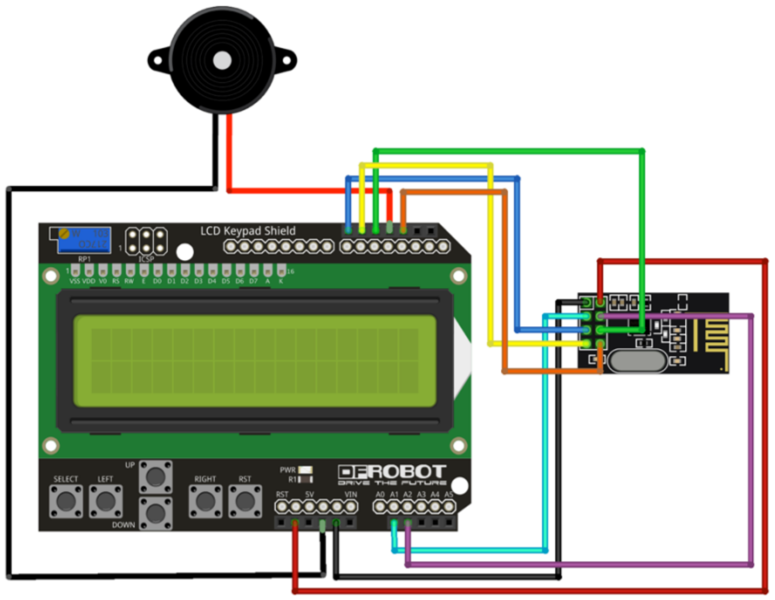
# Project: Morse Beeper

## Overview

A **beeper** is a wireless communication device that receives and sends messages to others using its internal transceiver. In this project, we design and prototype a device that resembles a beeper and is able to store contacts, send, and receive **morse code** messages.

You're building a **wireless Morse-code pager** using:

- **Arduino LCD Keypad Shield** — for user interface
- **NRF24L01+ transceiver** — for sending/receiving messages
- **Piezo Buzzer** — for beeps & notifications
- **EEPROM** — for saving data (contacts, messages, device info)



## In the process you will:

- Interface with a **Liquid Crystal Display** (LCD) component.
- Use a voltage divider as an **analog input** to detect button presses.
- Interface with an **NRF2401L+** module via SPI to communicate with other modules.
- Interface with a piezo **buzzer** to notify user of events.
- Interface with the Arduino's **EEPROM** to store long-term data that can be retrieved after a power cycle.
- Use the **Watchdog Timer**'s jitter to generate random byte sequences.
- Implement a beeper as a **state machine**.

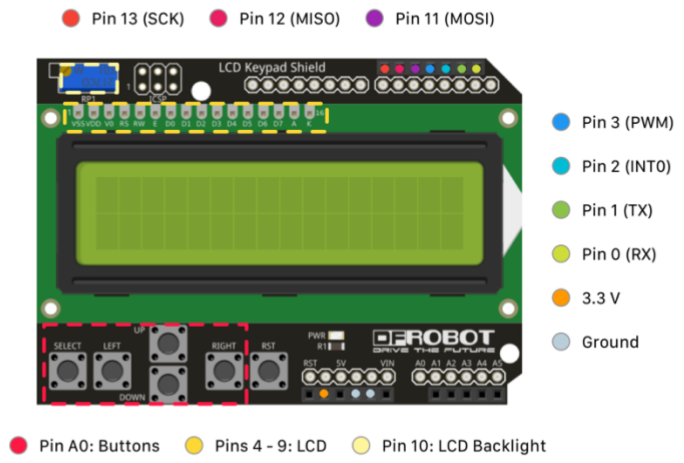
- Apply the concept of abstractions in C++ by defining **classes** and instantiating **objects** from those classes to promote re-usability and increased efficiency.
- Use **inheritance** to extend Arduino's classes.
- Use **pointers** to handle dynamic memory allocation.

## Main Components and What They Do

| Component | Purpose |
|---|---|
| Entropy Class | Generates a random 40-bit UUID for your device. |
| NRF24 Class | Manages the radio communication via SPI. You configure its pipes and addresses using the UUID. |
| LCD Keypad Class | Extends LiquidCrystal to handle analog buttons (Up/Down/Left/Right/Select) and debounce input. |
| Memory Class | Custom EEPROM interface to store initialization flags, contacts, messages. You must write this yourself. |
| Contact Class | Holds a name + UUID (15 bytes total). |
| Message Class | Holds sender UUID, receiver UUID, payload (Morse bits), and payload length (13 bytes total). |

## LCD Keypad Class

The LCD library, LiquidCrystal, allows you to control displays that are compatible with the **Hitachi HD 44780** driver. The LCD Keypad Shield provided looks like this:



The following table describes which pins are used by the shield.

