# Morse Code Translator

DESIGN DOCUMENTATION

*Robert McKay*

*Xuecong Fan*

December 8, 2017

ECE 3710

*Utah State University*

# Contents

# List of Figures

# 1 Introduction

The Morse Code Translator is a device that translates dots and dashes to plain English. The user inputs Morse code with the button. When the button is pressed, a green LED turns on, and a speaker buzzes. The Morse Code Translator outputs an ASCII translation to the Morse code input. The results are displayed on a PuTTY terminal. Using screen sharing on Skype, a transmitter can send a Morse code translation wirelessly.

# 2 Scope

The purpose of this document is to summarize the design and operation of the Morse Code Translator. Requirements, dependencies, design, implementation, and testing of the Morse Code Translator are included in this document.

# 3 Design Overview

This section will describe engineering requirements and hardware dependencies of the Morse Code Translator.

## 3.1 Requirements

The Morse Code Translator was tested for the following requirements:

1. The Morse Code Translator shall convert each letter of the Morse alphabet (including each digit, 0 through 9) to its ASCII equivalent and display the result on PuTTY
2. The Morse Code Translator shall output a space after initialization
3. The Morse Code Translator shall output a space for a completely debounced input
4. The Morse Code Translator shall beep when the button is pressed
5. The Morse Code Translator shall turn on an LED when the button is pressed
6. The Morse Code Translator shall work wirelessly by using Skype

## 3.2 Dependencies

The Morse Code Translator required the following hardware:

1. TM4C123gh6pm microcontroller
2. Wires
3. A breadboard or PCB
4. A 3.3 volt DC power supply

5. A 16 MHz crystal oscillator
6. An LED
7. A general-purpose diode
8. A speaker
9. Two computers with Skype and PuTTY installed

## 3.3 Theory of Operation

This section will include a high-level overview of how the Morse Code Translator works as well as a brief description of how to use the Morse Code Translator.

### 3.3.1 How the Morse Code Translator Works

The Morse Code Translator uses a double-edge triggered interrupt to determine whether the last input was a press or a pause. A general-purpose up-count timer is checked and reset on each button press and release to determine the length of the press or pause. A binary tree is created in the initialization that stores ASCII characters. A pointer to the binary tree takes the left path if the input was a dot and the right path if the input was a dash. When there is a pause of sufficient length, the data of the node to which the pointer points is output through PuTTY, and the pointer is moved back to the root of the tree.

### 3.3.2 How to Use the Morse Code Translator

To use The Morse Code Translator, open a PuTTY terminal. Configure the terminal for a baud rate of 9600. Use the COM corresponding to the TM4C123gh6pm. After the terminal window is open, reset the microcontroller. The Morse Code Translator requires a moment to initialize after startup or resetting the microcontroller. During this time, the button should not be pressed. After initialization, a space will be output to PuTTY, alerting the user that the Morse Code Translator awaits input.

For a wireless transmission, open Skype and make a call to the desired receiver. When the call is answered, select screen sharing from your Skype menu, then make the PuTTY terminal visible. This will allow the desired recipient to see the Morse code translation as the Morse code is input.

# 4 Design Details

## 4.1 Software

Morse code consists of dots, dashes, and pauses. The length of an ideal dash is the length of three dots. The pause between parts of a letter is, ideally, the length of a dot. The pause between letters in a word should be the length of three dots, and the pause between words should be seven dots in length. The Morse Code Translator uses 200 ms as the length of a dot. The length of a dot will be referred to interchangeably as an element and a dot length

for the remainder of this document.

The Morse Code Translator interprets any press length from 0.5 to 1.5 elements as a dot and any press length greater than 1.5 elements as a dash. The purpose of the lower limit on the dot is debouncing. 1.5 elements was chosen to be the border between a dot and a dash because 1.5 is a 50% a dot length longer than a dot and 50% a dash length shorter than the length of a dash. For the same reason, 0.5 elements was chosen to be the lower bound of a dot.

In a similar way, the Morse Code Translator interprets pauses. Any pause length shorter than 1.5 elements is considered a space between characters inside a letter. The pause is debounced by the dot debounce software. This allows more than 50% error for a dot-length pause. A pause length between 1.5 elements and 4.5 elements is considered a space between letters in a word. The value of 4.5 elements was chosen as the upper bound because it is 50% greater than the ideal length of a pause between letters in a word. The length of 1.5 elements was chosen to be the border between a dot length pause and a dash length pause for the same reason 1.5 elements was chosen to be the border between a dot length and a dash length.

