Virtual Operator Framework for FRC / FTC Robots

Presented by Frank Larkin

Expectations

- We will try to…
  - ***Inspire you to want to experience the joy we can get from programming. This can be the most challenging and fun things you will ever do!***
  - Show you a easily understandable frame work to program a competitive FRC or an FTC robot that includes multiple modes of autonomous operation.
  - Show some of the differences between FRC and FTC.
  - Give simple examples for a Tank drive. (later lessons will cover other drive types)
  - Make available some complex code concepts based upon this method with more to come. All available at FMA+ Training
- We will not…
  - Teach you Java!!! Goto FMA+ Training and learn from Java Basics Lessons
  - Tell you how to load all the FRC/FTC development code.
  - Write your code for you. You will do the heavy lifting but use this as a guide.
  - Have all the answers. You must follow all the available documentation on the APIs and hardware details.
- Remember…
  - This is only a ***suggested*** framework that has worked well for over 20 years. It has been adapted as things change. You carry it forward.
  - Make mistakes, learn a lot, stick with it, share what you learn and enjoy the ride.

Design Philosophy

- Major custom classes represent macro functionality.
  - Like: Inputs, Sensors, Tower, EndEffector, and RobotBase
    - Call them what you like just so the idea is clear to you and your team.
  - Just from the design, everyone knows where to look for any capability.
- Physically and Programmatically place components in the custom classes where they make sense to you and your team.
  - Example: Camera on the tower is defined in its own class or in the Tower Class
  - No real need to put it on Sensors class. (Sensors may be an out moded idea.)
- Cook input values as you gather them and output values as you use them
  - Manipulate (aka cook) them so they make sense in the real world, in your head and are not difficult to understand.
  - Example: Make it so a positive value of robot power indicates forward motion and negative means reverse no matter how you get it.
  - Makes reading code easy to follow and minimizes mistakes.
  - On site competition changes are easier and less error prone.
    (Yes Code Warrior, your team will expect miracles at the competitions!)

Design Philosophy

- Process in logical order through the classes on each pass
  - We read input values into public “fly by wire” FBW input variables.
  - FBW variable are shared with others classes to act on.
    - The is design simple yet flexible.
  - Classes downstream, using FBW values, can filter and modify the input control FBW variables as needed.
    - Think autonomous operation.
  - Eventually we tell the output classes to update FBW values to the outputs
    - Robot now moves
- ***We will not…***
  - Require getter and setter methods in these custom classes for handling access to variable values between these classes.
  - Java purists may not like this but history has shown this works best!

Design Philosophy (continued)

- Input type classes
  - Method **readValues()**; captures values into public **fly by wire** or **FBW** variables.
    - BTW: method readValues(); is a suggested method. Call it what you like.
  - FBW variables are ***only those that will be needed*** by other Custom classes to make decisions affecting their outputs.
- Input Examples
  - Inputs – Requests humans at the Driver Station
  - Sensors – (optional) gathers readings from several sensing classes or devices. Examples: Gyro, Camera, String Potentiometers
- Input/Output Classes may have input sources but ultimately update the output robot devices.
  - Inputs - Sensors like Integrated Motor Encoders, switches, analog potentiometers, various flavors
  - Output - Signals to motor controllers, servos, pneumatic components.
  - Examples: RobotBase, Tower, Tool or End Effector
  - Utilize **readValues();** method to read and set FBW values
  - Later in the dance, we call **update();** method to make final changes to the outputs.

Design Philosophy (continued)

- Utilize many non-customized classes
  - FRC: WPI Lib API (Application Program Interface) classes
  - FTC: Quallcom Lib API (Application Program Interface) classes
  - Home grown Configuration, Telemetry, ApplyPower, RampPower (more lessons to come)
  - Capability not custom to any FIRST program, year or challenge.
    - We expect these are traditionally designed with getters and setters
- Allow the Custom framework to be used year after year.
  - The flow stays basically the same but with new competition specific modifications.
  - Understanding is carried into following seasons by younger team members.

FRC Iterative Robot Model

- Use the WPI Lib (FRC) Iterative Robot Model
  - Allows for even a complex design to be easy to follow and understand.
- The Robot class is our starting point.
  - FRC: Robot class extends the TimedRobot WPI class.
    Screen shot from FRC Team 5407 Wolfpack Code

```
/**
 * The VM is configured to automatically run this class, and to call the
 * functions corresponding to each mode, as described in the TimedRobot
 * documentation. If you change the name of this class or the package after
 * creating this project, you must also update the build.gradle file in the
 * project.
 */
public class Robot extends TimedRobot {
```

Important: Notice is says you can change the name. If you do you are on your own!!!

