Project Braveheart

ECE 299

University of Victoria

Lab B03
Group 02

Date Submitted:
August 14th 2022

Group Members:
Connor Wiebe V00929520
Matthew Ebert V00884117

Executive Summary

The purpose of this report is to document the creation and testing process of the clock radio device that was constructed. With the wide scale availability of small electronics in recent years and ease of creating them with devices such as Raspberry Pi and Arduino products, the goal of this project was to create from scratch a clock radio. The theme of the casing of the clock radio was also set to resemble that of an animal.

The hardware specifications of this clock radio are that it must be created using a Raspberry Pi Pico board, coded using micropython through the Thonny IDE, communicate with an OLED display using SPI and the printed circuit board created for the project must be ordered through a commercial manufacturer.

The software requirements of the clock are that it must be able to tell the time in both 12 and 24 hour time, the clock must be able to be set, it must be able to set and trigger an alarm as well as have the ability to be snoozed and the alarm toggled on/off, all through the use of user interfaces. The clock must also have radio functionality, ranging from playing radio output on a speaker, changing the volume of the radio, tuning to at least one different radio station, and having the ability to display radio channel information to the user.

For the design, it was decided to add the fact that the clock radio would ‘run away’ from the user once the alarm had been triggered. This added additional constraints, such as the clock must be powered using AA batteries, it must be a mobile and self contained system and the snooze button must be large enough so that a thrown object would be able to easily snooze the alarm.

To complete and achieve all of these goals, the solution first involves using a turtle design. This design choice is deliberate since the shell of a turtle proves to be an excellent snooze button due to the large form factor, while allowing ample room for hardware components such as the PCB inside the ovular shape of the shell. The small mobile shape also allows the device to easily incorporate the mobile component by placing a gear system at the bottom of the enclosure. The display was chosen to be placed as the ‘head’ of the turtle and the user interface was made as minimal as possible in order to maintain the animal aesthetic of the project.

There were a few challenges throughout the creation of this project, such as noise from the circuit caused primarily by the AA batteries that were required to ensure that the clock radio was mobile. These challenges were overcome through the use of transistor circuits to turn the speaker off from a hardware standpoint, reducing the ambient buzzing produced beforehand. The software of the clock was also extensively tested to eliminate bugs in the code to ensure that the clock radio functioned correctly at all times.

Through the completion of this project, all requirements were met, both those set by the client and those created through the design process, resulting in a fully functioning, well made, aesthetically correct turtle clock radio that has been dubbed Project Brave Heart.

Table of Contents

List of Tables

List of Figures

Introduction

With the development of microcontrollers and compatible peripheral devices, developing small scale, embedded electronic devices has become more accessible. These devices can be diverse and powerful given the variety and ability of these components. To prove our understanding and skills with these devices, a task was set to design a clock radio.

Project Goals

The goal of this project was to design and create a clock radio that is able to trigger an alarm and tell the time. The goal for the enclosure and aesthetic of the clock radio is to resemble an animal. For the solution to the task, it was decided to create a mobile alarm clock in the shape of a turtle. This design would include regular radio and clock functionality with one addition: when the alarm was triggered, the turtle would move away from the user to force an active chase to snooze it. This feature was designed to improve the wakefulness of the user each time the snooze button was hit.

Project Constraints

Table 1: List of project constraints

No.	Constraint
a	- Use a Raspberry Pi Pico board (Pico) as the intelligence handling device
b	- Use SPI to communicate between the Pico and the OLED display
c	- Program the Pico board using Thonny, a micropython IDE
d	- The printed circuit board must be ordered commercially
e	- The final demonstration of the project must occur during the week of July 25th

Requirements

There are 2 main required subsystems for this project, each having their own set of expectations, they are:

Clock Functionality

Table 2: List of project clock requirements

No.	Requirement
1	- Set and edit the time through a user interface (UI)

2	- Display the time in 12 or 24 hour time as set by the user
3	- Set and edit an alarm through a UI
4	- Trigger the alarm that has been set by the user
5	- Snooze and turn off the alarm through a UI

Radio Functionality

Table 3: List of project radio requirements

No.	Requirement
6	- Tune to at least one radio channel through a UI
7	- Play output on a speaker
8	- Be able to adjust volume output through a UI
9	- Display radio channel information to the user

Design Considerations

The solution to the goal adds additional requirements to the system. These are listed in Table 4.

Table 4: List of project additional requirements

No.	Requirement
10	- Battery powered (AA)
11	- Mobile and self contained
12	- 3D Printed shell and components
13	- Mechanically actuated shell button (for snooze functionality)

Project Planning and Timeline

In terms of the overall project planning, the project was mainly split between software and hardware elements, with Wiebe taking on the software elements and Ebert taking on the hardware. Both team members came together when testing the clock however, to ensure that it was able to pass the tests presented so that the tasks could be called complete. The more in-depth roles for each member is laid out in the Gantt chart with the name or names of the group members working on the specific task beside the bar.

Gantt Chart

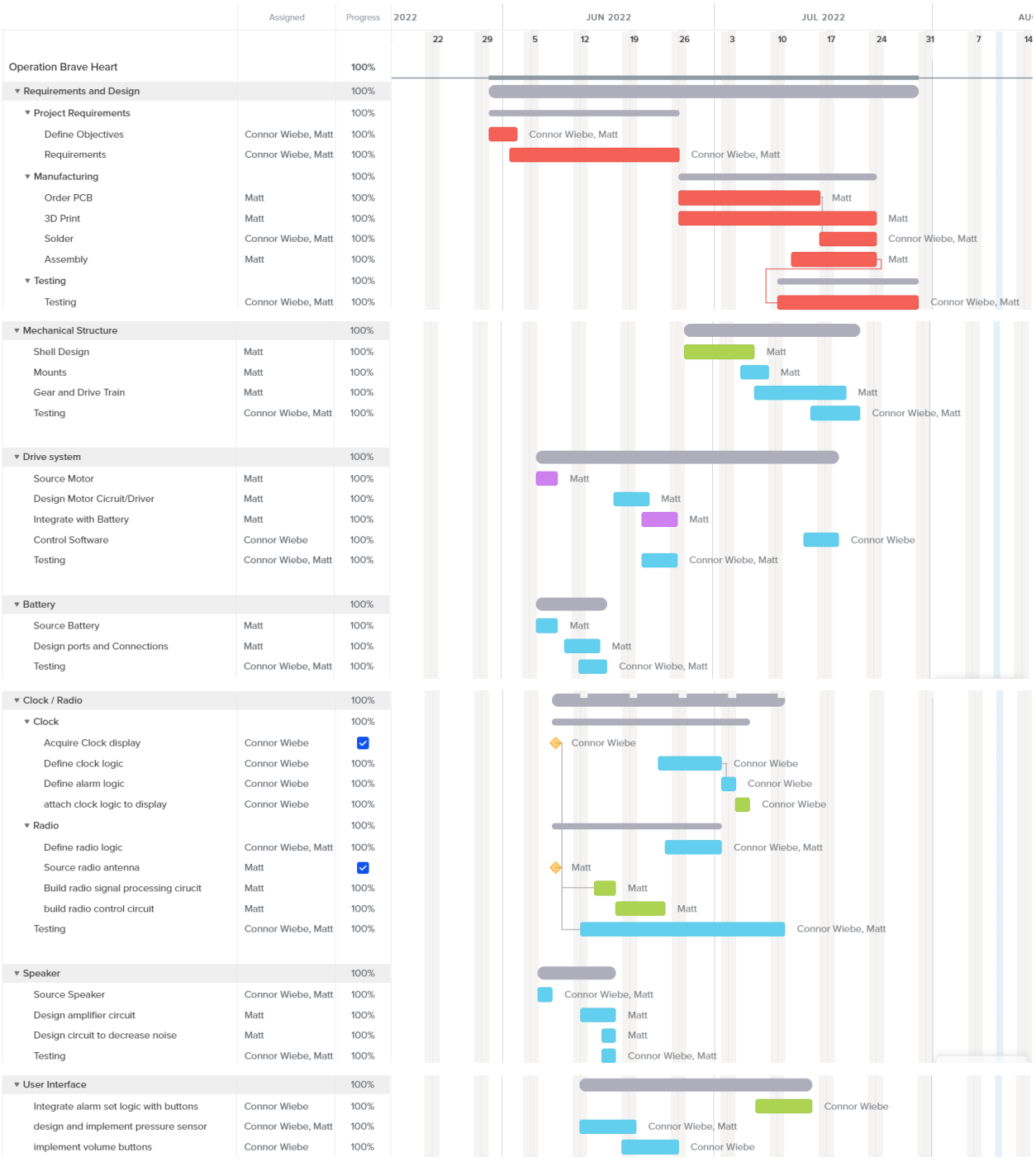
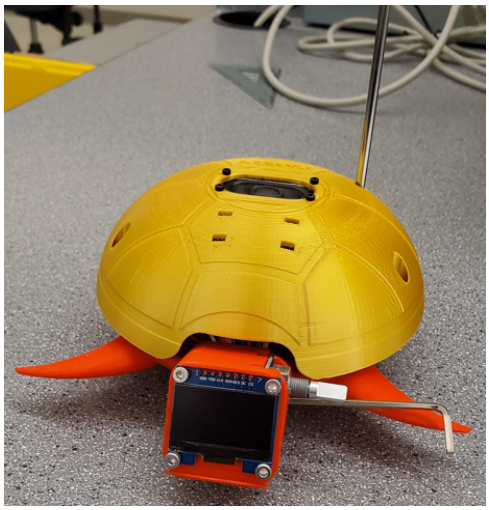
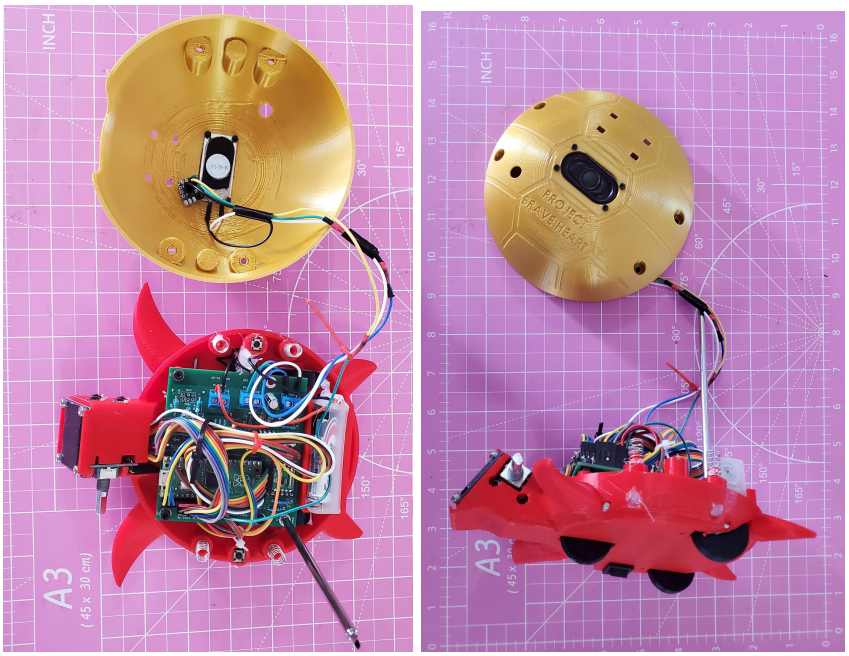


Figure 1: Project timeline Gantt Chart

Implemented Design

The completed design for project Brave Heart is shown in Figure 2. The product is designed to resemble a turtle with a 3D printed shell and body. A OLED display is mounted on the ‘face’ of the turtle and a speaker on the top of the shell. An encoder and several button are mounted on the side of the head which act as the UI. An additional push button system is worked into the shell itself and can be activated by pushing on the shell. All electrical equipment including the batteries are housed within the body of the turtle. The radio antenna protrudes through the top of the shell. A further breakdown of each sub system is found in the following sections.





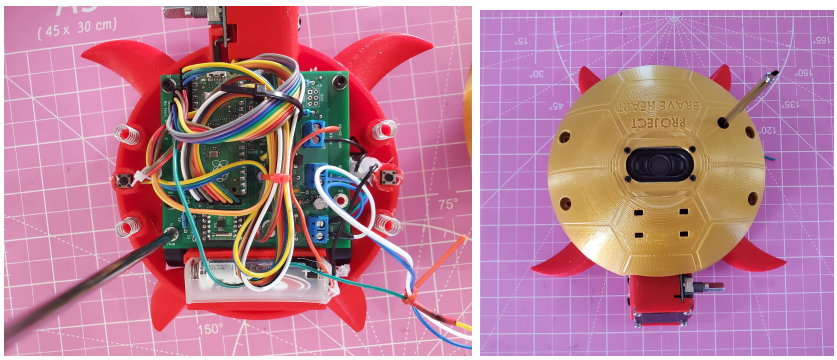


Figure 2: Demonstrated version of Project Braveheart

Electrical Systems

The electrical system of project Brave Heart consists of a

- Raspberry Pi Pico
- Custom PCB
- Audio Amplification Circuit
- FM radio module
- Boost Converter
- DC motor

Circuit Design

The circuit schematics detailing the use of these items is shown in Figure 3.

