WPILIB PROGRAMMING

WPILib programming

Table of Contents

WPILib programming

Driver Station Inputs and Feedback

Robot to driver station networking

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

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 for robot programs than previously used languages. The WPI Robotics Library is designed for maximum extensibility and software reuse with these languages.

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

Fundamentally, C++ offers the highest performance possible for your robot programs. 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.

Java programming with WPILib

Java programming with WPILib

- 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++ programming with WPILib

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

Choosing a Base Class

The base class is the framework that the robot code is constructed on top of. WPILib offers two different base classes, as well as a third option which is not technically a separate base class.

Simple Robot

```java
public class RobotTemplate extends SimpleRobot {
    /**
     * This function is called once each time the robot enters autonomous mode.
     */
    public void autonomous() {
        while(isAutonomous() && isEnabled())
        {
            //Put code here
            Timer.delay(.05);
        }
    }

    /**
     * This function is called once each time the robot enters operator control.
     */
    public void operatorControl() {
        while(isOperatorControl() && isEnabled())
        {
            //Put code here
            Timer.delay(.05);
        }
    }
}
```

The SimpleRobot class is the simplest to understand as most of the state flow is directly visible in your program. Your robot program overrides the operatorControl() and autonomous() methods that are called by the base at the appropriate time. Note that these methods are called only called once each time the robot enters the appropriate mode and are not automatically terminated. Your code in the operatorControl method must contain a loop that checks the robot mode in order to

keep running and taking new input from the Driver Station. The autonomous code shown uses a similar loop.

Iterative Robot

```java
public class RobotTemplate extends IterativeRobot {
    /**
     * This function is run when the robot is first started up and should be
     * used for any initialization code.
     */
    public void robotInit() {

    }

    /**
     * This function is called once each time the robot enters autonomous
     */
    public void autonomousInit() {

    }

    /**
     * This function is called periodically during autonomous
     */
    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 driverstation 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 cRIO CPU leaving more time for other tasks such as camera processing.

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 manual.

Sending data from the cRIO to an Arduino

Sometimes it is useful to use a coprocessor to handle operations on some sensors, lights, etc. A popular processor is the Arduino. This article shows sample code to send some data between the cRIO and an Arduino. Although it only sends data in one direction (from the cRIO to the Arduino), it serves as an example of how to do it.

This program sends one of two values (either 72 or 76) from the cRIO to either turn the LED (pin 13 on the Arduino) either on or off. The value is arbitrary and was just part of a larger sample program.

The cRIO program
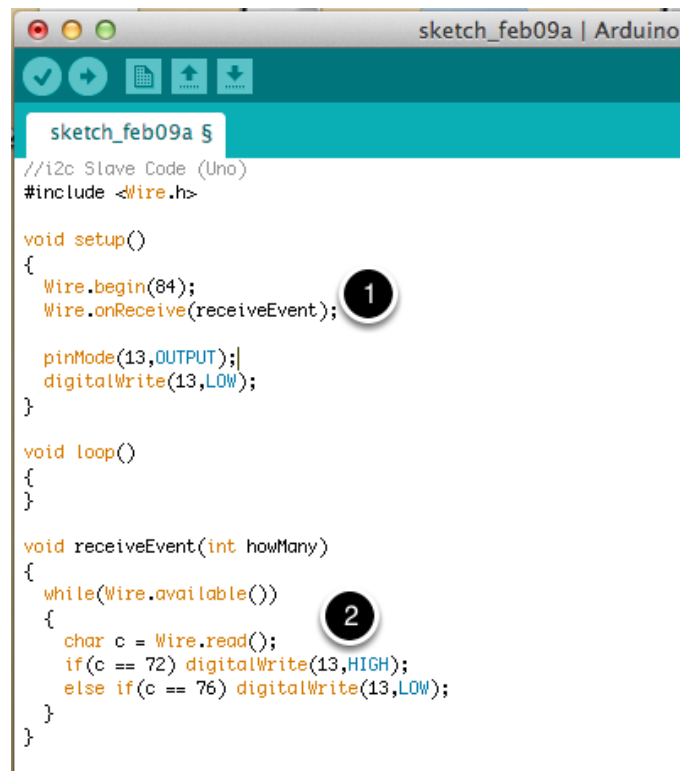
```
1   package edu.wpi.first.wpilibj.templates;
2
3   import edu.wpi.first.wpilibj.DigitalModule;
4   import edu.wpi.first.wpilibj.I2C;
5   import edu.wpi.first.wpilibj.SimpleRobot;
6   import edu.wpi.first.wpilibj.Timer;
7
8   public class RobotTemplate extends SimpleRobot {
9
10      I2C i2c;
11      byte[] toSend = new byte[1];
12
13      public void robotInit() {
14          DigitalModule module = DigitalModule.getInstance(2);
15          i2c = module.getI2C(168);                    (1)
16      }
17
18      public void autonomous() {
19
20      }
21
22      public void operatorControl() {
23          boolean on = false;
24          System.out.println("Starting OperatorControl");
25          while (isOperatorControl()) {
26              if (on)
27                  toSend[0] = 76;                       (2)
28              else
29                  toSend[0] = 72;
30              on = ! on;
31              i2c.transaction(toSend, 1, null, 0);     (3)
32              Timer.delay(.0005);
33          }
34      }
35
36      public void test() {
37
38      }
39  }
```

WPILib programming

The I2C protocol has a master and slave processors or devices. The master controls the bus and either sends data to a slave or requests data from a slave processor. Slaves cannot initiate transactions on their own. Each slave processor (or device) has a unique address that the master processor uses to select it. In this example, the Arduino slave processor recognizes address 84. The steps are:

1. Initialize the I2C connection on address 84. Because of differences between the implementation of the library for the cRIO and Arduino (the lower bit of the address selects either read or write) the cRIO uses address 168. 168 is the address 84 shifted by 1 bit (the read/write bit).
2. Use a byte array to fill with the data to send. In this case it's a single byte, either 76 or 72 that sets the light on or off on the Arduino.
3. Send the data to the Arduino without receiving any data. The parameters "toSend" and "1" specify a single byte from the "toSend" array. The second set of parameters (null and 0) would be a byte array and length if the master was expecting the slave to respond with some data.

The Arduino program

The Arduino program:

WPILib programming

1. Sets up an interrupt handler to receive inbound requests as address 84
2. Reads the data turning the light on or off depending on the value.

The results
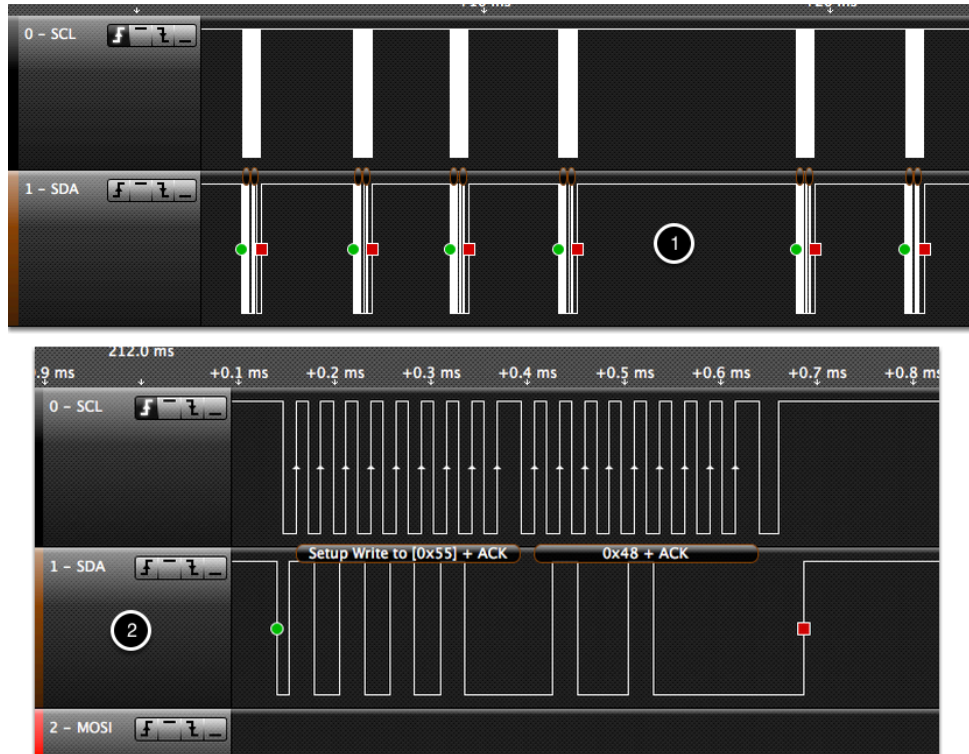
A few observations:

1. The timing using Timer.delay() on the cRIO wasn't very precise - likely due to randomness in the Java scheduler and getting the thread restarted. You can see that between transactions.
2. The program didn't work reliably with an Arduino Uno and seems to work correctly with a Mega. You can see the output of each transaction with the trace from the logic analyzer.

Getting your robot to drive with the RobotDrive class

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

Note: the examples illustrated in this section are generally correct but have not all been tested on actual robots. But should serve as a starting point for your projects.

Creating a RobotDrive object with Jaguar speed controllers

Creating a RobotDrive object with Jaguar speed controllers

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

Using other types of speed controllers

Using other types of speed controllers

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

These are Java programs but the C++ program is very similar.

Tank driving with two joysticks

Tank driving with two joysticks

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

Arcade driving with a single joystick

Arcade driving with a single joystick

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

Autonomous driving using the RobotDrive object

Autonomous driving using the RobotDrive object

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

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

WPILib programming

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

Mecanum driving

```java
public class RobotTemplate extends SimpleRobot {

    RobotDrive myDrive;
    Joystick moveStick, rotateStick;

