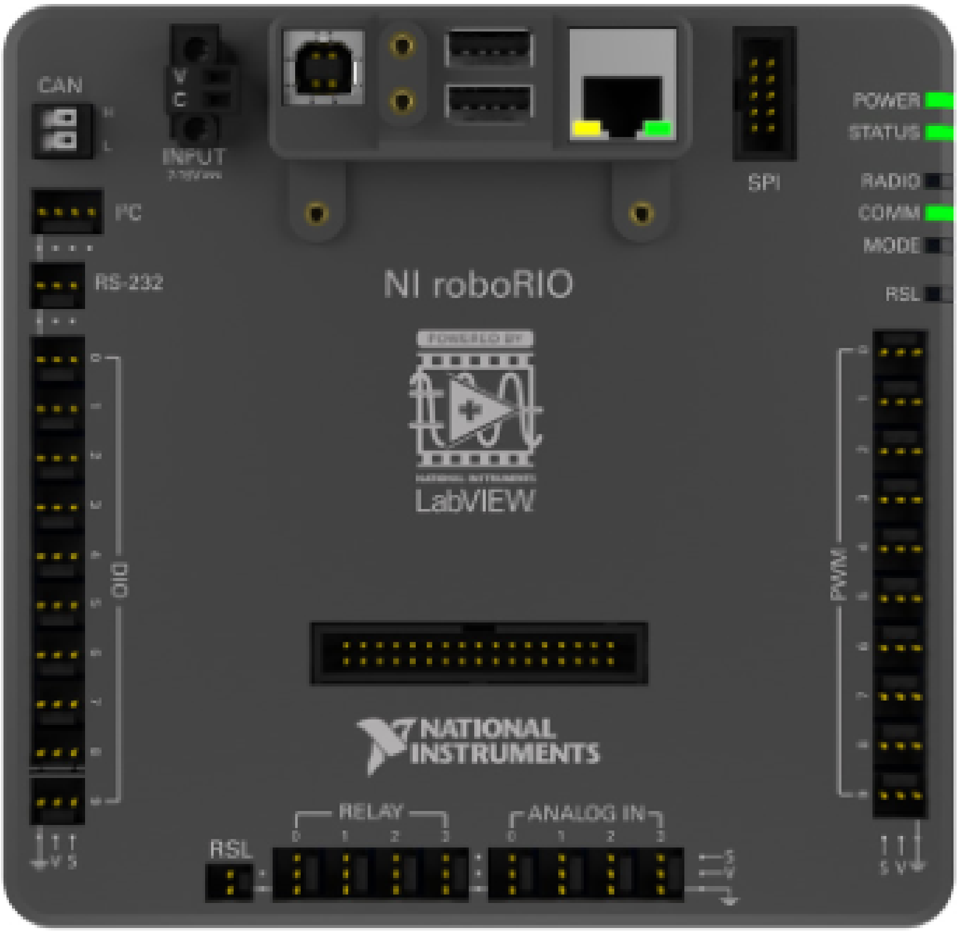
THE BOOK OF FRC ELECTRICAL

(BETA)



Trobobots
Team 2853
Mililani High School

updated 01/2015

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, our phones' cameras and endless Google searches. The fonts used are Arial, which can be downloaded **here**, and Ubuntu Monospace, which can be downloaded **here**.

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

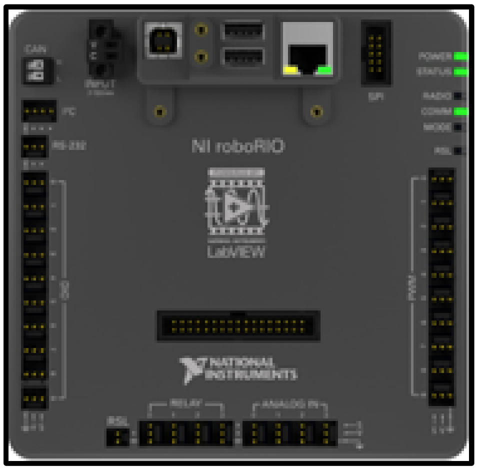
Table of Contents

The roboRIO

Jump to →

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

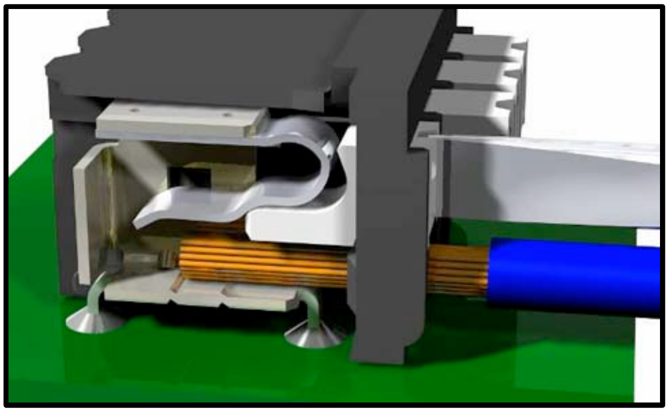
- 6.8 V to 16 V input power (staged brownout from 4.5 V to 6.8 V)
- 3.3 V user output (with 1.5 A maximum)
- 5 V user output (with 1 A maximum)

- 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

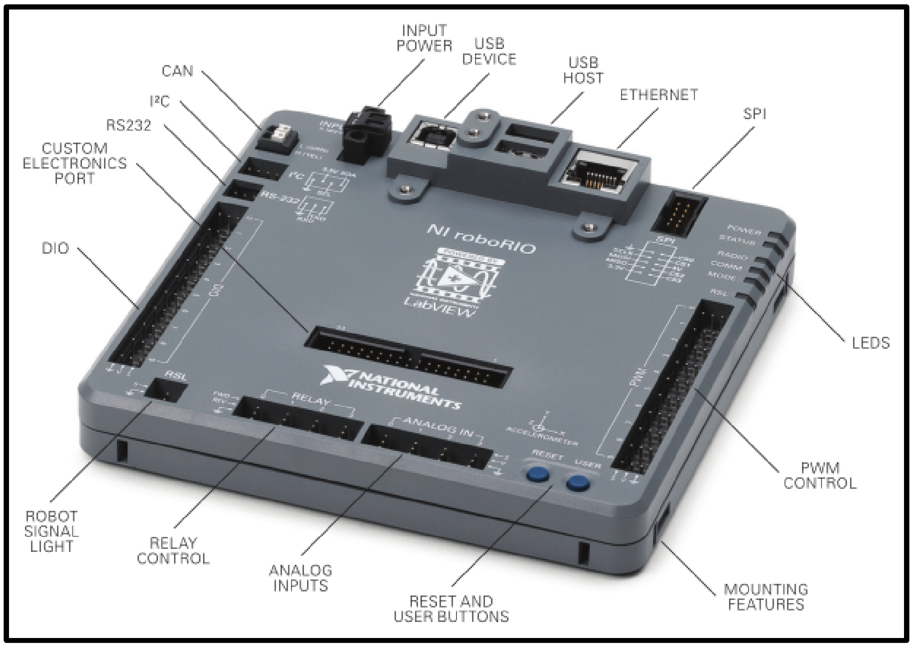
- 10 dedicated PWM channels
- 10 DIO dedicated channels
- 4 bi-directional Relay Control channels
- 2 USB Host ports
- 1 USB Device port
- 1 Ethernet port
- 1 CAN Port
- 1 Integrated, 3-axis accelerometer
- 12 V Robot signal light channel

➠ 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.

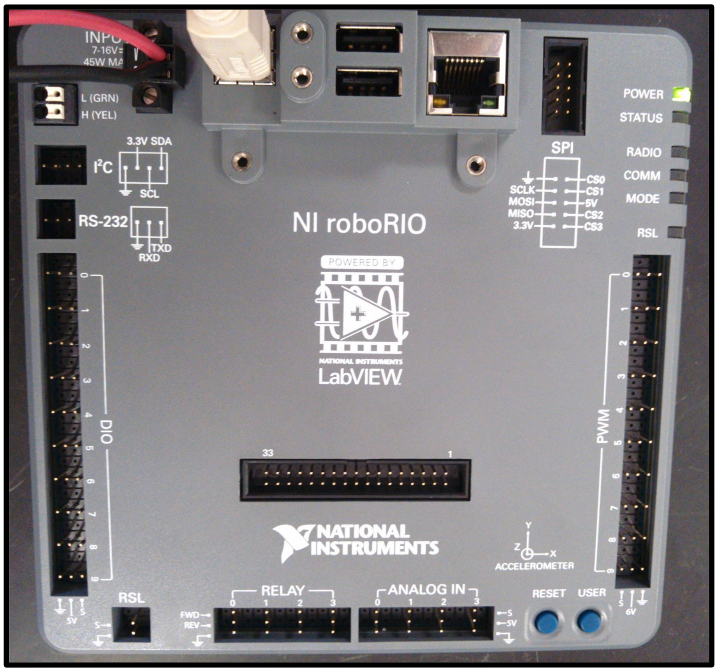
Configuring the roboRIO

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

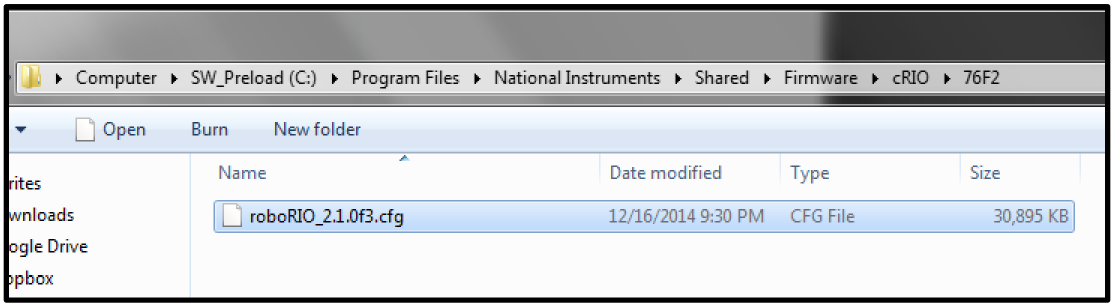
Prior to the imaging of the roboRIO, the firmware on the roboRIO must be upgraded to the latest version. This process will provide the bootloader, safemode, and firmware for the roboRIO. While it is possible to do this using an ethernet connection, it is **not** recommended. The firmware can be updated 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 (. file in your National Instruments folder. The default location of this file is under “Program Files\National Instruments\Shared\Firmware\cRIO\76F2”


Computer ▸ SW_Preload (C:) ▸ Program Files ▸ National Instruments ▸ Shared ▸ Firmware ▸ cRIO ▸ 76F2
Open
Burn
New folder
rites
wnloads
ogle Drive
opbox

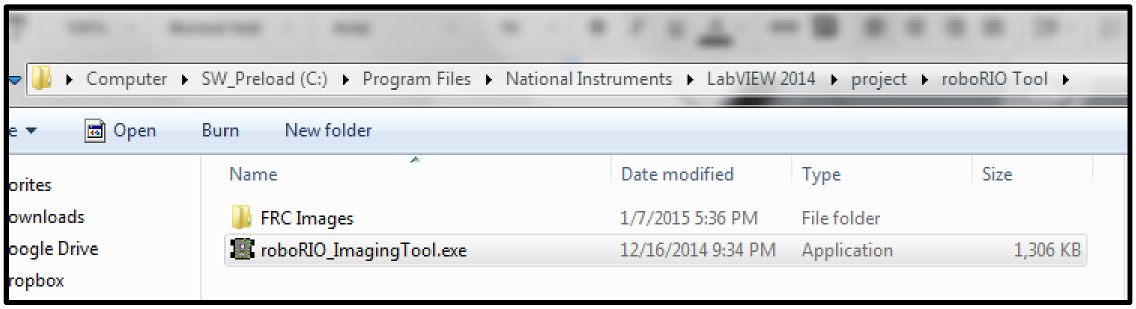
Name	Date modified	Type	Size
roboRIO_2.1.0f3.cfg	12/16/2014 9:30 PM	CFG File	30,895 KB

6. Click “Begin Update”

➠ Imaging and Reimaging the roboRIO

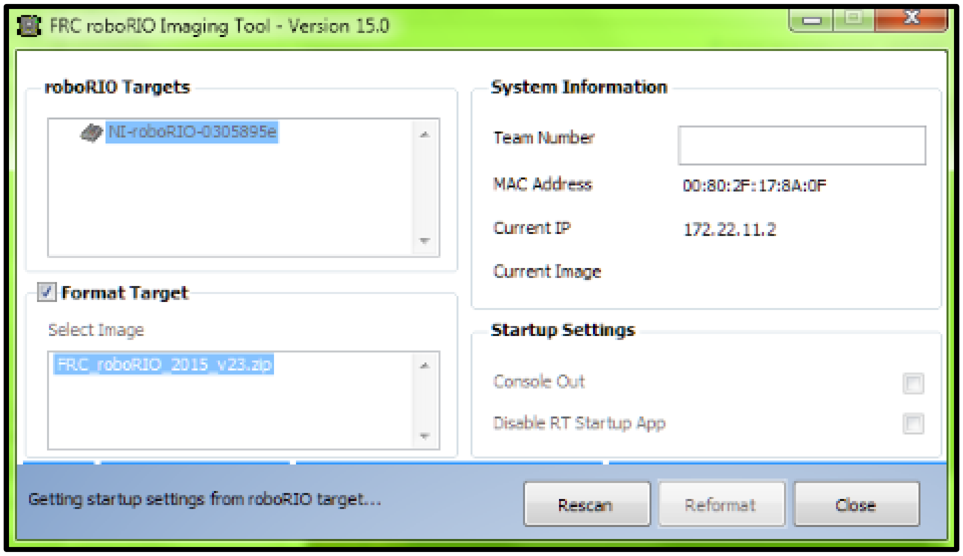
The roboRIO Image loads the FPGA, operating system, linux file system, and default settings for the roboRIO. Now that the roboRIO has the latest firmware installed, it is now possible to image the roboRIO in the follow steps:

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”


Computer ▸ SW_Preload (C:) ▸ Program Files ▸ National Instruments ▸ LabVIEW 2014 ▸ project ▸ roboRIO Tool ▸
Open
Burn
New folder
orites
ownloads
oogle Drive
ropbox

Name	Date modified	Type	Size
FRC Images	1/7/2015 5:36 PM	File folder	
roboRIO_ImagingTool.exe	12/16/2014 9:34 PM	Application	1,306 KB

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.



NI-roboRIO-0305895e

☑ Format Target

Select Image: FRC_roboRIO_2015_v23.zip

Team Number:

MAC Address: 00:80:2F:17:8A:0F

Current IP: 172.22.11.2

Current Image:

Console Out □

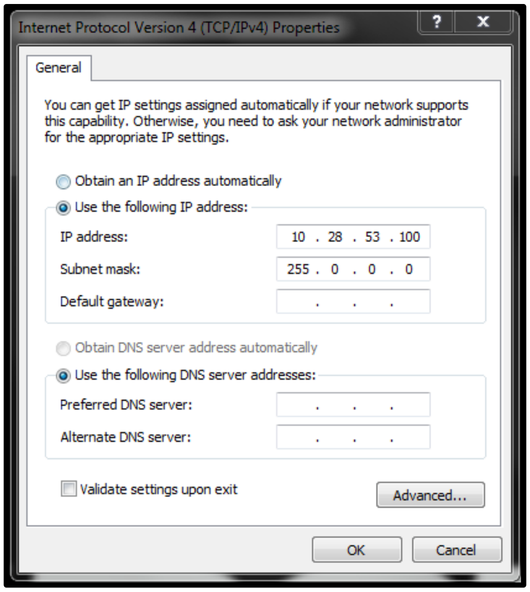
Disable RT Startup App □

Getting startup settings from roboRIO target...

5. Select the **Format Target** option but make sure that the **Console Out** and **Disable RT Startup App** options are **not** selected. Click “Reformat”
7. 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”



Internet Protocol Version 4 (TCP/IPv4) Properties

General

You can get IP settings assigned automatically if your network supports this capability. Otherwise, you need to ask your network administrator for the appropriate IP settings.

○ Obtain an IP address automatically
● Use the following IP address:

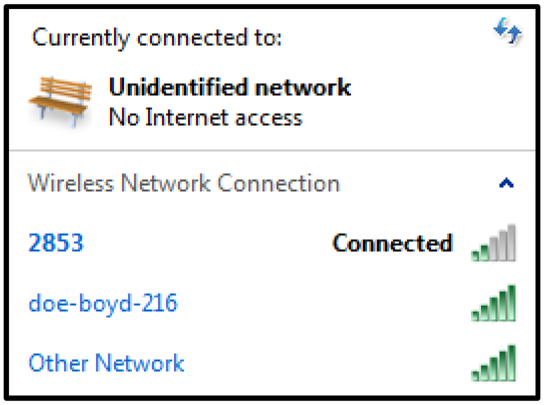
IP address: 10 . 28 . 53 . 100
Subnet mask: 255 . 0 . 0 . 0
Default gateway: . . .

○ Obtain DNS server address automatically
● Use the following DNS server addresses:

Preferred DNS server: . . .
Alternate DNS server: . . .

□ Validate settings upon exit

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



Currently connected to:

Unidentified network
No Internet access

Wireless Network Connection

2853 **Connected**

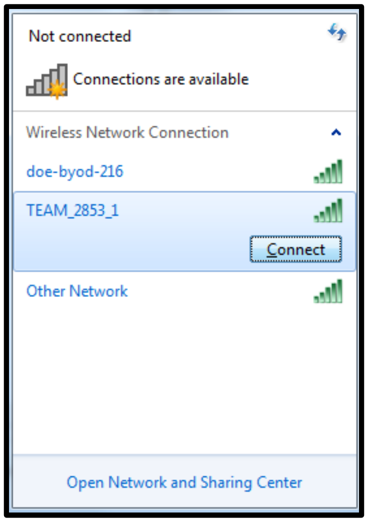
doe-boyd-216

Other Network

Uploading Code to the roboRIO

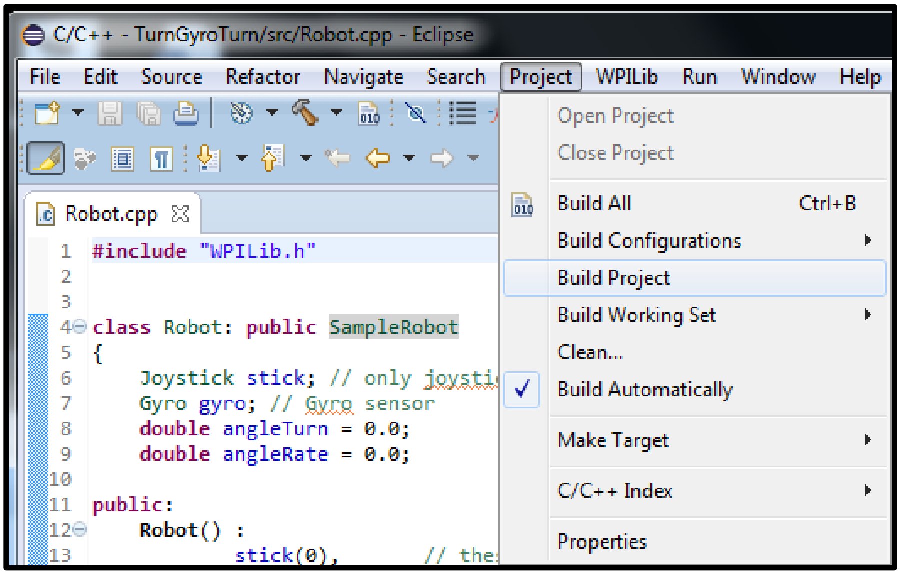
With the introduction of the roboRIO to FRC teams in 2015, uploading code to the robot is made easy! Here's the steps:

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.


Not connected
Connections are available
Wireless Network Connection
doe-byod-216
TEAM_2853_1
Connect
Other Network
Open Network and Sharing Center

Connecting to a network

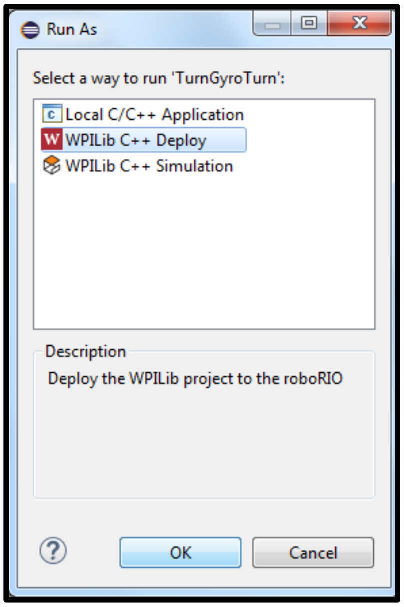
2. **Build your Project**. Click the "Project" option and then click "Build Project." To upload code to the roboRIO, it is only necessary to build the project for uploading code that hasn't been build in the past or was modified, but it's a good habit to always build your projects before uploading them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



```
#include "WPILib.h"


class Robot: public SampleRobot
{
    Joystick stick; // only joysti
    Gyro gyro; // Gyro sensor
    double angleTurn = 0.0;
    double angleRate = 0.0;

public:
    Robot() :
            stick(0),       // the
```

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!



Select a way to run 'TurnGyroTurn':

Deploy the WPILib project to the roboRIO

Running your code using “WPILib C++ Deploy”

The Controller Area Network (CAN) Bus

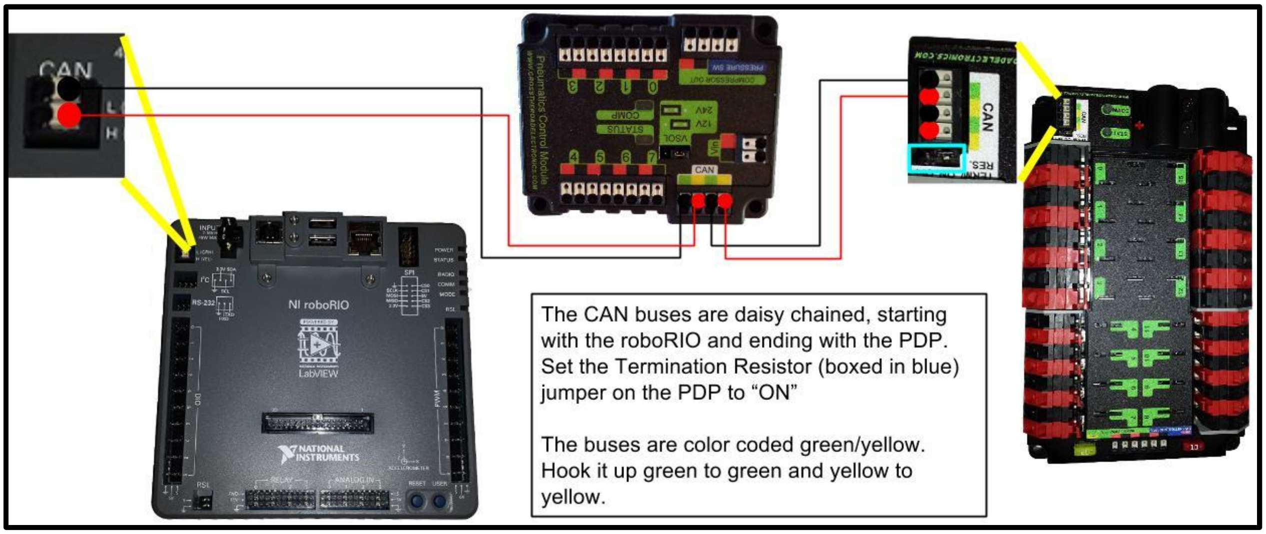
➠ Introduction

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



The CAN buses are daisy chained, starting with the roboRIO and ending with the PDP. Set the Termination Resistor (boxed in blue) jumper on the PDP to "ON"

The buses are color coded green/yellow. Hook it up green to green and yellow to yellow.

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 Power Distribution Panel (PDP)

The PDP

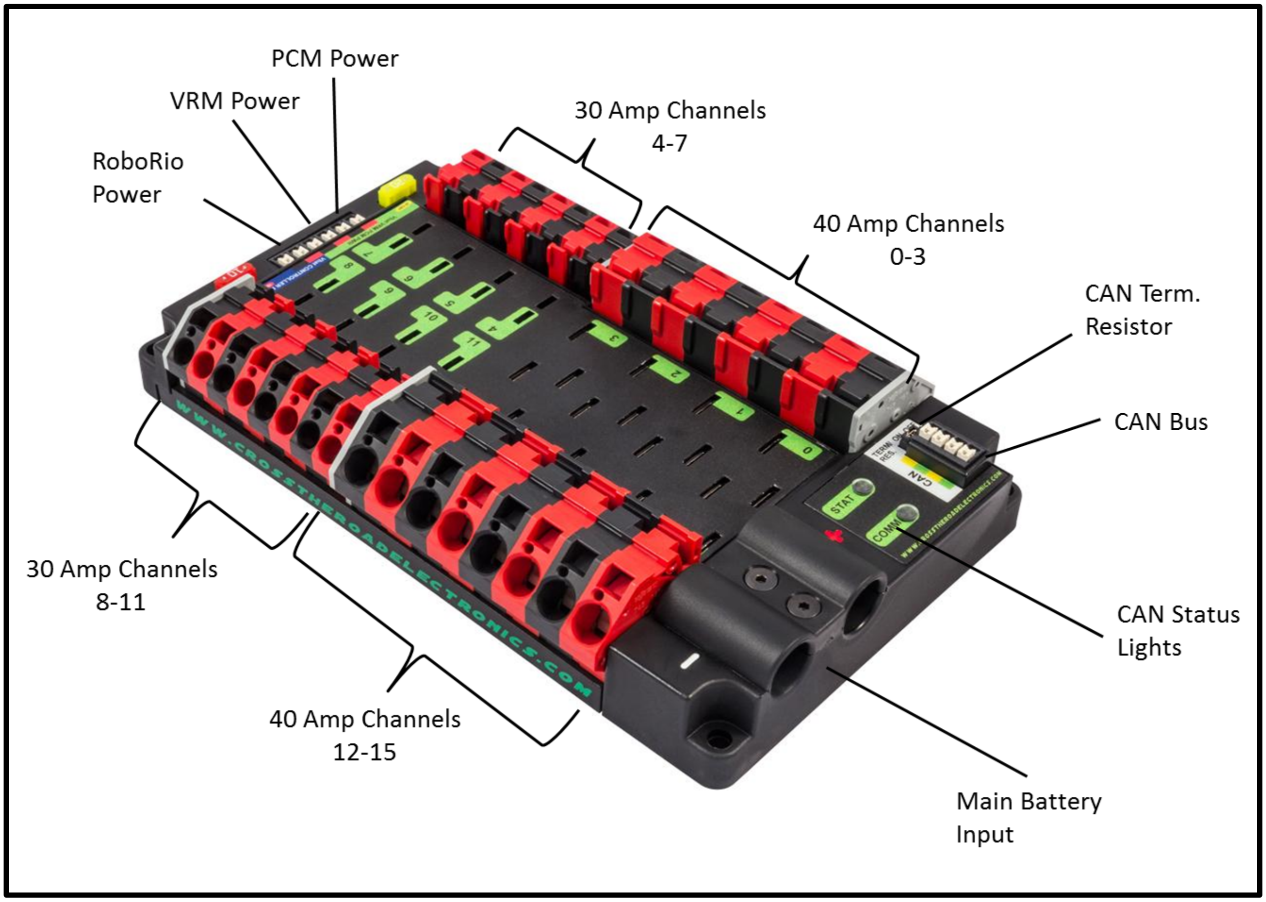


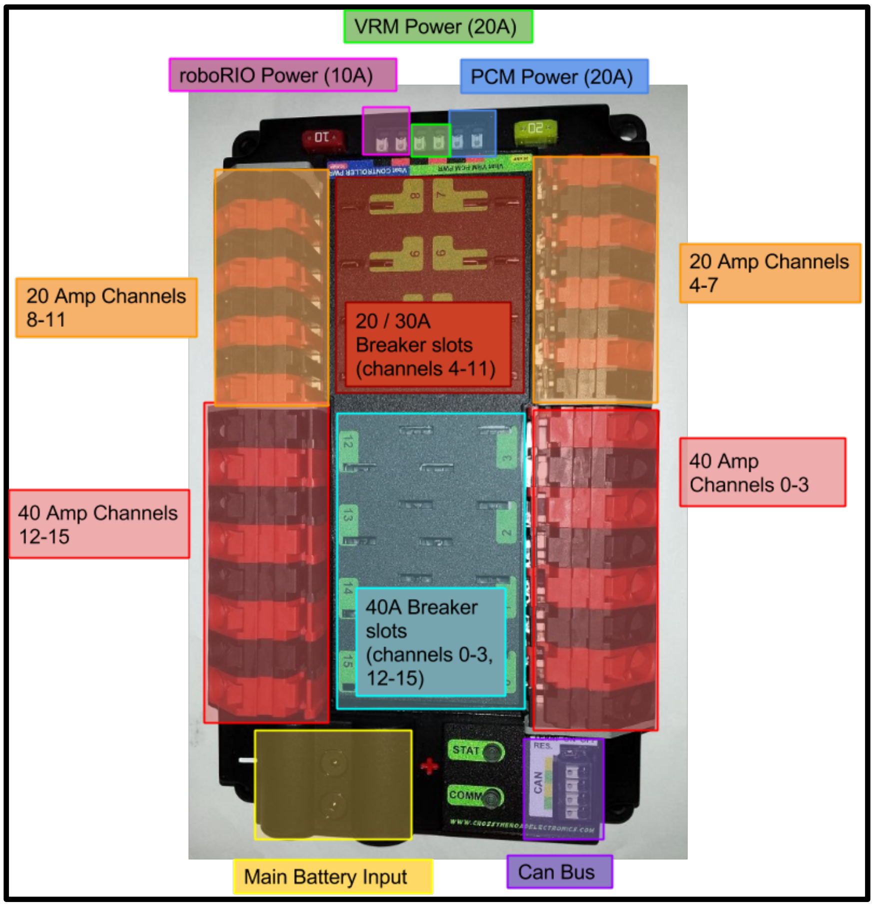
Image courtesy of http://khengineering.github.io/RoboRio/faq/pdp/

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.

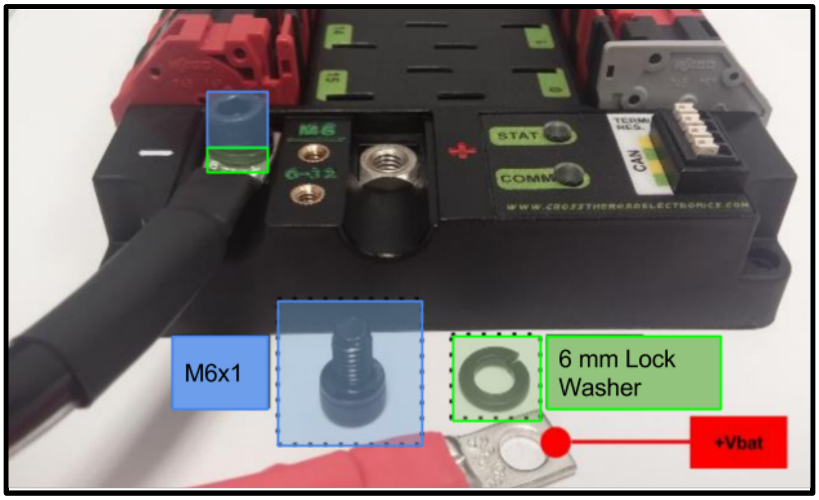
Specs

- 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

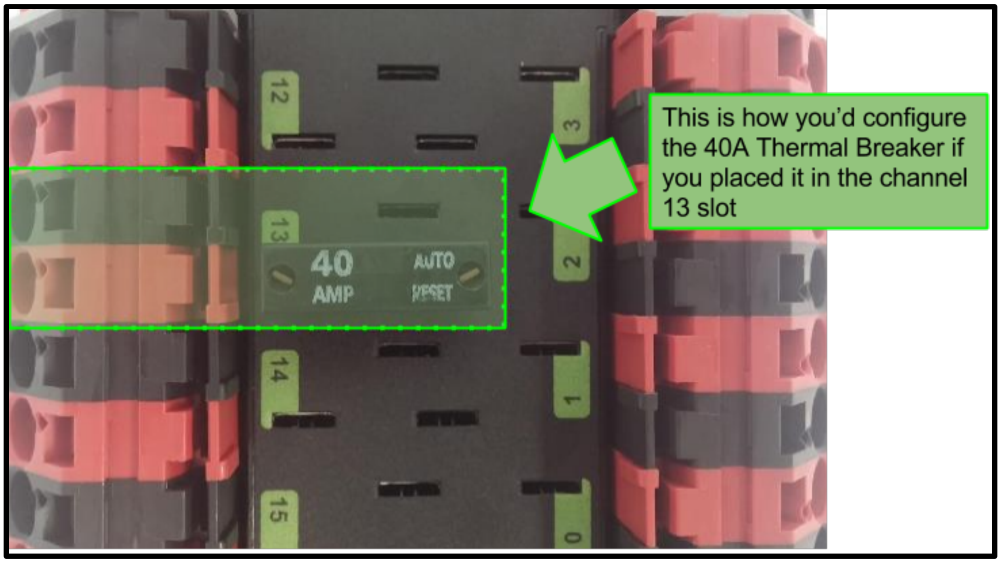
➠ Wiring



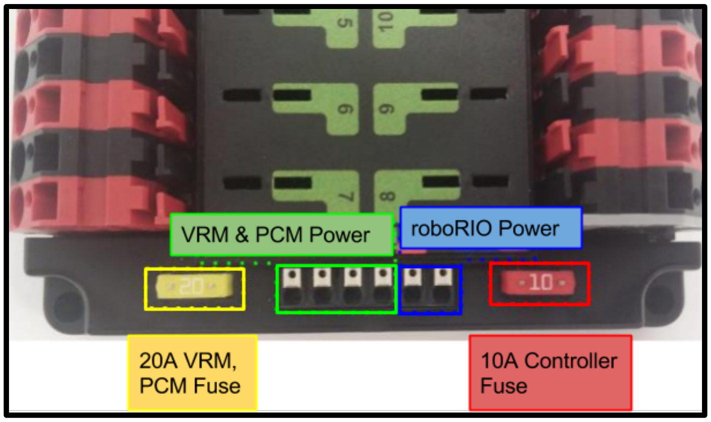
Main Battery Input



40 Amp Thermal Breaker by Snap Action slots (channels 0-3, 12-15)

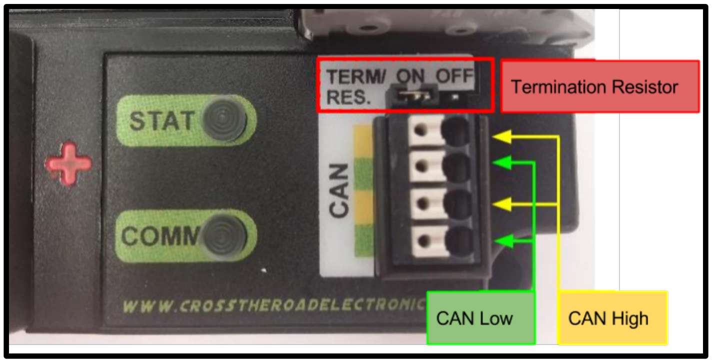

This is how you'd configure the 40A Thermal Breaker if you placed it in the channel 13 slot
12
13
14
15
3
2
1
0
40 AMP
AUTO RESET

6 Weidmuller connectors for PCM, VRM, and roboRIO



4 CAN connectors, Terminal Resistor Jumper, and the Status Lights

Only place Terminal Resistor Jumper when the PDP is at the end of the CAN bus.