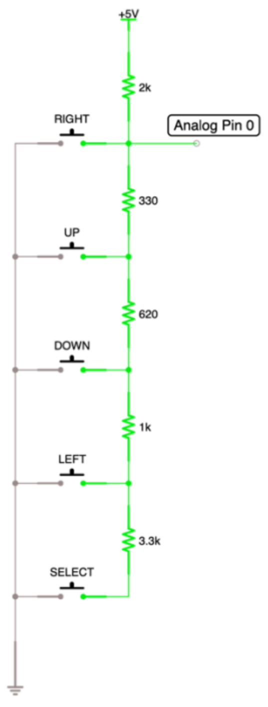
| Pin | Description | Pin | Description |
|---|---|---|---|
| A0 | Buttons | 7 | LCD DB7 |

| Pin | Description | Pin | Description |
|---|---|---|---|
| 4 | LCD DB4 | 8 | LCD RS |
| 5 | LCD DB5 | 9 | LCD Enable |
| 6 | LCD DB6 | 10 | Backlight Control |

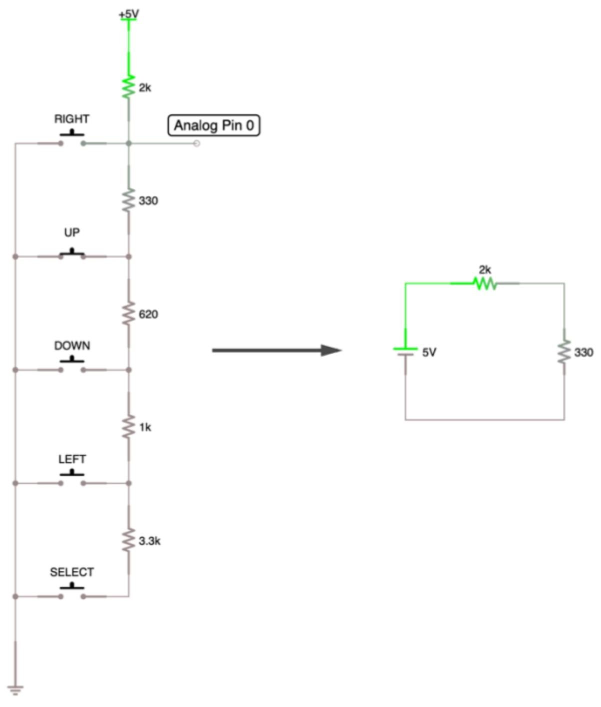
From the picture above you can see that the shield is equipped with the following buttons:

- Select
- Left
- Up
- Down
- Right
- Reset

These buttons (with the exception of the reset button) are wired to pin A0 using a **voltage divider**. The value at pin A0 depends on which button was pressed. A portion of the schematic is presented below.



Schematic



Button Up Pressed

The **resistances** used in your shield might vary depending on the manufacturer. You will need this information when calculating the expected value for a button press. For instance, when button UP is pressed, the voltage at pin A0 can be found using **Ohm's law**.

$$V_{out} = V_{in} \times \frac{R_2}{R_1 + R_2}$$

$$V_{pin0} = 5 \times \frac{330}{2330}$$

$$V_{pin0} = 0.7081V$$

Alternatively, you can obtain this information with a multimeter or by reading the value at the pin when a button is pressed. The complete schematic can be found here.

The Arduino reads the value at the pin and provides a number ranging from 0 1023 corresponding to the input voltage. Since we calculated the voltage at pin A0 to be 0.7081 V we can proceed to map this value to reflect the Arduino's **10-bit ADC** resolution. We know that 5V is represented by the decimal value 1023 and that 0V is represented by 0. We can now map the value accordingly using the **Rule of Three**. We find out that the value at the pin read by the Arduino is around 144. You will use **inheritance** to create an LCD Keypad class which will:

- *Extend the LiquidCrystal class.*
- *Define the following enum to make your code more readable:*

```
typedef enum {LEFT, RIGHT, UP, DOWN, SELECT, NONE} Button;
```

- Implement **debouncing** for the analog input wired at pin A0. Define and implement the function getButtonPress() which returns the button that was pressed. The function prototype is provided below.

```
Button getButtonPress();
```

**Entropy class**

Question 1: What is the *Entropy Class*?

Think of it as a **little random-number generator** that lives inside your Arduino.

Question 2: Why do we need it?

Just like every phone has a **unique phone number**, every “Morse Beeper” needs its own **unique ID** so messages go to the right device.

If two beepers had the same ID, messages could get mixed up like two people sharing the same phone number. So, we give each device a special tag called a **UUID** — a

*Universally Unique Identifier*. It's basically a long random number that's very unlikely to repeat anywhere else.

Question 3: How do we make it random?

The Arduino doesn't have a built-in random number chip, so we use something already inside it: the **Watchdog Timer**. The Watchdog Timer is a tiny circuit that keeps the microcontroller from freezing. It has a natural "jitter" — small, unpredictable timing variations. The **Entropy class** uses that tiny randomness ("noise") to create random bytes. These random bytes are then combined into a **5-byte (40-bit) number**, and that becomes the device's **UUID** — its personal identity number.

Question 4:  Why is this important?

Each Morse Beeper gets its *own identity*. When two devices talk over radio, they know *who* sent the message. The chance of two devices getting the same UUID is *so small* that we can ignore it.

**RF24 Class**

Question 1: What is the nRF24L01+?

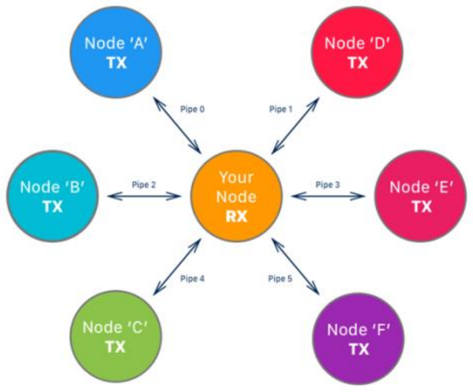
It's a **tiny wireless radio module** that lets your Arduino **send and receive data** — kind of like a walkie-talkie for microcontrollers.

