

FRC JAVA PROGRAMMING

Table of Contents

Setting up the Development Environment6

 Installing C++ and Java Development Tools for FRC7

 Installing the FRC Update Suite (All Languages)..... 16

Creating and Running Robot Programs 37

 Visual Studio Code Basics and the WPILib Extension..... 38

 WPILib Commands in VSCode 41

 Creating a robot program..... 43

 Creating your Benchtop Test Program 51

 Building and deploying to a roboRIO 57

 Viewing Console Output 59

 Debugging a robot program..... 63

 Importing an Eclipse project into VS Code 68

FRC Java References 71

 FRC Java WPILib API Documentation 72

 C++\Java Plugin Changelog 73

FRC Java Basics 80

 Java conventions for objects, methods and variables..... 81

 Multithreading in Java 83

Basic WPILib Programming features 85

 What is WPILib..... 86

 Choosing a Base Class..... 90

FRC Java Programming

Using actuators (motors, servos, and relays)	95
Actuator Overview	96
Driving motors with PWM speed controller objects	97
WPILib Drive classes: Drivetrain types	100
WPILib Drive classes: Conventions and Defaults	103
Driving a robot using Differential Drive	105
Driving a robot using Mecanum drive	110
Repeatable Low Power Movement - Controlling Servos with WPILib	115
Using the motor safety feature	117
On/Off control of motors and other mechanisms - Relays	120
Operating a compressor for pneumatics	122
Operating pneumatic cylinders - Solenoids	124
Using CAN Devices	127
Using the CAN subsystem with the RoboRIO	128
Pneumatics Control Module	129
Power Distribution Panel	130
Talon SRX CAN	132
WPILib sensors	133
WPILib Sensor Overview	134
Switches - Using limit switches to control behavior	135
How do I do _____? - Selecting the right sensor for the job	141
Accelerometers - measuring acceleration and tilt	147
Gyros - Measuring rotation and controlling robot driving direction	153
Ultrasonic Sensors - Measuring robot distance to a surface	158

FRC Java Programming

Counters - Measuring rotation, counting pulses and more	162
Encoders - Measuring rotation of a wheel or other shaft	168
Analog inputs	173
Potentiometers - Measuring joint angle or linear motion	179
Analog triggers	182
Operating the robot with feedback from sensors (PID control)	185
Driver Station Inputs and Feedback	189
Driver Station Input Overview	190
Joysticks.....	195
Displaying Data on the DS - Dashboard Overview	201
Command based programming.....	202
What is Command based programming?	203
Creating a robot project.....	209
Adding Commands and Subsystems to the project	210
Simple subsystems	214
PIDSubsystems for built-in PID control.....	217
Creating Simple Commands.....	219
Creating groups of commands	225
Running commands on Joystick input.....	229
Running commands during the autonomous period	233
Converting a Simple Autonomous program to a Command based autonomous program	237
Default Commands.....	248
Synchronizing two commands.....	250

FRC Java Programming

- Scheduling commands..... 254
- Using limit switches to control behavior..... 261


Setting up the Development Environment

Installing C++ and Java Development Tools for FRC

Windows

Offline Installer

Note:

 Windows 7: You must install the NI Update or .NET Version 4.62 (or later) before proceeding with the install of VSCode for FRC. The NI Update installer will automatically install the proper version of .NET. The stand alone .NET installer is [here: https://support.microsoft.com/en-us/help/3151800/the-net-framework-4-6-2-offline-installer-for-windows](https://support.microsoft.com/en-us/help/3151800/the-net-framework-4-6-2-offline-installer-for-windows)

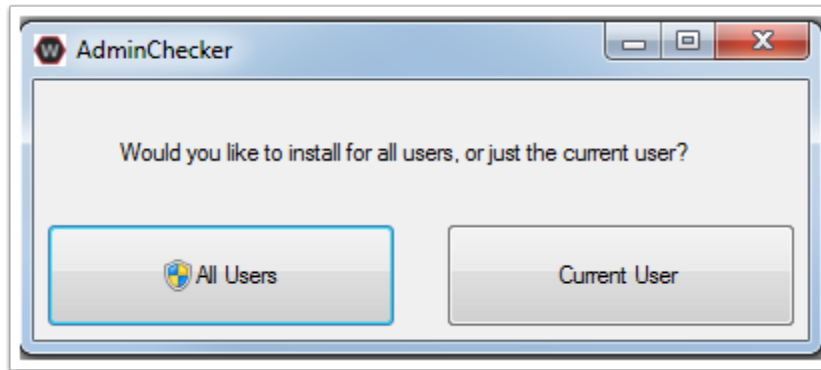
Download the appropriate offline installer for your Windows installation (32 bit or 64 bit). If you're not sure, open the Control Panel -> System to check.

For Beta, these installers are found in the File Releases section of the Teamforge Beta project.

Double click on the installer to run it. If you see any Security warnings, click Run (Windows 7) or More Info->Run Anyway (Windows 8+).

FRC Java Programming

Installation Type

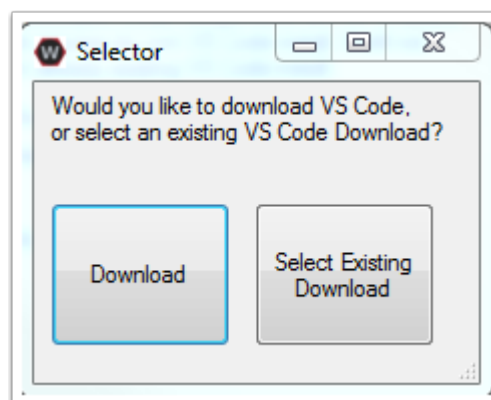


Choose whether to install for All Users on the machine or the Current User. The All Users option requires Admin privileges, but installs in a way that is accessible to all user accounts, the Current User install is only accessible from the account it is installed from.

If you select All Users, you will need to accept the security prompt that appears.

Download VSCode

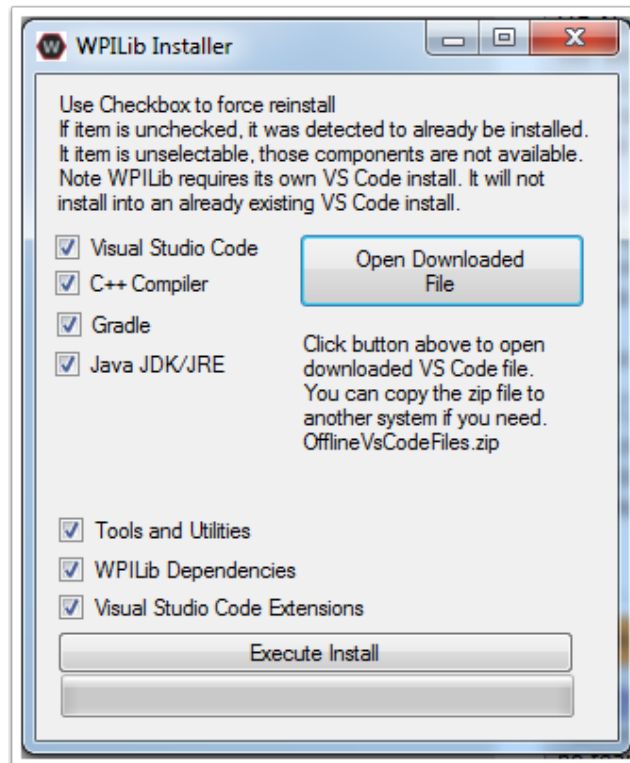
For licensing reasons, the installer cannot contain the VSCode installer bundled in. Click Select/Download VSCode to either Download the VSCode installer or select a pre-downloaded copy. If you intend to install on other machines without internet connections, after the download completes, you can click Open Downloaded File to be taken to the zip file on the file system to copy along with the Offline Installer.



FRC Java Programming

Execute Install

Make sure all checkboxes are checked (unless you have already installed 2019 WPILib software on this machine and the software unchecked them automatically), then click Execute Install.



What's Installed?

The Offline Installer installs the following components:

- Visual Studio Code - The supported IDE for 2019 robot code development. The offline installer sets up a separate copy of VSCode for WPILib development, even if you already have VSCode on your machine. This is done because some of the settings that make the WPILib setup work may break existing workflows if you use VSCode for other projects.
- C++ Compiler - The toolchains for building C++ code for the roboRIO
- Gradle - The specific version of Gradle used for building/deploying C++ or Java robot code

FRC Java Programming

- Java JDK/JRE - A specific version of the Java JDK/JRE that is used to build Java robot code and to run any of the Java based Tools (Dashboards, etc.). This exists side by side with any existing JDK installs and does not overwrite the JAVA_HOME variable
- WPILib Tools - SmartDashboard, Shuffleboard, Robot Builder, Outline Viewer, Pathweaver
- WPILib Dependencies - OpenCV, etc.
- VSCode Extensions - WPILib extensions for robot code development in VSCode

What's Installed - Continued

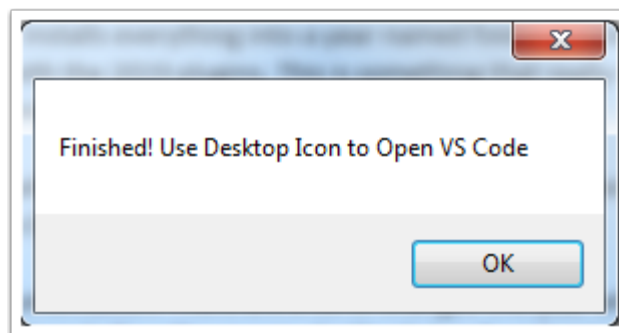
The Offline Installer also installs a Desktop Shortcut to the WPILib copy of VSCode and sets up a command shortcut so this copy of VSCode can be opened from the command line using the command "frccode2019"

Both of these reference the specific year as the WPILib C++\Java tools will now support side-by-side installs of multiple environments from different seasons.



Finished!

When the installer completes, you will now be able to open and use the WPILib version of VSCode. If you are using any 3rd party libraries, you will still need to install those separately before using them in robot code.



FRC Java Programming

Mac OS

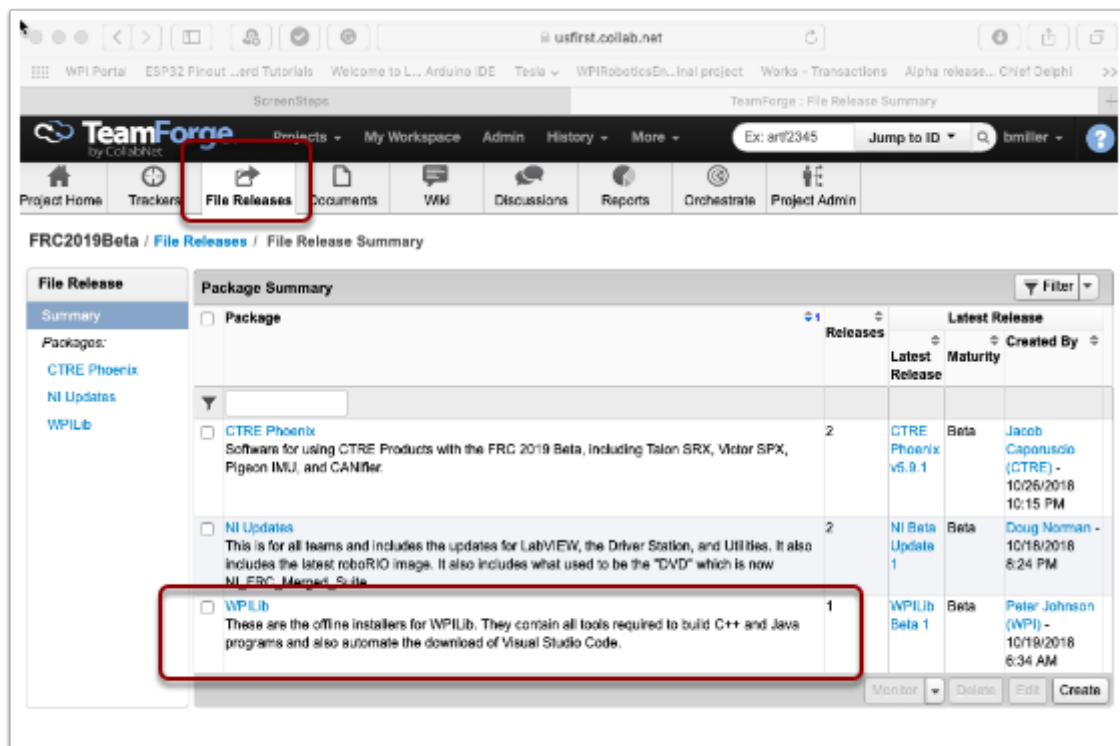
The beta tools (except the Driver Station and the roboRIO Imaging Tool) will run natively on a Mac.

Note: if you have the alpha release of VSCode for FRC installed, you should uninstall it before proceeding or create a new VSCode install. Failing to do this will have both versions installed at the same time causing things to not operate properly.

To install it follow these steps.

Download and move the directory

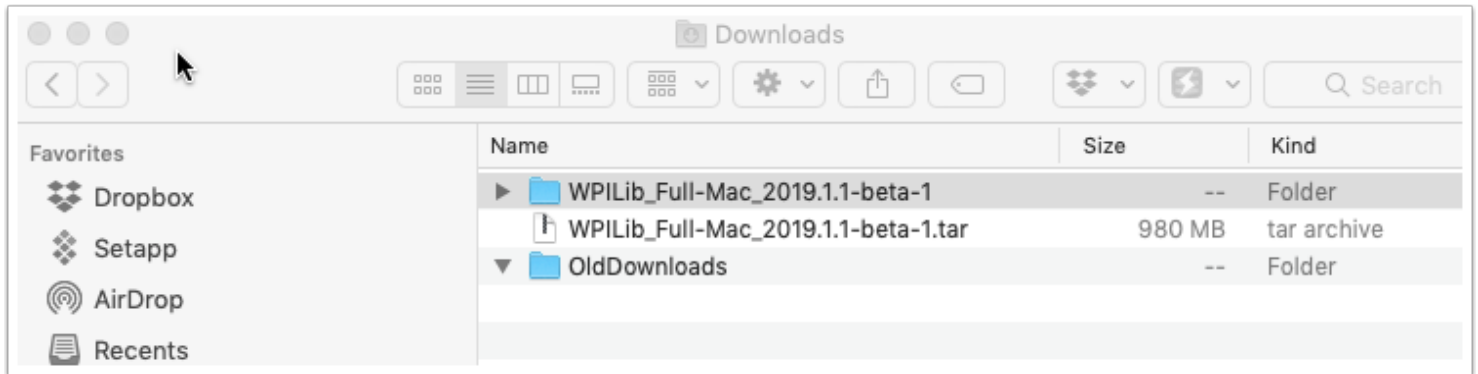
Download the software release by opening the File Releases tab of the FRC2019 Project in your browser. Then select the WPILib package as shown.



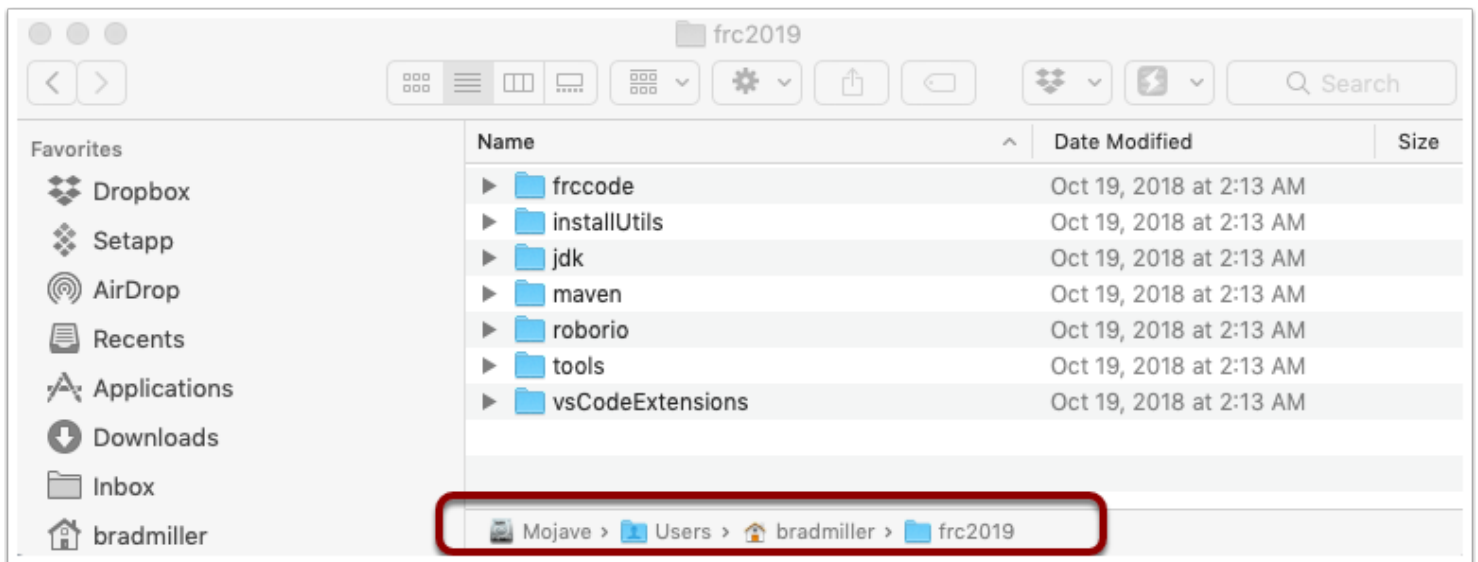
Then select the latest WPILib Beta and download the file WPILib_Full-Mac_2019.1.1-beta-1.tar.gz (the version number of this file may be different as newer updates are released). Uncompress the file by opening the downloads folder in Finder and double-clicking on the downloaded file .gz file

FRC Java Programming

(the gz file may have already been uncompressed depending on your settings). Then double-click on the .tar file so that the uncompressed folder is showing in Finder.



Using Finder (or command line) copy the contents of the folder to a new folder in your home directory, ~/frc2019 as shown below.

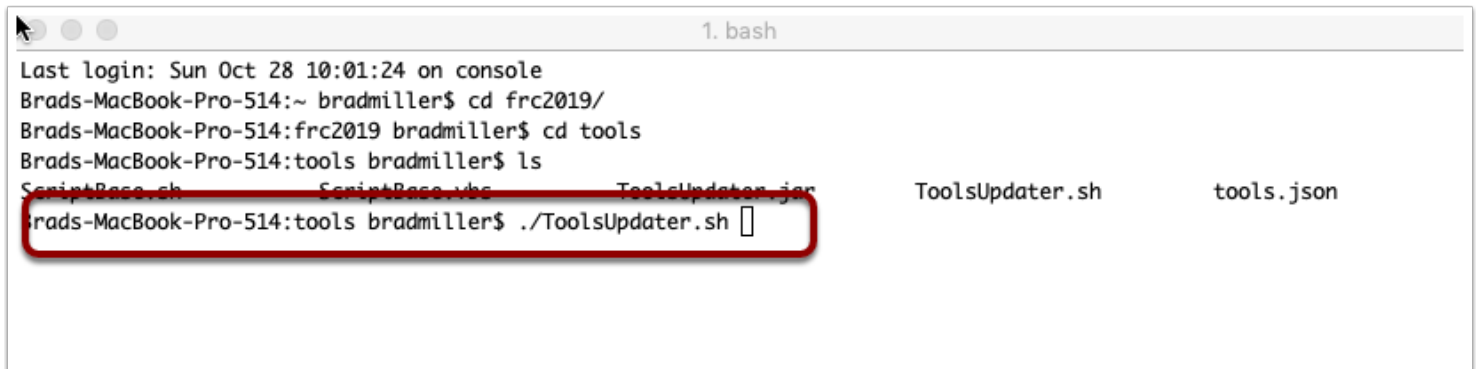


❗ **Known Issue:** The ToolsUpdater.sh script referenced in the next step does not currently work correctly on Mac (as of Beta 2), Skip that step and proceed to "Setting up VSCode..."

FRC Java Programming

Run the ToolsUpdater.sh script

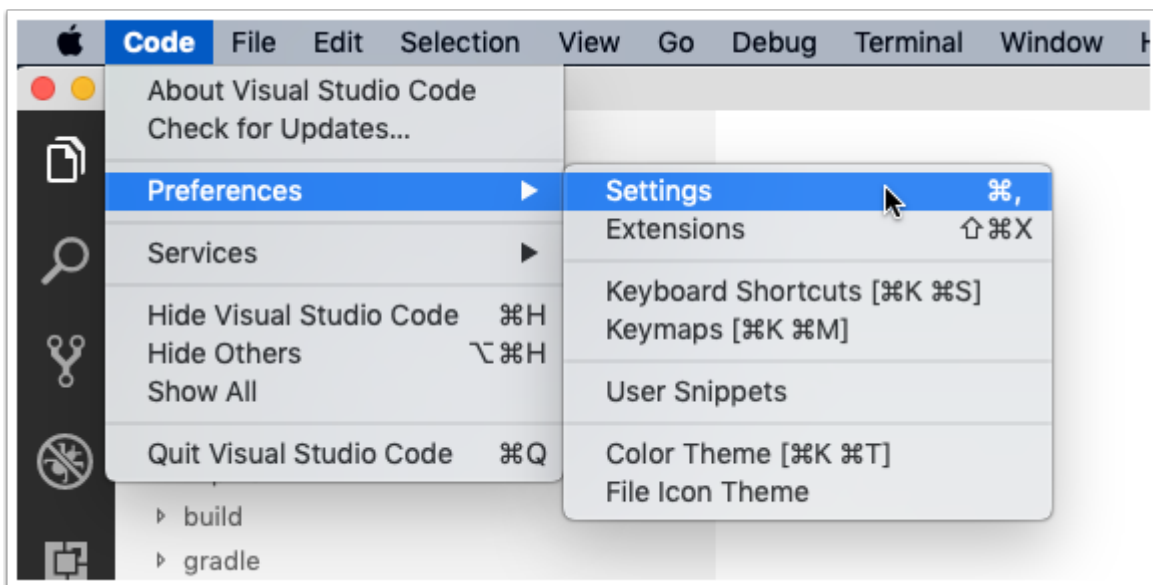
Open a terminal window and change directory to ~/frc2019/tools and run the script, ToolsUpdater.sh.



```
1. bash
Last login: Sun Oct 28 10:01:24 on console
Brads-MacBook-Pro-514:~ bradmiller$ cd frc2019/
Brads-MacBook-Pro-514:frc2019 bradmiller$ cd tools
Brads-MacBook-Pro-514:tools bradmiller$ ls
ScriptBase.ch  ScriptBase.vbs  ToolsUpdater.jar  ToolsUpdater.sh  tools.json
Brads-MacBook-Pro-514:tools bradmiller$ ./ToolsUpdater.sh
```

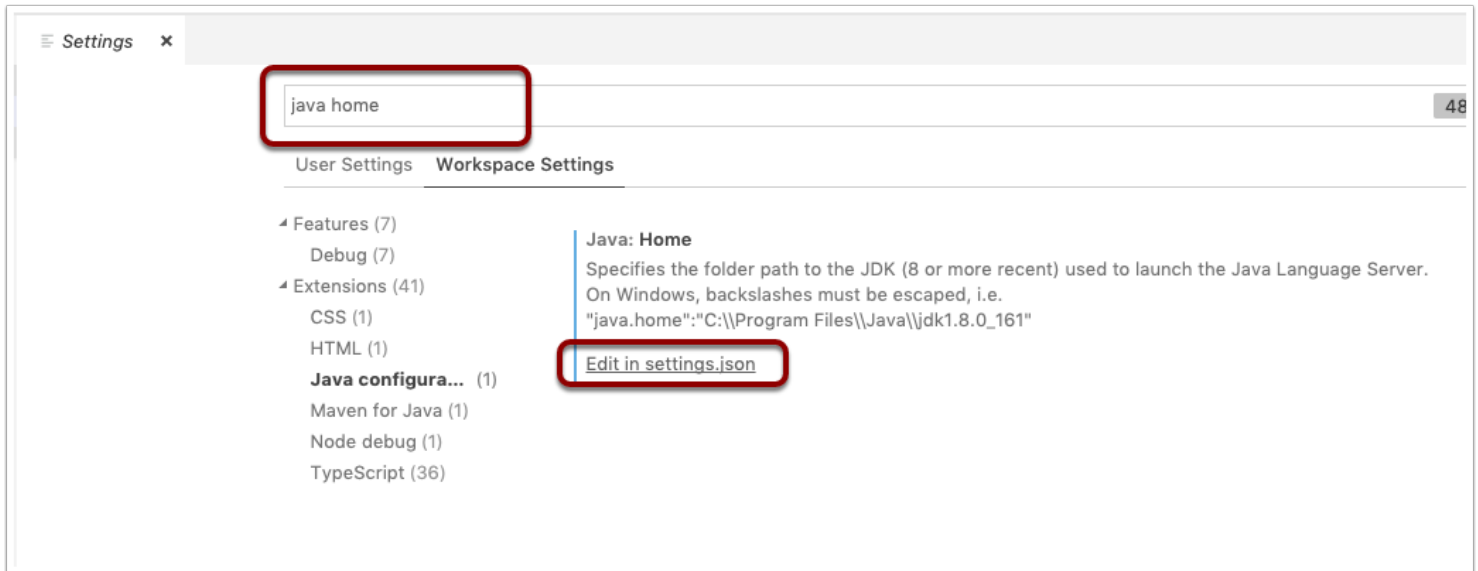
Setting up VSCode to use Java 11

Java 11 is required to be installed on systems that use VSCode or the 2019 tools. In VSCode it is required to set the "java.home" preference for the user or workspace to be the directory of the Java 11 home. To do that open the settings by selecting Settings from the Preference menu.



FRC Java Programming

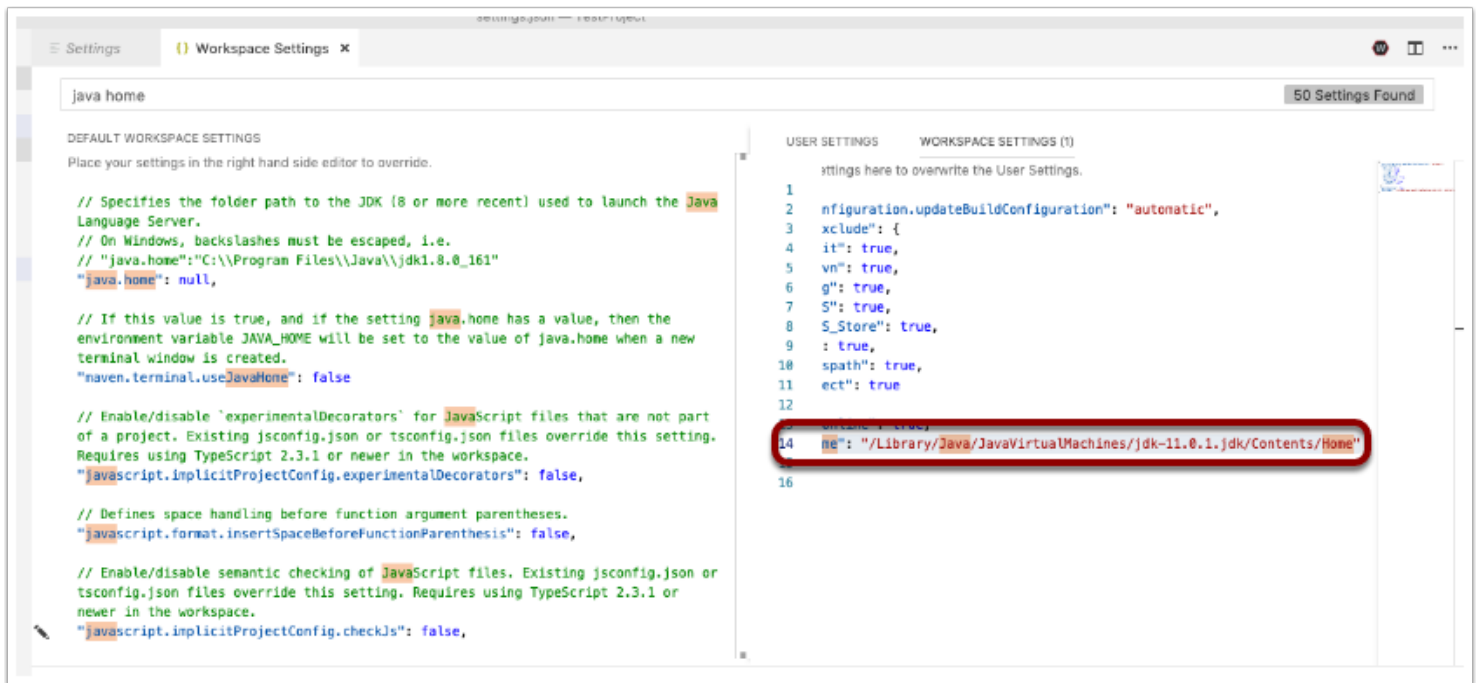
Search for the java home settings by entering "java home" into the search window. Then select "Edit in settings.json" to make changes to the java home setting.