- You instantiate all your Custom classes in the Robot class
  - Examples: Inputs, Sensors, Auton, RobotBase, Tower, Tool, EndEfector

FRC Iterative Robot Model

- FRC Modes
  - Disabled – All inputs work, no outputs. State when robot starts up.
  - Autonomous – human inputs zeroed out, other inputs, auton and outputs work.
  - Teleop – All inputs work, outputs work. Robot will move.
- Methods to implement the modes
  - disabledInit() – Called just before disabledPeroidic()
  - disabledPeriodic – Called every 200 ms in while mode.
  - teleopInit() – Called just before teleopPeriodic()
  - teleopPeriodic() – Called every 200 ms while in mode (live driving robot)
  - autonomousInit() – Called just before autonomousPeriodic mode
  - autonomousPeriodic() – Called every 200 ms while in mode
  - testInit() – Called just before testPeriodic()
  - testPeriodic() – Called every 200 ms while in mode.
- TimedRobot will call these methods
  - Make sure your case sensitive method names are spelled correctly!!

FTC OpMode Loop Robot Model

- The Robot class is our starting point.
  - The “Robot” class has the same name as the Program you call on the phone.

```
@TeleOp(name="FTC2019: MainCode", group="Iterative Teleop")
//@Autonomous(name="FTC2019: Autonomous", group="Iterative Autonomous")
//@Disabled
public class FTC2019IterativeOpMode extends OpMode {
```

Screen shot from 2019 FTC Team 9997 Wolfpack Code

- You instantiate all your Custom classes in the Robot class
  - Examples: Inputs, Sensors, Auton, RobotBase, Tower, Tool, EndEfector
- OpMode will call these methods
  - init_loop - method called periodically when operator hits Init on phone
  - loop - method called periodically when operator hits Play on phone

Note: Make sure these case sensitive method names are spelled correctly!!

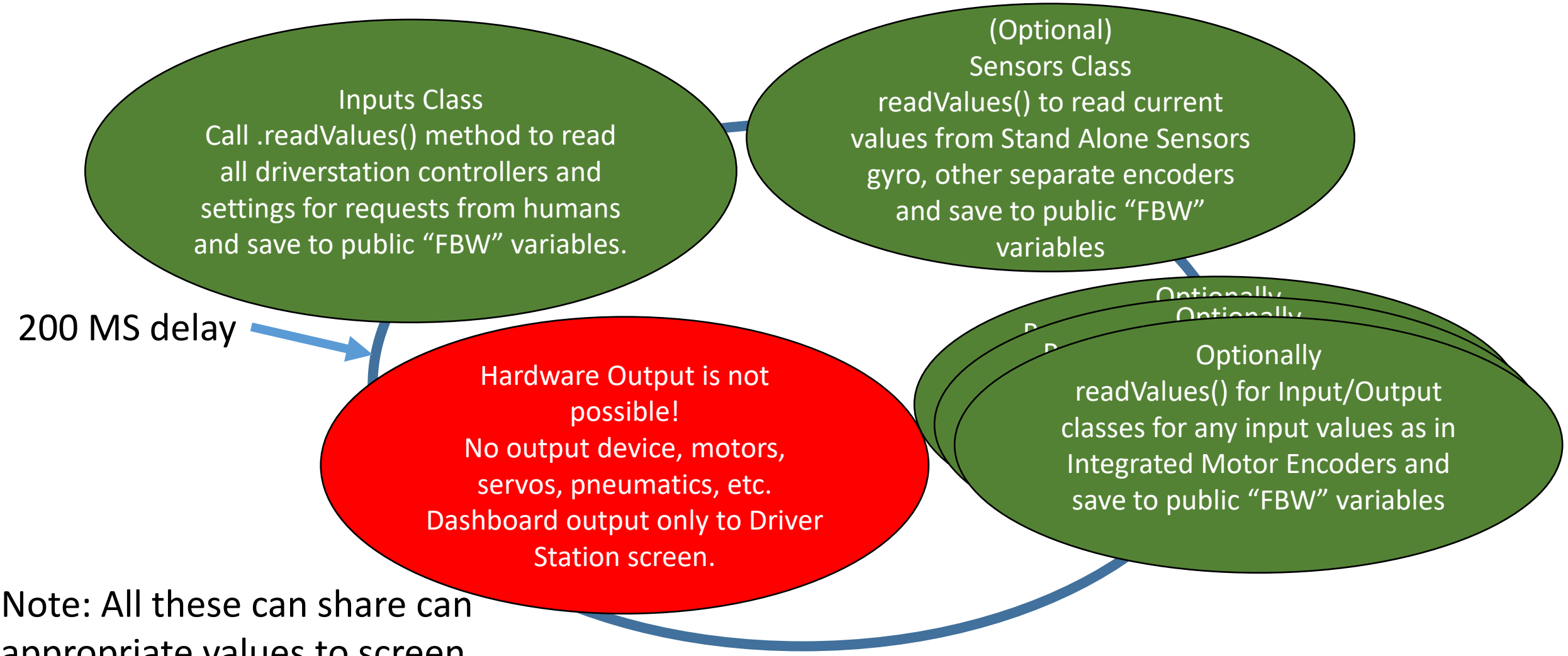
FTC OpMode Loop Robot Model

- FTC has modes of operation @TeleOp and @Autonomous
  - Heading above the Robot class determines what mode

```
@TeleOp(name="FTC2019: MainCode", group="Iterative Teleop")
//@Autonomous(name="FTC2019: Autonomous", group="Iterative Autonomous")
//@Disabled
public class FTC2019IterativeOpMode extends OpMode {
```

  - This framework allows both modes from 1 program.
- As you near the competition…
  - Copy the Robot class and give it new Class and file name.
    - Suggest you include Auton in it
  - Change the headings to indicate Autonomous.
  - In the class tell it to call the autonomous code.
  - Done…

FRC: disabledPeriodic(); FTC: init_loop()



Note: All these can share can appropriate values to screen so you can verify they are working.

