PROGRAMMER'S GUIDE

Introduction

This guide is structured into two comprehensive sections—"**Walkthrough**" and "T**ailoring**." Together, they aim to support both absolute beginners and more curious learners who want to understand not only how to program an FTC robot, but also why the code works the way it does. We'll mainly use Kotlin because its syntax may be more intuitive and cleaner, but all these steps can be easily followed in Java too.

The **Walkthrough** section serves as a practical, step-by-step introduction to FTC programming. It is specifically designed for individuals with no prior coding experience who may suddenly find themselves responsible for programming a competition robot. Each chapter breaks down the essentials in a clear, approachable format: setting up the development environment, writing your first OpModes, understanding motors and sensors, and building the confidence needed to control a real robot. The goal is to provide an accessible path for new programmers that is straightforward and can be easily used and further developed.

The **Tailoring** section transitions from hands-on tasks to deeper technical understanding. Here, we explore the core programming concepts that shape FTC robotics: control flow, data structures, program architecture, abstraction, and strategies for writing clean, scalable code. This material does not require access to a robot and can be studied comfortably with just a laptop. It helps programmers understand how to customize, optimize, and refine their robot's behavior, transforming basic scripts into robust, competition-ready software.Although the material may appear information-dense initially, each chapter becomes valuable only after further research and practical testing; don't merely read-apply the concepts by experimenting and developing your own implementations.

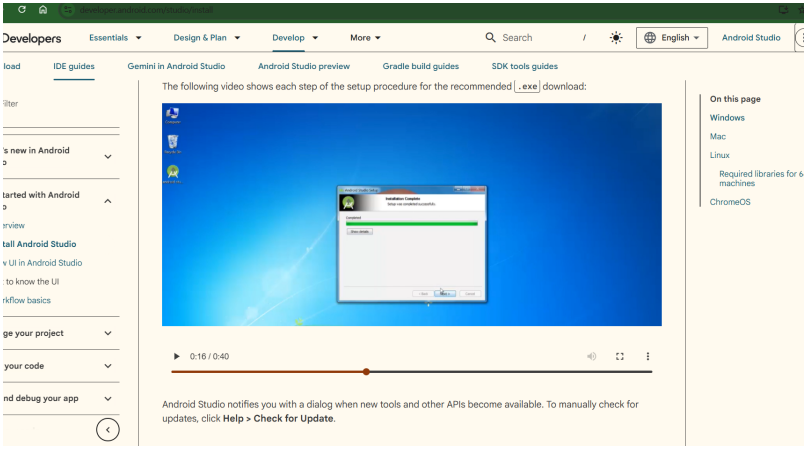
1. Walkthrough

1.1. Setting up Android studio

Without further ado go to https://developer.android.com/studio and click on download after reading and agreeing with the terms and conditions provided. Then an exe file will be downloaded and then you will have to follow the steps that are opened in the new windows. (video)

These steps are for Windows, for the sake of the majority, but you can also install it on mac and linux and the processes are similar, and you can check these out right here https://developer.android.com/studio/install.

Congrats, you just finished setting up Android Studio.



Screenshot of the video opened using the previous link

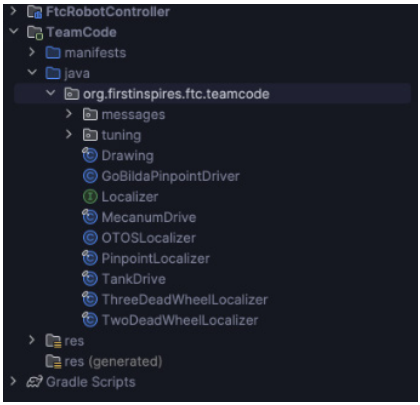
1.2. FTC-SDK

For us to actually start programming we can t just start a new empty project. We need a module that is used to build the android app for us to be able to control an actual robot. We recommend to start with roadrunner-quickstart since it has a built in ftc dashboard, and we can skip the step of adding it in our project later on. For us to actually start programming we can t just start a new empty project. We need a module that is used to build the android app for us to be able to control an actual robot. We recommend to start with roadrunner-quickstart since it has a built in ftc dashboard, and we can skip the step of adding it in our project later on.

Adding it to android studio should be pretty simple. On the welcome page you have these options:

Choose clone repository option and then paste the quickstart's url https://github.com/acmerobotics/road-runner-quickstart, and then wait a few seconds for it to load.

All the code will be written on this path:
TeamCode/src/main/java/org/firstinspires/ftc/teamcode



Screenshot from Code Structure

You can see that there are already some classes, we aren't going to use them right now so go ahead and make 2 new packages named "roadrunner", and "opmodes", and add all the files in it.Your project should look like this .

1.3. Installing ADB

ADB (Android Debug Bridge) is a versatile and flexible tool used to **communicate with your FTC control system devices**—in our case the Control Hub. It allows you to upload your code wireless, which we think gives a major advantage especially during competitions, due to the fact that you can be out of the field and update the code without worrying about other robots.