In the settings file you must edit the java.home variable in the right hand pane as shown in the screen capture below. On the default install of Java 11 on Mojave, the line is shown in the following picture. The text is:

"java.home": "/Library/Java/JavaVirtualMachines/jdk-11.0.1.jdk/Contents/Home"

FRC Java Programming




Installing the FRC Update Suite (All Languages)

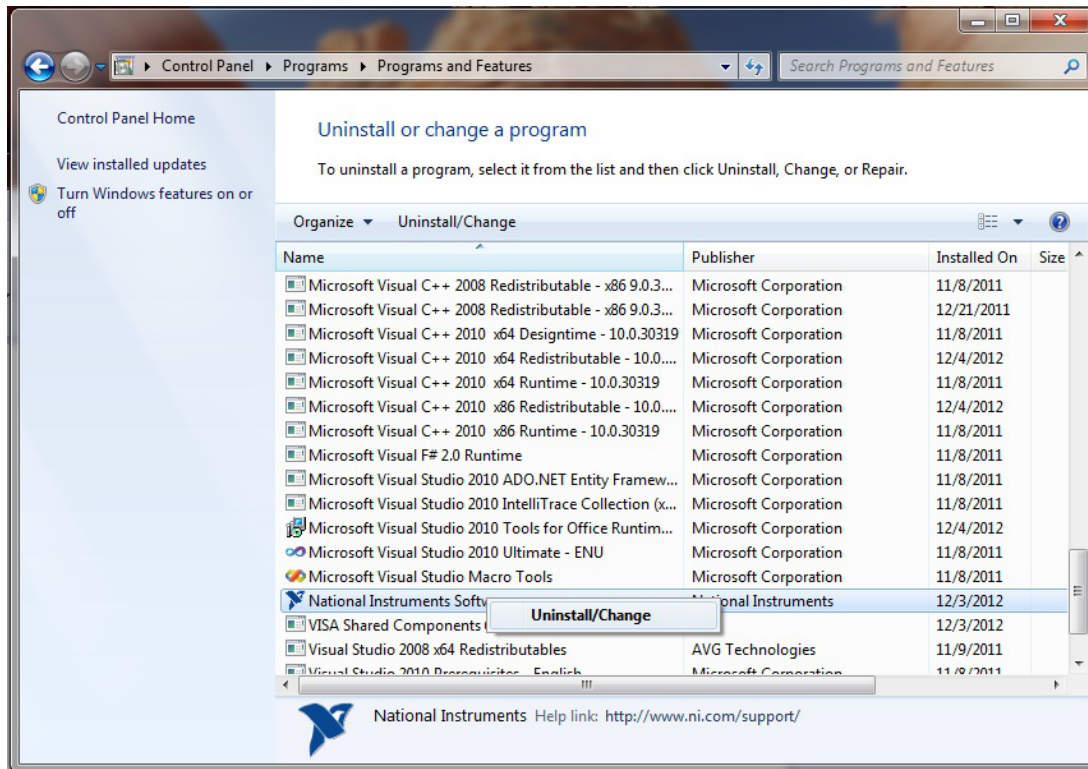


The FRC Update Suite contains the following software components: LabVIEW Update, FRC Driver Station, and FRC Utilities. If an FRC LabVIEW installation is found, the LabVIEW Update will be installed or updated, otherwise this step will be skipped. The FRC Driver Station and FRC Utilities will always be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

C++ and Java teams wishing to use NI Vision Assistant should run the full Suite installer as described in the article - [Installing LabVIEW for FRC \(LabVIEW only\)](#)

 **Note:** The Driver Station will only work on Windows 7, Windows 8, Windows 8.1, and Windows 10. It will not work on Windows XP.

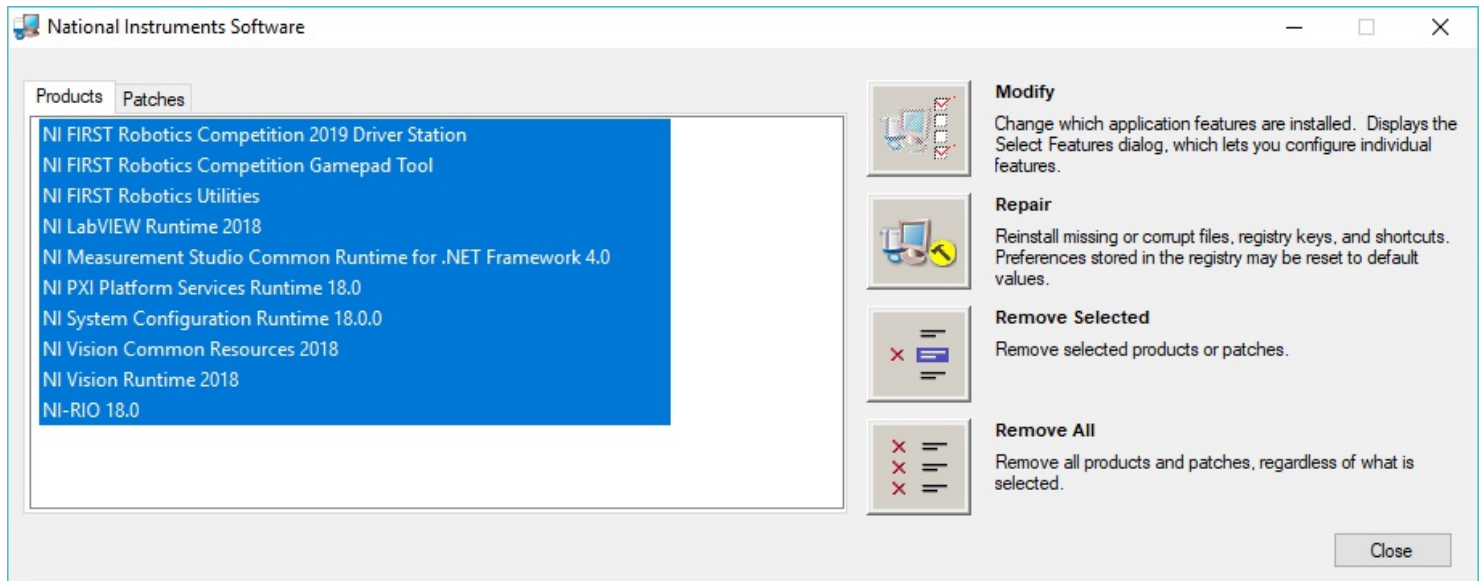
Uninstall Old Versions (Recommended)



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 2019 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.

FRC Java Programming

Select Components to Uninstall



Click **Remove All** and follow any prompts to remove all previous NI products.

Downloading the Update

Download the update from <http://www.ni.com/download/first-robotics-software-2017/7183/en/>



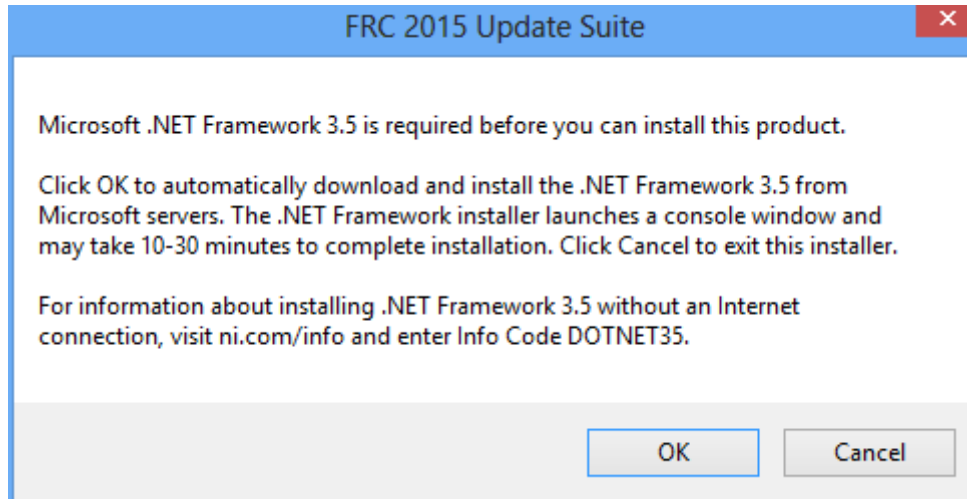
Note: This download will require the decryption key from the Kickoff broadcast

.NET Framework 4.6.2

The Update installer may prompt that .NET Framework 4.6.2 needs to be updated or installed. Follow prompts on-screen to complete the installation, including rebooting if requested. Then resume the installation of the NI FRC Update, restarting the installer if necessary.

FRC Java Programming

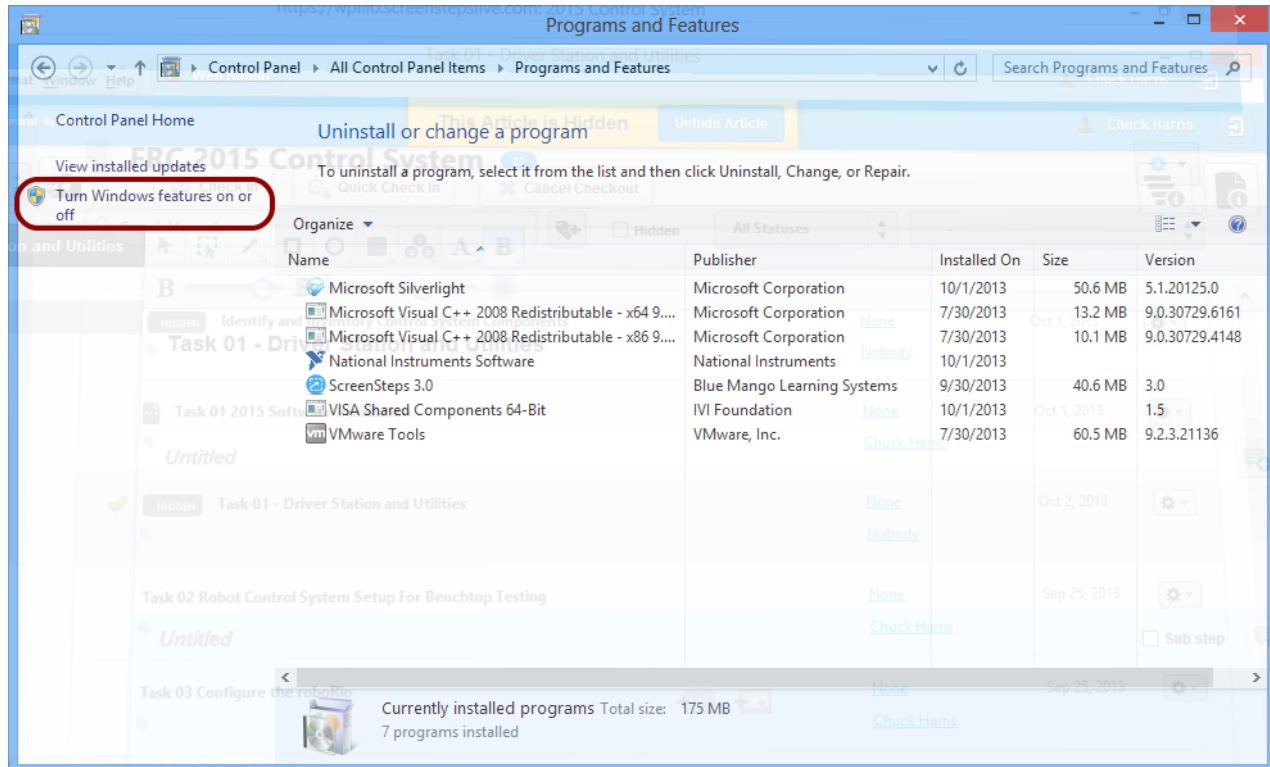
.NET Framework 3.5



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.

FRC Java Programming

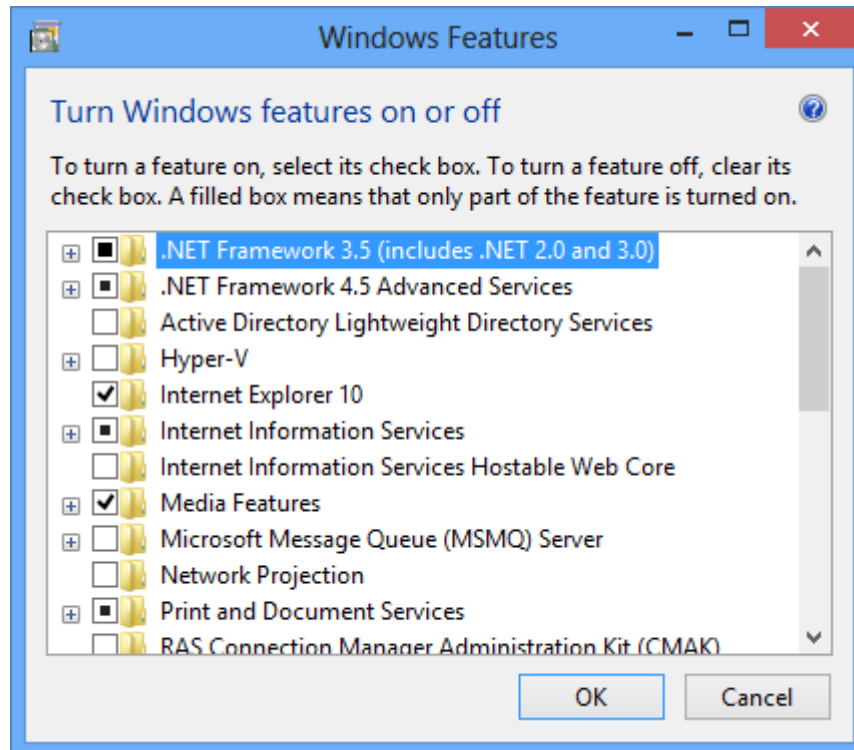
Programs and Features



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

FRC Java Programming

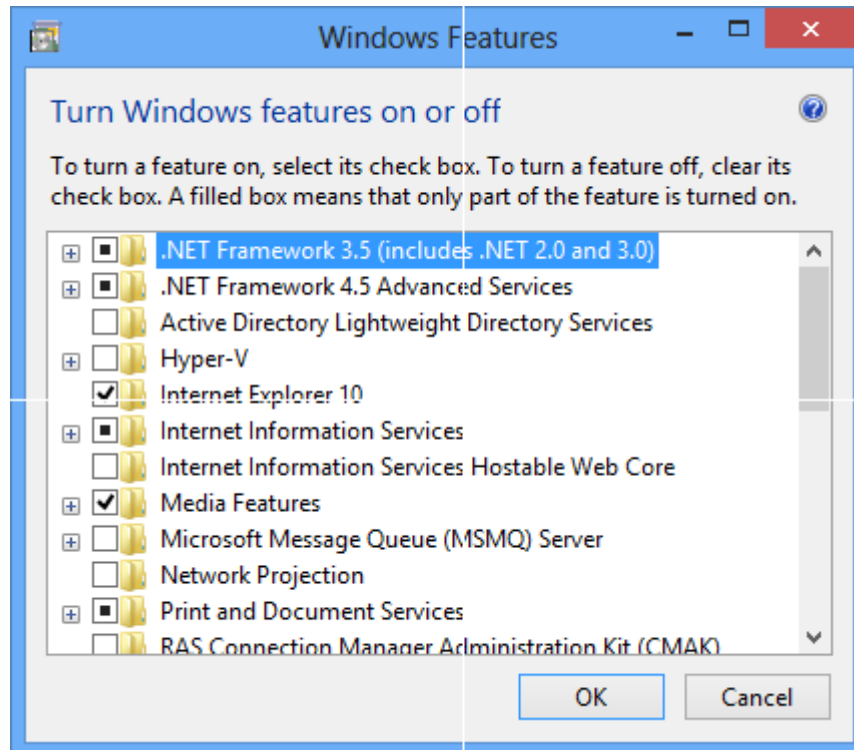
Windows Features (.NET Framework 3.5 not on)



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 2019 Update Suite](#).

FRC Java Programming

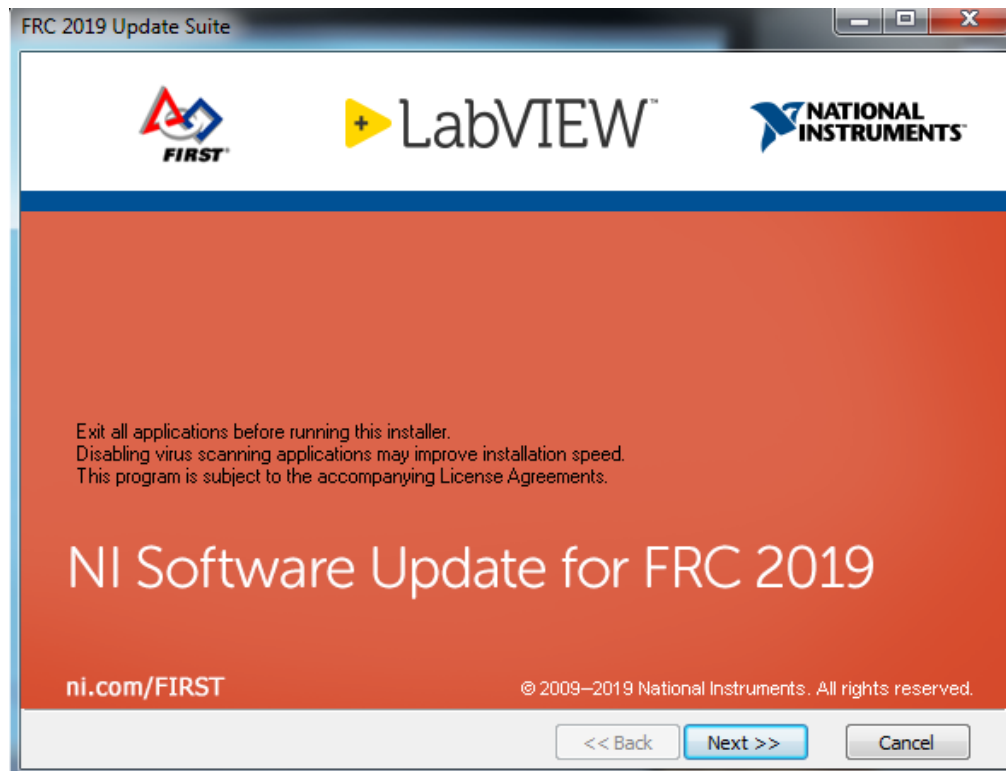
Windows Features (.NET Framework 3.5 already on)



If a black dot is shown next to ".NET Framework 3.5" the feature is already on. Click "Cancel" and [restart installation of FRC 2019 Update Suite](#).

FRC Java Programming

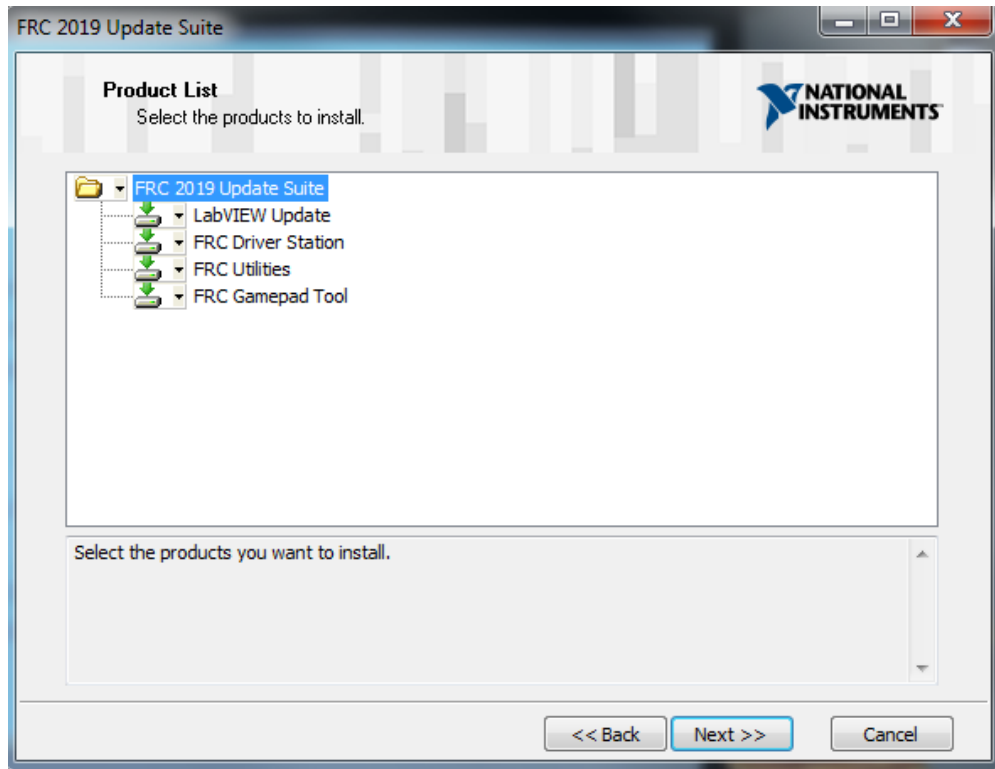
Welcome



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 will be released at Kickoff. 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.

FRC Java Programming

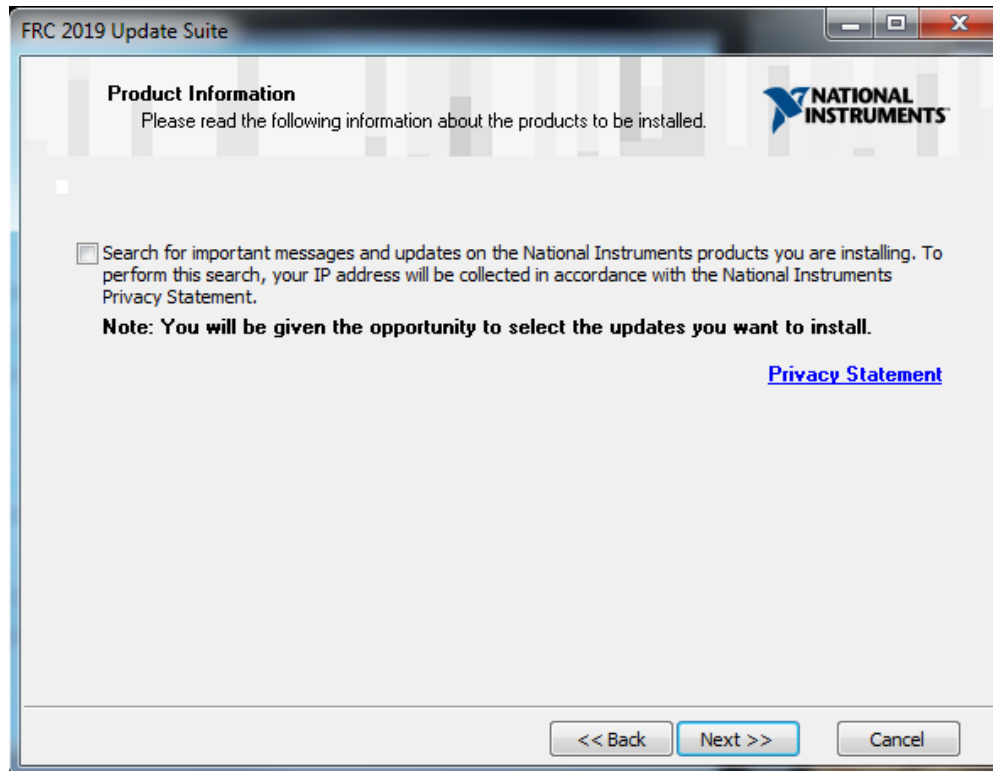
Product List



Click "Next". There is no need to de-select "LabVIEW Update" for C++ or Java teams, if you do not have the base LabVIEW installation (because you are not programming in LabVIEW) this installation will be skipped automatically.

FRC Java Programming

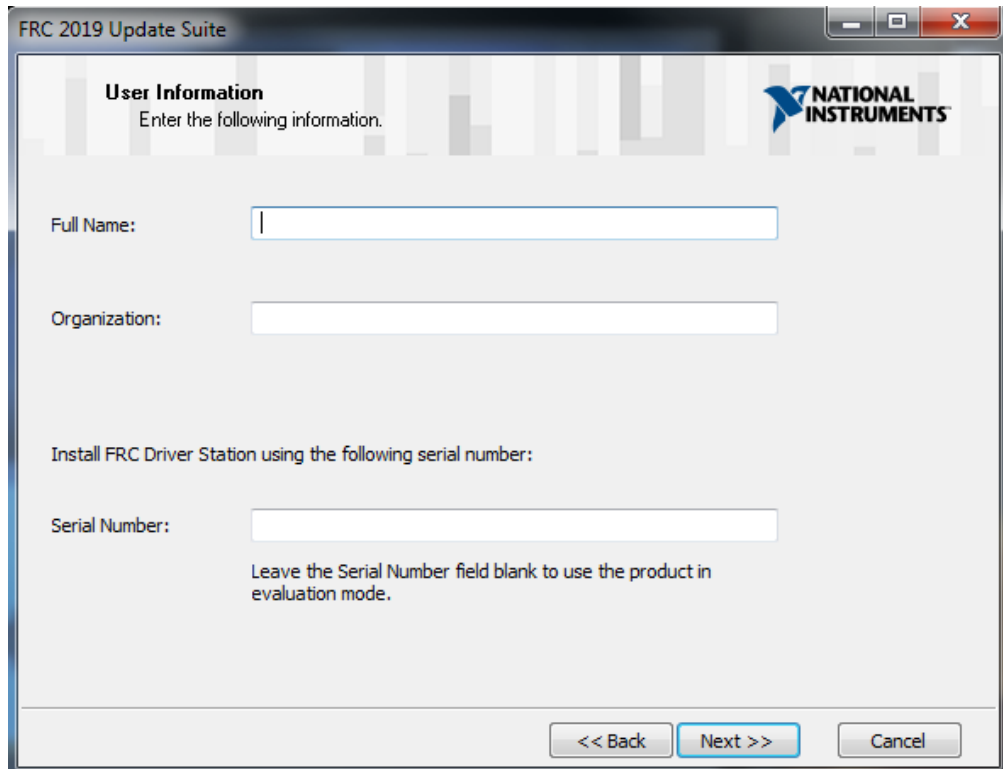
Product Information



Un-check the box, then Click "Next".

FRC Java Programming

User Information

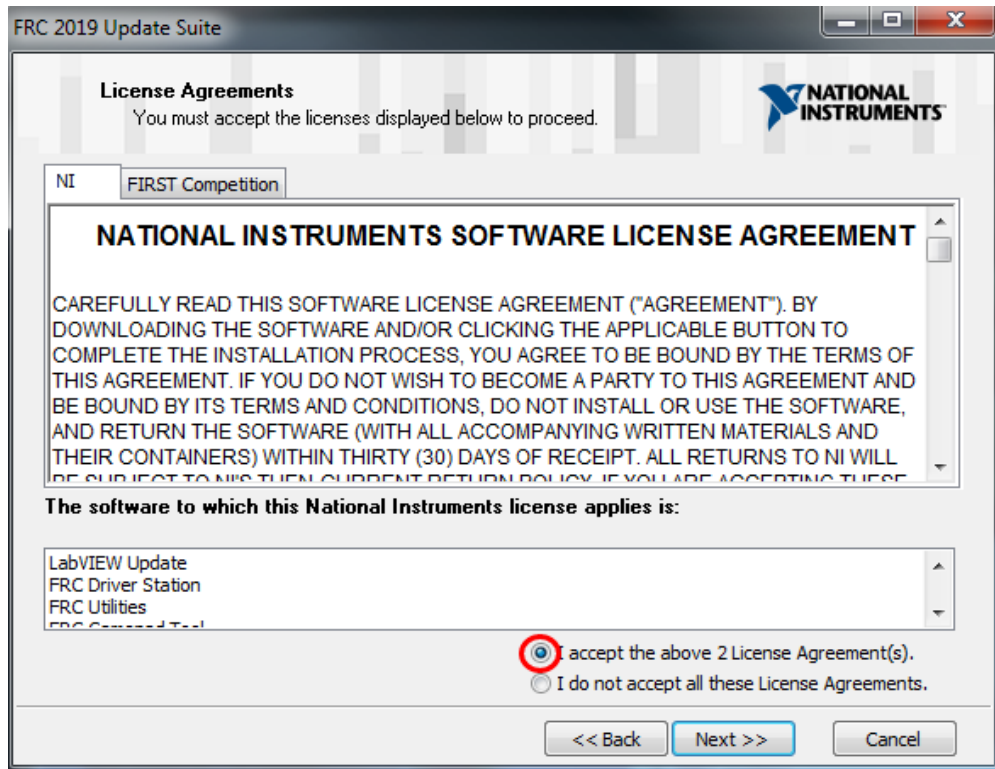


The screenshot shows a Windows-style dialog box titled "FRC 2019 Update Suite". Inside the dialog, the title "User Information" is displayed above the instruction "Enter the following information." The National Instruments logo is in the top right corner. There are three input fields: "Full Name:" with a text box containing a cursor, "Organization:" with an empty text box, and "Serial Number:" with an empty text box. Below the "Serial Number:" field, a note states: "Leave the Serial Number field blank to use the product in evaluation mode." At the bottom of the dialog are three buttons: "<< Back" (disabled), "Next >>" (active/highlighted), and "Cancel" (disabled).