Starting near the end of the initialization, an up-count timer records the time since the last button press or release. On a button press or release, the timer value is read into a variable called timerValue. Then, the timer is reset, and appropriate action is taken based on the value of timerValue. The timer counts up to a length of 4.5 elements, the longest length of a pause between letters of a word. If the timer reaches 4.5 elements, the Morse Code Translator outputs the current letter, followed by a space, then turns off the timer to save power. When the user begins inputting again, the timer is turned on.

A binary tree is used to decode the Morse code input. The binary tree is constructed in the initialization with a struct called a node. A node holds a letter and a left and right pointer. A pointer called toLetter traverses the tree as the general-purpose input and output (GPIO) port B interrupt is called and proper timing conditions are met. When it is time to output a letter, the data from the node to which toLetter points is read. See Appendix A for the binary tree.

See Figure 1, Figure 2, and Figure 3 for flowcharts of the main function, the timer interrupt, and the GPIO interrupt, respectively.

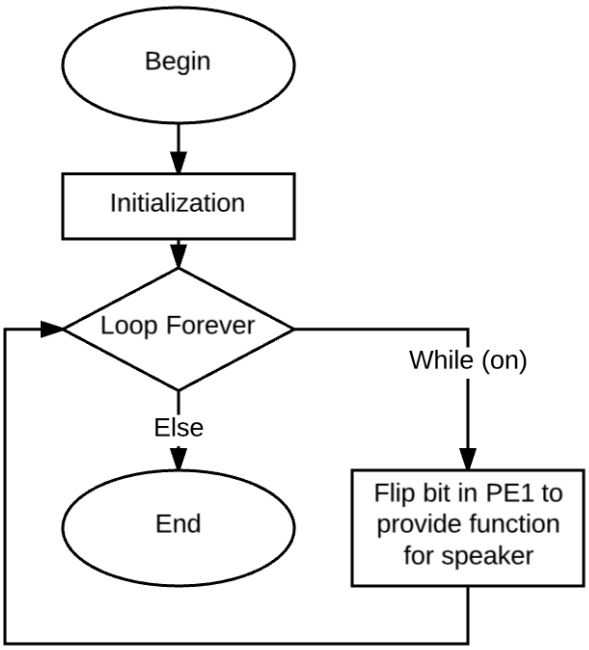



**Figure 1:** Flowchart of the main function

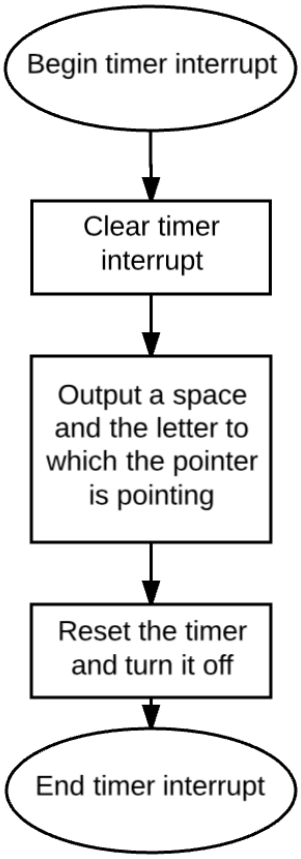



