C/C++ and Java WPILib Programming Guide for the FIRST Robotics Competition

Worcester Polytechnic Institute Robotics Resource Center

Brad Miller, Ken Streeter, Beth Finn, Jerry Morrison, Dan Jones, Ryan O'Meara,
Derek White, Stephanie Hoag

Rev 1.0

November, 2009

Contents

What is the WPI Robotics Library

The WPI Robotics library is a set of classes that interfaces to the hardware in the FRC control system and your robot. There are classes to handle sensors, motors, the driver station, and a number of other utility functions like timing and field management.

The library is designed to:

- Deal with all the low level interfacing to these components so you can concentrate on solving this year's "robot problem". This is a philosophical decision to let you focus on the higher-level design of your robot rather than deal with the details of the processor and the operating system.
- Understand everything at all levels by making the full source code of the library available. You can study (and modify) the algorithms used by the gyro class for oversampling and integration of the input signal or just ask the class for the current robot heading. You can work at any level.

First, something about our environment. The robot controller for 2009 and later is a National Instruments cRIO-9074 real-time controller, or "cRIO" for short. It provides about 500x more memory. Most of the high-speed sensor-interrupt logic that typically requires precise coding, hand optimization and lots of bugs has been replaced with dedicated hardware (FPGA). When the user requests the number of ticks on a 1000 pulse/revolution optical encoder it just asks the FPGA for the value. Another example is A/D sampling that used to be done with tight loops waiting for the conversions to finish. Now sampling across 16 channels is done in hardware.

There are two choices of text-based languages available, C++ and Java. Those were chosen because we felt they represented a better level of abstraction for robot programs than C which was used in the past. C++ and Java (when used properly) also encourage a level of software reuse that is not as easy or obvious in C. At all levels in the library, we have attempted to design it for maximum extensibility.

There are classes that support all the sensors, speed controllers, driver station, etc. that will be in the kit of parts. In addition most of the commonly used sensors that we could find that are not traditionally in the kit are also supported, like ultrasonic rangefinders. Another example is several robot classes that provide starting points for teams to implement their own robot code. These classes have methods that are called as the program transitions through the various phases of the match. One class looks like the old easyC/WPILib model with Autonomous and OperatorControl functions that get filled in and called at the right time. Another is closer to the old IFI default where user supplied methods are called continuously, but with much finer control. And the base class for all of these is available for teams wanting to implement their own versions.

Even with the class library, we anticipate that teams will have custom hardware or other devices that we haven't considered. For them we have implemented a generalized set of hardware and software to make this easy. For example there are general-purpose counters than count any input either in the up direction, down direction, or both (with two inputs). They can measure the number of pulses, the width of the pulses and number of other parameters. The counters can also count the number of times an analog signal reaches inside or goes outside of a set of voltage limits. And all of this without requiring any of that high speed interrupt processing that's been so troublesome in the past. And this is just the counters. There are many more generalized features implemented in the hardware and software.

We also have interrupt processing available where interrupts are routed to functions in your code. They are dispatched at task level and not as kernel interrupt handlers. This is to help reduce many of the real-time bugs that have been at the root of so many issues in our programs in the past. We believe this works because of the extensive FPGA hardware support.

We have chosen to not use the C++ exception handling mechanism, although it is available to teams for their programs. Our reasoning has been that uncaught exceptions will unwind the entire call stack and cause the whole robot program to quit. That didn't seem like a good idea in a finals match in the Championship when some bad value causes the entire robot to stop.

The objects that represent each of the sensors are dynamically allocated. We have no way of knowing how many encoders, motors, or other things a team might put on a robot. For the hardware an internal reservation system is used so that people don't accidentally reuse the same ports for different purposes (although there is a way around it if that was what you meant to do).
There are also classes to assist in use of the camera and with image processing that is often used in FRC games. The classes wrap lower level functions that are part of the NI Vision library that is also used in the LabVIEW implementation of WPILib.

We can't say that our library represents the only "right" way to implement FRC robot programs. There are a lot of smart people on teams with lots of experience doing robot programming. We welcome their input; in fact we expect their input to help make this better as a community effort. To this end all of the source code for the library will be published on a server. We are in the process of setting up a mechanism where teams can contribute back to the library. And we are hoping to set up a repository for teams to share their own work. This is too big for a few people to have exclusive control; we want this software to be developed as a true open source project like Linux or Apache.

We believe that the objet oriented programming paradigm best fits robot programming with WPILib, but C programming is available for those with C experience. Here is an example of using C with the WPI Robotics Library.

C++

The following C program demonstrates driving the robot for 2 seconds forward during the Autonomous period and driving with arcade-style joystick steering during the Operator Control period. Notice that constants define the port numbers used in the program. This is a good practice and should be used for C and C++ programs.

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

static const UINT32 LEFT_MOTOR_PORT = 1;
static const UINT32 RIGHT_MOTOR_PORT = 2;
static const UINT32 JOYSTICK_PORT = 1;

void Initialize(void)
{
	CreateRobotDrive(LEFT_MOTOR_PORT, RIGHT_MOTOR_PORT);
	SetWatchdogExpiration(0.1);
}

void Autonomous(void)
{
	SetWatchdogEnabled(false);
	Drive(0.5, 0.0);
	Wait(2.0);
	Drive(0.0, 0.0);
}

void OperatorControl(void)
{
	SetWatchdogEnabled(true);
	while (IsOperatorControl())
	{
		WatchdogFeed();
		ArcadeDrive(JOYSTICK_PORT);
	}
}