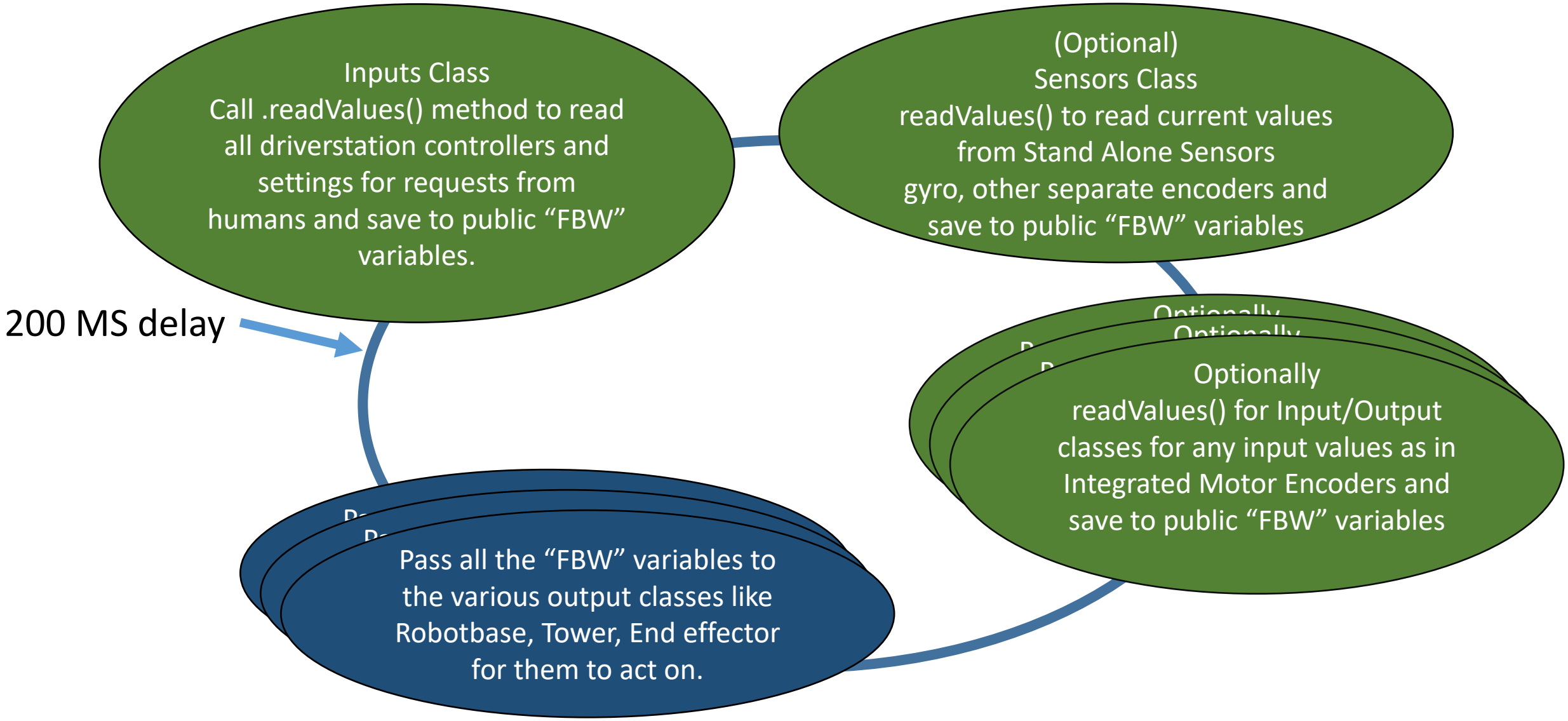
FRC: robot.disabledPeriodic(); FTC: init_loop();

```
public void disabledPeriodic(){
        inputs.readInputs();   // class sets FBW input type variables
        sensors.readInputs(); // class sets FBW input type variables
        tower.readInputs();   // class sets FBW input type variables.
        robotbase.readInputs();  // class sets FBW input type variables.

        // local robot method directly reads inputs FBW variables and allows
        // operator to set what auton program to run or auton delay start.
        // SmartDashboard on driver station shows the settings.
        updateAutonMode( inputs ); // Example Simple Idea
}
```

- Allow users to…
  - See sensor and other values on their displays in their pit and before round.
  - Change settings before round starts like Auton program or Auton Delay Start.

