Vendor Dependencies

Dependency URLs

These URLs are used by WPIlib to add external libraries to our codebase. They are hard to remember, so here is a list of the ones we use:

NAVX

`https://www.kauailabs.com/dist/frc/2019/navx_frc.json`

CTRE

`http://devsite.ctr-electronics.com/maven/release/com/ctre/phoenix/Phoenix-latest.json`

REV Robotics

`http://www.revrobotics.com/content/sw/max/sdk/REVRobotics.json`

PathFinder

`https://dev.imjac.in/maven/jaci/pathfinder/PathfinderOLD-latest.json`— layout: default title: "External Documentation" nav_order: 6 permalink: /docs/external authors: ['ewpratten'] —

External Documentation

Here is a list of other useful documentation hosted by other people on other websites. - CTRE Doxygen - CTRE Documentation - CTRE Example Code - WPIlib Doxygen (C++) - WPIlib Javadoc (Java) - Pathfinder v1 Documentation - List of "Blinky Things" - PID Tuning - Installing WPILib - Interesting projects from other teams - The history of FRC control systems

Inspirational code

Variable DriveTrain sensitivity based on elevator height

Team: 254

Description: If the elevator is currently sitting at a position above the defined threshold, rotation speed will be decreased.

Code Link: https://github.com/Team254/FRC-2018-Public (line 295)— layout: default title: “FAQ” nav_order: 14 permalink: /docs/faq authors: [‘ewpratten’] —

Frequently Asked Questions

Why do I get build errors from the Compressor subsystem?

This is due to a namespacing issue. We generally try to use other names that mean the same thing. In the past, the names: `cCompressor`, `Pressurizer`, and `Pneumatics` were used to fix this problem. Not enough information has been gathered yet to file a bug report upstream with the WPIlib team.

Gradle is throwing “class not found” errors

This is a common issue on Linux systems due to versioning issues. Generally, the best solution is to: - Uninstall Gradle from the system - Install the latest Gradle package from their website NOT your package manager - Run `gradle wrapper` in the project folder

Motors are behiving unexpectadly

There is a high chance that this is due to motor safety being enabled. To fix this, call the `.setSafetyEnabled(false)` method on all SpeedController, and DifferentialDrive objects related to, or using that motor.

Robot is spinning in circles when it should be driving forward

If your robot is spinning in circles when it should be driving forward, try invert the right motor.— layout: default title: “Configuring the CAN network” parent: “Guides” permalink: /docs/guides/can-config authors: [‘ewpratten’] —

Configuring the CAN network

This guide is adapted from the Toronto Coding Collective

The CAN bus tools supplied by CTRE can be used to update the CAN Bus device Firmware. This tool is required to update the firmware on the Power Distribution Panel (PDP) and Pneumatics Control Module (PCM) and to update any CAN based Victor or Talon speed controllers.

To install the CAN bus tools:

- Download and install the CTRE Phoenix Framework Installer from the link below. Ignore the fact that it is on the Hero Development Board page, this is the correct tool!
- Connect to the RoboRIO using a USB Cable (do not use an Ethernet connection)
- Start the Phoenix Tuner (not the LifeBoat)!
- Click 'Install Phoenix Library/Diagnostics' button to download the tools to the RoboRio.
- Use the CAN Devices tab to view and update CTRE connected CAN devices.

Download{: .btn }— layout: default title: "CommandBased Programming" parent: "Guides" permalink: /docs/guides/command-base authors: ['ewpratten'] —

CommandBased Programming

We use a modified version of WPIlib's CommandBase code structure. The first half of this guide will explain CommandBase to 1st and 2nd year programmers. With small corrections were we have changed functionality. The second half will explain our changes. This guide is partially based off of a guide from the Toronto Coding Collective.

Objective of the Command Based Robot

WPILib supports a method of writing programs called "Command based programming". Command based programming is a design pattern to help you organize your robot programs.

Some of the characteristics of robot programs that might be different from other desktop programs are:

- Activities happen over time, for example a sequence of steps to shoot a Frisbee or raise an elevator and place a tube on a goal.

- These activities occur concurrently, that is it might be desirable for an elevator, wrist and gripper to all be moving into a pickup position at the same time to increase robot performance.
- It is desirable to test the robot mechanisms and activities each individually to help debug your robot.
- Often the program needs to be augmented with additional autonomous programs at the last minute, perhaps at competitions, so easily extendable code is important.

Command based programming supports all these goals easily to make the robot program much simpler than using some less structured technique.

Structure of the Software Layers

The Command Based Robot has the following structure:

CommandBase structure diagram

Operator Interface

The operator input defines the mapping between the operator physical device and the command layer. When a button is pressed, the robot will take an action. The OI layer allows for quick re-configuration of button actions.

Commands

Commands are the glue in the robot connecting operator inputs to the robots motors and sensors. A command will read the operator control inputs and subsystem sensors to determine the appropriate robot action. Complex robot actions are performed though grouping of commands.

Subsystems

Subsystems are containers for all of the motors, pneumatic actuators and sensors on the robot. The subsystems provide feedback to the command layer, and take inputs from the command layer.

A correction to the diagram above: We do not support default commands. Our custom setup works on a "last in, first out" basis. This means that, in a case of conflicting control from commands, the last command to interact with a subsystem before the next loop will be the one that gets priority.

Commands

Command structure diagram

Commands run over a subsystem or set of subsystems to coordinate specific actions.

Each subsystem has a default command, and that command typically takes user inputs from the Joysticks (or OI layer) and translates those actions into motor movements. For instance, a typical DriveControl command reads Joystick values and sets the left and right motor speeds in a DriveTrain Subsystem.

In general, each command runs until it ends, or until it is interrupted by another command. That means there could be several concurrently active commands running - one for each subsystem on the robot!

All currently active commands in the robot are called at approx 50Hz, or once every 20ms.

Our modifications

Much like a command, our subsystems are designed to run in their own loop. When programming the robot, make sure to use an `frc.lib5k.loops.loopables.LoopableSubsystem` class instead of the standard `edu.wpi.first.wpilibj.command.Subsystem`. The `LoopableSubsystem` will automatically handle it's loops in the background. Any code that reads from a sensor should happen in the `periodicInput` method, and any code that writes to an output should happen in the `periodicOutput` method.

An example `LoopableSubsystem` might look like this:

```
package frc.robot.subsystems;

import frc.lib5k.loops.loopables.LoopableSubsystem;
import frc.lib5k.utils.RobotLogger;


public class MySubsystem extends LoopableSubsystem {
    RobotLogger logger = RobotLogger.getInstance();
    public static MySubsystem m_instance = null;

    public MySubsystem() {
        // Subsystem should be initialized here
    }

    /**
     * This is required for every Subsystem
     */
```

```
    public static MySubsystem getInstance() {
        if (m_instance == null) {
            m_instance = new MySubsystem();
        }

        return m_instance;
    }

    @Override
    public void periodicOutput() {
        // Here should be code that makes components move
    }

    public void periodicInput() {
       // Here should be code that reads from any required sensor(s)
    }

    @Override
    public void outputTelemetry() {
        // Here should be code that pushes telemetry data to SmartDashboard
    }

    @Override
    public void stop() {
        // Here should be code that stops any moving component controlled by this subsystem
    }

    @Override
    public void reset() {
        stop();
        // Here should be code that resets all sensors needed by this subsystem
    }

}
```

To register this `LoopableSubsystem` with the `SubsystemLooper`, the Subsystem instance must be passed to the `register` method of the `SubsystemLooper`.— layout: default title: "Command Groups" parent: "Guides" permalink: /docs/guides/commandgroups authors: ['catarinaburghi','slownie'] —

Command Groups

This guide is based off of the offical WPILib documentation which can be found here.

Once you have created commands to operate the subsystems in your robot, they can be grouped together to get more complex operations. These groups are called Command Groups. An example of a Command Group would be the following:

```
public class GrabBall extends CommandGroup {
    public GrabBall() {
        addSequential(new SetElevatorSetpoint(Elevator.TABLE_HEIGHT));
        addSequential(new SetWristSetPoint(Wrist.PICKUP));
        addSequential(new OpenClaw());
    }
}
```

This is an example of a command group that places a ball on a table. To accomplish this task, the robot has to set the elevator's destination (top of the table), then set it's wrist position, and finally open it's claw and pick up the ball. While this can all be done without the use of command groups, it's much easier and cleaner to create command groups.

When do we use Command Groups

Command Groups can be used in two different ways, for autonomous or to automate sequences for the drivers.

Sequential and Parallel Commands

When you add a command to the command group, it can be executed in two different ways; Sequential and Parallel. Sequential commands wait until they are finished (isFinished method returns true) before running the next command in the group. Parallel commands start running, then immediately schedule in the next command in the group. It is important to know which type to use when building a command group.

How to create a Command Group

The code for creating a Command Group is the following:

```
public class ExampleCommandGroup extends CommandGroup {
    public ExampleCommand() {
        addSequential(new ExampleCommandOne());
        addParallel(new ExampleCommandTwo());
    }
}
```

Guides

These are detailed guides on various topics. These guides come from our mentors, and other teams.— layout: default title: “Installing Development Tools” parent: “Guides” permalink: /docs/guides/installing-tools authors: [‘ewpratten’] —

Installing FRC Development Tools

Due to the fact that our team allows Linux, OSX, and Windows to be used by our programmers, creating one definitive installation guide is not feasible. Instead, here are three separate guides from the WPIlib team for each os we support. Keep in mind that we are a Java team when following instructions.

