

EECS 4352 Real Time Project

Morse Code Translator

Jeremi Boston (2121432399) _____

Dinesh Kalia (213273420) _____

TA: Navid Mohaghegh _____

Date of Submission: April 5, 2017

Last Update: April 1, 2017

Table of Contents

1. Introduction	3
2. Design	4
2.1 Theoretical Framework	4
2.2 Hardware Components	5
2.3 Software Components	5
3. Procedure	6
3.1 Tutorial	6
3.2 Testing	8
4. Discussion	11
4.1 Design and Implementation	11
4.1.1 Keypad	11
4.1.2 LCD	11
4.1.3 Speaker	12
4.1.4 Flags	12
4.2 Encountered Errors	13
4.2.1 LCD Display	13
4.2.2 Handling User Errors	14
4.3 Performance and Correctness	14
5. Applications	15
6. Conclusion	16
7. References	17
8. Appendices	18

1.0 Introduction

This project implements a system that translates Morse Code to text. Morse code is a form of communication which uses combination of dits (“-”) and dots (“.”) to form an english alphabet. Historically, telegraphs were used to transmit dits and dots with electronic signals (**Figure 1**). These signals would produce two distinct sounds, one for a dit and other for dot. The varying length of no signals would denote space between letters or words. The messages were received via a radio signal, sounding like dots and dashes of static. To an untrained individual, these sounds are just some random assortment of static and would be unaware of the hidden meaning. But, a person who has had morse code training would be able interpret the incoming signals into a code word such “Alpha”, “Bravo”, etc. Because of this unique property, morse code was used extensively by the military in World War I and II. It was a safe medium to relay messages from frontline to headquarters (or vice-versa) for enemy movements and strategy without the transmission being tapped and interpreted.

Even though such technology has progressed so far that telegraphs are looked as “primitive” electronic communication, but such simple device still exhibits significant complexity¹. The goal of this project is to implement a system that simulates telegraph on Dragon12 board which uses MicroC/OS-II real-time operating system. The input from the keypad is in the form of dits and dots and the decoded messages are shown on the LCD display. Moreover, the dits and dots also produce distinct sounds so that if a person knows code, they would be able to decode the message by only listening to the audio.

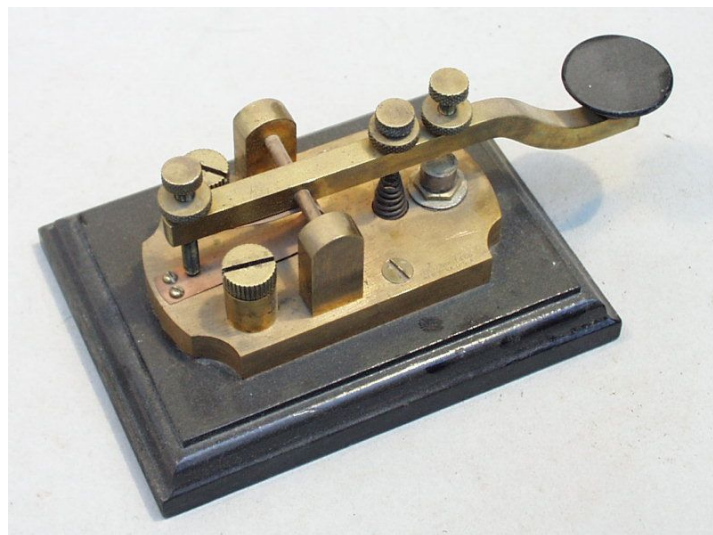


Figure 1: Morse Telegraph Key

¹ Sařsa and Madelaine Page Milić. "Morse Code Decoder." *Micro-Controller Applications*. N.p., 3 Dec. 2013. Web. 1 Mar. 2017.

2.0 Design

2.1 Theoretical Framework

At high level, the task was converting analog inputs (the pressing of a key) to digital values (letters appearing on LCD display). The dit-dot combination will be inputted via two keys (one key representing “-” and another for “.”). These inputs will congregate on the bottom display of the 16x2 LCD screen and the decoded english letter will appear on top of the LCD display. A third key can be pressed to compute the decoding process. After each decoding request, the appropriate english letter appends at the top of LCD screen in accordance to international morse code alphabet (**Figure 2**). Hence, it is possible to spell out whole words and sentences. After a desired word is spelled out, a fourth key can be pressed to reset the display for new words.



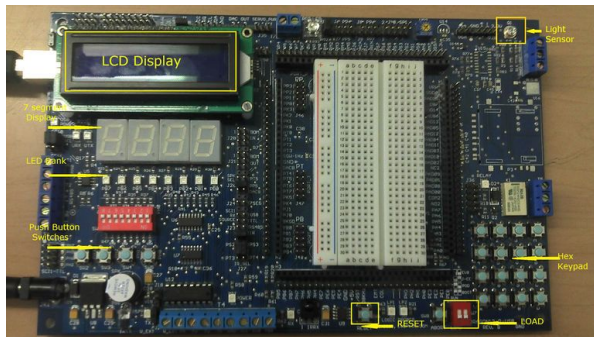
Figure 2: International Morse Code Alphabet²

Additionally, this implementation on MC9S12DP256B microcontroller also utilizes in-built speaker which makes two distinct sounds for dit and dots. The sounds are produced at different frequencies. This makes it so that there are two ways of interpreting Morse Code: visually (shown on LCD) or auditorily. There is no sound for the computation button so that the listener does not get confused. This duration of no-sound indicates end of the alphabet letter for those who are customized to audibly instructions.

² "SCPhillips.com." *International Morse Code*. N.p., n.d. Web. 01 Apr. 2017.

