ECEN 3074
Electrical Engineering Laboratory I
Design Project

Arduino FM Radio with Frequency Display

Cameron Biniamow

University of Nebraska-Lincoln
Department of Electrical and Computer Engineering
Peter Kiewit Institute
Date Completed: 12/04/2020

Abstract

This report provides a detailed understanding and demonstration of a FM radio with frequency display, which serves as the final project of ECEN 3074. Explanation of the entire project is specified, from the initial research to the design, implementation, production, and post-production analysis. The post-production analysis reviews the quality of the project, potential impacts, and areas for improvement.

Introduction

For the completion of ECEN 3074, a final design project is assigned with a large range of options for selection. A FM radio receiver with frequency display was selected and approved for the final project. Through researching numerous webpages for similar projects, it was determined that various options were available to meet the objectives. The success criteria of the project being is defined as:

- Create an operation FM radio that can tune to frequencies within the set band of 87.0 MHz - 108.0 MHz
- Read the current frequency and display on an LCD

Initial determinations were made to include the Arduino UNO as the microcontroller in the project. However, the remaining components could not be defined further than: an LCD display, a FM radio receiver, an audio amplifier, and switches for controlling the tuning of the radio.

Background and Theory

Inter-Integrated Circuit (I^2C) protocol allows the use of multiple peripheral digital integrated circuits to communicate with one or more controllers. I^2C requires only two connections to transmit and receive data between the controller and devices. The two signals included in the I^2C bus are: Serial Clock Line (SCL) and Serial Data Line (SDA). Since multiple devices can be connected to the same I^2C lines, each device has their own address in order to differentiate where data is transmitted to and received from. Due to the ability to connect numerous devices to a shared line, I^2C provides a more streamlined and simple circuit configuration as fewer connections are required.

The 1602 LCD allows for custom character creation as each of 32 blocks contain a 5 x 8 matrix of pixels. Through sending a HIGH signal to any one of the pixels, the pixel will illuminate. Custom character creation is performed by creating an array of values that correspond to the desired pixels in each character block. There are 8 values included in the array that are 5 bits wide. Starting with the top row of the character block, the first 5-bit wide value is defined in the array, where the most significant bit represents the leftmost column. The second 5-bit wide value then corresponds to the next row down on the character block.

```
byte CUSTOM_CHAR[8] = {
  B10000,
  B01000,
  B00100,
  B00010,
  B00010,
  B00100,
  B01000,
  B10000
};
```

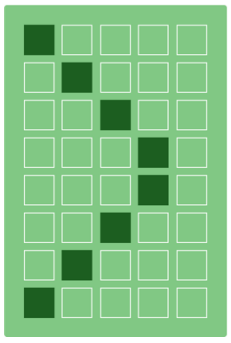

Figure I: Custom Character Definition with C Language Code & Corresponding Output on LCD

Control of the RDA5807 is performed through I^2C communication by addressing specific bits in defined registers. The RDA5807 includes registers 0x00, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, and 0x0F, where registers 0x0C – 0x0F are used for Radio Data System (RDS) block data. As can be seen below, the bit definition for registers 0x00 and 0x02 of the RDA5807 is shown. Bit definition of all I^2C registers for the RDA5807 can be seen in the RDA5807 datasheet.

REG	BITS	NAME	FUNCTION	DEFAULT
00H	15:8	CHIPID[7:0]	Chip ID.	0x58
02H	15	DHIZ	Audio Output High-Z Disable. *0 = High impedance; 1 = Normal operation*	0
	14	DMUTE	Mute Disable. *0 = Mute; 1 = Normal operation*	0
	13	MONO	Mono Select. *0 = Stereo; 1 = Force mono*	0
	12	BASS	Bass Boost. *0 = Disabled; 1 = Bass boost enabled*	0
	11	RCLK NON-CALIBRATE MODE	0=RCLK clock is always supply 1=RCLK clock is not always supply when FM work (when 1, RDA5807FP can't directly support -20℃ ~70℃ temperature. Only support ±20℃ temperature swing from tune point)	0
	10	RCLK DIRECT INPUT MODE	1=RCLK clock use the directly input mode	0
	9	SEEKUP	Seek Up. *0 = Seek down; 1 = Seek up*	0
	8	SEEK	Seek. *0 = Disable stop seek; 1 = Enable* *Seek begins in the direction specified by* SEEKUP *and ends when a channel is found, or the entire band has been searched.* *The* SEEK *bit is set low and the* STC *bit is set high when the seek operation completes.*	0
	7	SKMODE	Seek Mode 0 = wrap at the upper or lower band limit and continue seeking 1 = stop seeking at the upper or lower band limit	0
	6:4	CLK_MODE[2:0]	000=32.768kHz 001=12Mhz 101=24Mhz 010=13Mhz 110=26Mhz 011=19.2Mhz 111=38.4Mhz	000
	3	RDS_EN	RDS/RBDS enable If 1, rds/rbds enable	0
	2	NEW_METHOD	New Demodulate Method Enable, can improve the receive sensitivity about 1dB.	0
	1	SOFT_RESET	Soft reset. If 0, not reset; If 1, reset.	0

Figure II: RDA5807 I^2C Bit Definition for Registers 0x00 & 0x02

Design

The Arduino FM radio receiver consists of the following three design areas: component selection, circuit implementation, and software development. Additionally, the radio enclosure design is briefly explained.

Component Selection

The Arduino UNO development board is used to control and supply power to the FM radio. This microcontroller provides numerous I/O pins that can be taken advantage of to control the FM radio, LCD, and audio amplifier. Furthermore, the Arduino UNO supports I^2C communication, which is used for control of the FM radio and LCD. In addition to the hardware benefits, the Arduino IDE provides quick and easy configuration with the board while uploading a program. Arduino IDE also offers access to numerous libraries that reduce time spent developing software for the configuration of basic components.

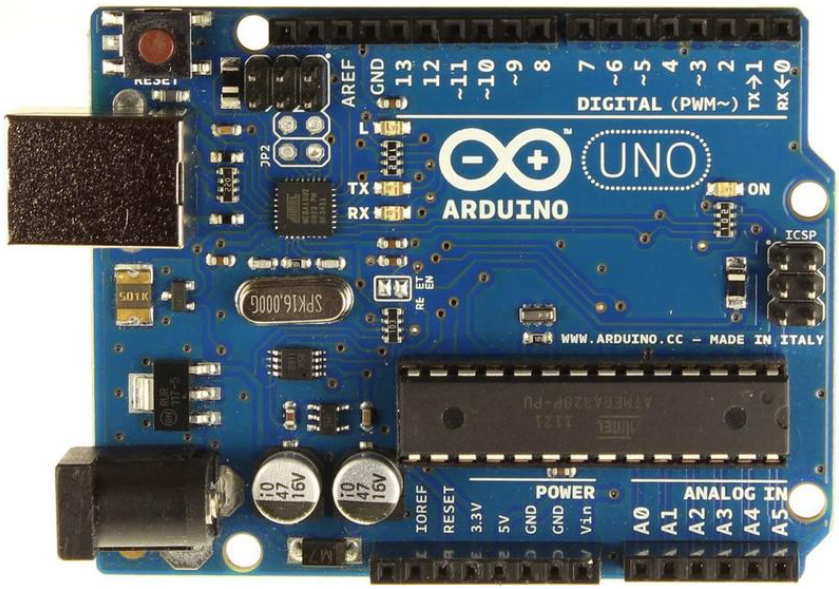

Figure III: Arduino UNO Development Board

The RDA5807M RRD-102V2.0 FM radio module allows for use of the RDA5807M broadcast FM radio tuner on a compact single chip where the need for external components is reduced. This provides a cost-effective and streamlined solution that produces minimal noise and station identification by supporting I^2C serial data communication.

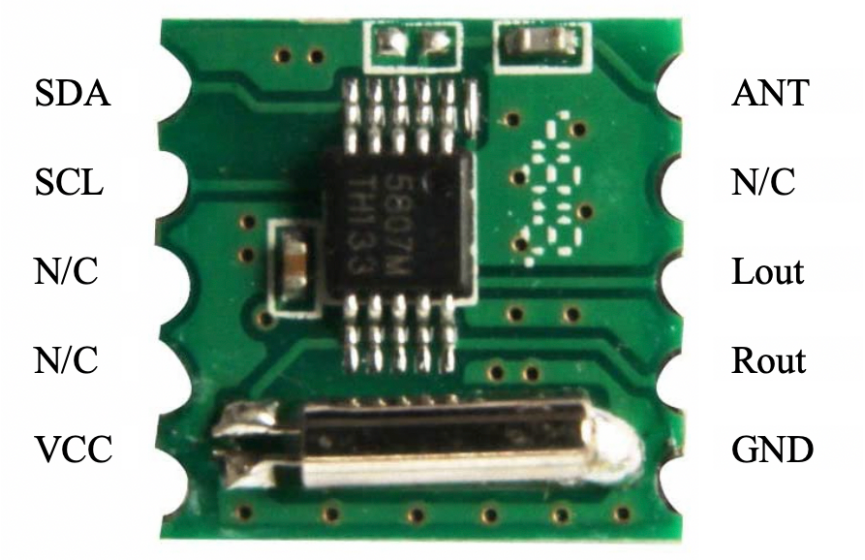

Figure IV: RDA5807M RRD-102V2.0 FM Radio Module Pinout

The PAM8403 2-channel digital audio amplifier module produces a 3-Watt output with a 4Ω load and a 5 Volt power supply. This module reduces the need for external components and therefore delivers a higher-definition sound quality with minimal connections. Furthermore, the PAM8403 amplifier module includes a mounted potentiometer for volume control that can switch to cut off the power supply directly. Thus, the FM radio can be turned off without disconnecting the power supply from the Arduino.

