Programming Bible with TurtleShell

Sweeney

2016

Contents

1 Introduction **4**
- 1.1 Welcome to Programming! . . . 4
- 1.2 A Programming Lead . . . 4
- 1.3 Programmer Etiquette . . . 4
  - 1.3.1 Cookies . . . 4
  - 1.3.2 Turtwig and Turtles . . . 5

2 Physical Hardware **6**
- 2.1 A Note On The Separation of Programming and Electrical . . . 6
- 2.2 Motor Controllers . . . 6
  - 2.2.1 Victor 88x . . . 6
  - 2.2.2 Victor SP . . . 7
  - 2.2.3 Talon SR . . . 7
  - 2.2.4 Talon SRX . . . 7
  - 2.2.5 Jaguar . . . 7
  - 2.2.6 Note on Programmer Control . . . 7
- 2.3 Chassis/Drive Train . . . 7
- 2.4 Electrical Devices . . . 8
  - 2.4.1 Power Distribution Panel (PDP) . . . 8
  - 2.4.2 Pneumatic Control Module (PCM) . . . 8
  - 2.4.3 Voltage Regulator Module (VRM) . . . 8
- 2.5 Sensors . . . 8
  - 2.5.1 Distance . . . 8
  - 2.5.2 Rotation . . . 9
  - 2.5.3 Vision . . . 9
  - 2.5.4 Contact . . . 10
  - 2.5.5 Position . . . 10
- 2.6 roboRio . . . 11
  - 2.6.1 MXP (MyRio eXpansion Port) . . . 11
  - 2.6.2 Digital I/O . . . 11
  - 2.6.3 Analog Input . . . 11
  - 2.6.4 PWM . . . 11
  - 2.6.5 I^2C . . . 12
  - 2.6.6 SPI . . . 12

2.6.7 CANbus . . . 12
2.6.8 Ethernet . . . 13
2.6.9 USB . . . 13
2.7 Radio . . . 13
2.8 The Human Body Analogy . . . 13

3 Programming on the roboRIO **15**
3.1 roboRIO Basics . . . 15
3.1.1 Basic Specifications . . . 15
3.1.2 Connecting . . . 15
3.1.3 Updating roboRIO Programs . . . 16
3.1.4 The roboRIO Web Interface . . . 17
3.2 roboRIO Intermediate and Advanced . . . 18
3.2.1 Moving Beyond the Web Interface . . . 18
3.2.2 Processing Capabilities and Limitations . . . 18
3.3 Languages . . . 19
3.3.1 Java . . . 19
3.3.2 C++ . . . 19
3.3.3 LabVIEW . . . 20
3.3.4 Direct Comparisons . . . 20
3.4 Basic Java . . . 21
3.4.1 Resources for Learning Java . . . 21
3.5 Simple Java on the roboRIO . . . 21
3.5.1 Sample, Command-Based, and Iterative Robot . . . 21
3.6 Using Eclipse with WPILib . . . 22
3.6.1 Installation . . . 22
3.6.2 Creating a Project . . . 22
3.6.3 Creating code . . . 23
3.6.4 Importing a project . . . 23
3.7 Switching to IntelliJ . . . 24
3.7.1 Importing from Eclipse . . . 24
3.7.2 Creating in IntelliJ . . . 25
3.8 Useful Classes and Methods in WPILib . . . 25

4 TurtleShell **26**
4.1 Design Philosophy . . . 26
4.2 Basic Functionality . . . 26
4.2.1 Program Flow . . . 26
4.2.2 Key Interfaces . . . 27
4.2.3 Key Implementations . . . 28
4.2.4 Utility . . . 29
4.3 Advanced Features . . . 30
4.3.1 PID . . . 30
4.3.2 Vision Processing . . . 31
4.4 Code Style . . . 31
4.4.1 Naming . . . 32

4.4.2 Comments 32
4.4.3 Language 32
4.4.4 Indenting, Spacing, and Braces 32

Chapter 1

Introduction

Hello world.

1.1 Welcome to Programming!

Welcome, and congratulations on your decision to learn more about programming! Programming is the best group within a robotics team, and you should be proud to be a member.

1.2 A Programming Lead

The Programming Lead is the leader of programming team. They are ultimately responsible for ensuring that the code works and fulfils its purpose.

The Responsibilities of the Programming Lead

- Design and implement the robot code.
- Teach younger programmers and pass on the knowledge.
- Work with drive team to design and implement a control system.

1.3 Programmer Etiquette

This section covers the standard programmer and behaviour at Team 1458, which may or may not be applicable to other teams.

1.3.1 Cookies

Programmers have a well-known sweet tooth, in particular for cookies. The best type of cookies are chocolate chip cookies, with the subtype of chocolate chip with M&Ms being the only improvement. "Cookies" with nuts, raisins,

or oatmeal are not real cookies, they should be safely disposed of in the trash. "Cookies" containing meat should be disposed of following standards, it should be treated as a health hazard if ingested. Programmers are known to consume copious amounts of cookies in one session, particularly during build season.

1.3.2 Turtwig and Turtles

Turtwig, a Pokémon with the National Pokédex Number 387, is the cutest and the best Pokémon. Turtwig is a turtle, and by association turtles are loved and appreciated as well. Numerous Turtwigs are brought in during build season, and a Turtwig should be present during every major test.

Chapter 2

Physical Hardware

This section covers the physical devices associated with the robot that must be understood in order to properly program the robot. A basic familiarity with all parts of the robot is required, but certain parts, namely those associated with the control system must be understood in greater depth. This section will dedicate time to each class of components in rough proportion to their importance to programming.

