Building Your First

Mecanum Drivetrain TeleOp

A Step-by-Step Guide for FTC Students

Learn how to program a robot with omnidirectional movement!

Welcome!

Congratulations on taking your first steps into FTC robotics programming! This guide will walk you through creating a TeleOp program for a mecanum drivetrain from start to finish. By the end, you'll have a fully functional robot that can move in any direction!

What you'll learn: How to set up motors, read gamepad inputs, understand the math behind mecanum movement, and write clean, professional code.

What you'll need: A robot with four mecanum wheels, the FTC Robot Controller app configured with four motors, and this guide!

1. What Is a Mecanum Drivetrain?

A mecanum drivetrain is special because it allows your robot to move in **any direction** without turning. Unlike regular wheels, mecanum wheels have small rollers at a 45-degree angle. This unique design creates something called **omnidirectional movement**.

What makes it special:

- Your robot can move forward, backward, left, right, and diagonally
- It can rotate in place without moving forward or backward
- It can combine movements—for example, moving forward while strafing left
- This gives you incredible maneuverability on the competition field!

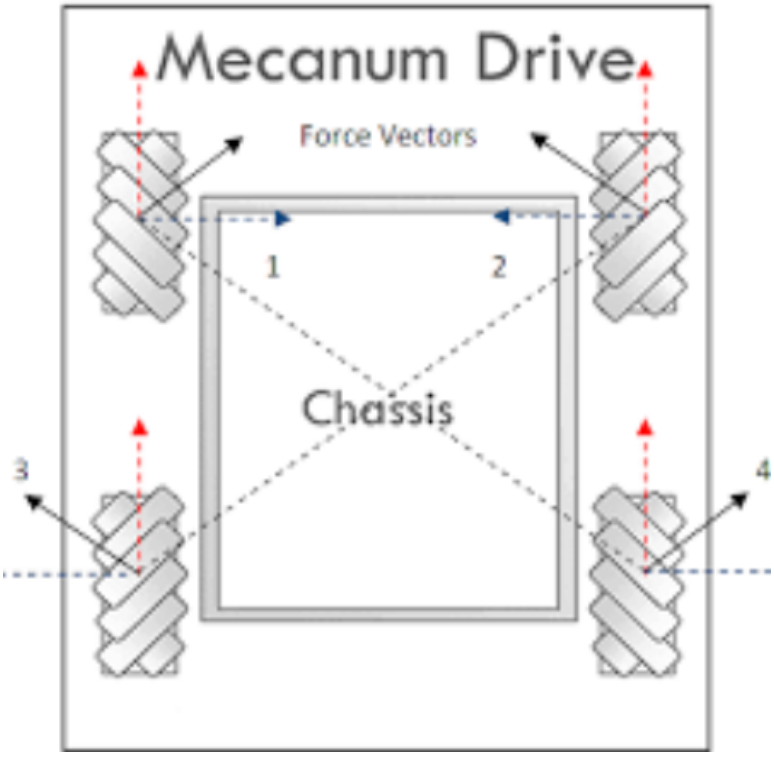

Figure 1: Mecanum Drivetrain Diagram - Notice how each wheel has angled rollers

Understanding the diagram: Look at the force vectors (arrows) in the diagram. The angled rollers on each wheel create forces in specific directions. When you combine the forces from all four wheels spinning at different speeds, you can move in any direction. The wheels are numbered 1-4 (Front Left, Front Right, Back Left, Back Right), and you'll notice that opposite wheels have their rollers angled in opposite directions.

2. Hardware Configuration

Before we write any code, we need to tell the Robot Controller app about our four motors. This is done through the **hardware configuration** process.