Fortunately, installing ADB is quick and straightforward.

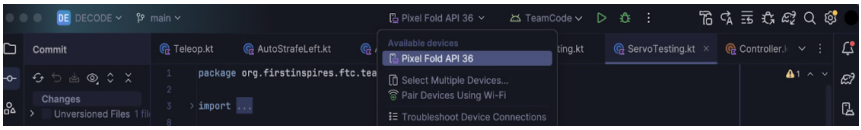
To begin, head to the official Android Platform Tools page on Google's developer site: **developer.android.com/tools/releases/platform-tools**. Look for the download links labeled "SDK Platform-Tools" for your operating system. After reading and agreeing to Google's terms, download the corresponding ZIP file.

Once the download is finished, extract the ZIP archive to a location you can easily access—many teams prefer creating a dedicated folder like **C:\adb on Windows** or placing it inside their home directory on **macOS or Linux**. Inside this extracted folder, you'll find the adb executable along with other useful platform tools.

On Windows, you may also want to add this folder to your system PATH. Doing so allows you to run **ADB** commands from any terminal window without navigating to the folder manually. While this step is optional, it greatly improves convenience when working with your **Robot Controller or Control Hub.**

If you're using **macOS or Linux**, the installation process is nearly identical: download, extract, and optionally add the tools to your PATH. A detailed guide for each platform can be found on the same page you downloaded the tools from.

And that's it—your computer is now equipped with ADB, to start using it just open a new terminal window and write adb to check if it is installed. Then plug in a battery in your control hub and connect to the wifi address of your robot.Write adb connect **192.168.43.1:5555** in the terminal's window and it should confirm that it's connected. Now you should see the control hub device in the available devices tab.



Other useful adb commands are **adb kill-server**, **adb disconnect** and **adb devices**. if you change the wifi address and then try to connect again, sometimes you may need to use the adb disconnect and adb kill-server commands and then try connecting again. Worst case, if this doesn't work, just power cycle(restart the robot) and disable and re-enable the wifi of your laptop.

If all the steps are followed correctly you should be able to **upload your code on the robot**, so go ahead and **press on the green row.**

1.4. What we program

In the FTC ecosystem, the **Control Hub** serves as the central brain of your robot—the place where all your code actually runs. Every decision your robot makes, every motor movement, and every sensor reading is processed through this device. When you write software in Android Studio, this is the hardware that ultimately executes your instructions during a match.

The Control Hub is often paired with an **Expansion Hub** when additional ports are needed. These two devices are linked using two cables:

- one power cable, which supplies electrical energy to both hubs;
- one data cable, which allows them to communicate seamlessly as a unified control system.

To upload, run, and test your code, your robot must have a battery connected. Without power, the hubs cannot boot, and your computer won't be able to communicate with them through **ADB or the FTC Driver Station**.

Both the Control Hub and Expansion Hub provide connection ports for all the essential hardware components of your robot. This includes:

- **DC motors** for drive trains and mechanisms,- we can use them without an encoder for continuous rotation or with encoder if we need our motor to hold a position;

- **servos** for precise positional control;

- **sensors** such as distance sensors, IMUs, limit switches, cameras, and more.

1.5. Using the driver station and dashboard

For this step you should install the ftc robot controller and ftc driver station app on an android device. On your device go on google and paste this link **https://github.com/FIRST-Tech-Challenge/FtcRobotController/releases** and press install these two apk's:

Once installed you can connect to the robots' wifi and then open ftc driver station app.

Your robot code is organized into different programs called OpModes (TeleOp, Autonomous, or Test).

With the Driver Station, **you can**:

- view the list of all available OpModes on the robot;
- select the one you want to run, and start or stop it during a match or while testing;
- connect controllers for driver practice.

Before running an opmode you should **go to three dots**->Configure and then make a new configuration in which you can add your motors and servo names that you are going to be using in the code.

Dashboard is like a driver station but on your laptop. You already have it installed if you choose the quickstart, so while you are connected to the robot's wifi address you can navigate to **192.168.43.1:8080/dash** on your browser.

If you started with another project you could set it up in a few easy steps

Installation

Basic

Open **build.dependencies.gradle.** In the repositories section add maven **{ url = 'https://maven.brott.dev/' }**, and in the dependencies section add implementation '**com.acmerobotics.dashboard:dashboard:0.5.1**'.

Note: If you're using OpenRC or have non-standard SDK dependencies, add the following exclusion.

```
implementation('com.acmerobotics.dashboard:dashboard:0.5.1') {
   exclude group: 'org.firstinspires.ftc'
}
```

After every gradle change you should press the sync button and eventually build your code while connected to a wifi that has internet.

1.6. Programming servos and motors

Now that we have everything done we can get to the code writing.

