Raspberry Pi - Morse Code Reader
Final Report

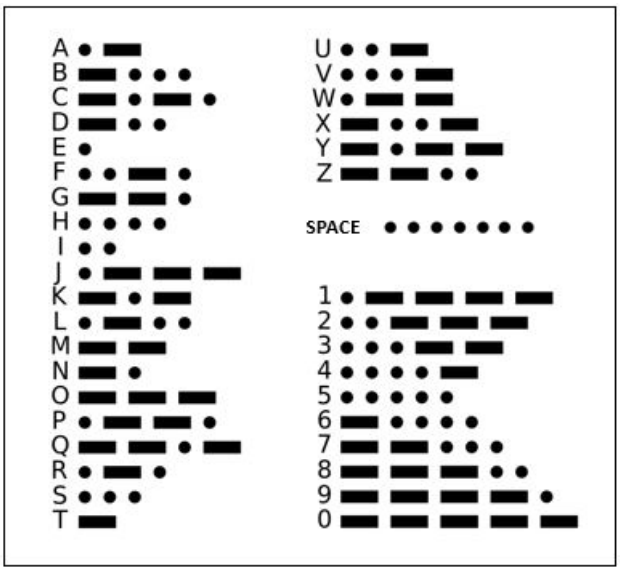
Author: Yashiv Fakir

Date: 20 January 2020

---

1 Introduction

This report offers a final analysis and summary of the Morse Code Reader Project. The report will cover the various input methods and the success rates. Then outline the C algorithm and further explain complicated functions. Ending with an outline of the challenges, changes and improvements made from the original design. Below is the alphanumeric symbol with its corresponding Morse Code pattern that the reader can interpret. (To expand the range of the reader the Morse code and alphanumeric array need simply be expanded with the new characters).



Please refer to Section 3 - Morse Code Input Methods to see the results of the various methods

2 Usage Requirements/Instructions

This section will outline how to use the reader and the format of the input message. Note, a great deal of effort was made to avoid the necessity of a calibrating Dash and Dot prior to reading a message however this was unavoidable to ensure accuracy. The dynamically calibrating algorithm, used initially had a very low accuracy rate. The design was also focused on minimising the amount of effort required by the user.

2.1 Usage Instructions

The entire program is meant to be executed in a terminal environment. To use the reader:

1. The user must first compile the intended code depending on the chosen input being the paper or LED method. Use the following commands to compile the two versions of code on a linux machine:

   ```
   $ gcc Paper_Input_Reader.c -lwiringPi -lpthread
   $ gcc LED_Input_Reader.c -lwiringPi -lpthread
   ```

   The two libraries at the end of the command may have to be downloaded prior to compiling the code. If the user is not using a Linux Distro, use the respective method for your current operating system.

2. Next is to run the built file in the command line. At that stage the user will see a set of instructions displayed in the terminal window.

3. Adhering to the instructions the user is to press the push button to begin reading followed by running the paper input under the LDR at a constant rate. The distance between the LDR and the paper message should be about 1[cm]. If the LED input is to be used, then the Morse code message is to be coded in C similar to the 'TEST' message (Shown in the code) prior to building. The LDR should also be about 1 [cm] from the LED displaying the message.

4. Once the message has been read in its entirety, then the user is to press the button for a second time to convert the encoded message. Then the calibrating parameters will be displayed followed by the converted message. Note, for the paper input when done, ensure that the white part of the paper remains under the LDR until the button is pressed again.

5. Finally, press crtl-z to terminate the program. NB: to decode another message the user would have to re-run the program.

2.2 Usage requirements

To ensure that the Morse Code reader works as intended for both methods a calibrating pattern is to be read first before the rest of the message is read. Below is an example of what the calibrating message should be:

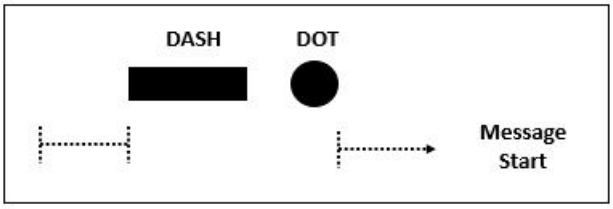


Figure 1: Calibrating Input Signal Example

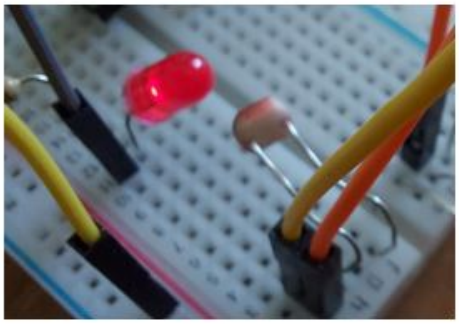
The pattern is to begin with a space (of any length on the LEFT), then a Dash, then a small space, then a dot and finally a large space. The remainder of the message is then to

follow the calibrating pattern on the RIGHT.

