E INK DESK CLOCK

PARTH KHARADE

FINAL PROJECT – ECEN-5613 – FALL 2023

Table of Contents

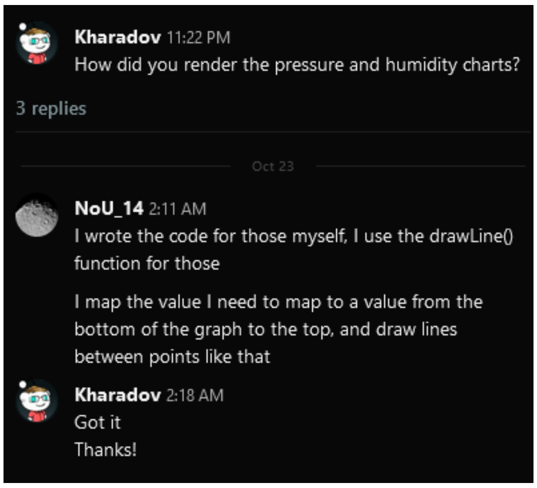
Figures

Overview

I have been intrigued by E-Paper displays ever since I got a Kindle in 2020. But it was not until last year that I discovered that WaveShare sold E-Paper Modules for hobbyists.

I have been wanting to make a project based on an E-Ink display since then but couldn't find the time to do so. The final project for the course demanded new hardware and a new software component and this presented me as a perfect opportunity to integrate an E-Ink Display into my project.

It had been decided that I was going to center my project on an E-Ink Display, in fact I had even ordered one before deciding what I was going to do with it. I was browsing the internet for ideas when I came across this beautiful project by u/unblended_melon on reddit. I loved how clean and natural the interface looked on an E-Paper display. This project by u/NoU_14 looked beautiful as well.


Kharadov 11:22 PM
How did you render the pressure and humidity charts?

3 replies

Oct 23

NoU_14 2:11 AM
I wrote the code for those myself, I use the drawLine() function for those

I map the value I need to map to a value from the bottom of the graph to the top, and draw lines between points like that

Kharadov 2:18 AM
Got it
Thanks!

Figure 1 : Conversation with u\NoU_14 on reddit.

I started going through the GitHub repository of the project to understand the various components involved and the level of complexity. I also reached out to u/NoU_14 to understand the details of their project. I saw a common thread in both these projects. They both used Adafruit GFX and GxEPD2 as their core library for graphics. This might have allowed them to focus on other aspects of their projects such as the beautiful charts and the IoT based features. I decided to write my own basic graphics library instead and deemed the IoT and the advanced graphics library to be the "bonus" objectives of my project. I decided to do this as I believed that it would help me understand the basics of writing display drivers better. Once I have been through the "rite of passage" of working with displays I could then leverage third party drivers.

Furthermore, this also helped me delve deeply into the technicalities of E-Ink displays such as Waveforms and the other problems associated with E-Ink Displays. While I cannot claim to be an expert on E-Ink Displays I know a lot more than I knew 3 months ago.

I initially planned to make an IoT device like u/unblended_melon's but had to ditch the idea because I ran into some complications using ESP-IDF. Instead, I ended up making an alarm clock with a buzzer and 4 user configurable alarms which displayed the current room temperature and humidity.

This report gives an outline of the major hardware and software components of my projects, the issues I faced and the compromises I had to make. I highly recommend reading this article and watching this video to get a good understanding of why exactly E-Ink displays are tricky. Happy reading and sincere apologies for any grammatical and typographical errors.

Hardware Components

I wanted this project to be as cost effective as possible. I chose a simple procedure for choosing products. I went on Mouser.com , searched for a particular product category and then sorted the results by price, package, and availability and then I chose the cheapest one that fit my requirements. While I was looking for cheap products, their "solderability" was an important factor as well. I was ready to pay a few extra bucks for a TSSOP over a SON.

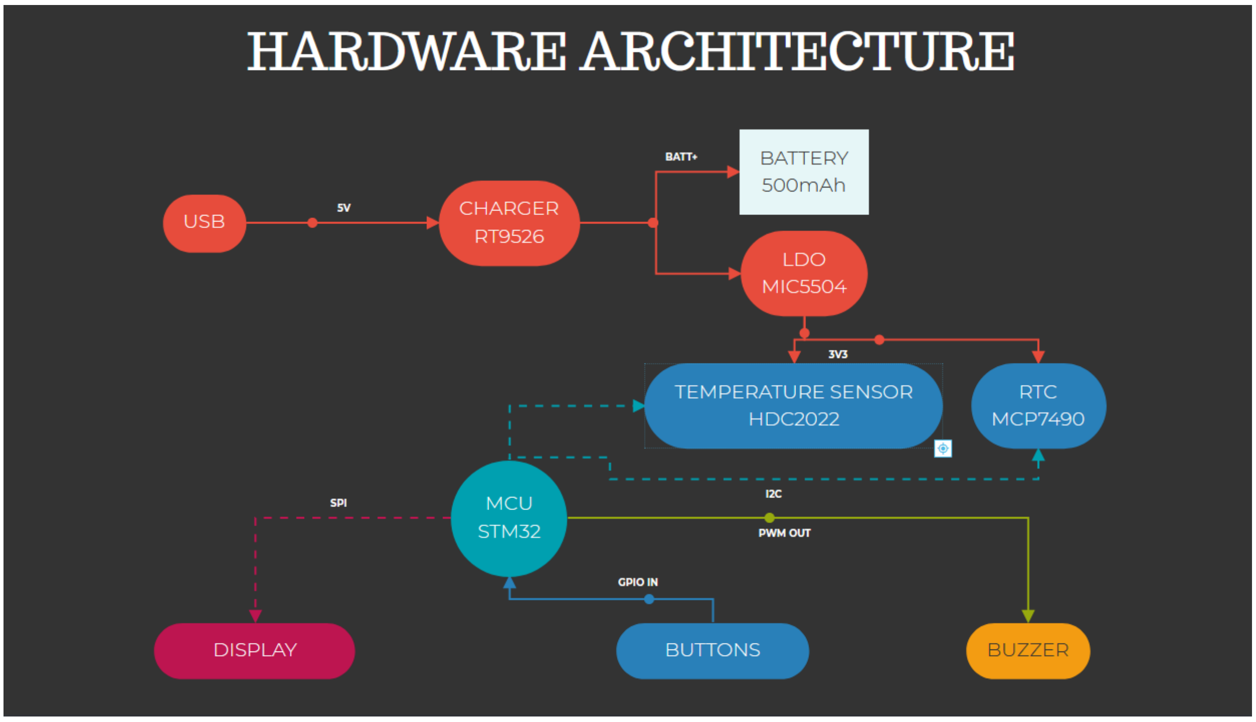


Figure 2 : Hardware Architecture

Real Time Clock - MCP7490

This RTC from microchip is driven by a standard 32768Hz crystal. It also features trimming registers to calibrate from crystal inaccuracies. Furthermore, it also provides a 64 Byte battery backed SRAM which can be used as a user-memory.

I initially planned to use this memory to store the 4 user-programmable alarms that I needed. I wasn't going to use the in-built alarm system as it only had functionality for 2 alarms.

Currently, my alarms are stored on the microcontroller's volatile (RAM) memory. This also means that I must leave the microcontroller powered-on to avoid erasing the user-programmed alarms. By using the SRAM, I could have significantly reduced the power consumption of the system.