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

FRC Java Programming

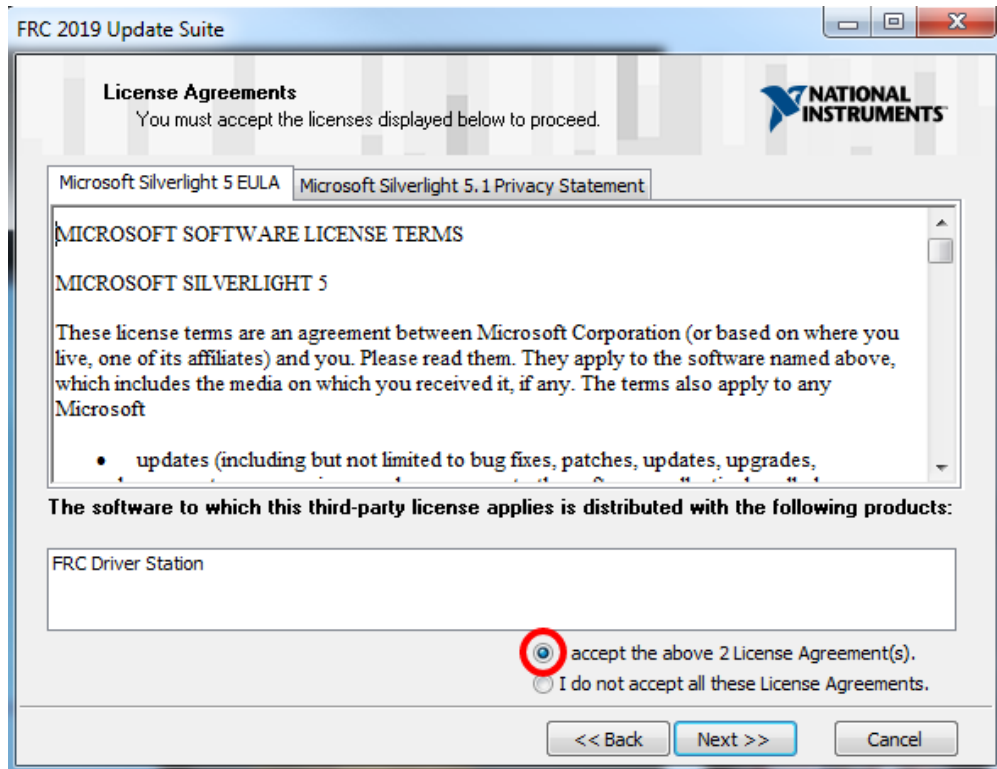
License Agreements



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

FRC Java Programming

License Agreements Page 2



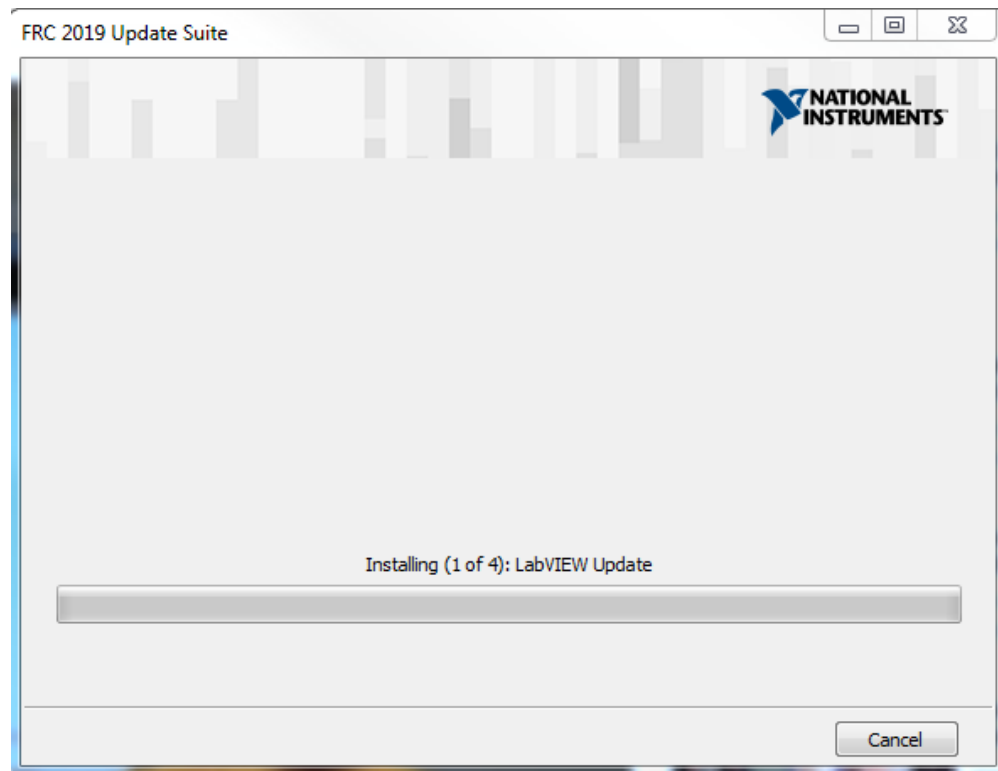
Select "I accept..." then click "Next"

If you see a screen asking to disable Windows Fast Startup, leave it at the recommended option (disable Fast Startup) and click Next.

If you see a screen talking about Windows Firewall, click Next.

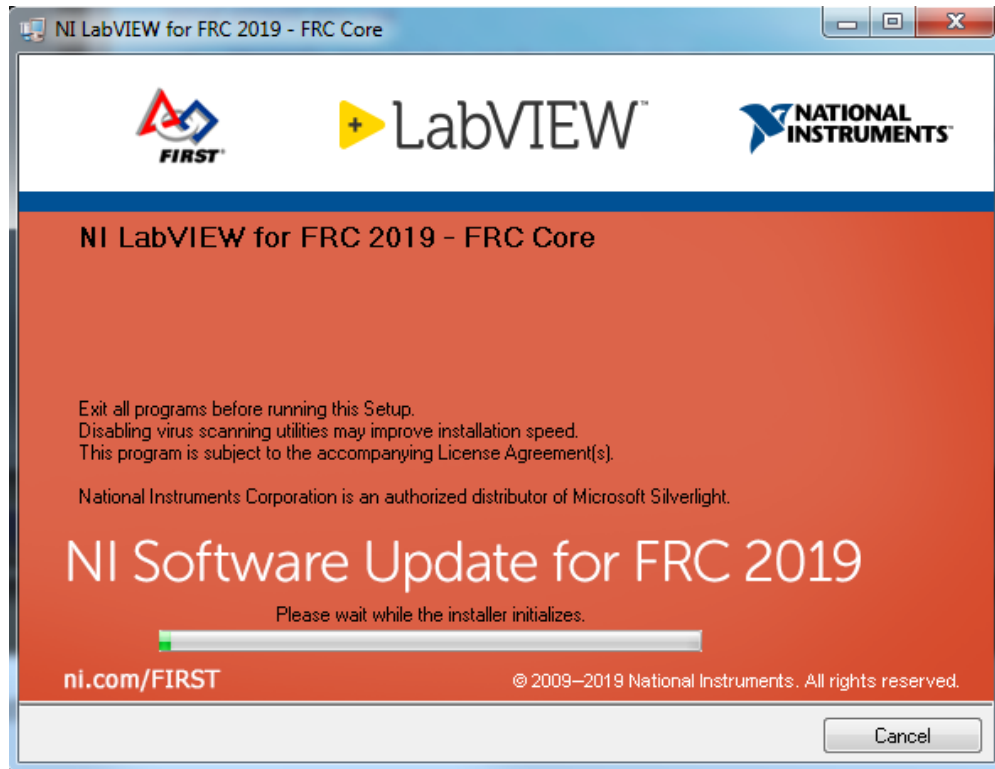
FRC Java Programming

Summary Progress



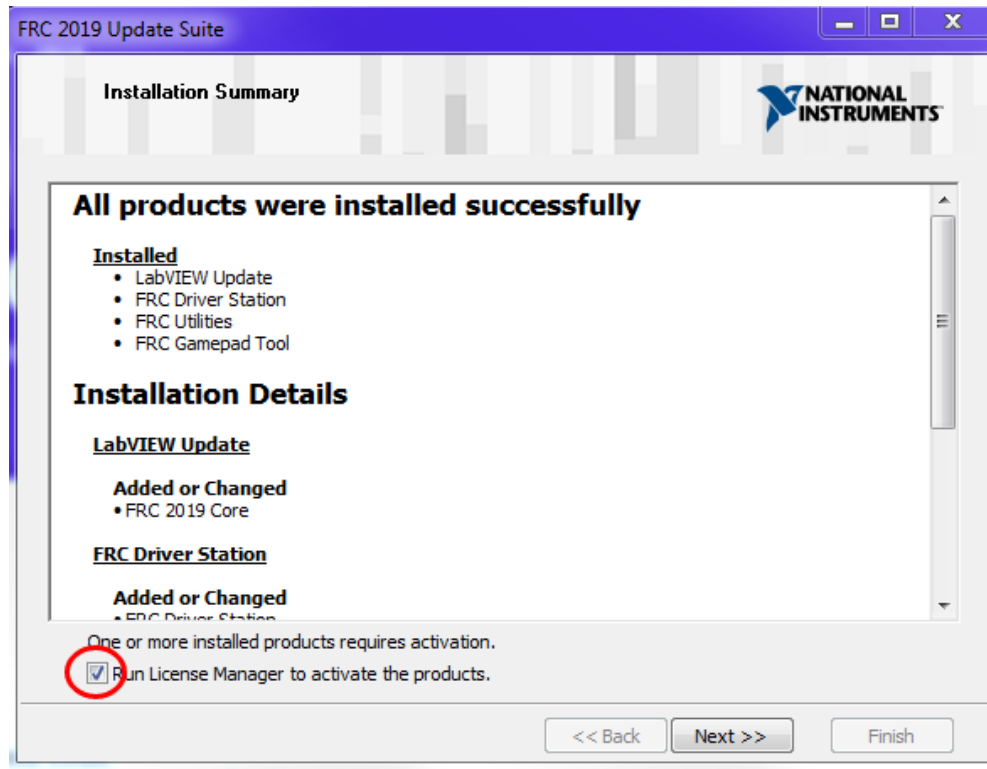
FRC Java Programming

Detail Progress



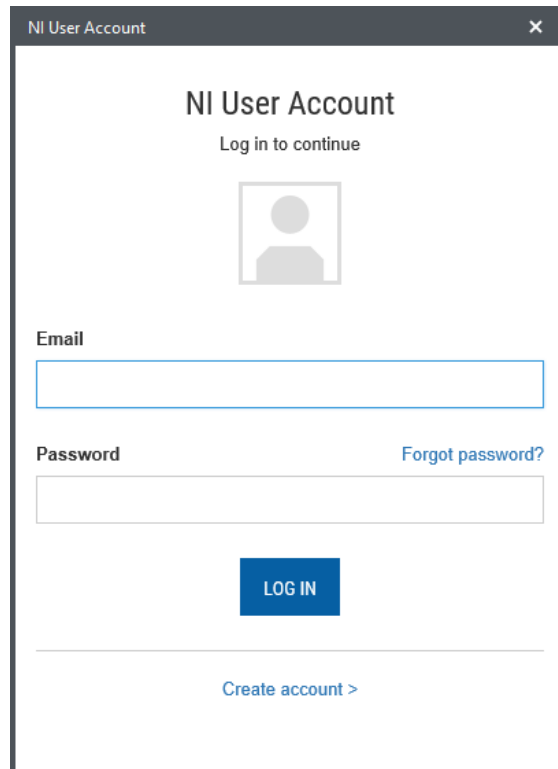
FRC Java Programming

Installation Summary



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

NI Activation Wizard



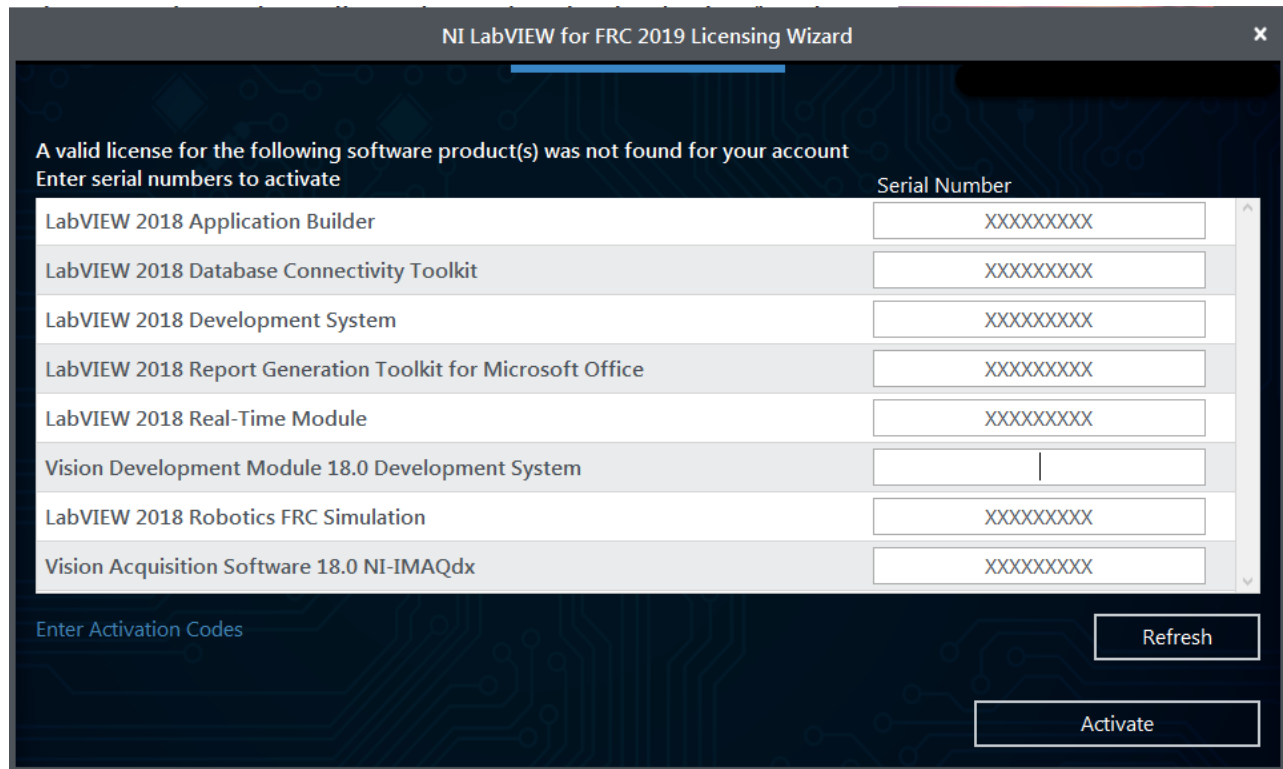
The image shows a screenshot of a web browser window titled "NI User Account". The window has a dark gray header bar with the title and a close button (X). The main content area is white and contains the following elements:

- NI User Account**: The main heading.
- Log in to continue**: A sub-heading.
- Profile Picture Placeholder**: A square box with a gray silhouette of a person's head and shoulders.
- Email**: A label above a text input field.
- Password**: A label above a text input field.
- Forgot password?**: A link in blue text next to the password field.
- LOG IN**: A blue button with white text.
- Create account >**: A link in blue text at the bottom.

Log into your ni.com account. If you don't have an account, select 'Create account' to create a free account.

FRC Java Programming

NI Activation Wizard (2)



NI LabVIEW for FRC 2019 Licensing Wizard

A valid license for the following software product(s) was not found for your account
Enter serial numbers to activate

	Serial Number
LabVIEW 2018 Application Builder	XXXXXXXXXX
LabVIEW 2018 Database Connectivity Toolkit	XXXXXXXXXX
LabVIEW 2018 Development System	XXXXXXXXXX
LabVIEW 2018 Report Generation Toolkit for Microsoft Office	XXXXXXXXXX
LabVIEW 2018 Real-Time Module	XXXXXXXXXX
Vision Development Module 18.0 Development System	
LabVIEW 2018 Robotics FRC Simulation	XXXXXXXXXX
Vision Acquisition Software 18.0 NI-IMAQdx	XXXXXXXXXX

Enter Activation Codes

Refresh

Activate

The serial number you entered at the "User Information" screen should appear in all of the text boxes, if it doesn't, enter it now. Click "Activate".

Note: If this is the first time activating the 2019 software on this account, you will see the message shown above about a valid license not being found. You can ignore this.

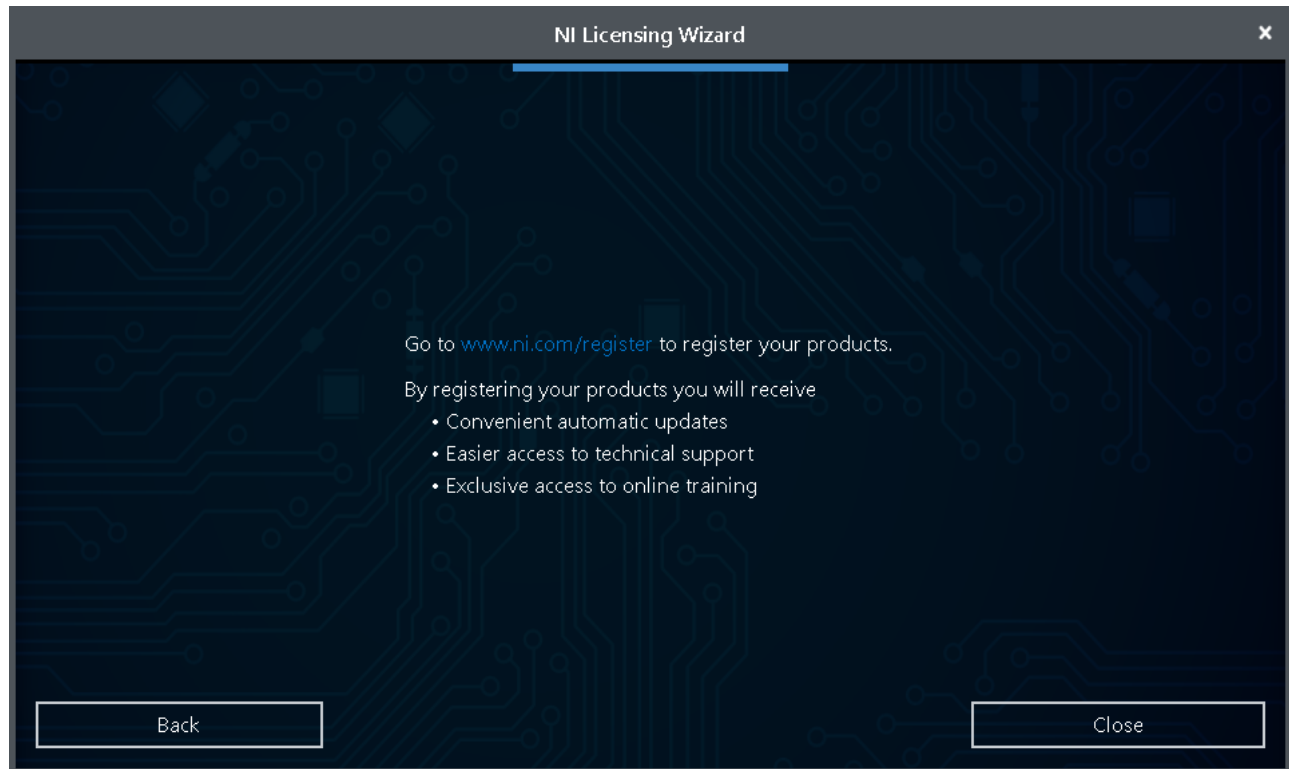
FRC Java Programming

NI Activation Wizard (3)



If your products activate successfully, an “Activation Successful” message will appear. If the serial number was incorrect, it will give you a text box and you can re-enter the number and select “Try Again”. If everything activated successfully, click “Next”.

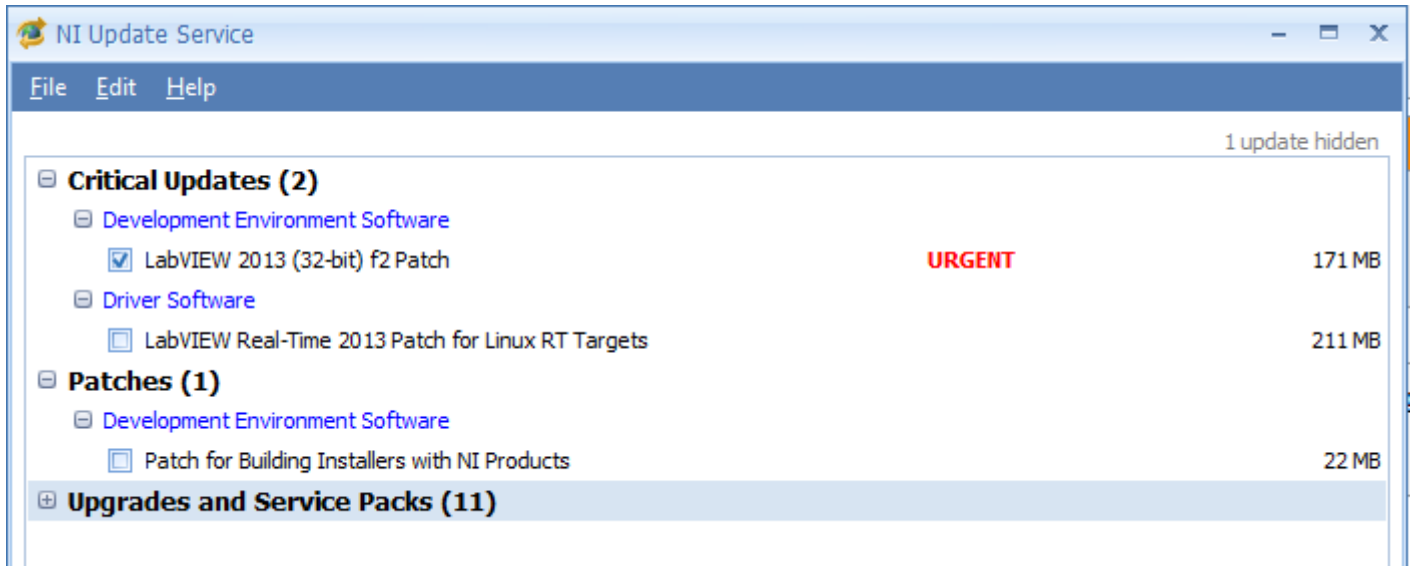
NI Activation Wizard (4)



Click "Close".

FRC Java Programming

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).

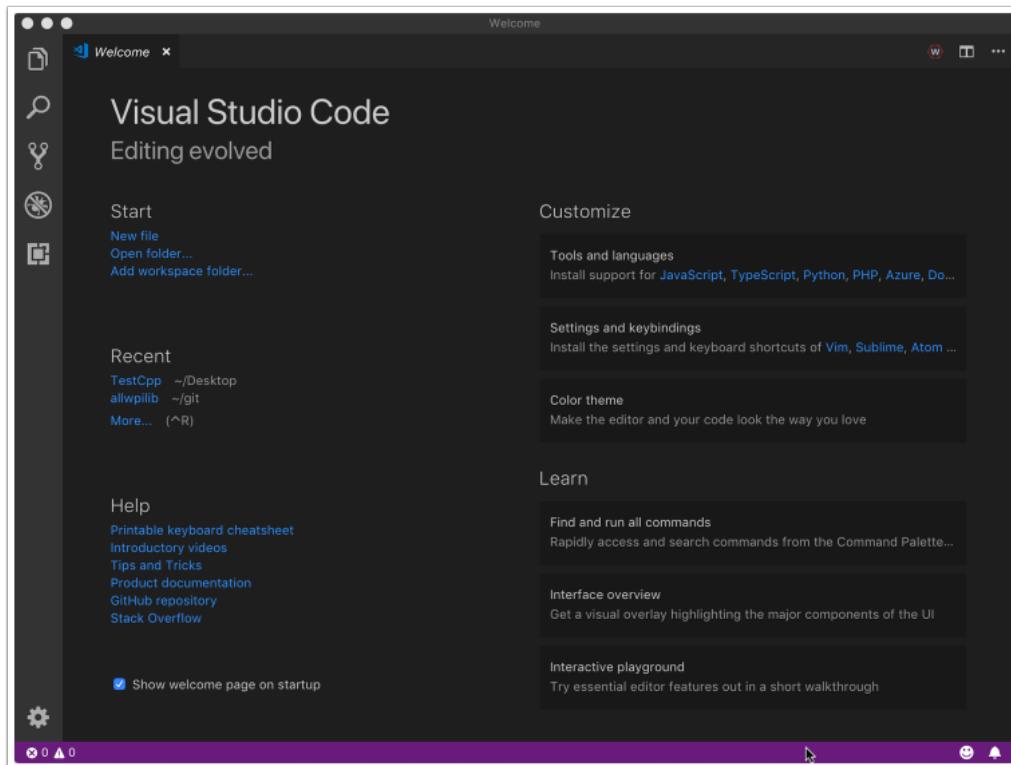
Creating and Running Robot Programs

Visual Studio Code Basics and the WPILib Extension

Microsoft's Visual Studio Code is the new supported IDE for C++ and Java development in FRC, replacing the Eclipse IDE used from 2015-2018. This article introduces some of the basics of using Visual Studio Code and the WPILib extension.

⚠ Note: If you used the publicly available Visual Studio Code Alpha or the closed Beta, you should create a new project or re-import your Eclipse project. There were breaking changes made to some of the configuration files (such as build.gradle) between releases.

Welcome Page



FRC Java Programming

When Visual Studio Code first opens, you are presented with a Welcome page. On this page you will find some quick links that allow you to customize Visual Studio Code as well as a number of links to help documents and videos that may help you learn about the basics of the IDE as well as some tips and tricks.

You may also notice a small WPILib logo way up in the top right corner. This is one way to access the features provided by the WPILib extension (discussed further below).

User Interface

The most important link to take a look at is probably the basic [User Interface document](#). This document describes a lot of the basics of using the UI and provides the majority of the information you should need to get started using Visual Studio Code for FRC.

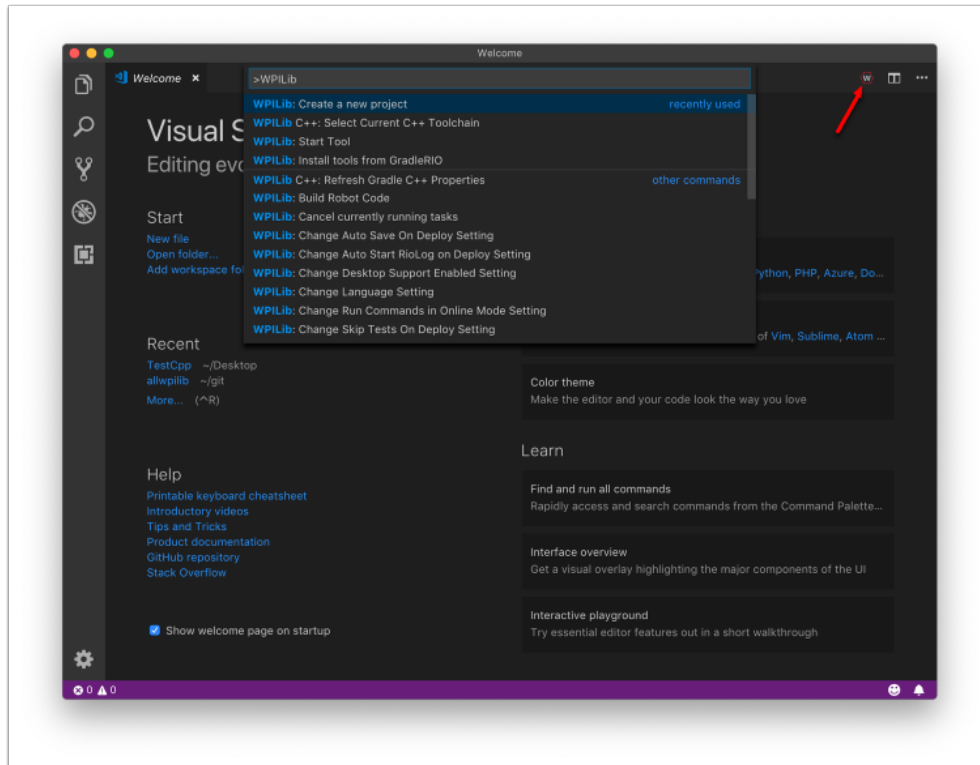
Command Palette

The Command Palette can be used to access or run almost any function or feature in Visual Studio Code (including those from the WPILib extension). The Command Palette can be accessed from the View menu or by pressing Ctrl+Shift+P (Cmd+Shift+P on Mac). Typing text into the window will dynamically narrow the search to relevant commands and show them in the dropdown.

In the following example "wpilib" is typed into the search box after activating the Command Palette, and it narrows the list to functions containing WPILib.

FRC Java Programming

WPILib Extension



The WPILib extension provides the FRC specific functionality related to creating projects and project components, building and downloading code to the roboRIO and more. You can access the WPILib commands one of two ways:

- By typing "WPILib" into the Command Palette
- By clicking on the WPILib icon in the top right of most windows. This will open the Command Palette with "WPILib" pre-entered

For more information about specific WPILib extension commands, see the other articles in this chapter.

WPILib Commands in VSCode

This document contains a complete list of the commands provided by the WPILib VSCode Extension and what they do.

To access these commands, press Ctrl+Shift+P to open the Command Palette, then begin typing the command name as shown here to filter the list of commands. Click on the command name to execute it.

