APS380 Final Project Report

James Yun, Madeline Zhao, Kamiar Lashgari

University of Toronto

APS380: Intro to Electric Vehicle Design

Prof. Trescases, Prof. Mackay

Dec 3, 2025

1. Introduction & Background

Modern electric and luxury vehicles are increasingly equipped with advanced driver assistance systems (ADAS) to enhance safety and comfort. However, the development and testing of these systems are costly due to the need for real-world testing. Our team addressed this challenge by implementing Adaptive Cruise Control (ACC) and Blind Spot Detection (BSD) within a hardware-in-the-loop (HIL) simulation platform. The HIL platform allowed for controlled and repeatable testing of the two systems by interfacing real hardware components (controllers, sensors, actuators) with vehicle and environment models. To increase test realism while keeping HIL repeatability, select HIL components were migrated onto a 1:15 scaled physical model environment, including a model vehicle equipped with the ACC and BSD sensors, as well as movable obstacles.

The system successfully implements ACC speed-regulation logic and BSD proximity-detection logic. This report outlines the system architecture, hardware and software implementation, testing methodology and results, and the changes in project scope.

2. Scope

Throughout the development of the Adaptive Cruise Control and Blind Spot Detection HIL Platform, several significant scope adjustments were made in response to team capacity, technical challenges, and evolving project priorities. These changes were essential to ensure successful project completion within the available time and resources.

Closer to the end of the project timeline, one team member withdrew from the course, reducing the team's available engineering capacity. This change significantly affected the feasibility of the original plan. The team initially intended to design a functional 1:15 scale vehicle capable of dynamic interactions with the environment. Although a CAD design (Appendix B) for the model EV was created, the team encountered hardware issues with the power delivery for the DC motors. The Arduino-based system was unable to deliver sufficient power to the DC motors for wheel actuation, so the decision was made to revise the project's physical design later in the project timeline. The team transitioned to a stationary platform that contained all the sensors,s and the other road vehicle models were moved relative to the base for system testing. This redesign eliminated the power-delivery challenges we faced while preserving the ability to conduct controlled and repeatable ADAS tests.

Originally, the plan was to create a custom motor driver PCB to provide the power delivery and switching control. However, due to time limitations and manufacturing constraints, fabricating the PCB became infeasible within the project schedule. As a result, the motor-driver circuitry had to be implemented on a perfboard.

Alongside the hardware redesign, the project's ADAS feature set was also adjusted. The original plan included implementing Lane-Keeping Assist (LKA), which required camera-based lane detection, consistent lighting, precise lane markings, and closed-loop steering control. Since the transition to a stationary platform made dynamic lane tracking impractical, LKA was replaced with BSD. BSD is well suited to a fixed sensor configuration and relies on side-mounted ultrasonic measurements, offering straightforward testing conditions. This change maintained the project's objective of implementing two

ADAS subsystems while aligning the scope with the available resources and the updated hardware capabilities.

3. Functional Objectives

The project was guided by a set of functional objectives intended to ensure that the final system achieved meaningful ADAS functionality while remaining feasible within resource and time limitations.

System Integration

- The module and HIL platform must execute with consistent latency - below 100 ms.

ACC Requirements

- Following distance is maintained within 10% of the target.
- The system must function correctly across speeds 0-140 km/h

BSD Requirements

- The BSD system must detect obstacles within 100 ms.

Primary Functions

- Measure distances to objects ahead
- Calculate required speed adjustments based on the current following distance and user-set parameters
- Generate motor control commands for speed increase, decrease, or maintenance
- Process user inputs for cruise speed and following distance settings
- Display system status, including current mode, detected distances, and active commands

Secondary Functions

- Filter sensor noise through digital signal processing algorithms
- Validate sensor readings for range and consistency checks
- Log operational data for performance analysis and debugging
- Communicate with external systems via standardized protocols
- Provide emergency override capability for immediate system shutdown
- Execute self-diagnostics on startup and during operation

Table 1. Objectives

Objective	**Metric**	**Goal**
Responsive	End-to-end latency from sensor detection to motor command	< 100ms
Accurate	Following distance error from the setpoint	Within 10% of the target distance
Safe	Emergency stop response time when an obstacle is detected < 1m	< 500ms
Portable	Combined weight of all system components	< 2kg

4. System Design

System Architecture

The implemented ACC system follows a simple control loop architecture shown in Figure 1. The Arduino continuously reads the distance data from the front ultrasonic sensor, executes the motor control algorithm. Three pushbuttons were also added to allow real time adjustment of system parameters and emergency override.