**Figure 2:** Flowchart of the timer interrupt

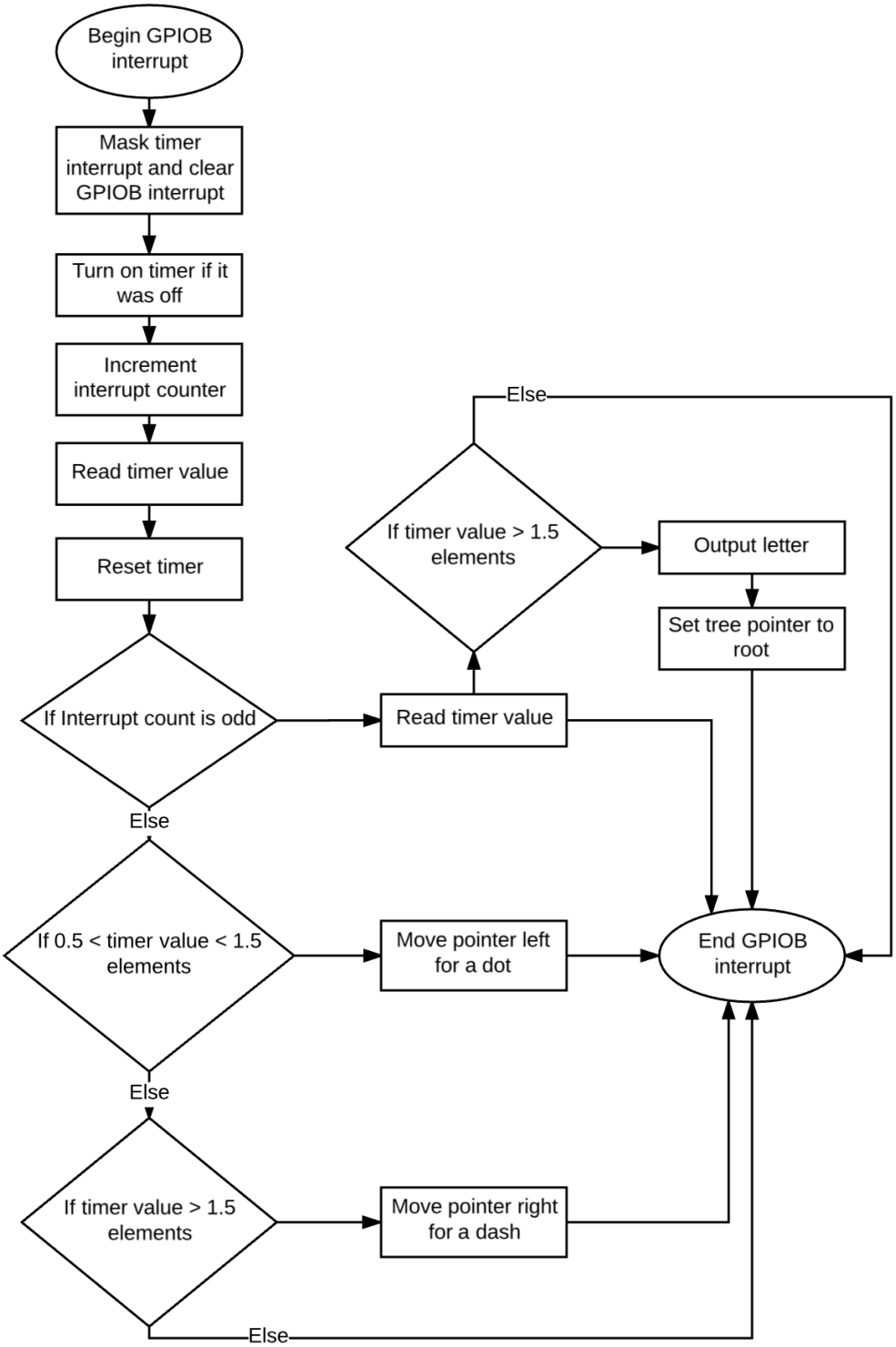



**Figure 3:** Flowchart of the GPIO interrupt

All source code may be viewed in Appendix B.

## 4.2 Hardware

The initialization process includes initializing GPIO pins PB4, PA1, and PE1. PA1 is configured for its alternate function, universal asynchronous receive and transmit (UART), while

PB4 is configured to be input and PE1 is configured to be output.

User input is received from the button on the Morse Code Translator. The button is wired to be active-high. PB4 is wired on the low side of the button. When the button is pressed, PB4 goes high; when the button is released, PB4 goes low. The same is true for the turning on of the LED and speaker. A diode placed between VIN of the speaker and the button serves to prevent current from flowing out of VIN and into PB4, which would disturb the logic of the Morse Code Translator. The diode also absorbs some of the current, making the speaker less blaring and more pleasant than it would be running with full input power.

PE1 is wired to the analog input of the speaker to provide a frequency for output. Despite the low-resolution quality of a square wave compared to the ideal sinusoidal wave, the signal is sufficient to cause the speaker to create sound.

A hardware schematic may be viewed in Figure 4.