- **WPILib: Build Robot Code** - Builds open project using GradleRIO
- **WPILib: Create Project** - Create a new robot project
- **WPILib C++: Refresh C++ Intellisense** - Force an update to the C++ Intellisense configuration.
- **WPILib C++: Select current C++ toolchain** - Select the toolchain to use for Intellisense (i.e. desktop vs. roboRIO vs...). This is the same as clicking the current mode in the bottom right status bar.
- **WPILib: Cancel currently running tasks** - Cancel any tasks the WPILib extension is currently running
- **WPILib: Change Auto Save On Deploy Setting** - Change whether files are saved automatically when doing a Deploy. This defaults to Enabled.
- **WPILib: Change Auto Start RioLog on Deploy Setting** - Change whether RioLog starts automatically on deploy. This defaults to Enabled.
- **WPILib: Change Desktop Support Enabled Setting** - Change whether building robot code on Desktop is enabled. Enable this for test and simulation purposes. This defaults to Desktop Support off.
- **WPILib: Change Language Setting** - Change whether the currently open project is C++ or Java.
- **WPILib: Change Run Commands in Online Mode Setting** - Change whether GradleRIO is running in Online Mode (will attempt to automatically pull dependencies from online). Defaults to disabled (offline mode).
- **WPILib: Change Skip Tests On Deploy Setting** - Change whether to skip tests on deploy. Defaults to disabled (tests are run on deploy)
- **WPILib: Change Stop Simulation on Entry Setting** - Change whether to stop robot code on entry when running simulation. Defaults to disabled (don't stop on entry).
- **WPILib: Check for WPILib Updates** - Check for an update to the WPILib extensions
- **WPILib: Create a new class/command** - Clicking this command in the palette will not do anything. This command is triggered by right-clicking on the desired folder in the Navigation Pane and selecting the appropriate option.
- **WPILib: Debug Robot Code** - Build and deploy robot code to RoboRIO in debug mode and start debugging
- **WPILib: Deploy Robot Code** - Build and deploy robot code to RoboRIO

FRC Java Programming

- **WPILib: Import a WPILib Eclipse Project** - Open a wizard to help you create a new VS Code project from an existing WPILib Eclipse project from a previous season.
- **WPILib: Install tools from GradleRIO** - Install the WPILib Java tools (e.g. SmartDashboard, Shuffleboard, etc.). Note that this is done by default by the offline installer
- **WPILib: Manage Vendor Libraries** - Install/update 3rd party libraries
- **WPILib: Open WPILib Command Palette** - This command is used to open a WPILib Command Palette (equivalent of hitting Ctrl+Shift+P and typing WPILib)
- **WPILib: Open WPILib Help** - This opens a simple page the links to the WPILib Screensteps documentation
- **WPILib: Reset Ask for WPILib Updates Flag** - This will clear the flag on the current project, allowing you to re-prompt to update a project to the latest WPILib version if you previously chose to not update.
- **WPILib: Run a command in Gradle** - This lets you run an arbitrary command in the GradleRIO command environment
- **WPILib: Set Team Number** - Used to modify the team number associated with a project. This is only needed if you need to change the team number from the one initially specified when creating the project.
- **WPILib: Set VS Code Java Home to FRC Home** - Set the VS Code Java Home variable to point to the Java Home discovered by the FRC extension. This is needed if not using the offline installer to make sure the intellisense settings are in sync with the WPILib build settings.
- **WPILib: Show Log Folder** - Shows the folder where the WPILib extension stores internal logs. This may be useful when debugging/reporting an extension issue to the WPILib developers
- **WPILib: Simulate Robot Code on Desktop** - This builds the current robot code project on your PC and starts it running in simulation. This requires Desktop Support to be set to Enabled.
- **WPILib: Start RioLog** - This starts the RioLog display used to view console output from a robot program
- **WPILib: Start Tool** - This allows you to launch WPILib tools (e.g. SmartDashboard, Shuffleboard, etc.) from inside VSCode
- **WPILib: Test Robot Code** - This builds the current robot code project and runs any created tests. This requires Desktop Support to be set to Enabled.

Creating a robot program

The simplest way to create a robot program, is to start from one of the four supplied templates (Sample, Iterative, Timed, 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. Timed robot provides a similar structure to Iterative robot, but with more consistent timing. Command-Based robot is a template that provides a modular, extensible structure with a moderate learning curve.

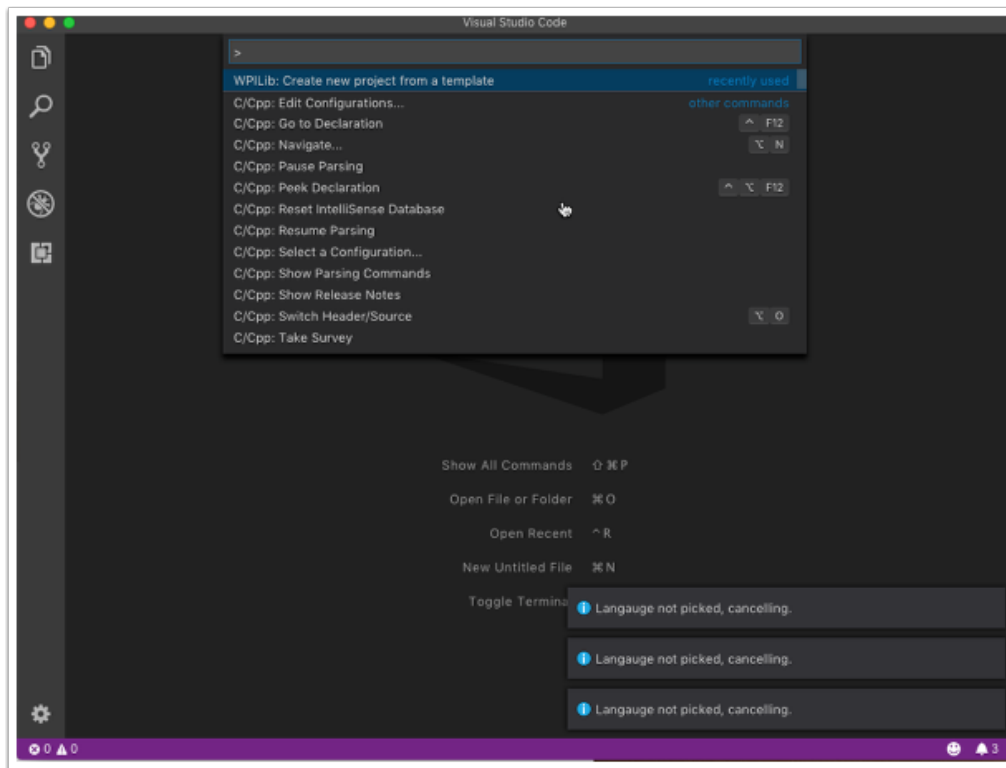
The templates will get you the basis of a robot program, organizing a larger project can often be a complex task. RobotBuilder is recommended for creating and organizing Command-Based robot programs. You can learn more about RobotBuilder [here](#). To create a command-based robot program that takes advantage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

In this article we will create a new WPILib project in Visual Studio Code. In this example we will be making a TimedRobot, however the same methods apply to creating a project from any of the existing templates or examples.

Accessing The Command Palette

Clicking Ctrl+Shift+P will open the command palette. The command palette contains the WPILib commands for creating and interacting with projects.

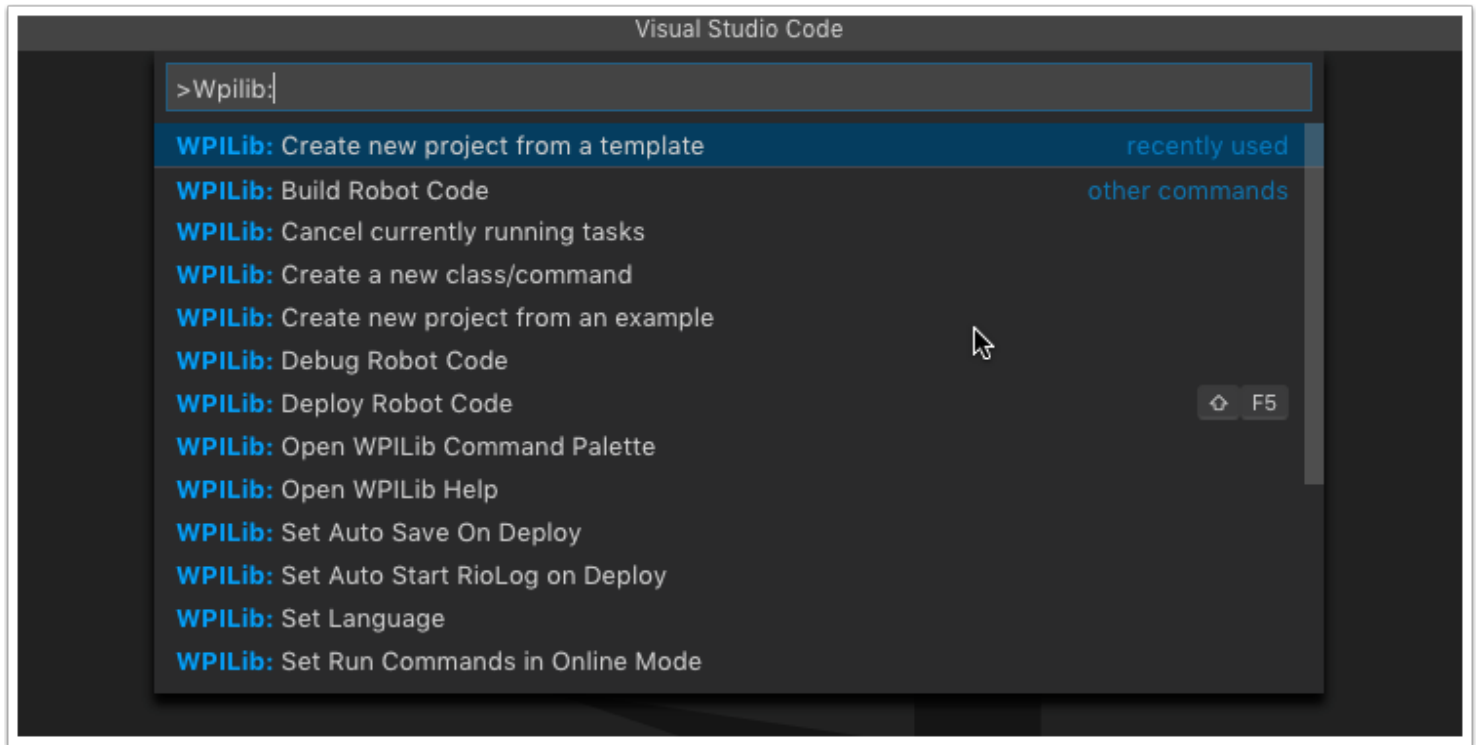
FRC Java Programming



Accessing The WPILib Commands

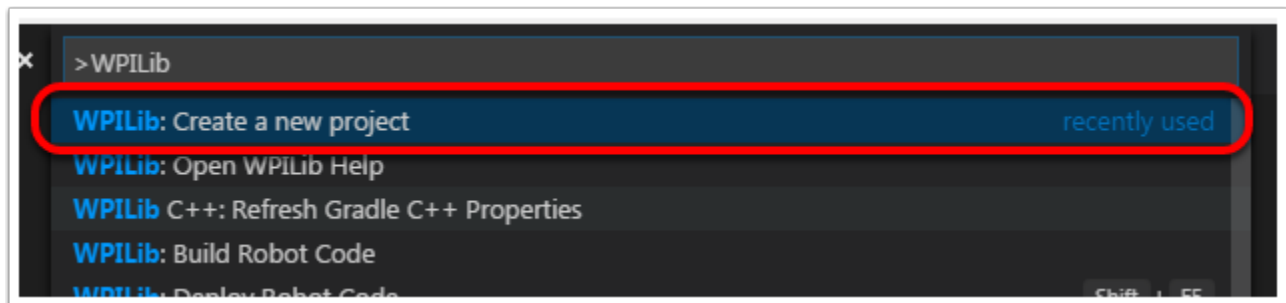
All WPILib commands start with "WPILib:", so in order to access the WPILib commands type "WPILib:" into the command palette search bar.

FRC Java Programming

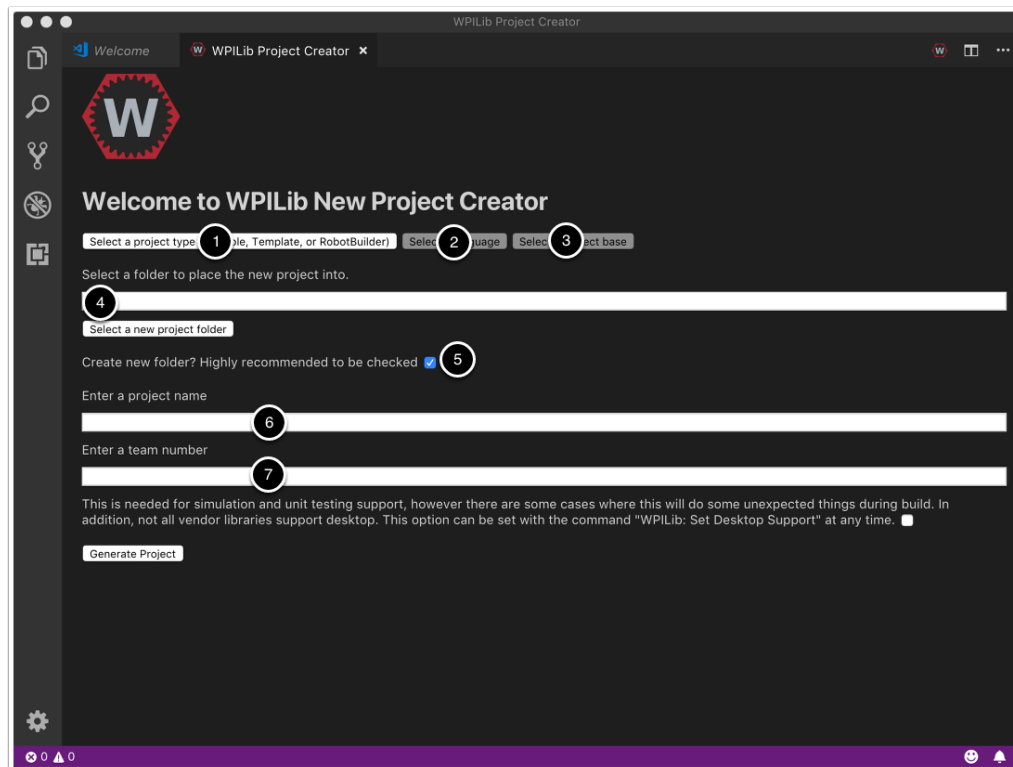


Creating A New WPILib Project

In order to create a new project select the "Create a new project" command. This will show a form with a number of fields where you enter the information required to create your new project.



New project creator window



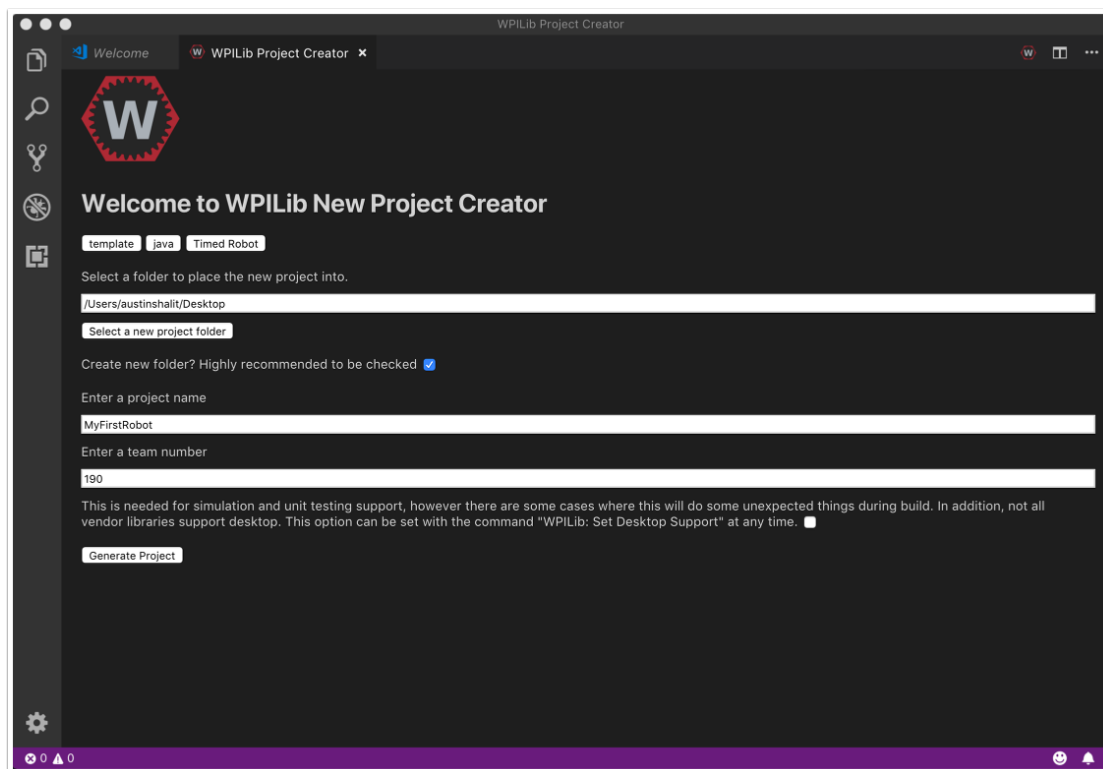
The steps to create the new project are outlined here:

1. Select the kind of project you want to create. It can be an example project or one of the template projects provided by WPILib.
2. Select the language that you are using for your project.
3. In the case of a template - select the template type (Timed robot, Iterative robot, Command robot, etc.)
4. Select the folder to place the project.
5. If the "Create new folder" checkbox is checked, a new folder named with the project name is created in the supplied folder. If the checkbox is NOT checked, then the folder supplied is assumed to be empty (will give an error if not) and the project files will be placed into that directory.
6. The project name is used in the project and also to optionally create the folder to place it if the checkbox from the previous step is checked.
7. The team number for the project. This will be used for package names and to locate your robot when deploying code.

FRC Java Programming

And last, click "Generate Project" and VS Code will create the project in the location specified.

i Note: If there are any errors generating the project (such as trying to use a non-empty folder with the checkbox unchecked), they will pop up in the bottom right corner of the screen.

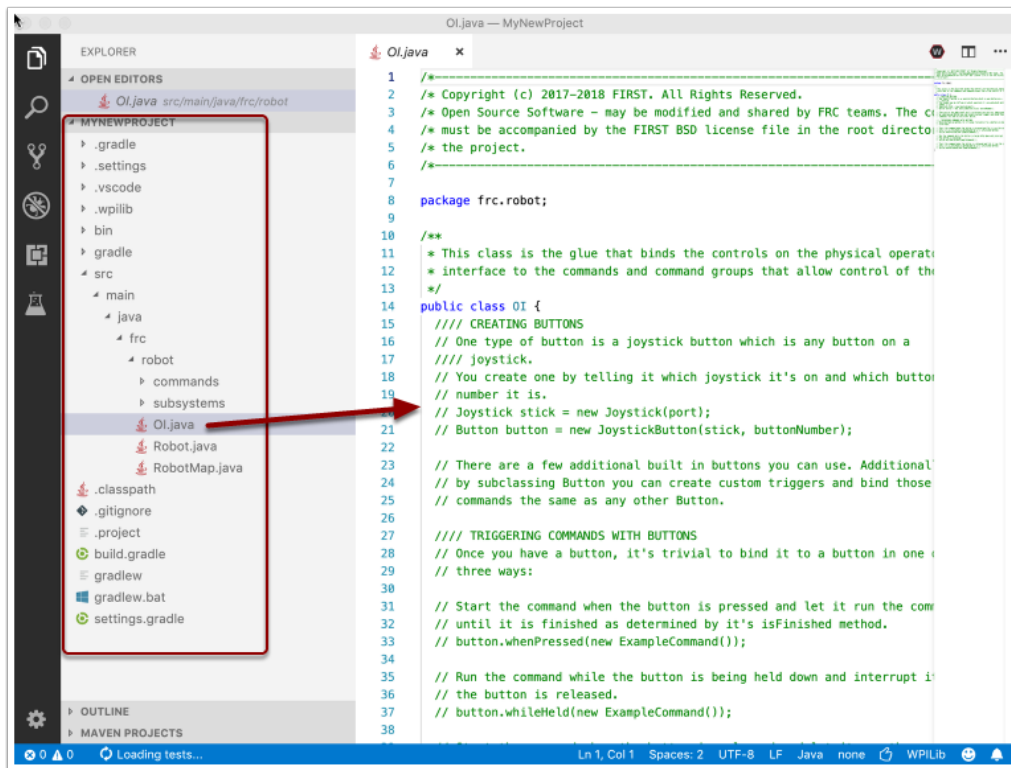


Opening The New Project

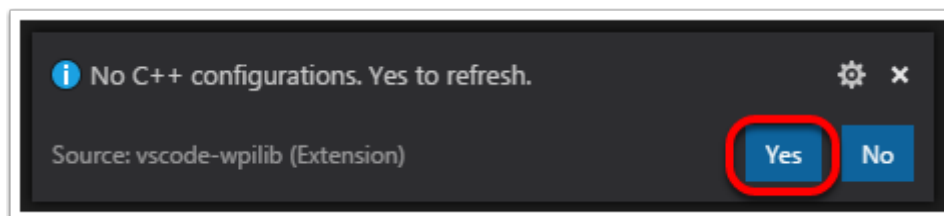
After successfully creating your project, Visual Studio Code will give you the option of opening the project as shown below. You can choose to do that now or later by typing Ctrl-O (Command+O on mac) and select the folder where you saved your project.

Once opened you will see the project hierarchy on the left. Double clicking on the file will open that file in the editor.

FRC Java Programming



C++ Configurations (C++ Only)



For C++ projects, there is one more step to set up IntelliSense. Whenever you open a project, you should get a pop-up in the bottom right corner asking to refresh C++ configurations, click Yes to setup IntelliSense.

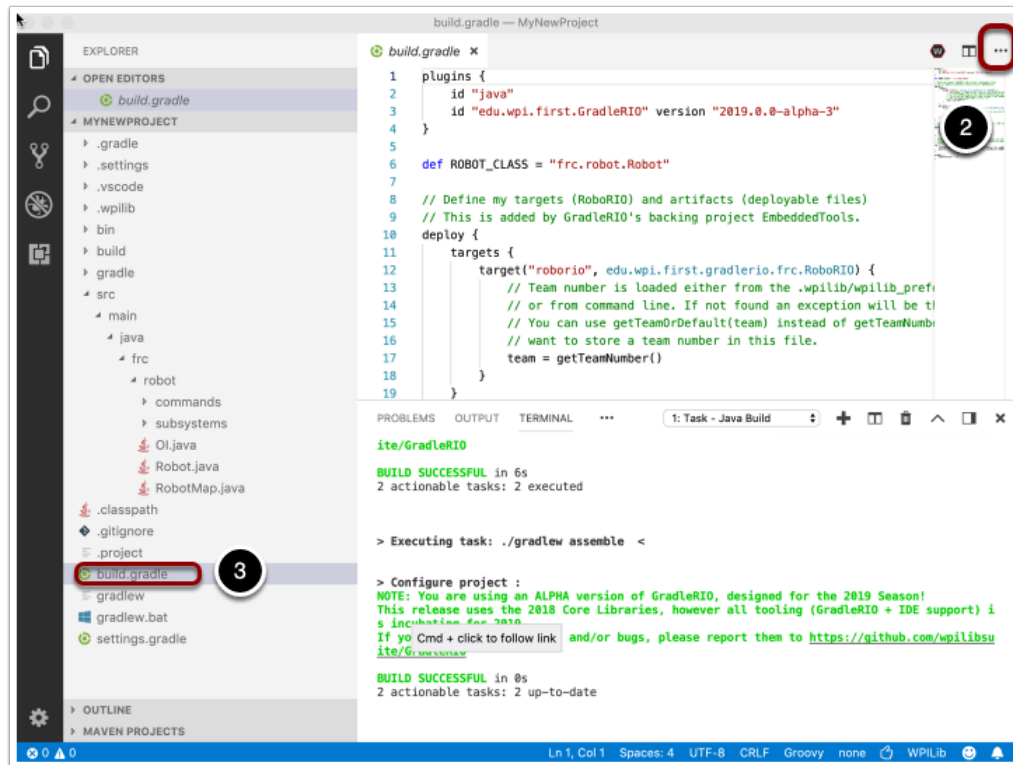
Building and Deploying Robot Code

To build the robot project, do one of:

1. Open the Command Palette and select "Build Robot Code"

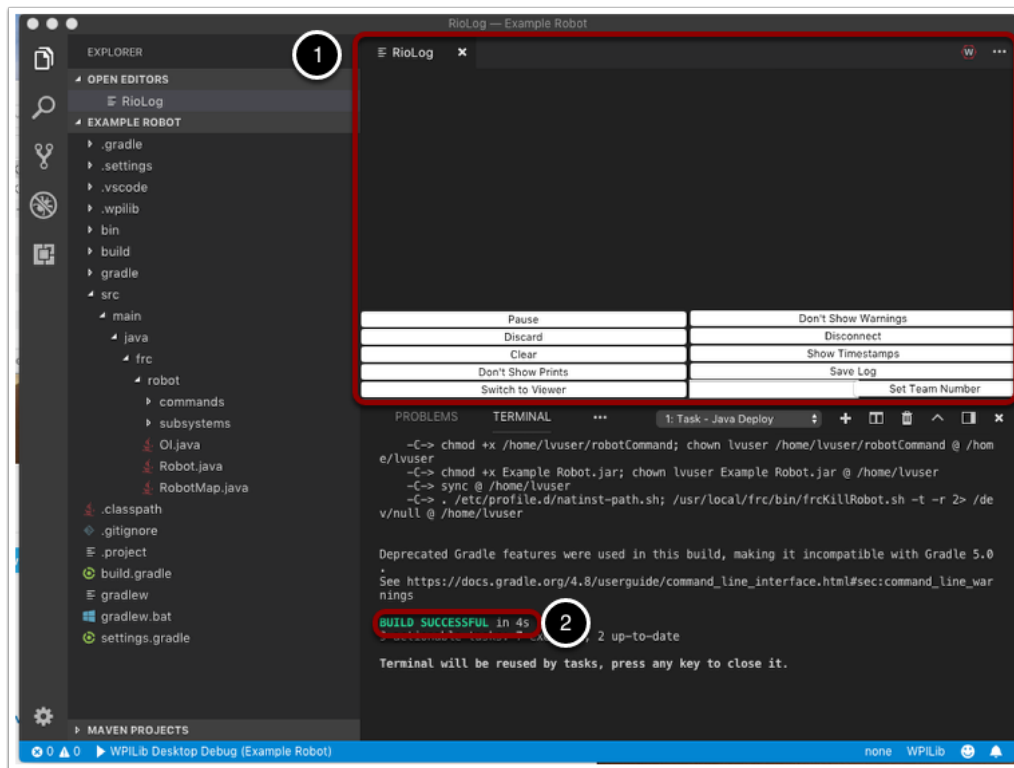
FRC Java Programming

2. Open the shortcut menu indicated by the ellipses in the top right corner of the VS Code window and select "Build Robot Code"
3. Right-click on the build.gradle file in the project hierarchy and select "Build Robot Code"



Deploy robot code by selecting "Deploy Robot Code" from any of the three locations from the previous instructions. That will build (if necessary) and deploy the robot program to the roboRIO. If successful, you see a "Build Successful" message (2) and the RioLog will open with the console output from the robot program as it runs.

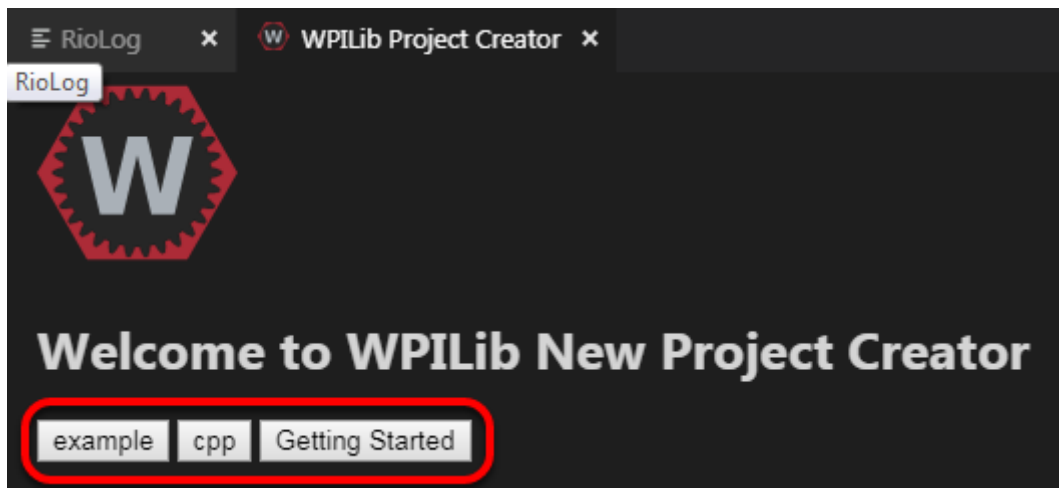
FRC Java Programming



Creating your Benchtop Test Program

This article describes the Benchtop test example program

Creating a project



Create a new **Getting Started** Example project. For more info about creating projects, see [Creating a robot program](#).

Imports/Includes

C++

```
#include <frc/Joystick.h>
#include <frc/PWMVictorSPX.h>
#include <frc/TimedRobot.h>
#include <frc/Timer.h>
#include <frc/drive/DifferentialDrive.h>
#include <frc/livewindow/LiveWindow.h>
```

Java

FRC Java Programming

```
import edu.wpi.first.wpilibj.Joystick;
import edu.wpi.first.wpilibj.PWMVictorSPX;
import edu.wpi.first.wpilibj.TimedRobot;
import edu.wpi.first.wpilibj.Timer;
import edu.wpi.first.wpilibj.drive.DifferentialDrive;
```

Our code needs to reference the components of WPILib that are used. In C++ this is accomplished using "#include" statements, in Java it is done with "import" statements. The program references classes for Joystick (for driving), PWMVictorSPX (for controlling motors), TimedRobot (the base class used for the example), Timer (used for autonomous), DifferentialDrive (for connecting the joystick control to the motors), and LiveWindow (C++ only).