START_ROBOT_CLASS(SimpleCRobot);
```

Choosing between C++ and Java

C++ and Java are very similar languages; in fact Java has its roots in C++ when it was designed. If you looked at a C++ or Java program from a distance, it would be hard to tell them apart. You'll find that if you can write a WPILib C++ program for your robot, then you can probably also write a Java program.

Language differences

There is a good detailed list of the differences between the two languages on Wikipedia available here: http://en.wikipedia.org/wiki/Comparison_of_Java_and_C++. The following is a summary of those differences as they will most likely effect robot programs created with WPILib.

- C++ memory is allocated and freed manually, that is the programmer is required to allocate objects and delete them. In Java objects are allocated the same way (through the use of the new operator), but it is freed automatically when there are no more references to them. This greatly simplifies memory management for the programmer.
- Java does not have pointers, only references to objects. All objects must be allocated with the new operator and are always referenced using the dot (.) operator, for example gyro.getAngle(). In C++ there are pointers, references, and local instances of objects.
- C++ uses header files and a preprocessor for including declarations in parts of the program where they are needed. In Java this happens automatically when the program is built by looking at the modules containing the references.
- C++ implements multiple inheritance where a class can be derived from several other classes combining the behavior of all of the base classes. In Java only single inheritance is supported, but interfaces are added to Java to get most of the benefits that multiple inheritance would provide without the complications.
- Java programs will check for array subscripts out of bounds, uninitialized references to objects and a number of other runtime errors that might occur a program under development.
- C++ programs will have the highest performance on the platform since it compiles to machine code for the power pc processor in the cRIO. Java on the other hand compiles to byte code for a virtual machine and is interpreted.

WPILib differences

We made every attempt to make the transition between C++ and Java as easy as possible in either direction. All the classes and methods have the same names. There are some differences that are brought on by the differences in the languages or the language conventions.

Item	**C++**	**Java**
Method naming convention	Methods are named with an upper case first letter and then camel case after that, for example, GetDistance().	Methods are named with a lower case first letter then camel case thereafter, for example getDistance().

Utility functions	Simply call functions for each of these functions, for example Wait(1.0) will wait for one second.	Java has no functions outside of classes so all library functions are implemented as methods, for example Timer.delay(1.0) will wait for one second.

Our version of Java

The Java virtual machine and implementation we are using is the Squawk platform based on the Java ME (micro edition) platform. Java ME is simplified version of Java designed for the limitations of embedded devices (like the cRIO). As a result there are no user interface classes or other classes that aren't meaningful in this environment. If you've done any Java programming in the past it was probably with the Java Standard Edition (SE). Some of the differences between SE and ME are:

- Dynamic class loading is not support class loading or unloading. Reflection, a way of manipulating classes while the program is running is also not supported.
- The Java compiler generates a series of byte codes for the virtual machine. When you create a program for the cRIO a step is automatically run after the program compiles called pre-verification. This pre-verification pass simplifies the program loading process and reduces the size of the Java virtual machine (JVM).
- Finalization is not implemented which means the system will not automatically call the finalize() method of an unreferenced object. If you need to be sure code runs when a class if no longer referenced, you must explicitly call a method that cleans up.
- Java Native Interface (JNI) is not supported. This is a method of calling C programs from Java using a standard technique. The JVM does support a similar way of calling to C code called JNA. There is more information about JNA in a later section of this document.
- Serialization and Remote Method Invocation (RMI) are not supported.
- The user interface APIs (Swing and AWT) are not implemented.
- Threads are supported by thread groups are not.
- Furthermore, since Java ME is based on an earlier version of Java SE (1.3), it doesn't include newer features such as Gernerics, Annotations and Autoboxing.

Sensors

The WPI Robotics Library includes built in support for all the sensors that are supplied in the FRC kit of parts as well as many other commonly used sensors available to *FIRST* teams through industrial and hobby robotics outlets.

Types of supported sensors

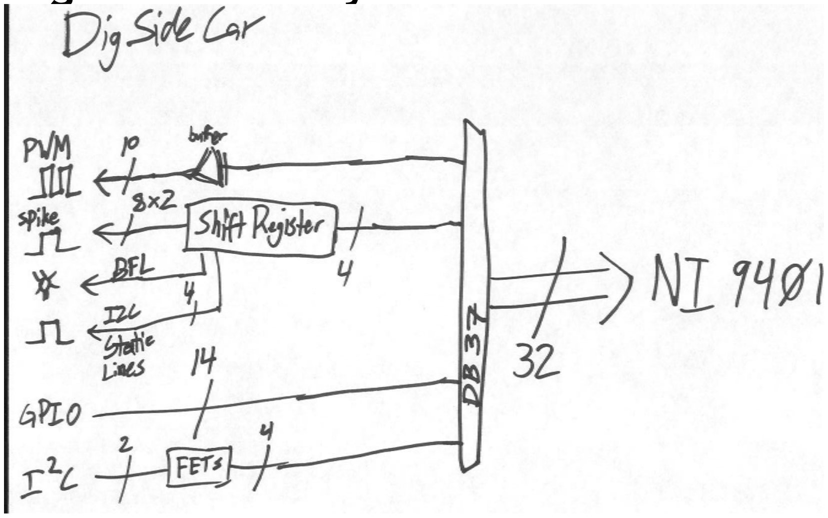
The library natively supports sensors of a number of categories shown below.

Category	**Supported sensors**
Wheel/motor position measurement	Gear-tooth sensors, encoders, analog encoders, and potentiometers
Robot orientation	Compass, gyro, accelerometer, ultrasonic rangefinder
Generic pulse output	Counters

In the past, high speed counting of pulses from encoders or accurate timing of ultrasonic rangefinders was implemented in complex real-time software and caused a number of problems as system complexity increased. 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 added to the robot.

In addition there are many features in the WPI Robotics Library that make it easy to implement many other types of sensors not directly supported with 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.

Digital I/O Subsystem



The NI 9401 digital I/O module has 32 GPIO lines. Through the circuits in the digital breakout board these map into 10 PWM outputs, 8 Relay outputs for driving Spike relays, the signal light, an I2C port, and 14 bidirectional GPIO lines.

The basic update rate of the **PWM lines** is a multiple of approximately 5 ms. Jaguar speed controllers update at slightly over 5ms, Victors update 2X (slightly over 10ms), and servos update at 4X (slightly over 20ms).

Digital Inputs

Digital inputs are typically used for switches. The DigitalInput object is typically used to get the current state of the corresponding hardware line: 0 or 1. More complex uses of digital inputs such as encoders or counters are handled by using the appropriate classes. Using these other supported device types (encoder, ultrasonic rangefinder, gear tooth sensor, etc.) doesn't require a digital input object to be created.

The digital input lines are shared from the 14 GPIO lines on each Digital Breakout Board. Creating an instance of a DigitalInput object will automatically set the direction of the line to input.

Digital input lines have pull-up resistors so an unconnected input will naturally be high. If a switch is connected to the digital input it should connect to ground when closed. The open state of the switch will be 1 and the closed state will be 0.

In Java, digital input values are true and false. So an open switch is true and a closed switch is false.

Digital Outputs

Digital outputs are typically used to run indicators or interface with other electronics. The digital outputs share the 14 GPIO lines on each Digital Breakout Board. Creating an instance of a Digital-Output object will automatically set the direction of the GPIO line to output. In C++ digital output values are 0 and 1 representing high (5V) and low (0v) signals. In Java the digital output values are true (5V) and false (0v).

Accelerometer

The accelerometer typically provided in the kit of parts is a two-axis accelerometer. It can provide acceleration data in the X and Y axes relative to the circuit board. In the WPI Robotics Library you treat it as two devices, one for the X axis and the other for the Y axis. This is to get better performance if your application only needs to use one axis. The accelerometer can be used as a tilt sensor – actually measuring the acceleration of gravity. In this case, turning the device on the side would indicate 1000 milliGs or one G.

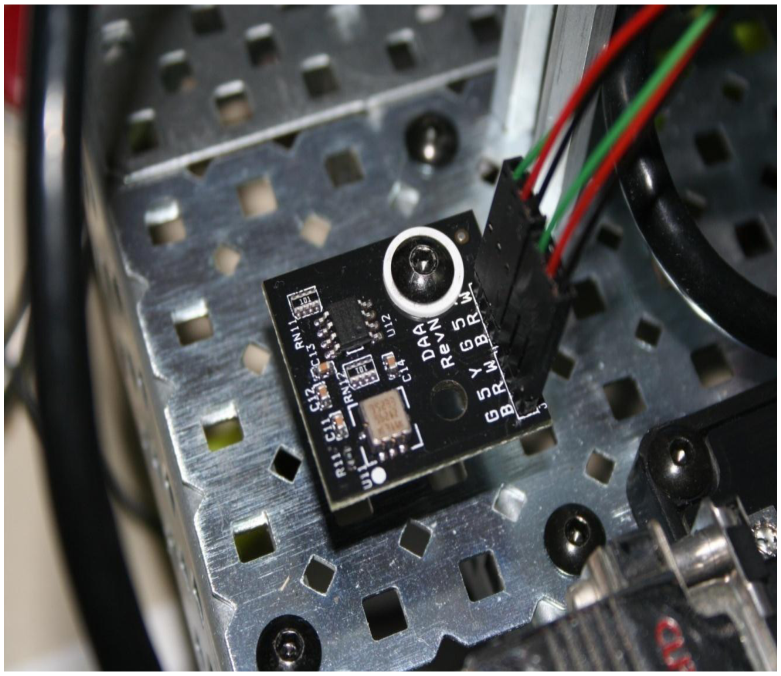


Figure 1: FRC supplied 2 -axis accelerometer board connected to a robot

Gyro

Gyros typically supplied by FIRST in the 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 normal to the gyro chip top package surface. 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 since a much larger range of values is 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 exactly matched the fastest rate of rotation your robot would ever experience. That would yield the highest accuracy possible, provided the robot never exceeded that range.

Using the Gyro class

The Gyro object is typically created in the constructor of the **`RobotBase`** derived object. When the Gyro object is instantiated it will go through a calibration period to measure the offset of the rate output while the robot is at rest. This requires that the robot be stationary while this is happening and that the gyro is unusable until after the calibration has completed.

Once initialized, the **`GetAngle()`**`(`**`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.

Setting the gyro sensitivity

The Gyro class defaults to the settings required for the 80°/sec gyro that was delivered by FIRST in the 2008 kit of parts.

To change gyro types call the **`SetSensitivity(float sensitivity)`** `(`**`setSensitivity(double sensitivity)`** `in Java)` method and pass it the sensitivity in volts/°/sec. Just be careful since the units are typically specified in mV (volts / 1000) in the spec sheets. A sensitivity of 12.5 mV/°/sec would require a **`SetSensitivity()`** (**`setSensitivity()`** in Java) parameter value of 0.0125.

C++

Example

This program causes 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 that varies either positive or negative degrees as the robot deviates from its initial heading. As long as the robot continues to go straight the heading will be zero. If the robot were to turn from the 0 degree heading, the gyro would indicate this with either a positive or negative value. This example uses the gyro to turn the robot back on course using the turn parameter of the Drive method.

```
class GyroSample : public SimpleRobot
{
	RobotDrive myRobot; // robot drive system
	Gyro gyro;
	static const float Kp = 0.03;

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

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

C++ Example 1: Driving in a straight line using a gyro

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

Java

```
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()
    {
        getWatchdog().setExpiration(0.1);
    }

    protected void Autonomous() {
        gyro.reset();
        while (isAutonomous()) {
            getWatchdog().feed();
            double angle = gyro.getAngle();   // get heading
            myRobot.drive(-1.0, -angle * Kp); // turn correct heading
            Timer.delay(0.004);
        }
        myRobot.drive(0.0, 0.0);    // stop robot
    }
};
```

Java Example 1: Driving in a straight line using a gyro

The angle is multiplied by 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 (maybe never getting back to the desired heading). This is proportional control.

HiTechnicCompass

Use of the compass is somewhat tricky since it uses the earth's magnetic field to determine the heading. The field is relatively weak and the compass can easily develop errors from other stronger magnetic fields from the motors and electronics on your robot. If you do decide to use a compass, be sure to locate it far away from interfering electronics and verify the readings on different headings.
Each digital I/O module has only one I2C port to connect a sensor to.

For example, let's create a compass on the I2C port of the digital module plugged into slot 4.

```
HiTechnicCompass compass(4);
compVal = compass.GetAngle();
```

C++ Example 2: Creating an instance of an I2C compass in C++

Ultrasonic rangefinder

The WPI Robotics library supports the common Daventech SRF04 or Vex ultrasonic sensor. This 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 uses 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.

Figure 2: SRF04 Ultrasonic Rangefinder connections

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 ports it is connect to in the constructor:

C++

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

C++ Example 3: Creating an Ultrasonic rangefinder instance

Java

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

Java Example 2: Creating an instance of an Ultrasonic rangefinder instance

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.
For ultrasonic rangefinders that do not have these connections don't use the Ultrasonic class. Instead use the appropriate class for the sensor, for example an AnalogChannel object for an ultrasonic sensor that returns the range as a voltage.

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

C++

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

C++ Example 4: Getting the range from an Ultrasonic rangefinder instance

Java

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

Java Example 3: Getting the range from an Ultrasonic rangefinder instance

Counter Subsystem

The counters subsystem represent a very complete set of digital signal measurement tools for interfacing with many sensors that you can put on the robot. There are several parts to the counter subsystem.