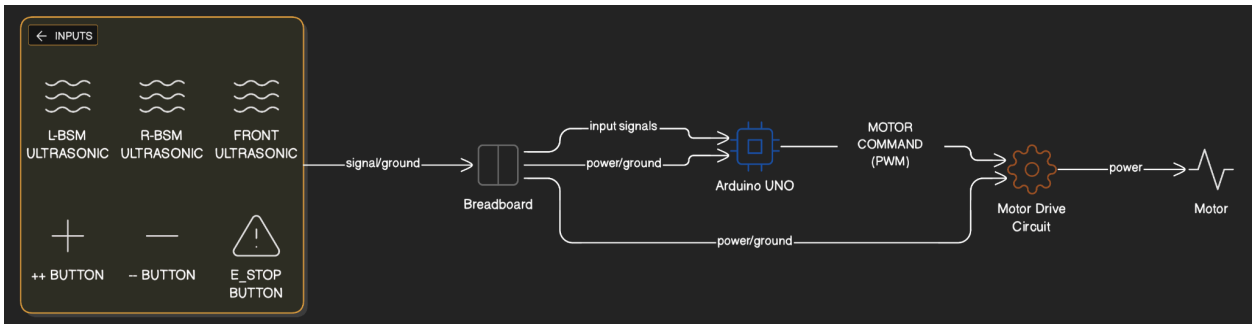

Figure 1. System Block Diagram

The system operates in three different states of a finite state machine, as depicted in Figure 2. The RUNNING state executes normal ACC functionality.

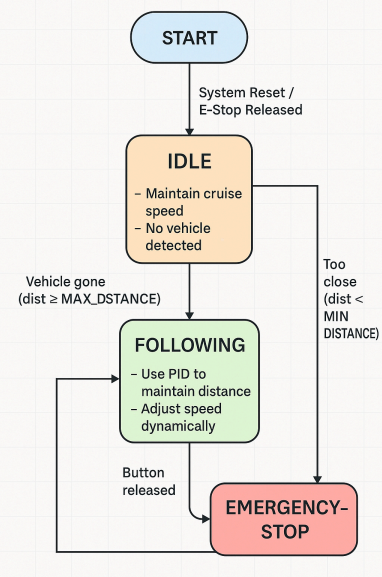

Figure 2. State Machine Diagram

Hardware Design and Component Justifications

The Arduino UNO R3 replaced the initially proposed RPi due to the benefit of the ability to run the control algorithm without operating system overhead. PWM generation and interrupt handling were directly accessible, and the Arduino's simplified programming framework and native USB serial communication enabled ease of debugging and power to the setup.

The Adafruit VL53L0X ToF sensor was also replaced with a SR04 ultrasonic sensor for its higher reliability and consistency in the distance measurement readings.

The physical user interface consists of three distinct pushbuttons and three LEDs. The emergency stop button connects to a digital pin with interrupt capabilities to ensure immediate response regardless of FSM state. Target distance adjustments button used a simple debounce logic to detect user presses. All buttons use the internal pull-up resistor, eliminating the need for external resistors in the system.

Table 2. Pin Assignments

Pin	Function	Hardware Component
2	Emergency Stop	Pushbutton
3	Target Distance Increment	Pushbutton
4	Target Distance Decrement	Pushbutton
5	BSM Trig (Left)	Ultrasonic Sensor
6	BSM Echo (Left)	Ultrasonic Sensor
7	BSM Trig (Right)	Ultrasonic Sensor
8	BSM Echo (Right)	Ultrasonic Sensor
9	Motor Command PWM	DC Motor
11	E-Stop LED	LED
12	BSM LED (Left)	LED
13	BSM LED (Right)	LED
A0	Front Trig	Ultrasonic Sensor
A1	Front Echo	Ultrasonic Sensor

The decision to design the motor driver was made to add complexity to the hardware components. See Appendix C for the schematic and proposed PCB layout.

Software Design

The software architecture was developed with a high focus on modularity and state-based design to separate sensor data acquisition, control logic, and UI handling for ease of implementation and debugging. The main loop executes a fixed sequence of operations, ordered to prioritize safety checks followed by button handling, sensor data, and motor command.

The control algorithm implements proportional speed adjustment based on measured distance to the car ahead. The algorithm uses four discrete zones mapped to specific regions of motor operation to simulate a moving vehicle. Please refer to Appendix D for how the motor command is calculated.

