VEX Robotics C++ Programming Guide

Written by Moises Lazo, 2025

Solo Programmer 2024-25 | Teams 96067X & 96067C Newark Tech Robotics Club

Introduction

Hello, this guide comes straight from personal experience, late nights, troubleshooting, testing new things, and figuring a few things out on my own. I was the only coder on our 2024-2025 team, so I had to rely on YouTube videos, the VEX Forum, and trial and error to become proficient. Along the way, I picked up a bunch of tricks and workarounds that I wish someone had shared with me from the start. So, I'm sharing it all here: everything I wish I had known when I started. This guide covers VEXcode V5 Pro (which I primarily used) and Visual Studio Code (which you'll likely use next).

Table of Contents

VexV5 Pro

1. Environment Setup

VEXcode V5 Pro (Pre-installed on your engineering laptops)

- When starting a new competition project:
    - Open VEXcode V5 Pro
    - Click “File” > “Open Examples” > “Competition Template”
    - Name your project and select the correct brain port configuration
    - Enable Expert Autocomplete
    - Enable Expert Robot Configuration (If you decide not to use the GUI Method—see section two below. This will allow you to delete unnecessary items on the robot config page before starting to write everything down in main.cpp.)

2. ROBOT CONFIGURATION

In VEXcode V5 Pro, you have two paths for setting up your robot devices:

1. The **Robot Configuration Page (GUI method)**.
2. Typing device declarations **manually** in `main.cpp`.

GUI Configuration

You can access the **Devices tab (represented by an ID icon)** and add devices with just a few clicks. While convenient, this hides the actual configuration behind auto-generated files, which **judges may not check** when you showcase your complete code.

> **Warning**: GUI-configured devices are stored in hidden files, such as `robot-config.cpp`, which judges don't prioritize when reviewing your `main.cpp` on comp day.

Manual Configuration in `main.cpp`

Take control of your robot from the ground up. When you manually type out every device, you're not just setting things up—you're training yourself for Visual Studio Code and PROS, where there's no GUI safety net, doing all of this in the `main.cpp` page is more efficient because you'll be adding the three-wire devices as you build your robot. If you need to reverse your motors, it'll be easier to do so this way, rather than going back and forth between pages, especially when the bot has more than four motors in the drivetrain. This is because the default method utilizes the GUI, which has a maximum of four drivetrain motors that can be configured.

Example from 96067X_Skills:

STEP 1: Define the Brain First in the `main.cpp` page

```
#include "vex.h"
using namespace vex;

// Competition Control Instance
competition Competition;

/*---------------------------------------------------------------------------
---*/
/*                          ROBOT CONFIGURATION
*/
/*---------------------------------------------------------------------------
---*/

// A global instance of the brain used for printing to the V5 Brain screen
brain  Brain;
```

Always define the `Brain` early — it's the heart of your code and necessary for any three-wire devices (like pneumatics and limit switches). If done incorrectly, the "Memory Permission Error 03802604" will appear in the robot's brain if you attempt to run the program after downloading.

STEP 2: Configure Controller and Three-Wire Devices (after Brain)

Use the `.ThreeWirePort` accessor for each device. These should be placed immediately after the brain declaration.

```
// VEXcode device constructors
controller Controller1 = controller(primary);
digital_out PushH = digital_out(Brain.ThreeWirePort.H);
digital_out PushG = digital_out(Brain.ThreeWirePort.G);
digital_out Hook = digital_out(Brain.ThreeWirePort.F);
limit LimSwitch = limit(Brain.ThreeWirePort.A);

// define variable for remote controller enable/disable
bool RemoteControlCodeEnabled = true;
```

STEP 3: Define Other Devices at the Top of the `main.cpp`

Keep your smart devices organized at the top of the file (under the `#include "vex.h"`). This includes motors, sensors, motor groups, and the drivetrain object.

