



# C/C++ Programming Guide for the FIRST Robotics Competition

Worcester Polytechnic Institute Robotics Resource Center

Rev 0.76

January 25, 2009

Brad Miller, Ken Streeter, Beth Finn, Jerry Morrison, Dan Jones, Ryan O'Meara

## Contents

Getting Started .....	5
What is the WPI Robotics Library .....	6
A simple robot program .....	8
Using objects .....	9
Writing C programs .....	11
Using Wind River Workbench.....	13
Setting up the environment .....	14
Creating a robot project .....	18
Building your project .....	20
Downloading the project to the cRIO .....	21
Debugging your robot program .....	22
Deploying the C/C++ Program .....	26
Creating a Robot Program .....	27
Pointers and addresses .....	30
Built-in Robot classes .....	31
SimpleRobot class .....	32
IterativeRobot class.....	33
WPI Robotics Library Conventions .....	34
RobotBase class .....	36
Watchdog timer class.....	37
Sensors.....	38
Digital I/O Subsystem.....	39
Digital Inputs .....	40
Digital Outputs.....	41
Accelerometer.....	42
Gyro .....	43
HiTechnicCompass.....	45
Ultrasonic rangefinder .....	46

Counter Subsystem .....	47
Counter Objects .....	48
Encoders .....	49
Geartooth Sensor .....	50
Quadrature Encoders.....	51
Analog Inputs.....	53
Analog Triggers.....	55
Camera.....	56
Controlling Motors .....	60
PWM.....	61
Victor .....	62
Jaguar.....	63
Servo.....	64
RobotDrive .....	65
Controlling Pneumatics.....	67
Compressor.....	68
Solenoid (Pneumatics) .....	70
Vision / Image Processing .....	71
Color Tracking.....	72
Concurrency.....	76
Creating tasks.....	77
Synchronized and Critical Regions.....	78
System Architecture.....	80
Digital Sources.....	81
Getting Feedback from the Drivers Station .....	82
Joysticks.....	84
Advanced Programming Topics .....	86
Using Subversion with Workbench .....	87
Getting the WPILib Source Code .....	91
Replacing WPI Robotics Library parts .....	97
Interrupts.....	98
Creating your own speed controllers .....	99

PID Programming .....	100
Using the serial port.....	102
Relays .....	103
Customizing analog sampling.....	105
Using I2C.....	106
C++ Tips.....	107
Creating an application in WorkBench.....	108
Contributing to the WPI Robotics Library.....	109
Glossary .....	110
Index .....	111

# Getting Started

DRAFT

# What is the WPI Robotics Library

The WPI Robotics library is a set of C++ 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 new environment. The robot controller for 2009 is a National Instruments cRIO-9074 real-time controller, or "cRIO" for short. It provides about 500x more memory. It runs 40x faster for fixed point and even faster for floating point over the PIC that we're used to using. The past years' high speed sensor-interrupt logic that required precise coding, hand optimization and lots of bugs has been replaced with dedicated hardware (FPGA). When the library wants 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.

We chose C++ as a language because we felt it represents a better level of abstraction for robot programs. C++ (when used properly) also encourages 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, drivers 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 will 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).

I 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.

# A simple robot program

Creating a robot program has been designed to be as simple as possible while still allowing a lot of flexibility. Here's an example of a template that represents the simplest robot program you can create.

```
#include "WPILib.h"
class RobotDemo : public SimpleRobot
{
public:
    RobotDemo()
    {
        // put initialization code here
    }

    void Autonomous()
    {
        // put autonomous code here
    }

    void OperatorControl()
    {
        // put operator control code here
    }
};

START_ROBOT_CLASS(RobotDemo);
```

There are several templates that can be used as starting points for writing robot programs. This one, SimpleRobot is probably the easiest to use. Simply add code for initializing sensors and anything else you need in the constructor, code for your autonomous program in the Autonomous function, and the code for your operator control part of the program in OperatorControl.

SimpleRobot is actually the name of a C++ class or object that is used as the base of this robot program called RobotDemo. To use it you create a subclass which is another name for your object that is based on the SimpleRobot class. By making a subclass, the new class, RobotDemo, inherits all the predefined behavior and code that is built into SimpleRobot.

# Using objects

In the WPI Robotics Library all sensors, motors, driver station elements, and more are all objects. For the most part, objects correspond to the physical things on your robot. Objects include the code and the data that makes the thing operate. Let's look at a Gyro. There are a bunch of operations, or methods, you can perform on a gyro:

- Create the gyro object – this sets up the gyro and causes it to initialize itself
- Get the current heading, or angle, from the gyro
- Set the type of the gyro, i.e. its Sensitivity
- Reset the current heading to zero
- Delete the gyro object when you're done using it

Creating a gyro object is done like this:

```
Gyro robotHeadingGyro(1);
```

`robotHeadingGyro` is a variable that holds the Gyro object that represents a gyro module connected to analog port 1. That's all you have to do to make an instance of a Gyro object.

*Note: by the way, an instance of an object is the chunk of memory that holds the data unique to that object. When you create an object that memory is allocated and when you delete the object that memory is deallocated.*

To get the current heading from the gyro, you simply call the `GetAngle` method on the gyro object. Calling the method is really just calling a function that works on the data specific to that gyro instance.

```
float heading = robotHeadingGyro.GetAngle();
```

This sets the variable `heading` to the current heading of the gyro connected to analog channel 1.

### Creating object instances

There are several ways of creating object instances used throughout the WPI Robotics Library and all the examples. Depending on how the object is created there are differences in how the object is referenced and deleted. Here are the rules:

Local variable declared inside a block or function (or inside another object)	Victor leftMotor(3);	leftMotor.Set(1.0);	Object is implicitly deallocated when the enclosing block is exited
Global declared outside of any enclosing blocks or functions; or a static variable	Victor leftMotor(3);	leftMotor.Set(1.0);	Object is not deallocated until the program exits
Pointer to object	Victor *leftMotor = new Victor(3);	leftMotor->Set(1.0);	Object must be explicitly deallocated using the C++ delete operator.

How do you decide what to use? The next section will discuss this.

# Writing C programs

You can also write C programs with the WPI Robotics Library using a set of C functions that map on top of the C++ classes and methods. To write C code:

- You need to create .cpp (C++ files) rather than .C files because the C wrapper functions take advantage of overloaded functions. This means that there are a number of functions that have the same name, but different argument lists. This increases the compatibility with the C++ programming interfaces and will make transition to C++ much easier if you choose to do that.
- Specify port and/or slot (module) numbers in most of the functions. Behind the scenes, the functions allocate C++ objects that correspond to the functions that you are using. This serves two purposes: it ensures that you are not using a particular port for two purposes accidentally since the C++ underlying functions track “reservations” and makes the code very similar to previous years where the port numbers were on each call.

You will find that there are a few C++ higher level code options that do not exist in C. The C wrapper is a lower level interface to the hardware connected to the cRIO and the driver station.

When you first use a sensor or an output device on a particular channel, the C wrapper will automatically allocate an object to control that device. Suppose you start using a Victor motor speed controller on port 5. On the first use, an object is created behind the scenes that corresponds to that motor. Each time you refer the Victor on port 5, the code calls the underlying object to set or get values. When you are finished using an object, you may delete it by calling the Delete function associated with the object, for example DeleteVictor(). This is usually not necessary because it is unlikely that you would ever need to delete a sensor once it is created—the I/O devices on a robot don’t usually change while the robot program is running. Those functions mostly exist for testing.

