Andy Campanella
Carlos Vila-Virella
6.111 Final Project Report

TextWand Interactive LED Display

Abstract

The TextWand is an innovative text messaging system. A ten button, full alphanumeric interface allows the user to enter a message using a method similar to the text messaging input on a standard cellular phone. A message that is entered is stored into an input buffer and later gets loaded to the output device. The output consists of a single row of seven LEDs, and to display the message a user waves his hand through the air. Each character is broken down into five lines of seven points, and the output LEDs are flashed at the correct rate and in the correct sequence to display the text message.

Overview (Andy Campanella)

The TextWand is an interactive text messaging device that was intended to be wearable and completely self-contained. For the purposes of this project, however, the input device was configured using ten pushbutton switches and was implemented on the labkit. The device features a cellphone-style input mode, with eight keys capable of inputting the standard 24 letters (most phones cannot represent the letters Q and Z) as well as all of the numerals. A user can enter up to eight characters to display at a time, which get loaded into an SRAM input buffer and are later sent to the output device. Characters in the buffer are then looked up in an output display ROM, which breaks each letter into a five by seven matrix of points. Each displayed character then consists of five lines, which are strobed on the output wand at the correct rate to display the message as a user waves his hand through the air.

Input Device Description (Andy Campanella)

The block diagram for the input portion of the project is shown in Figure 1. It consists of four major blocks- a Keypad FSM, a main Input FSM, a Timer, and finally an SRAM Control FSM. The input FSM is enabled and reset from a Main FSM, which simply allows us to toggle between display and input modes. The Input FSM controls the overall input process, and its functionality will be explained shortly.

The ten input buttons roughly correspond to the keys on a cell phone. The first button controls the space and backspace, while keys two through ten are each associated with three characters and a single number. Finally, a separate enter key allows the user to complete the input process and begin displaying the entered message.