Example Drivetrain Motor Group Configuration:

```
// Left Drive Motors (6:1 Gear Ratio)
motor leftF = motor(PORT7, ratio6_1, true);      // Front Left (Reversed)
motor leftTB = motor(PORT10, ratio6_1, false);   // Top Back Left
motor leftBB = motor(PORT9, ratio6_1, true);     // Bottom Back Left
(Reversed)
motor_group leftDrive = motor_group(leftF, leftTB, leftBB);

// Right Drive Motors (6:1 Gear Ratio)
motor rightF = motor(PORT3, ratio6_1, false);    // Front Right
motor rightTB = motor(PORT2, ratio6_1, true);    // Top Back Right
(Reversed)
motor rightBB = motor(PORT1, ratio6_1, false);   // Bottom Back Right
motor_group rightDrive = motor_group(rightF, rightTB, rightBB);
```

Example Sensor Setup:

```
/*---------------------------------------------------------------------------
---*/
/*                           SENSOR CONFIGURATION
*/
/*---------------------------------------------------------------------------
---*/

inertial Inertial = inertial(PORT20);             // IMU for Heading
Control
rotation rotationSensor = rotation(PORT11, true);// Lady Brown Position
Tracking (not used)
```

Note: If you decide to use the GUI method, do not include the inertial sensor in the drivetrain configuration; instead, follow the steps above. Doing this will allow you to create functions, such as the PID, using the rotation sensor only.

STEP 4: Group Your Motors and Set Up Your Drivetrain with its dimension specifications

```
/* Drivetrain Specifications:
 - Wheel Diameter: 299mm (for 3.25” wheels)
  - Wheelbase: 297mm (Distance between front and back wheels)
  - Track Width: 310mm (Distance between left and right wheels)
  - External Gear Ratio: 1.33 (Divide the driven gear teeth by the drive
gear teeth)
 */

drivetrain Drivetrain = drivetrain(leftDrive, rightDrive, 299, 310, 297,
mm, 1.33);
```

- Be sure to provide accurate physical specs for smoother turns and better pathing with PID or motion planning.

COMPLETE CONFIG TEMPLATE TO FOLLOW:

```
#include "vex.h"
using namespace vex;

// Competition Control Instance
competition Competition;

/*---------------------------------------------------------------------------
---*/
/*                          MOTOR CONFIGURATION
*/
/*---------------------------------------------------------------------------
---*/

// A global instance of the brain used for printing to the V5 Brain screen
brain  Brain;

// VEXcode device constructors
controller Controller1 = controller(primary);
digital_out PushH = digital_out(Brain.ThreeWirePort.H);
digital_out PushG = digital_out(Brain.ThreeWirePort.G);
digital_out Hook = digital_out(Brain.ThreeWirePort.F);
limit LimSwitch = limit(Brain.ThreeWirePort.A);

// VEXcode generated functions
// define variable for remote controller enable/disable
bool RemoteControlCodeEnabled = true;

// Left Drive Motors (6:1 Gear Ratio)
motor leftF = motor(PORT7, ratio6_1, true);      // Front Left (Reversed)
motor leftTB = motor(PORT10, ratio6_1, false);   // Top Back Left
motor leftBB = motor(PORT9, ratio6_1, true);     // Bottom Back Left
(Reversed)
motor_group leftDrive = motor_group(leftF, leftTB, leftBB);

// Right Drive Motors (6:1 Gear Ratio)
```

```
motor rightF = motor(PORT3, ratio6_1, false);     // Front Right
motor rightTB = motor(PORT2, ratio6_1, true);     // Top Back Right
(Reversed)
motor rightBB = motor(PORT1, ratio6_1, false);    // Bottom Back Right
motor_group rightDrive = motor_group(rightF, rightTB, rightBB);

/* Drivetrain Specifications:

  - Wheel Diameter: 299mm (for 3.25” wheels)
 - Wheelbase: 297mm (Distance between front and back wheels)
 - Track Width: 310mm (Distance between left and right wheels)
 - External Gear Ratio: 1.33 (Divide the driven gear teeth by the drive
gear teeth)

 */
drivetrain Drivetrain = drivetrain(leftDrive, rightDrive, 299, 310, 297,
mm, 1.33);
/*-----------------------------------------------------------------------
---*/
/*                          SENSOR CONFIGURATION
*/
/*-----------------------------------------------------------------------
---*/

inertial Inertial = inertial(PORT20);              // IMU for Heading
Control
rotation rotationSensor = rotation(PORT11, true);// Lady Brown Position
Tracking (not used)

/*-----------------------------------------------------------------------
---*/
/*                          LADYBROWN CONFIGURATION
*/
/*-----------------------------------------------------------------------
---*/
motor Intake = motor(PORT6, ratio6_1, true);      // Ring Intake System
motor lb = motor(PORT8, ratio36_1, false);       // "Lady Brown" Scoring
Arm (36:1 Gear Ratio)
```

