C/C++ Getting Started Guide

Worcester Polytechnic Institute Robotics Resource Center

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

Rev 2.0

Jan 5, 2011

Table of Contents

Changes from Version 4.0 to 4.1

There have been several changes from Version 4.0 to 4.1. Most are source-code compatible meaning you won't have to change your source code. One change may require you to change your source code.

- Improved camera support. The AxisCamera class now is more robust and more resilient to the order of cRIO and camera power up. Also, intermittent power failures are less likely to cause problems.
- The camera library is much more efficient with image copies and image buffer allocations.
- The PC Video server (images sent back to the dashboard) is much faster now: It should in most cases run at the full camera frame rate. Also the lag was reduced to around 0.5 seconds for 320x240 pixel images; less for smaller images.
- The kit I2C accelerometer is now supported in the ADXL345 class.
- Note: **The Axis Camera and Image classes have been renamed** to have upper case first letters with camel case. The names are the same as before but now everything is consistently using the same capitalization convention. This one-time change requires a small change to your vision code.
- Many cleanups in the I2C code.
- The default gyro sensitivity now matches the 2010 kit gyro. You can still explicitly set the sensitivity in V/deg/sec using the `Gyro.SetSensitivity` method on the gyro class.
- Other bug fixes and improvements throughout the library.

What is the WPI Robotics Library

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

We believe that the object oriented programming paradigm best fits robot programming with WPILib, hence the use of C++ in this Guide, but C programming is also available. Here's an example of using C with the WPI Robotics Library.

