CSE 321 Project 3:

Remote Tactile Communication Device

Wren Hobbs
University at Buffalo
Fall 2021

Table of Contents

Introduction

This device allows remote communication between blind, deaf, or deafblind individuals. One user has access to a braille keyboard, allowing them to enter a message into the system, and send the message to the other user, who receives the message in morse code via a vibration motor. Neither party needs to be hearing or sighted to be able to effectively use the device, as the user interface is purely tactile, save for an LED that outputs the morse code redundantly with the vibration motor.

Specifications

Inputs

- Braille keyboard
    - 3x2 bump button bus
        - buttons are held down based on what braille character the user wants to enter.
    - enter button
        - has the system read the bump buttons and add the appropriate character to the stored message.
    - backspace button
        - has the system erase the most recently added character to the stored message.
    - send button
        - has the system output the stored message as morse code.

Outputs

- Braille keyboard vibration motor
    - Acts as user interface; vibrates briefly when an input is received, and vibrates for a longer time when the user attempts to enter an invalid input
- Morse code vibration motor
    - System will output the stored message as morse code through this motor when the send button is pressed on the braille keyboard
- Morse code LED
    - Acts redundantly with the morse code vibration motor, outputting the stored message in morse code.

Threads

- main
    - Initializes event queues, threads, inputs, outputs, and the watchdog
    - kicks the watchdog every second
- isrThread
    - handles the ISR event queue
- enterThread
    - handles the functionality of the enter button
- backspaceThread
    - handles the functionality of the backspace button
- sendThread
    - handles the functionality of the send button and the morse code output

Functions

- main
    - Initializes event queues, threads, inputs, outputs, and the watchdog
    - loops forever
    - kicks the watchdog every second
- enter
    - run by enterThread
    - loops forever
    - when it receives a signal from enterISR, it reads the bump button input bus and calls brailleToText to convert it into an alphanumeric character and add to the stored message string
- enterISR
    - called when enterButton is pressed
    - gives signal to enterThread
- backspace
    - run by backspaceThread
    - loops forever
    - when it receives a signal from backspaceISR, it erases the most recently added character from the stored message string
- backspaceISR
    - called when backspaceButton is pressed
    - gives signal to backspaceThread
- send
    - run by sendThread
    - loops forever
    - when it receives a signal from sendISR, it calls printMorse to output the sent message as morse code
- sendISR
    - called when sendButton is pressed

        - gives signal to sendThread
    - brailleToText
        - called by enterThread
        - converts input integer (which enterThread set as the current bump button input) into alphanumeric character associated with that sequence of bumps
        - If the user inputs the “number mode” braille symbol, future braille inputs will be numbers until the user sends the message or enters the “letter mode” braille character.
    - dot
        - used by printMorse function
        - outputs a morse code “dot” to the vibration motor and LED
    - dash
        - used by printMorse function
        - outputs a morse code “dash” to the vibration motor and LED
    - printMorse
        - Reads stored message input, creating a local copy and erasing the original
        - loops through the copied message, outputting each character as morse code
        - releases control of the stored message string after clearing and making a copy, allowing the user to begin entering a new message while sendThread is outputting this message

Features

- Six-bump braille keyboard with tactile user feedback
- Alphanumeric message entry up to 140 characters long
- Morse code output via vibration motor and LED
- Functionality for concurrent user input and output
- System protection via watchdog
- Mutex-based critical section protection

Applications

- Digital 1-way conversion between braille and morse code
- Remote tactile communication between users without needing sight or hearing
- Remote alternative to tactile sign language

Internal Feature Integration

Watchdog

- Initialized by the main thread
- 8-second timeout
- Kicked every second by the main thread as the system runs

Threads/Tasks

- each non-bump button on the braille keyboard is controlled by a separate thread
- ISRs go through an event queue controlled by it own thread

Synchronization

- ISRs for the buttons signal the appropriate thread to stop idling and perform the action of the button
- each button thread has a mutex and conditional variable used to prevent bounce
- sendThread (which controls morse code output) releases critical sections before the slow process of outputting morse code, allowing the user to begin entering a new message while this thread is still outputting the previous one

Bitwise Driver Control

- Morse code LED output is initialized and controlled through bitwise drivers
- Other inputs/outputs are handled in a non-bitwise context to improve code readability

Critical Section Protection

- Message string is protected by a mutex. Each thread/function that accesses or changes the message string locks the mutex first.
- The input integer is protected by a mutex. This integer stores the state of the braille button bus, and is used by both the enter thread and the brailleToText function.

