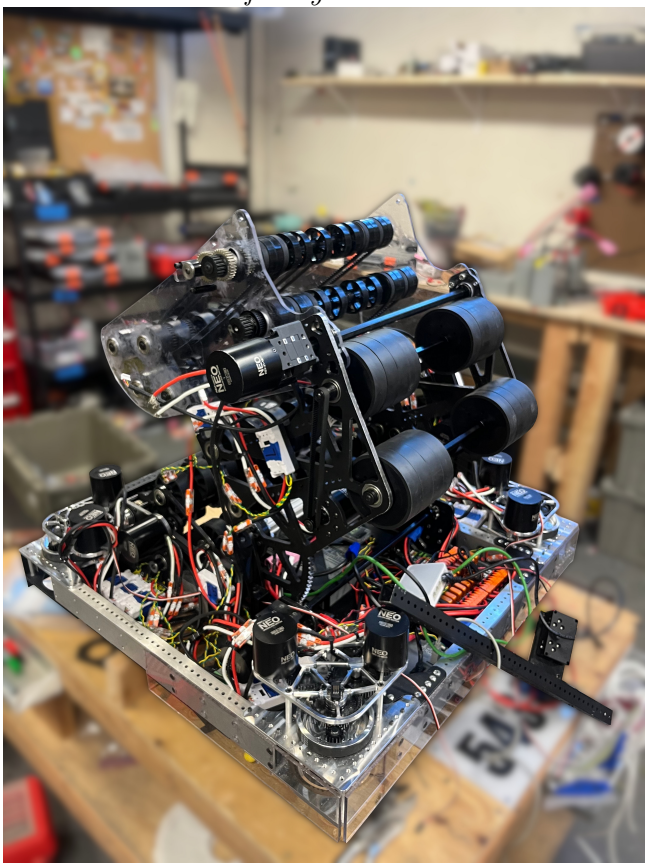
Zachary's guide on how to do stuff

Profanity Included



Zachary Scheiman

May 7, 2025

Contents

Chapter 1

Introduction

Hello, and welcome to my guide on how to do stuff, specifically how to program robots, but I also have some other stuff about networking and all that jazz.

First off I'm hoping that this guide will serve our team (5438) for many years to come, as such I will add some information about who I am and what I did in our team so that you know who wrote this and have some context around what I have to say.

It's also important to understand that while this guide is inspired by both "Kevin's guide on how to do stuff", and "Salomon's guide on how to do stuff", my intent is to make it at least twice as readable as the WPILIB docs, and actually useful unlike the WPILIB docs. No one likes bad docs (or the WPILIB docs).

1.1 Who Am I, and some context to this guide

My name is Zachary Scheiman (class of 2024), I joined the Tech Terrors right before our Montgomery event in 2023 (Charged Up), and had to reprogram the robot in roughly one week. After the season ended I was awarded the role of Controls Captain (note that controls contained both programming and electrical subteams).

As controls captain during the 2024 season I, and the other programmers were able to successfully implement vision with proper april-tag detection, swerve with minimal drift, and autos which were semi reliable. All this was important, and helped us in winning our Mount Olive event and getting a blue banner. However upon reflection this season was a setup season for y'all. What I mean

by this is that our 2024 season was not about winning, but about learning how to build, construct, and program a robust robot.

My intent for this guide is to dump all my knowledge into a place where our team can access and use it for many years. However I am bound to miss something essential so if needed contact me. My information is below.

1.2 Contacting Me

Please, please, please don't hesitate to reach out to me if you need help I want to see you guys win, have fun, and most importantly learn.

You can always find me in the team Discord (my username is @squibid), and if I don't respond in there feel free to email me@zacharyscheiman.com.

1.3 A small side note

Not everything in this guide will always be up to date (duh), and this guide will not be as verbose as the official WPILIB docs, however because our team has seemingly experienced every possible problem with our given hardware/software stack I have many solutions and hacks to get your robot working just in the nick of time, and warnings of what not to do (please read this guide before buying something expensive).

Chapter 2

Driver Station

The driverstation is nearly the most important part of your robotics experience if it doesn't work you're fucked. Before I go any further I'm going to make a quick distinction for the sake of comprehension: in FRC the term *driverstation* is used for both the computer being used to control the robot, and the software used to communicate with the robot. In this guide I will refer to the computer as the computer and the software as the driverstation.

There's not much to say about the software itself, there are some useful logging tools (more info may be found here on how to use them).

It's important to make sure your computer stays working for a long time don't make the same mistake I did; get a computer with a good battery life and set battery limits to maintain it as long as possible. Also if possible try to get one with USB-C charging as it's more universal than everything else (it's *probably* what your phone uses)

2.1 Common Problems

- Robot is E-Stopping for no reason, and dropping comms
  - This happens when your computer goes to sleep with the driverstation open, it's best practice to restart the driverstation after waking your computer from sleep.

Chapter 3

I2C

3.1 WARNING!

TL;DR: **Don't use sensors over I2C!** Instead use Beam Break sensors.

The I2C port on the RoboRio is, and has been broken and will eventually cause the RoboRio to crash. It is also important to note that though it may not crash immediately you will encounter CommandScheduler loop overrun errors eventually which will cause similar symptoms to a brownout.

Instead of using Color Sensors it is advised to use Beam Break sensors as they are more reliable, and do not cause your robot to turn into a glorified paper weight. If you **MUST** use a device requiring I2C please find the pinout of the MXP port on the RoboRio and plug it in there.