2.2 Hardware Components

The following components are required to simulate morse code on modern micro controller;



Dragon12 Plus Evaluation Board



Grounding mat with grounding cables



Power Supply



Micro Controller to USB cable

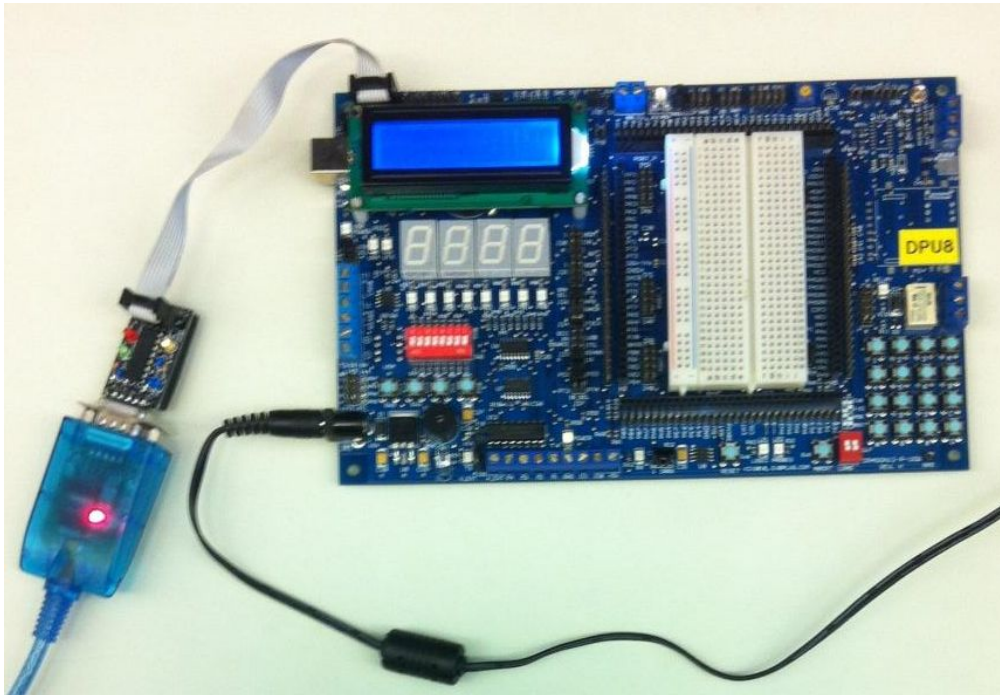
2.3 Software Components

This project uses the MicroC/OS-II real-time operating system within the CodeWarrior IDE environment. Furthermore, the Morse Code simulation uses Lab3.zip source files found at https://wiki.eecs.yorku.ca/course_archive/2016-17/W/4352/lab3.

3.0 Procedure

3.1 Tutorial

1. Open the project file using CodeWarrior located in:
Morse-Code/Micrium/Software/EvalBoards/Freescale/MC9S12DG256B/WytecDragon12/Metrowerks/Paged/OS-Probe-LCD/OS-Probe-LCD.mcp
2. Place grounding mat on workstation and plug in the grounding wire to grounding outlet.
3. Place Dragon12 board on the mat and hood up power source and micro-controller USB cable as shown;



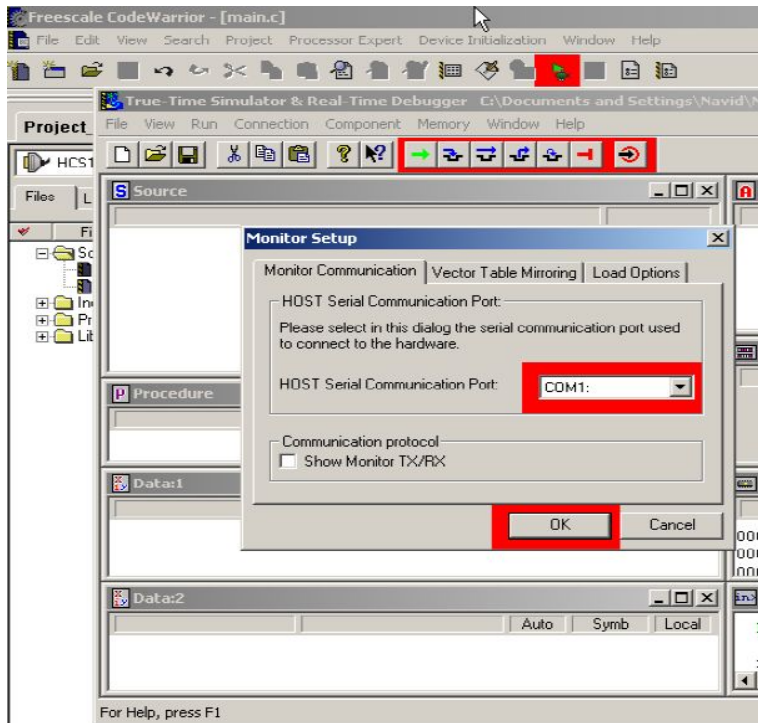
4. Compile the project using³



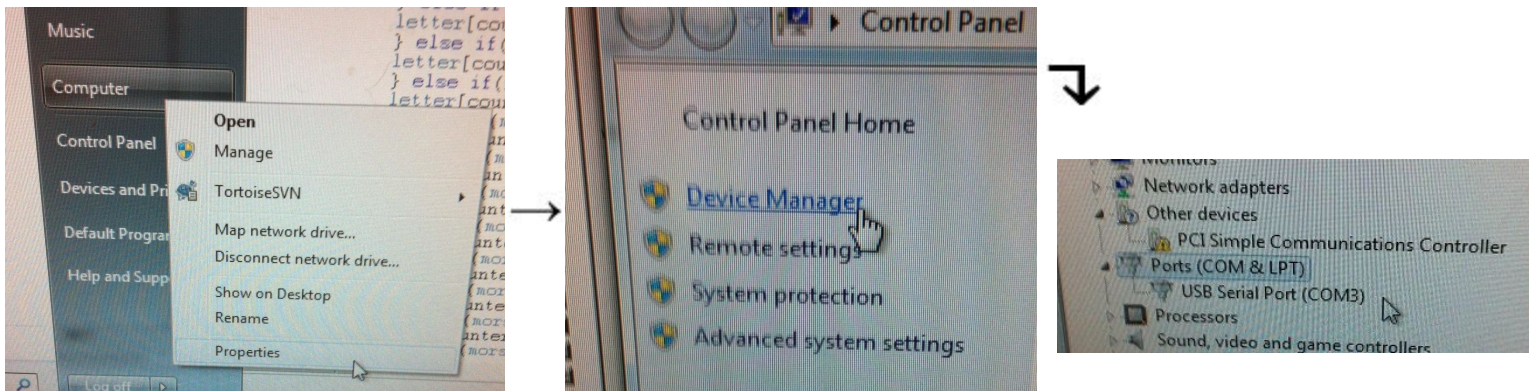
5. Upload the compiled program to the board using³

³ Xu, Jia. "EECS4352 Lab 1" York University. N.p., n.d. Web. 20 Mar. 2017.

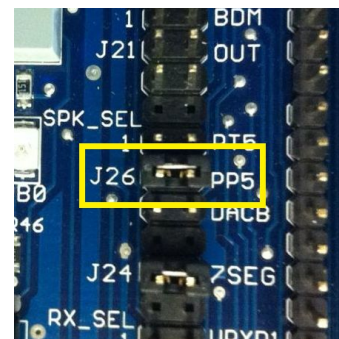
6. If it is the first time running the program, then setup the HOST Serial Communication Port to the same port as under Port settings⁴.