Figure V: PAM8406 2-Channel Digital Audio Amplifier Module

Due to the 3-Watt output of the PAM8403, two 4Ω 3-Watt speakers are included in the project. Use of these speakers allows for a higher quality sound that is produced from the PAM8403 amplifier.

Figure VI: 4Ω 3-Watt Speakers

With frequency display being a critical measure of this project, the 1602 serial LCD module is used. While a standard 16-pin LCD interface would suffice for this project, this LCD module supports I^2C communication, which drastically reduces the required connections to the Arduino from 16 to 4.

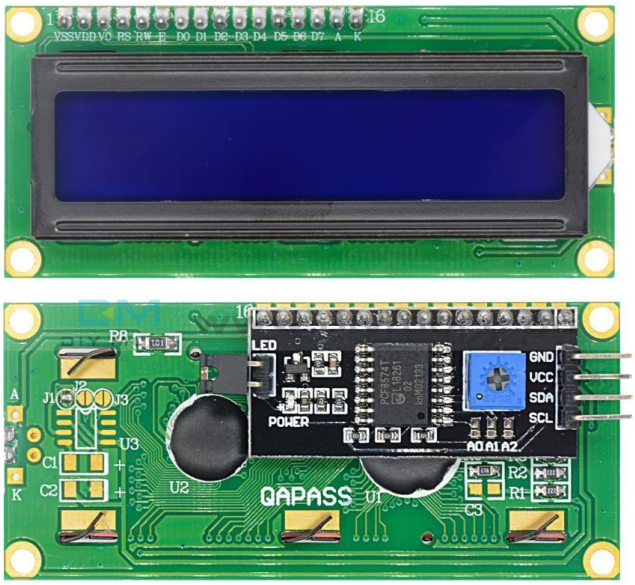

Figure VII: 1602 Serial LCD Module

Single Pole Single Throw (SPST) pushbutton switches are used for the seek up, seek down, and reset features of the FM radio. The pushbutton switch allows for simple interfacing with the Arduino and requires few connections.

Figure VIII: SPST Pushbutton Switches

Circuit Implementation

Design of the circuit includes all components listed above to construct a fully functional FM radio receiver with frequency display. Since the RDA5807 FM radio receiver, PAM8403 audio amplifier, and 1602 serial LCD come in single-chip modules, the circuit configuration and implementation is streamlined as almost no external components are required to operate the chips. This considerably reduces the need for in depth circuit analysis and the possibilities for error while constructing the circuit. The I^2C communication with the RDA5807 and LCD provide simple connections from Arduino UNO pins A4 and A5 to the devices. From the PAM8403, basic connections are made to the 4Ω 3-Watt speakers to produce the sound of the audio signal. The three pushbuttons used for seeking up, seeking down, and resetting the Arduino; the switches require simple connections to ground, 2 digital pins on the Arduino, and the reset pin. Again, while these single-chip modules allow for simplified use when designing the circuit, the need for no external components provides a better and more refined output as noise and risk of error are greatly reduced. As can be seen below, the final circuit schematic for the FM radio with frequency display is shown.

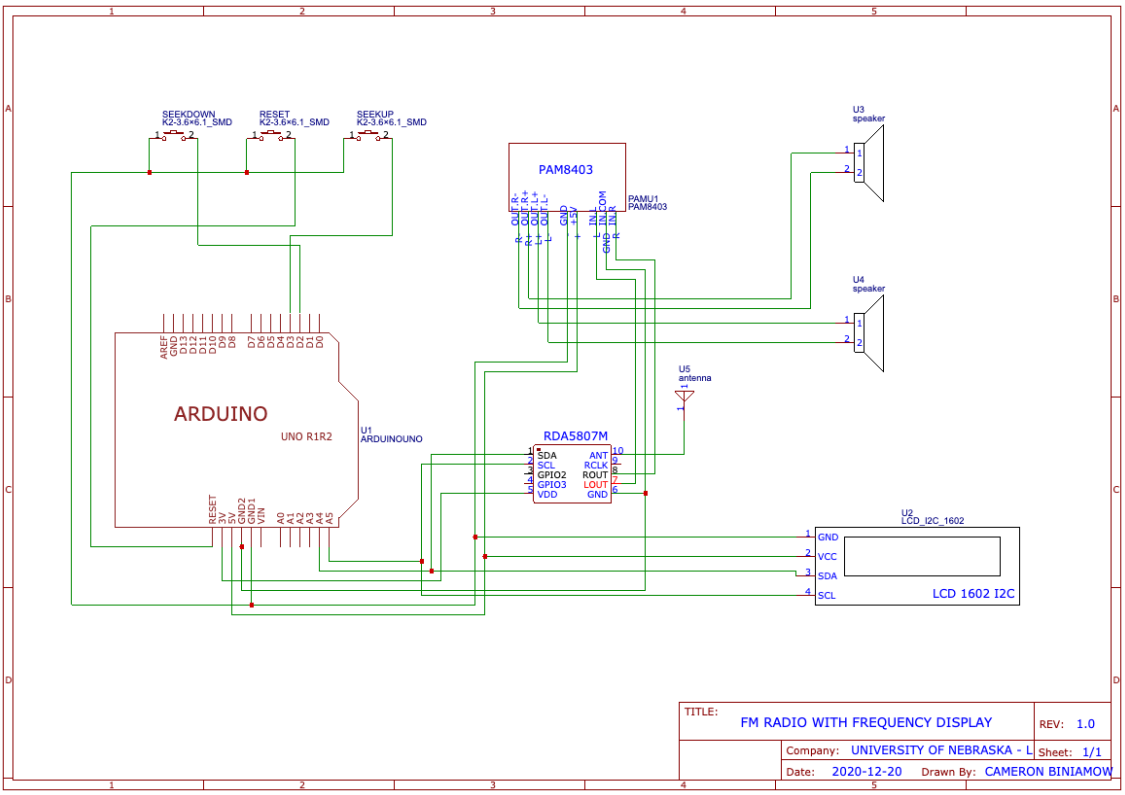

Figure IX: Arduino FM Radio with Frequency Display Schematic

Software Development

The logic and software are designed largely around the configuration and control of the RDA5807 module through the Arduino UNO. Additionally, the LCD is configured and controlled from the Arduino as well as from the state of the pushbutton switches. Design for the RDA5807 revolves almost entirely around I^2C communication. As can be seen in the background section of this report, the I^2C protocol and a brief understanding is explained. Use of I^2C communication allows for control of the RDA5807 as well as for reading data received by the RDA5807. With use of the "radio.h" library, full control of the RDA5807 is gained. As can be seen below, the following block diagrams take advantage of the "radio.h" library to design the necessary software.

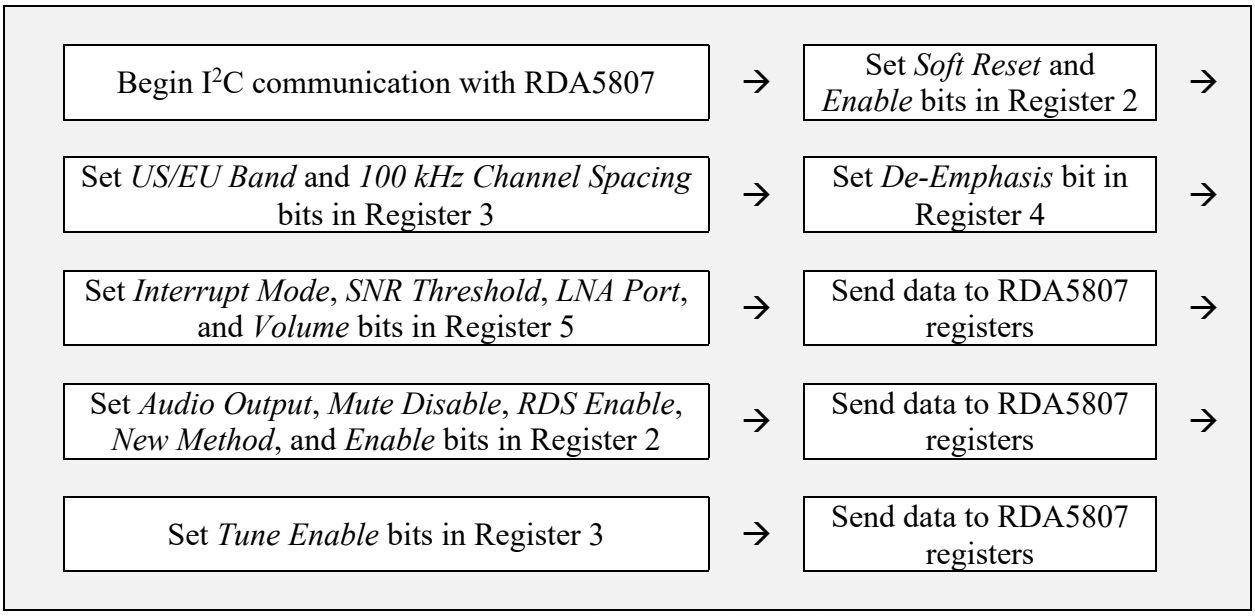

Figure X: RDA5807 Initialization Procedure Block Diagram

The RDA5807 *Seek Up* and *Seek Down* procedures are designed to operate by configuring the bits of register 2. For the *Seek Up* procedure, the *Seek Up* and *Seek Enable* bits are set then the data is sent to the RDA5807. As for the *Seek Down* procedure, the complement of *Seek Up* and *Seek Enable* bits are set then the data is sent to the RDA5807.

