Raspberry Pi - Morse Code Reader
Software Specifications

Author: Yashiv Fakir

Date: 20 January 2020

---

1 Introduction

This document aims to outline all the software aspects of the Morse Code project. The report begins with the programming language and libraries used for the project. Then moves onto the sequence and algorithm of the code. Finally concludes with a brief overview of the memory structure used by the algorithm.

2 Programming Language

The first decision was to determine which programming language to be used, to choose either Python or C (or possibly a combination of these). The decision was made to code using C. The motivation behind this was that C offers much faster speeds during program execution once compiled. There is also much better memory control which is advantageous given that the memory on a Raspberry Pi Zero is limited. A combination of the two would have been a better solution as interfacing with the Pi is much simpler to do in Python but offers many potential problems that would draw one's focus from the main objective being to decipher a Morse coded message rather than to combine two coding languages. As C is a compiled language, a compiler is necessary to execute the code. For this project the GNU GCC compiler will be used.

2.1 C Libraries

The main C libraries that are required for the project are as follows:

- wiringPi.h - This library is used to interface between the C code and the GPIO ports on the raspberry Pi. WiringPi also has functionality to deal with button interrupts.
- mcp3008.h - This library is part of the wiringPi.h library that will interface between the ADC chip and the C code
- stdio.h - This library will allow for the decoded message to be displayed on the console as well as any progress or failed errors to be displayed
- string.h - This library will aid in the conversion and combination of each character to form the deciphered string.