### Example of a C program

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);
```

# Using Wind River Workbench

Wind River Workbench is a complete C/C++ Interactive Development Environment (IDE) that handles all aspects of code development. It will help you:

- Write the code for your robot with editors, syntax highlighting, formatting, auto-completion, etc.
- Compile the source code into binary object code for the cRIO PowerPC architecture.
- Debug and test code by downloading the code to the cRIO robot controller and enabling you to step through line by line and examine variables of the running code.
- Deploy the program so that it will automatically start up when the robot is powered on.

You can even use Subversion, a popular source code repository server to manage your code and track changes. This is especially useful if there is more than one person doing software development.

# Setting up the environment

To use Workbench you need to configure it so that it knows about your robot and the programs that you want to download to it. There are three areas that need to be set up.

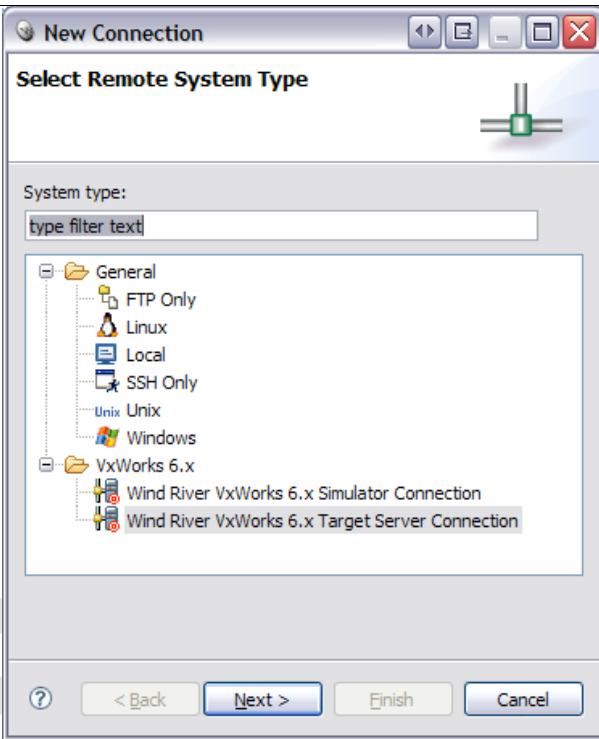
1. The target **remote system**, which is the cRIO that you will use to download and debug your programs.
2. The **run or debug configuration** that describes the program to be debugged and which remote system you want to debug it on.
3. The FIRST Downloader settings that tell which program should be deployed onto the cRIO when you are ready to load it for a competition or operation without the laptop.

## Creating a Remote System in Workbench

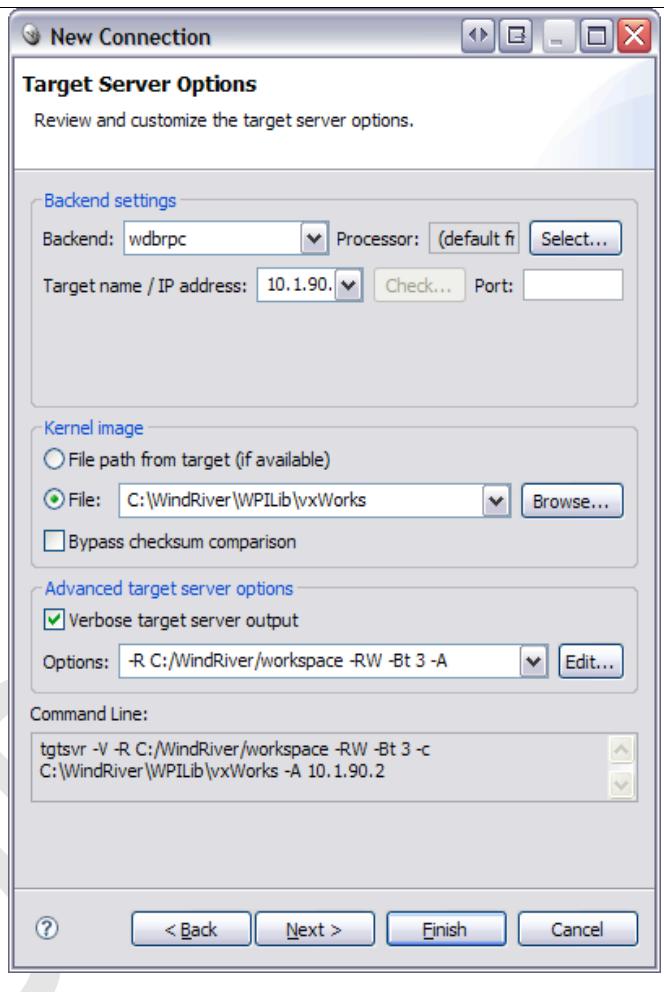
Workbench connects to your cRIO controller and can download and remotely debug programs running on it. In order to make that connection, Workbench needs to add your cRIO to its list of Remote Systems.

Each entry in the list tells Workbench the network address of your cRIO and has a kernel file that is required for remote access. To create the entry for your system do the following steps.

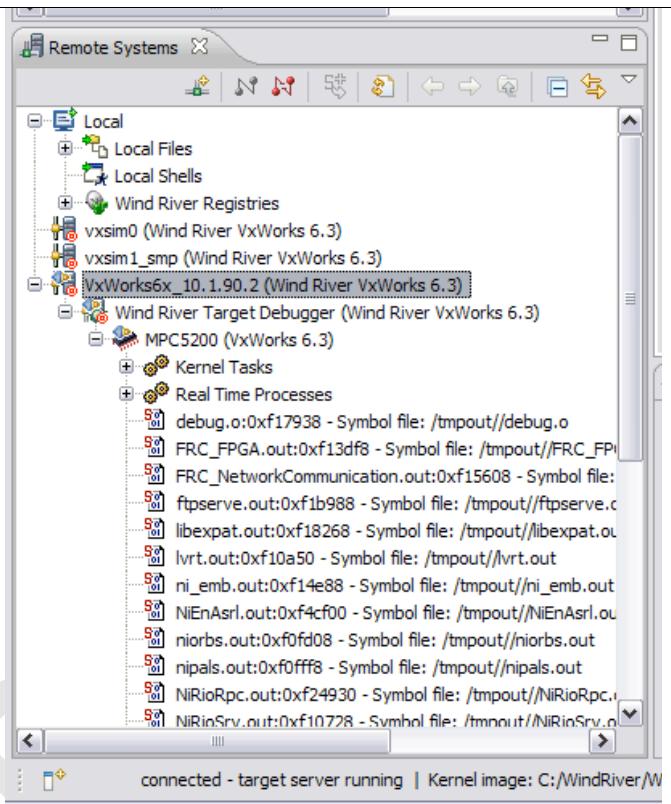
*Note: The “Reset connected Target” (reboot the server) command doesn’t work reliably and other features seem to have issues unless the “Console out” switch on the cRIO is set to on. Normally this switch enables the console output for viewing with a serial port, but leaving it on even if there is no serial cable connected improves system reliability.*

Right-click in the empty area in the “Remote Systems” window. Select “New Connection”.	
In the “Select Remote System Type” window select “Wind River VxWorks 6.x Target Server Connection” and click “Next”.	

Fill out the “Target Server Options” window with the IP address of your cRIO. It is usually 10.x.y.2 where x is the first 2 digits of your 4 digit team number and y is the last two digits. For example, team 190 (0190) would be 10.1.90.2. You must also select a Kernel Image file. This is located in the WindRiver install directory in the WPILib top level directory. This is typically called “C:\WindRiver\WPILib\VxWorks”.



If the cRIO is turned on and connected you will see the target server entry populated with the tasks currently running.



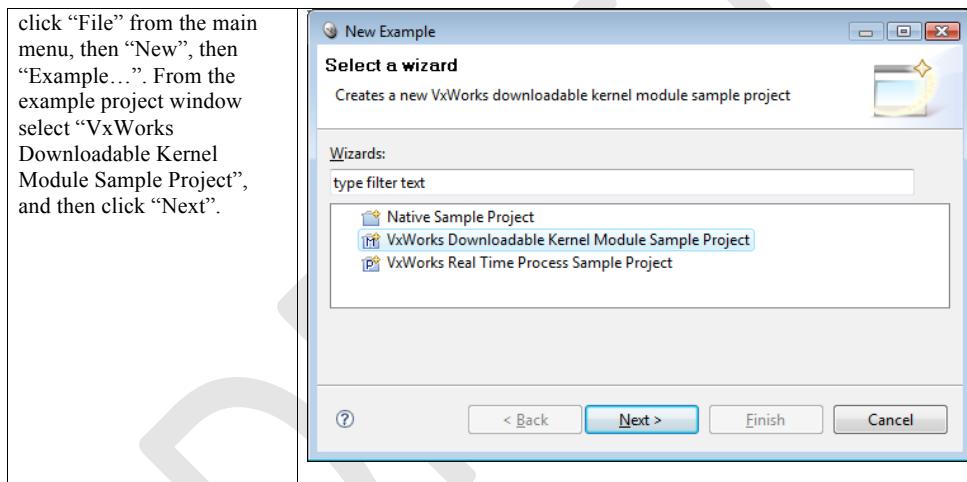
# Creating a robot project

The easiest way to create your own project for your robot is to start with one of the existing templates:

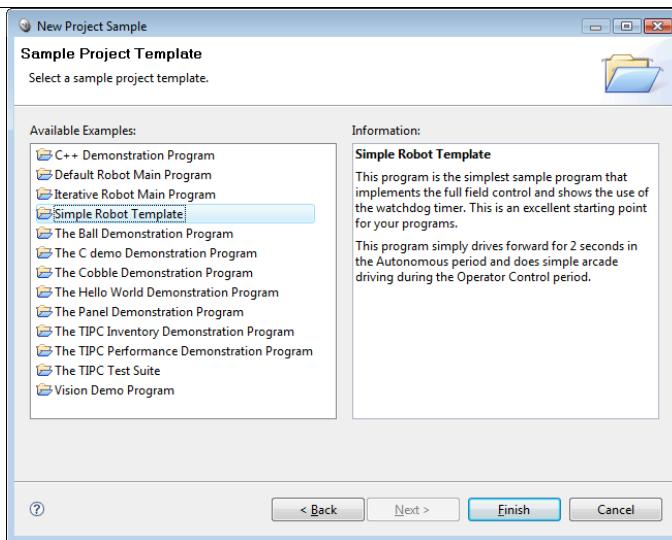
- SimpleRobotTemplate
- IterativeRobotTemplate (coming soon)

In both cases the templates are based on the RobotBase class and have some of the functions overridden to change the behavior. Additional templates can be implemented that implement other behaviors for example event driven models.

Follow these steps to create a sample project. In this case the sample is the SimpleRobotTemplate, but you can use any of the provided samples.



Select Simple Robot Template from Sample Project Template window. Notice that a description of the template is displayed in the Information window. Click "Finish" and a project will be created in your workspace that you can edit into your own program.



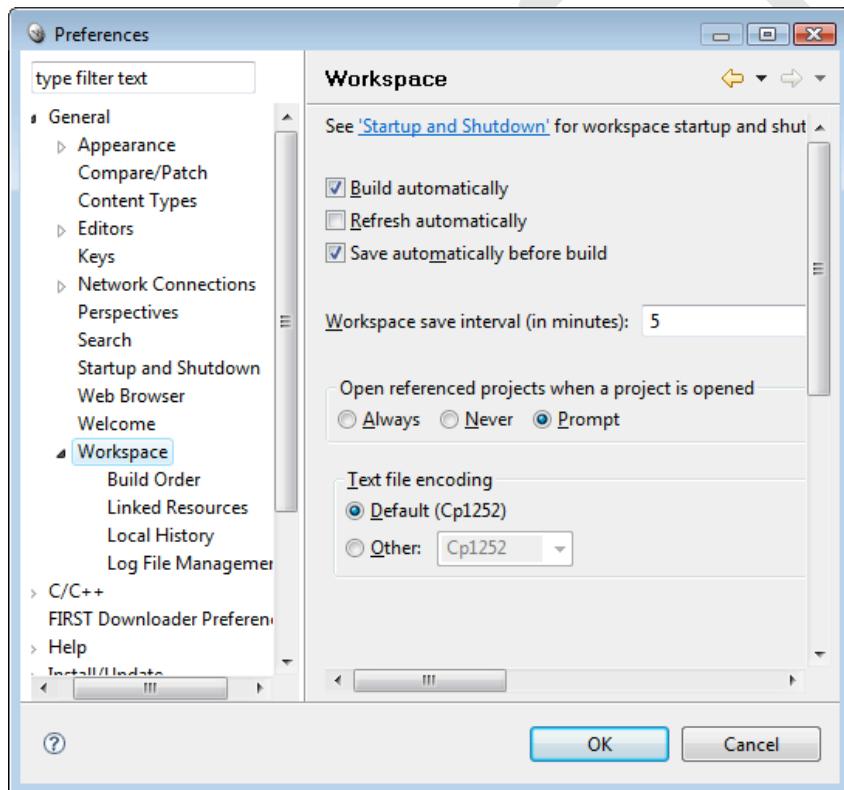
## Building your project

The project is built by right-clicking on the project name in the Project Explorer window and selecting “Build project” or “Rebuild project” from the popup context menu. This will cause Workbench to compile and link the project files into a .OUT executable file that may be either deployed or downloaded to the cRIO.

Another way of building the project is to automatic rebuild feature of Workbench. Whenever a file in the project is saved, a build will automatically be started to keep the project up to date. To enable this feature:

Select “Window”, then “Preferences”. In the Preferences panel, expand “General”, then “Workspace” and check the “Build automatically” option. A file can quickly be saved after editing by using the keyboard shortcut, Ctrl-S. Or save all open files at once using the Save All shortcut, Ctrl-Shift-S.

It's handy to turn on “Save automatically before build.” Then Workbench will always build with your latest changes to all files, saving lots of puzzlement.



## Downloading the project to the cRIO

There are two ways of getting your project into the cRIO:

1. Using a Run/Debug Configuration in Workbench. This loads the program into the cRIO RAM memory and allows it to run either with or without the debugger. When the robot is rebooted, the program will no longer be in memory.
2. Deploy the program through the FIRST Downloader option in Workbench. In this case the program will be written to the flash disk inside the cRIO and will run whenever it is rebooted until it is Undeployed (deleted from flash). This is the option to take a finished program and make it available for a match – so that it will run without an attached computer to always load it.

The deployed code only runs on reboot when the LabView Runtime goes through a list of programs to run on startup. Those programs are loaded from the flash disk on the cRIO. On each boot it tries to load the deployed program via a fixed filename (regardless of what you call the program). The Downloader “deploys” your program by copying it to the cRIO’s flash disk and can “undeploy” it by deleting it from the flash disk.

When you Run or Debug the program from Workbench, Workbench loads the program directly into cRIO RAM and runs from there.

If you have a robot program starting from flash disk and thus starting up automatically in the background then you also load one for debugging, things get very confusing.