FRC: robot.teleopPeriodic(); FTC: loop();



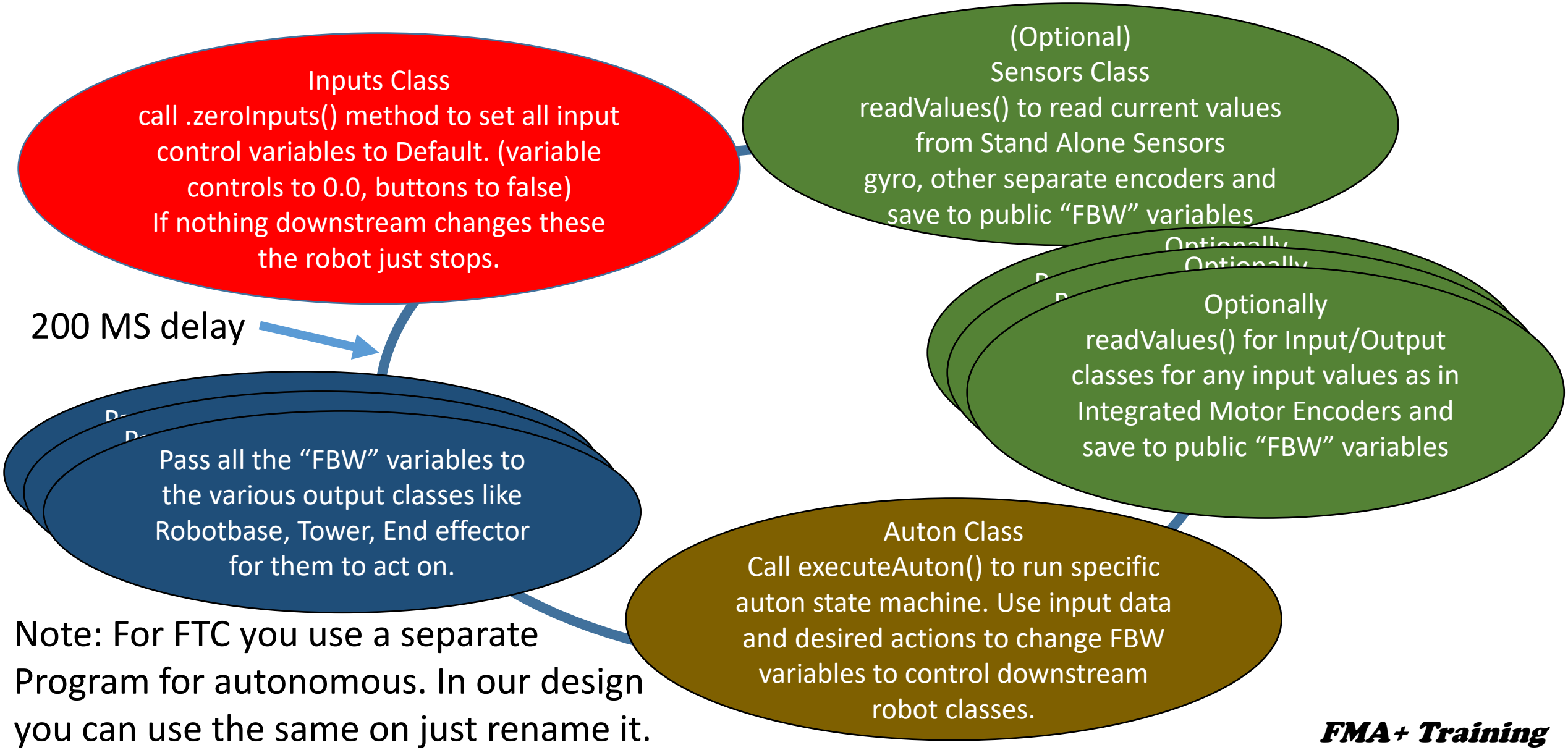
FRC: teleopPeriodic(); FTC: loop();

```
public void teleopPeriodic(){
        inputs.readValues();   // class sets FBW variables with input values.
        sensors.readValues(); // class sets FBW variables with input values.
        tower.readValues();   // class sets FBW variables with input values.
        robotbase.readValues();  // class sets FBW variables with input values.

        tower.update();       // class uses FBW variables to update the outputs.
        robotbase.update();   // class uses FBW variables to update the outputs.
}
```

- TeleopPeriodic
  - Flow is very easy to follow. We can easily see what is being executed when.
  - If you have a problem with drives you know where to look.

FRC: robot.autonomousPeriodic() workflow



Note: For FTC you use a separate Program for autonomous. In our design you can use the same on just rename it.

