

# Robot Programming with the WPI Robotics Library

Worcester Polytechnic Institute Robotics Resource Center

Brad Miller  
9/15/2008

# Contents

Getting Started.....	5
What is the WPI Robotics Library .....	6
A simple robot program.....	8
Using objects.....	9
Creating object instances.....	10
Creating a Robot Program .....	11
Pointers and addresses .....	14
RobotBase class .....	15
SimpleRobot class .....	16
IterativeRobot class .....	17
Watchdog timer class.....	18
Sensors.....	20
Types of supported sensors .....	20
Accelerometer.....	21
Gyro.....	22
Using the Gyro class.....	22
Setting the gyro sensitivity.....	22
HiTechnicCompass .....	23
Ultrasonic rangefinder .....	24
Encoders.....	<b>Error! Bookmark not defined.</b>
Background information .....	<b>Error! Bookmark not defined.</b>
Encoders.....	25
Geartooth Sensor.....	26
Quadrature Encoders.....	27
Background Information.....	27
Counter .....	29
Analog Inputs .....	30
Digital Inputs.....	31
Digital Outputs .....	32
Controlling Motors.....	33
PWM .....	34

Victor.....	35
Jaguar .....	36
Servo .....	37
RobotDrive .....	38
Programmed Operation .....	<b>Error! Bookmark not defined.</b>
Teleoperation.....	<b>Error! Bookmark not defined.</b>
Using Pneumatics.....	<b>Error! Bookmark not defined.</b>
Compressor .....	40
Example.....	40
Solenoid .....	41
Concurrency.....	42
Synchronized and Critical Regions .....	43
System Architecture.....	45
Digital I/O Subsystem.....	46
Analog to Digital Converter Subsystem .....	47
Oversample and Average Engine .....	47
Digital Sources.....	48
Digital Filter .....	48
Analog Triggers .....	49
Average Rejection Filter .....	49
Counters Subsystem .....	50
Getting Feedback from the Drivers Station .....	51
Joysticks .....	52
Driver station analog inputs.....	53
Driver station digital inputs.....	54
Driver station digital outputs .....	55
Advanced Programming Topics .....	56
Using Subversion with Workbench.....	57
Installing the Subclipse client into Workbench.....	57
Getting the WPILib Source Code.....	61
Importing the WPI Robotics Library into your workspace.....	61
Using the WPI Robotics Library source code in your projects .....	63

Replacing WPI Robotics Library parts .....	66
Interrupts .....	67
Creating your own speed controllers.....	68
PID Programming .....	69
Using the serial port.....	70
Creating your own robot type.....	<b>Error! Bookmark not defined.</b>
Relays .....	71
Analog Triggers .....	72
Customizing analog sampling.....	73
Using I2C .....	74
Using DMA for data analysis .....	75
Using WindRiver WorkBench .....	76
Setting up the environment.....	77
Creating a robot project.....	81
Building your project.....	83
Downloading the project to the cRIO .....	84
Debugging your robot program .....	85
What can you do with the debugger .....	85
Creating a Remote System in Workbench .....	<b>Error! Bookmark not defined.</b>
Creating a Debug Configuration for your project .....	<b>Error! Bookmark not defined.</b>
C++ Tips .....	86
Creating an application in WorkBench .....	87
Using C with the WPI Robotics Library.....	88
Contributing to the WPI Robotics Library .....	89
Glossary.....	90

# 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. We have about 500x more memory and probably 100x more processor speed 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 are 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
{
    RobotDemo(void)
    {
        // put initialization code here
    }

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

    void OperatorControl(void)
    {
        // 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 are 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, 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 represents the data unique to that object. When you create an object that memory is allocated and when the object is deleted, 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() ;
```

The variable `heading` will be set 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:

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

How do you decide what to use?

# Creating a Robot Program

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

<b>Autonomous period</b>	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.
<b>Operator Control period</b>	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
{
    RobotDemo(void)
    {
        // put initialization code here
    }

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

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

START_ROBOT_CLASS(RobotDemo);
```

Now add objects to represent the motors and joystick.

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

```
RobotDrive drive(1, 2);
Joystick stick(1);
```

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

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

```
void Autonomous(void)
{
    for (int i = 0; i < 4; i++)
    {
        drivetrain.Drive(0.5, 0.0); // drive 50% forward with 0% turn
        Wait(2000); // wait 2000 ms (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(void)
{
    while (1) // loop forever
    {
        drivetrain.Tank(&stick1, &stick2); // tank drive with the joystick
    }
}
```

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

```
#include "WPILib.h"

RobotDrive drivetrain(1, 2);
Joystick stick(1);

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

    void Autonomous(void)
    {
        for (int i = 0; i < 4; i++)
        {
            drivetrain.Drive(0.5, 0.0); // drive 50% forward, 0% turn
            Wait(2000); // wait 2000 ms (2 seconds)
            drivetrain.Drive(0.0, 0.75); // drive 0% forward and 75% turn
            Wait(750); // turn for almost a second
        }
        drivetrain.Drive(0.0, 0.0); // stop the robot
    }

    void OperatorControl(void)
    {
        while (1) // loop forever
        {
            drivetrain.Tank(&stick1, &stick2); // tank drive with the joystick
        }
    }
};

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` and `stick` are global variables. In computer science classes you would be discouraged from doing that. In the next section pointers will be introduced that make the code more “correct” and maintainable.
- The tank steering method `drivetrain.tank(&stick)` had this `&stick` construct in it. The ampersand (&) represents the address of a Joystick object. The next section will describe how pointers work and what that ampersand really means.
- 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

DRAFT

## 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.**

Class name	Description
<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.</li> </ul>
<b>IterativeRobot</b>	<p>This template gives additional flexibility in the code for responding to various field state changes 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, if 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 simple complex tasks. 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.**

Method	What it does
<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 code is allowed to run to completion and will not be stopped at the end of the autonomous period. If the code 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 continuously while the robot is in the autonomous mode of operation. When the robot changes state to operator control, then the TeleopInit() first, then the TeleopContinuous() method is 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.**

Method name	Description
<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 the 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.

The three Init methods are called only once when that state is entered for the first time. 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 time based algorithms like PID control. Any of the provided methods will be called at the appropriate time so if there is a TeleopPeriodic and TeleopContinuous, they will both be called.

## 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(void);
    virtual void Autonomous(void);
    virtual void OperatorControl(void);
    virtual void RobotMain(void);
    void StartCompetition(void);

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(void)
{
    while (IsDisabled()) Wait(10); // wait for match to start
    if (IsAutonomous()) // if starts in autonomous
    {
        Autonomous(); // run user supplied Autonomous code
    }
    while (IsAutonomous()) Wait(10); // wait until end of autonomous period
    while (IsDisabled()) Wait(10); // 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 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 stopping on the robot.

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

Use of the Watchdog timer is recommended for safety, but 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.

```
void Autonomous(void)
{
    GetWatchdog().SetEnabled(false); // disable the watchdog timer
    Drivetrain.Drive(0.75, 0.0);      // drive straight at 75% power
    Wait(2000);                       // wait for 2 seconds
    .
    .
    .
    GetWatchdog().SetEnabled(true);   // reenale the watchdog timer
}
```

You can get the address of the Watchdog timer object from the RobotBase class 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.

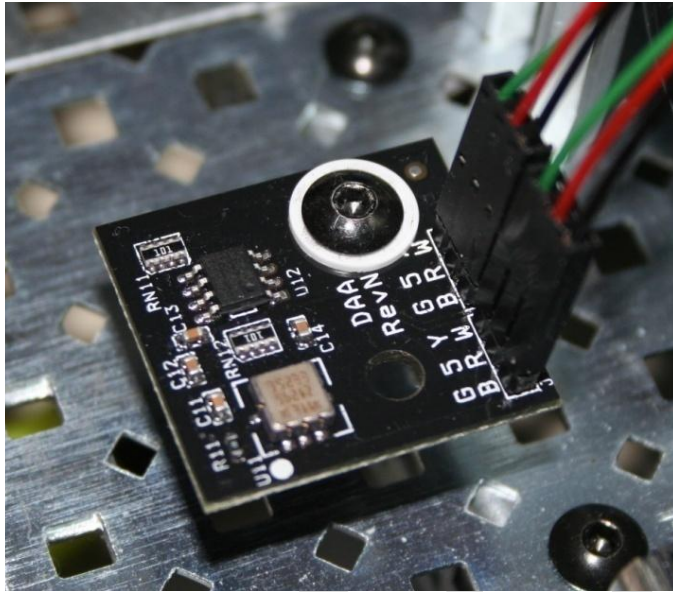
Category	Supported sensors
<b>Wheel/motor position measurement</b>	<a href="#">Geartooth sensors</a> , <a href="#">encoders</a> , <a href="#">analog encoders</a> , and <a href="#">potentiometers</a>
<b>Robot orientation</b>	<a href="#">Compass</a> , gyro, accelerometer, ultrasonic rangefinder
<b>Generic pulse output</b>	Counters

In the past high speed counting of pulses of 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 of any device generating pulses for its output. Another example is a generalized interrupt facility to catch high speed events without polling and potentially missing them. These features are described in the Synchronized and Critical Regions section of this manual.

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



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

# Gyro

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

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

## Using the Gyro class

The Gyro object is typically created in the constructor of the `RobotBase` derived object. When the Gyro object is instantiated it will go through a 1 second calibration period to determine 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°/sec gyro that was delivered by *FIRST* in the kit of parts last year. As soon as we find out if and what type of gyro will be in the kit this year, the default will change to match it.

If a different gyro is selected then the Gyro object has to change to match it. To change gyro types call the `SetSensitivity(float sensitivity)` method and pass as a parameter the sensitivity in volts/°/sec. Just be careful since the units are typically mV (volts / 1000) in the spec sheets. A gyro with a sensitivity of 12.5 mV/°/sec would require a `SetSensitivity()` parameter value of 0.0125.

# HiTechnicCompass

DRAFT

## Ultrasonic rangefinder

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

### SRF04 Connections

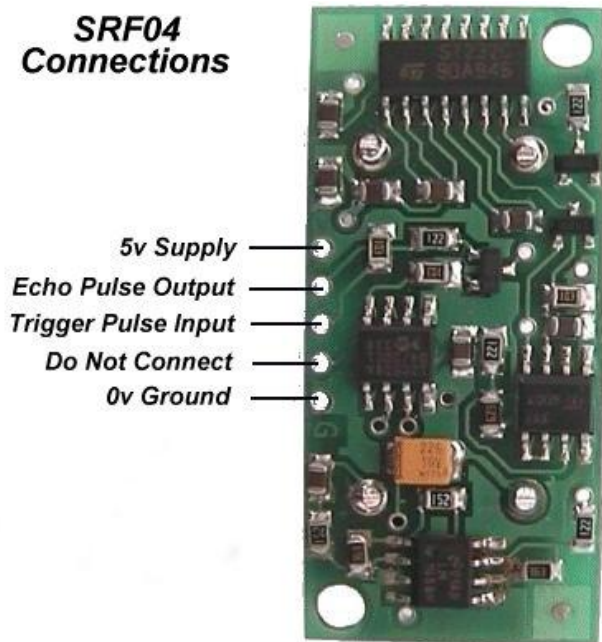


Figure 2: SRF04 Ultrasonic Rangefinder connections

Both the Echo Pulse Output and the Trigger Pulse Input have to be connected to digital I/O ports on a digital sidecar. When creating the Ultrasonic object, specify which ports it is connect to in the constructor:

```
Ultrasonic ultra(ULTRASONIC_PING, ULTRASONIC_ECHO);
```

In this case ULTRASONIC\_PING and ULTRASONIC\_ECHO are two constants that are defined to be the ports numbers.

Using ultrasonic rangefinders that do not have these connections should not be implemented with 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.



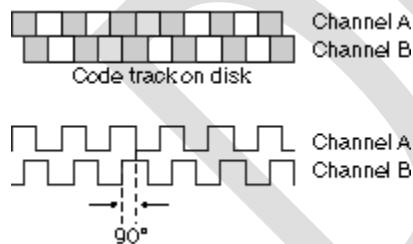
# Encoders

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	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 or index outputs of a quadrature encoder are examples of this type of device.
Quadrature encoders Encoder class	Quadrature encoders have two outputs typically referred to as the A channel and the B channel. The B channel is out of phase from the A channel. By measuring the relationship between the two inputs the software can determine the direction of rotation. The system looks for Rising Edge signals (ones where the input is transitioning from 0 to 1) on the A channel. 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.
Gear tooth sensor GearTooth class	This is a device supplied by FIRST as part of the FRC robot standard kit of parts. The gear tooth sensor is designed to monitor the rotation of a sprocket or gear that is part of a drive system. It uses a Hall-effect device to sense the teeth of the sprocket as they move past the sensor.

**Table 4: Encoder types that are supported by WPILib**

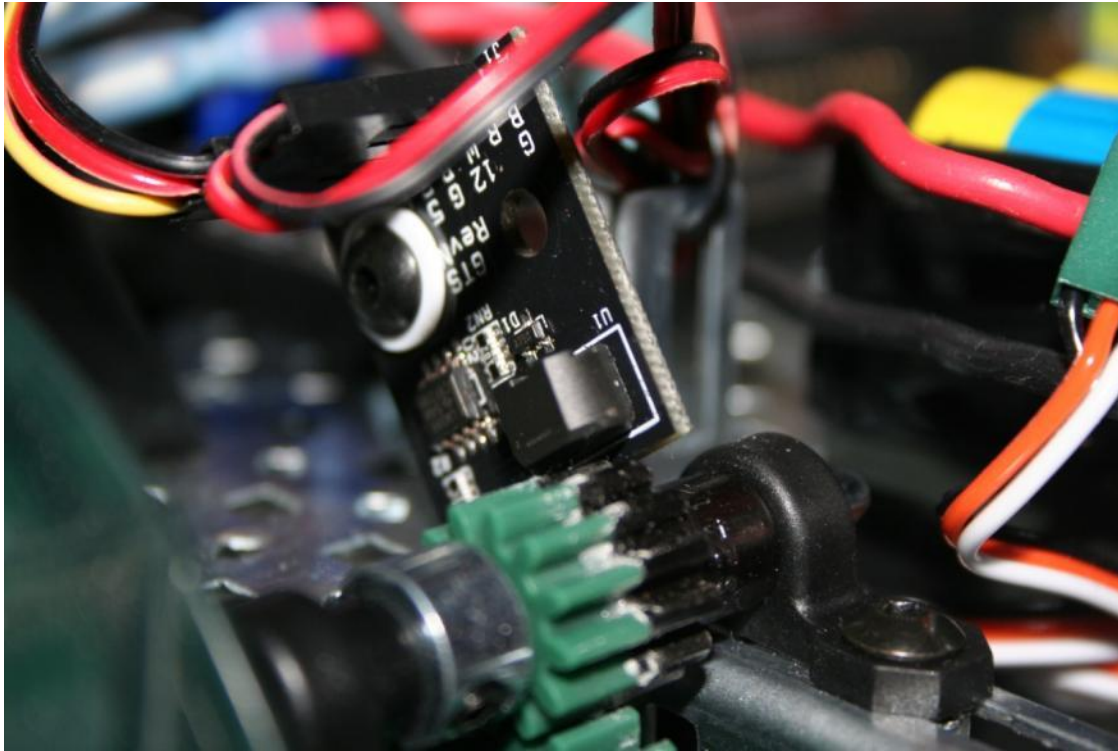
These types of encoders are described in the following sections.



**Figure 3: Quadrature encoder phase relationships between the two channels.**

## 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.



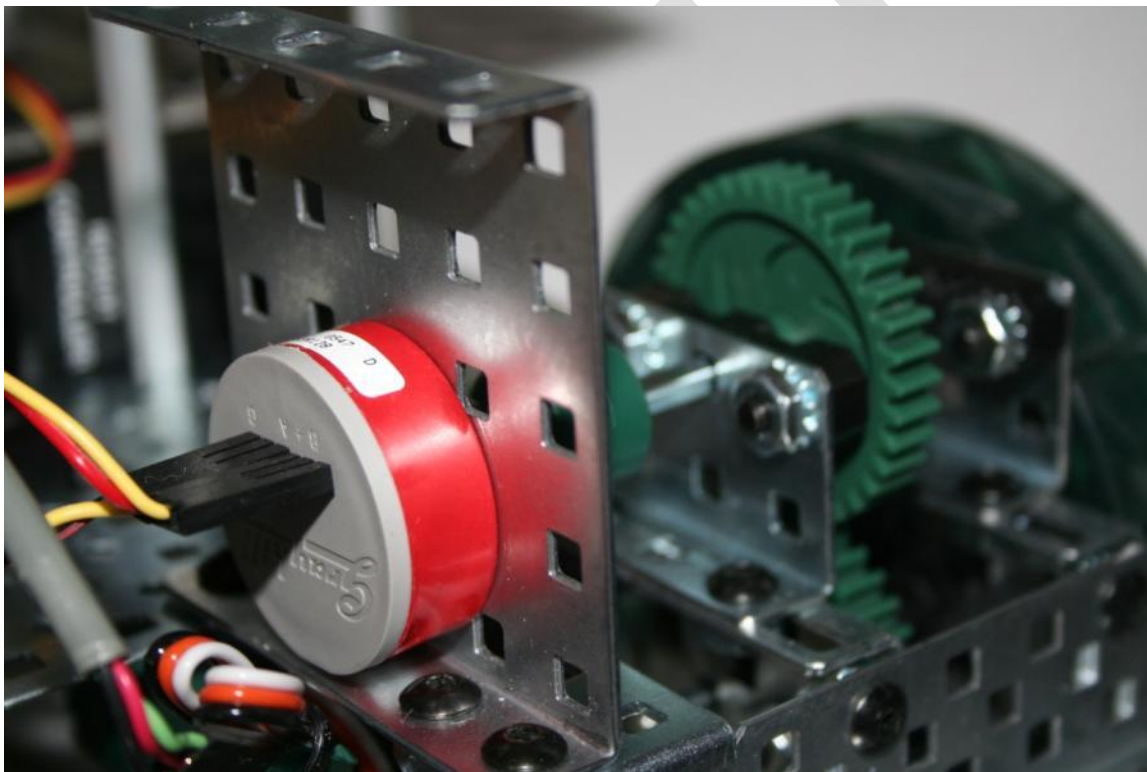
**Figure 4:** A gear tooth sensor mounted on a VEX robot chassis measuring a metal gear rotation. Notice that there is a metal gear attached to the plastic gear in this picture. The gear tooth sensor needs a ferrous material passing by it to detect rotation.

# 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 determines the rotational speed of the shaft and the number of slots that have passed the detector determine the number of rotations (or distance).



**Figure 5:** 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.

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

Where 1 and 2 are the port numbers for the two digital inputs and the true value 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.

DRAFT

# Counter

DRAFT

## Analog Inputs

The Analog to Digital converter system 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 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. The summed value is divided 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 larger than the average output.

# Digital Inputs

DRAFT

# Digital Outputs

DRAFT



## 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 motors (Jaguars or Victors) and servos.

Motor speed controller values floating point and 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.

DRAFT

## PWM

The PWM class is the base class for devices that operate on PWM signals and is the connection to the PWM 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 to 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 rate and only take a PWM control signal.

DRAFT

## Jaguar

The Jaguar class supports the Luminary Micro Jaguar speed controller. It has an update rate of slightly over 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.

DRAFT

## Servo

The Servo class supports the hitechnic servos supplied by FIRST. They have a 20ms update rate 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.

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(1000); // wait 1 second
}
```

# RobotDrive

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

To use it, create a RobotDrive object specifying the left and right motors on the robot:

```
RobotDrive drive(1, 2); // left, right motors on ports 1,2
Or
RobotDrive drive(1, 2, 3, 4); // four motor drive case
```

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

```
SetInvertedMotor(kFrontLeftMotor);
```

This method 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 axis.
<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. 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.

