Autonomous Programming

2016-2017 FTC Challenge Workshops
VCU School of Engineering
September 24, 2016

Presented by:
Team 8297 Geared UP!

Autonomous in VELOCITY VORTEX

- The Match starts with a 30 second Autonomous Period where Robots are operated via “Pre-programmed instructions only” and teams are NOT allowed to control Robot behavior with the Driver Station or any other actions during the Autonomous Period.

***Scoring* Achievement**	***Autonomous* Points**
Beacons Triggered - State of *Claim* is all lit at end of period	30 / *Claim* + Bonus *Particle* awarded (up to 2)
Cap Ball - *On Playing Field* - *Off Playing Field below crossbar* - *Raised above crossbar* - *Capped Center Vortex*	5 - - -
Particle - *Scored in Center Vortex* - *Scored in Corner Vortex*	15* 5*
Robot Parked - *On Center Vortex Base* - *Completely On Center Vortex Base* - *On Corner Vortex Ramp* - *Completely On Corner Vortex Ramp*	5 10 5 10

* - *Scored* in real time by field personnel

Why Choose Android Studio?

Pros	Cons
• Android Studio is applicable outside of FTC, as the program is used in many tech companies and schools. • Also, Android Studio helps you learn the fundamentals of Java unlike programs such as App Inventor.	• It is difficult to use Android Studio without a proper understanding of Java. • In addition, unlike App Inventor, it does not have a built-in menu.

High Level Strategies for Autonomous

- Use utility methods for quicker and easier programming (non-programmer team members will also have an easier time learning the basics)
- Use an overall class structure (One main “autonomous” class with all methods. All individual programs will extend “autonomous”). See code snippet later on.
- Get the simplest options completed first (try to solve the simplest autonomous challenge first before attempting to add multiple motors for complicated tasks)
- Use as much sensor input as possible (example - we used the gyro sensor to detect if the robot had climbed on top a block/ball)

High Level Strategies for Autonomous (contd.)

- Create a “Menu Driven System” that lets the driver/on field coach select the program to run dynamically (our team used four touch sensors for selection, since the field was NOT a mirror image. Two more sensors were used for adding a delay, and the other two were used for selecting red/blue alliance)
- In VELOCITY VORTEX, the Red Alliance robot turns left to the ramp or push the beacons and the Blue Alliance robot turns right for performing the same action. By using a menu, a team can literally have one program and select the direction of turns based the alliance choice (instead of having multiple versions of the same program to run)
- Our recommendation is to extend the Linear Op mode for Autonomous programs. Why?
    - Event Driven mode unnecessarily overcomplicates things for autonomous programs
    - In most cases, the autonomous program will perform only one action at a time i.e. drive and then shoot a ball after stopping (instead of shooting a ball while driving) etc. Linear Op Mode is best suited for such activities and can be coded without worrying about setting up a state machine

Example Menu for Autonomous Programs

- In order to effectively organize autonomous programs, it is preferred that a menu is designed to let the driver / on field coach select the appropriate program.

```
if ((BLUEtouchSensor.isPressed()) && (REDtouchSensor.isPressed())) {
    continue;
} else if (BLUEtouchSensor.isPressed()) {
    telemetry.addData(" ==BLUE==BLUE==BLUE==BLUE==BLUE==", 0);
    return allianceColor.BLUE;
} else if (REDtouchSensor.isPressed()) {
    telemetry.addData("==RED==RED==RED==RED==RED==RED==", 0);
    return allianceColor.RED;
```

- The following source code is part of a program that uses touch sensors, each one representing the alliance program, for menus.
- An “if loop” is created to identify the alliance program to use by returning the alliance color based on the sensor pressed by the user. If both sensors are pressed simultaneously, then the program ignores the command and returns to wait for the next sensor press event.

Linear vs Event Driven Programming

- With the introduction of the new Android based control system in FTC, there are two different programming options to choose from:
  - Event Driven (class - OpMode)
  - Linear Mode (class - LinearOpMode)
- Event Driven - the code that runs the robot is inside the loop method. As the robot runs, the loop method will be called repeatedly, where the program can constantly read input from gamepads or sensors. This is perfectly suited for TeleOp as it would make it easier to adjust to changes from the gamepad or on the field.
- Linear Op Mode - Unlike with the Event Driven mode, the main method in is executed just once. So, all the logic for a specific autonomous program should be included in this method directly or in methods invoked from inside this method. This is useful for autonomous, as it executes a set of instructions in sequence one at a time and then finishes.

Linear vs Event Driven Programming (Contd.)

Linear Op Mode	Event Driven
State machine logic is actually implemented behind the scenes and hidden from the programmer	Requires understanding and use of a state machine and tracking of state transitions
All user code goes inside the runOpMode method and will be executed once in a sequence	All user code goes inside the runOpMode method and will be executed millions of times at processor speed
Simpler and easier to understand (great starting point for beginners!)	Requires the understanding of the abstract concept of a state machine and associated conditional logic
Supports only one task/action being performed by the robot at a given instant	Can be used to program parallel tasks for the robot (not a typical use case in autonomous)

Linear Op Mode - Example

- In Linear Op mode, the runOpMode method is used to configure the robot parts (motors, servos, etc.), gamepad controls, and other robot definitions. In this code, the motors and servos are being set a default name, and left motor's direction is set as reverse. Since all this code gets executed sequentially and only once, it can be used to manage the tasks in a flow from start to finish without worrying about tracking the robot status (the code execution point gives the status of the robot at any instant!).

```
@Override
public void runOpMode() throws InterruptedException {
  motorLeft = hardwareMap.dcMotor.get("motor_1");
  motorRight = hardwareMap.dcMotor.get("motor_2");
  neck = hardwareMap.servo.get("servo_1");
  jaw = hardwareMap.servo.get("servo_6");

  motorLeft.setDirection(DcMotor.Direction.REVERSE);

  // set the starting position of the wrist and neck
  neckPosition = 0.5;
```