Add a new kotlin class in the opmodes package and name it ServoTesting, here is a sample of the code.Note that var are variables that can be changed, and val are values that are constant.

```
package org.firstinspires.ftc.teamcode.opmodes.debug.auxiliar
import com.acmerobotics.dashboard.config.Config
import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode
import com.qualcomm.robotcore.eventloop.opmode.TeleOp
import com.qualcomm.robotcore.hardware.PwmControl.PwmRange
import com.qualcomm.robotcore.hardware.ServoImplEx
@Config// for live variable editing via dashboard
@TeleOp// the type of the opmode, @TeleOp or @Autonomous
class ServoTesting: LinearOpMode() {//defining the class
    companion object {//the variable that we want to edit
        @JvmField
        var servoPosition = 0.0// a double variable since servos have (-1.0,1.0) range
    }

    override fun runOpMode() {
        //this is what happens during init
        val servo = hardwareMap.get(ServoImplEx::class.java,  deviceName: "servo")//declaring the servo
        waitForStart()
        while (opModeIsActive() && !isStopRequested) {
            //this is what happens while running the opmode in loop
            servo.position = servoPosition//asigning the given value to the servo
        }
    }
}
```

After adding it, try uploading your code and testing it using a dashboard. You can change the value of the servoposition on the left side of the page.

```
@Config// for live variable editing via dashboard
@TeleOp
class BasicMotorTesting: LinearOpMode() {
    companion object {//the variable that we want to edit
        @JvmField
        var power = 0.0// a double variable since motors have (-1.0,1.0) range
    }

    override fun runOpMode() {
        //this is what happens during init
        val motor = hardwareMap.get(DcMotorEx::class.java,  deviceName: "motor")//declaring the motor
        motor.direction=DcMotorSimple.Direction.FORWARD//1 becomes -1 and vice versa

        waitForStart()
        while (opModeIsActive() && !isStopRequested) {
            //this is what happens while running the opmode in loop
            motor.power = power//asigning the given power to the motor
        }
    }
}
```

if you are going to be using encoders your code should be slightly different

```
package org.firstinspires.ftc.teamcode.opmodes.debug

import ...

@Config
@TeleOp(name = "Motor RunToPosition Test")
class MotorTesting : LinearOpMode() {

    companion object {
        @JvmField var targetPosition = 500        // encoder ticks
        @JvmField var motorPower = 0.5            // power for RUN_TO_POSITION
    }

    override fun runOpMode() {
        val log = Log(telemetry)
        val motor = hardwareMap.dcMotor.get("motor")

        motor.mode = DcMotor.RunMode.STOP_AND_RESET_ENCODER
        motor.mode = DcMotor.RunMode.RUN_USING_ENCODER

        waitForStart()

        while (opModeIsActive() && !isStopRequested) {
            motor.targetPosition = targetPosition
            motor.mode = DcMotor.RunMode.RUN_TO_POSITION
            motor.power = motorPower
        }
    }
}
```

Make sure you add the servos and the motors in your config and activate it before testing your opmode.

Great, now you know how to program servos and motors.

1.7. Telemetry usage

Telemetry is one of the most powerful tools you have as an FTC programmer—it's your real-time window into what your robot is thinking, sensing, and doing. While the robot runs, telemetry lets you display live data on the Driver Station screen so you can understand what is happening internally and diagnose issues quickly. Think of it as your robot's voice: every variable, sensor reading, motor position, or debug message can be sent back to you through telemetry.

In the early stages of programming, telemetry helps you confirm that motors are running in the correct direction, that sensors are wired properly, and that your code is behaving the way you expect. As your robot becomes more complex, telemetry becomes essential for monitoring encoder values, PID loops, state machines, vision results, and even timing information.

Here's a simple example of how telemetry is used inside an OpMode:

```
telemetry.addData( caption: "Motor Power", motor.power)
telemetry.addData( caption: "Position", motor.currentPosition)
telemetry.update()
```

ddData() sends information to the Driver Station, while **update()** refreshes the display. Without the update call, telemetry will not show any new values.

You can also format messages, send multiple lines at once, or print debug information only when certain conditions are met. Telemetry is flexible enough to support anything from quick tests to advanced diagnostics.

As you continue writing more sophisticated code—especially autonomous routines—telemetry becomes your best friend. When something goes wrong (and believe us, it *will*), the fastest way to track down the issue is by printing the internal values you're curious about.

Use telemetry often, use it early, and use it smartly. Your robot always knows what's going on—telemetry simply lets it tell **you**.