DRAFT

# Compressor

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

- The Spike (relay) port that is controlling the power to the compressor
- The Digital input that the pressure switch is connected to 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 you would typically create an instance of the Compressor object and start it 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 Start() and Stop() methods.

The compressor class will create instances of the DigitalInput and Relay objects to read the pressure switch and operate the Spike. There is no need to do this yourself.

## 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 function the 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.



## Solenoid (Pneumatics)

The Solenoid object controls the outputs of the 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 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 pneumatic bumper breakout board.

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

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), 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
}
for (int i = 0; i < 8; i++)
{
    s[i]->Set(false); // turn them each off in turn
    Wait(1000);
}
for (int i = 0; i < 8; i++)
{
    s[i]->Set(true); // turn them back on in turn
    Wait(1000);
    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.

## 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 runs is scheduled to run by VxWorks based on its priority and availability of resources it might be waiting on. For example, if one task does a `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

## 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).

Typically semaphores are used to ensure only single task access to the shared data. 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. 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**.

The Synchronized object is a simple wrapper around semaphores that tries to solve the problem. 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 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.

DRAFT

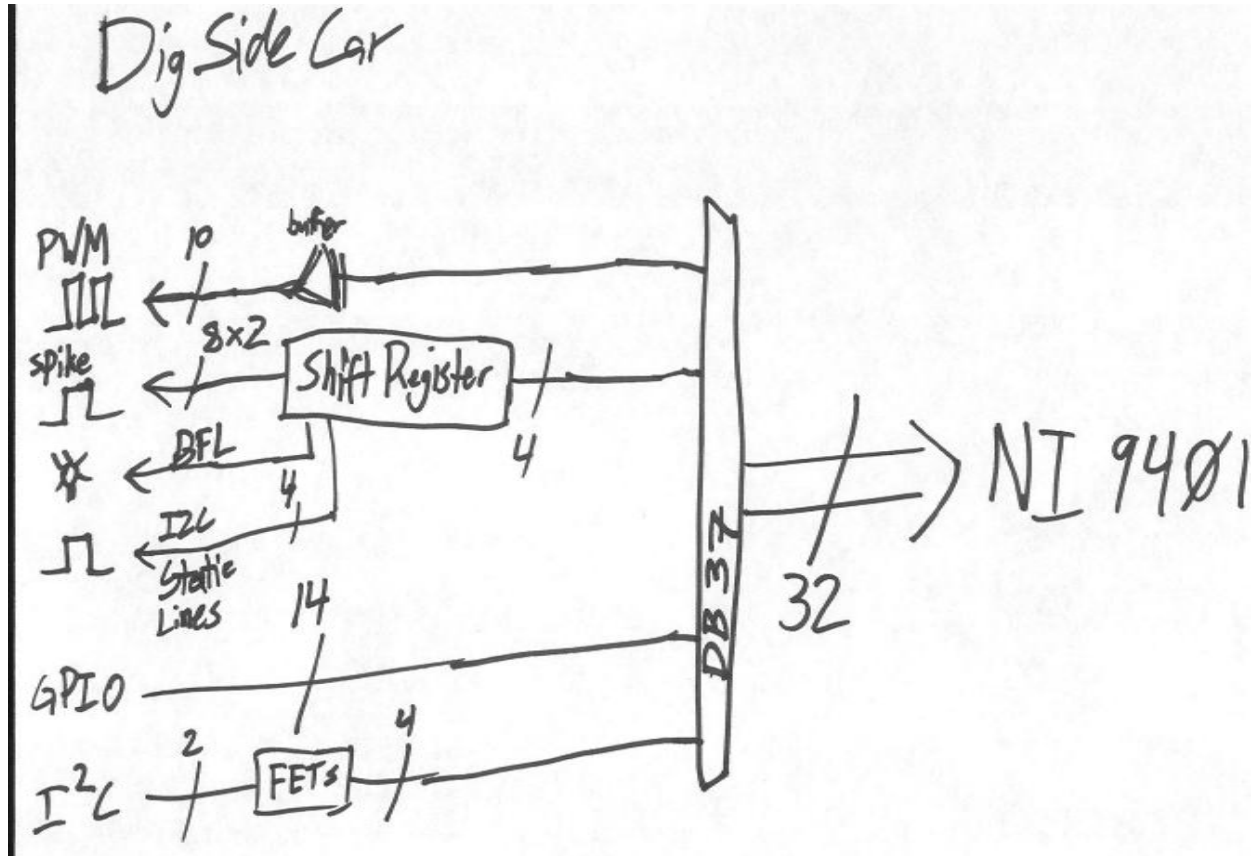
# System Architecture

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.

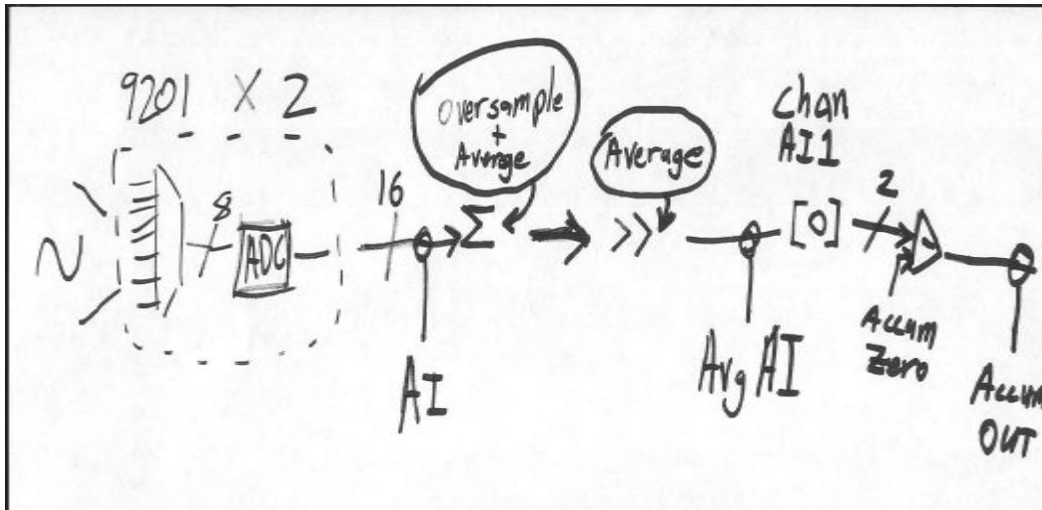
*Note: This is a work in progress, the pictures will be cleaned up and explanations will be soon added. We wanted to make this available to you in its raw form rather than leaving it out all together.*