FRC: autonomousPeriodic(); FTC: loop(); // Auton

```
public void autonomousPeriodic(){
      inputs.zeroInputs();   // Zero out human FBW inputs but not settings.
      sensors.readValues(); // class sets FBW variables with input values.
      tower.readValues();   // class sets FBW variables with input values.
      robotbase.readValuse();  // class sets FBW variables with input values.

      auton.executeAuton(iAutonProgram); // What state machine to run.

      tower.update();     // class uses FBW variables to update the outputs.
      robotbase.update();  // class uses FBW variables to update the outputs.
}
```

- Auton Class (Details Later)
  - Allows you to create many autonomous programs you can call up by changing the number passed to the auton.executeAuton() method.
  - Auton uses the FBW variables to help guide autonomous operations.
  - Modifies Input Control FBW variables later used by Output classes!

Review

- Define multiple custom classes that represent the parts of the “Robot”
- We use Fly-By-Wire variables to gather sensor data and control requests.
  - Control variables are used to communicate our desires to outputs.
  - Sensors variables tell us what is going on.
- Input type classes, gather information
  - From Driver station or sensors into FWB variables.
- Input/Output classes gather sensor FBW variables but also have output capability to be used later.
- Auton allows us to use all FBW values
  - Create Steps that change human inputs FBW values before output classes use them. (called virtual operator)
  - These follow in order. As one step completes and other starts. You control order in code.
  - When all steps eventually get to the end indicating Auton is done. Robot stops.
- Finally Input/Output classes use Input FBW values and various sensor FBW values to decide how to set control values to the Outputs.

Review: Robot Class

- You always start in FRC: Disabled or FTC: init_loop mode.
  - Software can read from the input type classes but cannot output to motors.
  - Can be used to set variables in code like what auton mode to run. (more to follow)
- FRC: Before switching to a mode the Init version will be called
  - disabledInit is called right before disabledPeriodic
  - This allows you to set any pre-mode conditions
- FTC: Robot is controlled by what you select before a round starts.
  - Press Init: Will start in **init_loop()** until you hit Play, then control goes to **loop();**
- In FRC, robot controlled by the Field Management System
  - Typical round: Disabled, Autonomous, Disabled, Teleop

Virtual Operator Framework Custom Classes And Cook Book

Just a taste of what you can do…

Robot Class

- You always start in FRC: Disabled or FTC: init_loop mode.
  - Software can read from controllers but cannot output to motors.
  - Can be used to set variables in code like what auton mode to run. (more to follow)
- FRC: Before switching to a mode the Init version will be called
  - disabledInit is called right before disabledPeriodic
  - This allows you to set any pre-mode conditions
- In FRC, robot controlled by the Field Management System
  - Typical round: Disabled, Autonomous, Disabled, Teleop
- In FTC, robot is controlled by what you select before a round starts.
  - Will start in **init_loop()** until you hit Play, then control goes to **loop();**

RobotPorts Class (FRC only, suggested)

- Class where the robot ports are defined.
  - Class is visible to all others so they can see the definitions.
  - All Vars must be **static final** so no need to have constructor.
  - FTC Has HardwareMap for this

```
class RobotPorts {


    // Driver Station USB Ports
    public static final int DriverStation_DriverXBox_UsbID          = 1;
    public static final int DriverStation_OperatorXBox_UsbID        = 2;


    // CAN Bus Ids
    public static final int Base_PCN_CanID              = 10;

    // PWM Bus Ids
    public static final int Base_RightFrontDrive_PwmID = 1;
    public static final int Base_RightRearDrive_PwmID  = 2;
    public static final int Base_LeftFrontDrive_PwmID  = 3;
    public static final int Base_LeftRearDrive_PwmID   = 4;

}
```

Inputs Class

- Everything coming from Driver Station
  - Joysticks, GamePads Buttons, Preset Driver Station values
- Joystick: (FRC) Has different types touchy things to allow CBU (people) to tell robot what to do.
  - On / Off: Called a button.
    - Return boolean values (true, false)
  - Variable Thingy: Allows CBU to set a range of values. Power (y), Turn (z), Crab (x)
    - Returns a floating point (decimal) from -1.0 to 1.0
    - Problem: Full forward, y bushed forward is -1.0.
    - Fix it with inversion. (don' freak out yet! ☹)

Logitech Extreme 3D Pro Joystick
FRC: Joystick Class

Input Class

- Example Declaration

```
class Inputs{

    // Decalre my input devices
    public XboxController padDriver     = null; // Driver GamePad
    public XboxController padOperator   = null; // Operator GamePad
    //public Gamepad padDriver      = null; // FTC Version Driver GamePad
    //public Gamepad padOperator    = null; // FTC Version Operator GamePad


  // FRC Definitions
  public Inputs(){                                                  // Class Constructor
    padDriver                   = new XboxController(RobotPorts.DriverStation_DriverXBox_UsbID);
    padOperator                 = new XboxController(RobotPorts.DriverStation_OperatorXBox_UsbID);
  }

  // FTC Definitions
  public Inputs(HardwareMap mapHwr, XboxController  gamepad1, XboxController gamepad2){       // Class Constructor
    // Passing HardwareMap here my not be necessary. Learning FTC so maybe can be eliminated.
    bIsFTC = true;
    padDriver                   = gamepad1;     // in FTC GamePad 1 and 2 are available in the Robot class and ref
    padOperator                 = gamepad2;     // we are using XboxController class here to make it work easier.
  }
```