The following steps show the current port on the computer:




7. Place J26 to PP5 on the board to enable speaker;



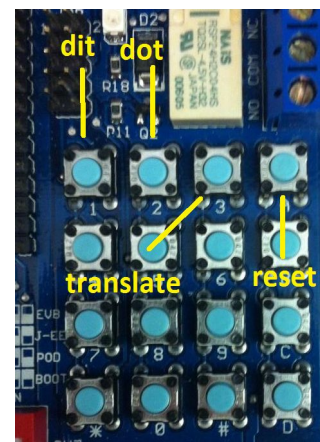
⁴ Xu, Jia. "EECS4352 Lab 1" York University. N.p., n.d. Web. 20 Mar. 2017.

3.2 Testing




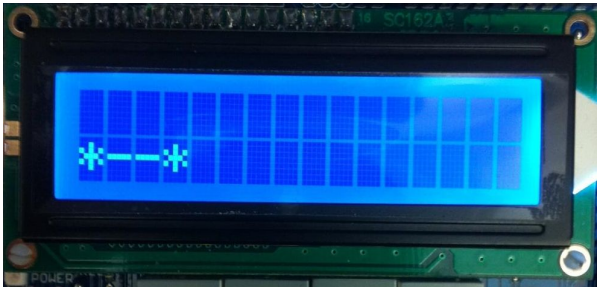
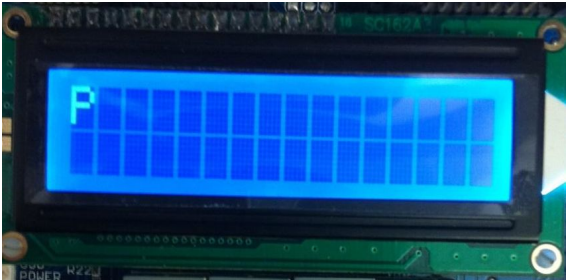

1. After uploading the program to the board, click  on debugger window to execute the code.
2. The following menu should show on the 16x2 LCD display module in consecutive order;



3. As the menu suggests, 1 on keypad inputs dit ("-"), 2 inputs dot ("*"), 3 translates Morse Code to english alphabet and 4 resets the LED display.



4. The following inputs test the implementation of Morse Code;

Input	Output
<p>All alphabet letters and numbers as input in the form of dits and dots as shown;</p> <div><div><div>A●■■■</div><div>B■■■●●●</div><div>C■■■●■■■●</div><div>D■■■●●●</div><div>E●</div><div>F●●■■■●●</div><div>G■■■●■■■●</div><div>H●●●●●●</div><div>I●●●</div><div>J●■■■●■■■</div><div>K■■■●■■■</div><div>L■■■●■■■●</div><div>M■■■●■■■</div><div>N■■■●■■■</div><div>O■■■●■■■</div><div>P●■■■●■■■●</div><div>Q■■■●■■■●■■■</div><div>R■■■●■■■●</div><div>S●●●●●</div><div>T■■■</div></div><div><div>U●●■■■</div><div>V●●●■■■</div><div>W●■■■</div><div>X■■■●■■■</div><div>Y■■■●■■■●■■■</div><div>Z■■■●■■■●■■■</div></div><div><div>1●■■■●■■■●■■■</div><div>2●●■■■●■■■●■■■</div><div>3●●●■■■●■■■●■■■</div><div>4●●●●■■■●■■■</div><div>5●●●●●■■■</div><div>6■■■●●●●■■■</div><div>7■■■●■■■●■■■</div><div>8■■■●■■■●●■■■</div><div>9■■■●■■■●■■■●</div><div>0■■■●■■■●■■■●■■■</div></div></div>	<div></div> <div></div> <div></div>
<p>Morse code Input: dot dit dit dot Keypad input: 2 1 1 2</p> <div></div>	<div></div>
<p>Morse Code Input: dot dit dot dit dot dot dot dit dit dot dot dot dot dot dot dit Keypad input: 21, 2122, 2112, 2222, 21</p>	<div></div>

4.0 Discussion

4.1 Design and Implementation

The following is the methods and techniques that were used in the design and implementation of the morse code translator in order to assure that the system matches its requirements.

4.1.1 Keypad

The keypad read task is responsible for keeping track of the keypad on the board. For instance noticing when a button is pressed. This is done using a method called `KeypadReadPort()`. This method scans the keypad and returns the button being pressed, or it returns `0xFF` in the case no button is being pressed. Thus, if we set a variable `key` to this method, and `key` is equal to `0xFF` then no button is being pressed. However, if `key` is not equal to `0xFF` then this means a button is being pressed. Furthermore, one can determine the number of the button being pressed using the key map like so `key_map[key]`. Using these functions one can determine when each key is being pressed, and then assign each key a function. In the case of this code when button number one is pressed it is a dot, and when button number two is pressed it is a dit. These values are being pushed onto a global variable which is being displayed on the LCD screen. Separate counters for each values being displayed, morse and letters, keep track of where to put the values on the screen. The big button within this task is three which translates the morse code to a letter. This is done by detecting when three is pressed, using the previously explained methods, and then checking the given morse code against the alphabet using hardcoded if-else statements. An example of such a statement would be `"if(morse[0] == '*' & morse[1] == '-' && morse[2] == '-' && morse[3] == '*'){letter = 'P'}"`. In the case the morse code matches a letter, the letter is written to the letters global variable. Similarly, number four, the clear button, detects when it is being pressed and simply clears both the letters global variable and the counters. Finally, a delay was added to the keypad using the `OSTimeDlyHMSM()` method. This was added as the keypad without it is too sensitive to use ideally.

4.1.2 LCD

The LCD is the means through which data is displayed to the user. This functionality is done in the LCD test task, where we begin by initializing the display with the method `Displnit()`, and then we use the method `DispStr()` to display messages. Furthermore, an important method of the LCD are the clear methods such as `DispClrLine()`, which clear the line of choice. In the case of our application, the LCD is used to display an initial welcome message, the morse code, and the letters.

