Department of Electrical and Computer Engineering University of Victoria
ECE-299

Title: ECE 299 Final Report

Report Submitted on: August 5, 2024

To: Dr. Illamparithi Thirumarai Chelvan

Names: Quinn Mudge – V01014091

Khephren Gould – V01012827

Executive Summary

The following document outlines the process followed to develop a clock radio for the University of Victoria's ECE 299 project. The device required a clock with time setting, formatting, and alarm capabilities in addition to a radio with channel reception, volume adjustment, and channel information display. The constraints under which development took place included the use of a Raspberry Pi Pico for processing, MicroPython for programming, SPI for communication with the Display, and I2C for communication with the radio module. Additional requirements included design and manufacturing of both a PCB and an enclosure with a theme of "Home Province."

We used an object-oriented approach to develop the software making individual Button, Icon, Radio, and Display classes as well as a class for each state of our device. The states included Menu state, Clock state, Radio state, Alarm state, and Play Alarm state.

Using KiCAD we designed a circuit schematic and PCB layout for the device. The alarm and radio were played on a single speaker using a diode circuit. We achieved proper radio and alarm volume using an LM386-N audio amplifier with a gain of 20. The circuit received power through the microUSB input of the Raspberry Pi Pico. The PCB layout featured a 3.3V power plane on the front copper, a ground plane on the back copper and 10, 20, and 30 mil traces connecting components. We selected a 30-mil trace width from the amplifier output to the speaker input to maximize current delivered to the speaker.

We modelled a two-piece enclosure to contain the PCB using SolidWorks. The enclosure contained mounting holes for the PCB, display, speaker, and input devices. We used #4-40 screws and nuts to secure all components. The cover was secured to the base using a wood screw and friction between the base's plastic extrusions and the wooden cover.

To improve the design, we would reduce the PCB size and match its shape with that of the enclosure. Alarm reliability was problematic during the demonstration and could be resolved by updating the triggering time when the time zone is modified. To improve the volume range of the radio, the audio amplifier could be powered with 9V instead of 5V increasing the minimum voltage at which signal distortion occurs.

Contents

List of Tables

List of Figures

I. Project goals

Design and prototype a clock radio that allows users to set and display time, manage alarms, and tune into radio channels through a user-friendly interface, utilizing a Raspberry Pi Pico for intelligence handling, and incorporating an OLED display and speaker for output.

II. List of constraints

Below are the list constraints set for this project, these constraints were taken from the course page [1].

1) Raspberry Pi Pico must be used for intelligence handling.
2) The display must communicate with the Pico board using SPI.
3) The Raspberry Pi Pico to be programmed using Python.
4) Design and order a PCB from a commercial manufacturer.
5) Final demonstration must happen in the week of July 29.

III. Project expectations

Below is the list of project expectations set for this project. These expectations were taken from the course page [1].

Clock Functionality:

1. Set/Edit time through a user interface.

2. Display time in 12-hour/24-hour format as desired by the user.

3. Set/Edit alarm time through a user interface.

4. Trigger alarm as set by the user.

5. Snooze alarm and turn off alarm through a user interface.

Radio Functionality:

1. Tune to at least one radio channel through a user interface.

2. Play radio output on a speaker.

3. Volume control of audio output through a user interface.

4. Display radio channel info to the user.

IV. Design choices

The section below describes our design process and the reasoning behind the design choices made.

Hardware

In general, cost, reliability, and ease of use were considered when selecting the hardware for our clock radio.

Display

To handle all the requirements related to displaying information from the clock radio system, we selected the SSD1306 OLED display. This display was chosen due to its SPI compatibility, power requirements, image quality, mounting requirements. An image of the display is shown in **Figure** 1.

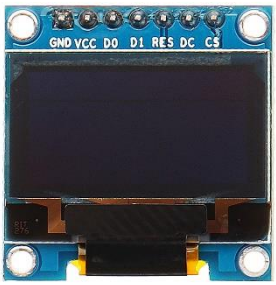

Figure 1: SSD1306 Display

Based on the project constraints, the display was required to communicate with the Raspberry Pi using the SPI communication protocol. As reflected in the SSD1306 datasheet, the internal MCU of the display supports 4-wire SPI communication [2]. Communication is done through the serial clock (SCK), serial data in (SDA), chip select (CS), and the data/command (DC) lines of the display.

The power requirements of the SSD1306 are satisfied by the 3.3V output voltage of the Raspberry Pi. According to the SSD1306 datasheet, the allowable supply voltage is between 1.65V and 3.3V [2]. This allowed us to power the display directly from the 3.3V plane on our PCB reducing the PCB's complexity.

Luo and Wu suggest that LCD displays are good alternatives to OLED displays due to LCDs having higher resolution and longer lifetime. We selected an OLED display since, according to Luo, they are thinner, have excellent contrast, and a wide viewing angle [3]. Since the

screen is small, contrast and viewing angle are highly important for viewing the information displayed effectively.

Another benefit of the SSD1306 is its simplistic and cheap mounting method. Based on the hole pattern found in the datasheet, the mounting holes are 3mm in diameter. Thus, it can be mounted with four #4-40 screws and nuts.

Finally, we selected the SSD1306 over the SSD1309 due to its far lower cost. Despite being smaller, the SSD1306 features the same size dot matrix (128x64) as the SSD1309 and matches the quantity of information it can display at once. Thus, in selecting the SSD1306 we lowered the cost of our final design by $16 [4] without sacrificing the quality of our UI.

Radio Module

The RDA5807M was selected to fulfill the device's radio functionality. The attributes possessed by this receiver that motivated our decision to use it include the frequency range it supports, the communication protocol it uses, its power requirements, volume and frequency control features, physical package characteristics, and cost. An image of the RDA5807M breakout board is shown in **Figure** 2

Figure 2: RDA5807M Breakout Board

According to the RDA5807M datasheet, it can decode FM (frequency modulated) frequencies in the range of 50MHz to 115 MHz [5]. As many local radio stations broadcast in this range [6] the module's frequency range permitted the clock radio to receive multiple channels exceeding the project expectations.

The radio module supports communication with an MCU through the I2C protocol. As the Raspberry Pi features two dedicated I2C controllers [7], the microcontroller interfaced with

the radio module directly through software. The speed of I2C communication and resilience to data loss referred to by Mankar were beneficial when implementing the radio functionality [8]. Leveraging the rapid response time of I2C communication, we were able to implement reliable real time volume and frequency adjustment without the need for an "enter" command from the user.

The RDA5807M features digital audio gain control (volume adjustment) and autonomous search tuning [5]. These features were leveraged when implementing the volume control and seek functionality of our device.

Additionally, the package of the RDA5807M was advantageous since it compatible with breadboard testing and through hole soldering. This allowed the radio functionality to be tested on the breadboard without the need to solder header pins saving both cost and time. The TEA5767 from NXP Semiconductors was considered as an alternative due to its low cost ($2.40) however its package requires externally soldered pins [9].

Finally, the low cost of the RDA5807M module makes it a better option than many similar products. For example, alternatives such as the SI4703 [10], and BK1086 [11] have similar features but are more expensive. A breakout board containing the SI4703 comes at a cost of $24.95 [12].

Audio Amplifier

The LM386 audio amplifier was chosen to amplify the radio output and alarm signals in our radio clock design. This decision was driven by several key factors, including the LM386's suitable technical specifications, performance characteristics, cost-effectiveness, and reliability.

According to the LM386's datasheet [13], it is an 8-pin power amplifier designed for low voltages operations. The LM386 has a default gain of 20, which can be increased to any value from 20 to 200 by the addition of an external resistor and capacitor between pins 1 and 8. This flexibility allows for easy customization to suit amplifying the radio output and alarm signals for the clock radio design. A pin configuration of the LM386 is shown in the **Figure 3** below:

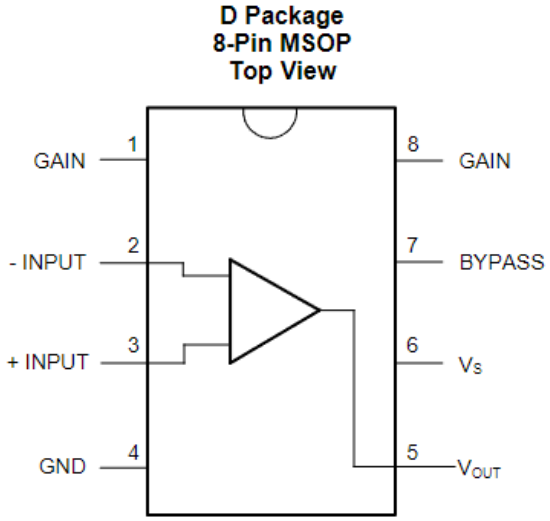

Figure 3: LM386 IC Top View

Figure 4 shown below depicts the bandwidth of the LM386 [13]. For the 20-gain circuit used in the radio clock, the bandwidth exceeds the audible range of humans (20Hz – 20kHz) [14]. This bandwidth is adequate for the purposes of alarm and radio output to the speaker, ensuring that the sound produced is clear and covers the full spectrum of audible frequencies.

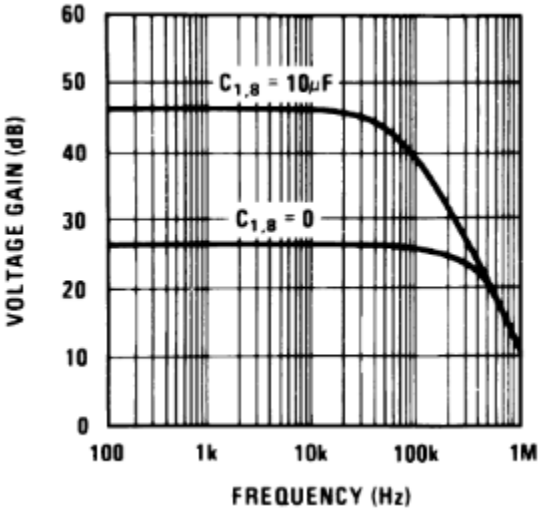

Figure 4: Voltage Gain vs Frequency

Pushbutton

From a hardware perspective, the qualities considered when selecting components to receive user input consisted of the button's dimensions, cost, mounting requirements and reliability. Push buttons were considered since they are low-cost and can be made reliable

through simple software debouncing. The R13-24A was selected since it satisfied all the desired qualities while being accessible through the UVic Tech Shop. The button has an outer diameter of 8mm and threads that extend 6mm [15]. 6mm threads allowed our enclosure to maintain a thickness of above 4mm, conserving durability, while leaving thread space for the nut.

Encoder

The PEC11R Series – 12mm Incremental Encoder was selected to take inputs from the user. The primary function of the PEC11R encoder is to convert the rotational position of its shaft into an electrical signal that the microcontroller can interpret.

The PEC11R has both a pushbutton and rotary encoder. The pushbutton can be debounced using a simple debouncing algorithm to ensure that only intentional presses are registered. The rotary encoder has a total of three pins. The first two, Pin A and B, output two quadrature signals that the microcontroller reads to determine the direction and amount of rotation [16]. The third pin is the common ground pin.

A potentiometer could have been selected over an encoder. The potentiometer's main benefit is its ease of use. The Pico can easily detect rotation through one of its ADC input pins. The encoder's main benefit over the potentiometer is continuous rotation detection [17]. For tuning the radio, the user requires access to a range of frequencies too large to be implemented using a potentiometer. To access the entire range the sensitivity of the potentiometer would not be practical. The same problem exists with setting the time. With an encoder, the user was given access to a wide range of frequencies and could set time easily.

Software

We developed the software using an object-oriented style approach with interrupts to detect use input. GitHub provided version control.

Program Structure

All the requirements for the clock radio required software to be written. For our clock radio, we used an object-oriented style of programming due to its readability, ability to efficiently develop software as a team, and scalability. **Figure 5** shows the class hierarchy (parent, child relationships) used to implement the clock radio's functionality.

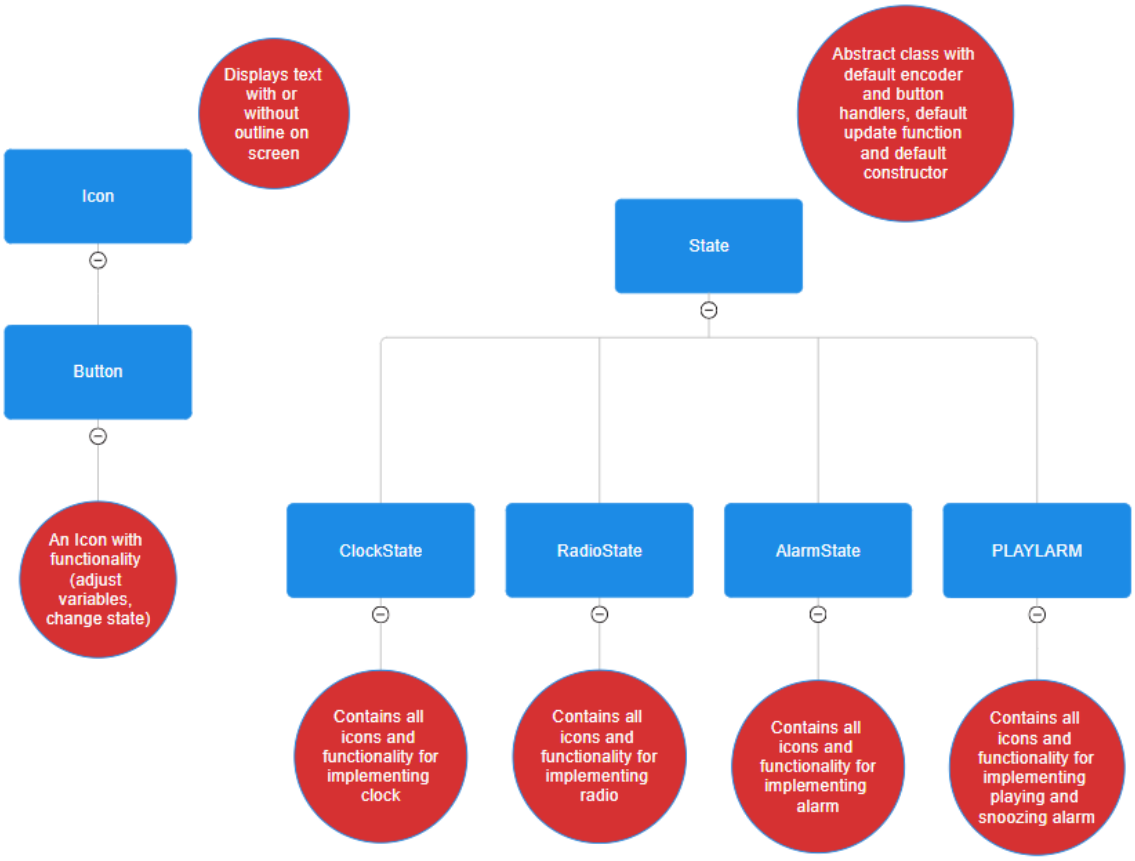

Figure 5: Clock Radio Class Hierarchy

In addition to understanding the hierarchical relationship between classes, understanding the interdependencies between the classes is also important. **Figure 6** show's the dependencies between the classes in our program.

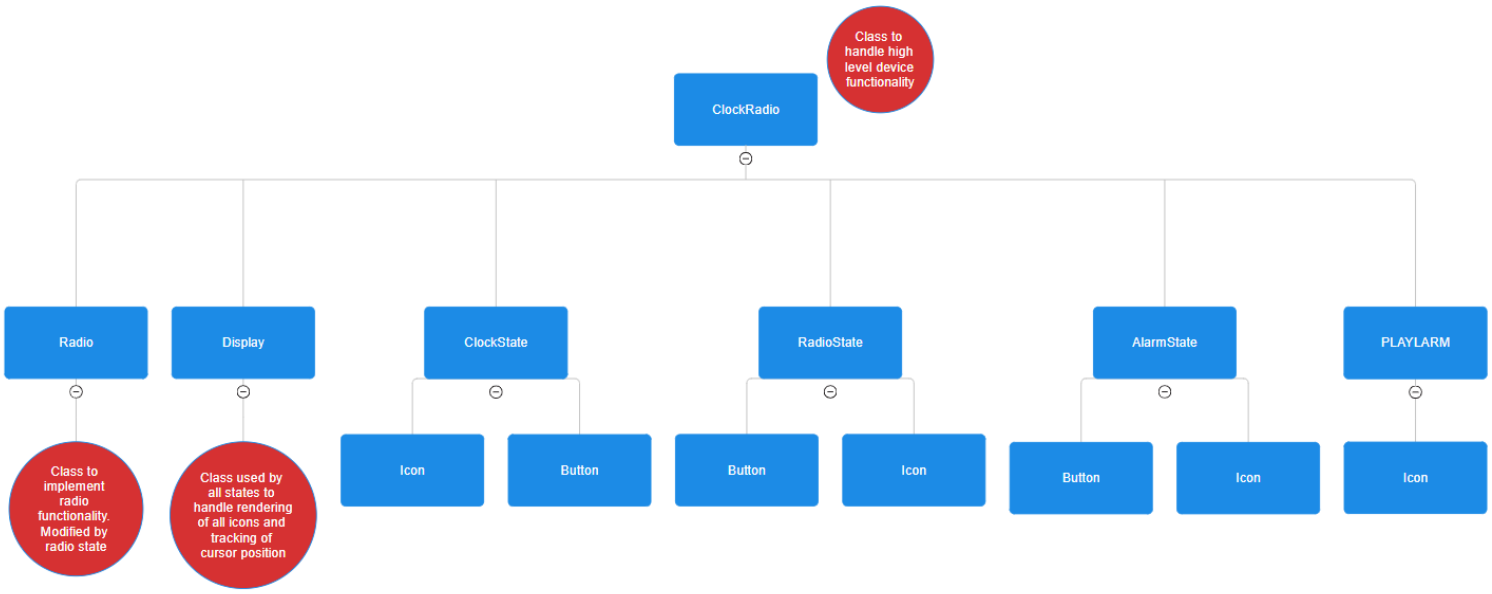

Figure 6: Class Interdependencies

Note that "button" refers to a software defined button appearing on the display, not a physical button. With an understanding of the program's structure, the basic functionality such as control flow and I/O can be understood. The code for the Icon and Button classes is included in **Figure 7** below for convenient reference.

```
46    class Icon:
47        def __init__(self, txt, pos_x, pos_y, border):
48            #text displayed by the icon
49            self.text = txt
50            #position of the icon on the screen
51            self.xpos_text = pos_x * grid_size_x + 2
52            self.ypos_text = pos_y * grid_size_y + 2
53            #width and height of the icon's border
54            self.width = grid_size_x + 2
55            self.height = grid_size_y + 2
56            #decide whether to show border (white rectangle) or not
57            self.has_border = border
58    class Button(Icon):
59        def __init__(self, txt, pos_x, pos_y, select, has_border):
60            super().__init__(txt, pos_x, pos_y, has_border)
61            self.grid_x = pos_x
62            self.grid_y = pos_y
63            #used to track if the selector's position matches the button's position
64            self.selected = select
65            #track what state change should occur when button is pressed
66            self.state = None  # Initially no state assigned
67        #setter function to set the "state" member
68        def configureState(self, the_state):
```

