FRC C++ Programming

Last Updated: 01-24-2017

Table of Contents

Setting up the Development Environment

Installing Eclipse (C++/Java)

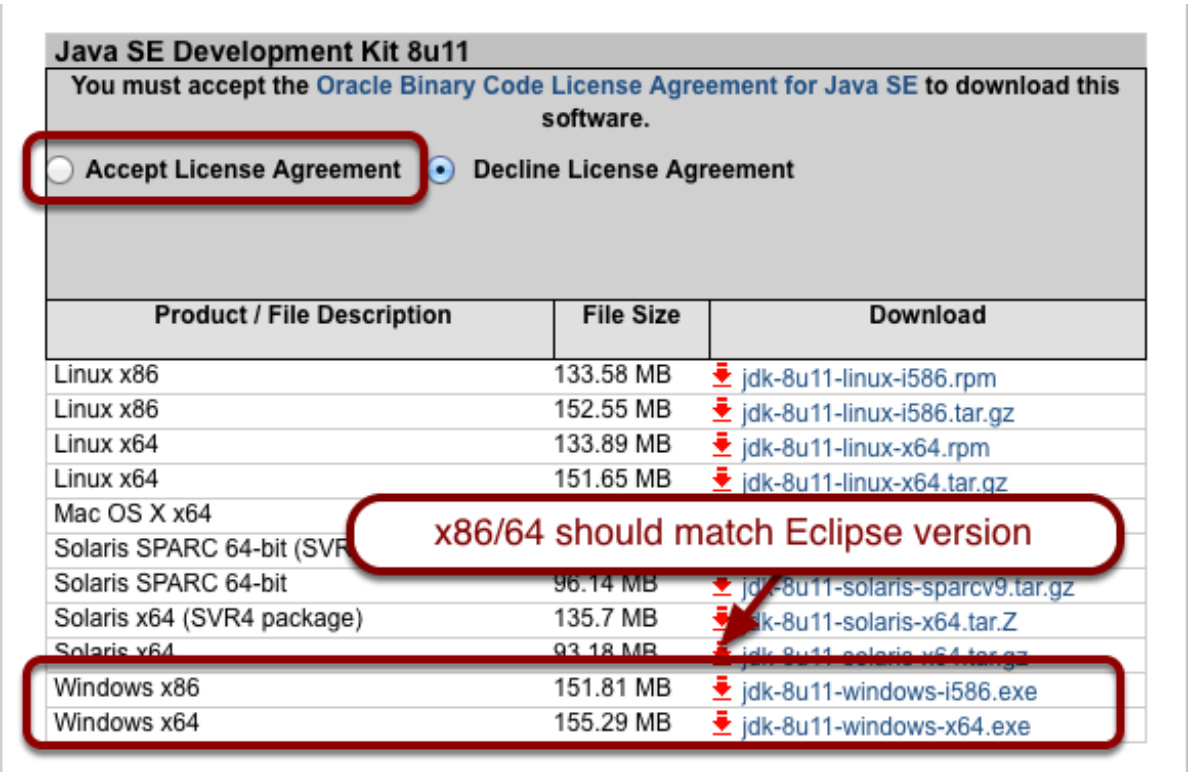
The 2017 suite of text-based languages, Java and C++, utilize the current version of Eclipse as a development environment. The FRC specific tools for the chosen language are installed as Eclipse plugins. You can install both the Java and C++ development tools into the same installation of Eclipse to allow programs to be written with either language using a common set of tools and user interface.

The 2017 Eclipse plugins have been tested with Eclipse Luna, Eclipse Mars and Eclipse Neon. Teams with existing installs from 2016 can update their installations to 2017 by following the updating the plugins when prompted by opening Eclipse (if automatic update is enabled) or following the Updating the plugins manually instructions below. C++ teams should also install the new toolchains (Installing the C++ Toolchains).

CAN Talon SRX has been removed from WPILib. See this blog for more info and find the CTRE Toolsuite installer here: http://www.ctr-electronics.com/control-system/hro.html#product_tabs_technical_resources

Note: The C++ and Java tools and environment are available for Windows, Mac OSX and Linux, though the Windows version is the one that has been the most heavily tested. You should be able to use any of the three for your development platform, however you should keep in mind that you will need a Windows computer to run the Driver Station software and roboRIO Imaging tool.

Getting Java



To use Eclipse you must have Java JDK installed on your system. You can get Java from the web site: http://www.oracle.com/technetwork/java/javase/downloads/index.html. Select "JDK Download", then scroll down the page to "Java SE Development Kit". Accept the license agreement and download the Java SDK for your platform. The version (either x86 or x64) should match the version of Eclipse that you have installed or plan to install on your computer. This has been tested with Java SE 8u11 but will probably work with later versions as well.

Java 8 is installed on the RoboRIO and to take advantage of all the features it offers, it is suggested that you use Java 8 on your development system. You may use an earlier version, however it should be noted that the Java-installer program for loading the JRE on the roboRIO requires Java 8 (so you will need at least one computer with it) as does the rioLog Eclipse plugin which is used to view console output.

Note: Java is required to be installed even if you are doing C++ development since Eclipse, the development environment, is a Java program. Also, the Oracle web page might change over time, so the images shown here might not exactly match what you see.

Installing the C++ Toolchains (C++ teams only)

Download the appropriate C++ Toolchains installer for your platform from http://first.wpi.edu/FRC/roborio/toolchains/

Note: Remaining Toolchain installation instructions describe Windows installation, instructions for the other two platforms are listed on the download pages.

Note: The Windows toolchains will always install to the root fo the system drive.

Windows: Double click on the downloaded file to launch it. If you receive a Security Warning, click Run. Check the box to accept the License Agreement, then click **Install.** When the install completes, click **Finish.**

Mac OSX: Double-click on the downloaded file in Finder to unzip it. In Finder, right-click on the "FRC ARM Toolchain.pgk" file, then press the option key on your keyboard, and click "Open". Follow the steps to install the package.

Linux: See the instructions in the text file on the toolchains page.

Getting Eclipse

You can get eclipse from the web site: http://www.eclipse.org then press the "Download" button to select the version to be used.

Download Eclipse

Select the version of eclipse that matches your desired programming language (there are instructions below for adding C++ to Java or Java to C++). You should choose the version of eclipse that matches your operating system and version of Java from above.

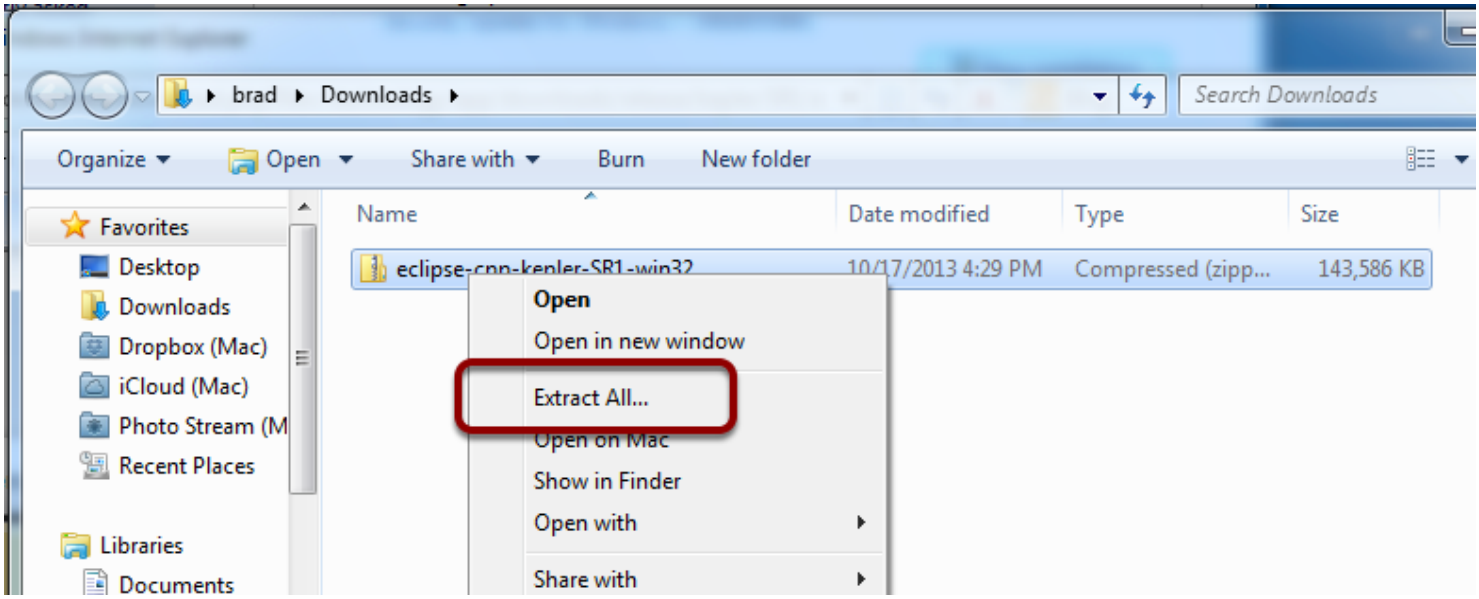
At the time of this writing the current version is Neon (4.6) and that is what we've been using for development of the tools. On the next screen choose a download site and start the download. Choose a location such as the downloads folder for the zip file.

Note: on 64 bit Linux systems it might be necessary to install 32 bit version of libc. For example, on Ubuntu Linux, the command would be:

```
sudo apt-get install libc6-i386
```

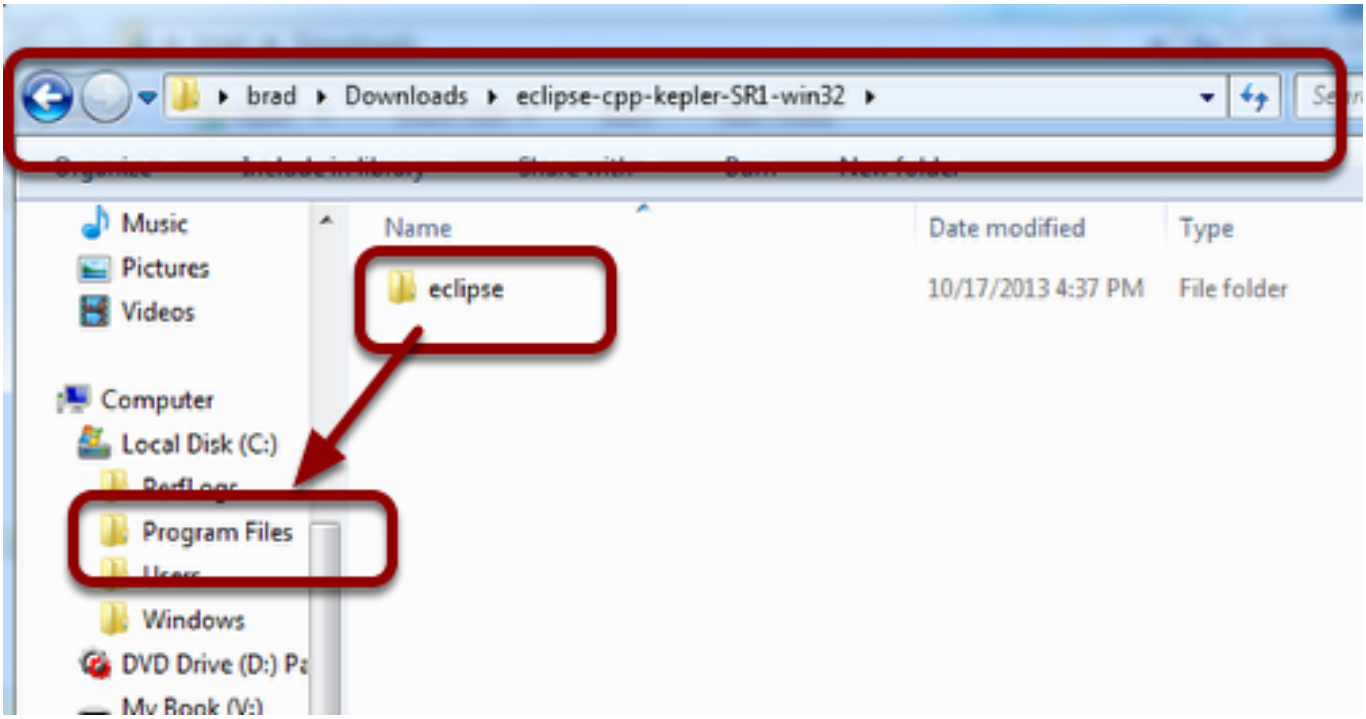
This is necessary to run the gcc binaries as part of the plugins since they are compiled for 32 bit linux.

Unpack the eclipse folder and move it to Program Files



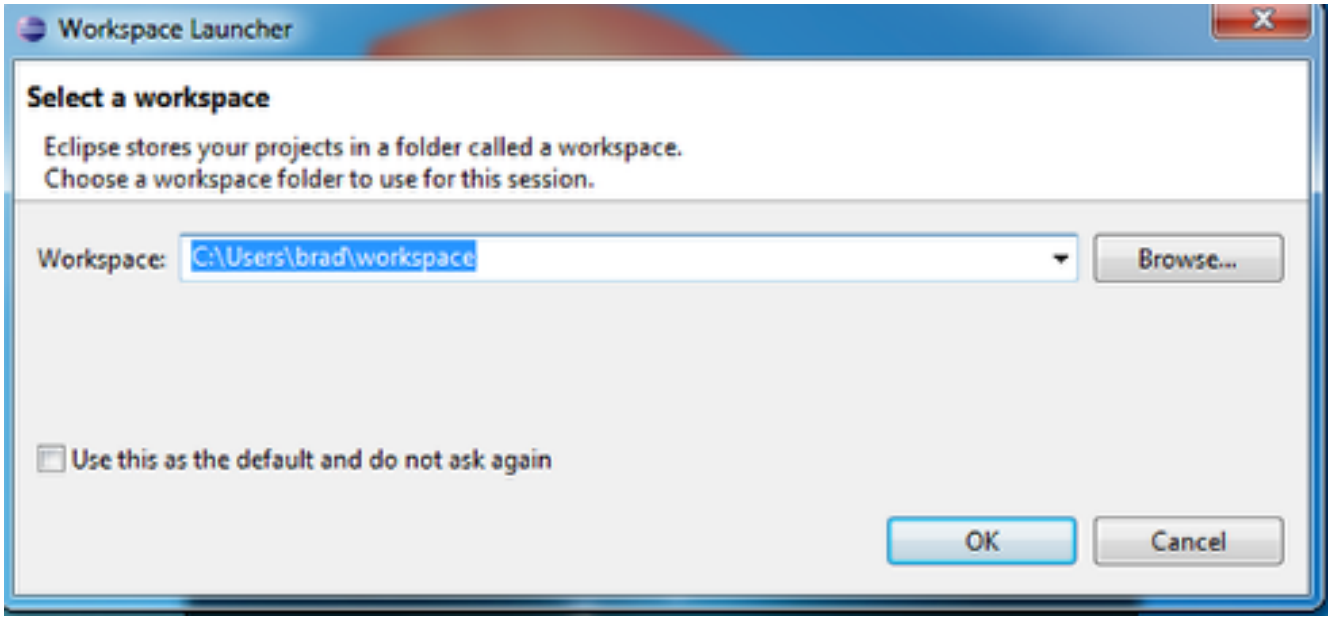
Extract the contents of the zip file by right-clicking on the .zip file in a windows explorer window and selecting "Extract All..." and taking the default for the location to extract it.

Move the extracted eclipse folder to Program Files



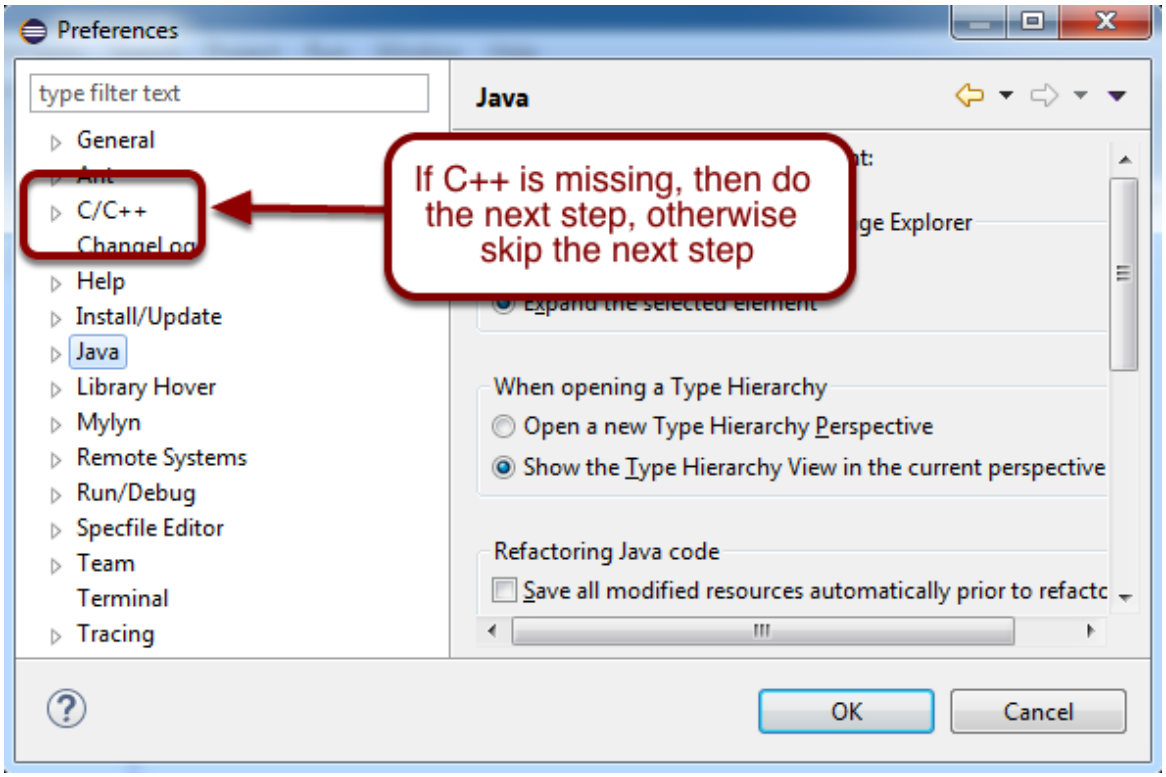
Move the extracted folder to Program Files or some other convenient location from which to easily run it. Within the eclipse folder you'll see the file "eclipse.exe". You can right-click on "eclipse.exe" and select "Pin to start menu" to make it easier to run eclipse without having to find the installation location.

Starting Eclipse for the first time



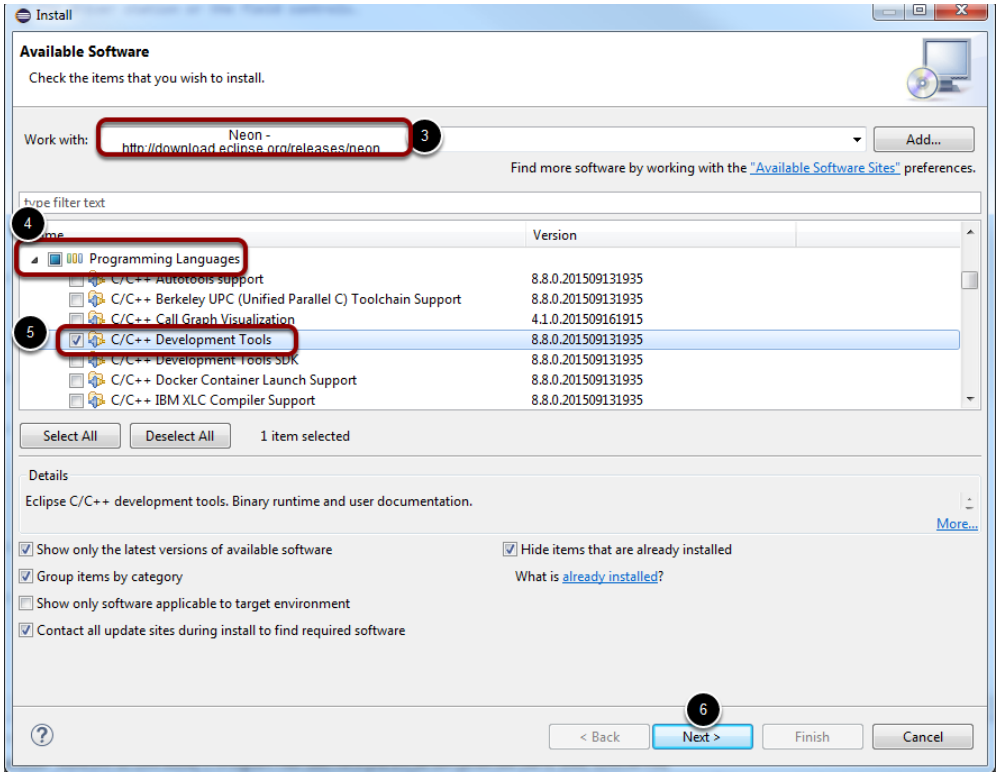
Start Eclipse (it will be on your start menu if you chose to pin it from the previous step.) The first time Eclipse starts it will ask you for the location of your workspace. A workspace is the location on disk where projects and files are stored by default. You can have more than one workspace, but it's suggested to take the default location until you have more experience with Eclipse.

Adding C++ to Java Eclipse



To program C++ with the Java version of Eclipse, you will to have the C++ Development Tools (CDT) installed. To determine if they're already installed, select Window, then Preferences from the menu bar. Then look for **C/C++** on the left side of the Preferences window. If it is missing then you must install it. The installation procedure is in the next step.

Install Eclipse C++ Development Tools

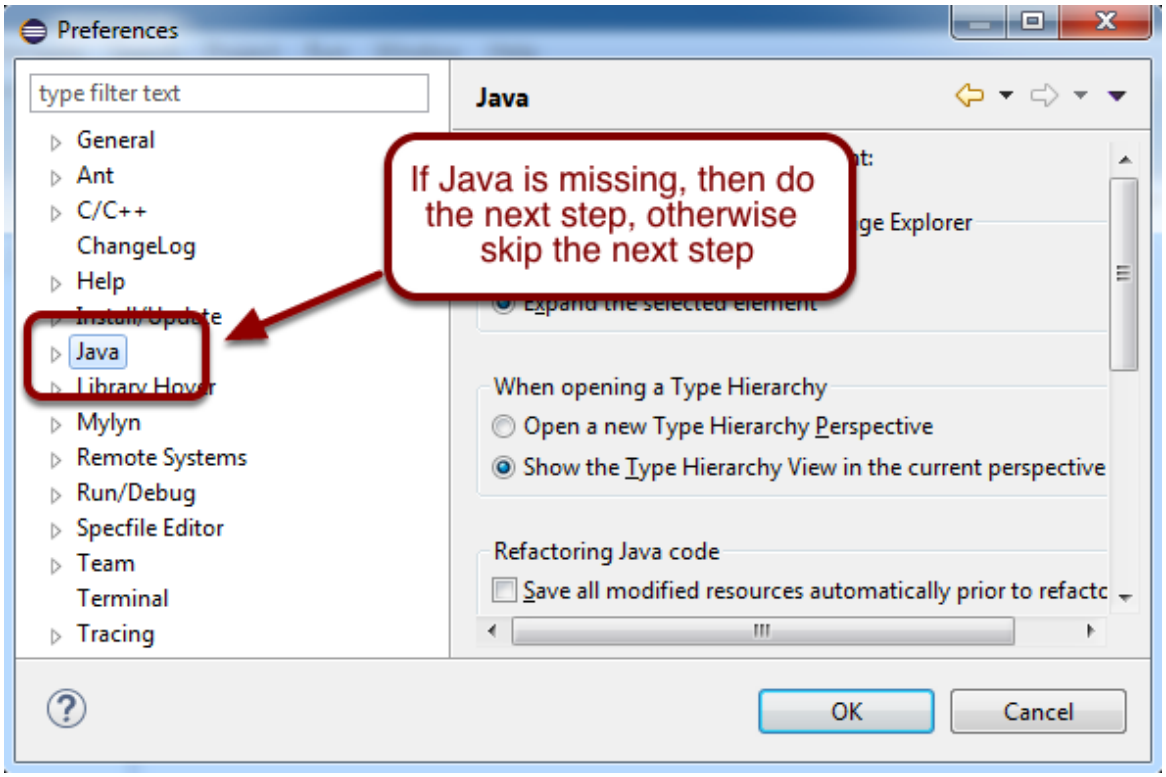


If C/C++ is missing from the preferences window (see previous step), then it must be installed

1. Close the Preferences window if it's open.
2. Select Help, then Install New Software... from the menu bar.
3. Click the dropdown and select the "Neon" site as shown.
4. Scroll down to the Programming Languages section and click the arrow to expand.
5. Click the checkbox to choose Eclipse C++ Development Tools
6. Click Next.
7. Take the defaults for the other options and let Eclipse restart.

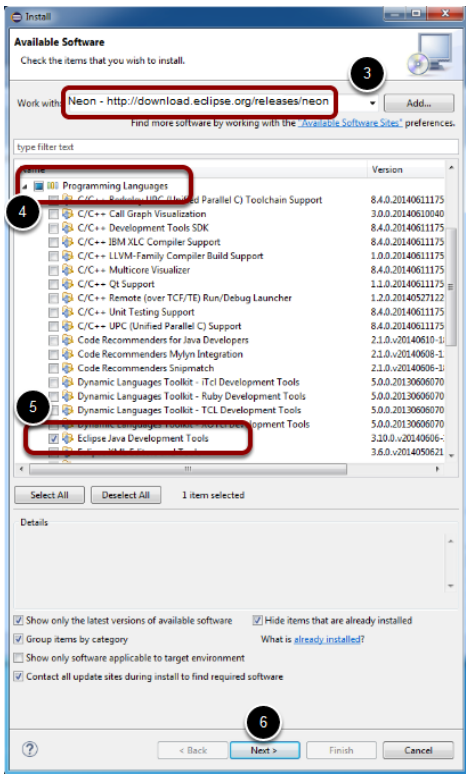
When these steps are finished, and Eclipse has restarted, C++ should be an available option on the Preferences window, and all the C++ perspectives will be available.

Adding Java to C++ Eclipse



To program Java with the C++ version of Eclipse, you will need to have the Java Development Tools (JDT) installed. To determine if they're already installed, select Window, then Preferences from the menu bar. Then look for Java on the left side of the Preferences window. If it is missing then you must install it. The installation procedure is in the next step. If you do have the Java development tools installed (Java is shown), then skip the next step and continue configuring to Setting up the JDK in Eclipse.

Install Eclipse Java Development Tools

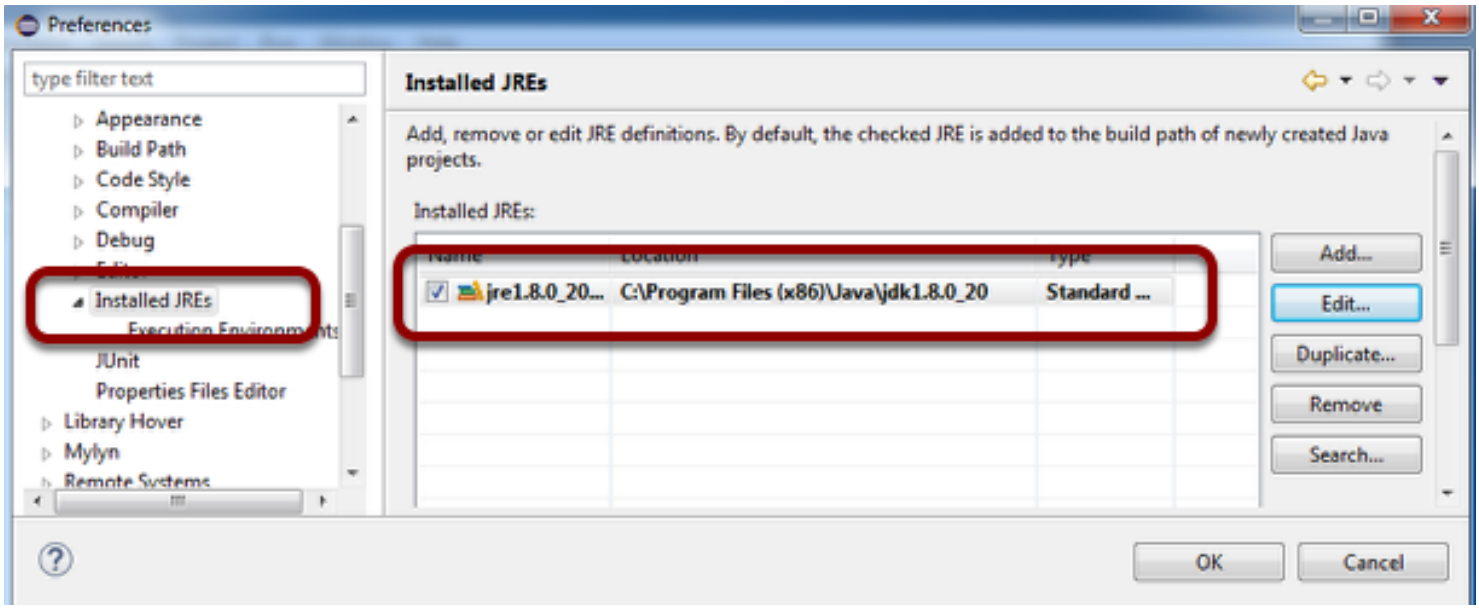


If Java is missing from the preferences window (see previous step), then it must be installed

1. Close the Preferences window if it's open.
2. Select Help, then Install New Software... from the menu bar.
3. Click the dropdown and select the "Neon" site as shown.
4. Scroll down to the Programming Languages section and click the arrow to expand.
5. Choose Eclipse Java Development Tools
6. Click Next.
7. Take the defaults for the other options and let Eclipse restart.

When these steps are finished, and Eclipse has restarted, Java should be an available option on the Preferences window, and all the Java perspectives will be available.

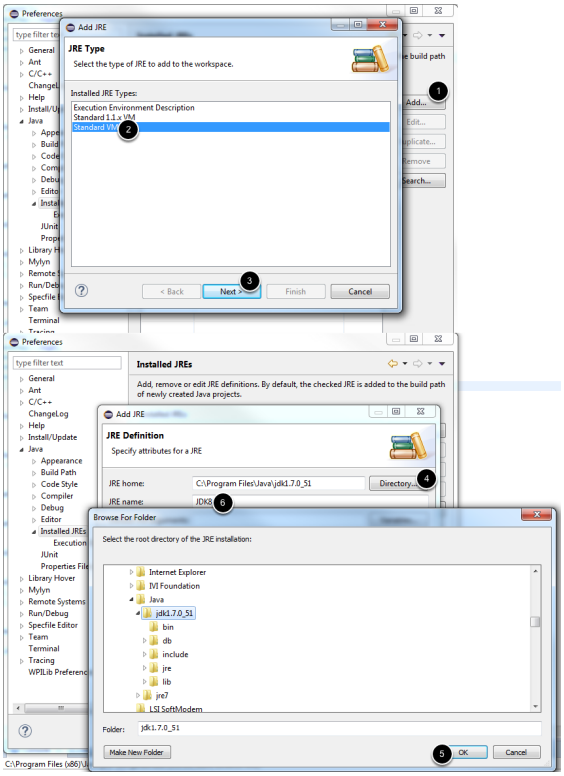
Setting up the JDK in eclipse (Java teams only)



Select Windows from the menu bar, then Preferences. Choose Java preferences in the list on the left of the Preferences window, then Installed JREs. Be sure that the installed JDK is selected as shown ¬†(make sure the "Location" field includes jdk 8 or 1.8, the name field may be the same in either location). This will enable eclipse to build Java programs for the RoboRIO. Without this setting you will see error messages about the JRE path not being set correctly.

If you do not see any option with the appropriate location, see the next step "Adding the JDK". If you do have the appropriate option, skip the next step.

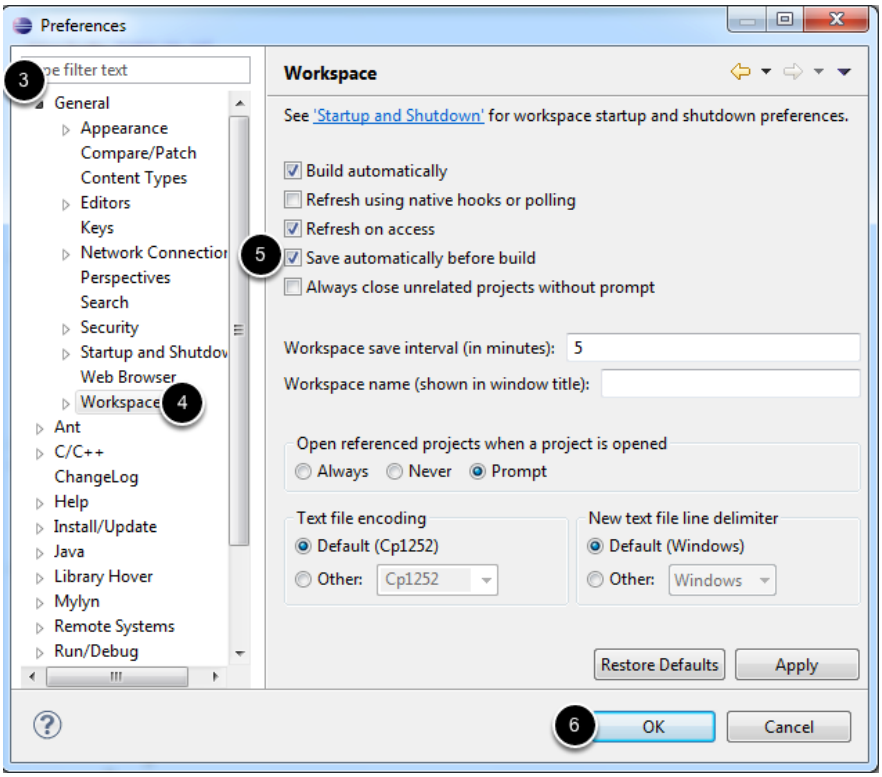
Adding the JDK



Only iIf the JDK is not shown in the step above:

1. Click **Add**
2. Select **Standard VM**
3. Click **Next**
4. Click **Directory** and browse to the folder for the JDK (usually C:\Program Files\Java* or C:\Program Files (x86)\Java*). The image shows jdk1.7.0_51, you will likely have a jdk1.8.* version.
5. Click **OK**. Pick a name for the JRE such as JDK8.
6. Click **Finish**
7. Make sure the box for the newly added JDK entry is checked.

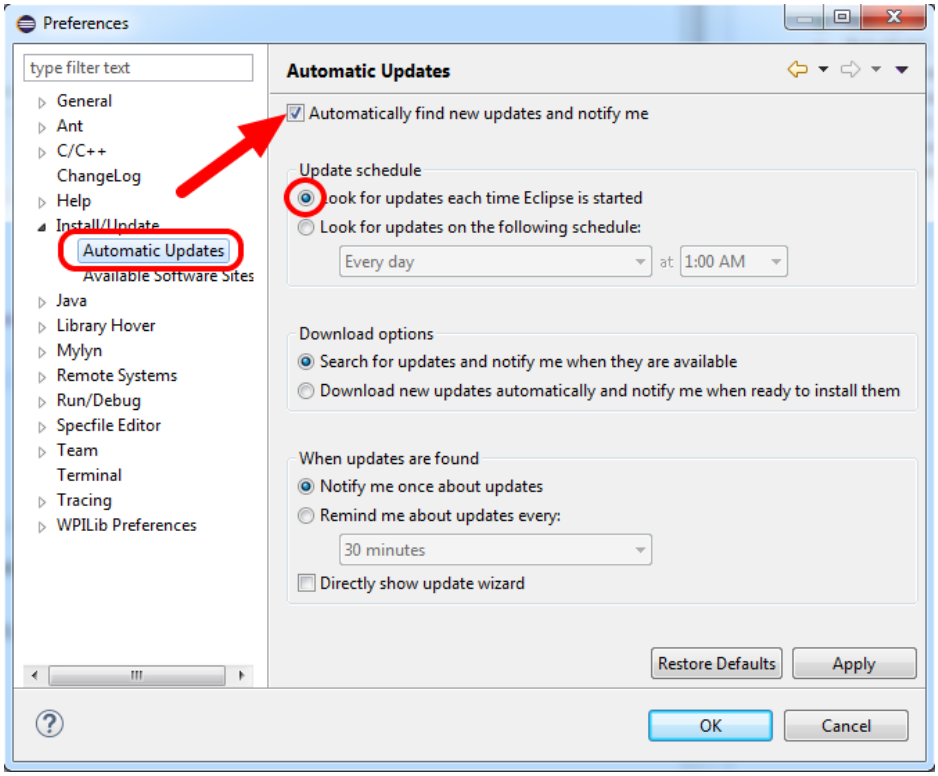
Configuring eclipse



There are a huge number of configuration options for eclipse to set up the environment for your preferences. One suggested setting to note is: "Save automatically before build." This setting will cause all of your workspace changes to be saved when you build the project. If you don't set this, remember to save changes before building, otherwise the rebuilt program won't reflect your newest updates.

To set this, go to **Window** -> **Preferences** -> **General** -> **Workspace** -> Check **Save automatically before build** -> **OK**

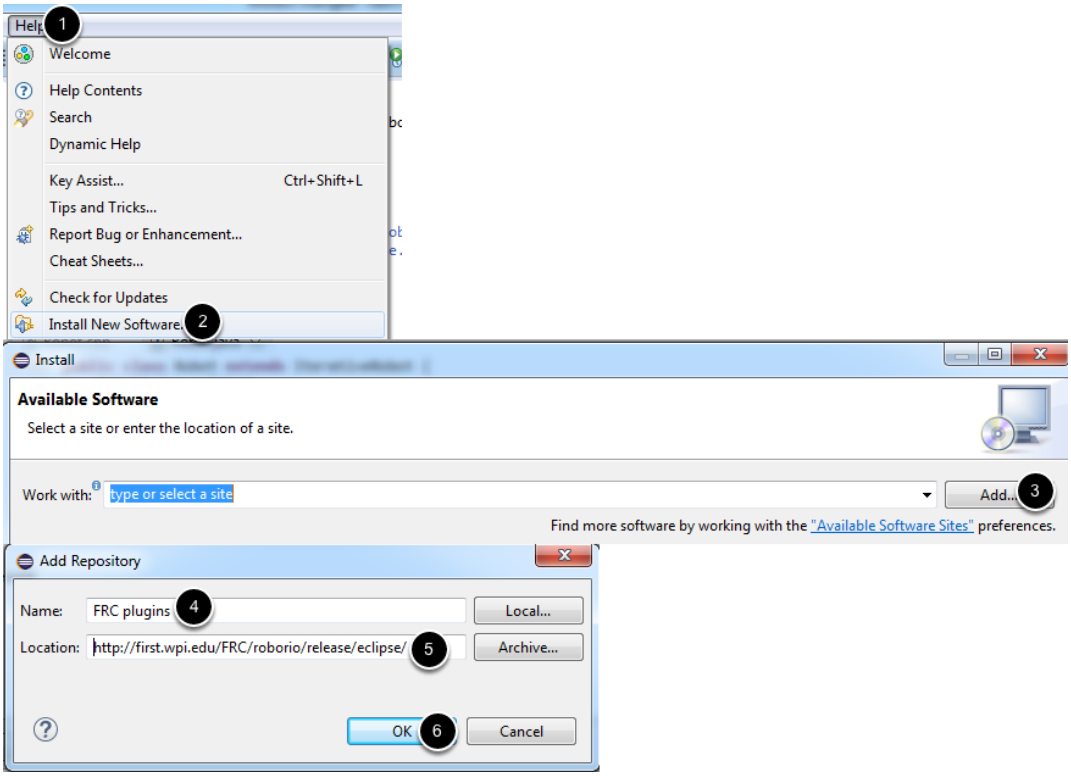
Automatic Updates



Another recommended setting is enabling Automatic Updates. With Automatic Updates enabled, Eclipse will check for updated versions of the plugins each time it starts and inform you if an update is available. This will help insure you are notified of new versions of the WPILib plugins.

To enable Automatic Updates, select **Install/Update** then **Automatic Updates**. Check the box at the top to **Automatically find new updates and notify me**. Select the radio button to **Look for updates each time Eclipse is started.** Then click **OK.**

Installing the development plugins - Option 1: Online Install



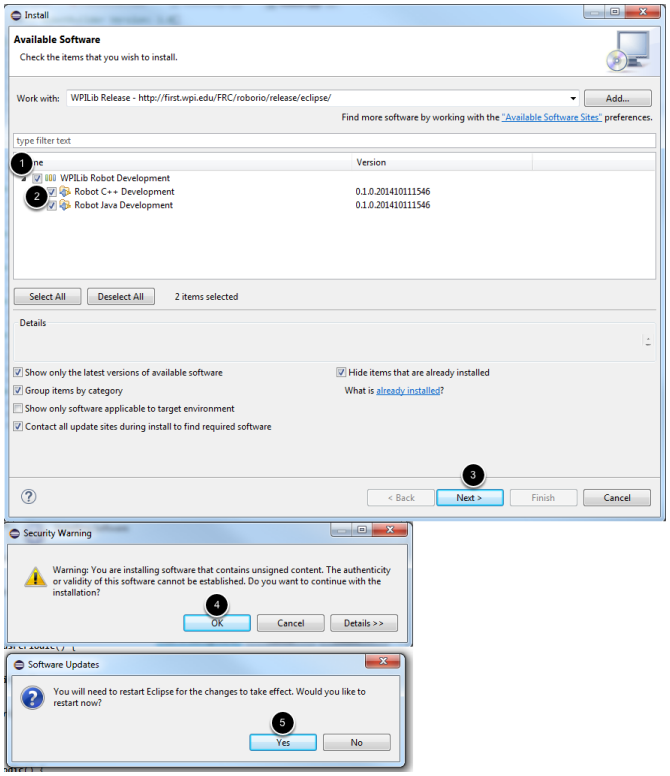
It is recommended to install the plugins using this method, which requires an active internet connection and fetches the plugins directly from the WPILib site. This will allow you to check for updates to the plugins using Eclipse.

Eclipse extensions are based on user-installed plugins. To get the WPILib development tools you will need to install the correct plugin for your language.

When Eclipse starts:

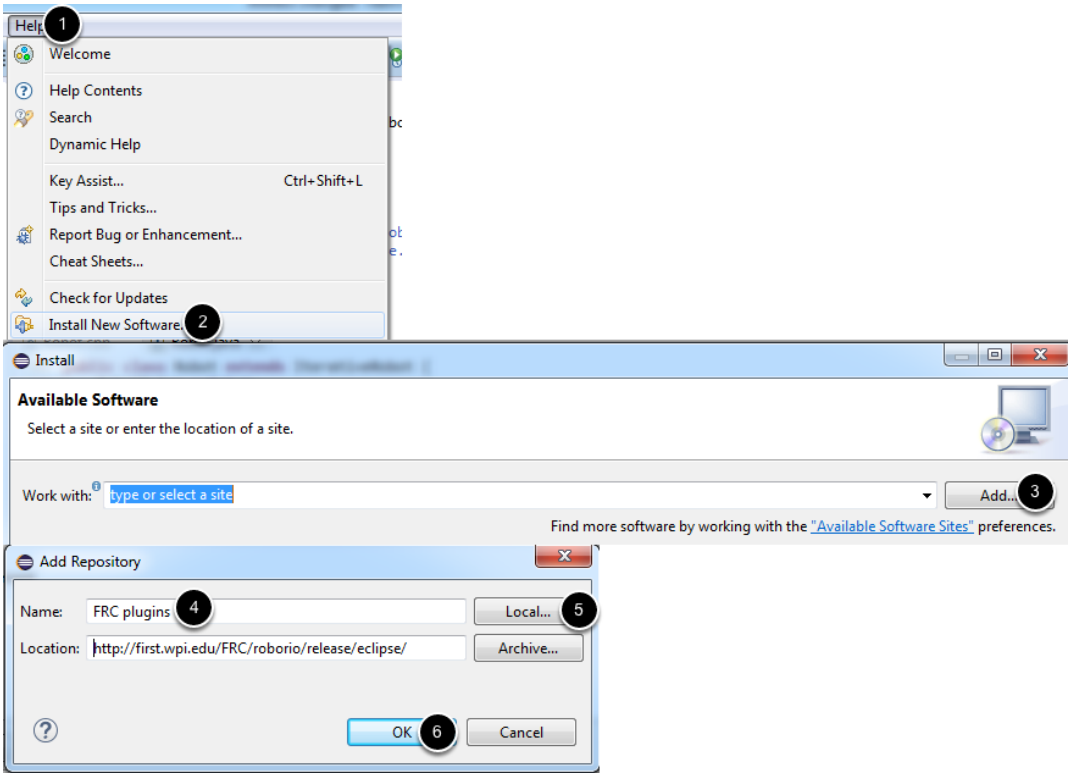
1. Select "Help"
2. Click "Install new software".
3. From here you need to add a software update site, the location where the plugins will be downloaded. Push the "Add..." button then fill in the "Add Repository" dialog with:
4. Name: FRC Plugins
5. Location: http://first.wpi.edu/FRC/roborio/release/eclipse/
6. Click "OK".

Selecting the correct plugins



1. Click the arrows if necessary to expand the WPILib Robot Development menu.
2. Select the WPILib Robot Development plugin for your desired language (you can install both if you wish to try programming in both languages)
3. Click **Next**, **Next** on the next page, then click the radio button to accept the license agreement and click **Finish**
4. If you receive a Security Warning prompt, click OK to continue.
5. When prompted, restart Eclipse. After Eclipse restarts and you select your Workspace (if prompted) you will see a dialog that says Installing Java. This details the installation progress of the plugins, wait for the install to complete before proceeding. This dialog should only appear when the plugins are first installed or updated.

Installing the development plugins - Option 2: Download and install offline

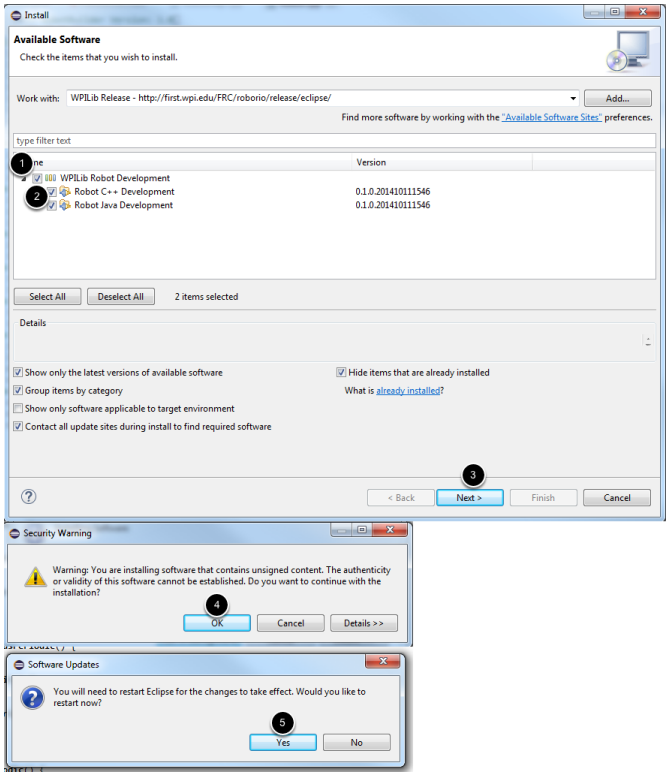


If you need to download the plugins and install them on a different machine offline, you will be unable to check for updates using Eclipse. Download the zipfile containing the plugins from¬†http://first.wpi.edu/FRC/roborio/release/ Right click on the zip and select Extract All to extract the files.

When Eclipse starts:

1. Select "Help"
2. Click "Install new software".
3. From here you need to add the downloaded plugin location. Push the "Add..." button then fill in the "Add Repository" dialog with:
4. Name: FRC Plugins Offline
5. Click Local
6. Browse to the "site" directory inside the directory you extracted the zip file to.
7. Click "OK".

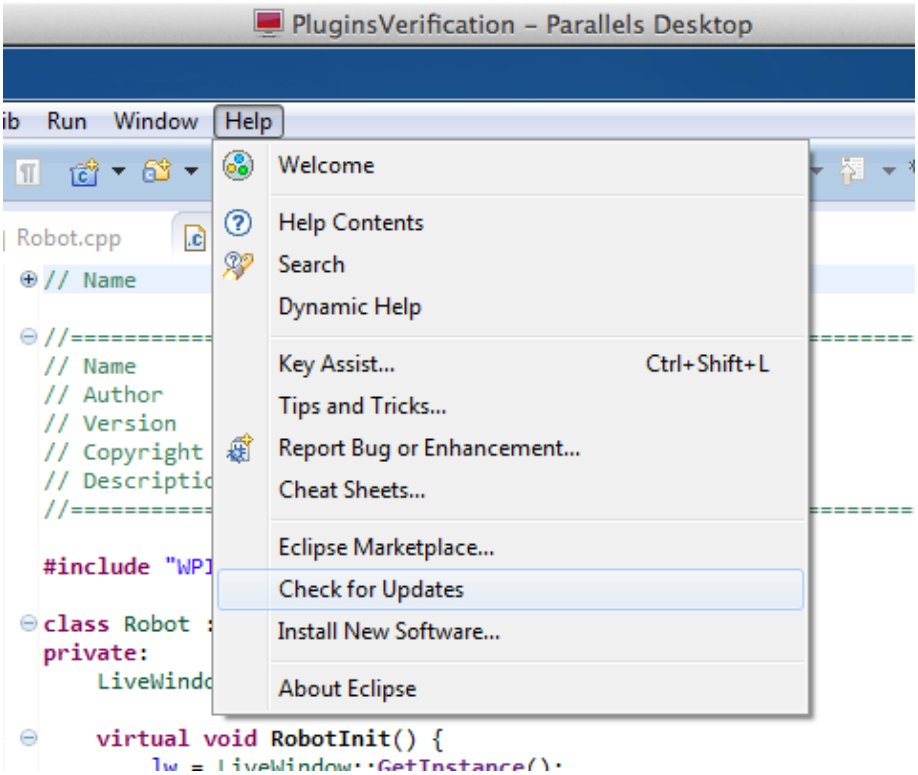
Selecting the correct plugins



1. Click the arrows if necessary to expand the WPILib Robot Development menu.
2. Select the WPILib Robot Development plugin for your desired language (you can install both if you wish to try programming in both languages)
3. Click **Next**, **Next** on the next page, then click the radio button to accept the license agreement and click **Finish**
4. If you receive a Security Warning prompt, click OK to continue.
5. When prompted, restart Eclipse. After Eclipse restarts and you select your Workspace (if prompted) you will see a dialog that says Installing Java. This details the installation progress of the plugins, wait for the install to complete before proceeding. This dialog should only appear when the plugins are first installed or updated.

If updated plugins are released, you can either repeat this process (you will get one additional Eclipse window telling you that the components are already installed and an upgrade will be performed instead of an install), or if online installation is an option, you can complete the online installation steps above, then get future updates using the Eclipse Automatic Updates (or the manual update check described below)

Updating the plugins manually



Note: This only works if the plugins were installed using Option 1 - Online Install from above. For updating plugins when the offline install was used, see the note at the end of the step above.

