double kPDriveVel = 8.5;
```

C++

```
// These are example values only - DO NOT USE THESE FOR YOUR OWN ROBOT!
// These characterization values MUST be determined either experimentally or
// theoretically for *your* robot's drive. The Robot Characterization
// Toolsuite provides a convenient tool for obtaining these values for your
// robot.
inline constexpr auto ks = 0.22_V;
inline constexpr auto kv = 1.98 * 1_V * 1_s / 1_m;
inline constexpr auto ka = 0.2 * 1_V * 1_s * 1_s / 1_m;

// Example value only - as above, this must be tuned for your drive!
inline constexpr double kPDriveVel = 8.5;
```

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

```
    public static final double kTrackwidthMeters = 0.69;
    public static final DifferentialDriveKinematics kDriveKinematics =
        new DifferentialDriveKinematics(kTrackwidthMeters);
```

C++

```
38 inline constexpr auto kTrackwidth = 0.69_m;
39 extern const frc::DifferentialDriveKinematics kDriveKinematics;
```

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.

> **Warning**
>
> Max velocity and acceleration, as defined here, are applied only during trajectory generation. They do not limit the `RamseteCommand` itself, which may give values to the `DriveSubsystem` that can cause the robot to greatly exceed these velocities and accelerations.

JAVA

```
57     public static final double kMaxSpeedMetersPerSecond = 3;
58     public static final double kMaxAccelerationMetersPerSecondSquared = 1;
```

C++

```
61 inline constexpr auto kMaxSpeed = 3_mps;
62 inline constexpr auto kMaxAcceleration = 1_mps_sq;
```

Ramsete Parameters

Finally, we must include a pair of parameters for the RAMSETE controller. The values `b` and `zeta` shown below should work well for most robots, provided distances have been correctly measured in meters.

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

```
60      // Reasonable baseline values for a RAMSETE follower in units of meters and␣
    ↪seconds
61      public static final double kRamseteB = 2;
62      public static final double kRamseteZeta = 0.7;
```

C++

```
64  // Reasonable baseline values for a RAMSETE follower in units of meters and
65  // seconds
66  inline constexpr auto kRamseteB = 2.0 * 1_rad * 1_rad / (1_m * 1_m);
67  inline constexpr auto kRamseteZeta = 0.7 / 1_rad;
```

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

```
5   package edu.wpi.first.wpilibj.examples.ramsetecommand.subsystems;
6
7   import edu.wpi.first.math.geometry.Pose2d;
8   import edu.wpi.first.math.kinematics.DifferentialDriveOdometry;
9   import edu.wpi.first.math.kinematics.DifferentialDriveWheelSpeeds;
10  import edu.wpi.first.util.sendable.SendableRegistry;
11  import edu.wpi.first.wpilibj.ADXRS450_Gyro;
12  import edu.wpi.first.wpilibj.Encoder;
13  import edu.wpi.first.wpilibj.drive.DifferentialDrive;
14  import edu.wpi.first.wpilibj.examples.ramsetecommand.Constants.DriveConstants;
15  import edu.wpi.first.wpilibj.motorcontrol.PWMSparkMax;
16  import edu.wpi.first.wpilibj2.command.SubsystemBase;
17
18  public class DriveSubsystem extends SubsystemBase {
19    // The motors on the left side of the drive.
20    private final PWMSparkMax m_leftLeader = new PWMSparkMax(DriveConstants.
    ↪kLeftMotor1Port);
21    private final PWMSparkMax m_leftFollower = new PWMSparkMax(DriveConstants.
    ↪kLeftMotor2Port);
22
23    // The motors on the right side of the drive.
24    private final PWMSparkMax m_rightLeader = new PWMSparkMax(DriveConstants.
    ↪kRightMotor1Port);
25    private final PWMSparkMax m_rightFollower = new PWMSparkMax(DriveConstants.
    ↪kRightMotor2Port);
26
```

(continues on next page)

(continued from previous page)

```
  // The robot's drive
  private final DifferentialDrive m_drive =
      new DifferentialDrive(m_leftLeader::set, m_rightLeader::set);

  // The left-side drive encoder
  private final Encoder m_leftEncoder =
      new Encoder(
          DriveConstants.kLeftEncoderPorts[0],
          DriveConstants.kLeftEncoderPorts[1],
          DriveConstants.kLeftEncoderReversed);

  // The right-side drive encoder
  private final Encoder m_rightEncoder =
      new Encoder(
          DriveConstants.kRightEncoderPorts[0],
          DriveConstants.kRightEncoderPorts[1],
          DriveConstants.kRightEncoderReversed);

  // The gyro sensor
  private final ADXRS450_Gyro m_gyro = new ADXRS450_Gyro();

  // Odometry class for tracking robot pose
  private final DifferentialDriveOdometry m_odometry;

  /** Creates a new DriveSubsystem. */
  public DriveSubsystem() {
    SendableRegistry.addChild(m_drive, m_leftLeader);
    SendableRegistry.addChild(m_drive, m_rightLeader);

    m_leftLeader.addFollower(m_leftFollower);
    m_rightLeader.addFollower(m_rightFollower);

    // We need to invert one side of the drivetrain so that positive voltages
    // result in both sides moving forward. Depending on how your robot's
    // gearbox is constructed, you might have to invert the left side instead.
    m_rightLeader.setInverted(true);

    // Sets the distance per pulse for the encoders
    m_leftEncoder.setDistancePerPulse(DriveConstants.kEncoderDistancePerPulse);
    m_rightEncoder.setDistancePerPulse(DriveConstants.kEncoderDistancePerPulse);

    resetEncoders();
    m_odometry =
        new DifferentialDriveOdometry(
            m_gyro.getRotation2d(), m_leftEncoder.getDistance(), m_rightEncoder.
↪getDistance());
  }

  @Override
  public void periodic() {
    // Update the odometry in the periodic block
    m_odometry.update(
        m_gyro.getRotation2d(), m_leftEncoder.getDistance(), m_rightEncoder.
↪getDistance());
  }

  /**
```

(continues on next page)

(continued from previous page)

```
   * Returns the currently-estimated pose of the robot.
   *
   * @return The pose.
   */
  public Pose2d getPose() {
    return m_odometry.getPoseMeters();
  }

  /**
   * Returns the current wheel speeds of the robot.
   *
   * @return The current wheel speeds.
   */
  public DifferentialDriveWheelSpeeds getWheelSpeeds() {
    return new DifferentialDriveWheelSpeeds(m_leftEncoder.getRate(), m_rightEncoder.
↪getRate());
  }

  /**
   * Resets the odometry to the specified pose.
   *
   * @param pose The pose to which to set the odometry.
   */
  public void resetOdometry(Pose2d pose) {
    m_odometry.resetPosition(
        m_gyro.getRotation2d(), m_leftEncoder.getDistance(), m_rightEncoder.
↪getDistance(), pose);
  }

  /**
   * Drives the robot using arcade controls.
   *
   * @param fwd the commanded forward movement
   * @param rot the commanded rotation
   */
  public void arcadeDrive(double fwd, double rot) {
    m_drive.arcadeDrive(fwd, rot);
  }

  /**
   * Controls the left and right sides of the drive directly with voltages.
   *
   * @param leftVolts the commanded left output
   * @param rightVolts the commanded right output
   */
  public void tankDriveVolts(double leftVolts, double rightVolts) {
    m_leftLeader.setVoltage(leftVolts);
    m_rightLeader.setVoltage(rightVolts);
    m_drive.feed();
  }

  /** Resets the drive encoders to currently read a position of 0. */
  public void resetEncoders() {
    m_leftEncoder.reset();
    m_rightEncoder.reset();
  }

```

(continues on next page)

(continued from previous page)

```
  /**
   * Gets the average distance of the two encoders.
   *
   * @return the average of the two encoder readings
   */
  public double getAverageEncoderDistance() {
    return (m_leftEncoder.getDistance() + m_rightEncoder.getDistance()) / 2.0;
  }

  /**
   * Gets the left drive encoder.
   *
   * @return the left drive encoder
   */
  public Encoder getLeftEncoder() {
    return m_leftEncoder;
  }

  /**
   * Gets the right drive encoder.
   *
   * @return the right drive encoder
   */
  public Encoder getRightEncoder() {
    return m_rightEncoder;
  }

  /**
   * Sets the max output of the drive. Useful for scaling the drive to drive more␣
↪slowly.
   *
   * @param maxOutput the maximum output to which the drive will be constrained
   */
  public void setMaxOutput(double maxOutput) {
    m_drive.setMaxOutput(maxOutput);
  }

  /** Zeroes the heading of the robot. */
  public void zeroHeading() {
    m_gyro.reset();
  }

  /**
   * Returns the heading of the robot.
   *
   * @return the robot's heading in degrees, from -180 to 180
   */
  public double getHeading() {
    return m_gyro.getRotation2d().getDegrees();
  }

  /**
   * Returns the turn rate of the robot.
   *
   * @return The turn rate of the robot, in degrees per second
   */
  public double getTurnRate() {
```

(continues on next page)

(continued from previous page)

```
    return -m_gyro.getRate();
  }
}
```

C++ (Header)

```
#pragma once

#include <frc/ADXRS450_Gyro.h>
#include <frc/Encoder.h>
#include <frc/drive/DifferentialDrive.h>
#include <frc/geometry/Pose2d.h>
#include <frc/kinematics/DifferentialDriveOdometry.h>
#include <frc/motorcontrol/PWMSparkMax.h>
#include <frc2/command/SubsystemBase.h>
#include <units/voltage.h>

#include "Constants.h"

class DriveSubsystem : public frc2::SubsystemBase {
 public:
  DriveSubsystem();

  /**
   * Will be called periodically whenever the CommandScheduler runs.
   */
  void Periodic() override;

  // Subsystem methods go here.

  /**
   * Drives the robot using arcade controls.
   *
   * @param fwd the commanded forward movement
   * @param rot the commanded rotation
   */
  void ArcadeDrive(double fwd, double rot);

  /**
   * Controls each side of the drive directly with a voltage.
   *
   * @param left the commanded left output
   * @param right the commanded right output
   */
  void TankDriveVolts(units::volt_t left, units::volt_t right);

  /**
   * Resets the drive encoders to currently read a position of 0.
   */
  void ResetEncoders();

  /**
   * Gets the average distance of the TWO encoders.
   *
```

(continues on next page)

(continued from previous page)

```
   * @return the average of the TWO encoder readings
   */
  double GetAverageEncoderDistance();

  /**
   * Gets the left drive encoder.
   *
   * @return the left drive encoder
   */
  frc::Encoder& GetLeftEncoder();

  /**
   * Gets the right drive encoder.
   *
   * @return the right drive encoder
   */
  frc::Encoder& GetRightEncoder();

  /**
   * Sets the max output of the drive.  Useful for scaling the drive to drive
   * more slowly.
   *
   * @param maxOutput the maximum output to which the drive will be constrained
   */
  void SetMaxOutput(double maxOutput);

  /**
   * Returns the heading of the robot.
   *
   * @return the robot's heading in degrees, from -180 to 180
   */
  units::degree_t GetHeading() const;

  /**
   * Returns the turn rate of the robot.
   *
   * @return The turn rate of the robot, in degrees per second
   */
  double GetTurnRate();

  /**
   * Returns the currently-estimated pose of the robot.
   *
   * @return The pose.
   */
  frc::Pose2d GetPose();

  /**
   * Returns the current wheel speeds of the robot.
   *
   * @return The current wheel speeds.
   */
  frc::DifferentialDriveWheelSpeeds GetWheelSpeeds();

  /**
   * Resets the odometry to the specified pose.
   *
```

(continues on next page)

(continued from previous page)

```
   * @param pose The pose to which to set the odometry.
   */
  void ResetOdometry(frc::Pose2d pose);

 private:
  // Components (e.g. motor controllers and sensors) should generally be
  // declared private and exposed only through public methods.

  // The motor controllers
  frc::PWMSparkMax m_left1;
  frc::PWMSparkMax m_left2;
  frc::PWMSparkMax m_right1;
  frc::PWMSparkMax m_right2;

  // The robot's drive
  frc::DifferentialDrive m_drive{[&](double output) { m_left1.Set(output); },
                                 [&](double output) { m_right1.Set(output); }};

  // The left-side drive encoder
  frc::Encoder m_leftEncoder;

  // The right-side drive encoder
  frc::Encoder m_rightEncoder;

  // The gyro sensor
  frc::ADXRS450_Gyro m_gyro;

  // Odometry class for tracking robot pose
  frc::DifferentialDriveOdometry m_odometry;
};
```

C++ (Source)

```
#include "subsystems/DriveSubsystem.h"

#include <frc/geometry/Rotation2d.h>
#include <frc/kinematics/DifferentialDriveWheelSpeeds.h>

using namespace DriveConstants;

DriveSubsystem::DriveSubsystem()
    : m_left1{kLeftMotor1Port},
      m_left2{kLeftMotor2Port},
      m_right1{kRightMotor1Port},
      m_right2{kRightMotor2Port},
      m_leftEncoder{kLeftEncoderPorts[0], kLeftEncoderPorts[1]},
      m_rightEncoder{kRightEncoderPorts[0], kRightEncoderPorts[1]},
      m_odometry{m_gyro.GetRotation2d(), units::meter_t{0}, units::meter_t{0}} {
  wpi::SendableRegistry::AddChild(&m_drive, &m_left1);
  wpi::SendableRegistry::AddChild(&m_drive, &m_right1);

  m_left1.AddFollower(m_left2);
  m_right1.AddFollower(m_right2);

```

(continues on next page)

(continued from previous page)

```
  // We need to invert one side of the drivetrain so that positive voltages
  // result in both sides moving forward. Depending on how your robot's
  // gearbox is constructed, you might have to invert the left side instead.
  m_right1.SetInverted(true);

  // Set the distance per pulse for the encoders
  m_leftEncoder.SetDistancePerPulse(kEncoderDistancePerPulse.value());
  m_rightEncoder.SetDistancePerPulse(kEncoderDistancePerPulse.value());

  ResetEncoders();
}

void DriveSubsystem::Periodic() {
  // Implementation of subsystem periodic method goes here.
  m_odometry.Update(m_gyro.GetRotation2d(),
                    units::meter_t{m_leftEncoder.GetDistance()},
                    units::meter_t{m_rightEncoder.GetDistance()});
}

void DriveSubsystem::ArcadeDrive(double fwd, double rot) {
  m_drive.ArcadeDrive(fwd, rot);
}

void DriveSubsystem::TankDriveVolts(units::volt_t left, units::volt_t right) {
  m_left1.SetVoltage(left);
  m_right1.SetVoltage(right);
  m_drive.Feed();
}

void DriveSubsystem::ResetEncoders() {
  m_leftEncoder.Reset();
  m_rightEncoder.Reset();
}

double DriveSubsystem::GetAverageEncoderDistance() {
  return (m_leftEncoder.GetDistance() + m_rightEncoder.GetDistance()) / 2.0;
}

frc::Encoder& DriveSubsystem::GetLeftEncoder() {
  return m_leftEncoder;
}

frc::Encoder& DriveSubsystem::GetRightEncoder() {
  return m_rightEncoder;
}

void DriveSubsystem::SetMaxOutput(double maxOutput) {
  m_drive.SetMaxOutput(maxOutput);
}

units::degree_t DriveSubsystem::GetHeading() const {
  return m_gyro.GetRotation2d().Degrees();
}

double DriveSubsystem::GetTurnRate() {
  return -m_gyro.GetRate();
}
```

(continues on next page)

(continued from previous page)

```

frc::Pose2d DriveSubsystem::GetPose() {
  return m_odometry.GetPose();
}

frc::DifferentialDriveWheelSpeeds DriveSubsystem::GetWheelSpeeds() {
  return {units::meters_per_second_t{m_leftEncoder.GetRate()},
          units::meters_per_second_t{m_rightEncoder.GetRate()}};
}

void DriveSubsystem::ResetOdometry(frc::Pose2d pose) {
  m_odometry.ResetPosition(m_gyro.GetRotation2d(),
                           units::meter_t{m_leftEncoder.GetDistance()},
                           units::meter_t{m_rightEncoder.GetDistance()}, pose);
}
```

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

```
  // The left-side drive encoder
  private final Encoder m_leftEncoder =
      new Encoder(
          DriveConstants.kLeftEncoderPorts[0],
          DriveConstants.kLeftEncoderPorts[1],
          DriveConstants.kLeftEncoderReversed);

  // The right-side drive encoder
  private final Encoder m_rightEncoder =
      new Encoder(
          DriveConstants.kRightEncoderPorts[0],
          DriveConstants.kRightEncoderPorts[1],
          DriveConstants.kRightEncoderReversed);
```

C++ (Source)

```
      m_leftEncoder{kLeftEncoderPorts[0], kLeftEncoderPorts[1]},
      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 SysId config. Remember, as well, that the distance should be measured in meters!

Java

```
    m_leftEncoder.setDistancePerPulse(DriveConstants.kEncoderDistancePerPulse);
    m_rightEncoder.setDistancePerPulse(DriveConstants.kEncoderDistancePerPulse);
```

C++ (Source)

```
  m_leftEncoder.SetDistancePerPulse(kEncoderDistancePerPulse.value());
  m_rightEncoder.SetDistancePerPulse(kEncoderDistancePerPulse.value());
```

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

```
  /**
   * Returns the current wheel speeds of the robot.
   *
   * @return The current wheel speeds.
   */
  public DifferentialDriveWheelSpeeds getWheelSpeeds() {
    return new DifferentialDriveWheelSpeeds(m_leftEncoder.getRate(), m_rightEncoder.
↪getRate());
  }
```

C++ (Source)

```
frc::DifferentialDriveWheelSpeeds DriveSubsystem::GetWheelSpeeds() {
  return {units::meters_per_second_t{m_leftEncoder.GetRate()},
          units::meters_per_second_t{m_rightEncoder.GetRate()}};
}
```

We wrap the measured encoder values in a `DifferentialDriveWheelSpeeds` object for easier integration with the `RamseteCommand` class later on.

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 was included in the kit of parts for several years:

Java

```
  // The gyro sensor
  private final ADXRS450_Gyro m_gyro = new ADXRS450_Gyro();
```

C++ (Header)

```
  // The gyro sensor
  frc::ADXRS450_Gyro m_gyro;
```

Gyroscope Accessor Method

To access the current heading measured by the gyroscope, we include the following method:

Java

```
  /**
   * Returns the heading of the robot.
   *
   * @return the robot's heading in degrees, from -180 to 180
   */
  public double getHeading() {
    return m_gyro.getRotation2d().getDegrees();
  }
```

C++ (Source)

```
76 units::degree_t DriveSubsystem::GetHeading() const {
77   return m_gyro.GetRotation2d().Degrees();
78 }
```

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

```
48   // Odometry class for tracking robot pose
49   private final DifferentialDriveOdometry m_odometry;
```

C++ (Header)

```
137   // Odometry class for tracking robot pose
138   frc::DifferentialDriveOdometry m_odometry;
```

Then we initialize the `DifferentialDriveOdometry`.

Java

```
69     m_odometry =
70         new DifferentialDriveOdometry(
71             m_gyro.getRotation2d(), m_leftEncoder.getDistance(), m_rightEncoder.
↪getDistance());
```

C++ (Source)

```
19       m_odometry{m_gyro.GetRotation2d(), units::meter_t{0}, units::meter_t{0}} {
```

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

```
  @Override
  public void periodic() {
    // Update the odometry in the periodic block
    m_odometry.update(
        m_gyro.getRotation2d(), m_leftEncoder.getDistance(), m_rightEncoder.
↪getDistance());
  }
```

C++ (Source)

```
void DriveSubsystem::Periodic() {
  // Implementation of subsystem periodic method goes here.
  m_odometry.Update(m_gyro.GetRotation2d(),
                    units::meter_t{m_leftEncoder.GetDistance()},
                    units::meter_t{m_rightEncoder.GetDistance()});
}
```

Odometry Accessor Method

To access the robot's current computed pose, we include the following method:

Java

```
  /**
   * Returns the currently-estimated pose of the robot.
   *
   * @return The pose.
   */
  public Pose2d getPose() {
    return m_odometry.getPoseMeters();
  }
```

C++ (Source)

```
frc::Pose2d DriveSubsystem::GetPose() {
  return m_odometry.GetPose();
}
```

Important

Before running a `RamseteCommand`, teams are strongly encouraged to deploy and test the odometry code alone, with values sent to the SmartDashboard or Shuffleboard during the `DriveSubsystem`'s `periodic()`. This odometry must be correct for a RamseteCommand to successfully work, as sign or unit errors can cause a robot to move at high speeds in unpredictable directions.

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 `MotorController` interface. The default WPILib drive class does not include this functionality, so we must write it ourselves:

Java

```
119   /**
120    * Controls the left and right sides of the drive directly with voltages.
121    *
122    * @param leftVolts the commanded left output
123    * @param rightVolts the commanded right output
124    */
125   public void tankDriveVolts(double leftVolts, double rightVolts) {
126     m_leftLeader.setVoltage(leftVolts);
127     m_rightLeader.setVoltage(rightVolts);
128     m_drive.feed();
129   }
```

C++ (Source)

```
49  void DriveSubsystem::TankDriveVolts(units::volt_t left, units::volt_t right) {
50    m_left1.SetVoltage(left);
51    m_right1.SetVoltage(right);
52    m_drive.Feed();
53  }
```

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 identification data), this is essential to ensuring their accuracy.

Warning

`RamseteCommand` itself does not internally enforce any speed or acceleration limits before providing motor voltage parameters to this method. During initial code development, teams are strongly encouraged to apply both maximum and minimum bounds on the input variables before passing these values to `setVoltage()` while ensuring the trajectory velocity and acceleration are achievable. For example, generate a trajectory with a little less than half of the Robot's maximum velocity and limit voltage to 6 volts.

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

```
  /**
   * Use this to pass the autonomous command to the main {@link Robot} class.
   *
   * @return the command to run in autonomous
   */
  public Command getAutonomousCommand() {
    // Create a voltage constraint to ensure we don't accelerate too fast
    var autoVoltageConstraint =
        new DifferentialDriveVoltageConstraint(
            new SimpleMotorFeedforward(
                DriveConstants.ksVolts,
                DriveConstants.kvVoltSecondsPerMeter,
                DriveConstants.kaVoltSecondsSquaredPerMeter),
            DriveConstants.kDriveKinematics,
            10);

    // Create config for trajectory
    TrajectoryConfig config =
        new TrajectoryConfig(
                AutoConstants.kMaxSpeedMetersPerSecond,
                AutoConstants.kMaxAccelerationMetersPerSecondSquared)
            // Add kinematics to ensure max speed is actually obeyed
            .setKinematics(DriveConstants.kDriveKinematics)
            // Apply the voltage constraint
            .addConstraint(autoVoltageConstraint);

    // An example trajectory to follow. All units in meters.
    Trajectory exampleTrajectory =
        TrajectoryGenerator.generateTrajectory(
            // Start at the origin facing the +X direction
            new Pose2d(0, 0, new Rotation2d(0)),
            // Pass through these two interior waypoints, making an 's' curve path
            List.of(new Translation2d(1, 1), new Translation2d(2, -1)),
            // End 3 meters straight ahead of where we started, facing forward
            new Pose2d(3, 0, new Rotation2d(0)),
            // Pass config
            config);

    RamseteCommand ramseteCommand =
        new RamseteCommand(
            exampleTrajectory,
            m_robotDrive::getPose,
            new RamseteController(AutoConstants.kRamseteB, AutoConstants.
↪kRamseteZeta),
```

(continues on next page)

(continued from previous page)

```
            new SimpleMotorFeedforward(
                DriveConstants.ksVolts,
                DriveConstants.kvVoltSecondsPerMeter,
                DriveConstants.kaVoltSecondsSquaredPerMeter),
            DriveConstants.kDriveKinematics,
            m_robotDrive::getWheelSpeeds,
            new PIDController(DriveConstants.kPDriveVel, 0, 0),
            new PIDController(DriveConstants.kPDriveVel, 0, 0),
            // RamseteCommand passes volts to the callback
            m_robotDrive::tankDriveVolts,
            m_robotDrive);

    // Reset odometry to the initial pose of the trajectory, run path following
    // command, then stop at the end.
    return Commands.runOnce(() -> m_robotDrive.resetOdometry(exampleTrajectory.
↪getInitialPose()))
        .andThen(ramseteCommand)
        .andThen(Commands.runOnce(() -> m_robotDrive.tankDriveVolts(0, 0)));
  }
}
```

C++ (Source)

```
frc2::CommandPtr RobotContainer::GetAutonomousCommand() {
  // Create a voltage constraint to ensure we don't accelerate too fast
  frc::DifferentialDriveVoltageConstraint autoVoltageConstraint{
      frc::SimpleMotorFeedforward<units::meters>{
          DriveConstants::ks, DriveConstants::kv, DriveConstants::ka},
      DriveConstants::kDriveKinematics, 10_V};

  // Set up config for trajectory
  frc::TrajectoryConfig config{AutoConstants::kMaxSpeed,
                               AutoConstants::kMaxAcceleration};
  // Add kinematics to ensure max speed is actually obeyed
  config.SetKinematics(DriveConstants::kDriveKinematics);
  // Apply the voltage constraint
  config.AddConstraint(autoVoltageConstraint);

  // An example trajectory to follow.  All units in meters.
  auto exampleTrajectory = frc::TrajectoryGenerator::GenerateTrajectory(
      // Start at the origin facing the +X direction
      frc::Pose2d{0_m, 0_m, 0_deg},
      // Pass through these two interior waypoints, making an 's' curve path
      {frc::Translation2d{1_m, 1_m}, frc::Translation2d{2_m, -1_m}},
      // End 3 meters straight ahead of where we started, facing forward
      frc::Pose2d{3_m, 0_m, 0_deg},
      // Pass the config
      config);

  frc2::CommandPtr ramseteCommand{frc2::RamseteCommand(
      exampleTrajectory, [this] { return m_drive.GetPose(); },
      frc::RamseteController{AutoConstants::kRamseteB,
                             AutoConstants::kRamseteZeta},
      frc::SimpleMotorFeedforward<units::meters>{
```

(continues on next page)

(continued from previous page)

```
          DriveConstants::ks, DriveConstants::kv, DriveConstants::ka},
      DriveConstants::kDriveKinematics,
      [this] { return m_drive.GetWheelSpeeds(); },
      frc::PIDController{DriveConstants::kPDriveVel, 0, 0},
      frc::PIDController{DriveConstants::kPDriveVel, 0, 0},
      [this](auto left, auto right) { m_drive.TankDriveVolts(left, right); },
      {&m_drive})};

  // Reset odometry to the initial pose of the trajectory, run path following
  // command, then stop at the end.
  return frc2::cmd::RunOnce(
             [this, initialPose = exampleTrajectory.InitialPose()] {
               m_drive.ResetOdometry(initialPose);
             },
             {})
      .AndThen(std::move(ramseteCommand))
      .AndThen(
          frc2::cmd::RunOnce([this] { m_drive.TankDriveVolts(0_V, 0_V); }, {}));
}
```

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

```
    // Create a voltage constraint to ensure we don't accelerate too fast
    var autoVoltageConstraint =
        new DifferentialDriveVoltageConstraint(
            new SimpleMotorFeedforward(
                DriveConstants.ksVolts,
                DriveConstants.kvVoltSecondsPerMeter,
                DriveConstants.kaVoltSecondsSquaredPerMeter),
            DriveConstants.kDriveKinematics,
            10);
```

C++ (Source)

```
// Create a voltage constraint to ensure we don't accelerate too fast
frc::DifferentialDriveVoltageConstraint autoVoltageConstraint{
    frc::SimpleMotorFeedforward<units::meters>{
        DriveConstants::ks, DriveConstants::kv, DriveConstants::ka},
    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

```
  // Create config for trajectory
  TrajectoryConfig config =
      new TrajectoryConfig(
              AutoConstants.kMaxSpeedMetersPerSecond,
              AutoConstants.kMaxAccelerationMetersPerSecondSquared)
          // Add kinematics to ensure max speed is actually obeyed
          .setKinematics(DriveConstants.kDriveKinematics)
          // Apply the voltage constraint
          .addConstraint(autoVoltageConstraint);
```

C++ (Source)

```
// Set up config for trajectory
frc::TrajectoryConfig config{AutoConstants::kMaxSpeed,
                             AutoConstants::kMaxAcceleration};
// Add kinematics to ensure max speed is actually obeyed
config.SetKinematics(DriveConstants::kDriveKinematics);
// Apply the voltage constraint
config.AddConstraint(autoVoltageConstraint);
```

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

```
    // An example trajectory to follow. All units in meters.
    Trajectory exampleTrajectory =
        TrajectoryGenerator.generateTrajectory(
            // Start at the origin facing the +X direction
            new Pose2d(0, 0, new Rotation2d(0)),
            // Pass through these two interior waypoints, making an 's' curve path
            List.of(new Translation2d(1, 1), new Translation2d(2, -1)),
            // End 3 meters straight ahead of where we started, facing forward
            new Pose2d(3, 0, new Rotation2d(0)),
            // Pass config
            config);
```

C++ (Source)

```
  // An example trajectory to follow.  All units in meters.
  auto exampleTrajectory = frc::TrajectoryGenerator::GenerateTrajectory(
      // Start at the origin facing the +X direction
      frc::Pose2d{0_m, 0_m, 0_deg},
      // Pass through these two interior waypoints, making an 's' curve path
      {frc::Translation2d{1_m, 1_m}, frc::Translation2d{2_m, -1_m}},
      // End 3 meters straight ahead of where we started, facing forward
      frc::Pose2d{3_m, 0_m, 0_deg},
      // Pass the config
      config);
```

Note

Instead of generating the trajectory on the roboRIO as outlined above, one can also *import a PathWeaver JSON*.

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

```
    // Reset odometry to the initial pose of the trajectory, run path following
    // command, then stop at the end.
    return Commands.runOnce(() -> m_robotDrive.resetOdometry(exampleTrajectory.
↪getInitialPose()))
```

C++ (Source)

```
  // Reset odometry to the initial pose of the trajectory, run path following
  // command, then stop at the end.
  return frc2::cmd::RunOnce(
```

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

```
    RamseteCommand ramseteCommand =
        new RamseteCommand(
            exampleTrajectory,
            m_robotDrive::getPose,
            new RamseteController(AutoConstants.kRamseteB, AutoConstants.
↪kRamseteZeta),
            new SimpleMotorFeedforward(
                DriveConstants.ksVolts,
                DriveConstants.kvVoltSecondsPerMeter,
                DriveConstants.kaVoltSecondsSquaredPerMeter),
            DriveConstants.kDriveKinematics,
            m_robotDrive::getWheelSpeeds,
            new PIDController(DriveConstants.kPDriveVel, 0, 0),
            new PIDController(DriveConstants.kPDriveVel, 0, 0),
            // RamseteCommand passes volts to the callback
            m_robotDrive::tankDriveVolts,
            m_robotDrive);
```

C++ (Source)

```
  frc2::CommandPtr ramseteCommand{frc2::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(); },
      frc::PIDController{DriveConstants::kPDriveVel, 0, 0},
      frc::PIDController{DriveConstants::kPDriveVel, 0, 0},
      [this](auto left, auto right) { m_drive.TankDriveVolts(left, right); },
      {&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 identification 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 identification 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 identification 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.

Video

If all has gone well, your robot’s autonomous routine should look something like this:

30.1.2 PathWeaver

> **Warning**
>
> PathWeaver is deprecated and will be removed for 2027. Users may find *Choreo* or PathPlanner more useful. They both have an intuitive user interface and swerve support.

Introduction to PathWeaver

> **Warning**
>
> PathWeaver is deprecated and will be removed for 2027. Users may find *Choreo* or PathPlanner more useful. They both have an intuitive user interface and swerve support.

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*.

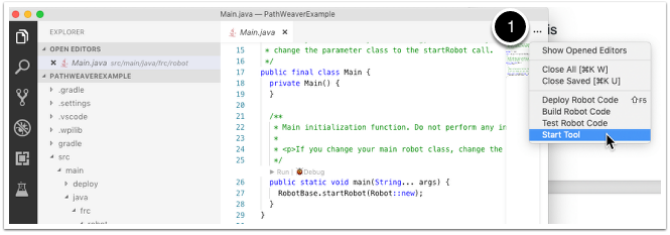
> **Note**
>
> *Trajectory following* code is required to use PathWeaver. We recommend that you start with Trajectory following and get that working with simple paths. From there you can continue on to testing more complicated paths generated by PathWeaver.

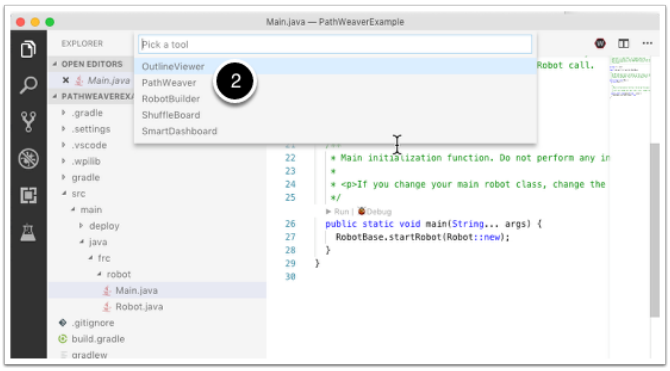
Creating a Pathweaver Project

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.

Starting PathWeaver

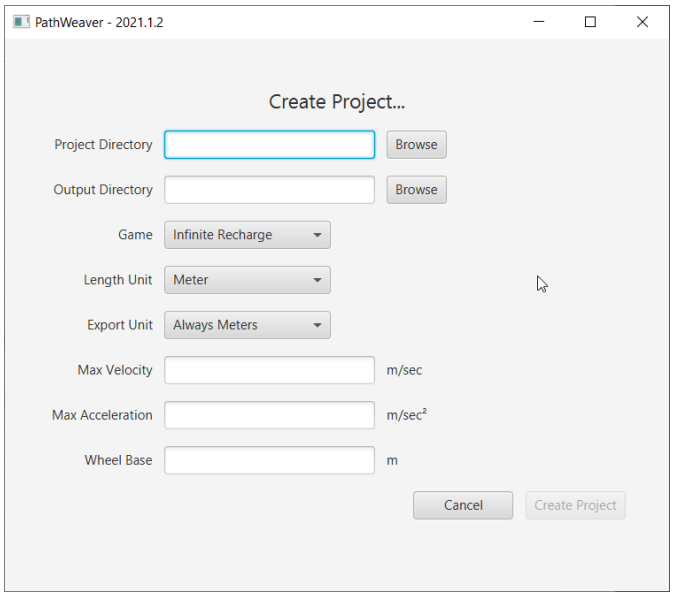
PathWeaver is started by clicking on the ellipsis icon in the top right of the corner of the Visual Studio Code interface. You must select a source file from the WPILib project to see the icon. Then click on "Start tool" and then click on "PathWeaver" as shown below.





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.



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

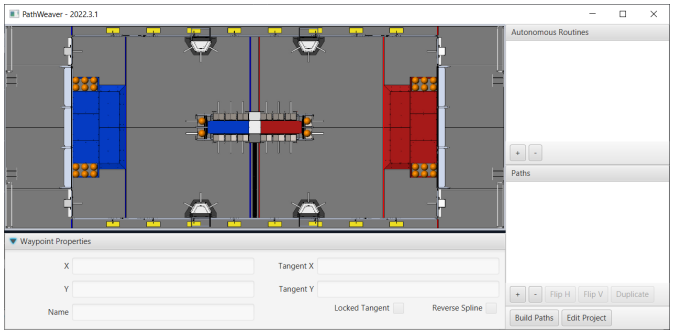
planning to use WPILib Trajectories, then you should choose *Always Meters*. Choosing *Same as Project* will export in the same units as *Length Unit* above.

Max Velocity: The max speed of the robot for trajectory tracking. The kitbot runs at ~10 ft/sec which is ~3 m/sec.

Max Acceleration: The max acceleration of the robot for trajectory tracking. Using a conservative 1 m/sec^2 is a good place to start if you don't know your drivetain's characteristics.

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.

PathWeaver User Interface

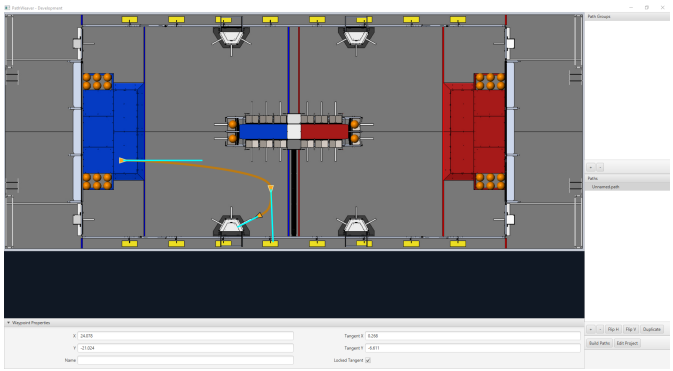


The PathWeaver user interface consists of the following:

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.

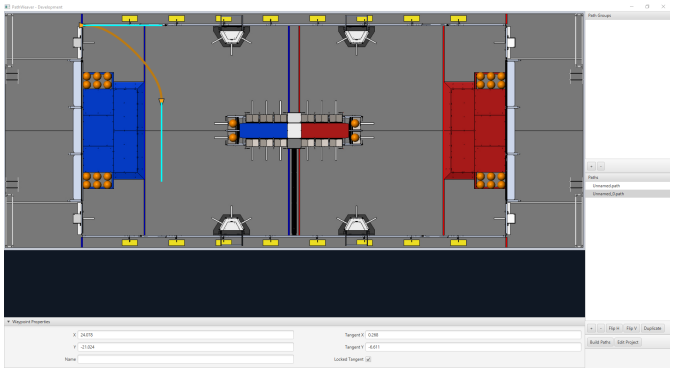
Visualizing PathWeaver Trajectories

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.



Creating the Initial Trajectory

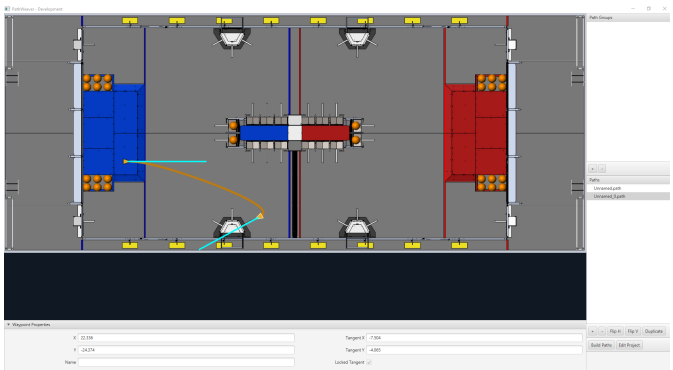
To start creating a trajectory, click the + (plus) button in the path window. A default trajectory will be created that probably does not have the proper start and end points that you desire. The path also shows the tangent vectors (teal lines) for the start and end points. Changing the angle of the 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.

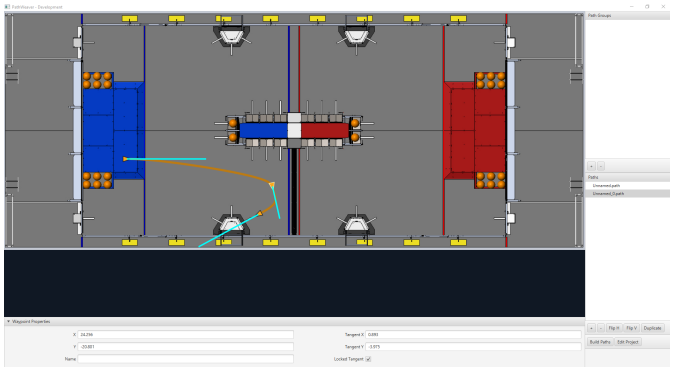
Changing a Waypoint Heading

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.



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.



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.

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.

Creating Autonomous Routines

Autonomous Routines (also known as 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 routine have common end and start points. By adding all the paths to a single autonomous routine and selecting the routine, all paths in that routine will be shown. Then each path can be edited while viewing all the paths.

Creating an Autonomous Routine

Press the + button underneath Autonomous Routines. Then drag the Paths from the Paths section into your Autonomous Routine.

Each path added to an autonomous routine 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 routine, selection works as follows:

1. Selecting the routine displays all paths in the routine making it easy to see the relationship between them. Any waypoint on any of the paths can be edited while the routine is selected and it will only change the path containing the waypoint.
2. Selecting on a single path in the routine 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.

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

```
String trajectoryJSON = "paths/YourPath.wpilib.json";
Trajectory trajectory = new Trajectory();

public Robot() {
   try {
      Path trajectoryPath = Filesystem.getDeployDirectory().toPath().
↪resolve(trajectoryJSON);
      trajectory = TrajectoryUtil.fromPathweaverJson(trajectoryPath);
   } catch (IOException ex) {
      DriverStation.reportError("Unable to open trajectory: " + trajectoryJSON, ex.
↪getStackTrace());
   }
}
```

C++

```
#include <frc/Filesystem.h>
#include <frc/trajectory/TrajectoryUtil.h>
#include <wpi/fs.h>
frc::Trajectory trajectory;
Robot::Robot()) {
   fs::path deployDirectory = frc::filesystem::GetDeployDirectory();
   deployDirectory = deployDirectory / "paths" / "YourPath.wpilib.json";
   trajectory = frc::TrajectoryUtil::FromPathweaverJson(deployDirectory.string());
}
```

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.

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 allwpilib GitHub repository.

File Layout

```
~/PathWeaver
  /Games
   /Custom Game
      custom-game.json
      field-image.png
   OtherGame.zip
```

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.

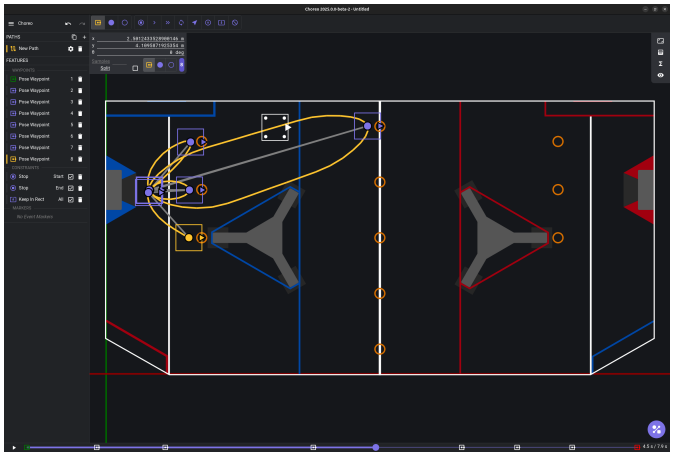
The field units are case-insensitive and can be in meters, cm, mm, inches, feet, yards, or miles. Singular, plural, and abbreviations are supported (e.g. “meter”,”meters”, and”m”are all valid for specifying meters)

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.

30.1.3 Choreo

Choreo is a tool for calculating time optimal autonomous trajectories. This allows simple and efficient creation of autonomous routines.



Note

Detailed documentation for Choreo can be found here.

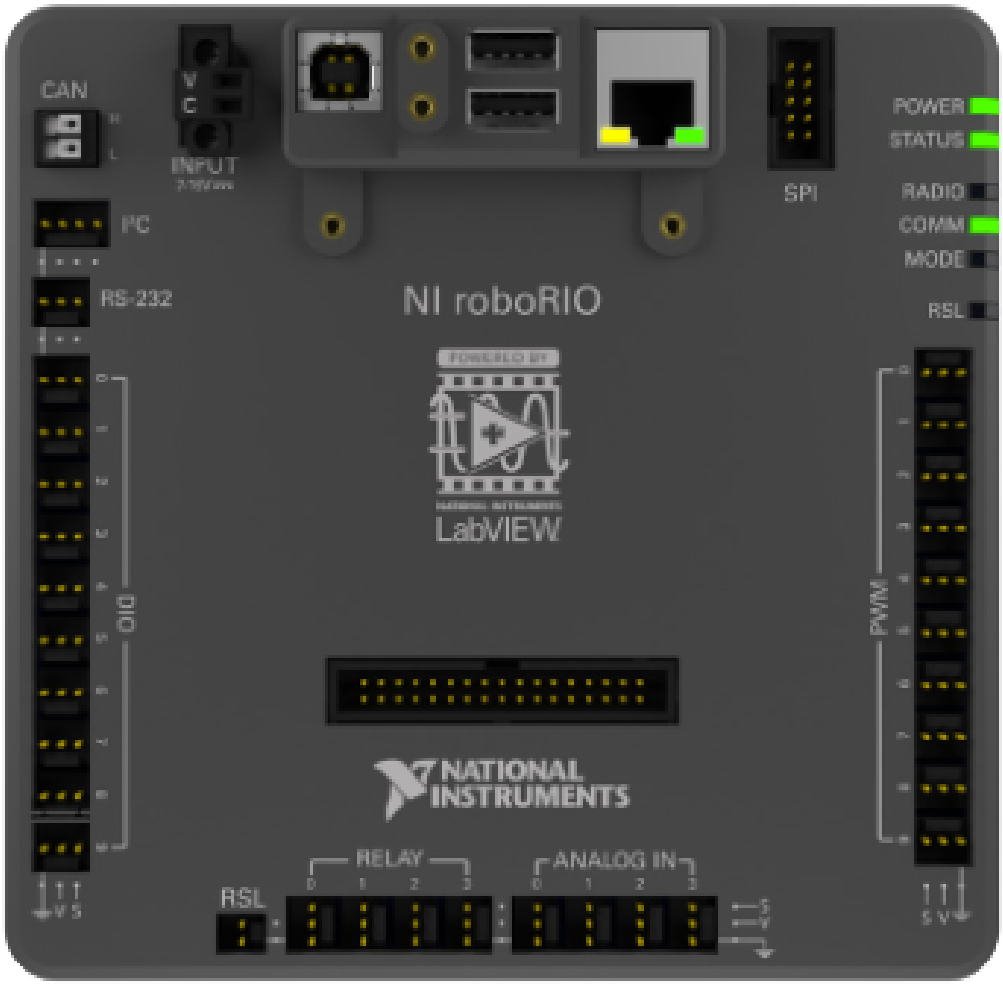
The capabilities of Choreo include

- creating trajectories with a variety of different constraints
- scheduling commands at different events on your path
- support for different drivebases, including swerve and tank

31

roboRIO

31.1 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.

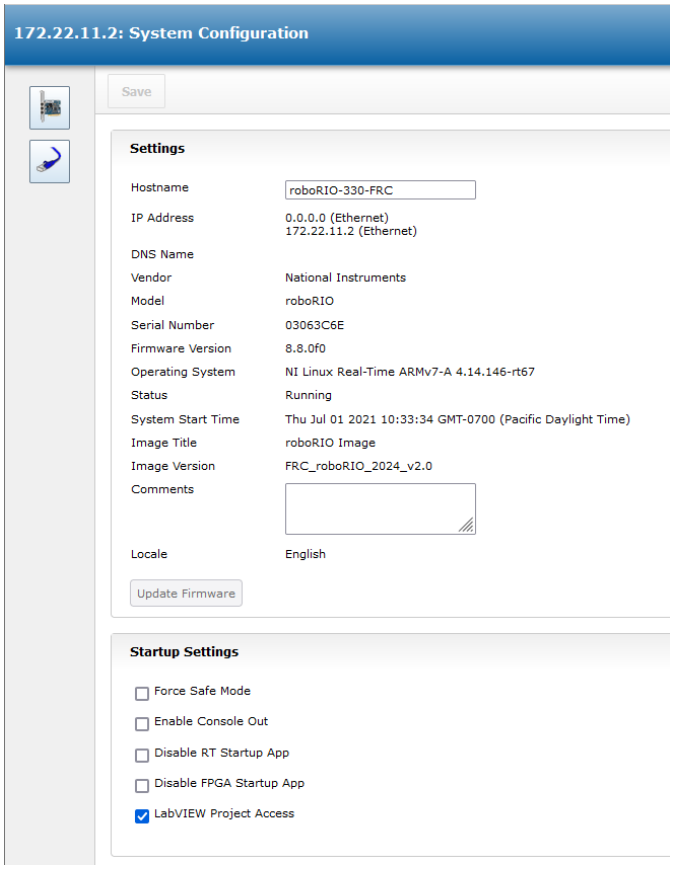
Detailed information on the roboRIO can be found in the roboRIO User Manual and in the roboRIO technical specifications.

Before deploying programs to your roboRIO, you must first image the roboRIO: *roboRIO 1 roboRIO 2*.

31.2 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.

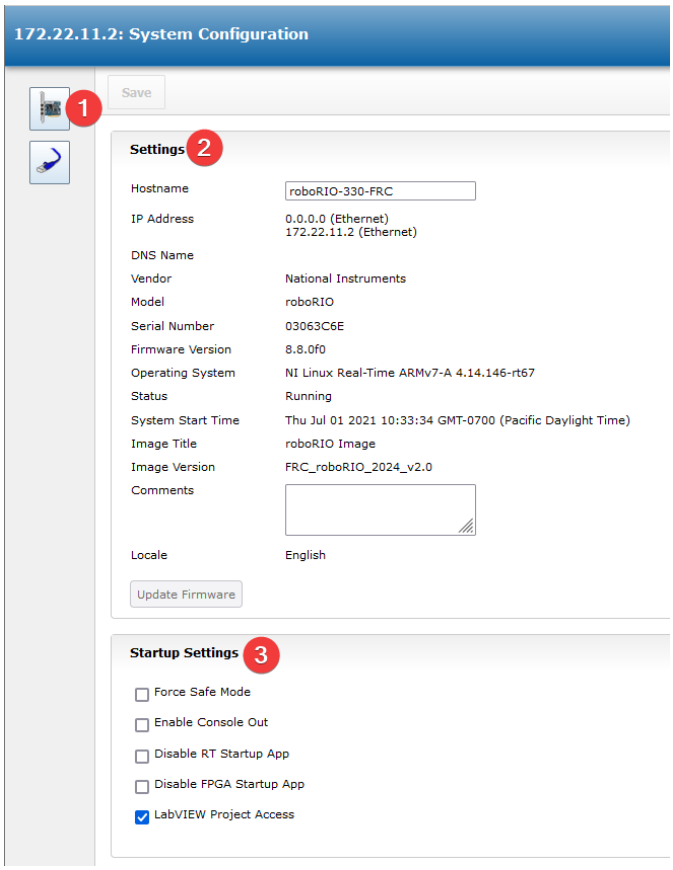
31.2.1 Opening the WebDash



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*

31.2.2 System Configuration Tab



The home screen of the web dashboard is the System Configuration tab which has 5 main sections:

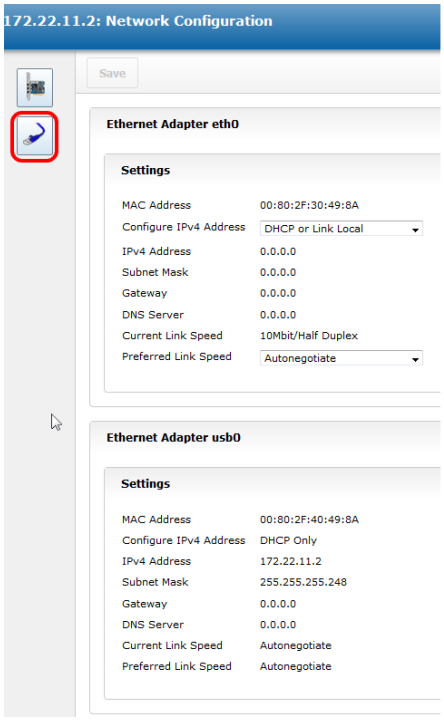
1. Navigation Bar - This section allows you to navigate to different sections of the web dashboard. The different pages accessible through this navigation bar are discussed below.
2. System Settings - This section contains information about the System Settings. The Hostname field should not be modified 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.
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



- 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 rest button for 5 seconds). **Default is unchecked.**
- Enable Console Out - This enables the on-board RS232 port to be used as a Console output. This port uses RS232 levels and should not be connected to many microcontrollers which use TTL levels). **Default is unchecked.**
- 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.**
- LabVIEW Project Access - **It is recommended to leave this box checked.** This setting allows LabVIEW projects to access the roboRIO.

31.2.3 Network Configuration



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*

31.3 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.

31.3.1 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.

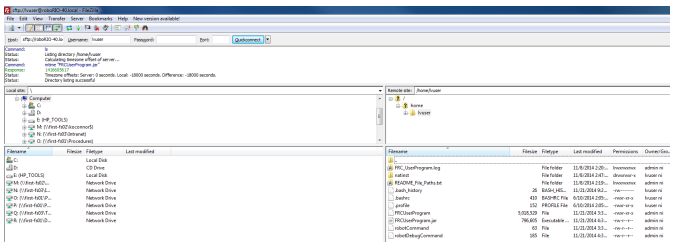
Connecting to the roboRIO



To connect to your roboRIO:

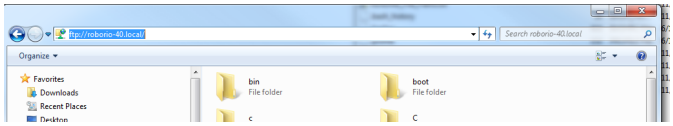
1. Enter the mDNS name (roboRIO-TEAM-frc.local) in the "Host" box
2. Enter "lvuser" in the Username box (this is the account your program runs under)
3. Leave the Password box blank
4. Enter "22" in the port box (the SFTP default port)
5. Click Quickconnect

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”.

31.3.2 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. 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.

31.4 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.

31.4.1 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++, Java, and Python teams from uploading code.

31.4.2 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. OpenSSH is included by default on Windows, macOS, and Linux.

Connect via SSH

To connect to the roboRIO, open a terminal or command prompt and run:

```
ssh admin@roboRIO-TEAM-frc.local
```

Replace `TEAM` with your team number (e.g., `ssh admin@roboRIO-40-frc.local` for team 40).

If you are connected over USB you can use `172.22.11.2` as the hostname:

```
ssh admin@172.22.11.2
```

If your roboRIO is set to a static IP you can use that IP as the hostname if connected over Ethernet/wireless.

Log In

When you see the password prompt, press enter (the password for both accounts is blank). You can also connect as the `lvuser` account by replacing `admin` with `lvuser` in the commands above.

If you see a prompt about host authenticity when connecting for the first time, type `yes` and press enter to continue.

31.5 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.

31.5.1 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
- Pneumatic Devices such as the CTRE Pneumatics Control Module and REV Pneumatic Hub are disabled

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

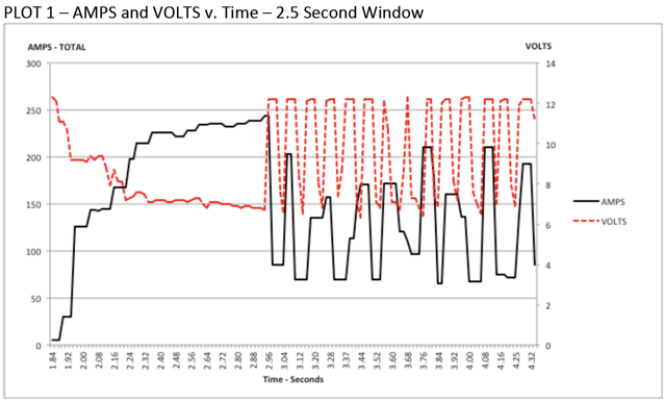
Below 4.5V the device may blackout. The exact voltage may be lower than this and depends on the load on the device.

The controller will remain in this state until the voltage rises above 4.65V when the device will begin the normal boot sequence.

31.5.2 Avoiding Brownout - Proactive Current Draw Planning

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).

31.5.3 Settable Brownout

The NI roboRIO 1.0 does not support custom brownout voltages. It is fixed at 6.3V as mentioned in Stage 2 above.

The NI roboRIO 2.0 adds the option for a software settable brownout level. The default brownout level (Stage 2) of the roboRIO 2.0 is 6.75V.

JAVA

```
RobotController.setBrownoutVoltage(7.0);
```

C++

```
frc::RobotController::SetBrownoutVoltage(7_V);
```

31.5.4 Measuring Current Draw using the PDP/PDH

The FRC® Driver Station works in conjunction with the roboRIO and PDP/PDH to extract logged data from the PDP/PDH and log it on your DS PC which is visible in the *Driver Station Log Viewer*

Teams can also 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/PDH channel using the Get PD Currents VI found on the Power pallet. For C++ and Java teams, use the PowerDistribution class as described in the *Power Distribution* 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).

Note

Only the CTRE PDP and Rev PDH have built-in current logging. The CTRE PDP 2.0 and AndyMark AMPD do not have current measurment capabilities, and teams would need to implement their own method of *logging* the current draw. Most CAN motor controllers have current sensors that can be used for this.

31.5.5 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).

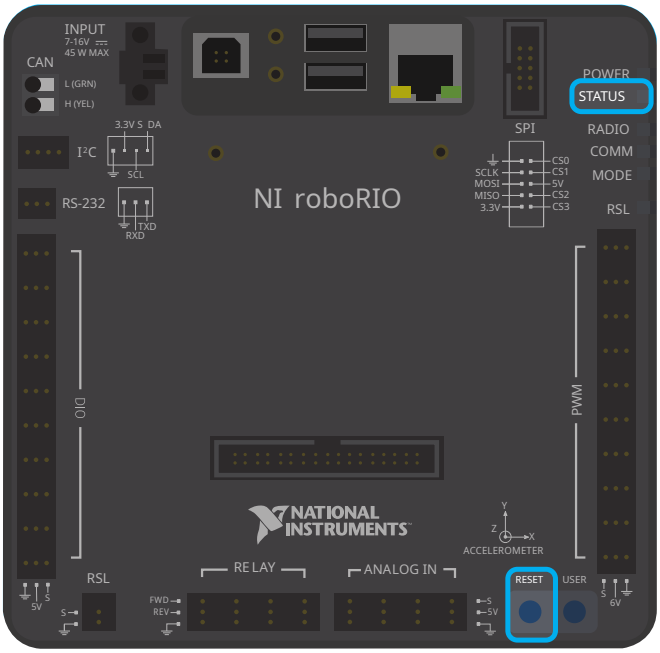
31.6 Recovering a roboRIO using Safe Mode

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.

Important

These steps are not applicable to the roboRIO 2. Reimaging the SD card following *roboRIO 2.0 microSD card imaging process* will fully reformat the device.

31.6.1 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)

31.6.2 Recovering the roboRIO

The roboRIO can now be imaged by using the roboRIO Imaging Tool as described in *Imaging your roboRIO*.

31.6.3 About Safe Mode

In Safe Mode, the roboRIO boots a separate copy of the operating system into a RAM Disk (the firmware). 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.

32 Advanced GradleRIO

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.

32.1 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.

32.1.1 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!

32.1.2 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.

32.2 Setting up CI for Robot Code using GitHub Actions

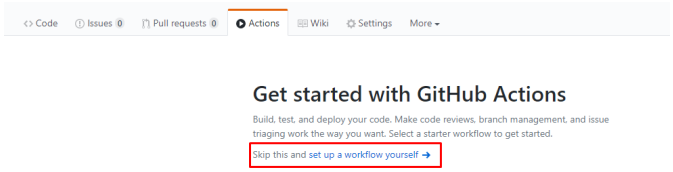
An important aspect of working in a team environment is being able to test code that is pushed to a central repository such as GitHub. For example, a project manager or lead developer might want to run a set of unit tests before merging a pull request or might want to ensure that all code on the main 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*.

32.2.1 Creating the Action

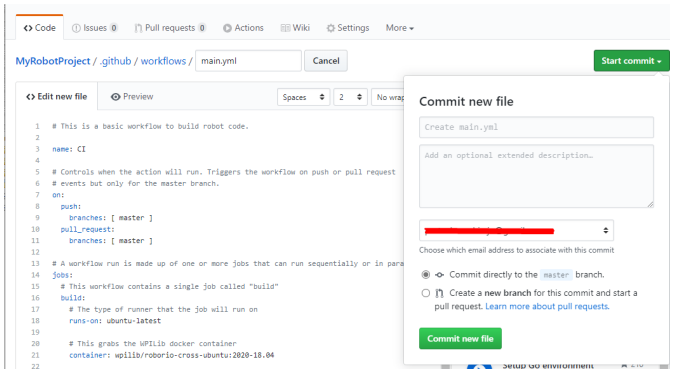
The instructions for carrying out the CI process are stored in a YAML file. To create this, click on the "Actions" tab at the top of your repository. Then click on the "set up a workflow yourself" hyperlink.



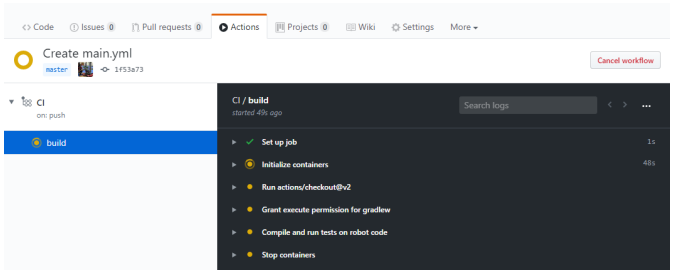
You will now be greeted with a text editor. Replace all the default text with the following:

```yaml
# 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-22.04
    # This grabs the WPILib docker container
    container: wpilib/roborio-cross-ubuntu:2025-22.04
    # 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@v6
    # Declares the repository safe and not under dubious ownership.
    - name: Add repository to git safe directories
      run: git config --global --add safe.directory $GITHUB_WORKSPACE
    # 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.



32.2.2 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-22.04
    # This grabs the WPILib docker container
    container: wpilib/roborio-cross-ubuntu:2025-22.04
```

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.

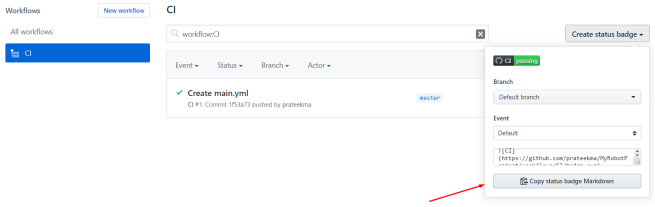
> **Note**
>
> Since none of the dependencies changed between 2025 and 2026, the 2025 docker images are still correct to use for 2026.

```
# 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@v6
# Declares the repository safe and not under dubious ownership.
- name: Add repository to git safe directories
  run: git config --global --add safe.directory $GITHUB_WORKSPACE
# 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
```

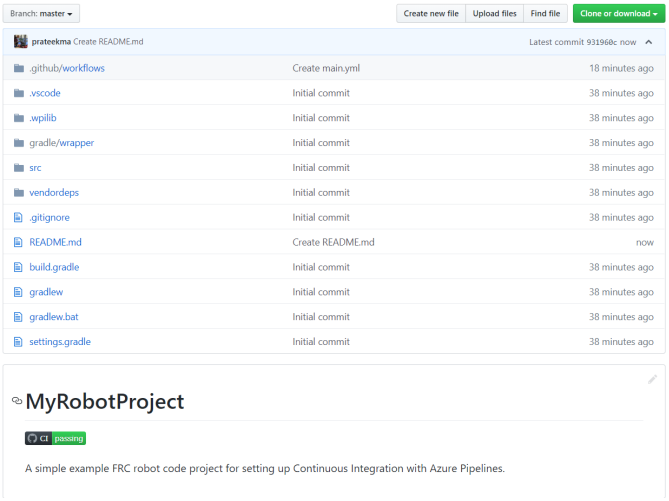
Each job has certain steps that will be executed. This job has four steps. The first step involves checking out the repository to access the robot code. The second step is a workaround for a GitHub Actions issue. The third step involves giving the virtual machine permission to execute gradle tasks using `./gradlew`. The final step runs `./gradlew build` to compile robot code and run any unit tests.

32.2.3 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.



32.3 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.

32.3.1 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 "2024.1.1"
   id 'com.diffplug.spotless' version '6.20.0'
}
```

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-*/**', '**/bin/**'
      }
      toggleOffOn()
      googleJavaFormat()
      removeUnusedImports()
      trimTrailingWhitespace()
```

(continues on next page)

(continued from previous page)

```
      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()
      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`.

In addition to formatting code, Spotless can also ensure the code is correctly formatted; this can be used by running `./gradlew spotlessCheck`. Thus, Spotless can be used as a *CI check*, as shown in the following GitHub Actions workflow:

```
on: [push]
# A workflow run is made up of one or more jobs that can run sequentially or in␣
↪parallel
jobs:
  spotless:
    # The type of runner that the job will run on
    runs-on: ubuntu-22.04
    # 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@v6
        with:
```

(continues on next page)

(continued from previous page)

```
          fetch-depth: 0
      - uses: actions/setup-java@v5
        with:
          distribution: 'temurin'
          java-version: 17
      - run: ./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
```

32.3.2 wpiformat

Requirements

- Python 3.11 or higher

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`. There are three configuration files one can place in the project root: `.clang-format` (optional), `.wpiformat` (optional), and `.wpiformat-license` (required).

- `.clang-format` style option docs
- `.wpiformat` configuration docs

An example `.wpiformat-license` is shown below:

```
// Copyright (c) FIRST and other WPILib contributors.
// Open Source Software; you can modify and/or share it under the terms of
// the WPILib BSD license file in the root directory of this project.
```

You can turn this into a *CI check* by running `git --no-pager diff --exit-code HEAD`, as shown in the example GitHub Actions workflow below:

```yaml
name: Lint and Format

on:
  pull_request:
  push:
jobs:
  wpiformat:
    name: "wpiformat"
    runs-on: ubuntu-22.04
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0
      - name: Fetch all history and metadata
        run: |
          git checkout -b pr
          git branch -f main origin/main
      - name: Set up Python 3.12
        uses: actions/setup-python@v5
        with:
          python-version: '3.12'
      - name: Install wpiformat
        run: |
          python -m venv ${{ runner.temp }}/wpiformat
          ${{ runner.temp }}/wpiformat/bin/pip3 install wpiformat==2025.36
      - name: Run
        run: ${{ runner.temp }}/wpiformat/bin/wpiformat
      - name: Check output
        run: git --no-pager diff --exit-code HEAD
```

32.4 Gradlew Tasks

This article aims to highlight the gradle commands supported by the WPILib team for user use. These commands can be viewed by typing `./gradlew tasks` at the root of your robot project. Not all commands shown in `./gradlew tasks` and unsupported commands will not be documented here.

32.4.1 Build tasks

`./gradlew build` - Assembles and tests this project. Useful for prebuilding your project without deploying to the roboRIO.

`./gradlew clean` - Deletes the build directory.

32.4.2 CompileCommands tasks

`./gradlew generateCompileCommands` - Generate compile_commands.json for C++ programs. This is a configuration file that is used by clangd, which is supported by many Integrated Development Environments and build tools. This file will be saved in `build/TargetedCompileCommands/targetplatformandbuildtype` but needs to be moved to the root of your project. Many IDE's will have a way to specify where this file is found, which you can use instead if you desire. Make sure you have gitignored this file.

> **Warning**
>
> Using compile commands isn't recommended for most users and has a variety of pitfalls compared to the built-in intellisense.

32.4.3 DeployUtils tasks

`./gradlew deploy` - Deploy all artifacts on all targets. This will deploy your robot project to the available targets (IE, roboRIO).

`./gradlew discoverRoborio` - Determine the address(es) of target roboRIO. This will print out the IP address of a connected roboRIO.

32.4.4 GradleRIO tasks

`./gradlew :math:`TOOL`` - Runs the tool `$TOOL:math:` (Replace `TOOL$ with the name of the tool. IE, Glass, Shuffleboard, etc)

`./gradlew :math:`TOOL`Install` - Installs the java tool `$TOOL:math:` (Replace `TOOL$ with the name of the tool. IE, Shuffleboard, SmartDashboard, etc)

`./gradlew InstallAllTools` - Installs all available tools. This excludes the development environment such as Visual Studio Code. It's the users requirement to ensure the required dependencies (Java) is installed. Only recommended for advanced users!

`./gradlew simulateExternalNativeRelease` - Simulate External Task for native executable. Exports a JSON file for use by editors / tools

`./gradlew simulateExternalJavaRelease` - Simulate External Task for Java/Kotlin/JVM. Exports a JSON file for use by editors / tools

`./gradlew simulateJava` - Launches simulation for the Java projects

`./gradlew simulateNative` - Launches simulation for C++ projects

`./gradlew vendordep` - Install vendordep JSON file from URL or local installation. See *3rd Party Libraries*

32.5 Including Git Data in Deploy

This article will explain how to include metadata from Git in robot code using the gversion Gradle plugin. This generates a file which can be used for accessing Git metadata in robot code. This can be used to track what revision of code is on the robot, such as by printing or logging it.

> **Note**
>
> For Python teams, Git metadata is always copied to your robot during the deploy process. You can use `wpilib.deployinfo.getDeployData()` to retrieve the stored information.

32.5.1 Installing gversion

To install gversion add the following line to the plugins block of `build.gradle`. This tells Gradle to use gversion in the project.

```
plugins {
    // ...
    id "com.peterabeles.gversion" version "1.10"
}
```

In order to generate the file when building the project, add the following block to the bottom of `build.gradle`.

```
project.compileJava.dependsOn(createVersionFile)
gversion {
  srcDir       = "src/main/java/"
  classPackage = "frc.robot"
  className    = "BuildConstants"
  dateFormat   = "yyyy-MM-dd HH:mm:ss z"
  timeZone     = "America/New_York" // Use preferred time zone
  indent       = "  "
}
```

The `srcDir`, `classPackage`, and `className` parameters tell the plugin where to put the manifest file, and what to name it. The `timeZone` field can be set to your team's time zone based on this list of timezone IDs. The `dateFormat` parameter is based on this Java class.

To test this, run a build of your project through the WPILib menu in the top right. Once the code has finished building, there should be a file called `BuildConstants.java` in your `src/main/java` folder. The following is an example of what this file should look like:

```
package frc.robot;
/**
* Automatically generated file containing build version information.
*/
public final class BuildConstants {
  public static final String MAVEN_GROUP = "";
  public static final String MAVEN_NAME = "GitVersionTest";
  public static final String VERSION = "unspecified";
  public static final int GIT_REVISION = 1;
  public static final String GIT_SHA = "fad108a4b1c1dcdfc8859c6295ea64e06d43f557";
```

(continues on next page)

(continued from previous page)

```
  public static final String GIT_DATE = "2023-10-26 17:38:59 EDT";
  public static final String GIT_BRANCH = "main";
  public static final String BUILD_DATE = "2023-10-27 12:29:57 EDT";
  public static final long BUILD_UNIX_TIME = 1698424197122L;
  public static final int DIRTY = 0;
  private BuildConstants(){}
}
```

`DIRTY` indicates whether there are uncommitted changes in the project. A value of 0 indicates a clean repository, a value of 1 indicates that there are uncommitted changes, and a value -1 indicates an error.

Ignoring Generated Files with Git

These files are regenerated every time code is built or deployed, so it isn't necessary to track them with Git. The aptly named .gitignore file tells Git not to track any listed files and should exist by default in any project generated by the WPILib VS Code extension. Below is the line you should add to `.gitignore` to ignore the generated file:

```
src/main/java/frc/robot/BuildConstants.java
```

32.6 Using Compiler Arguments

Compiler arguments allow us to change the behavior of our compiler. This includes making warnings into errors, ignoring certain warnings and choosing optimization level. When compiling code a variety of flags are already included by default which can be found here. Normally it could be proposed that the solution is to pass them in as flags when compiling our code but this doesn't work in GradleRIO. Instead modify the build.gradle.

> **⚠ Warning**
>
> Modifying arguments is dangerous and can cause unexpected behavior.

32.6.1 C++

Platforms

Different compilers and different platforms use a variety of different flags. Therefore to avoid breaking different platforms with compiler flags configure all flags per platform. The platforms that are supported are listed here

Configuring for a Platform

```
nativeUtils.platformConfigs.named('windowsx86-64').configure {
  it.cppCompiler.args.add("/utf-8")
}
```

native-utils is used to configure the platform, in this case, *windowsx86-64*. This can be replaced for any platform listed in the previous section. Then arguments, such as */utf-8* is appended to the C++ compiler.

32.6.2 Java

Compiler arguments can also be configured for Java. We do this by adding arguments in the *JavaCompile* task.

```
// Configure string concat to always inline compile
tasks.withType(JavaCompile) {
    options.compilerArgs.add '-XDstringConcat=inline'
}
```

JVM Arguments

Along with being able to configure compiler arguments Java also allows us to configure runtime options for the JVM. We do this by editing the *frcJava* artifact's arguments.

```
frcJava(getArtifactClass('FRCJavaArtifact')) {
  jvmArgs.add("-XX:+DisableExplicitGC")
}
```

32.7 Profiling with VisualVM

This document is intended to familiarize the reader with the diagnostic tool that is VisualVM for debugging Java robot programs. VisualVM is a tool for profiling JVM based applications, such as viewing why an application is using a large amount of memory. This document assumes the reader is familiar with the *risks* associated with modifying their robot `build.gradle`. This tutorial also assumes that the user knows basic terminal/commandline knowledge.

32.7.1 Unpacking VisualVM

To begin, download VisualVM and unpack it to the WPILib installation folder. The folder is located at `~/wpilib/` where ~ indicates the users home directory. On Windows, this is `C:\Users\Public\wpilib`.

32.7.2 Setting up Gradle

GradleRIO supports passing JVM launch arguments, and this is what is necessary to enable remote debugging. Remote debugging is a feature that allows a local machine (such as the user's desktop) to view important information about a remote target (in our case, a roboRIO). To begin, locate the `frcJava` code block located in the projects `build.gradle`. Below is what is looks like.

