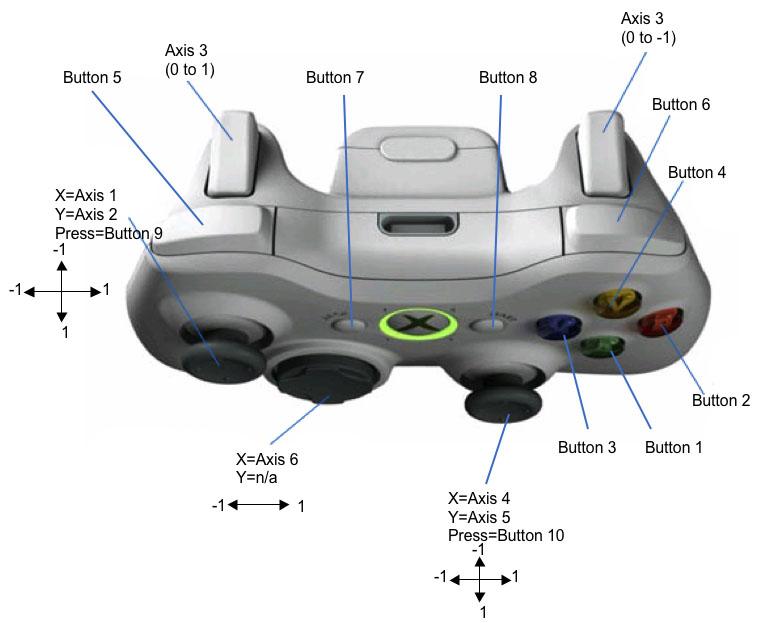
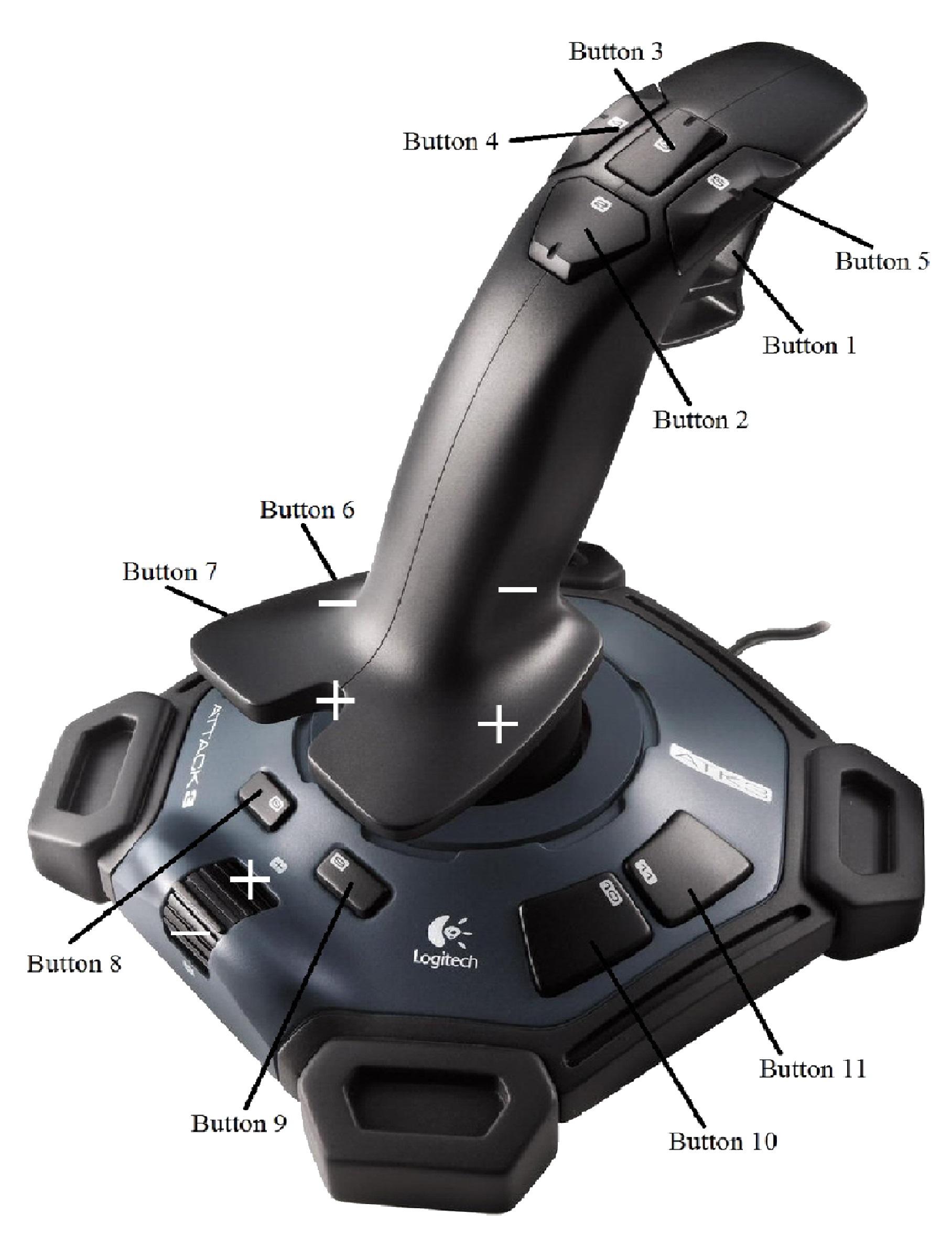
****

