FRC Programming

Team 1351

2020

Contents

1 Overview

This is team 1351s FRC programming documentation for the essential knowledge we use on this team. This covers motors and actuators (mostly Talons and Servos), sensors, controllers, pneumatics, the robot class, and command based programming.

2 Talons

2.1 What is a Talon

A Talon is a motor controller that allows to change the speed of the motor, which can be based on various different controlling methods. Each talon has a unique id between 0 - 63, so the code knows which talon to program when the code is executed. There are two types of talons: TalonSRX and TalonFX. TalonSRX is used for most motors while TalonFX is used for falcons.

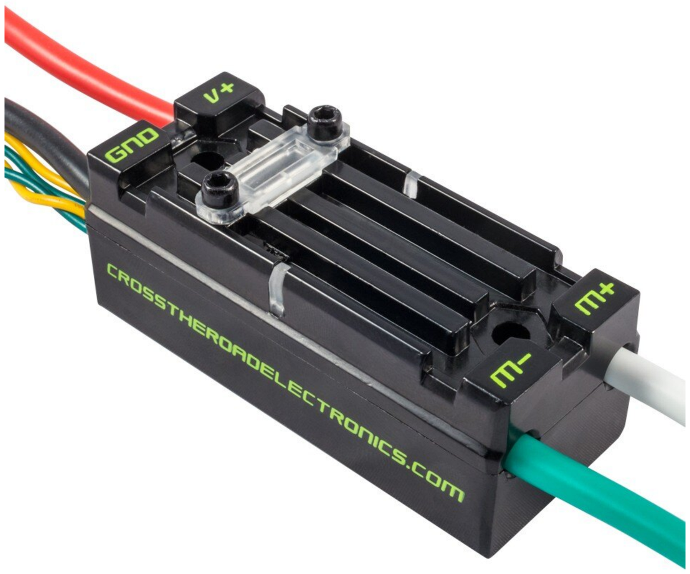

Figure 1: A picture of a talon

2.2 Configuring a Talon

2.2.1 Constructing a Talon

When creating a talon, use the WPI_TalonSRX object or the WPI_TalonFX object.

```
WPI_TalonSRX talon;
WPI_TalonFX falcon;
```

The Talon Constructor only takes in 1 parameter: the id of the talon.

```
talon = new WPI_TalonSRX(0);
falcon = new WPI_TalonFX(1);
```

2.2.2 Configuring Talon Hardware

It is generally safe to configure every talon to the default setting before messing with anything else.

```
talon.configFactoryDefault();
falcon.configFactoryDefault();
```

Talons may be inverted in hardware, and you may need to invert them in code again.

```
talon.setInverted(true);
falcon.setInverted(true);
```

If you are using followers (look at 2.3.3 for more info), it might be useful to use the InvertType class instead of a boolean for the setInverted() function.

```
talon.setInverted(InvertType.None);
falcon.setInverted(InvertType.InvertMotorOutput);
talon.setInverted(InvertType.OpposeMaster);
falcon.setInverted(InvertType.FollowMaster);
```

It can be also useful to limit the max speed. The code below limits the max speed of the motor to be 0.5 (percent output).

```
talon.configPeakOutputForward(0.5);
falcon.configPeakOutputForward(0.5);
```

To set the max speed of the motor in the negative direction, use:

```
talon.configPeakOutputReverse(0.5);
falcon.configPeakOutputReverse(0.5);
```

You can also limit the current draw of a motor. The code below sets the maximum current draw 20 amps.

```
talon.configContinuousCurrentLimit(20);
```

For falcons, you can set the stator current limit and the supply current limit. The stator current limit is the output current, while the supply current is the input current. The code below sets the stator current limit to 20, but only triggers if the current exceeds 25 for at least 2 seconds.

```
falcon.configStatorCurrentLimit(new
    StatorCurrentLimitConfiguration(true, 20, 25,
    2));
```

Do NOT set your trigger to be less than your current limit. The code below sets the supply current limit to 30 and triggers if it gets above 30 with no wait time.

```
falcon.configSupplyCurrentLimit(new
    SupplyCurrentLimitConfiguration(true, 30, 30,
    0));
```

Talons can be set to either coast or brake mode. Coast mode gives the talon less force when changing the speed so it "coasts" to a speed (or stop). Brake mode forces the talon to change speeds almost immediately, which is provides better control but takes more battery voltage. The code below sets a talon to coast mode and a falcon to brake mode.

```
talon.setNeutralMode(NeutralMode.Coast);
falcon.setNeutralMode(NeutralMode.Brake);
```

Sensor hardware attached to the talon may also need to be configured (read Sensor for more info).

2.3 Control Modes

Talons have various different Control Modes that can be used to control the power of the motor. A Talon can control a motor with the set function.

```
talon.set(ControlMode.xxx, double value);
```

for falcons use TalonFXControlMode.xxx instead

```
falcon.set(TalonFXControlMode.xxx, double value);
```

2.3.1 Disabled

Disabled stops the motor from moving at all.

```
talon.set(ControlMode.Disabled, 0);
```

2.3.2 PercentOutput (Duty Cycle)

PercentOutput sets the motor to go at a speed from range -1.0 to 1.0, and the actual speed of the motor is based on battery voltage multiplied by the input value. The code below sets the talon to got to 50%.

```
talon.set(ControlMode.PercentOutput, 0.5);
```

The default Control Mode is PercentOutput, so just inputting 0.5 will do the same thing.

```
talon.set(0.5);
```

2.3.3 Current

Current sets the motor to use a certain amperes to move the motor. The code below sets the motor to use 10 amps of current.

```
talon.set(ControlMode.Current, 10);
```

To make the motor spin in the opposite direction, just put a negative sign in front of input value

```
talon.set(ControlMode.Current, -10);
```

Remember that this controls current. The current draw of the robot depends on countless factors, and the current draw moving one way may be a different speeds as if you were moving the other way (especially if your are move with or against gravity).

2.3.4 Follower

Follower mode sets the talon to mimic the same talon with the inputted ID. This code sets the talon to mimic the talon with id 0.

```
talon.set(ControlMode.Follower, 0);
```

To make code less confusing, use the getDeviceID() function to make it clear which talon is being followed. The code below makes it obvious talon1 is following talon0.

```
talon1.set(ControlMode.Follower, talon0.getDeviceID
    ());
```

Instead, you can also use the follow() function.

```
talon1.follow(talon0);
```

2.3.5 Position

Position mode uses an encoder and PID to have to motor move to a certain position. The code below sets the talon to move 500 ticks (see Sensors for more info).

```
talon.set(ControlMode.Position, 500);
```

2.3.6 Velocity

Velocity mode uses an encoder and PID to have a motor move at a certain velocity. The code below sets the motor to a speed of 150 ticks / 100 ms (see Sensors for more info).

```
talon.set(ControlMode.Velocity, 150);
```

3 Sparks

3.1 What is a Spark

A Spark is a motor controller that allows to change the speed of the motor (specifically Neos). Although they do perform the same function as talons, the libraries are different and require a seperate section to be explained