```
deploy {
    targets {
        roborio(getTargetTypeClass('RoboRIO')) {
            // Team number is loaded either from the .wpilib/wpilib_preferences.json
            // or from command line. If not found an exception will be thrown.
            // You can use getTeamOrDefault(team) instead of getTeamNumber if you
            // want to store a team number in this file.
            team = project.frc.getTeamNumber()
            debug = project.frc.getDebugOrDefault(false)

            artifacts {
                // First part is artifact name, 2nd is artifact type
                // getTargetTypeClass is a shortcut to get the class type using a string

                frcJava(getArtifactTypeClass('FRCJavaArtifact')) {
                }

                // Static files artifact
                frcStaticFileDeploy(getArtifactTypeClass('FileTreeArtifact')) {
                    files = project.fileTree('src/main/deploy')
                    directory = '/home/lvuser/deploy'
                    deleteOldFiles = false // Change to true to delete files on roboRIO that no
                                           // longer exist in deploy directory of this project
                }
            }
        }
    }
}
```

We will be replacing the highlighted lines with:

```
frcJava(getArtifactTypeClass('FRCJavaArtifact')) {
   // Enable VisualVM connection
   jvmArgs.add("-Dcom.sun.management.jmxremote=true")
   jvmArgs.add("-Dcom.sun.management.jmxremote.port=1198")
   jvmArgs.add("-Dcom.sun.management.jmxremote.local.only=false")
   jvmArgs.add("-Dcom.sun.management.jmxremote.ssl=false")
   jvmArgs.add("-Dcom.sun.management.jmxremote.authenticate=false")
   jvmArgs.add("-Djava.rmi.server.hostname=10.TE.AM.2") // Replace TE.AM with team number
}
```

We are adding a few arguments here. In order:

- Enable remote debugging
- Set the remote debugging port to 1198

- Allow listening from remote targets
- Disable SSL authentication being required
- Set the hostname to the roboRIOs team number. Be sure to replace this. (*TE.AM IP Notation*)

Important

The hostname when connected via USB-B should be `172.22.11.2`.

32.7.3 Running VisualVM

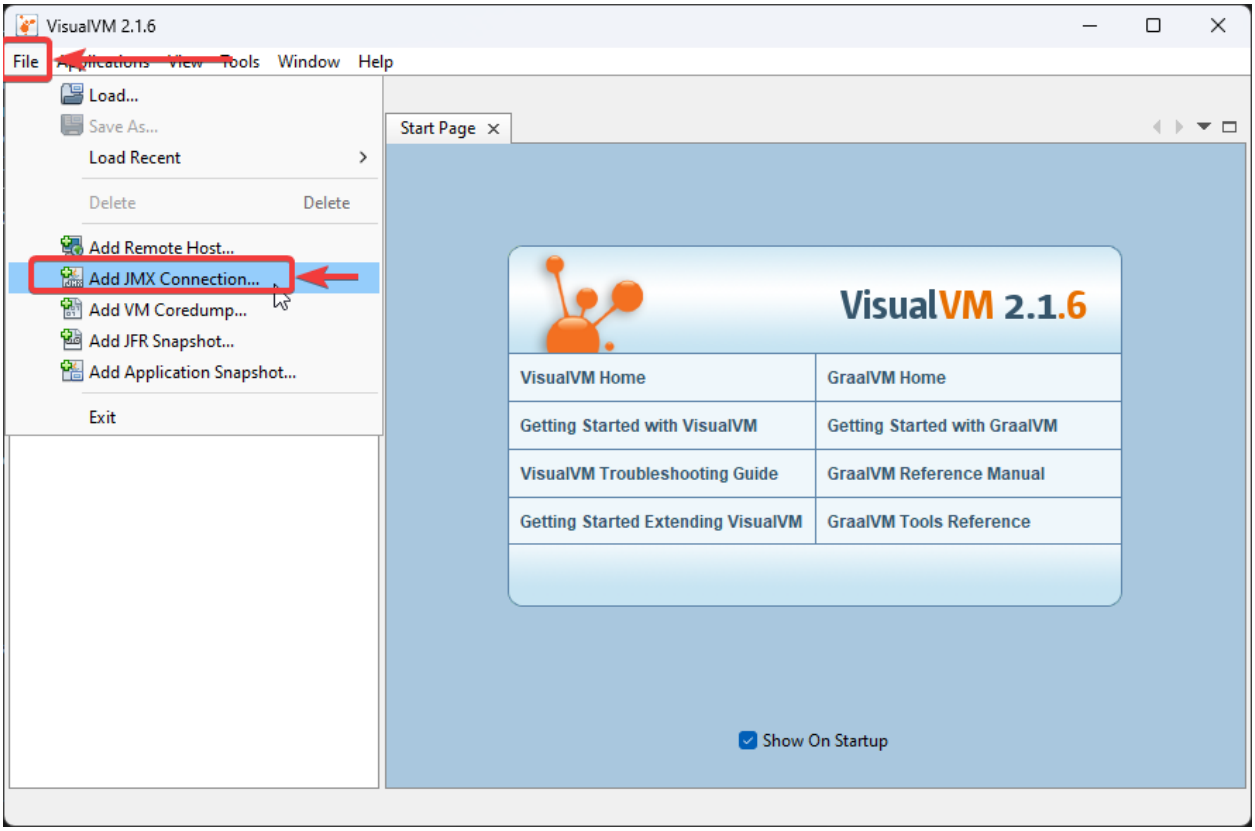
Launching VisualVM is done via the commandline with a few parameters. First, we navigate to the directory containing VisualVM. Then, launch it with parameters, passing it the WPILib JDK path. On a Windows machine, it looks like the following:

```
cd "C:\Users\Public\wpilib\visualvm_2110\bin"
./visualvm --jdkhome "C:\Users\Public\wpilib\2026\jdk"
```

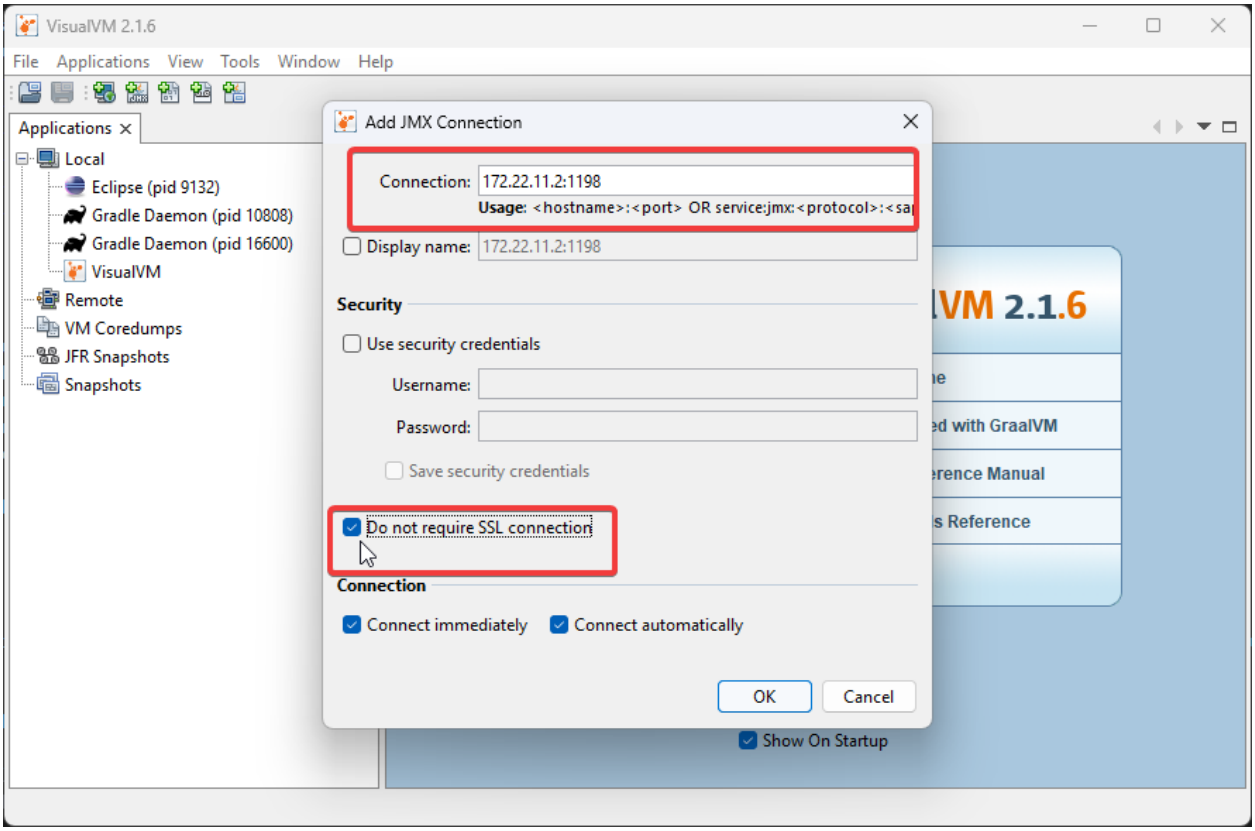
Important

The exact path `visualvm_2110` may vary and depends on the version of VisualVM downloaded.

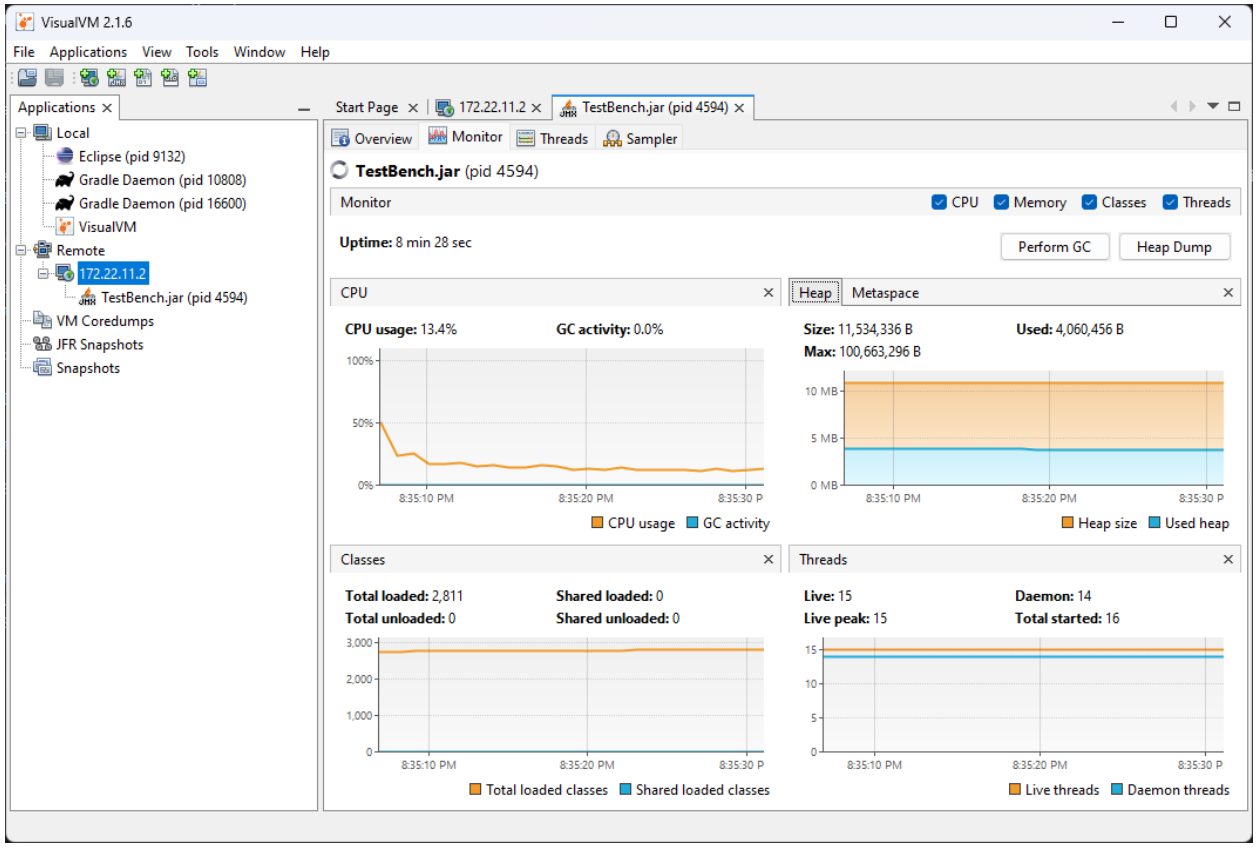
This should launch VisualVM. Once launched, open the *Add JMX Connection* dialog.



Once opened, configure the connection details and hostname. Ensure that *Do not require SSL connection* is ticked.

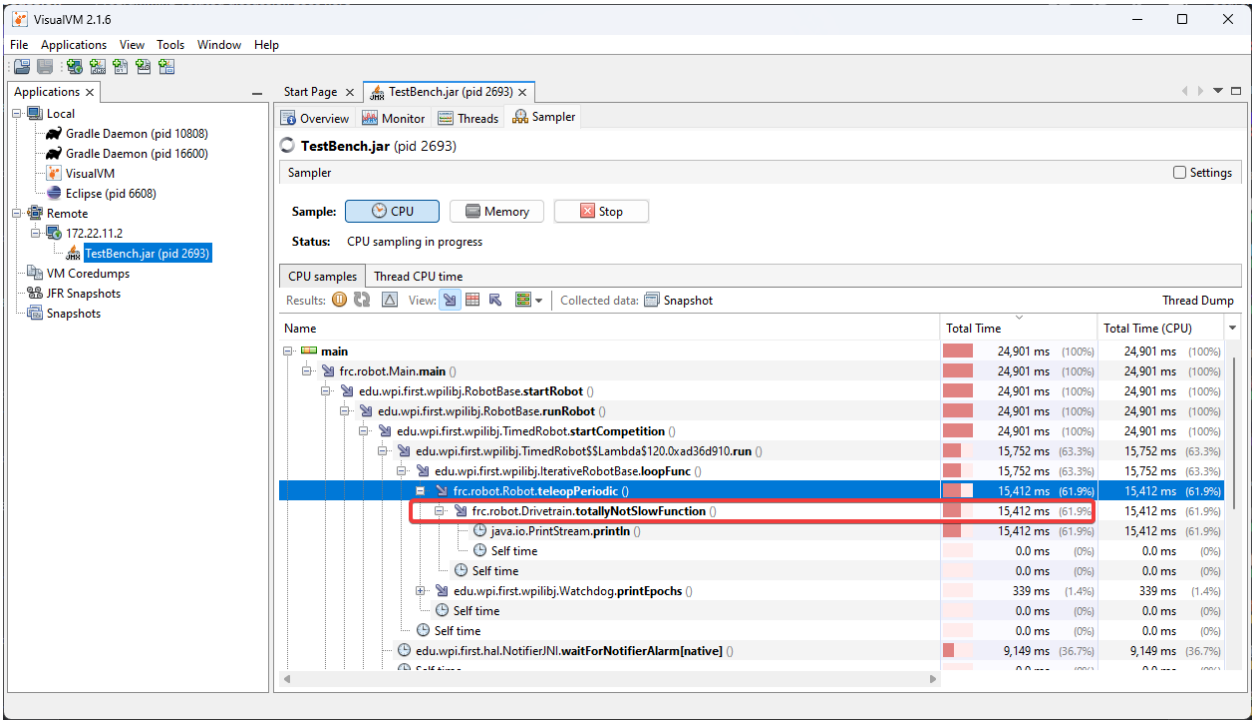


If correctly done, a new menu option in the left-hand sidebar will appear. Clicking on it will show you a detailed dashboard of the running JVM application.



32.7.4 Analyzing Function Timings

An important feature of VisualVM is the ability to view how much time a specific function is taking up. This is *without* having a code debugger attached. To begin, click on the *Sampler* tab and then click on *CPU*. This will immediately give a breakdown of what functions are taking CPU time.



The above screenshot shows a breakdown of the total time a specific function takes. You can see that `totallyNotSlowFunction()` accounts for `61.9%` of the robot program CPU time. We can then correlate this to our robot program. In `totallyNotSlowFunction()`, we have the following code.

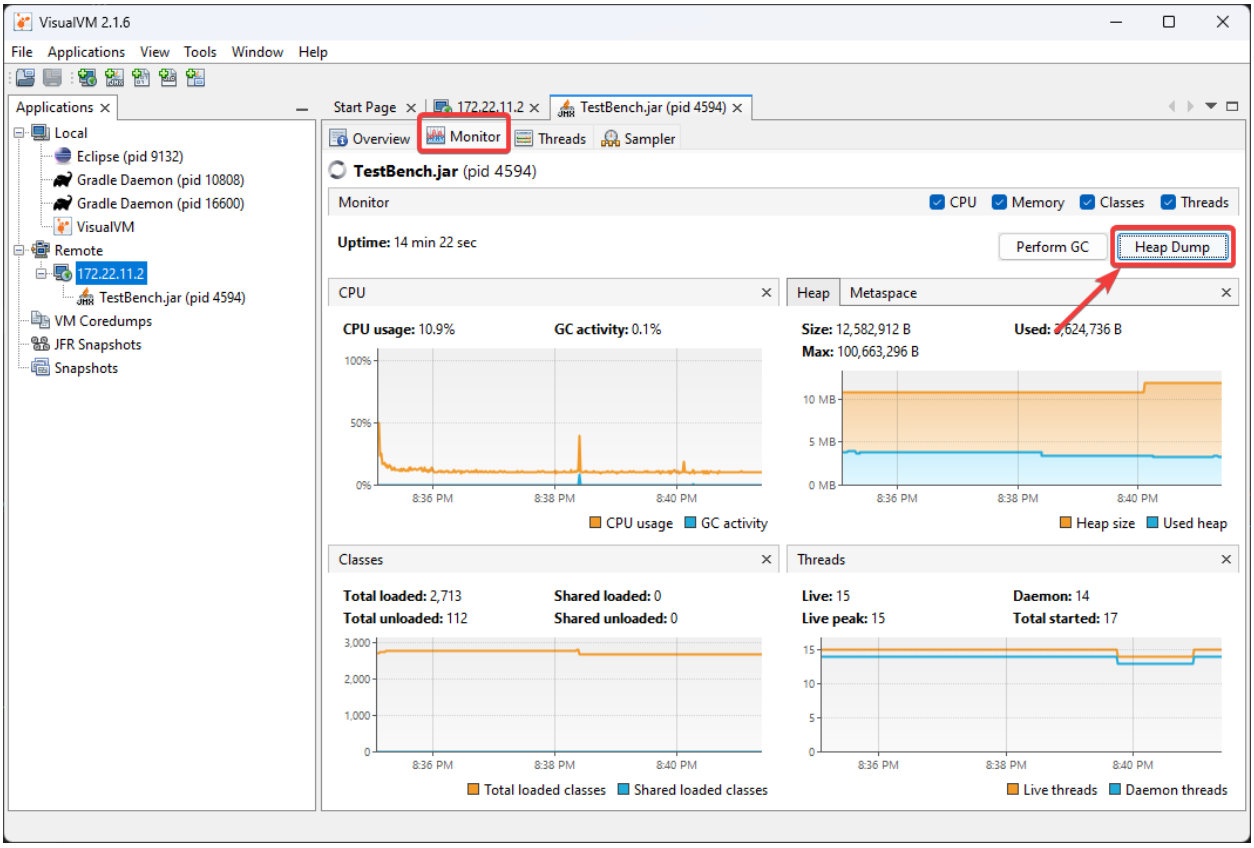
```
public static void totallyNotSlowFunction() {
   for (int i = 0; i < 2000; i++) {
      System.out.println("HAHAHAHA");
   }
}
```

In this code snippet, we can identify 2 major causes of concern. A long running `for` loop blocks the rest of the robot program from running. Additionally, `System.out.println()` calls on the roboRIO are typically quite expensive. We found this information by profiling the Java application on the roboRIO!

32.7.5 Creating a Heap Dump

Besides viewing the remote systems CPU and memory usage, VisualVM is most useful by creating a **Heap Dump**. When a Java object is created, it resides in an area of memory called the heap. When the heap is full, a process called garbage collection begins. Garbage collection can be a common cause of loop overruns in a traditional Java robot program.

To begin, ensure you are on the *Monitor* tab and click *Heap Dump*.



This heap dump will be stored on the target system (roboRIO) and must be retrieved using SFTP. See *this article* for information on retrieving the dump from the roboRIO.

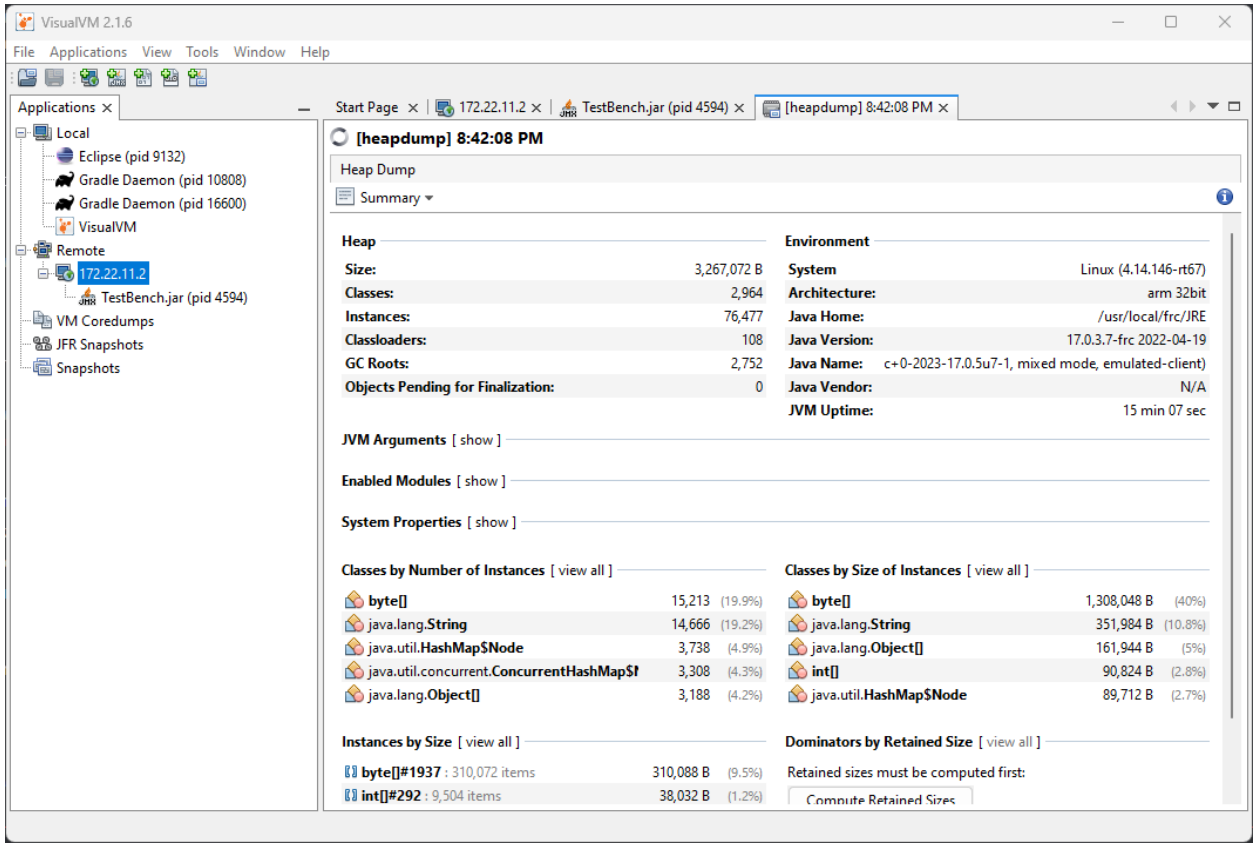
Once downloaded, the dump can be analyzed with VisualVM.

Tip

You can also *configure the JVM to take a heap dump automatically when your robot code runs out of memory*.

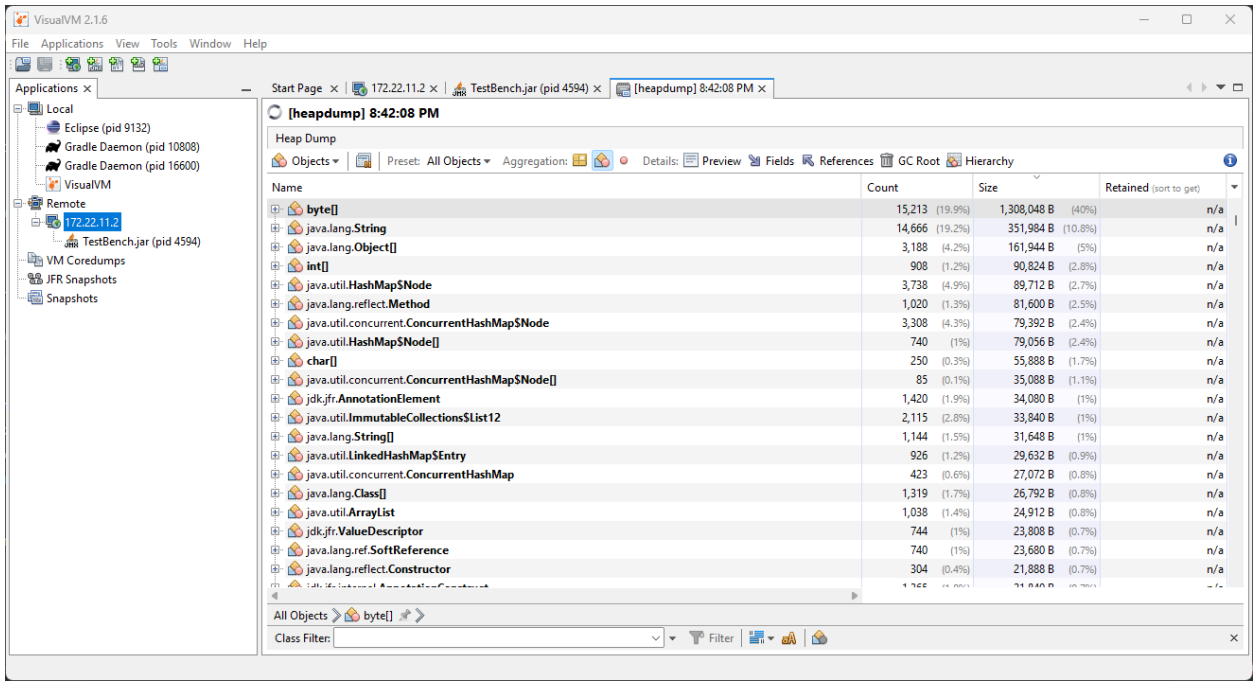
32.7.6 Analyzing a Heap Dump

Reopen VisualVM if closed using the previous instructions. Then click on *File* and *Load*. Navigate to the retrieved dump file and load it.

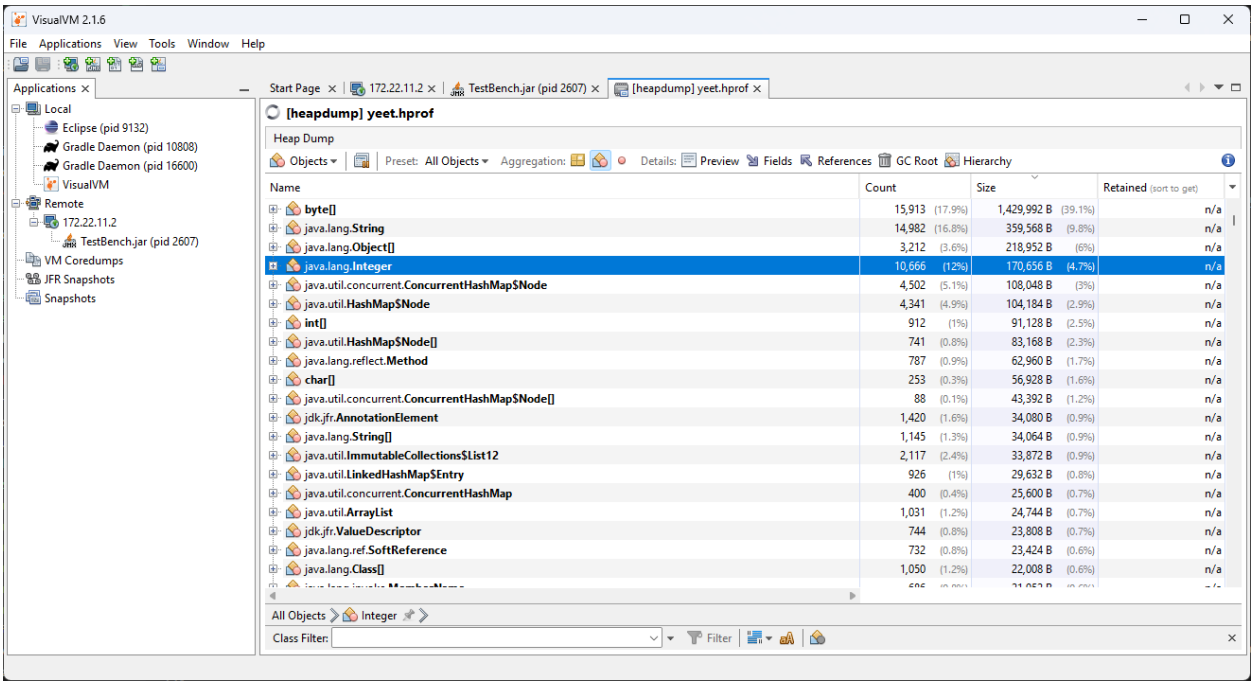


Clicking on *Summary* and selecting *Objects* instead will show a breakdown of objects by quantity. The below screenshot showcases a completely empty robot program, and then one that creates an million large `ArrayList` of integers.

Blank robot program:



with an `ArrayList` of ~10000 integers.



32.7.7 Additional Info

For more information on VisualVM, check out the VisualVM documentation pages.

32.8 JVM Runtime error

Programs compiled with Microsoft Visual C++ (MSVC) require a MSVC runtime that is equal to or newer then the newest library being executed. WPILib is compiled with a recent version of MSVC.

In the past, this hasn't really been used, but a change in the runtime in 2024 actually took advantage of this requirement.

Most JDKs ship with an MSVC runtime, and many of them are old. Additionally, many of them don't actually include the full runtime. Both of these are a problem, as all the JNI libraries loaded by WPILib both use a lot of the runtime, and are compiled against basically the newest runtime.

For the JDK shipped with WPILib, the newest runtime is installed to ensure this issue isn't run into. But other JDKs will fail. So in 2025, much more important than prior years, the WPILib JDK needs to be used.

The following error will be seen if a JDK with an incompatible MSVC runtime is used. This only affects Windows users.

```
> Task :simulateJavaRelease FAILED
If you receive errors loading the JNI dependencies, make sure you have the latest␣
↪Visual Studio C++ Redstributable installed.
```

(continues on next page)

(continued from previous page)

```
That can be found at https://support.microsoft.com/en-us/help/2977003/the-latest-
↪supported-visual-c-downloads
Exception in thread "main" edu.wpi.first.util.MsvcRuntimeException: Invalid MSVC␣
↪Runtime Detected.
Expected at least 14.40, but found 14.29
JVM Location: C:\Program Files\Amazon Corretto\jdk21.0.5_11\bin\java.exe
Runtime DLL Location: C:\Program Files\Amazon Corretto\jdk21.0.5_11\bin\msvcp140.dll
See https://wpilib.org/jvmruntime for more information

        at edu.wpi.first.util.WPIUtilJNI.checkMsvcRuntime(Native Method)
        at edu.wpi.first.wpilibj.RobotBase.startRobot(RobotBase.java:470)
        at frc.robot.Main.main(Main.java:23)

FAILURE: Build failed with an exception.
```

32.8.1 Setting Gradle to use WPILib JDK

This is not necessary if using the WPILib VS Code, which is the supported way to run. The following methods may be used for other Windows users.

Ensure you've *installed WPILib* before proceeding.

Commandline Parameter

To ensure that Gradle uses the WPILib JDK, you must set `org.gradle.java.home` to the WPILib JDK. This can be done by passing a parameter to gradle (where YEAR is the current WPILib installation):

```
gradlew -D org.gradle.java.home="C:\\Users\\Public\\wpilib\\YEAR\\jdk" ...
```

gradle.properties

It can also be done by putting `gradle.properties` in the project root with the following contents (where YEAR is the current WPILib installation):

Warning

`gradle.properties` should not be checked into git or other version management tools. This will only work if you only use your project on Windows, as the location is hardcoded to the location on Windows.

```
org.gradle.java.home=C:\\Users\\Public\\wpilib\\YEAR\\jdk
```

IDE

Some IDEs have a method to set the JDK used. Consult the documentation for your IDE.

33 Advanced Controls

This section covers advanced control features in WPILib, such as various feedback/feedforward control algorithms and trajectory following.

33.1 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.

33.2 Advanced Controls Introduction

33.2.1 Control System Basics

> **Note**
>
> This article includes sections of Controls Engineering in FRC by Tyler Veness with permission.

The Need for Control Systems

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).

All of these examples have a mechanism which does useful work, but cannot be *directly* commanded to the state that is desired.

For example, an air conditioner's fans and compressor have no mechanical or electrical input where the user specifies a temperature. Rather, some additional mechanism must compare the current air temperature to some setpoint, and choose how to cycle the compressor and fans on and off to achieve that temperature.

Similarly, an automobile's engine and transmission have no mechanical lever which directly sets a particular speed. Rather, some additional mechanism must measure the current speed of the vehicle, and adjust the transmission gear and fuel injected into the cylinders to achieve the desired vehicle speed.

Controls Engineering is the study of how to design those additional mechanisms to bridge the gap from what the user wants a mechanism to do, to how the mechanism is actually manipulated.

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.

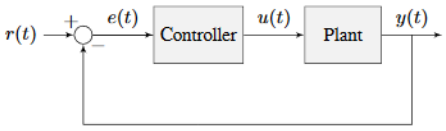


Figure 1.1: Control system nomenclature for a closed-loop system

$r(t)$	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.

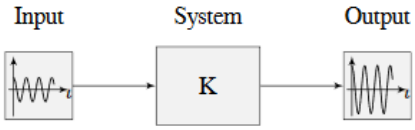


Figure 1.2: Demonstration of system with a gain of $K = 2$

What is a Model?

A *model* of your mechanism is a mathematical description of its behavior. Specifically, this mathematical description must define the mechanism's inputs and outputs, and how the output values change over time as a function of its input values.

The mathematical description is often just simple algebra equations. It can also include some linear algebra, matrices, and differential equations. WPILib provides a number of classes to help simplify the more complex math.

Classical Mechanics defines many of the equations used to build up models of system behavior. Many of the values inside those equations can be determined by doing experiments on the mechanism.

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.

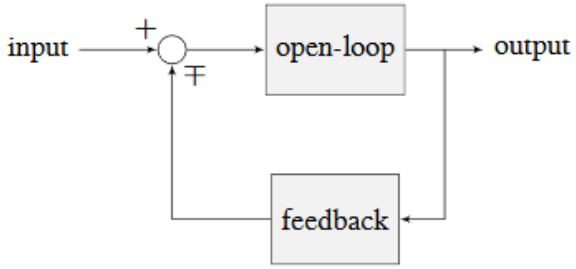


Figure 1.3: Block diagram with nomenclature

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.

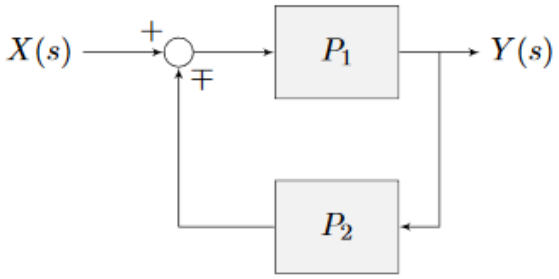


Figure 1.4: Feedback block diagram

$\mp$ means "minus or plus" where a minus represents negative feedback.

A Note on Dimensionality

For the purposes of the introductory section, all systems and controllers (except feedforward controllers) are assumed to be "single-in, single-out" (SISO) - this means they only map single values to single values. For example, a DC motor is considered to take an *input* of a single scalar value (voltage) and yield an *output* of only a single scalar value in return (either position or velocity). This forces us to consider *position controllers* and *velocity controllers* as separate entities - this is sometimes source of confusion in situations when we want to control both (such as when following a motion profiles). Limiting ourselves to SISO systems also means that we are unable to analyze more-complex "multiple-in, multiple-out" (MIMO) systems like drivetrains that cannot be represented with a single state (there are at least two independent sets of wheels in a drive).

Nonetheless, we restrict ourselves to SISO systems here to be able to present the following tutorials in terms of the PID Controller formalism, which is commonly featured in introductory course material and has extensive documentation and many available implementations.

The *state-space* formalism is an alternate way to conceptualize these systems which allows us to easily capture interactions between different quantities (as well as simultaneously represent multiple aspects of the same quantity, such as position and velocity of a motor). It does this, roughly, by replacing the single-dimensional scalars (e.g. the *gain*, *input*, and *output*) with multi-dimensional vectors. In the state-space formalism, the equivalent of a "PID" controller is a vector-proportional controller on a single vector-valued mechanism state, with a single *gain* vector (instead of three different *gain* scalars).

If you remember that a state-space controller is really just a PID controller written with dense notation, many of the principles covered in this set of introductory articles will transfer seamlessly to the case of state-space control.

33.2.2 Picking a Control Strategy

> **Note**
>
> This article includes sections of Controls Engineering in FRC by Tyler Veness with permission.

When designing a control algorithm for a robot mechanism, there are a number of different approaches to take. These range from very simple approaches, to advanced and complex ones. Each has *tradeoffs*. Some will work better than others in different situations, some require more mathematical analysis than others.

Teams should prioritize picking the easiest strategy which enables success on the field. However, as you do experiments, keep in mind there is almost always a "next-step" to take to improve your field performance.

There are two fundamental types of mechanism controller that we will cover here:

> **Note**
>
> These are not strict definitions - some control strategies are not easily classifiable and incorporate elements of both feedforward and feedback controllers. However, it is still a useful distinction in most FRC applications.

Feedforward control (or "open-loop control") refers to the class of algorithms which incorporate knowledge of how the mechanism under control is *expected* to operate. Using this "model" of operation, the control input is chosen to make the mechanism get close to where it should be.

Feedback control (or "closed-loop control") refers to the class of algorithms which use sensors to *measure* what a mechanism is doing, and issue corrective commands to move a mechanism from where it actually is, to where you want it to be.

These are not mutually exclusive, and in fact it is usually best to use both. The tutorial pages that follow will cover three types of mechanism (turret, flywheel, and vertical arm), and allow you to experiment with how each type of system responds to each type of control strategy, both individually and combined.

Feedforward Control: Making a Best Guess

"Feedforward control" means providing the mechanism with the control signal you think it needs to make the mechanism do what you want, without any knowledge of where the mechanism currently is. A feedforward controller feeds information we already know about the system *forward* into an estimate of the required *control effort*. The feedforward controller does *not* adjust this in response to the measured behavior of the system to try to correct for errors from the guess.

Feedforward control is also sometimes referred to as "open-loop control", because if you draw out a block diagram of the controlled system it consists of only a line from the controller to the plant, with no connection from the measured plant output back into the controller (hence an "open" loop, which really isn't a loop at all).

This is the type of control you are implicitly using whenever you use a joystick to "directly" control the speed of a motor through the applied voltage. It is the simplest and most straightforward type of control, and is probably the one you encountered first when programming a FRC motor, though it may not have been referred to by name.

When Do We Need Feedforward Control?

In general, feedforward control is *required* whenever the system requires some constant control signal to remain at the desired setpoint (such as position control of a vertical arm where gravity will cause the arm to fall, or velocity control where internal motor dynamics and friction will cause the motor to slow down over time). Feedback controllers naturally fall to zero output when they achieve their setpoint, and so a feedforward controller is needed to provide the signal to *keep* the mechanism where we want it.

Some control strategies instead account for this in the feedback controller with integral gain - however, this is slow and prone to oscillation. It is almost always better to use a feedforward controller to account for the output needed to maintain the setpoint.

Feedforward and Position Control

The WPILib feedforward classes require velocity and acceleration setpoints to generate an estimated control voltage. This is because the equations-of-motion of a permanent-magnet DC motor relate the applied voltage to velocity and acceleration; it is a fact of physics that we cannot change.

But what if we want to control position? When controlling a DC motor, there's no immediate relation between position and control signal. In order to use feedforward effectively for position control, we need to come up with a sequence of velocities that will take the robot mechanism to the desired position. This is called a *motion profile*.

Many teams do not wish to incur the extra technical cost of using a motion profile when doing position control, and instead omit the feedforward controller entirely and opt to use only feedback control. As we will discuss later, this may work in *some* situations, but has some important caveats.

Most FRC mechanisms are well-described by WPILib's feedforward classes, though pure feedforward control typically only yields acceptable results for velocity control of mechanisms with little external load. In other cases, errors from the system model will be unavoidable and a feedback controller will be necessary to correct for them.

Feedback Control: Correcting for Errors and Disturbances

Even with unlimited study, it is impossible to know every force that will be exerted on a robot's mechanism in perfect detail. For example, in a flywheel shooter, the timing and exact forces associated with a ball being put through the mechanism are extremely difficult to measure accurately. For another example, consider the fact that gearboxes gradually throw off grease as they operate, increasing their internal friction over time. This is a *very* complex process to model well.

In practice, this means that the "guess" made by our feedforward controller will never be perfect. There will always be some error - that is, some lingering difference between the state we want our mechanism to be in, and the state the feedforward controller leaves it in. In many situations, this error is large enough that we need to adjust our output to correct it;

this is the job of the feedback controller. Feedback controllers are also called "closed-loop" controllers, because the flow of information about the current state *back* through the system "closes" the loop in the system's block diagram.

The simplest feedback controller possible is a "proportional controller", which responds proportionally to the current error (i.e. difference between the desired state and measured state). More advanced controllers (such as the PID controller) add response to the rate-of-change of the error and to the total accumulated error. All of these operate on the principle that the system response is roughly linear, in order to "nudge" the system towards the setpoint based on local measurements of the error.

When Do We Need Feedback Control?

In general, there are two scenarios in which we *need* feedback control:

1. We are controlling the position of the system, so errors accumulate over time
2. There are a lot of difficult-to-dynamic external forces interacting with the mechanism that the feedforward loop cannot account for (e.g. a flywheel that is launching game pieces).

In each of these situations, the *best* solution is to combine a feedforward controller and a feedback controller by adding their outputs together. However, in the case of a simple position controller with no external loading, a pure feedback controller can work acceptably.

Feedback-Only Control

Feedforward controllers are extremely helpful and quite simple, but they require *explicit* knowledge of the system behavior in order to generate a guess at the required control signal. In many controls textbooks, you may see a set of techniques which rely on feedback control only. These are very common in industry, and works well in many cases, especially when the underlying system behavior is not easy to explicitly model, or when you want to quickly reach a "good enough" solution without spending the time to thoroughly investigate your system behavior.

Feedback-only control typically only works well in situations where:

1. The motors are fairly overpowered relative to loading.
2. The mechanism's position (not velocity) is being controlled.
3. There are no substantial or varying external forces on the mechanism.

When these criteria are met (such as in the turret tuning tutorial), feedback-only control can yield acceptable results. In other situations, it is necessary to use a feedforward model to reduce the amount of work done by the feedback controller. In FRC, our systems are almost all modeled by well-understood equations with working code support, so it is almost always a good idea to include a feedforward controller.

Modeling: How do you expect your system to behave?

It's easiest to control a system if we have some prior knowledge of how the system responds to inputs. Even the "pure feedback" strategy described above implicitly assumes things about the system response (e.g. that it is approximately linear), and consequently won't work in cases where the system does not respond in the expected way. To control our system *optimally*, we need some way to reliably predict how it will respond to inputs.

This can be done by combining several concepts you may be familiar with from physics: drawing free body diagrams of the forces that act on the mechanism, taking measurements of mass and moment of inertia from your *CAD* models, applying standard equations of how DC motors or pneumatic cylinders convert energy into mechanical force and motion, etc.

The act of creating a consistent mathematical description of your system is called *modeling* your system's behavior. The resulting set of equations are called a *model* of how you expect the system to behave. Not every system requires an explicit model to be controlled (we will see in the turret tutorial that a pure, manually-tuned feedback controller is satisfactory *in some cases*), but an explicit model is *always* helpful.

Note that models do not have to be perfectly accurate to be useful. As we will see in later tuning exercises, even using a simple model of a mechanism can make the tuning effort much simpler.

Obtaining Models for Your Mechanisms

If modeling your mechanism seems daunting, don't worry! Most mechanisms in FRC are modeled by well-studied equations and code for interacting with those models is included in WPILib. Usually, all that is needed is to determine a set of physical parameters (sometimes called "tuning constants" or "gains") that depend on the specific details of your mechanism/robot. These can be estimated theoretically from other known parameters of your system (such as mass, length, and choice of motor/gearbox), or measured from your mechanism's actual behavior through a system identification routine.

When in doubt, ask a mentor or *support resource*!

Theoretical Modeling

ReCalc is an online calculator which estimates physical parameters for a number of common FRC mechanisms. Importantly, it can generate estimate the `kV`, `kA`, and `kG` gains for the WPILib feedforward classes.

The *WPILib system identification tool* supports a "theoretical mode" that can be used to determine PID gains for feedback control from the `kV` and `kA` gains from ReCalc, enabling (in theory) full tuning of a control loop without running any test routines.

Remember, however, that theory is not reality and purely theoretical gains are not guaranteed to work well. There is *never* a substitute for testing.

System Identification

A good way to improve the accuracy of a simple physics model is to perform experiments on the real mechanism, record data, and use the data to *derive* the constants associated with different parts of the model. This is very useful for physical quantities which are difficult or impossible to predict, but easy to measure (ex: friction in a gearbox).

WPILib's system identification tool supports some common FRC mechanisms, including drivetrain. It deploys its own code to the robot to exercise the mechanism, record data, and derive gains for both feedforward and feedback control schemes.

Manual Tuning: What to Do with No Explicit Model

Sometimes, you have to tune a system without at an explicit model. Maybe the system is uniquely complicated, or maybe you're under time constraints and need something that works quickly, even if it doesn't work optimally. Model-based control requires a correct mathematical model of the system, and for better or for worse, we do not always have one.

In such cases, the physical parameters of the control algorithm can be tuned *manually*. This is generally done by systematically "sweeping" the controller gains by hand until the mechanism behaves as expected. Manual tuning can work quickly in cases where only one or two parameters (such as kV and kP) need to be adjusted - however, in more-complicated scenarios it can become a very involved and difficult process.

One common problem with manual tuning is that it can be hard to distinguish a well-founded controller architecture that is not yet tuned properly, from an inappropriate controller architecture that cannot work (for example, it is generally not possible to tune a velocity controller or vertical arm position controller that functions well without a feedforward). In such a case, we can waste a lot of time searching for correct gains, when no such correct gains exist. There is no substitute for understanding the mechanics of the systems being controlled well enough to determine a correct controller architecture for the mechanism, *even if* we do not explicitly use any model-based control methodologies.

The tutorials that follow include simulations that will allow you to perform the manual tuning process on several typical FRC mechanisms. The fundamental concepts that govern which control strategies are valid for each mechanism are covered on the individual mechanism pages; pay close attention to this as you work through the tutorials!

33.2.3 Introduction to DC Motor Feedforward

> **Note**
>
> For a guide on implementing feedforward control in code with WPILib, see *Feedforward Control in WPILib*.

This page explains the conceptual and mathematical workings of WPILib's SimpleMotorFeedforward (and the other related classes).

The Permanent-Magnet DC Motor Feedforward Equation

Recall from earlier that the point of a feedforward controller is to use the known dynamics of a mechanism to make a best guess at the *control effort* required to put the mechanism in the state you want. In order to do this, we need to have some idea of what kind of mechanism we are controlling - that will determine the relationship between *control effort* and *output*, and let us guess at what value of the former will give us the desired value of the latter.

In FRC, the most common system that we're interested in controlling is the *permanent-magnet DC motor*.

These motors have a number of convenient properties that make them particularly easy to control, and ideal for FRC tasks. In particular, they obey a particular relationship between applied voltage, rotor velocity, and rotor acceleration known as a "voltage balance equation".

$$V = K_s \cdot sgn(\dot{d}) + K_v \cdot \dot{d} + K_a \cdot \ddot{d}$$

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).

We can interpret the coefficients in the above equation as follows:

K_s 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.

K_v describes how much voltage is needed to hold (or "cruise") at a given constant velocity while overcoming the *counter-electromotive force* and any additional friction that increases with speed (including *viscous drag* and some *churning losses*). The relationship between speed and voltage (at constant acceleration) is almost entirely linear (for FRC-legal components) because of how permanent-magnet DC motors work.

K_a 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.

For more information, see this paper.

Variants of the Feedforward Equation

Some of WPILib's other feedforward classes 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:

Elevator Feedforward

An elevator consists of a permanent-magnet DC motor attached to a mass under the force of gravity. Compared to the feedforward equation for an unloaded motor, it differs only in the inclusion of a constant K_g term that accounts for the action of gravity:

$$V = K_g + K_s \cdot sgn(\dot{d}) + K_v \cdot \dot{d} + K_a \cdot \ddot{d}$$

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.

Arm Feedforward

An arm consists of a permanent-magnet DC motor attached to a mass on a stick held under the force of gravity. Like the elevator feedforward, it includes a K_g term to account for the effect of gravity - unlike the elevator feedforward, however, this term is multiplied by the cosine of the arm angle (since the gravitational force does not act directly on the motor):

$$V = K_g \cdot cos(\theta) + K_s \cdot sgn(\dot{\theta}) + K_v \cdot \dot{\theta} + K_a \cdot \ddot{\theta}$$

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.

Using the Feedforward

In order to use the feedforward, we need to plug in values for each unknown in the above voltage-balance equation *other than the voltage*. As mentioned *earlier*, the values of the gains K_g, K_v, K_a can be obtained through theoretical modeling with ReCalc. Explicit measurement with *SysId* will yield the aforementioned gains in addition to K_s. That leaves us needing values for velocity, acceleration, and (in the case of the arm feedforward) position.

Typically, these come from our setpoints - remember that with feedforward we are making a "guess" as to the output we need based on where we want the system to be.

For velocity control, this does not pose a problem - we can take the velocity value from our setpoint directly, and if necessary (it can often be omitted in practice) we can infer the acceleration from the difference between the current and previous velocity setpoints.

For position control, however, this can be difficult - except for the arm controller, there's no direct term in the feedforward equation for position. We often have no choice but to calculate our velocity from the difference between the current and previous setpoint positions, and to ignore acceleration entirely. In order to do better, we need to ensure that our setpoints vary *smoothly* according to some set of constraints - this is usually accomplished with a *motion profile*.

33.2.4 Introduction to PID

Note

For a guide on implementing PID control with WPILib, see *PID Control in WPILib*.

This page explains the conceptual and mathematical workings of a PID controller. *A video explanation from WPI is also available*.

What is a PID Controller?

The PID controller is a common *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 term behaves.

First, we'll get some nomenclature for PID controllers out of the way. In a PID context, we use the term *reference* or *setpoint* to mean the desired state of the mechanism, and the term *output* or *process variable* to refer to the measured state of the mechanism. Below are some common variable naming conventions for relevant quantities.

$r(t)$	*setpoint*, *reference*	$u(t)$	*control effort*
$e(t)$	*error*	$y(t)$	*output*, *process variable*

The *error* $e(t)$ is the difference between the *reference* and the *output*, $r(t) - y(t)$.

For those already familiar with PID control, this interpretation may not be consistent with the classical explanation of the P, I, and D terms corresponding to response to "past", "present", and "future" errors. While that model has merit, we will instead be approaching PID control from the viewpoint of modern control theory, as 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*.

Roughly speaking: the proportional term drives the position error to zero, the derivative term drives the velocity error to zero, and the integral term drives the total accumulated error-over-time to zero. All three terms are added together to produce the *control signal*. We'll go into more detail on each of these below.

Note

Throughout the WPILib documentation, you'll see two ways of writing the tunable constants of the PID controller.

For example, for the proportional gain:

- K_p is the standard math-equation-focused way to notate the constant.
- `kP` is a common way to see it written as a variable in software.

Despite the differences in capitalization, the two formats refer to the same concept.

Proportional Term

The *Proportional* term attempts to drive the position error to zero by contributing to the control signal proportionally to the current position error. Intuitively, this tries to move the *output* towards the *reference*.

$$u(t) = K_p e(t)$$

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.

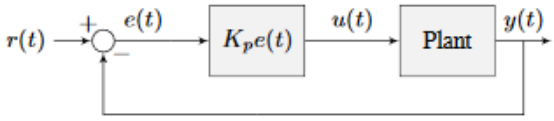


Figure 2.1: P controller block diagram

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 attempts to drive the derivative of the error to zero by contributing to the control signal proportionally to the derivative of the error. Intuitively, this tries to make the *output* move at the same rate as the *reference*.

$$u(t) = K_p e(t) + K_d \frac{de}{dt}$$

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.

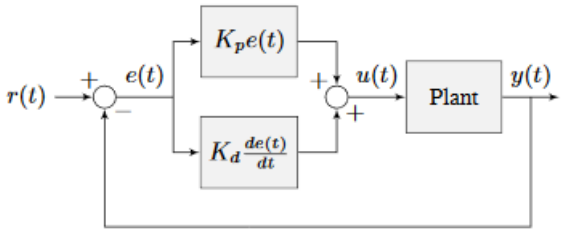


Figure 2.2: PD controller block diagram

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 effort* 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. It is almost always better to use a feedforward controller to eliminate steady-state error. If you do employ integral gain, it is crucial to provide some protection against *integral windup*.

The *Integral* term attempts to drive the total accumulated error to zero by contributing to the control signal proportionally to the sum of all past errors. Intuitively, this tries to drive the *average* of all past *output* values towards the *average* of all past *reference* values.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\, d\tau$$

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.

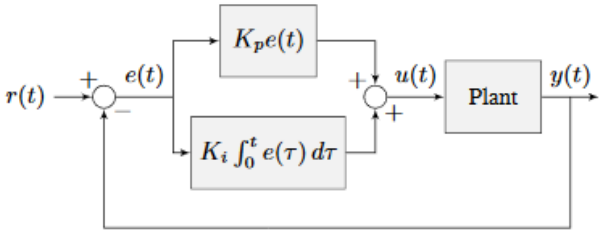


Figure 2.3: PI controller block diagram

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

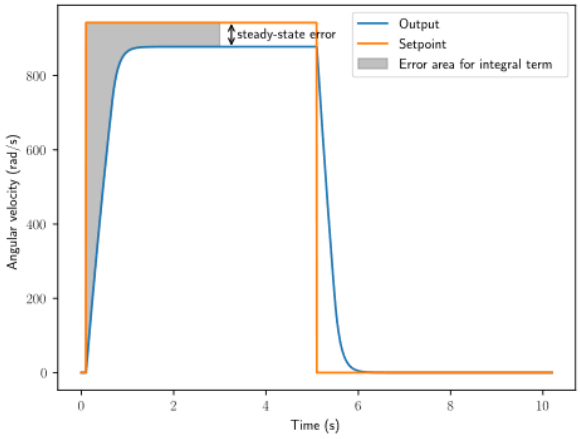


Figure 2.4: P controller with steady-state error

A common way of eliminating *steady-state error* is to integrate the *error* and add it to the *control effort*. 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.

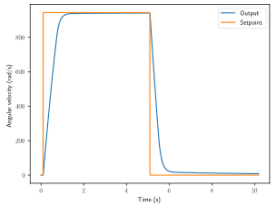


Figure 2.5: PI controller without steady-state error

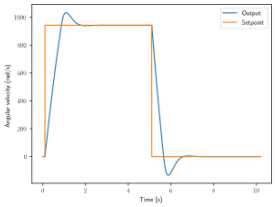


Figure 2.6: PI controller with overshoot from large K_i gain

Putting It All Together

Note

For information on using the WPILib provided PIDController, see the *relevant article*.

When these terms are combined by summing them all together, 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.

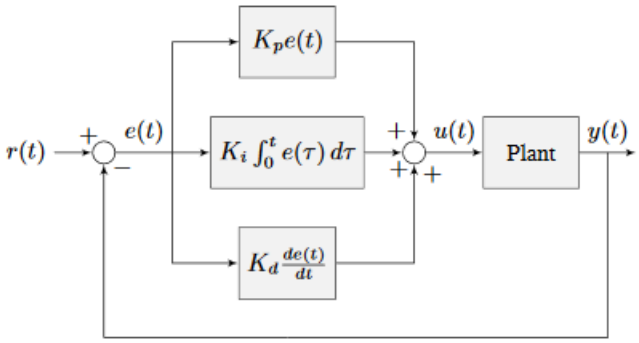


Figure 2.7: PID controller block diagram

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

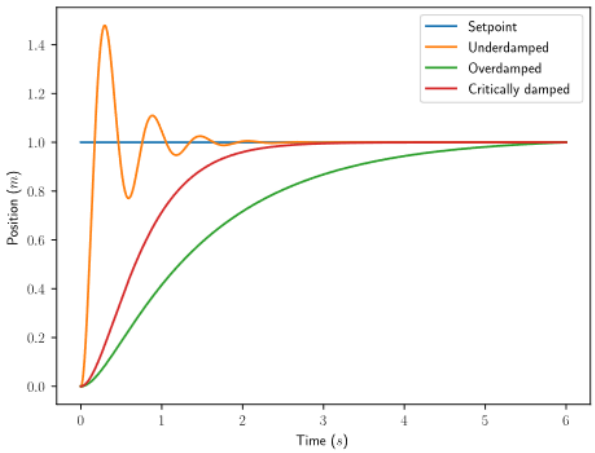


Figure 2.8: PID controller response types

is slow to rise and does not overshoot the *reference*. A *critically damped* response has the fastest *rise time* without overshooting the *reference*.

33.2.5 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.

33.2.6 Introduction To Controls Tuning Tutorials

The WPILib docs include multiple interactive tuning simulations. Their goal is to allow students to learn how tuning parameters impact system behavior, without having to deal with software bugs or other real-world behavior.

Even though WPILib tooling can provide you with optimal gains, it is worth going through the manual tuning process to see how the different control strategies interact with the mechanism.

Ultimately, students should use the examples to build intuition and make their time on the robot more productive.

This page details a few tips while working with the tutorials.

Parameter Exponential Search

While interacting with the simulations, you will get instructions to "increase" or "decrease" different parameters.

When "increasing" a value, multiply it by two until the expected effect is observed. After the first time the value becomes too large (i.e. the behavior is unstable or the mechanism overshoots), reduce the value to halfway between the first too-large value encountered and the previous value tested before that. Continue iterating this "split-half" procedure to zero in on the optimal value (if the response undershoots, pick the halfway point between the new value and the last value immediately above it - if it overshoots, pick the halfway point between the new value and the last value immediately below it). This is called an *exponential search*, and is a very efficient way to find positive values of unknown scale.

System Noise

The "system noise" option introduces random, gaussian error into the plant to provide a more realistic situation of system behavior.

Leave the setting turned off at first to learn the system's ideal behavior. Later, turn it on to see how your tuning works in the presence of real-world effects.

Be Systematic

As seen in *the introduction to PID*, a PID controller has *three* tuned constants.Feedforward components will add even more. This means searching for the "correct" constants manually can be quite difficult - it is therefore necessary to approach the tuning procedure systematically.

Follow the order of tuning presented in the tutorials - it will maximize your chances of success.

Resist checking the tuning solutions until you believe your solution is close to correct. Then check your answer, and try the provided one to compare against your own results.

Furthermore, work from easy to difficult. The tutorials progress in complexity in this order:

1. *Flywheel Mechanisms*
2. *Turret Mechanisms*
3. *Vertical Arm Mechanisms*
4. *Vertical Elevator Mechanisms (with Motion Profiling)*

33.2.7 Tuning a Flywheel Velocity Controller

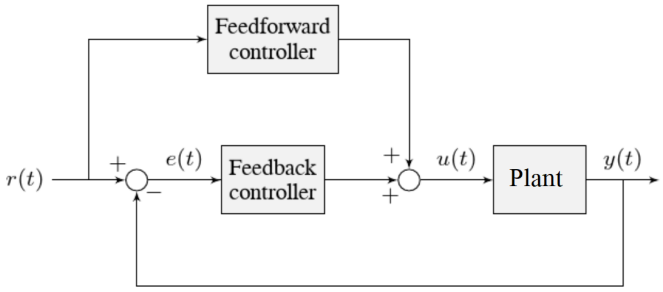
In this section, we will tune a simple velocity controller for a flywheel. The tuning principles explained here will also work for almost any velocity control scenario.

Flywheel Model Description

Our "Flywheel" consists of:

- A rotating inertial mass which launches the game piece (the flywheel)
- A motor (and possibly a gearbox) driving the mass.

For the purposes of this tutorial, this plant is modeled with the same equation used by WPILib's *SimpleMotorFeedforward*, with additional adjustment for sensor delay and gearbox inefficiency. The simulation assumes the plant is controlled by feedforward and feedback controllers, composed in this fashion:



Where:

- The plant's *output* $y(t)$ is the flywheel rotational velocity
- The controller's *setpoint* $r(t)$ is the desired velocity of the flywheel
- The controller's *control effort*, $u(t)$ is the voltage applied to the motor driving the flywheel's motion

Note

A more detailed description of the mathematics of the system *can be found here*.

Picking the Control Strategy for a Flywheel Velocity Controller

In general: the more voltage that is applied to the motor, the faster the flywheel will spin. Once voltage is removed, friction and *back-EMF* oppose the motion and bring the flywheel to a stop.

Flywheels are commonly used to propel game pieces through the air, toward a target. In this simulation, a gamepiece is injected into the flywheel about halfway through the simulation.[1]

To consistently launch a gamepiece, a good first step is to make sure it is spinning at a particular speed before putting a gamepiece into it. Thus, we want to accurately control the velocity of our flywheel.

[1] For this simulation, we model a ball being injected to the flywheel as a velocity-dependant (frictional) torque fighting the spinning of the wheel for one quarter of a wheel rotation, right around the 5 second mark. This is a very simplistic way to model the ball, but is sufficient to illustrate the controller's behavior under a sudden load. It would not be sufficient to predict the ball's trajectory, or the actual "pulldown" in *output* for the system.

> **Note**
>
> This is fundamentally different from the *vertical arm* and *turret* controllers, which both control *position*.

The tutorials below will demonstrate the behavior of the system under bang-bang, pure feedforward, pure feedback (PID), and combined feedforward-feedback control strategies. Follow the instructions to learn how to manually tune these controllers, and expand the "tuning solution" to view an optimal model-based set of tuning parameters.

Bang-Bang Control

Interact with the simulation below to see how the flywheel system responds when controlled by a bang-bang controller.

The "Bang-Bang" controller is a simple controller which applies a binary (present/not-present) force to a mechanism to try to get it closer to a setpoint. A more detailed description (and documentation for the corresponding WPILib implementation) can be found *here.*

There are no tuneable controller parameters for a bang-bang controller - you can only adjust the setpoint. This simplicity is a strength, and also a weakness.

Try adjusting the setpoint up and down. You should see that for almost all values, the output converges to be somewhat near the setpoint.

Common Issues with Bang-Bang Controllers

Note that the system behavior is not perfect, because of delays in the control loop. These can result from the nature of the sensors, measurement filters, loop iteration timers, or even delays in the control hardware itself. Collectively, these cause a cycle of "overshoot" and "undershoot", as the output repeatedly goes above and below the setpoint. This oscillation is unavoidable with a bang-bang controller.

Typically, the steady-state oscillation of a bang-bang controller is small enough that it performs quite well in practice. However, rapid on/off cycling of the control effort can cause mechanical issues - the cycles of rapidly applying and removing forces can loosen bolts and joints, and put a lot of stress on gearboxes.

The abrupt changes in control effort can cause abrupt changes in current draw if the system's inductance is too low. This may stress motor control hardware, and cause eventual damage or failure.

Finally, this technique only works for mechanisms that accelerate relatively slowly. A more in-depth discussion of the details *can be found here*.

Bang-bang control sacrifices a lot for simplicity and high performance (in the sense of fast convergence to the setpoint). To achieve "smoother" control, we need to consider a different control strategy.

Pure Feedforward Control

Interact with the simulation below to see how the flywheel system responds when controlled only by a feedforward controller.

To tune the feedforward controller, increase the velocity feedforward gain K_v until the flywheel approaches the correct setpoint over time. If the flywheel overshoots, reduce K_v.

The exact gain used by the simulation is $K_v = 0.0075$.

We can see that a pure feedforward control strategy works reasonably well for flywheel velocity control. As we mentioned earlier, this is why it's possible to control most motors "directly" with joysticks, without any explicit "control loop" at all. However, we can still do better - the pure feedforward strategy cannot reject disturbances, and so takes a while to recover after the ball is introduced. Additionally, the motor may not perfectly obey the feedforward equation (even after accounting for vibration/noise). To account for these, we need a feedback controller.

Pure Feedback Control

Interact with the simulation below to see how the flywheel system responds when controlled by only a feedback (PID) controller.

Perform the following:

1. Set K_p, K_i, K_d, and K_v to zero.
2. Increase K_p until the *output* starts to oscillate around the *setpoint*, then decrease it until the oscillations stop.
3. *In some cases*, increase K_i if *output* gets "stuck" before converging to the *setpoint*.

> **Note**
>
> PID-only control is not a very good control scheme for flywheel velocity! Do not be surprised if/when the simulation below does not behave well, even when the "optimal" constants are used.

In this particular example, for a setpoint of 300, values of $K_p = 0.1$, $K_i = 0.0$, and $K_d = 0.0$ will produce somewhat reasonable results. Since this control strategy is not very good, it will not work well for all setpoints. You can attempt to improve this behavior by incorporating some K_i, but it is very difficult to achieve good behavior across a wide range of setpoints.

Issues with Feedback Control Alone

Because a non-zero amount of *control effort* is required to keep the flywheel spinning, even when the *output* and *setpoint* are equal, this feedback-only strategy is flawed. In order to optimally control a flywheel, a combined feedforward-feedback strategy is needed.

Combined Feedforward and Feedback Control

Interact with the simulation below to see how the flywheel system responds under simultaneous feedforward and feedback (PID) control.

Tuning the combined flywheel controller is simple - we first tune the feedforward controller following the same procedure as in the feedforward-only section, and then we tune the PID controller following the same procedure as in the feedback-only section. Notice that PID portion of the controller is *much* easier to tune "on top of" an accurate feedforward.

In this particular example, for a setpoint of 300, values of $K_v = 0.0075$ and $K_p = 0.1$ will produce very good results across all setpoints. Small changes to K_p will change the controller behavior to be more or less aggressive - the optimal choice depends on your problem constraints.

Note that the combined feedforward-feedback controller works well across all setpoints, and recovers very quickly after the external disturbance of the ball contacting the flywheel.

Tuning Conclusions

Applicability of Velocity Control

A gamepiece-launching flywheel is one of the most visible applications of velocity control. It is also applicable to drivetrain control - following a pre-defined path in autonomous involves controlling the velocity of the wheels with precision, under a variety of different loads.

Choice of Control Strategies

Because we are controlling velocity, we can achieve fairly good performance with a *pure feedforward controller*. This is because a permanent-magnet DC motor's steady-state velocity is roughly proportional to the voltage applied, and is the reason that you can drive your robot around with joysticks without appearing to use any control loop at all - in that case, you are implicitly using a proportional feedforward model.

Because we must apply a constant control voltage to the motor to maintain a velocity at the setpoint, we cannot successfully use a *pure feedback (PID) controller* (whose output typically disappears when you reach the setpoint) - in order to effectively control velocity, a feedback controller must be *combined with a feedforward controller*.

Bang-bang control can be combined with feedforward control much in the way PID control can - for the sake of brevity we do not include a combined feedforward-bang-bang simulation.

Tuning with only feedback can produce reasonable results in cases where no *control effort* is required to keep the *output* at the *setpoint*. This may work for mechanisms like turrets, or swerve drive steering. However, as seen above, it does not work well for a flywheel, where the back-EMF and friction both act to slow the motor even when it is sustaining motion at the setpoint. To control this system, we need to combine the PID controller with a feedforward controller.

K_d is not useful for velocity control with a constant setpoint - it is only necessary when the setpoint is changing.

Adding an integral gain to the *controller* is often a sub-optimal way to eliminate *steady-state error* - you can see how sloppy and "laggy" it is in the simulation above! As we will see soon, a better approach is to combine the PID controller with a feedforward controller.

Velocity and Position Control

Velocity control also differs from position control in the effect of inertia - in a position controller, inertia tends to cause the mechanism to swing past the setpoint even if the control voltage drops to zero near the setpoint. This makes aggressive control strategies infeasible, as they end up wasting lots of energy fighting self-induced oscillations. In a velocity controller, however, the effect is different - the rotor shaft stops accelerating as soon as you stop applying a control voltage (in fact, it will slow down due to friction and back-EMF), so such overshoots are rare (in fact, overshoot typically occurs in velocity controllers only as a result of loop delay). This enables the use of an extremely simple, extremely aggressive control strategy called *bang-bang control*.

Feedforward Simplifications

For the sake of simplicity, the simulations above omit the K_s term from the WPILib SimpleMotorFeedforward equation. On actual mechanisms, however, this can be important - especially if there's a lot of friction in the mechanism gearing. A flywheel with a lot of static friction will not have a linear control voltage-velocity relationship unless the feedforward controller includes a K_s term to cancel it out.

To measure K_s manually, slowly increase the voltage to the mechanism until it starts to move. The value of K_s is the largest voltage applied before the mechanism begins to move.

Additionally, there is no need for a K_a term in the feedforward for velocity control unless the setpoint is changing - for a flywheel, this is not a concern, and so the gain is omitted here.

Footnotes

33.2.8 Tuning a Turret Position Controller

In this section, we will tune a simple position controller for a turret. The tuning principles explained here will also work for almost any position-control scenarios under no external loading.

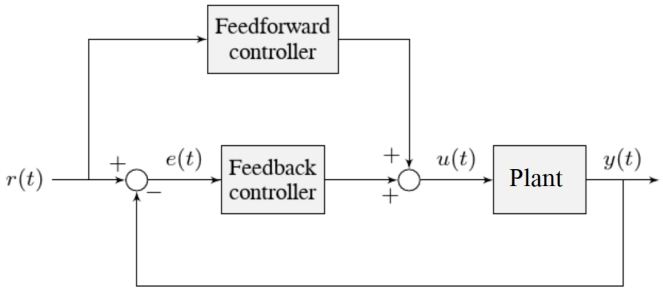
Turret Model Description

A turret rotates some mechanism side-to-side to position it for scoring gamepieces.

Our "turret" consists of:

- A rotating inertial mass (the turret)
- A motor and gearbox driving the mass

For the purposes of this tutorial, this plant is modeled with the same equation used by WPILib's *SimpleMotorFeedforward*, with additional adjustment for sensor delay and gearbox inefficiency. The simulation assumes the plant is controlled by feedforward and feedback controllers, composed in this fashion:



Where:

- The plant's *output* $y(t)$ is the turret's position
- The controller's *setpoint* $r(t)$ is the desired position of the turret
- The controller's *control effort*, $u(t)$ is the voltage applied to the motor driving the turret

Picking the Control Strategy for a Turret Position Controller

In general: the more voltage that is applied to the motor, the faster the motor (and turret) will spin. Once voltage is removed, friction and back-EMF slowly decrease the spinning until the turret stops. We want to make the turret rotate to a given position.

The tutorials below will demonstrate the behavior of the system under pure feedforward, pure feedback (PID), and combined feedforward-feedback control strategies. Follow the instructions to learn how to manually tune these controllers, and expand the "tuning solution" to view an optimal model-based set of tuning parameters. Even though WPILib tooling can provide you with optimal gains, it is worth going through the manual tuning process to see how the different control strategies interact with the mechanism.

This simulation does not include any motion profile generation, so acceleration setpoints are not very well-defined. Accordingly, the *kA* term of the feedforward equation is not used by the controller. This means there will be some amount of delay/lag inherent to the feedforward-only response.

Pure Feedforward Control

Interact with the simulation below to examine how the turret system responds when controlled only by a feedforward controller.

Note

To change the turret setpoint, click on the desired angle along the perimeter of the turret. To command smooth motion, click and drag the setpoint indicator.

To tune the feedforward controller, perform the following:

1. Set K_v to zero.
2. Increase the velocity feedforward gain K_v until the turret tracks the setpoint during smooth, slow motion. If the turret overshoots, reduce the gain.

Note that the turret may "lag" the commanded motion - this is normal, and is fine so long as it moves the correct amount in total.

> **Note**
>
> Feedforward-only control is not a viable control scheme for turrets! Do not be surprised if/when the simulation below does not behave well, even when the "correct" constants are used.

The exact gain used by the plant is $K_v = 0.2$. Note that due to timing inaccuracy in browser simulations, the K_v that works best in the simulation may be somewhat smaller than this.

Issues with Feed-Forward Control Alone

As mentioned above, our simulated mechanism perfectly obeys the WPILib *SimpleMotorFeedforward* equation (as long as the "system noise" option is disabled). We might then expect, like in the case of the *flywheel velocity controller*, that we should be able to achieve perfect convergence-to-setpoint with a feedforward loop alone.

However, our feedforward equation relates *velocity* and *acceleration* to voltage - it allows us to control the *instantaneous motion* of our mechanism with high accuracy, but it does not allow us direct control over the *position*. This is a problem even in our simulation (in which the feedforward equation is the *actual* equation of motion), because unless we employ a *motion profile* to generate a sequence of velocity setpoints we can ask the turret to jump immediately from one position to another. This is impossible, even for our simulated turret.

The resulting behavior from the feedforward controller is to output a single "voltage spike" when the position setpoint changes (corresponding to a single loop iteration of very high velocity), and then zero voltage (because it is assumed that the system has already reached the setpoint). In practice, we can see in the simulation that this results in an initial "impulse" movement towards the target position, that stops at some indeterminate position in-between. This kind of response is called a "kick," and is generally seen as undesirable.

You may notice that *smooth* motion below the turret's maximum achievable speed can be followed accurately in the simulation with feedforward alone. This is misleading, however, because no real mechanism perfectly obeys its feedforward equation. With the "system noise" option enabled, we can see that even smooth, slow motion eventually results in compounding position errors when only feedforward control is used. To accurately converge to the setpoint, we need to use a feedback (PID) controller.

Pure Feedback Control

Interact with the simulation below to examine how the turret system responds when controlled only by a feedback (PID) controller.

Perform the following:

1. Set K_p, K_i, K_d, and K_v to zero.
2. Increase K_p until the mechanism responds to a sudden change in setpoint by moving sharply to the new position. If the controller oscillates too much around the setpoint, reduce *K_p* until it stops.

3. Increase K_d to reduce the amount of "lag" when the controller tries to track a smoothly moving setpoint (reminder: click and drag the turret's directional indicator to move it smoothly). If the controller starts to oscillate, reduce *K_d* until it stops.

Gains of $K_p = 0.3$ and $K_d = 0.05$ yield rapid and stable convergence to the setpoint. Other, similar gains will work nearly as well.

Issues with Feedback Control Alone

Note that even with system noise enabled, the feedback controller is able to drive the turret to the setpoint in a stable manner over time. However, it may not be possible to smoothly track a moving setpoint without lag using feedback alone, as the feedback controller can only respond to errors once they have built up. To get the best of both worlds, we need to combine our feedback controller with a feedforward controller.

Combined Feedforward and Feedback Control

Interact with the simulation below to examine how the turret system responds under simultaneous feedforward and feedback control.

Tuning the combined turret controller is simple - we first tune the feedforward controller following the same procedure as in the feedforward-only section, and then we tune the PID controller following the same procedure as in the feedback-only section. Notice that PID portion of the controller is *much* easier to tune "on top of" an accurate feedforward.

The optimal gains for the combined controller are just the optimal gains for the individual controllers: gains of $K_v = 0.15$, $K_p = 0.3$, and $K_d = 0.05$ yield rapid and stable convergence to the setpoint and relatively accurate tracking of smooth motion. Other, similar gains will work nearly as well.

Once tuned properly, the combined controller should accurately track a smoothly moving setpoint, and also accurately converge to the setpoint over time after a "jump" command.

Tuning Conclusions

Choice of Control Strategies

Like in the case of the *vertical arm*, and unlike the case of the *flywheel*, we are trying to control the *position* rather than the *velocity* of our mechanism.

In the case of the flywheel *velocity* controller we could achieve good control performance with feedforward alone. However, it is very hard to predict how much voltage will cause a certain total change in *position* (time can turn even small errors in velocity into very big errors in position). In this case, we cannot rely on feedforward control alone - as with the vertical arm, we will need a feedback controller.

Unlike in the case of the vertical arm, though, there is no voltage required to keep the mechanism at the setpoint once it's there. As a consequence, it is often possible to effectively control a turret without any feedforward controller at all, relying only on the output of the feedback controller (if the mechanism has a lot of friction, this may not work well and both a feedforward and feedback controller may be needed). Simple position control in the absence of external forces is one of the only cases in which pure feedback control works well.

Controlling a mechanism with only feedback can produce reasonable results in cases where no *control effort* is required to keep the *output* at the *setpoint*. On a turret, this can work acceptably - however, it may still run into problems when trying to follow a moving setpoint, as it relies entirely on the controller transients to control the mechanism's intermediate motion between position setpoints.

We saw in the feedforward-only example above that an accurate feedforward can track slow, smooth velocity setpoints quite well. Combining a feedforward controller with the feedback controller gives the smooth velocity-following of a feedforward controller with the stable long-term error elimination of a feedback controller.

Reasons for Non-Ideal Performance

This simulation does not include any motion profile generation, so acceleration setpoints are not very well-defined. Accordingly, the *kA* term of the feedforward equation is not used by the controller. This means there will be some amount of delay/lag inherent to the feedforward-only response.

A Note on Feedforward and Static Friction

For the sake of simplicity, the simulations above omit the K_s term from the WPILib SimpleMotorFeedforward equation. On actual mechanisms, however, this can be important - especially if there's a lot of friction in the mechanism gearing. A turret with a lot of static friction will be very hard to control accurately with feedback alone - it will get "stuck" near (but not at) the setpoint when the loop output falls below K_s.

To measure K_s manually, slowly increase the voltage to the mechanism until it starts to move. The value of K_s is the largest voltage applied before the mechanism begins to move.

It can be mildly difficult to *apply* the measured K_s to a position controller without motion profiling, as the WPILib SimpleMotorFeedforward class uses the velocity setpoint to determine the direction in which the K_s term should point. To overcome this, either use a motion profile, or else add K_s manually to the output of the controller depending on which direction the mechanism needs to move to get to the setpoint.

33.2.9 Tuning a Vertical Arm Position Controller

In this section, we will tune a simple position controller for a vertical arm. The same tuning principles explained below will work also for almost all position-control scenarios under the load of gravity.

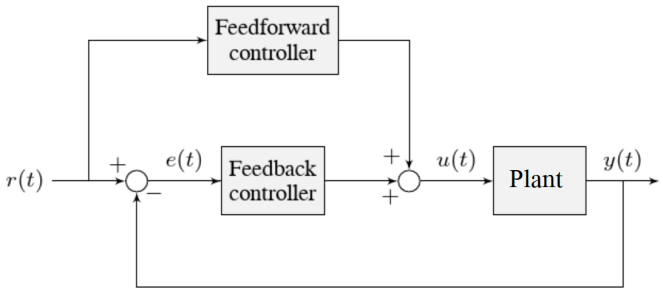
Arm Model Description

Vertical arms are commonly used to lift gamepieces from the ground up to a scoring position. Other similar examples include shooter hoods and elevators.

Our "vertical arm" consists of:

- A mass on a stick, under the force of gravity, pivoting around an axle.
- A motor and gearbox driving the axle to which the mass-on-a-stick is attached

For the purposes of this tutorial, this plant is modeled with the same equation used by WPILib's *ArmFeedforward*, with additional adjustment for sensor delay and gearbox inefficiency. The simulation assumes the plant is controlled by feedforward and feedback controllers, composed in this fashion:



Where:

- The plant's *output* $y(t)$ is the arm's rotational position
- The controller's *setpoint* $r(t)$ is the desired angle of the arm
- The controller's *control effort*, $u(t)$ is the voltage applied to the motor driving the arm

Picking the Control Strategy for a Vertical Arm

Applying voltage to the motor causes a force on the mechanism that drives the arm up or down. If there is no voltage, gravity still acts on the arm to pull it downward. Generally, it is desirable to fight this effect, and keep the arm at a specific angle.

The tutorials below will demonstrate the behavior of the system under pure feedforward, pure feedback (PID), and combined feedforward-feedback control strategies. Follow the instructions to learn how to manually tune these controllers, and expand the "tuning solution" to view an optimal model-based set of tuning parameters. Even though WPILib tooling can provide you with optimal gains, it is worth going through the manual tuning process to see how the different control strategies interact with the mechanism.

Pure Feedforward Control

Interact with the simulation below to examine how the turret system responds when controlled only by a feedforward controller.

Note

To change the arm setpoint, click on the desired angle along the perimeter of the turret. To command smooth motion, click and drag the setpoint indicator.

To tune the feedforward controller, perform the following:

1. Set K_g and K_v to zero.
2. Increase K_g until the arm can hold its position with as little movement as possible. If the arm moves in the opposite direction, decrease K_g until it remains stationary. You will have to zero in on K_g fairly precisely (at least four decimal places).

3. Increase the velocity feedforward gain K_v until the arm tracks the setpoint during smooth, slow motion. If the arm overshoots, reduce the gain. Note that the arm may "lag" the commanded motion - this is normal, and is fine so long as it moves the correct amount in total.

> **Note**
>
> Feedforward-only control is not a viable control scheme for vertical arms! Do not be surprised if/when the simulation below does not behave well, even when the "correct" constants are used.

The exact gains used by the simulation are $K_g = 1.75$ and $K_v = 1.95$.

Issues with Feed-Forward Control Alone

As mentioned above, our simulated mechanism almost-perfectly obeys the WPILib *ArmFeedforward* equation (as long as the "system noise" option is disabled). We might then expect, like in the case of the *flywheel velocity controller*, that we should be able to achieve perfect convergence-to-setpoint with a feedforward loop alone.

However, our feedforward equation relates *velocity* and *acceleration* to voltage - it allows us to control the *instantaneous motion* of our mechanism with high accuracy, but it does not allow us direct control over the *position*. This is a problem even in our simulation (in which the feedforward equation is the *actual* equation of motion), because unless we employ a *motion profile* to generate a sequence of velocity setpoints we can ask the arm to jump immediately from one position to another. This is impossible, even for our simulated arm.

The resulting behavior from the feedforward controller is to output a single "voltage spike" when the position setpoint changes (corresponding to a single loop iteration of very high velocity), and then zero voltage (because it is assumed that the system has already reached the setpoint). In practice, we can see in the simulation that this results in an initial "impulse" movement towards the target position, that stops at some indeterminate position in-between. This kind of response is called a "kick," and is generally seen as undesirable.

You will notice that, once properly tuned, the mechanism can track slow/smooth movement with a surprising amount of accuracy - however, there are some obvious problems with this approach. Our feedforward equation corrects for the force of gravity *at the setpoint* - this results in poor behavior if our arm is far from the setpoint. With the "system noise" option enabled, we can also see that even smooth, slow motion eventually results in compounding position errors when only feedforward control is used. To accurately converge to and remain at the setpoint, we need to use a feedback (PID) controller.

Pure Feedback Control

Interact with the simulation below to examine how the vertical arm system responds when controlled only by a feedback (PID) controller.

Perform the following:

1. Set K_p, K_i, K_d, and K_g to zero.
2. Increase K_p until the mechanism responds to a sudden change in setpoint by moving sharply to the new position. If the controller oscillates too much around the setpoint, reduce *K_p* until it stops.

3. Increase K_i when the *output* gets "stuck" before converging to the *setpoint*.
4. Increase K_d to help the system track smoothly-moving setpoints and further reduce oscillation.

Note

Feedback-only control is not a viable control scheme for vertical arms! Do not be surprised if/when the simulation below does not behave well, even when the "correct" constants are used.

There is no good tuning solution for this control strategy. Values of $K_p = 5$ and $K_d = 1$ yield a reasonable approach to a stable equilibrium, but that equilibrium is not actually at the setpoint!

Issues with Feedback Control Alone

A set of gains that works well for one setpoint will act poorly for a different setpoint.

Adding some integral gain can push us to the setpoint over time, but it's unstable and laggy.

Because a non-zero amount of *control effort* is required to keep the arm at a constant height, even when the *output* and *setpoint* are equal, this feedback-only strategy is flawed. In order to optimally control a vertical arm, a combined feedforward-feedback strategy is needed.

Combined Feedforward and Feedback Control

Interact with the simulation below to examine how the vertical arm system responds under simultaneous feedforward and feedback control.

Tuning the combined arm controller is simple - we first tune the feedforward controller following the same procedure as in the feedforward-only section, and then we tune the PID controller following the same procedure as in the feedback-only section. Notice that PID portion of the controller is *much* easier to tune "on top of" an accurate feedforward.

Combining the feedforward coefficients from our first simulation ($K_g = 1.75$ and $K_v = 1.95$) and the feedback coefficients from our second simulation ($K_p = 5$ and $K_d = 1$) yields a good controller behavior.

Once tuned properly, the combined controller accurately tracks a smoothly moving setpoint, and also accurately converge to the setpoint over time after a "jump" command.

Tuning Conclusions

Choice of Control Strategies

Like in the case of the *turret*, and unlike the case of the *flywheel*, we are trying to control the *position* rather than the *velocity* of our mechanism.

In the case of the flywheel *velocity* controller we could achieve good control performance with feedforward alone. However, it is very hard to predict how much voltage will cause a certain total change in *position* (time can turn even small errors in velocity into very big errors in

position). In this case, we cannot rely on feedforward control alone - as with the vertical arm, we will need a feedback controller.

Unlike in the case of the turret, though, there is a voltage required to keep the mechanism steady at the setpoint (because the arm is affected by the force of gravity). As a consequence, a pure feedback controller will not work acceptably for this system, and a combined feedforward-feedback strategy is needed.

The core reason the feedback-only control strategy fails for the vertical arm is gravity. The external force of gravity requires a constant *control effort* to counteract even when at rest at the setpoint, but a feedback controller does not typically output any control effort when at rest at the setpoint (unless integral gain is used, which we can see clearly in the simulation is laggy and introduces oscillations).

We saw in the feedforward-only example above that an accurate feedforward can track slow, smooth velocity setpoints quite well. Combining a feedforward controller with the feedback controller gives the smooth velocity-following of a feedforward controller with the stable long-term error elimination of a feedback controller.

Reasons for Non-Ideal Performance

This simulation does not include any motion profile generation, so acceleration setpoints are not very well-defined. Accordingly, the *kA* term of the feedforward equation is not used by the controller. This means there will be some amount of delay/lag inherent to the feedforward-only response.

The control law is good, but not perfect. There is usually some overshoot even for smoothly-moving setpoints - this is combination of the lack of K_a in the feedforward (see the note above for why it is omitted here), and some discretization error in the simulation. Attempting to move the setpoint too quickly can also cause the setpoint and mechanism to diverge, which (as mentioned earlier) will result in poor behavior due to the :math:'K_g' term correcting for the wrong force, as it is calculated from the setpoint, not the measurement. Using the measurement to correct for gravity is called "feedback linearization" (as opposed to "feedforward linearization" when the setpoint is used), and can be a better control strategy if your measurements are sufficiently fast and accurate.

A Note on Feedforward and Static Friction

For the sake of simplicity, the simulations above omit the K_s term from the WPILib SimpleMotorFeedforward equation. On actual mechanisms, however, this can be important - especially if there's a lot of friction in the mechanism gearing.

In the case of a vertical arm or elevator, K_s can be somewhat tedious to estimate separately from K_g. If your arm or elevator has enough friction for K_s to be important, it is recommended that you use the *WPILib system identification tool* to determine your system gains.

33.2.10 Tuning a Vertical Elevator with Motion Profiling

In this section, we will tune a simple position controller for a vertical elevator. In addition, we will discuss the advantages of using motion profiling for this situation.

About the Profiler

A *Motion Profiler* is the software logic which, given a goal, commands a changing position, velocity, acceleration which are physically realistic for how a mechanism can move. These position/velocity/acceleration commands can be passed to feedforward and feedback controllers to cause the mechanism to actually achieve the commands.

In this specific example: We assume there are one or more *fixed positions* that our elevator needs to move to. The operator commands which position is desired.

In one periodic loop, the operator's commanded position might instantly snap from one value to a different one. Naturally, we don't expect the elevator to physically move that fast.

Since it is not physically plausible for the elevator to move in that manner, using a motion profiler is useful.

Constraining to realistic motion is a generally good principle for robust controls: don't ask the physical system to do something it physically can't. Having desired velocity and acceleration signals are also useful for supporting a more complex feedforward model.

The result is now two *setpoint* concepts: An "unprofiled" input setpoint which is allowed to change freely, and a "profiled" intermediate setpoint which is constrained to move in a more physically realistic manner.

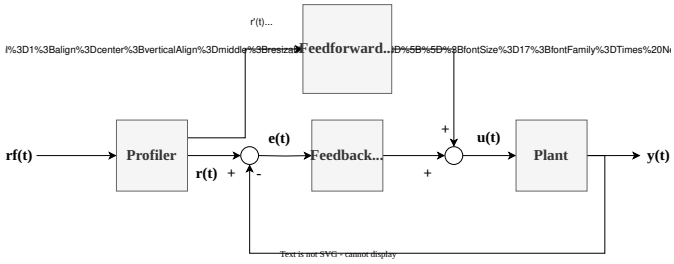
Elevator Model Description

Vertical elevators are commonly used to lift gamepieces from the ground up to a scoring position. Other similar examples include shooter hoods and vertical arms.

Our "vertical elevator" consists of:

- A mass on a carriage, under the force of gravity, traveling up and down in a constrained vertical path
- A motor and gearbox driving a linear chain, to which the mass-on-a-carriage is attached

The simulation assumes the plant (the elevator itself) is controlled by motion profiling, feedforward and feedback controllers, composed in this fashion:



Where:

- The plant's *output* $y(t)$ is the elevator's height
- The controller's *setpoint* $r_f(t)$ is the unprofiled, **final** desired height of the elevator

- The Motion Profiler's position *setpoint* $r(t)$ is where the elevator should currently be positioned
- The Motion Profiler's velocity *setpoint* $r'(t)$ is how fast the elevator should currently be moving
- The Motion Profiler's accelerator *setpoint* $r''(t)$ is how fast the elevator should currently be accelerating
- The controller's *control effort*, $u(t)$ is the voltage applied to the motor driving the elevator

Picking the Control Strategy for a Vertical Elevator

Applying voltage to the motor causes a force on the mechanism that drives the elevator up or down. If there is no voltage, gravity still acts on the elevator to pull it downward. Generally, it is desirable to fight this effect, and keep the elevator at a specific height.

The tutorials below will demonstrate the behavior of the system under just feedback (PID), and then combined feedforward-feedback with motion profiling control strategies. Follow the instructions to learn how to manually tune these controllers, and expand the "tuning solution" to view an optimal model-based set of tuning parameters. Even though WPILib tooling can provide you with optimal gains, it is worth going through the manual tuning process to see how the different control strategies interact with the mechanism.

Feedback Only Control

Interact with the simulation below to examine how the vertical elevator system responds when controlled only by a feedback (PID) controller.

Note

To change the elevator setpoint, click on the desired height along the vertical support.

Perform the following:

1. Set K_p, K_i, and K_d to zero.
2. Increase K_p until the mechanism responds to a sudden change in setpoint by moving sharply to the new position. If the controller oscillates too much around the setpoint, reduce K_p until it stops.
3. Increase K_i when the *output* gets "stuck" before converging to the *setpoint*.
4. Increase K_d to help the system track smoothly-moving setpoints and further reduce oscillation.

Note

Feedback-only control is not a good control scheme for vertical elevators! Do not be surprised if/when the simulation below does not behave consistently, even when the "correct" constants are used.

There is no perfect tuning solution for this control strategy. Values of $K_p = 10.0$, $K_i = 2.5$ and and $K_d = 0.0$ yield a possible solution, but with overshoot and large settling times. Additionally,

it will act very differently depending on the setpoint - aggressively overshooting at the top and undershooting at the bottom.

Motion Profiled, Feedforward, and Feedback Control

Interact with the simulation below to initially examine how the elevator system responds when controlled only by a feedforward controller and then transition to using a little bit of feedback to correct any leftover error.

> **Note**
>
> To change the elevator setpoint, click on the desired height along the vertical support.

To tune the feedforward controller, perform the following:

1. Start with fairly slow maximum velocity and maximum acceleration. 0.3 for both is a good guess.
2. Set K_g, K_v, K_a, K_p, K_i, and K_d to zero.
3. Increase K_g as much as you can without the elevator moving upward. You will have to zero in on K_g fairly precisely (at least two decimal places).
4. Increase the velocity feedforward gain K_v until the straight segments of the elevator actual motion have the same *slope* as the desired motion.
5. Increase the acceleration feedforward gain K_a until the curved segments of the elevator actual motion have the same *curvature* as the desired motion.

At this point, note how with *no sensors involved*, the elevator motion is fairly consistent. With the exception of a small amount of error, we are almost controlling the mechanism without issue.

Only as a last step, add in a bit of feedback gain.

6. Increase K_p until the actual position starts to overshoot the target, then back it off by 20%.

Finally, start to increase the maximum velocity and acceleration. Tweak your feed forward gains if needed.

$K_g = 2.28$, $K_v = 3.07$, $K_a = 0.41$, $K_p = 2.0$ will behave quite well for a range of acceleration, velocities, and setpoints, even in the presence of system noise.

A Note on Feedforward and Static Friction

For the sake of simplicity, the simulations above omit the K_s term from the WPILib SimpleMotorFeedforward equation. On actual mechanisms, however, this can be important - especially if there's a lot of friction in the mechanism gearing.

In the case of a vertical arm or elevator, K_s can be somewhat tedious to estimate separately from K_g. If your arm or elevator has enough friction for K_s to be important, it is recommended that you use the *WPILib system identification tool* to determine your system gains.

33.2.11 Common Control Loop Tuning Issues

There are a number of common issues which can arise while tuning feedforward and feedback controllers.

Integral Term Windup

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 effort*. As a result, the system overshoots and continues to increase until this accumulated error is unwound.

There are a few ways to mitigate this:

1. Decrease the value of K_i, down to zero if possible.
2. Add logic to reset the integrator term to zero if the *output* is too far from the *setpoint*. Some smart motor controllers and WPILib's `PIDController` implement this with a `setIZone()` method.
3. Cap the integrator at some maximum value. WPILib's `PIDController` implements this with the `setIntegratorRange()` method.

Important

Most mechanisms in FRC do not require any integral control, and systems that seem to require integral control to respond well probably have an inaccurate feedforward model.

Voltage Sag

When we operate mechanisms on our robot, we draw current from its battery. This causes the available "bus voltage" that all the robot mechanisms operate off of to drop. This means that the performance of our mechanisms will vary depending on the loading and action of the robot - this is not ideal.

To fix this, most voltage controllers offer a "voltage compensation" setting for their internal control loops that keep the output voltage of the control loops constant despite changes in the bus voltage. The WPILib `MotorController` class offers a `setVoltage` method can do the same thing if the control loops are being run on the RIO (provided you call it every robot loop iteration).

Keep in mind that voltage compensation cannot increase the voltage applied to the motor beyond what is available on the bus - if your actuator is saturating (described below), you'll have to account for that separately.

Actuator Saturation

A controller calculates its output based on the error between the *setpoint* and the current *state*. *Plant* in the real world don't have unlimited control authority available for the controller to apply - that is to say, real mechanisms have some maximum achievable torque/acceleration and velocity.

If our control gains are too aggressive, our control algorithm might try to move the mechanism faster than it is capable of actually going. In this case, the mechanism will "saturate", and behave as if the control gains were smaller than they are. This might adversely affect control response (i.e., result in errors and instability).

If you are encountering problems with actuator saturation, consider modifying your mechanism gearing or powering it with a bigger motor.

33.3 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.

33.3.1 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.

33.3.2 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. The mathematical term for this operation is *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++, , `Python`).

Creating a LinearFilter

Note

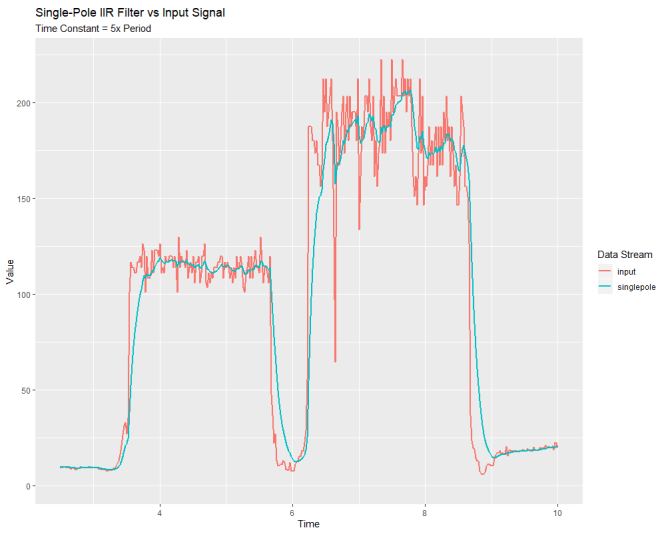
The C++ `LinearFilter` class is templated on the data type used for the input.

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 which performs *exponential smoothing*. This is the "go-to," "first-try" low-pass filter in most

applications; it is computationally trivial and works in most cases.

JAVA