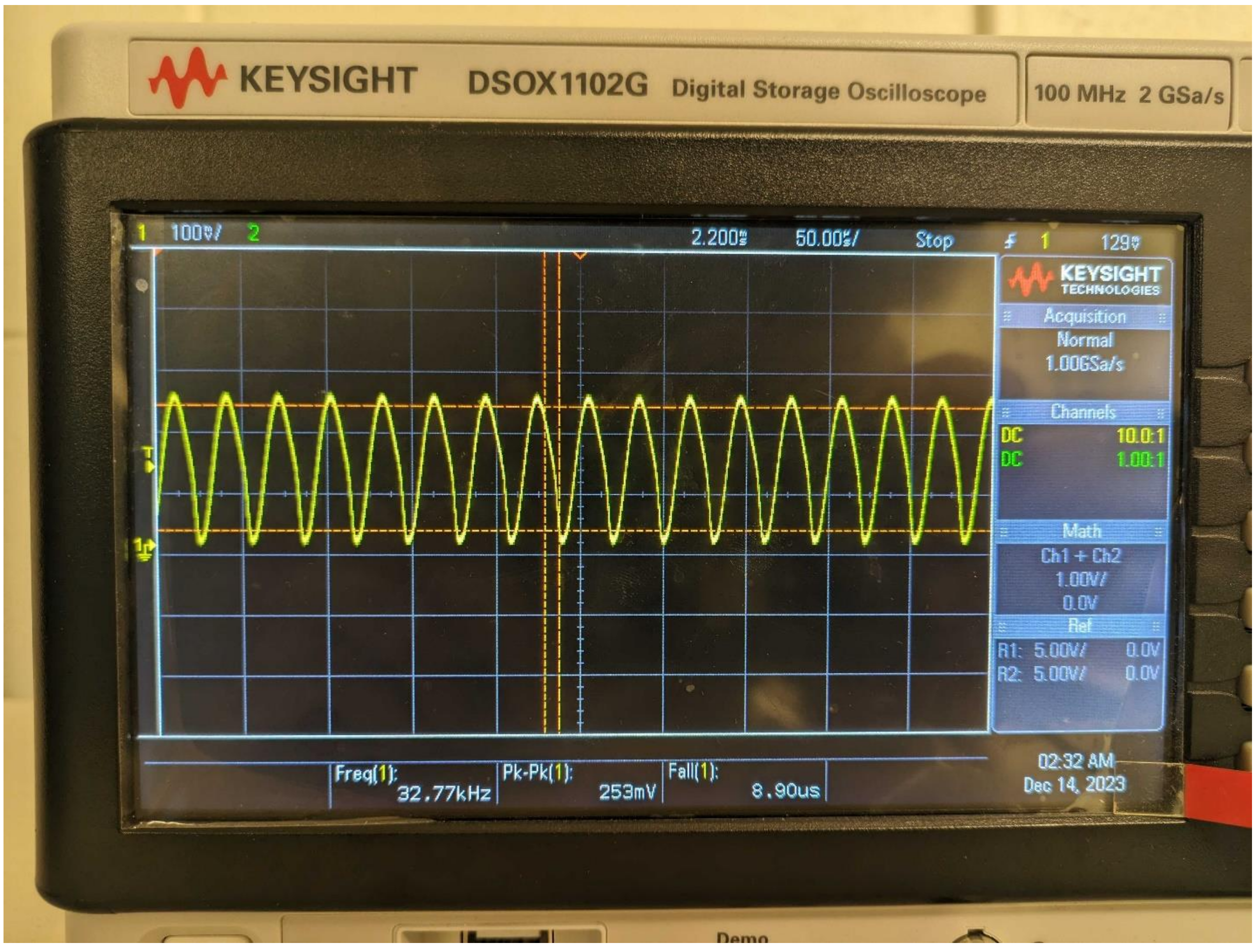

KEYSIGHT DSOX1102G Digital Storage Oscilloscope
100 MHz 2 GSa/s
Acquisition Normal 1.00GSa/s
Channels
Math Ch1 + Ch2 1.00V/ 0.0V
Ref R1: 5.00V/ 0.0V R2: 5.00V/ 0.0V
Freq(1): 32.77kHz
Pk-Pk(1): 253mV
Fall(1): 8.90us
02:32 AM Dec 14, 2023

Figure 3 : Oscillator at 32768Hz.

Single Cell Lithium Ion Charger – RT9526

An integrated single-cell lithium-ion charger from RICHTEK with a programmable charge current and cut-off current.

Calculations for Maximum Charge current.

- Assuming an ambient temperature of 25^oC, maximum power dissipation = 0.48W (Datasheet Page 10).
- Assuming a battery voltage of 3.6V and input voltage of 4.2V, PD = (5-3.6)*I_{charge}= 1.4*$I_{charge.}$
- Then 1.4*I_{charge} < 0.48.
- I_{charge} < 342mA.

The battery had a standard charge current of 0.5C which is 250 mA. With a safe margin I decided to go for a charge current of 200mA. The battery demands a termination charge current of 0.01C(5mA), I made a calculation mistake while ordering components and ended up with resistors which gave me a termination current of 20mA instead. This would mean that the battery charging is terminated before the required spec and the battery does not completely charge.

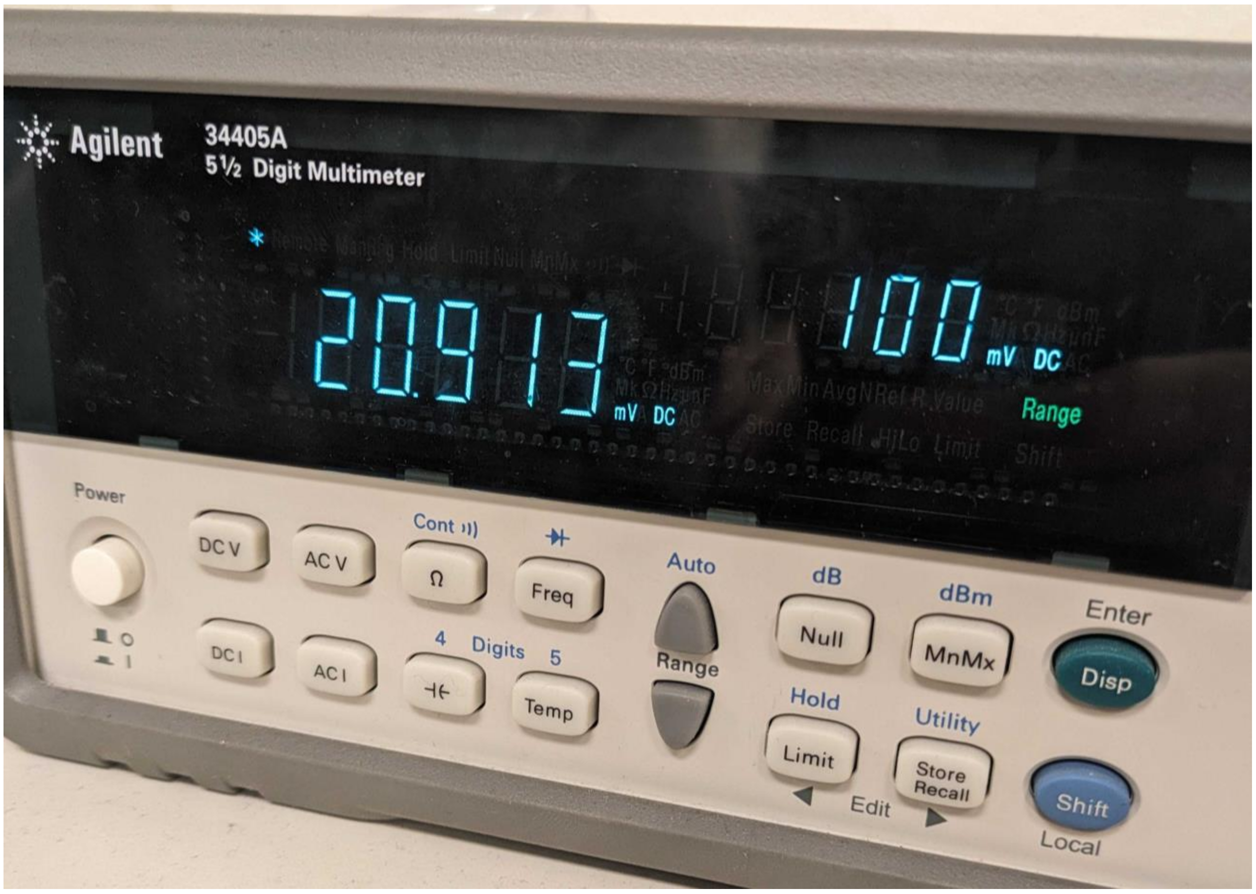


Figure 4 : Voltage measure across 100mOhm sense resistor at output of charger indicating a charge current of 210 mA.

