FRC Electrical Bible

mililanirobotics

Published with GitBook

Table of Contents

The Electrical Bible - Beta

Welcome to our Electrical FIRSTopedia!

We created this guide in hopes of combining a basic understanding of all the electronic components the FRC robot uses into one cohesive and comprehensive guide. This guide was made with the power of Google Docs, GitBook, our phones' cameras and endless Google searches.

This guide has been updated to include the 2015 control system. Our older document can be found here, and includes documentation on the cRIO.

Something to note is that we program in C++, so references to classes like "Encoder" and our sample codes are written in C++. However, classes are interchangeable in Java and C++, and there are topics covered in this guide that are not language-exclusive like the roboRIO and the Driver Station itself. We hope this is guide serves to be useful for you!

Team 2853 Electrical/Programming Team

The roboRIO

1.1 General roboRIO Overview

- What is a roboRIO?
- Connectors
- Wiring Diagram

1.2 Configuring the roboRIO

- Installing/Updating New Firmware
- Imaging/ Reimaging a roboRIO

1.3 Connecting to the roboRIO Wirelessly

1.4 Uploading Code to the roboRIO

1.5 The CAN Bus

- Introduction
- Wiring the CAN

General roboRIO Overview

➠ What is a roboRIO?

A roboRIO is a more recent version of the cRIO (compact Reconfigurable Input/Output) that was introduced to FRC teams in the 2015 season. It is a faster, smaller, and more powerful version of the previous controller. Like the cRIO, it acts as the brain of the robot, and connects to a D-Link router using an ethernet cable. Additionally, the roboRIO combines the functions of the digital sidecar including the digital and analog modules of the previous control system. The roboRIO is more robust than the digital sidecar, and is protected from shorts between its external pins.

Specs

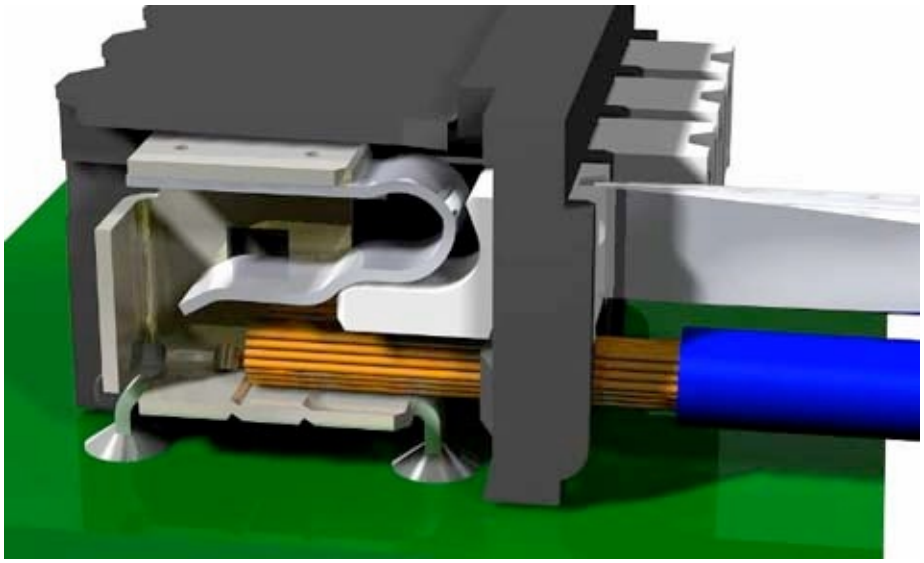
Basic Overview

- Dual-Core ARM Cortex 667 MHz processor
- 256 MB RAM
- 512 MB NAND storage memory
- Linux Operating System with real-time extensions Supports LABVIEW, C++, and Java

Physical/Electrical Characteristics

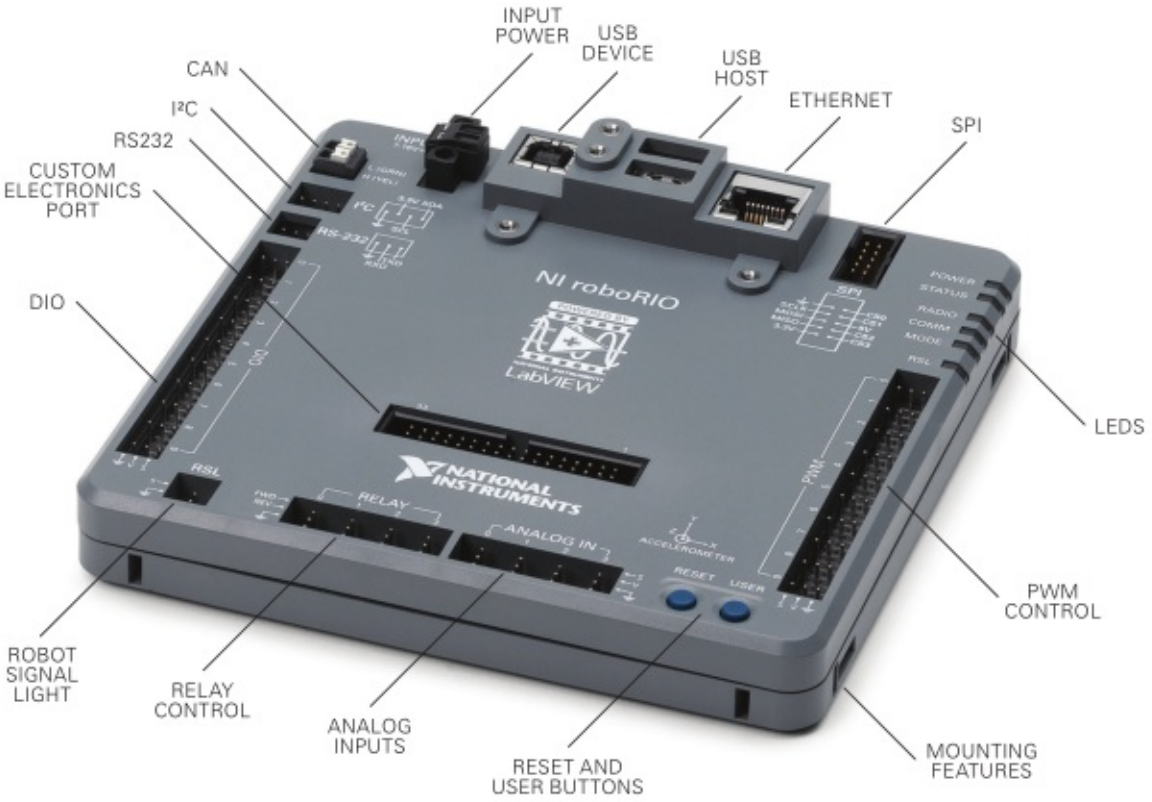
- 6.8 V to 16 V input power (staged brownout from 4.5 V to 6.8 V)
- 3.3 V user output (with 1.5 A maximum)
- 5 V user output (with 1 A maximum)
- 6 V servo output (with 2.2 A maximum)
- Operating temperature 0°C - 40°C
- Storage temperature -20°C - 70°C
- 5.7 inches by 5.6 inches, weighs 12 ounces
- I/O and Communication Ports
    - 10 dedicated PWM channels
    - 10 DIO dedicated channels
    - 4 bi-directional
    - Relay Control channels
    - 2 USB Host ports
    - 1 USB Device port
    - 1 Ethernet port
    - 1 CAN Port
    - 1 Integrated, 3-axis accelerometer
    - 12 V Robot signal light channel

➠ Connectors



Weidmuller Connectors are used to supply power to the roboRIO, PCM (Pneumatic Control Module), and VRM (Voltage Regulator Module). The connector accepts wire gauges from 16 AWG to 24 AWG. Wires can be inserted into or removed from the Wiedmuller connector by pushing down on the white tab using a tiny flathead.

➠ Wiring Diagram



Notes:

- The CAN Ports on the roboRIO are used to connect to the PCM and PDP
- The roboRIO is connected to the PDP through its input power ports. Do not connect the roboRIO directly to the robot battery.
- The USB Device port can be used to connect to a computer to update the roboRIO firmware and to reimage the roboRIO.
- The LEDs indicate the current status of the roboRIO. The roboRIO can be mounted using zip-ties through the mounting features.
- Please see the Appendix for an example wiring of the whole control system.

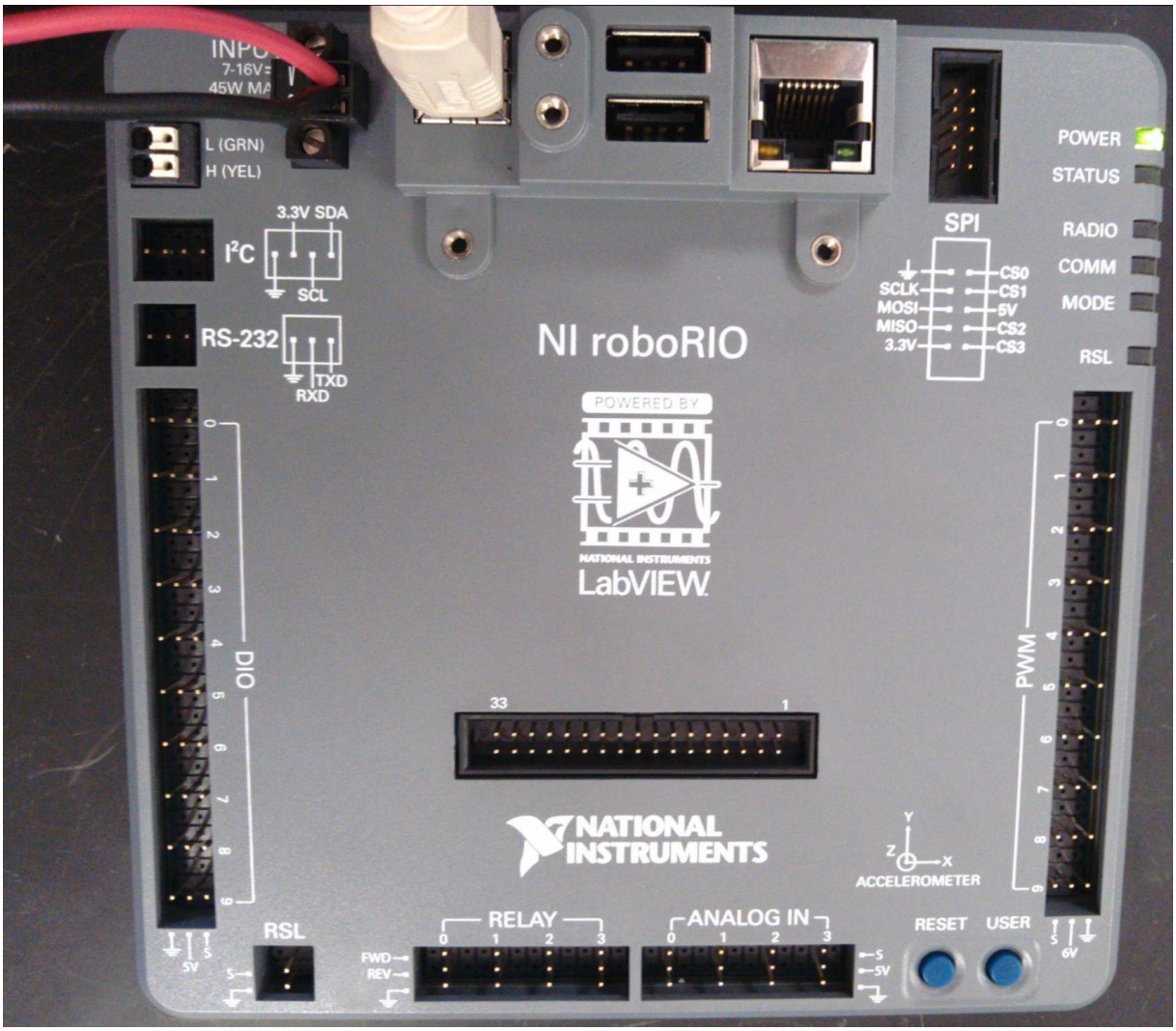
Configuring the roboRIO

Before a brand new roboRIO can be put into action, you must first install the latest roboRIO firmware and then re-image the software using the latest version. Before you begin, ensure that the NI (National Instruments) suite is updated. The NI update as well as instructions for downloading it can be found below:
https://wpilib.screenstepslive.com/s/4485/m/13503/l/144150-installing-the-frc-2015-update-suite-all-languages

➠ Installing and Updating the roboRIO firmware

Prior to the imaging of the roboRIO, the firmware on the roboRIO must be upgraded to the latest version. This process will provide the bootloader, safemode, and firmware for the roboRIO. While it is possible to do this using an ethernet connection, it is not recommended. The firmware can be updated using your system's web browser in the following steps:

1. As shown in the electrical layout below, connect the roboRIO to your computer using a printer USB cable and supply power to the roboRIO.

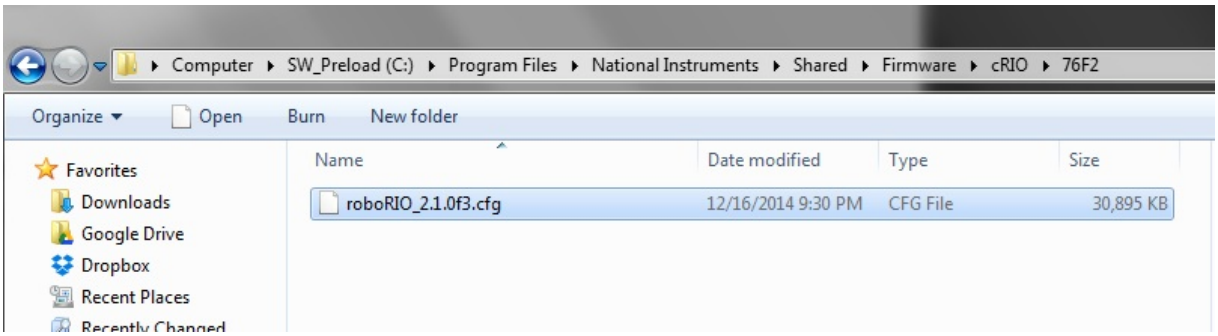


2. The driver should install automatically. Once the download is complete, open a web browser on your computer.

3. In the address bar of the web browser, type "172.22.11.2" and press enter.

4. Click login. "admin" will be the username and leave the password field blank. Then, click "Update Firmware"

5. Look for the roboRIO (.cfg file in your National Instruments folder. The default location of this file is under "Program Files\National Instruments\Shared\Firmware\cRIO\76F2"



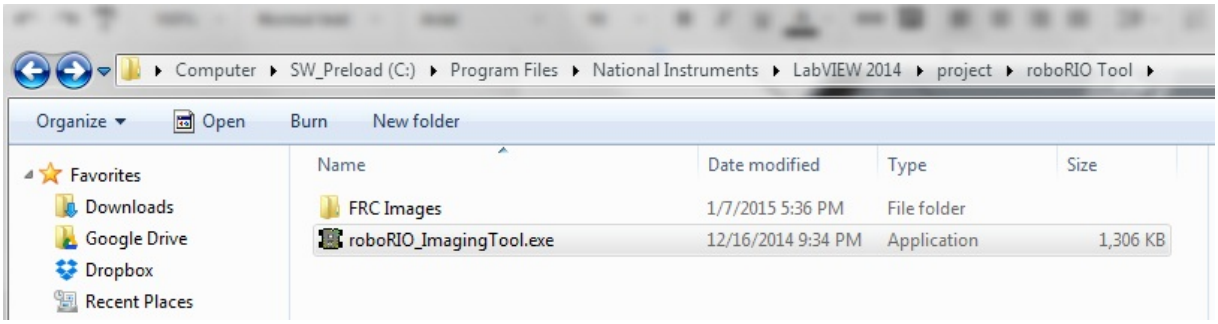
6. Click “Begin Update”

➠ Imaging and Reimaging the roboRIO

The roboRIO Image loads the FPGA, operating system, linux file system, and default settings for the roboRIO. Now that the roboRIO has the latest firmware installed, it is now possible to image the roboRIO in the follow steps:

1. Ensure that the roboRIO is still powered from the PDP (Power Distribution Panel) and is still securely plugged into the computer using a printer USB cable.

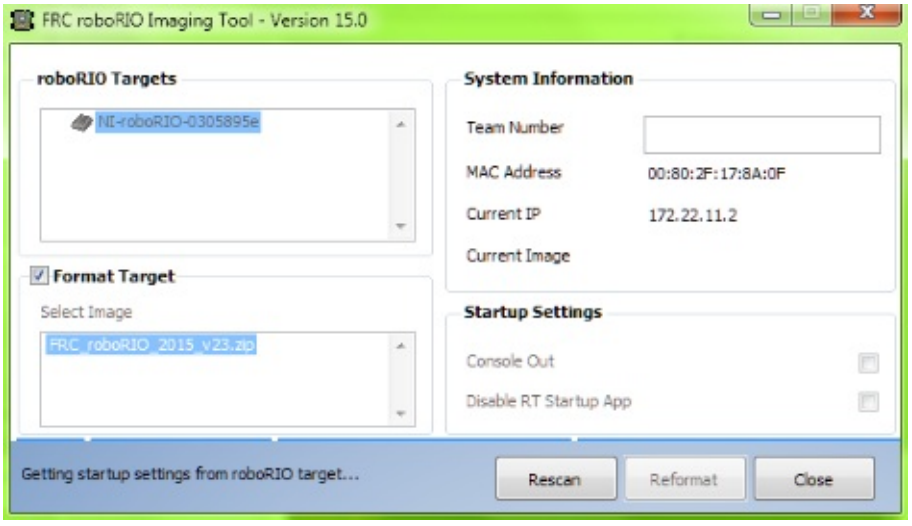
2. Look for the roboRIO Imaging tool (.exe file) as shown below. It is located within the National Instruments file folder, and its default location is “National Instruments\LabVIEW 2014\project\roboRIO”



3. Double click to run the “roboRIO_ImagingTool.exe” application, and the the interface will attempt to identify connected roboRIO devices.

4. Enter your team number under “System Information.” Select the latest version of the roboRIO Image (indicated by the highest version number) and the name of the roboRIO you intend to image.

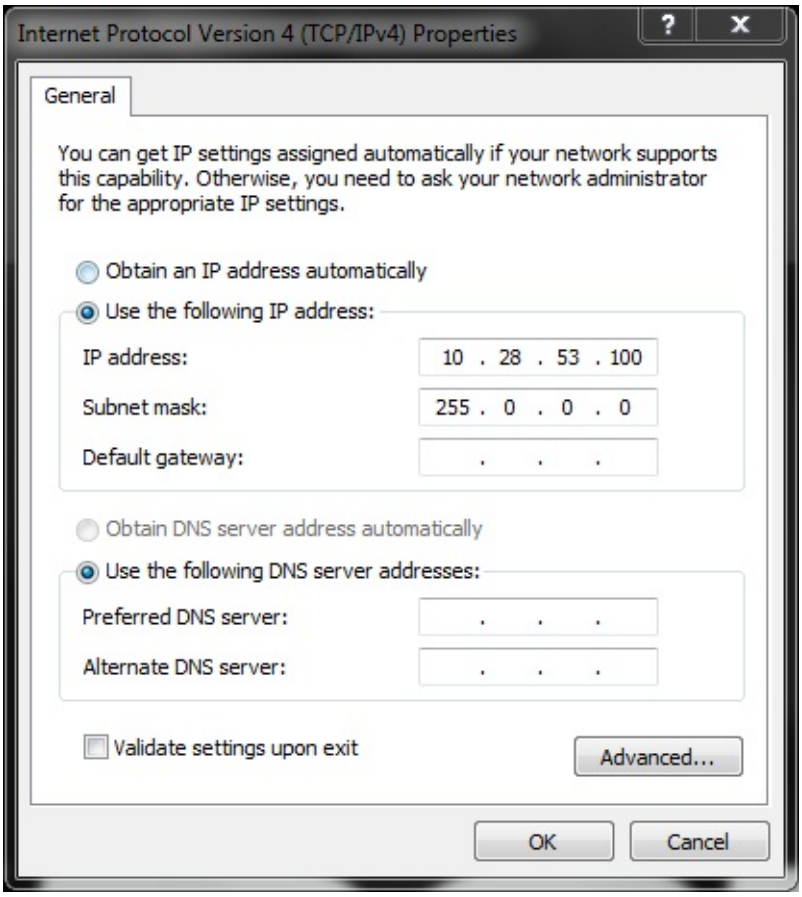
5. Select the Format Target option but make sure that the Console Out and Disable RT Startup App options are not selected. Click “Reformat”



6. Once the Imaging is complete, click “OK” on the completion message and Reboot the roboRIO using the Reset button.

Connecting to the roboRIO Wirelessly

1. Change the IP address and subnet mask of your computer. The IP address should be “10.xx.yy.100”, with xx and yy being your four digit team number, and the subnet mask should “255.0.0.0”



2. Connect to the team’s wireless router. It’s that easy!

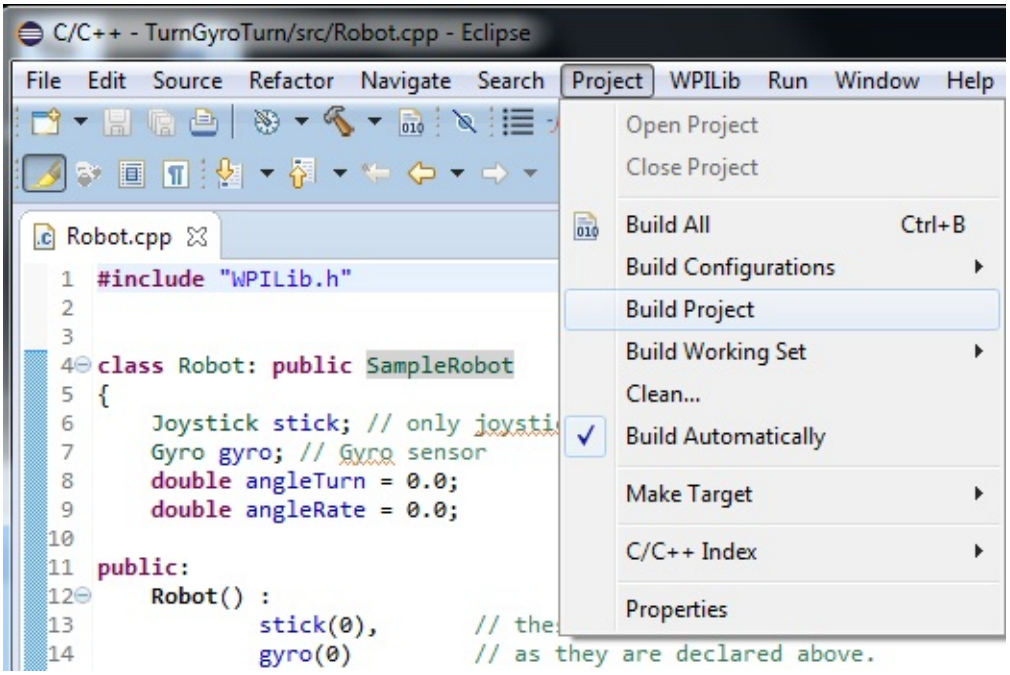
Uploading Code to the roboRIO

With the introduction of the roboRIO to FRC teams in 2015, uploading code to the robot is made easy! Here's the steps:

1. **Establish a network connection to your robot network.** On your computer, go to your network connections and connect to the router that is connected to the roboRIO. Don't worry if you're already connected to another network, as you will automatically disconnect from that network connection.

Connecting to a network

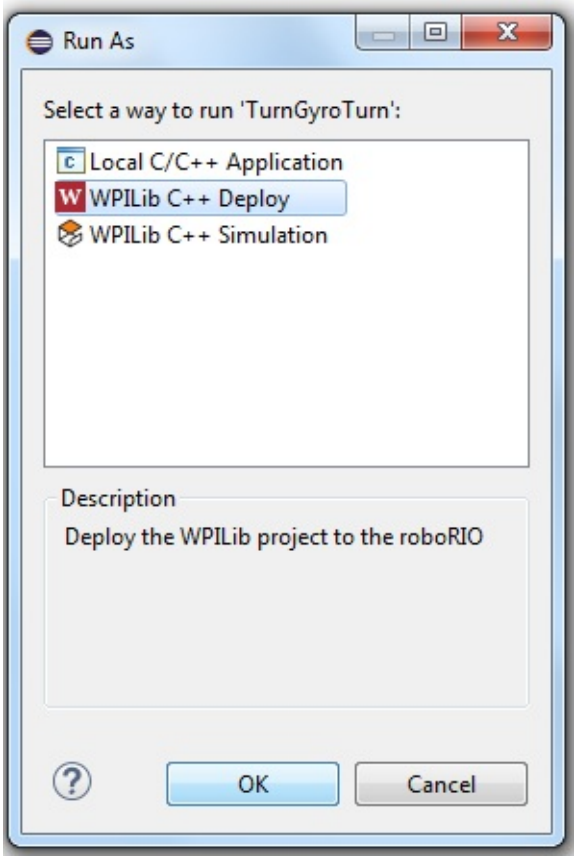
2. **Build your Project.** Click the "Project" option and then click "Build Project." To upload code to the roboRIO, it is only necessary to build the project for uploading code that hasn't been build in the past or was modified, but it's a good habit to always build your projects before uploading them to the robot. Therefore, we recommended having your "Build Automatically" setting enabled.