- It works on the **2.4 GHz frequency band** (the same band as Wi-Fi and Bluetooth).
- It's very **low power**, so it's perfect for small, embedded devices.
- It can transfer data at up to **2 megabits per second (2 Mbps)**.

You can download the datasheet here.

Question 2: How does it talk to the Arduino?

The Arduino communicates with this module using **SPI (Serial Peripheral Interface)** — a standard way to send fast data between chips. In SPI, there's one **master** (the Arduino) and one **slave** (the radio). The master tells the slave when to send or receive information.



Question 3: What are “Data Pipes”?

Think of the radio as a **post office with multiple mailboxes**.

- Each mailbox (pipe) has its own **address** (like a street address).
- When you send a message, you put it in one pipe; the receiver listens on that same pipe.
- The nRF24 can **listen to up to 6 pipes at once** — so one device can talk to several others.

Each pipe address is **40 bits long (5 bytes)** — this is like a *5-digit mailbox ID*.

| Name | Description | Connected To |
|---|---|---|
| CE | Chip Enable (RX / TX) | A1 |
| CSN | SPI Chip Select | A2 |
| MOSI | SPI Slave Data Input | 11 or ICSP-4 |
| MISO | SPI Slave Data Output | 12 or ICSP-1 |
| SCK | SPI Clock | 13 or ICSP-3 |
| IRQ | Maskable Interrupt | 2 |
| VCC | Power (1.9V - 3.6V) | 3.3 V |
| GND | Ground (0V) | GND |

Question 4: Why do we need a 40-bit UUID (Unique ID)?

If two radios in the lab use the **same pipe address**, they'll mix up messages — like two houses having the same postal address.

So, we generate a **unique 40-bit number (UUID)** using the **Entropy class** (random generator).
This becomes *your radio's address*, making it almost impossible for two boards to collide.

We will use the NR24 library to control the radio. Documentation for the library can be found here. The following table describes how to wire your NRF24L01+ module. **Please note that the module cannot operate at 5V.**

## How to Progress Step-by-Step

## Phase 1: Hardware Bring-Up

1. **LCD Shield**
   - Test LCD with "Hello World".
   - Read analog values from A0 for each button.
   - Map ADC values to enums: {LEFT, RIGHT, UP, DOWN, SELECT, NONE}.
   - Implement getButtonPress() with debouncing.
2. **Buzzer**
   - Use tone(pin, freq, duration) to generate beeps for notifications.
   - Test short beeps for dot/dash sounds.
3. **NRF24L01+**
   - Wire according to the provided table (CE→A1, CSN→A2, IRQ→2, MOSI→11, MISO→12, SCK→13).
   - Test with RF24 example "Getting Started" sketch to confirm send/receive.
4. **Entropy**
   - Integrate Entropy library.
   - Generate a 5-byte UUID and print to Serial.

Below is the "Getting Started" for checking the hardware

```
#include <Arduino.h>
#include <LiquidCrystal.h>
#include <RF24.h>
#include <Entropy.h>

/* ---------- hardware pins ---------- */
const int BUZZER_PIN = 3;
const int CE_PIN  = A1;
const int CSN_PIN = A2;

/* ---------- objects ---------- */
LiquidCrystal lcd(8, 9, 4, 5, 6, 7);   // LCD Shield pins
RF24 radio(CE_PIN, CSN_PIN);          // Radio object
byte uuid[5];                          // 5-byte unique ID

/* ---------- helper ---------- */
void beep(int f, int d) { tone(BUZZER_PIN, f, d); }

/* ---------- setup ---------- */
void setup() {
  pinMode(BUZZER_PIN, OUTPUT);
  lcd.begin(16, 2);

  //  Say hello on LCD
  lcd.print("Hello!");
  lcd.setCursor(0,1);
  lcd.print("Let's Begin :)");
  beep(1000,200);
  delay(1500);
  lcd.clear();

  // Make a random 5-byte UUID
  Entropy.initialize();
  for (int i=0;i<5;i++) uuid[i] = Entropy.randomByte();

  Serial.begin(115200);
  Serial.print("My UUID: 0x");
  for (int i=0;i<5;i++){ if(uuid[i]<0x10)
Serial.print('0');
Serial.print(uuid[i],HEX); }
  Serial.println();
  lcd.print("UUID:");
  lcd.setCursor(0,1);
  for (int i=0;i<5;i++){
if(uuid[i]<0x10)
lcd.print('0');
lcd.print(uuid[i],HEX); }
  delay(2000);
  lcd.clear();

  //  Test Radio power-up
  lcd.print("Radio check...");
  radio.begin();
  radio.setPALevel(RF24_PA_LOW);
  lcd.setCursor(0,1);
  lcd.print("OK!");
  beep(1200,150);
  delay(1500);
  lcd.clear();

  // Button & buzzer play
  lcd.print("Press buttons!");
  lcd.setCursor(0,1);
  lcd.print("A0 read test");
}

/* ---------- loop ---------- */
void loop() {
  int v = analogRead(A0);   // read button ladder
  lcd.setCursor(0,1);

  if (v < 50)      { lcd.print("RIGHT "); beep(1000,80); }
  else if (v <250) { lcd.print("UP    "); beep(1200,80); }
  else if (v <450) { lcd.print("DOWN  "); beep(800,80); }
  else if (v <650) { lcd.print("LEFT  "); beep(600,80); }
  else if (v <900) { lcd.print("SELECT"); beep(1500,80); }
  else             { lcd.print("NONE  "); }

  delay(200);
}
```

**After this, implement the *lcd.h* and buzzer.h according to the project requirements. (template inside the zip folder).**

## Phase 2: Software Foundations

### I. Contact class

A Contact object stores information on a contact's **name** and **UUID**. The **UUID** is 40-bit long, and the contact's name is up to ten characters long. This gives a Contact object a size of **15 bytes**.