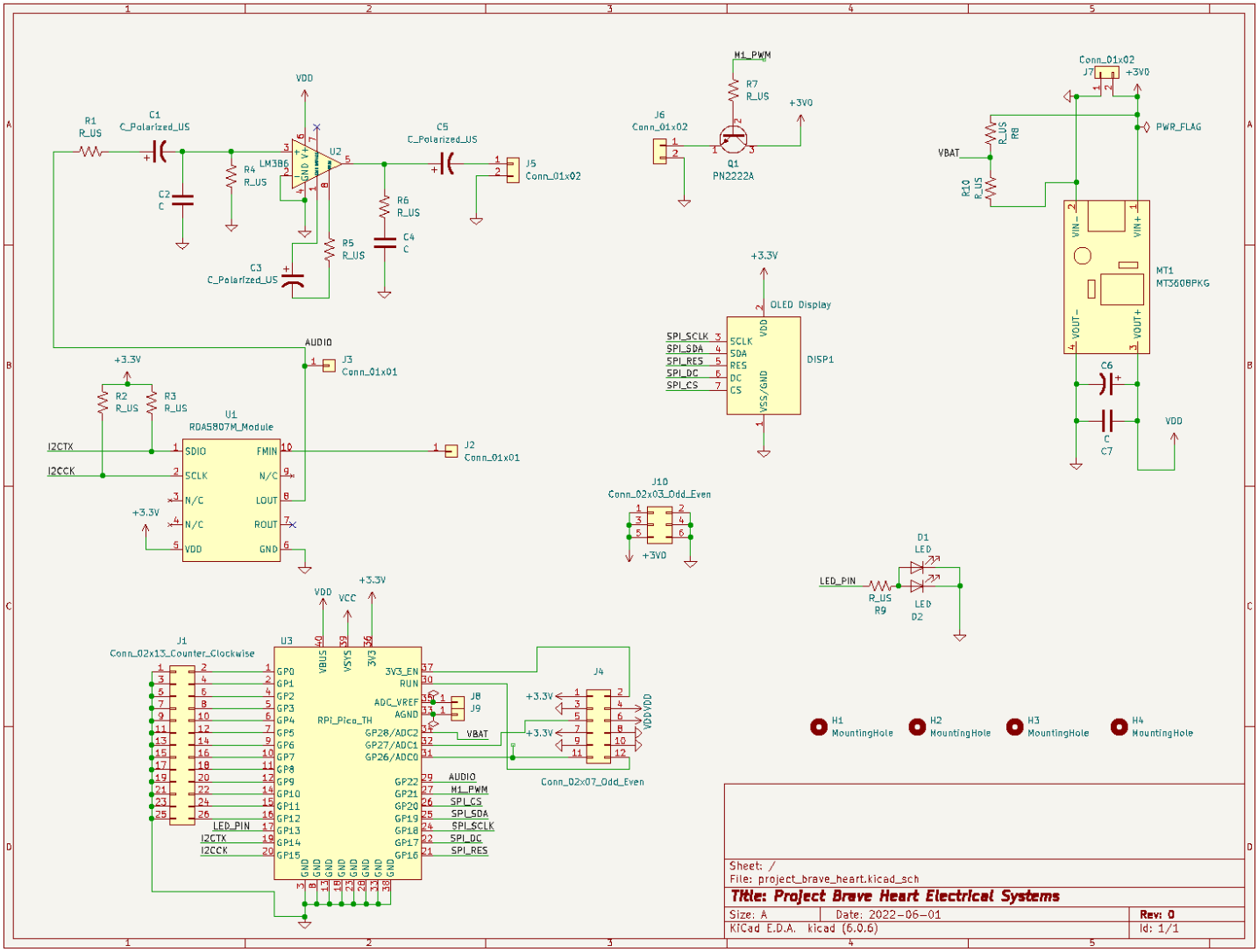


Figure 3: Circuit schematic for project Brave Heart

The Pico acts as the epicenter of the circuit and integrates all the functionality with its programmed control. The GPIO pins on the pico are used for button inputs, encoder inputs, display outputs, SPI interface, and motor control. The Pico is powered from the boost converter output (VDD) which provides a consistent converted 5.0V. This voltage was chosen because it allows for 5V peripherals to be powered and is 0.5V under the maximum voltage which gives a 10% safety margin in case of surges [7].

The audio amplifier circuit is powered by VDD from the boost converter. Some of the circuit was copied from the recommended circuits on the LM386 data sheet. The components C3 and R5 were added to allow for a custom audio in the circuit. For the implementation, C3 had a value of 10uF and R5 was shorted to give a gain of 200 [7]. The audio circuit takes input from either the FM radio or a PWM pin from the Pico. Only one of these inputs may be active at any time to achieve clear audio.

An additional band pass filter was added immediately before the audio amplifier. The values were calculated using the RC cutoff frequency formula.

$$f_c = 1/(2\pi RC)$$

The band pass was tuned to a range of 20Hz to 15kHz using the following values. The reasoning behind this filter range can be found in the Testing and Validation section of this report.

Table 5: Audio bandpass filter component values.

Component	Value	Unit
R1	800	Ohm
C1	0.1	uF
R4	100	Ohm
C4	10	uF

The RDA5807M radio module and the OLED display were connected through a SPI to the Pico. They are both powered with 3.3V from the Pico. This power set up was configured to achieve maximum isolation between the power for the FM radio and the motor. The FM module audio output connects to the audio amplifier circuit from a mono output. The OLED display connects to 7 pins on the Pico for power and control.

The DC motor is powered directly from the battery voltage, 2 AA in this case which provides between 3V and 2.7V. The motor is controlled with a simple bjt circuit. This circuit connects the DC motor to the battery voltage when a signal from the motor GPIO pin on the pico is set to HI. The circuit has space for a resistor for biasing purposes; however, in the implemented design this resistor was shorted so the BJT would become saturated when 3V3 volts was applied to the base. This circuit also allows for speed control of the motor using a PWM signal from the Pico.

The main power circuitry includes batteries, a boost converter, and capacitors. Three voltage values are used in this circuit: 3V0 (battery voltage), VDD (5V), and 3V3. The battery voltage connects to the inputs of the boost converter and the DC motor. The output of the boost converter powers the audio amplifier circuit and the Pico board. The 3V3 voltage pin on the pico (converted from the 5V system input) powers the other peripheral devices. 2 Capacitors are connected to the output of the boost converter for power storage and filtering purposes. The values of these components are

- C6 = 470uF
- C7 = 0.1uF

The explanation for these values can be found in the Testing and Validation section.

Other circuits on the PCB include a battery level detector and LED indicator circuit. Several pin connectors were also included to aid in any expansion and modification of project brave heart. These pins connect to unused GPIO pins on the Pico and also provide connections for voltage and ground sources.

The complete list of component values is shown in Table 6.

Table 6: Project Brave Heart circuit component values

Label	Value	Unit
C1	0.1	uF
C2	10	uF
C3	10	uF
C4	0.05	uF
C6	470	uF
C7	0.1	uF
R1	800	Ohms
R2	2.2	kOhms
R3	2.2	kOhms
R4	100	Ohms
R5	-0	Ohms
R6	10	Ohms
R7	-0	Ohms
R8	1	MOhms
R9	1	kOhms
R10	1	MOhms

PCB Design

The completed PCB design for project brave heart is shown in Figure 4. The PCB is 81.28 x 76.2 mm and has 2 Layers.

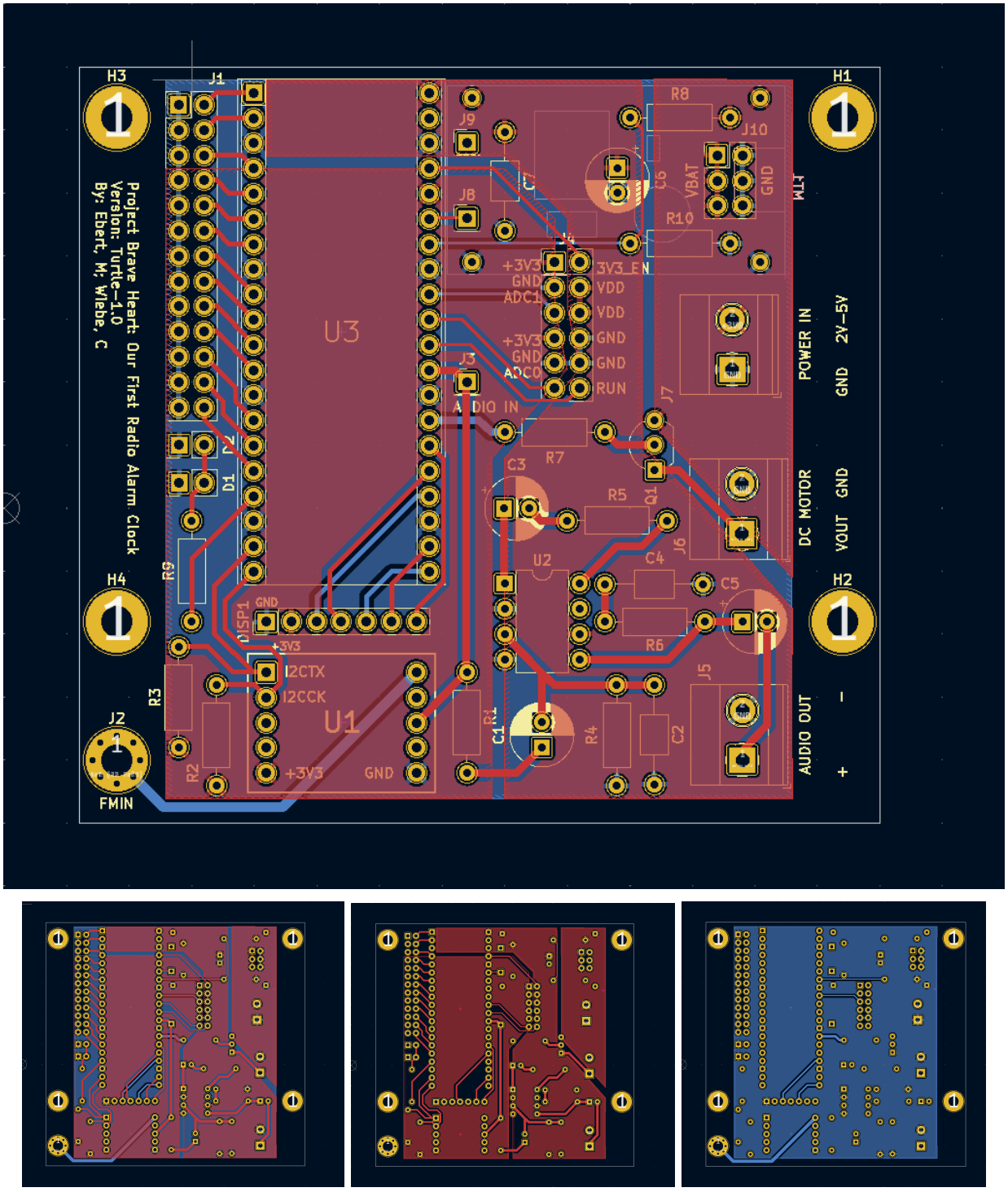


Figure 4: 2 Layer PCB Design for project Brave Heart

The bottom layer consists mostly of a ground fill with a few traces including the connection to the radio antenna, a large trace to allow for maximum signal. The upper layer contains 3 fill region, 1 for each voltage source, and the majority of the connection routing.The layout of the PCB places the radio modules and audio amplifier circuit at the bottom, the Pico and extra pin

connectors in the upper left, and the motor and power input circuits in the top right. This placement was selected to keep the noise sensitive components (radio module and audio amplifier) as far away from the boost converter as possible. Efforts were made to create large fill regions on the top and bottom PCB layers to create a filtering effect and reduce EMI and keep traces from voltage sources under 3:1 length-width ratio [1]. Screw terminal blocks were positioned on the same side to aid in assembly and cable management. The manufactured design is shown in Figure 5.The assembled PCB is shown in Figure 2.

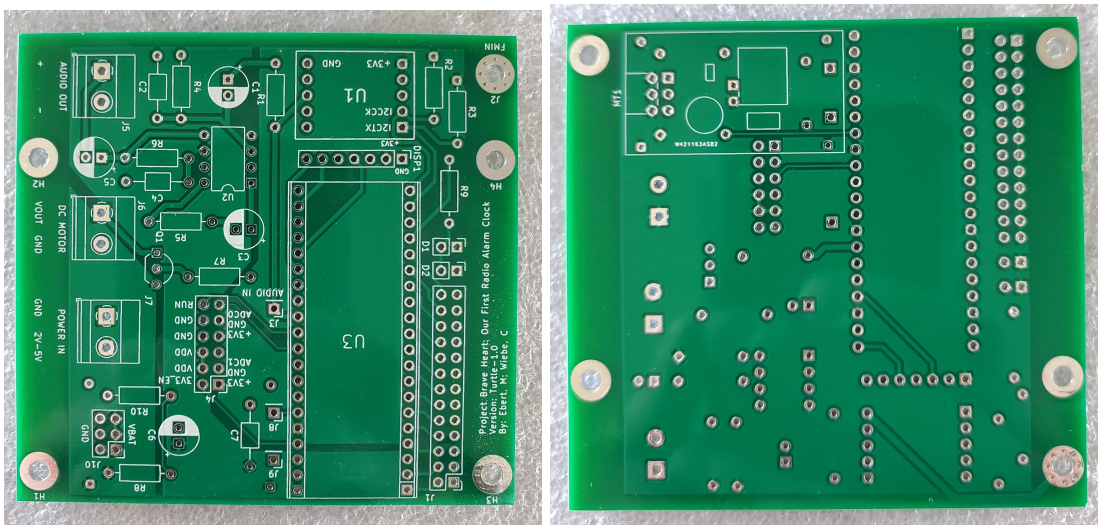


Figure 5: Manufactured PCB for project Brave Heart

Software and User Interface

In order to achieve the expectations of adjusting the volume, muting the output of the speaker and automatically seeking to the next radio station, push buttons were used on the user interface. Incrementing and decrementing the volume was done using the bh_fm_radio.py library, shown in figure A.4 in appendix A, by adding new functions to the library, while seeking was accomplished in a similar way and can be seen in the ‘Seek’ function in figure A.4.

The snooze feature of the alarm clock also used push buttons; however, they were used in a different way. The enclosure shell of the ‘turtle’ was attached to bolts and a spring, which allowed it to move and be pushed down simply by pressing on the shell of the alarm clock. By placing buttons at specific contact points inside the shell of the clock, the snooze function was able to be implemented while being integrated with the animal design theme of the clock.

All of the push buttons utilized the same debounce function which can be seen in main.py in figure A.8 in appendix A.