Building an Eclipse project

3. **Run your code.** Press Ctrl + F11, and when prompted how you would like to run your code, select the "WPILib C++

Deploy” option and press OK. Now you’re done! Wait for the Driver Station to show that communications has been established, and you’ll be on your way to testing your code!



Running your code using “WPILib C++ Deploy”

The CAN Bus

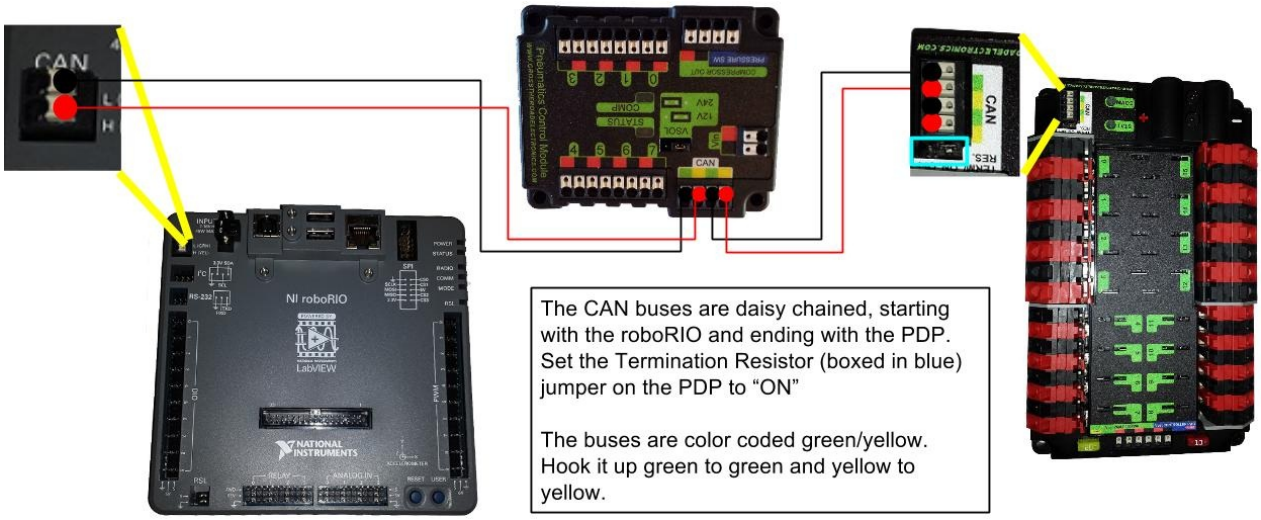
➠ Introduction

As of the 2015 FRC game, Recycle Rush, teams are required to use the CAN bus on the roboRIO when connecting to the PCM, PDP, Talon SRXs, and Jaguars (R 60-62).

This section will not go over how to use CAN on the Talon SRX (you can read CTR's Talon SRX User guide) or the Jaguar, because our team does not have experience with these items.

By connecting the control system together with CAN, the PDP and PCM can communicate with each other. The PDP provides monitoring for each of its outputs. A main advantage of this system is that compressors and pressure switches no longer need to be connected to a Relay Spike.

➠ Wiring the CAN



When wiring, we recommend twisting the two wires together. There are no specific buses for input/output but by convention, we use the left bus as input and right bus as output.

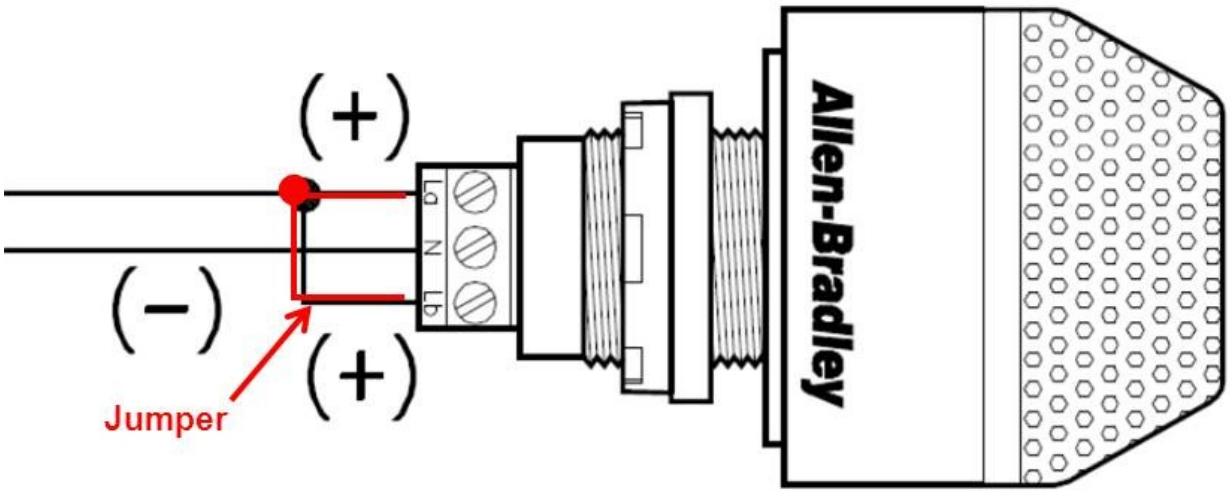
If you are not using the roboRIO or PDP as terminals for the CAN chain, terminate the CAN chain by inserting a 120 Ω resistor into the Weidmuller terminals.

The Robot Signal Light

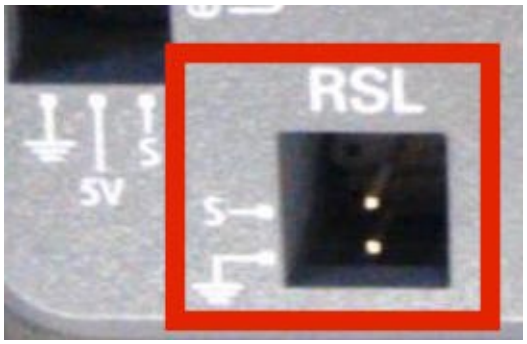
➠ Introduction

The Robot Signal Light (RSL) is **mandatory** during competition and acts as a signal to whether the robot is connected to the FCS, in teleop mode, etc.

➠ Wiring the RSL



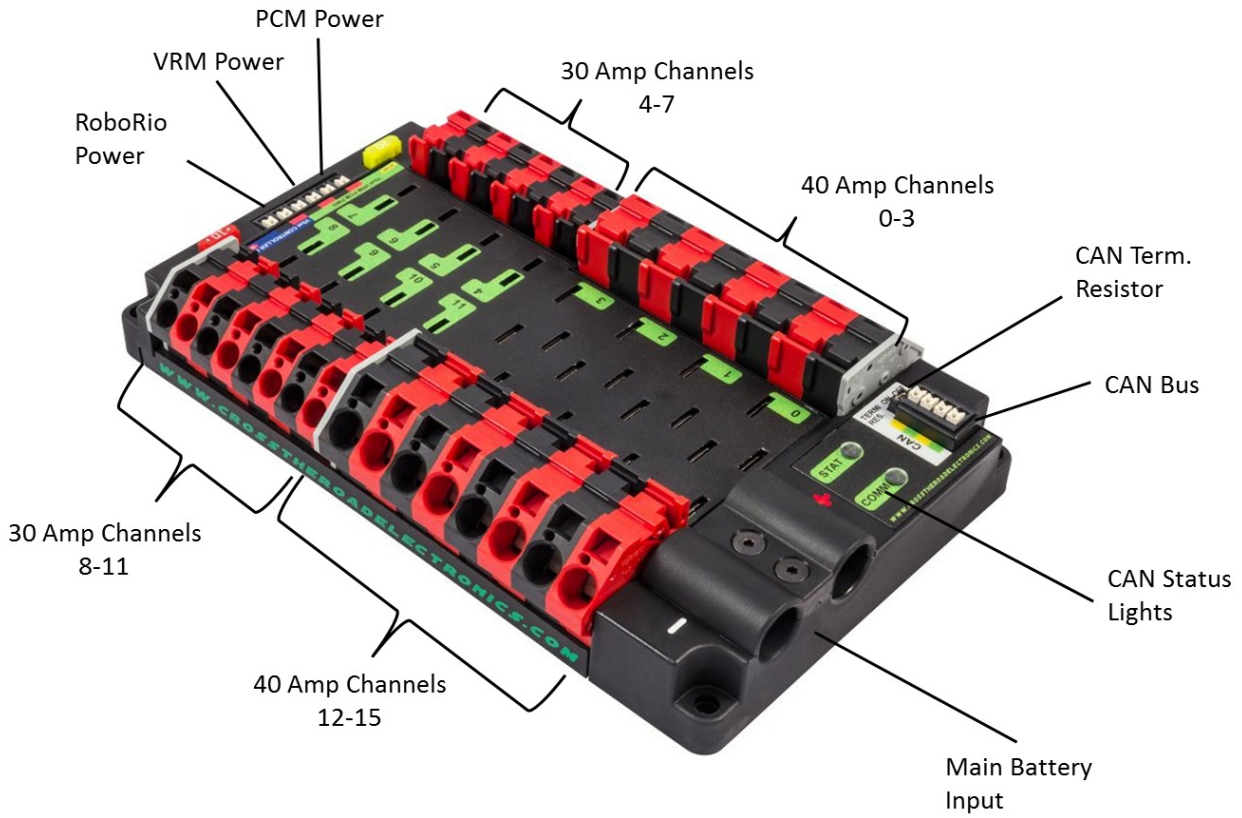
A separate wire acts as a jumper between La and Lb. Connect La to 'S' and N to Ground on the roboRIO.



➠ RSL Indicators

Light Code	Meaning
Solid ON	Autonomous enabled
Solid ON but blinks off every 1.5 sec	Teleop enabled
Slow blink (900ms ON / 900ms OFF)	System disabled by system watchdog, user watchdog, or driver station set to disabled
Fast-slow (200ms ON / 900ms OFF)	Low battery (<12V) or no user code AND system disabled either by system watchdog, user watchdog, or Driver Station set to disabled.
Fast (200ms ON / 200ms OFF)	System error; no Driver station communication; bad cRIO image, bad team ID, extensive communication errors

PDP



The PDP is a newer iteration of the PDB that was in the Kit of Parts given to teams in the 2015 FRC season. The main difference between the two is that the PDP is slightly smaller, lighter, and has updated connectors, which includes the Weidmuller Connectors used on the CAN bus. In general, the PDP serves the same function as the PDB, distributing power to the system, but is improved to better reflect the updates made to the rest of the control system.

Specs

- Dimensions: 7.586” x 4.748” x 1.442” tall (rough)
- Weight: 1 lb and 5.3 oz
- 4 Mounting holes (one at each corner, smaller than 1/4-20 and larger than 10-32 fasteners)
- Connectors
  - Main Battery Input
    - 2 Bushing lugs
    - Thread M6x1 (size 6 mm)
  - Power Channels (0-15)
    - 8 Red / 8 Black WAGO Connectors 30 Amp Channels (4-11)
    - 8 Red / 8 Black WAGO Connectors 40 Amp Channels (0-3, 12-15)
    - 6 Position Weidmuller Connectors (accept 24-16 AWG)
      - 1 x 20 amp fuse (used for PCM and VRM)
      - 1 x 10 amp fuse (used for RoboRIO)
  - 8 x 20 / 30 amp Thermal Breaker by Snap Action slots (channels 4-11)
  - 8 x 40 amp Thermal Breaker by Snap Action slots (channels 0-3, 12-15)
  - CAN
    - 4 Position Weidmuller Connectors
      - 2 Yellow CAN High
      - 2 Green CAN Low
- Status Lights

- Both Lights are always the same color / blinking pattern with the exception of booting up
- Fast Green Blink - Robot is enabled
- Slow Green Blink - Robot is disabled
- Slow Orange Blink - Robot is disabled & Sticky Fault present (low voltage, < 6.5 V
- Slow Red Blink - No CAN communication
- Boot Status lights (COMM LED only)
  - Green / Orange Blink - Device is in boot-loader / Field-upgrade necessary
- Both LEDs off - Device is NOT powered

The D-Link

Introduction

We control the robot through a Logitech gamepad controller. Generally, we wouldn’t want to follow the robot around with an ethernet cable, so we solve this problem by connecting wirelessly. We use the D-Link as a medium to communicate with the robot in this fashion.

The Physical Layer

➠ The Voltage Regulator Module

As of the 2015 FRC game, Recycle Rush, the DLink is required to be powered by the 5V/2A AKA “Radio”port on the Voltage Regulator Module. You can read more about it under “The Power Distribution Panel”

The D-Link itself is connected to a power adapter that is:

- 5V Output
- Power Cable to D-Link Model No: AMS3-0502000 FU
  - Barrel 5.5/2.1mm

AndyMark - Power Converter |

The D-Link connects to the cRIO via a Standard CAT-5 Ethernet Cable. It can also act as the “middle-man” with an Ethernet cable connecting to the Driver Station laptop.

- Make sure the router has power
- Make sure the ethernet cables are plugged in securely (on both ends)

NOTE: It doesn't matter which LAN Ports the ethernet cables are plugged into. However, by convention we usually make sure that the roboRIO connects to port 2 and a computer plugs into port 1.

Configuration

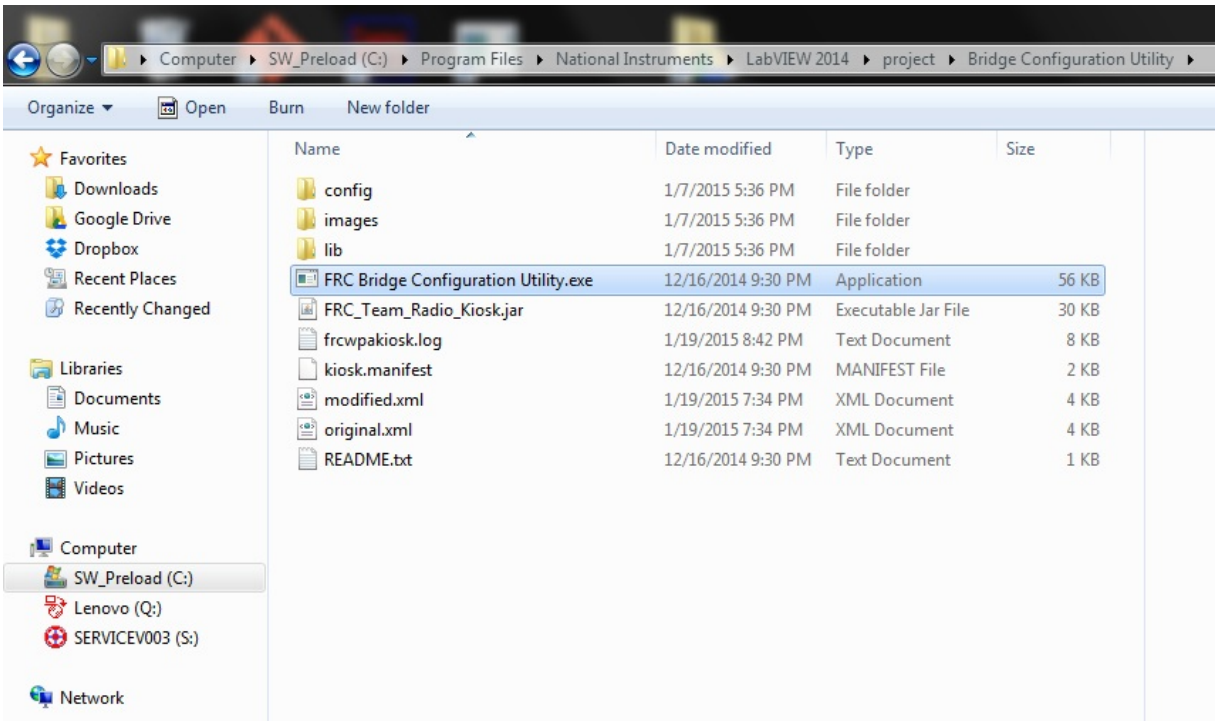
➠ 2015 Configuration Prerequisites

1. Make sure that the router is switched to the Access Point mode you request (2.4 Ghz, 5.0 Ghz, Bridge). A setting of 2.4 Ghz is appropriate for most FRC teams and should be used as the default.
2. Ensure that your computer has Java installed. If you're unsure or need to check or download the Java software, use this link: http://www.java.com/en/download/index.jsp
3. Also ensure that the NI Suite is updated, which includes the FRC Bridge Configuration Utility to configure a router to use in FRC. To download the New FRC Software, use this link: http://wpilib.screenstepslive.com/s/4485/m/13503/l/144150-installing-the-frc-2015-update-suite-all-languages NOTE: To reset the router press and hold the reset button for 30 seconds to perform a factory reset to wipe any previous configurations

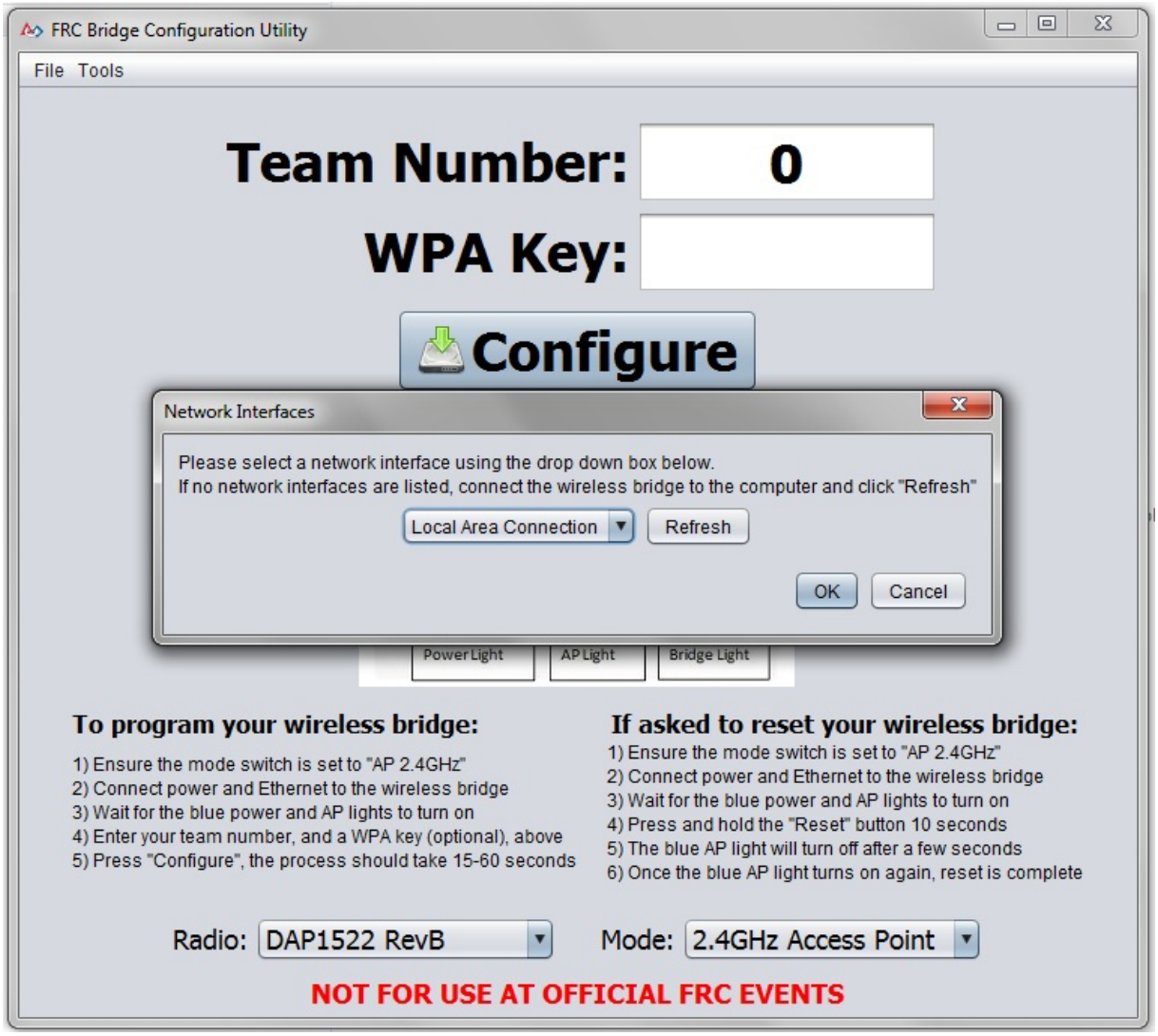
➠ FRC Bridge Configuration Utility

NOTE: If the router is not brand new, it is not necessary to press and hold the reset button for 30 seconds to perform a factory reset to wipe any previous configurations because the FRC software will do so for you.

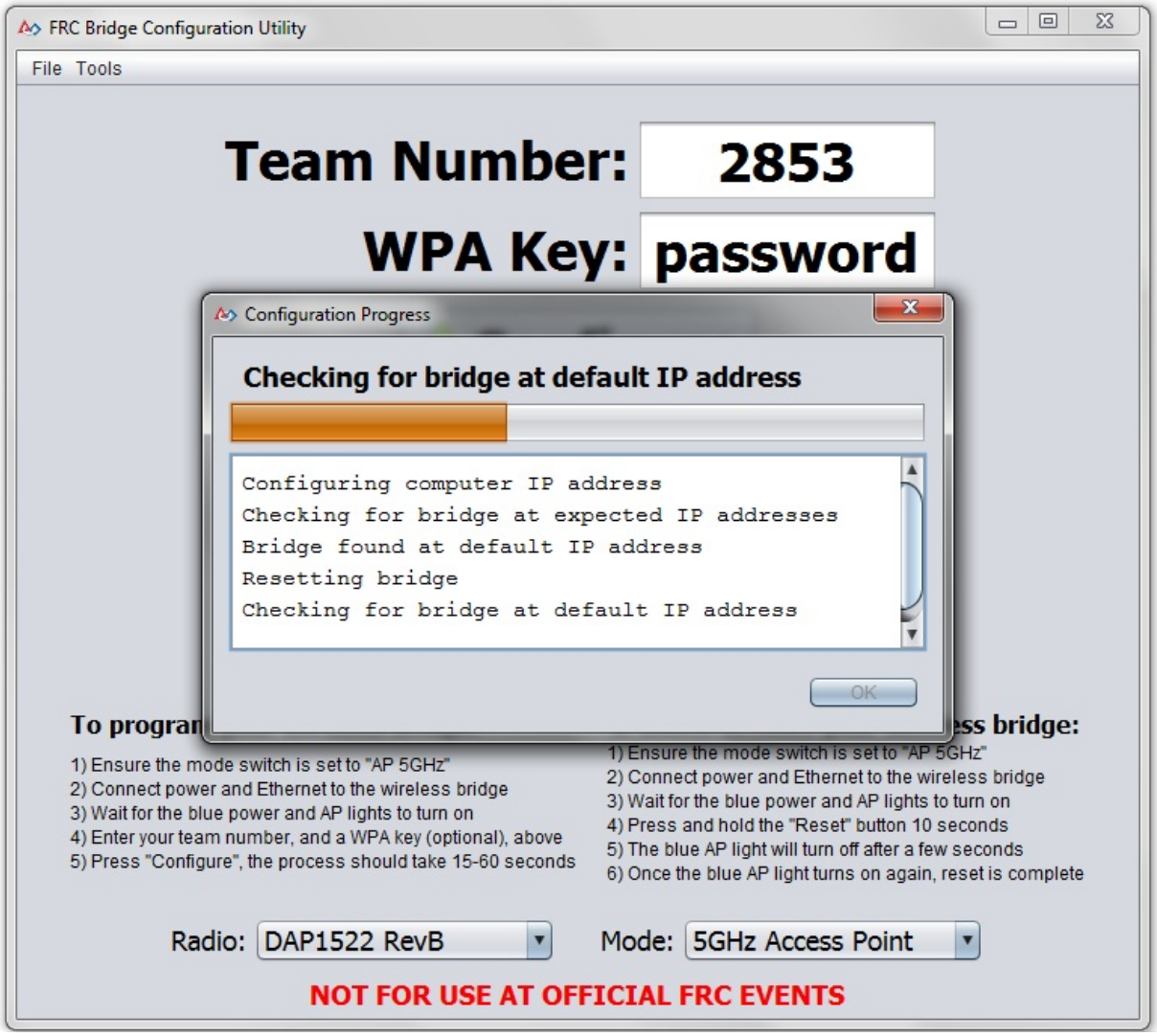
1. Turn off the Wifi on your computer
2. Make sure the physical layer on the router is setup properly.
3. Launch the FRC Bridge Tool software. It is located under the National Instruments folder, and its default location is C:\Program Files\National Instruments\LabVIEW 2014\project\FRC Bridge Configuration Utility.exe