Figure 7: Icon and Button Classes

Taking User Input - Interrupts

The program monitors for user input using interrupts. As each state has a different movement pattern for the selector on the screen and different values to be adjusted by the encoder, each state redefines the following member functions from the "State" Base class:

- "B1Handler" (button functionality)
- "ENCA" (encoder pin A)
- "ENCB" (encoder pin B)

These three member functions act as handlers for the interrupts triggered by the button and encoder respectively. An additional interrupt is configured for the "ENTER" button (the encoder pushbutton). Interrupt configuration, a sample "B1Handler" function (in this case from PlayALARM state), and the handler to implement "ENTER" are shown in **Figures 8**, **9** and **10** below.

```
button_1.irq(handler=current_state.B1Handler, trigger=Pin.IRQ_FALLING)
EncoderA.irq(trigger=machine.Pin.IRQ_RISING | machine.Pin.IRQ_FALLING, handler=current_state.ENCA)
EncoderB.irq(trigger=machine.Pin.IRQ_RISING | machine.Pin.IRQ_FALLING, handler=current_state.ENCB)
enter.irq(handler=Enter_Handler, trigger=Pin.IRQ_FALLING)
```

Figure 8: Interrupt configuration

```
def B1Handler(self,pin):
    global current_state, current_posx, current_posy
    if debounce_handler(pin):
        if(current_posx <= 0):
            current_posy-=1
            current_posx=0
        if(current_posy==0 and current_posx<=0):
            current_posx=3
            current_posy=5
        if(current_posx!=0):
            current_posx -= 1
        self.update()
        display.render(self.icons)
```

Figure 9: Sample Clock State B1Handler

```
#interrupt handler for the encoder button
def Enter_Handler(pin):
    global ENTER, current_state
    if debounce_handler(pin):
        ENTER = True
```

Figure 10: Encoder Pushbutton handler

Control Flow

Control flow of a program refers to the pattern in which the program changes behaviour. As seen in **Figure 5**, classes with the "_State" suffix define the behaviour the program should exhibit when the user has entered a given State. An instance of each "State" class is declared in the program and a global "current_state" variable is declared to track which state the program is in. In each state, variables of the "Button" class type (seen in **Figure 7**) are declared and assigned a specific state to change to when they are pressed.

The update buttons function in the "Display" class is responsible for monitoring for state changes. Note that the program structure ensures this function runs at each iteration of the program's main While loop. The code is shown in **Figure 11** below.

```
92      def update_buttons(self, icons):
93          global current_state, ENTER, current_posx, current_posy
94          #loop over the list of icons being rendered
95          for icon in icons:
96              #only buttons have functionality and must be updated
97              if isinstance(icon, Button):
98                  #if the button position matches the selector position set "selected"
99                  #boolean True
100                 if icon.grid_x == current_posx and icon.grid_y == current_posy:
101                     icon.selected = True
102                    # if the encoder button has been pressed and the
103                     # selected button has a state assigned, change states
104                     if ENTER and icon.state:
105                         current_state = icon.state
106                         #current_state.update()
107                         change_state(current_state)
108                     #reset enter variable (to wait for next encoder button press)
109                     ENTER = False
110                 else:
111                     icon.selected = False
```

Figure 11: Update Buttons Function

As seen on lines 100 to 107 in **Figure 11**, if the encoder pushbutton("Enter") has been pressed while a button with a state configured is selected, the current state is changed to the button's "state" member and the "change_state" function (shown in **Figure 12** below) is called with this state as an argument.

```
def change_state(state):
    global current_state, current_posx, current_posy
    current_state = state
    current_posx = current_state.start_posx
    current_posy = current_state.start_posy
    EncoderA.irq(trigger=machine.Pin.IRQ_RISING | machine.Pin.IRQ_FALLING, handler=current_state.ENCA)
    EncoderB.irq(trigger=machine.Pin.IRQ_RISING | machine.Pin.IRQ_FALLING, handler=current_state.ENCB)
    button_1.irq(handler=current_state.B1Handler, trigger=Pin.IRQ_FALLING)
    current_state.update()
    display.render(current_state.icons)
```

Figure 12: Change State Function

This function changes the value of "current_state", ensures the interrupt handlers for all input devices are reassigned to those of the new current state and ensures the selector's position on the screen is appropriate for the new current state (the position defined by the state's "start_posx and start_posy" members).

Displaying Output

The display class handles visual output. It uses a "render" function to render a list of icons to the screen. Each state class contains an "icons" member list with all icons and buttons to be rendered when the state is entered. In this way, different information can be displayed on the screen depending on the state. The "render" function is shown in **Figure 13** below.

```
    def render(self, icons):
        global current_state
        self.SSD.fill(0)
        for icon in icons:
            if isinstance(icon, Button) and icon.selected:
                # Draw the icon text first

                self.SSD.text(icon.text, icon.xpos_text, icon.ypos_text, 1)
                # Invert the region of the selected icon
                self.invert_region(icon.xpos_text - 2, icon.ypos_text - 2, icon.width, icon.height)

            else:
                self.SSD.text(icon.text, icon.xpos_text, icon.ypos_text, 1)  # Normal text color
                if icon.has_border:
                    self.SSD.rect(icon.xpos_text - 2, icon.ypos_text - 2, icon.width, icon.height, 1)  # Normal border
        self.SSD.show()
```

Figure 13: Render Function

The function is passed the icons list from the current state. This list is iterated over and for each icon, text and bordering is rendered to the screen at a position stored internally by the icon or button itself. If the icon's position matches the selector's position, the region

surrounding the button is inverted (black turns white and vice versa) to indicate this to the user.

Encoder Logic

Many functionalities relied on the encoder's proper function. We implemented an interrupt based algorithm to detect encoder rotation and determine the rotation direction. As with the push button, interrupts were used to maximizes the speed at which our program executed and minimize delay in detecting user input. When the encoder is rotated in the clockwise direction the encoder emits a rectangular pulse from Pin A before emitting a similar pulse from pin B. When the encoder is rotated in the counterclockwise direction, the reverse occurs. **Figure 14** shows the output from probing the encoder with Pin A on channel 1 and Pin B on channel 2 of the oscilloscope.

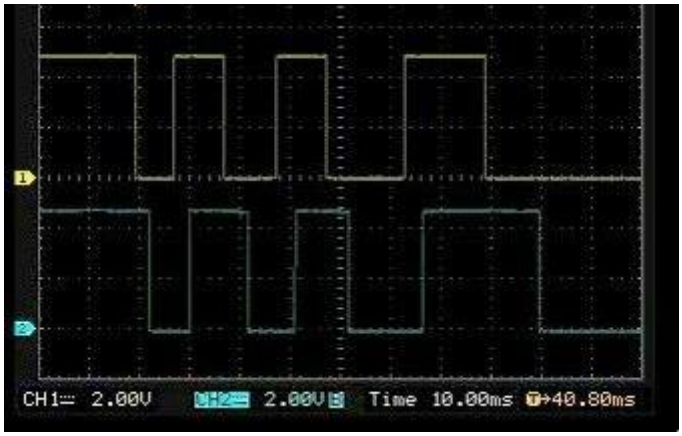

Figure 14: Capture of Clockwise Encoder Rotation

Pin A and Pin B were configured to trigger interrupts on both the rising and falling edges of their respective pulses. Separate handlers were configured for pins A and B. The handlers begin by reading the values from both encoder pins. They then perform edge detection. Due to the overlapping of pulses, the specific edge that triggered the interrupt is determined using the logic in **Figure 15** below.

```
295             # Determine edge detection on Encoder Pin B
296             if B_state == 1 and A_state == 0:
297                 B_rising_edge = True
298             elif B_state == 0 and A_state == 1:
299                 B_falling_edge = True
300
```

Figure 15: Edge Detection

Once both a rising and falling edge has been detected, we know the encoder is being rotated. To determine the direction, we compare the current values of each pin using the code in **Figure 16** below.

```
302             if B_rising_edge and B_falling_edge:
303                 year, month, day, weekday, hours, minutes, seconds, subseconds = rtc.datetime()
304                 if A_state != B_state:
305                     #clockwise
```

Figure 16: Determining Rotation Direction

If the interrupt is triggered (on what now must be a rising edge) and reading the pins results in opposite logic values, the signal resembles that of **Figure 14**. Pin A is leading Pin B, and the direction must be clockwise. If the logic values are equal, the opposite is true.

Debouncing Logic

The debouncing of pushbuttons is managed by the "debounce_handler" function as shown in **Figure 17**. This function is called from the interrupt handler functions for all the pushbuttons to ensure that the input is valid and not affected by mechanical bouncing.

```
def debounce_handler(pin):
    global last_pressed_time
    current_time = utime.ticks_ms()
    if current_time - last_pressed_time < debounce_delay:
        return False
    last_pressed_time = current_time
    return pin.value() == 0  # Check if button is pressed
```

Figure 17: Debouncing Algorithm

This algorithm operates by comparing the current time obtained using "utime.ticks_ms()", with the time of the last recorded button press "last_pressed_time". If the time elapsed since the last press is shorter than a predefined debounce delay (50ms), the function determined that the current press is due to bouncing and returns "False". Otherwise, if the time is greater than 50ms, the function stores the current time and checks the state of the button pin. If the pin's value is 0, the button is pressed, and the function returns true.

Clock Functionality

To implement the clock functionality, we utilized the RTC (Real-Time Clock) of the Raspberry Pi Pico, which ensures accurate and reliable timekeeping. This choice provides

consistency and minimizes errors in time display and alarm functionality. We found that writing data to the display too frequently introduced noise to the speaker output. To maintain optimal performance and avoid excessive noise, we implemented an interrupt triggered by a MicroPython “Timer” object. The interrupt updates the display with the current RTC time every minute, preventing the overuse of SPI lines while maintaining accurate time display.

Users interact with the clock through a simple interface: a pushbutton allows them to navigate through the menu, while a rotary encoder is used for adjusting parameters such as time and format. By default, the clock displays time in a 12-hour format, but the user can easily switch to 24-hour through the menu interface. Time adjustments are simple with buttons on the display for minutes and hours, which can be incremented/decremented with encoder inputs. As there is no extra input required to set the time (the time is updated with every encoder rotation) adjusting the time can be done rapidly. Time zones are managed using the UTC (Coordinated Universal Time) format, allowing users to set their local time from anywhere in the world.

For the alarm functionality, several customizable settings are provided. Users can set or edit the alarm time using the same method as adjusting the clock time. The snooze duration can be customized, with a minimum duration of one minute. Additionally, users have the option to select from three different alarm tones and five different volumes, providing variety for the alarm.

Radio Functionality

The main functionalities for the radio include volume adjustment, manual frequency adjustment, automatic frequency adjustment, and displaying relevant channel and radio data to the user.

Appendix C contains a link to the “rda5807.py” library used to communicate with the radio module. Through testing, we found that this library eliminated an issue in which the radio would output a severely distorted signal (static) for all channels until power cycled. This issue occurred when rapidly programing the radio module or selecting an even valued volume setting.

As our program relies heavily on interrupts, we suspect the issue may have been due to bus locking which can be caused if the main device is interrupted when transmitting data to a sub node [18]. The library enabled selection of any volume level in the 0 to 15 range instead of only the odd volume levels. Additionally, the library enabled very frequent programming of the radio module without causing the issue mentioned above.

Volume adjustment was implemented using the encoder as a dial. We placed an icon within the radio menu and included logic in the encoder interrupt handlers to increment or decrement the volume on the display, and program the radio with the updated volume when the user rotated the encoder dial. We ensured to limit the volume to range from 0 to 5 to prevent the user from selecting an invalid volume. A limit of 5 was selected because while still being loud, audio quality was maintained. Beyond this level distortion occurred. As discussed above, the use of the rda5807 library enabled rapid programming of the radio module which was leveraged to implement real time volume adjustment. As the user rotates the dial, the volume of the radio audibly changes (see GitHub lines 449 to 454).

Our clock radio exceeded the channel setting requirements by allowing any channel between the 88 and 108 MHz ranges to be tuned to. We implemented frequency adjustment identically to volume adjustment. An icon was added in the radio menu for frequency adjustment and when the user navigates to the icon the encoder can be rotated to increase or decrease the frequency. An additional icon was added to allow for automatic tuning. According to the RDA5807 datasheet, bits 8 and 9 of register 0x2 are responsible for initiating and terminating the module's seek operation [5]. The "rda5807.py" library contains "seek_up" and "seek_down" functions which write the appropriate 8-bit flags to register 0x2 (see GitHub lines 458 to 472).

V. Project planning

This section outlines the planning and coordination strategies used to accomplish the project goals effectively. We have included a Gantt chart that details the various steps involved in the project, as shown in **Figures 18, 19, 20**, and **21**, along with a table (Table 1) displaying the specific roles and responsibilities of each team member.

Table 1: Group Member Roles

Task	**Mainly Completed By**
Learn Git	Both
Test display	Both
Test speaker	Both
Test radio module	Both
Test the complete circuit	Khephren Gould
Circuit schematic	Quinn Mudge
PCB design	Both
Order PCB	Khephren Gould
Code radio functionality	Khephren Gould
Code clock Functionality	Quinn Mudge
Code alarm Functionality	Khephren Gould

Code display Functionality	Khephren Gould
Complete Final Program	Both
Design Enclosure SolidWorks	Khephren Gould
Order Enclosure	Khephren Gould
Assemble Enclosure	Both
Paint Enclosure	Both
Solder PCB	Quinn Mudge
Test PCB	Both
Assemble the final design	Khephren Gould
Test and troubleshooting	Both
Final Demonstration	Both

To facilitate communication and collaboration, we used Discord for regular updates and calls, and GitHub for version control. The use of these programs proved effective for sharing code, ensured that tasks were completed on schedule, and ensured everyone was aligned with the project objectives. See Appendix C for the GitHub repository link.

Gannt chart

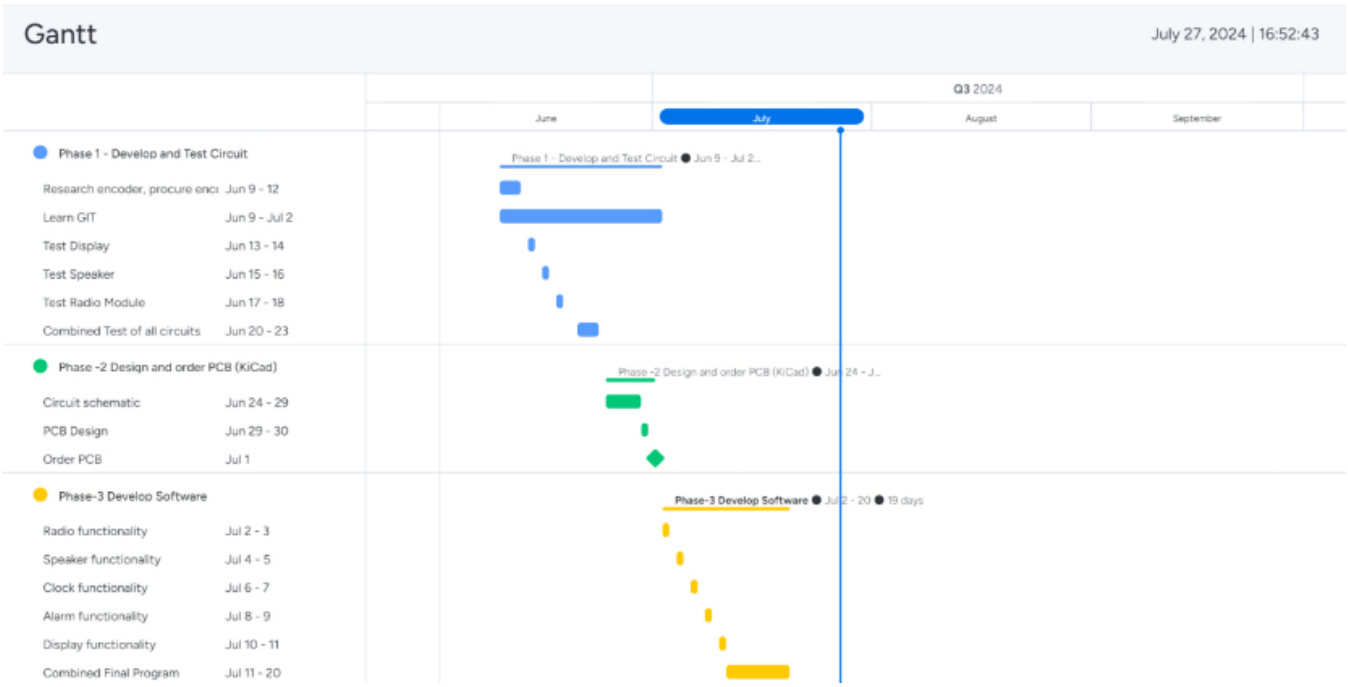

Figure 18: Gantt Chart 1

Figure 19: Gantt Chart 2

While the chart above shows the initial project plan, modifications were necessary to accommodate the schedules of both group members. **Figures 20** and **21** highlight the changes made to the original project schedule.