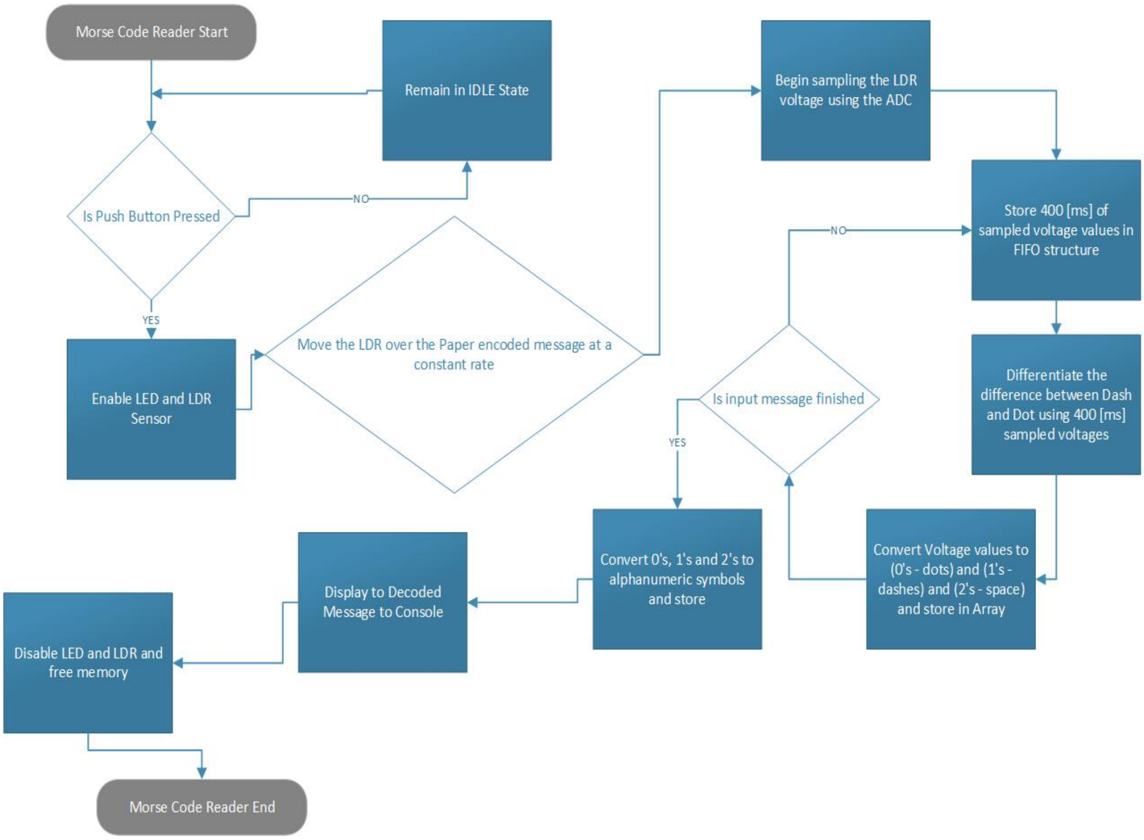
3 Program Flow

The overall sequence of events will be pretty similar only differing to cater for the various input encoded messages being the paper input and the LED input (As shown in the two Flowcharts below).

The actual algorithm for converting from the measured LDR voltage values to alphanumeric symbols will be the same for both inputs with one exception. For the paper input, only the black marks form the message and on the LDR, will resemble a high voltage. For the LED input, only the flash of the LED forms part of the message and on the LDR, will resemble a low voltage. Thus the exception is the identifiers for the message, that differ for both inputs being a high or low voltage. The program is also initiated to begin reading only when a push button is pressed. This was done to prevent the constant use of memory to see if there is an input being fed to the sensor or not. Instead due to the button, memory used to store the input signal will only be used when there is an actual message to be stored.

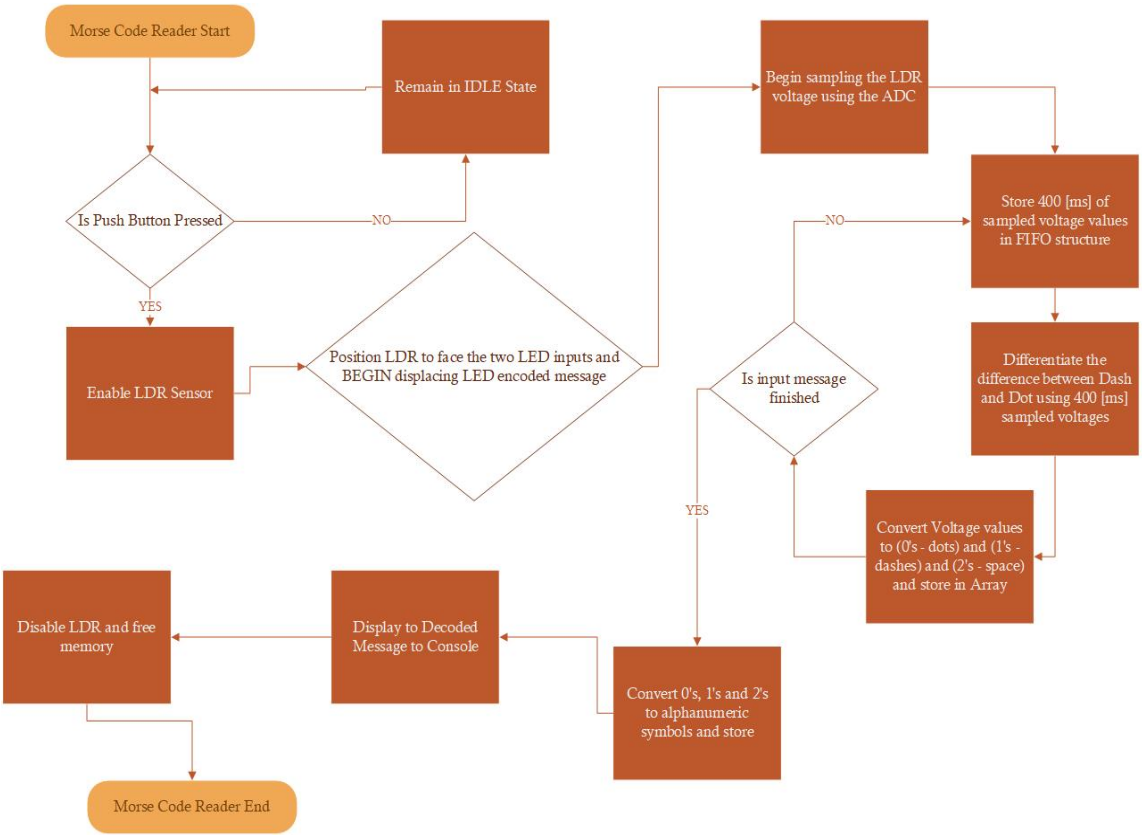
3.1 Program sequence for Paper Input

Below is a flow chart showing the program sequence for a paper encoded message.