2.1 A Note On The Separation of Programming and Electrical

The boundary between programming and electrical, especially with sensors, can be unclear. The generally accepted split is that electrical is in charge (no pun intended) of the hardware and wiring, while programming is responsible for the control logic. However, programming is often given control over some aspects of hardware for the placement of sensors, although it is still electrical's responsibility to wire the sensors.

2.2 Motor Controllers

Motor controllers are what regulate the motors. They have a connection to the Power Distribution Panel (PDP), the motors themselves, and the roboRio through a PWM (or Pulse Width Modulation) control. They take the PWM signal and use it to control the voltage going to the motor, and thus control the motor.

2.2.1 Victor 88x

Victor 884 and 888s are the simplest type of motor controller, and overall they are effective. The differences between the two are minor, with some changes

to the PWM curve. They are the second-largest motor controller, with the required fan taking up much of the space. They have been made obsolete by the introduction of the Victor SP, which is strictly better.

2.2.2 Victor SP

The Victor SP is VEX's new motor controller, and features several improvements over the previous models, including a much smaller form factor and an integrated heatsink which leaves a fan optional. Use-wise, it functions similar to the previous models with a PWM input.

2.2.3 Talon SR

The Talon SR is very similar to the Victor SP, although it has a larger footprint more comparable to the Victor 88x models. It has been deprecated in favour of the Talon SRX, but the Victor SP is a more direct successor in function.

2.2.4 Talon SRX

The Talon SRX is a much more advanced motor controller, comparable to a Jaguar in power. It can connect over CAN or PWM, and features functions that make it easier to set up PID control. However, this does lead to a more expensive motor controller and reduced programmer control over functionality.

2.2.5 Jaguar

Jaguars are the most advanced motor controller, but they suffer heavily for their features, which include connectiblity over CAN, PWM, and Ethernet. Jaguars are far larger than other motor controllers, in footprint, height, and cost. They suffer an even larger lack of programmer control, with them refusing to work if they are at risk of overheating.

2.2.6 Note on Programmer Control

Best practices developed at Team 1458 have been to do the majority of the coding manually. There have been issues with documentation and errors in WPILib, leading to the restriction of its use to the component interactions. Not only does coding it ourselves allow us to avoid these pitfalls, but allows programmers to get additional experience and learn more. Thus, WPILib is only used in TurtleShell for motor & sensor interfaces, no higher-order logic is taken from it.

2.3 Chassis/Drive Train

The chassis/drivetrain refers to the physical structure of the robot. This is the domain of mechanical, and programmers simply need to understand how the

motor movements correspond to actions in the physical world. More information on drive systems can be found in Part 3.

2.4 Electrical Devices

This section covers the Power Distribution Panel (PDP), Pneumatic Control Module (PCM), and the Voltage Regulator Module (VRM). These devices are usually not used directly by programming, but understanding their function is necessary.

2.4.1 Power Distribution Panel (PDP)

The PDP is what provides power to most of the robot. It has a connection to the battery through the circuit breaker, which serves as the on/off switch on the robot. It then connects through circuit breakers to all devices on the robot that require power. It is equipped with CAN in order to monitor the connections and their current draw.

2.4.2 Pneumatic Control Module (PCM)

The PCM is connected over CAN to the roboRIO. All controls for pneumatic devices go through here. It provides power for the compressor and regulates it, along with controlling any solenoids.

2.4.3 Voltage Regulator Module (VRM)

The VRM is connected to the PDP, and provides lower-voltage power. It provides power to the router on board, as well as other devices that don't require high amperages (so not most motors), such as LED lights.

2.5 Sensors

This section discusses a variety of types of different sensors available on the robot, and their possible uses and drawbacks. It doesn't cover their coding or integration into software. This section is divided based on utility, however some sensors may fit under multiple categories.

2.5.1 Distance

Distance sensors determine how far away something is from the sensor. It can be useful in aligning the robot or in preventing collisions.

Infrared

Infrared sensors detect how far away something is from the sensor based on the timing of pulses of infrared light. They are able to detect small objects and work in enclosed spaces, but are more expensive than ultrasonic sensors.

Ultrasonic

Ultrasonic sensors emit sounds above the range of human hearing and count the time it takes the sound to return. They are cheap, however they are bad at detecting small or curved objects, and are subject to interference from objects in the way.

Laser

Laser distance sensors are extremely accurate, quite often on the millimetre level, and do not suffer from as much interference as other sensors. However, they are the most expensive and fragile, qualities often unsuitable for the robot.

2.5.2 Rotation

Gyroscope

Gyroscopes are the best known rotation sensors. They are actually unable to measure the way the robot is facing, instead it measures the change and uses that to determine the direction relative to the start. They are widely available, but they have a problem in accuracy called yaw drift, where the angle will shift by 2-3 degrees per minute, even while stationary, for the best gyros, and larger drift for lower quality.

Magnetometer

Magnetometers measure the strength of magnetic fields on certain axes. In the context of the FRC, they are used to measure the Earth's magnetic field and thus act as a compass, providing a reference point for rotation. While they don't have the same problem with drift, they are much more difficult to calibrate, with the calibration changing based on where the robot is located and what ambient magnetic fields exist. This can prove a problem, as the robot's motors generate magnetic fields which can interfere with proper function. They are also much more difficult to program.

2.5.3 Vision

These sensors work with vision, which can be used for a variety of tasks depending on the year.

