Final Project Report

Author:

Lacey Semansky

Course: CSCI 5143 - Real Time and Embedded Systems

Instructor: Jack Kolb

Date: Dec 12, 2024

1 Introduction

Indoor climate control has long been a goal for both houseplant enthusiasts and commercial growers. For this project, I aimed to develop a climate-controlled display case that ensures the health of the plant inside while offering an aesthetically pleasing experience for the user. This device addresses two key challenges: maintaining optimal plant conditions and reducing the need for continuous, hands-on care. In addition, it provides a platform for experimentation, allowing users to fine-tune environmental variables and explore ideal growth conditions.

To achieve this, I integrated sensors to monitor the current conditions, components to adjust the environment, and a smartphone application to manage the desired variables. In developing this project, I gained hands-on experience with several important embedded systems concepts, including input/output operations, I2C and UART communication, pulse width modulation, timer and interrupt handling, and real-time scheduling with FreeRTOS.

A major goal was to deepen my understanding of embedded systems and apply this knowledge in a practical setting. By incorporating an ESP32 development board, I also explored Wi-Fi connectivity, expanding the project's capabilities and demonstrating the potential of wireless communication in embedded systems.

2 Design and Implementation

2.1 Hardware

The hardware used in this project broadly falls into the categories of microcontrollers, data collection, environment adjustment, user interaction, and circuit functionality. Below is a detailed explanation of each component. See Figure 1 for the complete circuit design.

2.1.1 Microcontrollers

- ATmega3208 AVR-BLE Development Board: This board features 12 GPIO pins and an array of advanced peripherals. It was used as both a communication device and to control all states within the system.
- SparkFun Thing Plus - ESP32 WROOM: Espressif's ESP32 WROOM is a WiFi and Bluetooth MCU module featuring the ESP32-D0WDQ6 chip. The ESP32 Thing Plus integrates many peripherals including capacitive touch sensors, SD card interface, Ethernet, SPI, UART, I2S and I2C, etc. For my project, this board acted as a communication bridge between the Android application and the ATmega3208 AVR-BLE. It was also used to control several hardware components when capabilities of the ATmega3208 board were limited.

2.1.2 Data collection

- SHT30-D Temperature and Humidity Sensor: I chose this sensor because it collects both temperature and humidity, has a true I2C interface, and is encased in a hard protective shell. It also is one of the more accurate sensors at a low price-point with $\pm 2\%$ relative humidity error and $\pm 0.5^{\circ}$C error. In my project, this sensor acts as a client for I2C communication with the ATmega3208 microcontroller.
- TSL2591 Light Sensor: This sensor contains diodes to detect both infrared and full-spectrum light, which was useful for accurately calculating light available for photosynthesis. It also has a built-in ADC enabling simple digital readings by a microcontroller. In my circuit, this sensor also interacts with the atMega3208 over I2C communication.
- DS3231 Precision RTC: This real-time clock was used to detect when the system should switch between a daytime state and a nighttime state. I chose to use an external RTC to ensure accuracy over long durations. I also chose this RTC because it comes equipped with an interrupt pin and two separate alarms. This was incredibly helpful for ensuring that day and night states did not get confused.