Defining the variables for our sample robot

C++

```
class Robot : public frc::TimedRobot
{
public:
    Robot() {
        m_robotDrive.SetExpiration(0.1);
        m_timer.Start();
    }

private:
    // Robot drive system
    frc::PWMVictorSPX m_left{0};
    frc::PWMVictorSPX m_right{1};
    frc::DifferentialDrive m_robotDrive{m_left, m_right};
    frc::Joystick m_stick{0};
    frc::LiveWindow& m_lw = *frc::LiveWindow::GetInstance();
    frc::Timer m_timer;
```

Java

```
public class Robot extends TimedRobot {
    private final DifferentialDrive m_robotDrive = new DifferentialDrive(new
    PWMVictorSPX(0), new PWMVictorSPX(1));
```

FRC Java Programming

```
private final Joystick m_stick = new Joystick(0);  
private final Timer m_timer = new Timer();
```

The sample robot in our examples will have a joystick on USB port 0 for arcade drive and two motors on PWM ports 0 and 1. Here we create objects of type DifferentialDrive (m_robotDrive), Joystick (m_stick) and time (m_timer). This section of the code does three things:

1. Defines the variables as members of our Robot class.
2. Initializes the variables.

Note: The variable initializations for C++ are in the "private" section at the bottom of the program. This means they are private to the class (Robot). The C++ code also sets the Motor Safety expiration to 0.1 seconds (the drive will shut off if we don't give it a command every .1 seconds and starts the timer used for autonomous.

Robot Initialization

C++

```
void RobotInit() {}
```

Java

```
@Override public void robotInit() { }
```

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). This method is omitted from the code, meaning the default version will be run (if we wanted to run something here we could provide the code above to override the default).

Simple autonomous sample

C++

```
void AutonomousInit() override {  
    m_timer.Reset();  
    m_timer.Start();  
}
```

FRC Java Programming

```
}

void AutonomousPeriodic() override {
    // Drive for 2 seconds
    if (m_timer.Get() < 2.0) {
        // Drive forwards half speed
        m_robotDrive.ArcadeDrive(-0.5, 0.0);
    } else {
        // Stop robot
        m_robotDrive.ArcadeDrive(0.0, 0.0);
    }
}
```

Java

```
@Override
public void autonomousInit() {
    m_timer.reset();
    m_timer.start();
}

@Override
public void autonomousPeriodic() {
    // Drive for 2 seconds
    if (m_timer.get() < 2.0) {
        m_robotDrive.arcadeDrive(0.5, 0.0); // drive forwards half speed
    } else {
        m_robotDrive.stopMotor(); // stop robot
    }
}
```

The `AutonomousInit` method is run once each time the robot transitions to autonomous from another mode. In this program, we reset the timer and then start it in this method.

`AutonomousPeriodic` is run once every period while the robot is in autonomous mode. In the `TimedRobot` class the period is a fixed time, which defaults to 20ms. In this example, the periodic code checks if the timer is less than 2 seconds and if so, drives forward at half speed using the `ArcadeDrive` method of the `DifferentialDrive` class. The value is negative for forward motion because of the convention for joysticks where a negative Y-axis value corresponds to moving the stick away from you (forward). If more than 2 seconds has elapsed, the code stops the robot drive.

Joystick Control for teleoperation

C++

```
void TeleopInit() override {}  
void TeleopPeriodic() override {  
    // Drive with arcade style (use right stick)  
    m_robotDrive.ArcadeDrive(m_stick.GetY(), m_stick.GetX());  
}
```

Java

```
@Override  
public void teleopInit() {  
}  
  
@Override  
public void teleopPeriodic() {  
    m_robotDrive.arcadeDrive(m_stick.getY(), m_stick.getX());  
}
```

Like in Autonomous, the Teleop mode has a TeleopInit and TeleopPeriodic function. In this example we don't have anything to do in TeleopInit, it is provided for illustration purposes only. In Teleop Periodic, the code uses the ArcadeDrive method to map the Y-axis of the joystick to forward/back motion of the drive motors and the X-axis to turning motion.

Test Mode

C++

```
void TestPeriodic() override {}
```

Java

```
@Override public void testPeriodic() { }
```

Test Mode is used for testing robot functionality. Similar to TeleopInit, the TestPeriodic is provided here for example.

FRC Java Programming

Next Steps

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

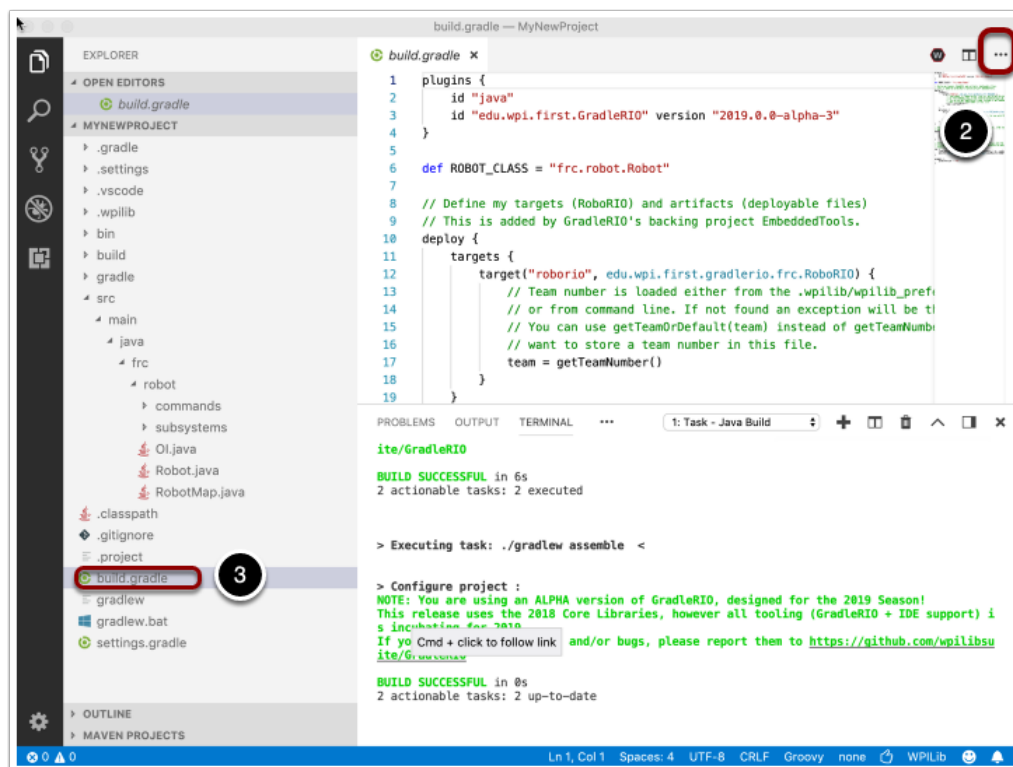
Building and deploying to a roboRIO

Building Robot Code

The first step to getting the program on the roboRIO is to build the code. This will compile and link the project files.

To build the robot project, do one of:

1. Open the Command Palette and select "Build Robot Code"
2. Open the shortcut menu indicated by the ellipses in the top right corner of the VS Code window and select "Build Robot Code"
3. Right-click on the build.gradle file in the project hierarchy and select "Build Robot Code"

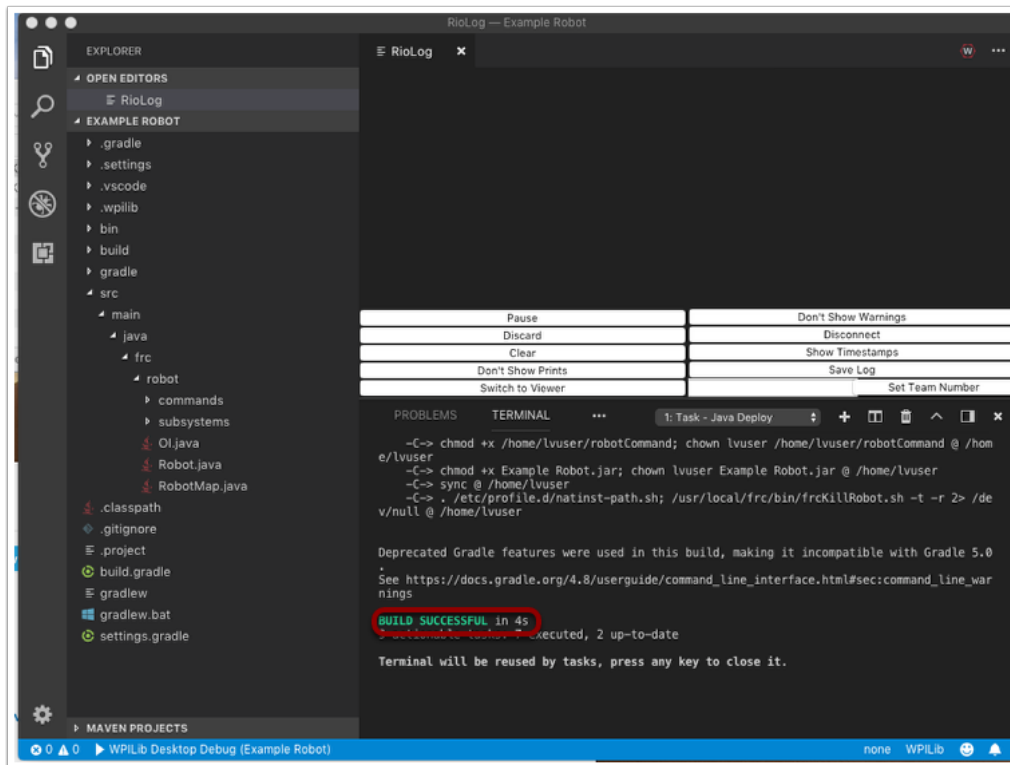


Deploying Robot Code

Deploy robot code by selecting "Deploy Robot Code" from any of the three locations from the previous instructions. That will build (if necessary) and deploy the robot program to the roboRIO.

FRC Java Programming

The deployment process will begin and start outputting status information to the console. If successful, you see a "Build Successful" message and the RioLog will open with the console output from the robot program as it runs.



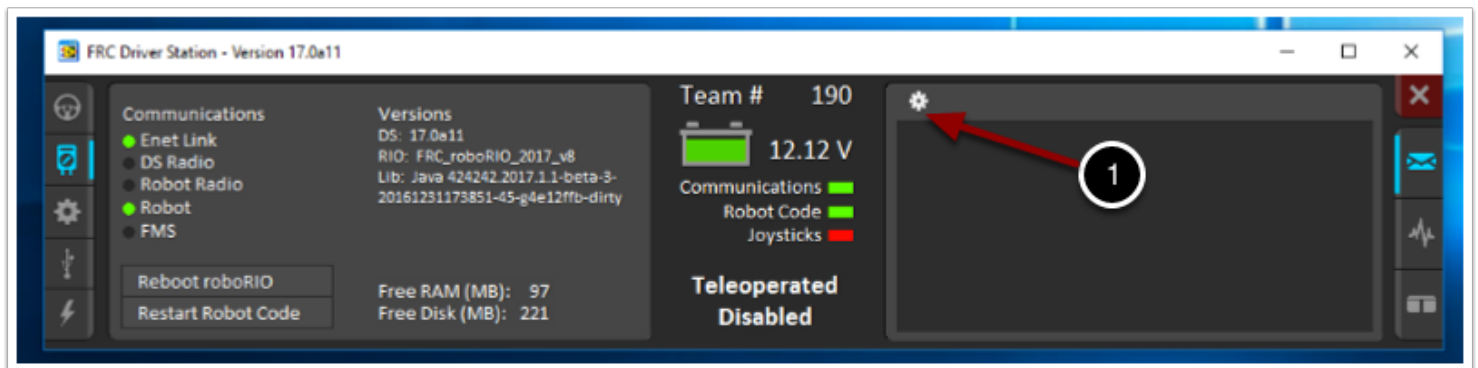
Viewing Console Output

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 VS Code.

Console Viewer

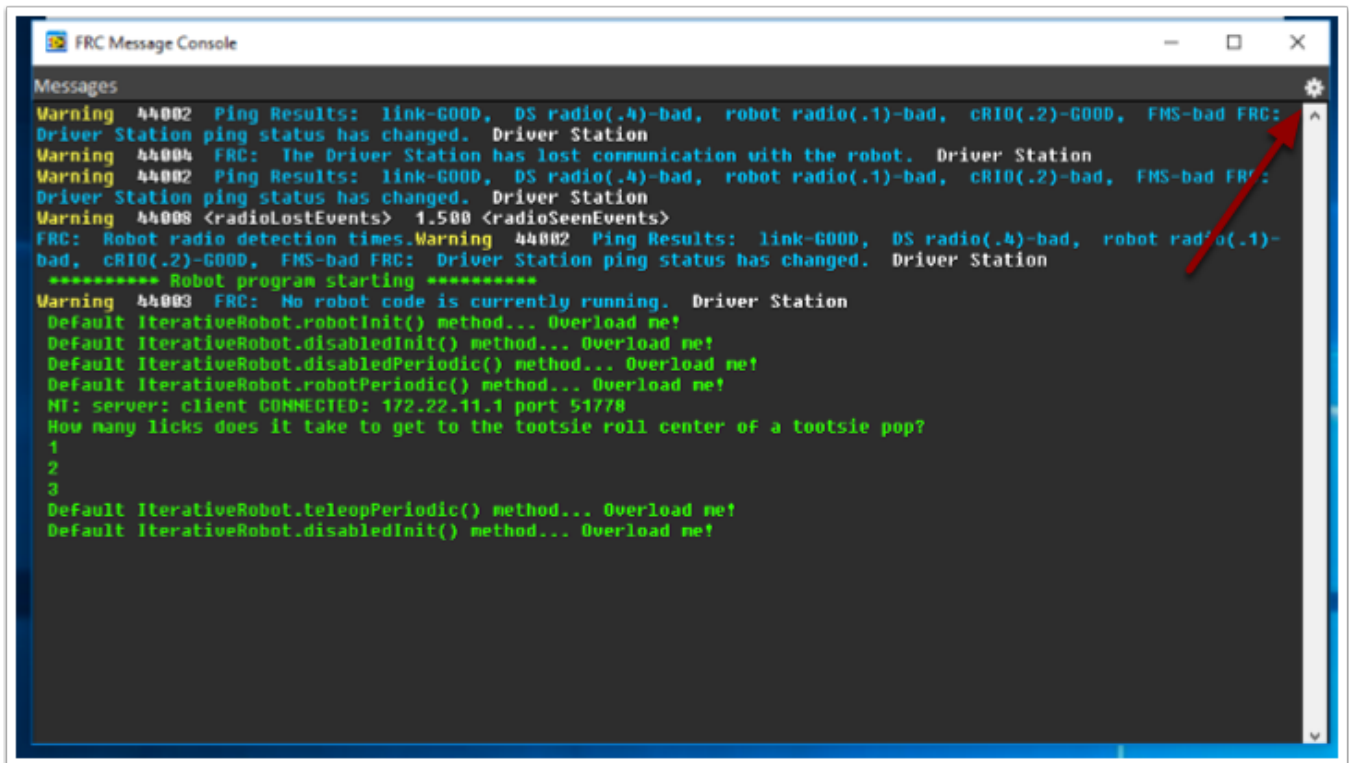
Opening the Console Viewer



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".

FRC Java Programming

Console Viewer Window



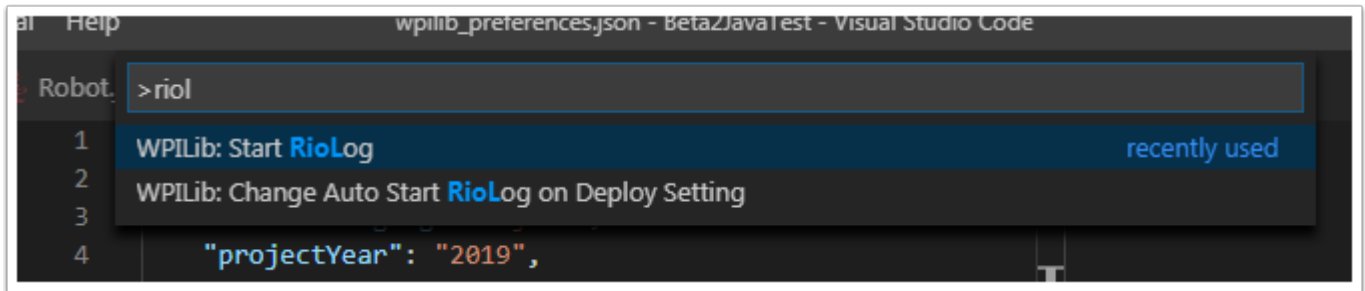
The Console Viewer window displays the output from our robot program in green. The gear in the top right can clear the window and set the level of messages displayed.

Riolog VS Code Plugin

The Riolog plugin is a VS Code view that can be used to view the NetConsole output in VS Code (credit for the original Eclipse version: Manuel Stoeckl, FRC1511).

FRC Java Programming

Opening the RioLog View



By default, the RioLog view will open automatically at the end of each roboRIO deploy. If you want to launch the RioLog view manually, press Ctrl+Shift+P to open the command palette and start typing "RioLog", then select the **WPIlib: Start RioLog** option.

RioLog Window



FRC Java Programming

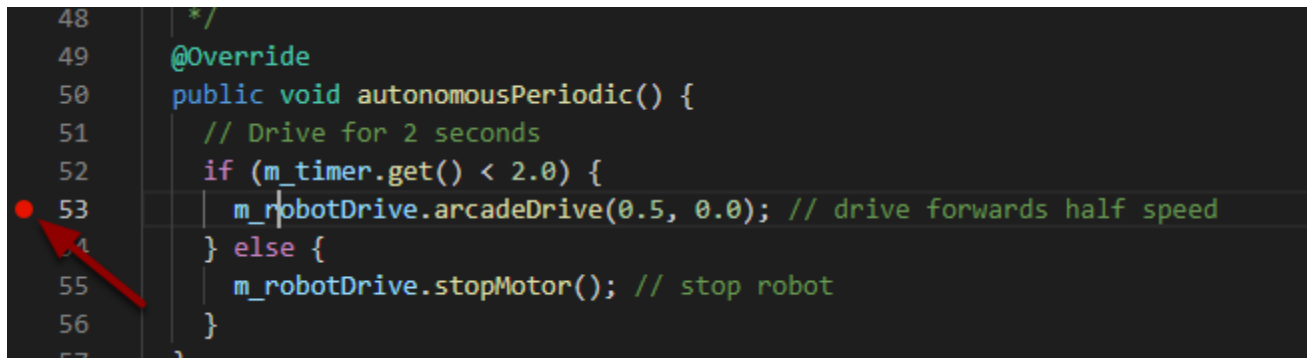
The RioLog view should appear in the top pane. Riolog contains a number of controls for manipulating the console.

- 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.
- 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.
- Clear - This will clear the current contents of the display.
- Don't Show/Show Prints - This shows or hides messages categorized as print statements
- Switch to Viewer - This switches to viewer for saved log files
- Don't Show/Show Warnings - This shows or hides messages categorized as warnings
- Disconnect/Reconnect - This disconnects or reconnects to the console stream
- Show/Don't Show Timestamps - Shows or hides timestamps on messages in the window
- Save Log - Copies the log contents into a file you can save and view or open later with the RioLog viewer (see Switch to Viewer above)
- Set Team Number - Sets the team number of the roboRIO to connect to the console stream on, set automatically if RioLog is launched by the deploy process

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 an FRC robot program.

Set a Breakpoint

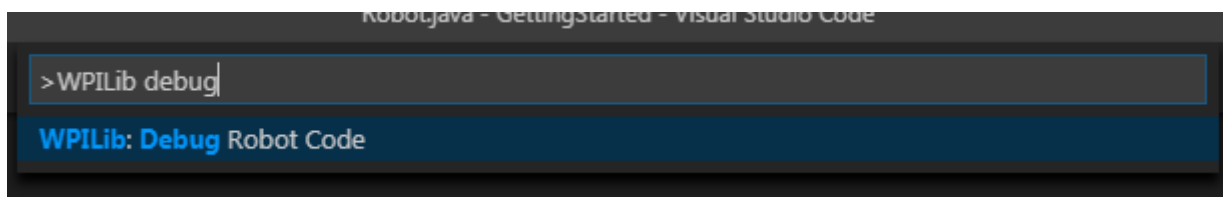


```
48  */
49  @Override
50  public void autonomousPeriodic() {
51      // Drive for 2 seconds
52      if (m_timer.get() < 2.0) {
53          m_robotDrive.arcadeDrive(0.5, 0.0); // drive forwards half speed
54      } else {
55          m_robotDrive.stopMotor(); // stop robot
56      }
57  }
```

A screenshot of a code editor window with a dark background. The code is in Java. A red circle, representing a breakpoint, is set on line 53. A red arrow points from the left margin to the red circle.

Double-click in the left margin of the source code window to set a breakpoint in your user program: A small red circle indicates the breakpoint has been set on the corresponding line.

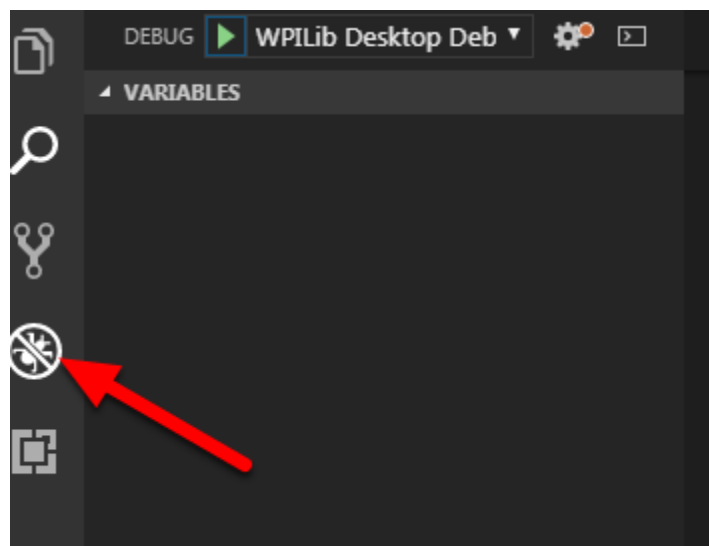
Start Debugging



Press **Ctrl+Shift+P** and type **WPIlib** or click on the **WPIlib Menu** item to open the Command palette with **WPIlib** pre-populated. Type **Debug** and select the **Debug Robot Code** menu item to start debugging. The code will download to the roboRIO and begin debugging.

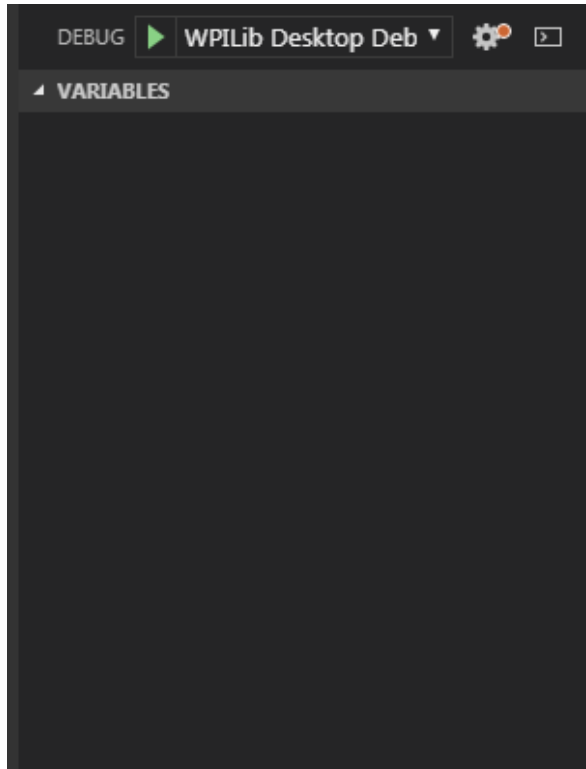
FRC Java Programming

The Debug tab



The Debug tab is accessed by clicking on the debug icon on the far left pane.

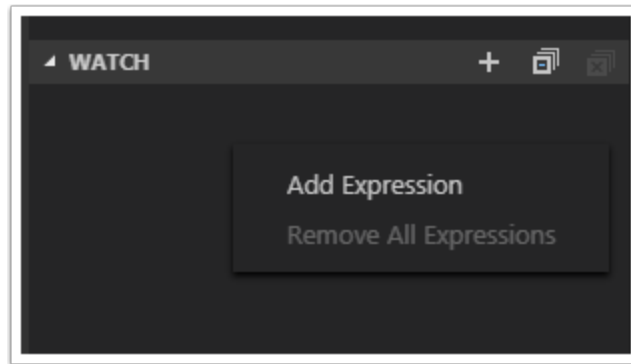
The Variables Pane



The Variables view shows the current values of variables. To see a variable that is not displayed, select the “Watch” pane 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.

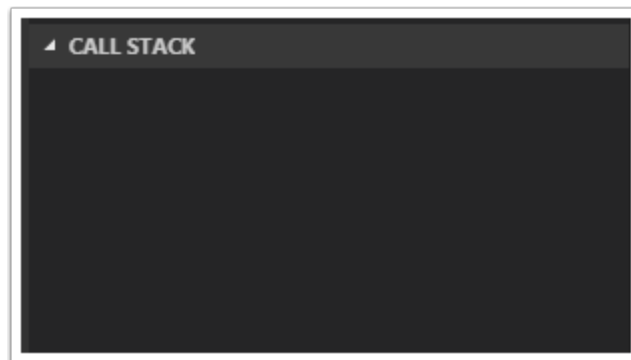
FRC Java Programming

Watch Pane



The Watch Pane can be used to monitor specific variables or expressions while debugging. To add an expression, right-click and select Add Expression.

Call Stack



The Call Stack pane is used to display the current Call Stack of the running program. This can be used to monitor the current call hierarchy of your program while debugging.

FRC Java Programming

Breakpoint Pane



The Breakpoint Pane displays all of the current breakpoints. To temporarily disable a breakpoint without permanently removing it, click the checkbox.

Your Breakpoint

```
50  public void autonomousPeriodic() {  
51      // Drive for 2 seconds  
52  if (m_timer.get() < 2.0) {  
53      m_robotDrive.arcadeDrive(0.5, 0.0); // drive forwards half speed  
54  } else {  
55      m_robotDrive.stopMotor(); // stop robot  
56  }
```

The program will start running and then pause at the breakpoint. From here you can use the panes on the left of the screen and the items in the Debug dropdown menu to monitor and control the execution of your program.

Debugging with Console

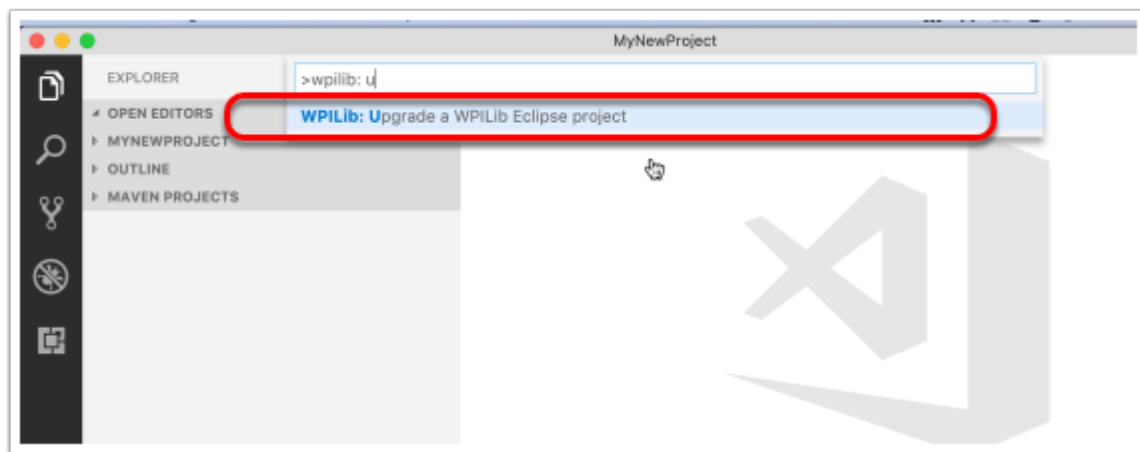
Another way to debug your program is to use `System.out.println` statements in your code and receive them using the RioLog in VSCode.

Importing an Eclipse project into VS Code

To make it easy for teams to use existing projects with the new IDE, we have implemented a wizard for importing Eclipse projects into VS Code. This will generate the necessary Gradle components and load the project into VS Code.

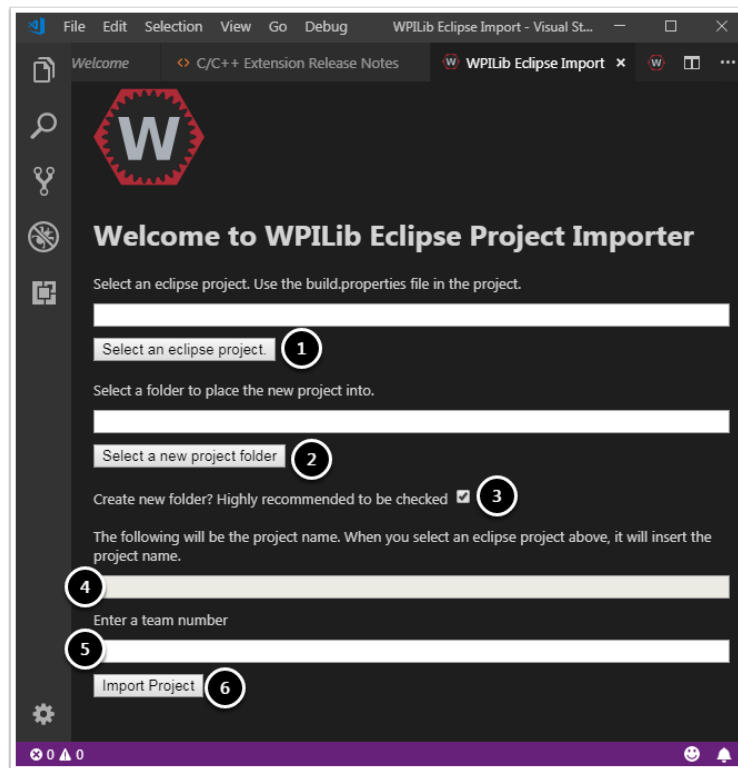
Launch the Import Wizard

Press Ctrl+Shift+P and type "WPILib" or click the WPILib icon to locate the WPILib commands. Select "Upgrade a WPILib Eclipse Project".