Step-by-step configuration:

1. **Open the Robot Controller app** on the Robot Controller phone
2. **Tap the three dots** in the upper right corner
3. **Select 'Configure Robot'**
4. **Create a new configuration** or edit your existing one
5. **Add four DC motors** and name them **exactly** as follows:

Motor Position	Exact Name	Hub Port
Front Left Motor	frontLeft	(example: port 0)
Front Right Motor	frontRight	(example: port 1)
Back Left Motor	backLeft	(example: port 2)
Back Right Motor	backRight	(example: port 3)

IMPORTANT: The names must match **exactly**—including capitalization! If you name a motor 'FrontLeft' instead of 'frontLeft', your code won't work.

Why some motors need to be reversed:

Look back at the mecanum diagram. Notice that the motors on the right side of the robot face the opposite direction from those on the left side. When you tell all motors to spin 'forward,' the right-side motors will actually make the robot spin in a circle! To fix this, we reverse the direction of the right-side motors in our code (coming up in step 5).

3. Adding Motor Variables to Your Class

Now we're ready to start coding! We need to create **variables** that will represent our four motors. Think of variables as labeled boxes where we store information.

What is DcMotor?

DcMotor is a special class provided by the FTC SDK (Software Development Kit). It represents a DC motor and gives us methods to control it, like setting its power or checking its position. When we write 'DcMotor frontLeft', we're creating a variable that will hold a reference to our actual physical motor.

Add this code at the top of your class:

```
package Teleops;

import com.qualcomm.robotcore.eventloop.opmode.OpMode;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.hardware.DcMotor;  // Add this import

@TeleOp(name = "Basic Drive Train", group = "Basic")
public class BasicDriveTrainTeleop extends OpMode {

    // Declare motor variables - add these four lines
    private DcMotor frontLeft;
    private DcMotor frontRight;
    private DcMotor backLeft;
    private DcMotor backRight;

    @Override
    public void init() {
        telemetry.addLine("Initialized");
        telemetry.update();
    }

    // ... rest of your code
}
```

Breaking it down:

• **import com.qualcomm.robotcore.hardware.DcMotor;** - This line tells Java that we want to use the DcMotor class
• **private** - This means only our class can access these variables
• **DcMotor** - The type of variable (a motor)
• **frontLeft, frontRight, backLeft, backRight** - The names we chose for our variables

4. Initializing Motors in the init() Method

Now we need to connect our variables to the actual motors on the robot. This happens in the **init()** method, which runs once when you press the INIT button on the Driver Station.

What is hardwareMap?

hardwareMap is a special object provided by the FTC system. It acts like a directory that connects the motor names you configured in the Robot Controller app to your code variables. When we write 'hardwareMap.get(DcMotor.class, "frontLeft")', we're saying: 'Find the motor named frontLeft in the hardware configuration and give me a reference to it.'

Update your init() method:

```
@Override
public void init() {
    // Initialize the motors using hardwareMap
    frontLeft = hardwareMap.get(DcMotor.class, "frontLeft");
    frontRight = hardwareMap.get(DcMotor.class, "frontRight");
    backLeft = hardwareMap.get(DcMotor.class, "backLeft");
    backRight = hardwareMap.get(DcMotor.class, "backRight");

    telemetry.addLine("Motors Initialized");
    telemetry.update();
}
```

Common mistake: If you see an error like 'Unable to find a hardware device with the name "frontLeft"', it means the name in your code doesn't match the name in your hardware configuration. Double-check both!

5. Setting Motor Directions

Remember how we talked about motors needing to spin in opposite directions? Now we'll fix that in code by **reversing** the right-side motors.

Why do we reverse motors?

Look at the mecanum diagram again. The motors on the right side of your robot are physically mounted facing the opposite direction from the left-side motors. If we don't reverse them, when you push the joystick forward, the left wheels will push the robot forward while the right wheels push it backward—making the robot spin instead of drive straight!

By calling **setDirection(DcMotor.Direction.REVERSE)** on the right-side motors, we tell them to spin in the opposite direction. Now when we set power to 1.0, all motors will work together to move the robot forward.

