Aurora Tracker with STM32 Motor Control

Embedded Systems Project Report

Tuan Kiet Le
T00404130
ECE5140 - Dr. Tarek Elfouly

January 17, 2026

Contents

1 Introduction

1.1 What This Project Does

This project combines position tracking with motor control. We use an NDI Aurora electromagnetic tracker to know where things are in 3D space, then use that info to drive motors on an STM32 microcontroller. There's also an OpenMV H7 camera hooked up for visual feedback.

The basic idea: two sensors tell us positions, we calculate how much the motors need to move, send commands over serial, and the STM32 handles the actual motor driving with PID control.

1.2 Why We Built This

Position tracking + precise motor control is useful for lots of stuff like robotic arms, automated alignment systems, and research setups that need repeatable positioning. This project gave us hands-on experience with real-time sensor processing, multi-threaded programming, serial communication, and closed-loop control.

1.3 Goals

- Track two sensors in real-time and filter out bad readings
- Calculate servo and linear motor angles from position data
- Build a reliable STM32 motor controller with PID
- Get solid UART communication working between PC and microcontroller
- Log everything to CSV for debugging and analysis
- Hook up an OpenMV H7 camera for visual monitoring

2 System Architecture

2.1 Overall Setup

Figure 1 shows how everything connects together.

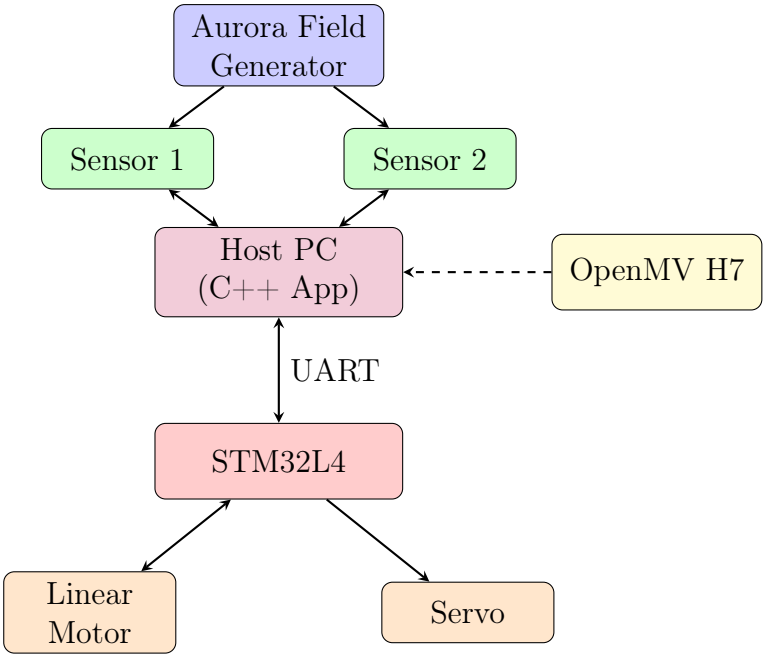

Figure 1: System Block Diagram

2.2 System Algorithm

Figure 2 shows the overall flow of the system.