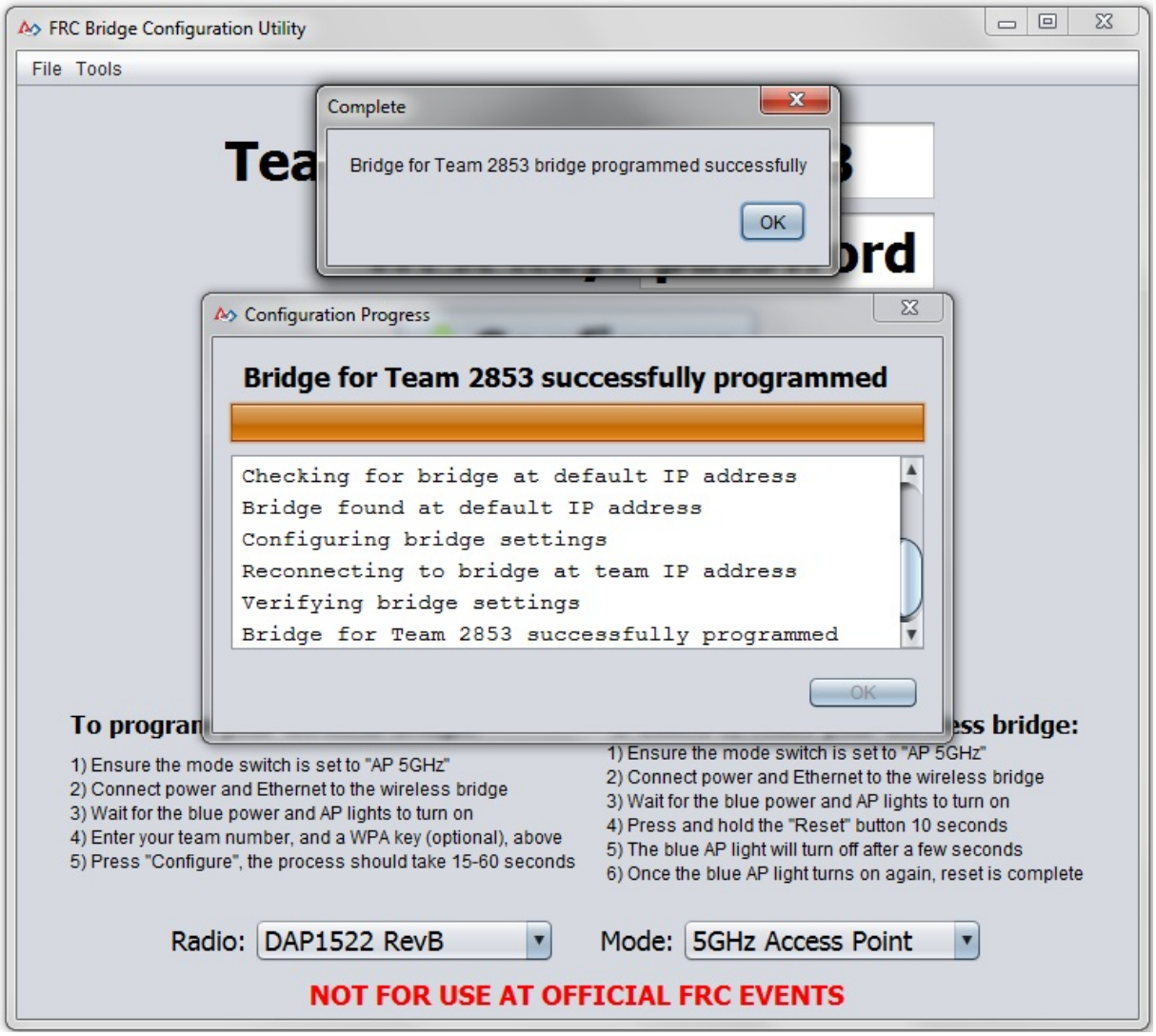
4. Under the Network Interfaces popup from the Bridge Configuration Utility, Select "Local Area Connection" and press OK. If there are no network interfaces shown, click the refresh button.



5. Under the FRC Bridge Configuration Utility, Type in your FRC team number as well as a password to be set under the section for “WPA Key.” Ensure that the Radio option is set to DAP1522 RevB, and that the Mode option is set to the current prerequisite setting (default of 2.4GHz Access Point should be used). Now click “Configure.”

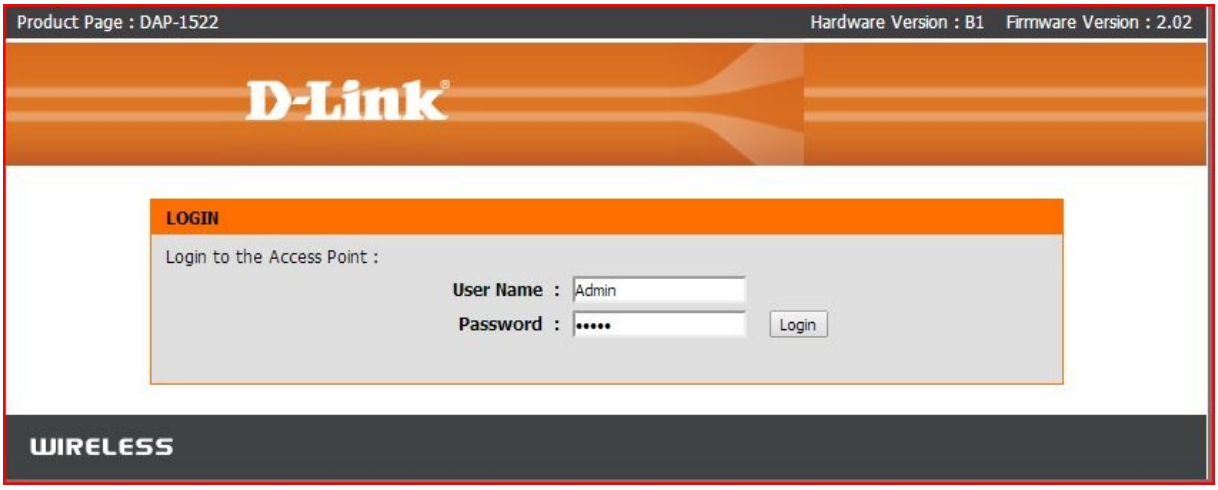


6. Wait until the Configuration Progress is complete, and then press OK once it is done! Note that the router SSID can be configured through the D-Link ap

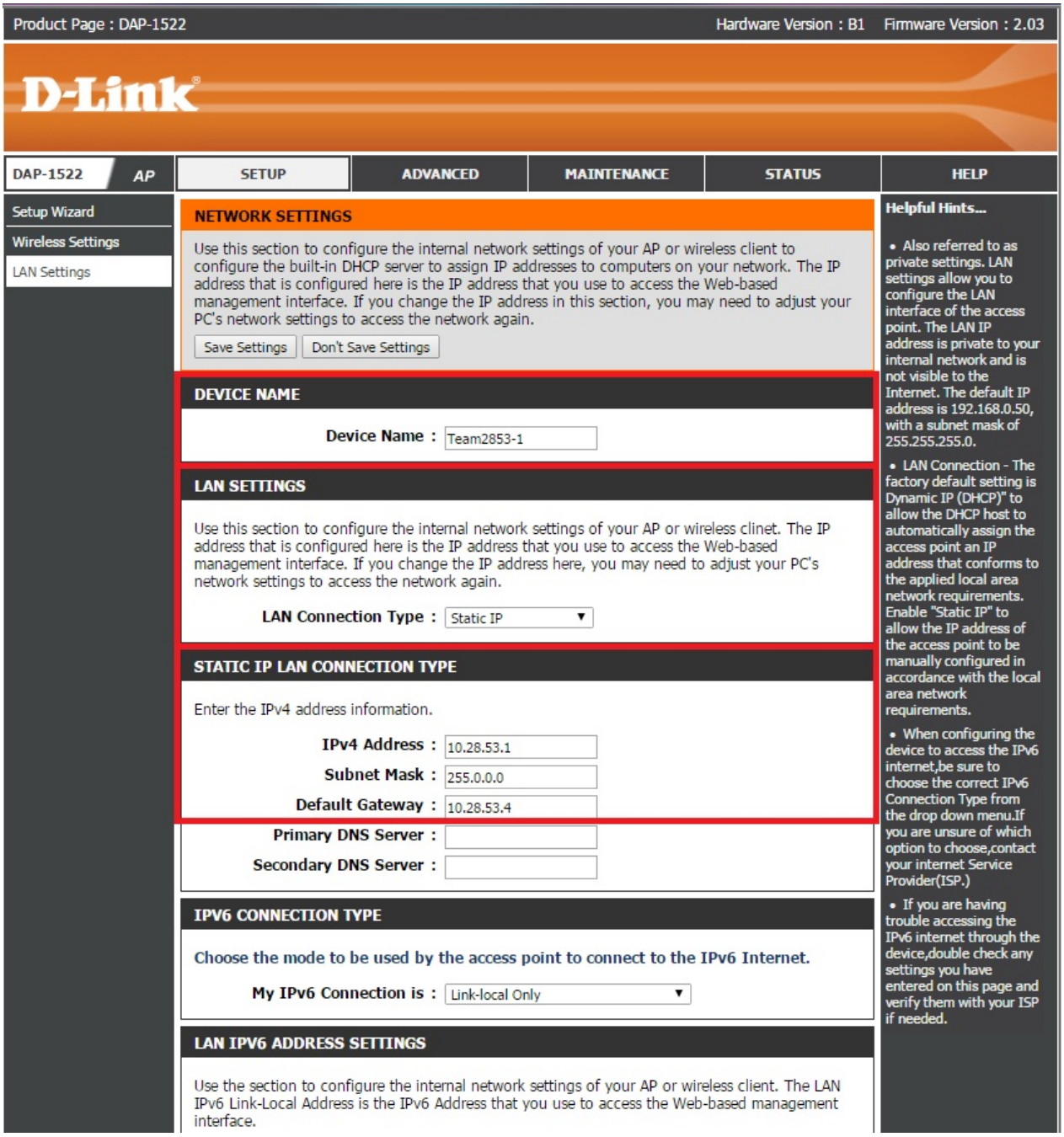


Manual Configuration

1. Open up your web browser (Firefox, Chrome, etc.)
2. In the address bar, type in the IP address of the router. It's either:
   192.168.1.1, the default IP address
   8OR 10.xx.yy.1, where **xx** is the first two digits of your team's # (it can be one digit if your team has a three-digit #) and **yy** is the last two
   OR type in dlinkap/ and hit enter.
3. You'll get a prompt for the router name and password like the one shown below. On a brand new router or a router that was reseted, the login info will be this:
   User Name: Admin
   No Password (by default)



➠ Setup: LAN Settings



➠ Set a Device Name: Team**xxxx-y**

xxxx = Your team number to avoid confusion.

y = Arbitrary but unique number to your router to avoid confusion. You should base it on the number of routers your team owns. Example Device Name: **Team2853-1**

➠ Set the LAN Connection type: **"Static IP"**

➠ Configure the *IPv4 Address, Subnet Mask, and Default Gateway*

IP Address: 10.**xx.yy.1**

xx.yy = Your Team Number

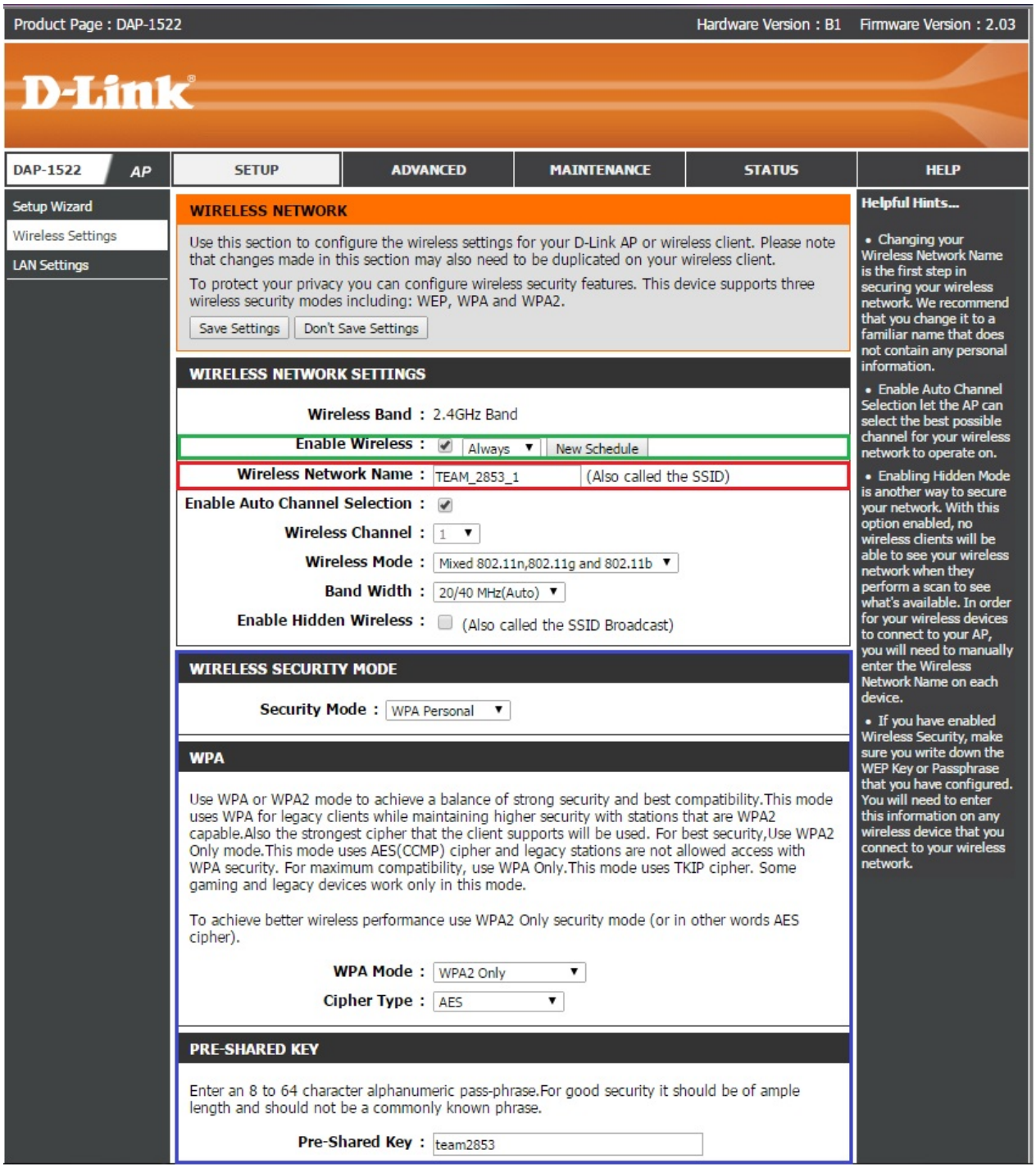
xx *can both be the one digit if your team has a three-digit number*

Example IP Address: **10.28.53.1**

Subnet Mask: **255.0.0.0**

Default Gateway: **10.28.53.4**

➠ Wireless Settings



Enable Wireless:

Checkmark the Box

Set to **Always**

Wireless Network Name:

Name it to (**whatever you want**)

Wireless Security Mode:

Security to **WPA Personal**

WPA mode to **WPA2 Only**

Cipher Type to **AES**

Pre-shared key is the **Network Security Key**

Troubleshooting the D-Link

- Ensure that all necessary cables (Power and/or Ethernet cables) are plugged in securely.
- Ensure that the switch on the back of the router is set to “AP 2.4 GHz” unless you have specifically set the router and its configuration to a different mode.
- Ensure your router is not labeled BROKEN with electrical tape.(Team specific)
- Try resetting the power on the router. Remove its power source, wait 10 seconds, then plug the power cable back in. Sometimes wizards come and magically fix the router when you do this.
- Try powering through an outlet using the power adapter that came with your router.
- Try resetting the router’s data. Get a toothpick or some other thin object, then use it to hold the reset button down for 10 seconds. Wait about 30 seconds for it to reboot. Then, refer to the initial configuration process above.
- Routers do break down and its possible that it’s simply busted. However, these things are expensive so make sure it’s broken before declaring that you need to get a new one.

Additional Resources

Getting Started with the 2015 Control System

WPILib - Getting Started with the 2014 FRC Control System

Getting Started with the 2013 FRC Control System

D-Link DAP-1522 User Manual

AndyMark Product Page

Driver Station

4.1 Introduction

4.2 The Interface

- Main Display
- Operation Tab
- Diagnostics Tab
- Setup Tab
- Power & Can
- Messages & Charts

4.3 Printing to the Driver Station

Introduction

This is your interface--neat, organized simplicity. It isn’t necessary to know how to write code in order to use the Driver Station, you simply need to know the following:

1. How to deploy code (connecting to the robot is a given)
2. What deployed code does

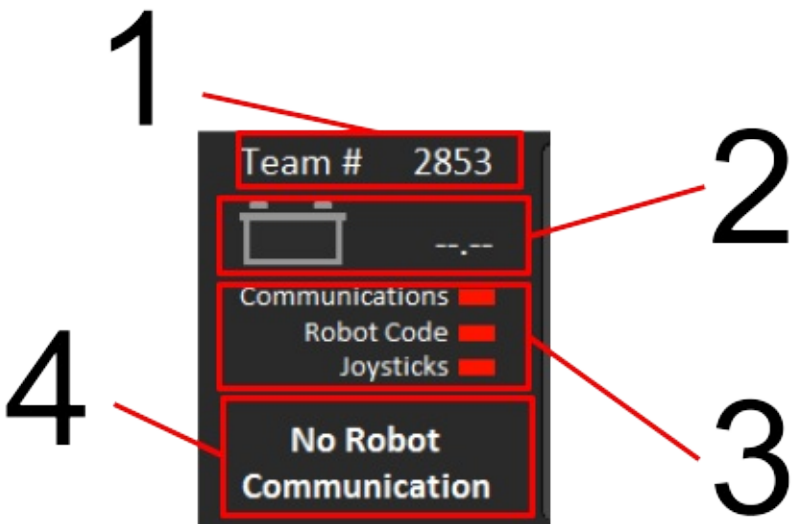
This is the program that allows to to test and use your code on FRC components. The Driver Station is the middle-man between you and the robot!, this is your middle man between your code and the robot!

Driver Station definitely got an update: revised appearance, more features, what’s not to love? Together we can relearn this adjusted Driver Station, it shouldn’t be all that different compared to its last iteration, depending on how you look at it.

The Interface

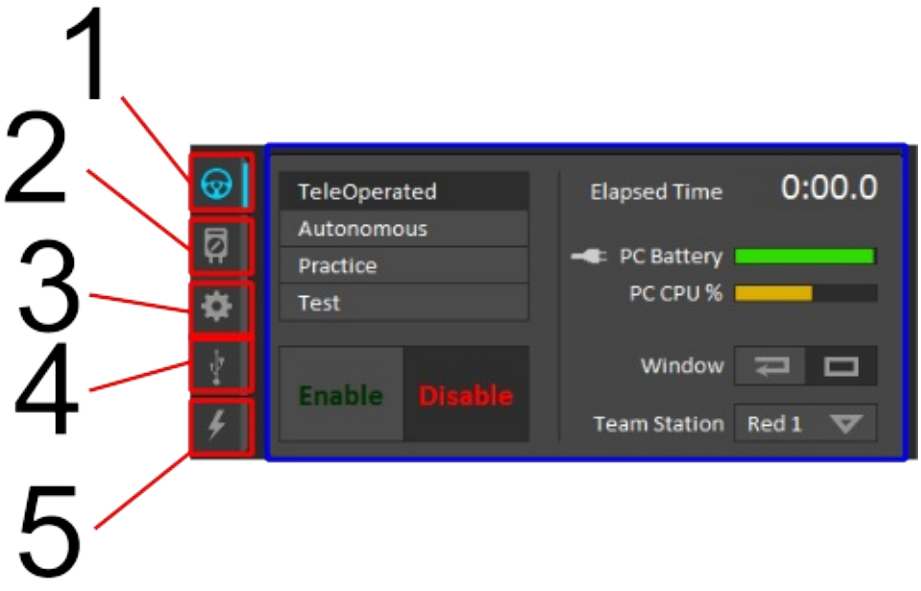
➠ Main Display

This display is independent from the sections left and right of this display, it will always be in view(while Driver Station is open of course) even if you switch tabs on the Driver Station(See Tab Selection, also Charts & Messages)



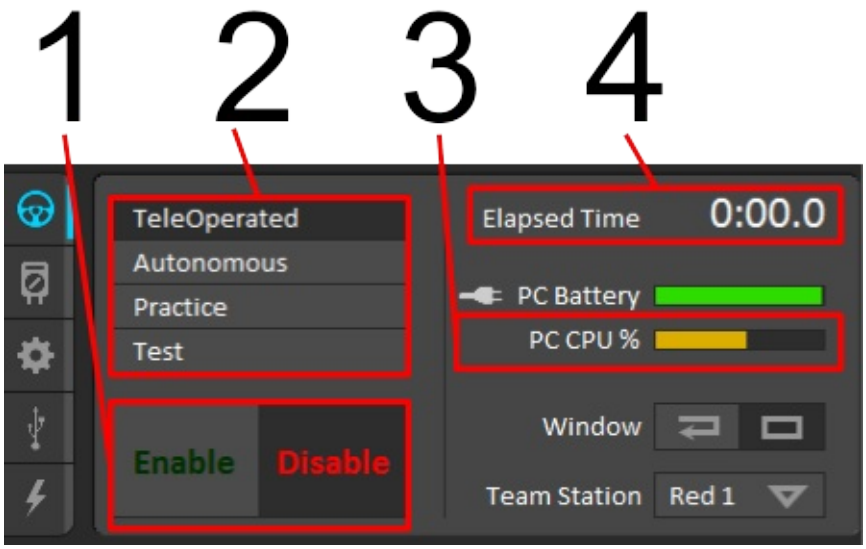
1. Your Team #, this can be configured personally in the Setup Tab (See Tab Selection Section : ➠Setup Tab)
2. Current voltage of the battery in V (volts) has a visual indicator to the left in the form of a pictorial representation of a battery (fills like a bar to represent amount of charge).
3. Lights indicating the Driver Station's status on detecting it: Red means there is no connection, green means a connection has been established. If you hover over each of the 3 lines, a troubleshooting message appears in the messages tab (See Charts & Messages) if light is red.
4. Displays current mode enabled or disabled unless the first two lights are red, in order, "No Robot Communication", "No Robot Code" (See Tab Selection : ➠Operation Tab for modes)

➠ Tab Selection



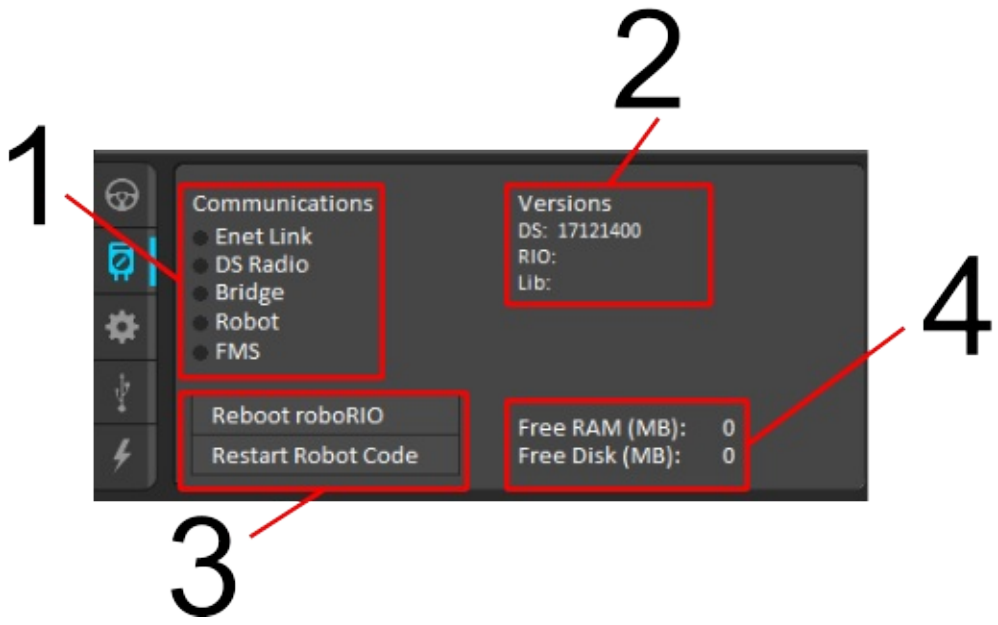
Red boxes indicate tab selection and the highlighted tab is the current tab, in the blue box is the currently displayed tab(automatically displays Operations Tab when Driver