Add these lines to your init() method:

```
@Override
public void init() {
    // Initialize the motors
    frontLeft = hardwareMap.get(DcMotor.class, "frontLeft");
    frontRight = hardwareMap.get(DcMotor.class, "frontRight");
    backLeft = hardwareMap.get(DcMotor.class, "backLeft");
    backRight = hardwareMap.get(DcMotor.class, "backRight");

    // Set motor directions - reverse the right side
    frontLeft.setDirection(DcMotor.Direction.FORWARD);
    frontRight.setDirection(DcMotor.Direction.REVERSE);
    backLeft.setDirection(DcMotor.Direction.FORWARD);
    backRight.setDirection(DcMotor.Direction.REVERSE);

    telemetry.addLine("Motors Initialized and Configured");
    telemetry.update();
}
```

Pro tip: If your robot moves in unexpected directions during testing, you may need to swap which motors are reversed. Every robot is slightly different depending on how the motors are mounted!

6. Reading the Gamepad

Now for the fun part—controlling the robot! The FTC gamepad has two joysticks, and we'll use them to control movement. Let's understand what each joystick does.

Gamepad controls:

• **Left Stick Y-axis (up/down):** Controls forward and backward movement
• **Left Stick X-axis (left/right):** Controls strafing (sideways movement)
• **Right Stick X-axis (left/right):** Controls rotation (spinning in place)

The Y-axis trick:

Here's something weird about gamepads: when you push the stick UP, the Y value is **negative**, and when you pull it DOWN, the Y value is **positive**. This is backwards from what feels natural! So we multiply the Y value by -1 to flip it around. Now pushing up gives positive values (moving forward) and pulling down gives negative values (moving backward).

Add this code to your loop() method:

```
@Override
public void loop() {
    // Read the gamepad values
    double y = -gamepad1.left_stick_y;   // Forward/backward (inverted)
    double x = gamepad1.left_stick_x;    // Strafe left/right
    double rx = gamepad1.right_stick_x;  // Rotate left/right

    telemetry.addData("Y (forward)", y);
    telemetry.addData("X (strafe)", x);
    telemetry.addData("RX (rotate)", rx);
    telemetry.update();
}
```

The **telemetry** lines send data back to the Driver Station so you can see the joystick values. This is super helpful for debugging! When you move the joysticks, you'll see the numbers change on the Driver Station screen.

7. The Mecanum Math (Made Simple!)

This is where the magic happens! We need to convert the gamepad inputs into motor powers. Don't worry—the math looks complicated, but we'll break it down step by step.

The formulas:

```
// Calculate power for each motor
double frontLeftPower = y + x + rx;
double frontRightPower = y - x - rx;
double backLeftPower = y - x + rx;
double backRightPower = y + x - rx;
```

Understanding each term using the diagram:

The Y term (forward/backward):

All four motors get the SAME y value (positive or negative). Look at the diagram's forward movement arrows—all wheels contribute to moving forward by spinning in the same direction. When y is positive (joystick pushed up), all motors spin forward. When y is negative (joystick pulled down), all motors spin backward.

The X term (strafing left/right):

This is where mecanum gets interesting! Look at the diagram's strafe vectors. Notice how the roller angles create diagonal forces:
• **Front Left** and **Back Right** get **+x** (they push the robot right)
• **Front Right** and **Back Left** get **-x** (they push the robot left)
When these opposite forces combine, the robot slides sideways without rotating!

The RX term (rotation):

To spin in place, we need the left and right sides to spin in opposite directions:
• **Left motors** (front left, back left) get **+rx**
• **Right motors** (front right, back right) get **-rx**
This creates a turning force while the robot stays in the same spot.

Putting it all together:

Each motor's power is a **combination** of these three movements. That's why we add and subtract the terms! For example:
• If you push forward (y=1) and strafe right (x=0.5), the front left motor gets 1 + 0.5 = 1.5 (we'll fix

values over 1.0 in a moment)
• If you rotate right (rx=0.5) while going forward (y=1), the left motors get more power and the right motors get less power, creating a curved path

8. Setting Motor Power

We've calculated the power for each motor, but there's one problem: the values might be greater than 1.0 or less than -1.0. Motors only accept power values between -1.0 and 1.0, so we need to fix this!