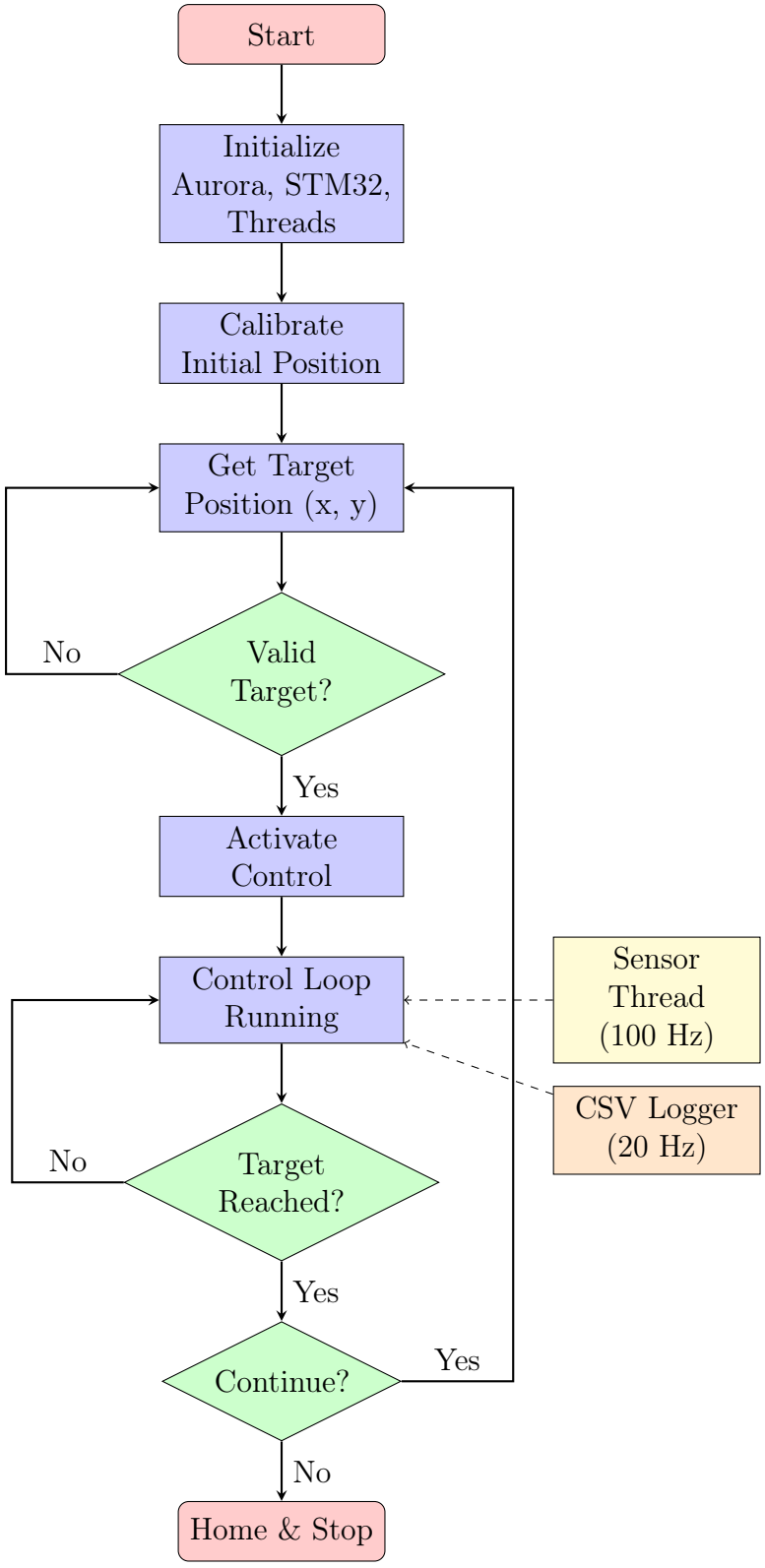

Figure 2: System Flowchart

2.3 Hardware

As shown in Figure 3, the robot arm has two degrees of freedom, one linear actuator and one servo motor.

Part	Model	Job
Tracker (Figure 4)	NDI Aurora	Tells us where the sensors are
MCU	STM32L4	Runs PID, drives motors
Linear Motor	GB37Y3530-12V-251R	Moves robot arm back and forth
Servo	MG996R	Rotates the end of point
Camera	OpenMV H7	Visual feedback
Driver (Figure 5)	L298N	Makes motor go both ways