It is also sometimes advantageous to reboot between debugging sessions. Sometimes things don't get completely cleaned up even if you try to unload the program. We're working on that. You can reboot remotely by right-clicking on the connection in the "Remote Systems" tab in Workbench and selecting "Reset connected Target". It takes about 15 seconds to reboot.

The steps for debugging if there is already a program deployed to the cRIO:

1. Undeploy
2. Reboot the cRIO
3. Do debug in Workbench

To deploy a program to the cRIO:

1. Download
2. Reboot the cRIO

## Debugging your robot program

You can monitor, control and manipulate processes using the debugger. This section will describe how to set up a debug session on a robot control program for the cRIO. (See the Wind River Workbench User's Guide for complete documentation on how to use the debugger: Help > Help Contents > Wind River Documentation > Guides > Host Tools > Wind River Workbench User's Guide.)

To run a program that derives from one of the WPILib robot base classes, such as SimpleRobot.cpp or IterativeRobot.cpp, a macro called START\_ROBOT\_CLASS is used that starts one task that just spawns the robot task with the correct run options. See SimpleDemo or IterativeDemo for examples. This makes it necessary to set up Debug to attach to the spawned robot task instead of the initial task.

To start a debug session, connect to the target and click on the bug icon or right click the project and select "Debug Kernel Task..." The Debug dialog is displayed.

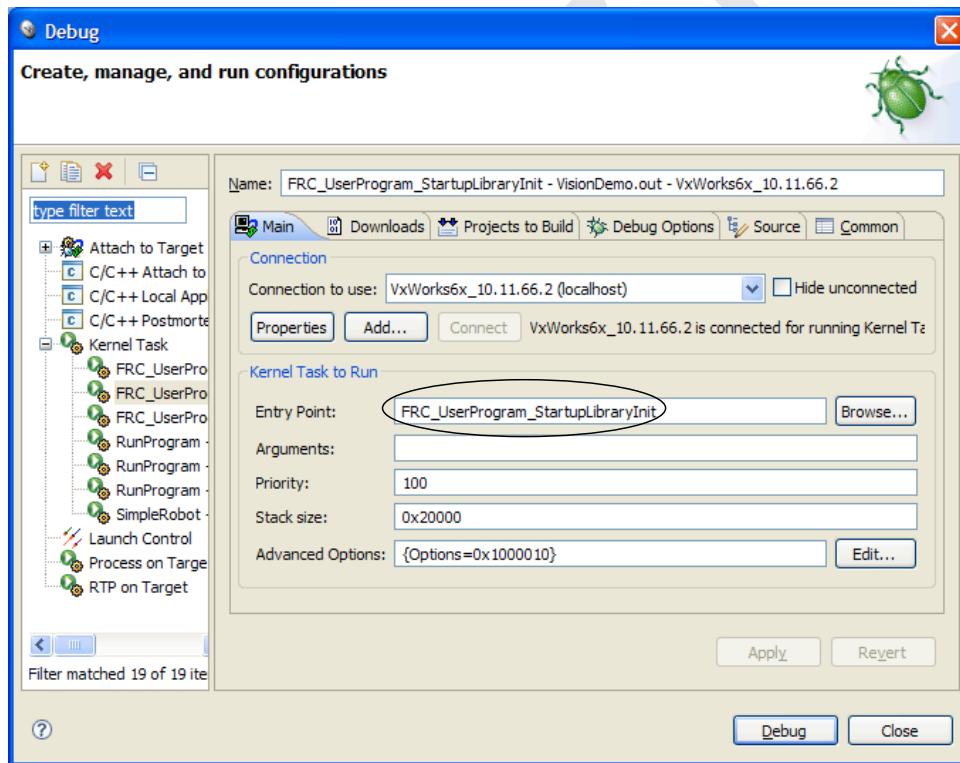


Figure 1: Setting the entry point on a Debug Configuration for a robot program.

Select as the entry point the function FRC\_UserProgram\_StartupLibraryInit. (If you have more than one of them, as in the figure above, you might want to give them names that are distinct in the visible part of the list.) On the debug options tab, select "Break on Entry" and "Automatically attach spawned Kernel

Tasks". This tells the debugger to stop at the first instruction, and to make the spawned task (your robot task) available to debug.

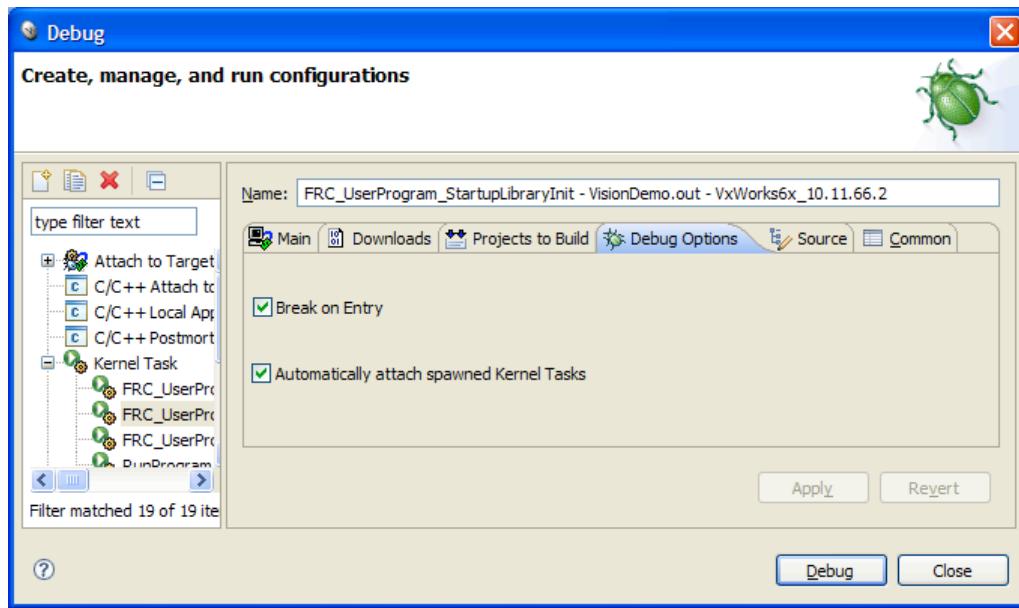
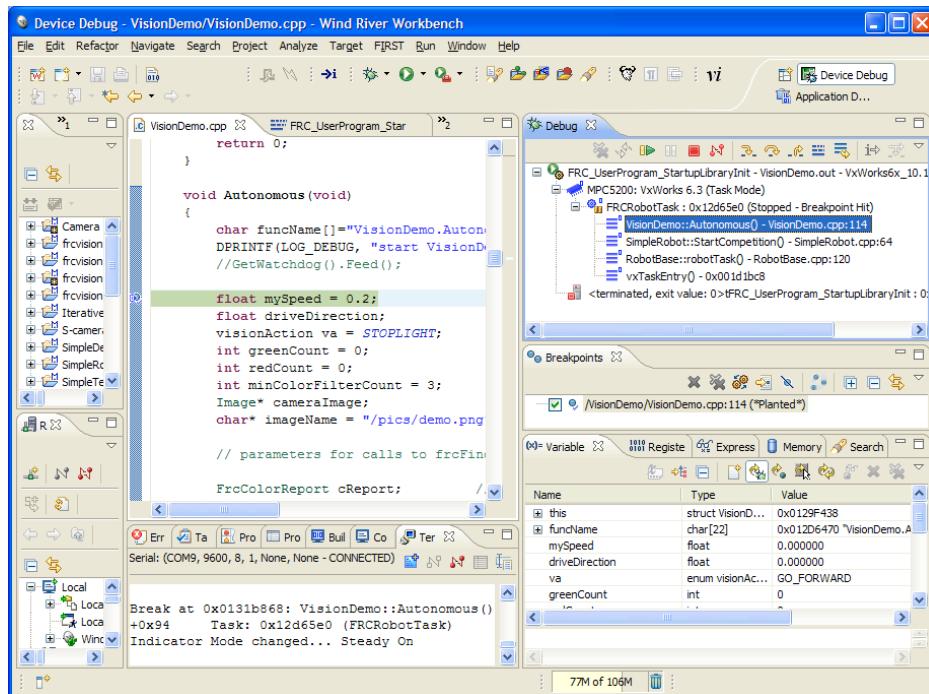


Figure 2: Setting the “Automatically attach spawned Kernel Tasks” option ensures that you will be able to debug the entire program including any tasks that it creates. The WPI Robotics Library automatically starts your program in a new task.

The other tabs can normally be left at default settings.

When the “Debug” button is selected, several things happen. Your Workbench display changes to the Debug Perspective, which has views Debug, Breakpoints and Variables along the right side. The task is kicked off and stops in the first instruction, in `FRC_UserProgram_StartupLibraryInit`. At this point double-click in the left margin of the source code window (VisionDemo.cpp, below) to set a breakpoint in your user program. A little blue circle indicates the breakpoint. Select the “Resume” icon (the green arrow) to continue until the first breakpoint is reached.



The Debug view shows all processes and threads running under the debugger. Selecting the stack frame will show the current instruction pointer and source code (if available) for the process selected. When your breakpoint is reached, make sure your program is selected in the task list, and your source code is displayed with a program pointer. You can continue through your code using “Resume”, “Step Into”, “Step Over” and “Step Return”. If you see assembly code displayed this is because you have gone into a lower level of code where the source is not available. A “Step Return” will bring you back up a level.

The lower right view shows the current value of variables. To see a variable that is not displayed, select the “Expressions” tab and enter the variable name. If it is in scope, its current value will be shown.

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

### Troubleshooting:

**Source code displayed is out of sync with cursor when debugging:** The source has changed since it was loaded onto the cRIO. Rebuild the project (build clean) and make sure included projects are up to date.

**Robot program not visible in the Debug View:** Make sure that “Automatically attach spawned Kernel Tasks” option is set. The first stop happens before your program is started. It will appear after you “Resume.”

### Getting printf/cout output on the PC

There are three ways to see output from printf/cout in Workbench, all with advantages and disadvantages:

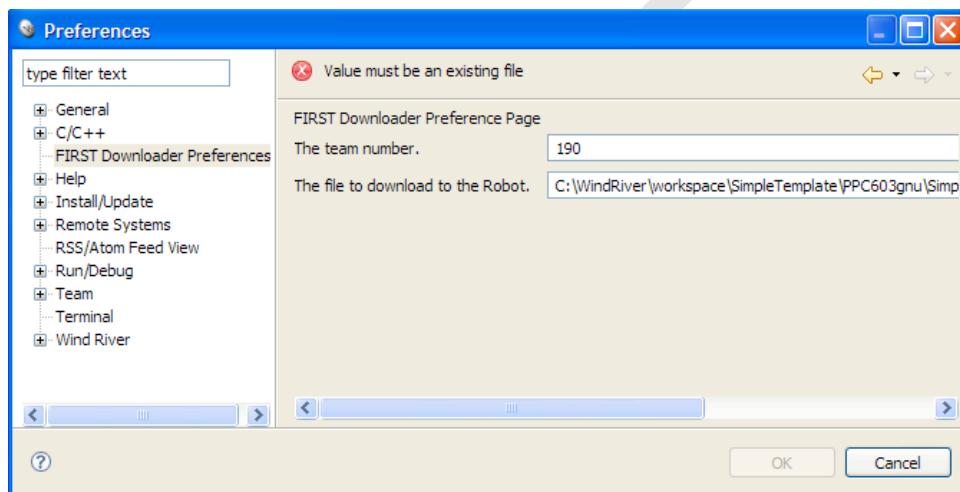
Connect a serial cable between the computer and robot controller	Always works over system reboots	Robot must be tethered
Use a network Target Console. To get that, right-click on the remote system, then "Target Tools", then "Target Console". This will create a console window over the network.	Gets everything with no tether cable	Goes away on reboot
Allocate a console. In the Run menu, select Open Run Dialog... or Open Debug Dialog... to open the run or debug configuration. Select your Kernel Task FRC_UserProgram_StartupLibraryInit in the left pane. Then look at the "Common" tab. There check the "Allocate Console" checkbox.	Survives reboots	Seems to only print current task output

## Deploying the C/C++ Program

Deploying a program to the cRIO copies it to flash memory so that it will automatically start when the robot is turned on. The FIRST Downloader plug-in for Workbench has commands to Download (i.e. deploy) the program to the cRIO and Undeploy (i.e. delete) the program from the cRIO.

Note: Don't try to debug a robot program as described above while there is a deployed robot program that automatically runs when the cRIO boots.

To set up Workbench to deploy your program use the menu command "Window > Preferences... > FIRST Downloader Preferences".



Fill in your team number and the .OUT file for your project that should be loaded. The .OUT file will typically be in the PPC603gnu directory in the Workbench workspace directory for your project—this assumes you already built it.

Once this is set up, to deploy the project use the menu command "FIRST > Download". This will copy the program to the correct filename and directory in the cRIO. The next time the cRIO is restarted, the program will start running.

To undeploy the project, use the menu command "FIRST > Undeploy".

# Creating a Robot Program

Now consider a very simple robot program that has these characteristics:

- Autonomous period** Drives in a square pattern by driving half speed for 2 seconds to make a side then turns 90 degrees. This is repeated 4 times.
- Operator Control period** Uses two joysticks to provide tank steering for the robot.

The robot specifications are:

<b>Left drive motor</b>	PWM port 1
<b>Right drive motor</b>	PWM port 2
<b>Joystick</b>	driver station joystick port 1

Starting with the template for a simple robot program we have:

```
#include "WPILib.h"
class RobotDemo : public SimpleRobot
{
public:
    RobotDemo()
    {
        // put initialization code here
    }

    void Autonomous()
    {
        // put autonomous code here
    }

    void OperatorControl()
    {
        // put operator control code here
    }
};

START_ROBOT_CLASS(RobotDemo);
```

Now add objects to represent the motors and joystick.

The robot drive object with motors in ports 1 and 2, and two joystick objects are declared using the following code:

```
RobotDrive drive(1, 2);
Joystick leftStick(1);
Joystick rightStick(2);
```

For the example and to make the program easier to understand, we'll disable the watchdog timer. This is a feature in the WPI Robotics Library that helps ensure that your robot doesn't run off out of control if the program malfunctions.

```
RobotDemo ()  
{  
    GetWatchdog () . SetEnabled ( false ) ;  
}
```

Now the autonomous part of the program can be constructed that drives in a square pattern:

```
void Autonomous ()  
{  
    for ( int i = 0; i < 4; i ++ )  
    {  
        drivetrain . Drive ( 0.5, 0.0 ); // drive 50% of full forward with 0% turn  
        Wait ( 2.0 ); // wait 2 seconds  
        drivetrain . Drive ( 0.0, 0.75 ); // drive 0% forward and 75% turn  
    }  
    Drivetrain . Drive ( 0.0, 0.0 ); // drive 0% forward, 0% turn (stop)  
}
```

Now look at the operator control part of the program:

```
void OperatorControl ()  
{  
    while ( 1 ) // loop forever  
    {  
        drivetrain . TankDrive ( leftStick, rightStick ); // drive with the joysticks  
        Wait ( 0.005 );  
    }  
}
```

Putting it all together we get this pretty short program that accomplishes some autonomous task and provides operator control tank steering:

```
#include "WPILib.h"

class RobotDemo : public SimpleRobot
{
    RobotDrive drivetrain(1, 2);
    Joystick leftStick(1);
    Joystick rightStick(2);

public:
    RobotDemo()
    {
        GetWatchdog().SetEnabled(false);
    }

    void Autonomous()
    {
        for (int i = 0; i < 4; i++)
        {
            drivetrain.Drive(0.5, 0.0); // drive 50% forward, 0% turn
            Wait(2.0); // wait 2 seconds
            drivetrain.Drive(0.0, 0.75); // drive 0% forward and 75% turn
            Wait(0.75); // turn for almost a second
        }
        drivetrain.Drive(0.0, 0.0); // stop the robot
    }

    void OperatorControl()
    {
        while (1) // loop forever
        {
            drivetrain.Tank(leftStick, rightStick); // drive with the joystick
            Wait(0.005);
        }
    }
};

START_ROBOT_CLASS(RobotDemo);
```

Although this program will work perfectly with the robot as described, there were some details that were skipped:

- In the example `drivetrain`, `leftStick` and `rightStick` are member objects of the `RobotDemo` class. In the next section pointers will be introduced as an alternate technique.
- The `drivetrain.Drive()` method takes two parameters, a speed and a turn direction. See the documentation about the `RobotDrive` object for details on how that speed and direction really work.

## Pointers and addresses

There are two ways of declaring objects: either as an instance of the object or a pointer to the object. In the case of the instance the variable represents the object, and it is created at the time of the declaration. In the case of a pointer you are only creating the space to store the address of the object, but the object remains uncreated. With pointers you have to create the object using the `new` operator. Look at these two snippets of code to see the difference.

```
Joystick stick1(1);    // this is an instance of a Joystick object stick1
stick1.GetX();          // instance dereferenced using the dot (.) operator
bot->ArcadeDrive(stick1); // and can be passed to methods as a reference

Joystick *stick2;      // a pointer to an uncreated Joystick object
stick2 = new Joystick(1); // creates the instance of the Joystick object
stick2->GetX();        // pointers are dereferenced with the arrow (->)
bot->ArcadeDrive(stick2); // and can be passed as pointers (notice, no &)
delete stick2;          // delete the object when you're done with it
```

The `ArcadeDrive` method in the library is taking advantage of a feature of C++ called function overloading. This allows us to have two methods with the same name that differ by the argument list. In the first `ArcadeDrive(stick1)`, the variable `stick1` is passed as a reference to a `Joystick` object. In the second `ArcadeDrive(stick2)`, it is being passed as a pointer to a `Joystick` object. There are actually two methods in the `RobotDrive` object, both called `ArcadeDrive` that each take a different type of argument. The cool thing is that the compiler figures out which one to call. The library is built this way to let it adapt to the style of programmers that prefer to use pointers while at the same time accommodating those who prefer to use references.

## Built-in Robot classes

There are several built-in robot classes that will help you quickly create a robot program. These are:

**Table 1: Built-in robot base classes to create your own robot program. Subclass one of these depending on your requirements and preferences.**

<b>SimpleRobot</b>	<p>This template is the easiest to use and is designed for writing a straight-line autonomous routine without complex state machines.</p> <p>Pros:</p> <ul style="list-style-type: none"> <li>• Only three places to put your code: the constructor for initialization, the Autonomous method for autonomous code and the OperatorControl method for teleop code.</li> <li>• Sequential robot programs are trivial to write, just code each step one after another.</li> <li>• No state machines required for multi-step operations, the program can simply do each step sequentially.</li> </ul> <p>Cons:</p> <ul style="list-style-type: none"> <li>• Automatic switching between Autonomous and Teleop code segments is not easy and may require rebooting the controller.</li> <li>• The Autonomous method will not quit running until it exits, so it will continue to run through the TeleOp period unless it finishes by the end of the Autonomous period (so be sure to make your loops check that it's still the autonomous period).</li> </ul>
<b>IterativeRobot</b>	<p>This template gives additional flexibility in the code for responding to various field state changes (autonomous, teleoperated, disabled) in exchange for additional complexity in the program design. It is based on a set of methods that are repeatedly called based on the current state of the field. The intent is that each method is called; it does some processing, and then returns. That way, when the field state changes, a different method can be called as soon as the change happens.</p> <p>Pros:</p> <ul style="list-style-type: none"> <li>• Can have very fine-grain control of field state changes, especially if practicing and retesting the same state over and over.</li> </ul> <p>Cons:</p> <ul style="list-style-type: none"> <li>• More difficult to write action sequences that unfold over time. It requires state variables to remember what the robot is doing from one call the next.</li> </ul>
<b>RobotBase</b>	<p>The base class for the above classes. This provides all the basic functions for field control, the user watchdog timer, and robot status. This class should be extended to have the required specific behavior.</p>

## SimpleRobot class

The **SimpleRobot** class is designed to be the base class for a robot program with straightforward transitions from Autonomous to Operator Control periods. There are three methods that are usually filled in to complete a SimpleRobot program.

**Table 2: SimpleRobot class methods that are called as the match moves through each phase.**

<b>the Constructor (method with the same name as the robot class)</b>	Put all the code in the constructor to initialize sensors and any program variables that you have. This code runs as soon as the robot is turned on, but before it is enabled. When the constructor exits, the program waits until the robot is enabled.
<b>Autonomous()</b>	All the code that should run during the autonomous period of the game goes in the Autonomous method. The method is allowed to run to completion and will not be interrupted at the end of the autonomous period. If the method has an infinite loop, it will never stop running until the entire match ends. When the method exits, the program will wait until the start of the operator control period.
<b>OperatorControl()</b>	Put code in the OperatorControl method that should run during the operator control part of the match. This method will be called after the Autonomous() method has exited and the field has switched to the operator control part of the match. If your program exits from the OperatorControl() method, it will not resume until the robot is reset.

## IterativeRobot class

The IterativeRobot class divides your program up into methods that are repeatedly called at various times as the robot program executes. For example, the AutonomousContinuous() method is called continually during the autonomous period. When the playing field (or the switch on the DS) changes to operator control, then the TeleopInit() first, then the TeleopContinuous() methods are called continuously.

WindRiver Workbench has a built in sample robot program based on the Iterative Robot base class. If you would like to use it, follow the instructions from the previous section, except select “Iterative Robot Main Program”. The project will be created in your workspace.

The methods that the user fills in when creating a robot based on the IterativeRobot base class are:

**Table 3: IterativeRobot class methods that are called as the match proceeds through each phase.**

<b>RobotInit</b>	Called when the robot is first turned on. This is a substitute for using the constructor in the class for consistency. This method is only called once.
<b>DisabledInit</b>	Called when the robot is first disabled
<b>AutonomousInit</b>	Called when the robot enters the autonomous period for the first time. This is called on a transition from any other state.
<b>TeleopInit</b>	Called when the robot enters the teleop period for the first time. This is called on a transition from any other state.
<b>DisabledPeriodic</b>	Called periodically during the disabled time based on a periodic timer for the class.
<b>AutonomousPeriodic</b>	Called periodically during the autonomous part of the match based on a periodic timer for the class.
<b>TeleopPeriodic</b>	Called periodically during the teleoperation part of the match based on a periodic timer for the class.
<b>DisabledContinuous</b>	Called continuously while the robot is disabled. Each time the program returns from this function, it is immediately called again provided that the state hasn't changed.
<b>AutonomousContinuous</b>	Called continuously while in the autonomous part of the match. Each time the program returns from this function, it is immediately called again provided that the state hasn't changed.
<b>TeleopContinuous</b>	Called continuously while in the teleop part of the match. Each time the program returns from this function, it is immediately called again provided that the state hasn't changed.

Jerry Morrison 1/10/09 9:20 PM

**Comment:** if the robot enters autonomous mode a second time, won't this get called again?

The three Init methods are called only once each time state is entered. The Continuous methods are called repeatedly while in that state, after calling the appropriate Init method. The Periodic methods are called periodically while in a given state where the period can be set using the SetPeriod method in the IterativeRobot class. The periodic methods are intended for timebased algorithms like PID control. Any of the provided methods will be called at the appropriate time so if there is a TeleopPeriodic and TeleopContinous, they will both be called (although at different rates).

# WPI Robotics Library Conventions

This section documents some conventions that were used throughout the library to standardize on its use and make things more understandable. Knowing these should make your programming job much easier.

## Class, method, and variable naming

Names of things follow the following conventions:

Type of name	Naming rules	Examples
<b>Class name</b>	Initial upper case letter then camel case (mixed upper/lower case) except acronyms which are all upper case	Victor, SimpleRobot, PWM
<b>Method name</b>	Initial upper case letter then camel case	StartCompetition, Autonomous, GetAngle
<b>Member variable</b>	"m_" followed by the member variable name starting with a lower case letter then camel case	m_deleteSpeedControllers, m_sensitivity
<b>Local variable</b>	Initial lower case	targetAngle
<b>Macro</b>	All upper case with _ between words, but you should use const values and inline function instead of macros.	DISALLOW_COPY_AND_ASSIGN

## Constructors with slots and channels

Most constructors for physical objects that connect to the cRIO take the port number in the constructor. The following conventions are used:

- Specification of an I/O port consists of the slot number followed by the channel number. The slot number is the physical slot on the cRIO chassis that the module is plugged into. For example, for Analog modules it would be either 1 or 2. The channel number is a number from 1 to n, where n is the number of channels of that type per I/O module.
- Since many robots can be built with only a single analog or digital module, there is a shorthand method of specifying port. If the port is on the first (lowest numbered) module, the slot parameter can be left out.

Examples are:

```
Jaguar(UINT32 channel);           // channel with default slot (4)
Jaguar(UINT32 slot, UINT32 channel); // channel and slot
Gyro(UINT32 slot, UINT32 channel);  // channel with explicit slot
Gyro(UINT32 channel);             // channel with default slot (1)
```

### Sharing inputs between objects

WPILib constructors for objects generally use port numbers to select input and output channels on cRIO modules. When you use a channel number in an object like an encoder, a digital input is created inside the encoder object reserving the digital input channel number.

DRAFT

## RobotBase class

The RobotBase class is the subclass for the SimpleRobot and IterativeRobot classes. It is intended that if you decide to create your own type or robot class it will be based on RobotBase. RobotBase has all the methods to determine the field state, set up the watchdog timer, communications, and other housekeeping functions.

To create your own base class, create a subclass of RobotBase and implement (at least) the StartCompetition() method.

For example, the SimpleRobot class definition looks (approximately) like this:

```
class SimpleRobot: public RobotBase
{
public:
    SimpleRobot();
    virtual void Autonomous();
    virtual void OperatorControl();
    virtual void RobotMain();
    virtual void StartCompetition();

private:
    bool m_robotMainOverridden;
};
```

It overrides the `StartCompetition()` method that controls the running of the other methods and it adds the `Autonomous()`, `OperatorControl()`, and `RobotMain()` methods. The StartCompetition method looks (approximately) like this:

```
void SimpleRobot::StartCompetition()
{
    while (IsDisabled()) Wait(0.01); // wait for match to start
    if (IsAutonomous())           // if starts in autonomous
    {
        Autonomous();           // run user-supplied Autonomous code
    }
    while (IsAutonomous()) Wait(0.01); // wait until end of autonomous period
    while (IsDisabled()) Wait(0.01); // make sure robot is enabled
    OperatorControl();           // start user-supplied OperatorControl
}
```

It uses the `IsDisabled()` and `IsAutonomous()` methods in RobotBase to determine the field state and calls the correct methods as the match is sequenced.

Similarly the IterativeRobot class calls a different set of methods as the match progresses.

## Watchdog timer class

The Watchdog timer class helps to ensure that the robot will stop operating if the program does something unexpected or crashes. A watchdog object is created inside the RobotBase class (the base class for all the robot program templates). Once created, the robot program is responsible for “feeding” the watchdog periodically by calling the `Feed()` method on the Watchdog. Failure to feed the Watchdog results in all the motors and pneumatics stopping on the robot.

The default expiration time for the Watchdog is 500ms (0.5 second). Programs can override the default expiration time by calling the `SetExpiration(expiration-time-in-seconds)` method on the Watchdog.

Use of the Watchdog timer is recommended for safety, but it can be disabled. For example, during the autonomous period of a match the robot needs to drive for 2 seconds then make a turn. The easiest way to do this is to start the robot driving, and then use the `Wait` function for 2 seconds. During the 2 second period when the robot is in the `Wait` function, there is no opportunity to feed the Watchdog. In this case you could disable the Watchdog at the start of the `Autonomous()` method and re-enable it at the end. *Alternatively a longer watchdog timeout period would still provide much of the protection from the watchdog timer.*

```
void Autonomous()
{
    GetWatchdog().SetEnabled(false); // disable the watchdog timer
    Drivetrain.Drive(0.75, 0.0);    // drive straight at 75% power
    Wait(2.0);                   // wait for 2 seconds

    .