The following C program demonstrates driving the robot forward for 2 seconds during the Autonomous period of the game and driving with arcade-style joystick steering during the Operator Control (or "teleop") period. (Notice that constants define the I/O 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);
```

Example 1: A simple C program to drive in autonomous and operator control modes.

The WPI 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.
- Let you understand it at all levels by making the full source code of the library available. E.g. you can study (and modify) the algorithms used by the gyro class for oversampling and integrating the input signal or just call it to get the current robot heading. You can work at any level.

Using Wind River Workbench and C/C++

The development tool suite to create C++ robot programs is called Workbench from Wind River. Wind River Workbench is a complete, professional 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 to the robot's flash memory so it will automatically start up when the robot is powered on.
- You can even use Subversion, a popular source code revision control system to manage your code and track changes. This is even more useful if your team has more than one programmer.

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 to set up.

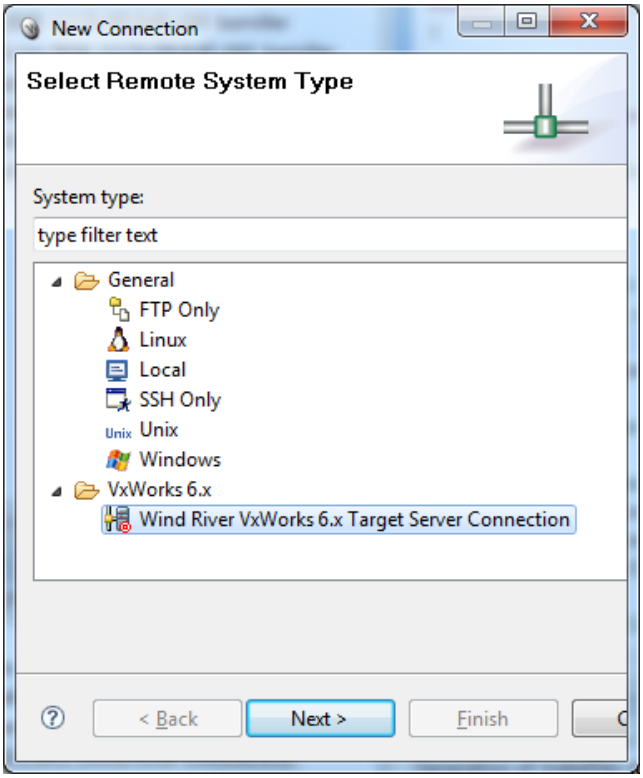
1. The target **remote system**, which is the robot cRIO that you will download your program to for testing and debugging.
2. The **run** or **debug configuration** that describes the program to debug and which remote system you want to debug it on.
3. The FIRST Downloader settings that tell which program to deploy to the cRIO when you're ready to download 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 that are 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 a cRIO and the location of a kernel file that is required for remote access. To create the entry for your system, perform the following steps.

Note: *To ensure the "Reset connected Target" command (this reboots the Target server, i.e. the Wind River OS on the cRIO) and other features work reliably, set the "Console out" switch on the cRIO to "on." This enables console text output via a serial cable to a laptop, which is very handy for debugging problems like when the robot program fails to start up. Leaving this switch on improves system reliability even when there's no serial cable connected. On the other hand, when you're using the serial port to control a Black Jaguar motor speed controller, you'll need to set the Console out switch to "off."*

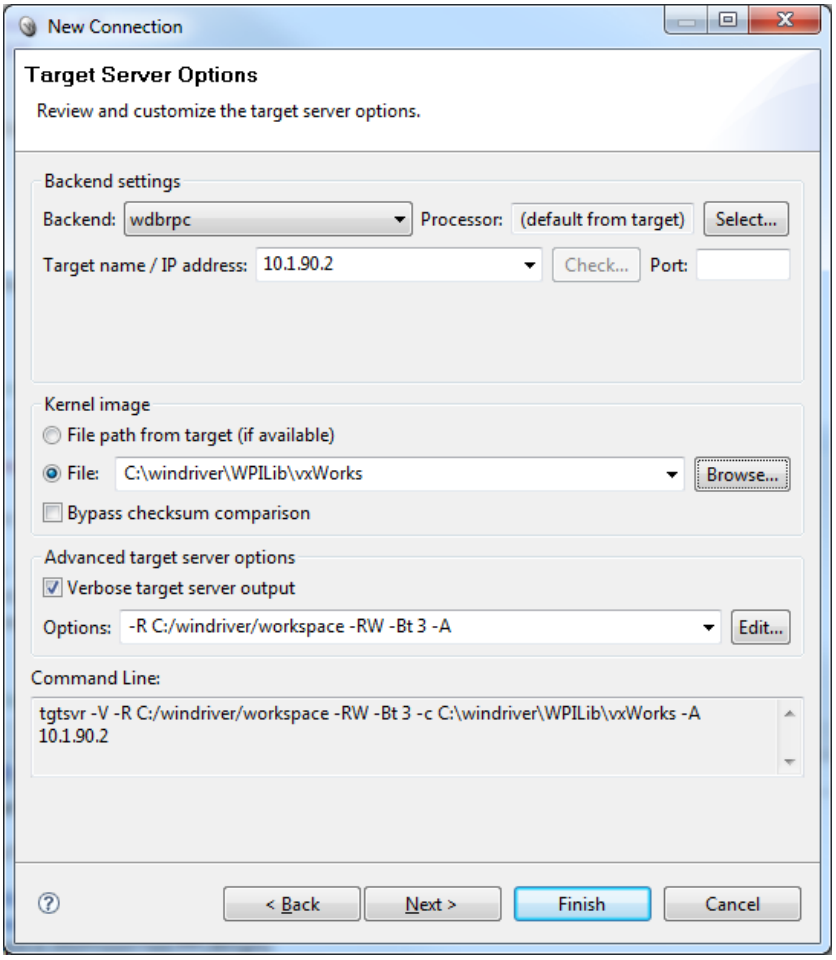
Right-click in the empty area in the "Remote Systems" window and select "New Connection."



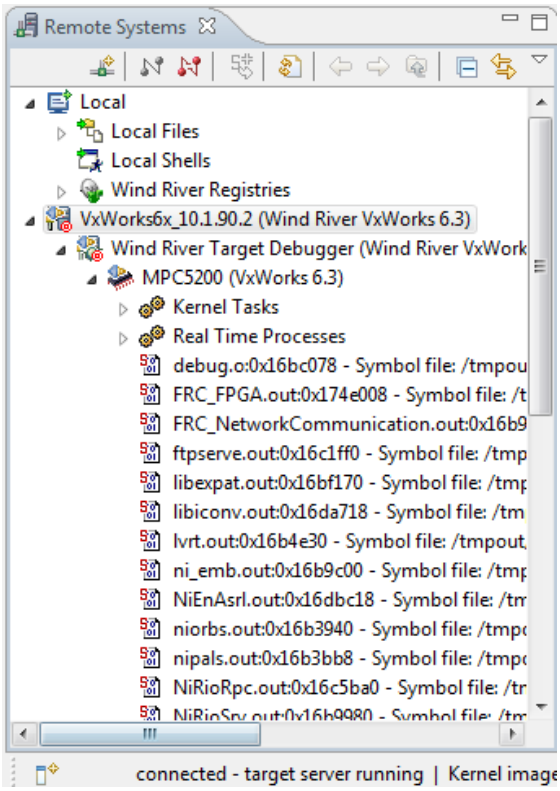
In the “Select Remote System Type” window choose “Wind River VxWorks 6.x Target Server Connection” and click “Next.”

