2025

FTC 24764 CODEBASE MANUAL

V1.0 NOVEMBER 29, 2025

BY: MENTOR LANDON SMITH

Contents

Introduction

The purpose of this manual is to provide a clear, structured guide for developing, maintaining, and extending the FTC Robotics Team 24764 codebase. This document is designed for both new and experienced team members, ensuring consistency and best practices across all programming tasks.

Overview of the Codebase

The codebase follows a modular architecture built on the FTC SDK and enhanced with the following components:

- **OpModes**: These are the entry points for TeleOp and Autonomous programs. Each OpMode defines the robot's behavior during a match.
- **Subsystems**: Subsystems encapsulate hardware components (motors, servos, sensors, and gamepads) and their associated logic. This design promotes code reuse, readability, modularity, and easier debugging, along with safeguards and better error handling. It also prevents spaghetti code.
- **Utilities**: Includes helper classes AxisState, ButtonState, TriStateButtonState, enum GamepadButton, enum BiStateButtonBehavior, and enum AxisBehavior for managing input and button states.

Configuration: Tunable constants are in Classes annotated with @Config for real-time adjustments using FTC Dashboard (Landon Smith refers to this has hot patching). These values are NOT persistent and will reset whencontrol hub is rebooted. Once a value is dialed in, change it in the code, and push the new code with the new constant to make it persistent.

- **Subsystem conventions flowchart**: A decision-making flowchart helps determine whether new functionality should be implemented as a private TeleOp function, or a new subsystem.

This architecture ensures:

- **Scalability**: Easy addition of new features without breaking existing functionality.
- **Maintainability**: Clear separation of concerns between hardware control, input handling, and high-level logic.
- **Flexibility**: Supports both TeleOp and Autonomous modes with reusable components. This is most evident in constructor overloading in the subsystems, so they can be initialized in either TeleOp mode, or Autonomous mode.

Setup & Usage

Required Tools (Install before proceeding)

- **Android Studio** – For writing, building, and deploying code.
    - Installing the Android virtual device (file size is >1gb and is not required for anything FTC Related, highly recommend not downloading it)
- **Git (Windows)** – For version control and managing changes. Can be installed from Android Studio or GitHub Desktop.
    - **Command (copy and paste into powershell on Windows 11)**:
        - `winget install --id Git.Git -e --source winget`
- **GitHub Desktop** – For an easy graphical interface to manage branches and commits.

---

1. Clone the Repository

- Open **GitHub Desktop** or a terminal.
- In GitHub Desktop:
    - Click **File → Clone Repository**.
    - Clone lando07/FTC-RC
    - Use the default GitHub Desktop Folder, it will be in: `C:\Users\YOUR_USER\Documents\GitHub Desktop\FTC-RC`

---

2. Open in Android Studio

- Launch **Android Studio**.
- Select **Open an Existing Project** and choose the cloned folder.
    - Hint: Look in the GitHub Desktop Folder for FTC-RC in your Documents Folder
- Wait for Gradle to sync. If prompted, install any missing SDK components. If configured correctly, Gradle should handle all dependencies, and you should not need to do anything.

    - Gradle Sync times can take a while, factors include CPU, RAM, Internet Speed/connection, and SSD speed.
        - The faster (or greater) these components are, the faster you can sync Gradle and push code.

---

3. Build & Deploy

- Connect the **REV Robotics Control Hub** via USB-C
- In Android Studio:
    - Wait until the REV Robotics Control Hub shows up as a connected device
    - Click **Run (The green play button next to TeamCode)** to deploy the selected OpMode.
    - Wait until the Indicator LED on the Control Hub is solid green, and the green play button is now highlighted green, and a red stop button appears. This will confirm that the new code has been launched and you can now use the robot.

Maintenance Guidelines

1) Coding Standards

Project structure

- Group related classes under org.firstinspires.ftc.teamcode (e.g., subsystems, opmodes, tuning, util).
- Autonomous Simulations under MeepMeepTesting
    - Package Name: com.example.meepmeeptesting
- Each **subsystem** is a single class focused on one mechanism (e.g., Claw, DriveTrain, RaiseArmSlider).
    - This mechanism can include any number of functions, tunable constants, motors, servos, etc.

File & class naming

- Classes: **PascalCase** (e.g., RedLaunchZoneAuto, XDriveDECODE).
- Methods & variables: **camelCase**
    - e.g., `openClaw(), updateDriveTrainBehavior()`
- Constants: `private final` when truly constant; tunables remain `public static` for FTC Dashboard.