Event Driven Mode - Example

Using Event Driven mode, the loop method is used to configure the gamepad controls in the example below. In this code from a TeleOp program, the Y button's function is defined to reverse the spool on a robot. This loop () method is invoked repeatedly (at processor speed) to pick up any events (button press, joystick motion, bumper push etc that get triggered by user action.

```
@Override
public void loop() {

    /*****************************************/
    /** GAMEPAD TWO CONTROLS (ARM and SPOOL) **/
    /*****************************************/

    if (gamepad2.y) { //If the y button on game pad 2 is pressed, set the spool direction to reverse and give it power. If hig
        SPOOL.setDirection(DcMotor.Direction.REVERSE);
        if (gamepad2.left_bumper) {
            SPOOL.setPower(0.75);
        } else {
            SPOOL.setPower(0.4);
        }
```

Setting up the overall Class Structure

This overall class WILL NOT contain your actual autonomous code, but will include utility methods, commonly used variables, etc. AutonmousVelVort is the class we developed for this purpose.

```
public abstract class AutonomousVelVort extends LinearOpMode {
```

Shown below are example methods that we used last season (Res-Q) and probably will use this season. The important methods were moveStraight and spinTurn used for navigation. Our team plans to use a Proportional Integral Derivative (PID) for straight moves this season, but we will go through a simplified moveStraight that calculates the required distance and sets power to the motors in the next few slides.

```
public void initializeGyro(GyroSensor gyro) throws InterruptedException {...}

public void initializeHardware() throws InterruptedException {...}

public void setAttributes(int pTicksPerRotation, float pWheelDia, float pWheelBase) {...}

public void stopDriveMotors() {...}

public void moveStraight (double nPower, double distance, boolean direction) throws InterruptedException{...}

public void spinTurn(float degrees, float power, moveDirection turn) throws InterruptedException {...}

public void moveByTime(double power, moveDirection dir, long timeMill, boolean abortWhenTilted) throws InterruptedException {...}
```

Setting up the overall Class Structure (contd.)

```
package org.firstinspires.ftc.teamcode;
import com.qualcomm.robotcore.eventloop.opmode.Autonomous;


/**
 * Created by govindb on 9/10/16.
 */
@Autonomous(name = "ShortBuildAuto", group = "ShortBuildAuto")

public class ShortBuildAuto extends AutonomousVelVort {

    @Override
    public void runOpMode() throws InterruptedException {

        {
            initializeHardware();

            setAttributes(1440, 3, (float)13.00);

            waitForStart();

            spinTurn(90, (float)0.5, moveDirection.RIGHT);

            sleep(1000);

            spinTurn(90, (float)0.5, moveDirection.LEFT);

            sleep(1000);

            spinTurn(90, (float)0.5, moveDirection.LEFT);

            sleep(1000);

            spinTurn(90, (float)0.5, moveDirection.RIGHT);

            sleep(1000);
        }

        exitOpMode();

    }
}
```

On the left, is an example autonomous program that turns the robot to the right by 90 degrees, left 90 degrees, left 90 degrees, and finally right 90 degrees. It invokes three methods from the “autonomous” class:

- initializeHardware (hardware mapping)
- setAttributes (wheel dia, wheel base, etc.)
- spinTurn (turns using a given degree number, power, and direction).

The key takeaway from this is the fact that instead of extending LinearOpMode directly, this class extends our own “autonomous” class - “AutonomousVelVort” (which extends the FTC class LinearOpMode). This way, it inherits whatever was in the LinearOpMode class along whatever we added to it as part of the class “AutonomousVelVort”. Essentially, your actual autonmous program looks like “Pseudo-Code” that even non-programmers on the team can manipulate as long as they know how to build and deploy to the phone via Android Studio.

Using Sensors to improve Navigation

- Sensors will be important in VELOCITY VORTEX. Unlike the past few seasons, scoring high in TeleOp depends on how much and what a team can accomplish during Autonomous based on the game set up.
- As much as possible, utilize at least 2 ways to measure something (for instance, use a color sensor to hit a line as well as encoders to set a max value in case the sensor misses the line).

- **Example Usage (* used in the past; + plan to use this season) :**
  - Gyro Sensor (z axis if the robot is on top of debris and tilted at an angle) *
  - Gyro Sensor (for turning) *
  - Color Sensor (for sensing beacon) +
  - Touch Sensor (finding the wall or selecting programs as mentioned earlier) *
  - Motor Encoder (probably the most important sensor on your robot) *
  - Ultrasonic sensor (for finding objects around the field) *
  - Limit switch (could be used for an arm) +
  - MR Rangefinder (Same use as ultrasonic sensor)
  - Optical Distance Sensor (Part of the range finder, for very close objects)
  - Compass Sensor (to identify the magnetic heading)
  - Android Phone Camera (With Vuforia software, you may be able to do some image recognition) +

Autonomous Program- full run

PID - Pranav

Testing Autonomous Programs - Neil

Tips & Tricks - Pranav

telemetry

Contact Info

Questions?

Email: tmgearedup@gmail.com

Website: http://www.gearedup.ashburnrobotics.com/