➠ Operation Tab



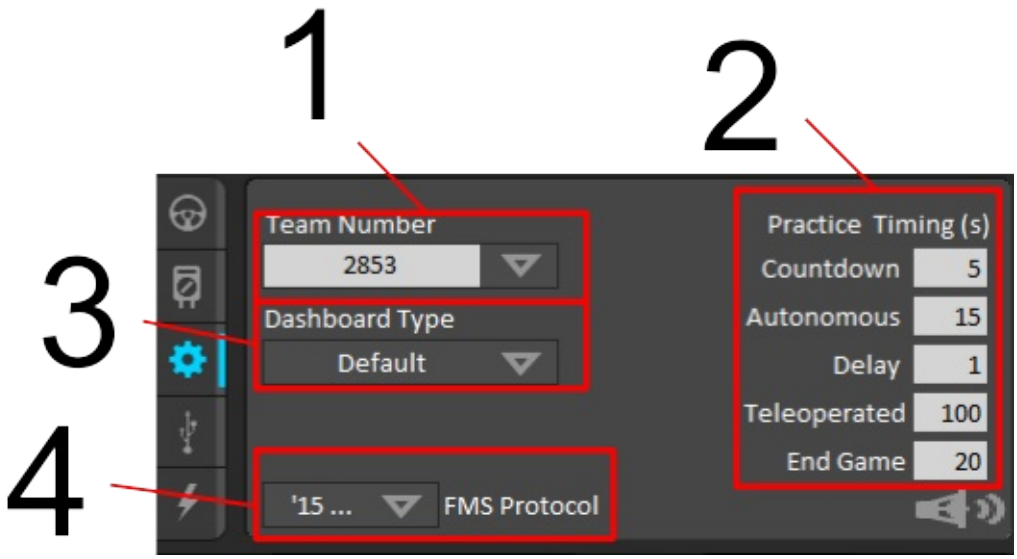
1. The depressed button is what state the period is in; can press []\ to quickly enable, can press Enter button to quickly disable if the first two lights in Main Display are lit (Communications & Robot Code)
2. Available modes, current mode is depressed(Teleoperated by default), you can switch modes by clicking one of the 3 buttons
3. NEW: Visual indicator of your PC CPU % usage Elapsed Time since you clicked “Enable” until disabled

➠ Diagnostics Tab



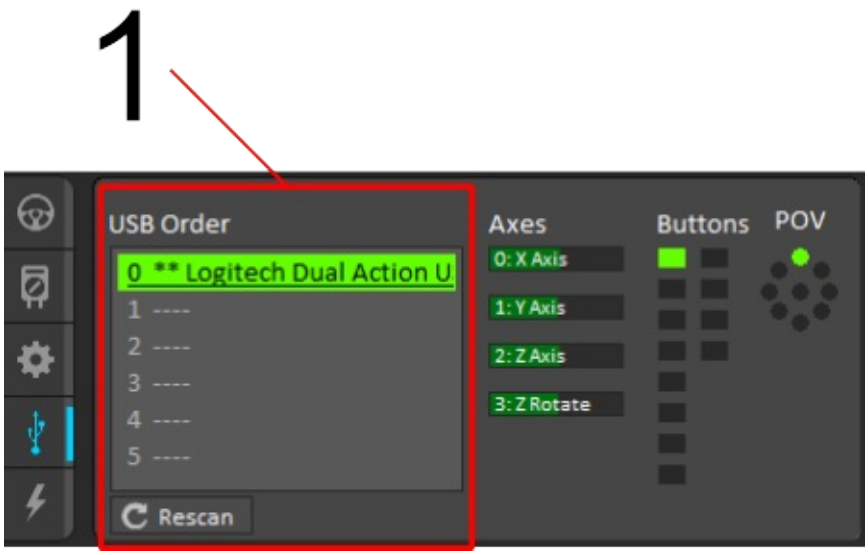
1. What the Driver Station has communications with; if plugged through Ethernet directly to something (router or roboRIO), Enet Link will light up. DS Radio is a legacy indicator of ping status of an external radio at 10.TE.AM.4(Compared to last Driver Station, this light will usually not be lit except under specific circumstances[TESTING]) Bridge will be lit if connected to the router, Robot will be lit if it can communicate with the roboRIO. FMS should be lit if at competition since it is mandatory to communicate with the Field Management System. If unlit, you can hover over them for troubleshooting tips in the messages box (See Messages & Charts)

➠ Setup Tab



1. Configure your team # here, click on the box type it in and boom
2. Practice timing controls how long Countdown till start of each period, Autonomous, Teleoperated, and how long endgame is. Delay is how much time is in between Autonomous and Teleoperated.
3. Type of dashboard you want to bring up, default auto brings up FRC PC Dashboard (ScreenSteps acknowledges an issue with setting Dashboard type to Java or C++ so to start up the SmartDashboard would require setting the default to SmartDashboard), Labview brings up FRC PC Dashboard, C++ and Java should bring up SmartDashboard, and remote is if the dashboard is on a separate computer/device
4. Field Management System protocol, protocol for DS to Field Management System communication; should be autoset to '15 which required for 2015 competition. Unless you were participating in week zero events in 2014, this won't have to be touched

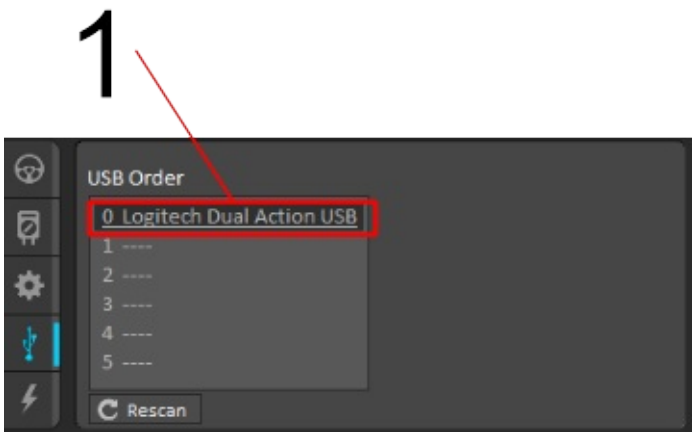
➠ USB Devices Tab



1. USB Setup list holds all compatible devices hooked up to the Driver Station (AKA, Laptop, usually 2, but with USB splitter, you can connect more) If you press a button on a connected device, it should be preceded by two asterisks (**) and highlighted in green. The rescan button forces a search of or for USB devices. While disabled, it automatically

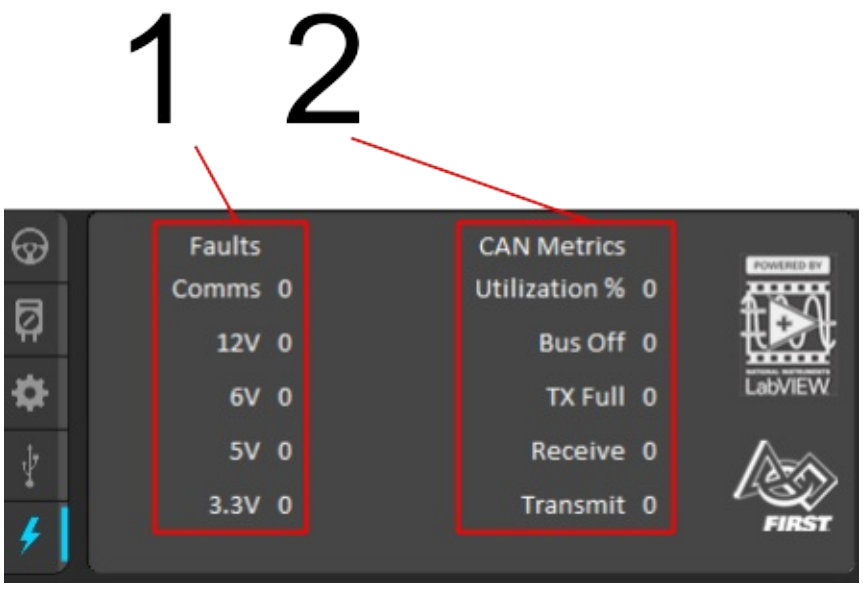
updates USB devices. Use the rescan button or press F1 to force search during a match.

Locking & Rearranging



1. To rearrange USB devices, drag & drop. When you drag & drop or double click on one of the devices, it underlines the device meaning it is “locked” Locked devices reserve the slot even while disconnected until reconnected, represented by greyed out and underlined.

➠ Power & CAN



1. Amount of faults that occurred since last connection to Driver station; Comms mean DS to robot communication, 12V is Brownouts(See roboRio for details), 6V/5V/3.3V are User Voltage Rail faults(typically short circuits)

2. Utilization % is as it states and the other 4 are types of CAN faults since last connection to Driver Station

➠ Messages & Charts

1. Logs is an independent button of the tabs on this section of the DS
2. Clears all messages currently in the box
3. Message Filter: Filled in icon filters out warnings from being posted in message box, Outline posts everything
4. The message box, troubleshooting tips appear here, messages will appear here

Logs

Loads up the DS Log File Viewer, you can view messages & event data in one display, a tool usually used for troubleshooting. This tool has its own section to be added elsewhere.

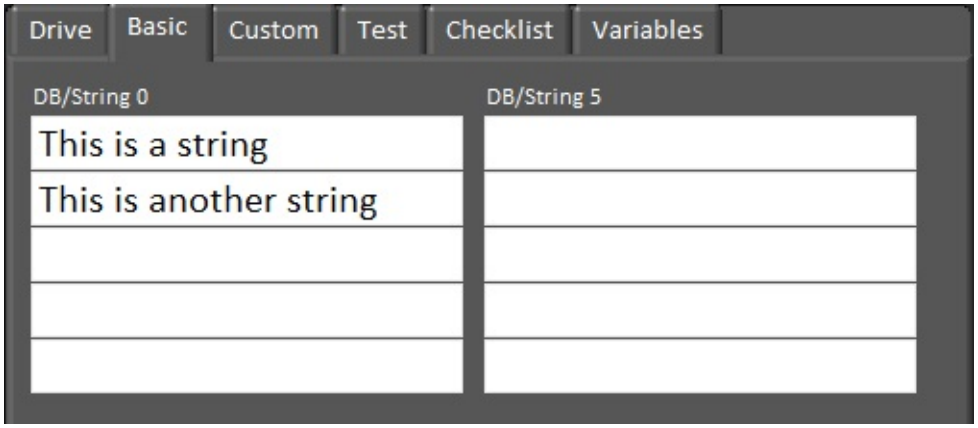
5.Charts trip time for data to robot with a green line vs the right axis, lost data packets to the robot is in “blue” vs the left axis

6.Graphs battery voltage with a yellow line vs the left axis, roboRIO cpu % usage with a red line vs the right axis

7.Time scale for the time axis of the graphs(12s, 1m, 5m)

Printing to Driver Station

With the 2015 update of the Driver Station, a maximum of 10 lines of strings (each allowing for 21 characters) can be manipulated to print to the Driver Station console. Note that the DriverStationLCD Class was removed entirely, and now printing is done through the Smartdashboard.



Unlike the DiverStationLCD print console, it is possible to type directly into the string fields as well as read these strings within your robot code. Some teams may find this useful when testing their robot code.

Strings can be sent to the Smartdashboard print console using the following code:

```
SmartDashboard::PutString("DB/String 0", "This is a string");
SmartDashboard::PutString("DB/String 1", "This is another string");
```

`SmartDashBoard::PutString` is calling the Smartdashboard and allows you to send strings to the Driver Station printing console. Each line is assigned a name from DB/String 0 to DB/String9 from top to bottom then left to right. The second set of quotations can be manipulated to send various strings.

Strings that are on or were sent to the Smartdashboard can then be retrieved using the following code:

```
std::string dashData = SmartDashboard::GetString("DB/String 0", "myDefaultData");
std::string dashData = SmartDashboard::GetString("DB/String 1", "myDefaultData");
```

Here we are creating a string within the code called dashData that is set to the string that was in the first and second line on the SmartDashboard printer console.

A Crash Course on C++

Variables

- Defining Variables
- Instantiating
- Constants

Functions

- Arithmetic Functions
- Relational Operators
- Logical Operators

Object Usage

- Defining Objects
- Instantiating Objects
- Using Methods

The Joystick

- Sample Code
- Explanation

Variables

A variable is a location in the computer's memory which allows you to store a value which can later be received. Variables are assigned names, which allow you to quickly and easily access the location in memory where the variable's data is stored without having to know the actual memory address.

When you define a variable, you must also declare the data type of that variable. The datatype of the variable determines what kind of data the variable is holding as well as how much memory must be set aside for that variable. An example of a data type would be an integer which is declared with the 'int' command. Integer variables store whole number values.

Different data types require a different amount of memory, but since the use of C++ in this competition does not require you to know the details to data type memory it will not be reviewed in this section.

The tables to follow were adapted from *Sams Teach Yourself C++ in 24 Hours*.

VARIABLE TYPES	VALUES
unsigned short integer	0 to 65,535
short integer	-32,768 to 32,767
unsigned long integer	0 to 4,294,967,295
long integer	-2,147,483,648 to 2,147,483,647
integer	-2,147,483,648 to 2,147,483,647
unsigned integer	0 to 4,294,967,295
long long	-9.2 quintillion to 9.2 quintillion
char	256 character values
boolean	true or false
float	1.2e-38 to 3.4e38
double	2.2e-308 to 1.8e308

➠ Defining Variables

Defining a variable is very easy. Generally, the format for defining a variable is the data type followed by the name you want to assign your variable followed by a semicolon.

```
int theExample;
bool iAmAmazing;
```

You must remember that there are rules and guidelines to naming variables. Among programmers, there is proper naming etiquette which allows someone who is reading your code to easily understand why you named a variable a certain way. Here is a list of rules and guidelines to follow to successfully name a variable:

Rules:

1. Variable names can be made with any combination of letters
2. Variable names cannot contain spaces, symbols, or punctuation marks
3. Variable names may include underscores
4. Variable names cannot begin with a number but may contain a number elsewhere in the name

5. Variable names cannot be the same as reserved keywords. See below table for a complete listing of reserved words.

6. C++ is case sensitive so keep this in mind while naming variables (int myInt refers to a different location than int MyInt or int myint)

Guidelines:

1. Constants are usually written in all caps

2. Variables are usually started with lower case and if a name contains more than one word, the first letter of the next word is capitalized (eg: int thisIsAnExample)

3. Variables should be named something the describes what the variable is going to be used for

4. Avoid giving variables long names, it is okay to abbreviate long words

Reserved Words (C++)

```
alignas (since C++11)
alignof (since C++11)
and
and_eq
asm
auto(changed in c++11)
bitand
bitor
bool
break
case
catch
char
char16_t (since C++11)
char32_t (since C++11)
class
compl
const
constexpr (since C++11)
const_cast
continue
decltype (since C++11)
default (changed in C++11)
delete (changed in C++11)
do
double
dynamic_cast
else
enum
explicit
export(1)
extern
false
float
for
friend
goto
if
inline
int
long
mutable
namespace
new
noexcept (since C++11)
not
not_eq
nullptr (since C++11)
operator
or
or_eq
private
protected
public
register
reinterpret_cast
```

```
return
short
signed
sizeof
static
static_assert (since C++11)
static_cast
struct
switch
template
this
thread_local (since C++11)
throw
true
try
typedef
typeid
typename
union
unsigned
using(1)
virtual
void
volatile
wchar_t
while
xor
xor_eq
```

Adapted from cppreference.com

Below is an example of how to define variables in the C++ language:

```
main()
{
     int a;
     int ohMyLord;
     char myChar;
}
```

To define more than one variable of the same data type, you can use the comma punctuation.

```
main()
{
     int a, b, c;
     char ohMahGlob, lumpySpacePrincess, partyTime;
     bool downLieToMe, cats;
}
```

➠ Instantiating

Instantiating a variable is also known as assigning a variable with a value. The operator used to assign values to a variable is the equals sign, (=).

The variable that the data is being assigned to is always on the left side, while the data that is being assigned to the variable is on the right. It is important to not mix this up because variables can also be assigned values from similar variable types. This is common while using math functions with other variables.

```
main()
{
     int a = 5;
     int b = 10;
     int c = a;
     char character = 'A';
     bool counter = true;
}
```

➠ Constants

Sometimes in programming, we have variables with data that we do not want to change at all. Although you could easily just not change the variable data, it is safer to declare a constant variable. A constant variable is a variable that cannot be changed once it is instantiated.

```
main()
{
     const int THIS_IS_A_CONSTANT = 100;
}
```

Functions

In C++, there are basically three types of functions: arithmetic functions, relational functions, and logical functions.

➠ Arithmetic Functions

These types of functions are your basic math functions and are used in conjunction with numeric data values and variables (int, double, float). It is important to remember that order of operations does apply to these functions.

You should be able to recognize these functions as you have used them since elementary school. The only one that you may not be familiar with is the modulus function, which divides two numbers together and returns the remainder. This function is commonly used in true-false statements or loops to recognize when a numerical variable being analyzed is positive or negative or a multiple of a certain number. Parentheses are used like in algebra to say which operations should be done first if those actions differ from the default order of operations.

In programming, it is also quite often to use variables in conjunction with arithmetic functions. To use variables in this way, all you need to do is use the variable's defined name where numbers would usually be used. It's the same as basic algebra.

Sign	Meaning
+	Addition
-	Subtraction
/	Division
*	Multiplication
%	Modulus
=	Assignment Operator
(operator here)	Parenthesis (order of operation)
++/--	Increment/decrement numeric value by 1

```
main()
{
      int a = 5;
      int b = 10;
      int c = a + b;
      int d = c * a;
      int e = a(3 + b);
      int f = b % 2;
      a++;
}
```

`int a` is assigned the value 5 and `int b` is assigned the value 10. `int c`, is the sum of a and b. Because `a` and `b` are assigned the value 5 and 10, respectively, `c` is currently at `a` value 15. It can be subject to change in the event the values `a` and b change. `int d` is the product of `c` and `a`, (15*5). int `e` is the product of `a` and the sum of `3+b`. `int f` is the remainder of `b` divided by 2 (it will return `a` value of either 0 or 1).

➠ Relational Operators

Relational operators are used to determine whether two numbers are equal, or whether one is greater or less than the other. Every relational expression returns either 1(true) or 0 (false). These operators are used in statements (if, else, while) in order to create expressions that set conditions for that code inside the statement. Be sure not to confuse the assignment operator (=) with the relation operator of equality (==).

Name	Operator	Sample	Evaluation

Equals	==	`100==50;`	false
		`50==50;`	true
Not Equals	!=	`100!=50;`	true
		`50!=50;`	false
Greater than	>	`100>50;`	true
		`50>50;`	false
Greater than or equals	>=	`100>=40;`	true
		`50>=50;`	true
Less than	<	`100<50`	false
		`50<50;`	false
Less than or equals	<=	`100<=50`	false
		`50<=50`	true

➠ Logical Operators

Logical operators are used in conjunction with relational operators to create more complex statement expressions.

OPERATOR	SYMBOL	EXAMPLE
AND	`&&`	2:2
OR	(2 pipes)	2:3
NOT	`!`	2:4

A logical AND statement evaluates two expressions, and if both expressions are true, the logical AND statement is true as well.

A logical OR statement also evaluates two expressions. If either or both are true, the entire expression is true.

A NOT statement compares whether a condition is NOT what is stated (i.e. switches the return value from true to false and vice versa).

When using logical operators to form statements, make sure you use parentheses to make the order of precedence clearer to the compiler to avoid any errors. For example,

```
if( x > 5 && y > 5 || z > 5)
```

This statement goes to show how ambiguous these statements can get if parentheses are not used. Do you want the statement to return true if x > 5 and y>5 or when just z >5, or did you want the statement to return true when x>5 and when either y>5 or z>5? If you wanted the latter, then you should rewrite the statement to make things clearer:

```
if((x > 5 && y > 5) || z > 5)
```

Object Usage

In object-oriented programs (OOP), objects are basically objects like that of the real world, and thus have certain behaviors and characteristics. Characteristics of an object in OOP include any variables or other objects that are declared in the class that the object is written in. Behaviors of an object are methods that the object contains in its class code.

We will use the Jaguar class from the WPI library to help explain this section. Click here to view it.

➠ Defining Objects

Defining objects is a similar to defining variables, but instead the data type is replaced by the name of the object. After the object name comes the name that you assign to the address location of the object. In terms of the Jaguar class, a Jaguar object is defined as:

```
Jaguar jag1;
Jaguar jag2;
Jaguar example;
```

Remember that each Jaguar object that you create is a different instance of the object and that different spaces in memory are created for instance variables of each Jaguar object. The name you gave to a Jaguar object is the only thing you have to distinguish between Jaguars.

➠ Instantiating Objects

Instantiating an object is quite different from defining a variable. When instantiating an object, you must use the `new` operator. Followed by the `new` operator is the object's constructor signature. An object's constructor signature may look like this:

```
Jaguar (UINT32 channel)
```

As you can see the object name is displayed along with some code inside parentheses. The code inside the parentheses is called a parameter. Parameters are where you pass data into the object so that the object can use it. Notice that data in parameters are separated by commas. In the above case, the object constructor asks you to input a float and a channel number. So an example instantiation would look something like the following:

```
jag1(1)
```

Where `1` is the number of a `UINT32 channel` (the PWM OUT port the jaguar occupies on the digital sidecar).

Object classes often have multiple constructors which ask for different sets of parameters. In the case of the Solenoid class, here are the following constructors for a Solenoid object.

```
Solenoid (uint32_t channel) //Constructor using the default PCM ID
Solenoid (uint8_t moduleNumber, uint32_t channel) //Constructor that specifies the PCM ID.
```

You should see that the only thing different is the parameters that must be called. The first calls for the PCM channel only (assumes the PCM ID as `0`). To call the first constructor after defining `sol1`, you would type:

```
sol1(1)
```

which means `sol1` is assigned to PWM port `1` on the PCM and is automatically assumed to be on the PCM ID of `0`.

Because the roboRIO can support more than one PCM, anything connected a PCM with an ID that is not `0` will need to use the second constructor.

➠ Using Methods

In programming, objects alone do no more than they do in reality. Simply stating “the Jaguar exists!” will not make your robot move, nor will merely declaring it in your code make anything happen.

The real work is done through methods, a unique set of commands that each object has. These are noted by a `.` after the name of a particular object and are always followed by `()`, though often there are some values passed into the parentheses. In the Jaguar class, for example, one might see the following code fragment:

```
jag1.Set(0.5);
```

The name of the object is `jag1`, which has previously been declared to be a Jaguar. The method is `Set`, which, as one might expect, sets the speed of the motor. The `0.5` inside the parentheses is called the parameter, which varies depending on the method used. Here, it is a float value from `-1` to `1`, inclusive, but often you must pass an int, bool, or even another object as a parameter. For object-specific information on methods and parameters, see the section on that particular object or use the WPI Library.

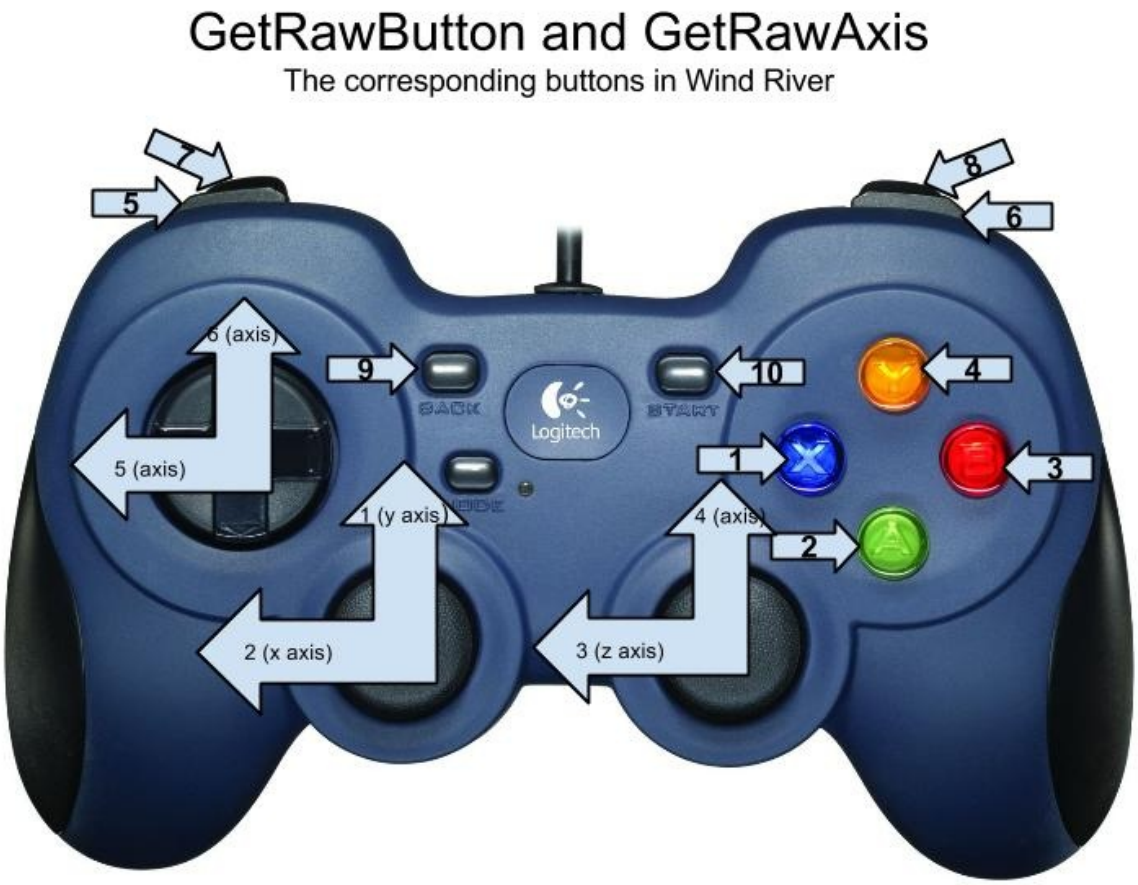
Note that just as each object will have multiple methods, different objects can have methods of the same name and may or may not do different things. The Victor class, for example, also has a Set method that functions exactly the same, but the Relay class takes an entirely different data type and functions purely as an on/off switch.