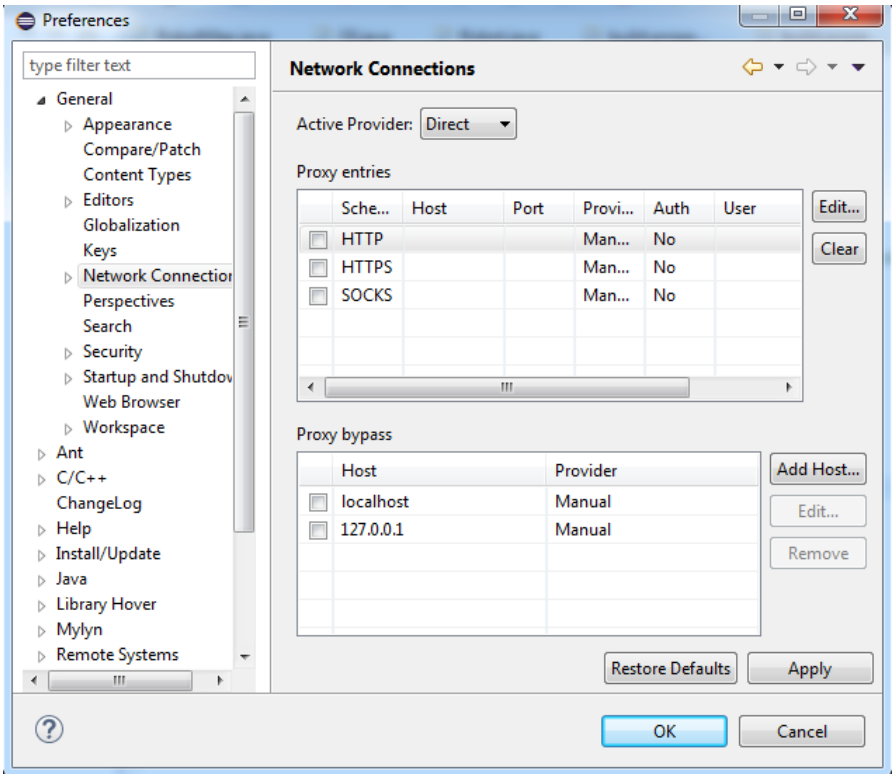
If you choose not to enable Automatic Updates as described above, you will need to manually have Eclipse check for updates to install new versions of the plugins.

Select **Help** from the menu bar, followed by **Check for updates**. Eclipse will check if there is a newer version available of any installed plugin and inform you if an update is found. Updating the plugins will ensure that your development system is up to date with the latest version of the development tools.

Troubleshooting

Below are some troubleshooting steps for commonly encountered issues

Unable to read repository at http://first.wpi.edu/FRC/roborio/release/eclipse/content.xml,Äù



This error occurs if Eclipse cannot contact the server to download the plugins. There are a couple possible causes of this issue:

1. Your computer is not connected to the Internet. Verify your network connection and try again.
2. Your firewall is blocking Eclipse. Try adding an exception for Eclipse or disabling your Firewall.
3. Your proxy settings were read improperly by Eclipse. In Eclipse Select **Window->Preferences->General->Network Connections**. If you don't use a proxy or don't know, set the Active Provider to **Direct.** If you use a proxy set the Active Provider to Manual and configure the proxy information by selecting the protocol and clicking Edit.

Need Java 1.7 or newer

If you get an error message when attempting to run Eclipse that says you "need Java 1.7 or newer", you have mismatched versions of Java and Eclipse installed. The easiest fix is to go back and download the other version of Eclipse (32 bit if you had 64, 64 if you had 32).

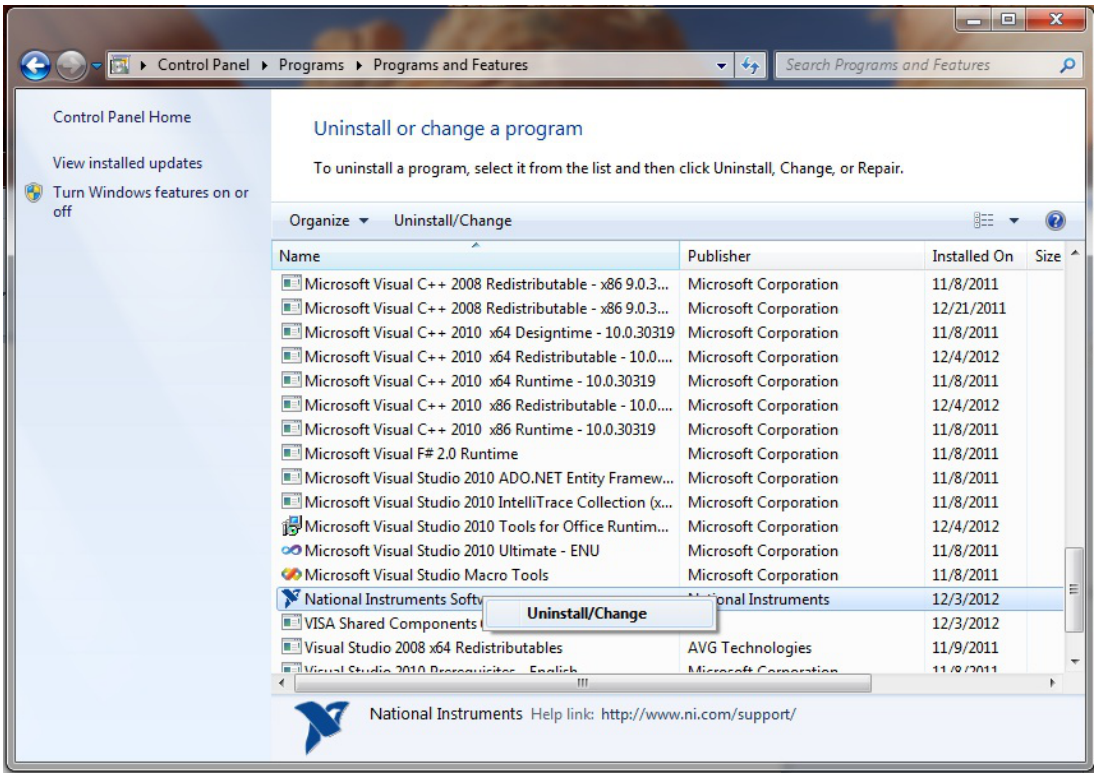
Installing the FRC 2017 Update Suite (All Languages)

The FRC 2017 Update Suite contains the following software components: LabVIEW Update, FRC Driver Station, and FRC Utilities. If an FRC 2017 LabVIEW installation is found, the LabVIEW Update will be installed or updated, otherwise this step will be skipped. The FRC Driver Station and FRC Utilities will always be installed or updated. The LabVIEW runtime components required for the driver station and utilities is included in this package. **No components from the LabVIEW DVD are required for running either the Driver Station or Utilities.**

Note: The 2017 DS will only work on Windows 7, Windows 8, Windows 8.1, and Windows 10. It will not work on Windows XP.

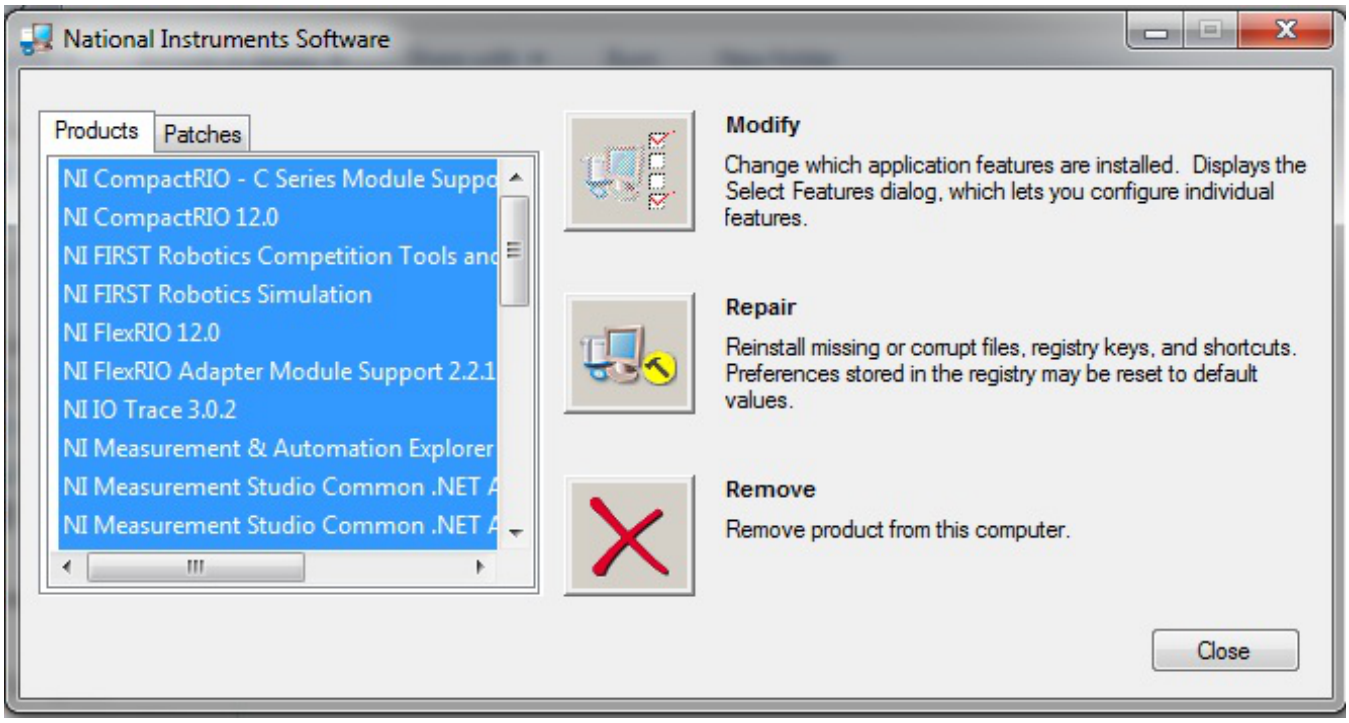
Note for LabVIEW Teams: CAN Talon SRX has been removed from WPILib. See this blog for more info and find the CTRE Toolsuite installer here: http://www.ctr-electronics.com/control-system/hro.html#product_tabs_technical_resources

Uninstall Old Versions (Recommended)



LabVIEW teams have already completed this step, do not repeat it. Before installing the new version of the NI Update it is recommended to remove any old versions. The new version will likely properly overwrite the old version, but all testing has been done with FRC 2015 only. Make sure to back up any team code located in the "User\LabVIEW Data" directory before un-installing. Then click **Start** >> **Control Panel** >> **Uninstall a Program.** Locate the entry labeled "**National Instruments Software",** right-click on it and select **Uninstall/Change.**

Select Components to Uninstall

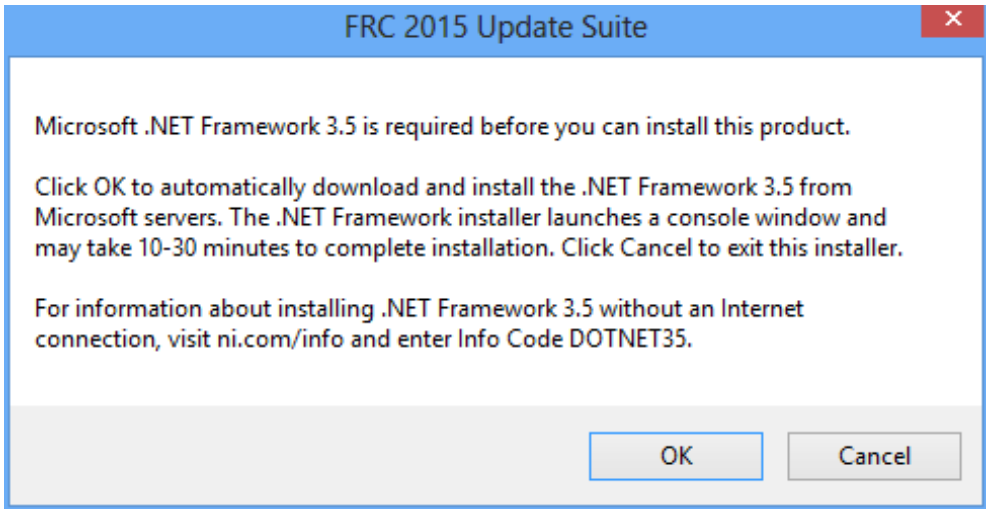


In the left pane of the dialog box that appears, **select all entries**. The easiest way to do this is to click the top entry to highlight it, then scroll down to the bottom entry, press and hold shift and click on the last entry then release shift. Click **Remove**. Wait for the uninstaller to complete and reboot if prompted.

Downloading the Update

Download the update from http://www.ni.com/download/first-robotics-software-2015/5112/en/

Windows 8

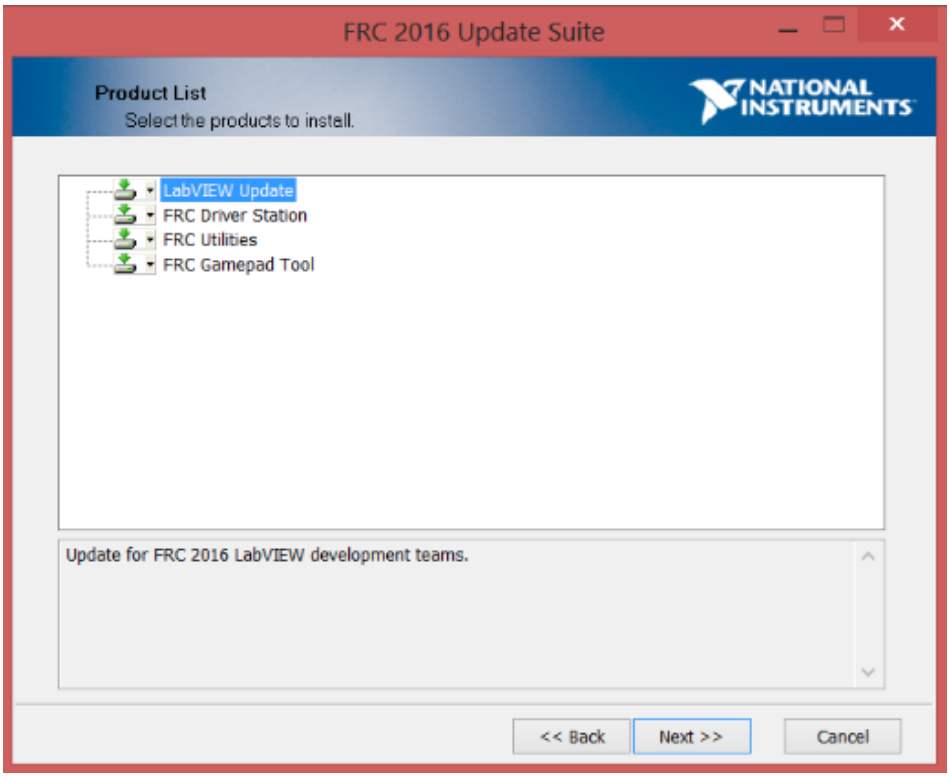


If installing on Windows 8 or 10 and the above error appears, jump down to the Addendum on Windows 8 installation before returning here to re-start the installation.

Welcome

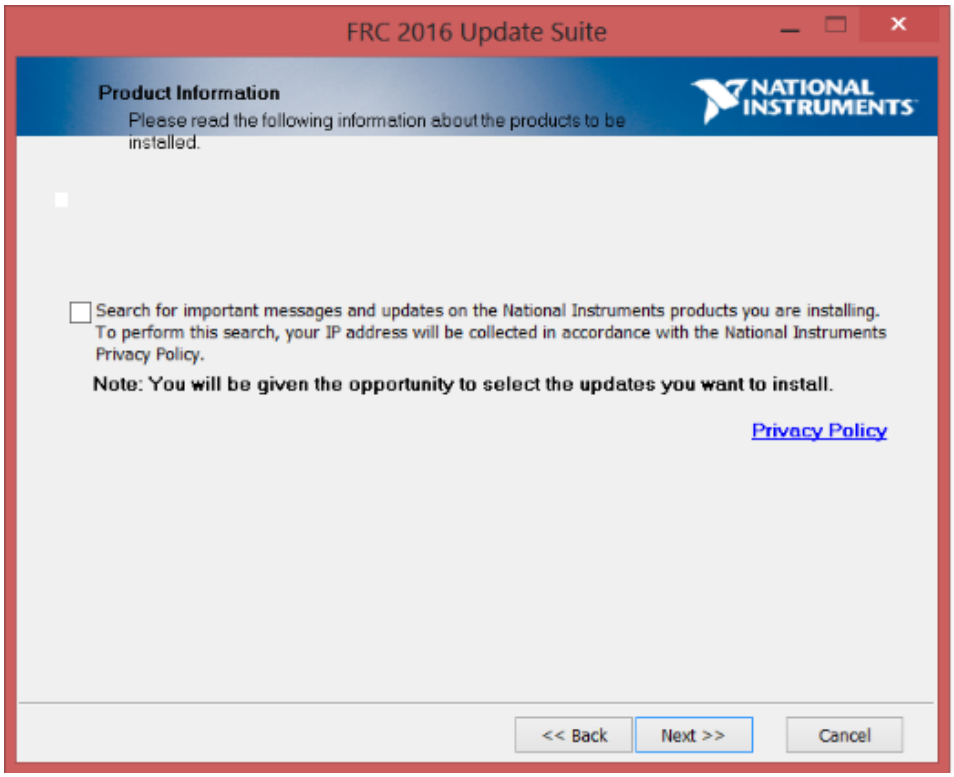
RIght click on the downloaded zip file and select Extract All. If you downloaded the encrypted zip file, you will be prompted for the encryption key which is **&Full$team^Ahead!** Open the extracted folder and any subfolders until you reach the folder containing "setup" (may say "setup.exe" on some machines). Double click on the setup icon to launch the installer. Click "Yes" if a Windows Security prompt appears. Click "Next" on the splash screen that appears.

Product List



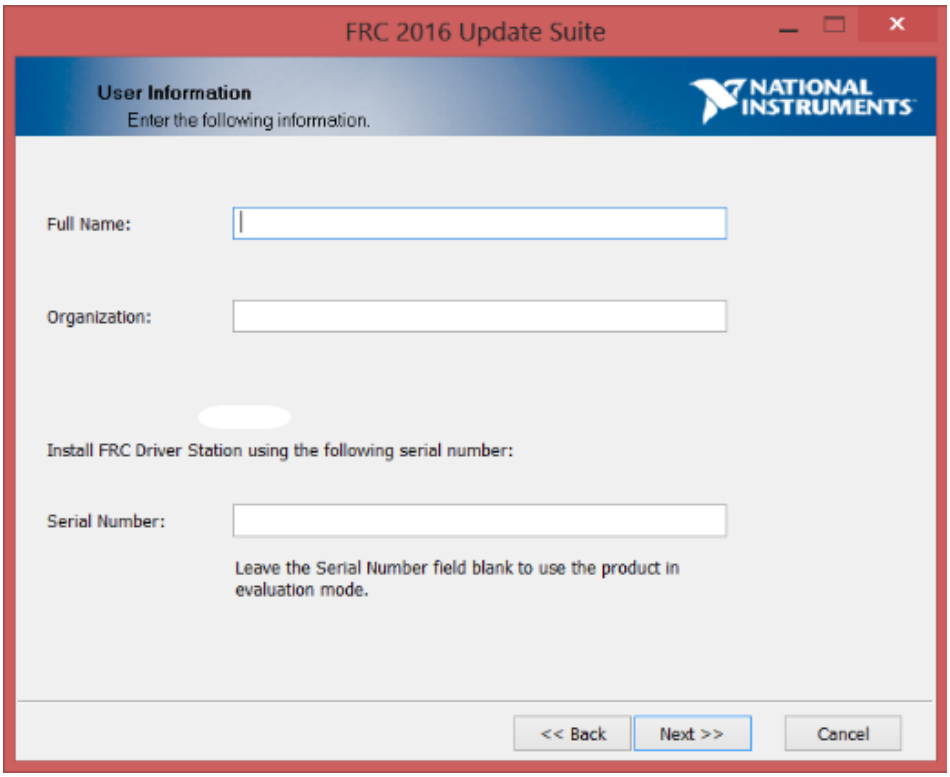
Click "Next"

Product Information



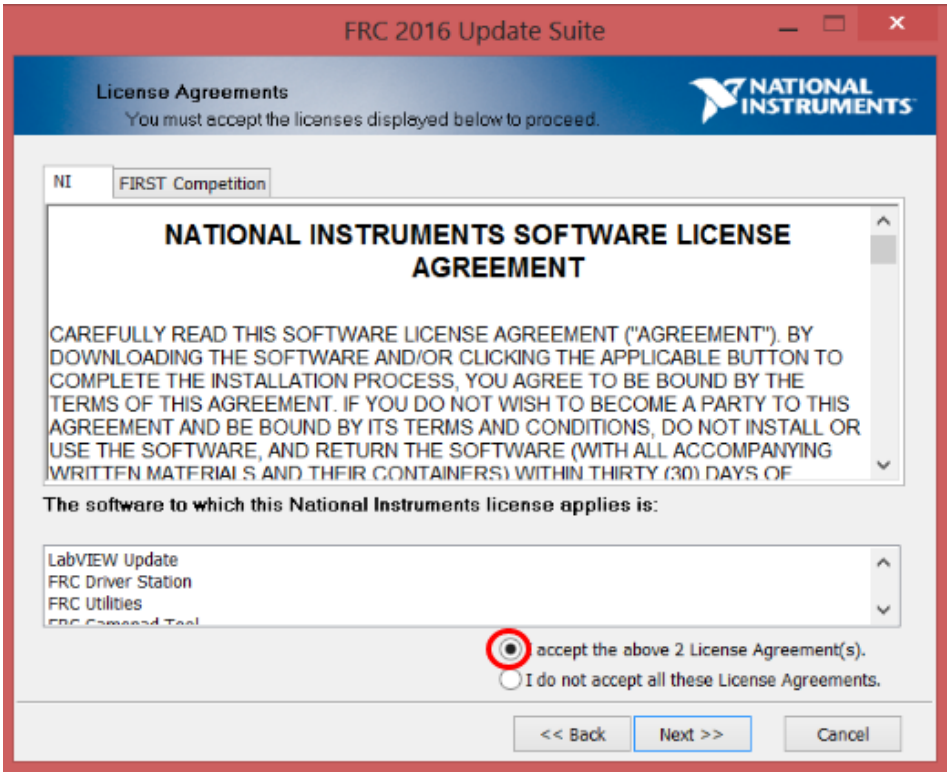
Un-check the box, then Click "Next"

User Information



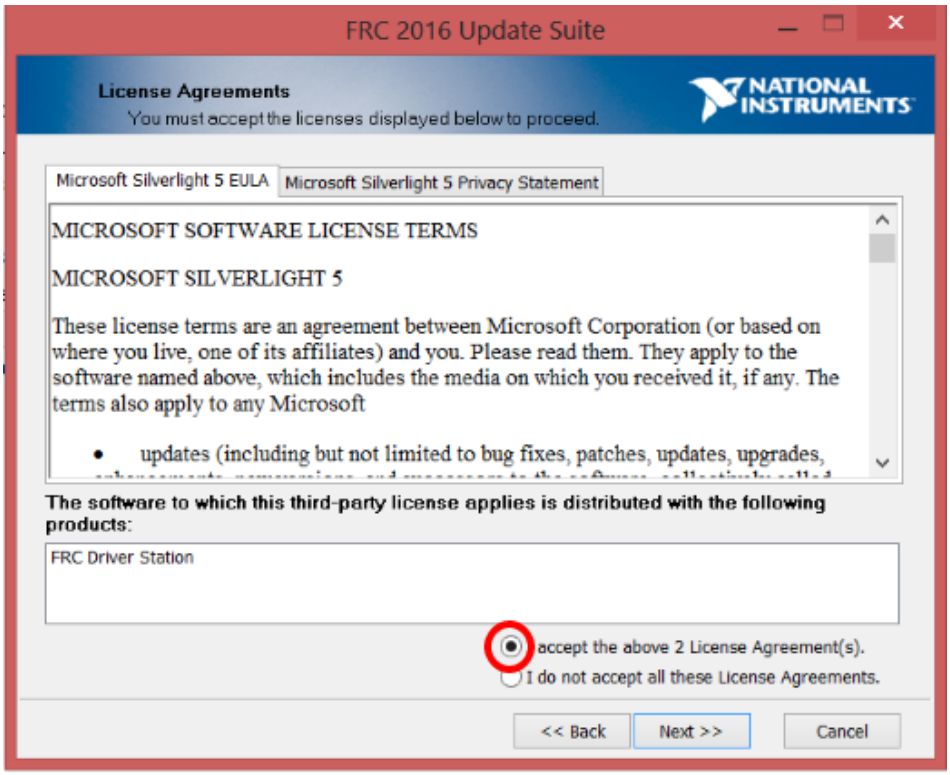
Enter full name and organization and the serial number from your kit of parts then click **Next**

License Agreements



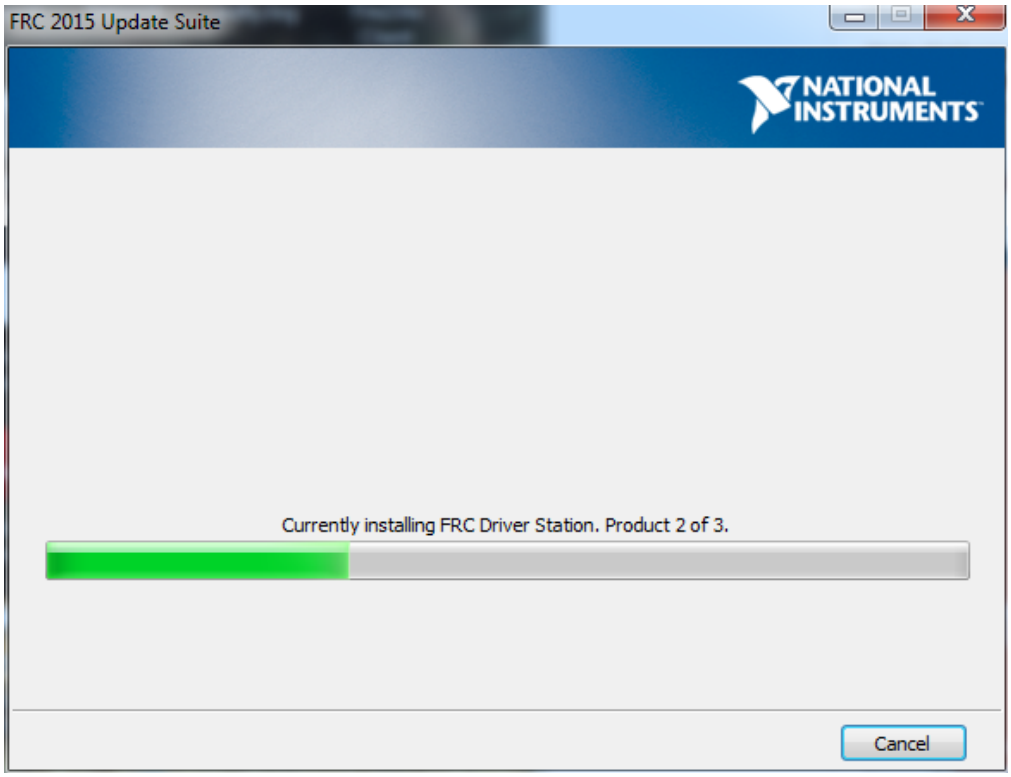
Select "I accept..." then click "Next"

License Agreements Page 2

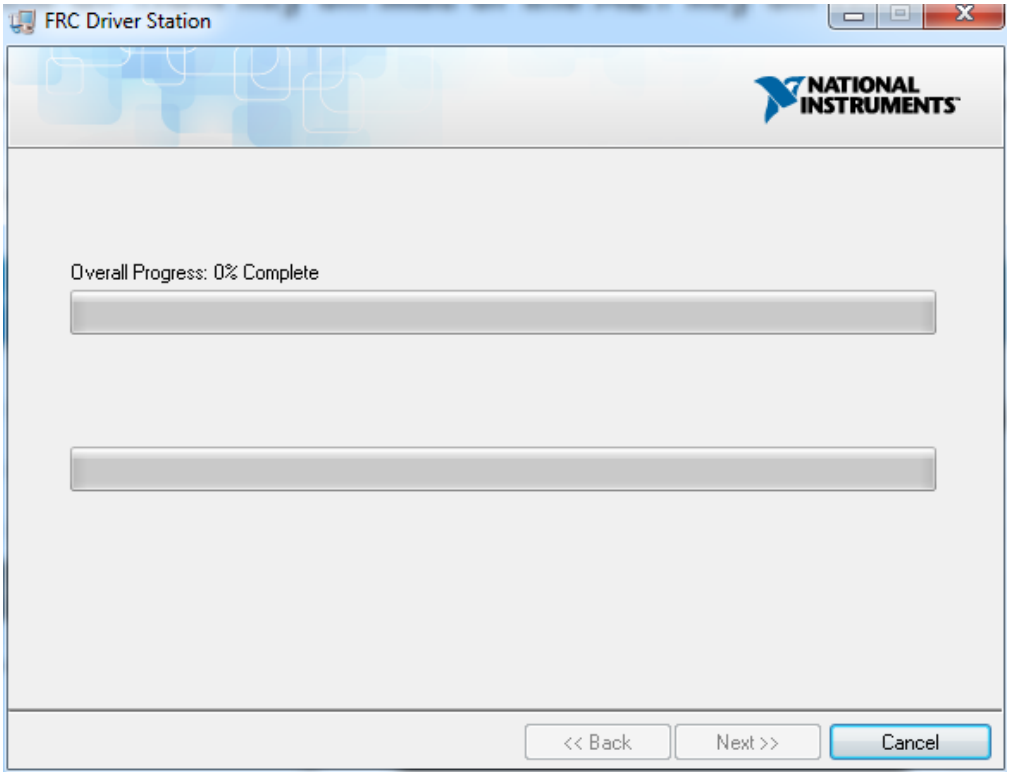


Select "I accept..." then click "Next"

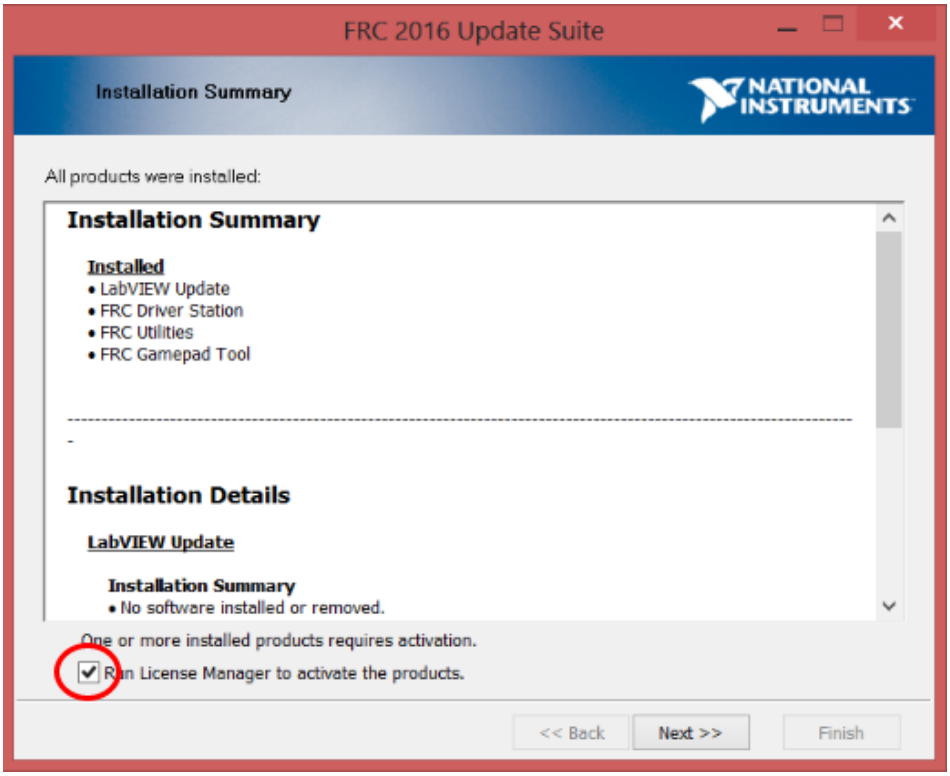
Summary Progress



Detail Progress

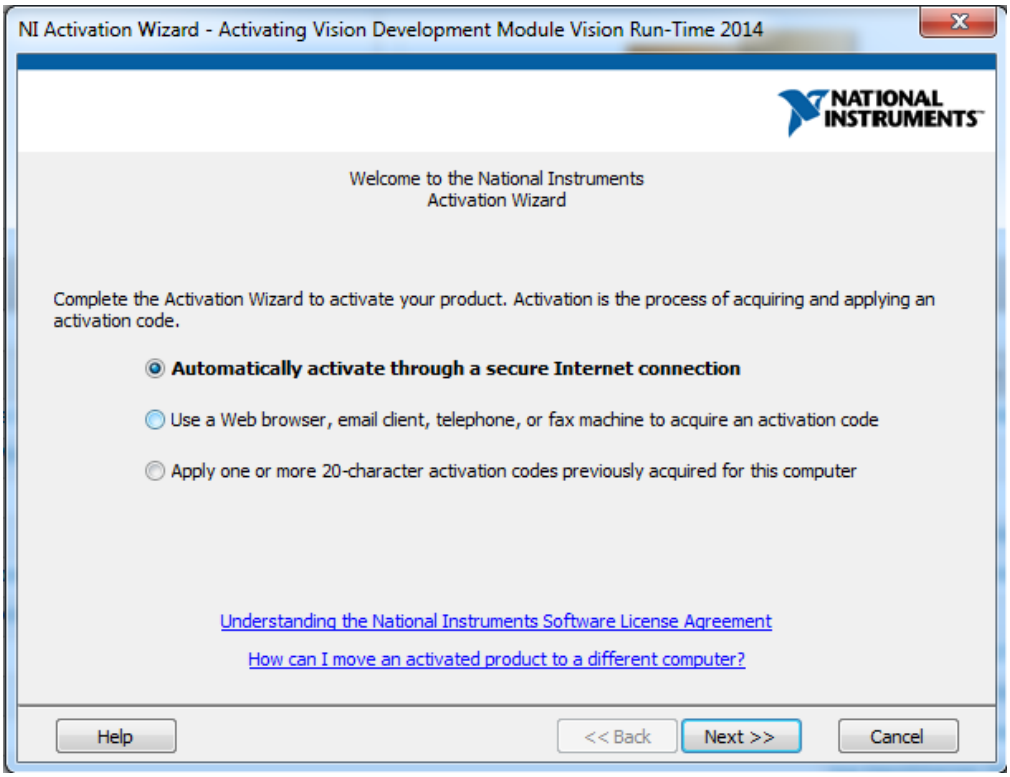


Installation Summary



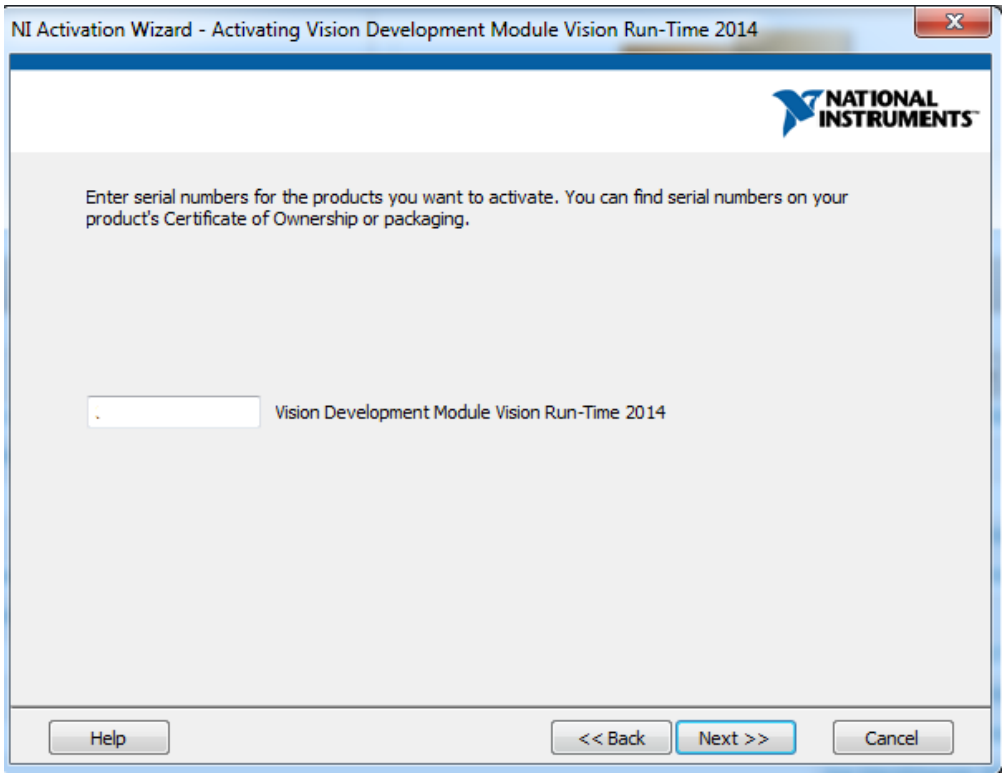
Make sure the box is checked to **Run License Manager...** then click **Next**

Activate



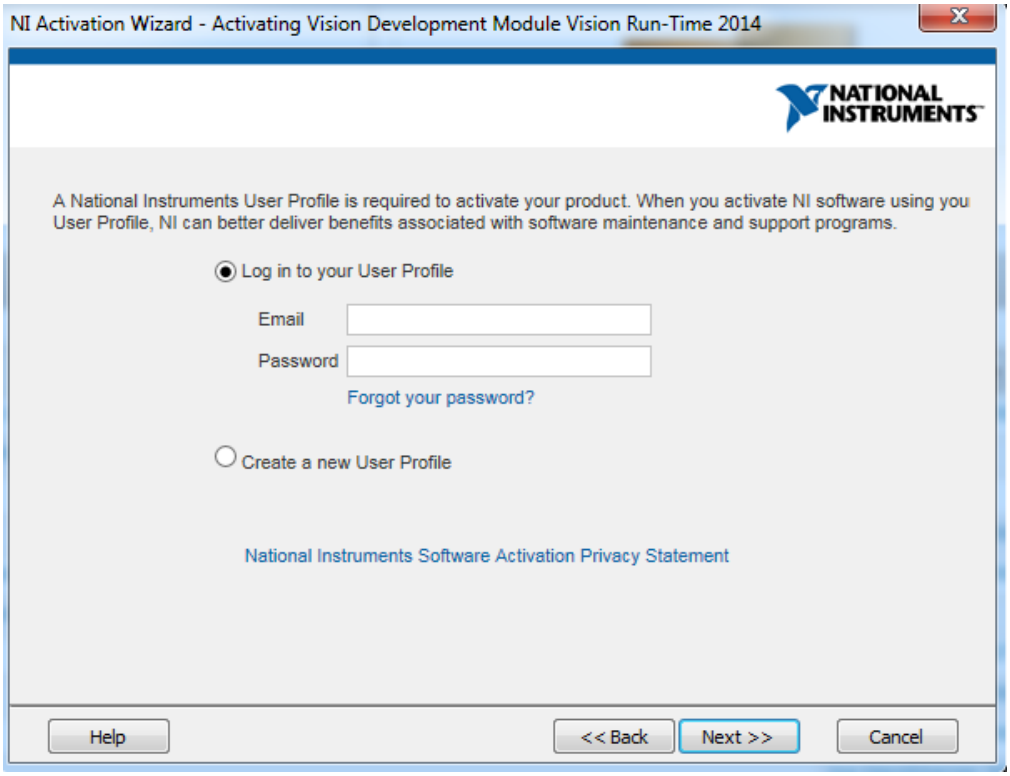
Select your desired activation method (Internet activation recommended), then click **Next**

Serial Number Confirmation



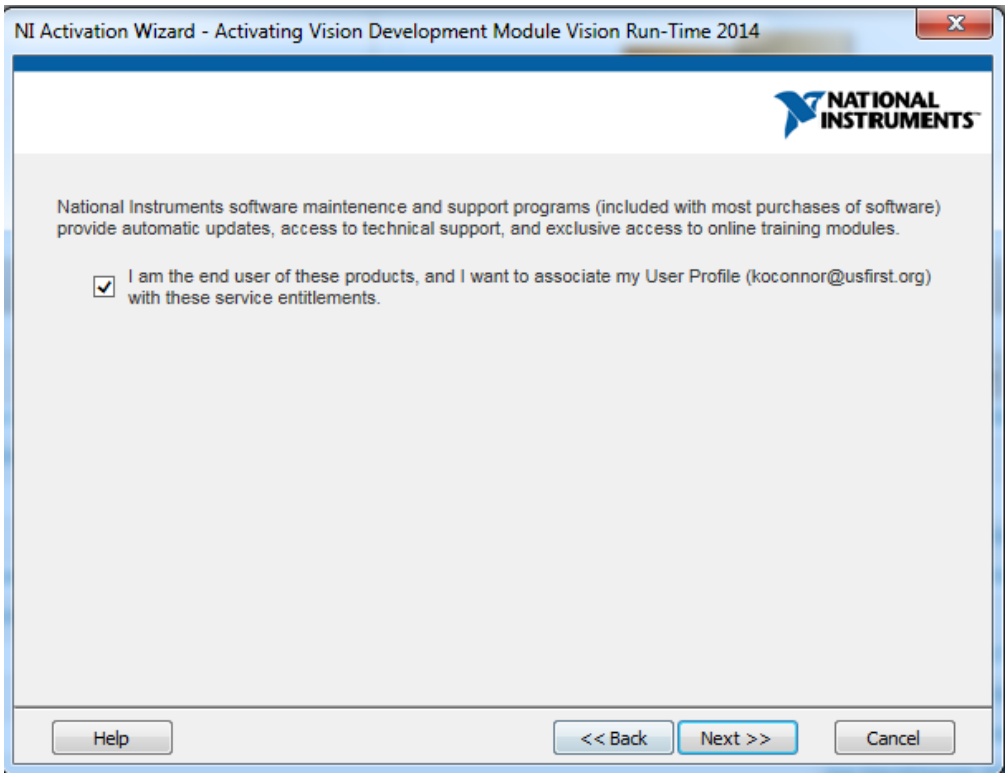
Make sure the serial number in the box matches the one from your kit, then click **Next**

User Profile



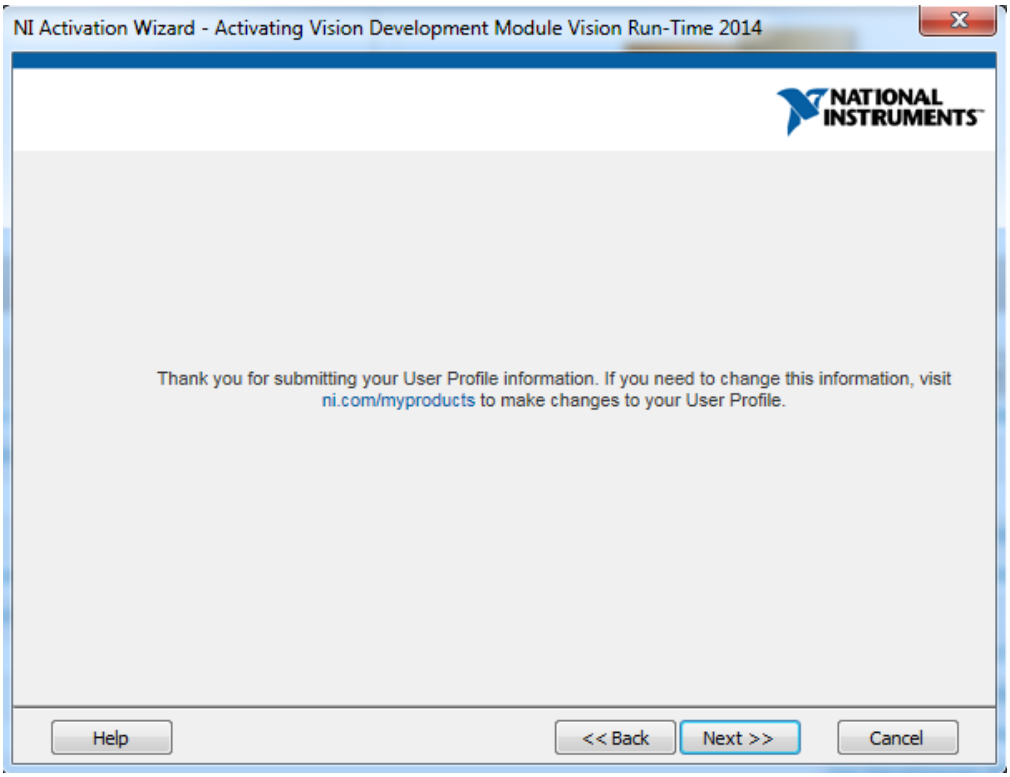
Log in or create an NI Profile. One profile may be used for multiple installations.

User Profile 2



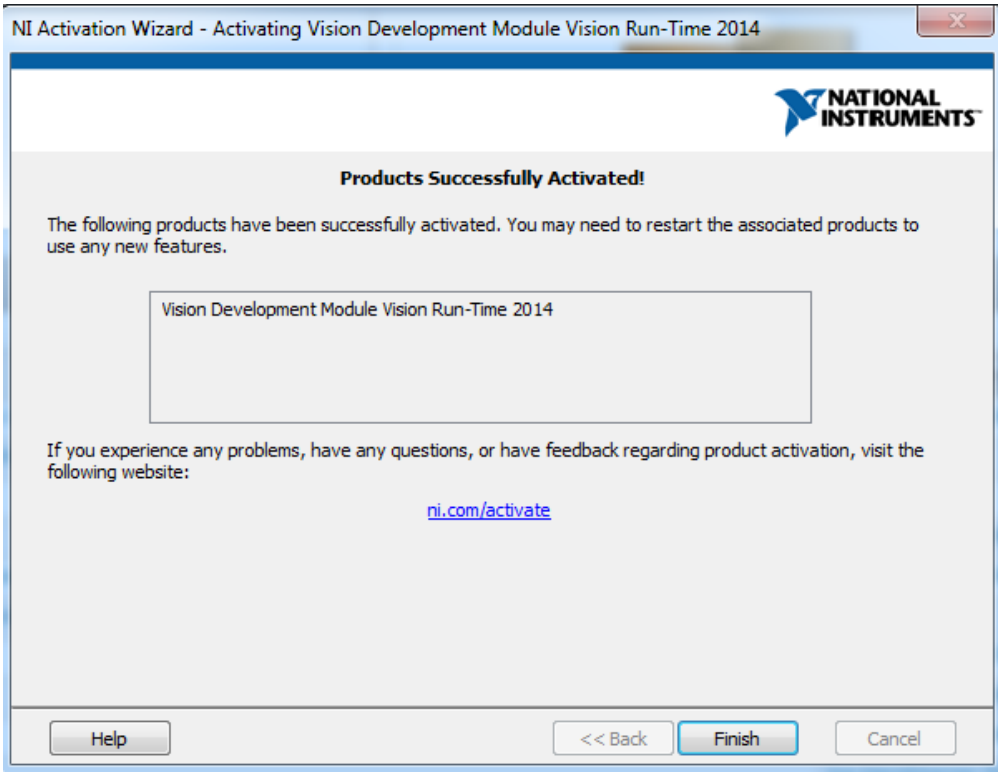
Click **Next**

User Profile 3



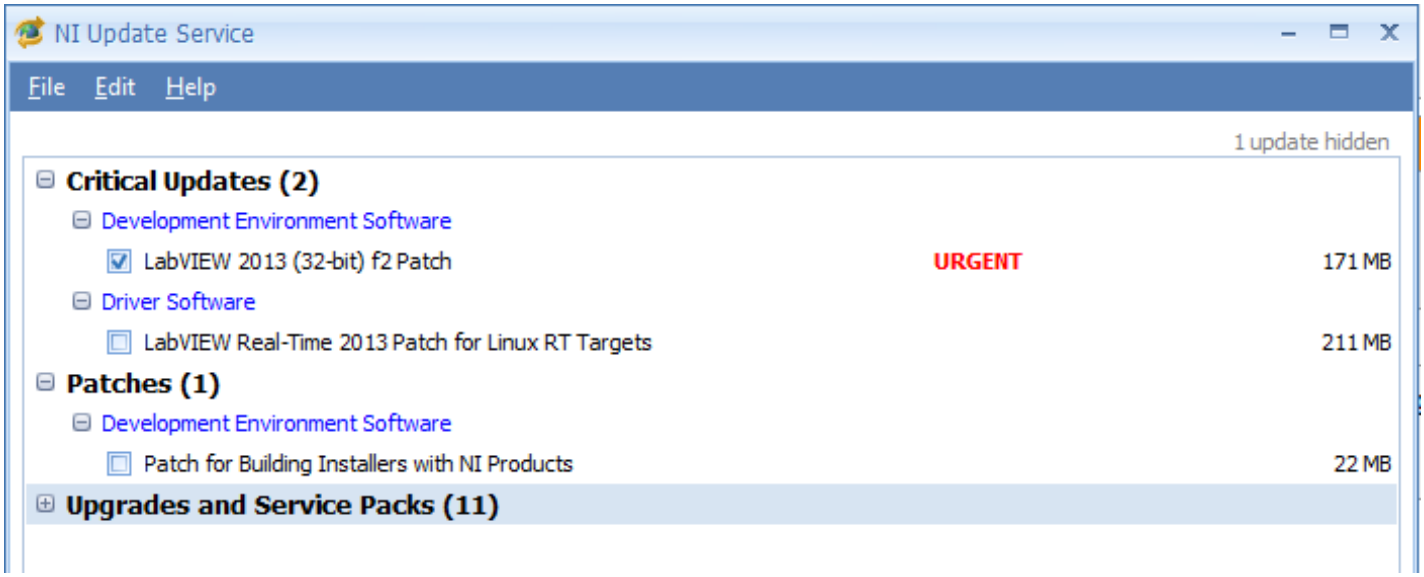
Click **Next**

Finish



After the product is activated, Click **Finish.** If prompted to Reboot, click **Yes**

NI Update Service

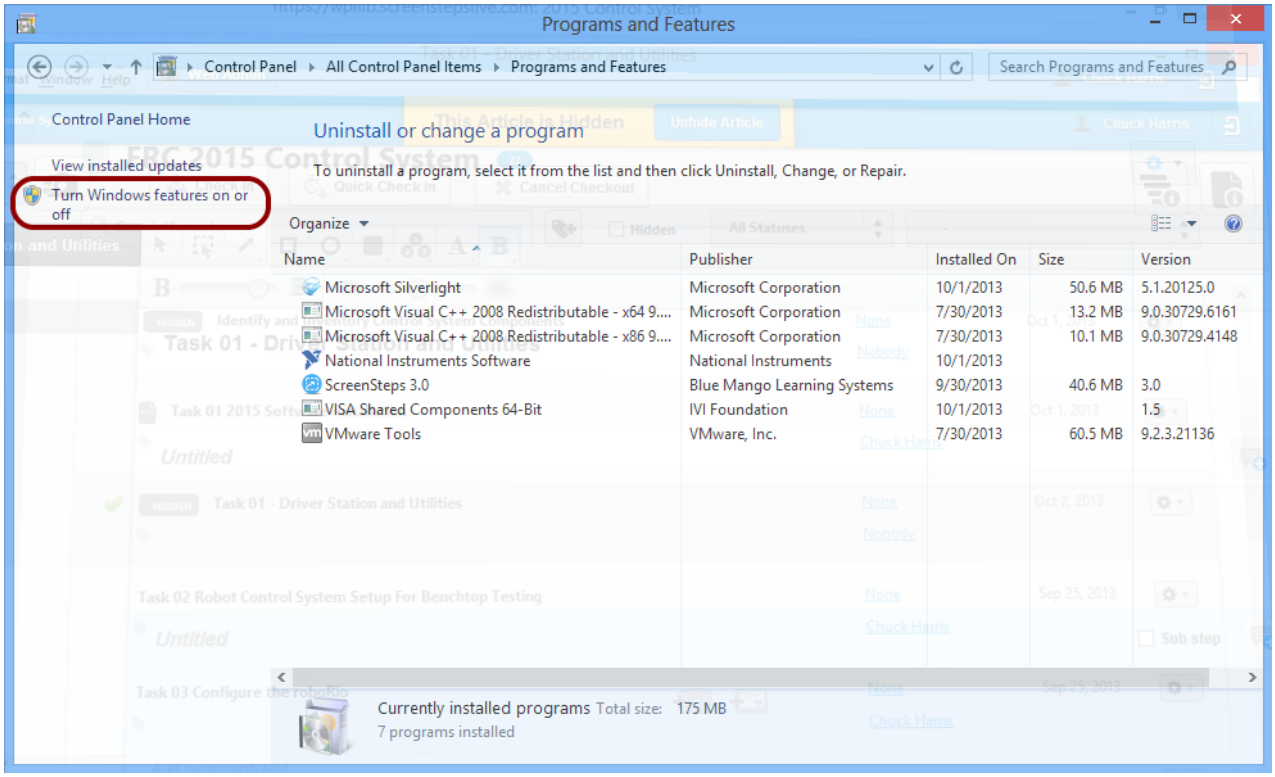


On occasion you may see alerts from the NI Update Service about patches to LabVIEW. It is not recommended to install these patches. **FRC will communicate any recommended updates through our usual channels** (Frank's Blog, Team Updates or E-mail Blasts).

