Final Project: Dits and Dahs

ECE 167 Sensing and Sensor Technologies

March 14, 2024

Commit ID: 12e9d5736bb6e2f77d85c66409e8580cc2c29aaa

Niki Mobtaker, Jacob Gutierrez, Aleida Diaz-Roque

Dits_and_Dahs

Contents

1 Introduction

In our final project, "Dits and Dahs," we created an interactive training game that enhances both writing and listening skills in Morse code. We implemented various sensors to transmit messages, including a capacitive touch sensor, a homemade capacitive touch sensor, and two flex sensors. Both the Custom 167 CapTouch and Peso CapTouch operated on duration of touch determining a dit or dah, similar to a one-pad Morse keyer. The flex sensors were attached to a thumb and index finger and the finger chosen to bend determines a dit or dah, similar to a two-pad Morse keyer.

The game is built with multiple levels that increase in difficulty. The user is prompted with a set of letters on the OLED display and must use the selected sensor to make a guess. The game then deciphers the user's input and determines if their guesses were correct or incorrect. Once the user's input matches the corresponding Morse code translation enough times, the user moves onto the next level. Additionally, the user can switch the sensor they want to use by using the STM’s onboard buttons.

The primary experiment was to determine which of the three sensors was most effective for transmitting Morse code. To do this, we conducted a comparative evaluation of commercially available and homemade capacitive touch sensors and flex sensors. Our evaluations focused on the qualitative and quantitative measurements of the sensors given a wide range of user input. These characteristics are most important for our sensors due to the way that Morse code is decoded. User input can be incorrectly decoded very easily if the sensors behave in an unreliable way, so one of our primary goals was to find out how to increase the accuracy of our sensors.

2 Background

2.1 Morse Code

Morse code is a method of encoding text characters as standardized sequences of two different signal durations, called dits and dahs. Specific combinations of these dits and dahs can be translated into letters using the International Morse Code Standard shown in Figure 1.

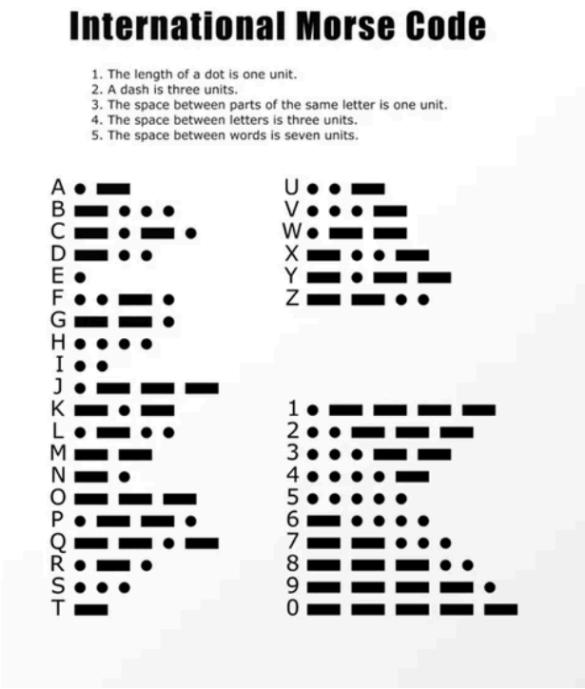


Figure 1 - International Morse Code Alphabet

This standard gave us a rubric for how we should be encoding and decoding the morse code. Referring back to our previous labs, we learned how to read, utilize and condition the output from several different sensors using the STM32 microcontroller. After considering all the sensors that we had worked with, we decided to use the two that we felt would be the best at transmitting morse code.

2.2 Capacitive Sensors

Capacitive sensors work by detecting changes in capacitance when a conductive object, such as a human finger, comes into close proximity. The basic principle is based on the capacitance shown in Equation 1.

$$C = \frac{\varepsilon_r \varepsilon_0 A}{d} \qquad [1]$$

where:

- C is the capacitance,
- ε_r is the relative permittivity of the dielectric material,
- ε_0 is the permittivity of free space,
- A is the area of the plates, and
- d is the distance between the plates.

In a capacitive sensor, one plate of the capacitor is formed by the sensor electrode, and the other plate is formed by the conductive object (e.g., a finger). When the object comes close to the sensor, it changes the effective distance *d* between the plates, which in turn changes the capacitance *C*. This change in capacitance can be detected and used to infer the presence or absence of the object. So, how do we use this to detect if a user is touching (or *not* touching) the sensor?

2.3 LM555 Timer - Astable Mode

The LM555 is a highly popular integrated circuit used primarily as a timer or pulse generator. It can be configured in several modes, with astable mode being one of them. In astable mode, the

LM555 operates as a free-running oscillator, continuously generating a square wave output without any external trigger. This mode is characterized by two main features: the output waveform oscillates between high and low states, and the frequency and duty cycle of the waveform can be adjusted by changing the values of external resistors and capacitors connected to the IC. Below is the schematic used to configure the LM555 in astable mode:

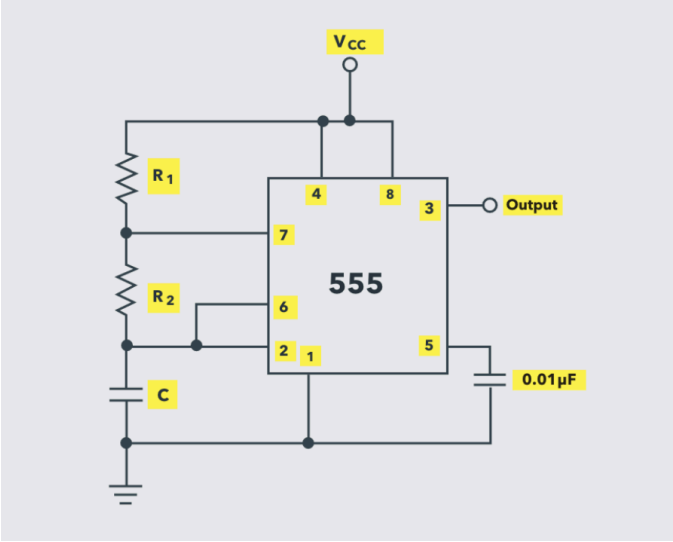


Figure 2 - Astable Multivibrator w/ 555

One of the primary benefits of this circuit is that we can adjust the output frequency of the oscillator, by varying the resistor and capacitor values. In our case, the capacitive touch sensors output a change in capacitance depending on user input. Equation 2.1 determines output frequency and Equation 2.2 determines duty cycle.

$$f = \frac{1.44}{(R_1+2R_2)*C} \qquad [2.1]$$

$$D = \frac{R_1+R_2}{R_1+2R_2} \times 100\% \qquad [2.2]$$

2.4 Capacitive Touch Sensor + LM555

If we connect a capacitive touch sensor in parallel with *C*, whenever the user touches the sensor, the overall capacitance will increase and result in a smaller frequency. The updated equation for the output frequency is Equation 2.3.

$$f = \frac{1.44}{(R_1+2R_2)*(C+C_{Sensor})} \qquad [2.3]$$

In Lab 2, we created a program to take in the square wave output and monitor the change in period. This allowed us to detect whether the sensor was touched or not touched, which we implemented into a function that returned a true or false value. But in order to integrate our own sensor, we needed to analyze how a different change in capacitance would affect the circuit, which is discussed in the *Implementation* and *Evaluation* sections.

2.4 Flex Sensors

A flex sensor or flexible potentiometer is composed of conductive ink and segmental conductors that serve as an overall path of current through the sensor, shown in Figure 3.

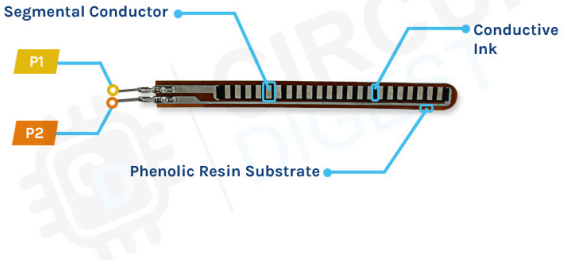


Figure 3 - Flex Sensor Diagram

The conductors serve as a consistent conductive path throughout the sensor. The conductive ink decreases in width as the sensor bends. The ink thinning provides less space for the electrons to

flow making it less conductive or more resistive. We use this property of the sensor to track motion of the user's fingers, which will be explained in *Implementation.*

3.0 Implementation

The project implemented several sensors and different algorithms/data structures to execute the game properly. In this section, we discuss the final product and why techniques were used.

Our project uses the following components:

Component	Quantity
STM32 Microcontroller	1
Custom ECE167 Capacitive Touch Sensor	1
SEN-10264 Flex Sensor + Mount	2
Peso (Coin)	1
LM555 Timer IC	1
COM-11089 Speaker	1
TPA2005D1 Mono Audio Amp Breakout	1
Passive Electronic Components (Resistors, Capacitors, etc.)	As Needed

Table 1 - Parts List

3.1 Sensor Implementation

The goal for each sensor was to encode dits and dahs accurately and the code/game would operate with the sensors smoothly. Figure 4 displays the block diagram of the hardware connections to the STM 32.

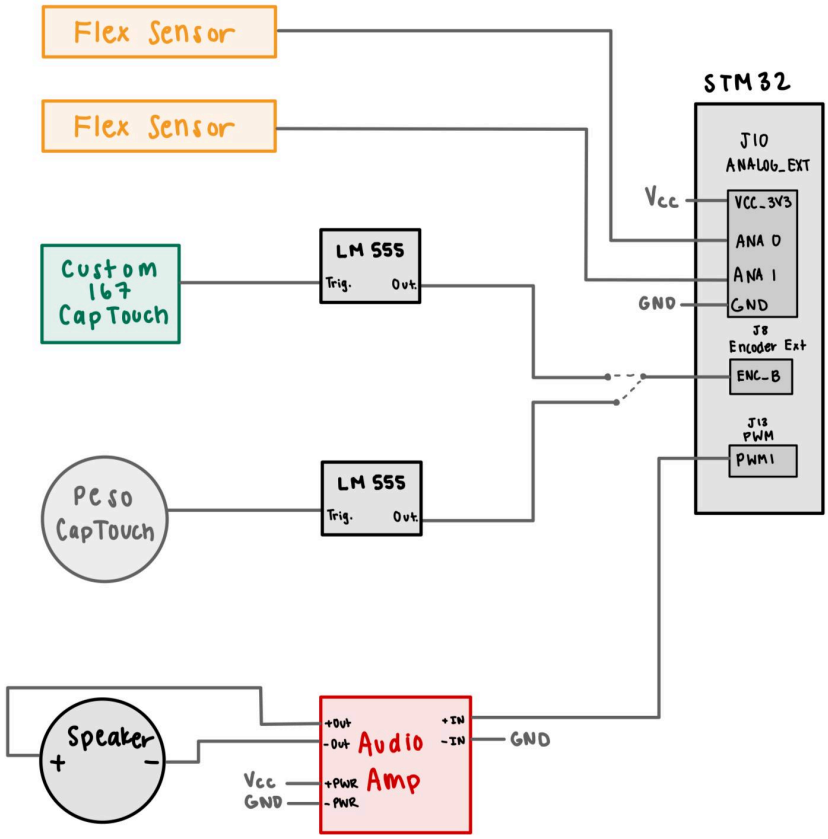


Figure 4 - Hardware to MC Block Diagram

3.1.1 Flex Sensor

We used the same voltage divider circuit for the flex sensors as Lab 1. Figure 5.1 shows the circuit schematic to the flex sensors.