    .

    GetWatchdog().SetEnabled(true); // reenable the watchdog timer
}
```

You can call `GetWatchdog()` from any of the methods inside one of the robot program template objects.

# 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

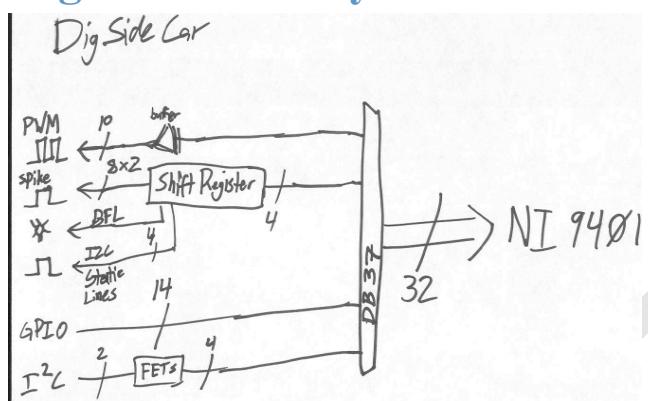
The library natively supports sensors of a number of categories shown below.

<b>Wheel/motor position measurement</b>	<a href="#">Geartooth sensors</a> , <a href="#">encoders</a> , <a href="#">analog encoders</a> , and <a href="#">potentiometers</a>
<b>Robot orientation</b>	<a href="#">Compass</a> , gyro, accelerometer, ultrasonic rangefinder
<b>Generic pulse output</b>	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 I<sup>2</sup>C 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 those classes. Using these other supported device types (encoder, ultrasonic rangefinder, gear tooth sensor, etc.) don'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. To use one of those lines as an input the direction must be set. 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.

## 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 `DigitalOutput` object will automatically set the direction of the GPIO line to output.

DRAFT

## 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 its side would indicate 1000 milliGs or one G.

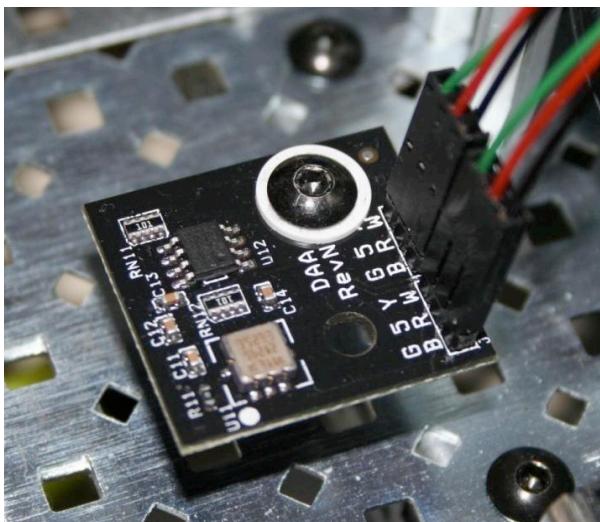


Figure 3: FRC supplied 2 axis accelerometer board connected to a robot

## 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/<sup>o</sup>/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 means that the robot must be stationary while this is happening and that the gyro is unusable until after it has completed the calibration.

Once initialized, the **GetAngle()** 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()** method on the Gyro object.

### Setting the gyro sensitivity

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

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

### 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),      // initialize the 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
    }
};

```

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.

### Example

Create a compass on the I2C port of the digital module plugged into slot 4.

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

## 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.

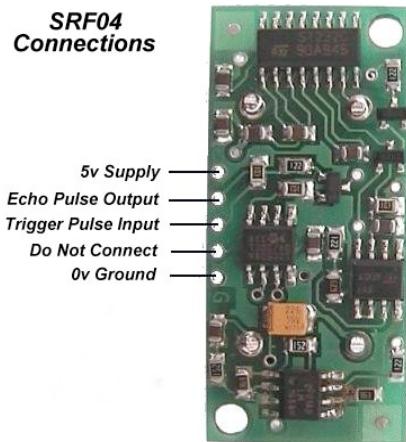


Figure 4: SRF04 Ultrasonic Rangefinder connections

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:

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

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.

### Example

Reads the range on an ultrasonic sensor connected to the output port ULTRASONIC\_PING and the input port ULTRASONIC\_ECHO.

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

## 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.

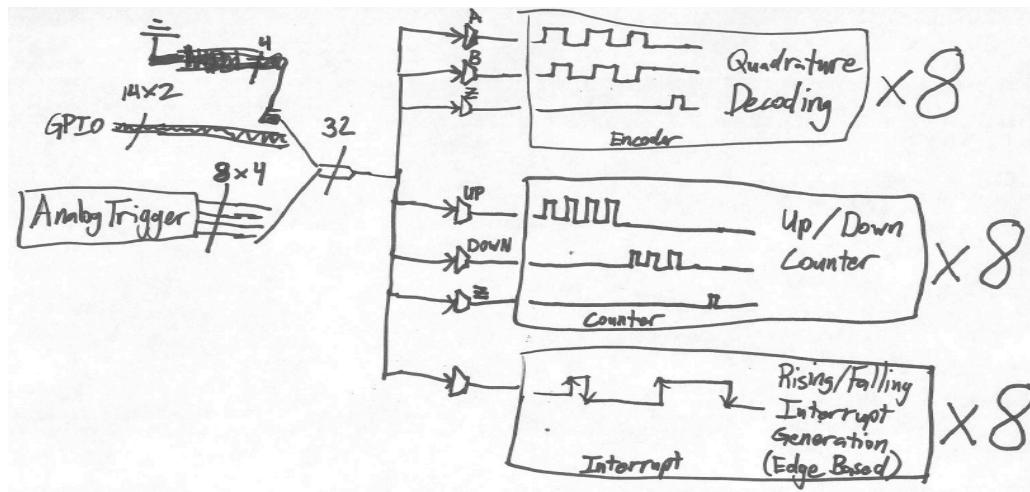


Figure 5: Schematic of the possible sources and counters in the Counter Subsystem in the cRIO.

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.

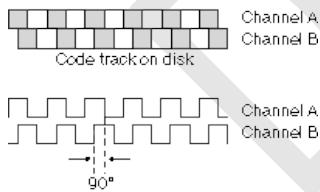
# Encoders

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 WPILib.

	Simple encoders Counter class	Quadrature encoders Encoder class	Gear tooth sensor GearTooth 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 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 counterclockwise 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.	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 4: Encoder types that are supported by WPILib**

These types of encoders are described in the following sections.



**Figure 6: 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.

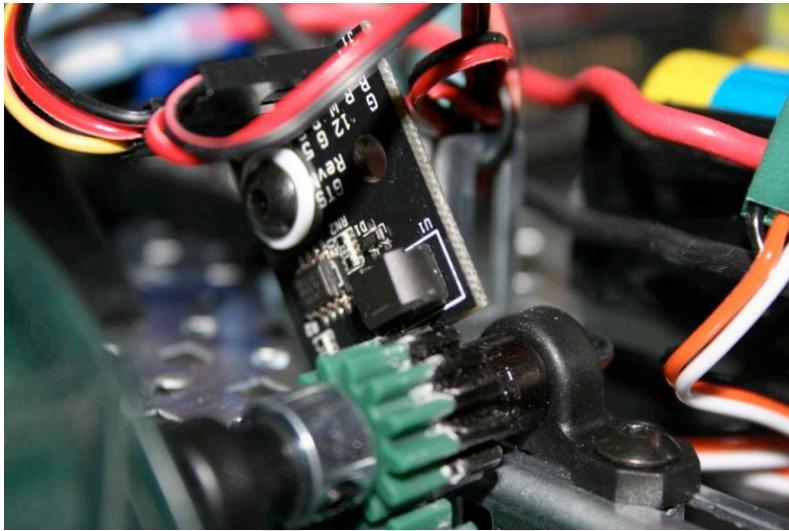


Figure 7: A gear tooth sensor mounted on a VEX robot chassis measuring a metal gear rotation. Notice that there is a metal gear attached to the plastic gear in this picture. The gear tooth sensor needs a ferrous material passing by it to detect rotation.

### 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).

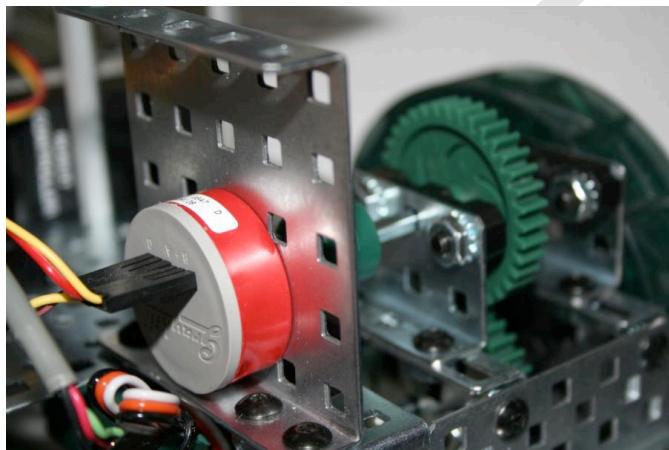


Figure 8: A Grayhill quadrature optical encoder. Note the two connectors, one for the A channel and one for the B channel.

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.

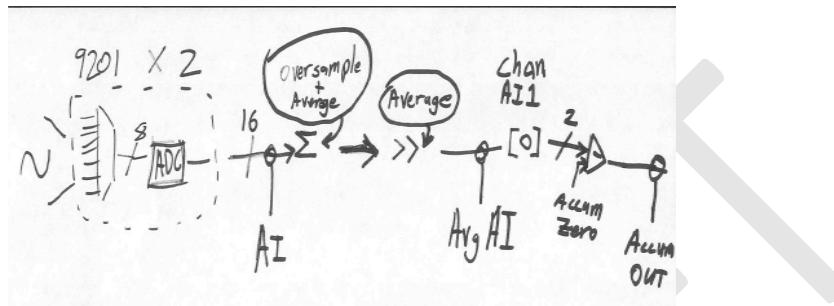
#### Example

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

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

$$\text{Avg AI} = \frac{\sum_{o=0}^{2^M-1} \left( \sum_{x=0}^{2^N-1} (\text{AI}_{x,o}) \right)}{2^M}$$

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

$N$  = oversample bits

$M$  = average bits

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

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

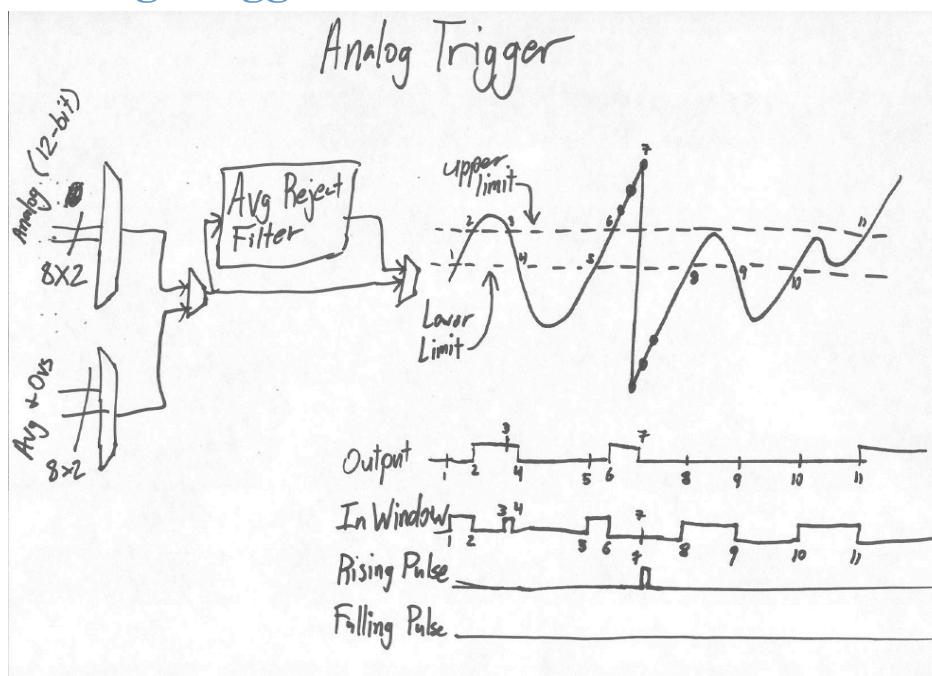
The number of averaged and oversampled values are always powers of 2 (number of bits of oversampling/averaging). Therefore the number of oversampled or averaged values is  $2^{\text{bits}}$ , where bits is passed to the methods: **SetOversampleBits (bits)** and **SetAverageBits (bits)**. The actual rate 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  $2^{4+2}$ , 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