3.2 Program sequence for LED Input

Below is a flow chart showing the program sequence for an LED encoded message.



4 Algorithm

4.1 Error Handling

1. Error: If either input being fed is blank (if a blank sheet of paper or if the LED's aren't illuminated as input to the system).

   Solution: Will have to have a check function that analyses the input voltages by the LDR. If no changes after a period of time say 2-3 [s] then stop the reading and display an error message.

2. Error: If the rate that the paper input is fed to the system is either extremely fast or slow to record the results. This error poses challenges when at a very fast rate.

   Solution: An extremely slow input would still work with the main algorithm however would have to have upper limit in the amount of voltage values measured to avoid running out of memory, after which the system times out. For a fast input the voltage signal would either be a constant signal or only have a single spike. Thus if constant the previous error would handle this. But if a single spike is detected then this could either be a 'T' = dash or 'E' = dot in Morse code (due to the input rate), or could be an error this making it difficult to differentiate.

3. Error: If the space between LED flashes is two quick or to slow.

   Solution: The same solution for the paper input can be used here, however and LED flash will have a much greater effect on the LDR thus making it easier to detect.

4. Error: If there is insufficient high and low voltage samples measured to determine the difference between a dash or dot or even the difference between white or black (Lit LED or unlit LED).

   Solution: If the measured data is full of all high voltages or all low voltages then a difference between the message and a space/break would not be established. Also if half of the samples are low voltages and the other half all high, then a difference would be established but the length of a dot or dash would not be established. If either of the 2 conditions are met, then the previous data is to be kept and the remainder of the voltages that were measured after the initial data set are then to be analysed.