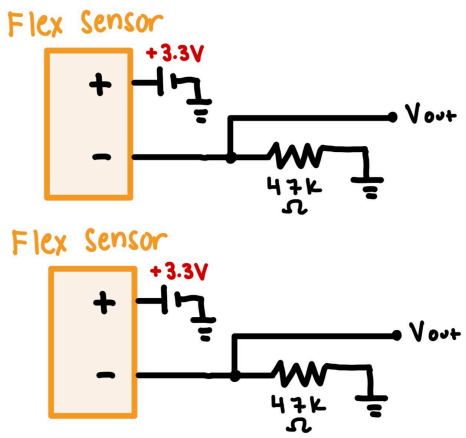


Figure 5.1 - Flex Sensors Circuit Schematic

Through using the voltage divider formula, Equation 3, and the range of resistance of the flex sensors, we found the output voltage ranged from 1.1 - 2 V.

$$V_{out} = V_{in} \frac{R_2}{R_1 + R_2} \quad [3]$$

$$30k\,\Omega\text{: } V_{out} = 3.3 \frac{47k}{30k + 47k} = 2.01\,V$$

$$90k\,\Omega\text{: } V_{out} = 3.3 \frac{47k}{90k + 47k} = 1.13\,V$$

These voltages output to the Analog Ext., shown in Figure 4, as a direct reading to the command line interface.

The flex sensor prototype is modeled after a two-pad morse keyer shown in Figure 5.2. One pad would represent a dit and the other a dah. We determined that the thumb flex sensor bending was a dit and the index flex sensor bending as a dah. The final prototype of the Flex Sensor Morse Keyer included weaving the flex sensors through the glove and using velcro strips to fasten the sensors to the fingers, shown in Figure 5.3.

Figure 5.2 - Two-Pad Morse Keyer

Figure 5.3 - Flex Sensor Morse Keyer

3.1.2 Capacitive Touch Sensors (Lab kit and Homemade)

In section **2.4**, the implementation derived from our second lab experiment was briefly described. But rather than just repeating the experiment, we expanded to determine what the best configuration of the LM555 would be for accurate readings. Since we needed to determine how long a user touched the sensor to decode the message, the sensitivity needed to be reduced. Referring to the Lab 2 manual, the change in capacitance of the lab kit cap sensor was said to be in the picoFarad range. So initially, a 22pF capacitor was placed in parallel with our CapTouch sensor.

The values that we landed on for the lab kit CapTouch were changed so that the change in the oscillator's frequency could be interpreted by our program accurately. Our experimental results and justification for this will be presented in the *Evaluation* section. Below in Figure 6.2 is the circuit schematic that we implemented into our game with an SPDT switch to easily flip between the two sensors, and a telegraph key that we modeled our CapTouch prototype after as seen in Figure 6.1.

