University of Victoria

PulseEdge Clock-Radio
ECE 299 Final Report

By PulseEdge Embedded Systems, a division of Emelu Technologies

David Emelu
V01025755

Kampamba Loongo
V01005806

Professor: Ilamparithi Thirumarai Chelvan
University of Victoria

Electrical and Computer Engineering
ECE299 Introduction to ECE Design
Section B02 (Group 3)

August 2025

Abstract

This report presents the design, development, and testing of an animal-themed clock-radio system developed by Pulse Edge Embedded Systems. The objective was to integrate core timekeeping and FM radio functionalities into a cohesive, user-friendly device housed within a custom-designed enclosure.

The system was designed to perform essential clock operations, including setting and editing time and alarms, toggling between 12-hour and 24-hour formats, and executing snooze and alarm-off actions. Additionally, the prototype allowed users to tune to FM channels, adjust volume, and view channel information through a simplified pushbutton interface.

A Raspberry Pi Pico microcontroller served as the system's core, supported by a DS3231 RTC for accurate timekeeping, an RDA5807M FM tuner for radio reception, and a 128×64 OLED display for output. Audio playback was handled by a compact speaker, with volume controlled via a potentiometer and amplified using a PAM8403 module. Initial testing was conducted on a breadboard before the circuit was migrated to a two-layer printed circuit board (PCB) designed in KiCad and manufactured commercially. All components were manually soldered, and the device was enclosed in an animal-themed casing modeled in SolidWorks to ensure mechanical compatibility and aesthetic appeal.

The final prototype successfully met all design requirements. This project provided valuable hands-on experience in embedded systems integration, PCB design and assembly, mechanical design using CAD software, and firmware development with MicroPython. Throughout the project, all actions were conducted in alignment with the professional standards outlined in the Engineers and Geoscientists BC (EGBC) Code of Ethics.

Contents

List of Tables

List of Figures

1

Project Goals

The goal of this project was to design and build a fully functional animal-themed clock-radio that combines accurate timekeeping with FM radio features in a user-friendly and visually appealing format. The device was intended to provide a hands-on opportunity to apply embedded systems knowledge through circuit design, programming, and mechanical enclosure development.

2

List of Constraints

The clock-radio prototype was developed under the following constraints, which guided both the design and implementation phases:

- A **single Raspberry Pi Pico W** was required to be used as the microcontroller responsible for all processing, control, and decision-making tasks.
- The **display module** was required to communicate with the Raspberry Pi Pico W using the **SPI protocol** and display both clock and radio information.
- All programming and firmware development were to be conducted using **MicroPython**.
- The printed circuit board (**PCB**) was to be **custom-designed and manufactured** by a commercial vendor; off-the-shelf development boards were not permitted.
- All team members were required to be from the **same lab section** to ensure coordination and consistent access to lab sessions.
- The **final demonstration** of the working prototype was to occur **during the week of July 29**.
- The project was expected to **incur costs** for materials, manufacturing, and other resources, which were to be managed by the team.

3

Project Expectations

1. A prototype animal-themed clock-radio was to be designed and developed, incorporating the following core functionalities:

   Clock Functionalities

   - The system was expected to allow users to set and edit the current time through a user interface.
   - The system was expected to allow users to set and edit the alarm time through a user interface.
   - The time was to be displayed in both 12-hour and 24-hour formats, selectable by the user.
   - The alarm was to be triggered at the set time.
   - The user was to be able to snooze or turn off the alarm via the interface.

   Radio Functionalities

   - The system was required to tune to at least one FM radio channel through a user interface.
   - Radio audio was to be played through a speaker.
   - Volume control was to be available to the user through a user interface.
   - Radio channel information was to be displayed on the screen.

2. A custom two-layer PCB was to be designed using KiCad and manufactured by a commercial PCB vendor.
3. All electronic components were to be soldered onto the manufactured PCB by the project team.
4. An animal-themed enclosure was to be designed using SolidWorks and fabricated to house the final assembled system.
5. The completed device, including its enclosure, was to be demonstrated during the final week of labs.
6. A formal technical report documenting the entire design process was to be submitted by the project deadline.

4

Design Choices

This section outlines the component selection and engineering decisions made to meet the project expectations, with justifications supported by component datasheets and reliable technical literature.

4.1 Clock Functionality (Timekeeping, Alarm, Display Format, Snooze/Off)

To ensure accurate timekeeping, especially during power loss, the **DS3231** Real-Time Clock (RTC) module was selected. It features a built-in **temperature-compensated crystal oscillator (TCXO)** and provides ±2 ppm accuracy, making it ideal for embedded systems requiring minimal drift [2]. Communication with the Raspberry Pi Pico W was achieved via I^2C, and backup power support allowed time retention during outages.

User input for alarm setting, snooze, and time adjustments was handled using **four pushbuttons** connected to GPIO pins. This method ensured reliable mechanical input without adding complexity to the system. The clock display supported both **12-hour and 24-hour formats**, with switching handled via software logic.

4.2 Display of Clock and Radio Information

A **2.42-inch 128×64 OLED display**, based on the **SSD1309 controller**, was selected for its **high contrast ratio**, **wider screen area**, and **low power consumption**. The larger display size (compared to standard 0.96-inch modules) improved visibility of time and FM frequency information. Communication was implemented using the **SPI protocol**, which provided faster data transfer rates while minimizing GPIO pin usage compared to parallel interfaces.

The SSD1309 driver's compatibility with MicroPython libraries and reliable monochrome output made it ideal for embedded clock-radio applications. Its performance characteristics are documented in the manufacturer's technical specifications [3].

4.3 FM Radio Functionality

The **RDA5807M** FM radio module was chosen due to its **digital tuning**, **stereo output**, and I^2C communication. It supports the global FM band (50–108 MHz), making it suitable for use in most regions. The module's automatic gain control and internal DSP ensured stable performance and good signal-to-noise ratio [4]. Its compact footprint made it ideal for integration onto a small custom PCB.

4.4 Audio Playback and Volume Control

To amplify the audio signal, the **PAM8403** stereo Class-D audio amplifier was used. The module delivers up to **3 W/channel** into 4 Ω speakers with high efficiency (>90%) and low total harmonic distortion (THD+N < 0.1%) [1]. It also included an **onboard potentiometer** for analog volume control, allowing intuitive manual adjustment without additional software complexity.

The amplifier output was connected to a 4 Ω speaker to meet the project's sound output requirement while maintaining portability.

4.5 PCB Design

The system was implemented on a **two-layer PCB** designed using **KiCad** and fabricated by a commercial manufacturer. A dual-layer board allowed clean routing of power, ground, and signal traces, reducing electromagnetic interference and simplifying layout. Trace widths and via sizes were calculated based on current requirements, referencing IPC-2221 standards. Silk screen labels were added to aid assembly.

4.6 Enclosure Design

The enclosure was modeled in **SolidWorks** and 3D printed to resemble an **elephant**, meeting the animal-themed design requirement. This form factor was chosen for its symbolic representation and visual appeal. The casing was designed to house the display, speakers, and buttons securely while aligning openings for visibility and accessibility.

5

Project Planning

The development of the clock-radio system was carried out following a structured plan, where responsibilities were distributed between the two team members. The project plan was submitted as a milestone activity early in the term, outlining detailed tasks, test criteria, and expected completion dates. The project was executed closely in alignment with this plan, with the exception of one modification related to the power system.

Table 5.1: *Project Planning Tasks by Date and Team Member*

Date	Team Member	Task
June 12	**David Emelu**	Wiring and testing the RTC and FM radio modules via I^2C.
June 12	**Kampamba Loongo**	Setting up pushbutton inputs and validating encoder input for early interface testing.
June 16	**David Emelu**	OLED display integration and UI layout verification.
June 20	**David Emelu**	Designing the menu and alarm logic in MicroPython.
June 20	**David Emelu**	[*Deviation*] Changed power system from battery-powered to USB-only.
June 20	**Kampamba Loongo**	Testing speaker output via the LM386 (later replaced with PAM8403 module).
July 10	**Kampamba Loongo**	Implementing FM tuning and volume control logic.
July 12–13	**David Emelu**	Designing the KiCad schematic and PCB layout, followed by PCB fabrication.
July 14	**Kampamba Loongo**	Creating a custom SolidWorks model of the elephant-themed enclosure.
July 18	**Kampamba Loongo**	Developing and rendering OLED-based radio and menu displays.
July 21	**David Emelu**	Developed web UI interface to replicate all hardware functions (time/alarm setting, FM tuning, volume, display format, timezone).
July 19–24	**Kampamba Loongo**	3D printing and assembling the enclosure with internal components.
July 23	**David Emelu**	Soldering and testing the custom PCB.

Deviation from the Plan

The only deviation from the original milestone schedule was the removal of the custom power system. While the initial plan included testing battery charging and regulation modules (e.g., USB-C and LiPo handling), these components were excluded in the final build. The system was instead powered directly via USB, which proved sufficient for stable performance throughout testing and demonstration.

This decision allowed the team to reduce complexity and focus on refining the core functionalities of the clock and radio system.

