
The FRC Programming Bible



V. 2021

Table of Contents

Preface	2
Resources	3
FRC Programming Tools	4
Physical Components of the Robot	5
Managing Motor Controllers	11
Maintenance	16
Source Control using Git	28
WPILib	15
- Working With Robot Projects	35
- Command-Based Programming	11
- Troubleshooting	13
Robot Code Best Practices	59
The Dashboard	61
Basic Robot Control	77
Autonomous Functions	87
- PID Loops	87
- Vision	105
- Autonomous Driving	111
- Designing an Autonomous Routine	112
- Tips / Best Practices	119
Tips for Build Season	120
Tips for Competition	124
Sign-Off	128
Glossary	129

Preface

Greetings FRC Programmers! This book was created to guide you through the sometimes confusing world of programming an FRC robot. It includes all knowledge needed to program a functional competition-ready robot with only a simple understanding of the Java programming language. However, do not rely only on this book to guide you through FRC robot programming. I recommend looking through past robot projects (which can be found in the Resources chapter) to see robot code that has successfully been used in the past. I especially recommend doing this when learning about programming autonomous functions, which are complex and usually differ from year to year.

Do not let the contents of the book or the code in the past repositories scare you. One of the best pieces of advice I was given in FRC was that as long as you come to robotics every day willing to learn and involve yourself, you will be successful. Keep that in mind as you read this book and prepare for your next FRC season.

Resources

These links will guide you to different resources that you might want to use as you program your robot.

Past Repositories:

- Infinite Recharge (2020 and 2021): github.com/BTK203/InfiniteRecharge-2021
- Destination: Deep Space (2019): github.com/wh1ter0se/DeepSpace-2019
- 2018 Fall Challenge (2018): github.com/wh1ter0se/FallChallenge-2018
- FIRST: Power Up (2018): github.com/wh1ter0se/PowerUp-2018

Current Repositories:

- Spirit Bot Robot Code: github.com/BTK203/BlowerBot
- KiwiLight: github.com/wh1ter0se/KiwiLight
- Hyperdrive: github.com/BTK203/Hyperdrive

Helpful Links:

- FRC Programming Done Right: frc-pdr.readthedocs.io/en/latest/
- FRC Control System Docs: docs.wpilib.org/en/stable/
- WPILib Java Docs: first.wpi.edu/FRC/roborio/release/docs/java/

Firmware:

- FRC Robot Radio: tinyurl.com/frcrobotradio
- Talon SRX: tinyurl.com/talonsrxfirmware
- Talon FX: tinyurl.com/talonfxfirmware
- Spark Max: tinyurl.com/sparkmaxfirmware

FRC Programming Tools

In order to successfully program and control your FRC robot, the below software must be installed. Note that Windows Defender or installed antivirus software may try to block some of the downloads or executables. If this happens, you should run the executables anyway. All below downloads are safe.

- NI Game Tools: tinyurl.com/nigametools
- WPILib: github.com/wpilibsuite/allwpilib/releases/latest/
 - WPILib Install Directions: tinyurl.com/wpilibinstall
- CTRE Phoenix: github.com/CrossTheRoadElec/Phoenix-Releases/releases
- REV Hardware Client: docs.revrobotics.com/rev-hardware-client/
- Git: git-scm.com/downloads
- TortoiseGit: tortoisegit.org/download/

Note: If you prefer to use the command-line form of Git, then TortoiseGit is not needed. However, TortoiseGit is much more user-friendly.

Physical Components of the Robot

An FRC robot contains multiple components that your code must interface with, such as motor controllers, pneumatic controllers, and other computers. The below components are very common on FRC robots. Information about how to code a few of these components is also provided.

Processors and Coprocessors

- RoboRIO



The RoboRIO is the “main computer” of the robot. Your robot code will be pushed to and executed from this device. It interfaces with almost all other devices on the robot and also communicates with the Driver Station Computer. Communication between the RIO and the Driver Station can be achieved in three ways:

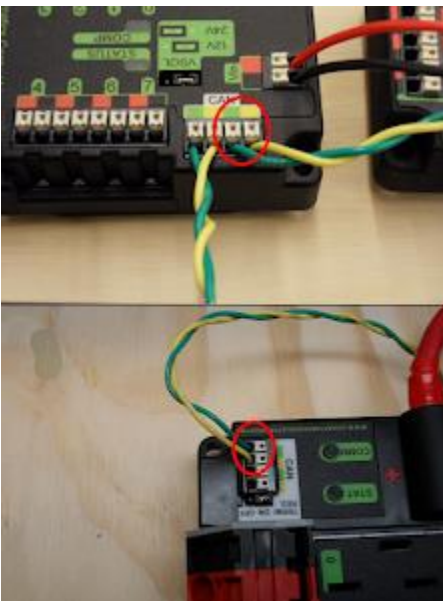
- Wireless communication by connecting the Driver Station Computer to the robot’s wi-fi network
- Wired communication by connecting a USB A to B cable (a.k.a “printer cable”) to the USB port
- Wired communication by connecting the Driver Station Computer to the robot radio with an Ethernet cable. An Ethernet tether between the Driver Station computer and the robot should not be used unless the USB tether does not work
- Wireless communication by connecting the Driver Station Computer to the Field Management System (FMS). This form of communication is used at competitions to allow the field to enable or disable all of the robots on the field. To connect FMS to the Driver Station Computer, connect the Ethernet cable at the alliance station to the Driver Station Computer.

- Robot Radio



The Robot Radio is a special wi-fi router that is powered by the robot and is used to establish communication between the robot and other remote devices, such as the Driver Station computer or FMS if you are at competition. It has two ethernet ports. One of those ports must be connected to the RIO, and the other can be used to connect a coprocessor such as a Raspberry Pi to the robot, so that it can communicate with the RIO.

- CAN Network



The CAN (Controller Area Network) is an interface that is used to connect motor controllers to the RoboRIO. All motors on the robot receive instructions and requests from the RIO using this two-wire network. Because the network has only two wires, you should be careful about how often you set motor speeds or access encoder counts for your motors. More about that can be found in the Robot Code Best Practices chapter.

CAN control is not the only method of controlling your motor controllers; PWM (pulse-width modulation) can be used to control your motors as well. However, PWM is not recommended because PWM noise and crosstalk can cause undesirable movement on the part of the motors, and may also interfere with other devices. CAN control gives you more reliable control of your motors and is the best motor control method for FRC applications.

A note about CAN IDs: Every motor controller on the CAN network should have a unique CAN ID number, which is used to differentiate that controller from others on the network. When creating a new motor controller in code, this unique ID must be provided to specify which motor controller you are programming. Guidance on how to change these IDs can be found in the Managing Motor Controllers chapter.

Motor Controllers

- Talon SRX



The Talon SRX is a motor controller developed by CTRE (Cross The Road Electronics). It is used to control brushed motors. In your code, this motor controller is represented by the `TalonSRX` class.

Basic Talon SRX methods (examples use a `TalonSRX` object named `motor`):

Function	Method
Create a new <code>TalonSRX</code>	<pre>new TalonSRX(int id)</pre> <p><code>id</code> is the unique CAN ID of the motor being controlled by the new object.</p> <p>Example: Create a new <code>TalonSRX</code> with a CAN ID of 4:</p> <pre>TalonSRX motor = new TalonSRX(4);</pre>
Drive the motor	<pre>motor.set(ControlMode mode, double value)</pre> <p><code>mode</code> is the method of control to use. The most commonly used mode is <code>ControlMode.PercentOutput</code>. This is commonly used to manually drive a motor. It sets the motor's power output to a percentage of its full power. If using this mode, then <code>value</code> should range from <code>-1</code> to <code>1</code>. Some other common control modes are listed below. You can read more about these in the PID Loops section in the Autonomous Functions chapter.</p> <ul style="list-style-type: none"> - <code>ControlMode.Position</code>: Uses a PID loop to try to reach and maintain an encoder position. The setpoint should be in encoder ticks.

	<ul style="list-style-type: none"> - <code>ControlMode.Velocity</code>: Uses a PID loop to try to reach and maintain an encoder velocity. The setpoint should be in encoder ticks per 100 milliseconds. - <code>ControlMode.Current</code>: Uses a PID loop to try to reach and maintain an amp draw. <p><code>value</code> is the drive value (or setpoint if mode uses a PID loop) to set the motor to.</p> <p>Example: Setting percent output to a <code>double</code> value called “percent” (from -1 to 1)</p> <pre>motor.set(ControlMode.PercentOutput, percent);</pre>
Invert the motor	<pre>motor.setInverted(boolean inverted)</pre> <p><code>inverted</code> should be <code>true</code> if the motor should be inverted, and <code>false</code> otherwise.</p> <p>This method should be called a maximum of one time when the motor is defined. It should NOT be called periodically.</p> <p>Example: Inverting the motor</p> <pre>motor.setInverted(true);</pre>
Set the motor to braking mode or coasting mode	<pre>motor.setNeutralMode(NeutralMode mode)</pre> <p><code>mode</code> can be one of two values:</p> <ul style="list-style-type: none"> - <code>NeutralMode.Brake</code> sets the motor to braking mode. In this mode, the motor will resist motion by actively working to stop itself if it is going faster than it is trying to. All drivetrain motors should be set to braking mode. - <code>NeutralMode.Coast</code> sets the motor to coasting mode. In this mode, the motor will not resist motion. <p>Example: Setting the motor to Braking Mode:</p> <pre>motor.setNeutralMode(NeutralMode.Brake);</pre>
Set an amp limit	<pre>motor.configContinuousCurrentLimit(int amps)</pre>

	<p><code>amps</code> is the maximum number of amps that the motor can pull at any given time.</p> <p>Example: Setting an amp limit of 40 amps:</p> <pre>motor.configContinuousCurrentLimit(40);</pre>
Set open-loop ramp rate	<pre>turretYaw.configOpenloopRamp(double seconds);</pre> <p><code>seconds</code> is the number of seconds that it should take for the motor to go from 0 power to full power. This does not apply to PID loops.</p> <p>Example: Setting an open loop (non-PID) ramp of 0.5 seconds</p> <pre>turretYaw.configOpenloopRamp(0.5);</pre>

Other methods for the TalonSRX are available to use. Information on these methods can be found in the TalonSRX documentation.

- Talon FX



The Talon FX was developed by CTRE to control the Falcon 500 motor. As seen in the picture, the Talon FX comes integrated with the motor itself. The Talon FX is represented by the `TalonFX` class in code, and all of its methods for motor control are the same as on the Talon SRX.

- Spark MAX



The Spark MAX is a motor controller that was developed by REV Robotics that can be used to control both brushed motors as well as brushless motors. It is commonly used to control NEO brushless motors. It is represented by the `CANSparkMax` class in code.

Basic Spark MAX methods (using a `CANSparkMax` object called `motor`):

Function	Method
Create a new <code>CANSparkMax</code>	<pre>new CANSparkMax(int id, MotorType type)</pre> <p><code>id</code> is the CAN ID of the motor that this new object will control. <code>type</code> is the type of motor that the Spark Max is controlling. It can be one of the following values:</p> <ul style="list-style-type: none"> - <code>MotorType.kBrushless</code>: Use when controlling a brushless motor such as a NEO - <code>MotorType.kBrushed</code>: Use when controlling a brushed motor. <p>Example: Creating a new <code>CANSparkMax</code>, with the CAN ID 4, which controls a brushless motor:</p> <pre>CANSparkMax motor = new CANSparkMax(4, MotorType.kBrushless);</pre>
Drive the motor (percent output only)	<pre>motor.set(double percent)</pre> <p><code>percent</code> is the percent output to set the motor to. It must range from <code>-1</code> to <code>1</code>.</p> <p>Example: Setting the percent output to a <code>double</code> value called "percent" (from -1 to 1):</p> <pre>motor.set(percent);</pre>

	<p>Note: For information about how to set the motor to a certain velocity or position using the on-board PID loop, see the PID Loops section in the Autonomous Functions chapter.</p>
Invert the motor	<pre>motor.setInverted(boolean inverted)</pre> <p><code>inverted</code> should be <code>true</code> if the motor should be inverted, and <code>false</code> otherwise.</p> <p>Example: Inverting the motor</p> <pre>motor.setInverted(true);</pre>
Set the motor to braking or coasting mode	<pre>motor.setIdleMode(IdleMode mode)</pre> <p><code>mode</code> can be one of two values:</p> <ul style="list-style-type: none"> - <code>IdleMode.kBrake</code> sets the motor to braking mode. In this mode, the motor will resist motion by actively working to stop itself if it is going faster than it is trying to. All drivetrain motors should be set to braking mode. - <code>IdleMode.kCoast</code> sets the motor to coasting mode. In this mode, the motor will not resist motion. <p>Example: Setting the motor to Braking Mode:</p> <pre>motor.setIdleMode(IdleMode.kBrake);</pre>
Set an amp limit	<pre>motor.setSmartCurrentLimit(int limit)</pre> <p><code>limit</code> is the maximum number of amps that the motor can be allowed to draw at any given time.</p> <p>Example: Setting an amp limit of 40 amps:</p> <pre>motor.setSmartCurrentLimit(40);</pre>
Set open-loop ramp rate	<pre>motor.setOpenLoopRampRate(double seconds)</pre> <p><code>seconds</code> is the number of seconds that it should take for the motor to go from 0 power to full power.</p> <p>This method does not affect the behavior of PID loops. To set a PID loop ramp rate, use <code>setClosedLoopRampRate(double seconds)</code>.</p>

Example: Setting an Open Loop (non-PID) ramp of 0.5 seconds:

```
motor.setOpenLoopRampRate(0.5);
```

Other Motor Devices

- Encoders



Encoders track the rotational position of a motor or other axis. You can use encoders to determine how many times a motor has turned, or how fast a motor is currently going. Some motors, such as the NEO or Falcon 500, have encoders built into them. Other motors, such as the one pictured (red-line motor), must have encoders attached to them. All encoders must be connected to the motor controller in some way in order to be accessible to the code.

Two different types of encoders exist: Analog and Quadrature. For most encoders that are not built into the motor itself, you will need to know what type of encoder is

being used in order to correctly access it in the code. Most encoders used in FRC applications will be quadrature.

Basic encoder methods:

- CTRE Motor Controllers (using a `TalonSRX` object named `motor`):

Function	Method
Get position of a quadrature encoder	<pre>motor.getSensorCollection().getQuadraturePosition()</pre> <p>Returns: an <code>int</code> value representing the position of the encoder in ticks.</p>
Get velocity of a quadrature encoder	<pre>motor.getSensorCollection().getQuadratureVelocity()</pre> <p>Returns: an <code>int</code> value representing the position of the encoder in ticks per 100 milliseconds.</p>
Set position of a quadrature encoder	<pre>motor.getSensorCollection().setQuadraturePosition(int position, int timeoutMS)</pre> <p><code>position</code> is the new position of the encoder, in ticks. <code>timeoutMS</code> is the number of milliseconds to wait before applying the new position.</p> <p>It should be noted that this method does not drive the motor to achieve the specified position; it directly sets the value of the</p>

	<p>encoder's current position. To drive the motor to achieve a position, use a PID loop. Information about PID loops can be found in the PID Loops section of the Autonomous Functions chapter.</p> <p>Example: Set the current position of the encoder to a value named "p":</p> <pre>motor.getSensorCollection().setQuadraturePosition(p, 0);</pre>
Invert the encoder	<pre>motor.setSensorPhase(int position)</pre> <p><code>position</code> should be <code>true</code> if the encoder should be inverted, otherwise <code>false</code>.</p> <p>This method should be used if a positive output on the motor causes a negative change in position.</p> <p>Example: Inverting the encoder</p> <pre>motor.setSensorPhase(true);</pre>

- REV Motor Controllers (using a `CANSparkMax` object named `motor`):

Function	Method
Get position of encoder	<pre>motor.getEncoder().getPosition()</pre> <p>Returns: a <code>double</code> value in rotations</p>
Get Velocity of encoder	<pre>motor.getEncoder().getVelocity()</pre> <p>Returns: a <code>double</code> value in rotations per minute</p>
Set the current position of the encoder	<pre>motor.getEncoder().setPosition(double position)</pre> <p><code>position</code> is the new position value of the encoder in rotations.</p> <p>As was the case with the CTRE motor controllers, it should be noted that this method does not drive the motor to achieve the specified position; it directly sets the value of the encoder's current position. To drive the motor to achieve a position, use a PID loop. Information</p>

	<p>about PID loops can be found in the PID Loops section of the Autonomous Functions chapter.</p> <p>Example: Set the current position of the encoder to a value named “p”:</p> <pre>motor.getEncoder().setPosition(p);</pre>
Invert the encoder	<pre>motor.getEncoder().setInverted(boolean inverted)</pre> <p><code>inverted</code> should be <code>true</code> if the encoder should be inverted, and <code>false</code> otherwise.</p> <p>This method should be used if a positive output on the motor causes a negative change in position.</p> <p>Example: Inverting the encoder:</p> <pre>motor.getEncoder().setInverted(true);</pre>

- *Limit Switches*

Shown below: Mechanical limit switch (top), magnetic limit switch (bottom)



Limit Switches are devices used to define the limit of a system's movement. Limit switches can be either mechanical (triggered by compressing a lever) or magnetic (triggered by passing over a fixed magnet). A limit switch will typically be mounted on both ends of a rotational or linear actuator. When used in a safe-bounds system, they prevent a motor from driving a system out of its intended range of motion to the point that the system breaks itself or another system.



When a system triggers the limit switch, the motor driving the system is stopped. In FRC, limit switches are typically connected to the controller of the motor driving the system, and the stopping of the motor is handled by the controller. FRC Programmers typically do not need to handle the stopping of the motors in their code (after connecting and assigning the limit switch in the code). However, some may wish to send the state of the switch (open or closed) to the dashboard or add some other functionality to the switch.

Basic Limit Switch Methods:

- CTRE (using a `TalonSRX` object named `motor`):

Function	Method
----------	--------

Get the state of the motor's forward limit switch	<pre>motor.isFwdLimitSwitchClosed()</pre> <p>Returns: an <code>int</code> value; <code>0</code> if open, <code>1</code> if closed.</p>
Get the state of the motor's reverse limit switch	<pre>motor.isRevLimitSwitchClosed()</pre> <p>Returns: an <code>int</code> value; <code>0</code> if open, <code>1</code> if closed.</p>

- REV (using a `CANSparkMax` object named `motor`):

Function	Method
Get the motor's forward limit switch	<pre>motor.getForwardLimitSwitch(LimitSwitchPolarity polarity)</pre> <p><code>polarity</code> is the natural state of the limit switch. It can be one of the following values:</p> <ul style="list-style-type: none"> - <code>LimitSwitchPolarity.kNormallyOpen</code>: Should be used if the limit switch is open when not triggered. - <code>LimitSwitchPolarity.kNormallyClosed</code>: Should be used if the limit switch is closed when not triggered. <p>Returns: a <code>CANDigitalInput</code> object. To get the state of the limit switch at any time, use the <code>CANDigitalInput</code> object's <code>get()</code> method, which returns a <code>boolean</code>; <code>true</code> if the switch is triggered, <code>false</code> otherwise.</p>
Get the motors' reverse limit switch	<pre>motor.getReverseLimitSwitch(LimitSwitchPolarity polarity)</pre> <p>Parameters and return value for this method are exactly the same as that for <code>getForwardLimitSwitch()</code>.</p>

Pneumatics

- Pneumatics Control Module (PCM)



The PCM is used to control Solenoids that make pneumatic pistons extend and retract. All solenoids are plugged into this one of the module's 8 numbered ports. This device also controls the pneumatic compressor.

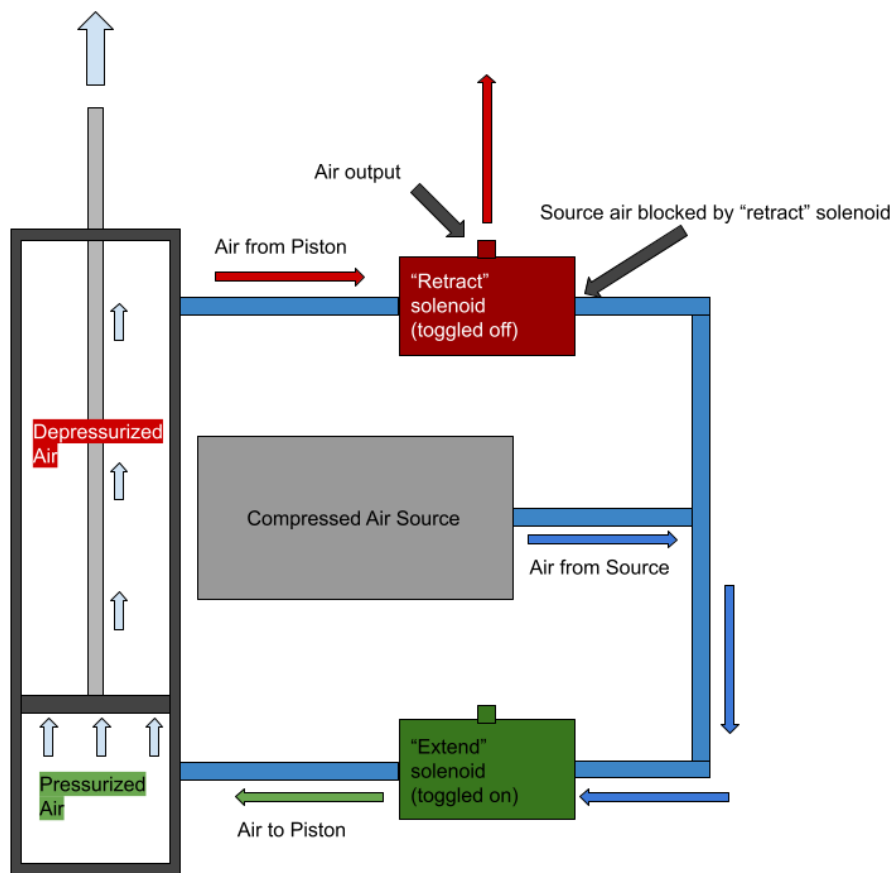
- Solenoids

Pictured: A double-solenoid. This device combines the functionality of two solenoids into one object.



A pneumatic solenoid is a kind of switch that allows air to enter a pneumatic system such as a piston. When toggled on, compressed air will enter the system. When toggled off, the air will leave the system. Some pneumatic systems, such as a pneumatic piston,

may require more than one solenoid to work. For example, to make a pneumatic piston both extend and retract, two solenoids are needed: one solenoid to extend the piston, and another to retract it. To extend the piston, the “extend” solenoid must be toggled on, allowing air to flow to the extending side of the piston, and the “retract” solenoid must be toggled off, allowing air to flow out of the retracting side of the piston. The difference in air pressure between the two sides will cause the piston to extend. To retract the piston, the opposite must happen (“extend” solenoid toggled off, “retract” solenoid toggled on). Solenoids are represented by the `Solenoid` class in code.



Shown Above: How solenoids are used to make a piston extend.

Basic Solenoid methods (using a `Solenoid` object named `solenoid`):

Function	Method
Create a new <code>Solenoid</code>	<pre>new Solenoid(int port)</pre> <p><code>port</code> is the ID of the port on the PCM that the solenoid is plugged into (0 to 7).</p> <p>Example: Creating a new Solenoid which is plugged into port 6 on the PCM</p> <pre>Solenoid solenoid = new Solenoid(6);</pre>
Set a piston's state	<pre>solenoid.set(boolean state)</pre>

`state` should be `true` if toggling the piston on and `false` otherwise.

Example: Toggle on a solenoid, allowing air into a pneumatic system:

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

Example: Toggle off a solenoid, releasing air from a pneumatic system:

```
solenoid.set(false);
```

- Compressor



The compressor compresses air for use in pneumatic systems. Without it, there would be no compressed air to extend or retract pneumatic pistons on the robot. All that is needed for the compressor to work is one instance of a `Compressor` object in the robot code. Typically, this object gets its own Subsystem (see WPILib chapter for more information on subsystems). This `Compressor` object will automatically turn the compressor on when needed to pressurize the system and turn it off when the system is pressurized, so no extra programming of the compressor is needed. However, a programmer is able to manually disable the compressor if desired.

Basic Compressor methods (using a `Compressor` object named `compressor`):

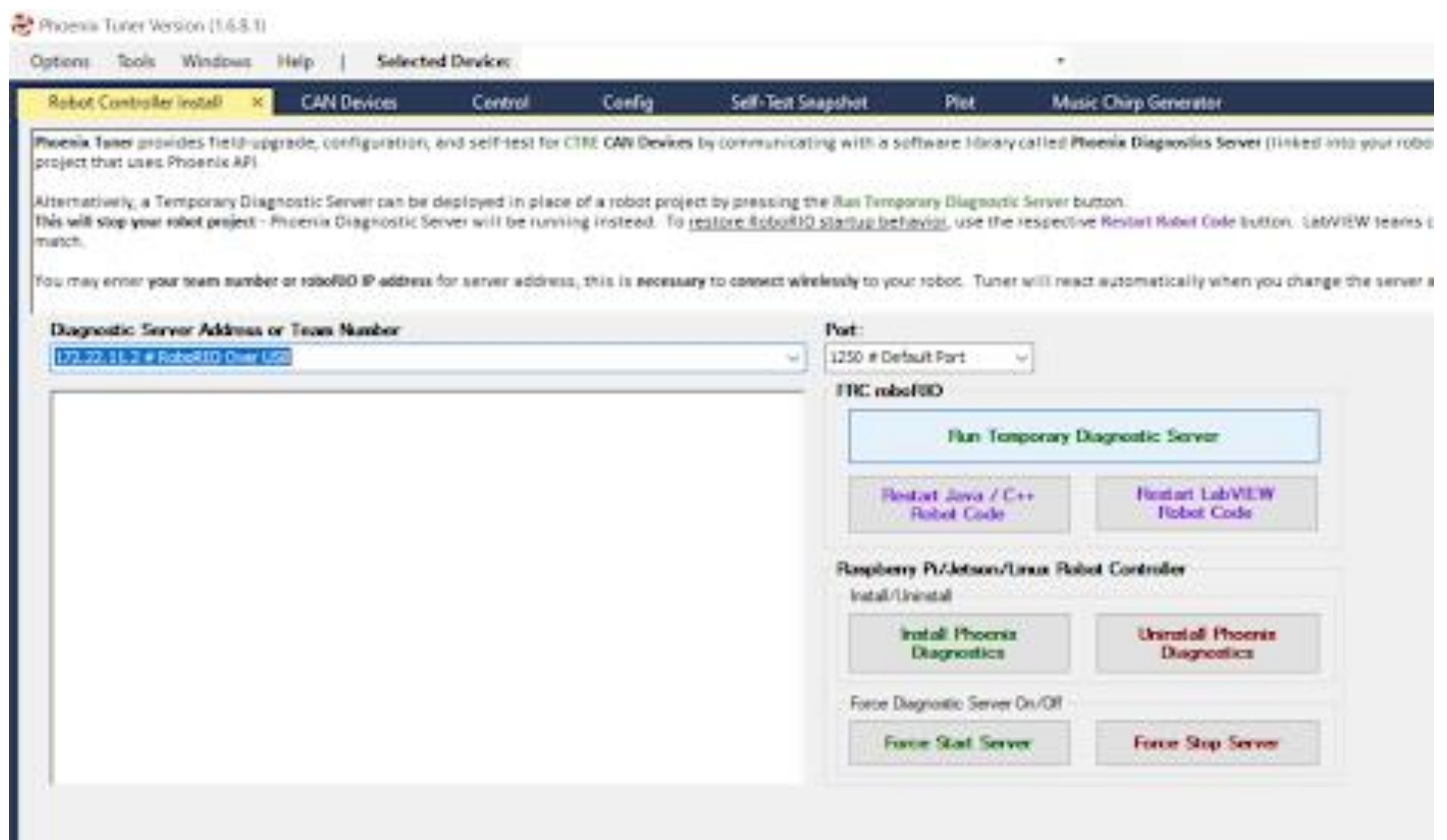
Function	Method
Create a new <code>Compressor</code>	<pre><code>new Compressor()</code></pre> <p>Example: creating a new Compressor:</p> <pre><code>Compressor compressor = new Compressor();</code></pre>

Disable the compressor (may be desired because compressors are loud, but not required)	<code>compressor.stop()</code>
Enable the compressor (only needed after disabling it using previous method)	<code>compressor.start()</code>

Managing Motor Controllers

In order for the Rio to communicate with a motor controller to make a motor move, the motor controller needs to have updated firmware and a unique ID. Failure to have either of these things can cause unpredictable behavior on the part of the motors. This chapter describes how to update your motor controllers and set their CAN IDs.