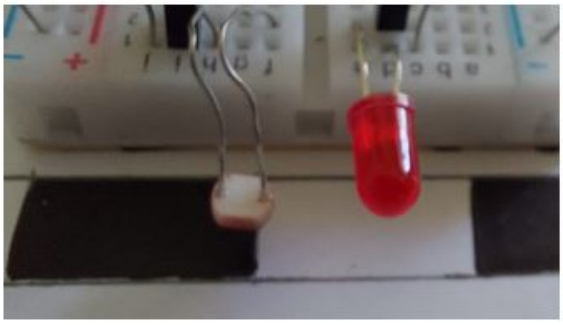
Secondly, the input patterns shown in figure 2 and 3 for the paper input method and the code snippet in section 3 for the LED input method can be drawn or coded using any length as long as there is a clear distinction in the length of a dash vs a dot and a long white space vs a short white space. Also for the paper input method the colour of a dash and dot or a space does not have to be Black and white as long as the dash/dot colour is significantly darker then the space colour, the code will work.

Thirdly, the reader is not affected by external light as long as the external light that is present remains constant while measuring the encoded message. Lastly, the LDR sensor should be a distance of at least 1 [cm] from the LED's or the paper message during reading, but should not be greater than 2.5x that distance to avoid measuring unwanted light at the most.

Lastly, ensure that the LED light direction range and the length of the drawn dot or dash on a piece of paper is larger then the area of the surface of the LDR as shown below:



LED Input Method



Paper Input Method

Note that for the LED input the LDR being the component on the left side of the first picture faces the LED directly and for the paper input, the LDR on the right side of the second picture is directly over the paper encoded sequence (With 1 [cm] space between the paper and the sensor and that that pattern is greater than the area of the LDR.

3 Morse Code Input Methods

Note, the report refers to LDR inconsistencies. What is meant by that is when measuring a drawn dash or active LED emitting a dash, several times, the length that the voltage was measured to be the same, differed from iteration to iteration.

3.1 Paper Input

Method Outline: This method involved a white paper sheet with the dashes and dots printed in black using a marker that is then passed under the LDR sensor at a constant rate. NB - The paper input should begin and end with white space (of any length) and the Morse coded message in between them (e.g white followed by dash then white then dot then white).

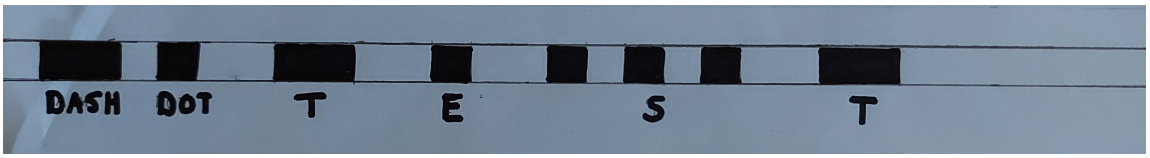


Figure 2: Paper Input Signal - 1

Figure 2: should be read (using the LDR sensor) from LEFT to RIGHT. Note: the calibrating pattern on the left followed by the encoded message.

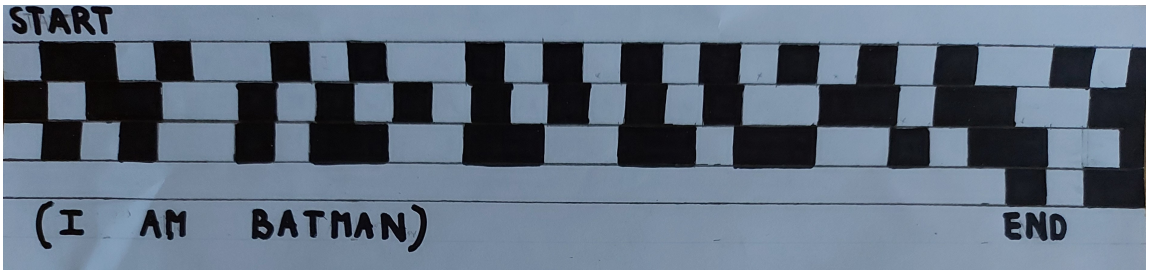


Figure 3: Paper Input Signal - 2

Figure 3: should also be read (using the LDR sensor) from LEFT to RIGHT on the first line signified by 'START'. Then, from RIGHT to LEFT on the line below followed by LEFT to RIGHT, continuing in the same manner until the 'END' is reached. Notice that when changing lines the pattern moves from being read horizontally to vertically. The length of BLACK and WHITE patterns should be kept constant while drawn though the code does account for minor inconsistencies allowing for some error when moving/drawing from one line to the next.