Addendum - Installing on Windows 8

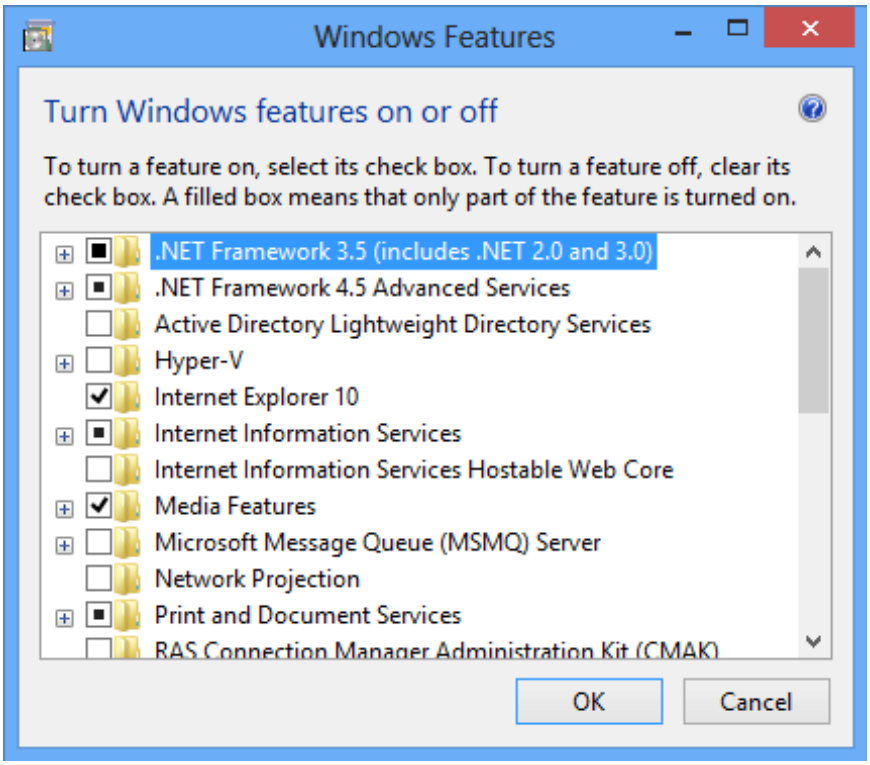
If installing on Windows 8 or 10, the Microsoft .NET Framework 3.5 may need to be installed. If you see the dialog shown above, click "Cancel" and perform the steps shown below. An internet connection is required to complete these steps.

Programs and Features



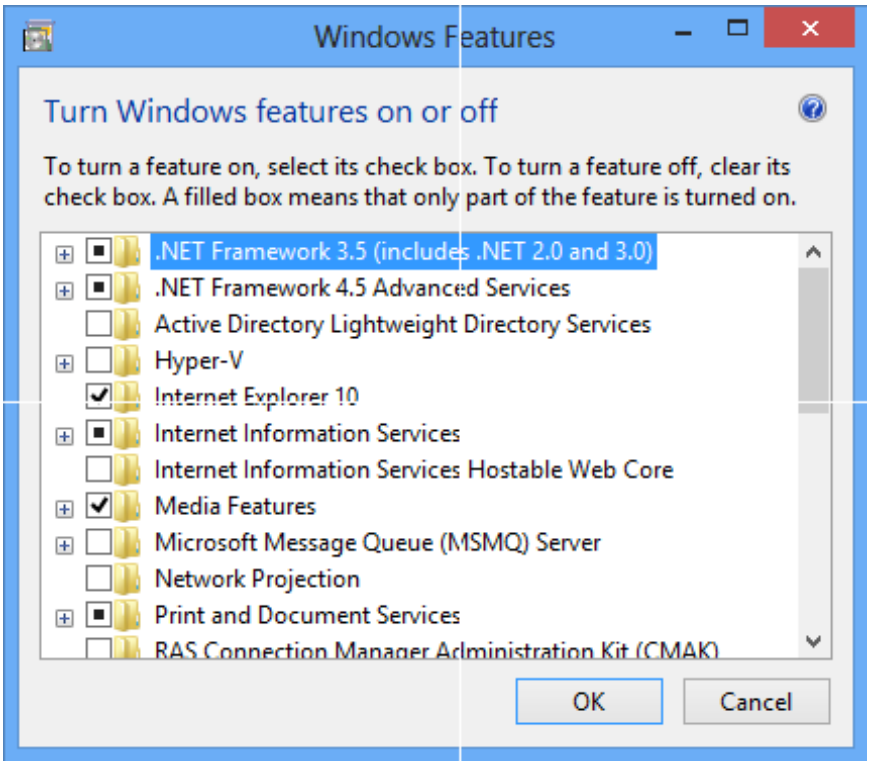
Open the "Programs and Features" window from the control panel and click on "Turn Windows features on or off"

Windows Features (.NET Framework 3.5 not on)



Select ".NET Framework 3.5 (includes .NET 2.0 and 3.0)" to enable it (a black dot, not a check box will appear) and then click "OK". When installation finishes restart installation of FRC 2016 Update Suite.

Windows Features (.NET Framework 3.5 already on)



If a black dot is shown next to ".NET Framework 3.5" the feature is already on. Click "Cancel" and restart installation of FRC 2017 Update Suite.

FRC C++ References

C++ WPILib API Documentation

http://first.wpi.edu/FRC/roborio/release/docs/cpp/

C++\Java Plugin Changelog

Changelog for the C++\Java Eclipse Plugins

2017.2.1 1/20/17

RobotBuilder:

- Added support for ConditionalCommand

WPILib:

- Fixed solenoid allocation error message

- Documentation fixes

CameraServer (WPILib and cscore):

- Removed NetworkTables-driven camera settings (could prevent camera settings changes from taking effect due to dashboard caching)

- Fixed exception on camera creation when camera is not plugged in

- Auto-increment startAutomaticCapture()

- Added properties setting capability through web server

Eclipse Plugins:

- Fix RoboRIO image version detection in presence of EOL in configuration

- Improve IP address resolution during deployment

- Excluded sources and javadoc JARs from user libs when building

- Java and C++ debug deploys now deploy libraries

- Added option to disable automatic library management

NetworkTables:

- Documentation fix

2017.1.1 (Kickoff Release) 1/5/17

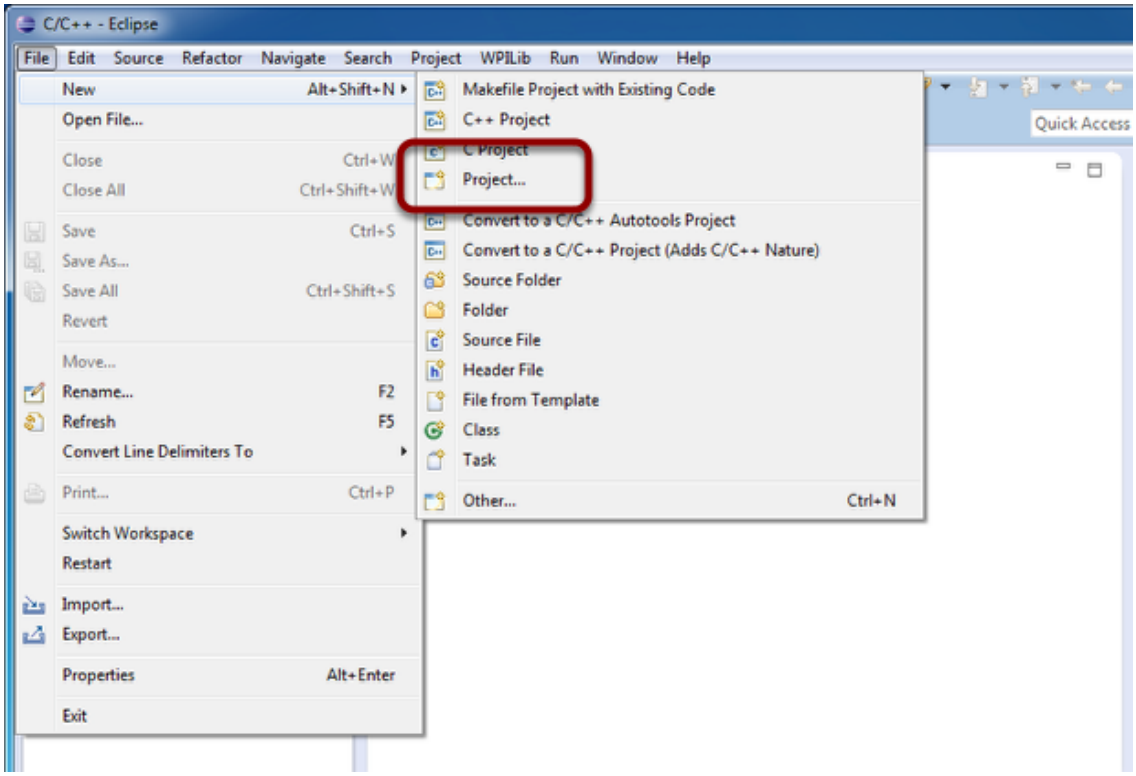
Creating and Running Robot Programs

Creating your Benchtop Test Program

The simplest way to create a robot program, is to start from one of the three supplied templates (Sample, Iterative or Command). Sample is best used for very small sample programs or advanced programs that require total control over program flow. Iterative Robot is a template which provides better structure for robot programs while maintaining a minimal learning curve. Command-Based robot is a template that provides a modular, extensible structure with a moderate learning curve.

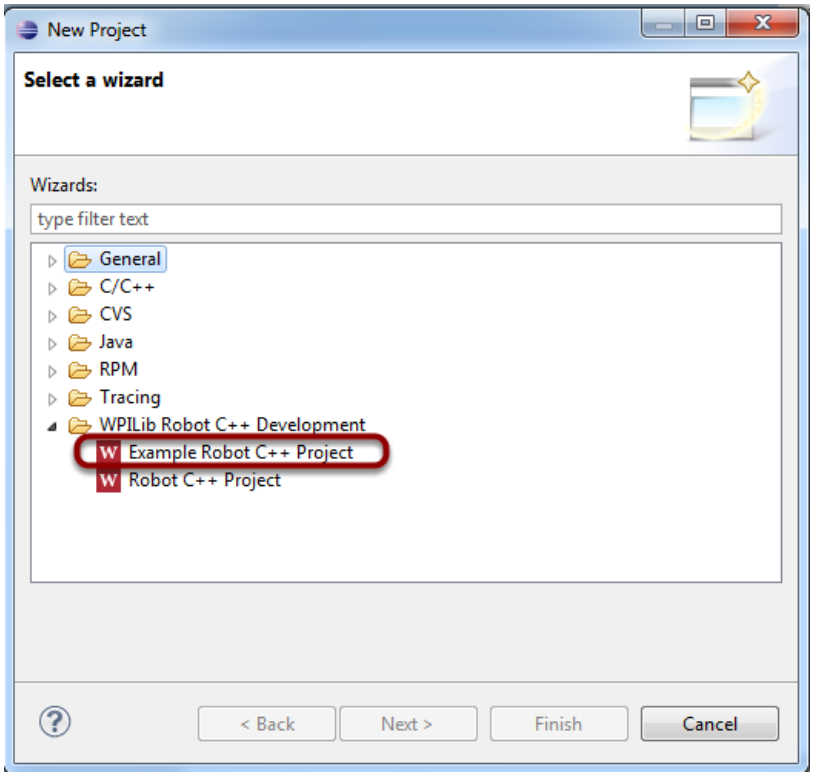
The templates will get you the basis of a robot program, organizing a larger project can often be a complex task. RobotBuilder is recommended for creating and organizing Command-Based robot programs. You can learn more about RobotBuilder here. To create a command-based robot program that takes advantage of all the newer tools look at Creating a Robot Project in the Command Based Programming Chapter.

Creating a project



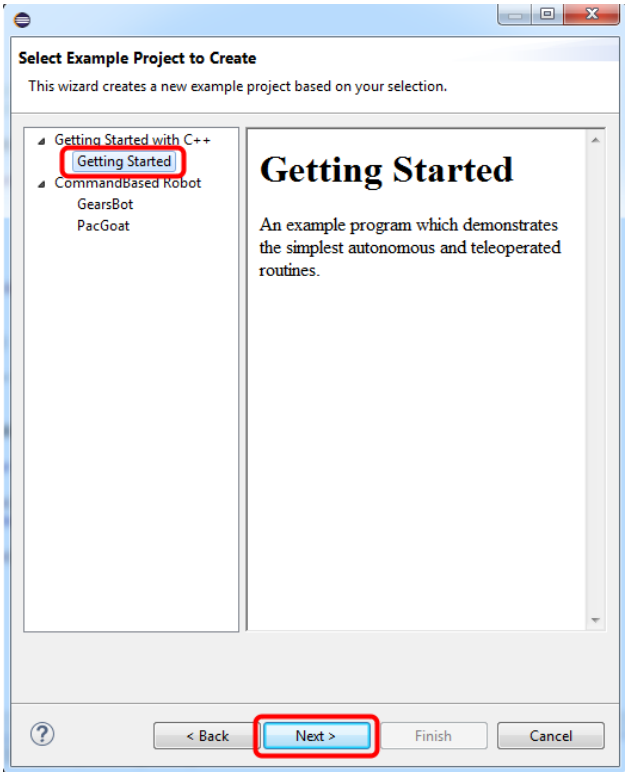
To create a project click "File" then "New" then click "Project...".

Selecting the project type (Example Robot C++ Project)



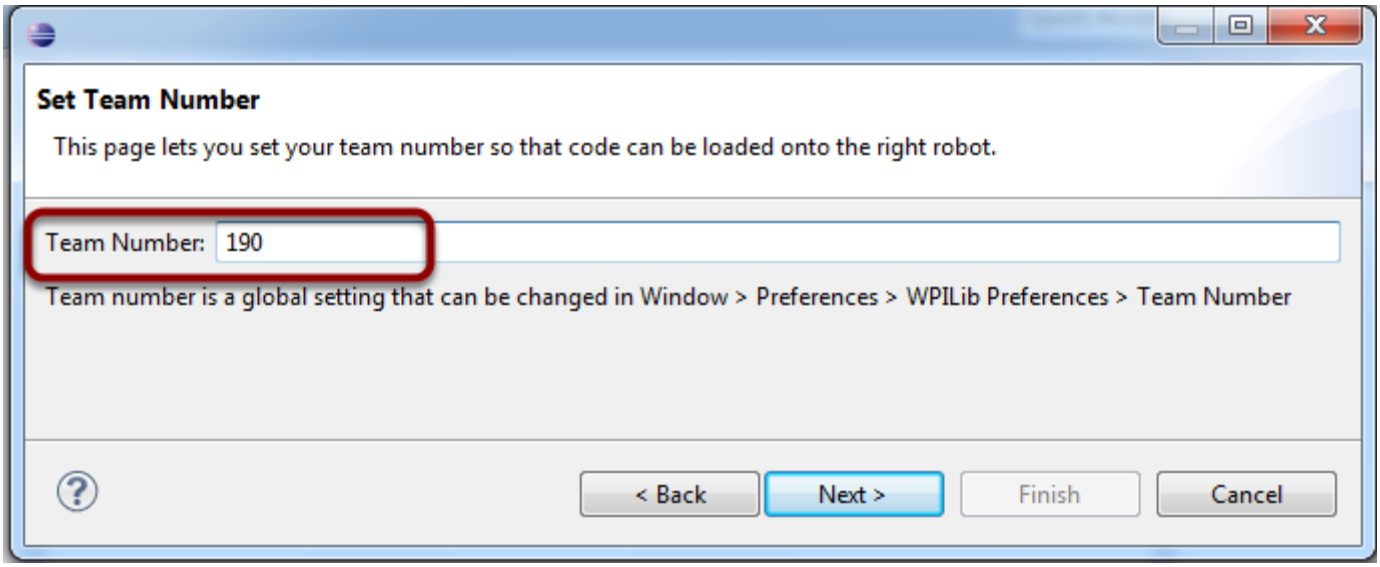
Choose **Example Robot C++ Project** as the project type. For creating future robot programs you will likely select **Robot C++ Project**.

Selecting an Example



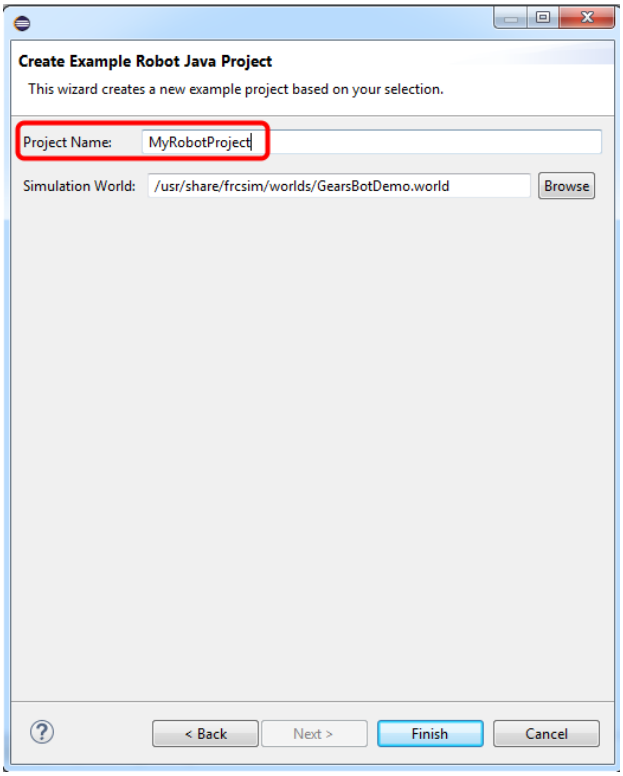
Select the **Getting Started** example, then click **Next.**

Entering the team number



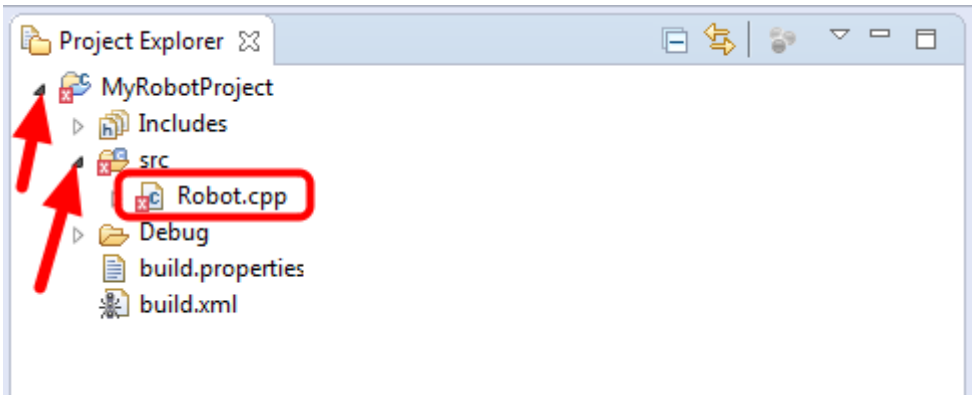
Enter your team number. This is used when the Eclipse tools download programs onto your RoboRIO. As described in the dialog this is a global setting that can also be accessed from Window->Preferences->WPILib Preferences (if you need to change what team number you are targeting). This dialog will only be shown if there is no team number currently set in Eclipse global settings.

Project Name



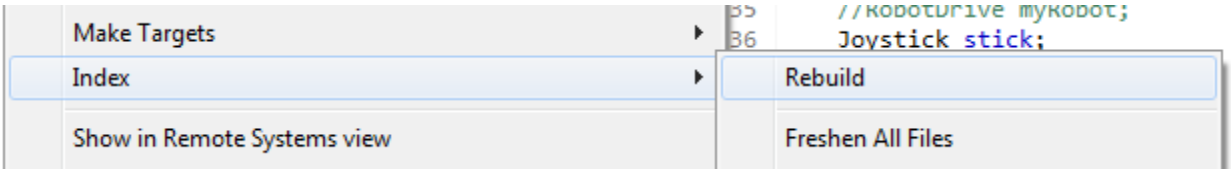
Enter a name for your robot project. The simulation world is used for simulation and can be left at the default option. Then click Finish.

Eclipse Project Explorer



The project will be created and appear in the Eclipse Project Explorer. Click the arrows to expand the project, then the **src** folder. Double click on **Robot.cpp** to open it.

Rebuilding the Index



It is currently necessary to re-build the index after creating a new template or example for the Eclipse syntax checker to work properly. To do this, right click on the project in the Project Explorer and select Index->Rebuild.

Defining the variables for our sample robot

C++

```
class Robot: public IterativeRobot
{
        /*
         * Defines the variables as members of our Robot class
         */
        RobotDrive myRobot; // robot drive system
        Joystick stick; // only joystick
        LiveWindow *lw;
        int autoLoopCounter;
```

```
public:
        Robot() :
                /* Initializes the variables as part of the constructor using an
Initialization
                 * list.
                 */
                myRobot(0,1), //these must be initialized in the same order
                stick(1),          //as they are declared above.
                lw(NULL),
                autoLoopCounter(0)
                {
                        myRobot.SetExpiration(0.1);
                        /*
                         * Performs robot initialization
                         * (in this case sets the safety timer expiration for
                         * the myRobot object to .1 seconds, see the next step for an
explanation of
                         * the motor safety timers).
                         */
                }
```

The sample robot in our examples will have a joystick on USB port 1 for arcade drive and two motors on PWM ports 0 and 1. Here we create objects of type RobotDrive (myRobot), Joystick (stick), LiveWindow (*lw), and integer (autoLoopCounter). This section of the code does three things:

1. Defines the variables as members of our Robot class.
2. Initializes the variables as part of the constructor using an Initialization List.
3. Performs robot initialization (in this case sets the safety timer expiration for the myRobot object to .1 seconds, see the next step for an explanation of motor safety timers).

Robot Initialization

C++

```
        void RobotInit()
        {
                lw = LiveWindow::GetInstance();
        }
```

The RobotInit method is run when the robot program is starting up, but after the constructor. The RobotInit for our sample program gets a pointer to the LiveWindow instance (this is used in the test method discussed below)

Simple autonomous sample

C++

```
        void AutonomousInit() //This method is called once each time the robot enters
Autonomous
        {
                autoLoopCounter = 0;
        }

        void AutonomousPeriodic() /* This method is called each time the robot receives a
        * packet instructing the robot to be in Autonomous Enabled mode (approximately
every 20ms)
        * This means that code in this method should return in 20 ms or less (no delays or
loops)
        */
        {
                /* This example checks if the loop counter has reached 100
                * (approximately 2 seconds) THis is a fairly simple example and is not
recommended for actual
                * autonomous routines, because delayed or missed packets will
significantly affect the timing
```

```
                * of this program
                */
                if(autoLoopCounter < 100)
                {
                        myRobot.Drive(-0.5, 0.0); /* If autoLoopCounter has not yet
reached 100,
                        * this instructs the robot to drive forward at half speed, and
increment the counter
                        **/
                        autoLoopCounter++;
                } else {
                        myRobot.Drive(0.0, 0.0); // If autoLoopCounter has reached 100,
this stops the robot
                }
        }
```

Joystick Control for teleoperation

C++

```
        void TeleopInit()
        {
                /* The TeleopInit method is called once each time the robot enters
teleoperated mode
                *  This is where you would put presets for the teleoperated mode.
                */
        }

        void TeleopPeriodic()
        {
                /* The TeleopPeriodic method is called each time the robot recieves a
packet instructing it
                * to be in teleoperated mode
                */
                myRobot.ArcadeDrive(stick); // This line tells the drivetrain to use the
joystick to control the
```

```
                // robot with the Arcade Drive scheme (Y-Axis throttle, X-Axis turn).
        }
```

Test Mode

```
C++

        void TestPeriodic()
        {
                lw->Run();
        }
```

Test Mode is used for testing robot functionality. The test mode of our sample program runs LiveWindow. LiveWindow is a part of the SmartDashboard that allows you to see inputs and control outputs on the robot from the dashboard when the robot is in Test Mode. You can read more about LiveWindow in the SmartDashboard section of the Driver Station Manual.

Final Details

Some final details:

- In this example myRobot, and stick are member objects of the RobotDemo class. They're accessed using references, one of the ways of accessing objects in C++. See the section on pointers as an alternative method of using WPILib objects.
- The myRobot.Drive() method takes two parameters: a speed and a turn rate. See the documentation about the RobotDrive object for details on how the speed and direction parameters work.

To learn about running this program on the roboRIO see the next article.

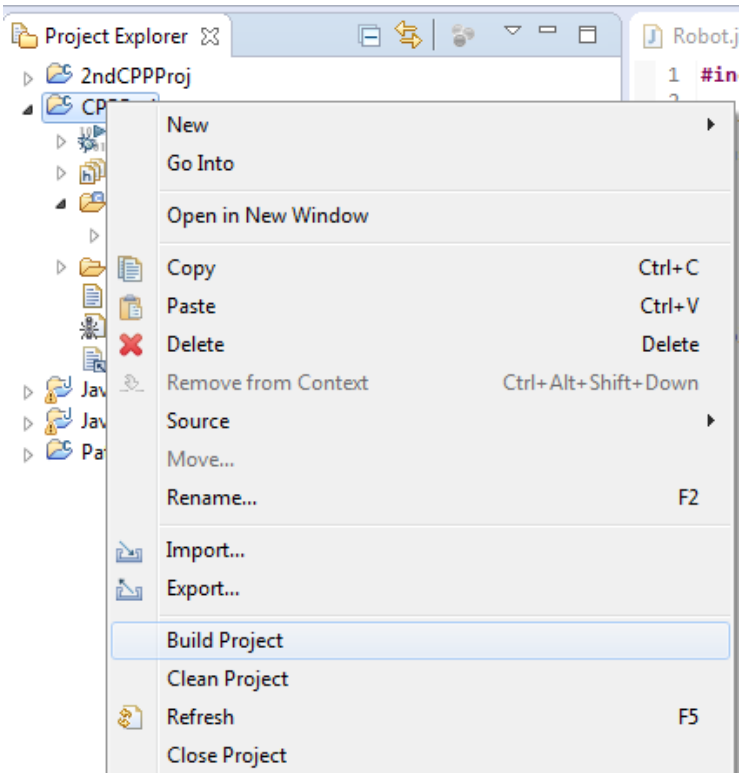
Building and downloading a robot project to the roboRIO

There are two ways of running programs onto the roboRIO. You can

1. Attach to the roboRIO and run the program using the debugger from your development system OR
2. Load it onto the roboRIO flash drive so it will run on reboot.

For tournaments you should always download the program so that it will be there when the robot is restarted and the match is played. This article will cover loading the program onto the roboRIO to run on reboot. The next article will cover the debugger.

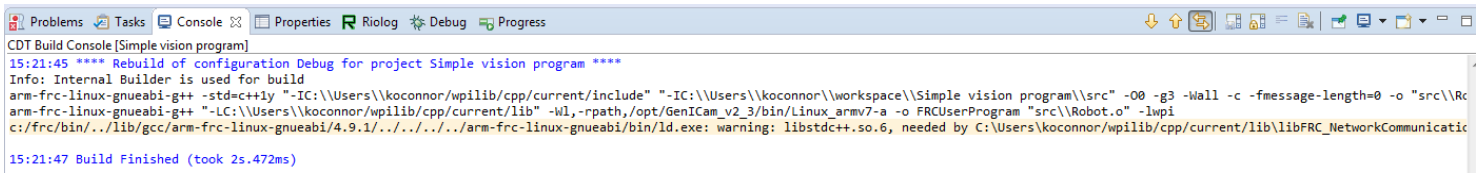
Building a robot program



The first step to getting the program on the roboRIO is to build the code. This will compile and link the project files. If you have set the option to build automatically as recommended in the Eclipse installation article, this step is done each time you save a file in the project. If not, or if you just

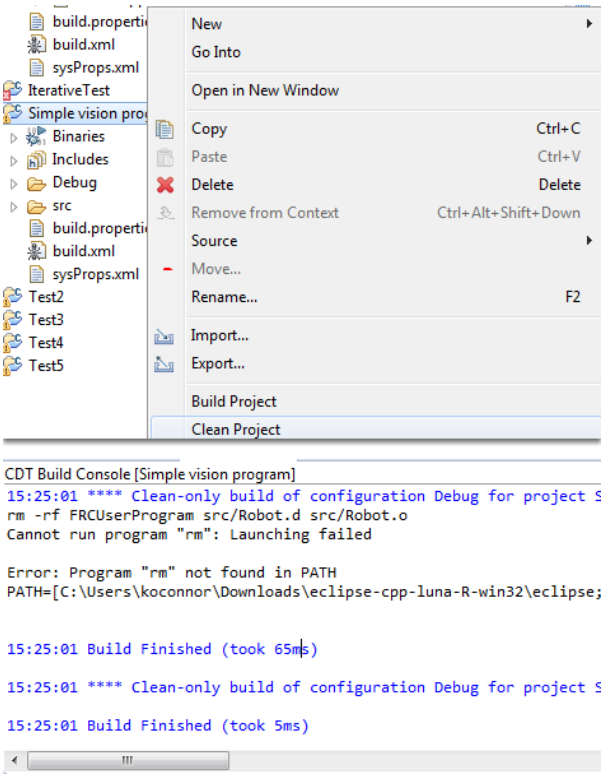
want to be sure, you can build the project by right clicking on it in the Project Explorer (left pane) and selecting Build Project

Build Console



A successful build will look similar to the image above. Note the warning in yellow, this warning is harmless and will occur each time you try to build your project, it can be safely ignored.

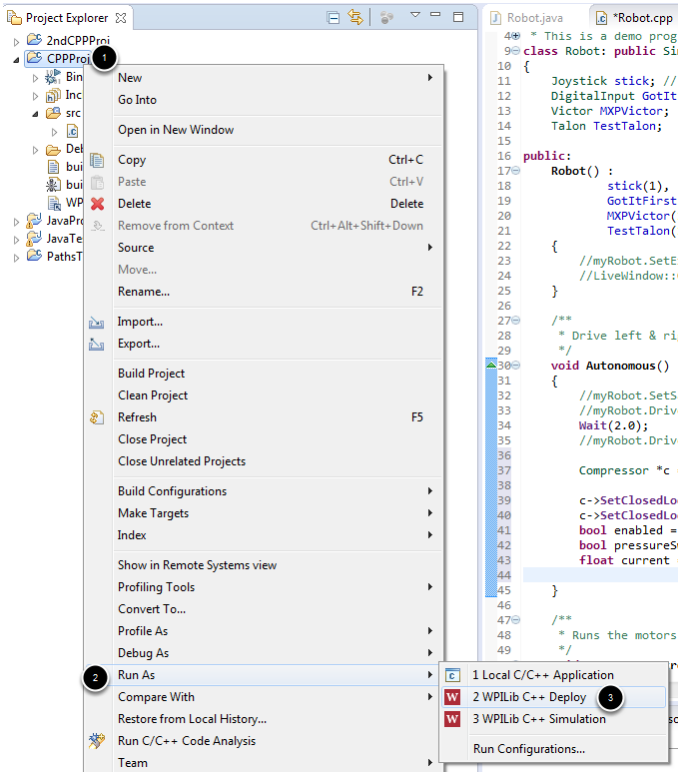
Clean



If you want to be absolutely sure your project is re-built, you may wish to do a Clean before building. Right click on the project and select "Clean Project". This will delete the build products for your project.

Note the errors shown in the image above. These errors are normal and do not indicate that the Clean did not complete successfully. To verify that the Clean succeeded, make sure that the "Debug" directory no longer shows up under the Project in the Project Explorer (the left pane of the Eclipse window).

Loading a program to run on robot startup



To have the program run automatically each time the roboRIO boots, it must be loaded in the roboRIO flash memory. For tournaments you should always download the program so that it will be there when the robot is restarted and the match is played. To do this:

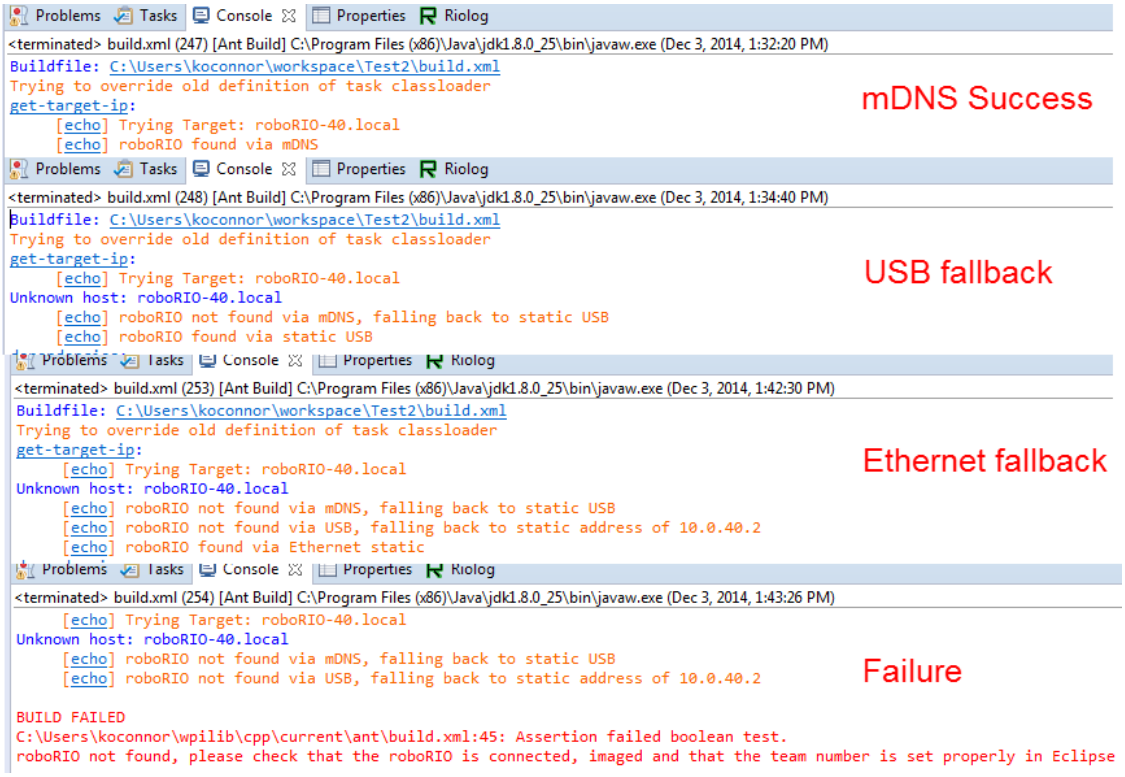
1. Right click on the project in the Project Explorer
2. Select "Run As"
3. Select "WPILib C++ Deploy"

The deployment process will then begin and start outputting status information to the Console (generally located in the bottom center or bottom right of the window). **After the program is downloaded it will be immediately launched, there is no need to reboot the roboRIO.**

Program Download Process

After you click the WPILib C++ Deploy button the deploy process will proceed through a number of steps.

Get Target IP

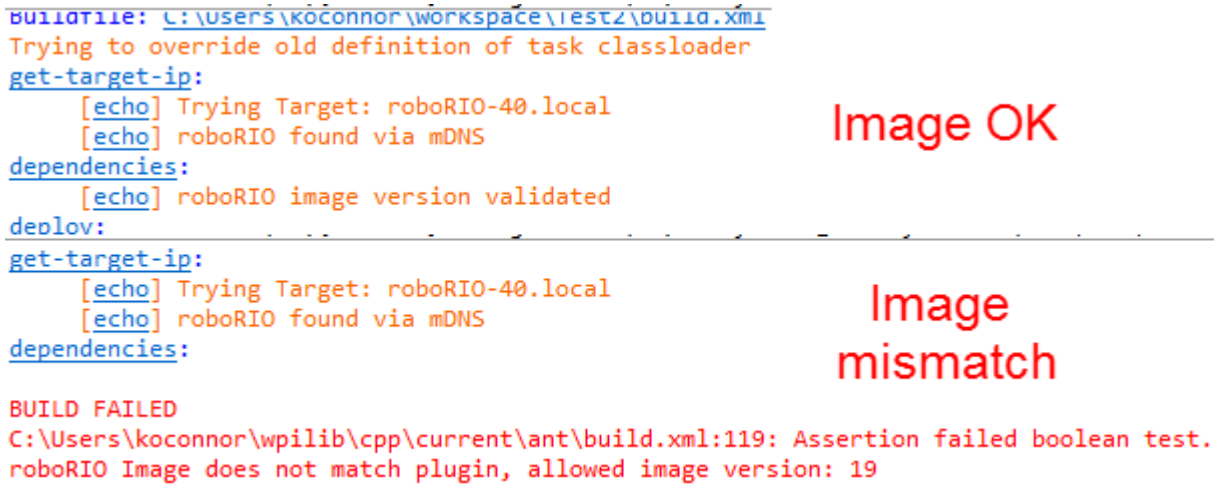


In this step, the plugin determines the IP of the roboRIO. The plugin will try to reach the roboRIO via the following methods, in order:

1. mDNS (roboRIO-TEAM.local where TEAM is your team number)
2. USB static IP (172.22.11.2)
3. Ethernet static IP (10.TE.AM.2)

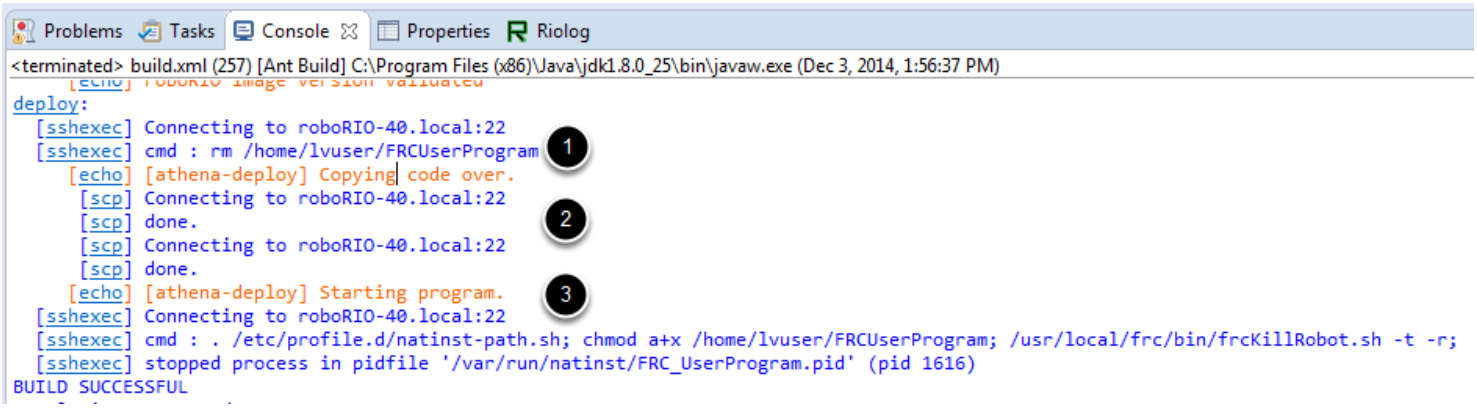
If the roboRIO is accessible via any of these methods, the deploy will proceed using that address as the target. If the roboRIO cannot be found using any of these methods, the deploy will abort. See Connection Failure in the Troubleshooting section below.

Dependencies



The dependencies section checks that the roboRIO contains any dependencies necessary for operation. For C++ at this time, this consists of a check of the roboRIO image version. If the validation succeeds you will get a message that the version was validated and the deploy will proceed. If the check fails, you will see a message indicating the allowed image version for this version of the plugins, see Image Check Failure under Troubleshooting below for more details

Copy and Start code



The remaining parts of the deploy then proceed as follows:

1. Delete existing program. This is required because you cannot perform an overwrite while the program is running. If there is no existing program you will see a red error message under this line, but the deploy will proceed.
2. Copy user program and robotCommand file. The robotCommand file tells the framework that launches the code what to launch (C++ vs. Java and Debug vs deploy).

3. Start code. The plugin calls a script file which kills the existing user program, then restarts the LabVIEW RT process. This starts the code in the same way it will be launched when the roboRIO is booted.

Deployed Code

There is no need to reboot the roboRIO to run the new code, it will begin running automatically. After the deploy completes, each time the roboRIO is rebooted it will run the deployed code until new code (from any FRC language) is downloaded.

Viewing program output

The console output from the program does not appear in the regular Eclipse console. To view the console output (cout or printf statements as well as WPILib output) see the next article, Using Riolog to view console output.

Next Steps

To continue Getting Started with the 2015 Control System, see Programming your radio for home use

To continue reading about C++, scroll down and click Next to read about Using Riolog to view console output.

Troubleshooting

This section contains troubleshooting steps for commonly encountered issues.

Connection Failure

If the deploy process fails to connect to the roboRIO:

- First verify that the roboRIO is powered on, connected to the PC and that the roboRIO is imaged and that your team number is set properly in Eclipse (Window->Preferences->WPILib Preferences).
- The next step is to verify that your computer networking is configured in a way it can communicate with the roboRIO. Visit the roboRIO Network Troubleshooting document in the Troubleshooting section for more details.

- Check that your firewall is disabled or that an exception has been added for Eclipse. For the Windows Firewall, see Configuring Windows Firewall in the Troubleshooting section for more details.

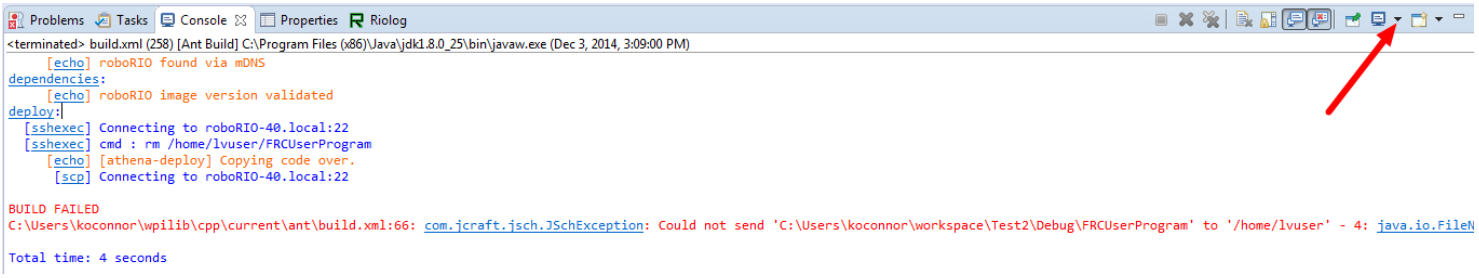
Image Check Failure

If the Image Check fails that means that the Image version on your roboRIO did not match the image version that the plugins were built against. To check for updates to the Eclipse plugins, make sure you installed the plugins from the web repository and that you have an active internet connection, then click Help->Check For Updates. To check for updates to the NI FRC 2015 Update Suite, which may contain a new roboRIO image, visit the URL from the Installing the FRC 2015 Update Suite (All Languages) document. If an updated version is available, after installing it as described in that document, re-image your roboRIO using the instructions in the Imaging your roboRIO document.

sysProps.xml doesn't exist

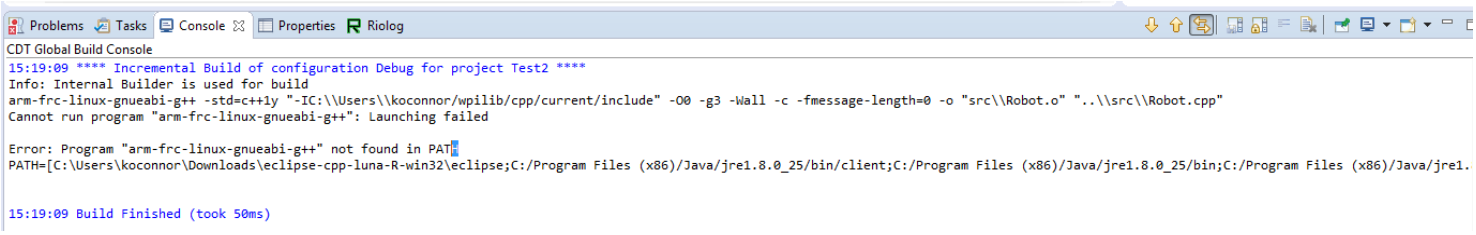
If you get an error that says something like "wpilib\cpp\current\ant\build.xml:117: Getting Started\sysProps.xml doesn't exist" this indicates that your roboRIO is not imaged or the image is corrupt. Re-image your roboRIO using the instructions in the Imaging your roboRIO document.

File Not Found



If the deploy fails with a "java.io.FileNotFoundException" this means that your program did not build correctly. To see the build failure, either build the program as described in Building a robot program above or click on the dropdown arrow next to the computer monitor icon and select "CDT Global Build Console". The output in the console should provide information about the reason the code failed to build.

Program "arm-frc-linux-gnueabi-g++" not found in PATH



If you receive the error "arm-frc-linux-gnueabi-g++" not found in PATH as shown above, it likely means you need to install the C++ toolchains as described in Installing Eclipse (C++/Java). If you have installed the toolchains, look through the PATH entry output by Eclipse and search for an entry that says C:\frc\bin (If it shows up as a drive other than C:\ verify that drive is a permanent drive on your machine, if not you may need to uninstall and re-install the toolchain, making sure to run the installer form a permanent drive). If the entry is not present, try re-installing the toolchains.

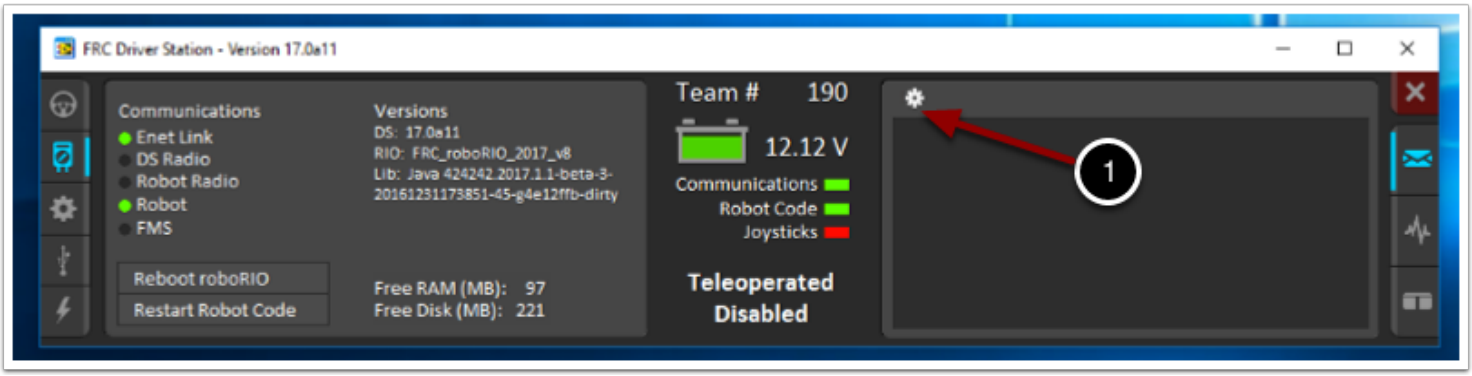
Viewing Console Output

For viewing the console output of text based programs the roboRIO implements a NetConsole very similar to the cRIO. Note that on the roboRIO, the NetConsole is only for program output, if you want to interact with the system console you will need to use SSH or the Serial console.

There are two main ways to view the NetConsole output from the roboRIO: The Console Viewer in the FRC Driver Station and the Riolog plugin in Eclipse.

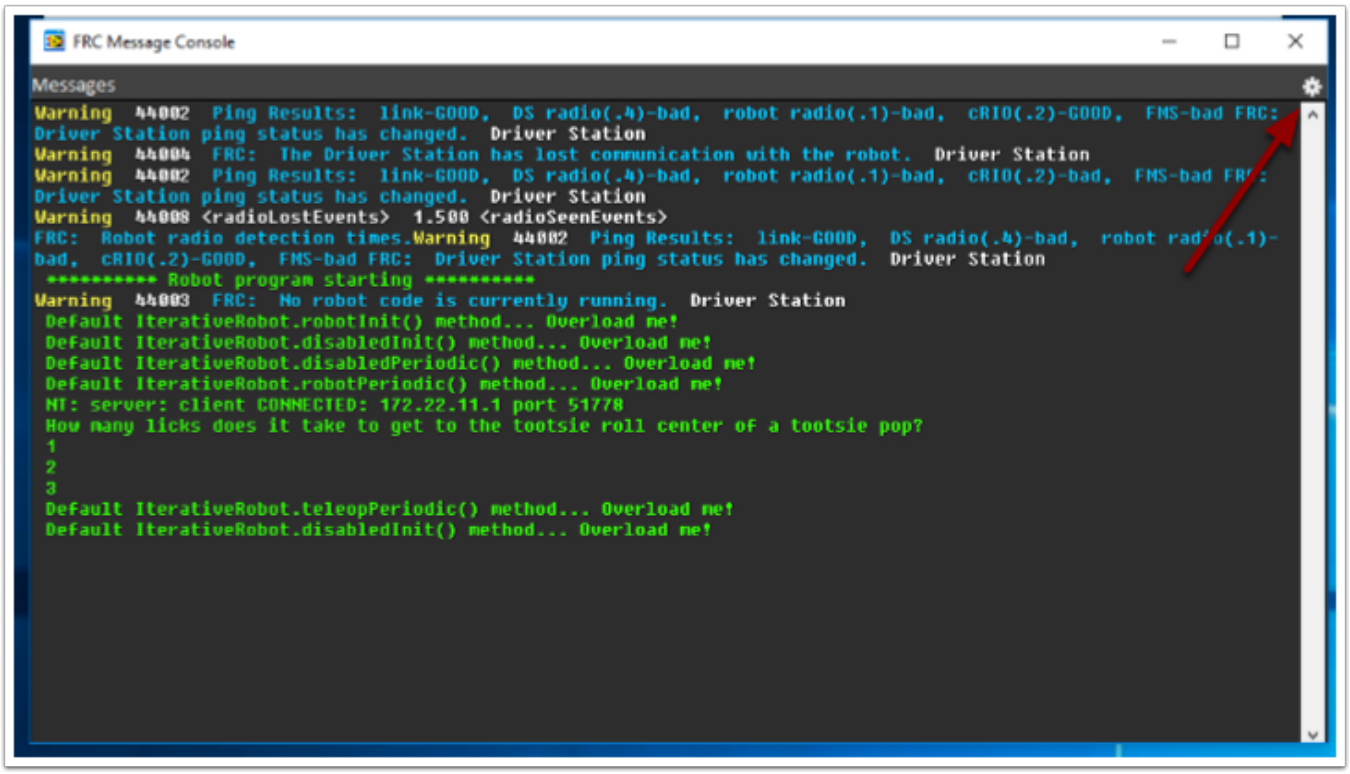
Console Viewer

Opening the Console Viewer



To open Console Viewer first open the FRC Driver Station. Then click on the gear at the top of the message viewer window (1) and select "View Console".

Console Viewer Window

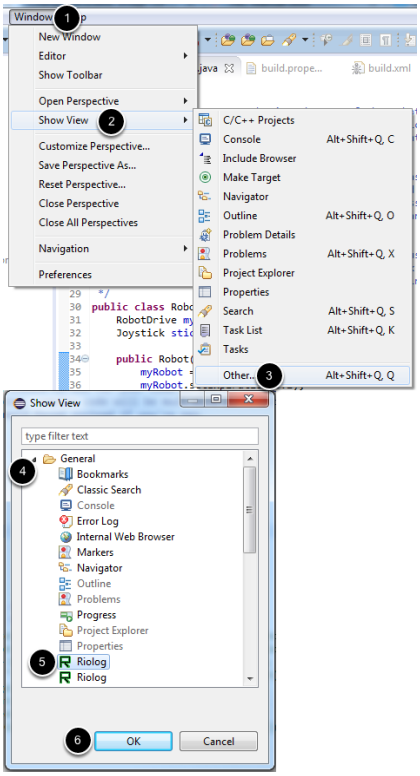


The Console Viewer window displays the output from our robot program in green. The gear in the top right can clear the window and set the level of messages displayed.