Camera

Cameras are really the only device that can be used for vision, the lights are only there to assist the camera. The camera can see the retroreflective targets that are often visible during the autonomous period, and so the robot can take action based on that. Coding the camera is a difficult task, several libraries such as GRIP and NIVision are able to work with it.

Light

The lights are to illuminate the retroreflective target so the camera can identify it. They are usually green, as green stands out the most from other colours present.

2.5.4 Contact

Pushbutton

Pushbutton switches consist of a button that makes (or sometimes breaks) an electrical connection. They can be useful as bumpers to know if the robot has made contact.

Limit Switch

Limit switches have some sort of object that when it is pressed down, it triggers. They are electronically identical to pushbuttons, but they have a switch rather than a button. They are useful in safety for moving devices, to ensure they don't go too far and damage something.

2.5.5 Position

Accelerometer

Accelerometers don't actually measure position, they measure acceleration. It's technically possible to derive[1] position from acceleration, but it is a difficult process. Accelerometers can still be used in autonomous for positioning, but aren't very useful beyond that. In years where the robot is tilting, it can be used to determine whether or not the robot is flat.

Rotary Encoder

Rotary encoders (usually just called encoders) are one of the most useful sensors. They record rotations of a shaft, which can be used to do everything from measuring the distance the robot has moved forward to the rotation of an appendage. They work by causing an electrical pulse whenever the shaft moves a certain amount (Such as 1°). The roboRio counts these pulses and uses that to determine how far the shaft has rotated. In order so that to roboRio can

[1] Well, integrate.

determine which direction the shaft is rotating, there are two channels that are slightly offset, so the roboRio can tell the direction by which triggers first. This means that it requires two digital inputs.

2.6 roboRio

The roboRio is the brain of the robot. The code runs on the roboRio, which runs a modified version of Linux. The roboRio is one of the few "smart" devices on the robot, with most of the remaining devices serving to interface between it and the physical world. It has digital, analog, I^2C, SPI, USB, CAN and Ethernet I/O (Input/Output), along with PWM output and the MXP.[2] By the FRC rules, all control of motors has be done through the roboRio.

2.6.1 MXP (MyRio eXpansion Port)

The MXP is the extra set of pins in the middle of the roboRio. It is meant for custom devices, although a few commercial ones are available. It has a massive amount of possible inputs, about doubling the possible amounts of each input and output. Its use is largely unnecessary unless a large amount of inputs or outputs are needed.

2.6.2 Digital I/O

The digital input and output ports use the same cables as the PWM inputs. The red and black are to power the sensors (with red power and black ground), while the white carries the signal in a digital fashion. Digital is where there are only two values, on and off. Sensors that use this include limit switches and rotary encoders.

2.6.3 Analog Input

Analog Inputs are similar to the digital inputs, with the same cables and colours for power, ground, and signal. Analog is where the voltage varies, so there is a continuous range of values. Analog inputs power different kinds of sensors, with some gyroscopes and accelerometers, and most infrared and ultrasonic sensors relying on an analog input. The voltage from these must be interpreted to get meaningful results.

2.6.4 PWM

PWM (Pulse Width Modulation) is how the roboRio communicates with the motor controllers. PWM is a system that changes the width of pulses, usually in the range of fractions of milliseconds, to alter motor power or servo position.

[2] Don't worry, all of those acronyms will be explained shortly

It uses the same cables as analog and digital inputs. The main use of PWM outputs is to control motors.

2.6.5 I^2C

I^2C(Pronounced Eye-two-see) is another communication protocol. It is very advanced, and is capable of exchanging large amounts of information and connecting multiple devices, but is very complicated (An interviewed company contracted it out rather than work with it). Luckily, some of the very low-level communication is done with the libraries, so the programmer only has to work with the bytes that are being transferred, rather than the process of transferring them.

2.6.6 SPI

SPI is another complicated communications protocol, one which is not commonly used by Team 1458. Wikipedia states:

> The Serial Peripheral Interface (SPI) bus is a synchronous serial communication interface specification used for short distance communication, primarily in embedded systems. The interface was developed by Motorola and has become a de facto standard. Typical applications include sensors, Secure Digital cards, and liquid crystal displays. SPI devices communicate in full duplex mode using a master-slave architecture with a single master. The master device originates the frame for reading and writing. Multiple slave devices are supported through selection with individual slave select (SS) lines. Sometimes SPI is called a four-wire serial bus, contrasting with three-, two-, and one-wire serial buses. The SPI may be accurately described as a synchronous serial interface, but it is different from the Synchronous Serial Interface (SSI) protocol, which is also a four-wire synchronous serial communication protocol, but employs differential signalling and provides only a single simplex communication channel.

2.6.7 CANbus

The CAN (controller area network) bus is another way of interfacing with devices. It is commonly used in industry, and it is found in modern cars. It involves the high, yellow wire and the low, green wire. They can be used for Jaguars, and other advanced electronics. One key aspect is their use in of pneumatics and the PDP, CAN is the only way to connect to those. All workings with the CAN system are hidden in WPILib, a reasonable approach given its complexity. The PDP and respective pneumatic classes can be used to interact with those systems.

2.6.8 Ethernet

There is an Ethernet port on the roboRIO. Ethernet is a standard for communication, often used as a wired way to obtain internet access. For the robot, the Ethernet port is used to connect to the router that provides wireless connectivity. Cat 5e or above is recommended, as well remember to make sure there are not significant kinks in the cable, it degrades performance.

2.6.9 USB

There are three USB ports on the roboRIO. Two are USB Type-A, the standard USB port. Devices such as cameras should be connected here, as well as any sensors that work over USB.
The USB Type-B[3] port is useful for connecting to the computer, use it instead of Ethernet for direct connections to the roboRIO.