4.1.3 Speaker

The speaker on the Dragon12 board allows for various sounds to be played. This speaker can be enabled by placing J26 pin to PP5 pin. Furthermore, in order for the speaker to display sound multiple variables must be set in the main task. These variables range from setting the speakers channel to adjusting the polarity. However, the two values most important to this project are PWME and PWMPRCLK. The PWME variable allows for the speaker to be turned on or off, setting it to zero is off and 0x20 is on. The PWMPRCLK variable allows one to change the pitch of the sound coming from the speaker. In the case of our application, when the keypad is idle no sound is displayed and when morse code is entered the speaker makes sound. This is accomplished using the previously explained keypad task. When the keypad is idle set PWME to zero, and when a button is pressed set it to 0x20. Furthermore, both dit and dot have unique sounds. For the pitch issue one simply needs to assign PWMPRCLK based on which button is being pressed. For example, if button number one is pressed set PWMPRCLK to five and if button number two is pressed set PWMPRCLK to one. This will allow the speaker to create a sound with a different pitch depending on which button is pressed. With regards to when other buttons are pressed it was decided to have no sound. This is due to the fact that if one were to simply listen to the sound of the dits and dots they can make out the morse code being entered. Thus, that concludes on how the speaker is used within the morse code translator.

4.1.4 Flags

In the project, group flags were utilized to protect critical sections of the code. For instance, `OS_FLAG_GRP *mlFlagGrp;` is initialized in variables section to protect the computation when translating the code. This flag post is used in static void `KeypadRdTask (void *p_arg)` method that periodically read the value from keypad and displays it on the bottom LCD display. When key 3 is pressed, series of computations take place where combination of the dit-dot input is compared and appropriate alphabet letter is matched. This is the most critical section of the system because it does computation as well as clear the stack which contained the input. Hence, the flag post `OSFlagPost(mlFlagGrp, 0x01, OS_FLAG_SET, &err);` is set at the beginning of if-else statement if `(key_map[key] == '3')` where computation takes place. Once the translation has concluded and variables `CPU_INT08U morse[16]` cleared, the flag is cleared by the following declaration: `OSFlagPost(mlFlagGrp, 0x01, OS_FLAG_CLR, &err)`. Another critical section of this project is when resetting the board. When key 4 is pressed, all the LCD displays need to be cleared which are stored in global variables: `CPU_INT08U morse[16];` and `CPU_INT08U letter[200]`. Therefore, `OSFlagPost(resetFlagGrp, 0x01, OS_FLAG_SET, &err)` was positioned inside `if (counterL > 15 || key_map[key] == '4')` which checks if key 4 is

pressed or when top display is full. This flag is cleared once global variables are emptied.

4.2 Encountered Errors

Over the course of the project, while still in the coding phase, many problems were encountered. In order to fix these problems multiple tests had to be run for each and every change made to the code. Through fixing these problems while the project wasn't working and not meeting its requirements, the correctness of the real-time embedded system application was ensured. The following are the main functionality problems that were encountered, and their respective solutions.

4.2.1 Problems with the LCD display

One of the first problems encountered was understanding how the LCD display functioned. To begin, the display was constantly flashing and not displaying crisp symbols, as can be seen in **Figure 3**. The solution to this problem was a `DispClrLine(0)` which was present inside of the infinite loop. This method was clearing the line at every instance of the loop, giving the symbols a grainy display. Furthermore, issues were encountered when the functionality of a welcome message was attempted to be implemented. The complications for this functionality were multi faceted, the display would carry over, constantly loop, and not display at ideal speeds. The solution to our welcome message problem was solved through both familiarizing oneself with the LCD manual that can be found at the Micrium OS website, and multiple tests to ensure the message was displaying correctly.

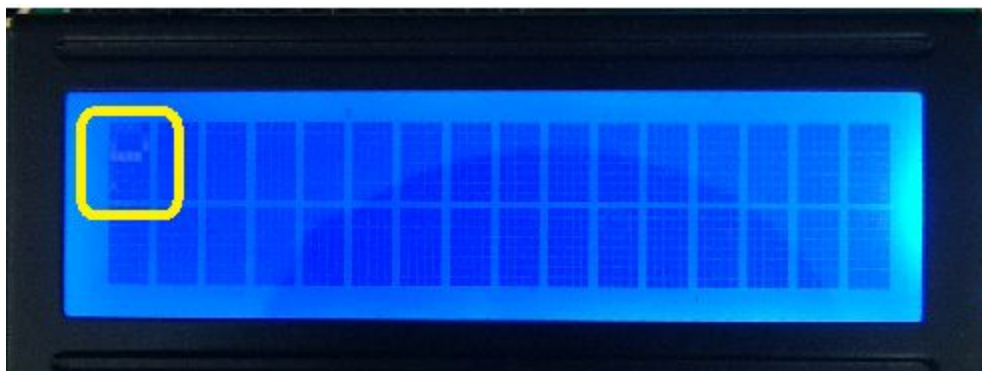


Figure 3: The letter 'P' blurry on the LCD display

4.2.2 Problems with handling user errors

The second set of problems encountered when developing the application was optimizing the application to be more user friendly. For one, when the top row of the device, which displays letters, would become full the application would cease to work. The first attempt to mend this error was to clear the line and reset the counters tracking where on the screen the letters should go. However, the old letter values would remain on the screen and simply be overridden by the newly added letters. To resolve this issue it was needed to implement a loop which would clear the letter variable on display, and then reset the counters. However, further tests of the application revealed that upon filling the top row a second time would once again break the application. This issue was easily solved by resetting the loop counter of the loop that cleared the letter variable. The simple fix, combined with previous solutions, resulted in the top row properly functioning when full. That is, it clears to give way to new letters to be inputted.

4.3 Performance and Correctness

In order to ensure the correctness and the performance of the morse code machine multiple physical test were ran. To begin, every symbol was translated from morse code, this ensures that not only can the application display said letters and numbers, but it also shows that each group of morse code symbols correctly translate to their respective letter or number. The tests then moved on to ensuring that the safeguards put in place functioned correctly. This meant filling the top row with letters multiple times to ensure that the row would clear when full. The bottom bar, for morse code, was also tested repeatedly and in different circumstances. Ensuring that the bottom bar clears in various scenarios means that the user cannot enter more morse symbols than the allotted letters require. To further test the functionality of the machine, false morse code was entered to ensure that the false morse code did not lead to an unwanted letter. Of course, every permutation of false morse code can't be tested. However, with over 10 cases of false morse code being tested, one can deduce that the likelihood of false morse code producing a letter is quite minimal. Finally, the functionality of each button was tested. Focusing more on the clear button, as dit, dot, and translate are quite trivial to test. The clear button, number 4 on the keypad, was tested to work regardless of how many letters are present on the screen. Furthermore, it was tested to work with various amounts of morse code present on the screen. These tests ensured that the clear functionality was quite robust, and was unlikely to not function. With all of these test's the applications correctness can be deduced. Not only does every functionality work in various situations, the safeguards in place also work ensuring the application never breaks.