The Voltage Regulator Module

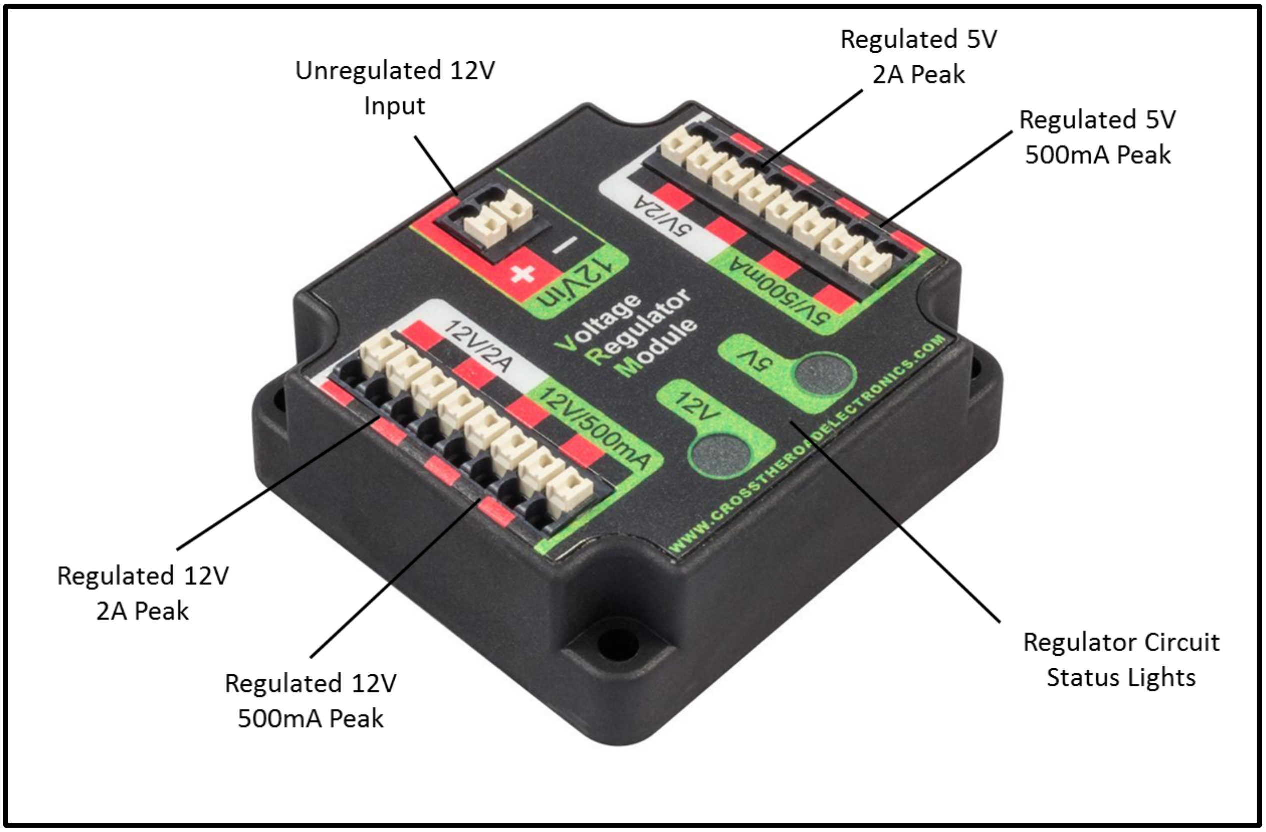


Image courtesy of http://khengineering.github.io/RoboRio/Images/vrminfo.png

The Voltage Regulator Module (VRM) is a new component introduced in the 2015 FRC season as part of the Kit of Parts. This replaces the 12V and 5V regulator and special protected and unprotected power outputs on the old PDB. The purpose of the VRM is to act as a DC to DC converter, and both boost and buck voltage regulation. The VRM is required when using components that require special protected power.

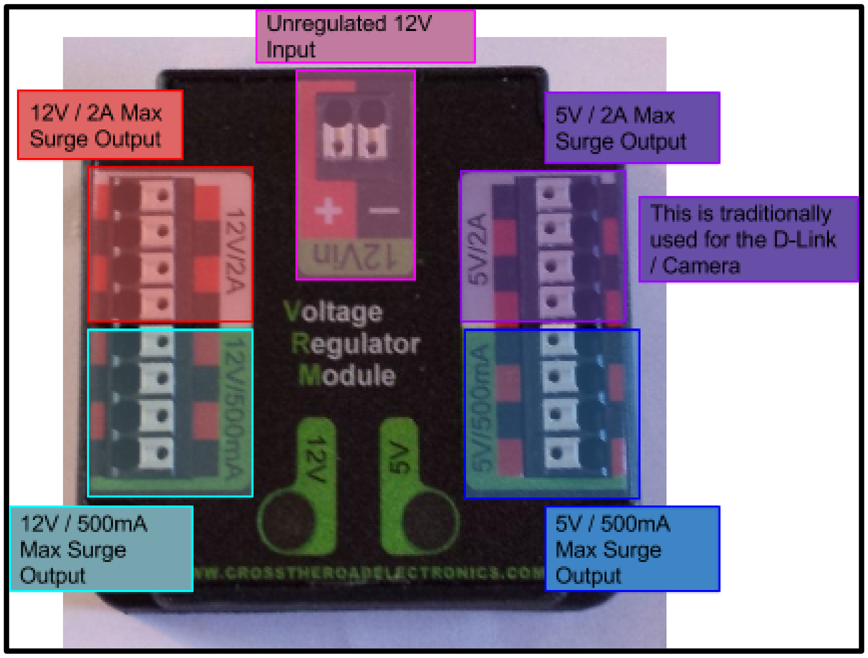
Specs

- Dimensions: 2.220” x 2.030” x 0.784” Tall
- Weight: 1.8 oz
- 4 mounting holes (one at each corner, 6-32 fasteners)
- Connectors (all use Weidmuller connectors, accepting 24 - 16 AWG)
  - 4 x connectors for regulated 12V / 2A max surge / 1.5A limit power output
  - 4 x connectors for regulated 12V / 500mA max surge power output
  - 4 x connectors for regulated 5V / 2A max surge / 1.5A limit power output
  - 4 x connectors for regulated 5V / 500mA max surge power output
  - 2 x connectors for unregulated 12V input
- Status Lights
  - 5V/500mA Status LED
    - On - 5V channels are powered
    - Off - Breaker has tripped
  - 12V/500mA Status LED

        - On - 12V channels are powered
        - Off - Breaker has tripped
    - LEDs not affected when 2A channels are limited
    - Flickering - Low voltage (<4V)

➠ Wiring

All ports use Weidmuller connectors, accepting 24 - 16 AWG. There are two separate power supplies, 12V and 5V. There is just enough power to run both a D-Link and Camera on the 5V rail, but you may need to use a separate 12V to 5V adapter or use a USB camera.


Unregulated 12V Input
12V / 2A Max Surge Output
5V / 2A Max Surge Output
This is traditionally used for the D-Link / Camera
12V/2A
12Vin
5V/2A
12V/500mA
5V/500mA
Voltage Regulator Module
12V
5V
WW.CROSSTHEROADELECTRONICS.COM
12V / 500mA Max Surge Output
5V / 500mA Max Surge Output

The D-Link

Jump to →

Introduction

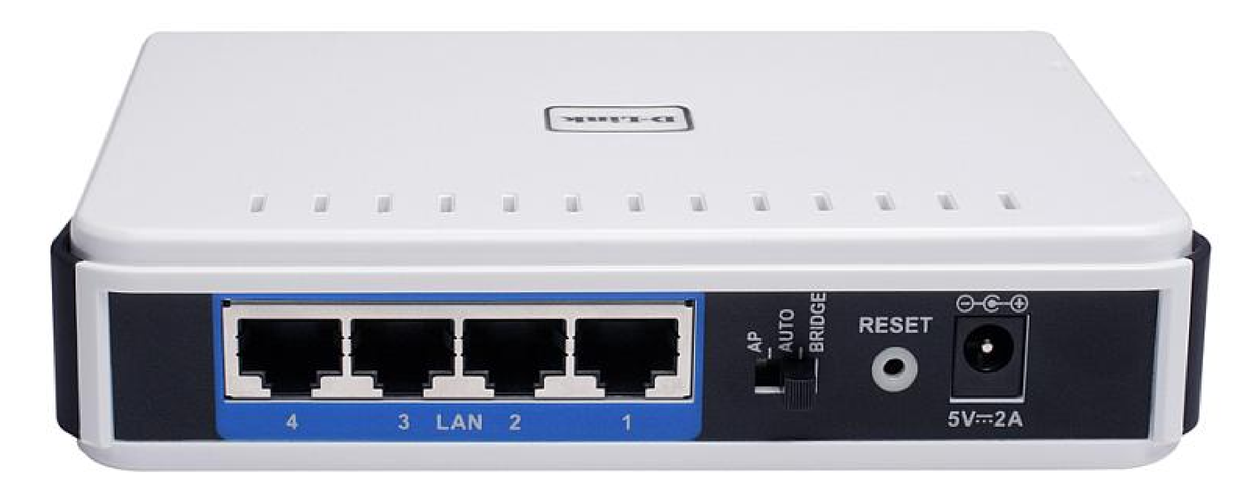
We control the robot through a **Logitech gamepad controller**. Generally, we wouldn't want to follow the robot around with the controller because the cable's short and plugged into the robot, so we solve this problem by connecting wirelessly. We use the **D-Link** as a medium to communicate with the 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.

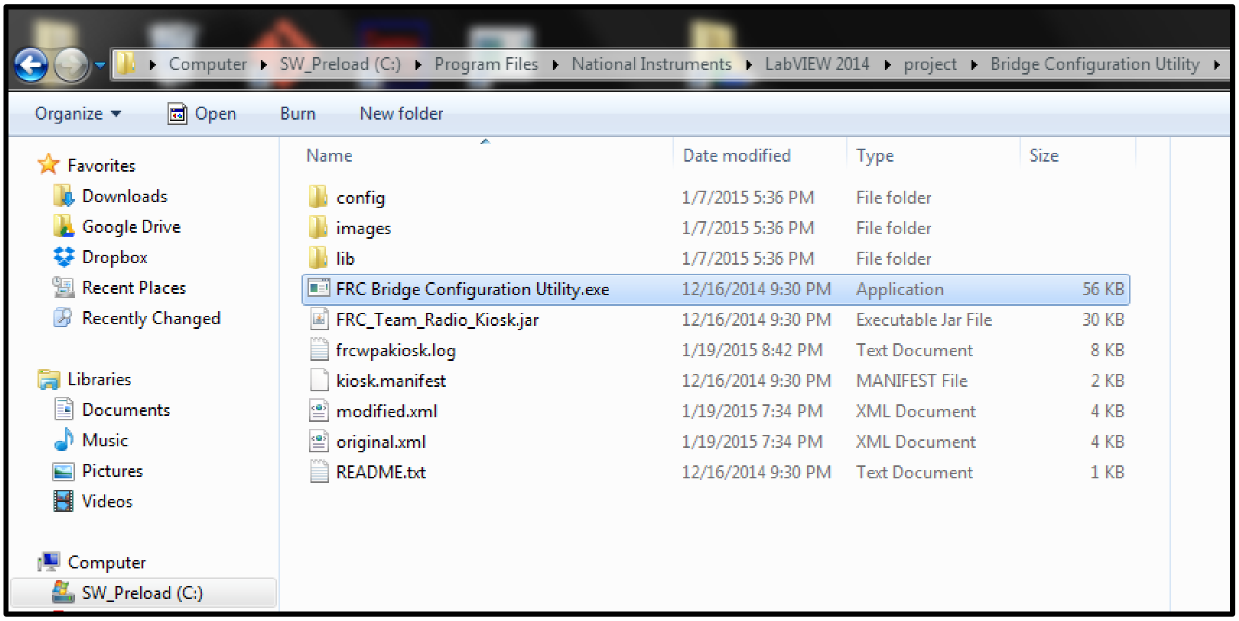
Configuration(For a Brand New or Reset Router)

➠ 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

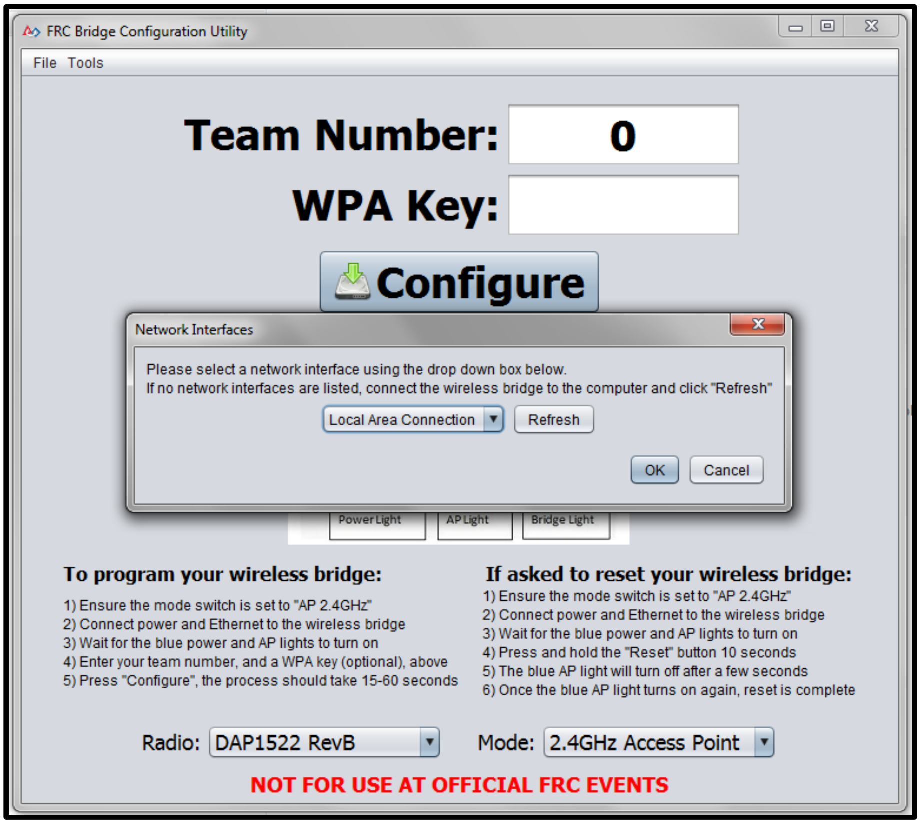
➠ FRC Bridge Configuration Utility

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



Name	Date modified	Type	Size
config	1/7/2015 5:36 PM	File folder	
images	1/7/2015 5:36 PM	File folder	
lib	1/7/2015 5:36 PM	File folder	
FRC Bridge Configuration Utility.exe	12/16/2014 9:30 PM	Application	56 KB
FRC_Team_Radio_Kiosk.jar	12/16/2014 9:30 PM	Executable Jar File	30 KB
frcwpakiosk.log	1/19/2015 8:42 PM	Text Document	8 KB
kiosk.manifest	12/16/2014 9:30 PM	MANIFEST File	2 KB
modified.xml	1/19/2015 7:34 PM	XML Document	4 KB
original.xml	1/19/2015 7:34 PM	XML Document	4 KB
README.txt	12/16/2014 9:30 PM	Text Document	1 KB

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.



Team Number: 0

WPA Key:

Please select a network interface using the drop down box below.
If no network interfaces are listed, connect the wireless bridge to the computer and click "Refresh"

Local Area Connection

To program your wireless bridge:

1) Ensure the mode switch is set to "AP 2.4GHz"
2) Connect power and Ethernet to the wireless bridge
3) Wait for the blue power and AP lights to turn on
4) Enter your team number, and a WPA key (optional), above
5) Press "Configure", the process should take 15-60 seconds

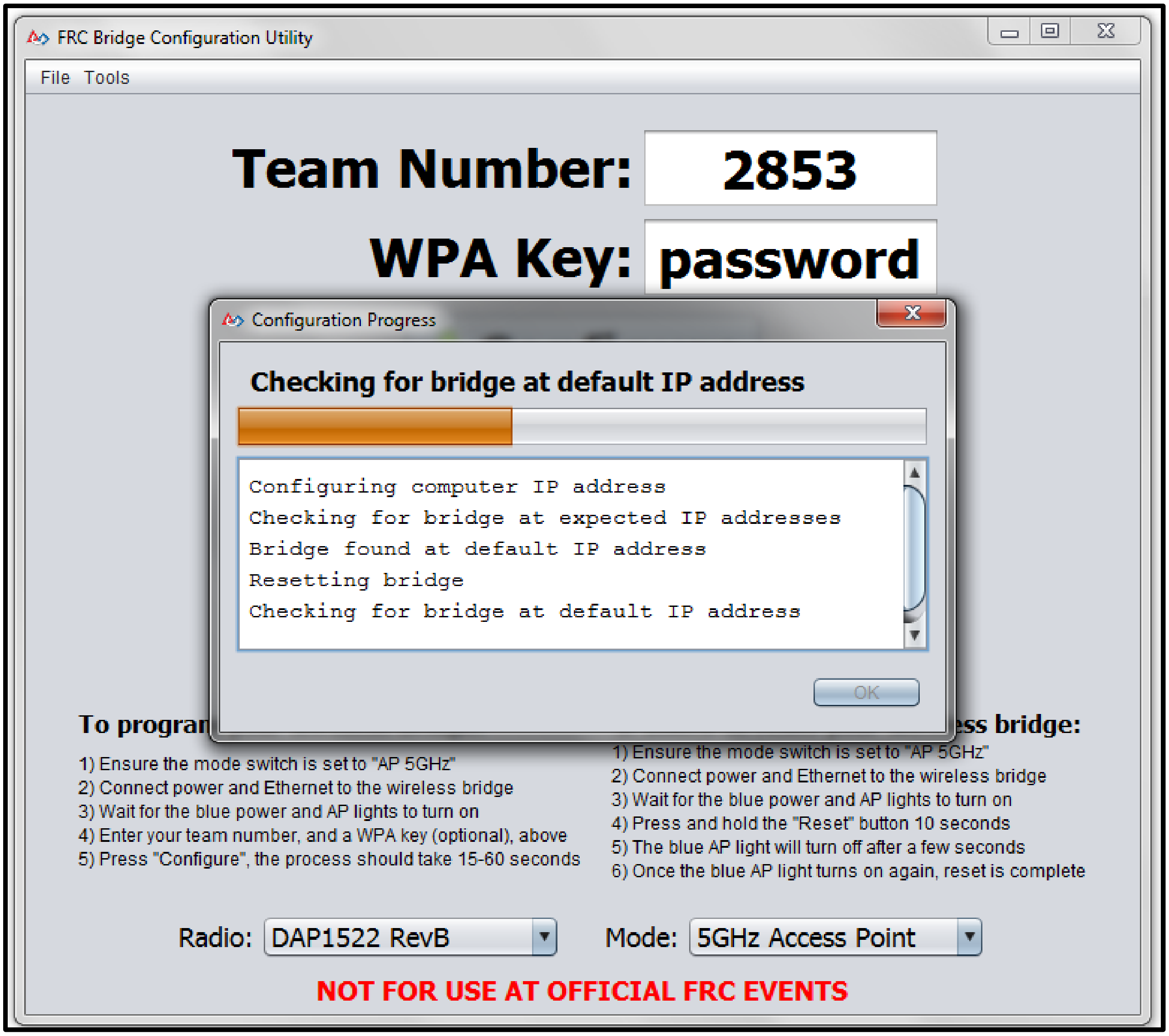
If asked to reset your wireless bridge:

1) Ensure the mode switch is set to "AP 2.4GHz"
2) Connect power and Ethernet to the wireless bridge
3) Wait for the blue power and AP lights to turn on
4) Press and hold the "Reset" button 10 seconds
5) The blue AP light will turn off after a few seconds
6) Once the blue AP light turns on again, reset is complete

Radio: DAP1522 RevB Mode: 2.4GHz Access Point

NOT FOR USE AT OFFICIAL FRC EVENTS

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.”



Team Number: 2853

WPA Key: password

Checking for bridge at default IP address

```
Configuring computer IP address
Checking for bridge at expected IP addresses
Bridge found at default IP address
Resetting bridge
Checking for bridge at default IP address
```

To progran

1) Ensure the mode switch is set to "AP 5GHz"
2) Connect power and Ethernet to the wireless bridge
3) Wait for the blue power and AP lights to turn on
4) Enter your team number, and a WPA key (optional), above
5) Press "Configure", the process should take 15-60 seconds

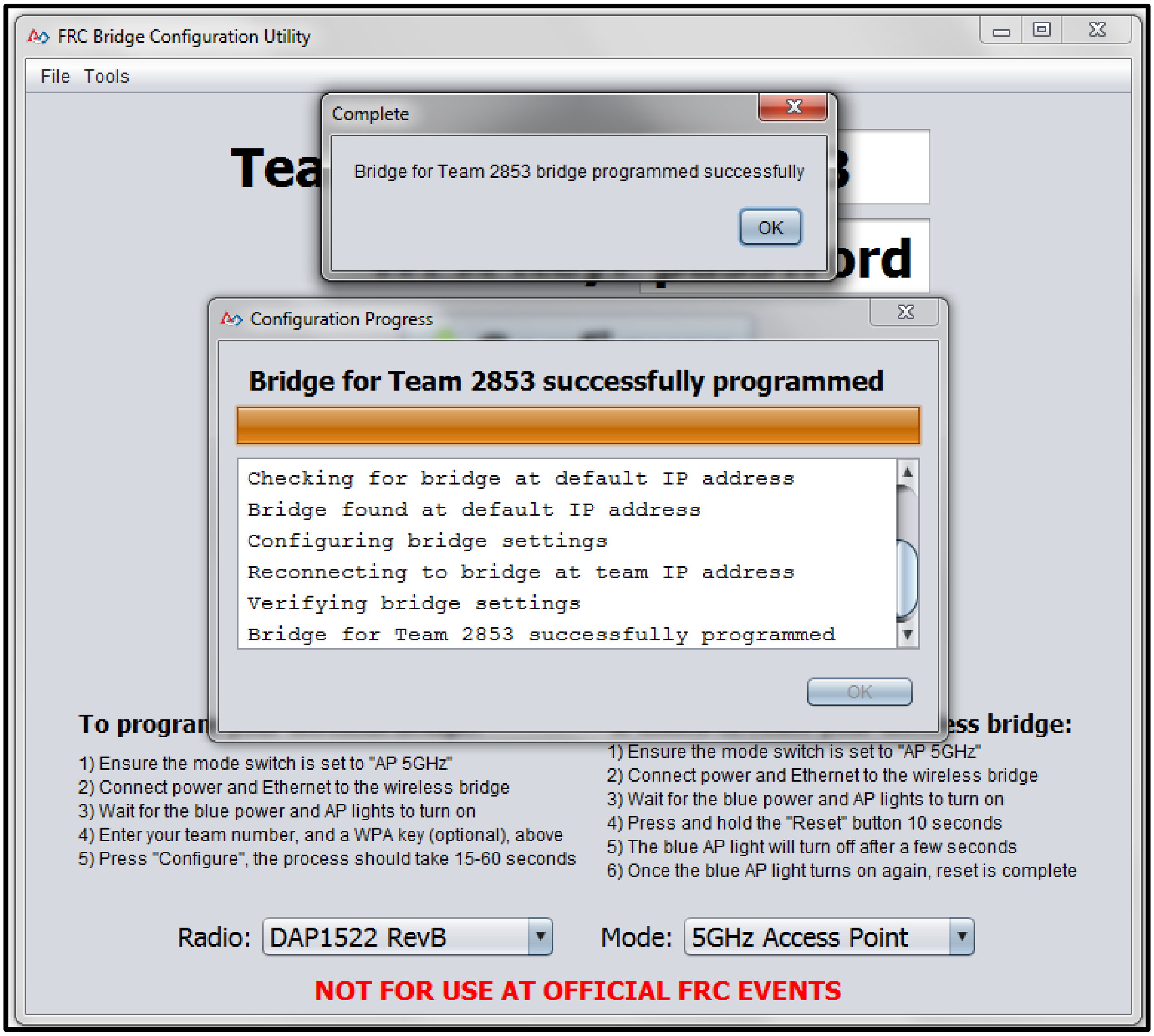
ss bridge:

1) Ensure the mode switch is set to "AP 5GHz"
2) Connect power and Ethernet to the wireless bridge
3) Wait for the blue power and AP lights to turn on
4) Press and hold the "Reset" button 10 seconds
5) The blue AP light will turn off after a few seconds
6) Once the blue AP light turns on again, reset is complete

Radio: DAP1522 RevB ▾ Mode: 5GHz Access Point ▾

NOT FOR USE AT OFFICIAL FRC EVENTS

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



Bridge for Team 2853 bridge programmed successfully

Bridge for Team 2853 successfully programmed

```
Checking for bridge at default IP address
Bridge found at default IP address
Configuring bridge settings
Reconnecting to bridge at team IP address
Verifying bridge settings
Bridge for Team 2853 successfully programmed
```

To progran

1) Ensure the mode switch is set to "AP 5GHz"
2) Connect power and Ethernet to the wireless bridge
3) Wait for the blue power and AP lights to turn on
4) Enter your team number, and a WPA key (optional), above
5) Press "Configure", the process should take 15-60 seconds

ss bridge:

1) Ensure the mode switch is set to "AP 5GHz"
2) Connect power and Ethernet to the wireless bridge
3) Wait for the blue power and AP lights to turn on
4) Press and hold the "Reset" button 10 seconds
5) The blue AP light will turn off after a few seconds
6) Once the blue AP light turns on again, reset is complete

Radio: DAP1522 RevB Mode: 5GHz Access Point

NOT FOR USE AT OFFICIAL FRC EVENTS

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

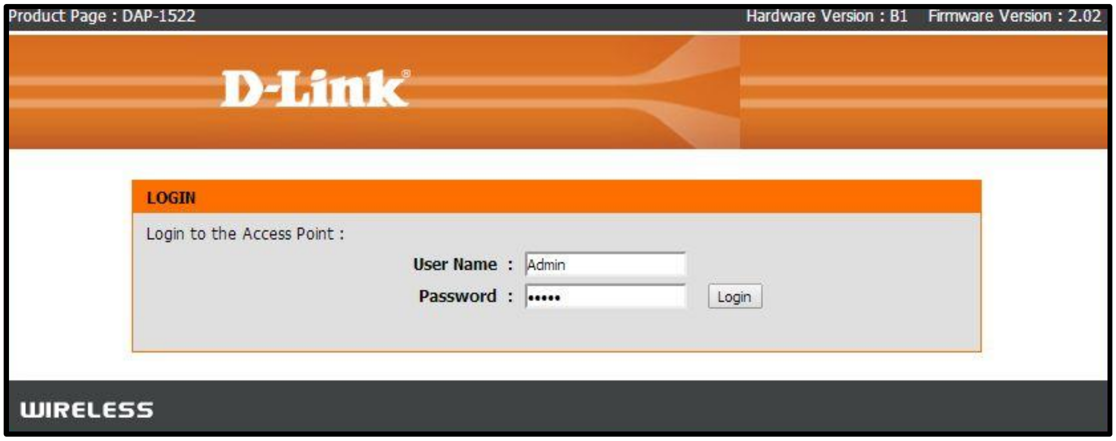
OR 10.xx.yy.1, where **xx** is the first two digits of your team's # (it can be one digit if your team has a three-digit #) and **yy** is the last two

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)



Product Page : DAP-1522

Hardware Version : B1 Firmware Version : 2.02

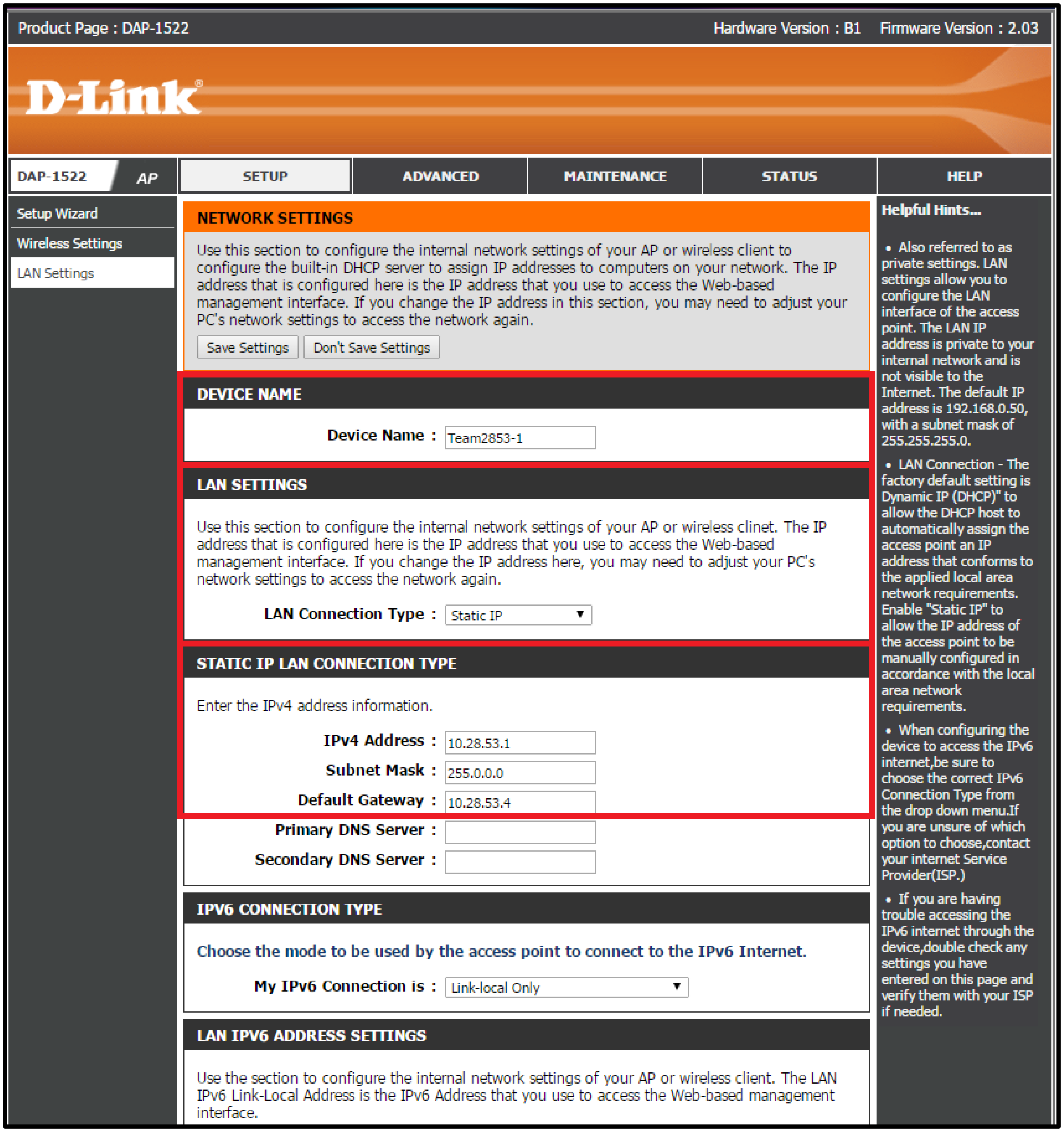
LOGIN

Login to the Access Point :

User Name : Admin

Password : •••••

➠ Setup: LAN Settings



NETWORK SETTINGS

Use this section to configure the internal network settings of your AP or wireless client to configure the built-in DHCP server to assign IP addresses to computers on your network. The IP address that is configured here is the IP address that you use to access the Web-based management interface. If you change the IP address in this section, you may need to adjust your PC's network settings to access the network again.

DEVICE NAME

Device Name : Team2853-1

LAN SETTINGS

Use this section to configure the internal network settings of your AP or wireless clinet. The IP address that is configured here is the IP address that you use to access the Web-based management interface. If you change the IP address here, you may need to adjust your PC's network settings to access the network again.

LAN Connection Type : Static IP ▼

STATIC IP LAN CONNECTION TYPE

Enter the IPv4 address information.

IPv4 Address : 10.28.53.1
Subnet Mask : 255.0.0.0
Default Gateway : 10.28.53.4
Primary DNS Server :
Secondary DNS Server :

IPV6 CONNECTION TYPE

Choose the mode to be used by the access point to connect to the IPv6 Internet.

My IPv6 Connection is : Link-local Only ▼

LAN IPV6 ADDRESS SETTINGS

Use the section to configure the internal network settings of your AP or wireless client. The LAN IPv6 Link-Local Address is the IPv6 Address that you use to access the Web-based management interface.

Helpful Hints...

- Also referred to as private settings. LAN settings allow you to configure the LAN interface of the access point. The LAN IP address is private to your internal network and is not visible to the Internet. The default IP address is 192.168.0.50, with a subnet mask of 255.255.255.0.
- LAN Connection - The factory default setting is Dynamic IP (DHCP)" to allow the DHCP host to automatically assign the access point an IP address that conforms to the applied local area network requirements. Enable "Static IP" to allow the IP address of the access point to be manually configured in accordance with the local area network requirements.
- When configuring the device to access the IPv6 internet,be sure to choose the correct IPv6 Connection Type from the drop down menu.If you are unsure of which option to choose,contact your internet Service Provider(ISP.)
- If you are having trouble accessing the IPv6 internet through the device,double check any settings you have entered on this page and verify them with your ISP if needed.

➠ 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.