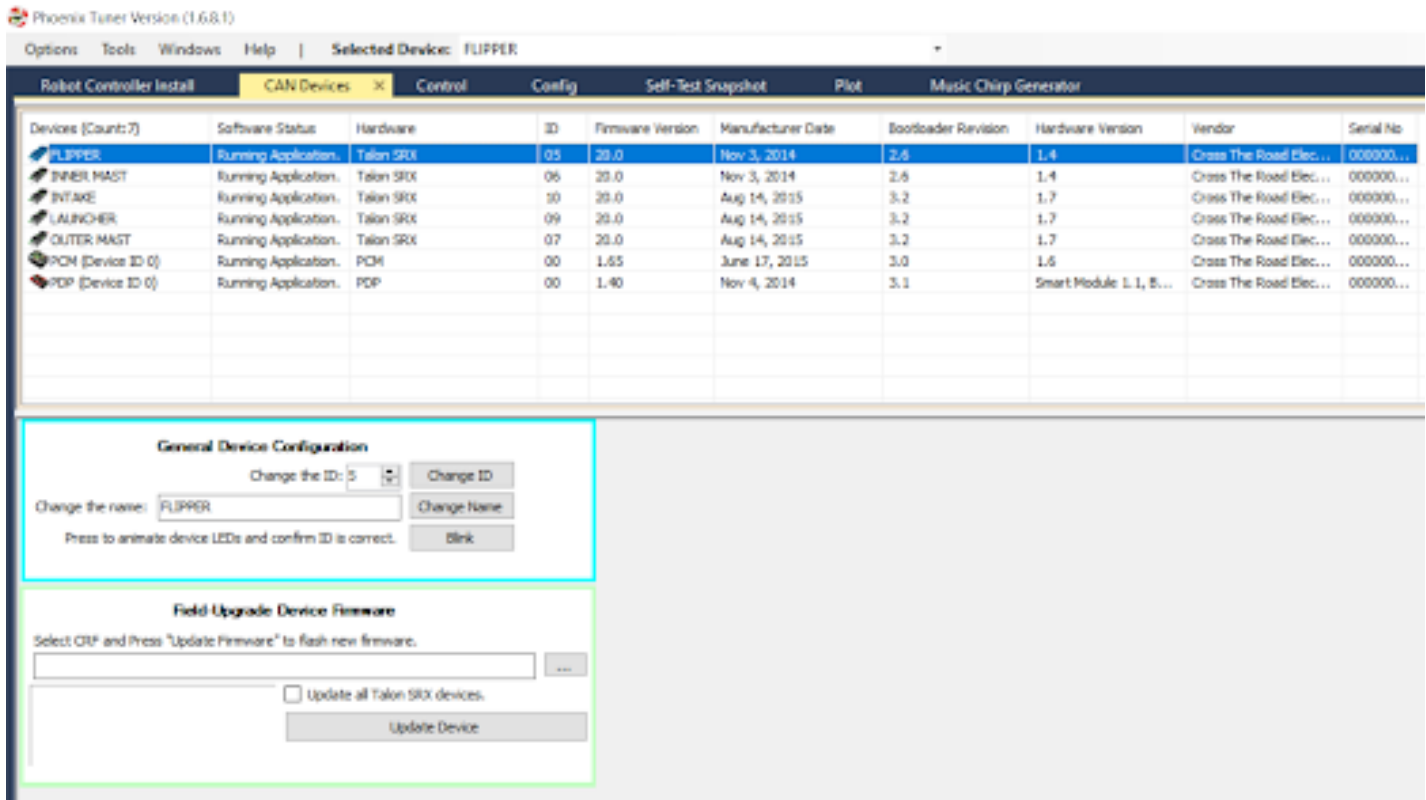
CTRE Motor Controllers



Shown above: Phoenix Tuner

- Before starting, ensure that you have the Phoenix Framework installed (see the FRC Programming Tools chapter), and the latest Talon SRX Firmware downloaded (see the Resources chapter).
- First, connect the computer to the RoboRIO with a USB A-to-B cable (printer cable).
- Open Phoenix Tuner. A window like the one pictured above should appear.

- Press the “Run Temporary Diagnostic Server” button.
- If the server install is successful, then click over to the “CAN Devices” tab. Give Phoenix Tuner a few seconds to discover and display connected devices. When it finishes, you should see a list of every CTRE device that is connected to the CAN network.



- From here, you can update firmware and assign names and CAN IDs to all of your motor controllers. Do not change names or IDs of the PCM or PDP devices. Those devices do not need to be differentiated between and will function as-is.
- **Assigning Names and CAN IDs:**
All motor controllers, especially on competition robots, should have a unique name that clearly states its function, as well as a unique CAN ID to distinguish it from other motor controllers in the code. See the best practices section below for general guidelines for naming and ID-ing motor controllers.
 - Click on the name of the device that you wish to change.
 - To confirm that you have the correct motor controller selected, click the “Blink” button in the “General Device Configuration” area. The selected motor controller’s LED status lights will blink rapidly for a few seconds.
 - To assign a new name to the motor controller, enter the new name into the “Change the Name” textbox and click the “Change Name” button.
 - To assign a new ID to the motor controller, enter the ID into the “Change the ID” textbox and click the “Change ID” button.

- *Updating Firmware*

Motor controller firmware should be updated every time the RoboRIO is updated. Before updating, ensure that the latest firmware is downloaded onto the computer. Firmware for some CTRE motor controllers can be found in the Resources chapter of this book.

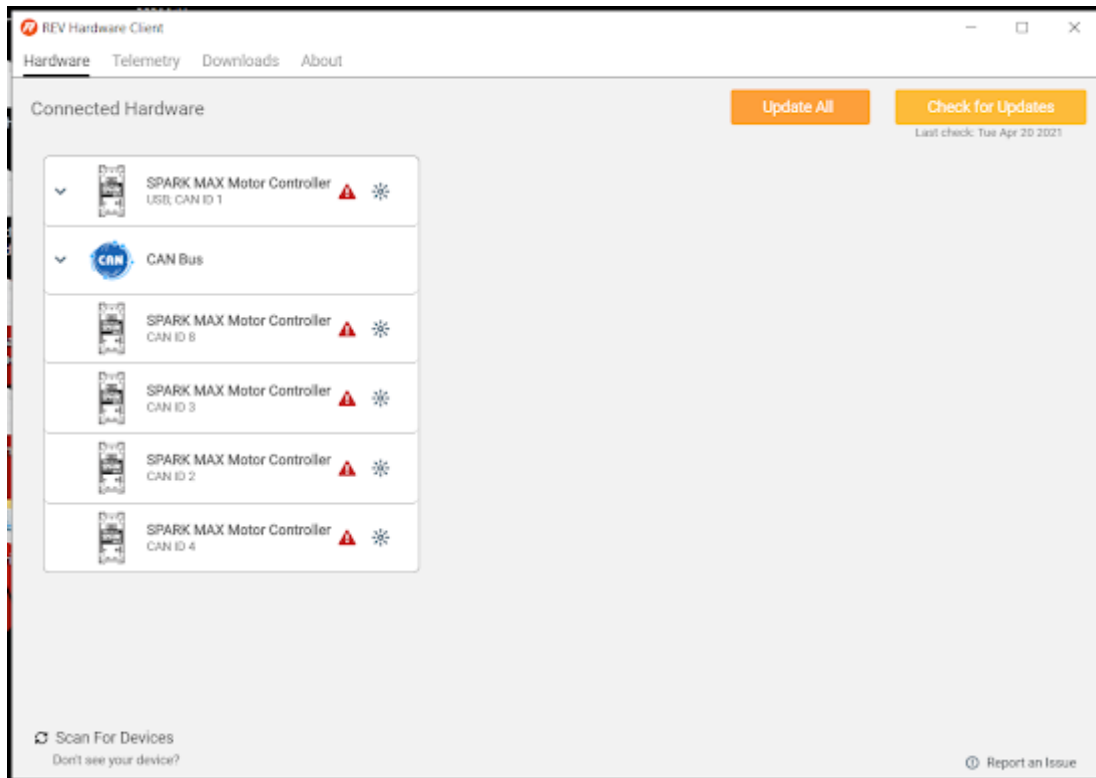
- In the “Field-Upgrade Device Firmware” area, browse your computer for the firmware using the “...” button. Ensure that the firmware selected is the correct firmware for the device being updated (i.e do not try to push Talon FX firmware to a Talon SRX).
- Select the motor controller you wish to update.
- Click the “Update Device” button.
- Repeat this process for every motor controller that needs to be updated.
- NOTE: You may have noticed the checkbox allowing you to update all devices with the same type. In the past, this option has not been reliable and has frozen Phoenix Tuner or the CAN network itself. It is far more reliable to simply update the motor controllers one-by-one.

REV Motor Controllers

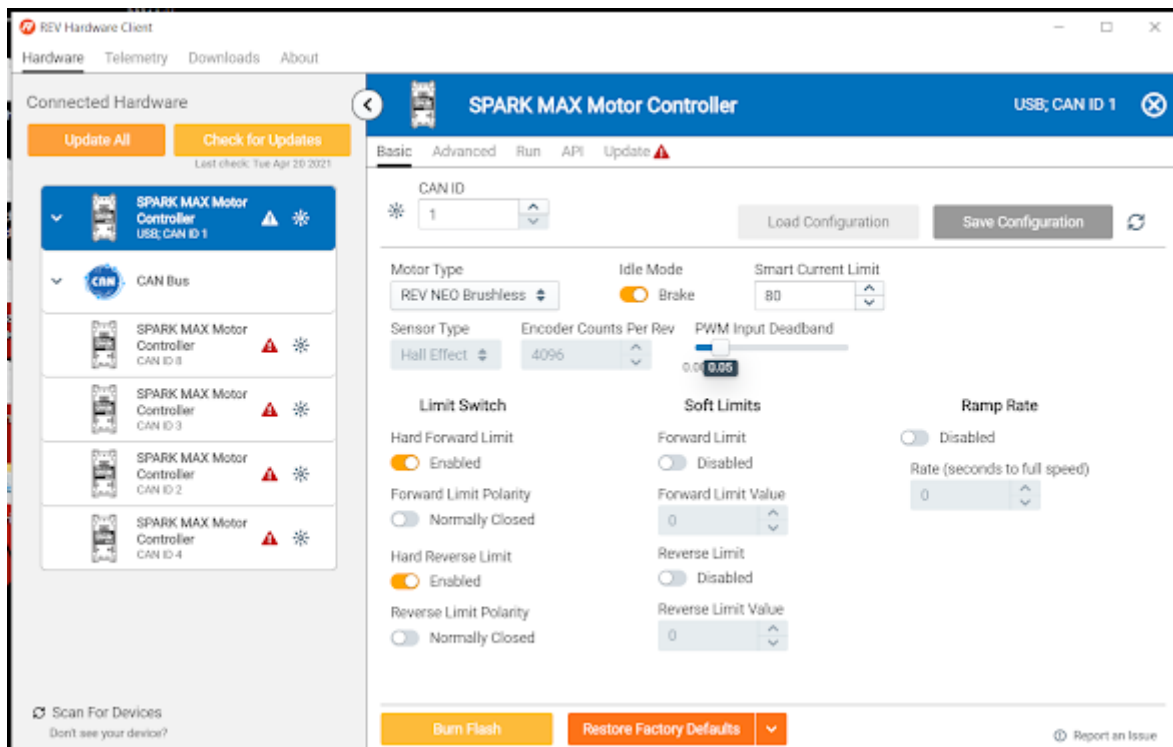
- To start, use a USB-C cable to connect the computer to any REV device that is connected to the CAN network. In this example, we will connect our computer to a Spark Max.



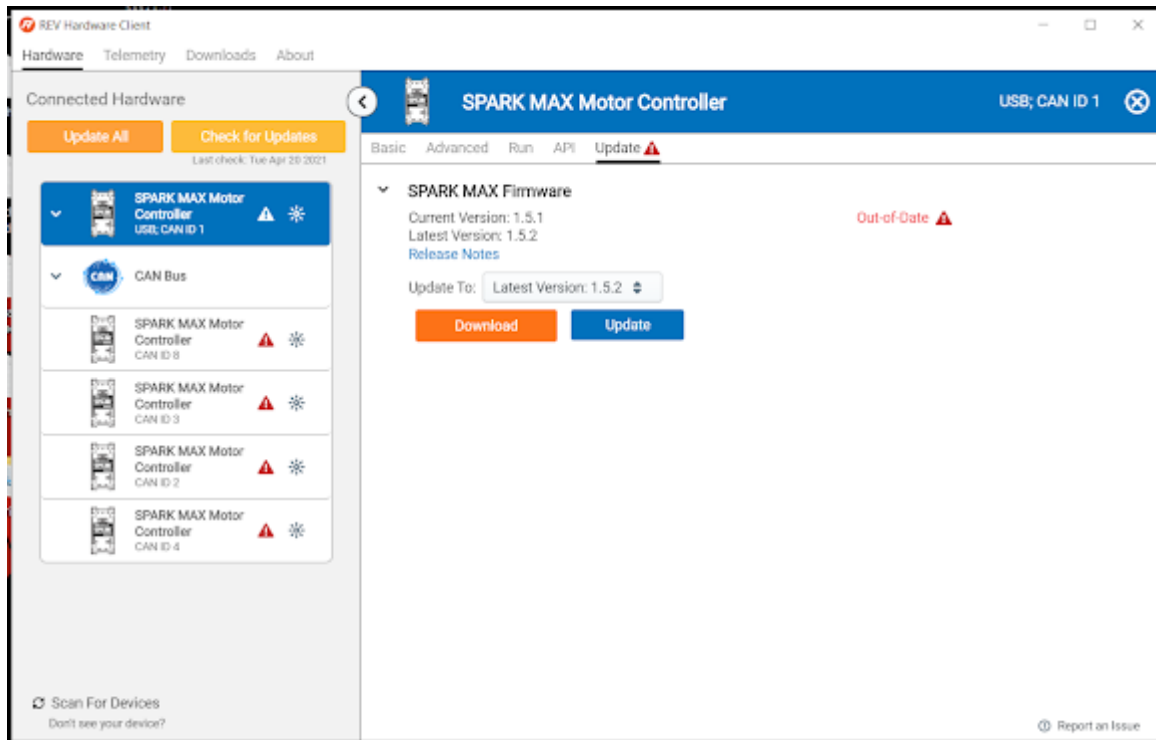
- Open REV Hardware Client. The application should automatically scan for connected devices and list them. The application will also allow you to connect to, update, and edit REV devices that are connected to the same CAN network as the device connected to the computer.



- To edit the properties of any connected device, click on it. A menu should appear that looks like the one pictured below. Using this menu, you can edit CAN IDs, Braking modes, and current limits.



- To update your device, switch from the “Basic” menu to the “Update” menu. It should look like this:



- From here, simply download and update the firmware using the buttons in the menu. You can also use the “Update All” button to update all of the devices to the most recent firmware.

Motor Controller Best Practices

- Never use an ID of 0. The PDP and PCM already use that ID.
- Never use duplicate IDs, even on motor controllers by separate vendors (i.e TalonSRX and SparkMax).
- Do not skip IDs. ID 1 through 10 should be assigned before 11 is used.
- Keep your IDs stored in the Constants class of your robot code. Whenever modifying IDs, have that file open.
- When naming a motor controller, use a name that is short, but accurately describes what the motor does. For example, a motor that drives an intake should be called “Intake”

Maintenance

New software and firmware for robots is periodically released throughout the season and off-season. Programmers should update software and firmware for any robots currently being used at the beginning of every season at the very least. Some systems may behave unpredictably or not work at all if the software and firmware is out of date. All software on the Driver Station computer should similarly be updated at least once per season.

Updating the Driver Station Computer

When updating the driver station computer, be sure that you update the below items. It may help to uninstall existing tools so there is no chance that the old tools are accidentally used.

- ***FRC Game Tools***
Download and install the latest version of the FRC Game Tools. The link can be found in the FRC Programming Tools chapter of this book.
- ***WPILib***
Download and install the latest version of WPILib. The link is also found in the FRC Programming Tools chapter of this book. You will need to re-download WPILib and re-run the installer.
- ***CTRE Phoenix***
Download and install the latest version of CTRE Phoenix (if using CTRE motor controllers). The link can be found in the FRC Programming Tools chapter of this book.
- ***REV Hardware Client***
Download and install the latest REV Hardware client (if using REV motor controllers). The link can be found in the FRC Programming Tools chapter of this book.
- ***Motor Controller Firmware***
Download the latest firmware for all motor controllers being used.
- ***Robot Projects***
After updating WPILib, open VSCode and ensure that the WPILib extension is on the latest version. Then, follow these steps on all robot projects that are currently being worked on:
 - Open the robot project.
 - Wait for about 10 seconds. If your robot project is incompatible with the latest version of WPILib, you will be prompted to import your project into the correct version. If this happens, follow the steps outlined in the “Importing Projects” section in the WPILib chapter of this book.

- *Vendordeps*

After ensuring that your robot project is updated to the correct version of WPILib, you need to update the libraries within the projects. Follow the steps outlined in the “Vendordeps” section, inside of the WPILib chapter of this book.

Updating the Robot

Do this after updating the Driver Station computer. Be sure that you update the following items:

- *RoboRIO*

Instructions for updating the RoboRIO can be found at tinyurl.com/rioimaging.

- *Robot Radio*

Instructions for updating the robot radio can be found at tinyurl.com/frcrobotradio

- *All Motor Controllers*

See above chapter on managing motor controllers.

Source Control using Git

Git is a powerful tool for version control. It stores your code in the cloud, logs all changes to your code, and allows you to revert to any previous version at any time. It also allows you to have multiple “branches” of your code - that is, multiple versions of your project where different features can be developed and tested. When a feature or version is completed, its branch can be merged into any other branch. In FRC, version control is extremely important because a programming team may need to work on different parts or versions of the same project. It is strongly recommended that all programmers have a GitHub account, which is free, to work on robot code, and Git installed on all computers where work on robot code will be done.

Version Control Basics

Repositories, Branches, Forks, and Clones

Code that is being managed with Git is stored in a repository. That repository can contain multiple different versions of the same project, in which different features are being developed. These different versions are called branches. Programmers can also create their own version of your repository, called a Fork. Forks work just like branches, but they belong to the person that created the fork, not the person that created the original repository. If a programmer wants to download a repository onto their computer, they may do so by creating a clone of the repository. Most editing of code in a repository will happen on a clone. Once edited, code on a clone can be uploaded to the repository that the clone was created from.

Creating a Repository

To create a new GitHub repository, follow these steps:

- Go to github.com.
- Log in, then click on the



The screenshot shows the GitHub 'Create a new repository' page. At the top, it says 'Create a new repository' and provides a brief explanation of what a repository is. Below this, there are fields for 'Owner' (set to 'BTK203') and 'Repository name'. A note suggests using short and memorable names. There is a 'Description (optional)' field. Under 'Visibility', 'Public' is selected with a radio button, and 'Private' is also an option. The 'Initialize this repository with' section has three checkboxes: 'Add a README file' (selected), 'Add .gitignore', and 'Choose a license'. Each checkbox has a brief description and a 'Learn more' link. At the bottom, there is a green 'Create repository' button.

- Fill out the form with the repository name and description.
- When creating a new repository, always add a readme.md, .gitignore, and license (GPL-3.0 is most commonly used).
- Click “Create Repository”.

Cloning a Repository

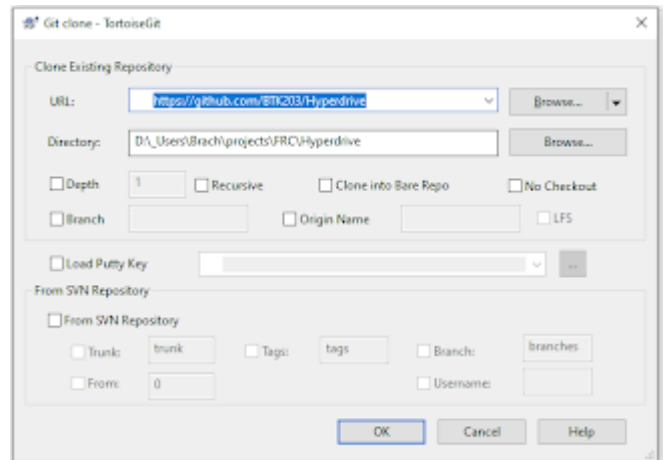
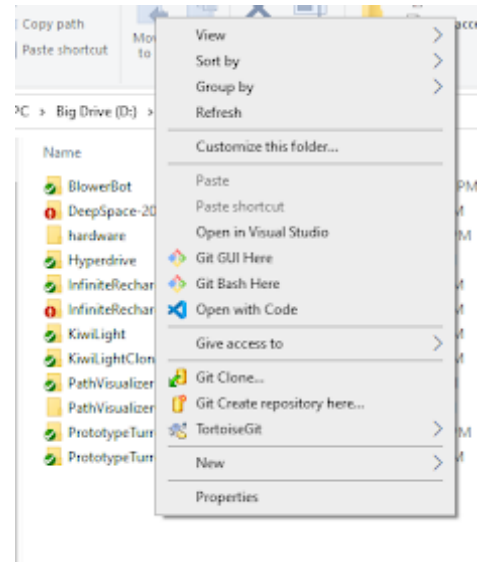
To create a clone of a repository, follow these steps:

- Using TortoiseGit

- In the file explorer, right click inside the folder you want to download the repository in.
- Select “Git Clone...”
- In the designated boxes, enter the URL of the repository to be cloned and the name of the directory that will be created for the repository to go into.
- If you want to clone a branch other than the main branch, select the “Branch” check box and enter the name of the branch you want to clone.
- Select OK.

- Using Git Bash

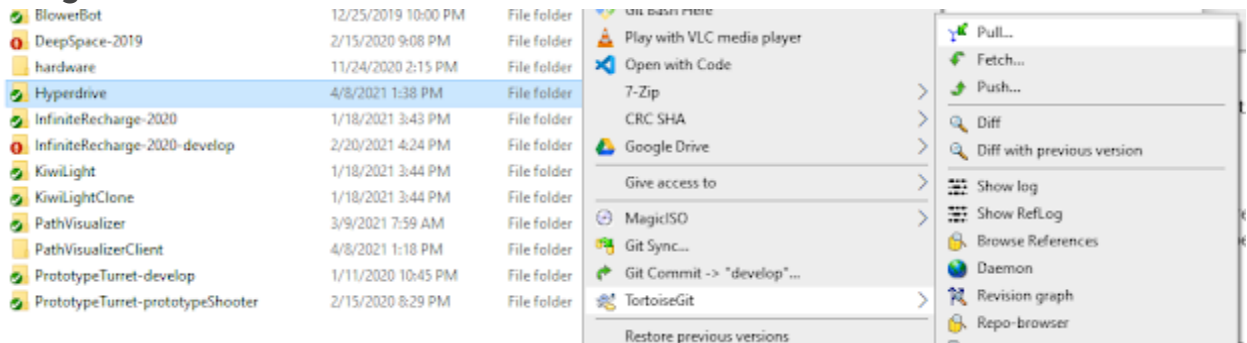
- In the terminal, `cd` to the directory you want to clone the repository in.
- Enter `git clone <URL to repo> <name of directory>`. The name of the directory to create the clone in is optional.



“Pulling” A Repository

Pulling a repository means downloading the latest changes from a repository to a clone. Before working with any code on a clone, it should be pulled first. Be sure all local changes are either reverted or committed before attempting to pull. To pull a repository, follow these steps:

- Using TortoiseGit

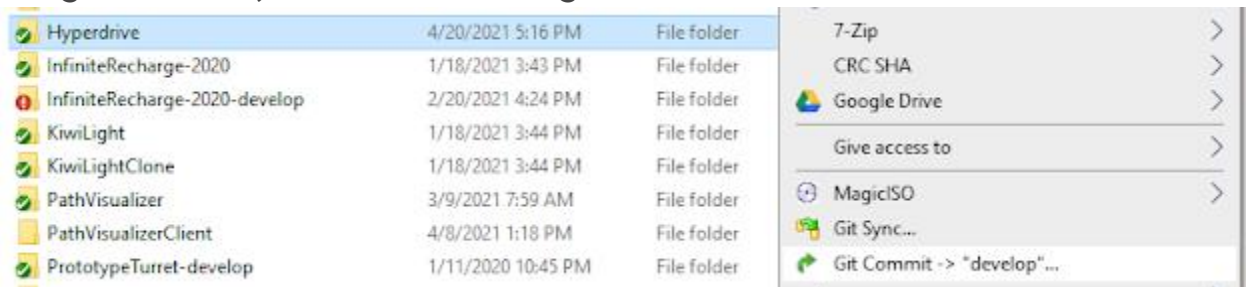


- In the file explorer, right click on the clone directory, or anything within it.
- Hover over “TortoiseGit” and select “Pull...” from the submenu. If no “TortoiseGit” menu exists, or “Pull...” is not a part of it, then the directory you selected is not a clone.
- Press “OK” in the dialog.
- **Using Git Bash**
 - In the terminal, `cd` to a location inside of the clone directory.
 - Enter `git pull`.

“Committing” to a Repository

Committing to a repository is the action of staging and logging changes made to a clone to prepare them to be “pushed” to the repository itself. This step must be taken before changes can be pushed. At the least, you should commit your changes at the end of every session of work. Other than that, you should commit changes only when needed. If you are using TortoiseGit, different steps must be taken to commit files that have never been committed to the repository before.

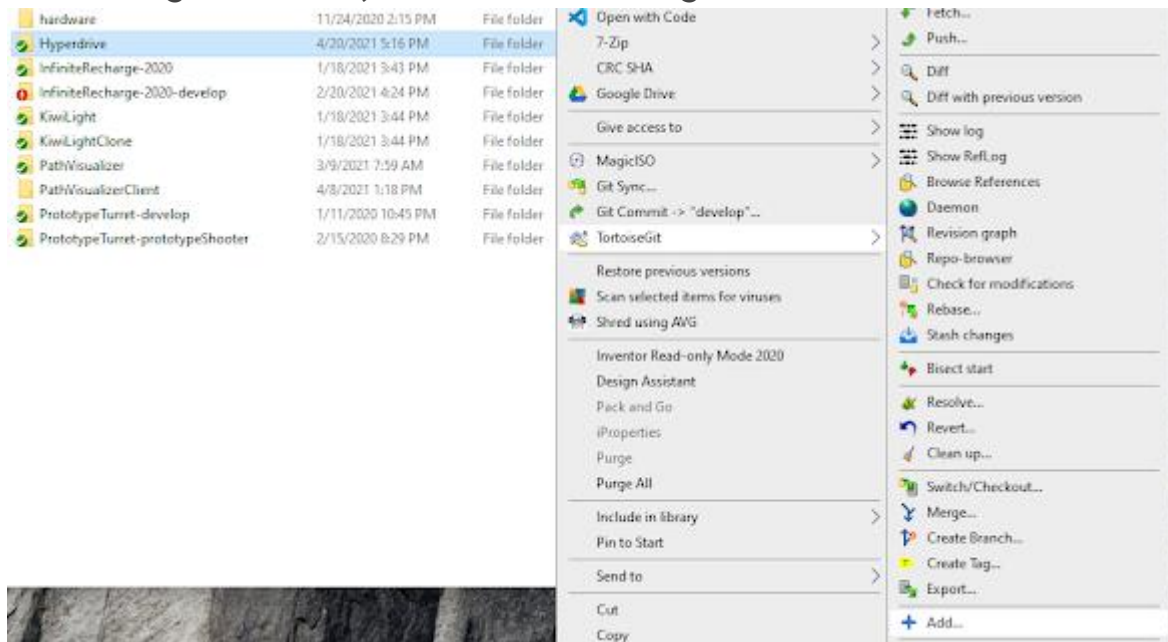
- Using TortoiseGit, if old files are being committed



- Right click on the clone directory and select “Git Commit -> <branch name>”
- In the big box at the top of the dialog, enter a commit message. This message is how you will identify the commit in the future if needed. That being said, the message should be short and memorable, but also offer a specific description as to what changes were made to the code.

- Select “Commit”. TortoiseGit also offers an option to “Commit and Push,” which commits and immediately pushes the changes to the repository. This option can be accessed by using the dropdown attached to the “Commit” button.

- Using TortoiseGit, if new files are being committed



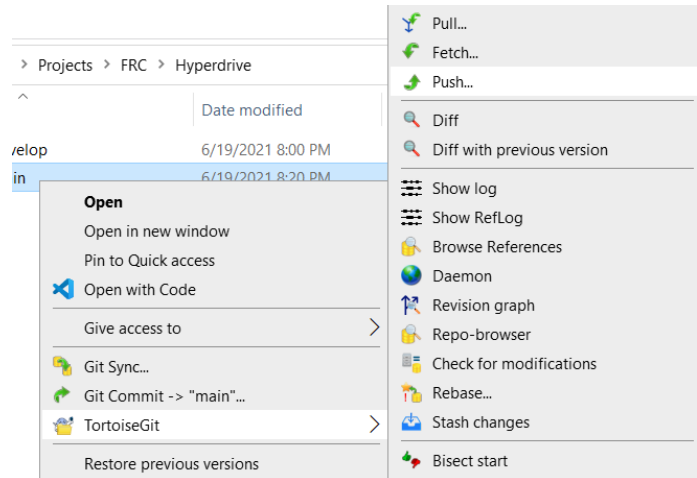
Note that this method will only work if you are committing files that have never been pushed before.