3.2 Programming

3.2.1 Initializing an I2C color sensor

Note: the following code is for the Rev color sensor and there will be differences between different I2C sensors.

To declare and initialize it you should plug it in over MXP and do the following:

```
ColorSensorV3 colorSensor = new ColorSensorV3(I2C.Port.kMXP);
```

If you **MUST** plug it into the designated I2C port on the RoboRio you have to replace `kMXP` with `kOnboard` (Don't do it).

3.2.2 Configuring an I2C sensor

Color sensors are fairly complex, and therefore I will not be going into the whole spec here, if you wish to read through all of it, it can be found here: APDS-9151.

To configure the accuracy of the color sensor there are two main settings you need to tinker with. These are the Proximity, and the Measurement Rate.

3.2.3 Proximity Resolution

- Determines how accurate the color sensor is at close, and far ranges
- Max of 11bits
- Minimum of 8bits **(default)**

3.2.4 Proximity Measurement Rate

- Determines the rate in which the color sensor takes measurements.
- 6.25ms, 12.5ms, 25ms, 50ms, 100ms **(default)**, 200ms, 400ms

3.2.5 Code Example

This code snippet:

```
colorSensor.configureProximitySensor(
                            ProximitySensorResolution.kProxRes11bit,
                            ProximitySensorMeasurementRate.kProxRate6ms);
```

Configures the color sensor to refresh the proximity readings to be as fast as possible. Be careful with this as if you use the color sensor over the kOnboard port you may encounter the following error:

```
CommandScheduler loop overrun
```

More info on this error can be found in chapter 12.

3.2.6 Retrieving Data

To Retrieve proximity data from your color sensor you can use the following method:

```
colorSensor.getProximity();
```

Chapter 4

Limelights

4.1 WARNING!

TL;DR: **Don't use limelights** Instead check out chapter 15 for my guide on vision.

Both limelight hardware and software have, and continue to be a scam! **DO NOT BUY IT!** Limelights provide very limited resolution, and frame rate for the outrageous price of $400. The cost simply does not reflect the quality you will receive, and will in fact limit your robot as vision becomes more essential.

Instead of using a limelight I urge you to read my guide on vision which can be found in chapter 15 as I have recommendations that will charge up your your robots vision software + hardware stack.

Chapter 5

Electrical

Because I don't know much about electricity, and how it works I'm just gonna go over some very stupid problems that I know how to fix. I also have some tips so that you don't fry half the components in the robot (trust me it's not fun).

5.1 Tips and Notes

- **Always** power off the robot before plugging or unplugging wires, I learned this the hard way when I fried both our VRM and Network Switch at Lehigh
- Notation
  - V: voltage
  - C: common/ground
  - I: current

5.2 RoboRio & it's common problems

A super advanced brick that learned how to be sentient for the low low price of $485, I'd value it at $200 tops. The following is a compilation of common problems that I've encountered:

5.2.1 Not turning on

If you're RoboRio isn't turning on first make sure it's getting power by using a multimeter. If it is getting sufficient power then there's a high chance there's shavings inside the Rio, at this point you should have someone clean it out.

5.2.2 Devices plugged into the analog ports returning funky values

You've got shavings in your RIO. There's likely one or two in the DIO section that's shorting the 5V power to the Analog ports. At this point you're gonna need to clean your RoboRio.

5.2.3 Cleaning a RoboRio

- Remove the casing
- Firmly bang the Rio on a table until there's not shavings coming out of it
  - Please don't bend any pins or split the board in half or I will personally come back to the shop and stick a brick of c4 onto to the robot
- Air dust the shit out of it
  - It should be clean enough to eat off of it (please don't)

If that didn't work you can either give up and buy another overpriced brick, or get some isopropyl alcohol and wipe down the whole thing.

5.3 PDH

The thingy that powers your whole robot.

5.3.1 The "Switchable Port"

TL;DR: don't plug **anything** into it!

On some newer PDH's there's a 12V low gauge port which is labeled "switchable" because there's a physical switch to turn it off. PLEASE NEVER PLUG

ANYTHING INTO IT not only is it against the rules to plug your RIO into it, but it's also just a stupid idea as historically it's caused us major problems. [1]

5.3.2 Somethings not receiving power

Check your wires, make sure they aren't shorting, and that they haven't been decapitated. If they're fine make sure your fuses haven't been incinerated.

5.4 CAN

For a full guide on CAN check out chapter 14.

[1] Imo it's really fucking stupid to have a switchable port when you can literally just take out the fuse, but I'm sure the engineers at REV had a half decent reason to essentially remove a port from the PDH.

Chapter 6

Networking

Networking is one of the most important parts of your robot, if you can't connect to it your can't use it (duh). Due to it's importance it's necessary to have a network setup that won't break. Below I've outlines many of the parts that I recommend using, and all the things that may be wrong with them that you can overcome.

6.1 Radio(s)

As of 2024 there are two different radios: the OpenMesh OM5P and the VH-109. Due to the VH-109 coming out at the end of the 2024 season I don't have any experience with it, and for that reason I'm only able to cover the old OM5P. If you have the ability to use the VH-109 I recommend doing so as it's objectively better (I'll include a link to the docs).

6.1.1 OpenMesh OM5P

6.1.1.1 Flashing

When flashing your radio it's important to select 5ghz over 2.4ghz as it can have less latency than 2.4ghz (but never more). Also make sure to plug the ethernet into the first port (from the left).