3. PRE-AUTON CONFIGURATION

Pre-autonomous setup is the first function that runs before any competition period begins. This is where you prepare the robot's sensors, especially the inertial sensor, and make sure all systems are reset and calibrated before movement begins. Many newer programmers overlook this step and rush into automation, but skipping proper setup will almost always lead to inconsistent behavior later.

One of the most critical steps is **calibrating the inertial sensor**. If the sensor is even slightly off at the start, every PID-based turn or heading-based move in your autonomous program will be inaccurate. The calibration takes two full seconds — during this time, the robot must remain perfectly still.

You can also use this time to reset encoders or initialize mechanisms, such as pneumatic states.

> **Note:** Do not try to move the robot during IMU calibration. Doing so will throw off its internal heading and lead to incorrect turning.

Example:

```
/**

  Initializing sensors and resetting all mechanisms

   - Calibrates IMU (2-second process)

   - Set all pneumatics to default states

  Ensures proper motor configurations

*/

void pre_auton(void) {

 vexcodeInit();                                  // VEX Library
Initialization

 Inertial.calibrate();                           // IMU Calibration

 while (Inertial.isCalibrating()) {              // Block until calibration
completes

   wait(50, msec);

 }
```

```
  // Initialize Pneumatics

 PushG.set(false);

 PushH.set(false);

 Hook.set(true);

}
```

This should be placed in your main competition file and linked in `main()` with `Competition.pre_auton(pre_auton);` before any other code is run. Once this setup is complete, your robot will be appropriately aligned to handle both autonomous and user-controlled operations.

4. AUTONOMOUS CONTROL SETUP

Autonomous is self-explanatory: no human backup. Every movement needs to be dialed, calculated, and consistent. One small miscalibration, and you're veering into a wall instead of scoring points. That's why we use PID, and why we modularize movement functions.

What is PID? (Proportional–Integral–Derivative)

PID control is the foundation of consistent movement, particularly when **turning to a heading** or **driving straight** with corrections.

The formula runs in a loop:

```
double turnSpeed = (error * kP) + (integral * kI) + (derivative * kD);
```

This adjusts the left and right motor speeds every few milliseconds, depending on the distance between the bot and its target.

Why PID matters:

Without PID, your robot will either overshoot, undershoot, or fail to turn accurately. With PID, you get consistent, reliable heading adjustments that give you control over how fast and smoothly the robot turns.

What affects PID?

There is no such thing as a universal PID setup. Even if two robots run the same code, their performance can vary significantly depending on hardware, build quality, and drive physics. Based on personal experience with three different bots, here are the **three main factors** that most impact the PID tuning and why they matter.

1. Weight of the Robot

Heavier bots take longer to start and stop. This affects how strong your `kP` and `kD` values need to be. More mass = more momentum = harder to control.

2. Wheel Type / Setup (Omni vs. Traction)

Omni wheels slide; traction wheels grip. Mixed setups (such as six omni wheels + two traction wheels) can cause uneven turning and make the PID inconsistent. Full omni drivetrains are smoother and easier to tune.

3. Field Friction or Drivetrain Drag

Anything that drags, such as wires, tight gears, or rubbing wheels, throws off PID correction. Even field surface or wall contact can affect your results. Always fix mechanical friction before adjusting the code.

Tuning Results: 3 Bot Comparison 2024-2025 Season

SkillsUSA Bot (Worst Experience): 96067X_Skills – found in 2024-25 ROBOTICS → Skills 2025 file

- **Config:** six motors, eight wheels (six omni, two traction)
- **Problem:** PID values had to be changed often. The two traction wheels created extra grip, leading to asymmetric friction and unstable turns.
- **Solution:** Lowered `kP`, raised `kD`, and added deadzone checks. It wasn't consistent at all in multiple conditions and ultimately failed to have a consistent autonomous segment.