- Right click on the repository folder and hover over “TortoiseGit”, then select “Add...” from the submenu.
- A dialog box will appear that displays the names of every new file that is being added. Ensure that it has selected only the files that you wish to push. You can select or deselect a file using the checkbox next to its name.
- Select “Commit”
- Fill out the commit dialog just like you would for normal files.
- **Using Git Bash (works for both new and old files)**
 - In the terminal, `cd` to somewhere inside of the clone directory.
 - Enter `git stage *`
 - Enter `git commit -m "<commit message>"`. As mentioned earlier, the commit message should be short and memorable, while offering a description of the changes that are being pushed.

“Pushing” to a Repository

Pushing to a repository is the action of merging the changes you made on the clone with the repository itself. Before pushing, the changes you made to the clone need to be committed to the repository (see “Committing” to a Repository, above). To push to a repository, follow these steps:

- **Using TortoiseGit:**

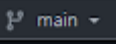


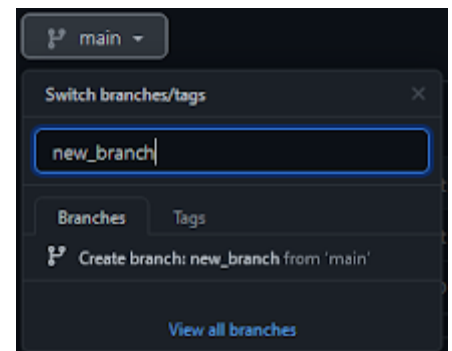
- Right click on the repository folder, hover over “TortoiseGit...”, then “Push” from the submenu.
- Press on the dialog that pops up.
- **Using Git Bash:**
 - In the terminal, `cd` to somewhere inside of the clone directory.
 - Enter `git push`.

Having Multiple Branches

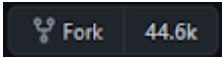
Branches exist so that a programmer can test out different features of their code without messing up the code that already works. FRC repositories should always have at least two branches: The main branch, and a “develop” branch, where the code is actually modified and tested. Only when the code works reliably and is satisfactory to the programmer should it be merged with the main branch (see “Pull Requests and Merging” below).

Creating a branch:

- Navigate to your repository on github.com.
- Locate the branch dropdown () and select it.
- In the text box, enter the name of your new branch and press the “Create branch” button to create it.



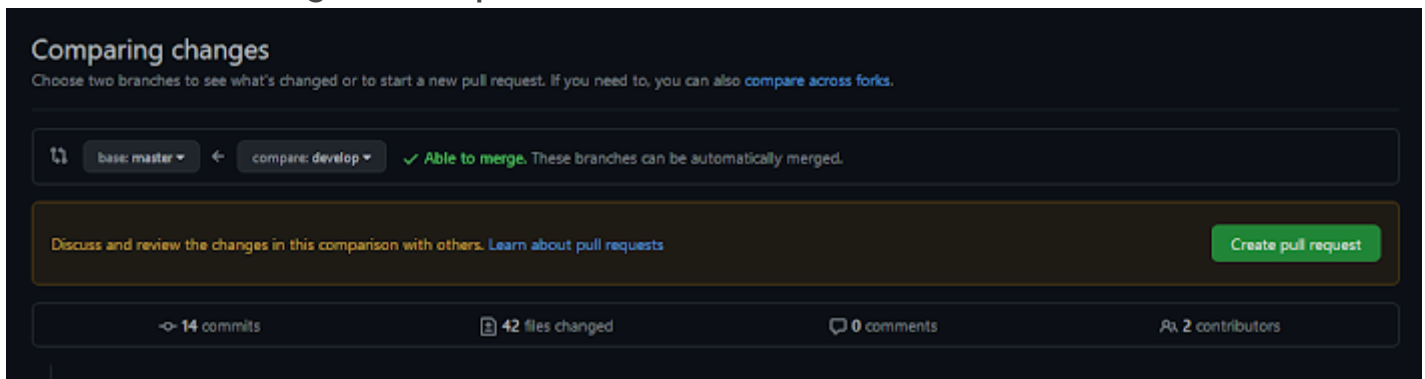
Forking a Repository

- At github.com, navigate to the repository you want to fork.
- Press the  button.



Pull Requests and Merging

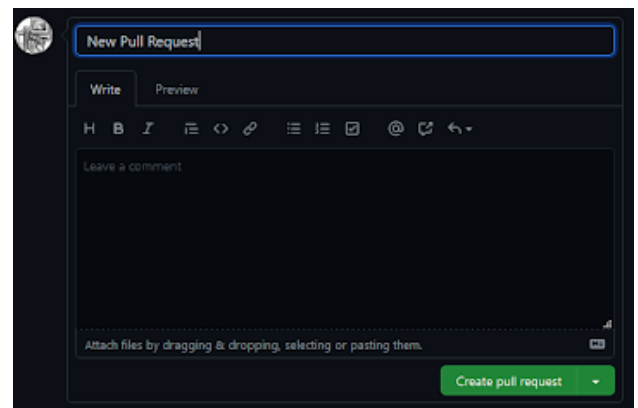
“Merging” is essentially the action of copying changes from one branch to another, “merging” it with the destination branch. A pull request is a mechanism for merging changes between branches or forks. If a programmer is done testing the code on their develop branch, they can put that code into the master branch using a pull request. The same thing goes for forks. If a programmer is happy with a new feature or fix they did, they can request to have their code merged with the code in the original repository. From there, the owner of the original repository would review their code and decide if it should be merged.

Creating a Pull Request

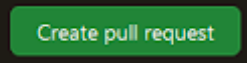



Note that in order to be able to create a pull request, you must be the owner of the repository OR own a fork of that repository.

- In github.com, navigate to the repository you want to make a pull request on.
- Click on the “Pull Requests” tab.
- Click on the  button.
- Use the dropdowns () to select which branch or fork will merge into which. The arrow between the dropdowns should



point to the branch that the changes will be copied into. If you wish to merge a branch from another fork, select [compare across forks](#).

- Select .
- Give the pull request a name and, optionally, describe the pull request to the owner of the repository.
- Select .
- If you own the repository, you will be given the option to merge the pull request. If not, GitHub will email you when your request has been either accepted or rejected.

Version Control Best Practices

- Commit messages should be kept short and sweet, while also effectively describing the changes that are being committed.
- Every repository should have a “main” branch and a “develop” branch. The main branch should only be updated when a benchmark such as a competition is reached and the code is reliable. Any code that is merged with the main branch should be thoroughly tested and debugged before merging.
- All repositories should be initialized with a readme.md, .gitignore, and license.

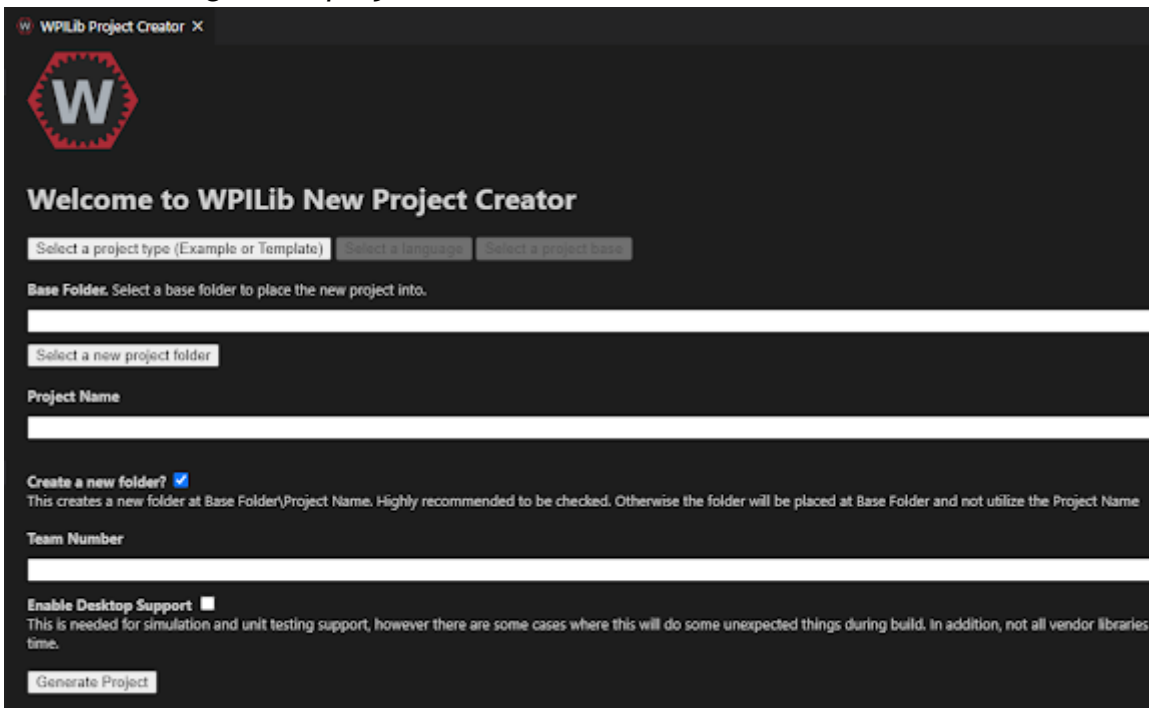
WPILib

WPILib is a code library that provides tools for programming FRC robots, as well as a common structure for robot projects. It includes useful classes such as the `Joystick` class, the `SmartDashboard` class, the `Solenoid` class, and more. It also includes a powerful command-based programming system, which is important when programming advanced robots.

Working With Robot Projects

As mentioned above, WPILib provides a common structure for your robot code. This structure is provided in the form of a robot project. Your robot project will include WPILib, your vendor libraries, and tools for building your code and pushing it to your robots. This section will guide you through how to create robot projects, import old projects, and manage existing projects. Before continuing on, ensure that you have WPILib installed on your computer.

Creating a new project

The image shows a screenshot of the 'WPILib Project Creator' application window. The window has a dark theme. At the top left is a logo with a white 'W' inside a red hexagon. Below the logo, the text 'Welcome to WPILib New Project Creator' is displayed. There are three buttons: 'Select a project type (Example or Template)', 'Select a language', and 'Select a project base'. Below these is a section for 'Base Folder' with a text input field and a 'Select a new project folder' button. The 'Project Name' section has a text input field. A checkbox labeled 'Create a new folder?' is checked, with a note below it: 'This creates a new folder at Base Folder/Project Name. Highly recommended to be checked. Otherwise the folder will be placed at Base Folder and not utilize the Project Name'. The 'Team Number' section has a text input field. At the bottom, there is a checkbox for 'Enable Desktop Support' which is unchecked, with a note: 'This is needed for simulation and unit testing support, however there are some cases where this will do some unexpected things during build. In addition, not all vendor libraries time.' At the very bottom is a 'Generate Project' button.

To create a new project, follow these steps:

- In VSCode, press ctrl+shift+p to open the command palette.

- Enter the command “**WPILib: Create a new Project**”. A form similar to the one above should appear.
- Click **Select a project type (Example or Template)**. Select “Template”, then “Java”, then “Command Robot”.
- Click **Select a new project folder**. A file dialog will appear. Click **Select Folder** to select the current folder.
- Enter the project’s name into the “Project Name” box.
- Enter your FRC Team Number (3695!) into the “Team Number” box.
- Select **Generate Project**.
- Press **Yes (Current Window)**. This loads the newly created project in the existing window. The part of the project that you will code is located at `src/main/java/frc/robot`.

Structuring the Project

Organizing a project is essential to keeping it maintainable. Many successful projects in the past have contained a set of 5 folders and 4 files. Follow these steps to structure your project.

- Navigate to your project’s robot code folder (`<project root>/src/main/java/frc/robot`).
- Ensure that your project contains the below files and folders. If any are missing, add them:
 - One folder named “**auto**”, where autonomous related classes go. Do not put commands in here. This folder is for the classes that link them together.
 - One folder named “**enumeration**”, where all `enum` objects will go.
 - One folder named “**subsystems**”, where all subsystems will go.
 - One folder named “**commands**”, where all commands will go.
 - One folder named “**util**”, where anything that does not fit into the above categories will go.
 - One file named “**Main.java**”. This is where the robot program starts. Do not, under any circumstances, modify this file.
 - One file named “**Robot.java**”. This is where the RobotContainer is defined along with the periodic and init methods.
 - One file named “**RobotContainer.java**”. This is where programmers should declare and define any subsystems and joysticks, and also where programmers should bind commands to controller buttons.
 - One file named “**Constants.java**”. This will contain important constant values including, but not limited to: motor CAN IDs, solenoid IDs, and motor inverts.

Importing an Old Project

Welcome to WPILib 2020 Project Importer

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. See [Importing a Gradle Project](#) on frc-docs for more details.

Gradle Project. Select the build.gradle file of the gradle project to import.

Select a gradle project

Base Folder. Select a base folder to place the new project into. The imported project must be placed into a new folder. Importing in place is not supported.

Select a new project folder

Project Name

When you select a gradle project above, it will insert the project name, however, it may be customized.

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

Team Number

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

Import Project

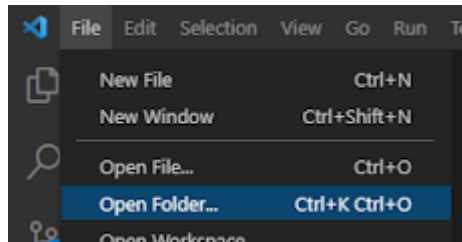
Projects from previous seasons are generally incompatible with WPILib. Programmers can use the project importer to import a project from another season into the current version of WPILib.

- In VSCode, press ctrl+shift+p to open the command palette.
- Enter the command “**WPILib: Import a 2020 Gradle Project**” for projects created after 2018 or “**WPILib: Import a WPILib Eclipse project**” for projects created in 2018 or earlier.
- Follow the form and specify the location of the existing project, the destination for your imported project, and your team number.
- Press **Import Project**.
- Press **Yes (Current Window)** to open the imported project in the existing window.
- Re-import your vendor libraries. See the “Managing Vendordeps” section for instructions.

Opening a Project Folder

To open a robot project to work on it, follow these steps:

- In VSCode, press “File” -> “Open Folder”





- Navigate into the folder in your project that contains the “src”, “vendordeps” and “.gradle” folders (it will contain other folders, but these folders are the most significant). Do not click any other folders or navigate any further.
- Press Select Folder.
- To check if your project was opened successfully, right click on any folder in the VSCode file hierarchy. One of the options should be “Create a new class/command”. If this option is not visible, then either the project was not opened correctly, or the WPILib extension is not installed in VSCode.



Managing Vendordeps

Vendor libraries, also called vendor dependencies or vendordeps, are libraries that can be added to your robot project. CTRE and REV Robotics both provide vendor libraries to assist programmers in using their motors. Without their libraries, the compiler would not be able to recognize classes such as the `TalonSRX` class or the `CANSparkMax` class. This section will guide you through how to add, remove, and update your vendordeps.

- **Adding Vendordeps**
 - Using VSCode
 - Ensure that the vendor software is properly installed on your computer. The vendor JSON file should be in the folder **C:/Users/Public/wpilib/<year>/vendordeps**. The CTRE framework installer does this automatically, but other vendors do not do this, so you will need to find their vendor JSONs and copy them into that folder.
 - In VSCode, press ctrl+shift+p to open the command palette.
 - Enter the command “**WPILib: Manage Vendor Libraries**”
 - Select “**Install new Libraries (offline)**”
 - Select your vendor dependencies from the list. If a dependency does not exist, try installing it manually.

- Press .
- Build your robot code. See “Building and Deploying Code” for instructions.
- Adding Dependencies Manually
 - Locate your vendor JSON file. Many JSON files are included somewhere in the vendor’s API download.
 - Copy the vendor JSON file and paste it into the “vendorddeps” folder in your project.
 - Build your robot code. See “Building and Deploying Code” for instructions.
- **Removing Vendorddeps**
 - In VSCode, press ctrl+shift+p to open the command palette.
 - Enter the command “**WPILib: Manage Vendor Libraries**”.
 - Select the libraries you wish to remove.
 - Press .
 - Build your robot code.
- **Updating Vendorddeps**
 - In VSCode, press ctrl+shift+p to open the command palette.
 - Enter the command “**WPILib: Manage Vendor Libraries**”.
 - Select “Check for Updates (online)”
 - When the process finishes, build your robot code.


Building and Deploying Code

Deploying or “pushing” code is the action that sends your code to the robot. To push code to the robot, follow these steps:

- **To JUST Build (not deploy)**
 - In VSCode, press ctrl+shift+p to open the command palette.
 - Enter the command “**WPILib: Build Robot Code**”
- **Build and Deploy**
 - In VSCode, press ctrl+shift+p to open the command palette.
 - Enter the command “**WPILib: Deploy Robot Code**”

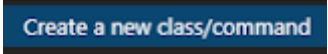
Simulating Code

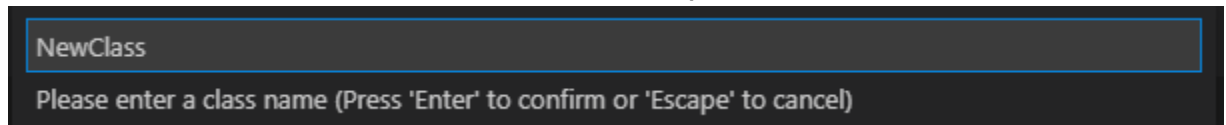
Simulating code is very useful for testing basic features or methods of the robot code without actually needing access to the robot. It is helpful to ensure that new code will not cause errors that cause the code to stop, and also that utility methods return the correct values. Follow these steps to simulate your robot code.

- In VSCode, press ctrl+shift+p.
- Enter the command “**WPILib: Simulate Robot Code On Desktop**”
- When the project builds, select the GUI you wish to use.
- Press .
- The robot project will start, and a GUI for controlling the virtual robot will appear. Using this GUI, you can enable the robot in autonomous mode, teleoperated mode, and test mode.

Creating new Subsystems, Commands, and Classes

Follow these steps to create a new subsystem, command, or empty class. See the “Command-Based Programming” section for more information on what subsystems and commands are.

- In the VSCode file hierarchy, right click on the folder that you want to create the new subsystem, command, or class in.
- Select .
- Select the type of object you want. Ensure that you are not selecting an old version of an object, which has “(old)” at the end of its name. Old classes are deprecated and can break the code.
- Enter the name of the class in the textbox and press Enter.



Using RioLog

RioLog is essentially the robot console. All warning and error messages from the robot appear here. This includes any user-created messages made using `DriverStation.reportWarning()` or `DriverStation.reportError()`. RioLog can be viewed with VSCode as well as on the DriverStation application.

- *Opening RioLog in VSCode*
 - In VSCode, press ctrl+shift+p to open the command palette.
 - Enter the command “**WPILib: Start RioLog**”

Sometimes, RioLog in VSCode will freeze up and stop reporting errors and warning messages. When this happens, it can usually be fixed by closing RioLog and restarting it. Other times, it can be fixed by power-cycling the robot. If none of those strategies work, restart the driver station computer. If restarting the computer does not work,

you may temporarily need to use the DriverStation application to view messages from the robot.

Command-Based Programming

Command-Based programming is a powerful way to add functionality to the robot because it defines a clear and easy-to-use structure for adding advanced functionality. Commands use one or multiple subsystems and run periodically until a certain task is completed. Any and all robot movement should be programmed using Commands. This section will guide you through command based programming on an FRC robot.

Basic Vocabulary

This section will describe basic terms in command-based programming, including subsystems, commands, and generic classes.

- Subsystems

A Subsystem is a physical part of the robot that serves some sort of robot function. In WPILib, a Subsystem is a class that extends the `SubsystemBase` class. How motors and solenoids and other devices are grouped into subsystems is entirely up to the programmer, but here are some examples of things that might deserve a subsystem:

- A motor, or group of motors that work together to achieve a common function, such as intaking or outputting a game piece.
- A pneumatic piston, or group of pneumatic pistons, that work together to achieve a common function.
- A group of motors and pneumatic pistons that work together to achieve a common function.
- The pneumatic compressor.
- A coprocessor such as a Raspberry Pi that sends data to the RoboRIO.

Examples of things that do **not** deserve a subsystem include:

- Any utility class whose sole purpose is to do calculations.
- Groups of subsystems.

When grouping components together into subsystems, it is important that they are grouped together by FUNCTION and not by location, size, or any other factor.

- Commands

A command achieves a task on a robot. When scheduled, they run periodically and do something until a certain condition is met, or function achieved. In WPILib, a

command is a class that extends the `CommandBase` class. Some examples of potential commands include:

- A command that uses controller input from the Driver Station computer to drive the robot around.
- A command to extend or retract a pneumatic piston.
- A command to autonomously drive the robot forwards or backwards x number of feet.
- A command that runs when the user presses a button on the dashboard that records encoder data from the drivetrain wheels and then saves it to a file on the RoboRIO.

Commands generally use a subsystem to accomplish a task. Anything that does not use a subsystem to accomplish a task can and should probably be called in one of the robot's `periodic()` methods. However, there are some exceptions to this rule, including:

- Anything that runs when the user presses a button on the dashboard or on a controller.
- Anything that should not run periodically and is important to the functionality of something else, such as:
 - A command that does nothing for 3 seconds that is used during an autonomous routine to allow an intake mechanism to finish collecting a game piece.
 - A command that records the position of the robot on the field and stores it to a file on the RoboRIO.
- **Generic Classes**

Some classes that are part of a robot project will not fit into the subsystem or command categories. If this is the case, then they can be defined as a plain class, extending nothing. Some examples of plain classes in a robot project include:

 - The Util class (Util.java) and Xbox class (Xbox.java) that are found in the “util” directory of past Foximus Prime robot projects.
 - The Hyperdrive class in the Hyperdrive library.

Requiring Subsystems

“Requiring” subsystems is an integral part of command-based programming. It prevents two commands from using the same subsystem to achieve different tasks, which can cause unpredictable behavior. For example, if a manual drive command was driving the robot around with user input, and another command started using the drivetrain to align the robot to a vision target, the robot could behave strangely and unpredictably which could be scary for the driver, operator, or anyone else that is nearby. If both commands “required”

the drivetrain, the manual drive command would be cancelled when the align command is scheduled so that only one of the commands is using the drivetrain at any given time. A command can require a subsystem, or multiple subsystems by using the `addRequirements()` method in its constructor.

Example: A command named `CyborgCommandZeroTurret` requires a `SubsystemTurret`.

```
public CyborgCommandZeroTurret(SubsystemTurret turret) {
    this.turret = turret;
    addRequirements(this.turret);
}
```

When requiring subsystems, a programmer should only require the subsystems that the command is actually going to move. Otherwise, they might find that commands end at undesired times because they are being cancelled by other commands that need the same subsystems. Examples of different commands that require (or do not require) subsystems include:

- A manual drive command that uses user input to drive the drivetrain motors should require the drivetrain subsystem only.
- A command that aligns a turret to a vision target using a “receiver” subsystem that reads vision target data from a Raspberry Pi coprocessor should require the turret subsystem, but NOT the receiver subsystem because the command never changes the receiver subsystem’s state.
- A command that shoots balls by driving a “feeder” motor that needs the “receiver” subsystem mentioned above to check if the turret is aligned with the target, as well as a flywheel subsystem to check if the turret’s flywheel is at a high enough velocity. In this case, an “align” command for the turret is already running in a separate command. The command should require the feeder subsystem but not the flywheel, turret, or receiver.
- A command that does the same thing as the example above using the same subsystems, but also extends a pneumatic piston if the flywheel’s velocity is less than 5,750 RPM, and retract the piston if the flywheel velocity is greater than 5,750 RPM. This command should require the feeder subsystem and the piston subsystem, but not the flywheel, turret, or receiver.

Command Class Layout

In WPILib, a command is simply a class that extends the `CommandBase` class. This class has 5 methods that a programmer may choose to use, but not all methods need to be filled out in order for a command to work. This section will guide you through the methods in the command class, using a command that rumbles a controller for a certain amount of time as an example.

- The command's **constructor** defines the Command's Subsystems and adds the subsystems it will drive as requirements of the command using the `addRequirements()` method.

```
public CyborgCommandRumble(Joystick controller, long length, RumbleType type) {
    this.controller = controller;
    this.length = length;
    this.type = type;
}
```

- The `initialize()` method is called once as soon as the command is scheduled to be run. This should do any setup that might be needed for the command to run successfully. For example, the `initialize()` method below sets the start time of the command to the current time.

```
public void initialize() {
    startTime = System.currentTimeMillis();
}
```

- The `execute()` method is called periodically while the command is scheduled. This is the part of the command that achieves the function that the command was created for.

```
public void execute() {
    controller.setRumble(type, 1);
}
```

- The `end(boolean interrupted)` method is called as soon as the command is cancelled or interrupted. This should be used to stop any motors that the command used so that they do not continue spinning even though the command is not running.

```
public void end(boolean interrupted) {
    controller.setRumble(type, 0);
}
```

- The `isFinished()` method is used to indicate whether or not the Command has finished its task. For any commands that run perpetually, this should always return `false`. However, in the case where the command only runs once and this method would always return `true`, the `InstantCommand` class should be used.

```
public boolean isFinished() {
    double elapsedTime = System.currentTimeMillis() - startTime;
    return elapsedTime > length;
}
```


Full Sample Command code. This command rumbles the driver/operator controllers.

```

14 public class CyborgCommandRumble extends CommandBase {
15     private Joystick controller;
16     private long
17         startTime,
18         length;
19
20     private RumbleType type;
21
22     /**
23      * Creates a new CyborgCommandRumble.
24      */
25     public CyborgCommandRumble(Joystick controller, long length, RumbleType type) {
26         this.controller = controller;
27         this.length = length;
28         this.type = type;
29     }
30
31     // Called when the command is initially scheduled.
32     @Override
33     public void initialize() {
34         startTime = System.currentTimeMillis();
35     }
36
37     // Called every time the scheduler runs while the command is scheduled.
38     @Override
39     public void execute() {
40         controller.setRumble(type, 1);
41     }
42
43     // Called once the command ends or is interrupted.
44     @Override
45     public void end(boolean interrupted) {
46         controller.setRumble(type, 0);
47     }
48
49     // Returns true when the command should end.
50     @Override
51     public boolean isFinished() {
52         double elapsedTime = System.currentTimeMillis() - startTime;
53         return elapsedTime > length;
54     }
55 }
56

```


Linking Commands to Buttons

Commands can be linked to the driver or operator's Xbox controller so that the driver and operator can press buttons or move joysticks to control the robot. There are several different methods and behaviors for linking commands to buttons. To link a command to a button, a `JoystickButton` object must be made, and then one of the following methods must be called on it:

- `whenPressed()`: Links a command such that it starts when the button is pressed.
- `toggleWhenPressed()`: Links a command such that it starts when the button is pressed, and is stopped when the button is pressed again.
- `whileHeld()`: Links a command such that it starts when the button is pressed, and is stopped when the button is released.

```
JoystickButton toggleFlywheel = new JoystickButton(OPERATOR, Xbox.START);
toggleFlywheel.toggleWhenPressed(new CyborgCommandFlywheelVelocity(SUB_FLYWHEEL));
```

An example of linking a command that drives a flywheel to the Start button on the operator's Xbox Controller, so that the flywheel is toggled every time the operator presses it. In this example, `SUB_FLYWHEEL` is the flywheel subsystem, and `OPERATOR` is a `Joystick` object.

All applicable commands should be linked to their respective buttons in the `RobotContainers` `configureButtonBindings()` method. See the 2021 robot project in the Resources chapter for an example.

Commands That Run All the Time

Some commands such as the robot's basic driving command need to run all of the time while the robot is enabled. These commands are commonly called "manual commands". Manual commands are constantly listening for user input and responding, like the steering command for the drive subsystem. To make them run all of the time, programmers should use the `setDefaultCommand()` method of the subsystem that the command requires. This will cause the command scheduler to automatically schedule that command if nothing else requires that subsystem.

```
SUB_DRIVE.setDefaultCommand(new ManualCommandDrive(SUB_DRIVE));
```

In this example, `SUB_DRIVE` is a `SubsystemBase`, and `ManualCommandDrive` is a `CommandBase`.

Because this method was called, `SUB_DRIVE` will now automatically schedule `ManualCommandDrive` when no other command requires it.

Command Types and Naming Schemes

When programming a robot, multiple different types of commands might be used. Some commands will serve an intelligent purpose, while others exist only to drive a motor

while it is scheduled. It is helpful to use a naming scheme to make a command's purpose obvious to anyone reading the code. The below list gives some command types and naming conventions for those types.

- **Button Commands**

- Button commands are commands that start when a button is pressed. The command may end when that button is released, but it does not have to. A good naming convention for these commands is for the name of the command to start with "ButtonCommand". For example, a command that drives an intake motor while a controller button is held might be called `ButtonCommandRunIntake`.

- **Toggle Commands**

- Toggle commands are commands that start when a button is pressed, and end when the button is pressed again (i.e the button "toggles" the command). The state of a ToggleCommand (running/not running) should ALWAYS be printed to the dashboard. A good naming convention for this type of command is for the name of the command to start with "ToggleCommand". For example, A command that toggles manual control over a climber mechanism might be called `ToggleCommandDriveClimber`.

- **Manual Commands**

- Manual commands are commands that run all of the time and allow the driver or operator to manually control a mechanism using the axes (joysticks or triggers; anything that is not just an on-or-off button) of their controllers. Manual commands should only require one subsystem and always be run as the default command of that subsystem. Manual commands' names typically start with the prefix "ManualCommand". For example, a command that runs all of the time and drives the robot around with human input might be called `ManualCommandDrive`.