Figure 6.1: OpenMesh OM5P wireless router

6.1.1.2 WARNING!

The second port is known to have issues, and I'd recommend plugging the first to a network switch and then plugging everything else into that.

6.1.2 Vivid-Hosting VH-109

Figure 6.2: VH-109 wireless router

Docs may be found here, good luck!

6.1.3 Ethernet

6.1.4 POE

When it comes to powering your radio it's imperative for it to maintain a connection throughout the match, in order to do this you must ensure that it does not go through power spikes due to other devices taking up power on the robot. Because of this using a REV POE injector—see figure 6.3a— without a buck boost converter such as the CTRE VRM can and will cause you to lose communication with your robot.

If you do not have enough open ports on your buck boost converter I'd recommend going with the REV Power Module pictured in figure 6.3b.

Figure 6.3: POE Injectors

(a) REV POE injector

(b) REV Power Module

6.1.5 Tethering

When it comes to tethering there are two ways to go about it, you can use ethernet, or tether over USB-B. Personally I prefer ethernet as not only is it faster, but it's also (usually) locks into the port meaning a wrong tug won't take it out causing you to need to re-tether wasting valueable time.

Before you go and tether to your robot make sure you have someone to handle the cable so it doesn't get damaged, and—because it will get damaged—you should bring a spare.

Chapter 7

Joysticks

7.1 PS4 / PS5

These two are functionally the same in code. Because of this I'm going to show examples for PS4, but they can easily be changed to PS5 like so:

`PS4Controller -> PS5Controller`

7.2 Xbox

There is one generic Xbox class that provides an interface for all Xbox-ish controllers.

7.3 Programming

7.3.1 Initializing

Initializing a controller is as simple as specifying the port it's plugged into like so:

```
PS4Controller newControllerPS4 = new PS4Controller(1);
/* or for xbox */
XboxController newControllerXbox = new XboxController(1);
```

7.3.1.1 Button Binds

Binding a button is also very simple requiring you to specify the controller, button, and outcome like so:

```
new JoystickButton(newControllerPS4,
    PS4Controller.Button.kCircle.value).onTrue(
/* callback */
new InstantCommand(() -> System.out.println("Circle pressed"))
);

/* or for xbox */
new JoystickButton(newControllerXbox,
    XboxController.Button.kA.value).onTrue(
/* callback */
new InstantCommand(() -> System.out.println("A pressed"))
);
```

7.3.1.2 Events

There are a few events that you can use to change how your bind is run, some of the events that exist are not very useful, and therefore I will only present you with the most useful ones that I could find:

- `.onTrue(Command)`
  - Runs the `Command` once when the button is pressed
- `.whileTrue(Command)`
  - Runs the `Command` continuously while the button is pressed

Both of the event above have opposites (named `.onFalse` and `.whileFalse`) which do the inverse of their True counterparts.

7.3.1.3 Callback

Note that the instant command may be replaced by any valid command (examples may be found in chapter 11.

7.3.2 Ports

When Initializing a controller you have to specify the port. The port is determined by the order that the controllers show up in the Driver Station.

While this shouldn't change it's always good to make sure that they're in the correct order before a match.

7.3.3 Spec

It's important that you initialize and bind the controllers in specific areas to keep your code as readable, and clean as possible. Therefore I will provide a short example of how to do so:

In `RobotContainer.java`:

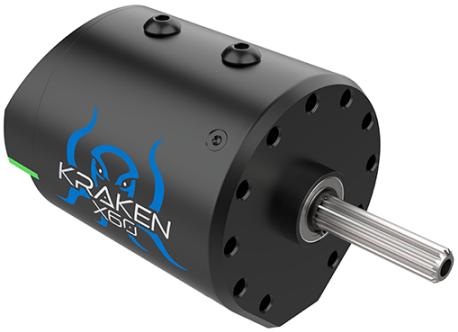
```
import edu.wpi.first.wpilibj.PS4Controller;
import edu.wpi.first.wpilibj.XboxController;

public class RobotContainer {
  /* joysticks should be initialized here like so:
   * make sure they are public so that you can access them outside of
   * the robot container
   */
  public final XboxController driver = new XboxController(0);
  public final PS4Controller operator = new PS4Controller(1);

  public RobotContainer() {
    configureBindings();
  }

  /* separate function to keep bindings organised */
  private void configureBindings() {
    new JoystickButton(operator,
        PS4Controller.Button.kSquare.value).onTrue(
      new InstantCommand(() -> System.out.println("Do something
          here"))
    );

    new JoystickButton(driver, XboxController.Button.kX.value).onTrue(
      new InstantCommand(() -> System.out.println("Do something
          here"))
    );
  }
}
```

Chapter 8

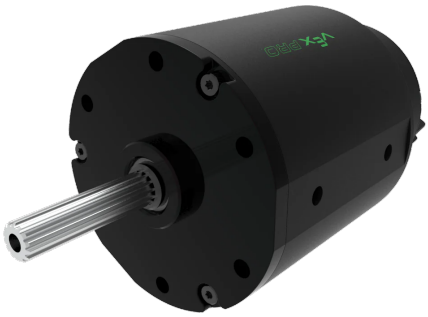
Motors

For FRC there are a few motors that you need to know how to use. The main ones are (Pictured in figure 8.1):

- Talons
  - 8.1a Kraken
  - 8.1b Falcon
- Sparks
  - 8.1c Neo Vortex
  - 8.1d Neo
  - 8.1e Neo 550



(a) Kraken x60



(b) Falcon 500

(c) Neo Vortex

(d) Neo

(e) Neo550

Figure 8.1: Motors

The reason they are separated by “Talons”, and “Sparks” is twofold, the first being because the motors in their respective groups can be initialized with the same method. The second being that they are from two separate brands. Despite this there is one commonality between the two, that being: CAN (more info can be found in chapter 14).

