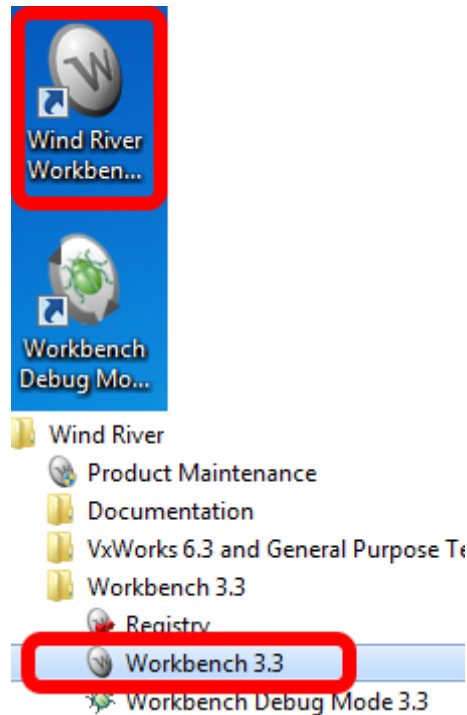


Creating a robot project

Creating a robot project

Liquid error: No route matches {:action=>"show", :controller=>"spaces/chapters", :space_id=>"3120", :manual_id=>"7952", :id=>nil}

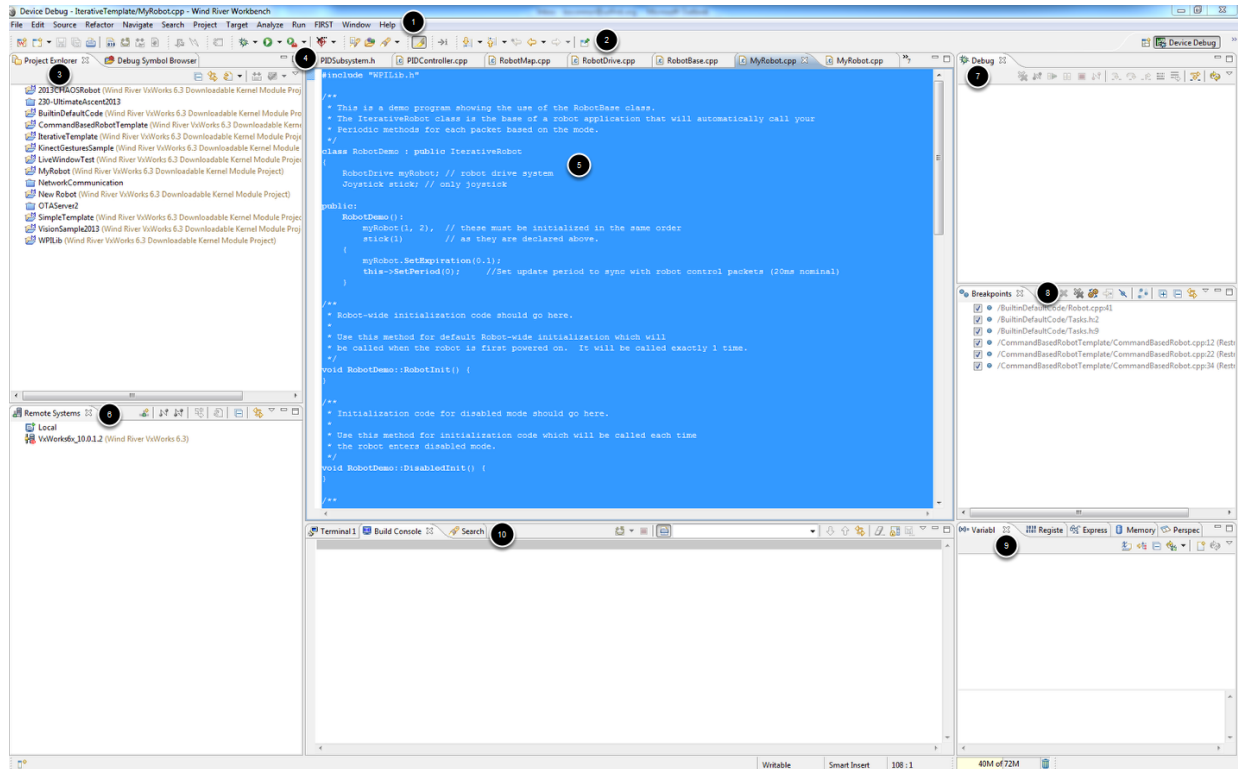
Launching WindRiver Workbench



WindRiver Workbench will create two icons on your desktop and two entries in your Start Menu program listing, one will be called Wind River Workbench 3.3 and one will be called Workbench Debug Mode 3.3. It is recommended to always use the regular Workbench 3.3 program, all documentation and testing is done using this mode and not the Debug Mode