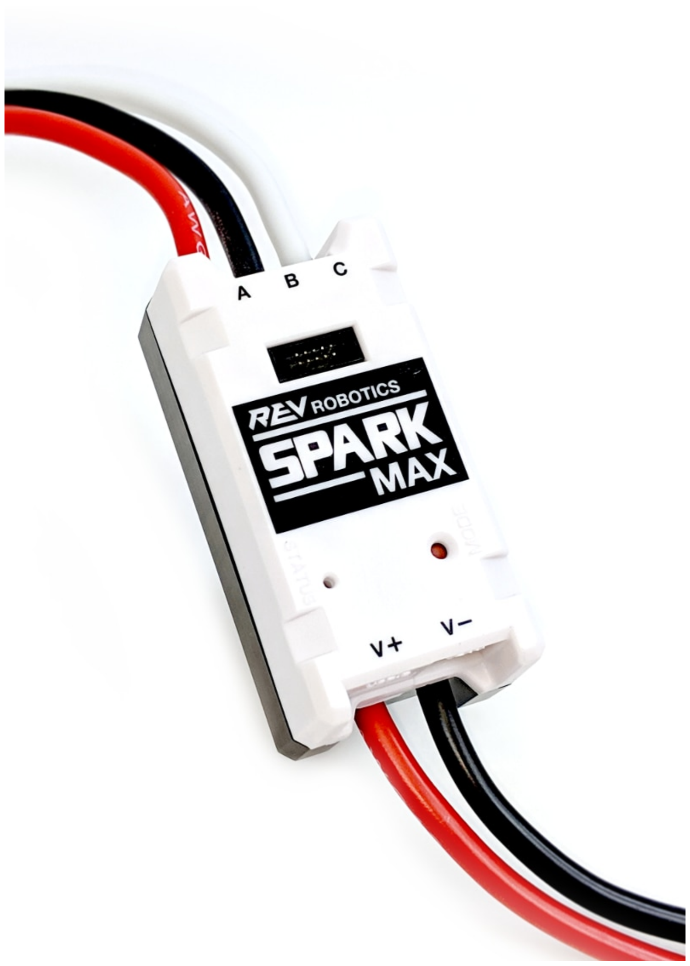

Figure 2: A picture of a spark

3.2 Configuring a Spark

3.2.1 Constructing a Spark

When creating a spark, use the CANSparkMax object.

```
CANSparkMax spark;
```

The CANSparkMax constructor takes in 2 parameters, the id of the Spark, and the type of motor (brushed or brushless). The code below configures a spark with id 0 for a brushless motor.

```
spark = new CANSparkMax(0, MotorType.kBrushless);
```

3.2.2 Configuring Spark Hardware

Similar to talons, configuring sparks to their default setting before messing with other hardware is generally safe.

```
spark.restoreFactoryDefaults();
```

To invert a spark, use the setInverted() function.

```
spark.setInverted(true);
```

To configure the current limit on the spark, there are several ways. First, a raw limit can be set to shut off the motor output when that current limit is exceeded. The code below sets the secondary current limit (when the output shuts off) to 30 amps.

```
spark.setSecondaryCurrentLimit(30);
```

There is also a way to set a current limit without shutting off the output (called a smart current limit). This method reduces the voltage applied when the current is exceeded to go under the limit. The code below sets the smart current limit to 20.

```
spark.setSmartCurrentLimit(20);
```

The smart current limit must be LESS then the secondary current limit. Smart current limits are also recommended for brushless motors.

Sparks have a built-in encoder and support an external encoder and limit switches. Information for these is under the sensors section.

3.3 Controlling Sparks

Sparks can be controlled in several ways. Most of these methods involve using a CANPIDController.

3.3.1 Follower

To set a spark to follow another spark, just use the follow() function. The code below sets spark2 to follow spark1.

```
spark2.follow(spark1);
```

3.3.2 Duty Cycle

To control a spark using voltage, use the set() function (between -1 and 1). The code below sets the spark to move at 50% voltage forward.

```
spark.set(0.5);
```

The rest of these sections involve using a PIDController and the setReference() method (read the PID section for more information ot control)

3.3.3 Position

To set the spark to a certain position, use the ControlType.kPosition mode. The code below sets the position to 100 (for sparks, its revolutions).

```
spark.getPIDController().setReference(100,
    ControlType.kPosition);
```

3.3.4 Velocity

To set the spark to a certain velocity (in rpm), use ControlType.kVelocity. The code below will run the spark to run at 3000 rpm.

```
spark.getPIDController().setReference(3000,
    ControlType.kVelocity);
```

3.3.5 Current

To set the motor to use current (controlling torque), use the ControlType.kCurrent mode (this is NOT affected by PID values). The code below sets the motor to use 20 amps

```
spark.getPIDController().setReference(20,
    ControlType.kCurrent);
```

3.3.6 Voltage

To set the motor to use voltage instead of current (controlling speed), use the ControlType.kVoltage mode (this is NOT affected by PID). Voltage differs from duty cycle as the input is the voltage given while duty cycle is a % of the maximum voltage. The code below sets the motor to use 5 volts

```
spark.getPIDController().setReference(5,
    ControlType.kVoltage);
```

4 Other Motors / Actuators

4.1 Servos

4.1.1 What is a Servo

A Servo is similar to a motor, but it only has 180 degrees of motion. The Servo is plugged into the PWM on the roborio, and the slot it is plugged into determines the id.

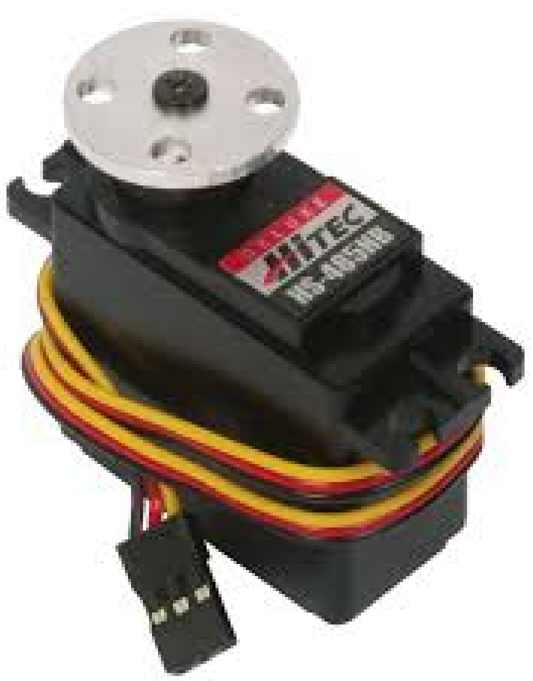

Figure 3: A picture of a servo

4.1.2 Uses of Servos

Servos are useful in situations where you need a ”mini motor” since they are a lot smaller. Of course, the job should not require more than 180 degrees of motion.

4.1.3 Programming Servos

In order to program a servo a new Servo object must be created

```
Servo servo;
```

The Servo constructor takes in one parameter, the ID of the servo. The code below initializes a servo with ID 0.

```
servo = new Servo(0);
```

In order to set the angle of a servo, use the setAngle() function. The code below sets the servo to 90 degrees

```
servo.setAngle(90);
```

4.2 Linear Actuators / Other PWM

Linear Actuators or other devices that plug into the PWM slot and have a mix and max position can use the PWM class. The class is very similar to the Servo class.

Figure 4: A picture of a linear actuator

4.2.1 Uses of Linear Actuators

Linear Actuators can be used whenever you need to move along a fixed position, similar to a servo. However, these move forward and back instead of turning and are much more durable.