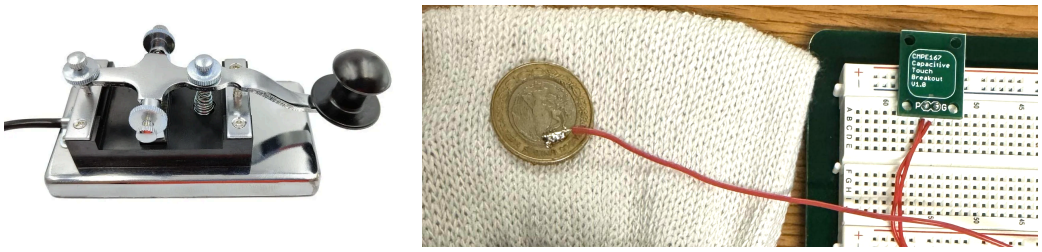


Figure 6.1 - Telegraph Keyer & CapTouch Prototype

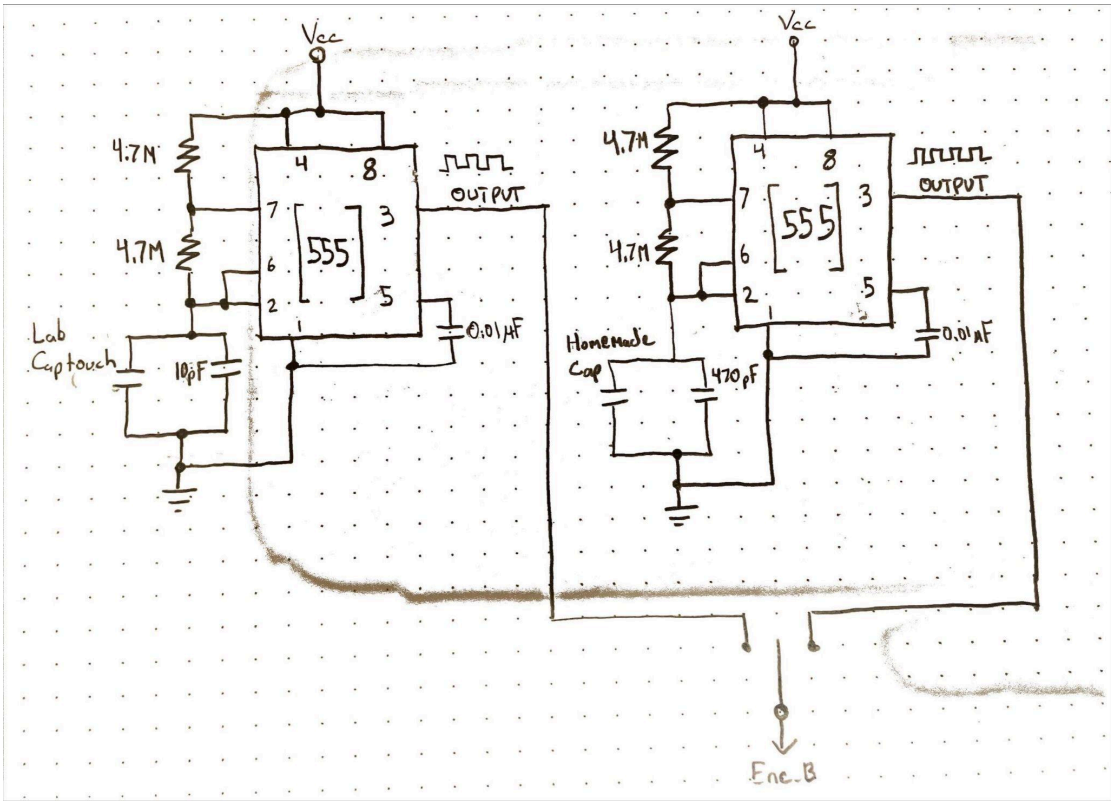


Figure 6.2 - CapTouch Wiring Diagram

3.2 State Machines

3.2.1 Flex Sensor State Machine

The implementation of the flex sensor readings and the algorithm required a state machine containing two states, Figure 8 shows the state diagram.

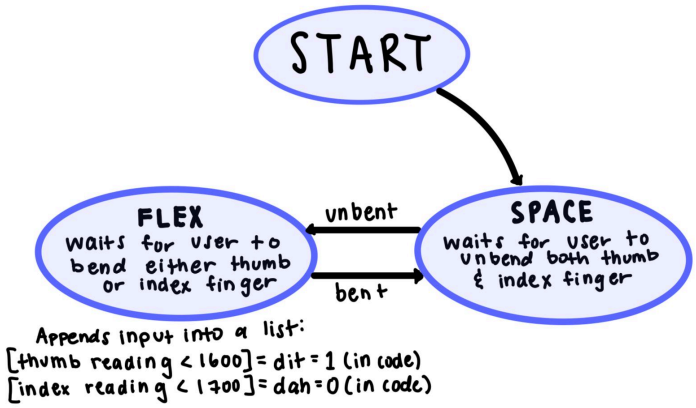


Figure 8 - Flex State Machine

With both states, the code reads inputs of ADC_0 and ADC_1 connections followed by an if loop that the algorithm can enter once the readings surpass certain thresholds. The readings were linear with the sensor's bend, so we used live readings to determine the minimum/maximum readings for thresholds. The state machine starts at "Space" where the reading of both thumb and index must be greater than 1800 and 1900 respectively. The thresholds varied per finger because the mobility of the thumb is less than the index finger. Once that condition has been met the state is set to "Flex". The thumb and index finger were implemented as an if else statement. The algorithm can enter either statement once the readings are less than 1600 for the thumb or 1700 for index. We decided the thumb bending translates to a dit and index to a dah. To make the code simple, we represented a dit as an integer 1 and dah as an integer 0. These inputs would append to the linked list "Combo" that was passed in from the function DecodeMorse(). After appending the readings, the state transitions back into "Space".

3.2.2 CapTouch State Machine

The flag that moves the state machine from state to state is initialized in the *StateMachine* struct called *Touch*. The machine consistently checks the touch status of the capacitive sensor. The '*CapTouch*' function processes the state machine based on the touch status and its duration:

State	**Action**
Wait	Waits for touch detection. Upon detection, records the start time and transitions to the 'Touch' state. Outputs "Just Touched" for debugging.
Touch	Remains in this state while the sensor is touched. Upon release, records the end time, calculates the touch duration, and outputs the duration for debugging. Transitions to the '*decode*' state if the duration exceeds the minimum threshold for a Dit.
Decode	Determines if the touch duration corresponds to a Dit, a Dah, or an invalid touch based on predefined duration thresholds. The thresholds follow the standard that a Dah is three times the length of a Dit. Adds the decoded symbol to a list representing the Morse code sequence, then transitions back to the '*wait*' state.

Table 2 - CapTouch States

The portions of the code set for debugging were removed once we determined that the algorithm worked.