1.8. Drivetrain class

Every robot has a drivetrain, and 90% of these are mecanum so now that we know how to program a motor, it's time to write a class that consists of four. Go ahead and add new package to your project named subsystems where you'll then add the drivetrain class:

```
package org.firstinspires.ftc.teamcode.subsystems
import com.qualcomm.robotcore.hardware.DcMotor
import com.qualcomm.robotcore.hardware.DcMotorSimple
import com.qualcomm.robotcore.hardware.HardwareMap
import kotlin.math.max
import kotlin.math.abs

class Drivetrain(hardwareMap: HardwareMap) {
    private val frontLeft = hardwareMap[DcMotor::class.java, "frontLeft"]
    private val frontRight = hardwareMap[DcMotor::class.java, "frontRight"]
    private val backLeft = hardwareMap[DcMotor::class.java, "backLeft"]
    private val backRight = hardwareMap[DcMotor::class.java, "backRight"]
    init {
        // Reverse left side for correct direction
        listOf(frontLeft, backLeft).forEach {
            it.direction = DcMotorSimple.Direction.REVERSE
            it.zeroPowerBehavior = DcMotor.ZeroPowerBehavior.BRAKE
        }
```

```
        listOf(frontRight, backRight).forEach {
            it.zeroPowerBehavior = DcMotor.ZeroPowerBehavior.BRAKE
        }
    }
    fun drive(x: Double, y: Double, rx: Double) {//this will be used in the loop
        val fl = y + x + rx//each value is the power of the respective motor
        val fr = y - x - rx//-> fr(front right)
        val bl = y - x + rx
        val br = y + x - rx
        val maxPower = max( a: 1.0, max(max(abs(fl), abs(fr)), max(abs(bl), abs(br))))
        frontLeft.power = fl / maxPower
        frontRight.power = fr / maxPower
        backLeft.power = bl / maxPower
        backRight.power = br / maxPower
    }
}
```

- the drive function will be called in the loop

Now it's time for you to give life to your class and write an actual op-mode, and then drive around your chassis.Add this DrivetrainTesting class in your opmodes package.

```
package org.firstinspires.ftc.teamcode.opmodes.debug

import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode
import com.qualcomm.robotcore.eventloop.opmode.TeleOp
import org.firstinspires.ftc.teamcode.subsystems.Drivetrain

@TeleOp
class DrivetrainTesting : LinearOpMode() {
    //declaring a variable that is the type Drivetrain
    private lateinit var drive: Drivetrain
    override fun runOpMode() {
        drive = Drivetrain(hardwareMap)//the drivetrain syncs the motors from the
        //hardwaremap with the actual running motors so you should update your config

        waitForStart()

        while (opModeIsActive()) {
            //the values that are given by the driver
            val y = -gamepad1.left_stick_y.toDouble()
            val x = gamepad1.left_stick_x.toDouble()
            val rx = gamepad1.right_stick_x.toDouble()

            drive.drive(x, y, rx)//this updates the power of the motors every tick
        }
    }
}
```

Now you can upload your code, navigate to the dashboard in your browser, connect a controller and test your chassis.Note that after connecting your controller you should press start+a/b.

1.9. Claw Class

Here is a code sample for a claw that you can later on edit and customize. This could also be a template for other subsystems that consist of servos.

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

import com.qualcomm.robotcore.hardware.HardwareMap
import com.qualcomm.robotcore.hardware.PwmControl.PwmRange
import com.qualcomm.robotcore.hardware.ServoImplEx
class Claw(hardwareMap: HardwareMap) {

    private val OPEN_POS = 0.0
    private val CLOSE_POS = 0.0
    private val servo: ServoImplEx =
        hardwareMap.get(ServoImplEx::class.java,  deviceName: "servoClaw")

    var isOpen = true
    fun setPosition(position: Double) {
        servo.position = position
    }
    fun close() {
        setPosition(CLOSE_POS)
        isOpen = false
    }
    fun open() {
        setPosition(OPEN_POS)
        isOpen = true
    }
    fun openFull() {
        setPosition(OPEN_POS)
        isOpen = true
    }
}
```

This can also be easily implemented into a ClawTesting opmode:

```
package org.firstinspires.ftc.teamcode.opmodes.debug

import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode
import com.qualcomm.robotcore.eventloop.opmode.TeleOp
import org.firstinspires.ftc.teamcode.subsystems.Claw

@TeleOp

class ClawTesting : LinearOpMode() {
    private lateinit var claw: Claw

    override fun runOpMode() {
        claw = Claw(hardwareMap)

        waitForStart()

        while (opModeIsActive()) {
            if (gamepad1.a) claw.open()
            if (gamepad1.b) claw.close()
        }
    }
}
```

1.10. Sample Teleop

In a ftc match there are two periods, teleop and auto. During the teleop period two drivers control the robot to complete tasks. You finally achieved all the knowledge to write a basic teleop.Right here is a simple code sample that handles the drivetrain and claw inputs.