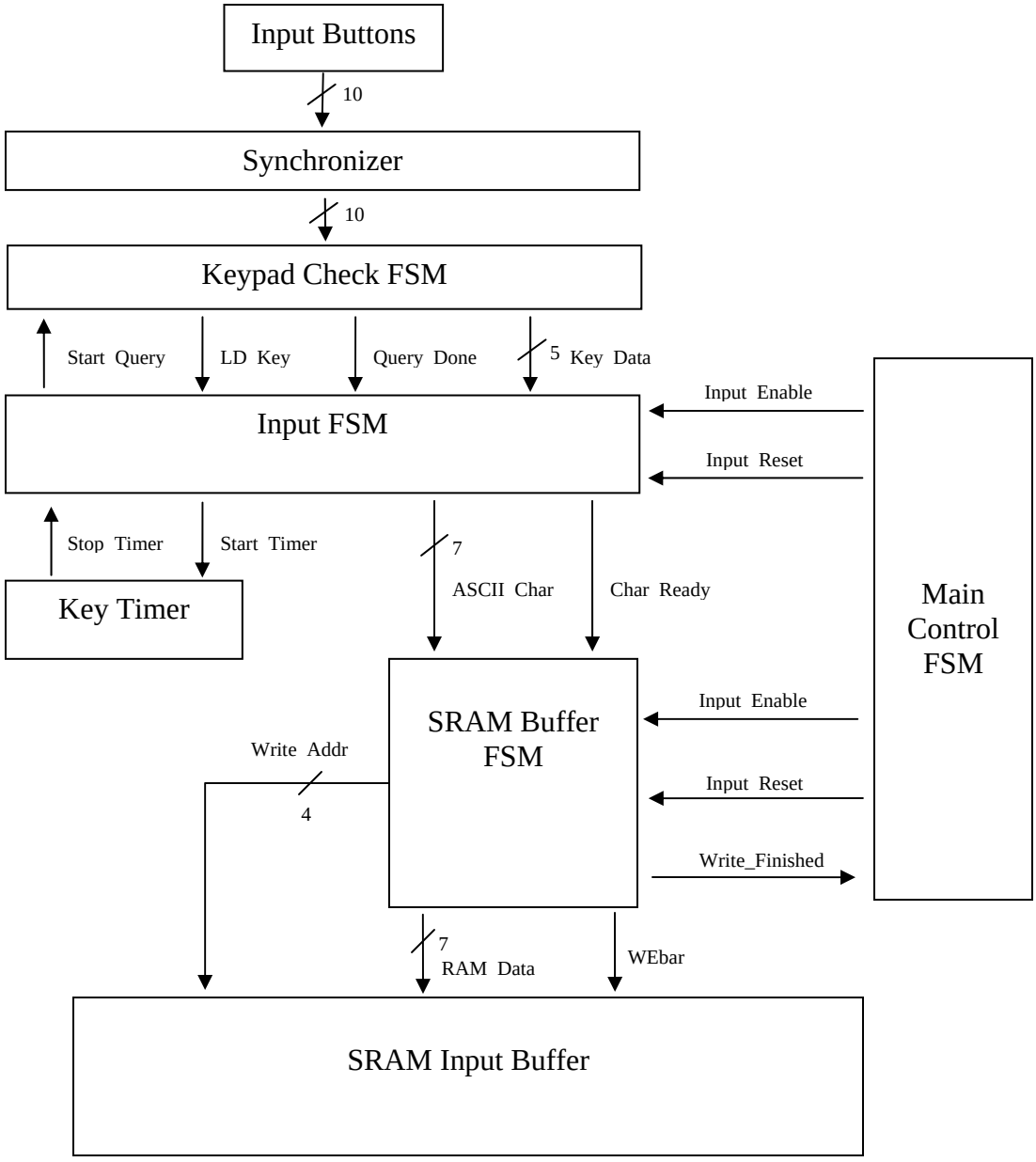

Figure 1: Block Diagram for the Input Portion

As illustrated in Figure 1, the Keypad FSM takes a single input telling it to begin a query of the keypad, and outputs two signals signifying the end of the query and a single five bit output with the data about which button was pressed. The purpose of the Keypad FSM is to decode the ten input signals to the appropriate key data, and also to stall the Input FSM until the user takes his finger off a an input button. The Input FSM

will continuously loop around and query the keypad for input data, which is supplied by the Keypad FSM. The state transition diagram for this FSM is shown in Figure 2.