DRAFT

# Digital I/O Subsystem



# Analog to Digital Converter Subsystem



## Oversample and Average Engine

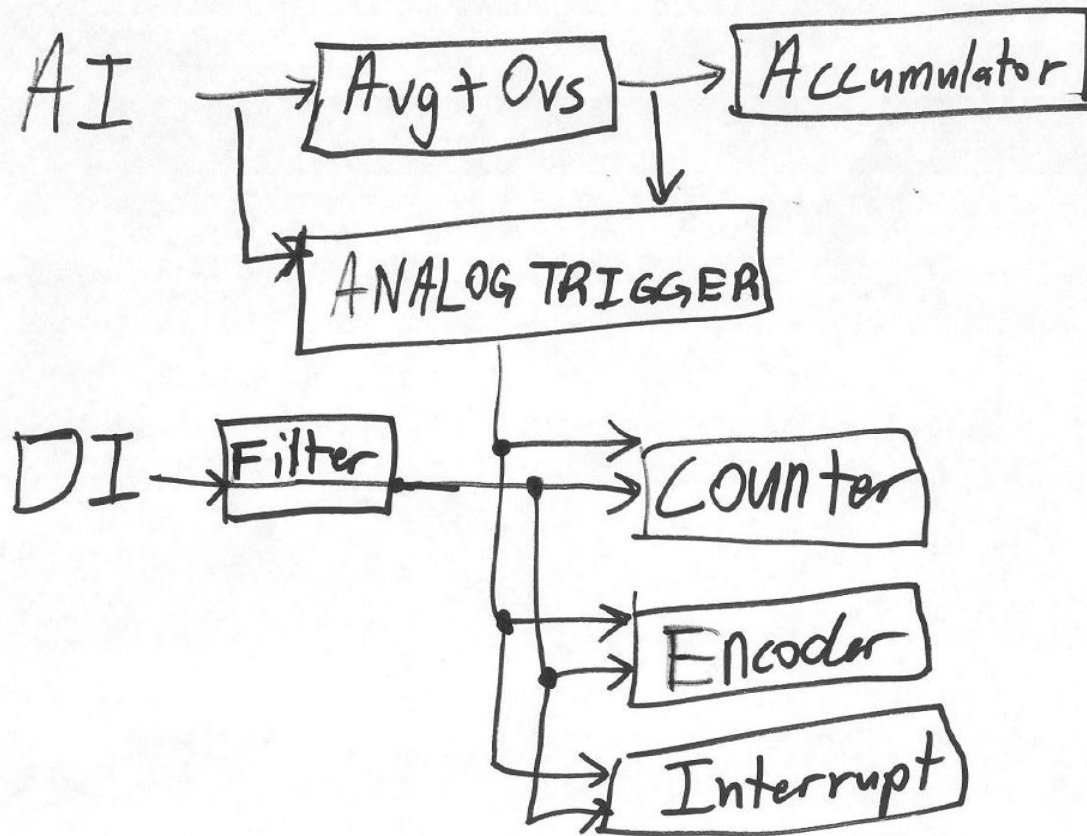
Oversample & Average Engine

$$Avg AI = \frac{\sum_0^{2^M-1} \left( \sum_0^{2^N-1} (AI_x) \right)}{2^M}$$

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

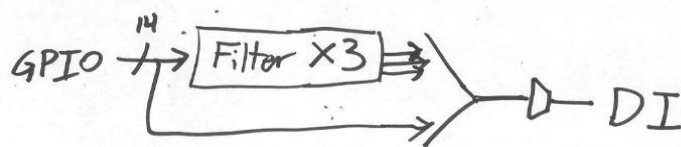
N = oversample bits  
M = average bits

# Digital Sources

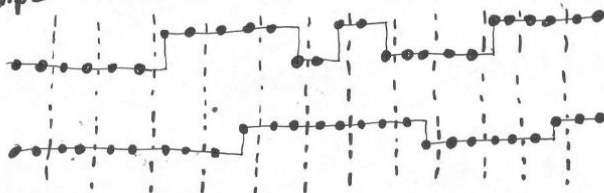


## Digital Filter

Digital Filter (\*per module)