1. Contact's UUID: The contact's 40-bit UUID.
2. Contact's name: The contact's name. This can hold up to ten characters.

Header file template is attached, implement the code for source file.

Below is the explanation of each function.

- Contact(): Default constructor for the Contact class. Creates an empty contact with an empty name and empty UUID.
- Contact(unsigned char* givenUUID, char const* givenName): Parametrized constructor for the Contact class. Takes a given 40-bit UUID and a C-style string as parameters to initialize the object's data members.
- Contact(unsigned char* givenUUID, char givenName): Parametrized constructor for the Contact class. Takes a given 40-bit UUID and a character as parameters to initialize the object's data members.
- void setUUID(unsigned char* givenUUID): Saves the given UUID as the contact's UUID.
- void setName(char const* givenName): Saves the given C-style string as the contact's name. Enforces that the contact's name is up to ten characters long by truncating the excess characters.
- void setName(char givenName): Saves the given character as the contact's name.
- unsigned char* getUUID(): Returns the contact's 40-bit UUID.
- char* getName(): Returns the contact's name as a C-style string.

## II. Message Class

A Message object stores the **UUID** of the sender and the receiver, alongside with the payload and its length. This gives a Message object the size of **13 bytes**.

1. Sender's UUID: The sender's 40-bit UUID.
2. Receivers's UUID: The receiver's 40-bit UUID. Used to verify that message was received by the intended party.
3. Payload: The payload consists of **morse code** messages that may vary in size. Their size cannot exceed 16 characters (since that's the maximum amount we can display on the LCD screen). Morse code uses the '**.**' and the '-' symbols and we will represent these with **zeroes** and **ones** respectively. This allows us to represent each character with a bit instead of a byte. We need 2 bytes to represent up to 16 characters.
4. Payload Length: Since a message can vary in **length**, the payload length tells us how many bits are valid to be interpreted as part of the message. We use 1 byte to store the payload's length.

Header file template is attached, implement the code for source file. Below is the explanation of each function.

1. Message(): Default constructor for the Message class. Creates an empty message.
2. Message(unsigned char* from, unsigned char* to, unsigned short payload, unsigned char length): Parametrized constructor for the Message class. Takes the sender and receiver's UUIDs, the payload, and its length.
3. Message(unsigned char* from, unsigned char* to, char const* message): Parametrized constructor for the Message class. Takes the sender and receiver's UUIDs. Additionally, it takes a message in a C-style string. The payload length is calculated after the conversion from a C-style string to the payload bits.
4. void setLength(unsigned char length): Sets the length of the payload for the Message object.
5. void setTo(unsigned char* to): Sets the receiver's UUID for the Message object.
6. void setFrom(unsigned char* from): Sets the sender's UUID for the Message object.
7. void setPayload(unsigned short payload): Sets the payload for the Message object.
8. unsigned char getLength(): Returns the payload's length.
9. unsigned char* getTo(): Returns the receiver's UUID.
10. unsigned char* getFrom(): Returns the sender's UUID.
11. unsigned short getPayload(): Returns the payload.
12. char* getPayloadString(): Returns the decoded version of the payload in a C-style string.
13. unsigned short stringToPayload(char const* message): Encodes up to a 16 character Morse code string into a 16 bit buffer. Returns the result of this encoding.
14. char* payloadToString(unsigned short payload, unsigned char length): Decodes a payload into a C-style string. Uses the payload's length to determine the size of the resulting string. Returns the decoded string.

## III. Memory class

In embedded systems where no disk drive exists, non-volatile memory is typically a variant of Read-Only Memory (ROM). The **ATMega328P** follows a Harvard architecture, where program code and data are separated. Program code is stored in Flash. Data, on the other hand, can be found in both **SRAM** and **EEPROM**. The microcontroller on the Arduino Uno board has 1KiB of **EEPROM** memory. You will be using the **EEPROM** to store configuration information, contacts, and messages. The table below describes the memory map implemented for this system.

| Address | Value | Purpose |
|---|---|---|
| 000 - 002 | 0xC0FFEE | Initialization Flag |

| Address | Value | Purpose |
|---|---|---|
| 003 - 017 | Contact Object | Node's Contact: UUID and Name |
| 018 - 019 | 0xFACE | Contact List Flag |
| 020 | Counter | Number of Contacts |
| 021 - 035 | Contact Object | Contact #1: UUID and Name |
| 036 - 050 | Contact Object | Contact #2: UUID and Name |
| 051 - 065 | Contact Object | Contact #3: UUID and Name |
| 066 - 080 | Contact Object | Contact #4: UUID and Name |
| 081 - 095 | Contact Object | Contact #5: UUID and Name |
| 096 - 110 | Contact Object | Contact #6: UUID and Name |
| 111 - 125 | Contact Object | Contact #7: UUID and Name |
| 126 - 140 | Contact Object | Contact #8: UUID and Name |
| 141 - 155 | Contact Object | Contact #9: UUID and Name |
| 156 - 170 | Contact Object | Contact #10: UUID and Name |
| 171 - 172 | 0xCA11 | Message List Flag |
| 173 | Counter | Number of Messages |
| 174 - 186 | Message Object | Message #1 |
| 187 - 199 | Message Object | Message #2 |
| 200 - 212 | Message Object | Message #3 |
| 213 - 225 | Message Object | Message #4 |
| 226 - 238 | Message Object | Message #5 |
| 239 - 251 | Message Object | Message #6 |
| 252 - 264 | Message Object | Message #7 |
| 265 - 277 | Message Object | Message #8 |
| 278 - 290 | Message Object | Message #9 |
| 291 - 303 | Message Object | Message #10 |
| 304 - 316 | Message Object | Message #11 |