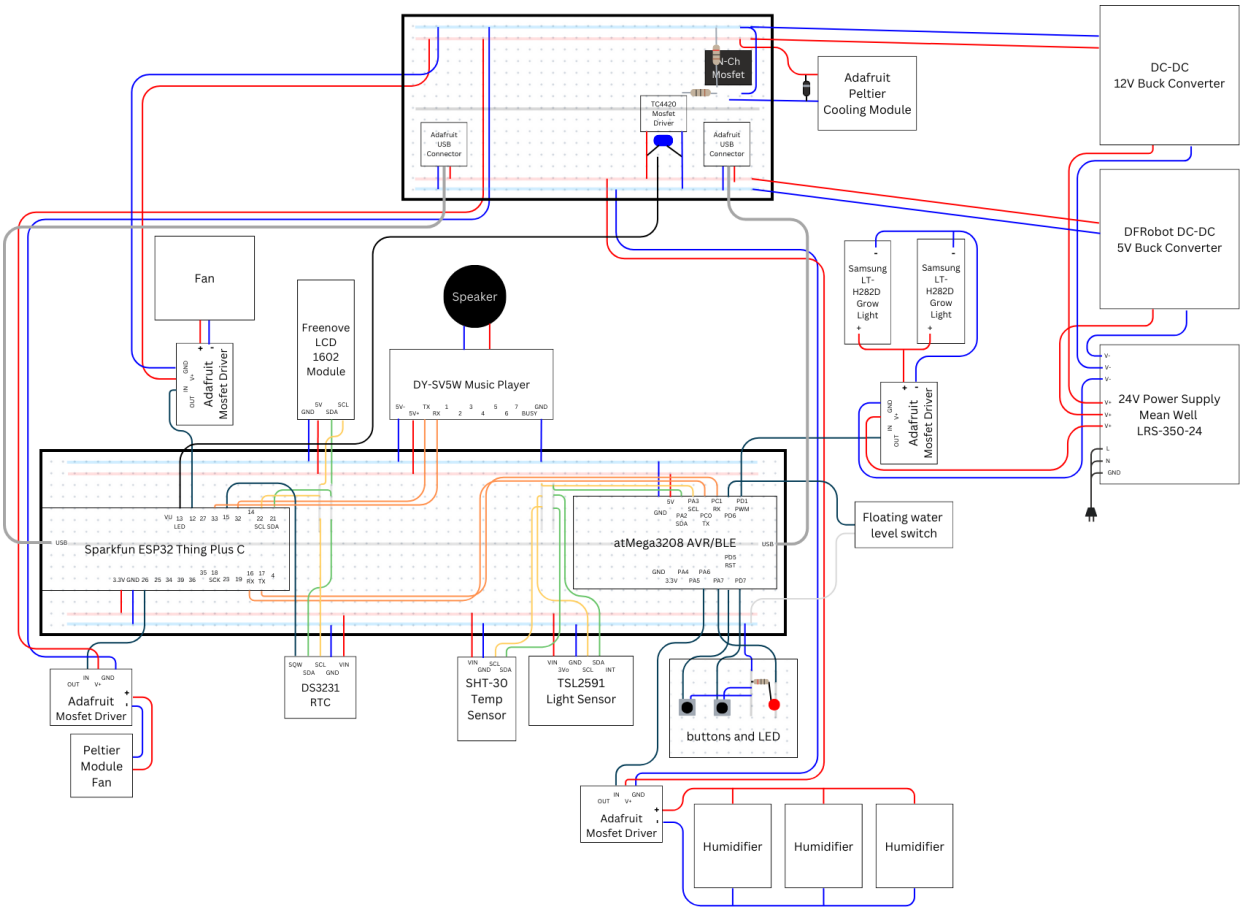


Figure 1: Complete circuit diagram

Finally, the external battery on this RTC module ensures that time only has to be reset if the battery dies or becomes disconnected. In my system, the RTC interacts with the ESP32 board over I2C.

- Floating Water Level Switch: This component is a normally open switch with a central floating element that rises and falls with the water level. When mounted top-side up, the floating component is pulled down by gravity and closes the switch if no water is present. This provided an ideal low-cost solution for detecting low water levels in the reservoir. In my project, the switch is connected to one of the ATmega3208's GPIO pins.

2.1.3 Environmental Adjustment

- Samsung LT-H282D LED Strip: After extensive research, these LED strips were found to be among the best in terms of efficiency and usable light output for photosynthesis. Each strip is capable of producing about 1000 lumens. I decided to use two strips to meet the light level requirements of the plant. In my circuit, the strips are controlled by an Adafruit MOSFET driver, which is connected to a 24V power supply and a GPIO pin on the ATmega3208.
- Adafruit Peltier Thermo-Electric Cooler Module: The Peltier device provides efficient heat transfer without moving parts. When an electric current passes through the junction of two different conductive materials, heat is absorbed on one side and released on the other. I chose this module for temperature control because it eliminates the need for moving liquids or evaporation, and can switch between heating and cooling by reversing the current. I selected the Adafruit module with an attached fan and heatsink to effectively dissipate heat from the back. The Peltier module is controlled by an

N-Channel MOSFET, with a TC4420 MOSFET driver to ensure full charge/discharge due to the high current. The MOSFET driver receives a PWM signal from a GPIO pin on the ESP32, which limits power to the cooling module. Using the ESP32's PWM functionality simplified the code, as the ATmega3208 was already handling more complex PWM operations for the grow lights.