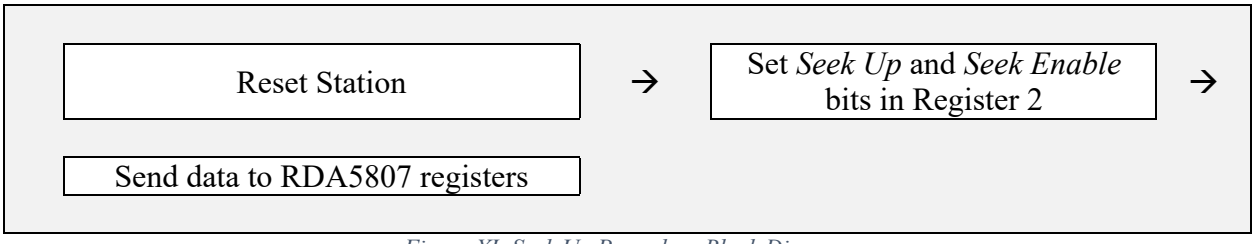

Figure XI: Seek Up Procedure Block Diagram

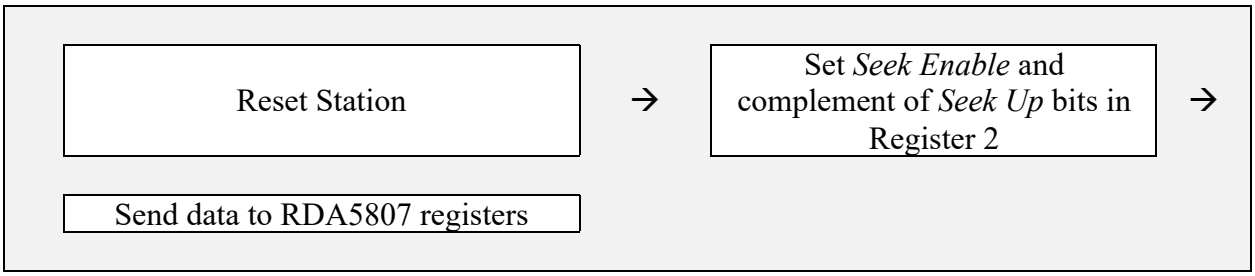

Figure XII: Seek Down Procedure Block Diagram

The RDA5807 *Get Frequency* procedure is designed to operate by reading the state of the bits in registers 3 and A then converting and returning the value in MHz.

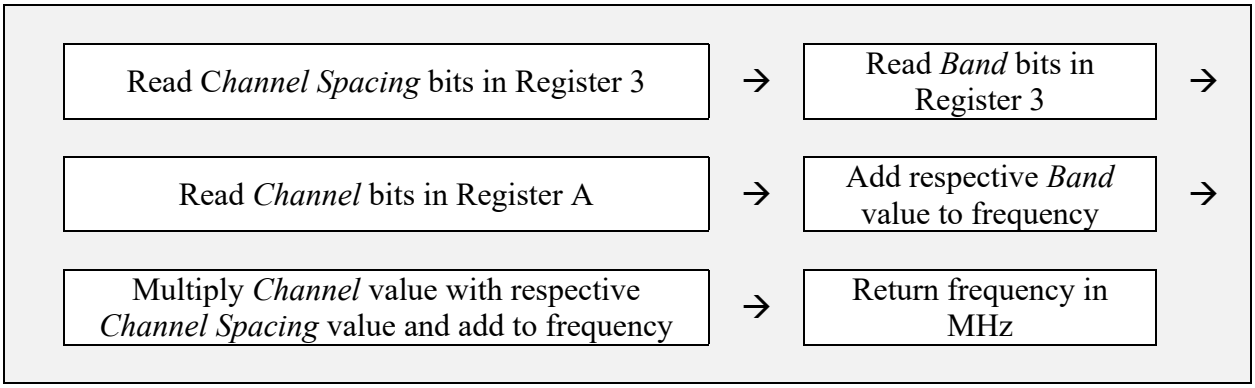

Figure XIII: Get Frequency Procedure Block Diagram

The *Radio Status* procedure allows for the ability to read the signal strength of the current frequency on the RDA5807 and display signal characters on the LCD. The signal strength is returned as a value between 0 and 63, where 63 is the highest signal strength. Since the signal strength is displayed in the form of ascending height bars next to an antenna symbol, the characters need to be created in order to display on the LCD. As stated in the Background section of this report, LCD characters are created within 5 x 8-pixel blocks where writing a 1 to a pixel illuminates the pixel. Therefore, design of the antenna and signal strength symbols are as follows:

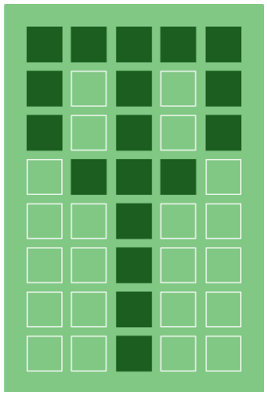

Figure XIV: Antenna Character for Display on LCD

The signal strength characters are designed have the ability to display 6 different characters across 2 blocks on the LCD. The characters shown below indicate the signal strength is within the highest range. Given the signal strength is within the lowest range, only the first bar of the left block would be illuminated.

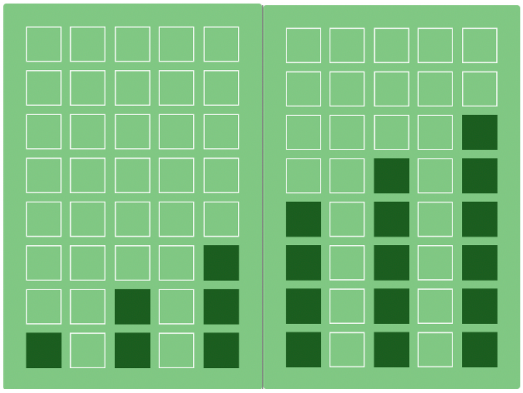

Figure XV: Signal Strength Characters for Display on LCD

Station tuning is designed to be completed through pressing one of two pushbutton switches. Due to the simple nature of the pushbutton switch, software design consists of setting a digital pin on the Arduino as an input and force a HIGH signal on the pin with a pullup resistor. Given the pushbutton is pressed, a LOW signal is sent to the pin and FM radio seeks up or down depending on which switch is pressed.

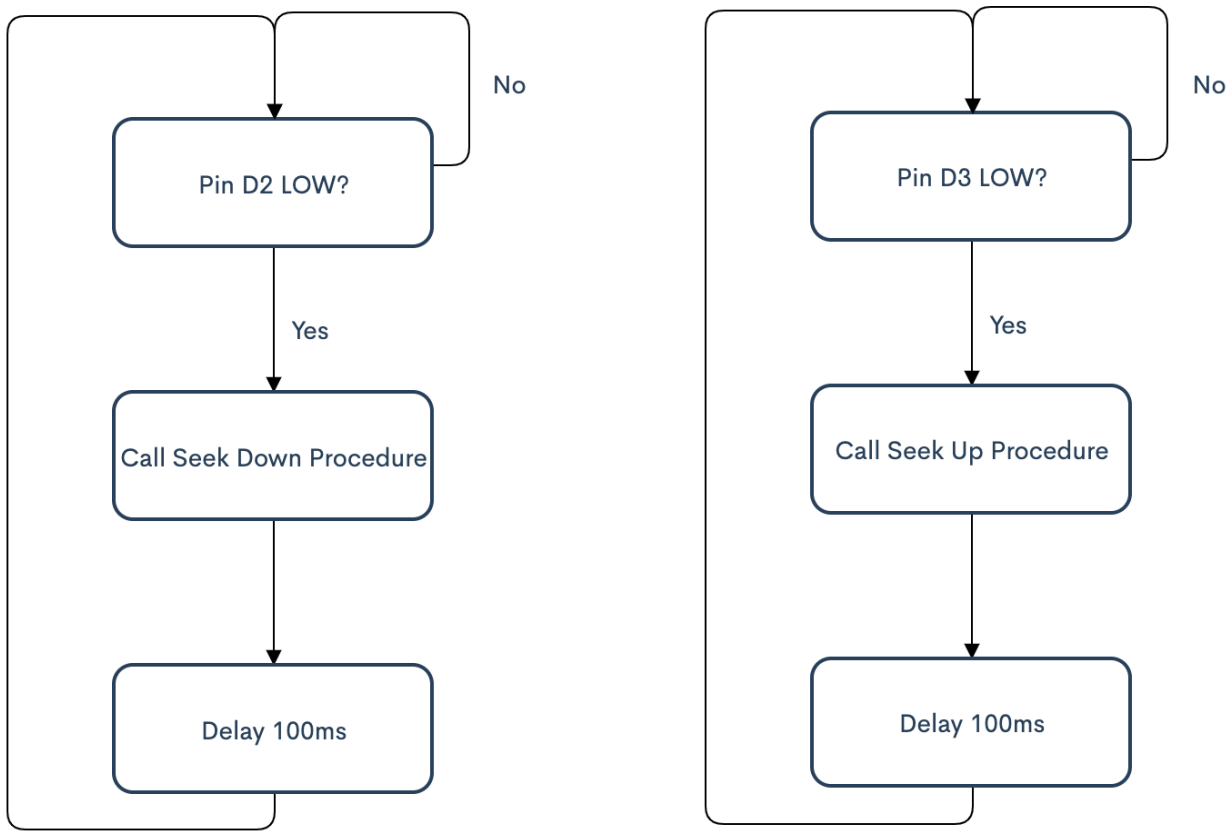

Figure XVI: Seek Up & Seek Down Flowcharts

Radio Enclosure

Design of the radio enclosure focused on providing ample space for the project while allowing for the ability to easily modify, demonstrate and observe the functionality of the FM radio. Due to these constraints, a plexiglass enclosure is selected and designed with a wood base for stability. The enclosure is sized with generous dimensions at 12” wide, 6” tall, and 6” deep. While the 4 vertical sides of the enclosure are glued together, the top and back panels are attached to a hinge that allows for easy access to project when needed. Otherwise, the top panel sits on top of the 4 vertical panels and assists in protecting the components and connections of the FM radio.