Creating a robot project

Workbench Overview



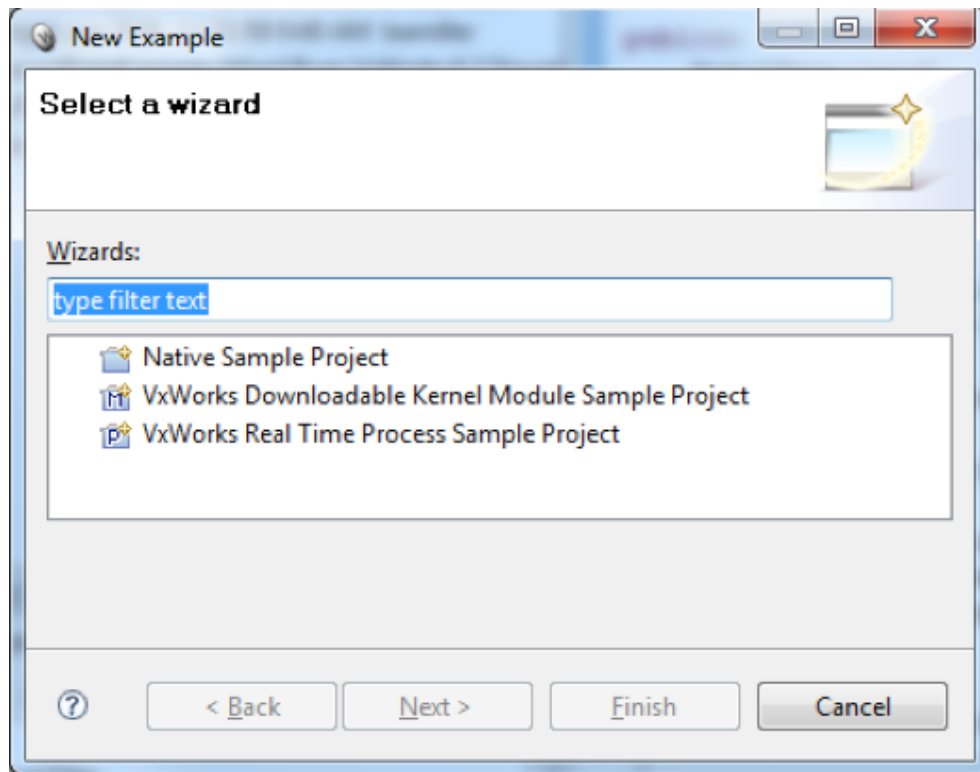
Above is an overview of the Workbench IDE in the Device Debug perspective. The Basic Development perspective is similar but with a few of the areas shown removed to maximize the space for the code window. To open the Device Debug perspective, go to Window->Open Perspective->Device Debug. Note that this image shows the default layout of the Device Debug Perspective, any tab other than the Editor tabs can be dragged to any of the other window sections and any of the sections can be resized as desired.

1. Menu Bar
2. Toolbar - Hover over each icon for a description
3. Project Explorer - This tab allows you to navigate the projects in the workspace and the files within the project. You can also right click on the project to perform operations such as builds.
4. Editor tabs - Tabs showing files open in the editor. Click the tab to switch to the file
5. Editor Window - Window for editing files
6. Remote Systems Tab - Tab for setting up remote systems connections. You will use this tab when setting up a connection to your cRIO to use the WR debugger
7. Debug tab - When debugging this tab will show information about the current threads loaded in the debugger and contain controls to manage the code execution flow.

Creating a robot project

8. Breakpoints - This tab contains a list of breakpoints. When not actively debugging a project it will show all breakpoints in all open projects in the workspace. (open projects are denoted by an open folder icon in the Project explorer tab)
9. Other Debugging Tools - This section contains tabs for the other debugging tools such as the Variables tab and the Expressions tab.
10. Build Console - This section contains the Build Console and search results tab. It will display diagnostic messages about the build process when executing a build and will show console messages when debugging.