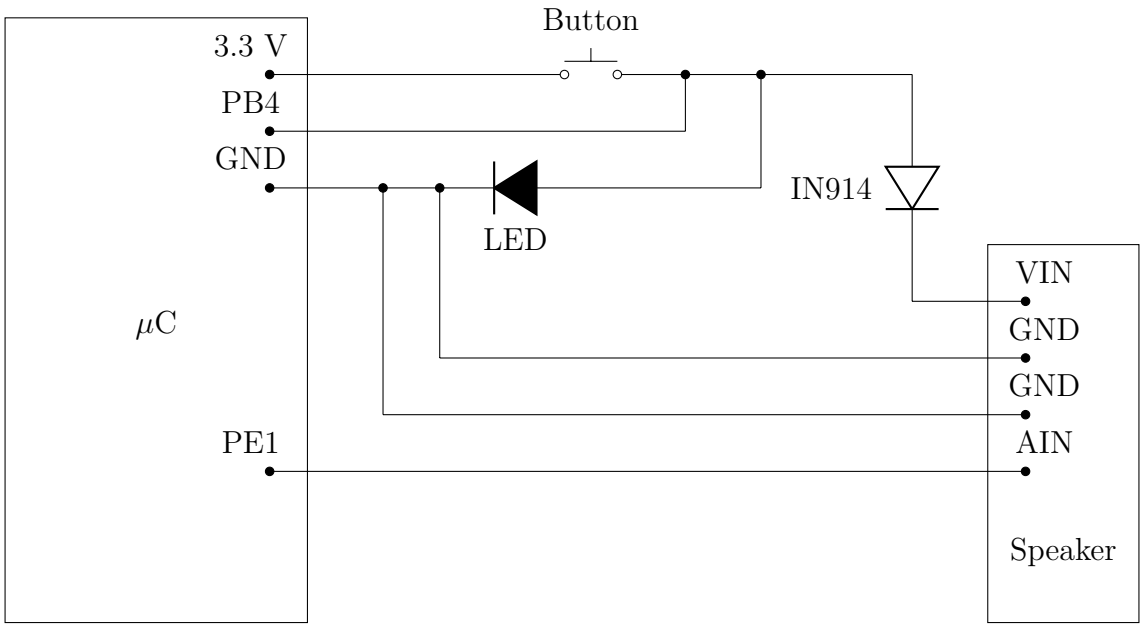



**Figure 4:** Schematic

# 5 Testing

This section includes a test for each requirement listed in Section 3.1. The tests are listed in the same order discussed in Section 3.1, so test 1 will correspond to requirement 1, and so forth.

## 5.1 Alphanumeric Verification

To test requirement 1, Morse Code was input for each of the 36 alphanumeric characters. Enough time was left between letters to output a space between each letter. A screenshot of the PuTTY terminal in Figure 5 demonstrates this.


**Figure 5:** Requirement 1 verification

As additional proof, a letter to which the toLetter pointer pointed in the debugger (the letter 'o') is shown in Figure 6. Figure 6 is a cropped screenshot taken while the program was running in the debugger.

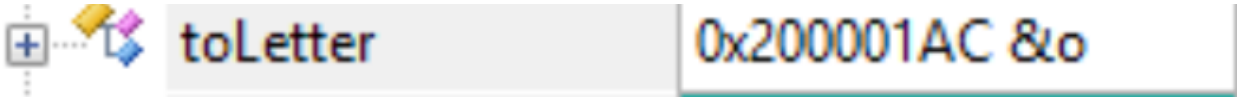



**Figure 6:** Additional debugger verification

## 5.2 Initialization Verification

To test requirement 2, a new PuTTY terminal was opened, and the board was reset. As expected, a space was output, signaling complete initialization. The result may be viewed in Figure 7.

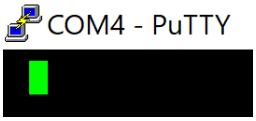



**Figure 7:** Initialization verification

## 5.3 Debounced Output Verification

To test requirement 3, the button on the Morse Code Translator was intentionally tapped and released too quickly. The first space is from the initialization, the second is from the debounced output. See Figure 8 for verification.

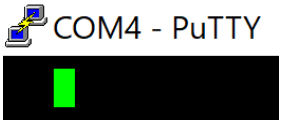



**Figure 8:** Debounced output verification

## 5.4 LED Verification

To test requirement 4, the button was pressed, and the LED turned on. Verification of requirement 5 may be viewed in Figure 9.