Figure 20: Gantt Chart 3

Figure 21: Gantt Chart 4

VI. Circuit design and schematic

Below is the circuit schematic for the clock radio project, **Figure 22**. This design integrates a variety of components to achieve the desired functionality, including audio amplification, and communication to display and radio module.

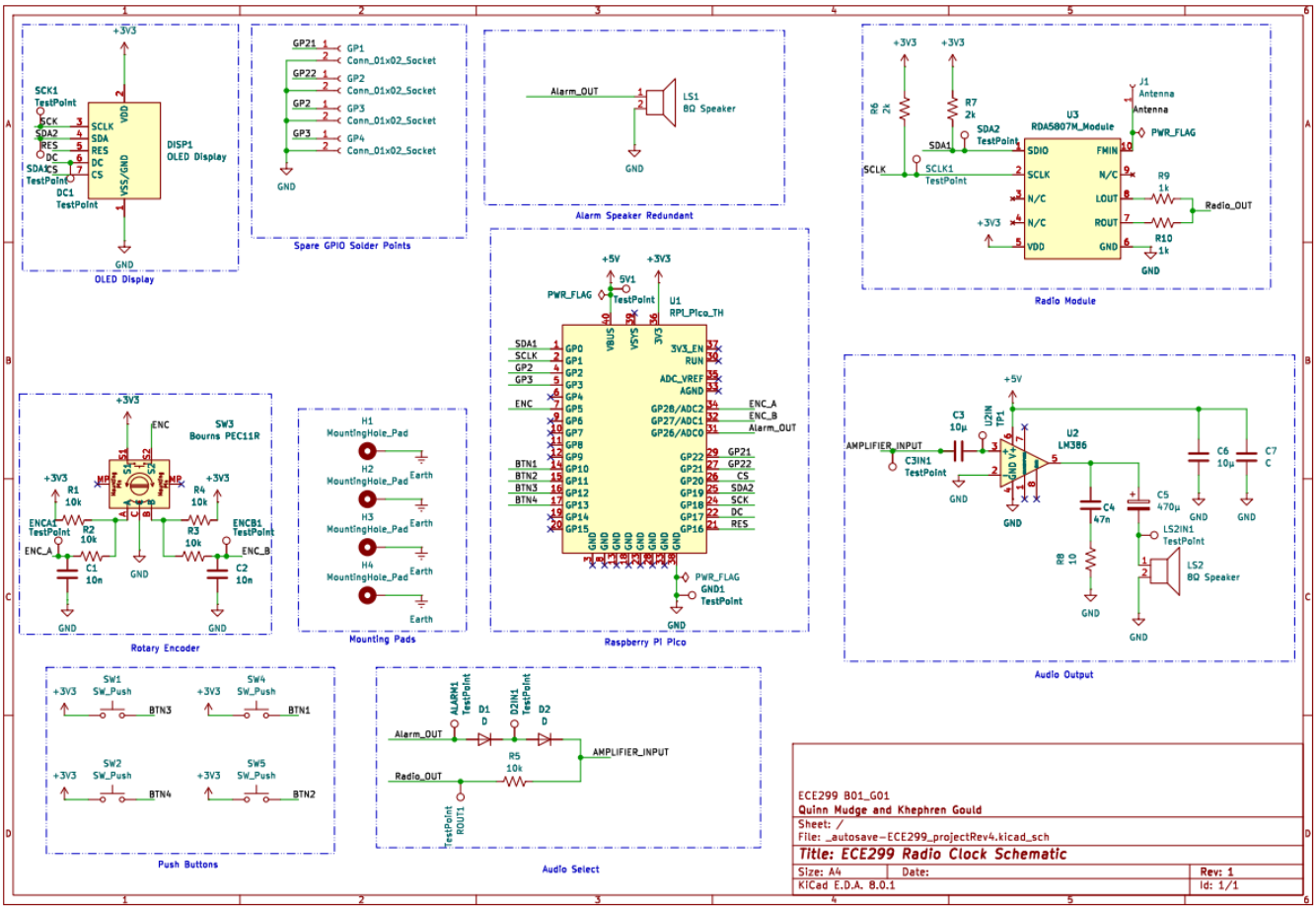

Figure 22: Circuit Schematic

OLED Display

The OLED display circuit schematic is shown in **Figure 23** below. The OLED display is connected to the Raspberry Pi Pico via SPI lines for communication. This setup enables efficient data transfer between the Pico and the display. The display is powered using a 3V3 supply, which is compatible with its operating voltage requirements as described in the datasheet [2].

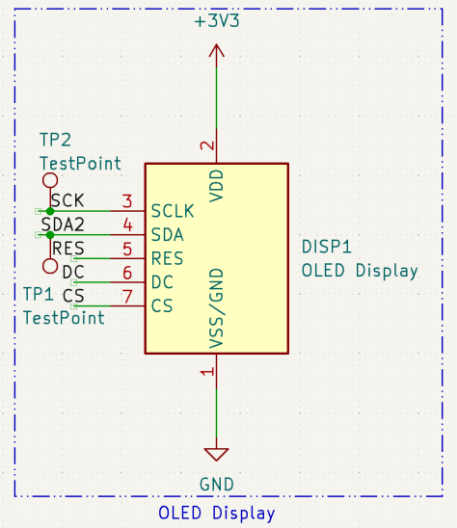

Figure 23: OLED Display Circuit Schematic

Radio Module

The radio module circuit schematic is shown in **Figure 24** below. The radio module uses two pullup resistors on the I2C lines connected to 3V3 when idle. These resistors ensure the SDA and SCL lines are pulled high when not in use, as recommended by the Texas Instruments I2C guide [19]. Pullup resistors are essential for I2C communication because they manage the RC time constant for line charging and discharging. Smaller resistors offer faster communication but higher power consumption, while larger resistors slow communication but reduce power use. The selected resistors balance these factors for reliable performance.

The radio module is powered by 3V3 to match the I2C logic levels and ensure stable operation.

The left and right audio outputs of the radio module are connected in parallel using two resistors. This configuration ensures that no audio channel is lost and maintains balanced audio output across both channels. However, we did not end up combining the signals in our final product. Instead, we used only the left audio output (LOUT), as testing showed that this was adequate for our needs.

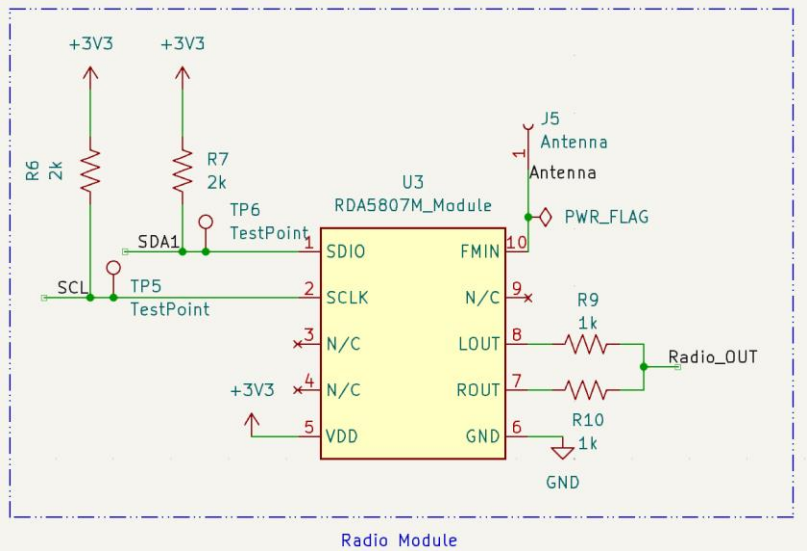

Figure 24: Radio Module Circuit Schematic

Rotary encoder

The rotary encoder circuit schematic is shown in **Figure 25** below. The circuit for the rotary encoder uses the suggestion filtering circuit provided in the PEC11R datasheet [16]. This hardware-based filtering helps to debounce the rotary encoder, addressing the challenges associated with software-based debouncing approaches, which can be complex and less reliable.

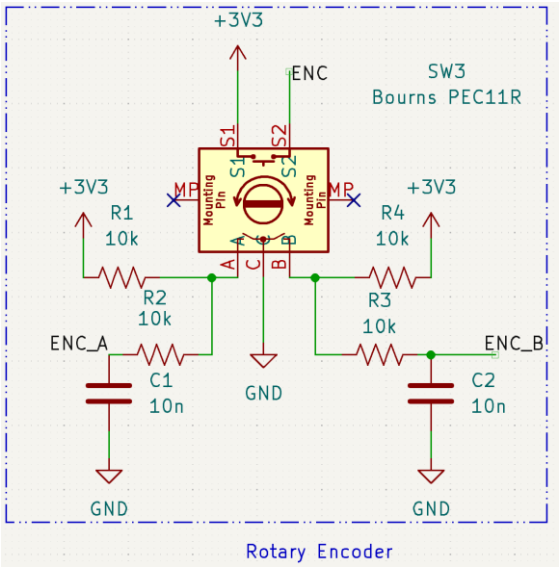

Figure 25: Rotary Encoder Circuit Schematic

Push Buttons

The push button circuit schematic is shown in **Figure 26** below. The four pushbuttons are wired directly from the 3V3 output of the Raspberry Pi Pico to the corresponding GPIO pins. This configuration ensures that pressing a button pulls the GPIO pin voltage to 3V3, which is then detected by the Pico.

In software, the pushbuttons are configured to use the internal pulldown resistors of the Pico. This setup ensures that when a button is not pressed, the GPIO pin reads a low voltage (0V), as the internal pulldown resistor pulls the voltage to ground. When the button is pressed, the GPIO pin reads a high voltage (3V3), providing a clear and reliable signal to the Pico.

Although the circuit was designed with four pushbuttons, the final product did not require all four, demonstrating flexibility in the design to accommodate evolving project needs. The final design only utilized one pushbutton.

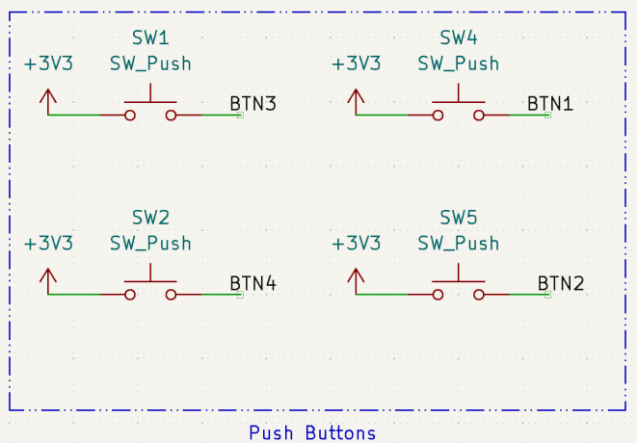

Figure 26: Push Buttons Circuit Schematic

Spare GPIO Solder Points

The circuit schematic is shown in **Figure 27** below. Spare GPIO solder points are included in the design to accommodate potential future expansions or additional features. These points provide flexibility, allowing for easy integration of extra components or functionalities as needed without requiring significant redesign or modification of the existing circuit.

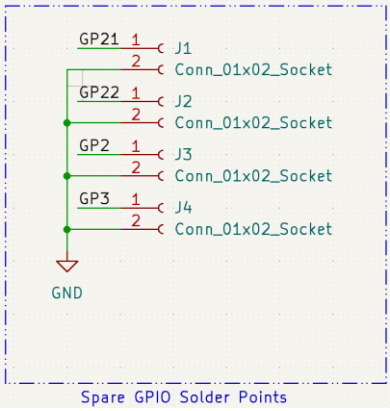

Figure 27: Spare GPIO Solder Points Circuit Schematic

Redundant Alarm Speaker

The redundant alarm speaker circuit schematic is shown in **Figure 28** below. A redundant circuit for an additional alarm speaker was added, connected to a GPIO pin of the Pico and ground. This speaker operated independently of the primary alarm speaker, providing an alternative option for sound output. However, in the final product, this redundant speaker circuit was not utilized.

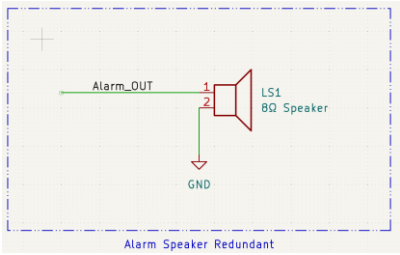

Figure 28: Alarm Speaker Redundant Circuit Schematic

Audio Select

The audio select circuit schematic is shown in **Figure 29** below. The alarm output, played through a GPIO pin, is connected to two diodes, which are the routed in parallel with the radio output. The diodes are used to prevent the output from the radio module from feeding back into the GPIO pin connected to the alarm output. This isolation ensures that the alarm and radio signals do not interfere with each other.

This resistor helps condition the signal from the radio module, managing signal levels to prevent distortion and protecting the circuit from potential overcurrent. The resistor ensures that the audio signal from the radio is clear and properly integrated with the alarm output.

When either the alarm or radio is selected, its audio is direct through the audio output circuit for playback.

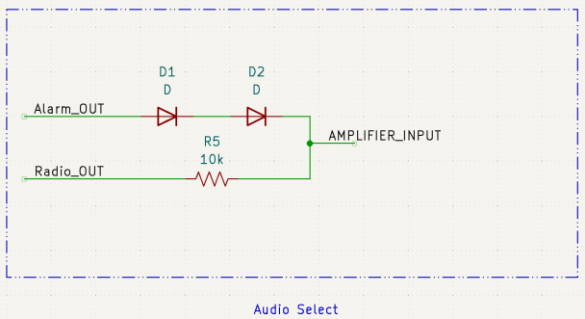

Figure 29: Audio Select Circuit Schematic

Audio Output

The audio output circuit schematic is shown in **Figure 30** below. The selected audio signal, whether from the alarm or the radio, is first fed into a capacitor for filtering DC signals. The filtered signal is then amplified by the LM386 audio amplifier, which is configured with a gain of 20. The LM386's Pin 6 is connected to the 5V power supply, with two capacitors in parallel to this pin for additional filtering. These capacitors are used to bypass high-frequency noises from the 5V power input.

The amplified signal is then passed through a 470 µF capacitor, which is connected between the amplifier output and the speaker. This capacitor blocks any DC component of the signal, preventing it from reaching the speaker and potentially causing damage. The filtered and amplified audio signal is then sent to the 8-ohm speaker, allowing the user to hear the sound clearly.

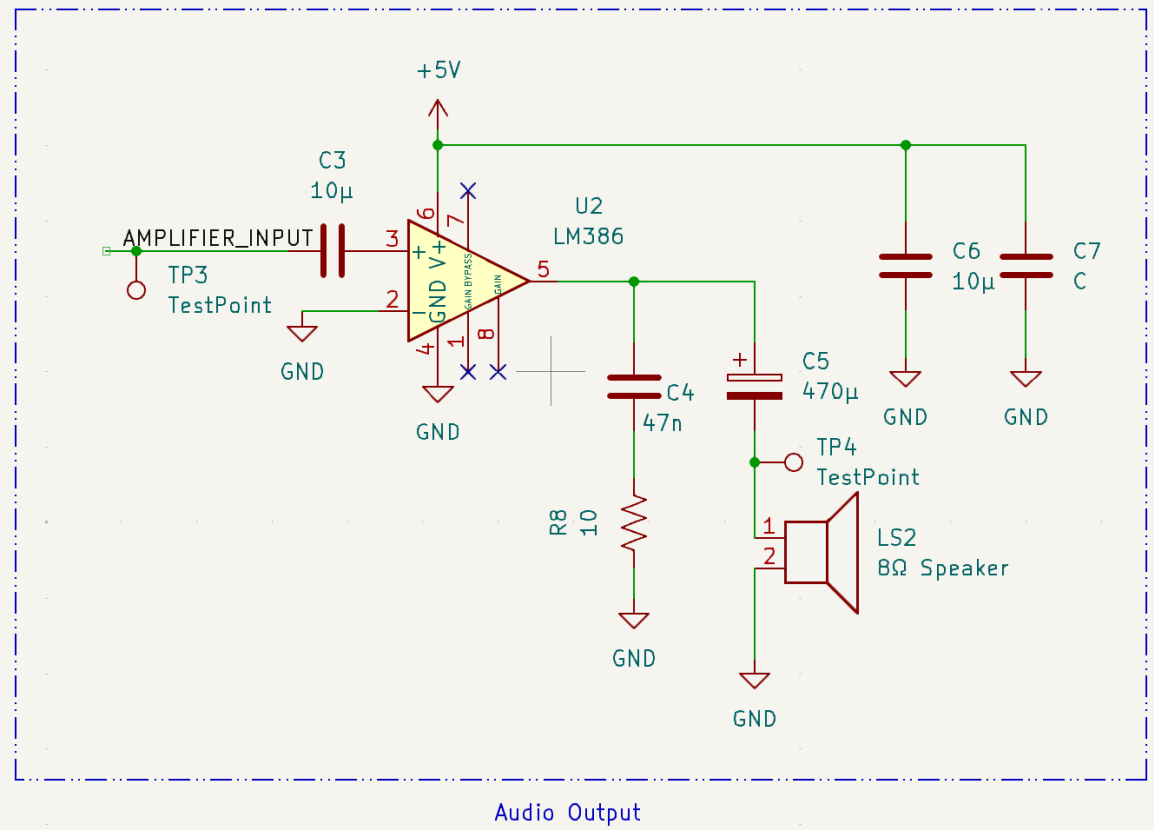

Figure 30: Audio Output Circuit Schematic

The choice for this amplifier circuit was made due to experiments done in the laboratory, following the test procedure described in the LM386 amplifier lab manual [20]. In this experiment, the circuit shown in **Figure 31** was wired on a bread board. The following key results were observed:

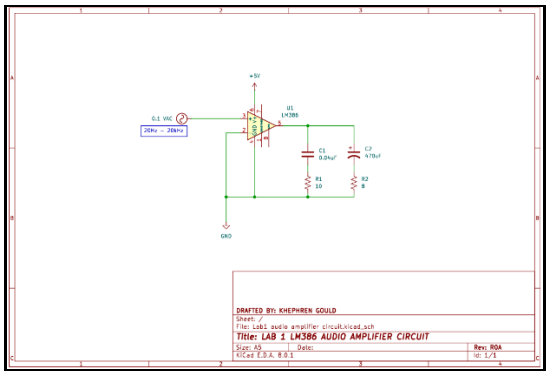

Figure 31: 20 Gain Amplifier Circuit Schematic

For a gain of 20, the amplifier maintained a constant output voltage of approximately 2.25V across the entire frequency range of 20Hz to 20kHz, indicating a bandwidth well above the

audible range. The circuit was powered with +5V, and a sine wave with 0.1V amplitude and 3kHz frequency was input using a signal generator.