```
// 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);
```

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
frc::LinearFilter<double> filter = frc::LinearFilter<double>::SinglePoleIIR(0.1_s, 0.
↪02_s);
```

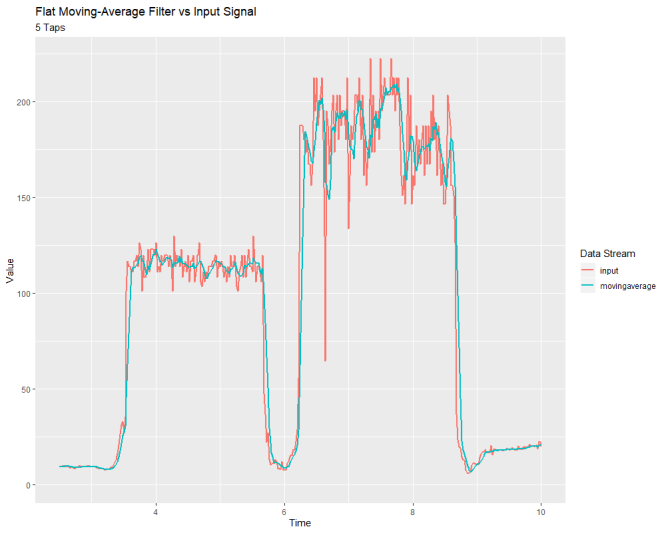
PYTHON

```
from wpimath.filter import LinearFilter
# 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
filter = LinearFilter.singlePoleIIR(0.1, 0.02)
```

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

```
// Creates a new flat moving average filter
// Average will be taken over the last 5 samples
LinearFilter filter = LinearFilter.movingAverage(5);
```

C++

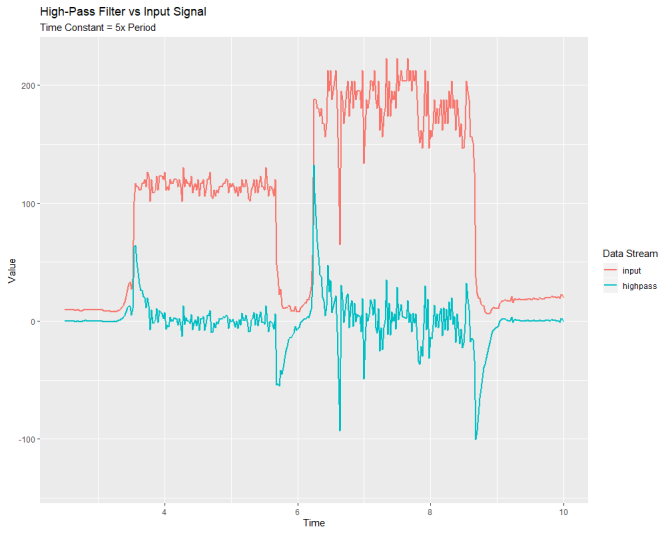
```
// Creates a new flat moving average filter
// Average will be taken over the last 5 samples
frc::LinearFilter<double> filter = frc::LinearFilter<double>::MovingAverage(5);
```

PYTHON

```
from wpimath.filter import LinearFilter
# Creates a new flat moving average filter
# Average will be taken over the last 5 samples
filter = LinearFilter.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

```
// 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);
```

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
frc::LinearFilter<double> filter = frc::LinearFilter<double>::HighPass(0.1_s, 0.02_s);
```

PYTHON

```
from wpimath.filter import LinearFilter
# 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
filter = LinearFilter.highPass(0.1, 0.02)
```

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

```
// Calculates the next value of the output
filter.calculate(input);
```

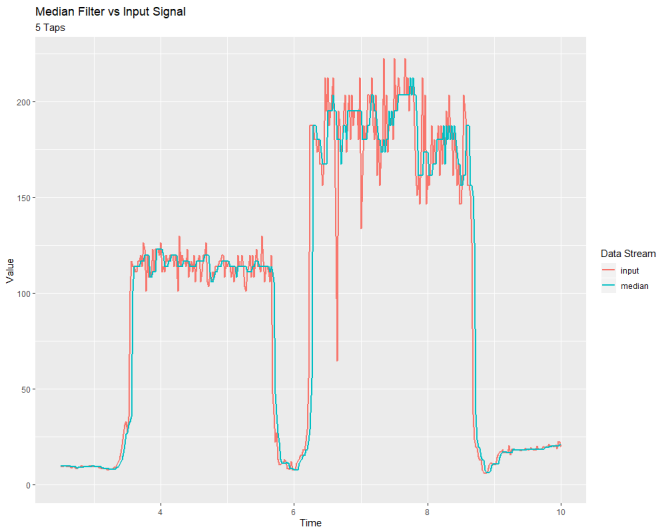
C++

```
// Calculates the next value of the output
filter.Calculate(input);
```

PYTHON

```
# Calculates the next value of the output
filter.calculate(input)
```

33.3.3 Median Filter



A *statistically 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++, , `Python`).

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

```
// Creates a MedianFilter with a window size of 5 samples
MedianFilter filter = new MedianFilter(5);
```

C++

```
// Creates a MedianFilter with a window size of 5 samples
frc::MedianFilter<double> filter(5);
```

PYTHON

```
from wpimath.filter import MedianFilter
# Creates a MedianFilter with a window size of 5 samples
filter = MedianFilter(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

```
// Calculates the next value of the output
filter.calculate(input);
```

C++

```
// Calculates the next value of the output
filter.Calculate(input);
```

PYTHON

```
# Calculates the next value of the output
filter.calculate(input)
```

33.3.4 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++, `Python`).

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

```
// Creates a SlewRateLimiter that limits the rate of change of the signal to 0.5␣
↪units per second
SlewRateLimiter filter = new SlewRateLimiter(0.5);
```

C++

```
// 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};
```

PYTHON

```
from wpimath.filter import SlewRateLimiter
# Creates a SlewRateLimiter that limits the rate of change of the signal to 0.5 units␣
↪per second
filter = SlewRateLimiter(0.5)
```

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

```
// Calculates the next value of the output
filter.calculate(input);
```

C++

```
// Calculates the next value of the output
filter.Calculate(input);
```

PYTHON

```
# Calculates the next value of the output
filter.calculate(input)
```

Using a SlewRateLimiter with DifferentialDrive

> **Note**
>
> The C++ example below templates the filter on `units::scalar` for use with doubles, since joystick values are typically dimensionless.

A typical use of a SlewRateLimiter is to limit the acceleration of a robot's drive. This can be especially handy for robots that are very top-heavy, or that have very powerful drives. To do this, apply a SlewRateLimiter to a value passed into your robot drive function:

JAVA

```
// Ordinary call with no ramping applied
drivetrain.arcadeDrive(forward, turn);
// Slew-rate limits the forward/backward input, limiting forward/backward acceleration
drivetrain.arcadeDrive(filter.calculate(forward), turn);
```

C++

```
// Ordinary call with no ramping applied
drivetrain.ArcadeDrive(forward, turn);
// Slew-rate limits the forward/backward input, limiting forward/backward acceleration
drivetrain.ArcadeDrive(filter.Calculate(forward), turn);
```

PYTHON

```
# Ordinary call with no ramping applied
drivetrain.arcadeDrive(forward, turn)
# Slew-rate limits the forward/backward input, limiting forward/backward acceleration
drivetrain.arcadeDrive(filter.calculate(forward), turn)
```

33.3.5 Debouncer

A debouncer is a filter used to eliminate unwanted quick on/off cycles (termed "bounces," originally from the physical vibrations of a switch as it is thrown). These cycles are usually due to a sensor error like noise or reflections and not the actual event the sensor is trying to record.

Debouncing is implemented in WPILib by the `Debouncer` class (Java, C++, Python), which filters a boolean stream so that the output only changes if the input sustains a change for some nominal time period.

Modes

The WPILib `Debouncer` can be configured in three different modes:

- Rising (default): Debounces rising edges (transitions from *false* to *true*) only.
- Falling: Debounces falling edges (transitions from *true* to *false*) only.
- Both: Debounces all transitions.

Usage

JAVA

```
// Initializes a DigitalInput on DIO 0
DigitalInput input = new DigitalInput(0);
// Creates a Debouncer in "both" mode.
Debouncer m_debouncer = new Debouncer(0.1, Debouncer.DebounceType.kBoth);
// So if currently false the signal must go true for at least .1 seconds before being␣
↪read as a True signal.
if (m_debouncer.calculate(input.get())) {
  // Do something now that the DI is True.
}
```

C++

```
// Initializes a DigitalInput on DIO 0
frc::DigitalInput input{0};
// Creates a Debouncer in "both" mode.
frc::Debouncer m_debouncer{100_ms, frc::Debouncer::DebounceType::kBoth};
// So if currently false the signal must go true for at least .1 seconds before being␣
↪read as a True signal.
if (m_debouncer.calculate(input.Get())) {
  // Do something now that the DI is True.
}
```

PYTHON

```
from wpilib import DigitalInput
from wpimath.filter import Debouncer
# Initializes a DigitalInput on DIO 0
self.input = DigitalInput(0)
# Creates a Debouncer in "both" mode with a debounce time of 0.1 seconds
self.debouncer = Debouncer(0.1, Debouncer.DebounceType.kBoth)
# If currently false, the signal must go true for at least 0.1 seconds before being␣
↪read as a True signal.
if self.debouncer.calculate(self.input.get()):
    # Do something now that the DI is True.
    pass
```

33.4 Geometry Classes

This section covers the geometry classes of WPILib.

33.4.1 Translation, Rotation, and Pose

Translation

Translation in 2 dimensions is represented by WPILib's `Translation2d` class (Java, C++, Python). 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/Python, such as the trajectory generator, make sure to use meters.

Rotation

Rotation in 2 dimensions is represented by WPILib's `Rotation2d` class (Java, C++, Python). 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/Python 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.

> **Note**
>
> `Rotation2d` does not wrap the value of the angle, so if a value of 400 degrees is passed into the constructor, then 400 degrees will be returned in subsequent value calls.

Pose

Pose is a combination of both translation and rotation and is represented by the `Pose2d` class (Java, C++, Python). 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 $\begin{bmatrix} x \\ y \\ \theta \end{bmatrix}$.

33.4.2 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++/Python. Addition adds the two vectors.
- Subtraction: Subtraction between two `Translation2d` can be performed using `minus` in Java, or the binary - operator in C++/Python. 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++/Python. 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++/Python. 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++/Python.

Rotation2d

Transformations for `Rotation2d` are just arithmetic operations on the angle measure represented by the `Rotation2d`.

- `plus` (Java) or + (C++/Python): Adds the rotation component of `other` to this `Rotation2d`'s rotation component
- `minus` (Java) or binary - (C++/Python): Subtracts the rotation component of `other` to this `Rotation2d`'s rotation component
- `unaryMinus` (Java) or unary - (C++/Python): Multiplies the rotation component by a scalar of -1.
- `times` (Java) or * (C++/Python) : Multiplies the rotation component by a scalar.

Transform2d and Twist2d

WPILib provides 2 classes, `Transform2d` (Java, C++, Python), which represents a transformation to a pose, and `Twist2d` (Java, C++, Python) 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.

33.5 System Identification

33.5.1 Introduction to System Identification

What is "System Identification?"

In Control Theory, *system identification* is the process of determining a mathematical model for the behavior of a system through statistical analysis of its inputs and outputs.

This model is a rule describing how input voltage affects the way our measurements (typically encoder data) evolve in time. A "system identification" routine takes such a model and a dataset and attempts to fit parameters which would make your model most closely-match the dataset. Generally, the model is not perfect - the real-world data are polluted by both measurement noise (e.g. timing errors, encoder resolution limitations) and system noise (unmodeled forces acting on the system, like vibrations). However, even an imperfect model is usually "good enough" to give us accurate *feedforward control* of the mechanism, and even to estimate optimal gains for *feedback control*.

Assumed Behavioral Model

If you haven't yet, read the full explanation of the feedforward equations used by the WPILib toolsuite in *The Permanent-Magnet DC Motor Feedforward Equation*.

The process of System Identification is to determine concrete values for the coefficients in the model that best-reflect the behavior of *your particular* real-world system.

To determine numeric values for each coefficient in our model, a curve-fitting technique (such as *least-squares regression*) is applied to measurements taken from the real mechanism. Careful selection of the data-producing experiments helps improve the accuracy of the curve-fitting.

Once these coefficients have been determined, 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:

The WPILib System Identification Tool (SysId)

The WPILib system identification tool consists of the SysId application that runs on the user's PC and a routine that lives in the code running on the user's robot. The routine will generate control signals which user-defined callbacks will send to the motors being characterized, while the robot records data into a log file. After the routine completes, the user will retrieve this file from the roboRIO and load it into SysId. SysId then processes the data and determines model parameters for the user's robot mechanism, as well as producing diagnostic plots.

Included Tools

> **Note**
>
> With a bit of ingenuity, these tools can be used to accurately characterize a surprisingly large variety of robot mechanisms. Even if your mechanism does not seem to obviously match any of the tools, an understanding of the system equations often reveals that one of the included routines will do.

The System Identification toolsuite currently supports:

- Simple Motor Setups
- Elevators
- Arms

Several of these options use nearly identical robot-side code, and differ only in the analysis used by SysId to interpret the data.

Simple Motor Identification

The simple motor identification tool determines the best-fit parameters for the equation:

$$V = K_s \cdot sgn(\dot{d}) + K_v \cdot \dot{d} + K_a \cdot \ddot{d}$$

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.

Elevator Identification

The elevator identification tool determines the best-fit parameters for the equation:

$$V = K_g + K_s \cdot sgn(\dot{d}) + K_v \cdot \dot{d} + K_a \cdot \ddot{d}$$

where V is the applied voltage, d is the displacement (position) of the elevator, $\dot{d}$ is its velocity, and $\ddot{d}$ is its acceleration. The constant term (K_g) is added to correctly account for the effect of gravity.

Arm Identification

The arm identification tool determines the best-fit parameters for the equation:

$$V = K_g \cdot cos(\theta) + K_s \cdot sgn(\dot{\theta}) + K_v \cdot \dot{\theta} + K_a \cdot \ddot{\theta}$$

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 (K_g) is added to correctly account for the effect of gravity.

Installing SysId

SysId is included with the WPILib Installer.

> **Note**
>
> The old Python characterization tool from previous years is no longer supported.

Launching the SysId Tool

The system identification tool can be opened from the `Start Tool` option in VS Code or by using the shortcut inside the WPILib Tools desktop folder (Windows).

33.5.2 Creating an Identification Routine

Types of Tests

A standard motor identification 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. 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). `SysIdRoutine` provides command factories that may be used to run the tests, for example as part of an autonomous routine. Previous versions of SysId used a project generator to create and deploy

robot code to run these tests, but it proved to be very fragile and difficult to maintain. The user code-based workflow enables teams to use mechanism code they already know works, including soft and hard limits.

User Code Setup

> **Note**
>
> Some familiarity with your language's units library is recommended and knowing how to use Consumers is required. This page assumes you are using the Commands framework.

To assist in creating SysId-compatible identification routines, WPILib provides the `SysIdRoutine` class. Users should create a `SysIdRoutine` object, which take both a `Config` object describing the test settings and a `Mechanism` object describing how the routine will control the relevant motors and log the measurements needed to perform the fit.

Routine Config

The `Config` object takes in a a voltage ramp rate for use in Quasistatic tests, a steady state step voltage for use in Dynamic tests, a time to use as the maximum test duration for safety reasons, and a callback method that accepts the current test state (such as "dynamic-forward") for use by a 3rd party logging solution. The constructor may be left blank to default the ramp rate to 1 volt per second and the step voltage to 7 volts.

> **Note**
>
> Not all 3rd party loggers will interact with SysIdRoutine directly. CTRE users who do not wish to use SysIdRoutine directly for logging should use the SignalLogger API and use Tuner X to convert to wpilog. REV users may use Team 6328's Unofficial REV-Compatible Logger (URCL). In both cases the log callback should be set to `null`. Once the log file is in hand, it may be used with SysId just like any other.

The timeout and state callback are optional and defaulted to 10 seconds and null (which will log the data to a normal WPILog file) respectively.

Declaring the Mechanism

The `Mechanism` object takes a voltage consumer, a log consumer, the subsystem being characterized, and the name of the mechanism (to record in the log). The drive callback takes in the routine-generated voltage command and passes it to the relevant motors. The log callback reads the motor voltage, position, and velocity for each relevant motor and adds it to the running log. The subsystem is required so that it may be added to the requirements of the routine commands. The name is optional and will be defaulted to the string returned by getName().

The callbacks can either be created in-place via Lambda expressions or can be their own standalone functions and be passed in via method references. Best practice is to create the routine and callbacks inside the subsystem, to prevent leakage.

JAVA

```
// Creates a SysIdRoutine
SysIdRoutine routine = new SysIdRoutine(
    new SysIdRoutine.Config(),
    new SysIdRoutine.Mechanism(this::voltageDrive, this::logMotors, this)
);
```

PYTHON

```
routine =  commands2.sysid.SysIdRoutine(
    commands2.sysid.SysIdRoutine.Config(),
    commands2.sysid.SysIdRoutine.Mechanism(self.voltageDrive, self.logMotors, self),
)
```

Mechanism Callbacks

The `Mechanism` callbacks are essentially just plumbing between the routine and your motors and sensors.

The `drive` callback exists so that you can pass the requested voltage directly to your motor controller(s).

The `log` callback reads sensors so that the routine can log the voltage, position, and velocity at each timestep.

See the SysIdRoutine (Java, C++) example project for example callbacks.

Test Factories

To be able to run the tests, SysIdRoutine exposes test "factories", or functions that each return a command that will execute a given test.

JAVA

```
public Command sysIdQuasistatic(SysIdRoutine.Direction direction) {
  return routine.quasistatic(direction);
}

public Command sysIdDynamic(SysIdRoutine.Direction direction) {
  return routine.dynamic(direction);
}
```

PYTHON

```
def sysIdQuasistatic(self, direction:commands2.sysid.SysIdRoutine.Direction) ->␣
↪commands2.Command:
  return commands2.sysid.SysIdRoutine.quasistatic(direction)

def sysIdDynamic(self, direction:commands2.sysid.SysIdRoutine.Direction) -> commands2.
↪Command:
  return commands2.sysid.SysIdRoutine.dynamic(direction)
```

Either bind the factory methods to either controller buttons or create an autonomous routine with them. It is recommended to bind them to buttons that the user must hold down for the duration of the test so that the user can stop the routine quickly if it exceeds safe limits.

33.5.3 Running the Identification Routine

Once the code has been deployed, we can now run the system identification routine, and record the resulting data for analysis.

> **Note**
>
> Ensure you have sufficient space around the robot before running any identification routine! The drive identification requires at least 10’ of space, ideally closer to 20’. The robot drive can not be accurately characterized while on blocks.

> **Warning**
>
> Only log files with a single routine in them are usable for analysis. Multiple motors can be run in one routine, but they must be run at the same time. If you run a routine on one motor and then run a routine on another motor without extracting the log or power-cycling the roboRIO in between, analysis will fail.

Running Tests

Perform the tests using the bindings you created in the previous section.

> **Warning**
>
> Watch out for your mechanism and stop the test early if it exceeds safe limits! The routine only creates voltage commands for you to connect to your motors, it is up to you to set up hard or soft limits to prevent injury or damage.

The entire routine should look something like this:

> **Note**
>
> A drivetrain routine is shown below, but the same motions will occur on any mechanism.

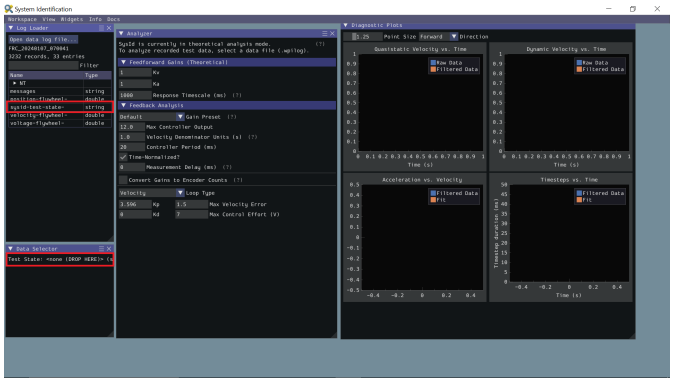
After all four tests have been completed, use the `DataLogTool` to retrieve the log file from the roboRIO.

33.5.4 Loading Data

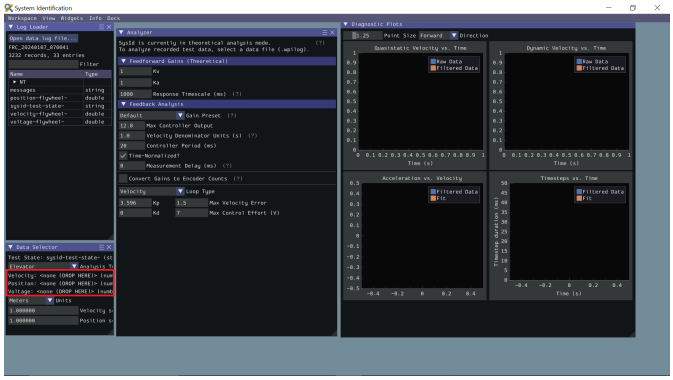
After downloading the WPILog containing the tests from the roboRIO, go to the `Log Loader` pane in SysId and click `Open data log file...`.

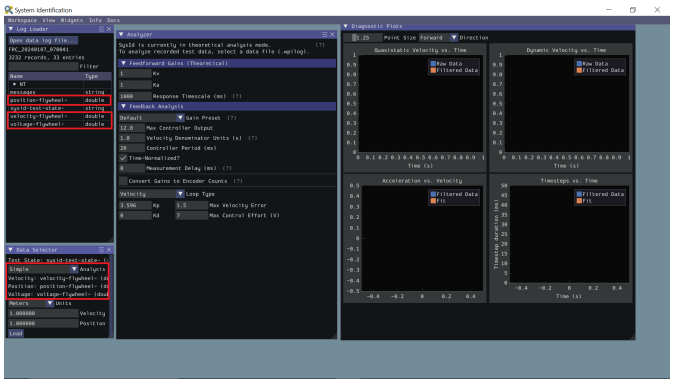
After the file loads, look for a `string` type entry with a name containing "state". Drag this entry into the Data Selector pane's Test State slot.

> **Note**
>
> SysIdRoutine will name the entry "sysid-test-state-mechanism", where "mechanism" is the name passed to the `Mechanism` constructor or the subsystem name.

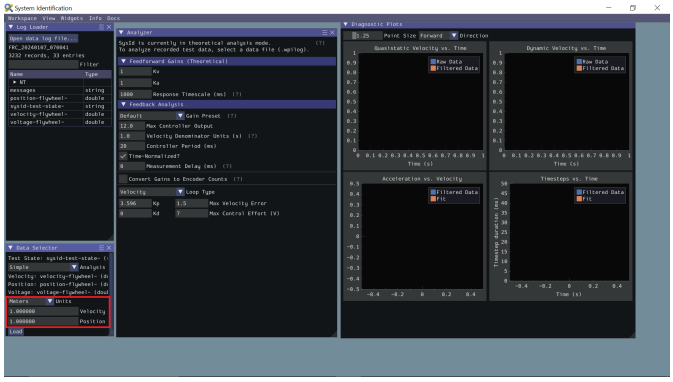


Now the `Data Selector` pane will present `Position`, `Velocity`, and `Voltage` slots. In the `Log Loader` pane, find entries starting with each of those terms and containing the motor name you set in the `log` callback, and drag those into the `Data Selector` slots.





Ideally, the correct units for the position and velocity entries would have been set in the code before running the tests. If this was not the case, use the Units dropdown in the `Data Selector` pane to correct it. Additionally, if you did not account for a gear ratio or some other factor that scales the recorded values up or down uniformly, you can compensate for that by setting position and velocity scaling factors in the provided boxes.

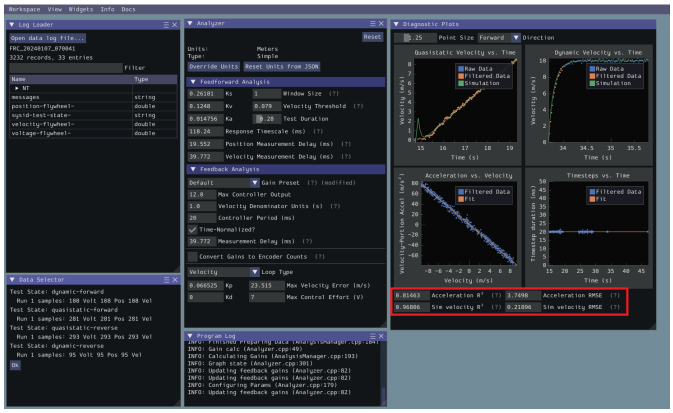


Ensure the correct analysis type has been selected, then click the `Load` button and move on to checking the fit diagnostics in the `Diagnostics` pane.

33.5.5 Viewing Diagnostics

Goodness-of-Fit Metrics

There are three numerical accuracy metrics that are computed with this tool: acceleration *r-squared*, simulated velocity r-squared, and the simulated velocity *RMSE*.



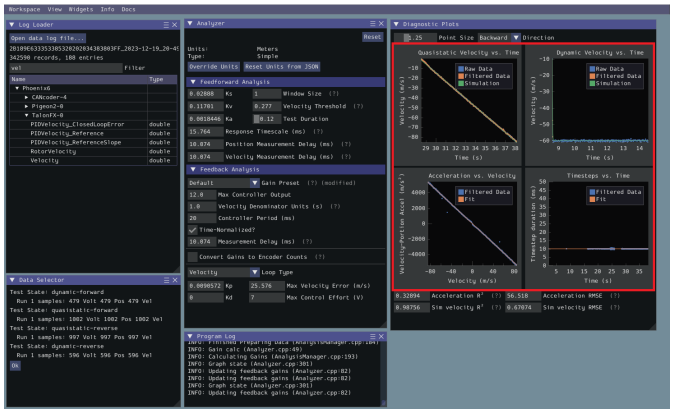
The acceleration r-squared is the fraction of the variance in measured acceleration (used as the independent variable in the SysId regression) explained by the linear model. This can be quite variable, because acceleration is very susceptible to system noise. Mechanisms tend to vibrate quite a bit, so this value rarely goes above `0.5`, even on very good datasets. If the acceleration r-squared goes below around `0.2`, the kA gain will be of dubious quality and the mechanism vibration should be reduced if possible. Even if your r-squared is outside this range it may still be valid, but it is recommended to improve the data if practical.

The simulated velocity r-squared is the fraction of the variance in measured velocity explained by a noiseless simulation of the motor movement stepped forward with the constants determined from the regression. A value north of `.9` indicates a good fit.

The simulated velocity RMSE is the standard deviation of the velocity error from the simulated model. This is a good estimation of the amount of process noise present during the test routine, and can be used as a low-end estimate for the model noise term in *state-space control*.

Diagnostic Plots

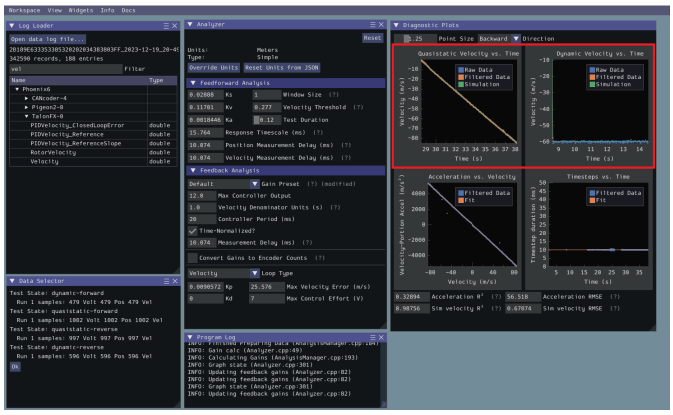
SysId also produces several diagnostic plots to help users evaluate the quality of their model fit.



Time-Domain Plots

> **Note**
>
> To improve plot quality, the diagnostic plots are separated by direction. Be sure to view both the forward *and* backward plots when troubleshooting!

The Time-Domain Diagnostics plots display velocity versus time over the course of the analyzed tests. These should look something like this:



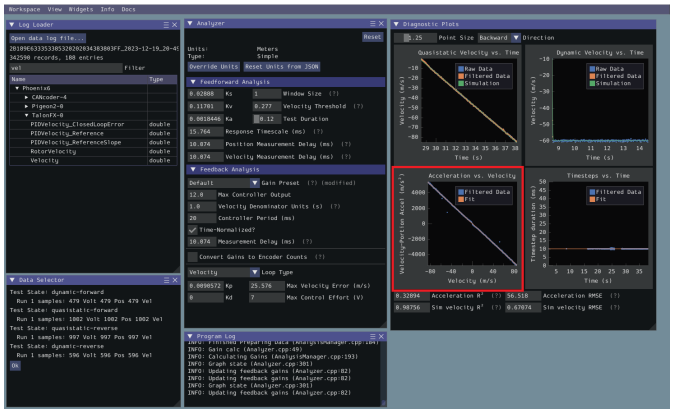
The velocity time domain plots contain three sets of data: Raw Data, Filtered Data, and Simulation. The Raw Data is the recorded data from your robot, the Filtered Data is the data after a median filter has been applied to the data, and the Simulation represents the velocity predictions of a model based off of the feedforward gains from the tool (these are used to calculate the "sim" error metrics mentioned above).

A successful quasistatic graph will be very nearly linear, while a successful dynamic graph will be an approximately exponential approach of the steady-speed.

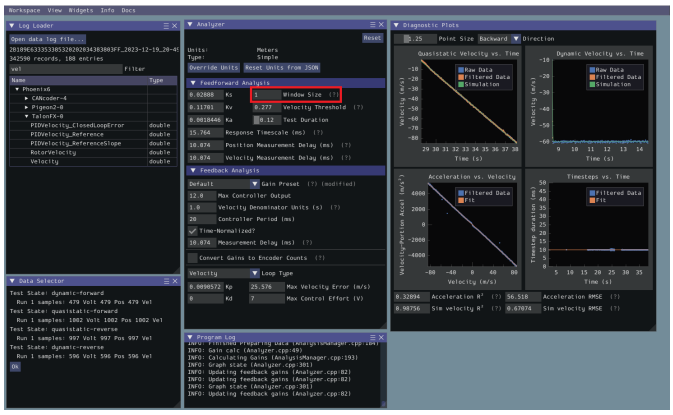
Deviation from this behavior is a sign of an *error*, either in your robot setup, analysis settings, or your test procedure.

Acceleration-Velocity Plot

The acceleration-versus-velocity plot displays the mechanism velocity versus the portion of acceleration corresponding to factors other than friction (ideally, this would leave only back-EMF) and applied voltage across *all* of the tests.



This plot should be quite linear, with patches of relatively noiseless quasistatic data intermixed with quite-noisy dynamic data. The noise on the dynamic sections of the plot may be reduced by increasing the *Window Size* setting.



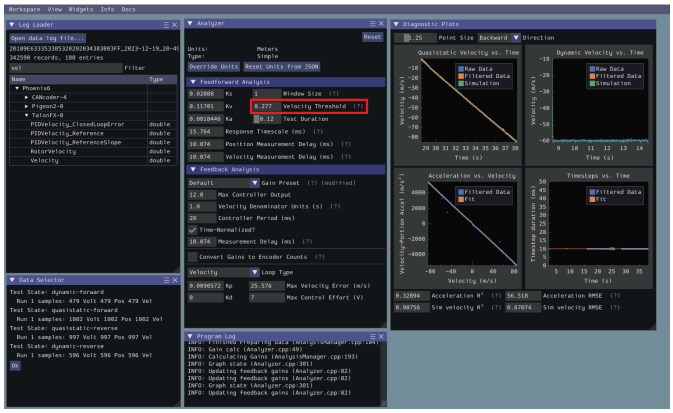
However, if your robot or mechanism has low mass compared to the motor power, this may "eat" what little meaningful acceleration data you have. In these cases kA will tend towards zero and can be ignored for feedforward purposes. However, if kA cannot be accurately measured, the calculated feedback gains are likely to be inaccurate, and manual tuning may be required.

Common Failure Modes

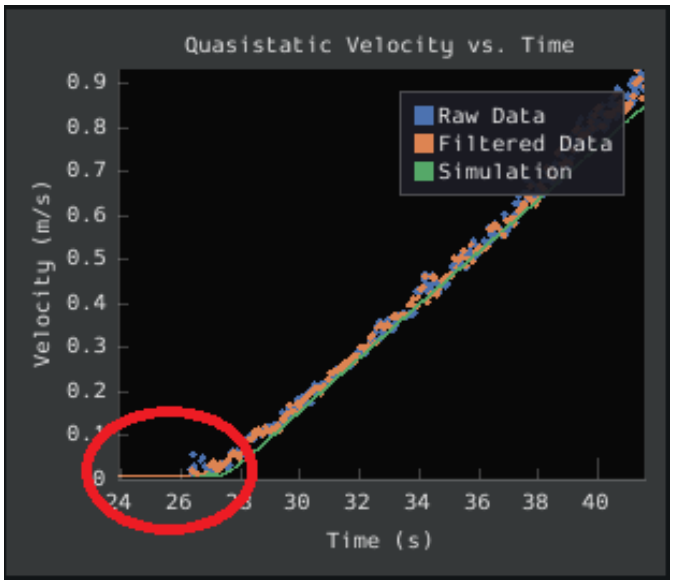
When something has gone wrong with the identification, diagnostic plots, and console output provide crucial clues as to *what* has gone wrong. This section describes some common failures encountered while running the system identification tool, the identifying features of their diagnostic plots, and 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.



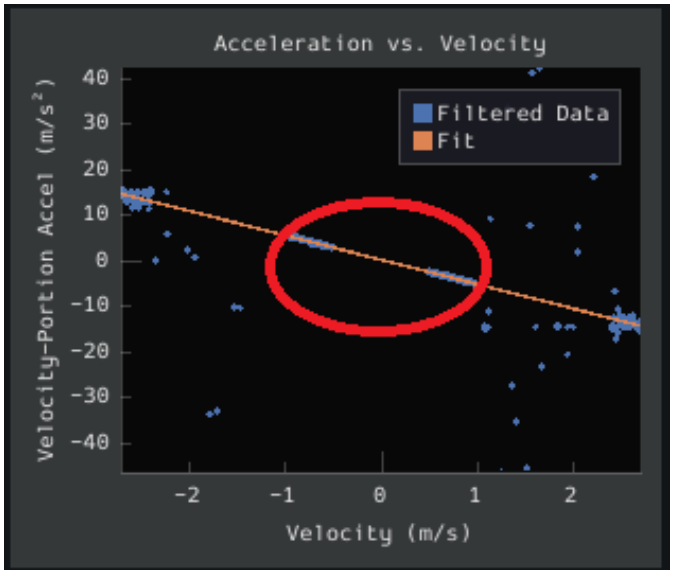
Velocity Threshold Too Low



The presence of a "leading tail" (emphasized by added red circle) in the quasistatic time-domain plot indicates that the *Velocity Threshold* setting is too low, and thus data points from before the robot begins to move are being included.

To solve this, increase the velocity threshold and re-analyze the data.

Motion Threshold Too High



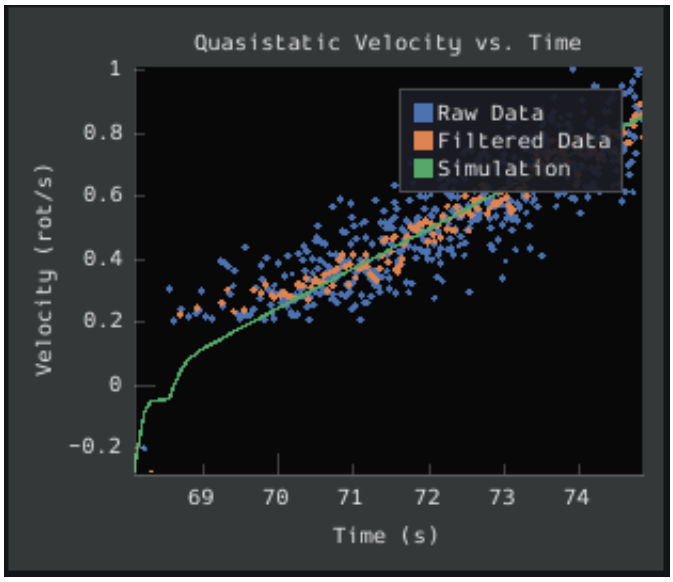
While not nearly as problematic as a too-low threshold, a velocity threshold that is too high will result in a large "gap" in the acceleration-versus-velocity plot.

To solve this, decrease the velocity threshold and re-analyze the data.

Noisy Velocity Signals

Note

There are two types of noise that affect mechanical systems - signal noise and system noise. Signal noise corresponds to measurement error, while system noise corresponds to actual physical motion that is unaccounted-for by your model (e.g. vibration). If SysId suggests that your system is noisy, you must figure out which of the two types of noise is at play - signal noise is often easier to eliminate than system noise.



Many FRC setups suffer from poorly-installed encoders - errors in shaft concentricity (for optical encoders) and magnet location (For magnetic encoders) can both contribute to noisy velocity signals, as can inappropriate filtering settings. Encoder noise will be immediately visible in your diagnostic plots, as can be seen above. Encoder noise is especially common on the toughbox mini gearboxes provided in the kit of parts.

System parameters can sometimes be accurately determined even from data polluted by encoder noise by increasing the window size setting. However, this sort of encoder noise is problematic for robot code much the same way it is problematic for the system identification 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, replacing them with a different type of encoder, or increasing the sample per average in project generation (adds an additional layer of filtering).

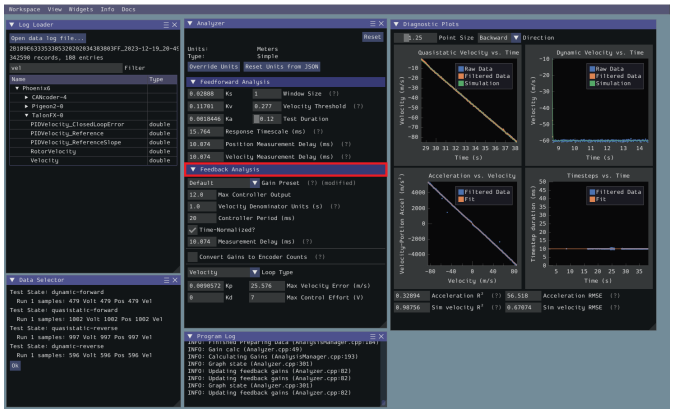
33.5.6 Analyzing Data

Feedforward Analysis

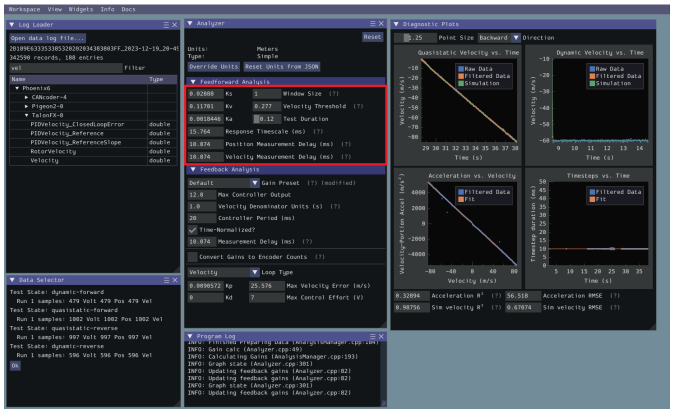
> **Note**
>
> For information on what the calculated feedback gains mean, see *The Permanent-Magnet DC Motor Feedforward Equation*. For information on using the calculated feedback gains in code, see *feedforward control*.

Click the dropdown arrow on the *Feedforward* Section.

> **Note**
>
> If you would like to change units, you will have to press the *Override Units* button and fill out the information on the popup.



The computed mechanism system parameters will then be displayed.

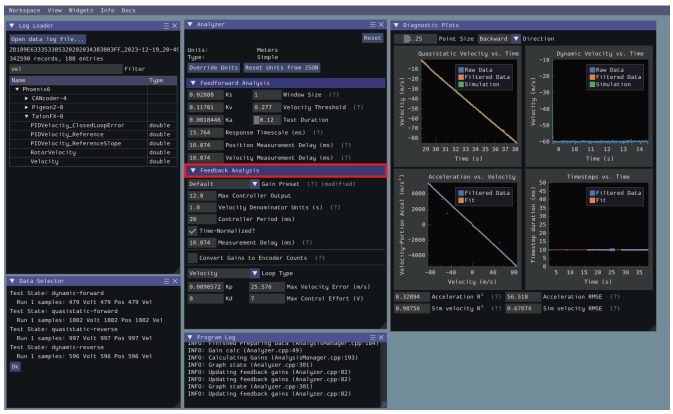


Feedback Analysis

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.

To view the feedback constants, click on the dropdown arrow on the *Feedback* section.



This view can be used to calculate optimal feedback gains for a PD or P controller for your mechanism (via *LQR*).

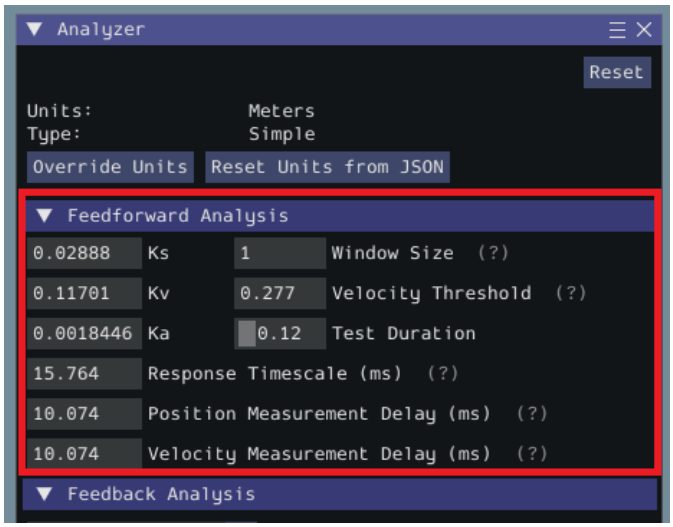
Enter Controller Parameters

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.



- *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.

Measurement Delays

Note

If you are using default smart motor controller settings or WPILib PID Control without additional filtering, SysId handles this for you.

Many "smart motor controllers" (such as the `Talon SRX`, `Venom`, `Talon FX`, and `SPARK MAX`) apply substantial *low-pass filtering* to their encoder velocity measurements, which can introduce a significant amount of phase lag. This can cause the calculated gains for velocity loops to be unstable. This can be accounted for with the *Measurement Delay* box.

However, the measurement delays have already been calculated for the default settings of the previously mentioned motor controllers so for most users this is handled by selecting the right preset in *Gain Settings Preset*.

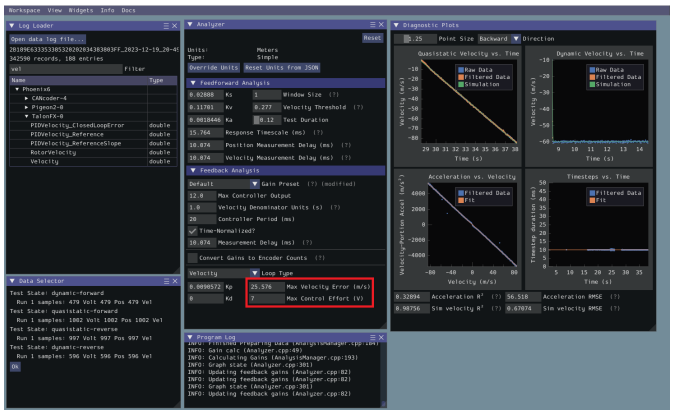
The following only applies if the user decides to implement their own custom filtering settings (e.g. adding a moving average filter to a WPILib PID loop or changing smart motorcontroller measurement period and/or measurement window size) as the measurement delay must be recalculated. Here is the general formula that can be used for filters with moving windows (e.g. median filter + moving average filter):

$$d = \frac{T(n-1)}{2}$$

Where `T` is the period at which measurements are sampled (RIO default is 20 ms) and `n` is the size of the moving window used.

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.

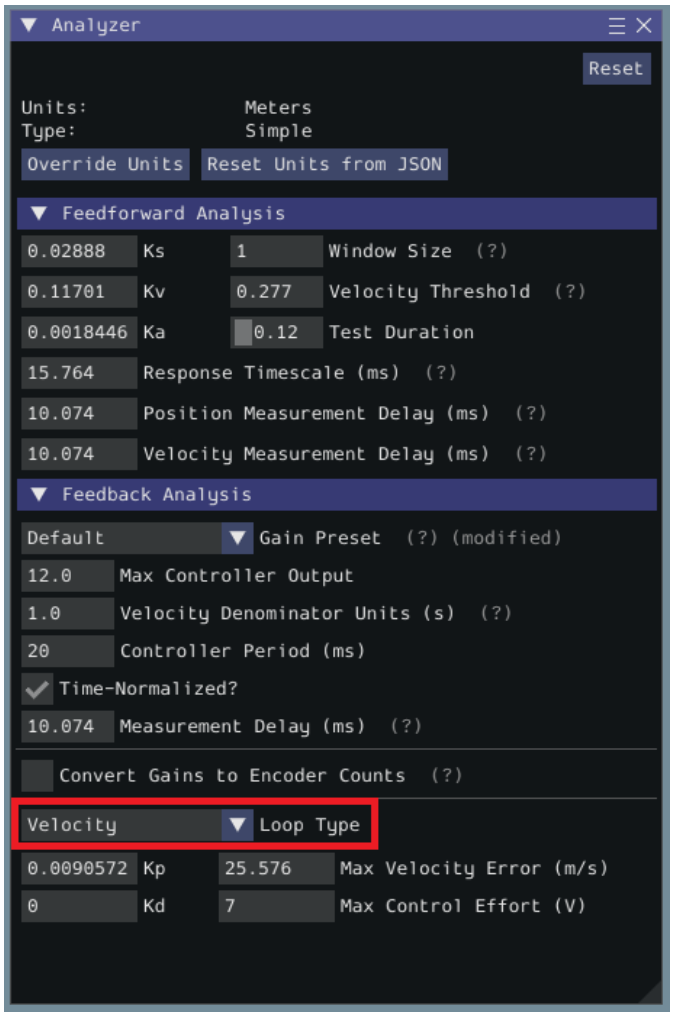


As a rule, smaller values for the *Max Acceptable Error* and larger values for the *Max Acceptable Control Effort* will result in larger gains - this will result in larger control efforts, which can grant better setpoint-tracking but may cause more violent behavior and greater wear on components.

The *Max Acceptable Control Effort* should never exceed 12V, as that corresponds to full battery voltage, and ideally should be somewhat lower than this.

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.



33.5.7 Additional Utilities and Tools

This page mainly covers useful information about additional functionality that this tool provides.

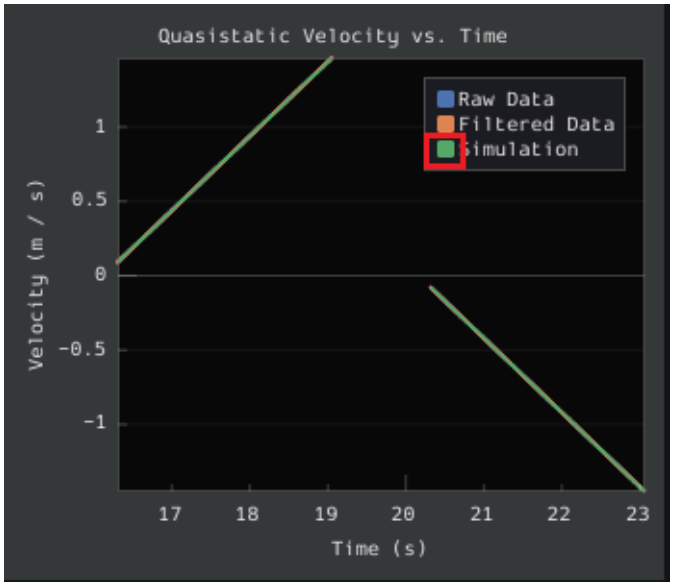
ImGui Tips

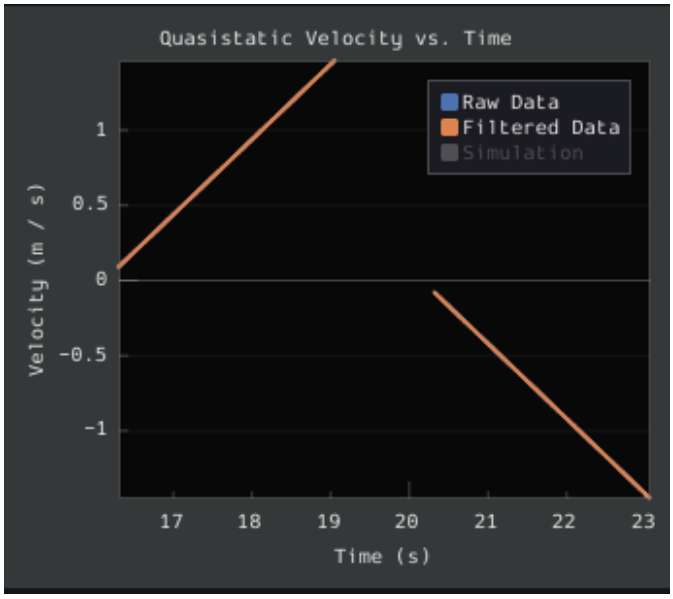
The following are essentially handy features that come with the ImGui framework that SysId uses:

Showing and Hiding Plot Data

To add or remove certain data from the plots, click on the color of the data that you would like to hide or remove.

For example, if we want to hide sim data, we can click the green color box.

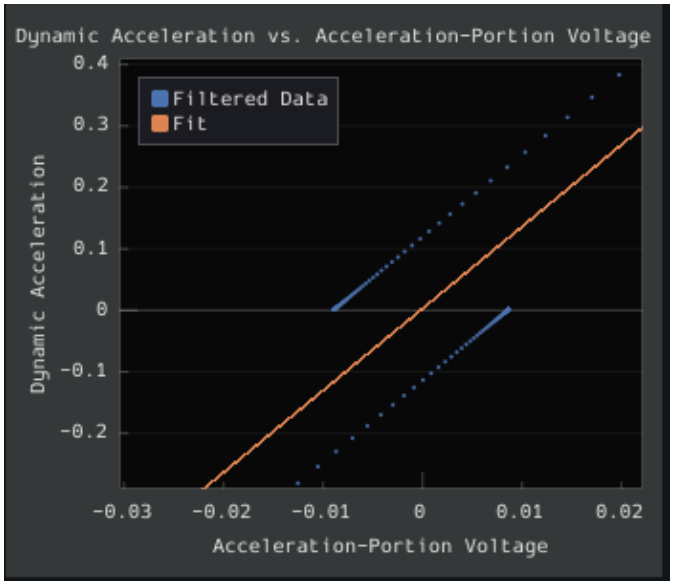




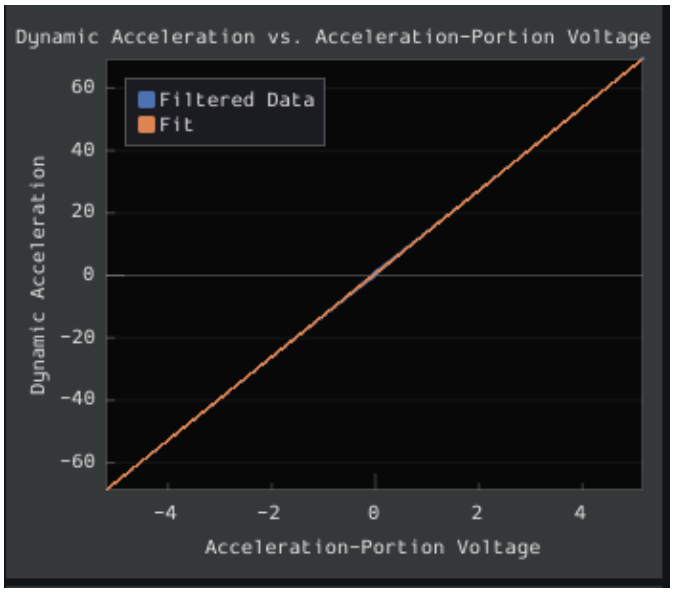
Auto Sizing Plots

If you zoom in to plots and want to revert back to the normally sized plots, just double click on the plot and it will automatically resize it.

Here is a plot that is zoomed in:



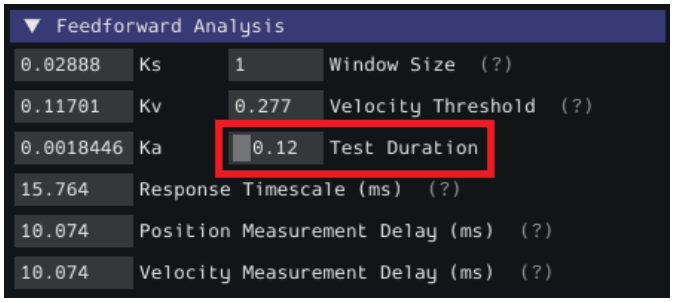
After double clicking, it is automatically resized:



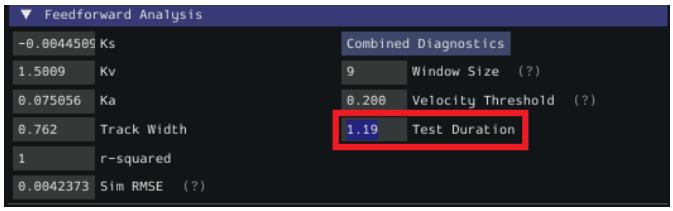
Setting Slider Values

To set the value of a slider as a number rather than sliding the widget, you can CTRL + Click the slider and it will allow you to input a number.

Here is a regular slider:



Here is the input after double clicking the slider:



33.6 Controllers

This section describes various WPILib feedback and feedforward controller classes that are useful for controlling the motion of robot mechanisms, as well as motion-profiling classes that can automatically generate setpoints for use with these controllers.

33.6.1 PID Control in WPILib

> **Note**
>
> This article focuses on in-code implementation of PID control in WPILib. For a conceptual explanation of the working of a PIDController, see *Introduction to PID*

> **Note**
>
> For a guide on implementing PID control through the *command-based framework*, see *PID Control in Command-based*.

WPILib supports PID control of mechanisms through the `PIDController` class (Java, C++, `Python`). 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

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

```
// Creates a PIDController with gains kP, kI, and kD
PIDController pid = new PIDController(kP, kI, kD);
```

C++

```
// Creates a PIDController with gains kP, kI, and kD
frc::PIDController pid{kP, kI, kD};
```

PYTHON

```
from wpimath.controller import PIDController
  # Creates a PIDController with gains kP, kI, and kD
pid = PIDController(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.

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

```
// 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));
```

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));
```

PYTHON

```
# 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

```
// 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();
```

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();
```

PYTHON

```
# 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` class enforces an integrator range limiter to help teams overcome this issue.

By default, the total output contribution from the integral gain is limited to be between -1.0 and 1.0.

The range limits may be increased or decreased using the `setIntegratorRange()` method.

JAVA

```
// The integral gain term will never add or subtract more than 0.5 from
// the total loop output
pid.setIntegratorRange(-0.5, 0.5);
```

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);
```

PYTHON

```
# The integral gain term will never add or subtract more than 0.5 from
# the total loop output
pid.setIntegratorRange(-0.5, 0.5)
```

Disabling Integral Gain if the Error is Too High

Another way integral "wind-up" can be alleviated is by limiting the error range where integral gain is active. This can be achieved by setting `IZone`. If the error is more than `IZone`, the total accumulated error is reset, disabling integral gain. When the error is equal to or less than IZone, integral gain is enabled.

By default, `IZone` is disabled.

`IZone` may be set using the `setIZone()` method. To disable it, set it to infinity.

JAVA

```
// Disable IZone
pid.setIZone(Double.POSITIVE_INFINITY);
  // Integral gain will not be applied if the absolute value of the error is
// more than 2
pid.setIZone(2);
```

C++

```
// Disable IZone
pid.SetIZone(std::numeric_limits<double>::infinity());
  // Integral gain will not be applied if the absolute value of the error is
// more than 2
pid.SetIZone(2);
```

PYTHON

```
# Disable IZone
pid.setIZone(math.inf)
  # Integral gain will not be applied if the absolute value of the error is
# more than 2
pid.setIZone(2)
```

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!

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

```
// Enables continuous input on a range from -180 to 180
pid.enableContinuousInput(-180, 180);
```

C++

```
// Enables continuous input on a range from -180 to 180
pid.EnableContinuousInput(-180, 180);
```

PYTHON

```
# Enables continuous input on a range from -180 to 180
pid.enableContinuousInput(-180, 180)
```

Clamping Controller Output

JAVA

```
// Clamps the controller output to between -0.5 and 0.5
MathUtil.clamp(pid.calculate(encoder.getDistance(), setpoint), -0.5, 0.5);
```

C++

```
// Clamps the controller output to between -0.5 and 0.5
std::clamp(pid.Calculate(encoder.GetDistance(), setpoint), -0.5, 0.5);
```

PYTHON

```
# Python doesn't have a builtin clamp function
def clamp(v, minval, maxval):
    return max(min(v, maxval), minval)
  # Clamps the controller output to between -0.5 and 0.5
clamp(pid.calculate(encoder.getDistance(), setpoint), -0.5, 0.5)
```

33.6.2 Feedforward Control in WPILib

> **Note**
>
> This article focuses on in-code implementation of feedforward control in WPILib. For a conceptual explanation of the feedforward equations used by WPILib, see *Introduction to DC Motor Feedforward*

You may have used feedback control (such as PID) 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.

The WPILib Feedforward Classes

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.

The WPILib feedforward classes closely match the available mechanism characterization tools available in the *SysId toolsuite*. The system identification toolsuite can be used to quickly and effectively determine the correct gains for each type of feedforward. If you are unable to empirically characterize your mechanism (due to space and/or time constraints), reasonable estimates of `kG`, `kV`, and `kA` can be obtained by fairly simple computation, and are also available from ReCalc. `kS` is nearly impossible to model, and must be measured empirically.

WPILib currently provides the following three helper classes for feedforward control:

- *SimpleMotorFeedforward* (Java, C++, `Python`)
- *ArmFeedforward* (Java, C++, `Python`)
- *ElevatorFeedforward* (Java, C++, `Python`)

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.

Note

The API documentation for Python feedforward components indicate which unit is being used as `wpimath.units.NAME`. Users *must* take care to use correct units, as Python 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

```
// Create a new SimpleMotorFeedforward with gains kS, kV, and kA
SimpleMotorFeedforward feedforward = new SimpleMotorFeedforward(kS, kV, kA);
```

C++

```
// Create a new SimpleMotorFeedforward with gains kS, kV, and kA
// Distance is measured in meters
frc::SimpleMotorFeedforward<units::meters> feedforward(kS, kV, kA);
```

PYTHON

```
from wpimath.controller import SimpleMotorFeedforwardMeters
  # Create a new SimpleMotorFeedforward with gains kS, kV, and kA
# Distance is measured in meters
feedforward = SimpleMotorFeedforwardMeters(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

```
// 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);
```

C++

```
// 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);
```

PYTHON

```
# 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, 20)
```

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 `kG` 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.

Note

The API documentation for Python feedforward components indicate which unit is being used as `wpimath.units.NAME`. Users *must* take care to use correct units, as Python 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

```
// Create a new ArmFeedforward with gains kS, kG, kV, and kA
ArmFeedforward feedforward = new ArmFeedforward(kS, kG, kV, kA);
```

C++

```
// Create a new ArmFeedforward with gains kS, kG, kV, and kA
frc::ArmFeedforward feedforward(kS, kG, kV, kA);
```

PYTHON

```
from wpimath.controller import ArmFeedforward
  # Create a new ArmFeedforward with gains kS, kG, kV, and kA
feedforward = ArmFeedforward(kS, kG, 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

```
// 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);
```

C++

```
// 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));
```

PYTHON

```
# 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, 2, 3)
```

ElevatorFeedforward

Note

In C++, 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 * 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.

Note

The API documentation for Python feedforward components indicate which unit is being used as `wpimath.units.NAME`. Users *must* take care to use correct units, as Python 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

```
// Create a new ElevatorFeedforward with gains kS, kG, kV, and kA
ElevatorFeedforward feedforward = new ElevatorFeedforward(kS, kG, kV, kA);
```

C++

```
// Create a new ElevatorFeedforward with gains kS, kV, and kA
// Distance is measured in meters
frc::ElevatorFeedforward feedforward(kS, kG, kV, kA);
```

PYTHON

```
from wpimath.controller import ElevatorFeedforward
  # Create a new ElevatorFeedforward with gains kS, kV, and kA
# Distance is measured in meters
feedforward = ElevatorFeedforward(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

```
// Calculates the feedforward for a 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(20, 30);
```

C++

```
// Calculates the feedforward for a velocity of 20 meters/second
// and an acceleration of 30 meters/second^2
// Output is in volts
feedforward.Calculate(20_mps, 30_mps_sq);
```

PYTHON

```
# Calculates the feedforward for a velocity of 20 meters/second
# and an acceleration of 30 meters/second^2
# Output is in volts
feedforward.calculate(20, 30)
```

Using Feedforward to Control Mechanisms

> **Note**
>
> Since feedforward voltages are physically meaningful, it is best to use the `setVoltage()` (Java, C++, Python) 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

```
public void tankDriveWithFeedforward(double leftVelocity, double rightVelocity) {
  leftMotor.setVoltage(feedforward.calculate(leftVelocity));
  rightMotor.setVoltage(feedForward.calculate(rightVelocity));
}
```

C++

```
void TankDriveWithFeedforward(units::meters_per_second_t leftVelocity,
                              units::meters_per_second_t rightVelocity) {
  leftMotor.SetVoltage(feedforward.Calculate(leftVelocity));
  rightMotor.SetVoltage(feedforward.Calculate(rightVelocity));
}
```

PYTHON

```
def tankDriveWithFeedforward(self, leftVelocity: float, rightVelocity: float):
    self.leftMotor.setVoltage(feedForward.calculate(leftVelocity))
    self.rightMotor.setVoltage(feedForward.calculate(rightVelocity))
```

33.6.3 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 may add any feedforward they like to the output of the controller before sending it to their motors:

JAVA

```
// Adds a feedforward to the loop output before sending it to the motor
motor.setVoltage(pid.calculate(encoder.getDistance(), setpoint) + feedforward);
```

C++

```
// Adds a feedforward to the loop output before sending it to the motor
motor.SetVoltage(pid.Calculate(encoder.GetDistance(), setpoint) + feedforward);
```

PYTHON

```
# 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++, Python) 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

```
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));
}
```

C++

```
void TankDriveWithFeedforwardPID(units::meters_per_second_t leftVelocitySetpoint,
                                 units::meters_per_second_t rightVelocitySetpoint) {
  leftMotor.SetVoltage(feedforward.Calculate(leftVelocitySetpoint)
      + leftPID.Calculate(leftEncoder.getRate(), leftVelocitySetpoint.value()));
  rightMotor.SetVoltage(feedforward.Calculate(rightVelocitySetpoint)
      + rightPID.Calculate(rightEncoder.getRate(), rightVelocitySetpoint.value()));
}
```

PYTHON

```
def tank_drive_with_feedforward_PID(
    left_velocity_setpoint: float,
    right_velocity_setpoint: float,
) -> None:
    leftMotor.setVoltage(
        feedforward.calculate(left_velocity_setpoint)
        + leftPID.calculate(leftEncoder.getRate(), left_velocity_setpoint)
    )
    rightMotor.setVoltage(
        feedforward.calculate(right_velocity_setpoint)
        + rightPID.calculate(rightEncoder.getRate(), right_velocity_setpoint)
    )
```

Other mechanism types can be handled similarly.

33.6.4 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 in Command-based*.

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++, `Python`).

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++, Python):

JAVA

```
// 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);
```

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
frc::TrapezoidProfile<units::meters>::Constraints{10_mps, 20_mps_sq};
```

PYTHON

```
from wpimath.trajectory import TrapezoidProfile
# Creates a new set of trapezoidal motion profile constraints
# Max velocity of 10 meters per second
# Max acceleration of 20 meters per second squared
TrapezoidProfile.Constraints(10, 20)
```

Start and End States

Next, we must specify the desired starting and ending states for our mechanisms using the `TrapezoidProfile.State` class (Java, C++, Python). Each state has a position and a velocity:

JAVA

```
// Creates a new state with a position of 5 meters
// and a velocity of 0 meters per second
new TrapezoidProfile.State(5, 0);
```

C++

```
// 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};
```

PYTHON

```
from wpimath.trajectory import TrapezoidProfile
# Creates a new state with a position of 5 meters
# and a velocity of 0 meters per second
TrapezoidProfile.State(5, 0)
```

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 1 parameter: the constraints.

