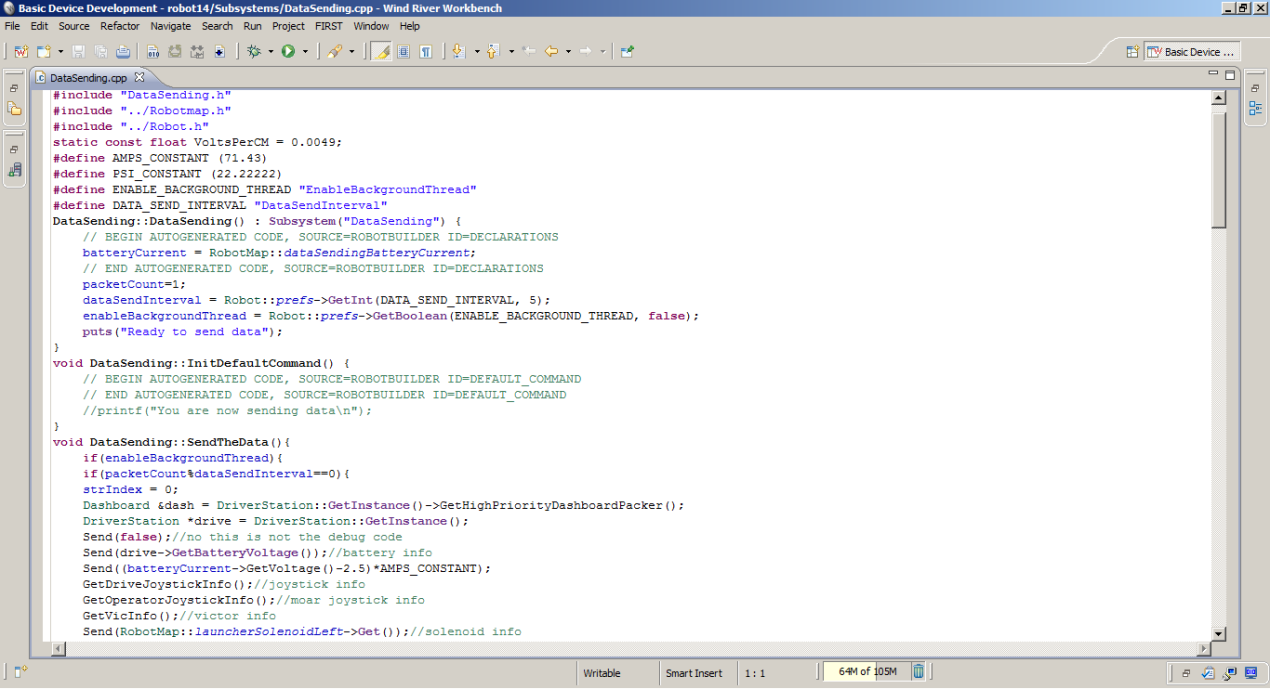
Debug and Diagnostic Code Documentation



Willard Wider - Team 1073

Software – Electrical – Integration

**Software Required to Compile (2014)**

In order to read, use and deploy the diagnostic code (for 2013-2014 build season), you need to download and install the following programs:

* WindRiver Workbench (and workbenchUpdate 20140325rev3887.zip) – Environment for developing C++ code for the Robot. Ask software subgroup at beginning of build season if there is a later workbenchUpdate zip)
* Visual Studio 2010 – The environment for making the DataCollection2014.exe program ---<http://www.microsoft.com/en-us/download/details.aspx?id=23691>--- you may need to ask Ken Cowan for the license to the application
* Git Client – The actual git binary ---<http://git-scm.com/>---
* TortoiseGit(optional) – A nice GUI for git ---<http://code.google.com/p/tortoisegit/wiki/Download>---
* GitHub application-only helpful for the git shell github.com
* Make sure you have a GitHub account
* FRC2014UpdateSuite.zip (Driver Station, NetConsole,etc) Other useful programs to Install/Run:
* (I) 2CAN Firmware Utility – Program, diagnose, and update firmware on the 2CAN
* (I) FRC Bridge Configuration Utility – An easy way to configure and program the radio
* (I) WireShark – An application used for listening to packets on the network.
* (R) BDC-Comm – Application to configure the Jaguars
* (R) Robotbuilder – A java jar file that makes a skeleton of the robot code, from a GUI.