Annotations & visibility

- **Every subsystem must be annotated with @Config** so constants are tunable in FTC Dashboard. Refer to the previous bullet on how to get tunable constants to appear in FTC Dashboard
- Mark OpModes with @TeleOp or @Autonomous. Use @Disabled to hide experimental or archived modes.
    - Example: `@TeleOp(name="XDriveDECODE", group="TeleOp")`
- Hardware fields are `private final` once initialized in the constructor.
- Expose **action methods** (e.g., openClaw(), doPrimaryOn()) publicly as wrappers for autonomous; keep helpers private.
    - Helpers can be public if also used in TeleOp

Input handling

- Define key binds as **`public static GamepadButton`** with matching **`BiStateButtonBehavior`**.
- Configure buttons in the subsystem constructor via
    - `configureBiStateButton(GamepadButton button, BiStateButtonBehavior behavior)`
- Use a dedicated, unique update() method in each subsystem to read new controller input values and apply behavior-driven changes.

Comments & documentation

- Use Javadoc on public classes/methods that are called from TeleOp/Autonomous.
    - Denoted as /** ... */
- Keep comments focused on **why** (intent), but also explain **what** (code behavior)
    - This allows new programmers and advanced programmers to understand what the documented method, variable, or field does.
- Prefer short method names with clear purpose (e.g., closeClawAction() returns an InstantAction).
    - In the case of Actions, any method that returns an action should return an InstantAction that uses a helper method already present. Action should only be appended to the method signature and not add any new functionality. This is for the sake of readability, simplicity, and excessive boilerplate surrounding the actual code by moving it to it's own place.
    - For Example, the method:

```
public InstantAction clipSpecimenAutoAction(){
    return new InstantAction(this::clipSpecimenAuto);
}
```

      Only exists to wrap `clipSpecimenAuto()` into an instant action for autonomous use.

---

2) Dependency & SDK Management

FTC SDK & Dashboard

- Keep the SDK and dashboard versions aligned with the current season (update once per off-season or when required by new hardware).
- After version changes, **sync Gradle**, clean build, and test on a physical device.
- In cases where Gradle starts throwing lots of errors, it is best to back up any modified code not committed to the repository, delete your local copy, and re-clone from GitHub following Gradle or SDK upgrades.

3) Debugging Best Practices

Telemetry as first-class feedback, do not forget exceptions

- Log concise, meaningful messages (e.g., current servo position, motor power, active states).
- If a method cannot run without certain conditions met (see GamepadController.java) throw verbose exceptions that explain exactly what conditions were met/not met to throw the exception.

State-change driven logic

- In update(), act **only when a logical state changes** to avoid redundant commands.
    - The less that is done in an update cycle when no input changes, the less time the main control loop takes, reducing input latency and subsystems competing for CPU time.

Hardware safety

- Always set sensible defaults:
    - Servos: an initial safe position in initialization, if permitted by rules, if not, start servo in a position that can be safely held without any power
    - Motors: ZeroPowerBehavior.BRAKE for mechanisms that must hold, FLOAT when motors do not need to enforce their position
        - For example, a motor on the FTC Into the Deep Robot that extended a rack-and-pinion into the submersible to grab samples had enough friction between the holder and rail that it could easily hold its position without braking.

- Guard against null hardware by initializing in the constructor and failing fast if a mapping is missing.

Reproducible experiments

- When tuning with Dashboard, ALWAYS **record final values** back into code after tests (push code to the robot).
- Use feature flags or simple booleans to toggle experimental paths without branching explosion.