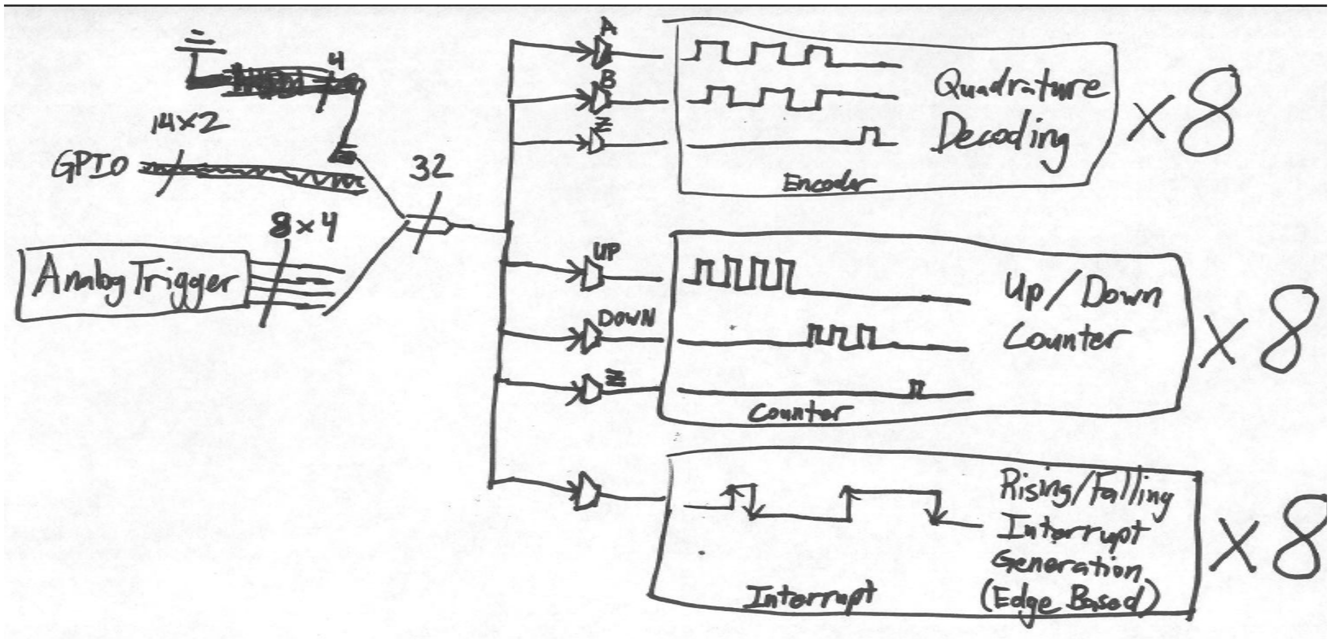


Figure 3: Schematic of the possible sources and counters in the Counter Subsystem in the cRIO.

Counters can be triggered by either Analog Triggers or Digital Inputs. The trigger source can either control up/down counters (Counter objects), quadrature encoders (Encoder objects), or interrupt generation.

Analog triggers count each time an analog signal goes outside or inside of a set range of voltages.

Counter Objects

Counter objects are extremely flexible elements that can count input from either a digital input signal or an analog trigger. They can also operate in a number of modes based on the type of input signal – some of which are used to implement other sensors in the WPI Robotics Library.

- Gear-tooth mode – enables up/down counting based on the width of an input pulse. This is used to implement the GearTooth object with direction sensing.
- Semi-period mode – counts the period of a portion of the input signal. This is used to measure the time of flight of the echo pulse in an ultrasonic sensor.
- Normal mode – can count edges of a signal in either up counting or down counting directions based on the input selected.

Encoders

Encoders are devices for measuring the rotation of a spinning shaft. Encoders are typically used to measure the distance a wheel has turned that can be translated into robot distance across the floor. Distance moved over a measured period of time represents the speed of the robot and is another common measurement for encoders. There are several types of encoders supported in WPILi

Simple encoders **Counter class**	Single output encoders that provide a state change as the wheel is turned. With a single output there is no way of detecting the direction of rotation. The Innovation First VEX encoder and the index outputs of a quadrature encoder are examples of this type of device.
Quadrature encoders **Encoder class**	Quadrature encoders have two outputs typically referred to as the A channel and the B channel. The B channel is out of phase from the A channel. By measuring the relationship between the two inputs the software can determine the direction of rotation. The system looks for Rising Edge signals (ones where the input is transitioning from 0 to 1) and falling edge signals. When a rising edge is detected on the A channel, the B channel determines the direction. If the encoder was turning clockwise, the B channel would be a low value and if the encoder was turning counter-clockwise then the B channel would be a high value. The direction of rotation determines which rising edge of the A channel is detected, the left edge or the right edge. The Quadrature encoder class can look at all edges and give an oversampled output with 4x accuracy.
Gear tooth sensor **GearTooth class**	This is a device supplied by FIRST as part of the FRC robot standard kit of parts. The gear tooth sensor is designed to monitor the rotation of a sprocket or gear that is part of a drive system. It uses a Hall-effect device to sense the teeth of the sprocket as they move past the sensor.

Table 1: Encoder types that are supported by WPILib

These types of encoders are described in the following sections.

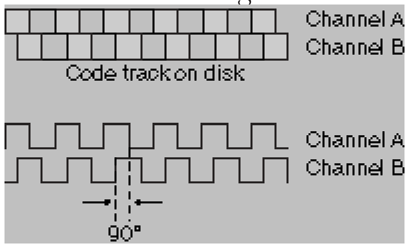


Figure 4: Quadrature encoder phase relationships between the two channels.

Example

Geartooth Sensor

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.

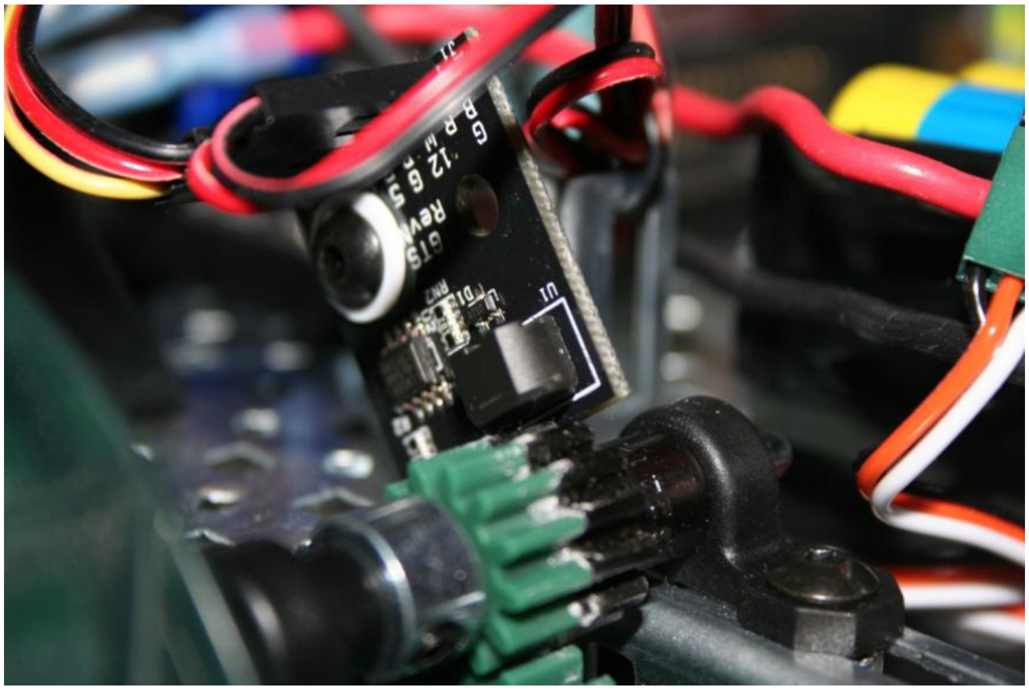


Figure 5: A gear tooth sensor mounted on a VEX robot chassis measuring a metal gear rotation. Notice that there is a metal gear attached to the plastic gear in this picture. The gear tooth sensor needs a ferrous material passing by it to detect rotation.

Example

Quadrature Encoders

Background Information

Encoders are devices for measuring the rotation of a spinning shaft. Encoders are typically used to measure the distance a wheel has turned that can be translated into robot distance across the floor. Distance moved over a measured period of time represents the speed of the robot and is another common measurement for encoders.

Encoders typically have a rotating disk with slots that spins in front of a photodetector. As the slots pass the detector, pulses are generated on the output. The rate at which the slots pass the detector indicates the rotational speed of the shaft and the number of slots that have passed the detector indicates the number of rotations (or distance).

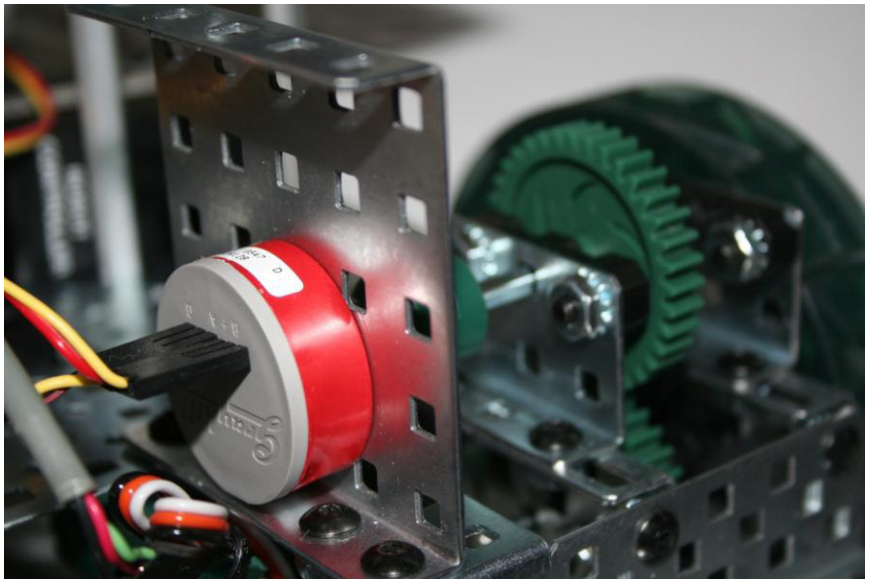


Figure 6: A Grayhill quadrature optical encoder. Note the two connectors, one for the A channel and one for the B channel.