E-Ink Display – Waveshare 4.2 Inch BW Display Module

This was the central component of my project. There were several reasons for choosing this model.

1. I knew WaveShare to be a trusted brand amongst hobbyists.
2. WaveShare had refences and examples which I could refer to if I got stuck.
3. I was easily able to order a module off Amazon without having to worry about shipping logistics and delivery times.

How E-Ink Displays Work.

- The fundamental principle behind E-Ink Displays is the use of micro-capsules filled with charged white and black particles.
- These microcapsules are embedded in a thin film and situated between two electrodes. When an electric field is applied, the particles move either to the top or bottom of the microcapsules, causing the display to appear either black or white.

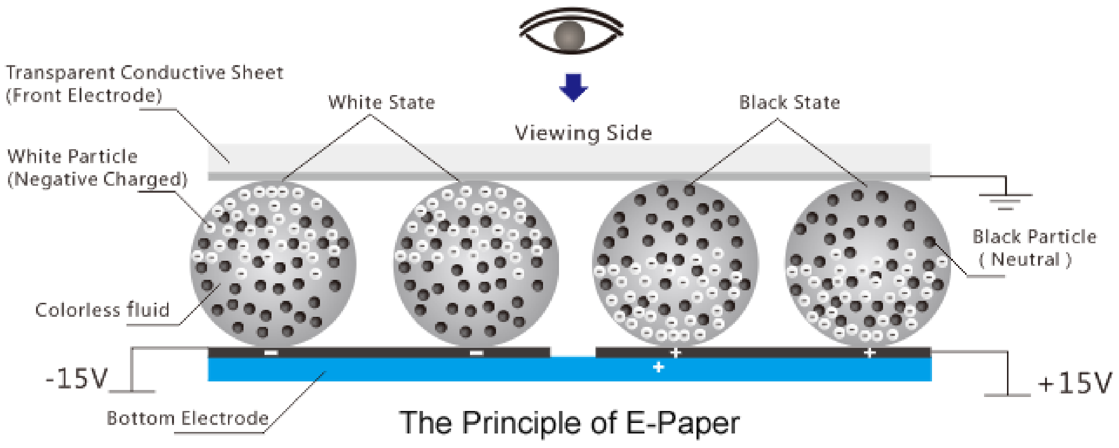

Transparent Conductive Sheet (Front Electrode)
White Particle (Negative Charged)
Colorless fluid
White State
Viewing Side
Black State
Black Particle (Neutral)
-15V
+15V
Bottom Electrode
The Principle of E-Paper

Figure 5 : Strucure of E-Ink Displays *Image Courtesy : www.e-paper-display.com*

- However, if a microcapsule is left in a state particular state of charge for too long the parts of the display (the walls of the microcapsules near the top and the bottom edges) might acquire charge themselves.
- To avoid this whenever E-Ink Displays are refreshed they are supposed to go through a sequence of voltage transitions. These voltage sequence for each color transition is stored in a One-Time-Programmable memory in the form of Look-Up Tables. Applied Science does a great job of explaining this(starting at the 5 min mark).

The E-Ink Module interfaces with the microcontroller using an SPI interface and follows a "command-data" format for data transfer.

Partial Refresh and Ghosting

It is possible to skip the entire voltage sequence of an E-Ink display which takes about 4 seconds and do a partial refresh instead. A partial refresh follows a different set of waveforms than a full refresh and provides a refresh time of about a second. However, to avoid permanent damage to the display a full refresh must be performed after a few partial refreshes. Partially refreshing the display also gives rise to ghosting on the panel wherein the previous image isn't cleared completely. This problem worsens with every partial refresh and the only way to "reset" the display is to do a full refresh.

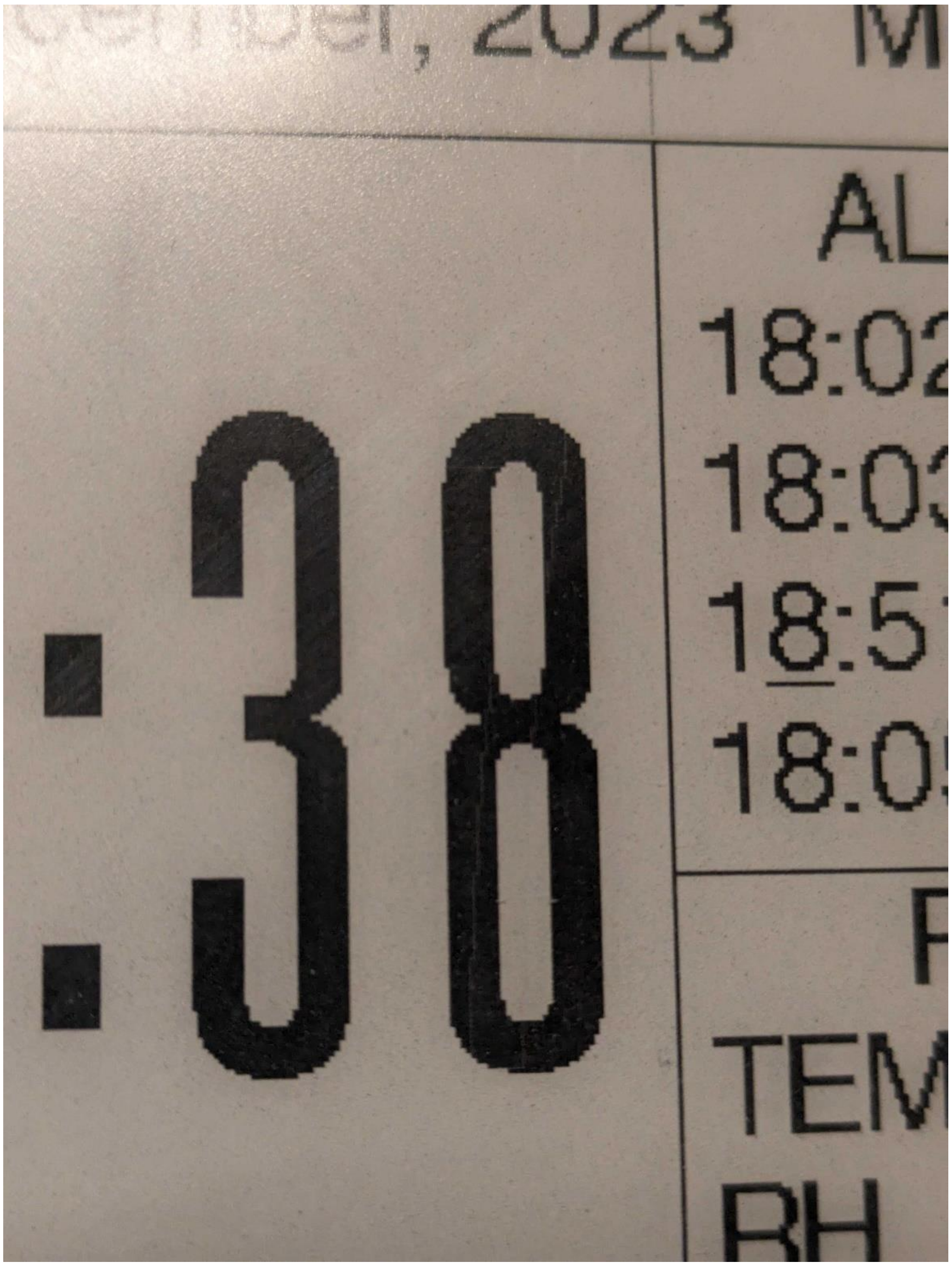


Figure 6 : Example of ghosting after parital refresh visible on number 8. Observe the faint image of 7.

