2023 Robot Code

An in Depth Explanation

Team Steam 5119

Zachary Hansen Terry

Contents

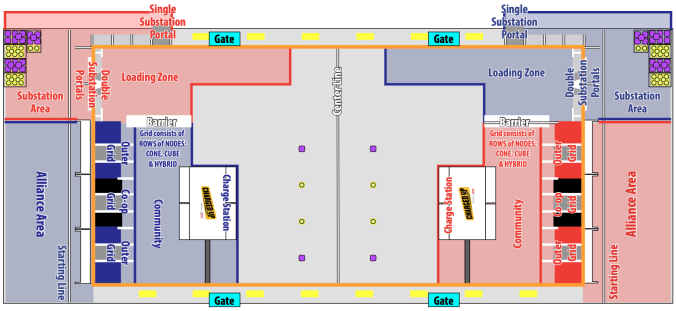

Figure 1: Game Diagram

1 Introduction

The First Robotics Competition game in 2023 was "Charged Up", a pick and place style game in which robots would attempt to move game pieces which consisted of cones (small yellow cones) and cubes (purple cube-like inflatables) to grids (multi-leveled assemblies with poles for cones to be placed on and shelves for cubes) for points. Placing a game piece on a higher level of the grid gains more points. The robot recives cones and cubes by:

1. Being pre-loaded in the robot at the start of the game (1 max)
2. Arranged on the floor of the game by alliance members (4 max)
3. Retrieved from human player station substation

The last 30 seconds of the game is the endgame in which robots should dock on the charge station, a balancing challenge with a platform that rocks forwards and back. It lights up when the charge station is level.

The 2023 robot, ANDY (Figure 2) was composed of 3 main mechanisms which would provide the complexity to our code:

1. Drivebase
2. Arm with motor at top for angle and motor for length control
3. Intake made up of rubber wheels which turn to "suck" the cone or cube in and hold it until it is released

2 Drivebase

2.1 Physical Explanation

The drivebase is made up of 3 wheels on each side of the chassis, the center one being dropped down by a couple millimeters relative to the outer two. They are driven over a belt that is turned by a single output shaft coming

Figure 2: 2023 Robot, ANDY

out of a 10.75:1 gearbox which has an input of two CTRE Falcon 500 motors. Falcon 500s are brushless motors which feature integrated motor controllers and encoders. Turning is achieved by differing the speed at which the two banks of motors spin.

2.2 Drive Subsystem

The drive subsystem was the largest subsystem we made in 2023 with over 190 lines of code. This guide will not go over all of the methods in the file because many are simple things such as getters or setters which are just methods that manipulate private variables. We will start with going over all objects that are created by the drive subsystem. First, driving related objects such as motor controllers and the differential drive are created:

```
// Physical motor controllers of the robot (4 Falcon 500s)
private WPI_TalonFX leftback = new WPI_TalonFX(MotorIDConstants.leftBackDrive);
private WPI_TalonFX leftfront = new WPI_TalonFX(MotorIDConstants.leftFrontDrive);
private WPI_TalonFX rightback = new WPI_TalonFX(MotorIDConstants.rightBackDrive);
private WPI_TalonFX rightfront = new WPI_TalonFX(MotorIDConstants.rightFrontDrive);

// Each side of the robot has a MotorController Group
private MotorControllerGroup left = new MotorControllerGroup(leftback, leftfront);
private MotorControllerGroup right = new MotorControllerGroup(rightback, rightfront);

// The DifferentialDrive contains the main methods of moving the robot
private DifferentialDrive diffDrive = new DifferentialDrive(left, right);
```

The next objects that are created are the NavX gyro meter and the SlewRate-Limiter. A rate limiter is a way to slow down the acceleration of the robot. We

used a rate limiter for our velocity because without it the robot was too sensitive and jerked forwards in response to input `SlewRateLimiter` takes in a value in it's constructor which is the units per second that a value should change by. The input for our speed is from zero to one so the the rate limiter should limit the robot to taking half a second before it achieves full output. Those are defined like this:

```
private AHRS navX = new AHRS(SPI.Port.kMXP);

private SlewRateLimiter rateLimitVelocity = new SlewRateLimiter(2);
```

To control the robot in the teleoperated period we used an Xbox One controller which was configured to change speed with the left joystick and rotation with the right. This is passed to a `curveatureDrive()` method which runs both sides of motors at different speeds to achieve turning. There is a variable called `halfSpeed` which when true will set the speed output of the robot to $\frac{1}{2}$ of the input. The drive code is as follows:

```
public void curveDrive(double speed, double roatation) {
    if(halfSpeed == false) {
      diffDrive.curvatureDrive(rateLimitVelocity.calculate(speed), -roatation, true);
    } else {
      diffDrive.curvatureDrive(speed / 2, -roatation / 2, true);
    }
  }
```