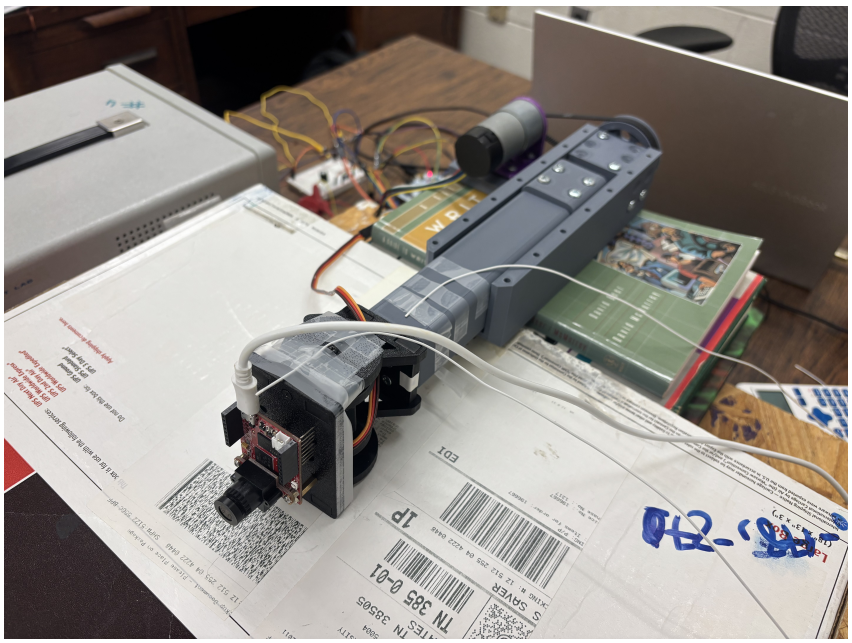

Figure 3: 2 DOF robot arm with camera attached

2.4 Software Structure

The PC runs three threads at the same time:

1. **Sensor Thread** – Grabs data from Aurora at 100 Hz, checks if it's valid, stores it in shared memory
2. **CSV Thread** – Writes sensor data to file at 20 Hz for later analysis
3. **Control Thread** – Every 5 seconds, calculates motor angles and sends commands to STM32

The STM32 side uses interrupts to receive UART data, runs PID at 100 Hz, and outputs PWM to both motors.