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) is 10.1.90.2.

You must also select a Kernel Image file. This is located in the WindRiver install directory in the WPILib directory called “C:\WindRiver\WPILib\VxWorks”.



If the cRIO is turned on and connected the target server entry will now list the cRIO's running tasks.



Creating a robot project

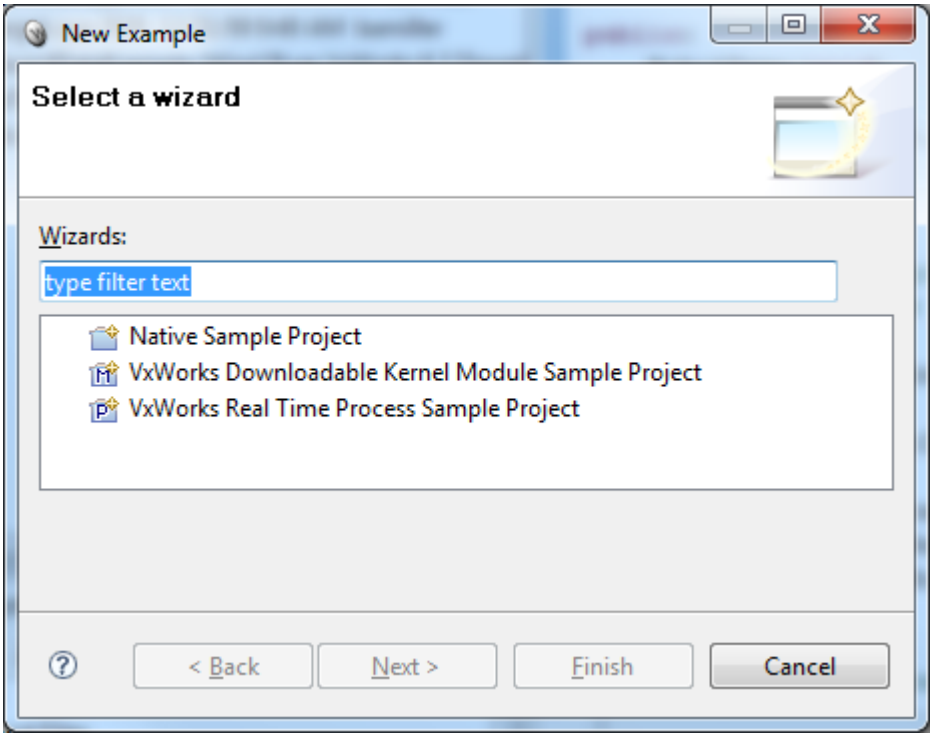
The easiest way to create your own project for your robot is to start with one of the existing templates:

- **`SimpleRobotTemplate`**
- **`IterativeRobotTemplate`**

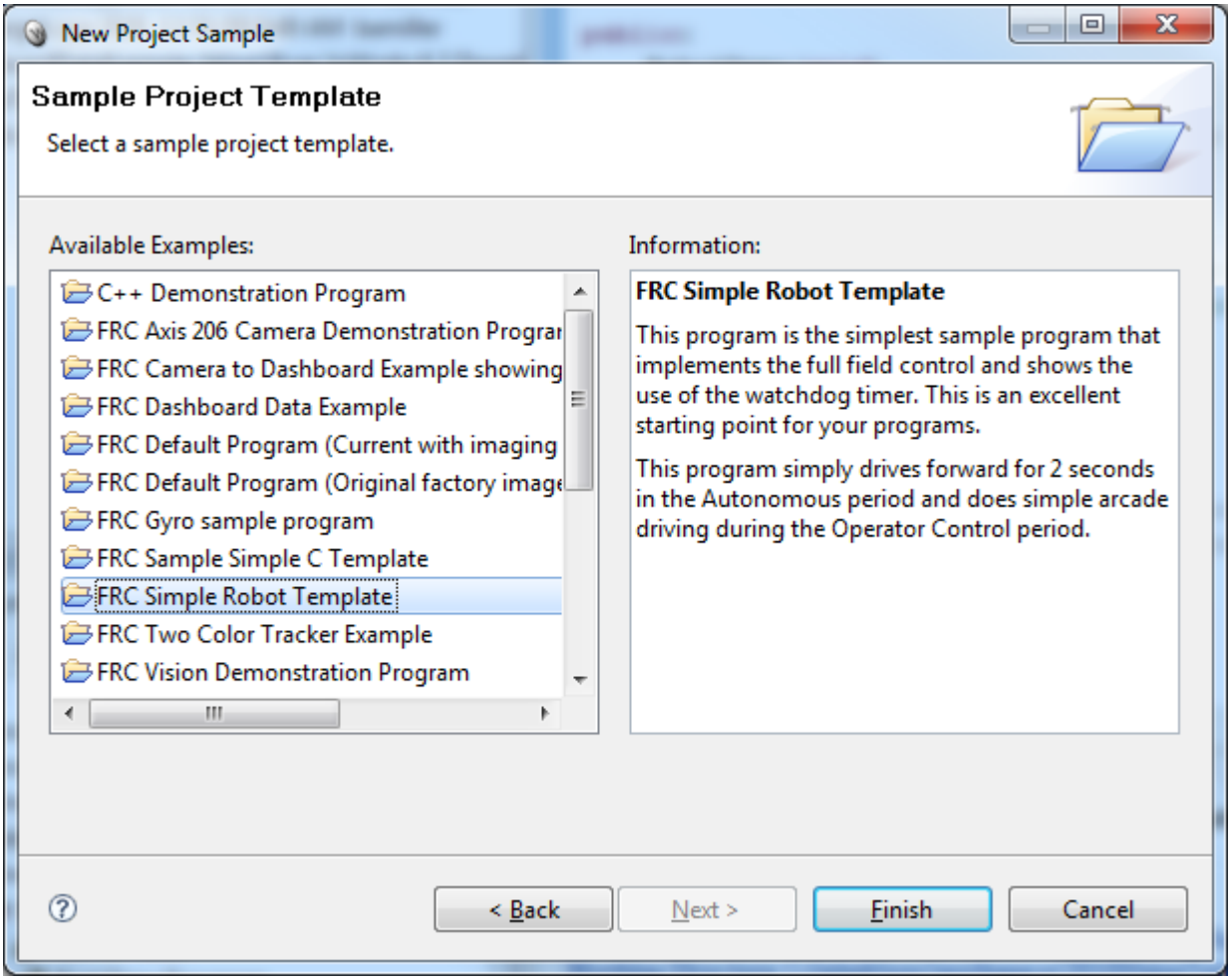
Both of these templates build on the **`RobotBase`** class and override some functions to change the behavior. More project templates are available.

Follow these steps to create a robot project. Here we'll use the **`SimpleRobotTemplate`** but you can start from any of the provided samples.

Click the main command File > New > Example… In the New Example wizard select "VxWorks Downloadable Kernel Module Sample Project" and then click "Next."



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


New Project Sample
Sample Project Template
Select a sample project template.
Available Examples:
C++ Demonstration Program
FRC Axis 206 Camera Demonstration Prograr
FRC Camera to Dashboard Example showing
FRC Dashboard Data Example
FRC Default Program (Current with imaging
FRC Default Program (Original factory image
FRC Gyro sample program
FRC Sample Simple C Template
FRC Simple Robot Template
FRC Two Color Tracker Example
FRC Vision Demonstration Program
Information:
FRC Simple Robot Template
This program is the simplest sample program that implements the full field control and shows the use of the watchdog timer. This is an excellent starting point for your programs.
This program simply drives forward for 2 seconds in the Autonomous period and does simple arcade driving during the Operator Control period.
< Back
Next >
Finish
Cancel

Building your project

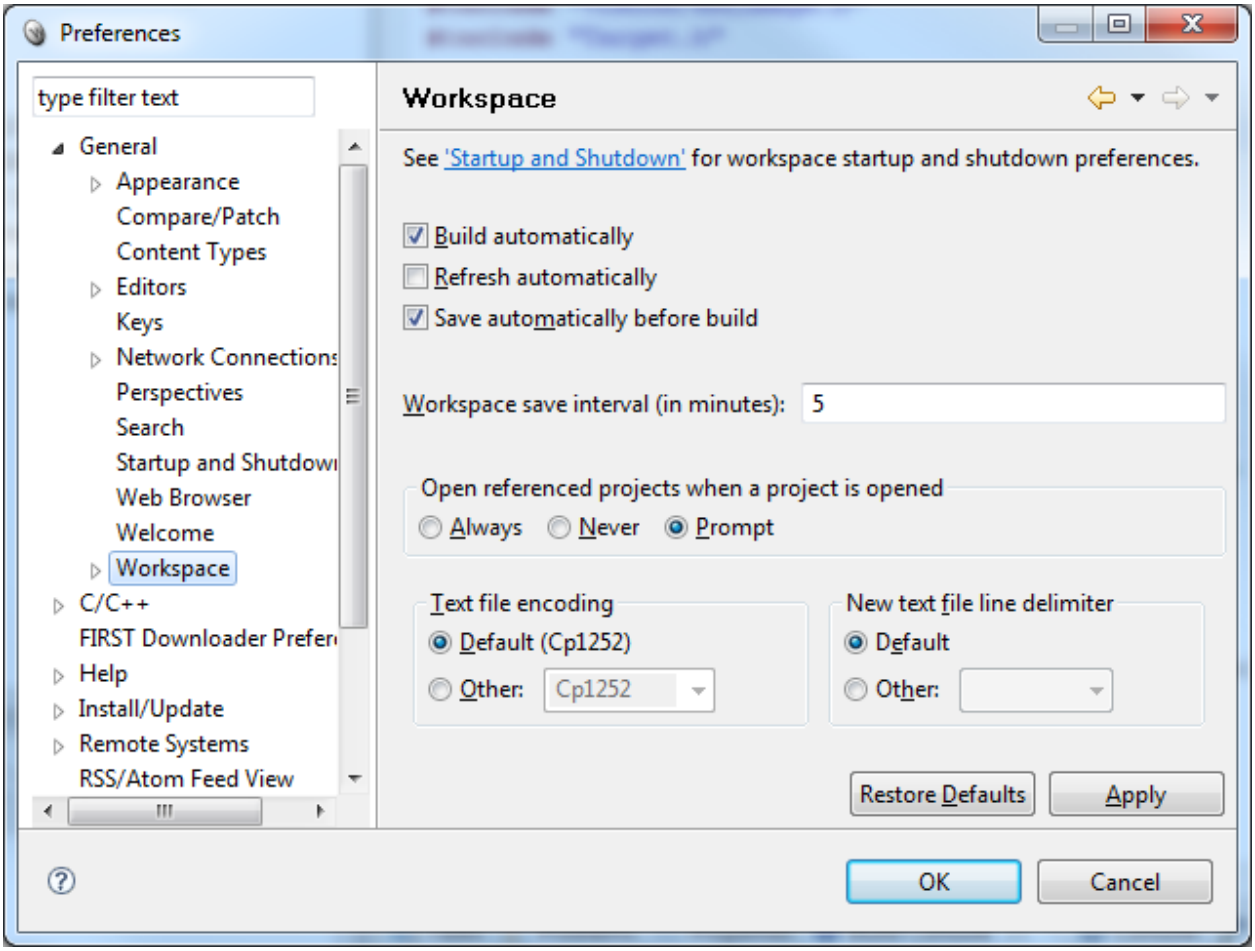
Build the project by right-clicking on the project name in the Project Explorer window and selecting "Build project" or "Rebuild project" from the popup context menu. This will make Workbench compile and link the project files into a .OUT executable file that may be deployed to or downloaded to the cRIO.

Note: *Sometimes after creating a new Workbench project, building it the first time doesn't actually build anything. You won't notice very much activity in the "Build Console" tab. If this happens, simply exit and restart Workbench and rebuild. This seems to only happen the first time a new project is built.*

Another way of building the project is to use Workbench's "automatic rebuild" feature. It will rebuild the project automatically whenever you save a source file. To enable this feature:

1. Select Window > Preferences.
2. In the Preferences panel, expand "General," then click "Workspace," and check the "Build automatically" option. Quickly save an edited file via the Ctrl-S keyboard shortcut, or save all open files at once using the Save All shortcut, Ctrl-Shift-S.

Tip: *Also turn on "Save automatically before build." Then Workbench will save your changes to all files before building. Otherwise it can build with only some of your edits, which doesn't work very well.*



Downloading the project to the cRIO

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

- Using a Run/Debug Configuration in Workbench. This loads the program into the cRIO RAM memory and allows it to run, with or without the debugger. When the robot is rebooted, the program will no longer be in memory.
- Deploy the program through the FIRST Downloader option in Workbench. This writes the program to flash memory in the cRIO where it will run whenever the cRIO reboots (that is, until you Undeploy it). This is how to make your program available for a match so it will run without an attached computer.

Caution: Be sure to not use the Run/Debug configuration if you have a robot program deployed and thus starting up automatically in the background. Having two robot programs trying to run at the same time is very confusing. Use the Undeploy menu item if you think there is already a deployed program.

It sometimes helps 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 the cRIO from your development computer by right-clicking on the connection in the "Remote Systems" tab in Workbench and selecting "Reset connected Target." Or reboot the cRIO via the Driver Station's Reboot option. It takes about 15 seconds to reboot.

Debugging your robot program

You can monitor, control, and manipulate cRIO processes using the debugger. This section will describe how to set up a debug session for a robot control program. (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, your program should call the macro **`START_ROBOT_CLASS`**. (See **`SimpleDemo`** or **`IterativeDemo`** for examples.) That starts an initial task that then starts the robot task with the correct run options. This makes it necessary to set up the debug options to attach to the spawned robot task instead of the initial task.

To start a debug session, first ensure the PC is connected to the target. Then right-click on the project name in the "Project Explorer" window and select "Debug Kernel Task..." to open the Debug dialog:

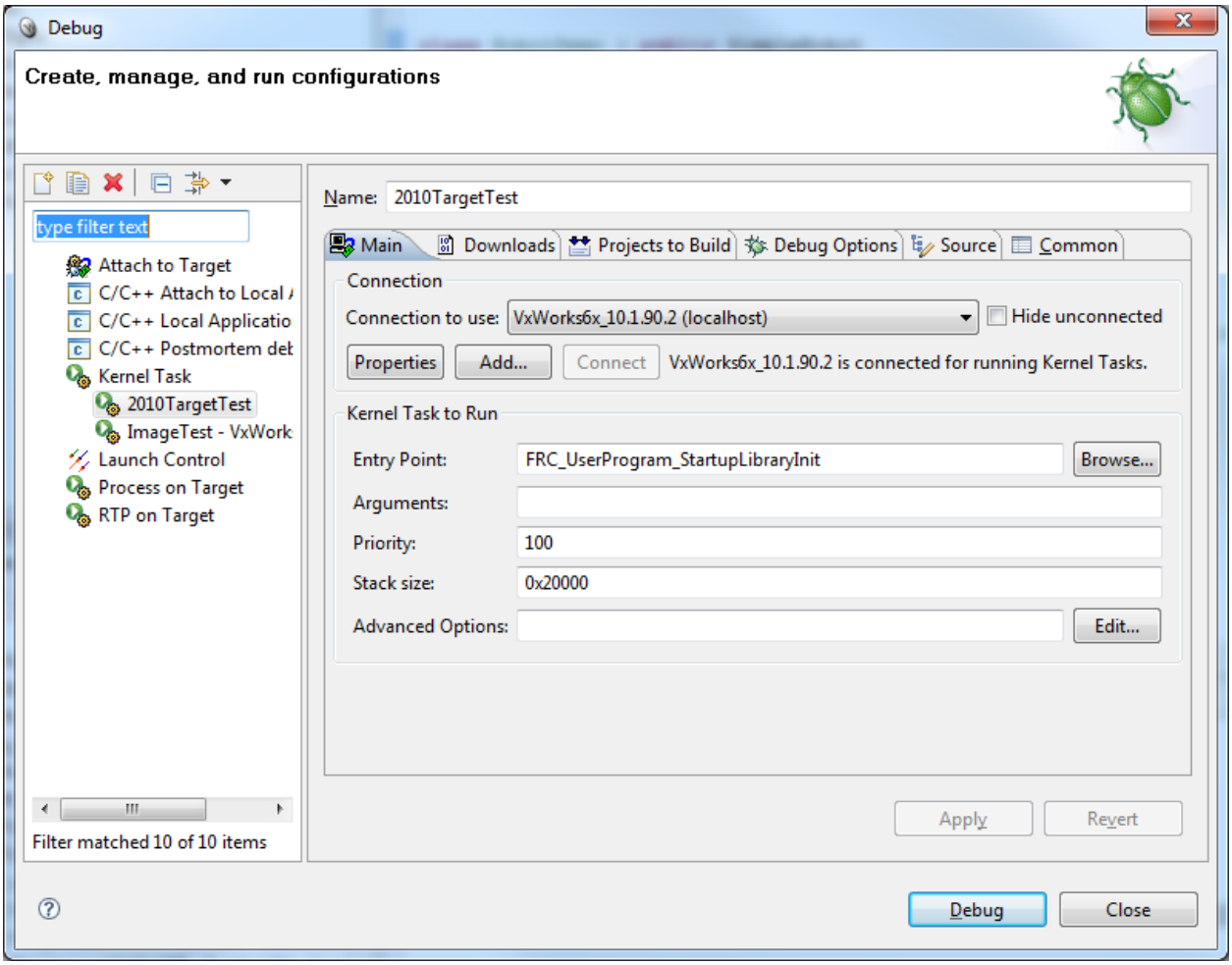


Figure 1: Setting the entry point on a Debug Configuration for a robot program.

Change the name of the debug target to something meaningful like "2010TargetTest" in the figure, or maybe "2011HighScoringRobot." Select as the entry point the function **`FRC_UserProgram_StartupLibraryInit`**.