Interrupts

- enter, backspace, and send buttons on braille keyboard are controlled via interrupts
- When the button interrupt is called, it signals the appropriate thread's condition variable, causing the thread to perform the button's actions

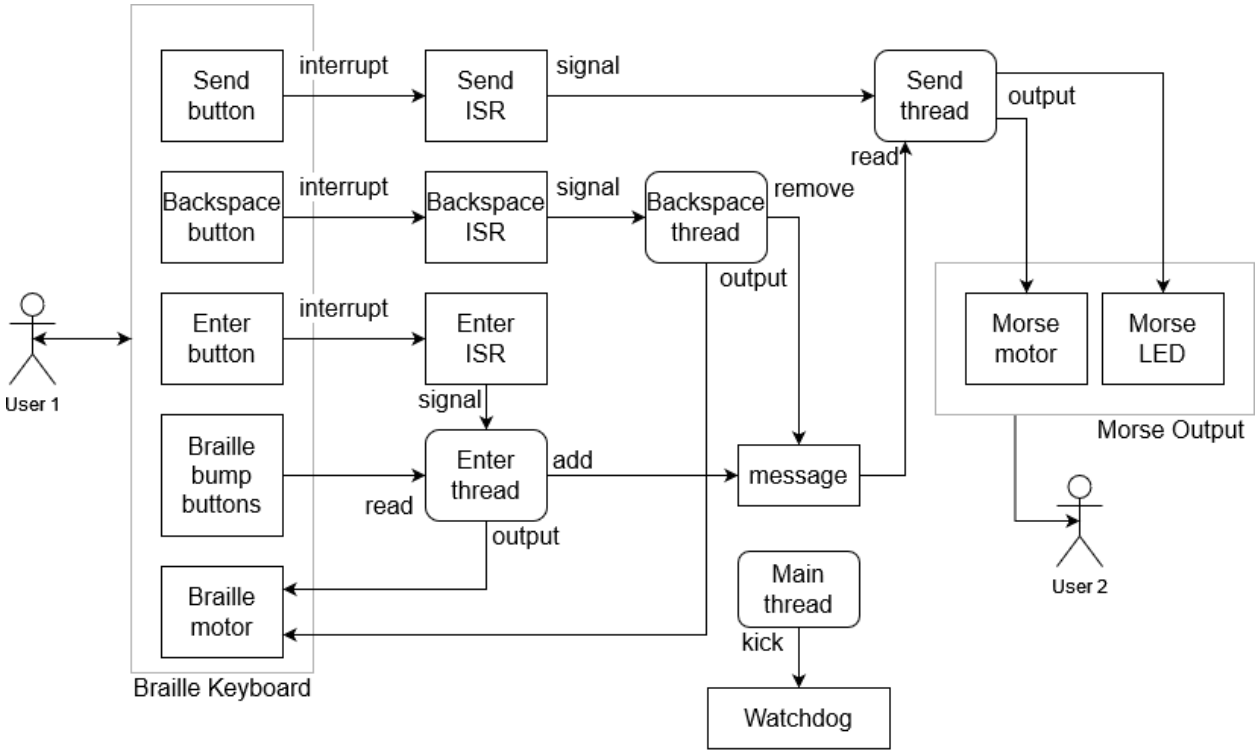
Design Process

The system was initially designed to utilize an ultrasonic sensor to detect a nearby user. This would cause the system to enter sleep mode if no user was detected for 30 seconds. However, this part of the system was determined to be cumbersome, difficult to set up for the user, and generally not especially necessary, and so was abandoned.