Some quadrature encoders have an extra index channel. This channel pulses once for each complete revolution of the encoder shaft. If counting the index channel is required for the application it can be done by connecting that channel to a simple Counter object which has no direction information.

Quadrature encoders are handled by the Encoder class. Using a quadrature encoder is done by simply connecting the A and B channels to two digital I/O ports and assigning them in the constructor for Encoder.

There are four QuadratureEncoder modules in the FPGA and 8 Counter modules that can operate as quadrature encoders. One of the differences between the encoder and counter hardware is that encoders can give an oversampled 4X count using all 4 edges of the input signal. Counters can either return a 1X or 2X result based on one of the input signals. If 1X or 2X is chosen in the Encoder constructor a Counter module is used with lower oversampling and if 4X (default) is chosen, then one of the four encoders is used.

C++

```
Encoder encoder(1, 2, true, k4X);
```

C++ Example 5: Creating an encoder on ports 1 and 2 with reverse sensing and 4X encoding.

Java

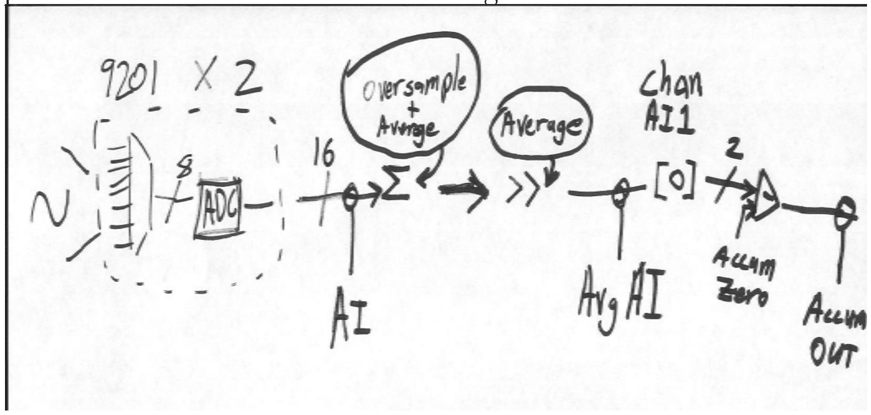
```
Encoder encoder;
encoder = new Encoder(1, 2, true, EncodingType.k4X);
```

Java Example 4: Creating an encoder on ports 1 and 2 with reverse sensing and 4X encoding.

Where **1** and **2** are the port numbers for the two digital inputs 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. To get the 4X value you should use the GetRaw() method on the encoder. The Get() method will always return the normalized value by dividing the actual count obtained by the 1X, 2X, or 4X multiplier.

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 an aggregate 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 **averaged value** is computed by summing a specified number of samples and performing a simple average, that is, dividing by the number of samples that are in the average. 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. That fraction represents how close the average values were to the next higher integer. **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.

Oversample & Average Engine

$$AvgAI = \frac{\sum_{0}^{2^M-1}\left(\sum_{0}^{2^N-1}(AI_x)\right)}{2^M}$$

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

N= oversample bits
M= average bits

To set the number of oversampled and averaged values use the methods:

C++

```
void SetAverageBits(UINT32 bits);
UINT32 GetAverageBits();
void SetOversampleBits(UINT32 bits);
UINT32 GetOversampleBits();
```

C++ Example 6: Methods used for setting up oversampling and averaging.

Java

```
void setAverageBits(int bits);
UINT32 getAverageBits();
void setOversampleBits(UINT32 bits);
UINT32 getOversampleBits();
```

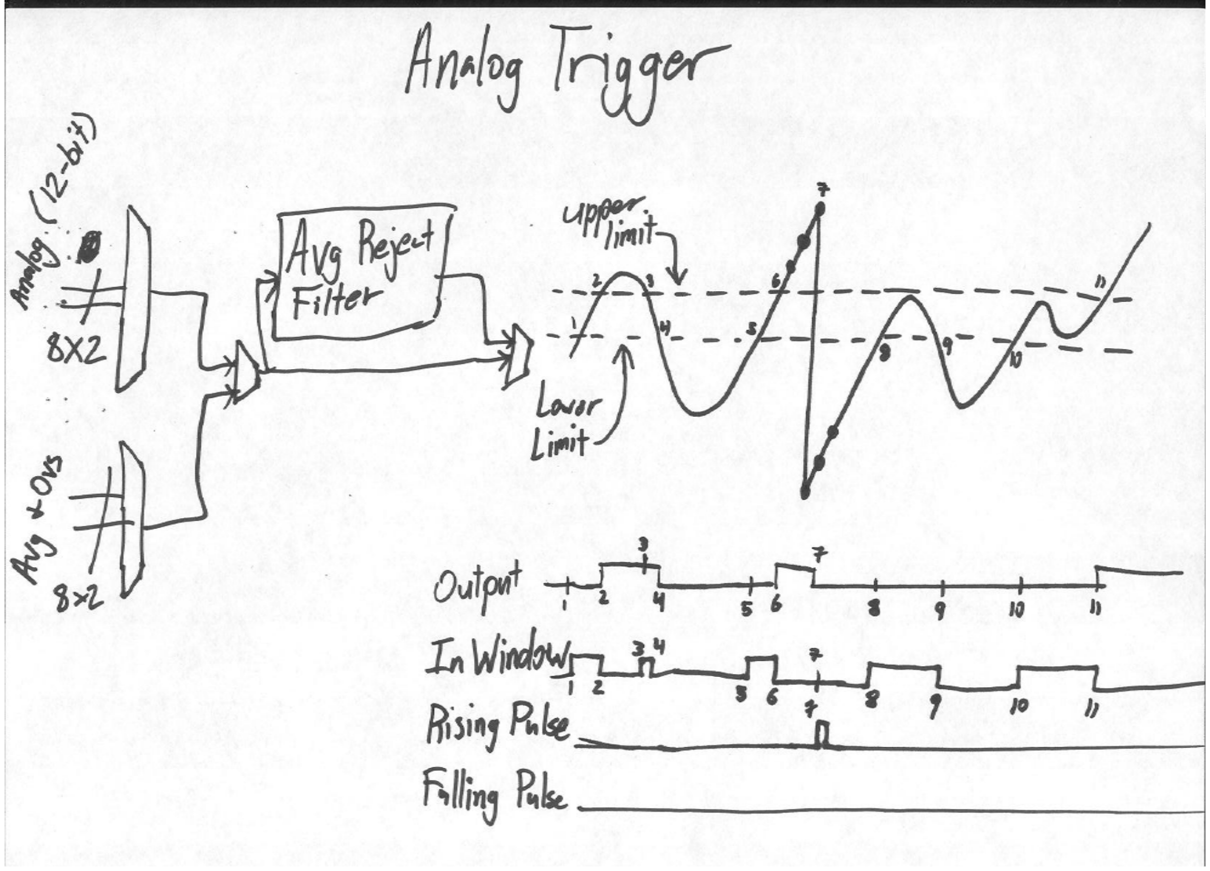
Java Example 5: Methods used for setting up oversampling and averaging

The number of averaged and oversampled values are always powers of 2 (number of bits of oversampling/averaging). Therefore the number of oversampled or averaged values is 2bits, 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 24+2, or 64.
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 WPI Robotics Library will allow the sample rate to be changed once for a module. Changing it to a different value will result in a runtime error being generated. 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.

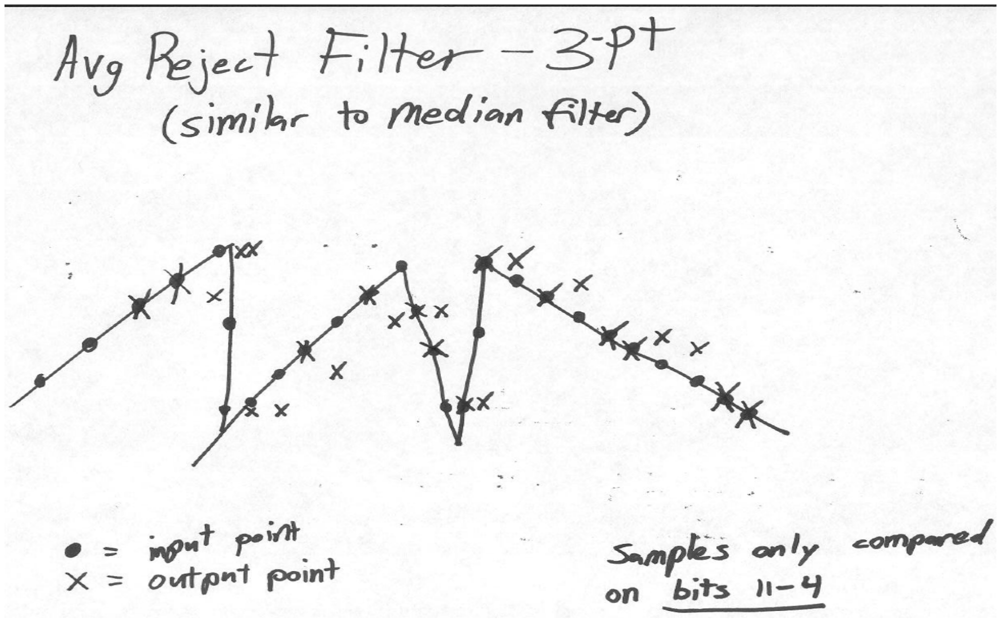
Summary

- There is one sample rate per module.
- The number of oversampled and averaged values is expressed as a power of 2.
- The delivered sample rate is reduced by the oversample and average values.
- There are 2 accumulators connected to analog channels 1 and 2 of the first Analog Module. This means that only two devices (such as gyros) that use the accumulators can be connected to the cRIO, and they must be connected to channel 1 or 2 of Analog Module 1.
- The returned analog value is 2n times larger than the actual value where n is the number of oversampled bits. Averaging doesn't change the returned values, except to average them.