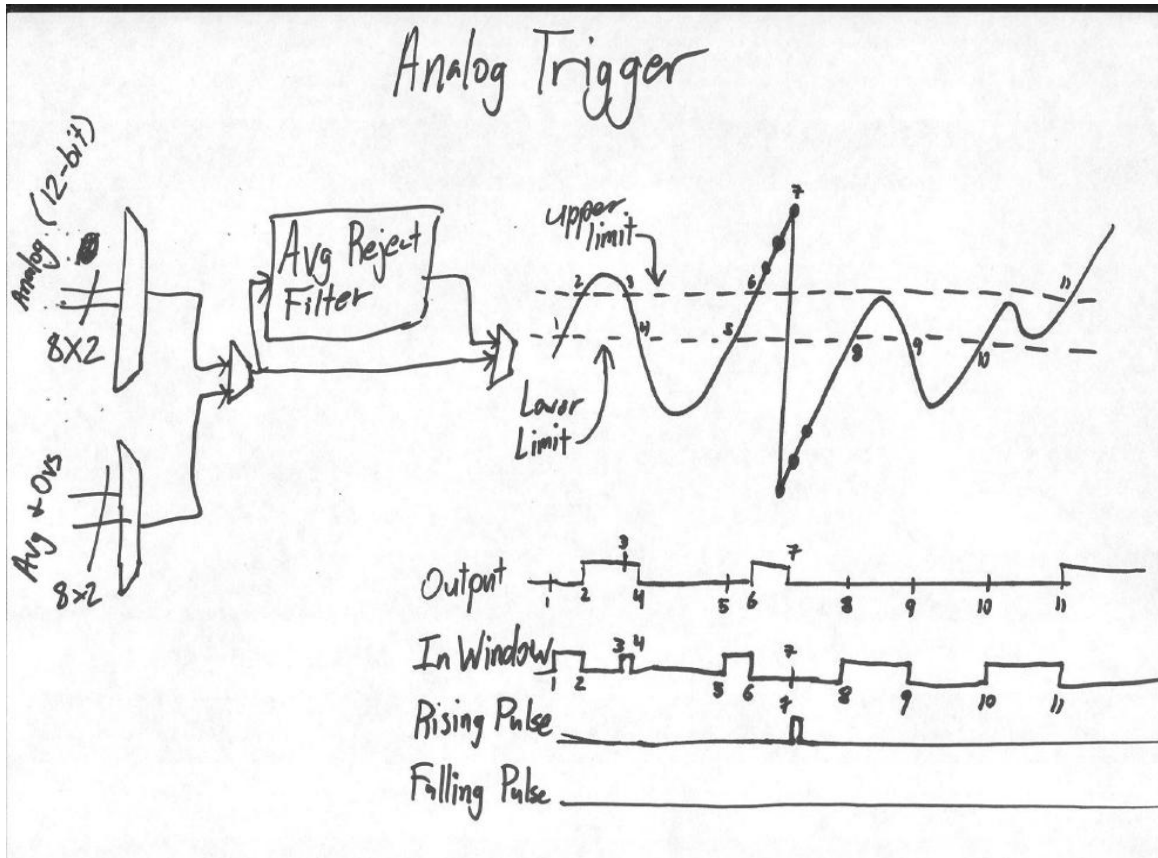


Example: Period = 2

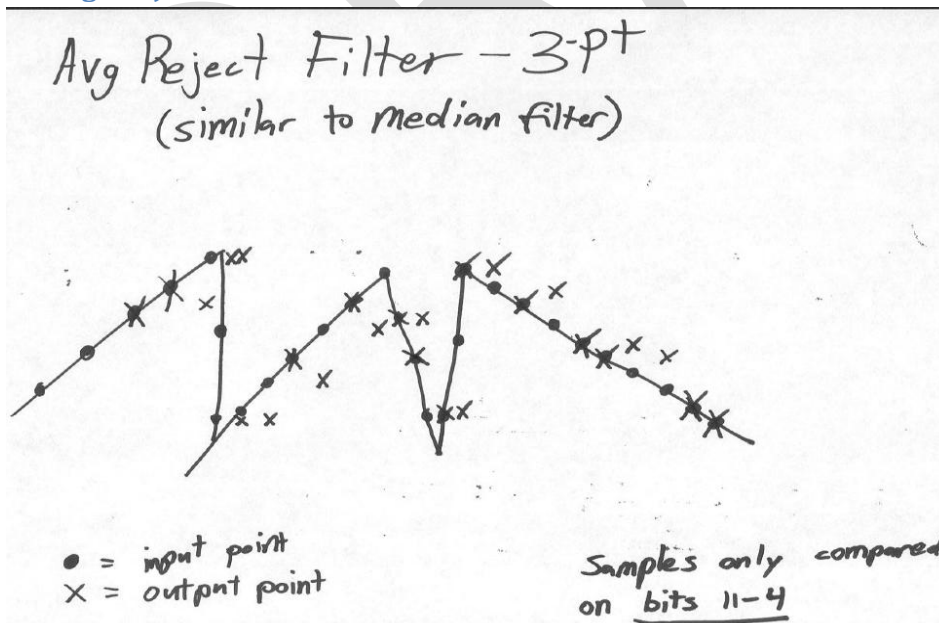




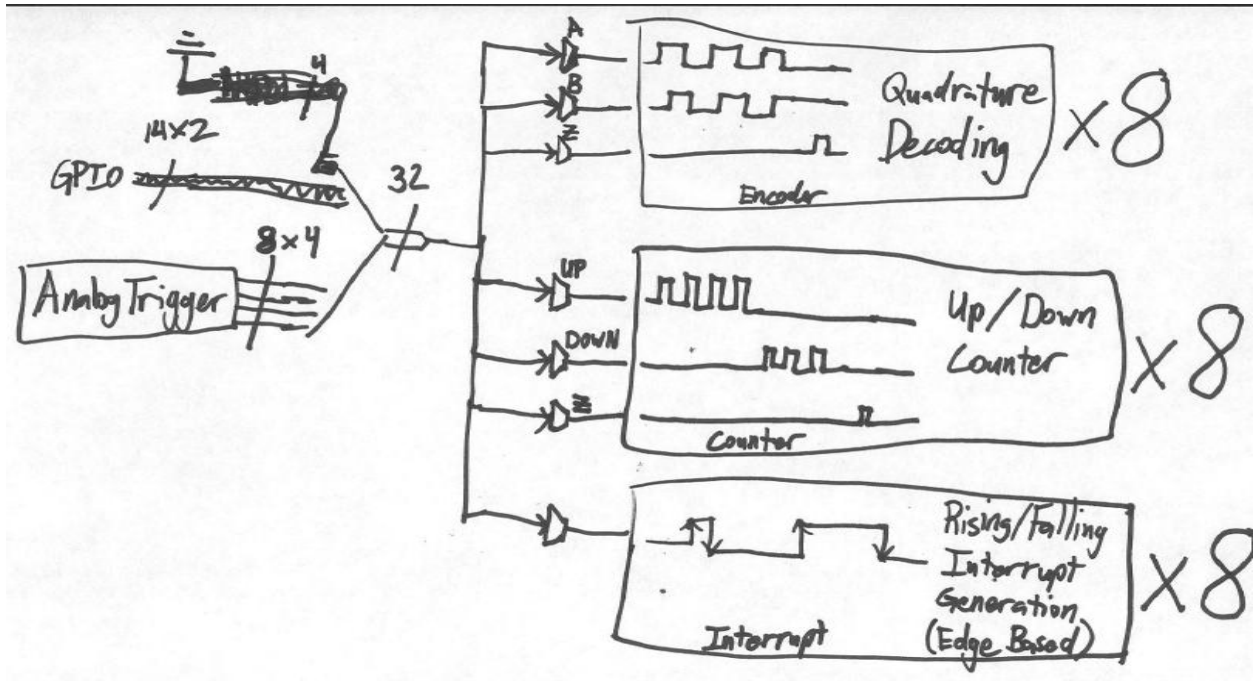
# Analog Triggers



## Average Rejection Filter



# Counters Subsystem



DRAFT

# Getting Feedback from the Drivers Station

DRAFT

# Joysticks

DRAFT

## Driver station analog inputs

DRAFT

## Driver station digital inputs

DRAFT

## Driver station digital outputs

DRAFT

# Advanced Programming Topics

DRAFT



## Using Subversion with Workbench

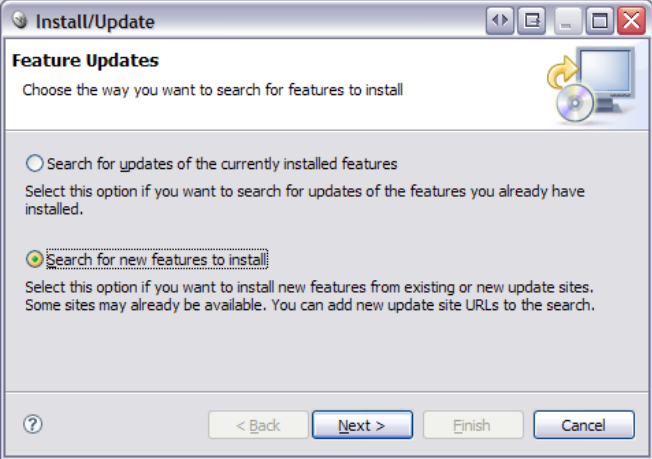
Subversion is a free source code management tool that is designed to track changes to a project as it is developed. You can save each revision of your code in a repository, go back to a previous revision, and compare revisions to see what changed. You should install a Subversion client if:

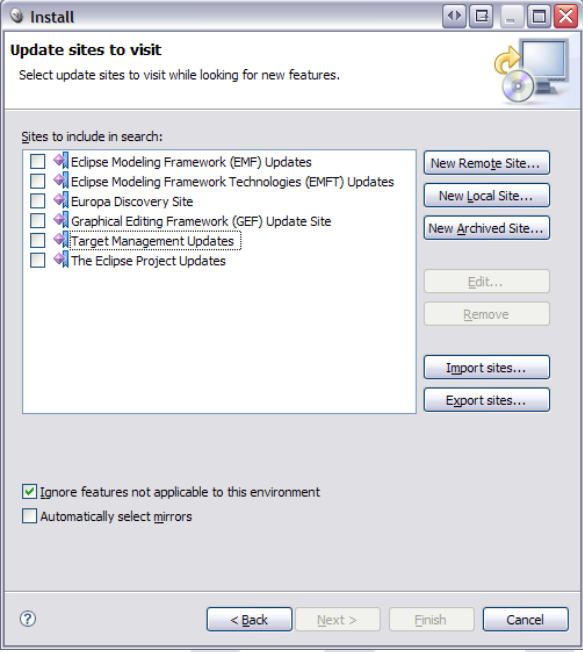
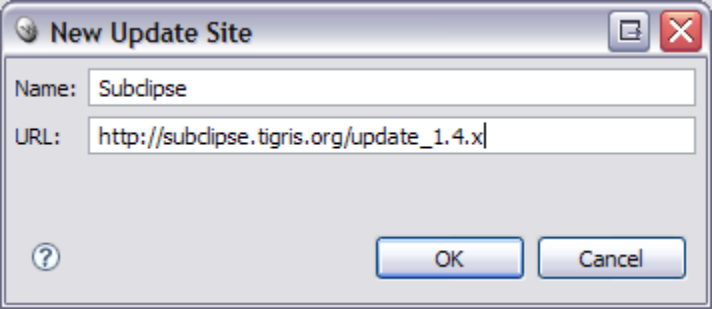
- You need access to the WPI Robotics Library source code installed on a Subversion server
- You have your own Subversion server for working with your team projects

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

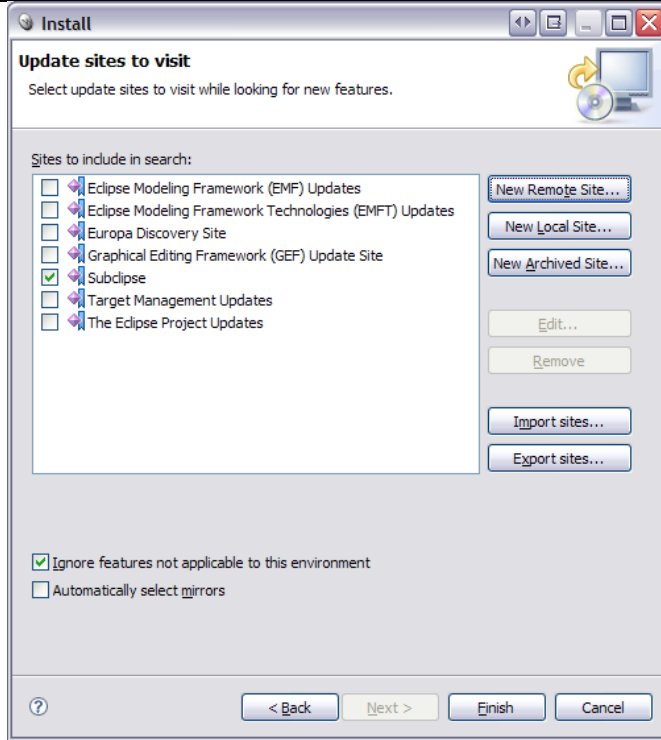
### Installing the Subclipse client into Workbench

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

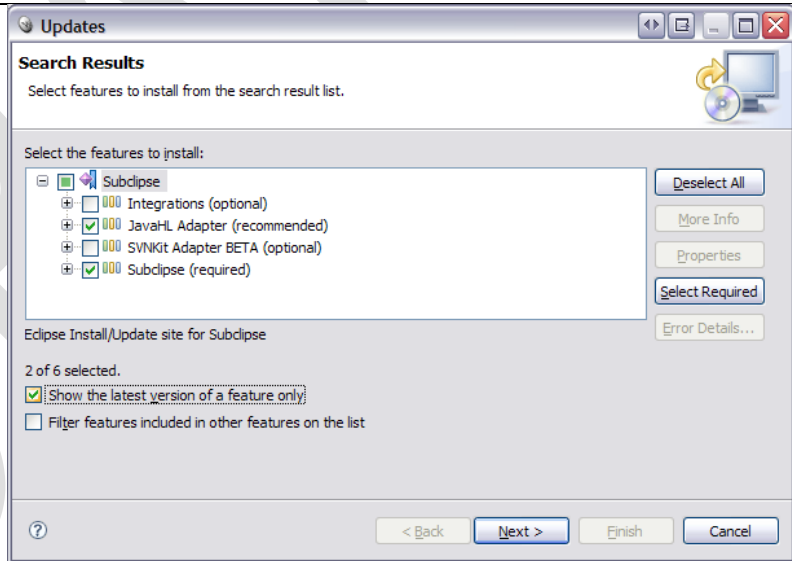
<p>On the help menu, select "Software updates", then "Find and Install".</p>	
<p>Select "Search for new features to install and click Next.</p>	

<p>Click, "New Remote Site..." to add the Subclipse update site.</p>	 <p>The screenshot shows the 'Install' dialog box with the 'Update sites to visit' section. A list of update sites is shown, and the 'New Remote Site...' button is highlighted with a red box. The list includes: Eclipse Modeling Framework (EMF) Updates, Eclipse Modeling Framework Technologies (EMFT) Updates, Europa Discovery Site, Graphical Editing Framework (GEF) Update Site, Target Management Updates, and The Eclipse Project Updates. Other buttons like 'New Local Site...', 'New Archived Site...', 'Edit...', 'Remove', 'Import sites...', and 'Export sites...' are also visible.</p>
<p>Enter the information for the update site and click OK.</p>	 <p>The screenshot shows the 'New Update Site' dialog box. The 'Name' field contains the text 'Subclipse' and the 'URL' field contains the text 'http://subclipse.tigris.org/update_1.4.x'. There are 'OK' and 'Cancel' buttons at the bottom.</p>

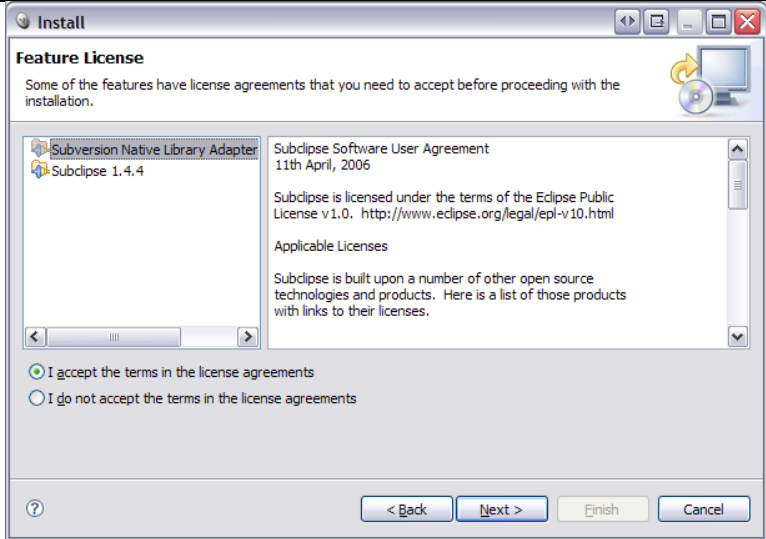
Now you should see Subclipse added to the list of “Sites to include in search:”. Click “Finish”.