Below is a diagram for the CapTouch to Morse Code Transmitter state machine:

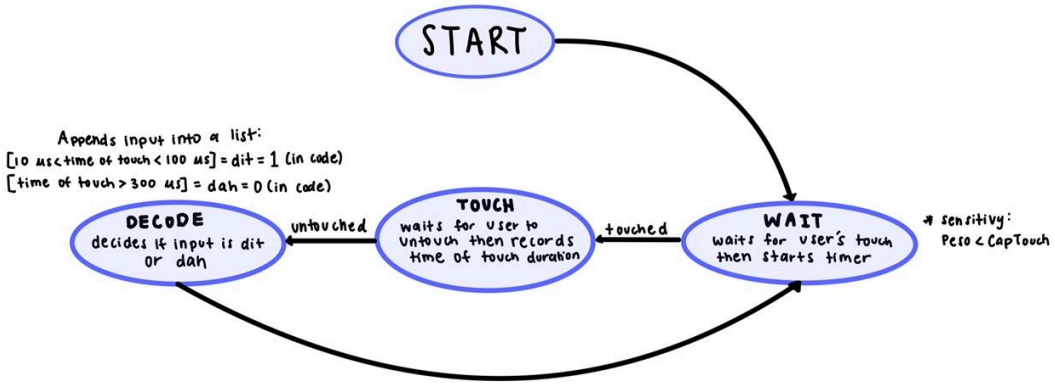


Figure 9 - CapTouch State Machine

3.3 Libraries

To implement the algorithm for the game, we used thirteen libraries. Five libraries were from the standard library, six libraries from class and two libraries created by the team. From class libraries we used *Board.h* to control the board, *Oled.h* to be able to display the user interface, timers.h to determine the duration of user inputs, *PMW.h* to control the speaker, *ADC.h* to take in inputs from the flex sensors, and *buttons.h* to determine the state of the buttons pressed during the game.

Two libraries were created by the team. CAPTOUCH.h was created in Lab 2 and was used to determine when the capacitive touch sensors, from the lab kit and homemade, were touched. List.h implements a linked list data structure with an embedded cursor that we used to store the set of letters to be tested in the game. This list structure is referenced as “LetterBank” in the pseudo code in Figure 10.1. The LetterBank changes size when the levels increase as new letters are appended into the list, so a linked list was necessary to accommodate variable length arrays.

```
Initialize game variables (level, sensor type, etc.)
Initialize LetterBank to contain the letters to be learned and tested

Display Welcome Screen:
      Show welcome message
      Prompt user to select a sensor

Select Sensor:
      Wait for user input to choose Flex, Peso, or CapTouch
      Update display to show selected sensor

Start Level:
      Learning Phase
      For each new letter in level:
            Display the letter and play the Morse code

            User Input Phase:
                  Read user input through the selected sensor
                  If the input matches the Morse code:
                        Display "Correct"
                        Move on to the next letter
                  Else:
                        Display "Incorrect"
                        Repeat the same letter

      Teaching Phase
      Repeat until all letters are guessed correctly at least 3 times:
            Randomly select a letter from the LetterBank
            Prompt user to enter Morse code for the selected letter

            User Input Phase:
                  Read user input through the selected sensor
                  Check user input against correct Morse code
                  If correct:
                        Display "Correct"
                        Decrease remaining guesses for the letter
                  Else:
                        Display "Incorrect"
                        Increase remaining guesses for the letter
                        If user has guessed incorrectly 3 or more times:
                              Play the Morse code

            Check Level Completion:
                  Check if all letters were guessed correctly at least 3
                  times
                  If yes:
                        Increase the level
                        Introduce new letters
                        Return to Start Level
```

Figure 10.1 - Game Overview Pseudocode

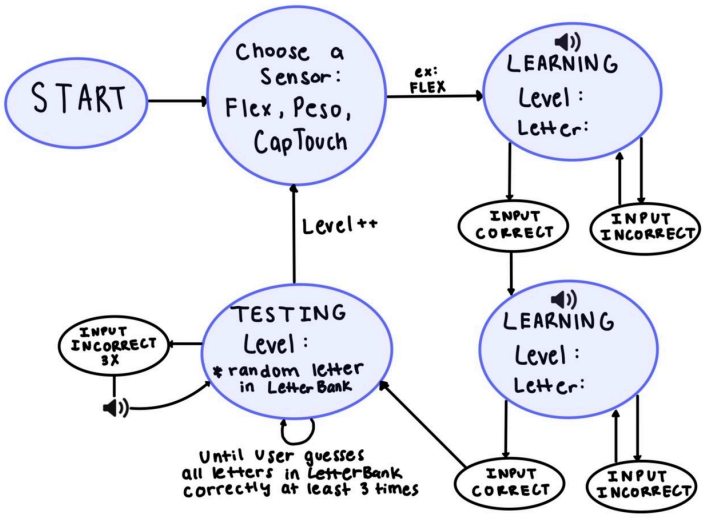
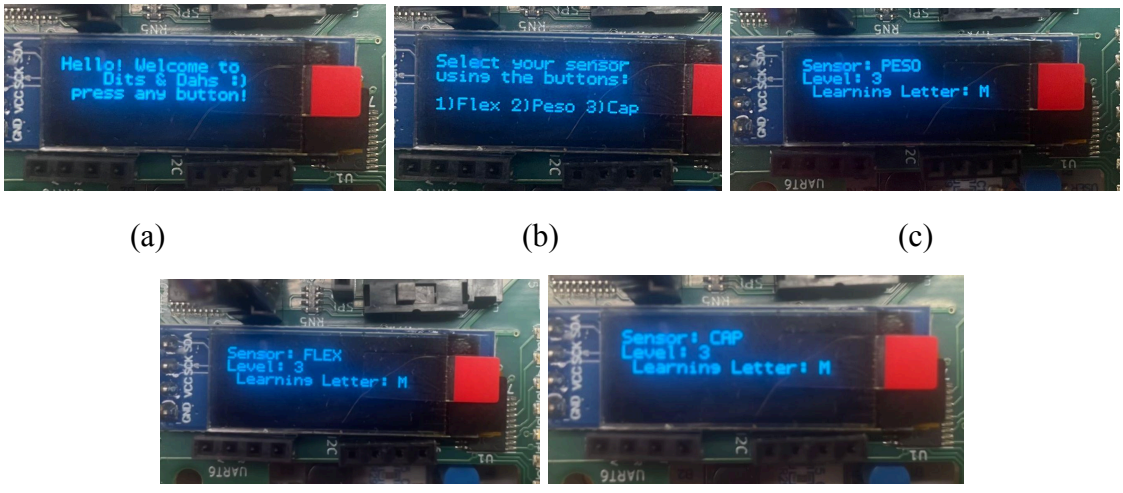


Figure 10.2 - Game Overview State Machine

3.4 General Algorithm

The algorithm of the game follows the pseudo code, Fig. 10.1, which implements the general state machine in Fig. 10.2**.** The user interface (UI) of the game is displayed in Fig. 11.1. The game runs continuously through the levels until the user has reached the final level and the user has learned all the letters in the alphabet.