Riolog Eclipse Plugin

The Riolog plugin is an Eclipse plugin that can be used to view the NetConsole output in Eclipse (credit: Manuel Stoeckl, FRC1511).

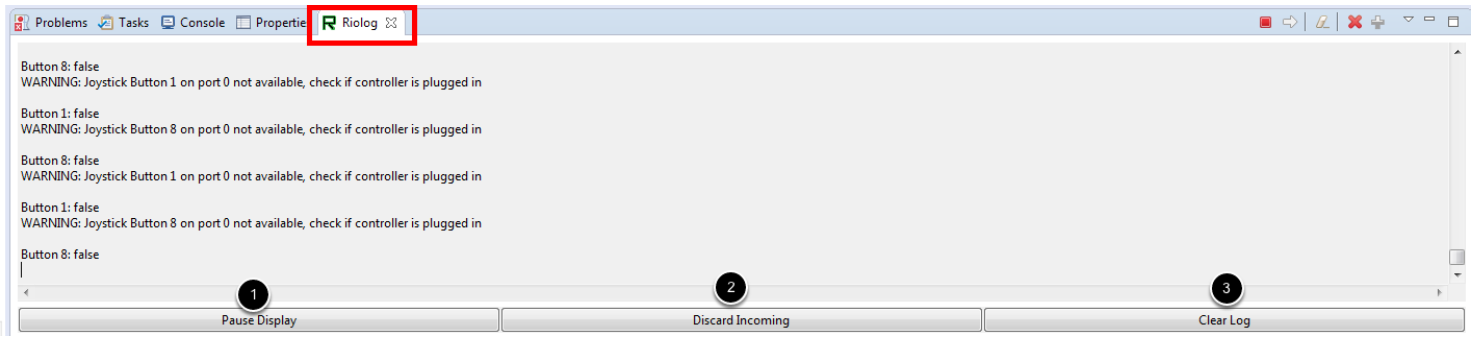
Opening the Riolog plugin



The first time you want to view the Riolog window in Eclipse, you will need to open the View. After doing this once, it should persist across restarts of Eclipse. To open the Riolog view:

1. Click Window
2. Hover over Show View
3. Click Other
4. Click the arrow to expand General
5. Click Riolog
6. Click OK

Riolog Window



The Riolog plugin window should appear in the bottom pane. To view the output at any time, simply click on the Riolog tab. Riolog contains a number of controls for manipulating the console. These controls can be accessed via the icon pairs in the top right or via multi-function buttons on the bottom of the window:

1. Pause/Resume Display - This will pause/resume the display. In the background, the new packets will still be received and will be displayed when the resume button is clicked.
2. Discard/Accept Incoming - This will toggle whether to accept new packets. When packets are being discarded the display will be paused and all packets received will be discarded. Clicking the button again will resume receiving packets.
3. Clear Log - This will clear the current contents of the display.

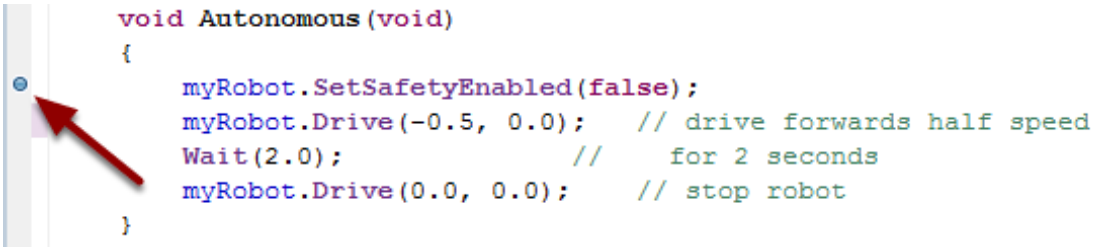
Troubleshooting

The Riolog plugin requires Java 8. If you are using Java 7, the Riolog plugin will not appear in your Show View window.

Debugging a robot program

A debugger is used to control program flow and monitor variables in order to assist in debugging a program. This section will describe how to set up a debug session for a C++ FRC robot program.

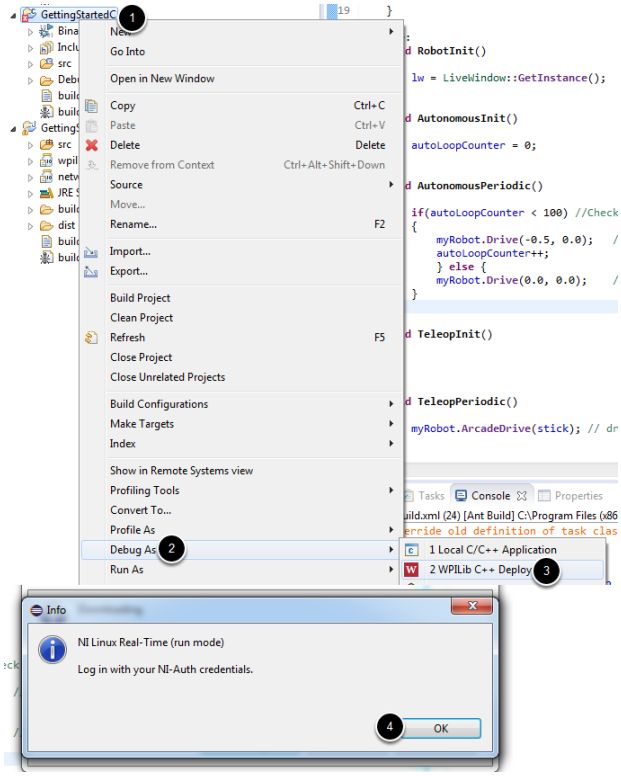
Set a Breakpoint



Clicking the “Debug” button does several things. It changes the Workbench to the Debug Perspective which has the views Debug, Breakpoints, and Variables along the right side of the window. It starts the robot task and pauses it at at the first breakpoint hit.

Double-click in the left margin of the source code window to set a breakpoint in your user program: A small blue circle indicates the breakpoint has been set on the corresponding line. (You can see all your breakpoints in the Workbench’s Breakpoints view.)

Launching Debug

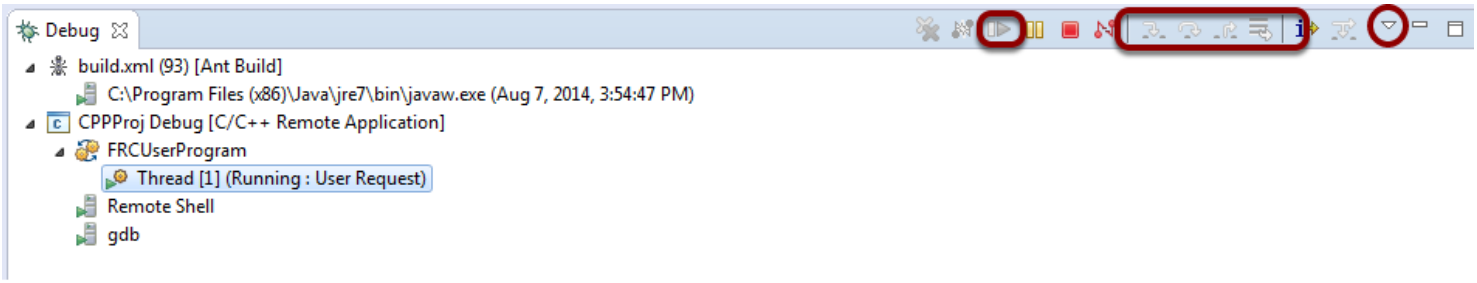


To launch the program in debug mode:

1. Right click on the project
2. Hover over Debug As
3. Select WPILib Deploy
4. When the window appears asking you to log in with your NI-Auth credentials, click ok.

If you receive a prompt about changing perspectives, click Ok.

The Debug window



The Debug window shows all processes and threads spawned by your code running on the roboRIO. Select the stack frame to see the current instruction pointer and source code (if available) for the selected process. When your breakpoint is reached, make sure your program is selected in the task list and your source code is displayed with a program pointer. You can continue stepping through your code using “Resume,” “Step Into,” “Step Over,” and “Step Return” buttons: If you see assembly code instead of C++ code displayed, it’s because you’ve stepped down into library code where the source is not available to the debugger. “Step Return” will bring you back up a level.

If the debug toolbar (resume, terminate and step controls) is not visible, click the down arrow at the top right of the debug window and select the **Show Debug Toolbar** option

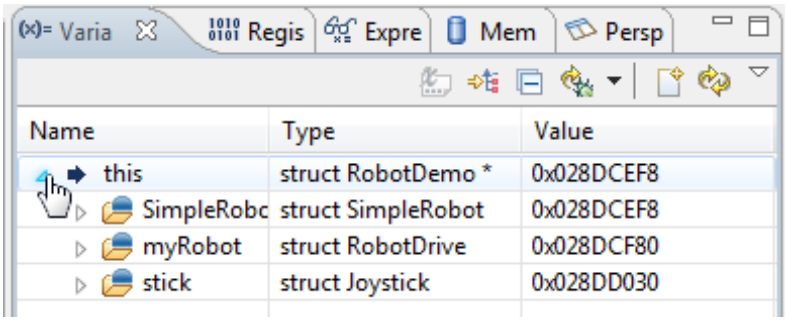
Click the “Resume” button (the green arrow) to resume program execution up to the first breakpoint. Note that if the breakpoint you have placed is inside one of the Teleoperated or Autonomous methods, the robot will have to be set to the appropriate mode and enabled with the Driver Station in order to reach the breakpoint.

Your Breakpoint

```
void Autonomous(void)
{
    myRobot.SetSafetyEnabled(false);
    myRobot.Drive(-0.5, 0.0);    // drive forwards half speed
    Wait(2.0);                   //    for 2 seconds
    myRobot.Drive(0.0, 0.0);     // stop robot
}
```

The program will start running and then pause at the breakpoint. From here you can use the three panes on the left of the screen (Debug, Breakpoints and Variables) to monitor and control the execution of your program.

The Variables Tab

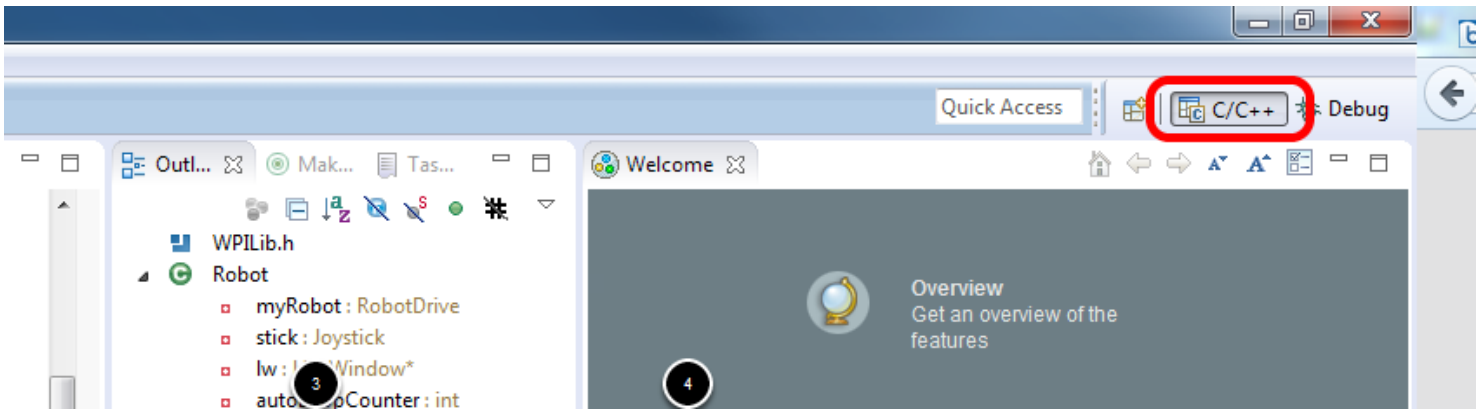


The Variables view shows the current values of variables. To see a variable that is not displayed, select the “Expressions” tab and enter the variable name. This will show the variable’s value if it’s in scope. You may also want to click on the arrows next to a variable to expand the tree and show it's members. For example, expanding our Robot Demo variable ("this") shows our "myRobot" and "stick" variables.

Stopping Debugging

To stop debugging, you can disconnect or terminate the process. Disconnecting detaches the debugger but leaves the process running in its current state. Terminating the process kills it on the target.

Exiting Debug Perspective Debugging



After you have terminated the Debug session, to either return to editing code, or to debug again, you need to return to the C/C++ Perspective. If the C++ perspective is still open you may do this by clicking the icon with the C in the top right corner of the screen or pressing Ctrl+F8. If you do not see the icon there, select Window->Open Perspective->Other->C/C++ and click OK.

FRC C++ Basics

C++ conventions for objects, methods, and variables

Every sensor, actuator and operator interface component is an object in either C++ or Java programs. To use one of these you must create an instance of it using the class name. Any references to the objects such as reading values, setting values, or setting parameters is done through the reference. There are a number of utility objects in WPILib such as the RobotDrive and Compressor that don't represent a single sensor or actuator, but a larger subsystem.

Another convention used throughout the library is the case of the method names. In **C++ all methods start with an upper case letter**, then are camel case (intermediate words capitalized). In **Java all methods start with lower case letters** then camel case for the remainder of the name.

Creating objects that are connected to the roboRIO in C++

C++

```
Gyro *headingGyro = new AnalogGyro(1); // Creates a Gyro object connected to channel 1
float heading  = headingGyro->GetAngle(); // Gets the current heading from the Gyro and
stores it in the variable heading
```

Generally all the objects in WPILib that connect to roboRIO have one argument in the constructor when created where you specify the channel or port number it is connected to. The above example illustrate the conventions used in WPILib for both C++ and Java.

Creating operator interface objects

C++

```
Joystick *stick = new Joystick(1); // Creates a Joystick object connected to USB on port 1
of the driverstation

double speed = stick->GetX(); // Gets the current X axis value of the joystick nd stores
```

```
it in the variable "speed"
```

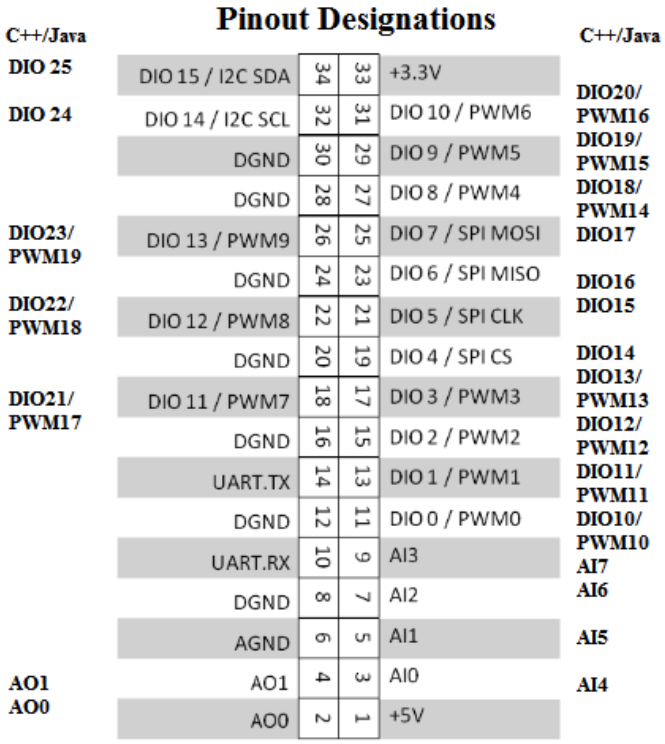
Generally objects connected to the Driver station PC via USB take a single argument indicating the USB port they are connected to. A single Joystick class is provided which should provide the functionality needed to interface with any joystick or gamepad which works with the FRC Driver Station.

Class, method, and variable naming

Type of name	Naming rules	Examples
Class name	Initial upper case letter then camel case (mixed upper/lower case) except acronyms which are all upper case	`Victor`, `SimpleRobot`, `PWM`
Method name	Initial upper case letter then camel case	`StartCompetition`, `Autonomous`, `GetAngle`
Member variable	"`m_`" followed by the member variable name starting with a lower case letter then camel case	`m_deleteSpeedControllers`, `m_sensitivity`
Local variable	Initial lower case	`targetAngle`
Macro	All upper case with _ between words. **Note:** It's better to use const values and inline functions than macros.	`DISALLOW_COPY_AND_ASSIGN`

Names in WPILib follow the conventions shown in the table above. It makes it very easy to determine what the scope and use of a variable is just by looking a the name and is a convention that is used throughout WPILib.

MXP IO Numbering



In C++ and Java the numbering for the MXP IO is a continuation of the numbering from the headers, meaning MXP DIO 0 is DIO 10, MXP DIO 1 is DIO 11 and so on. This applies to DIO, PWM and Analog Input on the MXP. The I2C and SPI buses have enumerations used to indicate which port you are using.

Namespacing in C++

Content coming soon

Pointers and addresses

Creating object instances

Location	Create object	Use the object	When the object is deleted
Variable declared inside an object, function, or block	`Victor leftMotor(3);`	`leftMotor.Set(1.0);`	Object is automatically deleted when the enclosing block exits or the enclosing object is deleted
Global declared outside of any enclosing objects or functions; or a static variable	`Victor leftMotor(3);`	`leftMotor.Set(1.0);`	Object is not deleted until the program exits
Pointer to object	`Victor *leftMotor = new Victor(3);`	`leftMotor->Set(1.0);`	Object must be explicitly deleted using the C++ `delete` operator.

There are several ways of creating object instances in C++. These ways differ in how the object should be referenced and deleted. This table shows the rules.

Pointers vs References

C++

```
Joystick stick1(1);          // this is an instance of a Joystick object stick1
stick1.GetX(); \              // access the instance using the dot (.) operator
bot->ArcadeDrive(stick1); \  //you can pass the instance to a method by reference
                             // \    ...   ArcadeDrive(Joystick& j);  ...

Joystick *stick2;            // a pointer to an uncreated Joystick object
stick2 = new Joystick(1);    // creates the instance of the Joystick object
stick2->GetX();              // access the instance with the arrow (->) operator
bot->ArcadeDrive(stick2);    // you can pass the instance by the pointer (notice, no &)
                             // \    ...   ArcadeDrive(Joystick* j);  ...
delete stick2;               // delete it when your done with it
```

In the first group of statements (1) a Joystick object is created as a reference - stick1 refers to the object itself. In the second group of statements, stick2 is a pointer to a Joystick object.

ArcadeDrive()in WPILib takes advantage of a C++ feature called *function overloading*. This allows it to have two methods with the same name that differ by argument lists. The one ArcadeDrive(Joystick &j) takes the parameter j as a reference to a Joystick instance. You supply a Joystick and the compiler automatically passes a reference to that instance. The other ArcadeDrive(Joystick *j) takes the parameter j as a pointer to a Joystick instance. You supply a pointer to a Joystick instance. The cool thing is that the compiler figures out which ArcadeDrive to call. The library is built this way to support both the pointer style and the reference style.

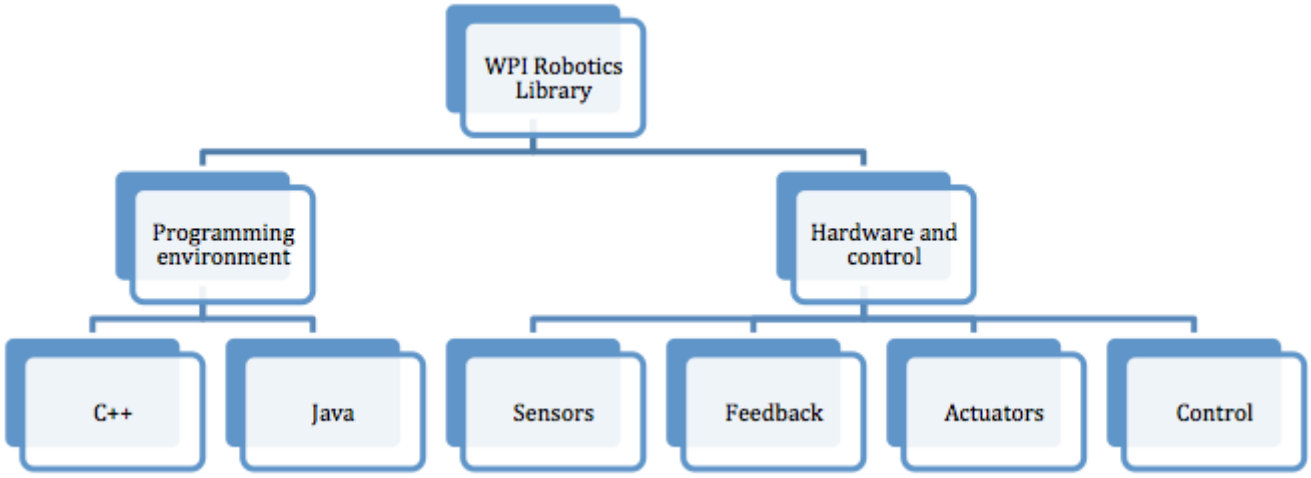
If you had non-overloaded functions Ref(Joystick &j) and Ptr(Joystick *j), you could still call them if you use the right C++ operators: Ref(*stick2) and Ptr(&stick1). At run time, references and pointers are both passed as addresses to the instance. The difference is the source code syntax and details like allocation and deletion.

Basic WPILib Programming features

What is WPILib

The WPI Robotics library (WPILib) is a set of software classes that interfaces with the hardware and software in your FRC robot's control system. There are classes to handle sensors, motor speed controllers, the driver station, and a number of other utility functions such as timing and field management. In addition, WPILib supports many commonly used sensors that are not in the kit, such as ultrasonic rangefinders.

What's included in the library



There are three versions of the library, one for each supported language. This document specifically deals with the text-based languages, C++ and Java. There is considerable effort to keep the APIs for Java and C++ very similar with class names and method names being the same. There are some differences that reflect language differences such as pointers vs. references, name case conventions, and some minor differences in functionality. These languages were chosen because they represent a good level of abstraction, are used heavily in industry, and are often taught in High School and college courses. The WPI Robotics Library is designed for maximum extensibility and software reuse with these languages.

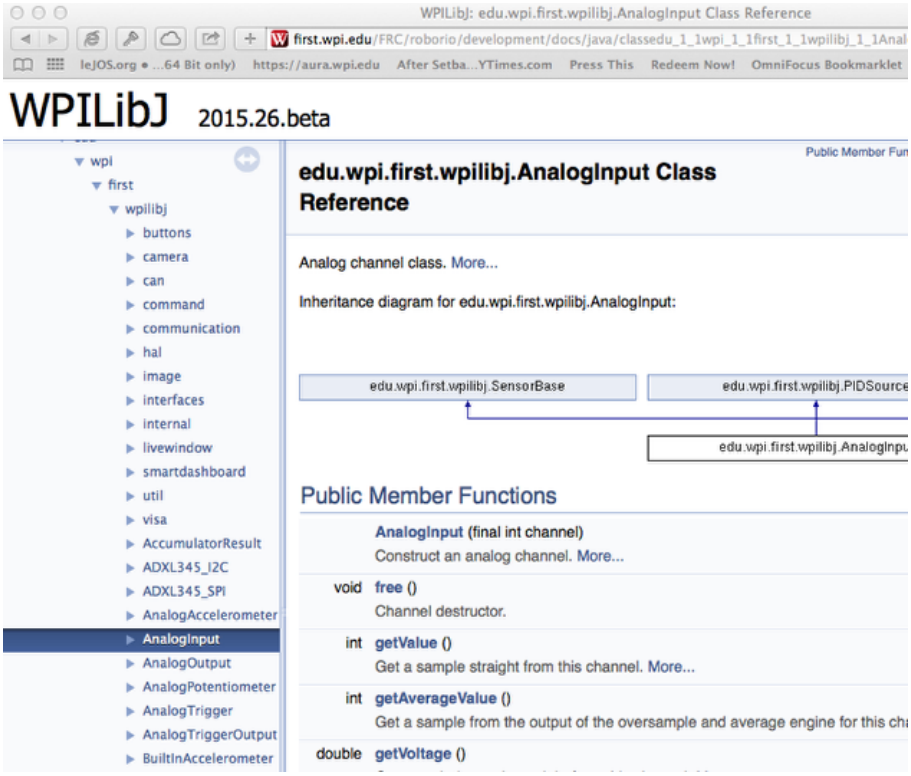
WPILib has a generalized set of features, such as general-purpose counters, to provide support for custom hardware and devices. The FPGA hardware also allows for interrupt processing to be dispatched at the task level, instead of as kernel interrupt handlers, reducing the complexity of many common real-time issues.

Fundamentally, C++ offers the highest performance possible for your robot programs (this only comes into effect with very tight timing or very CPU intensive processing). Java on the other hand

has acceptable performance and includes extensive run-time checking of your program to make it much easier to debug and detect errors. Those with extensive programming experience can probably make their own choices, and beginning might do better with Java to take advantage of the ease of use.

There is a detailed list of the differences between C++ and Java on Wikipedia available here. Below is a summary of the differences that will most likely effect robot programs created with WPILib.

WPILib Documentation



Documentation for WPILib APIs for C++ and Java can be found here:

```
C++: http://first.wpi.edu/FRC/roborio/release/docs/cpp
```

```
Java: http://first.wpi.edu/FRC/roborio/release/docs/java
```

with separate sections for C++ and Java documentation. Both languages are documented similarly with a tree showing all the classes, methods, and public constants. This will automatically update as new versions of the library are released.

WPILib Source Code

Source code for WPILib is not currently bundled with the Eclipse Plugins. To browse the source code for WPILib online or for information about checking out the repository using GIT, see the "allwpilib" project on GitHub: https://github.com/wpilibsuite/allwpilib

Java programming with WPILib

Java

```
public void autonomousInit() {
     isAuto = true;
     CommandBase.shooter.zeroRPMOffsets();
 \    CommandBase.turret.zeroAngleOffsets();
     // instantiate the command used for the autonomous period
     autonomousCommand = (Command) (OI.getInstance().auton.getSelected());
 \    // schedule the autonomous command (example)
     autonomousCommand.start();
}
```

- Java objects must be allocated manually, but are freed automatically when no references remain.
- References to objects instead of pointers are used. All objects must be allocated with the new operator and are referenced using the dot (.) operator (e.g. gyro.getAngle()).
- Header files are not necessary and references are automatically resolved as the program is built.
- Only single inheritance is supported, but interfaces are added to Java to get most of the benefits that multiple inheritance provides.
- Checks for array subscripts out of bounds, uninitialized references to objects and other runtime errors that might occur in program development.
- Compiles to byte code for a virtual machine, and must be interpreted.

C++ programming with WPILib

C++

```
void Claw::Open() {
     victor->Set(1);
}

void Claw::Close() {
     victor->Set(-1);
}

void Claw::Stop() {
     victor->Set(0);
}
```

- 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 ARM processor in the roboRIO

Choosing a Base Class

IterativeRobot

C++

```
RobotTemplate::RobotTemplate()
{
}

void RobotTemplate::RobotInit()
{
}

void RobotTemplate::AutonomousInit()
{
}

void RobotTemplate::AutonomousPeriodic()
{
}
```

Java

```
public class RobotTemplate extends IterativeRobot {

     public void robotInit() {

     }

     public void autonomousInit() {

     }
```

```
     public void autonomousPeriodic() {

 \     }
```

The Iterative Robot base class assists with the most common code structure by handling the state transitions and looping in the base class instead of in the robot code. For each state (autonomous, teleop, disabled, test) there are two methods that are called:

- Init methods - The init method for a given state is called each time the corresponding state is entered (for example, a transition from disabled to teleop will call teleopInit()). Any initialization code or resetting of variables between modes should be placed here.
- Periodic methods - The periodic method for a given state is called each time the robot receives a Driver Station packet in the corresponding state, approximately every 20ms. This means that all of the code placed in each periodic method should finish executing in 20ms or less. The idea is to put code here that gets values from the driver station and updates the motors. You can read the joysticks and other Driver Station inputs more often, but you'll only get the previous value until a new update is received. By synchronizing with the received updates your program will put less of a load on the roboRIO CPU leaving more time for other tasks such as camera processing.

SampleRobot

C++

```
RobotTemplate::RobotTemplate()
{
}

//This function is called once each time the robot enters autonomous mode.
void RobotTemplate::Autonomous()
{
 \     while (IsAutonomous() && IsEnable())
      {
           // Put code here
 \          Wait(0.05);
      }
```

```
}

// This function is called once each time the robot enters teleop mode.
void RobotTemplate::OperatorControl() {
     while (isOperatorControl() && isEnabled())
     {
          // Put code here
          Wait(0.05);
 \    }
}
```

Java

```
public class RobotTemplate extends SampleRobot {

     //This function is called once each time the robot enters autonomous mode.
     public void autonomous() {
          // Put code here
          Timer.delay(0.05);
     }

     // This function is called once each time the robot enters teleop mode.
     public void operatorControl() {
         while(isOperatorControl() && isEnabled()) {
               //Put code here
               //Timer.delay(0.05);
         }
 \    }
}
```

```
public class RobotTemplate extends SampleRobot {

    /**
```

```
     * This function is called once each time the robot enters autonomous mode.
     */
    public void autonomous() {
        while (isAutonomous() && isEnable()) {
            // Put code here
            Timer.delay(0.05);
        }
    }

    /**
     * This function is called once each time the robot enters teleop mode.
     */
    public void operatorControl() {
        while (isOperatorControl() && isEnabled()) {
            // Put code here
            Timer.delay(0.05);
        }
    }
}
```

The SampleRobot class is the simplest template as most of the state flow is directly visible in your program and not hidden in the WPILib code. The downside is that implementing this state flow incorrectly can lead to problems. Your robot program overrides the operatorControl() and autonomous() methods that are called by the base at the appropriate time. Note that these methods are called only called once each time the robot enters the appropriate mode and are not automatically terminated. Your code in the operatorControl method must contain a loop that checks the robot mode in order to keep running and taking new input from the Driver Station. The autonomous code shown uses a similar loop.

It is recommended for beginners to choose the Iterative Template or Command Based robot. SampleRobot can be used by advanced users wishing to have more control over the flow of their program.

Command-Based Robot

While not strictly a base class, the Command-based robot model is a method for creating larger programs, more easily, that are easier to extend. There is built in support with a number of classes to make it easy to design your robot, build subsystems, and control interactions between the robot and the operator interface. In addition it provides a simple mechanism for writing autonomous

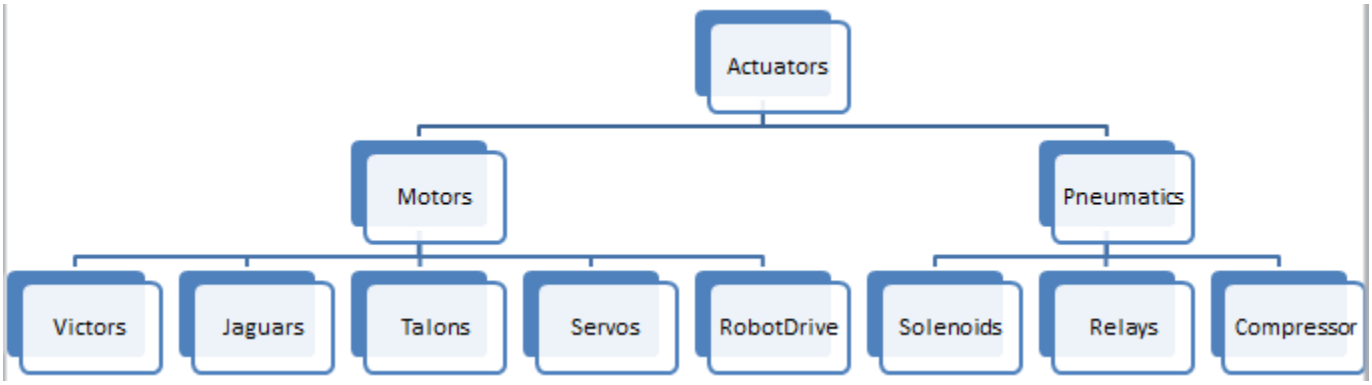
programs. The command-based model is described in detail in the Command-Based Programming section of the C++ and Java manuals.

Using actuators (motors, servos, and relays)

Actuator Overview

This section discusses the control of motors and pneumatics through speed controllers, relays, and WPILib methods.

Types of actuators



The chart shown above outlines the types of actuators which can be controlled through WPILib. The articles in this section will cover each of these types of actuators and the WPILib methods and classes that control them.

Driving motors with speed controller objects (Victors, Talons and Jaguars)

The WPI Robotics library has extensive support for motor control. There are a number of classes that represent different types of speed controllers and servos. The WPI Robotics Library currently supports two classes of speed controllers, PWM based motor controllers (Jaguars, Victors and Talons) and CAN based motor controllers (Jaguar). WPILIb also contains a composite class called RobotDrive which allows you to control multiple motors with a single object. This article will cover the details of PWM motor controllers, CAN controllers and RobotDrive will be covered in separate articles.

PWM Controllers, brief theory of operation

The acronym PWM stands for Pulse Width Modulation. For the Victor, Talon and Jaguar (using the PWM input) motor controllers, PWM can refer to both the input signal and the method the controller uses to control motor speed. To control the speed of the motor the controller must vary the perceived input voltage of the motor. To do this the controller switches the full input voltage on and off very quickly, varying the amount of time it is on based on the control signal. Because of the mechanical and electrical time constants of the types of motors used in FRC this rapid switching produces an effect equivalent to that of applying a fixed lower voltage (50% switching produces the same effect as applying ~6V).

The PWM signal the controllers use for an input is a little bit different. Even at the bounds of the signal range (max forward or max reverse) the signal never approaches a duty cycle of 0% or 100%. Instead the controllers use a signal with a period of either 5ms or 10ms and a midpoint pulse width of 1.5ms. The Talon and Victor controllers use typical hobby RC controller timing of 1ms to 2ms and the Jaguar uses an expanded timing of ~.7ms to ~2.3ms.

Raw vs Scaled output values

In general, all of the motor controller classes in WPILib are set up to take a scaled -1.0 to 1.0 value as the output to an actuator. The PWM module in the FPGA is capable of generating PWM signals with periods of 5, 10 or 20ms and can vary the pulse width in 2000 steps of ~.001ms each around the midpoint (1000 steps in each direction around the midpoint). The raw values sent to this module are in this 0-2000 range with 0 being a special case which holds the signal low (disabled). The class for each motor controller contains information about what the typical bound values (min, max and each side of the deadband) are as well as the typical midpoint. WPILib can then use these values to map the scaled value into the proper range for the motor controller. This allows for

the code to switch seamlessly between different types of controllers and abstracts out the details of the specific signaling.

Calibrating Speed Controllers

So if WPILib handles all this scaling, why would you ever need to calibrate your speed controller? The values WPILib uses for scaling are approximate based on measurement of a number of samples of each controller type. Due to a variety of factors, the timing of an individual speed controller may vary slightly. In order to definitively eliminate "humming" (midpoint signal interpreted as slight movement in one direction) and drive the controller all the way to each extreme, calibrating the controllers is still recommended. In general, the calibration procedure for each controller involves putting the controller into calibration mode then driving the input signal to each extreme, then back to the midpoint. Precise details for each controller can be found in the User Guides: Talon, Jaguar, Victor, VictorSP, TalonSRX

PWM and Safe PWM Classes

PWM

The PWM class is the base class for devices that operate on PWM signals and is the connection to the PWM signal generation hardware in the roboRIO. It is not intended to be used directly on a speed controller or servo. The PWM class has shared code for Victor, Jaguar, Talon, and Servo subclasses that set the update rate, deadband elimination, and profile shaping of the output.

Safe PWM

The SafePWM class is a subclass of PWM that implements the RobotSafety interface and adds watchdog capability to each speed controller object. The RobotSafety interface will be discussed further in another article.

Constructing a Speed Controller object

C++

```
Jaguar *exampleJaguar = new Jaguar(0);
Talon *exampleTalon = new Talon(1);
TalonSRX *exampleTalonSRX = new TalonSRX(2);
Victor *exampleVictor = new Victor(11);
VictorSP *exampleVictorSP = new VictorSP(12);
```

Java

```
Jaguar exampleJaguar = new Jaguar(0);
```

```
Talon exampleTalon = new Talon(1);
TalonSRX exampleTalonSRX = new TalonSRX(2);
Victor exampleVictor = new Victor(11);
VictorSP exampleVictorSP = new VictorSP(12);
```

Speed controller objects are constructed by passing in a channel. No other parameters are passed into the constructor.

Setting parameters

C++

```
Jaguar *exampleJaguar = new Jaguar(0);
exampleJaguar->EnableDeadbandElimination(true);
```

Java

```
Jaguar exampleJaguar = new Jaguar(0);
exampleJaguar.enableDeadbandElimination(true);
```

All of the settable parameters of the motor controllers inherit from the underlying PWM class and are thus identical across the controllers. The code above shows only a single controller type (Jaguar) as an example. There are a number of settable parameters of a PWM object, but only one is recommended for robot code to modify directly:

- Deadband Elimination - Set to true to have the scaling algorithms eliminate the controller deadband. Set to false (default) to leave the controller deadband intact.

Setting Speed

C++

```
Jaguar *exampleJaguar = new Jaguar(0);
exampleJaguar->Set(0.7);
```

Java

```
Jaguar exampleJaguar = new Jaguar(0);
exampleJaguar.set(0.7);
```

As noted previously, speed controller objects take a single speed parameter varying from -1.0 (full reverse) to +1.0 (full forward).

Getting your robot to drive with the RobotDrive class

WPILib provides a RobotDrive object that handles most cases of driving the robot either in autonomous or teleop modes. It is created with either two or four speed controller objects. There are methods to drive with either Tank, Arcade, or Mecanum modes either programmatically or directly from Joysticks.

Note: the examples illustrated in this section are generally correct but have not all been tested on actual robots.

Creating a RobotDrive object with Talon speed controllers

C++
```
RobotDrive *drive;
drive = new RobotDrive(0,1,2,3) //4 motor drive
drive = new RobotDrive(0,1) //2 motor drive
```

Java
```
RobotDrive drive = new RobotDrive(0,1,2,3) //4 motor drive
RobotDrive drive = new RobotDrive(0,1) //2 motor drive
```

Create the RobotDrive object specifying either two or four motor ports. By default the RobotDrive constructor will create Talon class instances attached to each of those ports.

Inverting the sense of some of the motors

C++
```
drive->SetInvertedMotor(kFrontLeftMotor, true);
```

Java
```
drive.setInvertedMotor(MotorType.kFrontLeft, true);
```

It might turn out that some of the motors used in your RobotDrive object turn in the opposite direction. This often happens depending on the gearing of the motor and the rest of the drive train. If this happens, you can use the SetInvertedMotor() method, as shown, to reverse a particular motor.

Using other types of speed controllers

C++

```
class Robot: public IterativeRobot
{
private:
        class RobotDrive *myDrive;
        class Victor *frontLeft, *frontRight, *rearLeft, *rearRight;

        void AutonomousInit()
        {

        }

        void AutonomousPeriodic()
        {

        }

        void TeleopInit()
        {
                frontLeft = new Victor(1);
                frontRight = new Victor(2);
                rearLeft = new Victor(3);
                rearRight = new Victor(4);
                myDrive = new RobotDrive(frontLeft,frontRight,rearLeft,rearRight);
        }
```

Java

```
public class RobotTemplate extends SampleRobot {

        RobotDrive myDrive;
        Victor frontLeft, frontRight, rearLeft, rearRight;
```

```
    public void robotInit() {
            frontLeft = new Victor(1);
            frontRight = new Victor(2);
            rearLeft = new Victor (3);
            rearRight = new Victor (4);
            myDrive = new RobotDrive(frontLeft, frontRight, rearLeft, rearRight);
    }
```

You can use RobotDrive with other types of speed controllers as well. In this case you must create the speed controller objects manually and pass the references or pointers to the RobotDrive constructor.

Tank driving with two joysticks

C++

```
class Robot: public IterativeRobot
{
private:
        RobotDrive *myDrive;
        Joystick *left, *right;

        void TeleopInit()
        {
                myDrive = new RobotDrive(1,2,3,4);
                left = new Joystick(1);
                right = new Joystick(2);
        }

        void operatorControl() {
                while (IsOperatorControl() && IsEnabled()) {
                        myDrive->TankDrive(left, right);
                        Wait(0.01);
```

```
                }
        }
```

Java

```
public class RobotTemplate extends SampleRobot {

        RobotDrive myDrive;
        Joystick left, right;

    public void robotInit() {
            myDrive = new RobotDrive(1,2,3,4);
            left = new Joystick(1);
            right = new Joystick(2);
    }

    public void autonomousPeriodic() {

    }

    public void operatorControl() {
            while (isOperatorControl() && isEnabled()) {
                    myDrive.tankDrive(left,  right);;
                    Timer.delay(0.01);
            }
    }
```

In this example a RobotDrive object is created with 4 default Talon speed controllers. In the operatorControl method the RobotDrive instance tankDrive method is called and it will select the Y-axis of each of the joysticks by default. There are other versions of the tankDrive method that can be used to use alternate axis or just numeric values.

Arcade driving with a single joystick

C++

```
class Robot: public IterativeRobot
{
private:
        RobotDrive *myDrive;
        Joystick *driveStick;

        void TeleopInit()
        {
                myDrive = new RobotDrive(1,2,3,4);
                driveStick = new Joystick(1);
        }

        void operatorControl() {
                while (IsOperatorControl() && IsEnabled()) {
                        myDrive->ArcadeDrive(driveStick);
                        Wait(0.01);
                }
        }
```

Java

```
public class RobotTemplate extends SampleRobot {

        RobotDrive myDrive;
        Joystick driveStick;

    public void robotInit() {
            myDrive = new RobotDrive(1,2,3,4);
            driveStick = new Joystick(1);
    }

    public void autonomousPeriodic() {
```

```
    }

    public void operatorControl() {
            while (isOperatorControl() && isEnabled()) {
                    myDrive.arcadeDrive(driveStick);
                    Timer.delay(0.01);
            }
    }
```

Similar to the example above a single joystick can be used to do single-joystick driving (called arcade). In this case, the X-axis is selected by default for the turn axis and the Y-axis is selected for the speed axis.

Autonomous driving using the RobotDrive object

C++

```
class Robot: public IterativeRobot
{
private:
        RobotDrive *myDrive;
        Joystick *driveStick;
        Gyro *gyro;

        void RobotInit()
        {
                myDrive = new RobotDrive(1,2,3,4);
                gyro = new AnalogGyro(1);
        }

        void AutonomousPeriodic()
        {
                while(IsAutonomous() && IsEnabled()){
                        float angle = gyro->GetAngle();
```

```
                        float Kp = 0.03;
                        myDrive->ArcadeDrive(-1.0, -angle * Kp);
                        Wait(0.01);
                }
        }
```

Java

```
public class RobotTemplate extends SampleRobot {

        RobotDrive myDrive;
        Joystick driveStick;
        Gyro gyro;
        static final double Kp = 0.03;

    public void robotInit() {
            myDrive = new RobotDrive(1,2,3,4);
            gyro = new AnalogGyro(1);
    }

    public void autonomousPeriodic() {
            while (isAutonomous() && isEnabled()) {
                    double angle = gyro.getAngle();
                    myDrive.arcadeDrive(-1.0, -angle * Kp);
                    Timer.delay(0.01);
            }
    }
```

The RobotDrive object also has a number of features that makes it ideally suited for autonomous control. This example illustrates using a gyro for driving in a straight line (the current heading) using the arcade method for steering. While the robot continues to drive in a straight line the gyro headings are roughly zero. As the robot veers off in one direction or the other, the gyro headings vary either positive or negative. This is very convenient since the arcade method turn parameter is

also either positive or negative. The magnitude of the gyro headings sets the rate of the turn, that is more off zero the gyro heading is, the faster the robot should turn to correct.

This is a perfect use of proportional control where the rate of turn is proportional to the gyro heading (being off zero). The heading is in values of degrees and can easily get pretty far off, possibly as much as 10 degrees as the robot drives. But the values for the turn in the arcade method are from zero to one. To solve this problem, the heading is scaled by a constant to get it in the range required by the turn parameter of the arcade method. This parameter is called the proportional gain, often written as kP.

In this particular example the robot is designed such that negative speed values go forward. Also, not that the the angle from the gyro is written as "-angle". This is because this particular robot turns in the opposite direction from the gyro corrections and has to be negated to correct that. Your robot might be different in both cases depending on gearing and motor connections.

Mecanum driving

C++

```
class RobotTemplate: public SampleRobot {

        RobotDrive *myDrive;
        Joystick *moveStick, *rotateStick;

        public void RobotInit() {
                myDrive = new RobotDrive(0, 1, 2, 3);
                moveStick = new Joystick(0);
                rotateStick = new Joystick(1);
        }

        public void Autonomous() {
        }

        public void OperatorControl() {
                while (IsOperatorControl() && IsEnabled()) {
                        myDrive->MecanumDrive_Cartesian(moveStick->GetY(), moveStick-
>GetX(), rotateStick->GetX(), 0);
                        Wait(0.02);
                }
```

```
        }
}
```

Java

```
public class RobotTemplate extends SampleRobot {

        RobotDrive myDrive;
        Joystick moveStick, rotateStick;

        public void robotInit() {
                myDrive = new RobotDrive(0, 1, 2, 3);
                moveStick = new Joystick(0);
                rotateStick = new Joystick(1);
        }

        public void autonomous() {
        }

        public void operatorControl() {
                while (isOperatorControl() && isEnabled()) {
                        myDrive.mecanumDrive_Cartesian(moveStick.getY(), moveStick.getX(),
rotateStick.getX(), 0);
                        Timer.delay(0.02);
                }
        }
}
```

The RobotDrive can also handle Mecanum driving. That is using Mecanum wheels on the chassis to enable the robot to drive in any direction without first turning. This is sometimes called Holonomic driving.

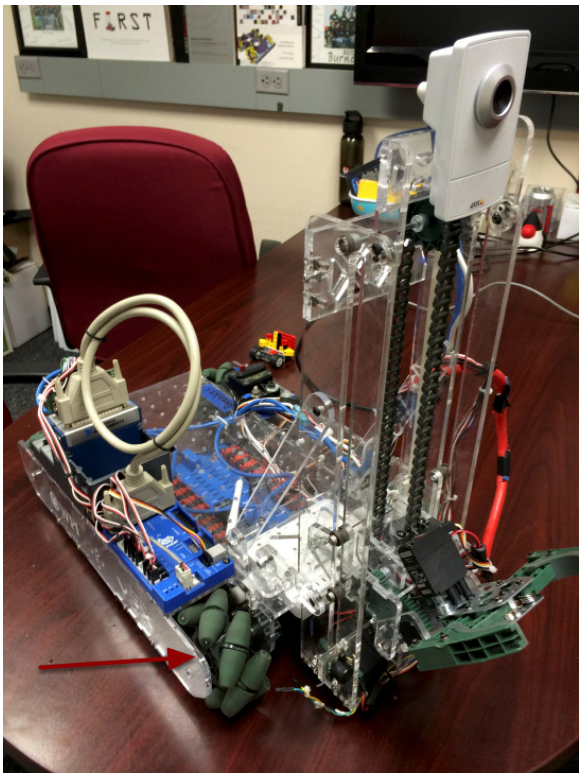
In this example there are two joysticks controlling the robot. moveStick supplies the direction vector for the robot, that is which way it should move irrespective of the heading. rotateStick supplies the rate of rotation using the x-axis of that joystick. If you push the moveStick full forward the robot will move forward (relative to the robot). At the same time, if you rotate the rotateStick, the robot will spin in the rotation direction with the rotation rate from the amount of twist, while the robot continues to move forward.

If you wan the movement commands to be relative to the field, rather than relative to the robot's current heading, you will need to pass in the robot's current angle in place of the 0 above. This angle is usually derived using a Gyro.

Driving a robot using Mecanum drive

Mecanum drive is a method of driving using specially designed wheels that allow the robot to drive in any direction without changing the orientation of the robot. A robot with a conventional drivetrain (all wheels pointing in the same direction) must turn in the direction it needs to drive. A mecanum robot can move in any direction without first turning and is called a holonomic drive.

Mecanum wheels



The wheels shown in this robot have rollers that cause the forces from driving to be applied at a 45 degree angle rather than straight forward as in the case of a conventional drive. You might guess that varying the speed of the wheels results in travel in any direction. You can look up how mecanum wheels work on various web sites on the internet.

Code for driving with mecanum wheels

```
C++
```

```
#include "WPILib.h"
/**
 * Simplest program to drive a robot with mecanum drive using a single Logitech
 * Extreme 3D Pro joystick and 4 drive motors connected as follows:
 *   - Digital Sidecar 1:
 *     - PWM 1 - Connected to front left drive motor
 *     - PWM 2 - Connected to rear left drive motor
 *     - PWM 3 - Connected to front right drive motor
 *     - PWM 4 - Connected to rear right drive motor
 */
class MecanumDefaultCode : public IterativeRobot
{
        RobotDrive *m_robotDrive;               // RobotDrive object using PWM 1-4 for
drive motors
        Joystick *m_driveStick;                      // Joystick object on USB port 1
(mecanum drive)public:
        /**
         * Constructor for this "MecanumDefaultCode" Class.
         */
        MecanumDefaultCode(void)
        {
                // Create a RobotDrive object using PWMS 1, 2, 3, and 4
                m_robotDrive = new RobotDrive(1, 2, 3, 4);
                // Define joystick being used at USB port #1 on the Drivers Station
                m_driveStick = new Joystick(1);
                // Twist is on Axis 3 for the Extreme 3D Pro
                m_driveStick->SetAxisChannel(Joystick::kTwistAxis, 3);
        }
        /**
         * Gets called once for each new packet from the DS.
         */
        void TeleopPeriodic(void)
        {
                m_robotDrive->MecanumDrive_Cartesian(m_driveStick->GetX(), m_driveStick-
>GetY(), m_driveStick->GetTwist());
        }
};
START_ROBOT_CLASS(MecanumDefaultCode);
```

Java

```
import edu.wpi.first.wpilibj.Joystick;
import edu.wpi.first.wpilibj.RobotDrive;
import edu.wpi.first.wpilibj.IterativeRobot;


/*
 * Simplest program to drive a robot with mecanum drive using a single Logitech
 * Extreme 3D Pro joystick and 4 drive motors connected as follows:
 *   - Digital Sidecar 1:
 *     - PWM 1 - Connected to front left drive motor
 *     - PWM 2 - Connected to rear left drive motor
 *     - PWM 3 - Connected to front right drive motor
 *     - PWM 4 - Connected to rear right drive motor
 */

public class MecanumDefaultCode extends IterativeRobot {
     //Create a robot drive object using PWMs 1, 2, 3 and 4
     RobotDrive m_robotDrive = new RobotDrive(1, 2, 3, 4);
     //Define joystick being used at USB port 1 on the Driver Station
     Joystick m_driveStick = new Joystick(1);

     public void teleopPeriodic() {
          m_robotDrive.mecanumDrive_Cartesian(m_driveStick.getX(), m_driveStick.getY(),
m_driveStick.getTwist(),0);
     }
}
```

Here's a sample program that shows the minimum code to drive using a single joystick and mecanum wheels. It uses the RobotDrive object that is available in both C++ and Java so even though this example is in C++ similar code will work in Java. The idea is to create the RobotDrive object with 4 PWM ports for the 4 speed controllers on the robot. The joystick XY position represents a direction vector that the robot should follow regardless of its orientation. The twist axis on the joystick represents the rate of rotation for the robot while it's driving.

Thanks to **FRC Team 2468** in Austin, TX for developing this example.

Updating the program for field-oriented driving

I would be remiss in not mentioning that there is a 4th parameter to the MecanumDrive_Cartesian() method that is the angle returned from a Gyro sensor. This will adjust the rotation value supplied, in this case, from the twist axis of the joystick to be relative to the field rather than relative to the robot. This is particularly useful with mecanum drive since, for the purposes of steering, the robot really has no front, back or sides. It can go in any direction. Adding the angle in degrees from a gyro object will cause the robot to move away from the drivers when the joystick is pushed forwards, and towards the drivers when it is pulled towards them - regardless of what direction the robot is facing!