It is valuable to know how far the robot has moved. This can be used for autonomous purposes, as an input to PID controllers, and for the calculation of velocity. Encoder data is returned to us in units of ticks. We can convert ticks to rotations by dividing by 4096. This is then multiplied by the circumference of the robots wheels to find the distance that it has traveled.

$$\frac{\text{ticks}}{1} * \frac{1}{4096 \text{ ticks}} * \frac{\text{circumference}}{1}$$

2.3 PID Control

This guide assumes that the reader already understands how a PID controller works and what it is meant to achieve. In our 2023 robot code the team used PID Commands in order be best integrated into the command based nature of the 2023 code base. The point of using the PID Command class that is built into WPILib instead of a homemade solution is that it can be considered a command for all purposes a command would be used for which is especially useful for autonomous commands which are often many commands run sequentially.

One example of a way that PID Control was used in 2023 is the GyroTurn command. It is meant to turn the robot a certain amount of degrees using the NavX gyro as a means of finding error. Listing 1 shows the source code for the GyroTurn class. It is obviously inherited from the PIDCommand class and so it calls the `super()` method with the following parameters:

```
public class GyroTurn extends PIDCommand {
    public GyroTurn(double turnAngleDegrees, DriveSubsystem drive) {
        super(
            new PIDController(
                GyroTurnConstants.kP,
                GyroTurnConstants.kI,
                GyroTurnConstants.kD),
            () -> drive.gyroAngleDegrees(),
            turnAngleDegrees,
            output -> {
            drive.drive(0, output);
            });
        addRequirements(drive);
        getController().setTolerance(GyroTurnConstants.tolerance);
    }
    @Override
    public boolean isFinished() {
        return getController().atSetpoint();
    }
}
```

Listing 1: GyroTurn.java

1. A PID controller to use which is itself initialized with P, I , and D constants.
2. A lambda expression that will return the current measurement of the Gyro
3. The setpoint which is a double and passed through the constructor
4. An inline lambda expression that describes how to use the output. In this case it will go directly to the `drive()` function we made in **DriveSubsystem.java**

We also set the tolerance in the constructor. Tolerance is a value that is added and subtracted to the setpoint and exists because a Gyro cannot be 100% precise. If we didn't have a tolerance, it is very unlikely that the PID command would ever end because the `isFinished()` method would never return true. A command that doesn't end is almost always a bad thing because a command is always using physical resources such as motors or pneumatics and those resources cannot be used by other commands until the command currently using them ends.

PID control is used in many parts of the robot code and turning is just one example. Other cases where PID control was used in the 2023 code are: driving straight, driving a distance, raising the arm, extending the arm, and balancing

Position	Extension	Angle	Intake Pos.
Reset	0	0.537	Retract
Low	0.31	0.8	Deploy
Medium	0.172	1.52	Retract
High	0.45	1.65	Deploy
Human Player	0.45	1.65	Retract

Figure 3: Arm Positions

on the charge station. All PID control in 2023 was implemented using the `PIDCommand` class for simplicity.

3 Arm and Intake

3.1 Physical Explanation

The arm of the robot has 2 axis of movement: forwards and backwards (extension) and up and down (angling). The extension axis is managed using a single REV Robotics Spark Max brushless motor. The motor is attached to a series of belts with a gear that moves a belt that moves the arm in and out. Measurements are taken using the integrated encoder in the Spark Max brushless motor which is relative, not absolute. The angling of the arm is achieved using two REV Robotics Spark Max Brushless motors which are connected together through two 1:100 gearboxes which drive a gear on a chain connected to the axle that the arm angles on.

The intake is what picks up and holds game pieces for scoring. It has rubber wheels that pull a cone or cube into it and the wheels are reversed to eject the game piece. Because of the geometry/design of the intake, to pick up a cube the wheels must be ran in the opposite direction to when picking up a cone. The rubber wheels are driven over a belt connected to a REV Robotics Spark Max brushless motor. The intake is also able to be angled forwards and backwards through the use of two pneumatic pistons.

3.2 Arm Positions

A major theory that we based much of the robot code on was that there are only 5 different positions that the robot arm should be in: rest low, medium, high, human player station height. Each position has 3 different characteristics: arm extension, arm angle, and intake position as shown in Figure 3.