- **Instant Commands**

- Instant Commands are commands that run and then end as soon as they are scheduled. These are particularly useful in autonomous when performing tasks such as zeroing motor encoders. Instant commands' names typically start with the prefix "InstantCommand". For example, a command that zeros all drivetrain encoders might be called `InstantCommandZeroDriveEncoders`.

- **Cyborg Commands**

- Cyborg commands are commands that perform an intelligent mechanical task and then exit. These are for functions that might be categorized as a 'driver assist'. They can be linked to buttons on the controller, but are also commonly used in autonomous routines (whereas ButtonCommands, ToggleCommands,

and ManualCommands are not). Cyborg commands commonly run some sort of PID loop, but they do not always. Their names typically start with the prefix “CyborgCommand”. For example, a command that aligns a turret to a vision target might be called `CyborgCommandAlignTurret`. Cyborg commands are particularly well suited for commands that automate/optimize a task that the driver or operator would normally perform.

- Other Commands

- Some commands that a programmer might need to make will not fit into any of the above categories. In this case, the programmer should use a prefix that makes sense to anyone reading the code. For example, three button commands (eat, spit, shoot) for controlling the intake and magazine of a shooter robot can be combined into one manual command that is named `ButtonCommandGroupRunMagazine`.

Using Commands to Achieve Basic Robot Function

Basic commands are very quick to write and achieve primitive but important tasks on the robot. They are typically linked to buttons or set as default commands for subsystems to give the driver and operator basic control over what the robot does. Some examples might include:

- A command that runs perpetually that uses a Joystick to drive the robot around.
- A command that toggles the state of a solenoid.
- A command that drives a motor at full speed until it is stopped.

Conveniently, simple commands like these can often be defined inline using the `RunCommand` or `InstantCommand` constructors, rather than defining a new class that extends `CommandBase`. An example of this is shown below. Commands defined with this inline constructor do not have an `initialize()` method, `end()` method, or `isFinished()` method (it is assumed that `isFinished()` will always return `false`). Any commands that require those three methods cannot be defined using this constructor and should have their own class.

The inline command constructor is laid out as such:

```
new RunCommand(Runnable toRun, Subsystem... requirements)
Command driveTurret = new RunCommand(() -> { SUB_TURRET.moveTurret(OPERATOR); }, SUB_TURRET);
```

An example of a basic turret driving command being declared and defined inline. The first argument of the `CommandBase` constructor is a `Runnable`, which is defined using a lambda expression. The second argument of the constructor is the `Subsystem` that the command requires.

```
InstantCommand zeroGyro = new InstantCommand( () -> { SUB_DRIVE.zeroGyro(); } );
```

An example of defining an Instant Command using the constructor. Most instant commands should be defined this way.

Using Commands to Achieve Intelligent Robot Function

Intelligent robot function on the robot should be handled by Cyborg Commands. CyborgCommands can be linked to controller buttons for user control of the robot, and can also be scheduled by other commands or events in the code. They are very commonly seen in autonomous routines. Some examples of Cyborg Commands include:

- A command that drives the robot forward 5 feet at 75% percent power.
- A command that drives a motor at 6000 RPM.
- A command that extends a piston for 6 seconds and then retracts it.
- A command that perpetually aligns a turret to a vision target.
- A command that drives a mechanism until it hits a limit switch.
- A command that lowers heavy parts to prevent tipping

Cyborg Commands will always need their own class because of the complexity of the tasks that they need to complete. It is impossible to define an intelligent command using the inline constructor.

Scheduling Commands

Commands are executed by the WPILib `CommandScheduler` class. This class runs scheduled commands by calling their `initialize()`, `execute()`, `end()`, and `isFinished()` methods at the appropriate times. Several methods for scheduling commands are available to programmers:

- **Using the `JoystickButton` class:** A command that is linked to a controller button with the `JoystickButton` class will automatically be scheduled when the controller button is pressed. Use this method for button and toggle commands.
- **Using the `Subsystem`'s `setDefaultCommand()` method:** A command that is set as a subsystem's default command will be automatically scheduled if no other command requires the subsystem. Use this method for manual commands and other commands that need to run all the time.
- **Using the `SmartDashboard.putData()` method:** Commands can be sent to the dashboard using this method. From there, The command can be displayed on the dashboard as a button. The command will be automatically scheduled when that button is clicked. This method should be used for test commands or other commands that do not need to be used during a competition match.

```
SmartDashboard.putData("Zero All Drivetrain", new InstantCommand(() -> zeroAllDrivetrain()));
```

Example of creating a dashboard button that runs a command to zero the drivetrain when pressed.

- **Using the `CommandScheduler.getInstance().schedule()` method:** Using this method, a programmer can manually schedule one or multiple commands in code.

```
//schedule a command the rumbles the driver controller for 1 second (1000 ms)
Command rumbleController = new CyborgCommandRumble(DRIVER, 1000, RumbleType.kLeftRumble);
CommandScheduler.getInstance().schedule(rumbleController);
```

Example of manually scheduling a command that rumbles the driver controller for 1 second. `CyborgCommandRumble` is the example command provided earlier in the book, in the “Command Class Layout” section.

- **Using the `Command`'s `schedule()` method:** Programmers can use this method on a `Command` to schedule it.

```
Command zeroTurret = new CyborgCommandZeroTurret(SUB_TURRET);
zeroTurret.schedule();
```

Example of manually scheduling a command that zeros a turret.

Chaining Commands Together

Some robot functions will require a command to run directly after another command finishes. WPIlib makes programming this kind of function easy with the help of sequential command groups and parallel command groups. A sequential command group is a group of commands that run in order; once the first command ends, then the second begins, when the second command ends, then the third begins, and so on. A parallel command group is a group of commands that run at the same time. This section will guide you through how to use WPIlib to create parallel command groups and sequential command groups. All of the below methods will return a `Command` that can be scheduled, linked to a button, or used in other command groups.

- **Using `andThen()`:** This method chains commands together to form a sequential command group. The command that `andThen()` was called on will be executed first, then the first argument, then the second argument, and so on.

```
Command zeroTurret = new CyborgCommandZeroTurret(SUB_TURRET);
Command positionTurret = new CyborgCommandSetTurretPosition(SUB_TURRET, 2500, 3000);
Command alignTurret = new CyborgCommandAlignTurret(SUB_TURRET, SUB_RECEIVER);

Command zeroThenPositionTurret = zeroTurret.andThen(positionTurret, alignTurret);
```

Example of chaining three commands together with `andThen` (the 3 commands involved are custom commands). When `zeroThenPositionTurret` is scheduled, it will zero a turret, THEN move it to a certain position, THEN align it to something.

- **Using `alongWith()`:** This method chains commands together to form a parallel command group. All commands passed into the method will be run at the same time, and the command group will only end when all commands are finished. Commands used in this method must all require different subsystems or else an `Exception` will be thrown.

```
Command driveForwards = new CyborgCommandDriveDistance(SUB_DRIVE, 100, 1);
Command positionTurret = new CyborgCommandSetTurretPosition(SUB_TURRET, 3000, 4000);

Command driveAndPositionTurret = driveForwards.alongWith(positionTurret);
```

In this example, the command `driveAndPositionTurret` will drive the robot forward a certain distance while moving the turret to a certain position. It will end after both of those tasks have been completed.

- **Using `raceWith()`:** This method chains commands together to form a parallel command group. All commands passed into the method will be run at the same time, but the command group will end, and all commands will be stopped, as soon as the first command finishes. Similar to `alongWith()`, commands passed into this method must all require different subsystems, or an exception will be thrown.

```
Command driveBackwards = new CyborgCommandDriveDistance(SUB_DRIVE, -100, 1);
Command runIntake       = new ConstantCommandDriveIntake(SUB_INTAKE, SUB_FEEDER);

Command driveWhileRunningIntake = driveBackwards.raceWith(runIntake);
```

In this example, the command `driveWhileRunningIntake` will drive the robot backwards a certain distance while running the intake (`runIntake` never ends). As soon as `driveBackwards` is finished, `runIntake` will stop.

Command Best Practices

Here are some guidelines to command-based programming.

- All subsystems that a command uses (even the ones that do not get passed to `addRequirements()`) should be private variables of the class.

```
public class CyborgCommandZeroTurret extends CommandBase {
    private SubsystemTurret turret;

    /**
     * Creates a new CyborgCommandZeroTurret.
     */
    public CyborgCommandZeroTurret(SubsystemTurret turret) {
        this.turret = turret;
        addRequirements(this.turret);
    }
}
```

- Commands should not contain any motor objects. The motor objects should strictly remain in their subsystems, and the subsystems should have methods that change their percent output, position target, velocity target, PID constants, etc.

- The constructor of the command should be used only to define the private subsystem variables and call `addRequirements()`. Any other sort of initialization goes in the `initialize()` method. This includes initialization of any PID controllers, objects, etc.
- Any motors being driven by a command should have their percent output set to 0 in the `end()` method. Doing this will cancel any currently running PID loops and ensure that the motor does not keep driving even though the command is not running.
- If a command's `isFinished()` method is set to always return `true`, then the command should be replaced with an `InstantCommand`.

Troubleshooting

While programming the robot, you will come across a wide range of errors from a few different places. You can receive errors from the compiler (called Gradle), the Driver Station, and the robot code itself. This section will guide you through some of the different errors you may encounter, different methods of fixing them, and useful strategies for debugging other errors in your robot code.

Common Gradle Errors

- “Compilation Error!”

This error happens when there is some sort of syntax problem in the code that causes it to not compile. The error should be underlined with red squiggly marks in VSCode. Most of the time, this error is easy to find and fix. Other times, it may be a little more difficult because the error itself might be incorrect. Below are some possible symptoms of this type of error and ways to fix it:

Symptom	Possible Fixes
Some or all WPILib classes or methods not being found	<ul style="list-style-type: none"> - Ensure that you have the WPILib New Commands vendordep (WPILibNewCommands.json) installed.
Some or all vendor classes or methods not being found	<ul style="list-style-type: none"> - Ensure that the latest vendor JSON file is installed. - In VSCode, update all vendordeps and reload the window.
Recently added classes or methods not being found	<ul style="list-style-type: none"> - This is likely caused by a Java Intellisense engine error. Follow these steps to fix it:

	<ul style="list-style-type: none"> - In VSCode, press ctrl+shift+p to open the command palette. - Enter the command “Java: Clean Java Language Server Workspace” - When prompted in the lower right hand corner, press Restart and delete. - VSCode will quickly reload the window, but it may take a minute or two for intellisense to come back. If intellisense is taking a while to come back, reload the VSCode window using the command “Developer: Reload Window”.
--	--

- **“Are you connected to the robot, and is it on?”**

This is caused by a lack of communication between the RIO and the computer. Check to see if the FRC Driver Station application has communication with the robot (The “Comms” indicator is green).

Comms?	Action to take
No	Follow the steps for solving the “No Robot Communication” error in the “Common Driver Station Errors” section.
Yes	This issue is likely caused by outdated software on either the RoboRIO or the computer. Follow the steps in the Maintenance chapter of this book to update both the robot and the computer. Make sure no conductive metal shavings are in/near the RoboRIO’s pins.

- **“Dependency Error!”**

This happens when the compiler is missing software from a vendor library and cannot download it. To fix the error, connect to the Internet and build your robot code. If the error persists, you may be behind a proxy server or firewall

that does not allow Gradle to download the dependencies that it needs to. This is common on school and work networks. Do the following to fix it:

- Connect the computer to a home network or wireless hotspot. Note that large downloads might not be reliable on hotspots so you may need to bring the computer home to download the dependencies.
- In the VSCode terminal, enter the command `./gradlew downloadAll`.
 - If there is no terminal, press ctrl+shift+` (yes, that is a hyphen) to open one.

```

29
30 // Called every time the scheduler runs while the command is scheduled.
31 @Override

TERMINAL  PROBLEMS 22  OUTPUT  DEBUG CONSOLE

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS D:\_Users\Brach\projects\FRC\InfiniteRecharge-2020-develop\InfiniteRecharge-2021> ./gradlew downloadAll
  
```

- After the downloadAll command finishes, try building your code normally. The build should now succeed.

Common Driver Station Errors

- Watchdog Not Fed

This message appears on the FRC Driver Station Application when the main robot loop has not iterated for a while. This is caused by some part of the code blocking the execution of the robot loop. This is commonly caused by:

- `while` Loops not running in a separate thread
- Conditionless `for` loops not running in a separate thread
- Sockets trying to receive data (not on a separate thread)
- Anything else in the main robot loop that blocks execution of the code





To fix this issue, look through your code and try to find and delete any of the above items.

- No Robot Communication

This happens when the robot is not communicating with the computer. Ensure that the following conditions are met:

- The robot is on, and the router has both booted up and is displaying healthy status lights.
- The RoboRIO is booted up and is displaying healthy status lights (power LED green, status LED off, comms LED green, RSL LED on or blinking)

- The RoboRIO is connected to the router and the ethernet status lights are on.
- The computer is connected to the robot's wi-fi network
- The computer's wi-fi adapter is using default IP settings. To check this, do the following:

- Click on the  button and type "View Network Connections".
- Open the  option.
- Open your wi-fi adapter properties.
- Open your   Internet Protocol Version 4 (TCP/IPv4) properties.
- Ensure that your IP address and DNS server address is being set automatically.

If all requirements are satisfied, but the error persists, reset the RoboRIO by either power-cycling the robot (turning it off and back on again) or pressing the blue reset button. If the problem persists after that, re-flash the router.

- No Robot Code

This happens when there is no code running on the robot. Do the following:

Was the RoboRio...	Action to take
Recently re-imaged?	Push your code to the robot.
Recently reset?	The RoboRIO is most likely finishing its bootup process. Your code should start soon.
None of the above?	An unhandled exception in your code caused it to stop. If RioLog was running when the exception occurred, it will have a full message and stack trace on the exception. Fix the code that caused the exception and push the fixed code to the robot.

Common RioLog (robot code) errors

- **"Robots should not quit, but yours did!"**
Some unhandled exception occurred while executing the robot code. Scroll up through your RioLog messages to find the exception that occurred.
- **Errors with "CAN" in them (CAN Errors)**
CAN Errors are caused by the motor controllers connected to the CAN network. They will likely be some sort of CAN timeout error, or CAN version error. This

might be an electrical issue, but check the following things before getting the electrical team to fix it:

Condition	Action to take
Could the motor have amped out? (do the errors happen while driving the motor, and do they stop after 5-10 seconds?)	Set an amp limit on the motor controller. You may need to discuss with the electrical team to determine what the amp limit should be. If the error occurred while running a PID loop, decrease the P and the I significantly.
Do the motor controllers have outdated firmware?	Update the firmware.

Other Robot Code Issues

- **Scheduled Commands are not running**

Ensure that the `Robot` class' `robotPeriodic()` method is calling `CommandScheduler.getInstance().run()`. That method call is what actually executes the scheduled commands.

Common Robot Router Issues

- **Robot Router Booted and Healthy, but no Wi-Fi Network is visible**

To fix, re-configure your router. If you were just at competition, then the router may still be using the competition configuration, which disables the Wi-Fi network. If this does not work, re-flash your router.

- **Radio Configuration Tool “Cannot find bridge”**

Ensure that the robot radio is directly connected via Ethernet to the computer and nothing else.

- **Radio Configuration Tool “Failed to set IP Address”**

Ensure that the ethernet adapter is having its IP address and DNS server name set automatically by viewing and editing the Ethernet adapters properties in Windows.

- **Radio Configuration Tool gives error “Error Finding NPF device name for adapter”**

Likely due to a bad ethernet adapter on the computer. Ensure that the ethernet adapter is not set to use a static IP address or DNS server by viewing your network adapter properties. If the adapter is having its IP address automatically set and this error still occurs, then the adapter is probably just bad. Use a different computer.

Debugging Tips and Techniques

The following techniques are helpful to use when diagnosing and resolving problems in your robot code:

- **Using the WPILib Desktop Simulation Tool or the robot's test mode**
 - If a utility method is not working these tools are very helpful because a programmer can run the code and debug it without actually moving the robot. When using the desktop simulation tool, access to the robot is not needed at all.
- **Taking pictures or videos of the dashboard while the robot is running**
 - This technique is helpful for debugging any function that requires the robot to move. Programmers can send values to the dashboard and take videos of them while the robot is moving to be able to see what they are and how they change at any point in time. While doing this, they can potentially uncover erroneous values that cause a command to work incorrectly.
- **Doing the math on paper**
 - This technique is useful for debugging utility methods that return erroneous values, especially if the method is written to do a complex physics or mathematics expression. If a utility method returns a wrong value for an unknown reason, do the math that it is supposed to do by hand, and see if your answer matches the one that the method gave. Doing this could uncover forgotten negative values or small errors in the math that could have led to the erroneous value.
- **Unit Testing**
 - This technique is also useful in conjunction with the previous one. Using the WPILib desktop simulation tool or the robot's test mode, set up several small tests for a utility method whose results you have calculated by hand. Running the tests may reveal some situations where the code works and others where it does not. This can provide insight into what the problem may be.
- **System.out.println() (simulation tool ONLY) and DriverStation.reportWarning()**
 - Use `System.out.println()` and `DriverStation.reportWarning()` to print out rapidly to the console. Note that `System.out.println()` will only work if using the simulation tool. When using `DriverStation.reportWarning()`, be mindful that the `DriverStation` class has a maximum rate at which it is allowed to send messages. Exceeding that rate will cause other messages to be dropped.

Robot Code Best Practices

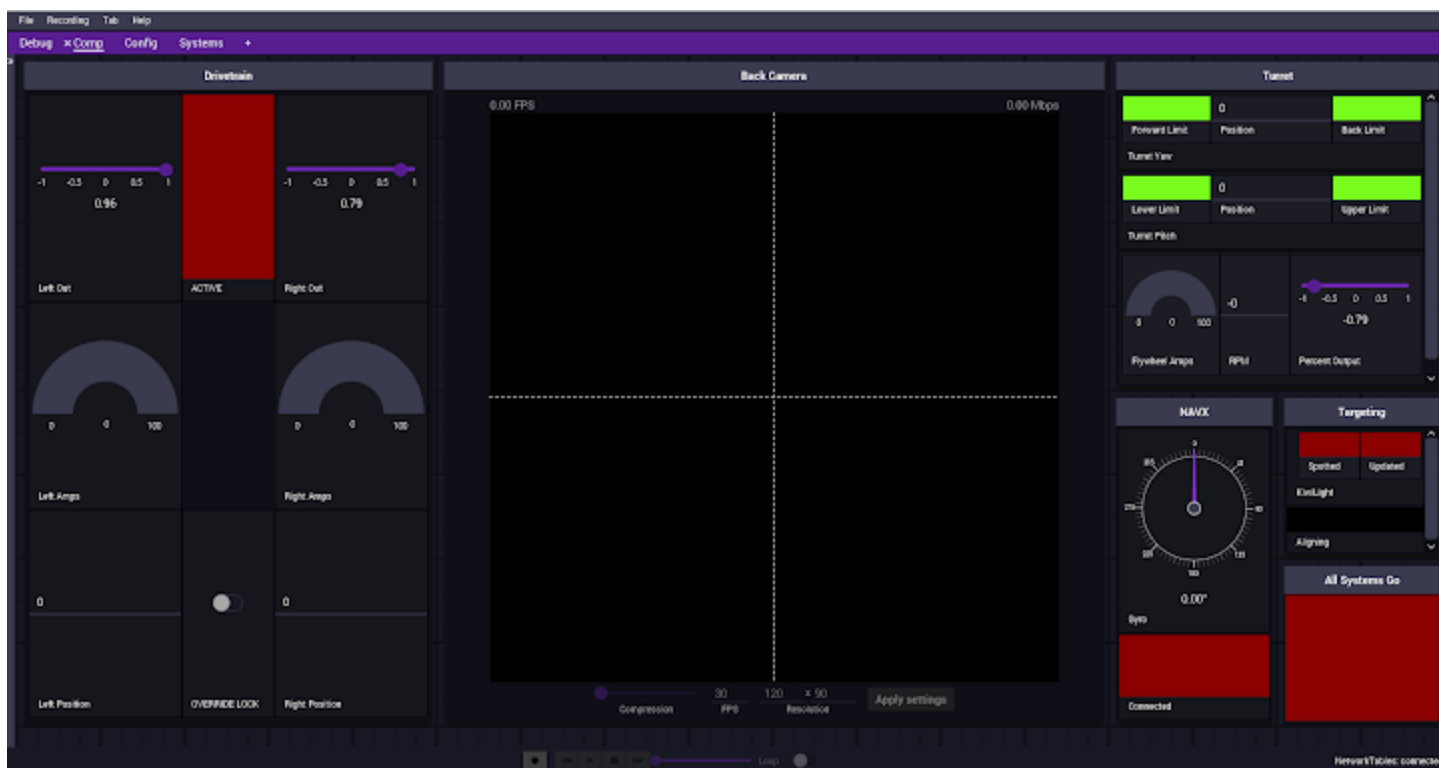
The guidelines outlined in this chapter should be followed in every robot project you write. They are based on lessons learned by past programmers in past FRC seasons. Many of these rules relate to keeping your code maintainable, which is vital to robots that consistently perform well at competition. Code that is “maintainable” can be quickly and easily fixed if it breaks.

- **Do NOT Modify any auto-generated code with the exception of comments and method bodies.** All auto-generated methods (such as the `isFinished()` method of a `Command`) should be kept even if they are unused, so that it is clear that they exist and programmers can easily use them in the future. However, it is okay to delete the contents of methods of classes such as `Robot` and `RobotContainer` as long as the methods themselves are not deleted. It is also acceptable to delete the `ExampleSubsystem` and `ExampleCommand` classes that come with new projects.
- **Do NOT implement useless functions into the robot code.** All code that exists on the robot must be specifically related to completing the season challenge in some way, whether it be an auto-alignment command, drive subsystem, test command, or something similar. So not program the robot to play battleship because that is a waste of your time and the teams' time.
- **Avoid using infinite loops in the main robot loop.** `while(true)` loops and `for(;;)` loops in the main robot loop could cause the main loop to freeze if no break statement is reached. This would cause a “Watchdog Not Fed” error.
- **NEVER use `Thread.sleep()` or any other sleep method in the main robot loop.** Doing so would cause the robot to freeze similar to how infinite loops would.
- **If a motor must be inverted (forwards and backwards directions flipped), use the inversion method in the motor controller class.** Store your inversion values in the `Constants` class in your robot code. Multiplying output values by -1 to invert a motor's direction is inconsistent and too difficult to maintain. NEVER invert the motor at any time outside of robot startup.
- **All code running on the robot should be as simple, efficient, elegant, and easy to read as possible.** If you find yourself copying and pasting code, then it is usually possible to create a method that achieves the functionality of the code that you are copying and pasting. At the same time, do not make a piece of code look more complicated just so it takes up less space. One line `if` statements are short, but multi-line statements with curly brackets are generally easier to look at.

- **Put all constant values in the `Constants` class.** If you have a constant value that is used in multiple different places, this makes it easy to change that value for all places. This includes values such as motor IDs, solenoid IDs, motor inverts, field dimensions, etc.
- **Comment your code.** Any method that you write should have a javadoc above it that describes what it does, the parameters it takes, and what it returns. Any class should have similar comments describing the purpose of the class. Any code within a method body that cannot “speak for itself” should have a comment describing its purpose and, if you have enough space, how it achieves that purpose. This makes it easier for other programmers to read and edit your code. However, this documentation should be kept as short and easy to read as possible to keep the code maintainable.
- **Take it easy on the CAN.** The CAN is a two-wire network that has a lot of important traffic going through it. NEVER reference encoder positions or velocities outside of the main robot loop because encoder requests can overwhelm the CAN, leaving little room for traffic that sets motor speeds. If you feel that the motors are not being responsive enough, or being jittery for no good reason, try to reduce the number of times that an encoder position or velocity is being referenced.
- **KEEP YOUR CODE MAINTAINABLE!** Whenever writing or editing a piece of code, always honestly ask yourself: “if this code breaks at competition, will I be able to diagnose and fix it quickly?” The difference between a 1 minute fix and a 20 minute fix can be the outcome of a match.

The Dashboard

The dashboard displays information about the robot and is helpful for debugging, driving, operating, and match analysis. It can display numbers, graphs, booleans, buttons, and video streams. The common dashboard application used for FRC is called “WPILib Shuffleboard”. This chapter will guide you through Shuffleboard’s features and how to use them.



The 2021 Robot Dashboard.

Launching Shuffleboard

The WPILib Shuffleboard application can be launched three different ways:

- It will be automatically launched when the FRC Driver Station application is launched if Shuffleboard is set as the default dashboard application.
- If WPILib is installed on your computer, Shuffleboard can be launched with a desktop icon that is placed on the desktop by the WPILib installer.
- It can be opened with VSCode by following these steps:
 - In VSCode, press ctrl+shift+p to open the command palette.
 - Enter the command “**WPILib: Start Tool**”

- Select “Shuffleboard”

Sending and Displaying Data


In your robot code, sending data to the dashboard is done using the `SmartDashboard` class. This class contains multiple different methods that allow you to send numbers, booleans, subsystems, commands, gyros, and other things to the dashboard to be displayed. The methods for sending different data types to the dashboard are outlined in the table below. Each method in the table takes two arguments:

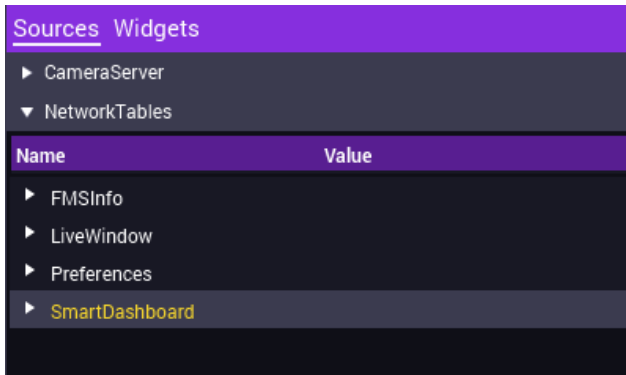
- **key**: The name of the value to send. This parameter is always a `String`. The value sent with the method will appear in the dashboard with this name.
- **value**: The actual value to send. The type depends on the method being used.

Data Type	Method to use
<code>double</code>	<code>SmartDashboard.putNumber(String key, double value)</code>
<code>boolean</code>	<code>SmartDashboard.putBoolean(String key, boolean value)</code>
<code>String</code>	<code>SmartDashboard.putString(String key, String value)</code>
<code>Sendable</code> (this includes <code>Subsystems</code> , <code>Commands</code> , gyro objects, and other things that extend the <code>Sendable</code> class.)	<code>SmartDashboard.putData(String key, Sendable value)</code>

Displaying data

Once a value has been sent to the dashboard using one of the `SmartDashboard` methods, the following steps can be taken to display it.

- In Shuffleboard, select the  button in the upper left corner of the window and make sure that you are viewing `Sources`.
- In the “NetworkTables” dropdown, search for the name of the value you wish to display. To make it easier to find, collapse all fields (FMSInfo, LiveWindow, Preferences, etc). Values are located in the “SmartDashboard” field.



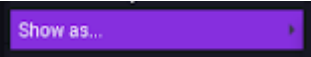
- Drag and drop the value you wish to display onto the dashboard.



- To rename your widget, double-click on the name at the top of the widget, type in the new name, and press enter. This will not affect the name used by the `SmartDashboard` method. Values will still be sent to that widget even if it has a different name than the source does.

The “Show As...” Feature

Once displayed, values on the dashboard can take on multiple different visual forms. These can make the dashboard more functional or better looking. To change the visual form a widget is being displayed as, follow these steps:

- Right-click on the widget that you wish to change.
- Hover over  and select the visual form you want the widget to take.

Note that graphs are very resource intensive and should be used for debugging only. Using a graph will slow down the update times of the dashboard to the point where it may be unusable. Do NOT put a graph in a competition dashboard.

The Preferences Table

Parameter	Value
Auto Expose Camera	<input checked="" type="checkbox"/>
Auto Start Offset	0
Camera Exposure	0
Camera FPS	22
Camera Height	128
Camera Width	252
Drive Inhibitor	1
Drive PID Ramp	0.5
Drive Ramp	0.25
Drive Velocity IZone	0
Drive Velocity kD	0
Drive Velocity kF	0
Drive Velocity kI	0
Drive Velocity kP	0
Drive Velocity Out Limit High	1
Drive Velocity Out Limit Low	-1
Drive Velocity Setpoint	12
Drivetrain IZone	0
Drivetrain kD	0
Drivetrain kF	0
Drivetrain kI	0
Drivetrain kP	0

Path	Name	Value
NetworkTables	FMSInfo	
FMSInfo	EventName	
FMSInfo	FMSControlData	33.0
FMSInfo	GameSpecificMessage	
FMSInfo	IsRedAlliance	true
FMSInfo	MatchNumber	0.0
FMSInfo	MatchType	0.0
FMSInfo	ReplayNumber	0.0
FMSInfo	StationNumber	1.0
LiveWindow	Ungrouped	
Scheduler	Ids	[]
Scheduler	Names	[]
Preferences	Some Value	42.0
SmartDashboard	InstantCommand	
running		false
Value		42.0

Shown above: The Preferences Table (left), and the location of the Preferences Table among the other sources (right).