The use of field-oriented driving makes often makes the robot much easier to drive, especially compared to a "robot-oriented" drive system where the controls are reversed when the robot is facing the drivers.

Just remember to get the gyro angle each time MecanumDrive_Caresian() is called.

Repeatable Low Power Movement - Controlling Servos with WPILib

Servo motors are a type of motor which integrates positional feedback into the motor in order to allow a single motor to perform repeatable, controllable movement, taking position as the input signal. WPILib provides the capability to control servos which match the common hobby input specification (PWM signal, 1.0ms-2.0ms pulse width)

Constructing a Servo object

C++

```
Servo *exampleServo = new Servo(1);
```

Java

```
Servo exampleServo = new Servo(1);
```

A servo object is constructed by passing a channel.

Setting Servo Values

C++

```
exampleServo->Set(.5);
exampleServo->SetAngle(75);
```

Java

```
exampleServo.set(.5);
exampleServo.setAngle(75);
```

There are two methods of setting servo values in WPILib:

- Scaled Value - Sets the servo position using a scaled 0 to 1.0 value. 0 corresponds to one extreme of the servo and 1.0 corresponds to the other
- Angle - Set the servo position by specifying the angle, in degrees. This method will work for servos with the same range as the Hitec HS-322HD servo (0 to 170 degrees). Any values passed to this method outside the specified range will be coerced to the boundary.

Using the motor safety feature

Motor Safety is a mechanism in WPILib that takes the concept of a watchdog and breaks it out into one watchdog (Motor Safety timer) for each individual actuator. Note that this protection mechanism is in addition to the System Watchdog which is controlled by the Network Communications code and the FPGA and will disable all actuator outputs if it does not receive a valid data packet for 125ms.

Motor Safety Purpose

The purpose of the Motor Safety mechanism is the same as the purpose of a watchdog timer, to disable mechanisms which may cause harm to themselves, people or property if the code locks up and does not properly update the actuator output. Motor Safety breaks this concept out on a per actuator basis so that you can appropriately determine where it is necessary and where it is not. Examples of mechanisms that should have motor safety enabled are systems like drive trains and arms. If these systems get latched on a particular value they could cause damage to their environment or themselves. An example of a mechanism that may not need motor safety is a spinning flywheel for a shooter. If this mechanism gets latched on a particular value it will simply continue spinning until the robot is disabled. By default Motor Safety is enabled for RobotDrive objects and disabled for all other speed controllers and servos.

Motor Safety Operation

The Motor Safety feature operates by maintaining a timer that tracks how long it has been since the feed() method has been called for that actuator. Code in the Driver Station class initiates a comparison of these timers to the timeout values for any actuator with safety enabled every 5 received packets (100ms nominal). The set() methods of each speed controller class and the set() and setAngle() methods of the servo class call feed() to indicate that the output of the actuator has been updated.

Enabling/Disabling Motor Safety

C++

```
exampleJaguar->SetSafetyEnabled(true);
exampleJaguar->SetSafetyEnabled(false);
```

Java

```
exampleJaguar.setSafetyEnabled(true);
exampleJaguar.setSafetyEnabled(false);
```

Motor safety can be enabled or disabled on a given actuator, potentially even dynamically within a program. However, if you determine a mechanism should be protected by motor safety, it is likely that it should be protected all the time.

Configuring the Safety timeout

C++

```
exampleJaguar->SetExpiration(.1);
```

Java

```
exampleJaguar.setExpiration(.1);
```

Depending on the mechanism and the structure of your program, you may wish to configure the timeout length of the motor safety (in seconds). The timeout length is configured on a per actuator basis and is not a global setting. The default (and minimum useful) value is 100ms.

On/Off control of motors and other mechanisms - Relays

For On/Off control of motors or other mechanisms such as solenoids, lights or other custom circuits, WPILib has built in support for relay outputs designed to interface to the Spike H-Bridge Relay from VEX Robotics. These devices utilize a 3-pin output (GND, forward, reverse) to independently control the state of two relays connected in an H-Bridge configuration. This allows the relay to provide power to the outputs in either polarity or turn both outputs on at the same time.

Relay connection overview

The roboRIO provides the connections necessary to wire IFI spikes via the relay outputs. The breakout board provides a total of eight outputs, four forward and four reverse. The forward output signal is sent over the pin farthest from the edge of the board, labeled as FWD on the silkscreen, while the reverse signal output is sent over the center pin, labeled REV. The final pin is a ground connection.

Relay Directions in WPILib

Within WPILib relays can be set to kBothDirections (reversible motor or two direction solenoid), kForwardOnly (uses only the forward pin), or kReverseOnly (uses only the reverse pin). If a value is not input for direction, it defaults to kBothDirections . This determines which methods in the Relay class can be used with a particular instance.

Setting Relay Directions

C++

```
Relay *exampleRelay = new Relay(1);
Relay *exampleRelay = new Relay(1, Relay::Value::kForward)

exampleRelay->Set(Relay::Value::kOn);
exampleRelay->Set(Relay::Value::kForward);
```

```
Java

exampleRelay = new Relay(1);
exampleRelay = new Relay(1, Relay.Value.kForward);

exampleRelay.set(Relay.Value.kOn);
exampleRelay.set(Relay.Value.kForward);
```

Relay state is set using the set() method. The method takes as a parameter an enumeration with the following values:

- kOff - Turns both relay outputs off
- kForward - Sets the relay to forward (M+ @ 12V, M- @ GND)
- kReverse - Sets the relay to reverse (M+ @ GND, M- @ 12V)
- KOn - Sets both relay outputs on (M+ @ 12V, M- @ 12V). Note that if the relay direction is set such that only the forward or reverse pins are enabled this method will be equivalent to kForward or kReverse, however it is not recommended to use kOn in this manner as it may lead to confusion if the relay is later changed to use kBothDirections. Using kForward and kReverse is unambiguous regardless of the direction setting.

Operating a compressor for pneumatics

The Pneumatics Control Module from Cross the Road Electronics allows for integrated closed loop control of a compressor. Creating any instance of a Solenoid or Double Solenoid object will enable the Compressor control on the corresponding PCM. **The Compressor object is only needed if you want to have greater control over the compressor or query compressor status.**

Instantiating, Starting and Stopping a Compressor

C++

```
Compressor *c = new Compressor(0);

c->SetClosedLoopControl(true);
c->SetClosedLoopControl(false);
```

Java

```
Compressor c = new Compressor(0);

c.setClosedLoopControl(true);
c.setClosedLoopControl(false);
```

To use the Compressor class create an instance of the Compressor object by passing in the PCM Node ID (default 0). For more information about PCM Node IDs see the Solenoid article and the Updating and Configuring Pneumatics Control Module and Power Distribution Panel article.

The compressor closed loop control can be enabled and disabled by using the SetClosedLoopControl() method. When closed loop control is enabled the PCM will automatically turn the compressor on when the pressure switch is closed (below the pressure threshold) and

turn it off when the pressure switch is open (~120PSI). When closed loop control is disabled the compressor will not be turned on.

Compressor Status

C++

```
bool enabled = c->Enabled();
bool pressureSwitch = c->GetPressureSwitchValue();
float current = c->GetCompressorCurrent();
```

Java

```
boolean enabled = c.enabled();
boolean pressureSwitch = c.getPressureSwitchValue();
float current = c.getCompressorCurrent();
```

The other reason to create a compressor object would be to query the status of the compressor. The state (currently on or not), pressure switch state, and compressor current can all be queried from the Compressor object.

Operating pneumatic cylinders - Solenoids

There are two ways to connect and operate pneumatic solenoid valves to trigger pneumatic cylinder movement using the current control system. One option is to hook the solenoids up to a Spike relay; to learn how to utilize solenoids connected in this manner in code see the article on Relays. The second option is to connect the solenoids to a Cross the Road Electronics Pneumatics Control Module. To solenoids connected to a PCM in code, use the WPILib "Solenoid" and/or "Double Solenoid" classes, detailed below.

Solenoid Overview

The pneumatic solenoid valves used in FRC are internally piloted valves. For more details on the operation of internally piloted solenoid valves, see this Wikipedia article. One consequence of this type of valve is that there is a minimum input pressure required for the valve to actuate. For many of the valves commonly used by FRC teams this is between 20 and 30 psi. Looking at the LEDs on the PCM itself is the best way to verify that code is behaving the way you expect in order to eliminate electrical or air pressure input issues.

Single acting solenoids apply or vent pressure from a single output port. They are typically used either when an external force will provide the return action of the cylinder (spring, gravity, separate mechanism) or in pairs to act as a double solenoid. A double solenoid switches air flow between two output ports (many also have a center position where neither output is vented or connected to the input). Double solenoid valves are commonly used when you wish to control both the extend and retract actions of a cylinder using air pressure. Double solenoid valves have two electrical inputs which connect back to two separate channels on the solenoid breakout.

PCM Module Numbers

PCM Modules are identified by their Node ID. The default Node ID for PCMs is 0. If using a single PCM on the bus it is recommended to leave it at the default Node ID. For more information about setting PCM Node IDs see this article in the Getting Started with the 2015 Control System manual.

Single Solenoids in WPILib

C++

```
Solenoid *exampleSolenoid = new Solenoid(1);

exampleSolenoid->Set(true);
exampleSolenoid->Set(false);
```

Java

```
Solenoid exampleSolenoid = new Solenoid(1);

exampleSolenoid.set(true);
exampleSolenoid.set(false);
```

Single solenoids in WPILib are controlled using the Solenoid class. To construct a Solenoid object, simply pass the desired port number (assumes Node ID 0) or Node ID and port number to the constructor. To set the value of the solenoid call set(true) to enable or set(false) to disable the solenoid output.

Double Solenoids in WPILib

C++

```
DoubleSolenoid exampleDouble = new DoubleSolenoid(1, 2);

exampleDouble->Set(DoubleSolenoid::Value::kOff);
exampleDouble->Set(DoubleSolenoid::Value::kForward);
exampleDouble->Set(DoubleSolenoid::Value::kReverse);
```

Java

```
DoubleSolenoid exampleDouble = new DoubleSolenoid(1, 2);

exampleDouble.set(DoubleSolenoid.Value.kOff);
exampleDouble.set(DoubleSolenoid.Value.kForward);
exampleDouble.set(DoubleSolenoid.Value.kReverse);
```

Double solenoids are controlled by the DoubleSolenoid class in WPILib. These are constructed similarly to the single solenoid but there are now two port numbers to pass to the constructor, a forward channel (first) and a reverse channel (second). The state of the valve can then be set to kOff (neither output activated), kForward (forward channel enabled) or kReverse (reverse channel enabled).

Using CAN Devices

Using the CAN subsystem with the RoboRIO

Using CAN with the RoboRIO has many advantages over previous connection methods between the robot controller and peripheral devices.

1. CAN connections are through a single wire that is daisy-chained between all the devices so *home run* wiring isn't required.
2. Since this is protocol-based signaling the devices can be smart and accept higher level commands besides start, stop and set speed.
3. Devices can report status back to the robot controller making it possible to have much better control algorithms with devices that use CAN.

There are a number of CAN devices supported in the FRC control system:

1. Jaguar speed controllers
2. CAN-Talon speed controllers
3. The Power Distribution Panel (PDP)
4. The Pneumatics Control Module (PCM)

The devices are typically connected to the RoboRIO CAN bus using twisted pair wiring (except the Jaguar which uses modular phone-style connectors).

CAN bus topology and termination

The CAN bus must be *terminated* at each end of the bus, that is bridged with a termination resistor of 120 ohms. Conveniently both the RoboRIO (start of bus) and the PDP board can supply termination. So a CAN bus that starts at the RoboRIO, goes through several devices, and ends at the PDP board (with the termination jumper installed) will provide the correct termination. Nothing else has to be done.

CAN operation and timing

It's helpful to understand the timing of the messages sent and received by your program to and from the CAN bus. With this information it will be easier to write and debug CAN-based robot programs.

Completely rewritten CAN subsystem. All the CAN code is now non-blocking for the program except in the constructor. When creating new CAN Jaguars the system waits for a short period of time to receive version and some status information back from the device being created. This delay is 50ms maximum, but will continue on as soon as the data is received. If no information is

received within 50ms, the device creation will fail (either an exception in Java or an error status in C++).

- Setpoints that the program sets are updated to the CAN devices (Jaguars for now) every 20ms regardless of how often your program sets them. That is to say, once your program sets a set point, it will continue to be sent to the Jaguar in the background.
- Device status is returned every 20ms for each device automatically - the program does not need to request those updates. Whenever the program requests status from a CAN device it will be no older than 20ms.
- Configuration data only needs to be sent once. The software will verify that the actual values match the configured values and resend as necessary. This means that a Jaguar rebooting while the robot is running should have all of it's configuration values restored within a few updates after coming back.
- The background updating of values happens whenever any parameter is set on the Jaguar. Each time a value is set, the Jaguar objects will scan for any unverified data and request updates to that data. This means that the restoring of data will only happen when the program is updating some value (any value) in the Jaguar. This is a little non-obvious, but it means that there are no background threads and blocking calls required. The assumption is that the program will periodically set at least the set point, triggering the update to happen.

Jaguar speed controllers

When used with CAN control, Jaguars are *smart* speed controllers. You can simply send a command to the Jaguar such as a position setpoint and it will use attached sensors to move to that position and maintain it. This relieves the robot controller from having to run control loops that read the sensors, determine the error, and supply correction values.

To use the Jaguar speed controller in one of the CAN modes, you set the speed controller mode to one of:

1. Voltage mode - you supply the positive or negative voltage that is within the bounds of the actual bus voltage (typically around 12V) that you would like the speed controller to output to the motor. The bus voltage will typically vary depending on how long the battery has been in use, so supplying absolute values will guarantee consistant output voltage provided that the request is less than or equal to the actual bus voltage.
2. Percentage mode - you supply a percentage of bus voltage expressed as a floating point between -1 and 1. The actual voltage supplied to the motor will vary depending on the charge of the battery and will actually change as the robot is operating.
3. Position mode - the CANJaguar object uses a specified sensor (either a quadrature encoder or potentiometer) and PID values to move the motor to a specified setpoint. All the closed loop control is done inside the speed controller itself so that your program doesn't need to hold the position in software.
4. Current mode - a current value is supplied in Amps which is effectively the requested motor torque (twisting effort). The Jaguar uses an internal current sensor and your PID values to keep the current to the requested value.
5. Speed mode - CANJaguar object uses a specified sensor (encoder or quadrature encoder) and PID values to move the motor at a requested speed in rotations per minutes (RPM).

When using speed or position mode a sensor is attached to the inputs on the Jaguar (not to the RoboRIO). This sensor is referred to as the **reference** device. This reference device is used to determine the error for closed loop feedback control. The choices of reference devices are potentiometers (for position mode), quadrature encoders (for position or speed mode), or single-input encoders (for speed mode). Even though there are separate inputs for potentiometers and encoders on the Jaguar, only one reference device can be in use at a time.

In addition, a reference device may be supplied for Voltage, Percentage, and Current modes that is not used for closed loop control, but can be read to monitor the performance of the speed controller.

The Jaguar also has inputs for forward and reverse **limit switches**. The limit switches will turn the Jaguar off when the switch corresponding to the direction of motion is pressed (should be wired so that this opens the switch, also known as "Normally Open" or NO) while allowing the motor to operate in the opposite direction. This is used to limit the travel of a mechanism without writing any code. For example, when the switch at the top of travel for an arm is closed, the motor will stop moving in that direction. Which is forward and which is reverse depends on the motor polarity so it is a good idea to test to make sure that it is stopping the motor in the correct direction at the correct limit.

In all case, a new CANJaguar object is instantiated, the mode is selected, then you can begin using it in your program.

Creating Jaguar objects

Each physical Jaguar has a corresponding Jaguar object in either C++ or Java. Jaguar objects are created using the new operator in either language as follows:

```
m_jaguar = new CANJaguar(CANJaguarIDValue);
```

The value of *CANJaguarIDValue* is the CAN ID programmed in the web interface of the RoboRIO for the device being instantiated. **Be sure that the most recent firmware version is installed on the Jaguar**. After creating the Jaguar instance, the operating mode should be set and the enableControl() method called to set the operating mode as shown:

```
m_jaguar->SetPercentMode(CANJaguar::QuadEncoder, 360);
m_jaguar->EnableControl();
m_jaguar->Set(0.0f);
```

The control mode is not actually set until EnableControl() is called to give the program a chance to set other parameters specific to the operating mode. In this example the Jaguar is set to Percent Mode with a quadrature encoder as the reference sensor. Specifying the encoder in the SetPercentMode() method will allow the program to get feedback on the encoders current position as shown:

```
double initialPosition = m_jaguar->GetPosition();
```

Using limits

There are two types of limits that can be used with the Jaguar:

1. Soft limits - reference device values that the Jaguar won't exceed in either direction such as a minimum or maximum number of encoder rotations.
2. Limit switches - switches that control the maximum movement of a mechanism

The soft limits can be enabled and disabled, the limit switch control is always operating in all Jaguar modes (including PWM). If limit switches are not used, jumpers should be installed to allow the Jaguar to operate properly. Refer to the Jaguar documentation for details on using the jumpers.

Using closed loop control in the Jaguar

Closed loop control means that the Jaguar looks at sensor inputs, computes an error value (difference between sensor value and setpoint), operates some actuators (motors) and loops. The Jaguars can perform this algorithm inside the device for Speed, Position and Current modes of operation. To use closed loop feedback you must do the following:

1. Set the requested mode using the SetMode() method
2. Supply P, I, and D constants in the correct set mode method
3. Supply a sensor appropriate for the operation
4. and call EnableControl() to start the controller running.

The first three operations are typically done with the SetMode() method - it is supplied with the sensor and P, I, and D constants.

Using voltage control

The output voltage can either be a percentage of the system bus voltage or an absolute voltage as described above. To set the Jaguar in Voltage mode use the following method:

```
                m_jaguar->SetVoltageMode();
```

and to set it into Percent Voltage mode use this method:

```
        m_jaguar->SetPercentMode();
```

In either case a reference device may be specified as shown here:

```
        m_jaguar->SetPercentMode(CANJaguar::QuadEncoder, 360);
```

In this case a quadrature encoder is selected as the reference device with a CPI (counts per revolution) value of 360. You can read the value of the encoder but it is not used for controlling the device - it is simply treated as an attached sensor.

Using position control

Position control can use either a potentiometer or quadrature encoder along with PID constants to move the motor to precise positions. When setting position mode, you need to specify the sensor (in this case a quadrature encoder), the number of counts per revolution, and the P, I, and D constants for the internal PID feedback loop.

```
        m_jaguar->SetPositionMode(CANJaguar::QuadEncoder, 360, 10.0f, 0.4f, 0.2f);
        m_jaguar->EnableControl();
        double setpoint = m_jaguar->GetPosition() + 10.0f;
        m_jaguar->Set(setpoint);
```

This program puts the Jaguar into position mode and drives the motor to 10 encoder rotations forward. Remember, this is not necessarily motor shaft rotations since the encoder could be mounted on an intermediate gear of a transmission.

Using current control

Using speed control

Put the Jaguar into speed mode by calling the SetMode method as shown:

```
setSpeedMode(EncoderTag tag, int codesPerRev, double p, double i, double d)
```

The EncoderTag is one of: kEncoder, kQuadEncoder, or kPotentiometer. The codesPerRev argument is the number of counts per revolution that the encoder uses.

Ramping the speed

Getting status from the Jaguar

Various status values can be read from the Jaguars as shown here:

```
        m_jaguar->GetBusVoltage();
        m_jaguar->GetOutputVoltage();
        m_jaguar->GetOutputCurrent();
        m_jaguar->GetTemperature();

        m_jaguar->GetFaults();
```

These methods return various operating values from the Jaguar. The last method, GetFaults() returns a bit field with individual bits describing if there were faults of various types. For a full description of the faults and the other status methods refer to the header file or the JavaDocs for your language of choice.

When the Jaguar is operating, it is set to send status values back to the RoboRIO every 20ms without the program needing to request this service. This means that the status that you read from these methods may be as old as 20ms or more recent depending on when the last set of messages was received.

Pneumatics Control Module

The Pneumatics Control Module (PCM) is a CAN-based device that provides complete control over the compressor and up to 8 solenoids per module. The PCM is integrated into WPILib through a series of classes that make it simple to use. The examples shown here are for Java programs, the C++ classes have the same names except for the method capitalization, following the same conventions as used throughout the rest of WPILib.

Moving from the old Compressor and Solenoid classes should be fairly easy. The closed loop control of the Compressor and Pressure switch is handled by the PCM hardware and the Solenoids are handled by the upgraded Solenoid class that now controls the solenoid channels on the PCM.

An additional PCM module can be used where the modules corresponding solenoids are differentiated by the module number in the constructors of the Solenoid and Compressor classes.

Controlling the Compressor

The PCM handles the closed loop control of the compressor internally when the pressure switch and compressor are properly wired. To enable this control, all that is needed is an instantiated Solenoid object and the robot to be Enabled. For more information see Operating a compressor for pneumatics.

Using Solenoids

For the RoboRIO the WPILib Solenoid class has been replaced by one that now implements solenoids using the PCM. The same methods will now control pneumatics plugged into the Solenoid ports on the PCM so your code should run mostly unchanged. For more information see Operating pneumatic cylinders - Solenoids

Power Distribution Panel

The Power Distribution Panel (PDP) for 2015 adds the capability to measure the current to each device connected to any of the circuit breaker protected 12V outputs. Having this capability offers the opportunity to use a number of algorithm requiring sensing of the torque being developed by motors without requiring additional hardware. The PDP is connected to the RoboRIO through the CAN bus and the libraries take care of managing the communications.

Create an instance of the PowerDistributionPanel object to use it:

PowerDistributionPanel pdp = new PowerDistributionPanel();

Note: it is not necessary to create a PowerDistributionPanel object unless you need to read values from it. The board will work and supply power on all the channels even if the object is never created.

PDP CAN ID

To work with the current versions of C++ and Java WPILib, the CAN ID for the PDP must be 0.

Reading the PDP voltage and temperature

You can read the incoming voltage to the PDP and the temperature of the components on the PDP. Measuring the voltage can be important if motors are operating at a high torque setting causing the system battery voltage to drop.

Reading the per-channel current on the PDP

You can read the current on individual channels of the PDP using the PowerDistributionPanel object. To read the current on channel 1 use the method getCurrent:

```
double current = pdp.getCurrent(1);
```

This will return the current value in amps.

Note: the channel numbers are 0-based.

Talon SRX CAN

The Talon SRX motor controller is a CAN-enabled "smart motor controller" from Cross The Road Electronics/VEX Robotics. The Talon SRX can be controlled over the CAN bus or PWM interface. When using the CAN bus control, this device can take inputs from limit switches and potentiometers, encoders, or similar sensors in order to perform advanced control such as limiting or PID(F) closed loop control on the device.

Extensive documentation about programming the Talon SRX in all three FRC languages can be found in the Talon SRX Software Reference Manual on CTRE's Talon SRX product page.

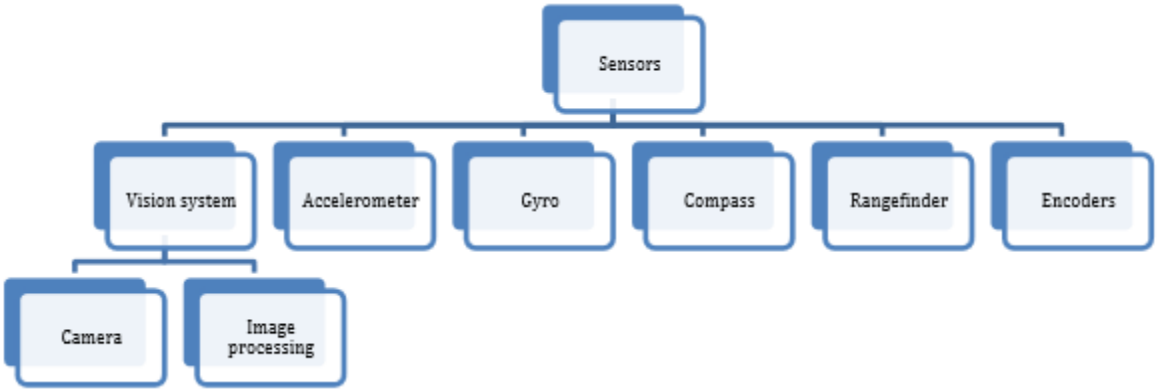
Note: CAN Talon SRX has been removed from WPILib. See this blog for more info and find the CTRE Toolsuite installer here: http://www.ctr-electronics.com/control-system/hro.html#product_tabs_technical_resources

WPILib sensors

WPILib Sensor Overview

The WPI Robotics Library supports the sensors that are supplied in the FRC kit of parts, as well as many commonly used sensors available to FIRST teams through industrial and hobby robotics suppliers.

Types of supported sensors



On the roboRIO, the FPGA implements all the high speed measurements through dedicated hardware ensuring accurate measurements no matter how many sensors and motors are connected to the robot. This is an improvement over previous systems, which required complex real time software routines. The library natively supports sensors in the categories shown below:

- Wheel/motor position measurement - Gear-tooth sensors, encoders, analog encoders, and potentiometers
- Robot orientation - Compass, gyro, accelerometer, ultrasonic rangefinder
- Generic - Pulse output Counters, analog, I2C, SPI, Serial, Digital input

There are many features in the WPI Robotics Library that make it easy to implement sensors that don't have prewritten classes. For example, general purpose counters can measure period and count from any device generating output pulses. Another example is a generalized interrupt facility to catch high speed events without polling and potentially missing them.

Switches - Using limit switches to control behavior

Limit switches are often used to control mechanisms on robots. While limit switches are simple to use, they only can sense a single position of a moving part. This makes them ideal for ensuring that movement doesn't exceed some limit but not so good at controlling the speed of the movement as it approaches the limit. For example, a rotational shoulder joint on a robot arm would best be controlled using a potentiometer or an absolute encoder, the limit switch could make sure that if the potentiometer ever failed, the limit switch would stop the robot from going to far and causing damage.

What values are provided by the limit switch

Limit switches can have "normally opened" or "normally closed" outputs. The usual way of wiring the switch is between a digital input signal connection and ground. The digital input has pull-up resistors that will make the input be high (1 value) when the switch is open, but when the switch closes the value goes to 0 since the input is now connected to ground. The switch shown here has both normally open and normally closed outputs.

Polling waiting for a switch to close

C++

```
#include "WPILib.h"


class Robot: public SampleRobot
{
        DigitalInput limitSwitch;

public:
        Robot() {

        }

        void RobotInit()
        {
                limitSwitch = new DigitalInput(1);
        }

        void OperatorControl() {
                // more code here
                while (limitSwitch.Get()) {
                        Wait(10);
                }
                // more code here
        }
```

Java

```
package org.usfirst.frc.team1.robot;

import edu.wpi.first.wpilibj.DigitalInput;
import edu.wpi.first.wpilibj.SampleRobot;
import edu.wpi.first.wpilibj.Timer;
```

```
public class RobotTemplate extends SampleRobot {

        DigitalInput limitSwitch;

 \   public void robotInit() {
            limitSwitch = new DigitalInput(1);
    }

 \   public void operatorControl() {
            // more code here
            while (limitSwitch.get()) {
                    Timer.delay(10);
            }
        // more code here
    }
```

You can write a very simple piece of code that just reads the limit switch over and over again waiting until it detects that its value transitions from 1 (opened) to 0 (closed). While this works, it's usually impractical for the program to be able to just wait for the switch to operate and not be doing anything else, like responding to joystick input. This example shows the fundamental use of the switch, but while the program is waiting, nothing else is happening.

Command-based program to operate until limit switch closed

```
package edu.wpi.first.wpilibj.templates.commands;

public class ArmUp extends CommandBase {
    public ArmUp() {
    }

    protected void initialize() {
        arm.armUp();
    }

    protected void execute() {
```

```
    }

    protected boolean isFinished() {
        return arm.isSwitchSet();
    }

    protected void end() {
        arm.armStop();
    }

    protected void interrupted() {
        end();
    }
}
```

Commands call their execute() and isFinished() methods about 50 times per second, or at a rate of every 20ms. A command that will operate a motor until the limit switch is closed can read the digital input value in the isFinished() method and return true when the switch changes to the correct state. Then the command can stop the motor.

Remember, the mechanism (an Arm in this case) has some inertia and won't stop immediately so it's important to make sure things don't break while the arm is slowing.

Using a counter to detect the closing of the switch

```
package edu.wpi.first.wpilibj.templates.subsystems;
import edu.wpi.first.wpilibj.Counter;
import edu.wpi.first.wpilibj.DigitalInput;
import edu.wpi.first.wpilibj.SpeedController;
import edu.wpi.first.wpilibj.Victor;
import edu.wpi.first.wpilibj.command.Subsystem;
public class Arm extends Subsystem {

    DigitalInput limitSwitch = new DigitalInput(1);
    SpeedController armMotor = new Victor(1);
    Counter counter = new Counter(limitSwitch);

    public boolean isSwitchSet() {
        return counter.get() > 0;
```

```
    }

    public void initializeCounter() {
        counter.reset();
    }

    public void armUp() {
        armMotor.set(0.5);
    }

    public void armDown() {
        armMotor.set(-0.5);
    }

    public void armStop() {
        armMotor.set(0.0);
    }
    protected void initDefaultCommand() {
    }
}
```

It's possible that a limit switch might close then open again as a mechanism moves past the switch. If the closure is fast enough the program might not notice that the switch closed. An alternative method of catching the switch closing is use a Counter object. Since counters are implemented in hardware, it will be able to capture the closing of the fastest switches and increment it's count. Then the program can simply notice that the count has increased and take whatever steps are needed to do the operation.

Above is a subsystem that uses a counter to watch the limit switch and wait for the value to change. When it does, the counter will increment and that can be watched in a command.

Create a command that uses the counter to detect switch closing

```
package edu.wpi.first.wpilibj.templates.commands;

public class ArmUp extends CommandBase {

    public ArmUp() {
    }
```

```
    protected void initialize() {
        arm.initializeCounter();
        arm.armUp();
    }

    protected void execute() {
    }

    protected boolean isFinished() {
        return arm.isSwitchSet();
    }

    protected void end() {
        arm.armStop();
    }

    protected void interrupted() {
        end();
    }
}
```

This command initializes the counter in the above subsystem then starts the motor moving. It then tests the counter value in the isFinished() method waiting for it to count the limit switch changing. When it does, the arm is stopped. By using a hardware counter, a switch that might close then open very quickly can still be caught by the program.

How do I do _____? - Selecting the right sensor for the job

The articles following this one provide details on the operation and use of a variety of sensors with WPILib, but how do you know which sensor to use for a particular task? This article attempts to explain possible sensor choices for a variety of common FRC tasks

Detecting one or two positions of a mechanism

Detecting one or two positions for a motor driven mechanism is a very common FRC task. The most common occurrence is detecting when a mechanism reaches a limit on either end, but detecting a desired position or home position is also fundamentally the same task.

Limit Switches

Mechanical limit switches are one of the most common solutions to this scenario. If the switches truly are defining the limits of the mechanism make sure that the switches are set up in a position where they can't be missed by the mechanism and wont get damaged by the mechanism. Limit switches are useful because they are very simple to implement, are fairly cheap, and can be used in a large variety of situations.

Switches - Using limit switches to control behavior

Detecting the position of a mechanism at many different points, or points that are not limits

Sometimes you need to know how high up your elevator is, without having that height be the top of your elevator, or how high up an arm is from its starting position, or what angle your shooter head is pointed at. These problems could be solved by a clever team with some tricky placements of limit switches and catches for them, but there are other sensors that are designed for that job.

Ultrasonic Sensors

Distance sensors like ultrasonic sensors can give you a fairly accurate measurement of how far away the closest object in its field of vision is from the sensor, meaning that if you set it up correctly, you will be able to stop your arm or elevator when it gets to the points you desire. Usually these measurements will be accurate to 2-3 inches, meaning if you need much greater

accuracy, you might want to look into a different sensor in this section, but this should be good enough for most cases.

Ultrasonic Sensors - Measuring robot distance to a surface

Infrared Distance Sensors

Infrared distance sensors are very similar to ultrasonic sensors, they just use a different method of measurement. These sensors are not explicitly covered in our tutorials, but most of these sensors have an analog output, meaning you can use the analog input class to get a voltage from the sensor, which converts to a specific distance in most cases. The advantage of infrared sensors compared to ultrasonic sensors is that they are not affected by a noisy stadium, in some cases an ultrasonic sensor can get a less accurate value because of how much noise exists at a competition.

Analog inputs

Counters and Encoders

If your arm or elevator is driven by a motor, you can measure the number of rotations the motor has turned to get how high up the arm or elevator is. This method is more of a guess and check method than a known height measurement check, but with a couple of tests and some print line statements, you can easily find the number of rotations you need to get to certain heights, just remember that your measurement is always based on something, and if your hard coded number of rotations is based on it resting on the floor, and it starts in the air slightly one match, it will be at the wrong set point when it gets up to the top, so it is important to know how to initially set it up.

Counters - Measuring rotation, counting pulses and more

Encoders - Measuring rotation of a wheel or other shaft

Potentiometers

If you need to know the angle of the arm, a potentiometer will be the job for you, it converts the angle of motion to a readable analog value. This works really well for knowing where your shooter is pointed, or how high up your arm is, and with some sensors you can measure distance traveled linearly too, so it can work with an elevator.

Potentiometers - Measuring joint angle or linear motion

Accelerometers

Measuring the tilt of a surface is possible with an accelerometer, for an arm that would give you a good idea of what the angle is, if that is how you wanted to measure it. These are really useful for limits that you keep for reasons like the robot shouldn't go further than that or it will possibly break itself, and sometimes isn't accurate enough for pinpoint aiming.

Accelerometers - measuring acceleration and tilt

Driving Straight

Sometimes your robot is not being controlled by a human that can easily correct any slight deviations to the robots direction. When you need you robot to drive itself straight, you have a couple sensors that will work to get the job done.

Gyros

The gyroscope is a sensor that points in a direction, and will tell you when you deviate from that direction, and how far. This can help us correct for one of the drive motors being slightly slower than the other, or to give us an accurate measurement of how far we have turned when we are in autonomous. They also measure off of an initial point, so if the robot is put in the wrong place, it will not know that.

Gyros - Measuring rotation and controlling robot driving direction

Encoders

If you have encoders on the drive motors, you can measure how far the wheels have turned, and if one of them measures further than the other, you can correct for it. This is not as effective especially when turning because wheels can slip, and encoders aren't quite as accurate as gyroscopes for these measurements.

Encoders - Measuring rotation of a wheel or other shaft

How far have I gone?

When you are programming an autonomous program, you will most likely need to drive, and because your robot doesn't have the senses we have without us adding them, it wont know how far its gone or how far it needs to go without sensors.

Encoders

This is where encoders really shine. Encoders measure the number of rotations a motor has gone since you last reset them. This means you can calculate the rotations to distance calculation for your robot by doing the math for the different gear and pulley ratios. This gets a little less accurate the further away from your wheels you put your encoder, because you can loose distance in slack from the pulleys, the belts jumping pegs on the pulleys, and the wheels slipping on the surface, all giving you a longer distance than you have really gone. This is somewhat avoided when you have multiple encoders by averaging the rotations they measure, so that any slippage is mitigated by having better data.

Encoders - Measuring rotation of a wheel or other shaft

Distance Sensors

Although it is not very common due to practical concerns of setting up the robot on the field, distance sensors can be used to tell how far you have gone if you have a point to measure from. Because you are measuring from a field element or wall, it is usually not possible to tell how far you have gone after a turn, or how far you have gone if its too far away from a static object.

Ultrasonic Sensors - Measuring robot distance to a surface

Cameras and Vision

Most of the time, when you are thinking about how to solve a problem, you are not trying to do it blindfolded with huge padded gloves on with ear plugs in. This is pretty much how the robot is experiencing the world without any sensors, it can't see the game piece, it can't feel how tightly it is grabbing that tube, without sensors to detect these things, the robot can't know if its done its job correctly. One of the major things humans do to get information from our world is look at it, judge things like position and distance, and identify locations of important things around us.

Why use vision?

Vision is a very powerful tool, it can give you an idea of how far you are from something, how many items you have in front of you, where you are pointing, and how fast you are moving, all from one sensor. Things like how far away something is can be measured by knowing the viewing angle of the camera, the resolution, and the size of a known object in the view. Counting the number of items is a matter of object detection and recognition, and movement is measuring how fast things move toward you. These can also be coupled with the ability to stream the cameras view to the driverstation, so the driver and operator can see from the robots perspective instead of all the way across the field behind the glass.

Why not use vision?

To use vision, you need to have a few things: A good quality camera, a way to process the visual information into meaningful data, and someone who knows or is willing to learn how very advanced visual identification is done by outside libraries. With the roborio it is possible for us to actually do some or all of the computations required to turn the camera video into meaningful data, and cameras are available in the kit of parts, but for more advanced visual operations you may need to have additional processing power and higher quality cameras. Because of this most teams use vision for basic things, or just uses it as more information that the driver can use, which can help immensely.

How fast is that wheel spinning?

Sometimes, especially with shooters, you want to know how fast a wheel is spinning.

Counters and Encoders

You can use a counter or encoder to measure the number of rotations over a given period of time to get the speed the wheel is spinning at, which can be really useful for shooter wheels so that you don't shoot unless your wheel is up to speed.

Counters - Measuring rotation, counting pulses and more

Encoders - Measuring rotation of a wheel or other shaft

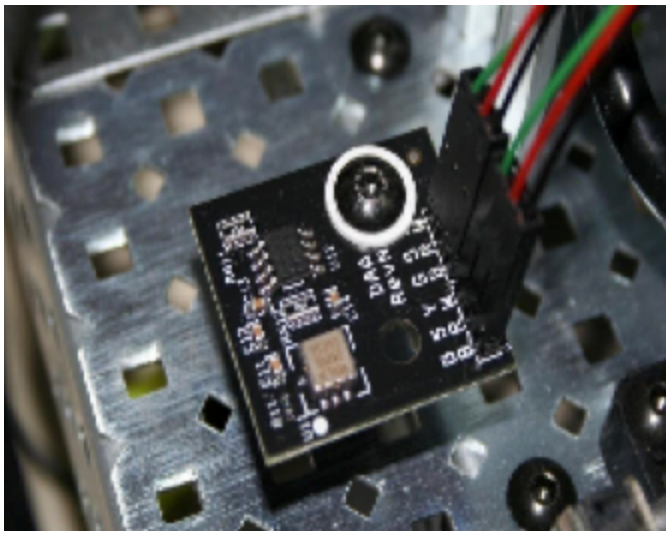
Other Sensors and Problems

Ultimately, it is up to the teams to find solutions to their individual problems when it comes to sensors on their robot. Sometimes the sensors available to the teams are not good enough, encoders not able to read at the speeds you need, ultrasonic sensors too inaccurate after a certain distance, these are all challenges to solve, and is the reason you are really here. These challenges are what teach students and mentors on teams how creative they can really be when the challenge and deadlines are put in front of them. This is when some of the best and most creative solutions to problems are created.

Accelerometers - measuring acceleration and tilt

Accelerometers measure acceleration in one or more axis. One typical usage is to measure robot acceleration. Another common usage is to measure robot tilt, in this case it measures the acceleration due to gravity.

Two-axis analog accelerometer



A commonly used part (shown in the picture above) is a two-axis accelerometer. This device can provide acceleration data in the X and Y-axes relative to the circuit board. The WPI Robotics Library you treats it as two separate devices, one for the X- axis and the other for the Y-axis. The accelerometer can be used as a tilt sensor – by measuring the acceleration of gravity. In this case, turning the device on the side would indicate 1000 milliGs or one G. Shown is a 2-axis accelerometer board connected to two analog inputs on the robot. **Note that this is not the accelerometer provided in the 2014 KOP.**

Analog Accelerometer code example

C++

```
class AccelerometerSample: public SampleRobot {
        AnalogAccelerometer *accel;
        double acceleration;

        AccelerometerSample()
```

```
        {
                accel = new AnalogAccelerometer(0); //create accelerometer on analog input
0
                accel->SetSensitivity(.018); // Set sensitivity to 18mV/g (ADXL193)
                accel->SetZero(2.5); //Set zero to 2.5V (actual value should be determined
experimentally)
        }

        public void OperatorControl() {
                while(IsOperatorControl() && IsEnabled())
                {
                        acceleration = accel->GetAcceleration();
                }
        }
}
```

Java

```
public class AccelerometerSample extends SampleRobot {
        AnalogAccelerometer accel;
        double acceleration;

        AccelerometerSample()
        {
                accel = new AnalogAccelerometer(0); //create accelerometer on analog input
0
                accel.setSensitivity(.018); // Set sensitivity to 18mV/g (ADXL193)
                accel.setZero(2.5); //Set zero to 2.5V (actual value should be determined
experimentally)
        }

        public void operatorControl() {
                while(isOperatorControl() && isEnabled())
                {
                        acceleration = accel.getAcceleration();
                }
        }
}
```

A brief code example is shown above which illustrates how to set up an analog accelerometer connected to analog channel 1. The sensitivity and zero voltages were set according to the

datasheet (assumed part is ADXL193, zero voltage set to ideal. Would need to determine actual offset of specific part being used).

Accelerometer interface

C++

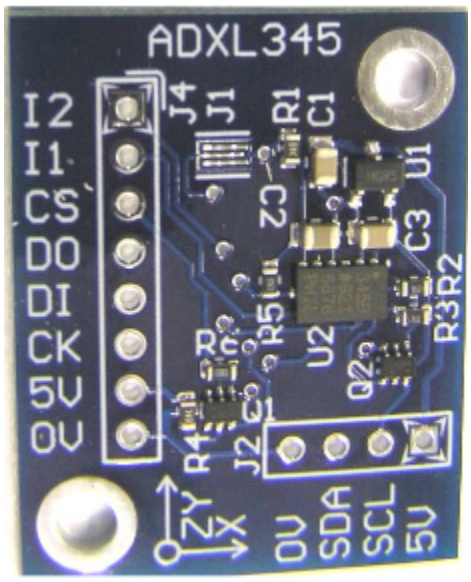
```
Accelerometer *accel;
accel = new BuiltInAccelerometer(Accelerometer:kRange_4G);
double xVal = accel->GetX();
double yVal = accel->GetY();
double zVal = accel->GetZ();
```

Java

```
Accelerometer accel;
accel = new BuiltInAccelerometer();
accel = new BuiltInAccelerometer(Accelerometer.Range.k4G);
double xVal = accel.getX();
double yVal = accel.getY();
double zVal = accel.getZ();
```

Both classes for the ADXL345 and the class for the Built-In accelerometer all inherit/implement a common Accelerometer interface. The plan in the future is to try to get the AnalogAccelerometer class to derive from this interface as well. If you are planning on using one of these sensors it is recommended to write your code against the generic interface. That way you can change between the underlying classes, if desired, with minimal changes to your code. It will also help make your code more compatible with simulation as that capability continues to develop.

ADXL345 Accelerometer



The ADXL345 is a three axis accelerometer provided as part of the sensor board in the 2012-2014 KOP. The ADXL345 is capable of measuring accelerations up to +/- 16g and communicates over I2C or SPI. Wiring instructions for either protocol can be found in the FRC component datasheet. Additional information can be found in the Analog Devices ADXL345 datasheet. WPILib provides a separate class for each protocol which handles the details of setting up the bus and enabling the sensor.

ADXL345 Code Example

C++

```
class AccelerometerSample: public SampleRobot {
        Accelerometer *accel;
        double accelerationX;
        double accelerationY;
        double accelerationZ;

        AccelerometerSample()
        {
                accel = new ADXL345_I2C(I2C::Port::kOnboard,
Accelerometer::Range::kRange_4G);
        }
```

```
        public void OperatorControl() {
                while(IsOperatorControl() && IsEnabled())
                {
                        accelerationX = accel->GetX();
                        accelerationY = accel->GetY();
                        accelerationZ = accel->GetZ();
                }
        }
}
```

Java

```
public class AccelerometerSample extends SampleRobot {
        Accelerometer accel;
        double accelerationX;
        double accelerationY;
        double accelerationZ;

        AccelerometerSample()
        {
                accel = new ADXL345_I2C(I2C.Port.kOnboard, Accelerometer.Range.k4G);
        }

        public void operatorControl() {
                while(isOperatorControl() && isEnabled())
                {
                        accelerationX = accel.getX();
                        accelerationY = accel.getY();
                        accelerationZ = accel.getZ();
                }
        }
}
```

A brief code example is shown above illustrating the use of the ADXL345 connected to the on-board I2C bus. The accelerometer has been set to operate in +/- 2g mode. The example illustrates both only the single axis method of getting the sensor values, using the Accelerometer interface. If you need synchronized readings of all 3 axes, you will have to forgo the interface and use the ADXL345 class directly to have access to the GetAccelerations() method. SPI operation is similar, refer to the Javadoc/Doxygen for the ADXL345_SPI class for additional details on using the sensor over SPI.

Built-In Accelerometer

The roboRIO contains a built-in 3-axis accelerometer with a range of +/- 8g, 12 bit resolution, and a 800 Sample/s sample rate. To use this accelerometer, use the BuiltInAccelerometer class. See the Accelerometer Interface section above for code illustrating the use of this accelerometer operating in the +/-4g mode using the generic Accelerometer interface (note when using this interface that the built-in accelerometer does not support the +/-16g mode).

Gyros - Measuring rotation and controlling robot driving direction

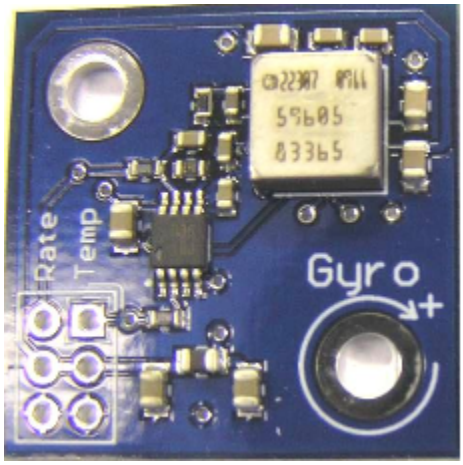
Gyros typically in the FIRST kit of parts are provided by Analog Devices, and are actually angular rate sensors. The output voltage is proportional to the rate of rotation of the axis perpendicular to the top package surface of the gyro chip. The value is expressed in mV/°/second (degrees/second or rotation expressed as a voltage). By integrating (summing) the rate output over time, the system can derive the relative heading of the robot.

Another important specification for the gyro is its full-scale range. Gyros with high full-scale ranges can measure fast rotation without “pinning” the output. The scale is much larger so faster rotation rates can be read, but there is less resolution due to a much larger range of values spread over the same number of bits of digital to analog input. In selecting a gyro, you would ideally pick the one that had a full-scale range that matched the fastest rate of rotation your robot would experience. This would yield the highest accuracy possible, provided the robot never exceeded that range.

Note: The AnalogGyro class in WPILib uses a hardware (implemented in the FPGA) accumulator to perform the integration. This means Gyros are supported on a specific, limited, set of channels. On the roboRIO this is currently Analog Inputs 0 and 1 on the on-board headers.

Note: The Gyro class has been renamed to AnalogGyro for FRC 2016 to better support newer gyros that are not necessarily connected through an analog input. There is now an interface, Gyro, used as the base for all gyros regardless of the connection type. Types should be declared using the interface, but initialized using the more specific device type.

Using the AnalogGyro class



The Gyro object should be created in the constructor of the RobotBase derived object. When the AnalogGyro object is used, it will go through a calibration period to measure the offset of the rate output while the robot is at rest to minimize drift. This requires that the robot be stationary and the gyro is unusable until the calibration is complete.

Once initialized, the GetAngle() (or getAngle() in Java) method of the Gyro object will return the number of degrees of rotation (heading) as a positive or negative number relative to the robot's position during the calibration period. The zero heading can be reset at any time by calling the Reset() (reset() in Java) method on the AnalogGyro object.

See the code samples below for an idea of how to use the AnalogGyro objects.

Setting Gyro sensitivity

The Gyro class defaults to the settings required for the 250°/sec gyro that was delivered by FIRST in the 2012-2014 Kit of Parts (ADW22307). It is important to check the documentation included with the gyro to ensure that you have the correct sensitivity setting.

To change gyro types call the SetSensitivity(float sensitivity) method (or **setSensitivity(double sensitivity)** in Java) and pass it the sensitivity in volts/°/sec. Take note that the units are typically specified in mV (volts / 1000) in the spec sheets. For example, a sensitivity of 12.5 mV/°/sec would require a SetSensitivity() (**setSensitivity()** in Java) parameter value of 0.0125.

Using a gyro to drive straight

The following example programs cause the robot to drive in a straight line using the gyro sensor in combination with the RobotDrive class. The RobotDrive.Drive method takes the speed and the turn rate as arguments; where both vary from -1.0 to 1.0. The gyro returns a value indicating the number of degrees positive or negative the robot deviated from its initial heading. As long as the robot continues to go straight, the heading will be zero. This example uses the gyro to keep the robot on course by modifying the turn parameter of the Drive method.

The angle is multiplied by a proportional scaling constant (Kp) to scale it for the speed of the robot drive. This factor is called the proportional constant or loop gain. Increasing Kp will cause the robot to correct more quickly (but too high and it will oscillate). Decreasing the value will cause the robot correct more slowly (possibly never reaching the desired heading). This is known as proportional control, and is discussed further in the PID control section of the advanced programming section.

```
C++
```

```
class GyroSample : public SampleRobot
{
        RobotDrive myRobot; // robot drive system
        AnalogGyro gyro;
        static const float kP = 0.03;

public:
        GyroSample():
                myRobot(1, 2), // initialize the sensors in initilization list
                gyro(1)
        {
                myRobot.SetExpiration(0.1);
        }

        void Autonomous()
        {
                gyro.Reset();
                while (IsAutonomous())
                {
                        float angle = gyro.GetAngle(); // get heading
                        myRobot.Drive(-1.0, -angle * kP); // turn to correct heading
                        Wait(0.004);
                }
                myRobot.Drive(0.0, 0.0); // stop robot
        }
};
```

Sample Java program for driving straight

Java

```
package edu.wpi.first.wpilibj.templates;
import edu.wpi.first.wpilibj.AnalogGyro;
import edu.wpi.first.wpilibj.RobotDrive;
import edu.wpi.first.wpilibj.SampleRobot;
import edu.wpi.first.wpilibj.Timer;
```

```
public class GyroSample extends SampleRobot {

 \   private RobotDrive myRobot; // robot drive system
    private Gyro gyro;

 \   double Kp = 0.03;

    public GyroSample() {
        gyro = new AnalogGyro(1); \            // Gyro on Analog Channel 1
        myRobot = new RobotDrive(1,2); \ // Drive train jaguars on PWM 1 and 2
        myRobot.setExpiration(0.1);
 \   }

    public void autonomous() {
        gyro.reset();
        while (isAutonomous()) {
            double angle = gyro.getAngle(); // get current heading
            myRobot.drive(-1.0, -angle*Kp); // drive towards heading 0
            Timer.delay(0.004);
        }
        myRobot.drive(0.0, 0.0);
 \   }
}
```

This is a sample Java program that drives in a straight line. See the comments in the C++ example (previous step) for an explanation of its operation.

Thanks to Joe Ross from FRC team 330 for help with this example.