To achieve the expectations to set and edit the time and alarm, change between 12 and 24 hour time, turn the alarm feature on and off, and tune to radio stations, a rotary encoder was used in conjunction with a library found on github which can be seen in figure A.5 and A.6 in

appendix A. The encoder was powered by the 3V3 out pin on the pico board 3 GPIO pins and 1 ground pin on the pico board as required and seen in the datasheet for the rotary encoder in [4]. In order to complete all of these very different tasks with one user interface device, a page feature was used for the main code of the alarm clock as seen in figure A.8, allowing the user to cycle between options on the two different pages with the rotational aspect of the encoder and click on the options using the button attached to the encoder. This allowed this one user interface device to complete a multitude of different tasks, while keeping the interface of the clock minimal. In order to do this, the minimum and maximum values for the encoder had to be updated as the page changed, which is achieved through the 'encoder.set' commands seen in the code of main.py in figure A.8. By using one rotary encoder for all of these aspects, the clock was able to maintain a sleek design and maintain the appearance of an animal, as well as reduce costs by simply using one encoder for all of these features instead of a multitude of push buttons.

To display the time and options to the user, a 1.3" OLED display was used. This display was integrated into the front of the Clock, to resemble the 'head' of the turtle. The display was interacted with by the board using SPI and the display library shown in figure A.9. The display was connected to pins 16,17,18,19 and 20 on the Pico board for the reset, data, serial clock, serial data and clock set pins respectively as well as the 3V3 out pin and a ground pin. The smaller display was chosen as it fits better with the smaller design of the enclosure for this project and allows the turtle's 'head' to look more natural.

All user interface devices including the rotary encoder and push buttons (excluding those for the snooze feature) were mounted to the side of the 'neck' of the turtle, putting them in a convenient location to press, while maintaining the form factor and look of the turtle.

Software was written to obtain RDS information from the current radio channel of the RDA module. When prompted, the RDA5807 module sends 4 bytes of data at regular intervals over the serial interface. The written program reads these bytes from the serial connection and decodes them to find radio information. The software checks the quality type of data received from the RDA module and stores the station name for display on the OLED. The station name is sent 1 letter (byte) at a time; thus, this function must be called repeatedly to display the full name. Other information can be found from the RDA module RDS registers but this functionality has not currently been programmed into the device.

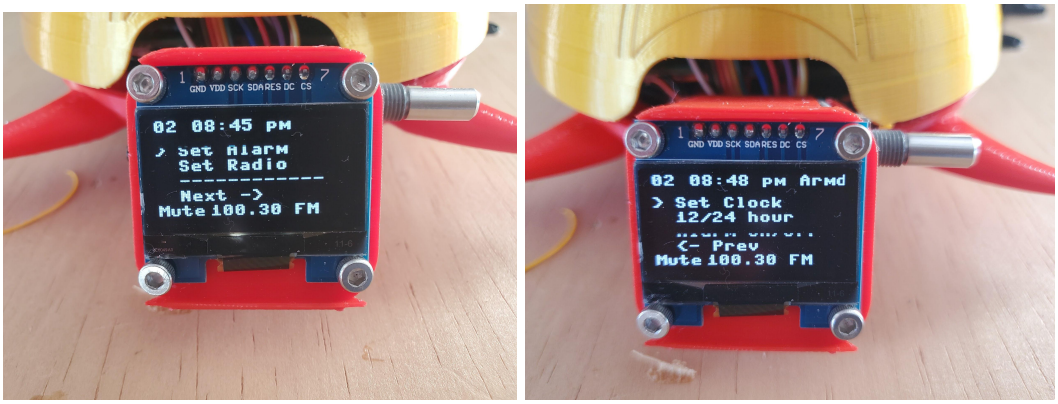
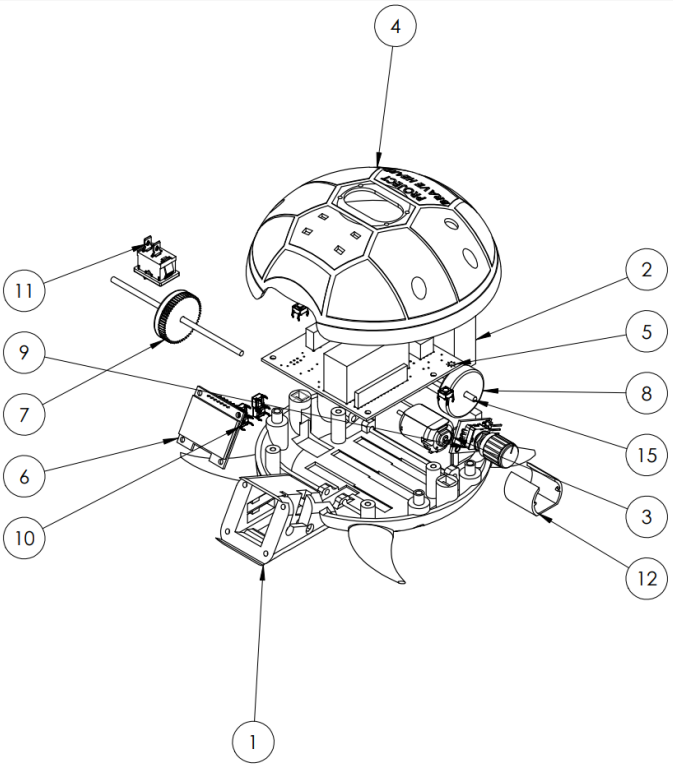


Figure 6: Implemented display for project Brave Heart

Mechanical Elements and Enclosure

The main structural components were designed in SolidWorks and 3D printed. SolidWorks drawings of the major design components are shown in Figure 7 to 9. The major mechanical assemblies include the motor gear train and the shell push button systems.

ITEM NO.	PART NUMBER	QTY.
1	bh_bottom_turtle	1
2	Battery Pack	1
3	bh-motor	1
4	ph-shell	1
5	bh_PCB	1
6	bh_OLED1_3	1
7	bh_wheel_gear	1
8	bh_wheel	2
9	bh_encoder	1
10	pushbutton	6
11	Power Switch	1
12	bh_motor_slidelock	1
13	motor_gear	1
14	speaker	1
15	3mmaxel	2



SIZE A | DWG. NO. pb_braveheart_assem | REV
SCALE: 1:2.5 | WEIGHT: | SHEET 3 OF 3

Figure 7: BOM exploded View of project SolidWorks assembly

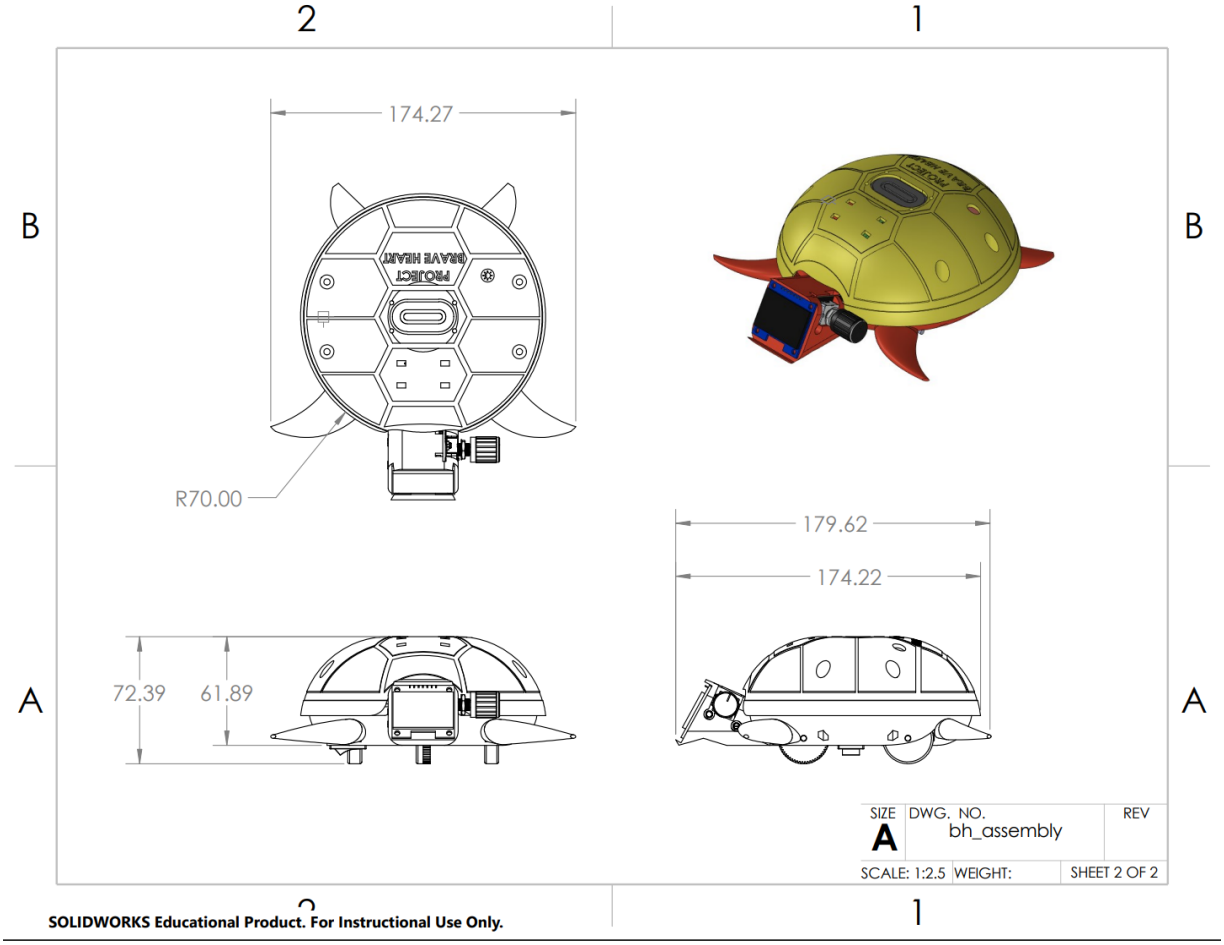
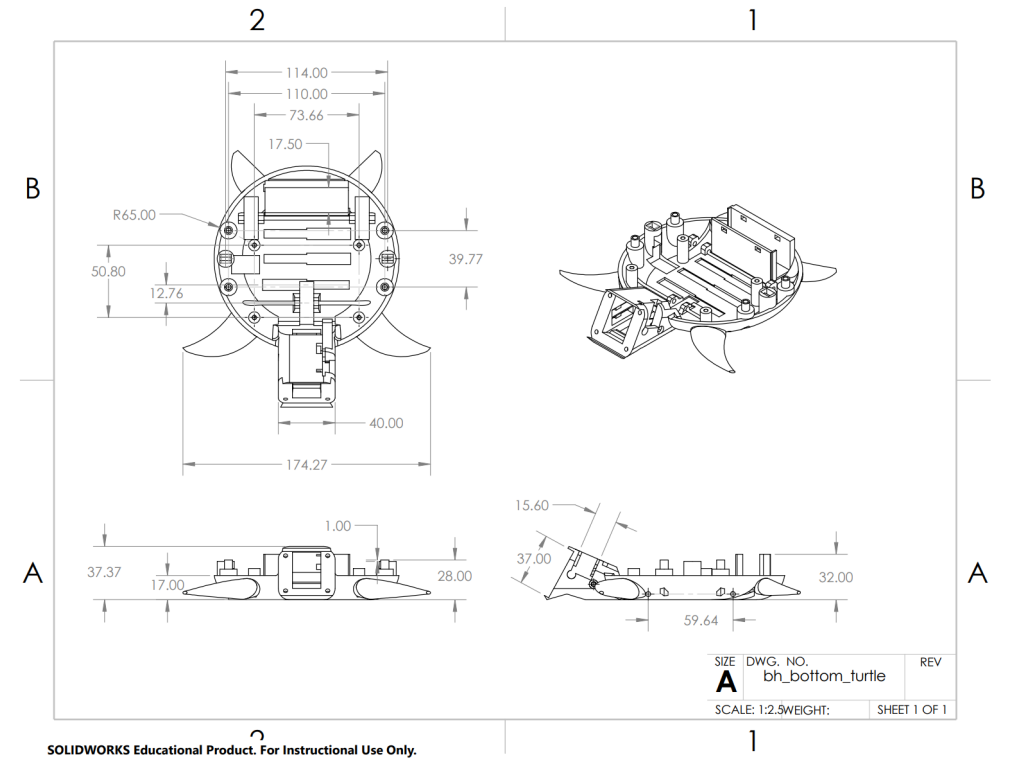


Figure 8: Assembly view of project Brave Heart

The motor gear train uses spur gearing with a gear ratio of 10:58. This reduces the rpm from the motor and increases the output torque. The gears were 3D printed and mounted on a 3mm axel and the motor shaft. No bearing or guides were needed.

The push button system makes use of 4 extension springs and fasteners with lock nuts. The shell is secured to the body with the fasteners; clearance was given so the shell could move along the screw shaft with 1 DoF. The extension springs were fitted on the screw shaft to provide clearance between the shell and button when pressed. Upon a user pressing on the shell, the springs compress causing the shell to press the push button and send a signal to the Pico controller.