4.2.2 Programming Linear Actuators

In order to program a linear actuator a new PWM object must be created

```
PWM linearActuator;
```

The PWM constructor takes in one parameter, the ID of the PWM. The code below initializes a PWM with ID 0.

```
linearActuator = new PWM(0);
```

In order to set the position of a linear actuator, use the set() function. The code below sets the linear actuator position to 0.5, or 1/2 way extended

```
linearActuator.set(0.5);
```

5 Sensors

5.1 What are sensors

Sensors return feedback that can be used in the code for several purposes, which includes controlling talons. There are hundreds of different sensors, but FRC uses three main ones. While the sensor values alone can be used to control motors or other actuators, they are generally combined with closed loop control (see section below for more info) to actually control the motors more accurately.

5.2 Switches

5.2.1 What is a switch

A switch is a device that returns true or false, depending on whether it is pressed or not pressed. A switch can be wired either to the DIO port on the Roborio, or through a sensor wire connected to a talon or spark.

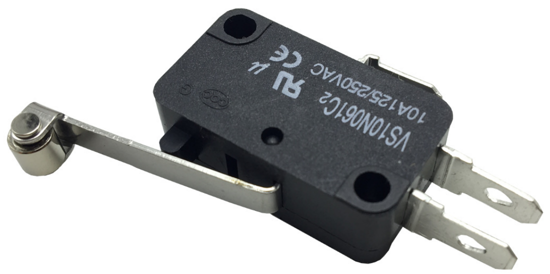

Figure 5: A picture of a switch

5.2.2 Uses of a Switch

A switch can be used for any of these reasons:
1. A limit switch, which is placed at the end of any mechanical system that stops something from moving past where it is supposed to.
2. To detect whether an object is present or not (such a frc game piece)
3. To reset an encoder (see Encoder for more info)

5.2.3 Switches through the roborio

If a switch is wired through the roborio, a new object needs to be created. This object is called a DigitalInput.

```
DigitalInput switch0; //switch is a key word in
    java
```

Each switched plugged into the roborio has an id, and the id is determined based on which DIO slot it is plugged into. the constructor for DigitalInput takes one input, the id for the switch. The code below initializes the switch to have an id of 0.

```
switch0 = new DigitalInput(0);
```

To read the value of a switch, use the get() function

```
boolean isSwitchPressed = switch0.get();
```

5.2.4 Switches through talons

If a switch is wired through the talon, then no new object needs to be created. Up to two switches can be wired to one talon. To get the value of the switch use getSensorCollection().

```
talon.getSensorCollection.isFwdLimitSwitchClosed();
```

This returns if the switch is pressed. If you want to get the value of the other switch, use:

```
talon.getSensorCollection.isRevLimitSwitchClosed();
```

The switches that are wired up automatically act as limit switches, and no limit switch code is necessary if they are wired to the talon. **Make sure that the switch is not wired backwards.** Since this cannot be reversed in code, the switch **must** be wired electrically correct; otherwise, the limit switch code will be broken.

5.2.5 Switches through sparks

Similar to switches through talons, no new object needs to be created. To get the value of each of the switches, use either the getForwardLimitSwitch() function or the getReverseLimitSwitch() function.

```
spark.getForwardLimitSwitch(LimitSwitchPolarity.
    kNormallyOpen);
spark.getReverseLimitSwitch(LimitSwitchPolarity.
    kNormallyOpen);
```

If the switch is wiring is inverted, then change kNormallyOpen to kNormally-Closed to fix it in code.

```
spark.getForwardLimitSwitch(LimitSwitchPolarity.
    kNormallyClosed);
```

5.3 Encoders

5.3.1 What is an encoder

An encoder is a device that counts "ticks" and is attached to the talon. For each revolution of the motor, x amount of ticks pass (otherwise known as the CPR). This can differ based on what encoder is being used, but the CPR is generally 1000.

Figure 6: A picture of an encoder

5.3.2 Uses of an encoder

An encoder has two main purposes. First, it tracks position of a motor. Position tracking is very useful, as there are countless cases where we want the robot to move to a certain position, or for a mechanism to be at a certain spot. Second, it is able to get a *near* instantaneous velocity of the motor. This can also be very useful, as it attempts to compensate for outside factors when setting speed, which PercentOutput fails to do.

5.3.3 Types of Encoders

There are two main types of encoders: Quad Encoders and Mag Encoders. Quad Encoders use mechanical ticks to count, making the more accurate but also more frail. Mag encoders use magnetic fields to count, making the less accurate but more durable. Generally, we use Quad Encoders on our team.

5.3.4 Ticks Per Inch

Since the value that an encoder returns is in ticks, it is easier to create a ticks per inch conversion so we can have position in a meaningful unit. Since velocity is in ticks / 100ms, we can just use the same value multiplied by 10 for a conversion to inches / second. To get ticks per inch, you can do two methods.
1. Take the $CPR * GearRatio/CircumferenceOfWheel$
2. Make the robot move a certain known distance in inches, find how many ticks was traveled in the process, and your ticks per inch is $ticks/inches$ (less accurate, but gear ratio is a hard number to get)

5.3.5 Programming an encoder through talons

An encoder must be configured through the talon. Use the configSelectedFeedbackSensor() function to do this.

```
talon.configSelectedFeedbackSensor(FeedbackDevice.
    QuadEncoder);
```

If you wanted to use a Mag Encoder, you would use FeedbackDevice.CTRE_MagEncoder_Absolute or FeedbackDevice.CTRE_MagEncoder_Relative (I won't go over the differences here). To get the current position of the encoder, use:

```
talon.getSelectedSensorPosition();
```

To get it in readable units, use the calculated ticks per inch value to convert. Since the encoder position is scrambled every time the robot is powered off and on again, it is important to be able to reset the encoder if you need position (not distance). Generally, a limit switch is used to reset the encoder (0 is where the switch is located). To reset the encoder, use setSelectedSensorPosition(). The code below sets the encoder value to 0.

```
talon.setSelectedSensorPosition(0);
```

When using an encoder to make the robot move, ControlMode.Position and ControlMode.Velocity are generally used. However, both of these use PID control in order to work. (see PID below).

5.3.6 Programming an encoder through Sparks

Sparks have both a built-in encoder and provide the option to access an external encoder. To get the built-in encoder, use the getEncoder() function, and to get the external encoder, use the getAlternateEncoder() function.

```
spark.getEncoder();
spark.getAlternateEncoder();
```

After getting the encoder, you can access all of its features. To get position and velocity, use the getPosition() and getVelocity() functions respectively.

```
spark.getEncoder().getPosition();
spark.getEncoder().getVelocity();
```

The units of the position and velocity values are the number of rotations and rpm (rotations per minute).

To reset the encoder position, use the setPosition() function. The code below sets the encoder position to 0;

```
spark.getEncoder().setPosition(0);
```

Conversion factors can be set in the encoder itself, so values like rotationperinch can be set inside the encoder instead of having to be constantly multiplied. This changes the value that getPosition() and getVelocity() return to be multiplied by the scale factor. The code below sets the conversion factor for position to 20 and for velocity to 30.

```
spark.getEncoder().setPositionConversionFactor(20);
spak.getEncoder().setVelocityConversionFactor(30);
```

5.4 Gyro

5.4.1 What is a Gyro

A gyro is a sensor measures the angle of the robot. It however does not read between 0 - 360, if the robot turns one full cycle and then another 90 degrees, the gyro will return 450, not 90. Clockwise is positive, counterclockwise is negative.

5.4.2 Uses of a gyro

Knowing the angle of the robot is very useful for autonomous, as the angle of the robot is necessary in order to turn the robot without driver input.