Creating a robot program using the SimpleRobot template

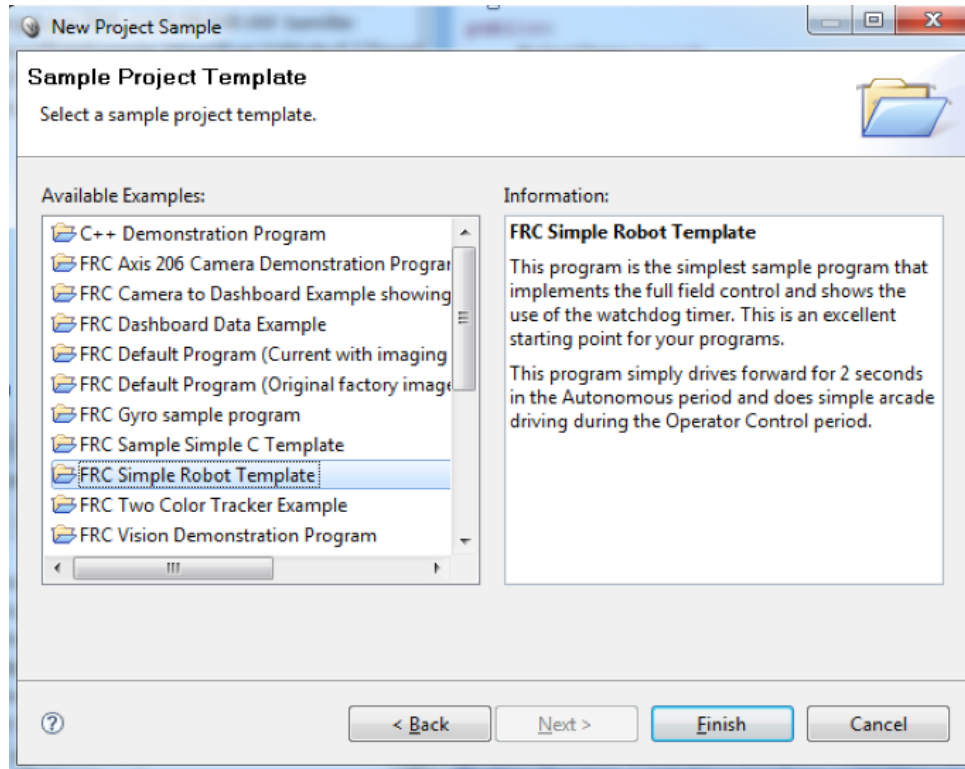


Follow these steps to create a robot project. Here we'll use the [SimpleRobotTemplate](#) but you can start from any of the provided samples. The SimpleRobot Template will generate code to use a joystick on USB port 1 for arcade drive with two motors on PWM ports 1 and 2. A very basic autonomous mode is also included.

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

Creating a robot project

Choosing the template



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.

Creating a robot project

The default SimpleRobot template project

```
#include "WPIlib.h"

class RobotDemo : public SimpleRobot
{
public:
  1 RobotDemo(void)
  {
    //put initialization code here
  }
  2 void Autonomous(void)
  {
    //put autonomous code here
  }
  3 void OperatorControl(void)
  {
    //put teleop code here
  }
};

START ROBOT CLASS(RobotDemo);
```

After creating the project using the SimpleRobot template you'll get a C++ project that has the basic form shown here, we'll explore the code contained inside each section in the steps below. It has three main parts:

1. Constructor: this is where you put initialization code that runs when the class is created. In this case, it will run when the program first starts, before the robot is enabled. It is a good place to initialize sensors, and create other WPIlib objects that you want to use. *Remember: since the robot isn't yet enabled, you can't use the constructor to position actuators because motors will not operate.*
2. Autonomous method: this is where you put any code that should run during the autonomous period. The Autonomous method will run every time the robot is put into Autonomous mode. Be aware that this method **will not be stopped at the end of the autonomous period**, so you must be careful to either ensure that the code you write doesn't take longer than the autonomous period in the match or put checks into the code to return when the autonomous period exits.
3. OperatorControl method: will be called when the robot enters the teleoperation part of the match. Your code here is typically a loop that reads operator interface values (joysticks and switches) and operates actuators until the teleop period has ended.

Creating a robot project

Defining the variables for our sample robot

```
class RobotDemo : public SimpleRobot
{
  1 RobotDrive myRobot; // robot drive system
  Joystick stick; // only joystick

public:
  RobotDemo(void) :
    2 myRobot(1, 2), // these must be initialized in the same order
      stick(1)      // as they are declared above.
    {
      3 myRobot.SetExpiration(0.1);
    }
}
```

The sample robot in our examples will have a joystick on USB port 1 for arcade drive and two motors on PWM ports 1 and 2 (If your robot has 4 motor controllers or the motor controllers are connected to different ports, make sure to change this line accordingly. The constructors are ordered left motor(s) then right motor(s).). Here we create objects of type RobotDrive (myRobot) and Joystick (stick). This section of the code does three things:

1. Defines the variables as members of our RobotDemo class.
2. Initializes the variables as part of the constructor using an Initialization List.
3. Performs robot initialization (in this case sets the safety timer expiration for the myRobot object to .1 seconds, see the next step for an explanation of motor safety timers).