Shuffleboard allows the user to input numbers, strings, and booleans for the robot to use on the fly without the need to even disable it. This functionality is achieved with the Preferences table. In Shuffleboard, the Preferences table can be found under “NetworkTables” as seen in the picture above.

In code, you can use the `Preferences` class to both get and set values on the table. The `Preferences` class works very closely to a `HashMap` in that it stores values with a `String` name, and the values can be accessed later with the same name.

It is recommended to have a util method that handles getting values from the Preferences table. Team 3695 has used the following two methods in their `Util` class to get `doubles` and `booleans` from the Preferences table and set them to a backup value if they do not exist:

```
public static double getAndSetDouble(String key, double backup) {
    if(!Preferences.getInstance().containsKey(key)) Preferences.getInstance().putDouble(key, backup);
    return Preferences.getInstance().getDouble(key, backup);
}
```

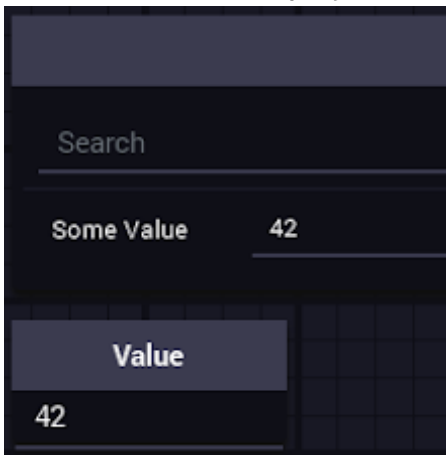
- The `Util.getAndSetDouble()` method returns a `double` by the name `key`. If that value does not exist, it is set to `backup`.

```
public static boolean getAndSetBoolean(String key, boolean backup) {
    if(!Preferences.getInstance().containsKey(key)) Preferences.getInstance().putBoolean(key, backup);
    return Preferences.getInstance().getBoolean(key, backup);
}
```

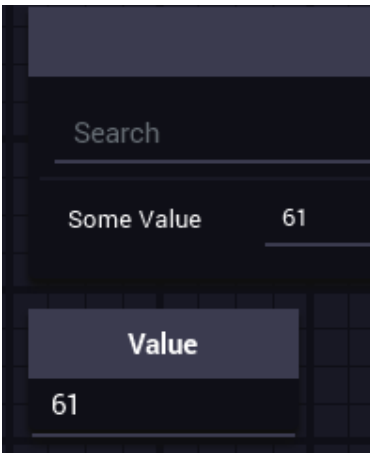
- The `Util.getAndSetBoolean()` method returns a `boolean` by the name `key`. Similar to `Util.getAndSetDouble()`, a backup value is used if the key does not exist.
- As an example, let's say `someMethod()` grabs a `double` named "Some Value" from the Preferences table and sends it to the dashboard as a readout:

```
private void someMethod() {
    double value = Util.getAndSetDouble("Some Value", 42);
    SmartDashboard.putNumber("Value", value);
}
```

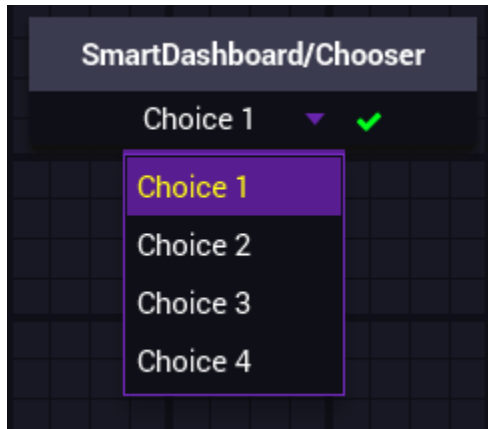
When `someMethod()` is called, "Some Value" appears in the table with a value of 42 and the "Value" readout displays the same value.



In the table, "Some Value" is now changed to 61. When `someMethod()` is called again, the "Value" indicator will now read 61:



Drop-Downs



Another method of user input similar to that of the Preferences table is the `SendableChooser`. This is a drop-down menu for the dashboard that can be used to allow a user to select driving schemes, autonomous routines, and more. This section will guide you through how to create and use `SendableChoosers`.

- Start by declaring your `SendableChooser`. In this example, we will create a `SendableChooser` in the `RobotContainer` class. Similar to an `ArrayList` or `HashMap`, a `SendableChooser` can be used with multiple different types. This example will use `Integers`.

```
private SendableChooser<Integer> chooser;
```
- In the constructor for your class, initialize the sendable chooser.

```
chooser = new SendableChooser<Integer>();
```
- Use the `setDefaultOption(String name, T object)` method to set the default option of your drop-down. In this example, the default option will be named “Choice 1”, and have a value of 1. The object must have the same type that the `SendableChooser` was initialized with. For example, if the `SendableChooser` was defined as a

`SendableChooser<Boolean>`, you add options with values of `true` or `false` instead of integers like 1, 2, 50, or 42.

```
chooser.setDefaultOption("Choice 1", 1);
```

- Use the `addOption(String name, T object)` to add any other options you wish for the drop-down to have. In this example, we will define 3 additional options:

```
chooser.addOption("Choice 2", 2);
chooser.addOption("Choice 3", 3);
chooser.addOption("Choice 4", 4);
```

- Finally, send your chooser to the dashboard:

```
SmartDashboard.putData("Chooser", chooser);
```

- So far, our class looks like this:

```
/**
 * 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
 * periodic methods (other than the scheduler calls). Instead, the structure of the robot
 * (subsystems, commands, and button mappings) should be declared here.
 */
public class RobotContainer {
    private SendableChooser<Integer> chooser;

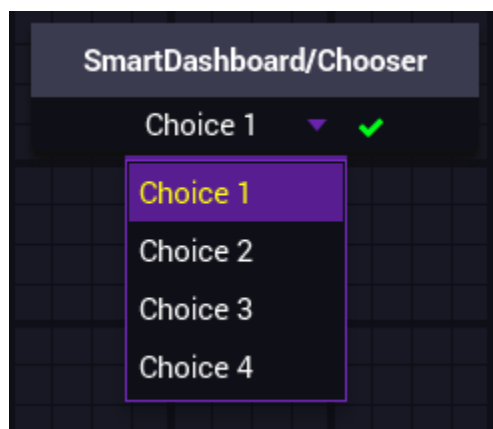
    /** The container for the robot. Contains subsystems, OI devices, and commands. */
    public RobotContainer() {
        // Configure the button bindings
        configureButtonBindings();

        //put a button on the dash that runs someMethod()
        SmartDashboard.putData(new InstantCommand( () -> { someMethod(); } ));

        //define and configure chooser
        chooser = new SendableChooser<Integer>();
        chooser.setDefaultOption("Choice 1", 1);
        chooser.addOption("Choice 2", 2);
        chooser.addOption("Choice 3", 3);
        chooser.addOption("Choice 4", 4);

        //put chooser on the dashboard
        SmartDashboard.putData("Chooser", chooser);
    }
}
```

- When you run the code, you should be able to display your drop-down on the dashboard.



- To query what option is selected by the user at any time, use the `getSelected()` method. This method will return the value of the currently selected option (NOT the name).

```
int value = chooser.getSelected();
```

- For this example, we defined `someMethod()` to display the selected value on a separate indicator:

```
private void someMethod() {
    int value = chooser.getSelected();
    SmartDashboard.putNumber("Value", value);
}
```

- When `someMethod()` is called, the “Value” indicator reads out the value of the choice that is selected.



Full example code on the next page.

```

public class RobotContainer {
    private SendableChooser<Integer> chooser;

    /** The container for the robot. Contains subsystems, OI devices, and commands. */
    public RobotContainer() {
        // Configure the button bindings
        configureButtonBindings();

        //put a button on the dash that runs someMethod()
        SmartDashboard.putData(new InstantCommand( () -> { someMethod(); } ));

        //define and configure chooser
        chooser = new SendableChooser<Integer>();
        chooser.setDefaultOption("Choice 1", 1);
        chooser.addOption("Choice 2", 2);
        chooser.addOption("Choice 3", 3);
        chooser.addOption("Choice 4", 4);

        //put chooser on the dashboard
        SmartDashboard.putData("Chooser", chooser);
    }

    private void someMethod() {
        int value = chooser.getSelected();
        SmartDashboard.putNumber("Value", value);
    }

    /**
     * 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() {}

    /**
     * Use this to pass the autonomous command to the main {@link Robot} class
     *
     * @return the command to run in autonomous
     */
    public Command getAutonomousCommand() {
        // An ExampleCommand will run in autonomous
        return null;
    }
}

```

Camera Streams

Adding video streams for the driver and operator to use is a feature that is very important for driver/operator performance. Typically, video from the robot is provided by a Microsoft Lifecam HD-3000 camera plugged into one of the RoboRIO's two USB ports. However, any USB camera can be used. Follow these steps to add video to a dashboard.

- In your robot project, create an empty class where video will be handled. In this example, the class will be called `CameraHub`.
- In your class, declare (but do not define) a `private UsbCamera`. In this example, it will be called `camera`.

```
private UsbCamera camera;
```

- Add a constructor and a method where camera settings will be set. In this example, the method will be called `configureCameras()`.
- So far, the class should look like this:

```
1 // Copyright (c) FIRST and other WPILib contributors.
2 // Open Source Software; you can modify and/or share it under the terms of
3 // the WPILib BSD license file in the root directory of this project.
4
5 package frc.robot;
6
7 import edu.wpi.cscore.UsbCamera;
8
9 /** Add your docs here. */
10 public class CameraHub {
11     private UsbCamera camera;
12
13     public CameraHub() {
14
15     }
16
17     private void configureCameras() {
18
19     }
20 }
21
```

- In the constructor, send an `InstantCommand` to the dashboard to configure camera settings. This way, it is quick and easy for drivers to revert to default camera settings.

```
SmartDashboard.putData(new InstantCommand( () -> { configureCameras(); } ));
```

- Define the camera as an automatic video stream using `CameraServer.getInstance().startAutomaticCapture(int device)`. `device` is the index of the camera, starting at 0.


```
camera = CameraServer.getInstance().startAutomaticCapture(0);
```

- Set the resolution of the camera using the `UsbCamera`'s `setResolution(int width, int height)` method. The width and height of the resolution should be constant values in the `Constants` class. For bandwidth purposes, the resolution should be the smallest possible resolution supported by the camera.

```
camera.setResolution(Constants.CAMERA_RESOLUTION_WIDTH, Constants.CAMERA_RESOLUTION_HEIGHT);
```

- Finally, call your camera configuration method.
- The constructor should look like this:

```
public CameraHub() {
    SmartDashboard.putData(new InstantCommand( () -> { configureCameras(); } ));

    camera = CameraServer.getInstance().startAutomaticCapture();
    camera.setResolution(Constants.CAMERA_WIDTH, Constants.CAMERA_HEIGHT);
    configureCameras();
}
```

- That is all that needs to happen in the constructor. Now let us turn our attention to configuring our camera settings.
- In your camera configuration method (ours is called `configureCameras()`), you will need to configure the camera's exposure, brightness, and FPS. Some settings like exposure can be set to automatic, but in competition it may be better for the drivers to have their video fine-tuned for the field's lighting conditions. At the end of the day, this is up to the drivers to decide. In this example, camera settings will be constants, but they can be set from the Preferences table as long as they are within a try-catch loop (`VideoExceptions` will be thrown for any invalid settings)
- Camera exposure can be configured to be automatic or manual. If you want the exposure to be automatic, simply call the `UsbCamera`'s `setExposureAuto()` method. If you want the exposure to be manual, then call the `setExposureManual(int value)` with a value from 0 (lowest exposure) to 100 (highest exposure). In this example, we will have both options:

```
//set camera exposure
if(Constants.CAMERA_AUTO_EXPOSE) {
    camera.setExposureAuto();
} else {
    camera.setExposureManual(Constants.CAMERA_EXPOSURE);
}
```

- To configure brightness, use the `setBrightness(int brightness)` method with a value from 0 (lowest brightness) to 100 (highest brightness).

```
//set camera brightness
camera.setBrightness(Constants.CAMERA_BRIGHTNESS);
```

- To configure FPS, use the `setFPS(int fps)` with the fps you wish to set the camera to.

```
//set camera FPS  
camera.setFPS(Constants.CAMERA_FPS);
```

- After completing your configuration method, add javadocs to your class and methods for maintainability.

Full example code is below. The constant values used in this example were:

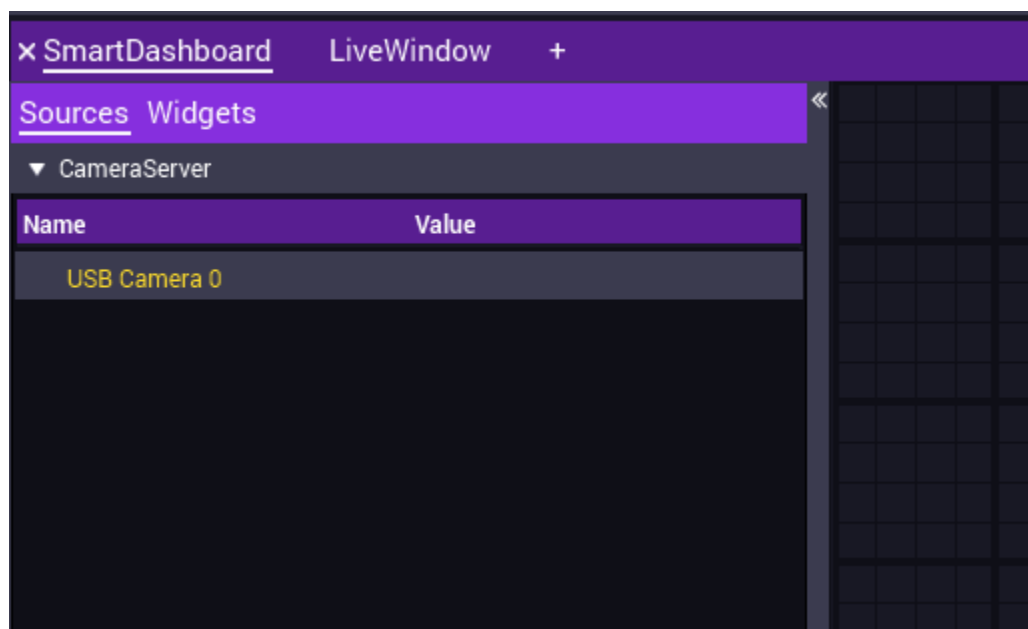
Name	Type	Value
<code>Constants.CAMERA_WIDTH</code>	<code>int</code>	160
<code>Constants.CAMERA_HEIGHT</code>	<code>int</code>	120
<code>Constants.CAMERA_AUTO_EXPOSE</code>	<code>boolean</code>	<code>false</code>
<code>Constants.CAMERA_BRIGHTNESS</code>	<code>int</code>	50
<code>Constants.CAMERA_FPS</code>	<code>int</code>	30

```

1  // Copyright (c) FIRST and other WPILib contributors.
2  // Open Source Software; you can modify and/or share it under the terms of
3  // the WPILib BSD license file in the root directory of this project.
4
5  package frc.robot;
6
7  import edu.wpi.cscore.UsbCamera;
8  import edu.wpi.first.cameraserver.CameraServer;
9  import edu.wpi.first.wpilibj.smartdashboard.SmartDashboard;
10 import edu.wpi.first.wpilibj2.command.InstantCommand;
11
12 /** A utility that streams video from the robot's driver cams. */
13 public class CameraHub {
14     private UsbCamera camera;
15
16     /**
17      * Creates and starts a new CameraHub.
18      */
19     public CameraHub() {
20         SmartDashboard.putData(new InstantCommand( () -> { configureCameras(); } ));
21
22         camera = CameraServer.getInstance().startAutomaticCapture();
23         camera.setResolution(Constants.CAMERA_WIDTH, Constants.CAMERA_HEIGHT);
24         configureCameras();
25     }
26
27     /**
28      * Configures all camera settings from values on the Preferences table.
29      */
30     private void configureCameras() {
31         //set camera exposure
32         if(Constants.CAMERA_AUTO_EXPOSE) {
33             camera.setExposureAuto();
34         } else {
35             camera.setExposureManual(Constants.CAMERA_EXPOSURE);
36         }
37
38         //set camera brightness
39         camera.setBrightness(Constants.CAMERA_BRIGHTNESS);
40
41         //set camera FPS
42         camera.setFPS(Constants.CAMERA_FPS);
43     }
44 }
45

```

- Now, simply declare and define a `CameraHub` in your `RobotContainer` and run the code. The video stream that you created can be found in the “CameraServer” section of the “Sources” tab in Shuffleboard.



- Drag and drop the `USB Camera 0` to display the video stream on your dashboard.



- As you can see, the video stream is taking up quite a lot of bandwidth (1.46 Mbps). Keep in mind that in 2019, the bandwidth limit for all 6 teams on the field was 4 Mbps. This means that the bandwidth is going to need to come down quite a bit. There are a few ways to do this.
 - **Reduce the FPS.** You can do this by changing the value used in the `setFPS(int fps)` method or using the FPS field on the video widget.
 - **Compress the image.** Use the Compression slider on the video widget to adjust how much an image is compressed when it is sent to the dashboard. 30 is a good value to start at.
 - **Decrease the resolution.** The resolution set with the `setResolution(int width, int height)` method should already be set to the smallest possible resolution your camera can support, but it can be further reduced with the Resolution fields on the image widget. However, this method will cause some of the video's color and definition to be lost.
- By just reducing FPS to 20 and setting a compression level of 30, the bandwidth of the video stream was reduced from 1.46 Mbps to 0.4 Mbps. When configuring your video stream at competition, use these methods effectively to reduce your bandwidth usage as much as possible.

Common Dashboard Issues

This section will run through issues you may encounter while creating, managing, or using your dashboard as well as strategies to fix those issues.

Problem	Potential Fix
Shuffleboard becomes very slow and resource-intensive when rendering graphs or video streams.	To start, delete any graphs or video streams that are not immediately needed at the moment. If Shuffleboard is still slow and resource-intensive, delete the existing graphs and video streams and put them back into the dashboard.
Shuffleboard takes a while to start up.	Ensure that Shuffleboard is up to date and that the correct version of Java is installed on the computer.
NetworkTables is connected, but some of the widgets are greyed out.	This is a bug in Shuffleboard. Delete the greyed-out widget and put it back on the dashboard. If the widget is still greyed-out, make sure that the robot is actually sending that value. For example, an indicator may be greyed out because

	<code>SmartDashboard.putNumber()</code> has not been called for that key yet.
Shuffleboard does not connect to NetworkTables.	Ensure that the FRC DriverStation application is open and has communication with the robot.

Dashboard Best Practices

These are some common dashboard best practices to create and maintain good dashboards:

- **In your competition dashboard, keep things minimal.** The less there is to display, the less there is to slow Shuffleboard down. This will ensure that the Driver Station computer has enough processor power and memory to control the robot. At the same time, you should still display important data such as important encoder readings, such as important motor RPMs, limit switch states, and drop-downs.
- **Group readouts together by subsystem.** This will make the dashboard easy to look at and important indicators easy to find.
- **Use Layouts.** Shuffleboard offers a feature called a layout that can group indicators together. This helps tremendously with dashboard organization, especially when grouping readouts together by subsystem. (To create a layout, right click anywhere inside of the dashboard and select “Add Layout”)
- **If the robot has a camera connected to the RIO, the video stream MUST be on the competition dashboard.**
- **Use multiple dashboards.** You should create a “Competition” dashboard with all of the important readouts and careful organization, and a “Debugging” dashboard where you can use any indicator you would like. This helps to ensure that changes made to the debugging dashboard will not make the competition dashboard any slower or harder to look at.

Basic Robot Control

One of the most important things that you can program on an FRC robot is basic control over all of the robot's systems. It is important for the drivers to be able to manually control *all* of the robot's systems in case any autonomous control of a system fails. This chapter will guide you through adding basic robot control to your robot.

The “Driver-Operator” Scheme

Most FRC robots have several systems that need to be controlled during competition. Most of the time, the controls for all of these systems cannot all be fit onto only one controller. Luckily, FRC allows each team to have two people on the field controlling their robot. To optimize robot control given these conditions, almost every FRC team out there uses a Driver-Operator control scheme. That is, one person “drives” the robot around and another person “operates” all of its other systems. This way, the job of controlling the robot during a match is split among two people; The driver only has to focus on getting the robot around the field, and the operator takes care of intaking and placing game pieces and performing other tasks.

Generally speaking, the driver's controller should only be able to control systems related to the drive system. The only exceptions to this are simple controls that would slow down the driver by forcing them to communicate frequently (like toggling a clamp). All other controls should be assigned to the operator to better distribute the workload. After all, the operator isn't being distracted by positioning the bot, so their abilities can be utilized to operate the robot's other systems.

Configuring the System being Controlled

Before programming control over a system, it is important that the system is configured correctly. If a system is not configured correctly, even good control schemes may become unusable. Generally, all components that you will program on the robot will have a set of methods that you can use to configure them.

Configuring Motors

When configuring a system's motors, keep the following items in mind:

- **Do the motors need to be in braking mode?** Braking mode is a mode that is offered by most (if not all) motor controllers. When set to this mode, motors will actively resist movement that they are not causing. If you are programming a system that

should not coast after it is driven, then the motor should be set to braking mode.

Examples of motors that should be set to braking mode include:

- All drivetrain motors. Failing to set braking mode on the robot's drivetrain motors would cause the robot to keep moving even if the triggers are not pressed. This would make 100+ pound robots very hard to drive and very easy to crash.
- Most mast systems. Masts, or other systems affected by gravity, will fall quickly if power is not applied to the motors. Braking mode will make the systems fall at a safer speed.

Examples of motors that probably do not need to be set to braking mode include:

- Flywheels. Because flywheels exist to retain angular momentum, it is usually a waste of power to actively stop them.

See the documentation for your motor controller for information about what method to use to set the motor to braking or coasting mode.

- **What should the motor's amp limit be?** Amp limits are useful for making sure that the motor does not work harder than it is supposed to. Using these can help protect both motors as well as motor controllers. Small motors may only need 20-30 amp limits, while bigger motors (NEOs, CIMs, and Falcon-500s) may need up to 60 or 70 amp limits.
- **Does the motor need a ramp rate?** Ramp rates are really good for making motors drive more smoothly. Ramp rates are especially helpful for the drivetrain motors. If a system is hard to control smoothly, then a ramp rate should be considered.
- **Does the motor need to be inverted?** Remember, inverts should be stored as booleans in the `Constants` class and applied using the motor controllers inversion method.

Configuring Solenoids

Configuring solenoids is much simpler than configuring motors. The only item that needs to be configured prior to use is the state that the solenoids should be configured to. Simply set the solenoid to be opened or closed based on what the behavior of the system should be upon robot startup.

Controllers

Several different controller options are available for controlling FRC robots. The type of controller used will depend on the systems being controlled and the ease-of-use of the controller itself. This section will guide you through some common controllers and how to connect and use them on your FRC robot.

Xbox Controller

Xbox controllers are the most popular type of controller for controlling FRC robots. They are familiar to a lot of people and typically have more buttons and axes than other controllers, making them a safe pick for almost any robot being controlled. Currently, wired Xbox One and Xbox 360 controllers are supported by the FRC Driver Station application. No bluetooth or other wireless controllers are supported.



Logitech Attack Joystick

Logitech attack joysticks are big flight simulator joysticks. They are good for controlling systems such as turrets, and great for drive styles such as true-tank drive. However, they are less commonly used than Xbox controllers so they may feel unfamiliar to some drivers.



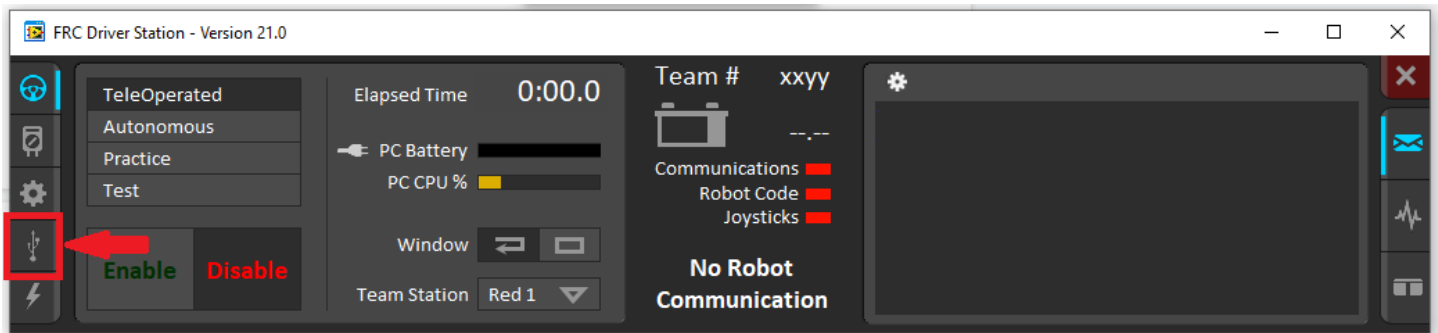
Pick the controller(s) that you will use to control the robot early in the season. As mentioned earlier, Xbox controllers are usually a safe pick because of how common and robust they are. However, if you think that another controller may be a better option, consult members of the team

who might be potential drivers or operators and get their opinion before committing to the other controller.

Using Controllers to Control the Robot

Using your controllers to control your FRC robot is easy: simply plug the controller into one of the USB ports on your Driver Station computer, and references to them will be provided by WPILib in the form of the `Joystick` or `XboxController` classes. Those classes can be used to access most or all of the controller's axis and button states. See WPILib's documentation of those classes for more information about how to access axis and button states.

When defining a `Joystick` or related controller class, the constructor will require an ID value to specify which controller is being referenced. If Xbox controllers are being used, the driver will typically get the ID 0 controller and the operator will typically get the ID 1 controller. However, the ID scheme of the controllers is entirely up to the programmers. The IDs of the controllers connected to the Driver Station computer can be viewed and changed in the "USB Devices" tab of the FRC Driver Station program.



Location of the "USB Devices" tab in FRC Driver Station. In this tab, you can view the controller layout and change the IDs of connected controllers using a drag and drop interface.

Driving Styles for Tank-Style Robots

Perhaps the most important form of basic robot control is driving the robot around from place to place. The preferred style of control used to drive a tank-style robot may vary from driver to driver, and programmers should be prepared to program several different "drive modes" as different potential drivers request them. This section lists a few common drive styles used to control tank-style robots.

Rocket League Style

This drive system is commonly used by the team, as it's useful for most competitions and is simple to tune. This drive style uses an Xbox controller to drive the robot around. The right trigger accelerates the robot forward, the left trigger accelerates the robot backward, and the X-axis of the left joystick turns the robot left and right. This scheme has a "subtractive" throttle, meaning that the throttle (forward-backward value) is equal to

right trigger value – left trigger value. This drive style is based on the controller scheme used by the game Rocket League. The most noticeable difference from other styles such as Forza style is that the turn rate is not throttle-dependent, allowing for zero-radius turning. This may feel less familiar, but it takes much better use of the advantages afforded by tank drive.



Below is a simple code implementation of a Rocket-League-Style drive. Note that the `xbox` class is not a commonly-available class and is being used for illustration.

```

/**
 * Drives the robot using human controller input
 * @param joystick The controller to use.
 */
public void driveManually(Joystick joystick) {
    //get throttle and steering values
    double throttle = Xbox.RT(joystick) - Xbox.LT(joystick); //right trigger minus left trigger
    double steering = Xbox.LEFT_X(joystick); //left joystick, X-axis

    //calculate left and right percents based on throttle and steering
    double leftDrive = throttle + steering;
    double rightDrive = throttle - steering;

    //make sure leftDrive and rightDrive are between -1 and 1
    leftDrive = ( leftDrive < -1 ? -1 : ( leftDrive > 1 ? 1 : leftDrive ) );
    rightDrive = ( rightDrive < -1 ? -1 : ( rightDrive > 1 ? 1 : rightDrive ) );

    //optionally inhibit leftDrive and rightDrive here to make the robot drive slower.

    //set motor controller percent outputs
    leftMaster.set(leftDrive);
    leftSlave.set(leftDrive);
    rightMaster.set(rightDrive);
    rightSlave.set(rightDrive);
}

```

Forza Style

This style is very similar to Rocket League Style. It uses an Xbox controller, the right trigger accelerates forward, the left trigger accelerates backward, and the left joystick's X-axis turns the robot left and right. However, in this drive style, the robot will only turn if throttle is being applied. The turn radius is controllable through a dashboard preference, so this parameter can be tuned to the driver's preference.

Importantly, the rotational control is affected by the throttle direction. When throttle is reversed, the steering is also reversed as it would be when reversing a car. This may make driving more intuitive to drivers that play games with driving (Rocket League, GTA, Forza, etc). This comes at the cost of maneuverability though, because zero-radius turns aren't achievable without setting a very low turn radius.

