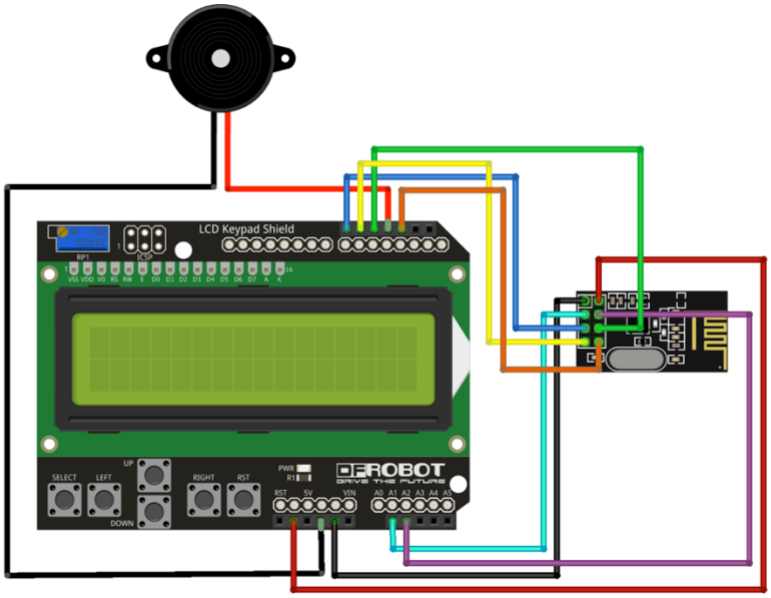
Project: Morse Beeper

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. This project involves interfacing with an **LCD** screen, a **radio** module, the **EEPROM**, and a **buzzer**. We also design and implement an easy to navigate user interface in a constrained 16x2 screen while managing external **interrupts** and handling data **storage**. In the end, all devices that follow this guideline should be able to communicate as described in this document.



What you will need:

- Arduino LCD Keypad Shield
- Piezo Buzzer
- NRF24L01+ module

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.
- Use **interrupts** to process incoming messages.

- 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.

Entropy Class

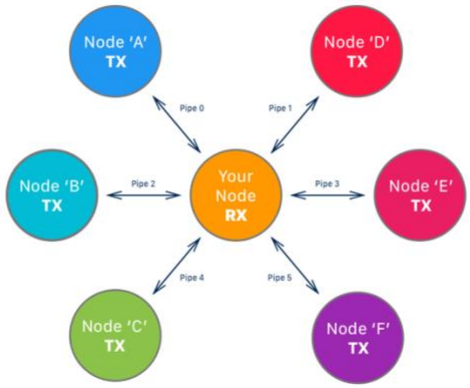
If we compare our beeper to a cell phone, you'll agree that we need to assign it a phone number. Imagine that your phone number is 321-123-4567. If you were not guaranteed that this number belonged **only** to you, then someone else (who shares your number) could also be getting your calls and texts.

We use the term **universally unique identifier (UUID)** to refer to a number that is used to uniquely identify some entity. The probability that a **UUID** will be duplicated is not zero, but it is low enough to be considered negligible. We are going to use the Entropy class to generate a sequence of random bytes using the Watchdog Timer's natural jitter. These bytes will become our unique identifier. You can find the class here.

NR24 Class

The nRF24L01+ is a single chip radio transceiver that operates in the 2.4 - 2.5 GHz band. It features ultra-low power consumption and speeds up to 2 Mbps. We need to use a microcontroller to configure this radio through a **Serial Peripheral Interface (SPI)**. You can download the datasheet here.

The nRF24L01+ implements the concept of data pipes. **Pipes** are logical channels in the physical RF **channel**. Each pipe is assigned its own physical address for subsequent write or read operations. Each address is 40-bit long. This radio is able to write data to one pipe or to listen for data from up to six pipes.