Ultrasonic Sensors - Measuring robot distance to a surface

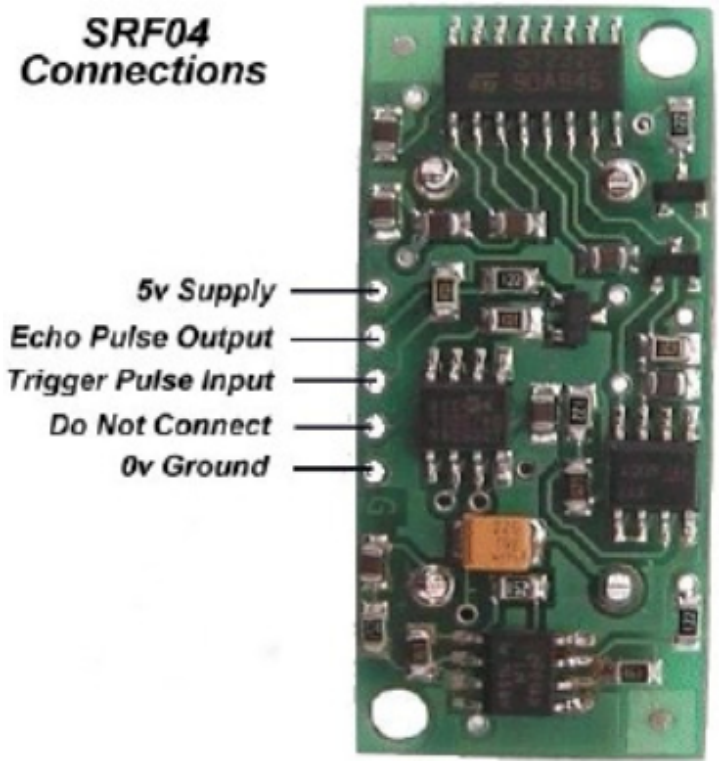
Ultrasonic sensors are a common way to find the distance from a robot to the nearest surface

Ultrasonic rangefinders

Ultrasonic rangefinders use the travel time of an ultrasonic pulse to determine distance to the nearest object within the sensing cone. There are a variety of different ways that various ultrasonic sensors communicate the measurement result including:

- Ping-Response (ex. Devantech SRF04, VEX Ultrasonic Rangefinder)
- Analog (ex. Maxbotix LV-MaxSonar-EZ1)
- I2C (ex. Maxbotix I2CXL-MaxSonar-EZ2)

Ping-Response Ultrasonic sensors



To aid in the use of Ping-Response Ultrasonic sensors such as the Devantech SRF04 pictured above, WPILib contains an Ultrasonic class. This type of sensor has two transducers, a speaker that sends a burst of ultrasonic sound, and a microphone that listens for the sound to be reflected off of a nearby object. It requires two connections to the roboRIO, one that initiates the ping and the other that tells when the sound is received. The Ultrasonic object measures the time between the transmission and the reception of the echo.

Creating an Ultrasonic object and reading the distance

C++

```
class ultrasonicSample : public SampleRobot
{
        Ultrasonic *ultra; // creates the ultra object

public:
        ultrasonicSample()
        {
                ultra = new Ultrasonic(1, 1); // assigns ultra to be an ultrasonic sensor
which uses DigitalOutput 1 for the echo pulse and DigitalInput 1 for the trigger pulse
                ultra->SetAutomaticMode(true); // turns on automatic mode
        }

        void Teleop()
        {
                int range = ultra->GetRangeInches(); // reads the range on the ultrasonic
sensor
        }
};
```

Java

```
import edu.wpi.first.wpilibj.SampleRobot;
import edu.wpi.first.wpilibj.Ultrasonic;

public class RobotTemplate extends SampleRobot {
```

```
        Ultrasonic ultra = new Ultrasonic(1,1); // creates the ultra object andassigns
ultra to be an ultrasonic sensor which uses DigitalOutput 1 for
        // the echo pulse and DigitalInput 1 for the trigger pulse
    public void robotInit() {
            ultra.setAutomaticMode(true); // turns on automatic mode
    }

    public void ultrasonicSample() {
            double range = ultra.getRangeInches(); // reads the range on the ultrasonic
sensor
    }
```

Both the Echo Pulse Output and the Trigger Pulse Input have to be connected to digital I/O ports on a Digital Sidecar. When creating the Ultrasonic object, specify which channels it is connected to in the constructor, as shown in the examples above. In this case, ULTRASONIC_ECHO_PULSE_OUTPUT and ULTRASONIC_TRIGGER_PULSE_INPUT are two constants that are defined to be the digital I/O port numbers. Do not use the ultrasonic class for ultrasonic rangefinders that do not have these connections. Instead, use the appropriate class for the sensor, such as an AnalogChannel object for an ultrasonic sensor that returns the range as an analog voltage.

Analog Rangefinders

Many ultrasonic rangefinders return the range as an analog voltage. To get the distance you multiply the analog voltage by the sensitivity or scale factor (typically in inches/V or inches/mV). To use this type of sensor with WPILib you can either create it as an Analog Channel and perform the scaling directly in your robot code, or you can write a class that will perform the scaling for you each time you request a measurement.

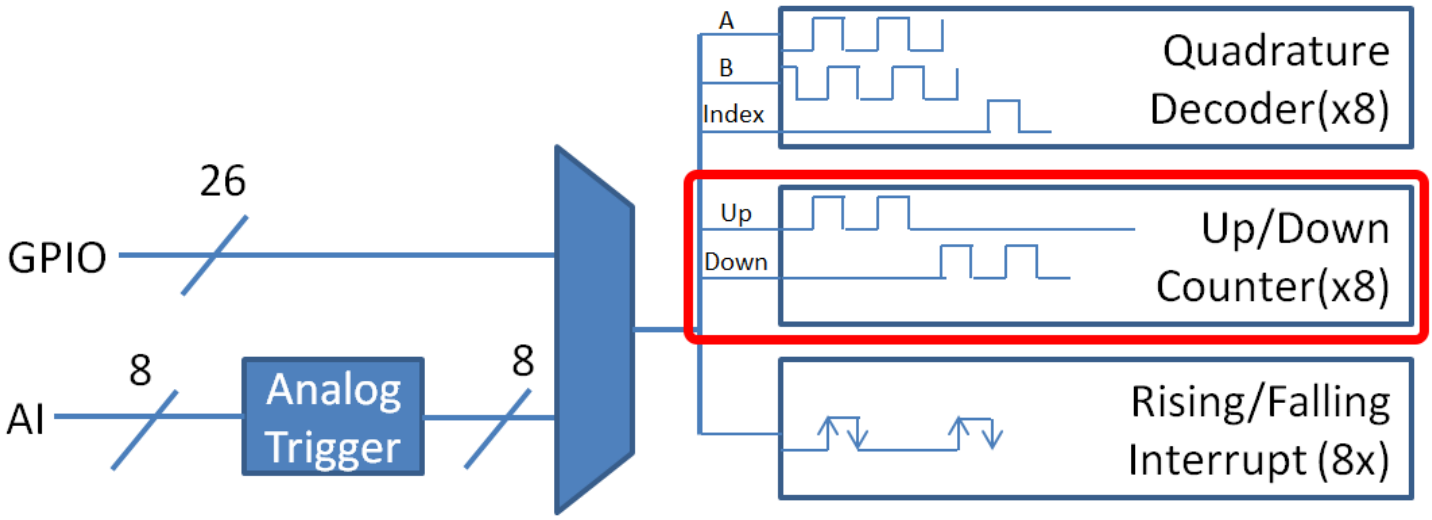
I2C and other Digital Rangefinders

Rangefinders that communicate digitally over I2C, SPI, or Serial may also be used with the roboRIO though no specific classes for these devices are provided through WPILib. Use the appropriate communication class based on the bus in use and refer to the datasheet for the part to determine what data or requests to send the device and what format the received data will be in.

Counters - Measuring rotation, counting pulses and more

Counter objects are extremely flexible elements that can count input from either a digital input signal or an analog trigger.

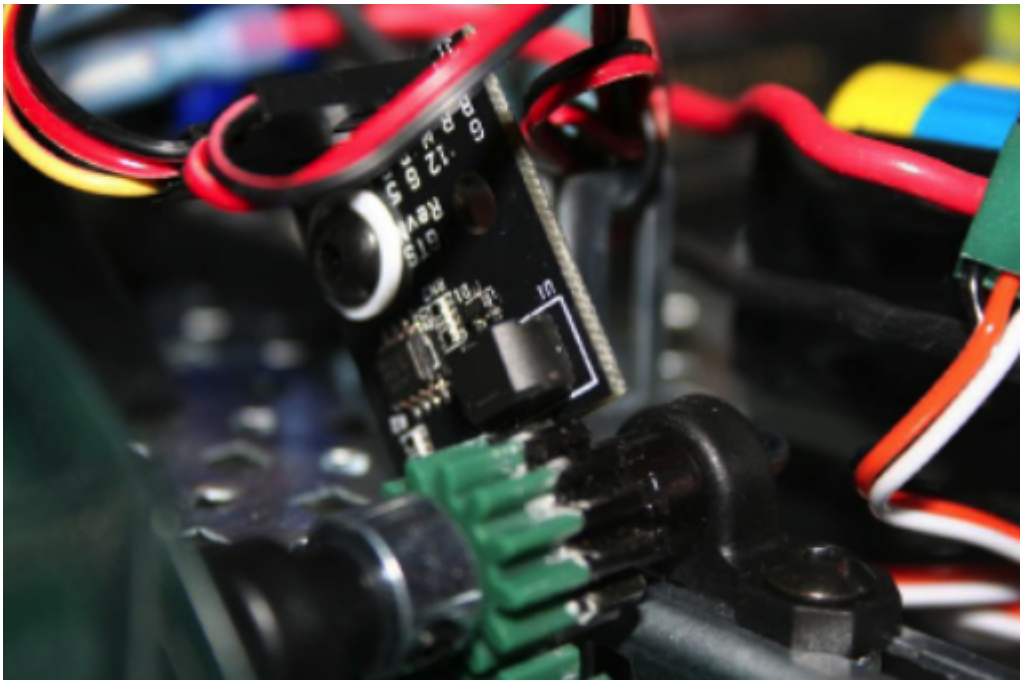
Counter Overview



There are 8 Up/Down Counter units contained in the FPGA which can each operate in a number of modes based on the type of input signal:

- Gear-tooth/Pulse Width mode - Enables up/down counting based on the width of an input pulse. This is used to implement the GearTooth sensor class with direction sensing.
- Semi-period mode - Counts the period of a portion of the input signal. This mode is used by the Ultrasonic class to measure the time of flight of the echo pulse.
- External Direction mode - Can count edges of a signal on one input with the direction (up/down) determined by a second input
- "Normal mode"/Two Pulse mode - Can count edges from 2 independent sources (1 up, 1 down)

Gear-Tooth Mode and GearTooth Sensors



Gear-tooth sensors are designed to be mounted adjacent to spinning ferrous gear or sprocket teeth and detect whenever a tooth passes. The gear-tooth sensor is a Hall-effect device that uses a magnet and solid-state device that can measure changes in the field caused by the passing teeth. The picture above shows a gear-tooth sensor mounted to measure a metal gear rotation. Notice that a metal gear is attached to the plastic gear. The gear tooth sensor needs a ferrous material passing by it to detect rotation.

The Gear-Tooth mode of the FPGA counter is designed to work with gear-tooth sensors which indicate the direction of rotation by changing the length of the pulse they emit as each tooth passes such as the ATS651 provided in the 2006 FRC KOP.

Semi-Period mode

C++

```
Counter *exampleCounterHi = new Counter(0);
Counter *exampleCounterLow = new Counter(3);
exampleCounterHi->SetSemiPeriodMode(true);
exampleCounterLow->SetSemiPeriodMode(false);
double highPulse = exampleCounterHi->GetPeriod();
double lowPulse = exampleCounterLow->GetPeriod();
```

Java
```
Counter exampleCounterHi = new Counter(0);
Counter exampleCounterLow = new Counter(3);
exampleCounterHi.setSemiPeriodMode(true);
exampleCounterLow.setSemiPeriodMode(false);
double highPulse = exampleCounterHi.getPeriod();
double lowPulse = exampleCounterLow.getPeriod();
```

The semi-period mode of the counter will measure the pulse width of either a high pulse (rising edge to falling edge) or a low pulse (falling edge to rising edge) on a single source (the Up Source). Call setSemiPeriodMode(true) to measure high pulses and setSemiPeriodMode(false) to measure low pulses. In either case, call getPeriod() to obtain the length of the last measured pulse (in seconds).

External Direction mode

The external direction mode of the counter counts edges on one source (the Up Source) and uses the other source (the Down Source) to determine direction. The most common usage of this mode is quadrature decoding in 1x and 2x mode. This use case is handled by the Encoder class which sets up an internal Counter object, and is covered in the next article Encoders - Measuring rotation of a wheel or other shaft.

Normal mode

C++
```
Counter *normalCounter = new Counter();
normalCounter->SetUpSource(1);
normalCounter->SetUpDownCounterMode();
```

Java
```
Counter normalCounter = new Counter();
normalCounter.setUpSource(1);
normalCounter.setUpDownCounterMode();
```

The "normal mode" of the counter, also known as Up/Down mode or Two Pulse mode, counts pulses occurring on up to two separate sources, one source for Up and one source for Down. A common use case of this mode is using a single source (the Up Source) with a reflective sensor or hall effect sensor as a single direction encoder. The code example above shows an alternate method of setting up the Counter sources, this method is valid for any of the modes. The method shown in the Semi-Period mode example is also perfectly valid for all modes of the counter including the Normal Mode.

Counter Settings

C++
```
Counter *normalCounter = new Counter(1);
normalCounter->SetMaxPeriod(.1);
normalCounter->SetUpdateWhenEmpty(true);
normalCounter->SetReverseDirection(false);
normalCounter->SetSamplesToAverage(10);
normalCounter->SetDistancePerPulse(12);
```

Java
```
Counter normalCounter = new Counter(1);
normalCounter.setMaxPeriod(.1);
normalCounter.setUpdateWhenEmpty(true);
normalCounter.setReverseDirection(false);
normalCounter.setSamplesToAverage(10);
normalCounter.setDistancePerPulse(12);
```

There are a few different parameters that can be set to control various aspects of the counter behavior:

- Max Period - The maximum period (in seconds) where the device is still considered moving. This value is used to determine the state of the getStopped() method and effect the output of the getPeriod() and getRate() methods.
- Update When Empty - Setting this to false will keep the most recent period on the counter when the counter is determined to be stalled (based on the Max Period described above). Setting this parameter to True will return 0 as the period of a stalled counter.
- Reverse Direction - Valid in external direction mode only. Setting this parameter to true reverses the counting direction of the external direction mode of the counter.
- Samples to Average - Sets the number of samples to average when determining the period. Averaging may be desired to account for mechanical imperfections (such as unevenly spaced reflectors when using a reflective sensor as an encoder) or as oversampling to increase resolution. Valid values are 1 to 127 samples.
- Distance Per Pulse - Sets the multiplier used to determine distance from count when using the getDistance() method.

Resetting the counter

C++
```
Counter *normalCounter = new Counter(1);
normalCounter->Reset();
```

Java
```
Counter normalCounter = new Counter(1);
normalCounter.reset();
```

Counters begin counting as soon as they are instantiated. To reset the counter value to 0 call reset().

Getting Counter Values

C++
```
Counter *normalCounter = new Counter(1);
int count = normalCounter->Get();
double distance = normalCounter->GetDistance();
double period = normalCounter->GetPeriod();
double rate = normalCounter->GetRate();
bool direction = normalCounter->GetDirection();
bool stopped = normalCounter->GetStopped();
```

Java
```
Counter normalCounter = new Counter(1);
int count = normalCounter.get();
double distance = normalCounter.getDistance();
double period = normalCounter.getPeriod();
double rate = normalCounter.getRate();
boolean direction = normalCounter.getDirection();
boolean stopped = normalCounter.getStopped();
```

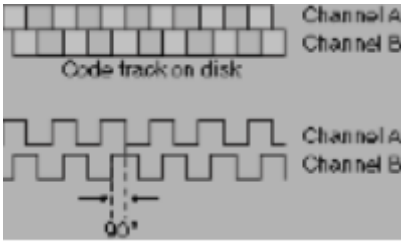
The following values can be retrieved from the counter:

- Count - The current count. May be reset by calling reset()
- Distance - The current distance reading from the counter. This is the count multiplied by the Distance Per Count scale factor.
- Period - The current period of the counter in seconds. If the counter is stopped this value may return 0, depending on the setting of the Update When Empty parameter.
- Rate - The current rate of the counter in units/sec. It is calculated using the DistancePerPulse divided by the period. If the counter is stopped this value may return Inf or NaN, depending on language.
- Direction - The direction of the last value change (true for Up, false for Down)
- Stopped - If the counter is currently stopped (period has exceeded Max Period)

Encoders - Measuring rotation of a wheel or other shaft

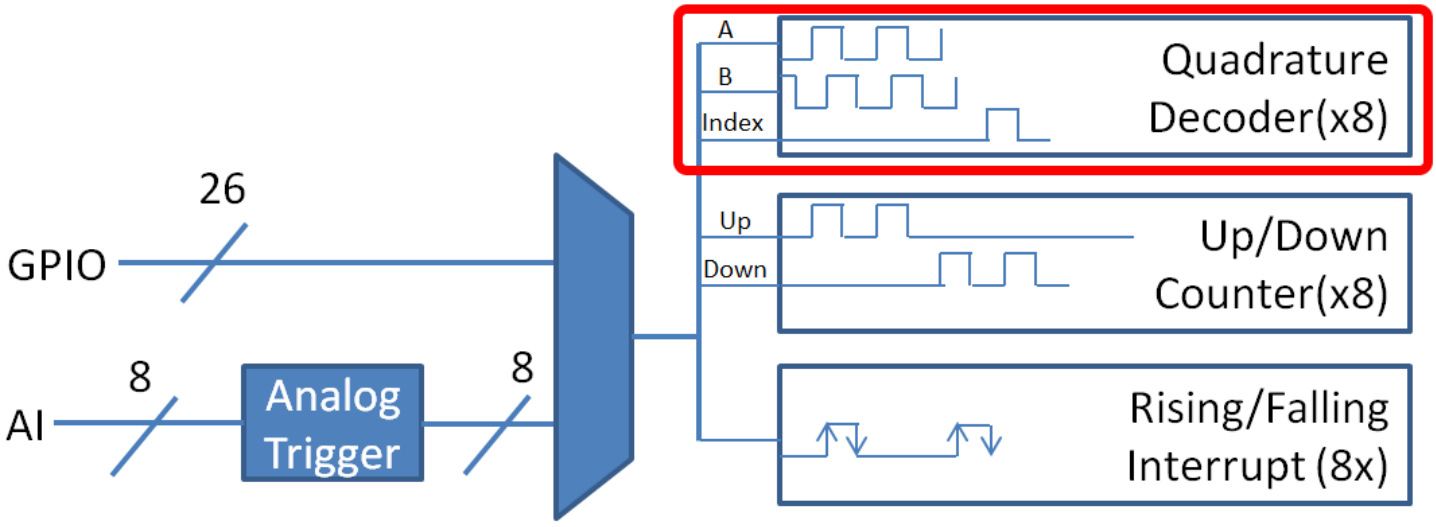
Encoders are devices for measuring the rotation of a spinning shaft. Encoders are typically used to measure the distance a wheel has turned which can be translated into the distance the robot has traveled. The distance traveled over a measured period of time represents the speed of the robot, and is another common use for encoders. Encoders can also directly measure the rate of rotation by determining the time between pulses. This article covers the use of quadrature encoders (defined below) For non-quadrature incremental encoders, see the article on counters. For absolute encoders the appropriate article will depend on the input type (most commonly analog, I2C or SPI).

Quadrature Encoder Overview



A quadrature encoder is a device for measuring shaft rotation that consists of two sensing elements 90 degrees out of phase. The most common type of encoder typically used in FRC is an optical encoder which uses one or more light sources (LEDs) pointed at a striped or slit code wheel and two detectors 90 degrees apart (these may be located opposite the LED to detect transmission or on the same side as the LED to measure reflection). The phase difference between the signals can be used to detect the direction of rotation by determining which signal is "leading" the other.

Encoders vs. Counters



The FRC FPGA has 8 Quadrature decoder modules which can do 4x decoding of a 2 channel quadrature encoder signal. This means that the module is counting both the rising and falling edges of each pulse on each of the two channels to yield 4 ticks for every stripe on the codewheel. The quadrature decoder module is also capable of handling an index channel which is a feature on some encoders that outputs one pulse per revolution. The counter FPGA modules are used for 1x or 2x decoding where the rising or rising and falling edges of one channel are counted and the second channel is used to determine direction. In either case it is recommended to use the Encoder class for all quadrature encoders, the class will assign the appropriate FPGA module based on the encoding type you choose.

Sampling Modes

The encoder class has 3 sampling modes: 1x, 2x and 4x. The 1x and 2x mode count the rising or the rising and falling edges respectively on a single channel and use the B channel to determine direction only. The 4x mode counts all 4 edges on both channels. This means that the 4x mode will have a higher positional accuracy (4 times as many ticks per rotation as 1x) but will also have more jitter in the rate output due to mechanical deficiencies (imperfect phase difference, imperfect striping) as well as running into the timing limits of the FPGA. For sensing rate, particularly at high RPM, using 1x or 2x decoding and increasing the number of samples to average may substantially help reduce jitter. Also keep in mind that the FPGA has 8 quadrature decoding modules (used for 4x decoding) and 8 counter modules (used for 1x and 2x decoding as well as Counter objects).

Constructing an Encoder object

C++
```
Encoder *enc;
enc = new Encoder(0, 1, false, Encoder::EncodingType::k4X);
```

Java
```
Encoder enc;
enc = new Encoder(0, 1, false, Encoder.EncodingType.k4X);
```

There are a number of constructors you may use to construct encoders, but the most common is shown above. In the example, 0 and 1 are the port numbers for the two digital inputs and false tells the encoder to not invert the counting direction. The sensed direction could depend on how the encoder is mounted relative to the shaft being measured. The k4X makes sure that an encoder module from the FPGA is used and 4X accuracy is obtained.

Setting Encoder Parameters

C++
```
Encoder *sampleEncoder = new Encoder(0, 1, false, Encoder::EncodingType::k4X);
sampleEncoder->SetMaxPeriod(.1);
sampleEncoder->SetMinRate(10);
sampleEncoder->SetDistancePerPulse(5);
sampleEncoder->SetReverseDirection(true);
sampleEncoder->SetSamplesToAverage(7);
```

Java
```
Encoder sampleEncoder = new Encoder(0, 1, false, Encoder.EncodingType.k4X);
sampleEncoder.setMaxPeriod(.1);
sampleEncoder.setMinRate(10);
sampleEncoder.setDistancePerPulse(5);
sampleEncoder.setReverseDirection(true);
sampleEncoder.setSamplesToAverage(7);
```

The following parameters of the encoder class may be set through the code:

- Max Period - The maximum period (in seconds) where the device is still considered moving. This value is used to determine the state of the getStopped() method and effect the output of the getPeriod() and getRate() methods. This is the time between pulses on an individual channel (scale factor is accounted for). It is recommended to use the Min Rate parameter

instead as it accounts for the distance per pulse, allowing you to set the rate in engineering units.
- Min Rate - Sets the minimum rate before the device is considered stopped. This compensates for both scale factor and distance per pulse and therefore should be entered in engineering units (RPM, RPS, Degrees/sec, In/s, etc)
- Distance Per Pulse - Sets the scale factor between pulses and distance. The library already accounts for the decoding scale factor (1x, 2x, 4x) separately so this value should be set exclusively based on the encoder's Pulses per Revolution and any gearing following the encoder.
- Reverse Direction - Sets the direction the encoder counts, used to flip the direction if the encoder mounting makes the default counting direction unintuitive.
- Samples to Average - Sets the number of samples to average when determining the period. Averaging may be desired to account for mechanical imperfections (such as unevenly spaced reflectors when using a reflective sensor as an encoder) or as oversampling to increase resolution. Valid values are 1 to 127 samples.

Starting, Stopping and Resetting Encoders

C++
```
Encoder *sampleEncoder = new Encoder(0, 1, false, Encoder::EncodingType::k4X);
sampleEncoder->Reset();
```

Java
```
Encoder sampleEncoder = new Encoder(0, 1, false, Encoder.EncodingType.k4X);
sampleEncoder.reset();
```

The encoder will begin counting as soon as it is created. To reset the encoder value to 0 call reset().

Getting Encoder Values

C++
```
Encoder *sampleEncoder = new Encoder(0, 1, false, Encoder::EncodingType::k4X);
int count = sampleEncoder->Get();
double distance = sampleEncoder->GetRaw();
double distance = sampleEncoder->GetDistance();
double period = sampleEncoder->GetPeriod();
double rate = sampleEncoder->GetRate();
boolean direction = sampleEncoder->GetDirection();
boolean stopped = sampleEncoder->GetStopped();
```

Java
```
Encoder sampleEncoder = new Encoder(0, 1, false, Encoder.EncodingType.k4X);
int count = sampleEncoder.get();
double distance = sampleEncoder.getRaw();
double distance = sampleEncoder.getDistance();
double period = sampleEncoder.getPeriod();
double rate = sampleEncoder.getRate();
boolean direction = sampleEncoder.getDirection();
boolean stopped = sampleEncoder.getStopped();
```

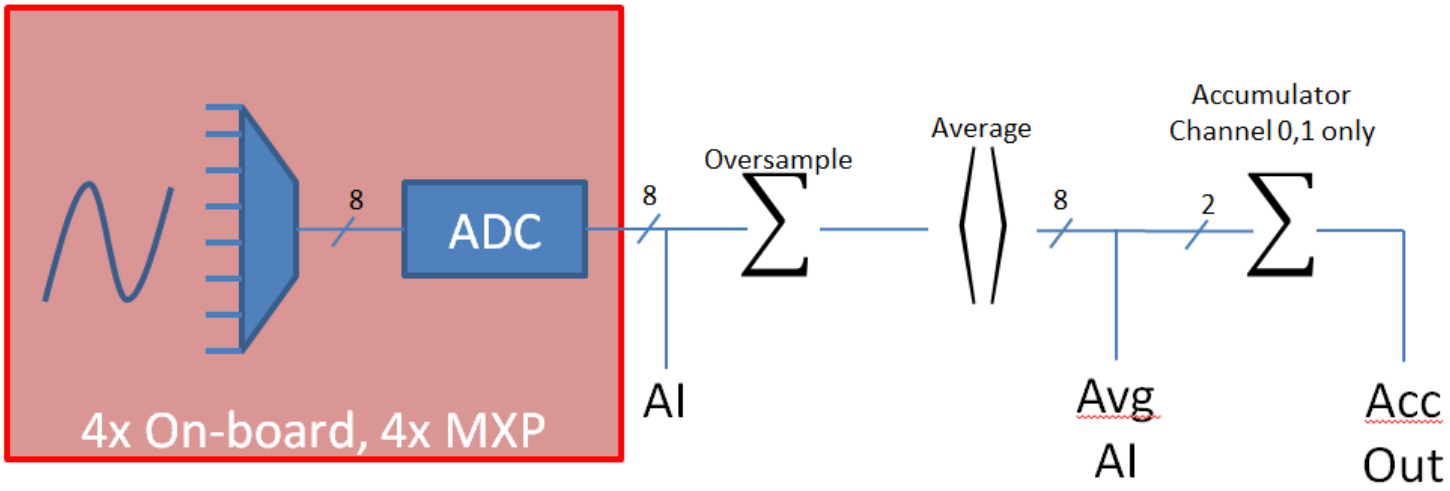
The following values can be retrieved from the encoder:

- Count - The current count. May be reset by calling reset().
- Raw Count - The count without compensation for decoding scale factor.
- Distance - The current distance reading from the counter. This is the count multiplied by the Distance Per Count scale factor.
- Period - The current period of the counter in seconds. If the counter is stopped this value may return 0. This is deprecated, it is recommended to use rate instead.
- Rate - The current rate of the counter in units/sec. It is calculated using the DistancePerPulse divided by the period. If the counter is stopped this value may return Inf or NaN, depending on language.
- Direction - The direction of the last value change (true for Up, false for Down)
- Stopped - If the counter is currently stopped (period has exceeded Max Period)

Analog inputs

The roboRIO Analog to Digital module has a number of features not available on simpler controllers. It will automatically sample the analog channels in a round robin fashion, providing a combined sample rate of 500 ks/s (500,000 samples / second). These channels can be optionally oversampled and averaged to provide the value that is used by the program. There are raw integer and floating point voltage outputs available in addition to the averaged values. The diagram below outlines this process.

Analog System Diagram



When the system averages a number of samples, the division results in a fractional part of the answer that is lost in producing the integer valued result. Oversampling is a technique where extra samples are summed, but not divided down to produce the average. Suppose the system were oversampling by 16 times – that would mean that the values returned were actually 16 times the average. Using the oversampled value gives additional precision in the returned value.

Constructing an Analog Input

C++
```
AnalogInput *ai;
ai = new AnalogInput(0);
```

Java
```
AnalogInput ai;
ai = new AnalogInput(0);
```

To construct an AnalogInput object, simply pass in the channel number for the desired input.

Oversampling and Averaging

$$Oversample\ (AI_O) = \sum_{0}^{2^{N-1}} AI$$

Where N is the number of Oversample bits

$$Average = \left(\sum_{0}^{2^{M}-1} AI_O\right) \Big/ 2^M$$

Where M is the number of Average bits

$$f_{avg} = \frac{f_s}{2^{(M+N)}}$$

Where f_s is the original sampling frequency

The number of averaged and oversampled values are always powers of two (number of bits of oversampling/averaging). Therefore the number of oversampled or averaged values is two ^ bits, where 'bits' is passed to the methods: SetOversampleBits(bits) and SetAverageBits(bits). The actual rate that values are produced from the analog input channel is reduced by the number of averaged and oversampled values. For example, setting the number of oversampled bits to 4 and the average bits to 2 would reduce the number of delivered samples by 16x and 4x, or 64x total.

Code example

C++
```
AnalogInput *exampleAnalog = new AnalogInput(0);
int bits;
exampleAnalog->SetOversampleBits(4);
bits = exampleAnalog->GetOversampleBits();
exampleAnalog->SetAverageBits(2);
bits = exampleAnalog->GetAverageBits();
```

Java
```
AnalogInput exampleAnalog = new AnalogInput(0);
int bits;
exampleAnalog.setOversampleBits(4);
bits = exampleAnalog.getOversampleBits();
```

```
exampleAnalog.setAverageBits(2);
bits = exampleAnalog.getAverageBits();
```

The above code shows an example of how to get and set the number of oversample bits and average bits on an analog channel

Sample Rate

C++
```
AnalogInput::SetSampleRate(62500);
```

Java
```
AnalogInput.setGlobalSampleRate(62500);
```

The sample rate is fixed per analog I/O module, so all the channels on a given module must sample at the same rate. However, the averaging and oversampling rates can be changed for each channel. The use of some sensors (currently just the Gyro) will set the sample rate to a specific value for the module it is connected to. The example above shows setting the sample rate for a module to the default value of 62,500 samples per channel per second (500kS/s total).

Reading Analog Values

C++
```
AnalogInput *exampleAnalog = new AnalogInput(0);
int raw = exampleAnalog->GetValue();
double volts = exampleAnalog->GetVoltage();
int averageRaw = exampleAnalog->GetAverageValue();
double averageVolts = exampleAnalog->GetAverageVoltage();
```

Java
```
AnalogInput exampleAnalog = new AnalogInput(0);
int raw = exampleAnalog.getValue();
double volts = exampleAnalog.getVoltage();
int averageRaw = exampleAnalog.getAverageValue();
double averageVolts = exampleAnalog.getAverageVoltage();
```

There are a number of options for reading Analog input values from an analog channel:

1. Raw value - The instantaneous raw 12-bit (0-4096) value representing the 0-5V range of the ADC. Note that this method does not take into account the calibration information stored in the module.

2. Voltage - The instantaneous voltage value of the channel. This method takes into account the calibration information stored in the module to convert the raw value to a voltage.
3. Average Raw value - The raw, unscaled value output from the oversampling and averaging engine. See above for information on the effect of oversampling and averaging and how to set the number of bits for each.
4. Average Voltage - The scaled voltage value output from the oversampling and averaging engine. This method uses the stored calibration information to convert the raw average value into a voltage.

Accumulator

The analog accumulator is a part of the FPGA that acts as an integrator for analog signals, summing the value over time. A common example of where this behavior is desired is for a gyro. A gyro outputs an analog signal corresponding to the rate of rotation, however the measurement commonly desired is heading or total rotational displacement. To get heading from rate, you perform an integration. By performing this operation at the hardware level it can occur much quicker than if you were to attempt to implement it in the robot code. The accumulator can also apply an offset to the analog value before adding it to the accumulator. Returning to the gyro example, most gyros output a voltage of 1/2 of the full scale when not rotating and vary the voltage above and below that reference to indicate direction of rotation.

Setting up an accumulator

C++
```
AnalogInput *exampleAnalog = new AnalogInput(0);
exampleAnalog->SetAccumulatorInitialValue(0);
exampleAnalog->SetAccumulatorCenter(2048);
exampleAnalog->SetAccumulatorDeadband(10);
exampleAnalog->ResetAccumulator();
```

Java
```
AnalogInput exampleAnalog = new AnalogInput(0);
exampleAnalog.setAccumulatorInitialValue(0);
exampleAnalog.setAccumulatorCenter(2048);
exampleAnalog.setAccumulatorDeadband(10);
exampleAnalog.resetAccumulator();
```

There are two accumulators implemented in the FPGA, connected to channels 0 and 1. Any device which you wish to use with the analog accumulator must be attached to one of these two

channels. There are no mandatory parameters that must be set to use the accumulator, however depending on the device you may wish to set some or all of the following:

1. Accumulator Initial Value - This is the raw value the accumulator returns to when reset. It is added to the output of the hardware accumulator before the value is returned to the code.
2. Accumulator Center - This raw value is subtracted from each sample before the sample is applied to the accumulator. Note that the accumulator is after the oversample and averaging engine in the pipeline so oversampling will affect the appropriate value for this parameter.
3. Accumulator Deadband - The raw value deadband around the center point where the accumulator will treat the sample as 0.
4. Accumulator Reset - Resets the value of the accumulator to the Initial Value (0 by default).

Reading from an Accumulator

C++
```
AnalogInput *exampleAnalog = new AnalogInput(0);
long count = exampleAnalog->GetAccumulatorCount();
long value = exampleAnalog->GetAccumulatorValue();
AccumulatorResult *result = new AccumulatorResult();
exampleAnalog->GetAccumulatorOutput(result);
count = result->count;
value = result->value;
```

Java
```
AnalogInput exampleAnalog = new AnalogInput(0);
long count = exampleAnalog.getAccumulatorCount();
long value = exampleAnalog.getAccumulatorValue();
AccumulatorResult result = new AccumulatorResult();
exampleAnalog.getAccumulatorOutput(result);
count = result.count;
value = result.value;
```

Two separate pieces of information can be read from the accumulator in three total ways:

1. Count - The number of samples that have been added to the accumulator since the last reset.
2. Value - The value currently in the accumulator

3. Combined - Retrieve the count and value together to assure synchronization. This should be used if you are going to use the count and value in the same calculation such as averaging.

Potentiometers - Measuring joint angle or linear motion

Potentiometers are a common analog sensor used to measure absolute angular rotation or linear motion (string pots) of a mechanism. A potentiometer is a three terminal device that uses a moving contact to from a variable resistor divider. When the outer contacts are connected to 5V and ground and the variable contact is connected to an analog input, the analog input will see an analog voltage that varies as the potentiometer is turned.

Potentiometer Taper

The taper of a potentiometer describes the relationship between the position and the resistance. The two common tapers are linear and logarithmic. A linear taper potentiometer will vary the resistance proportionally to the rotation of the shaft; For example, the shaft will measure 50% of the resistive value at the midpoint of the rotation. A logarithmic taper potentiometer will vary the resistance logarithmically with the rotation of the shaft. Logarithmic potentiometers are commonly used in audio controls due to human perception of audio volume also being logarithmic.

Most or all FRC uses for potentiometers should use linear potentiometers so that angle can be deduced directly from the voltage.

Using Potentiometers with WPILib

Potentiometers can either be read with the AnalogInput class (then work with the voltage or perform the scaling in your code) or the AnalogPotentiometer class which implements the Potentiometer interface. The AnalogPotentiometer class will read the sensor ratiometrically (compensate for the Analog supply voltage) and will scale and offset the voltage to return meaningful units.

Constructing a Potentiometer

C++
```
Potentiometer *pot;
pot = new AnalogPotentiometer(0, 360, 30);
AnalogInput *ai = new AnalogInput(1);
pot = new AnalogPotentiometer(ai, 360, 30);
```

Java
```
Potentiometer pot;
```

```
pot = new AnalogPotentiometer(0, 360, 30);
AnalogInput ai = new AnalogInput(1);
pot = new AnalogPotentiometer(ai, 360, 30);
```

The Potentiometer constructor takes 3 parameters: a channel number for the analog input, a scale factor to multiply the 0-1 ratiometric value by to return useful units, and an offset to add after the scaling. Generally, the most useful scale factor will be the angular or linear full scale of the potentiometer. For example, let's say you have an ideal single turn linear potentiometer attached to a robot arm. This pot will turn 360 degrees over the 0V-5V range so using that for the scale factor will result in the output being in degrees. In order to prevent the potentiometer from breaking due to minor shifting in alignment, it may be installed with the "zero-point" of the arm a little ways into the potentiometers range, the example above represents the potentiometer being installed with an initial value of 30 degrees, which is negated using the offset so that the output is 0 at the "zero-point" of the mechanism.

You can also pass an existing AnalogInput to the constructor in place of the channel if you wish to share the input with other code.

The Calculations

The potentiometer output is calculated using the following formula: (Analog Input Voltage/Analog Supply Voltage)*FullScale + Offset. As you can see the result of the first part of the calculation is a unitless quantity in the range 0 to 1. This means the units of the output are the same as the units of the scale factor. The offset is added to the scaled quantity so it should have the same units as the scale factor.

Reading the output

C++

```
Potentiometer *pot = new AnalogPotentiometer(0, 360, 30);
double degrees = pot->Get();
```

Java

```
Potentiometer pot = new AnalogPotentiometer(0, 360, 30);
double degrees = pot.get()
```

To read the potentiometer output, call the Get() method.

Analog triggers

An analog trigger is a way to convert an analog signal into a digital signal using resources built into the FPGA. The resulting digital signal can then be used directly or fed into other digital components of the FPGA such as the counter or encoder modules. The analog trigger module works by comparing analog signals to a voltage range set by the code. The specific return types and meanings depend on the analog trigger mode in use.

Creating an Analog Trigger

C++
```
AnalogTrigger *trigger0 = new AnalogTrigger(0);
AnalogInput *ai1 = new AnalogInput(1);
AnalogTrigger *trigger1 = new AnalogTrigger(ai1);
```

Java
```
AnalogTrigger trigger0 = new AnalogTrigger(0);
AnalogInput ai1 = new AnalogInput(1);
AnalogTrigger trigger1 = new AnalogTrigger(ai1);
```

Constructing an analog trigger requires passing in a channel number or a created Analog Channel object.

Setting Analog Trigger Voltage Range

C++
```
AnalogTrigger *trigger = new AnalogTrigger(0);
trigger->SetLimitsRaw(2048, 3200);
trigger->SetLimitsVoltage(0, 3.4);
```

Java
```
AnalogTrigger trigger0 = new AnalogTrigger(0);
trigger.setLimitsRaw(2048, 3200);
trigger.setLimitsVoltage(0, 3.4);
```

The voltage range of the analog trigger can be set in either raw units (0 to 4096 representing 0V to 5V) or voltages. In both cases the value set does not account for oversampling, if oversampling is used the user code must perform the appropriate compensation of the trigger window before setting.

Filtering and Averaging

C++
```
AnalogTrigger *trigger = new AnalogTrigger(0);
trigger->SetAveraged(true);
trigger->SetAveraged(false);
trigger->SetFiltered(true);
```

Java
```
AnalogTrigger trigger0 = new AnalogTrigger(0);
trigger.setAveraged(true);
trigger.setAveraged(false);
trigger.setFiltered(true);
```

The analog trigger can optionally be set to use either the averaged value (output from the average and oversample engine) or a filtered value instead of the raw analog channel value. A maximum of one of these options may be selected at a time, the filter cannot be applied on top of the averaged signal.

Filtering

The filtering option of the analog trigger uses a 3-point average reject filter. This filter uses a circular buffer of the last three data points and selects the outlier point nearest the median as the output. The primary use of this filter is to reject datapoints which errantly (due to averaging or sampling) appear within the window when detecting transitions using the Rising Edge and Falling Edge functionality of the analog trigger (see below).

Analog Trigger Direct Outputs

C++
```
AnalogTrigger *trigger = new AnalogTrigger(0);
bool value;
value = trigger->GetInWindow();
value = trigger->GetTriggerState();
```

Java
```
AnalogTrigger trigger0 = new AnalogTrigger(0);
boolean value;
value = trigger.getInWindow();
value = trigger.getTriggerState();
```

The analog trigger class has two direct types of output:

- In Window - Returns true if the value is inside the range and false if it is outside (above or below)
- Trigger State - Returns true if the value is above the upper limit, false if it is below the lower limit and maintains the previous state if in between (hysteresis)

Analog Trigger Output Class

The analog trigger output class is used to represent a specific output from an analog trigger. This class is primarily used as the interface between classes such as Counter or Encoder and an Analog Trigger. When used with these classes, the class will create the AnalogTriggerOutput object automatically when passed the AnalogTrigger object.

This class contains the same two outputs as the AnalogTrigger class plus two additional options (note these options cannot be read directly as they emit pulses, they can only be routed to other FPGA modules):

- Rising Pulse - In rising pulse mode the trigger generates a pulse when the analog signal transitions directly from below the lower limit to above the upper limit. This is typically used with the rollover condition of an analog sensor such as an absolute magnetic encoder or continuous rotation potentiometer.
- Falling Pulse - In falling pulse mode the trigger generates a pulse when the analog signal transitions directly from above the upper limit to below the lower limit. This is typically used with the rollover condition of an analog sensor such as an absolute magnetic encoder or continuous rotation potentiometer.

Operating the robot with feedback from sensors (PID control)

Without feedback the robot is limited to using timing to determine if it's gone far enough, turned enough, or is going fast enough. And for mechanisms, without feedback it's almost impossible to get arms at the right angle, elevators at the right height, or shooters to the right speed. There are a number of ways of getting these mechanisms to operate in a predictable way. The most common is using PID (Proportional, Integral, and Differential) control. The basic idea is that you have a sensor like a potentiometer or encoder that can measure the variable you're trying to control with a motor. In the case of an arm you might want to control the angle - so you use a potentiometer to measure the angle. The potentiometer is an analog device, it returns a voltage that is proportional to the shaft angle of the arm.

To move the arm to a preset position, say for scoring, you predetermine what the potentiometer voltage should be at that preset point, then read the arms current angle (voltage). The different between the current value and the desired value represents how far the arm needs to move and is called the error. The idea is to run the motor in a direction that reduces the error, either clockwise or counterclockwise. And the amount of error (distance from your setpoint) determines how fast the arm should move. As it gets closer to the setpoint, it slows down and finally stops moving when the error is near zero.

The WPILib PIDController class is designed to accept the sensor values and output motor values. Then given a setpoint, it generates a motor speed that is appropriate for its calculated error value.

Creating a PIDController object

```
Joystick turretStick(1);
Jaguar turretMotor(1);
AnalogChannel turretPot(1);
PIDController turretControl(0.1, 0.001, 0.0, &turretPot, &turretMotor);

turretControl.Enable();  // start calculating PIDOutput values

while(IsOperator())
{
   turretControl.SetSetpoint((turretStick.GetX() + 1.0) * 2.5);  (1)
   Wait(.02);      // wait for new joystick values
}
```

The PIDController class allows for a PID control loop to be created easily, and runs the control loop in a separate thread at consistent intervals. The PIDController automatically checks a PIDSource for feedback and writes to a PIDOutput every loop. Sensors suitable for use with PIDController in WPILib are already subclasses of PIDSource. Additional sensors and custom feedback methods are supported through creating new subclasses of PIDSource. Jaguars and Victors are already configured as subclasses of PIDOutput, and custom outputs may also be created by sub-classing PIDOutput.

A potentiometer that turns with the turret will provide feedback of the turret angle. The potentiometer is connected to an analog input and will return values ranging from 0-5V from full clockwise to full counterclockwise motion of the turret. The joystick X-axis returns values from -1.0 to 1.0 for full left to full right. We need to scale the joystick values to match the 0-5V values from the potentiometer. This is done with the expression (1). The scaled value can then be used to change the setpoint of the control loop as the joystick is moved.

The 0.1, 0.001, and 0.0 values are the Proportional, Integral, and Differential coefficients respectively. The AnalogChannel object is already a subclass of PIDSource and returns the voltage as the control value and the Jaguar object is a subclass of PIDOutput.

The PIDController object will automatically (in the background):

- Read the PIDSource object (in this case the turretPot analog input)
- Compute the new result value
- Set the PIDOutput object (in this case the turretMotor)

This will be repeated periodically in the background by the PIDController. The default repeat rate is 50ms although this can be changed by adding a parameter with the time to the end of the PIDController argument list. See the reference document for details.

Setting the P, I, and D values

The output value is computed by adding the weighted values of the error (proportional term), the sum of the errors (integral term) and the rate of change of errors (differential term). Each of these is multiplied by a scaling constant, the P, I and D values before adding the terms together. The constants allow the PID controller to be tuned so that each term is contributing an appropriate value to the final output.

The P, I, and D values are set in the constructor for the PIDController object as parameters.

The SmartDashboard in Test mode has support for helping you tune PID controllers by displaying a form where you can enter new P, I, and D constants and test the mechanism.

Continuous sensors like continuous rotation potentiometers

The PIDController object can also handle continuous rotation potentiometers as input devices. When the pot turns through the end of the range the values go from 5V to 0V instantly. The PID controller method SetContinuous() will set the PID controller to a mode where it will computer the shortest distance to the desired value which might be through the 5V to 0V transition. This is very useful for drive trains that use have continuously rotating swerve wheels where moving from 359 degrees to 10 degrees should only be a 11 degree motion, not 349 degrees in the opposite direction.

The Feed-forward Term

The Feed-forward term, F, is used to provide a baseline value to the controller based on the setpoint instead of the error. One use of the feed-forward term is velocity control:

Controlling motor speed is a a little different then position control. Remember, with position control you are setting the motor value to something related to the error. As the error goes to zero the motor stops running. If the sensor (an optical encoder for example) is measuring motor speed as the speed reaches the setpoint, the error goes to zero, and the motor slows down. This causes the motor to oscillate as it constantly turns on and off. What is needed is a base value of motor speed called the "Feed-forward" term. The feed-forward constant, F, is multiplied by the setpoint to provide a baseline value. This 4th value, F, is added in to the output motor voltage independently of the P, I, and D calculations and is a base speed the motor will run at. The P, I, and D values adjust the feed forward term (base motor speed) rather than directly control it. The closer the feed forward term is, the smoother the motor will operate.

Note: The feedforward term is multiplied by the setpoint for the PID controller so that it scales with the desired output speed.

Using PID controllers in command based robot programs

```
1  package org.usfirst.frc190.GearsBot.subsystems;
2  import edu.wpi.first.wpilibj.*;
3  import edu.wpi.first.wpilibj.command.PIDSubsystem;
4  import edu.wpi.first.wpilibj.livewindow.LiveWindow;
5  import org.usfirst.frc190.GearsBot.RobotMap;
6
7  public class Elevator extends PIDSubsystem {
8      SpeedController motor = RobotMap.elevatorMotor;
9      AnalogChannel pot = RobotMap.elevatorPot;
10
11     public Elevator() {
12         super("Elevator", 1.0, 0.0, 0.0);
13         setAbsoluteTolerance(0.2);
14         getPIDController().setContinuous(false);
15         LiveWindow.addActuator("Elevator", "PIDSubsystem Controller", getPIDController());
16     }
17
19     public void initDefaultCommand() {
20     }
21
22     protected double returnPIDInput() {
23         return pot.getAverageVoltage();
24     }
25
26     protected void usePIDOutput(double output) {
27         motor.pidWrite(output);
28     }
29 }
```

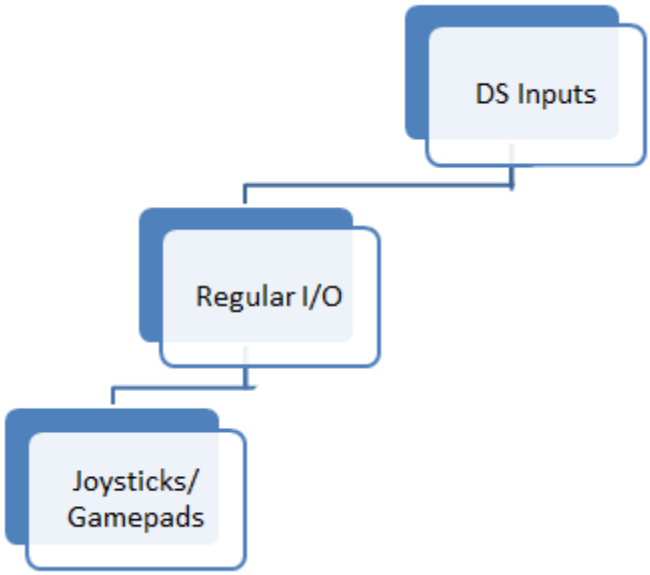
The easiest way to use PID controllers with command based robot programs is by implementing PIDSubsystems for all your robot mechanisms. This is simply a subsystem with a PIDController object built-in and provides a number of convenience methods to access the required PID parameters. In a command based program, typically commands would provide the setpoint for different operations, like moving an elevator to the low, medium or high position. In this case, the isFinished() method of the command would check to see if the embedded PIDController had reached the target. See the Command based programming section for more information and examples.

Driver Station Inputs and Feedback

Driver Station Input Overview

The FRC Driver Station software serves as the interface between the human operators and the robot. The software takes input from a number of sources and forwards it to the robot where the robot code can act on it to control mechanisms.

Input types



The chart above shows the different types of inputs that may be transmitted by the DS software. The most common input is an HID compliant joystick or gamepad such as the Logitech Attack 3 or Logitech Extreme 3D Pro joysticks which have been provided in the Kit of Parts since 2009. Note that a number of devices are now available which allow custom IO to be exposed as a standard USB HID device such as the The TI Launchpad and 16 Hertz Leonardo++ included in your Kit of Parts.

Driver Station Class

C++
```
DriverStation& ds = DriverStation::GetInstance();
ds.SomeMethod();

DriverStation::GetInstance().SomeMethod();
```

Java
```
DriverStation ds = DriverStation.getInstance();
ds.someMethod();

DriverStation.getInstance.someMethod();
```

The Driver Station class has methods to access information such as the robot mode, battery voltage, alliance color and team number. Note that while the Driver Station class has methods for accessing the joystick data, there is another class "Joystick" that provides a much more user friendly interface to this data. The DriverStation class is constructed as a singleton by the base class. To get access to the methods of the DriverStation object constructed by the base class, call DriverStation.getInstance() and either store the result in a DriverStation object (if using a lot) or call the method on the instance directly.

Robot Mode

C++
```
bool exampleBool;
exampleBool = IsDisabled();
exampleBool = IsEnabled();
exampleBool = IsAutonomous();
exampleBool = IsOperatorControl();
exampleBool = IsTest();

while(IsOperatorControl() && IsEnabled())
{
}

exampleBool = DriverStation::GetInstance()->IsDisabled();
```

Java
```
boolean exampleBool;
exampleBool = isDisabled();
exampleBool = isEnabled();

exampleBool = isAutonomous();
exampleBool = isOperatorControl();
exampleBool = isTest();

while(isOperatorControl() && isEnabled())
{
}
```

```
exampleBool = DriverStation.getInstance().isDisabled();
```

The Driver Station class provides a number of methods for checking the current mode of the robot, these methods are most commonly used to control program flow when using the SampleRobot base class. There are two separate pieces of information that define the current mode, the enable state (enabled/disabled) and the control state(autonomous, operator control, test). This means that exactly one method from the first group and one method from the second group should always return true. For example, if the Driver Station is currently set to Test mode and the robot is disabled the methods isDisabled() and isTest() would both return true. While the implementation of these methods is in the DriverStation class, the RobotBase class (which the templates extend from) provides proxies to these methods so they may be used without the class specification (as shown in the first 3 example groups above). To call these methods from another class, use the DriverStation instance as shown in the final example.