2.7 Radio

The radio is used on the robot to communicate with the FMS and Driver Station. It is a router running specialised software. The model is OM5P-AN. It has a power plug which connects to the VRM, as well as two Ethernet ports, one for connection to the roboRIO and one for connection to the computer. The Radio Configuration Utility, included with LabView, will install and flash the firmware. Note that the configuration for home and competition use are different.

2.8 The Human Body Analogy

For many people, an analogy comparing the parts of the robot to parts of the human body is useful for understanding the way the robot works and how things interact. Henceforth, let the analogy commence:

Skeletal System

The frame and mechanical portions of the robot.

Muscular System

Motors and solenoids.

Circulatory System

Components carrying electricity for the purpose of powering components (not signals).

[3]The cable with the house-shaped hexagon other end, often used for printers

Heart The PDP.

Arteries Red cables, where power flows out.

Veins Black cables, where power flows back.

Lungs The battery, the source of all electrical power on the robot.

Nervous System

"Smart" components as well as signal-carrying wires.

Axons Signal-carrying wires, most colours but red and black, especially white, yellow, and green.

Nerve Endings Sensors.

Brain The roboRIO, other computing devices would be secondary brains.

Chapter 3

Programming on the roboRIO

This chapter will cover the roboRIO, languages running on it, the Java language, and some basic Java code on the robot. This chapter will not cover TurtleShell, and some things introduced in 3.5 will be overridden in 4.

3.1 roboRIO Basics

This section covers basic roboRIO functionality from a programming perspective.

3.1.1 Basic Specifications

The roboRIO is a proper computer. It is produced by National Instruments, based on the Xilinix© Zynq^{TM}-7020 platform. It has as its processor an ARM Cortex^{TM}-A9, with 256MB of RAM and 512MB of flash storage. It runs a real-time variant of Linux, and supports LabVIEW, C++, and Java. This may be a restatement of the fact sheet.

3.1.2 Connecting

It is possible to connect to the roboRIO over both Ethernet and USB. Either is acceptable, Ethernet should have a higher throughput but this is often not the case. Whichever is more accessible (USB Type-B or Ethernet) is likely simpler to use. The standard mDNS address is "roboRIO-####-FRC.local", where #### is the team number. (No leading zeroes)

mDNS

The mDNS protocol is used for robot connection, reducing the need for static IP and other network setups. It allows the use of host names rather than IP addresses on small networks without the need for a nameserver (such as Google's 8.8.8.8 and 8.8.4.4). In order for computers to make use of it, they require external software, Apple's Bonjour works on Windows and Mac, and LabView comes with its own version. Linux mDNS software also exists.

Browser Use

For some ungodly reason, NI decided to use Microsoft Silverlight to create the web interface for the roboRIO. This restricts browser use to browsers which still allow that mess of a plugin, and NI happily recommends IE. Team 1458 uses Firefox for this purpose.

USB

The driver for the roboRIO should automatically install. If not, it should install with LabView. The mDNS address should work, alternatively the USB IP is "172.22.11.2". Again note the USB port that should be used is the USB Type B port, often found on printers. It is a house-looking hexagon.

Ethernet

The mDNS address can be used over Ethernet as well. The radio uses the IP address 10.TE.AM.1 (Where TE.AM is the zero-padded team number), and gives other devices IP addresses in the range 10.TE.AM.10 to 10.TE.AM.99 through DHCP.

3.1.3 Updating roboRIO Programs

The roboRIO's firmware and image may need to be updated. The firmware rarely needs to be updated, unless the device is being wiped, but the image is usually updated once per year.

Updating the Firmware

1. Connect to the roboRIO using USB, **not Ethernet**.
2. Connect to the web interface and log in with the Username "admin" and a blank password.
3. Click the update firmware button, and navigate to the location where the firmware is stored (`C:\Program Files (x86)\National Instruments\Shared\Firmware\cRIO\76F2` on Windows 64-bit), and select the most recent firmware (should be the numerically largest version).
4. Click the begin update button and let it complete the update.

Updating the Image

1. Connect to the roboRIO using USB, **not Ethernet**.
2. Launch the imaging tool, located at `C:\ProgramFiles(x86)\NationalInstruments\LabVIEW2015\project\roboRIOTool\roboRIO_ImagingTool.exe` for 64-bit Windows.
3. Scan for roboRIOs and select the appropriate one.
4. Enter your Team Number in the box.
5. Check the box Console Out under Startup Settings and **uncheck** the box Disable RT Startup App.
6. Check the Format Target box and select the most recent image.
7. Click Reformat.
8. After the reformatting, press the reset button on the roboRIO to complete the process.

Installing Java on the roboRIO

This step is only necessary to run Java programs.

1. Java 8 must be installed on the computer used to connect to the roboRIO.
2. Connect to the roboRIO using USB, **not Ethernet**.
3. Open the FRC Java Installer, located at `~/wpilib/tools/java-installer.jar`.
4. Follow the listed instructions in the Java Installer.
5. Java 8 is now installed on the roboRIO. This will be needed to be repeated each time the roboRIO is imaged.

3.1.4 The roboRIO Web Interface

The web interface allows use and customisation of many features on the roboRIO. Many of these features require logging in, the default login is "admin" with no password. However, a password must be set to access the file browser. It is a good idea to create a second account with the same permissions and a password so that the file browser can be used, and the admin account is still available with the default settings. For changing files in the lvuser account, the lvuser account must be logged in to, it has the password "password". Other than the file browser, the main configuration tool is the permission manager. Other than creating new accounts, the use and function is relatively complex and unimportant. Although there are a variety of tabs, most of them are not important. However, the main tab has the ability to update the firmware for the roboRIO, PCM, and PDP.