4.2 Pseudo Algorithm

NOTE: this algorithm only includes the base instructions and assumes that there is an ideal input message. The errors mentioned above will be coded in after the primary functions have worked successfully with an ideal input.

```
-----------------------------------------------------------------
GLOBAL VARIABLES
-----------------------------------------------------------------

(DYNAMIC ARRAY) float voltageValues[]   # type: link list

(DYNAMIC ARRAY) char alphanumericValues[]   # type: link list

(Dictionary) morseCodeREF{}    # Stores the character and morse code in 1's and 0's

int lenDot


-----------------------------------------------------------------
MAIN FUNCTION
-----------------------------------------------------------------

function int main()
inifite loop  ---  while(true)      #keeps application running
     until physically stopped

Button listener (with debouncing)   ---  call buttonEventHandler()
```

```
CREATE New Thread

Invoke the ADC   ---  enableADC ()  # Thread ensures LDR data is contineously
    # being recoded while main program runs

IF enableADC == 1
TERMINATE Thread  # terminate thread after program is complete
END

Print Message --- Output()       # Display final message

.................................................................



-----------------------------------------------------------------
SUPPORTING FUNCTION
-----------------------------------------------------------------

function void buttonEventHandler( )
# NOTE: the LDR is connected to to VCC and is always on

Print to console ---  Ask user to pass the message under the LDR
         ---  Then Display that system is running


Invoke LED       ---  enableLED ( gpioPinNumber )   # for the paper input
# this will emit light onto the paper and for
# the LED input this will begin the LED input
# sequence
.................................................................
function int enableADC( )
# Start ADC and send values to the PI

Fill voltageValues[] --- Add ADC voltage values using push()

while(voltageValues[] is not empty) # only creates thread if atleast one voltage
# value is present in the array

CREATE New Thread
Determine dot length ---  Call DashDot()
TERMINATE Thread

Begin the conversion from voltage to alphanumeric ---  conversion()
return 1

.................................................................
```

```
function void enableLED ( int gpioPinNumber )
# Simply starts ADC

Set gpioPinNumber ---  High   # Lights the LED or Begins LED coded message


.................................................................
function void DashDot( )
# Determines the length of a dot and anything larger is seen as a dash

avg ---  determine average of voltageValues[]
count = 0
lowestCount = 0


# The code below will continue until a dot and dash is found then stop

FOR x in voltageValues[]
IF x >= avg     # found black part of message
count + 1
ELSE            # found white space pasrt of message
IF lowestCount = 0
--- lowestCount = count
IF count < lowestCount
--- lowestCount = count
END-IF's
Reset count --- count = 0
END-IF
END-FOR

lenDot = lowestCount  # set the dot length to the global dot length variable

.................................................................
function void conversion( )
# Coverts voltages into alphanumeric symbols and adds to alphanumericValues[] link list.

avg ---  determine average of voltageValues[]
arrayCount = 0   # Used to itterate through array
count = 0  # Used to count length of high voltages either dash/dot
int tempArray[7]    # temp array has a length of 7 as a 'space' has seven dashes which
#is the largest possible pattern
char alphaSymbol



FOR x in voltageValues[]  # Begins from the beginning of array
IF x < avg  AND length(tempArray) = 0 # This would be white space
```

```
pop( x ) # remove from voltageValues[] array
free() # release that bit of memory
SKIP

ELSE IF length(tempArray) is 7  # completed deciphering a word
# and found a space that separates words

alphaSymbol --- compare contents of tempArray[] to morseCodeREF{}
#  To find the alphanumeric equivalent
#  of the morse Code

push( alphaSymbol ) --- add the character to the
2nd list being alphanumericValues[]
# add the current word

push( " ") --- Add a space to alphanumericValues[]  # adding a space separater
    # after current word

empty tempArray[]

set arrayCount = 0


ELSE IF x < avg  AND length(tempArray) > 0  # found a white space after finding
# a black pattern:Then this is the
# end of a pattern for a single
# character
IF count>lenDot   # Then found a dash
tempArray[arrayCount] = 1
arrayCount + 1
count = 0
ELSE  # found a dot
tempArray[arrayCount] = 0
arrayCount + 1
count = 0
END IF

ELSE # this is if there are still black pieces of the pattern being detected
count + 1
pop( x ) # remove from voltageValues[] array
free() # release that bit of memory



.................................................................
function void Output( )
# This outputs the deciphed messaged and frees the memory from the alphanumericValues[]
```

```
string finalMessage = ""

for y in alphanumericValues[]
finalMessage + y
pop( y ) # remove from voltageValues[] array
free() # release that bit of memory


print( finalMessage )


-----------------------------------------------------------------
ARRAY FUNCTION
-----------------------------------------------------------------

function void push( pointer*, value )
# add to the top of the list

.................................................................
function void pop( ponter*, value )
# removes from the bottom/ removes oldest value

.................................................................
function void free( )
# frees the memory after being popped

.................................................................
```

4.3 Memory Overview

The program itself will at most have three threads running. The main program will run in a single thread. The second thread will run the ADC and continually be measuring and saving the measured voltages to memory. The third thread will determine the length of a dash and dot using the initially measured data (This thread will be executed as soon as sufficient data points are recorded and then terminate once completed.

There will also be two blocks of memory in the form of a dynamic arrays or linked lists. The first block will retain the measured voltage values. the second block will retain the alphanumeric characters and spaces. Both the blocks of memory will be stored on the heap. Finally a temporary array of length seven will be used to hold the dash and dot pattern before converting to a alphanumeric symbol. As this is a set length, it can be stored on the stack.

All data measured and recorded will be stored in RAM and will only exist during the duration of the program execution. Not secondary/permanent storage functions have been added to this project scope.