What does motor power mean?

Motor power is a number between -1.0 and 1.0:

- **1.0** means full speed forward
- **0.0** means stopped
- **-1.0** means full speed backward
- **0.5** means half speed forward

Any value outside this range will cause an error!

Handling values over 1.0:

If we push forward (y=1) AND strafe right (x=1) at the same time, the front left motor would try to get a power of 1 + 1 = 2.0—which is too much! The solution is to find the largest power value, and if it's over 1.0, we divide all motor powers by that value. This keeps the **ratio** between motors correct while scaling everything down to fit.

Add this code to your loop() method:

```
@Override
public void loop() {
    // Read the gamepad
    double y = -gamepad1.left_stick_y;
    double x = gamepad1.left_stick_x;
    double rx = gamepad1.right_stick_x;

    // Calculate motor powers
    double frontLeftPower = y + x + rx;
    double frontRightPower = y - x - rx;
    double backLeftPower = y - x + rx;
    double backRightPower = y + x - rx;

    // Normalize the values so none exceed 1.0
    double maxPower = Math.max(Math.abs(frontLeftPower), Math.abs(frontRightPower));
    maxPower = Math.max(maxPower, Math.abs(backLeftPower));
    maxPower = Math.max(maxPower, Math.abs(backRightPower));

    if (maxPower > 1.0) {
        frontLeftPower /= maxPower;
        frontRightPower /= maxPower;
        backLeftPower /= maxPower;
        backRightPower /= maxPower;
    }

    // Set motor powers
    frontLeft.setPower(frontLeftPower);
    frontRight.setPower(frontRightPower);
    backLeft.setPower(backLeftPower);
    backRight.setPower(backRightPower);
```

```
        // Display motor powers on Driver Station
        telemetry.addData("Front Left", frontLeftPower);
        telemetry.addData("Front Right", frontRightPower);
        telemetry.addData("Back Left", backLeftPower);
        telemetry.addData("Back Right", backRightPower);
        telemetry.update();
    }
```

Understanding the normalization:

• **Math.abs()** gets the absolute value (turns negative numbers positive) so we can find the largest magnitude

• **Math.max()** compares two numbers and returns the larger one

• We find the maximum power across all four motors

• If that maximum is over 1.0, we divide all powers by it

• This scales all values proportionally—they all get smaller by the same percentage