FTC: Gamepad class
FRC: XboxController class

Input Devices: (cooking your inputs)

- Two sets of variable control on Controller or Gamepad
  - Analog Controls
    - These are “handed” as in Left/Right.
    - Two Joysticks, **X**/**Y** Axis, double -1.0 to 1.0
    - Front lower **Trigger** buttons. double 0.0 to 1.0

FTC: Gamepad class
FRC: XboxController class

```
// HAND is a Java  enumerator used to tell system which stick to read from.
dDriverLeftPower      = padDriver.getY(HAND.kLeft);      // FRC version
dDriverRightPower     = padDriver.right_stick_y;         // FTC version
```

- Force Values to make sense to you…
  - Power Example: Push Y stick full forward gives you -1.0. We want forward to be a + value because it makes sense to humans, and reverse to be a - value.
  - invert it * -1.0 flips to 1.0 or just put – in front to invert. Works the same for FRC/FTC.

```
dDriverLeftPower      = padDriver.getY(HAND.kLeft) * -1.0;      // inverted
dDriverRightPower     = -padDriver.right_stick_y;               // inverted
```

Input Devices: (cooking your inputs to desensitize axis controls)

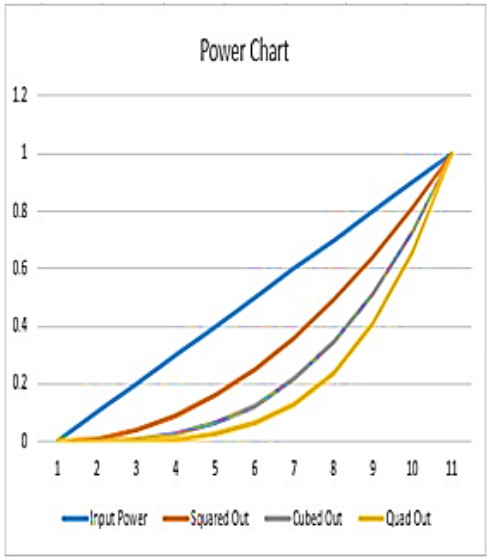
- Variable sticks can be very sensitive.
  - Robot makes quick move with very small deflection.
- Cook values to desensitize and give more low end control. Experiment with drivers
  - Use squaring, cubing and quading (I made quading up)
    - **Important: Must use care to preserve the sign aka direction! Yes you may not need it but I like to use Math.abs to help.**
    - Math.abs (absolute value) always returns positive.

FTC: Gamepad class
FRC: XboxController class

```
dDriverLeftPower = dDriverLeftPower *
                        Math.abs(dDriverLeftPower);        // square
dDriverLeftPower = dDriverLeftPower *
            Math.abs(dDriverLeftPower * dDriverLeftPower);   // cube
dDriverPower = dDriverLeftPower *  Math.abs(dDriverLeftPower *
                        dDriverLeftPower  * dDriverLeftPower);  // quad
```

Why not just raise to a power of 2, 3 or 4? Investigate and see….

Input Power	Squared Out	Cubed Out	Quad Out
0	0	0	0
0.1	0.01	0.001	0.0001
0.2	0.04	0.008	0.0016
0.3	0.09	0.027	0.0081
0.4	0.16	0.064	0.0256
0.5	0.25	0.125	0.0625
0.6	0.36	0.216	0.1296
0.7	0.49	0.343	0.2401
0.8	0.64	0.512	0.4096
0.9	0.81	0.729	0.6561
1	1	1	1



Input Devices: (cooking your inputs)

- Buttons: return a Boolean (true, false)
  - We say this is a “digital” input as there are only 2 values.

```
bTowerWinchUp        = padOperator.getXButton();  // FRC    (pad X button)
bUseGyroNavigation   = padDriver.Y;   // FTC                 (pad Y button)
                                                            // Not Y axis
```

FTC: Gamepad class
FRC: XboxController class

Note: As you can see above, FTC example does not use getters to pull the values from the gamepad controller class. You access all the variables directly.

Obviously the designers decided that it was better that way and less confusing to your programmers. We applaud that decision!

Sensors Class or Output Class with Sensors

- Sensor Class was more valid when we had few standalone sensors.
  - Sensors are becoming integrated into other components
  - May make more sense to access from RobotBase, Tower, or Tool Classes where integrated components reside.
- Everything coming from various sensors
  - Distance – (analog) Sonic Range Finder, IR Distance sensor, LIDAR (laser), Potentiometer, String Potentiometer, 10 turn potentiometer
  - Deflection – (+/- double) Gyro heading angle, X,Y, Z Accelerometer, Tilt
  - Various contact and non-contact limit Switches – (digital/boolean) return true, or false
- Most return measurement to robot as a voltage. Device/API converts voltage to relative number.
  - Returns a number to your code that you interpret for your use.
- Pro: Now, they are relatively cheap!!!
- Con: They can be a little noisy, cooking helps.
  - They can also interfere with each other so test, test, test.