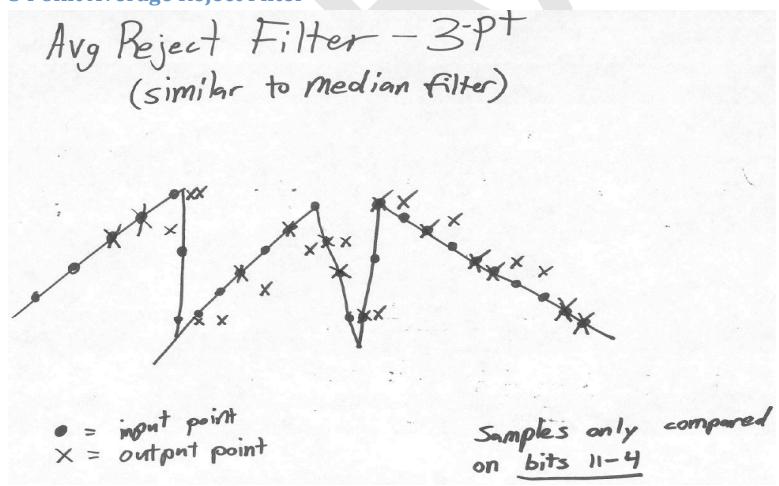
- There is one sample rate per module.
- The number of oversampled and averaged values is expressed as a power of 2.
- The delivered sample rate is reduced by the oversample and average values.
- There are 2 accumulators connected to analog channels 1 and 2 of the first Analog Module. This means that 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  $2^n$  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)



# Camera

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:

```
// 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:

```
#####
! 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

```
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.

```

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_90, ROT_180, ROT_270

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:

```

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.

```

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_SENDIMAGE_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:

```
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.

```
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 this group:

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 teleoperated 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 which set the update rate, deadband elimination, and profile shaping of the output signal.

DRAFT

## 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
<b>Max</b>	The maximum value where the motors stop changing speed and the light on the Victor goes to full green.
<b>DeadbandMax</b>	The value where the motor just stops operating.
<b>Center</b>	The value that is in the center of the deadband that turns off the motors.
<b>DeadbandMin</b>	The value where the motor just starts running in the opposite direction.
<b>Min</b>	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.

```
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

DRAFT

# 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.

## Example 1

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
}
```

## Example 2

The following code fragment pans a servo back and forth every 3 seconds

```
#include "BaeUtilities.h"

panInit(); // optional parameters can adjust pan speed
bool targetFound = false;

while (!targetFound) {
    panForTarget(servo, 0.0);      // sinStart from -1 to +1
    // code to identify target
}
```

Jerry Morrison 1/11/09 3:07 PM  
Comment: What's sinStart?

## 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.

### Example

```
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 operation:

Method	Description
<b>Drive(speed, turn)</b>	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.
<b>TankDrive(leftStick, rightStick)</b>	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.
<b>ArcadeDrive(stick)</b>	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.
<b>HolonomicDrive(magnitude, direction, rotation)</b>	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.
<b>SetLeftRightMotorSpeeds(leftSpeed, rightSpeed)</b>	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

# Controlling Pneumatics

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

Class	Purpose
<b>Solenoid</b>	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.)
<b>Compressor</b>	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.

## 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.

```
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.

```
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.

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

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

## 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.

```
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 nivision 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.

Jerry Morrison 1/13/09 1:46 AM

**Comment:** this doc should define "particles"

# 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.

```
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_normalized);
    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:

```
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);
```

Jerry Morrison 1/14/09 12:20 AM

**Comment:** If it's a float from -1.0 to 1.0, then the %i printf conversion won't work.

Jerry Morrison 1/14/09 12:22 AM

**Comment:** What are the supported ranges for H, S, and L?

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:

```
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.

```
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 environment
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 `center_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.

```
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:

```
// 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

```
// find a two color target
if (FindTwoColors(td1, td2, ABOVE, &par1, &par2) {
    // 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_normalized) / 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 parameters 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.

# Concurrency

VxWorks is the operation system that is running inside the cRIO and providing services to the running robot programs that you write. It provides many operations to support **concurrency**, or the simultaneous execution of multiple pieces of the program called **tasks**. Each task is scheduled to run by VxWorks based on its priority and availability of resources it might be waiting on. For example, if one task calls `Wait(time)`, then other tasks can run until the time runs out on the waiting task.

WPILib provides some classes to help simplify writing programs that do multitasking. However it should be stressed that writing multi-tasking code represents one of the most challenging aspects of programming. It may look simple; but there are many complications that could give your program unexpected and hard to reproduce errors.

DRAFT

## Creating tasks

In your program you may decide to subdivide the work into multiple concurrently running tasks. For example, you may have a requirement to operate a ball loading mechanism independently of a ball shooter and the driving code in your robot. Each of these functions can be split out into its own task to simplify the overall design of the robot code.

DRAFT

## Synchronized and Critical Regions

A critical region is an area of code that is always executed under mutual exclusion, i.e. only one task can be executing this code at any time. When multiple tasks try to manipulate a single group of shared data they have to be prevented from executing simultaneously otherwise a race condition is possible. Imagine two tasks trying to update an array at the same time. Task A reads the count of elements in the array, then task B changes the count, then task A tries to do something based on the (now incorrect) value of the count it previously read. This situation is called a race condition and represents one of the most difficult to find programming bugs since the bug only is visible when the timing of multiple tasks is just right (or wrong). It's called a "race" because the result depends on which task finishes first. It's difficult to find because the bug symptoms happen inconsistently.

Typically semaphores are used to ensure only one task has access to the shared data at a time. Semaphores are operating system structures that control access to a shared resource. VxWorks provides two operations on semaphores, **take** and **give**. When you take a semaphore, the code pauses until the semaphore isn't in use by another task, then the operating system marks it in use, but your code can now run. You give the semaphore when you are finished using the shared data. It now lets the next task trying to take the semaphore run.

Suppose that a function operates on some shared data. Understanding about the bad things that can happen with race conditions, you take a semaphore at the start of the function and give it at the end. Now inside the function, the data is protected from inappropriate shared use (that is, assuming that all functions that accesses this shared data take the semaphore first). Now someone else looks at the code and decides to change it and puts a return in the middle of the function, not noticing the take and give. The semaphore is taken, but the corresponding give operation never happened. That means that any other task waiting on that semaphore will wait forever. This condition is called **deadlock**.

### Example

The Synchronized object is a simple wrapper around semaphores to avoid this kind of deadlock. Here is an example of how it is used:

```
{
    Synchronized s(semaphore);
    // access shared code here
    if (condition) return;
    // more code here
}
```

At the start of the block a Synchronized object is allocated. This takes the semaphore. When the block exits, the object is automatically freed and its destructor is called. Inside the destructor the semaphore is given. Notice that the destructor will be called no matter how the block is exited. Even if a return is used inside the block, the destructor is guaranteed to be called by the C++ compiler. This eliminates a common cause of deadlock.

To make the code even more readable, there are two macros defined by WPILib and used like this:

```
CRITICAL_REGION(semaphore)
{
    // access shared code here
    if (condition) return;
    // more code here
}
END_REGION;
```

These macros just make the code more readable, but the expanded code is identical to the previous example.

There are other ways to deadlock. In particular, what happens if Task A takes semaphore X and calls a function that takes semaphore Y, while Task B takes the two semaphores in the reverse order, Y then X? This is another kind of race. If sometime A takes X, then B takes Y, then A will wait forever trying to take Y while B waits forever trying to take X.

The simplest solution to this problem is to only use leaf semaphores, that is, never take a semaphore while holding another semaphore. Just take the semaphore, access the shared data quickly, don't call any functions that might take other semaphores, and give back the semaphore. A more complex solution is to make all code take semaphores in the same order, X then Y.

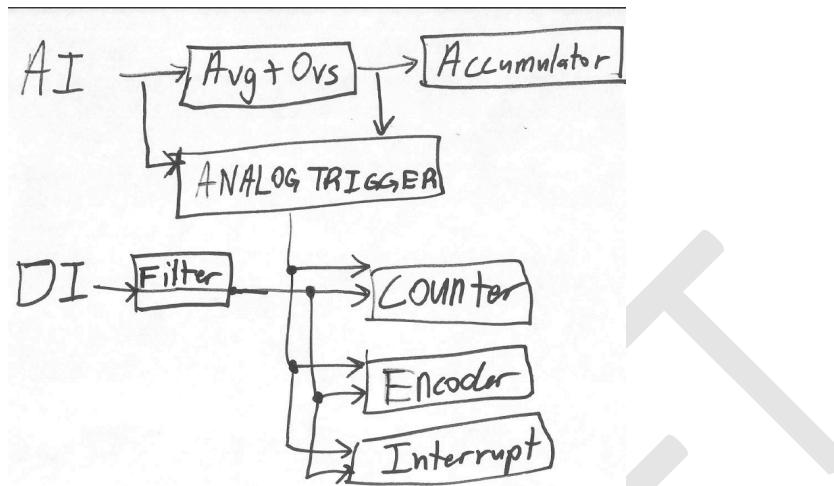
# System Architecture

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.

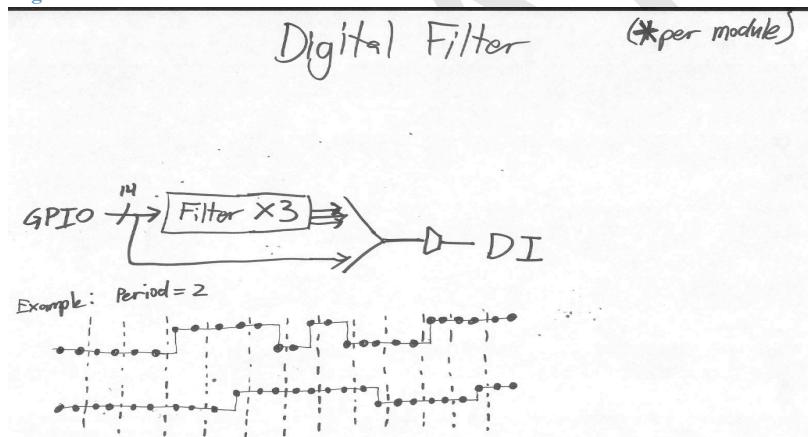
*Note: This is a work in progress, the pictures will be cleaned up and explanations will be soon added. We wanted to make this available to you in its raw form rather than leaving it out all together.*

DRAFT

## Digital Sources



Digital Filter



# 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:

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

*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 DriverStation features

The DriverStation 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

DRAFT

## 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.

```
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 axes, 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.GetAxis(); the twist and throttle axes are accessed by driveJoy.GetTwist() and driveJoy.GetThrottle(), respectively.

Alternatively, the axes can be accessed via 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.

#### Example