Main Match Bot: 96067X (Blue +, Blu 10 sec, Red +, Red 10 sec) – found in 2024-25 ROBOTICS → Bolt-Jr file

- **Config:** six motors, six omni wheels
- **Result:** Nearly perfect PID behavior. I made only a few tuning calibrations until I achieved near-perfect values, and it performed consistently throughout the season. Turns were smooth, symmetrical, and predictable.

Secondary Match Bot : 96067C (BLUE NEG- , BLUE (+), RED NEG- , R_GRINCH) – found in 2024-25 ROBOTICS → Grinch file

- **Config:** four motors, four omni wheels
- Even with fewer motors, the turning was highly consistent, thanks to its lower mass and all-omni layout—the most efficient bot in terms of turning.

Note!: Mixed wheels (especially traction and omni) significantly affect PID performance due to inconsistent traction across the drivetrain. Stick to consistent omni setups if possible, and test your tuning often.

PID Turning Snippet (Similar Layout Used in All Bots)

```
/*
  PID-Controlled Turning Algorithm
   targetAngle: Desired heading in degrees
   Features:
   - Tunable PID constants (kP=0.4, kI=0.0008, kD=0.24)
   - Anti-windup protection through speed clamping
   - 1-degree tolerance exit condition
   - Continuous angle correction
*/
void turnToAngle(double targetAngle) {
 // PID Constants
 const double kP = 0.4;    // Proportional - Main correction factor
 const double kI = 0.0008; // Integral - Eliminates steady-state error
 const double kD = 0.24;   // Derivative - Reduces overshoot

 // Control Variables
 double error = 0;
 double previousError = 0;
 double integral = 0;
```

```
double derivative = 0;

do {
  // Calculate PID Terms
  error = targetAngle - Inertial.rotation();
  integral += error;
  derivative = error - previousError;
  previousError = error;

  // Calculate Output
  double turnSpeed = (error * kP) + (integral * kI) + (derivative * kD);

  // Output Clamping
     if (turnSpeed > 100)
    turnSpeed = 100;
  if (turnSpeed < -100)
    turnSpeed = -100;

  // Differential Drive Execution
  leftDrive.spin(fwd, turnSpeed, pct);
  rightDrive.spin(reverse, turnSpeed, pct);

} while (fabs(error) > 1.0);  // Exit when within 1 degree of error

leftDrive.stop();
rightDrive.stop();
}
```

PID Tuning Tips

Situation	How to fix
Overshooting	Lower `kP`, increase `kD`
Stops short	Raise `kP`, try small `kI`
Constant oscillation	Reduce all values, especially `kI`
Sluggish turning	Raise `kP` lightly

For More Info, Please View:
VEX Robotics PID Control - By Sequam Robotics on YouTube
Design PID formula to Fit Your Robot - By Mathew chou on YouTube

Additional Autonomous Functions

1. Modular Movement Commands (All Bots)

These were used to abstract away `Drivetrain.driveFor()` and give more readable auton code:

```
/*
  Controlled Forward Movement
  Percentage velocity (0-100%)
  Travel distance in millimeters
*/
void driveFWD(double speed, double distance) {
 Drivetrain.setDriveVelocity(speed, pct);
 Drivetrain.setStopping(brake);                // Brake mode for precision
 Drivetrain.driveFor(fwd, distance, mm);
}

/**
 Controlled Reverse Movement
 Speed Percentage velocity (0-100%)
 Travel distance in millimeters
*/
void driveREV(double speed, double distance) {
 Drivetrain.setDriveVelocity(speed, pct);
 Drivetrain.setStopping(brake);    // Brake mode for precision
 Drivetrain.driveFor(reverse, distance, mm);
}
```

These were reused across all bots for ring pickups, mobile goal approach, and repositioning

2. Pneumatic Hook Control (SkillsUSA + Match Bot)

To grab mobile goals or hooks:

```
 PneumaticsA.set(true);
 PneumaticsB.set(true);
 pneumaticsClosed = true; task::sleep(200);
```

In some bots, `PneumaticsC` was added for a secondary arm mechanism.