(a) (b) (c)

(d) (e)

Figure 11.1 - User Interface

At the initialization, two linked lists are created, LetterBank, which will store the letters to be taught and learned and GuessesRequired that stores the number of guesses required for each letter to be “learned”. First, two new letters are appended into LetterBank and then the Game Overview State Machine (Fig. 10.2) is called.

Initially the user is welcomed and prompted to press any button to start the game, Fig. 11.1(a)**.** Once in the Choose a Sensor state they can use the buttons to pick which sensor they would like to play the current level, Fig. 11.1(b). After the user picks the sensor the game begins and we enter the learning mode of the game. The first letter is played and displayed on the OLED along with the sensor the user chose and the current level as can be seen in Fig. 11.1(c-d) where the sensor displayed varies. The binary representation of this letter will be stored as a list to use as a comparator to the user input list.

The user will input the corresponding dit or dah based on the letter. In the example UI, this letter is M which corresponds to “Dah Dah”. If the chosen sensor is Flex, the correct user input would correspond to two bends of the index finger. The algorithm would call the Flex Sensor State Machine to append two zeros to a list. If the chosen sensor was Peso or CapTouch, the correct user input would correspond to two long presses of the sensor. The CapTouch State Machine would be called to determine the duration of the touches and append the result into a list. Once the user input list was made it is compared to the correct binary representation of the Morse code letter being tested. In the example, if the user input is correct the comparison would be between [0, 0] and [0, 0]. The game moves on to teach the user the next new letter by following the same

procedure of playing the letter and getting the user input. For these two new letters, the user must guess them both correctly to move on to the next mode.

After learning the new letters the game moves on to teaching mode. Now, the user will be tested on letters chosen at random from the LetterBank. A correct guess would decrease the required guesses for the letter being tested. The list GuessesRequired maintains this tracking, where the index of the guess to be decreased aligns with the index of the corresponding letter in LetterBank. The relationship between these lists can be visualized in Fig.11.2 where the required guesses for a correct guess of letter A is decreased by one. For an incorrect guess, the guesses required would be increased for that letter. At any point if the user guesses the letter incorrectly three or more times in a row, the Morse code corresponding to the letter would play to help the user learn the letter.

At Level 3:
Testing Letter: A
LetterBank: [E, T, **A**, N, I, M]
GuessesRequired: [3, 3, **2**, 3, 3, 3]

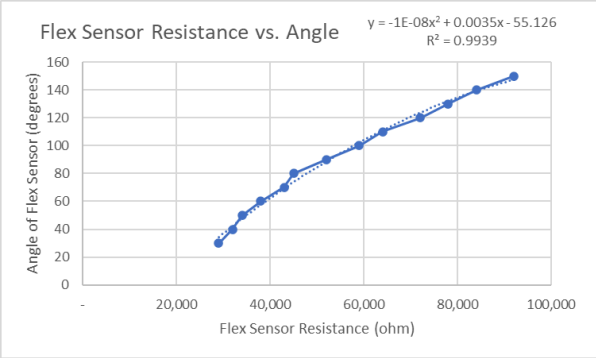
Figure 11.2 - User Interface Example

Once every letter was guessed correctly at least three times, the level would be completed. This would result in a return to the Chose a Sensor State where the user can pick what sensor they would like to use in the next level.

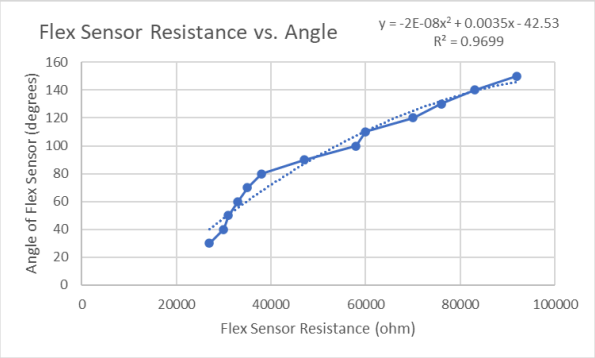
4.0 Evaluation

4.1 Flex Sensor

The plots of both flex sensors' angle vs. output resistance from Lab 1 is shown in Plots 1.1 and 1.2.



Plot 1.1 - Flex Sensor 1 Data



Plot 1.2 - Flex Sensor 2 Data

From this data, we determined that the flex sensors behave linearly and there was no need for calibrating raw readings.

We created a function for decoding flex sensor inputs for easy integration into the whole code of the game. The function would pass in the updated list "Combo" to DecodeMorse() where this function is called. We first implemented if loops, not a switch/case state machine. The general idea was the user had to unbend between bending to separate each input of a letter. The if loops did well on properly reading the different inputs, however, the flex sensor would read so fast that many readings of one input would append to the list. We tried using delays, but the code would get eaten up and not function properly. That's when we decided to transition to a state machine that would force the states to switch between each other (no external input) and have conditional if statements to transition. This worked really well and became the final code.