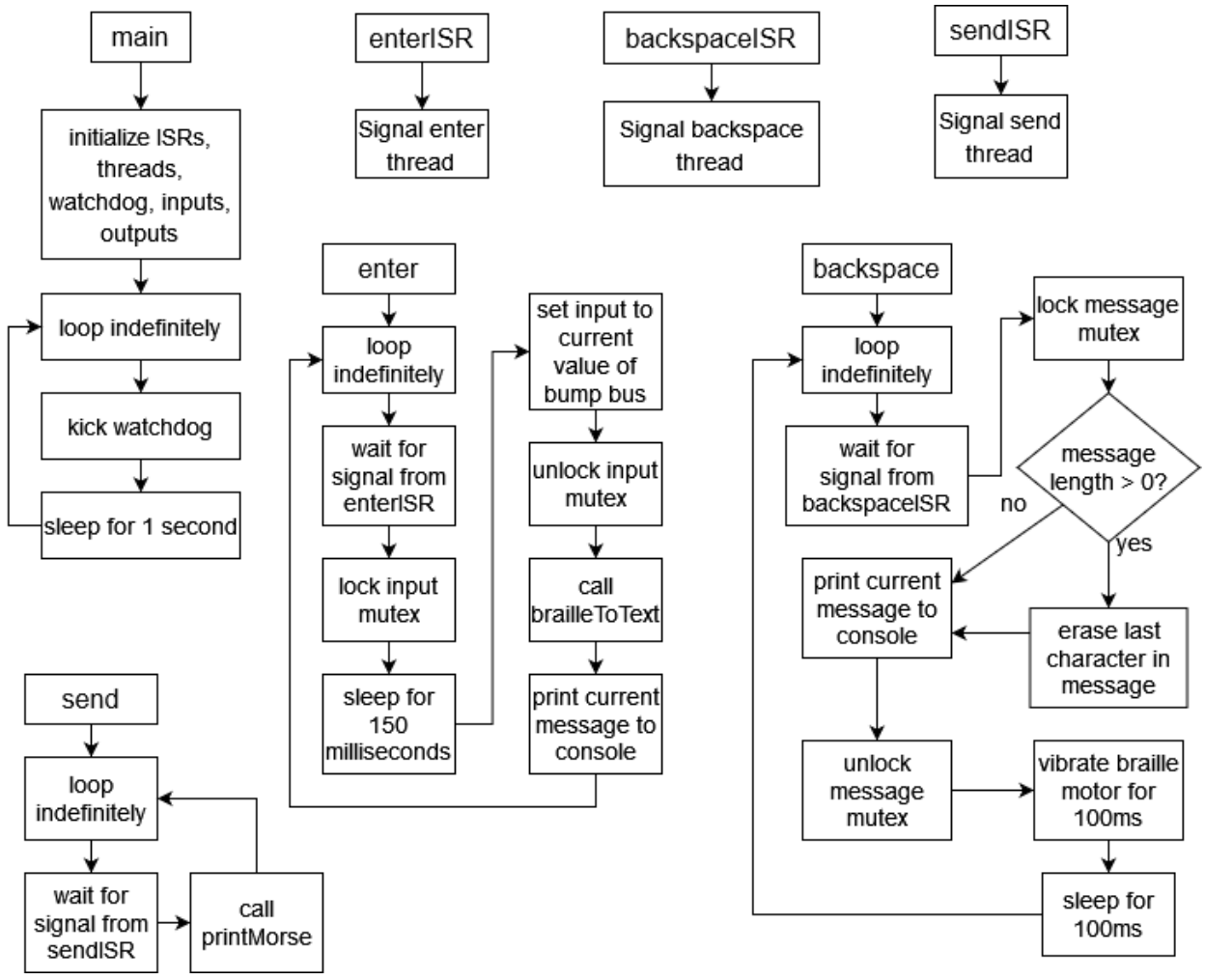
Additionally, the inclusion of LEDs or other visual-only outputs was considered carefully. Including these outputs would be of no benefit to many intended users of the system, so this component was included only redundantly with a purely-tactile output.