5. Testing and Results

testing was essential to ensure that both the software and hardware components of the ACC and BSD platform operated reliably before being integrated into the full HIL system. The following subsections describe the specific testing approaches used for both software and hardware components of the system.

Software Testing

The software was tested using a bottom-up integration approach, validating each software component before integrating into the complete ACC system. Each module was tested independently using a separate test program with serial communication to monitor output for real time debugging and observation, allowing for ease of isolating defects to specific subsystems before complete integration into the main algorithm. Please refer to Appendix D for more information

Hardware Testing

To test the motor driver, a signal generator was used to produce square waves at 500Hz with various duty cycles. The motor was observed to identify a change in RPM while measurements at the positive and negative legs of the motor were taken to ensure proper amplification of a control PWM signal, as shown in Figure 4 below.

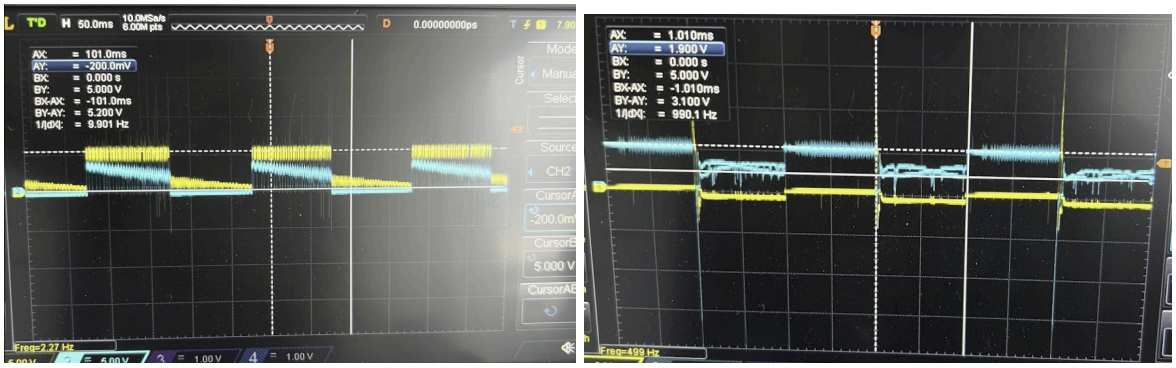

Figure 4. Motor Driver Behaviors Measured on an Oscilloscope

Performance Results

The end product can be seen in Figure 5. The Arduino, breadboard, motor driver, and motor were mounted on a wooden block, with the wheel facing upward for visibility.

Figure 5. Assembled System Being Tested

The system performed as expected. As the distance between the ACC vehicle and a leading vehicle decreased, so did the speed of the wheel. Additionally, a vehicle passing on either side of the system would trigger the blind spot detection LEDs.

Hardware Challenges

Due to timing and manufacturing constraints, the PCB for the motor driver had to be abandoned. The design was implemented on the perfboard shown in Figure 6.

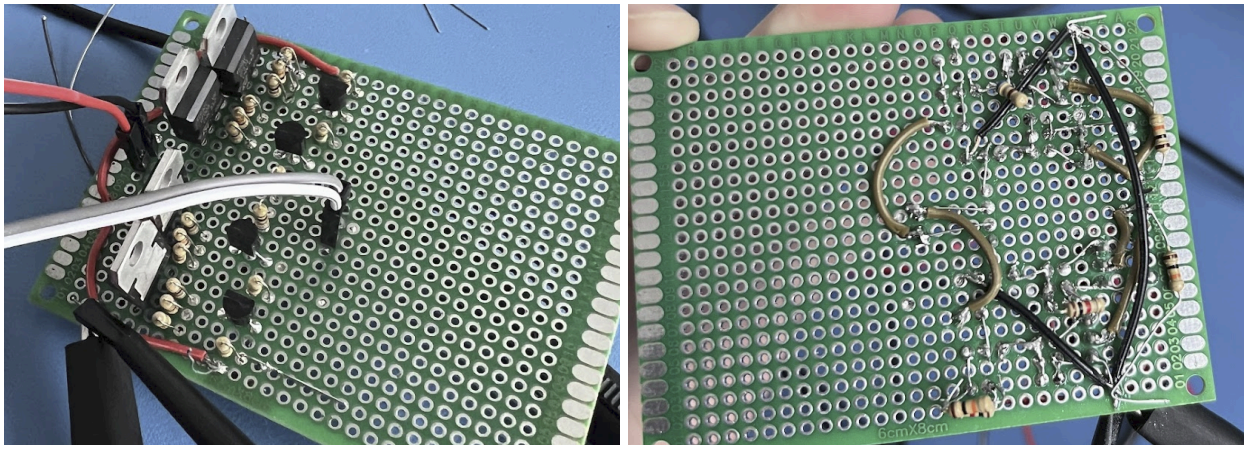

Figure 6. H-Bridge Motor Driver on Perfboard

This added significant difficulty to assembling the board, since all the connections had to be created with wires by hand.

