# **CS 350 Final Project Report – Smart Thermostat**

Christian Busca  
 Instructor: Roland Morales  
 CS350: Emerging Systems and Technologies  
 Southern New Hampshire University

## **Overview and Context**

This report outlines the development of a smart thermostat prototype built using Python and designed for a Raspberry Pi platform. The thermostat system aims to simulate embedded control logic, integrating real-time sensor feedback, GPIO-driven user interface components, and serial output. The broader objective of this project is to demonstrate embedded systems integration within a smart device context, preparing the design for future cloud connectivity.

The system responds to environmental input (room temperature) and user controls to determine whether to initiate heating, cooling, or remain idle. State transitions trigger visual indicators and data reporting. While full hardware validation was delayed, the software development process and system architecture were thoroughly implemented and documented to allow future hardware deployment.

## **Functional Code Design**

The thermostat code supports multiple embedded interfaces. It utilizes I2C to communicate with an AHT2 temperature sensor and reads temperature data continuously. GPIO outputs control two LEDs, signaling whether the system is heating or cooling. Button inputs are handled through GPIO interrupts, allowing users to increase or decrease the temperature set point or cycle between OFF, HEATING, and COOLING modes.

The LCD component displays the system state, current temperature, and set point in real time. A UART interface transmits formatted sensor data to simulate server-bound communication. These features represent a working embedded system prototype with all required functionality designed for headless operation.

## **State Machine Architecture**

The system is driven by a finite state machine (FSM) that controls behavior across three operational states: OFF, HEATING, and COOLING. The FSM is initialized at system start and updates based on button input and sensor readings. Transitions are deterministic: pressing the mode toggle button cycles through states, and temperature comparisons govern the LED outputs. The FSM ensures a modular, extensible structure that cleanly separates control logic from hardware operations.

A detailed state diagram was developed to support the software design, documenting how transitions occur and what system actions are tied to each state.

## **Interrupt-Based Input Handling**

Three physical buttons are integrated using the GPIOZero library, each tied to an interrupt handler. These non-blocking callbacks allow users to:

* Toggle between OFF, HEATING, and COOLING states
* Increase the temperature set point
* Decrease the temperature set point

The use of interrupts ensures that user input is responsive, even while the thermostat continues to read sensor data and update the display and UART output concurrently.

## **Sensor Integration via I2C**

The AHT2 sensor is initialized through the I2C protocol using the smbus2 library. The system continuously polls temperature values in a timed loop. These readings inform the FSM logic and are also sent to the LCD and UART output. Initialization routines include error handling to account for disconnected sensors or I2C bus issues, making the system robust and resilient to I/O problems.

## **GPIO Peripheral Usage**

Two status LEDs indicate the active state. When heating or cooling is active, the corresponding LED fades in and out using PWM via the GPIO library. When the current temperature matches the set point, the LEDs remain solid, indicating no adjustment is required. This provides intuitive visual feedback and helps simulate the thermostat's real-world interface.

GPIO pins also handle input buttons with software-based pull-down resistors to ensure clean digital signals, minimizing hardware debounce issues.

## **UART Serial Communication**

To emulate communication with a central server, the system includes UART output. The UART is initialized with standard 9600 baud settings, transmitting JSON-formatted strings that include timestamps, temperature, and system state. This simulates real-world server reporting, which would eventually be replaced with HTTP or MQTT protocols in cloud integration.

## **FSM Functionality in Code**

All logic is implemented within a Python class that handles FSM transitions, sensor polling, input events, and peripheral updates. The FSM governs when and how LEDs are lit, the data shown on the LCD, and what is sent over UART. The code reflects the structure and behavior described in the project’s Lab Guide and ensures the prototype acts consistently with the required functionality.

## **FSM Documentation and Diagram**

A visual FSM diagram was developed using draw.io. This diagram outlines all states, transitions, and actions based on user input and sensor readings. It provides a comprehensive reference for both development and future hardware debugging. The diagram has been exported to PDF format for submission.

## **Coding Style and Best Practices**

The project code adheres to modern Python standards. Functions are modular, named clearly, and logically grouped. Comments are used extensively to document purpose and implementation rationale. The file structure separates configuration from logic, and PEP 8 formatting has been applied to ensure readability and maintainability. Error handling is in place for sensor access, GPIO communication, and UART output.

## **Hardware Platform Comparison and Recommendations**

Three platforms were analyzed: Raspberry Pi 4B, Microchip SAMD51, and Freescale NXP i.MX RT1060.

* **Raspberry Pi 4B** offers the most rapid prototyping environment. It supports Python, comes with built-in Wi-Fi, and offers full I/O support with minimal setup. It’s suitable for development and demo purposes.
* **Microchip SAMD51** is more constrained in memory but ideal for low-power production environments. It requires external Wi-Fi but supports I2C, GPIO, and UART via configurable SERCOM.
* **Freescale i.MX RT1060** has the most flash and RAM among embedded-class controllers and is a good candidate for feature-rich embedded deployments. However, development requires MCUXpresso or similar C/C++ tools and additional hardware.

Each architecture supports the necessary peripherals, but Raspberry Pi’s versatility, ease of use, and integrated networking make it the best platform for initial development. Microchip and Freescale platforms would be better for production-ready, embedded deployments due to lower power consumption and tighter hardware control.

## **Peripheral Support and Cloud Integration Capabilities**

All three platforms are capable of supporting I2C, GPIO, and UART either natively or through extensions. For cloud connectivity:

* Raspberry Pi offers seamless integration using Python libraries (paho-mqtt, requests) for direct interaction with cloud APIs.
* Microchip SAMD51 can use lightweight clients like uIP or MQTT with external Wi-Fi modules.
* Freescale i.MX supports more advanced stacks like LWIP or FreeRTOS + TCP with Wi-Fi extensions.

While cloud integration is possible on all platforms, Raspberry Pi simplifies development with built-in networking and open-source tooling.

## **Conclusion and Next Steps**

The smart thermostat prototype code meets all functionality specified in the project guide. While testing the physical prototype was not possible due to travel and a family emergency, the software architecture, FSM design, peripheral integrations, and cloud considerations have been thoroughly addressed. The system is ready for deployment to the Raspberry Pi once access is restored, allowing final validation, testing, and the creation of a demonstration video.