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

    public void autonomous() {
    }

    public void operatorControl() {
        while (isAutonomous() && isEnabled()) {
            myDrive.mecanumDrive_Polar(moveStick.getY(), moveStick.getX(), rotateStick.getTwist());
            Timer.delay(0.01);
        }
    }
}
```

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

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

Using actuators (motors, servos, and relays)

Actuator Overview

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

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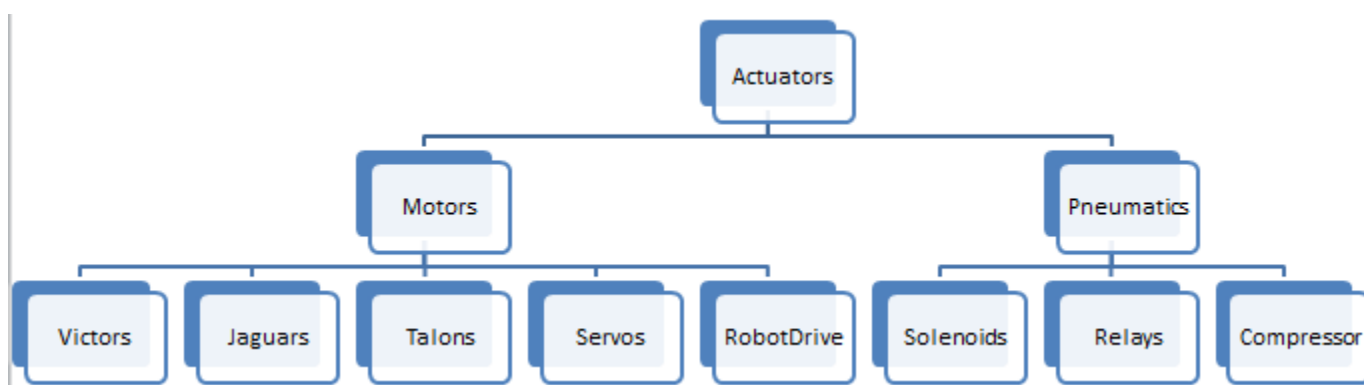
The chart shown above outlines the types of actuators which can be controlled through WPILib. The articles in this section will cover each of these types of actuators and the WPILib methods and classes that control them.

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

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

PWM Controllers, brief theory of operation

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

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

Raw vs Scaled output values

In general, all of the motor controller classes in WPILib are set up to take a scaled -1.0 to 1.0 value as the output to an actuator. The PWM module in the FPGA is capable of generating PWM signals

with periods of 5, 10 or 20ms and can vary the pulse width in 255 steps of ~.0065ms each around the midpoint. The raw values sent to this module are in this 0-255 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.

PWM and Safe PWM Classes

PWM

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

Safe PWM

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

Constructing a Speed Controller object

```
exampleTalon = new Talon(1);
exampleVictor = new Victor(1,2);
exampleJaguar = new Jaguar(3);
```

Speed controller objects are constructed by passing in either a channel (default module) or a channel and module. No other parameters are passed into the constructor.

Setting parameters

```
exampleJaguar.enableDeadbandElimination(true);
```

All of the settable parameters of the motor controllers inherit from the underlying PWM class and are thus identical across the controllers. The code above shows only a single controller type (Talon) 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

```
exampleJaguar.set(.7);
```

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

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

```
exampleServo = new Servo(1);
exampleServo = new Servo(1,2);
```

A servo object is constructed by passing either a channel (default module) or module and channel.

Setting Servo Values

```
exampleServo.set(.5);
exampleServo.setAngle(85);
```

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.

Composite controllers - RobotDrive

The **RobotDrive** class is designed to simplify the operation of the drive motors based on a model of the drive train configuration. The program describes the layout of the motors. Then the class can generate all the speed values to operate the motors for different configurations. For cases that fit the model, it provides a significant simplification to standard driving code. For more complex cases that aren't directly supported by the **RobotDrive** class it may be subclassed to add additional features or not used at all.

Create a RobotDrive object with 2 motors

Create a RobotDrive object with 2 motors

First, create a **RobotDrive** object specifying the left and right Jaguar motor controllers on the robot, as shown.

Creating a RobotDrive object with 4 motors

Creating a RobotDrive object with 4 motors

In this case, for a four motor drive all the motors are specified in the constructor.

Creating a RobotDrive object using speed controllers that are already created

By default, the RobotDrive object created with port numbers as shown in the previous two examples will allocate Jaguar speed controller objects for each of the motors. If the RobotDrive object creates the speed controllers, then it will also be responsible for deleting them when the RobotDrive object is deleted.

WPILib programming

In some case (as shown here) you might want to be in control of the speed controller objects, for example, at times your program might have a need to operate them independently from the RobotDrive object. Another case is if your robot is not using Jaguar speed controllers. In this case, allocate the desired speed controller objects and pass them as parameters to the constructor. Your program will be responsible for deleting the objects when you are done using them.

Operating the motors of the RobotDrive

Once set up, there are methods that can help with driving the robot either from the Driver Station controls or through programmed operations. These methods are described in the table below.

Drive(speed, turn) - Designed to take speed and turn values ranging from - 1.0 to 1.0. The speed values set the robot overall drive speed; with positive values representing forward and negative values representing backwards. The turn value tries to specify constant radius turns for any drive speed. Negative values represent left turns and the positive values represent right turns.

TankDrive(leftStick, rightStick) - Takes two joysticks and controls the robot with tank steering using the y-axis of each joystick. There are also methods that allow you to specify which axis is used from each stick.

ArcadeDrive(stick) - Takes a joystick and controls the robot with arcade (single stick) steering using the y-axis of the joystick for forward/backward speed and the x-axis of the joystick for turns. There are also other methods that allow you to specify different joystick axes.

HolonomicDrive(magnitud e, direction, rotation) - Takes floating point values, the first two are a direction vector the robot should drive in. The third parameter, rotation, is the independent rate of rotation while the robot is driving. This is intended for robots with 4 Mecanum wheels independently controlled.

SetLeftRightMotorSpeeds (leftSpeed, rightSpeed) - Takes two values for the left and right motor speeds. As with all the other methods, this will control the motors as defined by the constructor.

Inverting the sense of some of the motors

Inverting the sense of some of the motors

It might turn out that some of the motors used in your RobotDrive object turn in the opposite direction. This often happens depending on the gearing of the motor and the rest of the drive

train. If this happens, you can use the SetInvertedMotor() method, as shown, to reverse a particular motor.

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 (4 or six wheels) must turn in the direction it needs to drive. A mecanum robot can move in any direction without first turning and is called a holonomic drive.

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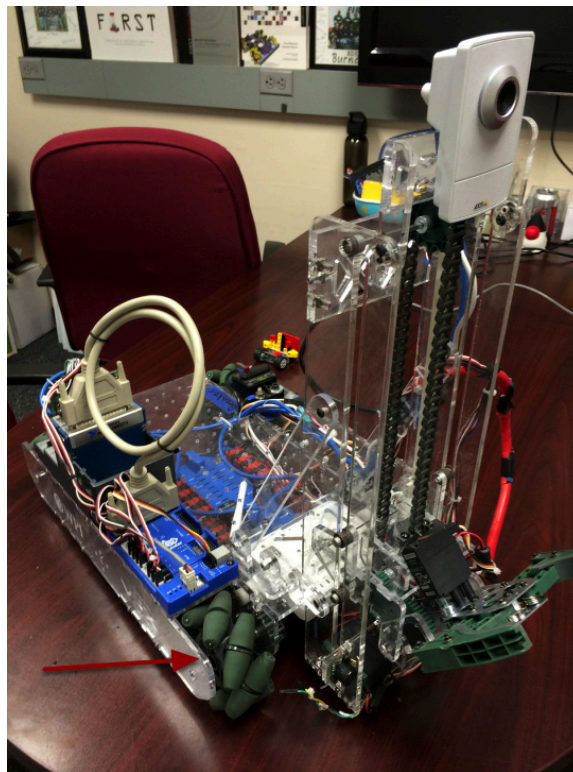
The wheels shown in this robot have rollers that cause the forces from driving to be applied at a 45 degree angle rather than straight forward as in the case of a conventional drive. You might guess that varying the speed of the wheels results in travel in any direction. You can look up how mecanum wheels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

Code for driving with mecanum wheels

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

```
    }
};
START_ROBOT_CLASS(MecanumDefaultCode);
```

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

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

Updating the program for field-oriented driving

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

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

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

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 (Motro 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

```
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

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

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

On/Off control of motors and other mechanisms - Relays

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

Relay connection overview

The cRIO provides the connections necessary to wire IFI spikes via the relay outputs on the digital breakout board. The breakout board provides a total of sixteen outputs, eight forward and eight reverse. The forward output signal is sent over the pin farthest from the edge of the breakout board, labeled as output A, while the reverse signal output is sent over the center pin, labeled output B. 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

```
exampleRelay = new Relay(1);
exampleRelay = new Relay(1, 2);
exampleRelay = new Relay(1, Relay.Direction.kForward);
exampleRelay = new Relay(1, 2, Relay.Direction.kReverse);


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 Compressor class is designed to operate any FRC supplied compressor on the robot. A **Compressor** object is constructed with 2 input/output ports:

• The Digital Relay output port connected to the Spike relay that controls the power to the compressor. (A digital output or Solenoid module port alone doesn't supply enough current to operate the compressor.)

• The Digital input port connected to the pressure switch that monitors the accumulator pressure.

The **Compressor** class will automatically create a task that runs in the background and twice a second turns the compressor on or off based on the pressure switch value. If the system pressure is above the high set point of the switch, the compressor turns off. If the pressure is below the low set point, the compressor turns on.

Starting the compressor

Starting the compressor

To use the Compressor class create an instance of the Compressor object and use the **Start()** method. This is typically done in the constructor for your Robot Program. Once started, it will continue to run on its own with no further programming necessary. If you do have an application where the compressor should be turned off, possibly during some particular phase of the game play, you can stop and restart the compressor using the **Stop()** and **Start()** methods.

The compressor class will create instances of the **DigitalInput** and **Relay** objects internally to read the pressure switch and operate the Spike relay.

Shown in the example is some C++ code that implements a compressor with a Spike relay connected to Relay port 2 and the pressure switch connected to digital input port 4. Both of these ports are connected to the primary digital input module.

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 solenoid breakout board on top of a NI 9472 Digital Sourcing module in the cRIO (slot 3). To use these solenoids 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 9472 module 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.

Note on port numbering

The port numbers on the Solenoid class range from 1-8 as printed on the Solenoid Breakout Board. The NI 9472 indicator lights are numbered 0-7 for the 8 ports, which is different numbering than used by the class or the Solenoid Breakout Board silk screen.

Single Solenoids in WPILib

```
exampleSolenoid = new Solenoid(1);
exampleSolenoid = new Solenoid(1, 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 or module 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

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

WPILib sensors

Using limit switches to control behavior

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

What values are provided by the limit switch

Limit switches can have "normally opened" or "normally closed" outputs. The usual way of wiring the switch is between a digital input signal connection and ground. The digital input has pull-up

WPILib programming

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

```java
 1    package edu.wpi.first.wpilibj.templates;
 2
 3    import edu.wpi.first.wpilibj.DigitalInput;
 4    import edu.wpi.first.wpilibj.SimpleRobot;
 5    import edu.wpi.first.wpilibj.Timer;
 6
 7    public class RobotTemplate extends SimpleRobot {
 8
 9        DigitalInput limitSwitch;
10
11        public void robotInit() {
12            limitSwitch = new DigitalInput(1);
13        }
14
15        public void operatorControl() {
16            // more code here
17            while (limitSwitch.get()) {      // wait for the limit switch
18                Timer.delay(10);
19            }
20            // more code here
21
22        }
23    }
```