Outcome: This method was for the most part successful. The best success rate occurred when the input was measured at a much slower pace. With this particular method, as the act of measuring the input is dependant on the user, some error was expected to occur. Thus the dash defined as length when measuring the calibration sequence in the beginning, may differ to another dash measured length a few seconds later. For the most part minor changes do not affect the reading however, during measuring, I found that occasionally, my hand would stutter if the surface that I was measuring was not absolutely flat. This would cause a split second pause resulting in the measured length differing from the calibration lengths measured at the start. This along with the slight inconsistencies in the LDR sensor would occasionally result in an incorrect reading. (However measuring at a much slower rate did aid in reducing this effect to almost zero).

3.2 Dual LED Input

Method Outline: This method involved two different coloured LED's. One LED was used to display a dash and the second LED was used to display a dot. The LED's were positioned in front of the LDR sensor.

Outcome: This method did not work so well compared to the other two methods. The problem arose when an attempt was made to measure the difference between the different coloured LED's. The readings measured by the LDR sensor were very similar and thus difficult to differentiate between the colour of the LED.

3.3 Single LED Input

Method Outline: This method involved a single LED that was lit for a short period to represent a dot and a long period to represent a dash. The LED was then position in front of the LDR sensor of about 1 [cm] from each other. Below is the example code of displaying the encoded message 'TEST'.

```
void *Red_LED_Input(void *vargp){
   // This function generates a LED Morse Code Message to display 'TEST'
   int small_WAIT = 500; // This is the short waiting period
   int large_WAIT = 1000; // This is the large_WAIT waiting period

   digitalWrite(LED_PIN_1,LOW); // RED -- DOT initially low


   delay(small_WAIT);
//-----------------------------------------------------------------------------
   // Calibrating Sequence START
   digitalWrite(LED_PIN_1,HIGH); // RED -- DASH -- ON
   delay(large_WAIT);
   digitalWrite(LED_PIN_1,LOW);  // RED -- DASH -- OFF
   delay(small_WAIT);
   digitalWrite(LED_PIN_1,HIGH); // RED -- DOT -- ON
   delay(small_WAIT);
   digitalWrite(LED_PIN_1,LOW);  // RED -- DOT -- OFF
   delay(large_WAIT);
   // Calibrating Sequence END
//-----------------------------------------------------------------------------
   digitalWrite(LED_PIN_1,HIGH); // RED -- DASH -- ON     //    T
   delay(large_WAIT)
   digitalWrite(LED_PIN_1,LOW); // RED -- DASH --OFF
   delay(large_WAIT);                                  //  large_WAIT SPACE
//-----------------------------------------------------------------------------
   digitalWrite(LED_PIN_1,HIGH); // RED -- DOT -- ON       //    E
   delay(small_WAIT);
   digitalWrite(LED_PIN_1,LOW); // RED -- DOT -- OFF
   delay(large_WAIT);                                  //  large_WAIT SPACE
//-----------------------------------------------------------------------------
   digitalWrite(LED_PIN_1,HIGH); // RED -- DOT -- ON       //    S
   delay(small_WAIT);
   digitalWrite(LED_PIN_1,LOW); // RED -- DOT -- OFF
   delay(small_WAIT);                                  //  small_WAIT SPACE
   digitalWrite(LED_PIN_1,HIGH); // RED -- DOT -- ON       //
   delay(small_WAIT);
   digitalWrite(LED_PIN_1,LOW); // RED -- DOT -- OFF
   delay(small_WAIT);                                  //  small_WAIT SPACE
   digitalWrite(LED_PIN_1,HIGH); // RED -- DOT -- ON       //
   delay(small_WAIT);
   digitalWrite(LED_PIN_1,LOW); // RED -- DOT -- OFF
   delay(large_WAIT);                                  //  large_WAIT SPACE
//-----------------------------------------------------------------------------
```

```
45      digitalWrite(LED_PIN_1,HIGH); // RED -- DASH -- ON     //    T
46      delay(large_WAIT);
47      digitalWrite(LED_PIN_1,LOW); // RED -- DASH --OFF
48      delay(large_WAIT);                                    //  large_WAIT SPACE
49
50      // Ensure LED pins are low
51      digitalWrite(LED_PIN_1,LOW);
52
53
54      pthread_exit(NULL);; // End thread
55  }
```

Outcome: This method proved most successful, the message was executed by the Raspberry Pi in a separate thread. The duration of the LED flashes as well as the delays between them were all constant. Therefore it was easy to analyse the voltage signal and differentiate between dashes, dots and spaces. As the measured length a dot/dash was very similar to the measured calibrating length, the accuracy was almost 100 % only faulting occasionally due to the inconsistencies of the LDR sensor.

4 Essential Code Breakdown

All resources that were used and aided during the development can be found in the Reference section at the end. For each function there will be an overview of what the function does, followed by the code with comments. Only the most essential and complicated functions are explained below. Note that the sequence that these functions occur in the actual code is:

1. The Middle_Voltage Function - determines difference between BLACK and WHITE
2. The DashDot_AND_Space_Length Function - determines the length of dash, dot, large space and small space by analysing the calibration message
3. The Conversion Function - determines the remainder of the message by comparing the measured BLACK and WHITE lengths to the lengths measured during the calibration sequence
4. The Input_Speed_Adjuster Function - this in invoked every iteration of the conversion function to adjust for the inconsistencies of both measuring the input and the LDR sensor by setting the measured length to as close to the measured calibration length as possible.