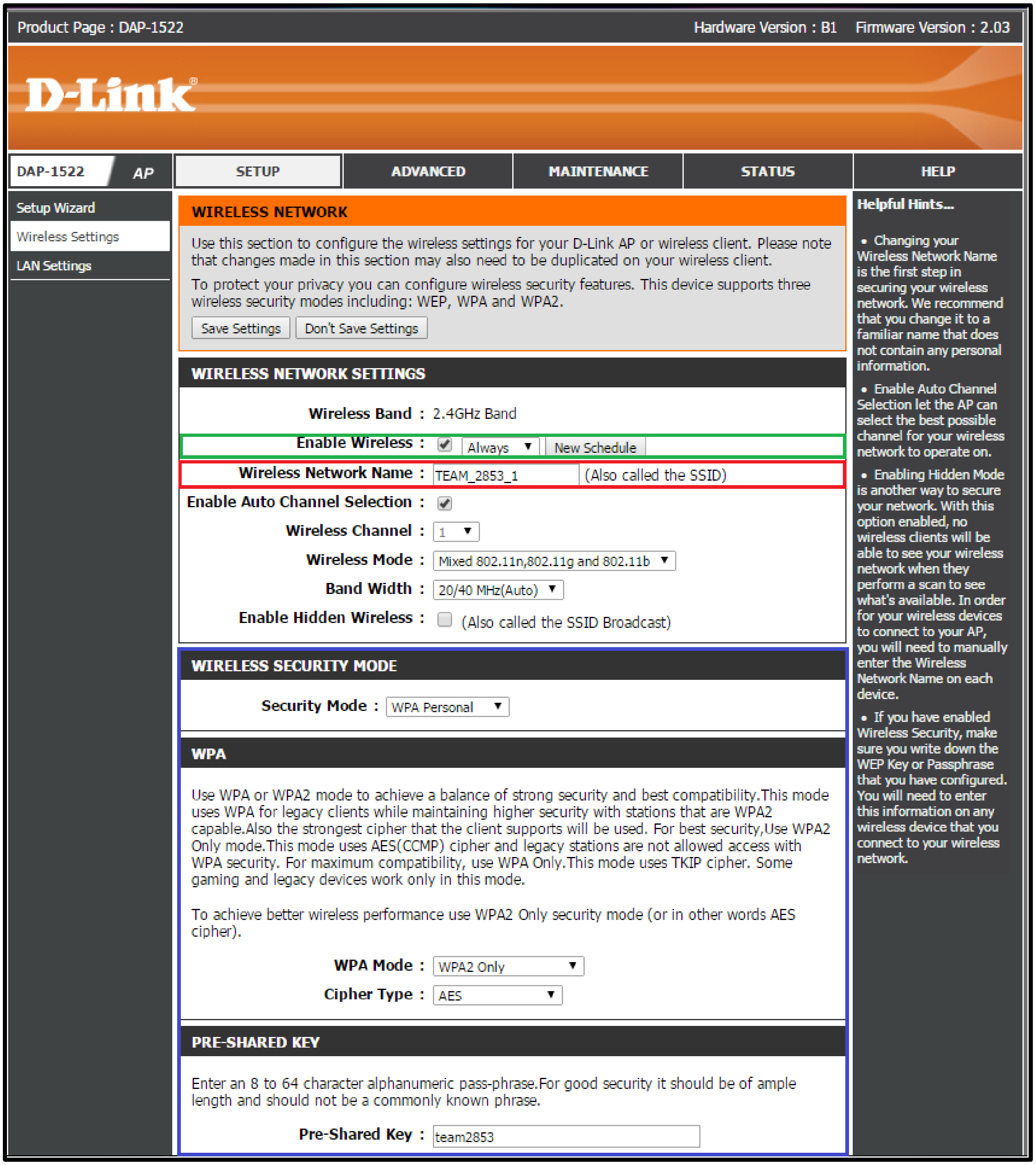
Example Device Name: **Team2853-1**

➠ 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



WIRELESS NETWORK

Use this section to configure the wireless settings for your D-Link AP or wireless client. Please note that changes made in this section may also need to be duplicated on your wireless client.

To protect your privacy you can configure wireless security features. This device supports three wireless security modes including: WEP, WPA and WPA2.

WIRELESS NETWORK SETTINGS

Wireless Band : 2.4GHz Band

Enable Wireless : ☑ Always ▼ New Schedule

Wireless Network Name : TEAM_2853_1 (Also called the SSID)

Enable Auto Channel Selection : ☑

Wireless Channel : 1 ▼

Wireless Mode : Mixed 802.11n,802.11g and 802.11b ▼

Band Width : 20/40 MHz(Auto) ▼

Enable Hidden Wireless : □ (Also called the SSID Broadcast)

WIRELESS SECURITY MODE

Security Mode : WPA Personal ▼

WPA

Use WPA or WPA2 mode to achieve a balance of strong security and best compatibility.This mode uses WPA for legacy clients while maintaining higher security with stations that are WPA2 capable.Also the strongest cipher that the client supports will be used. For best security,Use WPA2 Only mode.This mode uses AES(CCMP) cipher and legacy stations are not allowed access with WPA security. For maximum compatibility, use WPA Only.This mode uses TKIP cipher. Some gaming and legacy devices work only in this mode.

To achieve better wireless performance use WPA2 Only security mode (or in other words AES cipher).

WPA Mode : WPA2 Only ▼

Cipher Type : AES ▼

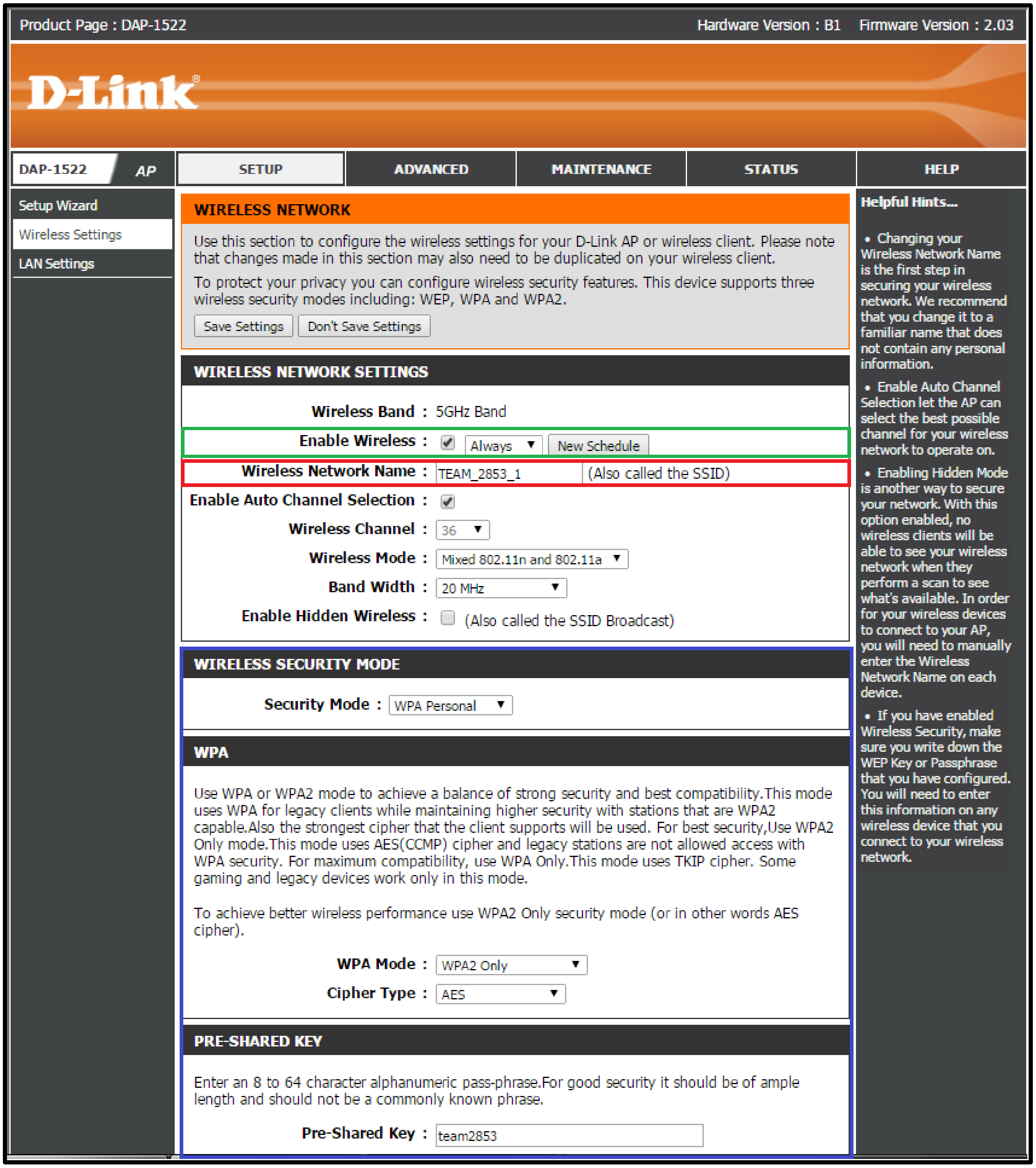
PRE-SHARED KEY

Enter an 8 to 64 character alphanumeric pass-phrase.For good security it should be of ample length and should not be a commonly known phrase.

Pre-Shared Key : team2853

Helpful Hints...

- Changing your Wireless Network Name is the first step in securing your wireless network. We recommend that you change it to a familiar name that does not contain any personal information.
- Enable Auto Channel Selection let the AP can select the best possible channel for your wireless network to operate on.
- Enabling Hidden Mode is another way to secure your network. With this option enabled, no wireless clients will be able to see your wireless network when they perform a scan to see what's available. In order for your wireless devices to connect to your AP, you will need to manually enter the Wireless Network Name on each device.
- If you have enabled Wireless Security, make sure you write down the WEP Key or Passphrase that you have configured. You will need to enter this information on any wireless device that you connect to your wireless network.



WIRELESS NETWORK

Use this section to configure the wireless settings for your D-Link AP or wireless client. Please note that changes made in this section may also need to be duplicated on your wireless client.

To protect your privacy you can configure wireless security features. This device supports three wireless security modes including: WEP, WPA and WPA2.

WIRELESS NETWORK SETTINGS

Wireless Band :	5GHz Band
Enable Wireless :	☑ Always ▼
Wireless Network Name :	TEAM_2853_1 (Also called the SSID)
Enable Auto Channel Selection :	☑
Wireless Channel :	36 ▼
Wireless Mode :	Mixed 802.11n and 802.11a ▼
Band Width :	20 MHz ▼
Enable Hidden Wireless :	□ (Also called the SSID Broadcast)

WIRELESS SECURITY MODE

Security Mode : WPA Personal ▼

WPA

Use WPA or WPA2 mode to achieve a balance of strong security and best compatibility.This mode uses WPA for legacy clients while maintaining higher security with stations that are WPA2 capable.Also the strongest cipher that the client supports will be used. For best security,Use WPA2 Only mode.This mode uses AES(CCMP) cipher and legacy stations are not allowed access with WPA security. For maximum compatibility, use WPA Only.This mode uses TKIP cipher. Some gaming and legacy devices work only in this mode.

To achieve better wireless performance use WPA2 Only security mode (or in other words AES cipher).

WPA Mode :	WPA2 Only ▼
Cipher Type :	AES ▼

PRE-SHARED KEY

Enter an 8 to 64 character alphanumeric pass-phrase.For good security it should be of ample length and should not be a commonly known phrase.

Pre-Shared Key : team2853

Helpful Hints...

- Changing your Wireless Network Name is the first step in securing your wireless network. We recommend that you change it to a familiar name that does not contain any personal information.
- Enable Auto Channel Selection let the AP can select the best possible channel for your wireless network to operate on.
- Enabling Hidden Mode is another way to secure your network. With this option enabled, no wireless clients will be able to see your wireless network when they perform a scan to see what's available. In order for your wireless devices to connect to your AP, you will need to manually enter the Wireless Network Name on each device.
- If you have enabled Wireless Security, make sure you write down the WEP Key or Passphrase that you have configured. You will need to enter this information on any wireless device that you connect to your wireless network.

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

Jump to →

How to Use 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

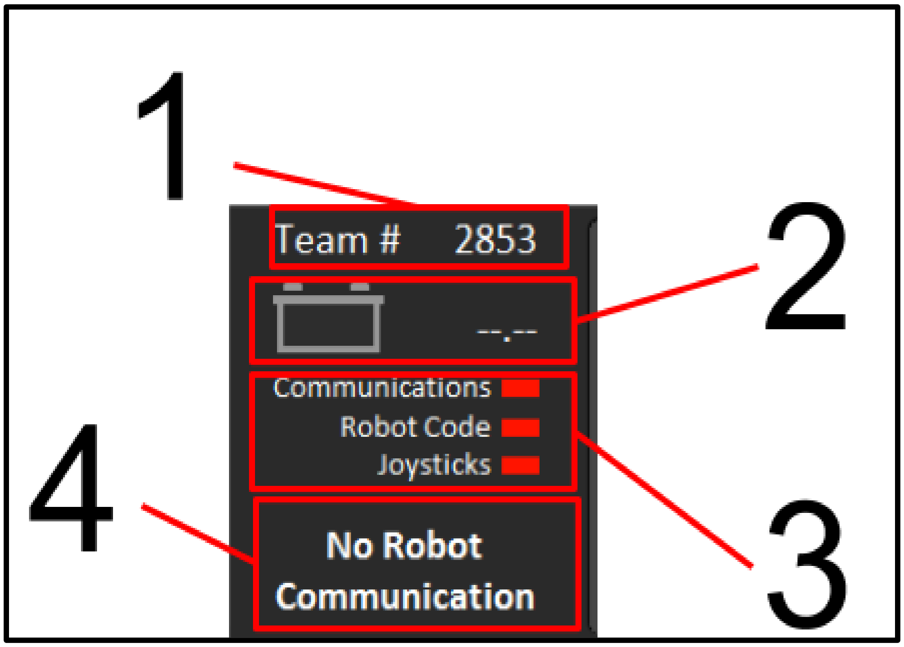
This is the program that allows to to test and use your code on FRC components. The Driver Station is the middle-man between you and the robot!, this is your middle man between your code and the robot!

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.

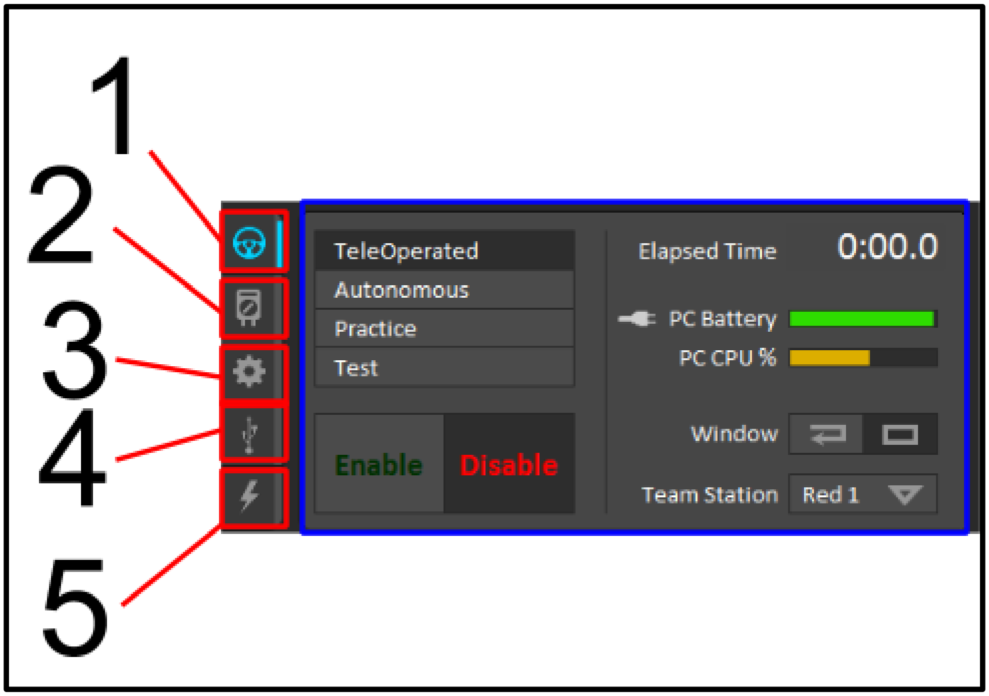
The Interface

➠ 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)



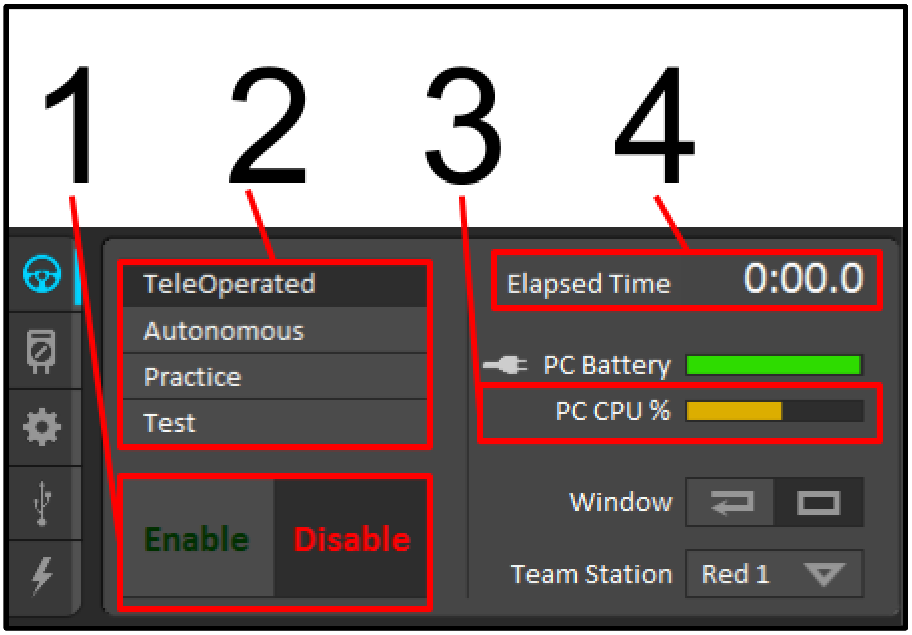
1. Your Team #, this can be configured personally in the Setup Tab (See Tab Selection Section : ➠Setup Tab)
2. Current voltage of the battery in V(volts) has a visual indicator to the left in the form of a pictorial representation of a battery(fills like a bar to represent amount of charge).
3. Lights 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 Station starts).

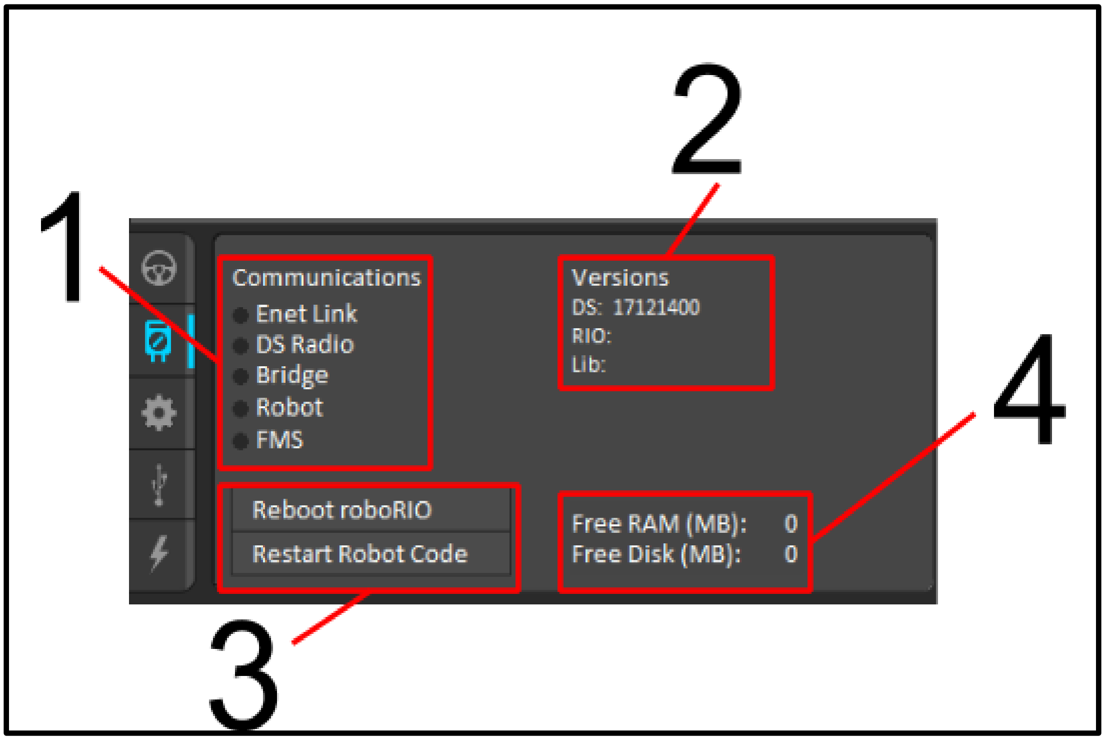
➠ Operation Tab



1. 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
4. Elapsed Time since you clicked “Enable” until disabled

➠ Diagnostics Tab

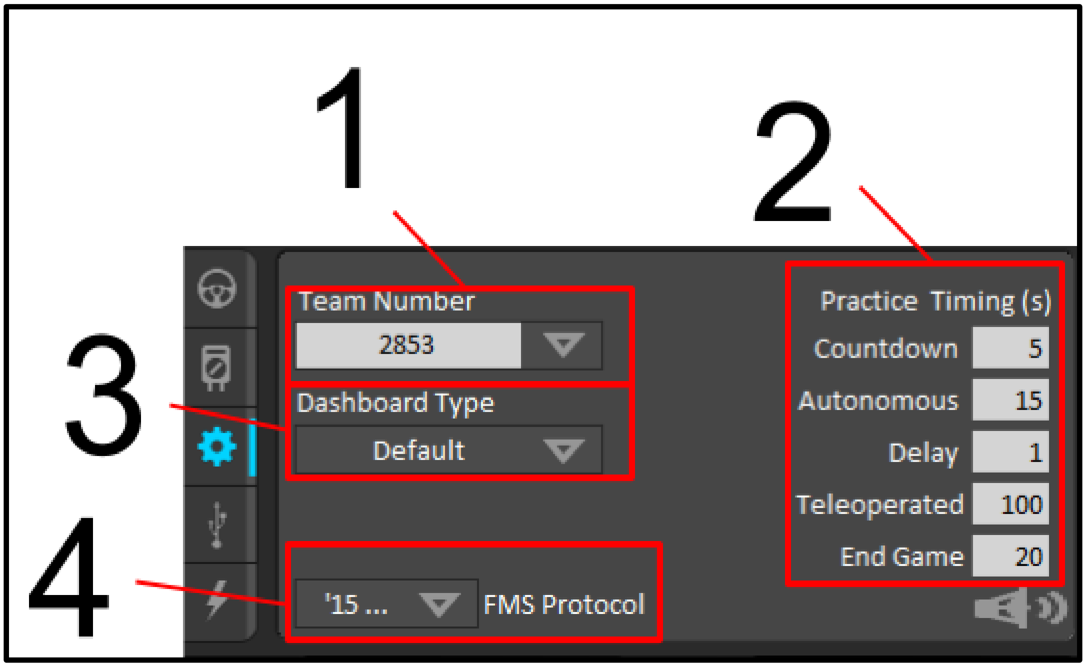


Versions
DS: 17121400
RIO:
Lib:

Free RAM (MB): 0
Free Disk (MB): 0

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



Team Number
2853

Dashboard Type
Default

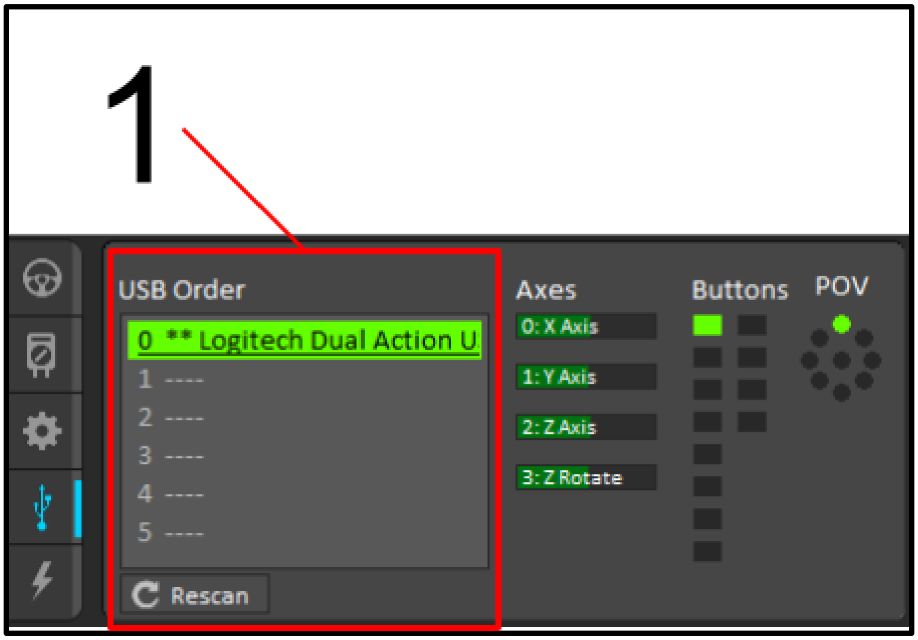
Practice Timing (s)

Countdown	5
Autonomous	15
Delay	1
Teleoperated	100
End Game	20

'15 ... FMS Protocol

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

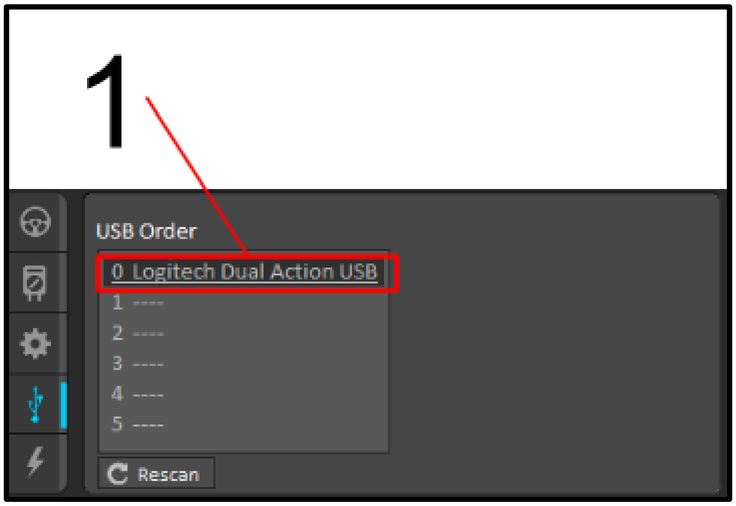
➠ USB Devices Tab



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.

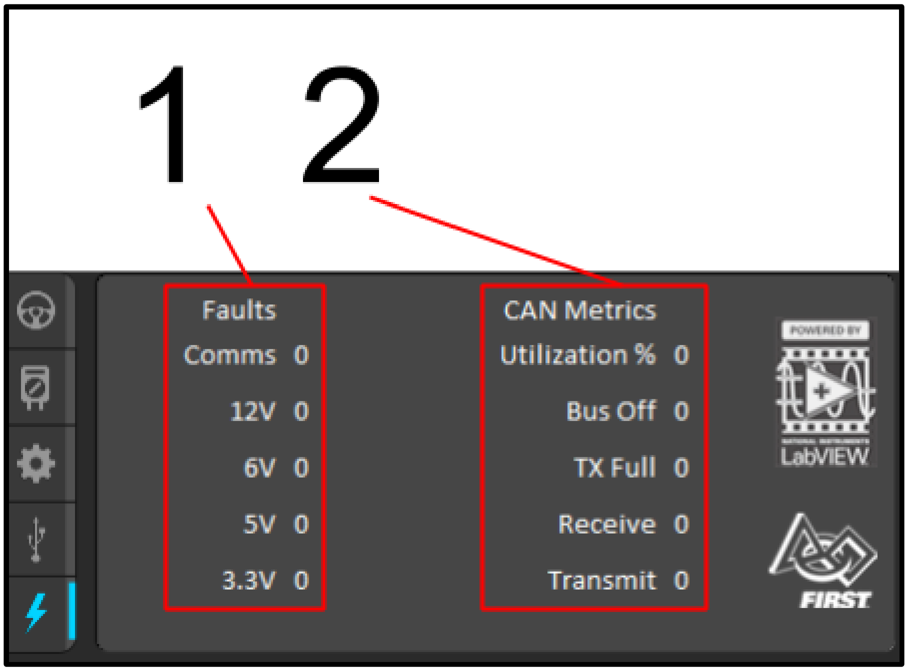
Locking & Rearranging

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.



USB Order
0 Logitech Dual Action USB
1 ----
2 ----
3 ----
4 ----
5 ----

➠ **Power & CAN**

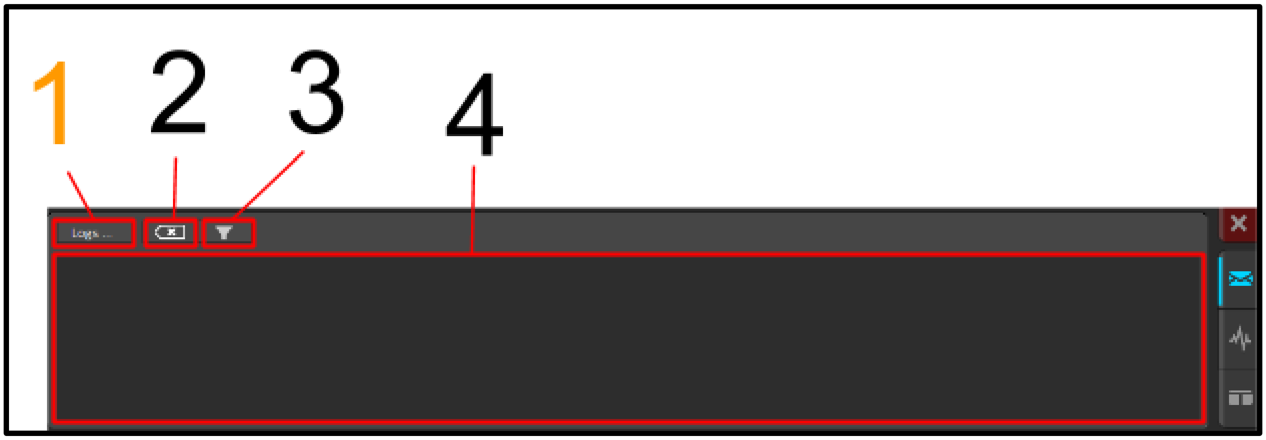


Faults	
Comms	0
12V	0
6V	0
5V	0
3.3V	0

CAN Metrics	
Utilization %	0
Bus Off	0
TX Full	0
Receive	0
Transmit	0

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

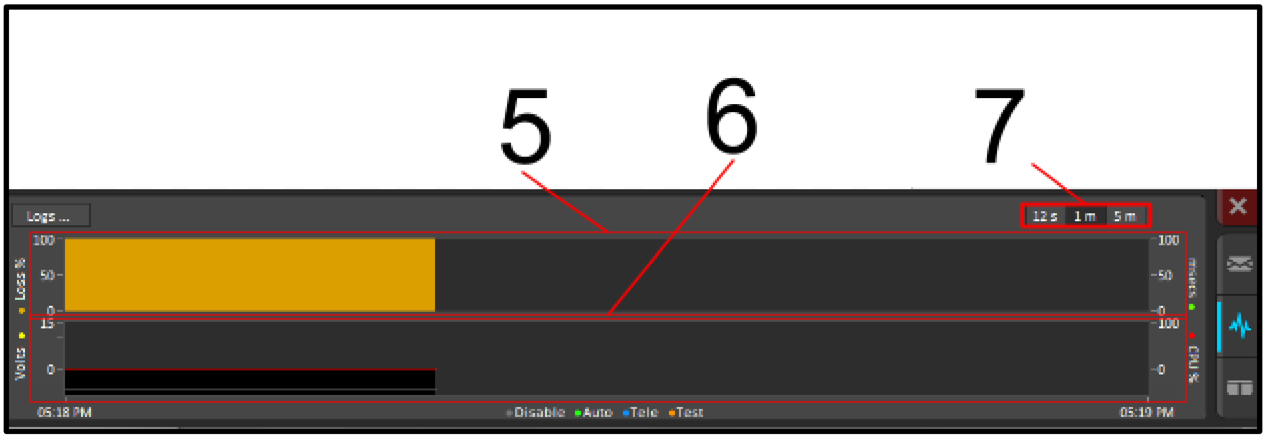
➠ Messages & Charts



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

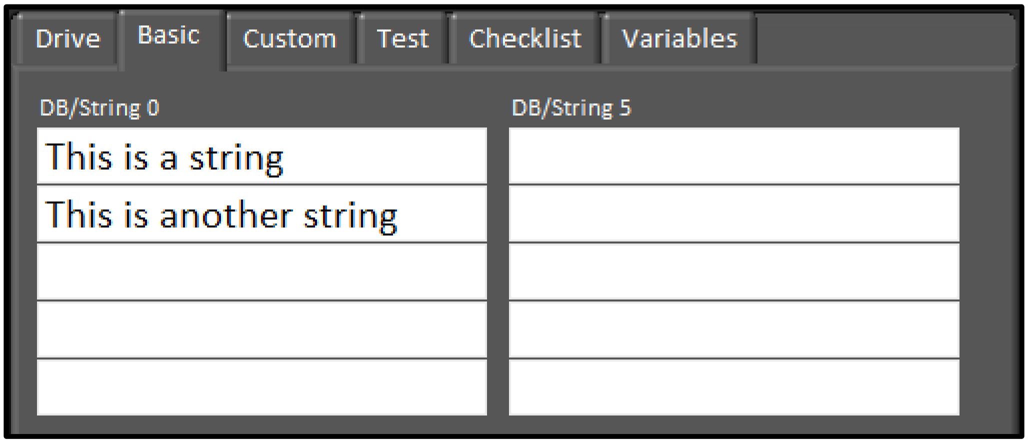
Loads up the DS Log File Viewer, you can view messages & event data in one display, a tool usually used for troubleshooting. This tool has its own section to be added elsewhere.



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 (Untested)

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.



DB/String 0

This is a string

This is another string

DB/String 5

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++

Jump to →

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.

+	Addition
-	Subtraction
/	Division
*	Multiplication
%	Modulus
=	Assignment Operator
( )	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 (==).

TABLE 3. Relational operators.

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 >= 50;	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.