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

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

public class ArmUp extends CommandBase {
```

WPILib programming

```
    public ArmUp() {
    }

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

    protected void execute() {
    }

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

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

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

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

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

Using a counter to detect the closing of the switch

```
package edu.wpi.first.wpilibj.templates.subsystems;
import edu.wpi.first.wpilibj.Counter;
import edu.wpi.first.wpilibj.DigitalInput;
import edu.wpi.first.wpilibj.SpeedController;
import edu.wpi.first.wpilibj.Victor;
```

WPILib programming

```java
import edu.wpi.first.wpilibj.command.Subsystem;
public class Arm extends Subsystem {

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

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

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

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

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

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

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

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

Create a command that uses the counter to detect switch closing

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

public class ArmUp extends CommandBase {

    public ArmUp() {
    }

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

    protected void execute() {
    }

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

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

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

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

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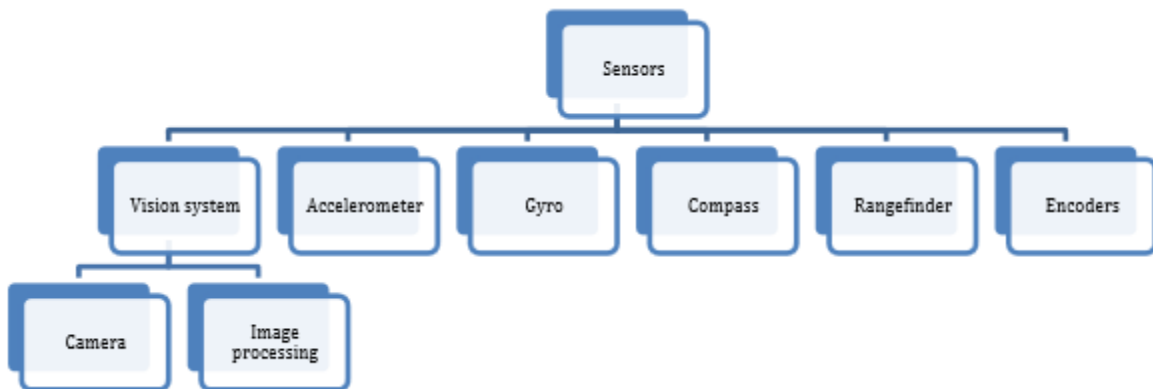
On the cRIO, the FPGA implements all the high speed measurements through dedicated hardware ensuring accurate measurements no matter how many sensors and motors are connected to the robot. This is an improvement over previous systems, which required complex real time software routines. The library natively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

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

Two-axis analog accelerometer

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

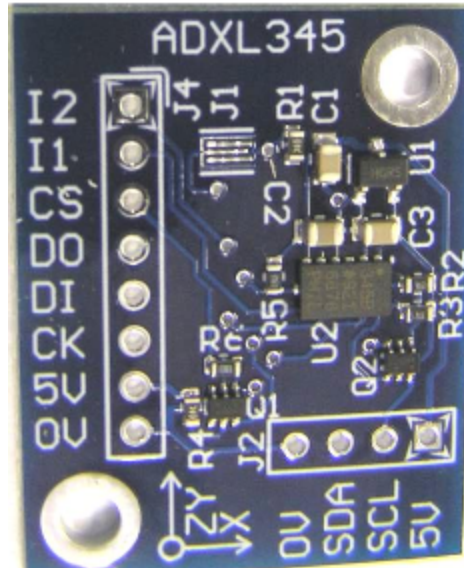## Analog Accelerometer code example

```java
public class AccelerometerSample extends SimpleRobot {
    Accelerometer accel;
    double acceleration;

    AccelerometerSample()
    {
        accel = new Accelerometer(1,1); //create accelerometer on analog input 1
        accel.setSensitivity(.018);   //Set sensitivity to 18mV/G (ADXL193)
        accel.setZero(2.5);         //Set zero to 2.5V (actual value would be determined experimentally)
    }

    /**
     * This function is called once each time the robot enters operator control.
     */
    public void operatorControl() {
        while(isOperatorControl() && isEnabled())
        {
            acceleration = accel.getAcceleration();
        }
    }
}
```

A brief code example is shown above which illustrates how to set up an analog accelerometer connected to analog module 1, 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).

WPILib 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.

WPILib programming

ADXL345 Code Example

```java
public class AccelerometerSample extends SimpleRobot {
    ADXL345_I2C accel;
    double accelerationX;
    double accelerationY;
    double accelerationZ;
    ADXL345_I2C.AllAxes accelerations;

    AccelerometerSample()
    {
        accel = new ADXL345_I2C(1, ADXL345_I2C.DataFormat_Range.k2G); //create accelerometer on module 1
    }

    /**
     * This function is called once each time the robot enters operator control.
     */
    public void operatorControl() {
        while(isOperatorControl() && isEnabled())
        {
            accelerationX = accel.getAcceleration(ADXL345_I2C.Axes.kX);
            accelerationY = accel.getAcceleration(ADXL345_I2C.Axes.kY);
            accelerationZ = accel.getAcceleration(ADXL345_I2C.Axes.kZ);

            accelerations = accel.getAccelerations();
            accelerationX = accelerations.XAxis;
        }
    }
}
```

A brief code example is shown above illustrating the use of the ADXL345 connected to the I2C bus on Digital Module 1. The accelerometer has been set to operate in +/- 2g mode. The example illustrates both the single axis and all axes methods of getting the sensor values, in practice select one or the other depending on whether you need a single axis or all three. SPI operation is similar, refer to the Javadoc/Doxygen for the ADXL345_SPI class for additional details on using the sensor over SPI.

Gyros to control 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.

Using the Gyro class
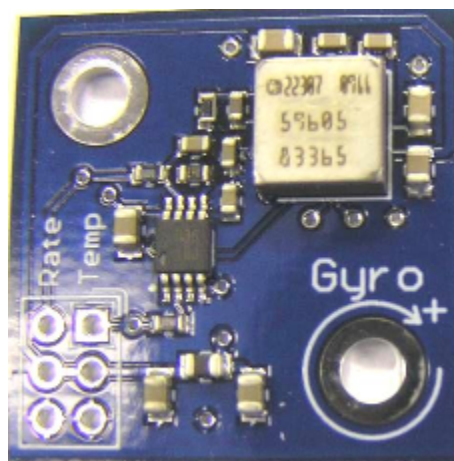
The Gyro object should be created in the constructor of the **RobotBase** derived object. When the Gyro object is used, it will go through a calibration period to measure the offset of the rate output

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 Gyro object.

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

Using a gyro to drive straight

Sample Java program for driving straight

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.

```java
package edu.wpi.first.wpilibj.templates;
import edu.wpi.first.wpilibj.Gyro;
import edu.wpi.first.wpilibj.RobotDrive;
import edu.wpi.first.wpilibj.SimpleRobot;
import edu.wpi.first.wpilibj.Timer;
public class GyroSample extends SimpleRobot {

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

 \   double Kp = 0.03;

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

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

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

Determine robot orientation with a compass

A compass uses the earth's magnetic field to determine the heading of the robot.

Placement Notes

This field is relatively weak causing the compass to be susceptible to interference from other magnetic fields such as those generated by the motors and electronics on your robot. If you decide to use a compass, be sure to mount it far away from interfering electronics and verify its accuracy.

HiTechnic Compass

WPILib directly supports one compass, the HiTechnic Compass. This part connects to the I2C port on the Digital Sidecar. It is important to note that there is only one I2C port on each of these modules.

Code Example

```
compass = new HiTechnicCompass(1);
heading = compass.getAngle();
```

The compass is constructed by passing in the Digital Module number that it is connected to. The current heading of the compass can then be retrieved by calling getAngle() in Java or GetAngle() in C++.

Measuring robot distance to a surface using Ultrasonic sensors

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
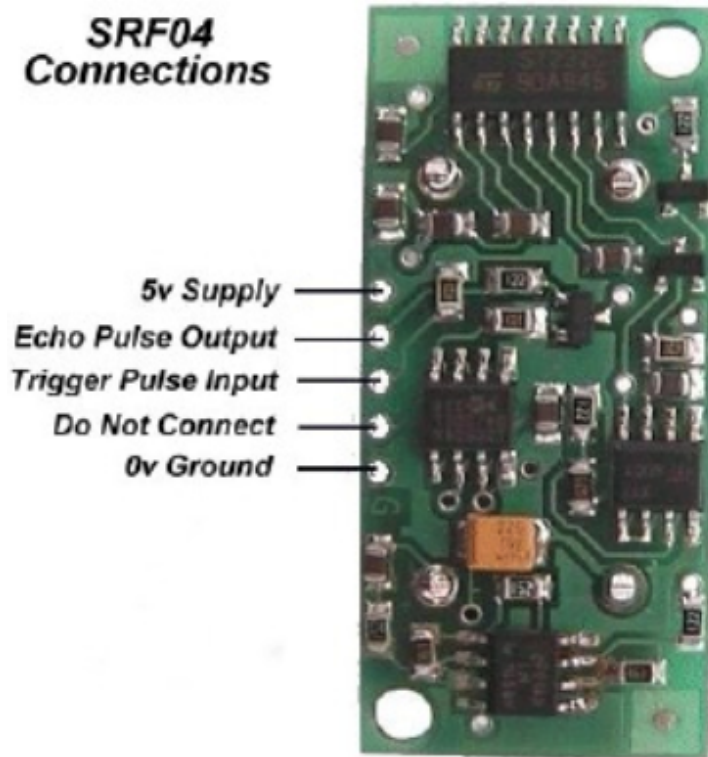
To aid in the sue of Ping-Response Ultrasonic sensors such as the Devantech SRF04 pictured above, WPILib contains an Ultrasonic class. This type of sensor has two transducers, a speaker that sends a burst of ultrasonic sound, and a microphone that listens for the sound to be reflected off of a nearby object. It requires two connections to the cRIO, 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

```
Ultrasonic ultra(ULTRASONIC_ECHO_PULSE_OUTPUT,
ULTRASONIC_TRIGGER_PULSE_INPUT);
```

Example 4:   C++ example of creating an ultrasonic rangefinder object

```
Ultrasonic ultra = new Ultrasonic(ULTRASONIC_ECHO_PULSE_OUTPUT,
                                  ULTRASONIC_TRIGGER_PULSE_INPUT);
```

Example 5:   Java example of creating an ultrasonic rangefinder object.

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.

Reading the distance

```
Ultrasonic ultra(ULTRASONIC_PING, ULTRASONIC_ECHO);
ultra.SetAutomaticMode(true);
int range = ultra.GetRangeInches();
```

Example 6:   C++ example of creating an ultrasonic sensor object in automatic mode and getting the range.

```
Ultrasonic ultra = new Ultrasonic(ULTRASONIC_PING, ULTRASONIC_ECHO);
ultra.setAutomaticMode(true);
int range = ultra.getRangeInches();
```

Example 7:   Java example of creating an ultrasonic sensor object in automatic mode and getting the range.