At first, to record readings for the thresholds of the conditional statements, we created a prototype of the sensor keyer using male to female jumper cables to fasten, shown in Figure 7.1.

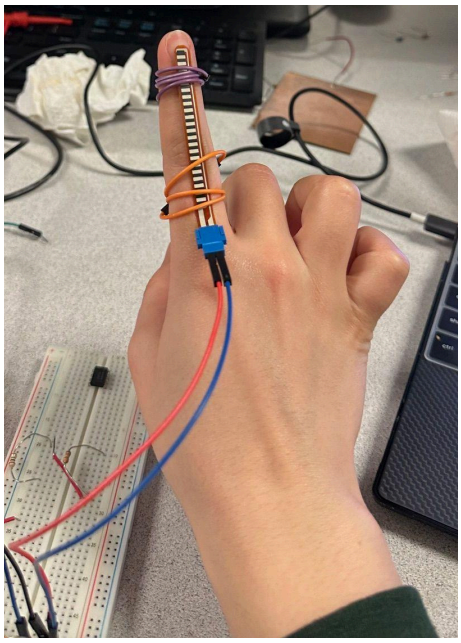


Figure 12.1 - Initial Flex Sensor Prototype

This initial prototype was implemented with only one jumper cable per terminal connected to the breadboard, this became important later on.

The final prototype, Flex Sensor Two-Pad Morse Keyer, required more jumper wires between the breadboard for more mobility. We noticed when testing that once in a while we'd get readings from the sensor that were in the double-digits. After evaluating, we found that the problems were that the jumper wires would move within their connections and become loose. That's why in Figure 5.2, there's additional tape to maintain a stable connection.

A user model error found with the flex sensors is one's ability to bend their thumb repeatedly. This applies to the new letter in Level 2 "I", which requires two dits or two thumb bends. It was a little difficult to play with the thresholds to lessen how much the thumb needs to bend while also considering the thumb bends in a relaxed state. So in testing, an experienced morse coder would get "I" incorrect sometimes when inputting it correctly.

4.2 CapTouch

To evaluate different high-copper pads, we compared the RC waveform and their difference in rise time between touched and untouched of a peso and FR4 board material. Both used a 10M ohm resistor and operated under 2.2 kHz. Below in Figure 13.1 and 13.2 are oscilloscope displays of the peso, and 13.3 and 13.4 are of the FR4.

Figure 13.1 - Untouched Peso

Figure 13.2 - Touched Peso

Figure 13.3 - Untouched FR4

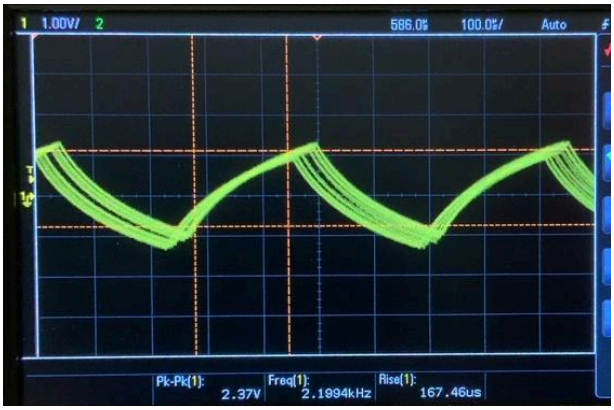


Figure 13.4 - Touched FR4

Both operated well with an RC delay. Below in Table 3 are the change in rise time between touched and untouched states of all homemade captouch sensors with the sensor inside a plastic sleeve (for paper) and without.

$\Delta\tau_r\ (us)$	Cover	No Cover
Peso	50	130
FR4	50	110

Table 3 - Change in Rise Time Data of Homemade CapTouch

The uncovered interfaces of both materials had a much greater change in rise time than with a plastic cover. The peso uncovered had a greatest change in rise time as seen in Table 3 and was part of the final prototype design.

Implementing the peso as our homemade capacitive touch sensor required a more nuanced approach than simply integrating it into the same circuit used in our lab demonstration. Initial experiments with this setup revealed a significant challenge: the oscillator's frequency would

occasionally come to a complete halt, rendering the system highly unreliable for transmitting Morse code. This instability can be attributed to the unique properties of the peso sensor. When a finger touches the peso, it effectively reduces the distance between the theoretical capacitor plates, leading to a pronounced increase in capacitance. Additionally, the larger surface area of the peso compared to the lab kit sensor further amplifies this effect. The combined impact of the reduced distance and increased surface area results in a much more substantial change in capacitance than that observed with commercially available sensors, leading to the observed fluctuations in the oscillator's frequency.

To mitigate this, we increased the magnitude of the capacitor in parallel with the peso until the change in frequency was no longer unstable. Ultimately, a 470pF capacitor with an output frequency of about 200 Hz gave us a reliable way to use the peso as a morse code keyer. The results of these adjustments are presented in Figures 13.1 and 13.2.