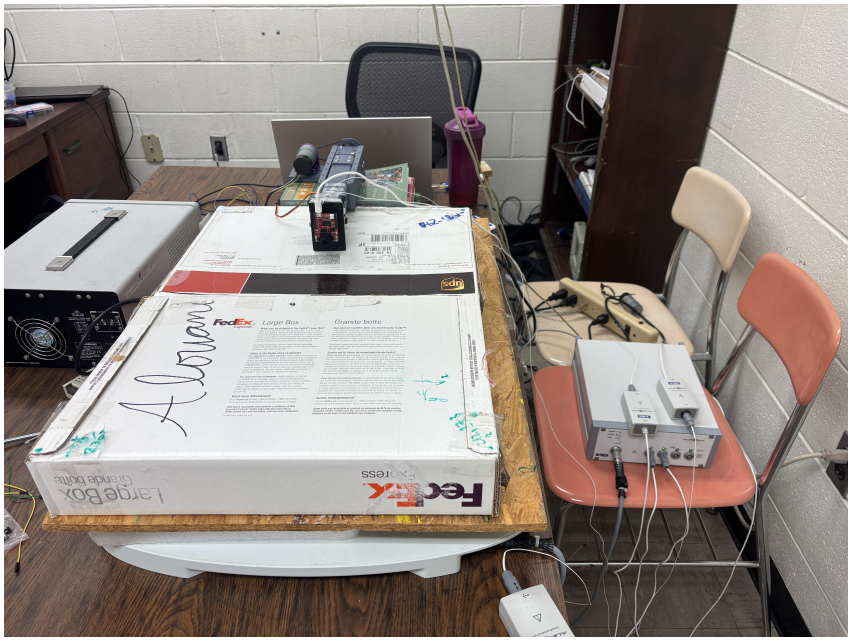

Figure 4: Aurora Sensor (top right) is sitting on the orange chair

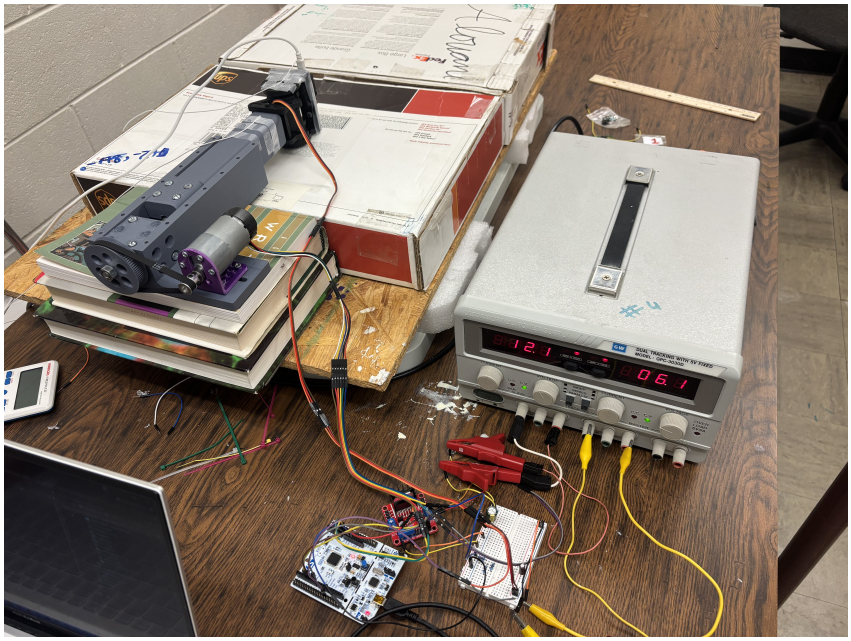

Figure 5: H - Bridge Driver, DC supply

3 Hardware Design

3.1 Aurora Tracking

The Aurora system uses electromagnetic fields to track sensor position and orientation. Each sensor gives us X, Y, Z position (in mm) plus a quaternion for rotation. Accuracy is about 1 mm which is pretty good for our application.

We use two sensors: one on the thing we're tracking, one as a reference on the arm.

3.2 STM32 Setup

Here's how we configured the peripherals:

- **TIM1** – PWM for DC motor speed (prescaler=1, ARR=999)
- **TIM2** – Encoder input for position feedback (32-bit counter)
- **TIM3** – PWM for servo at 50 Hz (prescaler=79, ARR=999)
- **USART2** – Serial at 9600 baud to talk to PC
- **PB4/PB5** – GPIO for motor direction (H-bridge IN1/IN2)

3.3 Motors

The linear motor uses a lead screw that converts rotation to linear motion – 0.25 mm per revolution. The servo swings from 45° to 135°.

3.4 OpenMV Camera

The OpenMV H7 has a Cortex-M7 running MicroPython. Right now it's mainly for visual monitoring, but could be used for computer vision feedback later.

4 Software Implementation

4.1 Inverse Kinematics

To figure out where to move the motors, we use some basic trig. The servo angle comes from:

$$\theta_{servo,n} = \arccos\left(\frac{\Delta x_n + L \cdot \cos(\theta_{current})}{L}\right) \tag{1}$$

where Δx is how far we need to go in X, L is the arm length (39 mm), and $\theta_{current}$ is where the servo is now. It's iterative so it converges smoothly.

For the linear motor, we just convert Y distance to rotations:

$$\theta_{linear} = -\frac{y_{ref,s2} - (L_{s1p} + L_{s2p}\sin\theta_{servo,n})}{0.25} \times 360 \tag{2}$$

The negative sign is because positive Δy means move forward, which needs the motor to spin backwards (lead screw thing).

Listing 1: Angle Calculation in Control Thread

```
// Calculate servo angle using iterative formula
double cos_arg = (delta_x + SENSOR2PIVOT *
    cos(g_controlData.current_servo_angle * PI / 180.0)) / SENSOR2PIVOT
    ;
if (cos_arg > 1.0) cos_arg = 1.0;
if (cos_arg < -1.0) cos_arg = -1.0;

angle_servo = acos(cos_arg) * 180.0 / PI;

// Clamp to physical limits
if (angle_servo < SERVO_ANGLE_MIN) angle_servo = SERVO_ANGLE_MIN;
if (angle_servo > SERVO_ANGLE_MAX) angle_servo = SERVO_ANGLE_MAX;

// Linear motor from Y delta
double rev_to_rotate = delta_y / REV_MM;
angle_linear = -rev_to_rotate * 360.0;
```

4.2 UART Protocol

We keep the communication simple. PC sends:

`L:<linear_angle>,S:<servo_angle>\n`

Example: `L:-45.23,S:90.00\n`

STM32 replies with `1\n` when it's done moving. The PC waits for this before sending the next command.