Sensors (delicious when cooked)

- Analog Sensors
  - Returns varying voltage for measurement or a number.
  - Use AnalogInput class to read noisy sensors.
    - getAverageValue() – Return averaged value over short duration.
      This helps minimize fluctuations. (Called over sampling)
      Range from 0 to 512, 1024, 2048 or 4096 depending on sensor.
    - getAverageVoltage() – Returns averaged voltage from sensor.
      This is a double value. IMHO: Not very useful better to use numbers.
- Deflection Sensors aka IMU (Inertial Measurement Unit)
  - Usually supplied from “add on” board of many flavors. (Google)
  - Detects acceleration in a several directions or turning (curvilinear).
  - Board APIs will return angles offset from 0.0, where 0.0 represents direction that robot was first pointed in.

```
bIsMoving  = ahrs.isMoving();        bIsRotating = ahrs.isRotating();
dNavxAngle = ahrs.getAngle();        dNavxPitch = ahrs.getPitch();
dNavxRoll  = ahrs.getRoll();         dNavxYaw   = ahrs.getYaw();
```

  - These can drift over time but good enough for robot round.
    - Never leave robot sitting running too long. Drift can add up.
    - Restart robot before round to be sure 0.0 is straight ahead.

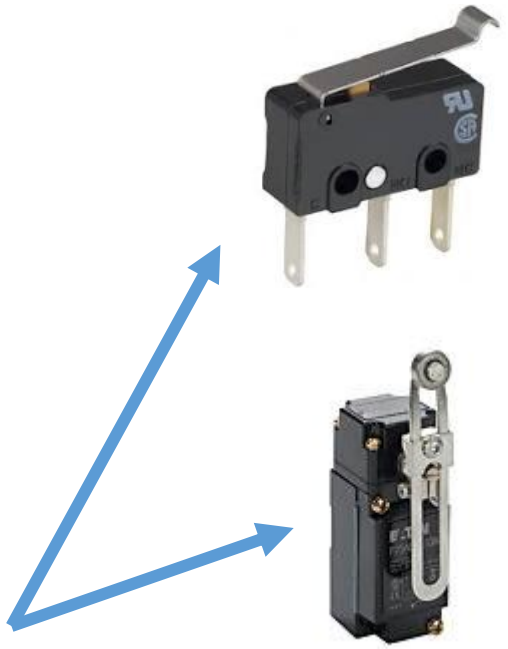
Sensors (delicious when cooked)

- Switches
  - Work as either on or off (digital). API returns a Boolean value (true/false)
  - Declaration:

```
DigitalInput digArmDownContact =
        new DigitalInput(RobotPorts.RobotBase_ArmDownContact);
boolean bArmIsDown = false;                    // declaration
```

  - Use example

```
bArmIsDown = digArmDownContact.get();          // return true, false
```



- Contact Switch
  - Will react when they physically press against something to indicate a limit or state.
- Non-Contact Switch
  - Sends out a beam of various flavors of light that, when blocked, indicates a limit or state change.

- Modes: Normally Open vs. Normally Closed
  - Website Explains NO vs NC
- Read FRC/FTC documentation (Google questions)

Sensors (delicious when cooked)

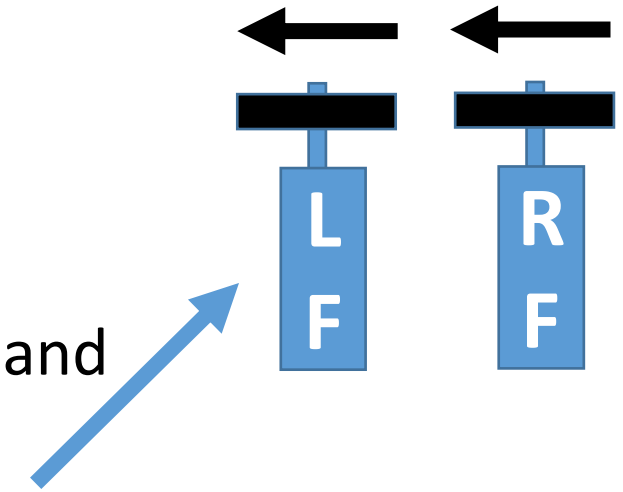
- Encoders
  - Encoders translate rotation into a distance in ticks
  - Encoders allow you to see how far your drive train has moved, measured in ticks or subdivisions of a full rotation.
    (Google for all the details)
  - Many encoders are now integrated into motors
  - Will usually start at 0 ticks and are not resettable.
  - How can you tell how far you moved in ticks?
    - Save the Current Ticks as iStartTicks.

    iStartTicks = motRigthFront.getCurrentPosition();
    - As you move subtract iStartTicks from CurrentPosition to get delta.

    iDeltaTicks = motRigthFront.getCurrentPosition() – iStartTicks;
    - iDeltaTicks is the distance in ticks.
    - Use ticks if you like or convert ticks to a unit of measure.
      - You determine conversion through a lot of testing and cogitating. ☺

RobotsBase – Outputs

- This is one place where you update your Base outputs
  - Set motors, pneumatics, lights, etc…
  - Better to do this in one place.
  - As always, cook the outputs too so they make sense.
- Speed Controllers run Motors that turn transmissions and (if lucky) spin wheels.
  - Motors are wired the same way. Red to Red, Black to Black
  - If mounted on the same side, when powered with same value, they rotate in the same direction.