- Atomization Disks: These devices create a fine mist by rapidly rotating a disk, breaking up water into tiny droplets. I chose this method of humidification because it doesn't require a fan. The disks I selected came with a controller module and mounting brackets, making setup straightforward. I connected three disks in parallel, all controlled by an Adafruit MOSFET driver, which receives a signal from the ATmega3208.
- 12V Brushless Fan: This fan is used in the enclosure to reduce temperature and humidity. I opted for a brushless fan due to its efficiency and quieter operation. The fan is switched on and off using an Adafruit MOSFET driver, with the signal sent from the ESP32.

2.1.4 User Interaction

- DY-SV5W Music Player: This low-cost music player was an ideal choice for this project, as it includes an audio amplifier, SD card slot, and supports UART or GPIO communication. Because the ESP32 communication occupied the ATmega UART bus, I initially tried using the GPIO switching function, where pins corresponding to tracks needed to be pulled low to play. However, I encountered issues with this method and did not fully investigate the cause. Instead, I connected the player to the second UART bus on the ESP32. This setup allowed for control of loop mode, volume, and EQ, and enabled adding more tracks without increasing the number of GPIO pins required. A simple 5W speaker was also connected to the DY-SV5W.
- FreeNove 1602 LCD Module: This LCD screen is capable of drawing two lines of text each at sixteen characters. This was just the right size for displaying temperature and humidity readings while maintaining a compact footprint. I chose to add this to the ESP32 I2C bus. Information about the I2C commands for this device were limited and my attempts to write my own functions for the atMega3208 to use failed. Instead I chose to use a library with the intent to revisit this if more time became available.
- Buttons and LED: Two simple press-buttons were used to control the system's light and music modes. A red LED was used to indicate low levels of water in the reservoir. These three items were all connected to individual GPIO pins on the atMega3208.

2.1.5 Circuit Functionality

- Adafruit MOSFET Driver Module: This module is equipped with an AO3406 N-Channel MOSFET, a 1N4007 flyback diode, and simple connection types. This module simplified my circuit significantly by providing everything needed to easily switch devices on and off with a low voltage microcontroller pin. I chose to use MOSFET switching for its efficiency and PWM capabilities. My circuit used four of these devices for the fans, humidifiers, and grow lights. Because the rated amperage for the A03406 MOSFET is slightly below what was needed for the Peltier cooling module, I could not use another Adafruit driver and had to design the circuit more directly.
- TC4420 MOSFET Driver and IRLZ44NPBF MOSFET: This MOSFET is rated for much higher current than the Adafruit module MOSFET. I used this to switch the Peltier device on the low side for simplicity of circuit design. The TC4420 driver was used to amplify the input signal from the microcontroller. Because the MOSFET is logic level, the driver was not entirely necessary but I chose to include it to ensure clean switching and minimize overheating of the MOSFET.
- Mean Well LRS-350-24 Power Supply: This power supply outputs 24 DC Volts, and 350 Watts, which was more than enough power for my circuit. It also incorporates power regulation and short circuit, overload, over voltage, and over temperature protections.

- DC-DC Buck Converters: I used two converters to bring the voltage down to 5 and 12 Volts for various components of my circuit. These converters were chosen because of their efficiency compared to regulators or other options, and the ability to easily adjust the amount of output they provide.
- Adafruit USB Breakout Board: This board provides a USB connection that can be through-hole wired. I used it with both microcontrollers to ensure safe power delivery through the regulators on the ATmega3208 and ESP32 via their USB connections.

2.2 Software

Broadly, the software for this project is designed to collect sensor readings, gather user-defined threshold settings, and adjust outputs to align the climate with the specified settings.

Three states are incorporated into the system: the "all-on" state, where all outputs such as lights, fans, and humidifiers are active; the "lights-off" state, which keeps all outputs running except for the grow lights, LCD screen, and water-level indicator LED; and the "night-mode" state, which disables all outputs and makes the system appear powered off. These states are controlled by a button, as well as timer interrupts for user-defined "wake-up" and "power-down" times.

The system also includes a music feature, with track 1 playing during normal operation and track 2 during watering. The music is toggled by a button, turned off when entering "night-mode," and resumed when returning to "all-on" if it was playing before entering "night-mode."