- Linux
- Windows
- Mac OS— layout: default title: “Developing on Linux” parent: “Guides” permalink: /docs/guides/linux authors: [‘ewpratten’] —

Developing robot code on a Linux host

Despite the fact that the tools provided by Autodesk, National Instruments, and FIRST do not natively support Linux, it is possible (and easier) to work with robots when using a Linux-based OS.

As of the 2019-2020 offseason, 100% of the 5024 programming team is developing with linux. This guide will outline the best practices that we have learned throughout the 2019 season (the first season we allowed the use of linux).

Writing code

If you are only looking to write, build, and deploy code, minimal effort is required. Follow our Installing Development Tools guide.

Organization

It is always recocommended to organize your code, but here are a few tips for linux: - Put all FRC-related GIT repos in `~/frc/` - NEVER use `~/frc<year>` to store your code. This directory is only for use by WPIlib. - Use symlinks to map directories to external drives

Git via SSH

Tired of being asked for your password when pushing to GIT, or want to use 2FA?

GIT over SSH is for you. The need for a password, or Oauth token is replaced by your SSH key.

To set up GIT to use SSH, you first must generate an SSH key.

```
ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
ssh-add ~/.ssh/id_rsa
```

When you're prompted to "Enter a file in which to save the key," press Enter. This accepts the default file location.

Next, print out your public key with:

```
cat ~/.ssh/id_rsa.pub
```

And copy it to your clipboard. Now, go to your SSH and GPG Settings, and add your key.

To test that your key was correctly added, try SSHing into GitHub.com:

```
ssh -T git@github.com
```

You should get the following message:

```
You've successfully authenticated, but GitHub does not provide shell access.
```

You can now clone GIT repos via ssh. Make sure to click the **Use SSH** button when you see the **Clone or Download** menu on GitHub.com.

Circumventing our shop's firewall to use GIT

Due to the fact that we *usually* use our school's student wifi network for development, we are restricted by the school board's firewalls. The firewall does it's job perfectly, but has an unfortunate restriction. Port 22 (The port required by GIT) is blocked by default.

We will not cover the use of VPNs, or other circumvention methods, because we do not support full circumvention of the network. On the other hand, we are unable to properly do our job with the firewall's port restrictions.

We have discovered that the only network traffic allowed through the firewall is TCP data on ports 443 and 80. Conveniently, GitHub listens to SSH connections on port 443 as a backup.

To use GIT, all we need to do is redirect SSH traffic for GitHub.com through port 443. This can be done by adding the following info to `~/.ssh/config`.

```
Host github.com
    hostname ssh.github.com
    port 443
```

That's it. GIT should now work.

Fixing SSH issues with multiple robots

Due to the fact that we have 2-3 operational robots at any given time, we are required to deploy code to all of them.

Ocationally, an issue can occur were, SSH will stop working due to the fact that our robots do not share the same SSH keys.

If you get a big warning message when deploying code to our robots, add the following lines to your `~/.ssh/config` file.

```
Host 10.50.24.2
    StrictHostKeyChecking no

Host frcvision.local
    hostname frcvision.local
    StrictHostKeyChecking no
```

Running DriverStation

DriverStation and the National Instruments' tool suite do not support Linux in any way. To solve this problem, we can use VirtualBox.

Qdriverstation

You may have heard of the Qdriverstation project before. This used to be the go-to solution for controlling robots from Linux, but is no longer actively developed.

If you are really wanting a native DriverStation, you can trick a robot into responding to Qdriverstation by doing the following: - Connect to a robot - Launch Qdriverstation - Set the year to 2016 - Open the status panel - Wait for "Robot Link" to light up - Quickly click "Reboot RIO" - This will send control data to the robot, and cause it to partially connect - 50% of the time, this will allow you to connect to, and enable the robot

As you can see, this is not at all stable, so a virtual machine is needed.

Choosing an OS

Unless your host has a large amount of allocatable resources, Windows 7 is going to be your best bet for speed and reliability.

An installation ISO can be downloaded from the Archive.org website. With this ISO, set up a VM normally, and install windows.

Configuring VirtualBox

Now, we need to configure VirtualBox to allow DriverStation to interact with the network.

Go to your **Machine Settings**, then **Network**, then set the network type to **Bridged Adapter**. This will allow the VM to have raw access to your network card, and communicate with a robot.

What to install

Now, follow the windows section of our Installing Development Tools guide, then finally, our Installing DriverStation guide.

You how have everything you need for FRC development and testing configured and enabled. Have fun!— layout: default title: "Installing DriverStation" parent: "Guides" permalink: /docs/guides/ni-update authors: ['ewpratten'] —

Installing DriverStation

This guide is adapted from the Toronto Coding Collective

The National Instruments Update Suite contains the DriverStation software and some of the tools required to configure the robot hardware. Click the download button below. You will need to create a National Instruments account to download the FRC Update Suite. The activation key can be found in the Programming cabinet at the back of the first shelf but is not required for a trial installation.

WPILib provides complete instructions for installing the National Instruments Update Suite. Key components in the suite are:

- Driver Station - used to connect to and drive the robot
- RoboRio Imaging Tool - used to re-image a RoboRio and assign a team number to the hardware

Download{: .btn }— layout: default title: "OpenMesh Configuration" parent: "Guides" permalink: /docs/guides/openmesh authors: ['ewpratten'] —

Configuring an OpenMesh Router

Also refered to as "Flashing a radio", the routers we use on our robots (models: om5p-an and om5p-ac) must run a custom linux-based image provided by FIRST. This guide is a slightly modified version of 118's pac-bot guide.

Flashing

Be patient when configuring the Open Mesh radios, it takes about 30seconds for them to boot. Watch the lights, they will come on, flash for a little while, then theyall go off and it starts over. Don't start configuring until the network light starts blinking thesecond time.

The Radio Flashing Utility provides many options for various routers. These are the three configurations that we commonly use:

Demonstrations and in the shop

In most cases, you will want the driver's laptop to connect directly to your robot, so the radio on your robot will need to be configured as an access point. Due to the age of our laptops, we always use the 2.4GHZ band for our network.

Screenshot of the Radio configuration Utility

After connecting directly to the radio via ethernet and waiting for the radio to power on,

1. Enter your team number
2. Enter `raiderrobotics` as the password
3. Select `OpenMesh`
4. Choose `2.4GHZ`
5. If this radio is being used at any time throuought the season (January to April) Both `BW Limit` and `Firewall` **MUST** be enabled. If this radio is being configured for offseason use only, disable these options.
6. Click on the Configure button and follow the on screen instructions. (If the radioconfiguration fails, a common fix is to temporarily disable any other network interfaces, such as wifi, that you might have running)

At a competition

At each competition, you will need to use the Radio Configuration Kiosk set-up by the event at pit admin. Try to get this done as early as possible, so the team can proceed with system checks in the pit. When running in this mode, the Radio Configuration Utility will configure each team's radio to act as a bridge with a unique SSID and password that team and competition.

At offseason events

During our offseason events and demonstrations that involve many robots in a small space, using a central router for everyone is a great way to reduce interference, and extend the range of every robot.

Offseason radio configuration

After connecting your laptop directly to the radio and waiting for the radio to power on,

1. Enter your team number
2. From the Tools Menu, select FMS Offseason Mode
3. Enter The ssid of the field router. If the event is using ThriftyField, use `ThriftyField` as the SSID.
4. The password should be the same as the field network, or blank.
5. Select `OpenMesh`
6. Both `BW Limit` and `Firewall` should be disables for optimal latency
7. Click on the Configure button and follow the on screen instructions.— layout: default title: "RoboRIO Configuration" parent: "Guides" permalink: /docs/guides/roborio authors: ['ewpratten'] —

Flashing a RoboRIO

This guide is based off of 118's pac-bot guide.

The RoboRIO is a computer designed for use on robots. It has several Reconfigurable Input andOutput (RIO) ports and several communication ports. The RoboRIO runs a customized real-time Linux operating system. Updates to the operating system are released as image files that can be installed using the RoboRIO Imaging tool (or NI MAX). The imaging tool can also be used to set a team number on the RoboRIO.

The imaging process

To image a RoboRIO, you should connect to the RoboRIO with a USB cable plugged into the RoboRIO USB device (USB-B) port. It is safest to disconnect your laptop from all other networks (including wireless networks) when imaging a RoboRIO. After connected, make sure the RoboRIO is powered on, and start the RoboRIO Imaging tool. It should automatically scan and locate your RoboRIO. If not, you can click on the rescan button. When your RoboRIO is found:

1. Make sure the RoboRIO is selected
2. Enter `5024` as the team number
3. Enable `Console Out`

4. Disable `Disable RT`
5. Enable `Format Target`
6. Select the desired (latest) image
7. Click Reformat to begin the imaging process

A full walkthrough of this process can be found HERE

Common sensors

This page was based off a guide from Team 2605.

Encoders

encoder

- Measures rotation of motor shaft.
- Connects to Talon, which tracks cumulative position and velocity.
- RoboRIO requests and receives encoder data from Talon over CAN bus.

Using Encoders

Encoders track the rotation and speed of the motor. They count upwards continuously as the motor rotates forwards and downwards as it rotates backwards. The Talon can try to approach a target position or velocity using encoders.