JAVA

```
// 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
TrapezoidProfile profile = new TrapezoidProfile(new TrapezoidProfile.Constraints(5,␣
↪10));
```

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
frc::TrapezoidProfile<units::meters> profile{
  frc::TrapezoidProfile<units::meters>::Constraints{5_mps, 10_mps_sq}};
```

PYTHON

```
from wpimath.trajectory import TrapezoidProfile
# 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 = TrapezoidProfile(TrapezoidProfile.Constraints(5, 10))
```

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 with the goal state and initial state:

JAVA

```
// Profile will start stationary at zero position
// Profile will end stationary at 5 meters
// Returns the motion profile state after 5 seconds of motion
profile.calculate(5, new TrapezoidProfile.State(0, 0), new TrapezoidProfile.State(5,␣
↪0));
```

C++

```
// Profile will start stationary at zero position
// Profile will end stationary at 5 meters
// Returns the motion profile state after 5 seconds of motion
profile.Calculate(5_s,
frc::TrapezoidProfile<units::meters>::State{0_m, 0_mps},
frc::TrapezoidProfile<units::meters>::State{5_m, 0_mps});
```

PYTHON

```
# Profile will start stationary at zero position
# Profile will end stationary at 5 meters
# Returns the motion profile state after 5 seconds of motion
profile.calculate(5, TrapezoidProfile.State(0, 0), TrapezoidProfile.State(5, 0))
```

Using the State

The `calculate` method returns a `TrapezoidProfile.State` class (the same one that was used to specify the initial/end states when calculating the profile state). To use this for actual control, simply pass the contained position and velocity values to whatever controller you wish (for example, a PIDController):

JAVA

```
var setpoint = profile.calculate(elapsedTime, initialState, goalState);
controller.calculate(encoder.getDistance(), setpoint.position);
```

C++

```
auto setpoint = profile.Calculate(elapsedTime, initialState, goalState);
controller.Calculate(encoder.GetDistance(), setpoint.position.value());
```

PYTHON

```
setpoint = profile.calculate(elapsedTime, initialState, goalState)
controller.calculate(encoder.getDistance(), setpoint.position)
```

Complete Usage Example

Note

In this example, the initial state 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++, Python):

JAVA

```
5  package edu.wpi.first.wpilibj.examples.elevatortrapezoidprofile;
6
7  import edu.wpi.first.math.controller.SimpleMotorFeedforward;
8  import edu.wpi.first.math.trajectory.TrapezoidProfile;
9  import edu.wpi.first.wpilibj.Joystick;
10 import edu.wpi.first.wpilibj.TimedRobot;
11
12 public class Robot extends TimedRobot {
```

(continues on next page)

(continued from previous page)

```
  private static double kDt = 0.02;

  private final Joystick m_joystick = new Joystick(1);
  private final ExampleSmartMotorController m_motor = new␣
↪ExampleSmartMotorController(1);
  // Note: These gains are fake, and will have to be tuned for your robot.
  private final SimpleMotorFeedforward m_feedforward = new SimpleMotorFeedforward(1,␣
↪1.5);

  // Create a motion profile with the given maximum velocity and maximum
  // acceleration constraints for the next setpoint.
  private final TrapezoidProfile m_profile =
      new TrapezoidProfile(new TrapezoidProfile.Constraints(1.75, 0.75));
  private TrapezoidProfile.State m_goal = new TrapezoidProfile.State();
  private TrapezoidProfile.State m_setpoint = new TrapezoidProfile.State();

  public Robot() {
    // Note: These gains are fake, and will have to be tuned for your robot.
    m_motor.setPID(1.3, 0.0, 0.7);
  }

  @Override
  public void teleopPeriodic() {
    if (m_joystick.getRawButtonPressed(2)) {
      m_goal = new TrapezoidProfile.State(5, 0);
    } else if (m_joystick.getRawButtonPressed(3)) {
      m_goal = new TrapezoidProfile.State();
    }

    // Retrieve the profiled setpoint for the next timestep. This setpoint moves
    // toward the goal while obeying the constraints.
    m_setpoint = m_profile.calculate(kDt, m_setpoint, m_goal);

    // Send setpoint to offboard controller PID
    m_motor.setSetpoint(
        ExampleSmartMotorController.PIDMode.kPosition,
        m_setpoint.position,
        m_feedforward.calculate(m_setpoint.velocity) / 12.0);
  }
}
```

C++

```
#include <numbers>

#include <frc/Joystick.h>
#include <frc/TimedRobot.h>
#include <frc/controller/SimpleMotorFeedforward.h>
#include <frc/trajectory/TrapezoidProfile.h>
#include <units/acceleration.h>
#include <units/length.h>
#include <units/time.h>
#include <units/velocity.h>
#include <units/voltage.h>
```

(continues on next page)

(continued from previous page)

```

#include "ExampleSmartMotorController.h"

class Robot : public frc::TimedRobot {
 public:
  static constexpr units::second_t kDt = 20_ms;

  Robot() {
    // Note: These gains are fake, and will have to be tuned for your robot.
    m_motor.SetPID(1.3, 0.0, 0.7);
  }

  void TeleopPeriodic() override {
    if (m_joystick.GetRawButtonPressed(2)) {
      m_goal = {5_m, 0_mps};
    } else if (m_joystick.GetRawButtonPressed(3)) {
      m_goal = {0_m, 0_mps};
    }

    // Retrieve the profiled setpoint for the next timestep. This setpoint moves
    // toward the goal while obeying the constraints.
    m_setpoint = m_profile.Calculate(kDt, m_setpoint, m_goal);

    // Send setpoint to offboard controller PID
    m_motor.SetSetpoint(ExampleSmartMotorController::PIDMode::kPosition,
                        m_setpoint.position.value(),
                        m_feedforward.Calculate(m_setpoint.velocity) / 12_V);
  }

 private:
  frc::Joystick m_joystick{1};
  ExampleSmartMotorController m_motor{1};
  frc::SimpleMotorFeedforward<units::meters> m_feedforward{
      // Note: These gains are fake, and will have to be tuned for your robot.
      1_V, 1.5_V * 1_s / 1_m};

  // Create a motion profile with the given maximum velocity and maximum
  // acceleration constraints for the next setpoint.
  frc::TrapezoidProfile<units::meters> m_profile{{1.75_mps, 0.75_mps_sq}};
  frc::TrapezoidProfile<units::meters>::State m_goal;
  frc::TrapezoidProfile<units::meters>::State m_setpoint;
};

#ifndef RUNNING_FRC_TESTS
int main() {
  return frc::StartRobot<Robot>();
}
#endif
```

PYTHON

```python
import wpilib
import wpimath.controller
from wpimath.trajectory import TrapezoidProfile

import examplesmartmotorcontroller


class MyRobot(wpilib.TimedRobot):
    kDt = 0.02

    def robotInit(self):
        self.joystick = wpilib.Joystick(1)
        self.motor = examplesmartmotorcontroller.ExampleSmartMotorController(1)
        # Note: These gains are fake, and will have to be tuned for your robot.
        self.feedforward = wpimath.controller.SimpleMotorFeedforwardMeters(1, 1.5)

        # Create a motion profile with the given maximum velocity and maximum
        # acceleration constraints for the next setpoint.
        self.profile = TrapezoidProfile(TrapezoidProfile.Constraints(1.75, 0.75))

        self.goal = TrapezoidProfile.State()
        self.setpoint = TrapezoidProfile.State()

        # Note: These gains are fake, and will have to be tuned for your robot.
        self.motor.setPID(1.3, 0.0, 0.7)

    def teleopPeriodic(self):
        if self.joystick.getRawButtonPressed(2):
            self.goal = TrapezoidProfile.State(5, 0)
        elif self.joystick.getRawButtonPressed(3):
            self.goal = TrapezoidProfile.State(0, 0)

        # Retrieve the profiled setpoint for the next timestep. This setpoint moves
        # toward the goal while obeying the constraints.
        self.setpoint = self.profile.calculate(self.kDt, self.setpoint, self.goal)

        # Send setpoint to offboard controller PID
        self.motor.setSetPoint(
            examplesmartmotorcontroller.ExampleSmartMotorController.PIDMode.kPosition,
            self.setpoint.position,
            self.feedforward.calculate(self.setpoint.velocity) / 12,
        )
```

33.6.5 Combining Motion Profiling and PID Control with ProfiledPIDController

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++, Python) 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

```
// 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));
```

C++

```
// 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});
```

PYTHON

```
from wpimath.controller import ProfiledPIDController
from wpimath.trajectory import TrapezoidProfile
  # Creates a ProfiledPIDController
# Max velocity is 5 meters per second
# Max acceleration is 10 meters per second
controller = ProfiledPIDController(
  kP, kI, kD,
  TrapezoidProfile.Constraints(5, 10))
```

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

```
// 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));
```

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));
```

PYTHON

```
# 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, is if 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

```
  public void goToPosition(double goalPosition) {
    double pidVal = m_controller.calculate(m_encoder.getDistance(), goalPosition);
    m_motor.setVoltage(
        pidVal
            + m_feedforward.calculateWithVelocities(
                m_lastSpeed, m_controller.getSetpoint().velocity));
    m_lastSpeed = m_controller.getSetpoint().velocity;
  }
```

C++

```
  void GoToPosition(units::meter_t goalPosition) {
    auto pidVal = m_controller.Calculate(
        units::meter_t{m_encoder.GetDistance()}, goalPosition);
    m_motor.SetVoltage(units::volt_t{pidVal} +
                       m_feedforward.Calculate(
                           m_lastSpeed, m_controller.GetSetpoint().velocity));
    m_lastSpeed = m_controller.GetSetpoint().velocity;
  }
```

PYTHON

```
from wpilib.controller import ProfiledPIDController
from wpilib.controller import SimpleMotorFeedforward
def __init__(self):
    # Assuming encoder, motor, controller are already defined
    self.lastSpeed = 0
    # Assuming feedforward is a SimpleMotorFeedforward object
    self.feedforward = SimpleMotorFeedforward(ks=0.0, kv=0.0, ka=0.0)
def goToPosition(self, goalPosition: float):
    pidVal = self.controller.calculate(self.encoder.getDistance(), goalPosition)
    self.motor.setVoltage(
        pidVal
        + self.feedforward.calculate(self.lastSpeed, self.controller.getSetpoint().
↪velocity))
    self.lastSpeed = self.controller.getSetpoint().velocity
```

Complete Usage Example

A more complete example of `ProfiledPIDController` usage is provided in the ElevatorProfilePID example project (Java, C++, Python):

JAVA

```
5  package edu.wpi.first.wpilibj.examples.elevatorprofiledpid;
6
7  import edu.wpi.first.math.controller.ElevatorFeedforward;
8  import edu.wpi.first.math.controller.ProfiledPIDController;
9  import edu.wpi.first.math.trajectory.TrapezoidProfile;
10 import edu.wpi.first.wpilibj.Encoder;
11 import edu.wpi.first.wpilibj.Joystick;
12 import edu.wpi.first.wpilibj.TimedRobot;
13 import edu.wpi.first.wpilibj.motorcontrol.PWMSparkMax;
14
15 @SuppressWarnings("PMD.RedundantFieldInitializer")
16 public class Robot extends TimedRobot {
17   private static double kDt = 0.02;
18   private static double kMaxVelocity = 1.75;
19   private static double kMaxAcceleration = 0.75;
20   private static double kP = 1.3;
21   private static double kI = 0.0;
22   private static double kD = 0.7;
23   private static double kS = 1.1;
24   private static double kG = 1.2;
25   private static double kV = 1.3;
26
27   private final Joystick m_joystick = new Joystick(1);
28   private final Encoder m_encoder = new Encoder(1, 2);
29   private final PWMSparkMax m_motor = new PWMSparkMax(1);
30
31   // Create a PID controller whose setpoint's change is subject to maximum
32   // velocity and acceleration constraints.
33   private final TrapezoidProfile.Constraints m_constraints =
```

(continues on next page)

(continued from previous page)

```
      new TrapezoidProfile.Constraints(kMaxVelocity, kMaxAcceleration);
  private final ProfiledPIDController m_controller =
      new ProfiledPIDController(kP, kI, kD, m_constraints, kDt);
  private final ElevatorFeedforward m_feedforward = new ElevatorFeedforward(kS, kG,␣
↪kV);

  public Robot() {
    m_encoder.setDistancePerPulse(1.0 / 360.0 * 2.0 * Math.PI * 1.5);
  }

  @Override
  public void teleopPeriodic() {
    if (m_joystick.getRawButtonPressed(2)) {
      m_controller.setGoal(5);
    } else if (m_joystick.getRawButtonPressed(3)) {
      m_controller.setGoal(0);
    }

    // Run controller and update motor output
    m_motor.setVoltage(
        m_controller.calculate(m_encoder.getDistance())
            + m_feedforward.calculate(m_controller.getSetpoint().velocity));
  }
}
```

C++

```
#include <numbers>

#include <frc/Encoder.h>
#include <frc/Joystick.h>
#include <frc/TimedRobot.h>
#include <frc/controller/ElevatorFeedforward.h>
#include <frc/controller/ProfiledPIDController.h>
#include <frc/motorcontrol/PWMSparkMax.h>
#include <frc/trajectory/TrapezoidProfile.h>
#include <units/acceleration.h>
#include <units/length.h>
#include <units/time.h>
#include <units/velocity.h>
#include <units/voltage.h>

class Robot : public frc::TimedRobot {
 public:
  static constexpr units::second_t kDt = 20_ms;

  Robot() {
    m_encoder.SetDistancePerPulse(1.0 / 360.0 * 2.0 * std::numbers::pi * 1.5);
  }

  void TeleopPeriodic() override {
    if (m_joystick.GetRawButtonPressed(2)) {
      m_controller.SetGoal(5_m);
    } else if (m_joystick.GetRawButtonPressed(3)) {
```

(continues on next page)

(continued from previous page)

```
      m_controller.SetGoal(0_m);
    }

    // Run controller and update motor output
    m_motor.SetVoltage(
        units::volt_t{
            m_controller.Calculate(units::meter_t{m_encoder.GetDistance()})} +
        m_feedforward.Calculate(m_controller.GetSetpoint().velocity));
  }

 private:
  static constexpr units::meters_per_second_t kMaxVelocity = 1.75_mps;
  static constexpr units::meters_per_second_squared_t kMaxAcceleration =
      0.75_mps_sq;
  static constexpr double kP = 1.3;
  static constexpr double kI = 0.0;
  static constexpr double kD = 0.7;
  static constexpr units::volt_t kS = 1.1_V;
  static constexpr units::volt_t kG = 1.2_V;
  static constexpr auto kV = 1.3_V / 1_mps;

  frc::Joystick m_joystick{1};
  frc::Encoder m_encoder{1, 2};
  frc::PWMSparkMax m_motor{1};

  // Create a PID controller whose setpoint's change is subject to maximum
  // velocity and acceleration constraints.
  frc::TrapezoidProfile<units::meters>::Constraints m_constraints{
      kMaxVelocity, kMaxAcceleration};
  frc::ProfiledPIDController<units::meters> m_controller{kP, kI, kD,
                                                         m_constraints, kDt};
  frc::ElevatorFeedforward m_feedforward{kS, kG, kV};
};

#ifndef RUNNING_FRC_TESTS
int main() {
  return frc::StartRobot<Robot>();
}
#endif
```

PYTHON

```
import wpilib
import wpimath.controller
import wpimath.trajectory
import math


class MyRobot(wpilib.TimedRobot):
    kDt = 0.02

    def robotInit(self) -> None:
        self.joystick = wpilib.Joystick(1)
        self.encoder = wpilib.Encoder(1, 2)
```

(continues on next page)

(continued from previous page)

```
        self.motor = wpilib.PWMSparkMax(1)

        # Create a PID controller whose setpoint's change is subject to maximum
        # velocity and acceleration constraints.
        self.constraints = wpimath.trajectory.TrapezoidProfile.Constraints(1.75, 0.75)
        self.controller = wpimath.controller.ProfiledPIDController(
            1.3, 0, 0.7, self.constraints, self.kDt
        )

        self.encoder.setDistancePerPulse(1 / 360 * 2 * math.pi * 1.5)

    def teleopPeriodic(self) -> None:
        if self.joystick.getRawButtonPressed(2):
            self.controller.setGoal(5)
        elif self.joystick.getRawButtonPressed(3):
            self.controller.setGoal(0)

        # Run controller and update motor output
        self.motor.set(self.controller.calculate(self.encoder.getDistance()))
```

33.6.6 Bang-Bang Control with BangBangController

The *bang-bang control* algorithm is a control strategy that employs only two states: on (when the measurement is below the setpoint) and off (otherwise). This is roughly equivalent to a proportional loop with infinite gain.

This may initially seem like a poor control strategy, as PID loops are known to become unstable as the gains become large - and indeed, it is a *very bad idea to use a bang-bang controller on anything other than velocity control of a high-inertia mechanism*.

However, when controlling the velocity of high-inertia mechanisms under varying loads (like a shooter flywheel), a bang-bang controller can yield faster recovery time and thus better/more consistent performance than a proportional controller. Unlike an ordinary P loop, a bang-bang controller is *asymmetric* - that is, the controller turns on when the process variable is below the setpoint, and does nothing otherwise. This allows the control effort in the forward direction to be made as large as possible without risking destructive oscillations as the control loop tries to correct a resulting overshoot.

Asymmetric bang-bang control is provided in WPILib by the BangBangController class (Java, C++, Python).

Constructing a BangBangController

Since a bang-bang controller does not have any gains, it does not need any constructor arguments (one can optionally specify the controller tolerance used by `atSetpoint`, but it is not required).

JAVA

```
// Creates a BangBangController
BangBangController controller = new BangBangController();
```

C++

```
// Creates a BangBangController
frc::BangBangController controller;
```

PYTHON

```
from wpimath.controller import BangBangController
  # Creates a BangBangController
controller = BangBangController()
```

Using a BangBangController

> **Warning**
>
> Bang-bang control is an extremely aggressive algorithm that relies on response asymmetry to remain stable. Be *absolutely certain* that your motor controllers have been set to "coast mode" before attempting to control them with a bang-bang controller, or else the braking action will fight the controller and cause potentially destructive oscillation.

Using a bang-bang controller is easy:

JAVA

```
// Controls a motor with the output of the BangBang controller
motor.set(controller.calculate(encoder.getRate(), setpoint));
```

C++

```
// Controls a motor with the output of the BangBang controller
motor.Set(controller.Calculate(encoder.GetRate(), setpoint));
```

PYTHON

```
# Controls a motor with the output of the BangBang controller
motor.set(controller.calculate(encoder.getRate(), setpoint))
```

Combining Bang Bang Control with Feedforward

Like a PID controller, best results are obtained in conjunction with a *feedforward* controller that provides the necessary voltage to sustain the system output at the desired speed, so that the bang-bang controller is only responsible for rejecting disturbances. Since the bang-bang controller can *only* correct in the forward direction, however, it may be preferable to use a slightly conservative feedforward estimate to ensure that the shooter does not over-speed.

JAVA

```
// Controls a motor with the output of the BangBang controller and a feedforward
// Shrinks the feedforward slightly to avoid overspeeding the shooter
motor.setVoltage(controller.calculate(encoder.getRate(), setpoint) * 12.0 + 0.9 *␣
↪feedforward.calculate(setpoint));
```

C++

```
// Controls a motor with the output of the BangBang controller and a feedforward
// Shrinks the feedforward slightly to avoid overspeeding the shooter
motor.SetVoltage(controller.Calculate(encoder.GetRate(), setpoint) * 12.0 + 0.9 *␣
↪feedforward.Calculate(setpoint));
```

PYTHON

```
# Controls a motor with the output of the BangBang controller and a feedforward
motor.setVoltage(controller.calculate(encoder.getRate(), setpoint) * 12.0 + 0.9 *␣
↪feedforward.calculate(setpoint))
```

33.7 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.

33.7.1 Trajectory Generation

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/Python) / `SetStartVelocity(units::meters_per_second_t startVelocity)` (C++)
- `setEndVelocity(double endVelocityMetersPerSecond)` (Java/Python) / `SetEndVelocity(units::meters_per_second_t endVelocity)` (C++)
- `setReversed(boolean reversed)` (Java/Python) / `SetReversed(bool reversed)` (C++)
- `addConstraint(TrajectoryConstraint constraint)` (Java/Python) / `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

```
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);

    var trajectory = TrajectoryGenerator.generateTrajectory(
        sideStart,
        interiorWaypoints,
        crossScale,
        config);
  }
}
```

C++

```
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);
}
```

Python

```python
def generateTrajectory():
    # 2018 cross scale auto waypoints.
    sideStart = Pose2d.fromFeet(1.54, 23.23, Rotation2d.fromDegrees(-180))
    crossScale = Pose2d.fromFeet(23.7, 6.8, Rotation2d.fromDegrees(-160))

    interiorWaypoints = [
        Translation2d.fromFeet(14.54, 23.23),
        Translation2d.fromFeet(21.04, 18.23),
    ]

    config = TrajectoryConfig.fromFps(12, 12)
    config.setReversed(True)

    trajectory = TrajectoryGenerator.generateTrajectory(
        sideStart, interiorWaypoints, crossScale, config
    )
```

Note

The Java code utilizes the Units utility, for easy unit conversions.

Note

Generating a typical trajectory takes about 10 ms to 25 ms. This isn't long, but it's still highly recommended to generate all trajectories on startup in the `Robot` constructor.

Concatenating Trajectories

Trajectories in Java can be combined into a single trajectory using the `concatenate(trajectory)` function. C++/Python users can simply add (+) the two trajectories together.

Warning

It is up to the user to ensure that the end of the initial and start of the appended trajectory match. It is also the user's responsibility to ensure that the start and end velocities of their trajectories match.

JAVA

```
var trajectoryOne =
TrajectoryGenerator.generateTrajectory(
   new Pose2d(0, 0, Rotation2d.fromDegrees(0)),
   List.of(new Translation2d(1, 1), new Translation2d(2, -1)),
   new Pose2d(3, 0, Rotation2d.fromDegrees(0)),
   new TrajectoryConfig(Units.feetToMeters(3.0), Units.feetToMeters(3.0)));
var trajectoryTwo =
TrajectoryGenerator.generateTrajectory(
   new Pose2d(3, 0, Rotation2d.fromDegrees(0)),
   List.of(new Translation2d(4, 4), new Translation2d(6, 3)),
   new Pose2d(6, 0, Rotation2d.fromDegrees(0)),
   new TrajectoryConfig(Units.feetToMeters(3.0), Units.feetToMeters(3.0)));
var concatTraj = trajectoryOne.concatenate(trajectoryTwo);
```

C++

```
auto trajectoryOne = frc::TrajectoryGenerator::GenerateTrajectory(
   frc::Pose2d(0_m, 0_m, 0_rad),
   {frc::Translation2d(1_m, 1_m), frc::Translation2d(2_m, -1_m)},
   frc::Pose2d(3_m, 0_m, 0_rad), frc::TrajectoryConfig(3_fps, 3_fps_sq));
auto trajectoryTwo = frc::TrajectoryGenerator::GenerateTrajectory(
   frc::Pose2d(3_m, 0_m, 0_rad),
   {frc::Translation2d(4_m, 4_m), frc::Translation2d(5_m, 3_m)},
   frc::Pose2d(6_m, 0_m, 0_rad), frc::TrajectoryConfig(3_fps, 3_fps_sq));
auto concatTraj = m_trajectoryOne + m_trajectoryTwo;
```

PYTHON

```
from wpimath.geometry import Pose2d, Rotation2d, Translation2d
from wpimath.trajectory import TrajectoryGenerator, TrajectoryConfig
trajectoryOne = TrajectoryGenerator.generateTrajectory(
   Pose2d(0, 0, Rotation2d.fromDegrees(0)),
   [Translation2d(1, 1), Translation2d(2, -1)],
   Pose2d(3, 0, Rotation2d.fromDegrees(0)),
   TrajectoryConfig.fromFps(3.0, 3.0),
)
trajectoryTwo = TrajectoryGenerator.generateTrajectory(
   Pose2d(3, 0, Rotation2d.fromDegrees(0)),
   [Translation2d(4, 4), Translation2d(6, 3)],
   Pose2d(6, 0, Rotation2d.fromDegrees(0)),
   TrajectoryConfig.fromFps(3.0, 3.0),
)
   concatTraj = trajectoryOne + trajectoryTwo
```

33.7.2 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

```
@Override
public double getMaxVelocityMetersPerSecond(Pose2d poseMeters, double␣
↪curvatureRadPerMeter,
                                            double velocityMetersPerSecond) {
  // code here
}
   @Override
public MinMax getMinMaxAccelerationMetersPerSecondSq(Pose2d poseMeters,
                                                     double curvatureRadPerMeter,
                                                     double velocityMetersPerSecond) {
  // code here
}
```

C++

```
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
}
```

PYTHON

```
from wpimath import units
from wpimath.geometry import Pose2d
from wpimath.trajectory.constraint import TrajectoryConstraint
class MyConstraint(TrajectoryConstraint):
    def maxVelocity(
        self,
        pose: Pose2d,
        curvature: units.radians_per_meter,
        velocity: units.meters_per_second,
    ) -> units.meters_per_second:
        ...
    def minMaxAcceleration(
        self,
        pose: Pose2d,
        curvature: units.radians_per_meter,
        speed: units.meters_per_second,
    ) -> TrajectoryConstraint.MinMax:
        ...
```

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++] (https://github.com/wpilibsuite/allwpilib/tree/v2026.2.1/wpimath/src/main/native/include/frc/trajectory/constraint)) for the WPILib-provided constraints for more examples on how to write your own custom trajectory constraints.

33.7.3 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) / `totalTime` (Python) method can be used to determine the time it takes to traverse the trajectory.

JAVA

```
// Get the total time of the trajectory in seconds
double duration = trajectory.getTotalTimeSeconds();
```

C++

```
// Get the total time of the trajectory
units::second_t duration = trajectory.TotalTime();
```

PYTHON

```
# Get the total time of the trajectory
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/Python) 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

```java
// 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);
```

C++

```cpp
// 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);
```

PYTHON

```python
# Sample the trajectory at 1.2 seconds. This represents where the robot
# should be after 1.2 seconds of traversal.
point = trajectory.sample(1.2)
```

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) / `states` (Python) 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.

33.7.4 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

```
Pose2d bOrigin = new Pose2d(3, 3, Rotation2d.fromDegrees(30));
Trajectory bTrajectory = aTrajectory.relativeTo(bOrigin);
```

C++

```
frc::Pose2d bOrigin{3_m, 3_m, frc::Rotation2d(30_deg)};
frc::Trajectory bTrajectory = aTrajectory.RelativeTo(bOrigin);
```

PYTHON

```
from wpimath.geometry import Pose2d, Rotation2d
bOrigin = Pose2d(3, 3, Rotation2d.fromDegrees(30))
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

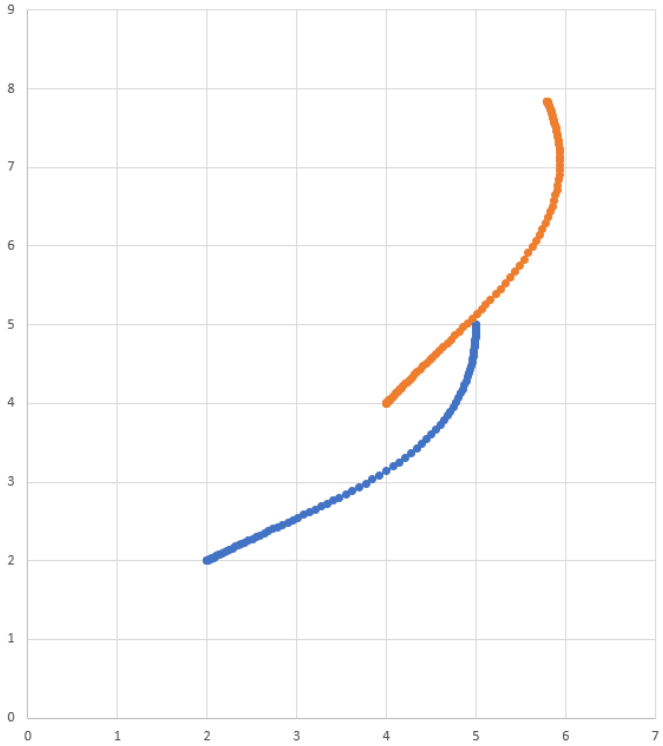
```
Transform2d transform = new Pose2d(4, 4, Rotation2d.fromDegrees(50)).minus(trajectory.
↪getInitialPose());
Trajectory newTrajectory = trajectory.transformBy(transform);
```

C++

```
frc::Transform2d transform = Pose2d(4_m, 4_m, Rotation2d(50_deg)) - trajectory.
↪InitialPose();
frc::Trajectory newTrajectory = trajectory.TransformBy(transform);
```

PYTHON

```
from wpimath.geometry import Pose2d, Rotation2d
transform = Pose2d(4, 4, Rotation2d.fromDegrees(50)) - trajectory.initialPose()
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.

33.7.5 LTV Unicycle Controller

The LTV Unicycle Controller (C++, Java, Python) is a trajectory tracker that is built in to WPILib. This tracker can be used to accurately track trajectories with correction for minor disturbances for differential drivetrains.

> **Note**
>
> LTV has replaced RAMSETE, because it has more intuitive tuning.

Constructing the LTV Unicycle Controller Object

The LTV Unicycle controller should be initialized with four parameters. *qelems* is a vector of the maximum desired error tolerance in the X and Y directions and heading. *relems* is a vector of the desired control effort in linear velocity and angular velocity. *dt* represents the timestep used in calculations (the default loop rate of 20 ms is a reasonable value) and *maxVelocity* should be the max velocity your robot can achieve. See *The State Space control LQR Tuning* for more information on the effect of *qelems* and *relems* on the controller.

The code example below initializes the LTV Unicycle Controller with *qelems* of 0.0625 m in X, 0.125 m in Y, and 2 radians in heading; *relems* of 1 m/s of linear velocity, and 2 rad/sec angular velocity; *dt* of 20 ms; and *maxVelocity* of 9 m/s.

> **Note**
>
> The maximum desired error of the heading is much larger then X and Y to ensure that the heading controller doesn't overpower the Y controller.

JAVA

```
LTVUnicycleController controller = new LTVUnicycleController(VecBuilder.fill(0.0625,␣
↪0.125, 2.0), VecBuilder.fill(1.0, 2.0), 0.02, 9);
```

C++

```
frc::LTVUnicycleController controller{{0.0625, 0.125, 2.0}, {1.0, 2.0}, 0.02_s, 9_mps}
↪;
```

PYTHON

```
controller = LTVUnicycleController([0.0625, 0.125, 2.0], [1.0, 2.0], 0.02, 9)
```

Getting Velocity Commands

The LTV Unicycle controller returns linear and angular velocity commands which, if followed, will make the robot accurately reach a goal state consisting of a desired pose, desired linear velocity, and desired angular velocity.

> **Note**
>
> The "reference pose" represents the position that the robot should be at a particular timestep when tracking the trajectory. It does NOT represent the final endpoint of the trajectory, which is the goal.

The controller can be updated using the `Calculate` (C++) / `calculate` (Java/Python) method. The first overload of this method accepts the current robot pose, desired robot pose, desired linear velocity, and desired angular velocity. The second overload accepts the current robot pose and desired pose, linear velocity, and angular velocity as a `Trajectory.State` object. This overload is ideal for those using the WPILib trajectory API.

JAVA

```
Trajectory.State reference = trajectory.sample(3.4); // sample the trajectory at 3.4␣
↪seconds from the beginning
ChassisSpeeds adjustedSpeeds = controller.calculate(currentRobotPose, reference);
```

C++

```
const Trajectory::State reference = trajectory.Sample(3.4_s); // sample the␣
↪trajectory at 3.4 seconds from the beginning
ChassisSpeeds adjustedSpeeds = controller.Calculate(currentRobotPose, reference);
```

PYTHON

```
reference = trajectory.sample(3.4)  # sample the trajectory at 3.4 seconds from the␣
↪beginning
adjustedSpeeds = controller.calculate(currentRobotPose, reference)
```

These calculations should be performed at every loop iteration, with an updated robot position and reference.

Using the Velocity Commands

The velocity commands 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 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

```
ChassisSpeeds adjustedSpeeds = controller.calculate(currentRobotPose, reference);
DifferentialDriveWheelSpeeds wheelSpeeds = kinematics.toWheelSpeeds(adjustedSpeeds);
double left = wheelSpeeds.leftMetersPerSecond;
double right = wheelSpeeds.rightMetersPerSecond;
```

C++

```
ChassisSpeeds adjustedSpeeds = controller.Calculate(currentRobotPose, reference);
DifferentialDriveWheelSpeeds wheelSpeeds = kinematics.ToWheelSpeeds(adjustedSpeeds);
auto [left, right] = kinematics.ToWheelSpeeds(adjustedSpeeds);
```

PYTHON

```
adjustedSpeeds = controller.calculate(currentRobotPose, reference)
wheelSpeeds = kinematics.toWheelSpeeds(adjustedSpeeds)
left = wheelSpeeds.left
right = wheelSpeeds.right
```

These new left and right velocities are still speeds and not voltages, so two PID Controllers, one for each side, should be used to track them. You can use either the WPILib PIDController (C++, Java, Python) or the Velocity PID feature on smart motor controllers such as the TalonSRX and the SPARK MAX.

33.7.6 Ramsete Controller

> **Warning**
>
> Ramsete Controller has been *deprecated*. Use *LTV Unicycle Controller* which has more intuitive tuning.

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 for differential drivetrains.

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

```
// 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);
```

C++

```
// 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};
```

PYTHON

```
from wpimath.controller import RamseteController
# Using the default constructor of RamseteController. Here
# the gains are initialized to 2.0 and 0.7.
controller1 = RamseteController()
# Using the secondary constructor of RamseteController where
# the user can choose any other gains.
controller2 = RamseteController(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 reference, which is where it should be at the current time. The reference 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 "reference pose" represents the position that the robot should be at a particular timestep when tracking the trajectory. It does NOT represent the final endpoint of the trajectory, which is the goal.

The controller can be updated using the `Calculate` (C++) / `calculate` (Java/Python) 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 reference as three separate parameters (pose, linear velocity, and angular velocity) whereas the other overload accepts a `Trajectory.State` object, which contains information about the reference pose. For its ease, users should use the latter method when tracking trajectories.

JAVA

```
Trajectory.State reference = trajectory.sample(3.4); // sample the trajectory at 3.4␣
↪seconds from the beginning
ChassisSpeeds adjustedSpeeds = controller.calculate(currentRobotPose, reference);
```

C++

```
const Trajectory::State reference = trajectory.Sample(3.4_s); // sample the␣
↪trajectory at 3.4 seconds from the beginning
ChassisSpeeds adjustedSpeeds = controller.Calculate(currentRobotPose, reference);
```

PYTHON

```
reference = trajectory.sample(3.4)  # sample the trajectory at 3.4 seconds from the␣
↪beginning
adjustedSpeeds = controller.calculate(currentRobotPose, reference)
```

These calculations should be performed at every loop iteration, with an updated robot position and reference.

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

```
ChassisSpeeds adjustedSpeeds = controller.calculate(currentRobotPose, reference);
DifferentialDriveWheelSpeeds wheelSpeeds = kinematics.toWheelSpeeds(adjustedSpeeds);
double left = wheelSpeeds.leftMetersPerSecond;
double right = wheelSpeeds.rightMetersPerSecond;
```

C++

```
ChassisSpeeds adjustedSpeeds = controller.Calculate(currentRobotPose, reference);
DifferentialDriveWheelSpeeds wheelSpeeds = kinematics.ToWheelSpeeds(adjustedSpeeds);
auto [left, right] = kinematics.ToWheelSpeeds(adjustedSpeeds);
```

PYTHON

```
adjustedSpeeds = controller.calculate(currentRobotPose, reference)
wheelSpeeds = kinematics.toWheelSpeeds(adjustedSpeeds)
left = wheelSpeeds.left
right = wheelSpeeds.right
```

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, Python) 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*.

33.7.7 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

```
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.
```

C++

```
frc::HolonomicDriveController controller{
  frc::PIDController{1, 0, 0}, frc::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.
```

PYTHON

```python
from wpimath.controller import (
   HolonomicDriveController,
   PIDController,
   ProfiledPIDControllerRadians,
)
from wpimath.trajectory import TrapezoidProfileRadians
controller = HolonomicDriveController(
   PIDController(1, 0, 0),
   PIDController(1, 0, 0),
   ProfiledPIDControllerRadians(
      1, 0, 0, TrapezoidProfileRadians.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.
```

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 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/Python) 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

```java
// 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));
```

C++

```
// Sample the trajectoty 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);
```

PYTHON

```
from wpimath.geometry import Rotation2d
# Sample the trajectory at 3.4 seconds from the beginning.
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).
adjustedSpeeds = controller.calculate(
   currentRobotPose, goal, Rotation2d.fromDegrees(70.0)
)
```

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

```
SwerveModuleState[] moduleStates = kinematics.toSwerveModuleStates(adjustedSpeeds);
SwerveModuleState frontLeft = moduleStates[0];
SwerveModuleState frontRight = moduleStates[1];
SwerveModuleState backLeft = moduleStates[2];
SwerveModuleState backRight = moduleStates[3];
```

C++

```
auto [fl, fr, bl, br] = kinematics.ToSwerveModuleStates(adjustedSpeeds);
```

PYTHON

```
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.

33.7.8 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.

Note

Sending your robot pose and trajectory to field2d can help verify that your robot is driving correctly relative to the robot trajectory.

1. Set up your code to record your robot's position after each odometry update:

JAVA

```
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());
    var translation = m_odometry.getPoseMeters().getTranslation();
    m_xEntry.setNumber(translation.getX());
    m_yEntry.setNumber(translation.getY());
}
```

C++

```
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().value());
    m_yEntry.SetDouble(translation.Y().value());
}
```

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, so try increasing the track width number or measuring it programmatically.

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

```
123             new PIDController(DriveConstants.kPDriveVel, 0, 0),
124             new PIDController(DriveConstants.kPDriveVel, 0, 0),
```

C++

```
79      frc::PIDController{DriveConstants::kPDriveVel, 0, 0},
80      frc::PIDController{DriveConstants::kPDriveVel, 0, 0},
```

1. Next, we want to disable the Ramsete controller to make it easier to isolate our problematic behavior. To do so, simply call `setEnabled(false)` on the `RamseteController` passed into your `RamseteCommand`:

JAVA

```
RamseteController m_disabledRamsete = new RamseteController();
m_disabledRamsete.setEnabled(false);
// Be sure to pass your new disabledRamsete variable
RamseteCommand ramseteCommand = new RamseteCommand(
    exampleTrajectory,
    m_robotDrive::getPose,
    m_disabledRamsete,
    ...
);
```

C++

```
frc::RamseteController m_disabledRamsete;
m_disabledRamsete.SetEnabled(false);
// Be sure to pass your new disabledRamsete variable
frc2::RamseteCommand ramseteCommand(
  exampleTrajectory,
  [this]() { return m_drive.GetPose(); },
  m_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

```
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");
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
);
```

C++

```
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");
frc::PIDController leftController(DriveConstants::kPDriveVel, 0, 0);
frc::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,
```

(continues on next page)

(continued from previous page)

```
    [=](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.value());
        leftReference.SetDouble(leftController.GetSetpoint());
        rightMeasurement.SetDouble(m_drive.GetWheelSpeeds().right.value());
        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 system identification 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 reuse 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 system identification data may 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 to 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.

33.8 State-Space and Model Based Control with WPILib

This section provides an introduction to and describes WPILib support for state-space control.

33.8.1 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 5 of Controls Engineering in FRC.

If you've used WPILib's feedforward classes for `SimpleMotorFeedforward` or its sister classes, or used SysId 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

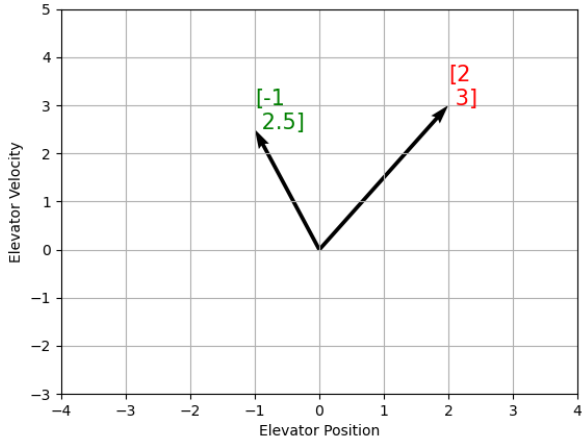
For the background vocabulary that will be used throughout this article, see the *Glossary*.

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 $[\text{position}, \text{velocity}]$:



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:

$$\begin{aligned} \text{Continuous: } \dot{\mathbf{x}} &= \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} \\ \mathbf{y} &= \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{u} \end{aligned}$$

$$\begin{aligned} \text{Discrete: } \mathbf{x}_{k+1} &= \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k \\ \mathbf{y}_k &= \mathbf{C}\mathbf{x}_k + \mathbf{D}\mathbf{u}_k \end{aligned}$$

$\mathbf{A}$	system matrix	$\mathbf{x}$	state vector
$\mathbf{B}$	input matrix	$\mathbf{u}$	input vector
$\mathbf{C}$	output matrix	$\mathbf{y}$	output vector
$\mathbf{D}$	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, **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 $V = K_v \cdot v + K_a \cdot a$, where V is voltage output, v is the flywheel's angular velocity and a is its angular acceleration. This equation can be rewritten as $a = \frac{V - K_v \cdot v}{K_a}$, or $a = \frac{-K_v}{K_a} \cdot v + \frac{1}{K_a} \cdot V$. 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 **x**, acceleration with $\dot{\mathbf{x}}$, and voltage **V** with **u**:

$$\dot{\mathbf{x}} = \left[\frac{-K_v}{K_a}\right]\mathbf{x} + \left[\frac{1}{K_a}\right]\mathbf{u}$$

The output and state are the same, so the output equation is the following:

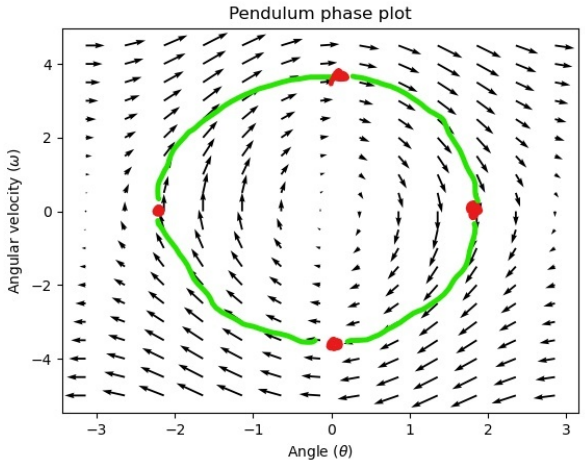
$$\mathbf{y} = [1]\,\mathbf{x} + [0]\,\mathbf{u}$$

That's it! That's the state-space model of a system for which we have the K_v and K_a constants. This same math is used in system identification 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 **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.



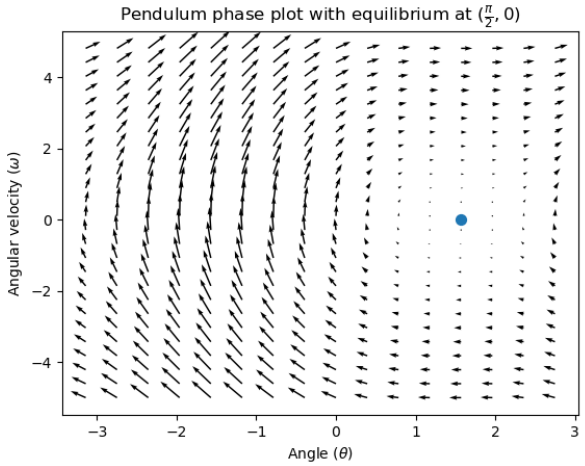
Note that near the edges of the phase portrait, the X axis wraps around as a rotation of π radians counter clockwise and a rotation of π radians clockwise will end at the same point.

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 $\frac{\tau}{I}$, 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.

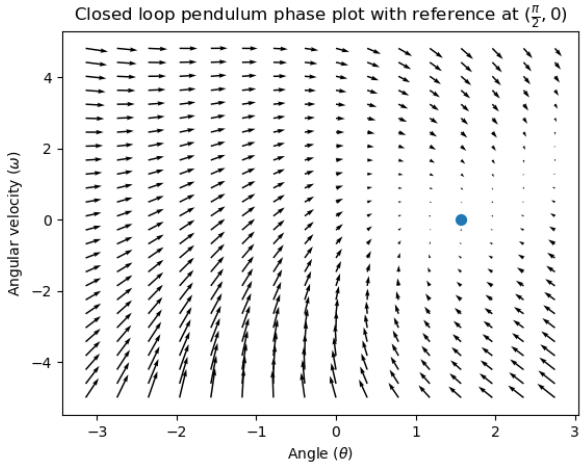
A PID controller is a form of feedback control. State-space control often uses the following *control law*, where $\mathbf{K}$ is some controller *gain* matrix, $\mathbf{r}$ is the *reference* state, and $\mathbf{x}$ is the current state in state-space. The difference between these two vectors, $\mathbf{r} - \mathbf{x}$, 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 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 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 **K**. To do this, we first have to define what a "good" or "bad" **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 $\mathbf{x_{k+1}} = \mathbf{Ax_k} + \mathbf{Bu_k}$.

$$J = \sum_{k=0}^{\infty} \left(\mathbf{x}_k^T \mathbf{Q} \mathbf{x}_k + \mathbf{u}_k^T \mathbf{R} \mathbf{u}_k \right)$$

The *control law* that minimizes **J** can be written as $\mathbf{u} = \mathbf{K}(\mathbf{r_k} - \mathbf{x_k})$, where $r_k - x_k$ is the *error*.

Note

LQR design's **Q** and **R** matrices don't need discretization, but the **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 **Q** and **R** matrices. In our cost function (which describes how "bad" our control law will perform), **Q** and **R** weight our error and control input relative to each other. In the spaceship example from above, we might use a **Q** with relatively small numbers to show that we don't want to highly penalize error, while our **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 Q and R matrices with Bryson's rule. We often use lowercase **q** and **r** to refer to these vectors, and **Q** and **R** to refer to the matrices.

Increasing the **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 **r**'s elements.

Similarly, decreasing the **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 **r** elements.

For example, we might use the following Q and R for an elevator system with position and velocity states.

JAVA

```
// 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
      VecBuilder.fill(12.0),
      // our dt
      0.020);
```

C++

```
// 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},
```

(continues on next page)

(continued from previous page)

```
    // our dt
    0.020_s};
```

PYTHON

```
from wpimath.controller import LinearQuadraticRegulator_2_1
from wpimath.system.plant import LinearSystemId
# Example system -- must be changed to match your robot.
elevatorSystem = LinearSystemId.identifyPositionSystemMeters(5, 0.5)
controller = LinearQuadraticRegulator_2_1(
   elevatorSystem,
   # q's elements
   (0.02, 0.4),
   # r's elements
   (12.0,),
   # our dt
   0.020,
)
```

LQR: example application

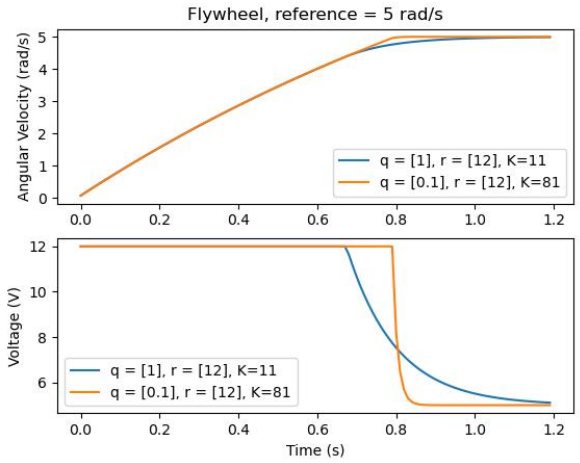
Let's apply a Linear-Quadratic Regulator to a real-world example. Say we have a flywheel velocity system determined through system identification to have $K_v = 1 \frac{\text{volts}}{\text{radian per second}}$ and $K_a = 1.5 \frac{\text{volts}}{\text{radian per second squared}}$. Using the flywheel example above, we have the following linear *system*:

$$\mathbf{x} = \left[\frac{-K_v}{K_a}\right] v + \left[\frac{1}{K_a}\right] V$$

We arbitrarily choose a desired state excursion (maximum error) of $q = [0.1 \text{ rad/sec}]$, and an **r** of $[12 \text{ volts}]$. After discretization with a timestep of 20ms, we find a *gain* of $\mathbf{K} = 81$. This K *gain* acts as the proportional component of a PID loop on flywheel's velocity.

Let's adjust **q** and **r**. We know that increasing the q elements or decreasing the **r** elements we use to create **Q** and **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 **K** drops from ~81 to ~11. Similarly, decreasing our maximum voltage r from 12.0 to 1.2 decreases **K**.

The following graph shows the flywheel's angular velocity and applied voltage over time with two different *gain*s. 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 ~8x.



LQR and Measurement Latency Compensation

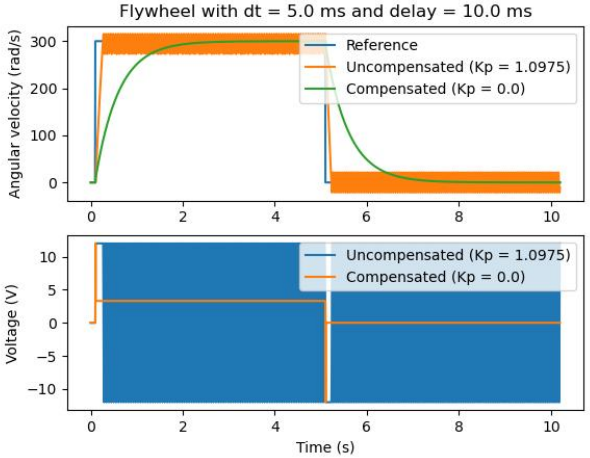
Oftentimes, our sensors have a delay associated with their measurements. For example the SPARK MAX motor controller over CAN can have up to 30ms of delay associated with velocity measurements.

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 system identification.

$$\mathbf{K_{compensated}} = \mathbf{K} \cdot (\mathbf{A} - \mathbf{BK})^{\text{delay}/dt}$$

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

```
// 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);
```

C++

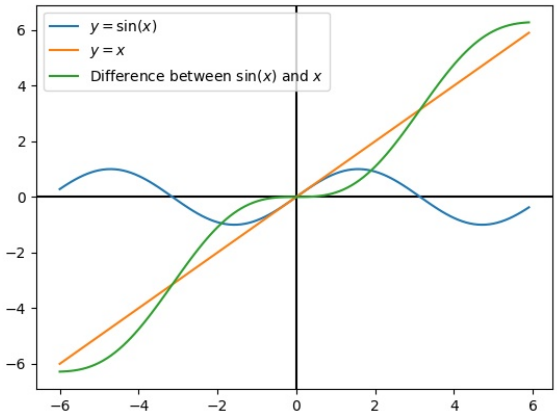
```
// Adjust our LQR's controller for 25 ms of sensor input delay. We
// provide the linear system, discretization timestep, and the sensor
// input delay as arguments.
controller.LatencyCompensate(elevatorSystem, 20_ms, 25_ms);
```

PYTHON

```
# 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, 0.020, 0.025)
```

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 looses 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.

33.8.2 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++/Python) and state-space flywheel system identification (Java/C++/Python) 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:

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$$
$$\mathbf{y} = \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{u}$$

Where *x-dot* is the rate of change of the *system*'s *state*, $\mathbf{x}$ is the system's current state, $\mathbf{u}$ is the *input* to the system, and $\mathbf{y}$ 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 SysId 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 chose 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, $\mathbf{x}$ 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**.

$$V = kV \cdot \mathbf{v} + kA \cdot \mathbf{a}$$
$$\mathbf{a} = \mathbf{v} = \left[\frac{-kV}{kA}\right] v + \left[\frac{1}{kA}\right] V$$

Where $\mathbf{v}$ is flywheel velocity, $\mathbf{a}$ and $\dot{\mathbf{v}}$ are flywheel acceleration, and V is voltage. Rewriting this with the standard convention of $\mathbf{x}$ for the state vector and $\mathbf{u}$ for the input vector, we find:

$$\mathbf{x} = \left[\frac{-kV}{kA}\right] \mathbf{x} + \left[\frac{1}{kA}\right] \mathbf{u}$$

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 $\mathbf{y}$ (or things that our sensors can measure) is our flywheel's velocity, which also happens to be an element of our *state* vector $\mathbf{x}$. Therefore, our output matrix is $\mathbf{C} = \begin{bmatrix}1\end{bmatrix}$, and our system feedthrough matrix is $\mathbf{D} = \begin{bmatrix}0\end{bmatrix}$. Writing this out in continuous-time state-space notation yields the following.

$$\mathbf{x} = \left[\frac{-kV}{kA}\right] \mathbf{x} + \left[\frac{1}{kA}\right] \mathbf{u}$$
$$\mathbf{y} = \begin{bmatrix}1\end{bmatrix} \mathbf{x} + \begin{bmatrix}0\end{bmatrix} \mathbf{u}$$

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

```
32    // Volts per (radian per second)
33    private static final double kFlywheelKv = 0.023;
34
35    // Volts per (radian per second squared)
36    private static final double kFlywheelKa = 0.001;
37
38    // The plant holds a state-space model of our flywheel. This system has the␣
      ↪following properties:
39    //
40    // States: [velocity], in radians per second.
41    // Inputs (what we can "put in"): [voltage], in volts.
42    // Outputs (what we can measure): [velocity], in radians per second.
43    //
44    // The Kv and Ka constants are found using the FRC Characterization toolsuite.
45    private final LinearSystem<N1, N1, N1> m_flywheelPlant =
46        LinearSystemId.identifyVelocitySystem(kFlywheelKv, kFlywheelKa);
```

C++

```
17  #include <frc/system/plant/LinearSystemId.h>
```

```
  // Volts per (radian per second)
  static constexpr auto kFlywheelKv = 0.023_V / 1_rad_per_s;

  // Volts per (radian per second squared)
  static constexpr auto kFlywheelKa = 0.001_V / 1_rad_per_s_sq;

  // The plant holds a state-space model of our flywheel. This system has the
  // following properties:
  //
  // States: [velocity], in radians per second.
  // Inputs (what we can "put in"): [voltage], in volts.
  // Outputs (what we can measure): [velocity], in radians per second.
  //
  // The Kv and Ka constants are found using the FRC Characterization toolsuite.
  frc::LinearSystem<1, 1, 1> m_flywheelPlant =
      frc::LinearSystemId::IdentifyVelocitySystem<units::radian>(kFlywheelKv,
                                                                 kFlywheelKa);
```

Python

```
# Volts per (radian per second)
kFlywheelKv = 0.023

# Volts per (radian per second squared)
kFlywheelKa = 0.001
```

```
        # The plant holds a state-space model of our flywheel. This system has the␣
↪following properties:
        #
        # States: [velocity], in radians per second.
        # Inputs (what we can "put in"): [voltage], in volts.
        # Outputs (what we can measure): [velocity], in radians per second.
        #
        # The Kv and Ka constants are found using the FRC Characterization toolsuite.
        self.flywheelPlant = (
            wpimath.system.plant.LinearSystemId.identifyVelocitySystemRadians(
                kFlywheelKv, kFlywheelKa
            )
        )
```

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 12.3 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++/Python).

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

```
  private static final double kFlywheelMomentOfInertia = 0.00032; // kg * m^2

  // Reduction between motors and encoder, as output over input. If the flywheel␣
↪spins slower than
  // the motors, this number should be greater than one.
  private static final double kFlywheelGearing = 1.0;

  // The plant holds a state-space model of our flywheel. This system has the␣
↪following properties:
  //
  // States: [velocity], in radians per second.
  // Inputs (what we can "put in"): [voltage], in volts.
  // Outputs (what we can measure): [velocity], in radians per second.
  private final LinearSystem<N1, N1, N1> m_flywheelPlant =
      LinearSystemId.createFlywheelSystem(
          DCMotor.getNEO(2), kFlywheelMomentOfInertia, kFlywheelGearing);
```

C++

```
#include <frc/system/plant/LinearSystemId.h>
```

```
  static constexpr units::kilogram_square_meter_t kFlywheelMomentOfInertia =
      0.00032_kg_sq_m;

  // Reduction between motors and encoder, as output over input. If the flywheel
  // spins slower than the motors, this number should be greater than one.
  static constexpr double kFlywheelGearing = 1.0;

  // The plant holds a state-space model of our flywheel. This system has the
  // following properties:
  //
  // States: [velocity], in radians per second.
  // Inputs (what we can "put in"): [voltage], in volts.
  // Outputs (what we can measure): [velocity], in radians per second.
  frc::LinearSystem<1, 1, 1> m_flywheelPlant =
      frc::LinearSystemId::FlywheelSystem(
          frc::DCMotor::NEO(2), kFlywheelMomentOfInertia, kFlywheelGearing);
```

Python

```
kFlywheelMomentOfInertia = 0.00032  # kg/m^2

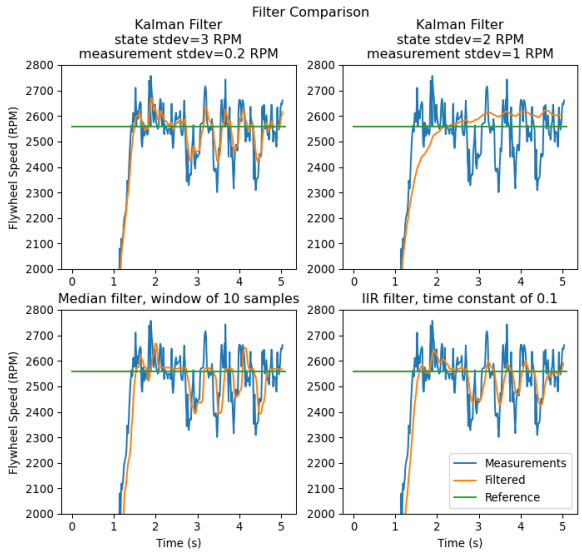
# Reduction between motors and encoder, as output over input. If the flywheel spins␣
↪slower than
# the motors, this number should be greater than one.
kFlywheelGearing = 1
```

```
        # The plant holds a state-space model of our flywheel. This system has the␣
↪following properties:
        #
        # States: [velocity], in radians per second.
        # Inputs (what we can "put in"): [voltage], in volts.
        # Outputs (what we can measure): [velocity], in radians per second.
        self.flywheelPlant = wpimath.system.plant.LinearSystemId.flywheelSystem(
            wpimath.system.plant.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 $\hat{\mathbf{x}}$. 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

```
48  // The observer fuses our encoder data and voltage inputs to reject noise.
49  private final KalmanFilter<N1, N1, N1> m_observer =
50      new KalmanFilter<>(
51          Nat.N1(),
52          Nat.N1(),
53          m_flywheelPlant,
54          VecBuilder.fill(3.0), // How accurate we think our model is
55          VecBuilder.fill(0.01), // How accurate we think our encoder
56          // data is
57          0.020);
```

C++

```
#include <frc/estimator/KalmanFilter.h>
```

```
  // The observer fuses our encoder data and voltage inputs to reject noise.
  frc::KalmanFilter<1, 1, 1> m_observer{
      m_flywheelPlant,
      {3.0},   // How accurate we think our model is
      {0.01},  // How accurate we think our encoder data is
      20_ms};
```

Python

```
        # The observer fuses our encoder data and voltage inputs to reject noise.
        self.observer = wpimath.estimator.KalmanFilter_1_1_1(
            self.flywheelPlant,
            (3,),  # How accurate we think our model is
            (0.01,),  # How accurate we think our encoder data is
            0.020,
        )
```

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

```
  // A LQR uses feedback to create voltage commands.
  private final LinearQuadraticRegulator<N1, N1, N1> m_controller =
      new LinearQuadraticRegulator<>(
          m_flywheelPlant,
          VecBuilder.fill(8.0), // qelms. Velocity error tolerance, in radians per␣
↪second. Decrease
          // this to more heavily penalize state excursion, or make the controller␣
```

(continues on next page)

(continued from previous page)

```
↪behave more
        // aggressively.
        VecBuilder.fill(12.0), // relms. Control effort (voltage) tolerance.␣
↪Decrease this to more
        // heavily penalize control effort, or make the controller less aggressive.␣
↪12 is a good
        // starting point because that is the (approximate) maximum voltage of a␣
↪battery.
        0.020); // Nominal time between loops. 0.020 for TimedRobot, but can be
 // lower if using notifiers.
```

C++

```
#include <frc/controller/LinearQuadraticRegulator.h>
```

```
// A LQR uses feedback to create voltage commands.
frc::LinearQuadraticRegulator<1, 1> m_controller{
    m_flywheelPlant,
    // qelms. Velocity error tolerance, in radians per second. Decrease this
    // to more heavily penalize state excursion, or make the controller behave
    // more aggressively.
    {8.0},
    // relms. Control effort (voltage) tolerance. Decrease this to more
    // heavily penalize control effort, or make the controller less
    // aggressive. 12 is a good starting point because that is the
    // (approximate) maximum voltage of a battery.
    {12.0},
    // Nominal time between loops. 20ms for TimedRobot, but can be lower if
    // using notifiers.
    20_ms};

// The state-space loop combines a controller, observer, feedforward and plant
// for easy control.
frc::LinearSystemLoop<1, 1, 1> m_loop{m_flywheelPlant, m_controller,
                                      m_observer, 12_V, 20_ms};
```

Python

```
      # A LQR uses feedback to create voltage commands.
      self.controller = wpimath.controller.LinearQuadraticRegulator_1_1(
          self.flywheelPlant,
          (8,),  # qelms. Velocity error tolerance, in radians per second. Decrease
          # this to more heavily penalize state excursion, or make the controller␣
↪behave more
          # aggressively.
          (12,),  # relms. Control effort (voltage) tolerance. Decrease this to more
          # heavily penalize control effort, or make the controller less aggressive.
↪ 12 is a good
          # starting point because that is the (approximate) maximum voltage of a␣
↪battery.
          0.020,  # Nominal time between loops. 0.020 for TimedRobot, but can be␣
```

(continues on next page)

(continued from previous page)

```
↪lower if using notifiers.
        )
```

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

```
  // The state-space loop combines a controller, observer, feedforward and plant for␣
↪easy control.
  private final LinearSystemLoop<N1, N1, N1> m_loop =
      new LinearSystemLoop<>(m_flywheelPlant, m_controller, m_observer, 12.0, 0.020);
```

C++

```
#include <frc/system/LinearSystemLoop.h>
```

```
  // The state-space loop combines a controller, observer, feedforward and plant
  // for easy control.
  frc::LinearSystemLoop<1, 1, 1> m_loop{m_flywheelPlant, m_controller,
                                        m_observer, 12_V, 20_ms};
```

Python

```
        # The state-space loop combines a controller, observer, feedforward and plant␣
↪for easy control.
        self.loop = wpimath.system.LinearSystemLoop_1_1_1(
            self.flywheelPlant, self.controller, self.observer, 12.0, 0.020
        )
```

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

```
  @Override
  public void teleopPeriodic() {
    // Sets the target speed of our flywheel. This is similar to setting the setpoint of a
    // PID controller.
    if (m_joystick.getTriggerPressed()) {
      // We just pressed the trigger, so let's set our next reference
      m_loop.setNextR(VecBuilder.fill(kSpinupRadPerSec));
    } else if (m_joystick.getTriggerReleased()) {
      // We just released the trigger, so let's spin down
      m_loop.setNextR(VecBuilder.fill(0.0));
    }

    // Correct our Kalman filter's state vector estimate with encoder data.
    m_loop.correct(VecBuilder.fill(m_encoder.getRate()));

    // Update our LQR to generate new voltage commands and use the voltages to predict the next
    // state with out Kalman filter.
    m_loop.predict(0.020);

    // Send the new calculated voltage to the motors.
    // voltage = duty cycle * battery voltage, so
    // duty cycle = voltage / battery voltage
    double nextVoltage = m_loop.getU(0);
    m_motor.setVoltage(nextVoltage);
  }
}
```

C++

```
#include <numbers>

#include <frc/DriverStation.h>
#include <frc/Encoder.h>
#include <frc/TimedRobot.h>
#include <frc/XboxController.h>
#include <frc/controller/LinearQuadraticRegulator.h>
#include <frc/drive/DifferentialDrive.h>
#include <frc/estimator/KalmanFilter.h>
#include <frc/motorcontrol/PWMSparkMax.h>
#include <frc/system/LinearSystemLoop.h>
#include <frc/system/plant/DCMotor.h>
#include <frc/system/plant/LinearSystemId.h>
```

```
  void TeleopPeriodic() override {
    // Sets the target speed of our flywheel. This is similar to setting the
    // setpoint of a PID controller.
    if (m_joystick.GetRightBumperButton()) {
      // We pressed the bumper, so let's set our next reference
      m_loop.SetNextR(frc::Vectord<1>{kSpinup.value()});
    } else {
      // We released the bumper, so let's spin down
```

(continues on next page)

(continued from previous page)

```
      m_loop.SetNextR(frc::Vectord<1>{0.0});
    }

    // Correct our Kalman filter's state vector estimate with encoder data.
    m_loop.Correct(frc::Vectord<1>{m_encoder.GetRate()});

    // Update our LQR to generate new voltage commands and use the voltages to
    // predict the next state with out Kalman filter.
    m_loop.Predict(20_ms);

    // Send the new calculated voltage to the motors.
    // voltage = duty cycle * battery voltage, so
    // duty cycle = voltage / battery voltage
    m_motor.SetVoltage(units::volt_t{m_loop.U(0)});
  }
```

Python

```
    def teleopPeriodic(self) -> None:
        # Sets the target speed of our flywheel. This is similar to setting the␣
↪setpoint of a
        # PID controller.
        if self.joystick.getTriggerPressed():
            # We just pressed the trigger, so let's set our next reference
            self.loop.setNextR([kSpinUpRadPerSec])

        elif self.joystick.getTriggerReleased():
            # We just released the trigger, so let's spin down
            self.loop.setNextR([0.0])

        # Correct our Kalman filter's state vector estimate with encoder data.
        self.loop.correct([self.encoder.getRate()])

        # Update our LQR to generate new voltage commands and use the voltages to␣
↪predict the next
        # state with out Kalman filter.
        self.loop.predict(0.020)

        # Send the new calculated voltage to the motors.
        # voltage = duty cycle * battery voltage, so
        # duty cycle = voltage / battery voltage
        nextVoltage = self.loop.U(0)
        self.motor.setVoltage(nextVoltage)
```

Angle Wrap with LQR

Mechanisms with a continuous angle can have that angle wrapped by calling the code below instead of `lqr.Calculate(x, r)`.

JAVA

```
var error = lqr.getR().minus(x);
error.set(0, 0, MathUtil.angleModulus(error.get(0, 0)));
var u = lqr.getK().times(error);
```

C++

```
Eigen::Vector<double, 2> error = lqr.R() - x;
error(0) = frc::AngleModulus(units::radian_t{error(0)}).value();
Eigen::Vector<double, 2> u = lqr.K() * error;
```

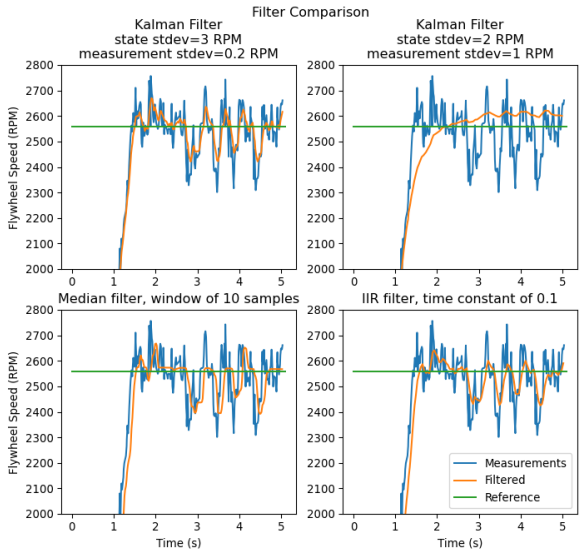
PYTHON

```
error = lqr.R() - x
error[0] = wpimath.angleModulus(error[0])
u = lqr.K() * error
```

33.8.3 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

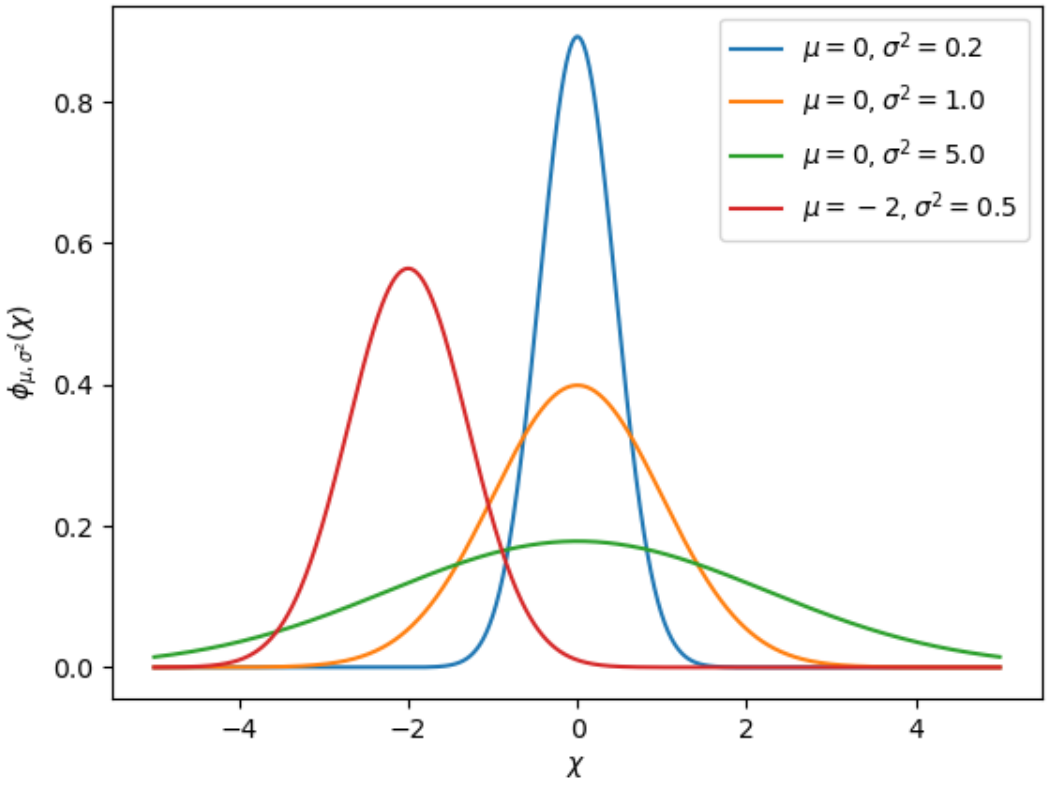
Kalman filters utilize a *Gaussian distribution* to model the noise in a process[1]. In the case of a Kalman filter, the estimated *state* of the system is the mean, while the variance is a measure of how certain (or uncertain) the filter is about the true *state*.

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 it's 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:

$$\square = \begin{bmatrix} \text{cov}(x_1, x_1) & \text{cov}(x_1, x_2) & \dots & \text{cov}(x_1, x_n) \\ \text{cov}(x_2, x_1) & \text{cov}(x_2, x_2) & \dots & \text{cov}(x_1, x_n) \\ \vdots & \vdots & \ddots & \vdots \\ \text{cov}(x_n, x_1) & \text{cov}(x_n, x_2) & \dots & \text{cov}(x_n, x_n) \end{bmatrix}$$

[1] In a real robot, noise comes from all sorts of sources. Stray electromagnetic radiation adds extra voltages to sensor readings, vibrations, and temperature variations throw off inertial measurement units, gear lash causes encoders to have inaccuracies when directions change... all sorts of things. It's important to realize that, by themselves, each of these sources of "noise" aren't guaranteed to follow any pattern. Some of them might be the "white noise" random vibrations you've probably heard on the radio. Others might be "pops" or single-loop errors. Others might be nominally zero, but strongly correlated with events on the robot. However, the *Central Limit Theorem* shows mathematically that regardless of how the individual sources of noise are distributed, as we add more and more of them up their combined effect eventually is distributed like a Gaussian. Since we do not know the exact individual sources of noise, the best choice of a model we can make is indeed that Gaussian function.


$\mu = 0, \sigma^2 = 0.2$
$\mu = 0, \sigma^2 = 1.0$
$\mu = 0, \sigma^2 = 5.0$
$\mu = -2, \sigma^2 = 0.5$
0.8
0.6
0.4
0.2
0.0
$\phi_{\mu,\sigma^2}(\chi)$
−4
−2
0
2
4
χ

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

$$\hat{\mathbf{x}}_{k+1}^{-} = \mathbf{A}\hat{\mathbf{x}}_{k}^{+} + \mathbf{B}\mathbf{u}_k$$
$$\mathbf{P}_{k+1}^{-} = \mathbf{A}\mathbf{P}_{k}^{-}\mathbf{A}^T + \square\mathbf{Q}\square^T$$

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	**u**	input vector
C	output matrix	**y**	output vector
D	feedthrough matrix	$\square$	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 **x**, together with **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 $\begin{bmatrix}0.01\end{bmatrix}$ would have the following **Q** and **R** matrices:

$$Q = \begin{bmatrix}0.01 & 0\\0 & 1.0\end{bmatrix}, R = \begin{bmatrix}0.0001\end{bmatrix}$$

Error Covariance Matrix

The error covariance matrix $\mathbf{P}$ describes the covariance of the state estimate $\hat{\mathbf{x}}$. Informally, $\mathbf{P}$ describes our certainty about the estimated *state*. If $\mathbf{P}$ is large, our uncertainty about the true state is large. Conversely, a $\mathbf{P}$ with smaller elements would imply less uncertainty about our true state.

As we project the model forward, $\mathbf{P}$ 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 $\mathbf{P}$ increases by the process noise covariance matrix $\mathbf{Q}$. Larger values of $\mathbf{Q}$ will make our error covariance $\mathbf{P}$ grow more quickly. This $\mathbf{P}$ 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 $\mathbf{K}$. Large values of $\mathbf{K}$ more highly weight incoming measurements, while smaller values of $\mathbf{K}$ more highly weight our state prediction. Because $\mathbf{K}$ is related to $\mathbf{P}$, larger values of $\mathbf{P}$ will increase $\mathbf{K}$ and more heavily weight measurements. If, for example, a filter is predicted for a long duration, the large $\mathbf{P}$ would heavily weight the new information.

Finally, the error covariance $\mathbf{P}$ 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 $\mathbf{Q}$ and $\mathbf{R}$ 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 $\mathbf{Q}$), 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

```
48  // The observer fuses our encoder data and voltage inputs to reject noise.
49  private final KalmanFilter<N1, N1, N1> m_observer =
50      new KalmanFilter<>(
51          Nat.N1(),
52          Nat.N1(),
53          m_flywheelPlant,
54          VecBuilder.fill(3.0), // How accurate we think our model is
55          VecBuilder.fill(0.01), // How accurate we think our encoder
```

(continues on next page)

(continued from previous page)

```
          // data is
          0.020);
```

C++

```
#include <numbers>

#include <frc/DriverStation.h>
#include <frc/Encoder.h>
#include <frc/TimedRobot.h>
#include <frc/XboxController.h>
#include <frc/controller/LinearQuadraticRegulator.h>
#include <frc/drive/DifferentialDrive.h>
#include <frc/estimator/KalmanFilter.h>
#include <frc/motorcontrol/PWMSparkMax.h>
#include <frc/system/LinearSystemLoop.h>
#include <frc/system/plant/DCMotor.h>
#include <frc/system/plant/LinearSystemId.h>
#include <units/angular_velocity.h>
```

```
  // The observer fuses our encoder data and voltage inputs to reject noise.
  frc::KalmanFilter<1, 1, 1> m_observer{
      m_flywheelPlant,
      {3.0},   // How accurate we think our model is
      {0.01},  // How accurate we think our encoder data is
      20_ms};
```

Python

```
        # The observer fuses our encoder data and voltage inputs to reject noise.
        self.observer = wpimath.estimator.KalmanFilter_1_1_1(
            self.flywheelPlant,
            (3,),  # How accurate we think our model is
            (0.01,),  # How accurate we think our encoder data is
            0.020,
        )
```

Footnotes

33.8.4 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 that also support fusing 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 $\begin{bmatrix} x & y & \theta \end{bmatrix}^T$, 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.

Here's how to initialize a `DifferentialDrivePoseEstimator`:

JAVA

```
  private final DifferentialDrivePoseEstimator m_poseEstimator =
      new DifferentialDrivePoseEstimator(
          m_kinematics,
          m_gyro.getRotation2d(),
          m_leftEncoder.getDistance(),
          m_rightEncoder.getDistance(),
          new Pose2d(),
          VecBuilder.fill(0.05, 0.05, Units.degreesToRadians(5)),
          VecBuilder.fill(0.5, 0.5, Units.degreesToRadians(30)));
```

C++

```
  frc::DifferentialDrivePoseEstimator m_poseEstimator{
      m_kinematics,
      m_gyro.GetRotation2d(),
      units::meter_t{m_leftEncoder.GetDistance()},
      units::meter_t{m_rightEncoder.GetDistance()},
      frc::Pose2d{},
      {0.01, 0.01, 0.01},
      {0.1, 0.1, 0.1}};
```

Add odometry measurements every loop by calling `Update()`.

JAVA

```
    m_poseEstimator.update(
        m_gyro.getRotation2d(), m_leftEncoder.getDistance(), m_rightEncoder.
↪getDistance());
```

C++

```
  m_poseEstimator.Update(m_gyro.GetRotation2d(),
                         units::meter_t{m_leftEncoder.GetDistance()},
                         units::meter_t{m_rightEncoder.GetDistance()});
```

Add vision pose measurements occasionally by calling `AddVisionMeasurement()`.

JAVA

```
    // Compute the robot's field-relative position exclusively from vision measurements.
    Pose3d visionMeasurement3d =
        objectToRobotPose(m_objectInField, m_robotToCamera, m_cameraToObjectEntry);

    // Convert robot pose from Pose3d to Pose2d needed to apply vision measurements.
    Pose2d visionMeasurement2d = visionMeasurement3d.toPose2d();

    // Apply vision measurements. For simulation purposes only, we don't input a latency delay -- on
    // a real robot, this must be calculated based either on known latency or timestamps.
    m_poseEstimator.addVisionMeasurement(visionMeasurement2d, Timer.getFPGATimestamp());
```

C++

```
  // Compute the robot's field-relative position exclusively from vision
  // measurements.
  frc::Pose3d visionMeasurement3d = ObjectToRobotPose(
      m_objectInField, m_robotToCamera, m_cameraToObjectEntryRef);

  // Convert robot's pose from Pose3d to Pose2d needed to apply vision
  // measurements.
  frc::Pose2d visionMeasurement2d = visionMeasurement3d.ToPose2d();

  // Apply vision measurements. For simulation purposes only, we don't input a
  // latency delay -- on a real robot, this must be calculated based either on
  // known latency or timestamps.
  m_poseEstimator.AddVisionMeasurement(visionMeasurement2d,
                                       frc::Timer::GetFPGATimestamp());
```

Tuning Pose Estimators

All pose estimators offer user-customizable standard deviations for model and measurements (defaults are used if you don't provide them). Standard deviation is a measure of how spread out the noise is for a random signal. Giving a state a smaller standard deviation means it will be trusted more during data fusion.

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.

When incorporating AprilTag poses, make the vision heading standard deviation very large, make the gyro heading standard deviation small, and scale the vision x and y standard deviation by distance from the tag.

33.8.5 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, does not invert any motors. You may need to call the `setInverted(true)` method on the motor controller object to invert so that positive input creates forward motion.

The Importance of Graphs

Reliable data of the *system's state*s, *input*s and *output*s 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

```java
@Override
public void robotPeriodic() {
   NetworkTableInstance.getDefault().flush();
}
```

C++

```cpp
void RobotPeriodic() {
   NetworkTableInstance::GetDefault().Flush();
}
```

PYTHON

```python
from ntcore import NetworkTableInstance
def robotPeriodic(self):
   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*.

33.9 Controls Glossary

bang-bang control
: A very simple, no-tuning-required closed-loop control technique. It simply "turns on" the *control effort* when the *process variable* is too small, and "turns off" the control effort when the process variable is too big. It works well in some cases, but not all. See "Bang-bang" control on Wikipedia for more info.

Cartesian coordinate system
: A set of points in space where each point is described by a set of numbers, indicating its *coordinates* within that space. These coordinates are an expression of the *orthogonal* distance of each point from a set of fixed, orthogonal axes (IE, a "rectangular" system). 2-dimension and 3-dimension spaces are most common in FRC (and likely what was learned in algebra 1), but any number of dimensions is theoretically possible. See Cartesian coordinate system on Wikipedia for more info.

churning losses
: Complex friction-like forces arising from the fact that when gears and bearings rotate, they must displace liquid lubricant. This reduces the efficiency of rotating mechanisms.

control signal
The driving signal sent to a *plant* by a *controller*, usually quantified as a voltage.

control effort
Control signal

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.

convolution
A mathematical operation that calculates a weighted moving average of one function, with the weights assigned by a second function. A common way to "filter" sensor input is to apply a *convolution* to it, using a carefully-chosen filtering function. See convolution on Wikipedia for more info.

counter-electromotive force
A *voltage* generated in a spinning motor. The voltage is a result of the fact that has a coil of wire rotating near a magnet. See Counter-electromotive_force on Wikipedia for more info.

current
The flow of electrons through a conductor. Current is described with a unit of "Amps" (or simply "A"), and is measured at a single point in a circuit. One amp is equal to 6241509074000000000 electrons moving past the measurement point in one second.

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.

derivative
A mathematical operation which evaluates the "rate-of-change" of a function at a given point. See derivative on Wikipedia for more info.

error
Reference minus an *output* or *state*.

exponential search
An iterative process of finding a specific value within a wide search range by applying a multiplicative factor to the search value. See exponential search on Wikipedia for more info.

exponential smoothing
A very common way to implement a simple low-pass filter, using an exponential window function in a *convolution* with an input signal. The convolution operation simplifies down to a very simple set of math operations on the current input and previous output. See exponential smoothing on Wikipedia for more info.

gain
A scalar value that relates the magnitude of an input signal to the magnitude of an output signal. For example, `gain` in `output = gain * input`. 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.

Gaussian distribution
A special mathematical function that describes distributions of averages. 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). See Gaussian distribution on Wikipedia for more info.

gradient
The *derivative*, but applied to a function with multiple inputs. As a result, the output is both the magnitude of the rate of change, and the vector direction along which it occurs.

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 **u**, a column vector with one entry per *input* to the *system*.

least-squares regression
A curve-fitting technique which picks a curve to minimize the *square* of the error between the fitted curve and the actual measured data. See ordinary least-squares regression on Wikipedia for more info.

LQR
Linear-Quadratic Regulator - A feedback control scheme which seeks to operate a system in a "most optimal" or "lowest cost" manner, in the sense of minimizing the square of some "cost function" that represents a combination of system error and control effort. This requires an accurate mathematical model of the system being controlled, and function describing the "cost" of any given system state. See LQR on Wikipedia for more info.

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.

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.

orthogonal
Having the property of being independent, or lacking mutual influence. For example, two lines are orthogonal if moving any number of units along one line causes zero displacement along the other line. In a *cartesian coordinate system*, orthogonal lines are often said to have 90-degree angles between each other.

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 a encoder that measures it's velocity.

- Ex. A drivetrain might use solvePNP and V-SLAM to find it's 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 **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.

phase portrait
A graph of a function's value and its *derivative* as they change in time, given some initial starting conditions. They are useful for analyzing system behavior (stable/unstable operating points, limit cycles, etc.) given a certain set of parameters or starting conditions. See phase portrait on Wikipedia for more info.

PID
Proportional-Integral-Derivative - A feedback controller which calculates a *control signal* from a weighted sum of the *error*, the rate of change of the error, and an accumulated sum of previous errors. See PID controller on Wikipedia for more info.

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.

r-squared
A statistical measurement of how well a model predicts a set of data, representing the fraction of the observed variation in the independent variable that is accurately predicted by the model. The value typically runs from 0.0 (a terrible fit, equivalent to just guessing the average value of your independent variable) to 1.0 (a perfect fit). See Coefficient_of_determination on Wikipedia for more info.

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*.

RMSE
Root Mean Squared Error - Statistical measurement of how well a curve is fit to a set of data. It is calculated as the square root of the average (mean) of the squares of all the errors between the actual sample and the curve fit. It has units of the original input data. See Root Mean Squared Error on Wikipedia for more info.

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.

signum function
A non-continuous function that expresses the "sign" of its input. It is equal to -1 for all negative input numbers, 0 for an input of 0, and 1 for all positive input numbers. See signum function on Wikipedia for more info.

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 **x**, a column vector with one entry per *state*.

statistically robust
The property of a data processing algorithm which makes it resilient to a noisy or outlier-prone data set. Designing statistically robust algorithms on robots is important because real-world sensor data can often be unpredictable, but unexpected robot behavior is never desirable. See Robust Statistics on Wikipedia for more info.

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 are 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 *system*'s *dynamics* in a mathematical model using measured data. The SysId toolsuite uses system identification to find kS, kV, and kA terms.

system response
The behavior of a *system* over time for a given *input*.

voltage
The measurement of how much an electric field is "pushing" electrons through a circuit. It is sometimes called "Electromotive Force", or "EMF". It is measured in units of "Volts". It always is defined between *two* points in a circuit. If one electron travels between two points that have one volt of EMF between them, it will have been accelerated to the point of having $\frac{1}{6241509074000000000}$ joules of energy.

viscous drag
The force generated from an object moving *relatively* slowly through non-turbulent fluid. In this region, the force is roughly proportional to the *velocity* of the object. It describes the most common type of "air resistance" an FRC robot would encounter, as well as losses in a gearbox from displacing grease. See Drag (physics) on Wikipedia for more info.

x-dot
$\dot{\mathbf{x}}$, or x-dot: the derivative of the *state* vector **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*.

34 Convenience Features

This section covers some general convenience features that be used with other advanced programming features.