Figure 13.1 - CapTouch + Oscillator response to No Touch (left) vs Touch (right)

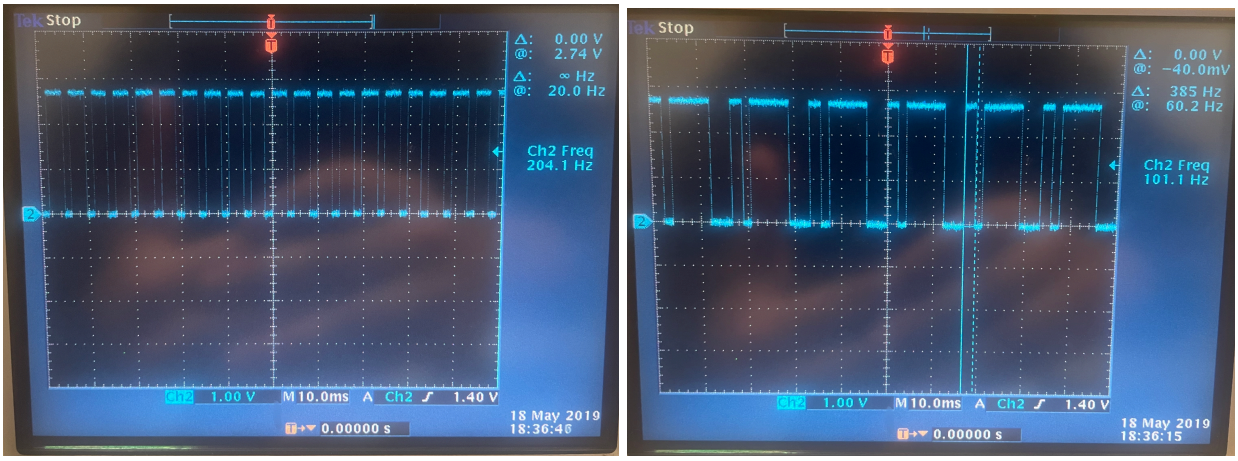


Figure 13.2 - Peso + Oscillator response to No Touch (left) v. Touch (right)

By using Equation [2.1] we can estimate the change in capacitance in the Peso sensor by rearranging and plugging in our measured values:

$$f = \frac{1.44}{(R_1+2R_2)*(C+C_{Sensor})} \Rightarrow C_{Sensor} = \frac{1.44}{(R_1+2R_2)*f} - C$$

$$C_{Sensor} = \frac{1.44}{(101.1\,Hz)*(4.7\,M\Omega+2*4.7M\Omega)} - 470\,pF \cong 5.41 * 10^{-10}[F]$$

The capacitance of the captouch was about **0.5 nF**. As anticipated, our change in capacitance is significantly greater. Surprisingly, the peso sensor turned out to be a much more reliable Morse code keyer than the sensor provided in our lab! We theorize that this reliability may stem from our adjustments to the oscillator, which resulted in more stable frequency changes. This stability made it simpler for the program measuring the changes to detect them, as they were more nuanced and consistent compared to the other sensor. Notably, the other sensor caused the oscillator's frequency to fluctuate by more than 1000 Hz, while the peso sensor only induced a change of about 100 Hz.

4.3 Game

Our game, inspired by the MorseMaria® app known for its Morse code instruction, was put to the test to evaluate its educational effectiveness. To compare learning outcomes, one team member utilized MorseMaria® while another engaged with our game. Both followed identical sequences of letters, focusing on the initial levels to assess knowledge retention. After a two-day break from both platforms, a test was conducted to evaluate their ability to accurately identify the Morse code sequences for the letters [E, T, A, N, I, M], reflecting three levels of learning. Remarkably, both individuals successfully recalled the correct sequences for all the letters, demonstrating that our game is on par with established learning tools in teaching Morse code, providing comprehensive practice even for novices.

5.0 Discussion and Conclusion

The major changes of the project between the proposal and implementation was the deliverables of measured data. In the proposal, parameters were thought out as much as possible, but as we experimented with the sensors and the code, the performance metrics changed. A parameter that was changed was comparing the percentage of error between the Peso and 167 CapTouch sensor. There was no qualitative error when testing than user model error consideration. With the custom circuity of each captouch sensor, the qualitative measurement was that the Peso was easier to use than the 167 sensor, discussed in *Evaluation.* Another measurement type that changed was going through all letters of the alphabet and evaluating the percentage error of each sensor; we did not have time to do that. From testing by an experienced morse coder Levels 1 - 3, a total of 6 different letters with a max size of 2, there were some misreadings with the flex sensors mentioned in *Evaluation*. Two of the measurements changed were the testing of the code and sensors with varying speeds as a percentage error. With the user model errors in mind, the

sensors worked the same between an inexperienced/slower coder and experienced/faster coder. Qualitative measurement added was realizing that an inexperienced morse coder, Jacob, actually learned from the game. Jacob was able to go through the first 3 levels and verbally tested the days after of the 6 letters he learned; he got them all correct.

In conclusion, the project "Dits and Dahs" successfully developed an interactive Morse code training game that effectively utilized various sensors to enhance learning through practical engagement. The team explored the functionalities of capacitive touch and flex sensors, achieving a balance between sensitivity and accuracy essential for accurate Morse code translation. Through iterative testing and evaluation, the project identified and addressed the challenges of sensor responsiveness and reliability, leading to a refined user experience.

The capacitive touch sensors, both commercial and homemade, were meticulously adjusted to optimize their response for Morse code inputs, revealing the intricacies of sensor behavior and the importance of precise calibration. Meanwhile, the flex sensors demonstrated linear response characteristics, allowing for straightforward integration into the game's mechanics.

Shortcomings such as occasional sensor instability and the need for physical modification to enhance connectivity were overcome through methodical experimentation and design adjustments. Future enhancements could focus on further refining sensor sensitivity, exploring alternative materials for improved performance, and expanding the game's complexity to cater to advanced users.

Overall, the project not only met its initial goals but also provided valuable insights into sensor integration and application in educational tools, highlighting the team's adaptability and problem-solving skills in the face of technical challenges.