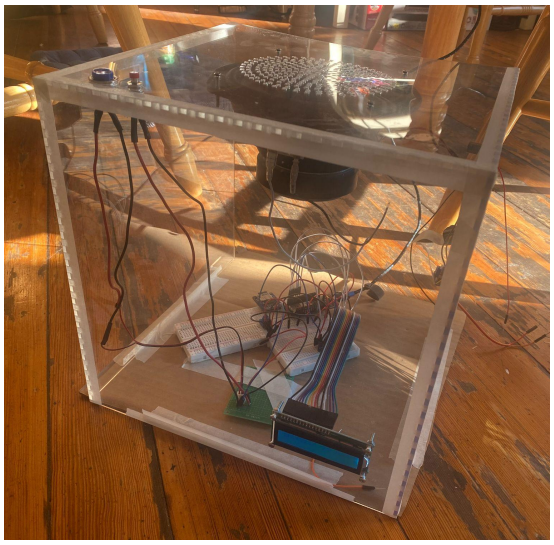
Daily Weather Reports via Audio-Visual Alarm Clock
cs1600 Capstone Project



As college students who struggle to get up and moving promptly each morning, we decided to build an audio-visual alarm clock to make this process just a little bit easier. The alarm clock has soothing lights (not pictured) and a large speaker that plays the user's desired song at their desired wake up time. We also built a web interface to allow users to set alarm times and snooze durations, as well as upload audio files to serve as the alarm's sound.

For my capstone component, I augmented our wake up message with current weather information so that users can be better prepared for the day ahead. I chose this additional functionality because I believe that integrating this information directly into our alarm clock will help streamline users' morning routines.

Design Challenges:

My capstone component added complexity to our project in a number of ways. The primary challenge was determining how and when to make API calls via the Arduino that was powering our alarm clock. This challenge included understanding how to actually make the API calls as well as the permissions required to make these calls, along with considerations like minimizing unnecessary calls, but ensuring that data was up to date and ready to be displayed when the alarm clock entered its "alarming" state.

After successfully retrieving the current weather report, one final design challenge was determining how to parse this data, ultimately transforming it into a succinct message that would fit on one line of our LCD screen, as shown on the right.

Approach:

My weather data came from calls to the free open-meteo weather API. I decided to request the daily high and low temperature, along with the WMO weather code corresponding to the daily forecast. I also wrote code to interpret the WMO code and generate a one-word descriptor of the daily weather.

One key issue that arose was determining when API calls should be made. My initial plan was to minimize the number of calls by only making a call when the alarm clock's FSM entered the "alarming" state, and then immediately using the resulting weather data in a display function to display a weather report on the LCD screen when the alarm went off. However, I realized that the time required to make this call as the FSM transitioned to the "alarming" state was too significant, so the display function could end up lagging and I couldn't guarantee that a weather report would be immediately displayed. I decided to circumvent this obstacle by making the weather GET request when all other GET requests were made for alarm settings, as the alarm clock FSM transitions into the "process updates" state. Since this state is guaranteed to be entered every 10 minutes but no less than 10 minutes before the alarm time, I can then guarantee that the weather data will still be current, but it will also be received, parsed, and stored far in advance of it needing to be displayed on screen.

Another key issue that I needed to address was how to make the actual API request. I spent a significant amount of time trying to get the Arduino to make a GET request to the free weather API, but consistently ran into permissions issues. However, I was able to successfully make GET requests to the web application I built for users to set alarm settings. Because of this, I ultimately decided to retrieve the weather data in a somewhat unconventional way. I used my web application as an intermediary, adding another endpoint to this API that essentially just makes the weather GET request itself. As a result, I was able to obtain the weather data without having to make a direct GET request to open-meteo itself, thus avoiding the permissions issues.