8.1 Talons

8.1.1 CAN ID

For Talons you need to open up Phoenix Tuner X, click on the motor you want to configure and on the left hand side you should be able to set the CAN ID number.

8.1.1.1 Tips

If you are having trouble with Phoenix Tuner X not connecting to the robot try the following:

If you're connected over ethernet or wifi:

- open up the hamburger menu on the left hand side
- in the box on the bottom left of the app type in "5438"

If you're tethering over USB-B:

- open up the hamburger menu on the left hand side
- on the right of the box in the bottom left of the app click the dropdown menu
  - select the option that says "roboRIO USB"

8.1.2 Programming

To initialize the motor:

```
import com.ctre.phoenix6.hardware.TalonFX;
TalonFX talon = new TalonFX(1); /* "1" is the CAN ID you've already
configured */
```

Running the motor is as simple as:

```
talon.set(0.9) /* percentage from 0 to 1 where 0.9 is 90% */
```

8.2 Sparks

8.2.1 CAN ID

For Sparks open up the Rev Hardware Client, and click on the device you want to configure, change the CAN ID, and the click "Burn Flash" at the bottom of the app.

8.2.1.1 Tips

Rev Hardware Client has only one method of connecting to the robot which is over a USB-C tether. All you have to do is connect to any Spark in the CAN loop, and you should be able to access every Spark (assuming the robot is powered on).

8.2.2 Programming

To initialize the motor:

```
import com.revrobotics.CANSparkMax;
import com.revrobotics.CANSparkLowLevel.MotorType;

/* "1" is the CAN ID you've already configured, and
 * "MotorType.kBrushless" is the type of motor, this can be either
 * brushed or brushless. If you're using one of the Spark motors I've
 * mentioned above it's brushless
 */
CANSparkMax spark = new CANSparkMax(1, MotorType.kBrushless);
```

And running it:

```
spark.set(0.9) /* percentage from 0 to 1 where 0.9 is 90% */
```

Chapter 9

Encoders

There's a few types of encoders which tend to be the most useful and those are the ones which I will detail here, all others are available on the docs here.

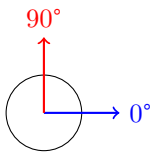
9.1 DutyCycleEncoders

These encoders tend to be simple to fit into the robot, and provide a relatively easy way of maintaining the current position of a mechanism. To use one you initialize it like so:

```
DutyCycleEncoder encoder = new DutyCycleEncoder(0);
```

Where `0` is the number of the DIO port where your encoder is plugged in. Once you've managed to initialize it you need to ensure that you've set the correct offset for your encoder to maintain consistency between code and real life. Here's a diagram to help illustrate my point:

Imagine that we have a mechanism, pictured below, and due to the way the encoder is positioned it's reporting measurements which are 90°s off (computer reading is red, actual position is in blue):



In order to fix this you must set an offset like so:

```
encoder.setPositionOffset(0.25);
```

The reason we've inputted `0.25` instead of 90°s is because DutyCycleEncoders use rotations instead of radians or degrees meaning that 0 and 1 are equal to 0°and 360°.

In order to actually use the encoder you can simply call the `getDistance` method like so:

```
double distance = encoder.getDistance();
```

Where the return value is a double containing the current distance from the origin, and not the total distance travelled because why not.

9.2 Relative Encoders

This type of encoder is specific to Sparks (see 8.1 for motors) as it uses the builtin encoder in a Spark Max and as such requires an already initialized motor.

```
/* dummy motor for example */
CanSparkMax motor = new CanSparkMax(0, MotorType.kBrushless);

RelativeEncoder encoder = motor.getEncoder();
```

Once this is done you can use it like so to get the velocity:

```
double velocity = encoder.getVelocity();
```

or the position:

```
double position = encoder.getPosition();
```

Although I'd be careful with this one because it only returns the rotations that the motor has been through as opposed to the actual position. In order to get the actual position you'd need an actual encoder, however for simply checking weather a motor is up to speed this is useful.

Chapter 10

Subsystems

Subsystems are an essential part of your robot and are used to initialize motors, sensors, and more. Here is an example of a basic subsystem:

```
package frc.robot.subsystems;

import edu.wpi.first.wpilibj2.command.SubsystemBase;

public class ExampleSubsystem extends SubsystemBase {
  public ExampleSubsystem() {
    /* constructor for the class used to initialize motors and whatnot */
  }

  @Override
  public void periodic() {
    /* run every CommandScheduler loop (defaults to every 20ms) */
  }
}
```

10.1 Proper use of a subsystem

It is important to note that a subsystem is a WPILIB specific concept, and you won't find the same thing outside of said ecosystem. With that said this means that you must be careful to use subsystems correctly as to not cause problems in your robot.

10.1.1 Spec

This is a "specification" on how to use a subsystem. It is important to note that this is not the required way, but the way I found to be easiest, and most organised.