The performance of the application was tested using the Sevensegdisplay. With the clock being displayed one can time the response time of the applications different functions. This can be done by going into the Sevensegdisplay task and changing the code such that the timer begins when the print button is pushed, and ends when the value is displayed. This can be done by tracking when the print button is pressed with the line `key_map[key] == '3'`. For tracking when the item is being displayed, one simply keeps track of the letters variable, and stops the timer when a value is added. Using this method the values in **table 1** were generated for the print button. As can be seen, the average time is under a second. This is a reasonable response time for a translator as it is not very noticeable to the naked eye.

	1	2	3	4	5	6	7	8	9
Time(s)	0.5	0.6	0.54	0.49	0.66	0.52	0.42	0.69	0.52

Table 1: Print button response time

5.0 Application

Before the invention of telegraphs, messages were handwritten and carried by horseback. After the invention, morse code was the fastest long distance form of communication. It was pivotal during world war two as it improved the speed of communication while maintaining secrecy of the message from the enemy. It not only plays key a role for military units on land but also for ship-to-shore communication (**Figure 4**). Although, today morse code is not as widely used as it once was, the US navy still uses it via sounds and signal lamps. This is an alternative for places that do not have internet to establish communication. In cases where soldiers need to send distress signals they are taught to rig up a simple transmitter in remote locations and send distress call⁵.



Figure 4: Navy soldier sending morse code signal.

⁵ Moss, Stephen. "Stephen Moss on the use of Morse code today." The Guardian. Guardian News and Media, 21 May 2008. Web. 01 Apr. 2017.

Besides military applications, it also serves as a communication device for people diagnosed with motor neuron disease such as amyotrophic lateral sclerosis disease. Patients with such condition suffer from muscle degeneration which leaves them wheelchair bound and inability to speak. Morse code allows these individuals to communicate with just push of a button. Furthermore, the deaf-blind individuals are also able to take advantage of this device when face the issue of interacting with the population around them⁶. This adaptive communication device is able to relay messages through two mediums if one should fail; audibly via distinct sounds for “-” and “*” or alphabetical display on LCD.

6.0 Conclusion

In conclusion, this project incorporates concepts learned throughout the lab sessions in the course. Signals of dit-dot combinations were inputted by pressing appropriate button and were then decoded in mikroC real-time kernel. This analog to digital conversion raised many design-related questions which were addressed and resolved as documented. The debugging tool provided in CodeWarrior IDE was used to monitor step by step execution of the program code to rectify problems. Furthermore, this implementation of morse code has military and assistive technology applications. By utilizing the different features of the Dragon12 board, morse code could be decoded visually or by audio. Overall, this project demonstrates how mikroC operating system and the Dragon12 micro-controller can be design and implement a real-time embedded system application.

⁶ "Morse code for the Deaf/Blind and disabled." CommonWealth. N.p., n.d. Web. 1 Apr. 2017.

7.0 References

Sařsa and Madelaine Page Milić. "Morse Code Decoder." *Micro-Controller Applications*. N.p., 3 Dec. 2013. Web. 1 Mar. 2017.

Mohd Ikmal Bin Amran. "Morse Code Reader." Rage University. N.p., Nov. 2012. Web. 1 Apr. 2017.

A. P. Thakare. "Morse Code Decoder - Using a PIC Microcontroller." IJSETR. N.p., Nov. 2012. Web. 1 Apr. 2017.

Treasure, Novel. "Is Morse code used Today? - The Brief History and Importance of Morse Code." Owlcation. Owlcation, 22 Jan. 2015. Web. 01 Apr. 2017.

Moss, Stephen. "Stephen Moss on the use of Morse code today." The Guardian. Guardian News and Media, 21 May 2008. Web. 01 Apr. 2017.

"Morse code for the Deaf/Blind and disabled." CommonWealth. N.p., n.d. Web. 1 Apr. 2017.

Xu, Jia. "uCOS-II-RefMan" York University. N.p., n.d. Web. 20 Mar. 2017.

Xu, Jia. "EECS4352 Lab 1" York University. N.p., n.d. Web. 20 Mar. 2017.

Xu, Jia. "an-1456_c_os-ii_dragon12_development_board_" York University. N.p., n.d. Web. 20 Mar. 2017.

8.0 Appendix

```
/*
*****
*
*          uC/OS-II
*      The Real-Time Kernel
*
*      (c) Copyright 1998-2003, Jean J. Labrosse, Weston, FL
*      All Rights Reserved
*
*      Sample code
*      MC9S12DP256B
*      Wytec Dragon12 EVB
*
* File : app.c
* By   : Eric Shufro
*****
*/

#include <includes.h>

/*
*****
*
*          DEFINES
*
*****
*/

/*
*****
*
*          CONSTANTS
*
*****
*/

/*
*****
*
*          VARIABLES
*
*****
*/
```

```

*/

OS_STK    AppStartTaskStk[APP_TASK_START_STK_SIZE];
OS_STK    LCD_TestTaskStk[LCD_TASK_STK_SIZE];
OS_STK    SevenSegTestTaskStk[SEVEN_SEG_TEST_TASK_STK_SIZE];
OS_STK    KeypadRdTaskStk[KEYPAD_RD_TASK_STK_SIZE];

CPU_INT08U morse[16];    /* Morse code variable */
CPU_INT08U letter[200]; /* Letter variable */

OS_FLAG_GRP *keypadEnFlagGrp; /* Flags */
OS_FLAG_GRP *mIFlagGrp;    /* Flag to lock the global variables */
OS_FLAG_GRP *resetFlagGrp;

/*
*****
*
*                      FUNCTION PROTOTYPES
*
*****
*/

static void AppStartTask(void *p_arg);
static void AppTaskCreate(void);
static void LCD_TestTask(void *p_arg);
static void SevenSegTestTask(void *p_arg);
static void KeypadRdTask(void *p_arg);

#if (uC_PROBE_OS_PLUGIN > 0) || (uC_PROBE_COM_MODULE > 0)
extern void AppProbeInit(void);
#endif

/*
*****
*
*                      main()
*
* Description : This is the standard entry point for C code. It is assumed that your code will call