The following two examples read the range on an ultrasonic sensor connected to the output port ULTRASONIC_PING and the input port ULTRASONIC_ECHO.

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 Channela nd 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 cRIO 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.

Using Counters

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

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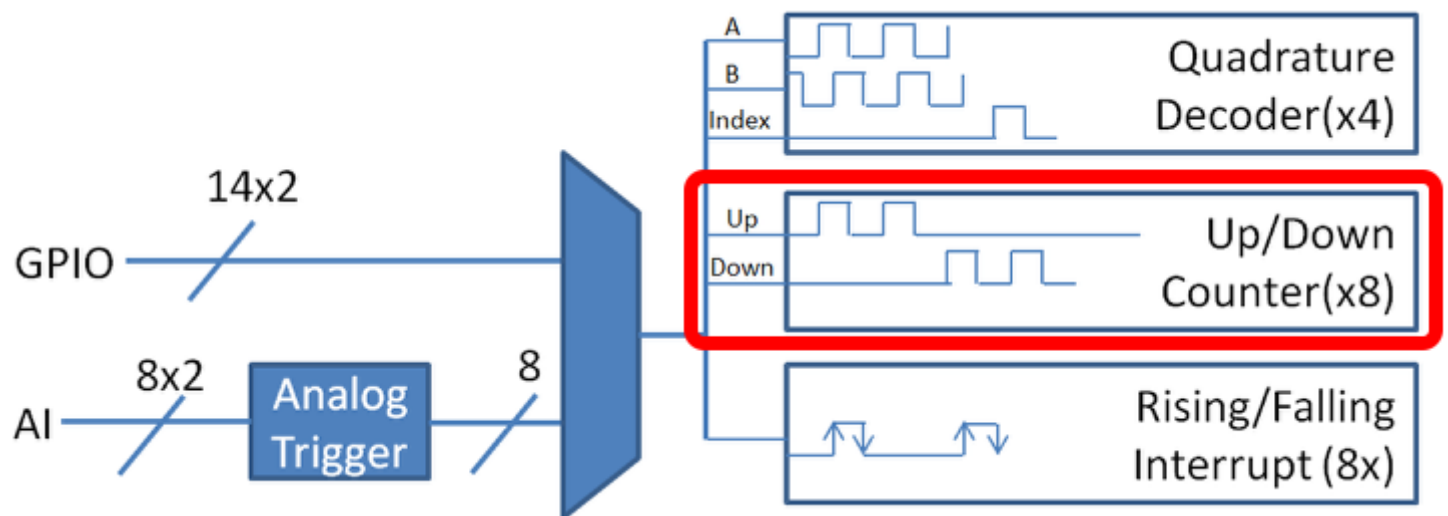
There are 8 Up/Down Counter units contained in the FPGA which can each operate in a number of modes based on the type of input signal:

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

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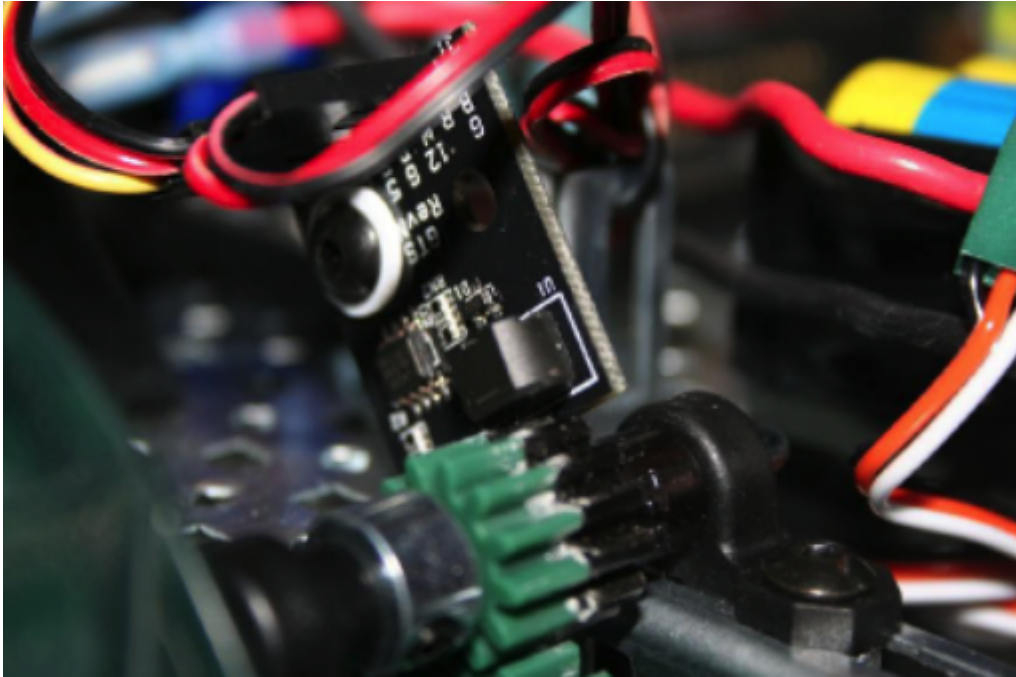
Gear-tooth sensors are designed to be mounted adjacent to spinning ferrous gear or sprocket teeth and detect whenever a tooth passes. The gear-tooth sensor is a Hall-effect device that uses a magnet and solid-state device that can measure changes in the field caused by the passing teeth. The picture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

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

Semi-Period mode

```
Counter exampleCounterHi = new Counter(1);
Counter exampleCounterLow = new Counter(3);
exampleCounterHi.start();
exampleCounterLow.start();
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 Measuring rotation of a wheel or other shaft using Encoders.

Normal mode

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

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

shown in the Semi-Period mode example is also perfectly valid for all modes of the counter including the Normal Mode.

Counter Settings

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

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

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

Starting, Stopping, and Resetting the counter

```
normalCounter.start();
normalCounter.stop();
normalCounter.reset();
```

Before a counter object will begin counting, the start() method must be called to start the counter. To stop the counter, call the stop() method. To reset the counter value to 0 call reset().

Getting Counter Values

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

The following values can be retrieved from the counter:

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

Measuring rotation of a wheel or other shaft using encoders

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

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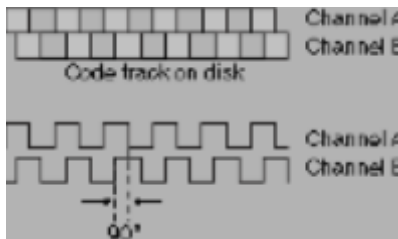
A quadrature encoder is a device for measuring shaft rotation that consists of two sensing elements 90 degrees out of phase. The most common type of encoder typically used in FRC is an optical encoder which uses one or more light sources (LEDs) pointed at a striped or slit code wheel and two detectors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

Encoders vs. Counters
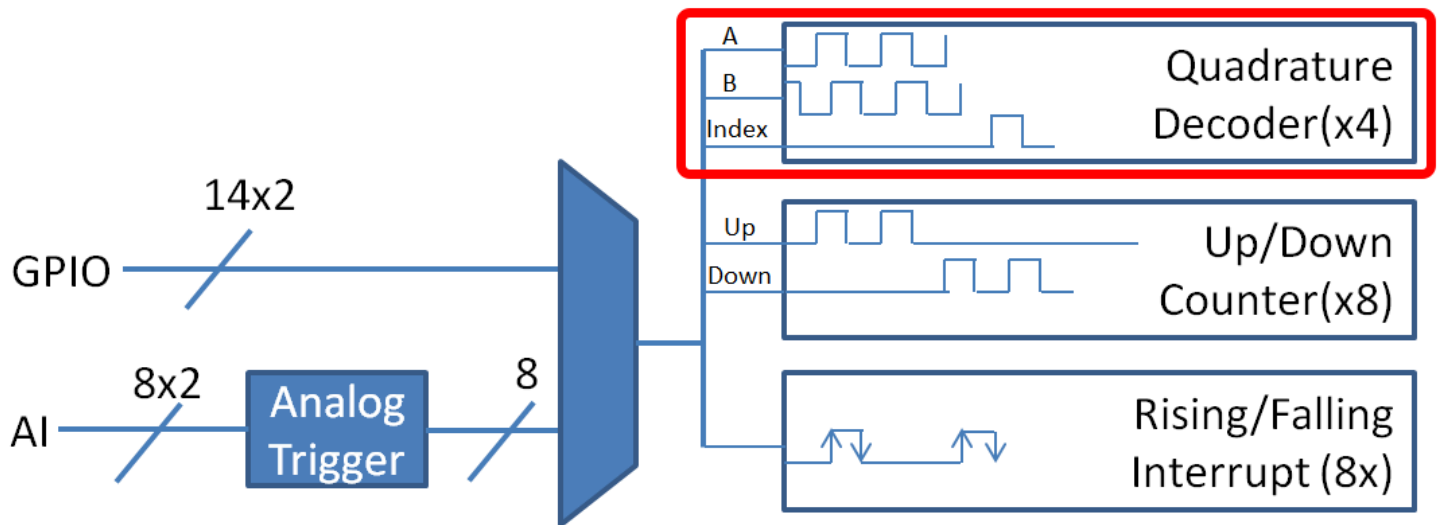
The FRC FPGA has 4 Quadrature decoder modules which can do 4x decoding of a 2 channel quadrature encoder signal. This means that the module is counting both the rising and falling edges of each pulse on each of the two channels to yield 4 ticks for every stripe on the codewheel. The quadrature decoder 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 4 quadrature decoding modules (used for

WPILib programming

4x decoding) and 8 counter modules (used for 1x and 2x decoding as well as Counter objects), depending on the number of encoders on your robot you may have to allocate the quadrature decoder modules to only the places where you need the most positional accuracy.

Constructing an Encoder object

```
Encoder encoder(1, 2, true, k4X);
```
Example 8:   C++ code creating an encoder on ports 1 and 2 with reverse sensing and 4X encoding.

```
Encoder encoder;
    encoder = new Encoder(1, 2, true, EncodingType.k4X);
```
Example 9:   Java code creating an encoder on ports 1 and 2 with reverse sensing and 4X encoding.

There are a number of constructors you may use to construct encoders, but the most common is shown above. In the example, 1 and 2 are the port numbers for the two digital inputs on the default module and true 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

```
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)

WPILib programming

- 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

```
sampleEncoder.start();
sampleEncoder.stop();
sampleEncoder.reset();
```

Before an encoder object will begin counting, the start() method must be called to start the encoder. To stop the encoder, call the stop() method. To reset the encoder value to 0 call reset().

