RoadRunner 1.0

RoadRunner 1.0 Actions Guide

This document explains how to use the **Action** system in RoadRunner 1.0 to build advanced autonomous routines.

What is an Action?

An `Action` is a unit of work that the robot performs. It can be moving a servo, driving a path, or waiting.

In `runOpMode`, you execute actions using:

```
Actions.runBlocking(myAction);
```

---

1. SequentialAction

Runs actions **one after another**. The next action starts only when the previous one finishes.

Use Case: Drive to a location, THEN drop a pixel, THEN park.

```
Action mySequence = new SequentialAction(
    drive.actionBuilder(startPose).lineToX(10).build(), // Step 1: Drive
    intake.spinIntakeAction(0),                         // Step 2: Stop
Intake
    outtake.setLiftPowerAction(0.5)                     // Step 3: Raise
Lift
);
```

---

2. ParallelAction

Runs actions **at the same time**.

- **Ends when**: The **LONGEST** action finishes.
- **Use Case**: Lifting the slides *while* driving effectively.

```
Action myParallel = new ParallelAction(
    drive.actionBuilder(startPose).lineToX(20).build(), // Takes 3 seconds
    outtake.setLiftPowerAction(1.0)                     // Takes 1 second
);
// Result: Robot drives and lifts simultaneously. The action finishes after
3 seconds.
```

---

3. RaceAction

Runs actions **at the same time**.

- **Ends when**: The **FIRST (shortest)** action finishes.
- **Use Case**: "Spin the intake *until* the sensor detects a pixel."

```
Action myRace = new RaceAction(
    intake.spinIntakeAction(1.0),                  // Infinite action (never
stops naturally)
    new WaitUntilPixelDetectedAction()             // Finishes when sensor
sees pixel
);
// Result: Intake spins, but as soon as the sensor triggers, the whole
action stops (cutting power to intake).
```

---

4. SleepAction

Waits for a specific mount of time. Useful for letting servos settle or waiting for a partner.

```
new SleepAction(1.5); // Waits for 1.5 seconds
```

4.5. NullAction

An action that does **nothing**.

- **Use Case**: A placeholder for testing sequences without deleting code, or as a default "idle" action.

```
new NullAction();
```

---

5. Instant Actions (Lambdas)

Sometimes you just want to set a servo position instantly or toggle a generic variable without writing a whole class. You can use a "Lambda" (a small anonymous function).

```
// Logic: "Do this one thing, then return false (I'm done)"
(packet) -> {
    robot.servo.setPosition(0.5);
    return false;
}
```

---

6. Writing Custom Actions (The "run" method) - IMPORTANT

The text you shared highlights a critical concept: **The return value of the `run()` method.**

- `return true;` -> **"I am still working."** RoadRunner will keep calling `run()` repeatedly.

- `return false;` -> **"I am done."** RoadRunner stops this action and moves to the next one.

Example: "Set and Forget" (Instant)

Use this for simple things like setting a servo position or motor power.

```
public Action setPower(double power) {
    return packet -> {
        motor.setPower(power);
        return false; // Done immediately
    };
}
```

Alternative (using `InstantAction` class found in docs):

You can also use the explicit class, which doesn't need a return value:

```
public Action setPower(double power) {
    return new InstantAction(() -> motor.setPower(power));
}
```

Example: "Wait Until" (Duration)

Use this if you want the action to **BLOCK** until a condition is met (like "Wait for slides to reach top").

```
public class WaitForSlides implements Action {
    @Override
    public boolean run(@NonNull TelemetryPacket packet) {
        // Send telemetry to dashboard
```

```
        packet.put("Slide Pos", motor.getCurrentPosition());
        // Return true if we are NOT at the target yet
        // Return false (done) if we ARE at the target (> 1000)
        return motor.getCurrentPosition() < 1000;
    }
}
```

Complex Example: The "AlbastruBara" Routine

Here is how you combine them to make a pro-level routine:

```
Actions.runBlocking(
    new SequentialAction(
        // 1. Drive to Spike Mark while prepping intake
        new ParallelAction(
            drive.actionBuilder(startPose).lineToY(24).build(),
            intake.setWristPosition(0.5)
        ),

        // 2. Drop Purple Pixel
        intake.openClawAction(),
        new SleepAction(0.5),

        // 3. Drive to Backboard
        drive.actionBuilder(new Pose2d(...)).splineTo(...).build(),
```

```
        // 4. Raise Slides AND Extend Arm simultaneously
        new ParallelAction(
            lift.setHeightAction(1000),
            arm.setExtensionAction(500)
        ),

        // 5. Score Yellow Pixel
        outtake.openDumpAction()
    )
);
```

---

7. Trajectory Builder Reference

How to tell the robot where to go.

Path Primitives

Assuming `beginPose` is at origin $(0, 0)$ with heading $\pi/6$.