The functions are called in a threaded manner at certain points in the program in the above sequence starting when the voltage array is at half capacity or if the voltage array never reaches half capacity then once the measuring of the input has concluded and the program moves from read mode to standby mode.

For a full understanding of the code please refer to the .c files themselves in this repository that are fully commented.

4.1 Voltage-Array Interface Definitions

There are two functions that allows one to write a recorded voltage value and read a recorded voltage value. These functions are pretty straight forward and thus the code is not shown but

can be found in either .c files in the repository named:

1. fill_Array - Add a voltage to the array
2. analyse_Array - Pop a voltage from the array

Both the functions work by recording/popping a value from the beginning of the array and then once the end of the array is reached starting again from beginning. The program is coded in such a manner so that the recording array function is always ahead of the popping function.

4.2 The Middle_Voltage Function

This function is intended to define the difference between BLACK and WHITE. There are no arguments required. The function achieves this by analysing the voltage values and determine the highest and lowest voltages.
The function only analyses the first half of the voltage values to determine the highest and lowest and it is assumed that the calibration sequence will have been measured and stored in the first half of values stored in the array. Then by summing the highest and lowest and dividing by 2, the average middle value between WHITE and BLACK is achieved. This is possible as the difference between the measured LDR voltages for BLACK and WHITE or lit and un-lit LED's are very definite.

You will notice that the function itself is a pointer type function, this was done as the function is executed in a thread and the Pthread library requires that type of function definition.

```
void *Middle_Voltage(){
    /* This function is used to determine the middle voltage and defines the
    difference between a BLACK and WHITE part or between a lit LED or non-lit
    LED

    The function only analyses the inital voltage values
    */
    int highest = 0;
    int lowest = 1000;
    int count = 0; // temp array count variable

    if (array_Append_COUNT >= array_LENGTH){
        // Analyse the  1/3 inital points in array if message is greater than 1/3 the length of the array
        while (count < array_LENGTH) {
            if (Voltage_Values[count] > highest){
                highest = Voltage_Values[count] ;
            } else if (Voltage_Values[count] < lowest){
                lowest = Voltage_Values[count];
            }
            count += 1;
        }


    } else {
        // Analyse the message if the message is less than the 1/3 length of array
        // Thus if only the calibrating pattern is provided
        while (count <= array_Append_COUNT) {
```

```
28          if (Voltage_Values[count] > highest){
29              highest = Voltage_Values[count] ;
30          } else if (Voltage_Values[count] < lowest && Voltage_Values[count] != 0){
31              lowest = Voltage_Values[count];
32          }
33          count += 1;
34      }
35  }
36
37
38  // Then calculate Median to define difference between BLACK and WHITE data points
39  BLACK_WHITE_Differentiator = (highest + lowest) / 2;
40  Middle_Function_STATUS = 1; // Set function status to completed
41  pthread_exit(NULL);
42 }
```

4.3 The DashDot_AND_Space_Length Function

In order to analyse a Morse code signal, one would need to be able to separate a signal in four parts. Namely: a dot (shown by 4 and 6 in Figure 4), a dash (shown by 2 and 8 in Figure 4), the small space separating dots and dashes (shown by 3 in Figure 4) and the large space separating patterns of dots and dashes (shown by 7 in Figure 4). The figure below is an example of a Morse code signal separated into the various parts.

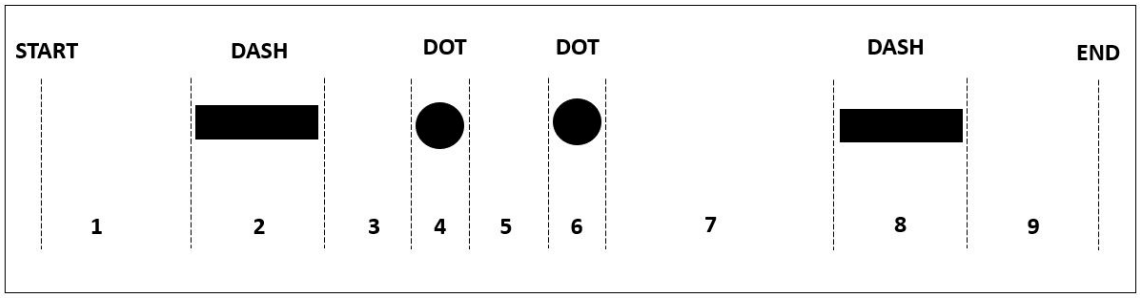


Figure 4: Morse Code Input Signal Example

This function only interprets the calibrating sequence at the beginning of the message shown in Figure 1. As the sequence that they will occur is known, the objective would simply be to identify the sequence.
The function achieves this by separating each voltage value into BLACK or WHITE, then counts the consecutive BLACK or WHITE values while temporarily storing the previous voltage value. Then when there is a change between BLACK and WHITE, the function then sets the four parameters in the sequence they are expected to appear. To identify the change in colour, the previous and current voltage value will differ significantly. The four parameters are then set as global variables to be used by the Conversion Function.