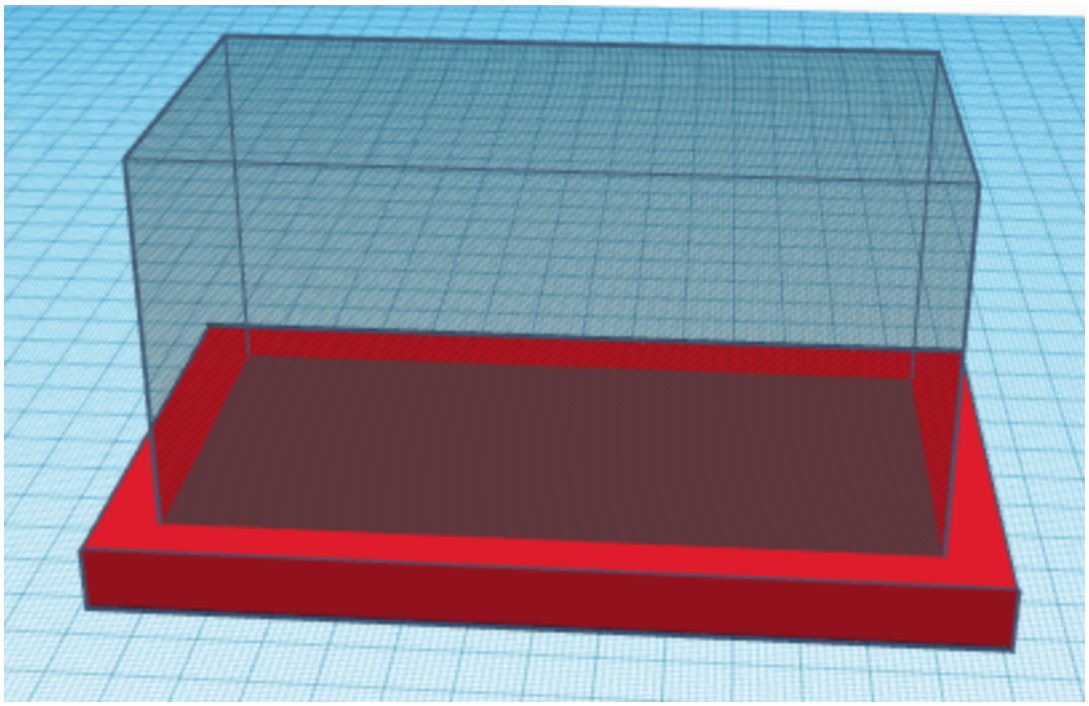

Figure XVII: FM Radio Enclosure Design

Procedure and Implementation

Similar to the design, the procedure and implementation of the FM radio receiver with frequency display is split into two main areas of focus. These areas are *circuit implementation* and *software development*.

Circuit Implementation

As stated in the design section of this report, the circuit implementation is made drastically easier by using the single-chip modules. Thus, only the following connections are performed:

1. RDA5807
    a. Arduino 3.3 Volt pin → RDA5807 pin 5

b. Arduino GND pin → RDA5807 pin 6

c. Arduino A4 → RDA5807 pin 1

d. Arduino A5 → RDA5807 pin 2

e. Antenna → RDA5807 pin 10

f. RDA5807 pin 7 → PAM8403 *IN.L* pin

g. RDA5807 pin 8 → PAM8403 *IN.R* pin

h. RDA5807 pin 6 → PAM8403 *IN.GND* pin

2. PAM8403

a. Arduino 5 Volt pin → PAM8403 *+5V* pin

b. Arduino GND pin → PAM8403 *GND* pin

c. Left speaker positive terminal → PAM8403 *OUT.L+* pin

d. Left speaker negative terminal → PAM8403 *OUT.L-* pin

e. Right speaker positive terminal → PAM8403 *OUT.R+* pin

f. Right speaker negative terminal → PAM8403 *OUT.R-* pin

3. 1602 Serial LCD

a. Arduino 5V pin → LCD pin 2

b. Arduino GND pin → LCD pin 1

c. Arduino A4 → LCD pin 3

d. Arduino A5 → LCD pin 4

4. Pushbutton Switches
    a. Arduino GND pin → Seek Down pushbutton pin 1
    b. Arduino GND pin → Seek Up pushbutton pin 1
    c. Arduino GND pin → Reset pushbutton pin 1
    d. Arduino D2 → Seek Down pushbutton pin 2
    e. Arduino D3 → Seek Up pushbutton pin 2
    f. Arduino RESET pin → Reset pushbutton pin 2

Software Development

The procedures for the implementation of software for the FM radio receiver with frequency display is split up into the following sections:

1. Program setup
    a. Define pins used for *Seek Up* and *Seek Down* pushbuttons
    b. Initialize *radio* library
    c. Begin I^2C communication with LCD
    d. Create antenna and signal strength characters for display on LCD
    e. Configure *Seek Up* and *Seek Down* pins as pullup inputs
    f. Initialize LCD / Turn LCD backlight ON / Clear LCD
    g. Initialize RDA5807 / Mute RDA5807 / Tune to stored station / Unmute RDA5807

```
#define SEEK_DOWN   2                     // SEEK DOWN PUSHBUTTON -> PD2
#define SEEK_UP     3                     // SEEK UP PUSHBUTTON -> PD3

Radio radio;
LiquidCrystal_I2C lcd(0x27,16,2);         // LCD ADDRESS 0x27. 16 COLUMNS. 2
ROWS.


/* CREATE ANTENNA CHARACTER FOR DISPLAY ON LCD */

byte CHR_ANTENNA[8] = {
  B11111, B10101, B10101, B01110, B00100, B00100, B00100, B00100
```

```
};

/* CREATE SIGNAL CHARACTERS FOR DISPLAY ON LCD */

byte CHR_STRENGTH[6][8] = {{
  B00000, B00000, B00000, B00000, B00000, B00000, B00000, B10000
}, {
  B00000, B00000, B00000, B00000, B00000, B00000, B00100, B10100
}, {
  B00000, B00000, B00000, B00000, B00000, B00001, B00101, B10101
}, {
  B00000, B00000, B00000, B00000, B10000, B10000, B10000, B10000
}, {
  B00000, B00000, B00000, B00100, B10100, B10100, B10100, B10100
}, {
  B00000, B00000, B00001, B00101, B10101, B10101, B10101, B10101
}
};


void setup() {
  pinMode(SEEK_UP,INPUT_PULLUP);            // CONFIGURE INPUT PINS
  pinMode(SEEK_DOWN,INPUT_PULLUP);

  lcd.init();                               // INITIALIZE LCD
  lcd.backlight();                          // TURN LCD BACK LIGHT ON
  lcd.clear();                              // CLEAR LCD

  radio.init();                             // INITIALIZE RADIO
  radio.setMute(true);                      // MUTE RADIO DURING STARTUP
  radio.tuneTo(getEepromFreq());            // TUNE TO LAST SAVED STATION
  delay(1000);                              // 1 SECOND DELAY FOR TUNING
  radio.setMute(false);                     // UNMUTE RADIO STARTUP
  radio.setVolume(15);                      // SET MAX VOLUME

}
```

Figure XVIII: FM Radio Program Setup Code

2. Storing current frequency to EEPROM
    a. Store hundreds place of frequency in EEPROM address 6
    b. Store tens places of frequency in EEPROM address 7
    c. Store ones places of frequency in EEPROM address 8
    d. Store tenths places of frequency in EEPROM address 9
    e. Store hundredths places of frequency in EEPROM address 10

```
void setEepromFreq(float frequency) {
  unsigned short freq = frequency * 100;       // 91.5 -> 9150
  EEPROM.update(6, (freq / 10000));            // 0
  EEPROM.update(7, (freq / 1000) % 10);        // 9
  EEPROM.update(8, (freq / 100) % 10);         // 1
  EEPROM.update(9, (freq / 10) % 10);          // 5
  EEPROM.update(10, freq % 10);                // 0
}
```

Figure XIX: Storing Current Frequency in EEPROM Procedure

3. Loading frequency from EEPROM
    a. Load hundreds place of frequency in EEPROM address 6
    b. Load tens places of frequency in EEPROM address 7
    c. Load ones places of frequency in EEPROM address 8
    d. Load tenths places of frequency in EEPROM address 9
    e. Load hundredths places of frequency in EEPROM address 10

```
float getEepromFreq() {
  float frequency = 0.0f;
  frequency += EEPROM.read(6)  * 100.0f;       // 0.00
  frequency += EEPROM.read(7)  * 10.0f;        // 90.00
  frequency += EEPROM.read(8);                 // 91.00
  frequency += EEPROM.read(9)  / 10.0f;        // 91.50
  frequency += EEPROM.read(10) / 100.0f;       // 91.50
  return frequency;
}
```