34.1 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

```
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()));
        }, 0.01, 0.005);
```

(continues on next page)

(continued from previous page)

```
    }
    @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);
            }
        }
    }
}
```

C++ (Header)

```
class Robot : public frc::TimedRobot {
 private:
  frc::Joystick m_joystick{0};
  frc::Encoder m_encoder{1, 2};
  frc::Spark m_motor{1};
  frc::PIDController m_controller{1.0, 0.0, 0.5, 10_ms};
  Robot();
  void TeleopPeriodic() override;
};
```

C++ (Source)

```
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 timeslots 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.).

34.2 Event-Based Programming With EventLoop

Many operations in robot code are driven by certain conditions; buttons are one common example. Conditions can be polled with an *imperative programming* style by using an `if` statement in a periodic method. As an alternative, WPILib offers an *event-driven programming* style of API in the shape of the `EventLoop` and `BooleanEvent` classes.

Note

The example code here is taken from the EventLoop example project (Java/C++).

34.2.1 EventLoop

The `EventLoop` class is a "container" for pairs of conditions and actions, which can be polled using the `poll()`/`Poll()` method. When polled, every condition will be queried and if it returns `true` the action associated with the condition will be executed.

JAVA

```
private final EventLoop m_loop = new EventLoop();
private final Joystick m_joystick = new Joystick(0);
@Override
public void robotPeriodic() {
  // poll all the bindings
  m_loop.poll();
}
```

C++

```
frc::EventLoop m_loop{};
void RobotPeriodic() override { m_loop.Poll(); }
```

Warning

The `EventLoop`'s `poll()` method should be called consistently in a `*Periodic()` method. Failure to do this will result in unintended loop behavior.

34.2.2 BooleanEvent

The `BooleanEvent` class represents a boolean condition: a `BooleanSupplier` (Java) / `std::function<bool()>` (C++).

To bind a callback action to the condition, use `ifHigh()`/`IfHigh()`:

JAVA

```
    BooleanEvent atTargetVelocity =
        new BooleanEvent(m_loop, m_controller::atSetpoint)
            // debounce for more stability
            .debounce(0.2);

    // if we're at the target velocity, kick the ball into the shooter wheel
    atTargetVelocity.ifHigh(() -> m_kicker.set(0.7));
```

C++

```
    frc::BooleanEvent atTargetVelocity =
        frc::BooleanEvent(
            &m_loop,
            [&controller = m_controller] { return controller.AtSetpoint(); })
            // debounce for more stability
            .Debounce(0.2_s);

    // if we're at the target velocity, kick the ball into the shooter wheel
    atTargetVelocity.IfHigh([&kicker = m_kicker] { kicker.Set(0.7); });
```

Remember that button binding is *declarative*: bindings only need to be declared once, ideally some time during robot initialization. The library handles everything else.

34.2.3 Composing Conditions

`BooleanEvent` objects can be composed to create composite conditions. In C++ this is done using operators when applicable, other cases and all compositions in Java are done using methods.

and() / &&

The `and()`/&& composes two `BooleanEvent` conditions into a third condition that returns `true` only when **both** of the conditions return `true`.

JAVA

```
    // if the thumb button is held
    intakeButton
        // and there is not a ball at the kicker
        .and(isBallAtKicker.negate())
        // activate the intake
        .ifHigh(() -> m_intake.set(0.5));
```

C++

```
    // if the thumb button is held
    (intakeButton
     // and there is not a ball at the kicker
     && !isBallAtKicker)
        // activate the intake
        .IfHigh([&intake = m_intake] { intake.Set(0.5); });
```

or() / ||

The `or()`/`||` composes two `BooleanEvent` conditions into a third condition that returns `true` only when **either** of the conditions return `true`.

JAVA

```
    // if the thumb button is not held
    intakeButton
        .negate()
        // or there is a ball in the kicker
        .or(isBallAtKicker)
        // stop the intake
        .ifHigh(m_intake::stopMotor);
```

C++

```
    // if the thumb button is not held
    (!intakeButton
     // or there is a ball in the kicker
     || isBallAtKicker)
        // stop the intake
        .IfHigh([&intake = m_intake] { intake.Set(0.0); });
```

negate() / !

The `negate()/!` composes one `BooleanEvent` condition into another condition that returns the opposite of what the original conditional did.

JAVA

```
        // and there is not a ball at the kicker
        .and(isBallAtKicker.negate())
```

C++

```
    // and there is not a ball at the kicker
    && !isBallAtKicker)
```

debounce() / Debounce()

To avoid rapid repeated activation, conditions (especially those originating from digital inputs) can be debounced with the *WPILib Debouncer class* using the *debounce* method:

JAVA

```
    BooleanEvent atTargetVelocity =
        new BooleanEvent(m_loop, m_controller::atSetpoint)
            // debounce for more stability
            .debounce(0.2);
```

C++

```
    frc::BooleanEvent atTargetVelocity =
        frc::BooleanEvent(
            &m_loop,
            [&controller = m_controller] { return controller.AtSetpoint(); })
            // debounce for more stability
            .Debounce(0.2_s);
```

rising(), falling()

Often times it is desired to bind an action not to the *current* state of a condition, but instead to when that state *changes*. For example, binding an action to when a button is newly pressed as opposed to when it is held. This is what the `rising()` and `falling()` decorators do: `rising()` will return a condition that is `true` only when the original condition returned `true` in the *current* polling and `false` in the *previous* polling; `falling()` returns a condition that returns `true` only on a transition from `true` to `false`.

> **Warning**
>
> Due to the "memory" these conditions have, do not use the same instance in multiple places.

JAVA

```
    // when we stop being at the target velocity, it means the ball was shot
    atTargetVelocity
        .falling()
        // so stop the kicker
        .ifHigh(m_kicker::stopMotor);
```

C++

```
    // when we stop being at the target velocity, it means the ball was shot
    atTargetVelocity
        .Falling()
        // so stop the kicker
        .IfHigh([&kicker = m_kicker] { kicker.Set(0.0); });
```

Downcasting `BooleanEvent` Objects

To convert `BooleanEvent` objects to other types, most commonly the `Trigger` subclass used for *binding commands to conditions*, the generic `castTo()`/`CastTo()` decorator exists:

JAVA