Getting Encoder Values

```
int count = sampleEncoder.get();
int rawCount = sampleEncoder.getRaw();
double distance = sampleEncoder.getDistance();
double period = sampleEncoder.getPeriod();
double rate = sampleEncoder.getRate();
boolean direction = sampleEncoder.getDirection();
boolean stopped = sampleEncoder.getStopped();
```

The following values can be retrieved from the encoder:

- Count - The current count. May be reset by calling reset().
- Raw Count - The count without compensation for decoding scale factor.
- Distance - The current distance reading from the counter. This is the count multiplied by the Distance Per Count scale factor.
- Period - The current period of the counter in seconds. If the counter is stopped this value may return 0. This is deprecated, it is recommended to use rate instead.

WPILib programming

- 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 NI 9201 Analog to Digital module has a number of features not available on simpler controllers. It will automatically sample the analog channels in a round robin fashion, providing a combined sample rate of 500 ks/s (500,000 samples / second). These channels can be optionally oversampled and averaged to provide the value that is used by the program. There are raw integer and floating point voltage outputs available in addition to the averaged values. The diagram below outlines this process.

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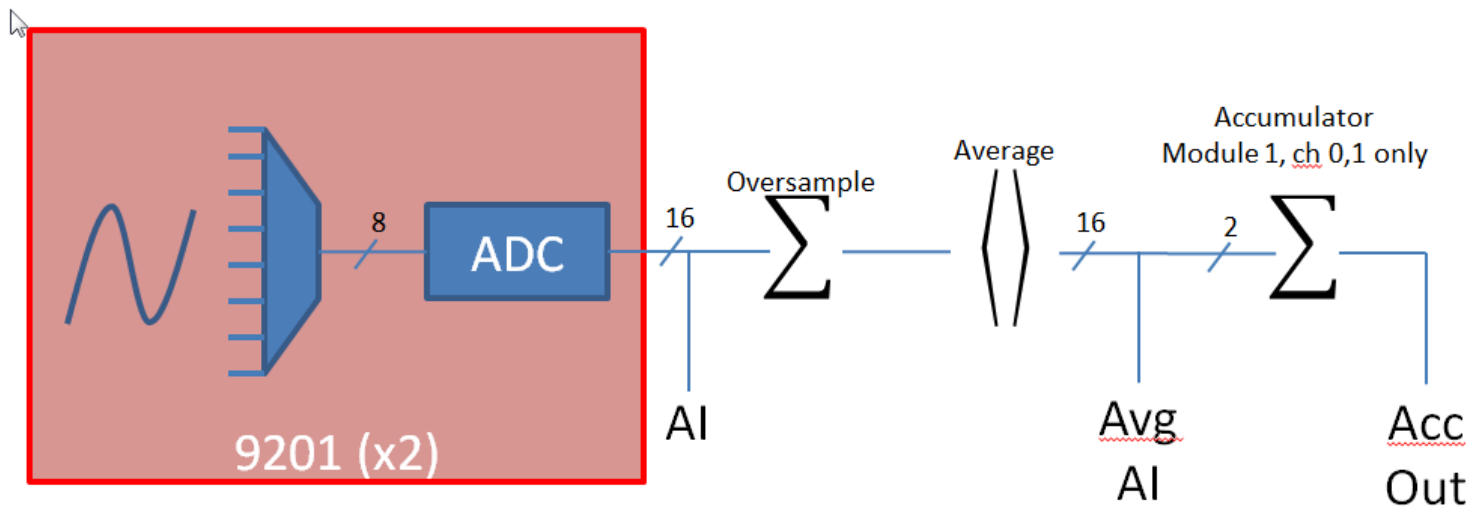
When the system averages a number of samples, the division results in a fractional part of the answer that is lost in producing the integer valued result. Oversampling is a technique where extra samples are summed, but not divided down to produce the average. Suppose the system were oversampling by 16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

Constructing an Analog Channel

Oversampling and Averaging

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

Where N is the number of Oversample bits

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

Where M is the number of Average bits

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

Where f_s is the original sampling frequency

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

Code example

```
int bits;
exampleAnalog.setOversampleBits(4);
bits = exampleAnalog.getOversampleBits();
exampleAnalog.setAverageBits(2);
bits = exampleAnalog.getAverageBits();
```

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

Sample Rate

```
exampleAnalog.getModule().setSampleRate(5000);
```

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 50,000 samples per channel per second (400kS/s total).

Reading Analog Values

```
1  int raw = exampleAnalog.getValue();
   double volts = exampleAnalog.getVoltage();  2
3  int averageRaw = exampleAnalog.getAverageValue();
   double averageVolts = exampleAnalog.getAverageVoltage();  4
```

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

1. Raw value - The instantaneous raw 12-bit (0-4096) value representing the full -10V to 10V range of the ADC. The typical 0V-5V swing will provide values in the approximate range 0-1024. 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 9201 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.
5. Accumulator - The purpose and use of the accumulator is discussed below.

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 tat reference to indicate direction of rotation.

Setting up an accumulator


```
1  exampleAnalog.setAccumulatorInitialValue(0);
   exampleAnalog.setAccumulatorCenter(512);    2
3  exampleAnalog.setAccumulatorDeadband(10);
   exampleAnalog.resetAccumulator();   4
```

There are two accumulators implemented in the FPGA, connected to channels 0 and 1 of Analog Module 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

```
1   long count = exampleAnalog.getAccumulatorCount();
    long value = exampleAnalog.getAccumulatorValue();   2
    AccumulatorResult result = new AccumulatorResult();
3   exampleAnalog.getAccumulatorOutput(result);
    count = result.count;
    value = result.value;
```

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

1. Count - The number of samples that have been added to the accumulator since the last reset.
2. Value - The value currently in the accumulator
3. Combined - Retrieve the count and value together to assure synchronization. This should be used if you are going to use the count and value in the same calculation such as averaging.

Potentiometers to measure joint angle or linear motion

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

Potentiometer Taper

The taper of a potentiometer describes the relationship between the position and the resistance. The two common tapers are linear and logarithmic. A linear taper potentiometer will vary the resistance proportionally to the rotation of the shaft; For example, the shaft will measure 50% of the resistave 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

WPILib does not contain an explicit class for using potentiometers, as an analog device code should use the Analog Channel class to interface with the potentiometer. Some teams choose to create a class in their code which extends Analog Channel which implements the scaling and offset operations which convert voltages to angles or other real world units used on the robot.

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

```
AnalogTrigger trigger = new AnalogTrigger(1);
AnalogTrigger trigger2 = new AnalogTrigger(1, 2);
AnalogTrigger trigger3 = new AnalogTrigger(channel);
```

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

Setting Analog Trigger Voltage Range

```
trigger.setLimitsRaw(2048, 3200);
trigger.setLimitsVoltage(0, 3.4);
```

The voltage range of the analog trigger can be set in either raw units (0 to 4096 representing -10V to 10V) 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

```
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

```
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 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.

Controlling the speed of a motor

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. 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 feedfoward term is multiplied by the setpoint for the PID controller so that it scales with the desired output speed.

Using PID controllers in command based robot programs

Using PID controllers in command based robot programs

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

reached the target. See the [Command based programming](#) section for more information and examples.

Driver Station Inputs and Feedback

Driver Station Input Overview

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

Input types
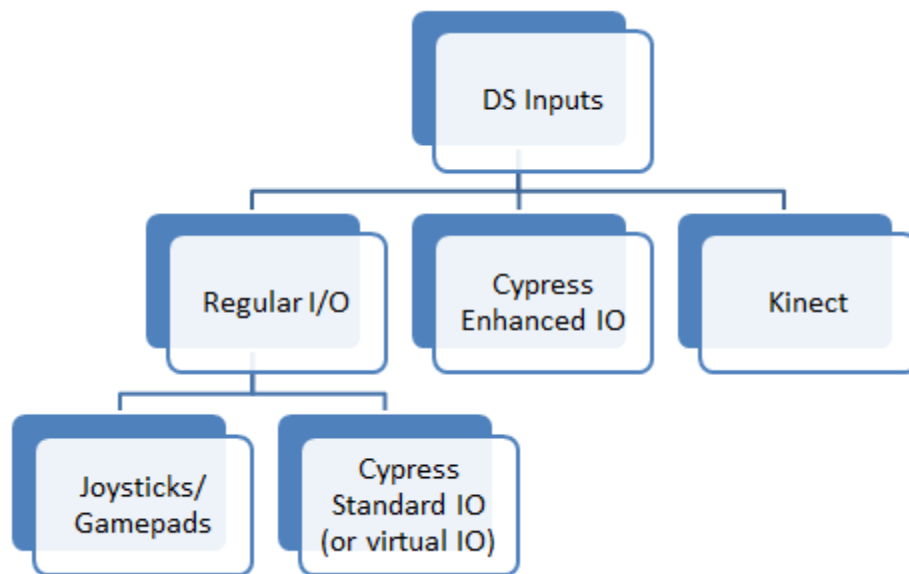
The chart above shows the different types of inputs that may be transmitted by the DS software. The most common input is an HID compliant joystick or gamepad such as the Logitech Attack 3 or Logitech Extreme 3D Pro joysticks which have been provided in the Kit of Parts since 2009. In addition to these devices teams can also use the Cypress FirstTouch board to design custom IO solutions such as buttons potentiometers or other custom input. This custom IO can then be accessed using either the standard IO methods of the Driver Station class or by using the Enhanced IO Class if additional customization or advanced features are required. 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 E-Stop Robotics CCI or the U-HID device.

WPILib programming

Driver Station Class

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

DriverStation.getInstance().someMethod();
```

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

Robot Mode

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

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

while(isOperatorControl() && isEnabled())
{

}
```

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 SimpleRobot base class. There are two separate pieces of information that define the current mode, the enable state (enabled/disabled) and the control state(autonomous, operator control, test). This means that exactly one method from the first group and one method from the second group should always return true. For example, if the Driver Station is currently set to Test mode and the robot is disabled the methods isDisabled() and isTest() would both return true.

