Command Based Programming

This will cover the basics of each file and directory of a command based robot. There will also be some snippets to see what can be changed to achieve different results.

Commands and Subsystems

In command based programming, the different tasks assigned to the robot are split up amongst what are called commands and subsystems. Typically, a command calls upon a subsystem to perform tasks. A subsystem has many commands associated to it.

Subsystems

A subsystem is a representation of a physical component on the robot. For example, if you had a drivetrain (which is the collection of wheels and motors on the robot), then you could represent it as a subsystem. Then, you could have that subsystem have various functions associated with it. For example, with a drivetrain, you could have it have a .stop() function and a .setSpeed() function. Then you would be able to refer to these functions in your commands.

Commands

A command is a series of functions that the robot executes. These functions stem from the subsystems. For example, if your drivetrain subsystem had .setSpeed() and .stop() functions, then you might have a command called goStraight. This command might be a combination of .setSpeed(1) and then a while later .stop(). With commands with you can place these functions successively one after the other (or in parallel to one another). It is then these commands that you assign to various buttons on your controller or joystick.

Robot.py

This file is the "entry point" of the program for the robot. This is the first file that the robot reads to know what it has to do.

When doing command based programming, this class must extend CommandBasedRobot. This "meta class" contains a few things that we need to define.

robotInit()

This function is the first function that gets called when the robot turns on. In this function you would put in all of the code that initializes the robot. For example, something that might need to be done before the robot does anything might be to calibrate any sensors, such as the gyro. For the competition, you usually have quite a bit of time for this function to run, so don't worry about your initializations taking too long.

autonomousInit()

This is a function that shouldn't receive too much attention. The code here from team to team shouldn't be too different. Essentially, here you would perform any code that initializes any systems that are unique to autonomous. Typically, the only thing done in this function is to select which autonomous routine to run from the dashboard (through a SendableChooser object from the SmartDashboard object).

That's it for robot.py in this example! In this file there really should not be too much going on. Most of the code should actually be delegated to the subsystems and their respective commands. From team to team, this file should be fairly similar.

OI.py (Operator Interface)

This file is where you would define all of the functionality of the peripherals that you need to run your robot (such as a joystick). This is

an *interface* to the *operator*'s tools. In here you assign different functions (commands) to various buttons on your joystick. This is a fairly straight forward process. First, you need an instance of your peripheral.

```
from wpilib.joystick import Joystick

global joystick
joystick = Joystick(0)
```

Now with this instance of Joystick you can assign different commands to various buttons. For example:

```
from wpilib.buttons.joystickbutton import
JoystickButton

trigger = JoystickButton(joystick,
Joystick.ButtonType.kTrigger)
trigger.whenPressed(someAction())
```

Here, we define trigger as an instance of a JoystickButton and assign it to the trigger of the joystick (assigning it to something different would be pretty confusing 😄). After that, we make sure that the .whenPressed() function knows which command to perform when we press the trigger (here we just have someAction() but you might have something like emergencyBrake() or something to actually have the trigger be useful). Of course, all of these functions that we assign are actions that must be defined, which we talked about a little earlier.

RobotMap.py

This is a very useful little file that allows us to "rewire" our robot. For example, if we were testing our robot with one of the motors being set

to port number 1 and then all of a sudden, the wiring team decides to make our lives hell by changing it to port number 3, then you might have to check a whole bunch of files in order to correctly reassign that motor to port 3. However, with RobotMap.py we are able to have that assignment of the motor to port 3 once and then use a reference to that port number all throughout the code. This means that instead of saying "use port number n for whatever motor, we can just say use port leftMotor for the left motor (or you could use some really confusing names like rightMotor for the left motor if you really wanted to). To do this there a few options. We could use a python dictionary, but typically for some reason, the standard for robotpy is to use a dummy class. All we do is make an instance of that class for each subsystem and then assign useful numbers to it.

For example, say we had a drivetrain subsystem. If we wanted to assign port numbers to the four motors of our drive train, all we would have to do is the following:

```
class PortsList:
    # This is the dummy class
    pass

# Make an instance of this dummy class
drivetrain = PortsList()

# Now we assign different port numbers to each motor
# which can be easily changed for any future
# annoyances that the electrical team gives us ;)
drivetrain.frontLeftMotor.portNum = 0
drivetrain.frontRightMotor.portNum = 1
drivetrain.backLeftMotor.portNum = 2
drivetrain.backRightMotor.portNum = 3
```

Now, for example, inside of a subsystem, instead of referring to a motor

by doing something like

```
self.motor = wpilib.Talon(0)
```

you would do something along the lines of of

```
self.motor =
wpilib.Talon(robotmap.frontLeftMotor.portNum)
```