3.2 roboRIO Intermediate and Advanced

This section covers more advanced topics with the roboRIO such as the command line and processing capabilities.

3.2.1 Moving Beyond the Web Interface

While the web interface is easy to use, it is quite clunky and is often less intuitive than other options. Most of these are command line based, such as scp and and ssh, and are from Unix.

ssh

One of the most powerful tools is ssh, the remote shell program. It allows logins over a network connection, giving full access to the command line. PuTTY is a Windows program available that can fulfil this function. The command line is a fairly standard Linux environment, but one important difference is the lack of a sudo or su command. Thus, in order to modify the code or output files, the "lvuser" account usually must be logged into.

scp

Another tool available is scp, which stands for secure copy. It allows the copying of files to and from the roboRIO, the ANT script that compiles the code uses scp to move the code onto the roboRIO. As a standard Unix command, documentation is available elsewhere.

3.2.2 Processing Capabilities and Limitations

The roboRIO is a moderately powerful computer. It is less powerful than a smartphone, but is comparable to portable game systems like the Nintendo 3DS. This is relevant from a programmer's perspective because it limits what a programmer can do. For example, if several million floating point calculations need to be done in order to execute a single motor movement, the code will be far too inefficient to be effective. However, eking every last bit of efficiency out of code requires destroying any pretence of readability or maintainability. For the roboRIO, and for modern programming on non-embedded systems in general, the balance is usually to simply neither do wildly wasteful things nor condense the code. However, this balance shifts in locations like loops. There, since the code must be repeatedly, it is usually beneficial to optimise the code at the expense of ease of use. For example, declaring a variable that would normally be declared inside a loop outside instead adds in an extra variable to work with, but it stops that variable from having to be allocated every time that code is run, and as such saves processing time. A key field where this is an issue is vision code. With hundreds of thousands of pixels, a slight mistake can greatly increase the amount of time required.

However, even optimally designed vision programs are quite possibly too much for the roboRIO. Thus, other computers can be used on the robot, such as Nvidia's Jetson TK1, to supplement it and offload vision processing, not only speeding up the processing but ensuring that the vision processing cannot hog time from more important tasks.

3.3 Languages

Many languages are supported on the roboRIO. The officially supported languages are Java, C++, and LabVIEW, although unofficial libraries for other languages such as Python are available. This guide primarily covers Java, although this section will attempt to provide a brief overview of the languages and aid in the selection of the best language for the team.

3.3.1 Java

Java is a C-like language developed at Sun Microsystems and currently maintained by Oracle. Java's main feature is its cross-platform support, available because of the JVM, or Java Virtual Machine, which runs the code, rather than the operating system itself. This additional layer of abstraction comes with a cost however, Java code is almost always less efficient than code written in C or C++. Java is quite often used for commercial applications, the cross platform nature reduces development costs and the relative ease with which it can be used make it quite popular as a server-side language, although it is being supplanted by Ruby and Python in many of these applications. As well, Java is a part of the Advanced Placement Computer Science A course, which many students enrolled in FIRST may take, and can serve as a breeding ground for programmers.

On the robot, virtually all Java features are available, and the additional overhead is only an issue in vision processing. Plenty of libraries and sample code are available for Java on the roboRIO.

3.3.2 C++

C++ is an industry-standard language that has stood the test of time. It is an expansion on the earlier C language, adding object-oriented programming. Most native programs are written with C++ or similar, likely including the program being used to view this file. C++ can be difficult to work with because of its use of pointers, but the difficulty increase compared to Java is relatively minor. C++ and other lower-level languages run more quickly than languages like Java or Python.

C++ is fully supported on the roboRIO and is a fine choice for robot development. Plenty of libraries and sample code are available for C++ on the roboRIO.

3.3.3 LabVIEW

LabVIEW is National Instruments's graphical programming language. It is primarily used for working with NI's products in robotics and research. The language is visual, rather than text, based, and is designed with blocks to represent operation and wires for data flow. The language suffers from extremely long compile times, often stretching into multiple minutes, and poorer performance than Java. It also suffers from limited applicability, NI's products are the only use of the language and while the visual aspect may be appealing to beginners, it is drastically unlike other programming languages and so skills are often not transferable. It can also be very difficult to debug, and the tangled mess the wires form is infamous.

LabVIEW is heavily optimised for the roboRIO, and includes utilities to ease development. NI provides support and software for the product.

3.3.4 Direct Comparisons

Overall, C++ and Java are relatively equal. Although C++ is more difficult to use, it makes up for this in performance. However, Java is more often used in server-based technology, and as such can be more useful in the career field. Java is also commonly taught with the Advanced Placement Computer Science A class, which can serve as a valuable source of programming experience. The choice between these languages should depend largely on the particular team, and the experience they have. While both languages have minor advantages over the other, mentor, student, school or sponsor experience can override this advantage and should be the deciding factor.

LabVIEW is overall inferior for robot development. It has limited applicability outside of FRC and lacks the massive repository of libraries that other languages have. Much of the skills and code are not transferable, making the learning wasted. It is also more difficult to develop and debug.