2.2.1 ATmega3208 Program

The ATmega3208 serves as the backbone of the project. It collects data from the sensors and ESP32, then calculates and controls the necessary outputs.

Its program includes files designed as libraries for communication with the light sensor and temperature/humidity sensor over I2C, as well as with the ESP32 over UART. Additional files were initially created for managing communication with the music player and LCD display but were later discarded when those devices were moved to the ESP32.

At the core of the ATmega3208 code are six FreeRTOS tasks, detailed below. Sensor data and threshold settings are organized into structs and protected by mutexes. The FreeRTOS event group data structure is used as a safe way to read and write state flags, such as light state, music on/off, water level, and fan control. This data structure was incredibly useful for the project. It consolidates all states into a single 32-bit variable, provides methods that guarantee safe read and write operations, and provides a method to block a task until a specified bit is set.

- Read sensors: This task serves as the heartbeat of the system. It collects light, temperature, and humidity readings, then updates the sensor data struct. Once it has finished, it releases a semaphore shared with the communication task to allow communication with the ESP32 to commence. The task then delays for one second to conserve processing power.
- Communication: This task manages all communication with the ESP32. After acquiring the communication semaphore, it collects sensor readings, threshold settings, and state data, then transfers this information to the ESP32. It subsequently receives data from the ESP32 and updates the global data based on the received information. This may include new threshold settings or notifications such as a triggered day/night shift alarm. The communication protocol is discussed in detail in Section 2.2.3.
- Button control: This task handles both the music and state buttons. If the music button is pressed, it toggles the music flag in the event bit group, which is used in the communication task. If the state button is pressed, it toggles the bits to set the next state in the rotation. It also toggles the music bit according to the state and whether or not music was previously playing before "night-mode" was entered. Button debouncing is handled by a small delay at the end of the task.

- Humidity control: This task first checks if the state is "night-mode." If so, it calls the xEventGroupWaitBits method to block until a different state is entered. Once this condition is met, the task checks whether the water level switch is open or closed. If open, it sets the "water-level-okay" bit and proceeds to compare the current humidity sensor reading with the minimum humidity threshold setting. If the current humidity is too low, it initiates the watering phase by turning on the humidifiers, setting the "watering-mode" bit (used in the communication task to request music track 2), delays for approximately two minutes to allow the humidifier to run, then turns off the humidifier and clears the "watering-mode" bit. I chose to hard-code the watering time because I created a music track that aligns with the watering phase.
- Temperature control: This task first checks the "night-mode" bit and ensures that the fan and cooling are turned off if "night-mode" is entered. It then blocks until either the "all-on" or "lights-off" states are entered. Once no longer in "night-mode," it compares the current temperature reading with the maximum temperature threshold setting. If the current reading exceeds the threshold, it sets the "cooling-mode" bit, which is communicated to the ESP32 in the communication task. It also checks if the humidity exceeds the maximum humidity threshold. If so, it sets the "fan-on" bit, which is also communicated to the ESP32.
- Light control: Similar to temperature control, light control checks for "night-mode" or "lights-off" to switch off lights in these states. It then blocks until the "all-on" state bit is set. Once this condition is met, it will check the "water-level-okay" bit to toggle the water-level-indicator led. It then compares the current light sensor reading to the desired light setting, calculates the change needed, and sets a new value in the Timer-A compare register to adjust the pulse width modulation duty cycle.

2.2.2 ESP32 Program

The ESP32 performs several key roles within the system. First, it utilizes the module's "station mode" WiFi functionality to connect to a local network. Once connected, it hosts a simple web server with a GET request for retrieving current sensor readings, and a POST request for updating threshold settings. This functionality was implemented using Arduino's WiFi library and the ESPAsyncWebServer library. These libraries significantly abstract any low-level operations necessary for working with the ESP32's WiFi capabilities. As a result, the implementation closely resembles the structure of a basic JavaScript server. In the future, I plan to rewrite this code without relying on these libraries to gain a deeper understanding of the underlying processes.

Secondly, the ESP32 acts as a communication bridge with the ATmega3208. Its code mirrors the ATmega's side of the communication protocol. The ESP32 receives sensor readings to send over WiFi, state information to adjust peripherals, and sends RTC alarm information and any new threshold settings received via WiFi. It continously polls the USART RX line to detect when the ATmega3208 initiates the next communication sequence.