6

Circuit Design and Schematic

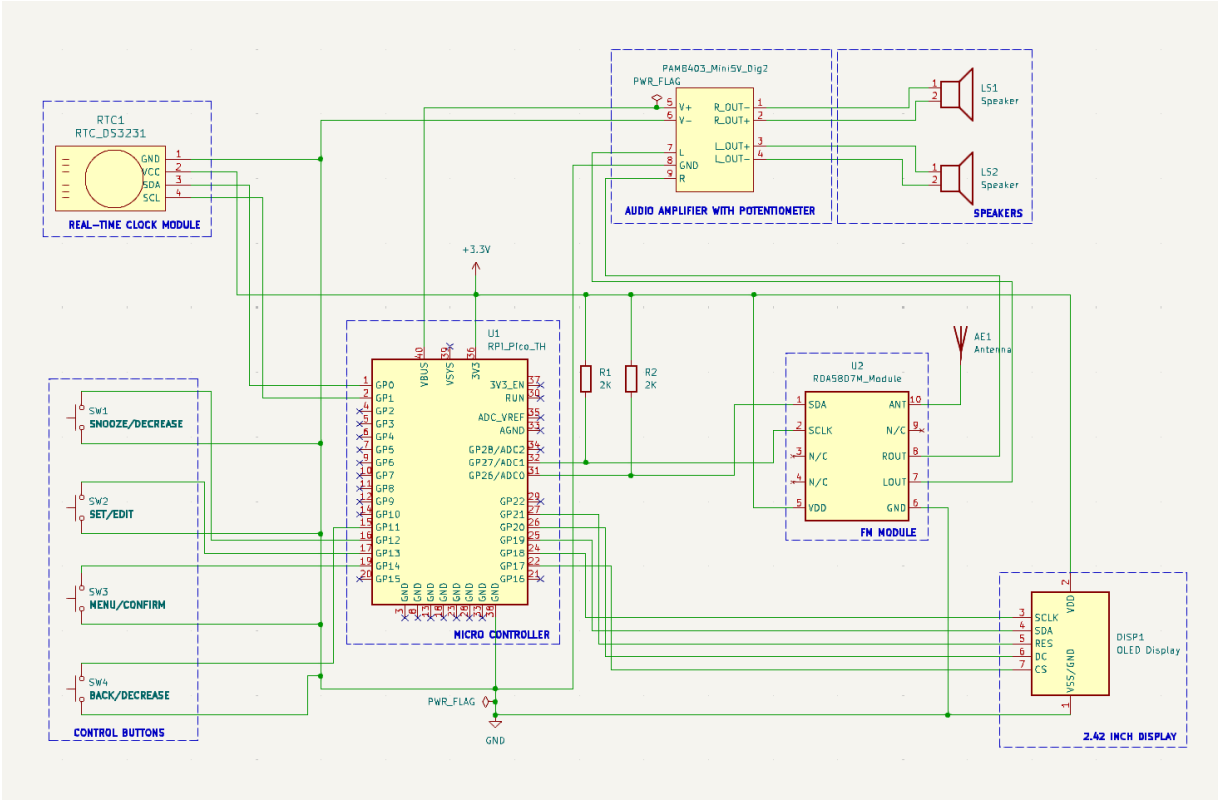

Figure 6.1: *The final schematic of the manufactured PCB*

The schematic diagram shown in Figure 6.1 served as the foundation for the design and implementation of the clock-radio system. It was developed collaboratively and refined through multiple iterations to ensure proper power distribution, and compatibility among all modules.

At the core of the system was the Raspberry Pi Pico W, which functioned as the central microcontroller responsible for managing user input, real-time clock synchronization, FM radio tuning, audio playback control, and display updates. The Pico was powered via its VBUS pin from a USB source, and its onboard regulator provided a steady 3.3 V

rail (VSYS) for all external components. GPIO pins were allocated to each peripheral, and ground connections were routed to minimize noise and ensure signal integrity.

Accurate timekeeping was maintained by a DS3231 RTC module connected over the I^2C bus (SDA, SCL). The module's temperature-compensated crystal oscillator and backup battery allowed continuous clock tracking, even during power interruptions.

FM radio functionality was provided by the RDA5807M module, also on the I^2C interface. Its antenna input was routed to an external wire antenna for improved reception, and its stereo outputs (LOUT, ROUT) fed the audio amplifier stage.

Audio amplification was handled by a PAM8403 stereo Class-D amplifier. The FM module's left and right outputs were connected to the amplifier's inputs, and a built-in potentiometer afforded manual volume control. The amplifier then drove a $4\,\Omega$ speaker.

Visual feedback was delivered via a 2.42-inch OLED (128×64, SSD1309 driver) over the SPI bus (SCLK, SDA, RES, DC, CS). This display presented the current time, FM frequency, alarm status, and menu options with high contrast and legibility.

User interaction was enabled through four tactile pushbuttons on dedicated GPIO lines. These buttons allowed menu navigation, time and alarm setting, snooze activation, and format toggling.

Pull-up resistors were added on I^2C lines to meet communication standards. All components shared the common 3.3 V rail and ground plane. The schematic was verified against datasheet specifications, and design-rule checks (DRC) in KiCad passed with no errors prior to PCB layout.

7

Bill of Materials and Cost Analysis

Table 7.1: *Bill of Materials and Cost Analysis*

Material	Description	Part #	Cost (CAD)	Source
Raspberry Pi Pico W	A low-cost, high-performance microcontroller board with flexible digital interfaces.	SC0918	$10.00	School Laboratory
DS3231 RTC Module	Highly accurate real-time clock module with I^2C interface and backup battery.	DS3231SN	$4.00	Amazon
2.42-in. OLED Display	Monochrome 128×64 display (SSD1309 driver) with SPI interface and wide viewing angle.	SSD1309	$13.00	School Laboratory
RDA5807M FM Module	Single-chip FM tuner with digital synthesis and I^2C control.	RDA5807M	$2.50	AliExpress
PAM8403 Audio Amplifier	3 W/ch Class-D amplifier with onboard potentiometer for volume control.	PAM8403	$3.00	Amazon
3 W, 4 Ω Mini Speakers	Compact speakers suitable for embedded audio output.	–	$1.50	Amazon
Tactile Pushbuttons	Standard momentary pushbuttons for user input.	PBS-110	$0.25	School Laboratory
FM Wire Antenna	Simple wire antenna for FM reception.	–	$0.50	Reused / Lab Stock
2 kΩ Resistors	Through-hole resistors for pull-ups and decoupling.	CF14JT2K00	$0.10	Digi-Key
Custom 2-layer PCB	Professionally fabricated PCB for the complete system.	–	$12.00	JLCPCB
3D Printed Enclosure	PLA-based elephant-themed enclosure printed in-house.	–	$5.00	In-house / PLA Filament
Miscellaneous	Connectors, headers, and wiring for interconnections.	–	$3.00	General Lab Use

8

Printed Circuit Board

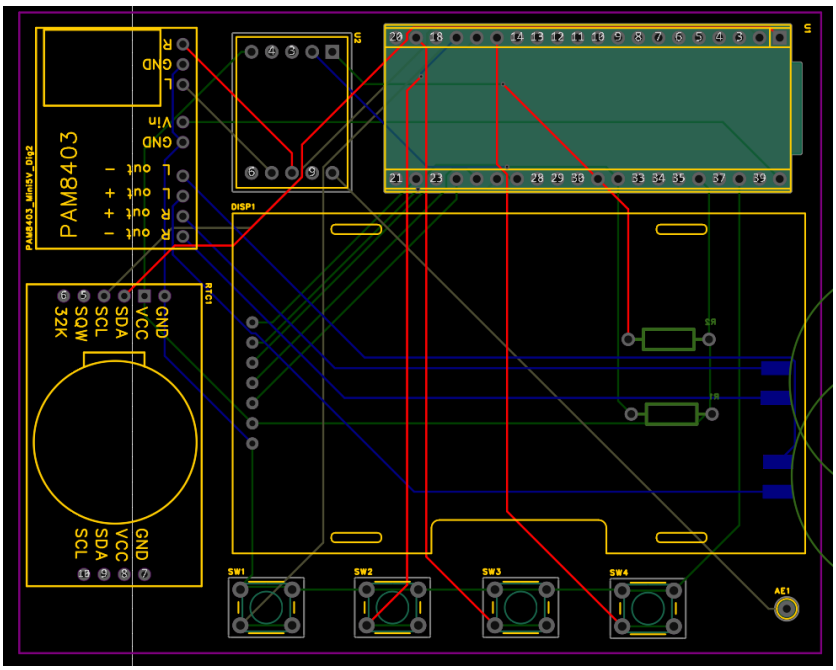

Figure 8.1: *Top-layer PCB layout showing signal routing, component placement, and silkscreen labels. Modules include the Raspberry Pi Pico W, DS3231 RTC, RDA5807M FM radio, PAM8403 amplifier, OLED display, and control buttons.*

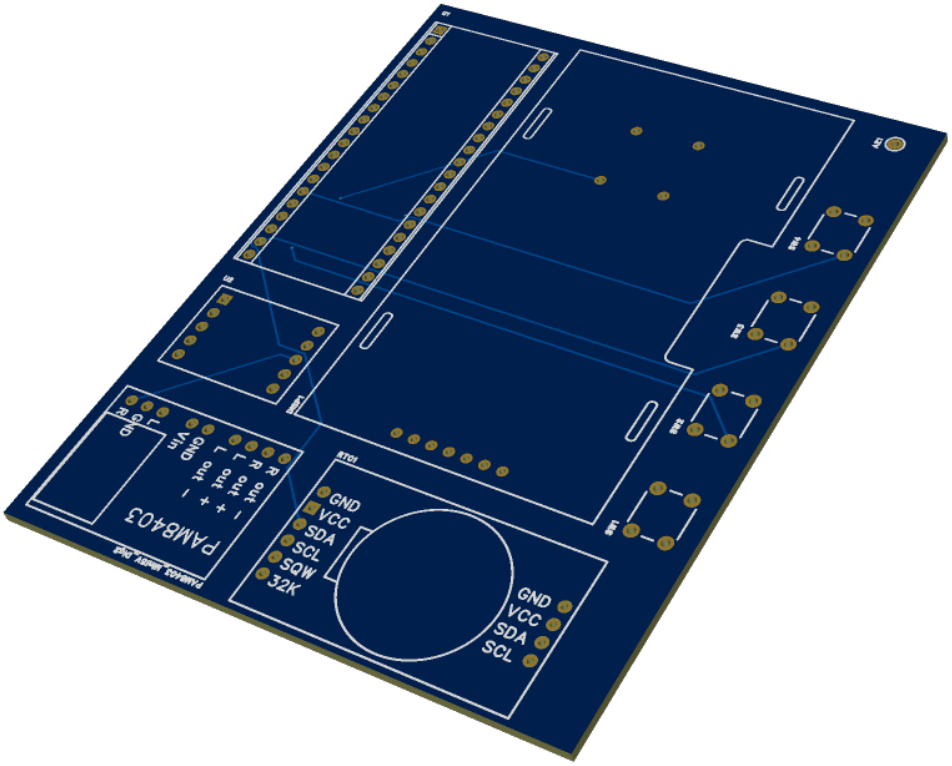

Figure 8.2: *3D rendered isometric view of the assembled PCB showing connector pads, cutouts, and silkscreen labels. This visual aids in understanding component orientation and physical dimensions prior to fabrication.*