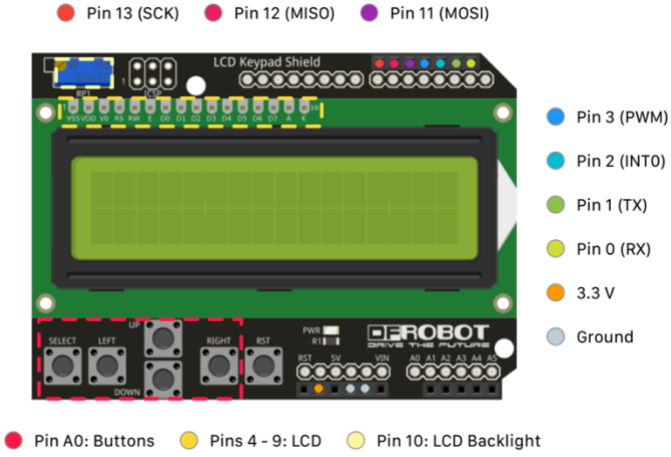
We need to generate a 40-bit UUID to assign to our radio's receiving data pipe to minimize the chances of two radios having the same address during lab. It is important to ensure that the address that you assign to the radio's receiving data pipe is **unique**. We will use the NR24 library to control 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.**

Name	Description	Connected To
CE	Chip Enable (RX / TX)	A1
CSN	SPI Chip Select	A2
MOSI	SPI Slave Data Input	11 or ICSP-4

Name	Description	Connected To
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

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:



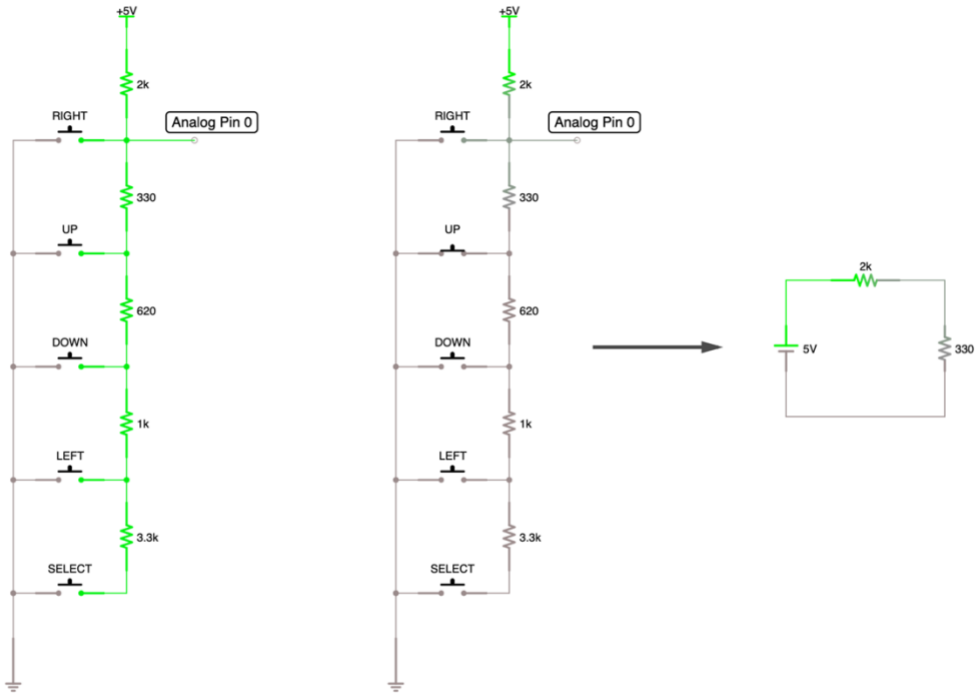
The following table describes which pins are used by the shield.

Pin	Description	Pin	Description
A0	Buttons	7	LCD DB7
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();

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
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

Address	Value	Purpose
278 - 290	Message Object	Message #9
291 - 303	Message Object	Message #10
304 - 316	Message Object	Message #11
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**. A template is provided below. You must not use Arduino's EEPROM class.

```
class Memory {
  public:
    Memory();
    Memory(Contact node);
    unsigned char* getNodeUUID();
    char* getNodeName();
    unsigned short getNumberContacts();
    unsigned short getNumberMessages();
    Contact getContact(unsigned short index);
    Message getMessage(unsigned short index);
    bool saveContact(Contact contact);
    void saveMessage(Message message);
    void saveNodeInformation(Contact contact);
    // Add as you see fit

  protected:
    bool hasSchema();
```

```
  void setSchema();
  void clearMessages();
  void clearContacts();
  unsigned short getMessagePointerOffset();
  // Add as you see fit

 private:
  const unsigned short MAX_CONTACTS = 10;
  const unsigned short MAX_MESSAGES = 20;
  // Add as you see fit

};
```

Let's start explaining what each of the functions must do.