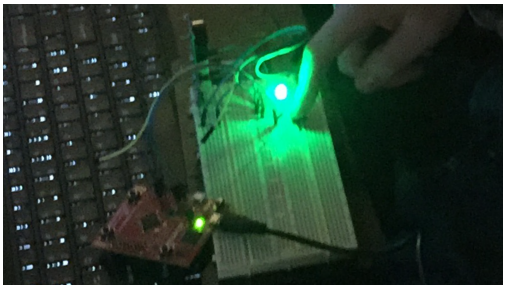


**Figure 9:** LED verification

## 5.5 Speaker Verification

To test requirement 5, the button was pressed, and the speaker buzzed. Listen to the video accompanying this document to hear the verification of requirement 6.

## 5.6 Wireless verification

To test requirement 6, a Skype screen share was performed. As expected, the PuTTY terminal displayed on the receiving computer. A photo of the PuTTY screen on the transmitting computer is shown in Figure 10. Figure 11 shows a photo of the screen of the receiving computer.

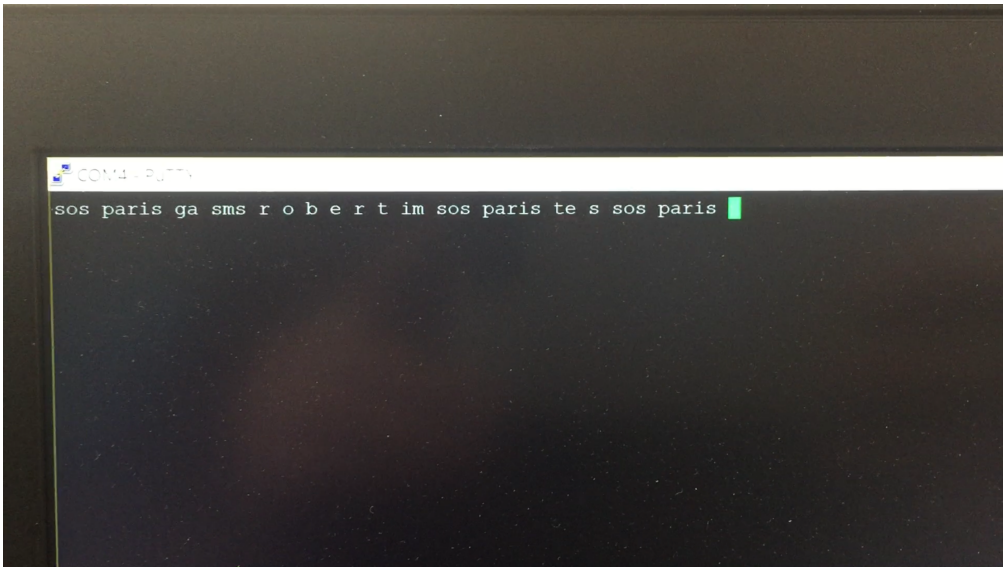



**Figure 10:** Transmitter verification

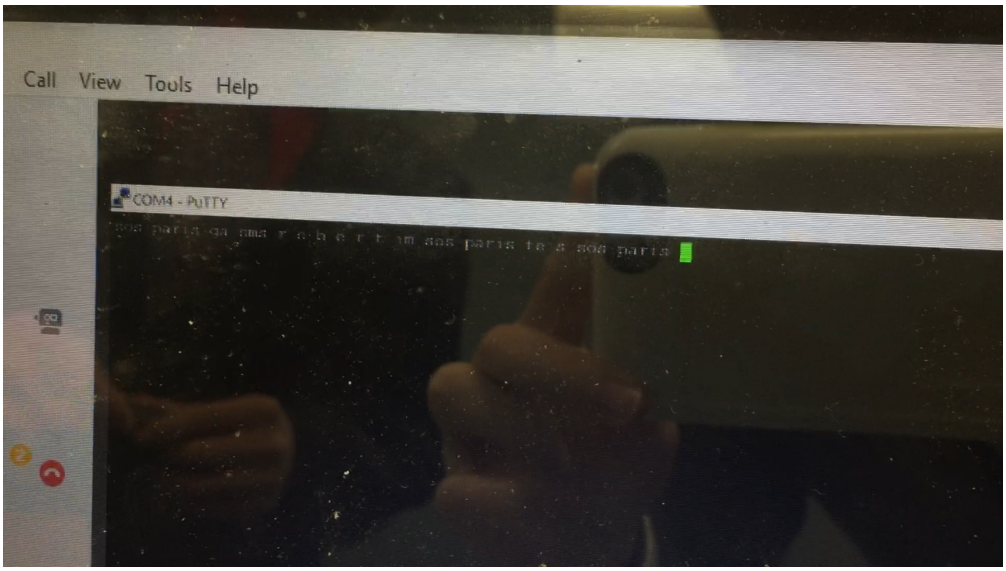


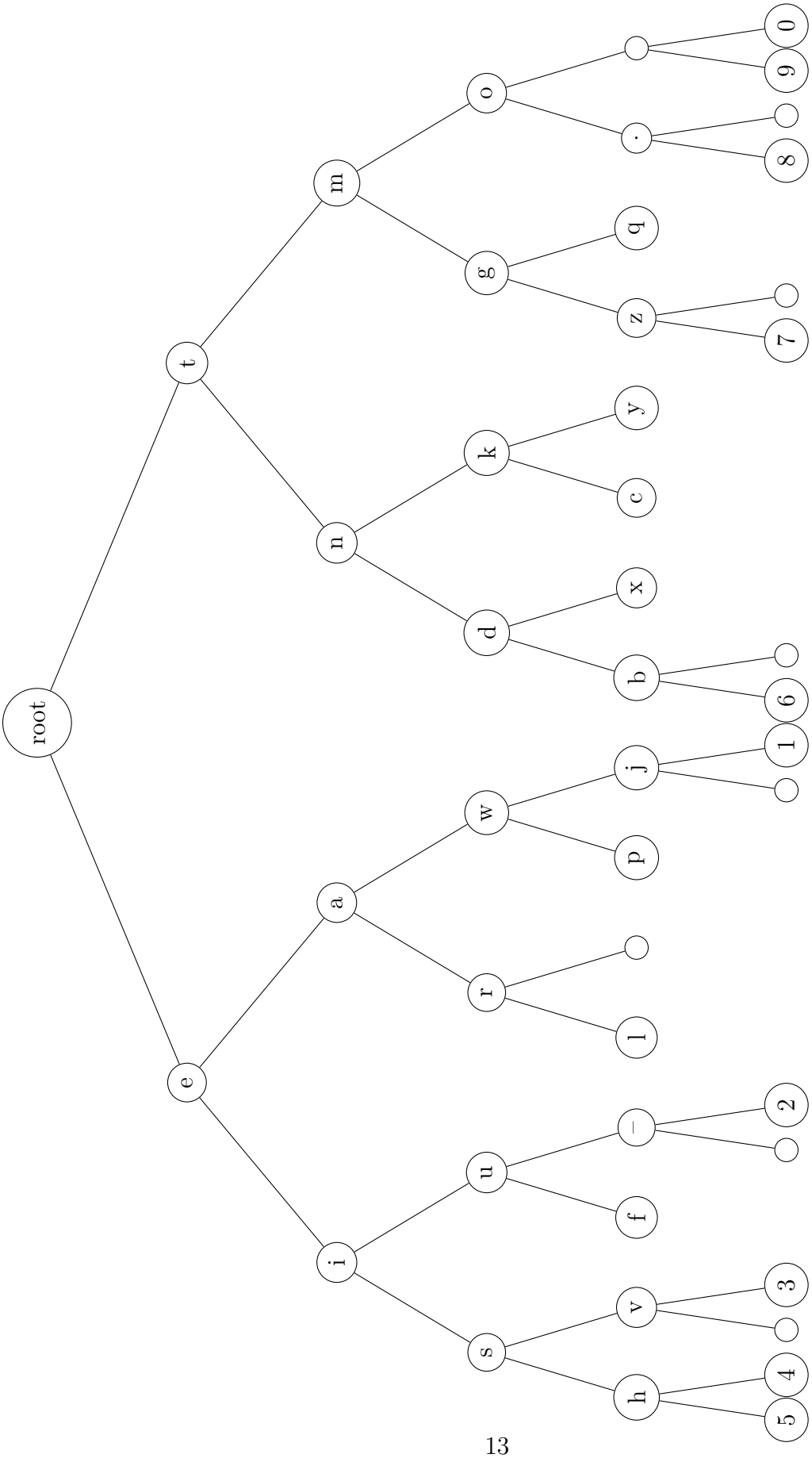

**Figure 11:** Receiver verification

# 6 Conclusion

The test results demonstrate the functionality of the Morse Code Translator. The Morse Code Translator successfully outputs all alphanumeric characters, as shown in Section 5.1. The Morse Code Translator alerts the user when it is initialized by outputting a space, as demonstrated in Section 5.2. The test in Section 5.3 demonstrated a correct input debounce. In the test in Section 5.4, the LED glowed brightly when the button was pressed. Section 5.5 referenced the video accompanying this document for verification of the speaker's buzzing when the button is pressed. Section 6 showed photos of a successful Skype screen share, demonstrating the wireless capability of the Morse Code Translator when combined with the correct hardware.

The Morse Code Translator met each requirement. The wide error range available to the user makes input simple, even for one learning Morse code. The binary tree in the code allows the Morse Code Translator to move quickly and efficiently through each character in the tree. The Morse Code Translator is a powerful and simple tool that performs everything it was designed to do.