	Java	C++	LabVIEW
Performance	3	5	1
Power	5	5	3
Support	4	4	3
Learning Curve	3	1	2
Ease of Use	4	3	2
Career Utility	4	3	2
Overall Recommendation	5	4	2

Table 3.1: Side by side comparison of languages on a 1-5 scale, with 1 being the worst and 5 being the best

3.4 Basic Java

This section covers the basics of the Java programming language which are necessary for roboRIO programming. These basics are approximately covered by the AP Computer Science A (AP Java) class, those enrolled in the class at the same time as their participation in programming should find they have the knowledge necessary.

3.4.1 Resources for Learning Java

A full introduction to Java is outside the scope of this guide, it will only be able to serve as a quick reference. Resources for learning Java include:

- AP Java Barron's Book (`http://www.amazon.com/Barrons-AP-Computer-Science-7th/dp/1438005946`)
- Oracle (`https://docs.oracle.com/javase/tutorial/java/`)
- Codecademy (`https://www.codecademy.com/learn/learn-java`)

3.5 Simple Java on the roboRIO

This section examines basic programming using Java on the roboRIO. Some information introduced here will be overridden in 4.

3.5.1 Sample, Command-Based, and Iterative Robot

In order for the robot code to run, a class in the package `org.usfirst.frc.team####.robot.Robot.java` must exist (Where #### is the team number) and extend `RobotBase` in some fashion. To do this, WPILib provides `SampleRobot.java`, `CommandBasedRobot.java`, and `IterativeRobot.java`, and most robot code should extend one of these three classes. Each of them has particular advantages and disadvantages.

Sample Robot

Sample Robot, previously known as Simple Robot, is quite simple, and allows the most control over program flow. It features a method that is called **once** at the beginning of each period. For example, the method that all teleoperated is contained in is:

```
public void teleOperated() { ... }
```

Of note is that a loop must be contained within this method in order to get the teleoperated code to run more than once. Sample Robot is easy to use for both very simple and very complex programs, but suffers for those of intermediate complexity.

Command-Based Robot

Command Based Robot is a very different paradigm than others. It focuses on building complex functionality out of simpler functionality through "commands". Team 1458 does not have any experience with it and as such cannot offer any advice.

Iterative Robot

Iterative Robot is similar to Sample Robot, but it externalises the loop. Instead of a single method for each mode, an init method and a loop method are provided, with the init method called to initialise and the loop being called about every 20 ms. It is a more efficient class than Sample Robot, because the Driver Station only sends information every 20 ms, and so it doesn't waste processing time. However, it does restrict flexibility in design, and can require global variables.

3.6 Using Eclipse with WPILib

This section covers the use of the Eclipse IDE. Eclipse is one of the more popular IDEs, and is a FOSS project.

3.6.1 Installation

The installation instructions are available online at `https://wpilib.screenstepslive.com/s/4485/m/13503/l/145002-installing-eclipse-c-java`.

3.6.2 Creating a Project

1. Press the "New" Button.
2. Scroll down to "WPILib Robot Java Development".
3. Click the arrow to drop down the options below.
4. Select "Robot Java Project".
5. Click "Next".
6. Enter a project name.
7. Select a project type (See 3.5.1 for choosing a Project Type).
8. Press "Finish".

3.6.3 Creating code

To create a class, do the following:

1. Right click on the project in the Package Explorer.
2. Navigate to "New".
3. Click "Class".

3.6.4 Importing a project

1. Right click in the Package Explorer.
2. Select Import.
3. Under "General", select "Existing Projects into Workspace".
4. Click "Next".
5. Click on "Browse" next to "Select root directory".
6. Navigate to the directory containing the root directory of the project.
7. Click "Open".
8. Click "Finish".

   The project is now imported, but for it to work properly, several additional steps must be taken.
9. Right click on the project in the Package Explorer.
10. Click on the "Properties" item.
11. Click on the "Java Build Path" item.
12. Click on the "Libraries" tab.
13. Click on the "Add External JARs..." button.
14. Navigate to `~/wpilib/java/current/java/lib`.[1]
15. Select all four items and click "Open".
16. Click "Apply".
17. Click on the "Source" tab.
18. Select all items in the view.
19. Click the "Remove" button.

[1] `~` refers to the home directory, located at `/Users/username/` on Mac, `C:\Users\username` on Windows, and `/home/username` on UNIX.

20. Click on the "Add Folder..." button.
21. Select the "src" folder.
22. Click the "OK" button.
23. Click the "Apply" button.
24. Click the "OK" button.

3.7 Switching to IntelliJ

After working with Eclipse, you may become tired with its idiosyncrasies, such as an inability to handle basic importing and a need to manage imports. IntelliJ, created by JetBrains, has acheived large adoption in commercial Java programming. This section covers using IntelliJ rather than Eclipse for development.

3.7.1 Importing from Eclipse

If there is a project already created in Eclipse, it can be imported into IntelliJ.

1. Open IntelliJ.
2. Import the Eclipse project.
3. Continue to import it as an .idea project.
4. When it is imported as an IntelliJ module, go to View > Tool Windows > Ant Build.
5. Press the "+" button.
6. Navigate to the build.xml file and select it.
7. On the top bar, click Edit Configurations.
8. Create a new Ant Target.
9. Set a descriptive name.
10. With the bar directly below the name, set the Ant target to "deploy".
11. Remove "Make" from the "Before Launch" items.
12. Click "OK".
13. Go to "Module Settings" for the imported module.
14. Go to "Dependencies".
15. For "wpilib" and "networktables", IntelliJ cannot handle the Ant references for their locations.
16. Therefore, edit them and set the "Classes" and "Sources" to their location on disk (`~/wpilib/java/current/lib`).