The Joystick

Logitech Gamepad F310

Logitech has made a home-touch-feely controller as it appears to be a standard controller. Most people have played video games with a controller like this, so there's nothing new to learn about it. On the back is a little slide button, make sure it is set to the right and tape it like that to prevent incidents of bad joystick. Make sure mode light is off.

AndyMark



Logitech Attack 3 USB Joystick

This Attack 3 Joystick looks awesome, doesn't it exude the feeling of robotics? One hand to move the single joystick, the other to press the buttons on the bottom. The joystick class in the WPI Library does support this joystick and all if its many inputs.

FIRST Choice

Joystick Class (C++)

➠ Sample Code

```
#include "WPILib.h"
class Robot: public SampleRobot
{
    Joystick stick;
public:
    Robot() :
        stick(0) // Use joystick on port 0.
    {
    }
    void OperatorControl()
    {
        while(IsOperatorControl())
        {
            if(stick.GetRawAxis(1) > .2)
            {
            }
            if(stick.GetRawAxis(2) > .2)
            {
            }
            if(stick.GetRawButton(1) == 1)
            {
            }
            if(stick.GetRawButton(4) == 1)
            {
            }
            if(stick.GetTop() == 1)
            {
                if(stick.GetRawAxis(3))
                {
                }
                if(stick.GetRawAxis(4))
                {
                }
            }
        }
    }
};
```

```
START_ROBOT_CLASS(Robot);
```

Explanation

```
Joystick stick;
```

Declare one `Joystick` object. Declared between `class RobotDemo : public SampleRobot` and `public : RobotDemo(void):`

```
stick(0);
```

Instantiate one `Joystick` object in USB port of computer (limited to # of USB ports on computer). Instantiation occurs between the `public : RobotDemo(void):` and the braces(`{ }`). If not the last instantiated object in list, it needs a comma after instantiation statement like listing. If it is, it does not need any punctuation after the instantiation before the braces; no comma, no semicolon, no period, etc. If syntax not followed, error occurs.

```
void OperatorControl()
{
    while(IsOperatorControl())
    {
        if(stick.GetRawAxis(1) > .2)
        {
        }
        if(stick.GetRawAxis(2) > .2)
        {
        }
        if(stick.GetRawButton(1) == 1)
        {
        }
        if(stick.GetRawButton(4) == 1)
        {
        }
        if(stick.GetTop() == 1)
        {
            if(stick.GetRawAxis(3))
            {
            }
            if(stick.GetRawAxis(4))
            {
            }
        }
    }
}
```

Joystick functions are the sauce for conditions in `OperatorControl`. By doing certain actions on the joystick object (this instance is using a Logitech F310 Gamepad), it executes the code that would be written in the braces. For example, `GetRawAxis(1)` corresponds to the left stick y-axis(up and down) of the F310 Gamepad or the y-axis of the Extreme 3D Pro joystick; the | |(or) corresponds to positive(up) or negative(down) input. The axis is usually associated with driving the robot. `GetRawButton()` only returns 1 if it is being pressed; `GetRawButton(1)` is the x-button on the F310 Gamepad or the trigger of the Extreme 3D Pro joystick. `GetTop` is the smaller stick on the Extreme 3D Pro, and it only returns `1` if top is being used or `0` if not, so extra conditions for axis 3 & 4, y-axis and x-axis of top respectively. For the F310 Gamepad it would be axis 6 & 5, y-axis and x-axis respectively.

Motor Controllers

General Overview

Motor controllers are what they sound like; they allow us to control the amount of power sent to the motor. They serve as the middlemen from the PDB to the motor itself.

Motors

CIM motor

Physical Specs

- Size: 2.5 inch diameter, 4.34 inch long body
- Output Shaft size: 0.313 +/- 0.0004, with 2mm keyway
- Weight: 2.82 pounds
- Mounting Holes: #10-32 tapped holes (2), on a 2" bolt circle

Performance

- Voltage: 12 volt DC
- No load RPM: 5,310 (+/- 10%)
- Free Current: 2.7 amps Maximum Power: 337 Watts (at 2655 rpm, 172 oz-in, and 68 amps)
- Stall Torque: 2.42 N-m, or 343.4 oz-in
- Stall Current: 133 amps

AndyMark

mini-CIM motor

⅔ power of CIM, similar form factor and same mounting

Physical Specs

- Output Shaft size:8mm (0.314in) with 2mm keyway

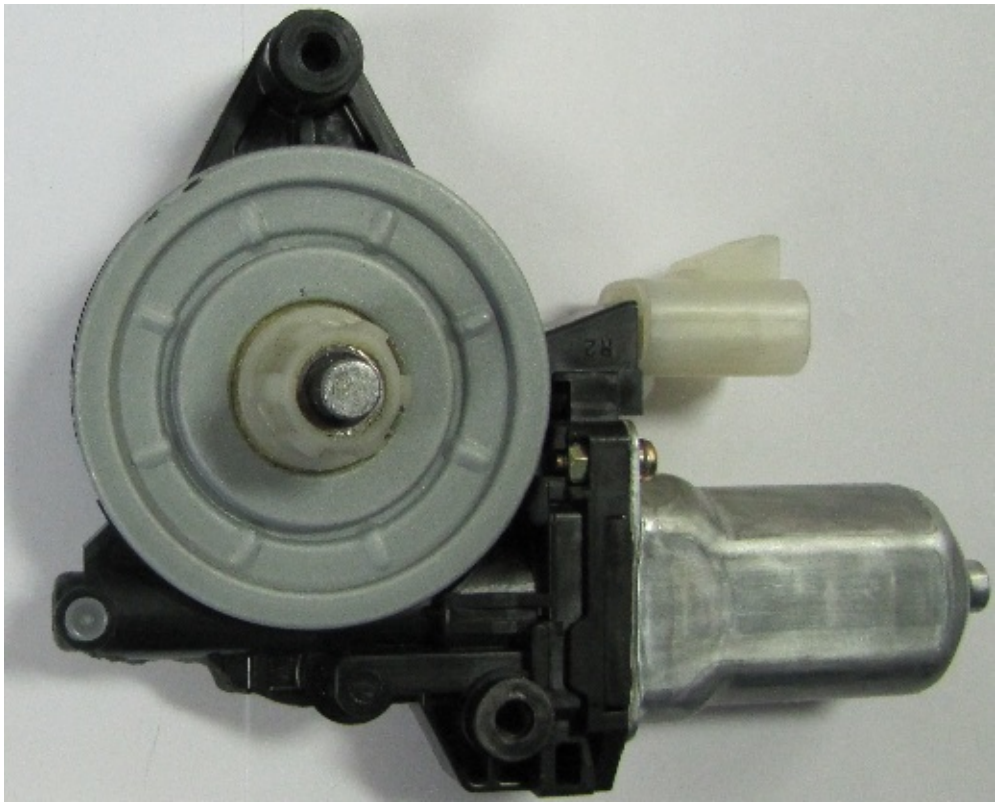
- Size: 2.5” diameter, 3.36” long
- Weight: 2.16 lbs

Performance

- Free Speed:6,200 rpm (+/- 10%)
- Free Current:1.5A
- Maximum Power:230 W
- Stall Torque:12.4 in-lbs [1.4 N-m]
- Stall Current:86A
- Mounting Holes:(4) #10-32 tapped holes on a 2" bolt circle

Vex

Window Motor



Make sure you remove the locking pins.

- Stall Torque: 9.3 Nm
- Free Speed: 92 RPM
- Free Current: 2.5 A
- Stall Current: 25 A

Servo

Should **not** be hooked up to **ANY** motor controllers and directly to the PWM port in the roboRIO.

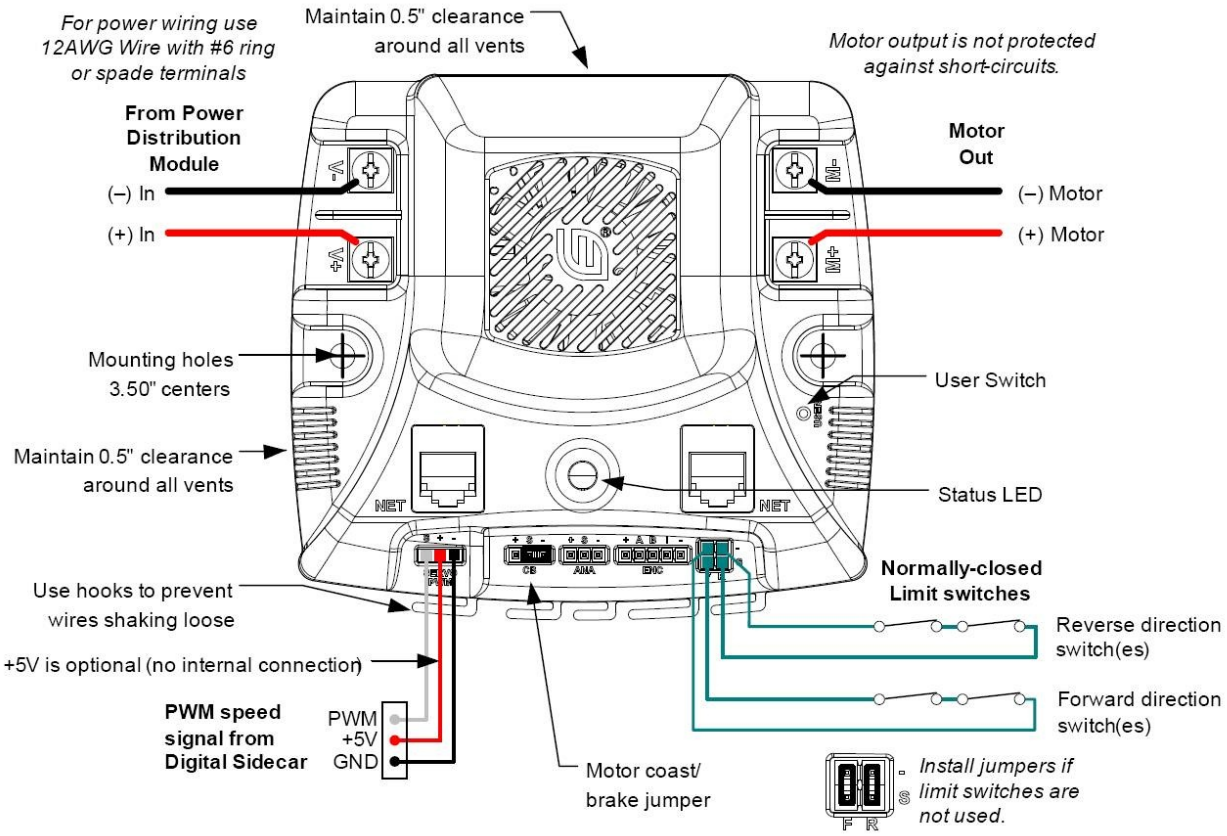
AndyMark

➠ Motor Controller Varieties

It is possible to control all the motors above (except the servo) with the below motor controllers; however, the breakers used have to be able to protect the wires and provide enough power for the motor used. For example, it is possible to connect 16 gauge wire with a 20 amp CB to a talon, but would not provide enough power if connected to say a CIM. There are two side wires that connect to each motor controller: the M-/M+ and the V-/V+ side. The ‘M’ stands for Motor, which denotes the wires attached here should be the ones also attached to the motor. The other side connects to the PDB. In both cases, the power goes to the + and ground to the -. There is also a thin slot where PWM cables plug into from the Digital Sidecar, with its direction based on the small notations on the motor controllers (often, ground is facing the side marked with a ‘B’).

Jaguar

There are jumpers that can be used in two places, the motor coast/brake, and the Limit Switches. The motor coast/brake controls if after the robot stops it slowly decelerates (coasts), or immediately decelerates (brakes). The jumpers are to be installed in the limit switch area if there are no limit switches being used. Jaguars use the CAN network folder. The status LED indicates many things like operation, fault, calibration, and other conditions using yellow, red, and green lights.



LED State	Module Status
Solid Yellow	Neutral
Fast Blinking Green	Forward
Fast Blinking Red	Reverse
Solid Green	Max Speed Forward
Solid Red	Max Speed Reverse
Slow Blinking Yellow	Loss of Servo or Network Link
Fast Blinking Yellow	Invalid CAN ID
Slow Blinking Red	Voltage, Temperature, or Limit Switch fault condition
Slow Blinking Red and Yellow	Current Fault Condition
Fast Blinking Red and Green	Calibration Mode Active
Fast Blinking Red and Yellow	Calibration Mode Failure
Slow Blinking Green and Yellow	Calibration Mode Success
Slow Blinking Red and Green	Calibration Mode Reset to Factory Default Success
Slow Blinking Green	Waiting in CAN assignment mode

➠ Sample Code

Jaguar Class (C++)

```
#include "WPILib.h"

class RobotDemo : public SampleRobot
{
    Jaguar jaguar;
    Joystick stick;

public:
    RobotDemo(void):
        jaguar(1),
        stick(1)
        {
        }

    void OperatorControl()
    {
        if(stick.GetRawButton(1))
        {
            jaguar.Set(1.0);
        }
        else if(stick.GetRawButton(2))
        {
            jaguar.Set(-1.0);
        }
        else
        {
            jaguar.Set(0);
        }
    }
```

```
};

START_ROBOT_CLASS(RobotDemo);
```

➠ Explanation

```
Jaguar jaguar;
```

Declare Jaguar motor controller as `jaguar` ; declared between `public SampleRobot` and `public : RobotDemo`

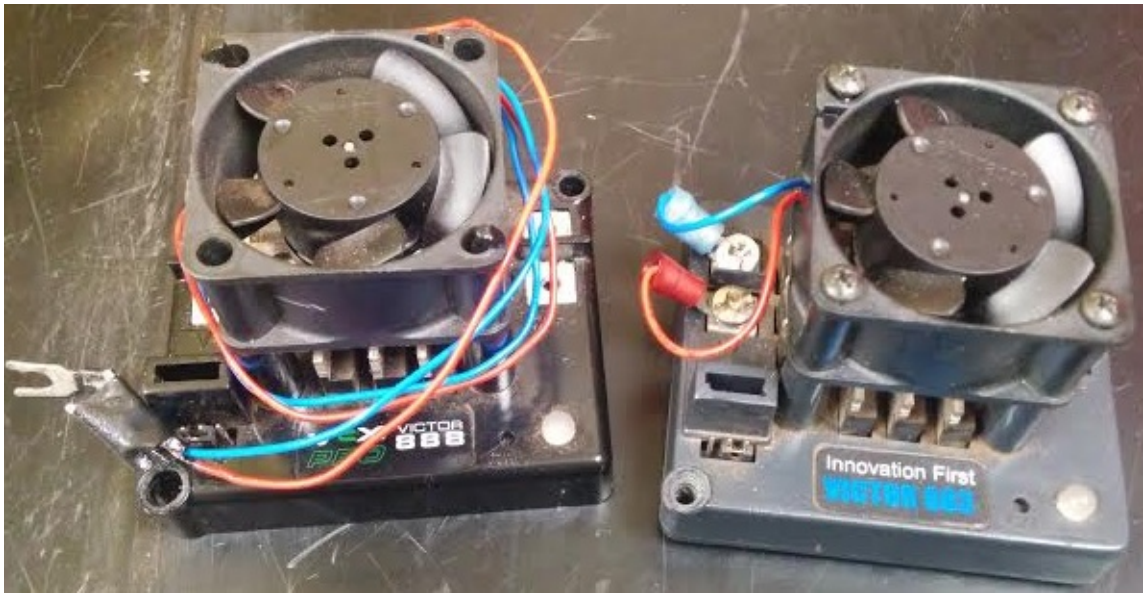
```
jaguar(1),
```

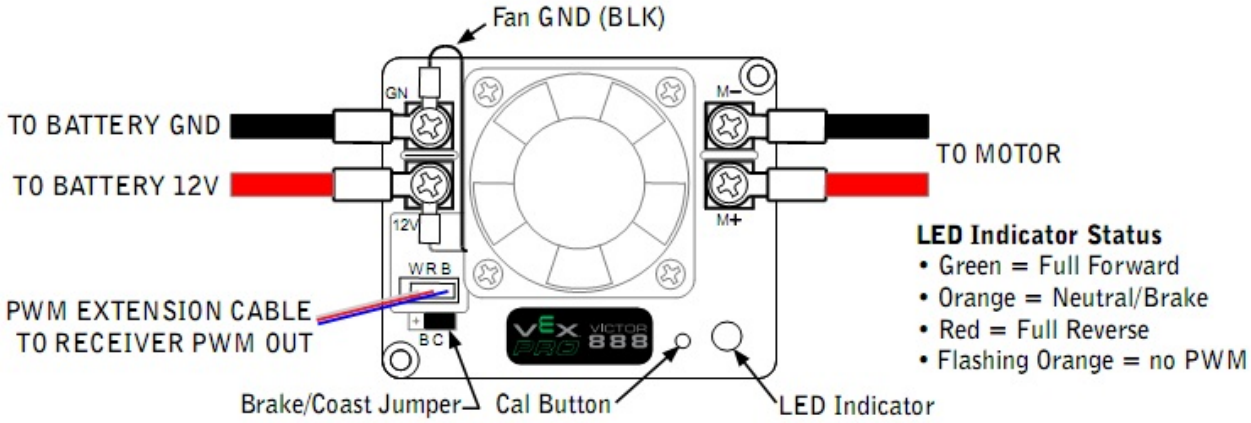
Initialize Jaguar motor controller as port # 1 in Digital Sidecar (PWM Out), initialized between `public : RobotDemo` and the braces({ }). If it is not the last object initialized, it needs a comma like a list. If it is the last object initialized, no punctuation; no comma, no semicolon, no period, etc. or you will get an error.

```
void OperatorControl()
{
    if(stick.GetRawButton(1))
    {
        jaguar.Set(1.0);
    }
    else if(stick.GetRawButton(2))
    {
        jaguar.Set(-1.0);
    }
    else
    {
        jaguar.Set(0);
    }
}
```

Joystick class is gone into depth in an earlier section of this manual. Motor controllers are put into results of conditions because a free-spinning motor is a waste of power and there is no control over the motor(which is why it is a motor controller) The `.Set` method of the class accepts a float between -1.0 to 1.0 as a parameter and sets the speed of the motor to that float. 1.0 is full speed “forward”, -1.0 is full speed “backward.” The motor when initialized begins at `.Set(0)`. The `else jaguar.Set(0)` is to stop the motor because unless the motor controller is set to 0, the motor remains at the last `.Set()` value.

Victor 888





The victor is similar to the jaguar, but sacrifices computing power for a lighter weight and a smaller size.

LED STATUS	CONDITION
Solid Green	positive output voltage equal to the input voltage
Solid Red	positive output voltage equal to the input voltage multiplied by -1
Blinking Orange	victor is disabled (PWM is not connected/not working or the robot is disabled)
Flashing Red	Indicates that victor has had unsuccessful calibration
Flashing Green	Indicates that victor has had successful calibration

When wiring, make sure that the PWM is plugged in so that the black wire is facing the inside (towards the fan). Pay special attention to the M+ M- V+ and V- on the sides of the Victor when wiring it to the motor and the power distribution board.

Victor 888 User Manual

Victor 888 - VEX Store

➠ Sample Code

Victor Class (C++)

```
#include "WPILib.h"

class RobotDemo : public SampleRobot
{
    Victor victor;
    Joystick stick;
public:
    RobotDemo(void):
        victor(1),
        stick(1)
        {
        }

    void OperatorControl()
    {
        if(stick.GetRawButton(1))
        {
            victor.Set(1.0);
        }
        else if(stick.GetRawButton(2))
        {
            victor.Set(-1.0);
        }
        else
        {
            victor.Set(0);
        }
    }
};

START_ROBOT_CLASS(RobotDemo);
```

➠ Explanation

```
Victor victor;
```

Declare victor motor controller as `victor` ; declared between `public SampleRobot` and `public : RobotDemo`

`victor(1),`

Initialize victor motor controller as port # 1 in Digital sidecar PWM Out. This is stated between `public : RobotDemo` and the braces(`{ }`). If it is not the last object initialized, it needs a comma like a list. If it is the last object initialized, no punctuation; no comma, no semicolon, no period, etc. or you will get an error.

```
void OperatorControl()
{
    if(stick.GetRawButton(1))
    {
        victor.Set(1.0);
    }
    else if(stick.GetRawButton(2))
    {
        victor.Set(-1.0);
    }
    else
    {
        victor.Set(0);
    }
}
```

Joystick class is gone into depth in an earlier section of this manual. Motor controllers are put into results of conditions

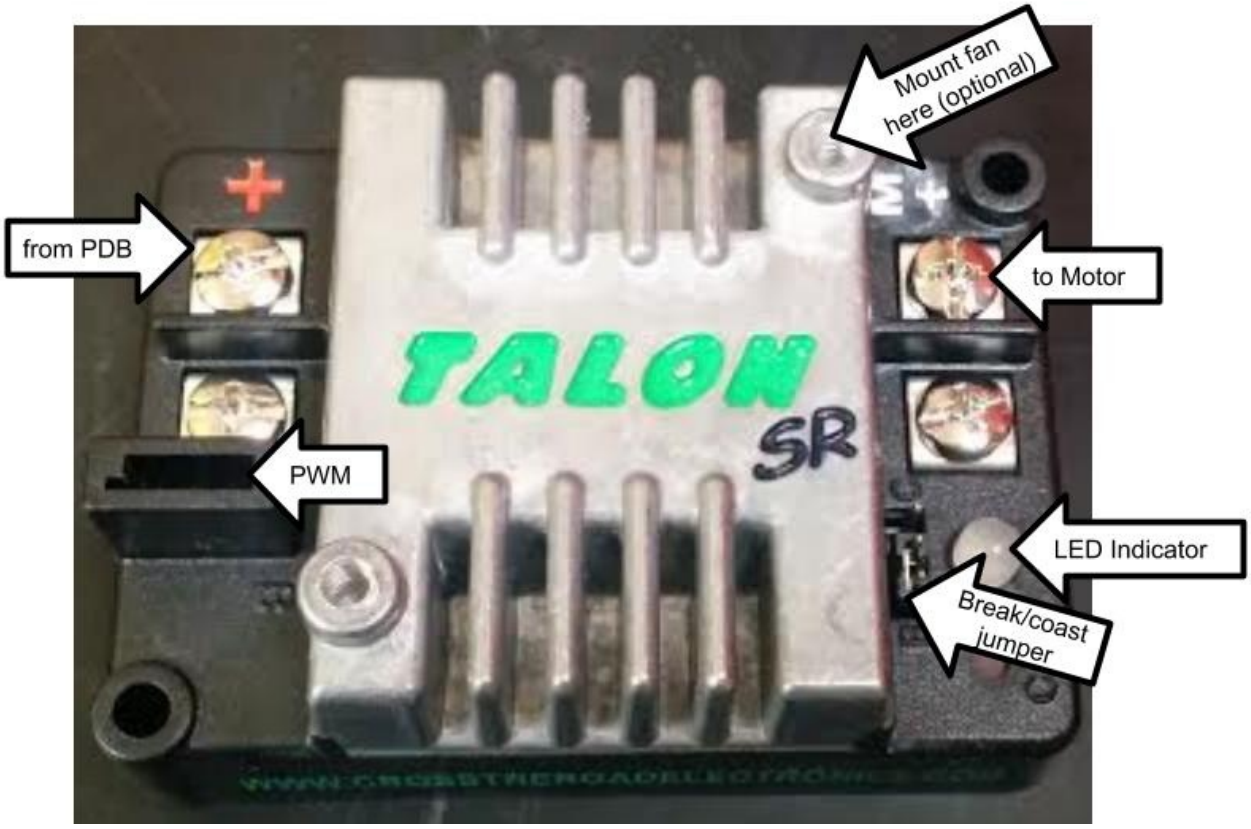
because a free-spinning motor is a waste of power and there is no control over the motor (which is why it is a motor controller) The `.Set` method of the class accepts a float between `-1.0` to `1.0` as a parameter which sets the speed of the motor to that float. `1.0` is full speed “forward”, `-1.0` is full speed “backward.” The motor when initialized begins at `.Set(0)`. The else `victor.Set(0)` is to stop the motor; unless the motor controller is set to `0`, the motor remains at the last `.Set()` value.

NOTE: The 883, 884 and 885 models have been discontinued, but the manufacturer’s documentation can be found below.

Victor 883/885 User Manual

Victor 884 User Manual

Talon



The talon is interchangeable with the jaguar. It has a peak output of 100A and 60A continuous current. There are mounting holes for an optional 40mm fan. The LED on the talon is a status indicator.