Lastly, the ESP32 manages communication with additional hardware components, including the RTC, LCD, music player, fan, and Peltier cooling module. The Adafruit RTClib library is used to communicate with the RTC. When day and night times are received over WiFi, new alarms are set on the RTC. If either alarm is fired, the RTC sends an interrupt to the ESP32, which forwards this information to the ATmega3208.

The music player is controlled using the UART commands found on its datasheet. Information received from the ATmega determines which track will play, or if the player must turn on or off. The LiquidCrystal_I2C library is used for the LCD screen. Current sensor readings are written to the display after each communication sequence with the ATmega. Data retrieved in this sequence also determines whether the LCD, fan, or Peltier device need to be turned on or off.

While I relied on libraries for these functions due to time constraints, I aim to reduce my dependency on them in the future to learn more about these devices.

2.2.3 ATmega and ESP32 Communication Protocol

The communication protocol warrants a detailed explanation, as extensive trial and error led to the development of a somewhat unconventional solution. My initial approach was to use I2C in order to reserve the UART bus for the music player. This led to a quirky I2C interaction where the ATmega3208 acted as host and ESP32 acted as client. If the ESP32 needed to send data, it would use a GPIO pin to send an interrupt to the ATmega, which would respond by initiating a "what do you want?" command, then reading the data from the ESP32.

However, I encountered significant challenges due to limitations in the Arduino IDE. Despite the ESP32 having two I2C buses, the available I2C libraries only supported one. Since the ESP32 was already operating as a host for other devices, I needed a second bus for client communication with the ATmega. Addressing this limitation would have required transitioning to a different development environment to gain low-level control over the ESP32's hardware. Ultimately, I decided to abandon the I2C-based approach in favor of UART.

The first iteration of UART communication included individual commands for each interaction. Each method was prefaced by a command byte so the receiving device knew what information would follow. Sensor readings would be sent once a second, and all other communication would occur as needed. This worked most of the time but occasionally the system would mix up bytes from different commands. I suspected that this was due to the 16 byte RX buffer on the ESP32 side. I believe if I were able to change the buffer to just one byte, I would not have the same problem. Unfortunately, this again would require moving to a different development environment.

The final iteration of the communication protocol is perhaps the worst, but works every time without fail. This method involves both devices sending and receiving all information once per second in a single sequence of transmissions. First, the ATmega sends it's sensor readings in four bytes. Next it sends four bytes of state information including which light state the system is in, whether the cooling device should be on or off, whether the fan should be on or off, and what state the music player should be in. Next the ESP32 sends one byte indicating which day/night alarm was triggered, if any. Then it will send one byte as either a 1 or 0 indicating whether it has new threshold settings to send. Finally, if a 1 was sent the ESP32 will transmit the settings data to the ATmega in 6 bytes.

2.2.4 Android Application

The Android application serves two main purposes: displaying current sensor data and allowing users to adjust threshold settings. These functionalities are implemented as two separate screens, with navigation handled through a tab row. On startup, the application attempts to make a GET request to the ESP32. If successful, the fetched sensor data is displayed. If no connection is made after five seconds, a timeout screen appears with an option to retry. While the sensor readings screen is active, the app will continue to make GET requests every second to keep the data up-to-date.

The settings tab features card objects with sliders that enable users to adjust threshold settings. There are also time-pickers for setting the day and night times. When the submit button is tapped, the app sends a POST request to the ESP32. If the data is successfully updated, a success screen is shown, and the app navigates back to the sensor readings screen. On failure, an error screen will appear. The settings page also includes preset functionality, allowing users to select from two preset configurations via a drop-down menu. These presets automatically populate the threshold settings, which can then be adjusted further as needed. Table 1 contains images of different screens within the application.

This application was developed in Android Studio using Kotlin and Android's Jetpack Compose toolkit. Data classes are used to model the ESP32 response and request data, the UI state, and preset values. The UI state is represented as a singleton object containing all of the data that is displayed in the app, as well as flags such as isLoading, and isTimeout. A ViewModel class stores and manages the UI state object, ensuring data consistency during configuration changes, such as screen rotations. The ViewModel also includes functions for communicating with the ESP32 and updating the UI state, providing a clear separation of concerns between the UI and the app's data logic.