```
Trigger trigger = booleanEvent.castTo(Trigger::new);
```

C++

```
frc2::Trigger trigger = booleanEvent.CastTo<frc2::Trigger>();
```

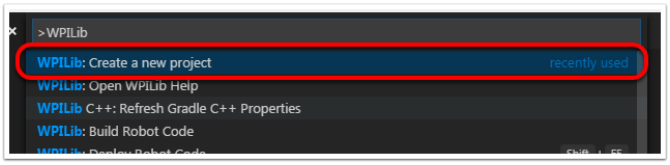
> **Note**
>
> In Java, the parameter expects a method reference to a constructor accepting an `EventLoop` instance and a `BooleanSupplier`. Due to the lack of method references, this parameter is defaulted in C++ as long as a constructor of the form `Type(frc::EventLoop*, std::function<bool()>)` exists.

35 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.



35.1 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++, Python): Demonstrates a simple differential drive implementation using "arcade"-style controls through the `DifferentialDrive` class.
- **Arcade Drive Xbox Controller** (Java, C++, Python): Demonstrates the same functionality seen in the previous example, except using an `XboxController` instead of an ordinary joystick.
- **Getting Started** (Java, C++, Python): Demonstrates a simple autonomous routine that drives forwards for two seconds at half speed.

- **Mecanum Drive** (Java, C++, Python): Demonstrates a simple mecanum drive implementation using the `MecanumDrive` class.
- **Motor Controller** (Java, C++, Python): Demonstrates how to control the output of a motor with a joystick with an encoder to read motor position.
- **Simple Vision** (Java, C++, Python): Demonstrates how to stream video from a USB camera to the dashboard.
- **Relay** (Java, C++, Python): Demonstrates the use of the `Relay` class to control a relay output with a set of joystick buttons.
- **Solenoids** (Java, C++, Python): Demonstrates the use of the `Solenoid` and `DoubleSolenoid` classes to control solenoid outputs with a set of joystick buttons.
- **TankDrive** (Java, C++, Python): Demonstrates a simple differential drive implementation using "tank"-style controls through the `DifferentialDrive` class.
- **Tank Drive Xbox Controller** (Java, C++, Python): Demonstrates the same functionality seen in the previous example, except using an `XboxController` instead of an ordinary joystick.

35.2 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++, Python): 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.
- **DifferentialDrivePoseEstimator** (Java, C++): Demonstrates an advanced differential drive implementation with all the features of `DifferentialDriveBot`. In addition this example uses `DifferentialDrivePoseEstimator` to track the robots position on the field. It demonstrates these features by using WPILib Simulation.
- **Elevator with Profiled PID Controller** (Java, C++, Python): Demonstrates the use of the `ProfiledPIDController` class to control the position of an elevator mechanism.
- **Elevator with Trapezoid Profiled PID** (Java, C++, Python): Demonstrates the use of the `TrapezoidProfile` class in conjunction with a "smart motor controller" to control the position of an elevator mechanism.
- **Elevator with Exponential Profiled PID** (Java, C++): Demonstrates the use of the `ExponentialProfile` class in conjunction with a "smart motor controller" to control the position of an elevator mechanism.
- **Flywheel Bang-Bang Controller** (Java, C++, Python): Uses the `BangBangController` class to simply yet effectively control a flywheel.
- **Gyro Mecanum** (Java, C++, Python): Demonstrates field-oriented control of a mecanum robot through the `MecanumDrive` class in conjunction with a gyro.
- **MecanumBot** (Java, C++, Python): 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.

- **Mecanum Drive PoseEstimator** (Java, C++): Demonstrates an advanced mecanum drive implementation with all the features of `MecanumBot`. In addition this example uses `MecanumDrivePoseEstimator` to track the robots position on the field.
- **PotentiometerPID** (Java, C++, Python): Demonstrates the use of the `PIDController` class and a potentiometer to control the position of an elevator mechanism.
- **SwerveBot** (Java, C++, Python): 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.
- **Swerve Drive PoseEstimator** (Java, C++): Demonstrates an advanced swerve drive implementation with all the features of `SwerveBot`. In addition this example uses `SwerveDrivePoseEstimator` to track the robots position on the field.
- **UltrasonicPID** (Java, C++, Python): Demonstrates the use of the `PIDController` class in conjunction with an ultrasonic sensor to drive to a set distance from an object.

35.3 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.

- **HTTP Camera** (Java, C++): Demonstrates the use of OpenCV and a HTTP Camera to overlay a rectangle on a captured video feed and stream it to the dashboard.
- **Power Distribution CAN Monitoring** (Java, C++, Python): Demonstrates obtaining sensor information from a Power Distribution module over CAN using the `PowerDistribution` class.
- **Duty Cycle Encoder** (Java, C++, Python): Demonstrates the use of the `DutyCycleEncoder` class to read values from a PWM-type absolute encoder.
- **DutyCycleInput** (Java, C++, Python): Demonstrates the use of the `DutyCycleInput` class to read the frequency and fractional duty cycle of a *PWM* input.
- **Encoder** (Java, C++, Python): Demonstrates the use of the `Encoder` class to read values from a quadrature encoder.
- **Gyro** (Java, C++, Python): Demonstrates the use of the `AnalogGyro` class to measure robot heading and stabilize driving.
- **Intermediate Vision** (Java, C++, Python): Demonstrates the use of OpenCV and a USB camera to overlay a rectangle on a captured video feed and stream it to the dashboard.
- **AprilTagsVision** (Java, C++, Python): Demonstrates on-roboRIO detection of AprilTags using an attached USB camera.
- **Ultrasonic** (Java, C++, Python): Demonstrates the use of the `Ultrasonic` class to read data from an ultrasonic sensor in conjunction with the `MedianFilter` class to reduce signal noise.
- **SysIdRoutine** (Java, C++, Python): Demonstrates the use of the SysIdRoutine API to gather characterization data for a differential drivetrain.

35.4 Command-Based Examples

These examples demonstrate the use of the *Command-Based framework*.

- **DriveDistanceOffboard** (Java, C++, Python): 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.
- **Rapid React Command Bot** (Java, C++): This project uses the latest command based best practices and the Epilogue logging system. It is capable of playing the FRC 2022 game Rapid React.
- **Inlined Hatchbot** (Java, C++, Python): 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++, Python): 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.
- **Select Command Example** (Java, C++, Python): 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.

35.5 State-Space Examples

These examples demonstrate the use of the *State-Space Control*.

- **StateSpaceFlywheel** (Java, C++, Python): Demonstrates state-space control of a flywheel.
- **StateSpaceFlywheelSysId** (Java, C++, Python): Demonstrates state-space control using SysId's System Identification for controlling a flywheel.
- **StateSpaceElevator** (Java, C++, Python): Demonstrates state-space control of an elevator.
- **StateSpaceArm** (Java, C++, Python): Demonstrates state-space control of an Arm.

35.6 Simulation Physics Examples

These examples demonstrate the use of the physics simulation.

- **ElevatorSimulation** (Java, C++, Python): Demonstrates the use of physics simulation with a simple elevator.
- **ElevatorSimulation with Exponential PID** (Java, C++): Demonstrates the use of physics simulation of an elevator being controlled with exponential profiled PID.
- **ArmSimulation** (Java, C++, Python): Demonstrates the use of physics simulation with a simple single-jointed arm.
- **SimpleDifferentialDriveSimulation** (Java, C++): A barebones example of a basic drivetrain that can be used in simulation.

35.7 Miscellaneous Examples

These examples demonstrate miscellaneous WPILib functionality that does not fit into any of the above categories.

- **Addressable LED** (Java, C++, Python): Demonstrates the use of the `AddressableLED` class to control RGB LEDs for robot decoration and/or driver feedback.
- **Digital Communication** (Java, C++, Python): This is a sample program demonstrating how to communicate to a light controller from the robot code using the roboRIO's DIO ports.
- **I2C Communication** (Java, C++, Python): This is a sample program demonstrating how to communicate to a light controller from the robot code using the roboRIO's I2C port.
- **DMA** (Java, C++): Demonstrates the use of DMA (Direct Memory Access) to read from sensors without using the RoboRIO's CPU.
- **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++, Python): Demonstrates the use of the "rumble" functionality for tactile feedback on supported HIDs (such as XboxControllers).
- **Shuffleboard** (Java, C++, Python): Demonstrates configuring tab/widget layouts on the "Shuffleboard" dashboard from robot code through the `Shuffleboard` class's fluent builder API.
- **RomiReference** (Java, C++, Python): A command based example of how to run the *Romi robot*.
- **XRPReference** (Java, C++): A command based example of how to run the *XRP robot*.
- **Mechanism2d** (Java, C++, Python): A simple example of using *Mechanism2d*.
- **EventLoop** (Java, C++): Demonstrates the use of the `EventLoop` class that allows code to be called based on a boolean condition in the event-driven programming style.
- **UnitTest** (Java, C++): Shows how to do *Unit Testing*. The test files need to be in a separate Test directory (Java, C++).

36 Third Party Example Projects

This list helps you find example programs for use with third party devices. You can find support for many of these third parties on the *Support Resources* page.

- Cross The Road Electronics (CTRE)
- Kauai Labs (navX)
- Limelight (additional examples, called tutorials, can be found on the left)
- PhotonVision
- REV Robotics

37 Hardware - Basics

37.1 Introduction to FRC Robot Wiring

Note

This document details the wiring of a basic electronics board for the kitbot or to allow basic drivetrain testing.

Some images shown in this section reflect the setup for a Robot Control System using SPARK or SPARK MAX 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.

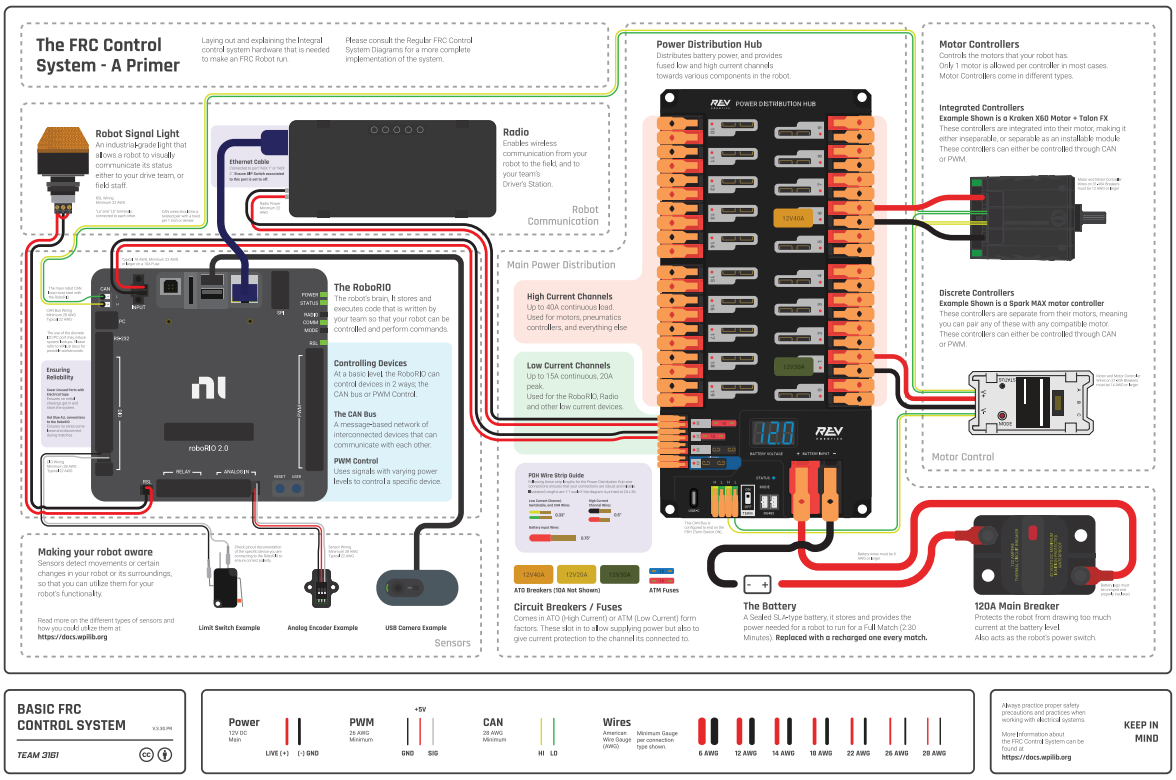
37.1.1 Overview

Warning

The video shown below contains the OM5P radio used through the 2024 season. Teams outside China and Chinese Taipei should see *Radio Power* for wiring info for the VH-109 radio used this season.

Basic

Diagram courtesy of FRC® Team 3161 and Stefen Acepcion.



REV

Diagram courtesy of FRC® Team 3161 and Stefen Acepcion.

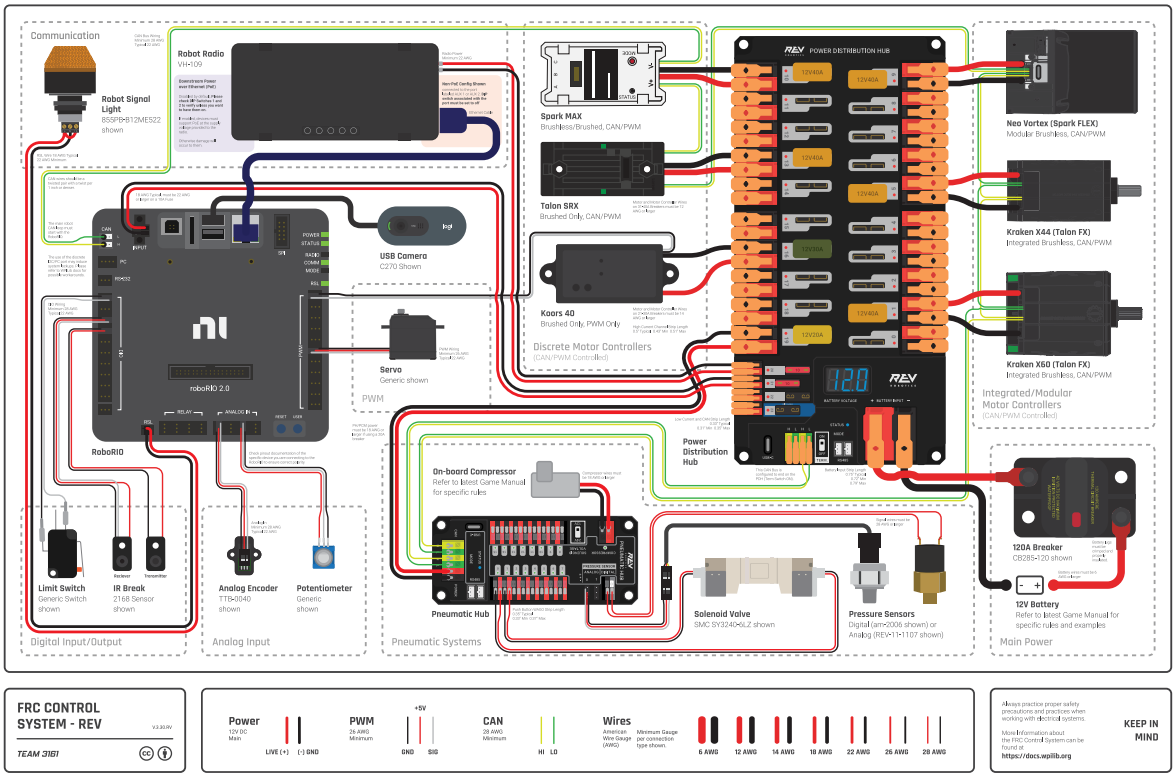
CTRE

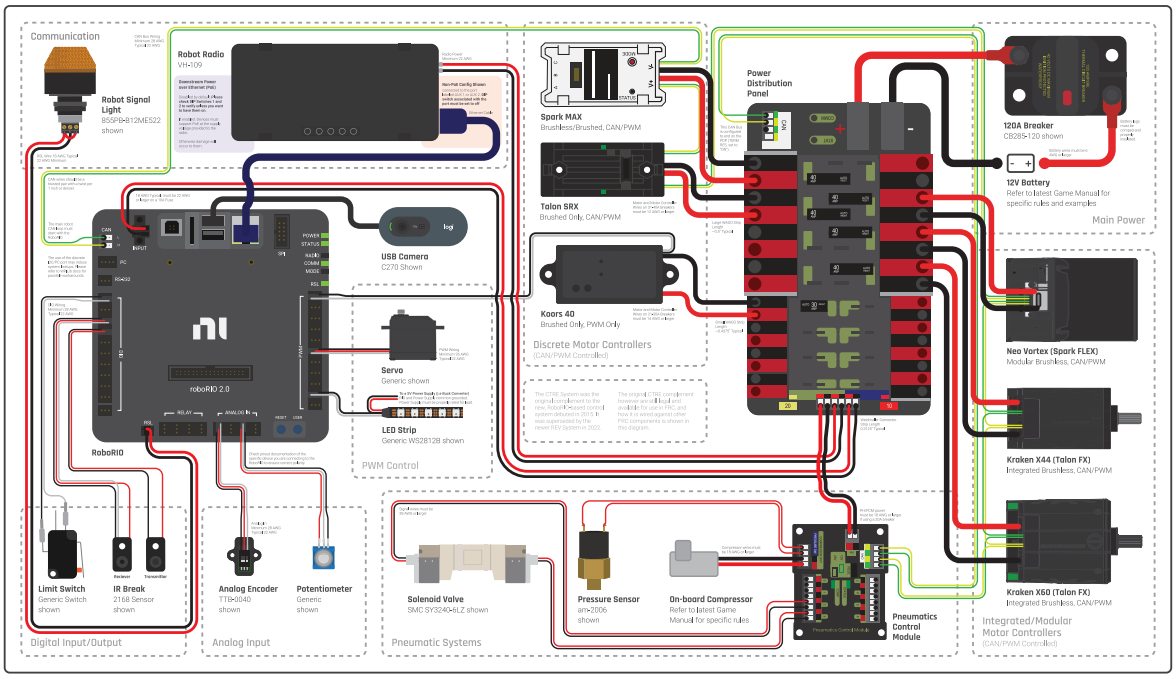
Diagram courtesy of FRC® Team 3161 and Stefen Acepcion.

37.1.2 Gather Materials

Locate the following control system components and tools

- **Kit Materials:**
  - Power Distribution Hub (*PDH*) / Power Distribution Panel (*PDP*)
  - roboRIO
  - Pneumatics Hub (*PH*) / Pneumatics Control Module (*PCM*)
  - VH-109 radio
  - Robot Signal Light (*RSL*)
  - 4x SPARK MAX or other motor controllers
  - 2x *PWM* y-cables
  - 120A Circuit breaker


Communication
Robot Radio
VH-109
Robot Signal Light
855PB-B12ME522 shown
USB Camera
C270 Shown
roboRIO 2.0
RoboRIO
Servo
Generic shown
PWM
Spark MAX
Brushless/Brushed, CAN/PWM
Talon SRX
Brushed Only, CAN/PWM
Koors 40
Brushed Only, PWM Only
Discrete Motor Controllers
(CAN/PWM Controlled)
POWER DISTRIBUTION HUB
Power Distribution Hub
Neo Vortex (Spark FLEX)
Modular Brushless, CAN/PWM
Kraken X44 (Talon FX)
Integrated Brushless, CAN/PWM
Kraken X60 (Talon FX)
Integrated Brushless, CAN/PWM
Integrated/Modular Motor Controllers
(CAN/PWM Controlled)
120A Breaker
CB285-120 shown
12V Battery
Refer to latest Game Manual for specific rules and examples
Main Power
On-board Compressor
Refer to latest Game Manual for specific rules
Pneumatic Hub
Solenoid Valve
SMC SY3240-6LZ shown
Pressure Sensors
Digital (am-2006 shown) or Analog (REV-11-1107 shown)
Pneumatic Systems
Limit Switch
Generic Switch shown
IR Break
2168 Sensor shown
Analog Encoder
TTB-0040 shown
Potentiometer
Generic shown
Digital Input/Output
Analog Input
FRC CONTROL SYSTEM - REV
TEAM 3161
Power
12V DC Main
LIVE (+)
(-) GND
PWM
26 AWG Minimum
GND
SIG
CAN
28 AWG Minimum
HI
LO
Wires
American Wire Gauge (AWG)
Minimum Gauge per connection type shown.
6 AWG
12 AWG
14 AWG
18 AWG
22 AWG
26 AWG
28 AWG
KEEP IN MIND
More Information about the FRC Control System can be found at
https://docs.wpilib.org


Communication
Robot Radio
VH-109
Robot Signal Light
855PB-B12ME522 shown
USB Camera
C270 Shown
roboRIO 2.0
RoboRIO
Servo
Generic shown
LED Strip
Generic WS2812B shown
PWM Control
Spark MAX
Brushless/Brushed, CAN/PWM
Talon SRX
Brushed Only, CAN/PWM
Koors 40
Brushed Only, PWM Only
Discrete Motor Controllers
(CAN/PWM Controlled)
Power Distribution Panel
Neo Vortex (Spark FLEX)
Modular Brushless, CAN/PWM
Kraken X44 (Talon FX)
Integrated Brushless, CAN/PWM
Kraken X60 (Talon FX)
Integrated Brushless, CAN/PWM
Integrated/Modular Motor Controllers
(CAN/PWM Controlled)
120A Breaker
CB285-120 shown
12V Battery
Refer to latest Game Manual for specific rules and examples
Main Power
Solenoid Valve
SMC SY3240-6LZ shown
Pressure Sensor
am-2006 shown
On-board Compressor
Refer to latest Game Manual for specific rules
Pneumatics Control Module
Pneumatic Systems
Limit Switch
Generic Switch shown
IR Break
2168 Sensor shown
Analog Encoder
TTB-0040 shown
Potentiometer
Generic shown
Digital Input/Output
Analog Input
FRC CONTROL SYSTEM - CTRE
TEAM 3161
Power
12V DC Main
LIVE (+)
(-) GND
PWM
26 AWG Minimum
GND
SIG
CAN
28 AWG Minimum
HI
LO
Wires
American Wire Gauge (AWG)
Minimum Gauge per connection type shown.
6 AWG
12 AWG
14 AWG
18 AWG
22 AWG
26 AWG
28 AWG
KEEP IN MIND
More Information about the FRC Control System can be found at
https://docs.wpilib.org

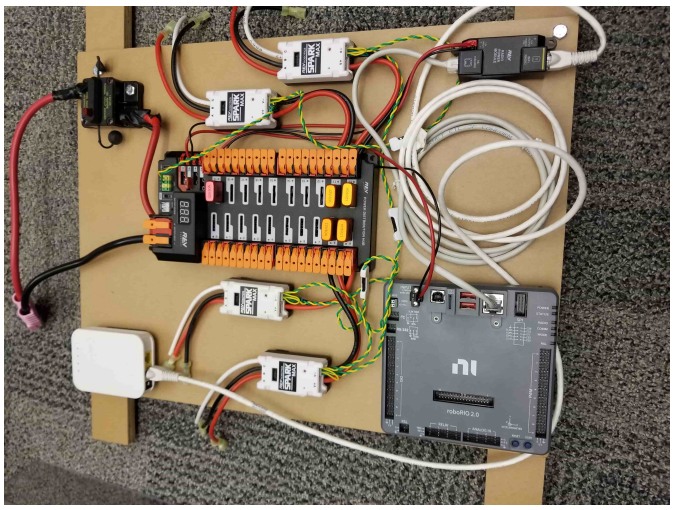
  - 4x 40A Circuit breaker
  - 6 AWG (16 mm^2) Red wire
  - 10 AWG (6 mm^2) Red/Black wire
  - 18 AWG (1 mm^2) Red/Black wire
  - 22 AWG (0.5 mm^2) Yellow/Green twisted *CAN* cable
  - 8x Pairs of 10-12 AWG (4 - 6 mm^2) (Yellow) quick disconnect terminals (16x ring terminals if using integrated wire controllers)
  - 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)
  - Wire cutters, strippers, and crimpers
  - 7/16” (11 mm may work if imperial is unavailable) box end wrench or nut driver
  - Additional 7/16” wrench/nut driver or Philips head screw driver
  - For CTR PDP only: 5 mm Hex key (3/16” may work if metric is unavailable)
  - For CTR PDP only: 1/16” Hex key

37.1.3 Create the Base for the Control System

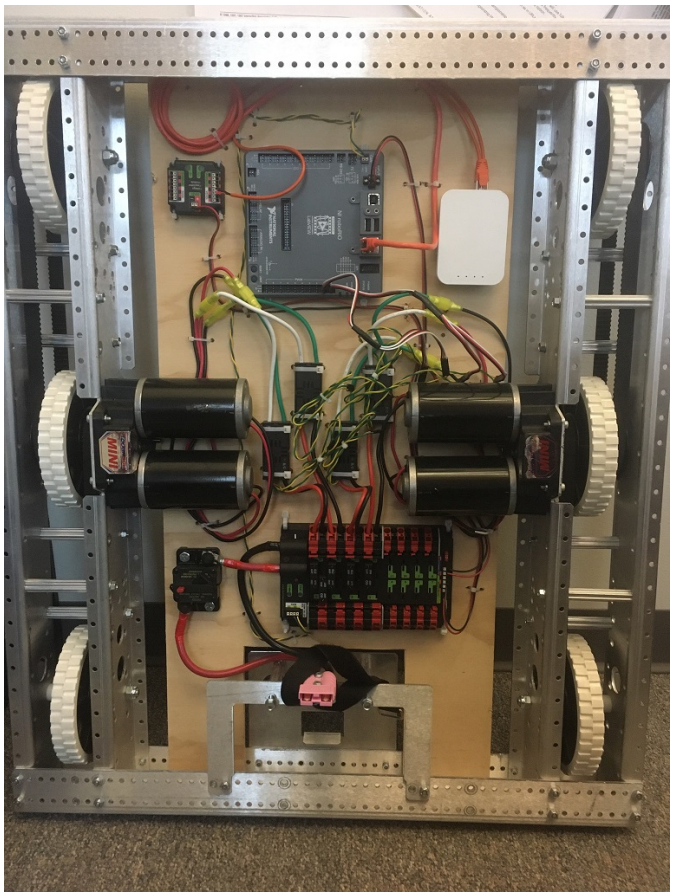
For a test board, cut piece of 1/4” or 1/2” (6-12 mm) material (wood or plastic) approximately 24” x 16” (60 x 40 cm). For a Robot Quick Build control board see the supporting documentation for the proper size board for the chosen chassis configuration.

37.1.4 Layout the Core Control System Components

REV



CTR

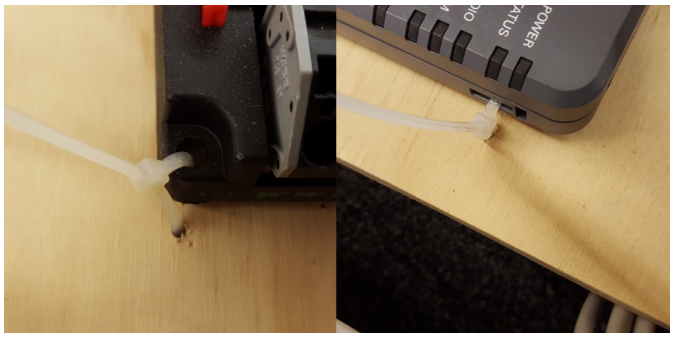


Lay out the components on the board. An example layout is shown in the image above.

Note

A Vivid-Hosting has a guide for mounting the VH-109 radio.

37.1.5 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.

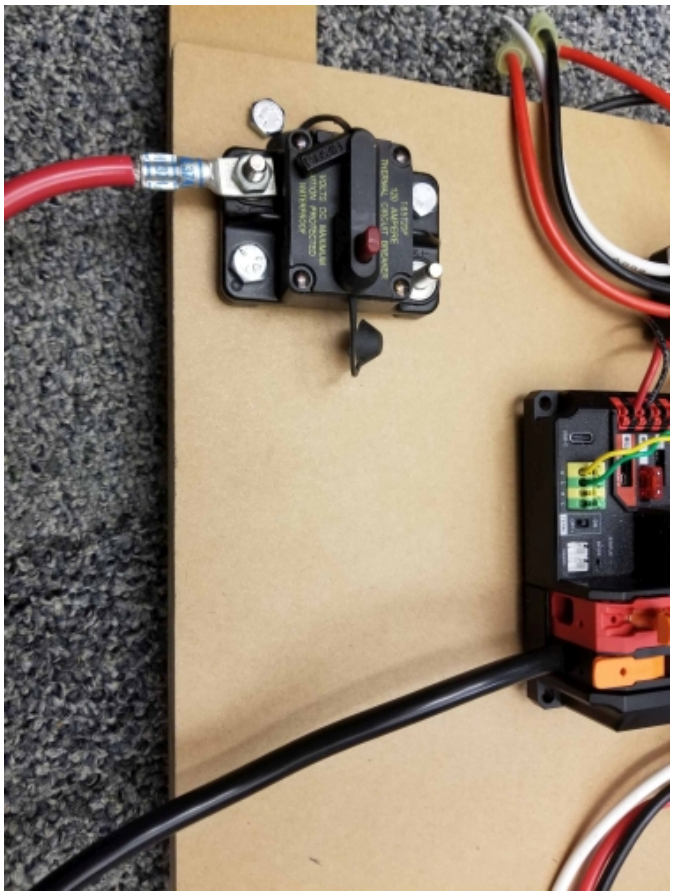
37.1.6 Attach Robot Side Battery Connector

REV

The next step will involve using the Wago connectors on the PDH. To use the Wago connectors, open the lever, insert the wire, then close the lever. Two sizes of Wago connector are found on the PDH:

- Main power connectors: Accept 4 - 18 AWG (.75 - 25 mm^2), strip 20 mm (~3/4”)
- High current channel connectors: Accept 8 - 24 AWG (.25 - 10 mm^2), strip 12 mm (~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.



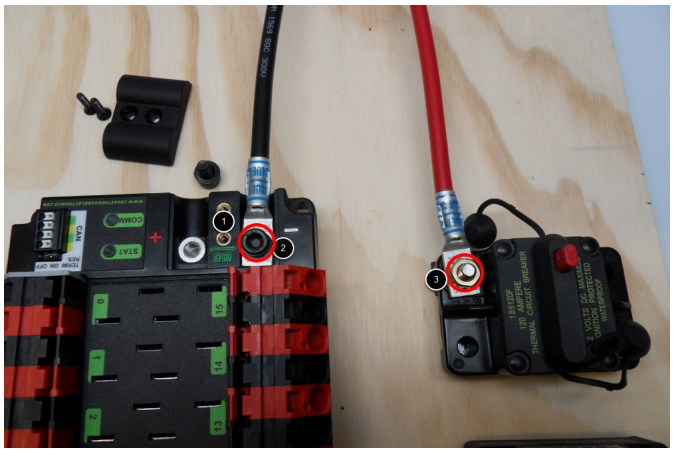
Requires: Battery Connector, 6 AWG (16 mm^2) terminal lugs, 7/16” (11 mm) Box end

Attach terminal lug to positive (red) wire of battery connector. Strip .75” off the black wire.

Lift the lever above the black main power input terminal on the PDH until it clicks into place. Insert the wire. Pull the lever down to secure the wire.

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

CTR



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

37.1.7 Wire Breaker to Power Distribution

REV

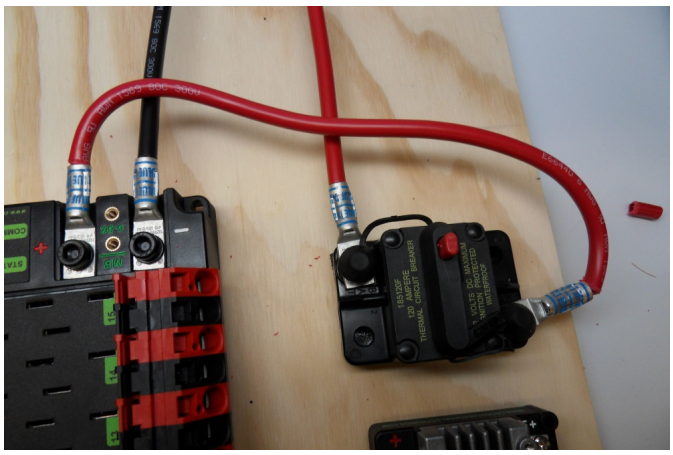


Requires: 6 AWG (16 mm^2) red wire, 1x 6 AWG (16 mm^2) terminal lugs, 7/16” (11 mm) wrench

Secure one terminal lug to the end of the 6 AWG (16 mm^2) red wire. Using the 7/16” (11 mm) wrench, 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 and strip the other end of the wire). Measure out the length of wire required to reach the positive terminal of the PDH.

1. Cut and strip the other end of the red wire.
2. Using the 7/16” (11 mm) wrench, secure the wire to the “AUX” side of the 120A main breaker.
3. Lift the lever on the positive (red) input terminal of the PDH, insert the wire, then close the terminal.

CTR



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.

37.1.8 Insulate power connections

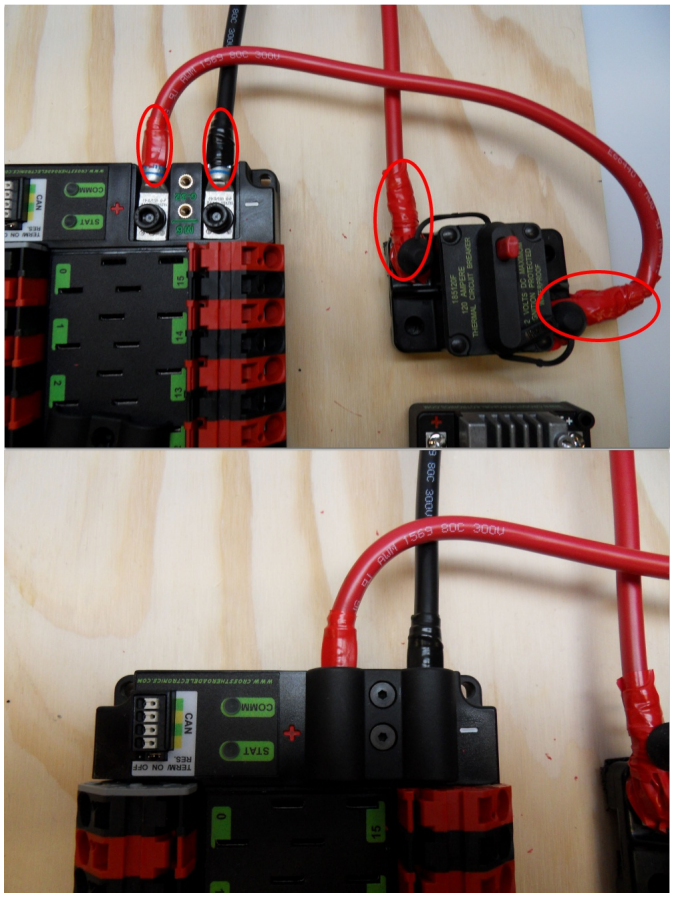
REV



Requires: Electrical tape

Using electrical tape, insulate the two connections to the 120A breaker.

CTR

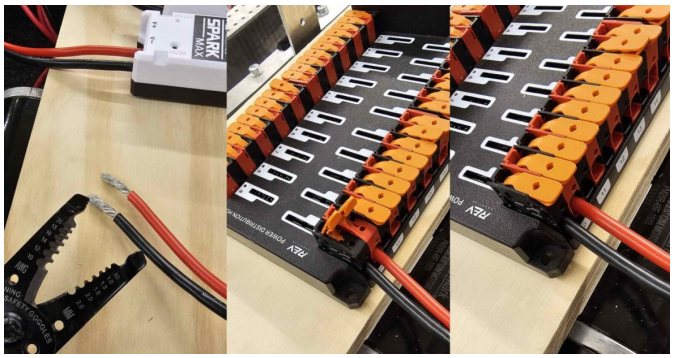


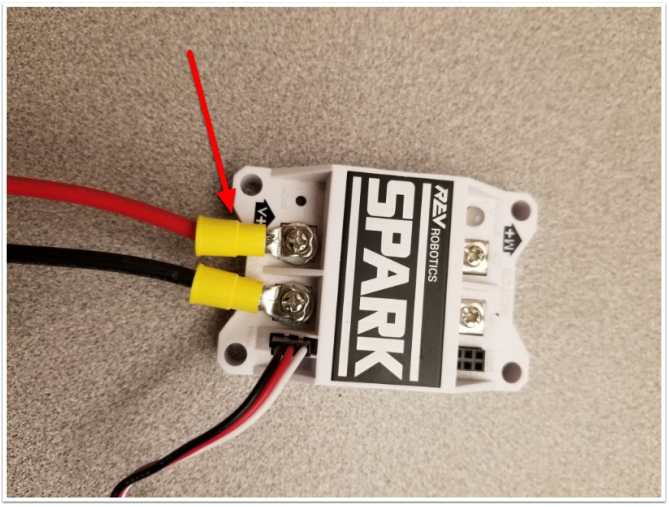
Requires: 1/16” Allen, Electrical tape

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.
2. Using the 1/16” Allen wrench, replace the PDP terminal cover

37.1.9 Motor Controller Power

REV





Requires: Wire Stripper Terminal Controllers only: 10 or 12 AWG (4 - 6 mm^2) wire , 10 or 12 AWG (4 - 6 mm^2) fork/ring terminals, 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 Wago terminal pairs.

For terminal motor controllers (bottom image):

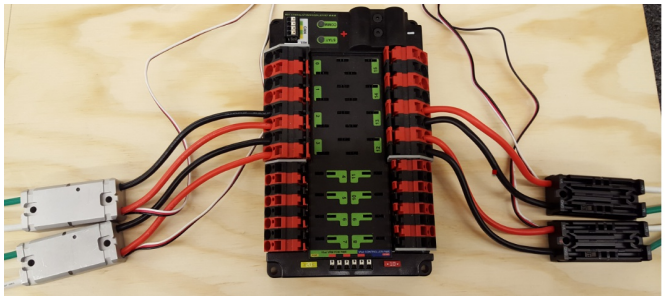
1. Cut red and black wire to appropriate length to reach from one of the Wago terminal pairs to the input side of the motor controller (with a little extra for the length that will be inserted into the terminals on each end)
2. Strip one end of each of the wires, then insert into the Wago terminals.
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 -)

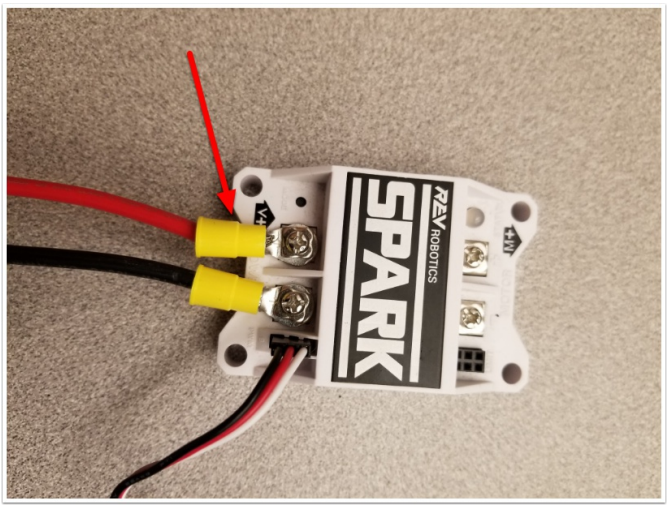
CTR

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 mm^2), strip 11-12 mm (~7/16”)
- Large Wago connector: Accepts 6 - 12 AWG (4 - 16 mm^2), strip 12-13 mm (~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.





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

37.1.10 Weidmuller Connectors

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)"
- A ferrule can be added to reinforce the connection and prevent splaying, as shown in the video above.
- 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. Occasionally the terminal may remain stuck open with the wire inserted and the button released even if the wire is stripped and inserted properly; in these cases wiggling the wire in and out a small amount will often allow the connector to latch shut and grip the wire.

37.1.11 roboRIO Power

REV

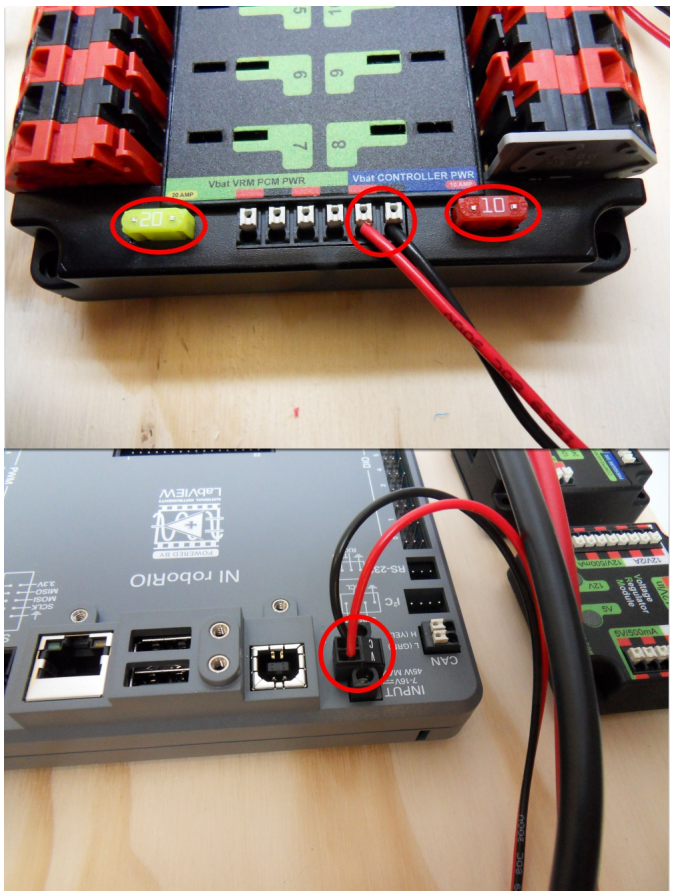

INPUT
7-16V
45W MAX
V
C
L (GRN)
H (YEL)
CAN
3.3V SDA
SCL
I²C
RS-232
TXD
RXD



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

1. Insert the 10A fuse into the PDH in one of the non-switchable fused channels (20-22).
2. Strip ~5/16” (~8 mm) on both the red and black 18 AWG (1 mm^2) wire and connect to the corresponding terminals on the PDH channel where the fuse was installed
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.

CTR



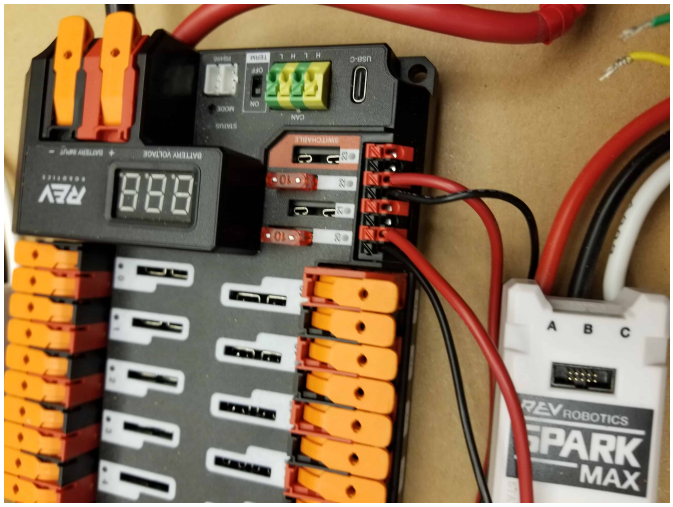
Requires: 10A/20A mini fuses, Wire stripper, very small flat screwdriver, 18 AWG (1 mm^2) Red and Black

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 ~5/16” (~8 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.

37.1.12 Radio Power

Vivid-Hosting Wiring Guide

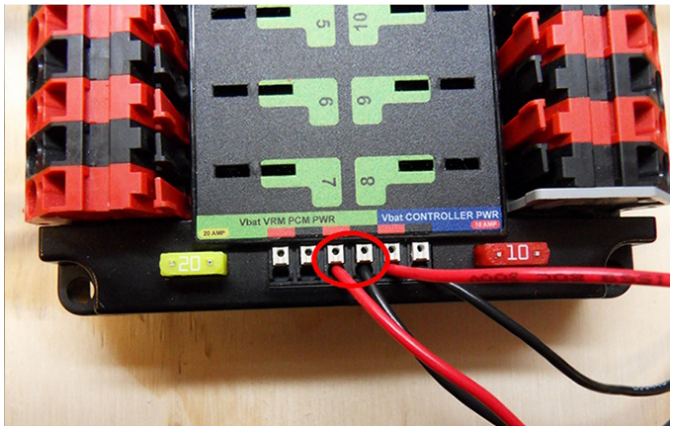
REV



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

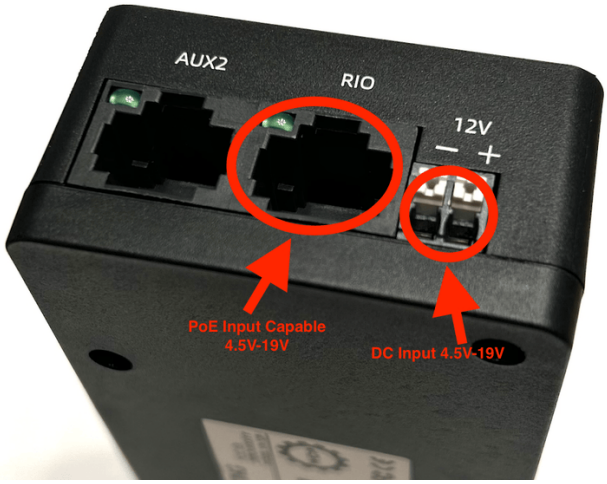
1. Insert the 10A fuse into the PDH in one of the non-switchable fused channels (20-22).
2. Strip ~5/16” (~8 mm) on the end of the red and black 18 AWG (1 mm^2) wire and connect the wire to the corresponding terminals on the PDH.

CTR



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

1. Strip ~5/16” (~8 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 “12V” Weidmuller terminals on the VH-109 Radio. 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 ~5/16” (~8 mm) from the end of the wire.
5. Connect the wire to the VH-109 “12V” Weidmuller terminals.

Note

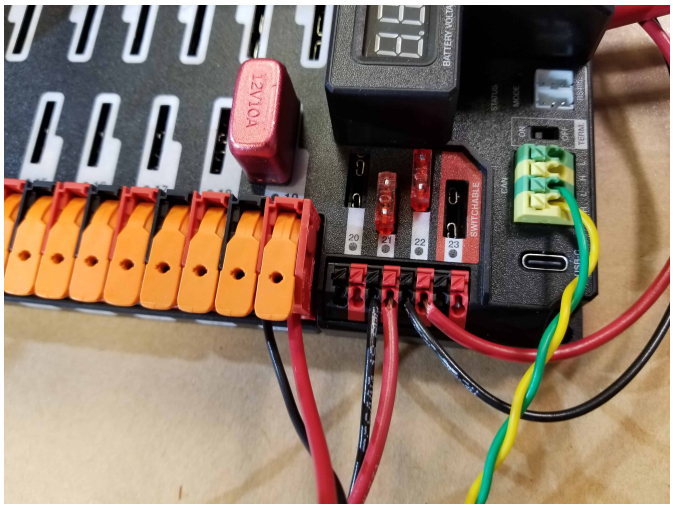
The radio can additionally be powered by *PoE* to create a more redundant connection.

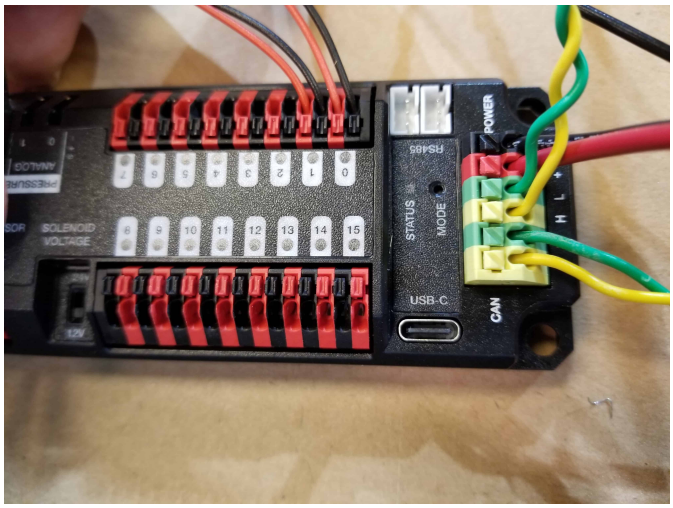
Warning

If powering the radio this way, the roboRIO must be connected via a PoE injector, modified ethernet cable, or to one of the *AUX* ports of the VH-109 with the corresponding DIP switch in the off (default) position.

37.1.13 Pneumatics Power (Optional)

REV





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

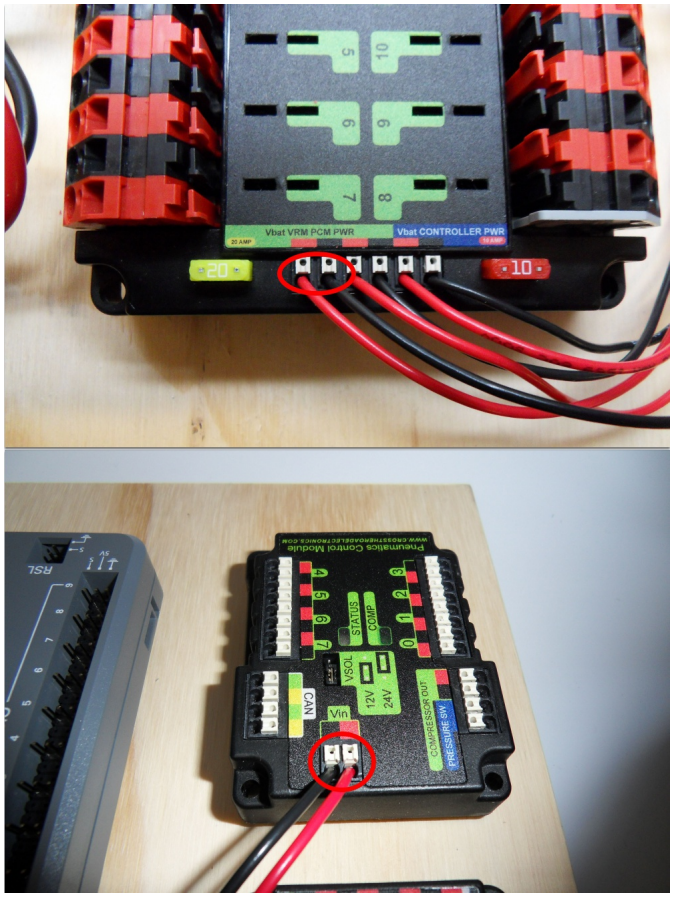
> **Note**
>
> The Pneumatics Hub is an optional component used for controlling pneumatics on the robot.

The Pneumatics Hub can be wired to either a non-switchable fused port on the PDH with a 15A or smaller fuse or to a circuit breaker protected port with a breaker up to 20A.

1. Strip ~5/16” (~8 mm) on the end of the red and black 18 AWG (1 mm^2) wire.
2. Connect the wire to the PDH in one of the two ways described above
3. Measure the length required to reach the red terminals on the short end of the PH labeled +/-. 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 ~5/16” (~8 mm) from the other end of the wire.
5. Connect the wire to the PH input terminals.

CTR



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

Note

The PCM is an optional component used for controlling pneumatics on the robot.

1. Strip ~5/16” (~8 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 ~5/16” (~8 mm) from the end of the wire.
5. Connect the wire to the PCM 12Vin terminals.

37.1.14 Ethernet Cables



Requires: an Ethernet cable

1. Connect an Ethernet cable from the RJ45 (Ethernet) socket of the roboRIO to the *RIO* port on the VH-109 Radio.

Note

If using *PoE* you may need a second Ethernet cable or longer power wires.

Warning

If the VH-109 is powered using the 12V input terminals, the roboRIO should be connected to the VH-109 through a PoE injector, a modified ethernet cable, or to one of the *AUX* ports on the VH-109 instead of the *RIO* port with the corresponding DIP switch in the off (default) position.

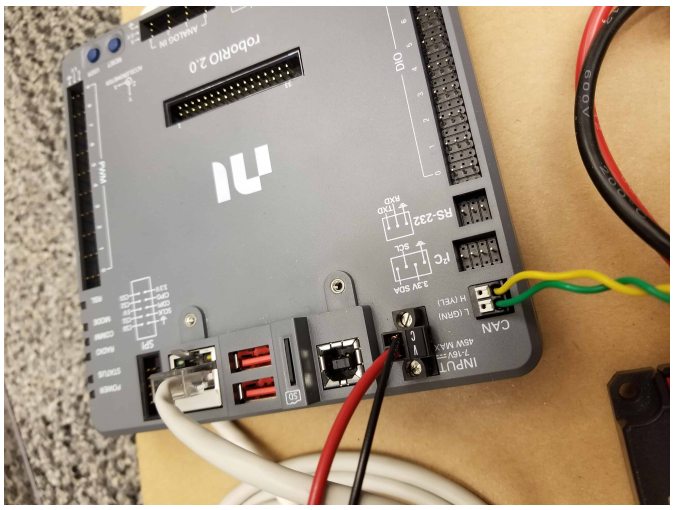
Warning

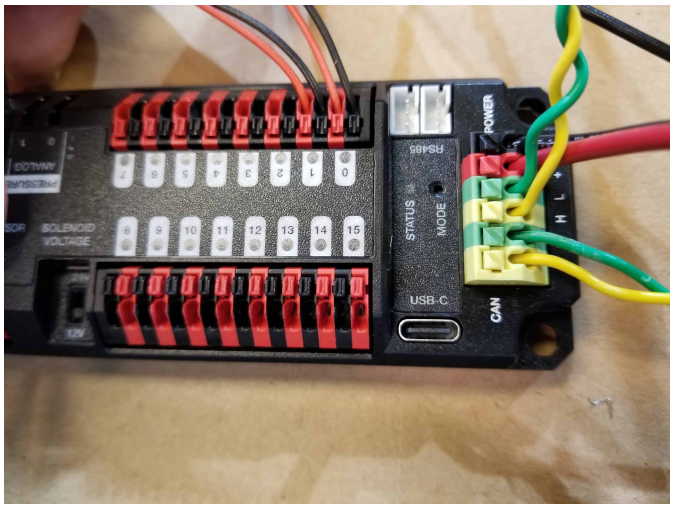
DO NOT connect the Rev passive *PoE* injector cable directly to the roboRIO. The roboRIO MUST connect to the socket end of the cable using an additional Ethernet cable as shown in the next step.

37.1.15 CAN Devices

roboRIO to Pneumatics CAN

REV



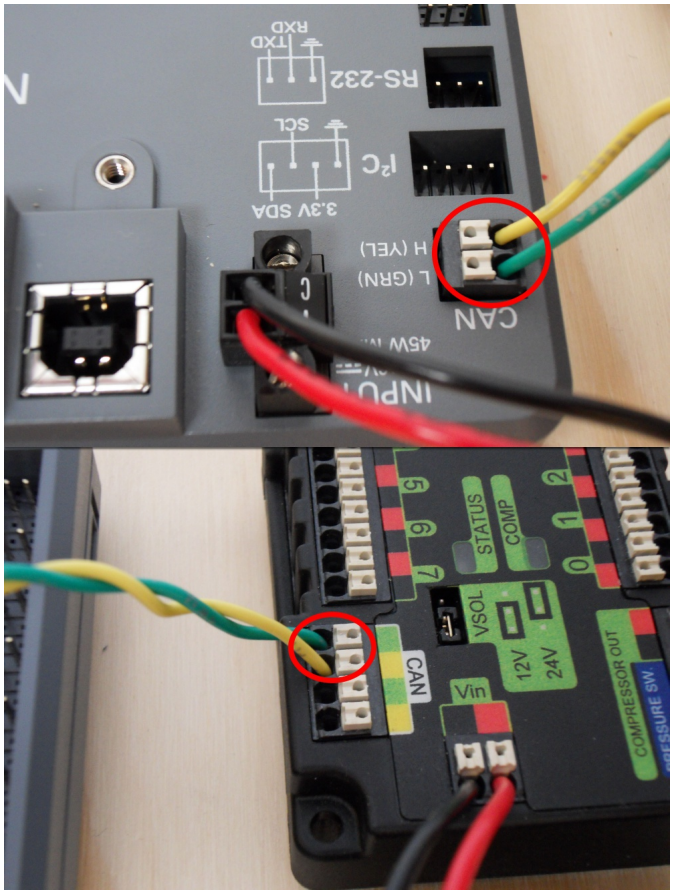


Requires: Wire stripper, small flat screwdriver (optional), yellow/green twisted CAN cable

> **Note**
>
> The PH is an optional component used for controlling pneumatics on the robot. If you are not using the PH, wire the CAN connection directly from the roboRIO (shown in this step) to the PDH (shown in the next step).

1. Strip ~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 ~5/16” (~8 mm) off this end of the wires.
4. Insert the wires into the appropriate color coded CAN terminals on the PH. You may use either of the Yellow/Green terminal pairs on the PH, there is no defined in or out.

CTR



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

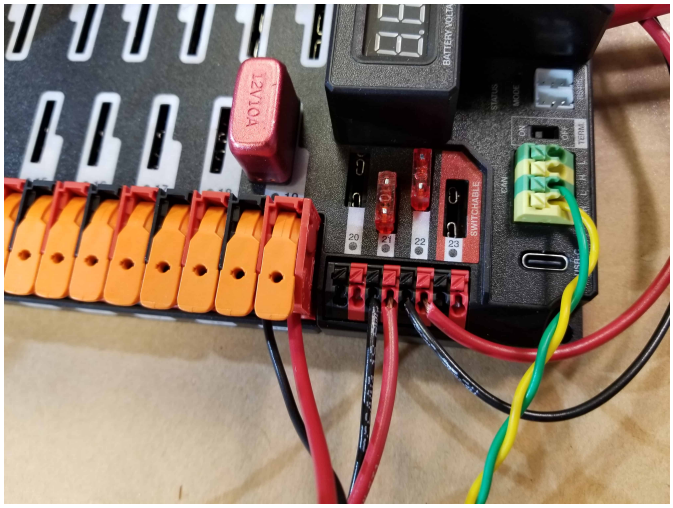
Note

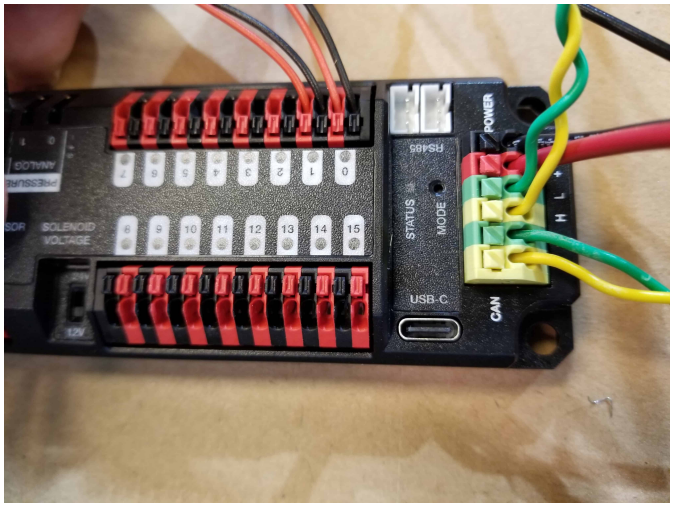
The PCM is an optional component used for controlling pneumatics on the robot. If you are not using the PCM, wire the CAN connection directly from the roboRIO (shown in this step) to the PDP (shown in the next step).

1. Strip ~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 ~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.

Pneumatics to PD CAN

REV





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

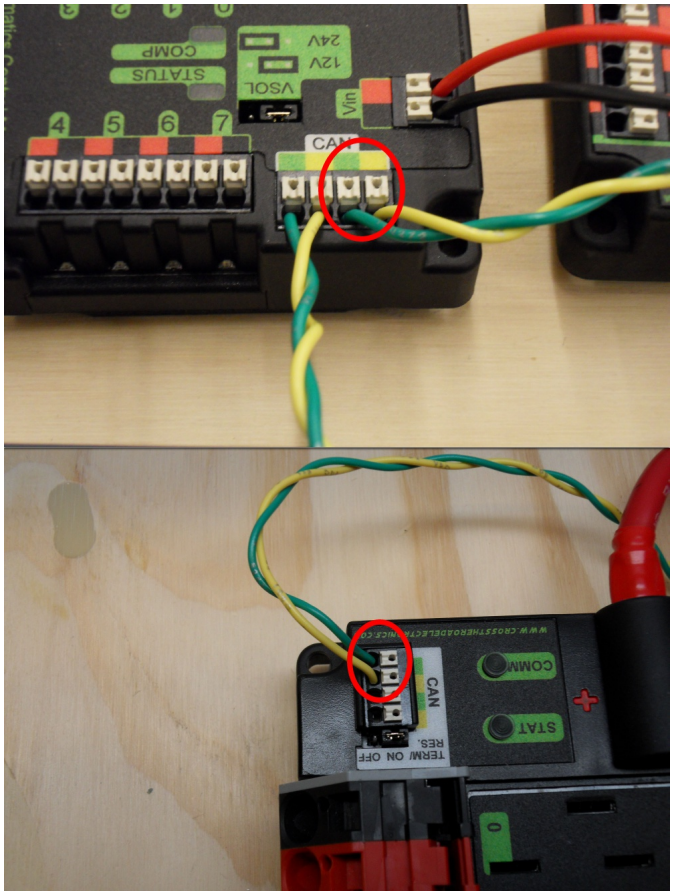
> **Note**
>
> The PH is an optional component used for controlling pneumatics on the robot. If you are not using the PH, wire the CAN connection directly from the roboRIO (shown in the above step) to the PDH (shown in this step).

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

Note

See the *CAN Wiring Basics* if you need to terminate the CAN bus somewhere other than the PDP.

CTR



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

Note

The PCM is an optional component used for controlling pneumatics on the robot. If you are not using the PCM, wire the CAN connection directly from the roboRIO (shown in the above step) to the PDP (shown in this step).

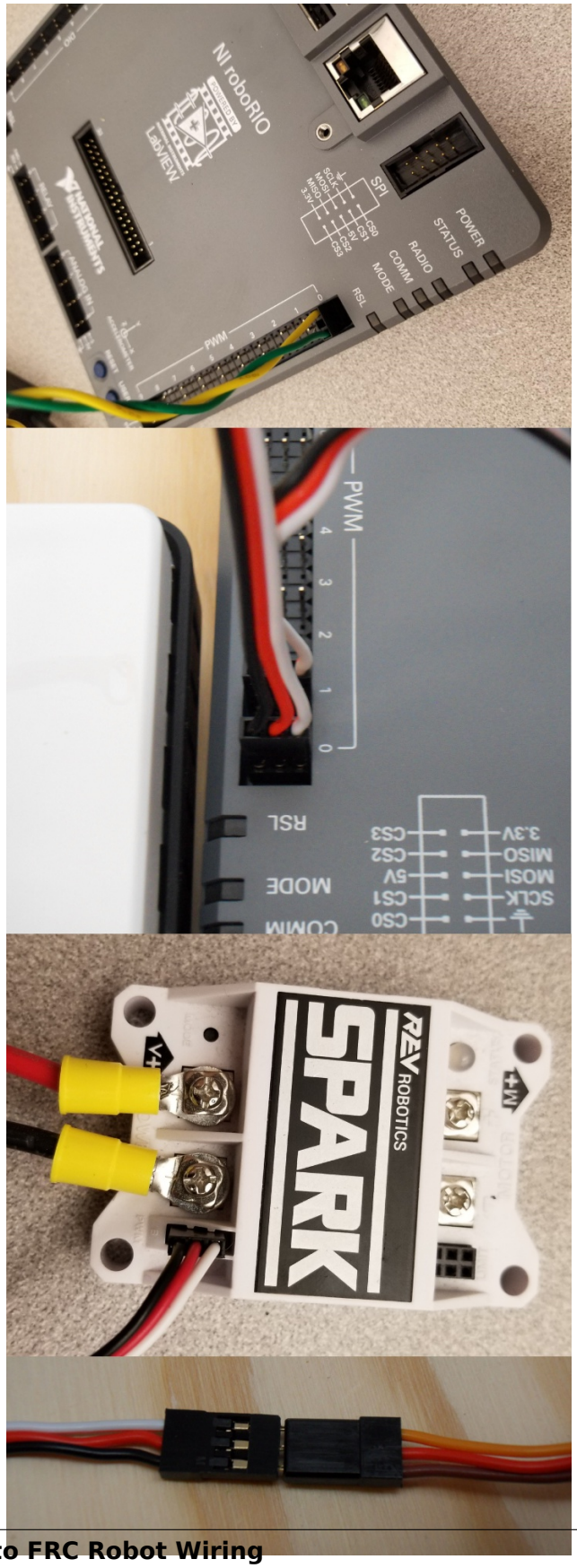
1. Strip ~5/16” (~8 mm) off of each of the CAN wires.
2. Insert the wires into the appropriate CAN terminals on the PCM.
3. Measure the length required to reach the CAN terminals of the PDP (either of the two available pairs). Cut and strip ~5/16” (~8 mm) off this end of the wires.
4. Insert the wires 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.

Note

See the *CAN Wiring Basics* if you need to terminate the CAN bus somewhere other than the PDP.

37.1.16 Motor Controller Signal Wires

PWM


NI roboRIO
LabVIEW
SPI
POWER
STATUS
RADIO
COMM
MODE
RSL
PWM
3.3V
MISO
MOSI
SCLK
CS0
CS1
CS2
CS3
5V
REV ROBOTICS
SPARK

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.

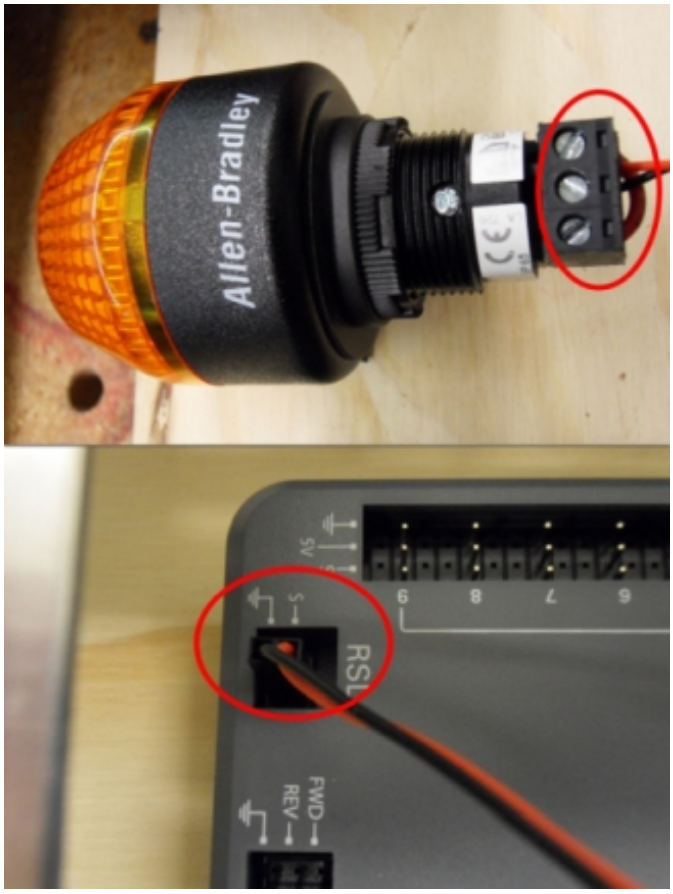
CAN

The Spark MAX controllers can also be wired using CAN. When wiring CAN the objective is to create a single complete bus running from the roboRIO on one end and running through all CAN devices on the robot. It is recommended to have either Power Distribution device at the other end of the bus because they have built-in termination. If you do not wish to locate one of these devices at the end of the bus see *CAN Wiring Basics* for info about terminating yourself.

The Spark MAX controllers come with CAN cables that are pre-terminated with connectors. You can chain these cables together directly, or buy or build extension cables to bridge larger gaps. To connect to other CAN devices such as pneumatics controllers, power distribution boards, or the roboRIO you will need to either cut off one of these pre-terminated connectors on the controller, cut off a connector on an extension, or build your own extension with just a single connector.

When chaining controllers together using the provided connectors, make sure to use the provided retaining clip. If unavailable, secure the connection with a small ziptie, electrical tape, or other similar method.

37.1.17 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)

37.1.18 Circuit Breakers

REV

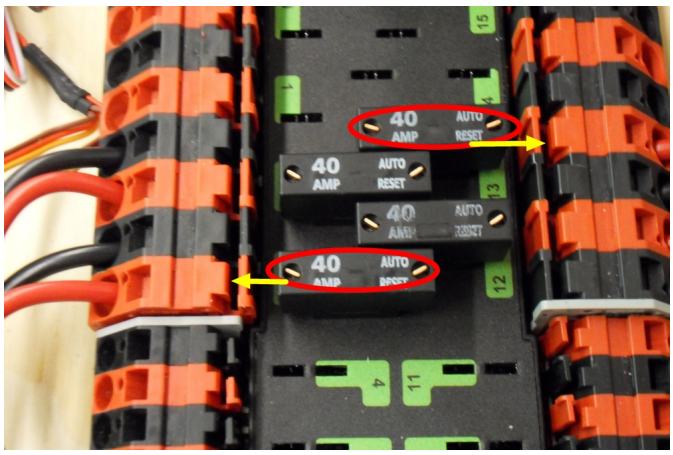


Requires: 4x 40A circuit breakers

Insert 40-amp Circuit Breakers into the positions on the PDH corresponding with the Wago connectors the motor controllers are connected to. Note that the white graphic indicates which breakers are associated with which terminal pairs.

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

CTR

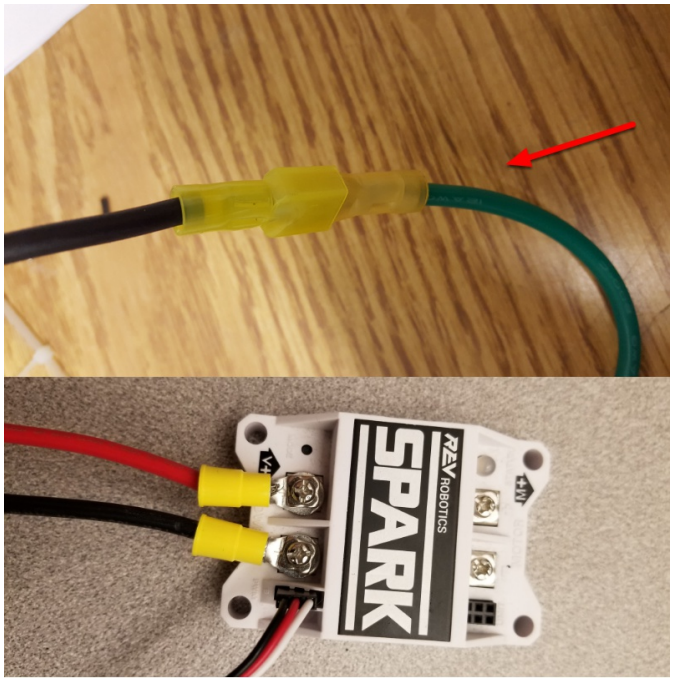


Requires: 4x 40A circuit breakers

Insert 40-amp Circuit Breakers into the positions on the PDP corresponding with the Wago connectors the motor controllers 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.

37.1.19 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 -)

37.1.20 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.

- 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 V+ terminal on the motor controller (not M+!!!!)

- For each non-motor controller device, verify that the red wire runs from a red terminal on the PD connects to a red terminal on the component.

> **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.

37.1.21 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.

37.1.22 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. If you hear any clicking, or see any smoke, power the system off immediately, clicking is likely the sound of circuit breakers tripping.

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 controls 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!

37.2 Wiring Best Practices

> **Tip**
>
> The article *Intro to FRC Robot Wiring* 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.

37.2.1 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.

37.2.2 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 VH-109 radio with a *PoE* connection in addition to the direct Weidmuller radio power. 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.

37.2.3 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.

37.2.4 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.

37.2.5 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!)

37.2.6 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.

37.2.7 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 rotate 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.

37.2.8 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.

37.2.9 Battery Maintenance

Take good care of your batteries! A bad battery can easily cause a robot to functional 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 in-

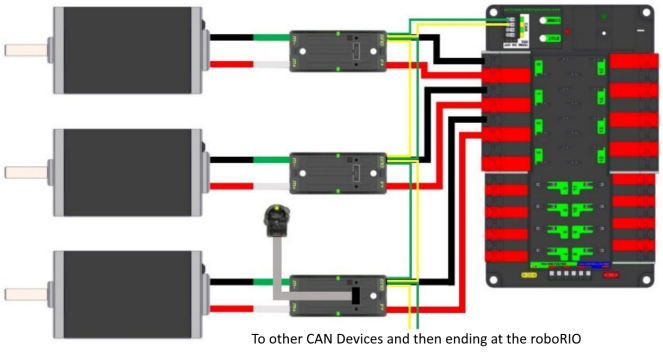
ternal resistance. This measurement is much more meaningful when it comes to match performance than a simple no-load voltage number provided by a multimeter.

37.2.10 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.

37.3 CAN Wiring Basics

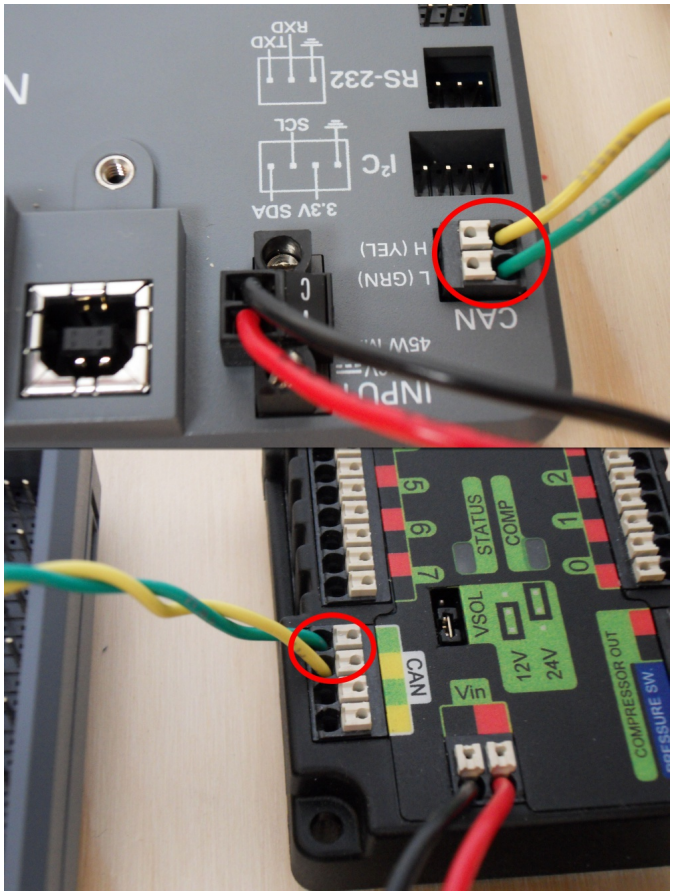
CAN is a two wire network that is designed to facilitate communication between multiple devices on your robot. It is recommended that CAN on your robot follow a "daisy-chain" topology. This means that the CAN wiring should usually start at your roboRIO and go into and out of each device successively until finally ending at the *PDP*.



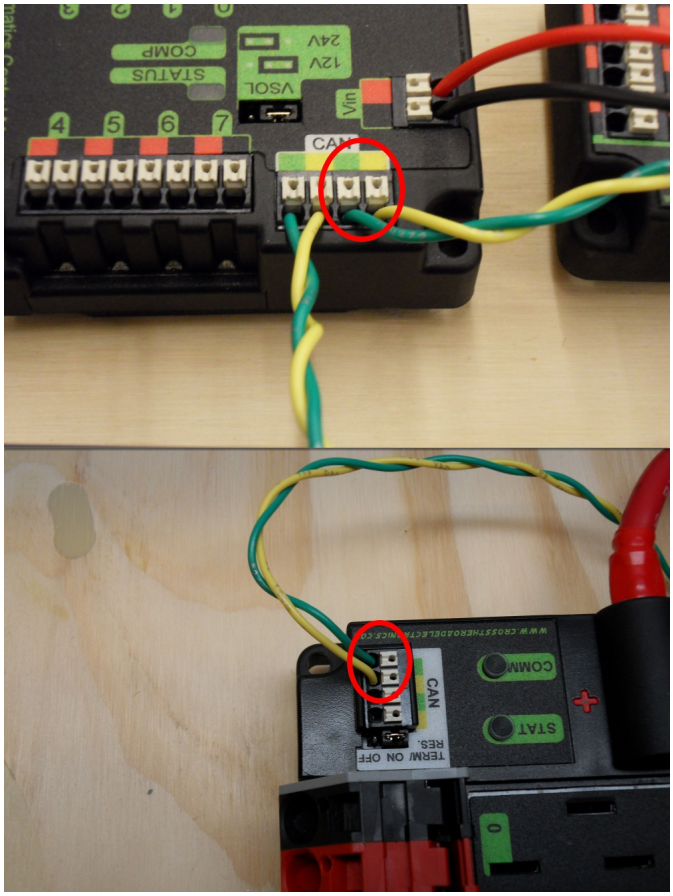
37.3.1 Standard Wiring

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.



37.3.2 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.

37.4 Wiring Pneumatics - CTRE Pneumatic Control Module

This page describes wiring pneumatics with the CTRE Pneumatic Control Module (PCM). For instructions on wiring pneumatics with the REV Pneumatic Hub (PH) see *this page*.

Hint

For pneumatics safety & mechanical requirements, consult this year's Robot Construction rules. For mechanical design guidelines, refer to the FIRST Pneumatics Manual.

See also

For Pneumatic Control Module configuration and firmware updates using Phoenix Tuner, see the CTRE documentation:

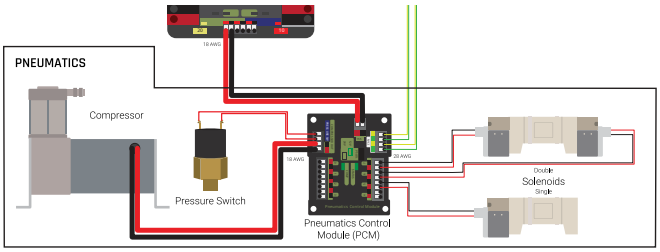
- CAN Bus Setup - Device IDs, firmware updates, and troubleshooting
- PCM Bring-Up - Testing solenoids and compressor operation

37.4.1 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 or PDH.

For complicated robot designs requiring more channels or multiple solenoid voltages, additional PCMs, or PHs can be added to the control system.

37.4.2 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 or from a 15 amp or 20 amp port on the PDH (20 amp recommended if controlling a compressor). 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 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.

37.4.3 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.

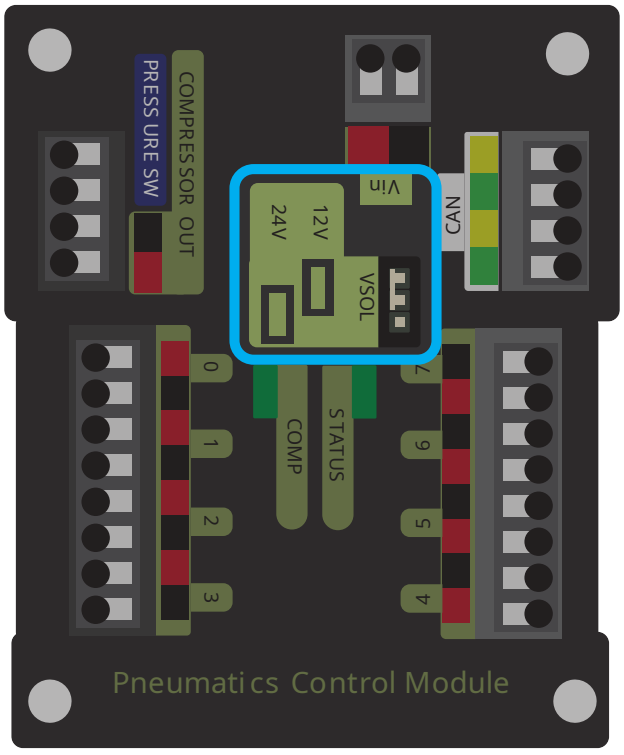
37.4.4 The Pressure Switch

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).

37.4.5 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.

37.4.6 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.

37.5 Wiring Pneumatics - REV Pneumatic Hub

This page describes wiring pneumatics with the REV Pneumatic Hub (*PH*). For instructions on wiring pneumatics with the CTRE Pneumatic Control Module (*PCM*) see *this page*.

Hint

For pneumatics safety & mechanical requirements, consult this year's Robot Construction rules. For mechanical design guidelines, refer to the FIRST Pneumatics Manual.

See also

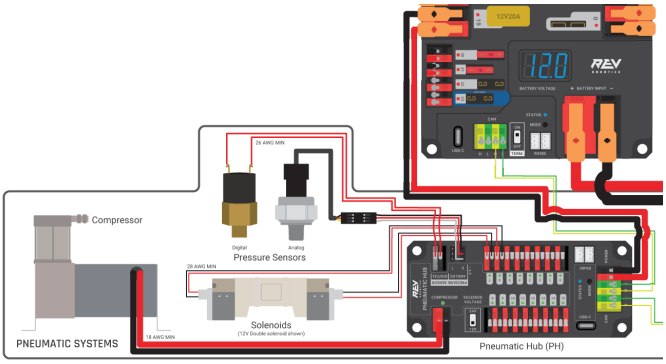
For Pneumatic Hub configuration, firmware updates, and detailed technical specifications, see the REV Pneumatic Hub Documentation.

37.5.1 Wiring Overview

A single PH will support most pneumatics applications, providing an output for the compressor, input for a pressure switch, and outputs for up to 16 solenoid channels (12V or 24V selectable). The module is connected to the roboRIO over the *CAN* bus and powered via 12V from the PDP/PDH.

For complicated robot designs requiring more channels or multiple solenoid voltages, additional PHs, or PCMs can be added to the control system.

37.5.2 PCM Power and Control Wiring



The first PH on your robot can be wired from the PDP VRM/PCM connectors on the end of the PDP or from a 15A or 20A port on the PDH (20 amp recommended if controlling a compressor). The PH 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 Power (Optional)*

Additional PHs can be wired to a standard WAGO connector on the side of the PDP and protected with a 20A or smaller circuit breaker or to a 15A port on the PDH. Additional PHs should also be placed anywhere in the middle of the CAN chain.

37.5.3 The Compressor

The compressor can be wired directly to the Compressor connectors on the PH. If additional length is required, make sure to use 18 AWG wire or larger for the extension.

37.5.4 The Pressure Switch

The PH has two options for detecting pressure, a digital pressure switch, or an analog pressure switch.

Digital

A digital pressure switch should be connected directly to the digital pressure sensor input terminals on the PCM. There is no polarity on the input terminals or on the pressure switch itself, either terminal on the PH 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).

Analog

An analog pressure switch (REV-11-1107 can be connected directly to the analog pressure sensor port 0 input terminals. Using an analog pressure sensor allows reading the pressure in the pneumatic system through code and setting custom trigger thresholds for turning on and off the compressor.

> **Warning**
>
> The Analog Pressure Sensor port is a very tight fit and requires special attention. See REV Wiring an Analog Pressure Sensor for more tips

37.5.5 Solenoids

Each solenoid channel should be wired directly to a numbered pair of terminals on the PH. 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.

37.5.6 Solenoid Voltage Switch

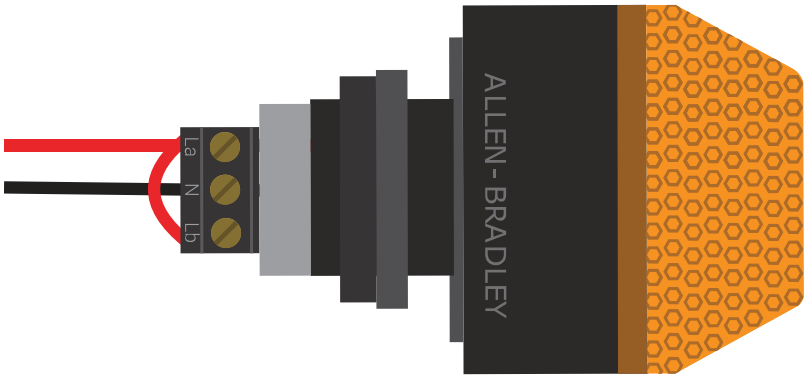
The PH is capable of powering either 12V or 24V solenoids, but all solenoids connected to a single PH must be the same voltage. Set the voltage switch to the appropriate voltage for solenoids prior to use.

37.6 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.

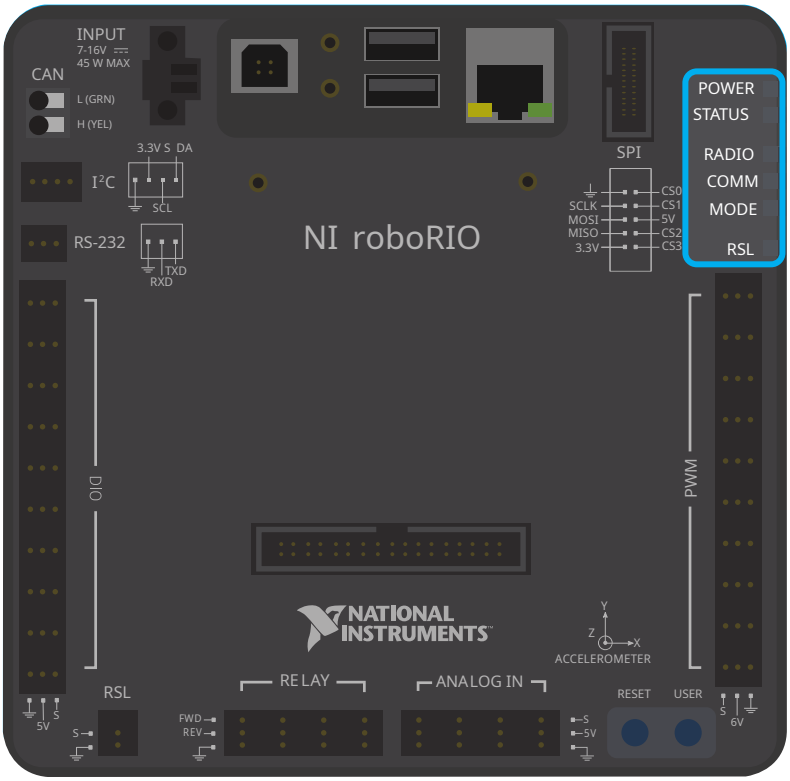
A compact and printable Status Light Quick Reference is available.

37.6.1 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

37.6.2 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	*See above*	

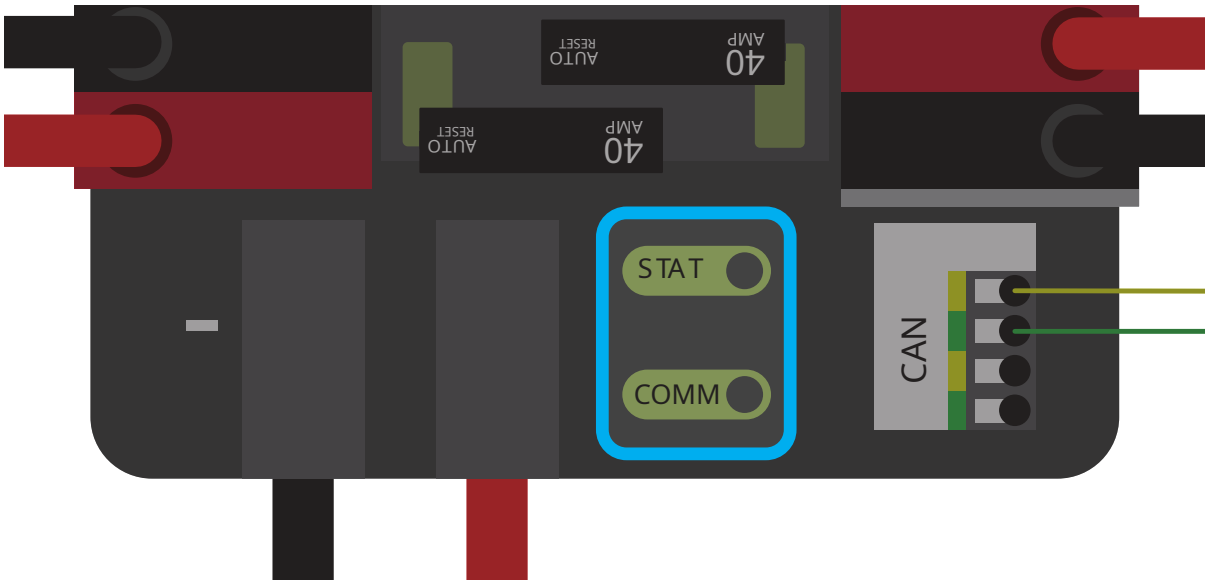
37.6.3 Vivid-Hosting VH-109 Radio



VH-109 Status Lights

Power	Green	On or Powering up
	Off	No Power
System Status	Solid Green	Able to ping the field
	Blinking (1Hz)	Unable to ping the field
	Blinking (20Hz)	Firmware being flashed
	Blinking (50Hz)	Performing “first boot”
	Off	Booting Up
2.4 GHz	Solid Green	Wifi available
	Blinking (20Hz) Sys Blinking	Battery detected, Wireless is disabled
6 GHz	Solid Blue	6 GHz connection made
	Blinking (20Hz) Sys Blinking	Battery detected, Wireless is disabled

37.6.4 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.

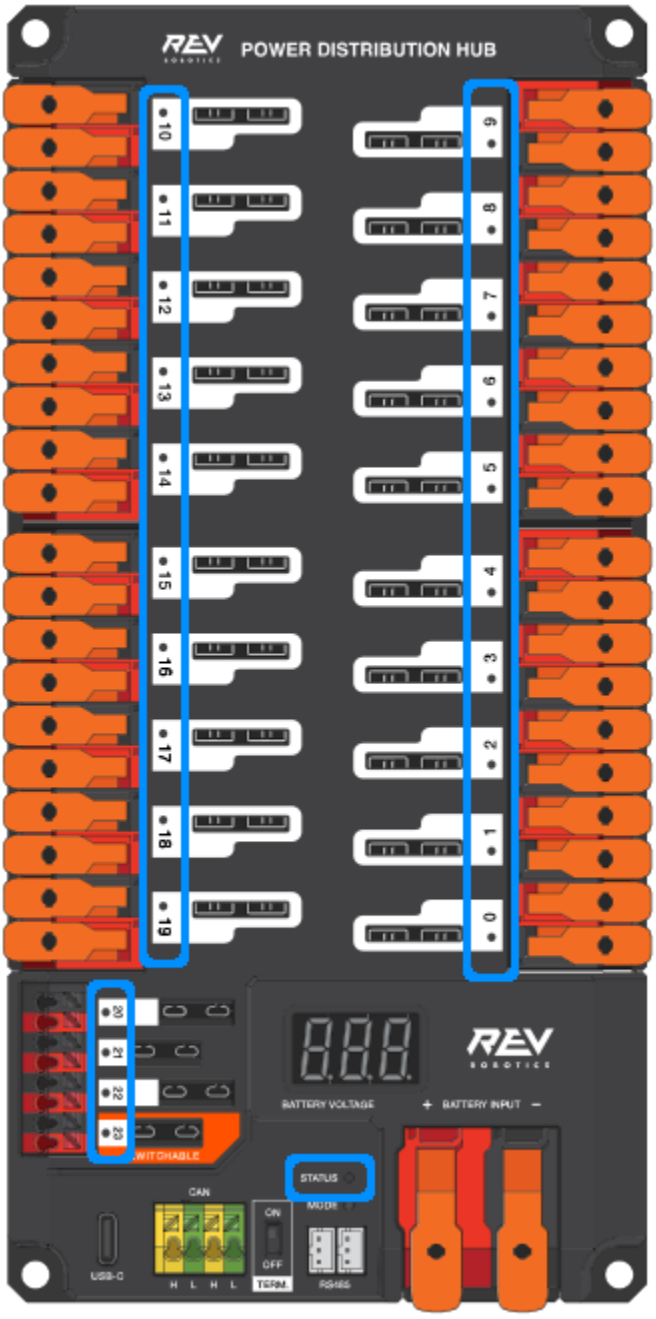
Note

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

37.6.5 Power Distribution Hub



Note

These led patterns only apply to firmware version 21.1.7 and later

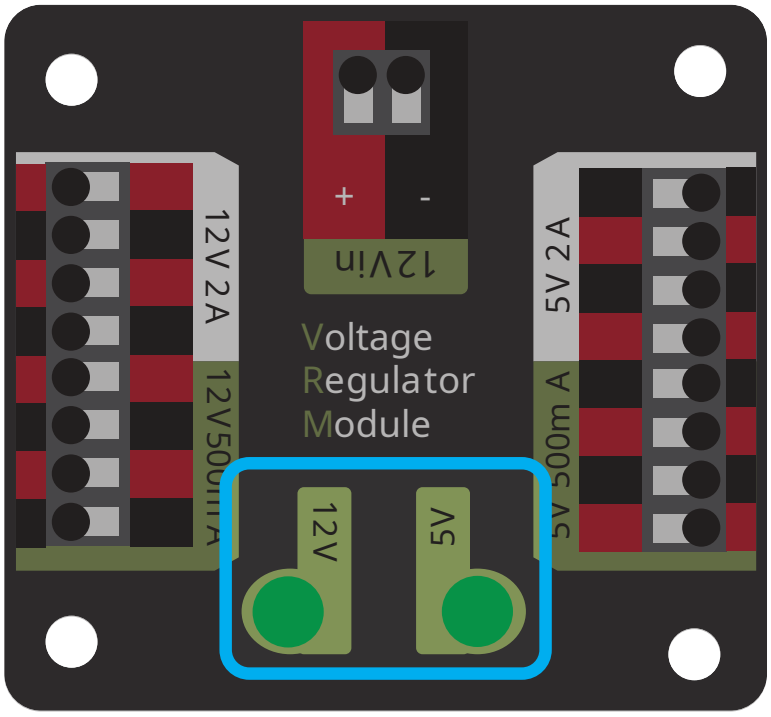
37.6.6 PDH Status LED

LED Color	Status
Blue Solid	Device on but no communication established
Green Solid	Main Communication with roboRIO established
Magenta Blinking	Keep Alive Timeout
Solid Cyan	Secondary Heartbeat (Connected to REV Hardware Client)
Orange/Blue Blinking	Low Battery
Orange/Yellow Blinking	CAN Fault
Orange/Cyan Blinking	Hardware Fault
Orange/Red Blinking	Fail Safe
Orange/Magenta Blinking	Device Over Current

Channel LEDs

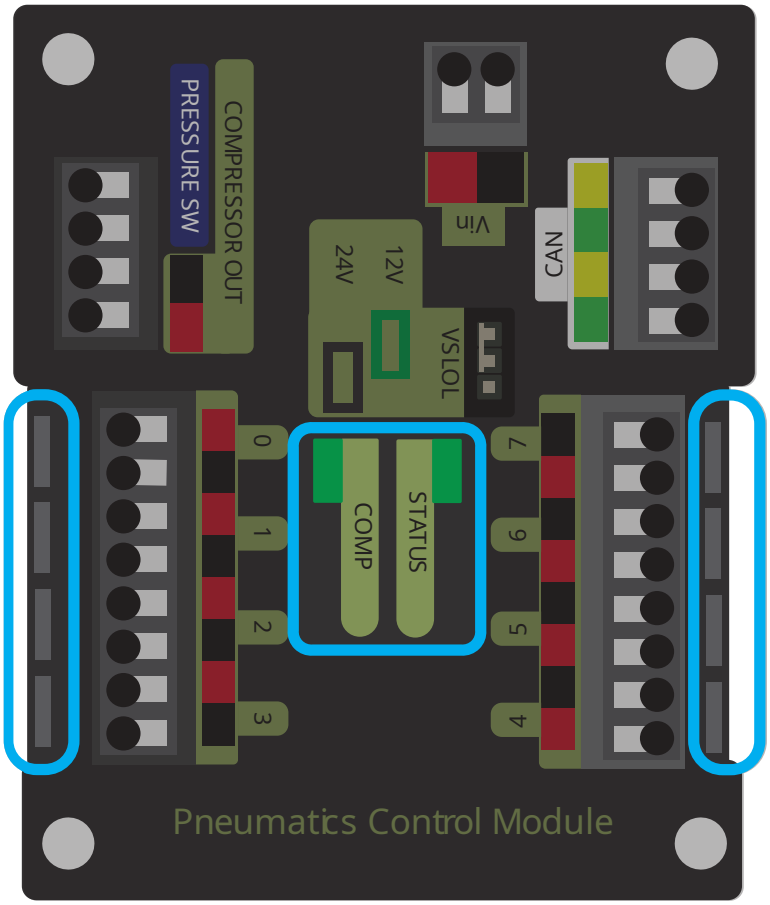
LED Color	Status
Off	Channel has voltage and is operating as expected
Red Solid	Channel has NO voltage and there is an active fault. Check for tripped or missing circuit breaker / fuse
Red Blinking	Sticky fault on the channel. Check for tripped circuit breaker / fuse.

37.6.7 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.

37.6.8 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

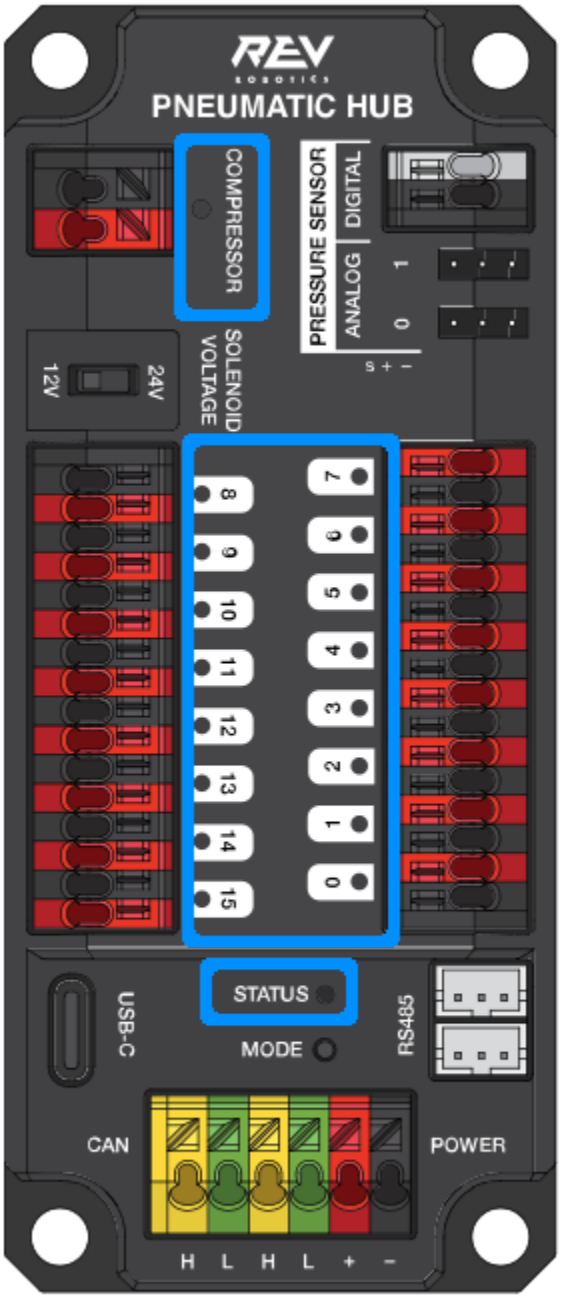
PCM Comp LED

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.

37.6.9 Pneumatic Hub



Note

These led patterns only apply to firmware version 21.1.7 and later

Pneumatic Hub Status LED

LED Color	Status
Blue Solid	Device on but no communication established
Green Solid	Main Communication established
Magenta Blinking	Keep Alive Timeout
Solid Cyan	Secondary Heartbeat (connected to REV HW Client)
Orange/Blue Blinking	Hardware Fault
Orange/Yellow Blinking	CAN Fault
Orange/Red Blinking	Fail Safe
Orange/Magenta Blinking	Device Over Current
Orange/Green Blinking	Compressor Over Current

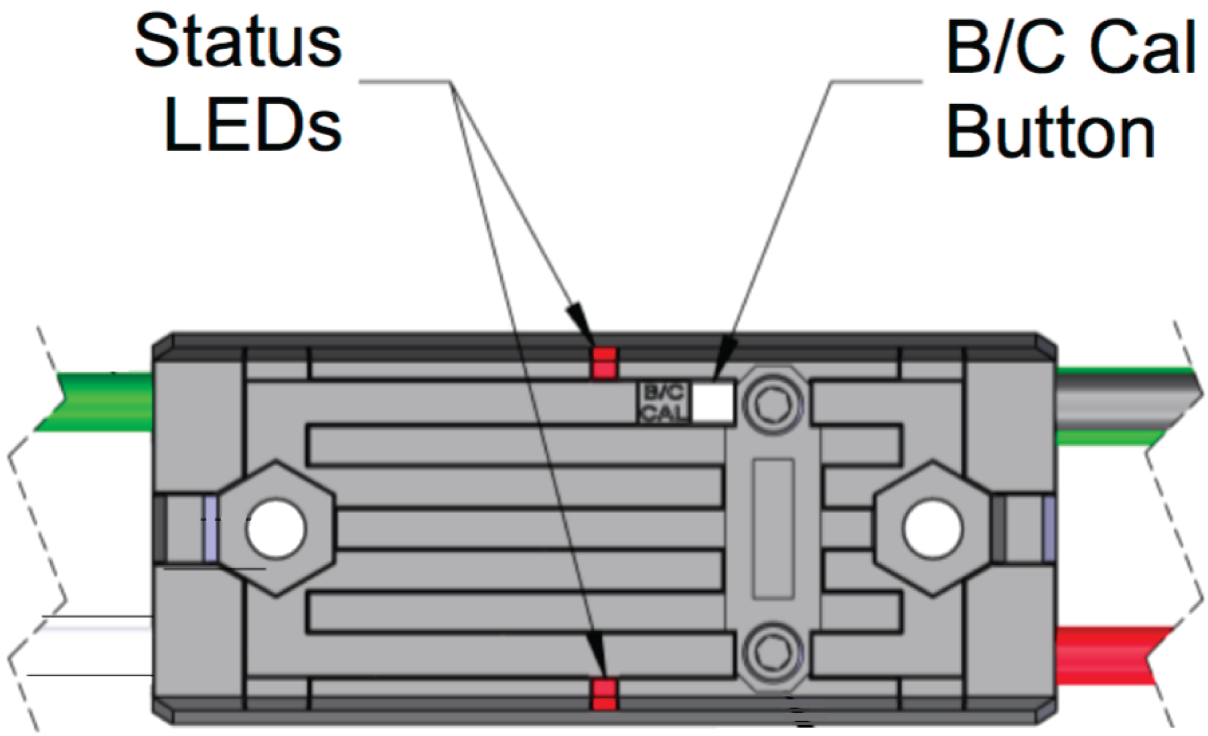
Compressor LED

LED Color	Status
Green Solid	Compressor On
Black Solid	Compressor Off

Solenoid LEDs

LED Color	Status
Green Solid	Solenoid On
Black Solid	Solenoid Off

37.6.10 Talon SRX & Victor SPX Motor Controllers



These motor controllers use Phoenix 5. For the most up-to-date information, see the CTRE Phoenix 5 Documentation.

Talon SRX/Victor SPX Status LEDs During Normal Operation

LEDs	Colors	Device 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
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/Orang	In Boot-loader

Talon SRX/Victor SPX Status LEDs During Calibration

Status LEDs Blink Code	Device State
Flashing Red/Green	Calibration Mode
Blinking Green	Successful Calibration
Blinking Red	Failed Calibration

Talon SRX/Victor SPX B/C CAL Blink Codes

B/C CAL Button Color	Device State
Solid Red	Brake Mode
Off	Coast Mode

37.6.11 Talon FX Motor Controller

The Talon FX uses Phoenix 6. For the most up-to-date information, see the CTRE Phoenix 6 Documentation.

Talon FX Status LEDs

LED State	Meaning
LEDs Off	No Power
Blinking Alternating Red	No valid CAN/PWM signal
Blinking Alternating Orange	CAN detected but no Phoenix running
Blinking Simultaneous Orange	Valid CAN signal, device disabled
Both Solid Orange	Enabled with neutral output
Blinking Simultaneous Red	Driving in reverse
Blinking Simultaneous Green	Driving forward
Offset Alternating Red/Off	Device limit reached
Offset Orange/Off	Thermal cutoff
Alternate Red/Green	Pro-only command while unlicensed
Alternate Red/Orange	Damaged Hardware
Single LED alternates Green/Orange	Device in bootloader

Talon FX B/C CAL Blink Codes

B/C CAL Button Color	Device State
Solid Red	Brake Mode
Off	Coast Mode

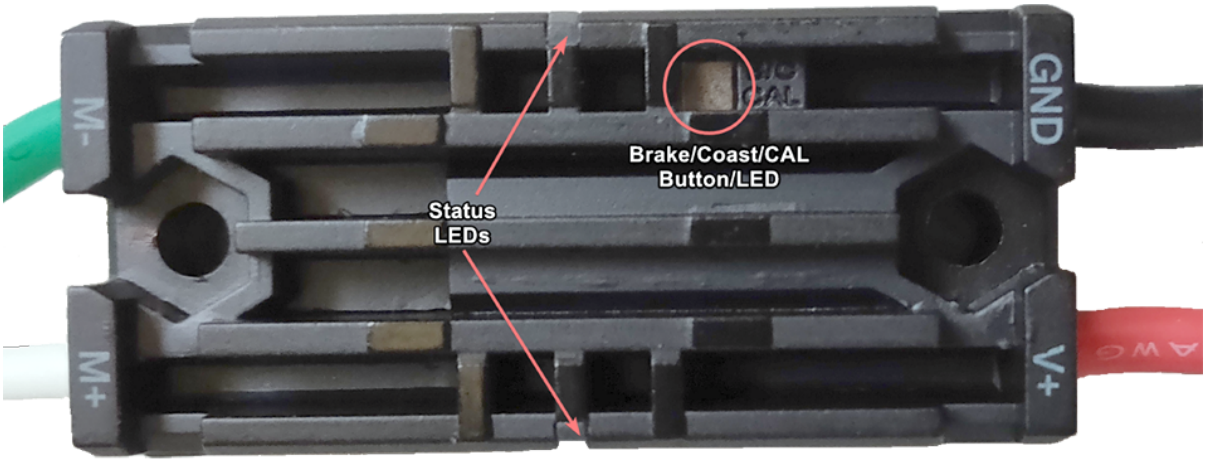
37.6.12 SPARK-MAX Motor Controller

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

37.6.13 REV Robotics SPARK



37.6.14 Victor-SP Motor Controller

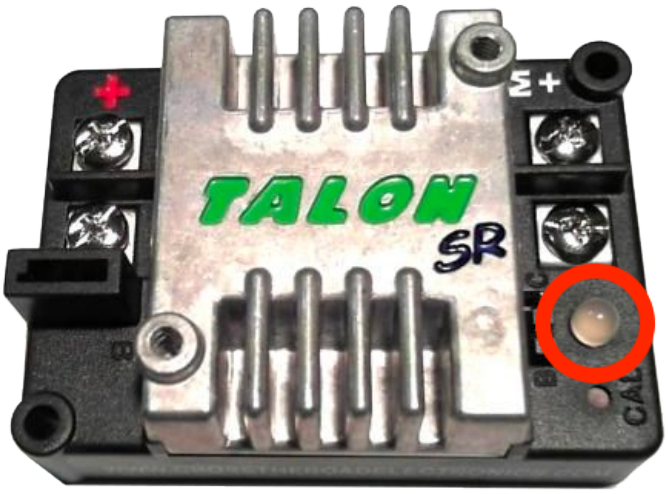


Brake/Coast/Cal Button/LED - Red if the controller is in brake mode, off if the controller is in coast mode

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.

37.6.15 Talon Motor Controller

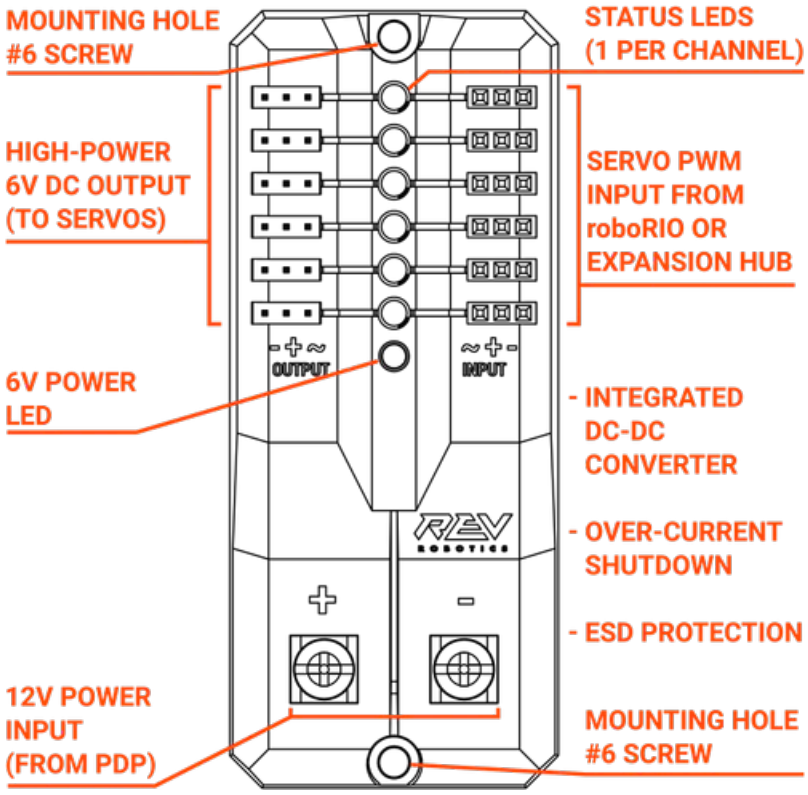


Green	Solid	Full forward output
	Blinking	Proportional to forward output voltage
Red	Solid	Full reverse output
	Blinking	Proportional to reverse output voltage
Orange	Solid	No CAN devices are connected
	Blinking	Disabled state, PWM signal lost, FRC robot disabled, or signal in deadband range (+/- 4% 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.

37.6.16 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 in 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 but 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

37.6.17 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

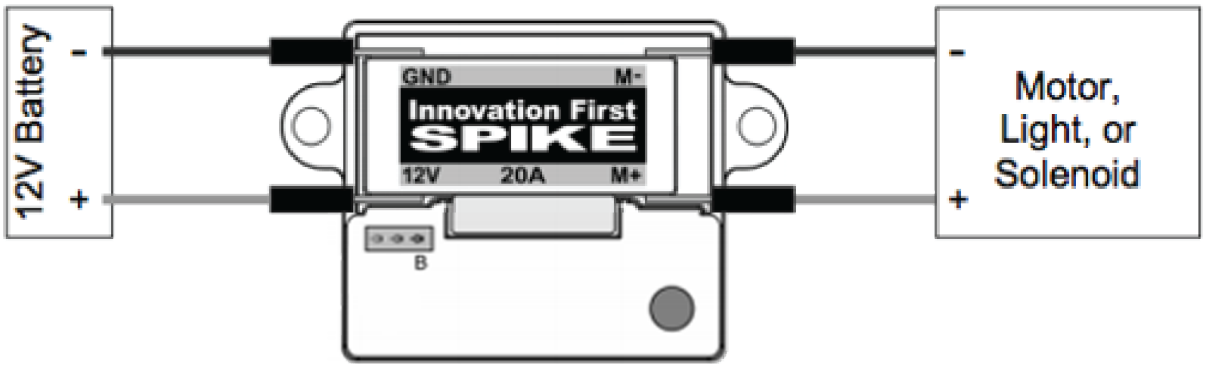
37.6.18 REV Robotics Servo Hub

REV Servo Hub Status LED Patterns

General Status LED

LED Color	Status
Magenta Blinking	Powered on but not connected to a controller or the REV Hardware Client
Blue Solid	Connected to REV Hardware Client
Solid Cyan	Connected to a roboRIO or CAN controller
Green Solid	Connected to a Control Hub or RS-485 controller
Orange/Cyan Blinking	Battery voltage is lower than 5.5V
Orange/Yellow Blinking	CAN fault detected
Orange/Magenta Blinking	Overcurrent fault

37.6.19 Spike relay configured as a motor, light, or solenoid switch

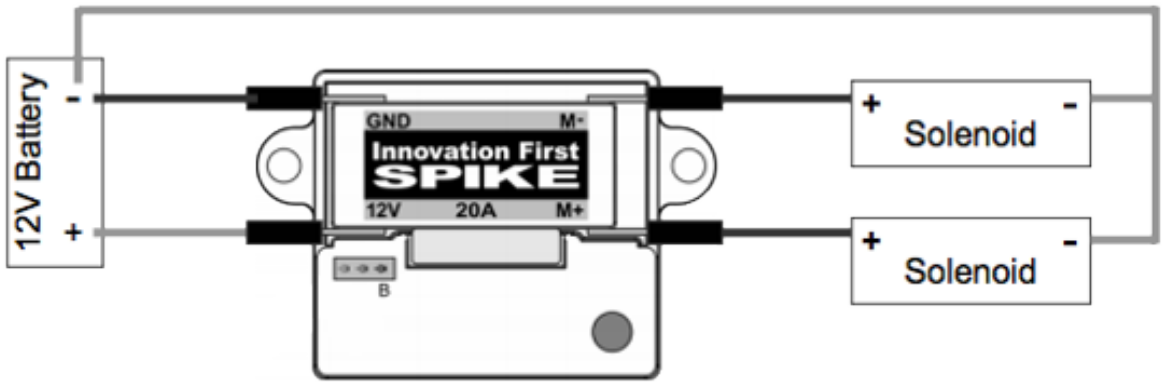


Inputs		Outputs		Indicator	Motor Function
Forward (White)	Reverse (Red)	M+	M-		
Off	Off	GND	GND	Orange	Off/Brake Condition (default)
On	Off	+12v	GND	Green	Motor rotates in one direction
Off	On	GND	+12v	Red	Motor rotates in opposite direction
On	On	+12v	+12v	Off	Off/Brake Condition

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.

37.6.20 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

37.6.21 CANCoder Encoder



LED Color	LED Brightness	CAN Bus detection	Magnet Field Strength	Description
Off	Off			CANCoder is not powered
Yellow/Gree	Bright			Device is in boot-loader. See user manual for more information.
Slow Red Blink	Bright	CAN bus has been lost		
Rapid Red Blink	Dim	CAN bus never detected since boot	Magnet is out of range (<25mT or >135mT)	
Rapid Yellow Blink			Magnet in range with slightly reduced accuracy (25-45mT or 75-135mT)	
Rapid Green Blink			Magnet in range (between 45mT - 75mT)	
Rapid Red Blink	Bright	CAN bus present	Magnet is out of range (<25mT or >135mT)	
Rapid Yellow Blink			Magnet in range with slightly reduced accuracy (25-45mT or 75-135mT)	
Rapid Green Blink			Magnet in range (between 45mT - 75mT)	

37.6.22 CANivore

LED Name	Behavior	Blink Style	Description
STAT	Red	Double-Blink (Note 2)	Device powered through V+/V-, but **USB not plugged in**.
	Red	Fast-Strobe (Note 3)	USB plugged in, but **no USB communication** *(USB not enumerated or USB suspended)*
	Orange	**Double-Blink** if V+/V- is **not** powered (Note 2)	**Good USB** connection, CAN streaming is disabled
	Green	**Fast-Strobe** if V+/V- is powered (Note 3) *TIP: Use this to confirm the V+/V- wiring!*	**Good USB** connection, CAN streaming is enabled
	Green / Orange	LED is never off – one of the two colors is **always illuminated**.	CANivore is in **Bootloader**. Most likely device was unplugged during field-upgrade (Note 4). Use Phoenix Tuner to field upgrade latest CRF firmware file. Alternatively, if CANivore already has application firmware, disconnect all power sources, then re-connect to cold-boot device.
	Orange / Red		CANivore has hardware damage
Wi-Fi	Green	Blink (Note 1)	Wi-Fi is enabled or ESP32 custom application is allowed to use Wi-Fi
	Off		Wi-Fi is disabled
BT (Bluetooth)	Green	Blink (Note 1)	Bluetooth is enabled or ESP32 custom application is allowed to use Bluetooth
	Off		Bluetooth is disabled
CAN (CAN bus)	Solid Red	LED is never off.	**Voltage too low** for CAN bus. CAN communication may **not** be reliable
	Red	**Double-Blink** if termination is disabled (Note 2)	**No CAN** communication
	Orange	**Fast-Strobe** if termination is enabled. (Note 3)	Reserved for CAN 2.0b legacy mode
	Green	*TIP: Use this to confirm termination setting!*	**CAN FD** active

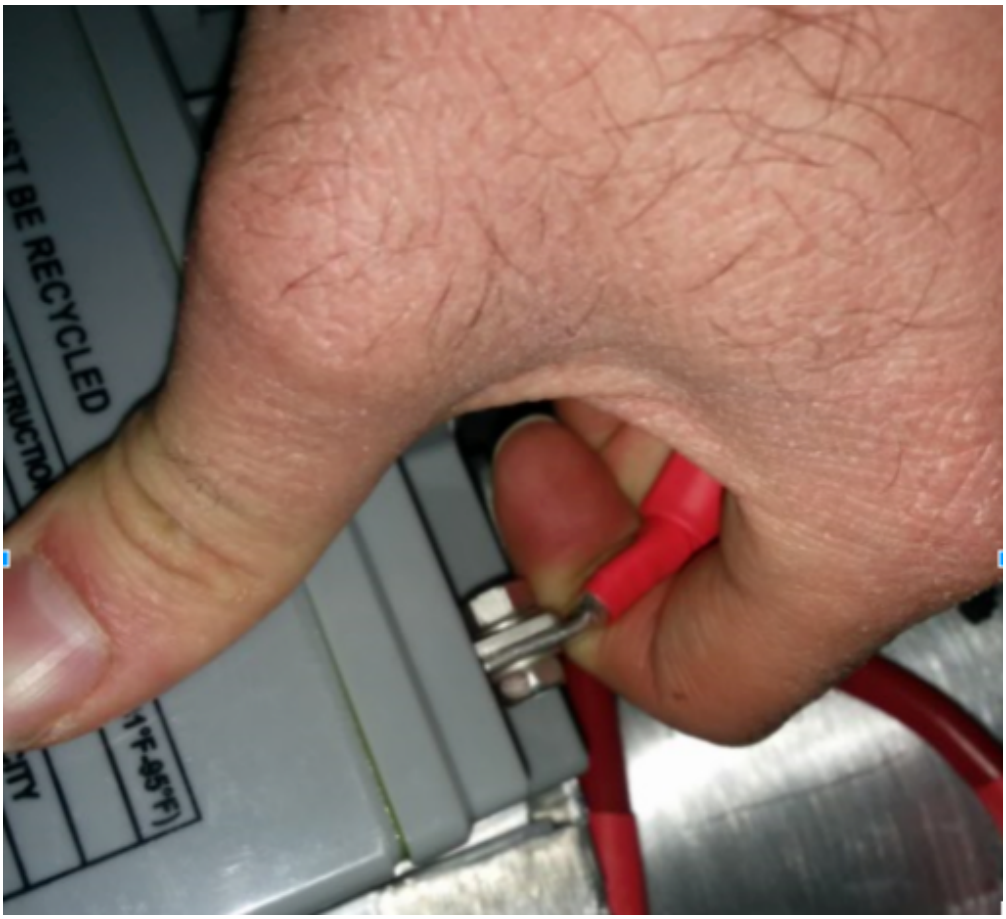
Note 1: "Blink" means LED transitions between on and off slowly at a fixed rate.
Note 2: "Double-Blink" means LED transitions between on and off twice, then pauses.
Note 3: "Fast-Strobe" means LED spends very little time off. It will appear excited or mostly-on.
Note 4: If using USB features, ensure proper signal path between CANivore USB connector and the root USB controller (including all the hubs and data/power connections in between).

37.7 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.

37.7.1 Check Battery Connections

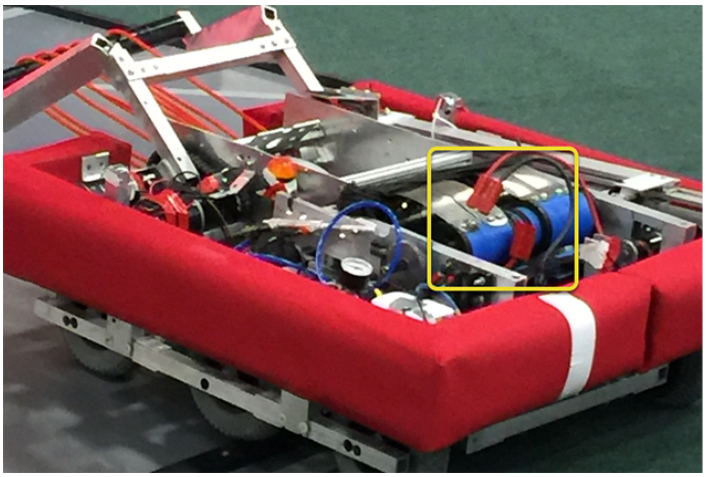


The tape that should be covering the battery connection in these examples has been removed to illustrate what is going on. On your robots, the connections should be covered.

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.

37.7.2 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. Its 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 peace 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.

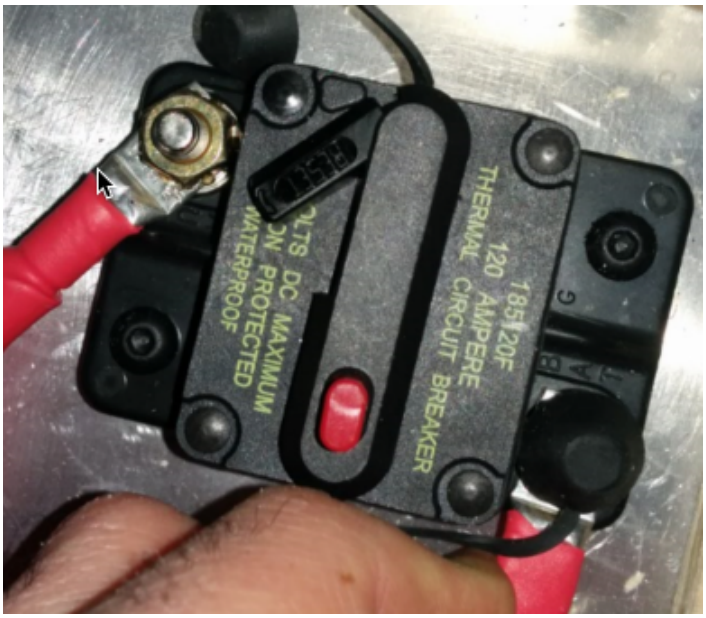
37.7.3 Securing the Battery Connector & Main Power Leads

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.

37.7.4 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.

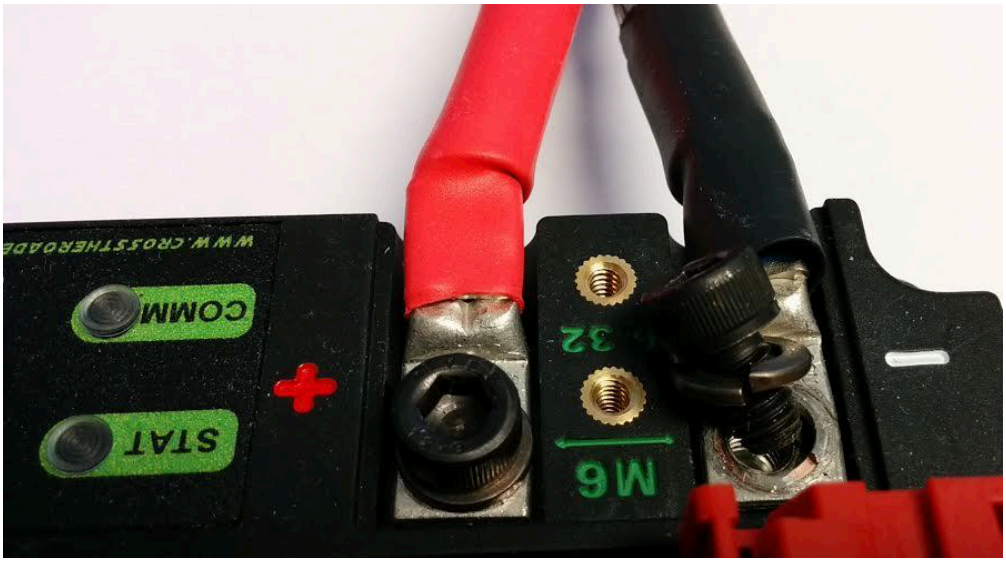
The original nut has a star locking feature, which can wear out over time: these may require checking every few matches, especially if your robot-side battery connector is not attached to the chassis.

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.

37.7.5 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 lose connections.

37.7.6 Tug Testing





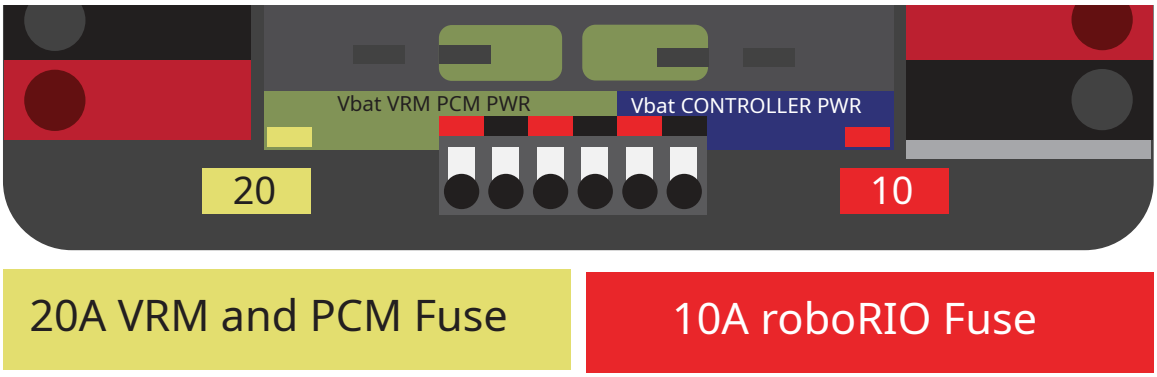
The Weidmuller contacts for power, compressor output, roboRIO power connector, and radio power are important to verify by tugging on the connections as shown. Make sure that none of the connections pull out.

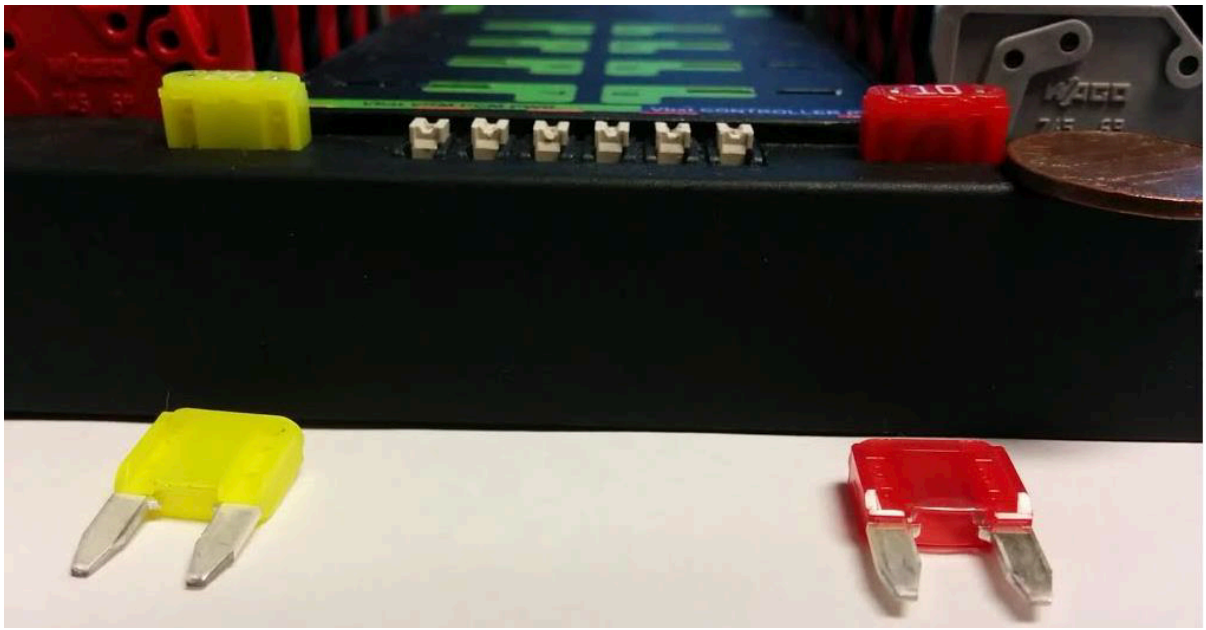
Look for possible or impending shorts with Weidmuller connections that are close to each other, and have too-long wire-lead lengths (wires that are stripped extra long).

Spade connectors can also fail due to improper crimps, so tug-test those as well.

37.7.7 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.

37.7.8 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.

37.7.9 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.

37.7.10 Loose Cables

Cables must be tightened down, particularly the radio power and ethernet cable.

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.

37.7.11 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 makes 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.

37.7.12 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.

37.7.13 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)

37.7.14 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.

37.8 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.

37.8.1 COTS Battery

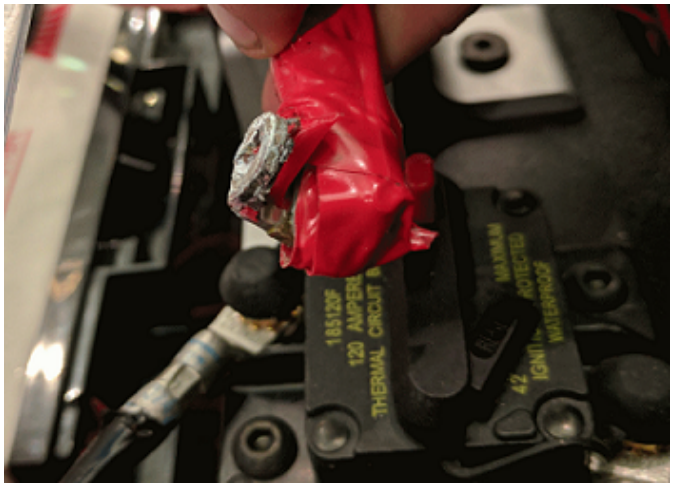
The Robot Rules in the Game Manual specify a COTS non-spillable sealed lead acid battery meeting specific criteria, and gives examples of legal part numbers from a variety of vendors.

37.8.2 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.

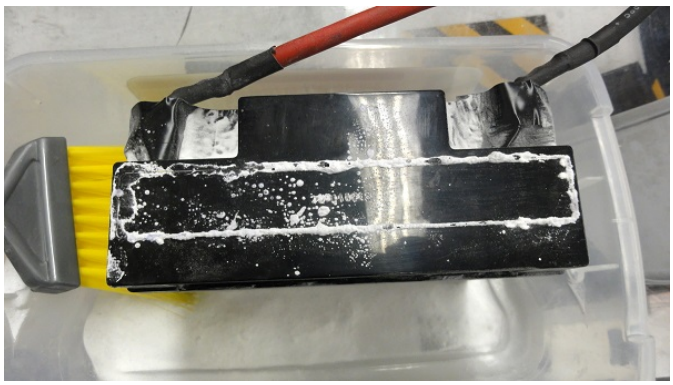
Do *NOT* carry a battery assembly by the cables, and always avoid pulling by them. Pulling on batteries by the cables will begin to damage the lugs, tabs, and the internal connection of the tab. Over time, fatigue damage can add up until the entire tab tears out of the housing! Even if it isn't clearly 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.

37.8.3 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: 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.

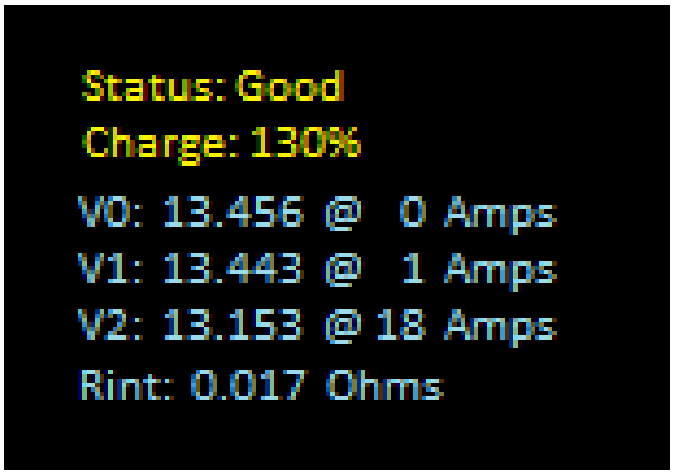
After you finish a driving session, you can *review the battery voltage in the Log Viewer.*

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.



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.

37.8.4 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.

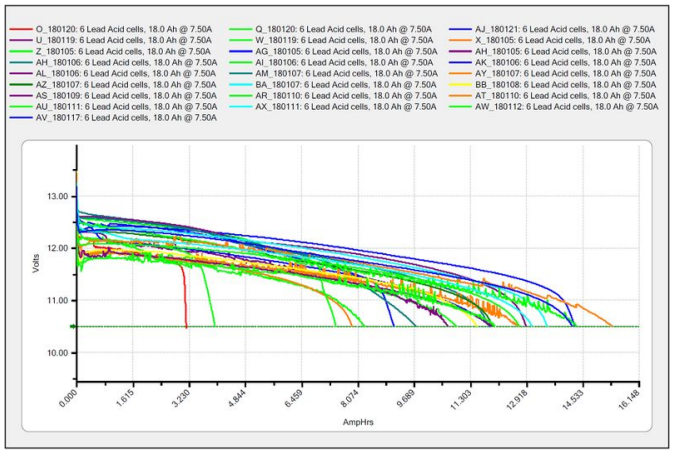
Once a load (like a robot) is connected, and any amount of current is flowing, the battery voltage will drop. So if you check a battery with a Voltmeter, and it reads 13.2, and then connect it to your robot and power on, it will read lower, maybe 12.9 on the Driver Station display. Those numbers will 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.

37.8.5 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 removable (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.



38 Hardware Tutorials

Note

The pages in this section of the documentation contain media which is only viewable from the web version of the documentation

38.1 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.

38.2 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.

38.3 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.

38.4 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 maximum performance from your robot design.

39 Sensors

39.1 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.

39.1.1 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.

39.2 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 resolution[1].

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.

39.2.1 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.

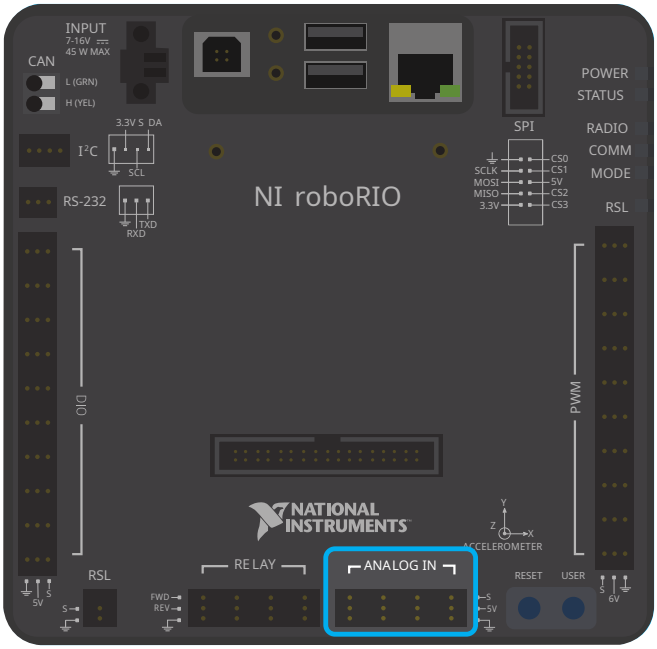
Warning

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.



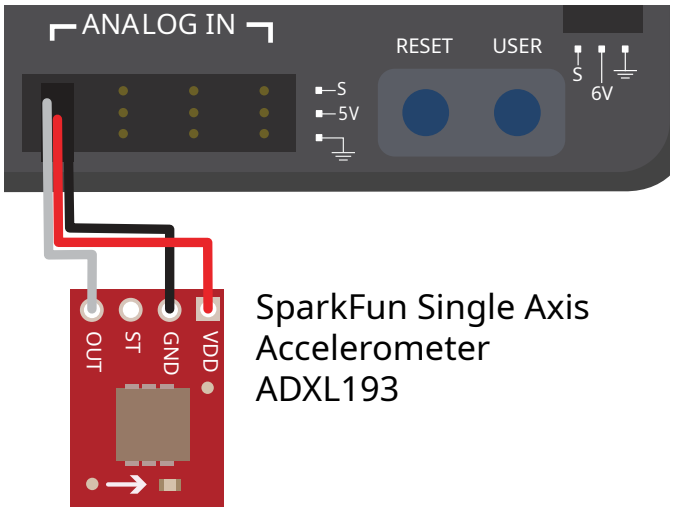
The roboRIO has 4 built-in analog input ports (numbered 0-3), as seen in the image above. Each port has three pins - signal (“S”), power (“V”), and ground (”⏚”). The “power” and “ground” pins are used to power the peripheral sensors that connect to the analog input ports - there is a constant 5V potential difference between the “power” and the “ground” pins[2]. The signal pin is the pin on which the signal is actually measured.

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.

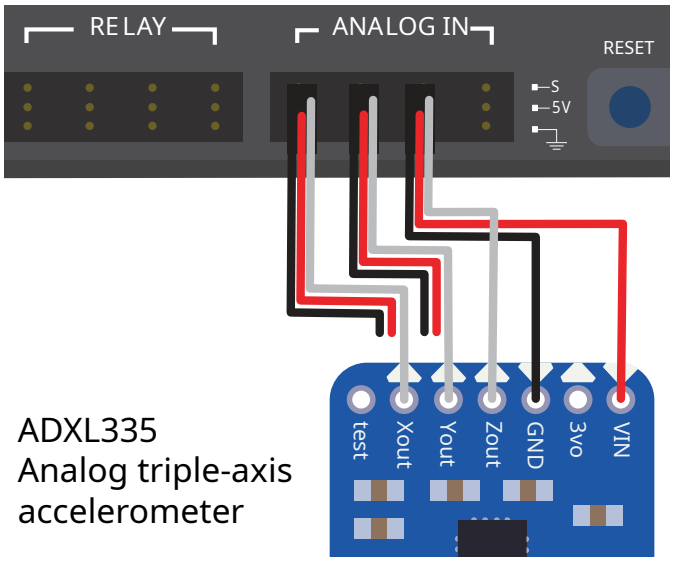
[2] All power pins are actually connected to a single rail, as are all ground pins - there is no need to use the power/ground pins corresponding to a given signal pin.



SparkFun Single Axis Accelerometer ADXL193

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 is shows an analog accelerometer that requires three analog input ports, but similar wiring can be used for analog sensors requiring two analog input ports.



ADXL335 Analog triple-axis accelerometer

39.2.2 Footnotes

39.3 Analog Potentiometers - Hardware

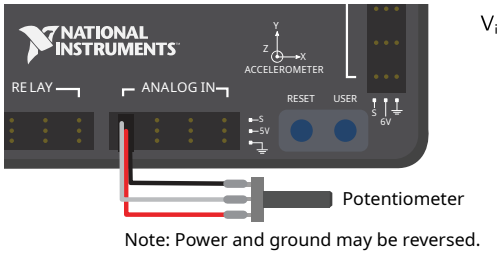
> **Note**
>
> This section covers analog potentiometer hardware. For a software guide to analog potentiometers, see *Analog Potentiometers - Software*.

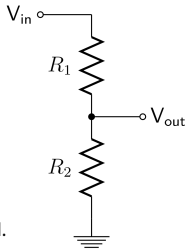
> **Warning**
>
> Potentiometers generally have a mechanically-limited travel range. Users should be careful that their mechanisms do not turn their potentiometers past their maximum travel, as this will damage or destroy the potentiometer.

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.

39.3.1 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 R1 and R2 change; however, their sum remains constant[1]. 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.

[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.

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.

39.3.2 Footnotes

39.4 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 0V[1][2].

39.4.1 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

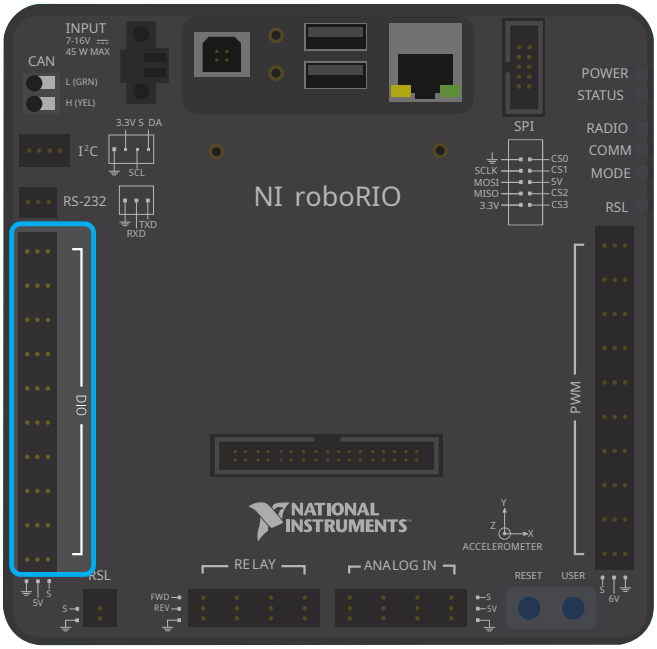
Always consult the technical specifications of the sensor you are using *before* wiring the sensor, to ensure that the correct wire is being connected to each pin. Failure to do so can result in damage to the device.

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.

[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, the use of this is far less common than the 5V.



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” pins[3] - 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).

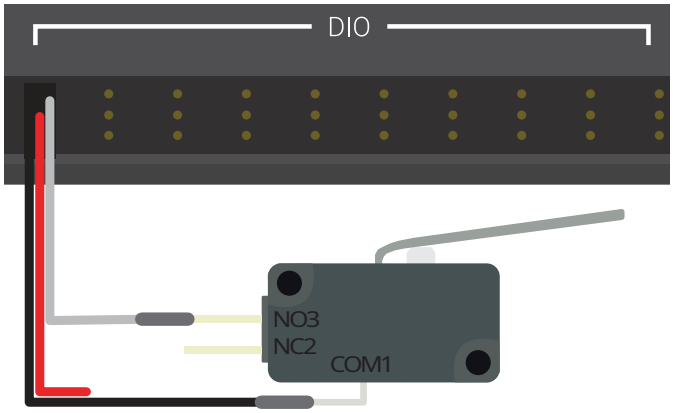
All DIO ports have built-in “pull-up” resistors between the power pins and the signal pins - these ensure that when the signal pin is “floating” (i.e. is not connected to any circuit), they consistently remain in a “high” state.

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.”

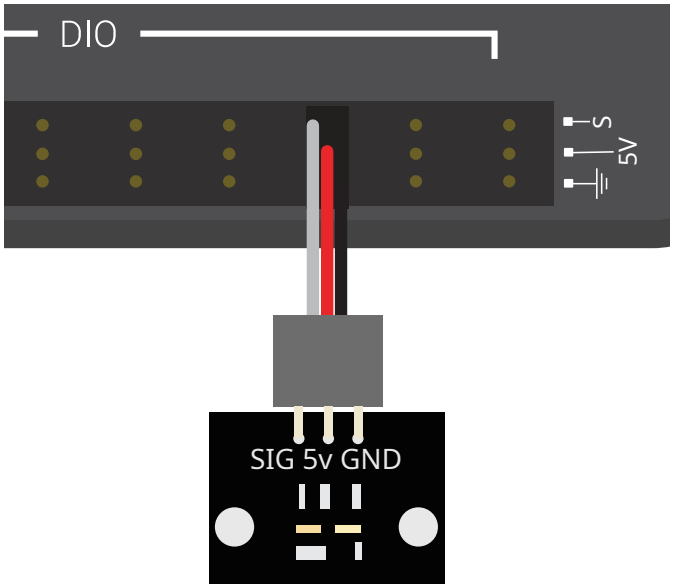
[3] All power pins are actually connected to a single rail, as are all ground pins - there is no need to use the power/ground pins corresponding to a given signal pin.



Limit Switch or Micro Switch

Connecting a powered sensor to a DIO port

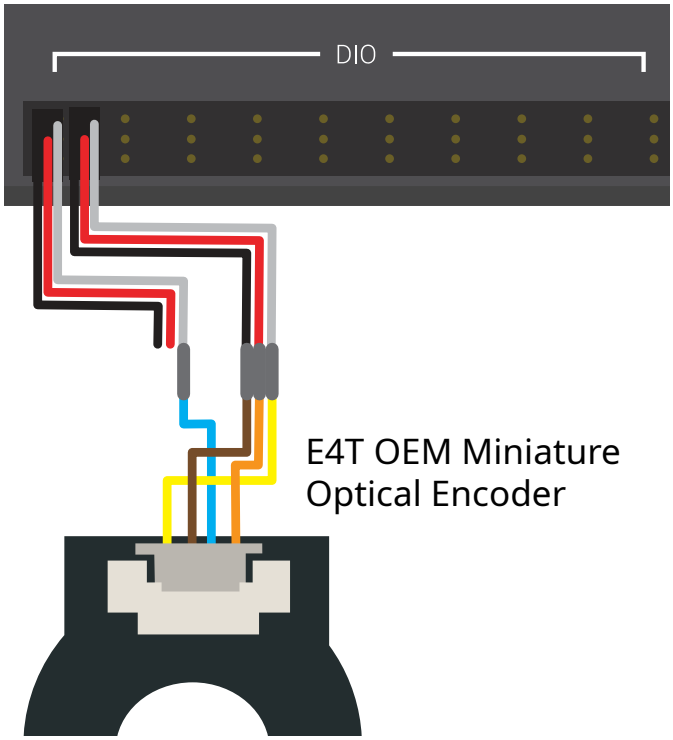
Many digital sensors (such as most no-contact proximity switches) require power in order to work. A powered sensor will generally have three wires - signal, power, and ground. These should be connected to the corresponding pins of the DIO port.



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.



39.4.2 Footnotes

39.5 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."

39.5.1 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.

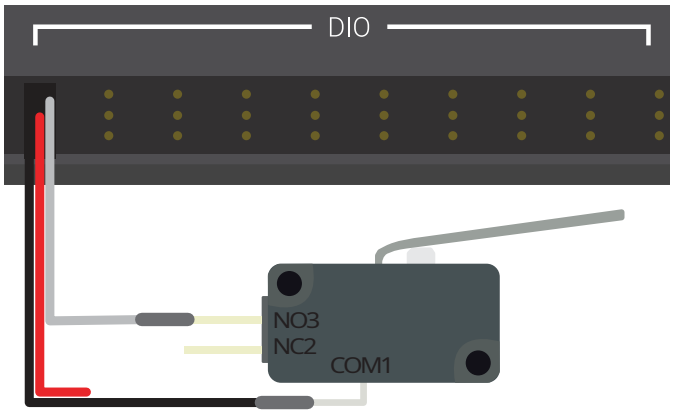
The digital inputs on the roboRIO have pull-up resistors that will make the input be high (1 value) when the switch is open, but when the switch closes the value goes to 0 since the input is now connected to ground.

39.5.2 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

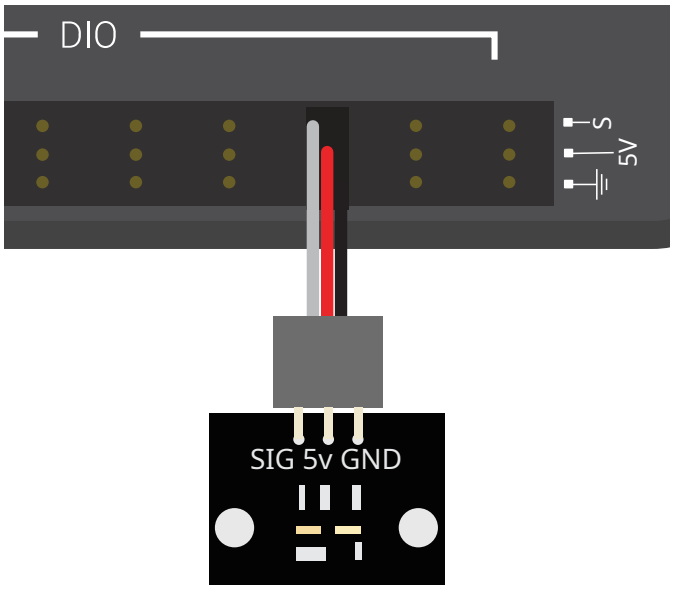
Mechanical proximity switches (more commonly known as "limit switches") are probably the most commonly used proximity switch in FRC, due to their simplicity, ease-of-use, and low cost. A limit switch is quite simply a switch attached to a mechanical arm, usually at the limits

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.

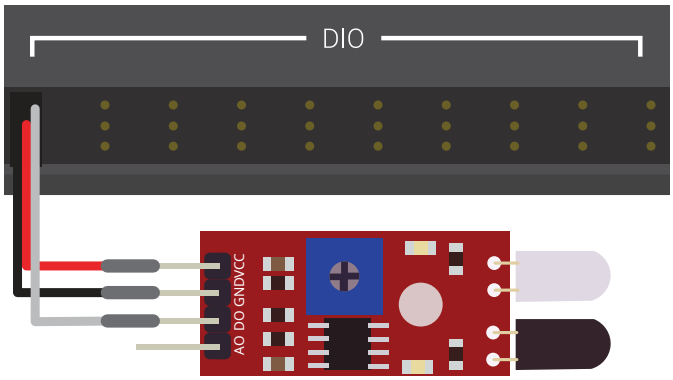
Inductive Proximity Switches



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



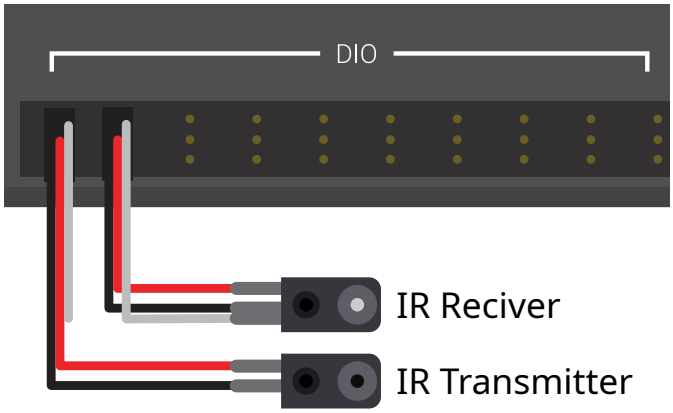
IR Digital Obstacle Avoidance Sensor

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 above.

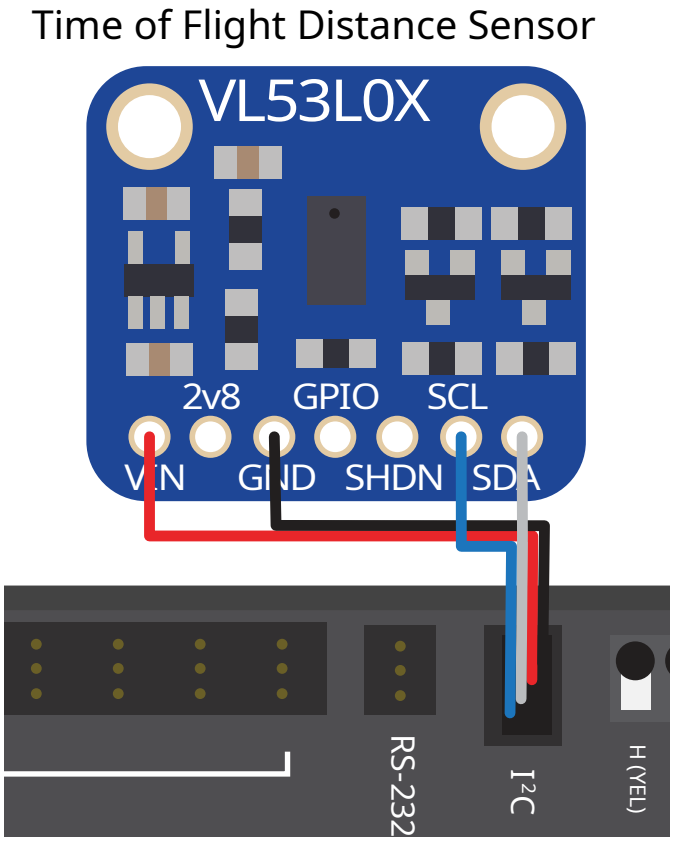
Since photoelectric proximity switches rely on measuring the amount of reflected light, they are often inconsistent in their triggering range between different materials - accordingly, most

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 below is a beam break sensor with an IR LED transmitter and IR receiver.



Time-of-flight Proximity Switches



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 versions 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.

39.6 Encoders - Hardware

Note

This section covers encoder hardware. For a software guide to encoders, see *Encoders - Software*.

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*).

39.6.1 Types of Encoders

There are three main ways encoders connect physically that are 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.

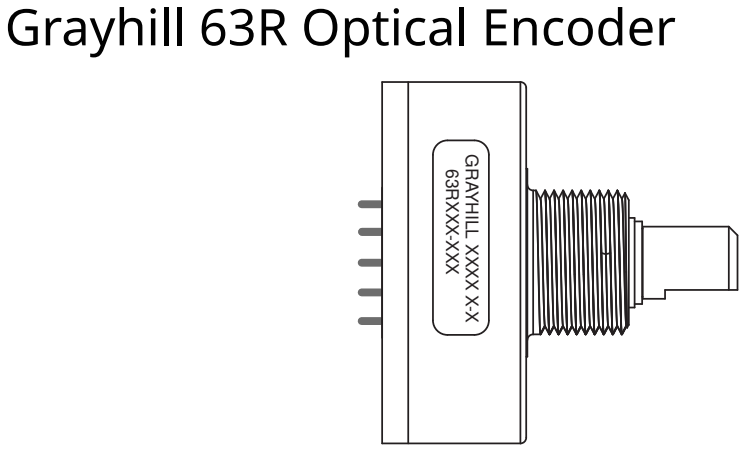
There are also three main ways the encoder data is communicated that are typically used in FRC:

- *Quadrature encoders*
- *Duty Cycle encoders*
- *Analog encoders*

Note

Some encoders may support more then one communication method

Shafted Encoders



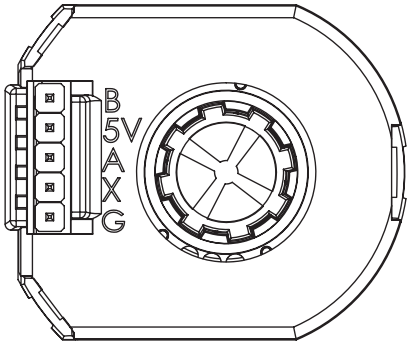
Shafted encoders have a sealed body with a shaft protruding out of it that must be coupled rotationally to a mechanism. This is often done with a helical beam coupling, or, more cheaply, with a length of flexible tubing (such as surgical tubing or pneumatic tubing), fastened with cable ties and/or adhesive at either end. Many commercial off-the-shelf FRC gearboxes have purpose-built mounting points for shafted encoders.

Examples of shafted encoders:

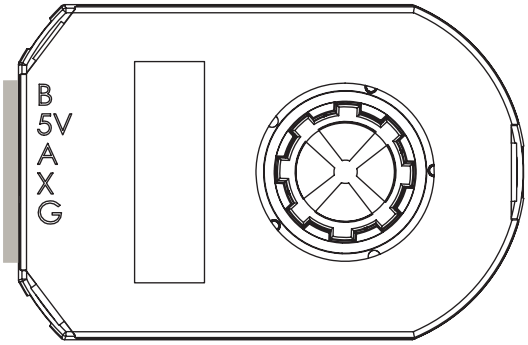
- Grayhill 63r
- US Digital MA3

On-shaft Encoders

AM10 Series Modular Incremental Encoder



AMT103



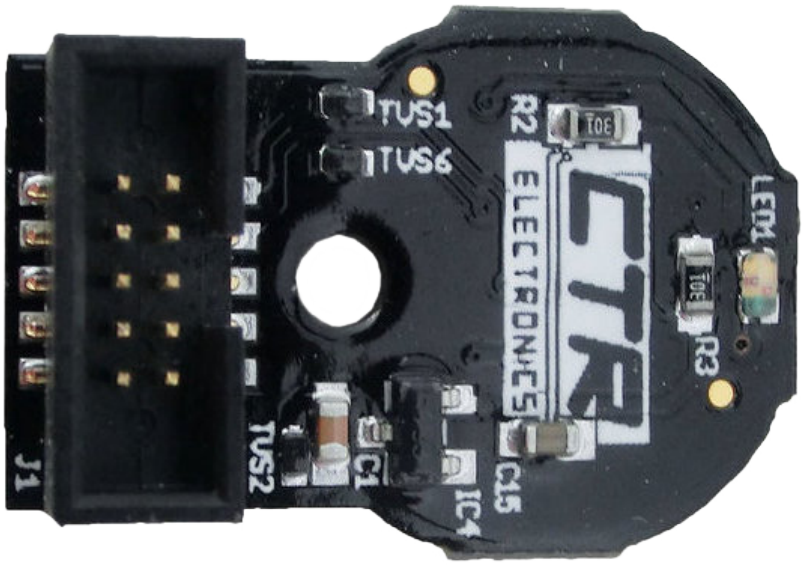
AMT102

On-shaft encoders couple to a shaft by fitting *around* it, forming a friction coupling between the shaft and a rotating hub inside the encoder.

Examples of On-shaft encoders:

- AMT103-V available through FIRST Choice
- CIMcoder
- REV Through Bore Encoder
- US Digital E4T

Magnetic Encoders



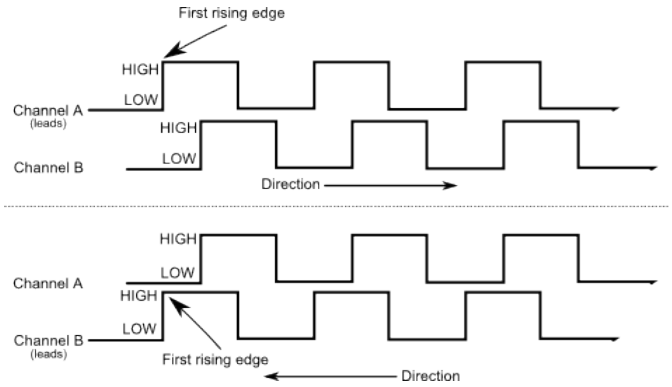
Magnetic encoders require no mechanical coupling to the shaft at all; rather, they track the orientation of a magnet fixed to the shaft. While the no-contact nature of magnetic encoders can be handy, they often require precise construction in order to ensure that the magnet is positioned correctly with respect to the encoder.

Examples of magnetic encoders:

- CTRE Mag Encoder
- Thrifty Absolute Magnetic Encoder
- Team 221 Lamprey2

Quadrature Encoders

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:



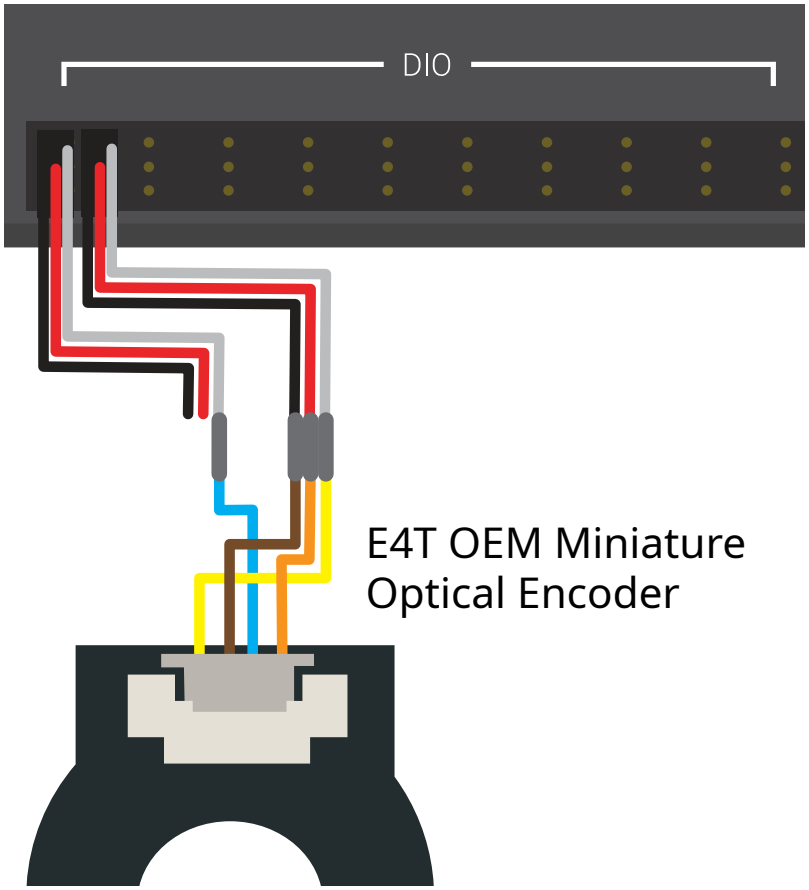
Thus, across both channels, there are four total “edges” per period (hence “quad”). The use of two out-of-phase pulses allows the direction of motion to be unambiguously determined from which pulse “leads” the other.

As each square wave pulse is a digital signal, quadrature encoders connect to the *digital input* ports on the roboRIO.

Examples of quadrature encoders:

- AMT103-V available through FIRST Choice
- CIMcoder
- CTRE Mag Encoder
- Grayhill 63r
- REV Through Bore Encoder
- US Digital E4T

Quadrature Encoder Wiring



Quadrature Encoders, such as the E4T OEM Miniature Optical Encoder, can be wired to two digital input ports as shown above.

Index

Some quadrature encoders have a third index pin which pulses when the encoder completes a revolution.

Quaderature 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.

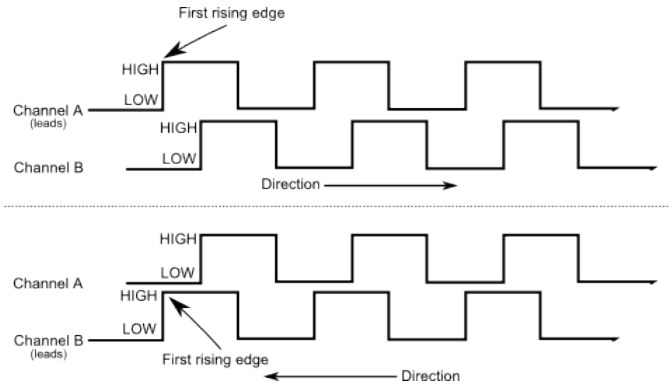
Edges per revolution refers to the total number of transitions from high-to-low or low-to-high across both channels per revolution of the encoder shaft. A full period contains *four* edges.

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.

Duty Cycle Encoders

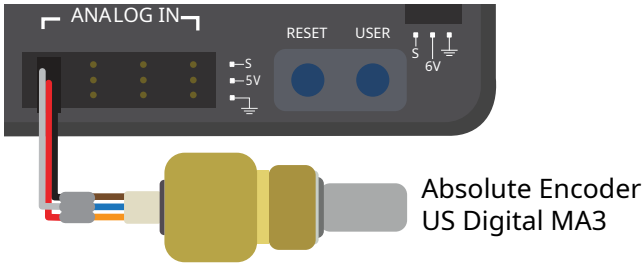


Duty cycle encoders connect to a single digital input on the roboRIO. They output a pulse where the length of a pulse is proportional to the absolute position of the encoder.

Examples of duty cycle encoders:

- AndyMark Mag Encoder
- CTRE Mag Encoder
- REV Through Bore Encoder
- Team 221 Lamprey2
- US Digital MA3

Analog Encoders



Analog encoders connect to a analog input on the roboRIO. They output a voltage proportional to the absolute position of the encoder.

Examples of analog encoders:

- Team 221 Lamprey2
- Thrifty Absolute Magnetic Encoder
- US Digital MA3

39.7 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.

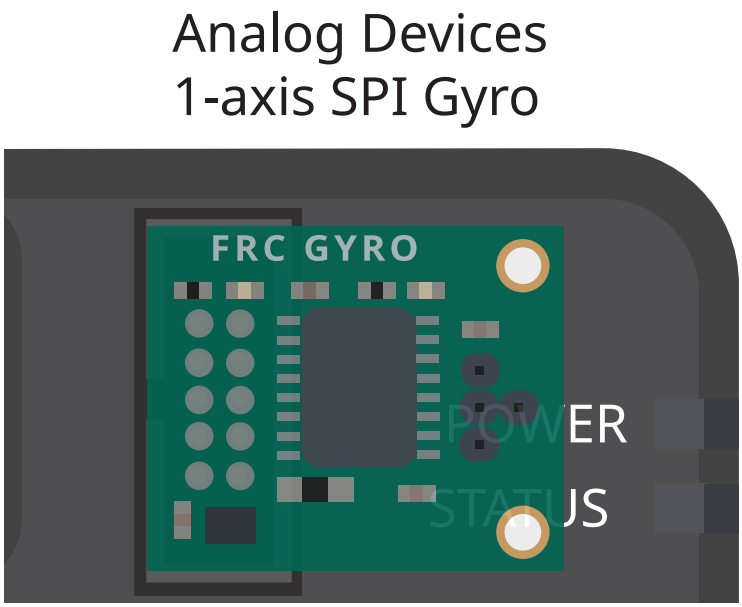
Several popular FRC® devices known as *IMUs* (Inertial Measurement Units) combine 3-axis gyros, accelerometers, and other position sensors into one device. Some popular examples are:

- Analog Devices ADIS16448 and ADIS 16470 IMUs
- CTRE Pigeon IMU
- Kauai Labs NavX

39.7.1 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



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 the ADXRS450 pictured above, use the *SPI port* on the roboRIO instead.

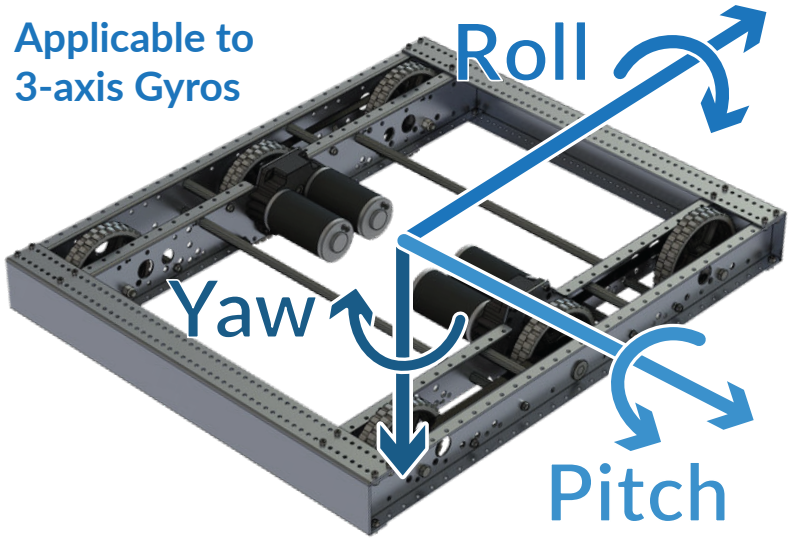
The Analog Devices ADXRS450 FRC Gyro Board that has been in FIRST Choice in recent years is a commonly used single axis gyro.

Three-axis Gyros



Three-axis gyros measure rotation rate around all three spatial axes (typically labeled x, y, and z). The motion around these axis is called pitch, yaw, and roll.

The Analog Devices ADIS16470 IMU Board for FIRST Robotics that has been in FIRST Choice in recent years is a commonly used three-axis gyro.



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*.

39.8 Ultrasonics - Hardware

Note

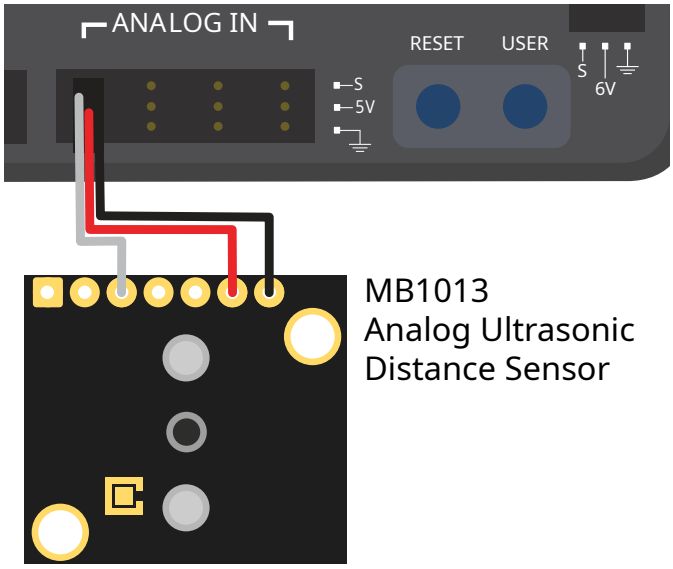
This section covers ultrasonic sensor hardware. For a software guide to ultrasonics, see *Ultrasonics - Software*.

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.

39.8.1 Types of ultrasonics

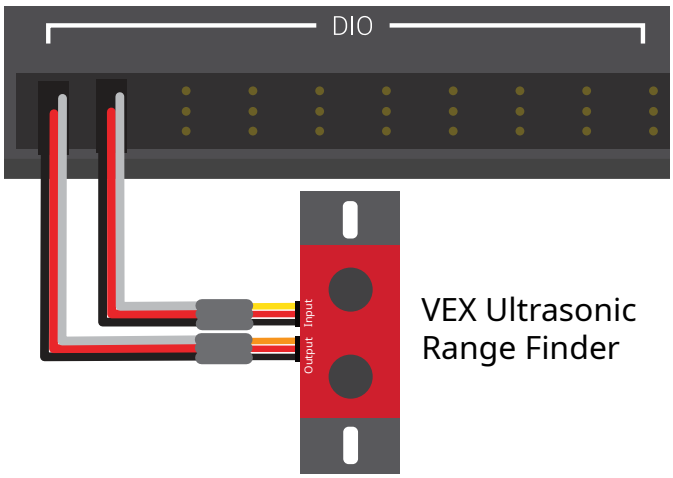
While all ultrasonic rangefinders operate on the "ping-response" principle outlined above, they may vary in the way they communicate with the roboRIO.

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



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.

39.8.2 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.

39.9 Accelerometers - Hardware

Accelerometers are common sensors used to measure acceleration.

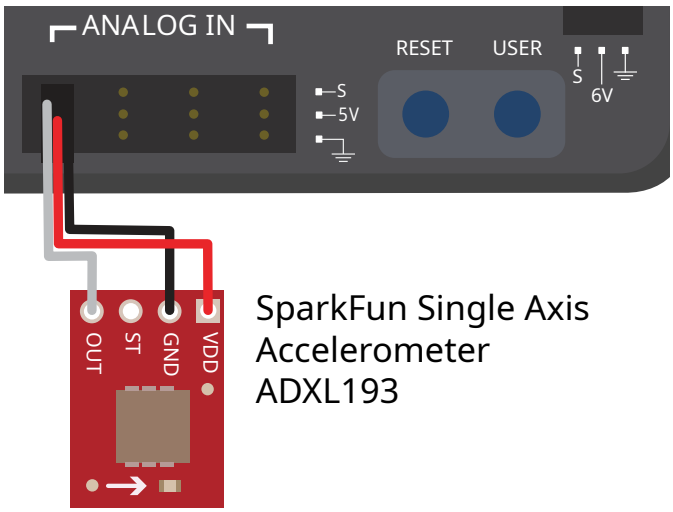
In principle, precise measurements of acceleration can be double-integrated and used to track position (similarly to how the measurement of turn rate from a gyroscope can be integrated to determine heading) - however, in practice, accelerometers that are available within the legal FRC® price range are 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.

39.9.1 Types of accelerometers

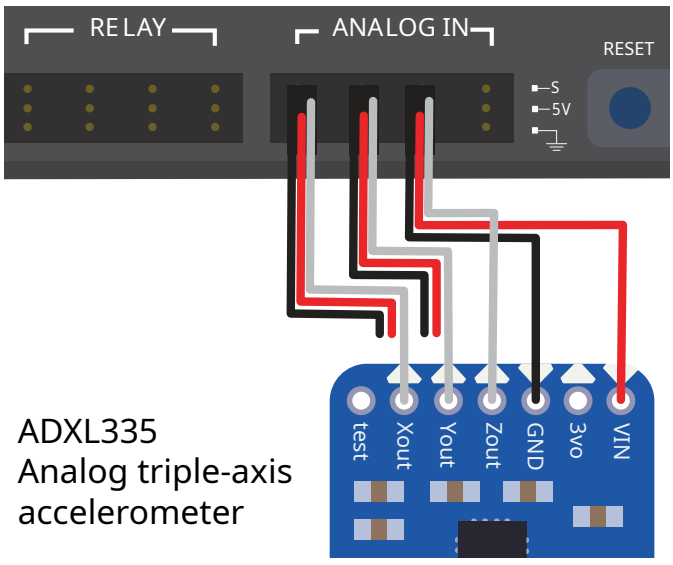
There are three types of accelerometers commonly-used in FRC: single-axis accelerometers, multi-axis accelerometers, and IMUs.

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.

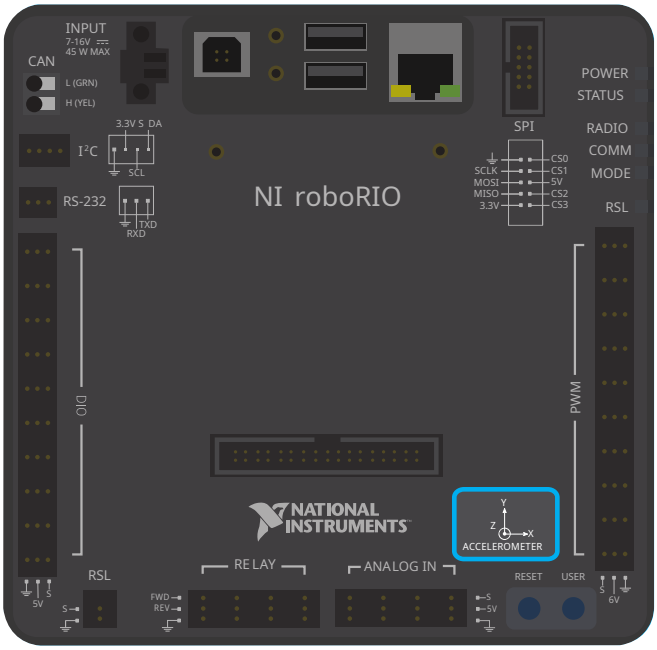
Multi-axis accelerometers



Multi-axis accelerometers measure acceleration along multiple spatial axes. The roboRIO's built-in accelerometer is a three-axis accelerometer.

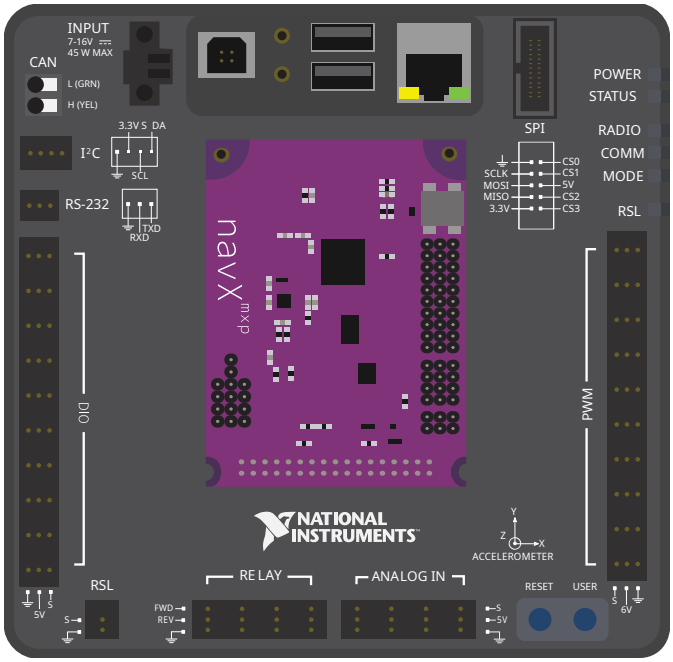
Peripheral multi-axis accelerometers may simply output multiple analog voltages (and thus connect to the *analog input ports*, or (more commonly) they may communicate with one of the roboRIO's *serial buses*.

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 example include:

- Analog Devices ADIS16448 and ADIS 16470 IMUs
- CTRE Pigeon IMU
- Kauai Labs NavX

39.10 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.

39.10.1 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*.

39.10.2 Caveats

LIDAR sensors do suffer from a few common drawbacks:

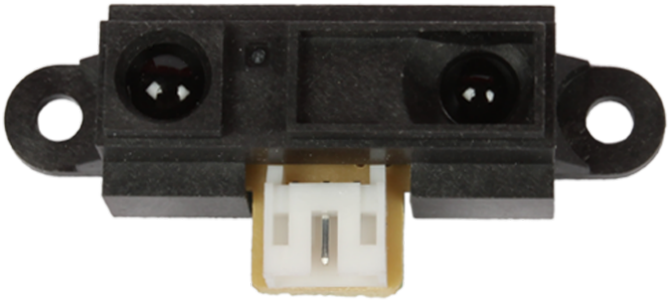
Like ultrasonics, LIDAR relies on the reflection of the emitted pulse back to the sensor. Thus, LIDAR critically depends on the reflectivity of the material in the wavelength of the laser. The FRC field wall is made of polycarbonate, which tends to be transparent in the infrared wavelength (which is 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.

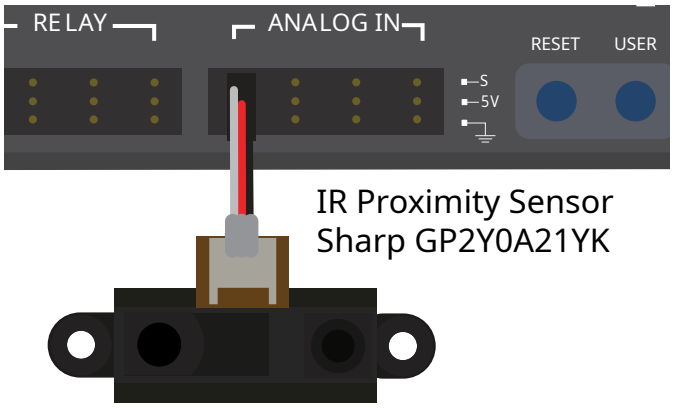
39.11 Triangulating Rangefinders



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.

39.11.1 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.

39.11.2 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).

39.12 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 sent 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.

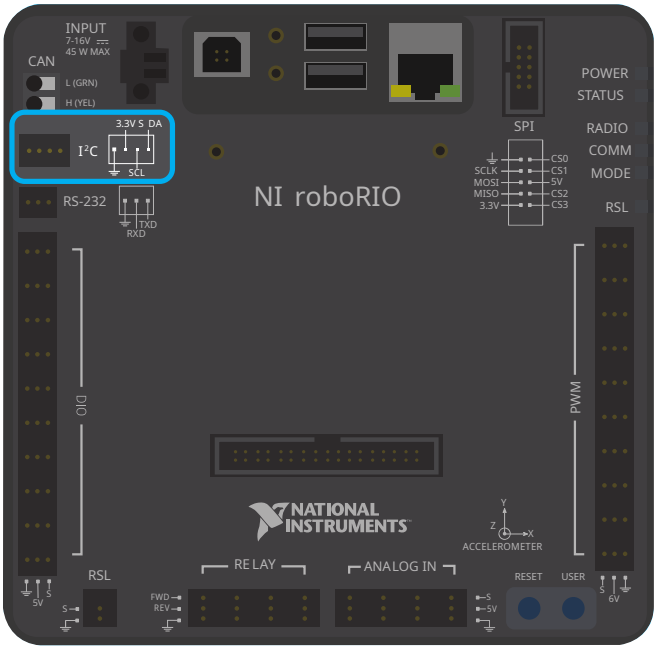
39.12.1 Types of supported serial buses

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).

39.12.2 I2C



I²C Port

Figure 6 and Table 5 describe the I²C port pins and signals.

Figure 6. I²C Port Pinout

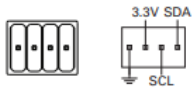


Table 5. I²C Port Signal Descriptions

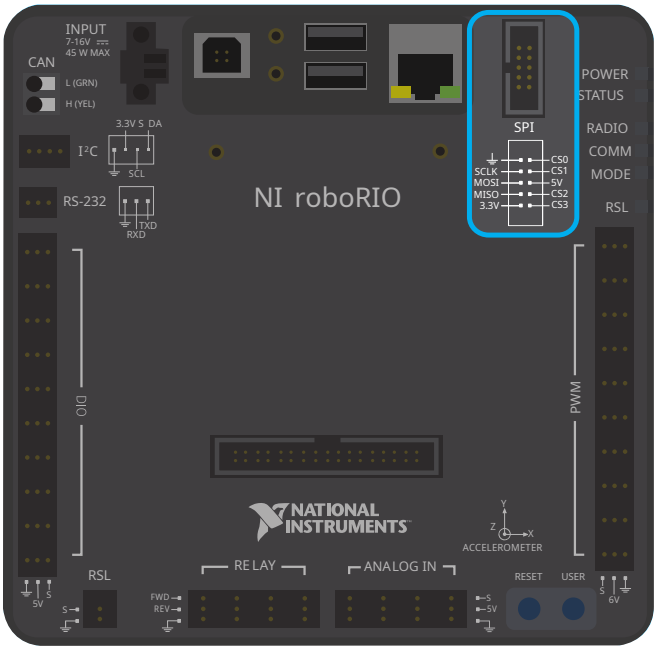
Signal Name	Direction	Description
GND	—	Reference for digital lines and +3.3 V power output.
3.3V	Output	+3.3 V power output.
SCL	Input or Output	I²C lines with 3.3 V output, 3.3 V/ 5 V-compatible input. Refer to the *I²C Lines* section for more information.
SDA	Input or Output	

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.

> **Warning**
>
> Be sure to familiarize yourself on the following known issue before using the onboard I2C port: *Onboard I2C Causing System Lockups*

39.12.3 SPI



SPI Port

Figure 13 and Table 12 describe the SPI port pins and signals.

Figure 13. SPI Port Pinout

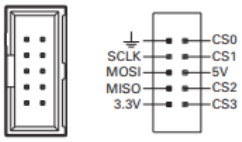


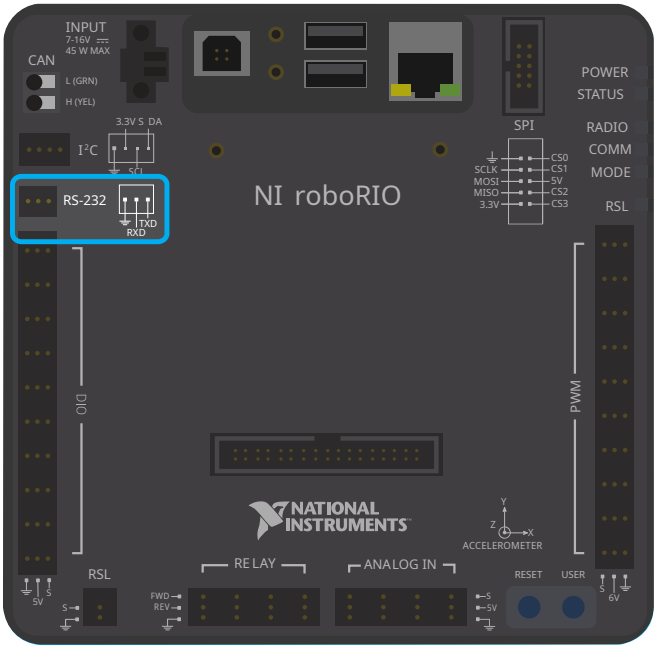
Table 12. SPI Port Signal Descriptions

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.

39.12.4 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

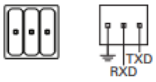


Table 6. RS-232 Serial Port Signal Descriptions

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 (+/- 15v). The MXP serial port uses CMOS signaling levels with a 3.3v output, and a 3.3v to 5v tolerant input.

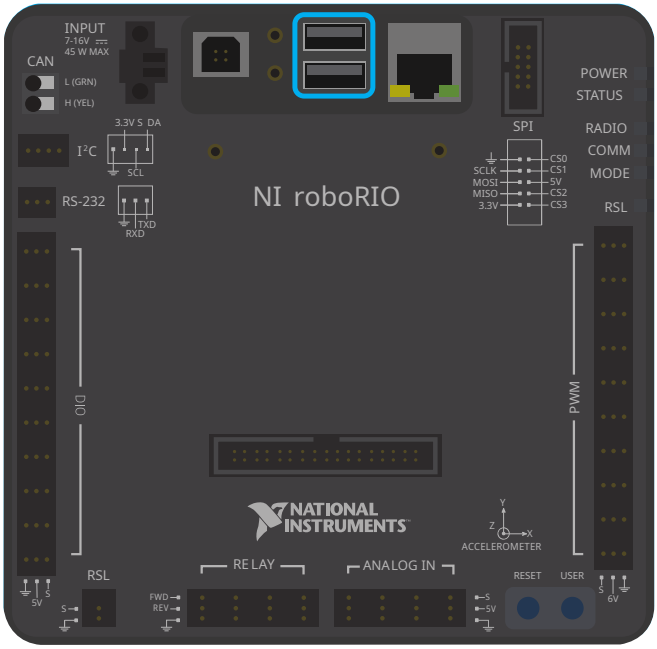
Note

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*.

39.12.5 USB Client

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.

39.12.6 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.

39.12.7 MXP Expansion Port

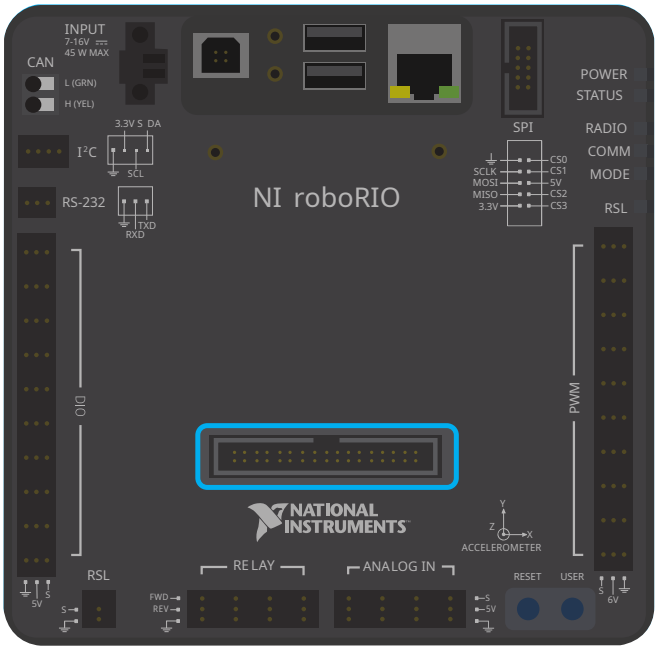
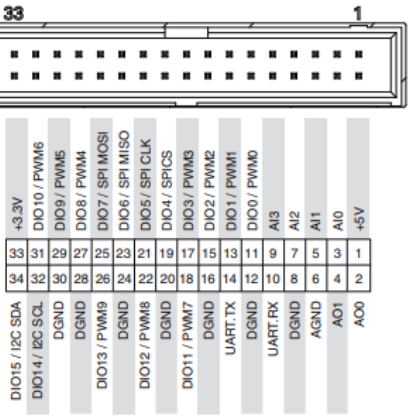


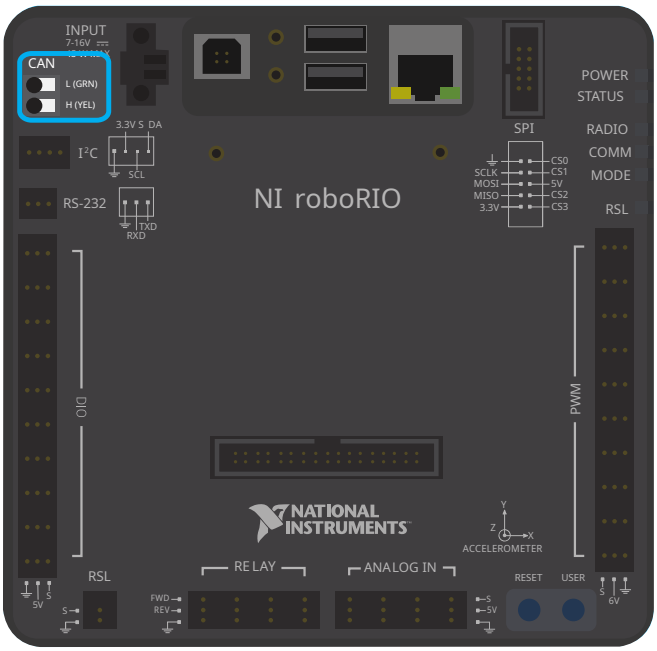
Figure 4. MXP Pinout



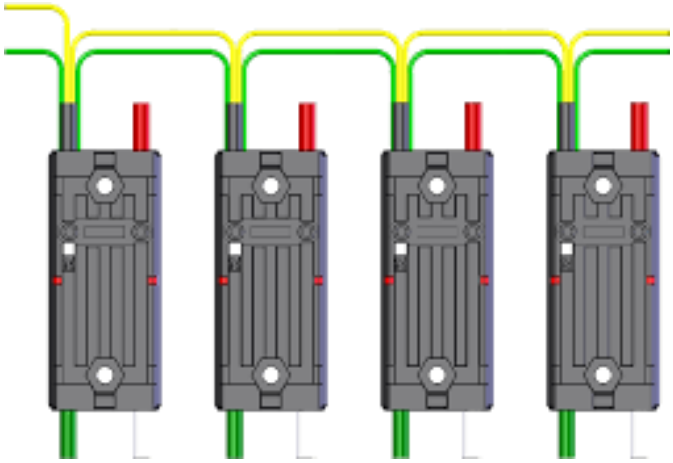
Several of the serial buses are also available for use through the roboRIO's *MXP* Expansion Port. This port allows users to make use of many additional *digital* and *analog* inputs, as well as the various serial buses.

Many peripheral devices attach directly to the MXP port for convenience, requiring no wiring on the part of the user.

39.12.8 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 *CTRE Power Distribution Panel (PDP)* and the *REV Power Distribution Hub (PDH)*

More information about using devices connected to the CAN bus can be found in the article about *using can devices*.

40

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.

Tip

A course that teaches programming using the Romi Robot is available via FIRST Training. Part 1 takes users through setup of the hardware and software, methods, objects, conditionals, variables, and equations. Part 2 covers command based programming, driving and inputs, advanced conditionals and logical operators, command groups, arrays, and loops, and errors and troubleshooting.



40.1 Romi Hardware & Assembly

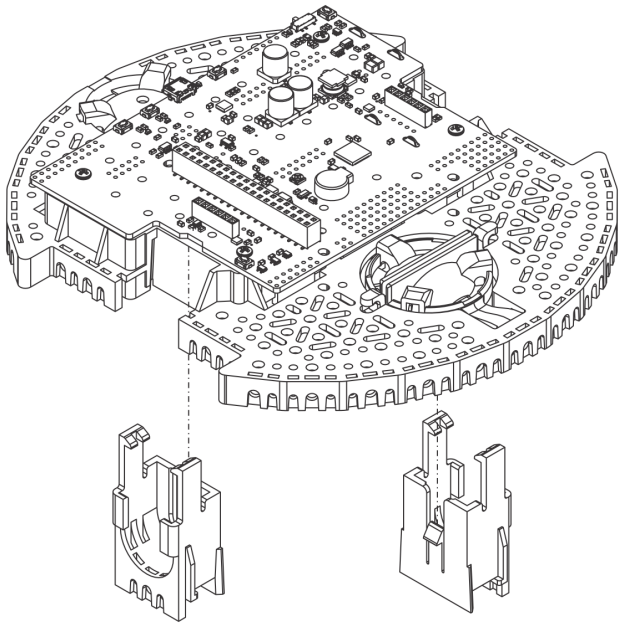
To get started with the Romi, you will need to have the necessary hardware.

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)

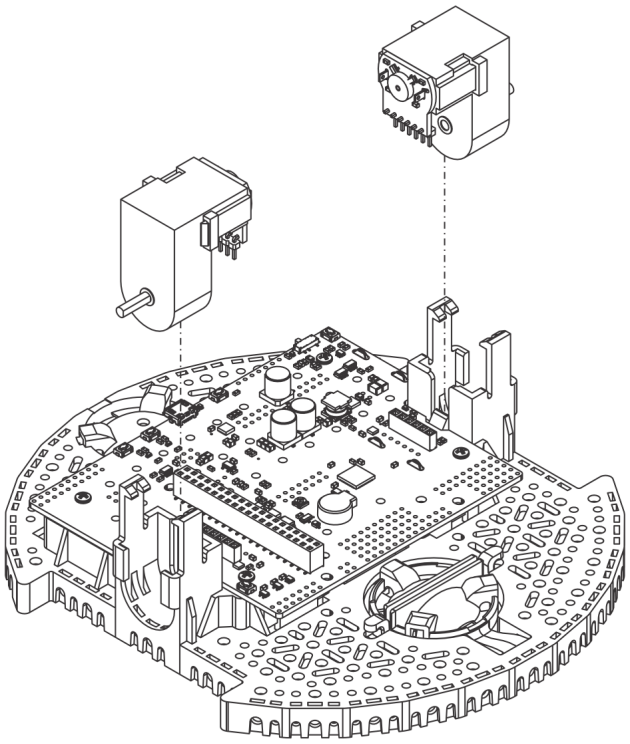
40.1.1 Assembly

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:

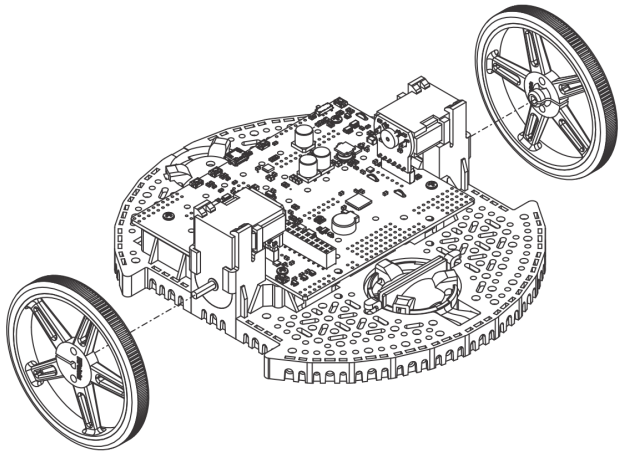
1. Align the motor clips with the chassis as indicated and press them firmly into the chassis until the bottom of the clips are even with the bottom of the chassis (you may hear several clicks).



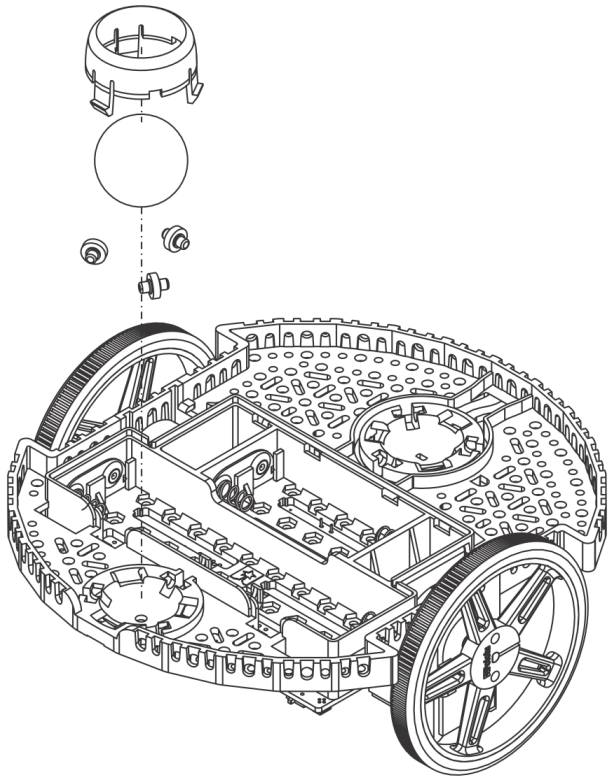
2. Push the Mini Plastic Gearmotors into the motor clips until they snap into place. Note that the motor blocks the clip release, so if you need to remove a motor bracket later, you will first need to remove the motor. The Mini Plastic Gearmotors that come with the kit have extended motor shafts to enable quadrature encoders for position feedback.



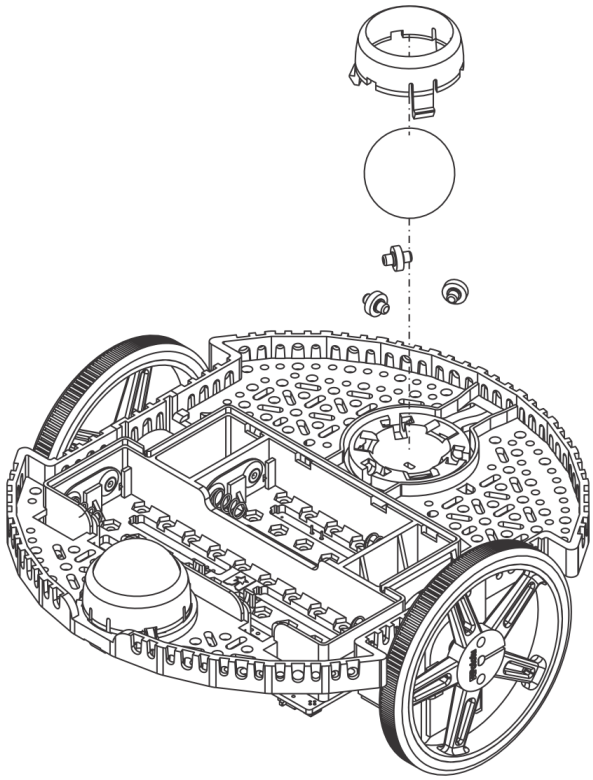
3. Press the wheels onto the output shafts of the motors until the motor shaft is flush with the outer face of the wheel. One way to do this is to set the wheel on a flat surface and line the chassis up with it so that the flat part of the motor's D-shaft lines up correctly with the wheel. Then, lower the chassis, pressing the motor shaft into the wheel until it contacts the surface.



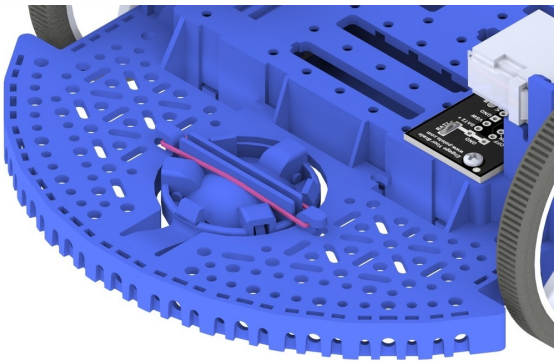
4. Flip the chassis upside down and place the three rollers for the rear ball caster into the cutouts in the chassis. Place the 1″ plastic ball on top of the three rollers. Then push the ball caster retention clip over the ball and into the chassis so the three legs snap into their respective holes.



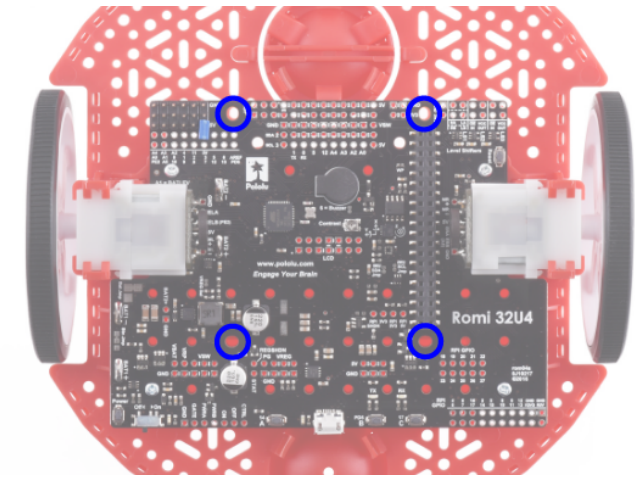
5. Repeat for the front ball caster so there is a caster on the front and the back of the robot.



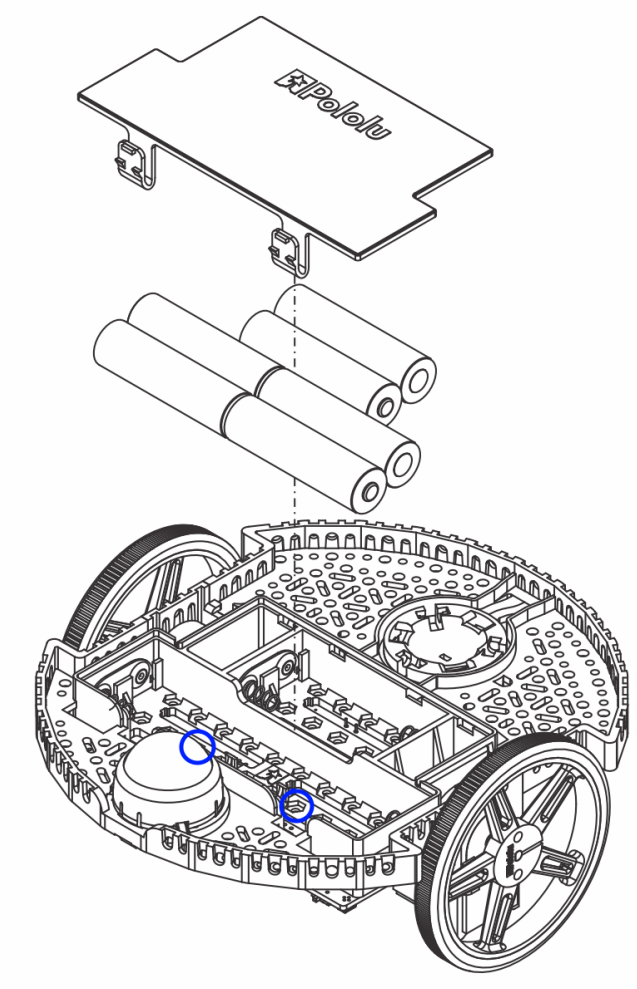
6. Optional: The front ball caster is supported by a flexible arm that acts as a suspension system. If you want to make it stiffer, you can wrap a rubber band around the two hooks located on either side of the ball caster on the top side of the chassis.



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!

40.2 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 operates properly.

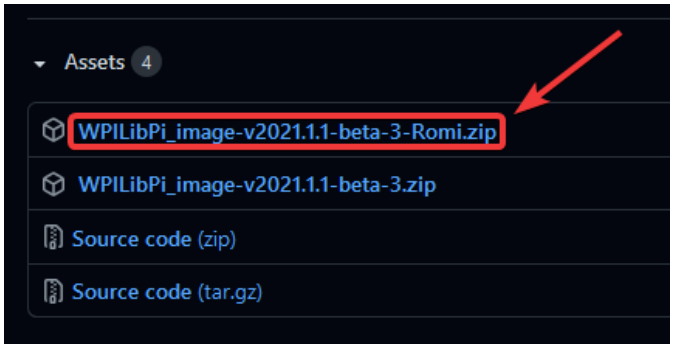
40.2.1 Raspberry Pi

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/`.

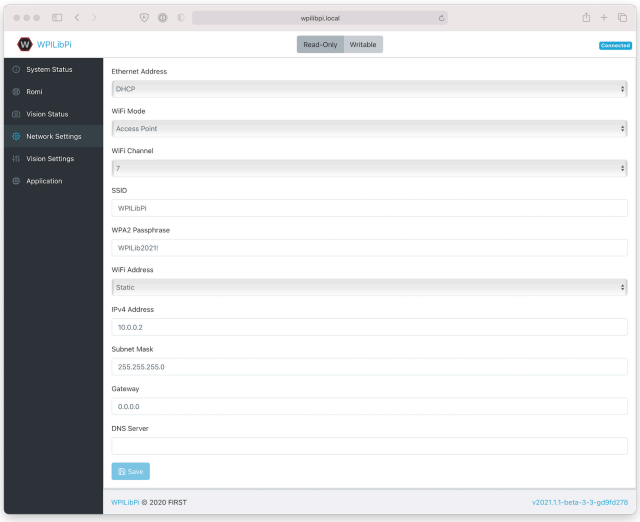
Note

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 a wireless network with other Romis.
8. Reconnect to the Romi's WiFi network with the new password you set.

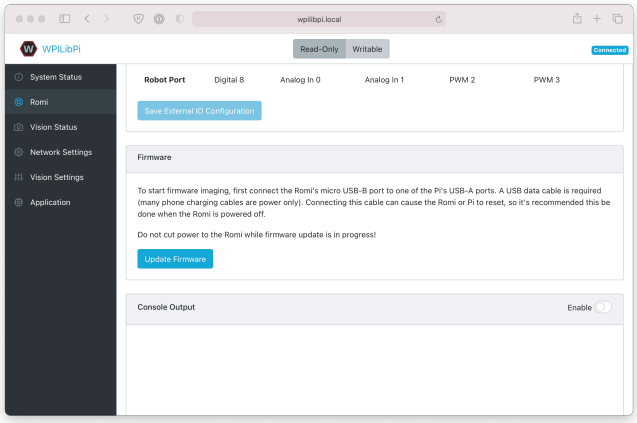
Be sure to set the Dashboard to `Read-only` when all the changes have been completed.



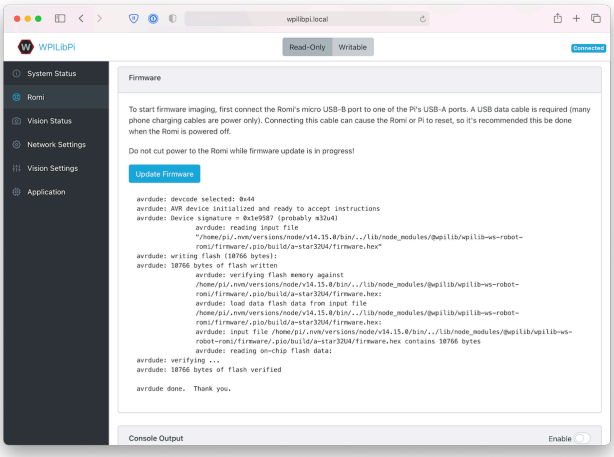
40.2.2 32U4 Control Board

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.



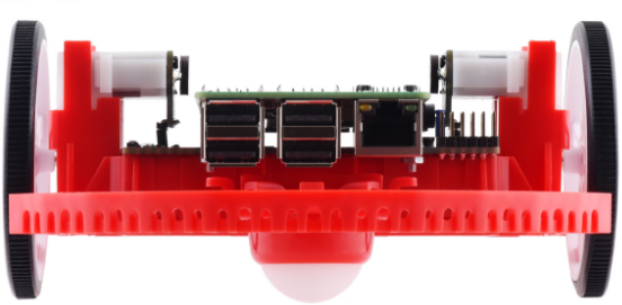
A console will appear showing a log of the firmware deploy process. Once the firmware has been deployed to the 32U4 Control Board, the message `avrdude done. Thank you.` will appear.



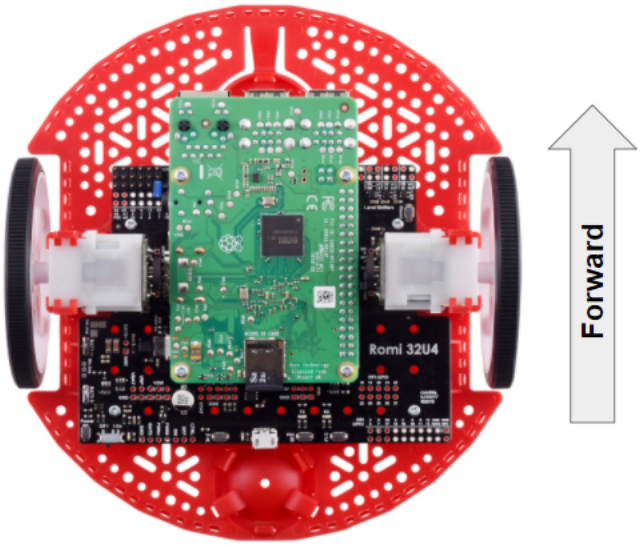
40.3 Getting to know your Romi

40.3.1 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”.



40.3.2 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 (EXT)
- Buzzer

Note

The Buzzer is currently not supported by WPILib.

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.

The motor *PWM* channels are listed in the table below.

Channel	Romi Hardware Component
PWM 0	Left Motor
PWM 1	Right Motor

Note

The right motor will spin in a backward direction when positive output is applied. Thus, the corresponding motor controller needs to be inverted in robot code.

The encoder channels are listed in the table below.

Channel	Romi Hardware Component
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

Note

By default, the encoders count up when the Romi moves forward.

Inertial Measurement Unit

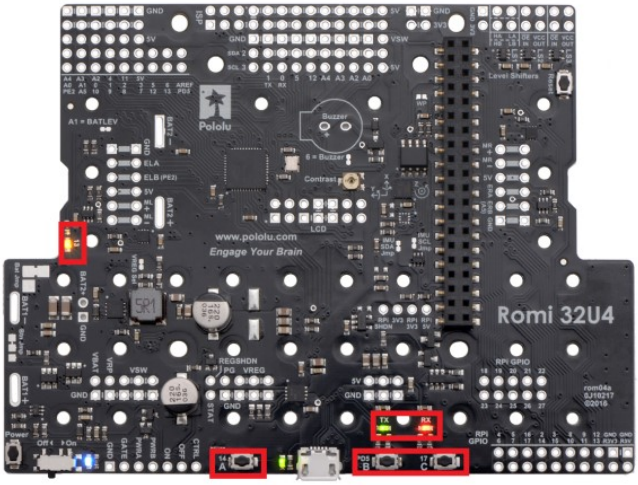
The Romi includes an STMicroelectronics LSM6DS33 Inertial Measurement Unit (IMU) which contains a 3-axis gyro and a 3-axis accelerometer.

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.

Onboard LEDs and Push Buttons

The Romi 32U4 control board has 3 push buttons and 3 LEDs onboard that are exposed as Digital IO (DIO) channels to robot code.

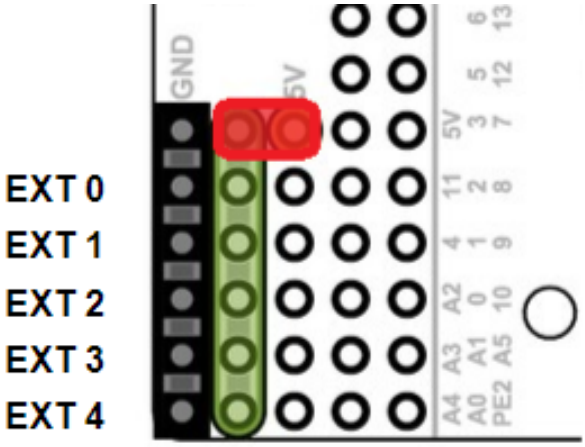


DIO Channel	Romi Hardware Component
DIO 0	Button A (input only)
DIO 1	Button B (input), Green LED (output)
DIO 2	Button C (input), Red LED (output)
DIO 3	Yellow LED (output only)

Writes to DIO 0, 4, 5, 6, and 7 will result in a *no-op*.

Configurable GPIO Pins

The control board has 5 configurable GPIO pins (named EXT0 through EXT4) that allow a user to connect external sensors and actuators to the Romi.



All 5 pins support the following modes: Digital IO, Analog In, and PWM (with the exception of EXT 0, which only supports Digital IO and PWM). The mode of the ports can be configured with *The Romi Web UI*.

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 pins 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.

GPIO Default Configuration

The table below shows the default configuration of the GPIO pins (EXT0 through EXT4). *The Romi Web UI* allows the user to customize the functions of the 5 configurable GPIO pins. The UI will also provide the appropriate WPILib channel/device mappings on screen once the IO configuration is complete.

Channel	Ext Pin
DIO 8	EXT0
Analog In 0	EXT1
Analog In 1	EXT2
PWM 2	EXT3
PWM 3	EXT4

40.4 Romi Hardware Support

The Romi robot, having a different hardware architecture than a roboRIO, is compatible with a subset of commonly used FRC control system components.

40.4.1 Compatible Hardware

In general, the Romi is compatible with the following:

- Simple Digital Input/Output devices (e.g. bumper switches, single LEDs)
- Standard RC-style *PWM* output devices (e.g. servos, PWM based motor controllers)
- Analog Input sensors (e.g distance sensors that report distance as a voltage)

40.4.2 Incompatible Hardware

Due to hardware limitations, the Romi Robot is not compatible with the following:

- Encoders other than the Romi-integrated encoders
- "Ping" style ultrasonic sensors (which require 2 DIO channels)
- Timing based sensors
- CAN based devices
- Romi built-in buzzer

40.4.3 Compatible Classes

All classes listed here are supported by the Romi Robot. If a class is not listed here, assume that it is not supported and *will not* work.

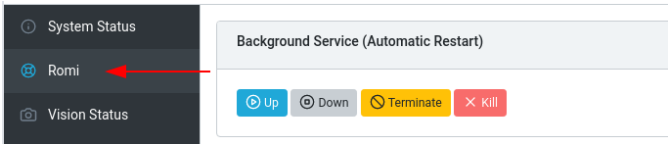
- PWM Motor Controllers (i.e. `Spark`)
- `Encoder`
- `AnalogInput`
- `DigitalInput`
- `DigitalOutput`
- `Servo`
- `BuiltInAccelerometer`

The following classes are provided by the Romi Vendordep.

- `RomiGyro`
- `RomiMotor`
- `OnboardIO`

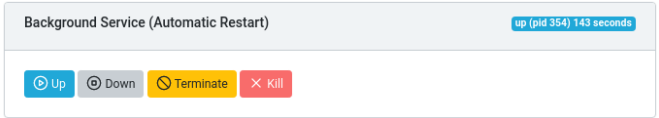
40.5 The Romi Web UI

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.

40.5.1 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.

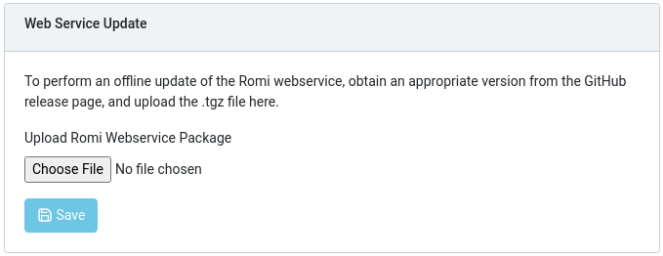
40.5.2 Romi Status



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*

40.5.3 Web Service Update

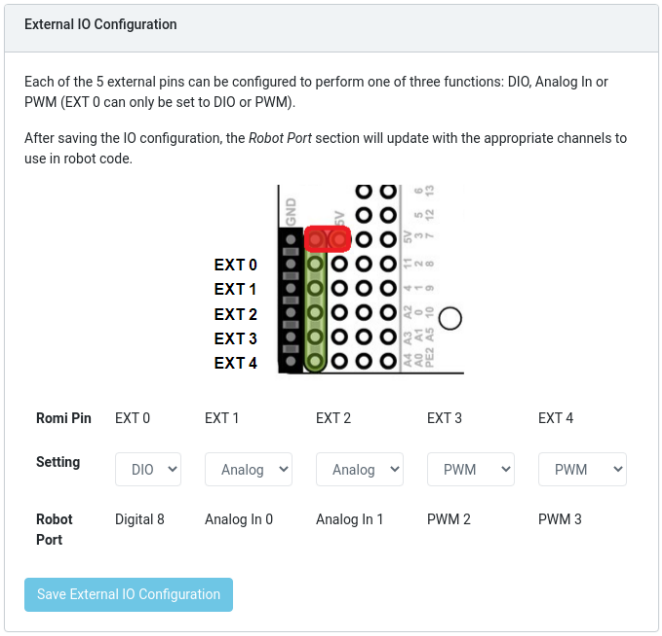


> **Note**
>
> The Raspberry Pi must be in **Writable** mode for this section to work.

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.

40.5.4 External IO Configuration



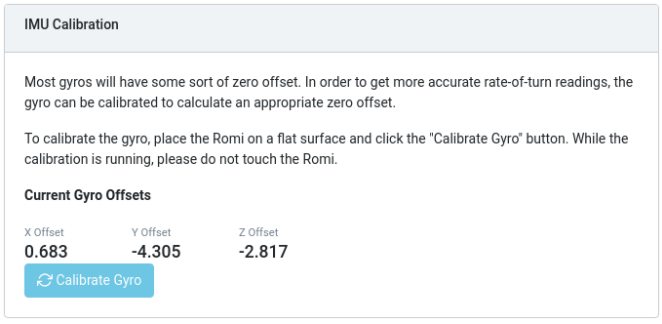
This section allows users to configure the 5 external GPIO channels on the Romi.

> **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.

40.5.5 IMU Calibration



> **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.

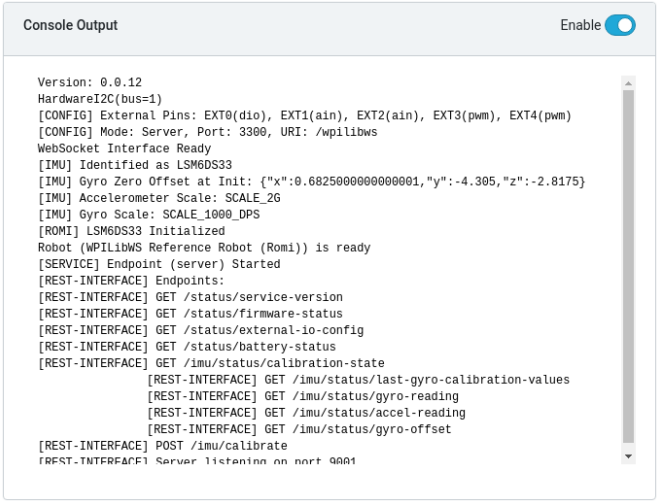
To begin calibration, place the Romi on a flat, stable surface. Then, click the *Calibrate Gyro* button. A progress bar will appear, showing the current calibration process. Once calibration is complete, the latest offset values will be displayed on screen and registered with the Romi web service.

These offset values are saved to disk and persist between reboots.

40.5.6 Firmware

> **Note**
>
> See the section on *Imaging your Romi*

40.5.7 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.

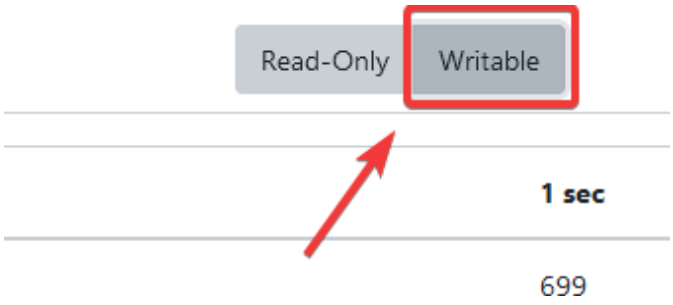
40.5.8 Bridge Mode

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.

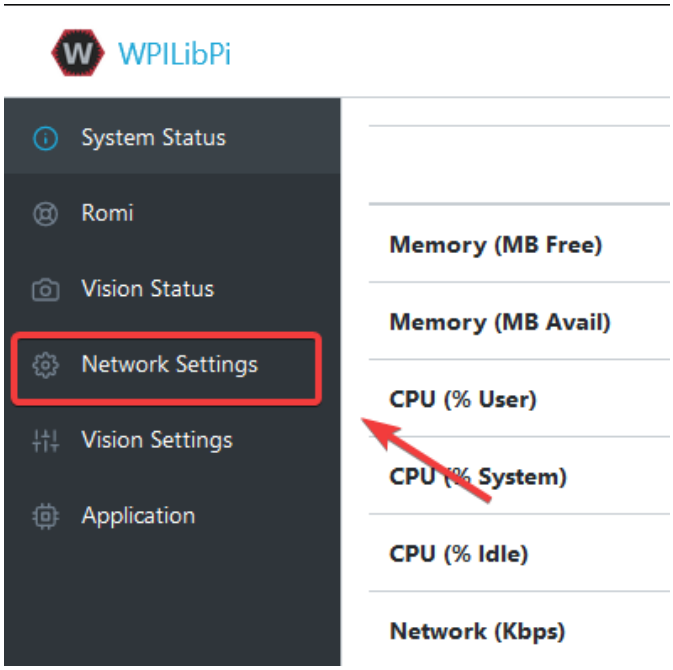
Note

Bridge mode is not likely to work properly in restricted network environments (Educational Institutions).

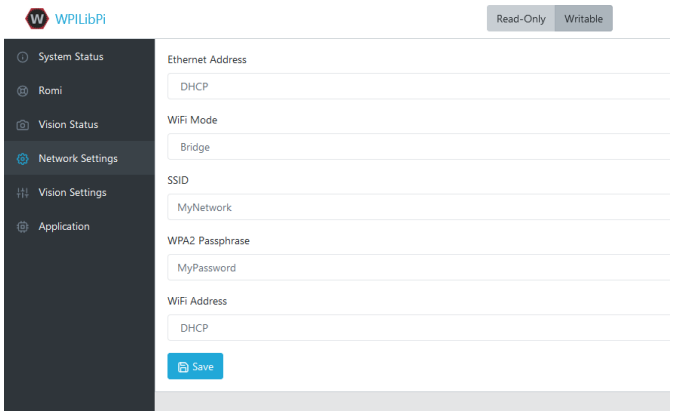
1. Enable *Writable* in the top menu.



2. Click on *Network Settings*.



3. The following network settings must be applied:



- **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!

40.6 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.

40.6.1 Creating a Romi Program

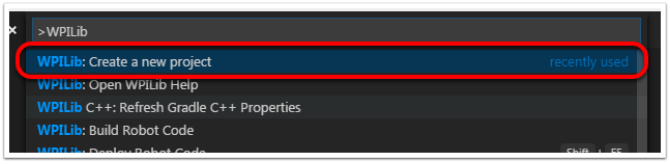
Creating a new program for a Romi is like creating a normal FRC program, similar to the *Zero To Robot* programming steps.

WPILib comes with two templates for Romi projects, including one based on TimedRobot, and a Command-Based project template. Additionally, an example project is provided which showcases some of the built-in functionality of the Romi. This article will walk through creating a project from this example.

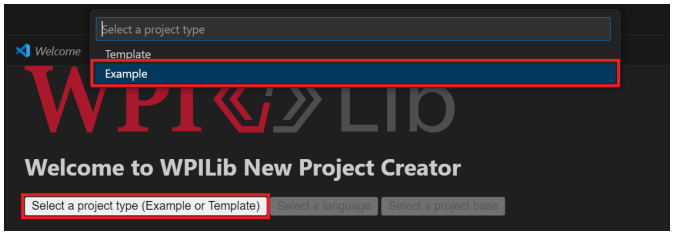
> **Note**
>
> In order to program the Romi using C++, a compatible C++ desktop compiler must be installed. See *Robot Simulation - Additional C++ Dependency*.

Creating a New WPILib Romi Project

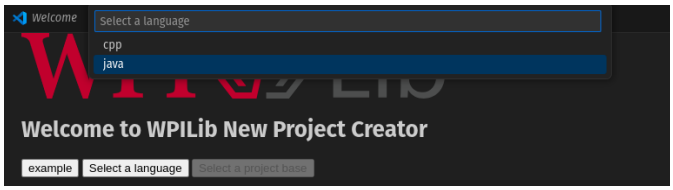
Bring up the Visual Studio Code command palette with `Ctrl+Shift+P`, and type "New project" into the prompt. Select the "Create a new project" command:



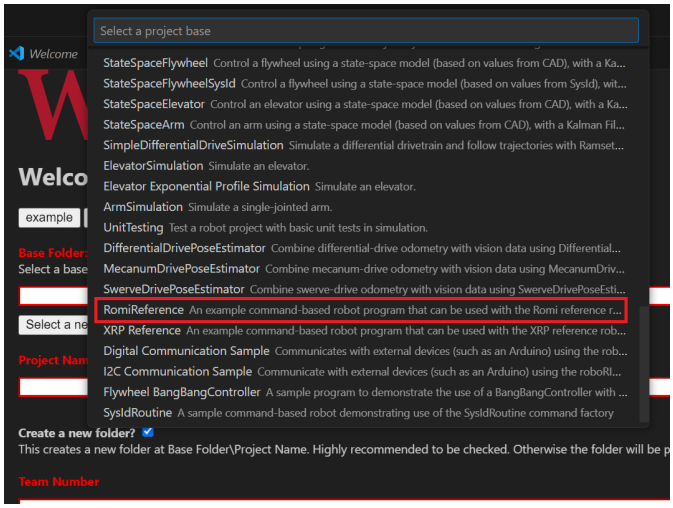
This will bring up the "New Project Creator Window". From here, click on "Select a project type (Example or Template), and pick "Example" from the prompt that appears:



Select the programming language you want to use:



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

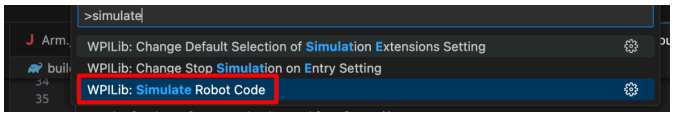
Once the robot project is generated, it is essentially ready to run. The project has a pre-built `Drivetrain` class and associated default command that lets you drive the Romi around using a joystick.

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.

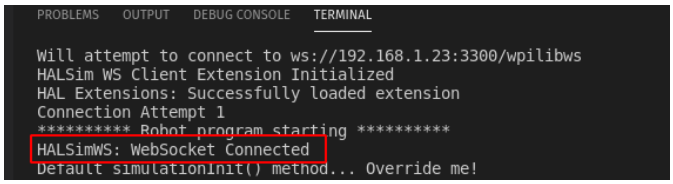
To run a Romi program, first, ensure that your Romi is powered on. Next, connect to the `WPILibPi-<number>` WiFi network broadcast by the Romi. If you changed the Romi network settings (for example, to connect it to your own WiFi network) you may change the IP address that your program uses to connect to the Romi. To do this, open the `build.gradle` file and update the `wpi.sim.envVar` line to the appropriate IP address.