In the Debug Options tab, select “Break on Entry” and “Automatically attach spawned Kernel Tasks.” This tells the debugger to stop at the program’s first instruction and make the spawned task (your robot task) available to debug.

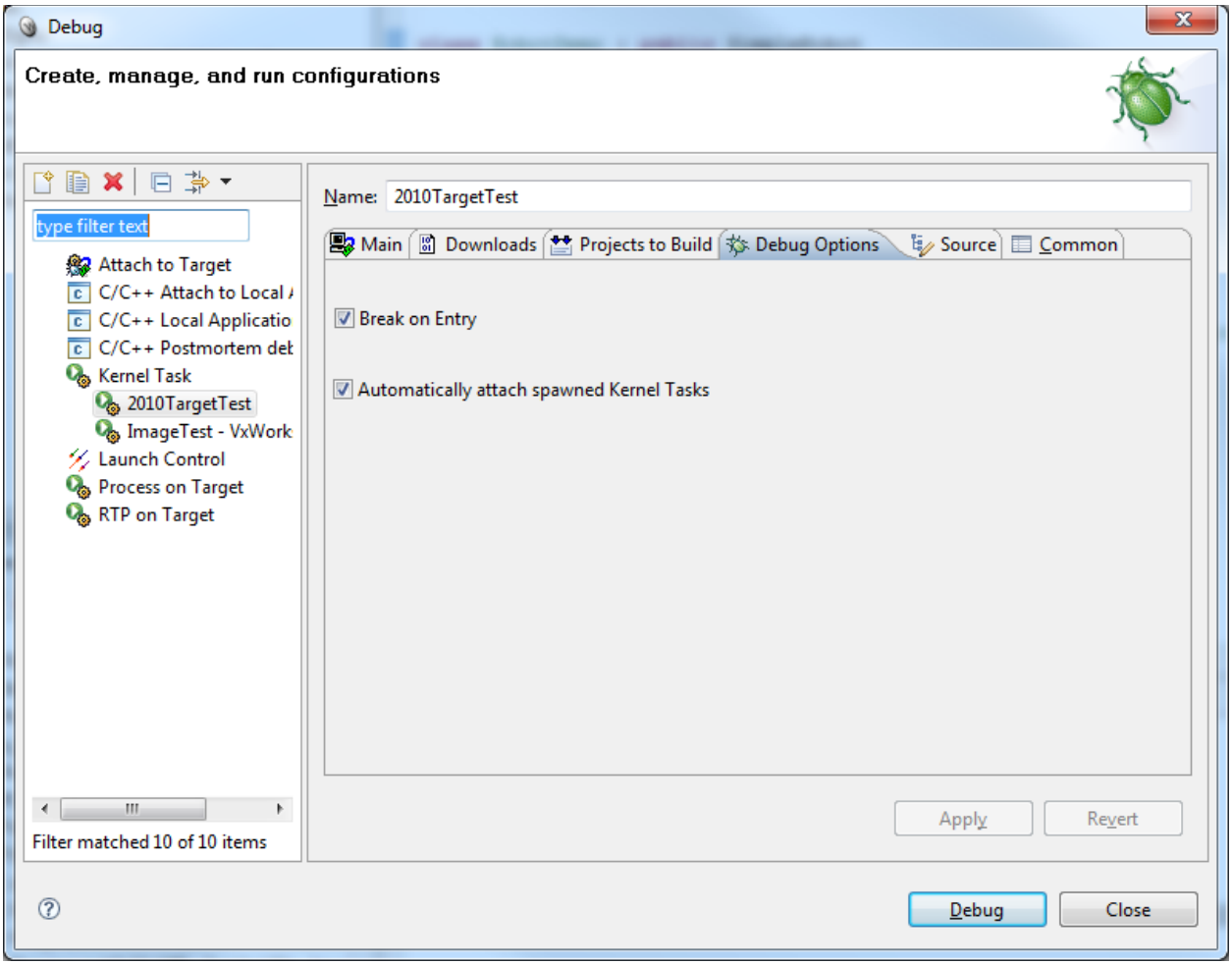
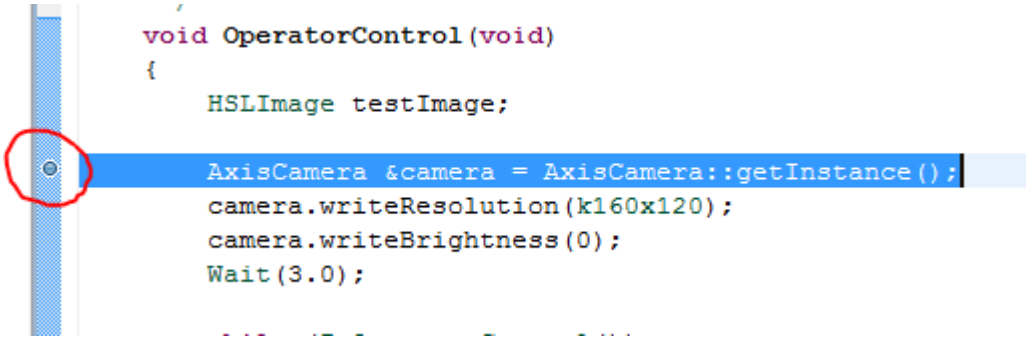


Figure 2: Setting the “Automatically attach spawned Kernel Tasks” option ensures that you will be able to debug the entire program including the robot's main task and any tasks that it creates.

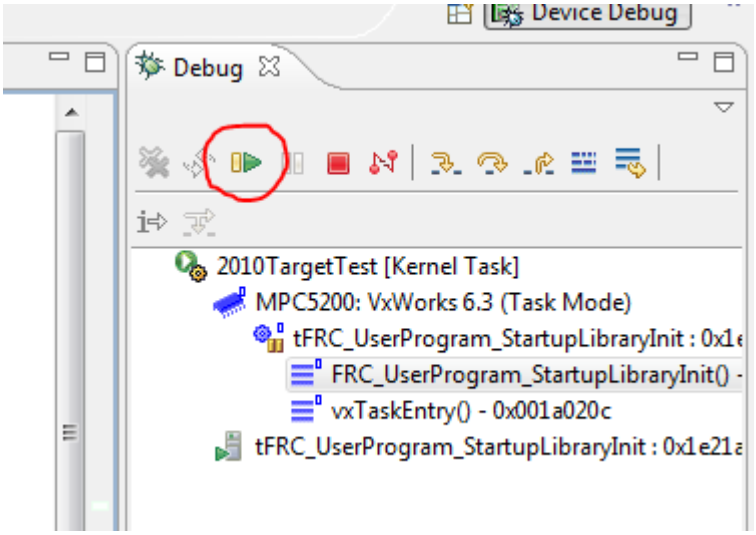
The other options can normally be left at default settings. Apply your changes.

Clicking the “Debug” button does several things. It changes the Workbench to the Debug Perspective which has the views Debug, Breakpoints, and Variables along the right side of the window. It starts the robot task and pauses it at the first program statement, in **`FRC_UserProgram_StartupLibraryInit`**. Now double-click in the left margin of the source code window to set a breakpoint in your user program:

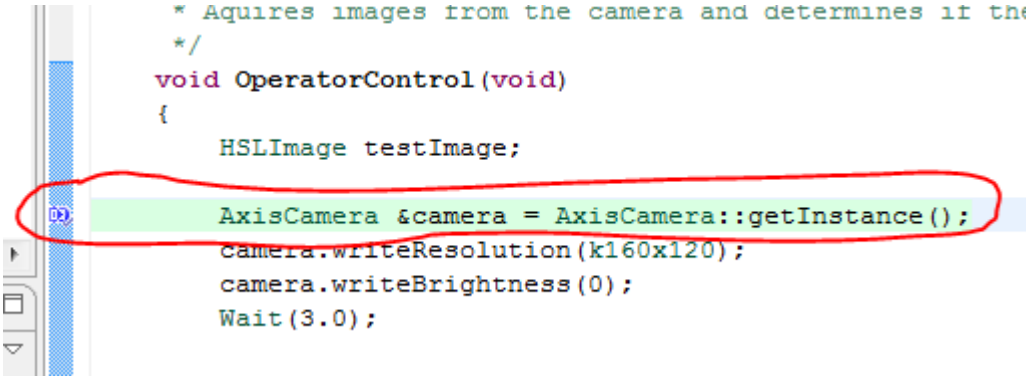


A small blue circle indicates the breakpoint has been set on the corresponding line. (You can see all your breakpoints in the Workbench’s Breakpoints view.)

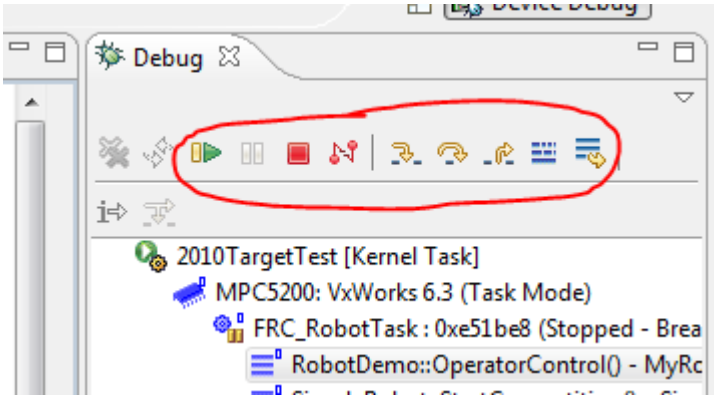
Click the “Resume” button (the green arrow) to resume program execution up to the first breakpoint.



The program will start running and then pause at the breakpoint:

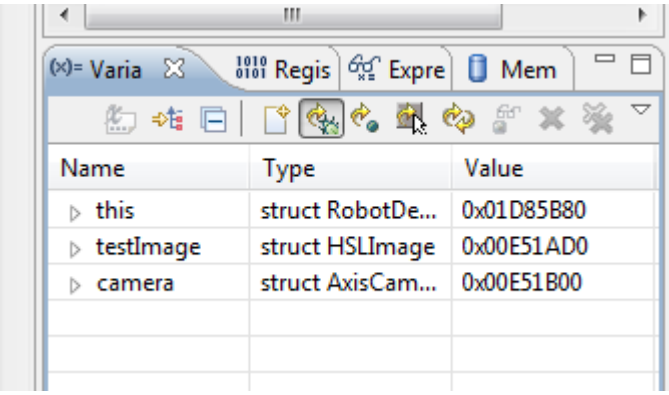


The Debug view shows all processes and threads running in the cRIO. Select the stack frame to see the current instruction pointer and source code (if available) for the selected process. When your breakpoint is reached, make sure your program is selected in the task list and your source code is displayed with a program pointer. You can continue stepping through your code using “Resume,” “Step Into,” “Step Over,” and “Step Return” buttons:



If you see assembly code instead of C++ code displayed, it’s because you’ve stepped down into library code where the source is not available to the debugger. “Step Return” will bring you back up a level.

The Variables view shows the current values of variables. To see a variable that is not displayed, select the “Expressions” tab and enter the variable name. This will show the variable’s value if it’s in scope.



To stop debugging, you can 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 the "Automatically attach spawned Kernel Tasks" option is on. When you click "Debug," the cRIO will first pause in initialization code, before it gets to your program. When you "Resume" it will soon begin your robot program.

Getting printf or cout output on the PC

Printing some text from your program is a handy way to debug and tune it. There are four ways to see printf or cout output, each with advantages and disadvantages:

Method of communication	Advantages	Disadvantages
Connect a serial cable between the computer and robot controller	Starts working early in bootup	Robot must be tethered. Takes up the serial port.
Use a network Target Console. To do that, right-click on the remote system, then "Target Tools," then "Target Console". This creates a console window that gets the text over the network.	Doesn't need the serial port or a serial cable	You have to do it again after each reboot. Doesn't start working early in bootup.
Allocate a console. In the Run menu, select Open Run Dialog… or Open Debug Dialog… Select your Kernel Task FRC_UserProgram_StartupLibraryInit in the left pane. Then find the "Common" tab and check the "Allocate Console" checkbox.	Survives reboots	Only shows the current task's output
NetConsole over Ethernet	Captures all stdout, stderr, and VxWorks commands	Requires NetConsole to be running on the cRIO

The NetConsole is now the preferred way to get output back from the cRIO. It has the advantage of starting up early on as the cRIO boots so all the informational messages are forwarded to the user and shows output from all tasks running. The use of Netconsole can be enabled by using the Imaging Tool and checking the NetConsole checkbox when reimaging the controller to enable NetConsole output.

NetConsole is automatically installed with the NI utilities at the same time as the Imaging tool and other utilities. You can start it by finding it in the start menu under LabVIEW.

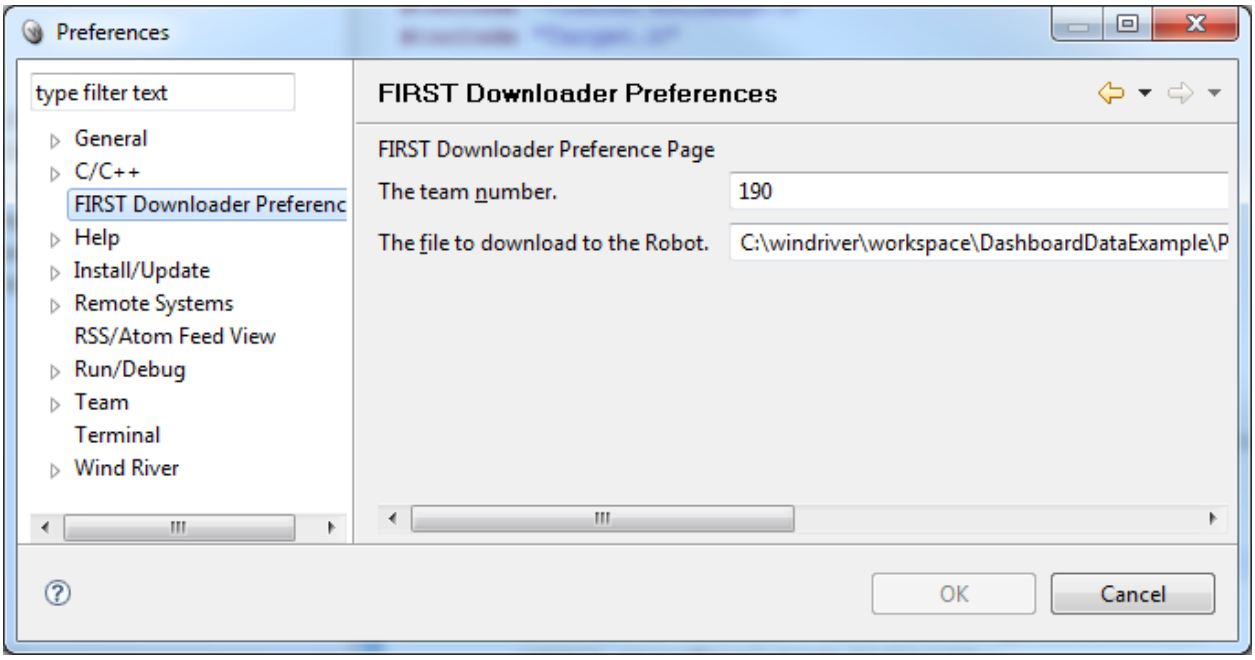
Note: Netconsole will only work if the netmask on the interface connected to the robot is set to 255.0.0.0. Everything else will work with a netmask of 255.255.255.0, but NetConsole will not connect.

Deploying the C/C++ Program

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

Caution: 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. Having both robot programs try to start up is unhelpful.

Before deploying a program for the first time, you need to configure the "FIRST Downloader Preferences." This is done via 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 step is easy to do after you built it.

Once this is set up, deploy the project using the menu command FIRST > Download. It will copy the .OUT file to the correct directory and filename on the cRIO. Now when the cRIO restarts, it will automatically run this program.

To undeploy the program, 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 at half speed for 2 seconds to make each side then turn 90 degrees. Repeat 4 times.
Operator Control period	Uses two joysticks to provide tank steering for the robot.

Robot specifications:

Left drive motor	PWM port 1
Right drive motor	PWM port 2
Joystick	Driver station joystick ports 1 and 2

Starting with the simple template for a 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);
```