TABLE 4. Logical operators.

OPERATOR	**SYMBOL**	**EXAMPLE**
AND	&&	expression1 && expression2
OR	| |	expression1 | | expression2
NOT	!	!expression

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 (UINT8 modulenumber, 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, 2)
```

Where 1 is the number of a UINT8 slot (specifically, the module slot on the CRIO) and 2 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 Jaguar class, here are the following constructors for a Jaguar object.

```
Jaguar(UINT32 channel) //Constructor that assumes the default digital
                            module.
Jaguar (UINT32 slot, UINT32 channel) //Constructor that specifies the
                                           digital module.
```

You should see that the only thing different is the parameters that must be called. The first calls for the PWM channel only (assumes that the cRIO module as 1). To call the **first** constructor after defining **jag1**, you would type:

```
        jag1(1)
```

which means **jag1** is assigned to PWM port 1 on the digital sidecar and is automatically assumed to be on the first digital sidecar module.

Because the fourth cRIO module slot can support any module, anything connected to the **second** digital sidecar will have to use the latter constructor which calls for the PWM channel *and* the cRIO module number the sidecar is connected to.

➠ 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

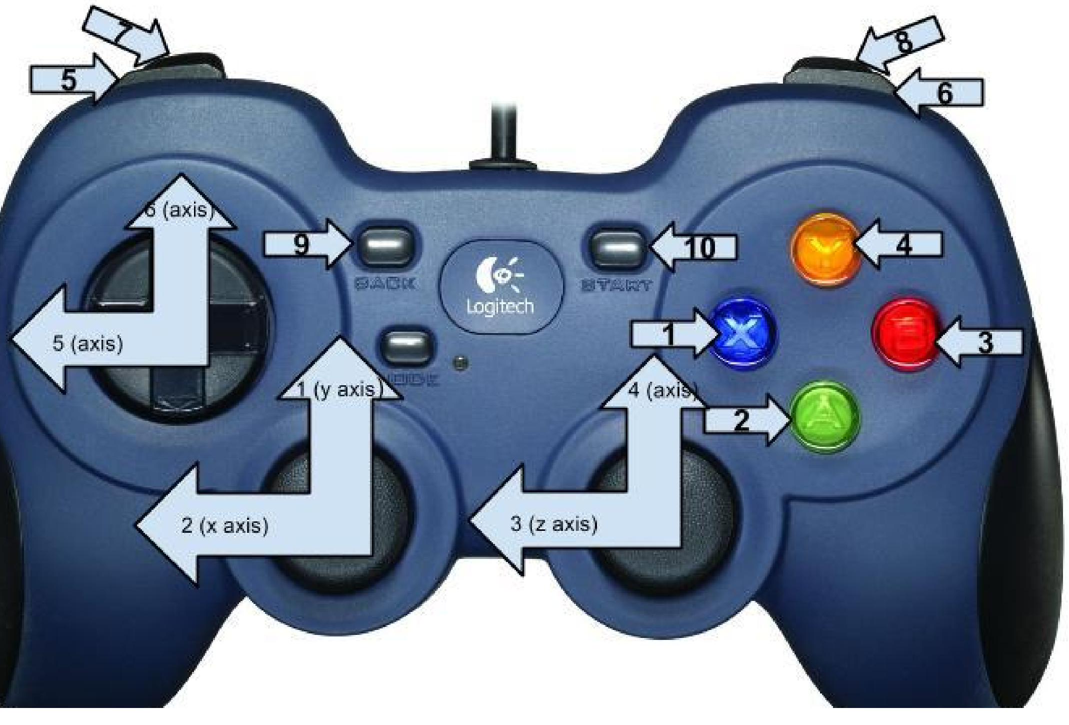
Logitech Gamepad F310

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

GetRawButton and GetRawAxis

The corresponding buttons in Wind River



Logitech Attack 3 USB Joystick

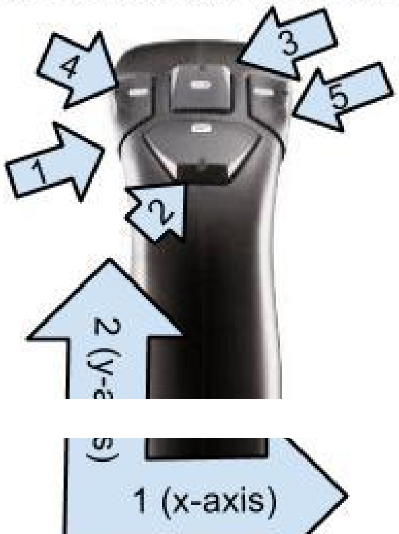
This Attack 3 Joystick looks awesome, doesn’t it exude the feeling of robotics? One hand to move the single joystick, the other to press the buttons on the bottom. The joystick class in the WPI Library does support this joystick and all if its many inputs.

FIRST Choice

Joystick Class (C++)

GetRawButton and GetRawAxis

The corresponding buttons in Eclipse



- 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

Jump to →

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

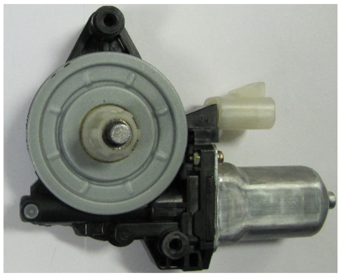
⅔ power of CIM, similar form factor and same mounting

- **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

anything else 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 **digital sidecar** with a **6v two pin jumper** in the two pin slot adjacent to where the servo is PWMed.

AndyMark

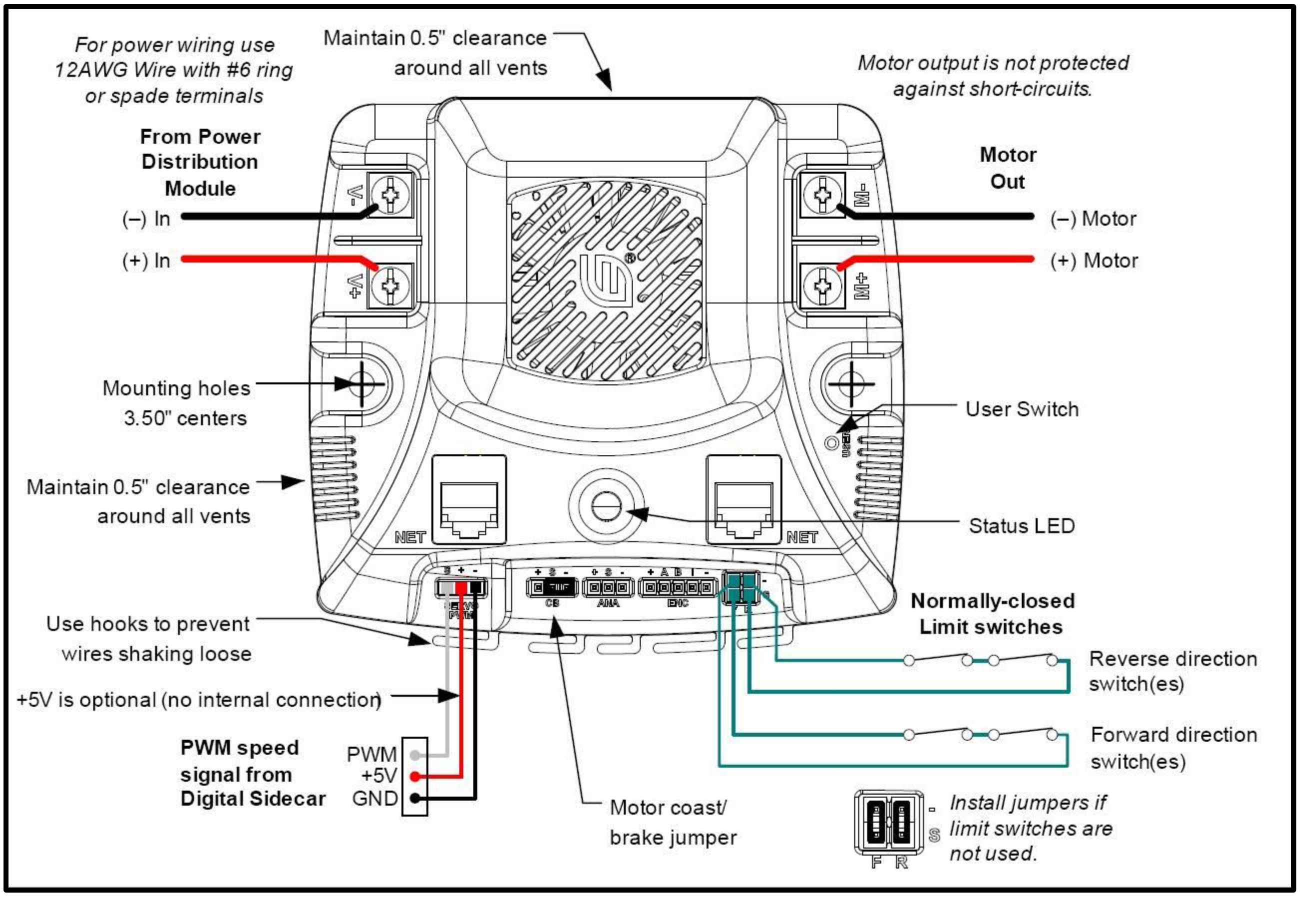
➠ Motor Controller Varieties

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.



For power wiring use 12AWG Wire with #6 ring or spade terminals
Maintain 0.5" clearance around all vents
Motor output is not protected against short-circuits.
From Power Distribution Module
(–) In
(+) In
V-
V+
M-
M+
Motor Out
(–) Motor
(+) Motor
Mounting holes 3.50" centers
User Switch
Maintain 0.5" clearance around all vents
NET
NET
Status LED
S + -
SERVO PWM
+ S -
CB
+ S -
ANA
+ A B I -
ENC
-
S
F R
Normally-closed Limit switches
Use hooks to prevent wires shaking loose
Reverse direction switch(es)
+5V is optional (no internal connection)
Forward direction switch(es)
PWM speed signal from Digital Sidecar
PWM
+5V
GND
Motor coast/ brake jumper
Install jumpers if limit switches are not used.

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**

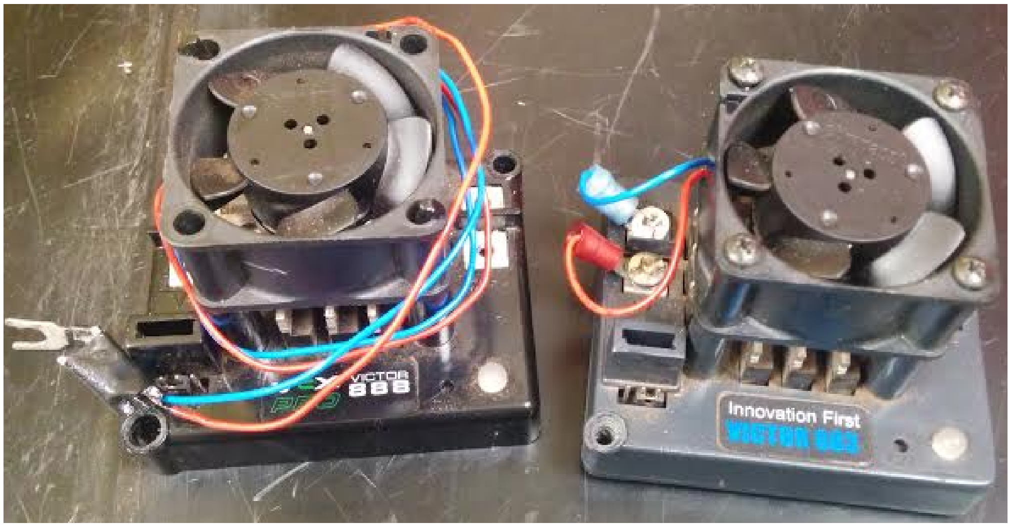
```
jaguar(1),
```

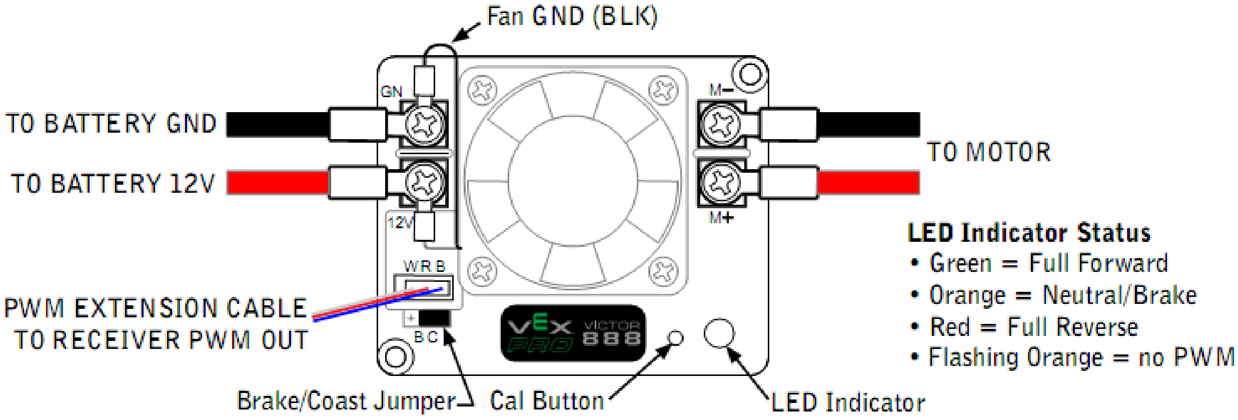
Initialize Jaguar motor controller as port # 1 in Digital Sidecar (PWM Out), initialized between **public : RobotDemo** and the braces({ }). If it is not the last object initialized, it needs a comma like a list. If it is the last object initialized, no punctuation; no comma, no semicolon, no period,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

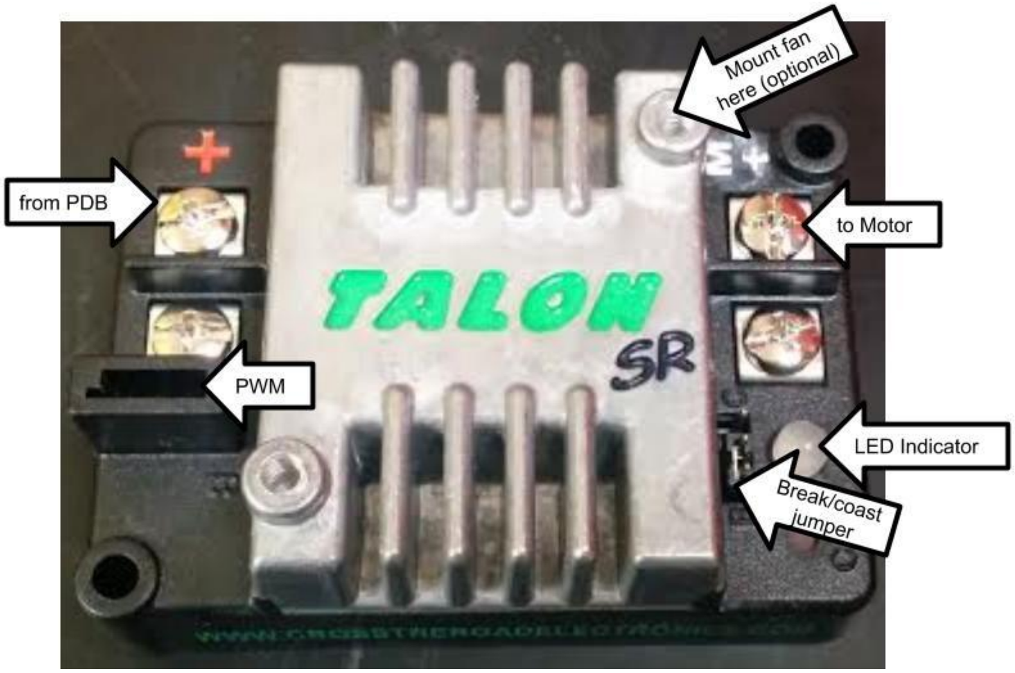
```
void OperatorControl()
{
if(stick.GetRawButton(1))
   {
   victor.Set(1.0);
   }
else if(stick.GetRawButton(2))
   {
   victor.Set(-1.0);
   }
else
   {
   victor.Set(0);
   }
}
```

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

Additional Resources

[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.

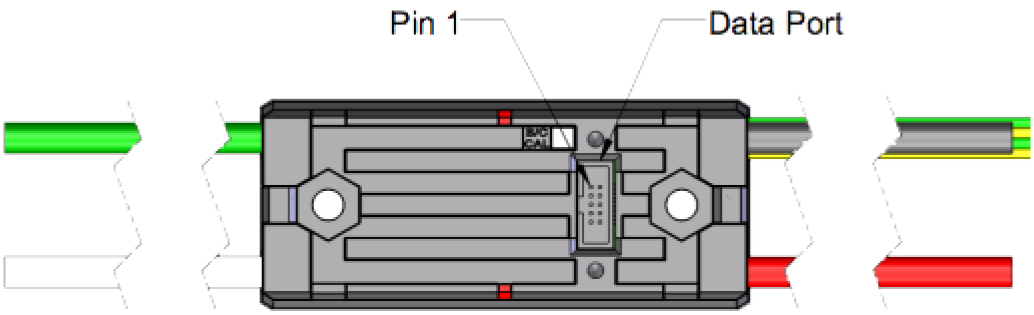
Talon SRX

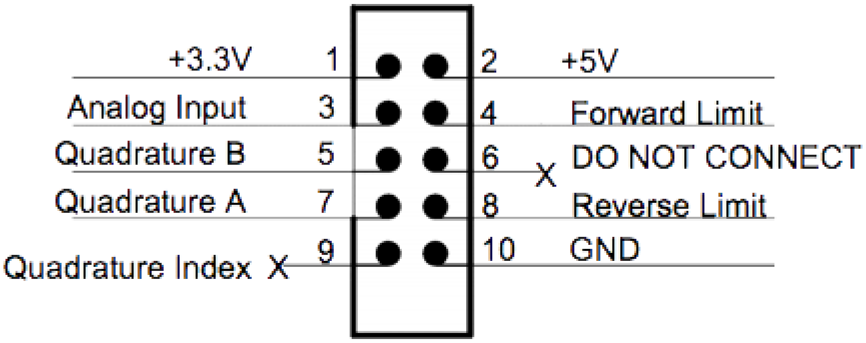
The Talon SRX is a new iteration of the Talon motor controller series that was introduced in the 2015 FRC season. The SRX is unique as it is CAN enabled and capable of operating with the roboRIO, PCM, and VRM, which all use CAN protocols. Because the Talon SRX was designed without a built-in ventilation 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)

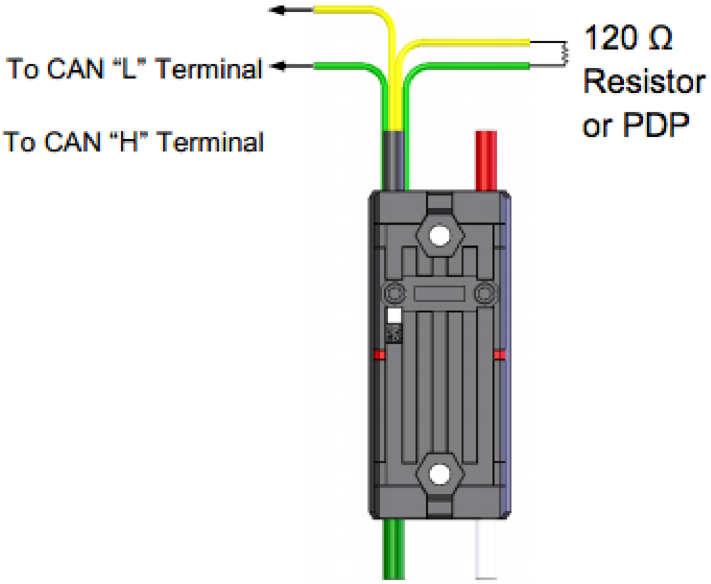
➠ Wiring





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

Additional Resources
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**

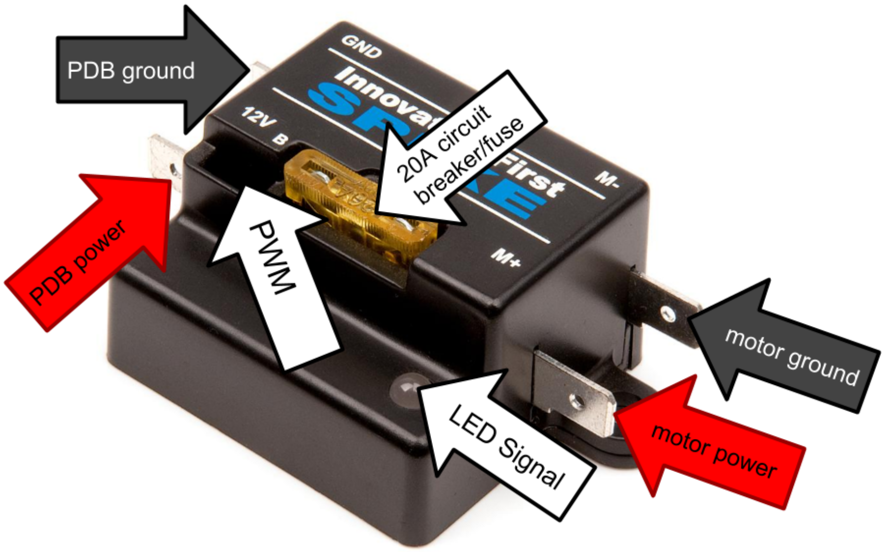
```
talonsrx(1),
```

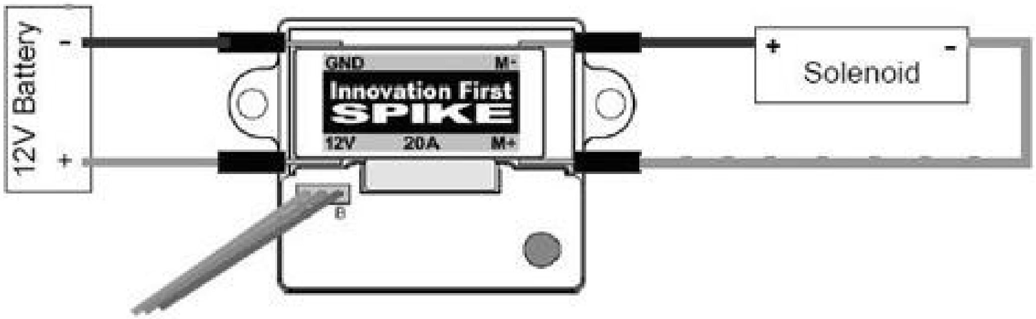
Initialize talon SRX motor controller as connected to port #1 in the Digital Sidecar (PWM Out); initialized between **public : RobotDemo** and the braces({ }). If it is not the last object initialized, it needs a comma like a list. If it is the last object initialized, no punctuation; no comma, no semicolon,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

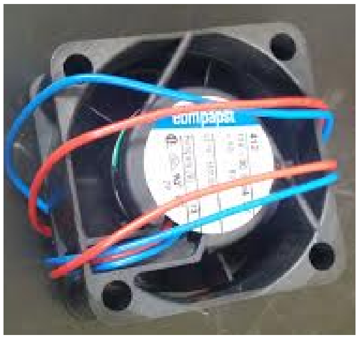
Instantiate the spike relay with the parameters [Digital Sidecar Port#], [direction of current [kForward, kBackward, or kBothDirections]] This is instantiated between “public : **RobotDemo**():” and the braces ({ }). If it is not the last object initialized, it needs a comma like a list. If it is the 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).



WPILib

Jump to →

Introduction

➠ What is the WPI Library?

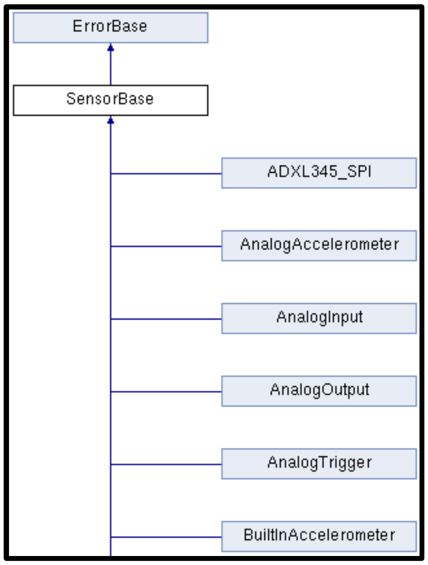
The WPI Library was developed by the Worcester Polytechnic Institute (WPI) in association with FIRST Robotics in order to assist teams and lower the barrier of entry for new teams. The WPI Library acts as a centralized resource for all information about software classes for C++ and Java used by the FIRST Robotics control system. There are classes for everything from hardware components like optical encoders and accelerometers to utility functions like field management and the driver station.

➠Advantages of using the WPI Library with C++

- Memory allocated and freed manually.
- Pointers, references, and local instances of objects.
- Header files and preprocessor used for including declarations in necessary parts of the program.
- Implements multiple inheritance where a class can be derived from several other classes, combining the behavior of all the base classes.
- Does not natively check for many common runtime errors.
- Highest performance on the platform, because it compiles directly to machine code for the PowerPC processor in the cRIO.

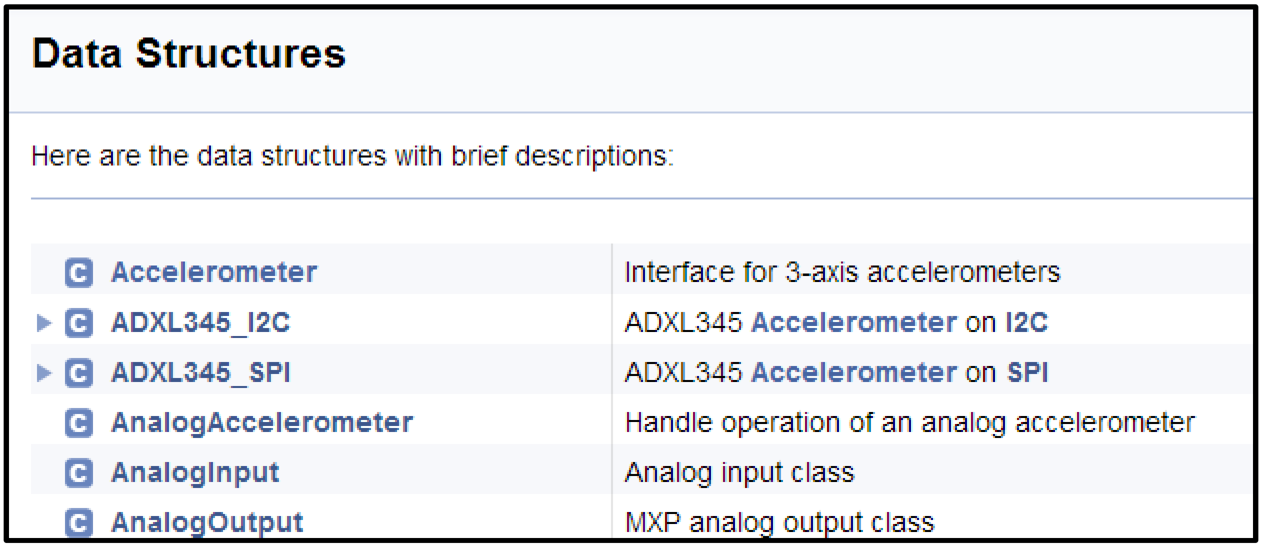
Navigating the WPI Library

Being a set of software classes to interact with your robot hardware, the WPI Library is organized into data structures with tree inheritance. As shown below, SensorBase is a class inherited from the ErrorBase structure, and most sensors such as an accelerometer or encoder will also inherit structures from the SensorBase class.



Another example would be Victors, Jaguars, and Servos are all members of inheritance within the Motor class. While the motor controllers may be physically very different, using the WPI Library they can be programmed under the Motor class so that these motor controllers can all utilize the Set function.

Each specific piece of hardware used in FRC has its own set of functions as well as parameters to allow you to manipulate it in the way. When navigating the tree of data structures, it is important to know which class the piece of hardware you are manipulating falls under. If you are unsure, you can click on each class for its description.

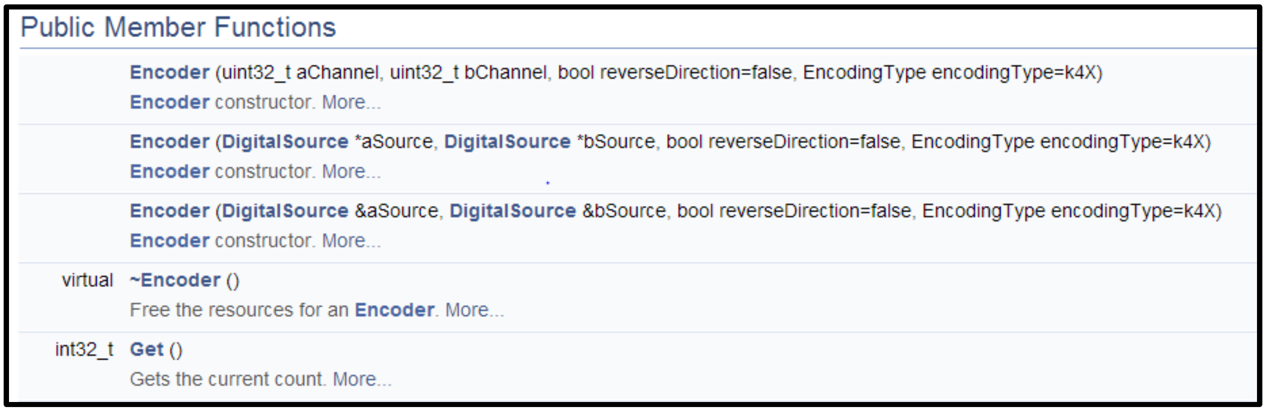


Data Structures

Here are the data structures with brief descriptions:

Accelerometer	Interface for 3-axis accelerometers
ADXL345_I2C	ADXL345 **Accelerometer** on **I2C**
ADXL345_SPI	ADXL345 **Accelerometer** on **SPI**
AnalogAccelerometer	Handle operation of an analog accelerometer
AnalogInput	Analog input class
AnalogOutput	MXP analog output class

➠How to use the WPI Library



Public Member Functions

	Encoder (uint32_t aChannel, uint32_t bChannel, bool reverseDirection=false, EncodingType encodingType=k4X) **Encoder** constructor. More...
	Encoder (**DigitalSource** *aSource, **DigitalSource** *bSource, bool reverseDirection=false, EncodingType encodingType=k4X) **Encoder** constructor. More...
	Encoder (**DigitalSource** &aSource, **DigitalSource** &bSource, bool reverseDirection=false, EncodingType encodingType=k4X) **Encoder** constructor. More...
virtual	**~Encoder** () Free the resources for an **Encoder**. More...
int32_t	**Get** () Gets the current count. More...

The WPI Library is mainly used as a reference when programming hardware components. It contains information on how to construct, instantiate, initialize, and use the component. After finding the component within the WPI Library, you will see a list of functions that either make the hardware piece return a value or perform an action. These are also called methods. The method name is the blue bolded text outside of the parenthesis while its parameters are within them. The black text to the left of the method name, if there is any, is the return type of the function. After finding the function you'd like to use, you can click on it to find a more detailed description.

Drive Code

Jump to →

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 application that will automatically call your Autonomous and
        OperatorControl methods at the right time as controlled by the switches on the driver
        station or the field controls.
         */
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 application that will automatically call your Autonomous
        and OperatorControl methods at the right time as controlled by the switches on the
        driver station or the field controls.
         */
```

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 methods.

```
        {
                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. Arcade Drive 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
                }
        }
```

```
	/**
	 * Runs during test mode
	 */
	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 (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 **SetExpirations** 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;

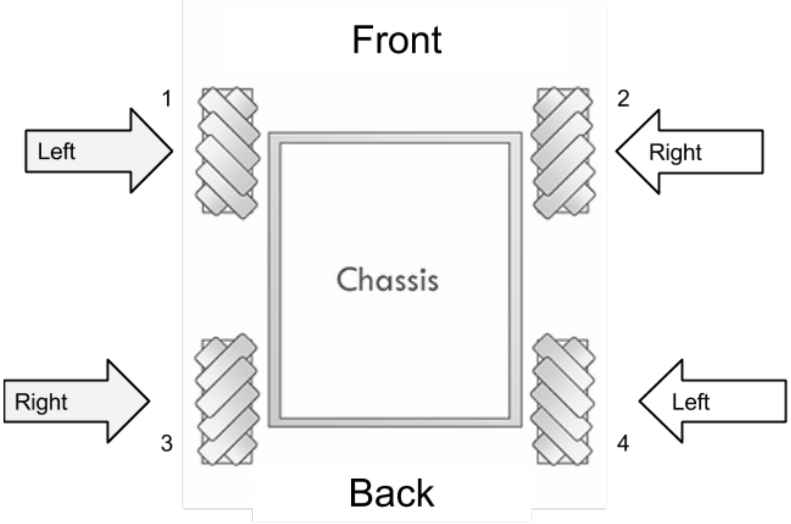
if the joystick is greater than a threshold **OR** if the joystick is below negative threshold. It would look like **joystick.GetRawAxis(2) > 0.2 || joystick.GetRawAxis(2) < -0.2**. Compared to what is in there, it is much easier to code, but less understandable. Reasoning is when you tilt the joystick 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);
```

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.