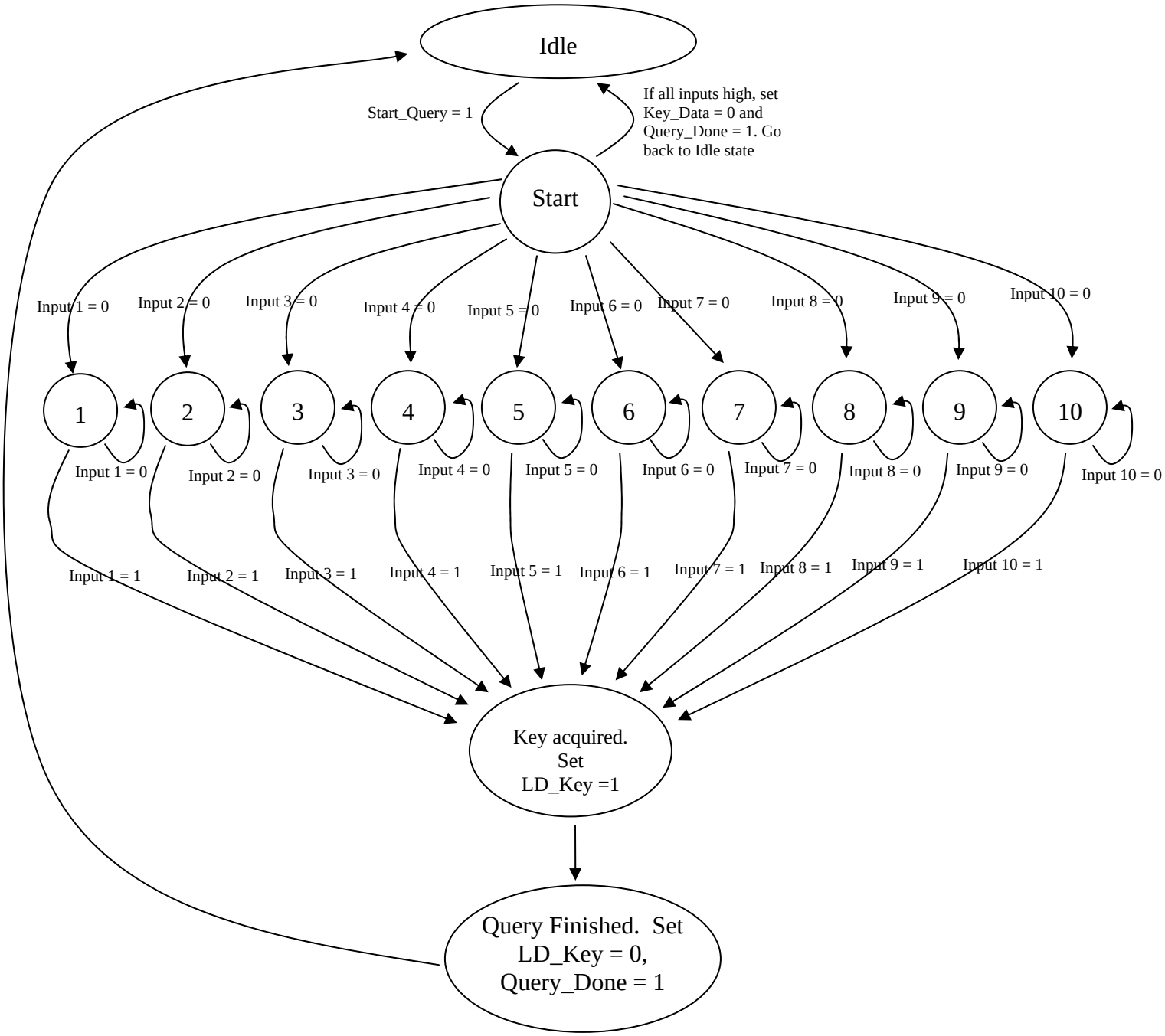

Figure 2: State Transition Diagram for the Keypad FSM

The Keypad FSM begins in a default Idle state, where it waits for a start signal from the Input FSM. When this signal arrives, the FSM moves to the start state, and from there transitions to the appropriate state for each input button. If no key is being pressed when the Keypad FSM enters this state, it returns to the Idle state and signals the Input FSM that no button is being pressed by setting the Key_Data output to zero, and the Input FSM in turn immediately loops back and re-queries the keypad. If a key is being pushed, the FSM transitions to the appropriate state and sets the Key_Data to the correct number, and then waits until the button is released before continuing. After the user lets go of a key, the Keypad FSM signals that the key data is valid with the LD_Key output and then signals that the query is complete. These signals are then used by the Input FSM to decode which character the user means to input, and also signals that the SRAM can be written with valid data.

The state transition diagram for the Input FSM is located at the end of this report. It appears overwhelmingly complex at first sight, but actually operates in a relatively straightforward manner. The FSM interfaces with both the Keypad FSM and a Timer module, which outputs a 1 Hz signal. If a user presses a key, he has one second to press the same key again and access the second character associated with that key, so the Timer provides the input signal for this type of operation.

From the input state, the timer is started and the FSM begins to loop though the first query cycle to look for user inputs. When a key is pressed, the FSM proceeds to the next state and restarts the key timer- giving the user one second from the last key press to hit the same key again. If a second key is pressed within one second of the first, the FSM stores that key data into a second register and proceeds to the first comparison state. If

the data from the two keys is different, the FSM outputs the first ASCII character associated with the first key, then copies the contents of the second register (the second key pressed) into the first and loops back to the Input #2 state to wait for a third input. In this manner, the FSM can keep track of the number of times a particular key was pressed so it always outputs the correct ASCII code for that key.

If the two keys were the same, the FSM stores the second ASCII character associated with that key to a temporary register and proceeds to the third query cycle to wait for another input. If the third key is pressed within one second I compare that key to the first two, otherwise I simply output the second ASCII character associated with the original key and go back to the start. If the third key is different from the first two, I copy the temporary register (which contains the second ASCII character for the first key) to the output and swap the two registers to look for another input. If the third key is the same as the first two, the temporary register gets re-written with the third ASCII character for that key and we proceed to wait for a fourth input. In this manner, the FSM always knows how many times an individual button has been pressed and always outputs the correct data. A user can press each button up to four times to get unique data, the first three times being the three letters associated with that key and the fourth being the number assigned to it.

The Input FSM outputs a seven bit ASCII character code, as well as a signal to tell the Buffer FSM that new data is available to be written into the SRAM. The state transition diagram for the Buffer FSM is also included at the end of this report. The purpose of this state machine is to control the SRAM and ensure that the correct data is written into sequential addresses. This FSM looks at the ASCII character code output by

the Input FSM and either writes that character into memory, or, in the case of an enter or backspace, takes the correct action. The Buffer FSM is triggered from the Char_Ready signal, and when that signal is received (except in the case of an enter or backspace) it sets the output data equal to the input ASCII character code and writes the data into an address, then increments the address and goes back to wait for another input.

When a backspace is received, I first check to see if the SRAM is at its first available address and if it is I ignore the input and go back to the start point. If the SRAM is not at address zero, I first set the output data to be written into the SRAM to zero, then decrement the address counter and write a null into the previous location. If the user presses the Enter key before he has input eight characters, I write zeros to all subsequent locations and signal the Main FSM that the user input is complete.

This concludes the discussion of the input portion of the project, I will now turn to my testing and debugging methods.