Methods for reading from an encoder attached to a TalonSRX: - `.configSelectedFeedbackSensor(FeedbackD` `0, 0)`: Required before using encoders. Tells the talon what type of encoder to check for. - `.getSelectedSensorPosition(0)`: Get the position of the encoder. - `.getSelectedSensorVelocity(0)`: Get the velocity of the encoder, in ticks per 100ms.

The talon.set function has an optional first argument that allows different control modes: - `.set(ControlMode.PercentOutput, speed)`: The default. Drive in voltage mode. Speed is any number between -1 (full speed backwards) and 1 (full speed forwards). - `.set(ControlMode.Position, position)`: move to an encoder position. - `.set(ControlMode.Velocity, speed)`: move at a target speed, in encoder ticks per 100ms. - `.config_kP/I/D/F(P, I, D)`: The PID values control how the talon tries to approach a position or speed. These take experimentation to figure out. Try an I value of 0 and a D value between 3 and 6, and adjust the P value to control how strongly the talon tries to approach a position (team 2605 has used P values anywhere from 0.15 to 30).

navX

- Measures acceleration and rotation of robot (6 axes).
- Connects to RoboRIO via pin header.

Using the navX

navx

The NavX is a device which detects movement and rotation of the robot. An "AHRS" object represents a reference to the NavX.

You will need to import the navX library: `import com.kauailabs.navx.frc.AHRS;`

Create an AHRS object: `AHRS m_gyro = new AHRS(Port.kMXP);`

`m_gyro.getAngle()`: Returns the total rotation of the robot in degrees. For example, if the robot rotates clockwise for 2 full rotations, this will be 720.

Limelight

limelight

- Tracks retroreflective vision targets.
- Includes an LED array for illuminating target and a camera.
- Onboard processor filters out and locates the target. Settings can be configured through a web interface.
- Communicates with RoboRIO over ethernet, using the "NetworkTables" protocol.

{:width="250px"}

{:width="250px"}

{:width="250px"}— layout: default title: "Cubic Deadbands" parent: "Learning" permalink: /docs/learn/deadbands authors: ['ewpratten'] —

Cubic Deadbands

In our robot code, we make use of deadbands to account for the fact that our joysticks drift slightly.

In previous years, we just did a simple cutoff of our values:

```
// Constants
final double deadband = 0.1;
```

```
// Loop
double rotation = oi.getTurn();
rotation = (Math.abs(rotation) > deadband)? 0.0 : rotation;
```

Although this worked, it raised the following problem that was mentioned by mimirgames. This causes a hard cut in the motor input. For example, with a deadband of `0.1`: when the joystick reads `0.09` the output will be `0.0%`, but with a reading of `0.11`, the output will be `0.11%`. While this does not seem too bad, it causes a loss of fine control from the driver's input. They can only go `0.0%` or `0.1%`. Nothing in between.

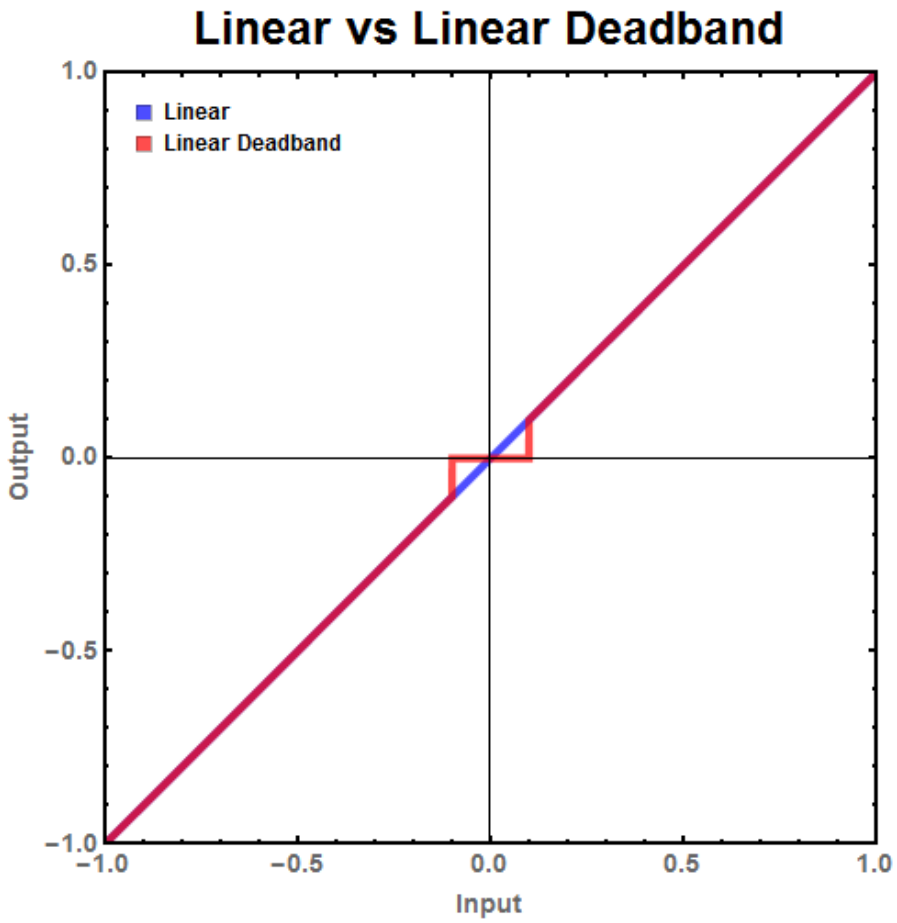

Figure 1: Linear Cutoff graph

The Solution

For most inputs, small changes of small values matter much more than small changes of big values. For example, going 5 mph faster then intended is much more serious when parking a car than when one is driving on the freeway (source: mimirgames).

To achive this, we use this "fancy equation". Essentially, we pass our input through a cubic scaling funciton:

```
w = precision //(0 <= w <= 1)
d = deadband //(0 <= d <= 0.9)
x = joystick_input //(-1 <= x <= 1)

output = ((w * (x ^ 3)  + (1.0 - w) * x) - (abs(x) / x) * (w * (d ^ 3) + (1.0 - w) * d)) / (
```

Visualization

This visualization of our cubic scaling function can be interacted with via the two labled sliders. To view the full Desmos project, click here

Example robot log

We like logs. A lot. Here are some example logs from various situations.

Robot Boot

```
********** Robot program starting **********
Robot starting...
Welcome 5024!
Current time: 0.0346
ROBOT: Starting CameraServer
CS: USB Camera 0: Connecting to USB camera on /dev/video0
CS: USB Camera 0: set format 1 res 320x180
CS: USB Camera 0: Connecting to USB camera on /dev/video0
CS: USB Camera 0: set format 1 res 320x240
CS: USB Camera 0: set FPS to 15
Main Camera initialized on TCP port 1184
ROBOT: Constructing Subsystems
ROBOT: [DriveTrain] Constructing GearBoxes out of motor pairs
NT: server: client CONNECTED: 127.0.0.1 port 36058
ROBOT: [DriveTrain] Limiting current on both gearboxes. Peak: 35A, Hold: 33A, Timeout: 33ms
ROBOT: [DriveTrain] Drivebase has been set to: Unsafe
```

```
ROBOT: [DriveTrain] Loading gyro for drivetrain
ROBOT: [DriveTrain] Gyro has been reset to: 0.0
ROBOT: [Ledring] Constructing "solenoid" for led control
ROBOT: [Slider] Constructing WPI_TalonSRX for slider
ROBOT: [Slider] Constructing Hall effect sensors for slider
ROBOT: [Slider] Configuring PID for slider
ROBOT: [Slider] Set PID gains to: 1.0P, 0.0I, 0.0D
ROBOT: [Finger] Constructing solenoid
ROBOT: [Piston] Constructing solenoid
ROBOT: [Compressor] Constructing Compressor class
ROBOT: [Flap] Constructing solenoid
ROBOT: Initializing Subsystems
ROBOT: Constructing Superstructure
ROBOT: [Vision] Starting monitor thread
ROBOT: Constructing Commands
ROBOT: Setting up Notifiers
ROBOT: FieldStatusThread Starting
ROBOT: [ConnectionMonitor] Started monitoring driverstation connection
ROBOT: Starting vision thread
ROBOT: Setting NetworkTables period time to: 0.01
LIBRARY: Main Camera's connection mode has been set to: Stay Awake
INFO: [DriveTrain] NeutralMode has been set to: Brake
ROBOT: Robot Disabled
Current time: 0.0346
INFO: [DriveTrain] NeutralMode has been set to: Coast
ROBOT: [ConnectionMonitor] The robot has lost connection!
INFO: [Ledring] Wanted state set to: kStrobe
```

CAN errors due to running in simulation mode instead of on a RoboRIO

This is perfectly normal to see. It can be ignored while running a simulation.

```
CTR: CAN frame not received/too-stale.         Talon SRX 1 GetSelectedSensorPosition
CTR: CAN frame not received/too-stale.         Talon SRX 1 GetSelectedSensorPosition
CTR: CAN frame not received/too-stale.         Talon SRX 3 GetSelectedSensorPosition
CTR: CAN frame not received/too-stale.         Talon SRX 3 GetSelectedSensorPosition
CTR: Firm Vers could not be retrieved. Use Phoenix Tuner to check
ID and firmware(CRF) version.          Talon SRX 6
```

— layout: default title: Learning nav_order: 4 has_children: true permalink: /docs/learn —

Learning

This section contains lesson write-ups and some extra material for learning to work with our codebases.— layout: default title: “Robot Localization” parent: “Learning” permalink: /docs/learn/localization authors: [‘ewpratten’] —

[WIP] Robot Localization

Localization is like a GPS for our robots. We use a group of precise sensors to determine the robot’s location withing a few centimeters accuracy.