True-Tank Style

This style uses two Logitech Attack Joysticks which each directly control one side of the drivetrain; the right joystick's Y-axis drives the robot's right-side wheels forward or backward, and the left joystick's Y-axis drives the robot's left-side wheels forward or backward. This drive style is similar to that of an actual tank. However, it is less commonly used than both of the above drive styles because it is unfamiliar to people and takes some time to get used to.



Selecting Between Drive Styles

Because there may be multiple potential drivers who may want multiple drive styles, it is smart to allow them to choose between the drive styles using a drop-down (a.k.a `SendableChooser`) on the dashboard. However, great care should be taken when allowing users to select between drive styles that require different controller layouts. In some situations, an axis (such as the right trigger of an Xbox controller) may actually read 100 percent if a controller other than an Xbox controller (such as a Logitech Attack Joystick) is plugged in. That being said, plugging in Attack joysticks and forgetting to switch the drive style from rocket league to true-tank may cause the robot to drive forward at 100 percent power when it is enabled. If implementing drive styles that have differing controller layouts, a safety should also be implemented that prevents the robot from driving if the wrong controllers are plugged in. Methods for reading controller types are provided by the `DriverStation` class.

Tuning a Drive Style

The same way that video games allow the user to tweak input settings, all user inputs should have scalar constants modifiable through the dashboard. This allows the driver to tune the sensitivity of their controller and parameters of a drive system. For example, the final motor outputs should always be multiplied by an “inhibitor” that can be modified in the Preferences table by the user. That way, users can easily make the robot slower if needed.

All magic numbers/scalars used in the drive system should be tuneable. Communicate the tunable parameters of the selected drive system to the driver, so that they can refine their own configurations.

Programming Non-Drivetrain Systems

Every system on the robot capable of moving requires some thought into how best to control it. All systems should first be programmed for manual control to test them, but in a match, the way in which a motion system is controlled has a great effect on driver/operator performance. Programming non-driving systems on the robot can be as simple as driving a motor while a button is held or using a manual command to control a motor using a controller's triggers.

Determining how to control a system

Depending on the game being played, similar systems may be controlled in very different ways. For example, the 2018 and 2019 games both involved placing game pieces at high heights, so robots typically needed extending masts. However, in 2019, it was common to use a PID loop to extend the mast to certain heights because robots needed to place objects at those exact heights, whereas in 2018, no such PID loop was needed because although game pieces needed to be placed on an elevated surface, the elevation of the surface varied and in some cases, the game pieces could simply be dropped into place.

When determining how to control a system on the robot, speak with the team to determine what method of control would be the most intuitive and easy to use. Try to minimize or eliminate all potential sources of user error. Controls given to the operator should be robust, simplifying mentally demanding tasks so that they are not likely to become a source of failure. Start with a simple control scheme and gradually make it more efficient. The “best” control style of a system may vary depending on the task that the system is supposed to achieve, but generally follows these guidelines:

- Motors that do not need to be modulated (they either drive at 0% or x%) should be controlled using a Button Command or Toggle Command.
- Motors that need to be modulated should be manually controlled using a Manual Command or automatically controlled using a CyborgCommand.
- Pneumatic pistons should typically be controlled using either a Toggle Command to switch the position or a ButtonCommand to set it to a specific position.
- Motors that switch between known positions should be controlled using buttons to switch between the target positions and a PID loop to maintain that target position (if the time needed to implement such a system is available).
- Linear actuators should typically be position-controlled rather than speed, current, or percent output-controlled. The position they are set to can typically be controlled using buttons similar to the previous item.
- Flywheels should typically be controlled with a ToggleCommand. They should be speed-controlled using a PID loop if their exact momentum is task-critical (such as in shooting a known distance).

Deciding what to map a control to

Controllers only have so many buttons and axes, so it is important that all robot controls are mapped efficiently and in a way that makes sense. When deciding what to map a control to, first determine how that system should be controlled. If you are unsure, these suggestions usually have worked pretty well:

- The A, B, X, and Y buttons on a controller can be used to control systems that manipulate game pieces. The systems must either be toggled or run while a button is held.
- Driver assists and important toggles (extending intake mechanisms, running flywheels, etc) can be mapped to the left and right bumpers. If those have been used, then the START and BACK buttons are good options.
- For systems with motors that need to be modulated, there are two options:
 - Systems that require fine control should be mapped to the triggers. Drivers and operators can achieve much finer motor control with the triggers as compared to the joysticks.
 - Systems that do not require fine control should be mapped to the joysticks. Systems that move horizontally should be mapped to the X-axis of the joysticks, while systems that move vertically should be mapped to the Y-axis.

Once the control style of the system has been determined, you can now select what specific button or axis to map the control to. Try to map the system to a button or axis that feels comfortable and natural for anyone controlling it. If you are unsure, speak with potential drivers and operators and get their opinion.

Documenting Controls

As you map controls to buttons and axes on a controller, they should also be documented in a place that the team can easily access, in a way that the team can understand. That way, if the team needs to know how to control the robot, they do not need the programmers present to figure it out.

Integrating Controls into the Dashboard

While the driver typically only uses the dashboard for the cameras, it is much more critical for the operator. All values that are readable and affect the ability to perform critical tasks should be on the dashboard. Being quickly interpretable should be the highest priority for dashboard design in this regard. For example, if a boolean value needs to be checked frequently during a match, it should be large and color-based rather than text-based so as to minimize the time spent interpreting it. If multiple indicators with similar appearance are used, they should either be spaced apart or well labelled and communicated to avoid reading the wrong value.

Temporary Control System Modifiers

Each year, the tasks the drivers have to work toward mastering change dramatically and a drag-and-drop drive style might be disadvantageous during specific tasks. During these tasks, a toggle can be added to the driver controls to temporarily modify the way the robot behaves or is controlled. It is always important to add a dashboard indicator if such a system is implemented to show if the modifier is enabled or not. Drive system modifiers have the potential to enhance a drivers ability, but a driver not understanding the system or when it's on can cost them a match. A few ways to implement temporary modifiers are listed in this section.

Modifying a Constant

If a drive style can simply be tuned for a specific task, modifying a constant such as an inhibitor to change its behavior is a more robust option. Mapping a ToggleCommand to a button on the driver's controller allows you to implement a modifier. Examples of different modifiers include:

- Slowing the linear or rotational speed of a motor for a task that requires delicate movement
- Increasing the power or current limit of the drive motors temporarily to push bots more *aggressively*
- Fast and slow speed modes for fields that have both long and short sprints
- Setting a tuned speed modifier for endgame/climb

Switching to Another Control Style

If dramatically different control modes are needed within the same match, a ToggleCommand should be added to change which drive style is used.

Creating a CyborgCommand

If a task's efficiency is bottlenecked by the time it takes for the driver to perform it, and an automated system can greatly reduce this time, adding a driver-assist using a Cyborg Command might be the best option. For example, if lining up to a target costs the driver time, a toggling vision-based aligner may improve their performance. However, Cyborg Command should only be added if the programming team determines that writing and testing the command is a good use of time.

Autonomous Functions

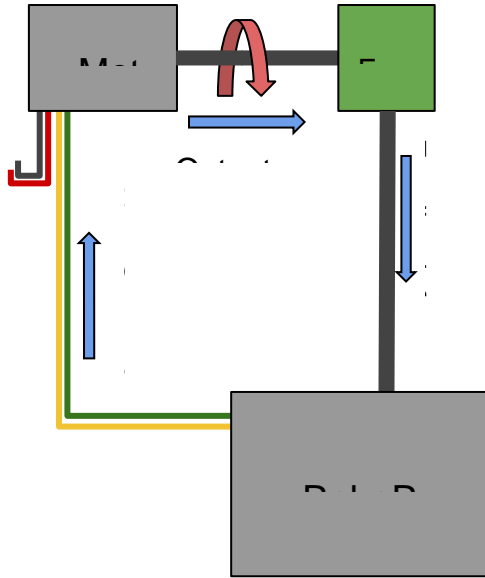
Autonomous functions are perhaps one of the most intimidating parts of FRC programming. As the programmer, you will need to code autonomous functions into almost every seasonal robot that the team builds. These autonomous functions include, but are not limited to target alignment using computer vision, motor position or velocity control, and even autonomous driving. Many FRC games feature a 15 second fully autonomous period at the beginning of every match. This period usually rewards more points per game piece scored along with extra points for alliances that can collectively perform an autonomous action. This chapter will guide you through how to program autonomous functions into your robot including smart motor control, vision targeting, autonomous driving, and putting all of those together into an autonomous routine.

Smart Motor Control with PID Loops

PID loops drive motors to precise locations or velocities and are essential to any kind of smart robot control. This section will explore what a PID loop is and how to use it in the robot code.

What is a Feedback Loop?

A feedback loop is a continuous cycle where something such as a motor is driven based on the position or state of something else, such as an encoder. The motor device, while being driven, changes the position of the encoder device which then changes the output of the motor device. The devices do not necessarily need to be motor and encoder devices. They can also be heating/cooling and thermometer devices, or a motor changing the state of a vision system. Below is a diagram of a feedback loop on the robot that drives a motor.



What is a PID loop?

A PID loop is a type of feedback loop that uses a smart algorithm to drive a motor or something else to a certain destination, or *setpoint*, as quickly as possible. They can be used to achieve almost any kind of smart movement on the robot. As an illustration of the problem that a PID loop solves, suppose that you have an oven that you want to heat to 360 degrees Fahrenheit.

You start the oven, and the heating coils turn on and get very hot. The oven warms quickly, and as soon as the internal temperature of the oven hits 360 degrees, the coils turn off. The problem is that the coils are still hot, so they will continue to give heat to the oven and the internal temperature might actually peak at 363 or 364 degrees before beginning to fall back down to 360. Eventually, the oven's internal temperature drops to 359 degrees and the coils turn back on to heat it back to 360. However, now that the coils are cool, they need some time to heat back up before the internal temperature starts rising again. The temperature might actually drop to 358 degrees before this happens.

This oscillation will keep happening at smaller and smaller magnitudes until the temperature is finally at a nice, somewhat steady, 360 degrees. A PID loop is designed to predict behavior such as that of the oven and control its output to compensate for it. For example, in the oven situation, a PID loop might have turned the coils off at 357 degrees and let the excess heat from the coils heat the oven the rest of the way.

PID loops are very useful in robotics for things like motor control. Because of principles such as momentum, the position of a motor can change even if no power is being applied. A well-tuned PID loop will be able to predict that momentum and drive the motor to an exact position or velocity.

The P, I, D, and F calculations

A PID loop gets its name from the calculations that it relies on (P, I, and D). Sometimes, an additional calculation, F, is also included. Each calculation uses one constant that the user can tune and changes the behavior of the PID loop in a unique way.

Calculation	Purpose
P - Proportional $P = kP * error$	The P, or <i>proportional gain</i> of a PID loop is calculated proportionally to the amount of error (<i>actual position – desired position</i>) at that exact moment in time. This means that the farther away a device is from its setpoint at the moment, the harder the motor will work to get there. It is very useful for making gross corrections for setpoints that are very far away. This calculation can be tuned by modifying the <i>kP</i> value of the PID Controller.
I - Integral $I = kI * \sum error$	The I, or <i>integral gain</i> of a PID loop is proportional to the sum of all of the past error ($\sum error$). With an <i>I</i> gain, the longer a device is not at its setpoint, the harder it will work to get there. That being said, it should only be used to achieve fine corrections, and should always be used with an IZone (see next table), or else the PID will cause unpredictable motor behavior. This calculation can be tuned by modifying the <i>kI</i> value of the PID controller.
D - Derivative $D = kD * \Delta error$	The D, or <i>differential gain</i> of a PID loop is proportional to the instantaneous change in error at the moment ($\Delta error$). It works to stop a motor to prevent it from overshooting the setpoint. D is almost never needed for FRC applications, but it is useful if you are programming a device that absolutely cannot overshoot its setpoint. This calculation can be tuned by modifying the <i>kD</i> value of the PID controller.
F - Feed-Forward $F = kF$	The F, or <i>feed -forward</i> is only used on PIDs running on board motor controllers. It is a constant value that is added to the output of a PID loop after it is calculated. It is usually used to counteract known forces acting on a system, such as gravity on an arm or mast. On motor controllers, this value can be tuned by modifying the <i>kF</i> value. For PID controllers on the RIO, simply add a constant value to the output of the controller.

The total output of a PID controller at any given time is simply $output = P + I + D + F$. In addition to the P, I, D, and F calculations, other notable values are available to the programmer to tune:

Value	Purpose
IZone	The IZone, or <i>Integral Zone</i> is the maximum amount of error at which the I calculation takes effect. As an example, if a PID controller has an IZone value of 200 ticks, and the current error is at 253 ticks, the value returned by the I calculation will be 0, and so whatever <i>kI</i> value is set will not take effect. Conversely, if the error is at 169 ticks, then the I calculation will have an effect on the PID loop's behavior. This value should be used by any PID loop for which the I is non-zero. Without it, any <i>kI</i> value that is greater than 0 could cause unpredictable behavior.
Output Limits	The Output Limit sets the maximum allowed percent output that a motor can use while running a PID loop. It is useful for inhibiting the speed at which a PID loop can drive a motor.
Ramp Rate	The ramp rate is the amount of time that it takes the motor controller to throttle from 0 to 100 percent. A longer time means smoother PID control. However, longer ramp rates will lead to less responsive PID loops.

Motor Controller PIDs

Most of the PIDs you will need can likely be done on the motor controller controlling the motor. Doing a PID on the motor controller is the best way to do a PID because the motor controller can do the PID calculations as fast as it can while the RoboRIO is limited by the watchdog timer, which dictates that the robot loop will run at about 50 Hz.

- CTRE Motor Controller PIDs

- The following table summarizes commonly used methods for running a PID loop on a TalonSRX or TalonFX. All methods are called on a `TalonSRX` object named `motor`.

Function	Method
Set kP value	<pre>motor.config_kP(int slot, double kP)</pre> <p><code>slot</code> is the ID of the PID slot to use. Multiple slots can be configured for different purposes; position PID constants might have one slot, and velocity PID constants might have another. Different PID slots do not necessarily need to be used if PID constants are being set in the <code>initialize()</code> method of your PID</p>

	<p>command, so for simplicity, <code>0</code> is a good slot ID to use. All other <code>slot</code> arguments in this table serve the same function as this one.</p> <p><code>kP</code> is the new <i>kP</i> value.</p>
Set kI value	<pre>motor.config_kI(int slot, double kI)</pre> <p><code>kI</code> is the new <i>kI</i> value to use.</p>
Set kD value	<pre>motor.config_kD(int slot, double kD)</pre> <p><code>kD</code> is the new <i>kD</i> value to use.</p>
Set kF value	<pre>motor.config_kF(int slot, double kF)</pre> <p><code>kF</code> is the new <i>kF</i> value to use.</p>
Set IZone value	<pre>motor.config_integralZone(int slot, double iZone)</pre> <p><code>iZone</code> is the new <i>IZone</i> value to use.</p>
Set Allowable Error Value	<pre>motor.configAllowableClosedLoopError(int slot, int allowableError, int timeoutMS)</pre> <p><code>allowableError</code> is the new allowable error (the range of error at which the PID loop will not work at all) to use.</p> <p><code>timeoutMS</code> is the number of milliseconds to wait before applying the value.</p>
Set Maximum Output	<pre>motor.configClosedLoopPeakOutput(int slot, double percentOut)</pre> <p><code>percentOut</code> is the new maximum percent output value to use.</p>
Set Setpoint or Drive Value	<pre>motor.set(ControlMode mode, double setpoint)</pre> <p><code>mode</code> is the method of control to use. Potential values include, but are not limited to:</p>

	<ul style="list-style-type: none"> - <code>ControlMode.PercentOutput</code>: Used for manual driving. Sets the motor output to a percentage of its full power. Does not use a PID loop. - <code>ControlMode.Position</code>: Uses a PID loop to try to reach and maintain an encoder position. The setpoint should be in encoder ticks. - <code>ControlMode.Velocity</code>: Uses a PID loop to try to reach and maintain an encoder velocity. The setpoint should be in encoder ticks per 100 milliseconds. - <code>ControlMode.Current</code>: Uses a PID loop to try to reach and maintain an amp draw. <p><code>setpoint</code> is the new setpoint.</p>
Set PID Slot	<pre>motor.selectProfileSlot(int slot, int pidIdx)</pre> <p><code>pidIdx</code> can be one of two values: <code>0</code> for primary closed loop, <code>1</code> for auxiliary closed loop.</p>
Set PID Ramp Rate	<pre>beater.configClosedloopRamp(double rate);</pre> <p><code>rate</code> is the number of seconds it takes for the motor to go from 0 to full power.</p>

- REV Motor Controller PIDs

- The following table summarizes the commonly used methods for running the PID loop on a motor controller made by REV Robotics, such as a Spark Max. All methods are called on a `CANSparkMax` object named `motor`.

Function	Method
Set kP Value	<pre>motor.getPIDController().setP(double p, int slot)</pre> <p><code>p</code> is the new <i>kP</i> value.</p> <p><code>slot</code> is similar to the argument on the CTRE motor controllers. See the CTRE method table above for the full explanation of this argument.</p>

Set kI Value	<pre>motor.getPIDController().setI(double i, int slot)</pre> <p><code>i</code> is the new <i>kI</i> value.</p>
Set kD Value	<pre>motor.getPIDController().setD(double d, int slot)</pre> <p><code>d</code> is the new <i>kD</i> value.</p>
Set kF Value	<pre>motor.getPIDController().setFF(double ff, int slot)</pre> <p><code>ff</code> is the new <i>kF</i> value.</p>
Set IZone Value	<pre>motor.getPIDController().setIZone(double ize, int slot)</pre> <p><code>ize</code> is the new <i>IZone</i> value.</p>
Set Output Limits	<pre>motor.getPIDController().setOutputRange(int lower, int upper)</pre> <p><code>lower</code> is the minimum percent output, from <code>-1.0</code> to <code>1.0</code>. <code>upper</code> is the maximum percent output, from <code>-1.0</code> to <code>1.0</code>.</p> <p>As an example, the call <pre>motor.getPIDController().setOutputRange(-1, 1)</pre> would configure the largest possible output range. The motor would be allowed to drive full backward and full forward. Conversely, the call <pre>motor.getPIDController().setOutputRange(0, 1)</pre> would allow the motor to drive full forward, but no magnitude of backwards power would be allowed.</p>
Set Setpoint	<pre>motor.getPIDController().setReference(double reference, ControlType type)</pre> <p><code>reference</code> is the new setpoint.</p> <p><code>type</code> is the type of control that the PID loop should achieve. Potential values include, but are not limited to:</p> <ul style="list-style-type: none"> - <code>ControlType.kPosition</code>: Uses the PID loop to try to reach the motor position designated by the setpoint. The setpoint should be in motor rotations.

	<ul style="list-style-type: none"> - <code>ControlType.kVelocity</code>: Uses the PID loop to try to reach the motor velocity designated by the setpoint. The setpoint should be in motor rotations per second. - <code>ControlType.kCurrent</code>: Uses the PID loops to try to reach the amp draw designated by the setpoint. <p>Additionally, an overload of this method exists where the programmer can specify which PID slot to use to achieve the setpoint:</p> <pre>motor.getPIDController().setReference(double reference, ControlType type, double slot)</pre> <p><code>slot</code> is the PID slot that the motor controller should use to achieve the setpoint which was set by this method call.</p> <p>NOTE: <code>ControlType</code>s such as <code>kSmartVelocity</code> and <code>kSmartPosition</code> are not documented in this book. Whenever I tried using them, I could not get them to work. More research is needed on these control types before they are used.</p>
Set PID Ramp Rate	<pre>scissors.setClosedLoopRampRate(double rate);</pre> <p><code>rate</code> is the number of seconds for the motor to go from 0 to full power.</p>

Motor Controller PID Example

As an example of a motor controller PID on a robot, suppose that your robot has a scissor lift that is driven by a NEO motor. You wish to write a PID command that would move the scissor lift motor to a certain position.

- In the subsystem that the scissor lift motor belongs to (in this case, it is called `SubsystemClimb` because the scissor lift is part of a climber mechanism), define a method that sets the PID constants of the scissor motor. In this example, it will be called `setScissorPIDF()`. The method should take the constants as arguments and set them on the motor controller using the methods in the previous table.


```

/**
 * Sets the PID Constants of the scissor motor.
 * @param p P gain
 * @param i I gain
 * @param d D gain
 * @param f F gain
 * @param IZone Range from target at which I should take effect
 * @param lowLimit lowest allowable output (max: -1)
 * @param highLimit highest allowable output (max: 1)
 */
public void setScissorPIDF(double p, double i, double d, double f, double IZone, double lowLimit, double highLimit) {
    scissors.getPIDController().setP(p, 0);
    scissors.getPIDController().setI(i, 0);
    scissors.getPIDController().setD(d, 0);
    scissors.getPIDController().setFF(f, 0);
    scissors.getPIDController().setIZone(IZone, 0);
    scissors.getPIDController().setOutputRange(lowLimit, highLimit);
}

```

Optionally, you may also choose to add a ramp rate to the PID loop to make it smoother. This is not required for all PID loops, but they should be added if a PID loop is jittery.

- Also in your subsystem, define a method that sets the target position of the motor. In this example it will be called `setScissorsPosition()`.

```

/**
 * Sets the target position (in rotations) of the scissor motor.
 * @param position
 */
public void setScissorsPosition(double position) {
    scissors.getPIDController().setReference(position, ControlType.kPosition);
}

```

- Next, create a new `Command`. In this example it will be called `CyborgCommandTestScissorPosition`.
- In the command, declare the subsystem that the scissor lift belongs to. Define it and add it as a requirement in the constructor.

```

public class CyborgCommandTestScissorPosition extends CommandBase {
    private SubsystemClimb scissors;

    /**
     * Creates a new CyborgCommandTestScissorPosition.
     */
    public CyborgCommandTestScissorPosition(SubsystemClimb scissors) {
        // Use addRequirements() here to declare subsystem dependencies.
        this.scissors = scissors;
        addRequirements(this.scissors);
    }
}

```

- In the command's `initialize()` method, get the p, i, d, f, izone, and output limit values from the Preferences table. Remember that `Util.getAndSetDouble()` is not a method provided by WPILib. See the Preferences section in the Dashboard chapter for the definition of that method. Use the method you defined in your subsystem earlier to set them on the motor controller.

```
// Called when the command is initially scheduled.
@Override
public void initialize() {
    double p = Util.getAndSetDouble("Scissor Position P", 0);
    double i = Util.getAndSetDouble("Scissor Position I", 0);
    double d = Util.getAndSetDouble("Scissor Position D", 0);
    double f = Util.getAndSetDouble("Scissor Position F", 0);
    double IZone = Util.getAndSetDouble("Scissor Position IZone", 0);

    double upperOutLimit = Util.getAndSetDouble("Scissor Position Max Out", 1);
    double lowerOutLimit = Util.getAndSetDouble("Scissor Position Min Out", -1);

    scissors.setScissorPIDF(p, i, d, f, IZone, lowerOutLimit, upperOutLimit);
}
```

- In the command's `execute()` method, set the position target of the motor. In this example, we will use a value from the Preferences table.

```
// Called every time the scheduler runs while the command is scheduled.
@Override
public void execute() {
    double scissorsTargetPosition = Util.getAndSetDouble("Scissors Target Position", 0);
    scissors.setScissorsPosition(scissorsTargetPosition);
}
```

- In the `end()` method, set the percent output of the motor to 0 to cancel the PID, and in the `isFinished()` method, set your condition for the command ending. In the example, the command will never end.

```
// Called once the command ends or is interrupted.
@Override
public void end(boolean interrupted) {
    scissors.setScissorsPercentOutput(0);
}

// Returns true when the command should end.
@Override
public boolean isFinished() {
    return false;
}
```

- Tune your PID loop (see Tuning PIDs)

- Example code starts on the next page.

CyborgCommandTestScissorPosition.java

```

src > main > java > frc > robot > commands > CyborgCommandTestScissorPosition.java > ...
1  /*-----*/
2  /* Copyright (c) 2019 FIRST. All Rights Reserved. */
3  /* Open Source Software - may be modified and shared by FRC teams. The code */
4  /* must be accompanied by the FIRST BSD license file in the root directory of */
5  /* the project. */
6  /*-----*/
7
8  package frc.robot.commands;
9
10 import edu.wpi.first.wpilibj2.command.CommandBase;
11 import frc.robot.subsystems.SubsystemClimb;
12 import frc.robot.util.Util;
13
14 public class CyborgCommandTestScissorPosition extends CommandBase {
15     private SubsystemClimb scissors;
16
17     /**
18      * Creates a new CyborgCommandTestScissorPosition.
19      */
20     public CyborgCommandTestScissorPosition(SubsystemClimb scissors) {
21         // Use addRequirements() here to declare subsystem dependencies.
22         this.scissors = scissors;
23         addRequirements(this.scissors);
24     }
25
26     // Called when the command is initially scheduled.
27     @Override
28     public void initialize() {
29         double p = Util.getAndSetDouble("Scissor Position P", 0);
30         double i = Util.getAndSetDouble("Scissor Position I", 0);
31         double d = Util.getAndSetDouble("Scissor Position D", 0);
32         double f = Util.getAndSetDouble("Scissor Position F", 0);
33         double IZone = Util.getAndSetDouble("Scissor Position IZone", 0);
34
35         double upperOutLimit = Util.getAndSetDouble("Scissor Position Max Out", 1);
36         double lowerOutLimit = Util.getAndSetDouble("Scissor Position Min Out", -1);
37
38         scissors.setScissorPIDF(p, i, d, f, IZone, lowerOutLimit, upperOutLimit);
39     }
40
41     // Called every time the scheduler runs while the command is scheduled.
42     @Override
43     public void execute() {
44         double scissorsTargetPosition = Util.getAndSetDouble("Scissors Target Position", 0);
45         scissors.setScissorsPosition(scissorsTargetPosition);
46     }
47
48     // Called once the command ends or is interrupted.
49     @Override
50     public void end(boolean interrupted) {
51         scissors.setScissorsPercentOutput(0);
52     }
53
54     // Returns true when the command should end.
55     @Override
56     public boolean isFinished() {
57         return false;
58     }
59 }
60

```

SubsystemClimb.java - showing only the parts that are relevant to the example. `winch` and `storedPosition` are not used in the example.

```
public class SubsystemClimb extends SubsystemBase {
    private static CANSparkMax
        scissors,
        winch;

    private static ClimbPosition storedPosition;

    /**
     * Creates a new SubsystemClimb.
     */
    public SubsystemClimb() {
        scissors = new CANSparkMax(Constants.CLIMBER_SCISSOR_ID, MotorType.kBrushless);
        winch = new CANSparkMax(Constants.CLIMBER_WINCH_ID, MotorType.kBrushless);
        storedPosition = ClimbPosition.LOWEST;
        configureMotors();
    }
}
```

```
/**
 * Sets the PID Constants of the scissor motor.
 * @param p P gain
 * @param i I gain
 * @param d D gain
 * @param f F gain
 * @param IZone Range from target at which I should take effect
 * @param lowLimit lowest allowable output (max: -1)
 * @param highLimit highest allowable output (max: 1)
 */
public void setScissorPIDF(double p, double i, double d, double f, double IZone, double lowLimit, double highLimit) {
    scissors.getPIDController().setP(p, 0);
    scissors.getPIDController().setI(i, 0);
    scissors.getPIDController().setD(d, 0);
    scissors.getPIDController().setFF(f, 0);
    scissors.getPIDController().setIZone(IZone, 0);
    scissors.getPIDController().setOutputRange(lowLimit, highLimit);
}
}
```

```
/**
 * Sets the target position (in rotations) of the scissor motor.
 * @param position
 */
public void setScissorsPosition(double position) {
    scissors.getPIDController().setReference(position, ControlType.kPosition);
}
}
```

PIDs on the RoboRIO

Whenever a PID cannot be done on a motor controller, it must be done on the RoboRIO. This can be done with WPILib's `PIDController` class. Some examples of situations where PIDs may need to be run on the RoboRIO include, but are not limited to:

- A command that runs a PID loop to drive a certain distance while keeping its gyro position constant. This would require 2 `PIDControllers`: One controlling the distance that the robot has driven, and one that controls the heading angle of the robot.
- A command that adjusts the heading of the robot while a velocity PID loop is being run on the drivetrain controllers. The only `PIDController` needed would be to control the heading.
- A command that drives a mast upwards at a certain velocity while travelling to a set position. A velocity PID loop would run on the mast motor controller while a `PIDController` on the RoboRIO would control the velocity setpoint based on how far away from the target position the mast is.

Running a PID loop on the RoboRIO should only be done when absolutely necessary. PID loops on the RoboRIO will be slower and therefore less responsive than PID loops on motor controllers.

To implement a PID loop on the RoboRIO using the `PIDController` class, follow the steps outlined below. In this example, a `PIDController` will be used to move a motor in a subsystem to a certain position.