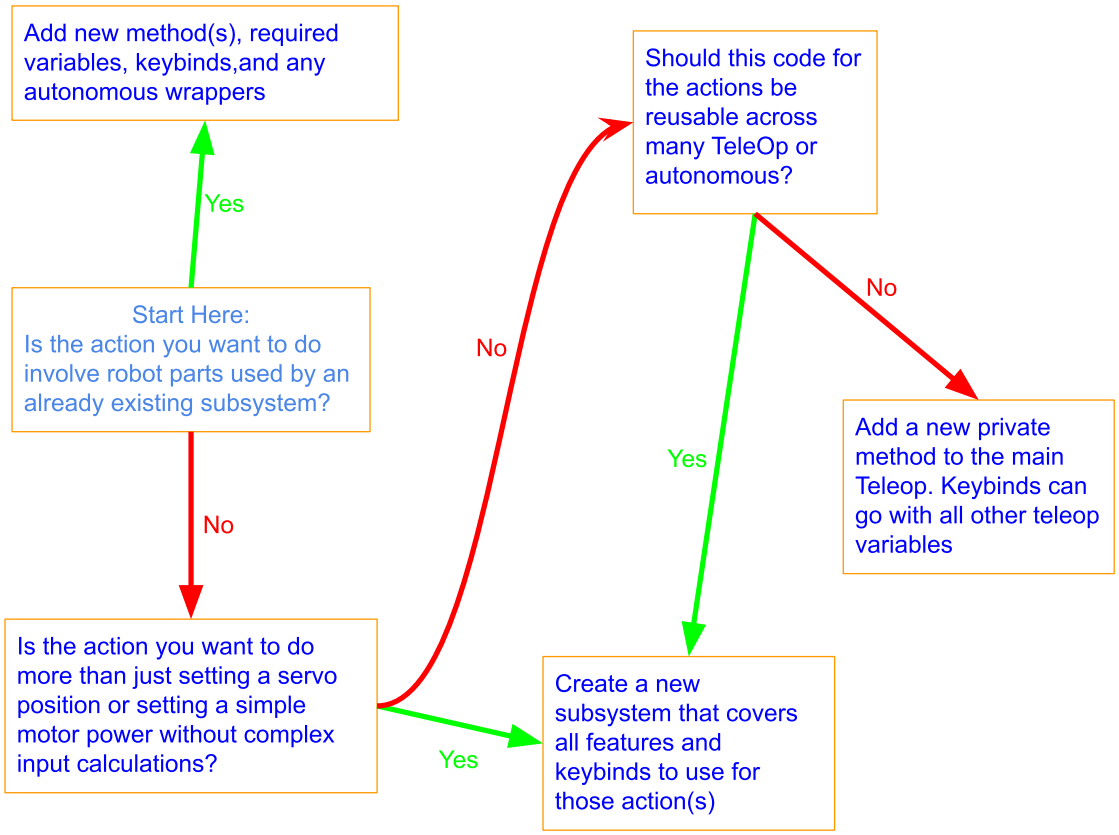
---

4) Routine Maintenance Cadence

- **Per Feature**
    - Add/adjust @Config tunables.
    - Update the **Templates** section if you discover a pattern worth reusing on all Subsystem, TeleOp, or Autonomous code.
- **Per Season**
    - Archive deprecated OpModes/subsystems (move to a legacy/ or mark with @Disabled and @Deprecated).
    - Upgrade SDK and libraries; retest drivetrain and key subsystems ended.