Available implementations

There are many ways to determine a robot’s location on the field. Some are much easier to accomplish than others. Here are a few common methods:

- Adding a light beacon to the robot, then using computer vision from a camera on the DriverStation to find the robot’s rough location
- 254 used a similar technique to determine the scale’s height in for their 2018 robot, Lockdown
- Using RFID beacons
- Requires support from the venue & field
- Provided by Zebra Technologies for some FRC events
- Using an overhead camera view to detect robot position
- Similar concept to the first option
- Not currently FRC field legal
- Using encoders & gyroscopes to track location history
- Requires precise sensors, and reliable code
- Can run anywhere without needing external sensors (cameras, radios)
- Used by many teams, and well documented

Our implementation

Angle offset—

layout: default title: “Network Tables” parent: “Learning” permalink: /docs/learn/networktables authors: [“wm-c”, “rsninja722”, “catarinaburghi”, “ewpratten”] —

What is A Network Table

A Network Table is a table of values that can be access by multiple devices over a network.

Data in a Network Table is arranged with a key and value

A key can have another table as its value

A Network Table can be used to update a device very quickly after the data has been written

What can it be used for

A Network Table can be used to quickly share live data between different nodes on a network.

It can be used to * Share live data about the Position of the robot * Share the information of sensors and their value * Share a set a data using an indented table

Structure

The data must be stored as boolean with double precision, numeric, or string. You can have arrays of those types of data. There is also the option of storing raw data.

Example

```
import edu.wpi.first.wpilibj.timedrobot;
import edu.wpi.first.networktables.NetworkTable;
import edu.wpi.first.networktables.NetworkTableEntry;
import edu.wpi.first.networktables.NetworkTableInstance;

public class EasyNetworkTableExample extends timedrobot{
    NetworkTableEntry yEntry;
    NetworkTableEntry xEntry;
    public void robotInit(){
        NetworkTableInstance inst = NetworkTableInstance.getDefault();
        Network table = inst.getTable("datatable");
        xEntry = table.getEntry("X")
        yEntry = table.getEntry("Y")
    }

    double x = 0;
```

```
    double y = 0;

    public void teleopPeriodic(){
        xEntry.setDouble(x);
        yEntry.setDouble(y);
        x += 0.05;
        y += 1.0;
    }
}
```

RoboRIO Documentation

RoboRIO

The RoboRIO is an industrial robotics controller built by National Instruments, and contracted by FIRST for use by FRC teams from 2015 to 2021.

Hardware

RoboRIO without cover

The RoboRIO is based on a Xilinx Zynq-7020 SoC. This chip both contains an ARM Cortex-A9 CPU, and an FPGA.

Specifications

The RoboRIO has a dual core ARM Cortex-A9 processor, running with a clock speed of 667mhz. The RIO also has 512 MB of on-board storage, and 256 MB of RAM, clocked at 533mhz.

A full overview of the the RoboRIO I/O and device info can be found on the RoboRIO Specification sheet.

The FPGA

{WIP}

Software

The RoboRIO runs a Linux + Busybox OS with realtime extensions for interaction with the FPGA. This OS also comes with it's own package manager,

OPKG, which can be used to install additional tools onto the RIO (Like GCC, or Python).

The OS, by default, has two notable users. `admin`, the system administrator, and `lvuser`. `lvuser` is the user that runs all software on the RoboRIO. Upon startup, the Bash script located at `/home/lvuser/robotProgram` is executed by the `lvuser` user, and the contents of this script should run the main user program.

By default, the RIO will only run LabVIEW or native programs. Other languages and tools can be installed and configured by the user. To run Java software, tools like GradleRIO will load a custom JVM onto the RIO. Some FRC teams choose to develop with ROS, and can use tools to deploy an ROS environment to their RoboRIO.

Hardware Access Layer

National Instruments provides a Hardware Access Layer (HAL) library for interacting with the FPGA. The HAL contains hooks for the following: - I/O - PWM - DIO - AIO - Relays - Networking - CAN - FRCNetConn - DriverStation - Usage Reporting -

Software libraries

{:width="250px"}

{:width="250px"}— layout: default title: "Robot Bases" parent: "Learning" permalink: /docs/learn/robotbase authors: ['ewpratten'] —

The Various Robot Bases

All good robot programs start with a good base. This is called the RobotBase.

WPIlib provides these three solid bases to choose from when programming your robot: - Iterative - Timed - Command-Based

The Iterative Robot

The IterativeRobot base class has **methods that are periodically called each time new data arrives from the Driver Station**. The idea is that for each mode that the robot is operating in (autonomous, teleop, or test) the appropriate periodic method is called where the program does a small amount of work. **It is important not to have any long running code in the periodic methods such as loops or delays. Doing so could result in missing**

driver station updates that can negatively impact robot performance. Each period is approximately 20 milliseconds by can vary depending on CPU load on the roboRIO, the driver station laptop, or network traffic. If you require precise timing, for example to implement robot control algorithms it is not recommended and you should instead use TimedRobot (below) which has precise timing between periods.

The Timed Robot

TimedRobot is the same as IterativeRobot except that it **uses a timer (Notifier) to guarantee that the periodic methods are called at a predictable time interval**. When getting driver station data such as joystick values the most recent value will be provided since the time interval may not line up with the 20 millisecond delivery of data. **This is the recommended base class for most robot programs**. Just as with IterativeRobot, **it is very important to not have long running code or loops in the periodic methods or the timing may slip**.

The Command-based Robot

While based on the TimedRobot base class, **the command based robot programming style is recommended for most teams**. It makes it easy to break up the program into Commands which each implement some robot behavior such as raising an arm to some position, driving for some distance, etc. It also makes the program easily extensible and testable. The RobotBuilder utility (included with the eclipse plugins) provides an easy way of organizing the program. The dashboards (SmartDashboard and Shuffleboard) allow you to easily debug and test command based programs.

The MagicBot Robot

This final robot base is specifically designed for use with the Python programming language. Due to the fact that this framework only exists in the Python library, we will not cover in our documentation.

If you would like to read more about the MagicBot framework, take a look at the robotpy docs

What Do We Use?

We currently use a modified version of the Command-based framework for all of our code.

We use the Command-based framework because it clearly separates the various functions of the robot, and allows the team to write effective, and clean code.

For anyone curious about the modifications, go talk to the programming team lead. The specific details are somewhat complex and not required knowledge.— layout: default title: “Multithreaded Programming” parent: “Learning” permalink: /docs/learn/threading authors: [‘ewpratten’] —

Multithreaded Programming

Occasionally, it is useful to truly parallelize two pieces of code. For our usecase, this requirement is rare, but sometimes necessary.

Some multithreaded components of our robot code are: - The `robotPeriodic` loop - The `RobotLogger`’s output handling - The `SubsystemLooper` and, by proxy, all periodic methods of any `LoopableSubsystem`

When working on these systems, it is important to understand how to write threaded code for our robots.

Notifiers

WPIlib provides a very useful helper class called a `Notifier`, which allows easy creation and management of time-based threads.

A notifier, once created, will simply call the provided method once every `n` seconds. Just like the periodic loops in the `Robot` class.

Here is an example of a `Notifier`’s use in Java:

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

public class MyThreadedClass{
    Notifier m_notifier;

    public MyThreadedClass(){
        // Configure the notifier to run the doThing method
        m_notifier = new Notifier(this::doThing);

        // Start the notifier to run doThing every 20ms
        m_notifier.startPeriodic(0.02);
    }

    // Method that we want to be run in it's own thread
    private void doThing(){
        // We do something here
```

```
    }
}
```

Note, when specifying the method to run, we need to use `this::` followed by the name of the method. This passes the method as a `Runnable` to the `Notifier`. To stop this thread, we simply need to call `m_notifier.stop()`.

True Threading

Using threads without a wrapper can be very dangerous, but may be needed. Here is an example of a *normal* thread: `java public class Robot extends IterativeRobot {  public void robotInit() {  Thread t = new Thread(() -> {  while (!Thread.interrupted()) {  // Do the thing here  }  });t.start();  } }`— layout: default title: "Your First Program" parent: "Learning" permalink: /docs/learn/your-first-program authors: ['ewpratten', 'exvacuum', 'slownie'] —

Writing your first robot program

This lesson assumes that you have already installed the FRC development tools on your computer, or you are using a computer provided by the team.

Open Visual Studio Code, and follow WPIlib's instructions for starting a new project. Choose 'Template' when selecting a project type. Make sure to set the team number to `5024`, Java as your programming language, and `Command Robot`

We will be writing a simple tank drive with two wheels, one on each side of the robot. Here is a visual: *Team 2605's Tank drive visual*

Preparing the project

WPIlib adds some extra template code to a new project to help teams out. We will not be needing that.

Start by deleting all the files inside of `src/main/java/frc/robot/subsystems`, then the same for `src/main/java/frc/robot/commands`.

Next, replace the contents of `src/main/java/frc/robot/Robot.java` with the following code:

```
package frc.robot;

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

```
public class Robot extends TimedRobot {

  public static OI m_oi;

  @Override
  public void robotInit() {
    m_oi = new OI();

  }

  @Override
  public void teleopInit() {

  }


  @Override
  public void teleopPeriodic() {
    Scheduler.getInstance().run();
  }


}
```

You now have a blank slate of a class. The `Robot` class is automatically called by the robot's hyperviser code running on the RoboRIO. You do not need to worry about how this works, as it is automatic.