5.4.3 Programming a gyro

In order to use a gyro, a new Gyro object must be created.

```
ADXRS450_Gyro gyro;
```

The constructor for the Gyro does not require any inputs, as only one gyro can be on a robot.

```
gyro = new ADXRS450_Gyro();
```

To get the current angle of the gyro, use the getAngle() function

```
double robotAngle = gyro.getAngle();
```

When moving using the Gyro, a PID loop is usually used to control the robot (see PID below)

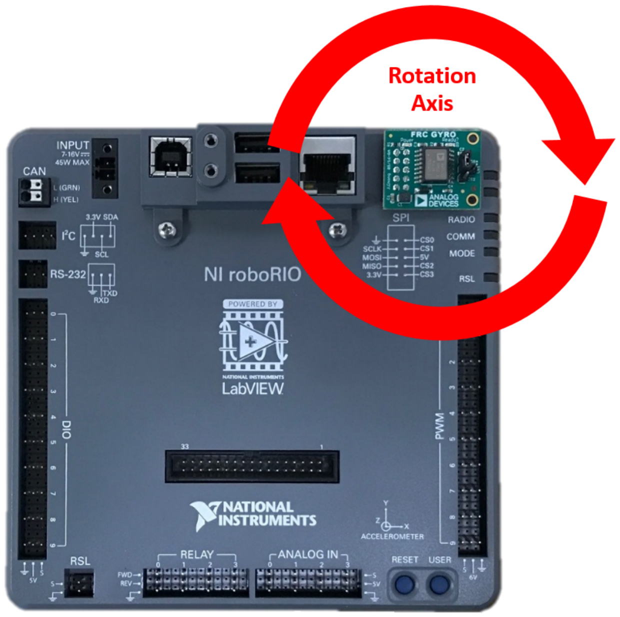

Figure 7: A picture of a gyroscope

6 Closed Loop Control

6.1 What is Closed Loop Control

Closed Loop control requires using feedback from sensors in a continuous loop to accurate move the motor to a certain position or velocity.

6.1.1 Types of Closed Loop Control

While there are several types of closed loop control used, the two main ones used in robotics are bang-bang control and PID control.

6.2 Feed Forward

Feed Forward is a constant value being applied to a system. This can be used for velocity control (see 6.3.4 for more info) or to fight against a constant force (such as gravity).

6.3 Bang-bang control

6.3.1 What is Bang-bang control

Bang bang control is very simple to write and easier to tune, but less accurate than PID control. It follows a very simple rule: Go forward if you are behind, and go backward if you are ahead. Bang bang just runs at the same speed or acceleration (based on if you are running position or velocity mode), just moving in a different direction. Basic bang-code would look like this (pseudo code):

```
while(!inTreshold){
    if(currentPos < desiredPos){
        setTalonSpeed(speed);
    } else if(currentPos < desiredPos){
        setTalonSpeed(-speed);
    } else {
        setTalonSpeed(0);
        inTreshold = true;
    }
}
```

In the pseudo code below, the loop goes until the position matches the desired position. The speed of the motor is set to a constant value called ”speed”.

6.3.2 Thresholding

Since it is nearly impossible to actually get to the to the exact desired position, a threshold is used. The threshold is basically a range that the loop is allowed to be in for the loop to end, so the loop can actually end in a reasonable amount of time. This is pseudo code of bang-bang control with thresholding in place.

```
while(!inTreshold){
    if(desiredPos - currentPos > threshold){
        setTalonSpeed(speed);
    } else if(currentPos - desiredPos < -threshold)
        {
        setTalonSpeed(-speed);
    } else {
        setTalonSpeed(0);
        inTreshold = true;
    }
}
```

6.3.3 Timing

While having a threshold is useful, it is also important to make sure the robot stays in that threshold for a certain amount of time. This can be done by either counting the amount of loop times that have passed or seeing if the amount of time spent in the loop is greater than a number. The code below does this with checking the number of loop times that have passed.

```
while(count < 5){
    if(desiredPos - currentPos > threshold){
        setTalonSpeed(speed);
        count = 0;
    } else if(currentPos - desiredPos < -threshold)
        {
        setTalonSpeed(-speed);
        count = 0;
    } else {
        setTalonSpeed(0);
        count++;
    }
}
```

6.4 PID Control

PID Control, while more difficult to write and tune, can be a lot more accurate is done correctly. PID Control uses three variables (P, I, and D) and feedback from a sensor to determine motor speeds.

6.4.1 Programming a PID Controller

There are a few different ways to write up PID code, but the main method to write PID is using the PIDController object.

```
PIDController pidController;
```

The PIDController constructor has three different parameters: p, i, and d. In the example below, a new PIDController is being constructed with pid values of 0.2, 0, 0.1.

```
pidController = new PIDController(0.2, 0, 0.1);
```

After initializing the PID controller, there are several configurations that can be placed on the loop. If you want to limit the range that the setpoint can be set to, use the function setInputRange(). The code below restricts the input setpoint to be between 0 and 360.

```
pidController.setInputRange(0, 360);
```

In situations where the robot may be taking the long route to get to where it needs to be (ex: 270 left instead of 90 right), the setContinuousInputRange() function can be useful. By enabling this setting, the min and max input range is considered the same value, and the shortest route to they desired position is calculated based on that.

```
pidController.setContinuousInputRange(0, 360);
```

To set the setpoint of the PID loop, use the setSetpoint function. The code below sets the setpoint to 180.

```
pidController.setSetpoint(180);
```

Use the calculate() function to get that speed, on a scale from -1.0 to 1.0 (or the output range if that is configured) The calculate function takes a parameter that is your current position and calculates PID based on that. The code below calcualtes the value based on the current gyroValue.

```
double output = pidController.calculate(gyroValue);
```

6.4.2 Programming Talon PID

When using PID with a talon and an encoder, using the ctre talon classes should be used instead of the PIDController classes. Instead of an object being created, the PID configurations should be set through the talon. TO set the P, I, and D values, use the config_kP (use kI or kD for I and D) to set the PID values. The code below configures the P = 0.2, I = 0.01, and D = 0.1.

```
talon.config_kP(0.2);
talon.config_kI(0.01);
talon.config_kD(0.1);
```

If you want to cap the maximum output of the PID loop, use the configClosedLoopPeakOutput() function instead. The code below sets the maximum closed loop output to 0.3.

```
talon.configClosedLoopPeakOutput(0.3);
```

This function sets the range to between -0.3 to 0.3 (you cannot make the minimum different than the maximum, although that is generally not necessary.) The talon has two main PID modes to be used: Position and Velocity. The position mode sets the talon to go to a certain position (in ticks), and the velocity mode sets the wheels to go at a certain velocity (in ticks / 100ms).

```
talon.set(ControlMode.Position, 1000);
talon.set(ControlMode.Velocity, 200);
```

The position code sets the talon to move to a position where the encoder reads 1000. The velocity code sets the talon to go at a speed where it travels at 200 ticks / 100ms. The TicksPerInch value (read Encoders to get more information) can be used to track the position in inches or in inches per second, so the input can be more realistic to the user.

Ramping position

It is important to ramp your value, since if you try to set PID to go directly to the position you are attempting to reach, the PID can go out of control. There are several different ways to ramp your position, but one way would be to set your temporary setpoint to your current position plus a certain ramp rate until you get your position within the ramp rate.