**Controller**

****

**Joystick**

****

Getting Started

**How to Install NetBeans**

For more detailed instructions in downloading NetBeans, go to: <http://wpilib.screenstepslive.com/s/3120/m/7885/l/79405-installing-the-java-development-tools>

**Download Bundle:** If you don’t have Java installed on your computer, go to: <http://www.oracle.com/technetwork/java/javase/downloads/index.html> and install the bundle of the JDK and NetBeans.

**Download NetBeans Standalone:** If you do have Java installed on your computer, go to the Netbeans website and navigate to their download page. Press the download button under the column Java SE.

**Java Robot Bases**

Robot bases in Java are templates used for making it easier to program your robot. There are three kinds: SimpleRobot, Command, and Iterative. SimpleRobot gives more freedom allowing custom loops to be made. Command organizes robot functions in subsystems and then allows you to easily set up a command that includes a certain order of these subsystems, which could be run at the same time as others (useful for autonomous where there isn’t a lot of time to do things). Iterative is the easiest to use and it periodically runs autonomous and teleop, meaning that the methods are called multiple times at a certain rate: the period.

**SimpleRobot Base**

SimpleRobot is split into methods that are each called only once and, therefore, require a loop to be placed inside each of them to run that code continuously. It allows for the first part of these methods to be used for code you want to run only once, with code running continuously to be inside the loop. The names of these methods are autonomous() and operatorControl(). No loops are needed to be made in autonomous() if you want things to be happening one function after another. In operatorControl(), however, you need this loop for detecting operator and driver inputs so the robot can adjust accordingly:

public void operatorControl() {

//Code here only runs once on teleop start.

while (isOperatorControl() & isEnabled()) {

//Code here runs repeatedly.

}

}

**Command Base**

Command is split into multiple packages that host multiple .java files: templates, commands, and subsystems. In the templates package, there are three files: OI.java, RobotMap.java, and robotMain.java. OI is used for setting up the joysticks and controller buttons to do a robot function. RobotMap is used for mapping out what sort of pwm or relay values each function uses. robotMain is exactly the same as the Iterative base, but is used for connecting all of the subsystems and commands into one place to be used. The subsystems created are all individual functions on the robot and that is where the motors and pistons are initialized, which are then used in commands. Autonomous includes a set of commands either accomplished concurrently or one after the other. Anything that either the operator and driver does, is controlled by a command, which consists of a subsystem, which consists of a function or functions.

**Iterative Base**

Iterative is very similar to SimpleRobot, but is split up into more methods: robotInit(), autonomousInit(), autonomousPeriodic(), teleopInit(), and teleopPeriodic(). robotInit() is called once when the robot is turned on, which is usually used for starting up the compressor. autonomousInit() is called once on the start of autonomous. autonomousPeriodic() is called multiple times at a certain period, running code so long as autonomous is enabled. teleopInit() is called once on the start of teleop. teleopPeriodic() is called multiple times at a certain period, running code so long as teleop is enabled. The benefits of using Iterative over SimpleRobot is that it doesn’t require any special modifications to setup the code properly and also you can modify the length of the period for autonomous and teleop. I personally prefer Iterative because it’s the simplest to learn from and all of the code is organized, and yet still in the same file.