What do we want to do?

Deciding what to do with a robot is hard. Luckily, our testing robot has some wheels (as described above). So let's write some code to make our bot move.

Controls

Firstly, we should start off with some controls. Our drive team uses Xbox controllers to drive our robots, so we will be writing some code for one of these controllers. To drive our bot, we will be using just a stingle joystick on our Xbox controller. This is not the configuration we use for competitions, but it is simpler to teach.

Let's add some code to the Operator Interface (OI) file (`src/main/java/frc/robot/OI.java`). This file is where we define every human input.

```
package frc.robot;

import edu.wpi.first.wpilibj.XboxController;
import edu.wpi.first.wpilibj.GenericHID;

public class OI {
    // Note, I have removed all of the comments for this example. Feel free to leave them in

    public XboxController m_driverController = new XboxController(0);

    public double getThrottle(){
        return m_driverController.getY(GenericHID.Hand.kLeft) * -1;
    }

    public double getTurn(){
      return m_driverController.getX(GenericHID.Hand.kLeft);
    }
}
```

Let's start off with out imports.

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

These imports simply tell the JVM that we will be using the `XboxController` and `GenericHID` helper classes from WPIlib.

We need to specify which controller we are reading from (we usually have more than one controller plugged in at a time).

```
public XboxController m_driverController = new XboxController(0);
```

In this case, we are reading from controller `0` (The driver's controller).

Next, we define our `getThrottle` method.

```
public double getThrottle(){
    return m_driverController.getY(GenericHID.Hand.kLeft) * -1;
}
```

This method will read data from the driver's Xbox controller and return it. Notice the `* -1`? That is required when reading from the **Y** axis of a joystick.

Now, we do the same for `getTurn`, except we are reading from the **X** axis this time, and do not need to invert the value.

```
public double getTurn(){
    return m_driverController.getX(GenericHID.Hand.kLeft);
}
```

That is it for reading user inputs. Now, on to controlling motors.

Controlling motors

In order to move the robot's wheels, we need to control their motors. This can be done with a Subsystem.

Create a new file called `DriveTrain.java` in the `src/main/java/frc/robot/subsystems` folder, and add the following template code.

```
package frc.robot.subsystems;

import com.ctre.phoenix.motorcontrol.can.WPI_TalonSRX;

import edu.wpi.first.wpilibj.command.Subsystem;
import edu.wpi.first.wpilibj.drive.DifferentialDrive;

public class DriveTrain extends Subsystem {

    WPI_TalonSRX left;
    WPI_TalonSRX right;
    DifferentialDrive drive;

    public DriveTrain() {

    }

    @Override
    protected void initDefaultCommand() {

    }

}
```

You will also need to add the CTRE vendordep. To do this, open the command palette using ctrl+shift+P and search "vendor", select "Manage Vendor Libraries", then "Install New Library (Online)", and paste the previous link. See the vendor dependencies page for more information. Feel free to ignore this step until you want to test the code. At that point, ask a returning team member for help.

The code you have added is just a template, and follows the same style as the Operator Interface. Next, we need to add two motors and link them together. Add the following to the `DriveTrain` method:

```
left = new WPI_TalonSRX(1);
right = new WPI_TalonSRX(3);

drive = new DifferentialDrive(left, right);
drive.setSafetyEnabled(false);
```

The first two lines set up two motors, with IDs `1` and `3`. These numbers are used by the RoboRIO to identify which motor to control. The last two lines link the two motors together into a wrapper class that we have assigned to the variable `drive`.

Now, we simply need a method to turn joystick data into motor commands:

```
public void arcadeDrive(double speed, double rotation) {
    drive.arcadeDrive(speed, rotation);
}
```

This passes `speed` and `rotation` to `drive`'s `arcadeDrive` method.

Feeding joystick data to the DriveTrain

We now need to read the joystick data, and push it to the DriveTrain. This can be done with a Command.

Create a new file called `DriveControl.java` in the `src/main/java/frc/robot/commands` folder, and add the following template code.

```
package frc.robot.commands;

import edu.wpi.first.wpilibj.command.Command;
import frc.robot.Robot;

public class DriveControl extends Command {

    @Override
    protected void execute() {

    }

    @Override
    protected boolean isFinished() {
        return false;
    }

}
```

This is a basic command class. As long as `isFinished` returns `false`, the `execute` method will be called once every 20ms by the robot. We can use this to our advantage by writing some code to read our joystick data here.

Inside the `execute` method, add these two lines. They will call the `getThrottle` and `getTurn` methods we made in the first section, and store the data.

```
double speed = Robot.m_oi.getThrottle();
double rotation = Robot.m_oi.getTurn();
```

Now, we just need to pass this data to the subsystem we just made. This can be done with the following line. It will call the `arcadeDrive` method we just made, and give it our speed and rotation valuse.

```
Robot.m_driveTrain.arcadeDrive(speed, rotation);
```

Adding the subsystem to the robot

That's almost everything! We just need to write a few lines of code to tie everything together.

In the `src/main/java/frc/robot/Robot.java` file, we need to add the following to the list of imports in order to import our new Command and Subsystem:

```
import frc.robot.commands.DriveControl;
import frc.robot.subsystems.DriveTrain;
```

Next, we need to tell the robot that we made a subsystem and command. Add the following under the line that says `public static OI m_oi;`:

```
public static DriveTrain m_driveTrain;

DriveControl m_driveControl;
```

Next, we need to create the two objects

```
m_driveControl = new DriveControl();
m_driveTrain = new DriveTrain();
```

These go under the `m_oi = new OI();` line.

Finally, add the following code to the `teleopInit()` method we created earlier in the `Robot` class:

```
if (m_driveControl != null){
    m_driveControl.start();
}
```

This code checks that `m_driveControl` has not started, and then starts it. Without this, the code we have written will not work.

Conclusion

That's it! You now have your first piece of robot code (and it can actually drive!).

This guide was not designed to teach you how to program, but to show you around the general file structure of our robot code, and where each type of code goes. Feel free to come talk to a returning mentor, or team lead to see what you can add to your code next.— layout: default title: "AutoCamera" parent:

"Lib5K Reference" permalink: /docs/lib5k/autocamera authors: ['ewpratten']
—

AutoCamera

AutoCamera is a wrapper for USB camera devices plugged directly into the RoboRIO that will handle configuration, management, and publishing automatically.

Connecting to a camera

To connect a USB camera to an AutoCamera object, the camera must be plugged in to one of the RoboRIO's two USB ports, or a USB splitter.

```
// Create an AutoCamera with only 1 webcam connected to the RoboRIO
// AutoCamera will ask WPIlib to find the device. If more than one camera is plugged in, the
AutoCamera myCamera = new AutoCamera();

// Create an AutoCamera with a specific USB port number (these are actually unix video devic
AutoCamera myCamera = new AutoCamera(0);

// Create an AutoCamera with a specific USB port and name
AutoCamera myCamera = new AutoCamera("Fancy Camera", 0);
```

Calling any of these will connect to the specified camera, configure the video feed to *240x320px* at *15fps*, and display the feed on Shuffleboard.

Configuring the video feed

240x320px at *15fps* may not be your desired video stream parameters. These can be changed via the `setResolution` method.

```
// Configure the camera to run at 800x600px and 30fps
myCamera.setResolution(600,800,30);
```

Loading configuration files

Camera configuration files can be generated via the web interface (see below). These files contain static values for each camera parameter (we use this to configure our cameras for each venue). To configure a camera with a file, use the `loadJsonConfig` method.

```
// Load a camera config stored at /home/lvuser/deploy/myCamera.json
myCamera.loadJsonConfig("/home/lvuser/deploy/myCamera.json");
```

Keeping the camera feed alive

When a camera feed is not being watched, it will shut down to save bandwidth. A downside is that, when viewed after shut down, the feed will take a few seconds to start up again. To prevent this, we can tell the camera to continue streaming when it's not being watched.

```
// Tell the camera to keep streaming
myCamera.keepCameraAwake(true);
```

(From java package `lib5k.kinematics`)

Drivebase Kinematics

"Kinematics is a branch of classical mechanics that describes the motion of points, bodies, and systems of bodies without considering the forces that cause them to move" -Wikipedia

Kinematics is very helpful for controlling, and defining robot movement during the autonomous period, along with various other in-game actions. This document will outline how we have integrated both forward, and inverse kinematics into Lib5K, and how to use these features in your robot program.

Robot Localization

Robot localization, a form of forward kinematics, is the process of locating a robot in 2D or 3D space using a variety of sensors. Think of it like a GPS.

Localization with an FRC robot

Assuming a kitbot-style drivetrain, we can use three simple sensors, and some trigonometry to determine the robot's location on the field with reasonable precision.