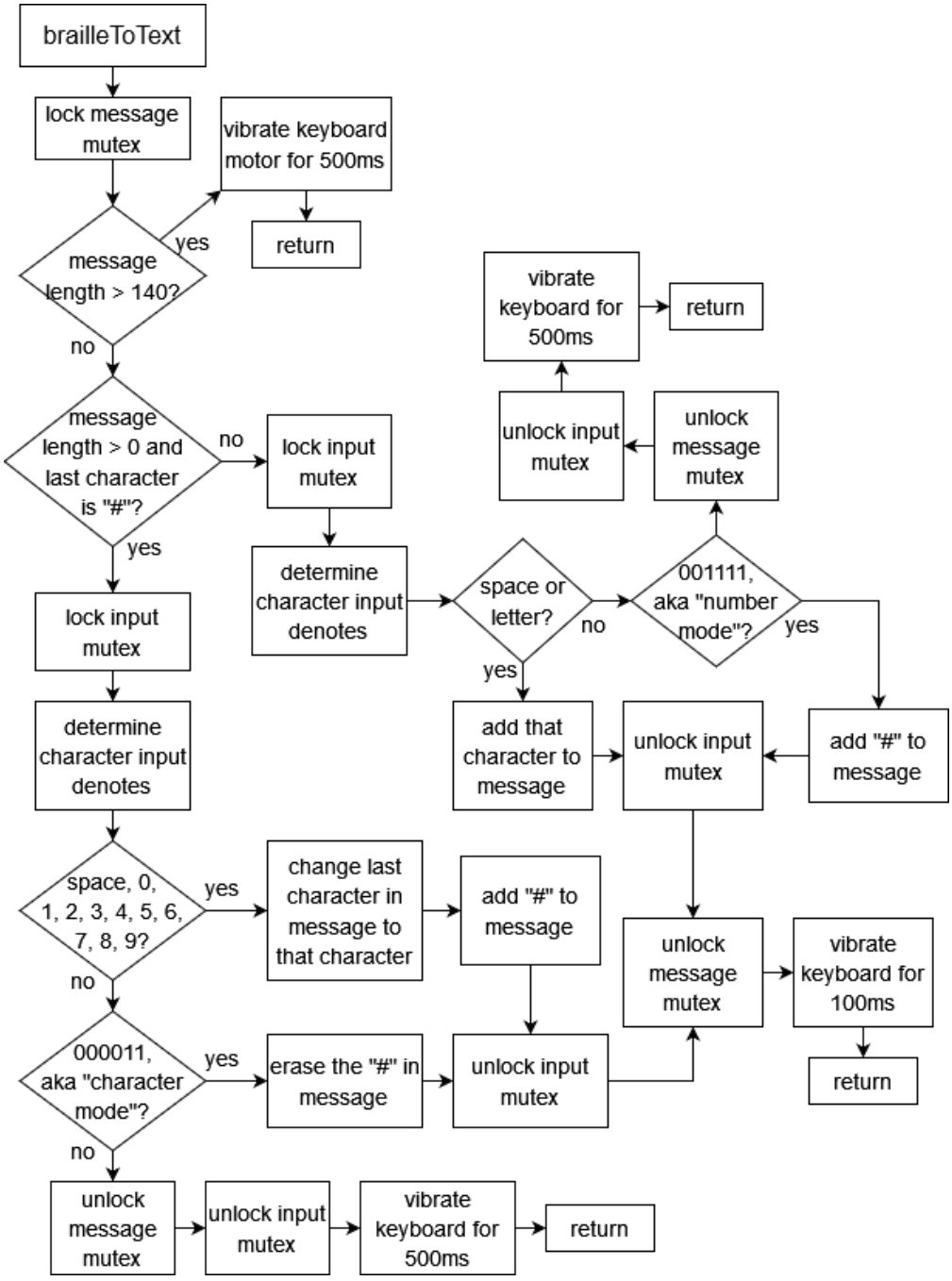
There were two different implementations of a braille keyboard that were considered for this system. The other would have the user press the bump buttons, and the system would add the button to the next character as they were pressed, moving to the next character when the enter button is pressed. This option is how physical braille typers work, but was not chosen, as it would have been more difficult to implement without error in a digital context, and also less convenient to erase mistakes.