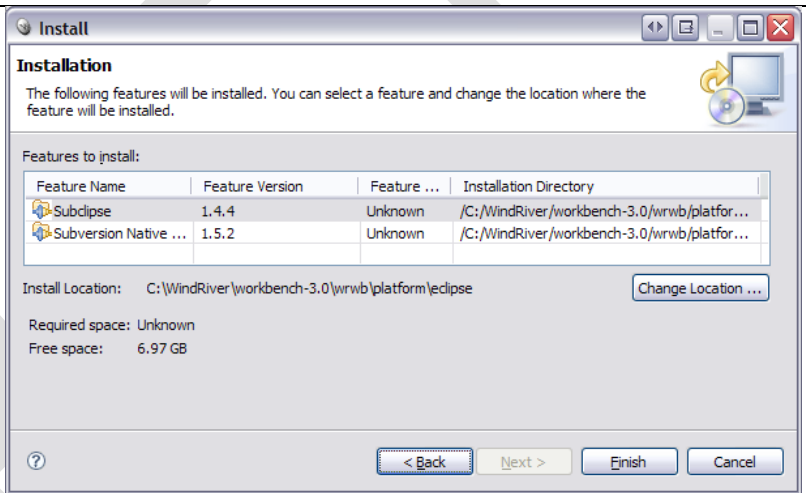
Select the “JavaHL Adapter” and “Subclipse” from the list of features to install.



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.

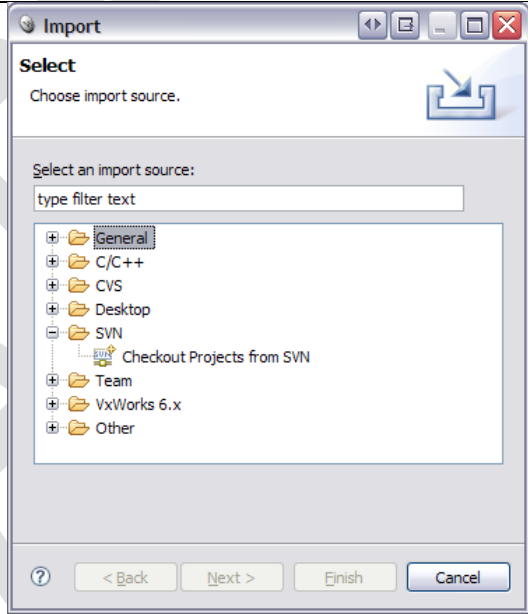
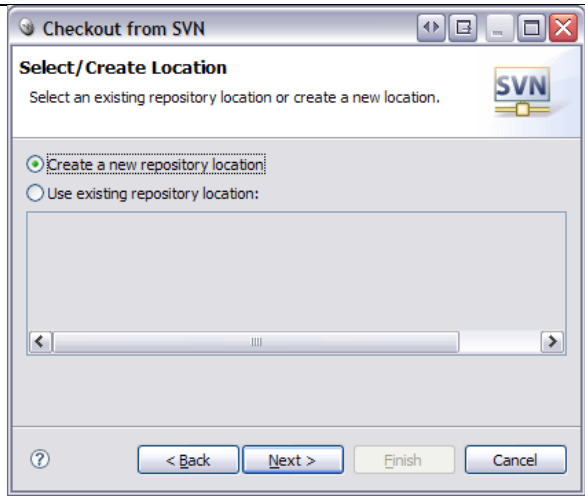


## Getting the WPILib Source Code

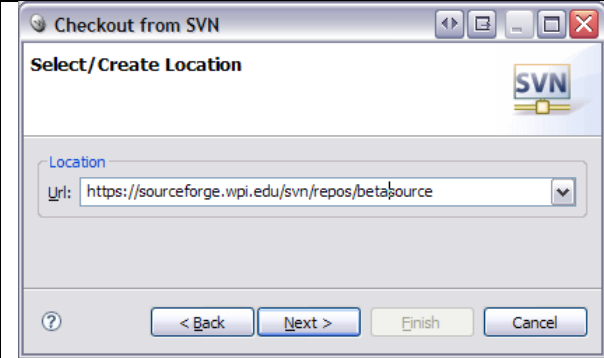
The WPI Robotics Library source code is installed on a Subversion server. To get it requires having a subversion client installed in your copy of Workbench. See Installing the Subclipse client into Workbench for instructions on how to set it up.

### Importing the WPI Robotics Library into your workspace

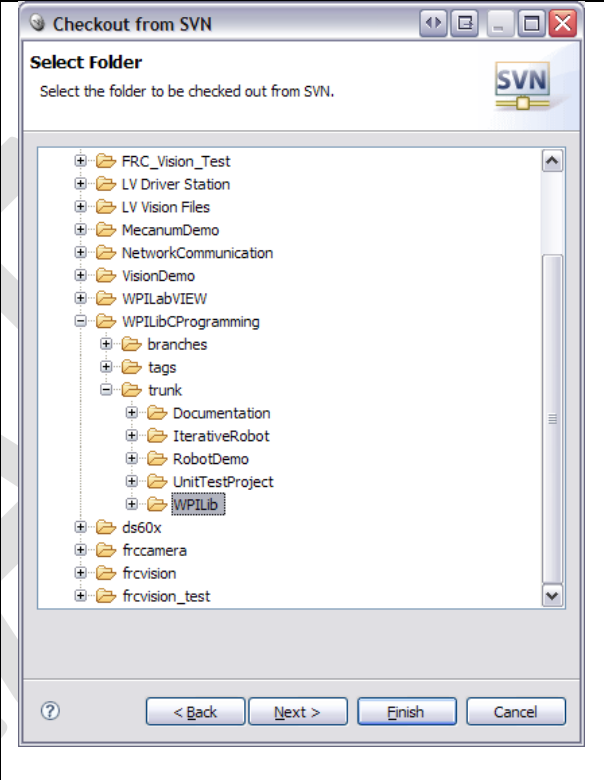
To get the source code requires setting up a “Repository location” then importing the code. The following steps show the process.

<p>Right-click in the “Project Explorer” window in Workbench. Select “Import...”</p>	
<p>Choose “Checkout Projects from SVN” and click next.</p>	
<p>Select “Create a new repository location” and click Next.</p>	

Enter the URL:  
<https://sourceforge.wpi.edu/svn/repos/betasource>  
 and click "Next".

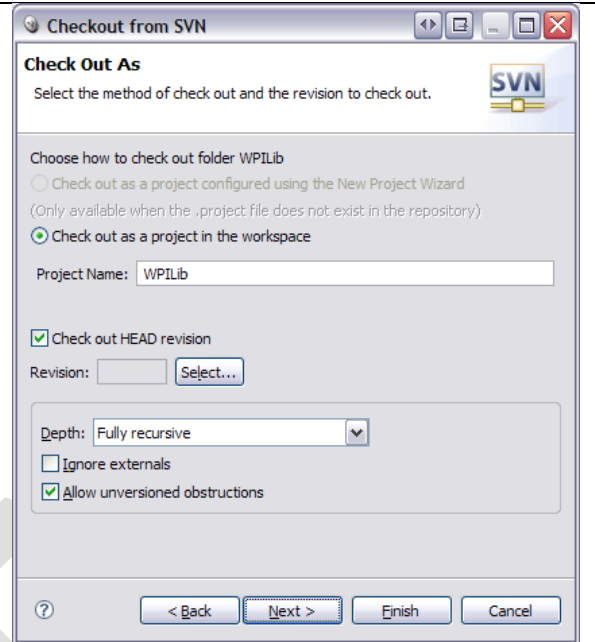


Choose the WPILib folder from the "Select Folder" window. The window on your screen will have a different list of files, but still choose WPILib.



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

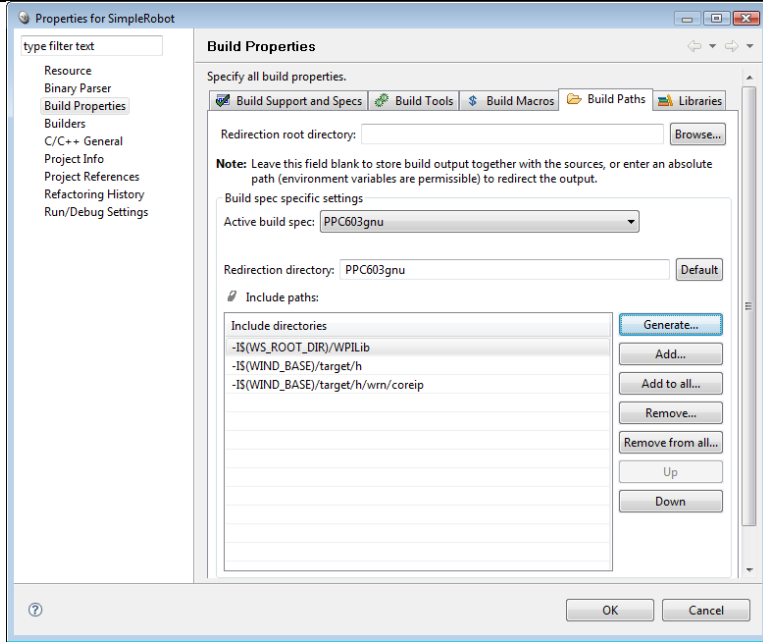
The sample projects provided by FIRST use a library file (WPILib.a) and header files from the Workbench install. If you intend to modify or debug the source copy of the library you just imported, the project settings have to change to refer to that copy of the library instead.