Analog Triggers



3 Point Average Reject Filter


Avg Reject Filter – 3-pt
(similar to median filter)
● = input point
X = output point
Samples only compared on bits 11–4

Camera

Note: the camera and image processing classes will be changing for 2010. The version documented here corresponds to the older 2009 library. The Java library is upgraded, but does not correspond to this documentation. The C++ library has not yet been upgraded.

The camera provided in the 2009 kit is the Axis 206. The C camera API provides initialization, control and image acquisition functionality. Image appearance properties are configured when the camera is started. Camera sensor properties can be configured with a separate call to the camera, which should occur before camera startup. The API also provides a way to update sensor properties using a text file on the cRIO. PcVideoServer.cpp provides a C++ API that serves images to the dashboard running on a PC. There is a sample dashboard application as part of the LabVIEW distribution that can interface with C and C++ programs.

Camera task management

A stand-alone task, called *FRC_Camera*, is responsible for initializing the camera and acquiring images. It continuously runs alongside your program acquiring images. It needs to be started in the robot code if the camera is to be used. Normally the task is left running, but if desired it may be stopped. The activity of image acquisition may also be controlled, for example if you only want to use the camera in Autonomous mode, you may either call *StopCameraTask()* to end the task or call *StopImageAcquisition()* to leave the task running but not reading images from the camera.

Camera sensor property configuration

ConfigureCamera () sends a string to the camera that updates sensor properties. GetCameraSetting () queries the camera sensor properties. The properties that may be updated are listed below along with their out-of-the-box defaults:

- brightness =50
- whitebalance=auto

- exposure=auto
- exposurepriority=auto
- colorlevel=99
- sharpness=0

GetImageSetting() queries the camera image appearance properties (see the Camera Initialization section below). Examples of property configuration and query calls are below:

C++
```
// set a property
ConfigureCamera("whitebalance=fixedfluor1");

// query a sensorproperty
char responseString[1024];                      // create string
bzero (responseString, 1024);             // initialize string
if (GetCameraSetting("whitebalance",responseString)=-1) {
    printf("no response from camera \n);
} else {printf("whitebalance: %s \n",responseString);}

// query an appearance property
if (GetImageSetting("resolution",responseString)=-1) {
    printf("no response from camera \n);
} else {printf("resolution: %s \n",responseString);}
```

The example program CameraDemo.cpp will configure properties in a variety of ways and take snapshots that may be FTP'd from the cRIO for analysis.

Sensor property configuration using a text file

The utility *ProcessFile()* sets obtain camera sensor configuration specified from a file called "cameraConfig.txt" on the cRIO. *ProcessFile()* is called the first time with 0 lineNumber to get the number of lines to read. On subsequent calls each lineNumber is requested to get one camera parameter. There should be one property=value entry on each line, i.e. "exposure=auto"   A sample cameraConfig.txt file is included with the CameraDemo project. This file must be placed on the root directory of the cRIO. Below is an example file:

C++
```
#######################
! lines starting with ! or # or comments
! this a sample configuration file
! only sensor properties may be set using this file
! - set appearance properties when StartCameraTask() is called
#######################
exposure=auto
colorlevel=99
```

Simple Camera initialization

StartCameraTask() initializes the camera to serve MJPEG images using the following camera appearance defaults:

- Frame Rate = 10 frames / sec
- Compression = 0
- Resolution = 160x120
- Rotation = 0

C++

```
if (StartCameraTask() == -1) {
                  dprintf( LOG_ERROR, "Failed to spawn camera
task; Error code %s",
                              GetErrorText(GetLastError()) );
}
```

Configurable Camera initialization

Image processing places a load on the cRIO which may or may not interfere with your other robot code. Depending on needed speed and accuracy of image processing, it is useful to configure the camera for performance. The highest frame rates may acquire images faster than your processing code can process them, especially with higher resolution images. If your camera mount is upside down or sideways, adjusting the Image Rotation in the start task command will compensate and images will look the same as if the camera was mounted right side up. Once the camera is initialized, it begins saving images to an area of memory accessible by other programs. The images are saved both in the raw (JPEG) format and in a decoded format (Image) used by the NI image processing functions.

C++

```
int frameRate = 15;                             // valid values 0 - 30
int compression = 0;                            // valid values 0 - 100
ImageSize resolution = k160x120;    // k160x120, k320x240, k640x480
ImageRotation rot = ROT_180;                    // ROT_0, ROT_180

StartCameraTask(frameRate, compression, resolution, rot);
```

Image Acquisition

Images of types IMAQ_IMAGE_HSL, IMAQ_IMAGE_RGB, and IMAQ_IMAGE_U8 (gray scale) may be acquired from the camera. To obtain an image for processing, first create the image structure and then call GetImage() to get the image and the time that it was received from the camera:

C++
```
double timestamp;                              // timestamp of image returned
Image* cameraImage = frcCreateImage(IMAQ_IMAGE_HSL);
if (!cameraImage)  { printf(“error: %s”, GetErrorText(GetLastError()) };

if ( !GetImage(cameraImage, &timestamp) )  {
      printf(“error: %s”, GetErrorText(GetLastError()) };
```

`GetImage()` reads the most recent image, regardless of whether it has been previously accessed. Your code can check the timestamp to see if it’s an image you already processed.
Alternatively, **`GetImageBlocking()`** will wait until a new image is available if the current one has already been served. To prevent excessive blocking time, the call will return unsuccessfully if a new image is not available in 0.5 second.

C++
```
Image* cameraImage = frcCreateImage(IMAQ_IMAGE_HSL);
double timestamp;                      // timestamp of image returned
double lastImageTimestamp;   // timestamp of last image, to ensure image is new

int success = GetImageBlocking(cameraImage, &timestamp, lastImageTimestamp);
```

Camera Metrics

Various camera instrumentation counters used internally may be accessed that may be useful for camera performance analysis and error detection. Here is a list of the metrics:
CAM_STARTS, CAM_STOPS, CAM_NUM_IMAGE, CAM_BUFFERS_WRITTEN, CAM_BLOCKING_COUNT, CAM_SOCKET_OPEN, CAM_SOCKET_INIT_ATTEMPTS, CAM_BLOCKING_TIMEOUT, CAM_GETIMAGE_SUCCESS, CAM_GETIMAGE_FAILURE, CAM_STALE_IMAGE, CAM_GETIMAGE_BEFORE_INIT, CAM_GETIMAGE_BEFORE_AVAILABLE, CAM_READ_JPEG_FAILURE, CAM_PC_SOCKET_OPEN, CAM_PC_SEND-IMGAGE_SUCCESS, CAM_PC_SENDIMAGE_FAILURE, CAM_PID_SIGNAL_ERR, CAM_BAD_IMAGE_SIZE, CAM_HEADER_ERROR
The following example call gets the number of images served by the camera:

C++
```
int result = GetCameraMetric(CAM_NUM_IMAGE);
```

Images to PC

The class PCVideoServer, when instantiated, creates a separate task that sends images to the PC for display on a dashboard application. The sample program DashboardDemo shows an example of use.

C++
```
StartCameraTask();       // Initialize the camera
PCVideoServer pc;              // The constructor starts the image server
task
pc.Stop();              // Stop image server task
pc.Start();             // Restart task and serve images again
```

To use this code with the LabVIEW dashboard, the PC must be configured as IP address 10.x.x.6 to correspond with the cRIO address 10.x.x.2.

Controlling Motors

The WPI Robotics library has extensive support for motor control. There are a number of classes that represent different types of speed controls and servos. The library is designed to support non-PWM motor controllers that will be available in the future. The WPI Robotics Library currently supports two classes of speed controllers, PWM-based motor controllers (Jaguars or Victors) and servos. Motor speed controller speed values floating point numbers that range from -1.0 to +1.0 where -1.0 is full speed in one direction, and 1.0 is full speed in the other direction. 0.0 represents stopped. Motors can also be set to disabled, where the signal is no longer sent to the speed controller.
There are a number of motor controlling classes as part of th

Type	**Usage**
PWM	Base class for all the pwm-based speed controllers and servos
Victor	Speed controller supplied by Innovation First, commonly used in robotics competitions, with a 10ms update rate.
Jaguar	Advanced speed controller used for 2009 and future FRC competitions with a 5ms update rate.
Servo	Class designed to control small hobby servos as typically supplied in the FIRST kit of parts.
RobotDrive	General purpose class for controlling a robot drive train with either 2 or 4 drive motors. It provides high level operations like turning. It does this by controlling all the robot drive motors in a coordinated way. It's useful for both autonomous and tele-operated driving.

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, and Servo subclasses that set the update rate, deadband elimination, and profile shaping of the output signal.

Victor

The Victor class represents the Victor speed controllers provided by Innovation First. They have a minimum 10ms update period and only take a PWM control signal. The minimum and maximum values that will drive the Victor speed control vary from one unit to the next. You can fine tune the values for a particular speed controller by using a simple program that steps the values up and down in single raw unit increments. You need the following values

Value	Description
Max	The maximum value where the motors stop changing speed and the light on the Victor goes to full green.
DeadbandMax	The value where the motor just stops operating.
Center	The value that is in the center of the deadband that turns off the motors.
DeadbandMin	The value where the motor just starts running in the opposite direction.
Min	The minimum value (highest speed in opposite direction) where the motors stop changing speed.

With these values, call the **`SetBounds`** method on the created Victor object.