3. Intake Control (All Bots)

Simplified to `Intake.spin(fwd)` or `reverse`, with added delays for partial cycles:

```
 Intake.setVelocity(100, pct);
Intake.spin(fwd);
wait(0.6, seconds);
Intake.stop(coast);
```

Useful when doing ring pickups mid-run, especially after autonomous turns.

Ex.

```
// Turn the robot to face the next ring

 turnToAngle(90);

 // Activate the intake to capture the pre-loaded ring

 Intake.spin(fwd);

 wait(0.6, seconds); // Spin intake for 0.6 seconds to capture the first
ring

 // Drive forward without stopping to collect the next ring

 driveForward(60, 760);

 // Stop the intake 1.5 seconds after reaching the second ring to ensure

 // collection

 wait(1.5, seconds);

 Intake.stop(coast);
```

Tips to Build an Autonomous Routine

- **Test each function separately.** Don't assume `turnToAngle()` works perfectly just because it worked once. Confirm every movement in isolation before combining them.
- **Start slow, then speed up.** Use lower drive speeds (30-50%) while testing to reduce the risk of overshooting or damaging the robot.
- **Record your PID values.** Keep a notebook (or a digital log) of each test's results and PID values. This will save you from having to start over when conditions change.
- **Retest.** Always run at least two full autonomous tests on the testing field before the match.
- **Use timeouts.** If a turn or drive stalls for too long, add a timeout to prevent the program from getting stuck in an infinite loop.
- **Align with field elements.** Bump the robot against walls, tape lines, and even mat edges when possible to reset the heading. It's way more consistent than relying on sensors alone.
- **Don't chase perfection.** Even with perfect code, game elements or field variances can throw off your heading by a few degrees. Plan your strategy around consistent scoring, not just "perfect" turns.

5. DRIVER CONTROL CONFIGURATION

Driver control is where your code meets your drivers' hands. It's the phase that feels the most interactive, where every input, button press, and joystick tilt translates to real motion on the field. This section covers the fundamentals of setting up driver control, including recommended drive styles, intake operation, and the nuanced art of toggling pneumatics.

You must refer to this controller map for the user interface and programming.

Choosing a Drive Type

The way your robot drives can make or break a match, especially under pressure. Each driver has their own style, so give them options:

- **Split Arcade Drive**: The most common. One joystick controls forward and reverse (Y-axis) and turning left and right (X-axis).

Example:

```
int leftStickX = Controller1.Axis1.position();

int leftStickY = Controller1.Axis3.position();

Drivetrain.arcade(leftStickY, leftStickX);
```

  - ➢ **Pros:** Simple, easy to learn, works for most drivers.
  - ➢ **Cons:** Some drivers may struggle to make smooth turns.

- **Tank Drive**: Left joystick controls left wheels, right joystick controls right wheels.

Example:

```
int leftStickY = Controller1.Axis3.position();

int rightStickY = Controller1.Axis2.position();

LeftDriveSmart.spin(fwd, leftStickY, pct);

RightDriveSmart.spin(fwd, rightStickY, pct);
```

  - ➢ **Pros:** Direct control over each side.
  - ➢ **Cons:** It can be harder to master diagonal movement.

- **Single Arcade Drive**: Left or Right (depending on which joystick axis you configure it for. Control the whole drivetrain with a single joystick..

Example:

```
int rightStickY = Controller1.Axis2.position();

int rightStickX = Controller1.Axis1.position();

Drivetrain.Arcade (rightStickY, rightStickX);
```

➢ **Pros:** Direct control with a single joystick.

➢ **Cons:** It can be harder to control a single direct movement, such as driving forward without accidentally throttling for a turn.

Always pick a style that matches your driver's comfort zone—consistency matters more than complexity.

Pneumatics Control: Two Methods

Pneumatics are a key part of competitive VEX builds—whether it's a hook, lift, or clamp. There are two popular coding approaches for toggling pneumatics, each with its advantages and disadvantages.

1. Toggle with Boolean State (if statement)

This method utilizes a dedicated variable (e.g., `pneumaticsClosed`) to track the mechanism's state. When the driver presses the button, the code checks the state and sets the pneumatics accordingly.