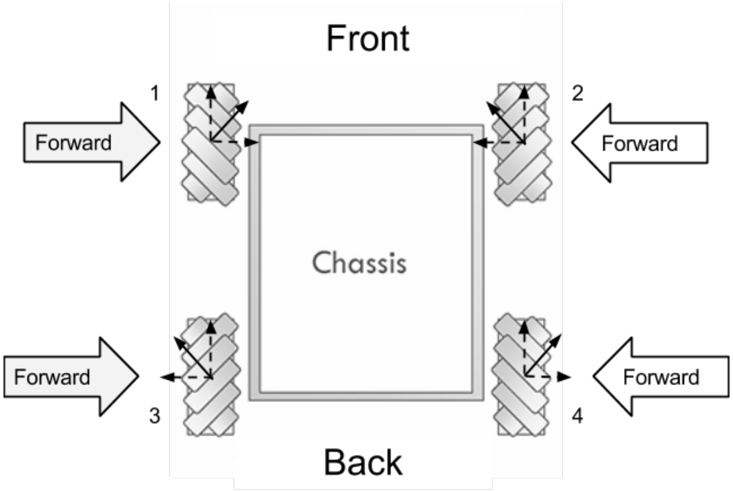
Right Configuration

- 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

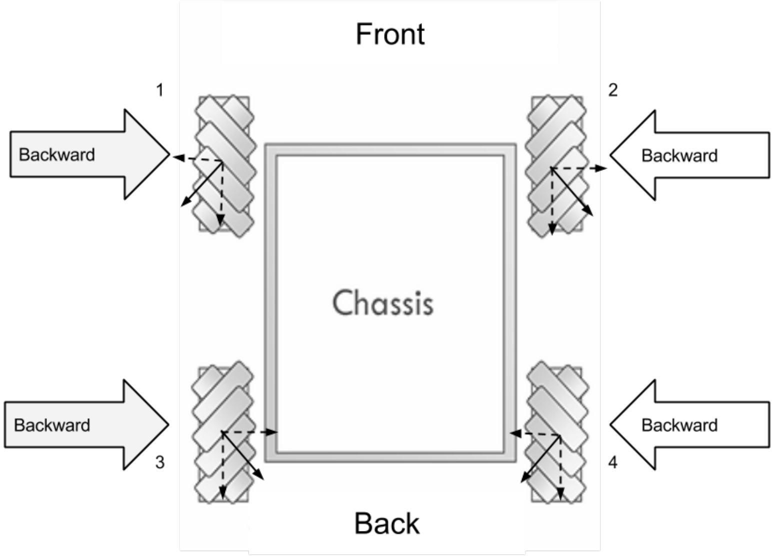
Driving Forward

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



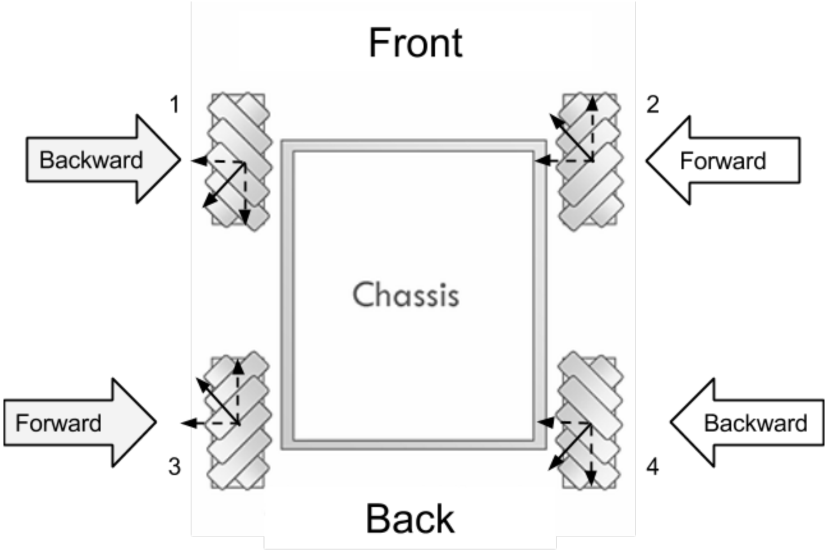
Driving Backward

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

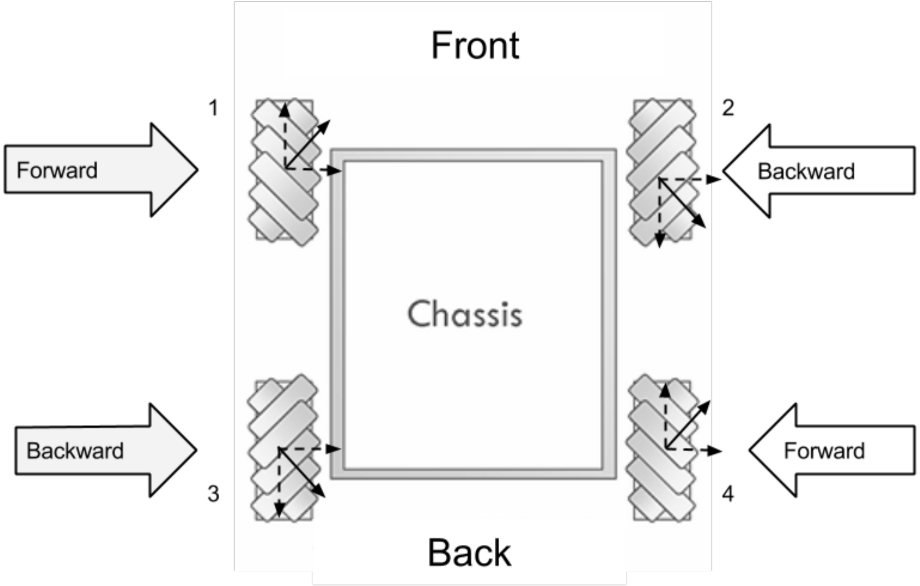


Strafing Left

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

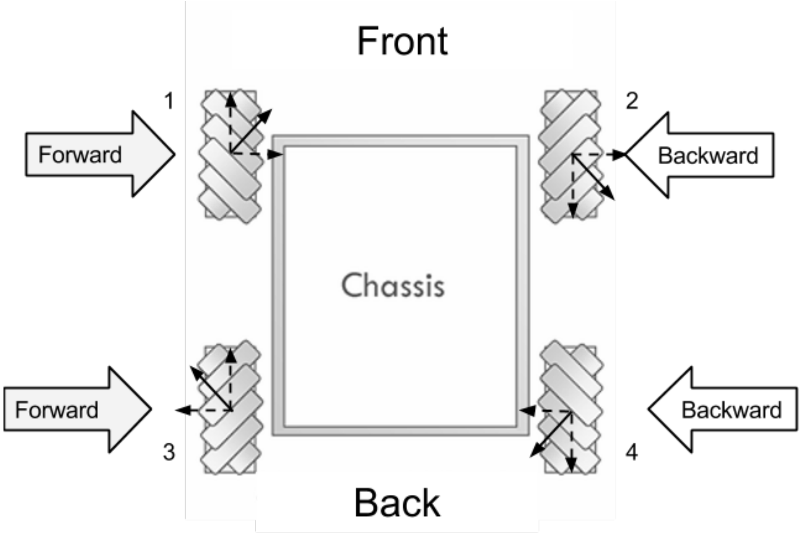


Strafing Right
Wheels 1 and 4 are rotating forward, Wheels 2 and 3 are rotating backward to allow the drive frame to strafe toward the right.



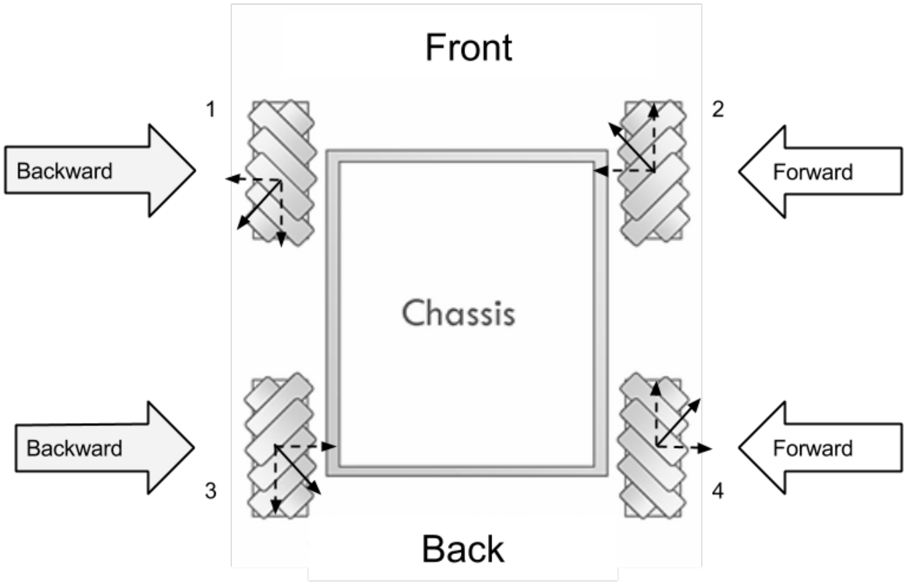
Turning Clockwise

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 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 Logitech Gamepad Controller here 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
   {
   }
```

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 Logitech Gamepad Controller utilizes the driver station ports

```
   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;
```

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)

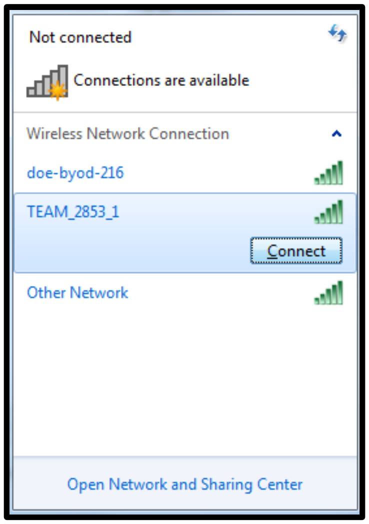
```
leftFront.Set(x-y-z);
leftBack.Set(-x-y-z);
rightFront.Set(x+y+z);
rightBack.Set(-x+y+z);
```

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.

➠ Deploying Code

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.



Not connected

Connections are available

Wireless Network Connection

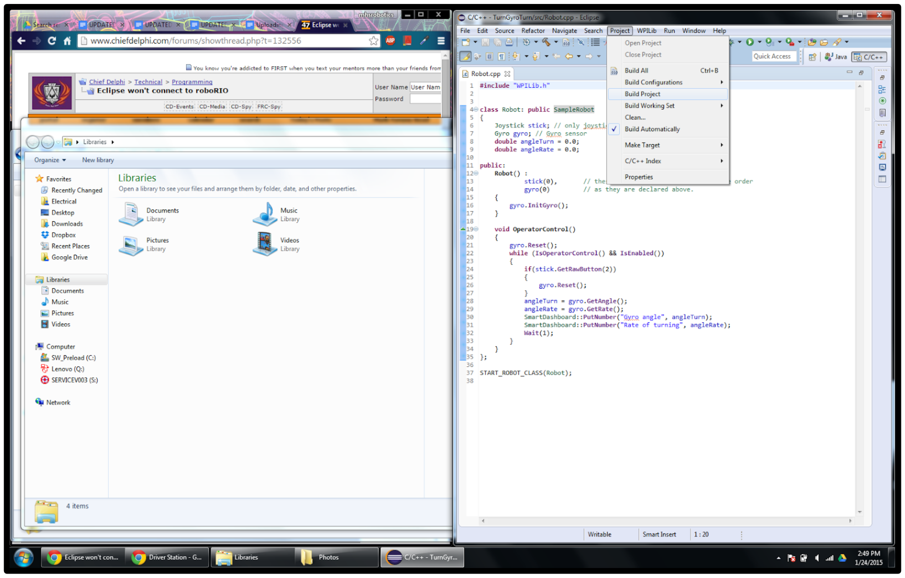
doe-byod-216

TEAM_2853_1

Other Network

Connecting to a network

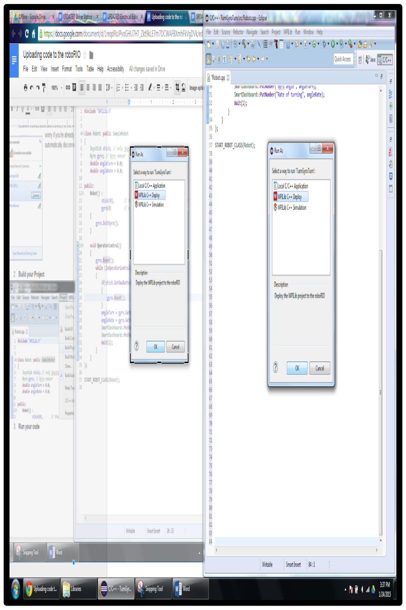
1. **Build your Project**. Click the "Project" option and then click "Build Project." To upload code to the roboRIO, it is only necessary to build the project for uploading code that hasn't been build in the past or was modified, but it's a good habit to always build your projects before uploading them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



```
#include "WPILib.h"


class Robot: public SampleRobot
{
    Joystick stick; // only joysti
    Gyro gyro; // Gyro sensor
    double angleTurn = 0.0;
    double angleRate = 0.0;

public:
    Robot() :
            stick(0),       // the                                          order
            gyro(0)         // as they are declared above.
    {
        gyro.InitGyro();
    }

    void OperatorControl()
    {
        gyro.Reset();
        while (IsOperatorControl() && IsEnabled())
        {
            if(stick.GetRawButton(2))
            {
                gyro.Reset();
            }
            angleTurn = gyro.GetAngle();
            angleRate = gyro.GetRate();
            SmartDashboard::PutNumber("Gyro angle", angleTurn);
            SmartDashboard::PutNumber("Rate of turning", angleRate);
            Wait(1);
        }
    }
};

START_ROBOT_CLASS(Robot);

```

Building an Eclipse project

1. **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"

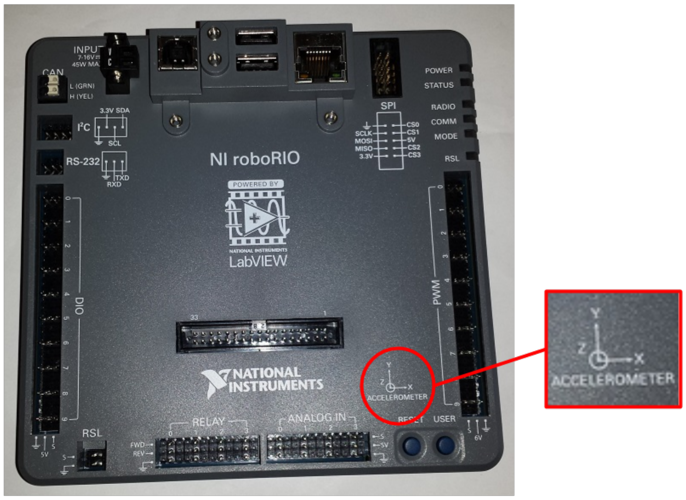
Sensors

The roboRIO accelerometer

Jump to →

The roboRIO Accelerometer

➠ General Overview



One of the many features that comes with the RoboRIO is the built-in 3-axis accelerometer, which has the potential to replace the ADXL345 accelerometer that also comes in the 2015 Kit of Parts. The purpose of this device is to determine the proper acceleration of an object, which is its acceleration relative to freefall. This can be used to determine how much the robot is tilted or a way to monitor motion.

➠ Specifications

- Axes: 3 (x, y, and z)
- Sample Rate: 800 Samples per second
- Resolution: 12 bits
- Range: ±8g (gravity)
- Noise: 3.9 mgms typical at 25° C

➠ The Code

This is the code we used to determine the direction of each axis as well as the stability of the returned values.

```
#include "WPILib.h"

class Robot: public SampleRobot
{
	BuiltInAccelerometer accelerometer;

	const double kUpdatePeriod = 0.005; // 5milliseconds / 0.005 seconds.
public:
	//sets the range of the accelerometer to be + or - 8G (units of gravity)
```

```
    Robot() : accelerometer(Accelerometer::Range::kRange_8G)
    {
    }
    void OperatorControl()
    {
        double xAcceleration;       //acceleration on the x-axis
        double yAcceleration;       //acceleration on the y-axis
        double zAcceleration;       //acceleration on the z-axis
        double previousX = 0;       //Previous recursive average on x-axis
        double previousY = 0;       //Previous recursive average on y-axis
        double previousZ = 1;       //Previous recursive average on z-axis

        while (IsOperatorControl() && IsEnabled())
        {
            xAcceleration = accelerometer.GetX(); //returns x-axis accel
            yAcceleration = accelerometer.GetY(); //returns y-axis accel
            zAcceleration = accelerometer.GetZ(); //returns z-axis accel

            SmartDashboard::PutNumber("X-Axis G:", xAcceleration);
            SmartDashboard::PutNumber("Y-Axis G:", yAcceleration);
            SmartDashboard::PutNumber("Z-Axis G:", zAcceleration);

            SmartDashboard::PutNumber("Recrusive X-Axis Average:",
((xAcceleration*0.1) + (0.9*previousX)));
            //returns a recursive average for the x-axis
            SmartDashboard::PutNumber(“Recursive Y-Axis Average:",
((yAcceleration*0.1) + (0.9*previousY)));
            //returns a recursive average for the y-axis
            SmartDashboard::PutNumber(“Recursive Z-Axis Average:”,
((zAcceleration*0.1) + (0.9*previousZ)));
            //returns a recursive average for the z-axis
            previousX = (xAcceleration*0.1) + (0.9*previousX);
            previousY = (yAcceleration*0.1) + (0.9*previousY);
            previousZ = (zAcceleration*0.1) + (0.9*previousZ);
            Wait(kUpdatePeriod);
```

```
                        // Wait a short bit before updating again
                }
        }
};

START_ROBOT_CLASS(Robot);
```

➠ The Explanation

```
BuiltInAccelerometer accelerometer;
```

Declare the RoboRIO accelerometer as BuiltInAccelerometer; declared between **public SampleRobot** and **public : RobotDemo**

```
public:
        //sets the range of the accelerometer to be + or - 8G (units of gravity)
        Robot() : accelerometer(Accelerometer::Range::kRange_8G)
        {
        }
```

Initializes the roboRIO accelerometer with a range of +/- 8Gs, which is the acceleration in units of gravity (9.81m/s). This is the maximum range that the device is capable of. There is no need to input pwm ports as the accelerometer is built in.

```
void OperatorControl()
        {
                double xAcceleration;       //acceleration on the x-axis
                double yAcceleration;       //acceleration on the y-axis
                double zAcceleration;       //acceleration on the z-axis
                double previousX = 0;       //Previous recursive average on x-axis
                double previousY = 0;       //Previous recursive average on y-axis
                double previousZ = 1;       //Previous recursive average on z-axis
```

In this section we initialize the variables that we will be using for output on the smartdashboard. Since the accelerometer has 3 axes, we need a variable for each one (x, y, and z). The previous axis variables are used to determine a recursive average that is explained later in this document. They are meant to store the value of the recursive average of all previously returned values. previousX and previousY are equal to 0 because that is the expected value when the

RoboRIO is at rest and on a perfectly horizontal surface. previousZ is equal to 1 because the RoboRIO is not in freefall, which is what an accelerometer measures acceleration in reference to.

```
while (IsOperatorControl() && IsEnabled())
                {
                xAcceleration = accelerometer.GetX(); //returns x-axis accel
                yAcceleration = accelerometer.GetY(); //returns y-axis accel
                zAcceleration = accelerometer.GetZ(); //returns z-axis accel

                        SmartDashboard::PutNumber("X-Axis G:", xAcceleration);
                        SmartDashboard::PutNumber("Y-Axis G:", yAcceleration);
                        SmartDashboard::PutNumber("Z-Axis G:", zAcceleration);
```

Here, the variables xAcceleration, yAcceleration, and zAcceleration are being set to the current values of the accelerometer pertaining to the x, y, and z axis. After updating these values, they are then sent to the SmartDashboard to be read in the form of a running value or a table.

```
              SmartDashboard::PutNumber("Recrusive X-Axis Average:",
((xAcceleration*0.1) + (0.9*previousX)));
                        //returns a recursive average for the x-axis
                        SmartDashboard::PutNumber(“Recursive Y-Axis Average:",
((yAcceleration*0.1) + (0.9*previousY)));
                        //returns a recursive average for the y-axis
                        SmartDashboard::PutNumber(“Recursive Z-Axis Average:”,
((zAcceleration*0.1) + (0.9*previousZ)));
                        //returns a recursive average for the z-axis
                        previousX = (xAcceleration*0.1) + (0.9*previousX);
                        previousY = (yAcceleration*0.1) + (0.9*previousY);
                        previousZ = (zAcceleration*0.1) + (0.9*previousZ);
                        Wait(kUpdatePeriod);
                        // Wait a short bit before updating again
        }
       }
};
```

```
START_ROBOT_CLASS(Robot);
```

Here we calculate the recursive average for each axis which can be useful during testing. The purpose of having this recursive average is to effectively reduce the sensitivity of the returned values to fluctuations. This allows you to get a more accurate and stable reading for each axis of the accelerometer, and could be used to test for varying angles that the accelerometer is tilted.

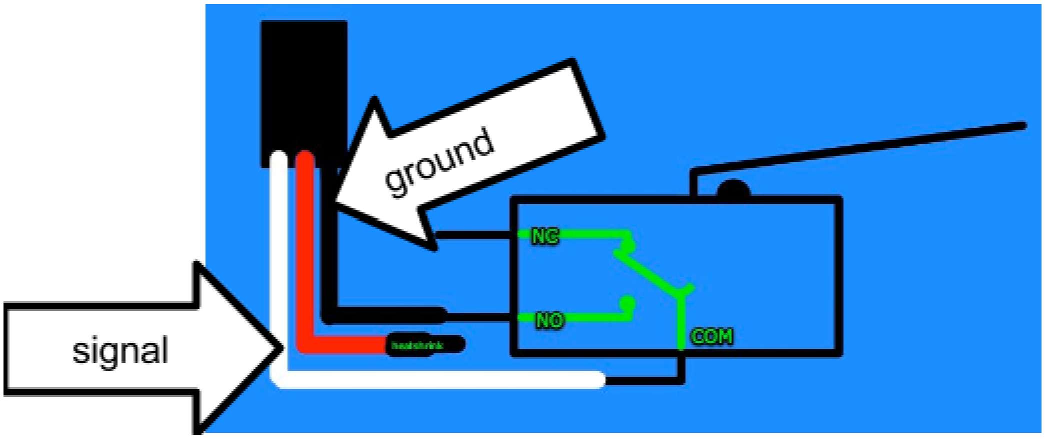
This simple recursive average algorithm is done by taking .1 of the current value for an axis and adding it to .9 of the previous average. This essentially means that we have a constantly changing average. Note that the sensitivity of this algorithm can be adjusted by changing the “0.1” and “0.9” values. However, remember that the two multipliers must add to a value of 1, because the average will tend to continue increasing or decreasing depending on how the multipliers are adjusted. If you have a multipliers of .01 and .99, the current average will be much less sensitive to change as the weight of each new number is drastically decreased.

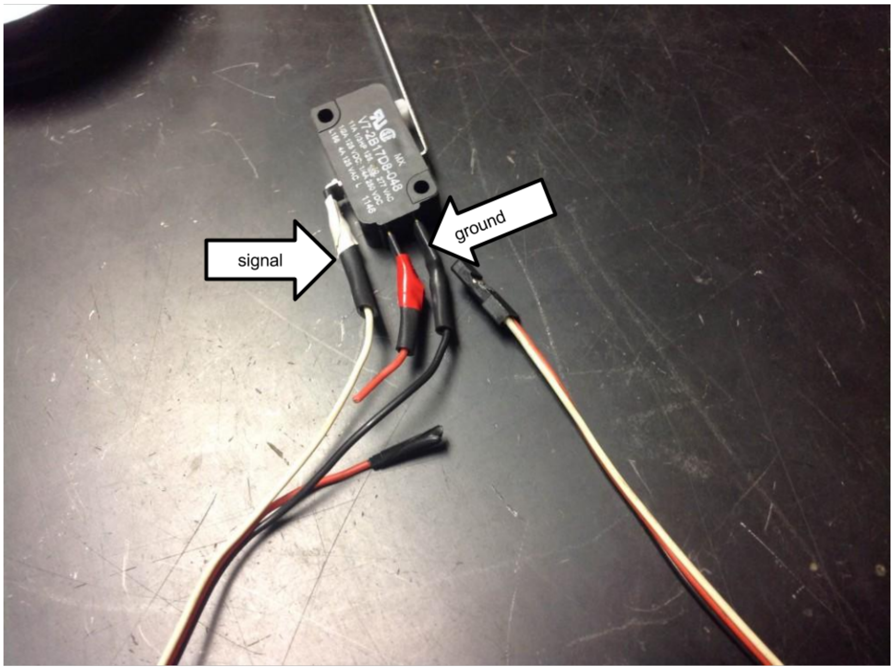
Microswitch

➠ Use

The microswitch is usually used to keep something from overextending or surpassing some distance. When the switch not pushed, the switch is NC (normally closed), returning 1. When the switch pushed, the switch is NO (Normally Open), returning 0.

➠ Wiring





The switch is plugged into the Digital IO section of the digital sidecar via PWM.

➠ Programming

Declared and instantiated as a **DigitalInput**

- When switch NC, returns 0
- When switch NO, returns 1

Declaration: DigitalInput switch;

Instantiation: switch (1) //port number in the digital sidecar

Using Microswitch: switch.Get(); //returns either 0 or 1

// A Barebones Code

```
#include "WPILib.h"    // WPILibrary.h
//Last modified: February 7, 2014 by: Vivian
class RobotDemo : public SampleRobot
{
        DigitalInput limitSwitch;
public:
        RobotDemo(void):
                limitSwitch(1)
        {
        }

        void Autonomous(void)
        {
        }
        void OperatorControl(void)
        {
                while (IsOperatorControl())
                {
                        if(limitSwitch.Get() == 1)
                        {
                                //you can do things here is the switch is pressed
                        }
                }
        }
        void Test() {
                while(IsTest())
                {
                }
        }
};
START_ROBOT_CLASS(RobotDemo);
```

Optical Encoder

➠ Use

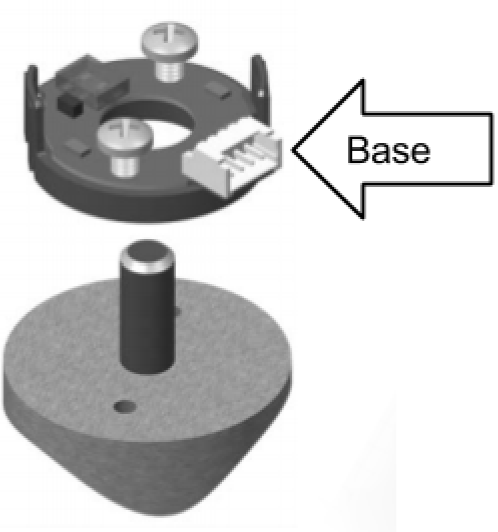
The optical encoder is the most common type of encoder in FRC that uses one or more LED's pointed at a strip or slit code wheel and two detectors 90 degrees apart to measure the rotation speed of a wheel or other shafts. The encoder pictured to the left is a **US Digital E4P** (am-0174) optical encoder.

Specs

- Max RPM: 10,000 RPM
- 100-360 cycles per revolution
- 400-1440 pulses per revolution
- Minimum shaft length: .375" (⅜)
- Shaft Diameter: .079" to .250"
- Weight: .018 lbs

➠ Assembly

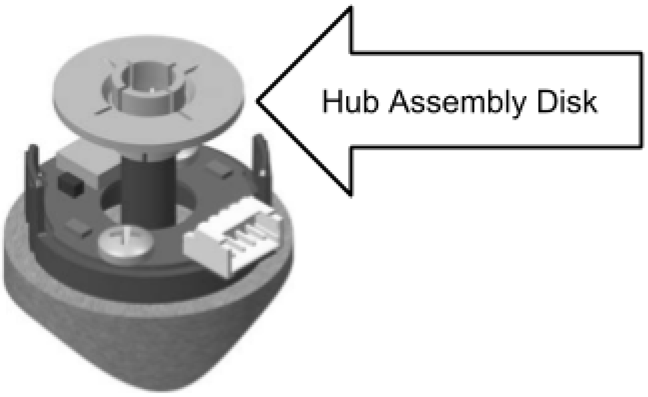
1. Place base over shaft. Secure base to mounting surface using either the two screws on the base or a mounting pad.



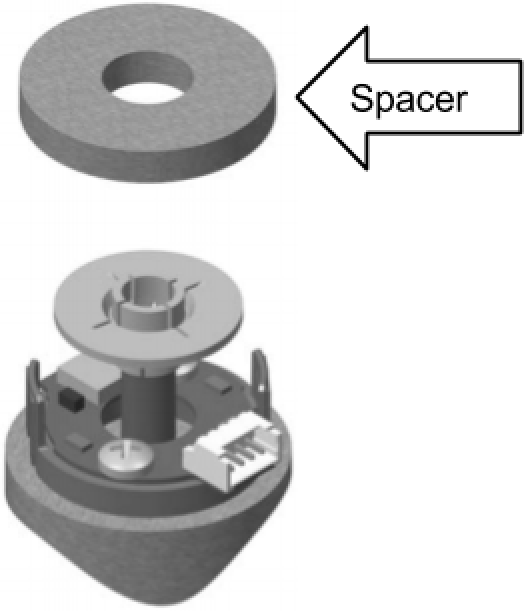
Mounting the base without a mount pad

Mounting pad placed under the base if used.

2. Place hub disk assembly onto shaft with pattern-side down towards base. It should not be completely pressed down.



3. Use the spacer (lip facing downwards) to push the hub disk assembly to the appropriate location on the shaft. The disk should not be touching anything besides the shaft and the spacer, and there should be a considerable gap between the disk and the base.



4. Remove the spacer while ensuring that the disk stays in place.

5. Place the housing on top of the encoder. Using your thumb and finger, squeeze ears together to ensure that the cover fully latches.

➠ Mounting

It is easiest if the optical encoder is placed on the output shaft as there is a direct correlation between the rotation of the shaft and the movement of whatever it controls. Also ensure that the optical encoder is perfectly centered around the shaft.

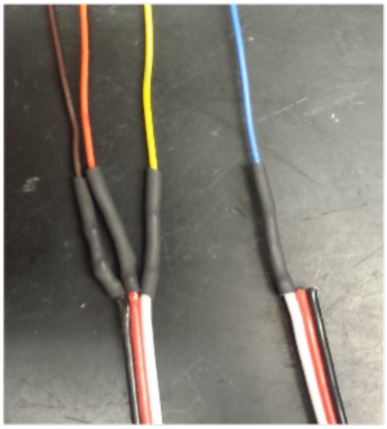
➠ Storage

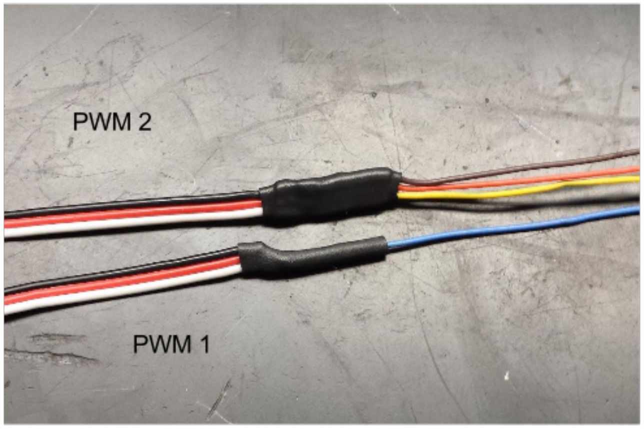
The optical encoder can get scratched easily so it should be stored in a special case to prevent scratching or in a place that will not scratch the surface of the disk.

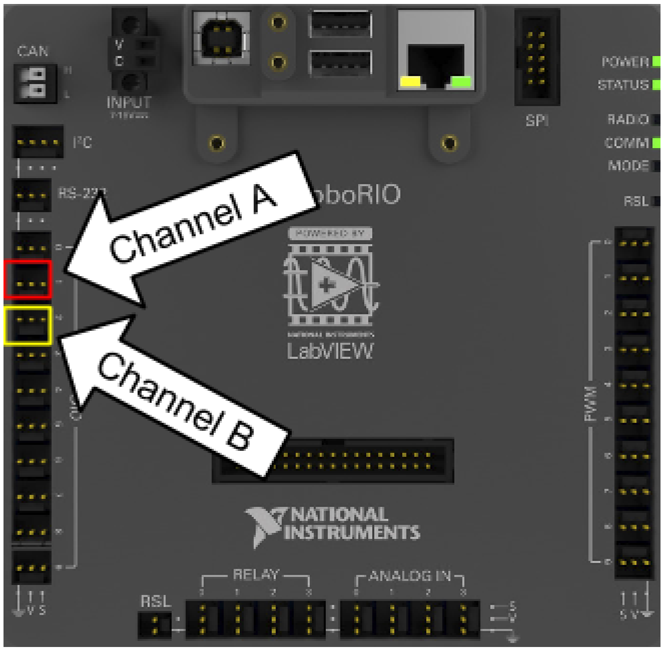
➠ Wiring

The wiring for the optical encoder uses two channels (Digital IO; A/B) for the PDP so four wires have to be soldered to two PWMs.

Orange: Power-------------------------------> PWM 2 Power
Blue: Channel A------------------------------> PWM 1 Signal
Brown: Ground-------------------------------> PWM 2 Ground
Yellow:Channel B----------------------------> PWM 2 Signal



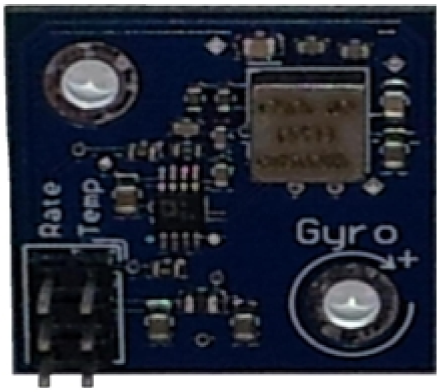




➠Programming

```
//Code Used for Testing with roboRIO
#include "WPILib.h"

/**
 * Encoder Test Using Motor
 */
class Robot : public SampleRobot {
        Encoder encoder;
        Joystick logitech;
        Talon talon;

        // update every 0.005 seconds/5 milliseconds.
        double kUpdatePeriod = 0.005;

public:
        Robot() :
                        encoder(1, 2, false, Encoder::k4X),
                        logitech(0), // Initialize logitech on port 0.
                        talon(0) // Initialize the Talon on channel 0.
        {
                encoder.SetSamplesToAverage(5); // Used to reduce noise in period
                encoder.SetDistancePerPulse(1.0/360);
                // This makes it so that GetDistance will return 1 when the shaft
        // makes a full rotation and that GetRate will be in Revs per second
    }
    void OperatorControl()
    {
        encoder.Reset();
        while (IsOperatorControl() && IsEnabled())
        {
            talon.Set(logitech.GetY());
            //gets the y-axis on the LEFT logitech
            while(encoder.GetDistance() < 2)
            {
                talon.Set(-0.2);
        SmartDashboard::PutNumber("Encoder Distance", encoder.GetDistance());
                // prints displacement in revolutions
        SmartDashboard::PutNumber("Encoder Rate", encoder.GetRate());
                // prints rate in Revs per second
                Wait(kUpdatePeriod);
            }
        }
    }
};