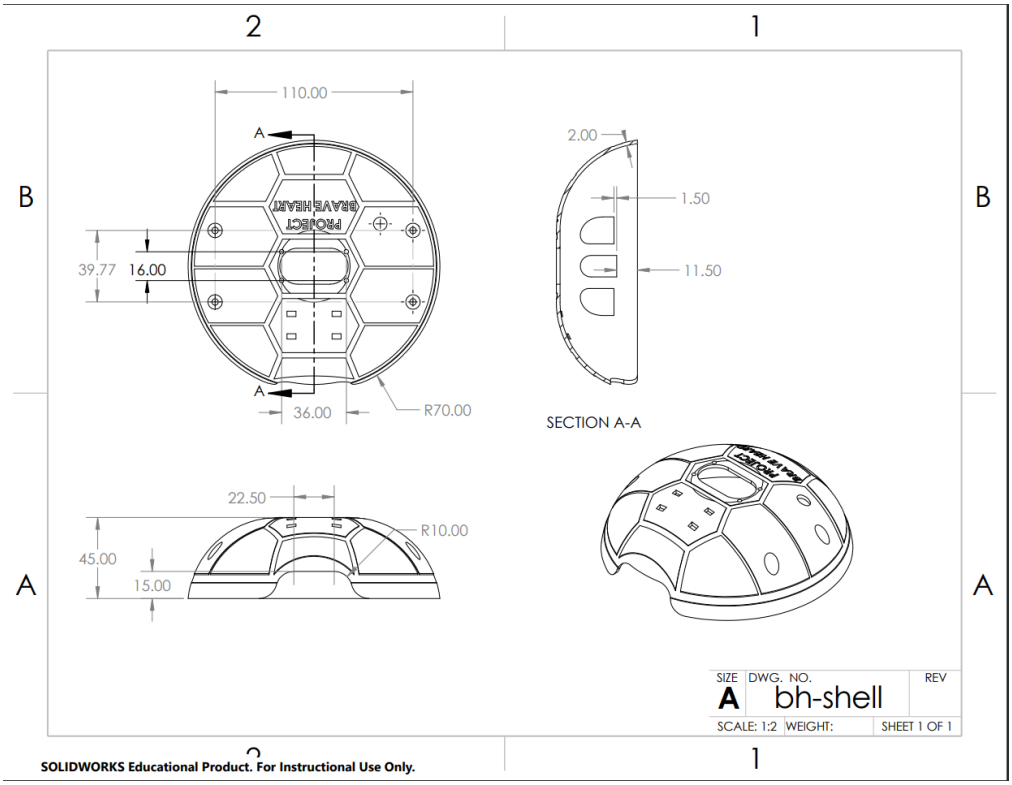


Figure 9: Major 3D printed components of project Brave Heart including the base and the shell.

Electrical and off the shelf components integrate with the body or shell using M3/2 fastener, hot glue, or a press fit. Hot glue was used to insulate several exposed conductors to prevent

accidental shorts. Cable management was achieved with zip ties and careful routing around extrusion in the body of the turtle. Figure 2 shows this in detail.

Bill of Materials and Cost Analysis

Table 7: BOM for OTS components used in the implementation of project Brave Heart

Label in schematic	Component Description	Part Number	Cost($) / quantity	Source of Cost information
U1	Radio Module/ surface mount board	RDA5807M	3.00/1	[2]
U3	Pico board	Raspberry Pi PicoBoard	6.00/1	[2]
DISP1	1.3” OLED display	1.3” OLED Dsiplay SPI	4.00/1	[2]
LM386	Operational amplifier	LM386N-4	1.50/1	[2]
MT1	Boost Converter	MT3608	0.22/1	[6]
Q1	NPN transistor	PN2222A	/1	[2]
J5	Speaker	Speaker 2W 8Ω	4.00/1	[2]
N/A	Rotary Encoder	KY-040	1.00/1	[5]
N/A	Push button	N/A	0.25/6	[2]
N/A	3D printed materials	N/A	3.50/in^3	[3]
N/A	Pin header	N/A	0.25/1	[2]
N/A	Socket header	N/A	1.50/1	[2]
N/A	Printed Circuit board	N/A	$5/5	PCBWay Reciept
N/A	Engineering time	N/A	6.22/hour	[4]

Testing and Validation

During development of project Brave Heart, testing and subsequent modification were made to the circuit, software, and hardware designs.

Software and UI

Throughout the testing process, many different subsystems were tested. One of the first that was tested was the radio module circuit to ensure that the setup was correct and that the vol_up() and vol_down() functions that were defined in the bh_fm_radio.py code were working properly. This circuit can be seen in figure 10. To test this, a button was then connected to the circuit and the interrupt request handler was set to trigger each of the functions and the radio volume was printed after the function was completed to ensure it was being properly updated.

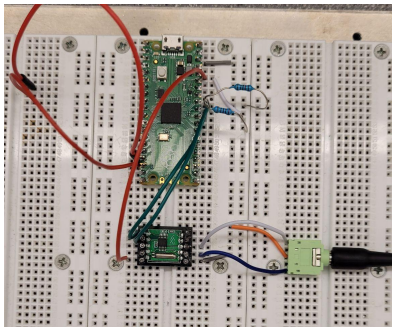


Figure 10: circuit setup to test the radio module using a 3.5mm jack

Another step in the testing process was interfacing the encoder with the SPI display to ensure that the button was not bouncing as well as that the display was being updated when the encoder was being turned. This code can be seen in the main while loop of main.py and an image of the testing process can be seen in Figure 11. To ensure that the button was not bouncing, it was pressed for both extended periods of time as well as very quickly, and when it only read one button press, the test was complete. To ensure that the rotational values were being properly read, the encoder was turned up and down to show that only the available values were able to be reached.

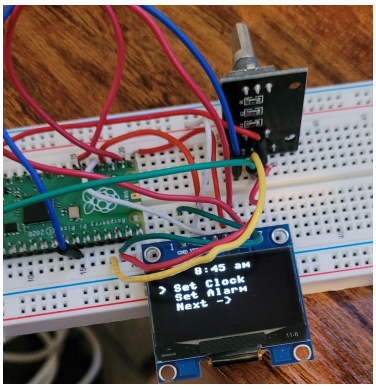


Figure 11: circuit setup to test interfacing the encoder with the SPI display

Another test that was performed was to ensure that the radio could be run at the same time as the motor on the clock radio. The code for this test is shown in figure A.1 of appendix A, and it was determined that the battery was able to run both processes at the same time. To perform this test, the code was saved to the Pico board as main.py and utilized the battery to run. The clock was then held as the code ran to ensure that the clock did not fall.

After the radio and motor test on the battery, it was discovered that the speaker played an ambient buzz caused by the noise of the battery circuit. This was rectified by adding a transistor circuit to the clock that is updated along with the mute command, resulting in the speaker not receiving power when the mute is activated and updating the pin value to push the transistor into saturation and give the speaker power when the mute feature is deactivated. This process can be seen in the 'SetMutePin' function of figure A.4 and the pin value being updated can be seen in both the 'SetMute' and 'toggle_mute' functions.

To test the timer to increment the clock as well as update the information coming from the radio module, a short code was created and can be seen in figure A.3, which allowed us to determine the unit of time that the frequency used as well as the fact that multiple timers could be used simultaneously.

To test multiple functions of the clock working together, the main.py code was run with the clock timer set to a lower frequency. The result of this test can be seen in figure 12. To perform this test, the encoder was turned to the 'next' option and the encoder button was clicked, then, the button was clicked again on 'set radio' and tuned to 100.3 by rotating the encoder and pressing the encoder's built in button. The result of this test was that the encoder button was found to be successfully debounced and the encoder values were being set properly so that only valid radio frequencies were able to be tuned to. Overall, this test yielded positive results since the code was functional and the radio was able to be set while multiple functions of the clock were working simultaneously.

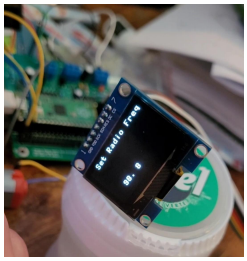


Figure 12: A test of the set radio functionality along with the set encoder functionality

In order to test updating the encoder values, a short code was created to ensure that the values were being properly updated, which can be seen in figure A.2 of appendix A. this test was necessary since the encoder value has to be updated frequently in order to be able to access all of the options for the clock. This code was run on the Pico board and the encoder

was turned up and down to ensure that no outside frequencies were able to be reached for both sides of the decimal place on the radio.

To test that the radio module was able to get the radio station information, the print statement at the bottom of figure A.4 was uncommented while the clock was moved to a location with a strong radio signal, this resulted in the console printing out the radio information correctly over time, resulting in a successful test.

Finally, to test the full functionality of the clock working together, the code in figure A.8 in appendix A was uploaded to the Pico board and a series of tests were run. First the clock was updated using the encoder near edge cases such as 12:00 am and pm to ensure that the am/pm feature was being correctly updated as well as to ensure that there was no bouncing on the encoder button in the final product. Then, an alarm was set and triggered to ensure that the motor and radio were being properly updated when the alarm was triggered, as well as to ensure that the snooze button was working properly. The alarm was also tested near edge cases such as 12:00am and pm to ensure that the snooze function would properly add 5 minutes in these specific situations. Then, the time was updated to 24hr time and the same series of tests were run, except for this time the edge cases were tested around 23:59 and 0:00. The alarm on/off feature was then tested to ensure that the alarm could be disabled for both time modes of the clock and that the ‘Armd’ label on the main pages of the clock display were being updated accordingly. This immediately followed each of the alarm tests after the snooze button had been activated to ensure that the alarm could be properly de-activated. To test the radio functionality, the encoder was used to set the radio station, it was validated that the radio station had been properly updated by playing the same radio station on a laptop nearby. Lastly in this test, the volume up/down and mute buttons were pressed to ensure no bouncing as well as to ensure that the volume on the top left of the display was being updated accordingly as well as the ‘Muted’ label on the bottom right of the display.

Electrical Noise Filtering

Upon testing of the audio output from the FM radio, we found excessive noise. After inspection, 2 primary noise sources were discovered: the power output from the boost converter and the DC motor.

Boost converters use an inductor and high frequency switching to boost voltage from a lower source. This method produces a saw tooth like voltage signal as seen in the left image in Figure 13. By placing a large capacitor across the output terminals of the output capacitor, we mitigated the amplitude of this signal by storing and dispersing energy from the capacitor. The largest capacitor available was 470uF. This capacitor produced better results over smaller ones. The results are shown in the right image of Figure 13.

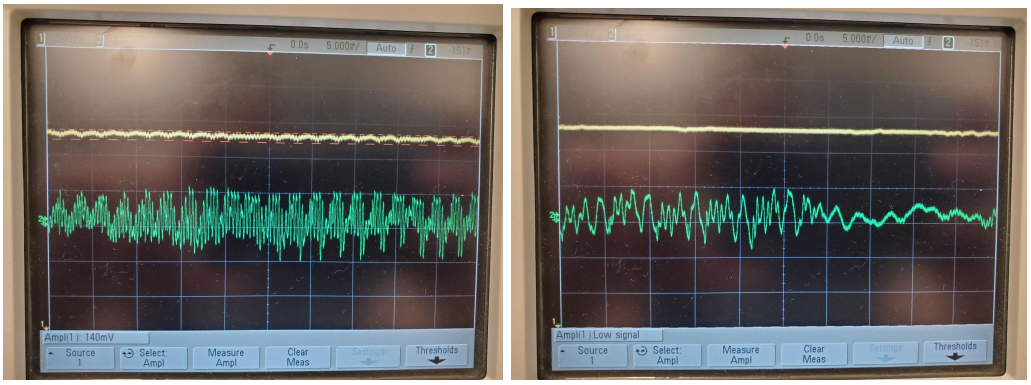


Figure 13: Result from power signal filtering without motor running (audio signal in green, power from boost converter in yellow); Left image without filtering; Right image with filtering; The subdivision for the green signal are 1V/.

When the DC motor was activated, additional noise was produced in the system. To compensate for this, a 0.1 uF capacitor was placed across the output terminals of the boost converter to filter high frequency signals. The effect of this capacitor can be seen on the yellow trace in Figure 14. Additionally, a band pass filter was added to the audio amplifier input to filter very low and high frequency signals. We were unable to detect the exact frequency of the noise filter; consequently, a trial and error method was used to clean the input signal. A pass frequency range of 20Hz to 15kHz produced the cleanest sound without distorting the audio signal. The results of the filter can be seen in the green trace in Figure 14.

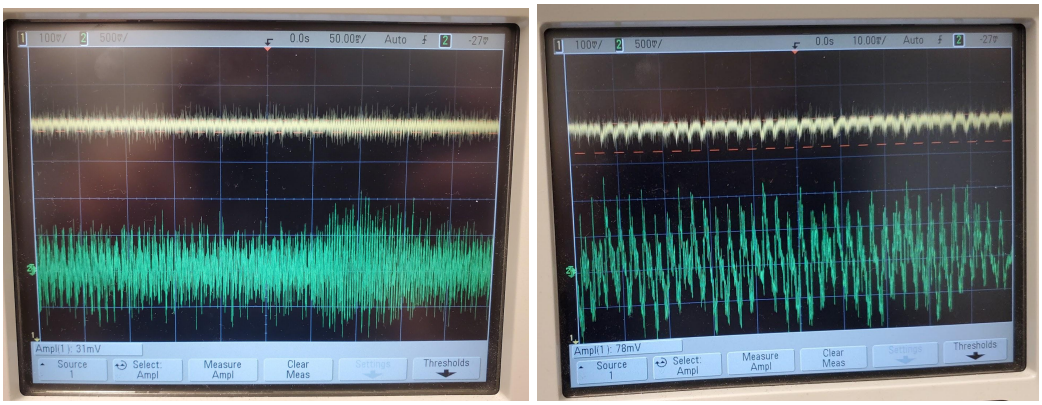


Figure 14: Result from power signal and audio filtering with motor running (audio signal in green, power from boost converter in yellow); Left image without filtering; Right image with filtering

FM Radio Testing

Radio Signal

The FM radio component of the design was tested in several locations. The test included rating the quality of the sound and detecting the radio channel information. The results are detailed in Table 6.

Table 8: Location radio testing results for project Brave Heart

Test No.	Location	Results
1.	Outdoor open space	Achieved clear audio signal and station information. Cleanest audio signal when antenna oriented correctly.
2	Indoor	Audio signal is present but can get noisy. Radio station information is inconsistent.
3.	Indoor shielded room	No audio signal or station information.

Radio Module Control

Software was used to control the radio module to test its reliability. Initial test showed that the radio could be tuned manual and the volume controlled precisely; however, during volume control testing it was noted that the RDA module ‘reset’ each time the volume increase was called. It was determined that this ‘reset’ occurred because each bit of the RDA module must be written everytime a register needs to be changed.

One issue was detected for which no solution has been developed currently. In the 03H register on the RDA module, bit 5 is required to be off in order to use the ‘seek’ functionality. However, this same bit is required to be on for direct tuning [8]. Attempts to change the state of this bit during a single operation failed. The module would stay in the ‘tuning’ or ‘seeking’ state regardless of the bits written to the registers. A full power cycle was required to change this. It is assumed a solution exists for this problem but was not developed over the scope of this project.