Example:

```
bool pneumaticsClosed = false;

   if (Controller1.ButtonA.pressing()) {

     if (pneumaticsClosed) {

       PneumaticsA.set(true); // Extend pneumatics
```

```
      PneumaticsB.set(true);
      pneumaticsClosed = false; // Update state to open
    } else {
      PneumaticsA.set(false); // Retract pneumatics
      PneumaticsB.set(false);
      pneumaticsClosed = true; // Update state to closed
    }
    this_thread::sleep_for(500); // Debounce delay ensures debounce and
    }                            // prevents rapid state toggling
```

- **What it does:** Checks the current state and flips it.
- **Pros:** Easy to understand, prevents rapid toggling if the driver holds the button too long.
- **Cons:** More lines of code, but highly readable.

2. Toggle with "!" Operator

This method flips the boolean variable directly with the "!" operator.

Example:

```
bool liftDeployed = false;
  // PNEUMATIC LIFT TOGGLE - X Button
  if (Controller1.ButtonX.pressing()) {
    liftDeployed = !liftDeployed;
    PushG.set(liftDeployed);
    PushH.set(liftDeployed);
    this_thread::sleep_for(200);     // Debounce delay     }
```

- **What it does:** Directly toggles the state on each press.
- **Pros:** Cleaner code, fewer lines.
- **Cons:** Can be less intuitive for new programmers, especially when multiple mechanisms share a single button.

Debounce Delays: Why They Matter

A debounce delay is a brief pause (typically 200-500 milliseconds) that occurs after a button press is registered. Without it, the controller's natural electrical bounce might cause multiple toggles in rapid succession, like a machine gun on your pneumatics.

Example:

```
this_thread::sleep_for(500); // Debounce delay ensures only one toggle per
press
```

Always include this after toggling pneumatics when using the 'pressing' callback, since the 'pressed' callback isn't functional due to the outdated V5 platform. Otherwise, your robot might activate, deactivate, and reactivate in milliseconds, confusing the driver and wasting air.

Lift Control – Grinch (96067C)

The Grinch robot had a lift system that required preset positions, one up for height scoring, and one down for resetting the ground scoring. This made it easy for drivers to focus on gameplay instead of manually positioning the lift every time, wasting time.

Lift Control Variables:

```
// Target positions for lift

int liftUpPosition = 1600;  // Adjust this to your desired height in
deci-degrees

int liftDownPosition = 0;  // Original 0 degree position



bool L1Pressed = false;

bool L2Pressed = false;
```

```
void controlLift() {
   // Lift Up to set height
   if (Controller1.ButtonL1.pressing() && !L1Pressed) {
       L1Pressed = true;
       L2Pressed = false;
       Lift.spinToPosition(liftUpPosition, degrees, 80,
velocityUnits::pct, true);
   }

   // Reset Down to 0 position
   if (Controller1.ButtonL2.pressing() && !L2Pressed) {
       L2Pressed = true;
       L1 Pressed = false;
       Lift.spinToPosition(liftDownPosition, degrees, 80,
velocityUnits::pct, true);
   }
}
void usercontrol(void) {
while (true) {
  controlLift();
       Lift.stop(hold);
     }
     }
```

BoltVader (96067X) — LadyBrown Function

BoltVader had a scoring arm called **Lady Brown, a dunking mechanism that could score up to two rings at a time in the hooks,** which required precise and rapid movement during both autonomous and user control. The function made it so that the lady brown would move rapidly to its scoring position, wait for a brief second, then revert back until its limit switch was pressed and it would stop.

```
void Score() {
 lb.spinToPosition(220, degrees);  // Spin to score position
 wait(250, msec);                  // Stabilization pause
 lb.spin(reverse);                 //  Descent
 waitUntil(LimSwitch.pressing());  // Stop when limSwitch is pressed (safe
position where rings can be placed for accurate scoring)
 lb.stop(hold);                    // Maintain position
}
```

Using LadyBrown in Autonomous:

```
void autonomous(void) {
 // System Initialization
 lb.setVelocity(100, pct);
Score();
}
```

Using LadyBrown in Driver Control:

```
if (Controller1.ButtonL1.pressing()) {
     Score();                          // Reuse ladybrown score routine
     this_thread::sleep_for(200);      // Debounce delay
   }
```

Driver Speed Control (experimental, not tested as of 6/5/25)

By default, joystick input (`Controller1.Axis3.position()`, for example) outputs a value between -100 and +100, representing the percentage of motor speed output. If the driver slams the joystick forward, they're asking for 100% speed — even if the drivetrain can't handle that safely.