The output voltage was measured across different frequencies in the audible range for humans, confirming stable performance, as shown in Table 2.

Frequency (Hz)	Output Voltage (V)
20	2.34
40	2.31
80	2.28
160	2.25
200	2.25
400	2.25
800	2.25
2000	2.25
4000	2.25
8000	2.25
16000	2.25
20000	2.22

Table 2: Frequency Response of LM386

Figures 32 and **33** illustrate the voltage across the resistor and the output voltage of the LM386 for different input frequencies, as measured on an oscilloscope.

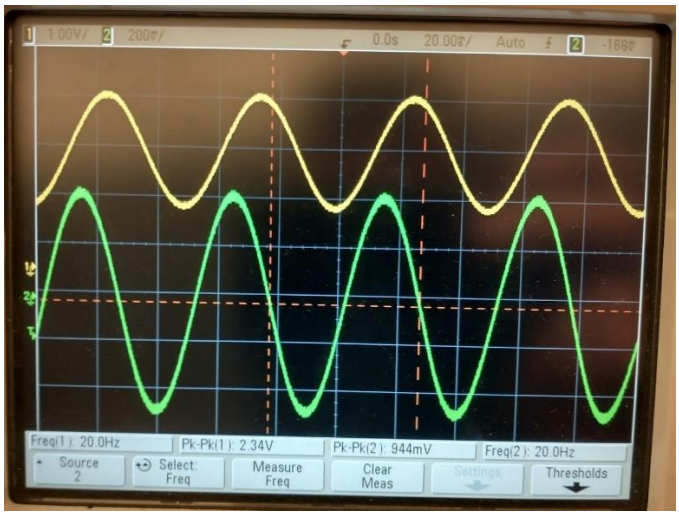

Figure 32: 100mV input at 20Hz, gain of 20, voltage across resistor plotted with output voltage of LM386

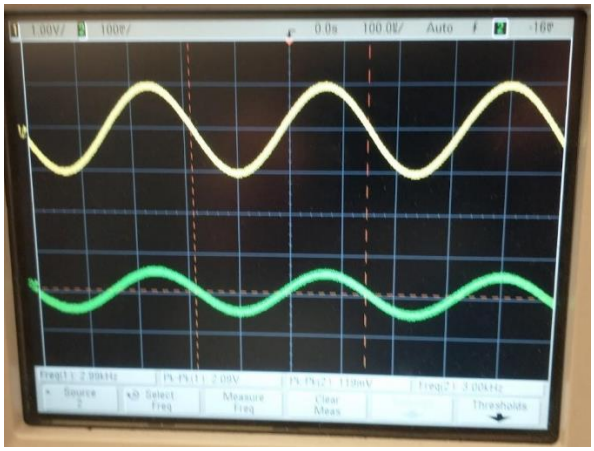

Figure 33: 100mV input at 3kHz, gain of 20, input plotted vs output

VII. Bill of Materials and Cost Analysis

PCB Materials

A list of all components used in the development of the PCB is shown in Table 3

Table 3: Cost of PCB Components

Label in schematic	Component Description	Part number	Cost($)/unit quantity	Source of cost information
C1, C2	10nF Bipolar Ceramic capacitor	C321C103K3G5TA	0.00/2	ECE299 Component List
C3,C6	10 uF Bipolar, Ceramic capacitor	C322C106K3R5TA	0.00/2	ECE299 Component List
C4	47nF Bipolar, Ceramic, capacitor	C327C473K3G5TA	0.00/1	ECE299 Component List
C5	470uF,Polarized, Electrolytic capacitor	35ZLH470MEFC10X16	0.00/1	ECE299 Component List
R1,R2,R3, R4,R5	10kΩ resistor	CF14JT10K0	0.00/5	ECE299 Component List
R6,R7	2kΩ resistor	CF14JT2K00	0.00/2	ECE299 Component List

R8	10Ω resistor	CF14JT10R0TR-ND	0.00/1	ECE299 Component List
R9	1kΩ resistor	CF14JT1K00TR-ND	0.00/1	ECE299 Component List
D1, D2	General purpose rectifier diode	1N4007	1.11/2	Digikey
SW1	Panel Mount Push Button	R13-24A	1.50/1	ECE299 Component List
LS1	8Ω speaker	A18111000UX0172	4.00/1	ECE299 Component List
SW3	Rotary Encoder	PEC11R-4215F-S0024	1.00/1	ECE299 Component List
U1	Raspberry Pi Pico	SC0916	5.00/1	ECE299 Component List
U2	Audio Amplifier	LM386N-4	4.00/1	ECE299 Component List
U3	Radio Module (Through Hole)	RDA5807M	2.00/1	ECE299 Component List
DISP1	1.3” OLED Display	SSD1306	4.00	ECE299 Component List
LS1, SW1	2 Position PCB Terminal Block	1984617	0.64/2	Digikey
DISP1	7 Position Pin Header	1945148	1.00/1	Mouser
U1	10 Position Socket Header	BC065-10-A-L-D	1.01/2	Digikey
U3	5 Position Socket Headers	PPTC051LFBN-RC	0.70/2	Digikey
N/A	PCB	N/A	21.64	PCBWay

The total cost of all PCB components including the cost of PCB manufacturing was $50.42.

Enclosure Materials

A list of all components required for manufacturing the enclosure is shown in Table 4.

Table 4: Cost of Enclosure Materials

Label in schematic	Component Description	Cost($)/unit quantity	Source of cost information
BASE	3D-Printed PLA Enclosure Base	25.60/1	University of Victoria Digital Scholarship Commons
COVER	6mm Laser Cut Plywood	12.30/1	University of Victoria Digital Scholarship Commons
N/A	½" #4-40 Screw	9.50/100	McMaster Carr
N/A	#4-40 Hex Nut	2.20/9	McMaster Carr
N/A	Blue Paint	2	Micheals Crafts
N/A	Green Paint	2	Micheals Crafts
N/A	Brown Paint	2	Micheals Crafts
N/A	White Paint	2	Micheals Crafts
N/A	Yellow Paint	2	Micheals Crafts

The total cost of all materials required for the enclosure was $47.90

Miscellaneous

Table 5 shows a list of all other materials required for manufacturing the clock radio.

Table 5 Cost of Miscellaneous Components

Component Description	Cost/unit quantity	Source of cost information
Paint Brushes	2	Micheals Crafts
USB Cable for Pico	5	ECE299 Component List

The total cost of the components above was $7.00

Analysis

Development of our Clock Radio had a total cost of $105.32. Upon completing cost analysis for each subcomponent (PCB and Enclosure) the PCB came at a cost of $50.42 and the enclosure came at a cost of $47.90. Numerous improvements to lower this cost

are possible including combining our PCB order with other groups, eliminating the use of sockets, and modifying the physical geometry of the PCB.

First, ordering PCBs in bulk (with other groups) would have reduced the shipping cost incurred in acquiring the PCB from $15.62 to $7.81. Additionally, the added cost of the female sockets (used for connecting U1, and U3 to the PCB) could have been eliminated by soldering the raspberry pi and radio module directly to the PCB. These connectors were purchased for convenient insertion and removal of these components for testing and would not be required for further prototypes or a final product.

Designing a smaller PCB with a geometry matching that of the enclosure would have enabled us to significantly reduce the enclosure's size. In doing so, the cost of the 3D printed base would have been reduced by approximately $5.00. In summary, by implementing the improvements suggested above, the overall cost of our Clock Radio would lower by $16.23.

VIII. Printed Circuit Board

The general design paradigms followed when designing the layout included using pours to minimize traces, including test points at power inputs to key components, maximizing trace width for relatively higher current connections, and creating a detailed silkscreen to facilitate the soldering process.

The final PCB layout accounts for an additional speaker and three additional pushbuttons. This was done in case using a single speaker for both alarm and radio output proved unfeasible or if the addition of features required additional pushbuttons.

According to Peterson, ground planes have many advantages, including minimizing Electromagnetic induction (which causes interference between signals), and reducing PCB size [21]. For our PCB, the back copper was used as a ground plane. All components were connected to common ground using this plane. According to Peterson, ground planes have many advantages, including minimizing Electromagnetic induction (which causes interference between signals), and reducing PCB size [21]. For our PCB, the back copper was used as a ground plane. All components were connected to common ground using this plane. The front copper was used as a 3.3V power plane. The 3.3V output of the Pico powered many of our components and adding a 3.3V plane further reduced the number of traces required on the PCB.

20mil trace width was used to for all 5V power supply traces and 30mil trace width was used from the amplifier output to the speaker input. Based on measurements performed in the lab, at volume 5 (the maximum volume setting for our clock radio) the voltage across

the speaker was approximately 4V. With a voltage (V) of 4V and a speaker resistance (R) of 8Ω the current (I) through the speaker can be obtained using Ohm's law:

$$I = \frac{V}{R} \tag{1}$$

$$I = 0.5A$$

As a 0.5A current is below the 2A limit for a trace width of 30mil suggested by Peterson [22] a 30-mil trace width was selected.

On the silkscreen, all passive circuit components such as resistors, and capacitors were labeled with their values and identifiers from the schematic. Additionally, all test points were labelled with their associated net. This made the soldering process much quicker while also facilitating the validation process.

Figures 33 and **34** show the final PCB layout.

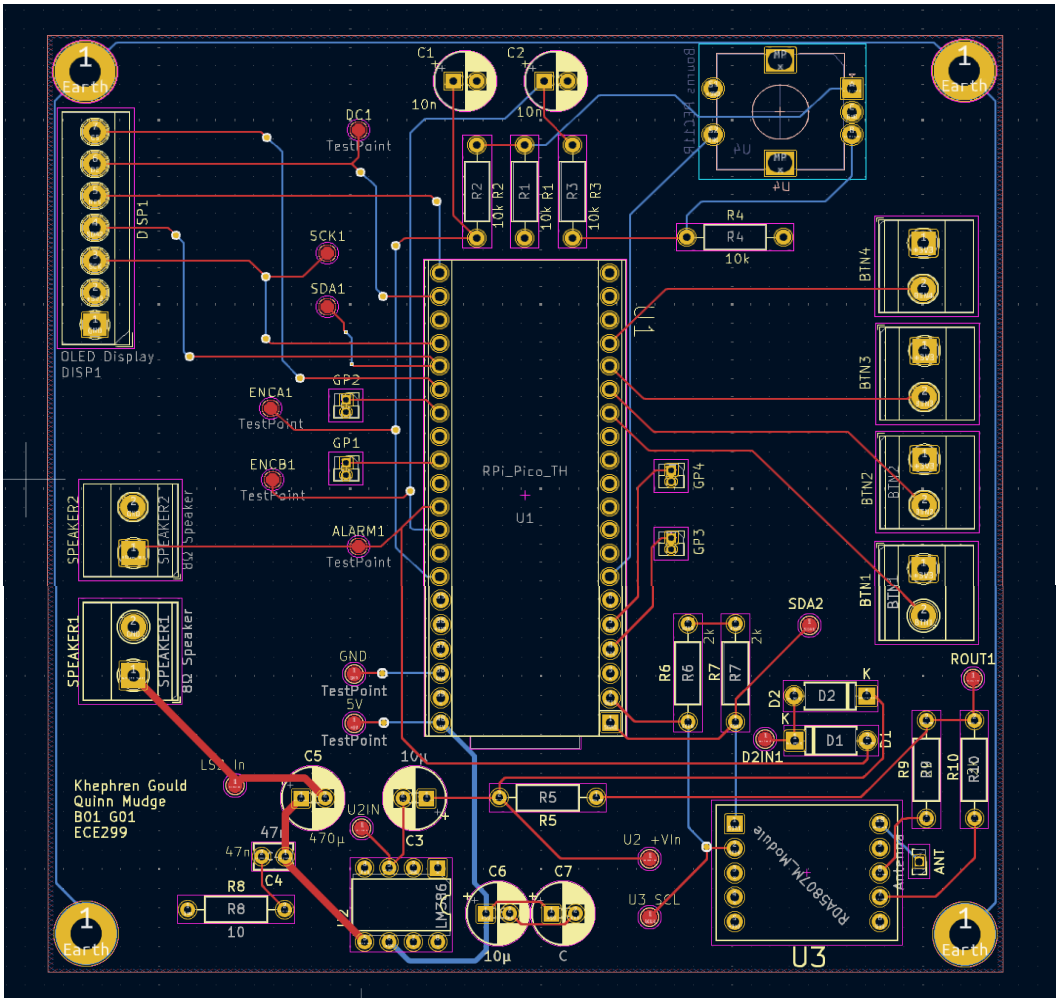

Figure 34: PCB Layout

Figure 35 shows the 3D model of our PCB generated from KICAD.

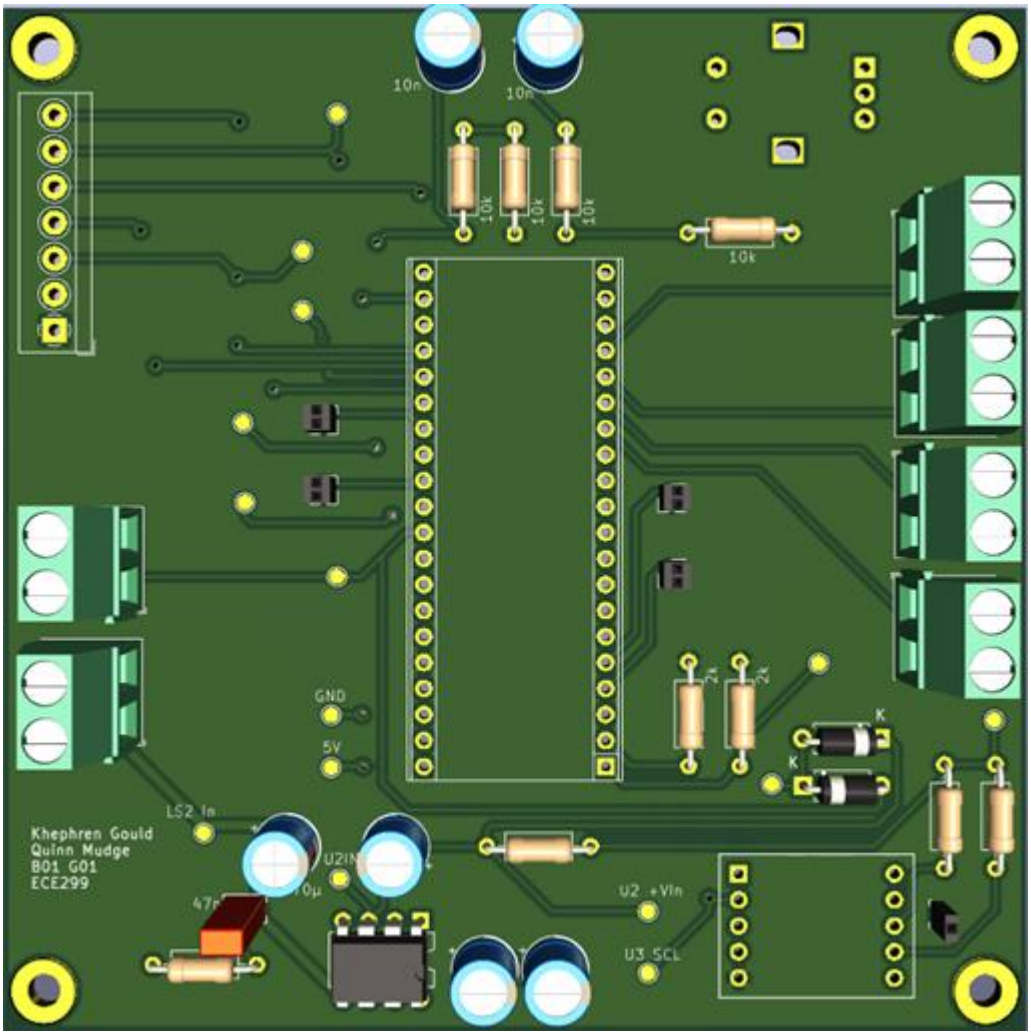

Figure 35: PCB 3D Model

Finally, **Figure 36** shows the manufactured PCB.