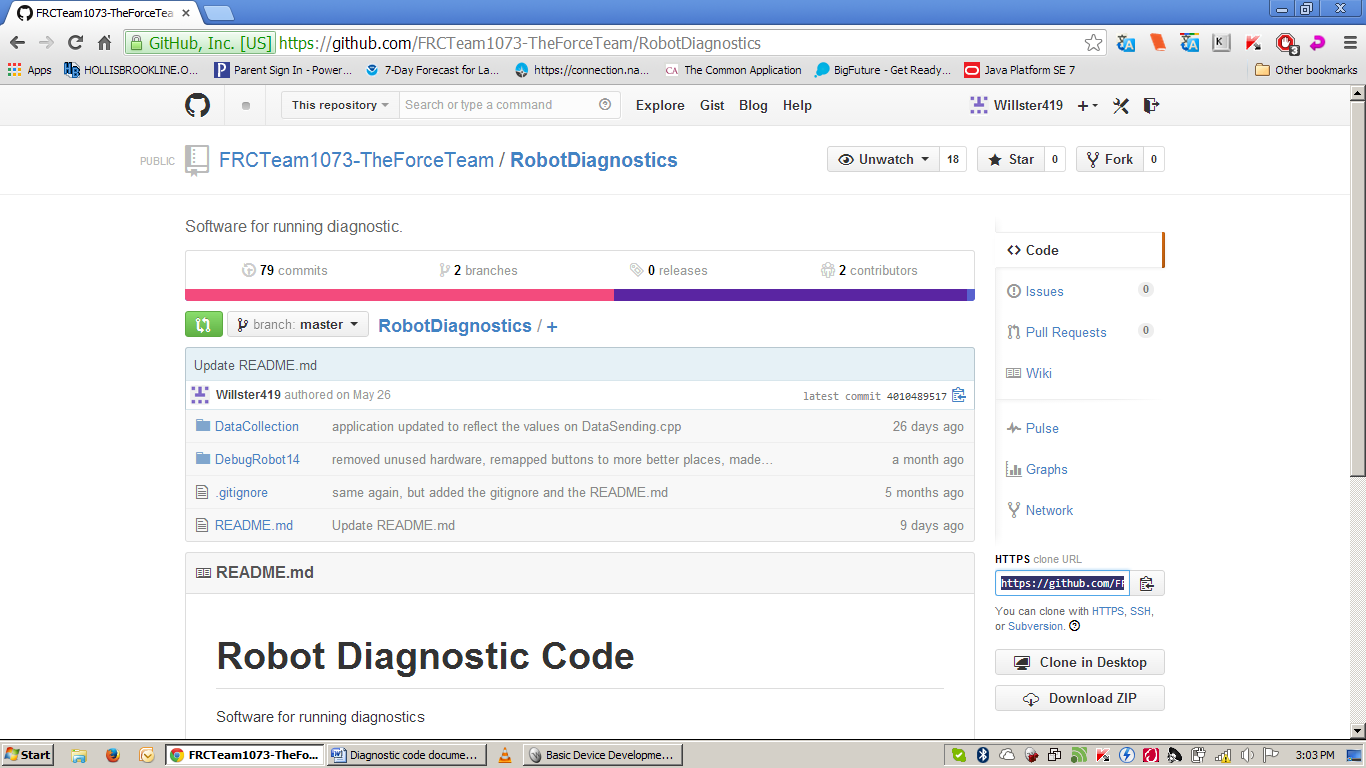
**Setting up your development environment**