- `lineToX(48)` : Move straight to X=48 (maintain heading).
- `lineToY(36)` : Move straight to Y=36 (maintain heading).
- `splineTo(new Vector2d(48, 48), Math.PI / 2)` : Curved path to target position. The robot will arrive facing 90° ($\pi/2$).

Heading Primitives

Assuming `beginPose` is origin with heading $\pi/2$.

- **Tangent Heading (Default)**: Robot faces the direction it is moving.

```
    .setTangent(0)
```

```
    .splineTo(new Vector2d(48, 48), Math.PI / 2)
```

- **Constant Heading**: Robot keeps facing the same direction while moving.

```
    .splineToConstantHeading(new Vector2d(48, 48), Math.PI / 2)
```

- **Linear Heading**: Robot rotates linearly from start heading to end heading.

```
    .splineToLinearHeading(new Pose2d(48, 48, 0), Math.PI / 2)
```

- **Spline Heading**: Complex heading interpolation (rarely needed).

```
    .splineToSplineHeading(new Pose2d(48, 48, 0), Math.PI / 2)
```

---

8. Cancellation & Advanced Control

Sometimes you need to stop a trajectory early (e.g., "Stop when pixel is Intaked").

Abrupt Cancellation

Stop immediately. This can be jarring but is fast.

Step 1: Create a FailoverAction

A wrapper that runs a `mainAction` until triggered to switch to a `failoverAction` (like stopping).

```
public class FailoverAction implements Action {

    private final Action mainAction;

    private final Action failoverAction;

    private boolean failedOver = false;
```

```
    public FailoverAction(Action mainAction, Action failoverAction) {
        this.mainAction = mainAction;
        this.failoverAction = failoverAction;
    }

    @Override
    public boolean run(@NonNull TelemetryPacket packet) {
        if (failedOver) return failoverAction.run(packet);
        return mainAction.run(packet);
    }

    public void failover() { derived = true; }
}
```

Step 2: Use it

```
Action safeTrajectory = new FailoverAction(
    drive.actionBuilder(pose).lineToX(10).build(), // Main: Drive
    new InstantAction(() -> drive.setDrivePowers(new PoseVelocity2d(new
Vector2d(0,0), 0))) // Failover: Stop
);
```

---

9. Teleop Actions

You can use Actions in Teleop too! This is great for automating complex sequences (like "Grab Pixel") with a single button press.

Pattern:

1. Maintain a list of `runningActions`.
2. In your loop, update all actions. Remove finished ones.

```
List<Action> runningActions = new ArrayList<>();


// In loop():
if (gamepad1.a) {
    runningActions.add(new SequentialAction(
        intake.spinAction(1),
        new SleepAction(0.5),
        intake.stopAction()
    ));
}


// Update Loop
List<Action> newActions = new ArrayList<>();
for (Action action : runningActions) {
    action.preview(packet.fieldOverlay());
    if (action.run(packet)) { // returns true if still running
        newActions.add(action);
    }
}
runningActions = newActions;
```

---

10. Constraints & Pose Mapping

Pose Mapping (Alliance Compatibility)

Write one auto, flip it for the other side.

```
// Mirror across X-axis

new TrajectoryBuilder(...,

    pose -> new Pose2dDual<>(pose.position.x, -pose.position.y, -
pose.heading)

);
```

Variable Constraints (Slow Down in Zones)

Slow down the robot in specific areas (e.g., near the backdrop).

```
VelConstraint slowNearBackdrop = (robotPose, path, disp) -> {

    if (robotPose.position.x > 24.0) { // If X > 24 (Near board)

        return 20.0; // Slow speed

    } else {

        return 50.0; // Fast speed

    }

};
```

Per-Segment Constraints

Override constraints for just one part of the path.

```
.splineTo(new Vector2d(48, 0), -Math.PI / 2,

    new TranslationalVelConstraint(20.0) // Slow down just for this spline
```

```
)
```

---

11. Full Autonomous Example (CENTERSTAGE Style)

Building a complete auto from scratch involves several steps.

Step 1: Define Mechanism Actions

Create classes for your hardware (Lift, Claw) and implementing `Action` classes inside them.

```
public class Lift {
    private DcMotorEx lift;
    public Lift(HardwareMap hw) { ... }
    public class LiftUp implements Action {
        private boolean initialized = false;
        @Override
        public boolean run(@NonNull TelemetryPacket packet) {
            if (!initialized) { lift.setPower(0.8); initialized = true; }
            if (lift.getCurrentPosition() < 3000) return true; // Keep
running
            else { lift.setPower(0); return false; } // Stop
        }
    }
    public Action liftUp() { return new LiftUp(); }
}
```