Testing and Debugging (Andy Campanella)

Since my portion of the project was broken down into a few relatively complex FSMs, I had to do a large number of simulations to ensure that the correct character codes were being output. Testing the input portion of this project required entering every possible combination of repeated inputs and delayed inputs (to test the timing functionality), so it was easier to do the simulations using the Altera platform since that program has a method of entering inputs that is easier and faster than in ModelSim. Once I was confident that the Input FSM worked I went back and verified the basic tests with ModelSim, but all of the extensive testing was done with Altera.

I got stuck at one point for several days because I could verify the FSM functioned correctly in simulation, but there was a problem with how the user input and output ports were defined in the labkit.v file. I thought my code didn't work for several days when in fact it was fine, so it was an incredibly frustrating process. The problem with the new labkit and the Xilinx platform is that the code almost always compiles even if there are syntax or other simple errors with the way output ports are defined, it makes debugging quite a bit more difficult that with the old labkit. The labkit.v file also always generates a huge number of warnings because of the large number of unused inputs and outputs, so it was even more difficult to sort through the warning messages to debug.

I ultimately solved the problem (although it took two days of me debugging and almost two hours of sitting there with Prof. Chandrakasan before we figured out what was being declared incorrectly), and was able to demonstrate a correctly functioning input system. Figuring out the Input FSM to emulate a cell phone style input system was a challenging and interesting task. Even though we were ultimately unable to integrate both parts of the TextWand project, for me it was a rewarding exercise in designing a very complex FSM and several simpler modular state machines. This concludes my section of the report, and Carlos will now describe the output device.

Load Stage

The load stage takes the data stored in the Message Ram buffer, converts the Ascii characters into the bits corresponding to the LED output using the character ROM. This ROM had to be manually programmed with the bit map image of letters which I drew up on graph paper. The LoadMSG FSM waits for an enable signal from the Main FSM

letting it know to begin mapping the Buffer SRAM into the LED ram. This is a very straight forwards process where the FSM grabs the first character in the buffer RAM and looks up the LED output value in the Character ROM. The data from the ROM is then written sequentially into the LED SRAM. This process is repeated for the eight characters in the Message RAM buffer until the complete message has been loaded into the LED SRAM. Once the sequence is complete, the Load MSG FSM gives the Main FSM a load complete signal, informing the Main FSM that the message is ready to be outputted to the LEDs.

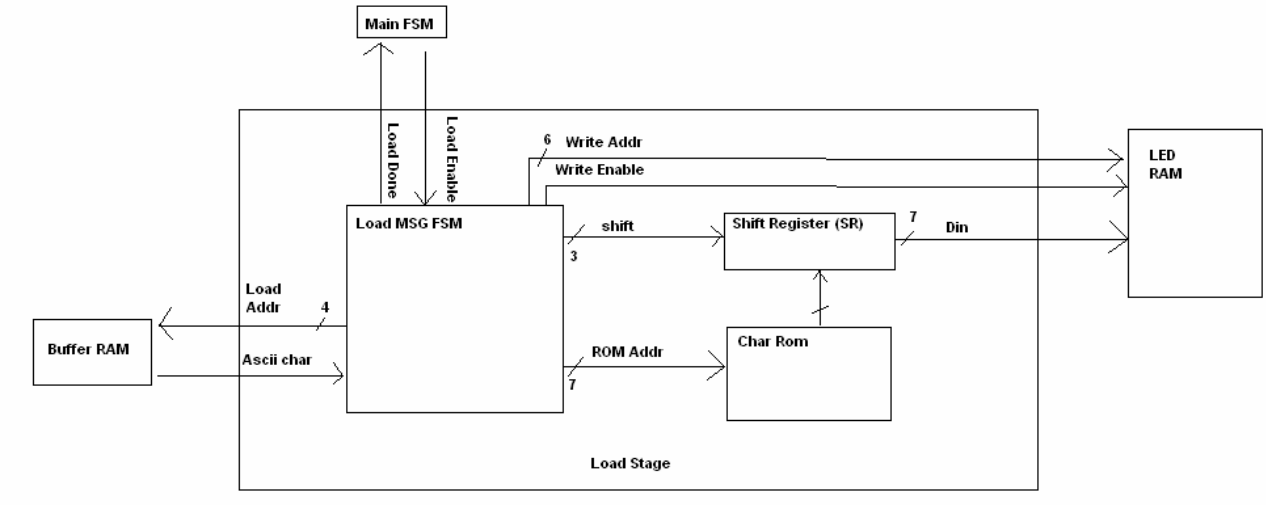

Figure 3: Load Message Stage

Output stage

The Output stage is responsible for flashing the LEDs with the proper timing in order to sweep out a legible message. One a message has been loaded into the LED SRAM, the main FSM provides the Output stage with an enable signal. Once this happens the Output stages reads the data in the first LED SRAM address and outputs it for a period of time set by the Timer Module. The Leds are then turned off and the next LED address is