Figure XX: Loading Last Stored Frequency from EEPROM Procedure

4. Displaying radio information on LCD
    a. Read radio signal strength
    b. Create respective signal strength characters
    c. Set LCD cursor to second row, first column
    d. Write antenna character & signal strength characters

```
void drawStatusLine() {
  switch (radio.state.signalStrength) {
    case 0 ... 13:                          // SIGNAL STRENGTH 1
      lcd.createChar(1, CHR_STRENGTH[0]);
      lcd.createChar(2, CHR_STRENGTH[0]);
      break;
    case 14 ... 23:                         // SIGNAL STRENGTH 2
      lcd.createChar(1, CHR_STRENGTH[1]);
```

```
      lcd.createChar(2, CHR_STRENGTH[0]);
      break;
    case 24 ... 33:                          // SIGNAL STRENGTH 3
      lcd.createChar(1, CHR_STRENGTH[2]);
      lcd.createChar(2, CHR_STRENGTH[0]);
      break;
    case 34 ... 43:                          // SIGNAL STRENGTH 4
      lcd.createChar(1, CHR_STRENGTH[2]);
      lcd.createChar(2, CHR_STRENGTH[3]);
      break;
    case 44 ... 53:                          // SIGNAL STRENGTH 5
      lcd.createChar(1, CHR_STRENGTH[2]);
      lcd.createChar(2, CHR_STRENGTH[4]);
      break;
    case 54 ... 63:                          // SIGNAL STRENGTH 6
      lcd.createChar(1, CHR_STRENGTH[2]);
      lcd.createChar(2, CHR_STRENGTH[5]);
      break;
  }

  lcd.setCursor(0,1);                        // MOVE CURSOR TO FIRST COLUMN, SECOND
ROW
  lcd.write(byte(0));                        // PRINT ANTENNA CHARACTER ON LCD
  lcd.write(byte(1));                        // PRINT SIGNAL STRENGTH 1 - 3
  lcd.write(byte(2));                        // PRINT SIGNAL STRENGTH 4 - 6
}
```

Figure XXI: Radio Information LCD Display Procedure

5. Infinite Loop
    a. Update radio data
    b. Print “FM”, current frequency, and “MHz” on first row of LCD
    c. Display signal strength on LCD
    d. Store current frequency in EEPROM
    e. Check state of pin 2
        i. Seek down if LOW signal
    f. Check state of pin 3
        i. Seek up if LOW signal

```
void loop() {
  radio.updateStatus();                    // UPDATE RADIO DATA
  lcd.setCursor(0,0);                      // LCD CURSOR: COL. 0, ROW 0
  lcd.print("FM ");

  if(radio.state.frequency >= 100.0){
    lcd.setCursor(6,0);
    lcd.print(radio.state.frequency);      // PRINT STATION FREQUENCY
```

```
  }
  else{
    lcd.setCursor(6,0);
    lcd.print(" ");
    lcd.print(radio.state.frequency);       // PRINT STATION FREQUENCY
  }

  lcd.setCursor(12,0);                      // LCD CURSOR: COL. 13, ROW 0
  lcd.print(" MHz");

  drawStatusLine();                         // PRINT SIGNAL STRENGTH

  if(lastFreq != radio.state.frequency){    // STORE CURRENT STATION IN EEPROM
    setEepromFreq(radio.state.frequency);
  }


  if(digitalRead(2) == LOW){                // SEEK DOWN WHEN PIN 2 IS LOW
    radio.seekDown();
    delay(100);
  }

  if(digitalRead(3) == LOW){                // SEEK UP WHEN PIN 3 IS LOW
    radio.seekUp();
    delay(100);
  }
}
```

Figure XXII: FM Radio Infinite Loop

Results and Discussion

The FM radio receiver with frequency display was implemented from the designs included in this report. As can be seen below, a fully functioning FM radio with frequency display is shown. The final design operates as expected and allows for FM radio tuning, audio signal output, variable volume control, LCD display of the current frequency and signal strength, and an external reset.

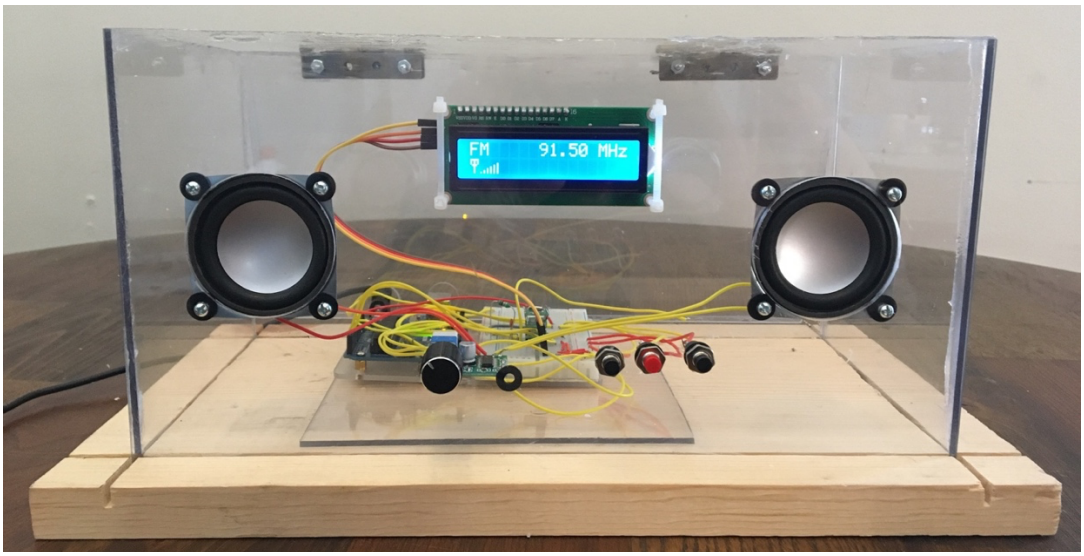

Figure XXIII: Final FM Radio Design (Front View)

The final design is presented in a plexiglass enclosure that houses and protects the project from potential damage. By rotating the control knob on the left side of the radio, the volume is varied and can be completely turned off through rotating all the way counterclockwise.

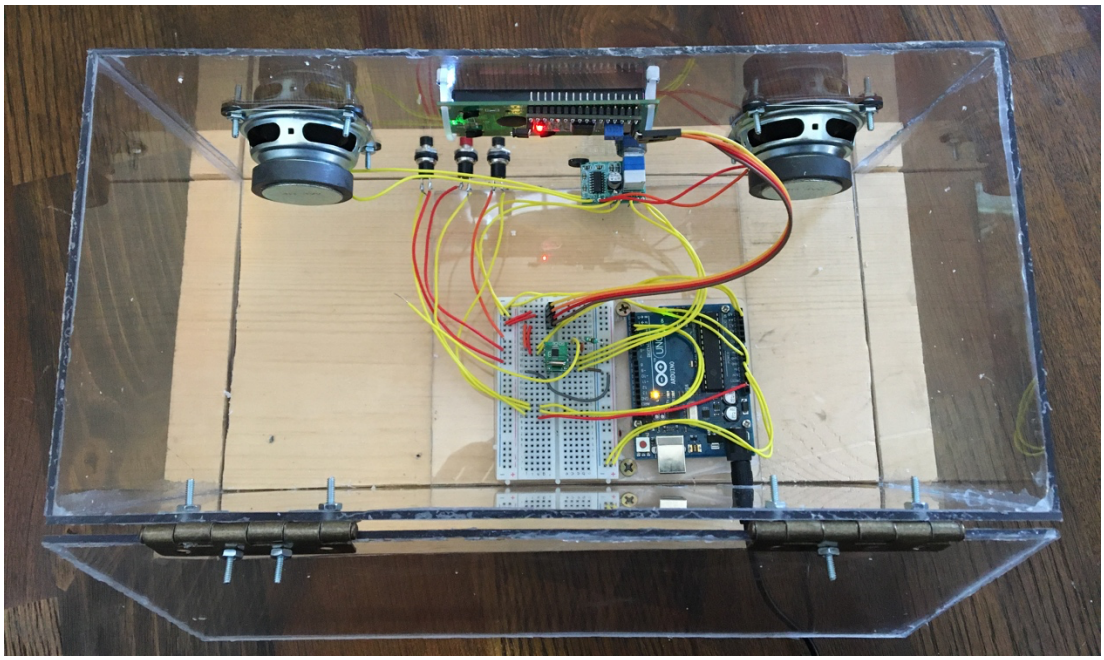

Figure XXIV: Final FM Radio Design (Top View)

Connections were made on a breadboard connected to the Arduino UNO. This allowed for easily modifying the circuit, given that the design changes or issues arise. Power for the FM radio is provided from the DC power jack on the Arduino. It is also possible to power the FM radio through the USB port on the Arduino.

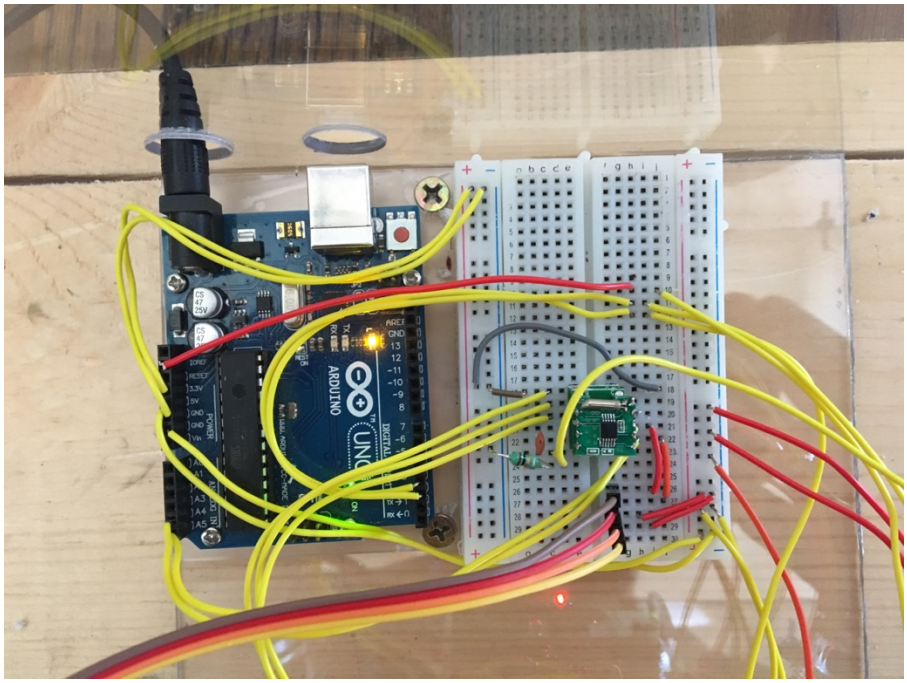

Figure XXV: Arduino UNO & Circuit Inside of the FM Radio

Observations

The FM radio was observed as fully operational and meeting the defined success criteria by providing the ability to tune to various FM frequencies and by displaying the current frequency on an LCD. However, further observations were made in relation to the functionality of the FM radio as well as possible advancements to the project.