You'll be presented with the WPILib Eclipse Project Upgrade window. This is similar to the process of creating a new project and the window and the steps are shown below.

FRC Java Programming



Perform the following steps to fill in the Eclipse Project Upgrade window:

1. Select the eclipse project to convert. Select the build.properties file in the root directory of the eclipse project.
2. Fill in the new project folder by pressing the "Select a new project folder" button.
3. If the "Create new folder" checkbox is checked, then the project will be stored in a new folder under the one selected in 2. If it is not checked, then the project will be placed in the folder specified. It must be empty in that case.
4. Enter the name of the new project.
5. Enter the team number for the creation of the project and for the robot deployment.
6. And finally, click "Upgrade Project" to begin the upgrade.

The eclipse project will be upgraded and copied into the new project directory from step 3 above. You can then either open the new project immediately or open it later using the Ctrl-O (or Command-O for Mac) shortcut.

FRC Java Programming

Welcome to WPILib Eclipse Project Upgrade

Select an eclipse project. Use the build.properties file in the project.

/Users/bradmiller/eclipse-workspace/SampleJavaProject/build.properties

Select an eclipse project.

Select a folder to place the new project into.

/Users/bradmiller/VSCoDe projects/SampleRobotProject

Select a new project folder

Create new folder? Highly recommended to be checked ☒

The following will be the project name. When you select an eclipse project above, it will insert the project name.

SampleJavaProject

Enter a team number

190

Upgrade Project

Would you like to open the folder?

Source: WPILi... Yes (Current Window) Yes (New Window) No

C++ Configurations (C++ Only)

For C++ projects, there is one more step to set up IntelliSense. Whenever you open a project, you should get a pop-up in the bottom right corner asking to refresh C++ configurations, click Yes to setup IntelliSense.

FRC Java References

FRC Java WPILib API Documentation

Online Documentation

<http://first.wpi.edu/FRC/roborio/release/docs/java/>

Local Javadoc

Local Javadoc

The Eclipse plugins also install a local copy of the Javadocs for the library. You can view the Javadoc comments for a particular class or method by hovering over the method in your code. You can also view the complete Javadoc in Eclipse using the built-in web browser. To do this, click Window -> Show View -> Other -> Internal Web Browser then enter the URL USERHOME/wpilib/java/current/javadoc/index.html (USERHOME on Windows is typically C:\Users\USERNAME)

C++\Java Plugin Changelog

Changelog for the C++\Java Eclipse Plugins

2018.4.1 (Recommended)

Network Tables (ntcore)

- Clients (e.g. dashboards) were incorrectly handling synchronization of keys that were modified on both the client and server during reconnects, resulting in ignoring later value changes. It's been made more attentive.

Camera Server (cscore)

- HTTP cameras were lovingly holding on to their existing connections even when the URL was changed (so changing camera settings via URL wasn't forcing a reconnect). They now callously dump the existing connection instead.
- The MJPEG server ignored the FPS setting. It's been made more attentive and bandwidth-conscious.
- Cameras now provide telemetry on the actual FPS and bandwidth. Coming soon to a dashboard near you!

SmartDashboard

- Save files could be corrupted due to a null pointer at random times. This was neither desired nor user-friendly, so it's been fixed.

FRC Java Programming

WPILib (C++\Java)

- The HAL Notifier could take a long time to delete the last notifier. This was unusual to run into (as there's almost always at least one notifier), but it's been fixed.
- PWM did not have a default PWM configuration, so `setSpeed()` and `getSpeed()` throw exceptions until it was set. We're reasonable people, so reasonable defaults have been added.
- Documentation for `RobotDriveBase::SetDeadband()` was confusing. It now mentions that the deadband is applied to the drive inputs.
- `CameraServer` has been made more robust against cameras throwing errors when trying to publish information.
- Calling `setName()` on `SendableChooser` made it stop working. Fixed.
- PWM was claiming it was a Speed Controller to `LiveWindow`. While true on some levels, it can be used in other ways, so now the raw value is published instead of the speed (and it rightfully calls itself a PWM).
- `SerialPort`: In some cases (e.g. chained USB hubs), the serial port constants might be insufficient or just not work. A named port option is now available, so it's now possible to build the multi-line BBS robot you always dreamed of. Please note this new API is already deprecated as we're planning to change it for 2019 (it's a temporary solution for teams that need this feature).
- **Java only:** `ADXRS450_Gyro` was not checking for null in the `reset()` function (unlike all other functions). Consistency is a good thing, as is not throwing a `NullPointerException`, so this has been fixed.

Eclipse Plugins

- C++ Command-based examples/templates left out some required files. While we hope you've written your robot program by now, if you're just starting, this should make things easier.

2018.3.3 (Optional)

C++

- C++: `SendableChooser::GetSelected()` was keeping a temporary to a pointer, which is not a wise thing to do in C++ and could cause it to not return the selected value. It's now been made wiser in the ways of C++ so you can trust what it tells you.

FRC Java Programming

2018.3.2 (Optional)

Java

- Added tristate DIO support to HAL, for those times when you really don't want the output to be 0 *or* 1.

Shuffleboard

- If no internet access was available at startup, a dialog regarding not being able to get updates would be displayed that was non-obvious to escape. This check is now significantly less intrusive and no longer requires using a rock hammer to tunnel through the wall at Shawshank (or the use of the Esc key).

2018.3.1 (Optional)

CameraServer (cscore)

The below items mainly affected Shuffleboard viewing of Limelight camera streams)

- HttpCamera was non-compliant to the HTTP spec because it did not accept lowercase content-length and content-type. We asked Q&A about this but didn't get the answer we hoped for, so we had to change our design.
- HttpCamera was dropping every other frame if the camera didn't send us a Content-Length header with each frame. It's considered polite for them to send it, but it was rude for us to reject frames without it.

C++\Java

- Java: Encoder.getDistancePerPulse() was truncating to an integer. This was due to a bad cast, but we prefer to blame the director (or the script).
- Java: TimedRobot was hanging if an exception was thrown by your code. While we don't recommend your robot code crash, we thought it was better to restart it if it does.

FRC Java Programming

- C++: Joystick GetTwist() and GetThrottle() did not obey SetChannel. They've been made more obedient.
- TimedRobot now provides a getPeriod() function. We had been implementing this with write-only memory, but decided to upgrade.
- DifferentialDrive.curvatureDrive() now normalizes the output to -1 to +1 to avoid clipping and maintain the ratio between wheel speeds. While your mentors may exhort you to give 120% effort, your motors can't do that.
- Low level CAN operations for the PCM and PDP have been made thread-safe.
- Similar to the DifferentialDrive changes in 2018.2.2, the right side motors of MecanumDrive are now inverted appropriately on the dashboard, so you can finally upgrade the code on your 2008 robot even if you used mecanum (or mecanum) that year.

Eclipse Plugins

- Improved the C++ command templates; we felt this was preferred to disimproving them.

Shuffleboard

- Shuffleboard has gained a new superpower: it can now update itself independently of Eclipse updates, just like Galactus.
- Previously didn't remember what plugins you had loaded the last time you ran it. It's been made less forgetful.
- Graph widgets were throwing errors when their backing data was deleted; they've been made more tolerant.
- Instead of silently failing to start up when Java 9 is installed, it now noisily fails to start up.
- Now that FMS info is published to NT, Shuffleboard uses that knowledge for good by providing a widget to display it.
- Dark themed scroll panes weren't very pretty; they've been given a makeover.
- Save files had some issues. They've been given extensive counseling and now are much better behaved and remember more things.
- The NetworkTable tree is now expanded by default instead of the CameraServer tree. While we know you love cameras, we know you love fewer clicks too.
- Fixed various odd display bugs caused by threading issues.

FRC Java Programming

2018.2.2 (Optional)

C++\Java

- Fixed: Java FMS data could be null before the DS connected. An empty string felt more logical.
- Fixed: Java PIDController.setContinuous(false) could throw an error when it's not supposed to. It now only does if there's a good reason.

Shuffleboard

Fixed: Gyro was not correctly displaying negative degrees (or negative radians). It's been fixed so that robots are no longer limited to only making right hand turns (after all, it's not 2008 anymore).

2018.2.1 (Optional)

C++\Java

- Fixed: Java getBatteryVoltage() was lacking static. Rubbed shoes on carpet to triboelectric charge it.
- Fixed: PIDController continuous operation was confused if no input range was provided. We applied a non-cursed unicorn horn, so now it just treats the input as non-continuous instead in that case.
- Fixed: SpeedControllerGroup was not inverting individual motor directions appropriately when get()ing or pidWrite()ing. Now they go the right way (or left way, if you've inverted them).
- Fixed: ConditionalCommands cancellation of inner commands did not always take place the way it should have. Now it does.
- Fixed: Driver station inputs were being delayed by 1 packet. One packet may not sound like a lot, but your robot is now ~40ms more responsive to your commands! (whether or not it obeys your commands is still up to you)
- Added: FMS information (game specific data, alliance station location and color, match number, and other info) is now automatically published to NetworkTables. We encourage you to use this knowledge, but only for good.

FRC Java Programming

Eclipse Plugins

- Fixed: When linking C++ programs, 3rd party libraries were being linked in random order (which sometimes caused spurious errors). They are now sorted and duplicated, but most importantly, are no longer random.

Added: When building the Java .jar, we now include everything in src/, not just .java files. For this reason, we now recommend you not store your music or movie collection in your robot code src/ folder.

Fixed: Deploy would fail if there were subdirectories in the user libraries folder. We now only copy from the top-level folder, which is as it should be.

Added: Time-travelers in a DeLorean added support for the v17 image. No, the v17 image doesn't exist yet; we're just thinking 4th dimensionally!

Shuffleboard

- Fixed: SingleKeyNetworkTableSource (buttons, sliders, etc.) updates were fragile and broke too easily, they have been toughened up.
- Fixed: FXML Widgets from external plugins wouldn't load properly. Now they do.
- Fixed: Loading saves with a source missing made it forget about it altogether. It has now been made less forgetful.
- Added: Widget for single axis accelerometer, in case you want to measure gravity, or maybe something more useful.
- Fixed: Quadrature Encoder is a different type than Encoder so it didn't have a default widget. Now it does.
- Fixed: Camera streams kept streaming even after they were closed, hogging bandwidth. They are now more bit-conscious and only stream when open.
- Added: Allow themes to be defined in Shuffleboard directory. Now you can make that rainbow dashboard you wanted without even writing any code!
- Added: Connection Indicators, so you know whether to blame the code or the network when your new value doesn't appear.

FRC Java Programming

2018.1.1

Kickoff Release

FRC Java Basics

Java conventions for objects, methods and variables

Creating objects that are connected to the roboRIO in Java

```
Gyro headingGyro = new AnalogGyro(1);  
double heading = headingGyro.getAngle();
```

Generally all the objects in WPILib that connect to one of the roboRIO breakout boards 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 Java.

1. Creates an AnalogGyro object connected to analog channel 1 and stores its address in "headingGyro".
2. Gets the current heading from the AnalogGyro in degrees and stores it in the variable "heading".

Creating operator interface objects in Java

Creating operator interface objects in Java

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.

1. Creates a Joystick object connected to USB port 1 on the DS (listed first in the Setup tab of the DS).
2. Gets the current X axis value of the joystick and stores it in the variable "speed".

Class, method and variable naming

Class, method and variable naming

MXP IO Numbering

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.

Multithreading in Java

Take a look at [this article](#) by Oracle for more information about concurrency and threads. Below you will find information important for robot programs written in WPILib that will cause unexplained errors.

Threads

The code below will never be able to exit! It will only stop when the entire JVM stops. There is no way in Java to stop a thread unless the thread exits by itself.

Bad Example

```
public class Robot extends IterativeRobot {
    public void robotInit() {
        Thread t = new Thread(() -> {
            while (true) {
                // We are stuck here
            }
        });
        t.start();
    }
}
```

We can solve this problem by setting a flag. In Java every thread has a flag designed for this. We just need to modify our code to check that flag. Take a look at this example:

```
public class Robot extends IterativeRobot {
    public void robotInit() {
        Thread t = new Thread(() -> {
            while (!Thread.interrupted()) {
                // Not stuck anymore!
            }
        });
        t.start();
    }
}
```

FRC Java Programming

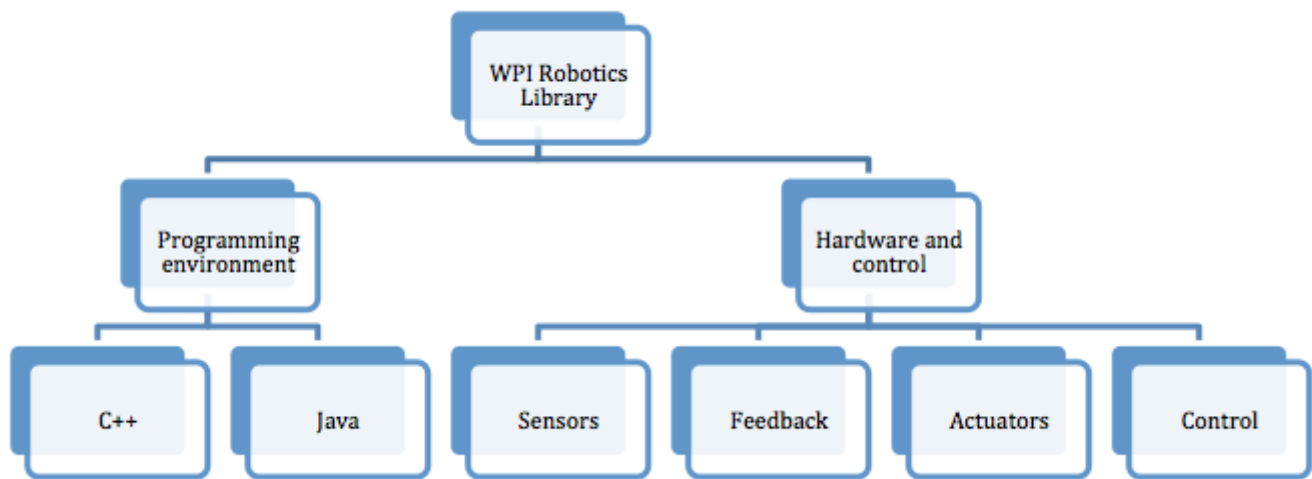
Everytime we run the loop, we check the interrupted flag to see if we should continue to execute.

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

FRC Java Programming

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

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>

FRC Java Programming

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);
}
```


FRC Java Programming

```
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

There are a number of base classes (starting points) for your robot program. Each base class sets the *style* and structure of your program. Be sure to read through this section before starting a robot project.

Base class	Application
SampleRobot	The SampleRobot base class is exactly what it sounds like, good for writing small sample programs , particularly to try out ideas. While it can be used for constructing a competition program it is not recommended because it is very hard to extend as additional capabilities are added. Instead choose any of the other templates described below.
IterativeRobot	The IterativeRobot base class has methods that are periodically called each time new data arrives from the Driver Station . The idea is that for each <i>mode</i> that the robot is operating in (autonomous, teleop, or test) the appropriate periodic method is called where the program does a small amount of work. It is important not to have any long running code in the periodic methods such as loops or delays . Doing so could result in missing driver station updates that can negatively impact robot performance . Each period is approximately 20 milliseconds but can vary depending on CPU load on the roboRIO, the driver station laptop, or network traffic. If you require precise timing, for example to implement robot control algorithms it is not recommended and you should instead use TimedRobot (below) which has precise timing between periods.
TimedRobot	TimedRobot is the same as IterativeRobot except that it uses a timer (Notifier) to guarantee that the periodic methods are called at a predictable time interval . When getting driver station data such as joystick values the most recent value will be provided since the time interval may not line up with the 20 millisecond delivery of data. This is the recommended base class for most robot programs . Just as with IterativeRobot, it is very important to not have long running code or loops in the periodic methods or the timing may slip .
Command based robot	While based on the TimedRobot base class, the command based robot programming style is recommended for most teams . It makes it easy to break up the program into Commands which each implement some robot behavior such as raising an arm to some position, driving for some distance, etc. It also makes the program easily extensible and testable. The RobotBuilder utility (included with the eclipse plugins) provides an easy way of organizing the

FRC Java Programming

Base class	Application
	program. The dashboards (SmartDashboard and Shuffleboard) allow you to easily debug and test command based programs.

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() {

    }

}
```

FRC Java Programming

```
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.

TimedRobot

TimedRobot base class is the same as as IterativeRobot (above) except that it calls the periodic functions using the specified time interval. The default time interval is 0.02 seconds (20 milliseconds) for each call to the appropriate periodic function. The default time interval can be overridden by calling the the setPeriod (java) or SetPeriod (C++) with the time in seconds as a double value. Internally an Notifier is used to set the interval.

SampleRobot

C++

```
RobotTemplate::RobotTemplate()  
{  
}
```

FRC Java Programming

```
//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);
        }
    }
}
```

FRC Java Programming

```
}
```

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 complexity in your programs.** 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 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

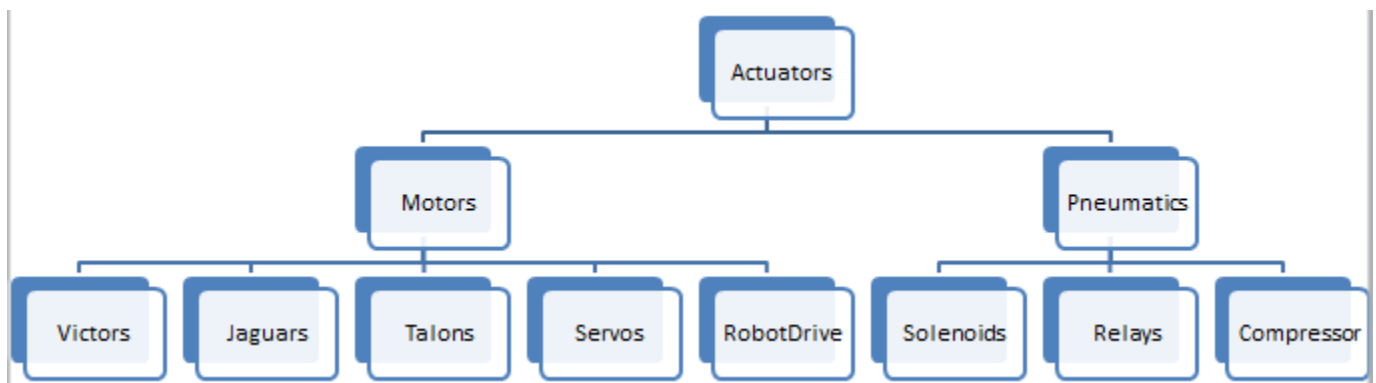
While not strictly a base class, the Command-based robot model is a method for creating larger programs, more easily, that are easier to extend. There is built in support with a number of classes to make it easy to design your robot, build subsystems, and control interactions between the robot and the 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 PWM speed controller objects

WPILib has extensive support for motor control. There are a number of classes that represent different types of speed controllers and servos. There are currently two classes of speed controllers, PWM based motor controllers and CAN based motor controllers. WPILib also contains composite classes (like DifferentialDrive) which allow you to control multiple motors with a single object. This article will cover the details of PWM motor controllers; CAN controllers and composite classes will be covered in separate articles.

PWM Controllers, brief theory of operation

The acronym PWM stands for Pulse Width Modulation. For 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. Many of the controllers use the typical hobby RC controller timing of 1ms to 2ms.

Raw vs Scaled output values

In general, all of the motor controller classes in WPILib take a scaled -1.0 to 1.0 value as the output to an actuator. The PWM module in the FPGA on the roboRIO 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

FRC Java Programming

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](#)

Constructing a Speed Controller object

C++

```
frc::Jaguar exampleJaguar{0};
frc::Talon exampleTalon{1};
frc::PWMTalonSRX examplePwmTalonSRX{2};
frc::Spark exampleSpark{3};
frc::Victor exampleVictor{11};
frc::VictorSP exampleVictorSP{12};
```

Java

```
Jaguar exampleJaguar = new Jaguar(0);
Talon exampleTalon = new Talon(1);
PWMTalonSRX examplePwmTalonSRX = new PWMTalonSRX(2);
Spark exampleSpark = new Spark(3);
Victor exampleVictor = new Victor(11);
VictorSP exampleVictorSP = new VictorSP(12);
```

FRC Java Programming

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

Setting parameters

C++

```
frc::Spark exampleSpeedController{0};  
exampleSpeedController.EnableDeadbandElimination(true);
```

Java

```
Spark exampleSpeedController = new Spark(0);  
exampleSpeedController.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 (Spark) 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++

```
exampleSpeedController.Set(0.7);
```

Java

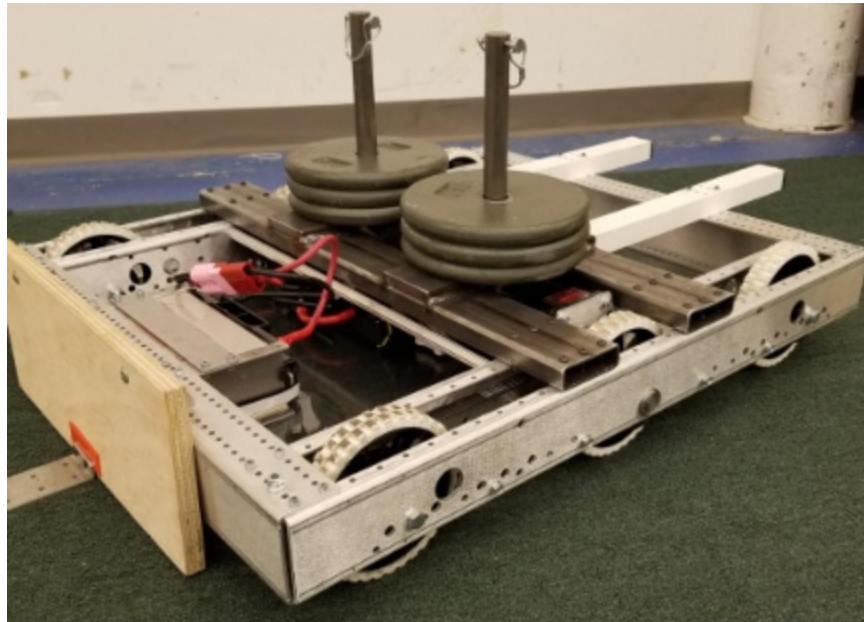
```
exampleSpeedController.set(0.7);
```

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

WPILib Drive classes: Drivetrain types

The WPILib Drive classes contain separate classes for each type of drivetrain. There are currently three types of drivetrains supported by WPILib classes. This article describes the three types.

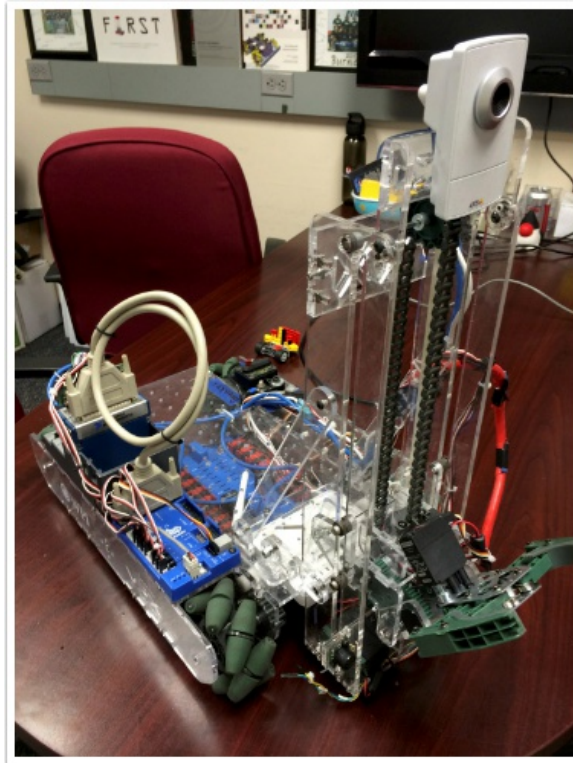
Differential Drive



These drive bases typically have two or more in-line traction or omni wheels per side (e.g., 6WD or 8WD) and may also be known as "skid-steer", "tank drive", or "West Coast Drive". The Kit of Parts drivetrain is an example of a differential drive. These drivetrains are capable of driving forward/backward and can turn by driving the two sides in opposite directions causing the wheels to skid sideways. These drivetrains are not capable of sideways translational movement.

For information on using the DifferentialDrive class, see [Driving a robot using Differential Drive](#).

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. The wheels (shown on 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.

When viewed from the top, the rollers on a mecanum drivetrain should form an 'X' pattern. This results in the force vectors (when driving the wheel forward) on the front two wheels pointing forward and inward and the rear two wheels pointing forward and outward. By spinning the wheels in different directions, various components of the force vectors cancel out, resulting in the desired robot movement. A quick chart of different movements has been provided below, drawing out the force vectors for each of these motions may help in understanding how these drivetrains work. By varying the speeds of the wheels in addition to the direction, movements can be combined resulting in translation in any direction and rotation, simultaneously.

FRC Java Programming

For information on using the MecanumDrive class, see [Driving a robot using Mecanum drive](#)

Direction of Movement	Front Left	Front Right	Rear Left	Rear Right
Forward	Forward	Forward	Forward	Forward
Reverse	Reverse	Reverse	Reverse	Reverse
Right Strafe	Forward	Reverse	Reverse	Forward
Left Strafe	Reverse	Forward	Forward	Reverse
Clockwise Turn	Forward	Reverse	Forward	Reverse
Counter-Clockwise Turn	Reverse	Forward	Reverse	Forward

Killough Drive

A Killough Drive (also known as a Kiwi Drive) is a holonomic drivetrain utilizing three omniwheels angled at 120 degrees from each other. Similar to the mecanum drive, wheels are run at different speeds in order to accomplish the desired overall motion. The control methods provided are the same as those for the Mecanum drive so for details on using the class, see the Javadoc/Doxygen and the [Driving a robot using Mecanum drive](#) article.

WPILib Drive classes: Conventions and Defaults

This article describes conventions and defaults used by the WPILib Drive classes (DifferentialDrive, MecanumDrive, and KilloughDrive). For further details on using these classes, see the subsequent articles.

Motor Inversion

By default, the class inverts the motor outputs for the right side of the drivetrain. Generally this will mean that no inversion needs to be done on the individual SpeedController objects. To disable this behavior, use the `setRightSideInverted()` method.

Squaring Inputs & Input Deadband

When driving robots, it is often desirable to manipulate the joystick inputs such that the robot has finer control at low speeds while still using the full output range. One way to accomplish this is by squaring the joystick input, then reapplying the sign. By default the Differential Drive class will square the inputs. If this is not desired (e.g. if passing values in from a PIDController), use one of the drive methods with the `squaredInputs` parameter and set it to false.

By default, the Differential Drive class applies an input deadband of .02. This means that input values with a magnitude below .02 (after any squaring as described above) will be set to 0. In most cases these small inputs result from imperfect joystick centering and are not sufficient to cause drivetrain movement, the deadband helps reduce necessary motor heating that may result from applying these small values to the drivetrain. To change the deadband, use the `setDeadband()` method.

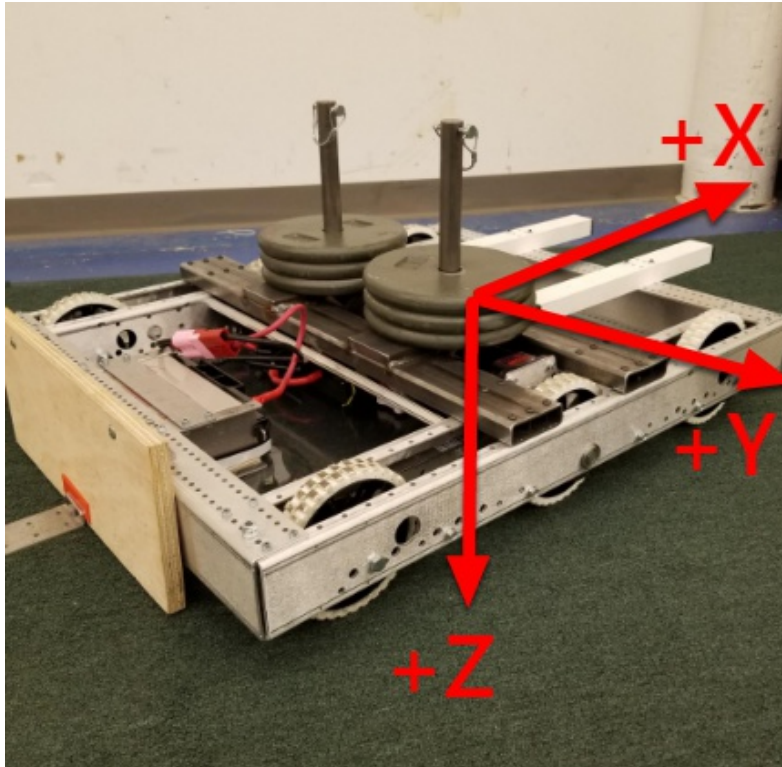
Motor Safety

By default all RobotDrive objects enable Motor Safety. This is a watchdog that disables the output if no update method is called within 100ms. To feed the watchdog without changing the input values, call `feedWatchdog()`. To change the watchdog timeout, call `setExpiration()`. To disable the

FRC Java Programming

watchdog, call `setSafetyEnabled(false)`. To learn more about Motor Safety, see the article: [Using the motor safety feature](#)

Axis Conventions



This library uses the NED axes convention (North-East-Down as external reference in the world frame). The positive X axis points ahead, the positive Y axis points right, and the positive Z axis points down. Rotations follow the right-hand rule, so clockwise rotation around the Z axis is positive.

⚠ Note: This convention is different than the convention for joysticks which typically have -Y as Up (commonly mapped to throttle) and +X as Right. Pay close attention to the examples below if you want help with typical Joystick->Drive mapping.

Driving a robot using Differential Drive

WPIlib provides separate Robot Drive classes for the most common drive train configurations (differential, mecanum, and Killough). The DifferentialDrive class handles the differential drivetrain configuration. These drive bases typically have two or more in-line traction or omni wheels per side (e.g., 6WD or 8WD) and may also be known as "skid-steer", "tank drive", or "West Coast Drive". The Kit of Parts drivetrain is an example of a differential drive. There are methods to control the drive with 3 different styles ("Tank", "Arcade", or "Curvature"), explained in the article below.

Conventions and Defaults

For more information about conventions and defaults of the DifferentialDrive class see [WPIlib Drive classes: Conventions and Defaults](#)

Creating a Differential Drive object

C++