WPILib programming

Battery Voltage

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

In order to report the robot battery voltage back to the Driver Station software the DriverStation class runs a task which is constantly measuring and updating the battery voltage using the Analog Breakout with the jumper installed on the 9201 module in slot 1 of the cRIO. 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

```
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

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

The getLocation() method of the Driver Station returns an integer indicating which station number the Driver Station is in (1-3). Note that the station that the DS and controls are located in is not typically related to the starting position of the robot so this information may be of limited use. When not connected to the FMS software this state is determined by the Team Station dropdown on the DS Operation tab.

WPILib programming

Team Number

```
int team;
team = DriverStation.getInstance().getTeamNumber();
```

The getTeamNumber method returns an integer indicating the FRC team number the Driver Station software is currently set to. One example of using this information would be to distinguish at runtime between multiple robots with identical code but different constants/tuning parameters.

Match Time

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

This method returns the approximate match time in seconds. Note that this time is derived by starting a timer at 0 when the enable signal is received for Autonomous and setting the timer to 15 seconds when the enable signal is received for Teleop. This is not an official time sent from the FMS. Another consequence of this is that if the controller reboots or disconnects from the DS during the match, then reconnects, this time will be incorrect.

Custom IO Methods

The DriverStation class also contains methods for accessing custom IO on the Cypress FirstTouch board in compatibility mode. If a Cypress board is not connected to the DS these inputs can be used as virtual IO and set with the keyboard and mouse inside the Driver Station software on the I/O tab. Additional information on accessing this data can be found in the Custom I/O article.

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 five analog and twelve digital inputs which allows for joysticks with more capabilities 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
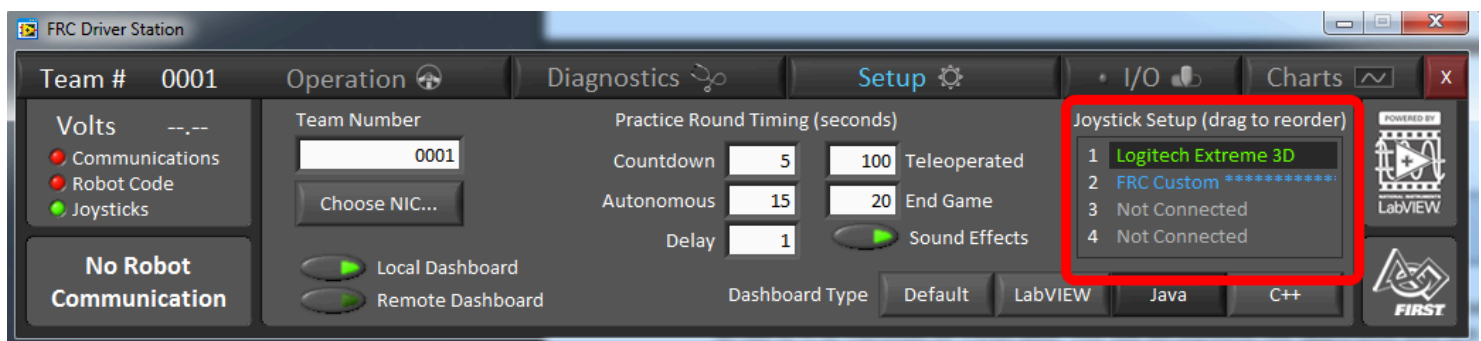
The joystick must be connected to one of the four available USB ports on the driver station. The startup routine will read whatever position the joysticks are in as the center position, therefore, when the station is turned on the joysticks must be at their center position. In general the Driver Station 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 Setup tab and viewing the order in the Joystick Setup box on the right 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.

New for 2014: The Driver Station will now show up to 6 devices in the Setup window. **The first 4 devices will be transmitted to the robot**. The additional devices are shown to allow teams to use

WPILib programming

one component of a composite device such as the TI Launchpad with FRC software without having to sacrifice one of the 4 transmitted devices.

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

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

The primary constructor for the Joystick class takes a single parameter representing the port number of the Joystick, this is the number (1-4) 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

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

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

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

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

Accessing Joystick Values - Option 2

```
double value;
value = exampleStick.getRawAxis(2);

boolean buttonValue;
buttonValue = exampleStick.getRawButton(1);
```

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
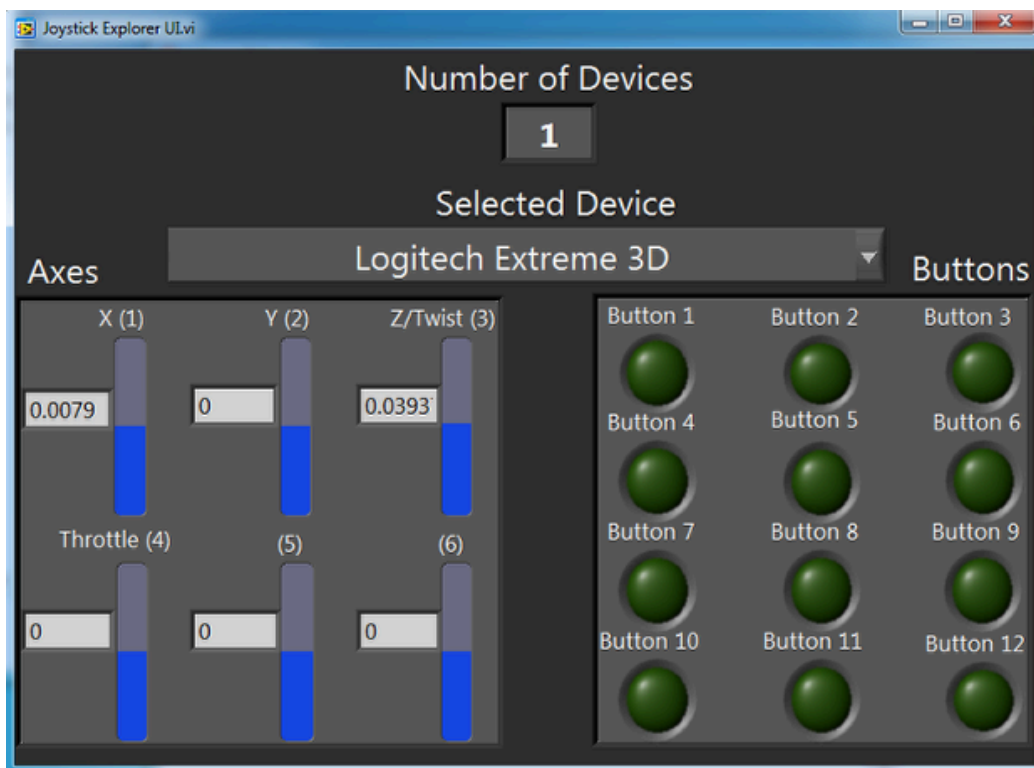
WPILib programming

buttons of your device and the appropriate channel number see the section "Determining Joystick Mapping" below.

Polar methods

```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

One way to determine joystick mapping is by writing robot code to display axis and button values via the dashboard or console, loading it on the robot, then testing the joystick. A simpler way is to

WPILib programming

download the [Joystick Explorer utility program from the WPILib project](#) which uses the same joystick code as the Driver's Station and displays the values of all 6 axes and 12 buttons. This program requires the LabVIEW 2012 runtime (any computer with the Driver Station installed will have it). Using this utility select your desired device from the drop-down menu then run through the physical axes and buttons on the joystick and note the corresponding channel number and range. Note that some features which may seem like buttons may actually show up as axes and that in some cases these features share an axis (X-Box controller triggers as an example).

Custom IO - Cypress FirstTouch Module

The Cypress FirstTouch IO module is a board that allows teams to interface to custom IO solutions such as potentiometers, buttons, switches, encoders, and much more. The methods used with the Cypress board in standard (compatible) mode may also be used to interface with virtual IO provided by the DS software if the Cypress board is not attached.

Programming the FirstTouch module

Before using the FirstTouch module the proper software must be loaded onto the board. For additional details see this article.

Configuring the mode
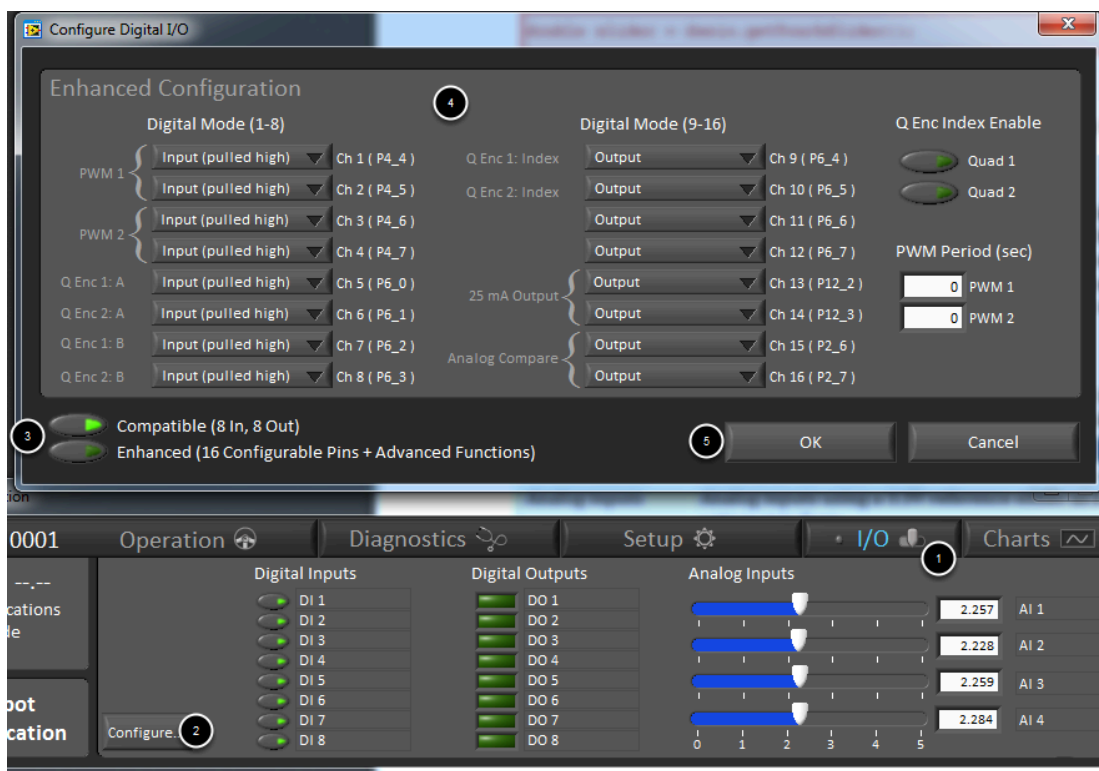
WPILib programming

The Cypress board can be set to one of two modes when used with the FRC Driver Station. Additionally, the function of each pin and another of other advanced features can be configured if the board is set to advanced mode. To set the mode of the board, click on the I/O tab of the Driver Station, then click the Configure button. Select Compatible or Enhanced at the bottom of the dialog, and configure the settings in the box above if using Advanced mode (these settings do not apply to Compatible mode), then click Ok.