You will notice that the function itself is a pointer type function, this was done as the function is executed in a thread and the Pthread library requires that type of function definition. Also on lines 18 and 44, the greater than and less than operaters should be switched for the LED Input method.

```
void *DashDot_AND_Space_Length(){
    // This function determines the length of a dot and the length of a space between dots and dashes
    // using the middle_Voltage() function to differentiate between BLACK and WHITE

    // This function only analyses the calibrating pattern at the beginning of the message

    int current_Space_Count = 0;
    int current_Voltage_Count = 0;

    int previous = 0;
    int count = 0;
    int while_CONDITION = 0;
    int Initial_Space_Ignore = 0;

        while (while_CONDITION != 4) {
            int temp_Voltage = Voltage_Values[count];

            if (temp_Voltage > BLACK_WHITE_Differentiator) {
                // Found WHITE/SPACE

                if (previous <= BLACK_WHITE_Differentiator && current_Voltage_Count != 0
                 && previous != 0){
                // Moved from BLACK to WHITE/SPACE

                    if (Initial_Dot_LENGTH == 0 && Initial_Dash_LENGTH == 0) {
                        Initial_Dash_LENGTH = current_Voltage_Count;
                        while_CONDITION += 1;
                    } else {
                        Initial_Dot_LENGTH = current_Voltage_Count;
                        while_CONDITION += 1;
                    }

                    current_Space_Count = 1; // include the current WHITE node
                    current_Voltage_Count = 0; // reset BLACK part counter

                } else {
                    // Still counting WHITE pattern
                    current_Space_Count += 1;
                }
                previous = temp_Voltage;
                count += 1;


            } else if (temp_Voltage <= BLACK_WHITE_Differentiator){
            // Found BLACK


                if (previous > BLACK_WHITE_Differentiator && current_Space_Count != 0
                && previous !=0){
                    // Moved from WHITE/SPACE to BLACK

                    if (Initial_SmallSpace_LENGTH == 0 && Initial_BigSpace_LENGTH == 0
                    && Initial_Space_Ignore == 1) {
                        Initial_SmallSpace_LENGTH = current_Space_Count;
                        while_CONDITION += 1;
                    } else if (Initial_Space_Ignore == 1) {
                        Initial_BigSpace_LENGTH = current_Space_Count;
```

```
                    while_CONDITION += 1;
                }

                Initial_Space_Ignore = 1;
                current_Voltage_Count = 1; // include the current BLACK node
                current_Space_Count = 0;

            } else {
                // Still counting BLACK pattern
                current_Voltage_Count += 1;
            }

            previous = temp_Voltage;
            count += 1;
        }
    }


    Dash_Dot_Space_Function_STATUS = 1; // Set function status to completed

    pthread_exit(NULL);
}
```

4.4 The Input_Speed_Adjuster Function

This function takes in the instantaneous accumulated length of either a BLACK/lit LED or WHITE/un-lit LED or large space or small space from the Conversion Function as the first argument. The function then checks to see if its BLACK ('1') or WHITE ('0') as per the second argument.
Then finds the difference between the measured length and the known calibration length of both the dot/dash or small space/large space determined by the Dash-dot function earlier. The smaller difference then defines what the measured length is classified as. The measured result is then set to either dash/dot or small space/large space before returning the variable to the Conversion Function.

```
int Input_Speed_Adjuster(int Current_Length, int Message_Type){
    // Message_Type = 1: means that the length is BLACK
    // Message_Type = 0: means that the length is WHITE
    int New_length;
    if (Message_Type == 0){
        // Adjusting a BLACK part
        int dot_Difference = abs(Initial_Dot_LENGTH - Current_Length);
        int dash_Difference = abs(Initial_Dash_LENGTH - Current_Length);

        if (dot_Difference < dash_Difference){
            New_length = Initial_Dot_LENGTH;
        } else{
            New_length = Initial_Dash_LENGTH;
        }

    } else {
        // Adjusting a WHITE part
        int small_Difference = abs(Initial_SmallSpace_LENGTH - Current_Length);
        int Big_Difference = abs(Initial_BigSpace_LENGTH - Current_Length);
```

```
20
21      if (small_Difference < Big_Difference){
22          New_length = Initial_SmallSpace_LENGTH;
23      } else{
24          New_length = Initial_BigSpace_LENGTH;
25      }
26
27  }
28  return New_length;
29 }
```

4.5 The Conversion Function

The function takes in no arguments with the purpose of converting the voltage signal to an alphanumeric sequence.
The function iterates through the voltage values measured by the LDR sensor. It begins by determining if each value is BLACK or WHITE by determining if the current value is greater than or less than the voltage determined by the Middle_Voltage function. Then counts the the consecutive BLACK or WHITE values. Similarly to the DashDot_AND_Space_Length function mentioned earlier, the function differs when a change from BLACK to WHITE or vice versa is discovered, which is comparing the previous colour state vs the current colour state.