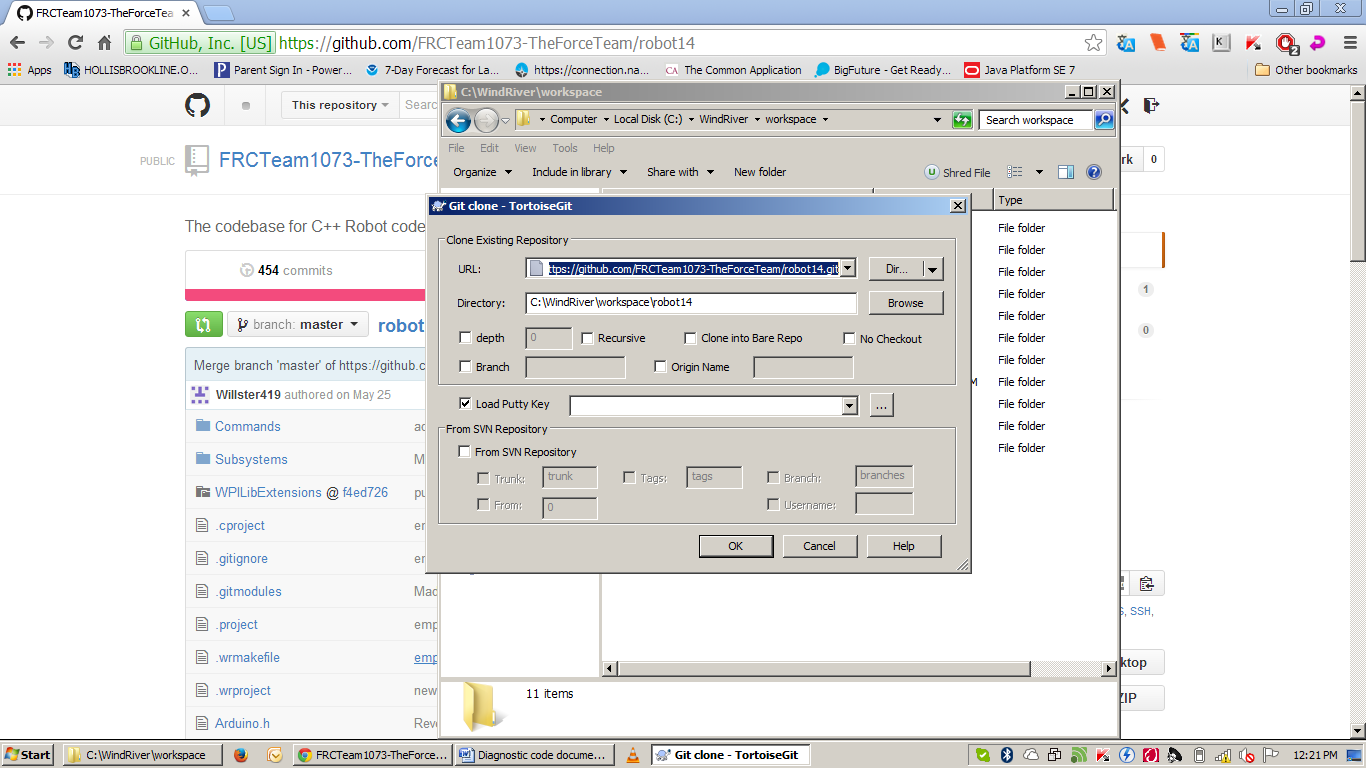
There are two applications of which you need to setup your development environment: WindRiver and Visual Studio

(The following assumes you installed everything)

WindRiver:

1. Open your web browser to github.com and find all of the software code for our team. A link for some of them are on the flash drive. Then copy the “clone url” to paste into tortoiseGit later.



1. Navigate into C:\WindRiver\workspace, and run git clone. If you use tortoiseGit, you can right click in the workspace folder, and select the “git clone” option. It will look like this:

If you already have the code folder there, and just need to update, then right click on the folder and select git sync, then the pull button. Of course, you should have committed/pushed you changes first. If you want to change branches, then use the switch/checkout menu.

1. Open WindRiver and select the default workspace (C:\Windriver\workspace)
2. Click on the file menu, and select the option import. Expand the general menu, and select the option, Existing projects into workspace. Click ok.
3. Click browse, make sure it says workspace in the folder text, and select ok.
4. Select the project you want to import and click finish.

Visual Studio:

1. Navigate to ..\workspace\RobotDiagnostics\DataCollection
2. Open DataCollection2014.sln

(Optional) You can also setup more git repositories from the team’s git site that are helpful if you want, like RobotData, previous data from the robot’s past, or Robot14, the actual code base for the robot. Some more helpful repositories are from my personal github account, 1073DataRecieverConsole, a simple console that receives data from the netConsole and driver station ports, 1073DataSimulator, a console that simulates the DataSending.cpp file, and 2013ElectricalCheckout, the first diagnostic code ever written.

**Understanding the Diagnostic Code**

The diagnostic code has two forms: A subsystem class that actually gathers all of the hardware information and sends it, and a completely standalone DebugRobot14 code base that uses basic code to test electrical components. A C++ source file has a .cpp file, and a .h file. The header files are the organizers of each cpp file. They contain all of the object and pointer references that are in the source code, both private (only to that class) and public (visible for all classes, therefore can be pointed to) Both were built with robotBuilder. The root directory has many folders and files, but we only focus on a few.

**OI.cpp** – The Operator interface. Joysticks and joystickButtons are created here. Each joystick gets 3 Axis (x,y,z) and buttons. Each joystickButton is constructed with a joystick object, the joystick that the button is on, and an integer, representing the number of the button on that joystick.