3.7.2 Creating in IntelliJ

This is currently not possible.

3.8 Useful Classes and Methods in WPILib

This section covers classes and methods that are needed in WPILib.[2]

[2]This is not a joke saying that WPILib is useless, this section is simply not yet written.

Chapter 4

TurtleShell

TurtleShell is the framework designed by Team 1458 to simplify best practices in robot coding as well as address issues within WPILib. It is available at `https://github.com/FRC1458/turtleshell/`.

4.1 Design Philosophy

The most important idea of TurtleShell is that of modularity, and towards that, interfaces are used throughout. The advantage of using modular components are that the code is more resuseable, and it can easily adapt to changes, such as when the motor controller is changed, it can be fixed in the code with a single line.

4.2 Basic Functionality

This section covers basic functionality of TurtleShell, and should allow a reader to successfully create basic robot code.

4.2.1 Program Flow

Similarly to WPILib, a `Robot.class` must exist, and it still extends `RobotBase` indirectly. However, instead of defining its own components, the `ObjectHolder` interface is used to define and hold all robot components. The `Robot` class calls the `teleUpdateAll()` method on the `ObjectHolder` in order to `teleUpdate()` all of the components.

The `ObjectHolder` can either be entirely custom written, or can extend `SampleRobotObjectHolder`, which defines an `ArrayList` to store the components and iterates over each element in the list and calls update on them when necessary. In the constructor of the `ObjectHolder`, it should declare all of the `RobotComponents` and pass in all data to properly initialise them.

The body of `Robot.class` should initialise an `ObjectHolder`, and in the `teleOperated()` method it should repeatedly call `teleUpdateAll()` on it.

4.2.2 Key Interfaces

This section covers key interfaces that must be understood to properly make use of TurtleShell.

RobotComponent

All components on the robot, from the chassis to an arm, implement this interface. It provides the `teleUpdate()` and `autoUpdate()` methods. These should all be contained in the `ObjectHolder`.

ObjectHolder

This interface is for containers for `RobotComponent`s and similar. It contains methods to call the `teleUpdate()` and `autoUpdate()` methods for its `RobotComponent`s.

Movement

TurtleMotor An interface representing a motor that is given a `MotorValue` to move, representing full forward to full backwards.

TurtleServo An interface representing servos, which are given an angle and move to it.

TurtleSolenoid An interface representing a solenoid, which is either extended or retracted.

Sensor

TurtleButtonSensor An interface representing a sensor that is either pressed or not.

TurtleRotationSensor An interface representing a sensor that measures rotation around an axis.

TurtleDistanceSensor An interface representing a sensor that measures distance travelled by something.

Input

TurtleDigitalInput `TurtleDigitalInput` represents an input which has discrete values.

TurtleAnalogInput `TurtleAnalogInput` represents an input which has continuous values.

4.2.3 Key Implementations

This section explains provided implementations for the above implementations and how they can be used.

SampleRobotObjectHolder

`SampleRobotObjectHolder` implements `ObjectHolder`, it is abstract and defines an `ArrayList` to store `RobotComponent`s. It also defines the `teleUpdateAll()` and `autoUpdateAll()` methods which simply iterate over the `ArrayList` and update each component.

Movement

All of the currently legal FRC motor controllers are implemented, with two separate implementations for the Talon SRX, one for PWM and one for CAN. There is also a `FakeMotor` which emulates a motor controller but in fact does nothing.

Sensors

TurtleXtrinsicMagnetometer `TurtleXtrinsicMagnetometer` is a class for an Xtrinsic magnetometer. It has 3 `TurtleRotationSensor`s representing the pitch, yaw, and roll axes.

TurtleAnalogGyro `TurtleAnalogGyro` is a wrapper for WPILib's `AnalogGyro` class.

TurtleDistanceEncoder `TurtleDistanceEncoder` is an encoder that measures distance travelled. The scale between encoder ticks and inches must be passed in.

TurtleRotationEncoder `TurtleRotationEncoder` is an encoder that measures amount rotated. The scale between encoder ticks and degrees must be passed in.

Input

TurtleJoystickAxis `TurtleJoystickAxis` implements `TurtleAnalogInput`, it takes a single axis from a Joystick and turns it into an `AnalogInput` with the range $[-1, 1]$.

TurtleJoystickButton `TurtleJoystickButton` implements `TurtleDigitalInput`. It returns 1 if the button is pressed, and 0 otherwise.

TurtleJoystickPOVSwitch `TurtleJoystickPOVSwitch` implements `TurtleDigitalInput`. It represents a POV switch, also known as a hat switch or a d-pad with diagonals. It returns -1 to 7, but it also contains an enum called `POVValue` which can accept this number and return a more usable value, such as North, South, and Southeast.

TurtleFlightStick This class represents the entire joystick for a flight stick, often used by FRC teams. It contains several `TurtleJoystickAxis`, which can be retrieved from it. It also contains a `TurtleJoystickPOVSwitch` and multiple `TurtleJoystickButton`.

TurtleXboxController This class represents an entire XBox controller. It contains several `TurtleJoystickAxis`, which can be retrieved from it. It also contains a `TurtleJoystickPOVSwitch` and multiple `TurtleJoystickButton`.

4.2.4 Utility

This section covers utilities included in TurtleShell, found in the `util` package.

Unit

This interface is used for classes the represent some type of measurement, such as distance, time, or angle. All implementations wrap doubles, primarily for type safety.

Time This implements `Unit`, and represents time in seconds.