```
44 //Sets the websocket client remote host.
45 wpi.sim.envVar("HALSIMWS_HOST", "10.0.0.2")
46 wpi.sim.addWebsocketsServer().defaultEnabled = true
47 wpi.sim.addWebsocketsClient().defaultEnabled = true
```

Now to start your Romi robot code, open the WPILib Command Palette (type `Ctrl+Shift+P`) and select "Simulate Robot Code", or press `F5`.



If all goes well, you should see a line in the console output that reads "HALSimWS: WebSocket Connected":



Your Romi code is now running!

40.7 Programming the Romi (LabVIEW)

Writing a LabVIEW program for the Romi is very similar to writing a program for a regular roboRIO based robot. In fact, all the same tools can be used with the Romi.

40.7.1 Creating a Romi Project

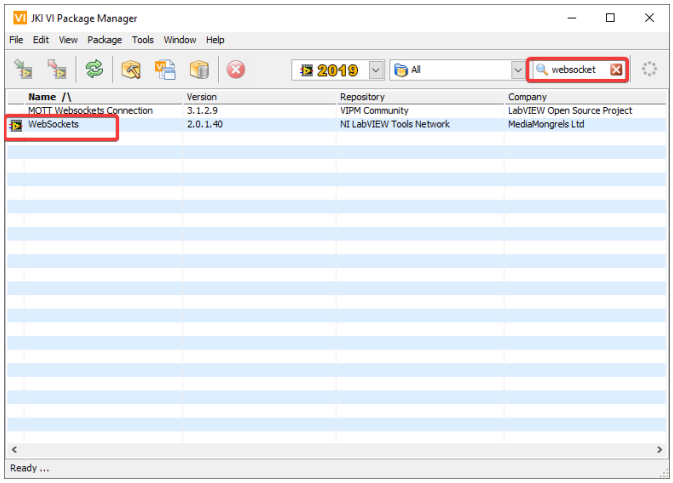
Creating a new program for a Romi is no different than creating a normal FRC |reg| program, similar to the *Zero To Robot* programming steps. Initially, you may wish to create a separate project for use on just the Romi as the Romi hardware may be connected to different ports than on your roboRIO robot.

The Romi Robot used *PWM* ports 0 and 1 for left and right side respectively.

Installing the WebSockets VI

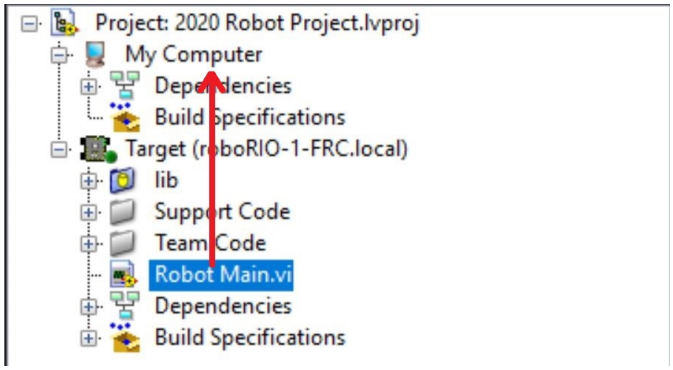
One aspect where a Romi project differs from a regular FRC |reg| robot project is that the code is not deployed directly to the Romi. Instead, a Romi project runs on your development computer, and leverages the WPILib simulation framework to communicate with the Romi robot. WebSockets is the protocol that LabVIEW uses to converse with the Romi.

Open the *VI Package Manager* application. Type `websockets` into the search box in the top right. Select the VI by *LabVIEW Tools Network*.



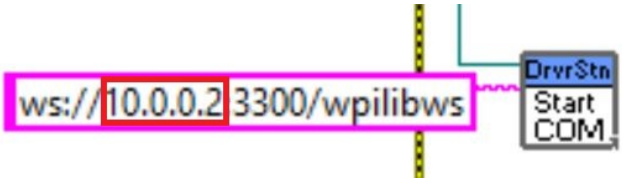
Changing the Project Target

The primary step needed to run your LabVIEW program on the Romi is to change the target to the Desktop. To change the project target, locate the Robot Main VI in the Project Explorer and click and drag it from the `Target` section to the `My Computer` section.



Setting the Target IP

By default, your LabVIEW program will attempt to connect to a Romi with the IP address of `10.0.0.2`. If you wish to use a different IP, you can specify it as an input to the `Driver Station Start Communication VI` inside `Robot Main`. Locate the pink input terminal for `Simulation URL` then right-click and select *Create Constant* to create a constant pre-filled with the default value. You can then modify the IP address portion of the text, taking care to leave the protocol section (at the beginning) and port and suffix (at the end) the same.



Running a Romi Program

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 the white *Run* arrow to start running the Romi program on your computer.

Your Romi code is now running! The program will automatically attempt to connect to either the IP you have specified, or the default if you have not specified an IP.

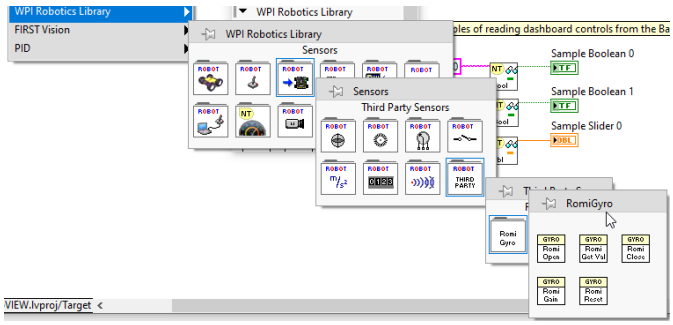
It is recommended to run the Driver Station software on the same computer as the LabVIEW code. Once your program successfully connects to the Driver Station, it will automatically notify the Driver Station that the code is running on the Desktop, allowing the Driver Station to connect without you changing any information inside the Driver Station. Next, you'll need to point the Driver Station to your Romi. This is done by setting the team number to `127.0.0.1`. You can then use the controls in the Driver Station to set the robot mode and enable/disable as normal.

> **Note**
>
> If your robot code is unable to connect to the Romi, the Driver Station will also show no connectivity.

Using the Gyro or Encoder

The gyro that is available on the Romi is available using the RomiGyro functions. This is located under