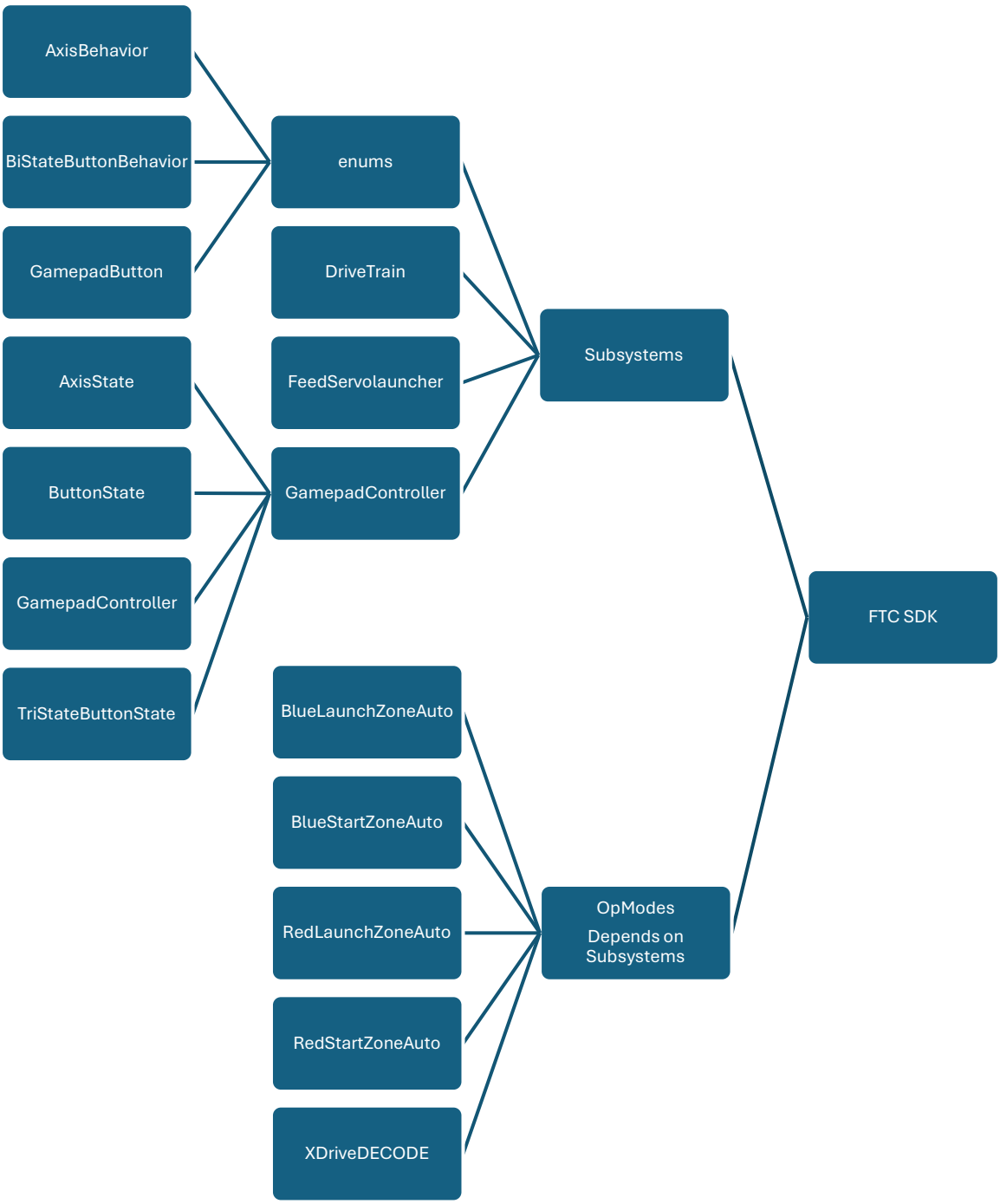
Adding Features

A simple flowchart to determine how to add new features and code. Follow as closely as possible to maintain readability, consistency, and easier debugging.



Subsystem Structure

Note: Only Shows currently used OpModes and Subsystems.



Templates

Template for Adding a New Subsystem:

Sample Code:

```
package org.firstinspires.ftc.teamcode.subsystems;

import androidx.annotation.NonNull;

import com.acmerobotics.dashboard.config.Config;
import com.qualcomm.robotcore.eventloop.opmode.OpMode;
import com.qualcomm.robotcore.hardware.Servo;

/**
 * This class is a template on how to create a subsystem,
 * the possibilities are endless on what you can do. The subsystem
 * architecture aims to allow for code reusability and readability,
 * while staying very simple
 * @author Mentor Landon Smith
 */
@Config
public class SubsystemTemplate {
    /*
     * Use public static variables and objects
     * to store constants that should be
     * tuned on the fly using FTCDashboard.
     */
    public static double modifiableConstant = 0.5;
    /*
     * States that must be accessed across multiple methods are typically
private,
     * for example, this boolean can store the state of a servo and which
position it is,
     * if it doesn't need absolute positioning control, but needs to jump
between 2 positions.
     */
    private boolean state;
    /*
     * Use the GamepadButton Enum to store keybinds associated
     * with said subsystem - "A" could be used to open a claw.
     */
    public static GamepadButton keybind1 = GamepadButton.A;
    /*
     * Use the BiStateButtonBehavior to assign how a button will act.
     * There are 2 modes, TOGGLE, and HOLD. Toggle will invert states when
the button is pressed,
     * while HOLD will only return 1 when the button is pressed.
     */
    public static BiStateButtonBehavior keybind1Behavior =
BiStateButtonBehavior.TOGGLE;
    /*
     * You can also create Tri-States with 2 buttons.
     * A tri-state works as follows:
     * 1 : buttonPositive is pressed
```

```
     * 0 : no buttons are pressed
     * -1 : buttonPegative is pressed
     * Great for d-pads and bumpers.
     */
    public static GamepadButton buttonPositive = GamepadButton.D_PAD_UP;
    public static GamepadButton buttonNegative = GamepadButton.D_PAD_DOWN;
    /*
     * An example of a servo object. Should be private, and if always
initialized,
     * will have the tag 'final' after 'private'.
     */
    private Servo servo1;
    /*
     * This will be present in every subsystem created,
     * this holds the reference, not a copy, of the gamepad controller
     * used to get input from the the users. This can be either gamepad,
     * but that information is not needed for any functionality. The gamepad
     * that is assigned is given in the constructor parameter by the teleop.
     */
    private GamepadController controller;
    /*
     * This is a constructor for TeleOp Use, it will get a reference to the
opmode,
     * the assigned controller, and if needed, a string with the hardware
name.
     * The string is optional, and should be defined in the constructor if
there
     * is more than one motor/servo being initialized
     */
    public SubsystemTemplate(@NonNull OpMode opmode, @NonNull
GamepadController controller, String hwName) {
        // This initializes the servo with it's name in the driver hub
configuration
        servo1 = opmode.hardwareMap.servo.get(hwName);
        // This stores the reference to the assigned gamepad from the TeleOp
        this.controller = controller;
        // These 2 method calls assign buttons to the controller we were
given.
        // By using gamepadController, only this class can use these buttons
to prevent
        //undefined behavior.
        controller.configureBiStateButton(keybind1, keybind1Behavior);
        controller.configureTristateButton(buttonPositive, buttonNegative);
        //This initializes the initial state of whatever it is being used for
        //It does not have to be a boolean
        state = false;
    }
    /*
     * This is an overloaded constructor, primarily used for autonomous
opmodes.
     * Since the autonomous does all of the required work, we do not/cannot
use
     * any controller.
     */
    public SubsystemTemplate(@NonNull OpMode opmode,String hwName){
        //Same initialization as above
        servo1 = opmode.hardwareMap.servo.get(hwName);
```

```
        //This initializes the initial state of whatever it is being used for
        //It does not have to be a boolean
        state = false;
        //Additional code can go here that's exclusive to autonomous coding
    }

    //Below is where you put any methods, sometimes I call features, where
all of your
    //reusable code goes.
    /*
     * This is a simple toggler that toggles the state
     */
    public void toggleState() {
        if (state) {
            state = false;
        } else {
            state = true;
        }
    }
    //See other subsystems to see what other features and methods are used in
a subsystem



}
```

Template for Adding a New TeleOp:

Sample Code:

```
package org.firstinspires.ftc.teamcode;

import com.acmerobotics.dashboard.config.Config;
import com.qualcomm.robotcore.eventloop.opmode.Disabled;
import com.qualcomm.robotcore.eventloop.opmode.OpMode;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.hardware.DcMotor;

import org.firstinspires.ftc.teamcode.subsystems.DriveTrain;
import org.firstinspires.ftc.teamcode.subsystems.GamepadController;

/**
 * This file contains a minimal example of a Linear "OpMode". An OpMode is a
'program' that runs in
 * the teleop period of an FTC match. The names of OpModes appear on the menu
of the FTC Driver Station. This includes
 * full drivetrain functionality, so if used, you can at lease move the robot
around using mecanum drive
 * according to it's specs.
```

```
 * @author Mentor Landon Smith
 */
@Disabled
@Config
@TeleOp(name="TeleOpTemplate_WithDriveTrain", group="TeleOp")
public class TeleOpTemplate_WithDriveTrain extends OpMode {
    /**
     * Stores the drivetrain subsystem object which handles all movement and
motor power.
     * All drivetrain configurations are stored in the DriveTrain class,
since the keybinds
     * are universal across mecanum drivetrains, ftc, and also the fact it
requires 2
     * joysticks.
     */
    private DriveTrain driveTrain;
    /**
     * These will be the 2 controllers to use and pass through to
     * other subsystems for user input
     */
    private GamepadController controller1, controller2;

    @Override
    public void init(){
        //Initialize subsystems and controllers here
        //Always create objects before configuring them, or you will get
        //a null pointer exception
        controller1 = new GamepadController(gamepad1);
        controller2 = new GamepadController(gamepad2);
        //pass through controller 2 if you want driver 2/gamepad2 to control
movement
        driveTrain = new DriveTrain(this, controller1);
    }
    @Override
    public void init_loop(){
        //Not needed, but serves as a loop for code that needs to run
        //continuously between initialization and pressing play on the driver
hub
    }
    //This is one-time code to run at the beginning of a match.
    //Here can be code to pre-start motors, servos, etc.
    @Override
    public void start(){
        driveTrain.setBrakingMode(DcMotor.ZeroPowerBehavior.BRAKE);
    }

    /**
     * This is the main control loop that runs during an FTC TeleOperated
period.
     * Because of our architecture, the 3 things allowed to go here are as
follows:
     * <br />
     * 1. Subsystem updates
     * <br />
     * 2. Controller updates
     * <br />
     * 3. Very primitive motor/servo movement(no complex functions, features,
```

```
or math)
     */
    @Override
    public void loop(){
        // Update controllers
        controller1.update();
        controller2.update();

        // Update drivetrain
        driveTrain.updateDriveTrainBehavior();

        //Update subsystems here


        //Run any primitive motor/servo commands here


        //Update telemetry here

        telemetry.update();

    }


}
```