3.3 PID Control

Controlling the arm is a precise process because the position of the arm must be consistently correct for any scoring to take place. The extension of the arm was trivial because it has no forces acting against it. The angle was much

more difficult. The force of gravity is acting against keeping the arm up so a constant torque must be applied through the motors to keep it in position. This is solved by running the PID command for the arm constantly so that the setpoint always exists and it is dynamically kept in position. Another issue that gravity adds is that when going from a higher position to a lower position, gravity and the PID controller are working in the same direction providing too much torque which can cause the intake to slam into the ground with enough force to shatter the sidings[1]. To solve this, we made a separate PIDCommand class `ArmAngleLowPID.java` which has the same P, I, and D constants but divides the calculated output by 3.

3.4 Intake

The intake has two components: a motor for intaking and two pistons for deploying and retracting the intake. The pistons are controlled through a single solenoid connected to a compressed air tank. Both components are easy to control, the solenoid is just a matter of turning on and off at different positions and the intake should just be run at positive or negative speeds depending on whether a cube or a cone is being picked up. With cubes, it is important to run the intake at a lower magnitude as they are inflatable and will pop if they are pinched to hard.

4 Triggers and Xbox Controllers

4.1 The `Trigger` Class

A paradigm that was new to 2023 robot code was the usage of "triggers" in code. A trigger is what runs commands based on the state of the trigger. A trigger is often bound to a physical button for example on an Xbox controller, however it can also be bound to a condition in code through a lambda expression or function reference.

```
// Binding a trigger to a button "b" on controller "xbox"
private final Trigger button = xbox.b();

// Binding a trigger to a conditional function
private final Trigger condition = new Trigger(m_exampleSubsystem::isTrue);
```

4.2 Xbox Controllers

In 2023 we had two Xbox controllers to control the robot. One was for the driver and controlled just the robots location. The other controller was the

[1] This was when the sides of the intake were made of acrylic which was extremely fragile. They are now made of poly carbonate which is much more resistant. In fact, we have sheared the threaded metal rods going through the intake and not the poly carbonate.

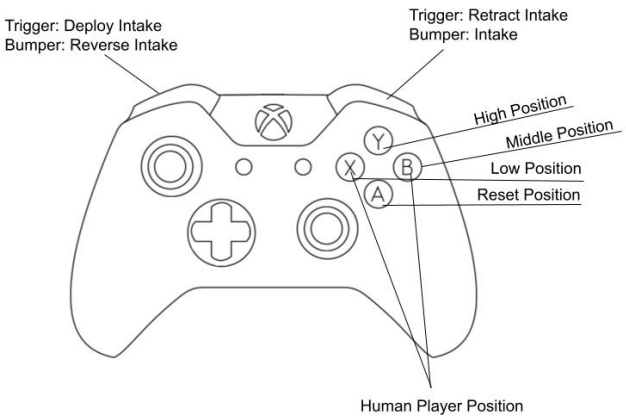

Figure 4: Operator Controller Button Bindings

operator and meant to control the other systems of the robot: arm, intake, extension, scoring. The operations are separated because we determined it would be too complex for our driver to control both driving the robot and operating the arm/scoring. In code we can define an Xbox controller as `CommandXboxController` which provides functions that return all the buttons as triggers.

4.3 Robot Container

The robot container is a file that contains all of the triggers, autonomous commands, subsystems, and default commands for the drive subsystem.
`RobotContainer.java` is on the highest level of the code directory. These are the parts that make it up:

1. Subsystem Declarations. This is where all subsystems that have been written are created. It is important that we consider these to be the "only" subsystems because a subsystem uses physical resources such as motors, solenoids, encoders, etc.. All use private access modifiers and they are passed into commands when they must be used.

2. Commands and Xbox Controllers. Declaring Xbox controllers (driver and operator) and triggers which are bound to buttons on the Xbox controllers.

3. Default Commands. Set inside of the constructor. A default command will always be running unless another command that requires the subsystem

is called. The drive default command is constantly reading the position of the Xbox controller joysticks and inputting them into the drive command.

4. Configure Bindings. Telling the library what commands to run when a trigger's specified state is true. Commands are started when they are created as objects in code using `new`.

5. Get Autonomous Command. Returns a command to run as soon as the autonomous period begins.

5 Command Based Programming

5.1 Commands

Commands are physical actions that the robot makes. All commands are inherited from the `CommandBase` class and contain `initialize()`, `execute()`, `end()`, and `isFinished()` methods to override. Initialize runs as soon as the command is scheduled and only once. Execute runs every 20 milliseconds for the duration of the command. End is called right before the command ends and `isFinished()` is called every 20 milliseconds for the duration of the command until it returns true in which case the end function is called and then the command is ended.