**Iterative Base Layout**

In the Iterative Base, specific .java files, which are only used for programming an FRC robot, are imported before anything can be done. All of the methods in the Iterative Base are nested in a class, looking something like this:

public class RobotTemplate extends IterativeRobot {

}

The RobotTemplate is just an example name that could be replaced by anything else when you first make a project. Outside of all the methods, but still inside the class, is where you initialize your robot drive, joysticks, motor controllers, relays, compressors, and any other variables. After that, the variables you initialized can be used in any of the methods. The next page shows the general setup of the Iterative Robot. Note: the “//” at the beginning of a line makes it a comment, meaning that it isn’t read by the computer and can be used for taking notes and explaining things.

**Example Drivetrain Code:**

package edu.wpi.first.wpilibj.templates;

import edu.wpi.first.wpilibj.IterativeRobot;

import edu.wpi.first.wpilibj.Joystick;

import edu.wpi.first.wpilibj.RobotDrive;

public class RobotTemplate extends IterativeRobot {

//variables\\

//drive motor PWM values

int FRONT\_LEFT = 1;

int REAR\_LEFT = 2;

int FRONT\_RIGHT = 3;

int REAR\_RIGHT = 4;

//initialize robot drive

RobotDrive robotDrive = new RobotDrive(

FRONT\_LEFT, REAR\_LEFT, FRONT\_RIGHT, REAR\_RIGHT);

//joysticks/controllers

Joystick leftJoy = new Joystick(1);

Joystick rightJoy = new Joystick(2);

//teleoperated\\

public void teleopPeriodic() {

robotDrive.setSafetyEnabled(true);

robotDrive.tankDrive(leftJoy, rightJoy);

}

}

Let’s look at this code line by line:

First, you have the package that this class is a part of, organizing all of the classes you’re using.

Second, there are three lines of code after that that represent the importation of other classes not included in your package, such as which robot base, joysticks, drive, motors, solenoids, relays, camera, and sensors. It may also include other classes not included in FRC.

Third, the name of the program in this example is RobotTemplate, but it could be anything else. So in this line of code, this class chooses the IterativeBase.

Fourth, the variables are initialized and declared to be used later in the code. Four integer constants (numbers that don’t ever change) are set up, representing which PWM connector on the digital sidecar the PWM wire is plugged into. This PWM wire connects to the motor controller which controls the motor which is one of the four wheels driving the robot.

Fifth, once we have those constants set up, we can initialize a RobotDrive object, which requires either two or four parameters, representing the number of motors that are controlling your wheels. Since we’ve already got the numbers, we plug them in as the parameters in a specific order for each drive motor.

Sixth, in order to drive, we need to know two things: what controls the driving and when. So we need to initialize two Joystick objects, since this example code is using tank drive (two joysticks, one for each side of wheels) rather than arcade drive (one joystick, controls all wheels and rotation). The parameter for each joystick represents the USB port that they’re connected to into the driver station laptop/netbook (ranging from 1 to 4), though the numbering and order can be changed in the driver station.

Seventh, the method “TeleopPeriodic()” is called, which represents the teleoperated (robot controlled by humans) time after autonomous (robot controlled by code only) in a match. “Periodic” means that the code repeats at a specific rate, which in this case is very fast: 20 milliseconds.

Eighth, “setSafetyEnabled(true)” ensures that the drive motors don’t keep going if you lost control of the robot, preventing any sort of damage or injury. The only time when this should be disabled is during autonomous, since the robot only relies on the code during that time.

Ninth, tank drive for the drive motors is set up and, as mentioned earlier, is necessary for allowing the driver to drive the robot with the two joysticks, which are its parameters.

**Example Non-Drivetrain Motors Code:**

package edu.wpi.first.wpilibj.templates;

import edu.wpi.first.wpilibj.IterativeRobot;

import edu.wpi.first.wpilibj.Joystick;

import edu.wpi.first.wpilibj.Victor;

public class RobotTemplate extends IterativeRobot {

//variables\\

//initialize motors

Victor motor1 = new Victor(5);

Victor motor2 = new Victor(6);

//joysticks/controllers

Joystick controller = new Joystick(1);

//teleoperated\\

public void teleopPeriodic() {

if (controller.getRawButton(1)) {

motor1.set(1);

} else {

motor1.set(0);

}

if (controller.getAxis(Joystick.AxisType.kY) > 0) {

motor2.set(-0.5);

} else {

motor2.set(0);

}

}

}

There are a few differences from this code and the drivetrain code. In this one, the motors are initialized individually, with their parameters representing where each motor’s PWM wires are plugged in on the digital sidecar. Something worth noting is that the controller plugged into the driver station computer would be classified as a joystick in the code. In “teleopPeriodic(),” we now need to allow the operator to be able to press a button and cause the motors to spin. Using the button layout for the controller, you can identify where the button one is. So if button one is pressed, then motor1 is set to 100% forwards, and if the up arrow on the d-pad is being pressed on the controller, then motor2 is set to 50% backwards. Once either button is pressed and released, they automatically stop moving because of the else statements coming right after each if statement.