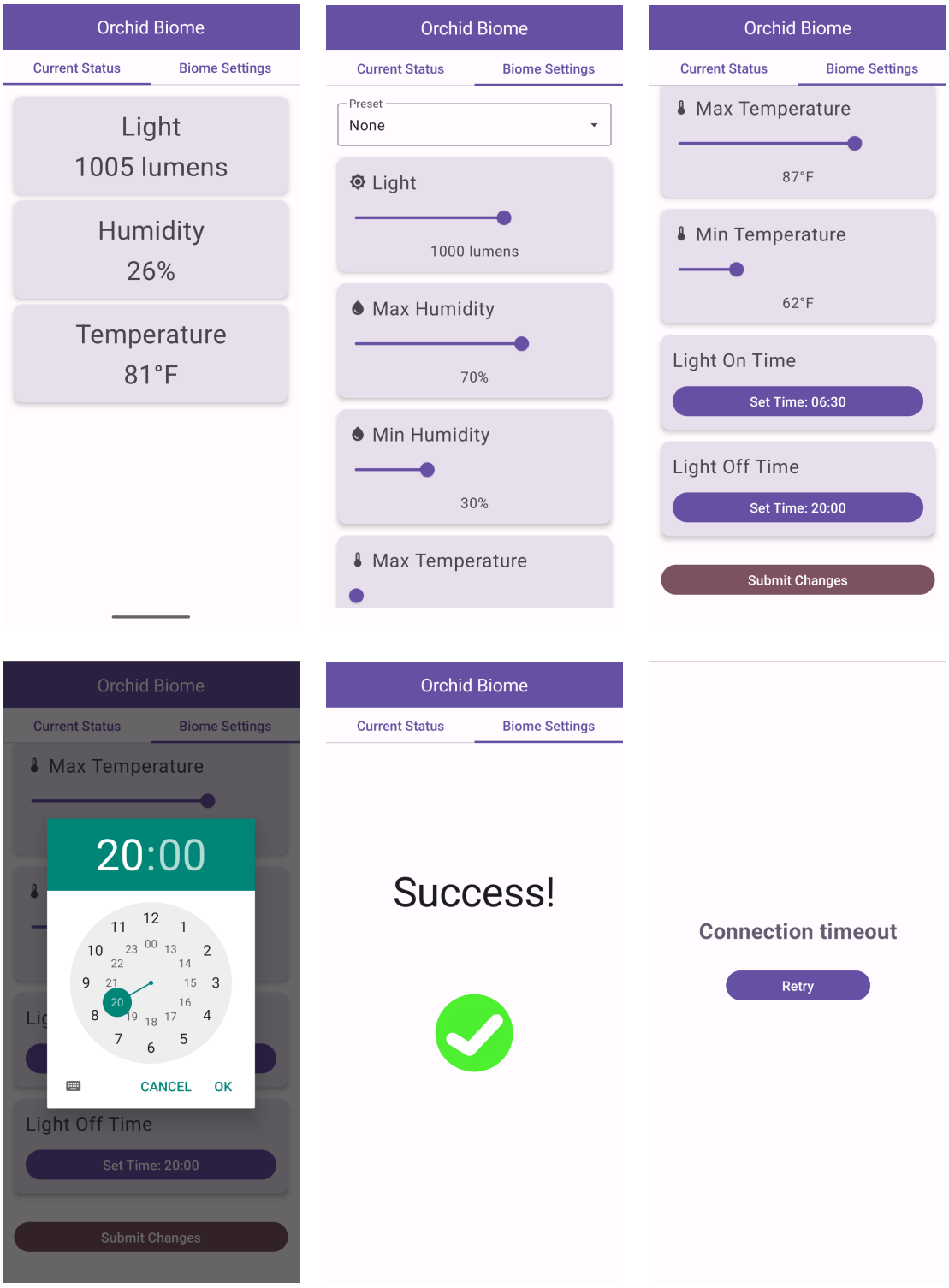
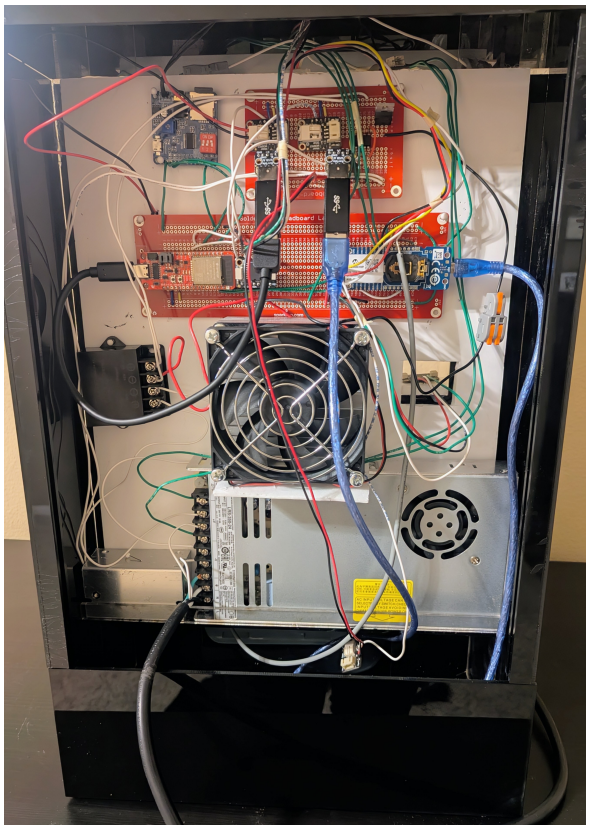