6.4.3 Programming Spark PID

To program Spark PID, use the getPIDController() function in the CANSparkMax to access the PIDController.

```
spark.getPIDController();
```

To set the PID Values, use the setP(), setI(), and setD() functions. The code below sets the PID values to 0.2, 0, 1

```
spark.getPIDController().setP(0.2);
spark.getPIDController().setI(0);
spark.getPIDController().setD(1);
```

To set a maximum output range, use the setOutputRange() function. The code below sets the output range from -0.5 to 0.5.

```
spark.getPIDController().setOutputRange(-0.5, 0.5);
```

To

6.4.4 Tuning PID

Tuning PID, while annoying, can lead to incredible accuracy if done well.

What P, I, and D control

P, I, and D each control the robot in different ways. P is a simple constant proportion: the bigger to speed, the faster the robot moves. I causes an increase of speed over time: the longer the loop is running, the faster the robot will go. A little bit of I can be useful to make a system run faster, but too much can cause the system to overshoot and go haywire. Lastly, D slows down the system when the robot is going is moving too fast. However, too much of D can cause the robot to jitter.

Ziegler-Nichols

The Ziegler-Nichols method is a very mathematical method to tune PID. First, set the P value at an arbitrarily low number, and I and D to 0. Then increase the P value until you have a stable oscillation (rocks back and forth around setpoint). If the P value is two low, it will undershoot. If the P value is too high, then the robot will have unstable oscillation (each rock goes farther and farther away from the setp int. After getting this value, set a variable called K to that value. Afterwards, time the amount of time it takes for one oscillation and label that variable T. Then, follow these formulas to determine your P, I, and D values.

$$P = K/2$$

$$I = 1.2K/T$$

$$D = 3KT/40$$

These values usually will not work first try, but based on what P, I, and D do, you can increase or reduce values to tune the control system.

Experimental Tuning

If you are in a time crunch and do not have the time to do the whole Ziegler-Nichols method, a quick, but not quite as precise, method can be used. Similar to Ziegler-Nichols, find the P value where steady oscillation exists. Afterwards, set your D value to by 10x of your P. Afterwards, you can tune your P and D values according to how the system behaves. This method may run slower because it avoids using I altogether, but it can be useful in a time crunch.

Tuning Velocity PID

For Velocity PID, using feed forward is a much better approach, rather than having your PID loop control acceleration. To tune Velocity PID, find the maximum speed of the mechanism, and set your feed forward constant to be 1 / maxSpeed. Then run your mechanism by multiplying your desired speed by 1 / maxSpeed. This will produce a roughly correct value. Then, incorporate P until it corrects error properly (it goes to the right value but doesn't over and under shoot a lot to get there). If needed, some D can also be added if you find that your P is over correcting a lot instead of reducing P. If you are using Velocity PID with Feed Forward, **DO NOT use I**.

7 Controllers

7.1 What is a controller

A controller is something that takes in user input through driver station and uses that to control the robot. Examples include joysticks and xbox controllers.

7.2 Digital Input

A digital input returns either true or false, such as a button press.

7.2.1 Get Button

any getButton__() function returns true if the button is pressed, and false if it is not pressed.

7.2.2 Get Button Pressed

any getButton__Pressed() function will return true if the button goes from not pressed to pressed, and will otherwise return false. This function will return true only one time when the button is pressed, while getButton__() will continuously return true while it is being held down.

7.2.3 Get Button Released

any getButton__Released() function will return true only when the button transitions from being pressed to being not pressed.

7.3 Analog Input

An analog input returns some number value, which can be used to determine more detailed control over the robot, rather than a simple yes or no. An example of an analog input is any joystick. Analog inputs are generally on a scale of -1.0 to 1.0.

7.4 The Joystick Class

The Joystick Class has all the controls needed for a Logitech Attack 3 Joystick, which is just a standard gaming joystick.

7.4.1 Initializing the Joystick

To use the joystick, a new Joystick object must be created

```
Joystick joystick;
```

The joystick constructor has one parameter for ID, which can be configured in driver station. The code below configures a joystick with id 0

```
joystick = new Joystick(0);
```

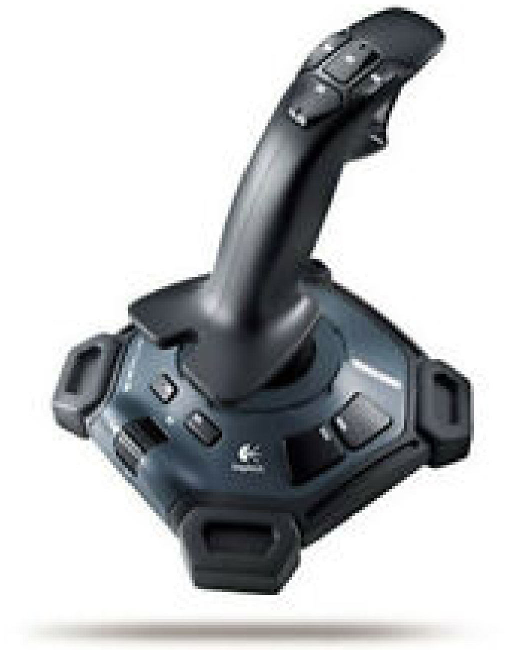

Figure 8: A picture of the logitech attack 3

7.4.2 Digital Input

The joystick has a trigger and 10 other buttons labeled from button 2 - 11. To get the trigger, use:

```
joystick.getTrigger();
joystick.getTriggerPressed();
joystick.getTriggerReleased();
```

To get any of the buttons, use the getRawButton() function. The code below gets the value of button 2

```
joystick.getRawButton(2);
joystick.getRawButtonPressed(2);
joystick.getRawButtonReleased(2);
```

7.4.3 Analog Input

The joystick class has two analog inputs: the x-axis and the y-axis of the joystick. Both of them go between -1.0 to 1.0. The code below gets the value of the x and y-axis

```
joystick.getX();
joystick.getY();
```

7.5 The XboxController Class

The Xbox Controller class is a class built for a standard Xbox Controller.

Figure 9: A Xbox Controller

7.5.1 Initializing the Joystick

Like the joystick, a XboxController object must be created.

```
XboxController controller;
```

The xboxcontroller constructor also has one parameter for ID. The code below configures a xboxcontroller with id 0.

```
XboxController = new XboxController(0);
```

7.5.2 Digital Input

The Xbox Controller has many sources of digital inputs, including both A, B, X, Y buttons, bumpers, and the D-pad. The code below checks for the A, B, X, and Y buttons.

```
controller.getAButton();
controller.getBButton();
controller.getXButton();
controller.getYButton();
```

Both bumpers use the getBumper() function, so in order to differentiate between left and right, you have an input of left or right in the functions. The code below shows the right way to get the left and right bumpers.

```
controller.getBumper(GenericHID.Hand.kLeft);
controller.getBumper(GenericHID.Hand.kRight);
```

In order to get the D-pad value, the getPOV() function returns an integer of the degrees the D-pad is facing (0 for up, 90 for right, etc.) The function returns -1 if none of the D-pad is pressed.

```
controller.getPOV();
```

7.5.3 Analog Input

Since the Xbox Controller has two joysticks, the same input to determine left or right bumper is also used to determine the joystick. Each joystick has an X and Y value between -1.0 to 1.0

```
controller.getX(GenericHID.Hand.kLeft);
controller.getX(GenericHID.Hand.kRight);
```

The Xbox Controller also has two triggers, each goign from a value between 0 and 1.0

```
controller.getTriggerAxis(GenericHID.Hand.kLeft);
controller.getTriggerAxis(GenericHID.Hand.kRight);
```

7.6 The GenericHID Class

Both the Joystick and the XboxController Class are based on the GenericHID Class. This is basically the master controller class. If you want to create a new controller class for a type of controller a class doesn't exist for, use this class. The class has five main functions

```
getRawAxis(int id) //gets the value of the axis with
    that id
getRawButton(int id) //gets the value of button with
    that id
getRawButtonPressed(int id) //see Digital Input for
    difference
getRawButtonReleased(int id) //see Digital Input for
    difference
getPOV() //see XboxController for more info
```

8 Pneumatics

8.1 What is pneumatics

Pneumatics is a control system that uses pressurized air to control pistons, instead of using motors. While a pneumatics system comprises of many different components, the only components that need to be programmed are the compresser and solenoids.