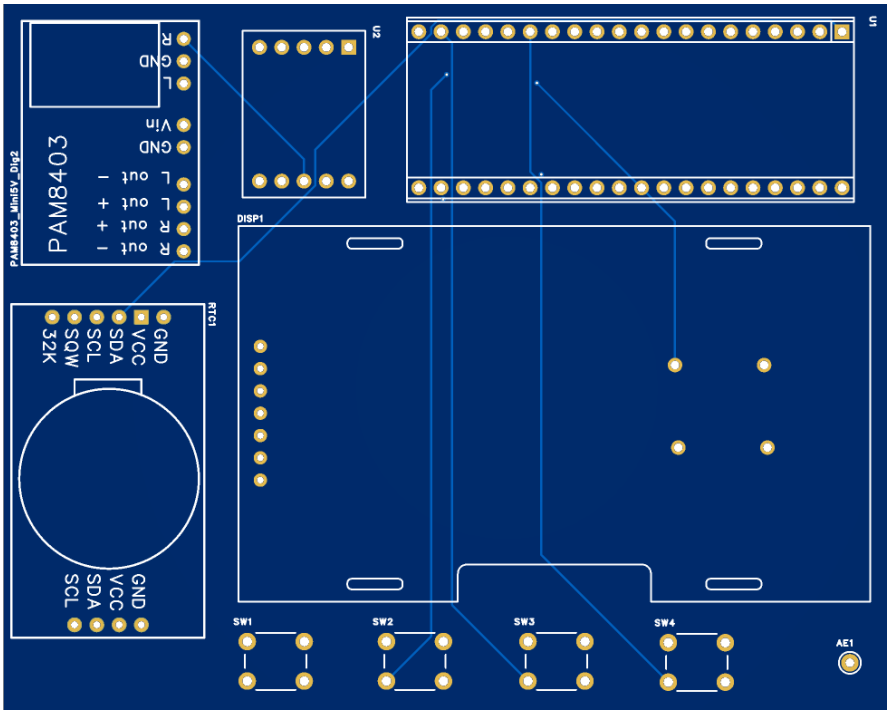

Figure 8.3: *Top-down 3D view of the PCB layout displaying key component footprints and button arrangements. The layout ensures optimal spacing and logical grouping of modules to simplify assembly.*

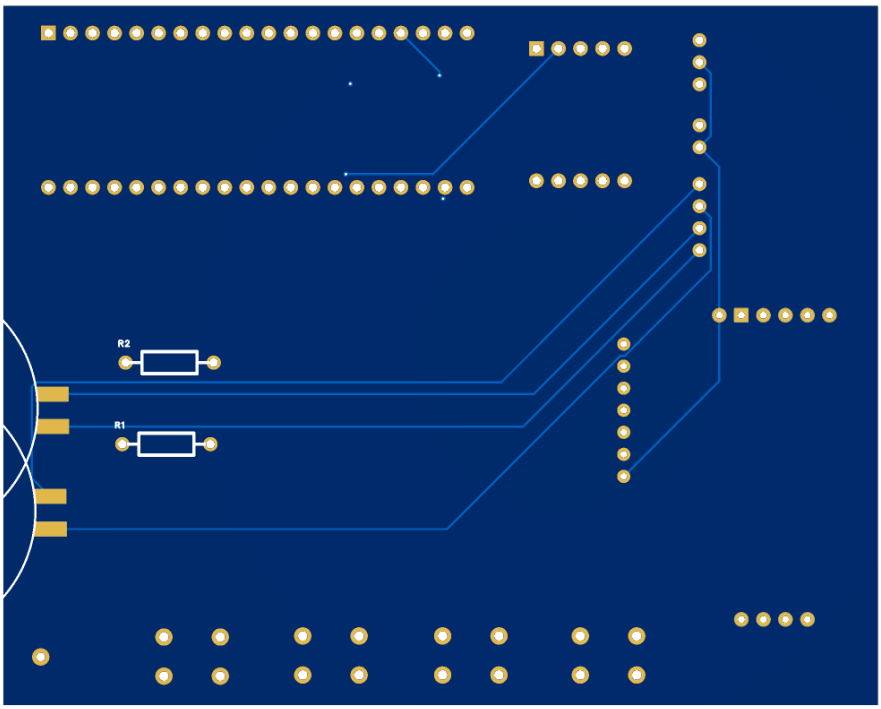

Figure 8.4: *Bottom-layer view of the PCB highlighting signal traces and via placements. This view verifies that two-layer routing constraints were properly managed to minimize interference and maintain design clarity.*

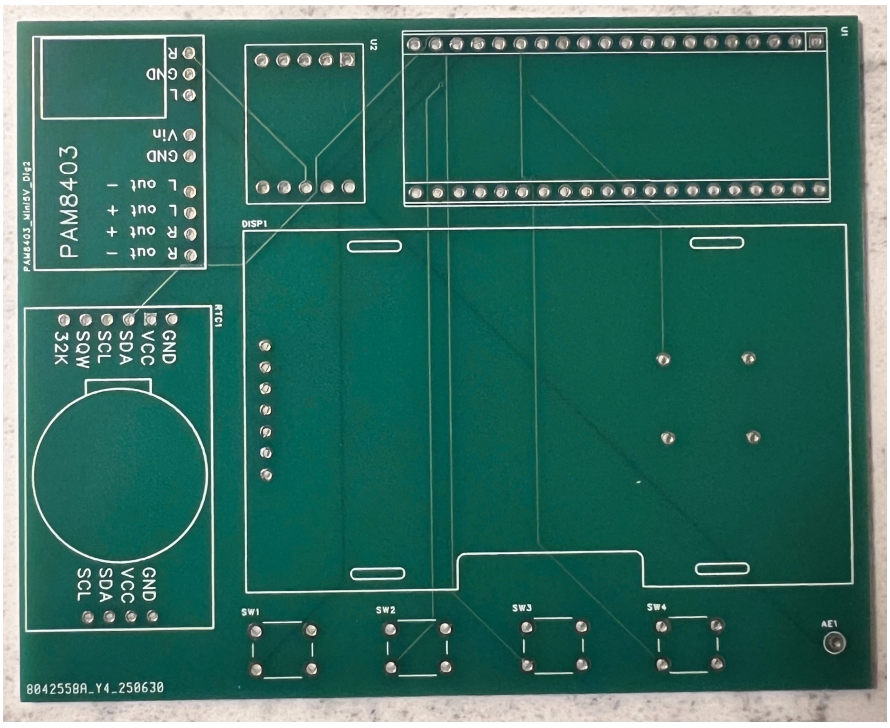

Figure 8.5: *Top view of the manufactured PCB*

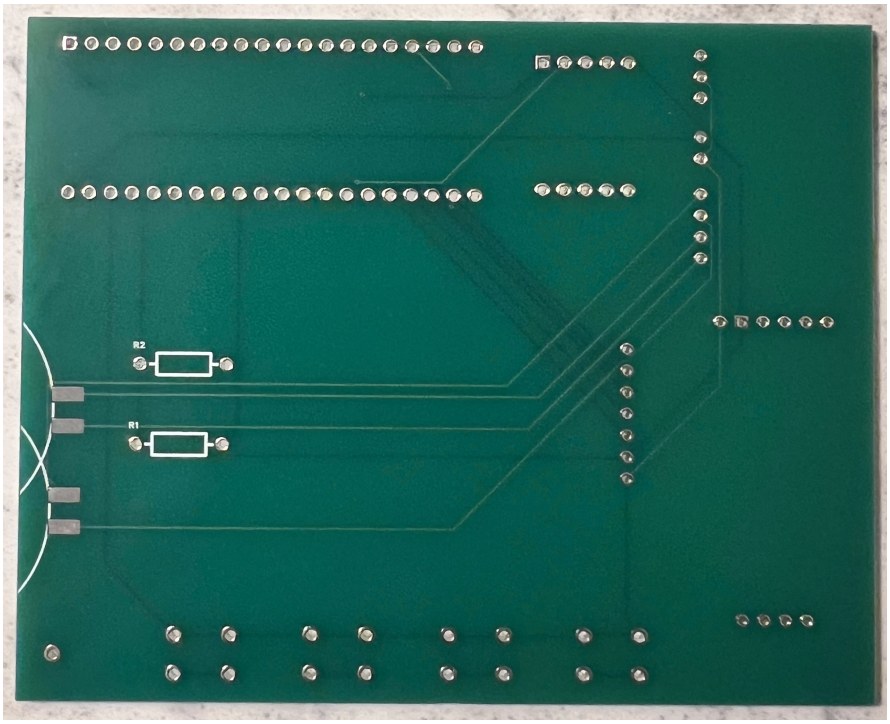

Figure 8.6: *Bottom view of the fabricated PCB*

The printed circuit board (PCB) for the clock-radio system was custom-designed using **KiCad** and measured 100 mm × 80 mm. The board was dimensioned to fit within a square-shaped enclosure and was designed to accommodate all modules in a compact yet organized manner. The final layout included footprints for the **Raspberry Pi Pico W**, **SSD1309 OLED display**, **DS3231 RTC module**, **RDA5807M FM module**, **PAM8403 amplifier**, a 4 Ω speaker, and four **tactile pushbuttons**.

A **two-layer PCB** was selected to simplify routing while keeping the design affordable and compatible with standard commercial manufacturers such as **JLCPCB** and **PCBWay**. A single-layer board would have introduced excessive trace overlap and complicated signal routing, while a four-layer board would have been unnecessarily expensive for this low-speed, low-current application. In this design, both layers were used interchangeably for signal and power traces, as isolation or dedicated planes were not critical.

No specialized trace-width calculations were performed, as all modules operate at low frequencies and currents. Uniform trace widths were used throughout, and the board passed design-rule checks (DRC) for spacing, clearance, and pad sizes.

The layout grouped components logically by functionality:

- **Central OLED display**: The 2.42″ SSD1309 module was placed centrally along the top edge for clear visibility of time and radio frequency.
- **Bottom-edge pushbuttons**: Four tactile switches were evenly spaced along the bottom edge to ensure consistent and accessible user interaction.

- **FM-amp adjacency**: The RDA5807M FM module was positioned next to the PAM8403 audio amplifier, allowing short, direct stereo audio traces and minimizing signal degradation.
- **Horizontal Pico alignment**: The Pico W was oriented horizontally across the top to maximize available GPIO routing area and provide sufficient space for signal fan-out.

Although no mounting holes or standoffs were included, cutouts were added to support **button access and display clearance**. The board was not shaped to resemble the elephant-themed enclosure, but it was dimensioned to fit inside it without internal obstructions.

Finally, the PCB was exported as **Gerber files** and submitted to a commercial manufacturer. The final unpopulated board was verified for trace quality, via plating, and alignment of footprints with the physical modules.