6. Conclusion

This project successfully developed a hybrid HIL platform capable of testing Adaptive Cruise Control and Blind Spot Detection systems using embedded hardware. Despite challenges and scope adjustments, the final system met its functional objectives and provided a foundation for future expansion into more advanced ADAS features.

Appendices

Appendix A: BOM

Appendix B: SolidWorks Design

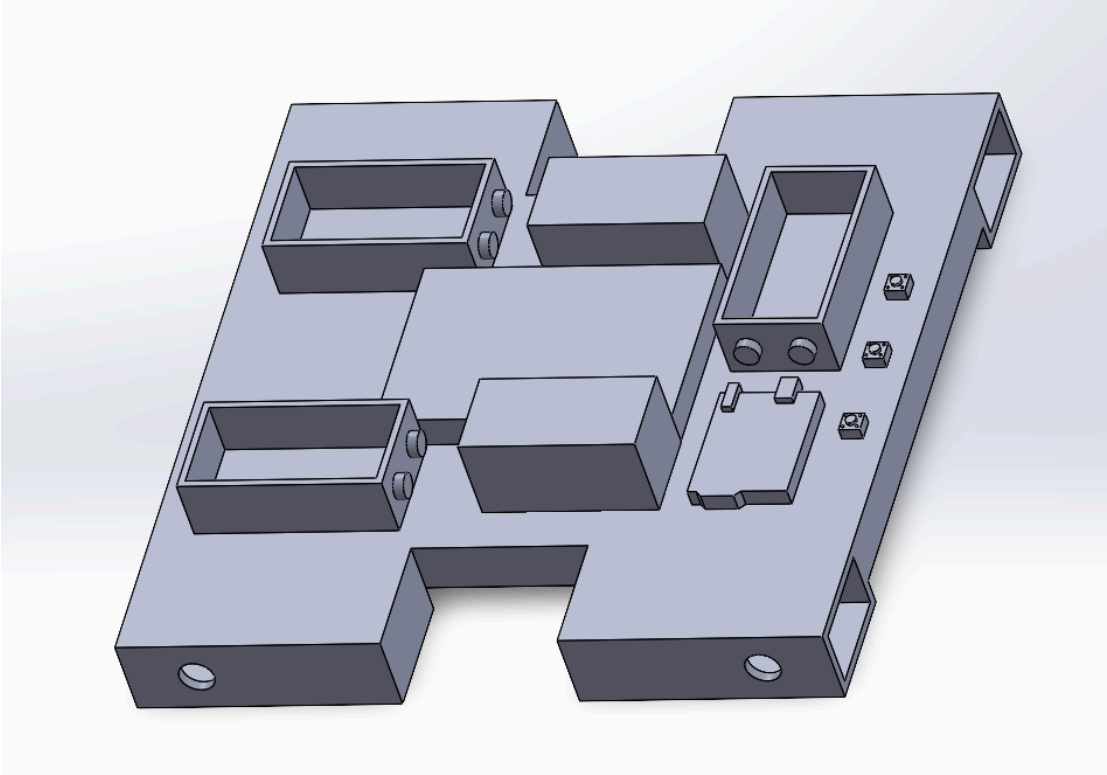

Figure 7: CAD design of the model EV

Appendix C: Motor Drive Schematic and PCB

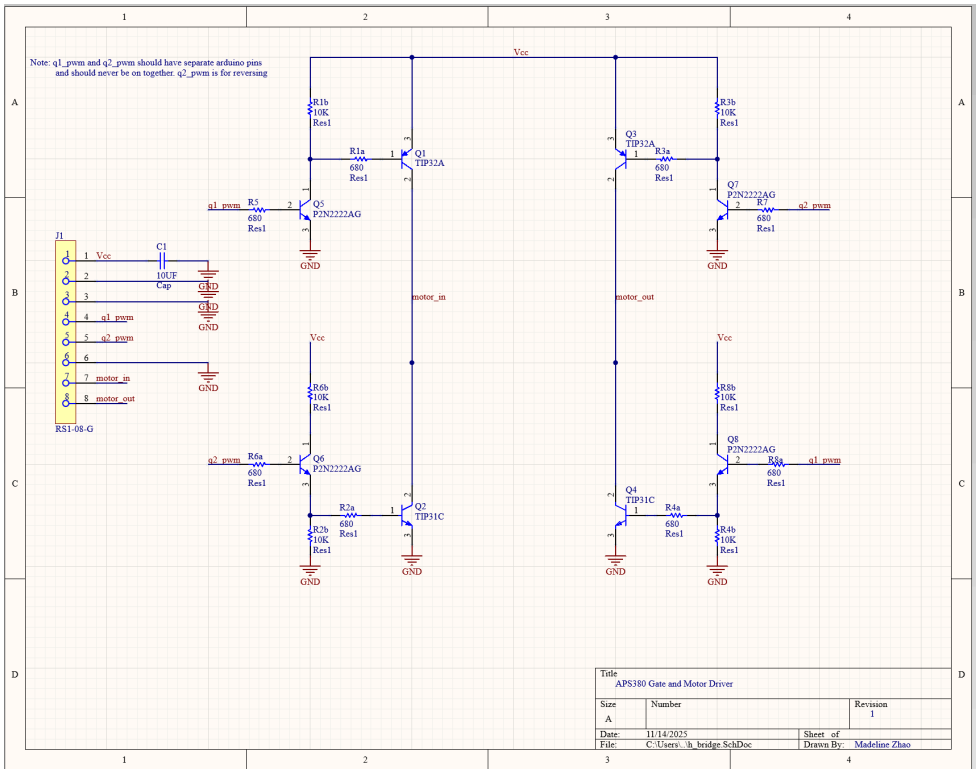

Figure 8. Motor Driver Schematic

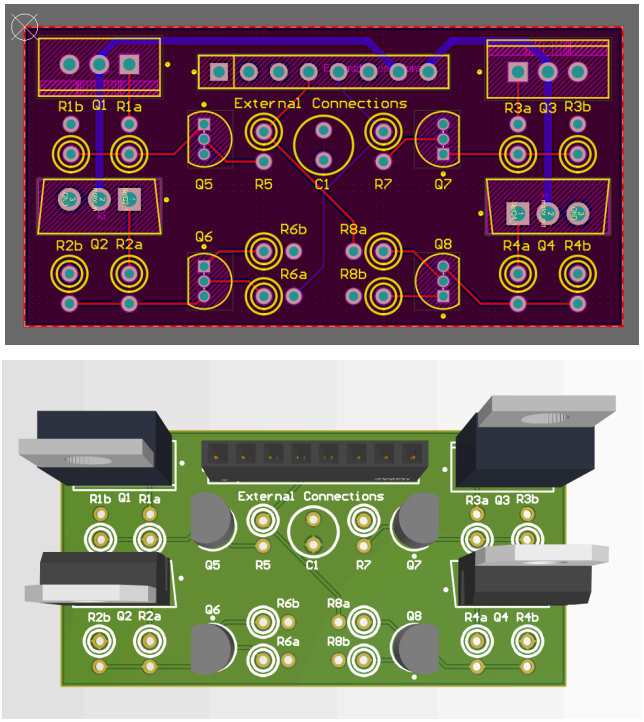

Figure 9: Proposed PCB Layout

Appendix D: Arduino Code Algorithms

```
void calculateMotorCommand() {
  int distance = (int)filteredDistance;

  // Case 1: No vehicle detected - cruise at full speed
  if (distance >= MAX_DETECTION_DIST) {
    motorCommand = CRUISE_SPEED;
  }
  // Case 2: Dangerously close - stop motor completely
  else if (distance <= DANGER_DISTANCE) {
    motorCommand = MOTOR_OFF;
  }
  // Case 3: Far away (beyond target) - cruise at full speed
  else if (distance >= targetDistance) {
    motorCommand = CRUISE_SPEED;
  }
  // Case 4: Between danger zone and target - proportional slowdown
  else {
    // Map distance to speed:
    //   At targetDistance → CRUISE_SPEED (255)
    //   At DANGER_DISTANCE → MIN_SPEED (150)
    motorCommand = map(distance, DANGER_DISTANCE, targetDistance, MIN_SPEED, CRUISE_SPEED);
  }

  // Constrain to valid range (but allow MOTOR_OFF for danger zone)
  if (motorCommand != MOTOR_OFF) {
    motorCommand = constrain(motorCommand, MIN_SPEED, CRUISE_SPEED);
  }
}
```