```



```

*      main() once you have performed all necessary 68HC12 and C initialization.
* Arguments  : none
*****
*/

void main (void)
{
    INT8U  err;

    OSInit();                      /* Initialize "uC/OS-II, The Real-Time Kernel"
*/

    OSTaskCreateExt(AppStartTask,
        (void *)0,
        (OS_STK *)&AppStartTaskStk[APP_TASK_START_STK_SIZE - 1],
        APP_TASK_START_PRIO,
        APP_TASK_START_PRIO,
        (OS_STK *)&AppStartTaskStk[0],
        APP_TASK_START_STK_SIZE,
        (void *)0,
        OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);

    OSTaskNameSet(APP_TASK_START_PRIO, "Start Task", &err);

    //ClockA=Fbus/2**4=24MHz/16=1.5MHz 2 or 0
    PWMSCLA=139; //ClockSA=1.5MHz/2x125=6000 Hz
    PWMCLK=0b00100000; //ClockSA for chan 5
    PWMPOL=0x20;  //high then low for polarity
    PWMCAE=0x0;   //left aligned
    PWMCTL=0x0;   //8-bit chan, pwm during freeze and wait
    PWMPER5=165;
    PWMDTY5=50;
    PWMCNT5=0; //clear initial counter. This is optional

    OSStart();                      /* Start multitasking (i.e. give control to uC/OS-II) */
}

```

```

/*$PAGE*/
/*
*****

*
*
*
* Description : This is an example of a startup task. As mentioned in the book's text, you MUST
* initialize the ticker only once multitasking has started.
*
* Arguments : p_arg is the argument passed to 'AppStartTask()' by 'OSTaskCreate()'.
*
* Notes : 1) The first line of code is used to prevent a compiler warning because 'p_arg' is
not
* used. The compiler should not generate any code for this statement.
* 2) Interrupts are enabled once the task start because the I-bit of the CCR register was
* set to 0 by 'OSTaskCreate()'.
* 3) After this created from main(), it runs and initializes additional application
* modules and tasks. Rather than deleting the task, it is simply suspended
* periodically. This tasks body could be used for additional work if desired.
*****
*/

static void AppStartTask (void *p_arg)
{
    (void)p_arg;

    BSP_Init(); /* Initialize the ticker, and other BSP related
functions */

    #if OS_TASK_STAT_EN > 0
        OSStatInit(); /* Start stats task */
    #endif

    #if (uC_PROBE_OS_PLUGIN > 0) || (uC_PROBE_COM_MODULE > 0)
        AppProbeInit(); /* Initialize uC/Probe modules
*/
    #endif

    AppTaskCreate(); /* Create additional tasks using this user
defined function */

```

```

    while (TRUE) {                                /* Task body, always written as an infinite loop
*/
        OSTimeDlyHMSM(0, 0, 5, 0);                /* Delay the task
*/
    }
}

/*$PAGE*/
/*
*****
*
*               CREATE APPLICATION TASKS
*
* Description : This function demonstrates how to create a new application task.
*
* Notes:    1) Each task should be a unique function prototypes as
*            static void mytaskname (void *p_arg).
*            2) Additionally, each task should contain an infinite loop and call at least one
*            OS resource on each pass of the loop. An OS resource may be a call to OSTimeDly(),
*            OSTimeDlyHMSM(), or one of the message box, semaphore or other OS handled
resource.
*            3) Each task must have its own stack. Be sure that the stack is declared large
*            enough or the entire system may crash or experience erratic results if your stack
*            grows and overwrites other variables in memory.
*
* Arguments : none
* Notes    : none
*****
*/

static void AppTaskCreate (void)
{
    INT8U err;

    OSTaskCreateExt(LCD_TestTask,
                    (void *)0,
                    (OS_STK *)&LCD_TestTaskStk[LCD_TASK_STK_SIZE-1],

```

```

        LCD_TEST_TASK_PRIO,
        LCD_TEST_TASK_PRIO,
        (OS_STK *)&LCD_TestTaskStk[0],
        LCD_TASK_STK_SIZE,
        (void *)0,
        OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
OSTaskNameSet(LCD_TEST_TASK_PRIO, "LCD Test Task", &err);

OSTaskCreateExt(SevenSegTestTask,
        (void *)0,
        (OS_STK *)&SevenSegTestTaskStk[SEVEN_SEG_TEST_TASK_STK_SIZE-1],
        SEVEN_SEG_TEST_TASK_PRIO,
        SEVEN_SEG_TEST_TASK_PRIO,
        (OS_STK *)&SevenSegTestTaskStk[0],
        SEVEN_SEG_TEST_TASK_STK_SIZE,
        (void *)0,
        OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
OSTaskNameSet(SEVEN_SEG_TEST_TASK_PRIO, "SevenSegTest Task", &err);

OSTaskCreateExt(KeypadRdTask,
        (void *)0,
        (OS_STK *)&KeypadRdTaskStk[KEYPAD_RD_TASK_STK_SIZE-1],
        KEYPAD_RD_TASK_PRIO,
        KEYPAD_RD_TASK_PRIO,
        (OS_STK *)&KeypadRdTaskStk[0],
        KEYPAD_RD_TASK_STK_SIZE,
        (void *)0,
        OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
OSTaskNameSet(KEYPAD_RD_TASK_PRIO, "KeypadRd Task", &err);
}

/*$PAGE*/
/*
*****
*
*           SevenSegWriteTask
*
* Description: This task displays messages on the Dragon12 (16x2) LCD screen and is
*             responsible for initializing the LCD hardware. Care MUST be taken to

```

```

*      ensure that the LCD hardware is initialized before other tasks
*      attempt to write to it. If necessary the Displnit() function
*      may be called from the start task or bsp_init().
*****
*/

static void LCD_TestTask (void *p_arg)
{
    CPU_INT08S i;
    CPU_INT08U err;
    CPU_INT08U welcome = 0;

    /* Power On Welcome Message. three seperate msgs /
rows */
const CPU_INT08U WelcomeStr[6][18] = {"Morse code", "translator",
    "1 = dit  ", "2 = dot  ",
    "3 = translate", "4 = reset"};

    (void)p_arg;

    Displnit(2, 16); /* Initialize uC/LCD for a 2 row by 16 column
display */

    while (DEF_TRUE) { /* All tasks bodies include an infinite loop
*/

        if (welcome == 0){
            /* If statement to ensure the welcome message only
starts once */
            for (i = 0; i < 6; i+=2) { /* With the Keypad task suspended, both LCD rows are
avail. */
                DispStr(0, 0, WelcomeStr[i]); /* Display row 0 of Welcome Message i
*/
                DispStr(1, 0, WelcomeStr[i+1]); /* Display row 1 of Welcome Message i
*/
                OSTimeDlyHMSM(0, 0, 2, 0); /* Delay between updating the message
*/
            }
        }
    }
}