9

Enclosure Design

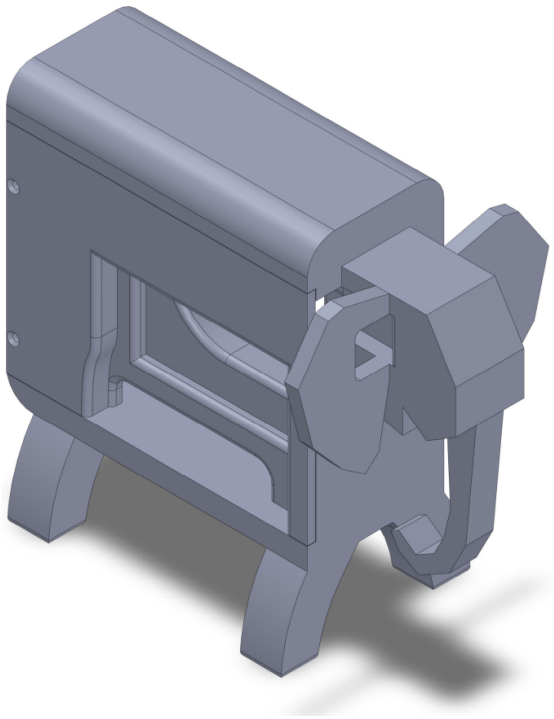

Figure 9.1: *3D isometric view of the elephant-themed enclosure, designed to house the full clock-radio system including the PCB, display, and interface buttons.*

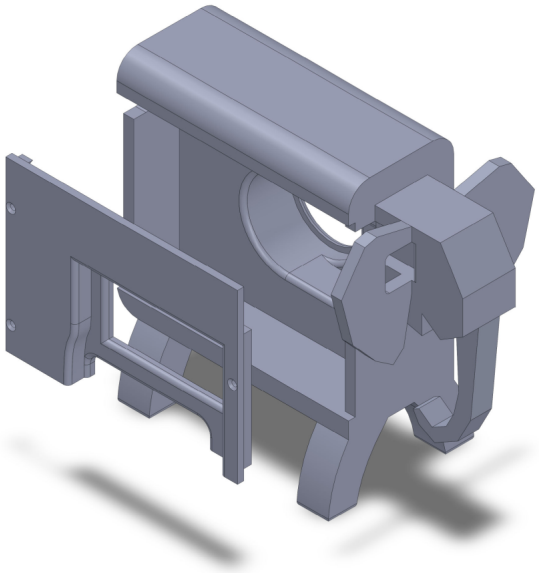

Figure 9.2: *Exploded view of the enclosure assembly showing the modular structure, including front panel, internal supports*

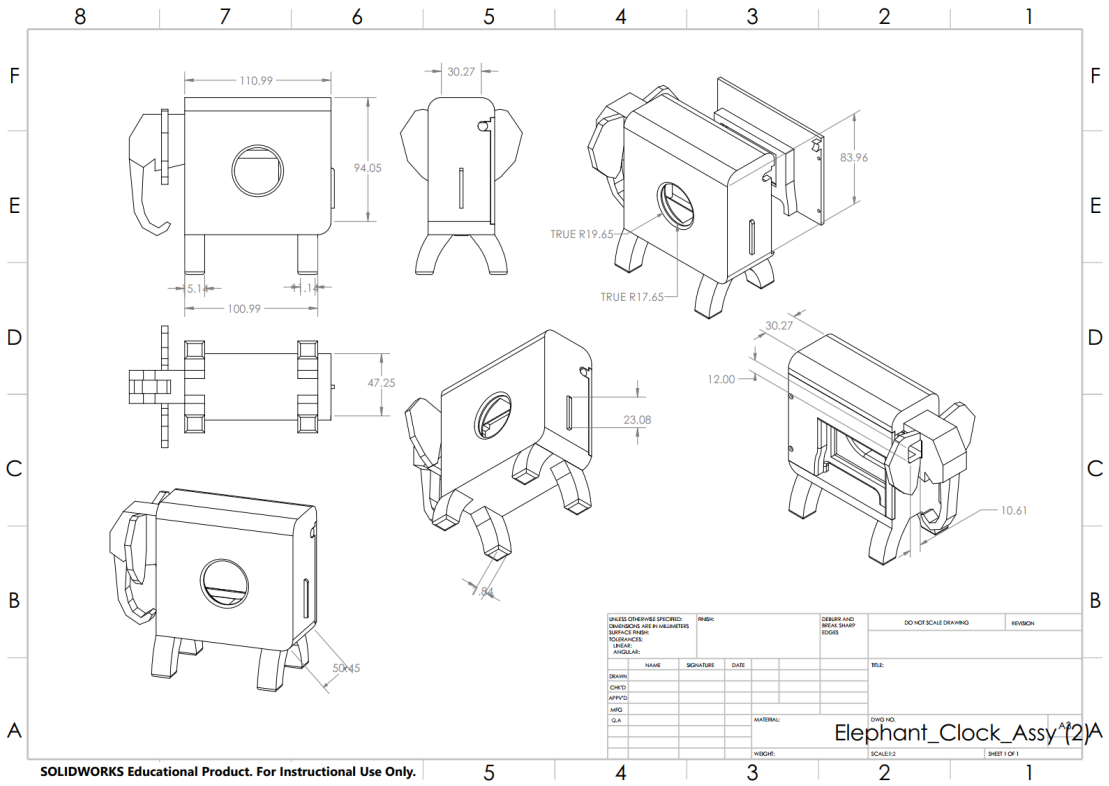

Figure 9.3: *2D technical drawing of the enclosure with dimensions (in mm), multiple views, and assembly projections created in SolidWorks.*

10

Testing and Validation

Table 10.1: *Raspberry Pi Pico GPIO Pin Assignments*

Component	GPIO Pin(s)
DS3231 RTC Module	GPIO 0 (SDA), GPIO 1 (SCL)
SSD1309 OLED Display	SPI0: GPIO 18 (SCK), GPIO 19 (MOSI); GPIO 17 (CS), GPIO 20 (DC), GPIO 21 (RST)
RDA5807M FM Radio Module	GPIO 26 (SDA), GPIO 27 (SCL)
PAM8403 Audio Amplifier	GPIO 2 (PWM), GPIO 3 (PWM)
Speaker Enable (FM Mode)	GPIO 10
Pushbutton: Menu	GPIO 12
Pushbutton: Set	GPIO 13
Pushbutton: Increment	GPIO 14
Pushbutton: Decrement	GPIO 11

Two subsystems, the DS3231 RTC module (clock functionality) and the RDA5807M FM radio module with audio output were tested independently to validate correct operation. The procedures below outline each subsystem's tests, expected outcomes, and verification methods.

10.1 DS3231 Real–Time Clock (RTC) Module

Objective: Verify that the DS3231 RTC module correctly maintains and reports the current time and date.
Test Script Reference: Integrated into main clock-radio firmware (see Appendix A).

Setup:

- DS3231 module connected to Raspberry Pi Pico via I^2C on GPIO 0 (SDA) and GPIO 1 (SCL).
- Imported and used the 'ds3231.py' library for communication.

Steps:

1. Power the system via USB to initialize the Pico and peripherals.
2. If the oscillator stop flag (OSF) was set, the firmware resets and programs the RTC with a predefined date/time.
3. Call `rtc.datetime()` to read the current date and time.
4. Observe the OLED display rendering live time, date, and weekday with updates each second.

Expected Result: The OLED display shows the correct current time, date, and weekday, updating every second.

Verification:

- Time updated once per second on the OLED.
- Manual time set via the 'set_clock()' menu correctly stored and retrieved, confirming I^2C communication and timekeeping.

10.2 RDA5807M FM Radio Module with PAM8403 Amplifier

Objective: Verify that the `Radio` class successfully initializes the RDA5807M FM module, tunes to known frequencies, and provides audio output via the PAM8403 amplifier and connected speakers.

Test Script Reference: Integrated into main clock-radio firmware (see Appendix B: RDA5807M FM Radio with Audio Output Test Script).

Steps:

1. Instantiate the class:

```
fm_radio = Radio(101.9, 4, False)
```

   which tuned to 101.9 MHz ("CFUV 101.9"), set volume to 4, and unmuted audio.
2. `ProgramRadio()` was called automatically, sending configuration via I^2C to the module at address `0x10`.
3. ROUT and LOUT outputs were routed to the PAM8403 amplifier on GPIO 2 and GPIO 3.
4. Speaker enable was controlled through GPIO 10.
5. `GetSettings()` retrieved runtime feedback (frequency, mute status, volume) for display or debugging.
6. `GetStationName()` mapped frequencies to station names for user display.

Expected Result: Audible FM broadcast is heard through the speakers; the OLED displays the correct frequency and station name; volume and mute controls operate correctly.

Verification:

- Audio output confirmed by ear immediately after tuning.
- Frequency changes (UP/DOWN buttons or web UI) occurred in 0.2 MHz steps.
- Display updated to show the matched station name (e.g., "The Q! Rock").

- Mute and unmute functionality operated as expected via method calls.

11

Code of Ethics (Engineers and Geoscientists BC)

The development of this clock-radio prototype was guided by several principles outlined in the Engineers and Geoscientists BC (EGBC) Code of Ethics. Below is a list of the relevant ethical standards and how they were applied throughout the project:

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.**
   Safety was prioritized during all phases of the project. Lead-free solder was used to minimize environmental and health hazards. Work in the lab was conducted in accordance with safety protocols, and all electrical connections were tested thoroughly to ensure reliable and safe operation of the prototype.
2. **Practice only in those fields where training and ability make the registrant professionally competent.**
   Team members participated in laboratory exercises and soldering workshops to build practical competence. Work was limited to areas in which members were trained, including circuit design, embedded systems programming, and enclosure modeling.
3. **Have regard for applicable standards, policies, plans, and practices established by the government or Engineers and Geoscientists BC.**
   The project was designed with awareness of regulatory practices related to safety. All components were operated within manufacturer-recommended conditions as defined in their datasheets.
4. **Maintain competence in relevant specializations, including advances in the regulated practice and relevant science.**
   The team researched current component options, consulted up-to-date datasheets, and referred to modern embedded system practices. For instance, lead-free solder and modular audio amplification techniques were adopted based on current best practices.
5. **Provide professional opinions that distinguish between facts, assumptions,**

and opinions.
Throughout the design process, engineering decisions were supported with data from datasheets and verified test results. Where design choices were based on usability or subjective preference (e.g., interface layout), these were clearly identified and separated from data-driven decisions.