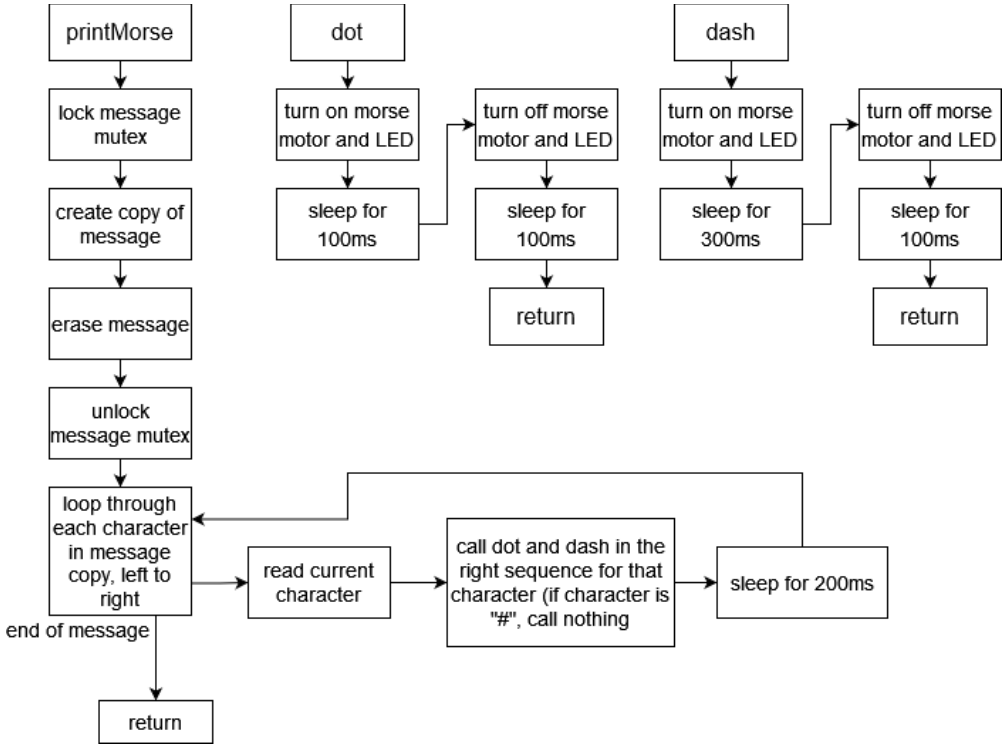
Block Diagram



Functionality




brailleToText
lock message mutex
vibrate keyboard motor for 500ms
return
message length > 140?
yes
no
message length > 0 and last character is "#"?
no
lock input mutex
yes
lock input mutex
determine character input denotes
determine character input denotes
space or letter?
no
001111, aka "number mode"?
yes
yes
vibrate keyboard for 500ms
return
unlock input mutex
unlock message mutex
add that character to message
unlock input mutex
add "#" to message
space, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9?
yes
change last character in message to that character
add "#" to message
no
000011, aka "character mode"?
yes
erase the "#" in message
unlock input mutex
unlock message mutex
vibrate keyboard for 100ms
return
no
unlock message mutex
unlock input mutex
vibrate keyboard for 500ms
return