Simple autonomous sample

```
void Autonomous(void)
{
  myRobot.SetSafetyEnabled(false);
  myRobot.Drive(-0.5, 0.0); // drive forwards half speed
  Wait(2.0);               // for 2 seconds
  myRobot.Drive(0.0, 0.0); // stop robot
}
```

The sample autonomous program here drives the program drives the robot at half speed (-0.5) and a turn rate of (0.0). A negative speed is used to make the robot drive forward because the joysticks provided in the Kit of Parts (and most other HID joysticks and gamepads) return a negative value when pushed forwards. Then the program waits for 2.0 seconds while the robot continues to drive at half speed. After the wait tell the RobotDrive object to stop (drive 0.0 speed forward).

Creating a robot project

The first line of the method disables motor safety for the autonomous program. Motor safety is a mechanism built into the RobotDrive object that will turn off the motors if the program doesn't continuously update the motor speed. In this case, the speed is updated once, then there is a 2 second delay before it's updated again. The default setting for motor safety is to require an update every 100 ms. By turning off motor safety, it will prevent the motors from turning off after the first 0.1 seconds.

Easy tank drive for teleoperation

```
void OperatorControl(void)
{
    myRobot.SetSafetyEnabled(true);
    while (IsOperatorControl())
    {
        myRobot.ArcadeDrive(stick); // drive with arcade style (use right stick)
        Wait(0.005);                // wait for a motor update time
    }
}
```

The teleoperation part of the program simply loops while the robot is in operator control mode and does arcade drive. Notice the 5 millisecond wait in the loop. This ensures that other threads in the program will have time to run. This won't effect performance since the driver station only updates the robot with new operator interface values every 20 milliseconds.

Creating a robot project

The finished program

```
#include "WPIlib.h"

class RobotDemo : public SimpleRobot
{
    RobotDrive myRobot; // robot drive system
    Joystick stick; // only joystick

public:
    RobotDemo(void) :
        myRobot(1, 2), // these must be initialized in the same order
        stick(1) // as they are declared above.
    {
        myRobot.SetExpiration(0.1);
    }

    void Autonomous(void)
    {
        myRobot.SetSafetyEnabled(false);
        myRobot.Drive(-0.5, 0.0); // drive forwards half speed
        Wait(2.0); // for 2 seconds
        myRobot.Drive(0.0, 0.0); // stop robot
    }

    void OperatorControl(void)
    {
        myRobot.SetSafetyEnabled(true);
        while (IsOperatorControl())
        {
            myRobot.ArcadeDrive(stick); // drive with arcade style (use right stick)
            Wait(0.005); // wait for a motor update time
        }
    }
};

START_ROBOT_CLASS(RobotDemo);
```

Some details:

- In this example `myRobot`, and `stick` are member objects of the `RobotDemo` class. They're accessed using references, one of the ways of accessing objects in C++. See the section on [pointers](#) as an alternative method of using WPIlib objects.
- The `myRobot.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 motor safety timer is a bad idea! You should enable the motor safety timer, set its feeding interval, and supply values at least that often.

Inverting Motors

```
//Inverts rear or only motor on left side
chassis->SetInvertedMotor(RobotDrive::MotorType::kRearLeftMotor, true);
//If using four motors, invert front motor as well
chassis->SetInvertedMotor(RobotDrive::MotorType::kFrontLeftMotor, true);
```


Creating a robot project

Depending on the wiring and construction of your robot, it is possible that you will need to invert the direction of one or motors in your code in order to have all motors spinning the correct direction. If pushing the joystick directly away from you results in anything other than the robot driving forward, one or more motors needs to be inverted. If you have 2 motors in the Robot Drive, invert the side of the robot that moves in the wrong direction. Note that the Robot Drive object refers to the single motor in a 2 motor drive as the rear motor.

If you have 4 motors in your Robot Drive and one side drives the wrong way, invert both motors on that side. If you have 4 motors and one side of the drive appears to not move at all when commanded the motors may be fighting each other, try inverting one of the two motors and observing if that side of the drive now moves when commanded.