6. **Undertake work and documentation with due diligence and in accordance with any guidance developed to standardize professional documentation.**
The report was structured based on ECE 299 requirements and the professional standards expected in engineering documentation. All calculations, design justifications, and component selections were recorded methodically.
7. **Conduct themselves with fairness, courtesy, and good faith toward clients, colleagues, and others.**
Although this project did not involve a client, the team collaborated respectfully and professionally. Responsibilities were shared, constructive feedback was welcomed, and credit was given to all contributors and referenced sources.

12

Conclusion and Recommendation

12.1 Conclusion

The design and implementation of the elephant-themed clock-radio prototype was accomplished through a structured and collaborative approach. The project was divided into distinct subsystems—software, clock and alarm logic, FM radio interface, user interface display, audio output, and enclosure design. This modular division allowed for parallel development and efficient delegation of tasks among team members.

Throughout the process, design decisions were made with considerations for cost, component availability, aesthetic value, and functional performance. The system successfully integrates an RTC (DS3231) for precise timekeeping, an FM radio module (RDA5807M) for entertainment, and an audio amplifier (PAM8403) for sound output, all managed by the Raspberry Pi Pico W. A responsive OLED display and intuitive button interface further enhance usability. The final result aligns with the project's goals: an interactive, low-cost, and user-friendly alarm clock with integrated radio functionality.

Although the prototype met most of the design expectations, a few improvements were identified that could enhance the design in future iterations.

12.2 Recommendations

1. **Optimize PCB Layout:** Consider a dedicated ground plane or separate signal layers to reduce noise and improve routing efficiency.
2. **Shield FM Radio:** Enhance reception quality by optimizing antenna trace layout and adding EMI shielding.
3. **Add Mounting Options:** Include mounting holes or clips in the enclosure design to secure the PCB.
4. **Include Battery Backup:** Integrate a Li-ion or coin-cell backup to maintain timekeeping and alarms during power outages.

5. **Improve Touch Interface:** Replace mechanical buttons with capacitive touch sensors for greater durability and a cleaner aesthetic.
6. **User Feedback Indicators:** Add LEDs or on-screen icons for alarm status, FM signal strength, and snooze countdown.
7. **Modular Architecture:** Design plug-and-play modules to simplify debugging and future upgrades.

Bibliography

[1] Maxim Integrated, "DS3231: Extremely Accurate I²C-Integrated RTC with TCXO/Crystal," Rev. 5, Jan. 2015. [Online]. Available: https://datasheets.maximintegrated.com/en/ds/DS3231.pdf

[2] Solomon Systech Ltd., "SSD1309: 128×64 Dot Matrix OLED/PLED Segment/Common Driver with Controller," Rev. 1.2, Apr. 2012. [Online]. Available: https://cdn-shop.adafruit.com/datasheets/SSD1309.pdf

[3] RDA Microelectronics, "RDA5807M: Single-Chip Broadcast FM Radio Tuner," Ver 1.1. [Online]. Available: https://datasheetspdf.com/pdf-file/732516/RDA-Microelectronics/RDA5807M/1

[4] Diodes Inc., "PAM8403: 3W Stereo Class-D Audio Amplifier," Mar. 2011. [Online]. Available: https://www.diodes.com/assets/Datasheets/PAM8403.pdf

A

Appendix A: DS3231 RTC Module Test Script

```python
from machine import I2C, Pin
from ds3231 import DS3231
import time

# Initialize I2C for RTC on GPIO 0 (SDA), GPIO 1 (SCL)
i2c = I2C(0, scl=Pin(1), sda=Pin(0))
rtc = DS3231(i2c)

# Set time if RTC has lost power
if rtc.OSF():
    rtc._OSF_reset()
    rtc.datetime((2025, 7, 29, 15, 30, 0, 2))  # Year, Month, Day, Hour, Minute, Second, Weekday

# Loop to print time every second
while True:
    dt = rtc.datetime()
    print(f"Time: {dt[4]:02}:{dt[5]:02}:{dt[6]:02} | Date: {dt[0]:04}-{dt[1]:02}-{dt[2]:02}")
    time.sleep(1)
```

Listing A.1: *RTC initialization and time-print test script (MicroPython).*

```
Shell
>>> %Run -c $EDITOR_CONTENT

MPY: soft reboot
Time: 15:36:34 | Date: 2025-07-29
Time: 15:36:35 | Date: 2025-07-29
Time: 15:36:36 | Date: 2025-07-29
Time: 15:36:37 | Date: 2025-07-29
Time: 15:36:38 | Date: 2025-07-29
Time: 15:36:39 | Date: 2025-07-29
Time: 15:36:40 | Date: 2025-07-29
Time: 15:36:41 | Date: 2025-07-29
Time: 15:36:42 | Date: 2025-07-29
Time: 15:36:43 | Date: 2025-07-29
Time: 15:36:44 | Date: 2025-07-29
Time: 15:36:45 | Date: 2025-07-29
Time: 15:36:46 | Date: 2025-07-29
Time: 15:36:47 | Date: 2025-07-29
Time: 15:36:48 | Date: 2025-07-29
Time: 15:36:49 | Date: 2025-07-29
Time: 15:36:50 | Date: 2025-07-29
Time: 15:36:51 | Date: 2025-07-29
Time: 15:36:52 | Date: 2025-07-29
Time: 15:36:53 | Date: 2025-07-29
Time: 15:36:54 | Date: 2025-07-29
Time: 15:36:55 | Date: 2025-07-29
Time: 15:36:56 | Date: 2025-07-29
Time: 15:36:57 | Date: 2025-07-29
Time: 15:36:58 | Date: 2025-07-29
Time: 15:36:59 | Date: 2025-07-29
```

Figure A.1: *Terminal output showing successful real-time timekeeping by the DS3231 RTC module, with continuous second-by-second updates.*

B

RDA5807M FM Radio with Audio Output Test Script

```
class Radio:
    # Preset frequencies and their names (Victoria, BC)
    station_presets = {
        88.1: "Campus Radio",
        90.5: "CBC Radio One",
        92.1: "CILS-FM (French)",
        98.5: "Ocean 98.5",
        100.3: "The Q! Rock",
        101.9: "CFUV 101.9",
        103.1: "Jack FM",
        107.3: "Virgin Radio"
    }

    def __init__(self, NewFrequency, NewVolume, NewMute):
        from machine import Pin, I2C
        self.presets = list(self.station_presets.keys())
        self.Volume = 2
        self.Mute = False
        self.Frequency = self._nearest_preset(NewFrequency)

        self.SetVolume(NewVolume)
        self.SetFrequency(NewFrequency)
        self.SetMute(NewMute)

        self.i2c_sda = Pin(26)
        self.i2c_scl = Pin(27)
        self.i2c_device = 1
        self.i2c_device_address = 0x10
        self.Settings = bytearray(8)
        self.radio_i2c = I2C(self.i2c_device,
                             scl=self.i2c_scl,
                             sda=self.i2c_sda,
                             freq=200000)

        self.ProgramRadio()

    def _nearest_preset(self, freq):
        return min(self.presets, key=lambda x: abs(x - freq))

    def SetVolume(self, NewVolume):
```

```
41          try:
42              NewVolume = int(NewVolume)
43          except:
44              return False
45          if not (0 <= NewVolume < 16):
46              return False
47          self.Volume = NewVolume
48          return True
49
50      def SetFrequency(self, NewFrequency):
51          try:
52              NewFrequency = float(NewFrequency)
53          except:
54              return False
55          self.Frequency = self._nearest_preset(NewFrequency)
56          return True
57
58      def SetMute(self, NewMute):
59          try:
60              self.Mute = bool(int(NewMute))
61          except:
62              return False
63          return True
64
65      def ComputeChannelSetting(self, Frequency):
66          ch = int(Frequency * 10) - 870
67          b = bytearray(2)
68          b[0] = (ch >> 2) & 0xFF
69          b[1] = ((ch & 0x03) << 6) & 0xC0
70          return b
71
72      def UpdateSettings(self):
73          self.Settings[0] = 0x80 if self.Mute else 0xC0
74          self.Settings[1] = 0x0D
75          self.Settings[2:4] = self.ComputeChannelSetting(self.Frequency)
76          self.Settings[3] |= 0x10
77          self.Settings[4] = 0x04
78          self.Settings[5] = 0x00
79          self.Settings[6] = 0x84
80          self.Settings[7] = 0x80 + self.Volume
81
82      def ProgramRadio(self):
83          self.UpdateSettings()
84          self.radio_i2c.writeto(self.i2c_device_address, self.Settings)
85
86      def GetSettings(self):
87          st = self.radio_i2c.readfrom(self.i2c_device_address, 256)
88          mute = not ((st[0xF0] & 0x40) != 0)
89          vol  = st[0xF7] & 0x0F
90          freq = (((st[0x00]&0x03)<<8)|(st[0x01]&0xFF)) * 0.1 + 87.0
91          stereo = (st[0x00] & 0x04) != 0
92          return (mute, vol, freq, stereo)
93
94      def GetStationName(self):
95          return self.station_presets.get(self.Frequency, "Unknown Station")
```

C

Appendix C: main.py code (Full code for this project)

Listing C.1: *Full MicroPython application code for the clock-radio prototype.*

```
from machine import I2C, SPI, Pin, PWM
from ds3231 import DS3231
from ssd1309 import Display
import network, socket, ure, time, utime

# === FM Radio Class with Preset Support ===
class Radio:
    # Preset frequencies and their names (Victoria, BC)
    station_presets = {
        88.1: "Campus Radio",
        90.5: "CBC Radio One",
        92.1: "CILS-FM (French)",
        98.5: "Ocean 98.5",
        100.3: "The Q! Rock",
        101.9: "CFUV 101.9",
        103.1: "Jack FM",
        107.3: "Virgin Radio"
    }

    def __init__(self, NewFrequency, NewVolume, NewMute):
        self.presets = list(self.station_presets.keys())
        self.Volume = 2
        self.Mute = False
        self.Frequency = self._nearest_preset(NewFrequency)

        self.SetVolume(NewVolume)
        self.SetFrequency(NewFrequency)
        self.SetMute(NewMute)

        self.i2c_sda = Pin(26)
        self.i2c_scl = Pin(27)
        self.i2c_device = 1
        self.i2c_device_address = 0x10
        self.Settings = bytearray(8)
        self.radio_i2c = I2C(self.i2c_device,
                             scl=self.i2c_scl,
                             sda=self.i2c_sda,
                             freq=200000)