When testing with the pushbutton switches for the tuning up and down, pins PD0 and PD1 on the Arduino were selected as the inputs. It was quickly determined that using pins PD0

and PD1 caused issues as they are the serial data receive and transmit pins. This caused the pushbuttons to behave inconsistently and unreliably. Following moving the switches to pins PD2 and PD3, proper operation of the seeking functions of the FM radio occurred.

A function of the FM radio that was discovered and observed through the creation of the project was the Radio Data System (RDS). RDS allows for the ability to send and receive data for a currently tuned into station. This data varies, but some of the features included: station name, song and artist names, traffic information, and radio station genre. Use of RDS could not be fully understood in how the data is transmitted, thus, implementation of RDS was not included in the FM radio design.

Difficulties

While designing and implementing the FM radio, numerous difficulties were encountered throughout the project. In particular, the majority of difficulties were experienced with frequency tuning on the FM radio receiver module. Initial designs included the use of the TEA5767 FM radio receiver module rather than the RDA5807. The TEA5767 module provided a similar set of functions as the RDA5807 with additional features, such as on-chip audio jack for the output signal and antenna. Issues with the TEA5767 became apparent quickly as the module was not being detected through I^2C communication. Despite using numerous different libraries found online that included full control of the device, only few controls worked. Specifically, the TEA5767 could be configured to mute and unmute, but any attempts at seeking up or down would not prove unsuccessful. The libraries were reviewed for errors while referencing the TEA5767 datasheet and the issue remains undetermined. Further testing of the TEA5767 was

conducted with a second, different TEA5767 module and similar results were produced. After the difficulties of interfacing with the TEA5767 occurred, the design was revised to include the RDA5807FP FM radio IC instead.

After switching the FM radio receiver from the TEA5767 to the RDA5807FP IC, the design required the inclusion of external components such as capacitors, resistors, and oscillators to control the chip. The new design was constructed and tested for operation. It was determined that the RDA5807FP IC operated as expected and provided full control of the chip features, including the ability to seek up or down to different frequencies. While this breakthrough propelled the project to an operational phase, the circuit subsequently increased in size tremendously in terms of components included, connections required, and space consumed. Quickly it was determined that the project would face greater risk of failure as well as noise produced while using such a design. Furthermore, reception difficulties arose during continued testing, causing the FM radio to be inoperable for extended periods of time. This appeared to be caused by I^2C communication connection issues that could not specifically be pinpointed. Once again, the design was revised to include the RDA5807F module, where no external components are required for use other than the Arduino. This provided a more simplified circuit and allowed for higher definition output sound quality.

Once the design was switched to include the RDA5807F module, the next hurdle was introduced. The difficulties were now with hardware configuration as the RDA5807F module is provided in a surface mount package that does not easily align with any DIP adapter readily available. This problem required the need to slightly modify an SOIC to DIP adapter by

soldering short pieces of wire from the pins of the RDA5807 module to the adapter. After performing the modifications, the RDA5807 module was able to easily be used on a breadboard.

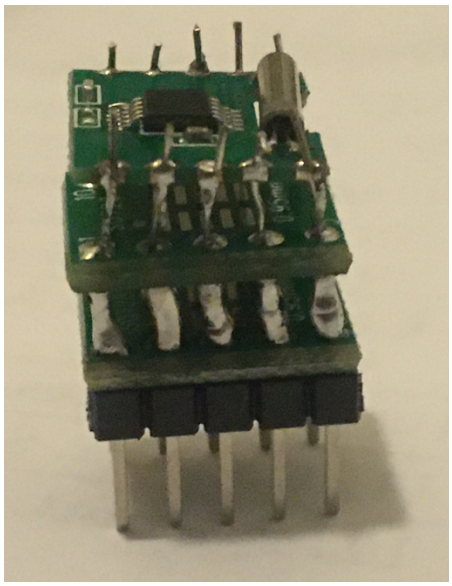

Figure XXVI: Modified RDA5807 Module in DIP Package

Initial designs included a rotary encoder to the tuning of the FM radio. The rotary encoder would allow for tuning to higher frequencies by rotating the encoder clockwise and tuning to lower frequencies by rotating counterclockwise. Testing was performed on the rotary encoder to understand the configuration of operation of the device. The conclusions from the independent testing were that the rotary encoder would work well for this project as it produced expected results. However, when including the rotary encoder into the FM radio circuit, it was determined that the encoder produced significant switch bounce and did not operate well with the RDA5807 module. Even after including logic to debounce the rotary encoder, the FM would continue to operate inconsistently while tuning to other frequencies. The decision was made to revise the design to include pushbutton switches for the seeking up and down rather than using the rotary encoder.

Recommendations

After constructing, testing, and reviewing the FM radio, some recommendations are suggested. These recommendations are as follows:

- Explore various options for circuit construction other than with a breadboard. While the breadboard does allow for easy use and modification, the potential for loose or poor connections is considerably higher than using perfboard or even designing a Printed Circuit Board (PCB).
- Further review and research should be conducted to determine alternative solutions for the connections to the LCD. Currently, the LCD is connected through I^2C communication along with the RDA5807 module. While listening to the FM radio, noise can be heard in the output, which has been determined to be caused by the sending of data to the LCD.

Standards and Society Impact

The standards and societal impact of the FM radio remain somewhat unknown, although, after review, some changes could occur to reduce societal impacts. Such impacts include the power consumption of the FM radio as there is no ON/OFF switch for the circuit as whole. While the volume can be turned all the way down and cut off power to the PAM8403, there is no method for cutting off power to the Arduino UNO or LCD without disconnecting the DC power jack from the Arduino UNO. This lack of ON/OFF switch allows the FM radio to remain ON and consuming power when not in use and plugged into a power source. Further design modifications could be made to include such a switch, thus, reducing power consumed by the device.

Conclusion

Design, implementation, production, and post-production analysis of a FM radio receiver is provided in this report. The FM radio was constructed and tested with the concluding results being that proper operation of a fully functioning FM radio with frequency display was achieved. These determinations were made through observing the FM radio tune through the US FM radio band and output the audio signal of the respective frequency displayed on the LCD. While the FM radio did meet all required success criteria, numerous finding and recommendations to improve the performance of the FM radio were made. This project allowed for a more in depth, start-to-finish understanding of the electrical engineering design process than what has been provided in previous laboratories and courses. It has been determined that all information and experiences of this project will prove beneficial in future courses and professional experiences.

References

- FM Radio with Arduino UNO & RDA5807M: https://maker.pro/arduino/projects/simple-fm-radio-receiver-with-arduino-uno-and-rda5807m
- RDA5807 Datasheet: https://opendevices.ru/wp-content/uploads/2015/10/RDA5807FP.pdf
- Arduino Radio with RDS: https://hackaday.io/project/9009-arduino-radio-with-rds
- Radio Data System: http://www.g.laroche.free.fr/english/rds/groupes/listeGroupesRDS.htm