A classic example of a command is the Drive command as shown in Listing 2. It takes in double suppliers[2] for speed and rotation and a drive subsystem (the one from `RobotContainer.java`). Another important part of a command is the `addRequirements()` method. It defines which subsystems a command uses and doesn't allow other commands to be using the same subsystem at the same time.

Often, it is useful to combine multiple commands together, for example in an autonomous command. We can sequentially run multiple different commands in an order to achieve our task. Another instance of combining commands in our 2023 robot code is the arm position commands. The arm should first angle up, then extend and retract the intake. This type of combining commands is called a command group and can be done using the following classes:

1. Sequential Command Group. Runs every command in it's contents in a sequence, one after the other. Not used very often because we don't let our PID commands ever return true in isFinished() so the sequence would never move onto the next command.

2. Parallel Command Group. Runs every command in it at once. It is important that commands in a parallel command group do not require the same subsystems because they **will** both attempt to access them at once and crash the code.

[2] A double supplier is like a lambda for numbers. It tells the code where to find a value instead of the value itself because the speed that we want the robot to drive and it's rotation constantly change.

```java
public class Drive extends CommandBase {
  DriveSubsystem m_driveSubsystem;
  DoubleSupplier driveSpeed;
  DoubleSupplier rotationSpeed;

  public Drive(DriveSubsystem driveSubsystem, DoubleSupplier driveSpeed, DoubleSupplier rota
    m_driveSubsystem = driveSubsystem;
    this.driveSpeed = driveSpeed;
    this.rotationSpeed = rotationSpeed;

    addRequirements(driveSubsystem);
  }

  @Override
  public void initialize() {}

  @Override
  public void execute() {
    m_driveSubsystem.drive(driveSpeed.getAsDouble(), rotationSpeed.getAsDouble());
  }

  @Override
  public void end(boolean interrupted) {
    m_driveSubsystem.stop();
  }

  @Override
  public boolean isFinished() {
    return false;
  }
}
```

Listing 2: Drive.java

```
public class Auto11 extends SequentialCommandGroup {
  public Auto11(DriveSubsystem drive, ArmExtensionSubsystem armExtention, ArmSubsystem armRo
    addCommands(
      new ReverseIntake(intake).raceWith(new WaitCommand(1)),
      new ParallelCommandGroup(
        new HighArmPosition(armExtention, pneumatics, armRotation),
        new SequentialCommandGroup(
          new WaitCommand(1),
          new Intake(intake).raceWith(new WaitCommand(1))
        )
      ).raceWith(new WaitCommand(4)),
      new Drive(drive, () -> 0.3, () -> 0).raceWith(new WaitCommand(7))
    );
  }
}
```

Listing 3: Auto11.java

3. Parallel Race Group. Runs every command at once and when one of the commands finish it ends every command in the group. Used very commonly with `WaitCommand(t)` to cause a command to only run for a certain amount of time.

An example of an autonomous command that uses these command groups is `Auto11.java` shown in Listing 3. In sequence it reverses the intake to hold a cube in, moves the arm to the high position, runs the intake to score the cube, and then returns to the reset position and drives backwards to leave the community and get the mobility points. One thing to note: a few commands are using something that looks like `raceWith(new WaitCommand(t))`. This is the same as `new ParallelRaceGroup(new Command(), new WaitCommand(t)` just more concise.

5.2 Subsystems

It often makes sense to group certain parts of the robot together in code. For example, all the drive motors and their encoders and the gyro meter are all concerned with the the location of the robot. Encapsulating these objects in one class, a *subsystem* is a way that we can create methods that use all of these components and keep our code organized. All objects and variables defined in a subsystem should be private and manipulated through getters and setters. A subsystem should contain the logic of a robot while the command defines the actions. There is only one method to override in a subsystem: `periodic()`. Periodic is run every 20 milliseconds for as long as a subsystem exists (which is always because java doesn't have manual memory management).

6 Autonomous

6.1 Intro

The first 15 seconds of the game are the autonomous period in which the robot can attempt to score points without the driver, operator, or human player's help. Points scored in autonomous are worth considerably more than during the teleoperated portion so it is very important to have a functional and advanced autonomous.

All of our autonomous commands are based off of sequential command groups. Sequential command groups are a way provided by WPILib to run multiple commands one after the other. A common method is to "race" a command against a `new WaitCommand(time)` for a command that should end after a set amount of time.