```

```
        self.ProgramRadio()

    def _nearest_preset(self, freq):
        return min(self.presets, key=lambda x: abs(x - freq))

    def SetVolume(self, NewVolume):
        try:
            NewVolume = int(NewVolume)
        except:
            return False
        if not (0 <= NewVolume < 16):
            return False
        self.Volume = NewVolume
        return True

    def SetFrequency(self, NewFrequency):
        try:
            NewFrequency = float(NewFrequency)
        except:
            return False
        if not 87.5 <= NewFrequency <= 108.0:
            return False
        self.Frequency = round(NewFrequency, 1)
        return True

    def SetMute(self, NewMute):
        try:
            self.Mute = bool(int(NewMute))
        except:
            return False
        return True

    def ComputeChannelSetting(self, Frequency):
        Frequency = int(Frequency * 10) - 870
        ByteCode = bytearray(2)
        ByteCode[0] = (Frequency >> 2) & 0xFF
        ByteCode[1] = ((Frequency & 0x03) << 6) & 0xC0
        return ByteCode

    def UpdateSettings(self):
        self.Settings[0] = 0x80 if self.Mute else 0xC0
        self.Settings[1] = 0x0D
        self.Settings[2:4] = self.ComputeChannelSetting(self.Frequency)
        self.Settings[3] |= 0x10
        self.Settings[4] = 0x04
        self.Settings[5] = 0x00
        self.Settings[6] = 0x84
        self.Settings[7] = 0x80 + self.Volume

    def ProgramRadio(self):
        self.UpdateSettings()
        self.radio_i2c.writeto(self.i2c_device_address, self.Settings)

    def GetSettings(self):
        st = self.radio_i2c.readfrom(self.i2c_device_address, 256)
        mute   = not ((st[0xF0] & 0x40) != 0)
        vol    = st[0xF7] & 0x0F
        freq   = (((st[0x00] & 0x03) << 8) | (st[0x01] & 0xFF)) * 0.1 + 87.0
        stereo = (st[0x00] & 0x04) != 0
```

```
99          return (mute, vol, freq, stereo)
100
101     def GetStationName(self):
102         return self.station_presets.get(self.Frequency, "Unknown Station")
103
104 # === RTC SETUP ===
105 i2c_rtc = I2C(0, scl=Pin(1), sda=Pin(0))
106 rtc = DS3231(i2c_rtc)
107
108 def set_initial_time():
109     initial_datetime = (2025, 7, 29, 15, 30, 0, 2)
110     rtc.datetime(initial_datetime)
111     print("RTC time set to:", initial_datetime)
112
113 if rtc.OSF():
114     print("RTC lost power. Resetting time.")
115     rtc._OSF_reset()
116     set_initial_time()
117
118 # === Display ===
119 spi = SPI(0, baudrate=10_000_000, sck=Pin(18), mosi=Pin(19))
120 oled = Display(spi=spi, cs=Pin(17), dc=Pin(20), rst=Pin(21),
121                width=128, height=64, flip=True)
122 fb = oled.monoFB
123
124 # === Audio Setup ===
125 speaker_enable = Pin(10, Pin.OUT)
126 speaker_enable.value(0)
127 fm_radio = Radio(101.9, 4, True)
128
129 alarm_pwm = PWM(Pin(2))
130 alarm_pwm.duty_u16(0)
131
132  === Note Frequencies (Hz) ===
133 NOTE_C4 = 262; NOTE_CS4 = 277; NOTE_D4 = 294; NOTE_E4 = 330; NOTE_F4 = 349; NOTE_G4 = 392;
    ↪  NOTE_A4 = 440; NOTE_B4 = 494
134 NOTE_C5 = 523; NOTE_D5 = 587; NOTE_E5 = 659; NOTE_F5 = 698; NOTE_G5 = 784; NOTE_A5 = 880; NOTE_B5
    ↪  = 988
135 NOTE_FS4 = 370; NOTE_GS4 = 415; NOTE_AS4 = 466; NOTE_CS5 = 554; NOTE_DS5 = 622; NOTE_FS5 = 740;
    ↪  NOTE_GS5 = 831
136
137 # === Melodies ===
138 nokia_tune = [
139     (NOTE_E5, 188), (NOTE_D5, 188), (NOTE_FS4, 375), (NOTE_GS4, 375),
140     (NOTE_CS5, 188), (NOTE_B4, 188), (NOTE_D4, 375), (NOTE_E4, 375),
141     (NOTE_B4, 188), (NOTE_A4, 188), (NOTE_CS4, 250), (NOTE_E4, 375),
142     (NOTE_A4, 750)
143 ]
144 mario_tune = [
145     (NOTE_E5, 100), (NOTE_E5, 100), (0, 100), (NOTE_E5, 100), (0, 100), (NOTE_C5, 100), (NOTE_E5,
    ↪  200),
146     (NOTE_G5, 200), (0, 200), (NOTE_G4, 200)
147 ]
148 mission_impossible_tune = [
149     (NOTE_G5, 100), (NOTE_G5, 100), (NOTE_A5, 100), (NOTE_C5, 100),
150     (NOTE_G5, 100), (NOTE_G5, 100), (NOTE_F5, 100), (NOTE_E5, 100),
151 ]
152 addams_family_tune = [
153     (NOTE_D4, 200), (NOTE_F4, 200), (NOTE_G4, 200), (NOTE_A4, 200), (NOTE_A4, 50), (NOTE_B4,
    ↪  400),
```

```
]
twinkle_twinkle_tune = [
    (NOTE_C4, 250), (NOTE_C4, 250), (NOTE_G4, 250), (NOTE_G4, 250),
    (NOTE_A4, 250), (NOTE_A4, 250), (NOTE_G4, 500)
]
alarm_tunes = [
    {"name": "Nokia", "melody": nokia_tune},
    {"name": "Mario", "melody": mario_tune},
    {"name": "Mission Imp", "melody": mission_impossible_tune},
    {"name": "Addams Fam", "melody": addams_family_tune},
    {"name": "Twinkle", "melody": twinkle_twinkle_tune}
]


# === I/O ===
class DebouncedButton:
    def __init__(self, pin_num):
        self.pin = Pin(pin_num, Pin.IN, Pin.PULL_UP)
        self.last_time = 0
    def pressed(self):
        if not self.pin.value():
            now = time.ticks_ms()
            if time.ticks_diff(now, self.last_time) > 200:
                self.last_time = now
                return True
        return False

btn_menu = DebouncedButton(12)
btn_set  = DebouncedButton(13)
btn_inc  = DebouncedButton(14)
btn_dec  = DebouncedButton(11)


# === STATE ===
alarm_time = [7, 0]
alarm_enabled = True
alarm_triggered = False
alarm_snoozed = False
is_24_hour_format = False
tz_index = 0
current_tune_idx = 0
snooze_duration = 5  # minutes
snooze_end_time = 0    # stores the utime.ticks_ms() when snooze ends

# **NEW GLOBALS FOR SCREEN MANAGEMENT**
screen_mode = 0       # 0=clock, 2=FM, 3=format
display_dirty = True  # marks when display should be refreshed
TZ_MODE = 4

# === TIMEZONE SUPPORT ===
time_zones = [("PST", -8), ("MST", -7), ("CST", -6), ("EST", -5), ("UTC", 0), ("GMT+1", 1),
↪  ("GMT+2", 2)]


# === RENDER CLOCK ===
def draw_clock():
    dt = rtc.datetime()
    year, month, day, weekday, hr, mi, se = dt[0], dt[1], dt[2], dt[3], dt[4], dt[5], dt[6]