C++

```
void SetBounds(INT32 max,
                              INT32 deadbandMax,
                              INT32 center,
                              INT32 deadbandMin,
                              INT32 min);
```

Example

Jaguar

The Jaguar class supports the Luminary Micro Jaguar speed controller. It has an update period of slightly greater than 5ms and currently uses only PWM output signals. In the future the more sophisticated Jaguar speed controllers might have other methods for control of its many extended functions.
The input values for the Jaguar range from -1.0 to 1.0 for full speed in either direction with 0 representing stopped.
Use of limit switches
TODO
Example
TODO

Servo

The Servo class supports the Hitechnic servos supplied by *FIRST*. They have a 20ms update period and are controlled by PWM output signals.
The input values for the Servo range from 0.0 to 1.0 for full rotation in one direction to full rotation in the opposite direction. There is also a method to set the servo angle based on the (currently) fixed minimum and maximum angle values.

For example, the following code fragment rotates a servo through its full range in 10 steps:

```
Servo servo(3);    // create a servo on PWM port 3 on the first module

float servoRange = servo.GetMaxAngle() - servo.GetMinAngle();

for (float angle = servo.GetMinAngle();    // step through range of angles
                   angle < servo.GetMaxAngle();
                   angle += servoRange / 10.0)
{
      servo.SetAngle(angle);                // set servo to angle
      Wait(1.0);                            // wait 1 second
}
```

RobotDrive

The RobotDrive class is designed to simplify the operation of the drive motors based on a model of the drive train configuration. The idea is to describe the layout of the motors. Then the class can generate all the speed values to operate the motors for different situations. 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.
To use it, create a RobotDrive object specifying the left and right Jaguar motor controllers on the robot:

```
RobotDrive drive(1, 2);        // left, right motors on ports 1,2
```

Or

```
RobotDrive drive(1, 2, 3, 4);  // four motor drive configuration
```

This sets up the class for a 2 motor configuration or a 4 motor configuration. There are additional methods that can be called to modify the behavior of the setup.

```
SetInvertedMotor(kFrontLeftMotor, true);
```

This sets the operation of the front left motor to be inverted. This might be necessary depending on the gearing of your drive train.

Once set up, there are methods that can help with driving the robot either from the Driver Station controls or through programmed operatio

Method	Description
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, positive values forward and negative values backwards. The turn value tries to specify constant radius turns for any drive speed. The negative values represent left turns and the positive values represent right turns.
TankDrive(leftStick, right-Stick)	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(magnitude, 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(left Speed, 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.

The Drive method of the RobotDrive class is designed to support feedback based driving. Suppose you want the robot to drive in a straight line despite physical variations in its parts and external forces. There are a number of strategies, but two examples are using GearTooth sensors or a gyro. In either case an error value is generated that tells how far from straight the robot is currently tracking. This error value (positive for one direction and negative for the other) can be scaled and used directly with the turn argument of the Drive method. This causes the robot to turn back to straight with a correction that is proportional to the error – the larger the error, the greater the turn.
By default the RobotDrive class assumes that Jaguar speed controllers are used. To use Victor speed controllers, create the Victor objects then call the RobotDrive constructor passing it pointers or references to the Victor objects rather than port numbers.
Example
TODO

Getting Feedback from the Drivers Station

The driver station is constantly communicating with the robot controller. You can read the driver station values of the attached joysticks, digital inputs, and analog inputs, and write to the digital outputs. The DriverStation class has methods for reading and writing everything connected to it including joysticks. There is another object, Joystick, that provides a more convenient set of methods for dealing with joysticks and other HID controllers connected to the USB ports.

Getting data from the digital and analog ports

Building a driver station with just joysticks is simple and easy to do, especially with the range of HID USB devices supported by the driver station. Custom interfaces can be constructed using the digital and analog I/O on the driver station. Switches can be connected to the digital inputs, the digital outputs can drive indicators, and the analog inputs can read various sensors, like potentiometers. Here are some examples of custom interfaces that are possible:

- Set of switches to set various autonomous modes and options
- Potentiometers on a model of an arm to control the actual arm on the robot
- Rotary switches with a different resistor at each position to generate unique voltage to add effectively add more switch inputs
- Three pushbutton switches to set an elevator to one of three heights automatically

The range of possibilities is limited to your imagination. These custom interfaces often give the robot faster control than is available from a standard joystick or controller.

You can read/write the driver station analog and digital I/O using the following DriverStation methods:

float GetAnalogIn(UINT32 *channel*)	Read an analog input value connected to port *channel*
bool GetDigitalIn(UINT32 *channel*)	Read a digital input value connected to port *channel*
void SetDigitalOut(UINT32 *channel*, bool *value*)	Write a digital output *value* on port *channel*
bool GetDigitalOut(UINT32 *channel*)	Read the currently set digital output value on port *channel*

Note: The driver station does not have pull-up or pull-down resistors on any of the digital inputs. This means that unconnected inputs will have a random value. You must use external pull-up or pull-down resistors on digital inputs to get repeatable results.

Other Driver Station features

The Driver Station is constantly communicating with the Field Management System (FMS) and provides additional status information through that connection:

bool IsDisabled()	Robot state
bool IsAutonomous();	Field state (autonomous vs. teleop)
bool IsOperatorControl();	Field state
UINT32 GetPacketNumber();	Sequence number of the current driver station received data packet
Alliance GetAlliance();	Alliance (red, blue) for the match
UINT32 GetLocation();	Starting field position of the robot (1, 2, or 3)
float GetBatteryVoltage();	Battery voltage on the robot

Joysticks

The standard input device supported by the WPI Robotics Library is a USB joystick. The 2009 kit joystick 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 axis control or buttons.

The joystick must be connected to one of the four available USB ports on the driver station. When the station is turned on, the joysticks must be at their center position, as the startup routine will read whatever position they are in as center. The constructor takes either the port number the joystick is plugged into, followed by the number of axes and then the number of buttons, or just the port number from the driver's station. The former is primarily for use in sub-classing (For example, to create a class or a non-kit joystick), and the latter for a standard kit joystick.

C++

```
Joystick driveJoy(1);
Joystick opJoy(2,5,8);
```

The above example would create a default joystick called driveJoy on USB port 1 of the driver station, and something like a Microsoft Sidewinder (which has five analog axises, i.e. x, y, throttle, twist, and the hat, and eight buttons) – which would be a good candidate for a subclass of Joystick.
There are two methods to access the axes of the joystick. Each input axis is labeled as the X, Y, Z, Throttle, or Twist axis. For the kit joystick, the applicable axes are labeled correctly; a non-kit joystick will require testing to determine which axes correspond to which degrees of freedom.
Each of these axes has an associated accessor; the X axis from driveJoy in the above example could be read by calling driveJoy.GetX(); the twist and throttle axes are accessed by driveJoy.GetTwist() and driveJoy.GetThrottle(), respectively.

Alternatively, the axes can be accessed via the the GetAxis() and GetRawAxis() methods. GetAxis() takes an AxisType – kXAxis, kYAxis, kZAxis, kTwistAxis, or kThrottleAxis – and returns that axis's value. GetRawAxis() takes an a number (1-6) – and returns the value of the axis associated with that number – these numbers are reconfigurable and generally used with custom control systems, since the other two methods reliably return the same data for a given axis.

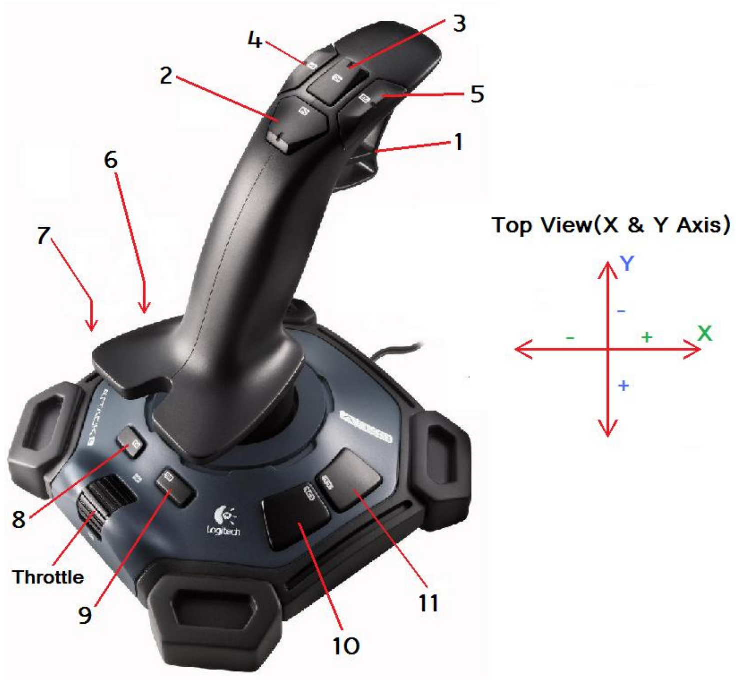
There are three ways to access the top button (defaulted to button 2) and trigger (button 1). The first is to use their respective accessor methods – GetTop() and GetTrigger(), which return a true or false value based on whether the button is currently being pressed. A second method is to call GetButton(), which takes a ButtonType – which can be either kTopButton or kTriggerButton. The last method is one that allows access to the state of every button on the joystick – GetRawButton(). This method takes a number corresponding to a button on the joystick (see diagram below), and return the state of that button.

In addition to the standard method of accessing the Cartesian coordinates (x and y axes) of the joystick's position, WPILib also has the ability to return the position of the joystick as a magnitude and direction. To access the magnitude, the GetMagnitude() method can be called, and to access the direction, either GetDirectionDegrees() or GetDirectionRadians() can be called.