| Address | Value | Purpose |
|---|---|---|
| 317 - 329 | Message Object | Message #12 |
| 330 - 342 | Message Object | Message #13 |
| 343 - 355 | Message Object | Message #14 |
| 356 - 368 | Message Object | Message #15 |
| 369 - 381 | Message Object | Message #16 |
| 382 - 394 | Message Object | Message #17 |
| 395 - 407 | Message Object | Message #18 |
| 408 - 420 | Message Object | Message #19 |
| 421 - 433 | Message Object | Message #20 |
| 434 | Offset | Next available spot @ <Base + Offset> |

The memory map consists of the following sections:

1. Flags: There are three flags in the memory map that get verified at every boot for integrity and schema.
   - Initialization Flag: Consists of three bytes (000 - 002) that spell 0xC0FFEE. These bytes are set during the device's setup stage.
   - Contact List Flag: Consists of two bytes (018 - 019) that spell 0xFACE. These bytes are set during the device's setup stage and mark the beginning of the contact list related entries in the EEPROM.
   - Message List Flag: Consists of two bytes (171 - 172) that spell 0xCA11. These bytes are set during the device's setup stage and mark the beginning of the message list related entries in the EEPROM.
2. Counters: There are two counter entries in the EEPROM.
   - Contact Counter: Keeps track the number of contacts stored in the EEPROM. May hold values from 0 to 10.
   - Message Counter: Keeps track the number of messages stored in the EEPROM. May hold values from 0 to 20.
3. Offsets: There is one offset entry in the EEPROM. According to the memory table above, the base address for the messages (or the address for the first message) is 174. By adding an offset to this base address, we can obtain the position of any message element relative to the first one. The offset entry can store up to 8-bits of information allowing you to traverse from location 174 all the way to location 429. This is useful to point to the next location available for saving a message.

4. Contact Objects: The EEPROM stores up to 10 contact objects. These objects are 15 bytes long and contain the contact's name and radio's UUID. The object located at address 003 contains information regarding the node and is set during setup time. Since we only have room for ten contacts, additional contacts must not be allowed. See the Contact class for more information.
5. Message Objects: The EEPROM stores up to 20 message objects. These objects are 13 bytes long and contain information on the sender, the receiver, the payload, and the payload's length. Since we only have room for twenty messages in the given design, we may need to reuse EEPROM locations if we receive more than twenty messages. See the Message class for more information.

You will be implementing the Memory class that handles writing and reading data to the **EEPROM**.

Header file template is attached, implement the code for source file. Below is the explanation of each function.

- Message(): Default constructor for the Message class. Creates an empty message.
- Message(unsigned char* from, unsigned char* to, unsigned short payload, unsigned char length): Parametrized constructor for the Message class. Takes the sender and receiver's UUIDs, the payload, and its length.
- Message(unsigned char* from, unsigned char* to, char const* message): Parametrized constructor for the Message class. Takes the sender and receiver's UUIDs. Additionally, it takes a message in a C-style string. The payload length is calculated after the conversion from a C-style string to the payload bits.
- void setLength(unsigned char length): Sets the length of the payload for the Message object.
- void setTo(unsigned char* to): Sets the receiver's UUID for the Message object.
- void setFrom(unsigned char* from): Sets the sender's UUID for the Message object.
- void setPayload(unsigned short payload): Sets the payload for the Message object.
- unsigned char getLength(): Returns the payload's length.
- unsigned char* getTo(): Returns the receiver's UUID.
- unsigned char* getFrom(): Returns the sender's UUID.
- unsigned short getPayload(): Returns the payload.
- char* getPayloadString(): Returns the decoded version of the payload in a C-style string.

- unsigned short stringToPayload(char const* message): Encodes up to a 16 character Morse code string into a 16 bit buffer. Returns the result of this encoding.
- char* payloadToString(unsigned short payload, unsigned char length): Decodes a payload into a C-style string. Uses the payload's length to determine the size of the resulting string. Returns the decoded string.