We need to know three things about the robot. - Total distance traveled - Current heading - Circumference of the wheels in meters

With this information, we can get an X, and Y coordinate of the robot in meters with the following code:

```
// For storing the robot position
double x,y = 0.0;
double lastPos = 0.0;

// Calculate the circumference in meters of the wheels
// This assumes 6.0 inch wheels on the drivebase
final double wheelCirc = (6.0 * 0.0254) * Math.PI;

// Number of encoder ticks produced per wheel revolution
final int ticksPerRev = 720;

/**
 * Will convert an encoder tick count to a distance traveled in meters
 */
double ticksToMeters(int tickCount){
    return (tickCount / ticksPerRev) * wheelCirc;
}

/**
 * This should be called once per 20ms (or the robot periodic loop)
 */
void loop(){
    // Read the robot's current angle from the gyroscope and bind it by 360 degrees
    double heading = getAngle() % 360;

    // Find the average distance traveled between each side of the robot. This
    // will give the total Y distance traveled by the robot, accounting for rotation
    double leftMeters = ticksToMeters(getLeftEncoderTicks());
    double rightMeters = ticksToMeters(getRightEncoderTicks());

    double position = (leftMeters + rightMeters) / 2.0;

    // Find the distance traveled by the robot since the last time loop() was called
    double distance = position - lastPos;

    // Calculate position
    x += (distance * Math.cos(Math.toRadians(heading)));
    y += (distance * Math.sin(Math.toRadians(heading)));

    // re-set the last position
    lastPos = position;
}
```

This example assumes that the robot's heading / angle can be read with `getAngle()`, and each encoder has a method to read its tick count. If using

the tools built in to Lib5K, all of this is handled for you.

How it works

Finding the robot location is relatively simple with the help of some trigonometry.

Field diagram with triangle to show how location is calculated

This diagram roughly demonstrates how location is calculated. In practice, calling the `loop()` method in a loop would generate a very large number of triangles, and provide much greater precision.

The diagram shows the calculation that would be completed if the robot started on HAB1 (at the red angle), and moved to the left rocket, then called `loop()` once. The robot's heading would be treated as *theta* (the red angle), and it's total distance traveled since the last loop as the *hypotenuse*. The labeled `X` and `Y` axis of the right-angled triangle are now the robot's `X,Y` location on the field.

Lib5K LocalizationEngine

Lib5K provides a tool called the `LocalizationEngine`. This tool wil automatically complete the localization calculations, and convert the data to a `FieldPosition` object for easy use with other Lib5K tools and components, like the `MovementPlanner`.

Using it in your code

The `LocalizationEngine` requires some other components to work. The below example will assume you are using a robot similar to MiniBot.

To use the `LocalizationEngine`, the same three sensors from the example above are required. Here, we will define our motors, their encoders, and a Gyroscope object. This code should be part of the DriveTrain subsystem.

```
// Create two gearboxes with encoders attached to the rear motor controllers
GearBox leftGearbox = new GearBox(new WPI_TalonSRX(1), new WPI_TalonSRX(2), true);
GearBox rightGearbox = new GearBox(new WPI_TalonSRX31), new WPI_TalonSRX(4), true);

// Create two encoders from each gearbox
EncoderBase leftEncoder = new GeaBoxEncoder(leftGearbox);
EncoderBase rightEncoder = new GeaBoxEncoder(rightGearbox);

// Create a gyroscope (this example wil use a NavX)
AHRS gyro = new AHRS(Port.kMXP);
```

We additionally will need to get a `LocalizationEngine` instance to work with

```java
// Get the current LocalizationEngine instance
LocalizationEngine le_instance = LocalizationEngine.getInstance();
```

The codebase should have these values defined elsewhere (e.g. in the `Constants.java` file), but for this example, we will define a few parameters of the robot here:

```java
// Calculate the circumference in meters of the wheels
// This assumes 6.0 inch wheels on the drivebase
final double wheelCirc = (6.0 * 0.0254) * Math.PI;

// Number of encoder ticks produced per wheel revolution
final int ticksPerRev = 720;
```

Next, in the subsystem's `periodic()` method, or any other constantly looping method, we can update the `LocalizationEngine`

```java
@Override
public void periodic(){
    // Get the robot heading
    double heading = gyro.getAngle();

    // Get each encoder's distance reading
    double leftMeters = leftEncoder.getMeters(ticksPerRev, wheelCirc);
    double rightMeters = rightEncoder.getMeters(ticksPerRev, wheelCirc);

    // Update the LocalizationEngine
    le_instance.calculate(leftMeters, rightMeters, heading);
}
```

The robot's position will now be constantly updated. At the start of autonomous, you may want to reset the robot's location.

```java
// Set the robot's field-relitive location in meters.
// This example uses an X of 0, and a Y of 5 meters.
// This should be called at the start of each autonomous
// commandgroup to set the location appropriately.
le_instance.setRobotPosition(new FieldPosition(0.0, 5.0));
```

Finally, to read the robot's current location (and heading), the following can be done: `java FieldPosition currentPosition = le_instance.getRobotPosition();`— layout: default title: "GearBox" parent: "Lib5K Reference" permalink: /docs/lib5k/gearbox authors: ['ewpratten'] —

GearBox

A GearBox is a wrapper for any collection of motor controllers. Currently, the only supported controller is the CTRE TalonSRX.

Creating a GearBox

To create a GearBox, pass in a pair of `WPI_TalonSRX` objects, and a `boolean` to specify if the sensor is attached to the rear (second) motor or not.

```
// Create a GearBox with talons 1 & 2, and an encoder attached to talon 1
GearBox myGearbox = new GearBox(new WPI_TalonSRX(1), new WPI_TalonSRX(2), false);

// Create a GearBox with talons 1 & 2, and an encoder attached to talon 2
GearBox myGearbox = new GearBox(new WPI_TalonSRX(1), new WPI_TalonSRX(2), true);
```

This will create, and connect to both motor controllers, automatically configure them, and pair them together to follow eachother.

Configuring current limiting

Limiting the maximum output current of a motor is very useful. It allows systems to operate together without browning out the robot. To configure current limiting on a GearBox, use the `limitCurrent` method.

```
// Configure the current limits with a trigger of 32Amps, a hold at 30Amps, and a duration
myGreabox.limitCurrent(32, 30, 15);
```

The peak / trigger current is the number of Amperes the system must draw before the limiter takes effect.

The current hold is the maximum number of Amperes the system can draw once the peak / trigger has been tripped.

The duration is the amount of time the system's current draw must exceed the peak (in ms) before the limit takes effect.

Directly controlling

To directly control the GearBox, use the `set` method. This will set the motor output percentage.

```
// Output full speed forwards
myGearbox.set(1.0);
```

```
// Output full speed beckwards
myGearbox.set(-1.0);
```

Using with `DifferentialDrive`

Our robot's drivetrains usually have at least 2 motors per side. This allows the use of a GearBox to "package" each motor group together, and pass to WPIlib's DifferentialDrive controller. To build a DifferentialDrive from two gearboxes (with two motors each), the following can be done:

```
// Create a GearBox for each side of the robot
GearBox leftGearbox = new GearBox(new WPI_TalonSRX(1), new WPI_TalonSRX(2), false);
GearBox rightGearbox = new GearBox(new WPI_TalonSRX(3), new WPI_TalonSRX(4), false);

// Create the DifferentialDrive object
DifferentialDrive drive = new DifferentialDrive(leftGearbox.getMaster(), rightGearbox.getMas
```

All regular DifferentialDrive functionality will now work with these motors.

Reading sensors

Each GearBox will allow an encoder to be attached to one of the talons. (make sure to specify which one with the boolean in the GearBox constructor).

To read the number of ticks from a GearBox's encoder, use it's `getTicks` method.

```
// Read the tick count of a GearBox encoder
int ticks = myGearbox.getTicks();
```

The GearBoxEncoder

Lib5K also includes an EncoderBase. This is a single class that can adapt any type of encoder into a unified interface. To create one from a GearBox, the `GearBoxEncoder` wrapper can be used.

```
// Create an EncoderBase from a GearBox
EncoderBase myEncoder = new GearBoxEncoder(myGearBox);
```

Now, additional data can be read from the encoder.

```
// Get the GearBox output speed in Ticks per second
double speed = myEncoder.getSpeed();

// Get the distance traveled in meters (this requires information about the GearBox's outpu
// This example will use the parameters of one of MiniBot's wheels
double distance = myEncoder.getMeters(720, (6.0*2.54) * Math.PI);
```

Don't forget to call the EncoderBase's `update` method from inside the DriveTrain's Subsystem!— layout: default title: Lib5K Reference nav_order: 4 has_children: true permalink: /docs/lib5k authors: ['ewpratten'] —

Lib5K Reference

This section contains the documentation for Lib5K, our team's library for all of our robots.

Why Lib5K?

Lib5K was designed by @ewpratten over the summer of 2019 to provide a unified collection of wrappers, utilities, and control systems for 5024.

Updates and adding to a project

The Lib5K repo is not actively updated. It is just a snapshot of the library. Every month or so, the current Lib5K lib should be copied into the repo. To add it to a project, just drop the library into `src/main/java/frc/`.— layout: default title: "PIDProfile" parent: "Lib5K Reference" permalink: /docs/lib5k/pidprofile authors: ['ewpratten'] —

PIDProfile

A PIDProfile is a data structure for storing PID Gains. This allows us to swap between profiles depending on the conditions (for example, we could have a profile for concrete, and a profile for carpet).