Code of Ethics

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

This was followed in the completion of the alarm clock as the clock was made to be very light so that if it were to fall on someone’s foot, it would not damage them.

2. practice only in those fields where training and ability make the registrant professionally competent

3. have regard for the common law and any applicable enactments, federal enactments, or enactments of another province

4. have regard for applicable standards, policies, plans, and practices established by the government or Engineers and Geoscientists BC

5. maintain competence in relevant specializations, including advances in the regulated practice and relevant science

This was demonstrated by utilizing skills from previous courses when constructing and coding the alarm clock, causing us to maintain previously learned skills and therefore, competence.

6. provide accurate information in respect of qualifications and experience

This was completed through the explanation of the alarm clock to the TA during the demonstration as well as through the creation of this report.

7. provide professional opinions that distinguish between facts, assumptions, and opinions

8. avoid situations and circumstances in which there is a real or perceived conflict of interest and ensure conflicts of interest, including perceived conflicts of interest, are properly disclosed and necessary measures are taken so a conflict of interest does not bias decisions or recommendations

9. report to Engineers and Geoscientists BC and, if applicable, any other appropriate authority, if the registrant, on reasonable and probable grounds, believes that:
a. the continued practice of a regulated practice by another registrant or other person, including firms and employers, might pose a risk of significant harm to the environment or to the health or safety of the public or a group of people;
or
b. a registrant or another individual has made decisions or engaged in practices which may be illegal or unethical

10. present clearly to employers and clients the possible consequences if professional decisions or judgments are overruled or disregarded

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

This is completed through the references on this report as well as the acknowledgement of both group members.

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

This was completed through the detailed completion of this report as well as competent documenting of the testing process.

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

This was completed through positive behavior between both group members.

Conclusions and Recommendations

Project Brave Heart produced a successful working product which met all its requirements. The design mimicked a turtle in shape, contained radio, clock, and alarm functionality. Additionally, the turtle contained a motor, gear, and wheel system which allowed it to move forward in a straight line when the alarm was activated. The clock and alarm functionality was programmed in micropython onto a Raspberry Pi Pico. The Pico plugged into a custom PCB which integrated a RDA5807M radio module, a battery and boost converter power system, a 2W speaker, a motor, and a SPI OLED display. The UI was achieved with the OLED display, an encoder, and several push buttons. An additional button system was integrated into the shell of the turtle to allow it to be pressed for snooze functionality.

All electrical and mechanical equipment were integrated into the self contained 3D printed turtle housing. The custom PCB was mounted directly to the enclosure with the battery module. The PCB was designed for potential expansion and modification of project Brave heart and as such included extra pins and component spaces.

The software and UI was tested in full to find bugs and errors within the system. The electrical and audio systems were measured to locate noise. Filter circuits were implemented to mitigate the electrical noise to produce a clean audio signal.

Several issues were encountered during the design of this project. Most of these issues revolved around noise filtering and the RDA radio module. For future development, we recommend research and development in these areas. In terms of noise filtering, more isolation or noise mitigation is required between the audio circuitry and the DC motor. For the RDA module, more understanding of the functionality of the device is required to enable tuning and seeking without power cycling the component. Additionally, a larger or more powerful antenna or FM reading system is needed to allow the radio to work better in internal environments.

References

[1] Texas Instruments, "PCB Design Guidelinse For Reduced EMI," NOV 1999.

[2] University of Victoria, “ECE 299 Parts Price List,” Price Data Sheet, May 2022

[3] 3Space, “How much does 3D printing cost?,” 3space.com, para. 17 June 2019. [Online] Available: https://3space.com/how-much-does-3d-printing-cost/ [Accessed Aug. 10, 2022]

[4] indeed, “Entry Level Engineer Salary in British Columbia,” indeed.com [Online]. Available: https://ca.indeed.com/career/entry-level-engineer/salaries/British-Columbia [Accessed Aug 10, 2022]

[5] Handson Technology, “Rotary Encoder for Arduino/Raspberry,” KY-040 datasheet. [Online]. [Accessed: June 15, 2022]

[6] Ali Express, “MT3608 DC-DC Step Up Converter Booster Power Supply Module Boost Step-up Board MAX output 28V 2A for arduino,” aliexpress.com. [Online]. Available: https://www.aliexpress.com/item/1005002190262803.html?spm=a2g0o.ppclist.product.2.451agHkdgHkd45&pdp_npi=2%40dis%21USD%21US%20%240.26%21US%20%240.17%21%21%21%21%21%402101d1b616603158041068009e8550%2112000019000845821%21btf&_t=pvid%3A4e27d134-6f01-4798-93df-bcebb2240a66&afTraceInfo=1005002190262803__pc__pcBridgePPC__xxxxxx__1660315804&gatewayAdapt=4itemAdapt. [Accessed: Aug. 12, 2022]

[7] Raspberry Pi Trading LTD, “Raspberry Pi Pico Datasheet: An RP2040-based microcontroller board.” 01-Feb-2021.

[8] RDA microelectronics, RDA5807M; SINGLE-CHIP BROADCAST FM RADIO TUNER, Rev.1.8., 2014.

Appendices

Appendix A - code

```
#Code for alarm clock
#Connor Wiebe, Matt Ebert
#July 19 2022

#libraries to import
from machine import Pin, SPI, Timer
import utime as time
from encoder_library import RotaryIRQ
from Clock_code import Clock,Alarm
import Clock_code
from bh_fm_radio import Radio
import math
#end of libraries to import

#button/motor setup
motor = machine.Pin(21,machine.Pin.OUT)
motor.value(0)
audio_out =machine.Pin(22,machine.Pin.OUT)

#end of button setup

#radio setup
fm_radio = Radio( 100.3, 2, True,9)
#end of radio setup

fm_radio.SetMute(True)
fm_radio.SetVolume(10)
while (True):
    motor.value(1)
    time.sleep(2)
    motor.value(0)
    time.sleep(2)
```

Figure A.1: code used to test that the radio was able to function while the motor is running

```
from bh_fm_radio import Radio

from ssd1306 import SSD1306_SPI # this is the driver library and the corresponding class
import framebuf # this is another library for the display.
SCREEN_WIDTH = 128 #number of columns
```

```
SCREEN_HEIGHT = 64 #number of rows
spi_sck = Pin(18) # sck stands for serial clock; always be connected to SPI SCK pin of the
Pico
spi_sda = Pin(19) # sda stands for serial data;  always be connected to SPI TX pin of the
Pico; this is the MOSI
spi_res = Pin(16) # res stands for reset; always be connected to SPI RX pin of the Pico; this is
the MISO
spi_dc  = Pin(17) # dc stands for data/commonda; always be connected to SPI CSn pin of the
Pico
spi_cs  = Pin(20) # can be connected to any free GPIO pin of the Pico
SPI_DEVICE = 0
oled_spi = SPI( SPI_DEVICE, baudrate= 100000, sck= spi_sck, mosi= spi_sda )
oled = SSD1306_SPI( SCREEN_WIDTH, SCREEN_HEIGHT, oled_spi, spi_dc, spi_res,
spi_cs, True )


fm_radio = Radio( 100.3, 2, True,9)
#end of radio setup

#encoder setup
encoder_button = machine.Pin(2, machine.Pin.IN, machine.Pin.PULL_UP)

encoder =
RotaryIRQ(pin_num_clk=0,pin_num_dt=1,min_val=0,max_val=3,range_mode=RotaryIRQ.RA
NGE_WRAP)
page =0

def debounce_encoder(x):#function for debouncing the button
  print("debounce_encoder")
  state =0xfff
  i=0
  while i<24:
     i=i+1
     print(x.value()|0x00)
     state = ((state<<1) | (x.value() | 0x000)) & 0xFFF
     print(hex(state))
  if (state == 0x000):
     return False
  else:
     return True

def work(x):#main encoder press function
   #debounce check
   check = debounce_encoder(x)
   if check == True:
      return
   #end of debounce check
   else:#main bulk of the function
      count+=1#incrementing the varible to indicate which step we are on
      if count>2:#set the encoder back to values for main pages
```

```
        encoder.set(value =
0,min_val=0,max_val=3,range_mode=RotaryIRQ.RANGE_WRAP)

def check_page(page):
   oled.fill(0)
      if count ==1:
         encoder.set(value =
0,min_val=0,max_val=9,range_mode=RotaryIRQ.RANGE_WRAP)
      if count ==2:
         encoder.set(value =
88,min_val=88,max_val=108,range_mode=RotaryIRQ.RANGE_WRAP)


      init_freq = fm_radio.get_Frequency()
      radio_freq_dec, radio_freq_main = math.modf(init_freq)
      radio_freq_dec = radio_freq_dec*100
      radio_freq_dec = int(radio_freq_dec)
      radio_freq_main = int(radio_freq_main)
      while (True):
         #print("oops")
         if count>2:#statement to end the loop an revert to main operation
            count =0
            page =1
            final_freq = float(radio_freq_main)+float(radio_freq_dec)/10
            fm_radio.SetFrequency(final_freq)
            fm_radio.SetMute(False)
            end_change_flag = True
            fm_radio.ProgramRadio()
            break
         oled.fill(0)
         oled.text("Set Radio Frequency:",15,0)
         if count ==1:
            oled.text("%2d.%02d"%(radio_freq_main,encoder.value()),25,45)
            radio_freq_dec=encoder.value()
         if count ==2:
            oled.text("%2d.%02d"%(encoder.value(),radio_freq_dec),25,45)
            radio_freq_main=encoder.value()
         oled.show()
```

Figure A.2: code to test the function of the encoder as well as the ability to update the rotational values

```
from machine import Timer
timer = Timer(-1)


def oops(x):
    print("aaaaaaaaaaaaaaaaaaa")

timer.init(mode=Timer.ONE_SHOT,period=1000,callback=oops)
```

Figure A.3: a short test code to ensure that the timer was operating properly

```
from machine import Pin, I2C
import machine
import utime


RDA_RDSR = 0b1000000000000000
RDA_BLERB = 0b0000000000000011


class Radio:

    def __init__( self, NewFrequency, NewVolume, NewMute, NewMutePin):


#
# set the initial values of the radio
#
        self.Volume = 2
        self.Frequency = 88
        self.Mute = False

#
# Update the values with the ones passed in the initialization code
#
        self.SetVolume( NewVolume )
        self.SetFrequency( NewFrequency )
        self.SetMutePin(NewMutePin)
        self.SetMute( NewMute )


# Initialize I/O pins associated with the radio's I2C interface

        self.i2c_sda = Pin(14)
        self.i2c_scl = Pin(15)
```

```
#
# I2C Device ID can be 0 or 1. It must match the wiring.
#
# The radio is connected to device number 1 of the I2C device
#
        self.i2c_device = 1
        self.i2c_device_address = 0x10

#
# Array used to configure the radio
#
        self.Settings = bytearray( 8 )
        self.Reg_info = bytearray( 12 )
        self.stationName = "----------"
        self.stationName_tmp1 = ['-'] * 12
        self.stationName_tmp2 = ['-'] * 12

        self.radio_i2c = I2C( self.i2c_device, scl=self.i2c_scl, sda=self.i2c_sda, freq=200000)
        self.ProgramRadio()



    def increase_volume (self):
        self.SetVolume(self.Volume+1)
        #print(self.Volume)
        self.ProgramRadio()

    def decrease_volume(self):
        self.SetVolume(self.Volume-1)
        self.ProgramRadio()
    def get_volume(self):
        return self.Volume

    def get_mute(self):
        return self.Mute

    def toggle_mute(self):
        self.Mute = not self.Mute
        self.MutePin.value(not self.Mute)
    def Seek(self):
        self.Settings[3] = self.Settings[3] | 0x00
        self.radio_i2c.writeto( self.i2c_device_address, self.Settings )
        self.Settings[0] = self.Settings[0] | 0x03
        self.radio_i2c.writeto( self.i2c_device_address, self.Settings )
        self.UpdateSettings()


    def SetMutePin(self, NewMutePin):
        self.MutePin = machine.Pin(NewMutePin,machine.Pin.OUT)
        self.MutePin.value(0)
```

```
   def SetVolume( self, NewVolume ):
      print("sel vol")
#
# Conver t the string into a integer
#
      try:


         NewVolume = int( NewVolume )

      except:
         return( False )

#
# Validate the type and range check the volume
#
      if ( not isinstance( NewVolume, int )):
         return( False )

      if (( NewVolume < 0 ) or ( NewVolume >= 16 )):
         return( False )

      self.Volume = NewVolume
      return( True )



   def SetFrequency( self, NewFrequency ):
#
# Convert the string into a floating point value
#
      try:
         NewFrequency = float( NewFrequency )

      except:
         return( False )

#
# validate the type and range check the frequency
#
      if ( not ( isinstance( NewFrequency, float ))):
         return( False )

      if (( NewFrequency < 88.0 ) or ( NewFrequency > 108.0 )):
         return( False )

      self.Frequency = NewFrequency
      return( True )
```

```
    def get_Frequency(self):
        return self.Frequency

    def SetMute( self, NewMute ):
        print("set mute")

        try:
            self.Mute = bool( int( NewMute ))
            self.MutePin.value(not self.Mute)

        except:
            return( False )

        return( True )

#
# convert the frequency to 10 bit value for the radio chip
#
    def ComputeChannelSetting( self, Frequency ):
        Frequency = int( Frequency * 10 ) - 870

        ByteCode = bytearray( 2 )
#
# split the 10 bits into 2 bytes
#
        ByteCode[0] = ( Frequency >> 2 ) & 0xFF
        ByteCode[1] = (( Frequency & 0x03 ) << 6 ) & 0xC0
        return( ByteCode )

#
# Configure the settings array with the mute, frequency and volume settings

#
    def UpdateSettings( self ):
        self.Settings[0] = 0xC0
        self.Settings[1] = 0x09 | 0x00
        print(len(self.Settings))
        cs = self.ComputeChannelSetting( self.Frequency )
        self.Settings[2] = cs[0]
        self.Settings[3] = cs[1]
        self.Settings[3] = self.Settings[3] | 0x10#change to 0x10 if it stops working
        self.Settings[4] = 0x00
        self.Settings[5] = 0x00
        self.Settings[6] = 0x84
        self.Settings[7] = (0x80 + self.Volume)


# Update the settings array and transmitt it to the radio
#
    def ProgramRadio( self ):
```

```
        #utime.sleep(5)
        print("programming")
        self.UpdateSettings()
        self.radio_i2c.writeto( self.i2c_device_address, self.Settings )

#
# Extract the settings from the radio registers
#
    def GetSettings( self ):
#
# Need to read the entire register space. This is allow access to the mute and volume settings
# After and address of 255 the
#
        self.RadioStatus = self.radio_i2c.readfrom( self.i2c_device_address, 256 )

        if (( self.RadioStatus[0xF0] & 0x40 ) != 0x00 ):
            MuteStatus = False
        else:
            MuteStatus = True

        VolumeStatus = self.RadioStatus[0xF7] & 0x0F

 #
 # Convert the frequency 10 bit count into actual frequency in Mhz
 #
        FrequencyStatus = (( self.RadioStatus[0x00] & 0x03 ) << 8 ) | ( self.RadioStatus[0x01] &
0xFF )
        FrequencyStatus = ( FrequencyStatus * 0.1 ) + 87.0

        if (( self.RadioStatus[0x00] & 0x04 ) != 0x00 ):
            StereoStatus = True
        else:
            StereoStatus = False

        return( MuteStatus, VolumeStatus, FrequencyStatus, StereoStatus )


    def GetInfo(self):
        self.Reg_info = self.radio_i2c.readfrom( self.i2c_device_address, 12 )
        a = self.Reg_info
        b = a[4]& 0x03
        if(b != 0x4):
            reg_a = (a[0] << 8) | a[1]
            reg_b = (a[2] << 8) | a[3]
            #if reg_b & RDA_BLERB != 0:
            if reg_a & RDA_RDSR == 0 or reg_b & RDA_BLERB != 0:
                # no new rds group ready
                #print('continue')
                return self.stationName
```

```
        block_b = (a[6] << 8) | a[7]
        block_c = (a[8] << 8) | a[9]
        block_d = (a[10] << 8) | a[11]
        group_type = 0x0a + ((block_b & 0xf000) >> 8) | ((block_b & 0x0800) >> 11)
        if group_type in [0x0a, 0x0b]:
            # PS name
            idx = (block_b & 3) * 2
            c1 = chr(block_d >> 8)
            c2 = chr(block_d & 0xff)
            if (c1.isalpha() or c1 == ' ' ) and (c2.isalpha() or c2 ==' '):
                #print("c1 = " + c1 + "  c2 = " + c2)
                if self.stationName_tmp1[idx:idx + 2] == [c1, c2]:
                        self.stationName_tmp2[idx:idx + 2] = [c1, c2]
                        if  self.stationName_tmp1 ==  self.stationName_tmp2:
                            self.stationName = ''.join( self.stationName_tmp1)
                            #savedName = savedName + str(stationName_tmp1)
                if  self.stationName_tmp1[idx:idx + 2] != [c1, c2]:
                        self.stationName_tmp1[idx:idx + 2] = [c1, c2]
    return self.stationName

  #print(str(savedName))
```