- **Memory():** Default constructor for the Memory class. Makes sure that the **schema** is set up by verifying the flag bytes. If the **schema** is corrupted or has never been set up, we:
  - Set the **flag** bytes in their appropriate location.
  - Clear the locations reserved for contacts and messages.
  - Reset the contact and messages counters.
  - Reset the offset entry.
- **Memory(Contact node):** Parametrized constructor for the Memory class. Takes a Contact object as an argument. The contact object contains the information on the node's **name** and **UUID**. This performs the same operations as the default constructor plus saving the Contact object in the space reserved for the node's **name** and **UUID**.
- **unsigned char* getNodeUUID():** Returns the node's 40-bit **UUID**.
- **char* getNodeName():** Returns the node's **name**.
- **unsigned short getNumberContacts():** Returns the contents of contact counter entry.
- **unsigned short getNumberMessages():** Returns the contents of messages counter entry.
- **Contact getContact(unsigned short index):** Returns the Contact object specified by the index. If index points to an unsaved location, return an empty Contact.
- **Message getMessage(unsigned short index):** Returns the Message object specified by the index. If index points to an unsaved location, return an empty Message.

- **bool saveContact(Contact contact):** Saves the given Contact object into an empty location in the **EEPROM** and returns true. If no more empty locations are available, returns false.
- **void saveMessage(Message message):** Saves the given Message object into the next available location in the **EEPROM**. Note that this location might have a previous message that will get overwritten if there are no more empty message locations. In other words, after the twenty messages have been saved, the system will overwrite older messages to make room for the new ones.
- **void saveNodeInformation(Contact contact):** Saves the given Contact object into the node's contact location.
- **bool hasSchema():** Verifies that the **flag** bytes and **schema** are set for the node. Returns true if that's the case, false otherwise.
- **void setSchema():** Sets the **flag** bytes and **schema** for the node.
- **void clearMessages():** Clears the **EEPROM** locations associated with Message entries. Resets the messages counter.
- **void clearContacts():** Clears the **EEPROM** locations associated with Contact entries. Resets the contact's counter.
- **unsigned short getMessagePointerOffset():** Returns the value of the message pointer offset entry from the **EEPROM**.

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.

A template is provided below.

```
class Contact {
  public:
    Contact();
    Contact(unsigned char* givenUUID, char const* givenName);
    Contact(unsigned char* givenUUID, char givenName);
    void setUUID(unsigned char* givenUUID);
    void setName(char const* givenName);
    void setName(char givenName);
    unsigned char* getUUID();
    char* getName();
    // Add as you see fit
```

```
  private:
    // ...
    // ...
    // ...
};
```

Let's start explaining what each of the functions must do.

- **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.

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.

A template is provided below.

```
class Message {
  public:
    Message();
    Message(unsigned char* from, unsigned char* to, unsigned short payload, unsigned char length);
    Message(unsigned char* from, unsigned char* to, char const* message);
    void setLength(unsigned char length);
    void setTo(unsigned char* to);
    void setFrom(unsigned char* from);
    void setPayload(unsigned short payload);
    unsigned char getLength();
    unsigned char* getTo();
    unsigned char* getFrom();
    unsigned short getPayload();
    char* getPayloadString();

  protected:
    unsigned short stringToPayload(char const* message);
    char* payloadToString(unsigned short payload, unsigned char length);

  private:
    // ...
    // ...
    // ...
};
```

Let's start explaining what each of the functions must do.

- **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**.