Creating a profile

To create a profile, just pass in the gains.

```
// PID
new PIDProfile(p,i,d);

// PI
new PIDProfile(p,i);

// P
new PIDProfile(p);
```

These profiles can be passed into the constructor of a PID object.

```
new PID(new PIDProfile(p,i,d));
```

Profile auto-generation

PIDProfile can also automatically generate a profile based on the Zeigler-Nichols tuning method.

```
// Auto-generated profile with a maximum P of 5, and an oscillation period of 1 second

PIDProfile profile = PIDProfile.autoConfig(5.0,1.0);
```

This will generate the `p`, `i`, and `d` values.

Profile modification

You may want to auto-generate a profile, but make a slight change to a value. This is where `PIDProfile.modify` comes in to play.

```
java // Same generated profile from above, but we are going to
increase the P gain by 1 PIDProfile profile = PIDProfile.autoConfig(5.0,1.0).modify(new
PIDProfile(1,0.0,0.0));— layout: default title: “RobotLogger” parent:
“Lib5K Reference” permalink: /docs/lib5k/robotlogger authors: [‘ewpratten’]
—
```

RobotLogger

The RobotLogger is a threaded logging system that reduces stress on the network logging system (NetConsole), and reduces console spam.

Logging

To log data with the RobotLogger, You can do the following in any class

```
// Get a logger instance
RobotLogger logger = RobotLogger.getInstance();

// Log some standard text
logger.log("Some text");

// Log a warning
logger.log("This is a warning", Level.kWarning);
```

```
// Log data before the scheduler starts
logger.log("This will be immediately pushed to the console", Level.kRobot);
```

Stating the logger

The RobotLogger must be started when the robot is initialized.

```
class Robot{
    RobotLogger logger = RobotLogger.getInstance();

    void robotInit(){
        logger.start(0.02);
    }
}
```

`logger.start()` must be passed a loop period. This should be the same as the robot period (by default `0.02`).

How it works

The RobotLogger uses an `ArrayList` to buffer any message to be logged, then upon a call to `update` (usually by the FRC Notifier), it will print every message from the list and clear it.

Any message logged at the `Level.kRobot` level, will be immediately pushed to the console.

DriverStation

This document outlines various features of DriverStation and its networking protocol

Using DriverStation

- Viewing Laptop-side Logs
- Using the logs to prove that a brownout happened

Comms Protocol

- DS -> RIO
- RIO -> DS
- DS -> FMS

- FMS -> DS

The "Secret DS protocol"

There is a protocol built into the driverstation that is used by gradleRIO and NetowrkTables to find the robot's ip address. To read this data, open a TCP connection to `localhost` on port `1742` on a computer running driverstation. A JSON string will be returned containing the magical data!

This was found in the NetworkTables Native library

layout: default title: "Events API" parent: "Low Level" permalink: /docs/lowlevel/eventsapi authors: ['ewpratten'] —

FMS Event API

The event api allows pulling of match data almost immeadiatly after a match ends.

Documentation

The endpoint documentation can be found HERE. **NOTE:** This api requires an API key and login.

Obtaining an API key

To get your api key, head over to the new request site https://frc-events.firstinspires.org/services/API and request a token. Make sure to list 5024 as your frc team.

Your API key for HTTP access is created by adding your username to your key with a colon in the middle, then base64 encoding it. For example, if your username is `sampleuser` and your key is `7eaa6338-a097-4221-ac04-b6120fcc4d49` you would have this string:

`sampleuser:7eaa6338-a097-4221-ac04-b6120fcc4d49`

Bae64 encoding this would give:

`c2FtcGxldXNlcjo3ZWFhNjMzOC1hMDk3LTQyMjEtYWMwNC1iNjEyMGZjYzRkNDk=`

Wrapper APIs

While nobody talks about the event api, you have seen it used before under a different name. **The Blue Alliance API** is actually just a wrapper around the Events API. The only difference is that The Blue Alliance uses a Docker caching server (This is why the TBA app is always slightly out of date during an event).

I ([@ewpratten](https://github.com/ewpratten)) also have my own wrapper at api.retrylife.ca that allows for limited polling of the API.

layout: default title: “Field Management System” parent: “Low Level” permalink: /docs/lowlevel/fms authors: [‘ewpratten’] —

Field Management System

The following is a list of resources for learning about the Field Management System - Whitepaper - Documentation - 2014+ FMS video - Match logs - Robot status display - Android app to control field from a tablet

Notes

These are other random notes found by looking through source code and reverse-engineering FMS and it’s hardware - The main FMS computer always has the ip address `10.0.100.5` and listens for udp connections from a driverstation on each alliance station — layout: default title: “Low Level” nav_order: 4 has_children: true permalink: /docs/lowlevel authors: [‘ewpratten’] —

Low Level Systems Documentation

This section is not required knowledge, but a place for team members to keep notes about obscure systems, networking protocols, and components.— layout: default title: “RoboRIO Hardware” parent: “Low Level” permalink: /docs/lowlevel/roborio authors: [‘ewpratten’] —

RoboRIO Hardware

The RoboRIO is a black (grey) box that does magical things. We put code in, it makes things move. These documents and notes describe low-level details about the RoboRIO and how it works.

- User manual
- FPGA (Field Programmable Gate Array) and HAL(Hardware Access Layer)
- Simplified Manual
- Guide for accessing the HAL without WPIlib
-

(very) Low level FPGA Info

layout: default title: "Meta" nav_order: 12 permalink: /docs/meta authors: ['ewpratten'] —

The webdocs' docs

This page contains the documentation for keeping our documentation updated.

Updating webdocs

Webdocs is built out of a collection of text files hosted on GitHub. To update this site, clone and edit the files in the `docs/` folder. Once your addition is ready for publishing, push to the master branch, and the site will automatically update after a few minutes.

Formatting

All documents are written in a markup language called Markdown. Markdown is easy to learn, and a great guide is available to help you learn the syntax.

For members looking to work with some extra features, you should know that we do not use standard Markdown, but instead, Kramdown (it has the same syntax) with a bunch of plugins.

File headers

For easy development and parsing, webdocs uses the liquid parser. This allows for the use of variables and other bits of code to be written directly into our documents. The way out site is set up, every file must start with a liquid header. Note, this must literally be the first thing in the file, line 1 cannot be blank.

Here is a standard header for this page

```
---
layout: default
title: "Meta"
nav_order: 12
permalink: /docs/meta
authors: ['ewpratten']
---
```

The start and end of a header are defined by three dashes.

The layout must be `default`. Title is what shows in the nvigation bar, and permalink is the url for this page. Change these to match the page you are writing. The `nav_order` variable is the order this item appears in the navigation bar. Generally, put new pages at the bottom of the list.

To create a folder in the nav bar, create a folder inside of the `docs/` folder, and make a file with the same name inside that folder. This file acts at the table of contents for that folder, and must have an extra line in it's header:

```
---
# <The rest of the header goes here>
has_children: true
---
```

To add children to this sub-section, add new files to it's folder, and give them this extra attribute:

```
---
# <The rest of the header goes here>
parent: "<Title of the index file>"
---
```

The parent must exactly match the title of the index (table of contents) file for that folder.

The contributors list

Every file also has an `authors` tag in it's header. This is used to keep track of who has worked on each file. To add yourself as an author to the page, add your GitHub username (all lowercase) to the list.

```
---
# Example of only one author
authors: ['slownie']

# Example of two authors
authors: ['retrax24', 'ewpratten']
---
```

This list will be displayed at the bottom of the page.

Adding new members to the config

The way our contributors list is set up allows for automatically generated profile pages for each member (currently disabled). These pages include the member's name, some information about what they do, and their github account details. This is all configured through the `_data/members.yml` file.

Here is an example entry for one of our graduated members, Matthew Eppel.

```
faceincake:
  name: Matthew Eppel
  is_lead: false
  is_mentor: false
  github: faceincake
```

These variables should be pretty self-explanatory. The key must be the same as the value of `github`.

To add a new member to the website, add them to this list.

How webdocs is hosted

Webdocs is hosted on GitHub pages and automatically updated by a CD pipeline. This is all automatically configured.

Webdocs' backend

The backend of webdocs is Jekyll. This tool turns some markdown and html into a nice website, automatically. For anyone looking to make majour changes to this site, or the programming homepage, you will need to know how to use Jekyll. Luckily, the Jekyll Docs have everything you need to know for this job.

The site's theme

We are using a modified version of Just the Docs. Their website has some useful information on how to use their theme-specific tools.

The site's plugins

We have a few plugins installed for this website. They are:

- jekyll-feed
- jekyll-redirect-from
- jemoji
- jekyll-mentions
- jekyll-seo-tag

Jekyll-feed automatically generates an RSS feed for this website.

Jekyll-redirect-from allows us to set up link shorteners and redirects.

jemoji enables slack-like emoji. :tada: (That was done by adding `:tada:` to the file)

Jekyll-mentions allows us to automatically link to any github account. @frc5024 (That was done by adding `@frc5024` to the file)

Jekyll-seo-tag automatically handles Search Engine Optimization for this site.

The difference between homepage and webdocs

The programming team has two different websites.

The first is our homepage. This is a separate website, also hosted with jekyll, that can be found at the frc5024.github.io GitHub repo.

Second, is webdocs, which is hosted in the webdocs repo.

For instructions on updating the homepage, take a look at it's repo's README.md file.