**RobotMap.cpp** – The mapping of the robot components. Each hardware device gets set here.

**Robot.cpp** – The main entry point of the code base. Here is where the methods correspond with the buttons of the driverStation of Teleoperated, Autonomous, and Test. There are two versions of each, a periodic version, and an Init version. The init method is run first, and only run once. The periodic method version is run while it is in each corresponding mode 50 times a second.

There are two folders of importance, Commands and Subsystems the commands in the Commands folder are the equivalent to runners, while the Subsystems in the Subsystems folder are like the classes with actual information and methods. RobotBuilder puts in comments about what each one is, and how to set it up, so there is no need for the redundancy. There are two main ways of making robot components function from what I have done in my work: while a button is held, do a command, or when a button is pressed, perform the action once, and exit. A while button is held example would be ..\Commands\angleDown.cpp. An example of when pressed, or a toggle, would be the old method for lauching a ball, ..\Subsystems\Laucher.cpp(commented out, if it is still there), and a toggle would be ..\Commands\compress.cpp

**Understanding the Data Collection**

The Data Collection application is made in many files, but we only deal with one. Here are some that you may encounter

**Program.cs** – The main entry point of the application. If we had more than one form, then this would be modified.

**Form1.cs** – The Class that contains all of the code for form1, known as DataCollection2014. It is the main file that we work with.

**Form1.resx** – The XML layout of the GUI of the form. Maintained by Visual Studio.

**Form1.Desigener.cs** – The visual representation of the form. Maintained by Visual Studio.

The entire form is completed in one class, in one namespace. There are several methods, many made by the user, some by the application. All private instance variables are listed in order by type at the top of the program. The program is made in 4 threads, a camera pinger, a data listener, a console listener, and the main UI thread. The main concept of the program is this:

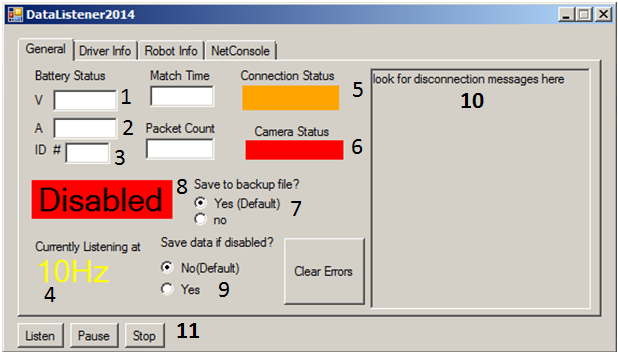
The pinger thread constantly pings the robot’s camera. If the robot’s camera is there, a panel on the form will be green. Else, it will be red.

The listener threads constantly listen on their specified ports for information. The listen method calls, are waiting calls, so the CPU is not overtaxed. When is has information, it is enqueued.

The main UI thread is run on timers to save CPU time. When there is data in each queue, it is parsed, displayed, and saved respectively. The dataTimer is the main timer, as it controls when data is shown on the screen. All text boxes are updated in the display methods, and all other UI components are updated in the UpdateUI method. Many of the methods have explaining comments.