- Subsystems should only have one instance
  - To keep the code organised the instance should be declared in `RobotContainer.java`
- A subsystem should have a default command linked to it (for more information go to chapter 11)
  - Default commands are used to define the behavior of the things initialized in the subsystem.
- Make variables public unless you explicitly want to stop the user from doing something

Chapter 11

Commands

Commands, similar to subsystems are essential to programming your robot. They provide a simple yet intuitive way to define actions linked to subsystems.

Here's a simple example of how to define a command:

```
package frc.robot.commands;

import edu.wpi.first.wpilibj2.command.Command;

public class ExampleCommand extends Command {
  public ExampleCommand() {
    /* handle arguments passed in via the constructor */
  }

  @Override
  public void initialize() {
    /* run once when the command is first called */
  }

  @Override
  public void execute() {
    /* run every CommandScheduler loop (defaults to every 20ms) */
  }

  @Override
  public boolean isFinished() {
    /* run every CommandScheduler loop (defaults to every 20ms)
     * If this returns true the command is ended and the end method is
     * called
     */
    return false;
```

```
  }

  @Override
  public void end(boolean interrupted) {
    /* run when the command is told to end
     * interrupted is set to true if the command has been canceled
     */
  }
}
```

11.1 Types of Commands

11.1.1 Wait Command

A Wait Command can be used to wait for something to happen or some time to pass.

Here's an example of waiting for time to pass:

```
new WaitCommand(1); /* waits one second and then ends */
```

And waiting for a condition to become true:

```
import java.util.function.BooleanSupplier;
BooleanSupplier condition = () -> false;
new WaitUntilCommand(condition); /* waits until condition returns true */
```

11.1.2 Instant Command

An Instant Command is mainly used to wrap around methods that don't extend the Command Class.

Here's an example of how to use it:

```
public void myMethod() {
  System.out.println("Something Very Important");
}
new InstantCommand(() -> myMethod());
```

When run this will print out the following:

```
Something Very Important
```

PS: the `() ->` syntax is called a lambda, and it allows you to put methods inline instead of declaring one just to use it one time.

You can also require subsystems using an Instant Command, however if you really need that I'd instead look at making your own Command.

11.1.3 Sequential Command Group

A Sequential Command Group is quite intuitive, it's used to run commands one after another.

Here's an example of how to use it:

```
new SequentialCommandGroup(
  new InstantCommand(() -> System.out.println("1")),
  new InstantCommand(() -> System.out.println("2")),
  new InstantCommand(() -> System.out.println("3")),
  new InstantCommand(() -> System.out.println("4")),
  new InstantCommand(() -> System.out.println("5"))
);
```

When run this will print out the following:

```
1
2
3
4
5
```

11.2 Creating a command

When creating a command it is important to think through what the command needs to do. For example If your command needs to take full control of a subsystem you will need to make sure to use the `addRequirements()` method to stop other commands from messing with your subsystem.

Here is an example of how to do exactly that:

```
package frc.robot.commands;
```

```
import edu.wpi.first.wpilibj2.command.Command;

public class ExampleCommand extends Command {
  public ExampleCommand(ExampleSubsystem exampleSubsystem) {
    addRequirements(exampleSubsystem);
  }
}
```

11.3 Calling a command

To call a command like a method you can simply do:

```
new ExampleCommand().schedule();
```

While this is important to understand we don't want to have to schedule every command on our own, as such we let the command scheduler (chapter 12) do all the work for us by registering events that we want the command to be run on, and letting it handle the execution for us.

Here's how to tell the CommandScheduler to wait for a button press to run a command:

```
/* This code example won't work in practice because:
 * 1. you need to initialize a controller inside of a class
 * 2. you need to add a listener inside of a method
 */
import edu.wpi.first.wpilibj2.command.button.JoystickButton;
import edu.wpi.first.wpilibj.XboxController;

/* initalize a new controller */
XboxController driver = new XboxController(0);

/* add listener for a button on the controller */
new JoystickButton(driver, XboxController.Button.kY.value).onTrue(new
    InstantCommand(() -> System.out.println("Hello World")));
```

It is important to note that this only needs to be declared once, and that it's best practice to declare it in your `RobotConstainer.java`.

Chapter 12

Command Scheduler

12.1 How does it work?

All information in this section has been pulled from the wiki.

The command scheduler runs (by default) every 20ms and runs in 3 steps:

- Run Subsystem Periodic Methods
- Poll Command Scheduling Triggers
  - this means registering new commands which have been scheduled and running their initialize method
- Run/Finish Scheduled Commands
  - executes every currently scheduled command
- Schedule Default Commands
  - the "Default Command" refers to a subsystems default command
  - This step also runs the initialize method from the default command

12.2 Using the Command Scheduler

Both commands (chapter 11) and subsystems (chapter 10) are interfacing with the command scheduler in the background, but if you need to do something more complex you're gonna need to directly interface with it. Therefore here are a few special things you can do with the command scheduler:

12.2.1 Asynchronous Interrupt

An Asynchronous Interrupt allows you to run code as soon as something in your program changes.

```
DigitalInput d = new DigitalInput(0);
AsynchronousInterrupt interrupt = new AsynchronousInterrupt(d, (a, b) ->
    {
  /* put code here that you wan't to run when the digital input changes
  */
  System.out.println("Inputted");
});
```

When run this code will print "Inputted" after the Digital Input is triggered. The advantage over doing this in a regular command is that this is triggered immediately whereas a command can run anywhere from 20-0ms after the Digital Input is triggered.