MotorValue This implements `Unit`, and represents a motor power, from -1 to 1.

Distance This implements `Unit`, and represents a distance in inches.

Angle This implements `Unit`, and represents an angle in degrees.

Rate This implements `Unit`. It is generified, so it represents another `Unit` per `Time`.

TurtleMaths

`TurtleMaths` is full of mathematical functions which have proven useful. All are public static, see the JavaDocs for documentation.

4.3 Advanced Features

This section covers advanced features of TurtleShell, such as PID and vision processing.

4.3.1 PID

PID (Proportional Integral Derivative) is a form of closed-loop control system. It uses sensor inputs to determine the proper actions to take in order to reach a target. It is commonly used in industrial control mechanisms for its relative simplicity and accuracy.

Explanation

PID control works by comparing sensor input to a target. It usually uses the error (difference between actual and target), derivative (rate of change of error)[1], and integral (summed error). Sometimes, the double derivative (rate of change of derivative) is used as well. Additionally, the integral is often not used in practice as it is often not relevant to the control system.

The PID controller has a set of constants that it uses, along with the error, derivative, and integral (from sensors) to calculate the motor movements. These constants are empirically tuned. The constants are multiplied by the error in each (PID), and the result is constrained to $[-1, 1]$. This output is then fed to the motor, and the cycle begins anew.

Constants

The behaviour of the PID controller, whether it effectively guides the error to zero or makes it wildly swing, is controlled by the constants. They are multiplied by each of the values to produce the output. One is provided for each value, i.e. one for P, one for I, etc. In TurtleShell, a class in the util package, TurtlePIDConstants, is provided. This class can be used to pass kP, kI, kD, and kDD in a convenient manner, PID constructors should accept it.[2]

Tuning

PID tuning is more of an art than a science. This will cover tuning a PDD2, with the P and D being the primary parameters used. If the system is not closing fast enough, the P should be increased or the D decreased. If the system is oscillating around the desired value, the D should be increased. The DD should work slightly against the D, and can be sued to give it a bit extra ability to reach the target. However, the DD should be far smaller (about 2 orders of magnitude) than D, which should be smaller (one order of magnitude) than P.

[1] n.b. Technically, it is looking for the error in the derivative, but the desired is assumed to be zero.

[2] The prefix k for constants is extremely common.

Interfaces

Two interfaces are provided in TurtleShell. They are both quite simple, providing a boolean to check whether the target has been reached, and a method to put in inputs and return MotorValue(s). `TurtlePID` returns a single `MotorValue`, while `TurtleDualPID` returns an array of `MotorValue`s (with two values). The order of the inputs and outputs is ultimately up to the implementation, but the standard is:

- Inputs should be in the order P → I → D → DD.[3] If there are both left and right, they should be interspersed with left → right → left → etc.
- Outputs, if there are multiple, should be left → right → other (if applicable).

Implementations

TurtlePDD2 `TurtlePDD2`, or Turtle Proportional-Derivative-Double Derivative, is the main single-motor implementation.[4] It uses the error, the derivative, and second derivative. It attempts to make all of these zero.

TurtleZeroPID `TurtleZeroPID` is a dummy controller, it always returns zero. It can be used to disable a PID controller.

TurtleStraightDrivePID `TurtleStraightDrivePID` uses two `TurtlePDD2`s, along with the differences between them, to drive the robot straight for a certain difference. It only uses the encoder readings, it does not have gyro support.

TurtleTurnPID `TurtleTurnPID` is similar to `TurtleStraightDrivePID`, but it turns in place rather than moving straight. It again only uses the encoder readings, it does not have gyro support.

4.3.2 Vision Processing

This is not currently implemented through TurtleShell. NIVision, as well as GRIP, may be used.

4.4 Code Style

This section covers the code style used throughout TurtleShell and the code of Team 1458.

[3] DD and D2 are used interchangeably, they both mean double derivative, or the second derivative.

[4] Fun Fact: The use of PDD2, rather than PID, was learned from the SAS in Kerbal Space Program.

4.4.1 Naming

The naming convention is very similar to the standard Java naming convention. Classes begin with a capital letter, and camel case is used throughout, with the first letter lower case for instance variables. Global constants should be in all capitals.

The prefix "Turtle" is used for most classes in TurtleShell. This helps to avoid namespace collisions (having the same name for two different classes), and also serves to easily categorise classes which are a part of the library.

4.4.2 Comments

The main use of comments should be for JavaDocs on publicly accessible classes, methods, and constructors. They should detail the interactions with the object, but the object itself should be able to be considered as a black box. Comments inside methods or other blocks of code are in general discouraged, because if they become out of sync with the code they add to confusion, but should be used for complicated or seemingly illogical methods or algorithms.

Variables

Variables may be commented with their purpose or expected values. If possible, variables should be restricted to that range by type.

4.4.3 Language

All code and naming should be done in English. Minimal amounts of Latin are acceptable. The spelling should be international, i.e. metre rather than meter, organise rather than organize. Emojis (defined as Unicode code points greater than U+1F600 and above) must not be used in any official or unofficial programming activities.

4.4.4 Indenting, Spacing, and Braces

The default Eclipse settings should be close to suitable for formatting, Control-Shift-F, Command-Shift-F, or Meta-Shift-F should automatically format the code correctly. Tabs should be used, with a width of 4 spaces. Braces should appear at the end of the line, not on a new line by themselves. If a line of code is too unwieldy to fit on a single line, line breaks should take place at logical breaks, such as an "and" or an "or" statement. Line Feeds should be used as a line terminator, there should be a trailing newline.