PCB Design and Assembly

- To make the hardware compact and efficient I designed a PCB integrating the sensors, battery charger, buttons and the LDO.
- This board could be powered using a standard USB C cable.
- I had to add some fallback options to the board. As I wasn't fully confident about the battery charger working correctly, I added an option to bypass the battery and directly power the LDO from the 5V supply.

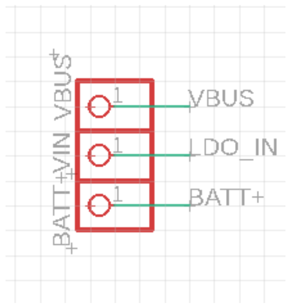


Figure 7 : Jumper to select between VBUS and VBATT+ for LDO Input

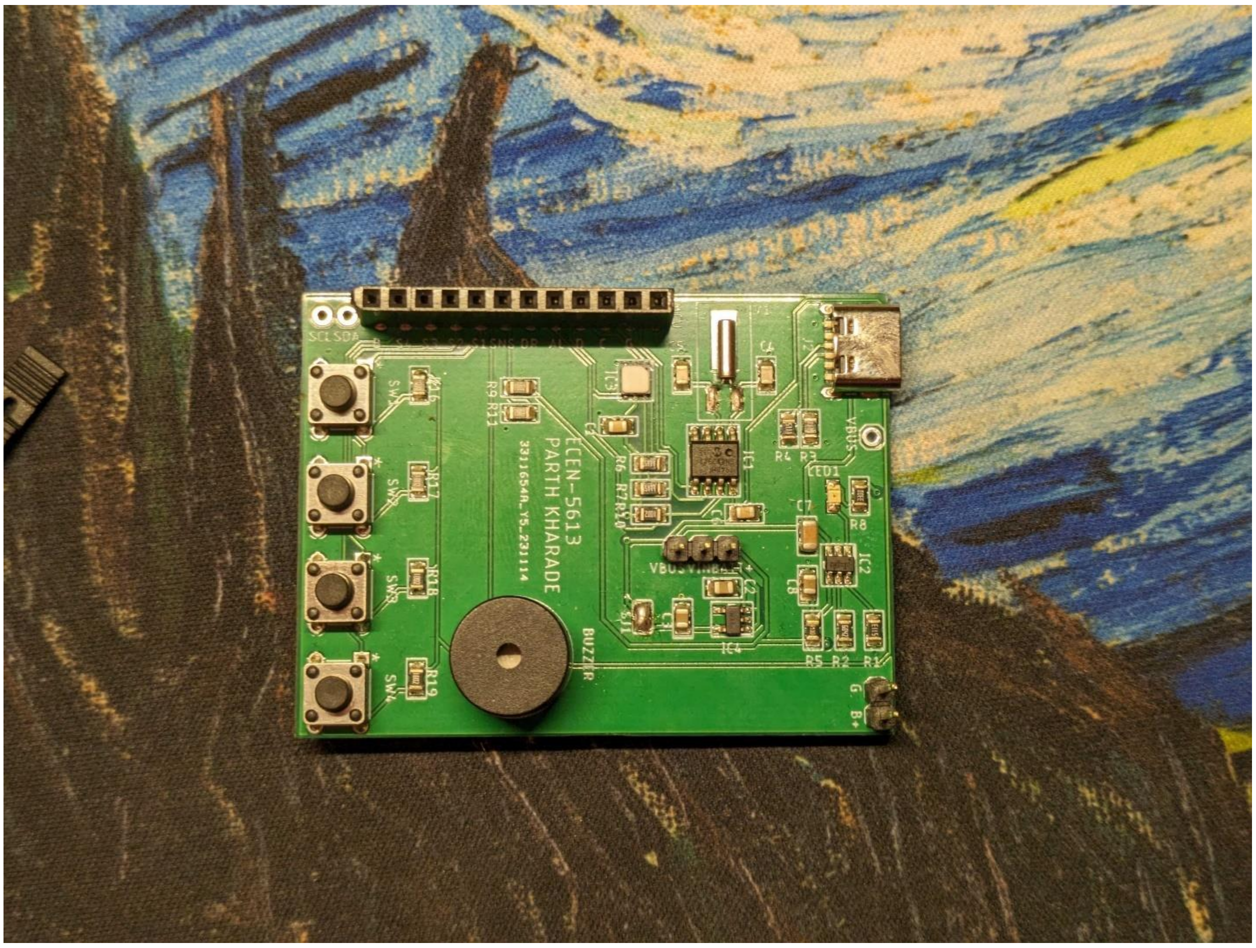


Figure 8 : Assembled PCB TOP

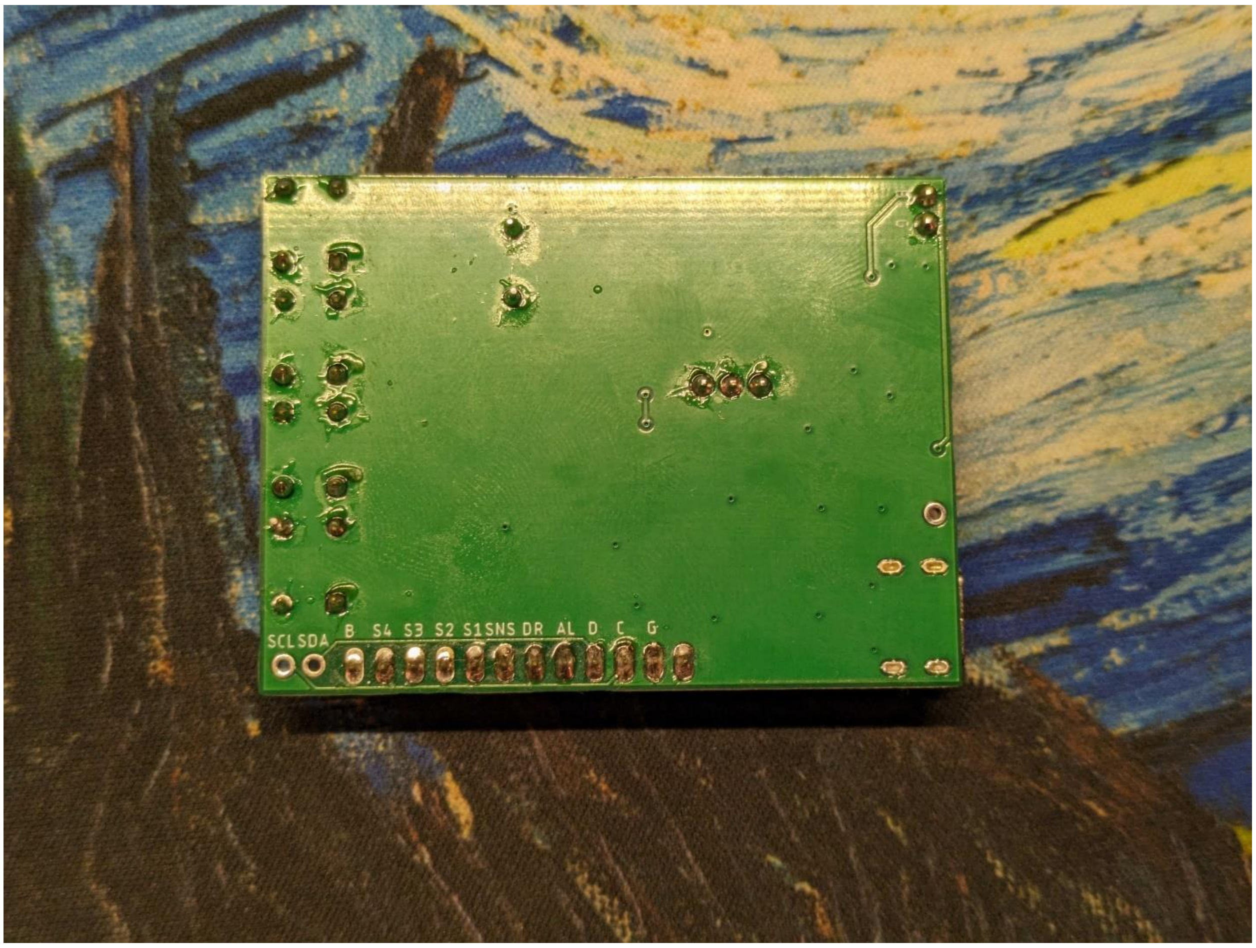


Figure 9 : Assembled PCB Bottom

Software Components

The software stack can be broadly classified into 4 layers. The lower layers expose API's which are used by the upper layers to implement their functions.

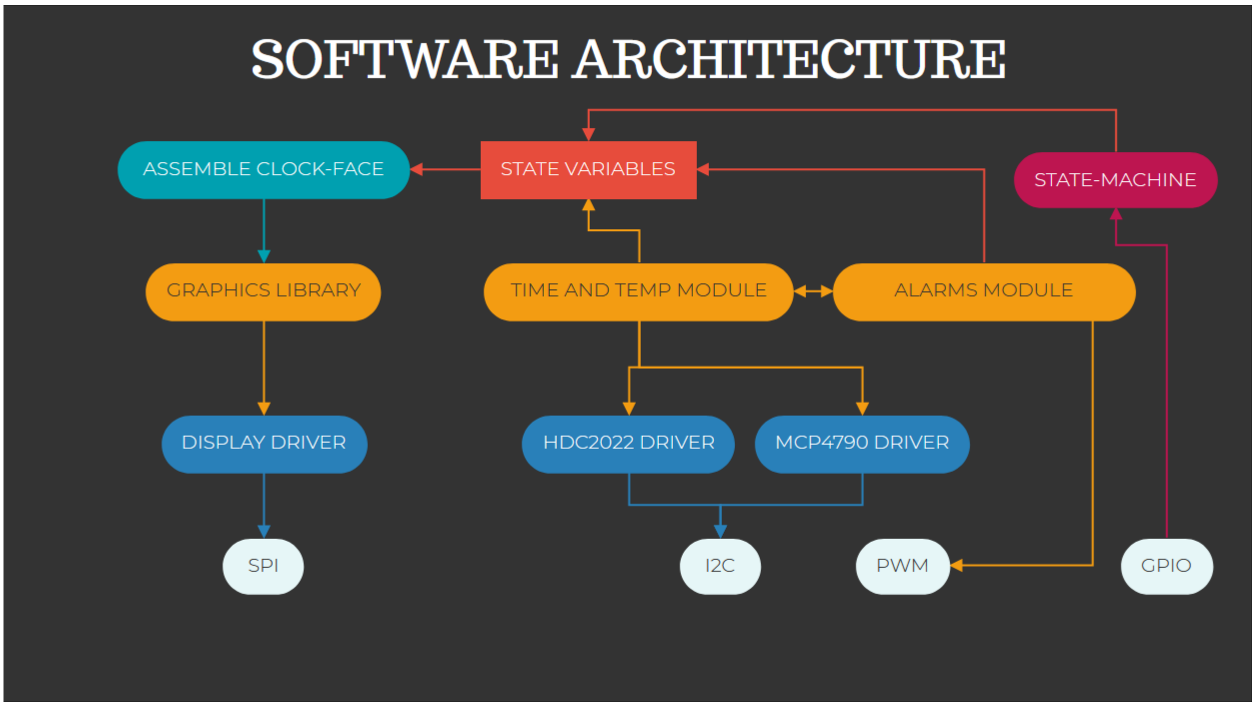


Figure 10 : Software Architecture

Bare-metal drivers

These involve the functions to initialize the microcontroller peripherals and provide basic APIs to leverage their functionality. For example, the I2C and SPI drivers provide functions to send and receive bytes on the I2C and SPI buses. The GPIO library provides a function to get the status of the 4 GPIO pins connected to the user buttons. The timer library provides a function to start and stop the timers. The timer library is used by the alarm module to sound the buzzer in beeps at intervals of 1s. The files concerning this module are named as *peripheral.h/peripheral.c*

Sensor Libraries and Display Library

This layer includes the code for reading and writing data to the sensors. It also provides an abstraction to meaningfully interpret the data read from the sensor.

For the display this layer contains functions for initializing the display, sending command and data, and reading the display busy status. The files concerning this module are named as *device.h/device.c.*