12.2.2 addPeriodic

An addPeriodic is a way of running certain parts of your program faster than others. However unless you **need** to run something faster than everything else I don't recommend doing it as it may take more cpu resources, and cause a Command Scheduler loop overrun.

With that said there are some cases in which it's useful and therefore this is how you could do it:

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

addPeriodic(() -> {
  System.out.println("Hi");
}, 0.01);
```

When run this will print "Hi" every 10ms.

12.3 Common Problems

12.3.1 Command Scheduler loop overrun

The command scheduler loop overrun error is one of the worst things you can encounter. At it's essence the command scheduler loop overrun error signifies that the code is running too slow, and WPILIB is trying to catch up, but is unable to.

12.3.1.1 Debugging

Typically you can find text like so near the command scheduler loop overrun error message:

```
...
SwerveSubsystem.periodic(): 0.00037s
LEDSubsystem.periodic(): 0.00000s
SpeakerSubsystem.periodic(): 0.086499s
ClimberSubsystem.periodic(): 0.000018s
ProblematicSubsystem.periodic(): 0.210000s
...
```

If you see something with an abnormally high second count that is where the command scheduler is hanging, and the code is in need of optimization. In the above example you can see that the "ProblematicSubsystem" has a second timing of 0.210000s which is 21ms. Because the ProblematicSubsystem is taking longer than 20ms it's causing the whole command scheduler to hang.

12.3.1.2 Causes & Solutions

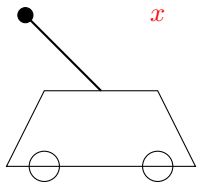
There are many causes of the command scheduler loop overrun error, however some the of main ones are as follows:

- Using I2C over the builtin port
  - More info on this can be found in chapter 3
- Retrieving data more than once
  - For example: when dealing with cameras getting an image from them takes a log of processing power therefore the result should be stored in a variable to ensure that it's not requested more than once per command scheduler loop

Chapter 13

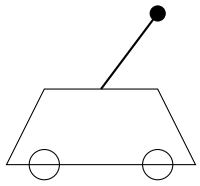
PID

PID is a formula which can be used to get a motor to turn to a specific position. Here's an example where we want our robot to move it's arm to a specific position (in this case x):



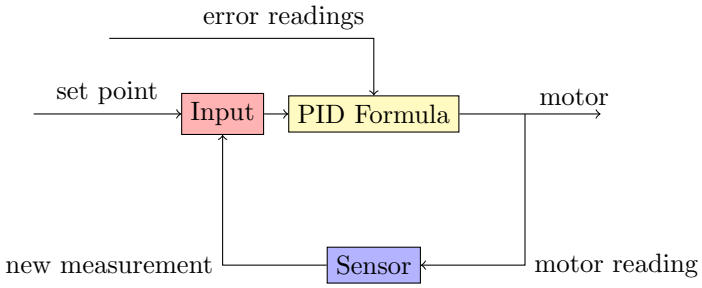
Without PID you'd have to guestimate when it reached x however with PID you are able to tell it to go to x encoder position and it will go there. The only slight caveat is that you need to tune it to get there perfectly.

End example:



13.1 Concept

When it comes to controlling a system which requires precision you need to create something called a feedback loop. A feedback loop is when you change something based on another reading here's a look into the one that you are implementing when you use a PID loop:



The PID formula:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de}{dt}$$

Where t is time. More detail may be found here: introduction to pid.

13.2 Programming

13.2.1 Initializing your PID controller

To start we're going to need 3 things:

- a motor to control
- a motor encoder to get feedback from
- a pid controller

In the following examples I'm going to use code for Spark Max's because that's what I'm most familiar with however it should be pretty straight forward to convert it to something else.

Assuming you've set up your motors properly it should look something like this:

```
CanSparkMax myMotor = new CANSparkMax(1, MotorType.kBrushless);
```

And for your encoder (this code may change drastically depending on your encoder):

```
DutyCycleEncoder myMotorEncoder = new DutyCycleEncoder(1);
```

Now the important part, Initializing the pivot PID controller:

```
int kp, ki, kd;
PIDController myMotorPID = PIDController(kp, ki, kd);
```

The PIDController takes 3 arguments on initialization (in order of importance):

- kp
  - the most essential argument, kp controls the speed at which the motor turns towards the setpoint depending on your motor and what you need it's not abnormal to have values around 10
- kd
  - this argument determines how fast the motor should go when it has overshot its destination this is nearly always important to have set
- ki
  - this argument increases the speed at which the formula corrects for error in most scenarios this is not the solution you're looking for, and **it will cause more problems by using it**

13.2.2 Using the output

To actually make the motor go to a specific position you're going to need to pull a few things together. The first thing you're going to need is the `PIDController` in the PIDController class there is a method which enables you to calculate the motor output based on where you want to be and where you are in encoder units of measurement. You can do so like this:

```
double currentPosition, desiredPosition;
double motorSpeed = myMotorPID.calculate(currentPosition,
    desiredPosition);
```

To get your current position you need to take a look at the `DutyCycleEncoder`:

```
double currentPosition = myMotorEncoder.getDistance();
```

More information on encoders may be found in chapter 9.

Your desired position should be predetermined based off of an encoder reading. I recommend creating a constant in `Constants.java` with a descriptive name such as: `defaultShootingPositon`.

Now that you understand how to use the calculate method it's also important to understand where to run it, and what to do with it's output. First this must be run in some kind of periodic loop, I recommend creating a command—see chapter 11—which can be set as the default command of the subsystem in which your motor resides. Once you've done that inside the `execute` method you can put the following:

```
motorSpeed = myMotorPID.calculate(speakerSubsystem.pivotEncoderDistance,
    desiredPosition);
myMotor.set(motorSpeed);
```

This will set the motors speed to the correct number every 20ms. If you need a fully implemented example take a look at this bit of our 2024 code.

13.2.3 Tuning your PID controller

The most important part besides actually using the output of your PID controller is to make sure it does what you want. I'm going to break this up into multiple steps, but it's important to understand that there is no definitive way to tune PID, and up to a certain point it takes a bit of finesse.

- Set kp to 1
  - If this is to high, and your mechanism is overshooting try and half it (continue this till you see a difference)
- Increase/Decrease (by .1) until your mechanism starts getting close to the target point at this point if you're seeing no improvement repeat these steps with kd
  - This part is impossible to get 100% accurate

An imperative thing to understand about this process is that it's not uncommon to have kp values at 20. However this is not the case for kd which as per the formula controls the reaction to the last error, if this is too high it will cause oscillations. For ki I recommend leaving this at 0 as it's job is to cumulatively correct errors, and if there are too many your robot may try and rip itself apart.

13.3 Alternatives

The alternative that I know of:

- Motion Magic
  - Motion Magic is specific to CTRE motors like Krakens
  - Motion Magic, unlike PID, slows down as it approaches the target like you would in a car

Chapter 14

CAN

This chapter will be a brief overview on how CAN works under the hood rather than an explanation on how to use it. If that's what you're looking for I'd suggest checking the official docs for your hardware, or if it's a motor checking out chapter 8 for more info.

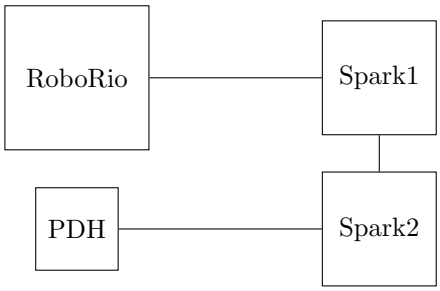
With that said lets start with an explanation of what CAN is. CAN stands for Controller Area Network and is a bus that is used to send signals to components in a vehicle. Due to it's versatility it's been adopted by WPILIB and many FRC component manufacturers.

14.1 How does it work?

A CAN network consists of many devices which are always listening for new messages, when one of these devices sends a message all devices on the network receive it, and individually determine if it's useful.

In general CAN is fairly simple to setup, and you should be able to infer the necessary information from figure 14.1.

Figure 14.1: Simple CAN network/loop



For more information you can take a look at the full documentation here.

Chapter 15

How to build a competent FRC vision system

I need to start this section with a warning: **Don't use Limelights!** Now that that's out of the way:

This section of the guide is one of the most important and should be read carefully. I've broken this into two parts: software, and hardware. It's important to note that the hardware is very interchangeable, but the software is not! There is a very small amount of software for vision in FRC and it is imperative that you use the right stuff.

15.1 Software

The software stack for vision is very simple. All you need is PhotonVision.

There is an alternative to photonvision called limelight which is technically easier, however much worse in quality and accuracy and therefore I don't recommend using it. But it you must head, over to chapter 4 for more information.

15.1.1 Getting PhotonVision

15.1.1.1 Downloading

The first thing you're gonna need to do is download the latest version of photonvision which can be found here: latest release, scroll down to the assets, and download the `.img.xz` corresponding to the hardware you use. For example if I had an Orange Pi5 I would download:

`photonvision-v2024.3.1-linuxarm64_orangepi5.img.xz`.

15.1.1.2 Installing

Now that you have the image you need to flash it to your hardware. If you don't know how to do this just download balena etcher, and it should be pretty straight forward.

15.1.2 Configuring Your Camera

15.1.2.1 Calibration

If you don't currently have a flashed device in front of you, and just want to see what I'm talking about go to the photonvision demo.

1. Go to the photonvision dashboard
    - For example on our 2024 robot it was `10.54.38.89:5800`, but you can *usually* go to `photonvision.local` without any problems.
2. Click on the "Cameras" button on the left hand side
3. Choose your desired resolution
    - Make sure to choose one with a frame rate of at least 40fps unless you really need to see far away objects
4. Click the "Start Calibration" button
    - Make sure you have a chessboard from photonvision that way you can calibrate the camera, if you don't one can be downloaded by clicking the "Generate Board" button under the "Start Calibration" button

15.1.2.1.1 Tips for calibration

- Make sure the chessboard covers as much of the screen as possible before taking a picture
- (across all your pictures) Make sure to cover every corner of the screen
- The calibration process helps photonvision determine what is and is not an april tag, taking extra time now will help you in the future

15.1.2.2 Pipeline

A pipeline is a configuration for a camera in photonvision. When creating a new pipeline you can choose from a few tracking types such as:

- Reflective
- Colored Shape
- AprilTag
- Aruco

For most cases the one you're gonna want to use is "AprilTag".

15.1.2.2.1 AprilTag Pipeline Settings

April Tags are pretty easy to detect if you've got a good camera. From what I've found the optimal settings are as follows:

- Decimate: 1
- Blur: 0
- Threads: 1 - maximum
- Refine Edges: true
- Decision Margin Cutoff: 0
- Pose Estimation Iteration: 40

It's also important to change the Target Family to the relevant April Tag family (it will most likely be 36h11, but you should double check the official docs).

15.1.2.3 Exposure

Just turn on Auto Exposure.

15.1.2.4 Brightness

Adjust until the image you're seeing is clear, from my experience the default is good.

15.1.2.5 Gain

This one is tricky, I've always just messed with it until I have a visually clear image. Typically you can just set and forget, but it's always good to double check that your camera can see stuff when the lighting you're in changes.

If you see the red and blue camera gain sliders you're typically gonna want red higher than blue. However if that doesn't work just tune it like it's PID.

15.1.3 Programming

15.1.3.1 Initializing a Camera

To initialize a camera you must do the following:

```
/* note the PhotonCamera object is private to ensure that others can't
 * waste resources by getting an image by themselves
 *
 * Make sure to replace "Camera_Module_v1" with whatever camera you are
 * using (the name can be found on your photonvision dashboard)
 */
private PhotonCamera cam0 = new PhotonCamera("Camera_Module_v1");

/* because we've made the PhotonCamera private we have to give others a
 * way to access an image if need be therefore we create a public image
 */
public PhotonPipelineResult cam0Image;
```

It's important to note that photonvision doesn't check if there's a photonvision server running on your robot so if for some reason it isn't and you don't want your robot to crash you'll need to check for it manually.

This can be done by checking if photonvision is available on your network tables instance. Here's a simple example:

```
/* the NetowrkTable instance should be private to ensure that no one
 * else can access it.
 */
private NetworkTable NT =
    NetworkTableInstance.getDefault().getTable("photonvision");

if (NT == null) {
  /* photon vision is unavailable */
}
```

15.1.3.2 Reading Values

Getting information from your camera is pretty easy, adding onto the previous code do the following:

```
/* this should probably be run periodically to make sure you have an up
 * to date image from the camera */
cam0Image = cam0.getLatestResult();
```

15.1.3.3 Processing Data

To process the data you first have to make sure that you don't have any uninitialized variables. It's important to note that some variables may be wrapped inside of optionals, if you don't know how to deal with these check out StOptional, which is a small snippet of code I wrote to reduce those pesky optionals down to their original types[1].

Now, assuming that you've dealt with all potentially null variables processing data from photonvision is fairly easy, and can be done like so:

15.1.3.3.1 Detecting April Tags

To check if a certain april tag is in view of your camera you just have to check if it's available within the camera image, which can be done like so:

```
long desiredTagId = 1L;
```

[1] Optionals are not inherently bad, but the way that the photonvision devs have decided to use them is—in my opinion—so horrendous that I feel as though it's just easier to drop it back to its original type and go from there.

```
for (PhotonTrackedTarget target : cam0Image.getTargets()) {
    if (target.getFiducialId() == desiredTagId) {
        return target;
    }
}
```

15.1.3.3.2 Pose Estimation

Please refer to the photonvision docs here as they are much more accurate for information on pose estimation.

Chapter 16

Swerve Drive

While it would be super fun to code your own SwerveDrive I would recommend only doing it in an off-season, and as soon as the season starts if you do not have a working swerve drive resort to using YAGSL [1] to ensure that you've got a working drivebase.

Therefore instead of this being focused on how to make one I'll focus on how to maintain it and make sure it's match ready every round.

16.1 Maintenance

How to keep your swerve drive a clean pristine speedy machine.

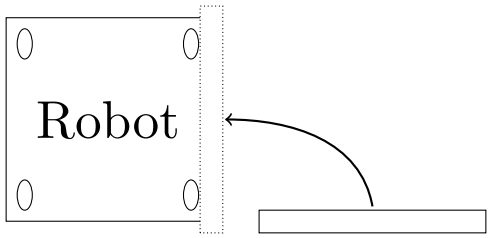
Here's some general tips that don't require too much detail:

- replace the treads whenever they get worn out, but this should be done by the mechanical subteam.
- make sure to check that the encoders are reading sensible values, and if not refer to the section below on how to center your wheels and if they are super wonky values check out how to clean a RoboRio in chapter 5.2.3.

[1] If you have all talons then it may be worth it to check out CTRE's swervedrive tool although don't get your hopes up as you need to be bought into their ecosystem for it to work right

16.1.1 Centering The Wheels

1. open up your logging interface and find the place where the raw position from your encoders is reported
2. find a long straight object at least as long as your robot and put it aside for later
3. make sure all your wheels are facing the same direction such that the bearings are all pointed left or right relative to the front of the robot
4. take that long object and place it against the wheels (as pictured below) and push, this will make it easier to actually straighten the wheels
5. repeat previous step with the other side
6. now take all the raw positions and put them into the offset in your swervedrive code



Chapter 17

Final Notes

Before I say anything else: Near the end of me writing this I became aware of a new RoboRio on the horizon, seeing as this will affect the team in the future FRC's document specifying their timeline is available here.

Thank you for reading my guide, if you haven't actually gone through the whole thing I'd encourage you to, you've got nothing to lose ;).

As I said before in the Introduction (chapter 1.2) if you've got any questions please contact me, if you think anything needs to be added let me know and I will add it. I'm going to keep this brief because I don't have much to say besides good luck, worlds is light work, and as Kevin always says: steal from the best invent the rest.

- Zachary Scheiman 3, January 2025[1]

[1] one day before kickoff :)