Figure A.4: the bh_fm_radio library used for the clock

```
# The MIT License (MIT)
# Copyright (c) 2022 Mike Teachman
# https://opensource.org/licenses/MIT

# Platform-independent MicroPython code for the rotary encoder module

# Documentation:
#   https://github.com/MikeTeachman/micropython-rotary

import micropython

_DIR_CW = const(0x10)  # Clockwise step
_DIR_CCW = const(0x20)  # Counter-clockwise step

# Rotary Encoder States
_R_START = const(0x0)
_R_CW_1 = const(0x1)
_R_CW_2 = const(0x2)
_R_CW_3 = const(0x3)
_R_CCW_1 = const(0x4)
_R_CCW_2 = const(0x5)
```

```
_R_CCW_3 = const(0x6)
_R_ILLEGAL = const(0x7)

_transition_table = [

    # |------------- NEXT STATE -------------|            |CURRENT STATE|
    # CLK/DT    CLK/DT     CLK/DT    CLK/DT
    #   00        01         10        11
    [_R_START, _R_CCW_1, _R_CW_1,  _R_START],            # _R_START
    [_R_CW_2,  _R_START, _R_CW_1,  _R_START],            # _R_CW_1
    [_R_CW_2,  _R_CW_3,  _R_CW_1,  _R_START],            # _R_CW_2
    [_R_CW_2,  _R_CW_3,  _R_START, _R_START | _DIR_CW],  # _R_CW_3
    [_R_CCW_2, _R_CCW_1, _R_START, _R_START],            # _R_CCW_1
    [_R_CCW_2, _R_CCW_1, _R_CCW_3, _R_START],            # _R_CCW_2
    [_R_CCW_2, _R_START, _R_CCW_3, _R_START | _DIR_CCW], # _R_CCW_3
    [_R_START, _R_START, _R_START, _R_START]]            # _R_ILLEGAL

_transition_table_half_step = [
    [_R_CW_3,           _R_CW_2,  _R_CW_1,  _R_START],
    [_R_CW_3 | _DIR_CCW, _R_START, _R_CW_1,  _R_START],
    [_R_CW_3 | _DIR_CW,  _R_CW_2,  _R_START, _R_START],
    [_R_CW_3,           _R_CCW_2, _R_CCW_1, _R_START],
    [_R_CW_3,           _R_CW_2,  _R_CCW_1, _R_START | _DIR_CW],
    [_R_CW_3,           _R_CCW_2, _R_CW_3,  _R_START | _DIR_CCW],
    [_R_START,          _R_START, _R_START, _R_START],
    [_R_START,          _R_START, _R_START, _R_START]]

_STATE_MASK = const(0x07)
_DIR_MASK = const(0x30)


def _wrap(value, incr, lower_bound, upper_bound):
    range = upper_bound - lower_bound + 1
    value = value + incr

    if value < lower_bound:
        value += range * ((lower_bound - value) // range + 1)

    return lower_bound + (value - lower_bound) % range


def _bound(value, incr, lower_bound, upper_bound):
    return min(upper_bound, max(lower_bound, value + incr))


def _trigger(rotary_instance):
    for listener in rotary_instance._listener:
        listener()
```

```
class Rotary(object):

    RANGE_UNBOUNDED = const(1)
    RANGE_WRAP = const(2)
    RANGE_BOUNDED = const(3)

    def __init__(self, min_val, max_val, reverse, range_mode, half_step, invert):
        self._min_val = min_val
        self._max_val = max_val
        self._reverse = -1 if reverse else 1
        self._range_mode = range_mode
        self._value = min_val
        self._state = _R_START
        self._half_step = half_step
        self._invert = invert
        self._listener = []

    def set(self, value=None, min_val=None,
            max_val=None, reverse=None, range_mode=None):
        # disable DT and CLK pin interrupts
        self._hal_disable_irq()

        if value is not None:
            self._value = value
        if min_val is not None:
            self._min_val = min_val
        if max_val is not None:
            self._max_val = max_val
        if reverse is not None:
            self._reverse = -1 if reverse else 1
        if range_mode is not None:
            self._range_mode = range_mode
        self._state = _R_START

        # enable DT and CLK pin interrupts
        self._hal_enable_irq()

    def value(self):
        return self._value

    def reset(self):
        self._value = 0

    def close(self):
        self._hal_close()

    def add_listener(self, l):
        self._listener.append(l)

    def remove_listener(self, l):
```

```
        if l not in self._listener:
            raise ValueError('{} is not an installed listener'.format(l))
        self._listener.remove(l)

    def _process_rotary_pins(self, pin):
        old_value = self._value
        clk_dt_pins = (self._hal_get_clk_value() <<
                       1) | self._hal_get_dt_value()

        if self._invert:
            clk_dt_pins = ~clk_dt_pins & 0x03

        # Determine next state
        if self._half_step:
            self._state = _transition_table_half_step[self._state &
                                                      _STATE_MASK][clk_dt_pins]
        else:
            self._state = _transition_table[self._state &
                                            _STATE_MASK][clk_dt_pins]
        direction = self._state & _DIR_MASK

        incr = 0
        if direction == _DIR_CW:
            incr = 1
        elif direction == _DIR_CCW:
            incr = -1

        incr *= self._reverse

        if self._range_mode == self.RANGE_WRAP:
            self._value = _wrap(
                self._value,
                incr,
                self._min_val,
                self._max_val)
        elif self._range_mode == self.RANGE_BOUNDED:
            self._value = _bound(
                self._value,
                incr,
                self._min_val,
                self._max_val)
        else:
            self._value = self._value + incr

        try:
            if old_value != self._value and len(self._listener) != 0:
                micropython.schedule(_trigger, self)
        except:
            pass
```

Figure A.5: part 1 of the encoder library used for the clock which was saved as rotary.py

```
# The MIT License (MIT)
# Copyright (c) 2020 Mike Teachman
# Copyright (c) 2021 Eric Moyer
# https://opensource.org/licenses/MIT

# Platform-specific MicroPython code for the rotary encoder module
# Raspberry Pi Pico implementation

# Documentation:
#   https://github.com/MikeTeachman/micropython-rotary

from machine import Pin
from rotary import Rotary

IRQ_RISING_FALLING = Pin.IRQ_RISING | Pin.IRQ_FALLING


class RotaryIRQ(Rotary):
    def __init__(
        self,
        pin_num_clk,
        pin_num_dt,
        min_val=0,
        max_val=10,
        reverse=False,
        range_mode=Rotary.RANGE_UNBOUNDED,
        pull_up=False,
        half_step=False,
        invert=False,
    ):
        super().__init__(min_val, max_val, reverse, range_mode, half_step, invert)

        if pull_up:
            self._pin_clk = Pin(pin_num_clk, Pin.IN, Pin.PULL_UP)
            self._pin_dt = Pin(pin_num_dt, Pin.IN, Pin.PULL_UP)
        else:
            self._pin_clk = Pin(pin_num_clk, Pin.IN)
            self._pin_dt = Pin(pin_num_dt, Pin.IN)

        self._hal_enable_irq()

    def _enable_clk_irq(self):
        self._pin_clk.irq(self._process_rotary_pins, IRQ_RISING_FALLING)

    def _enable_dt_irq(self):
        self._pin_dt.irq(self._process_rotary_pins, IRQ_RISING_FALLING)
```

```
    def _disable_clk_irq(self):
        self._pin_clk.irq(None, 0)

    def _disable_dt_irq(self):
        self._pin_dt.irq(None, 0)

    def _hal_get_clk_value(self):
        return self._pin_clk.value()

    def _hal_get_dt_value(self):
        return self._pin_dt.value()

    def _hal_enable_irq(self):
        self._enable_clk_irq()
        self._enable_dt_irq()

    def _hal_disable_irq(self):
        self._disable_clk_irq()
        self._disable_dt_irq()

    def _hal_close(self):
        self._hal_disable_irq()
```

Figure A.6: part 2 encoder library used for the clock which was saved as encoder_library.py

```
from bh_fm_radio import Radio
import machine
import utime as time
def motor_off(x):
    motor = machine.Pin(21, machine.Pin.OUT)
    motor.value(0)

class Clock:
    def __init__ (self,hour,minute,am_pm,time_type,set_flag):
        self.hour = hour
        self.minute = minute
        self.am_pm = am_pm
        self.time_type=time_type
        self.set_flag=False

    def get_hour(self):
        return self.hour

    def get_minute(self):
        return self.minute

    def get_am_pm(self):
```

```
    if self.time_type == False:
        if self.am_pm == False:
            return "am"
        else:
            return "pm"
    else:
        return ""
def get_time_type(self):
    return self.time_type

def hour_set(self, hour):
    self.hour = hour

def minute_set(self,minute):
    self.minute = minute

def am_pm_set(self,am_pm):
    if am_pm == 1:
        self.am_pm = True
    else:
        self.am_pm = False

def time_type_change(self,time_type):
    if self.time_type == False:
        if self.am_pm ==True and self.hour !=12:
            self.hour = self.hour+12
            self.time_type=time_type
        else:
            self.time_type = time_type

    elif self.time_type ==True:
        if self.hour>12 and (self.hour !=12 or self.am_pm != False):
            self.hour = self.hour-12
            self.time_type=time_type
        else:
            self.time_type = time_type


def update(self):
    if self.time_type == False:
        if self.minute>=0 and self.minute<59:
           self.minute = self.minute+1

        elif self.hour<12 and self.hour>0:
            self.hour = self.hour+1
            self.minute =0
            edge_flag=False

        if self.hour==12 and self.minute ==0:
            self.am_pm = not self.am_pm
```

```
            elif self.hour==12 and self.minute==59:
                self.hour=1
                self.minute=0

        if self.time_type == True:
            if self.minute>=0 and self.minute<59:
                self.minute = self.minute+1

            elif self.hour<23 and self.hour>=0:
                self.hour = self.hour+1
                self.minute =0
            elif self.hour==23 and self.minute==59:
                self.hour=0
                self.minute=0

class Alarm(Clock):
    def __init__ (self,hour=1,minute=1,am_pm=False,time_type=False,set_flag=False):
        self.motor = machine.Pin(21, machine.Pin.OUT)
        self.goin_off =False
        super().__init__(hour,minute,am_pm,time_type,set_flag)
    def get_set_flag(self):
        return self.set_flag
    def update_set_flag(self):
        self.set_flag = not self.set_flag
    def pop_off(self,Clock,Radio):
        if self.set_flag == True:
            #print("here")
            if Clock.get_time_type()==False:
                #print(self.am_pm)
                if self.hour==Clock.get_hour() and self.minute == Clock.get_minute() and
self.am_pm==Clock.am_pm:
                    print("start runnin 12hr")
                    Radio.SetMute(False)
                    Radio.SetVolume(15)
                    Radio.ProgramRadio()
                    self.goin_off = True
                    #self.motor.value(1)
                    #time.sleep(2)
                    #self.motor.value(0)
                    #print(self.goin_off)

            if Clock.get_time_type() ==True:
                if self.hour==Clock.get_hour() and self.minute == Clock.get_minute():
                    print ("start runnin 24hr")
                    Radio.SetMute(False)
                    Radio.SetVolume(15)
                    Radio.ProgramRadio()
                    self.goin_off = True
                    #self.motor.value(1)
```

```
            #time.sleep(2)
            #self.motor.value(0)
        else:
            return



def snooze(self,Clock,Radio):
    if self.goin_off == True:
        self.motor.value(0)
        Radio.SetMute(True)
        Radio.ProgramRadio()
        if Clock.get_time_type() == False:
            if Clock.get_minute()<55:
                self.minute = Clock.get_minute()+5
                #goin_off = False
            elif Clock.get_minute()>=55 and Clock.get_hour()==11:
                self.minute = Clock.get_minute()-55
                self.hour = 12
                self.am_pm = not self.am_pm

            elif Clock.get_minute()>=55 and Clock.get_hour()==12:
                self.minute = Clock.get_minute()-55
                self.hour = 1
                #goin_off = False
            elif Clock.get_minute()>=55 and Clock.get_hour()<11:
                self.minute = Clock.get_minute()-55
                self.hour = Clock.get_hour()+1
                #goin_off = False
        elif Clock.get_time_type()==True:
            if Clock.get_minute()<55:
                self.minute = Clock.get_minute()+5
                #goin_off = False
            elif Clock.get_minute()>=55 and Clock.get_hour()<23:
                self.minute = Clock.get_minute()-55
                self.hour = Clock.get_hour()+1
                #goin_off = False
            elif Clock.get_minute()>=55 and Clock.get_hour()==23:
                self.minute = Clock.get_minute()-55
                self.hour = 0
                #goin_off = False
        self.goin_off =False
    else:
        return
```