8.2 The Compressor Class

8.2.1 What is a compressor

A compressor is what pressurizes all of the air that is used in pneumatics system.

Figure 10: A picture of a compressor

8.2.2 Programming a compressor

In order to run the compressor, a new compressor object must be declared.

```
Compressor compressor;
```

The Compressor constructor takes in no parameters, as only one compressor is allowed on each robot

```
compressor = new Compressor();
```

To start the compressor, use the start() function

```
compressor.start();
```

This is usually run in robotInit() (see The Robot Class for more info). In order to stop the compressor, use the stop() function.

```
compressor.stop();
```

If you find that the compressor is losing PSI, try using the setClosedLoopControl() function.

```
compressor.setClosedLoopControl(true);
```

This is not a substitute for fixing a pnuematics system. This function, while useful, will not completely nullify a leak, and it is not advised to start pressurizing in the middle of a match, which is what this function will do.

8.3 Solenoids

8.3.1 What is a solenoid

A solenoid is what controls a piston. A single solenoid can push a piston in one direction, while a double solenoid can control a piston to go in both directions. On our team, we mostly use double solenoids.

8.3.2 Programming single solenoids

To use a single solenoid, a Solenoid object must be created.

```
Solenoid solenoid;
```

The Solenoid constructor has one parameter for the solenoid channel (aka the ID). The code below creates a new solenoid with channel set as 0.

```
solenoid = new Solenoid(0);
```

In order to actuate the solenoid, use the set function.

```
solenoid.set(true);
```

Setting the solenoid to false does not put pressure in the other direction. Since it is a single solenoid, it either applies air pressure in one direction or does not apply air pressure at all.

8.3.3 Programming double solenoids

To use a double solenoid, a DoubleSolenoid object must be created.

```
DoubleSolenoid doubleSolenoid;
```

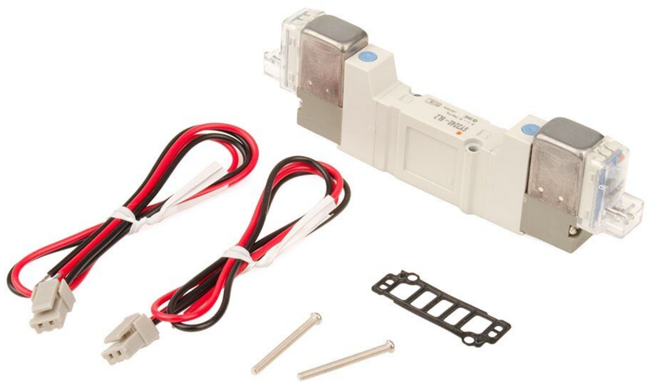

Figure 11: A double solenoid

The double solenoid constructor has two parameters, both being the two channels the solenoid uses (1 channel for each direction it applies air pressure). The code below creates double solenoids with channels 0 and 1.

```
doubleSolenoid = new DoubleSolenoid(0, 1);
```

In order to set the direction of the solenoid, use the set() function.

```
doubleSolenoid.set(DoubleSolenoid.Value.kForward);
```

The code above sets the solenoid the apply air pressure in the forward channel direction. To apply air pressure in the reverse channel direction (generally the opposite direction), use the kReverse enum.

```
doubleSolenoid.set(DoubleSolenoid.Value.kReverse);
```

If you want to set the solenoid to not apply air pressure, use the kOff enum.

```
doubleSolenoid.set(DoubleSolenoid.Value.kOff);
```

9 The Robot Class

9.1 What is the Robot Class

The Robot Class is the main file that your code is setup in. It contains various functions that allow for the control of the robot.

9.1.1 Timed Robot

Timed Robot is the recommended class for most projects. It contains a changeable loop time, as well as initialize and periodic functions for all of the different modes.

9.1.2 Iterative Robot

Iterative Robot is an older version of Timed Robot that is now deprecated. Timed Robot is recommended over this.

9.2 Robot

9.2.1 Robot Init

The function robotInit() is where all of the hardware initialization should be located. It is also where all subsystems should be initialized (see Command Based Programming for more info).

9.2.2 Robot Periodic

robotPeriodic() is where anything that should be running regardless of which mode the robot is in is ran. Usually, only Scheduler is run here (see Command Based Programming for more info)

9.3 Autonomous

The Autonomous functions run for during the first 15 seconds of a match, and usually ban driver control (exception of Deep Space). **With the exception of Deep Space, any controller input will return false or 0 in these functions in most years**.

9.3.1 Autonomous Init

autonomousInit() runs once right after the match starts. Usually, any commands or command groups specific to the autonomous period get initialized here.

9.3.2 Autonomous Periodic

autonomousPeriodic() runs continuously during the autonomous period of a match. Generally, nothing is needed to be run here.

9.4 Teleop

The Teleop functions run from after Autonomous ends to the end of the match.

9.4.1 Teleop Init

teleopInit() runs once after the autonomous periods end. Generally, any autonomous commands get cancelled here if they are not already finished (see Command Based Programming for more info)

9.4.2 Teleop Periodic

teleopPeriodic() runs continously during the entire match, except for the first 15 seconds. If you are not using Command Based Programming, this is where your controls should be setup.

9.5 Test

Test is not run during a match, but these functions are good for seeing is pieces of code will work without editing your main code. **Commands do not work in test** (see 10.4 - 10.6 for more info).

9.5.1 Test Init

testInit() runs once every time the robot is enabled in test mode. Can also be used as an area to store reset encoder functions for before a match starts if necessary.

9.5.2 Test Periodic

testPeriodic() runs continously while the robot is enabled on test mode.

9.6 Disabled

Disabled runs code while the robot is disabled. **This does NOT allow you to actually move anything while disabled**.

9.6.1 Disabled Init

disabledInit() runs once every time the robot is disabled. Can be useful if you want to brake the robot (switch motors from coast to brake) or just stopping anything in general.

9.6.2 Disabled Periodic

disabledPeriodic() can be used if you want to constantly update smart dashboard (like to see if the robot is still moving) only when disabled.

10 Command Based Programming

10.1 What is Command Based Programming

Command Based Programming is a structure of programming that organizes code to run based on loop times and deals with running multiple tasks at the same time for you. It is recommended to create any complex code using Command Based Programming.

10.1.1 Parts of Command Based Programming

Command Based Programming involves 3 main parts: subsystem classes, command classes, and the OI class.

Each subsystem class is created for any one set of motor / pneumatics controllers.

Each command class is designated to run on a certain subsystem class, and they execute tasks using functions written in the subsystem class.