# 7 Appendix A: Binary Tree



# 8 Appendix B: Code

```
unsigned int* portA = (unsigned int*) 0x40004000;
unsigned int* portB = (unsigned int*) 0x40005000;
unsigned int* portE = (unsigned int*) 0x40024000;
unsigned int* systemClock = (unsigned int*) 0x400FE000;
unsigned int* ssi0 = (unsigned int*) 0x40008000;
unsigned int* uart0 = (unsigned int*) 0x4000C000;
unsigned int* timer0 = (unsigned int*) 0x40030000;
unsigned int* corePeripheral = (unsigned int*) 0xE000E000;
unsigned int timerValue;
unsigned int interruptCounter = 0;
unsigned int adjustSpeakerFrequency;

/*
The following code block
creates the binary search
tree that will be used for
translation. Left for dot,
right for dash.
*/
struct node {
        unsigned char letter;
        struct node* left;
        struct node* right;
};

typedef struct node letter;

letter five = {'5', 0, 0};
letter four = {'4', 0, 0};
letter three = {'3', 0, 0};
letter two = {'2', 0, 0};
letter one = {'1', 0, 0};
letter six = {'6', 0, 0};
letter seven = {'7', 0, 0};
letter eight = {'8', 0, 0};
letter nine = {'9', 0, 0};
letter zero = {'0', 0, 0};
letter h = {'h', &five, &four};
letter v = {'v', 0, &three};
letter f = {'f', 0, 0};
letter hyphen = {'-', 0, &two};
letter l = {'l', 0, 0};
letter p = {'p', 0, 0};
letter j = {'j', 0, &one};
letter b = {'b', &six, 0};
```

```
letter x = {'x', 0, 0};
letter c = {'c', 0, 0};
letter y = {'y', 0, 0};
letter z = {'z', &seven, 0};
letter q = {'q', 0, 0};
letter period = {'.', &eight, 0};
letter nothing = {'~', &nine, &zero};
letter s = {'s', &h, &v};
letter u = {'u', &f, &hyphen};
letter r = {'r', &l, 0};
letter w = {'w', &p, &j};
letter d = {'d', &b, &x};
letter k = {'k', &c, &y};
letter g = {'g', &z, &q};
letter o = {'o', &period, &nothing};
letter i = {'i', &s, &u};
letter a = {'a', &r, &w};
letter n = {'n', &d, &k};
letter m = {'m', &g, &o};
letter e = {'e', &i, &a};
letter t = {'t', &n, &m};
letter root = {0, &e, &t};

// This pointer will traverse the tree.

letter *toLetter = &root;

void GPIOB_Handler(void) {

    // acknowledge interrupt
        portB[0x41C/4] = 0x10;

    // mask timer interrupt
        corePeripheral[0x100/4] &= 0xEFFFF;

    // Re-enable the timer if it was turned off
    // due to a period of inactivity
        timer0[0xC/4] = 0x1;

    // The interrupt counter is incremented to determine
    // whether the last timing was a press or release
        interruptCounter++;

    // look at the timer
        timerValue = timer0[0x50/4];

    // reset the timer
```

```
        timer0[0x50/4] = 0x0;

    //If this is true, the previous state was a wait
        if ((interruptCounter % 2) == 1) {

        // Wait time was longer than the minimum
        // space between letters, output
                if (timerValue >= 4800000) {  the letter
                        uart0[0x0] = toLetter->letter;

            // Reset toLetter to start the
            // translation of the next letter
                        toLetter = &root;
                }
        }

    // Wait time was a dot, go left. Also
    // debounces with the minimum value.
        else if (timerValue <= 4800000 && timerValue >= 1600000) {
                if (toLetter->left != 0) {
                        toLetter = toLetter->left;