Figure 36 Manufactured PCB

IX. Enclosure Design

Figure 37 shows a front view of the SolidWorks model for our enclosure. The enclosure consists of a wooded cover and a plastic base. As seen in the figure, the cover connects to the base through three 20mm extrusions and a #4-40 tapped hole. A grid was created on the cover for the speaker to play audio through

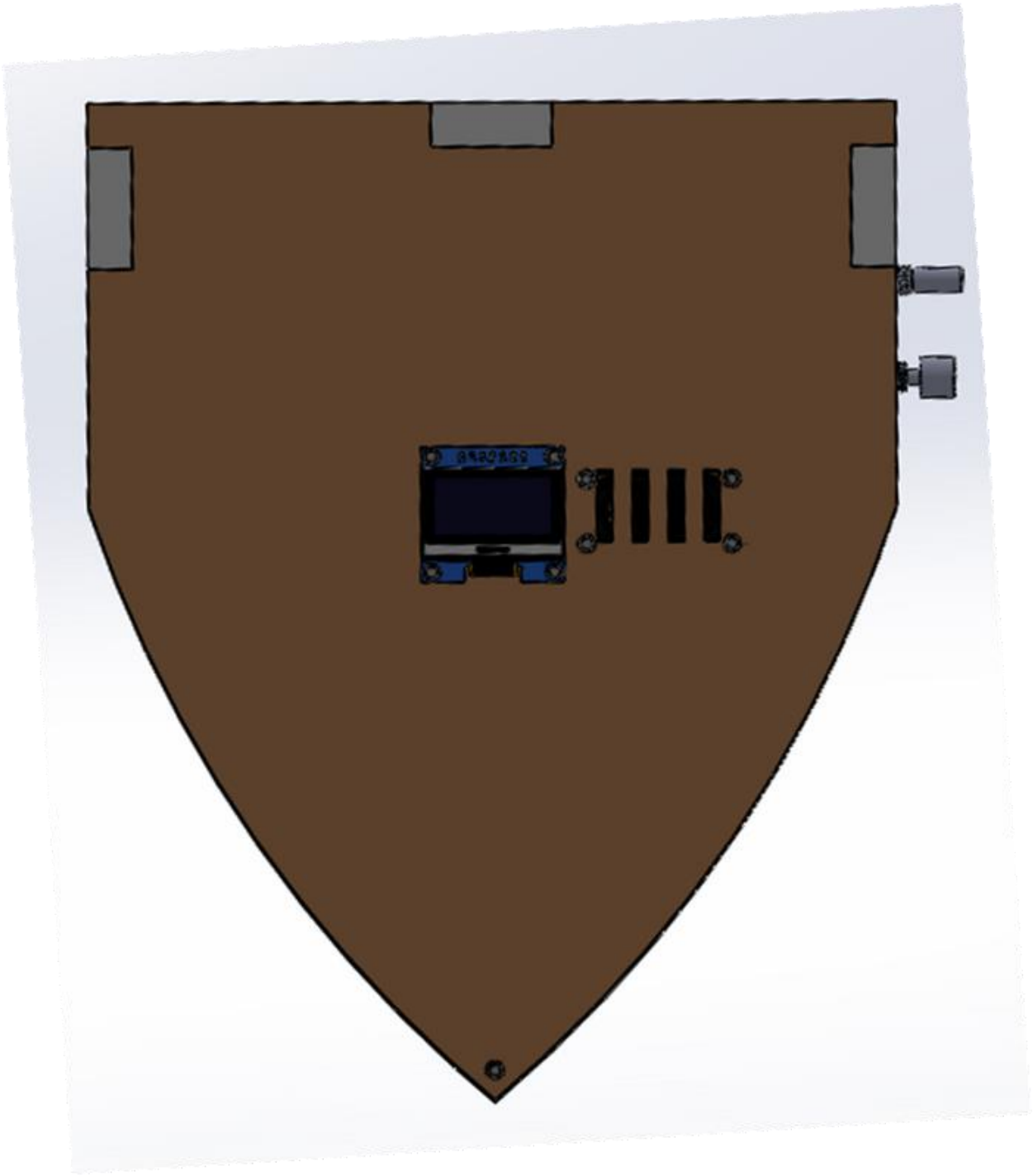

Figure 37: Enclosure Assembly

Figures 38 and 39 show transparent and side views of the enclosure. As seen in **Figure 39** models for all components was included in the device to ensure all holes were aligned and enough space was included for all components. Appendix B contains a drawing set for the enclosure.

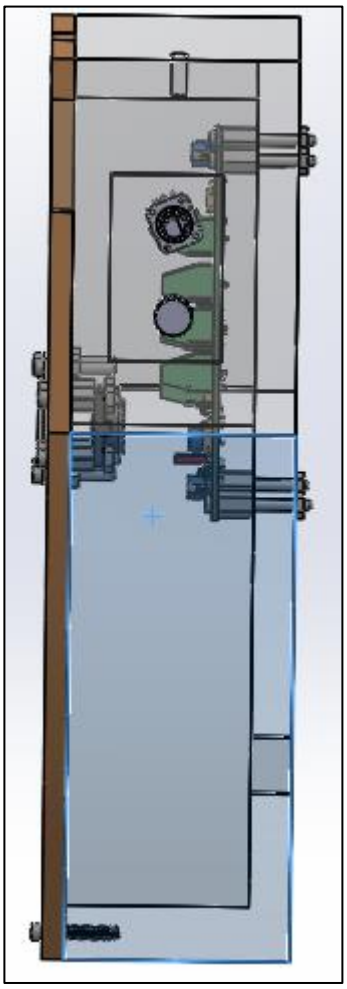

Figure 39: Enclosure Assembly - Transparent

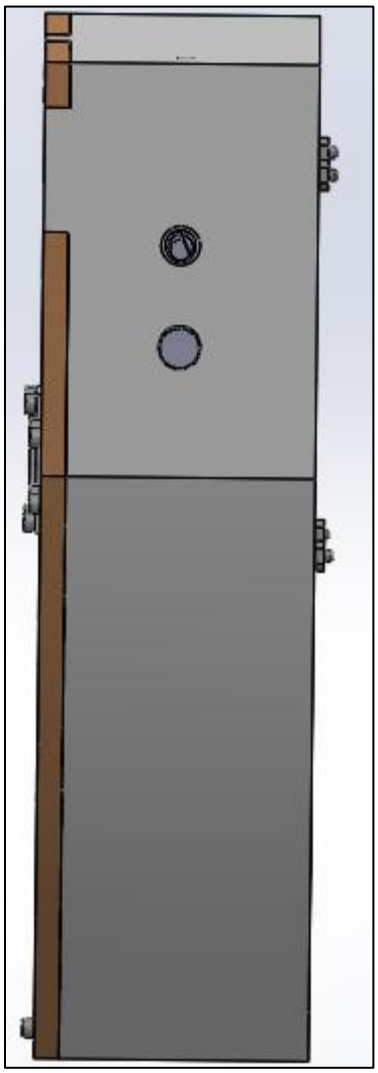

Figure 38: Enclosure Assembly – Side View

Figure 40 shows the SVG file designed in Inkscape. Black and white images for the Bison and BC Flag designs and the "Bucket Fill" tool from Inkscape was used to assign the appropriate colours. We selected the bison as it is featured in the crest on the Manitoba flag and the rising sun as it is featured on the BC flag. All red objects were engraved, and all black lines were cut. We ensured the dimensions of the cover matched those of the base by exporting the SVG directly from the SolidWorks model.

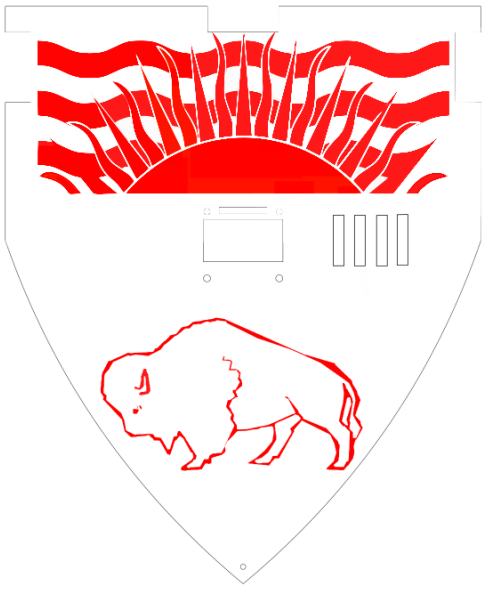

Figure ~~41.~~40: Cover SVG File

Additional design choices for the enclosure included using #4-40 screws and #4-40 hex nuts to fasten all components. This added significant durability to the final product. Additionally, the base was 3D printed at 20% infill.

X. Testing and validation

Set/Edit Time Through User Interface.

Objective:

To verify that the user can correctly set and edit time using the interface, and that the system behaves as expected in both 24-hour and 12-hour formats.

Equipment Required:

- Raspberry Pi Pico
- SSD1306 OLED Display
- Rotary Encoder (with pushbutton)
- Pushbutton
- Breadboard and wires
- OLED Display Circuit (shown in **Figure 23**)
- Rotary Encoder Circuit (shown in **Figure 25**)
- Provided Code

Pre-test Setup:

- Download the provided test code from Appendix A
- Upload the code to the Raspberry Pi Pico using a compatible IDE (e.g., Thonny)

Test Steps:

1. **Select Clock Mode:**

- Using the menu and enter buttons, navigate to and select the clock button on the user interface

For 24 hour format:

2. **Set Minutes:**

- Use the menu button to navigate to the “minute” setting
- Rotate the encoder to select the desired value for the minutes
- Verify that the minute value displayed in the center of the display updates as the encoder is rotated both clockwise and counterclockwise.
- Ensure that the minute value cannot exceed 59 or be less than 0

3. Set Hours:

- Use the menu button to navigate to the “hour” setting
- Rotate the encoder to select the desired hour value

- Verify that the hour value displayed in the center of the display updates as the encoder is rotated both clockwise and counterclockwise
- Ensure that the hour value cannot exceed 23 or be less than 0

For 12 hour format:

4. **Set Minutes:**
- Repeat step 2 for setting minutes
- Verify that the minute value cannot exceed 59 or be less than 0
5. **Set Hours:**
- Use the menu button to navigate to the “hour” setting
- Rotate the encoder to select the desired value for the minutes
- Verify that the hour value displayed in the center of the display updates as the encoder is rotated both clockwise and counterclockwise.
- Ensure that the hour value ranges correctly from 1am to 11am, 12pm to 11pm, 12am, then back to 1am

Post-test Verification:

6. **System Stability:**
- Observe the system for a few minutes to ensure the time updates correctly and the system remains active
7. **Edge Cases:**
- Attempt to set invalid times (e.g., 60 minutes, 24 hours) and verify that the system prevents these settings
- Set the time to 23:59 (or 11:59 pm) and verify that the time turns to 00:00 (or 1:00am)

Set and Trigger Alarm

Objective:

To verify that the alarm functionality works correctly as set by the user, including setting and editing the alarm time, selecting different alarm tones, and adjusting the volume.

Equipment Required:

- Raspberry Pi Pico
- SSD1306 OLED Display
- Rotary Encoder (with pushbutton)
- Pushbutton
- Provided Code

- Audio Output Circuit (shown in **Figure 30**)
- OLED Display Circuit (shown in **Figure 23**)
- Rotary Encoder Circuit (shown in **Figure 25**)
- Breadboard and wires
- Provided Code from Appendix A

Pre-test Setup:

1. Ensure that the Raspberry Pi Pico is powered on, and the clock radio software is correctly loaded and running
2. Build the OLED display and audio output circuit

Test Steps:

1. **Set Alarm Time**
- Set an alarm for one minute from the current time using the user interface
- Wait for the alarm to trigger and verify that the alarm sound plays correctly
- Observe and record the behavior of the alarm when it triggers (e.g., sound, display notification)
2. **Edit Alarm Time**
- Restart the project
- Set a new alarm time for two minutes from the current time
- Verify that the alarm triggers correctly after two minutes and that it matches the set time
3. **Test Different Alarm Tones**
- Using the user interface, select each of the three different alarm tones one by one
- Set an alarm for a short period (e.g., one minute) and verify that each tone plays correctly when the alarm triggers
4. **Test Volume Control**
- Set the volume to different levels ranging from 0 to 5
- Verify that the volume adjustment affects the alarm sound as expected
- Check that the volume levels are distinct and operate as expected (e.g. no distortion)
5. **Edge Case Testing:**
- Verify that the alarm can only be set to valid times, and ensure the system correctly handles any attempts to set invalid alarm times

Tune to Radio Channel Through the User Interface

Objective:

To verify that the radio tuning functionality works correctly through the user interface, including manual frequency adjustment and the seek function

Equipment Required:

- Raspberry Pi Pico
- RDA5807 Radio Module
- SSD1306 OLED Display
- Rotary Encoder
- Pushbutton
- Audio Output Circuit (shown in **Figure 30**)
- Radio Module Circuit (shown in **Figure 24**)
- OLED Display Circuit (shown in **Figure 23**)
- Rotary Encoder Circuit (shown in **Figure 25**)

Pre-test Setup:

1. Build and test the above circuits to ensure proper connections and functionality.
2. Verify that the radio module is correctly interfaced with the Raspberry Pi Pico, and the "rda5807.py" library is loaded

Test Steps:

1. **Manual Frequency Adjustments**

- Access the radio menu on the OLED Display
- Navigate to the frequency adjustment icon
- Use the rotary encoder to increment the frequency by 0.1Mhz. Verify that the frequency displayed on the OLED changes accordingly.
- Continue to increment the frequency in 0.1 MHz steps until the radio module reaches a known radio frequency. Verify that the correct frequency is tuned and that the radio output corresponds to this frequency
- Test decrementing the frequency by 0.1 MHz in a similar manner, ensuring the radio module tunes to frequencies correctly and the display updates as expected

2. Seek Function

- Access the radio menu on the OLED display
- Navigate to the seek function icon
- Initiate the seek-up function and verify that the radio module automatically tunes to the next available station. Ensure that the frequency displayed on the OLED updates to reflect the new station and that a clear signal is received
- Repeat the process with the seek-down function to verify that the radio module correctly tunes to previous stations

3. **Edge Case Testing:**

- Test the frequency adjustment by setting the frequency to the lowest (88MHz) and highest (108MHz) allowable limits. Verify that the program handles these boundary values and does not allow the user to input an invalid frequency

XI. Code of Ethics

List the relevant code of ethics of EGBC and describe how you have followed the relevant codes in your project

The relevant EGBC code of ethics [23] that apply to this project are as follows:

1. Hold paramount the safety, health, and welfare of the public, including the protection of the environment and the promotion of health and safety in the workplace;

Relevance:

By ordering lead-free PCBs, we ensured that our project minimizes environmental hazards and promotes safety. Lead-free materials reduce the risk of toxic exposure, contributing to a safer environment and workplace. Additionally, we made sure the device is safe to use by powering it with a COTS 5V phone charger, which significantly reduces the risk of electrocution.

7. Provide professional opinions that distinguish between facts, assumptions, and opinions;

Relevance:

We tried to base our project decisions on solid research and provided references for our sources. By clearly separating facts from assumptions and personal opinions in our documentation, we aimed to make sure our work was reliable and transparent.

11. Clearly identify each registrant who has contributed professional work, including recommendations, reports, statements, or opinions;

Relevance:

We used a Gantt chart to break down the tasks of the project by role, which helped us clearly identify and assign responsibilities. This approach ensured that everyone's contributions were documented and acknowledged, making it easier to recognize who was involved in each aspect of the project.

12. Undertake work and documentation with due diligence and in accordance with any guidance developed to standardize professional documentation for the applicable profession; and

Relevance:

We documented various aspects of our project, including the design of the enclosure, the circuit schematic, the PCB layout, the code, and three test procedures. This thorough documentation ensured that our work followed standard practices and provided a clear and comprehensive record of our project.

13. Conduct themselves with fairness, courtesy, and good faith towards clients, colleagues, and others, give credit where it is due and accept, as well as give, honest and fair professional comment.

Relevance:

We made sure to be respectful and professional with everyone involved, including classmates, teammates, lab TAs, and instructors.

XII. Conclusions and recommendations

This project aimed to successfully design, develop, present, and document a clock radio. Many design and project management decisions benefitted the project's results.

As almost every desired feature was software reliant, programming efficiently and maintaining organized was critical. Using an object-oriented organization allowed for division of the main program into smaller components (classes) which were assigned between group members. Another important aspect of the design was maintaining efficiency of operation while minimizing input devices. By implementing software defined icons and buttons we implemented all required functionalities with only two input devices. Using SolidWorks was beneficial to ensure all components mounted off the PCB integrated with the enclosure properly. By using SolidWorks, the enclosure required only a single iteration, was durable, supported all components, and fit the theme.

As many improvements can be made to the design including PCB layout, alarm software, and power supply. Additionally, numerous features could have been added to further improve the user experience.

The first recommendation consists of reducing the PCB size and modifying the PCB shape. As the enclosure manufacturing cost came to almost $40, size was a significant issue for the enclosure. The sized could have been reduced if the PCB was made smaller and it's shape

was changed from rectangular to crest like in order to scale down the enclosure. An additional improvement consists of revising the alarm software. In the final demonstration, a bug in the alarm software caused it to trigger prematurely. We determined adjusting the time zone before setting the alarm caused this problem. In the future, further edge case testing should be done to mitigate similar issues. Finally, we limited the volume range on the final product to only range from 0 to 5 due to distortion occurring at higher levels. Providing the LM386 with a higher input voltage (such as 9V) could eliminate distortion and allow for a larger volume range.

References

[1] "Brightspace ECE 299," Project Expectations and Constraints, [Online]. Available: https://bright.uvic.ca/d2l/le/content/345595/viewContent/2785614/View. [Accessed 28 07 2024].

[2] "SSD1306," Solomon Systech Semiconductor, 04 2008. [Online]. Available: https://www.solomon-systech.com/product/ssd1306/. [Accessed 28 07 2024].

[3] Z. Luo and S.-T. Wu, "OLED Versus LCD: Who Wins?," *Opt. Photonics News,* pp. 19-21, 2015.

[4] "ECE 299 Brightspace," ECE 299 Components List, [Online]. Available: https://bright.uvic.ca/d2l/le/content/345595/viewContent/2785613/View. [Accessed 28 07 2024].

[5] "RDA5807M," RDA Microelectronics, May 2011. [Online]. Available: https://cdn-shop.adafruit.com/product-files/5651/5651_tuner84_RDA5807M_datasheet_v1.pdf. [Accessed 30 07 2024].

[6] "Radio Stations in Victoria BC," WorldRadioMap, [Online]. Available: https://worldradiomap.com/ca/victoria. [Accessed 30 July 2024].

[7] "Raspberry Pi Pico Datasheet," Raspberry Pi, 02 May 2024. [Online]. Available: https://datasheets.raspberrypi.com/pico/pico-datasheet.pdf. [Accessed 30 July 2024].

[8] J. Mankar, C. Darode, K. Trivedi, M. Kanoje and P. Shahare, "Review of I2C Protocol," *International Journal of Research in Advent Technology,* vol. 2, no. 1, pp. 474-479, 2014.