Lastly, there is the OI class, which initializes a controllers attached to driver station and sets up when commands should be executed based on controller input.

10.2 Subsystems

10.2.1 How to differentiate subsystems

Before programming any subsystem, you need to know how to determine the amount of subsystems needed in the code, which is more complicated than it might seem at first. Remember, each subsystem runs on its own thread, so any two commands that want to be run simultaneously should be on different subsystems.

Lets use an ball intake as an example. While this constitutes as one mechanical subsystem, it may not be the same in code. An intake generally has two parts: rollers to take the ball in and motors to change the position of the intake. While this may not always be true, having the ability to intake a ball and change the position of the intake at the same time can be useful, so they would constitute as two separate subsystems.

Lets use a drive train as another example. While it is obvious the left and right side of the robot should both be able to move at the same time, it would still constitute as a subsystem most of the time (again, there can be exceptions to this). This is because there is generally no command that involves controlling one side and not the other, so having two separate subsystems is not necessary. While on the intake, there are situations where you would run the rollers and change the position either separately or simultaneously, which is why they are separate subsystems.

10.2.2 Creating a subsystem

To create a subsystem, a new class must be created that extends the SubsystemBase class, since the SubsystemBase class is an abstract class.

```
public class DriveTrain extends SubsystemBase {

}
```

10.2.3 Making a Singleton

Since each subsystem class that gets called only needs to be initialized once (I hope you don't have two different drive trains on the same robot), a singleton can be used. A singleton basically converts the class into a single instance, and in order to call any other functions in that class, you have to use the getInstance() method to access the one instance. The code below converts the DriveTrain class into a singleton

```
private static DriveTrain ourInstance = new
    DriveTrain();
public static DriveTrain getInstance() {
    return ourInstance;
}
private DriveTrain() {
    super("DriveTrain");
}
```

Since the constructor is made private, and the only instance of the class is within the DriveTrain class, only the getInstance() method can be used to get the only instance of the class. The super("DriveTrain") is calling the constructor from the subsystem class, and the parameter is just the name of the subsystem.

10.2.4 The Init Hardware function

To initialize all of the hardware used in the subsytem, create an initHardware() function and call that function in robotInit(). The code below initialized two talons. In this function, you can also configure the default command using the setDefaultCommand() function. A default command runs when no other command is assigned to run on that subsystem

```
WPI_TalonSRX leftTalon, rightTalon;
public void initHardware(){
    leftTalon = new WPI_TalonSRX(0);
    leftTalon.configFactoryDefault();
    leftTalon.setInverted(true);
    rightTalon = new WPI_TalonSRX(1);
    rightTalon.configFactoryDefault();
    setDefaultCommand(new TankDrive());
}
```

The default command is optional. If you do not want to set a default command, just don't write any setDefaultCommand() function.

10.2.5 The periodic function

The periodic function runs once per loop cycle after each command has been run. Any code that needs to ALWAYS be run, REGARDLESS of any commands should be put here. Often times, it is not necessary although it can be useful at times. The code below ensures that the left side of the robot will never move (not an example of real use in main code, but can be useful for debugging purposes)

```
@Override
public void periodic(){
    leftTalon.set(ControlMode.PercentOutput, 0);
}
```

10.3 The OI Class

The OI class is used to actually control the robot through driver station through commands. There are two ways to control commands: either use the button class for digital inputs or accessing joystick values for analog input.

10.3.1 Configuring the OI class

Since there is only one OI class that is needed, the OI class is turned into a singleton like each of the subsystem classes are.

```
private static OI ourInstance = new OI();
public static OI getInstance() {
    return ourInstance;
}
private OI() {

}
```

Inside the OI class, each of the controllers should be declared privately and accessed through a function. The example below as a XboxController and a Joystick

```
XboxController controller;
Joystick joystick;
public XboxController getController(){
    if(controller == null){
        controller = new XboxController(0);
    }
    return controller;
}
public Joystick getJoystick(){
    if(joystick == null){
        joystick = new Joystick(1);
    }
    return joystick;
}
```

In the code below, the joysticks get initialized after the first time that they are called.

10.3.2 Digital Inputs

In order to use digital inputs, a Button object must be created. The code below creates a button object with the xbox controller A button.

```
Button aButton = new Button() {
    @Override
    public boolean get() {
        return getContoller().getAButton();
    }
};
```

After creating a button, you can use one of the following functions to control when a command executes.

```
whenPressed();
whenReleased();
whileHeld();
```

These functions run either when a button is pressed, released, or while held (look at digital inputs in controllers for more info). The code below will run the GearShift command when the a button is pressed.

```
aButton.whenPressed(new GearShift());
```

10.3.3 Analog Inputs

To use analog inputs, use the OI class functions that access the controllers and use those values in the command itself.

10.4 Commands

Each command has various different parts that allow for control of the subsystem.

10.4.1 Creating a command

To create a command, a new class must be created that extends the CommandBase class, since the CommandBase class is an abstract class.

```
public class GearShift extends CommandBase {

}
```

10.4.2 The constructor

Each subsystem should have a constructor, and the constructor should have the following parts: naming the command using super(”Name”), parameters (if necessary), and the requires() function, which sets the command to run on the inputted subsystem. Below is an example constructor for Gear Shifting.

```
public GearShift(boolean shiftHigh){
    super("GearShift");
    this.shiftHigh = shiftHigh;
    addRequirements(Shifter.getInstance());
}
```

In this example, the constructor has a super to name the command, and the parameter value is set to a local variable, and the command is set to require the Shifter subsystem (Shifter.getInstance() gets the only instance of the Shifter subsystem).

10.4.3 The Initialize function

Every command can have the initialize() function, which runs one time after the command is started. Anything that needs to be run just once at the beginning of the command (variable initialization w/o parameters, solenoid shifting, etc.). In the code below the solenoid shifts based on the shiftHigh boolean.

```
@Override
protected void initialize(){
    if(shiftHigh){
        DriveTrain.getInstance().shiftHigh();
    } else {
        DriveTrain.getInstance().shiftLow();
    }
}
```

10.4.4 The Execute function

The execute() function runs once every loop cycle until either the command is finished or the isFinished() boolean returns true (see is finished function for more info). Anything that needs to be constantly run with just slight variation should be put in the execute function. Below is tank drive code put in the execute function for a tank drive command.

```
@Override
protected void execute(){
    double left = OI.getInstance().getController().
        getY(
        GenericHID.Hand.kLeft);
    double right = OI.getInstance().getController()
        .getY(
        GenericHID.Hand.kRight);
    DriveTrain.getInstance().tankDrive(left, right)
        ;
}
```

The code below sets tankDrive using the x and y values of the xbox controller. While the code may look a bit convoluted, it is really just taking advantage of the singleton system, and the code would involve a lot of static variables otherwise, which can get pretty messy. Also, the code demonstrates how to using analog input with commands and subsystems works.

10.4.5 The Is Finished function

This function returns a boolean to check if the execute function should end or not, and it is **mandatory** to have whenever the command class is extended. Think of the way isFinished() works like this:

```
while(!isFinished()){
    execute();
}
```

The code below is an example isFinished function for the tank drive command.

```
@Override
protected boolean isFinished(){
    double left = OI.getInstance().getController.
        getY(
        GenericHID.Hand.kLeft);
    double right = OI.getInstance().getController.
        getY(
        GenericHID.Hand.kLeft);
    return Math.abs(left) < 0.05 && Math.abs(right)
         < 0.05;
}
```

The code returns true only when both joysticks are under the threshold value, which is when the command is supposed to end. **Commands can simply return true or false.** If a command is set as the default command, it can return false and just get overridden by other commands (a TankDrive command may do this). A command can also just return true if nothing needs to be run in execute (such as a gear shift command).