DS Attached, FMS Attached and System status

C++
```
bool exampleBool;
exampleBool = DriverStation::GetInstance().IsDSAttached();
exampleBool = DriverStation::GetInstance().IsFMSAttached();
exampleBool = DriverStation::GetInstance().IsSysActive();
exampleBool = DriverStation::GetInstance().IsSysBrownedOut();
```

Java
```
boolean exampleBool;
exampleBool = DriverStation.getInstance().isDSAttached();
exampleBool = DriverStation.getInstance().isFMSAttached();
exampleBool = DriverStation.getInstance().isSysActive();
exampleBool = DriverStation.getInstance().isSysBrownedOut();
```

The DriverStation class also has methods for determining if the DS is connected to the robot, if the DS is connected to FMSquerying if the FPGA outputs are enabled (IsSysActive), and querying if the roboRIO is in brownout protection. The FPGA outputs may be disabled for a variety of reasons including: DS is commanding Disabled or E-Stop, the system watchdog has timed out (generally because the DS is not communicating with the roboRIO), or the roboRIO is in brownout protection.

Battery Voltage

C++
```
double voltage = DriverStation::GetInstance().GetBatteryVoltage();
```

Java
```
double voltage = DriverStation.getInstance().getBatteryVoltage();
```

For compatibility purposes the battery voltage can be retrieved using the DriverStation class (it is now also available from the ControllerPower class as the roboRIO input voltage). This information can be queried from the DriverStation class in order to perform voltage compensation or actively manage robot power draw by detecting battery voltage dips and shutting off or limiting non-critical mechanisms,

Alliance

C++
```
DriverStation::Alliance color;
color = DriverStation::GetInstance().GetAlliance();
if(color == DriverStation::Alliance::kBlue){
}
```

Java
```
DriverStation.Alliance color;
color = DriverStation.getInstance().getAlliance();
if(color == DriverStation.Alliance.kBlue){
}
```

The DriverStation class can provide information on what alliance color the robot is. When connected to FMS this is the alliance color communicated to the DS by the field. When not connected, the alliance color is determined by the Team Station dropdown box on the Operation tab of the DS software.

Location

C++
```
int station;
station = DriverStation::GetInstance.GetLocation();
```

Java
```
int station;
station = DriverStation.getInstance().getLocation();
```

The getLocation() method of the Driver Station returns an integer indicating which station number the Driver Station is in (1-3). Note that the station that the DS and controls are located in is not typically related to the starting position of the robot so this information may be of limited use.

When not connected to the FMS software this state is determined by the Team Station dropdown on the DS Operation tab.

Match Time

C++
```
double time;
time = DriverStation::GetInstance.GetMatchTime();
```

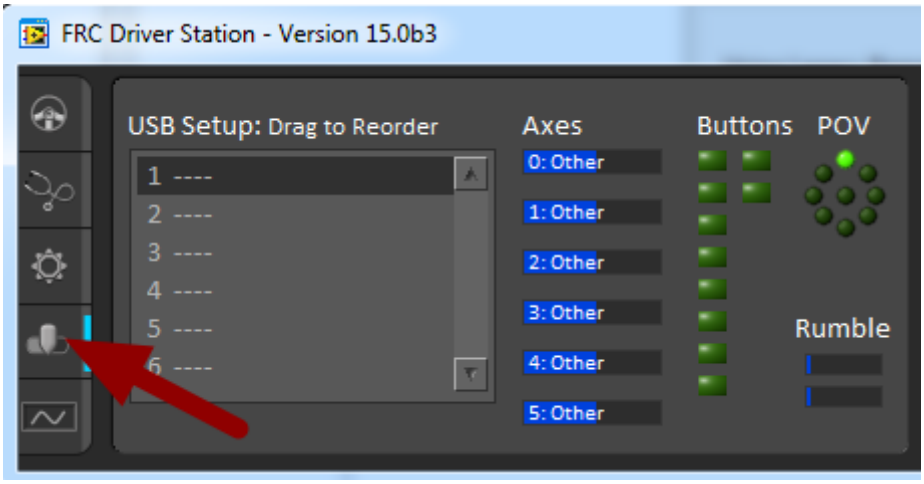
Java
```
double time;
time = DriverStation.getInstance().getMatchTime();
```

This method returns the approximate time remaining in the current period (auto, teleop, etc.) in seconds. Note that this time is derived from the FMS, however due to various latencies involved it is not an official timer. The Driver Station's Practice Match functionality will approximate the behavior of this method when connected to FMS. Running the DS directly in Autonomous or Teleop mode will behave differently with respect to this method.

Joysticks

The standard input device supported by the WPI Robotics Library is a USB joystick or gamepad. The Logitech Attack 3 joystick provided in the KOP from 2009-2012 comes equipped with eleven digital input buttons and three analog axes, and interfaces with the robot through the Joystick class. The Joystick class itself supports joysticks with more capabilities as well such as the Logitech Extreme 3D Pro included in the 2013 KOP which has 4 analog axes and 12 buttons. Note that the rest of this article exclusively uses the term joystick but can also be referring to a HID compliant USB gamepad.

USB connection



The joystick must be connected to one of the available USB ports on the driver station. The startup routine will read whatever position the joysticks are in as the center position, therefore, when the station is turned on the joysticks must be at their center position. In general the Driver Station software will try to preserve the ordering of devices between runs but it is a good idea to note what order your devices should be in and check each time you start the Driver Station software that they are correct. This can be done by selecting the USB Devices tab and viewing the order in the USB Setup box on the left hand side. Pressing a button on a joystick will cause its entry in the table to light up blue and have asterisks appear after the name. To reorder the joysticks simply click and drag.

Joystick Refresh

When the Driver Station is in disabled mode it is routinely looking for status changes on the joystick devices, unplugged devices are removed from the list and new devices are opened and

added. When not connected to the FMS, unplugging a joystick will force the Driver Station into disabled mode. To start using the joystick again plug the joystick back in, check that it shows up in the right spot, then re-enable the robot. While the Driver Station is in enabled mode it will not scan for new devices as this is a time consuming operation and timely update of signals from attached devices takes priority.

When the robot is connected to the Field Management System at competition the Driver Station mode is dictated by the FMS. This means that you cannot disable your robot and the DS cannot disable itself in order to detect joystick changes. A manual complete refresh of the joysticks can be initiated by pressing the F1 key on the keyboard. Note that this will close and re-open all devices so all devices should be in their center position as noted above.

Constructing a Joystick Object

C++

```
Joystick *exampleStick

public:
     Robot(){
     }
     void RobotInit() {
          exampleStick = new Joystick(1);
     }
```

Java

```
exampleStick = new Joystick(1);
```

The primary constructor for the Joystick class takes a single parameter representing the port number of the Joystick, this is the number (1-6) next to the joystick in the Driver Station software's Joystick Setup box (shown in the first image). There is also a constructor which takes additional

parameters of the number of axes and buttons and can be used with the get and set axis channel methods to create subclasses of Joystick to use with specific devices.

Accessing Joystick Values - Option 1

C++

```
double value;
value = exampleStick->GetX();
value = exampleStick->GetY();
value = exampleStick->GetZ();
value = exampleStick->GetThrottle();
value = exampleStick->GetTwist();

boolean buttonValue;
buttonValue = exampleStick->GetTop();
buttonValue = exampleStick->GetTrigger();
```

Java

```
double value;
value = exampleStick.getX();
value = exampleStick.getY();
value = exampleStick.getZ();
value = exampleStick.getThrottle();
value = exampleStick.getTwist();

boolean buttonValue;
buttonValue = exampleStick.getTop();
buttonValue = exampleStick.getTrigger();
```

There are two ways to access the current values of a joystick object. The first way is by using the set of named accessor methods or the getAxis method. The Joystick class contains the default mapping of these methods to the proper axes of the joystick for the KOP joystick. If you are using a another device you can subclass Joystick and use the setAxisChannel method to set the proper mappings if you wish to use these methods. Note that there are only named accessor methods for 5 of the 6 possible axes and 2 of the possible twelve buttons, if you need access to other axes or buttons, see Option 2 below.

Joystick axes return a scaled value in the range 1,-1 and buttons return a boolean value indicating their triggered state. Note that the typical convention for joysticks and gamepads is for Y to be negative as they joystick is pushed away from the user, "forward", and for X to be positive as the joystick is pushed to the right. To check this for a given device, see the section below on "Determining Joystick Mapping".

Accessing Joystick Values - Option 2

C++

```
double value;
value = exampleStick->GetRawAxis(2);

boolean buttonValue;
buttonValue = exampleStick->GetRawButton(1);
```

Java

```
double value;
value = exampleStick.getRawButton(1);

boolean buttonValue;
buttonValue = exampleStick.getRawButton(2);
```

The second way to access joystick values is to use the methods getRawAxis() and getRawButton(). These methods take an integer representing the axis or button number as a parameter and return the corresponding value. For a method to determine the mapping between the physical axes and buttons of your device and the appropriate channel number see the section "Determining Joystick Mapping" below.

Polar methods

C++

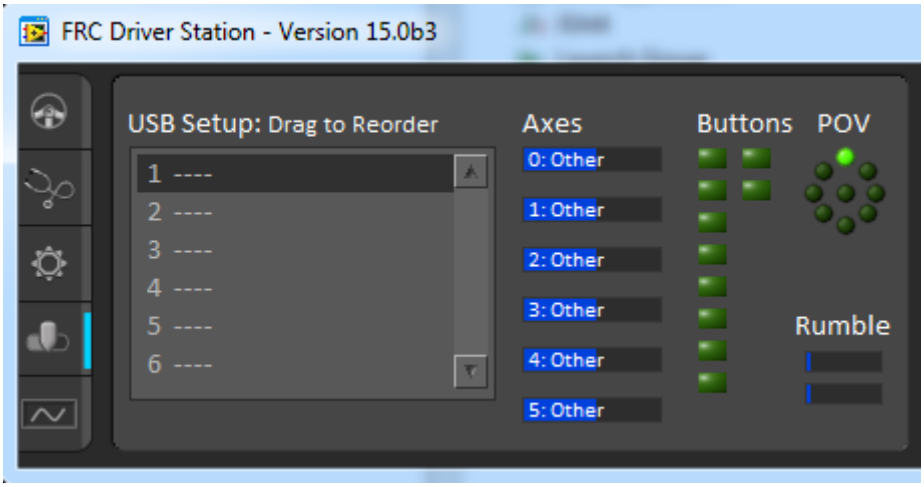
```
double value;
value = exampleStick->GetDirectionDegrees();
value = exampleStick->GetDirectionRadians();
value = exampleStick->GetMagnitude();
```

Java

```
double value;
value = exampleStick.getDirectionDegrees();
value = exampleStick.getDirectionRadians();
value = exampleStick.getMagnitude();
```

The Joystick class also contains helper methods for converting the joystick input to a polar coordinate system. For these methods to work properly, getX and getY have to return the proper axis (remap with setChannel() if necessary).

Determining Joystick Mapping



The 2015 FRC Driver Station contains indicators of the values of axes buttons and the POV that can be used to determine the mapping between physical joystick features and axis or button numbers. Simply click the joystick in the list to select it and the indicators will begin responding to the joystick input.

Displaying Data on the DS - Dashboard Overview

Often it is desirable to get feedback from the robot back to the drivers. The communications protocol between the robot and the driver station includes provisions for sending program specific data. The program at the driver station that receives the data is called the dashboard.

Network Tables - What is it?

Network Tables is the name of the client-server protocol used to share variables across software in FRC. The robot acts as the Network Tables server and software which wishes to communicate with it connects as clients. The most common Network Tables client is the dashboard.

Smart Dashboard

The term Smart Dashboard originally referred to the Java dashboard client first released in 2011. This client used the Network Tables protocol to automatically populate indicators to match the data entered into Network Tables on the robot side. Since then the term has been blurred a bit as the LabVIEW dashboard has also converted over to using Network Tables. Additional information on SmartDashboard can be found in the SmartDashboard Manual.

More information on the LabVIEW Dashboard, including an article about using the LabVIEW Dashboard with C++ or Java code can be found in the FRC Driver Station manual.

Command based programming

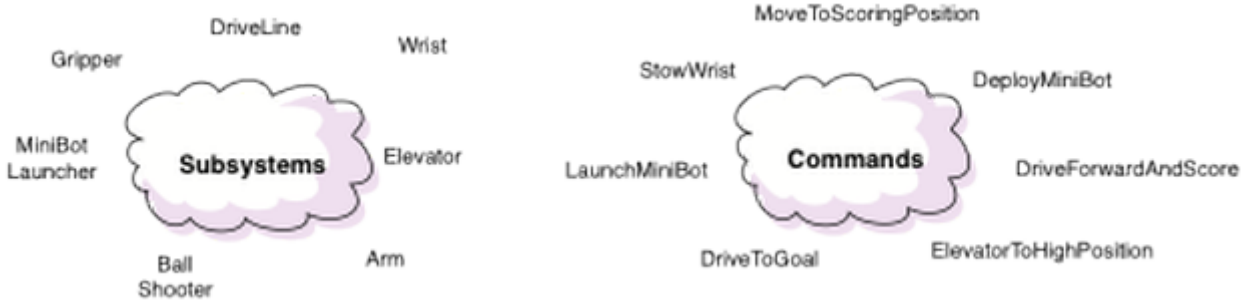
What is Command based programming?

WPILib supports a method of writing programs called "Command based programming". Command based programming is a design pattern to help you organize your robot programs. Some of the characteristics of robot programs that might be different from other desktop programs are:

- Activities happen over time, for example a sequence of steps to shoot a Frisbee or raise an elevator and place a tube on a goal.
- These activities occur concurrently, that is it might be desirable for an elevator, wrist and gripper to all be moving into a pickup position at the same time to increase robot performance.
- It is desirable to test the robot mechanisms and activities each individually to help debug your robot.
- Often the program needs to be augmented with additional autonomous programs at the last minute, perhaps at competitions, so easily extendable code is important.

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

Commands and subsystems

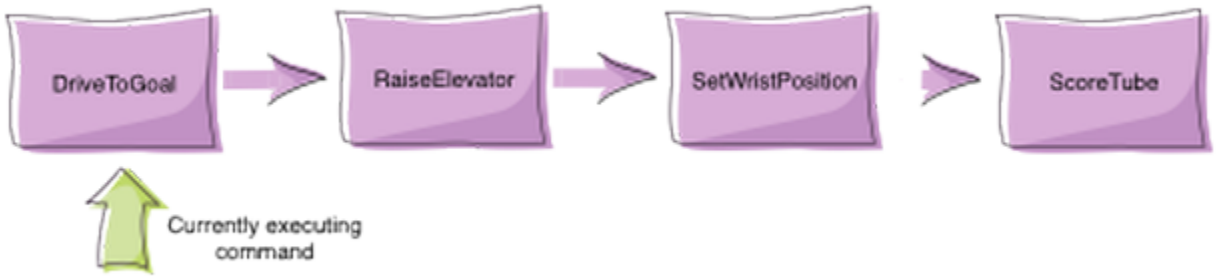


Programs based on the WPILib library are organized around two fundamental concepts: **Subsystems** and **Commands**.

Subsystems - define the capabilities of each part of the robot and are subclasses of Subsystem.

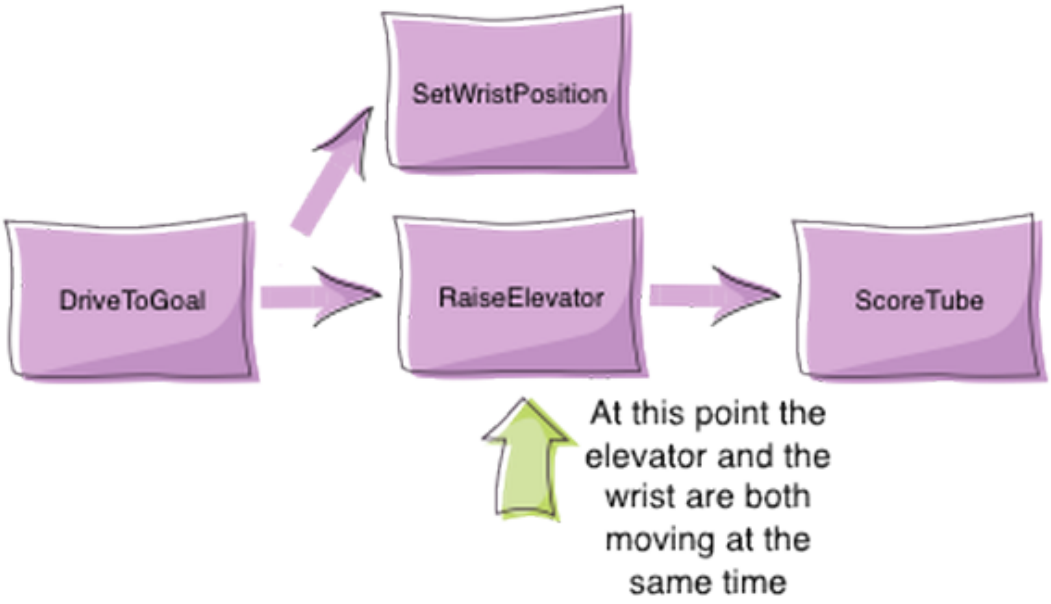
Commands - define the operation of the robot incorporating the capabilities defined in the subsystems. Commands are subclasses of Command or CommandGroup. Commands run when scheduled or in response to buttons being pressed or virtual buttons from the SmartDashboard.

How commands work



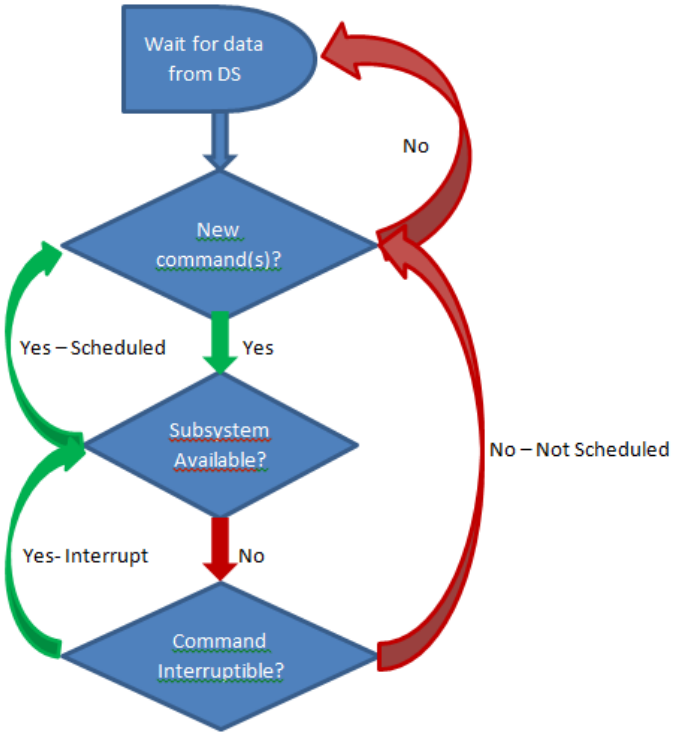
Commands let you break up the tasks of operating the robot into small chunks. Each command has an execute() method that does some work and an isFinished() method that tells if it is done. This happens on every update from the driver station or about every 20ms. Commands can be grouped together and executed sequentially, starting the next one in the group as the previous one finishes.

Concurrency



Sometimes it is desirable to have several operations happening concurrently. In the previous example you might want to set the wrist position while the elevator is moving up. In this case a command group can start a parallel command (or command group) running.

How It Works - Scheduling Commands



There are three main ways commands are scheduled:

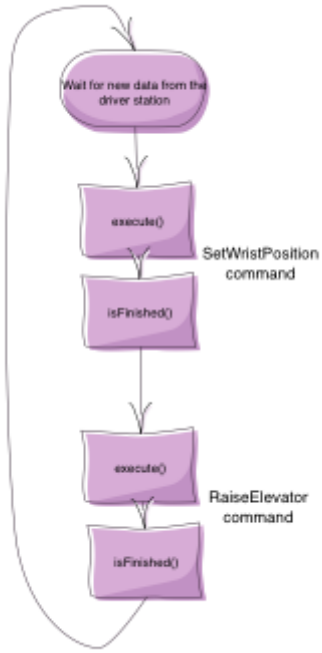
1. Manually, by calling the start() method on the command (used for autonomous)
2. Automatically by the scheduler based on button/trigger actions specified in the code (typically defined in the OI class but checked by the Scheduler).
3. Automatically when a previous command completes (default commands and command groups).

Each time the driver station gets new data, the periodic method of your robot program is called. It runs a Scheduler that checks the trigger conditions to see if any commands need to be scheduled or canceled.

When a command is scheduled, the Scheduler checks to make sure that no other commands are using the same subsystems that the new command requires. If one or more of the subsystems is

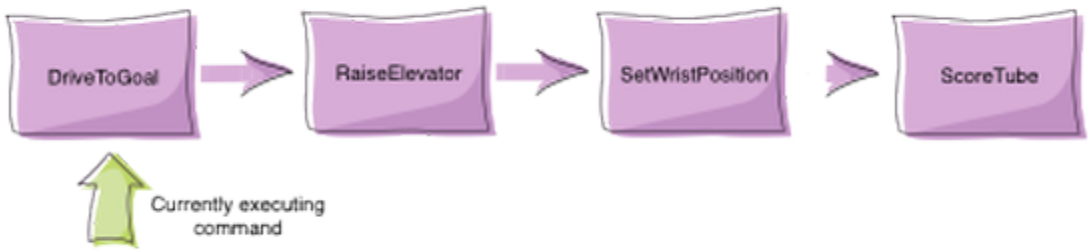
currently in use, and the current command is interruptible, it will be interrupted and the new command will be scheduled. If the current command is not interruptible, the new command will fail to be scheduled.

How It Works - Running Commands



After checking for new commands, the scheduler proceeds through the list of active commands and calls the execute() and isFinished() methods on each command. Notice that the apparent concurrent execution is done without the use of threads or tasks which would add complexity to the program. Each command simply has some code to execute (execute method) to move it further along towards its goal and a method (isFinished) that determines if the command has reached the goal. The execute and isFinished methods are just called repeatedly.

Command groups

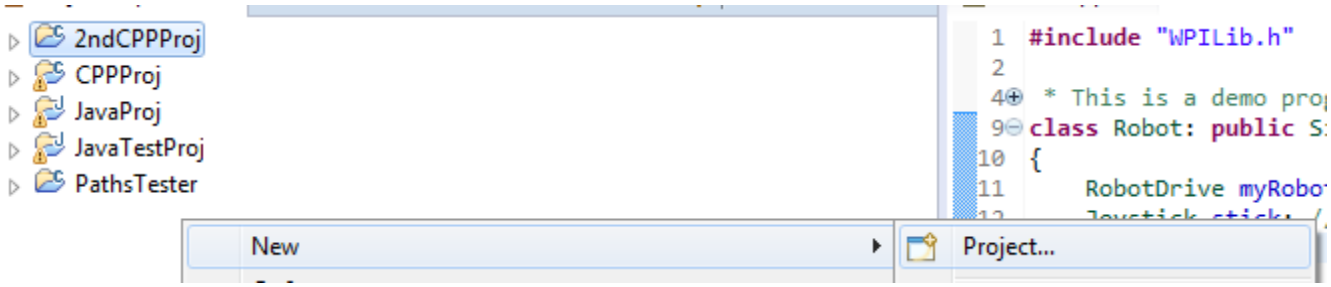


More complex commands can be built up from simpler commands. For example, shooting a disc may be a long sequence of commands that are executed one after another. Maybe some of these commands in the sequence can be executed concurrently. Command groups are commands, but instead of having an isFinished and execute method, they have a list of other commands to execute. This allows more complex operations to be built up out of simpler operations, a basic principle in programming. Each of the individual smaller commands can be easily tested first, then the group can be tested. More information on command groups can be found in the Creating groups of commands article.

Creating a robot project

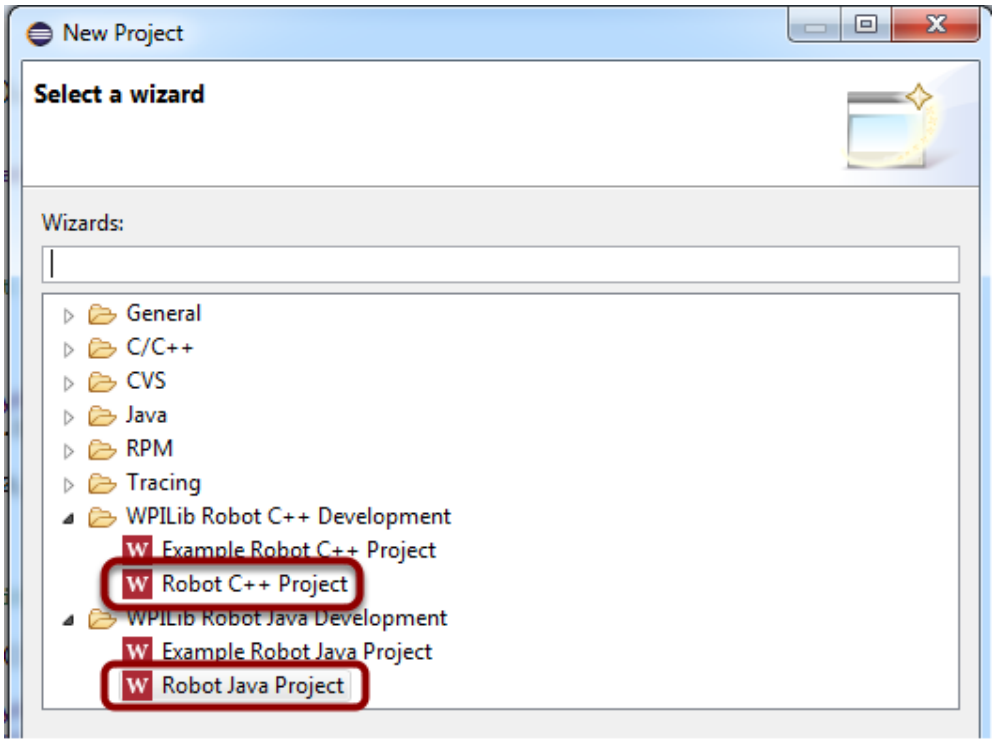
Create a command-based robot project by using one of the template projects that are provided with the Eclipse plugins.

Create the project



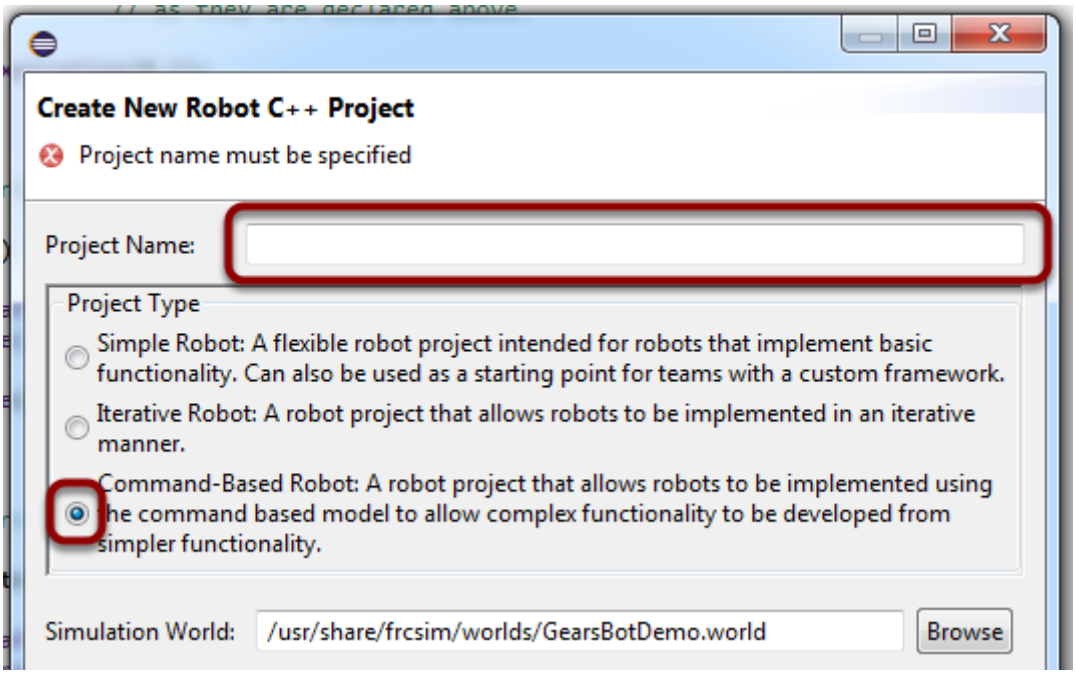
Right-click in the project explorer window in some empty space. Select "New" then "Project...".

Selecting the project type



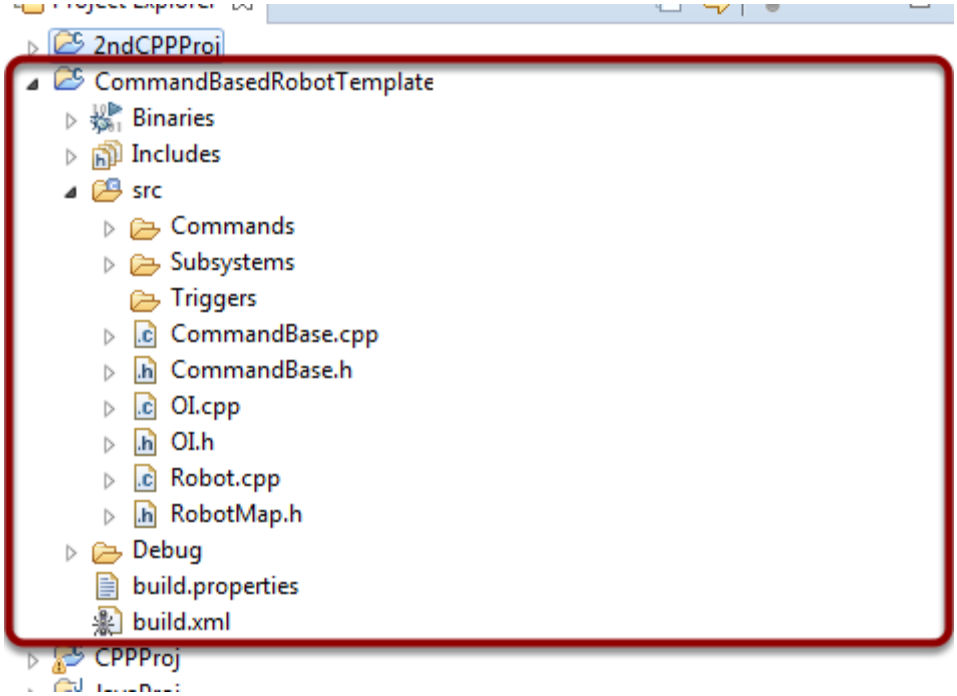
Expand the WPILib Robot C++ or Java Development folders if necessary and select the appropriate project type, Robot C++ or Java Project. You will only see the choices for the plugin you have installed.

Select the project example



Name your project in box, then select the radio button for Command-Based Robot.

Observe sample project in the Project Explorer window

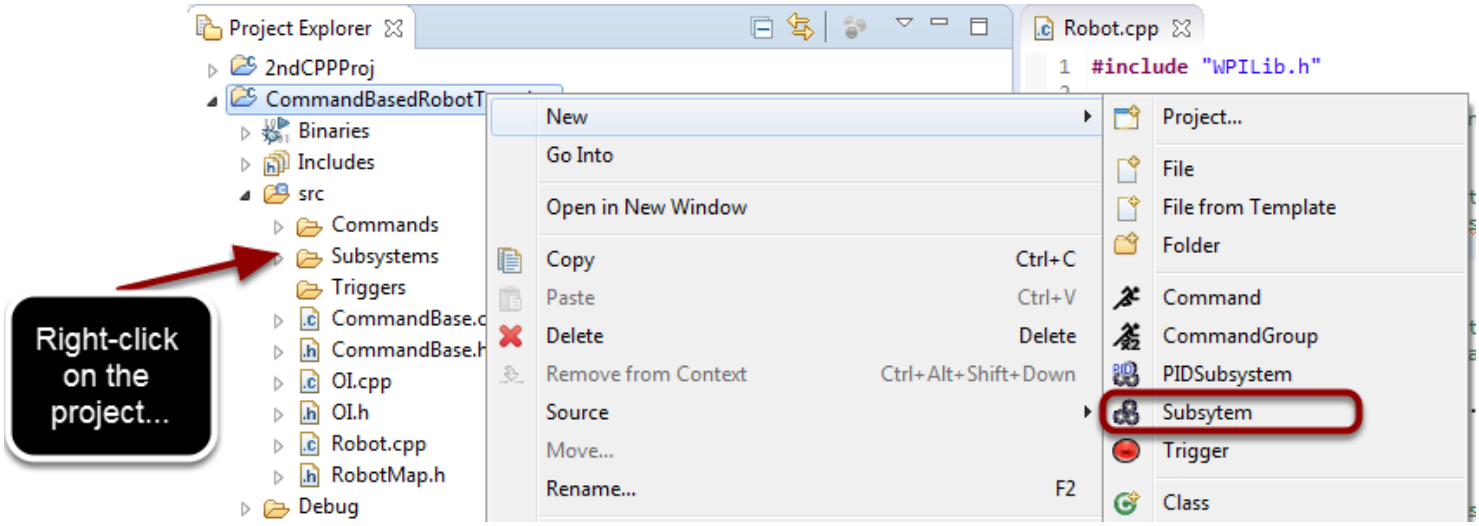


Notice that the CommandBasedRobotTemplate project has been added to the other projects that might have been in the Project Explorer window. There is a folder for Commands and another folder for Subsystems.

Adding Commands and Subsystems to the project

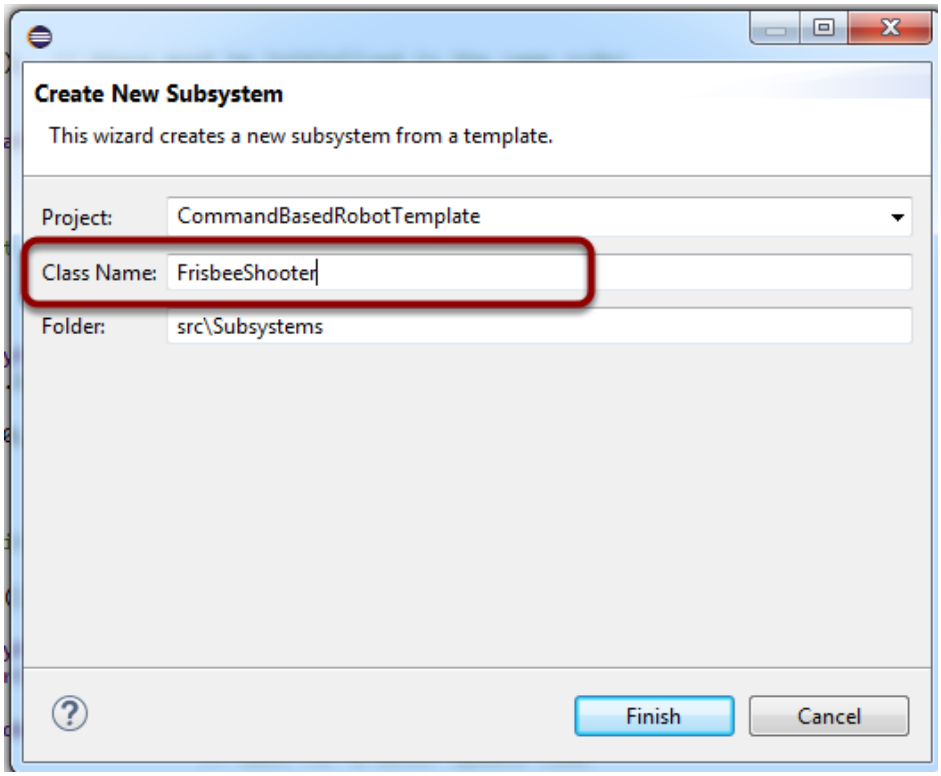
Commands and Subsystems each are created as classes. The plugin has built-in templates for both Commands and Subsystems to make it easier for you to add them to your program.

Adding subsystems to the project



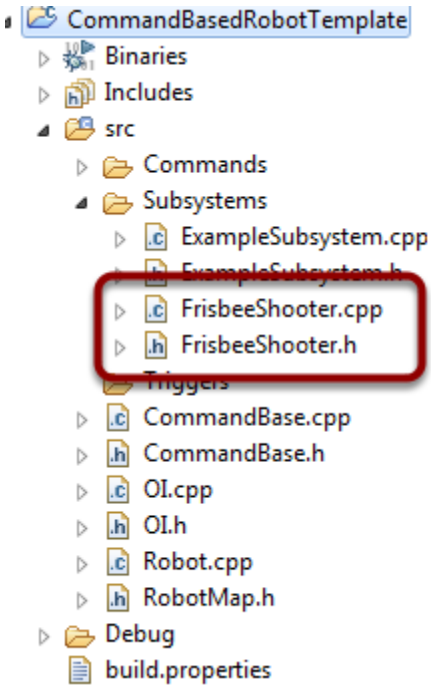
To add a subsystem, right-click on the project name and select "New" then "Subsystem" in the drop down menu.

Naming the subsystem



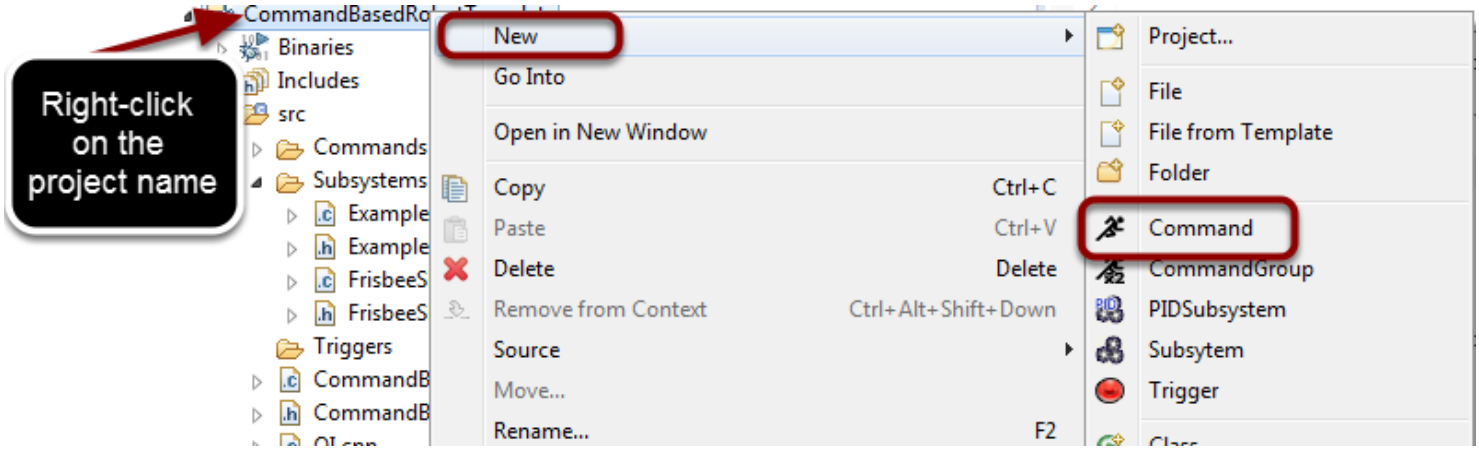
Fill in a name for the subsystem. This will become the resultant class name for the subsystem so the name has to be a valid class name for your language.

Subsystem created in project



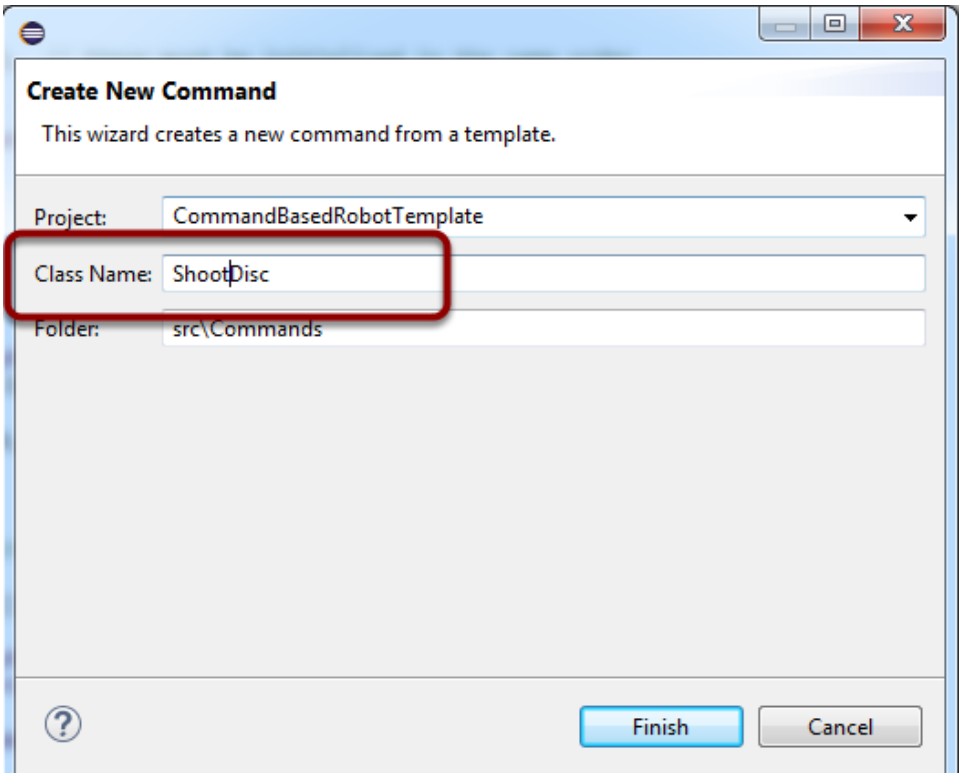
You can see the new subsystem created in the Subsystems folder in the project. To learn more about creating subsystems, see the Simple Subsystems article.

Adding a command to the project



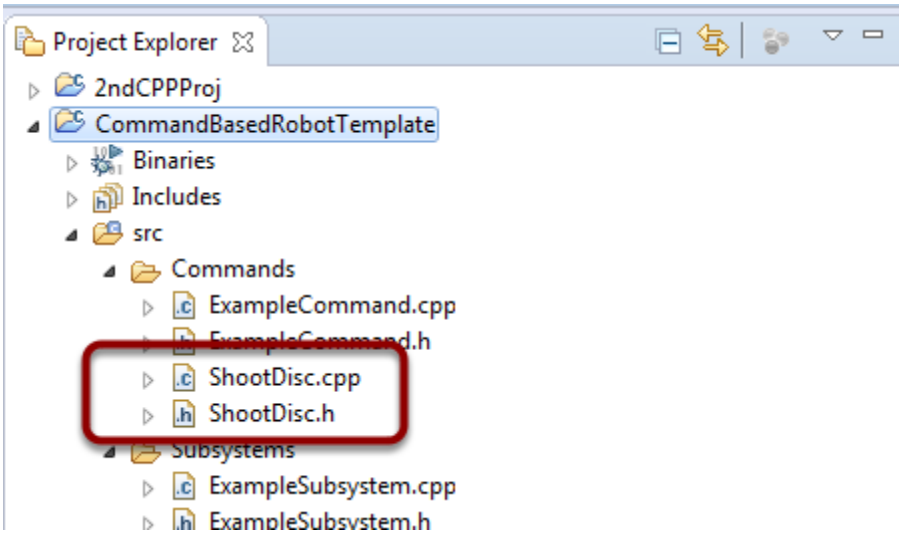
A command can be created for the project using steps similar to creating a subsystem. First right-click on the project name in the Project Explorer and select **New->Command**.

Set the command name



Enter the Command name into the "Class Name" field in the dialog box. This will be the class name for the Command so it must be a valid class name for your language.

Command created in the project



You can see that the Command has been created in the Commands folder in the project in the Project Explorer window. To learn more about creating commands, see the Creating Simple Commands article.

Simple subsystems

Subsystems are the parts of your robot that are independently controlled like collectors, shooters, drive bases, elevators, arms, wrists, grippers, etc. Each subsystem is coded as an instance of the Subsystem class. Subsystems should have methods that define the operation of the actuators and sensors but not more complex behavior that happens over time.

Creating a subsystem

Java

```
import edu.wpi.first.wpilibj.*;
import edu.wpi.first.wpilibj.command.Subsystem;
import org.usfirst.frc.team1.robot.RobotMap;

public class Claw extends Subsystem {

        Victor motor = RobotMap.clawMotor;

    public void initDefaultCommand() {
    }

    public void open() {
            motor.set(1);
    }

    public void close() {
            motor.set(-1);
    }

    public void stop() {
            motor.set(0);
    }
}
```

This is an example of a fairly straightforward subsystem that operates a claw on a robot. The claw mechanism has a single motor to open or close the claw and no sensors (not necessarily a good idea in practice, but works for the example). The idea is that the open and close operations are simply timed. There are three methods, open(), close(), and stop() that operate the claw motor. Notice that there is not specific code that actually checks if the claw is opened or closed. The open method gets the claw moving in the open direction and the close method gets the claw moving in the close direction. Use a command to control the timing of this operation to make sure that the claw opens and closes for a specific period of time.

Operating the claw with a command

Java

```
package org.usfirst.frc.team1.robot.commands;

import edu.wpi.first.wpilibj.command.Command;
import org.usfirst.frc.team1.robot.Robot;
/**
 *
 */
public class OpenClaw extends Command {

    public OpenClaw() {
        requires(Robot.claw);
        setTimeout(.9);
    }

    protected void initialize() {
            Robot.claw.open()
    }

    protected void execute() {
    }

    protected boolean isFinished() {
```

```
        return isTimedOut();
    }

    protected void end() {
            Robot.claw.stop();
    }

    protected void interrupted() {
            end();
    }
}
```

Commands provide the timing of the subsystems operations. Each command would do a different operation with the subsystem, the Claw in this case. The commands provides the timing for opening or closing. Here is an example of a simple Command that controls the opening of the claw. Notice that a timeout is set for this command (0.9 seconds) to time the opening of the claw and a check for the time in the isFinished() method. You can find more details in the article about using commands.

PIDSubsystems for built-in PID control

If a mechanism uses a sensor for feedback then most often a PID controller will be used to control the motor speed or position. Examples of subsystems that might use PID control are: elevators with potentiometers to track the height, shooters with encoders to measure the speed, wrists with potentiometers to measure the joint angle, etc.

There is a PIDController class built into WPILib, but to simplify its use for command based programs there is a PIDSubsystem. A PIDSubsystem is a normal subsystem with the PIDController built in and exposes the required methods for operation.

A PIDSubsystem to control the angle of a wrist joint

In this example you can see the basic elements of a PIDSubsystem for the wrist joint:

Java

```
package org.usfirst.frc.team1.robot.subsystems;
import edu.wpi.first.wpilibj.*;
import edu.wpi.first.wpilibj.command.PIDSubsystem;
import org.usfirst.frc.team1.robot.RobotMap;


public class Wrist extends PIDSubsystem { // This system extends PIDSubsystem

        Victor motor = RobotMap.wristMotor;
        AnalogInput pot = RobotMap.wristPot();

        public Wrist() {
                super("Wrist", 2.0, 0.0, 0.0);// The constructor passes a name for the
subsystem and the P, I and D constants that are sueed when computing the motor output
                setAbsoluteTolerance(0.05);
                getPIDController().setContinuous(false);
        }

    public void initDefaultCommand() {
    }
```

```
    protected double returnPIDInput() {
            return pot.getAverageVoltage(); // returns the sensor value that is providing
the feedback for the system
    }

    protected void usePIDOutput(double output) {
            motor.pidWrite(output); // this is where the computed output value fromthe
PIDController is applied to the motor
    }
}
```

Creating Simple Commands

This article describes the basic format of a Command and walks through an example of creating a command to drive your robot with Joysticks.

Basic Command Format

To implement a command, a number of methods are overridden from the WPILib Command class. Most of the methods are boiler plate and can often be ignored, but are there for maximum flexibility when you need it. There a number of parts to this basic command class:

C++

```
#include "MyCommandName.h"

/*
         * 1.        Constructor - Might have parameters for this command such as target
positions of devices. Should also set the name of the command for debugging purposes.
         *  This will be used if the status is viewed in the dashboard. And the command
should require (reserve) any devices is might use.
         */
MyCommandName::MyCommandName() : CommandBase("MyCommandName")
{
        Requires(Elevator);
}

// initialize() - This method sets up the command and is called immediately before the
command is executed for the first time and
// every subsequent time it is started . Any initialization code should be here.
void MyCommandName::Initialize()
{

}

/*
 *        execute() - This method is called periodically (about every 20ms) and does the
work of the command. Sometimes, if there is a position a
 *  subsystem is moving to, the command might set the target position for the subsystem in
```

```
initialize() and have an empty execute() method.
 */
void MyCommandName::Execute()
{

}

bool MyCommandName::IsFinished()
{
        return false;
}

void MyCommandName::End()
{

}

// Make this return true when this Command no longer needs to run execute()
void MyCommandName::Interrupted()
{

}
```

Java

```
public class MyCommandName extends Command {

        /*
         * 1.        Constructor - Might have parameters for this command such as target
positions of devices. Should also set the name of the command for debugging purposes.
         *  This will be used if the status is viewed in the dashboard. And the command
should require (reserve) any devices is might use.
         */
    public MyCommandName() {
            super("MyCommandName");
        requires(elevator);
    }
```

```
    //         initialize() - This method sets up the command and is called immediately
before the command is executed for the first time and every subsequent time it is started .
    //  Any initialization code should be here.
    protected void initialize() {
    }

    /*
     *        execute() - This method is called periodically (about every 20ms) and does
the work of the command. Sometimes, if there is a position a
     *  subsystem is moving to, the command might set the target position for the
subsystem in initialize() and have an empty execute() method.
     */
    protected void execute() {
    }

    // Make this return true when this Command no longer needs to run execute()
    protected boolean isFinished() {
        return false;
    }
}
```

Simple Command Example

This example illustrates a simple command that will drive the robot using tank drive with values provided by the joysticks.

C++

```
#include "DriveWithJoysticks.h"
#include "RobotMap.h"

DriveWithJoysticks::DriveWithJoysticks() : CommandBase("DriveWithJoysticks")
{
        Requires(Robot::drivetrain); // Drivetrain is our instance of the drive system
}
```

```
// Called just before this Command runs the first time
void DriveWithJoysticks::Initialize()
{

}

    /*
     * execute() - In our execute method we call a tankDrive method we have created in our subsystem. This method takes two speeds as a parameter which we get from methods in the OI class.
     * These methods abstract the joystick objects so that if we want to change how we get the speed later we can do so without modifying our commands
     * (for example, if we want the joysticks to be less sensitive, we can multiply them by .5 in the getLeftSpeed method and leave our command the same).
     */
void DriveWithJoysticks::Execute()
{
        Robot::drivetrain->Drive(Robot::oi->GetJoystick());
}

    /*
     * isFinished - Our isFinished method always returns false meaning this command never completes on it's own. The reason we do this is that this command will be set as the default command for the subsystem. This means that whenever the subsystem is not running another command, it will run this command. If any other command is scheduled it will interrupt this command, then return to this command when the other command completes.
     */
bool DriveWithJoysticks::IsFinished()
{
        return false;
}

void DriveWithJoysticks::End()
{
        Robot::drivetrain->Drive(0, 0);
}

// Called when another command which requires one or more of the same
// subsystems is scheduled to run
```

```
void DriveWithJoysticks::Interrupted()
{
        End();
}
```

Java

```
public class DriveWithJoysticks extends Command {

    public DriveWithJoysticks() {
            requires(drivetrain);// drivetrain is an instance of our Drivetrain subsystem
    }

    protected void initialize() {
    }

    /*
     * execute() - In our execute method we call a tankDrive method we have created in our
subsystem. This method takes two speeds as a parameter which we get from methods in the OI
class.
     * These methods abstract the joystick objects so that if we want to change how we get
the speed later we can do so without modifying our commands
     * (for example, if we want the joysticks to be less sensitive, we can multiply them
by .5 in the getLeftSpeed method and leave our command the same).
     */
    protected void execute() {
            drivetrain.tankDrive(oi.getLeftSpeed(), oi.getRightSpeed());
    }

    /*
     * isFinished - Our isFinished method always returns false meaning this command never
completes on it's own. The reason we do this is that this command will be set as the
default command for the subsystem. This means that whenever the subsystem is not running
another command, it will run this command. If any other command is scheduled it will
interrupt this command, then return to this command when the other command completes.
     */
    protected boolean isFinished() {
```

```
        return false;
    }

    protected void end() {
    }

    protected void interrupted() {
    }
}
```

Creating groups of commands

Once you have created commands to operate the mechanisms in your robot, they can be grouped together to get more complex operations. These groupings of commands are called CommandGroups and are easily defined as shown in this article.