```
class Robot
{
    public:
        frc::Spark m_left{1};
        frc::Spark m_right{2};
        frc::DifferentialDrive m_drive{m_left, m_right};
}
```

Java

```
public class Robot
{
    Spark m_left = new Spark(1);
    Spark m_right = new Spark(2);
    DifferentialDrive m_drive = new DifferentialDrive(m_left, m_right);
}
```

Multi-Motor Drives

Many FRC drivetrains have more than 1 motor on each side. In order to use these with DifferentialDrive, the motors on each side have to be collected into a single SpeedController, using the SpeedControllerGroup class. The examples below show a 4 motor (2 per side) drivetrain. To extend to more motors, simply create the additional controllers and pass them all into the SpeedController group constructor (it takes an arbitrary number of inputs).

C++

```
class Robot
{
    public:
        frc::Spark m_frontLeft{1};
        frc::Spark m_rearLeft{2};
        frc::SpeedControllerGroup m_left{m_frontLeft, m_rearLeft};

        frc::Spark m_frontRight{3};
        frc::Spark m_rearRight{4};
        frc::SpeedControllerGroup m_right{m_frontRight, m_rearRight};

        frc::DifferentialDrive m_drive{m_left, m_right};
}
```

Java

```
public class Robot
{
    Spark m_frontLeft = new Spark(1);
    Spark m_rearLeft = new Spark(2);
    SpeedControllerGroup m_left = new SpeedControllerGroup(m_frontLeft, m_rearLeft);

    Spark m_frontRight = new Spark(3);
    Spark m_rearRight = new Spark(4);
    SpeedControllerGroup m_Right = new SpeedControllerGroup(m_frontRight, m_rearRight);
    DifferentialDrive m_drive = new DifferentialDrive(m_left, m_right);
}
```

FRC Java Programming

Drive Modes

The DifferentialDrive class contains 3 drive modes:

- Tank Drive - This mode uses one value each to control the individual sides of the drivetrain.
- Arcade Drive - This mode uses one value to control the throttle (speed along the X-axis) of the drivetrain and one for the rate of rotation.
- Curvature Drive - Also known as "Cheesy Drive" this is an alternate way of using one value to control throttle and one value for rotation. The rotation argument controls the curvature of the robot's path rather than its rate of heading change. This makes the robot more controllable at high speeds. Also handles the robot's quick turn functionality - "quick turn" overrides constant-curvature turning for turn-in-place maneuvers.

Tank Drive

The Tank Drive mode is used to control each side of the drivetrain independently (usually with an individual joystick axis controlling each). This example shows how to use the Y-axis of two separate joysticks to run the drivetrain in Tank mode. Construction of the objects has been omitted, for above for drivetrain construction and [here](#) for Joystick construction.

C++

```
class Robot: public frc::TimedRobot
{
    //Object construction

    void TeleopPeriodic() override {
        myDrive.TankDrive(leftStick.GetY(), rightStick.GetY());
    }
}
```

Java

```
public class RobotTemplate extends TimedRobot
{
    //Object construction
```

FRC Java Programming

```
public void teleopPeriodic() {  
    myDrive.tankDrive(leftStick.getY(), rightStick.getY());  
}  
}
```

Arcade Drive

The Arcade Drive mode is used to control the drivetrain using speed/throttle and rotation rate. This is typically used either with two axes from a single joystick, or split across joysticks (often on a single gamepad) with the throttle coming from one stick and the rotation from another. This example shows how to use a single joystick with the Arcade mode. Construction of the objects has been omitted, for above for drivetrain construction and [here](#) for Joystick construction.

C++

```
class Robot: public frc::TimedRobot  
{  
    //Object construction  
  
    void TeleopPeriodic() override {  
        myDrive.ArcadeDrive(driveStick.GetY(), driveStick.GetX());  
    }  
}
```

Java

```
public class RobotTemplate extends TimedRobot  
{  
    //Object construction  
  
    public void teleopPeriodic() {  
        myDrive.arcadeDrive(driveStick.getY(), driveStick.getX());  
    }  
}
```

FRC Java Programming

Curvature Drive

Like Arcade Drive, the Curvature Drive mode is used to control the drivetrain using speed/throttle and rotation rate. The difference is that the rotation control is attempting to control radius of curvature instead of rate of heading change. This mode also has a quick-turn parameter that is used to engage a sub-mode that allows for turning in place. This example shows how to use a single joystick with the Curvature mode. Construction of the objects has been omitted, for above for drivetrain construction and [here](#) for Joystick construction.

C++

```
class Robot: public frc::TimedRobot
{
    //Object construction

    void TeleopPeriodic() override {
        myDrive.CurvatureDrive(driveStick.GetY(), driveStick.GetX(), driveStick.
GetButton(1));
    }
}
```

Java

```
public class RobotTemplate extends TimedRobot
{
    //Object construction

    public void teleopPeriodic() {
        myDrive.curvatureDrive(driveStick.getY(), driveStick.getX(), driveStick.
GetButton(1));
    }
}
```

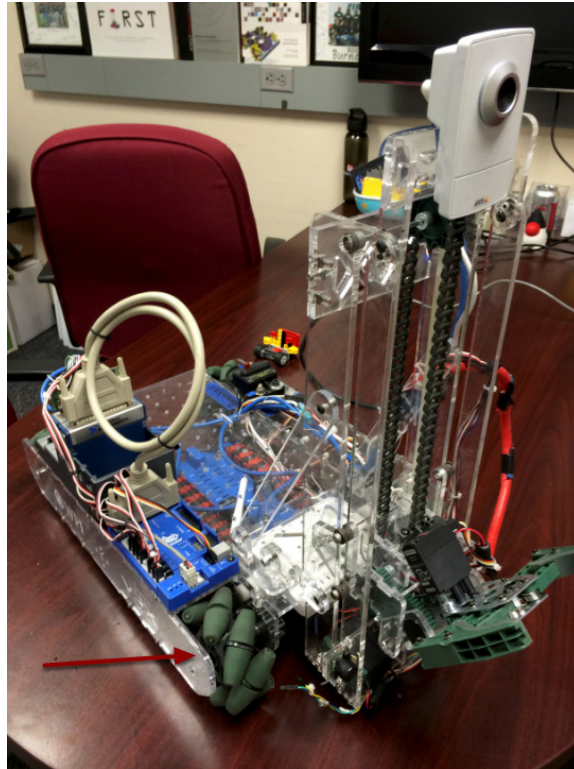
Driving a robot using Mecanum drive

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.

Conventions and Defaults

For information on conventions and defaults of the MecanumDrive class, see the [WPILib Drive classes: Conventions and Defaults](#) article.

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.

Controlling Mecanum: Cartesian vs Polar

The MecanumDrive class contains two ways of controlling the drivetrain:

- Cartesian: This method takes X, Y, and Rotation parameters and is commonly used when mapping joysticks to mecanum drive movement. The resulting robot translation is a combination of the desired X and Y movement.
- Polar: This method takes Magnitude, Angle, and Rotation parameters and is commonly used when controlling the robot autonomously. The angle should be specified in degrees around the Z-axis (between -180 and 180).

FRC Java Programming

Code for teleop driving with mecanum wheels

Here's a sample program that shows the minimum code to drive using a single joystick and mecanum wheels. The joystick XY position represents a robot-relative direction vector that the robot should follow. The twist (Z) axis on the joystick represents the rate of rotation for the robot while it's driving.

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:
 *   - PWM 0 - Connected to front left drive motor
 *   - PWM 1 - Connected to rear left drive motor
 *   - PWM 2 - Connected to front right drive motor
 *   - PWM 3 - Connected to rear right drive motor
 */
class MecanumDefaultCode : public frc::TimedRobot
{
    frc::Spark m_frontLeft{0};
    frc::Spark m_rearLeft{1};
    frc::Spark m_frontRight{2};
    frc::Spark m_rearRight{3};
    frc::MecanumDrive m_drive{m_frontLeft, m_rearLeft, m_frontRight, m_rearRight};
    frc::Joystick m_driveStick{1};

    /**
     * Gets called once for each new packet from the DS.
     */
    void TeleopPeriodic override (void)
    {
        m_robotDrive.MecanumDrive_Cartesian(m_driveStick.GetX(), m_driveStick.
GetY(), m_driveStick.GetZ());
    }

};
START_ROBOT_CLASS (MecanumDefaultCode);
```


FRC Java Programming

Java

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

/*
 * Simplest program to drive a robot with mecanum drive using a single Logitech
 * Extreme 3D Pro joystick and 4 drive motors connected as follows:
 *     - PWM 0 - Connected to front left drive motor
 *     - PWM 1 - Connected to rear left drive motor
 *     - PWM 2 - Connected to front right drive motor
 *     - PWM 3 - Connected to rear right drive motor
 */

public class MecanumDefaultCode extends TimedRobot {
    //Create a robot drive object using PWMs 0, 1, 2 and 3
    Spark m_frontLeft = new Spark(1);
        Spark m_rearLeft = new Spark(2);
        Spark m_frontRight = new Spark(3);
        Spark m_rearRight = new Spark(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.getZ());
    }
}
```

Updating the program for field-oriented driving

There is also an optional 4th parameter to the `MecanumDrive_Cartesian()` method that is the angle returned from a Gyro sensor. This will adjust the X/Y values supplied, in this case, from 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

FRC Java Programming

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 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_Cartesian() is called.

C++

```
m_robotDrive.MecanumDrive_Cartesian(m_driveStick.GetX(), m_driveStick.GetY(), m_driveStick.  
GetZ(), m_gyro.GetAngle());
```

Java

```
m_robotDrive.mecanumDrive_Cartesian(m_driveStick.getX(), m_driveStick.getY(), m_driveStick.  
getZ(), m_gyro.getAngle());
```

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);
```

FRC Java Programming

```
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

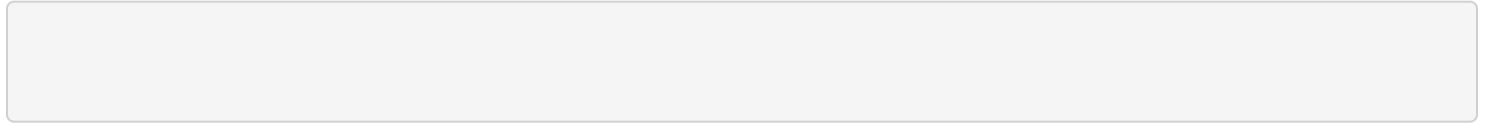
C++

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

Java

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

FRC Java Programming



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);
```


FRC Java Programming

```
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();  
double current = c->GetCompressorCurrent();
```

Java

```
boolean enabled = c.enabled();  
boolean pressureSwitch = c.getPressureSwitchValue();  
double 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 [Updating and Configuring Pneumatics Control Module and Power Distribution Panel](#).

Single Solenoids in WPILib

C++

```
frc::Solenoid exampleSolenoid {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++

```
frc::DoubleSolenoid exampleDouble {1, 2};  
  
exampleDouble.Set(frc::DoubleSolenoid::Value::kOff);  
exampleDouble.Set(frc::DoubleSolenoid::Value::kForward);  
exampleDouble.Set(frc::DoubleSolenoid::Value::kReverse);
```

Java

FRC Java Programming

```
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. CAN speed controllers
2. The Power Distribution Panel (PDP)
3. The Pneumatics Control Module (PCM)

The devices are typically connected to the RoboRIO CAN bus using twisted pair wiring.

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.

If you wish to terminate your bus somewhere other than the PDP, the PDP terminator jumper must be moved to disable the PDP terminator and a user provided 120 Ohm resistor must be placed at the end of the bus.

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.

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);
```

FRC Java Programming

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](#).

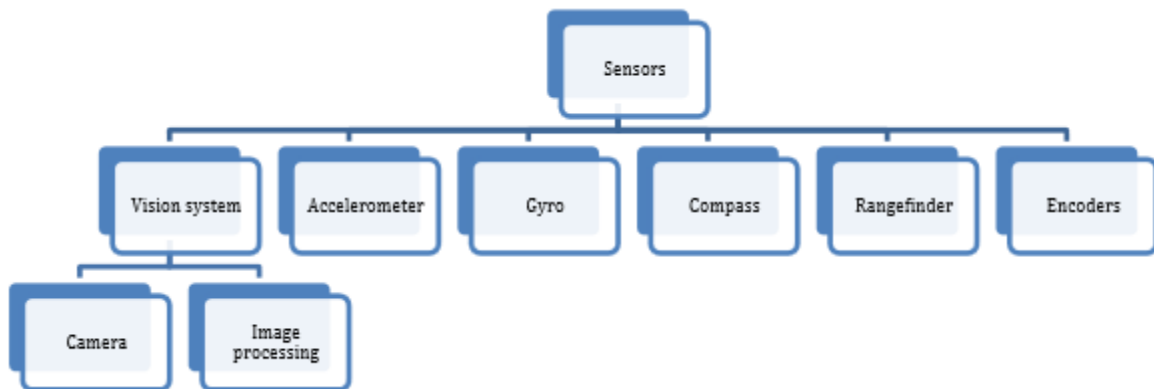
Note: CAN Talon SRX has been removed from WPILib. See [this blog](#) for more info and find the CTRE Toolsuite installer here: http://www.ctr-electronics.com/control-system/hro.html#product_tabs_technical_resources

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

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;
```

FRC Java Programming

```
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() {
```


FRC Java Programming

```
        // 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();
    }
}
```

FRC Java Programming

```
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);  
    }  
}
```

FRC Java Programming

```
}

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() {
```

FRC Java Programming

```
    }

    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 won't 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.

FRC Java Programming

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](#)

FRC Java Programming

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 your 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](#)

FRC Java Programming

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 won't know how far it's 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 lose distance in slack from the pulleys, the belts jumping over 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 it's 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.

FRC Java Programming

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](#)

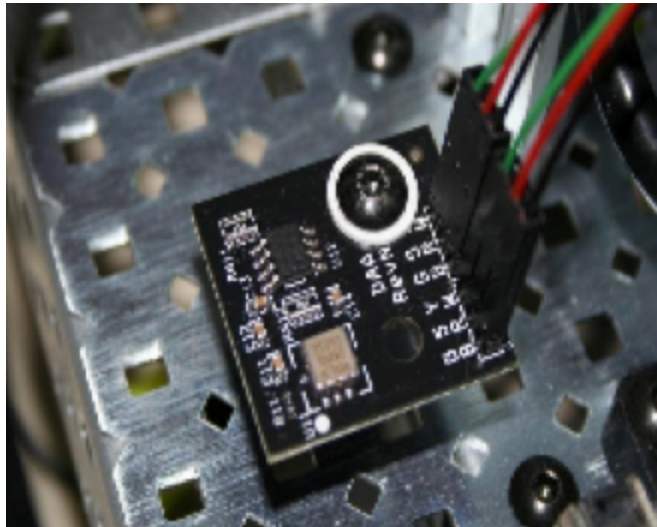
Other Sensors and Problems

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.**

FRC Java Programming

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() {
```

FRC Java Programming

```
        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();
```

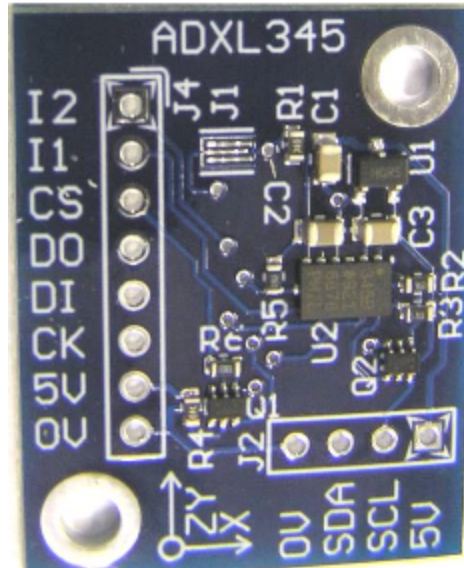
Java

```
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.

FRC Java Programming

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()
    {
```

FRC Java Programming

```
        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,

FRC Java Programming

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.

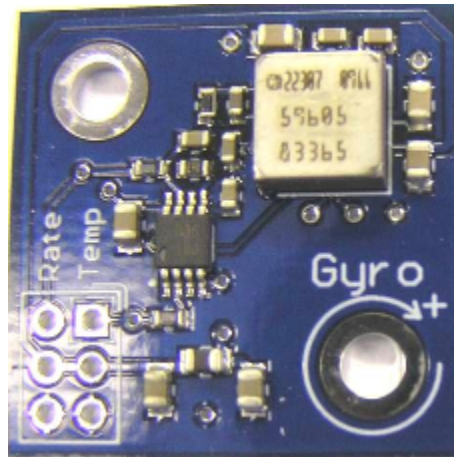
Another important specification for the gyro is its full-scale range. Gyros with high full-scale ranges can measure fast rotation without “pinning” the output. The scale is much larger so faster rotation rates can be read, but there is less resolution due to a much larger range of values spread over 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.

FRC Java Programming

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 initialization 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
        }
    }
}
```

FRC Java Programming

```
        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);
        }
    }
}
```

FRC Java Programming

```
    }  
    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

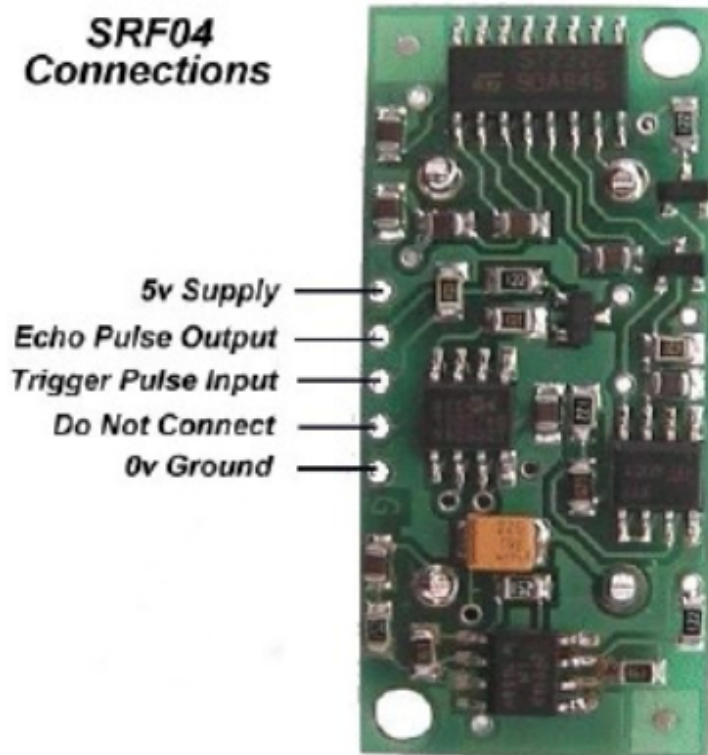
Ultrasonic sensors are a common way to find the distance from a robot to the nearest surface

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
```

FRC Java Programming

```
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 and assigns
    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
    }
}
```


FRC Java Programming

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.

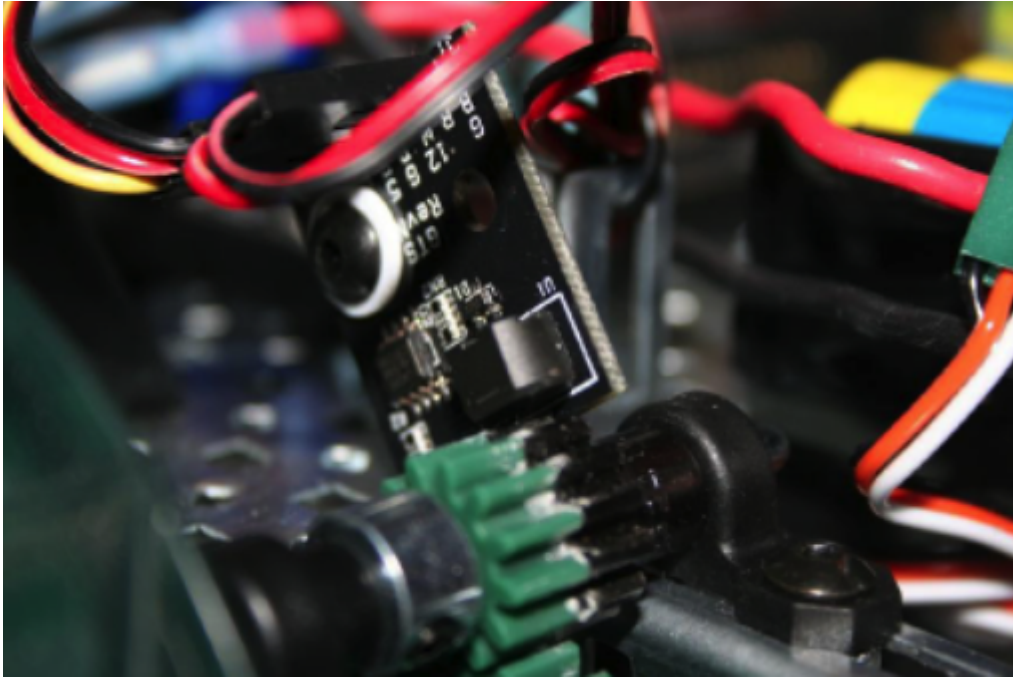
Counter Overview

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);
```

FRC Java Programming

```
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();
```

FRC Java Programming

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

FRC Java Programming

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:
```

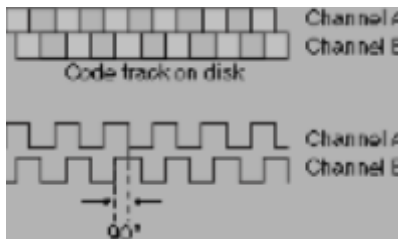
FRC Java Programming

- 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

Encoders vs. Counters

FRC Java Programming

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().

FRC Java Programming

- 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

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

$$\text{Oversample}(AI_O) = \sum_0^{2^N-1} AI$$

Where N is the number of Oversample bits

$$\text{Average} = \left(\sum_0^{2^M-1} AI_O \right) / 2^M$$

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;
```

FRC Java Programming

```
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:

FRC Java Programming

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);
```


FRC Java Programming

```
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

FRC Java Programming

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 form 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: $(\text{Analog Input Voltage} / \text{Analog Supply Voltage}) * \text{FullScale} + \text{Offset}$. 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);
```

FRC Java Programming

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;
```

FRC Java Programming

```
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

Creating a PIDController object

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`

FRC Java Programming

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.

FRC Java Programming

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
18     public void initDefaultCommand() {
19     }
20
21     protected double returnPIDInput() {
22         return pot.getAverageVoltage();
23     }
24
25     protected void usePIDOutput(double output) {
26         motor.pidWrite(output);
27     }
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

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 [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

FRC Java Programming

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

FRC Java Programming

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 FMS querying 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();
```


FRC Java Programming

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.

FRC Java Programming

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

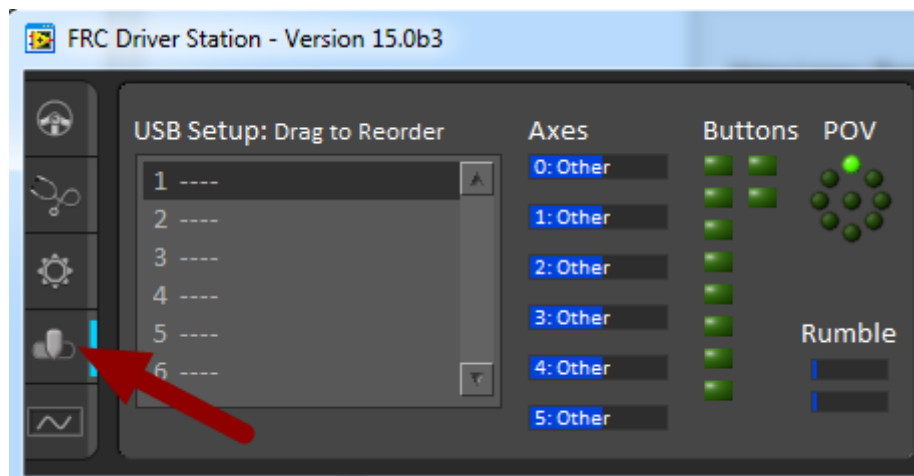
```
double time;  
time = DriverStation.getInstance().getMatchTime();
```

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.

FRC Java Programming

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);
```

FRC Java Programming

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();
```

FRC Java Programming

```
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.getRawAxis(1);
```

FRC Java Programming

```
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++

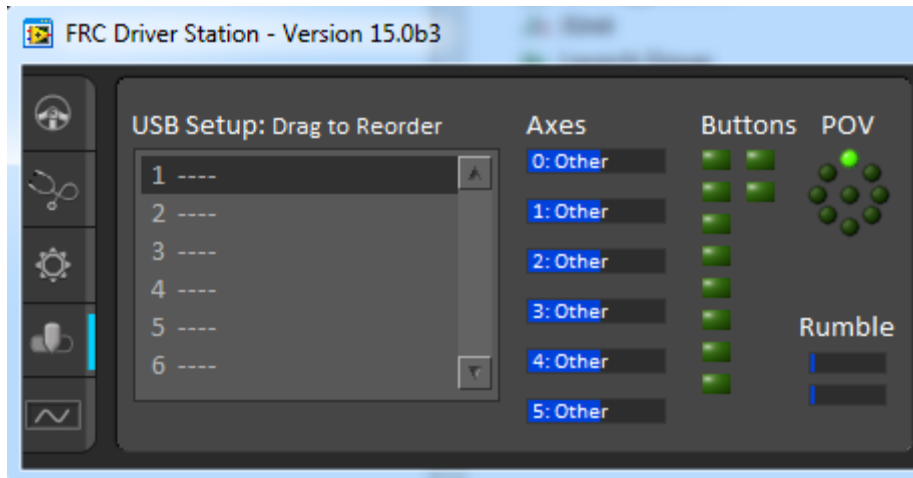
```
double value;  
value = exampleStick->GetDirectionDegrees();  
value = exampleStick->GetDirectionRadians();  
value = exampleStick->GetMagnitude();
```

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

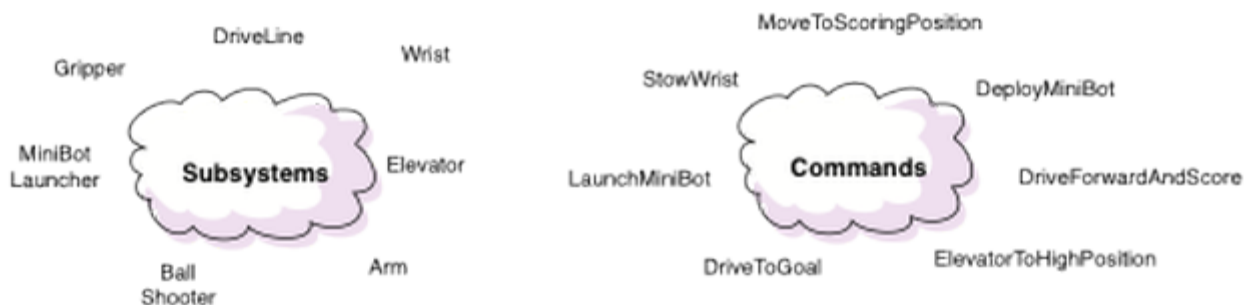
What is Command based programming?

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



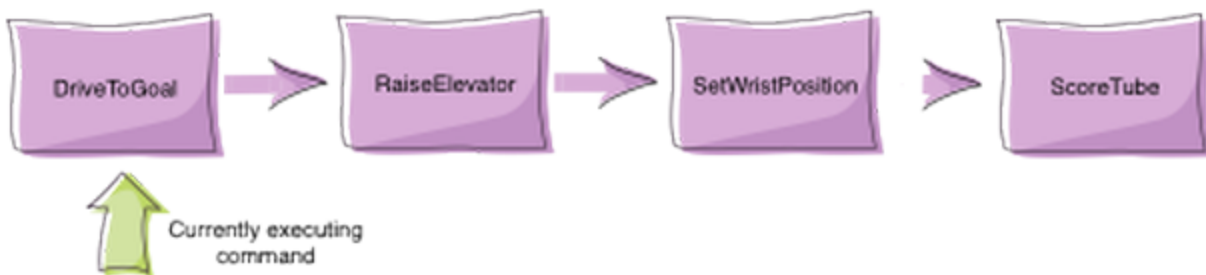
Programs based on the WPILib library are organized around two fundamental concepts: **Subsystems** and **Commands**.