- To start, ensure that the subsystem has a method that sets the percent output of the motor (in this example, it is called `setPercentOutput()`) and a method that returns the position of the motor (in this example, it is called `getPosition()`)
- Create a new `Command`. In this example, it will be called `ExampleCommand`.
- Declare the subsystem that the motor belongs to as a private variable. Also declare a private `PIDController`.
 - NOTE: as of 2021, two versions of the `PIDController` class exist. Use the one in the `edu.wpi.first.wpilibj.controller` import. The other one is deprecated.

```
/** An example command that uses an example subsystem. */
public class ExampleCommand extends CommandBase {
    private ExampleSubsystem subsystem;
    private PIDController controller;
```

- In the command's constructor, define the subsystem and add it as a requirement. Do not do anything with the `PIDController` yet.


```
/**
 * Creates a new ExampleCommand.
 * @param subsystem The subsystem used by this command.
 */
public ExampleCommand(ExampleSubsystem subsystem) {
    this.subsystem = subsystem;
    addRequirements(this.subsystem);
}
```

- In the command's `initialize()` method, pull the PID constants and motor setpoint from the Preferences table and use them to define the `PIDController`. The setpoint of the `PIDController` is set using the `setSetpoint()` method.

```
// Called when the command is initially scheduled.
@Override
public void initialize() {
    double
        p = Util.getAndSetDouble("kP", 0),
        i = Util.getAndSetDouble("kI", 0),
        d = Util.getAndSetDouble("kD", 0),
        setpoint = Util.getAndSetDouble("setpoint", 0);

    controller = new PIDController(p, i, d);
    controller.setSetpoint(setpoint);
}
```

- In the `execute()` method, use the `PIDController` to calculate the output of the motor and then set it using the appropriate method in your subsystem. Optionally, you can add a constant value to the output as a feed-forward.

```
// Called every time the scheduler runs while the command is scheduled.
@Override
public void execute() {
    double output = controller.calculate(subsystem.getPosition());
    output += Util.getAndSetDouble("kF", 0); //this is how a kF would be achieved on a PIDController

    //ensure that the output is within the bounds of -1 to 1. This can also be achieved using two if statements.
    output = (output < -1 ? -1 : (output > 1 ? 1 : output));

    subsystem.setPercentOutput(output);
}
```

- In the `execute()` method, set the percent output of the motor to 0. In the `isFinished()` method, define the condition for the command ending. For this example, the command will never end.

```
// Called once the command ends or is interrupted.  
@Override  
public void end(boolean interrupted) {  
    subsystem.setPercentOutput(0);  
}  
  
// Returns true when the command should end.  
@Override  
public boolean isFinished() {  
    return false;  
}
```

- Tune your PID loop (see Tuning PIDs)
- Full example code is on the next page.

ExampleCommand.java

```

src > main > java > frc > robot > commands > ● ExampleCommand.java > ...
1  // Copyright (c) FIRST and other WPILib contributors.
2  // Open Source Software; you can modify and/or share it under the terms of
3  // the WPILib BSD license file in the root directory of this project.
4
5  package frc.robot.commands;
6
7  import frc.robot.subsystems.ExampleSubsystem;
8  import frc.robot.util.Util;
9  import edu.wpi.first.wpilibj.controller.PIDController;
10 import edu.wpi.first.wpilibj2.command.CommandBase;
11
12 /** An example command that uses an example subsystem. */
13 public class ExampleCommand extends CommandBase {
14     private ExampleSubsystem subsystem;
15     private PIDController controller;
16
17     /**
18      * Creates a new ExampleCommand.
19      * @param subsystem The subsystem used by this command.
20      */
21     public ExampleCommand(ExampleSubsystem subsystem) {
22         this.subsystem = subsystem;
23         addRequirements(this.subsystem);
24     }
25
26     // Called when the command is initially scheduled.
27     @Override
28     public void initialize() {
29         double
30             p = Util.getAndSetDouble("kP", 0),
31             i = Util.getAndSetDouble("kI", 0),
32             d = Util.getAndSetDouble("kD", 0),
33             setpoint = Util.getAndSetDouble("setpoint", 0);
34
35         controller = new PIDController(p, i, d);
36         controller.setSetpoint(setpoint);
37     }
38
39     // Called every time the scheduler runs while the command is scheduled.
40     @Override
41     public void execute() {
42         double output = controller.calculate(subsystem.getPosition());
43         output += Util.getAndSetDouble("kF", 0); //this is how a kF would be achieved on a PIDController
44
45         //ensure that the output is within the bounds of -1 to 1. This can also be achieved using two if statements.
46         output = (output < -1 ? -1 : (output > 1 ? 1 : output));
47
48         subsystem.setPercentOutput(output);
49     }
50
51     // Called once the command ends or is interrupted.
52     @Override
53     public void end(boolean interrupted) {
54         subsystem.setPercentOutput(0);
55     }
56
57     // Returns true when the command should end.
58     @Override
59     public boolean isFinished() {
60         return false;
61     }
62 }
63

```

Tuning PIDs

If you jumped right into the examples and ran them without reading this section first, you may have found that the PIDs you just programmed did not do anything at all. The subsystems may have simply sat there, idle. A well functioning PID loop relies on a carefully tuned set of PID constants. An ill-tuned PID loop might oscillate, jitter, not reach the target, or not move at all. Follow these steps to tune your PID loop:

Tuning kP

- To start, ensure that all of your PID constants have been set to 0. If you cannot find your values on the Preferences table, you may need to quickly run your command so that the `initialize()` method is called and the default values are stored.
- It is recommended that you put a graph or readout of the setpoint and the current position or velocity on the dashboard. This will help you easily see if the PID loop is oscillating or not reaching the target.
- Set the kP value to a very small value, like 0.000001. On the SparkMAX motor controller, start smaller (closer to 0.0000000001).
- Run the PID command. The motor should barely move if it even moves at all.
- Increase the kP value by deleting a zero or two and re-run the PID command. Do this until the motor gets close to the setpoint without overshooting it.
- Now, make smaller corrections (change the numbers instead of deleting zeros) to the kP value to try to get the motor to exactly achieve the setpoint. Remember, the PID should fully reach the setpoint without overshooting and/or oscillating. If the mechanism cannot seem to do this, then kI may need to be added.
- At this point, if the motor achieves the setpoint without overshooting, the PID is now tuned. You can skip to “After Tuning”.

Tuning kI

- If the motor cannot achieve the setpoint without overshooting, set the kP to a value such that the motor gets close to the setpoint and does not oscillate.
- Set the IZone of the PID loop to a positive value such that the current range of error is well within the zone, but not so large that adding a kI would cause unpredictable behavior.
 - Recommended: $IZone = (desired\ position - actual\ position) * 1.3$.
Remember, the IZone needs to be positive, so invert it if you need to.
- Add a very small kI value (start smaller than what you started kP at).
- Tune the kI the same way you tuned the kP. Get rid of 0's until the motor gets close, then make fine adjustments.
- At this point, if the motor achieves the setpoint without overshooting, the PID is now tuned.

Tuning kD

- If the motor simply does not achieve the setpoint fast enough without overshooting, then you may need to add some kD. This is almost never needed in an FRC application, so only do this for systems that must be fast but absolutely cannot overshoot.
- Set the kP value to the faster value that overshoots. The kD will help to counteract the overshooting.
- Tune the kD the same way you tuned kP.

Tuning kF

- If your mechanism needs a constant force to counteract another force (such as gravity or a constant-tension spring), then add kF.
- Tune kF the same way you tune kP.

After Tuning

- After you are satisfied with your PID constants, write them down in a text file and keep it somewhere safe, like the root of your robot project. The RoboRIO will forget your tuned PID Preferences if it gets re-imaged and programmers don't have the time to tune PID loops twice.
- Non-zero backup constants should only be set once your PID constants have been tested for a long time (weeks or months) without failing.

PID Best Practices

Follow these best practices when implementing PID loops onto the robot:

- P, I, D, F, IZone, and output limit values should be accessible from the Preferences table.
- The P, I, D, F, and IZone values should ALWAYS default to 0 when the PID is untested. Only when the PID has been thoroughly tested (as in weeks or months of flawless operation) should you set non-zero defaults.
- P, I, D, F, IZone, and output limits should be set on the motor controllers or `PIDController` objects in the `initialize()` method of your command.
- In the `end()` method on the PID command, the percent output of any motor controllers running a PID should be set to `0`.
- Any motor controller needing a PID loop should have a set amp limit.

Vision

Computer vision is a very important part of an FRC robot. FRC challenges in the past have usually included some sort of computer vision challenge where a retro-reflective target is placed on a field element. Teams that are able to recognize this target can align systems to it to score

points easier, and typically have an advantage over teams that cannot recognize the targets. This chapter will cover different ways to add vision to your robot.

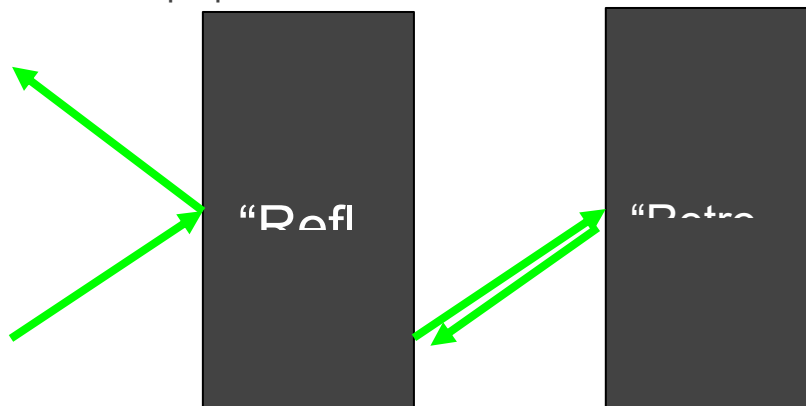
Algorithm

In FRC, the computer vision challenges have almost always featured some sort of retro-reflective target. This allows teams to use a simple, yet smart solution to solve the challenges. This section will cover the very basics about how an efficient FRC vision algorithm works. More on the basic algorithm along with basic code that achieves it can be found at tinyurl.com/frcvisionalgorithm.

To illustrate how the algorithm works, let's suppose we have the following image that is being taken from a camera. Pictured in the image is our target of interest: the piece of wood with the retro-reflective brackets.

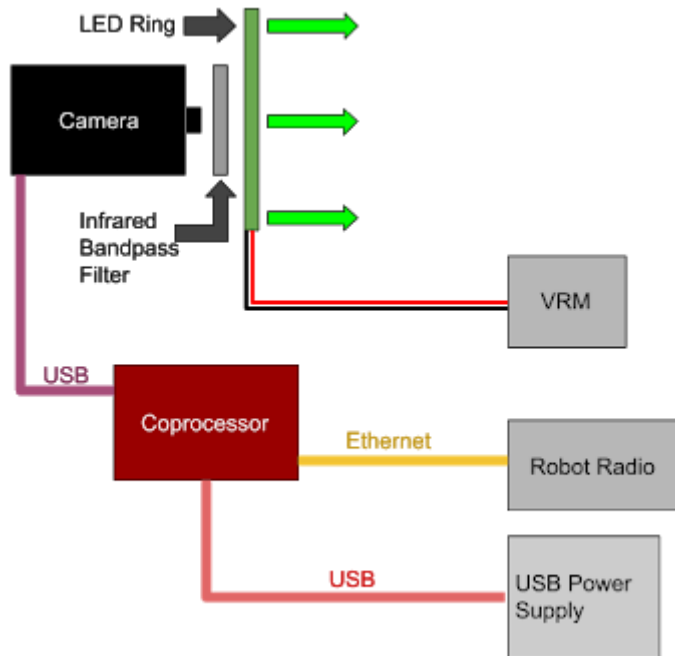


The retro-reflective aspect of the target is extremely important to the FRC vision algorithm. Below is an illustration of the difference between the properties of just “reflective” items versus the properties of “retro-reflective” items.



As seen in the image, light bounces off of a retro-reflective object directly towards the light source that it came from. So if we put the camera directly behind a light source such as an LED ring, any retro-reflective object will appear to the camera as a direct light source.

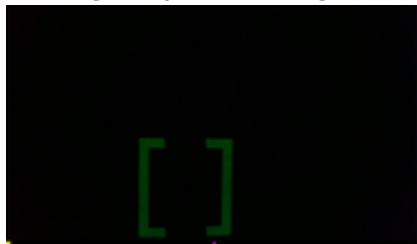
That being said, we can now look at the common FRC vision setup.



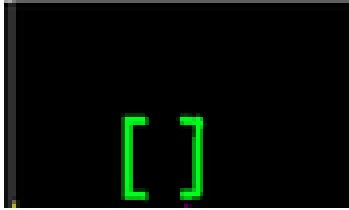
The camera should be located behind an LED ring which shines light towards the retro-reflective targets. In FRC, this LED ring should be green or infrared (but infrared is better), because neither of those colors are present on a typical FRC field (FRC fields are typically red and blue). The Infrared bandpass filter seen in the image should only be used if an infrared LED ring is used. This filter eliminates all colors except for infrared, ensuring that the camera sees as little noise from ambient light as possible. The camera is connected to a coprocessor (computer other than the RoboRIO), which is typically a Raspberry Pi. It is important that a coprocessor is used to do the computer vision calculations because the RoboRIO is not powerful enough to do

them on its own. The coprocessor will do the vision calculations and then send any target data to the RoboRIO.

In order for the algorithm to work, the camera needs to be configured a certain way. The camera's auto-exposure setting should be disabled, and its manual exposure setting should be set as low as possible. This ensures that the camera can only see direct light sources. In addition, the camera's auto white balance setting should be disabled. With the proper settings in place, images taken of our bracket target should look like this:



The next step is to apply a “threshold” to the image. This step tests the brightness of the pixels in the image. If a pixel is not above the “threshold,” then it is turned black. If it passes the threshold, then it is set to its maximum brightness.



Next, we can optionally apply dilation and/or erosion to an image. Dilations take objects in an image and make them thicker. This can help to eliminate any imperfections that might be present in the image. Erosions make objects smaller, which can help to eliminate noise in an image.

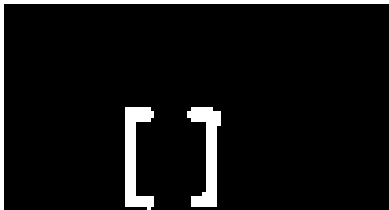


Left: undilated image | Right: dilated image.

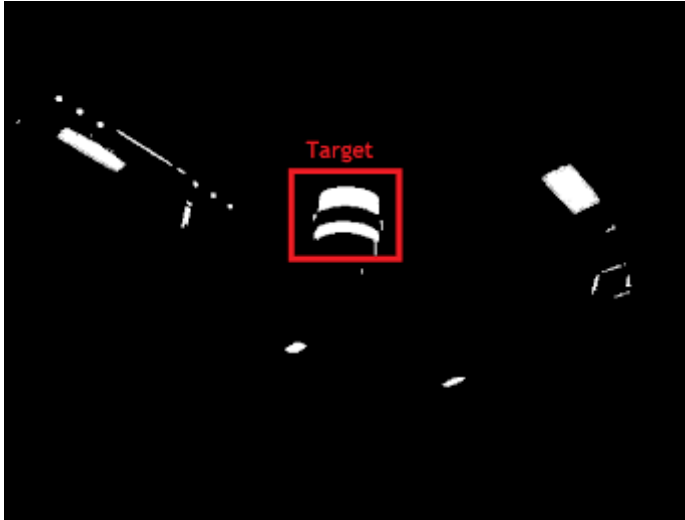


Left: uneroded image | Right: eroded image.

Next, we eliminate all pixels in the image that are not green. By performing this “in-range” test, all pixels that are not in a range of possible shades of green are turned black. Those that remain are turned white. After the in-range test, the target might look like this:

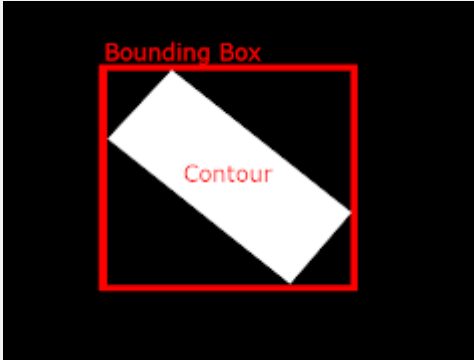


Now we need to look for the target itself. In a noiseless image like this one, that would be easy. Realistically however, the image probably has a few other objects, or “contours” in it. It will probably look more like this one:



Shown: Target from 2017 game, but with some undesired contours.

In order to eliminate the extraneous, invalid contours, you can perform a series of tests on specific properties. Contours whose properties are not within a range of acceptable values are eliminated. The table below shows a few of the different properties that a contour can be tested on.

Property	Description
Area	The actual area of the contour itself.
Solidity	<p>The ratio of the actual area of a contour to the area of its bounding box:</p> $\text{solidity} = \frac{\text{actual contour area}}{\text{bounding box area}}$  <p>That being said, the more space inside the bounding box the solid takes up, the greater this solidity value will be.</p>
Angle	The angle of tilt of a contour's closest-fitting bounding box. This test was particularly useful for the 2019 game.

Aspect Ratio	<p>The ratio of the width of the contour to the height of the contour:</p> $aspect\ ratio = \frac{contour\ width}{contour\ height}$
--------------	---

The contours that remain are ideally the targets (or part of them) we are looking for. For targets with multiple contours, the only task remaining is to group the contours into the targets themselves, a task which will differ depending on the target's composition.

Once the final targets are determined, information about the targets must be sent to the RoboRIO so that they can be used by the robot code. At the very least, the image coordinates of the targets should be sent. However, other good values to send include horizontal and vertical angles between the robot and the target and the distance from the robot to the target. Data can be sent to the robot using a UDP (unreliable datagram protocol) or TCP (transmission control protocol) socket sender, which will send messages through Ethernet to the RoboRIO. Because your vision code will be sending multiple messages per second, a UDP socket is the best choice because it is the easiest to set up.

How to add Vision to a Robot

The algorithm described above is complex and can be hard to program. However, you do not actually need to program it unless you want to. Listed below are a few ways that a team can put vision on their robot.

- **KiwiLight**
 - KiwiLight is a free tool developed by Foximus Prime (FRC 3695) that can solve FRC vision challenges using any USB camera that OpenCV (Open Source Computer Vision Library) can talk to. It requires a Raspberry Pi or other Linux computer, a USB camera, and an LED ring. It features a graphical user interface that allows a user to “teach” the robot a target that it is spotting and then do fine tuning on it. This makes vision easy for beginner teams, but is also effective for advanced teams. KiwiLight can be found at github.com/wh1ter0se/KiwiLight.
- **Writing a program with OpenCV**
 - You might choose to write your own vision program for your robot, using the Open Source Computer Vision Library (OpenCV). This is a powerful library that can be used to write programs that solve FRC computer vision problems. KiwiLight, Limelight, and GRIP all use OpenCV. OpenCV can be found at opencv.org, and documentation can be found at docs.opencv.org. This method

of adding computer vision to the robot can be long and difficult and is not recommended unless you have OpenCV experience prior to the season's start.

- **GRIP**

- GRIP (Graphically Represented Image Processing) is a program that allows a programmer to use a graphical interface to design a computer vision algorithm, then have the code generated for them. More information about GRIP can be found at tinyurl.com/frcgrip.

- **Limelight**

- Limelight is a camera for FRC robots that solves FRC vision challenges. It connects to the robot via Ethernet and sends its data to NetworkTables. This is an easy solution for beginner teams, but mounting the camera on the robot can be quite difficult because the camera is boxy, and the field of view, which cannot be changed, is rather restricting. In addition, Limelight uses a green light and has no infrared option. More about Limelight can be found at limelightvision.io/.

Autonomous Driving

Something that you will almost always need for a good autonomous routine is some sort of autonomous driving. Moving the robot around is essential to getting things done, and there are almost always points awarded to alliances whose robots drive off of their starting points during the autonomous period.

Unlike vision, which, in FRC, has one commonly accepted algorithm that almost every team uses, autonomous driving algorithms differ vastly, from tank-style robots to swerve robots to mecanum robots. Different solutions use different methods of calculating position and trajectory as well. This section will not cover algorithms for achieving autonomous driving, but rather different ways to add it to your robot.

Driving in a Straight Line with PID Loops

An easy and effective way of adding simple autonomous driving to your robot is to use a PID loop (or multiple PID loops) to drive the robot in a straight line. This method of driving autonomously requires only a PID command and a little bit of PID tuning. Depending on the game, this type of driving can be very effective and help the robot score high numbers of points. For example, in the 2020 game, a robot could score 6 to 8 game pieces (out of 13 possible game pieces) by driving in a straight line. However, this method of autonomous driving has its limits. If a robot needs to change direction during the autonomous routine, this method of driving could make for a rather clunky auto.

Hyperdrive

Hyperdrive is an autonomous driving system developed by Foximus Prime that can “remember” paths that were driven by a human driver. It can then drive the same paths autonomously. Hyperdrive is an easy to implement library and can reliably drive almost any path at decent speeds. Hyperdrive can be found at github.com/BTK203/Hyperdrive along with a manual that guides you through implementing it on your robot.

PathFinder / PathWeaver

PathFinder (also called PathWeaver) is another option that you have for autonomous guidance. It is a library similar to Hyperdrive, but the paths are defined using a graphical user interface.

Designing an Autonomous Routine

During the autonomous period, a robot is going to need to do several things. It might calibrate a subsystem, then drive somewhere, then perform a task, then drive somewhere else, then perform another task. This sequence of tasks is commonly called a routine. The tasks that the robot might perform during a routine are nothing more than simple commands. An autonomous routine simply consists of a bunch of commands that have been chained together into one. For more information about chaining commands together, see the Command-Based Programming section in the WPILib chapter. After linking the commands together, the resulting command should be scheduled by the `autonomousInit()` method in the `Robot` class. The autonomous can then be run by enabling the robot in “Autonomous” mode.

One of the trickiest parts of designing an autonomous routine is keeping your code readable. A good way to do this is to define a separate class for each autonomous routine. These classes should implement some sort of interface. Not only does this create a structure for routines to use, it also makes it easy to create a `SendableChooser` that the user can use to select the autonomous routine to run.

As a basic example of designing an autonomous routine, we will design a routine that will drive a robot backwards 36 inches, align a turret, and shoot 3 pre-loaded dodgeballs. We will start with our autonomous interface, which is called `IAuto`. This interface features one method:

Method	Purpose
<code>Command getCommand()</code>	Returns the <code>Command</code> that will be run during the autonomous period.

```

/**
 * Basic interface for autonomous classes.
 */
public interface IAuto {
    /**
     * Returns the command that should be run during auto.
     */
    public Command getCommand();
}

```

Now we define our autonomous class. This class will implement our `IAuto` interface.

```

public class BareMinimumAuto implements IAuto {

    public BareMinimumAuto() {

    }

    public Command getCommand() {
        return null;
    }

}

```

Next, declare all of the `Commands` that we need and then define them in the constructor. Also take this time to write and test any new commands that you may need to make the autonomous functional. In the case of this autonomous, we will declare and define five commands:

```

public class BareMinimumAuto implements IAuto {
    private Command
        init, //Zeros drivetrain encoders and gyro, drives off of the line, zeros the turret.
        positionTurret, //Sets the turret to a position where vision can see the target.
        alignTurret, //Aligns the turret to the target.
        shootPayload,
        driveFlywheel; //Shoots the 3 preloaded power cells.

    public BareMinimumAuto(
        SubsystemDrive drivetrain,
        SubsystemTurret turret,
        SubsystemReceiver kiwilight,
        SubsystemIntake intake,
        SubsystemFeeder feeder,
        SubsystemFlywheel flywheel
    ) {
        //drive off line and zero turret
        this.init = new InitAuto(drivetrain, turret).getCommand();

        //set turret position and align
        this.positionTurret = new CyborgCommandSetTurretPosition(turret, Constants.AUTO_INIT_YAW_TARGET, Constants.AUTO_INIT_PITCH_TARGET);
        this.alignTurret = new CyborgCommandAlignTurret(turret, kiwilight, false);

        //shoot preloaded power cells
        this.shootPayload = new CyborgCommandShootPayload(intake, feeder, flywheel, turret, 3, false);

        //drive flywheel
        this.driveFlywheel = new CyborgCommandFlywheelVelocity(flywheel);
    }
}

```

After this, define the `getCommand()` method. This is where the commands will be chained together. Before writing the method, make sure you know exactly how the commands should be

chained together. In this autonomous, we know that the robot will do the following in chronological order:

- Zero drivetrain encoders and gyro, drive off of the line, and zero turret. (`InitAuto`)
 - Set the turret to a position where the vision system can see the target (`CyborgCommandSetTurretPosition`)
 - Shoot the 3 preloaded dodgeballs WHILE aligning the turret to the target.
- All of these tasks get done while the flywheel is spinning.

```
public Command getCommand() {
    //align and shoot command
    Command
        alignAndShoot = this.alignTurret.raceWith(this.shootPayload),
        doAuto         = init.andThen(positionTurret, alignAndShoot);

    //init then position, then align and shoot
    return doAuto.raceWith(driveFlywheel);
}
```

Now, we need to schedule that command in the `Robot` class' `autonomousInit()` method, which will be run when the robot is enabled in autonomous mode. It is best to create a method in the `RobotContainer` class that is called by `autonomousInit()` (let's call this one `startAuto()`). In its most basic form, that method will look something like this:

```
public void startAuto() {
    IAuto autoToRun = new BareMinimumAuto(SUB_DRIVE, SUB_TURRET, SUB_RECEIVER, SUB_INTAKE, SUB_FEEDER, SUB_FLYWHEEL);
    Command autonomousCommand = autoToRun.getCommand();
    autonomousCommand.schedule();
}
```

However, it is better to let the user choose the autonomous mode to run instead of hard-coding it. This functionality can be achieved using a `SendableChooser<IAuto>`. We can define one as a `private` variable of the `RobotContainer` class.

```
private SendableChooser<IAuto> autoChooser;
```

Then, define the chooser in the constructor:

```
public RobotContainer() {
    BareMinimumAuto bareMinimumAuto = new BareMinimumAuto(SUB_DRIVE, SUB_TURRET, SUB_RECEIVER, SUB_INTAKE, SUB_FEEDER, SUB_FLYWHEEL);
    this.autoChooser = new SendableChooser<IAuto>();
    autoChooser.setDefaultOption("Bare Minimum Auto", bareMinimumAuto);
    //use autoChooser.addOption() to add more autonomous options to the chooser
}
```

In `startAuto()`, we can now reference the `SendableChooser` when starting our auto.

```
public void startAuto() {  
    IAuto autoToRun = autoChooser.getSelected();  
    Command autonomousCommand = autoToRun.getCommand();  
    autonomousCommand.schedule();  
}
```

In the `Robot` class' `autonomousInit()` method, make sure that you call `startAuto()`.

```
@Override  
public void autonomousInit() {  
    DriverStation.reportWarning("AUTO STARTING", false);  
    DriverStation.reportWarning("AAAAAAAAAAAA", false);  
  
    robotContainer.startAuto();  
}
```

A video of this autonomous running on 3695's 2020 robot can be found at the link tinyurl.com/bareminimumauto. The full example code starts on the next page.

IAuto.java

src > main > java > frc > robot > auto > IAuto.java > ...