```

```
212     weekday_names = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
213     weekday_str = weekday_names[weekday]
214 
215     tz_name, _ = time_zones[tz_index]
216     raw_hr = hr
217 
218     fb.fill(0)
219     fb.text(f"{weekday_str}", 0, 0)
220     fb.text(f"{year:04}-{month:02}-{day:02} {tz_name}", 0, 10)
221 
222     if is_24_hour_format:
223         time_str = f"{raw_hr:02}:{mi:02}:{se:02}"
224         fb.text(time_str, 0, 20)
225     else:
226         ampm = "AM" if raw_hr < 12 else "PM"
227         h12 = raw_hr % 12 or 12
228         time_str = f"{h12:02}:{mi:02}:{se:02} {ampm}"
229         fb.text(time_str, 0, 20)
230 
231     alarm_status = "On" if alarm_enabled else "Off"
232     alarm_str = f"Alarm: {alarm_time[0]:02}:{alarm_time[1]:02} [{alarm_status}]"
233     tune_name = alarm_tunes[current_tune_idx]['name']
234     fb.text(alarm_str, 0, 30)
235     fb.text(f"Tune: {tune_name}", 0, 40)
236 
237     if alarm_snoozed:
238         fb.text("Snoozing...", 0, 50)
239 
240     oled.present()
241 
242 def draw_wakeup_screen():
243     fb.fill(0)
244     fb.text("Wake up!", 40, 28)
245     oled.present()
246 
247 
248 
249 def draw_tz_screen():
250     fb.fill(0)
251     fb.text("Select Time Zone", 0, 0)
252     # list each entry, marking the current one
253     for idx, (tz_name, _) in enumerate(time_zones):
254         prefix = ">" if idx == tz_index else " "
255         fb.text(f"{prefix}{tz_name}", 0, 10 + idx * 10)
256     oled.present()
257 
258 
259 def draw_fm_radio_screen():
260     fb.fill(0)
261     fb.text("== FM RADIO ==", 10, 0)
262     fb.text(f"Freq: {fm_radio.Frequency:05.1f} MHz", 0, 20)
263     fb.text(f"Name: {fm_radio.GetStationName()}", 0, 35)
264     fb.text("Menu=Exit  ↑↓=Tune", 0, 55)
265     oled.present()
266 
267 def update_display():
268     global screen_mode
269     if alarm_triggered:
270         draw_wakeup_screen()
```

```
271     elif screen_mode == 2:
272         draw_fm_radio_screen()
273     elif screen_mode == 3:
274         draw_format()
275     elif screen_mode == TZ_MODE:
276         draw_tz_screen()
277     else:
278         draw_clock()
279 
280 
281 def handle_active_alarm():
282     global alarm_triggered, alarm_snoozed, alarm_enabled, snooze_end_time
283 
284     print("Alarm is active. Playing tune and waiting for input.")
285     melody = alarm_tunes[current_tune_idx]["melody"]
286 
287     while True: # Loop indefinitely until a button is pressed
288         # Play through the melody once
289         for note, duration in melody:
290             # --- Check for buttons after each note for responsiveness ---
291             # Snooze Button
292             if btn_set.pressed():
293                 print("Set button pressed. Snoozing alarm.")
294                 alarm_triggered = False
295                 alarm_snoozed = True
296                 snooze_end_time = utime.ticks_add(utime.ticks_ms(), snooze_duration * 60000)
297                 alarm_pwm.duty_u16(0) # Silence alarm
298                 return # Exit the alarm handling
299 
300             # Stop Button
301             if btn_inc.pressed():
302                 print("Inc button pressed. Stopping alarm.")
303                 alarm_triggered = False
304                 alarm_enabled = False # Disable alarm completely
305                 alarm_pwm.duty_u16(0) # Silence alarm
306                 return # Exit the alarm handling
307 
308             # Play the note
309             if note == 0:
310                 alarm_pwm.duty_u16(0)
311             else:
312                 alarm_pwm.freq(note)
313                 alarm_pwm.duty_u16(49152)
314 
315             utime.sleep_ms(duration)
316             alarm_pwm.duty_u16(0)
317             utime.sleep_ms(30)
318 
319 def play_alarm_tune(preview=False):
320     melody = alarm_tunes[current_tune_idx]["melody"]
321     notes_to_play = melody[:4] if preview else melody
322 
323     for note, duration in notes_to_play:
324         if not preview and (btn_menu.pressed() or btn_set.pressed() or btn_inc.pressed() or
        ↪  btn_dec.pressed()):
325             return False
326         if note == 0:
327             alarm_pwm.duty_u16(0)
328         else:
```

```
            alarm_pwm.freq(note)
            alarm_pwm.duty_u16(49152)
        utime.sleep_ms(duration)
        alarm_pwm.duty_u16(0)
        utime.sleep_ms(30)
    return True

def set_clock():
    current_dt = rtc.datetime()
    time_to_set = {
        "year": current_dt[0], "month": current_dt[1], "day": current_dt[2],
        "hour": current_dt[4], "minute": current_dt[5]
    }
    fields = ["Year", "Month", "Day", "Hour", "Minute"]
    field_keys = ["year", "month", "day", "hour", "minute"]
    field_idx = 0
    while True:
        fb.fill(0)
        fb.text("Set Clock:", 0, 0)
        for i, label in enumerate(fields):
            prefix = ">" if i == field_idx else " "
            fb.text(f"{prefix}{label}: {time_to_set[field_keys[i]]:02}", 0, 10 + i * 10)
        oled.present()
        key = field_keys[field_idx]
        if btn_inc.pressed(): time_to_set[key] += 1
        if btn_dec.pressed(): time_to_set[key] -= 1
        if btn_set.pressed(): field_idx = (field_idx + 1) % len(fields)
        if btn_menu.pressed():
            y, m, d, h, mi = time_to_set["year"], time_to_set["month"], time_to_set["day"],
            ↪  time_to_set["hour"], time_to_set["minute"]
            weekday = time.localtime(time.mktime((y, m, d, h, mi, 0, 0, 0)))[6]
            rtc.datetime((y, m, d, h, mi, 0, weekday))
            return
        time.sleep_ms(100)

# === SET ALARM ===
def set_alarm():
    global alarm_time
    # Default to current time when setting alarm
    dt = rtc.datetime()
    hour, minute = dt[4], dt[5]

    field = 0
    while True:
        fb.fill(0)
        fb.text("Set Alarm:", 0, 0)
        h_disp, m_disp = (f"[{hour:02}]", f"{minute:02}") if field == 0 else (f"{hour:02}",
        ↪  f"[{minute:02}]")
        fb.text(f"{h_disp}:{m_disp}", 0, 10)
        oled.present()
        if btn_inc.pressed():
            if field == 0: hour = (hour + 1) % 24
            else: minute = (minute + 1) % 60
        if btn_dec.pressed():
            if field == 0: hour = (hour - 1) % 24
            else: minute = (minute - 1) % 60
        if btn_set.pressed(): field = (field + 1) % 2
        if btn_menu.pressed():
            alarm_time = [hour, minute]
```

```
            return
        time.sleep_ms(100)

# === Menu ===
def menu_loop():
    global is_24_hour_format, tz_index, alarm_enabled, current_tune_idx, screen_mode
    menu_items = [
        "Set Clock", "Set Alarm", "Select Tune", "Set Snooze",
        "Alarm On/Off", "Play Radio", "Toggle 24H", "Change TZ", "Exit"
    ]
    menu_idx = 0
    scroll_top_idx = 0
    visible_items = 5

    while True:
        if menu_idx < scroll_top_idx:
            scroll_top_idx = menu_idx
        if menu_idx >= scroll_top_idx + visible_items:
            scroll_top_idx = menu_idx - visible_items + 1

        fb.fill(0)
        fb.text("== MENU ==", 0, 0)
        for i in range(visible_items):
            item_idx = scroll_top_idx + i
            if item_idx < len(menu_items):
                item = menu_items[item_idx]
                prefix = ">" if item_idx == menu_idx else " "
                fb.text(f"{prefix}{item}", 0, 10 + i * 10)
        oled.present()

        if btn_inc.pressed():
            menu_idx = (menu_idx + 1) % len(menu_items)
        if btn_dec.pressed():
            menu_idx = (menu_idx - 1 + len(menu_items)) % len(menu_items)

        if btn_menu.pressed():
            action = menu_items[menu_idx]
            if action == "Set Clock": set_clock()
            elif action == "Set Alarm": set_alarm()
            elif action == "Select Tune": select_tune_menu()
            elif action == "Set Snooze": set_snooze_duration()
            elif action == "Alarm On/Off": alarm_enabled = not alarm_enabled
            elif action == "Play Radio":
                speaker_enable.value(1)
                fm_radio.SetMute(False)
                fm_radio.ProgramRadio()
                screen_mode = 2
                return
            elif action == "Toggle 24H": is_24_hour_format = not is_24_hour_format
            elif action == "Change TZ": tz_index = (tz_index + 1) % len(time_zones)
            elif action == "Exit": return
            draw_clock()
        time.sleep_ms(100)


def set_snooze_duration():
    global snooze_duration
    duration = snooze_duration
    while True:
```

```
        fb.fill(0)
        fb.text("Set Snooze (min):", 0, 0)
        fb.text(f"{duration}", 0, 10)
        oled.present()

        if btn_inc.pressed(): duration = (duration + 1) % 61 # 0-60 min
        if btn_dec.pressed(): duration = (duration - 1) % 61

        if btn_set.pressed():
            snooze_duration = duration
            return

        if btn_menu.pressed():
            return

        time.sleep_ms(100)

def select_tune_menu():
    global current_tune_idx
    tune_idx = current_tune_idx
    while True:
        fb.fill(0)
        fb.text("Select Tune:", 0, 0)
        for i, tune in enumerate(alarm_tunes):
            prefix = ">" if i == tune_idx else " "
            fb.text(f"{prefix}{tune['name']}", 0, 10 + i * 10)
        oled.present()
        if btn_inc.pressed():
            tune_idx = (tune_idx + 1) % len(alarm_tunes)
            play_alarm_tune(preview=True)
        if btn_dec.pressed():
            tune_idx = (tune_idx - 1) % len(alarm_tunes)
            play_alarm_tune(preview=True)
        if btn_set.pressed():
            current_tune_idx = tune_idx
            return
        if btn_menu.pressed():
            return
        time.sleep_ms(100)

import network, socket, time

# === Set up Pico as Access Point ===
ap = network.WLAN(network.AP_IF)
ap.config(essid="AlarmClock_Pulse", password="12345678")
ap.active(True)

print("Starting AP…")
while not ap.active():
    time.sleep(0.5)

ip, netmask, gw, dns = ap.ifconfig()
print("AP Active. Connect to Wi-Fi network →", ap.config('essid'))
print("Visit: http://%s:8080/" % ip)

# === Start server bound to 0.0.0.0 on port 8080 ===
addr = socket.getaddrinfo("0.0.0.0", 8080)[0][-1]
s = socket.socket()
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

```
s.bind(addr)
s.listen(1)
s.settimeout(0.1)
print("Web server listening on port 8080…")


# === Web Interface ===
def web_page():
    dt = rtc.datetime()
    tz_name, _ = time_zones[tz_index]
    hr, mi, se = dt[4], dt[5], dt[6]
    ampm = "AM" if hr < 12 else "PM"
    h12 = hr % 12 or 12
    time_display = (
        f"{hr:02}:{mi:02}:{se:02}"
        if is_24_hour_format
        else f"{h12:02}:{mi:02}:{se:02} {ampm}"
    )
    mute, vol, freq, _ = fm_radio.GetSettings()
    alarm_str = f"{alarm_time[0]:02}:{alarm_time[1]:02}"
    curr_date = f"{dt[0]:04d}-{dt[1]:02d}-{dt[2]:02d}"
    curr_time = f"{hr:02d}:{mi:02d}"

    return f"""<!DOCTYPE html>
<html>
<head>
  <title>Clock-Radio</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <style>
    body {{
      font-family: sans-serif;
      background: #f0f0f0;
      text-align: center;
      padding: 20px;
    }}
    .card {{
      background: white;
      padding: 20px;
      margin: 10px auto;
      border-radius: 10px;
      box-shadow: 0 2px 5px rgba(0,0,0,0.2);
      max-width: 400px;
    }}
    button {{
      padding: 10px 20px;
      margin: 5px;
      font-size: 16px;
      border: none;
      border-radius: 5px;
      background: #007BFF;
      color: white;
      cursor: pointer;
    }}
    button:hover {{
      background: #0056b3;
    }}
    input {{
      padding: 8px;
      margin: 5px;
```

```
      border-radius: 5px;
      font-size: 16px;
      width: 60%;
    }}
  </style>
</head>
<body>
  <div class="card">
    <h2>Clock-Radio</h2>
    <p><strong>Time:</strong> {time_display}</p>
    <p>
      <strong>Format:</strong> {"24-hour" if is_24_hour_format else "12-hour"} |
      <strong>Timezone:</strong> {tz_name}
    </p>
    <form action="/toggle_format"><button>Toggle Format</button></form>
    <form action="/change_tz"><button>Next Timezone</button></form>
  </div>