Appendix

Figure XXVII: FM Radio Main C Language Code

```
#include <EEPROM.h>
#include <LiquidCrystal_I2C.h>
#include <radio.h>
#include <Wire.h>


#define SEEK_DOWN   2                     // SEEK DOWN PUSHBUTTON -> PD2
#define SEEK_UP     3                     // SEEK UP PUSHBUTTON -> PD3

Radio radio;
LiquidCrystal_I2C lcd(0x27,16,2);         // LCD ADDRESS 0x27. 16 COLUMNS. 2 ROWS.


/* CREATE ANTENNA CHARACTER FOR DISPLAY ON LCD */

byte CHR_ANTENNA[8] = {
  B11111, B10101, B10101, B01110, B00100, B00100, B00100, B00100
};

/* CREATE SIGNAL CHARACTERS FOR DISPLAY ON LCD */

byte CHR_STRENGTH[6][8] = {{
  B00000, B00000, B00000, B00000, B00000, B00000, B00000, B10000
}, {
  B00000, B00000, B00000, B00000, B00000, B00000, B00100, B10100
}, {
  B00000, B00000, B00000, B00000, B00000, B00001, B00101, B10101
}, {
  B00000, B00000, B00000, B00000, B10000, B10000, B10000, B10000
}, {
  B00000, B00000, B00000, B00100, B10100, B10100, B10100, B10100
}, {
  B00000, B00000, B00001, B00101, B10101, B10101, B10101, B10101
}
};


void setup() {
  pinMode(SEEK_UP,INPUT_PULLUP);            // CONFIGURE INPUT PINS
  pinMode(SEEK_DOWN,INPUT_PULLUP);

  lcd.init();                               // INITIALIZE LCD
  lcd.backlight();                          // TURN LCD BACK LIGHT ON
  lcd.clear();                              // CLEAR LCD

  radio.init();                             // INITIALIZE RADIO
  radio.setMute(true);                      // MUTE RADIO DURING STARTUP
  radio.tuneTo(getEepromFreq());            // TUNE TO LAST SAVED STATION
  delay(1000);                              // 1 SECOND DELAY FOR TUNING
  radio.setMute(false);                     // UNMUTE RADIO STARTUP
  radio.setVolume(15);                      // SET MAX VOLUME

}

void loop() {
  radio.updateStatus();                     // UPDATE RADIO DATA
  lcd.setCursor(0,0);                       // LCD CURSOR: COL. 0, ROW 0
  lcd.print("FM ");
```

```
  if(radio.state.frequency >= 100.0){
    lcd.setCursor(6,0);
    lcd.print(radio.state.frequency);       // PRINT STATION FREQUENCY
  }
  else{
    lcd.setCursor(6,0);
    lcd.print(" ");
    lcd.print(radio.state.frequency);       // PRINT STATION FREQUENCY
  }

  lcd.setCursor(12,0);                      // LCD CURSOR: COL. 13, ROW 0
  lcd.print(" MHz");

  drawStatusLine();                         // PRINT SIGNAL STRENGTH

  if(lastFreq != radio.state.frequency){    // STORE CURRENT STATION IN EEPROM
    setEepromFreq(radio.state.frequency);
  }


  if(digitalRead(2) == LOW){                // SEEK DOWN WHEN PIN 2 IS LOW
    radio.seekDown();
    delay(100);
  }

  if(digitalRead(3) == LOW){                // SEEK UP WHEN PIN 3 IS LOW
    radio.seekUp();
    delay(100);
  }
}


/* LOAD LAST STORED FREQUENCY FROM EEPROM */

float getEepromFreq() {
  float frequency = 0.0f;
  frequency += EEPROM.read(6)  * 100.0f;       // 0.00
  frequency += EEPROM.read(7)  * 10.0f;        // 90.00
  frequency += EEPROM.read(8);                 // 91.00
  frequency += EEPROM.read(9)  / 10.0f;        // 91.50
  frequency += EEPROM.read(10) / 100.0f;       // 91.50
  return frequency;
}


/* STORE LAST TUNED FREQUENCY TO EEPROM */

void setEepromFreq(float frequency) {
  unsigned short freq = frequency * 100;       // 91.5 -> 9150
  EEPROM.update(6, (freq / 10000));            // 0
  EEPROM.update(7, (freq / 1000) % 10);        // 9
  EEPROM.update(8, (freq / 100) % 10);         // 1
  EEPROM.update(9, (freq / 10) % 10);          // 5
  EEPROM.update(10, freq % 10);                // 0
}


/* CREATE AND PRINT SIGNAL STRENGTH OF CURRENT STATION ON LCD */

void drawStatusLine() {
  switch (radio.state.signalStrength) {
    case 0 ... 13:                             // SIGNAL STRENGTH 1
      lcd.createChar(1, CHR_STRENGTH[0]);
      lcd.createChar(2, CHR_STRENGTH[0]);
```

```
      break;
    case 14 ... 23:                          // SIGNAL STRENGTH 2
      lcd.createChar(1, CHR_STRENGTH[1]);
      lcd.createChar(2, CHR_STRENGTH[0]);
      break;
    case 24 ... 33:                          // SIGNAL STRENGTH 3
      lcd.createChar(1, CHR_STRENGTH[2]);
      lcd.createChar(2, CHR_STRENGTH[0]);
      break;
    case 34 ... 43:                          // SIGNAL STRENGTH 4
      lcd.createChar(1, CHR_STRENGTH[2]);
      lcd.createChar(2, CHR_STRENGTH[3]);
      break;
    case 44 ... 53:                          // SIGNAL STRENGTH 5
      lcd.createChar(1, CHR_STRENGTH[2]);
      lcd.createChar(2, CHR_STRENGTH[4]);
      break;
    case 54 ... 63:                          // SIGNAL STRENGTH 6
      lcd.createChar(1, CHR_STRENGTH[2]);
      lcd.createChar(2, CHR_STRENGTH[5]);
      break;
  }

  lcd.setCursor(0,1);                        // MOVE CURSOR TO FIRST COLUMN, SECOND
ROW
  lcd.write(byte(0));                        // PRINT ANTENNA CHARACTER ON LCD
  lcd.write(byte(1));                        // PRINT SIGNAL STRENGTH 1 - 3
  lcd.write(byte(2));                        // PRINT SIGNAL STRENGTH 4 - 6
}
```

Figure XXVIII: "radio.h" Library Code

```
#include <Arduino.h>

#define I2C_SEQ   0x10
#define I2C_INDEX 0x11

#define RADIO_REG_2 0x02
#define RADIO_REG_3 0x03
#define RADIO_REG_4 0x04
#define RADIO_REG_5 0x05
#define RADIO_REG_6 0x06
#define RADIO_REG_7 0x07

#define RADIO_REG_A 0x0A
#define RADIO_REG_B 0x0B
#define RADIO_REG_C 0x0C
#define RADIO_REG_D 0x0D
#define RADIO_REG_E 0x0E
#define RADIO_REG_F 0x0F

// Register 2 bits
#define R2_AUDIO_OUTPUT   0x8000
#define R2_MUTE_DISABLE   0x4000
#define R2_MONO_SELECT    0x2000
#define R2_BASS_BOOST     0x1000
#define R2_CLOCK_CAL      0x0800
#define R2_RCLK_DIRECT    0x0400
#define R2_SEEK_UP        0x0200
#define R2_SEEK_ENABLE    0x0100
#define R2_SEEK_NO_WRAP   0x0080
#define R2_CLOCK_MODE     0x0070
#define R2_CLOCK_MODE_32K 0x0000
#define R2_CLOCK_MODE_12M 0x0010
#define R2_CLOCK_MODE_13M 0x0020
#define R2_CLOCK_MODE_19M 0X0030
#define R2_CLOCK_MODE_24M 0X0050
#define R2_CLOCK_MODE_26M 0x0060
#define R2_CLOCK_MODE_38M 0x0070
#define R2_RDS_ENABLE     0x0008
#define R2_NEW_METHOD     0x0004
#define R2_SOFT_RESET     0x0002
#define R2_ENABLE         0x0001

// Register 3 bits
#define R3_CHANNEL         0xFFC0
#define R3_DIRECT_MODE     0x0020
#define R3_TUNE_ENABLE     0x0010
#define R3_BAND            0x000C
#define R3_BAND_US_EU      0x0000
#define R3_BAND_JP         0x0004
#define R3_BAND_WORLD      0x0008
#define R3_BAND_E_EU       0x000C
#define R3_CHAN_SPACING    0x0003
#define R3_CHAN_SPACE_100K 0x0000
#define R3_CHAN_SPACE_200K 0x0001
#define R3_CHAN_SPACE_50K  0x0002
#define R3_CHAN_SPACE_25K  0x0003

// Register 4 bits
#define R4_SEEK_TUNE_IE     0x4000
#define R4_DE_EMPHASIS_50   0x0800
#define R4_SOFT_MUTE_ENABLE 0x0200
#define R4_AFC_DISABLE      0x0100
#define R4_I2S_ENABLE       0x0040
```

```
#define R4_GPIO3            0x0030
#define R4_GPIO2            0x000C
#define R4_GPIO1            0x0003

// Register 5 bits
#define R5_INTERRUPT_MODE 0x8000
#define R5_SNR_THRESHOLD  0x0800
#define R5_LNA_PORT       0x0080
#define R5_VOLUME         0x000F

#define RA_RDS_READY     0x8000
#define RA_TUNE_COMPLETE 0x4000
#define RA_SEEK_FAIL     0x2000
#define RA_RDS_SYNC      0x1000
#define RA_RDS_BLK_E     0x0800
#define RA_STEREO        0x0400
#define RA_CHANNEL       0x03FF

#define RB_RSSI        0xFE00
#define RB_IS_STATION  0x0100
#define RB_RDS_BLOCK_E 0x0010
#define RB_RDS_ERR     0x000F

#define RDS_GROUP     0xF800
#define RDS_GROUP_A0  0x0000
#define RDS_GROUP_A2  0x2000
#define RDS_GROUP_A4  0x4000
#define RDS_GROUP_A10 0xA000
#define RDS_GROUP_B0  0x0800


class Radio {
      public:
            struct RadioState {
                  bool isEnabled;
                  bool isMuted;
                  bool isSoftMute;
                  bool isMonoForced;
                  bool isBassBoost;
                  byte volume;
                  byte sensitivity;

                  bool isTuning;
                  bool tuningComplete;
                  bool tuningError;
                  bool isTunedToChannel;
                  bool isStereo;
                  float frequency;
                  byte signalStrength;

                  bool hasRdsData;
                  bool hasRdsBlockE;
                  byte rdsBlockErrors;
                  unsigned short rdsBlockA;
                  unsigned short rdsBlockB;
                  unsigned short rdsBlockC;
                  unsigned short rdsBlockD;
                  unsigned short rdsBlockE;

                  bool hasStationName;
                  char stationName[9];
            } radioState;
            RadioState state;

            Radio();
```

```
            bool init();
            void setVolume(byte volume);
            void setMute(bool mute);
            void setSoftMute(bool softMute);
            void setForceMono(bool forceMono);
            void setBassBoost(bool bassBoost);
            void setEnable(bool enable);
            void seekUp();
            void seekDown();
            void tuneTo(float frequency);
            void updateStatus();


      private:
            unsigned short regs[16];
            char rdsStationName[8];

            float getFrequency();
            void resetStation();
            void sendRegister(byte reg);
            void decodeRdsMessage();
};
```