Example 2: Starting point for robot program

Now define a robot drive object for motors in ports 1 and 2 and joystick objects for joystick ports 1 and 2:

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

Example 3: C++ Example 2: Adding robot drive and joystick objects to the program

To simplify this example, we'll disable the watchdog timer. (The watchdog is a safety feature in the WPI Robotics Library that helps keep your robot from running away, out of control if the program malfunctions. You don't really want to disable it. If necessary, give it a longish timeout.)

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

Example 4: Disabling the watchdog timer for this simplified example

Now write the autonomous part of the program to drive in a square:

```
void Autonomous()
{
	for (int i = 0; i < 4; i++)
	{
		drive.Drive(0.5, 0.0);  // drive 50% forward speed with 0% turn
		Wait(2.0);              // wait 2 seconds
		drive.Drive(0.0, 0.75); // drive 0% forward speed with 75% turn
		Wait(0.75)              // Wait for the robot to turn 90 degrees
	}
	drive.Drive(0.0, 0.0);        // drive 0% forward speed with 0% turn (stop)
}
```

Example 5: Autonomous program that drives in a square

Now write the operator control part of the program:

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

Example 6: Simple tank drive with two joysticks

This applies joystick control values to the drive motors, every 5 milliseconds.

Putting it all together we get this very 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++)
		{
			drive.Drive(0.5, 0.0);  // drive 50% forward speed with 0% turn
			Wait(2.0);              // wait 2 seconds
			drive.Drive(0.0, 0.75); // drive 0% forward speed with 75% turn
			Wait(0.75);             // turn for 3/4 second
		}
		drive.Drive(0.0, 0.0);          // drive 0% forward speed with 0% turn (stop)
	}

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

START_ROBOT_CLASS(RobotDemo);
```

Example 7: Completed example program. Notice the 0.75 second Wait that is a amount computed for the sample robot to complete a 90 degree turn.

Some details:

- In this example **`drive`**, **`leftStick`**, and **`rightStick`** are member objects of the **`RobotDemo`** class. They're accessed using references, one of the ways of accessing objects in C++. The next section will introduce pointers as an alternate technique.
- The **`drive.Drive()`** method takes two parameters: a speed and a turn rate. See the documentation about the **`RobotDrive`** object for details on how the speed and direction parameters work.
- Disabling the Watchdog safety timer is a bad idea! You should enable the watchdog timer, set its feeding interval, and "**`Feed`**" it at least that often.

Using objects

The WPI Robotics Library accesses all sensors, motors, driver station elements, and more through objects. Most objects correspond to the physical things on your robot like sensors. Objects have the code and the data needed to operate that physical thing. Let's look at a gyro. The operations (or methods) you can perform on a gyro are:

- Create the gyro object – this sets up gyro communications and calibrates the gyro
- Configure the gyro parameters, i.e. its sensitivity
- Get the current heading, or angle, from the gyro
- 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);
```

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

Note: *An instance of an object has a block of memory for that instance's data. When you create an object, that memory block gets allocated and when you delete the object that memory block gets deallocated.*

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

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

This sets the variable **`heading`** to the current heading from the gyro on analog channel 1.

Creating object instances

There are several ways of creating object instances in C++. These ways differ in how the object should be referenced and deleted. Here are the rules:

Location	Create object	Use the object	When the object is deleted
Variable declared inside an object, function, or block	`Victor leftMotor(3);`	`leftMotor.Set(1.0);`	Object is automatically deleted when the enclosing block exits or the enclosing object is deleted
Global declared outside of any enclosing objects or functions; or a static variable	`Victor leftMotor(3);`	`leftMotor.Set(1.0);`	Object is not deleted until the program exits
Pointer to object	`Victor *leftMotor = new Victor(3);`	`leftMotor->Set(1.0);`	Object must be explicitly deleted using the C++ **`delete`** operator.

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

Pointers and addresses

There are two ways of declaring an object variable: either as an instance of the object or a pointer to an instance of the object. In the former case the variable holds the object and the object is created ("instantiated") at the same time. In the latter case the variable only has space to hold the address of the object. It takes another step to create the object instance using the **new** operator and assign its address to the variable. Look at these two code snippets to see the difference:

```
Joystick stick1(1);           // this is an instance of a Joystick object stick1
stick1.GetX();                // access the instance using the dot (.) operator
bot->ArcadeDrive(stick1);     // you can pass the instance to a method by reference
                              //    … ArcadeDrive(Joystick& j) …

Joystick *stick2;             // a pointer to an uncreated Joystick object
stick2 = new Joystick(1);     // creates the instance of the Joystick object
stick2->GetX();               // access the instance with the arrow (->) operator
bot->ArcadeDrive(stick2);     // you can pass the instance by pointer (notice, no &)
                              //    … ArcadeDrive(Joystick* j) …
delete stick2;                // delete the object when you're done with it
```

ArcadeDrive() in WPILib takes advantage of a C++ feature called *function overloading*. This allows it to have two methods with the same name that differ by argument lists. The one **ArcadeDrive(Joystick &j)** takes the parameter **j** as a reference to a **Joystick** instance. You supply a **Joystick** and the compiler automatically passes a reference to that instance. The other **ArcadeDrive(Joystick *j)** takes the parameter **j** as a pointer to a **Joystick** instance. You supply a pointer to a **Joystick** instance. The cool thing is that the compiler figures out which **ArcadeDrive** to call. The library is built this way to support both the pointer style and the reference style.

If you had non-overloaded functions **Ref(Joystick &j)** and **Ptr(Joystick *j)**, you could still call them if you use the right C++ operators: **Ref(*stick2)** and **Ptr(&stick1)**. At run time, references and pointers are both passed as addresses to the instance. The difference is the source code syntax and details like allocation and deletion.