Table 1: Android Application Screens

3 Final Product

3.1 Construction

For the physical construction of this project, I used 1/8" acrylic sheets in both clear and black. A laser cutter was used to create precise pieces that could be glued together with acrylic welding glue. I used a plastic container for the water reservoir, and several magnets to hold the front piece and the plant in place. I also incorporated a metal funnel and some tubing to add water to the reservoir. Hot glue and silicone caulk were used to mount various pieces and keep water contained within the reservoir and display. Electronics were mounted on foam board and housed in the back. Finally, plastic mesh was used to cover the back but still provide airflow. Below are images of the final product.



3.2 Goals

The only component I was unable to fully implement was the physical wiring of the Peltier cooling module. Unfortunately, the MOSFET driver used to control the fan broke during development. Without adequate heat dissipation from the fan, I was concerned that the Peltier module would be ineffective and could overheat, potentially stressing the rest of the system. I plan to integrate this component as soon as I acquire a replacement MOSFET driver.

I met nearly all of my initial goals for this project. I developed a responsive system capable of taking sensor readings and adjusting outputs accordingly. Additionally, I achieved my stretch goal of creating a smartphone application, as well as my goal to enhance the user experience with features like music and a display.

The only component I was unable to fully implement was the physical wiring of the Peltier cooling module.

Unfortunately, the MOSFET driver used to control the fan broke during development. Without adequate heat dissipation from the fan, I was concerned that the Peltier module would be ineffective and would stress the rest of the system with heat not being moved out. I plan to integrate this component as soon as I acquire a replacement MOSFET driver.

4 Reflection

Generally this project went fairly smoothly. I adopted a "slow is fast" approach when doing research, ensuring that by the time I began implementation, I had a clear and well-structured plan, particularly for the hardware setup and physical construction. If I were to do this project again with the knowledge I have now, I would opt to use the native ESP32 IDF instead of the Arduino IDE. I often found myself needing more low-level control of the ESP32 that I could not access in Arduino.

Looking ahead, there are several areas I plan to expand upon in this project. First, I intend to finalize the setup of the Peltier cooling module and incorporate an H-bridge to reverse the current, enabling the module to provide heat to the enclosure as needed. I also aim to enhance the climate control system to improve its precision and reliability. This would involve implementing more sophisticated control over the humidity and temperature variables. Additionally, I originally envisioned integrating a camera to monitor the plant's growth. With the existing smartphone connectivity, it would be possible to incorporate image processing to track growth patterns and analyze the effects of environmental variables.

This project provided a valuable opportunity to apply and deepen my understanding of embedded systems. I was able to successfully integrate several hardware components and a smartphone interface to develop a functional climate-controlled plant enclosure. Throughout the process, I gained hands-on experience with real-time scheduling, I2C and UART communication, PWM control, and Wi-Fi-enabled data exchange. While there's still room for improvement, this project has been an important step in developing my skills and exploring the possibilities of climate-controlled technology.