                }
        }

    // Wait time was longer than the maximum time for a dot, go
       right
        else if (timerValue > 4800000) {
                if (toLetter->right != 0) {
                        toLetter = toLetter->right;
                }
        }
        corePeripheral[0x100/4] |= 0x80000; // unmask timer
           interrupt
}

void TIMER0A_Handler(void) {
        timer0[0x24/4] = 0x1; // clear timer interrupt
        uart0[0x0] = toLetter->letter;
        toLetter = &root;
        uart0[0x0] = ' ';
        timer0[0x50/4] = 0x0;
        timer0[0xC/4] = 0x0;
}

int main(void)
{
        systemClock[0x618/4] |= 0x1; //turn on UART0
```

```
systemClock[0x608/4] |= 0x13; // Enable Port A, B, and E
systemClock[0x604/4] |= 0x1; // Enable GPTM 0A

portE[0x400/4] = 0xFF; // output on port E
portE[0x420/4] = 0x0; // disable alternate functions on
    port E
portE[0x51C/4] = 0xFF; // digital enable port E
portB[0x400/4] = 0xF; // output on PB0-PB3, input on PB4-
    PB7
portB[0x420/4] = 0x0; // disable alternate functions on
    port B
portB[0x51C/4] = 0xFF; // digital enable port B
portA[0x400/4] = 0x10; // 0000 0010, PA1 (UART0 Tx)
portA[0x420/4] = 0x10; // 0000 0010, enable alternate
    function on PA1
portA[0x51C/4] = 0x10; // Digital enable PA1

portB[0x404/4] = 0x0; // Edge-triggered interrupts on port
     B
portB[0x408/4] = 0x10; // Double-edge triggered interrupt
    on PB4
portB[0x410/4] = 0x10; // Unmask interrupt on PB4

uart0[0x30/4] = 0x0; //disables the UART before any of the
     control registers are reprogrammed
uart0[0x24/4] = 0x68; //Writes 104 to the integer part of
    UART0, for the baud rate divisor
uart0[0x28/4] = 0xB; //Writes 1/6 to the fractional baud
    rate divisor register of UART0
uart0[0x2C/4] = 0x70; //Tells the UARTLCRH (line control)
    we want 8 bits of data, and 1 stop bit
uart0[0xFC8/4] = 0x0; //Tells the UART communication clock
     to use the system clock
uart0[0x30/4] = 0x101; //enable UART0 with Transmit

timer0[0xC/4] = 0x0; // disable timer 0A in the control
    register for the timer
timer0[0x0/4] = 0x0; // configure timer 0A to be a normal
    timer
timer0[0x4/4] = 0x12; // configure timer 0A to count up
    and be periodic
timer0[0x28/4] = 14400000; // 200 ms * 16,000,000 * 4.5,
    one machine cycle more than the maximum length of a
    dash
timer0[0x38/4] = 0x0; // no prescale value
timer0[0x24/4] = 0x1; // clear timer 0A's interrupt, just
    in case a garbage value has set it high
```

```
        timer0[0x18/4] = 0x1; // Enable timer 0A's interrupt with
            the interrupt mask register
        corePeripheral[0x100/4] |= 0x80002; // Turn on pin 19 in
            NVIC to enable timer 0A interrupt, also GPIOB interrupt
             enable
        corePeripheral[0x410/4] |= 0x20000000; // make priority of
             timer interrupt lower
        // corePeripheral[0x100/4] |= 0x2;
        timer0[0xC/4] = 0x1;

        portE[0x3FC/4] = 0xFF;
        while(1) {
                for (adjustSpeakerFrequency = 0;
                    adjustSpeakerFrequency < 2500;
                    adjustSpeakerFrequency++) {

                }
                portE[0x8/4] = ~portE[0x8/4];
        }
}
```