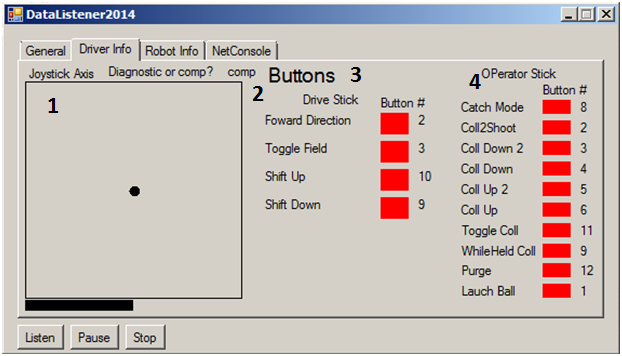
Layout of the Application

The “General” Tab



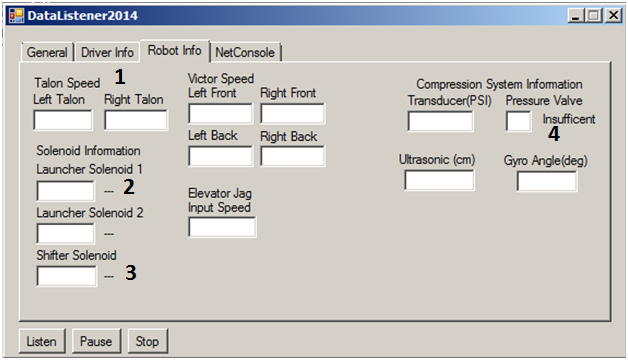
1. The Voltage box. Displays the voltage of the battery. Labeled batteryVolts.
2. The Amp box. Displays the amps of the battery. Amps is actually Columbs per second, if you ever take physics. Labeled in the code as batteryAmps
3. The battery ID. This was never used, because somebody \*coughElectrical\* never got around to RFID’ing the batteries. I leave this challenge up to you, and I don’t mean just the coding part. Labeled as batteryID. Do not click on the text, “ID”.
4. The rate, in Hertz, of the data being displayed. How this is determined is done in the code. It has to do with the number being send from the robot to the application. Labeled label87 in panel25.
5. The status of the connection. Green means the driver station is open and the netConsole has received data, orange if only 1 of the above is true, and red if they are both false. Done with the BackColor property, labeled panel1.
6. The status of the camera. Gets updated by a camera pinger thread. Red means no camera, green means camera. Labeled cameraStats, uses panel.backgroundColor property.
7. Yes - saves the data to a file. In the event of the program crashing, stopping unexpexctedly, etc. it is saved as “temp0.rtf/txt” if that file exists, then is saves as temp1, temp2, etc. No – Does not save. Use this for when you don’t need to collect data, only view information about the robot. A cool feature that you could make is a preferences file so it comes up with the last known configuration.
8. The big indicator for it the robot is enabled or not. Labeled label37, uses backgroundColor and text properties.
9. No – Only collects data if the robot en Enabled, in all modes, Teleop, Auto, pretice, etc. Yes – Collects in all enabled modes, and if it is disabled. Note: It will display data on the screen regardless of this setting.
10. Everytime the robot is not there, it will let you know with a message. Cleverly labled with the name disconnectionMessages. Pressing the clear errors button clears that text box.
11. Listen – Starts the listening and displaying process. If it is already listening, this button has no effect. Pause – pauses the listening, but keeps the data on-screen. Stop – Stops the listening, and clears all tabs of data. Labeled Listen, Pause, and Stop.

The “Driver Info” Tab



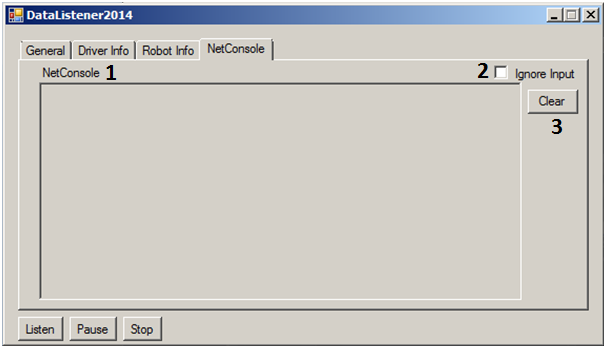
1. The Joystick Axis Display. The circle in the middle (point) moves around within the black square (border) to show the x and y axis of the joystick. The bar bellow (rotation) shows the z axis of the joystick.
2. Displays the “mode” of the application. Comp mode gives the UI above, versus diag mode gives the button mapping for the buttons on the diagnostic code’s joystick. The mode of the application is determined by the code on the robot.
3. The button layout. Determined by the application mode, the button layout is done by having a button name (it’s function), a panel who’s backgroundColor changes based on the status on the button, red or green. The number to the right is the button on the joystick that will run that command.
4. This is the same thing as 3, but I guess it’s showing the operator stick buttons. This whole column goes away if it’s in diag mode. Besides that, I really don’t need this bullet.

The “Robot Info” Tab



1. Talon/Victor/Jag Speed – the “speed” is the percent voltage you are letting out versus what you are letting in. if the input voltage is 10v, and you tell the motor controller to run at half speed, then 5v will be output and a “.5” will be put onto the speed on the application.
2. Launcher Solenoid(s) – single solenoids - they are set/gotten by a true and false, and are sent and show on here as a 1 or 0, and then translated to on or off
3. The shifter solenoid (double solenoid) is set by using the doubleSolenoid class, setting as forward, reverse, or off. They are gotten by setting them to 0,1,or 2. They correspond to being off, being in low, and being in high, respectively.
4. The pressure valve is a simple Boolean, either there is enough or not enough. When received, the text (label52) is set to either sufficient or insufficient.

The “NetConsole” tab



1. The netConsole output is displayed in the large read-only text box (netConsoleDisplay). If you click inside it, it will automatically scroll.
2. The check box (checkBox1) toggles displaying the information on the screen. It is still recorded. Useful for if you want to look in the long stream of output for a specific output.
3. The clear button clears the input on the netConsole. This does not clear the text from the saved file.

This concludes documentation. Best of luck when coding.

“*Good luck at the competition!”*