**Example Pneumatics Code:**

package edu.wpi.first.wpilibj.templates;

import edu.wpi.first.wpilibj.IterativeRobot;

import edu.wpi.first.wpilibj.Joystick;

import edu.wpi.first.wpilibj.Compressor;

import edu.wpi.first.wpilibj.Relay;

public class RobotTemplate extends IterativeRobot {

//variables\\

//initialize compressor

Compressor compressor = new Compressor(1, 1);

//initialize relays

Relay piston1 = new Relay(2);

//joysticks/controllers

Joystick controller = new Joystick(1);

//called once on robot activation\\

public void robotInit() {

compressor.start();

}

//teleoperated\\

public void teleopPeriodic() {

if (controller.getRawButton(1)) {

piston1.set(Relay.Value.kForward);

}

else if (controller.getRawButton(2)) {

piston1.set(Relay.Value.kBackward);

} else {

piston1.set(Relay.Value.kOff);

}

}

}

In this code, notice the difference in the files being imported: compressor and relay. A relay like a motor controller except that it can only output either 100% forwards, none, or 100% backwards. This makes a relay perfect for using it for pistons because they can only be forwards, holding its current position (off), or backwards Relays can also be used for making lights blink on or off.

The compressor is required for increasing the air pressure for pistons to be able to extend and retract on demand. Usually, it takes a while for this pressure to build up, but once it gets past a certain pressure, the compressor needs to turn itself off in order to not cause any sort of explosion that could result from it. When we first initialize the compressor, it requires two inputs for its parameters: the first one being which Digital I/O slot the pressure switch plugs into, and the second one being which relay slot the compressor’s relay plugs into. For the compressor to work immediately, we need to use a new method called “robotInit(),” which starts the compressor the moment the robot is enabled.

In teleop, if the controller’s button one is pressed, piston1 extends, and if button two is pressed, piston1 will retract. If neither button is pressed, the piston will remain where it is based on which button was pressed last, being set to the off state. If both buttons are pressed at the same time, piston1 will extend no matter what. To change that, you need to replace “kForward” with “kBackward” and vice versa when either button is being pressed, which are under the “if” and “else if” statements.

**Example Autonomous Code (Drivetrain):**

package edu.wpi.first.wpilibj.templates;

import edu.wpi.first.wpilibj.IterativeRobot;

import edu.wpi.first.wpilibj.Timer;

import edu.wpi.first.wpilibj.RobotDrive;

public class RobotTemplate extends IterativeRobot {

//variables\\

int countLoops = 0; //loop variable

//drive motor PWM values

int FRONT\_LEFT = 1;

int REAR\_LEFT = 2;

int FRONT\_RIGHT = 3;

int REAR\_RIGHT = 4;

//initialize robot drive

RobotDrive robotDrive = new RobotDrive(

FRONT\_LEFT, REAR\_LEFT, FRONT\_RIGHT, REAR\_RIGHT);

//called once when autonomous first starts\\

public void autonomousInit() {

robotDrive.setSafetyEnabled(false);

}

//autonomous\\

public void autonomousPeriodic() {

if (countLoops == 0) {

robotDrive.drive(-0.5, 0.0);

Timer.delay(7.0);

countLoops++;

}

else if (countLoops == 1) {

robotDrive.drive(0.0, 0.5);

Timer.delay(7.0);

countLoops++;

}

else if (countLoops == 2) {

robotDrive.drive(0.0, 0.0);

countLoops++;

}

}

}

Autonomous is usually easy to make, but hard to master and perfect. Debugging autonomous can take a very long time, having to tweak certain numbers to perfection. Sensors are very useful because you don’t have to find those “perfect numbers” that make your robot do exactly as you want it to do. To start off with the basics of autonomous, we’ll wait on learning about how to program sensors.