  <div class="card">
    <h3>Date</h3>
    <p><strong>Set Date:</strong></p>
    <form action="/set_date">
      <input type="date" name="date" value="{curr_date}">
      <button type="submit">Set</button>
    </form>
  </div>

  <div class="card">
    <h3>Time</h3>
    <p><strong>Set Time:</strong></p>
    <form action="/set_time">
      <input type="time" name="time" value="{curr_time}">
      <button type="submit">Set</button>
    </form>
  </div>


  <div class="card">
    <h3>Alarm</h3>
    <p><strong>Set Alarm:</strong></p>
    <form action="/set_alarm">
      <input type="time" name="alarm" value="{alarm_str}">
      <button type="submit">Set</button>
    </form>
    <form action="/snooze"><button>Snooze</button></form>
  </div>

  <div class="card">
    <h3>FM Radio</h3>
    <p><strong>Frequency:</strong> {freq:.1f} MHz</p>
    <form action="/set_freq">
      <input type="number" step="0.1" name="freq" value="{freq:.1f}" min="87.5" max="108.0">
      <button type="submit">Tune</button>
    </form>
    <form action="/mute"><button>Mute</button></form>
    <form action="/unmute"><button>Unmute</button></form>
  </div>
</body>
</html>"""
```

```
622
623 # === Connect to Wi-Fi
624 #ssid = "Rich_AfricanPrince"
625 #password = "Engineering2027"
626
627
628
629 print("  Clock-Radio is running...")
630 last_sec = -1# === Main Loop ===
631 while True:
632     # --- Web Server (non-blocking) ---
633     try:
634         cl, addr = s.accept()
635         req = cl.recv(1024).decode()
636
637         if 'GET /set_alarm?' in req:
638             m = ure.search(r"alarm=(\d+):(\d+)", req)
639             if m:
640                 alarm_time[0], alarm_time[1] = int(m.group(1)), int(m.group(2))
641                 alarm_snoozed = False
642                 display_dirty = True
643
644         elif 'GET /set_freq?' in req:
645             m = ure.search(r"freq=([\d.]+)", req)
646             if m:
647                 fm_radio.SetFrequency(float(m.group(1)))
648                 fm_radio.ProgramRadio()
649                 display_dirty = True
650
651         elif 'GET /toggle_format' in req:
652             is_24_hour_format = not is_24_hour_format
653             screen_mode = 3
654             display_dirty = True
655
656         elif 'GET /change_tz' in req:
657             screen_mode = TZ_MODE   # switch into the TZ selection screen
658             display_dirty = True
659
660         elif 'GET /mute' in req:
661             fm_radio.SetMute(True)
662             fm_radio.ProgramRadio()
663             screen_mode = 2
664             display_dirty = True
665
666         elif 'GET /unmute' in req:
667             fm_radio.SetMute(False)
668             fm_radio.ProgramRadio()
669             screen_mode = 2
670             display_dirty = True
671
672         elif 'GET /snooze' in req:
673             alarm_triggered = False
674             alarm_snoozed = True
675             alarm_pwm.duty_u16(0)
676             total = alarm_time[0] * 60 + alarm_time[1] + snooze_duration
677             alarm_time[0] = (total // 60) % 24
678             alarm_time[1] = total % 60
679             display_dirty = True
680
```

```
681         elif 'GET /set_time?' in req:
682             m = ure.search(r"time=(\d+):(\d+)", req)
683             if m:
684                 h, mi = int(m.group(1)), int(m.group(2))
685                 curr = rtc.datetime()
686                 weekday = time.localtime(time.mktime((curr[0], curr[1], curr[2], h, mi, 0, 0,
                    ↪  0)))[6]
687                 rtc.datetime((curr[0], curr[1], curr[2], h, mi, 0, weekday))
688                 print("  RTC updated to:", h, mi)
689                 display_dirty = True
690
691
692
693
694         elif 'GET /set_date?' in req:
695             m = ure.search(r"date=(\d+)-(\d+)-(\d+)", req)
696             if m:
697                 y, mo, da = map(int, m.groups())
698                 curr = rtc.datetime()
699                 rtc.datetime((y, mo, da, curr[3], curr[4], curr[5], curr[6], curr[7]))
700                 display_dirty = True
701
702         # send response
703         # after
704         resp = "HTTP/1.0 200 OK\r\n" \
705                "Content-Type: text/html\r\n" \
706                "\r\n" + web_page()
707         cl.send(resp)
708         cl.close()  # <- this is the missing piece
709
710     except OSError:
711         pass
712
713     # --- Alarm & Time Management ---
714     dt = rtc.datetime()
715     hr, mi, se = dt[4], dt[5], dt[6]
716
717     if alarm_triggered:
718         handle_active_alarm()
719         update_display()
720         utime.sleep_ms(200)
721     else:
722         if alarm_snoozed and utime.ticks_diff(utime.ticks_ms(), snooze_end_time) > 0:
723             alarm_snoozed = False
724             update_display()
725
726         if alarm_enabled and not alarm_snoozed and (hr, mi) == tuple(alarm_time):
727             alarm_triggered = True
728             if speaker_enable.value():
729                 speaker_enable.value(0)
730                 fm_radio.SetMute(True)
731                 fm_radio.ProgramRadio()
732             draw_wakeup_screen()
733
734         if se != last_sec:
735             last_sec = se
736             update_display()
737
738         # FM radio screen controls
```

```
739         if screen_mode == 2:
740             if btn_menu.pressed():
741                 screen_mode = 0
742                 fm_radio.SetMute(True)
743                 fm_radio.ProgramRadio()
744                 speaker_enable.value(0)
745                 update_display()
746             elif btn_inc.pressed():
747                 new_freq = fm_radio.Frequency + 0.2
748                 if new_freq > 108.0:
749                     new_freq = 87.5
750                 fm_radio.SetFrequency(new_freq)
751                 fm_radio.ProgramRadio()
752                 update_display()
753             elif btn_dec.pressed():
754                 new_freq = fm_radio.Frequency - 0.2
755                 if new_freq < 87.5:
756                     new_freq = 108.0
757                 fm_radio.SetFrequency(new_freq)
758                 fm_radio.ProgramRadio()
759                 update_display()
760 
761         # Time-zone selection screen controls
762         elif screen_mode == TZ_MODE:
763             if btn_inc.pressed():
764                 tz_index = (tz_index + 1) % len(time_zones)
765                 update_display()
766             elif btn_dec.pressed():
767                 tz_index = (tz_index - 1) % len(time_zones)
768                 update_display()
769             elif btn_set.pressed() or btn_menu.pressed():
770                 screen_mode = 0
771                 update_display()
772 
773         # Global menu entry
774         if btn_menu.pressed():
775             menu_loop()
776             update_display()
777 
778     utime.sleep_ms(5)
779 
```

ECE 299 Report: Checklist

Format:

1. **Body-text** uses consistent font type and size
   YES NO □ NOT SURE □

2. **Titles** and **Headings** are easy to distinguish from the body text
   YES NO □ NOT SURE □

3. **Pages** and **equations** are numbered
   YES NO □ NOT SURE □

4. **Symbols** used in equations are defined the first time you use them
   YES □ NO □ NOT SURE
   There wasn't anything relating to this.

5. Information in paragraphs has been made easy to read whenever possible by way of lists, figures, tables etc.
   YES NO □ NOT SURE □

6. All information derived from external sources is properly and fully cited in-text, using IEEE citation style
   YES NO □ NOT SURE □

7. A properly formatted **IEEE-style References list** closes the document:
   YES NO □ NOT SURE □

Figures and Tables:

1. Whenever possible, images are original, not directly copied from a source
   YES NO □ NOT SURE □

2. Figures and Tables are referred to in text (e.g. See fig 1; see Table 2, etc.) *before* they appear on the page
   YES NO □ NOT SURE □

3. Each image and table includes ***sequentially numbered captions*** (in Times New Roman font, size 10) that explain what the image or table is about
   YES NO □ NOT SURE

4. **Graphs** provide clearly labeled **axes**, with accompanying **legend**.
   YES NO □ NOT SURE □

5. **Images** are completely readable without magnification (i.e. zooming in).
   YES NO □ NOT SURE □

Style

1. Consistently uses either **first person** or **third person** narration.
   YES NO □ NOT SURE □

2. Supplies **written descriptions** of figures and tables to clarify their relevance to the reader.
   YES NO □ NOT SURE □

3. Makes **direct reference** to each figure and table in the presented description (e.g. *See Fig. 1; as shown in Table 2,* and so forth).
   YES NO □ NOT SURE □

4. Includes **signal phrases** (e.g. *in addition; however; in contrast; as a result,* etc.) to clearly map the logic of the report.
   YES NO □ NOT SURE □

5. Maintains a **formal**, though not **"stuffy"** or **"inflated"** tone; it avoids informal **contractions** (*won't; can't;* etc.).
   YES NO □ NOT SURE □

6. Uses **clear active sentences,** i.e. state the main topic or **subject** at the beginning of the sentence, closely followed by an **action word—**a verb.
   YES NO □ NOT SURE □

7. Maintains **agreement between subjects and verbs** (i.e, a singular/plural sentence topic is followed by a singular/plural verb).
   YES NO □ NOT SURE □

8. Uses **simple words** when simple words will do the job, following the **plain language** principles.
   YES NO □ NOT SURE □

9. Eliminates **phrases** (e.g. *in view of the fact that*) when a **single word** (e.g. *because*) would do instead.
   YES NO □ NOT SURE □

10. Links **pronouns** (stand-in words like *this, they,* or *it),* back to a specific noun (i.e. an object, concept, or person) in the previous phrase or sentence.
    YES NO □ NOT SURE □

11. Supplies **grammatically parallel** lists and headings, meaning each item in a list and each heading takes the **same grammatical form** (e.g. all verb phrases or all noun phrases).
    YES NO □ NOT SURE □

12. Has been carefully proofread to ensure **punctuation** (commas, colons, semi-colons, and periods) aligns with Standard English conventions to enhance accuracy and clarity.
    YES NO □ NOT SURE □

REFLECTION: what improvements did you make to your report because of completing this checklist? Identify 2 specific changes.

This page intentionally left blank.