LED STATUS	CONDITION
Solid Green	positive output voltage equal to the input voltage
Solid Red	positive output voltage equal to the input voltage multiplied by -1
Blinking Orange	talon is disabled (PWM is not connected/not working or the robot is disabled)
Flashing Red	Indicates that talon has had unsuccessful calibration
Flashing Green	Indicates that talon has had successful calibration

Talon User Manual

➠ Sample Code

Talon Class (C++)

```
#include "WPILib.h"

class RobotDemo : public SampleRobot
{
    Talon talon;
```

```
    Joystick stick;
public:
    RobotDemo(void):
        talon(1),
        stick(1)
        {
        }

    void OperatorControl()
    {
        if(stick.GetRawButton(1))
        {
            talon.Set(1.0);
        }
        else if(stick.GetRawButton(2))
        {
            talon.Set(-1.0);
        }
        else
        {
            talon.Set(0);
        }
    }
};

START_ROBOT_CLASS(RobotDemo);
```

➠ Explanation

```
Talon talon;
```

Declare talon motor controller as `talon`; declared between `class : SampleRobot` and `public : RobotDemo`.

```
talon(1),
```

Initialize talon motor controller as connected to port #1 in the Digital Sidecar (PWM Out); initialized between `public : RobotDemo` and the braces(`{ }`). If it is not the last object initialized, it needs a comma like a list. If it is the last object initialized, no punctuation; no comma, no semicolon, no period, etc. or you will get an error.

```
void OperatorControl()
{
    if(stick.GetRawButton(1))
    {
        talon.Set(1.0);
    }
    else if(stick.GetRawButton(2))
    {
        talon.Set(-1.0);
    }
    else
    {
        talon.Set(0);
    }
}
```

Joystick class is gone into depth in an earlier section of this manual. Motor controllers are put into results of conditions because a free-spinning motor is a waste of power and there is no control over the motor (which is why it is a motor controller) The .Set method of the class accepts a float between `-1.0` to `1.0` as a parameter which sets the speed of the motor to that float. `1.0` is full speed “forward”, `-1.0` is full speed “backward.” The motor when initialized begins at `.Set(0)`. The else `talon.Set(0)` is to stop the motor; unless the motor controller is set to `0`, the motor remains at the last `.Set()` value.

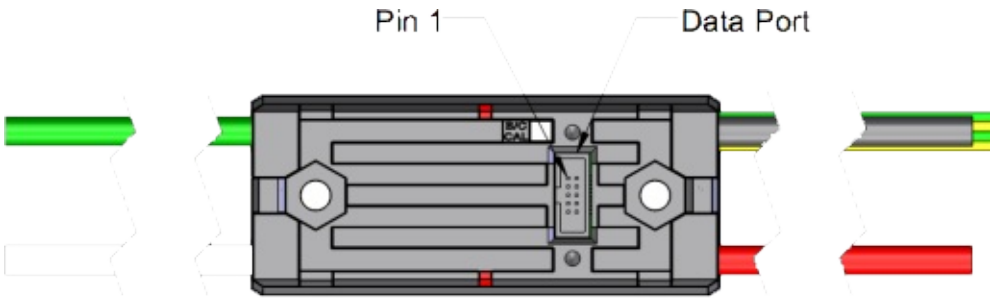
Talon SRX

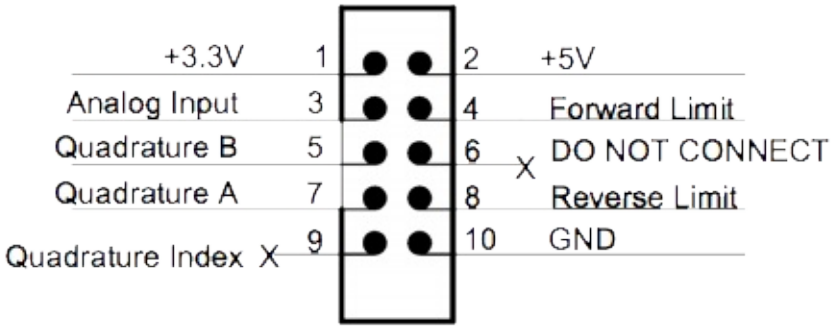
The Talon SRX is a new iteration of the Talon motor controller series that was introduced in the 2015 FRC season. The SRX is unique as it is CAN enabled and capable of operating with the roboRIO, PCM, and VRM, which all use CAN protocols. Because the Talon SRX was designed without a built-in ventilation system, you should mount it in an area with adequate airflow. The user guide recommends mounting it to the robot's metal frame because it will act like a giant heatsink.

Specs

- Dimensions: 2.75" x 1.85" x .96" tall
- Weight: .2lbs including wires
- 15 kHz output switching frequency
- 60 Amp Continuous current, 100 Amp
- 2 x Mounting Holes (one at each end, 6-32 fasteners)
- Supports CAN (Controller Area Network), SPI (Serial Peripheral Interface), Digital I/O, and USART (Universal Synchronous/Asynchronous Receiver/Transmitter)

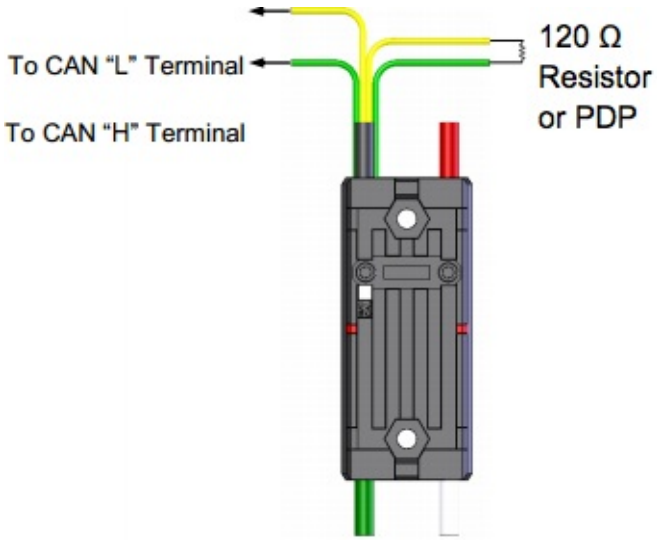
➠ Wiring





Data Port Pinout

➠ **Can**



LED STATUS	CONDITION
Flashing Green & Red	During calibration - Calibration mode
Flashing Green	During calibration - Successful calibration
Flashing Red	During calibration - Faile calibration
Both Flashing Green	Forward throttle is applied
Both Flashing Red	Reverse throttle is applied
Flashing Orange	CAN bus detected, robot disabled
Flashing Red (slow)	CAN bus / PWM not detected
Flashing Red (fast)	Fault detected
Flashing Red & Orange	Damaged hardware
Solid Red	Brake mode
Off	Coast mode

Talon SRX User Manual

➠ Sample Code

TalonSRX Class (C++)

```
#include "WPILib.h"

class RobotDemo : public SampleRobot
{
    TalonSRX talonsrx;
    Joystick stick;
public:
    RobotDemo(void):
        talonsrx(1),
        stick(1)
        {
        }
    void OperatorControl()
    {
        if(stick.GetRawButton(1))
        {
            talonsrx.Set(1.0);
        }
        else if(stick.GetRawButton(2))
        {
            talonsrx.Set(-1.0);
        }
        else
        {
            talonsrx.Set(0);
        }
    }
};
START_ROBOT_CLASS(RobotDemo);
```

➠ Explanation

```
TalonSRX talonsrx;
```

Declare Talon SRX motor controller as `talon`; declared between public SampleRobot and public : RobotDemo

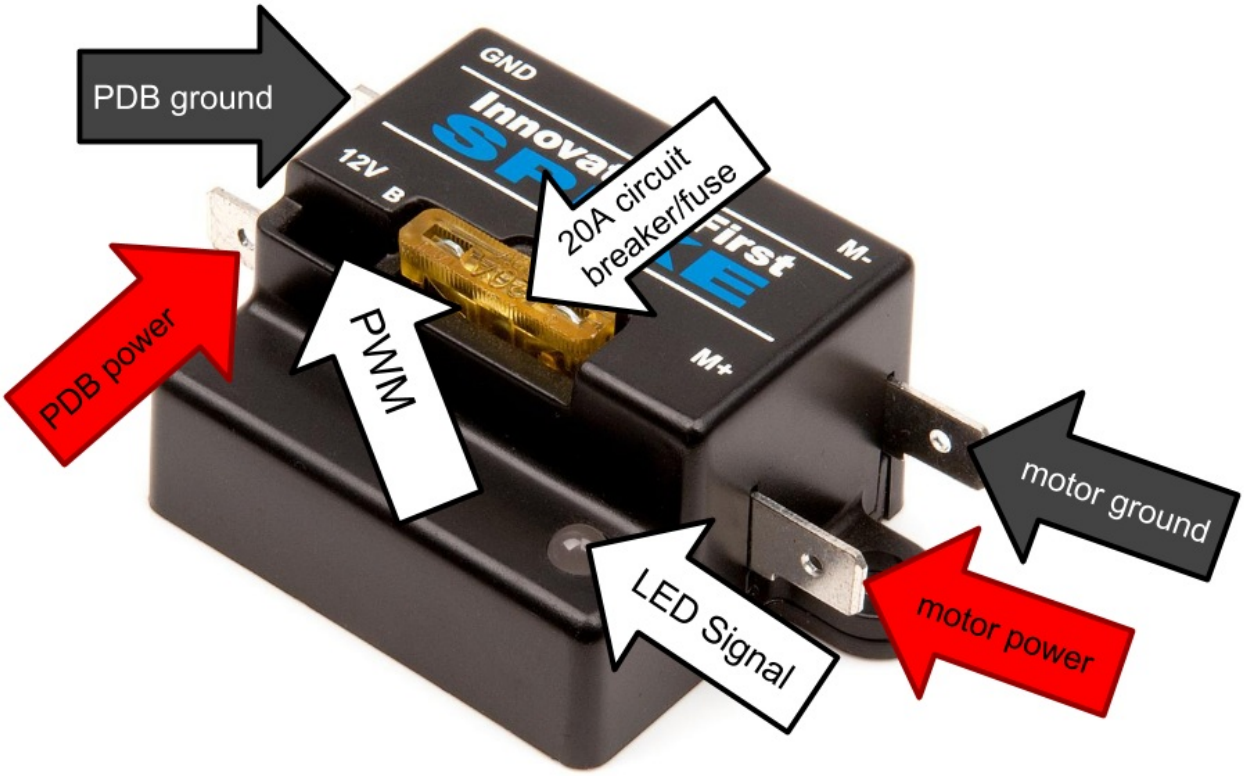
```
talonsrx(1),
```

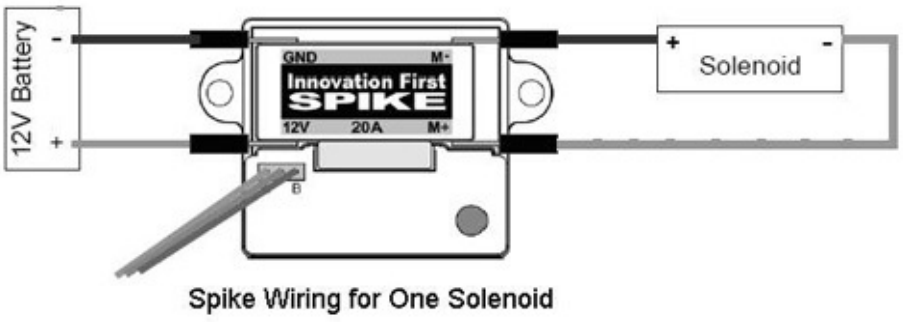
Initialize talon SRX motor controller as connected to port #1 in the Digital Sidecar (PWM Out); initialized between `public : RobotDemo` and the braces(`{ }`). If it is not the last object initialized, it needs a comma like a list. If it is the last object initialized, no punctuation; no comma, no semicolon, no period, etc. or you will get an error.

```
void OperatorControl()
{
    if(stick.GetRawButton(1))
    {
        talonsrx.Set(1.0);
    }
    else if(stick.GetRawButton(2))
    {
        talonsrx.Set(-1.0);
    }
    else
    {
        talonsrx.Set(0);
    }
}
```

Joystick class is gone into depth in an earlier section of this manual. Motor controllers are put into results of conditions because a free-spinning motor is a waste of power and there is no control over the motor (which is why it is a motor controller) The .Set method of the class accepts a float between `-1.0` to `1.0` as a parameter which sets the speed of the motor to that float. `1.0` is full speed “forward”, `-1.0` is full speed “backward.” The motor when initialized begins at `.Set(0)`. The else `talonsrx.Set(0)` is to stop the motor; unless the motor controller is set to `0`, the motor remains at the last `.Set()` value.

Spike





Spike Wiring for One Solenoid

B indicates that the ground side of the PWM faces inward

Spikes are motor controllers used in driving small motors in forward, reverse, or stop (brake). It uses a **20A** circuit breaker. It can also be wired to compressors and solenoids and its indicator lights are different for motors and solenoids, as shown in the table below.

LED STATUS	CONDITION (MOTOR)	CONDITION (SOLENOID)
Orange	OFF / Brake Condition (default)	Both Solenoids OFF (default)
Green	Motor rotates in one direction	Solenoid connected to M+ is ON
Red	Motor rotates in opposite direction	Solenoid connected to M- is ON
OFF	OFF / Brake Condition	Both Solenoids ON

Spike User Manual

➠ Sample Code

Relay Class (C++)

```
#include "WPILib.h"

class RobotDemo : public SampleRobot
{
    Relay spikeblue;
    Joystick stick;

public:
    RobotDemo():
        spikeblue(1,Relay::kForward),
        stick(1)
    {
    }

    void Autonomous()
    {
        spikeblue.Set(Relay::kOn);
    }

    void OperatorControl()
    {
        while (IsOperatorControl())
        {
            if(stick.GetRawButton(1))
            {
                spikeblue.Set(Relay::kOn);
            }
            else
            {
                spikeblue.Set(Relay::kOff);
            }
        }
    }
};

START_ROBOT_CLASS(RobotDemo);
```

➠ Explanation

```
Relay spikeblue;
```

Declare spike relay as name `spikeblue`. The declaration occurs between `class RobotDemo : public SampleRobot` and `public : RobotDemo():`

```
spikeblue(1,Relay::kForward),
```

Instantiate the spike relay with the parameters [Digital Sidecar Port#], [direction of current [kForward, kBackward, or

kBothDirections]] This is instantiated between `public : RobotDemo():` and the braces (`{ }`). If it is not the last object initialized, it needs a comma like a list. If it is the last object initialized, no punctuation; no comma, no semicolon, no period, etc. or you will get an error.

```
void Autonomous()
    {
        spikeblue.Set(Relay::kOn);
    }

void OperatorControl()
{
    while (IsOperatorControl())
    {
        if(stick.GetRawButton(1))
        {
            spikeblue.Set(Relay::kOn);
        }
        else
        {
            spikeblue.Set(Relay::kOff);
        }
    }
}
```

The Joystick class has already been covered in a previous section of the manual. Controls current to whatever is on the other side of the spike(one side connected to the PDB). In autonomous, if there is something wired to the spike that needs to be turned on, it can be `Set(Relay::kOn)` . To turn it off, use `Set(Relay::kOff)` . Note that it will not shut off automatically and hence manual off command. In `OperatorControl` , the relay will often be inserted inside control statements to prevent loose relay on/off.Usually turning it on if button set in if condition is pressed otherwise relay off. Or vice versa if need be.

Fans

Before the pedantic comment regarding our table of contents, no, fans are not motor controllers. They are, however, secured on top of motor controllers to cool them down; being fans and all. Talons and Victors have mounting holes that require 6-32 inch screws. The terminals connect to the V +/- side on the motor controller (you don’t want your fans turning off and on in unison with your motors).

The D-Link

Box on Wheels

You have now opened up a bare bones template to write your robot code; congratulations you've made a box on wheels, now to understand what you're box on wheels does, before you destroy it and create your own completely improved code. It is necessary to understand the classes that created this box on wheels so you know that you just didn't create another box on wheels program, but it is a good start.

➠ The Code

The drive program in it's entirety:

```
#include "WPILib.h"
//Last modified: January 19, 2014 by: Alan
/*
 This is a demo program showing the use of the RobotBase class.The SampleRobot class is the base of a robot applicatior
 Autonomous and OperatorControl methods at the right time as controlled by the switches on the driver station or the fi
 */
class RobotDemo : public SampleRobot

{
    RobotDrive myRobot;    // robot drive system
    Joystick stick;    // only joystick

public:
    RobotDemo(void):
        myRobot(1, 2),    // these must be initialized in the same order
        stick(1)          // as they are declared above.
    {
        myRobot.SetExpiration(0.1), //you can initialize things here like gyros at construction
    }

// Drive left & right motors for 2 seconds then stop

    void Autonomous(void)
    {
        myRobot.SetSafetyEnabled(false);
        myRobot.Drive(0.5, 0.5);    // drive forward at half speed
        Wait(2);                  // for 2 seconds
        myRobot.Drive(0.0, 0.0);    // stop robot
    }

// Runs the motors with arcade steering
    void OperatorControl(void)
    {
        myRobot.SetSafetyEnabled(true);
        while (IsOperatorControl())
        {
            myRobot.ArcadeDrive(stick); // drive with arcade style (use
                                right stick)
        Wait(0.005);            // wait for a motor update time
    }
}

// Runs during test mode
    void Test()
    {

    }
};
START_ROBOT_CLASS(RobotDemo);
```

➠ The Explanation

Breakdown of the code follows as so:

```
#include "WPILib.h"
```

This including of the WPI Library is the inclusion of a spellbook for almost every class needed for the robot: motors, pneumatics, Axis cameras, etc. NOTE: There will still be much time spent using the “WPILib C++ Reference”, but the use of the basic manual reduces most of the time spent by presenting sample/proper usage so the learning process does not need to repeat itself and time can be best allocated elsewhere.

```
/*
This is a demo program showing the use of the RobotBase class. The SampleRobot class is the base of a robot applicatior
Autonomous and OperatorControl methods at the right time as controlled by the switches on the driver station or the fie
 */
```

If you’ve commented in programming, then you know what you should be doing, if you are lazy and have some personal belief or dogma that, “Tough beans, figure out my giant program,” you are impeding the progress of your subteam and you are absolutely terrible. However, if you’re just in the mood and have started a program, comment what needs to be understood because you never know when you might be gone the next day and someone else has to run your program.

```
class RobotDemo : public SampleRobot

This is your physical robot, RobotDemo is the name of the class, SampleRobot is the class that RobotDemo inherits it’s

{
    RobotDrive myRobot;    // robot drive system
    Joystick stick;    // only joystick
```

Declaration of the robots parts, `RobotDrive` is a simplified drive system class that declares motors for you and has preprogrammed drive functions; `Joystick` also declared last because it is a part of the robot, but how else are you going to control it? With your mind? I THINK NOT!

```
public:
        RobotDemo(void):
        myRobot(1, 2),    // these must be initialized in the same order
        stick(1)        // as they are declared above.
    {
        myRobot.SetExpiration(0.1), //you can initialize things here like gyros at construction
    }
```

`RobotDemo` is the constructor of your robot, it will now initialize what you declared previously as ports in the sidecar for most of the declared objects, with the exception of the joystick. As noted, `RobotDemo` has the same name as the class because of how objects work, and it is void, but you don’t have to put void as in C++ it automatically assumes void. The pair of braces after you instantiate the ports of your controllers allows you to initialize/run commands (like sensors) at the very beginning.

```
// Drive left & right motors for 2 seconds then stop
    void Autonomous(void)
    {
        myRobot.SetSafetyEnabled(false);
        myRobot.Drive(0.5, 0.5);    // drive forward at half speed
        Wait(2);                // for 2 seconds
        myRobot.Drive(0.0, 0.0);    // stop robot
    }
```

This is the `Autonomous` method of the `RobotDemo` class, and it was inherited from `SampleRobot`. It will only run during the Autonomous period of the game. `SetSafetyEnabled` is to protect everyone in case of loss of communication or other problems when set to true in the `()`. Drive sets the speed of the motors in the order initialized respectively, from a value of (`-1.0` to `1.0`). Wait is a method that stops the program from reading any further lines for the time specified in the `()` in seconds. To stop the robot, the motors after being set to move must be set back to zero.

```
// Runs the motors with arcade steering
    void OperatorControl(void)
```

```
    {
        myRobot.SetSafetyEnabled(true);
        while (IsOperatorControl())
        {
            myRobot.ArcadeDrive(stick); //drive with arcade style (use right stick)
            Wait(0.005);    // wait for a motor update time
    }
}
```

`OperatorControl` is a method of the `RobotDemo` class, this method was also inherited from `SampleRobot`. This is where the code for your tele-op or driver control period goes. `SetSafetyEnabled` was already mentioned, but as a reminder, it's for the safety of all others and the robot in case of communication problems with the robot. The while loop is there to make sure that you are always in control during the period, without the loop, the code would only run once and your robot would then become a stationery box. `ArcadeDrive` is the type of drive using arcade joystick, the robot will move according to the joystick input.

```
// Runs during test mode
    void Test()
    {

    }
```

This is where you would input test code that you wouldn't put into either Autonomous or Tele-op without being sure it would work first or if it would conflict with other parts of the code inside those methods.

```
};
START_ROBOT_CLASS(RobotDemo);
```

`START_ROBOT_CLASS` sets up a "user class factory", which is a function that returns a pointer new instance of your robot class. It also creates the entry point function, `FRC_UserProgram_StartupLibraryInit`.

Box on Wheels Template vs Custom Program

While Box on Wheels Template is already made, there is not a lot of room to edit this drive code unless you are using two Logitech Extreme 3D Pro USB Joysticks. If you search in the WPILib Reference for RobotDrive, the constructors and drive methods are designed for something like the above. Also your choices of # of motors for drive is only either 2 or 4. If you wish to use a Logitech F310 Gamepad, you are better off writing your own drive code. When I mean custom, delete the unnecessary parts of the template that would be not conducive to the word custom. Below is barebones code that you may modify to meet your own desires.

➠ The Code

```
#include "WPILib.h”
//Last modified: January 25, 2014 by: Alan
class RobotDemo : public SampleRobot
{
    Joystick stick;

public:
    RobotDemo(void):
    stick(1)
    {
    }
    // Insert your own comment
    void Autonomous(void)
    {
    }

    // Insert your own comment
    void OperatorControl(void)
    {
        while (IsOperatorControl())
        {
            Wait(0.005);    // wait for a motor update time
        }
    }
    // Insert your own comment
    void Test()
    {
    }
};
START_ROBOT_CLASS(RobotDemo);
```

➠ The Explanation

Here’s a pancake sandwich with sprinkles.

```
#include "WPILib.h"
```

This is your spellbook. Live it, breathe it.

```
class RobotDemo : public SampleRobot
```

The `RobotDemo` class + `SampleRobot` from the template gives you the inherited methods to use in the Driver Station, will need it

```
Joystick stick;
```

You will always need a joystick, again, unless you can control your robot with your mind, or it is completely autonomous, make the joystick.

```
public:
    RobotDemo(void):
        stick(1)
    {
    }
```

Instantiate the stick, you can't use it if you don't instantiate it. The braces have to be there, part of the compiling. When you start adding in motors and sensors, their expirations/initialization will be set in those braces.

```
    void Autonomous(void)
    {
    }
```

Still need autonomous section, part of the inheritance. If you use it or not is up to that year's game, but you still need this!

```
    void OperatorControl(void)
    {
        while (IsOperatorControl())
        {
        Wait(0.005);    // wait for a motor update time
        }
    }
```

This is where you write the god code, where you let your driver feel like a king moving the robot...with your code :D Make sure the code that will let the driver execute commands is in the while loop, wouldn't it suck that they can only do it once because it was not in the loop? Before the loop is when you can instantiate specific things such as the resolution of an axis camera.

```
    void Test()
    {
    }
```

Testing space for small portions of questionable code. Also part of inheritance, required.

```
};
START_ROBOT_CLASS(RobotDemo);
```

Closes brace from the beginning of the class. `START_ROBOT_CLASS(RobotDemo);` notice how `RobotDemo` is in the parameter, isn't that the class? So this is also important for it runs the class.

Custom Program (Tank Drive)

An example of where the driver uses the Logitech F310 Gamepad, but because of the way the RobotDrive class is made, it is preferable to make one's own code.