C++

```
Joystick driveJoy(1);
      Jaguar leftControl(1);
      Jaguar rightControl(2);

      if(driveJoy.GetTrigger())             //If the trigger is pressed
      {
            //Have the left motor get input from Y axis
            //and the right motor get input from X axis
            leftControl.Set(driveJoy.GetY());
            rightControl.Set(driveJoy.GetAxis(kXAxis));
      }
      else if(driveJoy.GetRawButton(2))   //If button number 2 pressed (top)
      {
            //Have both right and left motors get input
            //from the throttle axis
            leftControl.Set(driveJoy.GetThrottle());
            rightControl.Set(driveJoy.GetAxis(kThrottleAxis));
      }
      //If button number 4 is pressed
      else if(driveJoy.GetRawButton(4))     //If button number 4 is pressed
      {
            //Have the left motor get input from the
            //magnitude of the joystick's position
            leftControl.Set(driveJoy.GetMagnitude());
      }
```



Controlling Pneumatics

These classes make it easier to use pneumatics in your robot.

Class	Purpose
Solenoid	Can control pneumatic actuators directly without the need for an additional relay. (In the past a Spike relay was required along with a digital output port to control a pneumatics component.)
Compressor	Keeps the pneumatics system charged by using a pressure switch and software to turn the compressor on and off as needed.

Compressor

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

- The Digital output port connected to the Spike relay that is controlling the power to the compressor. (A digital output or Solenoid module port alone doesn't supply enough current to operator the compressor.)
- The Digital input port connected to the pressure switch that is monitoring the accumulator pressure.

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

To use the Compressor class create an instance of the Compressor object and **`Start()`** it. 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.

For example, suppose you had a compressor and 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. You could create and start the compressor running in the constructor of your RobotBase derived object using the following 2 lines of code.

C++

```cpp
Compressor *c = new Compressor(4, 2);
c->Start();
```

Note: The variable c is a pointer to a compressor object and the object is allocated using the **`new`** operator. If it were allocated as a local variable in the constructor, at the end of the constructor function its local variables would be deallocated and the compressor would stop operating.

That's all that is required to enable the compressor to operate for the duration of the robot program.

C++ Object Life Span

You need the Compressor object to last the entire game. If you allocate it with **`new`**, the best practice is to store the pointer in a member variable then **`delete`** it in the Robot's destructor.

C++

```
class RobotDemo : public SimpleRobot
{
      Compressor *m_compressor;

public:
      RobotDemo()
      {
            m_compressor = new Compressor(4, 2);
            m_compressor->Start();
      }

      ~RobotDemo()
      {
            delete m_compressor;
      }
}
```

Alternatively, declare it as a member object then initialize and **`Start()`** it in the Robot's constructor. In this case you need to use the constructor's "initialization list" to initialize the Compressor object. The C++ compiler will quietly give RobotDemo a destructor that deletes the Compressor object.

C++

```
class RobotDemo : public SimpleRobot
{
      Compressor m_compressor;

public:
      RobotDemo() : m_compressor(4, 2)
      {
            m_compressor.Start();
      }
}
```

In Java you would accomplish the same thing as follows:

Solenoid (Pneumatics)

The Solenoid object controls the outputs of the NI 9472 Digital Output Module. It is designed to apply an input voltage to any of the 8 outputs. Each output can provide up to 1A of current. The module is designed to operate 12v pneumatic solenoids used on FIRST robots. This makes the use of relays unnecessary for pneumatic solenoids.

Note: The NI 9472 Digital Output Module does not provide enough current to operate a motor or the compressor so relays connected to Digital Sidecar digital outputs will still be required for those applications.

The port numbers on the Solenoid class range from 1-8 as printed on the pneumatics breakout board.

Note: The NI 9472 indicator lights are numbered 0-7 for the 8 ports which is different numbering than used by the class or the pneumatic bumper case silkscreening.

Example

Setting the output values of the Solenoid objects to true or false will turn the outputs on and off respectively. The following code fragment will create 8 Solenoid objects, initialize each to true (on), and then turn them off, one per second. Then it turns them each back on, one per second, and deletes the objects.

C++

```
Solenoid *s[8];
      for (int i = 0; i < 8; i++)
            s[i] = new Solenoid(i + 1);     // allocate the Solenoid objects
      for (int i = 0; i < 8; i++)
      {
            s[i]->Set(true);                // turn them all on
      }
      Wait(1.0);
      for (int i = 0; i < 8; i++)
      {
            s[i]->Set(false);               // turn them each off in turn
            Wait(1.0);
      }
      for (int i = 0; i < 8; i++)
      {
            s[i]->Set(true);                // turn them back on in turn
            Wait(1.0);
            delete s[i];                    // delete the objects
      }
```

You can observe the operation of the Solenoid class by looking at the indicator lights on the 9472 module.

Vision / Image Processing

Access to National Instrument's nivison library for machine vision enables automated image processing for color identification, tracking and analysis. The VisionAPI.cpp file provides open source C wrappers to a subset of the proprietary library. The full specification for the simplified FRC Vision programming interface is in the FRC Vision API Specification document, which is in the *WindRiver\docs\extensions\FRC* directory of the Wind River installation with this document. The FRC Vision interface also includes high level calls for color tracking (TrackingAPI.cpp). Programmers may also call directly into the low level library by including nivision.h and using calls documented in the NI Vision for LabWindows/CVI User Manual.

Naming conventions for the vision processing wrappers are slightly different from the rest of WPILib. C routines prefixed with "imaq" belong to NI's LabVIEW/CVI vision library. Routines prefixed with "frc" are simplified interfaces to the vision library provided by BAE Systems for FIRST Robotics Competition use.

Sample programs provided include SimpleTracker, which in autonomous mode tracks a color and drives toward it, VisionServoDemo, which also tracks a color with a two-servo gimbal. VisionDemo demonstrates other capabilities including storing a JPEG image to the cRIO, and DashboardDemo sends images to the PC Dashboard application.

Image files may be read and written to the cRIO non-volatile memory. File types supported are PNG, JPEG, JPEG2000, TIFF, AIDB, and BMP. Images may also be obtained from the Axis 206 camera. Using the FRC Vision API, images may be copied, cropped, or scaled larger/smaller. Intensity measurements functions available include calculating a histogram for color or intensity and obtaining values by pixel. Contrast may be improved by equalizing the image. Specific color planes may be extracted. Thresholding and filtering based on color and intensity characteristics are used to separate particles that meet specified criteria. These particles may then be analyzed to find the characteristics.

Color Tracking

High level calls provide color tracking capability without having to call directly into the image processing routines. You can either specify a hue range and light setting, or pick specific ranges for hue, saturation and luminance for target detection.

Example 1 using defaults

Call GetTrackingData() with a color and type of lighting to obtain default ranges that can be used in the call to FindColor(). The ParticleAnalysisReport returned by FindColor() specifies details of the largest particle of the targeted color.

C++

```
TrackingThreshold tdata = GetTrackingData(BLUE, FLUORESCENT);
ParticleAnalysisReport par;

if (FindColor(IMAQ_HSL, &tdata.hue, &tdata.saturation,
                                &tdata.luminance, &par)
{
      printf("color found at x = %i, y = %i",
            par.center_mass_x_normalized, par.center_mass_y_normal-
ized);
      printf("color as percent of image: %d",
            par.particleToImagePercent);
}
```

The normalized center of mass of the target color is a range from −1.0 to 1.0, regardless of image size. This value may be used to drive the robot toward a target.

Example 2 using specified ranges

To manage your own values for the color and light ranges, you simply create Range objects:

C++

```
Range hue, sat, lum;

hue.minValue = 140;      // Hue
hue.maxValue = 155;
sat.minValue = 100;      // Saturation
sat.maxValue = 255;
lum.minValue = 40; // Luminance
lum.maxValue = 255;

FindColor(IMAQ_HSL, &hue, &sat, &lum, &par);
```

Tracking also works using the Red, Green, Blue (RGB) color space, however HSL gives more consistent results for a given target.

Example 3 using return values

Here is an example program that enables the robot to drive towards a green target. When it is too close or too far, the robot stops driving. Steering like this is quite simple as shown in the example. The following declarations in the class are used for the example:

C++

```
RobotDrive *myRobot
Range greenHue, greenSat, greenLum;
```

This is the initialization of the RobotDrive object, the camera and the colors for tracking the target. It would typically go in the RobotBase derived constructor.

C++

```
if (StartCameraTask() == -1) {
            printf( “Failed to spawn camera task; Error code %s”,
                        GetErrorText(GetLastError()) );
}
myRobot = new RobotDrive(1, 2);

// values for tracking a target - may need tweaking in your envi-
ronment
greenHue.minValue = 65; greenHue.maxValue = 80;
greenSat.minValue = 100; greenSat.maxValue = 255;
greenLum.minValue = 100; greenLum.maxValue = 255;
```

Here is the code that actually drives the robot in the autonomous period. The code checks if the color was found in the scene and that it was not too big (close) and not too small (far). If it is in the limits, then the robot is driven forward full speed (1.0), and with a turn rate determined by the **cen-**

ter_mass_x_normalized value of the particle analysis report.
The **center_mass_x_normalized** value is 0.0 if the object is in the center of the frame; otherwise it varies between -1.0 and 1.0 depending on how far off to the sides it is. That is the same range as the Drive method uses for the turn value. If the robot is correcting in the wrong direction then simply negate the turn value.