[9] "TEA5767HN," NXP Semiconductors, 26 January 2007. [Online]. Available: https://www.sparkfun.com/datasheets/Wireless/General/TEA5767.pdf. [Accessed 30 July 2024].

[10] "Si7402/03," Silicon Laboritories, 2006. [Online]. Available: https://pdf1.alldatasheet.com/datasheet-pdf/view/201123/SILABS/SI4703.html. [Accessed 30 July 2024].

[11] "BK1086/88," Beken, 2010. [Online]. Available: https://xn--p8jqu4215bemxd.com/wp-content/uploads/2019/08/BK1086-88-Datasheet-v1.0.pdf. [Accessed 30 July 2024].

[12] "SparkFun FM Tuner Evaluation Board - Si7403," SparkFun, 13 August 2015. [Online]. Available: https://www.sparkfun.com/products/12938. [Accessed 30 July 2024].

[13] Texas Instruments, "LM386 Low Voltage Audio Power Amplifier datasheet (Rev. D)," August 2023. [Online]. Available: https://www.ti.com/lit/ds/symlink/lm386.pdf. [Accessed 3 August 2024].

[14] A. G. F. D. e. a. Purves D, "The Audible Spectrum," *Neuroscience,* no. 2, 2001.

[15] "R1324ABR," Shin Chin, 13 April 2023. [Online]. Available: https://www.mouser.ca/datasheet/2/209/Shin_Chin_R13_24A_05_BR_112_R13_24A_R-3388682.pdf. [Accessed 2 August 2024].

[16] Bourns, "PEC11R Series 12 mm Incremental Encoder," Bourns, 2018. [Online]. Available: https://www.bourns.com/docs/product-datasheets/pec11r.pdf. [Accessed 3 August 2024].

[17] J. Cook, "Rotary Encoders vs. Potentiometers: Which Is Right for Your Project?," Arrow Electronics, 19 May 2019. [Online]. Available: https://www.arrow.com/en/research-and-events/articles/encoder-vs-potentiometer-how-to-choose. [Accessed 1 August 2024].

[18] R. Ayeras, "What Causes the I2C Bus Lock and How Do I Fix It So I Can Program a Device?," Total Phase, 1 January 2020. [Online]. Available: https://www.totalphase.com/blog/2020/01/what-causes-the-i2c-bus-lock-and-how-do-i-fix-it-so-i-can-program-a-device/. [Accessed 13 July 2024].

[19] J. Wu, "A Basic Guide to I2C," Texas Instruments, 2022. [Online]. Available: https://www.ti.com/lit/an/sbaa565/sbaa565.pdf?ts=1722706958089#:~:text=I2C%20uses%20a%20sequence%20of,line%20to%20a%20high%20level.. [Accessed 3 August 2024].

[20] E. 2. Brightspace, "LM386-amplifier-lab manual - Summer 2024 ECE 299 A01 (30316)," [Online]. Available: https://bright.uvic.ca/d2l/le/content/345595/viewContent/2779409/View. [Accessed 4 August 2024].

[21] Z. Peterson, "Understanding 2-Layer PCB Ground Planes," Altium, 30 March 2018. [Online]. Available: https://resources.altium.com/p/understanding-ground-planes-your-two-layer-pcb. [Accessed 3 August 2024].

[22] P. Zacharia, "PCB Trace Width vs. Current Table for High Power Designs," Altium, 1 December 2019. [Online]. Available: https://resources.altium.com/p/pcb-trace-width-vs-current-table-high-voltage-design. [Accessed 3 August 2024].

[23] "Engineers Geoscientists British Columbia," Code of Ethics, [Online]. Available: https://www.egbc.ca/complaints-discipline/code-of-ethics/code-of-ethics. [Accessed 27 07 2024].

[24] [Online].

[25] john, "doe," 2024. [Online]. Available: http://www.adatum.com.

[26] "Si4703-B17," Silicon Labs, 2007. [Online]. Available: https://www.digikey.com/en/htmldatasheets/production/1006291/0/0/1/si4703-b17. [Accessed 30 July 2024].

Appendix A – Test Procedure Source Code and Results

Set and Edit Time

Set_Edit_Time_Test.py

ECE299-Radioclock/Set_Edit_Time_Test.py at main · quinnmudge/ECE299-Radioclock (github.com)

Test Results

Clock Test 1 (youtube.com)

Clock Test 2 (youtube.com)

Clock Test 3 (youtube.com)

Set and Trigger Alarm

Set_Trigger_Alarm_Test.py

ECE299-Radioclock/Set_Trigger_Alarm_Test.py at main · quinnmudge/ECE299-Radioclock (github.com)

Test Results

Trigger Alarm Test (youtube.com)

Tune to Radio Channel Through User Interface

Radio_Test.py

ECE299-Radioclock/Radio_Test.py at main · quinnmudge/ECE299-Radioclock (github.com)

Test Results

Tune Radio Test (youtube.com)

Appendix B– Enclosure Drawing Set

The drawing set for our clock radio consists of **Figures C1, C2**, and **C3** below

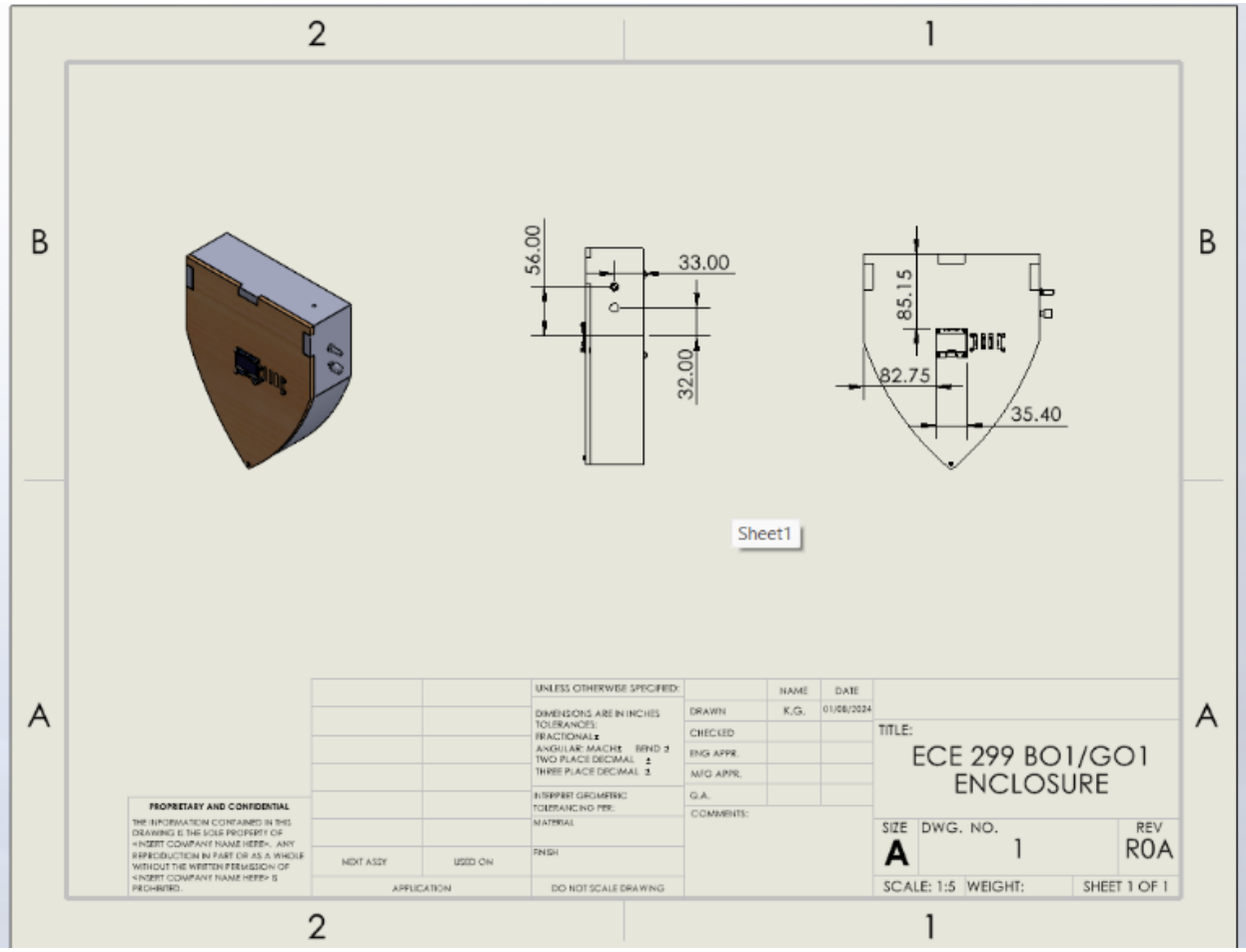

Figure B 1 Assembly Drawing

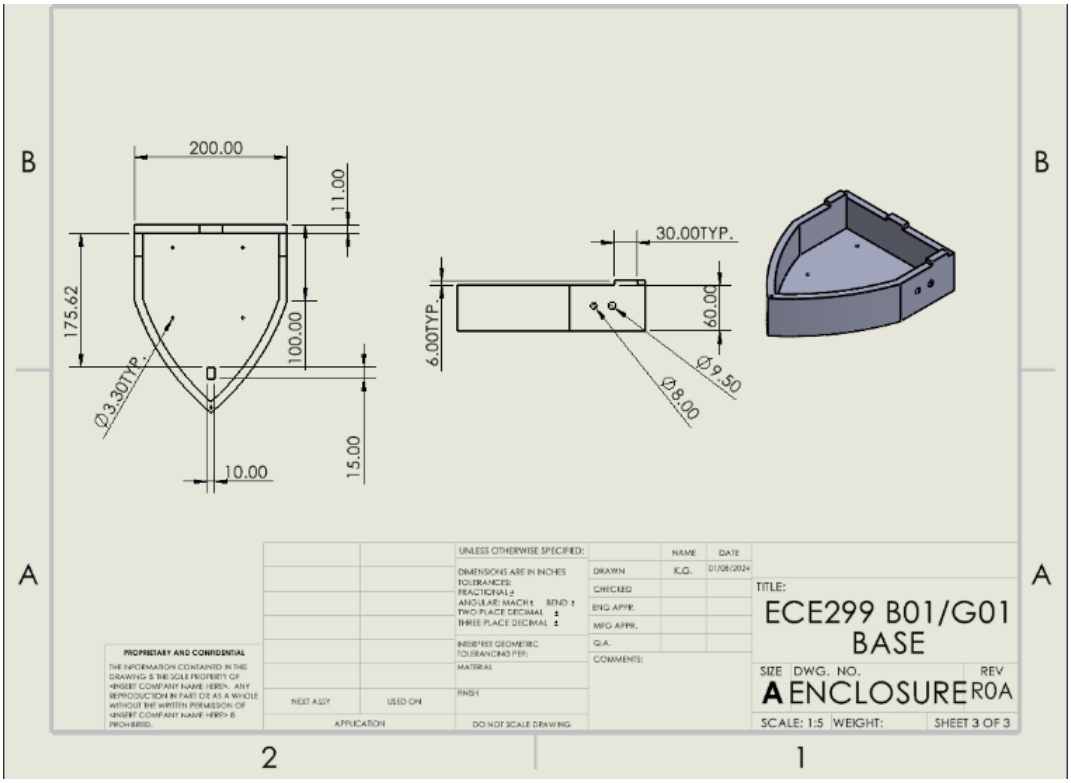

Figure B 2 Base Drawing

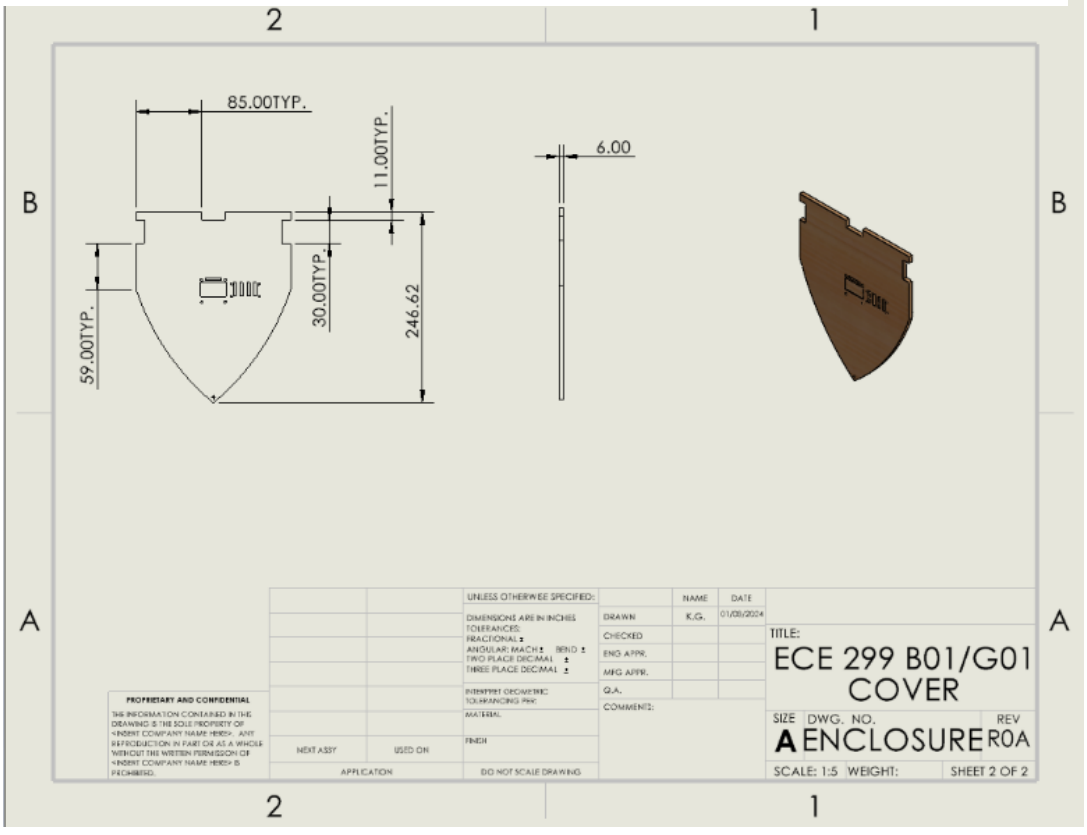

Figure B 3 Cover Drawing

Appendix C – Source Code

External Radio Module Communication Library

101Things/19_fm_radio/rda5807.py at master · dawsonjon/101Things (github.com)