Figure A.7: A clock and alarm library created for the Clock radio which was saved as Clock_code.py

```
#Code for alarm clock
#Connor Wiebe, Matt Ebert
#July 19 2022

#libraries to import
from machine import Pin, SPI, Timer
import utime as time
from encoder_library import RotaryIRQ
from Clock_code import Clock,Alarm
import Clock_code
from bh_fm_radio import Radio
import math
#end of libraries to import

#button/motor setup
snooze_button1 = machine.Pin(7,machine.Pin.IN,machine.Pin.PULL_UP)
snooze_button2 = machine.Pin(8,machine.Pin.IN,machine.Pin.PULL_UP)
vol_up_button = machine.Pin(3,machine.Pin.IN,machine.Pin.PULL_UP)
vol_down_button = machine.Pin(4,machine.Pin.IN,machine.Pin.PULL_UP)
seek_button = machine.Pin(6,machine.Pin.IN,machine.Pin.PULL_UP)
mute_toggle_button = machine.Pin(5,machine.Pin.IN,machine.Pin.PULL_UP)
motor = machine.Pin(21,machine.Pin.OUT)
motor.value(0)
audio_out =machine.Pin(22,machine.Pin.OUT)
#radio_on_off = machine.Pin(8,machine.Pin.OUT)
#radio_on_off.value(0)

#end of button setup

#radio setup
fm_radio = Radio( 100.3, 2, True,9)
#end of radio setup

#encoder setup
encoder_button = machine.Pin(2, machine.Pin.IN, machine.Pin.PULL_UP)

encoder =
RotaryIRQ(pin_num_clk=0,pin_num_dt=1,min_val=0,max_val=3,range_mode=RotaryIRQ.RA
NGE_WRAP)
#end of encoder setup

#Stuff for the oled screen

from ssd1306 import SSD1306_SPI # this is the driver library and the corresponding class
import framebuf # this is another library for the display.
SCREEN_WIDTH = 128 #number of columns
SCREEN_HEIGHT = 64 #number of rows
spi_sck = Pin(18) # sck stands for serial clock; always be connected to SPI SCK pin of the
Pico
spi_sda = Pin(19) # sda stands for serial data;  always be connected to SPI TX pin of the
```

```
Pico; this is the MOSI
spi_res = Pin(16) # res stands for reset; always be connected to SPI RX pin of the Pico; this is the MISO
spi_dc  = Pin(17) # dc stands for data/commonda; always be connected to SPI CSn pin of the Pico
spi_cs  = Pin(20) # can be connected to any free GPIO pin of the Pico
SPI_DEVICE = 0
oled_spi = SPI( SPI_DEVICE, baudrate= 100000, sck= spi_sck, mosi= spi_sda )
oled = SSD1306_SPI( SCREEN_WIDTH, SCREEN_HEIGHT, oled_spi, spi_dc, spi_res, spi_cs, True )
#End of OLED setup stuff

#Setting up varibales and classes for later
count = 0
page = 0
hour = 8
minute = 45
end_change_flag =False
am_pm = True
time_type=False
Clock = Clock(hour,minute,am_pm,time_type,False)
Alarm = Alarm(hour,minute,am_pm,time_type,set_flag=False)
#end of varibale and class setup

def motor_off(x):
    motor.value(0)

def vol_up(x):
    check = debounce(x)
    if check == True:
        return
    else:
        fm_radio.increase_volume()
        check_page(page)
def vol_down(x):
    check = debounce(x)
    if check == True:
        return
    else:
        fm_radio.decrease_volume()
        check_page(page)
def snooze(x):
    check = debounce(x)
    if check == True:
        return
    else:
        #motor.value(1)
        #time.sleep(2)
        motor.value(0)
        print("snoozed")
```

```
        Alarm.snooze(Clock,fm_radio)
def mute_toggle(x):
    check = debounce(x)
    if check == True:
        return
    else:
        fm_radio.toggle_mute()
        check_page(page)
def seek(x):
    #print(fm_radio.get_Frequency())
    check = debounce(x)
    if check == True:
        return
    else:
        fm_radio.Seek()
        check_page(page)

def update_time(x):#function for updating and displaying clock time
    if page == 0 or page==1:
        oled.text("%02d:%02d %s"
%(Clock.get_hour(),Clock.get_minute(),Clock.get_am_pm()),25,0,0)
        Clock.update()
        Alarm.pop_off(Clock,fm_radio)
        oled.text("%02d:%02d %s"
%(Clock.get_hour(),Clock.get_minute(),Clock.get_am_pm()),25,0,1)
        oled.show()
    else:
        Clock.update()
        Alarm.pop_off(Clock,fm_radio)
    if fm_radio.get_mute()==False:
            oled.text("Mute",0,55,0)
    if Alarm.goin_off==True:
          #print("motor 1")
          time.sleep(0.01)
          motor.value(1)
          #print("motor 2")
          #time.sleep(2)
          #motor.value(0)
          #print("motor 3")
        #check_page(page)
def update_radio(x):
    if page ==0:
        oled.fill_rect(0,35,300,10,0)
        oled.text(fm_radio.GetInfo(),15,35)
        oled.show()


def debounce(x):#function for debouncing the button
   state =0xff
   i=0
```

```
  while i<16:
     i=i+1
    # print(x.value()|0x00)
     state = ((state<<1) | (x.value() | 0x00)) & 0xFF
     #print(hex(state))
  if (state == 0x00):
     return False
  else:
     return True

def debounce_encoder(x):#function for debouncing the button
  print("debounce_encoder")
  state =0xfff
  i=0
  while i<24:
     i=i+1
     print(x.value()|0x00)
     state = ((state<<1) | (x.value() | 0x000)) & 0xFFF
     print(hex(state))
  if (state == 0x000):
     return False
  else:
     return True

def work(x):#main encoder press function
   #debounce check
   check = debounce_encoder(x)
   if check == True:
      return
   #end of debounce check
   else:#main bulk of the function
      global page
      global count
      if page==0 or page==1:
         print("press")
         if page == 0:
            if encoder.value() == 0:#set alarm
               page = 3
            #print("notyet")
            elif encoder.value() == 1:#set radio
               page =4
            if encoder.value() == 2:#radio info button
               #print("radio not done yet")
               page =5
            if encoder.value() ==3:#next button
               page = 1
               encoder.set(value =
0,min_val=0,max_val=3,range_mode=RotaryIRQ.RANGE_WRAP)
         elif page == 1:
            if encoder.value() == 0:#set clock
```

```
                #print("here")
                page =2
            elif encoder.value() == 1:#toggle 12/24hr time
              Clock.time_type_change(not Clock.get_time_type())
              Alarm.time_type_change(Clock.get_time_type())
            elif encoder.value() ==2:#toggle alarm
                Alarm.update_set_flag()
            elif encoder.value() == 3:#prev button
                page = 0
                encoder.set(value =
0,min_val=0,max_val=3,range_mode=RotaryIRQ.RANGE_WRAP)
        if page==2 or page==3:
            #print("clock change press")
            count = count+1 #incrementing the varible to indicate which step we are on
            if Clock.get_time_type() == False:
                if count>3:#set the encoder back to values for main pages
                    #page=0
                    encoder.set(value =
0,min_val=0,max_val=3,range_mode=RotaryIRQ.RANGE_WRAP)
            elif Clock.get_time_type() == True:
                if count>2:#set the encoder back to values for main pages
                    #page=0
                    encoder.set(value =
0,min_val=0,max_val=3,range_mode=RotaryIRQ.RANGE_WRAP)
        if page ==4:
            count+=1#incrementing the varible to indicate which step we are on
            if count>2:#set the encoder back to values for main pages
                encoder.set(value =
0,min_val=0,max_val=3,range_mode=RotaryIRQ.RANGE_WRAP)
        if page==5:#set encoder back to values for main pages
            if encoder.value()==0:
                page=0
                encoder.set(value =
0,min_val=0,max_val=3,range_mode=RotaryIRQ.RANGE_WRAP)



        check_page(page)

def check_page(page):#function to display correct stuff for page on screen and update
encoder values
    mute_stat,vol_stat,freq_stat,sterio_stat=fm_radio.GetSettings()
    if page ==0:#this is to display page 0
        oled.fill(0)
        oled.text("%02d"%vol_stat,0,0)
        oled.text("%02d:%02d %s"
%(Clock.get_hour(),Clock.get_minute(),Clock.get_am_pm()),25,0)
        oled.text("Set Alarm",15,15)
        oled.text("Set Radio",15,25)
        #oled.text("Radio Info",15,35)
```

```
        oled.text("Next ->",15,45)
        if fm_radio.get_mute()==True:
            oled.text("Mute",0,55,1)
        elif fm_radio.get_mute()==False:
            oled.text("Mute",0,55,0)
        if Alarm.get_set_flag()==True:
            oled.text("Armd",98,0)
        oled.text("%.2f FM"%freq_stat,35,55)
        oled.show()
    elif page==1:#this is to display page 1
        oled.fill(0)
        oled.text("%02d"%vol_stat,0,0)
        oled.text("%02d:%02d %s"
%(Clock.get_hour(),Clock.get_minute(),Clock.get_am_pm()),25,0)
        oled.text("Set Clock",15,15)
        oled.text("12/24 hour",15,25)
        oled.text("Alarm On/Off",15,35)
        oled.text("<- Prev",15,45)
        if fm_radio.get_mute()==True:
            oled.text("Mute",0,55,1)
        elif fm_radio.get_mute()==False:
            oled.text("Mute",0,55,0)
        if Alarm.get_set_flag()==True:
            oled.text("Armd",98,0)
        oled.text("%.2f FM"%freq_stat,35,55)
        oled.show()
    elif page==2 or page==3: #this is to change the encoder to have valid values for the clock
        oled.fill(0)
        if Clock.time_type == False: #12hr time
            if count ==1:
                #oled.text("Change Minute:",25,25,1)
                encoder.set(value =
0,min_val=0,max_val=59,range_mode=RotaryIRQ.RANGE_WRAP)
            if count ==2:
                encoder.set(value =
1,min_val=1,max_val=12,range_mode=RotaryIRQ.RANGE_WRAP)
                #oled.text("Change Minute:",25,25,0)
                #oled.text("Change Hour:",25,25,1)
            if count ==3:
                encoder.set(value =
0,min_val=0,max_val=1,range_mode=RotaryIRQ.RANGE_WRAP)
                #oled.text("Change Hour:",25,25,0)
                #oled.text("Am or Pm?",25,25,1)
            #if count>3:
              # page=0
        else:
            if count ==1:
                encoder.set(value =
0,min_val=0,max_val=59,range_mode=RotaryIRQ.RANGE_WRAP)
                #oled.text("Change Minute",25,25,1)
```

```
            if count ==2:
                encoder.set(value =
0,min_val=0,max_val=23,range_mode=RotaryIRQ.RANGE_WRAP)
                #oled.text("Change Minute",25,25,0)
                #oled.text("Change Hour",25,25,1)
            # if count>2:
              #   page=0
    elif page ==4:#set encoder values for the radio
        oled.fill(0)
        if count ==1:
            encoder.set(value =
0,min_val=0,max_val=9,range_mode=RotaryIRQ.RANGE_WRAP)
        if count ==2:
            encoder.set(value =
88,min_val=88,max_val=108,range_mode=RotaryIRQ.RANGE_WRAP)
    elif page==5:#essentially make the encoder a back button
        encoder.set(value = 0,min_val=0,max_val=0,range_mode=RotaryIRQ.RANGE_WRAP)
        oled.fill(0)


oled.fill(0)
oled.text("%02d"%fm_radio.get_volume(),0,0)
oled.text("%02d:%02d %s" %(Clock.get_hour(),Clock.get_minute(),Clock.get_am_pm()),25,0)
oled.text("Set Alarm",15,15)
oled.text("Set Radio",15,25)
#oled.text("Radio Info",15,35)
oled.text("Next ->",15,45)
oled.text("%.2f FM"%fm_radio.get_Frequency(),35,55)
if fm_radio.get_mute()==True:
            oled.text("Mute",0,55,1)
elif fm_radio.get_mute()==False:
            oled.text("Mute",0,55,0)
oled.show()
clk_update = Timer(-1)
clk_update.init(period=6000,callback=update_time)
radio_update = Timer(-1)
radio_update.init(period=300,callback=update_radio)

#interrupt handler setup
encoder_button.irq(trigger=machine.Pin.IRQ_FALLING, handler=work)
seek_button.irq(trigger=machine.Pin.IRQ_FALLING, handler=seek)
snooze_button1.irq(trigger=machine.Pin.IRQ_FALLING, handler=snooze)
snooze_button2.irq(trigger=machine.Pin.IRQ_FALLING, handler=snooze)
vol_up_button.irq(trigger=machine.Pin.IRQ_FALLING, handler=vol_up)
vol_down_button.irq(trigger=machine.Pin.IRQ_FALLING, handler=vol_down)
mute_toggle_button.irq(trigger=machine.Pin.IRQ_FALLING, handler=mute_toggle)
#end of interuppt handler setup
#radio bug stoppage
mute_stat,vol_stat,freq_stat,sterio_stat=fm_radio.GetSettings()
if freq_stat>108:
```

```
    print("called")
    fm_radio.DefaultSettings()
#end of radio bug stoppage
while (True):
    #print(fm_radio.GetInfo())
    #print(encoder.value())
    if page==1 or page ==0:#this is for using the 2 main pages
        #update_radio()
        if encoder.value() == 0:
            #oled.fill(0)
            oled.text(">",0,25,0)
            oled.text(">",0,35,0)
            oled.text(">",0,15,1)
            oled.text(">",0,45,0)
            #oled.show()
        elif encoder.value() == 1:
            #oled.fill(0)
            oled.text(">",0,25,1)
            oled.text(">",0,35,0)
            oled.text(">",0,15,0)
            #oled.show()
        elif encoder.value() == 2:
            #oled.fill(0)
            oled.text(">",0,25,0)
            oled.text(">",0,35,1)
            oled.text(">",0,15,0)
            oled.text(">",0,45,0)
        elif encoder.value() == 3:
            #oled.fill(0)
            oled.text(">",0,15,0)
            oled.text(">",0,25,0)
            oled.text(">",0,35,0)
            oled.text(">",0,45,1)
            #oled.show()
        oled.show()
    if page==4:#page for setting radio frequency
        init_freq = fm_radio.get_Frequency()
        radio_freq_dec, radio_freq_main = math.modf(init_freq)
        radio_freq_dec = radio_freq_dec*100
        radio_freq_dec = int(radio_freq_dec)
        radio_freq_main = int(radio_freq_main)
        while (True):
            #print("oops")
            if count>2:#statement to end the loop an revert to main operation
                count =0
                page =1
                final_freq = float(radio_freq_main)+float(radio_freq_dec)/10
                fm_radio.SetFrequency(final_freq)
                fm_radio.SetMute(False)
                end_change_flag = True
```

```
            fm_radio.ProgramRadio()
            break
        oled.fill(0)
        oled.text("Set Radio Frequency:",15,0)
        if count ==1:
            oled.text("%2d.%02d"%(radio_freq_main,encoder.value()),25,45)
            radio_freq_dec=encoder.value()
        if count ==2:
            oled.text("%2d.%02d"%(encoder.value(),radio_freq_dec),25,45)
            radio_freq_main=encoder.value()
        oled.show()
if page ==5: #this page displays the radio information
    mute_stat,vol_stat,freq_stat,sterio_stat=fm_radio.GetSettings()
    #next 2 if statements are to print if mute is true or false
    if mute_stat ==1:
        mute_stat_string = "True"
    elif mute_stat==0:
        mute_stat_string = "False"
    oled.fill(0)
    oled.text("Radio Freq:",0,0)
    oled.text("          %2f FM"%freq_stat,0,0)
    oled.text("Mute: %s"%mute_stat_string,0,25)
    oled.text("Volume: %d"%vol_stat,0,35)
    oled.text("Station Info: %s"%sterio_stat,0,15)
    oled.text("<- Back",0,45)
    oled.show()



if page ==2 or page==3:#this if statement is for setting the clock
    while (True):
        if page==3: #this page is for setting the alarm
            if Alarm.time_type == False: #12hr time
                if count>3:#statement to end the loop an revert to main operation
                    count=0
                    page=0
                    Alarm.set_flag=True
                    end_change_flag = True
                    print("broken")
                    break
                oled.fill(0)
                oled.text("Set Alarm:",25,0)
                if count ==1:
                    oled.text("Change Minute:",25,25,1)
                    oled.text("%02d:    %s" %(Alarm.get_hour(),Alarm.get_am_pm()),25,45)
                    oled.text("    %02d"%encoder.value(),25,45,1)
                    #oled.text("    %2d"%encoder.value(),25,45,0)
                    Alarm.minute_set(encoder.value())
                    #print(page)
                if count ==2:
```

```
                oled.text("   :%02d %s" %(Alarm.get_minute(),Alarm.get_am_pm()),25,45)
                oled.text("%02d"%encoder.value(),25,45,1)
                #oled.text("%2d"%encoder.value(),25,45,0)
                oled.text("Change Minute:",25,25,0)
                oled.text("Change Hour:",25,25,1)
                Alarm.hour_set(encoder.value())
            if count ==3:
                oled.text("%02d:%02d " %(Alarm.get_hour(),Alarm.get_minute()),25,45)
                oled.text("       %s"%Alarm.get_am_pm(),25,45,1)
                #oled.text("       %s"%Clock.get_am_pm(),25,45,0)
                oled.text("Change Hour:",25,25,0)
                oled.text("Am or Pm?",25,25,1)
                Alarm.am_pm_set(encoder.value())
            oled.show()
        else: #24 hr time
            if count>2:#statement to end the loop an revert to main operation
                count=0
                page=0
                Alarm.set_flag=True
                end_change_flag = True
                break
            oled.fill(0)
            oled.text("Change Time:",25,0)
            if count ==1:
                oled.text("%02d:%02d" %(Alarm.get_hour(),encoder.value()),25,45)
                oled.text("Change Minute:",25,25,1)
                Alarm.minute_set(encoder.value())
            if count ==2:
                oled.text("%02d:%02d" %(encoder.value(),Alarm.get_minute()),25,45)
                oled.text("Change Minute:",25,25,0)
                oled.text("Change Hour:",25,25,1)
                Alarm.hour_set(encoder.value())
            oled.show()


        oled.show()
    if page ==2: #this page is for setting the clock
        if Clock.time_type == False: #12hr time
            if count>3:#statement to end the loop an revert to main operation
                count=0
                page=0
                end_change_flag = True
                print("broken")
                break
            oled.fill(0)
            oled.text("Change Time:",25,0)
            if count ==1:
                oled.text("Change Minute:",25,25,1)
                oled.text("%2d:    %s" %(Clock.get_hour(),Clock.get_am_pm()),25,45)
                oled.text("    %2d"%encoder.value(),25,45,1)
```

```
                #oled.text("    %2d"%encoder.value(),25,45,0)
                Clock.minute_set(encoder.value())
                #print(page)
            if count ==2:
                oled.text("   :%2d %s" %(Clock.get_minute(),Clock.get_am_pm()),25,45)
                oled.text("%2d"%encoder.value(),25,45,1)
                #oled.text("%2d"%encoder.value(),25,45,0)
                oled.text("Change Minute:",25,25,0)
                oled.text("Change Hour:",25,25,1)
                Clock.hour_set(encoder.value())
            if count ==3:
                oled.text("%2d:%2d " %(Clock.get_hour(),Clock.get_minute()),25,45)
                oled.text("       %s"%Clock.get_am_pm(),25,45,1)
                #oled.text("       %s"%Clock.get_am_pm(),25,45,0)
                oled.text("Change Hour:",25,25,0)
                oled.text("Am or Pm?",25,25,1)
                Clock.am_pm_set(encoder.value())
            oled.show()
        else:
            if count>2:#statement to end the loop an revert to main operation
                count=0
                page=0
                end_change_flag = True
                break
            oled.fill(0)
            oled.text("Change Time:",25,0)
            if count ==1:
                oled.text("%2d:%2d" %(Clock.get_hour(),encoder.value()),25,45)
                oled.text("Change Minute:",25,25,1)
                Clock.minute_set(encoder.value())
            if count ==2:
                oled.text("%2d:%2d" %(encoder.value(),Clock.get_minute()),25,45)
                oled.text("Change Minute:",25,25,0)
                oled.text("Change Hour:",25,25,1)
                Clock.hour_set(encoder.value())
            oled.show()


        oled.show()
if end_change_flag ==True:
    end_change_flag =False
    check_page(page)
```