➠ The Code

```
#include "WPILib.h"    // WPILibrary.h
#include "Math.h"    // Math.h required for fabs function
//Last modified: January 30, 2014 by: Alan
class RobotDemo : public SampleRobot
{
    Talon frontLeft, frontRight, backLeft, backRight; //Talon Motor Controllers
    Joystick logitech;     // Logitech F310 Gamepad/Controller

public:
    RobotDemo(void):
        frontLeft(1),
        frontRight(2),
        backLeft(3),
        backRight(4),
        logitech(1)
/*
 * Set motor expiration to prevent unwarranted movement if connection lost or
 * disabled
 */
    {
        frontLeft.SetExpiration(0.1),
        frontRight.SetExpiration(0.1),
        backLeft.SetExpiration(0.1),
        backRight.SetExpiration(0.1);
    }

    void Autonomous(void)
    {
    }

    /**
     * Runs the motors with tank steering
     */
    void OperatorControl(void)
    {
        while (IsOperatorControl())
        {
            if(fabs(logitech.GetRawAxis(2)) > 0.2)    /*left joystick, forward & back*/
            {
                frontLeft.Set(logitech.GetRawAxis(2) * -.65);
                backLeft.Set(logitech.GetRawAxis(2) * -.65);
            }
            else
            {
                frontLeft.Set(0);
                backLeft.Set(0);
            }
            if(fabs(logitech.GetRawAxis(4)) > 0.2)                /*right joystick, forward & back*/
            {
                frontRight.Set(logitech.GetRawAxis(4) * .65);
                backRight.Set(logitech.GetRawAxis(4) * .65);
            }
            else
            {
                frontRight.Set(0);
                backRight.Set(0);
            }
                Wait(0.005);    // wait 0.005 seconds before repeating loop
        }
    }

    /**
     * Runs during test mode
     */
    void Test() {
        while(IsTest())
        {
        }
```

```
    }
};
START_ROBOT_CLASS(RobotDemo);
```

➠ The Explanation

```
#include "WPILib.h"    // WPILibrary.h
#include "Math.h"    // Math.h required for fabs function
```

`WPILib.h` is always our spellbook. `Math.h` is a different library that is generally used in C++ for math functions. As stated in the comment, it is used for the `fabs` (float absolute value) function that appears in the tank drive portion of this code.

```
class RobotDemo : public SampleRobot
{
    Talon frontLeft, frontRight, backLeft, backRight; //Talon Motor Controllers
    Joystick logitech;     // Logitech F310 Gamepad/Controller
```

`RobotDemo` class with inherited methods from `SampleRobot` makes templating easier. The talons are one of several motor controllers explained in an earlier section of this manual. `RobotDrive` in the original template declares them in the background, but then you are limited to it's drive methods. This custom program uses a logitech gamepad which is one joystick object. If you look in the WPILib reference, it will show that to use the tank drive method of the `RobotDrive`, you need two joysticks.

```
public:
    RobotDemo(void):
        frontLeft(1),
        frontRight(2),
        backLeft(3),
        backRight(4),
        logitech(1)
// Set motor expiration to prevent unwarranted movement if connection lost or disabled
    {
        frontLeft.SetExpiration(0.1),
        frontRight.SetExpiration(0.1),
        backLeft.SetExpiration(0.1),
        backRight.SetExpiration(0.1);
    }
```

Now in the constructor, the motor controllers are instantiated in the ports of the digital sidecar corresponding to the number in the parentheses. The joystick is instantiated using the USB port. As mentioned in the comment, inside the other braces, there are `SetExpiration`s to prevent continued movement in event of disablement or lost connection. It does this by shutting down power to the object that has expired; now as long as you're connected, the motors are "fed" and the expirations refresh.

```
    /**
     * Insert own comment
     */
    void Autonomous(void)
    {
    }
```

Autonomous code goes here. However this custom program is to show tank drive not full autopilot.

```
    /**
     * Runs the motors with tank steering
     */
    void OperatorControl(void)
    {
        while (IsOperatorControl())
        {
```

```
            if(fabs(logitech.GetRawAxis(2)) > 0.2)                /*left joystick, forward & back*/
            {
                frontLeft.Set(logitech.GetRawAxis(2) * -.65);
                backLeft.Set(logitech.GetRawAxis(2) * -.65);
            }
            else
            {
                frontLeft.Set(0);
                backLeft.Set(0);
            }
            if(fabs(logitech.GetRawAxis(4)) > 0.2)                /*right joystick, forward & back*/
            {
                frontRight.Set(logitech.GetRawAxis(4) * .65);
                backRight.Set(logitech.GetRawAxis(4) * .65);
            }
            else
            {
                frontRight.Set(0);
                backRight.Set(0);
            }
            Wait(0.005);    // wait for a motor update time
        }
    }
```

This here is the juicy part, this is tank drive. For those who do not know tank drive, one joystick controls one side of the drive, so when one stick is pushed, only one side moves and the other joystick controls the other respective side. This is where the fabs function is used to shorten coding lines. Normally there would have to be two conditions in those ifs for tank drive to work; if the joystick is greater than a threshold OR if the joystick is below negative threshold. It would look like `joystick.GetRawAxis(2) > 0.2 || joystick.GetRawAxis(2) < -0.2`. Compared to what is in there, it is much easier to code, but less understandable. Reasoning is when you tilt the joystick back it is negative, but it does not pass > `0.2`. Use `fabs` or absolute value(for floats, just abs for ints), less coding.

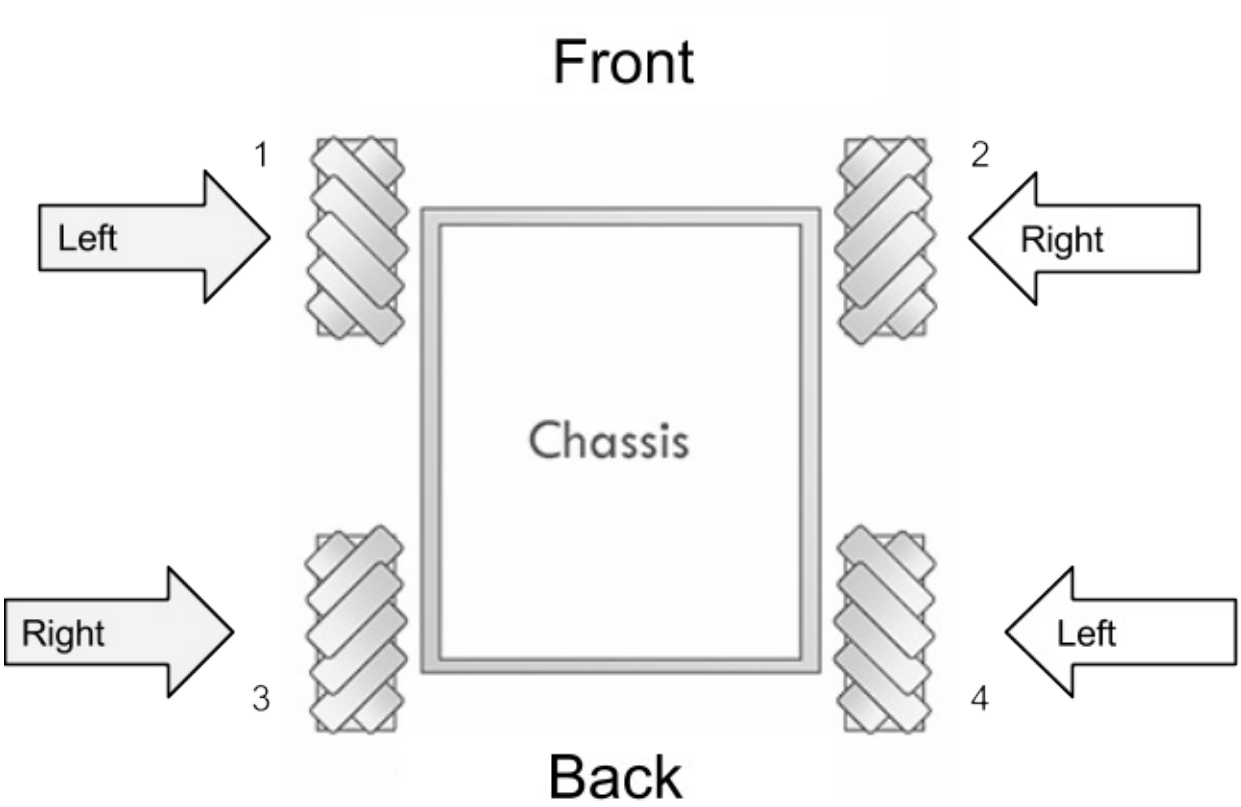
```
    /**
     * Runs during test mode
     */
    void Test() {
        while(IsTest())
        {
        }
    }
};
START_ROBOT_CLASS(RobotDemo);
```

Closes brace from the beginning of the class. `START_ROBOT_CLASS(RobotDemo);` notice how `RobotDemo` is in the parameter, isn't that the class? So this is also important for it runs the class.

Custom Program (Mecanum Drive)

➠ Introduction & Wheel Configuration

The Mecanum Drive allows the robot to move forward, backward, and strafe. This is possible due to the nature of the wheels, which slip because of the rollers on them. They will naturally travel in a 45 degree motion in the direction that the entire wheel is rotating. When working with mecanum wheels, it is important to consider the weight distribution of the robot frame because mecanum wheels are designed for robots with an even weight distribution. An uneven weight distribution will cause wheels supporting more weight to have more traction than the wheels supporting less weight. This difference in traction will modify the effective rotation of each wheel, and the effect of the mecanum drive is lost.



Left configuration

- Rotating Forwards: motion 45 degrees **north of east**
- Rotating Backwards: motion 45 degrees **south of west**
- Used by: **Wheel 1**, front left, and **Wheel 4**, back right.

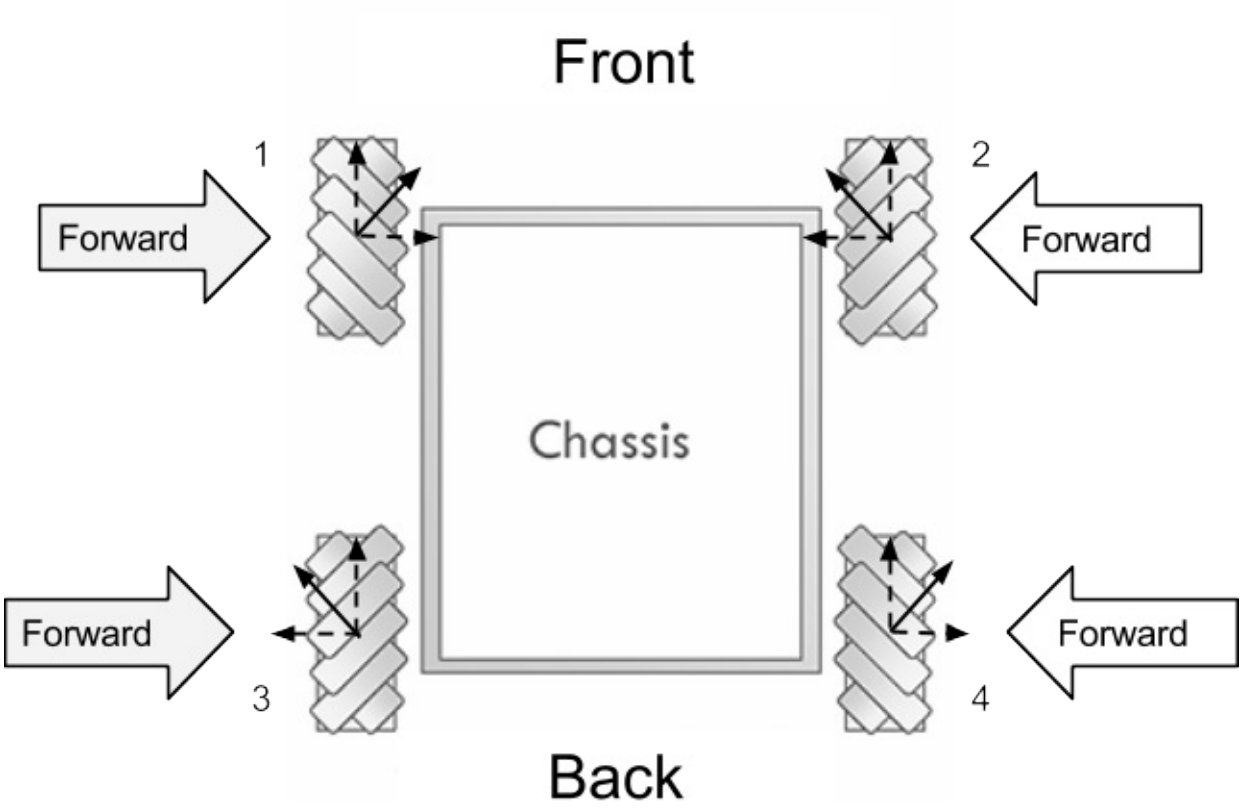
Right Configuration

- Rotating Forwards: motion 45 degrees **north of west**
- Rotating Backwards: motion 45 degrees **south of east**
- Used by: **Wheel 2**, front right, and **Wheel 3**, back left.

➠ Movement Configuration

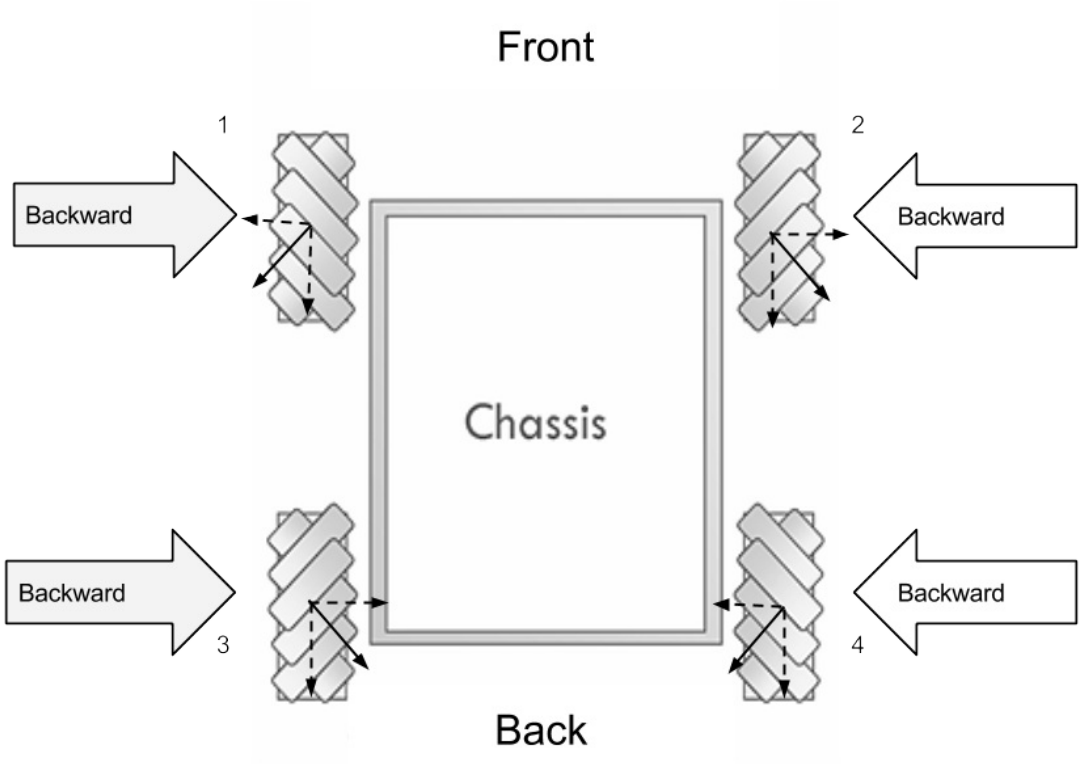
Driving Forward

Wheels 1, 2, 3, and 4 are rotating forward to allow the drive frame to drive forward.



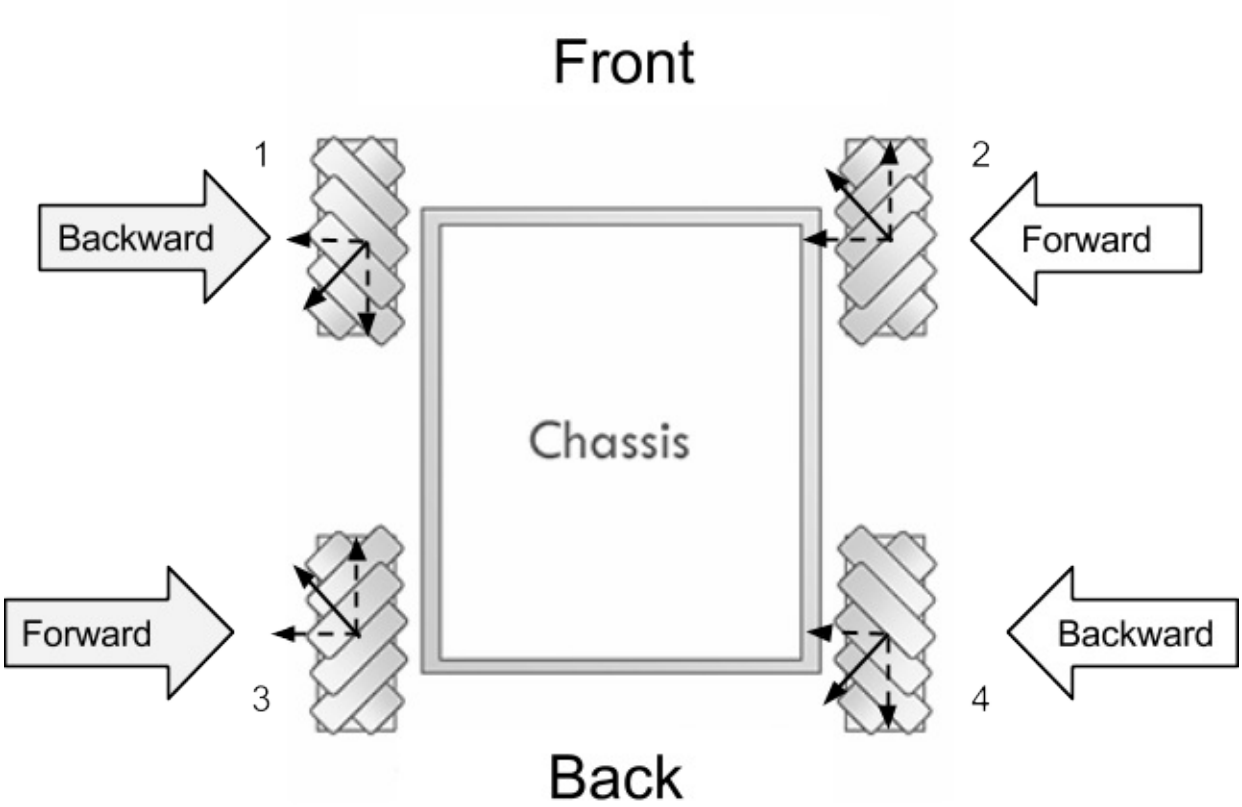
Driving Backward

Wheels 1, 2, 3, and 4 are rotating backward to allow the drive frame to drive backward.



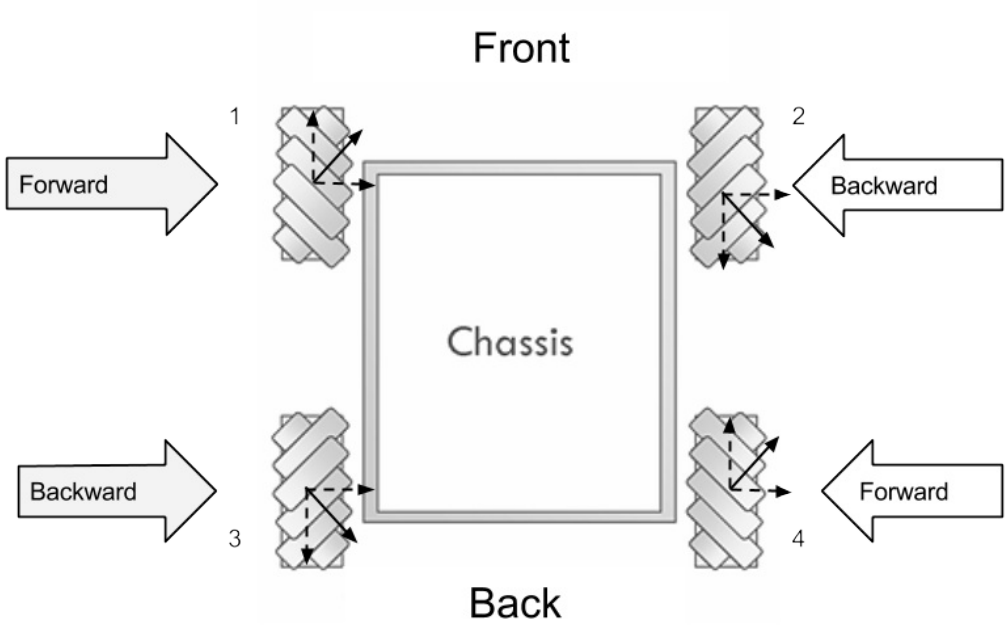
Strafing Left

Wheels 2 and 3 are rotating forward, Wheels 1 and 4 are rotating backward to allow the drive frame to strafe toward the left.



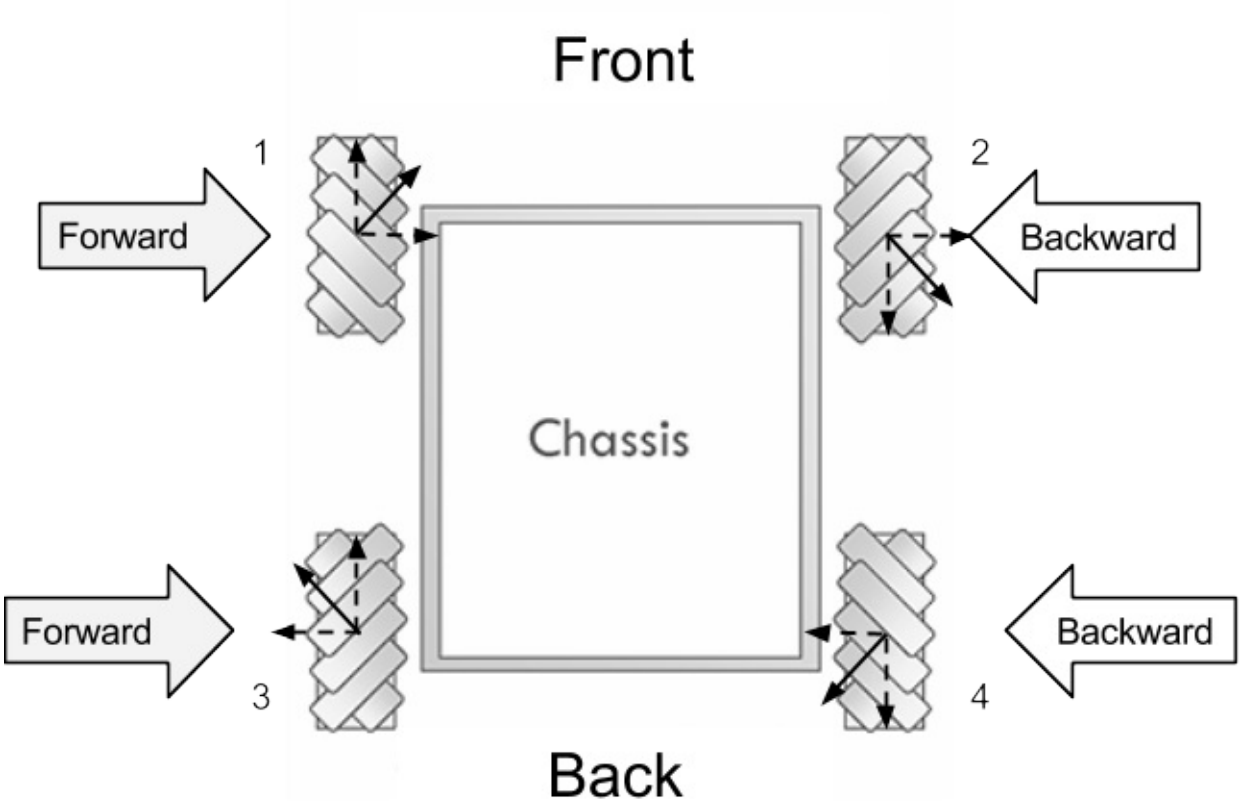
Strafing Right

Wheels 1 and 4 are rotating forward, Wheels 2 and 3 are rotating backward to allow the drive frame to strafe toward the right.