Bill of Materials

Technology Requirements

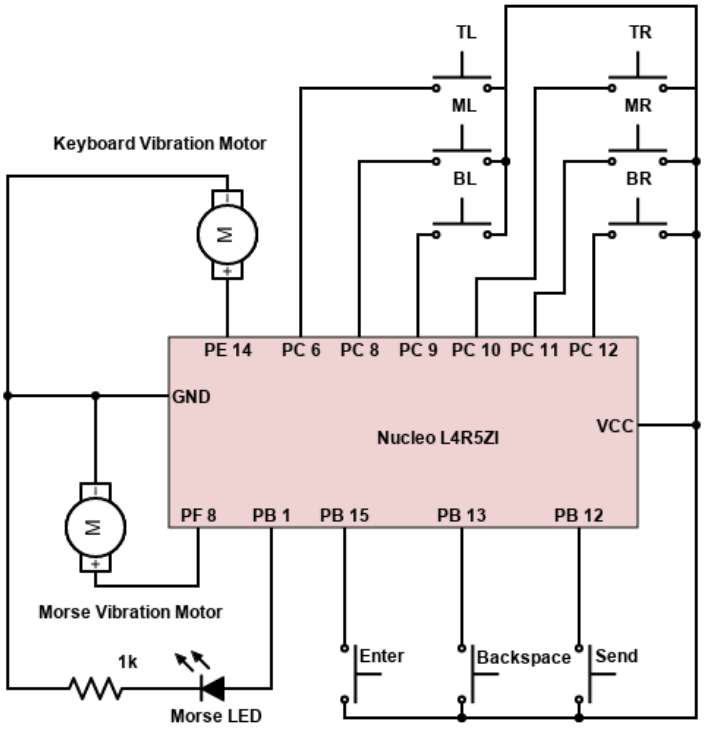
- MBed OS Framework/ Mbed Studio

Required Hardware

- 1x Nucleo-L4R5ZI microcontroller
- 9x push button
- 2x vibration motor
- 1x LED
- 1x 1kΩ resistors

User Instructions

Schematic



Building the System

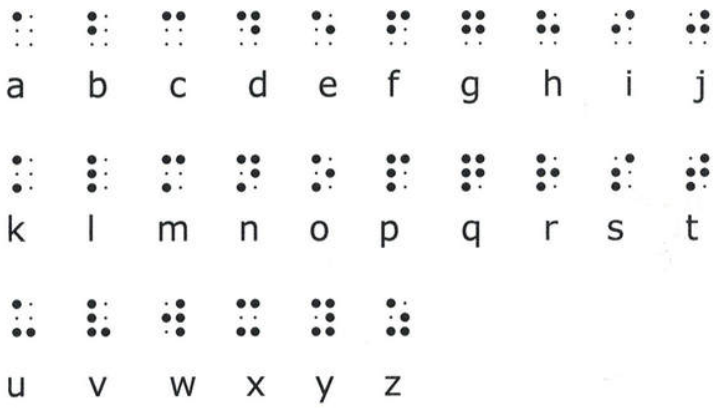
- Connect one end of all nine push buttons to PD 0 (vcc)
- For the other end of each push button:
    - Top-left bump button: connect to PC 6
    - Middle-left bump button: connect to PC 8
    - Bottom-left bump button: connect to PC 9
    - Top-right bump button: connect to PC 10
    - Middle-right bump button: connect to PC 11
    - Bottom-right bump button: connect to PC 12
    - Enter button: connect to PB 15
    - Backspace button: connect to PB 13
    - Send button: connect to PB 12
- Connect the resistor and LED in series; connect the positive end to PB 1 and the negative end to ground.

- Connect one end of both vibration motors to ground.
- Connect the other end of the morse vibration motor to PF 8.
- Connect the other end of the braille keyboard vibration motor to PE 14.

Using the System

Entering a letter:

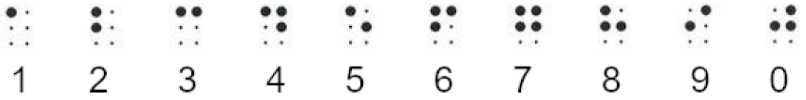
1. If the keyboard was put into “number mode”, return the system to “letter mode” by holding down the middle-right and bottom-right bump buttons (⠒) and press the enter button.
2. Hold down the appropriate bump buttons for the desired letter (see chart below). Raised bumps are held down, unraised bumps are left alone. Holding down *no* bumps will enter a space.
3. Press the enter button. The letter will be added to the message and the keyboard will vibrate for 100 milliseconds. If the keyboard vibrates for 500 milliseconds, it has received an invalid input, and did not add a character to the message.



source: louisbrailleonlineresource.org

Entering a number:

1. If the keyboard is in “letter mode”, put it into “number mode” by holding down the bottom-left bump button and the entire right column of bump buttons (⠸).
2. Hold down the appropriate bump buttons for the desired number (see chart below). Raised bumps are held down, unraised bumps are left alone.
3. Press the enter button. The number will be added to the message and the keyboard will vibrate for 100 milliseconds. If the keyboard vibrates for 500 milliseconds, it has received an invalid input, and did not add a character to the message.