Main Program

Editing ECE299-Radioclock/main.py at main · quinnmudge/ECE299-Radioclock (github.com)

```
#import all required libraries
from machine import Pin, SPI, I2C, RTC, Timer, PWM
from buzzer_music import music
from time import sleep
import ssd1306
import utime
import rda5807

#configure encoder pins
EncoderA = machine.Pin(27, machine.Pin.IN, machine.Pin.PULL_UP)
EncoderB = machine.Pin(28, machine.Pin.IN, machine.Pin.PULL_UP)
machine.Pin(23, machine.Pin.OUT)
# Initialize variables to keep track of encoder state
A_state = 0
B_state = 0
A_rising_edge = False
A_falling_edge = False
B_rising_edge = False
B_falling_edge = False
rotation_direction = 0  # 1 for clockwise, -1 for counterclockwise
#The current position of the "selector" on the screen
current_posx = 1
current_posy = 4
#the gridsized used to organize buttons and icons on the display
grid_size_x = 42
grid_size_y = 10
ENTER = False


# Define button pins
button_1 = Pin(13, Pin.IN, Pin.PULL_DOWN)  # Button for moving left
enter = Pin(5, Pin.IN, Pin.PULL_DOWN)     # Button for enter

# Debouncing variables
last_pressed_time = 0
debounce_delay = 50  # Adjusted for non-blocking debounce
```

```
    radio_programming_timer = utime.ticks_ms()
    # RTC Setup
    rtc = RTC()
    rtc.datetime((2024, 7, 10, 0, 0, 0, 0, 0))  # Set initial RTC time (year, month, day, weekday, hours, minutes, seconds, subseconds)
    SNOOZE = [0,0,0,0,0,0,0,0]

    def floats_are_equal(a: float, b: float, tolerance: float = 0.05) -> bool:
        return abs(a - b) <= tolerance

    class Icon:
        def __init__(self, txt, pos_x, pos_y, border):
            #text displayed by the icon
            self.text = txt
            #position of the icon on the screen
            self.xpos_text = pos_x * grid_size_x + 2
            self.ypos_text = pos_y * grid_size_y + 2
            #width and height of the icon's border
            self.width = grid_size_x + 2
            self.height = grid_size_y + 2
            #decide whether to show border (white rectangle) or not
            self.has_border = border

    class Button(Icon):
        def __init__(self, txt, pos_x, pos_y, select, has_border):
            super().__init__(txt, pos_x, pos_y, has_border)
            self.grid_x = pos_x
            self.grid_y = pos_y
            #used to track if the selector's position matches the button's position
            self.selected = select
            #track what state change should occur when button is pressed
            self.state = None  # Initially no state assigned
        #setter function to set the "state" member
```

```
69          def configureState(self, the_state):
70              self.state = the_state
71
72      class Display:
73          def __init__(self, width, height):
74              #configure SPI communication
75              self.spi_sck = Pin(18)
76              self.spi_sda = Pin(19)
77              self.spi_res = Pin(16)
78              self.spi_dc = Pin(17)
79              self.spi_cs = Pin(20)
80              SPI_DEVICE = 0
81              self.oled_spi = SPI(SPI_DEVICE, baudrate=100000, sck=self.spi_sck, mosi=self.spi_sda)
82              self.SSD = ssd1306.SSD1306_SPI(width, height, self.oled_spi, self.spi_dc, self.spi_res, self.spi_cs)
83              self.SSD.fill(0)
84              self.SSD.show()
85
86          def invert_region(self, x, y, width, height):
87              # Invert a specific region on the display
88              for i in range(x, x + width):
89                  for j in range(y, y + height):
90                      pixel = self.SSD.pixel(i, j)
91                      self.SSD.pixel(i, j, not pixel)
92
93          def update_buttons(self, icons):
94              global current_state, ENTER, current_posx, current_posy
95              #loop over the list of icons being rendered
96              for icon in icons:
97                  #only buttons have functionality and must be updated
98                  if isinstance(icon, Button):
99                      #if the button position matches the selector position set "selected"
100                     #boolean True
101                     if icon.grid_x == current_posx and icon.grid_y == current_posy:
102                         icon.selected = True
103                         # if the encoder button has been pressed and the
```

```
102                     icon.selected = True
103                    # if the encoder button has been pressed and the
104                     # selected button has a state assigned, change states
105                     if ENTER and icon.state:
106                         current_state = icon.state
107                         #current_state.update()
108                         change_state(current_state)
109                     #reset enter variable (to wait for next encoder button press)
110                     ENTER = False
111                 else:
112                     icon.selected = False
113 
114     def render(self, icons):
115         global current_state
116         self.SSD.fill(0)
117         for icon in icons:
118             if isinstance(icon, Button) and icon.selected:
119                 # Draw the icon text first
120 
121                 self.SSD.text(icon.text, icon.xpos_text, icon.ypos_text, 1)
122                 # Invert the region of the selected icon
123                 if(len(icon.text)==6):
124                     self.invert_region(icon.xpos_text - 2, icon.ypos_text - 2, icon.width+5, icon.height)
125                 else:
126                     self.invert_region(icon.xpos_text - 2, icon.ypos_text - 2, icon.width, icon.height)
127 
128             else:
129                 self.SSD.text(icon.text, icon.xpos_text, icon.ypos_text, 1)  # Normal text color
130                 if icon.has_border:
131                     self.SSD.rect(icon.xpos_text - 2, icon.ypos_text - 2, icon.width, icon.height, 1)  # Normal border
132         self.SSD.show()
133 
```

```
134     class State:
135         def __init__(self):
136             global display
137             self.display = display
138         def B1Handler(pin):
139             pass
140         def ENCA(self, pin):
141             pass
142         # Interrupt handler for EncoderB pin (optional, if needed)
143         def ENCB(self, pin):
144             pass
145         def convert_to_12h(self,hour, minute):
146             if hour >= 0 and hour < 12:
147                 period = "AM"
148                 if hour == 0:
149                     hour = 12  # 0 AM is 12 AM in 12-hour format
150             else:
151                 period = "PM"
152                 if hour > 12:
153                     hour -= 12  # Convert PM hours to 12-hour format
154 
155             return f"{hour}:{minute:02d} {period}"
156 
157     def update(self):
158         pass
159     class ClockState(State):
160         def __init__(self):
161             global Alarm_s
162             super().__init__()  # Initialize superclass Display
163             #configure all icons
164             self.menu = Button("Menu",0,5,True,True)
165             self.menu.configureState(Menu_s)
166             self.format_time = "12h"
167             self.Alarm_on = Icon("Al: " +Alarm_s.is_on,2,0,False)
168             self.hour_adj = Button("Hr.",1,5,False,True)
169             self.min_adj = Button("Min",2,5,False,True)
```

```
159  class ClockState(State):
160      def __init__(self):
161          global Alarm_s
162          super().__init__()  # Initialize superclass Display
163          #configure all icons
164          self.menu = Button("Menu",0,5,True,True)
165          self.menu.configureState(Menu_s)
166          self.format_time = "12h"
167          self.Alarm_on = Icon("Al: " +Alarm_s.is_on,2,0,False)
168          self.hour_adj = Button("Hr.",1,5,False,True)
169          self.min_adj = Button("Min",2,5,False,True)
170          self.format_adj = Button(self.format_time,0,0,False,True)
171          self.zone = -7
172          self.zone_text = "UTC"+str(self.zone)
173          self.time_zone = Button(self.zone_text,1,0,False,True)
174          self.start_posx = 0
175          self.start_posy = 5
176          self.clock = Icon("", 1, 3, False)  # Placeholder for the clock icon
177          self.update_time()  # Initialize the clock display
178          self.icons = [self.clock, self.menu, self.hour_adj, self.min_adj,self.format_adj,self.time_zone,self.Alarm_on]
179          # Timer setup for minute update
180          self.timer = Timer()
181          self.timer.init(period=60000, mode=Timer.PERIODIC, callback=self.timer_callback)
182
183      #function to update the time displayed on screen
184      def update_time(self):
185          #read time from rtc
186          year, month, day, weekday, hours, minutes, seconds, subseconds = rtc.datetime()
187
188          if(self.format_time=="24h"):
189              clock_text = '{:02d}:{:02d}'.format(hours, minutes)
```

```
190             else:
191                 clock_text = self.convert_to_12h(hours,minutes)
192 
193             self.clock.text = clock_text
194 
195         def update(self):
196             self.menu.configureState(Menu_s)
197             self.Alarm_on.text = " Al:" +Alarm_s.is_on
198             display.update_buttons(self.icons)
199         def timer_callback(self, timer):
200             global current_state
201             self.update_time()
202             if(isinstance(current_state,ClockState)):
203                 display.render(self.icons)
204             else:
205                 pass
206 
207 
208         def B1Handler(self,pin):
209              global current_state, current_posx, current_posy
210              if debounce_handler(pin):
211                 if(current_posx <= 0 and current_posy ==5):
212                     current_posy=0
213                     current_posx = 1
214                 elif(current_posx<=0 and current_posy ==0):
215                     current_posy=5
216                     current_posx=2
217                 else:
218                     current_posx -= 1
219                 self.update()
220                 display.render(self.icons)
221 
```

```
222         def ENCA(self, pin):
223             global A_state, A_rising_edge, A_falling_edge, rotation_direction, radio, rtc,SNOOZE
224             # Read current state of EncoderA and EncoderB pins
225             A_state = EncoderA.value()
226             B_state = EncoderB.value()
227 
228             # Determine edge detection on EncoderA
229             if A_state == 1 and B_state == 0:
230                 A_rising_edge = True
231             elif A_state == 0 and B_state == 1:
232                 A_falling_edge = True
233 
234             # Check for both rising and falling edges on EncoderA
235             if A_rising_edge and A_falling_edge:
236                 year, month, day, weekday, hours, minutes, seconds, subseconds = rtc.datetime()
237                 if A_state != B_state:
238                     if self.hour_adj.selected:
239                         if hours+1<=23:
240                             rtc.datetime((year, month, day, weekday, (hours+1), minutes, seconds, subseconds))
241                             if(minutes + Alarm_s.snoozeLength >= 60):
242                                 hours+=1
243                                 minutes = (minutes + Alarm_s.snoozeLength) % 60
244                                 SNOOZE = [year, month, day, weekday, hours+1, minutes, seconds, subseconds]
245                             else:
246                                 SNOOZE = [year, month, day, weekday, hours+1, minutes+Alarm_s.snoozeLength, seconds, subseconds]
247                         else:
248                             pass
249                     if self.min_adj.selected:
250                         if minutes+1<=59:
251                             rtc.datetime((year, month, day, weekday, hours, (minutes+1), seconds, subseconds))
252                             if(minutes + Alarm_s.snoozeLength >= 60):
253                                 hours+=1
254                                 minutes = (minutes + Alarm_s.snoozeLength) % 60
255                                 SNOOZE = [year, month, day, weekday, hours, minutes+1, seconds, subseconds]
256                             else:
```

```
257                             SNOOZE = [year, month, day, weekday, hours, minutes+Alarm_s.snoozeLength+1, seconds, subseconds]
258                         else:
259                             pass
260                     if self.format_adj.selected:
261 
262                         if(self.format_time =="24h"):
263                             self.format_time = "12h"
264                         else:
265                             self.format_time = "24h"
266                         self.format_adj.text = self.format_time
267                     if self.time_zone.selected:
268                         if(self.zone<14):
269                             if(hours+1 > 23):
270                                 day+=1
271                                 weekday+=1
272                                 hours = -1
273                             self.zone+=1
274                             rtc.datetime((year, month, day, weekday, (hours+1), minutes, seconds, subseconds))
275                             if(minutes + Alarm_s.snoozeLength >= 60):
276                                hours+=1
277                                minutes = (minutes + Alarm_s.snoozeLength) % 60
278                                SNOOZE = [year, month, day, weekday, hours+1, minutes, seconds, subseconds]
279                             else:
280                                SNOOZE = [year, month, day, weekday, hours+1, minutes+Alarm_s.snoozeLength, seconds, subseconds]
281                             self.time_zone.text = "UTC"+str(self.zone)
282                 else:
283                     pass
284 
285                 # Reset edge detection flags
286                 A_rising_edge = False
287                 A_falling_edge = False
288                 self.update_time()
289                 display.render(self.icons)
290 
291         # Interrupt handler for EncoderB pin (optional, if needed)
292         def ENCB(self, pin):
293             global B_state, B_rising_edge, B_falling_edge, rotation_direction, radio, SNOOZE
```

```
294             # Read current state of EncoderA and EncoderB pins
295             A_state = EncoderA.value()
296             B_state = EncoderB.value()
297 
298             # Determine edge detection on EncoderB
299             if B_state == 1 and A_state == 0:
300                 B_rising_edge = True
301             elif B_state == 0 and A_state == 1:
302                 B_falling_edge = True
303 
304             # Check for both rising and falling edges on EncoderB
305             if B_rising_edge and B_falling_edge:
306                 year, month, day, weekday, hours, minutes, seconds, subseconds = rtc.datetime()
307                 if A_state == B_state:
308                     pass
309                 else:
310                     if self.hour_adj.selected:
311                         if hours-1>=0 :
312                             rtc.datetime((year, month, day, weekday, (hours-1), minutes, seconds, subseconds))
313                             if(minutes + Alarm_s.snoozeLength >= 60):
314                                hours+=1
315                                minutes = (minutes + Alarm_s.snoozeLength) % 60
316                                SNOOZE = [year, month, day, weekday, hours-1, minutes, seconds, subseconds]
317                             else:
318                                SNOOZE = [year, month, day, weekday, hours-1, minutes+Alarm_s.snoozeLength, seconds, subseconds]
319 
320                         else:
321                             pass
322 
323                     if self.min_adj.selected:
324                         year, month, day, weekday, hours, minutes, seconds, subseconds = rtc.datetime()
325                         if minutes -1>=0:
326                             rtc.datetime((year, month, day, weekday, hours, (minutes-1), seconds, subseconds))
```

```
327                         if(minutes + Alarm_s.snoozeLength >= 60):
328                             hours+=1
329                             minutes = (minutes + Alarm_s.snoozeLength) % 60
330                             SNOOZE = [year, month, day, weekday, hours, minutes-1, seconds, subseconds]
331                         else:
332                             SNOOZE = [year, month, day, weekday, hours, minutes+Alarm_s.snoozeLength-1, seconds, subseconds]
333                     else:
334                         pass
335                 if self.format_adj.selected:
336 
337                     if(self.format_time =="24h"):
338                         self.format_time = "12h"
339                     else:
340                         self.format_time = "24h"
341 
342                     self.format_adj.text = self.format_time
343                 if self.time_zone.selected:
344                     if(self.zone>-11):
345                         if(hours-1 < 0):
346                             hours=24
347                             weekday-=1
348                             day-=1
349                         self.zone-=1
350 
351                         rtc.datetime((year, month, day, weekday, (hours-1), minutes, seconds, subseconds))
352                         if(minutes + Alarm_s.snoozeLength >= 60):
353                             hours+=1
354                             minutes = (minutes + Alarm_s.snoozeLength) % 60
355                             SNOOZE = [year, month, day, weekday, hours-1, minutes, seconds, subseconds]
356                         else:
357                             SNOOZE = [year, month, day, weekday, hours-1, minutes+Alarm_s.snoozeLength, seconds, subseconds]
358                         self.time_zone.text = "UTC"+str(self.zone)
359 
```

```
361                     # Reset edge detection flags
362                 B_rising_edge = False
363                 B_falling_edge = False
364                 self.update_time()
365                 display.render(self.icons)
366 
367     class RadioState(State):
368         def __init__(self):
369             super().__init__()
370             global radio, Menu_s
371            #configure radio state icons
372             self.freq = 101.9
373             self.volume = 1
374             self.is_on = "N"
375             self.stationName = "CFUV"
376             self.radioOn = Button("On: " + self.is_on,1,0,False,True)
377             self.station = Icon(str(self.stationName),1,4,False)
378             self.frequency_text = Icon("Freq:",0,3,False)
379             self.volume_text = Icon("Volume:",0,2,False)
380             self.seek = Button("Seek",0,0,False,True)
381             self.frequ_disp = Button("  "+str(self.freq)+"FM", 1, 3,False, False)
382             self.menu = Button("Menu",0,5,True,True)
383             self.menu.configureState(Menu_s)
384             self.vol_adj = Button("Vol.",1,5,False,True)
385             self.vol_disp = Icon(str(self.volume),2,2,False)
386             self.freq_adj = Button("Freq.",2,5,False,True)
387             #initialize the list of icons to display
388             self.icons = [self.frequ_disp, self.menu, self.vol_adj, self.freq_adj,
389                           self.vol_disp,self.frequency_text, self.volume_text,self.radioOn, self.seek, self.station]
390             self.start_posx = 0
391             self.start_posy = 5
392 
393         def update(self):
394             self.menu.configureState(Menu_s)
395             display.update_buttons(self.icons)
396         #function to display the name of the station currently  playing
397         def setStationName(self):
398             if (floats_are_equal(self.freq , 101.9)):
399                 self.station.text = "CFUV"
```

```
400             elif(floats_are_equal(self.freq , 88.9)):
401                 self.station.text = "CBUX"
402             elif(floats_are_equal(self.freq , 90.5)):
403                 self.station.text = "CBCV"
404             elif(floats_are_equal(self.freq , 91.3)):
405                 self.station.text = "CJZN"
406             elif(floats_are_equal(self.freq , 92.1)):
407                 self.station.text = "CBU"
408             elif(floats_are_equal(self.freq , 98.5)):
409                 self.station.text = "CIOC"
410             elif(floats_are_equal(self.freq , 99.7)):
411                 self.station.text = "CBUF"
412             elif(floats_are_equal(self.freq , 100.3)):
413                 self.station.text = "CKKQ"
414             elif(floats_are_equal(self.freq , 103.1)):
415                 self.station.text = "CHTT"
416             elif(floats_are_equal(self.freq , 106.5)):
417                 self.station.text = "KWPZ"
418             elif(floats_are_equal(self.freq , 107.3)):
419                 self.station.text = "CHBE"
420             elif(floats_are_equal(self.freq , 107.9)):
421                 self.station.text = "CILS"
422             elif(floats_are_equal(self.freq , 104.1)):
423                 self.station.text = "KAFE"
424             else:
425                 self.station.text = ""
426 
427 
428         def ENCA(self,pin):
429             global A_state, A_rising_edge, A_falling_edge, rotation_direction, radio
430 
431             # Read current state of EncoderA and EncoderB pins
432             A_state = EncoderA.value()
433             B_state = EncoderB.value()
```

```
435             # Determine edge detection on EncoderA
436             if A_state == 1 and B_state == 0:
437                 A_rising_edge = True
438             elif A_state == 0 and B_state == 1:
439                 A_falling_edge = True
440 
441             # Check for both rising and falling edges on EncoderA
442             if A_rising_edge and A_falling_edge:
443                 if A_state != B_state:
444                     #adjust volume
445                     if(self.vol_adj.selected):
446                         if(self.volume+1 <=5):
447                             radio.set_volume(self.volume+1 )
448                             radio.update_rds()
449                             self.volume+=1
450                             self.vol_disp.text = str(self.volume)
451                     #turn radio on and off
452                     if(self.radioOn.selected):
453                         if(self.is_on == "N"):
454                             self.is_on = "Y"
455                             self.radioOn.text = "On: " + self.is_on
456                             radio.mute(False)
457                             radio.update_rds()
458                         else:
459                             self.is_on = "N"
460                             self.radioOn.text = "On: " + self.is_on
461                             radio.mute(True)
462                             radio.update_rds()
463                      #adjust frequency
464                      if(self.freq_adj.selected):