- Declaration

```
TalonFX     motLeftFrontDrive  = null;
TalonFX     motRightFrintDrive = null;
```

- Initialization (in RobotBase class constructor)

```
motLeftFrontDrive  = new TalonFX(RobotPorts.kCANId_motLeftFrontDrive);
motRightFrontDrive = new TalonFX(RobotPorts.kCANId_motRightFrontDrive);
```

Note: These represent motors on an FRC CAN Bus. The API is from Talon not WPI.

RobotsBase – Outputs

- When Motors are on opposite sides, (right on right, left on left) one side will run in opposite direction
  - To run on the robot one side needs to be flipped around.
  - In this case, testing shows right is fine but we need to ***invert*** the left side final output to change its direction.
- Motor Inversion Options
  - When initialized we can use the motor's invert method.

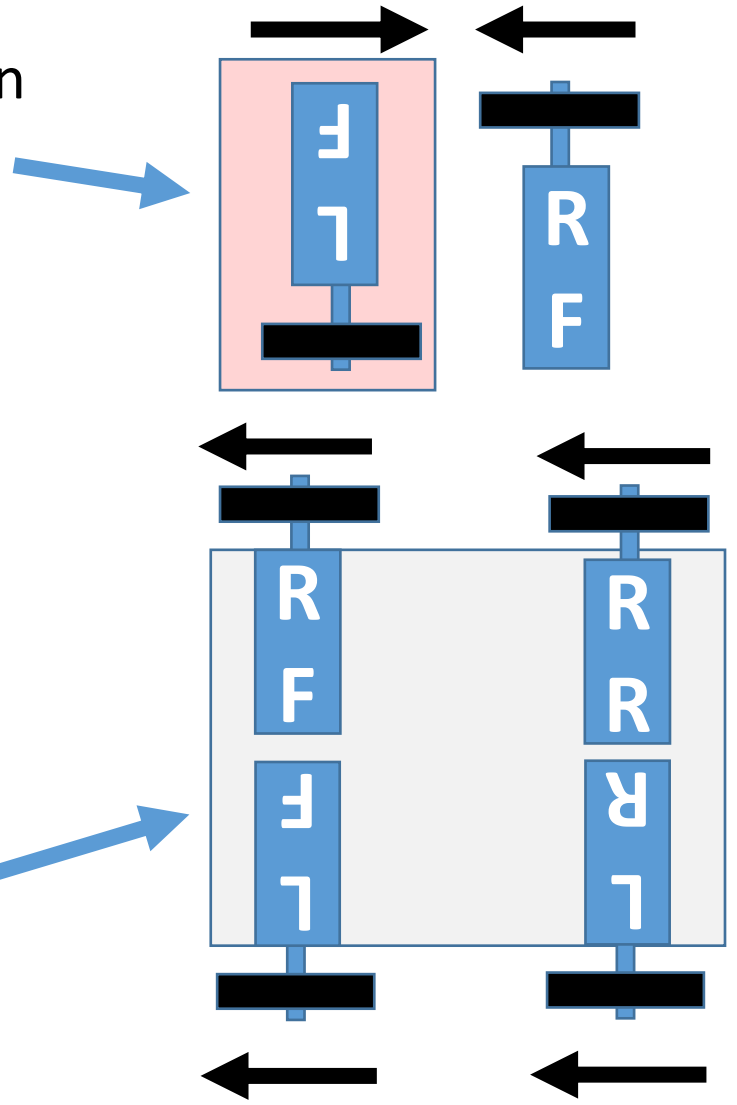
```
motRightFrontDrive.SetInverted(false);    // TalonFX API
motLeftFrontDrive.SetInverted(true);      // TalonFX API
// Later apply same power to both motors, MC converts output.
```

  - Invert Left side power when set to motor.

```
motRightFrontDriveMotor.set(inputs.dRightDrivePower); //
motLeftFrontDriveMotor.set(-inputs.dRightDrivePower);  // invert
```

-- OR –

```
motRightFrontDriveMotor.set(inputs.dRightDrivePower);
motLeftFrontDriveMotor.set(inputs.dLeftDrivePower * -1.0); // invert
```



Next Steps…

Move on to FMA+ FTC/FRC 02

https://midatlanticrobotics.com/fma-plus-training-ftc-frc-programming/