Standard Mode

```
double analogValue;
boolean boolValue;
analogValue = DriverStation.getInstance().getAnalogIn(1);
boolValue = DriverStation.getInstance().getDigitalIn(6);
DriverStation.getInstance().setDigitalOut(3, boolValue);
```

When the Cypress Board is in standard/compatible IO mode the data it provides is accessed through the Driver Station class. Each of the three types of IO requires a parameter of the channel number and digital outputs require a value. Valid channels for the IO types are:

- Analog: 1-4
- Digital Input: 1-8
- Digital Output: 1-8

Virtual IO
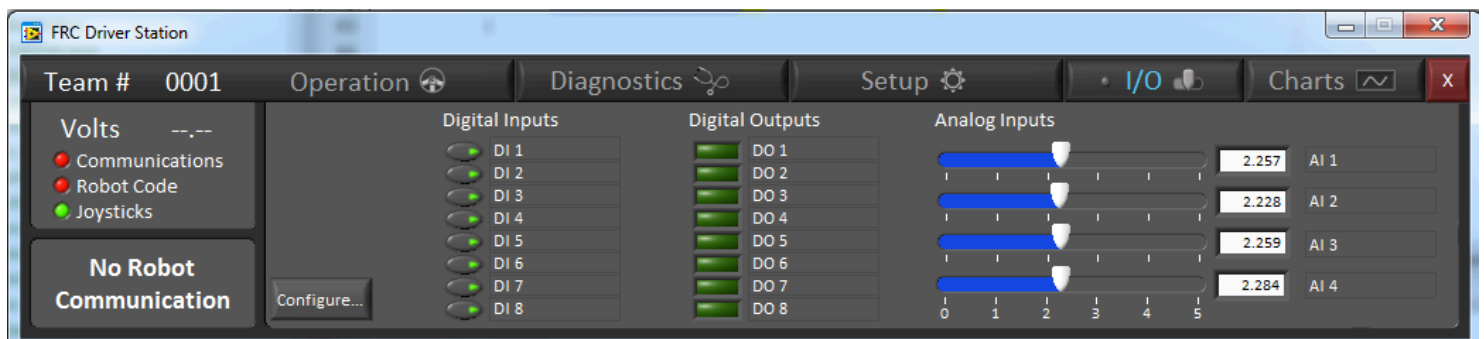
The same Driver Station methods can be used to interact with Virtual IO if no Cypress Board is present. On the IO tab of the Driver Station there are controls for each of the 8 Digital Inputs (click

to toggle), indicators for the 8 Digital Outputs, and controls for the 4 Analog Inputs (click and drag or type in the box to set).

Enhanced Mode

```
DriverStationEnhancedIO dseio;
dseio = DriverStation.getInstance().getEnhancedIO();
```

If the Cypress board is set to enhanced mode, you must use the DriverStationEnhancedIO class to set and retrieve values. This class has additional methods to handle configuration of the board in enhanced mode and getting and setting values associated with the advanced features such as PWM generation and quadrature decoding.

Configuration

The Enhanced IO configuration can be set either using the IO tab of the Driver Station as shown above or by using the methods in the DriverStationEnhancedIO class. Changes made in the Driver Station will persist across runs of the DS software by using a configuration file, but will not persist across different machines even if the same Cypress board is used. Configuration set in the robot code will override the configuration loaded from the file by the DS but may not override any changes that are made on the IO tab after the robot has linked to the DS.

WPILib programming

Data

Function	Information
Accelerometer	Returns acceleration in Gs
Analog inputs	Analog inputs using a 3.3V reference either as a voltage or in ratiometric form
Analog outputs	2 channels (either A01 or A02) of analog output. About 0-3V and about 100uA of drive current
Button state	Get the state of either the button on the board or 5 additional configurable buttons
LED output	Set the state of any of 8 LEDs
Digital input	Read digital input values for switches, etc.
Digital output	Set digital output lines
PWM outputs	2 pairs of I/O lines capable of PWM output. There is a PWM generator for each pair. Pairs have common periods but independent duty cycles.
Quadrature encoder input	There are 2 signed 16 bit 4X quadrature encoder inputs with optional index functionality.
Capacitive touch slider	There is a touch sensitive capacitive slider on the board. Returns value between 0-1 with -1 indicating no touch.

The Enhanced I/O module has a very powerful and expanded set of capabilities beyond just simple analog and digital I/O. The table above details the available options.

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 chapter](#).

Robot to driver station networking

Writing a simple NetworkTables program in C++ and Java with a Java client (PC side)

NetworkTables is an implementation of a distributed "dictionary". That is named values are created either on the robot, driver station, or potentially an attached coprocessor, and the values are automatically distributed to all the other participants. For example, a driver station laptop might receive camera images over the network, perform some vision processing algorithm, and come up with some values to sent back to the robot. The values might be an X, Y, and Distance. By writing these results to NetworkTable values called "X", "Y", and "Distance" they can be read by the robot shortly after being written. Then the robot can act upon them.

NetworkTables can be used by programs on the robot in either C++, Java or LabVIEW and is built into each version of WPILib.

Using NetworkTables from a Java robot program

Using NetworkTables from a Java robot program

NetworkTables programs on the robot are easiest to write. The program simply reads or writes values from within the program. The instance of NetworkTables is automatically created by the WPILib runtime system. This example is the simplest robot program that can be written that continuously writes pairs of values (X, and Y) to a table called "datatable". Whenever these values are written on the robot, they can be read shortly after on the desktop client.

1. The variable "table" is of type NetworkTable. NetworkTables are hierarchical, that is tables can be nested by using their names for representing the position in the hierarchy.
2. The table is associated with values within the hierarchy, in this case the path to the data is /datatable/X and /datatable/Y.
3. Values are written to the "datatable" NetworkTable. Each value will automatically be replicated between all the NetworkTable programs running on the network.

When this program is run on the robot and enabled in Teleop mode, it will start writing incrementing X and Y values continuously, updating them 4 times per second (every 0.25 seconds).

Using Network Tables from a C++ robot program

Using Network Tables from a C++ robot program

NetworkTables programs on the robot are easiest to write. The program simply reads or writes values from within the program. The instance of NetworkTables is automatically created by the WPILib runtime system. This example is the simplest robot program that can be written that continuously writes pairs of values (X, and Y) to a table called "datatable". Whenever these values are written on the robot, they can be read shortly after on the desktop client.

1. The variable "table" is of type NetworkTable. NetworkTables are hierarchical, that is tables can be nested by using their names for representing the position in the hierarchy.
2. The table is associated with values within the hierarchy, in this case the path to the data is /datatable/X and /datatable/Y.
3. Values are written to the "datatable" NetworkTable. Each value will automatically be replicated between all the NetworkTable programs running on the network.

When this program is run on the robot and enabled in Teleop mode, it will start writing incrementing X and Y values continuously, updating them 4 times per second (every 0.25 seconds).

Using the client version of NetworkTables on a desktop computer

Using the client version of NetworkTables on a desktop computer

The NetworkTables libraries are built into all versions of robot-side WPILib. You can set values from the robot in C++, Java or LabVIEW with simple put and get methods. To use it on a laptop (usually the driver station computer), there are several options:

1. a client library that you can reference from Java programs that you write.
2. from plugins that you write for the SmartDashboard (it's included there)

The Java library is part of the NetBeans Java plugin installation and can be found in the <user-directory>/sunspotfrcsdk/desktop-lib directory as shown here.

For C++ WindRiver installations the .jar files are located in the C:\WindRiver\WPILib\ desktop-lib directory.

Setting up NetBeans to create the client-side (laptop/desktop computer) program

Setting up NetBeans to create the client-side (laptop/desktop computer) program

To write a program that runs on your PC that uses NetworkTables the Java project must reference the JAR file from the NetBeans installation shown above. The project has to reference the networktables-desktop.jar file. This is an example of doing it with NetBeans but any IDE will have a way of adding .JAR files to a project. In this example the .jar file was added to the project properties.

Note: this is not necessary for a robot program since NetworkTables is built into WPILib. You simply have to add the necessary java import statements or C++ #includes for the NetworkTable classes that are used in the program.

The client (laptop) side of the program

The client (laptop) side of the program

This program is the simplest program that you can write on a PC to use NetworkTables. It continuously reads the values from robot example in the previous step.

1. Set NetworkTables to client mode (not on the robot) and specifiy the IP address of the robot.
2. Create a NetworkTable variable ("table") that is associated with the "datatable" NetworkTable.
3. Loop continuously and sleep for 1 second each time through the loop.
4. Read the X and Y values from the /datatable NetworkTable that was written on the robot in the previous program and print the values. The program output is shown below.

Program output from the simple client example

Program output from the simple client example

This output is from the NetBeans "output" window. This is the results from the System.out.println() method from the previous program that is running on a desktop computer retrieving values written on the robot from the earlier Robot program.

Viewing the NetworkTables variables in TableViewer

Viewing the NetworkTables variables in TableViewer

There is a diagnostic tool called TableViewer that will display the current state of the NetworkTables table. In this case, running it will show the current values of all the variables in the variables created in this example are shown in the red box above. TableViewer is located in the sunspotfrcsdk folder for NetBeans intstalls or in the C:\WindRiver\Workbench\WPILib folder for C++ installs.

Receiving notifications of changes to a NetworkTable

Receiving notifications of changes to a NetworkTable

A PC or Robot program can receive notifications of changes to a NetworkTable. This example is a client-side (PC) program, but the same concepts will work on a robot program. These notifications are received asynchonously as the new values are received by the NetworkTable library.

1. Connect to the NetworkTable server using the same technique as in the previous example.
2. Register this class as a ITableListener. Changes to the "datatable" will be reported to this class through the "valueChanged" callback method (below)
3. Sleep for 100 seconds while values are reported. The program could do anything here, but in this simple example, it only waits for 100 seconds while waiting for values to arrive.
4. This valueChanged method is called whenever there are changes or additions to the NetworkTable "datatable". The boolean value bln will be true if this is a new value or false if it is just an update to a previously reported variable. The Object is the new value that has been received. The output from this program is shown in the next step.

Results of running the client-side (PC) TableListener example

Results of running the client-side (PC) TableListener example

In this screen image the values returned from the TableListener example are shown. Notice that at the top of the output X and Y are returned with their respective values and "true" for the boolean

---

value. This indicates that they are new values. In all the other cases, the boolean value is "false" indicating that it is just an update to a previously reported value.

Using NetworkTables with RoboRealm

Using NetworkTables with RoboRealm

RoboRealm is a program that does client-side (PC) vision processing. RoboRealm can connect to a camera on a robot and do real-time tracking of field targets and sending the results back to the robot. In the past this required writing custom networking code for the PC to robot communications. RoboRealm now has a built-in NetworkTables client and this allows the RoboRealm program to send values directly back to the robot via some shared variables.

For further information see: http://www.roborealm.com/help/Network_Tables.php

Using TableViewer to see NetworkTable values

TableViewer is a program to help debug NetworkTables applications. It acts as a NetworkTables client and allows the viewing of all the keys and associated values in a tabular format. You can use this to quickly see the value of a variable or set a value for a variable. This is a java program making it platform independent - it can run anywhere that the java runtime is installed.

Starting TableViewer

Starting TableViewer

TableViewer is a java application and is distributed as a .jar file. It is named with the version number, so the actual name you'll see will be dependent on the version of the build. It should be located in the tools directory in either the C++ or Java installation. In the case of C++ it will be in C:\WindRiver\WPILib and in Java it will be in *<user-home-directory>*/sunspotfrcsdk/tools, where the *<user-home-directory>* is the operating system installed users home directory. On some operating systems this can be started by simply double-clicking on the TableViewer.jar file using a file browser. On other systems it might have to be explicitly run from a command line by entering, "java -jar TableViewer.jar". The TableViewer application

Once it is running, enter the host IP address of the robot. This is the FRC standard IP naming convention, 10.TE.AM.2 where TE.AM are replaced with the team number. For example it would be 10.1.90.2 for Team 190.