Upon discovering a colour state change, if the change was from BLACK to WHITE, then the algorithm analyses the current BLACK count to define it as a DASH or DOT by comparing it to the calibrating Dash and Dot from the beginning of the pattern. The newly defined dash or 1 and dot or 0 is added to a temporary array. If the change was from WHITE to BLACK, then the algorithm analyse the current WHITE or Space count by comparing it to the short and long spaces measured in the calibrating sequence. If a short space is discovered, it assumes there is another dot or dash to analyse and continues. If a long space is discovered, that meant that a Morse code pattern has concluded and needs to be converted.

The temporary array filled with ones and zeros (dashes and dots) is then compared to a dictionary of alphanumeric symbols with the corresponding morse code pattern (which is also in a zero - one format) until either a match is found or not found. If found, the matching alphanumeric symbol is added to second global array to be displayed later. You will notice that this piece of code is repeated as the message would end on a WHITE voltage value and as explained, the algorithm on analyses the dash dot pattern when a change in colour is detected. Thus that portion of the code needs to be repeated to analyse the most recent BLACK pattern measured.

The function and thread will remain in existence until the current voltage value to be analysed is equivalent to zero as shown on line 15 of the code. This condition works as when the reader moves from reading a message to the completion of reading message (indicated by pushing the button) a zero is added to the array in the next open position of the array as the terminating symbol of the recorded voltages.

You will notice that the function itself is a pointer type function, this was done as the