In the code, notice that a new file is imported: Timer, which will allow your robot to wait a certain number of seconds before moving on to do its next task in autonomous. In where the variables are initialized, notice there aren’t any joystick objects. This is because we’re only focusing on autonomous and in autonomous, the robot does everything by itself, only running off of the code its given, meaning that there are no driver/operator inputs (joysticks and controller).

Something worth noting is that autonomous, in most games, lasts 15 seconds and in “autonomousPeriodic(),” the robot reads the autonomous code again and again until those 15 seconds are up. In order to prevent it from doing the exact same code more than once in a match, we need to create an “if” statement around parts of autonomous and then increment the variable every time the robot reads the code and loops. In the above example, we used “countLoops” and there are three parts of this autonomous code that are ran once: the first two do something, and the last one turns everything off to prepare the robot for the teleoperated portion of the match.

When autonomous starts, “autonomousInit()” runs and disables the safety put in place to prevent accidents from occurring when the robot loses connection with the driver/operator, which obviously means that the robot safety should be on only when it’s teleop. In this place, anything you want to occur immediately should be here, followed by a timer delay to give time in between this step and the next.

The first set of instructions given to robot in “autonomousPeriodic()” controls the driving of the robot, with the first parameter controlling the robot’s speed, either forwards or backwards, and the second parameter controlling the robot’s turning, either left or right. The robot continues this commanded movement until told otherwise, unlike in teleop because it loops that part of the code very quick after each iteration. Each step includes a timer delay (in seconds), stopping the robot from reading the following code and to complete each loop, resulting in the robot moving for a certain amount of time.

Finally, when “countLoops” equals two, the robot is stopped from moving any longer and countLoops no longer fits any of the “if” statements conditions because it got incremented to three. Notice that in the example above, the robot runs through the code in only 14 seconds, rather than 15, which is setup like that in order to give the robot time to slow down for the driver and operator to be ready for teleoperation in the second part of each match.

**Example SmartDashboard Code:**

Let’s say you’re trying to debug your robot code, but you have no idea what your sensors, motors, relays, etc. are reading back to the robot since nothing is working, for example. Or, let’s say your operator needs to know when the robot is lined up and ready to score. This is where the SmartDashboard comes in, which allows you to add any variables’ values into a list or interactive display for everyone to understand what’s going on with their robot while it’s running. There are only a few modifications that are needed to be done on top of the code you’ve already created.

First, you need to import the SmartDashboard:

import edu.wpi.first.wpilibj.smartdashboard.SmartDashboard;

Second, you need to take your variables’ outputs and plug them into one of these (depending on each value’s data type):

SmartDashboard.putNumber(“Shooter Motor”, motor1.get());

SmartDashboard.putDouble(“Wheel Encoder”, encoder1.get());

SmartDashboard.putInt(“Front Left Wheel - PWM”, FRONT\_LEFT);

SmartDashboard.putBoolean(“Looping?”, !isStopped);

SmartDashboard.putString(“Type of Autonomous?”, autoType);

The first parameter is the String name of the title of each item displayed on the SmartDashboard, and the second parameter is value with its associated data type.

Finally, you need to place these either at the end of “teleopPeriodic()” or at the end of “autonomousPeriodic()” (which would be located outside of the autonomous “if” statements so that it displays the information every time autonomous loops).

Not only can you write values to the SmartDashboard, but you can also read values from it, giving you instant access to the values controlling certain variables in areas of your code, especially autonomous. This also can be used to give the driver/operator freedom over selecting a specific autonomous mode each match to deal with whichever position you start at on the field. All you have to do is set a variable equal to one of the following (with matching data types):

getBoolean(<name>); , getDouble(<name>); , getInt(<name>); , getNumber(<name>); , getString(<name>);

<name> corresponds to each item’s name on the SmartDashboard.

**Connecting with the Robot**

There are two ways that you can connect your programming laptop/driver station to the robot: wired and wireless. Wired tends to be more reliable because you know for sure that any problems you have won’t be caused by no clear connection. To establish wired connection, grab an ethernet cable and plug in one end into one of the robot’s radio’s ethernet ports, and then plug in the other end into the laptop’s ethernet port.

If this is the first time the laptop is being used for programming, you need to setup the laptop’s IP address and subnet mask. Navigate to the laptop’s Network and Sharing Center and click on the new network’s connection. Then, click on Properties and double-click on Internet Protocol Version 4 (TCP/IPv4). If it’s not already selected, click on “Use the following IP address” bubble. For the IP address, type in 10.03.13.6 and for the Subnet mask, type in 255.255.255.0 and ignore the Default gateway. Press OK and you should be able to connect to the robot.