```
package org.firstinspires.ftc.teamcode.opmodes.debug

import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode
import com.qualcomm.robotcore.eventloop.opmode.TeleOp
import org.firstinspires.ftc.teamcode.subsystems.Claw
import org.firstinspires.ftc.teamcode.subsystems.Drivetrain

@TeleOp

class Teleop : LinearOpMode() {
    private lateinit var claw: Claw
    private lateinit var drive:Drivetrain
    override fun runOpMode() {
        claw = Claw(hardwareMap)
        drive= Drivetrain(hardwareMap)
        waitForStart()

        while (opModeIsActive()) {
            //claw inputs
            if (gamepad1.a) claw.open()
            if (gamepad1.b) claw.close()

            //drivetrain inputs
            val y = -gamepad1.left_stick_y.toDouble()
            val x = gamepad1.left_stick_x.toDouble()
            val rx = gamepad1.right_stick_x.toDouble()

            drive.drive(x, y, rx)
        }
    }
}
```

Done! You just wrote your first teleop. For further development you can add your own subsystems such as an Extendo, Lift, or Linkage.

1.11. The autonomous period

The autonomous period is where FTC programming begins to feel truly advanced—and truly exciting. During these 30 seconds, your robot must operate **entirely on its own**, without any input from drivers. To make this possible, your robot must know:

- *where it is on the field;*
- *where it needs to go, and how to get there accurately.*

This is where **motion planning** comes in.

A **motion planner** is an open-source library that handles robot localization (tracking your position on the field) and trajectory generation (computing smooth, precise paths for your robot). Instead of hard-coding motor powers or guessing distances, a motion planner lets you command your robot using real-world units like inches, degrees, and coordinates.

In this guide, we'll be using **Pedro Pathing**, a motion planning library widely used in the FTC community for its:

- excellent precision;
- simplicity;
- easy integration;
- powerful trajectory features;
- and reliable localization during the entire autonomous period.

With **Pedro Pathing**, your robot can move smoothly, correct its position, follow curves, and **perform complex tasks with confidence** - even in a tight 30-second autonomous window.

For this step you'll need a localizer: go to **2.3** to pick your localizer.

Adding Pedro Pathing to Your Project

Here's a simple, clean process to get Pedro Pathing installed inside your FTC codebase.

Step 1: Download the Library

Go to the Pedro Pathing GitHub repository and download the latest release of the library.

(You will usually download a folder named something like PedroPathing or a .zip file with the library's source.)

Step 2: Place It Into Your TeamCode Module

Inside Android Studio:

1. Open your project.
2. Navigate to:
   TeamCode/src/main/java/org/firstinspires/ftc/teamcode/

3. Create a new folder called pedroPathing or localization (your choice).
4. Copy all the Pedro Pathing files into that folder.

Your project tree should now include the Pedro Pathing classes alongside your subsystems.

Step 3: Add Dependencies to build.gradle If Requiredv

Some versions of Pedro Pathing use RoadRunner math utilities or require Kotlin.

Make sure your TeamCode/build.gradle includes:

```
gradle

implementation 'org.jetbrains.kotlin:kotlin-stdlib:1.8.0'
```

and ensure Kotlin is enabled in your FTC project (which you already have).

Step 4: Sync and Rebuild Step 4: Sync and Rebuild

Click:

File → Sync Project with Gradle Files

Android Studio will rebuild your project, registering the new library automatically.

Step 5: Constants and tuning

Before starting the actual writing you have to configure your constants, pick your localizer and tune it.All the required steps are right here and should be followed carefully, step by step https://pedropathing.com/.Come back after reading all the documentation.

1.12. Just Write

Now that you understand the basics, it's time to do the most important (and honestly, the most exciting) part of learning to program: **just write code**. Start experimenting, breaking things, fixing them, and trying out every idea that pops into your mind. This is where your creativity becomes the driving force behind your robot.

Don't worry if things don't work right away—**they won't**. In fact, around 90% of the time your code will fail, crash, behave strangely, or just refuse to run at all. This is completely normal and happens to everyone, from beginners to professional developers.

But here's the magic: **that remaining 10%—when your code finally works, when the robot moves exactly how you imagined, when everything clicks—that moment is unbelievably rewarding.** It's the feeling that makes all the debugging, searching, rewriting, and experimenting absolutely worth it.

Programming isn't about writing perfect code on the first try. It's about building, breaking, learning, and improving. So jump in, write boldly, try new concepts, play with different approaches, and watch your ideas slowly evolve into real, functioning systems on your robot.

Every line of code you write gets you one step closer to becoming a stronger, more confident programmer.

2.0 Tailoring

This section moves from the Walkthrough's "how-to" into "how and why." It focuses on the architecture, tools, and engineering practices that let a team turn working code into reliable, competitive robot software. The material is intended for programmers who know the basics and want to structure, tune, and scale their codebase.

Contents:

2.1 Version control

Why it matters

- Tracks history, enables rollbacks, and supports collaboration.
- Facilitates code review and reliable releases for competition builds.

Recommendations

- Use Git with a hosted provider (GitHub, GitLab).
- Branching model: trunk-based or short-lived feature branches.
- Keep `main`/`master` stable. Create `feature/*` or `bugfix/*` branches for work.
- Commit messages: short summary line + optional body explaining why.
- Use `.gitignore` tuned for Android/FTC (ignore build/, .gradle/, .idea/, local.properties).
- Tag releases (v2025.1-auton-tested) for known-good competition builds.
- Protect `main` with branch protection rules and require PR reviews before merge.

Practices

- Open small pull requests; include screenshots, logs, or a short test plan.
- Code reviews: check architecture, tests, and telemetry clarity—don't only check syntax.
- Use CI to run build and lint tasks before merging.

Example .gitignore entries:

```text
/.gradle/
/build/
/local.properties
/.idea/
/*.iml
```

2.2 Gradle & build tooling

Why it matters

- Builds, dependency management, and packaging OpModes into Android APKs.

Key concepts

- Gradle wrapper (`gradlew`) ensures reproducible builds.
- `build.gradle(.kts)` in the app module configures compileSdk, dependencies, and ProGuard/R8 rules.
- Use dependency versions centrally and document changes.

Practical tips

- Keep the FTC SDK version pinned; document upgrade steps.
- Use flavors or build types if you need different configs (e.g., `debug`/`release`).

- Add custom Gradle tasks for common ops (generate docs, run static analysis).
- Keep builds fast: enable Gradle daemon, configure parallel workers, avoid heavyweight tasks on every change.
- Cache generated artifacts and consider CI runners with Android SDK preinstalled.

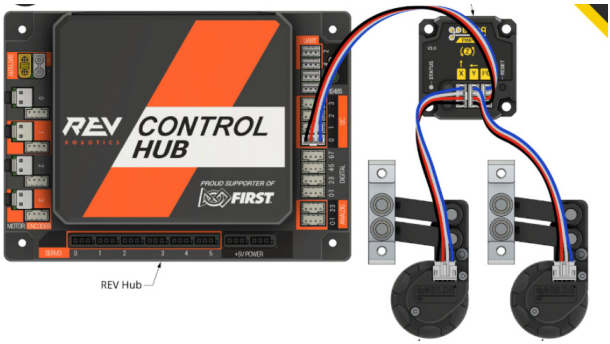
Useful tasks

- `./gradlew installDebug` — installs debug APK to the connected device.
- `./gradlew lint` — run static analysis.
- Create tasks that export a "competition" APK with settings optimized for matches.

2.3 Localizers (pose estimation)

Goal

- Estimate the robot's pose (x, y, heading) reliably in the field coordinate frame.



Common approaches

- Odometry-only: encoders on three tracking wheels (two lateral, one perpendicular)- good baseline.
- IMU fusion: combine IMU heading with odometry to handle slippage and drift.
- Sensor fusion: combine vision (AprilTags), range sensors, and odometry (Kalman/Extended Kalman or complementary filters).

Design considerations

- Frame of reference: choose field-centric coordinates and keep conversions central.
- Update rate: localizer should run at a constant, reasonably high rate (e.g., 50-200 Hz) depending on your planner.
- Latency: minimize and account for it in control loops.
- Error modeling: characterize encoder ticks → distance, IMU noise, and slip.

Implementation sketch (Kotlin-style)

```kotlin
interface Localizer {
  fun getPose(): Pose2d
  fun update() // read sensors and integrate
  fun reset(pose: Pose2d)
}
```

Tips

- Test with short runs, lateral movements, and rotations to find systematic errors (wheelbase, lateral offset).
- Add diagnostics: log wheel deltas, IMU headings, and final pose errors.

2.4 Motion planners & trajectory followers

What they do

- Motion planner: generate a feasible path/trajectory from start to goal considering kinematic/physical limits.
- Trajectory follower: execute the planned path using closed-loop controllers.

Types of planners

- Geometric planners (splines, lines) for smooth paths.
- Kinematic motion profiles (trapezoidal, S-curve) for 1D motion.
- Full trajectory generation (profiles for x,y,heading) used by libraries like Road Runner.

Controllers & profiles

- PID controllers for position/heading.
- Feedforward (kV, kA, kStatic) to overcome predictable dynamics.
- Motion profiling: control velocity/acceleration/jerk to respect hardware limits and maintain stability.

Popular libraries

- Road Runner — widely used for FTC trajectories and localizers.
- Pure Pursuit — simpler path-following algorithm good for some robots.

Tuning checklist

- Measure physical constraints: max velocity and acceleration under load.
- Tune feedforward first (kV from velocity test), then PID.
- Validate with short trajectories before full auton sequences.

2.5 Autonomous strategies

Design patterns

- State machines: deterministic flow that is easy to debug.
- Command/Action scheduling: each action (drive, turn, intake) is a command with start/stop and completion conditions.
- Behavior tree or simple sequencer for more complex branching.

Robustness

- Timeouts for every action.
- Sensor checks (e.g., ensure goals reached within tolerances).
- Safe abort behavior: stop motors, set servos to safe positions.

Concurrency

- Allow parallel actions where safe (move + intake).
- Keep shared resource arbitration explicit (who drives the drivetrain?).

Example command pseudo-structure