Listing 2: Parsing Commands on STM32

```
int Parse_Dual_Angle_Command(char* buffer,
                             float* linear_angle,
                             float* servo_angle) {
    int result = sscanf(buffer, "L:%f,S:%f", linear_angle, servo_angle)
        ;
    return (result == 2) ? 1 : 0;
}
```

5 Control System

5.1 PID Controller for motor angle

The PID runs on the STM32 to control the linear motor position. Here are the tuned values:

Parameter	Value	What it does
K_p	1.0	Main correction force
K_i	0.5	Eliminates steady-state error
K_d	0.1	Dampens oscillations
Sample Time	10 ms	How often PID runs
Dead Zone	$\pm$10 counts	"Close enough" range
Min PWM Duty	15%	Overcomes motor friction

The implementation includes anti-windup (caps the integral term) and a minimum PWM threshold so the motor actually moves when it should.

Listing 3: PID Core Logic

```
float PID_Compute(PID_Controller *pid, float setpoint, float measured)
    {
    float error = setpoint - measured;

    // Dead zone - if we're close enough, stop
    if (fabs(error) < pid->dead_zone) {
        pid->integral = 0;
        return 0;
    }

    // P term
    float p_term = pid->Kp * error;

    // I term with anti-windup
    pid->integral += error * pid->dt;
    if (pid->integral > pid->integral_max)
        pid->integral = pid->integral_max;
    else if (pid->integral < -pid->integral_max)
        pid->integral = -pid->integral_max;
    float i_term = pid->Ki * pid->integral;

    // D term
    float derivative = (error - pid->prev_error) / pid->dt;
    float d_term = pid->Kd * derivative;
    pid->prev_error = error;

    float output = p_term + i_term + d_term;

    // Clamp output
    output = fmax(fmin(output, 100.0), -100.0);

    // Minimum PWM to overcome friction
    if (output != 0 && fabs(output) < pid->min_pwm_threshold)
        output = (output > 0) ? pid->min_pwm_threshold : -pid->
            min_pwm_threshold;

    return output;
}
```

5.2 Timing

Different parts of the system run at different speeds:

- Sensor polling: 100 Hz
- CSV logging: 20 Hz
- Control commands: every 5 seconds
- STM32 PID loop: 100 Hz

6 Conclusion

We built a working system that tracks electromagnetic sensors and uses that data to control motors in real-time. The main pieces all work together: Aurora gives us positions, the PC calculates angles and coordinates everything, and the STM32 handles the actual motor control with PID feedback.

What works well:

- Reliable sensor tracking with good filtering
- Smooth motor control with minimal overshoot
- Simple but robust communication protocol
- Useful CSV logging for debugging

Things that could be better:

- Control loop could run faster than every 5 seconds
- OpenMV camera isn't fully integrated yet

Overall, the project demonstrates how to integrate multiple embedded subsystems into something that actually works. The multi-threaded architecture keeps things responsive, and the PID controller gives us the precision we need. The link for demo has been attached below:

- Robot Arm Movement: https://drive.google.com/file/d/1VKm69nz09LB0u3aUUcMf79Sfyy_caBB2/view?usp=sharing
- User Interface: https://drive.google.com/file/d/18wPMM6UHqs1dqIHGXfHYBYghuFead4py/view?usp=sharing

References

1. NDI Aurora User Guide

   https://github.com/Oct19/NDI-API-Sample-v1.4.0

2. STM32L4 Reference Manual (RM0394)
3. OpenMV H7 Documentation
4. PID Control for motor https://www.steppeschool.com/blog/pid-implementation-in-c