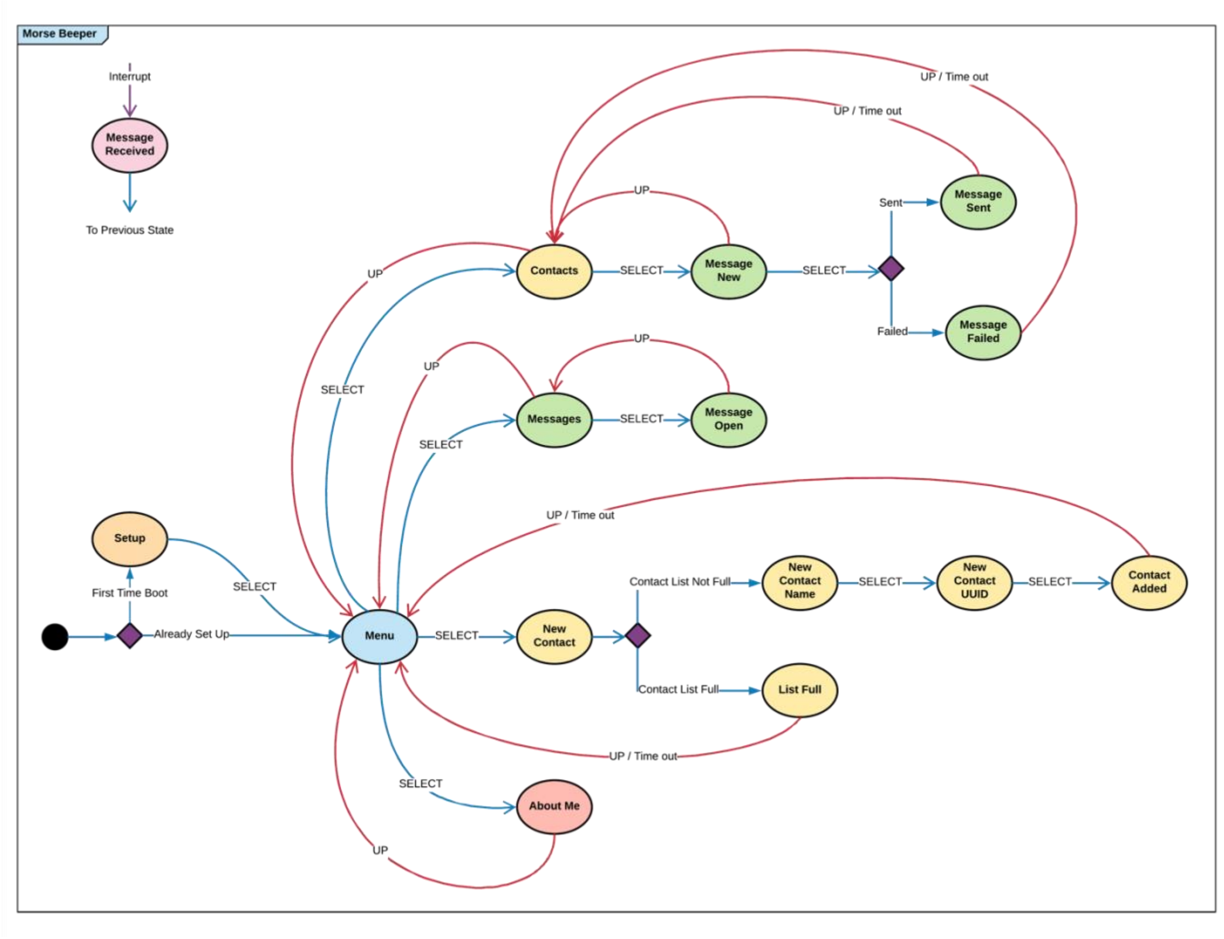
## IV. State Machine Implementation for User Interface

The following section depicts the user interface that must be implemented for the beeper as a **state machine** based on its behavior. Take a look at the diagram below.

The following states can be identified:

- **Setup**: First time boot. User enters his name and a UUID gets generated. User information is stored in the EEPROM. Subsequent boots skip this state and start in the Menu state.
  - **SELECT**: Saves the user's name and goes to the main menu.
  - **LEFT**: Erases the last character and moves cursor to the left.
  - **RIGHT**: Confirms character moves cursor to right.
  - **UP**: Scrolls letter
  - **DOWN**: Scrolls letter
- **Menu**: Displays the menu options. There are blinking arrows on the ends of the second row. These arrows indicate that the user can press the Left or Right buttons to scroll to the next option. The options wrap around once all have been displayed. The arrows complete a blink cycle in a second.
  - **SELECT**: Goes to the selected option. The following is a list of valid menu options.
    - **Contacts**: Goes to Contacts.
    - **Messages**: Goes to Messages.
    - **N. Contact**: Goes to New Contact.
    - **About Me**: Goes to About Me.
  - **LEFT**: Scrolls options.
  - **RIGHT**: Scrolls options.
  - **UP**: None
  - **DOWN**: None
- **Contacts**: Displays the contacts stored in the device. There are blinking arrows on the ends of the second row. These arrows indicate that the user can press the Left or Right buttons to scroll to the next option. The options wrap around once all have been displayed. The arrows complete a blink cycle in a second.
  - **SELECT**: Compose message for selected contact.
  - **LEFT**: Scrolls contact options.
  - **RIGHT**: Scrolls contact options.
  - **UP**: Goes back to the previous screen.
  - **DOWN**: None