Viewing Table Values

Viewing Table Values

The TableViewer will start up and show all the keys (variable names) and values for those keys. In addition it shows a sequence number which is an internal NetworkTables field used to determine

if values are updated and need refreshing. The sequence number increments every time the value of a NetworkTable variable changes. The values wrap around at 65535.

The table rows can be sorted by either the Key, Value or Sequence Number by clicking on the column heading in the table.

Using NetworkTables with RoboRealm

RoboRealm is a desktop vision application that you run on your driver station and can connect to a camera on your robot, do a set of vision processing steps that you define, then send the results back to the robot using NetworkTables. Using RoboRealm is easy since you don't need a robot to try it. In fact, you can write programs with just images that were taken such as those that come with any of the three language distributions. For Java and C++, installing the 2014 Sample Vision program will include a bunch of pictures taken with an Axis camera of the actual field that you can use to make sure your vision algorithm works.

There is a card included with your kit of parts that contains instructions for getting RoboRealm.

The idea is that you create a sequence of image processing steps with RoboRealm that create the results in variables. Then send those variables to the robot using NetworkTables. The robot gets the results and uses them to control the robot behavior such as aiming, driving to a target, setting shooter speed, etc.

WPILib programming

Creating the RoboRealm program
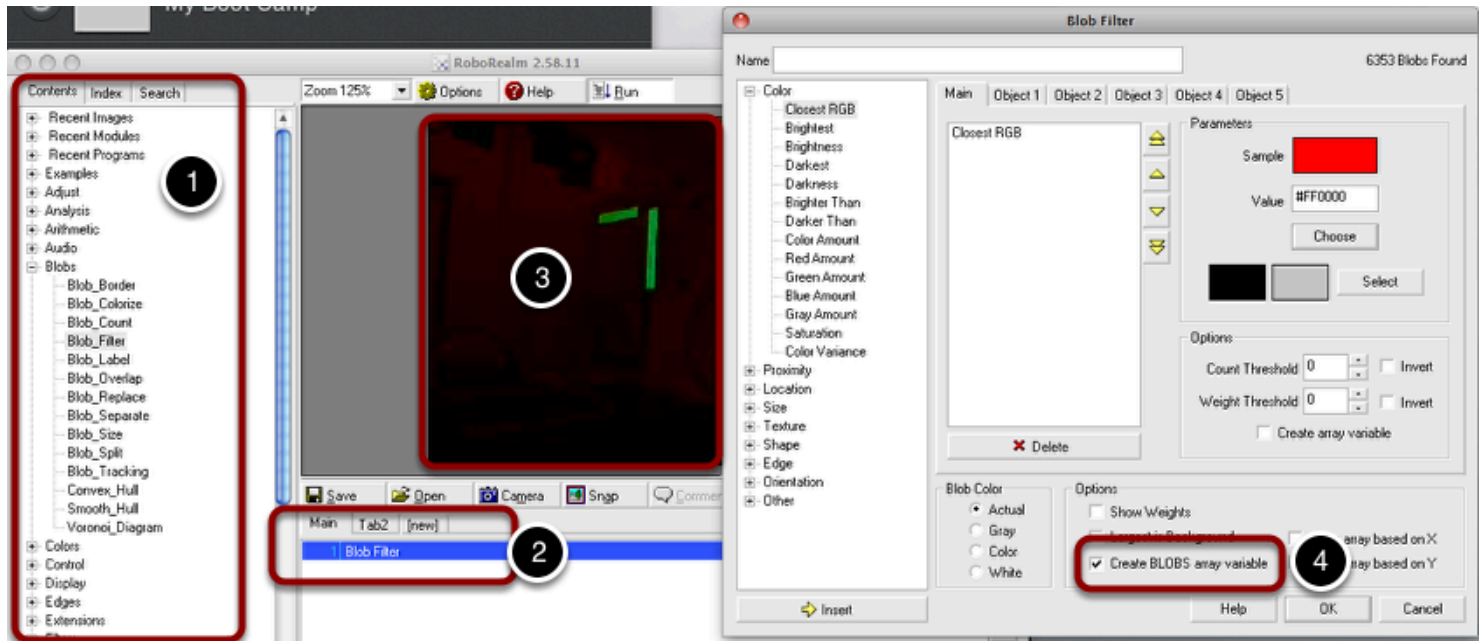
Create the RoboRealm program using the image processing elements shown in the "Contents" tab (1) on the left side of the interface. The program will appear in the bottom window (2). You can drag images from Windows Explorer into the Image window (3) to make sure that your algorithm works well and is repeatable for all the positions that the robot might be in.

By clicking on any of the steps in the program window, you can see the result of that processing step to make sure it's doing what you expect. Many of the steps can create variables with the results of that step. For example, the Blob Filter step shown has a checkbox to create an array of information for the detected blobs (4).

WPILib programming

Interfacing with NetworkTables
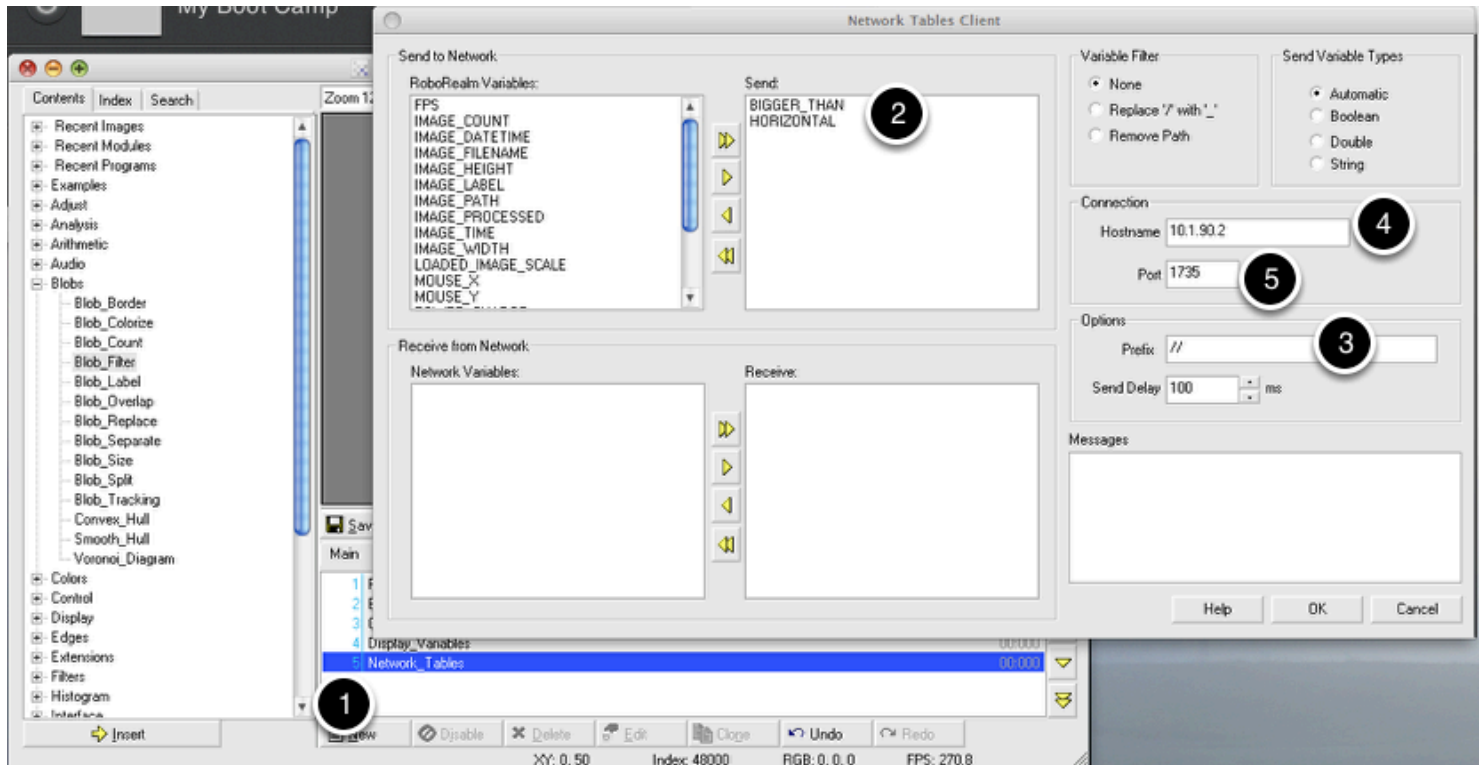
Variables that are created in RoboRealm can be sent back to the robot using NetworkTables. To do that add a "Network_Tables" step (1) to your RoboRealm program and add the variables you defined to be sent to the robot (2). Set the "Prefix" to "//" (3) and the "Hostname" to the value of your robot network address (4) and the port to 1735 (5).

In this case it will send an array of values back to the robot for each of the detected blobs.

Retrieving the values on the robot

```java
5    import edu.wpi.first.wpilibj.Victor;
6    import edu.wpi.first.wpilibj.networktables.NetworkTable;
7    import edu.wpi.first.wpilibj.networktables2.type.NumberArray;
8
9    public class RobotTemplate extends SimpleRobot {
10
11       NetworkTable table;  1
12       Victor claw;
13
14       public void robotInit() {
15           table = NetworkTable.getTable("");   2
16           claw = new Victor(3);
17       }
18
19       public void setClaw(double value) {
20           claw.set(value);
21           Timer.delay(1);
22           claw.set(0);
23       }
24
25       public void operatorControl() {
26
27           boolean isHot = false;
28
29           while (isOperatorControl() && isEnabled()) {
30               NumberArray values = new NumberArray();   3
31               try {
32                   table.retrieveValue("HORIZONTAL", values);
33            4      if (values.size() > 0 != isHot) {
34                       isHot = values.size() > 0;
35                       setClaw(isHot ? -1 : 1);
36                   }
37               } catch (Exception e) {
38                   e.printStackTrace();
39               }
40               Timer.delay(0.1);
41           }
42       }
43   }
```

Here's a Java program that retrieves those values on the robot and opens or closes a claw (just as a test) depending on whether there is at least one element in the array of values sent back. The steps to make this work are:

1. Declare the NetworkTable that will contain the values
2. Get an instance of the table and make sure everything is initialized
3. Create a new NumberArray object that will hold the results
4. Retrieve the array elements (values) for the NetworkTable variable called "HORIZONTAL" that was exported by RoboRealm

Now whenever the size of the array changes and contains at least one element the method setClaw(-1) will be called. Whenever the size changes and there are no elements in the array, setClaw(1) is called.