Enum Creation for wrapping magic numbers:

```
package org.firstinspires.ftc.teamcode.subsystems.enums;

import androidx.annotation.NonNull;

/**
 * This is a template for creating enums. They enumerate
 * magic numbers into friendly names and are quite useful
 * for repetitive features like buttons or offsets.
 * @author Mentor Landon Smith
 */
public enum EnumTemplate {
    /*
     * Use typical naming conventions for the name of the variable
     * in code, but give it a friendly name for identification in
     * exceptions and telemetry. As defined in the parenthesis,
     * when an enum is initialized, the friendly name is the default
     * string to pass through to the constructor.
     */
    VALUE_1("Friendly Name 1"),
    VALUE_2("Friendly Name 2");
    /*
     * This string is what stores the friendly name of
     * the enum value initialized.
```

```
     */
    private final String displayName;
    /*
     * I haven't seen enums typically use constructors for more
     * than just this, but it is a constructor nonetheless.
     */
    EnumTemplate(String displayName) {
        this.displayName = displayName;
    }
    /*
     * Every Java object inherits Object, the parent class
     * of all objects. In this instance, the Object class
     * has it's own toString() method that this class inherits,
     * but we want different behavior. This override defines that
     * behavior, and is annotated with the @Override annotation
     * so that if anything goes wrong, the code will error out
     * before compilation completes.
     */
    @NonNull
    @Override
    public String toString(){return displayName;}
}
```

MeepMeep autonomous template for simulation:

```
package com.example.meepmeeptesting;

import com.acmerobotics.roadrunner.Pose2d;
import com.noahbres.meepmeep.MeepMeep;
import com.noahbres.meepmeep.roadrunner.DefaultBotBuilder;
import com.noahbres.meepmeep.roadrunner.entity.RoadRunnerBotEntity;

/**
 * This is an example class on how to create an autonomous
 * simulation using MeepMeep.
 */
public class ExampleMeepMeep {
    /*
     * Every MeepMeep will use a main method to run it's code,
     * as it is fully self-contained.
     */
    public static void main(String[] args) {
        //This enables hardware acceleration so the simulation
        //runs without slow-downs
        System.setProperty("sun.java2d.opengl", "true");
        //This is our meepmeep instance. The 700 is the dimension
        //of the window in pixels.
        MeepMeep meepMeep = new MeepMeep(700);
        //This creates a virtual robot where all of your actions
        //will be assigned to. The constraints are found in the
        //MecanumDrive class in the teamCode.
```

```
        //Remember to update these constraints as
        //you do in the MecanumDrive class, to keep your
        //simulations and real-world semi-accurate
        RoadRunnerBotEntity myBot = new DefaultBotBuilder(meepMeep).
                setConstraints(90, 70, 55, 60, 14).
                build();
        myBot.runAction(myBot.getDrive().actionBuilder(new Pose2d(0,0,0))
                //Place your actions here
                //NOTE: Only strafes, splines, and lines work here,
                //since all you're simulating/creating are the waypoints
                //and paths the robot will take
                .build());
        //This final line should not be modified, except for changing the
background
        //This line tells the meepMeep object to set some parameters and then
        //run our actions laid out above.
        meepMeep.setBackground(MeepMeep.Background.FIELD_DECODE_JUICE_DARK)
                .setDarkMode(true)
                .setBackgroundAlpha(0.95f)
                .addEntity(myBot)
                .start();

    }
}
```