Figure A.8: The final code used in main.py to run the Clock for the final demonstration

```
# MicroPython SSD1306 OLED driver, I2C and SPI interfaces

from micropython import const
```

```
import framebuf


# register definitions
SET_CONTRAST = const(0x81)
SET_ENTIRE_ON = const(0xA4)
SET_NORM_INV = const(0xA6)
SET_DISP = const(0xAE)
SET_MEM_ADDR = const(0x20)
SET_COL_ADDR = const(0x21)
SET_PAGE_ADDR = const(0x22)
SET_DISP_START_LINE = const(0x40)
SET_SEG_REMAP = const(0xA0)
SET_MUX_RATIO = const(0xA8)
SET_IREF_SELECT = const(0xAD)
SET_COM_OUT_DIR = const(0xC0)
SET_DISP_OFFSET = const(0xD3)
SET_COM_PIN_CFG = const(0xDA)
SET_DISP_CLK_DIV = const(0xD5)
SET_PRECHARGE = const(0xD9)
SET_VCOM_DESEL = const(0xDB)
SET_CHARGE_PUMP = const(0x8D)

# Subclassing FrameBuffer provides support for graphics primitives
# http://docs.micropython.org/en/latest/pyboard/library/framebuf.html
class SSD1306(framebuf.FrameBuffer):
    def __init__(self, width, height, external_vcc):
        self.width = width
        self.height = height
        self.external_vcc = external_vcc
        self.pages = self.height // 8
        self.buffer = bytearray(self.pages * self.width)
        super().__init__(self.buffer, self.width, self.height, framebuf.MONO_VLSB)
        self.init_display()

    def init_display(self):
        for cmd in (
            SET_DISP,  # display off
            # address setting
            SET_MEM_ADDR,
            0x00,  # horizontal
            # resolution and layout
            SET_DISP_START_LINE,  # start at line 0
            SET_SEG_REMAP | 0x01,  # column addr 127 mapped to SEG0
            SET_MUX_RATIO,
            self.height - 1,
            SET_COM_OUT_DIR | 0x08,  # scan from COM[N] to COM0
            SET_DISP_OFFSET,
            0x00,
            SET_COM_PIN_CFG,
```

```
            0x02 if self.width > 2 * self.height else 0x12,
            # timing and driving scheme
            SET_DISP_CLK_DIV,
            0x80,
            SET_PRECHARGE,
            0x22 if self.external_vcc else 0xF1,
            SET_VCOM_DESEL,
            0x30,  # 0.83*Vcc
            # display
            SET_CONTRAST,
            0xFF,  # maximum
            SET_ENTIRE_ON,  # output follows RAM contents
            SET_NORM_INV,  # not inverted
            SET_IREF_SELECT,
            0x30,  # enable internal IREF during display on
            # charge pump
            SET_CHARGE_PUMP,
            0x10 if self.external_vcc else 0x14,
            SET_DISP | 0x01,  # display on
        ):  # on
            self.write_cmd(cmd)
        self.fill(0)
        self.show()

    def poweroff(self):
        self.write_cmd(SET_DISP)

    def poweron(self):
        self.write_cmd(SET_DISP | 0x01)

    def contrast(self, contrast):
        self.write_cmd(SET_CONTRAST)
        self.write_cmd(contrast)

    def invert(self, invert):
        self.write_cmd(SET_NORM_INV | (invert & 1))

    def rotate(self, rotate):
        self.write_cmd(SET_COM_OUT_DIR | ((rotate & 1) << 3))
        self.write_cmd(SET_SEG_REMAP | (rotate & 1))

    def show(self):
        for Page in range( 0, 8 ):
            self.write_cmd( 0xB0 | ( Page & 0x0F ))
            self.write_cmd( 0x02 )
            self.write_cmd( 0x10 )
            self.write_data( self.buffer[Page<<7:(Page<<7)+128] )


class SSD1306_I2C(SSD1306):
```

```
    def __init__(self, width, height, i2c, addr=0x3C, external_vcc=False):
        self.i2c = i2c
        self.addr = addr
        self.temp = bytearray(2)
        self.write_list = [b"\x40", None]  # Co=0, D/C#=1
        super().__init__(width, height, external_vcc)

    def write_cmd(self, cmd):
        self.temp[0] = 0x80  # Co=1, D/C#=0
        self.temp[1] = cmd
        self.i2c.writeto(self.addr, self.temp)

    def write_data(self, buf):
        Page = 0

        for Page in range( 0, 8 ):
            self.i2c.writeto_mem( 0x3C, 0x80, ( 0xB0 | ( Page & 0x0F ) ).to_bytes(1, 0 ))
            self.i2c.writeto_mem( 0x3C, 0x80, b'\x02' )
            self.i2c.writeto_mem( 0x3C, 0x80, b'\x10' )
            buf1 = buf[Page<<7:(Page<<7)+128]
            self.write_list[1] = buf1

            self.i2c.writevto( 0x3C, self.write_list )

class SSD1306_SPI(SSD1306):
    def __init__(self, width, height, spi, dc, res, cs, external_vcc=False):
        self.rate = 10 * 1024 * 1024
        dc.init(dc.OUT, value=0)
        res.init(res.OUT, value=0)
        cs.init(cs.OUT, value=1)
        self.spi = spi
        self.dc = dc
        self.res = res
        self.cs = cs
        import time

        self.res(1)
        time.sleep_ms(1)
        self.res(0)
        time.sleep_ms(10)
        self.res(1)
        super().__init__(width, height, external_vcc)

    def write_cmd(self, cmd):
        self.spi.init(baudrate=self.rate, polarity=0, phase=0)
        self.cs(1)
        self.dc(0)
        self.cs(0)
        self.spi.write(bytearray([cmd]))
        self.cs(1)
```

```
def write_data(self, buf):
    self.spi.init(baudrate=self.rate, polarity=0, phase=0)
    self.cs(1)
    self.dc(1)
    self.cs(0)
    self.spi.write(buf)
    self.cs(1)
```

Figure A.9: the Display library given in lab, saved to the pico as ssd1306.py