```kotlin
class DriveToPose(val target: Pose2d): Command {
 override fun init() = planner.followTrajectoryAsync(planner.trajectoryTo(target))
  override fun update() = planner.update()
  override fun isFinished() = planner.isIdle()
}
```

Testing strategies

- Simulate trajectories on a localizer beforehand (plot paths).
- Record telemetry from test runs and replay logs to find systematic errors.

2.6 TeleOp strategies

Principles

- Prioritize driver intention: map controls so drivers can reliably predict robot behavior.
- Keep toggles and stateful controls simple and well-documented.

Common techniques

- Deadzones and response shaping: apply deadzone and optionally exponential scaling to sticks for precision.
- Field-centric vs robot-centric control:
- Field-centric: convert joystick vectors to field frame using gyro heading (makes direction intuitive).
- Edge detection for toggles: act on button press transitions rather than held state.
- Mode switches: speed multipliers (precision mode, turbo mode) and subsystem locks.

Example control smoothing

```kotlin
fun deadzone(input: Double, threshold: Double) =
   if (kotlin.math.abs(input) < threshold) 0.0 else input

fun expo(input: Double, exponent: Double) = Math.signum(input) * Math.pow(Math.abs(input), exponent)
```

Driver development

- Provide in-op mode telemetry to show current mode and important sensor values.
- Build a driver practice mode with simulated field elements.

2.7 Camera pipelines & vision

Photo of the Limelight camera

Use cases

- AprilTag detection for field-relative positioning.
- Object detection (cones, rings) for decision making.
- Line detection for alignment.

Architecture

- Camera acquisition → image transform → detection → postprocessing → result broadcast (telemetry, shared object).
- Keep heavy processing off the main robot thread. Use an executor or async pipeline.

OpenCV tips

- Start with simple color segmentation in HSV, then refine with morphological operations.
- Calibrate thresholds under the team's actual lighting conditions; use a tuning UI.
- Use contour filtering (area, aspect ratio) to reduce false positives.

AprilTag example (Kotlin pseudo)