WPI Robotics Library Conventions

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

Class, method, and variable naming

Names follow these conventions:

Type of name	Naming rules	Examples
Class name	Initial upper case letter then camel case (mixed upper/lower case) except acronyms which are all upper case	**Victor**, **SimpleRobot**, **PWM**
Method name	Initial upper case letter then camel case	**StartCompetition**, **Autonomous**, **GetAngle**
Member variable	"**m_**" followed by the member variable name starting with a lower case letter then camel case	**m_deleteSpeedControllers**, **m_sensitivity**

Local variable	Initial lower case	`targetAngle`
Macro	All upper case with _ between words. **Note:** It's better to use const values and inline functions than macros.	`DISALLOW_COPY_AND_ASSIGN`

Constructors with slots and channels

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

- Specify an I/O port with the slot number followed by the channel number. The slot number identifies the cRIO chassis slot that the I/O module is plugged into. An analog module can be connected to chassis slot 1 or 2. The channel number identifies which of the module's I/O channels the device is wired to.
- Since many robots can be built with only a single analog or digital module, there is a shorthand way to specify a 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)
```

Example 8: Sharing inputs between objects

WPILib object constructors generally use the port number(s) to select and reserve cRIO input and output channels. E.g. when you instantiate an encoder object, it reserves that digital input channel.

Built-in Robot classes

There are several built-in robot base classes to help you quickly create a robot program. Subclass the one that best fits your requirements and preferences.

Table 1: Built-in robot base classes to create your own robot program.

Class name	Description
SimpleRobot	This template is the easiest to use and is designed for writing a straight-line autonomous routine without complex state machines. Pros: • There are only three places to put your code: the constructor for initialization, the `Autonomous` method for autonomous code and the `OperatorControl` method for teleop code. • Sequential robot programs are trivial to write. Just code each step one after another. • No state machine is required for multi-step operations. The program can simply do each step sequentially. Cons: • Switching between autonomous and operator control code may require

	rebooting the controller if your program gets stuck in a loop. • The **`Autonomous`** method will keep running until it decides to exit, so it could continue to run during the operator control period. You don't want that. So be sure to make your loops check if the field state is still in the autonomous period.
IterativeRobot	This template gives additional flexibility in the code for responding to various field state changes (autonomous, operator control, disabled) in exchange for additional complexity in your program. **`IterativeRobot`** repeatedly calls one of your methods depending on the current field state. The intent is that each method will do some processing for that field state and then return. That way, as soon as the field state changes, **`IterativeRobot`** starts calling a different method. Pros: • You get fine-grain control of field state changes, especially if you're practicing and retesting the same field state over and over. Cons: • It's more difficult to write action sequences that unfold over time, e.g. an autonomous sequence. That requires state variables to remember what the robot was doing from one call to the next.
RobotBase	The base class for the above classes. This provides all the basic functions for field control, the user watchdog timer, and robot status. Extend this class to if you need more flexibility.

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 to fill in to complete a **`SimpleRobot`** program.

Table 2: SimpleRobot class methods that are called as the field state progresses through the match

Method	What it does
the Constructor (method with the same name as the robot class)	Put code in the constructor to initialize the sensors, motors, pneumatics, and your robot program variables. This code runs as soon as the robot is turned on, before it's enabled, that is, before it can operate motors and other actuators. When the constructor exits, the program will wait until the robot is enabled.
Autonomous()	Put code in the **`Autonomous`** method to run the robot during the autonomous period of the match. When this method exits, the program will wait until the start of the operator control period. This method had better detect the end of the autonomous period since it won't be interrupted when it's time for operator control. If this method has an infinite loop, it won't stop until the entire match ends.

OperatorControl()	Put code in the `OperatorControl` method to run the robot 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 organizes your robot program (your subclass) into methods that it calls according to the match state. For example, it calls your `AutonomousContinuous` method repeatedly during the autonomous period. When the playing field (or the Driver Station setting) changes to operator control, it calls the `TeleopInit` method then starts calling the `TeleopContinuous` method repeatedly.

WindRiver Workbench has a sample robot program based on the `IterativeRobot` base class. If you want to use it, follow the instructions from the previous section except select "Iterative Robot Main Program." This will create the project and a skeletal robot program in your workspace.

When basing a robot program on the `IterativeRobot` base class, you implement these methods:

Table 3: IterativeRobot calls these methods of your robot program as the match proceeds:

Method name	Description
RobotInit	Called when the robot is first turned on. You put initialization code here or in the constructor. This method is only called once.
DisabledInit	Called once each time the robot becomes disabled.
AutonomousInit	Called once when the match enters the autonomous period from any other state.
TeleopInit	Called once when the match enters the teleoperated period from any other state.
DisabledPeriodic	Called periodically during the disabled part of the match, using a periodic timer.
AutonomousPeriodic	Called periodically during the autonomous part of the match, using a periodic timer.
TeleopPeriodic	Called periodically during the teleoperated part of the match, using a periodic timer.
DisabledContinuous	Called continually during the disabled part of the match. When this method returns, it gets called again if the match state hasn't changed.
AutonomousContinuous	Called continually during the autonomous part of the match. When this method returns, it gets called again if the match state hasn't changed.
TeleopContinuous	Called continually during the teleoperated part of the match. When this method returns, it gets called again if the match state hasn't changed.

The three Init methods are called on transition into the relevant field state. 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. Call the `IterativeRobot` class's `SetPeriod` method to set the period interval. The periodic methods are intended for time-based algorithms like PID control. During each match state, its periodic and continuous methods will both be called, at different rates.

RobotBase class

The **RobotBase** class is the superclass of the **SimpleRobot** and **IterativeRobot** classes. If you decide not to build on **SimpleRobot** or **IterativeRobot**, then subclass **RobotBase** directly. **RobotBase** has all the methods to determine the field state, set up the watchdog timer, handle communications, and do other housekeeping work.

Create a subclass of **RobotBase** and implement at least the **StartCompetition** method, much like the **SimpleRobot** class does.

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);	// wait out the disabled period
	OperatorControl();			// start user-supplied OperatorControl code
}
```

It uses the **IsDisabled** and **IsAutonomous** methods of **RobotBase** to determine the field state, then calls the correct methods as the match progresses.

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

Watchdog timer class

The Watchdog timer will stop the robot's motors and pneumatics if the program goes into an infinite loop or crashes. A watchdog object is created inside the **RobotBase** class (the base class for all robot programs). Your robot program is responsible for "feeding" the watchdog periodically by calling the **Feed()** method on the Watchdog. If you don't feed the Watchdog often enough, it will stop all of the robot's motors and pneumatics.

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.

Using 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 drive for 2 seconds then make a turn. The easiest way to do this is to start the robot driving, and then call **Wait** to wait for 2 seconds. During the 2-

second wait, it can't 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. *A better approach is to set a longer watchdog timeout period so you still get most of the watchdog protection.*

```
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 in a **`RobotBase`** subclass.

Waiting for 2 seconds has another problem: This robot program's teleoperated code will start up to 2 seconds late if autonomous mode ends near the start of those 2 wait seconds since the teleoperated code won't start until the **`Autonomous`** method returns.

C++ Tips

Contributing to the WPI Robotics Library

We are accepting requests from teams or groups of team members who would like to create Community projects to share with other FRC teams. Those projects can be hosted on our SourceForge server located at http://firstforge.wpi.edu. These community projects will have an administrator member and contributors to the project. All the project code will be readable by the entire FIRST community but only designated project members can make changes.