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

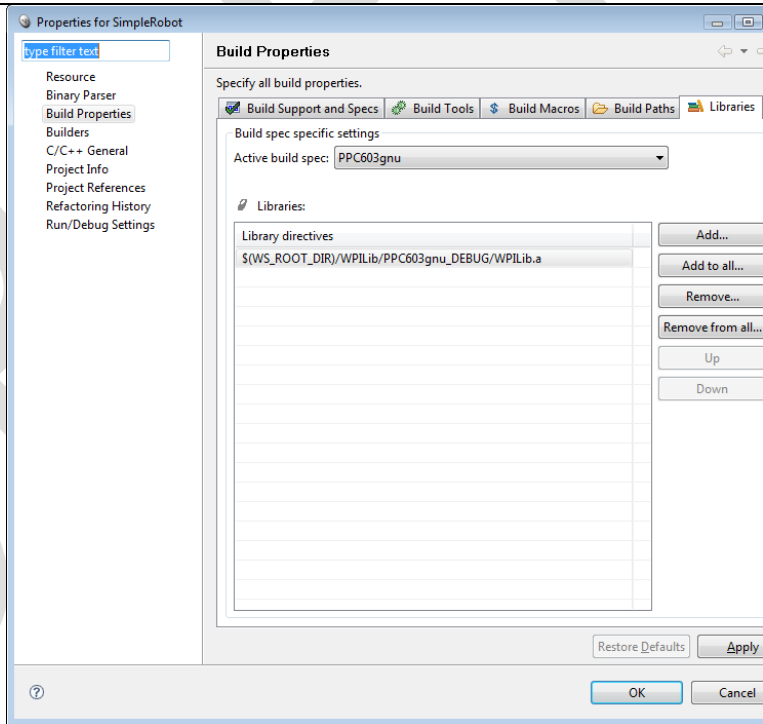
Right-click on the project name in the “Project Explorer” pane in Workbench and select “Properties”.

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

DRAFT

# Creating your own speed controllers

DRAFT

# PID Programming

DRAFT

## Using the serial port

DRAFT

# Relays

DRAFT

# Analog Triggers

DRAFT



# Customizing analog sampling

DRAFT

# Using I2C

DRAFT

## Using DMA for data analysis

DRAFT

## Using WindRiver WorkBench

WindRiver 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.

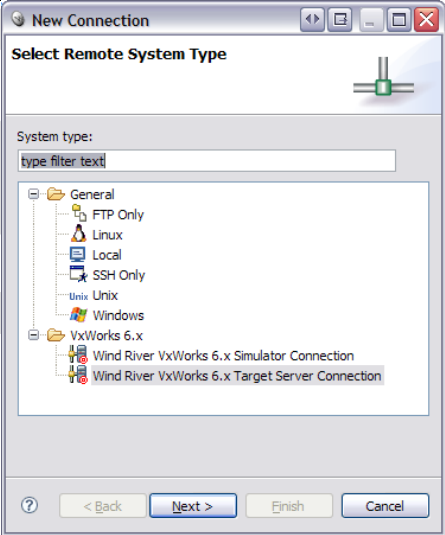
## Setting up the environment

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.

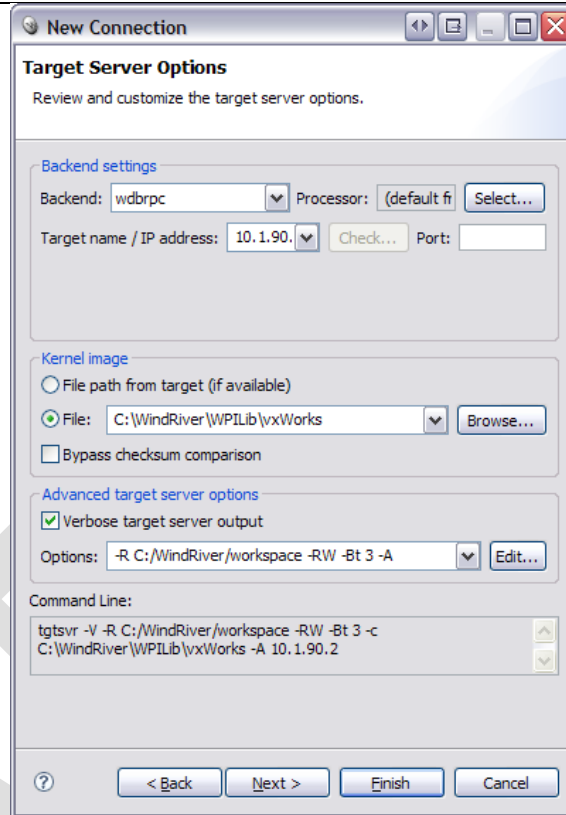
1. The target **remote system**, which is the cRIO that you will use to download and debug your programs.
2. The **run** or **debug configuration** that describes the program to be debugged and which remote system you want to debug it on.
3. The FIRST Downloader settings that tell which program should 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

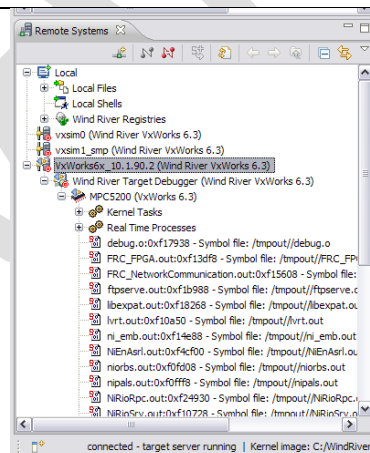
Workbench connects to your cRIO controller and can download and remotely debug programs running on it. In order to make that connection, Workbench needs to add your cRIO to its list of Remote Systems. Each entry in the list tells Workbench the network address of your cRIO and has a kernel file that is required for remote access. To create the entry for your system do the following steps.

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.

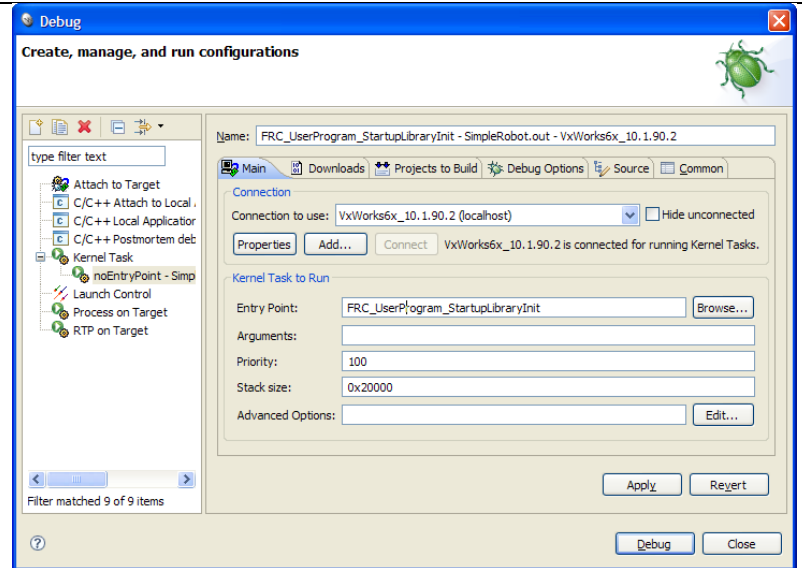


### Creating a Debug Configuration for your project

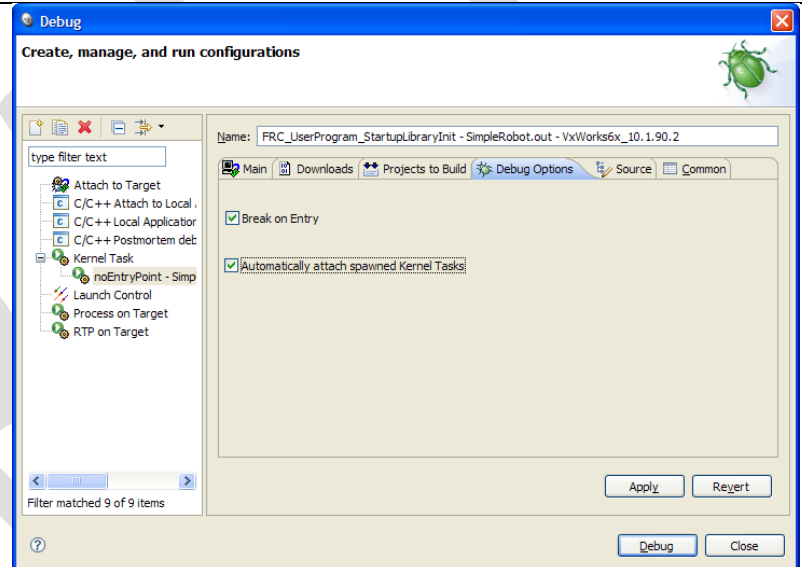
Workbench has specifications called Run and Debug Configurations that describe which program to download to the cRIO and how the debugger should connect to it. One must be set up prior to debugging your new project.

Right-click on the project you wish to debug in the Project Explorer window. Select “Debug Kernel Task...”