Figure XXIX: "radio.cpp" Library Code

```
#include <Arduino.h>
#include <Wire.h>
#include "radio.h"


Radio::Radio() {
}


bool Radio::init() {
	for (int i = 0; i < 16; i ++) {
		regs[i] = 0x0000;
	}

	state.isEnabled    = true;
	state.isMuted      = false;
	state.isSoftMute   = false;
	state.isMonoForced = false;
	state.isBassBoost  = false;
	state.volume       = 5;
	state.sensitivity  = 0;

	Wire.begin();
	Wire.beginTransmission(I2C_INDEX);
	if (Wire.endTransmission()) return false;

	regs[RADIO_REG_2] = R2_SOFT_RESET | R2_ENABLE;
	regs[RADIO_REG_3] = R3_BAND_US_EU | R3_CHAN_SPACE_100K;
	regs[RADIO_REG_4] = 2048;
	regs[RADIO_REG_5] = R5_INTERRUPT_MODE | R5_SNR_THRESHOLD | R5_LNA_PORT |
state.volume;
	regs[RADIO_REG_6] = 0x0000;
	regs[RADIO_REG_7] = 0x0000;

	for (char i = RADIO_REG_2; i <= RADIO_REG_7; i ++) {
		sendRegister(i);
	}
	regs[RADIO_REG_2] = R2_AUDIO_OUTPUT | R2_MUTE_DISABLE | R2_RDS_ENABLE |
R2_NEW_METHOD | R2_ENABLE;
	sendRegister(RADIO_REG_2);

	regs[RADIO_REG_3] |= R3_TUNE_ENABLE;
	sendRegister(RADIO_REG_3);

	return true;
}


void Radio::setVolume(byte volume) {
	state.volume = volume;
	regs[RADIO_REG_5] = (regs[RADIO_REG_5] & 0xFFF0) | (volume & 0x0F);
	sendRegister(RADIO_REG_5);
}


void Radio::setMute(bool mute) {
	state.isMuted = mute;
	if (mute) {
		regs[RADIO_REG_2] &= (~R2_MUTE_DISABLE);
	} else {
		regs[RADIO_REG_2] |= R2_MUTE_DISABLE;
	}
	sendRegister(RADIO_REG_2);
```

```
}


void Radio::setSoftMute(bool softMute) {
	state.isSoftMute = softMute;
	if (softMute) {
		regs[RADIO_REG_4] |= R4_SOFT_MUTE_ENABLE;
	} else {
		regs[RADIO_REG_4] &= (~R4_SOFT_MUTE_ENABLE);
	}
	sendRegister(RADIO_REG_4);
}


void Radio::setForceMono(bool forceMono) {
	state.isMonoForced = forceMono;
	if (forceMono) {
		regs[RADIO_REG_2] |= R2_MONO_SELECT;
	} else {
		regs[RADIO_REG_2] &= (~R2_MONO_SELECT);
	}
	sendRegister(RADIO_REG_2);
}


void Radio::setBassBoost(bool bassBoost) {
	state.isBassBoost = bassBoost;
	if (bassBoost) {
		regs[RADIO_REG_2] |= R2_BASS_BOOST;
	} else {
		regs[RADIO_REG_2] &= (~R2_BASS_BOOST);
	}
	sendRegister(RADIO_REG_2);
}


void Radio::setEnable(bool enable) {
	state.isEnabled = enable;
	if (enable) {
		regs[RADIO_REG_2] |= R2_ENABLE;
	} else {
		regs[RADIO_REG_2] &= (~R2_ENABLE);
	}
	sendRegister(RADIO_REG_2);
}


void Radio::seekUp() {
	resetStation();
	regs[RADIO_REG_2] |= R2_SEEK_UP | R2_SEEK_ENABLE;
	sendRegister(RADIO_REG_2);
}


void Radio::seekDown() {
	resetStation();
	regs[RADIO_REG_2] &= (~R2_SEEK_UP);
	regs[RADIO_REG_2] |= R2_SEEK_ENABLE;
	sendRegister(RADIO_REG_2);
}


/**
 * Tune to the given frequency in MHz.
 */
```

```
void Radio::tuneTo(float frequency) {
	unsigned short channelSpacing = regs[RADIO_REG_3] & R3_CHAN_SPACING;
	unsigned short band = regs[RADIO_REG_3] & R3_BAND;

	// Subtract frequency base depending on current band.
	switch (band) {
		case R3_BAND_US_EU:
			frequency -= 87;
			break;
		case R3_BAND_JP:
		case R3_BAND_WORLD:
			frequency -= 76;
			break;
		case R3_BAND_E_EU:
			frequency -= 65;
			break;
	}

	// Find channel number depending on current channel spacing.
	unsigned short channel;
	switch (channelSpacing) {
		case R3_CHAN_SPACE_25K:
			channel = frequency / 0.025f;
			break;
		case R3_CHAN_SPACE_50K:
			channel = frequency / 0.05f;
			break;
		case R3_CHAN_SPACE_100K:
			channel = frequency / 0.1f;
			break;
		case R3_CHAN_SPACE_200K:
			channel = frequency / 0.2f;
			break;
	}

	resetStation();
	regs[RADIO_REG_3] &= (~R3_TUNE_ENABLE);
	sendRegister(RADIO_REG_3);
	regs[RADIO_REG_3] |= ((channel << 6) & R3_CHANNEL) | R3_TUNE_ENABLE;
	sendRegister(RADIO_REG_3);
}


void Radio::resetStation() {
	state.isTuning = true;
	state.isTunedToChannel = false;
	state.hasStationName = false;
	for (byte i = 0; i < 8; i ++) {
		state.stationName[i] = ' ';
	}
}


void Radio::updateStatus() {
	Wire.requestFrom(I2C_SEQ, (6 * 2) );
	for (int i = RADIO_REG_A; i <= RADIO_REG_F; i ++) {
		regs[i] = (Wire.read() << 8) + Wire.read();
	}
	Wire.endTransmission();

	// Toggle RDS enable flag when new data is received to wait for new data.
	if (regs[RADIO_REG_A] & RA_RDS_READY) {
		regs[RADIO_REG_2] &= (~R2_RDS_ENABLE);
		sendRegister(RADIO_REG_2);
		regs[RADIO_REG_2] |= R2_RDS_ENABLE;
```

```
		sendRegister(RADIO_REG_2);
	}

	// When tuned disable tuning and stop seeking.
	if (state.isTuning && regs[RADIO_REG_A] & RA_TUNE_COMPLETE) {
		regs[RADIO_REG_3] &= (~R3_TUNE_ENABLE);
		sendRegister(RADIO_REG_3);
		regs[RADIO_REG_2] &= (~R2_SEEK_ENABLE);
		sendRegister(RADIO_REG_2);
	}

	// Transform raw data into radioState.
	state.hasRdsData       = regs[RADIO_REG_A] & RA_RDS_READY;
	state.isTuning         = !(regs[RADIO_REG_A] & RA_TUNE_COMPLETE);
	state.tuningError      = regs[RADIO_REG_A] & RA_SEEK_FAIL;
	state.hasRdsBlockE     = regs[RADIO_REG_A] & RA_RDS_BLK_E;
	state.isStereo         = regs[RADIO_REG_A] & RA_STEREO;
	state.frequency        = getFrequency();
	state.signalStrength   = (regs[RADIO_REG_B] & RB_RSSI) >> 9;
	state.isTunedToChannel = regs[RADIO_REG_B] & RB_IS_STATION;
	if (state.hasRdsData) {
		state.rdsBlockErrors   = regs[RADIO_REG_B] & RB_RDS_ERR;
		if (!(regs[RADIO_REG_B] & RB_RDS_BLOCK_E)) {
			state.rdsBlockA = regs[RADIO_REG_C];
			state.rdsBlockB = regs[RADIO_REG_D];
			state.rdsBlockC = regs[RADIO_REG_E];
			state.rdsBlockD = regs[RADIO_REG_F];

			// if (state.rdsBlockErrors == 0) {
			decodeRdsMessage();
			// }
		} else {
			state.rdsBlockE = regs[RADIO_REG_C];
		}
	}
}


float Radio::getFrequency() {
	unsigned short channelSpacing = regs[RADIO_REG_3] & R3_CHAN_SPACING;
	unsigned short band = regs[RADIO_REG_3] & R3_BAND;
	unsigned short channel = regs[RADIO_REG_A] & RA_CHANNEL;

	// Subtract frequency base depending on current band.
	float freq;
	switch (band) {
		case R3_BAND_US_EU:
			freq = 87.0f;
			break;
		case R3_BAND_JP:
		case R3_BAND_WORLD:
			freq = 76.0f;
			break;
		case R3_BAND_E_EU:
			freq = 65.0f;
			break;
	}

		// Add frequency offset depending on channel spacing.
	switch (channelSpacing) {
		case R3_CHAN_SPACE_25K:
			return freq + (channel * 0.025f);
		case R3_CHAN_SPACE_50K:
			return freq + (channel * 0.05f);
		case R3_CHAN_SPACE_100K:
```

```
                return freq + (channel * 0.1f);
            case R3_CHAN_SPACE_200K:
                return freq + (channel * 0.2f);
            default:
                return 0;
        }
}


void Radio::decodeRdsMessage() {
      switch (state.rdsBlockB & RDS_GROUP) {
            case RDS_GROUP_A0:
            case RDS_GROUP_B0:
                byte offset = (state.rdsBlockB & 0x03) << 1;
                char c1 = (char)(state.rdsBlockD >> 8);
                char c2 = (char)(state.rdsBlockD & 0xFF);

                // Copy station name byte only if received it twice in a row...
                if (rdsStationName[offset] == c1) {
                    state.stationName[offset] = c1;
                    state.hasStationName = true;
                } else {
                    rdsStationName[offset] = c1;
                }

                if (rdsStationName[offset + 1] == c2) {
                    state.stationName[offset + 1] = c2;
                    state.hasStationName = true;
                } else {
                    rdsStationName[offset + 1] = c2;
                }

                break;
      }
}


void Radio::sendRegister(byte reg) {
      Wire.beginTransmission(I2C_INDEX);
      Wire.write(reg);
      Wire.write(regs[reg] >> 8);
      Wire.write(regs[reg] & 0xFF);
      Wire.endTransmission();
}
```