function is executed in a thread and the Pthread library requires that type of function definition. Also on lines 34 and 71, the greater than and less than operaters should be switched for the LED Input method.

```
1  void *Conversion(){
2      printf("..................................................\n");
3      printf("Currently Converting:\n");
4      Conversion_Function_STATUS = 1;
5
6
7      printf("\n");
8      printf("Dot Length: %d\n",Initial_Dot_LENGTH);
9      printf("Dash Length: %d\n",Initial_Dash_LENGTH);
10     printf("Small Space Length: %d\n",Initial_SmallSpace_LENGTH);
11     printf("Large Space Length: %d\n",Initial_BigSpace_LENGTH);
12     printf("BLK/WHT Mid-Value: %d\n",BLACK_WHITE_Differentiator);
13     printf("\n");
14
15         int Conversion_Function_DashDot_Count = 0; // used to count the length of a dash or dot
16         int Conversion_Function_Space_Count = 0;
17         char Conversion_Function_MorseCode_Current[8]; // this is used to store the current 0's
18         // and 1's to compare later
19
20         int Conversion_Function_MorseCode_Current_CHECK = 0; // This is used to ignore the first
21         //white space of the input message
22         int Conversion_Function_MorseCode_Current_COUNT = 0; // counter for the array
23
24         int Conversion_Function_Previous_Voltage = 0;
25
26
27         // This means that the message has endend
28         // Not possible to exceed the length of array
29         int voltage_Value = analyse_Array();
30         while (voltage_Value!=0) {
31
32
33
34         if ( voltage_Value > BLACK_WHITE_Differentiator){
35             // Found WHITE
36             if (Conversion_Function_Previous_Voltage <= BLACK_WHITE_Differentiator
37             && Conversion_Function_Previous_Voltage != 0){
38                 // Moved from BLACK to WHITE
39                 printf("BLACK: %d\n",Conversion_Function_DashDot_Count);
40
41                 Conversion_Function_DashDot_Count = Input_Speed_Adjuster(Conversion_Function
42                 _DashDot_Count,0); // Invoke for BLACK
43
44
45                 // Analyse if the BLACK part is a dash or dot
46                 if (Conversion_Function_DashDot_Count == Initial_Dash_LENGTH){
47
48                     // Found a DASH
49                     Conversion_Function_MorseCode_Current[Conversion_Function_MorseCode_Current_
50                     COUNT] = '1'; // for a dash
51                     Conversion_Function_MorseCode_Current_COUNT += 1;
```

```
                Conversion_Function_MorseCode_Current_CHECK = 1;  // Ensures that the first white
                // space is ignored
            } else {
                // Found a DOT
                Conversion_Function_MorseCode_Current[Conversion_Function_MorseCode_Current_
                COUNT] = '0'; // for a dot
                Conversion_Function_MorseCode_Current_COUNT += 1;
                Conversion_Function_MorseCode_Current_CHECK = 1;  // Ensures that the first white
                // space is ignored
            }
            Conversion_Function_DashDot_Count = 0;  // Reset BLACK part counter
            Conversion_Function_Space_Count = 1; // Reset WHITE space count including
             // current WHITE part
        }
        else{
            // Just counting WHITE
            Conversion_Function_Space_Count += 1;
        }

    } else if ( voltage_Value <= BLACK_WHITE_Differentiator) {
        // Found BLACK
        if (Conversion_Function_Previous_Voltage > BLACK_WHITE_Differentiator && Conversion_
        Function_Previous_Voltage != 0){
            // Moved from WHITE to BLACK
            printf("White: %d\n", Conversion_Function_Space_Count);



            Conversion_Function_Space_Count= Input_Speed_Adjuster(Conversion_Function_
            Space_Count,1); // Invoke for WHITE

            // Analyse if the WHITE part is a short or long space
            if (Conversion_Function_Space_Count == Initial_BigSpace_LENGTH && Conversion_
            Function_MorseCode_Current_CHECK != 0){
                // Found a long space meaning end of a alphanumeric symbol
                Conversion_Function_MorseCode_Current[Conversion_Function_MorseCode_
                Current_COUNT] = '.';
                int stop = 0;
                for (int i = 0; i<37 && stop == 0; i++){
                    for (int j = 0; j<7; j++){
                        if (Conversion_Function_MorseCode_Current[j] == morseCode[i][j]){
                            if (j == 6){
                                // Found a ' ' or space between words in a message
                                Final_Message[Final_Message_COUNT] = symbol[i];
                                Final_Message_COUNT += 1;
                                stop = 1;
                                break;
                            } else if (Conversion_Function_MorseCode_Current[j] == '.'
                            && morseCode[i][j] == '.') {
                                // Found a matching pattern thats less than 7 units long
                                Final_Message[Final_Message_COUNT] = symbol[i];
                                Final_Message_COUNT += 1;
                                stop = 1;
                                break;
                            } else{
                                // keep checking to see if remainder matches
                                continue;
```

```
                        }
                    } else if (Conversion_Function_MorseCode_Current[j] != morseCode[i][j]) {
                        break; // used to break the inner for loop and continue to the outer loop
                    }
                }
            }

            memset(Conversion_Function_MorseCode_Current, 0, 8); // Empties Array for the
            //next BLACK pattern
            Conversion_Function_MorseCode_Current_COUNT = 0; // reset temp array counter
        }
        Conversion_Function_Space_Count = 0;  // Reset WHITE part counter
        Conversion_Function_DashDot_Count = 1; // Reset BLACK space count including
        // current BLACK part

    }
    else{
        // Just counting BLACK
        Conversion_Function_DashDot_Count += 1;
    }
}


    Conversion_Function_Previous_Voltage = voltage_Value;  // Save the current value for use later
    voltage_Value = analyse_Array();
}



// Run the code below again to convert the last BLACK pattern
int stop = 0;
Conversion_Function_MorseCode_Current[Conversion_Function_MorseCode_Current_COUNT] = '.';
for (int i = 0; i<37 && stop == 0; i++){
    for (int j = 0; j<7; j++){
        if (Conversion_Function_MorseCode_Current[j] == morseCode[i][j]){
            if (j == 6){
                // Found a ' ' or space between words in a message
                Final_Message[Final_Message_COUNT] = symbol[i];
                Final_Message_COUNT += 1;
                stop = 1;
                break;
            } else if (Conversion_Function_MorseCode_Current[j] == '.' && morseCode[i][j] == '.') {
                // Found a matching pattern thats less than 7 units long
                Final_Message[Final_Message_COUNT] = symbol[i];
                Final_Message_COUNT += 1;
                stop = 1;
                break;
            } else{
                // keep checking to see if remainder matches
                continue;
            }
        } else if (Conversion_Function_MorseCode_Current[j] != morseCode[i][j]) {
            break; // used to break the inner for loop and continue to the outer loop
        }

    }
}
```

```
166     memset(Conversion_Function_MorseCode_Current, 0, 8); // Empties the array
167     Conversion_Function_STATUS = 2;
168     pthread_exit(NULL);
169 }
```

5 Modifications/Changes from the Original Design

As with most projects the final outcome does not mirror the planned outcome exactly and some changes were required including:

- **Memory Allocation:** Initially a dynamic array structure was supposed to be used to stored the measured voltages and the final alphanumeric symbols in the form of a linked list as the exact size of the message cannot be unknown. This however was not compatible with the threaded structure of the code making parallelism impossible. Therefore a set array length of size equal to 400 is used. Then as explained in the code section, the array first fills until capacity and then cycles back and begins from position zero once full. This allows for any length of message while still being able to run in a threaded, parallel manner.