9. Your Complete TeleOp Code

Congratulations! You've learned every piece of the puzzle. Here's your complete, working TeleOp program. This code is ready to copy, paste, and run on your robot!

```
package Teleops;

import com.qualcomm.robotcore.eventloop.opmode.OpMode;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.hardware.DcMotor;

@TeleOp(name = "Basic Drive Train", group = "Basic")
public class BasicDriveTrainTeleop extends OpMode {

    // Declare motor variables
    private DcMotor frontLeft;
    private DcMotor frontRight;
    private DcMotor backLeft;
    private DcMotor backRight;

    @Override
    public void init() {
        // Initialize the motors using hardwareMap
        frontLeft = hardwareMap.get(DcMotor.class, "frontLeft");
        frontRight = hardwareMap.get(DcMotor.class, "frontRight");
        backLeft = hardwareMap.get(DcMotor.class, "backLeft");
        backRight = hardwareMap.get(DcMotor.class, "backRight");

        // Set motor directions (reverse right side)
        frontLeft.setDirection(DcMotor.Direction.FORWARD);
        frontRight.setDirection(DcMotor.Direction.REVERSE);
        backLeft.setDirection(DcMotor.Direction.FORWARD);
        backRight.setDirection(DcMotor.Direction.REVERSE);

        telemetry.addLine("Motors Initialized and Configured");
        telemetry.update();
    }

    @Override
    public void start() {
        telemetry.addLine("TeleOp Started - Ready to Drive!");
        telemetry.update();
    }

    @Override
    public void loop() {
        // Read gamepad inputs
        double y = -gamepad1.left_stick_y;   // Forward/backward (inverted)
        double x = gamepad1.left_stick_x;    // Strafe left/right
        double rx = gamepad1.right_stick_x;  // Rotate left/right

        // Calculate motor powers using mecanum drive formulas
        double frontLeftPower = y + x + rx;
        double frontRightPower = y - x - rx;
        double backLeftPower = y - x + rx;
        double backRightPower = y + x - rx;

        // Normalize powers so no value exceeds 1.0
        double maxPower = Math.max(Math.abs(frontLeftPower), Math.abs(frontRightPower));
        maxPower = Math.max(maxPower, Math.abs(backLeftPower));
        maxPower = Math.max(maxPower, Math.abs(backRightPower));

        if (maxPower > 1.0) {
            frontLeftPower /= maxPower;
            frontRightPower /= maxPower;
            backLeftPower /= maxPower;
            backRightPower /= maxPower;
        }

        // Set motor powers
```

```
            frontLeft.setPower(frontLeftPower);
            frontRight.setPower(frontRightPower);
            backLeft.setPower(backLeftPower);
            backRight.setPower(backRightPower);

            // Display telemetry
            telemetry.addData("Front Left Power", "%.2f", frontLeftPower);
            telemetry.addData("Front Right Power", "%.2f", frontRightPower);
            telemetry.addData("Back Left Power", "%.2f", backLeftPower);
            telemetry.addData("Back Right Power", "%.2f", backRightPower);
            telemetry.addLine();
            telemetry.addData("Y (Forward)", "%.2f", y);
            telemetry.addData("X (Strafe)", "%.2f", x);
            telemetry.addData("RX (Rotate)", "%.2f", rx);
            telemetry.update();
        }
    }
```

Testing Your Robot

You're almost ready to drive! Here's how to test your code and fix common issues.

How to test:

1. **Upload your code** to the Robot Controller
2. **Select your TeleOp** on the Driver Station
3. **Press INIT** and check that you see 'Motors Initialized and Configured'
4. **Press START**
5. **Gently push the left stick forward**—the robot should move forward
6. **Try each direction**: forward, backward, left, right
7. **Try rotating** with the right stick
8. **Try combining movements**—move forward while strafing!

Common issues and solutions:

Problem	Solution
Robot spins when moving forward	Check motor directions—you may need to reverse different motors
Robot moves backward when pushing forward	Flip which motors are reversed (swap FORWARD and REVERSE)
Robot strafes in the wrong direction	Try changing the sign of x in your formulas (+ to - or - to +)
Error: "Unable to find motor"	Motor names in code must exactly match hardware configuration
Robot moves very slowly	Check that normalization isn't dividing unnecessarily—try printing maxPower
One motor doesn't work	Check physical connections and verify motor name in hardware config

Pro testing tip: Always test on blocks first! Lift your robot so the wheels aren't touching the ground. This way you can safely verify each wheel spins the correct direction without the robot driving away.

Congratulations!

You've just built your first mecanum drivetrain TeleOp from scratch! You now understand:

- How mecanum wheels create omnidirectional movement
- How to configure hardware and initialize motors
- How to read gamepad inputs
- The math behind mecanum drive formulas
- How to normalize motor powers
- How to troubleshoot common issues

Next steps to level up your skills:

- Add a **speed control** using the triggers (multiply all powers by a factor)
- Implement **field-centric driving** using the IMU sensor
- Add **encoders** for precise autonomous movement
- Create **macros**—preset movements triggered by buttons
- Learn about **PID control** for smoother, more accurate driving

Remember: every expert programmer started exactly where you are now. Keep experimenting, keep learning, and most importantly—have fun building and competing with your robot!

Good luck this season!