Speed Scaling

To cap that speed, we apply a **scaling factor**. Instead of setting the motor speed directly from the joystick value, we multiply the joystick input by a percentage cap. For example, if we want to limit to 70% max speed:

```
int leftStickY = Controller1.Axis3.position();

int rightStickX = Controller1.Axis1.position();

// Apply scaling factor

double driveScale = 0.7; // 70% maximum speed

Drivetrain.arcade(leftStickY * driveScale, rightStickX *
driveScale);
```

This lets the driver move the stick all the way forward and still cap the speed at 70%. They get the full range of control, but at a safer top speed. You can create and assign individual drivescale values for the left and right joysticks if you want different driving and turning speeds.

Dynamic Speed Scaling

If you want to get even fancier, you can add a **"speed mode" toggle** so the driver can switch between slow and fast modes mid-match:

```
bool speedModeFast = false;

if (Controller1.ButtonUp.pressing()) {

   speedModeFast = true;

   this_thread::sleep_for(200);

}
```

```
if (Controller1.ButtonDown.pressing()) {

   speedModeFast = false;

   this_thread::sleep_for(200);

}

double driveScale = speedModeFast ? 1.0 : 0.6; // Fast or slow mode

int leftStickY = Controller1.Axis3.position();

int leftStickX = Controller1.Axis1.position();

Drivetrain.arcade(leftStickY * driveScale, leftStickX * driveScale);
```

Notes

- **Never hard-code 100%** unless the driver specifically requests it. Even the best drivers can't always handle max speed under pressure.
- **Test different scaling values** (such as 0.5, 0.6, 0.75) and get feedback from your driver during practice. They might prefer different scales for various stages of the match.
- **Avoid mixing this scaling with `Drivetrain.setDriveVelocity()`**, as the function is intended for use in autonomous routines. In driver control, the joystick inputs are dynamic.

Visual Studio Code:

This chapter is for those who refuse to settle for the basics, the ones hungry to push their code beyond the confines of VEXcode V5 Pro. If you want to join the ranks of top-tier teams at VEX Worlds, embrace **Visual Studio Code (VS Code)** with **PROS**, **LemLib**, and **Jerry Path**. This environment enables advanced programming, path following, and dynamic control, which can significantly impact your performance on the field. Here's how to get started.

Environment Setup

Before diving into advanced programming, build a solid foundation by setting up Visual Studio Code with the necessary tools and essential extensions.

Step 1: Install Visual Studio Code
Head over to Visual Studio Code's website and download the latest stable release for Windows or the OS you'll be using it on. Follow the installer prompts and accept all agreements.

Step 2: Install the PROS Extension
Open VS Code and click the **Extensions tab** (left sidebar). Search "PROS" and install the first result. This extension sets up the environment for VEX V5 development.

Important: If you see an "Client's chain is not working" error, click **"Install Now"**. If it doesn't appear, restart VS Code.

Step 3: Create a New Project
In the PROS sidebar, click **"Create Project"** and select a folder (e.g., `PROS_Tutorial`). Name your project (e.g., `test_project`). Select **V5** as the platform, using the latest version (V4 recommended for added features). A new folder with `main.cpp` will be created.

> **Pro Tip:** V4 includes features like three-wire expander support, so stick with it unless you have a reason to use V3.

Step 4: Open the Integrated Terminal
In VS Code, go to **View > Terminal** (or press `Ctrl+`). This lets you run commands for managing libraries.

Step 5: Install ClangD
Return to the Extensions tab, search "ClangD," and install it. This helps compile LemLib and other advanced features.

LemLib Installation and Setup

LemLib supercharges your autonomous code with odometry and PID. Here's how to get it working:

Step 1: Install LemLib
In the terminal, run:

```
pros install lemLib
```

This fetches the latest version.