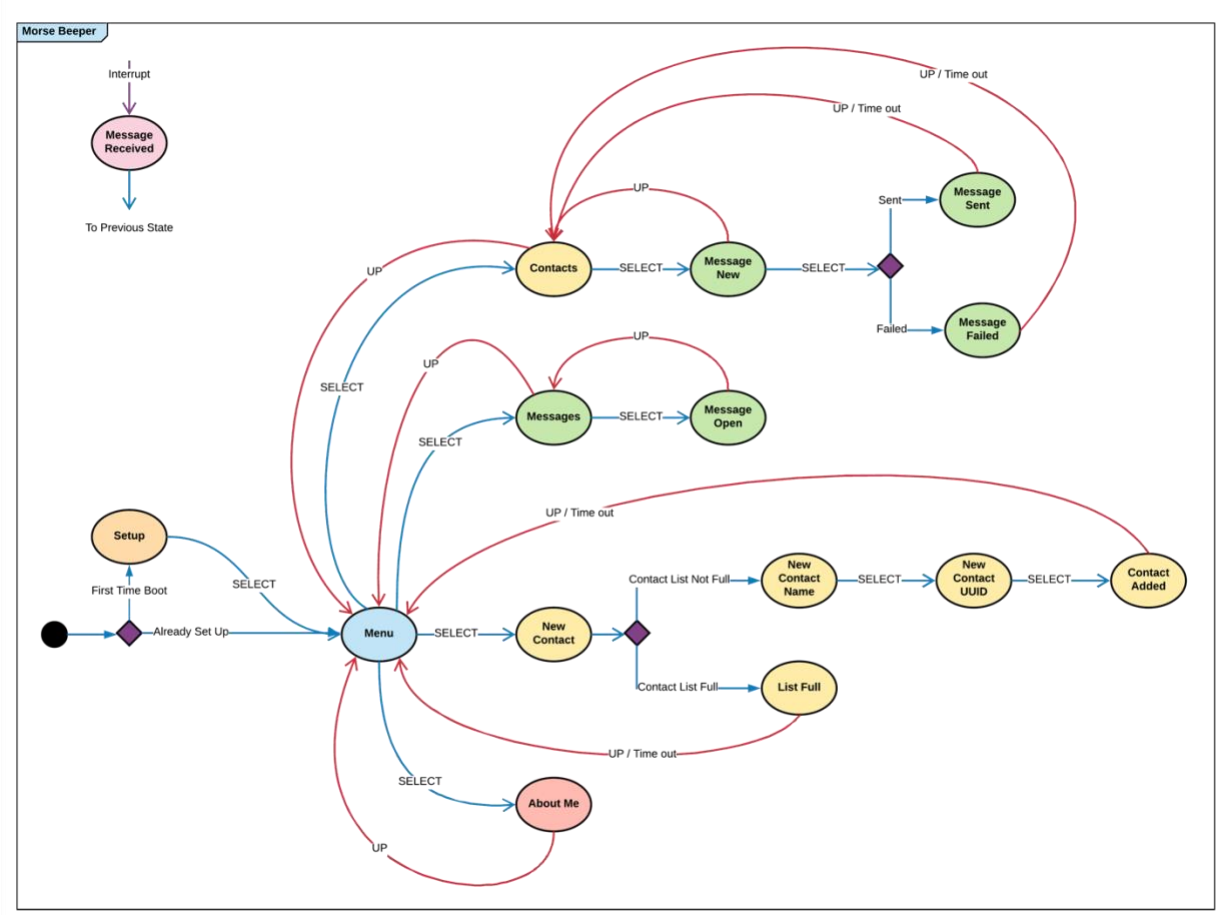
- **Message(unsigned char* from, unsigned char* to, char const* message):** Parametrized constructor for the Message class. Takes the sender and receiver's **UUIDs**. Additionally, it takes a message in a C-style string. The payload **length** is calculated after the conversion from a C-style string 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.

State Machine

We have modeled 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.

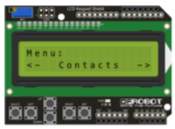
User Interface

The following section depicts the user interface that must be implemented for the beeper.

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

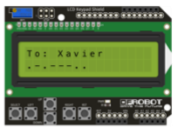
SELECT: None
LEFT/RIGHT: None
DOWN: None
UP: Back to previous screen.
Time out: Back to previous screen.

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.



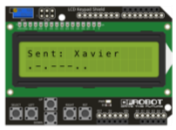
SELECT: None
LEFT/RIGHT: None
DOWN: None
UP: Back to menu.
Time out: Goes to menu.

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

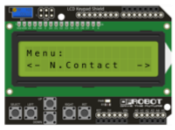
SELECT: None
LEFT: None
RIGHT: None
UP: Go back.
DOWN: None



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



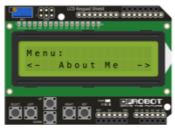
SELECT: Save name. Go to next screen.
LEFT: Erase last character.
RIGHT: None
UP: Scroll letter.
DOWN: Scroll letter.

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

Submission

- Once your team has completed the assignment show it your TA. You might be asked to reconfigure the device, explain your code, test communicating with other groups, etc.