```



```

    }
    DispClrLine(1);
    DispClrLine(0);
    welcome = welcome + 1;          /* Increment the welcome counter so that
it never runs again */
    }

```

```

    OSFlagPost(keypadEnFlagGrp, 0x01, OS_FLAG_SET, &err);    /* Set flag bit 0 of the
keypad enable flag group */

    /* This will unblock the keypad task which will use the
    /* bottom row of the LCD while not disabled
    */

```

```

    while (err != OS_NO_ERR) {          /* If a flag posting error occurred,
    */
        OSTimeDlyHMSM(0, 0, 1, 0);      /* delay and try again until NO ERROR is
returned */
        OSFlagPost(keypadEnFlagGrp, 0x01, OS_FLAG_SET, &err);    /* Set flag bit 0 of the
keypad enable flag group */
    }

```

```

    /* Display the morse code in the bottom row of the
LCD */
    DispStr(1, 0, morse);                /* Display the letters in the top row of the LCD
    */
    DispStr(0, 0, letter);

```

```

    OSFlagPost(keypadEnFlagGrp, 0x01, OS_FLAG_CLR, &err);    /* Disable the keypad task
by clearing its event flag 0 */

```

```

        while (err != OS_NO_ERR) {                                /* If an error occurred (perhaps the flag
group is not */
            OSTimeDlyHMSM(0, 0, 1, 0);                            /* initialized yet, then delay and try again
until no error */
            OSFlagPost(keypadEnFlagGrp, 0x01, OS_FLAG_CLR, &err); /* Clear flag bit 0 of the
keypad enable flag group */
        }

    }
}

/*$PAGE*/
/*
*****
*
*                               SevenSegWriteTask
*
* Description: This task displays messages on the Dragon12 (16x2) LCD screen
*****
*/

static void SevenSegTestTask(void *p_arg)
{
    CPU_INT16U num;

    (void)p_arg;

    SevenSegDisp_Init();                                          /* Initialize the 7-Seg I/O and periodic refresh
interrupt */

    num = 0;

    while (DEF_TRUE) {                                           /* All tasks bodies include an infinite loop
*/
        SevenSegWrite(num);                                       /* Output the value to the screen
*/
        num = ((num + 1) % 10000);                                /* Increment the # being displayed, wrap
after 9,999 */
    }
}

```

```

        OSTimeDlyHMSM(0, 0, 0, 10);                /* Delay the task for 50ms and repeat
the process */
    }
}

/*$PAGE*/
/*
*****
*
*                               Keypad Read Task
*
* Description: This task periodically reads the Wytec Dragon12 EVB keypad and
*              displays the value on the bottom row of the LCD screen.
*****
*/

static void KeypadRdTask (void *p_arg)
{
    CPU_INT08U key;
    CPU_INT08U i = 0;
    CPU_INT08U delay;
    CPU_INT08U x = 0;
    CPU_INT08U out_str[17];
    CPU_INT08U key_map[] = {'1', '2', '3', 'A',
                            '4', '5', '6', 'B',
                            '7', '8', '9', 'C',
                            '*', '0', '#', 'D'
                            };
    CPU_INT08U err;
    CPU_INT08U counter = 0;
    CPU_INT08U counterL = 0;                        /* Counter to keep track of the morse
code */

                                                    /* Counter to keep track of the letters */

    (void)p_arg;

    KeypadInitPort();                               /* Initialize the keypad hardware
*/

```

```

    keypadEnFlagGrp = OSFlagCreate(0, &err);          /* Create an event flag group. All
flags initialized to 0 */

    while (err != OS_NO_ERR) {                        /* If an error code was returned, loop until
successful */
        OSTimeDlyHMSM(0, 0, 1, 0);                  /* Delay for 1 second, wait for resources
to be freed */
        keypadEnFlagGrp = OSFlagCreate(0, &err);      /* Try to create the flag group
again */
    }

    OSFlagPend(keypadEnFlagGrp, 0x01, OS_FLAG_WAIT_SET_ALL, 0, &err); /* Wait until bit 1
of the flag group to become set */

    /* The goal is to prevent this task from accessing the
    /* bottom row of the LCD until the LCD task has
finished */
    /* displaying its introduction message. This message
will */
    /* require both lines of the LCD, so we must wait to use
it */

    DispClrLine(1);                                  /* Clear the bottom line of the 2 line display
    */

    while (DEF_TRUE) {                                /* All tasks bodies include an infinite loop
    */
        OSFlagPend(keypadEnFlagGrp, 0x01, OS_FLAG_WAIT_SET_ALL, 0, &err); /* Suspend the task
if flag 0 has been cleared */
        key = KeypadReadPort();                      /* Scan the keypad. Returns 0-15 or 0xFF if
nothing pushed */
        if (key == 0xFF){                             /* If nothing is being pressed */
            PWME = 0; // Sound off                    /* Speaker sound is turned off
        */
        }
        else{                                           /* Else if a button is pressed */
            PWME = 0x20; // Sound on                    /* Speaker sound is turned on when a dit
or dot is pressed */

```

```

        if(key_map[key] == '1'){
            morse[counter] = '-';
counter    */
            counter = counter + 1;
value is displayed correctly */
            PWMPRCLK=0;
        */
        }
        if(key_map[key] == '2'){
            morse[counter] = '*';
counter    */
            counter = counter + 1;
value is displayed correctly */
            PWMPRCLK=2;
        */
        }
        if (counterL > 15 || key_map[key] == '4'){
presed on the keypad*/
            OSFlagPost(resetFlagGrp, 0x01, OS_FLAG_SET, &err)
            PWME = 0;
        */
            DispClrScr();
            counterL = 0;
begining of display */
            counter = 0;
tracked correctly */
            while (x < 17){
                letter[x] = 0;
                x = x + 1;
            }
            x = 0;
in one use */
            OSFlagPost(resetFlagGrp, 0x01, OS_FLAG_CLR, &err)
        }

        if (counter > 5 ){
            DispClrLine(1);
            morse[0] = 0;