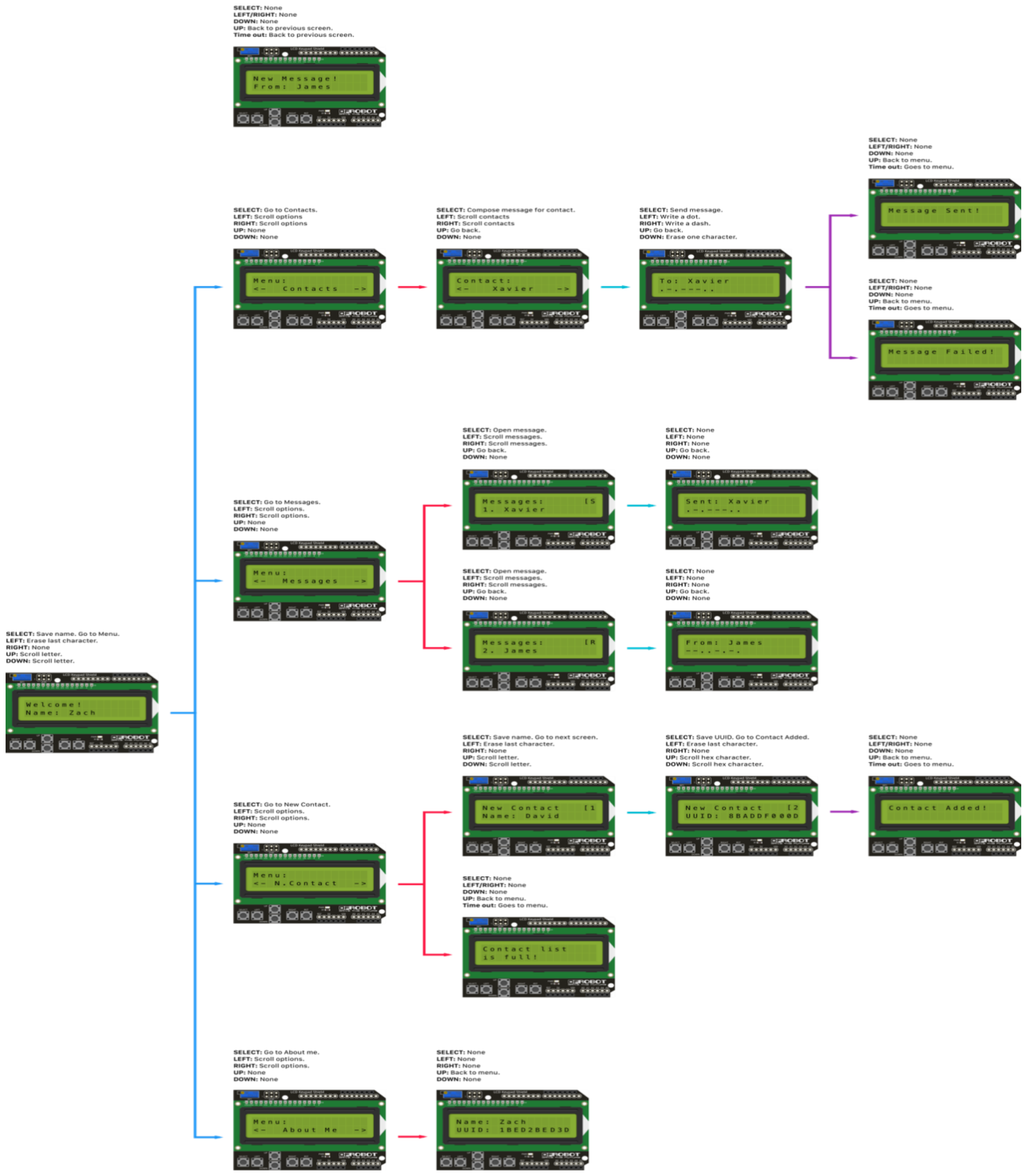
- **Messages**: Displays the messages stored in the device. A list of sent and received messages. A marker on the top right of the screen determines whether the message was sent or received.
  - **SELECT**: Open the selected option.
  - **LEFT**: Scroll messages.
  - **RIGHT**: Scroll messages.
  - **UP**: Goes back to the previous screen.
  - **DOWN**: None



- **New Contact**: Validates whether or not there is space for a new contact. If there is, it transitions to the New Contact Name state. Otherwise, it transitions to the List Full state.
- **New Contact Name**: Screen for new contact name input. A marker on the top right of the screen determines whether this is the first or second screen in the process of adding a contact. The first screen consists of inputting the new contact's name while the second screen consists of inputting the new contact's UUID.
  - **SELECT**: Saves name and goes to the New Contact UUID screen.
  - **LEFT**: Erases the last character and moves cursor to the left.

    - **RIGHT**: Confirms character and moves cursor to the right.
    - **UP**: Scrolls letter.
    - **DOWN**: Scrolls letter.
- **New Contact UUID**: Screen for new contact name input. A marker on the top right of the screen determines that this is the second screen in the process of adding a new contact.
    - **SELECT**: Saves UUID and goes to the New Contact Added screen. Saves contact to the EEPROM.
    - **LEFT**: Erases the last character and moves cursor to the left.
    - **RIGHT**: Confirms character and moves cursor to the right.
    - **UP**: Scrolls letter.
    - **DOWN**: Scrolls letter.
- **Contact Added**: Informative screen that let's the user know that the contact was successfully added. Times out in two seconds returning back to the main menu.
    - **SELECT**: None
    - **LEFT**: None
    - **RIGHT**: None
    - **UP**: Goes back to main menu.
    - **DOWN**: None
    - **Time out**: Goes back to main menu in two seconds. The delay should be non-blocking.
- **List Full**: Informative screen that let's the user know that there is no space for a new contact. Times out in two seconds returning back to the main menu.
    - **SELECT**: None
    - **LEFT**: None
    - **RIGHT**: None
    - **UP**: Goes back to main menu.
    - **DOWN**: None
    - **Time out**: Goes back to main menu in two seconds. The delay should be non-blocking.
- **About Me**: Shows the user's name and UUID.
    - **SELECT**: None
    - **LEFT**: None
    - **RIGHT**: None
    - **UP**: Goes back to main menu.
    - **DOWN**: None
- **Message New**: Displays the username to who we are sending the message to. It also allows constructing a morse string to be sent.
    - **SELECT**: Attempts to send the message.
    - **LEFT**: Write a dot.
    - **RIGHT**: Write a dash.
    - **UP**: Goes back to the previous screen.
    - **DOWN**: Erases one character.