Figure 10: Calculating motor command PWM

```
void loop() {
  if (millis() - lastSample >= SAMPLE_INTERVAL) {
    lastSample = millis();

    // Read both sensors
    leftDistance  = measureDistance(LEFT_TRIG, LEFT_ECHO);
    rightDistance = measureDistance(RIGHT_TRIG, RIGHT_ECHO);

    // Check detection (valid reading AND within threshold)
    leftDetected  = (leftDistance > 0) && (leftDistance <= DETECTION_THRESHOLD);
    rightDetected = (rightDistance > 0) && (rightDistance <= DETECTION_THRESHOLD);

    // Update LEDs
    digitalWrite(LEFT_LED, leftDetected ? HIGH : LOW);
    digitalWrite(RIGHT_LED, rightDetected ? HIGH : LOW);

    // Print results
    printResults();
  }
}

// ============================================================================
// DISTANCE MEASUREMENT
// ============================================================================

long measureDistance(uint8_t trigPin, uint8_t echoPin) {
  // Send trigger pulse
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);

  // Measure echo duration (timeout after 30ms = ~5m max range)
  unsigned long duration = pulseIn(echoPin, HIGH, 30000);

  // No echo received
  if (duration == 0) {
    return -1;
  }

  // Convert to mm: distance = (duration * speed of sound) / 2
  // Speed of sound = 343 m/s = 0.343 mm/µs
  long distance_mm = (duration * 343L) / 2000L;

  return distance_mm;
}

// ============================================================================
// OUTPUT
// ============================================================================

void printResults() {
  // Left sensor
  Serial.print(F("Left: "));
  if (leftDistance > 0) {
    Serial.print(leftDistance);
    Serial.print(F(" mm "));
    Serial.print(leftDetected ? F("[DETECTED]") : F("[      ]"));
  } else {
    Serial.print(F("--- mm [      ]"));
  }

  Serial.print(F("  |  Right: "));

  // Right sensor
  if (rightDistance > 0) {
    Serial.print(rightDistance);
    Serial.print(F(" mm "));
    Serial.print(rightDetected ? F("[DETECTED]") : F("[      ]"));
  } else {
    Serial.print(F("--- mm [      ]"));
  }

  Serial.println();
}
```

```
  pinMode(EMERGENCY_STOP_PIN, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(EMERGENCY_STOP_PIN),
                  emergencyStopISR, FALLING);
}

void loop() {
  if (emergencyStopPressed) {
    Serial.println("EMERGENCY STOP ACTIVATED");


    delay(50);
    while (digitalRead(EMERGENCY_STOP_PIN) == LOW) {
      delay(10);
    }

    Serial.println("Emergency stop released, system would reset here.");


    emergencyStopPressed = false;


    delay(500);
  }
}
```

```
void handleUserButtons() {
  unsigned long now = millis();


  int incState = digitalRead(TARGET_INCREMENT_BTN);
  if (incState != lastIncBtnState) {
    if (incState == LOW && (now - lastIncBtnTime) >= BUTTON_DEBOUNCE_TIME) {
      incrementTargetDistance();
      lastIncBtnTime = now;
    }
  }
  lastIncBtnState = incState;


  int decState = digitalRead(TARGET_DECREMENT_BTN);
  if (decState != lastDecBtnState) {
    if (decState == LOW && (now - lastDecBtnTime) >= BUTTON_DEBOUNCE_TIME) {
      decrementTargetDistance();
      lastDecBtnTime = now;
    }
  }
  lastDecBtnState = decState;
}

void setup() {
  Serial.begin(115200);
  Serial.println("starting pushbutton test");

  pinMode(TARGET_INCREMENT_BTN, INPUT_PULLUP);
  pinMode(TARGET_DECREMENT_BTN, INPUT_PULLUP);

  Serial.print("Initial target distance: ");
  Serial.print(targetDistance);
  Serial.println(" mm");
}

void loop() {
  handleUserButtons();

  static unsigned long lastPrint = 0;
  if (millis() - lastPrint >= 1000) {
    lastPrint = millis();
    Serial.print("Current target distance: ");
    Serial.print(targetDistance);
    Serial.println(" mm");
  }
}
```

Figure 11: Modular testing protocol