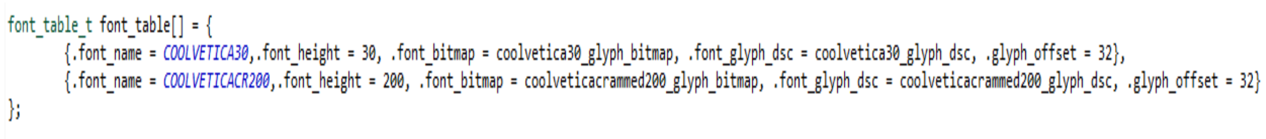
The code to initialize the display and the lookup tables for partial updating mode was taken from here.

Specifically, the **EPD_4IN2_Partial_SetLut(), EPD_4IN2_Init_Partial() and the EPD_4IN2_Init_Fast()** functions.

Graphics Library

The graphics library was the most challenging part of the project for me as this was the first time I had worked on a graphic display. The GxEPD2 and the WaveShare libraries proved to be an incredibly helpful resource in understanding the paradigm of writing display drivers. I got the idea of using a framebuffer as an abstraction between the user and the actual display-ram from here. This means that whenever the user calls a “draw” function the manipulations are made to a frame buffer which is stored in the microcontrollers RAM. An “update_display()” can then transfer this frame buffer or a part of this frame buffer to the display as required. This minimizes the number of writes and refreshes required from the display.

I learnt that graphic drivers often manage “assets” which is basically a collection of bitmaps of objects that it must render on the screen. These can be fonts, icons, or entire images. When writing a graphics library, one also must come up with a system to reference and manage these assets. I used a table of structures to reference a particular asset. This is what the table looks like for fonts



```
font_table_t font_table[] = {
        {.font_name = COOLVETICA30,.font_height = 30, .font_bitmap = coolvetica30_glyph_bitmap, .font_glyph_dsc = coolvetica30_glyph_dsc, .glyph_offset = 32},
        {.font_name = COOLVETICACR200,.font_height = 200, .font_bitmap = coolveticacrammed200_glyph_bitmap, .font_glyph_dsc = coolveticacrammed200_glyph_dsc, .glyph_offset = 32}
};
```

Figure 11 : Font table entry examples.