START_ROBOT_CLASS(Robot);
```

The Gyro



➠ General Overview

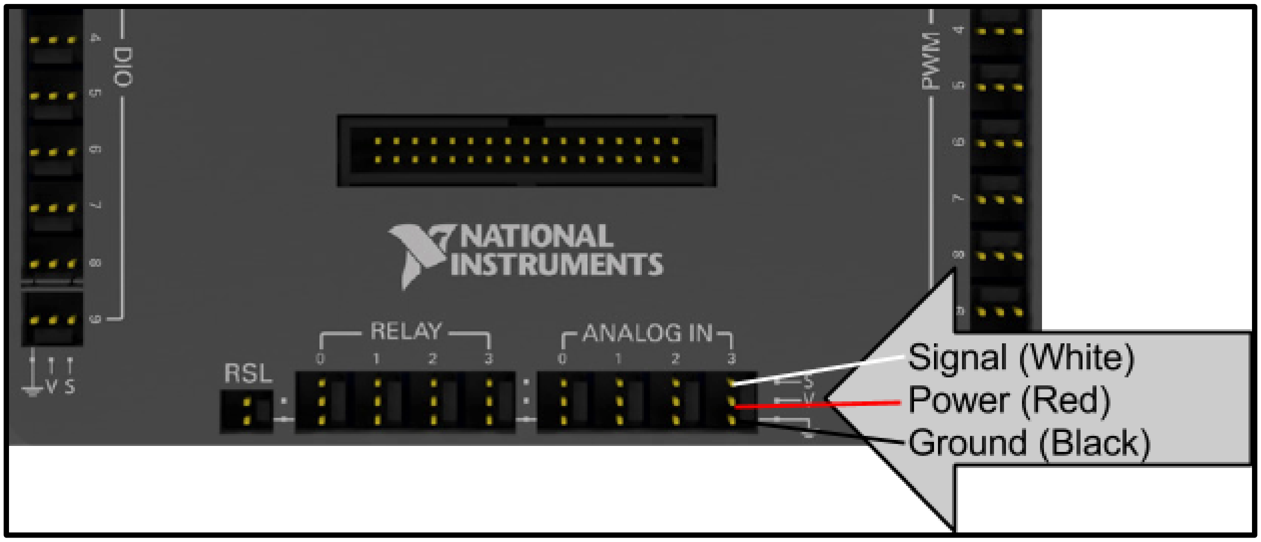
The gyro measures angular changes on the top surface axis. The voltage output depends on the angular change it detects. It can also measure the rate of angular change. It normally would be used in tandem with the accelerometer, since the accelerometer can detect absolute angle vs angular motion. The gyro best functions at the center of the robot’s axis of rotation. When mounting it, keep the gyro away from anything that might fry the gyro, so it is best to electronically isolate it from the main mounting board. Noteworthy is that it also contains a temperature sensor, useful for detecting heat within the robot system during operation if a thermal detector is not available or during a match.

➠ Specifications

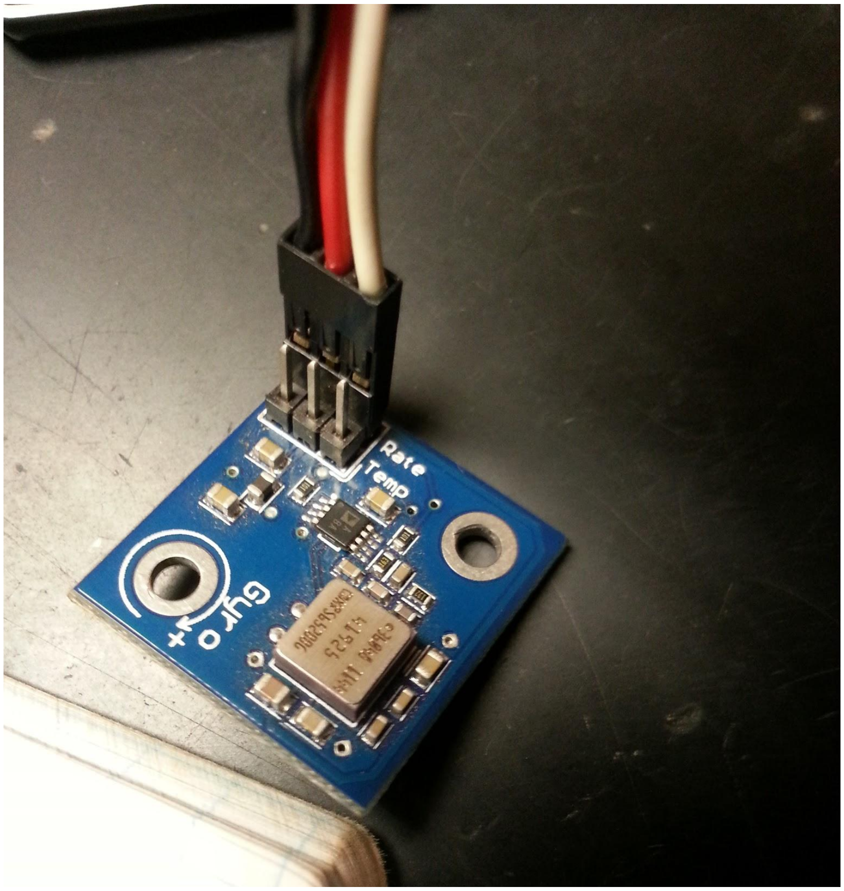
Gyro accepts a +5V for power, can record up to 250^{O}/s, has a nominal output of 2.5V at standstill, adds 7mV/^{O}/s. Board carries a filter set to 400 Hz. Contains integrated temperature sensor which accepts a +5V for power with nominal output at 2.5V at 25^{O}C, adds 9mV/^{O}C.

➠ Wiring

Wires to the roboRIO Analog IN ports using a female-to-female PWM. Ground, Power, Signal from outside in respectively.



Wires to the Gyro sensor with the other end of the female-to-female PWM. Ground, Power, Signal from outside in respectively



➠ Programming

Utilizing Eclipse for roboRIO

```
#include "WPILib.h"

class Robot: public SampleRobot
{
    Joystick stick; // only joystick
    Gyro gyro; // Gyro sensor
    double angleTurn = 0.0;
    double angleRate = 0.0;
    double driftRate = 0.0;

public:
    Robot() :
            stick(0),      // these must be initialized in the same order
            gyro(0)// as they are declared above.
    {
        gyro.InitGyro();
    }

    void OperatorControl()
    {
        gyro.Reset();
        while (IsOperatorControl() && IsEnabled())
        {
            if(stick.GetRawButton(2))
            {
                gyro.Reset();
                Wait(2);
                driftRate = gyro.GetAngle();
                SmartDashboard::PutNumber("drift rate", driftRate);
            }
            angleTurn = gyro.GetAngle();
            angleRate = gyro.GetRate();
            SmartDashboard::PutNumber("Gyro angle", angleTurn);
            SmartDashboard::PutNumber("Rate of turning", angleRate);
            Wait(0.05);
        }
    }
};

START_ROBOT_CLASS(Robot);
```

Camera

Jump to

Hardware

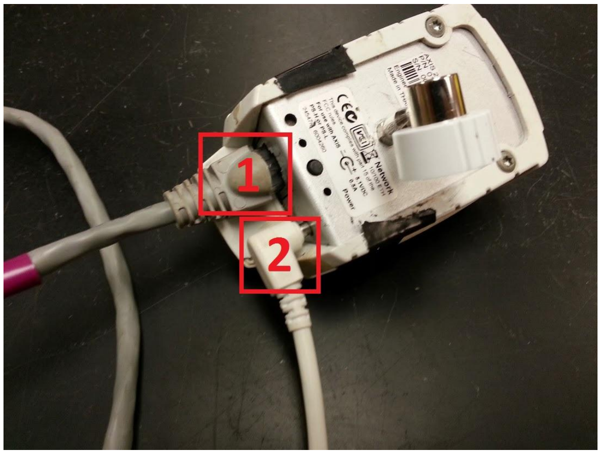
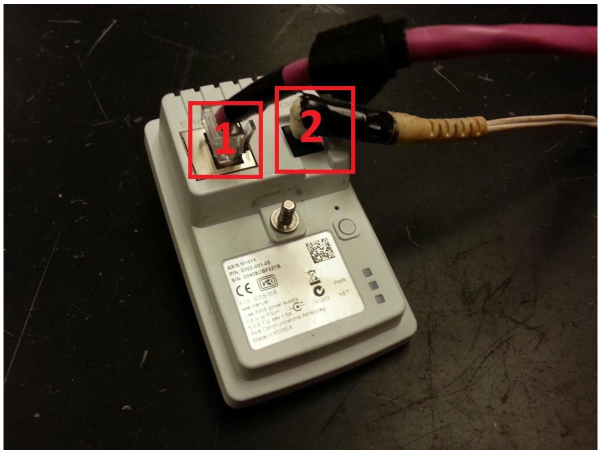
➠ Supported Cameras

	Axis 206 ● Oldest Model (Discontinued) ● Field of View: 55º ● Max Resolution: 640x480 ● Dimensions(mm): 85x55x34 ● Weight: 177 g (including stand)

	Axis M1011 ● Discontinued ● Field of View: 47º ● Max Resolution: 640x480 ● Dimensions(mm): 95x59x34 ● Weight: 94 g
	Axis M1013 ● Current model ● Field of View: 67º ● Max Resolution: 800x600 ● Dimensions(mm): 95x59x41 ● Weight: 110 g **AndyMark** **Axis Product Page**

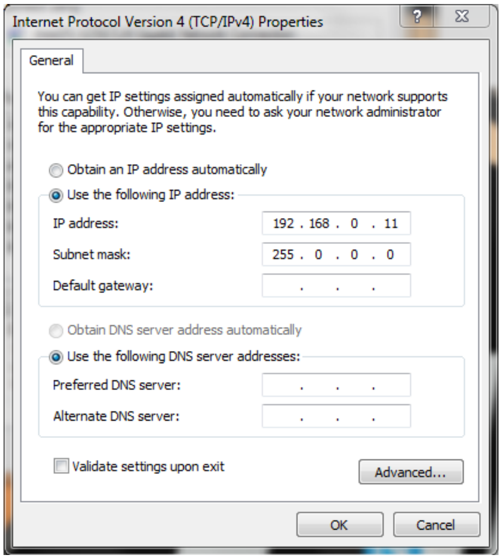
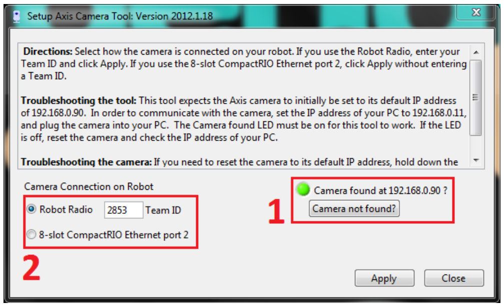
Setting Up the Camera

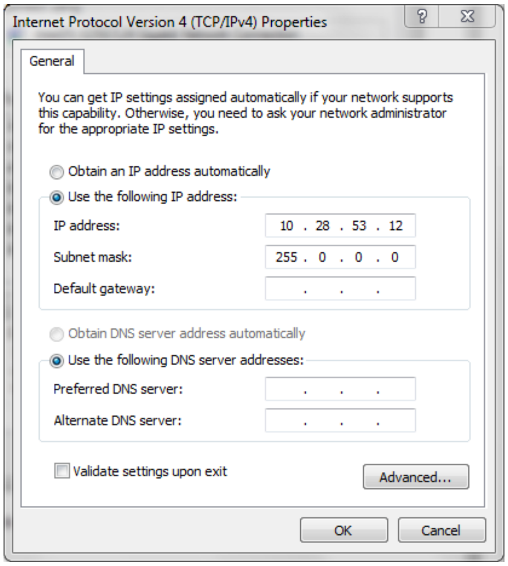
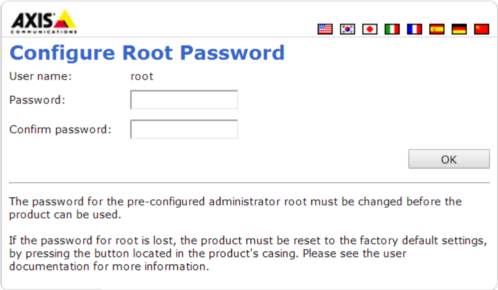
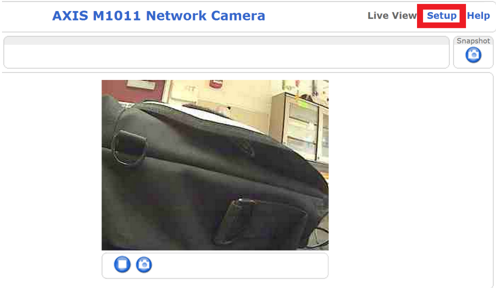
➠ Wiring

	
Axis 206 1 - Ethernet 2 - DC Power	Axis M1011 1 - Ethernet 2 - DC Power

➠ Network Connections

Setting up the Camera for the first time

1) Open the Setup Axis Camera	 Setup Axis Camera
2) Change the computer's IP to 192.168.0.11	 Internet Protocol Version 4 (TCP/IPv4) Properties General You can get IP settings assigned automatically if your network supports this capability. Otherwise, you need to ask your network administrator for the appropriate IP settings. ○ Obtain an IP address automatically ● Use the following IP address: IP address: 192 . 168 . 0 . 11 Subnet mask: 255 . 0 . 0 . 0 Default gateway: . . . ○ Obtain DNS server address automatically ● Use the following DNS server addresses: Preferred DNS server: . . . Alternate DNS server: . . . □ Validate settings upon exit Advanced... OK Cancel
3) Return to the Setup Axis Camera window. 1 - If camera isn't connected, the circle will be black 2 - Once it is connected to the camera, input your team number in the team ID with the robot radio selected 3 - Click Apply	 Setup Axis Camera Tool: Version 2012.1.18 **Directions:** Select how the camera is connected on your robot. If you use the Robot Radio, enter your Team ID and click Apply. If you use the 8-slot CompactRIO Ethernet port 2, click Apply without entering a Team ID. **Troubleshooting the tool:** This tool expects the Axis camera to initially be set to its default IP address of 192.168.0.90. In order to communicate with the camera, set the IP address of your PC to 192.168.0.11, and plug the camera into your PC. The Camera found LED must be on for this tool to work. If the LED is off, reset the camera and check the IP address of your PC. **Troubleshooting the camera:** If you need to reset the camera to its default IP address, hold down the Camera Connection on Robot ● Robot Radio 2853 Team ID ○ 8-slot CompactRIO Ethernet port 2 ● Camera found at 192.168.0.90 ? Camera not found? 1 2 Apply Close

4) The camera's IP is 10.TE.AM.11 in which TE and AM are the first two and last two digits of your team number respectively. To connect to the live feed and to the camera in future times, change the computer's IP to 10.TE.AM.YY, where YY is two random numbers that isn't being used on your network.	 Internet Protocol Version 4 (TCP/IPv4) Properties General You can get IP settings assigned automatically if your network supports this capability. Otherwise, you need to ask your network administrator for the appropriate IP settings. ○ Obtain an IP address automatically ● Use the following IP address: IP address: 10 . 28 . 53 . 12 Subnet mask: 255 . 0 . 0 . 0 Default gateway: . . . ○ Obtain DNS server address automatically ● Use the following DNS server addresses: Preferred DNS server: . . . Alternate DNS server: . . . ☐ Validate settings upon exit Advanced... OK Cancel
5) Open the Live Feed by opening the browser and typing the IP into the address bar. You will be greeted by this screen. Setup a root password	 AXIS Configure Root Password User name: root Password: Confirm password: OK The password for the pre-configured administrator root must be changed before the product can be used. If the password for root is lost, the product must be reset to the factory default settings, by pressing the button located in the product's casing. Please see the user documentation for more information.
6) Go to Setup to setup other settings of your camera	 AXIS M1011 Network Camera Live View Setup Help Snapshot

LiveFeed

But what if I don’t want to visual process, what if I just want live feed for this year’s game? Well then, we can also help you there…

➠ The Code

```
#include "WPILib.h"

class AxisCameraSample : public SampleRobot
{
   AxisCamera *camera;

public:
   void RobotInit() override
   {
      camera = new AxisCamera("10.28.53.103");
   }

   void OperatorControl() override
   {
      camera->WriteResolution(AxisCamera::kResolution_160x120);
      camera->WriteColorLevel(50);
      camera->WriteCompression(0);
      camera->WriteMaxFPS(30);
   }
};

START_ROBOT_CLASS(AxisCameraSample);
```

➠ The Explanation

```
class AxisCameraSample : public SampleRobot
{
   AxisCamera *camera;

public:
   void RobotInit() override
   {
      camera = new AxisCamera("10.28.53.103");
   }
```

This is the declaring and initializing the camera. The Axis camera is declared with the line AxisCamera *camera which it is using a pointer. To initialize the camera use **new** to allocate the memory for the IP address of the camera. In the sample code for the camera, it states the

IP address of the camera is 10.TE.AM.103 with “TE” and “AM” being the first two and last two digits of your team number.

```
void OperatorControl() override
{
   camera->WriteResolution(AxisCamera::kResolution_160x120);
   camera->WriteColorLevel(50);
   camera->WriteCompression(0);
   camera->WriteMaxFPS(30);
}
```

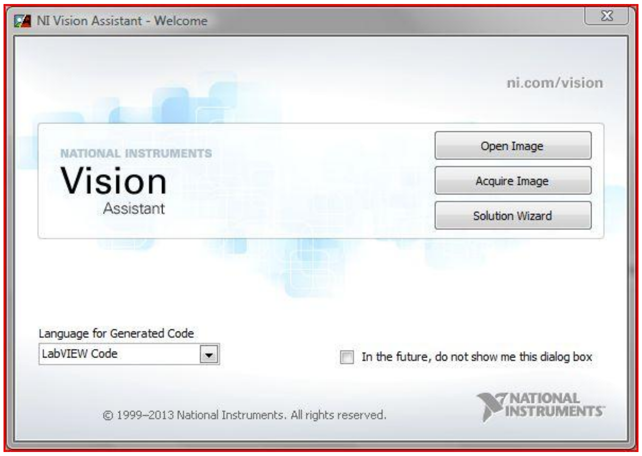
This is settings configuration of the camera. **WriteResolution()** writes the resolution of the camera from the options of kResolution_160x120, kResolution_176x144, kResolution_240x180, kResolution_320x240, kResolution_480x360, and kResolution_640x480 which are specific to the AxisCamera class. **WriteColorLevel()** takes in an integer to set the color level of the camera. Setting the value to 0 turns the image to black and white, while 100 is in full color. **WriteCompression()** also takes in an integer to set how much the image gets compressed. At a compression of 0, it is at its best quality, and a compression of 100, it is at its worst quality. **WriteMaxFPS()** quite obviously sets the highest frames per seconds that the camera will reach. The input taken in from this is an integer that maxes out at at 30 FPS for the Axis cameras.

Developing Code for the Camera

Using NI Vision Assistant

➠ Imaging Functions

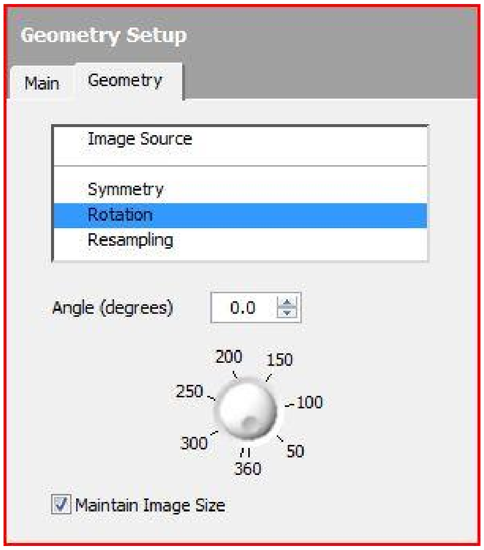
Important functions to keep in mind



Language for Generated Code: LabVIEW Code

□ In the future, do not show me this dialog box

Geometry



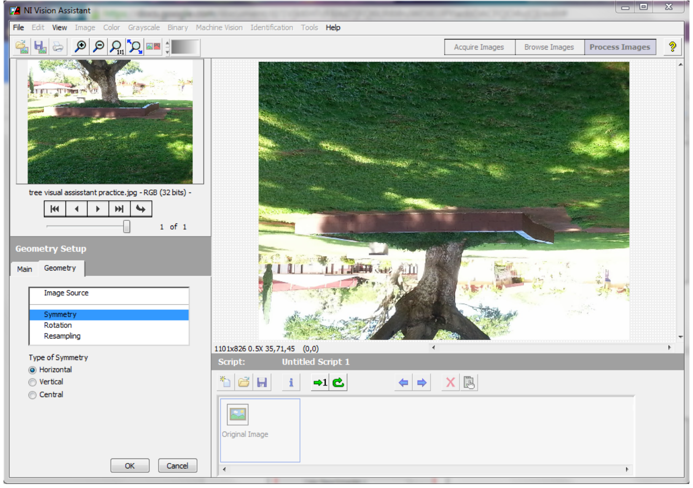
Angle (degrees) 0.0

☑ Maintain Image Size

Changes orientation of image

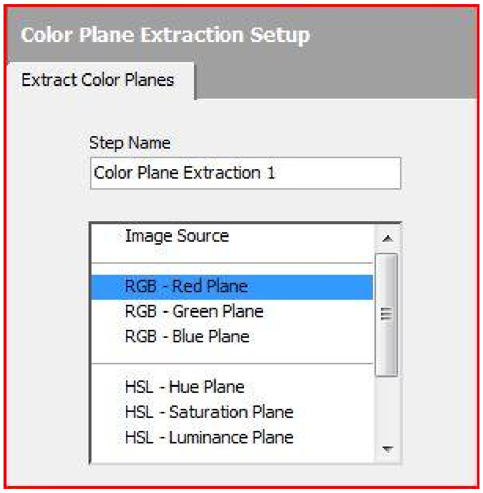
Usage

- Choose setting (symmetry, rotation, resampling)
- Manipulate using the specific options in each setting



Type of Symmetry
● Horizontal
○ Vertical
○ Central

Color Plane Extraction



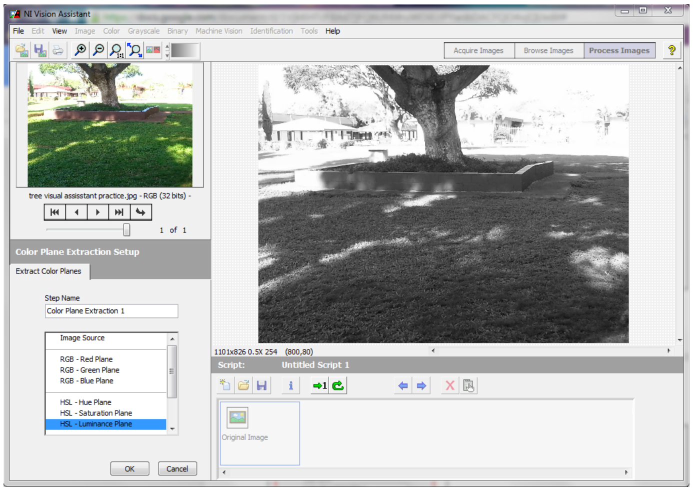
Step Name: Color Plane Extraction 1

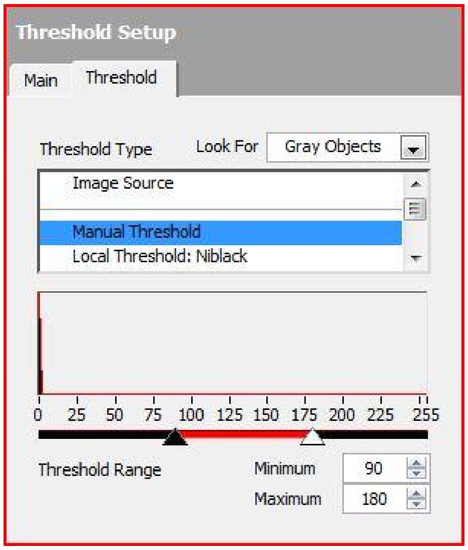
Image Source:
- RGB - Red Plane (selected)
- RGB - Green Plane
- RGB - Blue Plane
- HSL - Hue Plane
- HSL - Saturation Plane
- HSL - Luminance Plane

Extract 3 color planes from an image (RBG, HSV, HSL)

Usage

- Click any option to remove that pallet and make the image grayscale



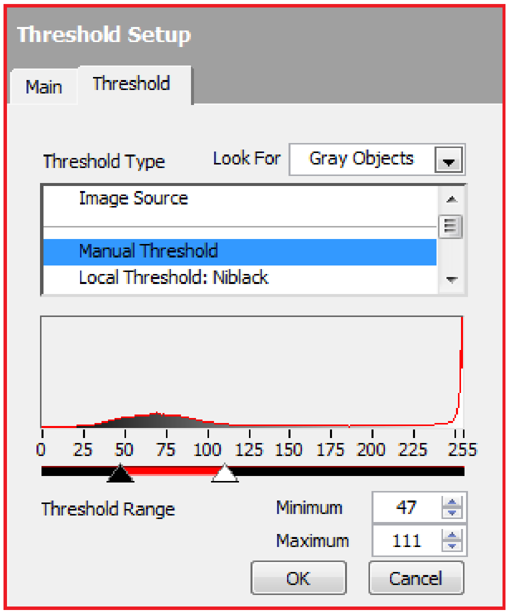


Threshold Type — Look For: Gray Objects

Image Source:
- Manual Threshold (selected)
- Local Threshold: Niblack

Threshold Range — Minimum: 90, Maximum: 180

Threshold



Threshold Type: Manual Threshold

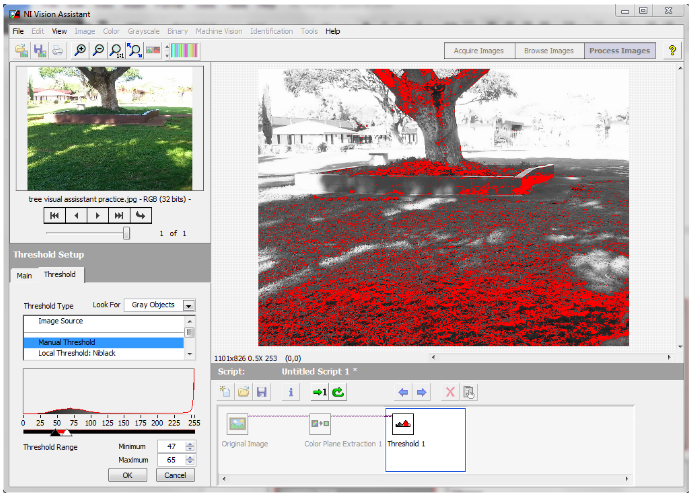
Look For: Gray Objects

Threshold Range: Minimum 47, Maximum 111

Allows you to look for objects of a specific brightness in a 16-bit image

Usage

- Choose to look for bright objects, dark objects or gray objects
- Adjust sliders until you get desired results

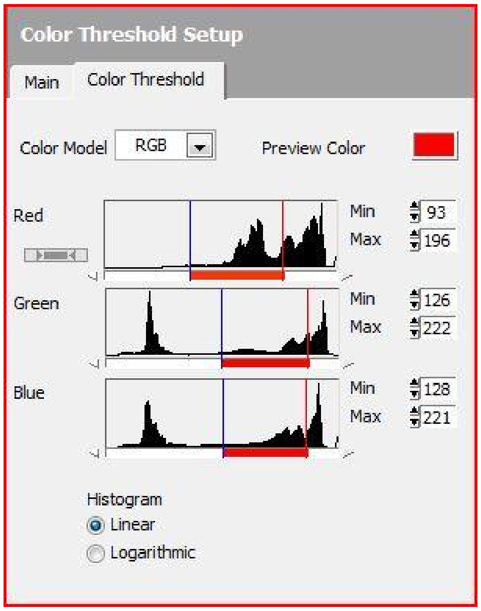


Threshold Type: Manual Threshold

Look For: Gray Objects

Threshold Range: Minimum 47, Maximum 65

Color Threshold



Color Model: RGB

Preview Color: 🟥

Red: Min 93, Max 196

Green: Min 126, Max 222

Blue: Min 128, Max 221

Histogram

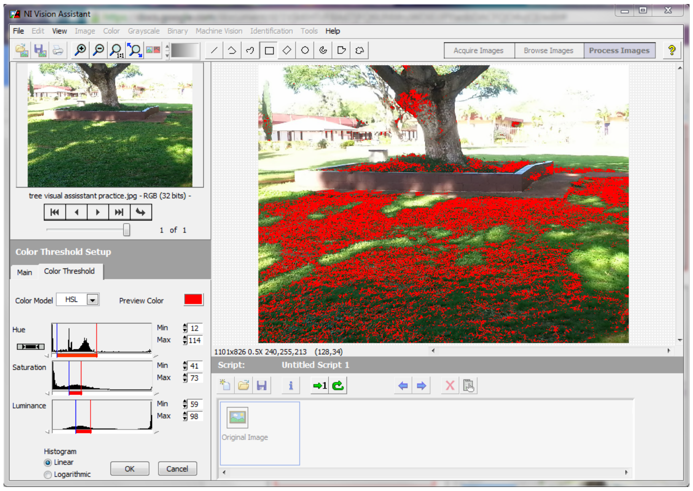
● Linear

○ Logarithmic

Allows you to look for a specific colored or bright object

Usage

- Move RGB/HS sliders for maximum
- Move other RGB/HS sliders for minimum
- Adjust until desired object selected


NI Vision Assistant
File Edit View Image Color Grayscale Binary Machine Vision Identification Tools Help
Acquire Images
Browse Images
Process Images
tree visual assistant practice.jpg - RGB (32 bits) -
1 of 1
Color Threshold Setup
Main
Color Threshold
OK
Cancel
1101x826 0.5X 240,255,213 (128,34)
Script: Untitled Script 1
Original Image

Color Model: HSL

Preview Color: 🟥

Hue: Min 12, Max 114

Saturation: Min 41, Max 73

Luminance: Min 59, Max 98

Histogram

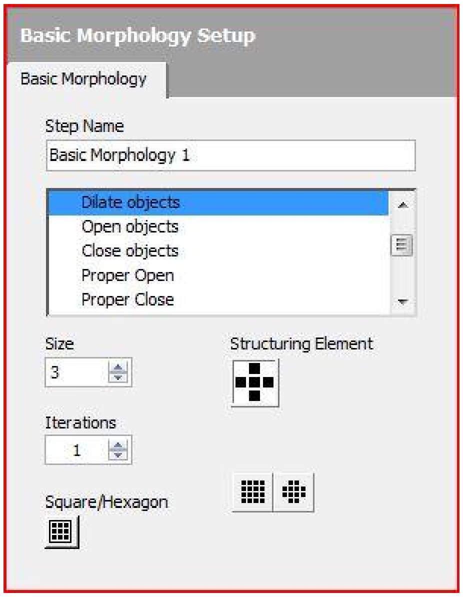
● Linear

○ Logarithmic

Basic Morphology

Modifies binary objects in shapes

Options



Basic Morphology Setup

Basic Morphology

Step Name: Basic Morphology 1

- Dilate objects
- Open objects
- Close objects
- Proper Open
- Proper Close

Size: 3

Structuring Element

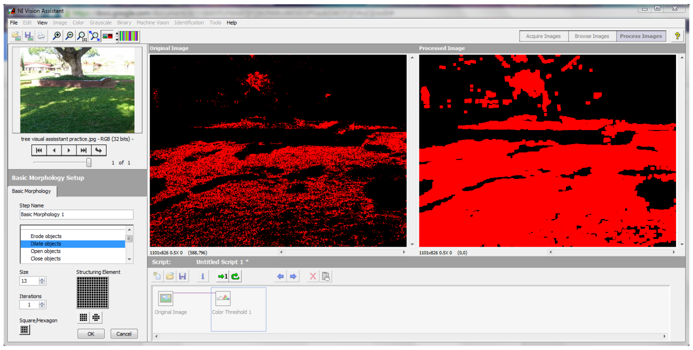
Iterations: 1

Square/Hexagon

Erode, Dilate, Open, Close, Proper Open, Proper Close, Gradient In, Gradient Out, Thin, Thick , Auto Median

Usage

- Play around with the options until you get the desired results



tree visual assisstant practice.jpg - RGB (32 bits) -

1 of 1

Basic Morphology Setup

Basic Morphology

Step Name: Basic Morphology 1

- Erode objects
- Dilate objects
- Open objects
- Close objects

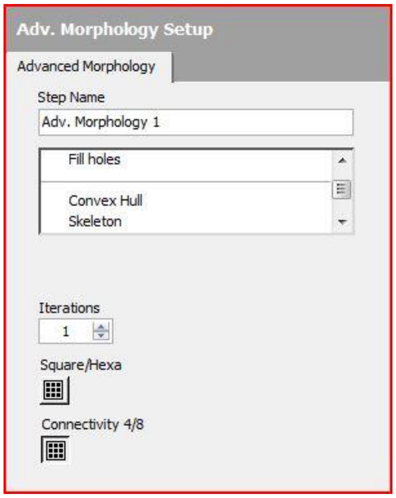
Size: 13

Iterations: 1

Structuring Element

Square/Hexagon

Advanced Morphology



Adv. Morphology Setup

Advanced Morphology

Step Name: Adv. Morphology 1

Fill holes
Convex Hull
Skeleton

Iterations: 1

Square/Hexa

Connectivity 4/8

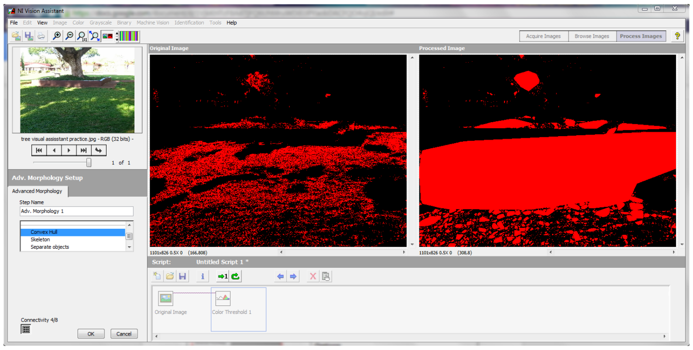
Perform high level operations on blobs in binary objects

Options

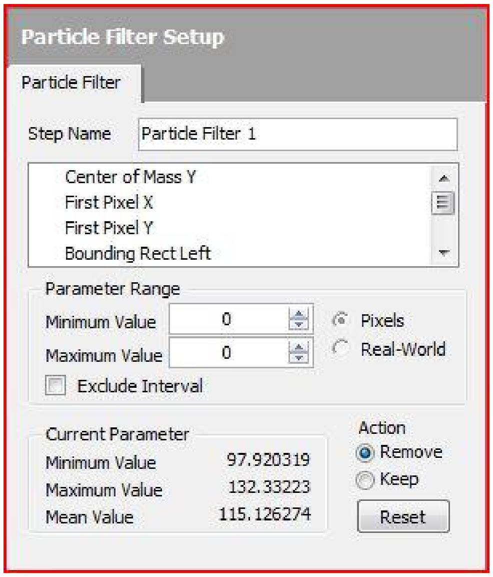
Remove small objects, Remove large objects, Remove border objects, Fill holes, Convex hull, Skeleton, Separate objects, Label objects, Distance, Danielsson, Segment Image

Usage

- Play with these options as well until you get the desired results



Particle Filter



Particle Filter Setup

Particle Filter

Step Name: Particle Filter 1

Center of Mass Y
First Pixel X
First Pixel Y
Bounding Rect Left

Parameter Range
Minimum Value: 0
Maximum Value: 0
● Pixels
○ Real-World
□ Exclude Interval

Current Parameter
Minimum Value: 97.920319
Maximum Value: 132.33223
Mean Value: 115.126274

Action
● Remove
○ Keep

Reset

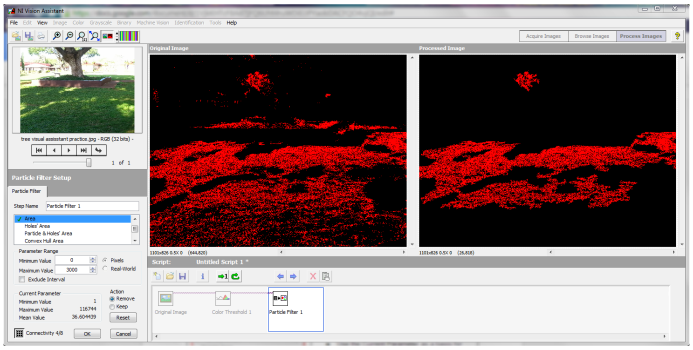
Filters out particles by certain parameters

Usage

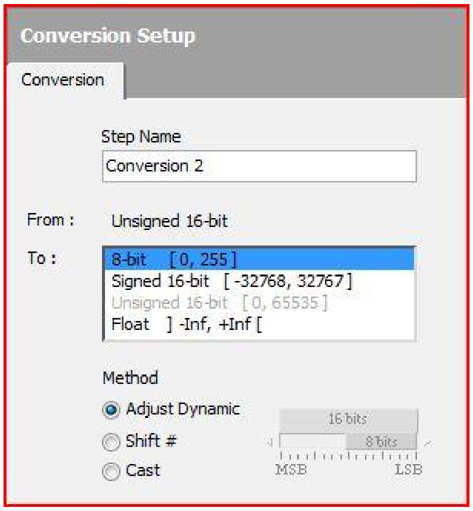
- Select parameter
- Set the parameter range
- Set the action

Notes

- Use the Current Parameter as a basis for your parameters



Bit Conversion



Conversion Setup

Conversion

Step Name: Conversion 2

From : Unsigned 16-bit

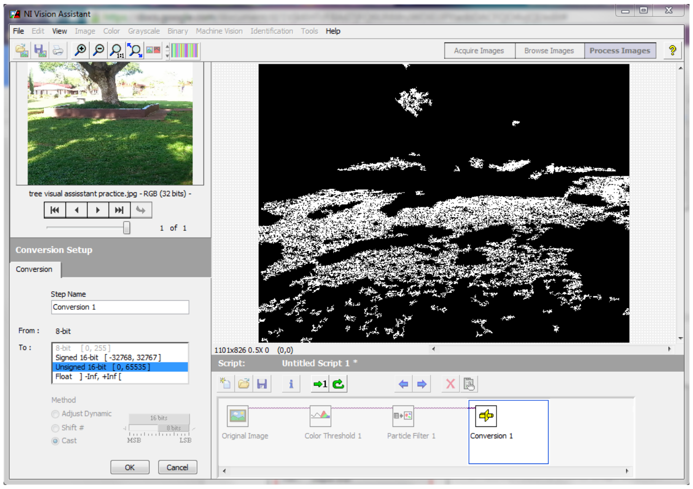
To :
- 8-bit [0, 255]
- Signed 16-bit [-32768, 32767]
- Unsigned 16-bit [0, 65535]
- Float] -Inf, +Inf [

Method
- ● Adjust Dynamic
- ○ Shift #
- ○ Cast

Converts image between 16-bit and 8-bit (and float as well)

Usage

- After threshold and morphology convert into 16-bit (signed or unsigned)
- Convert back to 8-bit for shape detection



tree visual assisstant practice.jpg - RGB (32 bits) -

Conversion Setup

Conversion

Step Name: Conversion 1

From : 8-bit

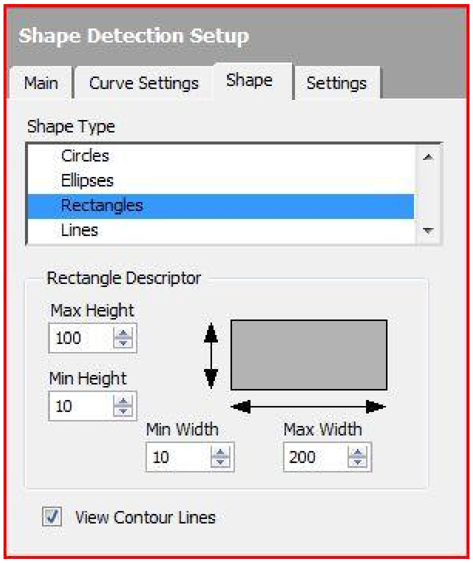
To :
- 8-bit [0, 255]
- Signed 16-bit [-32768, 32767]
- Unsigned 16-bit [0, 65535]
- Float] -Inf, +Inf [

Method
- ○ Adjust Dynamic
- ○ Shift #
- ● Cast

1101x826 0.5X 0 (0,0)

Script: Untitled Script 1 *

Shape Detection Setup



Shape Detection Setup

Shape Type
- Circles
- Ellipses
- Rectangles
- Lines

Rectangle Descriptor

Max Height: 100

Min Height: 10

Min Width: 10

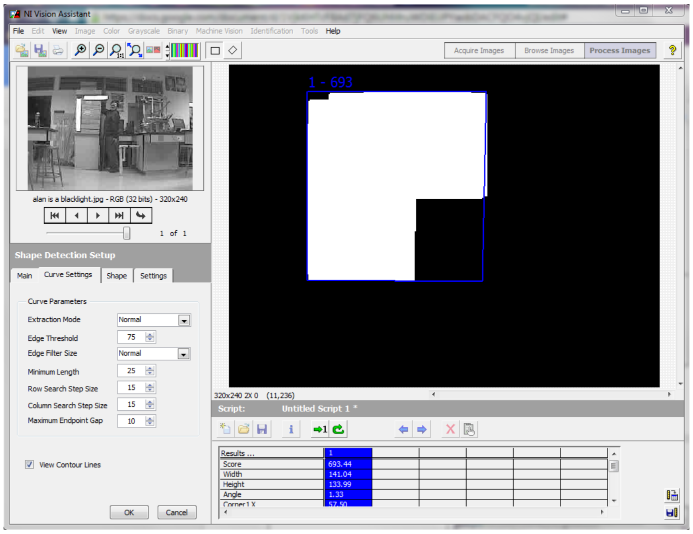
Max Width: 200

☑ View Contour Lines

Detects shapes in image or region of interest

Usage

- Choose desired shape that is being searched for (circle, ellipse, rectangle, or line)
- Then select shape descriptor (max/min dimensions)



alan is a blacklight.jpg - RGB (32 bits) - 320x240

Shape Detection Setup

Curve Parameters

Extraction Mode	Normal
Edge Threshold	75
Edge Filter Size	Normal
Minimum Length	25
Row Search Step Size	15
Column Search Step Size	15
Maximum Endpoint Gap	10

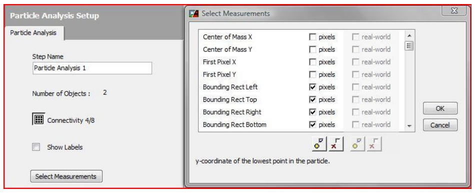
☑ View Contour Lines

320x240 2X 0 (11,236)

Script: Untitled Script 1 *

Results ...	1
Score	693.44
Width	141.04
Height	133.99
Angle	1.33
Corner 1 X	57.50

Particle Analysis



Particle Analysis Setup

Particle Analysis

Step Name: Particle Analysis 1

Number of Objects : 2

Connectivity 4/8

□ Show Labels

Select Measurements

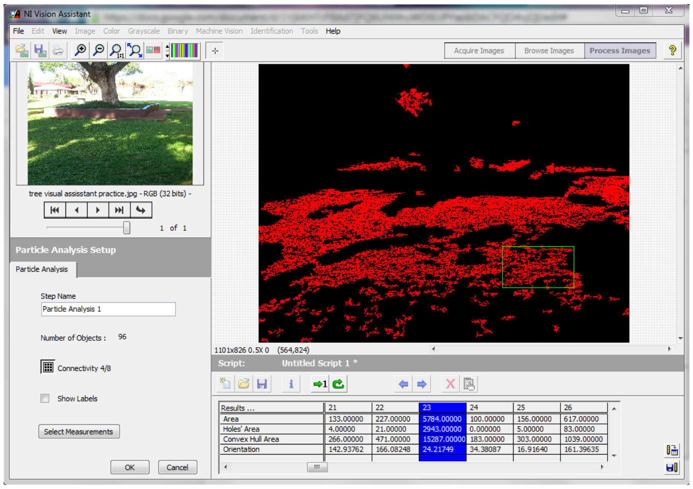
Select Measurements		
Center of Mass X	□ pixels	□ real-world
Center of Mass Y	□ pixels	□ real-world
First Pixel X	□ pixels	□ real-world
First Pixel Y	□ pixels	□ real-world
Bounding Rect Left	☑ pixels	□ real-world
Bounding Rect Top	☑ pixels	□ real-world
Bounding Rect Right	☑ pixels	□ real-world
Bounding Rect Bottom	☑ pixels	□ real-world

y-coordinate of the lowest point in the particle.

Shows measurements of particles

Usage

- Select measurements
- Look for measurements and use them at your will



tree visual assisstant practice.jpg - RGB (32 bits) -

1 of 1

Particle Analysis Setup

Particle Analysis

Step Name: Particle Analysis 1

Number of Objects : 96

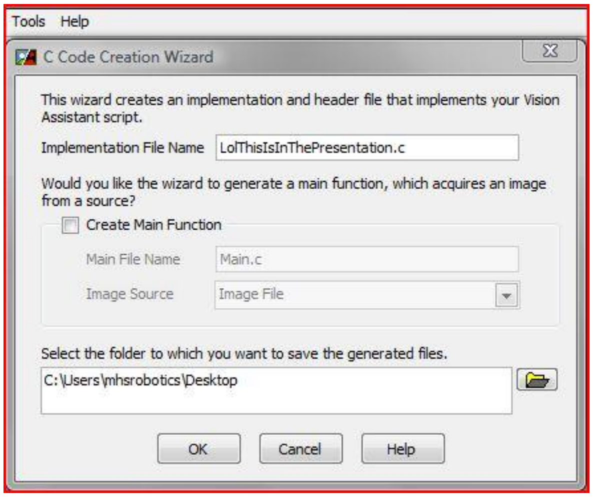
Connectivity 4/8

□ Show Labels

1101x826 0.5X 0 (564,824)

Script: Untitled Script 1 *

Results ...	21	22	23	24	25	26
Area	133.00000	227.00000	5784.00000	100.00000	156.00000	617.00000
Holes' Area	4.00000	21.00000	2943.00000	0.000000	5.00000	83.00000
Convex Hull Area	266.00000	471.00000	15287.00000	183.00000	303.00000	1039.00000
Orientation	142.93762	166.08248	24.21749	34.38087	16.91640	161.39635



C Code Creation Wizard

This wizard creates an implementation and header file that implements your Vision Assistant script.

Implementation File Name: LolThisIsInThePresentation.c

Would you like the wizard to generate a main function, which acquires an image from a source?

□ Create Main Function

Main File Name: Main.c

Image Source: Image File

Select the folder to which you want to save the generated files.

C:\Users\mhsrobotics\Desktop

Exporting the code to Objective C, Java, or CNET

1. Click Tools Tab
2. Click Create LABView Code, C Code, or .NET Code
3. Name, set the directory, then apply

Developing Camera Code in C++

Jump to →

Introduction

Programming an axis camera to retrieve the data you want can sometimes be difficult, so be careful when setting up parameters. As an additional warning, be aware that the robot’s cRIO does have memory limitations so be careful if you choose to code using pointers and arrays as these could easily cause the cRIO to crash if used improperly.

At this point, you should already have a basic script written in the NI Vision Assistant (that hopefully works). Although it’s a rather friendly interpretation, your next step is to simply translate this script into C++ using the WindRiver Workbench. When translating, it is important to remember the same parameters and values that were found to be effective in the Vision Assistant should be kept in order to expect similar results.

To detect an object, you will need to:

- Initialize a camera
- Create a format
- Employ a broad filter (HSL is recommended but RGB is also viable)
- Employ a more specific filter (a small particle filter)
- Detect the number of particles that remain
- Measure the remaining particles (imaqParticleFilter4)
- Print results to the Driver Station

➠ The Code

The camera code in it’s entirety:

```
#include “WPILib.h”
#include “stdlib.h”
#include “math.h”
#include “nivision.h”
#include “WPILib.h”

class RobotDemo : public SampleRobot
{
    RobotDrive camera; /* Allows use of the camera class, only necessary
     if you’re using method 2 for Camera Initialization */
    public:
        RobotDemo(void):
            camera(AxisCamera::GetInstance(“10.28.53.11”)) /*
             Initializes your camera and establishes its connection
             to the camera’s network*/

        {
        camera.WriteResolution(AxisCamera::kResolution_320x240); /* Sets
         the format of the camera so that it can be used properly */
        }
    void Autonomous(void)
    {
    }
    void OperatorControl(void)
    {
        AxisCamera &camera = AxisCamera::GetInstance("10.28.53.11"); /*
     Initializes your camera and establishes its connection to
     the camera’s network */
        camera.WriteResolution(AxisCamera::kResolution_320x240); /* Sets
         the format of the camera so that it can be used properly */
        while (IsOperatorControl())
        {
            HSLImage* imageOne = camera.GetImage(); /* Takes a raw
       snapshot from the camera and sends it to the variable
       named imageOne*/
            BinaryImage* binIMG = NULL; /* Creates a Binary Image
             shell
                                         in 8 bit */
            binIMG = imageOne -> ThresholdHSL(0, 255, 0, 255, 235, 255);
                /* Fills the Binary Image shell with a filtered
                version of the snapshot taken from the camera */
            delete imageOne; /* Deletes the initial raw snapshot */
            Image* imageOneModified = imaqCreateImage(IMAQ_IMAGE_U8, 0);
                /* Sends the resulting binary image to a variable named
                imageOneModified */
        ParticleFilterCriteria2 particleCriteria; /* Must be
                specified to use imaqParticleFilter4 */
        ParticleFilterOptions2 particleFilterOptions; /* The
        options under ParticleFilterCriteria2 */
            int numParticles; /* Assigning a variable for the number
          of particles */
            particleCriteria.parameter = IMAQ_MT_AREA; /* Optional
                parameter kept to comply with Filter Options */
            particleCriteria.lower = pLower; /* Filter a min height */
            particleCriteria.upper = pUpper; /* Filter a max height */
            particleCriteria.calibrated = FALSE; /* Unchanged state */
            particleCriteria.exclude = TRUE; /* True to exclude
                particles that do not fit in the parameters set */
            particleFilterOptions.rejectMatches = rejectMatches; /*
                Option for number of particles to exclude at a time */
            particleFilterOptions.rejectBorder = 0; /* Optional
                parameter kept to comply with Filter Options */
            particleFilterOptions.connectivity8 = connectivity; /*
                Allows for 8-bit conversion and formatting */

            float pLower = 175; // The minimum height for a particle
            float pUpper = 500; // The maximum height for a particle
            int criteriaCount = 1; // number of elements to include/exclude at a time
            int rejectMatches = 1; // when set to true (1), particles that do not meet the
            criteria are discarded
            int connectivity = 1; // declares connectivity value as 1; so corners are not
            ignored
            int removeSmallParticles; // removes small blobs
            int borderSetting;       // variable to store border settings, limit for rectangle
            int cloningDevice; // we create another image because the
            ParticleMeasuring steals the image from particlecounter
            int borderSize = 1;  // border for the camera frame so that Driverstation is
            happy

            removeSmallParticles = imaqParticleFilter4(imageOneModified,
                binIMG -> GetImaqImage(), &particleCriteria,
                criteriaCount, &particleFilterOptions, NULL,
                &numParticles); /* Creating a small particle filter
                based imaqParticleFilter4 from the WPI Library for NI Vision*/

            int ParticleCounter; // Function to count particles
            int countparticles; // Variable to store particle values
            int countparticlesTwo; // Variable to store particle
             values
            ParticleCounter = imaqCountParticles(imageOneModified,
             TRUE, &countparticles);
            ParticleCounter = imaqCountParticles(imageOneModifiedCopy,
                TRUE, &countparticlesTwo);

            int leftFinder; // Function to find a particle’s location
            int rightFinder; // Function to find a particle’s location
            double rightLocation; // Variable to store location values
            double leftLocation; // Variable to store location values
            leftFinder = imaqMeasureParticle(Kirby, 0, FALSE,
                IMAQ_MT_MAX_HORIZ_SEGMENT_LENGTH_LEFT,
                &leftLocation);
            rightFinder = imaqMeasureParticle(Kirby, 0, FALSE,
                IMAQ_MT_MAX_HORIZ_SEGMENT_LENGTH_RIGHT,
                &rightLocation);
            int TinyRuler; // Function to find a particle’s width
            int BabyYardstick; // Function to find a particle’s width
            double particleOneWidth; // Variable to store the
             particle’s width */
            double particleTwoWidth); /* Variable to store the
             particle’s width */
            TinyRuler = imaqMeasureParticle(imageOneModified, 0,
             FALSE,
                IMAQ_MT_BOUNDING_RECT_WIDTH, &particleOneWidth); /*
                Measures particle 1's width and sends a value back to the variable
                ParticleOneWidth */
            BabyYardstick = imaqMeasureParticle(imageOneModified, 1,
                FALSE, IMAQ_MT_BOUNDING_RECT_WIDTH,
                &particleTwoWidth); /* Measures particle 2's width and sends a
                value back to the variable ParticleTwoWidth */
            imaqDispose(imageOneModified); // Deletes the snapshot
            imaqDispose(imageOneModifiedCopy); // Deletes the snapshot

        }
    }
    void Test()
    {
    }
};
START_ROBOT_CLASS(RobotDemo); // Runs all code under the RobotDemo function
```

➠ The Explanation

Breakdown of the code:

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

As usual, you'll want to include the WPI Library. For camera code, the WPI Library also contains the library for NI Vision which will make it much easier to manipulate the camera's output. You'll be able to use the same imaging tools in the vision assistant, which means that you can capture images in HSL format and use NI Vision's particle filtering criteria.

```
#include "stdlib.h"
```

Importing standard functions isn't necessary when using NI Vision, but should be included if your team is using RoboRealm instead of NI Vision Assistant.

```
#include "math.h"
```

Importing the math library will allow for extended calculations, ranging from the use of absolute values to functions that will measure the distance of various objects from the camera.

Camera Initialization

➠ **Method 1**

Declaring the Camera as an Object in a Loop:

```
// Method 1 of Camera Initializaiton
void OperatorControl(void)
   {
         AxisCamera &camera = AxisCamera::GetInstance("10.28.53.11");
         camera.WriteResolution(AxisCamera::kResolution_320x240);
      }
```

Before an Axis camera can be used, the camera must be declared as an object (Which will be a piece of hardware throughout robotics). Through this method, initialization can be done under any of the various Robot modes including **OperatorControl**, **Autonomous**, and **Test**. Here the **AxisCamera** is declared under **OperatorControl** while its corresponding network IP address is specified. Its resolution is set using the function **WriteResolution** so that the cRIO doesn't crash from processing the large and high quality default image size.

➠ **Method 2**

The Traditional Method of Declaring the Camera:

```
// Method 2 of Camera Initialization
class RobotDemo : public SampleRobot
{
        RobotDrive camera;
        public:
                RobotDemo(void):
                        camera(AxisCamera::GetInstance("10.28.53.11"))
                {
                camera.WriteResolution(AxisCamera::kResolution_320x240);
                }
```

RobotDrive as a class contains most of the options that can be used to enable FRC robot hardware, and among those options exist a layout to declare a camera. In the same way other objects are declared under **class** & **public** and then specified later with corresponding ports under the **RobotDemo** function, the camera is declared as a piece of equipment and then is specified with a specific with its corresponding IP address. In addition to this specification, the camera's resolution is set and is placed in the expirations curly brace because **WriteResolution** is actually a function under the WPI Library and is not a declaration of an object.

Filling The Image

➠ HSL

A Rough Filter Based on Hue, Saturation, and Luminance:

```
void OperatorControl(void)
    {
while (IsOperatorControl())
                    {
                    HSLImage* imageOne;
                    imageOne = camera.GetImage();
               }
          }
```

Note: From here on, code is placed under while (IsOperatorControl()) so that the camera and cRIO continually process images in real time.

Essentially, these lines take a snapshot from the camera live feed and put the snapshot in HSL format. This modified snapshot is assigned to a variable called **imageOne** that will hold this **HSLImage**. So far, **imageOne** is unmodified and is simply placed into an HSL format so that it can be later modified and filtered to the program's needs.

```
void OperatorControl(void)
    {
while (IsOperatorControl())
                    {
                    BinaryImage* binIMG = NULL;
                    binIMG = imageOne -> ThresholdHSL(0, 255, 0, 255, 235,
                       255);
                    delete imageOne;
                    Image* imageOneModified = imaqCreateImage(IMAQ_IMAGE_U8,
                       0);
               }
          }
```

The cRIO on a robot can only process images in a binary 8-bit format because of its limitations on memory. Here, a shell of a **BinaryImage** is created and named as **binIMG** (**NULL** means 8-bit format). Once the shell is created, it is possible to fill the shell with **imageOne**. Now that in an 8-bit format, on the same line **imageOne** is filtered using the function **ThresholdHSL**. These ideal minimum and maximum values of hue, saturation, and luminance are when tinkering with the NI Vision Assistant. It is a good idea to then **delete** **imageOne** before continuing so that you free up some memory on the cRIO. To finish this process, a new image with a new name is created from this same 8-bit format to allow for further image morphology.

➠ Particle Filter

A Fine Filter Based on Particle Size:

```
void OperatorControl(void)
    {
while (IsOperatorControl())
                {
                ParticleFilterCriteria2 particleCriteria;

                ParticleFilterOptions2 particleFilterOptions;
                    int numParticles;
                            particleCriteria.parameter = IMAQ_MT_AREA;
                        particleCriteria.lower = pLower;
                    particleCriteria.upper = pUpper;
                    particleCriteria.calibrated = FALSE;
                    particleCriteria.exclude = TRUE;
                        particleFilterOptions.rejectMatches = rejectMatches;
                        particleFilterOptions.rejectBorder = 0;
                        particleFilterOptions.connectivity8 = connectivity;
            }
        }
```

ParticleFilterCriteria2 must be set with the according parameters in **ParticleFilterOptions2** to allow image morphology. Here, these options specify all of the requirements in the imaq class so that a particle filter can be used. Understand that the first filter function, **ThresholdHSL**, is an imperfect method of detection since there will still be some unwanted particles that survive this filter. The image must be cleaned further to perform calculations, so a particle filter is used to clean up remaining particles that are too small. These are all of the default parameters under **ParticleFilterOptions2**, and actual values for the unspecified **particleCriteria** are set later in the next step.

```
void OperatorControl(void)
    {
while (IsOperatorControl())
                  {
              float pLower = 175;
                           float pUpper = 500;
                           int criteriaCount = 1;
                           int rejectMatches = 1;
                           int connectivity = 1;
                           int removeSmallParticles;
                           int borderSetting;
                           int borderSize = 1;
              }
         }
```

Now that variables have been created to placehold the values for options and criteria involved in **ParticleFilterOptions2**, it is time to assign specific values to these variables. **pLower** and **pUpper** are the ranges set and found through NI Vision Assistant to filter out particles that do not fit inside this range. **criteriaCount** dictates the amount of particles to consider with each function. The **rejectMatches** parameter can be set to either 0 or 1. A value of 0 will keep only the particles that are not in the parameters set with **ParticleFilterOptions2** while a value of 1 will keep only the particles that are in the parameters set with **ParticleFilterOptions2**. **removeSmallParticles** is the name of the particle filter to be used (it is a user defined name) while **borderSetting** and **borderSize** are unused but kept to comply with the formatting requirements of **ParticleFilterOptions2**.

```
void OperatorControl(void)
    {
while (IsOperatorControl())
                  {
                  Polturgust3000 = imaqParticleFilter4(Kirby, binIMG ->
                     GetImaqImage(), &particleCriteria, criteriaCount,
                     &particleFilterOptions, NULL, &numParticles);
              }
         }
```

As a side note, **imaqParticleFilter4** is not a user defined name but is the actual function name taken from the WPI Library for NI Vision. The **removeSmallParticles** function that is created uses the **imaqParticleFilter4** function (that subsequently uses the options defined under **ParticleFilterOptions2**) to filter small particles out of the image.

```
void OperatorControl(void)
    {
while (IsOperatorControl())
                 {
             int ParticleCounter;
             int countparticles;
             int countparticlesTwo;
             ParticleCounter = imaqCountParticles(imageOneModified, TRUE,
                &countparticles);
             ParticleCounter = imaqCountParticles(imageOneModifiedCopy,
                TRUE, &countparticlesTwo);
             }
         }
```

ParticleCounter is the name of the function that will find return a value for the number of particles detected. **countparticles** and **countparticlesTwo** are stored as integers, and they will hold values for the amount of particles that **ParticleCounter** detects. Now variables have been set-up, the actual function that count the number of particles can be created. The **ParticleCounter** is used twice to return a value to both **countparticles** and **countparticlesTwo** for the number of particles found. (Two are set up so that the first variable can be used to measure these particles while the second variable can be used to directly print a value to Driver Station).

Measurements

➠ Location

Finding the X-Coordinate Location of a Particle

```
void OperatorControl(void)
    {
while (IsOperatorControl())
                   {
                   int leftFinder;
                   int rightFinder;
              double leftLocation;
                   double rightLocation;
                   leftFinder = imaqMeasureParticle(imageOneModified, 0,
                      FALSE,
                   IMAQ_MT_MAX_HORIZ_SEGMENT_LENGTH_LEFT, &leftLocation);
                   rightFinder = imaqMeasureParticle(imageOneModified, 0,
                      FALSE,
                   IMAQ_MT_MAX_HORIZ_SEGMENT_LENGTH_RIGHT, &rightLocation);
              // Note:Delete imageOneModified here using the line
                "imaqDispose(imageOneModified);"when you are done using
                measurement functions
              }
         }
```

leftFinder and **rightFinder** are the two separate functions that each measure their respective particle's X-Coordinate location (in terms of their bounding left rectangular segment). When the loop runs and **leftFinder** and **rightFinder** have completed their measurements, these measurements are sent to the variables assigned as **leftLocation** and **rightLocation**. Now, these two variables return their saved values as a **double** when called upon.

➠ **Width**

Finding the Width of a Particle:

```
void OperatorControl(void)
   {
while (IsOperatorControl())
               {
               int TinyRuler;
               int BabyYardstick;
           double unowidth;
               double doswidth;
               TinyRuler = imaqMeasureParticle(imageOneModified, 0,
              FALSE,
               IMAQ_MT_BOUNDING_RECT_WIDTH, &particleOneWidth);
               BabyYardstick = imaqMeasureParticle(imageOneModified, 1,
                  FALSE,
               IMAQ_MT_BOUNDING_RECT_WIDTH, &particleTwoWidth);
               imaqDispose(imageOneModified);
           }
       }
```

TinyRuler and **BabyYardstick** are the two separate functions that each measure their respective particle's width (in terms of the particle's bounding rectangle). When the loop runs and **TinyRuler** and **BabyYardstick** have completed their measurements, these measurements are sent to the variables assigned as **unowidth** and **doswidth**. Now, these two variables return their saved values as a **double** when called upon. At the end of your code, don't forget to delete the modified image to clear up memory on the cRIO with **imaqDispose**! From here, the variables formatted with **double** will still hold their values until the loop is run again so these values can be manipulated as needed.

Additional Steps

➠ Alternative Measurements

More on Measurement Syntax and Other Methods:

```
int myMeasurementFunction;
double leftLocation;
myMeasurementFunction = imaqMeasureParticle(imageOneModified, 0, FALSE,
   "Measurement_Type_Goes_Here!", &leftLocation);
imaqDispose(imageOneModified);
```

The parameters that exist for the **imaqMeasureParticle** function dictate that the function using it be declared as an integer. The parameters for the function also require that the following are specified: a source image (**imageOneModified)**, the bit format, the existence of Convex Hull, a measurement type, and the address of the variable that will placehold the value found through particle measurement. Once all the prior code is put into place, only the specified measurement type needs to be changed in order to use a different measurement type. From here, the placeholder variables can be called upon in the Driver Station Print for testing and can be used to detect particles of certain parameters.

➠ More Measurement Types

IMAQ_MT_AREA_BY_IMAGE_AREA

Percentage of the particle Area covering the Image Area.

IMAQ_MT_BOUNDING_RECT_BOTTOM

Y-coordinate of the lowest particle point.

IMAQ_MT_BOUNDING_RECT_HEIGHT

Distance between the y-coordinate of highest particle point and the y-coordinate of the lowest particle point.

IMAQ_MT_CENTER_OF_MASS_X

X-coordinate of the point representing the average position of the total particle mass, assuming every point in the particle has a constant density.

IMAQ_MT_IMAGE_AREA

Area of the image.

IMAQ_MT_MAX_FERET_DIAMETER_ORIENTATION

The angle of the line segment connecting the two perimeter points that are the furthest apart.

IMAQ_MT_MAX_HORIZ_SEGMENT_LENGTH_LEFT

X-coordinate of the leftmost pixel in the longest row of contiguous pixels in the particle.

IMAQ_MT_MAX_HORIZ_SEGMENT_LENGTH_RIGHT

X-coordinate of the rightmost pixel in the longest row of contiguous pixels in the particle.

IMAQ_MT_MAX_HORIZ_SEGMENT_LENGTH_ROW

Y-coordinate of all of the pixels in the longest row of contiguous pixels in the particle.

IMAQ_MT_ORIENTATION

The angle of the line that passes through the particle Center of Mass about which the particle has the lowest moment of inertia.

IMAQ_MT_PERIMETER

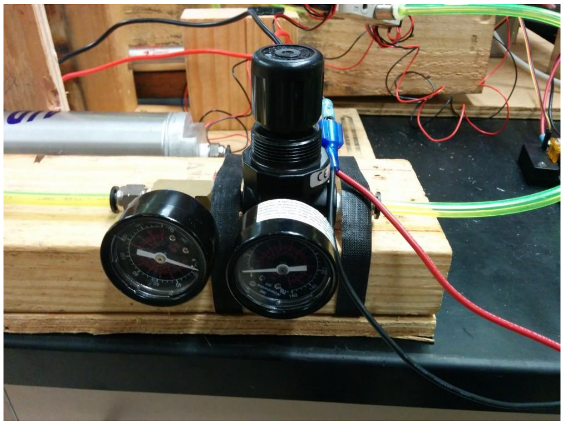
Length of the outer boundary of the particle.

Pneumatics

The Physical Layer

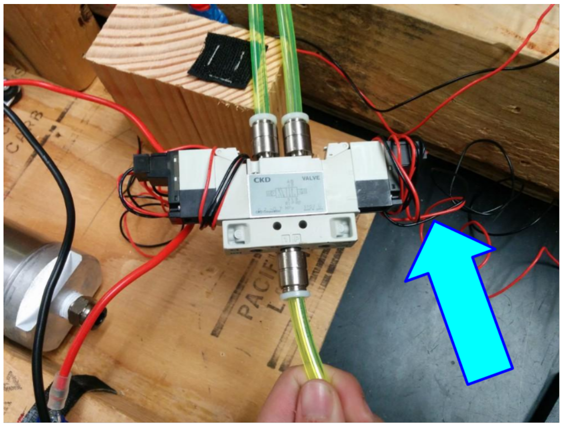
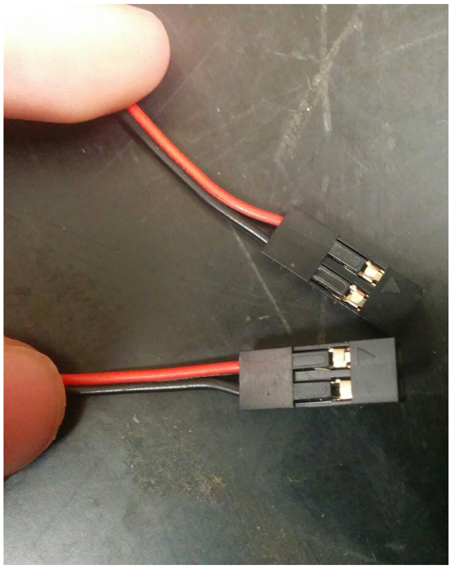
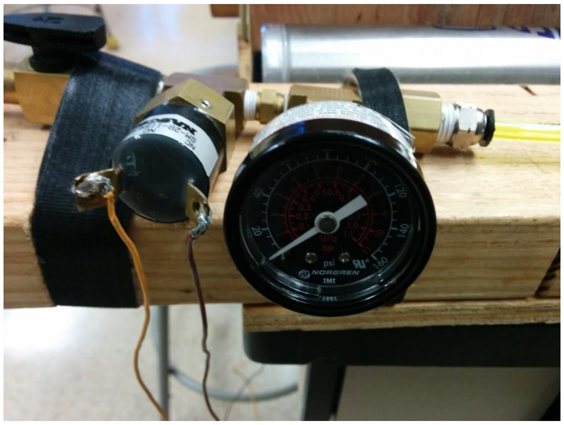
➠ Mechanical Components

	Compressor Increases air pressure by decreasing its volume to be used in pneumatic components Specifications: Weight: 2.39 lbs Size: 4.53” x 2.11” x 5.94” Voltage: 12 Volt DC Part #: 00090 [AndyMark]
	Air Tank Holds pressurized air to be used in pneumatic components Specifications: Weight: 0.56 lbs Length: 10.13” Diameter: 2.7” Capacity: 30.5 cubic inches (500 mL) Part #: am-2477 [AndyMark]

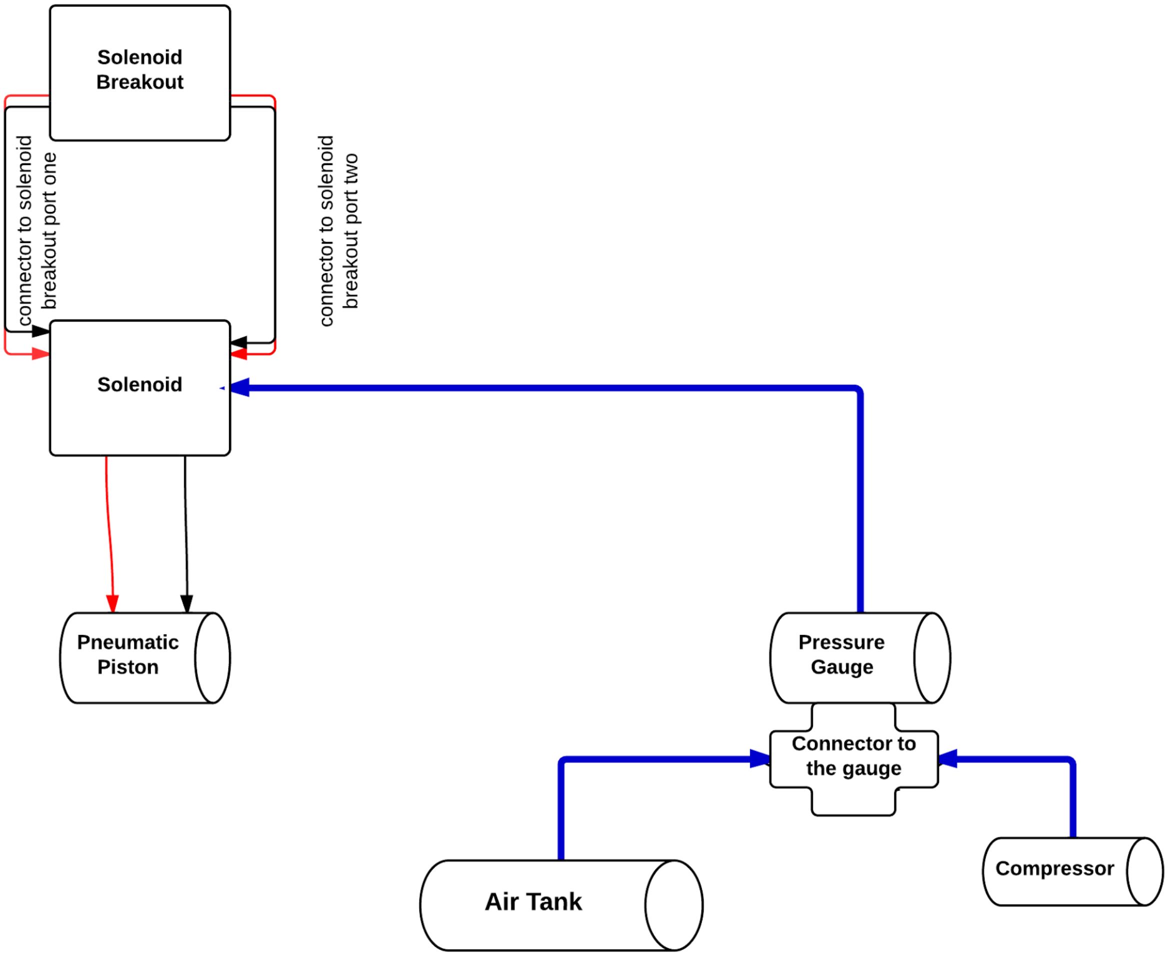
	**Mechanical Regulator** Uses a valve to automatically cut off the air flow at 120 PSI Specifications: Weight: 0.41 lbs Depth: 1.5” Height: 2.91” Part #: R07-100-RNEA AndyMark
	Pneumatic Piston (cylinder) Uses power from compressed air to produce force that has an equal force in a linear path Specifications: Weight: 0.45 lbs Length: 9.97” Part #: NCME075-0700 AndyMark

- Electrical Components

	Spike Simple controller that sends electrical signal for on/off or forward/backward Specifications: Weight: 0.12 lbs Size: 1.65” x 2.684” x 1.025” Part #: 217-0220 VEX

	**Double Solenoid** Opens and closes two valves to control air flow (can use with pistons) Specifications: Weight: 0.176 lbs Width: 0.47” Part #: 4GA120-M5-E01-4 **AndyMark**
	**2-Pin Cable** Connects solenoid to the solenoid breakout **AndyMark**
	**Electronic Pressure Gauge** Reads current pressure in air tank and sends electronic signal if pressure reaches 120 PSI Specifications: Weight: 0.12 lbs Length: 1.6” Diameter: 1.5” Part #: 18-013-212 **AndyMark**

Pneumatics Circuit



1. Connect the air tube from the air compressor and air tank to the gauge connector
2. Connect an air tube from the gauge connector to the pressure gauge
3. Hook up the digital sensor from the pressure gauge to the digital sidecard
4. Connect an air tube from the pressure gauge to the solenoid
5. Connect the solenoid to the analog breakout ports 1 and 2

NOTE: The solenoid used is a double solenoid. Double solenoids have two connections to the solenoid breakout while singles have one.

Pneumatics Code

➠ The Code

The Code