```

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());
}

```



# Advanced Programming Topics

DRAFT

# Using Subversion with Workbench

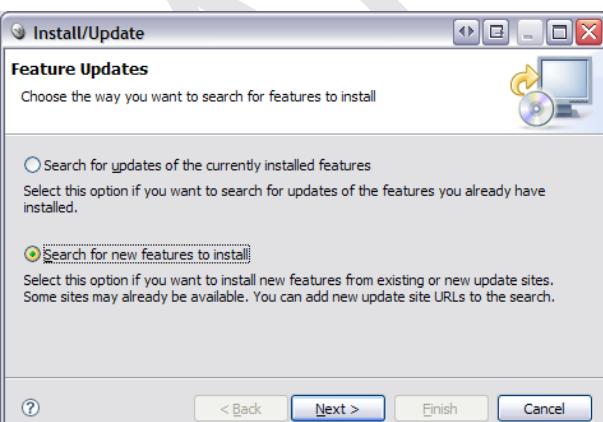
Subversion is a free source code management tool that is designed to track changes to a project as it is developed. You can save each revision of your code in a repository, go back to a previous revision, and compare revisions to see what changed. You should install a Subversion client if:

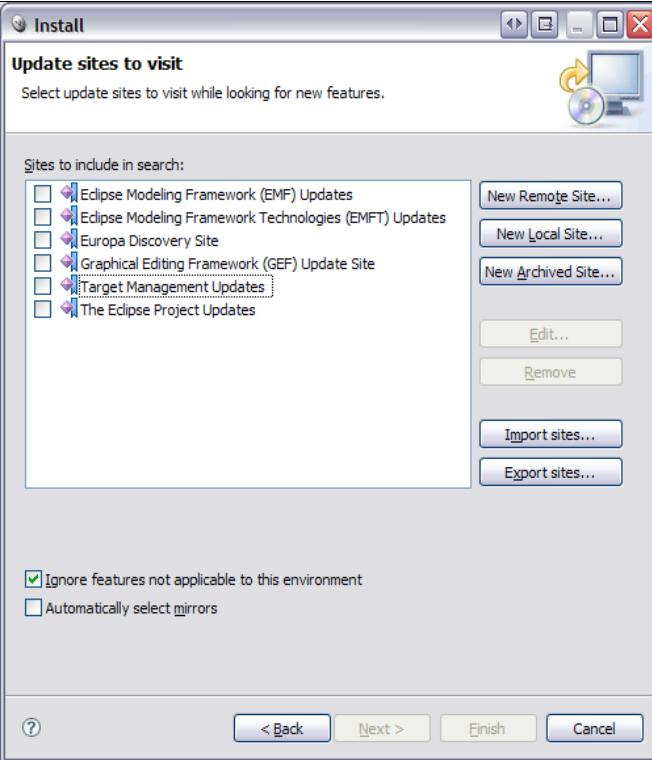
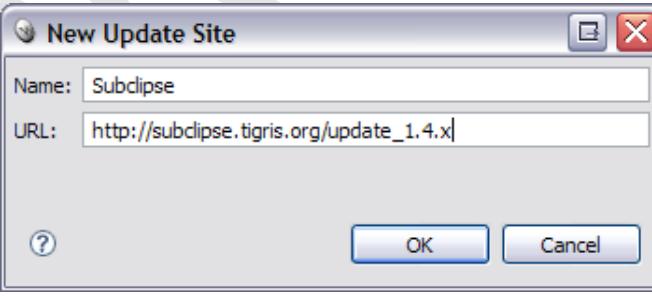
- You need access to the WPI Robotics Library source code installed on a Subversion server
- You have your own Subversion server for working with your team projects

There are a number of clients that will integrate with Workbench, but we've been using Subclipse.

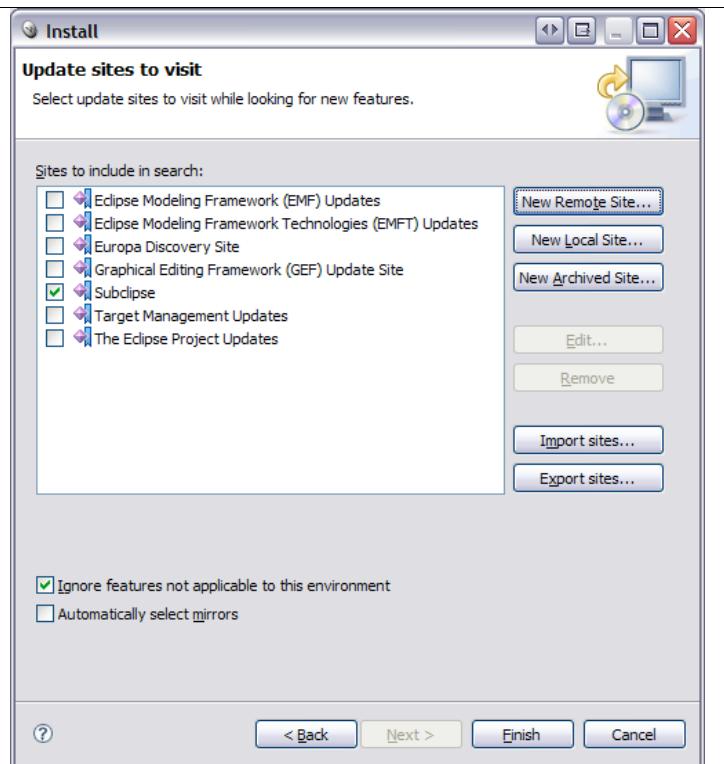
## Installing the Subclipse client into Workbench

Subclipse can be downloaded from the internet and installed into Workbench. The following instructions describe how to do it.

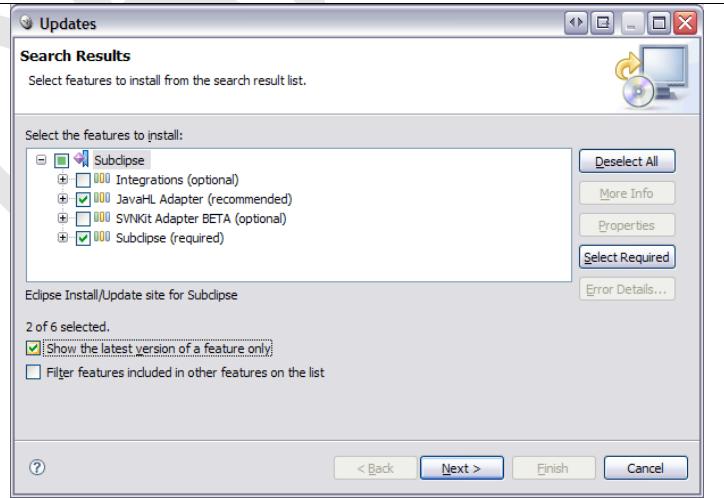
On the help menu, select "Software updates", then "Find and Install".	
Select "Search for new features to install and click Next."	

Click, “New Remote Site...” to add the Subclipse update site.	
Enter the information for the update site and click OK.	

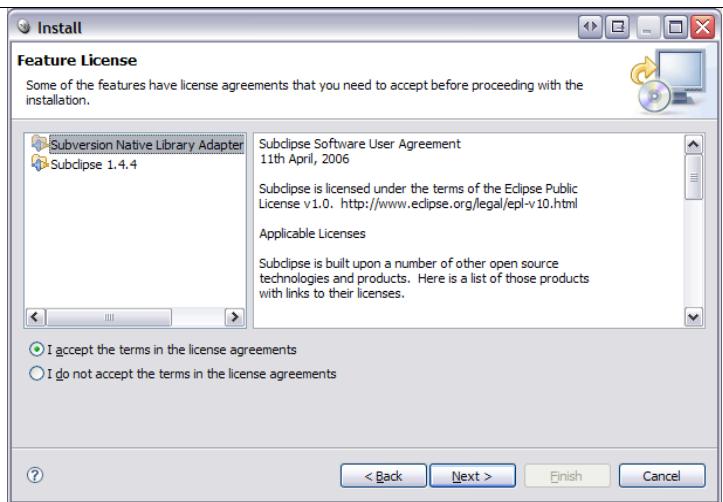
Now you should see Subclipse added to the list of “Sites to include in search.”. Click “Finish”.



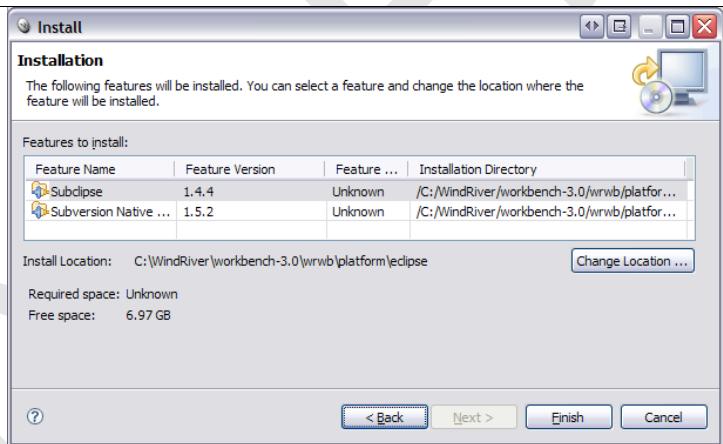
Select the “JavaHL Adapter” and “Subclipse” from the list of features to install.



Accept the license and click “Next”.



Click “Finish” and the install will start. If asked, select “Install All” in the Verification window. You should allow Workbench to restart after finishing.



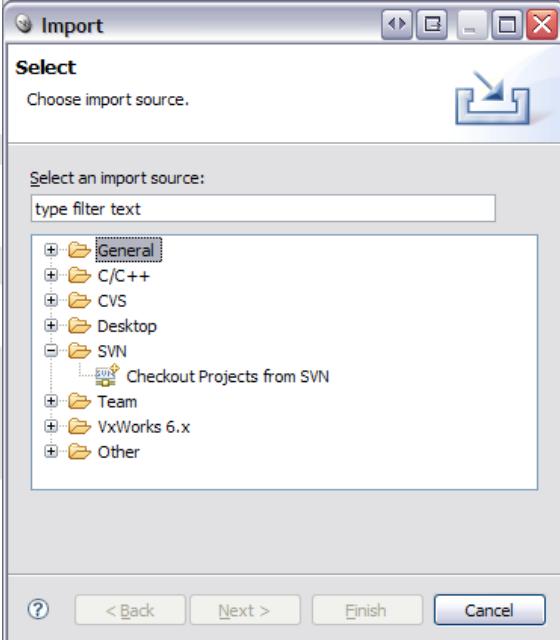
# Getting the WPILib Source Code

The WPI Robotics Library source code is installed on a Subversion server. To get it requires having a subversion client installed in your copy of Workbench. See [Installing the Subclipse client into Workbench](#) for instructions on how to set it up.

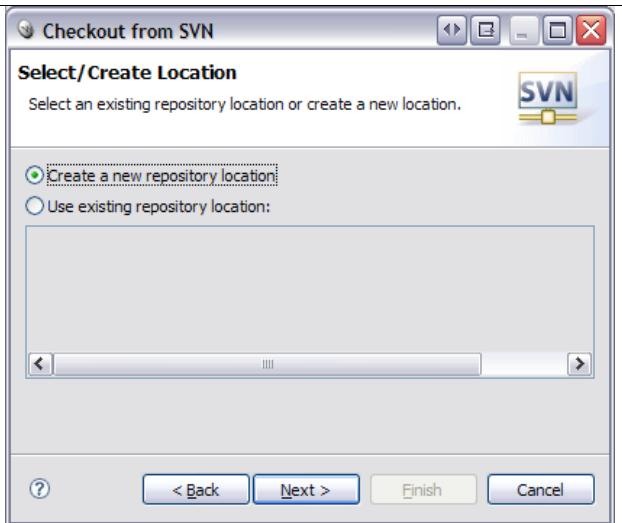
Note: These examples show urls to an internal WPI SourceForge server and it is not available for teams to use. There will be a new server available soon, and we will post the details at that time. For now, the source code is available on the WPILib C/C++ update page. Look at the *FIRST* web site for a link to that page.

## Importing the WPI Robotics Library into your workspace

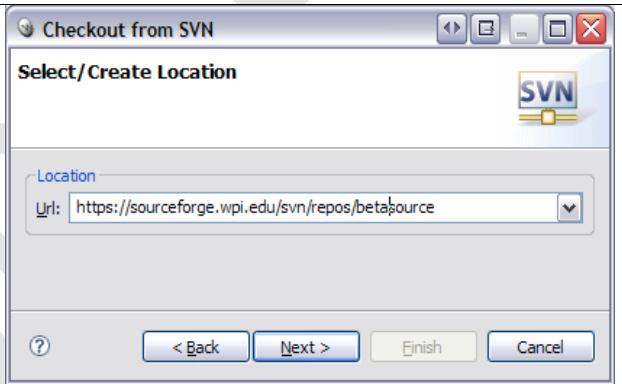
To get the source code requires setting up a “Repository location” then importing the code. The following steps show the process.

Right-click in the “Project Explorer” window in Workbench. Select “Import...”	
Choose “Checkout Projects from SVN” and click next.	

Select “Create a new repository location” and click Next.

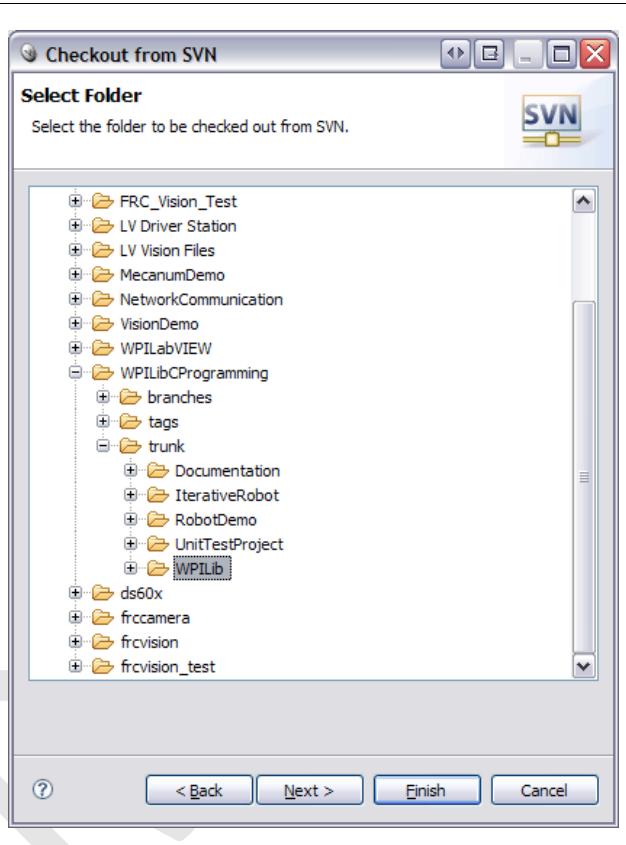


Enter the URL: The URL will be announced as soon as the Subversion server is up and running. This is only an example and cannot be used except by WPI students and faculty.



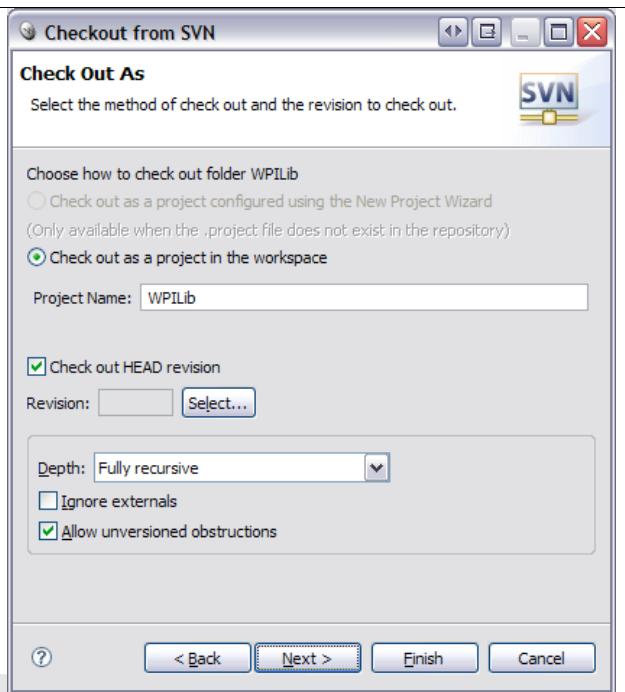
Choose the WPILib folder from the “Select Folder” window. The window on your screen will have a different list of files, but still choose WPILib.

This is an example and the actual details will be announced when the Subversion server is running.



Check out the code as a project in the Workspace by leaving all the default options and clicking “Finish”.

If you are asked for a username and password, it is your username for SourceForge. Checking the “Remember password” box will make this easier since it will ask multiple times.



#### [Using the WPI Robotics Library source code in your projects](#)

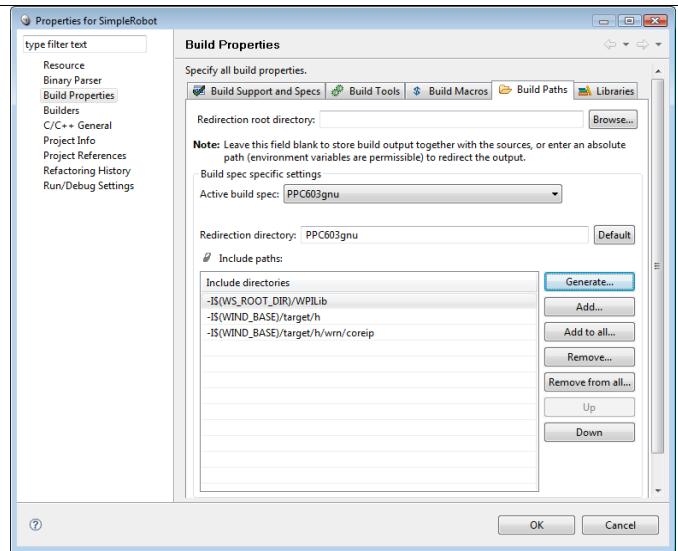
The sample projects provided by FIRST use a library file (WPILib.a) and header files from the Workbench install. If you intend to modify or debug the source copy of the library you just imported, the project settings have to change to refer to that copy of the library instead.

Note: Before doing these steps you must have built the WPILib project once so that the WPILib.a target file has been generated.

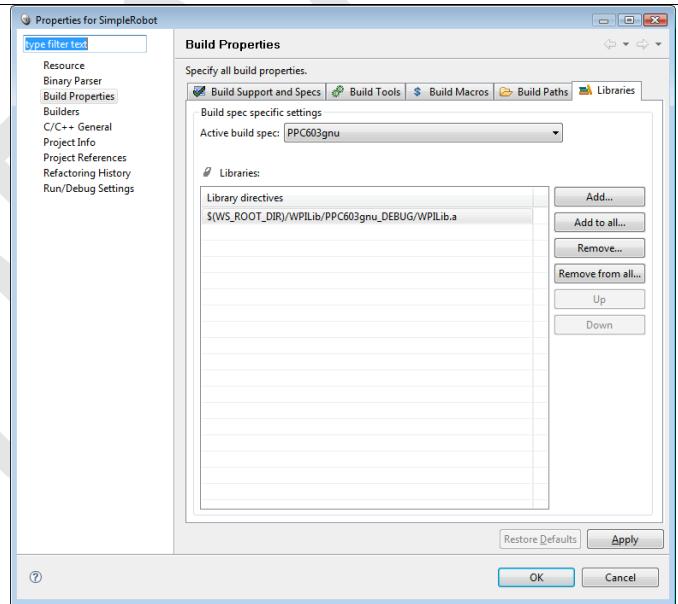
Right-click on the project name in the “Project Explorer” pane in Workbench and select “Properties”.

In the project properties window, select “Build Properties”. Here you can see all the options that Workbench will use to build your project.	
--	--

Select the “Build Paths” tab to use the downloaded WPILib include files to your project rather than the installed version.



Select the Libraries tab and select the WPILib.a library file from the downloaded WPILib project instead of the version preinstalled in Workbench.



*Note: to build WPILib you must install SlikSVN from <http://www.sliksvn.com/en/download>. Once downloaded and installed the builds will run without errors. SlikSVN is a command line interface to Subversion that our build system uses for tracking library versions.*

Now if you rebuild your project it will use the imported version of the WPI Robotics Library rather than the preinstalled version.

DRAFT

## Replacing WPI Robotics Library parts

You can replace any component of the WPI Robotics Library with your own version of that component without having to replace the entire library. When your projects are built, the last step is Linking. This two step process creates a single executable .OUT file by:

1. Combining all the modules (object files) from your project together into the .OUT file
2. Finding all the unresolved pieces such as classes referenced from WPILib and adding those pieces to your .OUT file executable.

Only the pieces of WPILib that are unresolved after step 1 are included from the library and that's the key to substituting your own version of classes.

Suppose you want to use your own version of the Encoder class because you had some extra features you wanted to add. To use your version rather than the WPILib version simply:

1. Get the WPILib version of the file (.cpp and .h) files from the WPILib source code and add them to your project.
2. Make whatever modifications you would like to.
3. Rebuild your project. The library version of the Encoder objects will be included with your set of object modules, so the linker won't take the ones in WPILib.

# Interrupts

## Example

Below is a sample program that generates a square wave on a digital output port that is connected to a digital input port. An interrupt handler is set up on the input port to count the number of cycles.