A useful tip for seeing if the laptop is communicating with the robot, go to “run” through the laptop’s search bar and type “cmd”. Once you see a black screen with some text in it, type “ping 10.03.13.2”, and if you receive a message back with how long it took to get there and back, then you know that you have a connection.

**Deploying Code**

Deploying your code to the robot is easy if you have communication between the laptop and the robot. In NetBeans, on the left tab where the file directories are, right-click on the root folder of your project and select “Set as Main Project.” Once your code is ready to be deployed, click the green arrow button on the top tab in NetBeans.

If this is your first time using Java after switching the cRIO from another programming language, you should be faced with a wait message in the bottom output tab that continuously searches for connection. This error is most likely to occur if you have Windows 8, so you need to go to your laptop’s Control Panel and navigate to System and Security and then click on Windows Firewall. On the left tab, you should be able to click the button “Turn Windows Firewall on or off.” Once you do so, close out of NetBeans and manually turn off the robot. Turn the robot back on and open up NetBeans again. When you click the green arrow button again, it should successfully deploy. The one last thing to do is to manually restart the robot again, which will be after fifteen seconds in the output tab in NetBeans, since it won’t do it automatically the first time. Now it should work any other time you deploy code. Don’t forget to turn your firewall back on.

**Understanding the Driver Station**

**What is the Driver Station?**

The Driver Station allows you to get information about how many volts the battery has, if the laptop has communication with the robot, what USB devices are plugged into the laptop for operation of the robot, how long the robot has been enabled, which mode the robot is running, how much battery power the laptop has, which team number you are, and what USB configuration the joysticks are in.

Also, the Driver Station allows you to choose the desired mode for the robot to run, to enable and disable the robot, to set/change your team number, to set the practice round timing, to rearrange the USB configuration for the joysticks, to type notes for yourself, to change the dashboard type, and to reboot the cRIO.

**Operation**

In order to enable the robot, the three lights need to be green, located on the far-left side of the Driver Station under the Operation tab. But before you enable it, you need to choose the correct mode the robot will run. Teleoperated and Autonomous runs each separate part of the code, being “teleopInit()” and teleopPeriodic(),” or “autonomousInit()” and “autonomousPeriodic(),” respectively. Running Practice mode simulates a match played at the competition, including teleop and autonomous. Lastly, the least used one is the Test mode, which allows you to test any kind of code you put inside of the methods called “testInit()” and “testPeriodic().”

**Diagnostics**

The Diagnostics tab tells you about what USB devices are plugged into the laptop and what communication it has with which parts of the robot, including the ethernet link, radio, and cRIO. Also, you can see all of the error messages and status updates to tell you of any mistakes in the code or with communication. In the bottom-right corner, you can reboot the cRIO.

**Setup**

The Setup tab allows you to type in your team number and also allows you to setup the USB joystick configuration and ordering, which is located in your code, depending on which USB slot controls what. Another important thing you can do is modify how long each part of the match is for when you run the Practice mode. Selecting your Dashboard Type should always be Java, since we’re using it. Once selected, the Java dashboard should appear, granting you the ability to layout the variables and their values when the robot is running, but we’ll go into that later with greater detail.

**Understanding the Robot**

**The Robot Signal Light**

The Robot Signal Light (RSL) blinks a specific pattern to tell us of what state the robot is currently in. There are five patterns: fast blink (cRIO isn’t communicating with the Driver Station), slow blink (the robot is disabled), short on/long-off blink (either the battery voltage is low or there isn’t code and the robot is disabled), long on/short off blink (the robot is in teleop mode), and solid on (the robot is in autonomous mode). Knowing what each specific pattern is gives you a general idea of what’s going on with a robot even when it’s far away.

**Victor Motor Controllers**

Even though the victor motor controllers follow a simple pattern, it’s still easy to mistake one for the other. There are four LED states: solid yellow (neutral with speed set to 0.0), solid green (full-speed forward), solid red (full-speed backward), and flashing orange (no PWM wire connection). When the victor’s speed is set to 1.0 in one direction, it can throw off some people if moving forwards with the drivetrain results in the victor’s LED turning red and 1.0 backwards yielding a green LED color.