Step 2: Build the Auto Class

```java
@Autonomous(name = "BlueSideTest")
public class BlueSideTestAuto extends LinearOpMode {
    @Override
    public void runOpMode() {
        // 1. Setup
        MecanumDrive drive = new MecanumDrive(hardwareMap, new Pose2d(11.8, 61.7, Math.toRadians(90)));
        Lift lift = new Lift(hardwareMap);
        Claw claw = new Claw(hardwareMap);


        // 2. Build Trajectories (INIT Phase)
        Action traj1 = drive.actionBuilder(drive.pose)
                .lineToY(33)
                .setTangent(Math.toRadians(90))
                .lineToX(32)
                .build();
        // 3. Init Loop
        while(!isStopRequested() && !opModeIsActive()) {
            telemetry.addData("Status", "Waiting for Start");
            telemetry.update();
        }
        // 4. Run (START Phase)
        if (isStopRequested()) return;
        Actions.runBlocking(
            new SequentialAction(
```

```
                traj1,
                lift.liftUp(),
                claw.openClaw(),
                lift.liftDown()
            )
        );
    }
}
```

---

12. Path Continuity & Smoothness

To keep the robot moving smoothly without stopping, your path must be "Continuous".

1. Tangent Continuity

The robot cannot instantly change direction (velocity vector).

Bad:

```
.lineToX(24) // Moving East
.setTangent(Math.PI/2) // Suddenly tries to move North
.lineToY(24)
```

Fix: Use splineTo or turn actions to blend the motion.

2. Heading Continuity

The robot cannot instantly change its rotation speed.

Bad:

```
.lineToXLinearHeading(24, Math.PI/2) // Spinning one way
```

```
.lineToXLinearHeading(48, 3*Math.PI/4) // Suddenly spinning at different
speed
```

Fix: Use `splineToSplineHeading` to smooth out rotation changes.

---

13. Tangents vs Headings

It is critical to understand the difference:

- **Tangent**: The direction the robot is *moving* (Velocity Vector).
- **Heading**: The direction the robot is *facing* (Orientation).

By default, **Heading follows Tangent** (Tank drive style).

For Mecanum, they are decoupled.

- `.lineToX(48)` : Tangent is East, Heading stays constant.
- `.splineTo(pos, tangent)` : You are setting the *Tangent* (arrival angle of the curve).

---

14. FTCLib Integration

If you use FTCLib's Command-based system, you can wrap Actions into Commands.

```
public class ActionCommand implements Command {

    private final Action action;

    private final Set<Subsystem> requirements;

    private boolean finished = false;


    public ActionCommand(Action action, Set<Subsystem> requirements) {

        this.action = action;

        this.requirements = requirements;

    }
```

```
    @Override
    public void execute() {
        TelemetryPacket packet = new TelemetryPacket();
        action.preview(packet.fieldOverlay());
        finished = !action.run(packet); // Run until false
        FtcDashboard.getInstance().sendTelemetryPacket(packet);
    }

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

Usage:

```
new ActionCommand(drive.actionBuilder(pose).lineToX(10).build(),
driveSubsystem);
```

Explanation: How it Works

This wrapper translates between the two systems:

1. `implements Command`: Tells FTCLib to treat the Action as a Command.
2. `requirements`: Locks the subsystem (preventing conflicts with TeleOp code).
3. `execute()`: Runs the RoadRunner Action loop.

* `!action.run(packet)`: RoadRunner returns `false` when done, but FTCLib expects `true`. The `!` flips it.

When to use this?

- **YES**: If you are using the **Command-Based** paradigm (`CommandOpMode`) and want to schedule trajectories.

- **NO**: If you are using `LinearOpMode` with `runBlocking()` (like `BasicAuto.java`).

---

15. Pro Tips & Common Pitfalls

1. NEVER Block the `run()` Method

The `run()` method runs inside the robot's main loop.

- **BAD**: Using `Thread.sleep()` or `while(...)` inside `run()`. This freezes the whole robot.
- **GOOD**: Check the condition, and return `true` immediately if not done.

```
    // BAD
    public boolean run(packet) {
        while(motor.isBusy()) { } // FREEZES ROBOT
        return false;
    }
    // GOOD
    public boolean run(packet) {
        if (motor.isBusy()) return true; // Returns immediately, checks
again next loop
        return false;
    }
```

2. Enable Bulk Reads (Critical for Speed)

Running many Actions at once requires many sensor reads. This slows down the loop.

Use `LynxModule` to read all sensor data at once per loop.

```
// In your init():
List<LynxModule> allHubs = hardwareMap.getAll(LynxModule.class);
```

```
for (LynxModule module : allHubs) {

    module.setBulkCachingMode(LynxModule.BulkCachingMode.AUTO);

}
```

3. Debugging with Dashboard

- **Packet put**: `packet.put("Status", "Waiting")` sends data to the Dashboard graph/telemetry.
- **Preview**: The `.preview()` method draws the action on the Dashboard field view. Use this to verify your logic visually!

End of Guide