Step 2: Find the Example Project
Search **LemLib GitHub** and scroll to their example project. Copy the example code (which includes odometry setup, PID tuning, etc.) into your `main.cpp`.

Key Components:

- **Motor Setup**
  Negative means reversed. Set your gear ratio carefully—mixing them up can cause problems with PID.

- **Inertial Sensor**
  Always include an inertial sensor—it's a reliable way to track heading and odometry.

- **Tracking Wheels and Encoders**
  Place your horizontal and vertical encoders in a strategic location. They track your robot's position, and their offsets determine accuracy.

    - **Offsets:**

        - Backwards? Use negative values.
        - Forwards? Positive values.
        - Left? Negative.
        - Right? Positive.

Step 3: Drivetrain and Chassis Configuration
Define your drivetrain's **track width**, **wheel size**, and **horizontal drift factor** (omni wheels have lower drift, while traction wheels have higher drift).

Step 4: PID Tuning
Use LemLib's **PID tuning flowchart** for accurate adjustments. Refer to their documentation—don't just take my word for the PID, as it's based on my personal experiences.

Jerry Path and Pure Pursuit

If you're feeling adventurous, integrate **Jerry Path** for dynamic autonomous path planning.

Step 1: Create a Path
Visit Path Jerry and plot your desired path using waypoints. Right-click to add points and adjust orientations, speeds, and other parameters.

Step 2: Export the Path
Download the path file as `.jerry.io.txt`.

Step 3: Place the File
Move it into your project's `src/static` folder.

Step 4: Use the Path
In your code:

```
chassis.follow("example_file.jerry.io.txt", 10, 1000);
```

This instructs the robot to follow the path with a lookahead of 10 inches and a timeout of 1 second (1000 milliseconds).

> **Pro Tip:** Pure Pursuit breaks the path into dots, allowing your robot to follow each waypoint fluidly.

Key Functions

Here's a breakdown of core functions:

Function	Description
`moveToPoint`	Moves to an (X, Y) coordinate. Use `maxSpeed` and `forwards` options for fine control.
`moveToPose`	Same as `moveToPoint`, but lets you specify a final heading.
`turnToHeading`	Rotates in place to a given heading.
`swingToHeading`	Rotates around one locked side—great for quick turns.
`purePursuit`	Follows a dynamic path with smooth movement.

Driver Control

Driver control uses loops that continuously update joystick values and button presses.

Example Joystick Setup:

```
int leftStickY = controller.get_analog(ANALOG_LEFT_Y);

int leftStickX = controller.get_analog(ANALOG_LEFT_X);

chassis.arcade(leftStickY, leftStickX);
```

You can switch to **tank drive** or invert controls for driver preference.

Pneumatics

Pros support pneumatics via 3-wire ports using `ADI`. Here's how to toggle them:

Simple Toggle Example:

```
bool pistonToggle = false;

if (controller.get_digital(DIGITAL_R1)) {

  if (!pistonToggle) {

    piston.set_value(true);

    pistonToggle = true;

  } else {

    piston.set_value(false);

    pistonToggle = false;

  }

  pros::delay(500); // Debounce delay

}
```

> **Why Debounce?**
> Without a delay, the loop reads the button press dozens of times, toggling the piston repeatedly and unpredictably.

Uploading Code

Once your code is written:

1. Save your project.
2. Click **Build** and **Upload**.
3. Watch it appear on your V5 brain screen.

> **Pro Tip:** You can customize the project name, description, and icon in the `project.pros`

My Perspective and Resources

Although I never fully transitioned to Visual Studio, I encourage future programmers to embrace it. VEXcode V5 Pro is no longer updated (since 2022), and moving to VS with PROS and LemLib is the next step for competitive robotics.

For more information, refer to the official LemLib documentation, the Jerry Path website, and "Setting Up PROS with LemLib" by Steam Labs, on which this Visual Studio chapter is based on.

Final Thoughts

This guide was made from the ground up during my 2024-2025 season with teams 96067X and 96067C. I hope it saves you time, gives you confidence, and helps you win. Thank you and good luck on your tournaments.

— Moises Lazo