```
- WPI Robotics Library
   - Sensors
      - Third Party Libraries
         - RomiGyro
```



The encoders can be used using the standard encoder function. The DIO ports are:

- Left (4, 5)
- Right (6, 7)

41

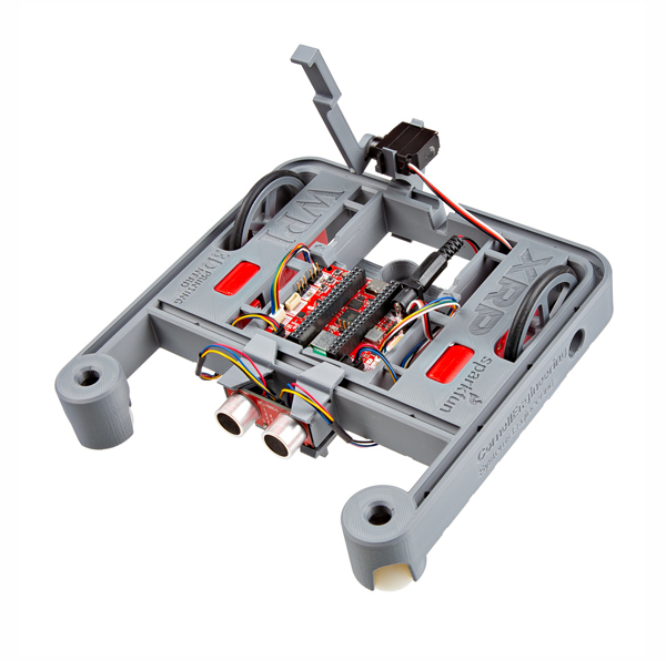
Getting Started with XRP

The XRP, powered by Worcester Polytechnic Institute, 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 XRP. The XRP 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 XRP to follow the steps in the program.

Tip

A course that teaches programming using the XRP Robot is available via FIRST Training. Part 1 takes users through setup of the hardware and software, methods, objects, conditionals, variables, and equations. Part 2 covers command based programming, driving and inputs, advanced conditionals and logical operators, command groups, arrays and loops, and errors and troubleshooting.

The XRP provides a similar use case as the *Romi* with similar functionality, albeit using a lower power processor and is overall lower in cost.



41.1 XRP Hardware, Assembly, and Imaging

To get started with the XRP, you will need to have the necessary hardware.

1. XRP Kit from SparkFun or from DigiKey - Available at a discount for educational institutions or FIRST teams. See individual vendors for details.
2. USB-C cable. Beta versions of XRP use a Micro-USB cable - Ensure that this is a data cable
3. 4 AA batteries - Rechargeable (example) is best (don't forget the charger)

41.1.1 Assembly

Note

See the assembly instructions on the XRP User Guide.

You should follow the instructions up to and including the point where the XRP arm is mounted to the servo.

41.1.2 Imaging your XRP

The XRP uses a Raspberry Pi Pico W as its main processor. A special firmware will need to be installed so that the robot operates properly.

Download

There are different firmware versions for each version of the XRP robot. Use this link to identify which one you have.

The XRP firmware must be downloaded and written to the Pico W. Click on `Assets` at the bottom of the description to see the available image files:

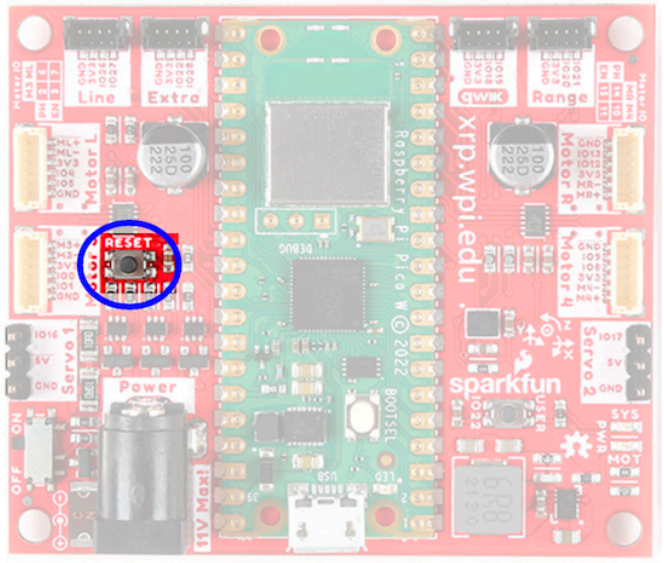
XRP-WPILib Firmware

Beta versions of the XRP have 'Beta' in the name of the file. V1 releases do not. If you choose the wrong one it will simply not be installed, just return and download the correct one.

Imaging

To image the XRP, perform the following steps:

1. Plug the XRP into your computer with a Micro-USB cable. You should see a red power LED that lights up.
2. While holding the `BOOTSEL` button (the white button on the green Pico W, near the USB connector), quickly press the reset button (circled below), and then release the `BOOTSEL` button.

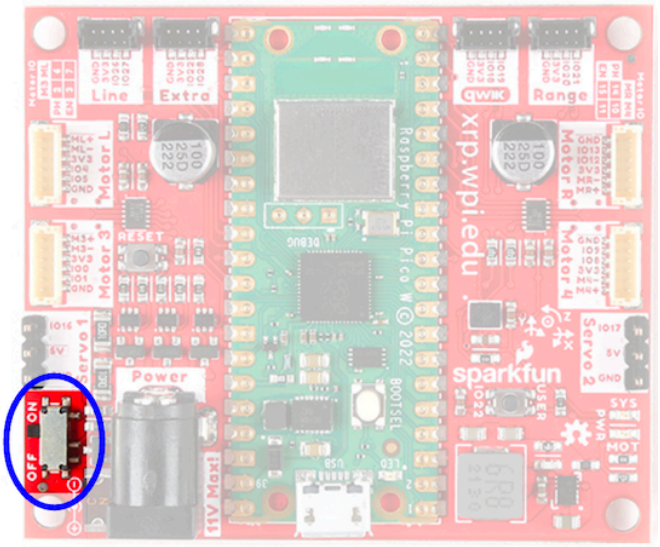


3. The board will temporarily disconnect from your computer, and then reconnect as a USB storage device named `RPI-RP2`.
4. Drag the `.uf2` firmware file into the `RPI-RP2` drive, and it will automatically update the firmware.
5. Once complete, the `RPI-RP2` USB storage device will disconnect. At this point, you can disconnect the XRP board from your computer and run it off battery power.

First Boot

Perform the following steps to get your XRP ready for use:

1. Ensure that you have 4 AA batteries installed
2. Turn the XRP on by sliding the power switch (circled below) on the XRP board to the on position. A red power LED will turn on.



3. Using your computer, connect to the XRP WiFi network using the SSID `XRP-<IDENT>` (where `<IDENT>` is based on the unique ID of the Pico W) with the WPA2 passphrase `xrp-wpilib`.

> **Note**
>
> If powering on the XRP in an environment with multiple other XRPs, the SSID can also be found by connecting the XRP to a computer, navigating to the USB storage device (`PICODISK`) that appears and opening the `xrp-status.txt` file.

4. Open a web browser and connect to the web UI at `http://192.168.42.1:5000`. If the page loads, you have established connectivity with the XRP.

> **Note**
>
> More information about the Web UI and configuration can be found in the *Web UI section*.

41.2 Getting to know your XRP

41.2.1 Booting up the XRP

Upon start up (when power is applied to the XRP either via battery or USB), the following will happen:

1. The IMU will calibrate itself. This lasts approximately 3-5 seconds, and will be indicated by the green LED blinking rapidly. Ideally, the XRP should be placed on a flat surface prior to power up, and if necessary, users can hit the reset button to restart the firmware and IMU calibration process.
2. The network will be configured, depending on the configuration settings. See the section on *the Web UI* for more information on how to configure the network settings. By default the XRP will broadcast its own WiFi Access Point.
3. After this, the XRP is ready for use.

41.2.2 Hardware, Sensors, and GPIO

The XRP has the following built-in hardware/peripherals:

- 2x geared drive motors with encoders
- 2x additional geared motor connectors with encoder support (marked Motor3 and Motor4)
- 2x Servo connectors (marked Servo1 and Servo2)
- 1x Inertial Measurement Unit (IMU)
- 1x LED (green)
- 1x pushbutton (marked USER)
- 1x Line following sensor (exposed as 2 Analog inputs)
- 1x Ultrasonic PING style rangefinder (uses 2 digital IO pins, exposed as an analog input)

Motors, Wheels, and Encoders

The motors used on the XRP have a 48.75:1 gear reduction and a no-load output speed of 90 RPM at 4.5V.

The wheels have a diameter of 60mm (2.3622”). They have a trackwidth of 155mm (6.1”).

The encoders are connected directly to the motor output shaft and have 12 Counts Per Revolution (CPR). With the provided gear ratio, this nets 585 counts per wheel revolution.

The motor channels are listed in the table below.

> **Note**
>
> We use “motor channels” here instead of “PWM channels” as the XRP requires the use of a special `XRPMotor` object in WPILib code to interact with the hardware.

Channel	XRP Hardware Component
XRPMotor 0	Left Motor
XRPMotor 1	Right Motor
XRPMotor 2	Motor 3
XRPMotor 3	Motor 4

Note

The right motor will spin in a backward direction when positive output is applied. Thus the corresponding motor controller needs to be inverted in robot code.

The servo channels are listed in the table below.

Note

We use "servo channels" here instead of "PWM channels" as the XRP requires the use of a special `XRPServo` object in WPILib code to interact with the hardware.

Channel	XRP Hardware Component
XRPServo 4	Servo 1
XRPServo 5	Servo 2

The encoder channels are listed in the table below.

Channel	XRP Hardware Component
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
DIO 8	Motor3 Encoder Quadrature Channel A
DIO 9	Motor3 Encoder Quadrature Channel B
DIO 10	Motor4 Encoder Quadrature Channel A
DIO 11	Motor4 Encoder Quadrature Channel B

Note

By default, the encoders count up when the XRP moves forward.

Inertial Measurement Unit

The XRP includes an STMicroelectronics LSM6DSOX Inertial Measurement Unit (IMU) which contains a 3-axis gyro and a 3-axis accelerometer.

The XRP will calibrate the gyro and accelerometer upon each boot (the onboard LED will quickly flash for about 3-5 seconds at startup time).

Onboard LED and Push Button

The XRP has a push button (labeled USER) and a green LED onboard that are exposed as Digital IO (DIO) channels to robot code.

DIO Channel	XRP Hardware Component
DIO 0	USER Button
DIO 1	Green LED

Note

DIO 2 and 3 are reserved for future system use.

Line Following (Reflectance) Sensor

When assembled according to the instructions, the XRP supports a line following sensor with 2 sensing elements. Each sensing element measures reflectance exposes these as AnalogInput channels to robot code. The returned values range from 0V (pure white) to 5V (pure black).

AnalogInput Channel	XRP Hardware Component
AnalogInput 0	Left Reflectance Sensor
AnalogInput 1	Right Reflectance Sensor

Ultrasonic Rangefinder

When assembled according to the instructions, the XRP supports an ultrasonic, PING style, rangefinder. This is exposed as an AnalogInput channel to robot code. The returned values range from 0V (20mm) to 5V (4000mm).

AnalogInput Channel	XRP Hardware Component
AnalogInput 2	Ultrasonic Rangefinder

41.3 XRP Hardware Support

The XRP robot, having a different hardware architecture than a roboRIO, is compatible with a subset of commonly used FRC control system components.

41.3.1 Compatible Hardware

In general, the XRP is compatible with the following:

- Hobby DC motors with built-in encoders (6-pin connector)
- Standard RC-style *PWM* output devices (e.g. servos, PWM based motor controllers)
- "Ping" style ultrasonic sensors (only when connected to the RANGE port)

41.3.2 Incompatible Hardware

Due to hardware limitations, the XRP is not compatible with the following:

- Encoders other than those already integrated into hobby motors
- Timing based sensors
- CAN based devices

41.3.3 Compatible Classes

All classes listed here are supported by the XRP. If a class is not listed here, assume that it is not supported and *will not* work.

- `Encoder`
- `AnalogInput`
- `DigitalInput`
- `DigitalOutput`
- `BuiltInAccelerometer`

Note

The PWM motor controller classes (e.g. `Spark`) and `Servo` are not supported. The XRP requires use of specialized `XRPMotor` and `XRPServo` classes.

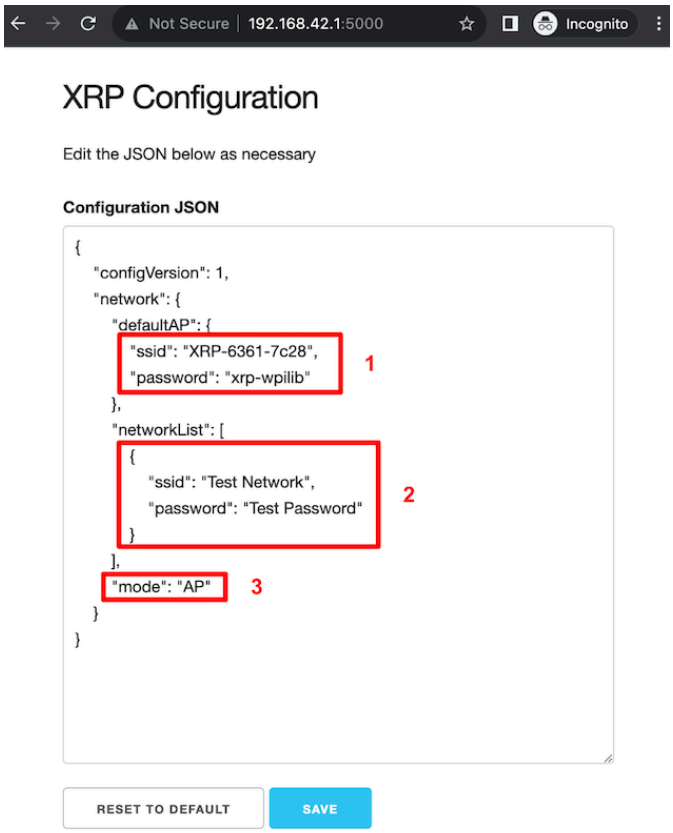
The following classes are provided by the XRP Vendordep (built-in to WPILib).

- `XRPGyro`
- `XRPMotor`
- `XRPServo`
- `XRPOnBoardIO`

41.4 The XRP Web UI

The XRP provides a simple Web UI for configuration. It is accessible at `http://<IP Address of XRP>:5000`. By default, this is `http://192.168.42.1:5000`.

The XRP configuration is a simple JSON object that allows a user to configure the network settings of the XRP.



41.4.1 Switching Network Modes

Box 3 in the image above shows the field that needs to be changed in order to switch the XRP from Access Point mode to/from Station mode. In Access Point (`AP`) mode, the XRP will broadcast a WiFi network. In Station (`STA`) mode, the XRP will connect to an existing WiFi network. Update the `mode` field with the appropriate value (`AP`/`STA`).

41.4.2 Setting up a default Access Point (AP)

By default, the XRP will operate in Access Point mode, where it broadcasts a WiFi network. Box 1 in the image above shows which fields control the settings for the AP SSID and passphrase.

If the operating mode is set to `AP`, the access point information will be used to create the WiFi Access Point. If the mode is set to `STA` (station) and the XRP is unable to connect to any of the listed WiFi networks, then it will fall back to AP mode, again, using the information specified in box 1.

41.4.3 Connecting to an existing WiFi network

Box 2 in the image above shows an example of listing a WiFi network that you want the XRP to connect to. the `networkList` array can be populated with as many preferred networks as you would like (following the same format as Box 2). When set to `STA` mode, the XRP will attempt to connect to each listed network in order. If none of the networks are available, the XRP will fallback into AP mode.

> **Note**
>
> If you are unsure about what mode the XRP is operating in, or which WiFi network it is connected to, you can connect the XRP to a computer via a USB cable. A USB storage device named `PICODISK` will appear, and the `xrp-status.txt` file within it will list the appropriate network information.

41.5 Programming the XRP

Writing a program for the XRP 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 XRP.

41.5.1 Creating an XRP Program

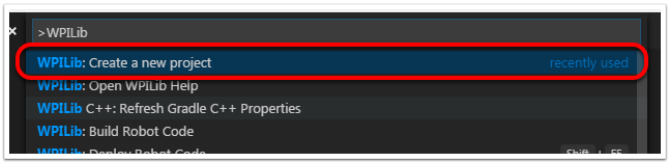
Creating a new program for an XRP is like creating a normal FRC program, similar to the *Zero To Robot* programming steps.

WPILib, created by Worcester Polytechnic Institute, comes with two templates for XRP projects, including one based on TimedRobot, and a Command-Based project template. Additionally, an example project is provided which showcases some of the built-in functionality of the XRP, and shows how to use the vendordep exposed XRP classes. This article will walk through creating a project from this example.

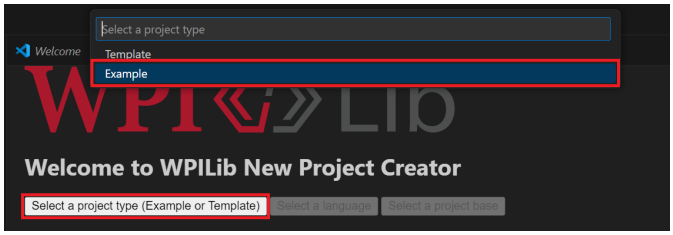
> **Note**
>
> In order to program the XRP using C++, a compatible C++ desktop compiler must be installed. See *Robot Simulation - Additional C++ Dependency*.

Creating a New WPILib XRP Project

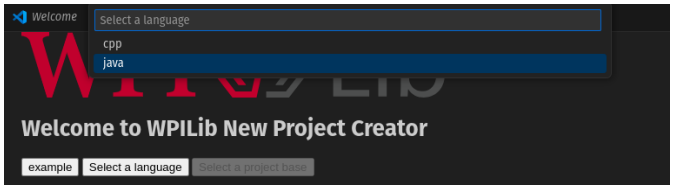
Bring up the Visual Studio Code command palette with `Ctrl+Shift+P`, and type "New project" into the prompt. Select the "Create a new project" command:



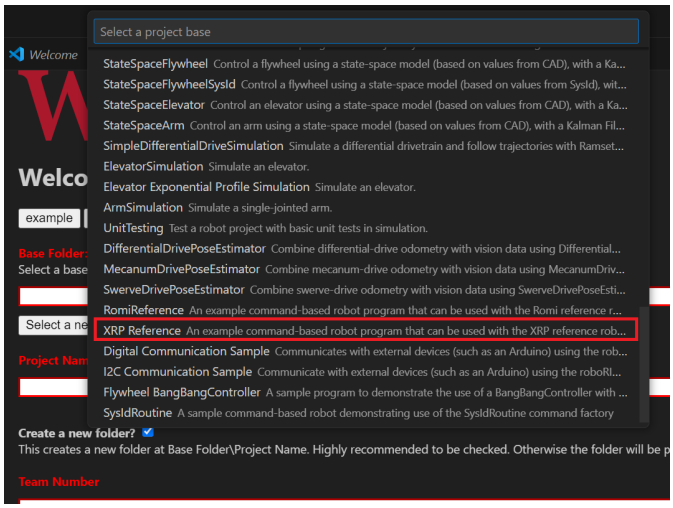
This will bring up the "New Project Creator Window". From here, click on "Select a project type (Example or Template)", and pick "Example" from the prompt that appears:



Select the programming language you want to use:



Next, a list of examples will appear. Scroll through the list to find the "XRP Reference" example:



Fill out the rest of the fields in the "New Project Creator" and click "Generate Project" to create the new robot project.

Running an XRP Program

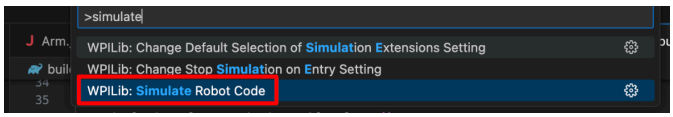
Once the robot project is generated, it is essentially ready to run. The project has a pre-built `Drivetrain` class and associated default command that lets you drive the XRP around using a joystick.

One aspect where an XRP project differs from a regular FRC robot project is that the code is not deployed directly to the XRP. Instead, an XRP project runs on your development computer and leverages the WPILib simulation framework to communicate with the XRP.

To run an XRP program, first, ensure that your XRP is powered on. Next, connect to `XRP-<IDENT>` WiFi network broadcast by the XRP. If you changed the XRP network settings (for example, to connect it to your own network), you may change the IP address that your program uses to connect to the XRP. To do this, open the `build.gradle` file and update the `wpi.sim.envVar` line to the appropriate IP address.

```
44 //Sets the XRP Client Host
45 wpi.sim.envVar("HALSIMXRP_HOST", "192.168.42.1")
46 wpi.sim.addXRPClient().defaultEnabled = true
```

Now to start your XRP robot code, open the WPILib Command Palette (type `Ctrl+Shift+P`) and select “Simulate Robot Code”, or press `F5`.



If all goes well, you should see the simulation GUI pop up and see the gyro and accelerometer values updating.

Your XRP code is now running!

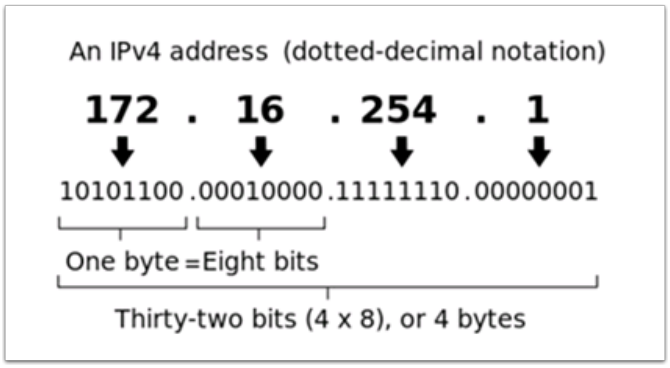
42 Networking Introduction

This section outlines basic robot configuration and usage relating to communication between the driver station and roboRIO.

42.1 Networking Basics

42.1.1 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.

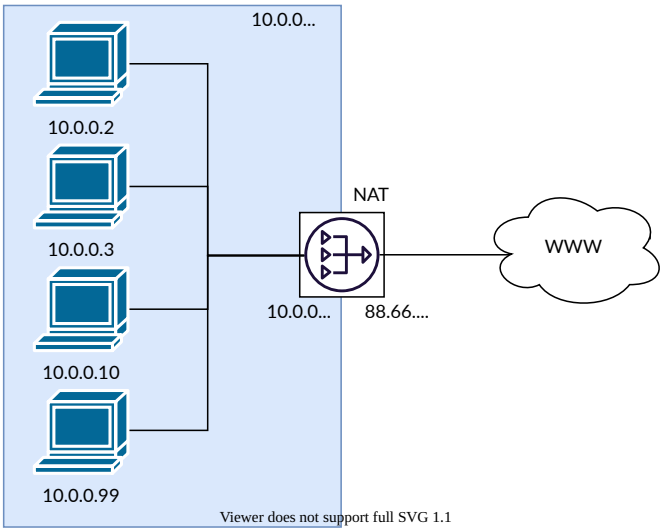
42.1.2 Public vs Private IP Addresses

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.

42.1.3 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.

42.1.4 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.

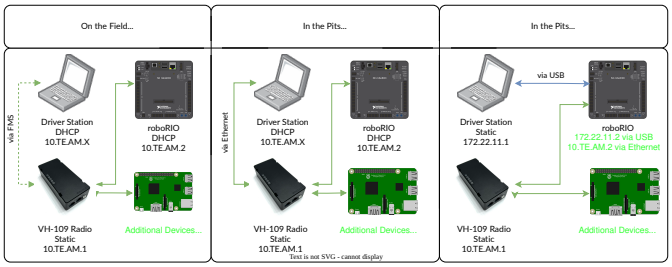
42.1.5 IP Addressing for FRC

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 VH-109 radio (on the green network shown above).

Available Network Ports

Please see R704 of the 2024 Game Manual for information regarding available network ports.

42.1.6 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 website.

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 website, 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 hostnames to IP addresses on small networks with no dedicated name server. To resolve a hostname 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.

42.1.7 USB

If using the USB interface, no network setup is required, but you do need 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.

42.1.8 Ethernet/Wireless

The *Programming your Radio* will enable the DHCP server on the VH-109 radios. Both Robot Radio (`10.TE.AM.1`) and AP Radio (`10.TE.AM.4`) will have the team-based IP address 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.

42.1.9 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.

42.2 IP Configurations

This document describes the IP configuration used at events, both on the fields and in the pits, potential issues, and workaround configurations.

42.2.1 TE.AM IP Address Notation

The notation TE.AM is used as part of IPs in numerous places. This notation refers to splitting your five digit team number into digits for the IP address octets. Where AM is the last two digits of the team number, and TE is the first three digits. Leading zeros are optional. This scheme supports team numbers up to 25599.

Example: `10.TE.AM.2`

Team 1 - `10.0.1.2`

Team 12 - `10.0.12.2`

Team 122 - `10.1.22.2`

Team 1002 - `10.10.2.2`

Team 1212 - `10.12.12.2`

Team 1202 - `10.12.2.2`

Team 1220 - `10.12.20.2`

Team 3456 - `10.34.56.2`

Team 10000 - `10.100.0.2`

Team 12345 - `10.123.45.2`

42.2.2 DHCP configuration

This section describes networking when using *DHCP* to assign IP addresses on the field, in the pits, and at home.

On the Field DHCP Configuration

Both the Field Network and Robot Radio run DHCP servers with non-overlapping pools. The Field Network DHCP server provides an IP address for the Driver Station, and the Robot Radio provides IP addresses for devices onboard the robot. The IP ranges, subnet mask, and default gateway are listed in the table below.

	Field Network	Robot Radio
DHCP Range	`10.TE.AM.20` to `10.TE.AM.199`	`10.TE.AM.200` to `10.TE.AM.219`
Subnet Mask	`255.255.255.0`	`255.255.255.0`
Default Gateway	`10.TE.AM.4`	`10.TE.AM.4` (`10.TE.AM.1` if using the OpenMesh radio)

- Vivid-Hosting VH-109 Robot 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
- Other devices (if used) - DHCP `10.TE.AM.YY` assigned by Robot Radio

In the Pits DHCP Configuration

- VH-109 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
- Other devices (if used) - DHCP, `10.TE.AM.Y`, assigned by Robot Radio

> **Note**
>
> If the roboRIO is not being assigned a DHCP IP address of `10.TE.AM.2`, ensure you've *Set the roboRIO team number*

At Home DHCP Configuration

- VH-109 radio - Static `10.TE.AM.1` programmed when configuring robot radio
- roboRIO - `10.TE.AM.2`, assigned by Robot Radio
- VH-109 access point radio - Static `10.TE.AM.4`, programmed when configuring access point radio
- Driver Station - DHCP ("Obtain an IP address automatically"), `10.TE.AM.X`, assigned by Robot Radio
- Other devices (if used) - DHCP, `10.TE.AM.Y`, assigned by Robot Radio

> **Note**
>
> If the roboRIO is not being assigned a DHCP IP address of `10.TE.AM.2`, ensure you've *Set the roboRIO team number*

42.2.3 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` for the VH-109 radio, `10.TE.AM.3` and `10.TE.AM.4` for the field network, and anything `10.TE.AM.20` or greater which may be assigned to a device configured for DHCP or else reserved. The roboRIO network configuration can be set from the webdashboard.

- VH-109 radio - Static `10.TE.AM.1` programmed when configuring robot radio
- roboRIO - Static `10.TE.AM.2` subnet mask of `255.255.255.0` (default)

- VH-109 access point radio - Static `10.TE.AM.4`, programmed when configuring access point radio
- Driver Station - Static `10.TE.AM.5` subnet mask **must** be `255.0.0.0` to enable the DS to reach both the robot and *FMS* Server, without additionally configuring the default gateway. If a static address is assigned and the subnet mask is set to `255.255.255.0`, then the default gateway must be configured to `10.TE.AM.4`.
- Other devices - Static `10.TE.AM.6-.19` subnet `255.255.255.0`

42.3 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

42.3.1 Ping the roboRIO using mDNS

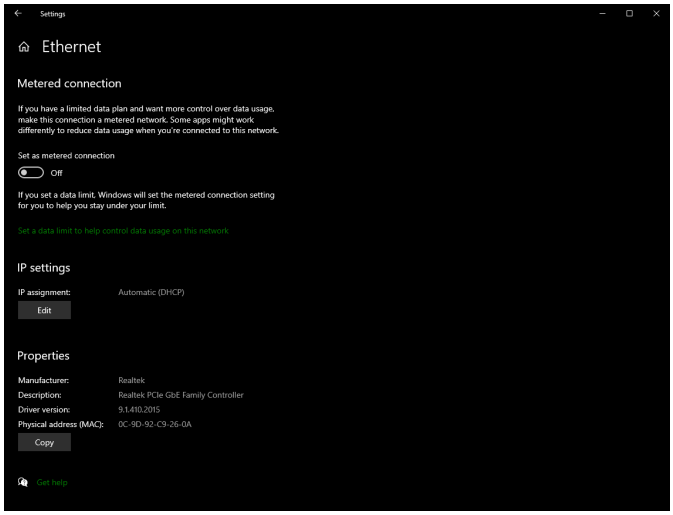
The first step to identifying roboRIO networking issues is to isolate if it is an application issue or a general network issue. To do this, click **Start -> type cmd -> press Enter** to open the command prompt. Type `ping roboRIO-#####-FRC.local` where `#####` is your team number (with no leading zeroes) 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.

42.3.2 Ping the roboRIO IP Address

If there is no response, try pinging `10.TE.AM.2` (*TE.AM 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 *Network Basics article*.
- Disconnect your computer from any other networks. Removing any other routers from the system will help verify that there is not a DNS server causing the issue.

42.3.3 Ping Fails



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.

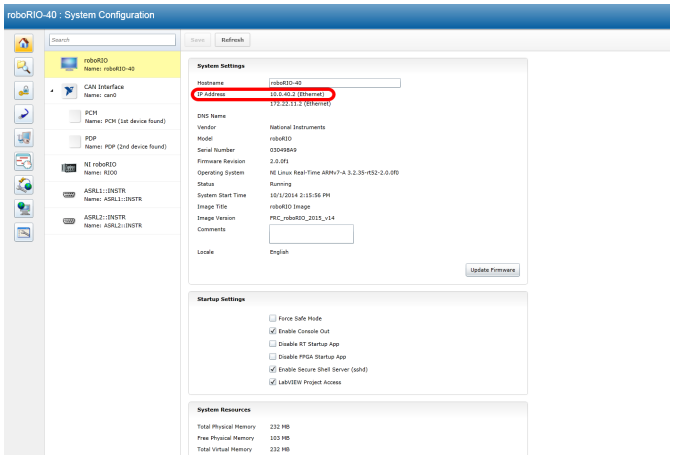
42.3.4 USB Connection Troubleshooting

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: *roboRIO 1 roboRIO 2*. This sets the hostname used by mDNS.
- *Disable all other network adapters*

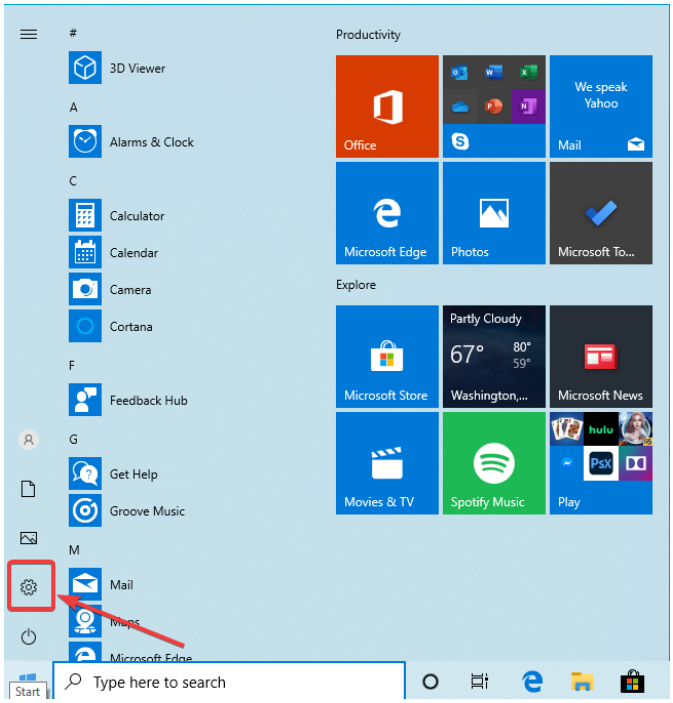
42.3.5 Ethernet Connection



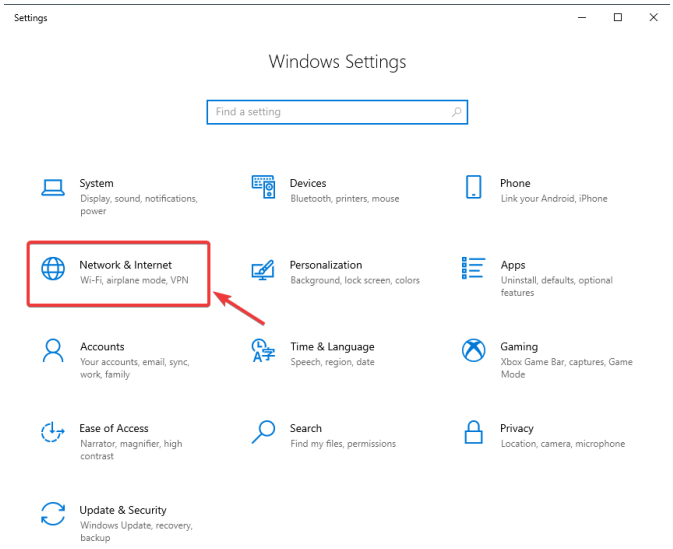
If you are troubleshooting an Ethernet connection, it may be helpful to first make sure that you can connect to the roboRIO using the USB connection. Using the USB connection, open the *roboRIO webdashboard* and verify that the roboRIO has an IP address on the ethernet interface. If you are tethering to the roboRIO directly this should be a self-assigned `169.*.*.*` address, if you are connected to the VH-109 radio, it should be `10.TE.AM.2` where TEAM is your five digit FRC team number(*TE.AM IP Notation*). If the only IP address here is the USB address, verify the physical roboRIO ethernet connection. If the IP address is `10.TE.AM.XX`, ensure you've *set the roboRIO team number*

42.3.6 Disabling Network Adapters

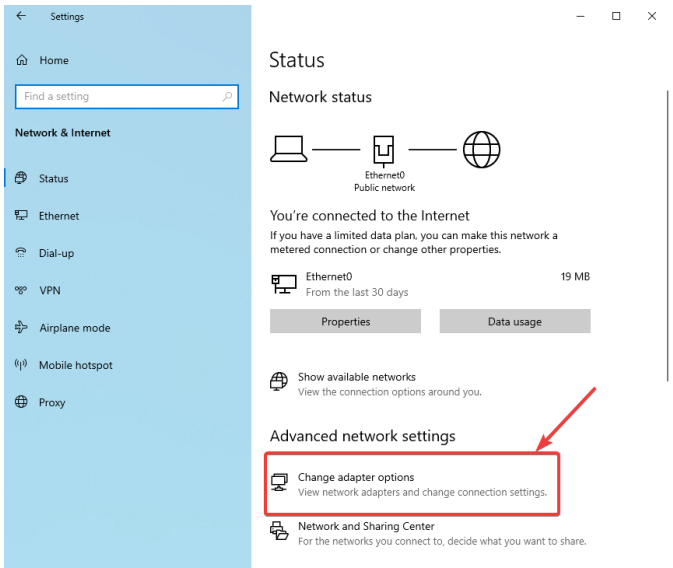
This is not always the same as turning the adapters off with a physical button or putting the PC into airplane mode. The following steps provide more detail on how to disable adapters.



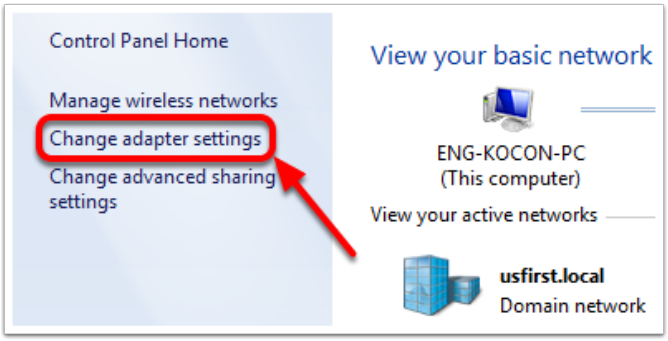
Open the Settings application by clicking on the settings icon.



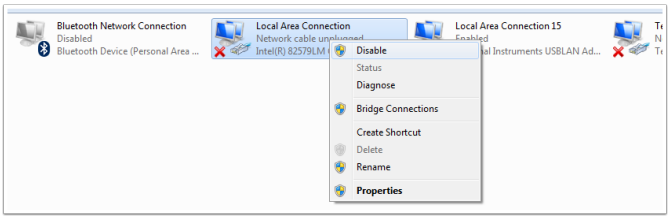
Choose the *Network & Internet* category.



Click on *Change adapter options*.



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.

42.3.7 Proxies

- Proxies. Having a proxy enabled may cause issues with the roboRIO networking.

42.4 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.

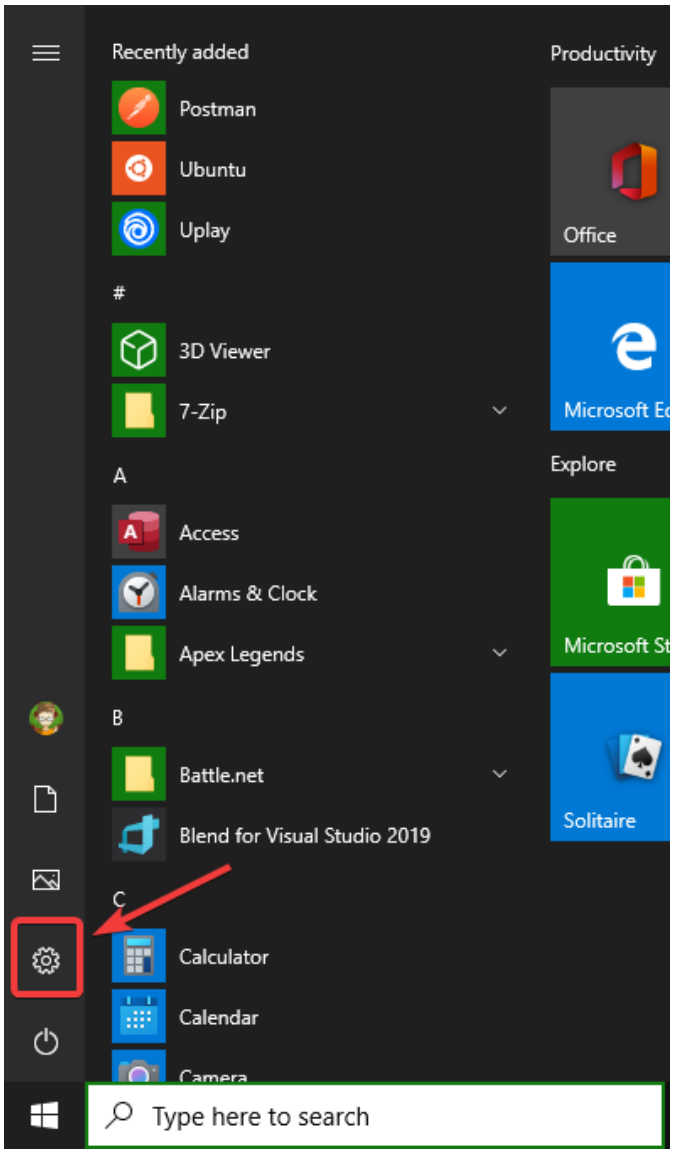
42.4.1 Disabling Windows Firewall

Important

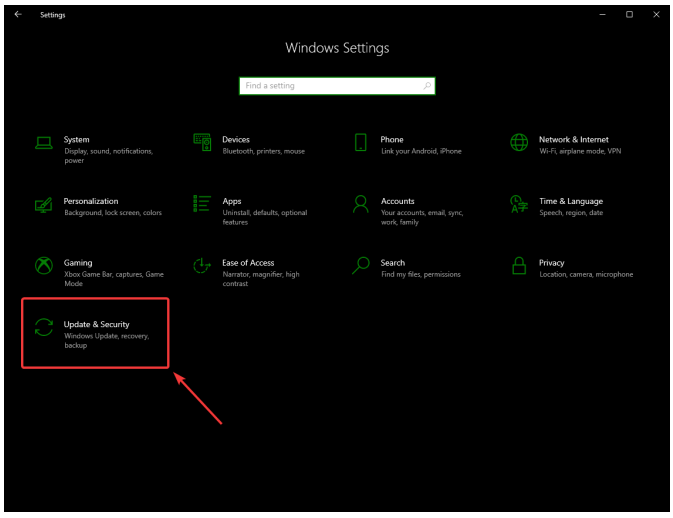
Disabling your firewall requires administrator privileges to the PC. Additionally note that disabling the firewall is not recommended for computers that connect to the internet.

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.

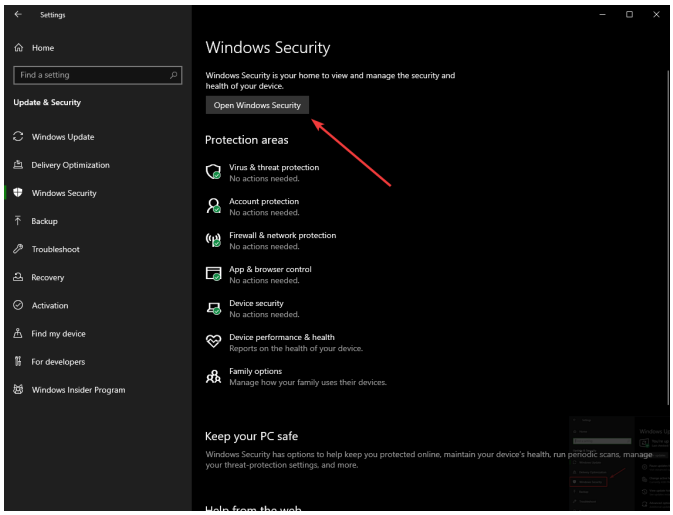
Click *Start -> Settings*



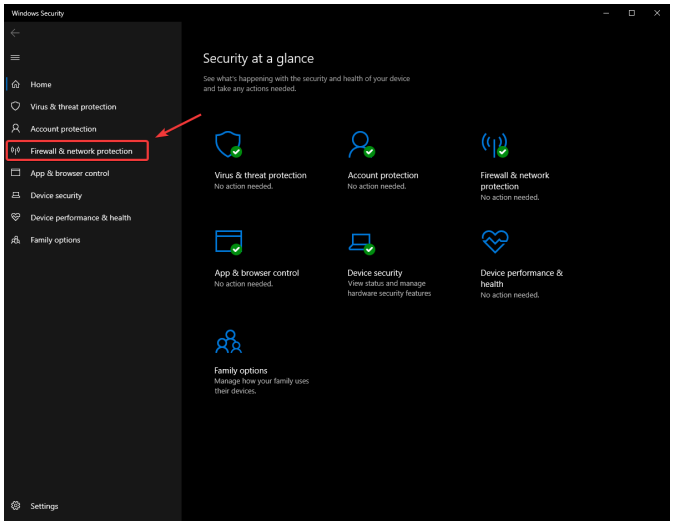
Click *Update & Security*



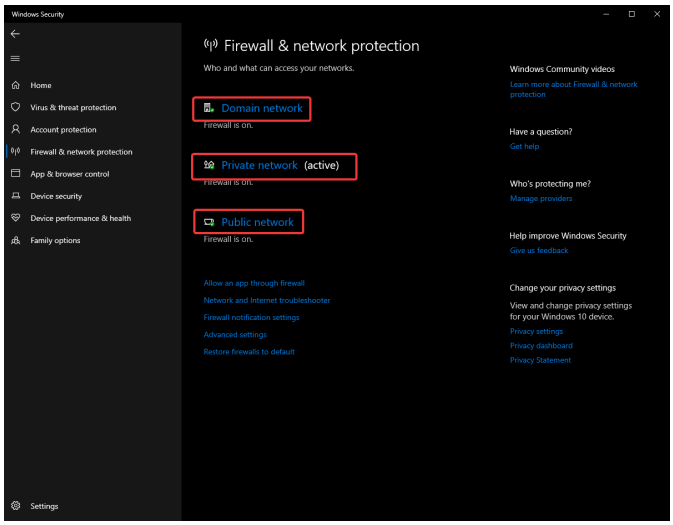
In the right pane, select *Open Windows Security*



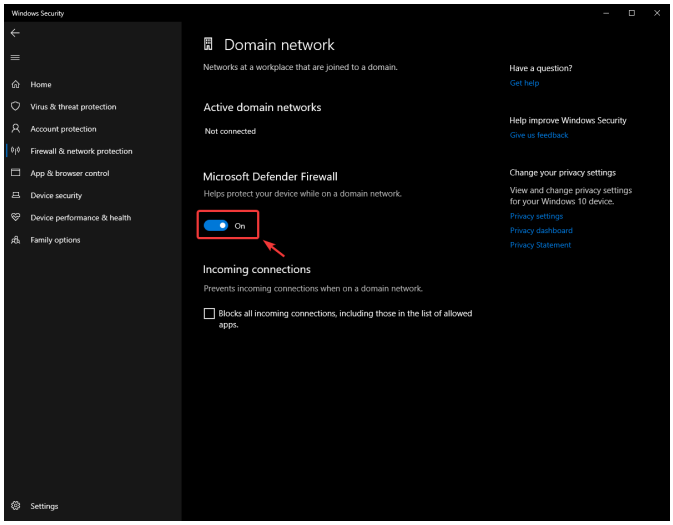
In the left pane, select *Firewall and network protection*



Click on **each** of the highlighted options



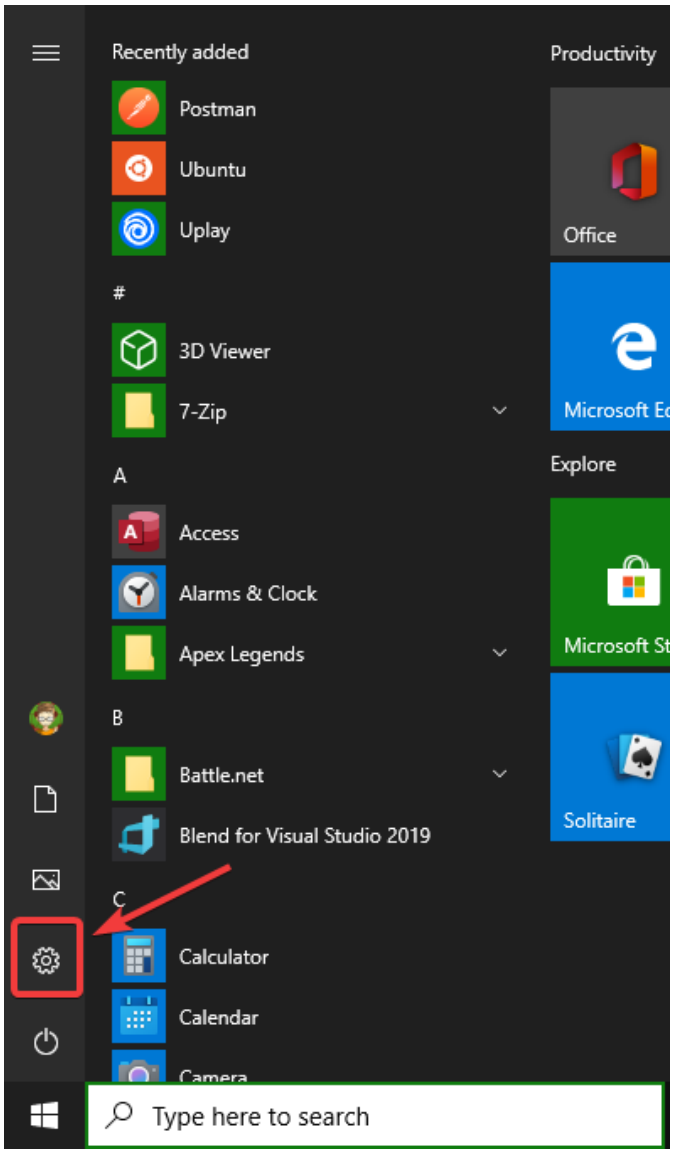
Then click on the **On** toggle to turn it off.



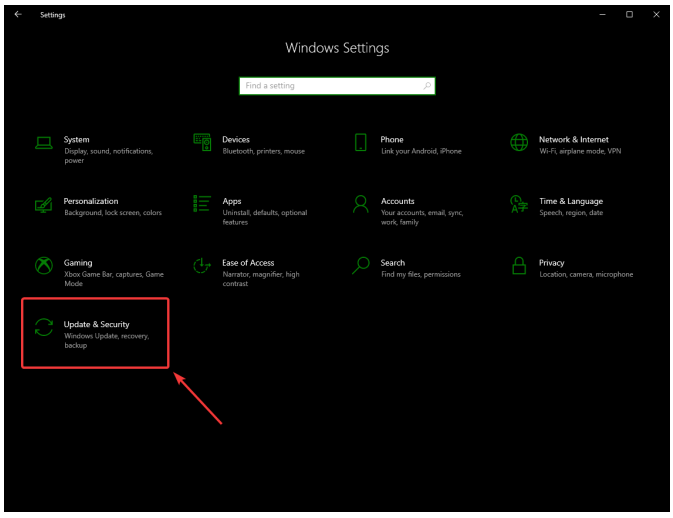
42.4.2 Whitelisting Apps

Alternatively, you can add exceptions to the Firewall for any FRC programs you are having issues with.

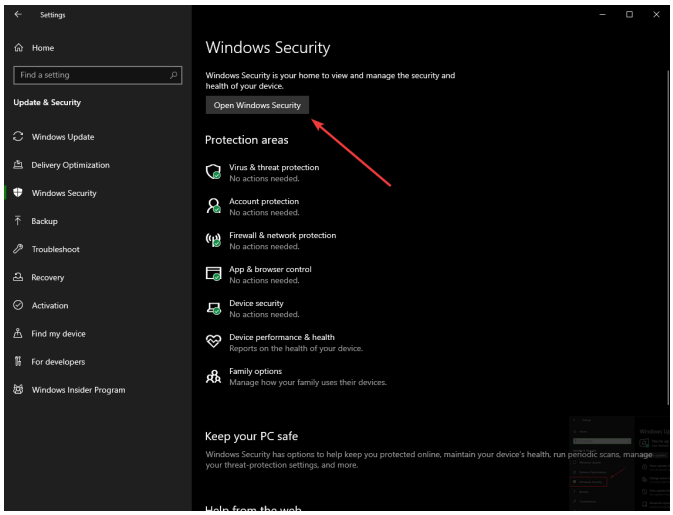
Click *Start -> Settings*



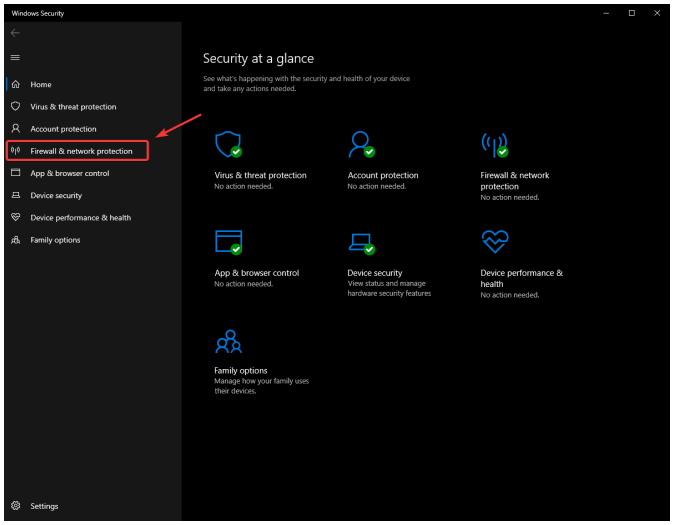
Click *Update & Security*



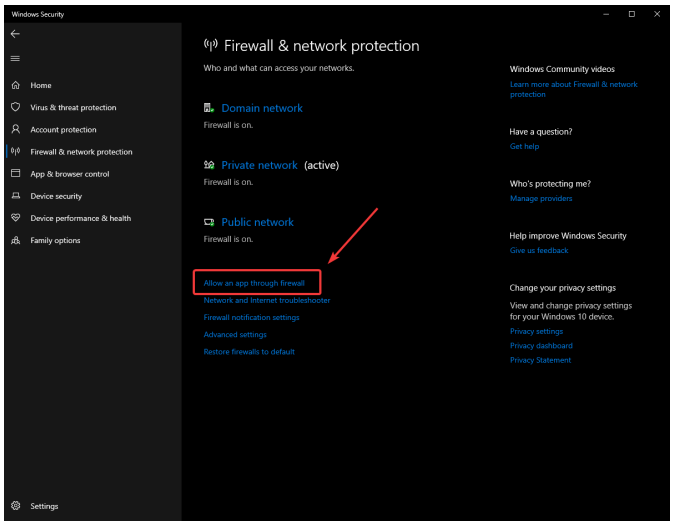
In the right pane, select *Open Windows Security*



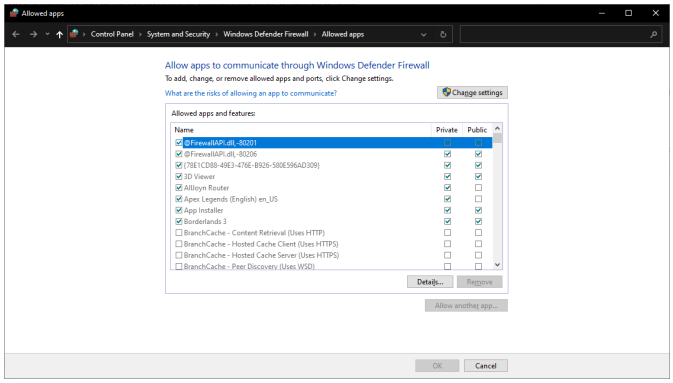
In the left pane, select *Firewall and network protection*



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.



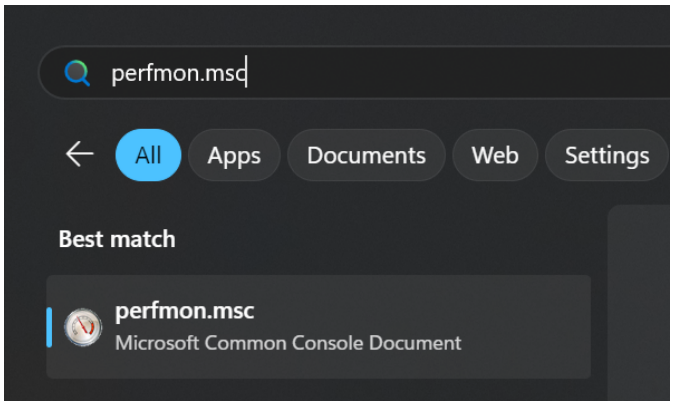
42.5 Measuring Bandwidth Usage

On the FRC® Field each team is allocated limited network bandwidth (see R704 in the 2026 manual). Some teams may wish to measure their overall bandwidth consumption. This document details how to make that measurement.

42.5.1 Measuring Bandwidth Using the Performance Monitor (Win 7/10)

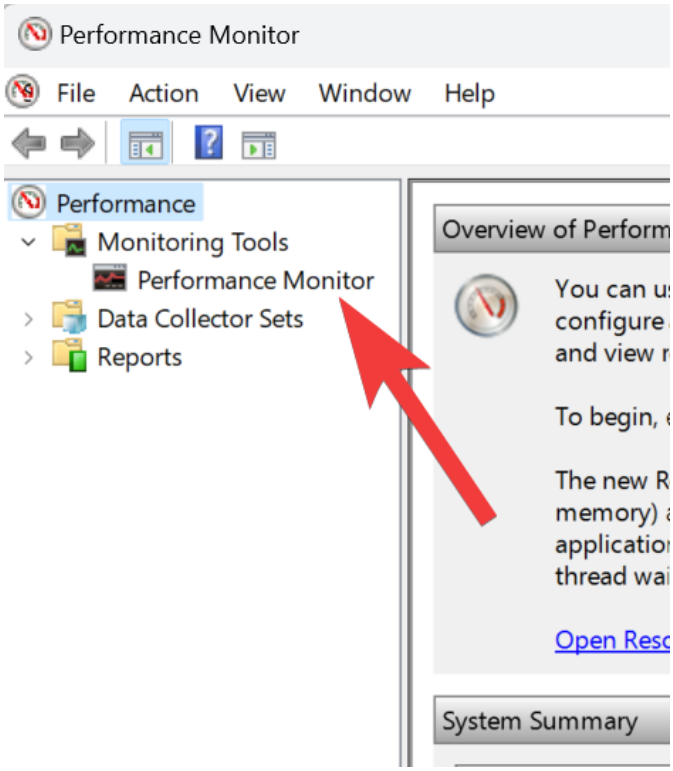
Windows contains a built-in tool called the Performance Monitor that can be used to monitor the bandwidth usage over a network interface.

Launching the Performance Monitor



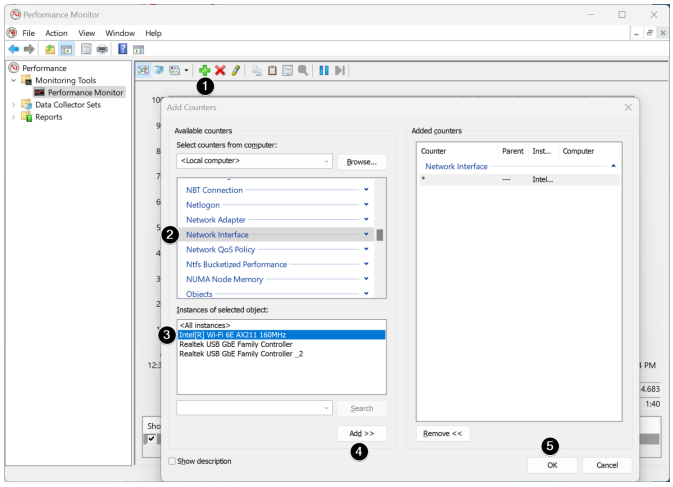
Open the Start menu and in the search box, type `perfmon.msc` and press Enter.

Open Real-Time Monitor



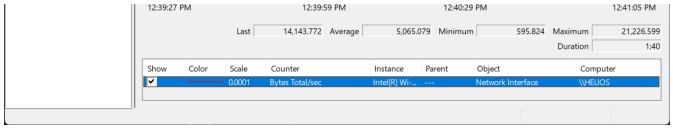
In the left pane, click *Performance Monitor* to display the real-time monitor.

Add Network Counter



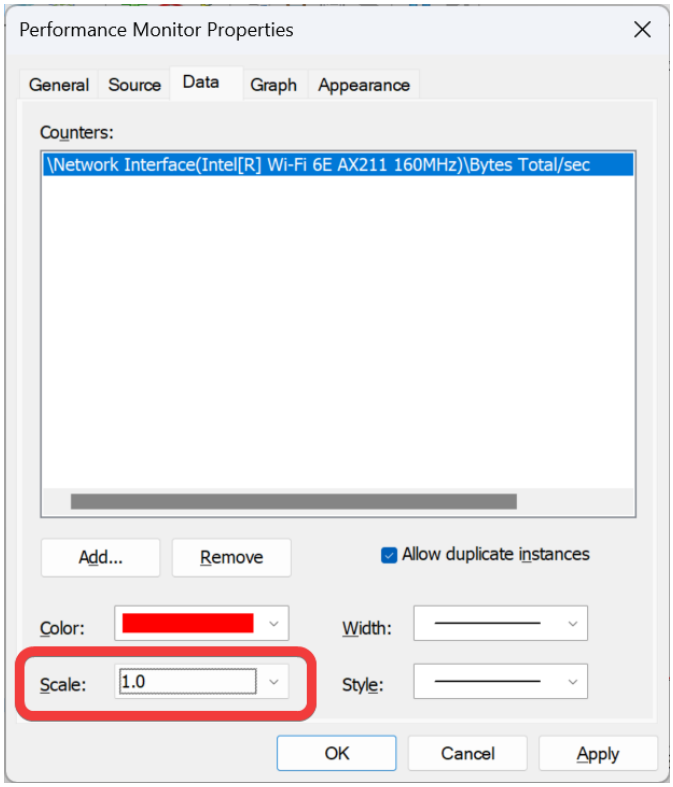
1. Click the green plus near the top of the screen to add a counter
2. In the top left pane, locate and click on `Network Interface` to select it
3. In the bottom left pane, locate the desired network interface (or use All instances to monitor all interfaces)
4. Click *Add >>* to add the counter to the right pane.
5. Click *OK* to add the counters to the graph.

Remove Extra Counters



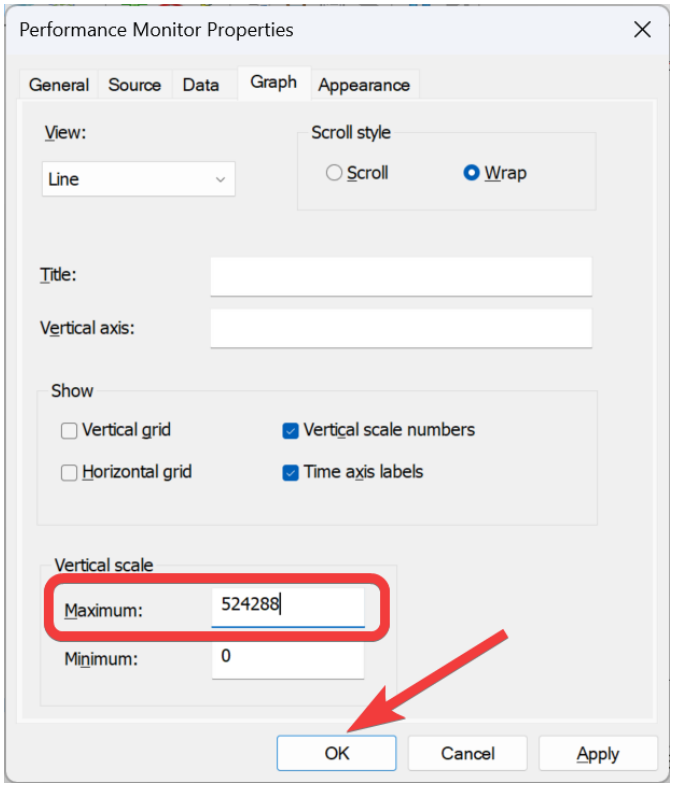
In the bottom pane, select each counter other than `Bytes Total/sec` and press the `Delete` key. The `Bytes Total/sec` entry should be the only entry remaining in the pane.

Configure Data Properties



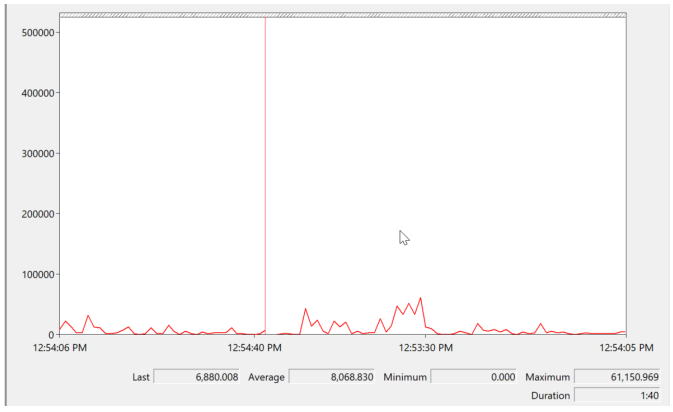
Press `Ctrl+Q` to bring up the Properties window. Click on the dropdown next to `Scale` and select `1.0`. Then click on the *Graph* tab.

Configure Graph Properties



In the `Maximum Box` under `Vertical Scale` enter 524288 (this is 4 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

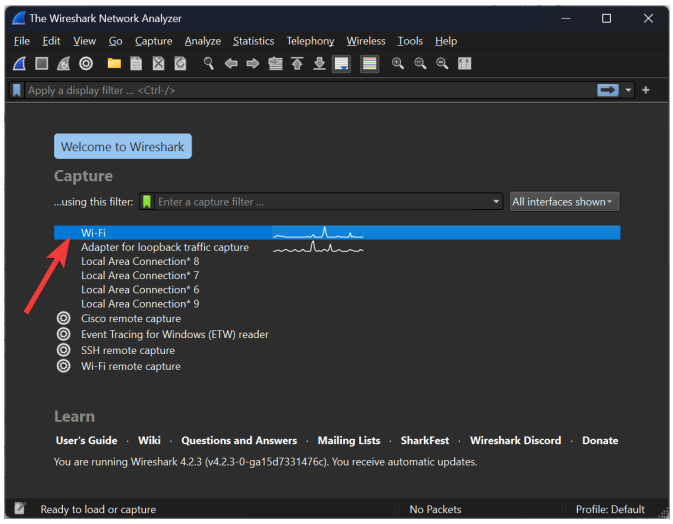


You may now connect to your robot as normal over the selected interface (if you haven't done so already). The graph will show the total bandwidth usage of the connection, with the bandwidth cap at the top of the graph. he Last, Average, Min, and Max values are also displayed at the bottom of the graph. Note that these values are in Bytes/Second meaning the cap is 524,288. With just the Driver Station open you should see a flat line at ~100000 Bytes/Second.

42.5.2 Measuring Bandwidth Usage using Wireshark

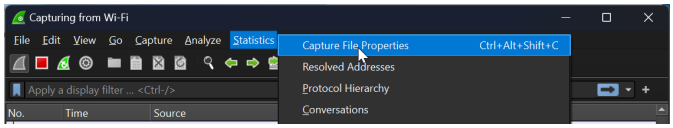
If you cannot use Performance Monitor, you will need to install a 3rd party program to monitor bandwidth usage. One program that can be used for this purpose is Wireshark. Download and install the latest version of Wireshark for your version of Windows from this page. After installation is complete, 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



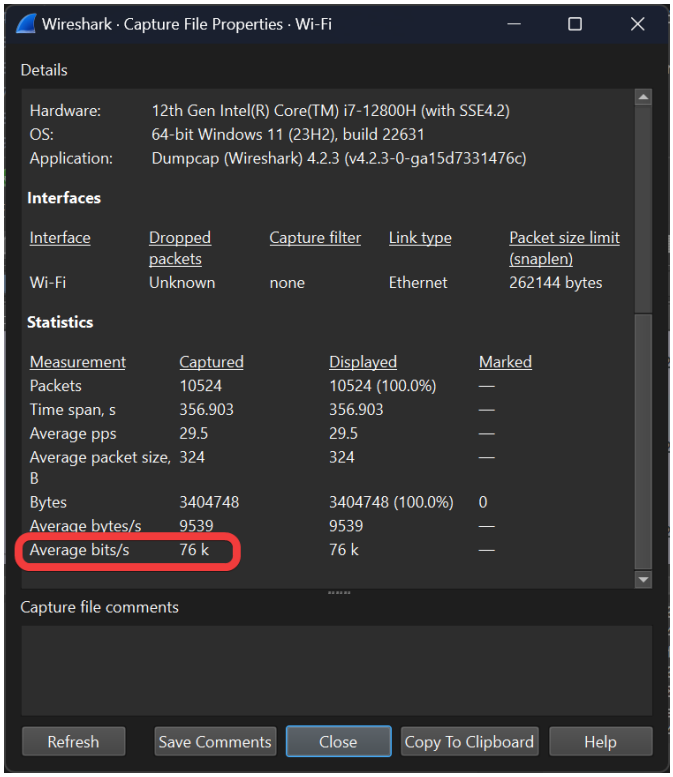
In the Wireshark program on the left side, double click the interface you are using to connect to the robot.

Open Capture File Properties



Let the capture run for at least 1 minute, then click *Statistics* then *Capture File Properties*.

View Bandwidth Usage



Average bandwidth usage, in bits/second is displayed near the bottom of the window.

43 Networking Utilities

43.1 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.

43.1.1 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

```
public Robot() {
   PortForwarder.add(8888, "wpilibpi.local", 80);
}
```

C++

```
Robot::Robot() {
   wpi::PortForwarder::GetInstance().Add(8888, "wpilibpi.local", 80);
}
```

PYTHON

```
wpinet.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.

43.1.2 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

```
public Robot() {
   PortForwarder.remove(8888);
}
```

C++

```
Robot::Robot() {
   wpi::PortForwarder::GetInstance().Remove(8888);
}
```

PYTHON

```
wpinet.PortForwarder.getInstance().remove(8888)
```

44 Contributing to frc-docs

44.1 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.

44.1.1 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.

44.1.2 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

44.1.3 Creating a PR

PRs should be made to the frc-docs repo on GitHub. They should point to the `main` branch and *not* `stable`.

44.1.4 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).

44.2 Style Guide

This document contains the various MD and RST/Sphinx specific guidelines for the frc-docs project. For guidelines related to the various WPILib code projects, see the WPILib GitHub

44.2.1 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 VS Code with the reStructuredText extension.

44.2.2 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```.

44.2.3 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 line 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.

44.2.4 Headings

Headings should be in the following structure.

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.

44.2.5 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
```

44.2.6 Code blocks

All code blocks should have a language specified.

````
```python
print("Hello!")

import antigravity
```
````

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.

44.2.7 RLI (Remote Literal Include)

When possible, instead of using code blocks, an RLI should be used. This pulls code lines directly from GitHub, most commonly using the example programs. This automatically keeps the code up to date with any changes that are made. The format of an RLI is:

```
.. remoteliteralinclude:: https://raw.githubusercontent.com/wpilibsuite/allwpilib/
↪v2026.2.1/wpilibjExamples/src/main/java/edu/wpi/first/wpilibj/examples/
↪gettingstarted/Robot.java
   :language: java
   :lines: 19-25
   :lineno-match:

.. remoteliteralinclude:: https://raw.githubusercontent.com/wpilibsuite/allwpilib/
↪v2026.2.1/wpilibcExamples/src/main/cpp/examples/GettingStarted/cpp/Robot.cpp
   :language: c++
   :lines: 50-60
   :lineno-match:

.. remoteliteralinclude:: https://raw.githubusercontent.com/robotpy/examples/
↪242924b3843fdcc6efc2cefa8eac7bfff8b6bc48/GettingStarted/robot.py
   :language: python
   :lines: 12-29
   :lineno-match:
```

Make sure to link to the raw version of the file on GitHub. There is a handy Raw button in the top right corner of the page.

Note

RLIs should use a tag instead of main to ensure the documentation isn't broken the next time there is a change to the RLIed code. If a tag hasn't been created, use the full (40 character) commit hash.

44.2.8 Tabs

To create code tabs in an article, you can use the `.. tab-set-code::` directive. You can use md style ``` codeblocks and `rli` directives inside. The format is:

```
.. tab-set-code::
   .. remoteliteralinclude:: https://raw.githubusercontent.com/wpilibsuite/allwpilib/
↪v2026.2.1/wpilibjExamples/src/main/java/edu/wpi/first/wpilibj/examples/
↪gettingstarted/Robot.java
      :language: java
      :lines: 19-25
      :lineno-match:

   ```c++
   // Start the timer.
   m_timer.Start();
   // Send Field2d to SmartDashboard.
   frc::SmartDashboard::PutData(&m_field);
   // Reset the drivetrain's odometry to the starting pose of the trajectory.
   m_drive.ResetOdometry(m_trajectory.InitialPose());
```

(continues on next page)
```

(continued from previous page)

```
   // Send our generated trajectory to Field2d.
   m_field.GetObject("traj")->SetTrajectory(m_trajectory);
   ```
```

If you need to use more than one tab per language, multiple RLIs per language, or text tabs, you can use the `.. tab-set::` and `.. tab-item::` directive. The format is:

```
.. tab-set::
   .. tab-item:: Title
      :sync: sync-id
         Content
```

This example uses the sync argument to allow all of the tabs with the same key to be synced together. This means that when you click on a tab, all of the tabs with the same key will open.

If you have a mix of `tab-set` and `tab-set-code` directives on a page, you can sync them by setting the sync id on the `tab-item` directives to `tabcode-LANGUAGE`. For example, a java tab would have a sync id of `tabcode-java`.

44.2.9 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 an 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!
```

44.2.10 Links

Internal Links

Internal Links will be auto-generated based on the ReStructuredText filename and section title.

For example, here are several ways to link to sections and documents.

Use this format to reference a document section. You must use the absolute path of the document. `:ref:`docs/software/hardware-apis/sensors/ultrasonics-software:Analog ultrasonics`` renders to *Analog ultrasonics*.

Use this format to reference a section of the same document. Note the single underscore. `` `Images`_ `` renders to *Images*.

Use this format to reference the top-level of a document. You can use relative paths `:doc:`build-instructions`` renders to *Build Instructions* Or to use absolute paths, put a
```

forward slash at the beginning of the path `:doc:`/docs/software/hardware-apis/sensors/ultrasonics-software`` renders to *Ultrasonics - Software*. Note that the text rendered is the main section title of the target page regardless of the target filename.

When using `:ref:` or `:doc:` you may customize the displayed text by surrounding the actual link with angle brackets `<>` and adding the custom text between the first backtick `` ` `` and the first angle bracket `<`. For example `:ref:`custom text <docs/software/hardware-apis/sensors/ultrasonics-software:Analog ultrasonics>`` renders to *custom text*.

External Links

It is preferred to format external links as md style hyperlinks.

```
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
```

Python API Links

Links to the RobotPY API documentation should use the following sphinx python syntax (example linking to the DriverStation API docs).

```
:py:class:`Python <robotpy:wpilib.DriverStation>`
```

This expands to be equivalent to:

```
[Python](https://robotpy.readthedocs.io/projects/robotpy/en/stable/wpilib/
↪DriverStation.html#wpilib.DriverStation)
```

44.2.11 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.

44.2.12 File Extensions

File extensions should use code formatting. For example, use:

```
``.png``
```

instead of:

```
.png
".png"
"``.png``"
```

44.2.13 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`.

44.2.14 Examples

```
# Title
This is an example article
```java
System.out.println("Hello World");
```
Section
This is a section!
```

44.2.15 Important Note!

This list is not exhaustive and administrators reserve the right to make changes. Changes will be reflected in this document.

44.3 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.

44.3.1 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`.

You will need to install uv, a Python package and project manager. Follow the installation instructions for your operating system.

Text Editors / IDE

For development, we recommend that you use VS Code along with the reStructuredText extension. However, any text editor will work.

By default, the reStructuredText extension enables linting with all doc8 features enabled. As frc-docs does not follow the line length lint, add the following to your VS Code `settings.json` to disable line length linting.

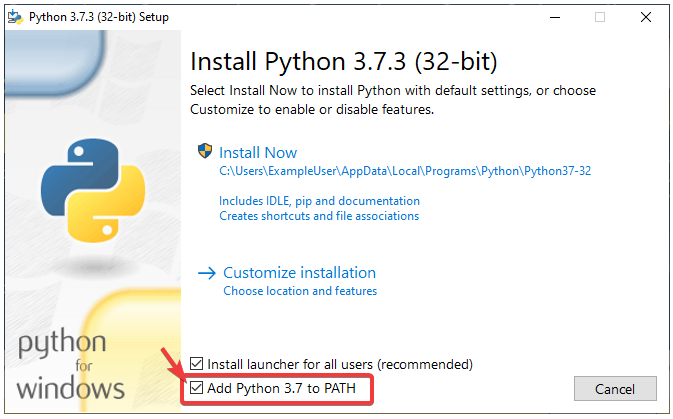
```
"restructuredtext.linter.doc8.extraArgs": [
   "--ignore D001"
]
```

Windows

> **Note**
>
> MikTeX and `rsvg-convert` are not required for building HTML, they are only required for Windows PDF builds.

- Python 3.10 to 3.13
- Perl
- MiKTeX (Only needed for PDF builds)
- rsvg-convert (Only needed for PDF builds)

Ensure that Python is in your Path by selecting the **Add Python to PATH** toggle when installing Python.



Install the missing MikTex packages by navigating to the frc-docs directory, then running the following command from Powershell: `miktex --verbose packages require --package-id-file miktex-packages.txt`

Linux (Ubuntu)

```
$ sudo apt update
$ sudo apt install python3
$ 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
```

44.3.2 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 CI if it does not pass. To check, run `uv run .\make.bat lint`

Link Check

The link checker makes sure that all links in the documentation resolve. This **will** fail CI if it does not pass. To check, run `uv run .\make.bat linkcheck`

Image Size Check

Please run `uv run .\make.bat 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 `uv run .\make.bat 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 CI if it does not pass. To check, run `uv run .\make.bat 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 `uv run .\make.bat html` to generate HTML content. The content is located in the `build/html` directory at the root of the repository.

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 `uv run .\make.bat latexpdf` to generate PDF content. The PDF is located in the `build/latex` directory at the root of the repository.

Building EPUB

Type the command `uv run .\make.bat epub` to generate EPUB content. The EPUB is located in the `build/epub` directory at the root of the repository.

44.3.3 Adding Python Third-Party libraries

> **Important**
>
> After modifying frc-docs dependencies in any way, `requirements.txt` must be regenerated by running `uv export --frozen --no-dev --no-editable -o source/requirements.txt` from the root of the repo.

Adding a Dependency

Add the dependency to the `dependencies` array in `pyproject.toml`. Make sure to specify an exact version. Then, run the following command: `uv lock`.

Updating a Top-Level Dependency

Update the dependency's version in the `dependencies` array in `pyproject.toml`. Then, run the following command: `uv lock`.

Updating All Hidden Dependencies

Run the following command: `uv lock --upgrade`.

Updating a Specific Hidden Dependency

Run the following command: `uv lock --upgrade-package <package>`.

44.4 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.

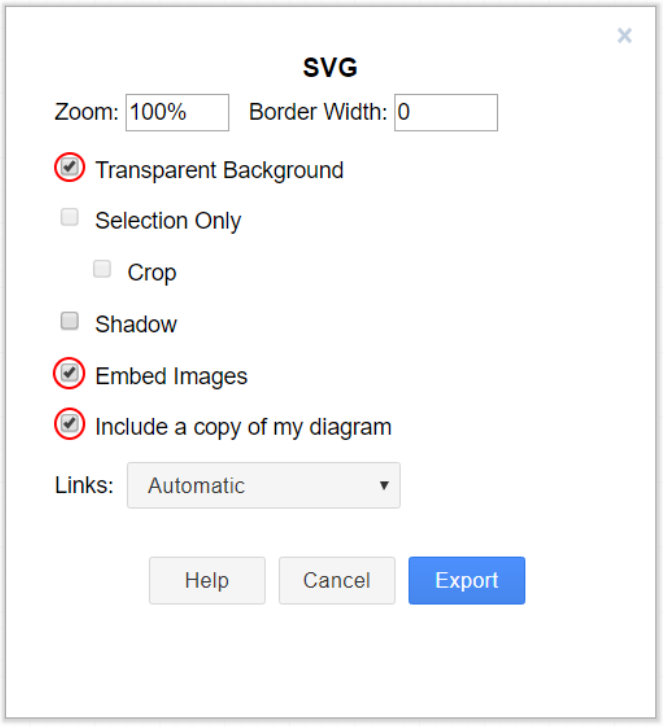
Draw.io (also known as diagrams.net) are supported when saved as `svg` files, with embedded XML metadata for the draw.io source file (normally stored as `.drawio`). This allows these images to both act as source files for the diagrams that can be edited in the future, and be rendered as normal `svg` 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*

44.5 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.

44.5.1 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!

44.5.2 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-simple-robot:How to Wire an FRC Robot>`
```

44.5.3 Publishing Translations

Translations are pulled from Transifex and published automatically each day.

44.5.4 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.

44.6 Top Contributors

- sciencewhiz

  760 contributions
- Daltz333

  327 contributions
- jasondaming

  145 contributions
- Oblarg

  117 contributions
- prateekma

  85 contributions

- Starlight220
  57 contributions
- Kevin-OConnor
  55 contributions
- TheTripleV
  53 contributions
- rzblue
  47 contributions
- calcmogul
  47 contributions

44.7 Top Translators

44.7.1 Chinese

- 8192 Dhc
- Atlus Zhang
- Jiangshan Gong
- Keseterg
- Michael Zhao
- Nathan Zhou
- Ningxi Huang
- Ran Xin
- Shawphy
- Team 5308
- Tianrui Wu
- Tianshuang Zhang
- Xun Sun
- Yitong Zhao
- Yuhao Li
- □□ □
- □□ □
- □□ □
- □□ □
- □□ □

44.7.2 French

- Alexandra Schneider
- Andre Theberge
- Andy Chang
- Austin Shalit
- Dalton Smith
- Daniel Renaud
- Étienne Beaulac
- Félix Giffard
- Kaitlyn Kenwell
- Laura Luna Bedard
- Marc Lalonde
- Martin Regimbald
- Martin Rioux
- Regis Bekale
- Sami G.-D.
- Sidney Lavoie
- Youdlain Marcellus

44.7.3 Portuguese

- Amanda Carolina Wilmsen
- Bibiana Oliveira
- Bruno Osio
- Bruno Toso
- Gabriel Silveira
- Gabriela Tomaz Do Amaral Ribeiro
- Luca Carvalho
- Lucas Fontes Francisco
- Maria Eduarda Grabin Gisse
- Matheus Heitor Timm Chanan
- Miguel Ramos
- Miguel S. Ramos
- Natan Feijó Tristão
- Nathany Santiago
- Pedro Braga

- Pedro Henrique Dias Pellicioli
- Rodrigo Anholon Novo
- Tales Dias De Almeida Silva
- Vinícius Castro
- Vinícius Gabriel Da Silva

44.7.4 Spanish

- Austin Shalit
- Cesar Ernesto
- Diana Ramos
- Diego Lozano Rangel
- Fernanda Reveles
- Fernando Soltero
- Gibrán Verástegui
- Heber Sepúlveda
- Heriberto Gutierrez
- Hugo Espino
- Isabella Rodriguez
- Mariano
- Miguel Angel De León Adame
- Óscar Ariel Gutiérrez
- Paulina Maynez
- Pierre Cote
- Rodrigo Acosta
- Román Hernandez Sosa
- Sofia Fernandez
- Zara Moreno

44.7.5 Turkish

- Hasan Bilgin
- Müfit Alkaya
- Deniz Şentürk
- Esra Özemre
- Ceren Oktemer
- Demet T

- Demet Tumkaya
- Lal Serdaroğlu
- Melis Aldeniz
- Çağan Uslu
- Duru Ünlü
- Arhan Ünay
- Ada Zagyapan
- Doruk Akdoğan
- Müfit Alkaya_3390
- Tuna Özer
- Mayra Şengel
- Elif Akın
- Duru Hatipoğlu
- Nesrin Serra Köşkeroğlu

44.7.6 Hebrew

- Aric Radzin
- Dalton Smith
- Itay Ziv
- Ofek Ashery
- Shai Grossman
- Starlight220
- Yotam Shlomi

45 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.

45.1 Quick Start

Below is a list of instructions that guide you through cloning, building, publishing, and using local allwpilib binaries in a robot project. This quick start is not intended as a replacement for the information that is further listed in this document.

- Clone the repository with `git clone https://github.com/wpilibsuite/allwpilib.git`
- Build the repository with `./gradlew build` or `./gradlew build --build-cache` if you have an internet connection
- Publish the artifacts locally by running `./gradlew publish`
- Update your robot project's `build.gradle` to use the artifacts

45.2 Core Repository

45.3 NetworkTables

46 Privacy Policy

WPILib is a project developed by volunteers who are largely based in the United States. We take privacy seriously and collect only the minimum amount of information needed to support our mission of providing user-friendly libraries and software tools to enable FIRST Robotics Competition teams to be successful in writing software for their robots.

46.1 Information We Collect

WPILib-developed desktop applications do not collect personal information. They do create and store logs of application errors and user interactions on the user's system, but this data is not shared online unless the user takes action to upload or share it.

The WPILib libraries do not collect personal information. When run on the roboRIO, the libraries collect limited metrics data ("usage reporting") regarding what pieces of the libraries are used by the deployed software (user-written code linked to the WPILib libraries). Only when the robot code is running and connected to the competition field at a FIRST Robotics Competition event is this information shared with and stored by the Field Management System. This information is associated solely with the FRC team and not with any individual. An anonymized (team number removed) version of this metrics data is publicly shared by FIRST after each competition season, and is used to inform future development of library features.

FIRST Privacy Policy.

46.2 Third-Party Data Collection

WPILib uses and installs a number of third-party applications, each of which has their own privacy policy. We also use a variety of third-party services for distribution of WPILib software, each of which has their own privacy policy, detailed below.

46.2.1 GitHub

The WPILib source code repositories are hosted by GitHub. The WPILib Visual Studio Code extension also connects to GitHub for online browsing of vendor dependency updates.

GitHub Privacy Policy

46.2.2 ReadTheDocs (docs.wpilib.org)

Our online documentation site is hosted by ReadTheDocs.

ReadTheDocs Privacy Policy

46.2.3 CloudFlare

WPILib uses Cloudflare for distribution of the WPILib installer.

Cloudflare Privacy Policy

46.2.4 JFrog Artifactory

WPILib uses a JFrog-hosted Artifactory instance for distribution of WPILib software components. If a build is performed while connected to the Internet, Artifactory may be contacted to check for and download updates.

JFrog Privacy Policy

46.2.5 Gradle

WPILib uses Gradle to build robot projects. If a build is performed while connected to the Internet, Gradle may check for updates to software components and will upload information about the build to its services.

Gradle Privacy Policy

46.2.6 Visual Studio Code

Visual Studio Code is a desktop application that is separately distributed by Microsoft. The WPILib installer prompts to download and install Visual Studio Code as part of the installation process. Visual Studio Code auto-updates are disabled by the WPILib installer, but crash reporting and usage reporting are still enabled by default.

Visual Studio Code Privacy Policy

46.2.7 AdvantageScope

The AdvantageScope desktop application is distributed and installed with the WPILib installer.
AdvantageScope Privacy Policy

46.2.8 Elastic

The Elastic desktop application is distributed and installed with the WPILib installer.
Elastic Privacy Policy

46.3 Contact

Questions about this policy can be directed to wpilib@wpi.edu.

Index

A

accelerometer, **613**
AM, **613**
AprilTags, **613**
auto, **613**

B

back-EMF, **613**
bang-bang control, **1264**
boolean, **613**

C

C++, **614**
CAD, **613**
call stack, **613**
CAM, **613**
CAN, **613**
Cartesian coordinate system, **1264**
CD, **614**
central limit theorem, **614**
churning losses, **1264**
CIM, **614**
Classical Mechanics, **614**
composition, **614**
control effort, **1265**
control law, **1265**
control signal, **1265**
controller, **1265**
convolution, **1265**
COTS, **614**
counter-electromotive force, **1265**
CRTP, **614**
CSA, **614**
CTRE, **614**
current, **1265**

D

declarative programming, **614**
dependency injection, **614**
deprecated, **614**
derivative, **1265**
design pattern, **615**
DHCP, **615**
dynamics, **1265**

E

encapsulation, **615**
entry, **615**
enumeration, **615**
EPA, **615**
error, **1265**
event-driven programming, **615**
exponential search, **1265**
exponential smoothing, **1265**

F

FIRST, **615**
FLL, **615**
floating point, **615**
FMS, **615**
FPGA, **615**
FRC, **616**
FTA, **616**
FTC, **616**

G

gain, **1265**
Gaussian distribution, **1265**
GDC, **616**
GP, **616**
gradient, **1266**
GradleRIO, **616**
gyroscope, **616**

H

heading, **616**
hidden state, **1266**

I

I2C, **616**
imperative programming, **616**
IMU, **616**
input, **1266**

J

Java, **616**
JSON, **616**

K

KOP, 616
KOP chassis, 617

L

LabVIEW, 617
least-squares regression, 1266
LED, 617
LQR, 1266

M

mass, 617
measurement, 1266
model, 1266
moment of inertia, 617
mutable, 617
MXP, 617

N

NetworkTables, 617
no-op, 617

O

observer, 1266
odometry, 617
OPR, 617
orthogonal, 1266
output, 1266

P

PCM, 617
PDH, 617
PDP, 617
permanent-magnet DC motor, 618
persistent, 618
PH, 618
phase portrait, 1267
PID, 1267
plant, 1267
PoE, 618
pose, 618
pose estimation, 618
process variable, 1267
property, 618
publisher, 618
PWM, 618
Python, 618
Python Enhancement Proposals
    PEP 600, 54

R

r-squared, 1267
RAII, 618
recursive composition, 619
reference, 1267
retained, 619
retro-reflection, 618
REV, 619
rise time, 1267
RMSE, 1267
RPM, 619
RSL, 619

S

serialized, 619
setpoint, 1267
settling time, 1267
signum function, 1267
simulation, 619
software library, 619
solenoid valve, 619
SPI, 619
state, 1267
state machine, 619
statistically robust, 1268
steady-state error, 1268
step input, 1268
step response, 1268
subscriber, 620
system, 1268
system identification, 1268
system response, 1268

T

TBA, 620
telemetry, 620
teleop, 620
topic, 620
torque, 620
trajectory, 620
transitory, 620

V

viscous drag, 1268
voltage, 1268
VRM, 620

W

WCP, 620
WFA, 620

X

x-dot, 1268
x-hat, 1269