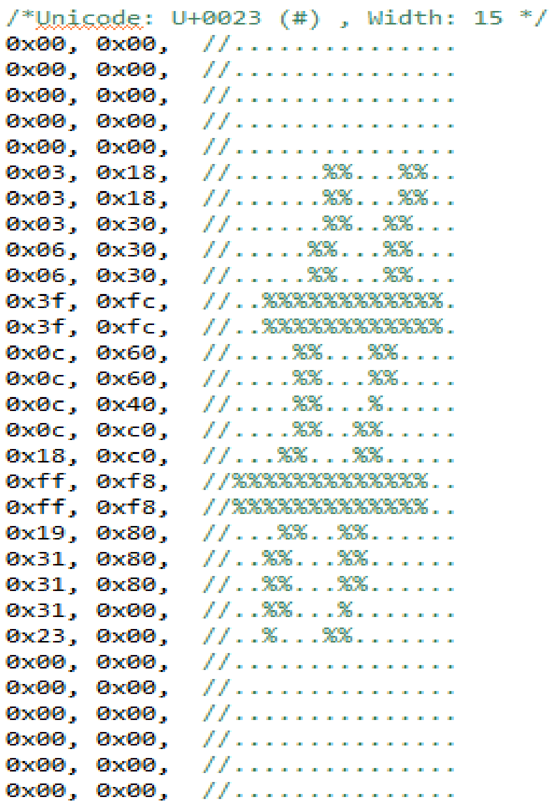
Whenever the code refers to a particular font by its font name, the corresponding glyph (an individual character/symbol in a font is referred to as a glyph)table for that font can be dereferenced from this table.

As evident, there are two important arrays associated with a font: the font bitmap and the font glyph description. The font bitmap is a single dimensional array of unsigned 8 bits numbers containing the bitmap of the font. The glyph description table is an index for the font, it has the array index offsets and widths for each glyph in the font. Using an entry in the glyph description table we can exactly dereference a particular character of a font from its bitmap array. An example of a bitmap and its description table will further clarify what I exactly mean.

To generate bitmaps of fonts I first downloaded TTF files of free opensource fonts from dafont. Then I used Light and Versatile Graphic Library’s(LVGL) online tool to convert my TTF files into C array styled bitmaps. This generated both the bitmap table and the glyph description table. The LVGL also has an online tool to convert .bmp images into C arrays which is incredibly useful for generating objects like icons.

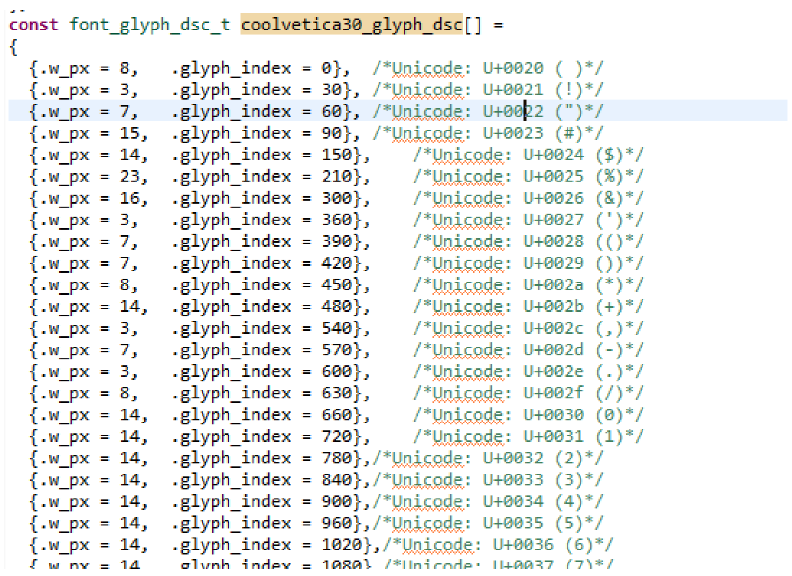
I have organized all these into an “assets” folder in my source code. While there are only 2 fonts right now and only a single alarm bell icon, I do plan to add more to this later. Because the system is highly modular not a lot of work has to be done when adding a new font.

1. Add the bitmap and the description table in the assets folder as a part of a .c files.
2. Declare these variables as externs in the fonts.h file.
3. Add an entry to the font_table in the fonts.c file and a corresponding entry with the font name to the enum in the fonts.h file. Make sure that the enum index and the index for the entry in the font table matches.



```
/*Unicode: U+0023 (#) , Width: 15 */
0x00, 0x00,  //...............
0x00, 0x00,  //...............
0x00, 0x00,  //...............
0x00, 0x00,  //...............
0x00, 0x00,  //...............
0x03, 0x18,  //......%%...%%..
0x03, 0x18,  //......%%...%%..
0x03, 0x30,  //......%%..%%...
0x06, 0x30,  //.....%%...%%...
0x06, 0x30,  //.....%%...%%...
0x3f, 0xfc,  //..%%%%%%%%%%%%.
0x3f, 0xfc,  //..%%%%%%%%%%%%.
0x0c, 0x60,  //....%%...%%....
0x0c, 0x60,  //....%%...%%....
0x0c, 0x40,  //....%%...%.....
0x0c, 0xc0,  //....%%..%%.....
0x18, 0xc0,  //...%%...%%.....
0xff, 0xf8,  //%%%%%%%%%%%%%..
0xff, 0xf8,  //%%%%%%%%%%%%%..
0x19, 0x80,  //...%%..%%......
0x31, 0x80,  //..%%...%%......
0x31, 0x80,  //..%%...%%......
0x31, 0x00,  //..%%...%.......
0x23, 0x00,  //..%...%%.......
0x00, 0x00,  //...............
0x00, 0x00,  //...............
0x00, 0x00,  //...............
0x00, 0x00,  //...............
0x00, 0x00,  //...............
0x00, 0x00,  //...............
```

Figure 12 : An example of bitmap entry for the "#" character.



```
const font_glyph_dsc_t coolvetica30_glyph_dsc[] =
{
  {.w_px = 8,   .glyph_index = 0},  /*Unicode: U+0020 ( )*/
  {.w_px = 3,   .glyph_index = 30}, /*Unicode: U+0021 (!)*/
  {.w_px = 7,   .glyph_index = 60}, /*Unicode: U+0022 (")*/
  {.w_px = 15,  .glyph_index = 90}, /*Unicode: U+0023 (#)*/
  {.w_px = 14,  .glyph_index = 150},    /*Unicode: U+0024 ($)*/
  {.w_px = 23,  .glyph_index = 210},    /*Unicode: U+0025 (%)*/
  {.w_px = 16,  .glyph_index = 300},    /*Unicode: U+0026 (&)*/
  {.w_px = 3,   .glyph_index = 360},    /*Unicode: U+0027 (')*/
  {.w_px = 7,   .glyph_index = 390},    /*Unicode: U+0028 (()*/
  {.w_px = 7,   .glyph_index = 420},    /*Unicode: U+0029 ())*/
  {.w_px = 8,   .glyph_index = 450},    /*Unicode: U+002a (*)*/
  {.w_px = 14,  .glyph_index = 480},    /*Unicode: U+002b (+)*/
  {.w_px = 3,   .glyph_index = 540},    /*Unicode: U+002c (,)*/
  {.w_px = 7,   .glyph_index = 570},    /*Unicode: U+002d (-)*/
  {.w_px = 3,   .glyph_index = 600},    /*Unicode: U+002e (.)*/
  {.w_px = 8,   .glyph_index = 630},    /*Unicode: U+002f (/)*/
  {.w_px = 14,  .glyph_index = 660},    /*Unicode: U+0030 (0)*/
  {.w_px = 14,  .glyph_index = 720},    /*Unicode: U+0031 (1)*/
  {.w_px = 14,  .glyph_index = 780},/*Unicode: U+0032 (2)*/
  {.w_px = 14,  .glyph_index = 840},/*Unicode: U+0033 (3)*/
  {.w_px = 14,  .glyph_index = 900},/*Unicode: U+0034 (4)*/
  {.w_px = 14,  .glyph_index = 960},/*Unicode: U+0035 (5)*/
  {.w_px = 14,  .glyph_index = 1020},/*Unicode: U+0036 (6)*/
  {.w_px = 14,  .glyph_index = 1080},/*Unicode: U+0037 (7)*/
```

Figure 13 : Glyph Description table showing glyph widths and offsets.