465 
466                        if(self.freq+0.1 <= 108):
467                             radio.set_frequency_MHz(self.freq +0.1)
468                             self.freq +=0.1
469                             radio.update_rds()
470                             # Update the frequency displayed on the icon
471                             self.frequ_disp.text = "  "+f"{self.freq:.1f}"+"FM"
472                             self.setStationName()
473                     #seek up
```

```
474                     if(self.seek.selected):
475                         radio.seek_up()
476                         self.freq = radio.get_frequency_MHz()
477                         self.frequ_disp.text = "  "+f"{self.freq:.1f}"+"FM"
478                         self.setStationName()
479                     #render updated information to display
480                     display.render(self.icons)
481                 else:
482                    pass
483 
484                 # Reset edge detection flags
485                 A_rising_edge = False
486                 A_falling_edge = False
487 
488         # Interrupt handler for EncoderB pin (optional, if needed)
489         def ENCB(self,pin):
490             global B_state, B_rising_edge, B_falling_edge, rotation_direction, radio
491             # Read current state of EncoderA and EncoderB pins
492             A_state = EncoderA.value()
493             B_state = EncoderB.value()
494 
495             # Determine edge detection on EncoderB
496             if B_state == 1 and A_state == 0:
497                 B_rising_edge = True
498             elif B_state == 0 and A_state == 1:
499                 B_falling_edge = True
500 
501             # Check for both rising and falling edges on EncoderB
502             if B_rising_edge and B_falling_edge:
503                 if A_state == B_state:
504                     pass
505 
506                 else:
507                     if(self.freq_adj.selected):
508                         if(self.freq-0.1 >= 88):
509                             radio.set_frequency_MHz(self.freq-0.1)
510                             self.freq-=0.1
511                             radio.update_rds()
512                             # Update the frequency displayed on the icon
513                             self.frequ_disp.text = "  "+f"{self.freq:.1f}"+"FM"
```

```
516                     if(self.vol_adj.selected):
517                         if(self.volume-1 >=0):
518                             radio.set_volume( self.volume-1 )
519                             radio.update_rds()
520                             self.volume-=1
521                             self.vol_disp.text = str(self.volume)
522                     if(self.radioOn.selected):
523                         if(self.is_on == "N"):
524                             self.is_on = "Y"
525                             self.radioOn.text = "On: " + self.is_on
526                             radio.mute(False)
527                             radio.update_rds()
528                         else:
529                             self.is_on = "N"
530                             self.radioOn.text = "On: " + self.is_on
531                             radio.mute(True)
532                             radio.update_rds()
533                     if(self.seek.selected):
534                         radio.seek_down()
535                         self.freq = radio.get_frequency_MHz()
536                         self.frequ_disp.text = "  "+f"{self.freq:.1f}"+"FM"
537                         self.setStationName()
538 
539                     display.render(self.icons)
540 
541                 # Reset edge detection flags
542                 B_rising_edge = False
543                 B_falling_edge = False
544 
545         def B1Handler(self,pin):
546             global current_posx, current_posy
547             if debounce_handler(pin):
548                 current_posx -=1
549                 if(current_posx<0 and current_posy != 0):
550                     current_posx = 1
551                     current_posy = 0
552                 if(current_posx==-1 and current_posy ==0):
553                     current_posx = 2
```

```
556                 self.update()
557                 display.render(self.icons)
558 
559         def B2Handler(self,pin):
560             pass
561 
562 
563     class AlarmState(State):
564         def __init__(self):
565             super().__init__()
566             global rtc
567             self.alarmOn = Button("Alarm",0,4,False,True)
568             self.snooze_adj = Button("Sleep",0,3,False,True)
569             self.song_adj = Button("Song",0,2,False,True)
570             self.volume_adj = Button("Vol:",0,1,False,True)
571             self.volume = 3
572             self.start_posx = 0
573             self.start_posy = 5
574             self.snoozeLength = 1
575             self.alarm_hour = 0
576             self.alarm_minute = 0
577             self.song_id = 1
578             str_num = '{:02d}'.format(self.alarm_minute)
579             self.volume_disp = Icon(" "+str(self.volume),1,1,False)
580             self.alarm_disp = Icon(" "+str(self.alarm_hour)+":"+str_num,1,4,False)
581             self.snooze_disp = Icon(" "+str(self.snoozeLength) + " Mins",1,3,False)
582             self.song_disp = Icon(" " + str(self.song_id),1,2,False)
583             self.hour_adj = Button("Hr.",1,5,False,True)
584             self.minute_adj = Button("Min.",2,5,False,True)
585             self.menu = Button("Menu",0,5,True,True)
586             self.menu.configureState(Menu_s)
587             self.is_on = "N"
588             self.alarm_on = Icon("On: " + self.is_on,2,0,False)
589             self.icons = [self.alarm_on,self.volume_disp,self.alarm_disp,self.snooze_disp,self.hour_adj,
590                           self.minute_adj,self.menu,self.alarmOn,self.song_adj,self.song_disp,self.snooze_adj, self.volume_adj]
591 
592         def update(self):
593             self.menu.configureState(Menu_s)
```

```
            display.update_buttons(self.icons)

        def ENCA(self,pin):
            global A_state, A_rising_edge, A_falling_edge, rotation_direction, radio, mySong

            # Read current state of EncoderA and EncoderB pins
            A_state = EncoderA.value()
            B_state = EncoderB.value()

            # Determine edge detection on EncoderA
            if A_state == 1 and B_state == 0:
                A_rising_edge = True
            elif A_state == 0 and B_state == 1:
                A_falling_edge = True

            # Check for both rising and falling edges on EncoderA
            if A_rising_edge and A_falling_edge:
                if A_state != B_state:
                    if(self.song_adj.selected):
                        if(self.song_id+1 <= 3):
                            self.song_id +=1
                            # Update the frequency displayed on the icon
                            mySong = music(songs[self.song_id-1], pins=[Pin(26)])
                            self.song_disp.text = " " + str(self.song_id)
                    if(self.snooze_adj.selected):
                        if (self.snoozeLength+1 < 60 ):
                            self.snoozeLength+=1
                            self.snooze_disp.text = " "+str(self.snoozeLength) + " Mins"
                    if self.hour_adj.selected:
                        if self.alarm_hour+1<=23:
                            self.alarm_hour+=1
                            str_num = '{:02d}'.format(self.alarm_minute)
                            if(Clock_s.format_time=="24h"):
                                self.alarm_disp.text = " "+str(self.alarm_hour)+":"+str_num
                            else:
                                self.alarm_disp.text = " "+self.convert_to_12h(self.alarm_hour,self.alarm_minute)
                    if self.minute_adj.selected:
                        if self.alarm_minute+1<=59:
                            self.alarm_minute+=1
                            str_num = '{:02d}'.format(self.alarm_minute)
```

```
634                             if(Clock_s.format_time=="24h"):
635                                 self.alarm_disp.text = " "+str(self.alarm_hour)+":"+str_num
636                             else:
637                                 self.alarm_disp.text = " "+self.convert_to_12h(self.alarm_hour,self.alarm_minute)
638                     if(self.alarmOn.selected):
639                         if(self.is_on == "N"):
640                             self.is_on = "Y"
641                             self.alarm_on.text = "On: " + self.is_on
642                         else:
643                             self.is_on = "N"
644                             self.alarm_on.text = "On: " + self.is_on
645                     if self.volume_adj.selected:
646                         if (self.volume + 1) <= 5:
647                             self.volume += 1
648                             self.volume_disp.text = " "+str(self.volume)
649                             mySong.duty = self.volume*2000 + 100
650 
651                     display.render(self.icons)
652                 else:
653                     pass
654 
655                 # Reset edge detection flags
656                 A_rising_edge = False
657                 A_falling_edge = False
658         # Interrupt handler for EncoderB pin (optional, if needed)
659         def ENCB(self,pin):
660             global B_state, B_rising_edge, B_falling_edge, rotation_direction, mySong
661             # Read current state of EncoderA and EncoderB pins
662             A_state = EncoderA.value()
663             B_state = EncoderB.value()
664             # Determine edge detection on EncoderB
665             if B_state == 1 and A_state == 0:
666                 B_rising_edge = True
667             elif B_state == 0 and A_state == 1:
668                 B_falling_edge = True
669             # Check for both rising and falling edges on EncoderB
670             if B_rising_edge and B_falling_edge:
671                 if A_state == B_state:
672                     pass
673                 else:
```

```
674                     #DECREASE LOGIC
675                     if(self.song_adj.selected):
676                         if(self.song_id-1 >= 1):
677                             self.song_id -=1
678                             self.song_disp.text = " " + str(self.song_id)
679                             mySong = music(songs[self.song_id-1], pins=[Pin(26)])
680                         # Update the frequency displayed on the icon
681                     if(self.snooze_adj.selected):
682                         if (self.snoozeLength-1 >0 ):
683                             self.snoozeLength-=1
684                             self.snooze_disp.text = " "+str(self.snoozeLength) + " Mins"
685                     if self.hour_adj.selected:
686                         if self.alarm_hour-1>=0:
687                             self.alarm_hour-=1
688                             str_num = '{:02d}'.format(self.alarm_minute)
689                             if(Clock_s.format_time=="24h"):
690                                 self.alarm_disp.text = " "+str(self.alarm_hour)+":"+str_num
691                             else:
692                                 self.alarm_disp.text = " "+self.convert_to_12h(self.alarm_hour,self.alarm_minute)
693                         else:
694                             pass
695                     if self.minute_adj.selected:
696                         if self.alarm_minute-1>=0:
697                             self.alarm_minute-=1
698                             str_num = '{:02d}'.format(self.alarm_minute)
699                             if(Clock_s.format_time=="24h"):
700                                 self.alarm_disp.text = " "+str(self.alarm_hour)+":"+str_num
701                             else:
702                                 self.alarm_disp.text = " "+self.convert_to_12h(self.alarm_hour,self.alarm_minute)
703                         else:
704                             pass
705                     if(self.alarmOn.selected):
706                         if(self.is_on == "N"):
707                             self.is_on = "Y"
708                             self.alarm_on.text = "On: " + self.is_on
709                         else:
710                             self.is_on = "N"
711                             self.alarm_on.text = "On: " + self.is_on
712                     if(self.volume_adj.selected):
713                         if(self.volume - 1 >= 0):
```

```
714                             self.volume -= 1
715                             self.volume_disp.text = " "+str(self.volume)
716                             mySong.duty = self.volume*2000 + 100
717
718                     display.render(self.icons)
719                     pass
720                 # Reset edge detection flags (reset finite state machine)
721                 B_rising_edge = False
722                 B_falling_edge = False
723
724         def B1Handler(self,pin):
725             global current_state, current_posx, current_posy
726             if debounce_handler(pin):
727                 if(current_posx <= 0):
728                     current_posy-=1
729                     current_posx=0
730                 if(current_posy==0 and current_posx<=0):
731                     current_posx=3
732                     current_posy=5
733                 if(current_posx!=0):
734                     current_posx -= 1
735                 self.update()
736                 display.render(self.icons)
737
738         def B2Handler(self,pin):
739             pass
740
741     class PlayALARM(State):
742         def __init__(self):
743             super().__init__()
744             self.ALARM = Icon("ALARM", 1, 2,True)
745             self.Instructions1 = Icon("SNOOZE: CCW", 0,5,False)
746             self.Instructions2 = Icon("OFF: CW", 0,0,False)
747             self.icons = [self.ALARM, self.Instructions1, self.Instructions2]
748             self.start_posx = 1
749             self.start_posy = 0
750         def update(self):
751             display.update_buttons(self.icons)
752         def B1Handler(self,pin):
753             pass
```

```
754         def B2Handler(self,pin):
755             pass
756         def ENCA(self,pin):
757             global A_state, A_rising_edge, A_falling_edge, rotation_direction,current_state, Clock_s,SNOOZE, rtc, mySon
758 
759             # Read current state of EncoderA and EncoderB pins
760             A_state = EncoderA.value()
761             B_state = EncoderB.value()
762 
763             # Determine edge detection on EncoderA
764             if A_state == 1 and B_state == 0:
765                 A_rising_edge = True
766             elif A_state == 0 and B_state == 1:
767                 A_falling_edge = True
768 
769             # Check for both rising and falling edges on EncoderA
770             if A_rising_edge and A_falling_edge:
771                 if A_state != B_state:
772                    change_state(Clock_s)
773                    year, month, day, weekday, hours, minutes, seconds, subseconds = rtc.datetime()
774                    if(minutes + Alarm_s.snoozeLength >= 60):
775                        hours+=1
776                        minutes = (minutes + Alarm_s.snoozeLength) % 60
777                        SNOOZE = [year, month, day, weekday, hours, minutes, seconds, subseconds]
778                    else:
779                        SNOOZE = [year, month, day, weekday, hours, minutes+Alarm_s.snoozeLength, seconds, subseconds]
780 
781                 else:
782                    pass
783                 mySong.stop()
784                 # Reset edge detection flags
785                 A_rising_edge = False
786                 A_falling_edge = False
787 
788         # Interrupt handler for EncoderB pin (optional, if needed)
789         def ENCB(self,pin):
790             global B_state, B_rising_edge, B_falling_edge, rotation_direction, current_state,SNOOZE, mySong
791             # Read current state of EncoderA and EncoderB pins
792             A_state = EncoderA.value()
793             B_state = EncoderB.value()
```

```
794             # Determine edge detection on EncoderB
795             if B_state == 1 and A_state == 0:
796                 B_rising_edge = True
797             elif B_state == 0 and A_state == 1:
798                 B_falling_edge = True
799             # Check for both rising and falling edges on EncoderB
800             if B_rising_edge and B_falling_edge:
801                 if A_state == B_state:
802                     pass
803                 else:
804                     #DECREASE LOGIC
805                     change_state(Clock_s)
806                     SNOOZE = [0,0,0,0,0,0,0,0]
807                 mySong.stop()
808                 # Reset edge detection flags (reset finite state machine)
809                 B_rising_edge = False
810                 B_falling_edge = False
811 
812 
813     class MainMenuState(State):
814         def __init__(self):
815             super().__init__()
816             global icons
817             self.clock_but = Button("CLOCK", 1, 0, False, True)
818             self.radio_but = Button("RADIO", 1, 2, False, True)
819             self.alarm_but = Button("ALARM", 1, 4, True, True)
820             self.clock_but.configureState(Clock_s)
821             self.radio_but.configureState(Radio_s)
822             self.alarm_but.configureState(Alarm_s)
823             self.icons = [self.radio_but, self.alarm_but, self.clock_but]
824             self.start_posx = 1
825             self.start_posy = 4
826             display.render(self.icons)
827         def update(self):
828             display.update_buttons(self.icons)
829         def B1Handler(self,pin):
830              global current_state, current_posy
831              if debounce_handler(pin):
832                 current_posy -= 2
833                 if(current_posy<0):
```

```
834                     current_posy = 4
835                 self.update()
836                 display.render(self.icons)
837 
838         def ENCA(self,pin):
839             pass
840 
841         # Interrupt handler for EncoderB pin (optional, if needed)
842         def ENCB(self,pin):
843             pass
844 
845     def debounce_handler(pin):
846         global last_pressed_time
847         current_time = utime.ticks_ms()
848         if current_time - last_pressed_time < debounce_delay:
849             return False
850         last_pressed_time = current_time
851         return pin.value() == 0  # Check if button is pressed
852 
853     #interrupt handler for the encoder button
854     def Enter_Handler(pin):
855         global ENTER, current_state
856         if debounce_handler(pin):
857             ENTER = True
858 
859     def change_state(state):
860         global current_state, current_posx, current_posy
861         current_state = state
862         current_posx = current_state.start_posx
863         current_posy = current_state.start_posy
864         EncoderA.irq(trigger=machine.Pin.IRQ_RISING | machine.Pin.IRQ_FALLING, handler=current_state.ENCA)
865         EncoderB.irq(trigger=machine.Pin.IRQ_RISING | machine.Pin.IRQ_FALLING, handler=current_state.ENCB)
866         button_1.irq(handler=current_state.B1Handler, trigger=Pin.IRQ_FALLING)
867         current_state.update()
868         display.render(current_state.icons)
869 
870     class ClockRadio:
871         def __init__(self):
872             global Radio_s, current_state
873             button_1.irq(handler=current_state.B1Handler, trigger=Pin.IRQ_FALLING)
```

```
874             EncoderA.irq(trigger=machine.Pin.IRQ_RISING | machine.Pin.IRQ_FALLING, handler=current_state.ENCA)
875             EncoderB.irq(trigger=machine.Pin.IRQ_RISING | machine.Pin.IRQ_FALLING, handler=current_state.ENCB)
876             enter.irq(handler=Enter_Handler, trigger=Pin.IRQ_FALLING)
877
878         def update(self, state):
879             global radio_programming_timer
880             state.update()
881             pass
882
883     #initial display object
884     display = Display(128,64)
885
886
887     #initialize instance of radio class, configure  I2C communication
888     radio_i2c = I2C(0, sda=Pin(0), scl = Pin(1), freq=100000) # What frequency should we use?
889     radio = rda5807.Radio(radio_i2c)
890     utime.sleep(1)
891     radio.set_frequency_MHz(101.9)
892     radio.set_volume(3)
893     radio.mute(True)
894     radio.update_rds()
895
896     #declare instances of all state classes (used to store permanent data throughout program operation
897     #instead of declaring new classes instances for each state change and losing stored data)
898     Menu_s = None
899     Clock_s = State()
900     Alarm_s = AlarmState()
901     Clock_s = ClockState()
902     Radio_s = RadioState()
903     Menu_s = MainMenuState()
904     #define states used by the clock radio before the clock radio
905     current_state = Menu_s
906     clock_radio = ClockRadio()
907     Playalarm_s = PlayALARM()
```

```
909     #the three song options for the alarm
910     #    https://onlinesequencer.net/195547
911     song1 = '0 A#4 1 1;2 F5 1 1;4 D#5 1 1;8 D5 1 1;11 D5 1 1;6 A#4 1 1;14 D#5 1 1;18 A#4 1 1;20 D#5 1 1;22 A#4 1 1;24
912     #    https://onlinesequencer.net/1864273
913     song2 = '0 D5 4 14;4 A5 4 14;8 C6 4 14;12 B5 4 14;16 G5 2 14;18 F5 2 14;20 E5 2 14;22 F5 2 14;24 G5 8 14;4 E5 8 1
914     #    https://onlinesequencer.net/1864297 - Tetris
915     song3 = '0 E3 1 0;2 E4 1 0;4 E3 1 0;6 E4 1 0;8 E3 1 0;10 E4 1 0;12 E3 1 0;14 E4 1 0;16 A3 1 0;18 A4 1 0;20 A3 1 0
916     songs = [song1, song2, song3]
917     mySong = music(songs[Alarm_s.song_id-1], pins=[Pin(26)])
918     #function to check if the alarm should be played
919     def check_for_alarm():
920         global current_state, current_posx, current_posy, Playalarm_s
921         #condition checks if alarm time matches current time, or snooze time matches current time
922         if((Alarm_s.alarm_hour==rtc.datetime()[4] and Alarm_s.alarm_minute == rtc.datetime()[5] and
923             rtc.datetime()[6]==0  and Alarm_s.is_on=="Y") or (SNOOZE[0] == rtc.datetime()[0] and
924             SNOOZE[1] == rtc.datetime()[1] and SNOOZE[2] == rtc.datetime()[2] and SNOOZE[3] == rtc.datetime()[3] and
925             SNOOZE[4] == rtc.datetime()[4] and  SNOOZE[5] == rtc.datetime()[5] and SNOOZE[6] == rtc.datetime()[6])):
926            change_state(Playalarm_s)
927            mySong.resume()
928     while True:
929         clock_radio.update(current_state)
930         check_for_alarm()
931         #play the alarm
932         if(isinstance(current_state,PlayALARM)):
933             radio.mute(True)
934             radio.update_rds()
935             mySong.tick()
936             sleep(0.04)
937             if(Radio_s.is_on =="Y"):
938                 radio.mute(False)
939                 radio.update_rds()
940
```