FRC Java Programming

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

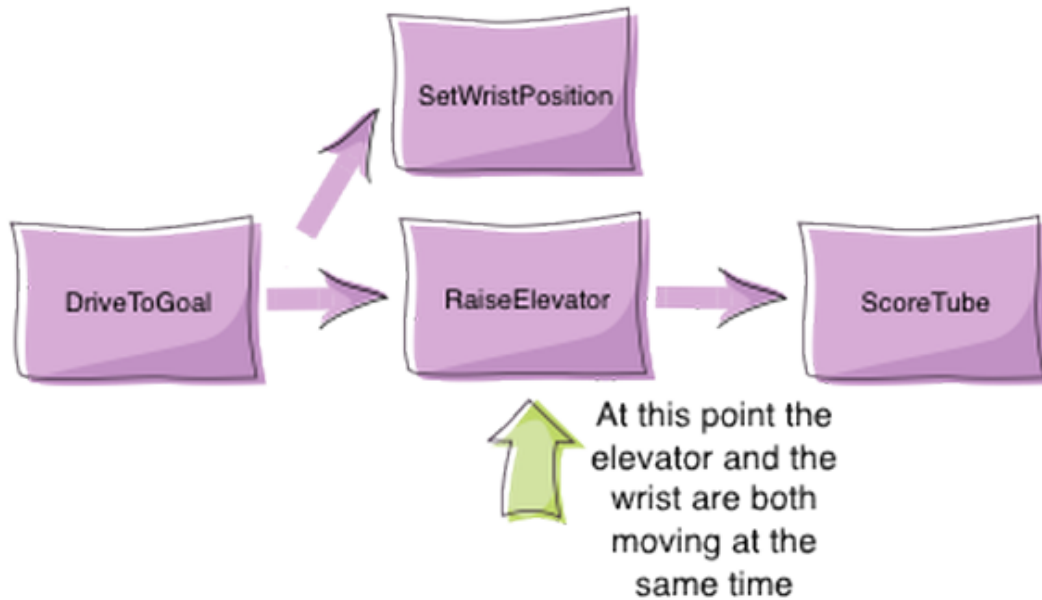
Commands - define the operation of the robot incorporating the capabilities defined in the subsystems. Commands are subclasses of Command or CommandGroup. Commands run when scheduled or in response to buttons being pressed or virtual buttons from the SmartDashboard.

How commands work



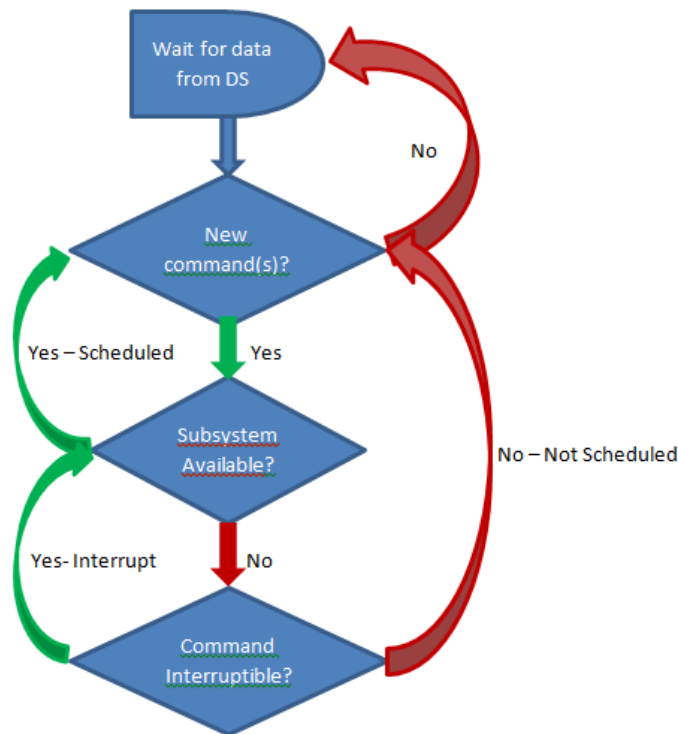
Commands let you break up the tasks of operating the robot into small chunks. Each command has an `execute()` method that does some work and an `isFinished()` method that tells if it is done. This happens on every update from the driver station or about every 20ms. Commands can be grouped together and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



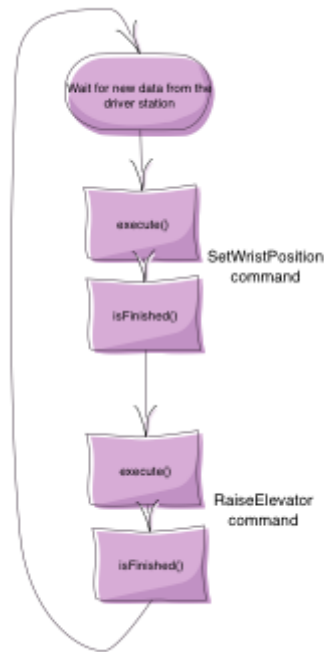
There are three main ways commands are scheduled:

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.

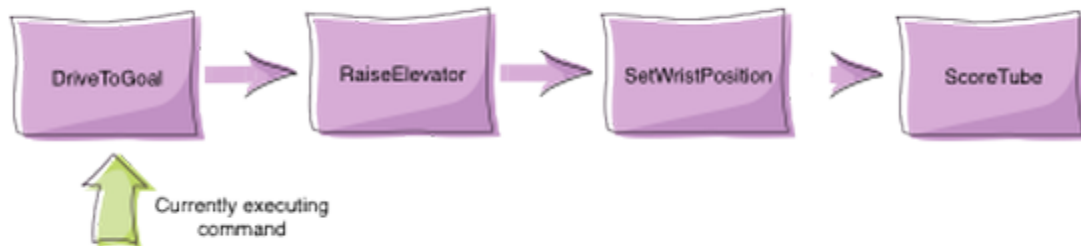
When a command is scheduled, the Scheduler checks to make sure that no other commands are using the same subsystems that the new command requires. If one or more of the subsystems is currently in use, and the current command is interruptible, it will be interrupted and the new command will be scheduled. 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

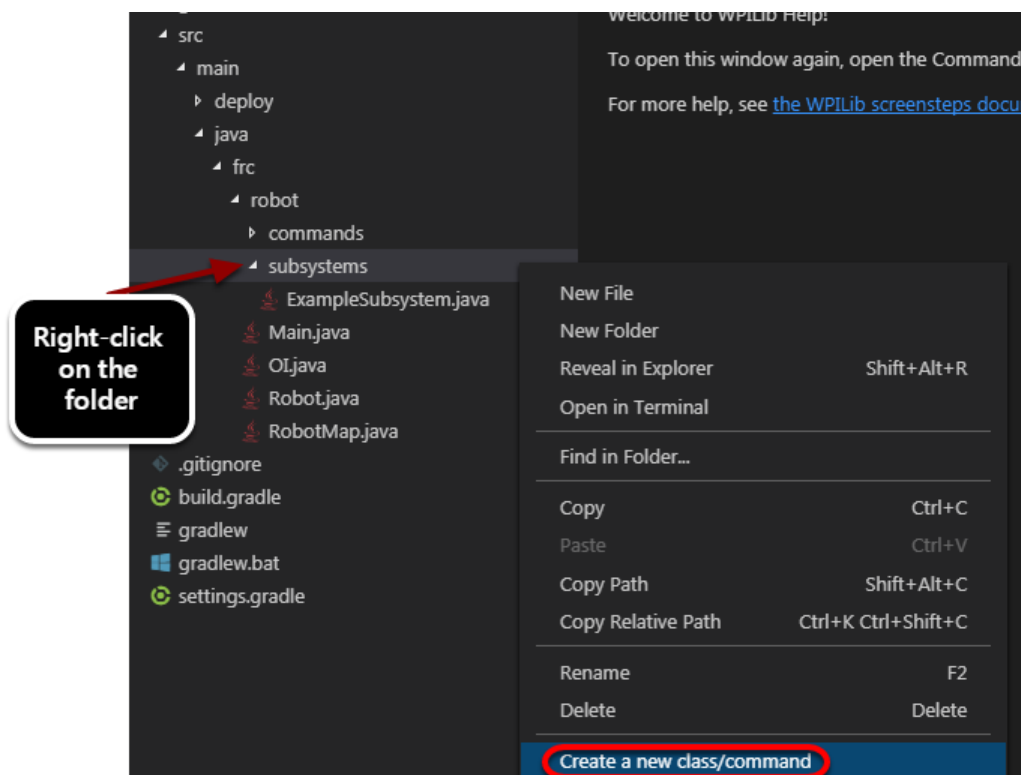
Create a command-based robot project by using one of the template projects that are provided with the VSCode plugins.

Creating a project is detailed in [Creating a robot program](#). Select "Template" then "Command Robot" to create a basic Command Based Robot program. Alternately you can use RobotBuilder to create the framework of your Command Based Robot project as detailed in [RobotBuilder](#).

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.

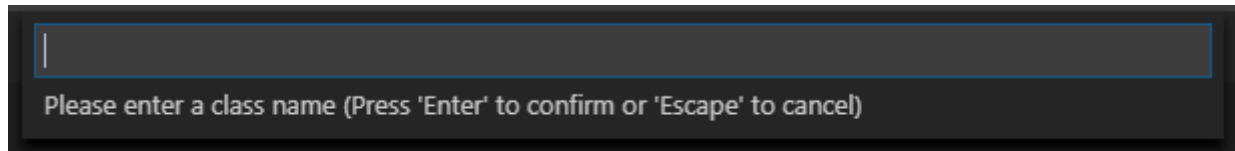
Adding subsystems to the project



To add a subsystem, right-click on the desired folder and select **Create a new class/command** in the drop down menu. Then select **Subsystem** or **PID Subsystem**.

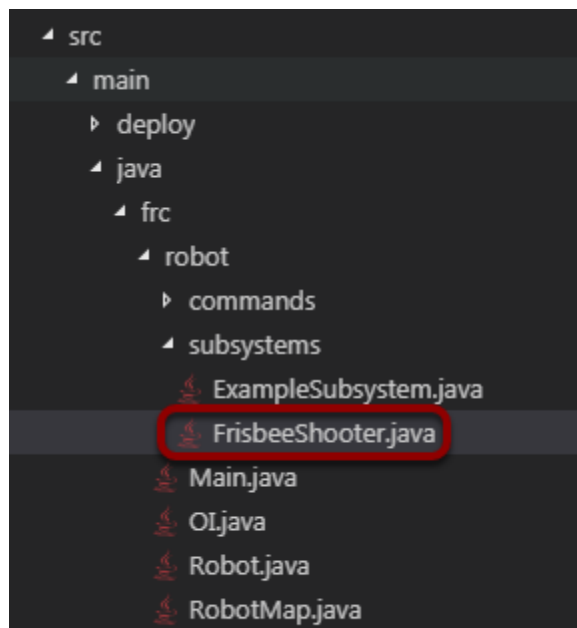
FRC Java Programming

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.

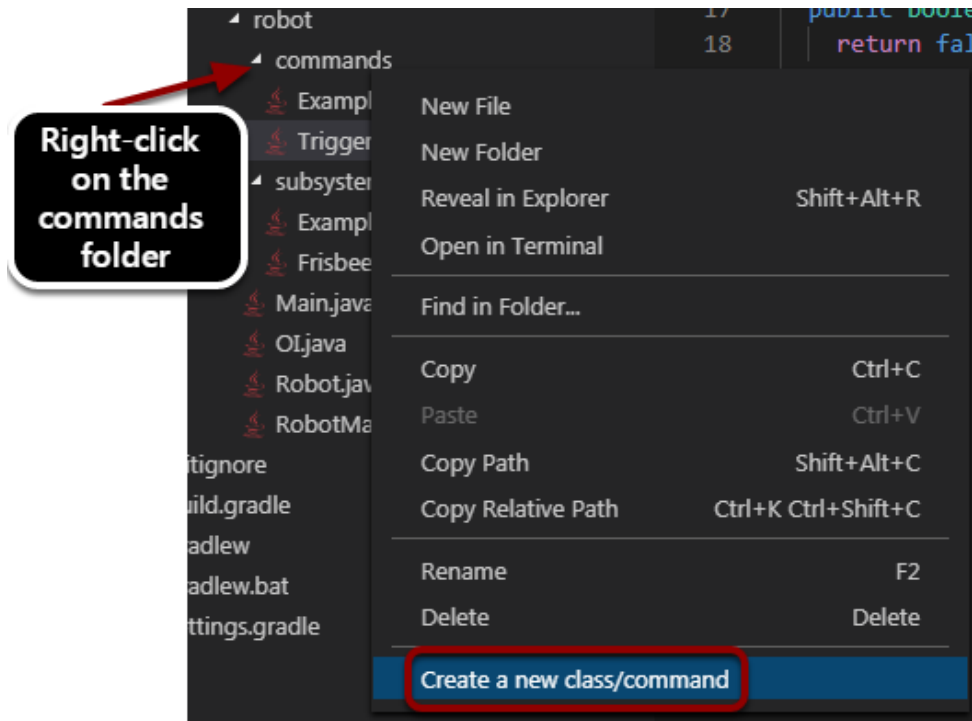
Subsystem created in project



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.

FRC Java Programming

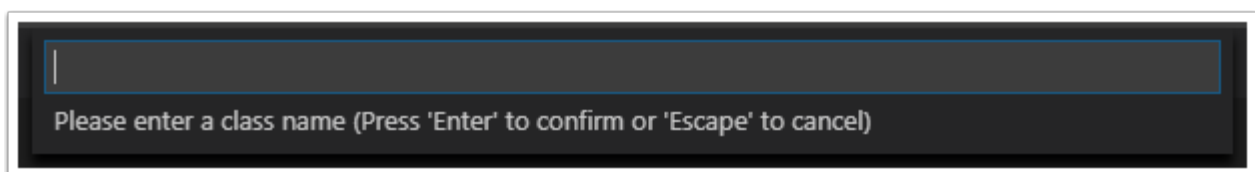
Adding a command to the project



A command can be created for the project using steps similar to creating a subsystem. First right-click on the folder name in the project, then select **Create a new class/command** in the drop down menu. Then select **Command**, **Instant Command**, **TimedCommand** or **Trigger**:

- **Command** - A basic command that operates on a subsystem
- **Instant Command** - A command that runs and completes instantly
- **Timed Command** - A command that runs for a specified time duration
- **Trigger** - A command that is easily tied to a button input on a joystick.

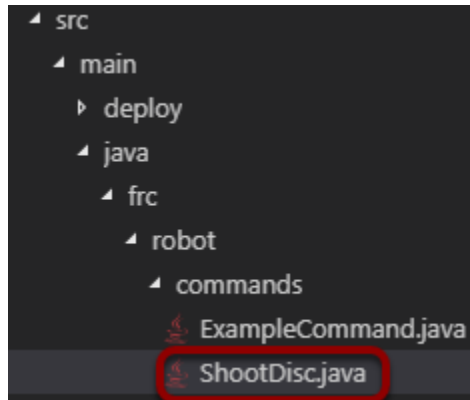
Set the command name



FRC Java Programming

Enter the Command name into 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);
    }
}
```

FRC Java Programming

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() {
    }
```

FRC Java Programming

```
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 used when computing the motor output
        setAbsoluteTolerance(0.05);
        getPIDController().setContinuous(false);
    }
}
```

FRC Java Programming

```
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 from the  
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()
{
}

/*
```

FRC Java Programming

```
*      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.
    */
}
```

FRC Java Programming

```
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"
```

FRC Java Programming

```
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()
{
}
```

FRC Java Programming

```
        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
```

FRC Java Programming

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());
    }
}
```

FRC Java Programming

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());
    }
}
```

FRC Java Programming

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());
    AddParallel(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));
    }
}
```

FRC Java Programming

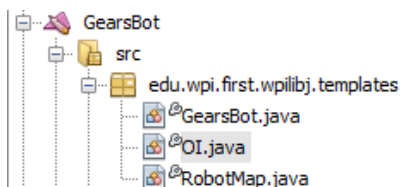
```
}
```

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);
```

FRC Java Programming

```
}
```

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));
```

FRC Java Programming

```
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()`.

FRC Java Programming

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());  
}
```

FRC Java Programming

```
        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();
    }
}
```

FRC Java Programming

```
}

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() {
```

FRC Java Programming

```
// 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. RobotInit() is called only once when the robot starts so it is a good time to create the command instance.
2. Start it during the AutonomousInit() method. AutonomousInit() is called once at the start of the autonomous period so we schedule the command there.
3. Be sure the scheduler is called repeatedly during the AutonomousPeriodic() method. AutonomousPeriodic() is called (nominally) 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
```

FRC Java Programming

```
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 performed.
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 shooter up 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()
{
}
```


FRC Java Programming

```
    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();  
    }  
}
```

FRC Java Programming

```
}
```

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()
{
```

FRC Java Programming

```
\    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() {
```

FRC Java Programming

```
        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
```

FRC Java Programming

```
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
```

FRC Java Programming

```
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.

FRC Java Programming

- **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());
}
```


FRC Java Programming

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));
}
```

FRC Java Programming

Java

```
public class CoopBridgeAutonomous extends CommandGroup {

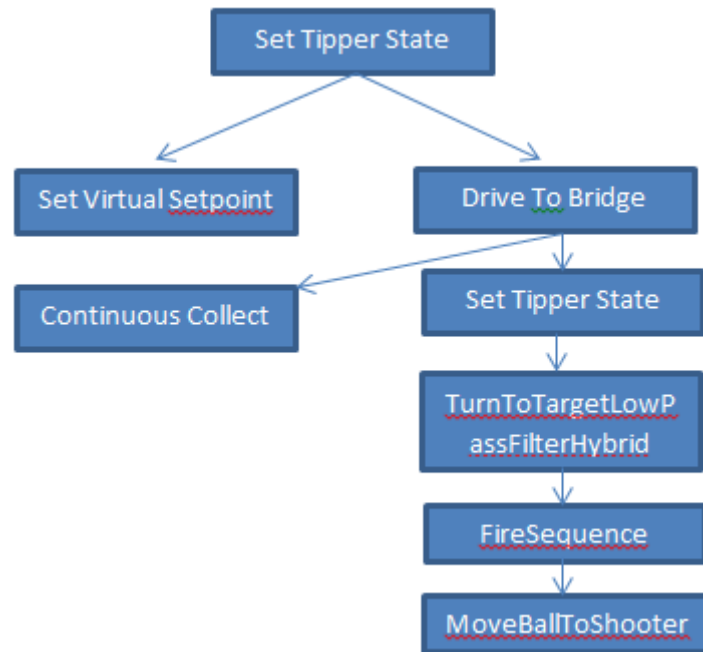
    public CoopBridgeAutonomous() {
        //SmartDashboard.putDouble("Camera Time", 5.0);
        addSequential(new SetTipperState(BridgeTipper.READY_STATE)); // 1
        addParallel(new SetVirtualSetpoint(SetVirtualSetpoint.HYBRID_LOCATION)); // 2
        addSequential(new DriveToBridge()); // 3
        addParallel(new ContinuousCollect());
        addSequential(new SetTipperState(BridgeTipper.DOWN_STATE));

        // addParallel(new WaitThenShoot());

        addSequential(new TurnToTargetLowPassFilterHybrid(4.0));
        addSequential(new FireSequence());
        addSequential(new MoveBallToShooter(true));
    }
}
```

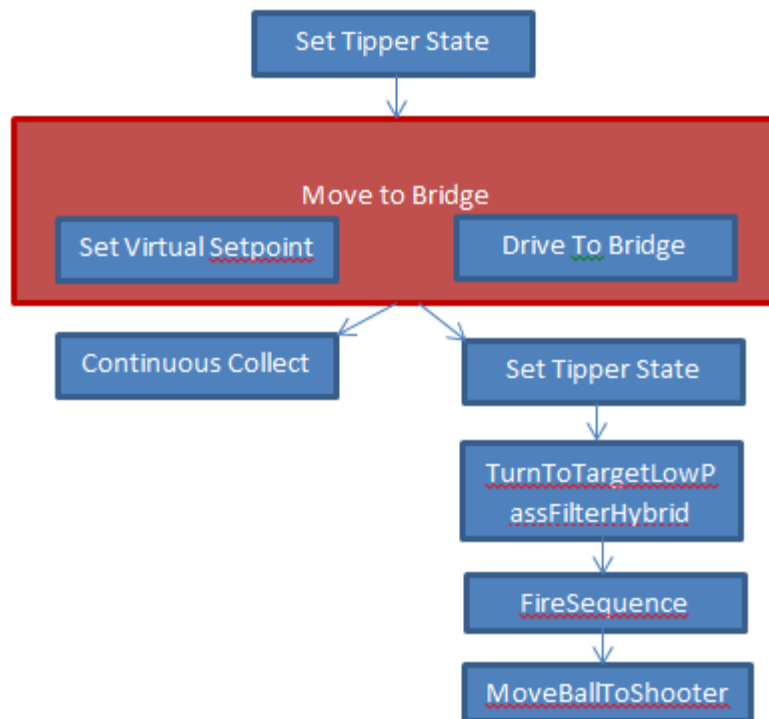
In this example some commands are added in parallel and others are added sequentially to the CommandGroup CoopBridgeAutonomous (1). The first command "SetTipperState" is added and completes before the SetVirtualSetpoint command starts (2). Before SetVirtualSetpoint command completes, the DriveToBridge 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.

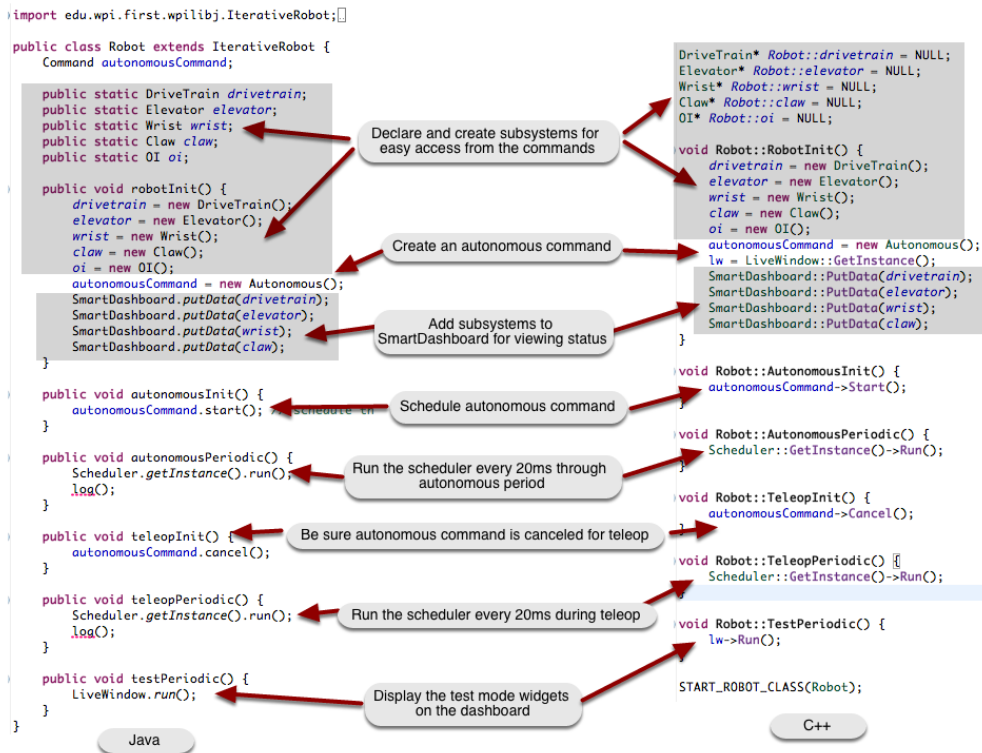
In this example you can see that the addition of a command group "Move to Bridge" containing the Set Virtual Setpoint and Drive to Bridge commands forces both to complete before the next commands are scheduled.

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.

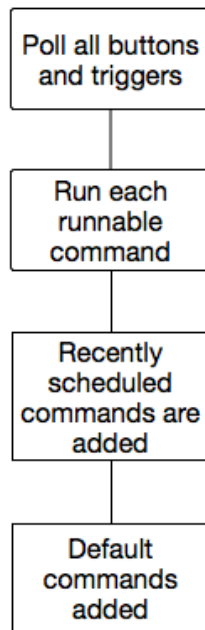
Anatomy of a command-based robot program



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

Scheduler.run()



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

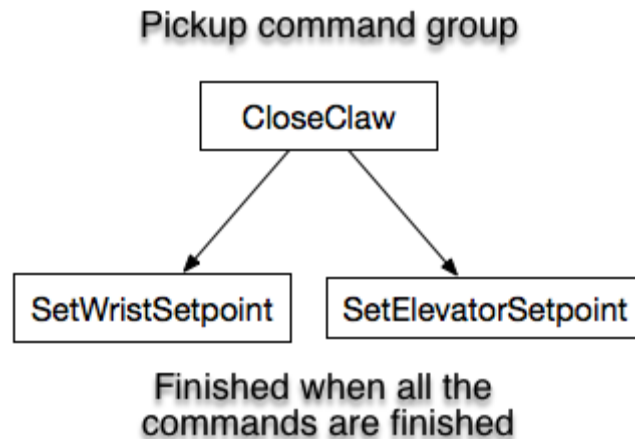
```
public class Pickup extends CommandGroup {  
    public Pickup() {  
        addSequential(new CloseClaw());  
        addParallel(new SetWristSetpoint(-45));  
        addSequential(new SetElevatorSetpoint(0.25));  
    }  
}
```

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 its 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 start() method (Java) or Start() 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++

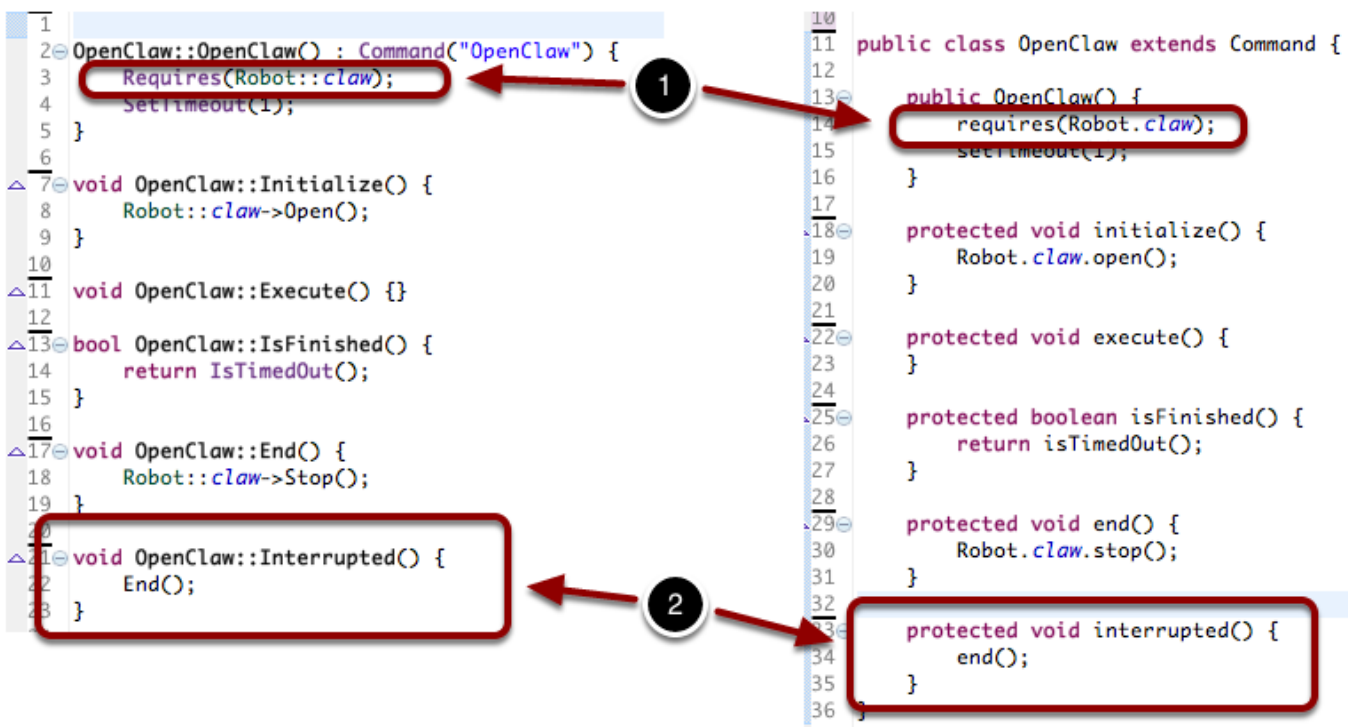
```
Scheduler::RemoveAll();
```

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?



FRC Java Programming

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.

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 too far and causing damage.

What values are provided by the limit switch

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;
}
```

FRC Java Programming

```
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);
        }
    }
}
```

FRC Java Programming

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()
{
}
```

FRC Java Programming

```
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

FRC Java Programming

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);
}
```

FRC Java Programming

```
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();
    }
}
```

FRC Java Programming

```
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()  
{  
}
```

FRC Java Programming

```
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();
    }
}
```

FRC Java Programming

```
        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.