You, seconds ago | 2 authors (BTK203 and others)

```

1  /*-----*/
2  /* Copyright (c) 2018-2019 FIRST. All Rights Reserved. */
3  /* Open Source Software - may be modified and shared by FRC teams. The code */
4  /* must be accompanied by the FIRST BSD license file in the root directory of */
5  /* the project. */
6  /*-----*/
7

```

BTK203, 4 months ago

```

8  package frc.robot.auto;
9
10 import edu.wpi.first.wpilibj2.command.Command;
11

```

You, seconds ago | 2 authors (BTK203 and others)

```

12 /**
13  * Basic interface for autonomous classes.
14  */
15 public interface IAuto {
16     /**
17      * Returns the command that should be run during auto.
18      */
19     public Command getCommand();
20 }
21

```



```

23 You: seconds ago | 2 authors (BTK203 and others)
24 /**
25  * "Bare Minimum" autonomous command, driving off the line and shooting all
26  * preloaded power cells.
27  */
28 public class BareMinimumAuto implements IAuto {
29     private Command
30     init,          //Zeros drivetrain encoders and gyro, drives off of the line, zeros the turret.
31     positionTurret, //Sets the turret to a position where vision can see the target.
32     alignTurret,    //Aligns the turret to the target.
33     shootPayload,
34     driveFlywheel;  //Shoots the 3 preloaded power cells.
35
36     public BareMinimumAuto(
37         SubsystemDrive drivetrain,
38         SubsystemTurret turret,
39         SubsystemReceiver kiwilight,
40         SubsystemIntake intake,
41         SubsystemFeeder feeder,
42         SubsystemFlywheel flywheel
43     ) {
44         //drive off line and zero turret
45         this.init = new InitAuto(drivetrain, turret).getCommand();
46
47         //set turret position and align
48         this.positionTurret = new CyborgCommandSetTurretPosition(turret, Constants.AUTO_INIT_YAW_TARGET, Constants.AUTO_INIT_PITCH_TARGET);
49         this.alignTurret = new CyborgCommandAlignTurret(turret, kiwilight, false);
50
51         //shoot preloaded power cells
52         this.shootPayload = new CyborgCommandShootPayload(intake, feeder, flywheel, turret, 3, false);
53
54         //drive flywheel
55         this.driveFlywheel = new CyborgCommandFlywheelVelocity(flywheel);
56     }
57
58     public Command getCommand() {
59         //align and shoot command
60         Command
61         alignAndShoot = this.alignTurret.raceWith(this.shootPayload),
62         doAuto        = init.andThen(positionTurret, alignAndShoot);
63
64         //init then position, then align and shoot
65         return doAuto.raceWith(driveFlywheel);
66     }
67 }

```

BareMinimumAuto.java

RobotContainer.java - Only important parts given

```

72 public class RobotContainer {
73     /**
74      * Subsystems
75      */
76     private final SubsystemDrive SUB_DRIVE = new SubsystemDrive();
77     private final SubsystemTurret SUB_TURRET = new SubsystemTurret();
78     private final SubsystemIntake SUB_INTAKE = new SubsystemIntake();
79     private final SubsystemFeeder SUB_FEEDER = new SubsystemFeeder();
80     private final SubsystemFlywheel SUB_FLYWHEEL = new SubsystemFlywheel();
81     private final SubsystemSpinner SUB_SPINNER = new SubsystemSpinner();
82     private final SubsystemClimb SUB_CLIMB = new SubsystemClimb();
83     private final SubsystemReceiver SUB_RECEIVER = new SubsystemReceiver();
84     private final SubsystemJevois SUB_JEVOIS = new SubsystemJevois();
85
86     private SendableChooser<IAuto> autoChooser;
87
88     /**
89      * The container for the robot. Contains subsystems, OI devices, and commands.
90      */
91     public RobotContainer() {
92         BareMinimumAuto bareMinimumAuto = new BareMinimumAuto(SUB_DRIVE, SUB_TURRET, SUB_RECEIVER, SUB_INTAKE, SUB_FEEDER, SUB_FLYWHEEL);
93         this.autoChooser = new SendableChooser<IAuto>();
94         autoChooser.setDefaultOption("Bare Minimum Auto", bareMinimumAuto);
95         //use autoChooser.addOption() to add more autonomous options to the chooser
96
97         SmartDashboard.putData(autoChooser);
98     }
99
100     public void startAuto() {
101         IAuto autoToRun = autoChooser.getSelected();
102         Command autonomousCommand = autoToRun.getCommand();
103         autonomousCommand.schedule();
104     }

```


Autonomous Tips / Best Practices

These are a few things to keep in mind while adding autonomous ability to the robot:

- **Program multiple autonomous routines.** Program some really simple, fail-safe ones and some more complex ones. That way, if your complex ones fail, you can fall back to the simpler, more reliable ones.
- **Make sure that your autonomous routines do not have the robot starting in one spot.** If another robot at competition has a better autonomous that starts in the same spot, it would be best for the other robot to start in that spot, and you still want your robot to be able to do something, even if it is not in its normal spot. Typically, robots score points for driving off of their starting positions during the autonomous period.
- **Communicate with the team and research strategy to spend time on autonomous efficiently.** The most useful autonomous routines are based on a balance of mechanical ability, strategy, and complexity, all of which change significantly each year. Some goals in autonomous may be more desirable than others.
- **Consider the positions and potential movements of the bots on your alliance.** The path of another team's auto could interfere with yours, so it's best to plan ahead and have options ready.
- **Your autonomous routines should be selectable using a `SendableChooser`.**
- **Each autonomous routine should have its own class.** These classes should extend a "base" autonomous class or interface.

Tips for Build Season

The Build Season is the period of time between kickoff and the first competition. During this time, the robot must be designed, built, and programmed. This part of the season can be particularly confusing for programmers because at times it can be confusing as to what exactly needs to get done. During the first part of the season, the programming team has no robot to program, but when the robot is completed, the programming team must cope with implementing basic robot control, driver assists, and autonomous routines. This chapter will cover some tips for how to navigate this period.

What to Expect to Code

FRC robots, at the very least, should have basic manual control over its systems. However, they are usually capable of doing much more. For new FRC programmers, it may be unclear what the robot should be programmed to do. This section will cover what abilities FRC programmers should expect to program their robot to have.

- **Basic manual control over all systems.** This should be the first thing that programmers do before focusing on more advanced robot abilities. That way, if the abilities do not work as well as desired, the drivers can fall back to manual control. This category also includes any dashboard video streams that the drivers will use for driving.
- **Target Alignment using Computer Vision.** As mentioned in the Autonomous Functions chapter, most FRC games feature some sort of retro-reflective target that marks an important feature on the field. Being able to align to this target using images from a camera gives robots a competitive edge during the autonomous period, and gives drive teams a competitive edge during teleop. See the Autonomous Functions chapter for more information about how to do computer vision.
- **Driver Assists other than Target Alignment.** Depending on the robot, your team may ask you to program various driver assists that the drivers can use during a match. Such driver assists might include a command that drives a flywheel at 6000 RPM, or a command that lowers a heavy telescoping mast until it hits its bottom limit switch, which drivers can use to prevent the robot from tipping. These driver assists will vary from robot to robot, so programmers should be prepared to program anything that the team asks them to program, within reason. If it is determined that the drivers do not need the assist or that the assist is impractical, programmers should resist programming it.
- **Fully Autonomous Operation.** The Autonomous Functions chapter also mentions the 15 second fully autonomous period that is usually at the beginning of every match at competition. Programmers should be prepared to write multiple autonomous routines that

the robot can perform at competition. See the Autonomous Functions chapter for more information about how to write autonomous routines.

Coding Before the Bot is Running

For much of the build season, the build team will be building the robot, meaning that the programmers cannot work with it. However, there is not enough time for the programming team to simply sit idle, waiting for the robot to be completed. There are many ways to be productive while the build team is working on the robot:

Using WPILib's Simulation Tool to code Basic Robot Control

Robots are not needed for programmers to code basic robot systems. Once the team decides on what subsystems a robot will have, and what components will be used in each subsystem, the programming team can code and test basic robot control. By simulating robot code on the desktop, programmers can ensure that their basic control code will work and even design the robot's dashboard, all without access to a robot. Programmers should use these tools to ensure that the dashboard and basic control code are ready by the time the robot is built.

Code Prototypes

Programming basic robot control using WPILib's desktop simulation tool will not keep the programming team busy throughout the whole build season. Time spent with the programming team sitting idle would be better spent getting ready for things that will need to get done later.

If the team has other functional robots that are not being used, it may be useful to use other functional robots to program new or complex things that will need to be programmed later. For example, a team might program an old robot to align to the current season's vision target, or set up and program a motor controller that has never before been used by the team. Doing this makes programming difficult or lesser-known things easier, quicker, and less stressful for the programming team.

Mechanical Prototypes

As the build team designs and builds the robot, the programming team may need to help them program functional prototypes for different subsystems. Programming these prototypes may be as simple as driving a motor (which usually can be done with the vendor's hardware client and does not require code), or get as complex as making an entire subsystem function. Programmers should also be prepared to program some of those prototypes, and for the prototypes to change rapidly as the build team figures out which ideas work and which ideas do not. Giving assistance to the build team when they need it should be the programming team's highest priority because without it, the build team may not be able to produce the best systems that they can.

After the Robot is Complete

When the build team finishes building the robot, attention will turn to the programmers to get the robot working at a competition level. The team will want to see basic control, driver assists, and autonomous routines as quickly as possible, and the programming team should be prepared to face pressure from the build team to do this. This part of the season can be particularly stressful for the programmers, who may have a short amount of time to program some pretty complex things.

Priorities

When the robot is completed, it can be hard to know what to program first. This list will outline a good order of priority, with the top items being of the highest priority, and the lower items being lower priority. Bolded items are things that absolutely have to get done before competition.

- **Basic manual controls**
- **Important driver assists that are absolutely needed by the drivers ***
- **Autonomous routines**
- Driver assists that are convenient, but not necessarily needed

*: examples include vision target alignment for shooting games such as the 2017 or 2020 game, or intelligent movement of systems that are hard to control such as multi-stage masts or flywheels that must turn at a specific speed.

Sharing time with the Drive Team

When the robot is completed, the programming team will need to split time with the drivers who need to practice driving the robot. Drive practice and programming are both very time-intensive. That being said, the team will most likely need to add more time during which it can work with the robot. The programming team should have their own time to work with the robot, and the drive team should have their own time slot as well. When discussing with the team about time-sharing, be honest about the amount of time that you think you will need to get the robot programmed, and make sure that the programming team gets enough time to complete all of its programming. The programming team may need 2-4 hours per day depending on the work that needs to get done. The more time the programming team has to work with the robot, the better the team will perform at competition. However, do not cheat the drivers out of their time to practice. Try to find a healthy balance between programming time and driving time.

Other Tips

This section will cover some other tips that programmers should keep in mind as they navigate the build season.

- **Expect to commit large amounts of time to getting the robot programmed, especially when the build team completes it.** As mentioned earlier, you will likely need to schedule 2-4 hours per day for programming once the robot has been built.
- **Hold yourself accountable to get your programming done, and get help if you can't.** Some tasks, such as auto or vision, are very complex and take large amounts of time to complete. However, these tasks are very important to the performance of the robot. If you are in charge of getting these tasks done, make sure that you do so. If you don't think you will be able to, get help from your team.
- **Only ONE person does auto.** It is perfectly fine to have other programmers programming autonomous sub-commands and functions, but piecing them together should be the job of one person. Having multiple people working on one autonomous routine is too hard to manage with the small amount of time you have before competition.
- **Be open to the team's requests, but also be realistic.** The drive team may request that you program multiple different driver assists. Some of these might be good ideas, and some might be unnecessary, impractical, or unrealistic. If you think that programming a driver assist is a good use of time, then go ahead and program it. However, if you don't, it is okay to say that. Remember, you have autonomous routines to program too!

Tips for Competition

Eventually, you and your team of fellow programmers will have programmed the robot and it will be time for competition. In competition, your code, along with you, the programmer, will be tested. You may need to modify your code as systems on the robot break and then get fixed. You may need to write autonomous routines while in the queue for your next match as the strategy team discusses with your alliance members about what the robots should do during autonomous. This chapter will cover a few tips to keep in mind while preparing and participating in a competition.

Preparing for Competition

Being well-prepared for the competition is important to your team's performance. You will want to make sure that you have the right equipment available to you at the competition so that you can use it to effectively work with the robot.

- **While preparing for competition, be sure to pack the following items:**
 - Driver Station
 - Driver Station computer with updated code
 - Backup Driver Station computer with updated code (test before leaving)
 - Any chargers needed to charge above computers
 - LONG USB A-to-B cable for connecting to RoboRIO
 - Backup USB tether
 - Multiple Ethernet cables
 - USB-C cable to connect to REV products
 - Non-competition robot battery to power Driver Station
 - Driver and Operator controllers, with backups
 - Extra coprocessors with updated code
 - Keyboard and mouse to connect to coprocessor (even if using VNC or SSH - those things can fail)
 - Video cable for coprocessor (even if using VNC or SSH)
 - Any batteries needed to power coprocessors, with backups
 - Multiple chargers for said batteries
 - Backups of any cable needed for the vision system or other coprocessor system
 - Backups of any mounting hardware for vision
 - Backup vision cameras (If using JeVois bring backup SD cards with updated code)
 - USB-to-Ethernet adapter in case Driver Station Computer's Ethernet fails

- **Ensure that you keep the following items INSIDE of the Driver Station:**
 - Driver and Operator controllers
 - USB tether
 - USB-C cable
 - Ethernet cable
 - USB-to-Ethernet adapter
 - Driver Station computer Charger
 - Backup vision cameras, cables, and batteries
- **Take note of the trusted distributions of the code and make copies before leaving.**
- **If you forget anything or need something that you do not have, see the Pit Administration.** They can announce that your team needs a certain item, and teams that have that item may go to your pit to lend it to you. The Pit Administration usually has a table set up somewhere inside of the pits.
- **Prepare a pre-match checklist.** This checklist should include anything that the team should verify before indicating that they are ready for a match. An example of an item that might appear on such a checklist might be “Check that the vision system is sending data to the RIO.” Communicate this checklist with those that will be in the pit before each match.
- **Prepare a spare-time TODO list.** Especially if auto was written using a practice bot instead of the competition bot, spare time with the competition bot should be well utilized. Prepare a list of things you want to do with the comp bot whenever there’s time *before* you get onto the floor for pit day. This list might include things like testing auto on the practice fields, running system tests, or verifying that the PID(F) calibration values are well tuned to the comp bot. Time with hands off the bot might be scarce, so prioritize this list.

Pit Day Tips

Pit day is arguably the most important day of the competition. During this day, your team must weigh the robot in, have it inspected, and can optionally compete in practice matches.

- **As soon as you get into the pits on Pit Day, send someone to get the robot router configured.** This configuration is MANDATORY for all teams. There will be a table, somewhere in the pits, with a computer that has an Ethernet cable connected to it. Take the robot radio to this table, plug it in, and use the configuration tool on the computer to configure it.
- **Take advantage of the field calibration time.** From the beginning of the day on Pit Day to approximately lunchtime, the competition field will be open for teams to configure their robots. If your robot uses a vision system, use this time to bring the robot to the field to test the vision system because the lighting and coloration of the field will likely be different than that of your practice field. This being said, you may need to adjust the parameters that your vision system uses such as exposure, brightness, threshold, or color range. While on the field, keep in mind that you are not allowed to enable your robot during calibration time.

- **Be prepared for code to break on Pit Day.** Pit Day is stressful for the whole team. Mechanical needs to get the robot weighed and inspected, and the software team needs to test systems and get the robot configured and calibrated. If you prepare yourself to fix code, the Pit Day experience will be a lot less stressful for you.
- **Use the practice field to test vision systems and autonomous routines.** If time permits, ensure that any vision systems and autos actually work before running them on the competition field.

During the Competition

During qualification and playoff matches, you may need to change your code to conform with strategies or systems breaking.

- **Be prepared to solve issues in a time crunch.** Issues can happen at competition, especially if the program team didn't get enough time to sort out all the issues before pit day. A major issue can pop up at any time, so stay in tune with the task on hand. Even if only the code on the driver station computer is being modified, two programmers debugging can be better than one.
- **Know the rulebook.** You may notice the other alliance does something illegal that influences the outcome of the match being played. If this happens, one person can talk with the head referee to get the problem resolved. In addition, keep in mind that many games also have a rule that prohibits teams from pushing code on the field. Generally, it is not enforced during practice matches, but will be enforced during qualification and playoff matches.

Tips for the Technician

The technician is responsible for helping to diagnose and solve any robot-related problems that happen while the robot is on the field. This person is usually a member of the programming team.

- **Stay in communication with each subteam.** Changes in mechanical or electrical systems can easily go unannounced, so it's important to stay up to date with the bot. All modified subsystems should be tested before competition.
- **Know how the mechanical, electrical, and software systems work.** The technician needs to be able to identify and solve a problem quickly. It is easier to do this when the technician knows the robot inside and out.
- **Communicate autonomous routine with the strategy and drive teams early.** The strategy selection might change in the queue, but it's best to know the strategy consensus.
- **Solve as many problems as you can in the queue.** As mentioned earlier, many games include a rule that prohibits teams from pushing code to their robot when it is on the field. This being said, if any problems on the robot can be solved by updating and pushing robot code, do that before the robot is put on the field.

- **Be prepared to write new autonomous routines at your alliance's request.** The alliance's strategy for a match may involve your robot doing something during the autonomous period that it is not yet programmed to do. If the autonomous is simple enough that you think you can get it done before the robots go out on field, write it. Otherwise, be honest with your alliance partners and ask for something simpler.
- **Eliminate as many sources of user error as you can during bot placement.** Check that the ethernet and power cables are connected. Check communications and camera feeds. Check autonomous position. Check all outputs on both driver controllers, read all important values, and communicate any issues immediately. Make a routine of this.
- **During a match, stand by the carts.** Technicians are not allowed to interfere with a match in any way, so they should stay by the carts, away from the alliance station. However, technicians can and should still cheer for the drivers.

Sign-Off

Alas, you have reached the end of the book. Having read the entire thing, you now have the knowledge to write competitive robot code. It is now up to you to apply that knowledge to the current and future FRC games and robots to help your team be a competitive force in competition. I wish you good luck in your journey, and many wonderful FRC seasons.

Sincerely,

Brach Knutson
Foximus Prime (FRC team 3695) alumnus, class of 2021
github.com/BTK203
brach2003@gmail.com

Credits

Brach Knutson - Writer

<https://github.com/BTK203>

Colton Kreischer - Editor

<https://github.com/wh1ter0se>

Jacob Roake - Editor

Glossary

This glossary outlines and defines terms that are used in an FRC programming context, and also displays page numbers that the terms can be found on, if applicable. The terms are sorted in order of A-Z.

- **Alliance:** A group of teams (typically 3 teams). Alliances work together to win matches against other alliances. During qualification matches, alliances are randomly selected every match. During the playoff matches, alliances are selected by the 8 highest-ranking teams.

Pages: 87, 111, 119, 124, 126, 127

- **Alliance Selection:** A period of time during competition, typically before lunch on playoff day, where 8 top ranking teams select their alliances for the playoff matches.

Pages: None

- **Alliance Station:** An area on the field where Alliances control their robots.

Pages: 5, 127

- **Autonomous:** Acting independently, free of all human control.

Pages: 1-2, 36, 40, 42, 48, 77, 87, 111-115, 119-120, 123

- **Autonomous Period:** In most FRC games, the period of time at the beginning of a match where the robots operate entirely autonomously.

Pages: 87, 111-112, 119-120, 126

- **Autonomous Routine:** A sequence of actions that the robot performs autonomously during the autonomous period.

Pages: 42, 48, 50, 66, 87, 111-112, 119-124, 126

- **Bandwidth:** The capacity of a router, or network of routers, to transmit messages.

Pages: 71, 75

- **Branch (git):** An independent version of a project in a repository where changes can be made without affecting the main project itself.

Pages: 28-30, 32-34

- **Brownout:** A dip in the electrical voltage or capacity that a system relies on to function. This happens when the robot tries to draw more power than the battery can provide, likely because it needs to be charged. Brownout can easily be identified on the robot by driving it forwards or backwards at high power draw. A robot that is not browning out will drive smoothly, but a robot that is browning out will jitter.

Pages: None

- **CAN:** Controller Area Network. This network connects the motor controllers to the RoboRIO so that they can be controlled.

Pages: 6-7, 10, 21-24, 37, 56, 60

- **Clone (git):** A duplicate of a repository where programmers can make changes to be uploaded to the repository itself.

Pages: 28-32

- **Command:** Achieves a task on a robot by running code periodically until the task is complete.

Pages: 36, 40-44, 46-53, 57-59, 62, 84, 86, 91, 94-96, 100-101, 104-105, 111-114, 120, 123

- **Command Scheduler:** A utility in the robot code that runs commands.

Pages: 47

- **Command-Based Programming:** A style of programming that uses commands to achieve tasks.

Pages: 35, 40-42, 52, 112

- **Committing (git):** Staging and logging changes made to a clone to be uploaded to a repository.

Pages: 30, 32

- **Comms:** Communication with the robot.

Pages: 54-55

- **Competition Bot:** A Robot that is to be used in a competition, as opposed to a practice robot.

Pages: 22, 125

- **Coprocessor:** A computer on the robot, other than the RoboRIO, that serves to offload heavy calculations or tasks from the RoboRIO.

Pages: 5-6, 41, 43, 107, 124

- **CTRE:** Cross The Road Electronics. A company that makes motors, motor controllers, the PDP, PCM, VRM, and other components.

Pages: 7, 9, 12-13, 15, 21-23, 26, 38, 90, 92

- **Dashboard:** A graphical collection of readouts and video streams which makes controlling and debugging the robot easier.

Pages: 15, 42, 48, 50, 58, 61-63, 65-67, 70, 74-76, 82-83, 85-86, 96, 104, 120-121

- **Driver Station:** A structure which can be placed at the alliance station, on which the Driver Station Computer sits. Most Driver Stations also have a space inside of them where controllers and cables can be stored.

Pages: 124-125

- **Driver Station Computer:** The computer that is used at competition to control the robot.

Pages: 5-6, 26-27, 40, 42, 76, 80, 124-126

- **Encoder:** A device that tracks the rotational position of an axis.

Pages: 6-8, 12-14, 42, 48, 60, 76, 87, 92, 114

- **End-Game Period:** A period at the end of a match, typically the last 30 seconds, where a special final task, such as a climb, can be performed.

Pages: 86

- **E-Stop:** Emergency-Stop. This can be triggered by pressing the Driver Station's spacebar while the robot is enabled. While the robot is E-Stopped, it cannot be re-enabled until it is power-cycled.

Pages: None

- **Feedback Loop:** A continuous cycle where something such as a motor is driven based on the position or state of something else, such as an encoder.

Pages: 87-88

- **Flywheel:** A disk-like device that stores rotational energy by conserving angular momentum. They are typically used in applications where a motor or wheel must turn at a constant velocity.

Pages: 43, 47, 78, 84-85, 114, 120, 122

- **FMS:** Field Management System. This is what is used at competitions to control the states of all 6 robots on the field. This system enables and disables robots in autonomous and teleoperated mode, and allows referees to disable and E-Stop robots as they deem necessary.

Pages: 5-6

- **Fork (git):** A version of a repository that belongs to another owner.

Pages: 28, 33-34

- **Fuse:** An electrical component that shuts off if too much current is passing through it. Typically, when this happens, the fuse must be replaced.

Pages: None

- **Git:** A version control platform used by many FRC teams.

Pages: 4, 28-29, 30-32

- **Gradle:** A Java compiler and project manager that is used by WPILib to manage robot projects.

Pages: 37, 53, 55

- **Gyro:** A component that tracks its own rotational position.

Pages: 62, 100, 114

- **Inhibitor:** A value, typically modifiable on the dashboard, that is used to inhibit (restrain) a throttle or drive value by means of multiplication. For example, if a throttle value of 0.82 has an inhibitor of 0.5, the final throttle value would be calculated as:

$$\text{final throttle value} = \text{throttle} * \text{inhibitor} = 0.82 * 0.5 = 0.41$$

Pages: 83, 86

- **JRE (Java):** Java Runtime Environment. An application that runs Java code. Any computer that is being used to run java code for any reason should have this program downloaded and installed.

Pages: None

- **JDK (Java):** Java Development Kit. A set of tools that assist a developer with managing and working on Java projects. This will be downloaded and installed with WPILib.

Pages: None

- **LabView:** A visual (drag-and-drop) coding language used by some of the teams in FRC, but limited in scalability.

Pages: None

- **Limit Switch:** A component that defines a limit to a system's movement. When triggered, it will typically emit a signal that tells the motor driving the system to stop.

Pages: 15-16, 50, 76, 120

- **Magic Number:** An arbitrary (possibly calculated) number. Defined only once and referenced throughout the code.

Pages: 83

- **Manipulator:** A part of a robot that manipulates a game piece.

Pages: None

- **Match:** A contest between two alliances who play the current season's challenge against each other to score the most points. Matches are typically split into 3 periods: the "autonomous" period which typically lasts 15 seconds, the "teleop" period, which typically lasts 1 minute and 45 seconds, and the "endgame" period, which typically lasts 30 seconds.

Pages: 50, 60-61, 77, 84-87, 120, 124-127

- **Merge Error (git):** A merge error can occur if a locally modified file is updated with a pull from GitHub and there are lines with conflicting changes. It can also occur if conflicting changes are pushed before pulling.

Pages: None

- **Merging (git):** Copying changes made from one branch, clone, or fork to another branch, clone, or fork.

Pages: 32-34

- **Motor:** An electrical component that generates torque.

Pages: 6-16, 21-22, 25, 37-38, 41, 43-44, 47-50, 52-23, 56-57, 59-60, 76, 78, 83-90, 92-96, 100-101, 104-105, 121

- **Motor Controller:** A component that controls a motor.

Pages: 5-7, 9, 12-13, 21-23, 25-27, 56-57, 59, 77-78, 89-90, 92, 94, 96, 100, 104-105, 121

- **PCM:** Pneumatics Control Module. This component controls Solenoids and the Pneumatic Compressor.

Pages: 17-18, 22, 25

- **PDP: Power Distribution Panel.** This component takes power from the battery and distributes it to several different terminals.

Pages: 22, 25

- **PID Loop:** A smart feedback loop that uses proportional, integral, and differential gain to determine output. This type of loop is used to drive a mechanism to a specific setpoint as quickly and accurately as possible.

Pages: 7-11, 13-14, 49, 53, 57, 84, 87-95, 97, 100, 102, 104, 105, 111

- **Pneumatics:** A set of hardware that uses compressed air to generate motion or force.

Pages: 5, 17, 19, 41-43, 84

- **Power Cycle:** Turning the robot off and then back on again.

Pages: None

- **Practice Bot:** An exact clone of a competition bot that is primarily used for driver practice.

Pages: 125

- **Preferences Table:** A table on the dashboard where users can modify different values.

Pages: 64-66, 71, 83, 96, 101, 104-105

- **Pull Request (git):** A request by a programmer to merge changes made on a branch to another branch.

Pages: 32-34

- **Pulling (git):** Downloading recently made changes from a repository to a clone.

Pages: 29

- **Pushing (git):** Uploading changes made on a clone to a repository.

Pages: 32

- **Pushing / Deploying (Robot Code):** Sending robot code to the robot to be run by the RoboRIO.

Pages: 35, 39, 126

- **PWM:** Pulse-Width Modulation. This is a method of controlling the speed of a motor or the brightness of an LED by sending quick “pulses” of electricity to the component. The longer the pulse, the brighter the LED or faster the motor.

Pages: 6

- **Ranking Point:** A point awarded to alliances for completing difficult tasks during a match. The more ranking points a team has, the higher they rank among other teams.

Pages: None

- **Repository (git):** A receptacle where code is stored in Git.

Pages: 28-34

- **Retro-Reflective:** a material property where light that is shined at the material bounces off of it in the direction that it came.

Pages: 105-107, 120

- **REV Robotics:** A company like CTRE that creates motors, motor controllers, and other components to be used on FRC robots.

Pages: 4, 9, 13, 16, 23, 26, 38, 92, 124

- **RioLog:** A console where warnings and errors from the robot can be viewed by the programmer.

Pages: 40, 56

- **RoboRIO:** The main computer of the robot where robot code is stored and executed.

Pages: 5-6, 21, 23, 27, 41-42, 54-56, 70, 90, 100, 105, 107, 110, 124

- **Scheduling:** The marking of a command to be run by the command scheduler.

Pages: 41-42, 44, 47-51, 57, 112, 114, 123

- **SDK:** Software Development Kit. A set of tools that a developer can use to manage and create a program for any platform.

Pages: None

- **Setpoint:** A target metric, such as position or velocity, that is to be achieved by a PID loop.

Pages: 7-8, 88-89, 91-94, 100-101, 104-105

- **Solenoid:** A switch-like component. In FRC, these typically control airflow in a pneumatic system.

Pages: 17-19, 37, 41, 19, 60, 78

- **SSH:** Secure Shell. This is an interface with which a programmer can remotely execute commands on a computer.

Pages: 124

- **Subsystem:** A group of components on a robot that work together to accomplish a task.

Pages: 19, 36, 40-44, 47-53, 59, 62, 76, 94-96, 100-101, 104, 112, 121, 126

- **Teleop:** Tele-Operated. A control mode where humans remotely control the robot.

Pages: 40, 120

- **Tele-Operated Period:** The period in a match where humans control their robots.

Pages: 120

- **Tether:** A cable that connects the RoboRIO to the Driver Station which allows drivers and programmers to have wired control of the robot.

Pages: 5, 124-125

- **Util:** A shortening of the word “Utility.” In FRC programming, this refers to classes, methods, or other tools that exist simply to perform important calculations or tasks at the convenience of the user or programmer.

Pages: 36, 39, 41-42, 58, 64

- **Vendordep:** Vendor Dependency. These must be added to your robot project to gain access to classes and methods other than those from WPILib.

Pages: 27, 37-39, 53

- **Vision:** A system on a robot that uses cameras to identify targets on the field.

Pages: 42-43, 49-50, 86-87, 105-107, 110-111, 114, 120-126

- **VNC:** Virtual Network Computing. An application that allows you to remotely access the desktop of a host computer. This is useful when working with coprocessors such as the Raspberry Pi so that you do not need to connect a monitor to it while it is on the robot.

Pages: 124

- **Voltage Drop:** A drop in voltage which may be caused by low battery, heavy power draws, or both.

Pages: None

- **VRM:** Voltage Regulator Module. This component provides semi-stable 12-volt and 5-volt power for components that have a lower power draw.

Pages: None

- **VSCode:** Visual Studio Code. This is the IDE that is used for robot projects.
Pages: 26, 35, 37-40, 53-55, 61
- **Watchdog:** A timer on the robot that runs the robot project's periodic methods at approximately 50 hz.
Pages: 55, 59, 90
- **WPILib:** A library created for FRC for programming advanced robots.
Pages: 3-4, 19, 26-27, 35-41, 43, 50-51, 53, 58, 61, 80, 96, 100, 112, 121