```
#include "WPILib.h"

class Robot : public SampleRobot
{
        Joystick stick; // only joystick
        Compressor compressor;
        Solenoid piston;
public:
        Robot() :
                        stick(0), //the joystick is in the first USB port
                        compressor(0),
                        piston(0)
        {
                compressor.SetClosedLoopControl(true);
        }
        void Autonomous()
        {
                while(IsAutonomous())
                {
                }
        }

        void OperatorControl()
        {
                while (IsOperatorControl())
                {
                        if(stick.GetRawButton(6)) //press the upper right trigger
                        {
                                piston.Set(true);
                        }
                        else if(stick.GetRawButton(8)) //press the lower right trigger
                        {
                                piston.Set(false);
                        }
```

```
				}
			}
			void Test()
			{
				while(IsTest())
				{
					//do stuff
				}
			}
};

START_ROBOT_CLASS(Robot);
```

➠ The Explanation

```
#include "WPILib.h"

class Robot : public SampleRobot
{
		Joystick stick; // only joystick
		Compressor compressor;
		Solenoid piston;
```

The operation of the compressor and double solenoid requires the inclusion of the **WPILib.h** class. Here, we declare the robot parts, including a joystick to mainpulate the pneumatics system that incorporates a compressor and a piston.

```
public:
		Robot() :
					stick(0), //the joystick is in the first USB port
					compressor(0),
					piston(0)
		{
			 compressor.SetClosedLoopControl(true);
		}
```

Assigning port numbers to each robot part. The **compressor** The **compressor** class only turns on the compressor when the PSI is <120 and turns off the compressor when the PSI hits 120 automatically.

```
void Autonomous()
        {
                while(IsAutonomous())
                {
                }
        }
```

Autonomous mode. (currently left blank)

```
void OperatorControl()
        {
                while (IsOperatorControl())
                {
                        if(stick.GetRawButton(6)) //press the upper right trigger
                        {
                                piston.Set(true);
                        }
                        else if(stick.GetRawButton(8)) //press the lower right trigger
                        {
                                piston.Set(false);
                        }
                }
        }
```

When button 6 is pressed, the solenoid will be engaged and push the pneumatic piston forward. When button 8 is pressed, the solenoid will disengage and the pneumatic piston will reset to its initial position.

```
void Test()
        {
                while(IsTest())
                {
                        //do stuff
                }
        }
};
```

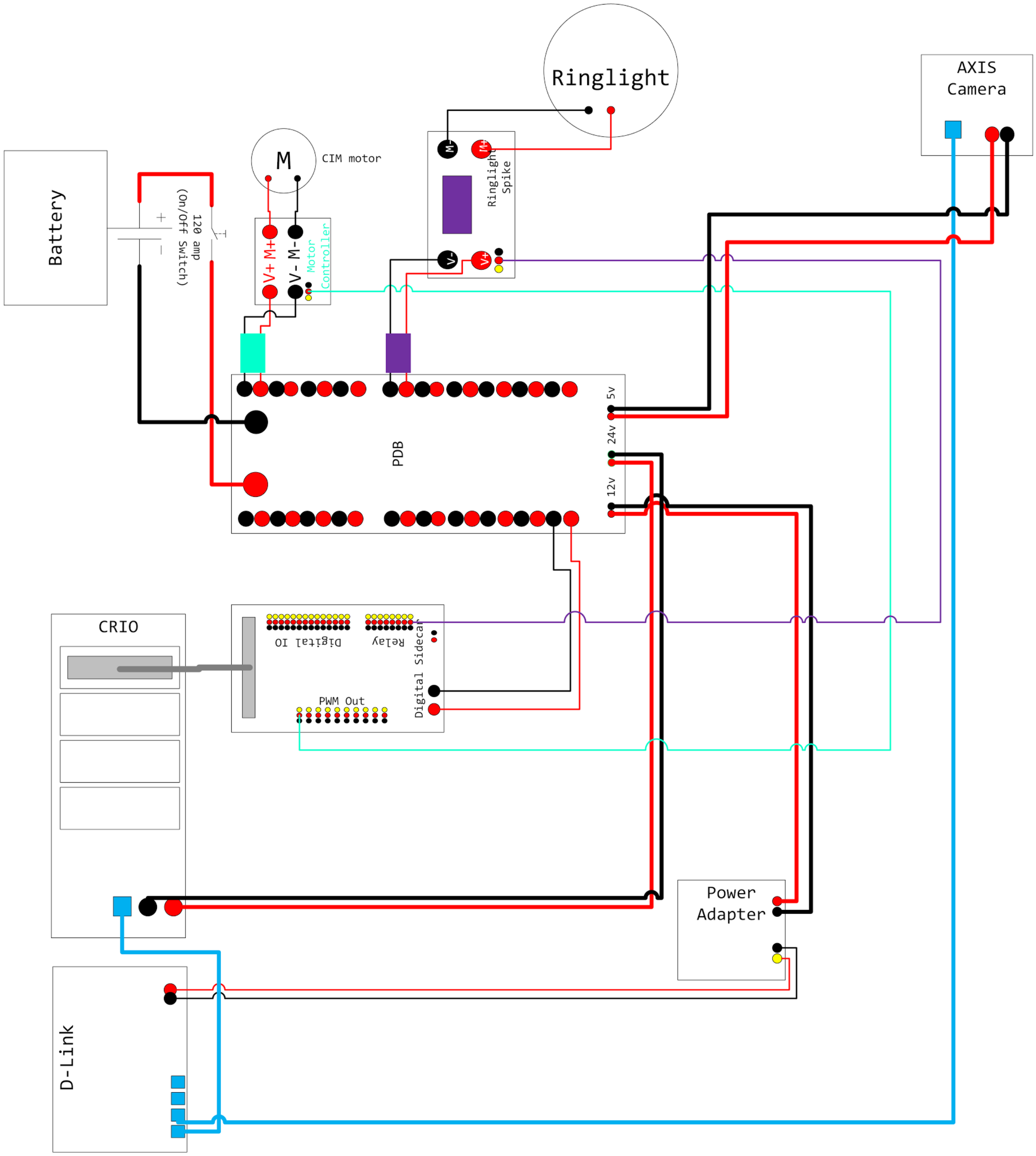
```
START_ROBOT_CLASS(Robot);
```

Test code and START_ROBOT_CLASS. Refer to the basic drive code documentation for explanation of these commands and modes

Appendixes

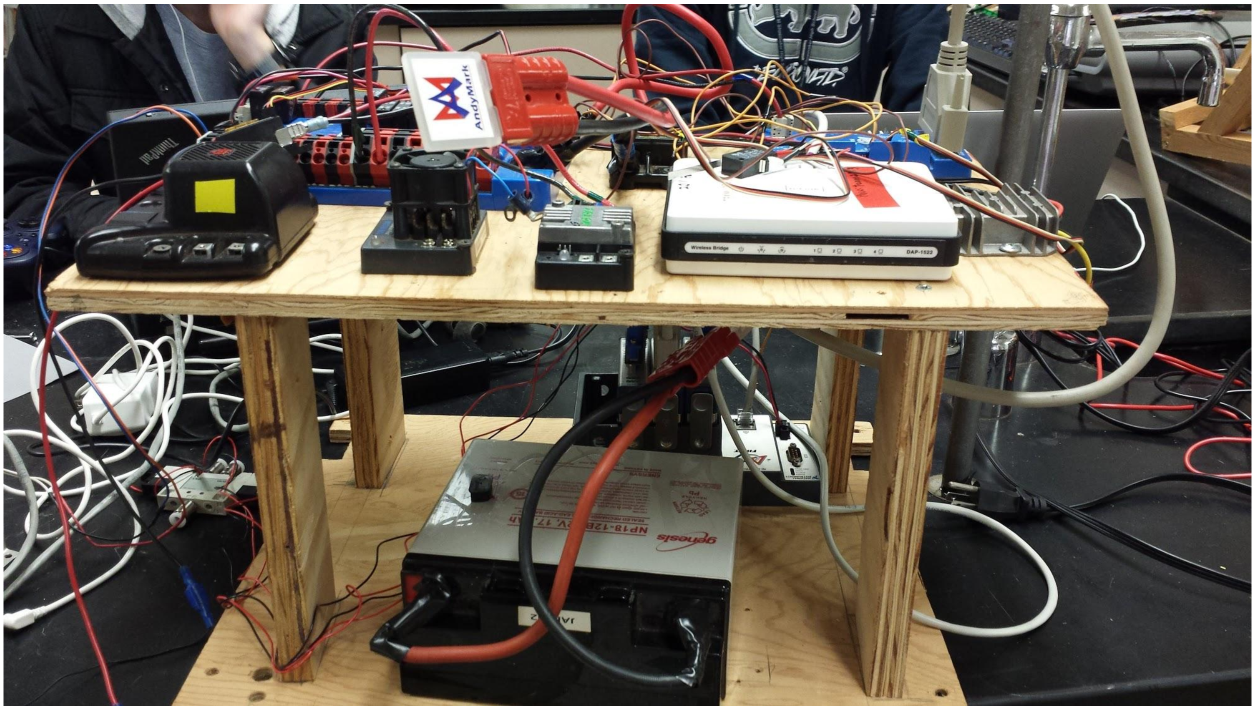
Appendix A: General Wiring Diagram (Testbed)

A very basic electrical layout that can be used for testing purposes. It consists of the core electrical components (battery, cRIO, digital sidecar, D-Link, power adapter, and PDB) and extensions of the control system (motor controller, spike, camera, and ringlight).



It is recommended to prepare a testbed with a victor, jaguar ***or*** *talon, and spike so you will be able to test a wide variety of components.*

In real life, wires are ugly. Manage well, may the conduit be with you. Remember: right angles and zipties are your friend.



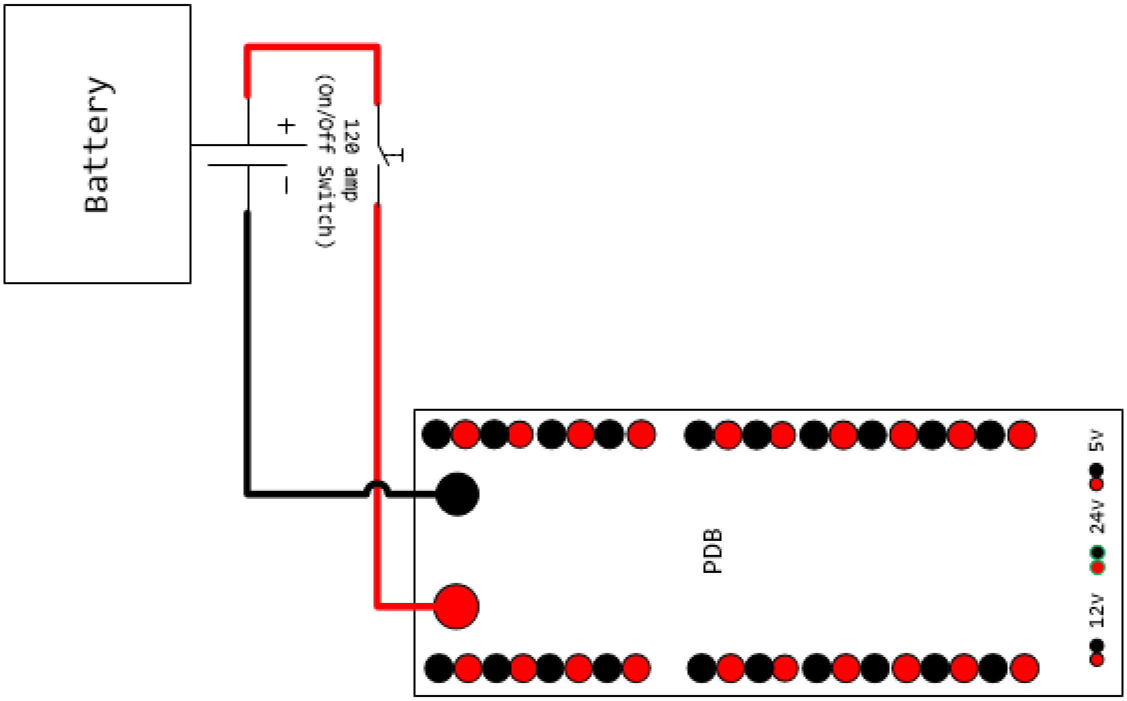
Top Layer

- 1 PDB
- 1 On Switch
- 1 Digital Sidecar
- 1 D-Link
- 1 Power Convertor
- Motor Controllers
    - 1 Victor
    - 1 Talon
    - 1 Jaguar

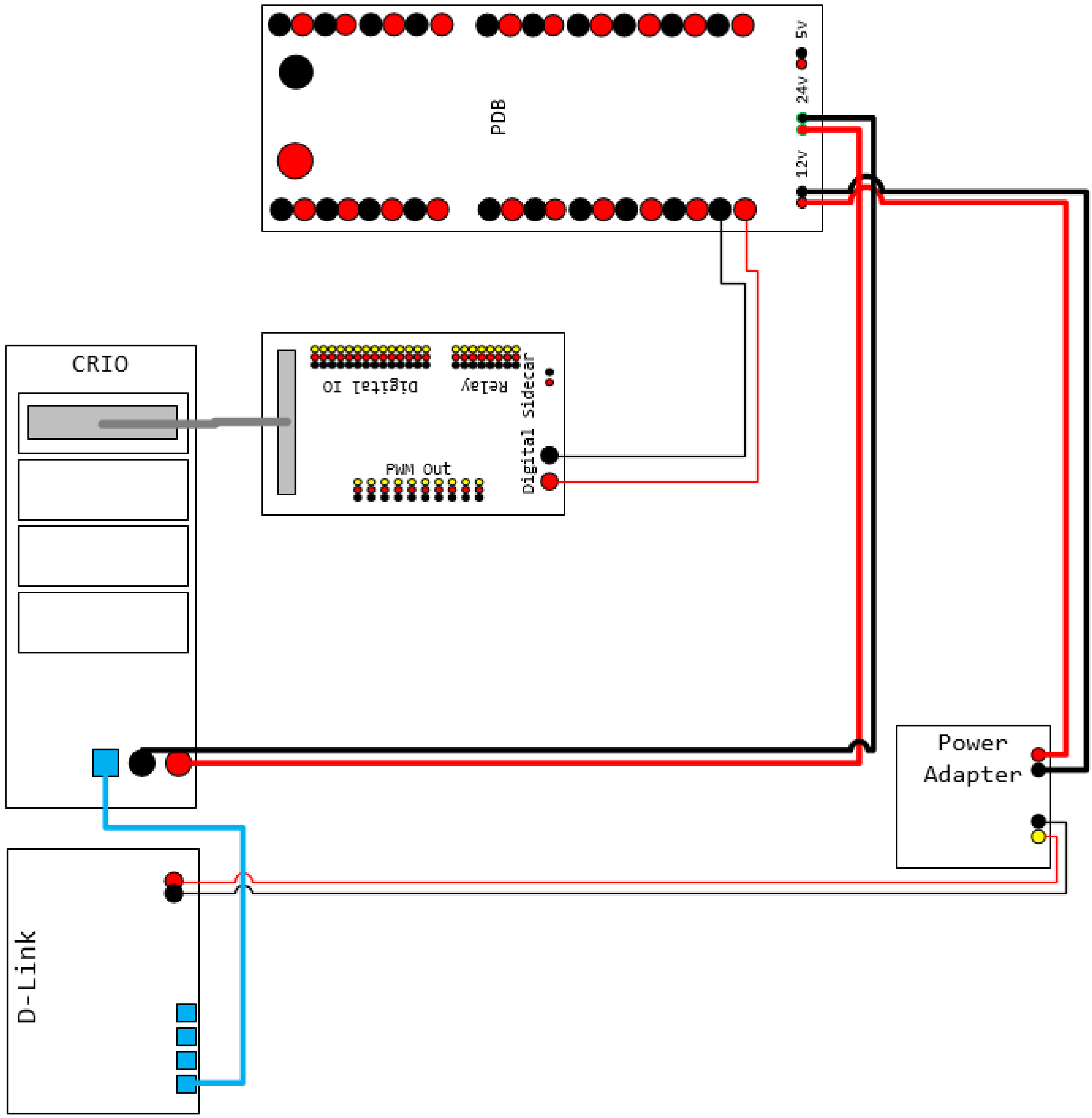
Bottom Layer

- cRio (usually w/ 4 slots)
    - cRIO modules 9403, 9201, and 9472
- Battery

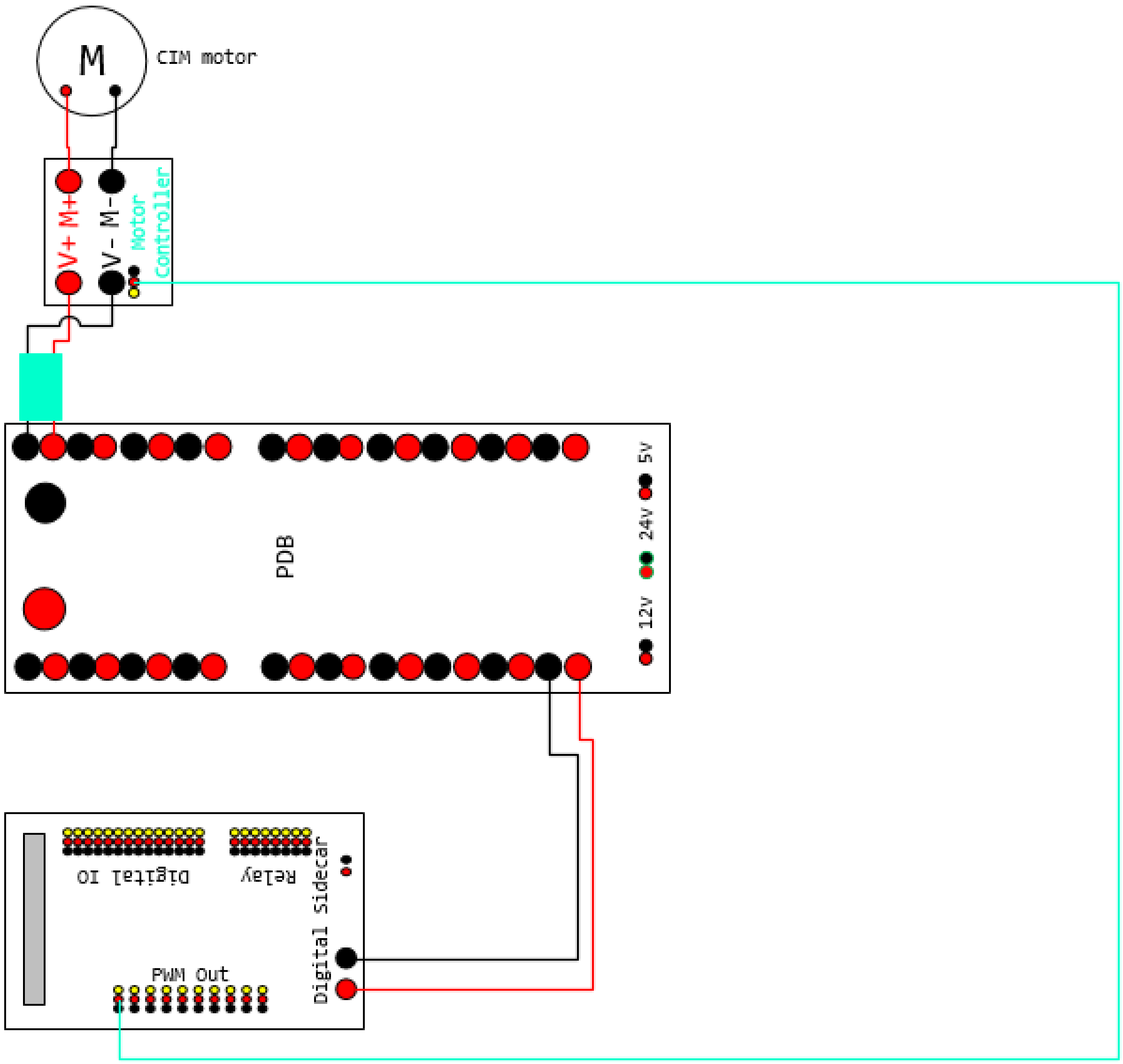
Wiring Connections



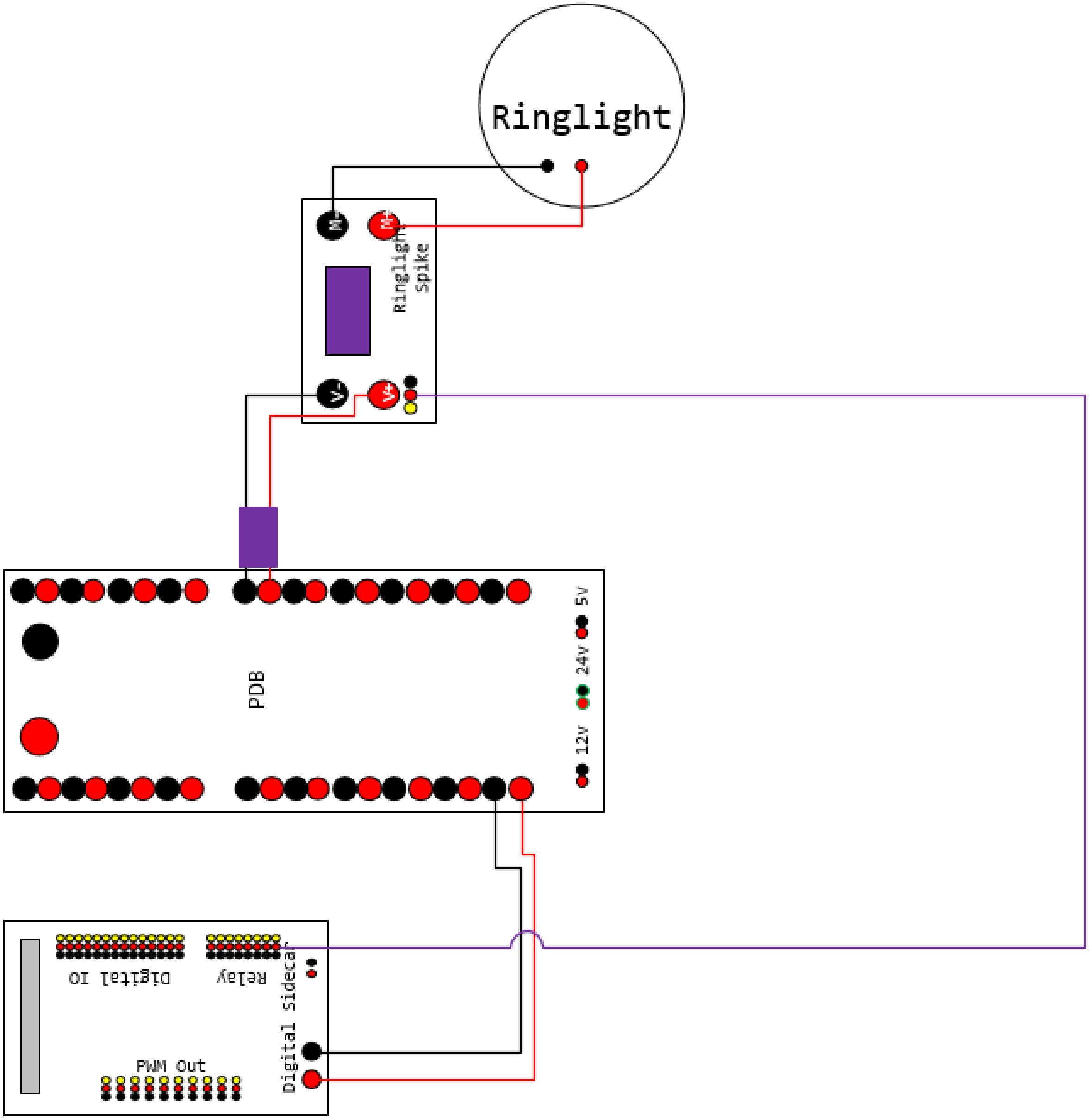
The battery is connected to the Power Distribution Board; the battery supplies the power to the power distribution board. The circuit is controlled by a 120 amp circuit breaker on the positive wire, acting as an on/off switch, breaking the power supply when not connected; and completing the circuit when on.



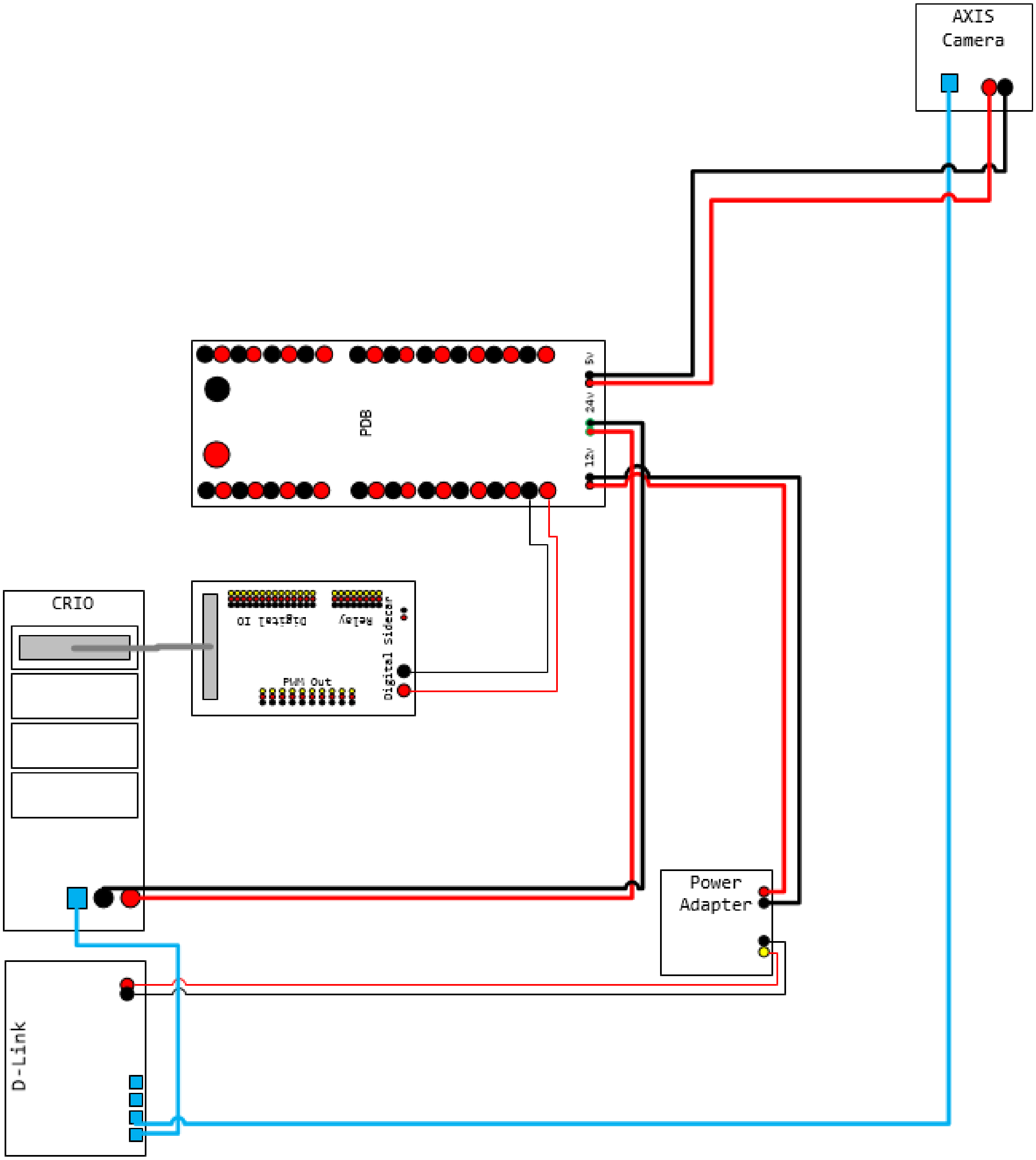
The Power Distribution Board powers the all-important cRIO from its special 24 V port; the ethernet port on the cRIO connects to the d-link router with a straight-through ethernet cable; which is powered by the Power Adapter (***part of the wire that is from the adapter to the D-link is yellow***) that reduces the power from the special 12 V port on the PDB down to a safe 5 V for the D-Link.(If it isn't hooked up properly with the half yellow wire to the D-Link, you risk a dead D-Link, never to live again) The PDB also powers the Digital Sidecar; which is connected to a cRIO module by a ribbon cable that is slotted into the cRIO.



The PDB directs power to a motor controller which is hooked up to the Digital Sidecar through a PWM from the motor controller to the PWM Out on the Digital Sidecar. The PWM should match the port # in the program; while the cRIO is not depicted, the cRIO sends the commands through the ribbon cable to the Digital Sidecar to the port # in the program whether the motor is plugged in there or not. Therefore if it is in a different port than the program, it probably won't do what you want; however that is a communication issue. The motor controller is wired to the motor, in this case a CIM motor(To know what controllers can hook up to which motors refer to Motor Controllers), it controls the amount of current that goes to the motor doing its job as a motor controller; main idea it controls speed/power at which motor turns, and relays the data through the PWM.



The PDB powers the Spike relay in a 30A (use a 20A circuit breaker) slot which is connected to the ring light and powers it when it turns on. The Spike gets the signal to turn on from the PWM that connects it to the Digital Sidecar which receives the commands from the cRIO through the ribbon cable. The ring light is used in conjunction with either a light sensor or camera.



The PDB powers the axis camera through its special 5V port and the axis camera plugs into the D-Link through an ethernet cable. It receives commands from the cRIO through the ethernet cable from the cRIO to the D-Link to the axis camera; and then the axis camera sends data through the cable to the D-Link through the other ethernet cable to the cRIO.