- **Dash/Dot Identification:** In the planning stage, the expectation was that one could define the length of a dot and a small white space and then assume that everything larger than that is to be considered as a dash or large white space. Due to the inconsistencies, that was not the case and thus the modification requiring the calibration sequence and the Input speed adjuster method explained in the section prior was used instead.

  An attempt was made to measure the time of dashes and dots to adjust according to the speed that the input was being measured. The possible solution involved determining a mathematical relationship between the length of a dash, dot, small space and large space at a known speed and then determine the best fit line equation to determine the necessary correction required at the current speed. The problem arose when the length of a dash and dot could change depending no how the user decided to draw/code them. Thus a mathematical relationship for each possible length of the 4 possible parameters would be required. This was not an effective solution as the measured length of the 4 parameters was never the same when measured by the LDR due to inconsistencies of the LDR. Therefore even if the relationships where determined, knowing which relationship to use would not be possible. (A general relationship was attempted but for the most part ended in false readings)

- **LED GPIO Pins:** The final change refers specifically to the LED input code. In the planning stage for the two LED GPIO pins 26 and 6 were to be used but in the code GPIO pins 5 and 6 were used instead. This change was made solely to accommodate both the implementations (paper and LED input methods) on a single bread board without requiring any rewiring.

6 Challenges

6.1 Speed of the LDR vs the Paper Message Input

One of the biggest problems that was en-counted was the rate that the paper message was fed under the LDR sensor. Ideally the rate was supposed to be constant however, due to the human component of the process, the rate is impossible to be truly constant. This was important as the process of determining the difference between a dash and a dot id dependent on the rate.

6.2 The LDR Sensitivity

When comparing the measured voltages to the input messages, be it the LED method or paper method, the LDR would not always be able to measure the input accurately. Noticeable examples would be when the spaces between dots or dashes were less then 4 [mm] or the time delay between LED flashes were less than 0.5 [s] and the measuring rate would be kept as constant as possible. The LDR would see them as a single long dash/dot or space instead as separate dashes/dots. The problem would only worsen as the rate that the LDR speed measured the input message increased. Another observation was that the readings for two separate dashes for example, would differ in magnitude, even when the external environment light remained constant (Note: this was more prominent in the paper input method).

To reduces the effects of the problem measuring the input message at a much slower rate and ensuring that the messages of both the paper and LED methods are distinctly defined for dashes, dots and spaces.

7 Possible Improvements

7.1 The LDR Sensor

As mentioned in the Challenges section, the accuracy of the LDR varied quite a significant amount. Therefore, possibly using a more sensitive LDR would result in a much better voltage reading of the Morse code input.

7.2 Speed Control

The rate at which the paper input was measured affected the measured length used to differentiate between a dot or dash. Thus having some sort of speed identifier would allow for better adjustments to the measured length of BLACK or WHITE patterns. A possible solution would be to have a second parallel constant dotted line. Then using a second LDR sensor that would read the dotted line, could determine the change in speed of the message relative to the length measured between the dots. If the LDR is more accurate than the current version with a mathematical approach could be used if the exact speed is known.

8 Conclusion

To conclude, the single LED method and the paper input method were to a greater extent successful in determining the message while the dual LED input was a failure. The LED input method is still slightly better than the paper input but if the measuring speed of the paper input is reduced, the success rates are on par. The two major problems were the inconsistency of the LDR sensor and the change in the measuring speed that occurred for the paper input method.

9 References and Useful Resources

[1] B. Barney, "POSIX Threads Programming", Computing.llnl.gov, 2008. [Online]. Available: https://computing.llnl.gov/tutorials/pthreads/. [Accessed: 21- Feb- 2021]. (Usefull for understanding how to implement threads in C)

[2]"Learn C - Free Interactive C Tutorial", Learn-c.org, 2021. [Online]. Available: https://www.learn-c.org/. [Accessed: 20- Feb- 2021]. (Very good resource to check C syntax)

[3] mcp3002.c and mcp3004.c - Raspberry Pi Forums", Raspberrypi.org, 2021. [Online]. Available:https://www.raspberrypi.org/forums/viewtopic.php?t=61597. [Accessed: 08- Feb- 2021]. (Useful for understanding how to interface with the ADC chip in C)

[4] e. notes, "Light Dependent Resistor LDR, Photoresistor » Electronics Notes", Electronics-notes.com, 2021. [Online]. Available: https://www.electronics-notes.com/articles/electronic_components/resistors/light-dependent-resistor-ldr.php. [Accessed: 18- Feb- 2021]. (Notes on how an LDR works)

[5]"Raspberry Pi GPIO Pinout", Pinout.xyz, 2021. [Online]. Available: https://pinout.xyz/. [Accessed: 18- Feb- 2021]. (The pin layout of a raspberry Pi)

[6] WiringPi", Wiringpi.com, 2021. [Online]. Available:http://wiringpi.com/. [Accessed: 08- Feb- 2021]. (Useful for understanding how to interface with the raspberry Pi)