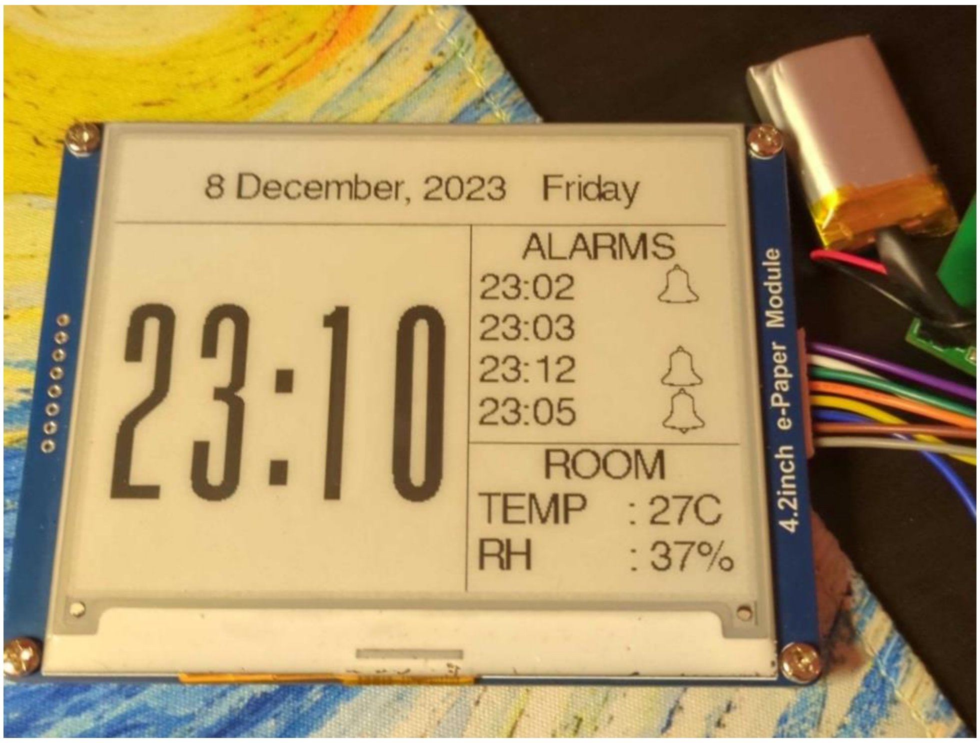


Figure 14 : Final Clockface

State machine

The actions corresponding to each button press change depending on the state/mode the system currently is. The display manages user interactions by internally switching between modes. In each mode the actions of the button change accordingly.

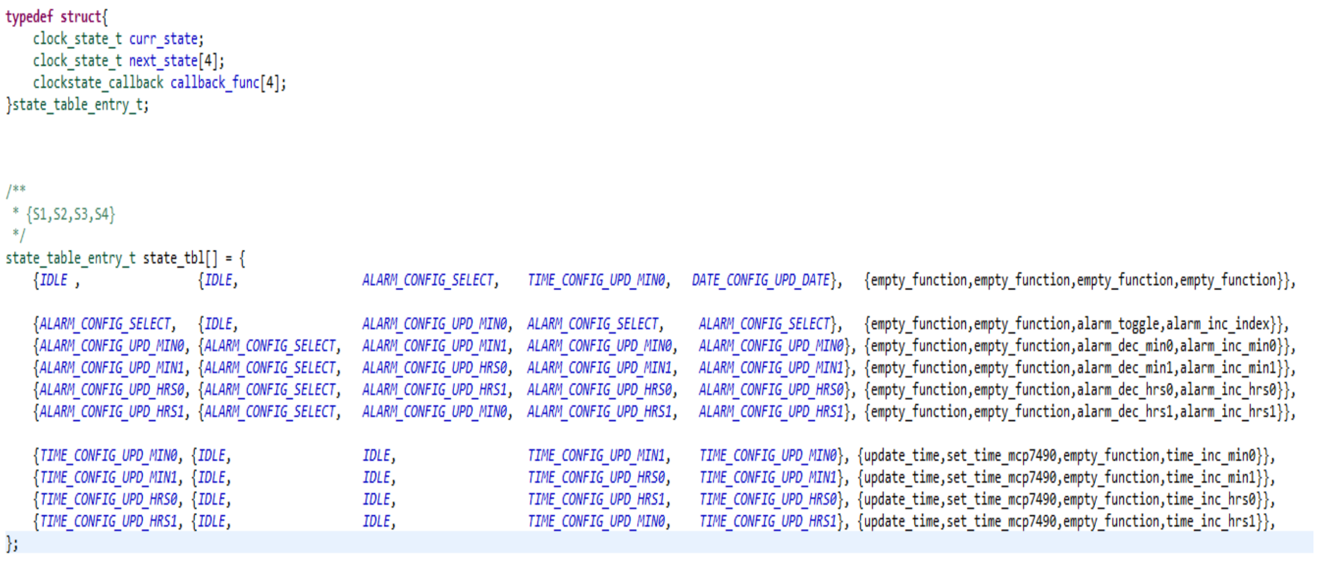
1. IDLE MODE : The user has not pressed any button yet. The display will continue to update time date and temperature.
    a. S1 : Does nothing
    b. S2 : Enters ALARM_CONFIG_SELEC T mode.
    c. S3 : Enters TIME_CONFIG mode.
    d. S4 : Does nothing.
2. ALARM_CONFIG_SELECT : In this mode the user can toggle individual alarms and select which alarms to configure.
    a. S1 : Back to IDLE MODE
    b. S2 : ALARM_CONFIG_UPD for currently selected alarm.
    c. S3 : Toggle the currently selected alarm.
    d. S4 : Cycles through the 4 alarms.
3. ALARM_CONFIG_UPD : The user can configure the time for the selected alarm.
    a. S1 : Back to ALARM_CONFIG_SELECT
    b. S2 : Cycle between the digits of the alarm to be updated.
    c. S3 : Decrement the selected digit.
    d. S4 : Increment the selected digit.
4. TIME_CONFIG_UPD
    a. S1 : Discard time and back to IDLE

b. S2 : Set time and back to IDLE.
c. S3 : Cycle through the digits.
d. S4 : Increment the selected digit.

These modes are difficult to explain solely using text. In the presentation video I have demonstrated each of these modes with the buttons and their corresponding actions. Using the state table as a reference when watching the video will be the best course of action for understanding how this works.

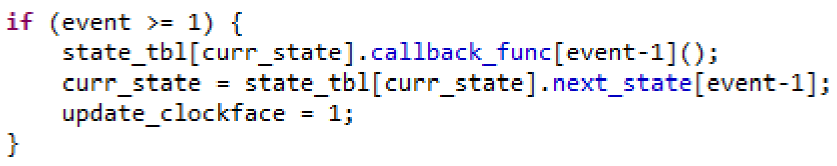
To manage all these modes, I used a "state table" based approach to design my state machine. We learnt about this approach in ECEN 5813 (Principles of Embedded Software). It essentially maintains a array of structs where each entry corresponds to a state. The entry then contains callback functions and the next state to transition to for each input.