Running a development server locally

To run a development version of this site locally, - Install Ruby - Install Jekyll with: `gem install jekyll bundler` - Then run `bundle exec jekyll serve` in the root of this project - The site is now live at localhost:4000 — layout: default title: "Team Reference" nav_order: 3 has_children: true permalink: /docs/reference authors: ['ewpratten'] —

Team Reference

Due to the fact the our team manual only releases yearly, this section of the webdocs contains a copy of many pages from the programming manual. This copy is constantly updated, and is generally the first place a programmer should look for information on team rules and guides.

All additions and edits to this section will be taken into consideration when building the next year's manual.— layout: default title: "Version Control" parent: "Team Reference" permalink: /docs/reference/vcs authors: ['ewpratten', 'slownie'] —

Version Control for 5024

GitKracken

GitKraken is a GUI that streamlines the Git workflow. It doesn't require Git Bash but you should have that installed on your computer anyway. To use GitKraken, download it from GitKracken.com. When it has finished installation, log in with your GitHub Account.

However, before you use the application, you should learn your way around the interface. This can be done by reading the GitKracken quick-start Guide.

Learning the Basics

These are the basic actions that every programmer must know in order to use GitKracken:

- Clone a Repo
- Make a Branch
- Stage, Commit, and Push Code

The GitKracken team has also provided a great guide for learning Git. It is avalible at the link below: https://support.gitkraken.com/start-here/guide

Using the Command Line Interface

Git Bash is the Windows command-line version of Git. It works exactly the same as regular Git, but instead of using cmd, the stock windows terminal, commands are entered into the Git Bash terminal. Git Bash can be downloaded from the Git website.

Git has many commands for it's various tasks. The commands most used by out team are the following:

```
git clone https://example.com/project.git
git add .
git commit -m "This is a commit message"
git push
git checkout branch-name
```

A very useful cheat-sheet is avalible on the git-tower website.

Proper Branch Usage

Before you make a new branch, make sure you have the newest code on your computer. To do this, run the `git pull` command or press the **pull** button in GitKraken.

Never push to the master branch without a code review. The master branch should always contain fully operation and review code that has been approved by a mentor, unless a mentor specifically asks you to push some non-reviewed code.

Do NOT do work in the master branch on your computer. If you need to change code, do it in a separate branch. This makes it easier if you need to copy your code to another computer and will make sure that you don't get errors while pushing code.

When pushing code to the robot, always push the master branch, unless you are testing something on your own branch. At the end of the day/meeting, make sure the robot has the newest code from master pushed to it in case another subteam needs to use the robot.

Further Learning

To learn more about GIT, including setting up a GitHub account, and working with a team, take a look at WPIlib's GIT Documentation.— layout: default title: "Benson" parent: "Our Robots" permalink: /docs/robots/benson authors: ['ewpratten'] nav_order: -1 —

Benson

Benson was our robot for the 2014 season. 2014's game was Aerial Assist.

- Code

Robot Photo

robot

{:width="250px"}— layout: default title: "Robot codenames" parent: "Our Robots" permalink: /docs/robots/codenames authors: ['ewpratten'] nav_order: 0 —

Codenames

In the time between kickoff and revealing the official name of our robot, the programming team uses codenames to refer to each bot. It has become tradition to name a robot after each graduating member of the season. These are the codenames we have used in the past:

Codename	Robot
FraserIsTheBest	Q*bert
MattIsBetter	HATCHfield
ParsaIsPreferable	2019 Practice bot
SamIsSuperior	MiniBot
DarianIsDaring	MiniBot 2.0
CarterIsChaotic	???

Guinevere

Guinevere was our robot for the 2016 season. 2016's game was Stronghold.

- Code

Robot Photo

robot

{:width="250px"}— layout: default title: "HATCHfield" parent: "Our Robots" permalink: /docs/robots/hatchfield authors: ['ewpratten'] nav_order: -7 —

HATCHfield

HATCHfield was our robot for the 2019 season. 2019's game was Deep Space.

- Code

- Offseason Code
- Documentation
- Offseason Documentation
- Devices Info

Robot Photo

robot

{:width="250px"}— layout: default title: "Herbert" parent: "Our Robots" permalink: /docs/robots/herbert authors: ['ewpratten'] nav_order: -2 —

Herbert

Herbert was our robot for the 2015 season. 2015's game was Recycle Rush.

- Code

Robot Photo

robot

{:width="250px"}— layout: default title: "MiniBot" parent: "Our Robots" permalink: /docs/robots/minibot authors: ['ewpratten'] nav_order: -6 —

MiniBot

MiniBot is our prototyping robot. It is also used for teaching programming to new members.

- 2018 Code
- 2019 Code
- 2018 Documentation
- 2019 Documentation
- Devices Info

Mesurements

The encoders have `360` ticks per revolution.

Object	Mesurement (cm)
drivebase width (wheel to wheel)	50.8 cm

Object	Mesurement (cm)
Length (back to front)	61.595 cm
Width (left to right)	59.69 cm
Wheel diameter	15.24 cm
Wheel spacing (from axel to axel)	19.685 cm
DriveTrain height	8.255 cm
DriveTrain cutout depth	22.86 cm
DriveTrain height from ground	5.715 cm
Cage height from drivetrain	30.48 cm
Cage length (back to front)	58.42 cm
Cage width (left to right)	50.8 cm
Gyro position offset (left to right)	10.16 cm to left of centre
Gyro position offset (front to back)	17.78 cm to the rear of robot

Notes

On the roborio used on this bot, there are a few broken IO pins: - Digital 2 - Digital 3— layout: default title: “Q*bert” parent: “Our Robots” permalink: /docs/robots/qbert authors: [‘ewpratten’] nav_order: -5 —

Q*Bert* QBert was our robot for the 2018 season.

2018’s game was Power Up.

- Code
- Offseason Code
- 2019 Beta Test Code
- Offseason Computer Vision Demo Code
- Offseason Documentation

Robot Photo

robot

{:width=“250px”}— layout: default title: “Our Robots” nav_order: 1 has_children: true permalink: /docs/robots authors: [‘ewpratten’] —

Our Robots

These pages contain information specific to each of our robots.

layout: default title: “Watt” parent: “Our Robots” permalink: /docs/robots/watt authors: [‘ewpratten’] nav_order: -4 —

Watt

Watt was our robot for the 2017 season. 2017’s game was Steamworks.

- Code

Robot Photo

robot

{:width=“250px”}— layout: default title: “Team Projects” nav_order: 13 permalink: /docs/projects authors: [‘ewpratten’] —

Team Projects

Here is a list of open-source projects developed by 5024

PocketLogger

PocketLogger is a web-based tool for recording and viewing match logs during events. This tool is open source, and available for use by any team.

Launch Tool{: .btn }— layout: default title: “Whitepapers” nav_order: 9 permalink: /docs/whitepapers authors: [‘ewpratten’] —

Whitepapers

Here is a list of whitepapers that are recommended to read if you want to better understand a topic

- PID Control Theory (Team 358)— layout: default title: “Who Owns What?” nav_order: 7 permalink: /docs/who-owns-what authors: [‘ewpratten’] —

Who owns what?

This is a slide taken from the 2018 WPIlib presentation at Worlds.

Who owns what slide— layout: default title: "WPIlib reference" nav_order: 8 permalink: /docs/wpilib authors: ['ewpratten'] —

WPIlib Reference

This guide is adapted from the Toronto Coding Collective

What is WPILib

From the WPILib reference:

The WPI Robotics library (WPILib) is a set of software classes that interfaces with the hardware and software in your FRC robot's control system. There are classes to handle sensors, motor speed controllers, the driver station, and a number of other utility functions such as timing and field management.

Most Useful WPILib Classes

This is a list of the primary WPILib classes used in robot programming:

Command Based Programming

- Scheduler - scheduling a command
- Command - the basic framework for all commands
- Subsystem - a container for sensors, motors and pneumatics actuators. Keep in mind that we use our own LoopableSubsystem

Operator Input (OI) Layer

- XboxController - An interface to the Xbox controllers we use to drive our bots
- SendableChooser - a way to select an auto pattern on the SmartDashboard
- SmartDashboard - put debug/status information on the SmartDashboard

Drive Calculator Classes

- DifferentialDrive - calculator for left/right drive robots
- MecanumDrive - calculator for 4 wheel mecanum drive robots

Speed Controllers (Motors)

- Jaguar, PWMSpeedController, SD540, Spark, Talon, TalonSRX, Victor, VictorSP - a motor connected to a PWM port
- TalonSRX - a TalonSRX connected to the CAN bus (requires the CTRE vendordep)

Sensors

- AnalogInput - an analog input (ie. proximity sensor)
- DigitalInput - a limit switch or other digital input plugged into a DIO port
- Counter - a counter attached to a DIO channel used to count fast DIO pulses

Pnematics

- Solenoid - a pneumatic single solenoid
- DoubleSolenoid - a penumatic double solenoid

Key IP Address Lists

- roboRIO USB: 172.22.11.2
- roboRIO mDNS: roboRIO-####-FRC.local (where #### is your team number with no leading zeroes) You should be able to use this address to communicate with the roboRIO over either interface through ping, browser, etc.
- Robot Radio: 10.TE.AM.1 (where TE.AM is your 4 digit team number with leading zeroes if required)
- DHCP range: 10.TE.AM.20 to 10.TE.AM.199