```kotlin
class AprilTagPipeline: CameraPipeline {
  override fun processFrame(frame: Mat): List<Detection> {
    // convert to grayscale, run detector, return tag poses
  }
}
```

Performance

- Use cropping and downscaling to keep frame size modest.
- Consider running detection at lower FPS (e.g., 10-20) and interpolate results.

2.8 Programming sensors

Sensor types and tips

- Encoders: calibrate ticks → distance; watch for gear backlash.
- IMU: calibrate at startup; use fused-heading where available; filter gyro noise.
- Color & distance sensors: calibrate for reflectance and ambient light; use multiple readings and median filters.
- Range sensors: apply smoothing and handle timeouts.

Filtering & fusion

- Simple filters: moving average, median, exponential smoothing.
- Complementary/Kalman filters for pose: use odometry for short-term accuracy and vision for long-term correction.

Calibration

- Always document and automate calibration steps (scripts or opmodes).
- Store calibration constants in a central config file or preferences.

Diagnostics

- Add a sensor dashboard showing raw and processed values.
- Log sensor values to CSV during test runs for offline analysis.

2.9 Other useful tools

Telemetry & logging

- Use structured logging for events and errors.
- Telemetry should be concise during matches; verbose options for debugging builds.

Dashboards and simulators

- FTC Dashboard — live telemetry, camera preview, and tuning UI.
- Local simulation frameworks (Road Runner simulator) and unit-test-friendly components.
- MeepMeepTesting- simulator for autonomous trajectories
- Scrcpy - mirrors your driver station on your laptop-useful for editing your config fast

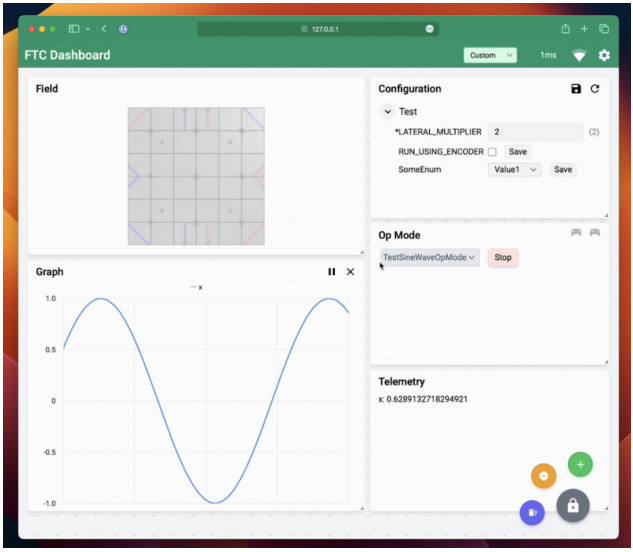
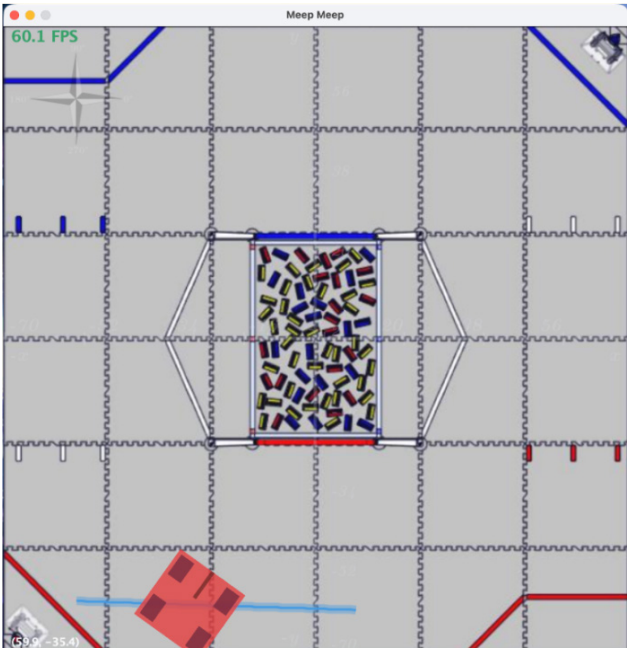


Photo from FTC Dashboard Template



Meep Meep Trajectory Simulator

Static analysis & testing

- Run linters and Kotlin/Java formatters.
- Write unit tests for pure logic (trajectory math, state machines).
- Integration tests: hardware-in-the-loop where possible.

Performance tools

- Android Studio profiler to check CPU and memory.
- Measure loop timings and garbage collection; avoid frequent allocations in tight loops.

2.10 Ways of learning

Practical approaches

- Read, run, and dissect working teams' repositories.
- Hands-on: build small projects that exercise a single subsystem (e.g., PID-based slide).
- Pair programming with drivers and mentors to align code with operator needs.

Resources

- FTC SDK docs and sample code.
- Road Runner docs and examples.
- FTCLib for helpful abstractions and hardware wrappers.
- Community: FTC Discord servers, local mentors, and official forums.

Learning plan

- Foundations: motor & sensor basics, event loops, unit conversions.
- Intermediate: localizers, planners, command patterns.
- Advanced: modeling robot dynamics and automated tuning.

2.11 How to improve (team & code)

Code quality

- Use static analysis and consistent style.
- Refactor: keep subsystems small and single-responsibility.
- Write small, testable functions; minimize global state.

Process

- Daily or weekly code reviews; share demo runs.
- Keep a "competition" branch for stability; only merge well-tested changes.
- Practice runs with drivers under match-like conditions.

Tuning & iteration

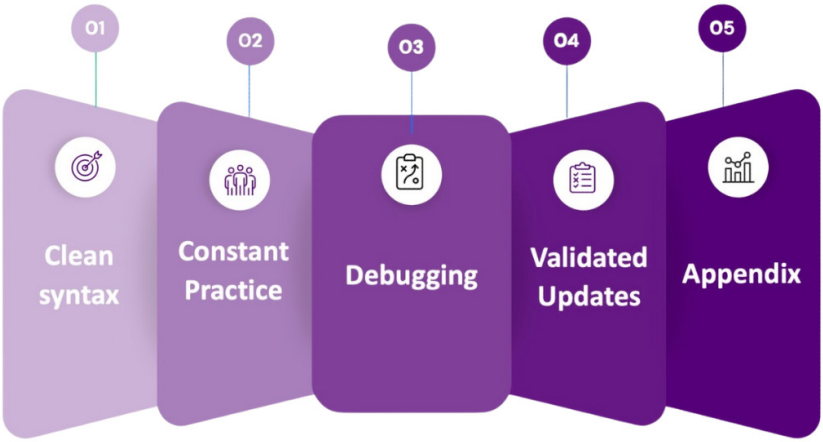
- Collect telemetry data and make tuning part of your workflow.
- Use repeatable test scripts to validate controller changes.
- Track changes with versioned tags and changelogs.

Competition-readiness checklist (short)

- Core subsystems tested and within tolerances.
- Autonomous routines run reliably with timeouts.
- TeleOp control mapping validated with drivers.
- Safety behaviors confirmed (estop, safe defaults).
- Backup build on phone/robot and a rollback commit/tag.

Appendix: Patterns and templates

- Command pattern for actions (init, update, isFinished, done).
- "Subsystem" interface: encapsulates hardware and exposes commands.
- Central scheduler: coordinates commands and handles resource conflicts.



Closing notes

- The Tailoring section is intentionally pragmatic: pick a small set of patterns and tools, get comfortable with them, then expand.
- Prioritize repeatability, testability, and clear telemetry—these give you the confidence to iterate quickly and perform under pressure.

Impresions