looked up to be outputted. Once the Timer module gives the signal, the next line is outputted. This sequence continues until every line in the LED RAM has been displayed. Then the Back timer is activated to count up until the wand changes directions, which is indicated by a left or right accelerometer activated switch on the wand itself. The Back Timer then counts down the same amount of time counted up before starting to display the entire message again in the reverse direction. This is simply done by reading the LED Ram addresses from the bottom up. The Back Timer assumes that the sweep speed of the wand is constant and attempts to display the message in the same location in space.

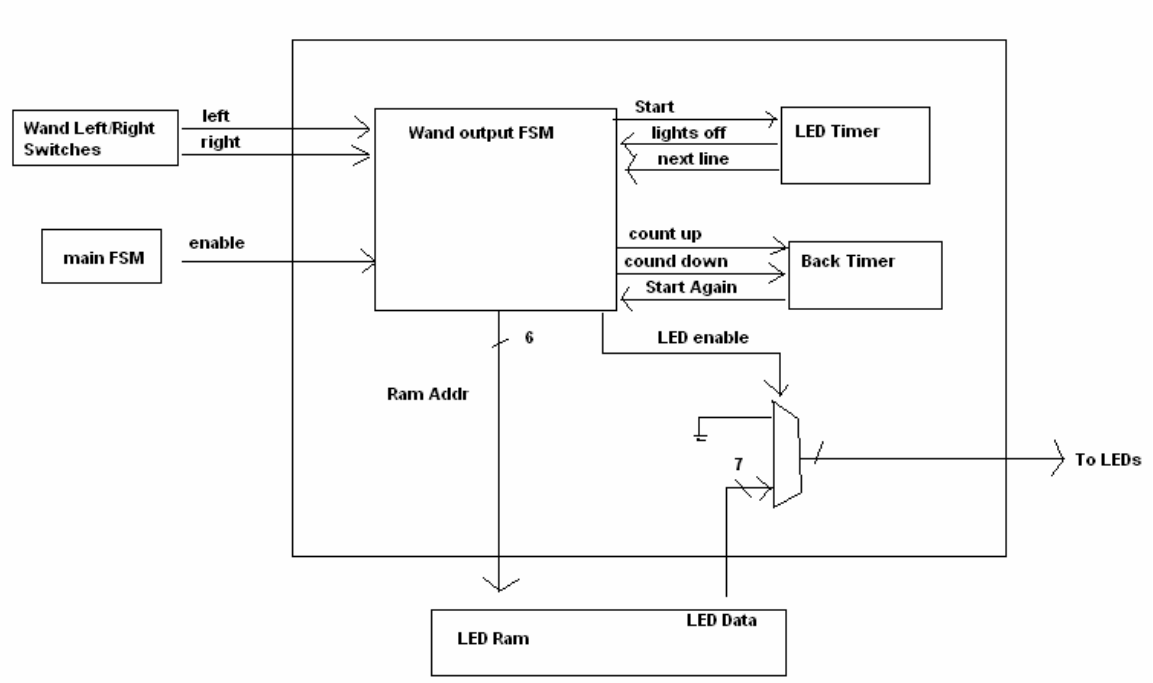

Figure 4: Output Stage

Testing

The testing phases of this project were very important in determining the timing specifications for flashing the LEDs. This was done empirically by loading a test

message, setting up timing specs, and waving the wand to try to view a message. This stage of testing took a long time to complete because of difficulties triggering the message to start at the right time. Unfortunately the accelerometer we first tried to use to trigger the start of the message was very poor and was not adequate for this purpose. For this reason I decided that a switch letting the device know when my hand had finished sweeping in one direction was a better solution. To implement this switch, I put a penny on the end of a stiff wire which could move slightly back and forth and make contact with another wire on either side. The stiff wire served as a spring and the penny as a mass for this makeshift accelerometer. This method of triggering turned out to be much more effective than the original accelerometer we got.

Conclusions

We were pleased with outcome of our project even thought we had a little bit of difficulty in the final integration stages of the project. This experienced served as a proof of concept for the originally intended wearable device. For a future implementation I recommend spending a lot more time with a good accelerometer in order to find the exact position in space of the wand. This would allow the wand to be triggered consistently in the same position in space making the message much more stable and easy to read. Overall working on this project provided a invaluable practice and understanding of the design and implementation process involved in coming up with a complex design product.