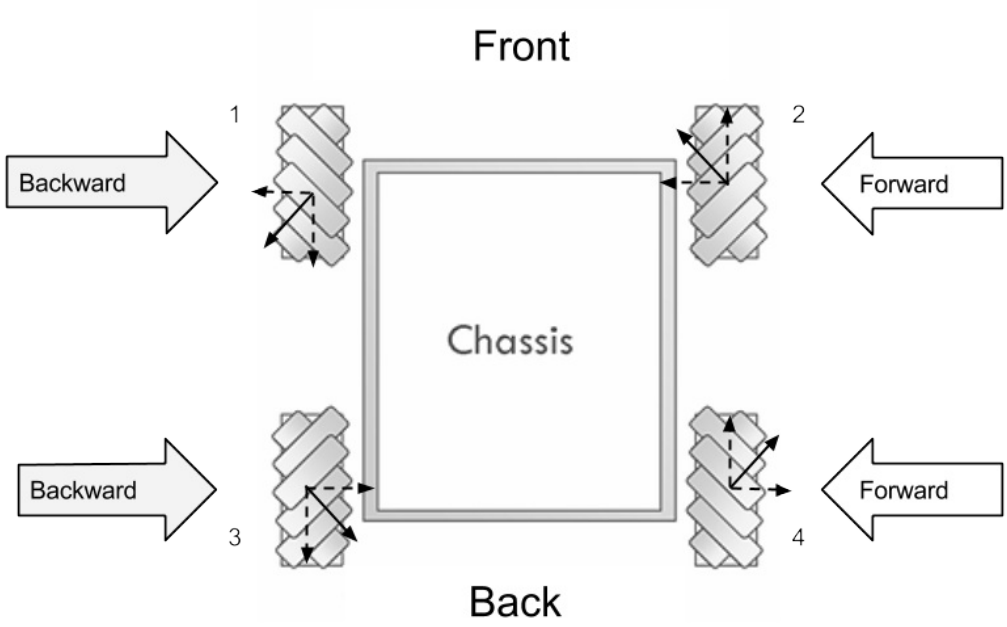
Turning Clockwise

Wheels 1 and 3 are rotating forward, Wheel 2 and 4 are rotating backward to allow the drive frame to rotate clockwise about its center



Turning Counter-Clockwise

Wheels 2 and 4 are rotating forward, Wheels 1 and 3 are rotating backward to allow the drive frame to rotate counter-clockwise about its center



➠ Sample Testing Code

```
#include "WPILib.h"
#include “Math.h”

class RobotDemo : public SampleRobot
{
    Victor leftFront; // Initializing motor 1; front-left motor
    Victor leftBack; // Initializing motor 3; back-left motor
    Victor rightFront;// Initializing motor 2; front-right motor
    Victor rightBack; // Initializing motor 4; back-right motor
    Joystick logitech; // Logitech Gamepad Controller

public:
    RobotDemo():
        leftFront(1),   // leftFront motor uses PWM port 1
        leftBack(2), // leftBack motor uses PWM port 2
        rightFront(3), // rightBack motor uses PWM port 3
        rightBack(4), // rightBack motor uses PWM port 4
        logitech(1) // Logitech Game Controller with Driverstation port 1
    {
    }

    void OperatorControl()
    {
          int leftFrontPolarity = 1; // These variables flip the sign value of
          int leftBackPolarity = 1; // the motors in the situation that they are
          int rightFrontPolarity = -1; // flipped
          int rightBackPolarity = -1;
          while (IsOperatorControl())
          {
            float x = 0; // x-axis motion-right (+), left (-)
            float y = 0; // y-axis motion-forward (+), backward (-)
            float z = 0; // z-axis motion-clockwise (+), counterclockwise (-)
            if (fabs(stickOne.GetRawAxis(1)) > .2)
                z = stickOne.GetRawAxis(1); // z-axis threshold
```

```
            if (fabs(stickOne.GetRawAxis(2)) > .2)
                y = -(stickOne.GetRawAxis(2)); // y-axis threshold
            if (fabs(stickTwo.GetRawAxis(1)) > .2)
                x = stickTwo.GetRawAxis(1); // x-axis threshold

        // y-axis motion
            if (fabs(y) > fabs(x) && fabs(y) > fabs(z)) //Activates if y is largest
            {
                leftFront.Set(y * leftFrontPolarity);
                rightFront.Set(y * rightFrontPolarity);
                leftBack.Set(y * leftBackPolarity);
                rightBack.Set(y * rightBackPolarity);
            }
        // x-axis motion
            if (fabs(x) > fabs(y) && fabs(x) > fabs(z)) //Activates if x is largest
            {
                leftFront.Set(x * leftFrontPolarity);
                rightFront.Set(x * rightFrontPolarity * -1);
                leftBack.Set(x * leftBackPolarity * -1);
                rightBack.Set(x * rightBackPolarity);
            }

        // z-axis motion
            else if (fabs(z) > fabs(y) && fabs(z) > fabs(x))
            {
                leftFront.Set(z * leftFrontPolarity);
                rightFront.Set(z * rightFrontPolarity * -1);
                   leftBack.Set(z * leftBackPolarity);
                rightBack.Set(z * rightBackPolarity * -1);
            }
            else // Otherwise sticks are not pushed
            {
                leftFront.Set(0);
                leftBack.Set(0);
                rightFront.Set(0);
                rightBack.Set(0);
            }
            Wait(0.005);
        }
    }
};
START_ROBOT_CLASS(RobotDemo);
```

➠ The Explanation

Breakdown of the code follows as so:

```
#include "WPILib.h"
#include "Math.h"
class RobotDemo : public SimpleRobot
{
    Victor leftFront; // Initializing motor 1; front-left motor
    Victor leftBack; // Initializing motor 3; back-left motor
    Victor rightFront;// Initializing motor 2; front-right motor
    Victor rightBack; // Initializing motor 4; back-right motor
    Joystick logitech; // Logitech Gamepad Controller
```

Here we instantiate the 4 motor controllers we are using to manipulate the 4 mecanum wheels on the robot under the `Victor` class (here we used Victor motor controllers). We also instantiated our Logitech Gamepad Controller under the `Joystick` class.

```
public:
    RobotDemo():
        leftFront(1),   // leftFront motor uses PWM port 1
        leftBack(2), // leftBack motor uses PWM port 2
        rightFront(3), // rightBack motor uses PWM port 3
        rightBack(4), // rightBack motor uses PWM port 4
        logitech(1) // Logitech Game Controller with Driverstation port 1
    {
    }
```

We further define our constructors by associating each piece of hardware to their respective ports. The `Victor` Class, which is a category of motor controllers, utilize PWM ports while the `Joystick` Class utilized for the Logitech Gamepad Controller utilizes the driver station (USB) ports

```
void OperatorControl()
{
    DriverStationLCD *screen = DriverStationLCD::GetInstance();
      int leftFrontPolarity = 1; // These variables flip the sign value of
      int leftBackPolarity = 1; // the motors in the situation that they are
      int rightFrontPolarity = -1; // flipped
      int rightBackPolarity = -1;
```

These variables are in place to control the polarity of the motors (whether they rotate forwards or backwards when pushing the left and right sticks in a certain direction). This makes it easier to fix the code in the event of a motor being reversed.

```
      while (IsOperatorControl())
      {
         float x = 0; // x-axis motion-right (+), left (-)
         float y = 0; // y-axis motion-forward (+), backward (-)
         float z = 0; // z-axis motion-clockwise (+), counterclockwise (-)
```

This splits the axes of the motion so that they can be assigned based on how the thumbsticks are pushed. The pushing the left thumbstick on it's y-axis will give a y-value (`+` = forward, `-` = backward), pushing the left stick on it's x-axis gives a z-value (`+` = clockwise, `-` = backward), and pushing the right stick on it's x-axis give a x-value (`+` = right, `-` = left).

```
        if (fabs(stickOne.GetRawAxis(1)) > .2)
            z = stickOne.GetRawAxis(1); // z-axis threshold
        if (fabs(stickOne.GetRawAxis(2)) > .2)
            y = -(stickOne.GetRawAxis(2)); // y-axis threshold
        if (fabs(stickTwo.GetRawAxis(1)) > .2)
            x = stickTwo.GetRawAxis(1); // x-axis threshold
```

This section assigns a value to the axes based on the orientation of the thumbsticks. A threshold is placed so that tiny accidental movements do not cause the robot to drift.

```
        // y-axis motion
        if (fabs(y) > fabs(x) && fabs(y) > fabs(z)) //Activates if y is largest
        {
            leftFront.Set(y * leftFrontPolarity);
            rightFront.Set(y * rightFrontPolarity);
            leftBack.Set(y * leftBackPolarity);
            rightBack.Set(y * rightBackPolarity);
        }
```

If the left stick is pushed more on it's y-axis (forward / backward) than it or the right stick is pushed on their x-axis, then the robot will move forward or backwards depending on the direction of the thumbstick. Pushing forward will make all wheels rotate forward and pushing backwards makes all wheels rotate backward. Also, the speed of the motors depends on how much the left thumbstick is pushed along the y-axis.

```
      // x-axis motion
        if (fabs(x) > fabs(y) && fabs(x) > fabs(z))
         //Activates if x is largest
        {
            leftFront.Set(x * leftFrontPolarity);
            rightFront.Set(x * rightFrontPolarity * -1);
            leftBack.Set(x * leftBackPolarity * -1);
            rightBack.Set(x * rightBackPolarity);
        }
```

If the right stick's x-axis magnitude is greater than any of the left stick's axes, then the robot will strafe either right or left.

Pushing the thumbstick to the right makes the left front and right back motors rotate forward while the other two reverse (used the vector diagram to determine direction). Pushing the thumbstick to the left makes the opposite happen, with the right front and left back rotating forward while the left front and right back reverse. Again, the speed of the motors depends on how large the magnitude of the right thumbstick's x-axis is.

```
// z-axis motion
    else if (fabs(z) > fabs(y) && fabs(z) > fabs(x))
            {
                leftFront.Set(z * leftFrontPolarity);
                rightFront.Set(z * rightFrontPolarity * -1);
                   leftBack.Set(z * leftBackPolarity);
                rightBack.Set(z * rightBackPolarity * -1);
            }
```

If the magnitude of the left stick's x-axis is greater than it's own y-axis and the right thumbsticks x-axis, then the robot will rotate. If the left stick is pushed to the right, the left wheels will rotate forward and the right wheels will rotate backwards, making it turn clockwise, much like tank drive. The opposite happens when you push the stick to the left.

```
             else
             {
                    leftFront.Set(0);
                    leftBack.Set(0);
                    rightFront.Set(0);
                    rightBack.Set(0);
             }
```

This sets all the motors to `0` when the joysticks are not pushed in an assigned direction or are not pushed past the threshold

Alternate Code (put this in place of all of the if else statements)

```
leftFront.Set(x-y-z);
leftBack.Set(-x-y-z);
rightFront.Set(x+y+z);
rightBack.Set(-x+y+z);
```

This code is used as a simplification of the one posted earlier. However, there is a fundamental difference in how these two operate. In the original code, you can only move in specific directions, like forward, backwards, right, left, and rotate, whereas in this version you can move in any combination of the three axes. This is achieved by bypassing if statements and just using addition and subtraction. This way also allows you to bypass the issue of making a specific variable for polarity, as you can just change the `+` or `-` for the specific motor. In this situation, the left motors were reversed, so it was necessary to change `+y` to `-y` and `+z` to `-z` as it is now. You also have to switch the sign of the `x` variable. If the left motors were not reversed, the code would be `leftFront.Set(-x+y+z)` and `leftBack.Set(x+y+z)`. However, there is an issue that exists within this code as it is possible for the set value for each motor to exceed 1 if you were to rotate while moving in another direction.

➠ Tested and Modified Code

```
// This is temporary code that will be replaced in the bible upon final code completion
#include "WPILib.h"
#include "Math.h"

class RobotDemo : public SimpleRobot
{
    Victor leftFront; // Initializing motor 1; front-left motor
    Victor leftBack; // Initializing motor 3; back-left motor
    Victor rightFront; // Initializing motor 2; front-right motor
    Victor rightBack; // Initializing motor 4; back-right motor
    Joystick stickOne; // Logitech Gamepad Controller
    Joystick stickTwo; // Logitech Gamepad Controller

public:
```

```
RobotDemo():
    leftFront(1),   // leftFront motor uses PWM port 1
    leftBack(2), // leftBack motor uses PWM port 2
    rightFront(3), // rightBack motor uses PWM port 3
    rightBack(4), // rightBack motor uses PWM port 4
    stickOne(1), // Logitech Game Controller with Driverstation port 1
  stickTwo(2) // Joystick with driverstation port 2
{
}

void OperatorControl()
{
    DriverStationLCD *screen = DriverStationLCD::GetInstance();
    int leftFrontPolarity = 1; // These variables flip the sign value of
    int leftBackPolarity = 1; // the motors in the situation that they are
    int rightFrontPolarity = -1; // flipped
    int rightBackPolarity = -1;
    float x = 0; // x-axis motion-right (+), left (-)
    float y = 0; // y-axis motion-forward (+), backward (-)
    float z = 0; // z-axis motion-clockwise (+), counterclockwise (-)
    while (IsOperatorControl())
    {
        if (fabs(stickOne.GetRawAxis(1)) > .2)
            z = stickOne.GetRawAxis(1); // z-axis threshold
        if (fabs(stickOne.GetRawAxis(2)) > .2)
            y = -(stickOne.GetRawAxis(2)); // y-axis threshold
        if (fabs(stickTwo.GetRawAxis(1)) > .2)
            x = stickTwo.GetRawAxis(1); // x-axis threshold

        // y-axis motion
        if (fabs(y) > fabs(x) && fabs(y) > fabs(z)) //Activates if y is largest
        {
            leftFront.Set(y * leftFrontPolarity);
            rightFront.Set(y * rightFrontPolarity);
                    leftBack.Set(y * leftBackPolarity);
                    rightBack.Set(y * rightBackPolarity);
        }
        // x-axis motion
 //Activates when x is largest
        else if (fabs(x) > fabs(y) && fabs(x) > fabs(z))
        {
            if(x > 0)
            {
             //Executes if x is greater than deadband of 0.5
                if(fabs(x) >= 0.5)
                {
                    leftFront.Set(x * leftFrontPolarity * 1.1);
                    rightFront.Set(x * rightFrontPolarity * -0.95);
                    leftBack.Set(x * leftBackPolarity * -1.1);
                    rightBack.Set(x * rightBackPolarity);
                }
                else if(fabs(x) > 0.35)
                {
                leftFront.Set(x * leftFrontPolarity * 0.9);
                rightFront.Set(x * rightFrontPolarity *-0.9);
                leftBack.Set(x * leftBackPolarity * -1.1);
                rightBack.Set(x * rightBackPolarity);
                }
            }
            else if(x < 0)
            {
                if(fabs(x) >= 0.5)
                {
                    leftFront.Set(x * leftFrontPolarity * 1.05);
                    rightFront.Set(x * rightFrontPolarity * -1);
                    leftBack.Set(x * leftBackPolarity * -1);
                    rightBack.Set(x * rightBackPolarity);
                }
                else if(fabs(x) > 0.35)
                {
                    leftFront.Set(x * leftFrontPolarity * 1.1);
                    rightFront.Set(x * rightFrontPolarity * -1);
                    leftBack.Set(x * leftBackPolarity * -1.1);
                    rightBack.Set(x * rightBackPolarity);
                }
            }
        }
         // z-axis motion
        else if (fabs(z) > fabs(y) && fabs(z) > fabs(x))
        {
            leftFront.Set(z * leftFrontPolarity);
            rightFront.Set(z * rightFrontPolarity * -1);
```

```
                leftBack.Set(z * leftBackPolarity);
                rightBack.Set(z * rightBackPolarity * -1);
            }
            else if(stickOne.GetRawButton(5)) //turn left when pressing 5
            {
                leftFront.Set(0.5);
                rightFront.Set(0.3);
                leftBack.Set(0.3);
                rightBack.Set(0.3);
            }
            else if(stickOne.GetRawButton(4)) //turn right when pressing 4
            {
                leftFront.Set(-0.3);
                rightFront.Set(-0.5);
                leftBack.Set(-0.3);
                rightBack.Set(-0.5);
            }

             else // Otherwise sticks are not pushed
            {
                leftFront.Set(0);
                leftBack.Set(0);
                rightFront.Set(0);
                rightBack.Set(0);
            }
            x = 0; // x-axis motion-right (+), left (-)
            y = 0; // y-axis motion-forward (+), backward (-)
            z = 0; // z-axis motion-clockwise (+), counterclockwise (-)
            screen -> PrintfLine(DriverStationLCD::kUser_Line1,"X: %f", stickOne.GetRawAxis(1));
            screen -> PrintfLine(DriverStationLCD::kUser_Line2,"Y: %f", stickOne.GetRawAxis(2));
            screen -> PrintfLine(DriverStationLCD::kUser_Line3,"Rotation: %f", stickTwo.GetRawAxis(1));
            screen -> UpdateLCD();
            Wait(0.1);
         }
     }
    void Test()
    {
        while (IsTest())
        {
            // Forward polarity test
            if (stickOne.GetRawButton(6))
                leftFront.Set(.3);
            else if (stickOne.GetRawButton(7))
                leftBack.Set(.3);
            else if (stickOne.GetRawButton(11))
                rightFront.Set(-.3);
            else if (stickOne.GetRawButton(10))
                rightBack.Set(-.3);
            else
            {
                leftFront.Set(0);
                leftBack.Set(0);
                rightFront.Set(0);
                rightBack.Set(0);
            }
        }
    }
};
START_ROBOT_CLASS(RobotDemo);
```

➠ The Explanation

Breakdown of the code follows as so:

```
#include "WPILib.h"

class RobotDemo : public SimpleRobot
{
    Victor leftFront; // Initializing motor 1; front-left motor
    Victor leftBack; // Initializing motor 3; back-left motor
    Victor rightFront;// Initializing motor 2; front-right motor
    Victor rightBack; // Initializing motor 4; back-right motor
    Joystick stickOne; // Logitech Gamepad Controller
    Joystick stickOne; // Logitech Gamepad Controller
```

Here we instantiate the four motor controllers we are using to manipulate the 4 mecanum wheels on the robot under the

Victor class (here we used Victor motor controllers). We also instantiated our two joysticks that will be controlling the motion of the robots

```
public:
    RobotDemo():
        leftFront(1),   // leftFront motor uses PWM port 1
        leftBack(2), // leftBack motor uses PWM port 2
        rightFront(3), // rightBack motor uses PWM port 3
        rightBack(4), // rightBack motor uses PWM port 4
        stickOne(1) // Logitech attack 3 with Driverstation port 1
        stickTwo(2) // Logitech attack 3 with Driverstation port 2
    {
    }
```

Here, we further define our constructors by associating each piece of hardware to their respective ports. The Victor Class, which is a category of motor controllers, utilize PWM ports while the Joystick Class utilized for the Logitec Attack 3 utilizes the driversation ports.

```
void OperatorControl()
    {
        DriverStationLCD *screen = DriverStationLCD::GetInstance();
        int leftFrontPolarity = 1; // These variables flip the sign value of
        int leftBackPolarity = 1; // the motors in the situation that they are
        int rightFrontPolarity = -1; // flipped
        int rightBackPolarity = -1;
        float x = 0; // x-axis motion-right (+), left (-)
        float y = 0; // y-axis motion-forward (+), backward (-)
        float z = 0; // z-axis motion-clockwise (+), counterclockwise (-)
```

The top half of this section is meant to be in preparation for the situation where one or more wheels need to have their polarity (going forwards or backwards) flipped. The bottom half is meant to instantiate and construct the variables that will be representing our various axes of motion. These will be used to control the voltage sent to each individual motor.

```
while (IsOperatorControl())
    {
        if (fabs(stickOne.GetRawAxis(1)) > .2)
            z = stickOne.GetRawAxis(1); // z-axis threshold
        if (fabs(stickOne.GetRawAxis(2)) > .2)
            y = -(stickOne.GetRawAxis(2)); // y-axis threshold
        if (fabs(stickTwo.GetRawAxis(1)) > .2)
            x = stickTwo.GetRawAxis(1); // x-axis threshold
```

This section serves two main purposes. The first one is setting a threshold for all axes of motion. The joysticks must be pushed past a value of .2 in order for its value to be considered valid. This is meant to prevent the robot from drifting due to the joystick not perfectly resting at 0. The second function is to assign each axis of motion to a joystick direction. In our scenario, we preferred to make pushing stickOne left and right rotate the vehicle counter clockwise and clockwise repsectively. Pushing stickOne forwards and backwards correlates to forwards and backwards motion. Pushing stickTwo to the right and left correlates to strafing right and left.

```
//y-axis motion
if (fabs(y) > fabs(x) && fabs(y) > fabs(z)) //Activates if y is largest
            {
                leftFront.Set(y * leftFrontPolarity);
                rightFront.Set(y * rightFrontPolarity);
                leftBack.Set(y * leftBackPolarity);
                rightBack.Set(y * rightBackPolarity);
            }
```

The first line is dedicated to determining if the y-component (front and back) of stickOne's position is greater in magnitude than its z-component (left and right) and stickTwo's x-component (left and right). This is meant to make the robot only move in one direction at a time. The rest of the block is dedicated to making the robot move forward and backwards. Since all wheels rotate in the same direction, nothing needs to be flipped.

```
// x-axis motion
//Activates x when largest
else if (fabs(x) > fabs(y) && fabs(x) > fabs(z))
                    {
                if(x > 0)
                {
                    if(fabs(x) >= 0.5)
                    {
                    leftFront.Set(x * leftFrontPolarity * 1.1);
                            rightFront.Set(x * rightFrontPolarity * -0.95);
                           leftBack.Set(x * leftBackPolarity * -1.1);
                            rightBack.Set(x * rightBackPolarity);
                    }
                    else if(fabs(x) > 0.35)
                    {
                    leftFront.Set(x * leftFrontPolarity * 0.9);
                    rightFront.Set(x * rightFrontPolarity *-0.9);
                    leftBack.Set(x * leftBackPolarity * -1.1);
                    rightBack.Set(x * rightBackPolarity);
                    }
                }
```

Much like the top, this section is only activated when the x-component of stickTwo's position is larger than any of stickOne's. However, this section is different as our robot would rotate slightly clockwise and drift slightly backwards. To address this, we had to manually add multipliers to certain motors to make them move slower or faster at certain intervals. We also had to separate left strafing and right strafing because they behaved differently. In the block above, we only see the right strafing portion of the code. This section is subdivided into two more sections, when the wheels are supplied at least half of their maximum voltage (>=.5) and when they are supplied only a little bit of voltage (.5 > v > 0.35). This was due to our drive reacting differently at different voltages. You'll see multipliers like 1.1, -0.95 and 0.9 in the above code, this is because some wheels were rotating slower / faster than others. Multipliers with magnitudes below 1 are meant to slow the speed of that specific motor. Multipliers with magnitudes above 1 are meant to speed them up. Different signs (+ or -) are meant to reverse the direction of the wheel in order for the vectors to make the card move in the desired direction.

```
else if(x < 0)
    {
        if(fabs(x) >= 0.5)
        {
            leftFront.Set(x * leftFrontPolarity * 1.05);
            rightFront.Set(x * rightFrontPolarity * -1);
            leftBack.Set(x * leftBackPolarity * -1);
            rightBack.Set(x * rightBackPolarity);
        }
        else if(fabs(x) > 0.35)
        {
            leftFront.Set(x * leftFrontPolarity * 1.1);
            rightFront.Set(x * rightFrontPolarity * -1);
            leftBack.Set(x * leftBackPolarity * -1.1);
            rightBack.Set(x * rightBackPolarity);
        }
```

This is the same as the portion before this, but for strafing to the left. Since this motion had different errors that strafing to the right did, we had to edit the multipliers until the robot strafed nicely.

```
// z-axis motion
else if (fabs(z) > fabs(y) && fabs(z) > fabs(x))
  {
     leftFront.Set(z * leftFrontPolarity);
     rightFront.Set(z * rightFrontPolarity * -1);
     leftBack.Set(z * leftBackPolarity);
     rightBack.Set(z * rightBackPolarity * -1);
  }
```

This section of the code is for rotating the robot. Again, this is activated only when the magnitude of stickOne's z-component (how much to the left or right it is) is larger than both the x and y components. In order to rotate, the right sight must always be going the direction opposite of where the joystick tells it to. This is why they have a -1 applied in their statements. Pushing stickOne to the right makes the robot rotate clockwise and pushing it to the left makes it rotate

counterclockwise.

```
else if(stickOne.GetRawButton(5)) //turn left when pressing 5
            {
                leftFront.Set(0.5);
                rightFront.Set(0.3);
                leftBack.Set(0.5);
                rightBack.Set(0.3);
            }
else if(stickOne.GetRawButton(4)) //turn right when pressing 4
            {
                leftFront.Set(-0.3);
                rightFront.Set(-0.5);
                leftBack.Set(-0.3);
                rightBack.Set(-0.5);
            }
```

This section of the code is just a quick preset that we found to be useful. They make the robot move forwards and rotate slightly when either the 4 or 5-button is pressed. Pressing the 5-button makes it turn right and pushing the 4-button makes it turn left.

```
else // Otherwise sticks are not pushed
            {
                leftFront.Set(0);
                leftBack.Set(0);
                rightFront.Set(0);
                rightBack.Set(0);
            }
```

The purpose of this section is to ensure that when the joysticks are at their resting positions or very near it that the motors will not rotate the wheels.

```
x = 0; // x-axis motion-right (+), left (-)
y = 0; // y-axis motion-forward (+), backward (-)
z = 0; // z-axis motion-clockwise (+), counterclockwise (-)
```

Since this portion of the code is outside of the IsOperatorControl while loop, this makes sure that the motors will NOT rotate the wheels when the robot is not under control of the driver.

The D-Link

The D-Link

The D-Link

The D-Link

The D-Link

Changelog

February 23, 2015, *modified by Vivian*

- Added Drive Code
- Fixed Table of Contents

February 5, 2015, *modified by Vivian*

- Added Driver Station Documentation
- Added Motor Controller Documentation

February 2, 2015, *modified by Kayli and Alex*

- Added PDP Documentation
- Added D-Link Documentation