Note: When a message is sent, the keyboard will be put in “letter mode”.

Removing a character from the message:

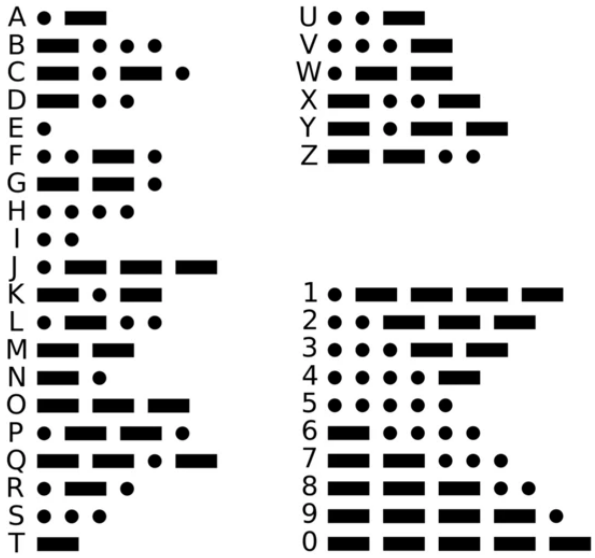
Press the backspace key on the braille keyboard. it will erase the most recently entered character from the message.

Sending a message:

After entering a message, you can send it to the other user by pressing the send button. This will erase the message and output it as morse code to the output vibration motor and LED. It will also set the keyboard to “letter mode”.

Receiving a message:

When the send button is pressed on the braille keyboard, the output motor will vibrate in morse code to output the message (chart for translation shown below).



source: modernout.com

Test Plan

Unit Tests

- Send a message that includes every character to ensure that they are all translated correctly.
- Send an empty message and ensure there is no unexpected output.

- Attempt to enter invalid braille characters. the keyboard should vibrate for a half second to denote an error.
- Toggle between numbers and letters in a message to ensure the correct type of character is entered.
- Attempt to add more characters to the message after it is at maximum size. Attempting to add a 141st character to the message should cause the keyboard to vibrate for a half second and not add the character.
- Attempt to write a new message while the previous message is being sent. If the braille keyboard does not work completely while the system is outputting a message means there is a system error.
- Attempt to overflow the event queue by pressing enter, backspace, and send in rapid succession. If it is possible to overflow the queue in this way, there is a system error.

Results

The system succeeded all of these tests by its final implementation. Some problems were encountered during the implementation process, including the following:

- There was an error with the bump bus giving incorrect values to what was pressed by the user. Changing the bus' mode to pull up and rewiring the system slightly solved the issue.
- Numerical input was originally designed with a separate global variable. If it was true, the system was in "number entry" mode, and if it was false, in "character entry" mode. This was later simplified by having the brailleToText simply add a "#" character to the message when they enter the number symbol. when brailleToText is run, it reads the last character in the message, and treats the input as a number if it is a "#", and the new character is entered before the number symbol. When the user enters the "character" symbol, the "# is erased. This solution avoids the creation of a global variable, which would have to be protected by something like a mutex.
- The braille keyboard buttons were initially written to run through the ISRs via the event queue, however this caused bounce to be an issue, and made overflowing the event queue a real issue. With the more recent implementation, these problems have been addressed.

Recommendations for Improvement

- Option to change output speed of morse code. As of now, the speed of one "time unit" for morse code output is hardcoded to 100ms. An ability to change this would be a valuable addition to the system.
- An additional "cancel" button, which would erase the entire message
- An infrared detector to allow wireless communication between parties
- Ability to convert between different braille languages