This makes adding new states and managing state machines with a large number of states incredibly easier. All one has to do is add a new entry to the state table and define its corresponding callback functions.



```
typedef struct{
    clock_state_t curr_state;
    clock_state_t next_state[4];
    clockstate_callback callback_func[4];
}state_table_entry_t;



/**
 * {S1,S2,S3,S4}
 */
state_table_entry_t state_tbl[] = {
    {IDLE ,              {IDLE,                ALARM_CONFIG_SELECT,   TIME_CONFIG_UPD_MIN0,   DATE_CONFIG_UPD_DATE},   {empty_function,empty_function,empty_function,empty_function}},

    {ALARM_CONFIG_SELECT,   {IDLE,                ALARM_CONFIG_UPD_MIN0, ALARM_CONFIG_SELECT,    ALARM_CONFIG_SELECT},   {empty_function,empty_function,alarm_toggle,alarm_inc_index}},
    {ALARM_CONFIG_UPD_MIN0, {ALARM_CONFIG_SELECT, ALARM_CONFIG_UPD_MIN1, ALARM_CONFIG_UPD_MIN0,  ALARM_CONFIG_UPD_MIN0}, {empty_function,empty_function,alarm_dec_min0,alarm_inc_min0}},
    {ALARM_CONFIG_UPD_MIN1, {ALARM_CONFIG_SELECT, ALARM_CONFIG_UPD_HRS0, ALARM_CONFIG_UPD_MIN1,  ALARM_CONFIG_UPD_MIN1}, {empty_function,empty_function,alarm_dec_min1,alarm_inc_min1}},
    {ALARM_CONFIG_UPD_HRS0, {ALARM_CONFIG_SELECT, ALARM_CONFIG_UPD_HRS1, ALARM_CONFIG_UPD_HRS0,  ALARM_CONFIG_UPD_HRS0}, {empty_function,empty_function,alarm_dec_hrs0,alarm_inc_hrs0}},
    {ALARM_CONFIG_UPD_HRS1, {ALARM_CONFIG_SELECT, ALARM_CONFIG_UPD_MIN0, ALARM_CONFIG_UPD_HRS1,  ALARM_CONFIG_UPD_HRS1}, {empty_function,empty_function,alarm_dec_hrs1,alarm_inc_hrs1}},

    {TIME_CONFIG_UPD_MIN0, {IDLE,               IDLE,                  TIME_CONFIG_UPD_MIN1,    TIME_CONFIG_UPD_MIN0}, {update_time,set_time_mcp7490,empty_function,time_inc_min0}},
    {TIME_CONFIG_UPD_MIN1, {IDLE,               IDLE,                  TIME_CONFIG_UPD_HRS0,    TIME_CONFIG_UPD_MIN1}, {update_time,set_time_mcp7490,empty_function,time_inc_min1}},
    {TIME_CONFIG_UPD_HRS0, {IDLE,               IDLE,                  TIME_CONFIG_UPD_HRS1,    TIME_CONFIG_UPD_HRS0}, {update_time,set_time_mcp7490,empty_function,time_inc_hrs0}},
    {TIME_CONFIG_UPD_HRS1, {IDLE,               IDLE,                  TIME_CONFIG_UPD_MIN0,    TIME_CONFIG_UPD_HRS1}, {update_time,set_time_mcp7490,empty_function,time_inc_hrs1}},
};
```

Figure 15 : State table



```
if (event >= 1) {
    state_tbl[curr_state].callback_func[event-1]();
    curr_state = state_tbl[curr_state].next_state[event-1];
    update_clockface = 1;
}
```

Figure 16 : Function to handle state changes.

Corresponding code to change state and call callback functions based on user inputs.

The Model-View-Controller Pattern

The final implementation of the clock resembles a model-view-controller implementation. The switch statuses, current time, temperature, and alarms all form the state-variables or the system model which are manipulated by the system(reading sensor values , alarm match) or the user (button presses) which is then reflected on the display and on the buzzer. Given below is a representation of the software architecture with the state variables included.

Alarms and PWM

A slightly unexplained link appears on the software architecture between the alarm and the PWM module. The piezo buzzer needs a PWM to generate sound. The input to the buzzer looks like an amplitude shift keyed PWM signal.

Results and Target Overview

At the time of submitting the projects I was able to achieve the following goals.

1. A modular graphics library with the ability to easily add new fonts and functions to render vertical and horizontal lines.
2. A desk clock with 4 alarms.
3. A user interface to configure and toggle each of the 4 individual alarms. (Bonus Goal)
4. A user interface for configuring the time of the clock. (Bonus Goal)

Proposed Objectives which I was not able to achieve.

1. IoT based device with live weather updates.

   This would have required the use of a Wi-Fi module like an ESP-32 in the design. I initially tried using the ESP-IDF(IoT Development Framework) to write an SPI driver but failed. In fact, I spent a lot of time figuring out a way to install the framework on my laptop. The installation was failing because I had a space in my Windows username. This led to an invalid installation PATH for one of the python tools. This problem itself took me a couple of days to resolve.

   Later I realized that the ESP-IDF projects are based on an RTOS framework with which I had zero practical experience. I tried writing an SPI driver using the bare-metal constructs that the IDF provided but failed. At this point I was already half a week late according to my proposed timeline and decided to ditch the ESP32 in favor of the STM32.

Future Scope

Power Optimizations

Currently the temperature and the time are read continuously in the main while loop. The frequency of this can be reduced allowing the temperature sensor to sleep and update only once every minute. Also, the microcontroller can be put to sleep and woken up using a timer every 5s to read the current time. Currently the user inputs are continuously read during the while loop. This can be changed to an interrupt-based system where a button interrupt can wake up the microcontroller from deep sleep mode.

PCB Improvements

The microcontroller can be integrated into the PCB to further save on wired connections. I did not attempt to do this during the semester due to lack of time to debug if anything goes wrong while soldering the microcontroller.

IoT Based Device

The clockface can be updated to show the current weather and the maximum and minimum temperature for the day. Furthermore, using a Wi-Fi capable microcontroller the system can read time directly from a webserver and the RTC can then be used as a back-up in case the connection fails.

Improved Graphics Library

Currently, the graphics library has support to draw fonts, icons, vertical and horizontal lines. It can be further extended to draw shapes like circles, arcs, and diagonal lines(Bresenham's Line Algorithm) which can then be used to draw line graphs and pie charts.

Windowed Update Mode

Earlier I had claimed that it wasn't possible to address a specific part of the display. Well, it turns out that I was incorrect. The display indeed does have a windowed updating mode which can be used to address and update only a specific part of the display. This can be used to avoid partially updating the whole display and can help in reducing artifacts and increasing general contrast.

Acknowledgements

I would like to thank Prof. McClure for this amazing project opportunity which allowed me to challenge myself and learn something new. It was also very kind of Prof. McClure to grant me an extension on this report.
Thanks to Jordi, Amey and Aneesh who have been excellent TA's during the semester and helped out during some of the important decisions.

I would also like to thank Lauren Darling from ITLP for her help with PCB assembly. Without her support it wouldn't have been possible to get 2 PCBs assembled and working to full capacity in a matter of a few hours.

I would also like to thank my friends and family back home who have been on endless calls with me, acting as "rubber ducks" and helping me debug problems with my code.

References

- https://goodereader.com/blog/e-paper/e-ink-waveforms-are-a-closely-guarded-secret
- https://github.com/ZinggJM/GxEPD2 - GxEPD2 E-Paper Library
- https://github.com/adafruit/Adafruit-GFX-Library - The Adafruit GFX Library
- https://github.com/lmarzen/esp32-weather-epd - u/unblended_melon's project
- https://github.com/waveshareteam/e-Paper/tree/master/STM32/STM32-F103ZET6 - Waveshare Guide
- https://www.youtube.com/watch?v=MsbiO8EAsGw – A good explanation of E-Paper Displays
- https://lvgl.io/tools/imageconverter -- Images to Bitmaps
- https://www.dafont.com/theme.php?cat=302 – Free Fonts
- https://lvgl.io/tools/font_conv_v5_3 -- Fonts to Bitmaps
- https://www.iconfinder.com/ -- Free Icons