C++

```
while (IsAutonomous())
{
      if ( FindColor(IMAQ_HSL, &greenHue, &greenSat, &greenLum, &par)
            && par.particleToImagePercent < MAX_PARTICLE_TO_IMAGE_PERCENT
            && par.particleToImagePercent > MIN_PARTICLE_TO_IMAGE_PERCENT )
      {
            myRobot->Drive(1.0, (float)par.center_mass_x_normalized);
      }
      else myRobot->Drive(0.0, 0.0);
      Wait(0.05);
}
myRobot->Drive(0.0, 0.0);
```

Example 4 two color tracking

An example of tracking a two color target is in the demo project TwoColorTrackDemo. The file Target.cpp in this project provides an API for searching for this type of target. The example below first creates tracking data:

C++

```
// PINK
            sprintf (td1.name, "PINK");
            td1.hue.minValue = 220;
            td1.hue.maxValue = 255;
            td1.saturation.minValue = 75;
            td1.saturation.maxValue = 255;
            td1.luminance.minValue = 85;
            td1.luminance.maxValue = 255;
// GREEN
            sprintf (td2.name, "GREEN");
            td2.hue.minValue = 55;
            td2.hue.maxValue = 125;
            td2.saturation.minValue = 58;
            td2.saturation.maxValue = 255;
            td2.luminance.minValue = 92;
            td2.luminance.maxValue = 255;
```

Call FindTwoColors() with the two sets of tracking data and an orientation (ABOVE, BELOW, RIGHT, LEFT) to obtain two ParticleAnalysisReports which have details of a two-color target.
Note: The FindTwoColors API code is in the demo project, not in the WPILib project

C++

```
// find a two color target
if (FindTwoColors(td1, td2, ABOVE, &par1, &par) {
      // Average the two particle centers to get center x & y
of combined target
      horizontalDestination = (par1.center_mass_x_normalized +
                                    par2.center_mass_x_
normalized) / 2;
      verticalDestination = (par1.center_mass_y_normalized +
                                    par2.center_mass_y_normal-
ized) / 2;
{
```

To obtain the center of the combined target, average the x and y values. As before, use the normalized values to work within a range of -1.0 to +1.0. Use the *center_mass_x* and *center_mass_y* values if the exact pixel position is desired.
Several parameters for adjusting the search criteria are provided in the Target.h header file (again, provided in the TwoColorTrackDemo project, not WPILib). The initial settings for all of these paramters are very open, to maximize target recognition. Depending on your test results you may want to adjust these, but remember that the lighting conditions at the event may give different results. These parameters include:

- FRC_MINIMUM_PIXELS_FOR_TARGET – (default 5) Make this larger to prevent extra processing of very small targets.
- FRC_ALIGNMENT_SCALE – (default 3.0) scaling factor to determine alignment. To ensure one target is exactly above the other, use a smaller number. However, light shining directly on the target causes significant variation, so this parameter is best left fairly high.
-
- FRC_MAX_IMAGE_SEPARATION (default 20) Number of pixels that can exist separating the two colors. Best number varies with image resolution. It would normally be very low but is set to a higher number to allow for glare or incomplete recognition of the color.
- FRC_SIZE_FACTOR (default 3) Size difference between the two particles. With this setting, one particle can be three times the size of the other.
- FRC_MAX_HITS (default 10) Number of particles of each color to analyze. Normally the target would be found in the first (largest) particles. Reduce this to increase performance, Increase it to maximize the chance of detecting a target on the other side of the field.
- FRC_COLOR_TO_IMAGE_PERCENT (default 0.001) One color particle must be at least this percent of the image.

Other Classes

PID Programming

PID controllers are a powerful and widely used implementation of closed loop control. 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.
The following example shows how to create a PIDController to set the position of a turret to a position related to the x-axis on a joystick using a single motor on a Jaguar and a potentiometer for angle feedback. As the joystick X value changes, the motor should drive to a position related to that new value. The PIDController class will ensure that the motion is smooth and stops at the right point.
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 can be done with the following expression:

```
(turretStick.GetX() + 1.0) * 2.5
```

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.

C++

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

      turretControl.Enable();  // start calculating PIDOutput val-
ues

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

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.

Relays

The cRIO provides the connections necessary to wire IFI spikes via the relay outputs on the digital sidecar. The sidecar 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 sidecar, which is labeled as output A, while the reverse signal output is sent over the center pin, which is labeled output B. The final pin is a ground connection.
When a Relay object is created in WPILib, its constructor takes a channel and direction, or a slot, channel and direction. The slot is the slot number that the digital module is plugged into (the digital module being what the digital sidecar is connected to on the cRIO) – this parameter is not needed if only the first digital module is being used. The channel is the number of the connection on being used on the digital sidecar. The direction can be kBothDirections (two direction solenoid), kForwardOnly (uses only the forward pin), or kReverseOnly, which 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 of the object.
Included in the Relay class

Method	**Description**
Void Set(Value value)	This method sets the the state of the relay – Valid inputs: *All Directions:* kOff – turns off the Relay *kForwardOnly or kReverseOnly:* kOn – turns on forward or reverse of relay, depending on direction *kForwardOnly:* kForward – set the relay to forward *kReverseOnly:* kReverse – set the relay to reverse
Void SetDirection(Direction direction)	Sets the direction of the relay – Valid inputs: *kBothDirections:* Allows the relay to use both the forward and reverse pins on the channel *kForwardOnly:* Allows relay to use only the forward signal pin *kReverseOnly:* Allows relay to use only the reverse signal pin

C++

```
Relay m_relay(1);
Relay m_relay2(2,Relay::kForwardOnly);

m_relay.SetDirection(Relay::kReverseOnly);
m_relay.Set(Relay::kOn);
m_relay2.Set(Relay::kForward);
m_relay.Set(Relay::kOff);
```

In this example, m_relay is initialized to be on channel 1. Since no direction is specified, the direction is set to the default value of kBothDirections. m_relay2 is initialized to channel 2, with a direction of kForwardOnly. In the following line, m_relay is set to the direction of kReverseOnly, and is then turned on, which results in the reverse output being turned on. m_relay2 is then set to forward – since it is a forward only relay, this has the same effect as setting it to on. After that, m_relay is turned off, a command that turns off any active pins on the channel, regardless of direction.

Using the serial port

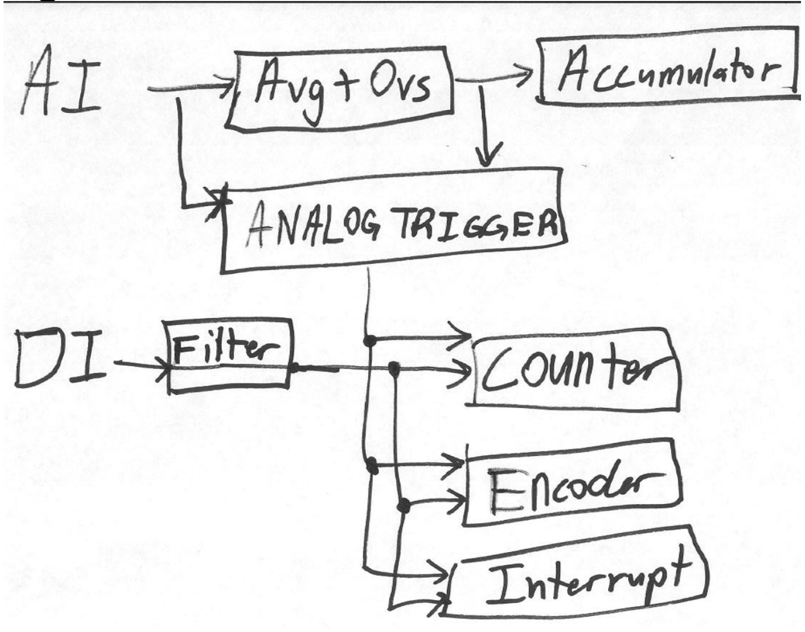
Using I2C

System Architecture

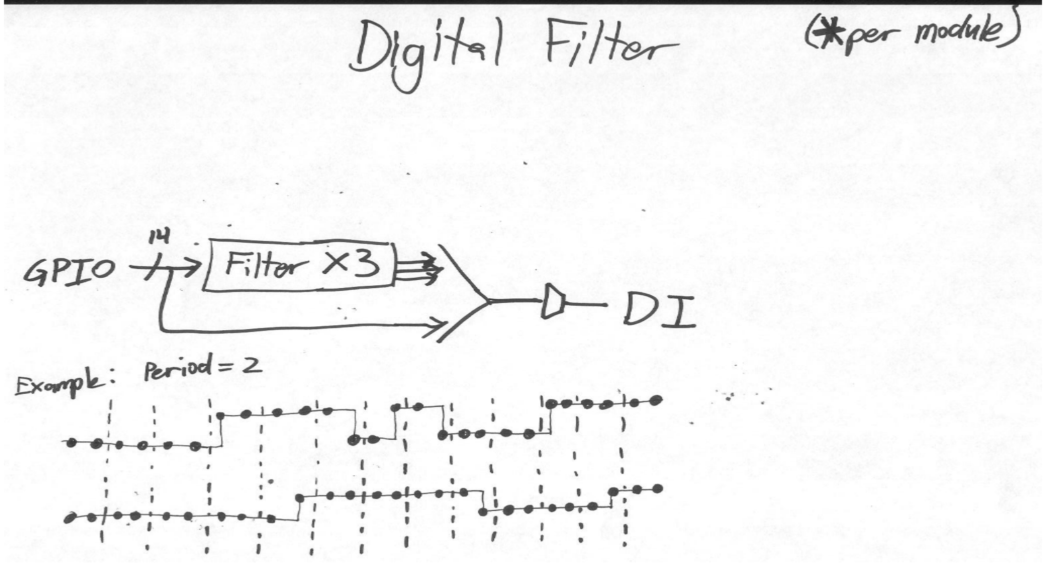
This section describes how the system is put together and how the libraries interact with the base hardware. It should give you better insight as to how the whole system works and its capabilities.

Note: This is a work in progress, the pictures will be cleaned up and explanations will be soon added. We wanted to make this available to you in its raw form rather than leaving it out all together.

Digital Sources



Digital Filter



Contributing to the WPI Robotics Library

Glossary

Concurrency
cRIO
deadlock
particle
quadrature encoder
race condition
semaphore
task
VxWorks