```

static int interruptCounter = 0;

// The interrupt handler that counts number of square wave cycles
static void tiHandler(tNIRIO_u32 interruptAssertedMask, void *param)
{
    interruptCounter++;
}

void InterruptTestHandler(void)
{
    // create the two digital ports (Output and Input)
    DigitalOutput digOut(CROSS_CONNECT_A_PORT1);
    DigitalInput digIn(CROSS_CONNECT_A_PORT2);

    // create the counter that will also count square waves
    Counter counter(&digIn);

    // initialize the digital output to 0
    digOut.Set(0);

    // start the counter counting at 0
    counter.Reset();
    counter.Start();

    // register and enable the interrupt handler
    digIn.RequestInterrupts(tiHandler);
    digIn.EnableInterrupts();

    // count 5 times
    while (counter.Get() < 5)
    {
        Wait(1.0);
        digOut.Set(1);
        Wait(1.0);
        digOut.Set(0);
    }

    // verify correct operation
    if (interruptCounter == 5 && counter.Get() == 5)
        printf("Test passed!\n");

    // free resources
    digIn.DisableInterrupts();
    digIn.CancelInterrupts();
}
END_TEST(InterruptTestHandler)

```

# Creating your own speed controllers

DRAFT

# 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.

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

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

while(IsOperator())
{
    turretControl.SetSetpoint((turretStick.GetX() + 1.0) * 2.5);
    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.

DRAFT

## Using the serial port

DRAFT

# Relays

The cRIO provides the connections necessary to wire I<sup>2</sup>C 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 state of the relay – Valid inputs: <i>All Directions</i> : kOff – turns off the Relay <i>kForwardOnly or kReverseOnly</i> : kOn – turns on forward or reverse of relay, depending on direction <i>kForwardOnly</i> : kForward – set the relay to forward <i>kReverseOnly</i> : kReverse – set the relay to reverse
Void SetDirection(Direction direction)	Sets the direction of the relay – Valid inputs: <i>kBothDirections</i> : Allows the relay to use both the forward and reverse pins on the channel <i>kForwardOnly</i> : Allows relay to use only the forward signal pin <i>kReverseOnly</i> : Allows relay to use only the reverse signal pin

## Example

```
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.

DRAFT

## Customizing analog sampling

DRAFT

## Using I2C

DRAFT

# C++ Tips

DRAFT

## Creating an application in WorkBench

DRAFT

# Contributing to the WPI Robotics Library

DRAFT

## Glossary

Concurrency

cRIO

deadlock

particle

quadrature encoder

race condition

semaphore

task

VxWorks

DRAFT

# Index

- Accelerometer, 42
- Analog Inputs, 54
- Analog Triggers, 56
- C Programs, 11
- Camera, 57
- Compass, 45
- Compressor, 69
- Conventions, 34
- Counter, 48
- Dashboard
  - Images, 60
- Digital I/O, 39
- Digital Inputs, 40
- Digital Output, 41
- Driver Station, 83
- Encoders, 50
- Examples
  - C Program, 12
  - Color Tracking, 73, 74, 75
  - Gyro, 43
  - Image Processing, 72
  - Interrupts, 99
  - Servo, 65, 66
  - Simple, 8, 27
- Solenoid, 71
- Synchronized Object, 79
- Ultrasonic Rangefinder, 46
- Geartooth Sensor, 51
- Gyro, 43
- Image Processing, 72
- Color Tracking, 73
- Using FRC Vision API, 72
- Using NI Vision API, 72
- IterativeRobot**, 31, 33
- Jaguar, 64
- Motor Controllers, 61
  - Jaguar, 64
  - PWM, 62
  - Victor, 63
- Multitasking, 77
- Objects, 9
- Pneumatics, 68
- Quadrature Encoders, 52
- Robot Base Classes, 31
- RobotBase**, 31, 36
- RobotDrive, 66
- Semaphores, 79
- sensors, 38

- Servo, 65  
SimpleRobot, 8, 31, 32  
Solenoid. *See* Pneumatics  
Subversion, 88  
    Accessing Source Code, 92  
    Installing Subclipse, 88  
Ultrasonic rangefinder, 46  
Version Control. *See* Subversion  
Vision. *See* Image Processing  
VxWorks  
    Concurrency, 77  
    Creating Tasks, 78  
    Critical Regions, 79  
watchdog timer, 28, 37  
Workbench, 13  
    Building Projects, 20  
    Creating Remote System, 14  
    Creating Robot Programs, 18  
    Debug output to PC, 25  
    Debugging, 22  
    Deploying Programs, 26  
    Downloading Projects, 21  
    Troubleshooting, 24  
WPILib  
    Replacing Source Files, 98  
    Using Source Files, 95