- **Message Sent**: Informative screen that lets the user know that the message was sent successfully. Play a tone through the buzzer indicating that a message was sent. Times out in two seconds returning back to the main menu.
    - **SELECT**: None
    - **LEFT**: None
    - **RIGHT**: None
    - **UP**: Goes back to main menu
    - **DOWN**: None
    - **Time out**: Goes back to main menu in two seconds. The delay should be non-blocking.
- **Message Failed**: Informative screen that let's the user know that the message could not be sent. Play a tone through the buzzer indicating that sending the failed. Times out in two seconds returning back to the main menu.
    - **SELECT**: None
    - **LEFT**: None
    - **RIGHT**: None
    - **UP**: Goes back to main menu
    - **DOWN**: None
    - **Time out**: Goes back to main menu in two seconds. The delay should be non-blocking.
- **Message Open**: Displays a message that has been saved in the device. Displays whether the message was sent or received, the user, and the message.
    - **SELECT**: None
    - **LEFT**: None
    - **RIGHT**: None
    - **UP**: Goes back to the previous screen.
    - **DOWN**: None
- **Message Received**: Informative screen that let's the user know that a new message has been received. Times out in two seconds returning back to the previous screen. Play a tone through the buzzer indicating that a message was received.
    - **SELECT**: None
    - **LEFT**: None
    - **RIGHT**: None
    - **UP**: Goes back to the previous screen.
    - **DOWN**: None
    - **Time out**: Goes back to the previous screen in two seconds. The delay should be non-blocking.



**SELECT:** Save name. Go to Menu.
**LEFT:** Erase last character.
**RIGHT:** None
**UP:** Scroll letter.
**DOWN:** Scroll letter.

**SELECT:** None
**LEFT/RIGHT:** None
**DOWN:** None
**UP:** Back to previous screen.
**Time out:** Back to previous screen.

**SELECT:** Go to Contacts.
**LEFT:** Scroll options
**RIGHT:** Scroll options
**UP:** None
**DOWN:** None

**SELECT:** Compose message for contact.
**LEFT:** Scroll contacts
**RIGHT:** Scroll contacts
**UP:** Go back.
**DOWN:** None

**SELECT:** Send message.
**LEFT:** Write a dot.
**RIGHT:** Write a dash.
**UP:** Go back.
**DOWN:** Erase one character.

**SELECT:** None
**LEFT/RIGHT:** None
**DOWN:** None
**UP:** Back to menu.
**Time out:** Goes to menu.

**SELECT:** None
**LEFT/RIGHT:** None
**DOWN:** None
**UP:** Back to menu.
**Time out:** Goes to menu.

**SELECT:** Go to Messages.
**LEFT:** Scroll options.
**RIGHT:** Scroll options.
**UP:** None
**DOWN:** None

**SELECT:** Open message.
**LEFT:** Scroll messages.
**RIGHT:** Scroll messages.
**UP:** Go back.
**DOWN:** None

**SELECT:** None
**LEFT:** None
**RIGHT:** None
**UP:** Go back.
**DOWN:** None

**SELECT:** Open message.
**LEFT:** Scroll messages.
**RIGHT:** Scroll messages.
**UP:** Go back.
**DOWN:** None

**SELECT:** None
**LEFT:** None
**RIGHT:** None
**UP:** Go back.
**DOWN:** None

**SELECT:** Go to New Contact.
**LEFT:** Scroll options.
**RIGHT:** Scroll options.
**UP:** None
**DOWN:** None

**SELECT:** Save name. Go to next screen.
**LEFT:** Erase last character.
**RIGHT:** None
**UP:** Scroll letter.
**DOWN:** Scroll letter.

**SELECT:** Save UUID. Go to Contact Added.
**LEFT:** Erase last character.
**RIGHT:** None
**UP:** Scroll hex character.
**DOWN:** Scroll hex character.

**SELECT:** None
**LEFT/RIGHT:** None
**DOWN:** None
**UP:** Back to menu.
**Time out:** Goes to menu.

**SELECT:** None
**LEFT/RIGHT:** None
**DOWN:** None
**UP:** Back to menu.
**Time out:** Goes to menu.

**SELECT:** Go to About me.
**LEFT:** Scroll options.
**RIGHT:** Scroll options.
**UP:** None
**DOWN:** None

**SELECT:** None
**LEFT:** None
**RIGHT:** None
**UP:** Back to menu.
**DOWN:** None

**Marks Distribution:**

| Part/Functionality | Marks |
|---|---|
| Buzzer.h | 5 |
| Contact .h and .cpp | 10 |
| Message .h and .cpp | 10 |
| Memory .h and .cpp | 15 |
| Eeprom.h | 10 |
| Lcd.h | 10 |
| Graphical User Interface/State Machine Implementation | 20 |
| Explanation/Viva | 10 |
| Progress Evaluation (November 19,2025) | 5 |
| Working Project | 5 |

**Note:**

1. Form a group of 4-5 students. It is strongly recommended to complete this project in a group.
2. Collect the LCD shield, radio module, and buzzer from the TA.
3. You must demonstrate completed Phase 1 (buzzer.h, lcd.h, Eeprom.h) and Phase 2 (Contact and Message classes) to your TA by **November 19, 2025,** for progress evaluation.
4. Submit **one** zip file per group on Canvas.
2. After completing the project, test your code within your group to verify both transmission and reception. Free to add more function if required as per your implementation way.
3. Make sure, the interface is separated from the implementation by creating .h and .cpp files for each functionality.