10.4.6 The End function

The end() function runs once right after the execute function stops looping (aka isFinished returns true). This can be useful, especially for stopping motors after a command is finished or running another command that needs to always follow the current command. The code below is example code for a TankDrive commands end() function. The end function also takes a boolean parameter of interrupted, which can be used to cause different results if the command was finished or interrupted.

```
@Override
protected void end(boolean interrupted){
    DriveTrain.getInstance().tankDrive(0, 0);
}
```

This code sets the drive train to stop moving after the command finishes.

10.4.7 Instant Commands

Instant Commands can be created instead if you have just one function that needs to be run or your command needs to end immediately. To make an instant command, extend the class InstantCommand and super a runnable and a subsystem to run the command on. The code below is an example of an instant command to make a DriveTrain stop.

```
public class StopDriveTrain extends InstantCommand{
    public StopDriveTrain(){
        super( ()-> DriveTrain.getInstance().
            tankDrive(0, 0), DriveTrain.getInstance
            ());
    }
}
```

10.4.8 Run Commands

Run Commands can be used if a command needs to run just one function perpetually (or until interrupted) with no initialization. The make a RUn Command, extend the RunCommand class and super a runnable and a subsystem to run on. The code below performs tank drive using a Run Command instead of a regular command.

```
public class TankDrive extends RunCommand {
    public TankDrive(){
        super( ()-> DriveTrain.getInstance().
            tankDrive(OI.getInstance().
            getXboxController().getY(Hand.kLeft),
            OI.getInstance().getXboxController().
            getY(Hand.kRight)), DriveTrain.
            getInstance());
    }
}
```

10.4.9 Wait Commands

Wait Commands can be used if you just need a command to wait for x amount of time. It is most useful in command groups (see 10.5 for more info). The code below creates a new wait command for 5 seconds.

```
Command exampleWait = new WaitCommand(5);
```

10.4.10 Wait Until Commands

Wait Until Commands return false until a certain condition is met, and do nothing else. Like Wait Commands, they are most useful in command groups (see 10.5 for more info). The code below creates a Wait Until Command with the condition to return true if a switch is pressed.

```
Command exampleWaitUntil = new WaitUntilComand(()->
    switch.get());
```

10.4.11 Conditional Commands

Conditional Commands run one of two commands based on a certain condition. It can be especially useful if multiple of the command functions change depending if the condition is true or false rather than one line (then an if statement inside a regular command is better) or if one ore both options for commands are being called outside of the conditional command. The code below creates a new command that decides to shoot a ball at high or low rpm based on a distance. If the distance is greater than 10, it shoots at a high rpm; otherwise, it shoots at a low rpm.

```java
public class Shoot extends ConditionalCommand {
    public Shoot(){
        super(()->distance > 10, new ShootHigh(),
            new ShootLow());
    }
}
```

10.5 Command Groups

Command groups allow to run commands either sequentially or parallel which provide the ability to execute more complicated macros.

10.5.1 Sequential Command Groups

Sequential Command Groups run each command added one after another. Unless you want your last command to run perpetually, **do NOT make your commands never ending**. To create a Sequential Command Group, have your class extend SequentialCommandGroup and use the addCommands() function to add your commands. The code below sets a macro up for a robot to drive in a square.

```java
public class SquareDrive extends
    SequentialCommandGroup {
    public SquareDrive(){
        for(int i = 0; i < 4; i++){
            addCommands(new MoveForward(10), new
                Turn(90));
        }
    }
}
```

10.5.2 Parallel Command Groups

Parallel Command Groups run each of the commands given parallel and do not end until all of the commands have finished. **do NOT schedule 2 commands requiring the same subsystem in the same parallel command group**. The Syntax for Parallel Command Groups is the same as sequential command groups, except ParallelCommandGroup is extended instead. Be careful with the isFinished() statements as all the commands need to end for the ParallelCommandGroup to end. The code below drives the robot and shoots a ball at the same time.

```
public class ShootDrive extends
    ParallelCommandGroup {
    public ShootDrive(){
        addCommands(new ShootBall(3000), new
            DriveForward(100));
    }
}
```

10.5.3 Parallel Race Groups

Parallel Race Groups are the same as Parallel Command Groups, but they end as soon as one of the commands finishes. Below there is a WaitCommand and a WaitUntilCommand that will end once either one returns true. This can be useful if you want a condition to be true but can only wait x amount of time. In this example, the wait time is 5 seconds and the condition is if a switch is triggered.

```
public class DoubleWait extends ParallelRaceGroup {
    public ShootFeed(){
        addCommands(new WaitCommand(5), new
            WaitUntilCommand(()->randomSwitch.get()
            );
    }
}
```

10.5.4 Parallel Deadline Groups

Parallel Deadline Groups are an intermediate between parallel race groups and parallel command groups. All the commands run parallel, but the whole command groups ends once one specified command ends. The code below runs a shoot ball and feed ball command parallel (assume they are different subsystems). Here, the feed ball command wouldn't no to end, but it ends because the shoot ball command knows when to end.

```
public class ShootFeed extends
    ParallelDeadlineGroup {
    public ShootFeed(){
        addCommands(new ShootBall(3000), new
            FeedBall());
    }
}
```

10.5.5 More Complex Command Groups

When dealing with more complicated command groups, it is possible that a combination of the four types of command groups is required. Since all command groups extend the Command class, you can input command groups with command groups. The code below will ramp up a shooter to speed wait for one of two conditions (a shooter gets up to speed or 2 seconds pass) and then shoot a ball.

```
public class Shoot extends SequentialCommandGroup {
    public Shoot(){
        addCommands(new RampUpShooter(),
            new ParallelRaceGroup(
                new WaitUntilCommand(()->Shooter.
                    getInstance().getError < 5),
               new WaitCommand(2)),
            new ReleaseBall());
    }
}
```

There are also functions called sequential() and parallel() that create a sequential or parallel command group withe parameters being an array of commands. It is the same as using new SequentialCommandGroup or new ParallelCommandGroup.

10.6 Command Scheduler

The Scheduler class is a class provided in the frc libraries that controls when commands actually get executed. The details on how that works are explained in official frc docs, but basically, the command CommandScheduler.getInstance().run() needs to be called in all periodic functions if commands are being used in those functions.

10.6.1 Running Command Scheduler

The Command Scheduler be called inside robotPeriodic() if its required in autonomous and teleop. **Commands do not run in test**. The code below demonstrates this:

```
@Override
public void robotPeriodic() {
    CommmandScheduler.getInstance().run();
}
```

10.6.2 Scheduling Commands

If you need to schedule a command that is not being scheduled through other means (default command, OI Buttons, etc.), the CommandScheduler class can also be used for that. the schedule() function with schedule any command. The code below will schedule a Command called autonCommand;

```
CommmandScheduler.getInstance().schedule(
    autonCommand);
```

10.6.3 Cancelling Commands

You can also cancel commands through command scheduler. The code below will cancel the same auton command.

```
CommmandScheduler.getInstance().cancel(autonCommand
    );
```

If you want to cancel all the commands, you can use the code below.

```
CommmandScheduler.getInstance().cancelAll();
```