Creating a command to do a complex operation

C++

```
#include "PlaceSoda.h"

PlaceSoda::PlaceSoda()
{
        AddSequential(new SetElevatorSetpoint(TABLE_HEIGHT));
        AddSequential(new SetWristSetpoint(PICKUP));
        AddSequential(new OpenClaw());
}
```

Java

```
public class PlaceSoda extends CommandGroup {

    public  PlaceSoda() {
            addSequential(new SetElevatorSetpoint(Elevator.TABLE_HEIGHT));
            addSequential(new SetWristSetpoint(Wrist.PICKUP));
            addSequential(new OpenClaw());
    }
}
```

This is an example of a command group that places a soda can on a table. To accomplish this, (1) the robot elevator must move to the "TABLE_HEIGHT", then (2) set the wrist angle, then (3) open the claw. All of these tasks must run sequentially to make sure that the soda can isn't dropped. The addSequential() method takes a command (or a command group) as a parameter and will execute them one after another when this command is scheduled.

Running commands in parallel

C++

```
#include "PrepareToGrab.h"

PrepareToGrab::PrepareToGrab()
{
        AddParallel(new SetWristSetpoint(PICKUP));
        AddParallel(new SetElevatorSetpoint(BOTTOM));
        AddParallel(new OpenClaw());
}
```

Java

```
public class PrepareToGrab extends CommandGroup {

    public  PrepareToGrab() {
            addParallel(new SetWristSetpoint(Wrist.PICKUP));
            addParallel(new SetElevatorSetpoint(Elevator.BOTTOM));
            addParallel(new OpenClaw());
    }
}
```

To make the program more efficient, often it is desirable to run multiple commands at the same time. In this example, the robot is getting ready to grab a soda can. Since the robot isn't holding

anything, all the joints can move at the same time without worrying about dropping anything. Here all the commands are run in parallel so all the motors are running at the same time and each completes whenever the isFinished() method is called. The commands may complete out of order. The steps are: (1) move the wrist to the pickup setpoint, then (2) move the elevator to the floor pickup position, and (3) open the claw.

Mixing parallel and sequential commands

C++

```
#include "Grab.h"

Grab::Grab()
{
        AddSequential(new CloseClaw());
        AddSequential(new SetElevatorSetpoint(STOW));
        AddSequential(new SetWristSetpoint(STOW));
}
```

Java

```
public class Grab extends CommandGroup {

    public  Grab() {
            addSequential(new CloseClaw());
            addParallel(new SetElevatorSetpoint(Elevator.STOW));
            addSequential(new SetWristSetpoint(Wrist.STOW));
    }
}
```

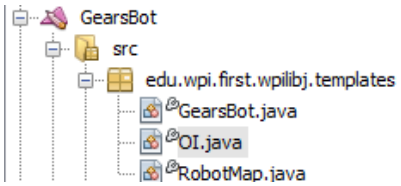
Often there are some parts of a command group that must complete before other parts run. In this example, a soda can is grabbed, then the elevator and wrist can move to their stowed

positions. In this case, the wrist and elevator have to wait until the can is grabbed, then they can operate independently. The first command (1) CloseClaw grabs the soda and nothing else runs until it is finished since it is sequential, then the (2) elevator and (3) wrist move at the same time.

Running commands on Joystick input

You can cause commands to run when joystick buttons are pressed, released, or continuously while the button is held down. This is extremely easy to do only requiring a few lines of code.

The OI Class




```
21  public class OI {
22      // Create the joystick and of the 8 buttons on it
23      Joystick leftJoy = new Joystick(1);
24      Button button1 = new JoystickButton(leftJoy, 1),
25             button2 = new JoystickButton(leftJoy, 2),
```

The command based template contains a class called OI, located in OI.java, where Operator Interface behaviors are typically defined. If you are using RobotBuilder this file can be found in the org.usfirst.frc####.NAME package

Create the Joystick object and JoystickButton objects

C++

```
OI::OI()
{
        joy = new Joystick(1);

        JoystickButton* button1 = new JoystickButton(joy, 1),
                        button2 = new JoystickButton(joy, 2),
                        button3 = new JoystickButton(joy, 3),
                        button4 = new JoystickButton(joy, 4),
                        button5 = new JoystickButton(joy, 5),
                        button6 = new JoystickButton(joy, 6),
                        button7 = new JoystickButton(joy, 7),
                        button8 = new JoystickButton(joy, 8);

}
```

Java

```
public class OI {
    // Create the joystick and the 8 buttons on it
        Joystick leftJoy = new Joystick(1);
        Button button1 = new JoystickButton(leftJoy, 1),
                        button2 = new JoystickButton(leftJoy, 2),
                        button3 = new JoystickButton(leftJoy, 3),
                        button4 = new JoystickButton(leftJoy, 4),
                        button5 = new JoystickButton(leftJoy, 5),
                        button6 = new JoystickButton(leftJoy, 6),
                        button7 = new JoystickButton(leftJoy, 7),
                        button8 = new JoystickButton(leftJoy, 8);

}
```

In this example there is a Joystick object connected as Joystick 1. Then 8 buttons are defined on that joystick to control various aspects of the robot. This is especially useful for testing although generating buttons on SmartDashboard is another alternative for testing commands.

Associate the buttons with commands

C++

```
        button1->WhenPressed(new PrepareToGrab());
        button2->WhenPressed(new Grab());
        button3->WhenPressed(new DriveToDistance(0.11));
        button4->WhenPressed(new PlaceSoda());
        button6->WhenPressed(new DriveToDistance(0.2));
        button8->WhenPressed(new Stow());

        button7->WhenPressed(new SodaDelivery());
```

Java

```
        public OI() {
                button1.whenPressed(new PrepareToGrab());
                button2.whenPressed(new Grab());
                button3.whenPressed(new DriveToDistance(0.11));
                button4.whenPressed(new PlaceSoda());
                button6.whenPressed(new DriveToDistance(0.2));
                button8.whenPressed(new Stow());

                button7.whenPressed(new SodaDelivery());
        }
```

In this example most of the joystick buttons from the previous code fragment are associated with commands. When the associated button is pressed the command is run. This is an excellent way to create a teleop program that has buttons to do particular actions.

Other options

In addition to the "whenPressed()" condition showcased above, there are a few other conditions you can use to link buttons to commands:

- Commands can run when a button is released by using whenReleased() instead of whenPressed().
- Commands can run continuously while the button is depressed by calling whileHeld().
- Commands can be toggled when a button is pressed using toggleWhenPressed().
- A command can be canceled when a button is pressed using cancelWhenPressed().

Additionally commands can be triggered by arbitrary conditions of your choosing by using the Trigger class instead of Button. Triggers (and Buttons) are usually polled every 20ms or whenever the scheduler is called.

Running commands during the autonomous period

Once commands are defined they can run in either the teleop or autonomous part of the program. In fact, the power of the command based programming approach is that you can reuse the same commands in either place. If the robot has a command that can shoot Frisbees during autonomous with camera aiming and accurate shooting, there is no reason not to use it to help the drivers during the teleop period of the game.

Creating a command to use for Autonomous

C++

```
        button1->WhenPressed(new PrepareToGrab());
        button2->WhenPressed(new Grab());
        button3->WhenPressed(new DriveToDistance(0.11));
        button4->WhenPressed(new PlaceSoda());
        button6->WhenPressed(new DriveToDistance(0.2));
        button8->WhenPressed(new Stow());

        button7->WhenPressed(new SodaDelivery());
```

Java

```
        public OI() {
                button1.whenPressed(new PrepareToGrab());
                button2.whenPressed(new Grab());
                button3.whenPressed(new DriveToDistance(0.11));
                button4.whenPressed(new PlaceSoda());
                button6.whenPressed(new DriveToDistance(0.2));
                button8.whenPressed(new Stow());

                button7.whenPressed(new SodaDelivery());
```

```
        }
```

Our robot must do the following tasks during the autonomous period: pick up a soda can off the floor then drive a set distance from a table and deliver the can there. The process consists of:

1. Prepare to grab (move elevator, wrist, and gripper into position)
2. Grab the soda can
3. Drive to a distance from the table indicated by an ultrasonic rangefinder
4. Place the soda
5. Back off to a distance from the rangefinder
6. Re-stow the gripper

To do these tasks there are 6 command groups that are executed sequentially as shown in this example.

Setting that command to run as the autonomous behavior

C++

```
Command* autonomousCommand;

class Robot: public IterativeRobot {

    /**
     * This function is run when the robot is first started up and should be
     * used for any initialization code.
     */
        void RobotInit()
        {
        // instantiate the command used for the autonomous period
                autonomousCommand = new SodaDelivery();
                oi = new OI();

        }
```

```
        void AutonomousInit()
        {
        // schedule the autonomous command (example)
                if(autonomousCommand != NULL) autonomousCommand->Start();
        }
        /*
         * This function is called periodically during autonomous
         */
        void AutonomousPeriodic()
        {
                Scheduler::GetInstance()->Run();
        }
```

Java

```
public class Robot extends IterativeRobot {



    Command autonomousCommand;

    /**
     * This function is run when the robot is first started up and should be
     * used for any initialization code.
     */
    public void robotInit() {
                oi = new OI();
        // instantiate the command used for the autonomous period
        autonomousCommand = new SodaDelivery();
    }

    public void autonomousInit() {
        // schedule the autonomous command (example)
        if (autonomousCommand != null) autonomousCommand.start();
    }

    /**
```

```
     * This function is called periodically during autonomous
     */
    public void autonomousPeriodic() {
        Scheduler.getInstance().run();
    }
```

To get the SodaDelivery command to run as the Autonomous program,

1. Instantiate it in the robotInit() method
2. Start it during the autonomousInit() method
3. Be sure the scheduler is called repeatedly during the autonomousPeriodic() method.

RobotInit() is called only once when the robot starts so it is a good time to create the command instance. AutonomousPeriodic() is called once at the start of the autonomous period so we schedule the command there. AutonomousPeriodic() is called every 20ms so that is a good time to run the scheduler which makes a pass through all the currently scheduled commands.

Converting a Simple Autonomous program to a Command based autonomous program

The initial autonomous code with loops

C++

```
// Aim shooter
SetTargetAngle(); // Initialization: prepares for the action to be performed
while (!AtRightAngle()) { // Condition: keeps the loop going while it is satisfied
        CorrectAngle(); // Execution: repeatedly updates the code to try to make the
condition false
        delay(); // Delay to prevent maxing CPU
}
HoldAngle(); // End: performs any cleanup and final task before moving on to the next
action

// Spin up to Speed
SetTargetSpeed(); // Initialization: prepares for the action to be performed
while (!FastEnough()) { // Condition: keeps the loop going while it is satisfied
        SpeedUp(); // Execution: repeatedly updates the code to try to make the condition
false
        delay(); // Delay to prevent maxing CPU
}
HoldSpeed();

// Shoot Frisbee
Shoot(); // End: performs any cleanup and final task before moving on to the next action
}
```

Java

```
// Aim shooter
```

```
SetTargetAngle(); // Initialization: prepares for the action to be performed
while (!AtRightAngle()) { // Condition: keeps the loop going while it is satisfied
        CorrectAngle(); // Execution: repeatedly updates the code to try to make the
condition false
        delay(); // Delay to prevent maxing CPU
}
HoldAngle(); // End: performs any cleanup and final task before moving on to the next
action

// Spin up to Speed
SetTargetSpeed(); // Initialization: prepares for the action to be performed
while (!FastEnough()) { // Condition: keeps the loop going while it is satisfied
        SpeedUp(); // Execution: repeatedly updates the code to try to make the condition
false
        delay(); // Delay to prevent maxing CPU
}
HoldSpeed();

// Shoot Frisbee
Shoot(); // End: performs any cleanup and final task before moving on to the next action
}
```

The code above aims a shooter, then it spins up a wheel and, finally, once the wheel is running at the desired speed, it shoots the frisbee. The code consists of three distinct actions: aim, spin up to speed and shoot the Frisbee. The first two actions follow a command pattern that consists of four parts:

1. Initialization: prepares for the action to be per- formed.

2. Condition: keeps the loop going while it is satisfied.

3. Execution: repeatedly updates the code to try to make the condition false.

4. End: performs any cleanup and final task before moving on to the next action.

The last action only has an explicit initialize, though depending on how you read it, it can implicitly end under a number of conditions. The most obvious one two in this case are when it's done shooting or when autonomous has ended.

Rewriting it as Commands

C++

```
#include "AutonomousCommand.h"

AutonomousCommand::AutonomousCommand()
{
     AddSequential(new Aim());
     AddSequential(new SpinUpShooter());
     AddSequential(new Shoot());
}
```

Java

```
public class AutonomousCommand extends CommandGroup {

    public  AutonomousCommand() {
            addSequential(new Aim());
            addSequential(new SpinUpShooter());
            addSequential(new Shoot());
    }
}
```

The same code can be rewritten as a CommandGroup that groups the three actions, where each action is written as it's own command. First, the command group will be written, then the commands will be written to accomplish the three actions. This code is pretty straightforward. It does the three actions sequentially, that is one after the other. Line 3 aims the robot, then line 4

spins the shooterup and, finally, line 5 actually shoots the frisbee. The addSequential() method sets it so that these commands run one after the other.

The Aim command

C++

```
#include "Aim.h"

Aim::Aim()
{
     Requires(Robot::turret);
}

// Called just before this Command runs the first time
void Aim::Initialize()
{
     SetTargetAngle();
}

// Called repeatedly when this Command is scheduled to run
void Aim:Execute()
{
 \    CorrectAngle();
}

// Make this return true when this Command no longer needs to run execute()
bool Aim:IsFinished()
{
     return AtRightAngle();
}

// Called once after isFinished returns true
void Aim::End()
{
     HoldAngle();
}
// Called when another command which requires one or more of the same
// subsystems is scheduled to run
```

```
void Aim:Interrupted()
{
     End();
}
```

Java

```
public class Aim extends Command {

    public Aim() {
            requires(Robot.turret);
    }

    // Called just before this Command runs the first time
    protected void initialize() {
            SetTargetAngle();
    }

    // Called repeatedly when this Command is scheduled to run
    protected void execute() {
            CorrectAngle();
 \    }

    // Make this return true when this Command no longer needs to run execute()
    protected boolean isFinished() {
        return AtRightAngle();
 \    }

    // Called once after isFinished returns true
    protected void end() {
            HoldAngle();
    }

    // Called when another command which requires one or more of the same
    // subsystems is scheduled to run
    protected void interrupted() {
            end();
```

```
    }
}
```

As you can see, the command reflects the four parts of the action we discussed earlier. It also has the interrupted() method which will be discussed below. The other significant difference is that the condition in the isFinished() is the opposite of what you would put as the condition of the while loop, it returns true when you want to stop running the execute method as opposed to false. Initializing, executing and ending are exactly the same, they just go within their respective method to indicate what they do.

SpinUpShooter command

C++

```
#include "SpinUpShooter.h"

SpinUpShooter::SpinUpShooter()
{
     Requires(Robot::shooter)
}

// Called just before this Command runs the first time
void SpinUpShooter::Initialize()
{
 \    SetTargetSpeed();
}

// Called repeatedly when this Command is scheduled to run
void SpinUpShooter::Execute()
{
     SpeedUp();
}

// Make this return true when this Command no longer needs to run execute()
bool SpinUpShooter::IsFinished()
{
```

```
 \    return FastEnough();
}

// Called once after isFinished returns true
void SpinUpShooter::End()
{
     HoldSpeed();
}

// Called when another command which requires one or more of the same
// subsystems is scheduled to run
void SpinUpShooter::Interrupted()
{
     End();
}
```

Java

```
public class SpinUpShooter extends Command {

    public SpinUpShooter() {
        requires(Robot.shooter);
 \   }

    // Called just before this Command runs the first time
    protected void initialize() {
            SetTargetSpeed();
    }

    // Called repeatedly when this Command is scheduled to run
    protected void execute() {
            SpeedUp();
 \   }

    // Make this return true when this Command no longer needs to run execute()
    protected boolean isFinished() {
        return FastEnough();
```

```
 \   }

    // Called once after isFinished returns true
    protected void end() {
            HoldSpeed();
    }

    // Called when another command which requires one or more of the same
    // subsystems is scheduled to run
    protected void interrupted() {
            end();
    }
}
```

The spin up shooter command is very similar to the Aim command, it's the same basic idea.

Shoot command

C++

```
#include "Shoot.h"

Shoot::Shoot()
{
     Requires(Robot.shooter);
}

// Called just before this Command runs the first time
void Shoot::Initialize()
{
 \    Shoot();
}

// Called repeatedly when this Command is scheduled to run
void Shoot::Execute()
```

```
{
}

// Make this return true when this Command no longer needs to run execute()
bool Shoot::IsFinished()
{
     return true;
}

// Called once after isFinished returns true
void Shoot::End()
{
}

// Called when another command which requires one or more of the same
// subsystems is scheduled to run
void Shoot::Interrupted()
{
     End();
}
```

Java

```
public class Shoot extends Command {

    public Shoot() {
        requires(shooter);
    }

    // Called just before this Command runs the first time
    protected void initialize() {
            Shoot();
    }

    // Called repeatedly when this Command is scheduled to run
    protected void execute() {
    }
```

```
    // Make this return true when this Command no longer needs to run execute()
    protected boolean isFinished() {
        return true;
    }

    // Called once after isFinished returns true
    protected void end() {
    }

    // Called when another command which requires one or more of the same
    // subsystems is scheduled to run
 \   protected void interrupted() {
            end();
    }
}
```

The shoot command is the same basic transformation yet again, however it is set to end immediately. In CommandBased programming, it is better to have it's isFinished method return true when the act of shooting is finished, but this is a more direct translation of the original code.

Benefits of the command based approach

Why bother re-writing the code as CommandBased? Writing the code in the CommandBased style offers a number of benefits:

- Re-Usability You can reuse the same command in teleop and multiple autonomous modes. They all reference the same code, so if you need to tweak it to tune it or fix it, you can do it in one place without having to make the same edits in multiple places.
- Testability You can test each part using tools such as the SmartDashboard to test parts of the autonomous. Once you put them together, you'll have more confidence that each piece works as desired.
- Parallelization If you wanted this code to aim and spin up the shooter at the same time, it's trivial with CommandBased programming. Just use AddParallel() instead of AddSequential() when adding the Aim command and now aiming and spinning up will happen simultaneously.

- **Interruptibility** Commands are interruptible, this provides the ability to exit a command early, a task that is much harder in the equivalent while loop based code.

Default Commands

In some cases you may have a subsystem which you want to always be running a command no matter what. So what do you do when the command you are currently running ends? That's where default commands come in.

What is the default command?

Each subsystem may, but is not required to, have a default command which is scheduled whenever the subsystem is idle (the command currently requiring the system completes). The most common example of a default command is a command for the drivetrain that implements the normal joystick control. This command may be interrupted by other commands for specific maneuvers ("precision mode", automatic alignment/targeting, etc.) but after any command requiring the drivetrain completes the joystick command would be scheduled again.

Setting the default command

C++

```
#include "ExampleSubsystem.h"

ExampleSubsystem::ExampleSubsystem()
{
     // Put methods for controlling this subsystem
     // here. Call these from Commands.
}

ExampleSubsystem::InitDefaultCommand()
{
     // Set the default command for a subsystem here.
     SetDefaultCommand(new MyDefaultCommand());
}
```

Java

```
public class ExampleSubsystem extends Subsystem {

    // Put methods for controlling this subsystem
    // here. Call these from Commands.

    public void initDefaultCommand() {
        // Set the default command for a subsystem here.
        setDefaultCommand(new MyDefaultCommand());
    }
}
```

All subsystems should contain a method called initDefaultCommand() which is where you will set the default command if desired. If you do not wish to have a default command, simply leave this method blank. If you do wish to set a default command, call setDefaultCommand from within this method, passing in the command to be set as the default.

Synchronizing two commands

Commands can be nested inside of command groups to create more complex commands. The simpler commands can be added to the command groups to either run sequentially (each command finishing before the next starts) or in parallel (the command is scheduled, and the next command is immediately scheduled also). Occasionally there are times where you want to make sure that two parallel command complete before moving onto the next command. This article describes how to do that.

Creating a command group with sequential and parallel commands

C++

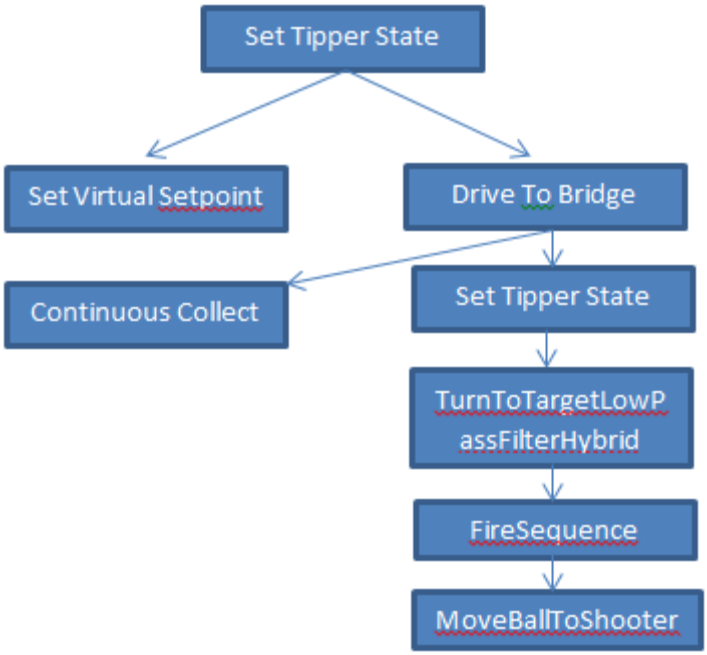
```
#include "CoopBridgeAutonomous.h"

CoopBridgeAutonomous::CoopBridgeAutonomous()
{
     // SmartDashboard->PutDouble("Camera Time", 5.0);
     AddSequential(new SetTipperState(READY_STATE);
     AddParallel(new SetVirtualSetpoint(HYBRID_LOCATION);
     AddSequential(new DriveToBridge());
     AddParallel(new ContinuousCollect());
     AddSequential(new SetTipperState(DOWN_STATE));

     // addParallel(new WaitThenShoot());

     AddSequential(new TurnToTargetLowPassFilterHybrid(4.0));
     AddSequential(new FireSequence());
     AddSequential(new MoveBallToShooter(true));
}
```

Java

```
public class CoopBridgeAutonomous extends CommandGroup {

    public  CoopBridgeAutonomous() {
            //SmartDashboard.putDouble("Camera Time", 5.0);
            addSequential(new SetTipperState(BridgeTipper.READY_STATE)); // 1
            addParallel(new SetVirtualSetpoint(SetVirtualSetpoint.HYBRID_LOCATION)); // 2
            addSequential(new DriveToBridge()); // 3
            addParallel(new ContinuousCollect());
            addSequential(new SetTipperState(BridgeTipper.DOWN_STATE));

            // addParallel(new WaitThenShoot());

            addSequential(new TurnToTargetLowPassFilterHybrid(4.0));
            addSequential(new FireSequence());
            addSequential(new MoveBallToShooter(true));
    }
}
```

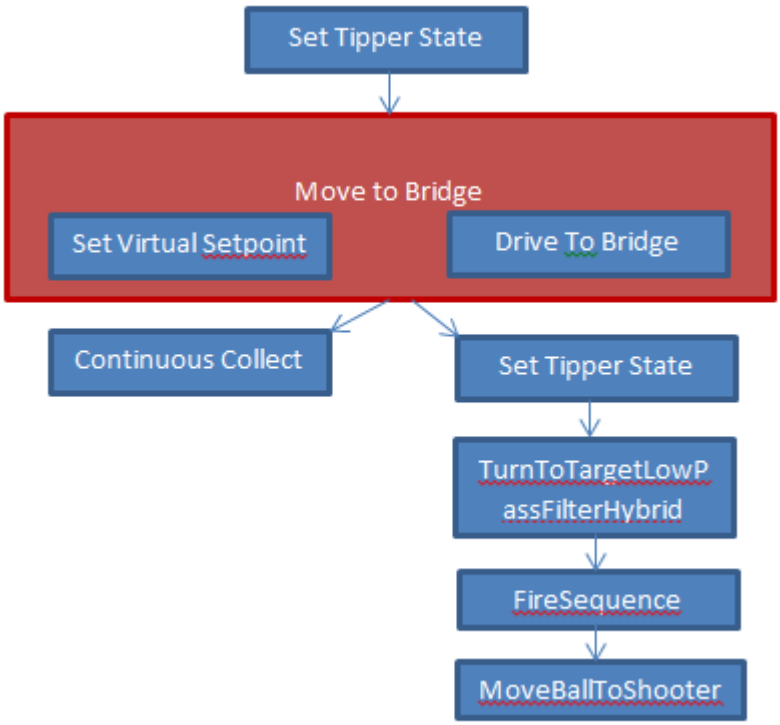
In this example some commands are added in parallel and others are added sequentially to the CommandGroup CoopBridgeAutonomous (1). The first command "SetTipperState" is added and completes before the SetVirtualSetpoint command starts (2). Before SetVirtualSetpoint command completes, the DriveToBridge command is immediately scheduled because of the SetVirtualSetpoint is added in parallel (3). This example might give you an idea of how commands are scheduled.

Example Flowchart



Here is the code shown above represented as a flowchart. Note that there is no dependency coming from the commands scheduled using "Add Parallel" either or both of these commands could still be running when the MoveBallToShooter command is reached. Any command in the main sequence (the sequence on the right here) that requires a subsystem in use by a parallel command will cause the parallel command to be canceled. For example, if the FireSequence required a subsystem in use by SetVirtualSetpoint, the SetVirtualSetpoint command will be canceled when FireSequence is scheduled.

Getting a command to wait for another command to complete



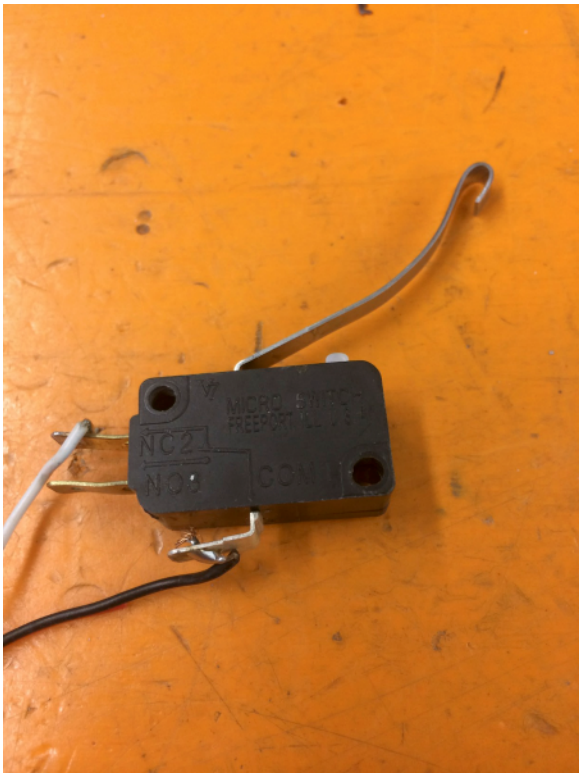
If there are two commands that need to complete before the following commands are scheduled, they can be put into a command group by themselves, adding both in parallel. Then that command group can be scheduled sequentially from an enclosing command group. When a command group is scheduled sequentially, the commands inside it will all finish before the next outer command is scheduled. In this way you can be sure that an action consisting of multiple parallel commands has completed before going on to the next command.

In this example you can see that the addition of a command group "Move to Bridge" containing the Set Virtual Setpoint and Drive to Bridge commands forces both to complete before the next commands are scheduled.

Using limit switches to control behavior

Limit switches are often used to control mechanisms on robots. While limit switches are simple to use, they only can sense a single position of a moving part. This makes them ideal for ensuring that movement doesn't exceed some limit but not so good at controlling the speed of the movement as it approaches the limit. For example, a rotational shoulder joint on a robot arm would best be controlled using a potentiometer or an absolute encoder, the limit switch could make sure that if the potentiometer ever failed, the limit switch would stop the robot from going to far and causing damage.

What values are provided by the limit switch



Limit switches can have "normally opened" or "normally closed" outputs. The usual way of wiring the switch is between a digital input signal connection and ground. The digital input has pull-up resistors that will make the input be high (1 value) when the switch is open, but when the switch closes the value goes to 0 since the input is now connected to ground. The switch shown here has both normally open and normally closed outputs.

Polling waiting for a switch to close

C++

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

RobotTemplate::RobotTemplate()
{
     DigitalInput* limitSwitch;
}

void RobotTemplate::RobotInit()
{
     limitSwitch = new DigitalInput(1);
}

void RobotTemplate::operatorControl()
{
     while(limitSwitch->Get())
     {
          Wait(10);
     }
}
```

Java

```
import edu.wpi.first.wpilibj.DigitalInput;
import edu.wpi.first.wpilibj.SampleRobot;
import edu.wpi.first.wpilibj.Timer;

public class RobotTemplate extends SampleRobot {

        DigitalInput limitSwitch;

        public void robotInit() {
```

```
                limitSwitch = new DigitalInput(1);
        }

        public void operatorControl() {
                // more code here
                while (limitSwitch.get()) {
                        Timer.delay(10);
                }
        }
}
```

You can write a very simple piece of code that just reads the limit switch over and over again waiting until it detects that its value transitions from 1 (opened) to 0 (closed). While this works, it's usually impractical for the program to be able to just wait for the switch to operate and not be doing anything else, like responding to joystick input. This example shows the fundamental use of the switch, but while the program is waiting, nothing else is happening.

Command-based program to operate until limit switch closed

C++

```
#include "ArmUp.h"

ArmUp::ArmUp()
{

}

void ArmUp::Initialize()
{
     arm.ArmUp();
}

void ArmUp::Execute()
{
}
```

```
void ArmUp::IsFinished()
{
     return arm.isSwitchSet();
}

void ArmUp::End()
{
     arm.ArmStop();
}

void ArmUp::Interrupted()
{
     End();
}
```

Java

```
package edu.wpi.first.wpilibj.templates.commands;

public class ArmUp extends CommandBase {
    public ArmUp() {
    }

    protected void initialize() {
        arm.armUp();
    }

    protected void execute() {
    }

    protected boolean isFinished() {
        return arm.isSwitchSet();
    }

    protected void end() {
        arm.armStop();
```

```
    }

    protected void interrupted() {
        end();
    }
}
```

Commands call their execute() and isFinished() methods about 50 times per second, or at a rate of every 20ms. A command that will operate a motor until the limit switch is closed can read the digital input value in the isFinished() method and return true when the switch changes to the correct state. Then the command can stop the motor.

Remember, the mechanism (an Arm in this case) has some inertia and won't stop immediately so it's important to make sure things don't break while the arm is slowing.

Using a counter to detect the closing of the switch

C++

```
#include "WPILIB.h"
#include "Arm.h"

DigitalInput* limitSwitch;
SpeedController* armMotor;
Counter* counter;

Arm::Arm()
{
     limitSwitch = new DigitalInput(1);
     armMotor = new Victor(1);
     counter = new Counter(limitSwitch);
}

bool Arm::IsSwitchSet()
{
     return counter->Get() >0;
```

```
}

void Arm::InitializeCounter()
{
     counter->Reset();
}

void Arm::ArmUp()
{
     armMotor->Set(.5);
}

void Arm::ArmDown()
{
     armMotor->Set(-0.5);
}

void Arm::ArmStop()
{
     armMotor->Set(0);
}

void InitDefaultCommand()
{
}
```

Java

```
package edu.wpi.first.wpilibj.templates.subsystems;
import edu.wpi.first.wpilibj.Counter;
import edu.wpi.first.wpilibj.DigitalInput;
import edu.wpi.first.wpilibj.SpeedController;
import edu.wpi.first.wpilibj.Victor;
import edu.wpi.first.wpilibj.command.Subsystem;
public class Arm extends Subsystem {

    DigitalInput limitSwitch = new DigitalInput(1);
```

```
    SpeedController armMotor = new Victor(1);
    Counter counter = new Counter(limitSwitch);

    public boolean isSwitchSet() {
        return counter.get() > 0;
    }

    public void initializeCounter() {
        counter.reset();
    }

    public void armUp() {
        armMotor.set(0.5);
    }

    public void armDown() {
        armMotor.set(-0.5);
    }

    public void armStop() {
        armMotor.set(0.0);
    }
    protected void initDefaultCommand() {
    }
}
```

It's possible that a limit switch might close then open again as a mechanism moves past the switch. If the closure is fast enough the program might not notice that the switch closed. An alternative method of catching the switch closing is use a Counter object. Since counters are implemented in hardware, it will be able to capture the closing of the fastest switches and increment it's count. Then the program can simply notice that the count has increased and take whatever steps are needed to do the operation.

Above is a subsystem that uses a counter to watch the limit switch and wait for the value to change. When it does, the counter will increment and that can be watched in a command.

Create a command that uses the counter to detect switch closing

C++

```
#include "ArmUp.h"

ArmUp::ArmUp()
{
}

void ArmUp::Initialize()
{
     arm.InitializeCounter();
     arm.ArmUp();
}

void ArmUp::Execute()
{
}

bool ArmUp::IsFinished()
{
     return arm->IsSwitchSet();
}

void ArmUp::End()
{
     arm->ArmStop();
}

void ArmUp::Interrupted()
{
     End();
}
```

Java

```
package edu.wpi.first.wpilibj.templates.commands;

public class ArmUp extends CommandBase {

    public ArmUp() {
    }

    protected void initialize() {
        arm.initializeCounter();
        arm.armUp();
    }

    protected void execute() {
    }

    protected boolean isFinished() {
        return arm.isSwitchSet();
    }

    protected void end() {
        arm.armStop();
    }

    protected void interrupted() {
        end();
    }
}
```

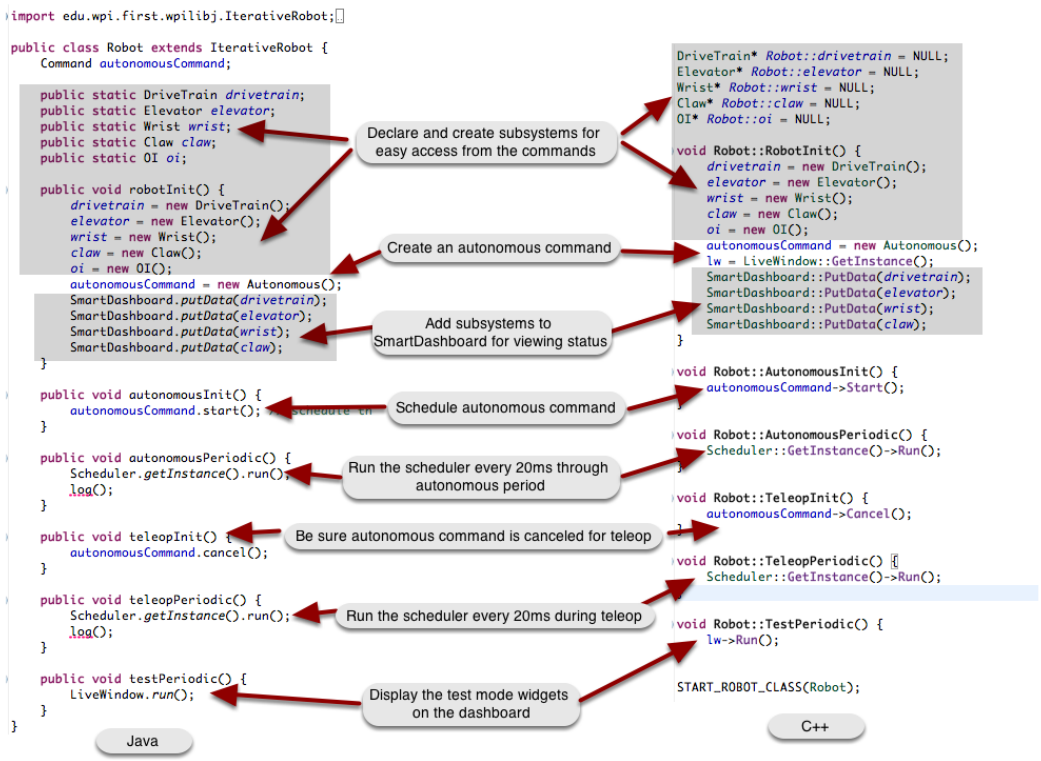
This command initializes the counter in the above subsystem then starts the motor moving. It then tests the counter value in the isFinished() method waiting for it to count the limit switch changing. When it does, the arm is stopped. By using a hardware counter, a switch that might close then open very quickly can still be caught by the program.

Scheduling commands

Commands are scheduled to run based on a number of factors such as triggers, default commands when no other running commands require a subsystem, a prior command in a group finishes, button presses, autonomous period starting, etc. Although many commands may be running virtually at the same time, there is only a single thread (the main robot thread). This is to reduce the complexity of synchronization between threads. There are threads that run in the system for systems like PID loops, communications, etc. but those are all self contained with very little interaction requiring complex synchronization. This makes the system much more robust and predictable.

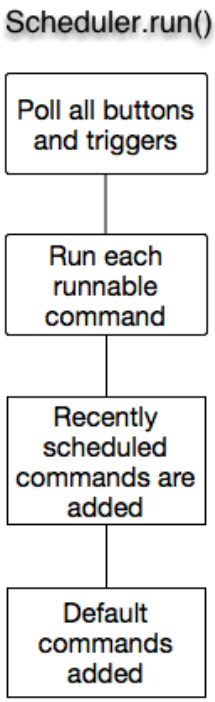
This is accomplished by a class called Scheduler. It has a run() method that is called periodically (typically every 20ms in response to a driver station update) that tries to make progress on every command that is currently running. This is done by calling the execute() method on the command followed by the isFinished() method. If isFinished() returns true, the command is marked to be removed from execution on the next pass through the scheduler. So if there are a number of commands all scheduled to run at the same time, then every time the Scheduler.run() method is called, each of the active commands execute() and isFinished() methods are called. This has the same effect as using multiple threads.

Anatomy of a command-based robot program



This shows a typical command-based Robot program and all the code needed to ensure that commands are scheduled correctly. The Scheduler.run method causes one pass through the scheduler which will let each currently active command run through its execute() and isFinished() methods. Ignore the log() methods in the Java example.

The Scheduler.run method: the command life cycle



The work in command-based programs occurs whenever the Scheduler.Run (C++) or Scheduler.run (Java) method is called. This is typically called on each driver station update which occurs every 20 ms or 50 times per second. The pseudo code illustrates what happens on each call to the run method.

1. Buttons and triggers are polled to see if the associated commands should be scheduled. If the trigger is true, the command is added to a list of commands that should be scheduled.
2. Loop through the list of all the commands that are currently runnable and call their execute and isFinished methods. Commands where the isFinished method returns true are removed from the list of currently running commands.
3. Loop through all the commands that have been scheduled to run in the previous steps. Those commands are added to the list of running commands.
4. Default commands are added for each subsystem that currently has no commands running that require that subsystem.

Optimizing command groups

C++

```
Pickup::Pickup() : CommandGroup("Pickup") {
    AddSequential(new CloseClaw());
    AddParallel(new SetWristSetpoint(-45));
    AddSequential(new SetElevatorSetpoint(0.25));
}
```

Java

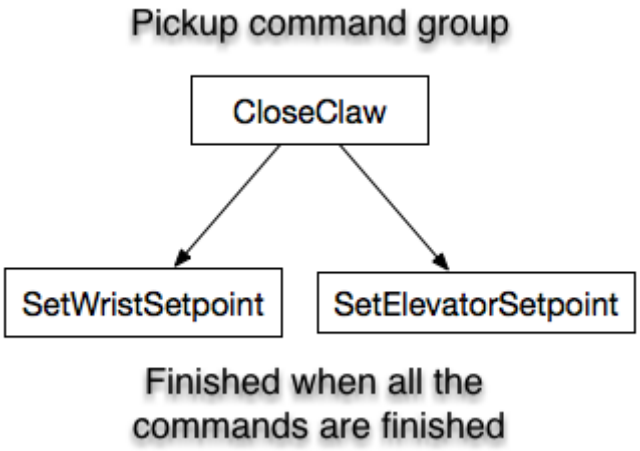
```
public class Pickup extends CommandGroup {
    public  Pickup() {
            addSequential(new CloseClaw());
            addParallel(new SetWristSetpoint(-45));
            addSequential(new SetElevatorSetpoint(0.25));
    }
}
```

Once you have working commands that operate the mechanisms on your robot you can combine those commands into groups to make more complex actions. Commands can be added to command groups to execute sequentially or in parallel. Sequential commands wait until they are finished (isFinished method returns true) before running the next command in the group. Parallel commands start running, then immediately schedule the next command in the group.

It is important to notice that the commands are added to the group in the constructor. The command group is simply a list of command instances that run when scheduled and any parameters that are passed to the commands are evaluated during the constructor for the group.

Imagine that in a robot design, there is a claw, attached to a wrist joint and all of those on an elevator. When picking up something, the claw needs to close first before either the elevator or wrist can move otherwise the object may slip out of the claw. In the example shown above the CloseClaw command will be scheduled first. After it is finished (the claw is closed), the wrist will move to it's setpoint and in parallel, the elevator will move. This gets both the elevator and wrist moving simultaneously optimizing the time required to complete the task.

When do command groups finish?



A command group finishes when all the commands started in that group finish. This is true regardless of the type of commands that are added to the group. For example, if a number of commands are added in parallel and sequentially, the group is finished when all the commands added to the group are finished. As each command is added to a command group, it is put on a list. As those child commands finish, they are taken off the list. The command group is finished when the list of child commands is empty.

In the Pickup command shown in the example above, the command is finished when CloseClaw, SetWristSetpoint, and SetElevatorSetpoint all finish. It doesn't matter that some of the commands are sequential and some parallel.

How to schedule a command from within a running command

Commands can be scheduled by calling the schedule() method (Java) or Schedule() method (C++) on a command instance. This will cause the command to be added to the currently running set of commands in the scheduler. This is often useful when a program needs to conditionally schedule one command or another. The newly scheduled command will be added to a list of new commands on this pass through the run method of the scheduler and actually will run the first time on the next pass through the run method. Newly created commands are never executed in the same call to the scheduler run method, always queued for the next call which usually occurs 20ms later.

Removing all running commands from the scheduler

C++

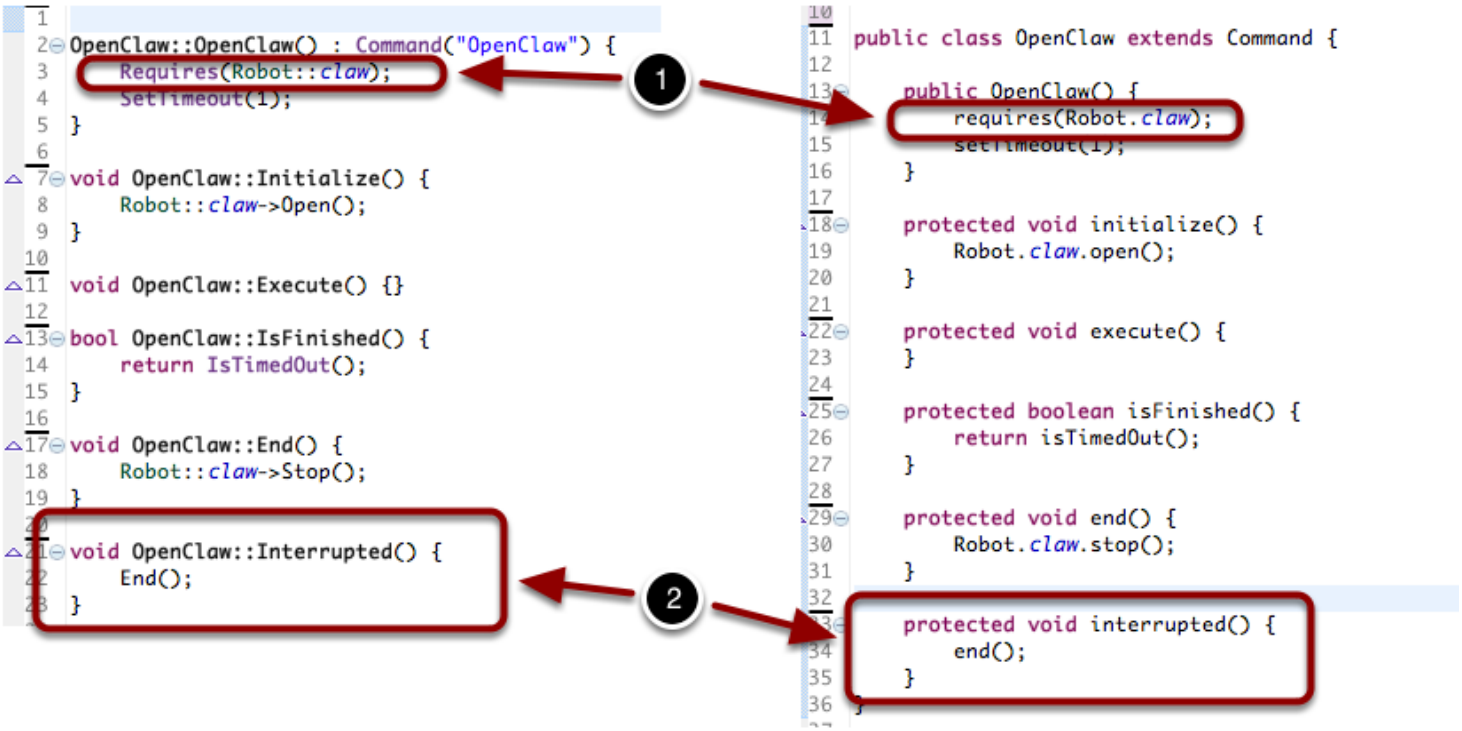
```
Scheduler::RemoveAll();
```

Java

```
Scheduler.getInstance().removeAll();
```

It is occasionally useful to make sure that there are no running commands in the scheduler. To remove all running commands use the Scheduler.removeAll() method (Java) or Scheduler::RemoveAll() method (C++). This will cause all currently running to have their interrupted() method (Java) or Interrupted() method (C++) called. Commands that have not yet started will have their end() method (Java) or End() method (C++) called.

What does the "requires" method do?



If you have multiple commands that use the same subsystem it makes sense that they don't run at the same time. For example, if there is a Claw subsystem with OpenClaw and CloseClaw commands, they can't both run at the same time. Each command that uses the Claw subsystem declares that by 1 calling the requires() method (Java) or Requires() method (C++). When one of the commands is running, say from a joystick button press, and you try to run another command that also requires the Claw, the second one preempts the first one. Suppose that OpenClaw was running, and you press the button to run the CloseClaw command. The OpenClaw command is

interrupted - 2 it's interrupted method is called on the next run cycle and the CloseClaw command is scheduled. If you think about it, this is almost always the desired behavior. If you pressed a button to start opening the claw and you change your mind and want to close it, it makes sense for the OpenClaw command to be stopped and the CloseClaw to be started.

A command may require many subsystems, for example a complex autonomous sequence might use a number of subsystems to complete its task.

Command groups automatically require all the subsystems for each of the commands in the group. There is no need to call the requires method for a group.

How are the requirements of a group evaluated?

The subsystems that a command group requires is the union of the set of subsystems that are required for all of the child commands. If a 4 commands are added to a group, then the group will require all of the subsystems required by each of the 4 commands in the group. For example, if are three commands scheduled in a group - the first requires subsystem A, the second requires subsystem B, and the third requires subsystems C and D. The group will require subsystems A, B, C, and D. If another command is started, say from a joystick button, that requires either A, B, C, or D it will interrupt the entire group including any parallel or sequential commands that might be running from that group.

Creating more robust commands with timeouts

Often you might write a command that has an "isFinished" condition that is based on a sensor reaching a particular value or a switch closing. For example, a command might rotate a robot 45 degrees by turning until the heading from a gyro reaches 45. Suppose the gyro fails or a wire pulls loose. Now the gyro will always report the same heading, probably zero degrees, and the robot will keep turning for ever. One way of making your commands more robust is to have them "time out" or quit after some time has elapsed. In the case of the turn to 45 degrees, you might know that this will always complete within 2 seconds. You could set a timeout for 3 or 4 seconds to the command will finish even if the robot never reaches the desired heading. This will allow the program to continue running rather than be stuck waiting for the roboRIO to read a heading that will never happen.

A command with no timeout

C++

```
TurnTo45Degrees::TurnTo45Degrees()  : Command("TurnTo45Degrees") {
}

void TurnTo45Degrees::Initialize() {
}

void TurnTo45Degrees::Execute() {
    Robot::drivetrain->turn(0.5);  // start turning
}

bool TurnTo45Degrees::IsFinished() {
    return Robot::driveTrain->getHeading() >= 45.0; // wait for 45 degrees
}

void TurnTo45Degrees::End() {
    Robot::driveTrain->turn(0);  // stop turning
}

void TurnTo45Degrees::Interrupted() {
    End();
}
```

Java

```
public class TurnTo45Degrees extends Command {
    public TurnTo45Degrees() {
    }

    protected void initialize() {
    }

    protected void execute() {
            Robot.chassis.turn(0.5); // start turning
    }

    protected boolean isFinished() {
        return Robot.chassis.getGyroHeading() >= 45.0; // wait for 45 degrees
    }

    protected void end() {
            Robot.chassis.turn(0);  // stop turning
    }

    protected void interrupted() {
            end();
    }
}
```

The code above shows a command that will never time out, and will keep running until the desired heading is reached. Just a note, this is an over-simplified example to illustrate the use of timeouts and not as a way to necessarily get your robot to turn to headings - a PID controller would make the robot performance better.

Adding a timeout to the command

C++

```
TurnTo45Degrees::TurnTo45Degrees()  : Command("TurnTo45Degrees") {
}

void TurnTo45Degrees::Initialize() {
    SetTimeout(4);  // set 4 second timeout
}
```

```
void TurnTo45Degrees::Execute() {
    Robot::drivetrain->turn(0.5);  // start turning
}

bool TurnTo45Degrees::IsFinished() {
    // turn until 45 degrees OR timed out
    return Robot::driveTrain->getHeading() >= 45.0 || IsTimedOut();
}

void TurnTo45Degrees::End() {
    Robot::driveTrain->turn(0);  // stop turning
}

void TurnTo45Degrees::Interrupted() {
    End();
}
```

Java

```
public class TurnTo45Degrees extends Command {
    public TurnTo45Degrees() {
    }

    protected void initialize() {
        setTimeout(4);  // set 4 second timeout
    }

    protected void execute() {
            Robot.chassis.turn(0.5); // start turning
    }

    protected boolean isFinished() {
        // wait for 45 degrees OR timed out
        return Robot.chassis.getGyroHeading() >= 45.0 || isTimedOut();
    }

    protected void end() {
            Robot.chassis.turn(0);  // stop turning
    }
```

```
    protected void interrupted() {
            end();
    }
}
```

This example shows the same command but with a 4 second timeout. This guarantees that the robot will stop turning and the command will end in 4 seconds regardless of the gyro heading. Hopefully the gyro or its wiring will never fail, but at least you know that if it does, the command will eventually end.

Adding a command to a group with a timeout

The timeout feature can also be used when adding a command to a command group even if the command itself has no timeout. To do this, add a second parameter to the AddSequential() or AddParallel() methods (addSequential() and addParallel() in Java) that is the timeout in seconds. This will cause the sequential command to finish either when the command itself finishes from the IsFinished (C++) or isFinished (Java) method returns true OR the command runtime reaches the timeout period. When using this feature, there is no indication in the command that the timeout is happening. If the command already has a timeout, the results are undefined.