Fill in the “Main” tab of the “Create, Manager, and Run Configurations” window. You should always put `FRC_UserProgram_StartupLibraryInit` for the entry point. If you browse in the list of possible entry points it will be under “Downloads”. The other entry points refer to code preloaded into the cRIO. *Note: you will not be able to edit the entry point unless the cRIO Remote System is connected to the PC. See the previous section for details.*

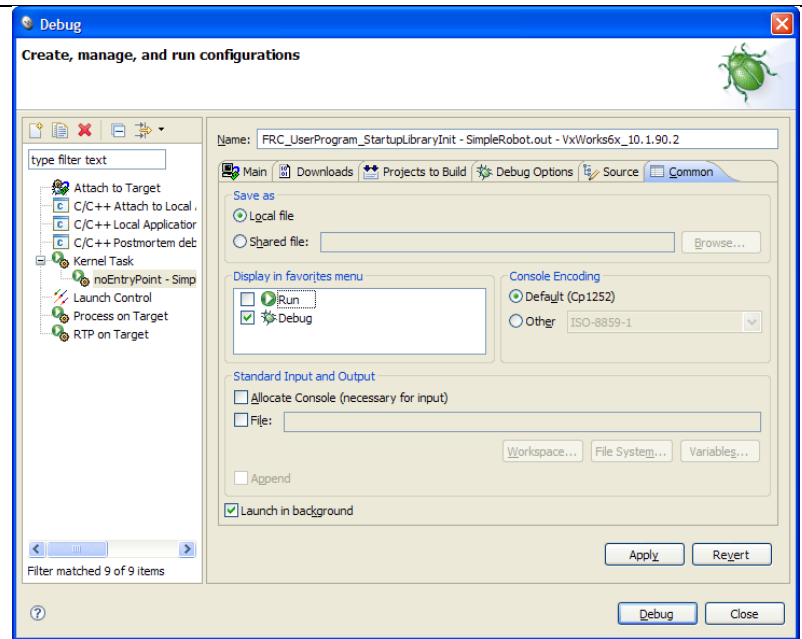


In the “Debug Options” tab, be sure to set “Automatically attach spawned kernel tasks”. This will make sure you can debug any tasks and interrupt handlers that you might start in your program.



In the common tab you can select “Debug” under the “Display in the favorites”. This will make it easier to start your program later using the debug toolbar item in Workbench.

Then apply the settings.



To download the program in the cRIO, click the “Debug” button.



## Creating a robot project

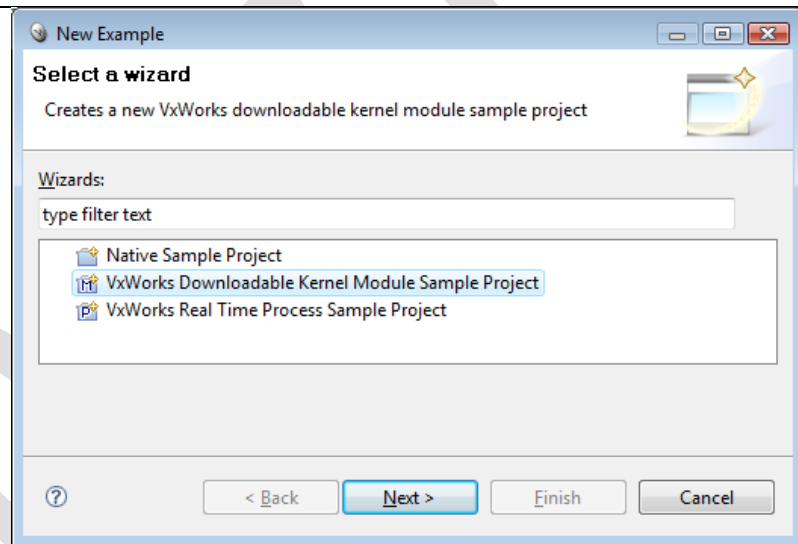
The easiest way to create your own project for your robot is to start with one of the existing templates:

- SimpleRobotTemplate
- IterativeRobotTemplate (coming soon)

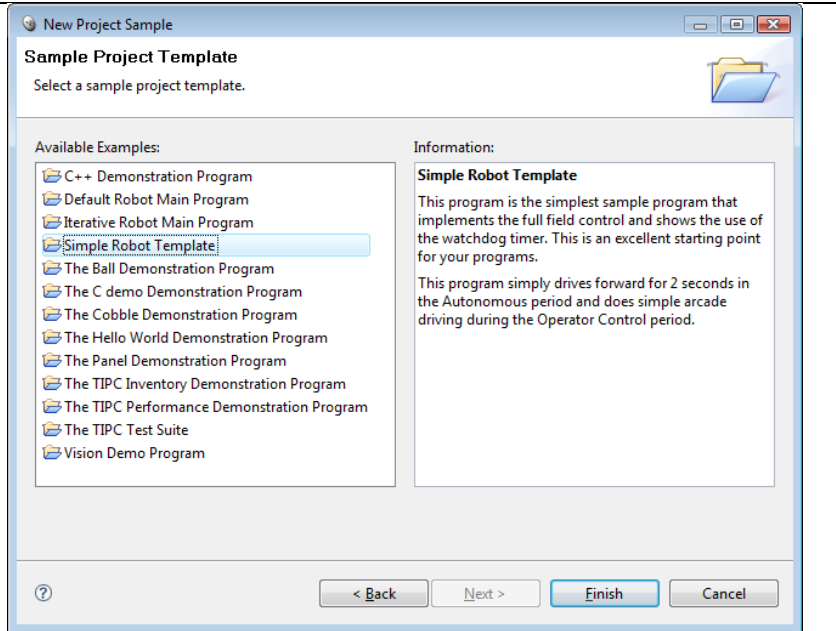
In both cases the templates are based on the RobotBase class and have some of the functions overridden to change the behavior. Additional templates can be implemented that implement other behaviors for example event driven models.

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.

click "File" from the main menu, then "New", then "Example...". From the example project window select "VxWorks Downloadable Kernel Module Sample Project", and then click "Next".



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.

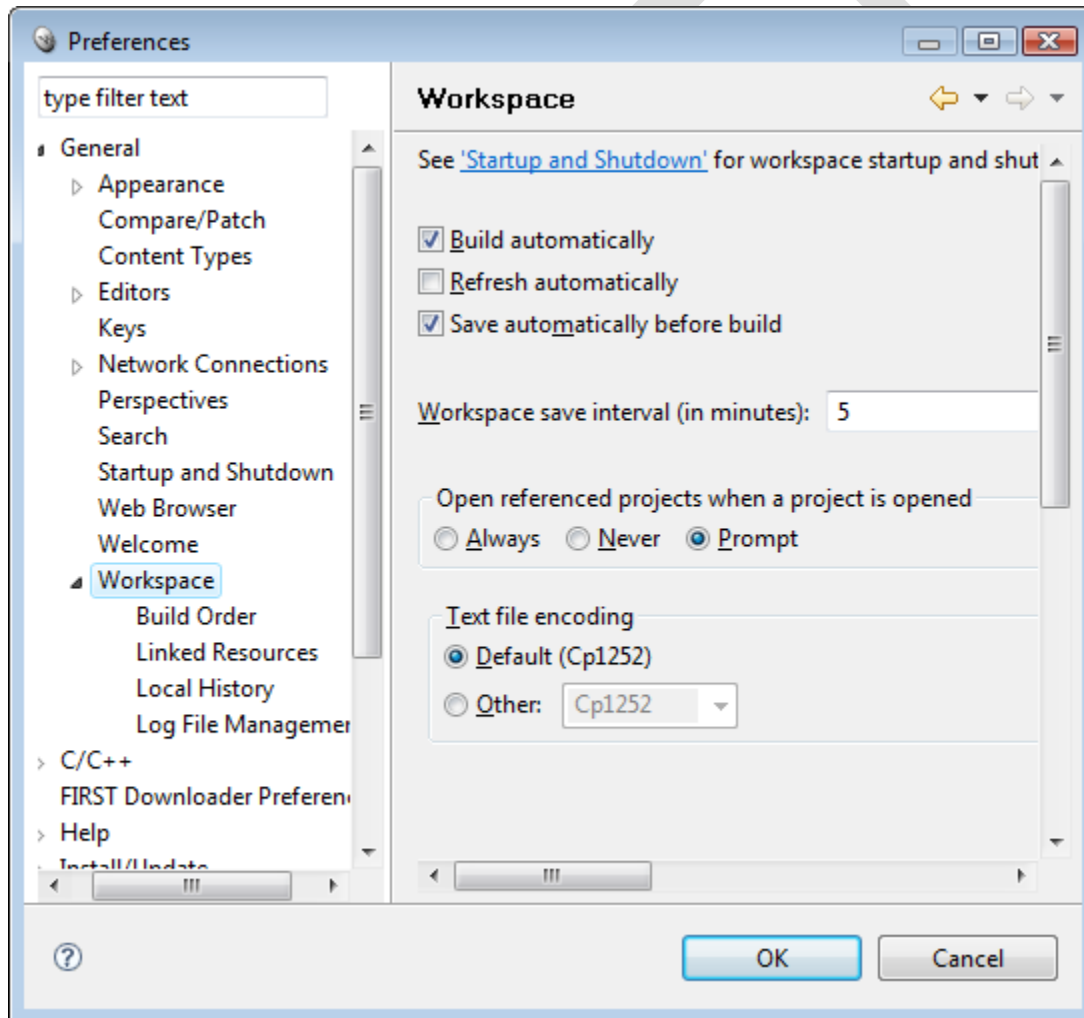


## Building your project

The project is built by right-clicking on the project name in the Project Explorer window and select “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. Files can quickly be saved after editing by using the shortcut, Ctrl-S.



## 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.

To deploy the program you must set up the FIRST download preferences.

Description coming soon.

# Debugging your robot program

What can you do with the debugger

DRAFT

## C++ Tips

DRAFT

# Creating an application in WorkBench

DRAFT

# Using C with the WPI Robotics Library

DRAFT



# Contributing to the WPI Robotics Library

DRAFT

# Glossary

Concurrency

cRIO

deadlock

quadrature encoder

semaphore

task

VxWorks

DRAFT