```

/* Press 1 and it's a dit "-" */
 /* Place a dit in the morse display at location
 /* Increment the counter so that the next
 /* This value changes the pitch of the speaker
 /* Press 2 and it's a dot "*" */
 /* Place a dot in the morse display at location
 /* Increment the counter so that the next
 /* This value changes the pitch of the speaker
 /* If there are too many letters or 4 is
 /* No sound when 4 is pressed
 /* Clear the screen
 /* Clear the counter so that it returns to the
 /* Clear the counter so that the morse code is
 /* Clear the contents of letter*/
 /* Clear x so that this function works more then once


```

    morse[1] = 0;
    morse[2] = 0;
    morse[3] = 0;
    morse[4] = 0;
    morse[5] = 0;
    counter = 0;                                /* Reset the counter */
}
if (key_map[key] == '3'){                        /* Press 3 to display letter */
    PWME = 0;
    OSFlagPost(mIFlagGrp, 0x01, OS_FLAG_SET, &err);    /* Lock the global variables*/
    DispClrLine(1);
    if(morse[0] == '*' & morse[1] == '-' && morse[2] == '-' && morse[3] == '*'){ /* Check
which number or letter the morse code represents */
        letter[counterL] = 'P';
    } else if(morse[0] == '-' & morse[1] == '-' && morse[2] == '*' && morse[3] == '*' &&
morse[4] == ""){
        letter[counterL] = 'Z';                                /* Put this letter into
the letter variable */
    } else if(morse[0] == '-' & morse[1] == '*' && morse[2] == '-' && morse[3] == '-' &&
morse[4] == ""){
        letter[counterL] = 'Y';
    } else if(morse[0] == '-' & morse[1] == '*' && morse[2] == '*' && morse[3] == '-' &&
morse[4] == ""){
        letter[counterL] = 'X';
    } else if(morse[0] == '*' & morse[1] == '-' && morse[2] == '*' && morse[3] == '*' &&
morse[4] == ""){
        letter[counterL] = 'L';
    } else if(morse[0] == '*' & morse[1] == '*' && morse[2] == '*' && morse[3] == '-' &&
morse[4] == ""){
        letter[counterL] = 'V';
    } else if(morse[0] == '*' & morse[1] == '-' && morse[2] == '-' && morse[3] == '-' &&
morse[4] == ""){
        letter[counterL] = 'J';
    } else if(morse[0] == '-' & morse[1] == '*' && morse[2] == '*' && morse[3] == '*' &&
morse[4] == ""){
        letter[counterL] = 'B';
    } else if(morse[0] == '-' & morse[1] == '*' && morse[2] == '-' && morse[3] == '*' &&
morse[4] == ""){
        letter[counterL] = 'C';
    }
}

```

```

        } else if(morse[0] == '*' & morse[1] == '*' && morse[2] == '-' && morse[3] == '*' &&
morse[4] == ""){
            letter[counterL] = 'F';
        } else if(morse[0] == '*' & morse[1] == '*' && morse[2] == '*' && morse[3] == '*' &&
morse[4] == ""){
            letter[counterL] = 'H';
        } else if(morse[0] == '-' & morse[1] == '-' && morse[2] == '*' && morse[3] == '-' &&
morse[4] == ""){
            letter[counterL] = 'Q';
        } else if(morse[0] == '*' & morse[1] == '*' && morse[2] == '*' && morse[3] == " &&
morse[4] == ""){
            letter[counterL] = 'S';
        } else if(morse[0] == '*' & morse[1] == '-' && morse[2] == '-' && morse[3] == " &&
morse[4] == ""){
            letter[counterL] = 'W';
        } else if(morse[0] == '*' & morse[1] == '*' && morse[2] == '-' && morse[3] == " &&
morse[4] == ""){
            letter[counterL] = 'U';
        } else if(morse[0] == '-' & morse[1] == '-' && morse[2] == '-' && morse[3] == " &&
morse[4] == ""){
            letter[counterL] = 'O';
        } else if(morse[0] == '-' & morse[1] == '*' && morse[2] == '-' && morse[3] == " &&
morse[4] == ""){
            letter[counterL] = 'K';
        } else if(morse[0] == '*' & morse[1] == '-' && morse[2] == '*' && morse[3] == " &&
morse[4] == ""){
            letter[counterL] = 'R';
        } else if(morse[0] == '-' & morse[1] == '-' && morse[2] == '*' && morse[3] == " &&
morse[4] == ""){
            letter[counterL] = 'G';
        } else if(morse[0] == '-' & morse[1] == '*' && morse[2] == '*' && morse[3] == " &&
morse[4] == ""){
            letter[counterL] = 'D';
        } else if(morse[0] == '*' & morse[1] == '*' && morse[2] == " && morse[3] == " &&
morse[4] == ""){
            letter[counterL] = 'I';
        } else if(morse[0] == '-' & morse[1] == '-' && morse[2] == " && morse[3] == " &&
morse[4] == ""){
            letter[counterL] = 'M';

```

```

        } else if(morse[0] == '-' & morse[1] == '*' && morse[2] == " && morse[3] == " &&
morse[4] == "){
            letter[counterL] = 'N';
        } else if(morse[0] == '*' && morse[1] == '-' && morse[2] == " && morse[3] == " &&
morse[4] == "){
            letter[counterL] = 'A';
        } else if(morse[0] == '*' && morse[1] == " && morse[2] == " && morse[3] == " &&
morse[4] == "){
            letter[counterL] = 'E';
        } else if(morse[0] == '-' && morse[1] == " && morse[2] == " && morse[3] == " &&
morse[4] == "){
            letter[counterL] = 'T';
        } else if(morse[0] == '*' && morse[1] == '*' && morse[2] == '-' && morse[3] == '-' &&
morse[4] == '-'){
            letter[counterL] = '2';
        } else if(morse[0] == '*' && morse[1] == '*' && morse[2] == '*' && morse[3] == '-' &&
morse[4] == '-'){
            letter[counterL] = '3';
        } else if(morse[0] == '*' && morse[1] == '*' && morse[2] == '*' && morse[3] == '*' &&
morse[4] == '-'){
            letter[counterL] = '4';
        } else if(morse[0] == '*' && morse[1] == '*' && morse[2] == '*' && morse[3] == '*' &&
morse[4] == '*'){
            letter[counterL] = '5';
        } else if(morse[0] == '-' && morse[1] == '*' && morse[2] == '*' && morse[3] == '*' &&
morse[4] == '*'){
            letter[counterL] = '6';
        } else if(morse[0] == '-' && morse[1] == '-' && morse[2] == '*' && morse[3] == '*' &&
morse[4] == '*'){
            letter[counterL] = '7';
        } else if(morse[0] == '-' && morse[1] == '-' && morse[2] == '-' && morse[3] == '*' &&
morse[4] == '*'){
            letter[counterL] = '8';
        } else if(morse[0] == '-' && morse[1] == '-' && morse[2] == '-' && morse[3] == '-' &&
morse[4] == '*'){
            letter[counterL] = '9';
        } else if(morse[0] == '*' && morse[1] == '-' && morse[2] == '-' && morse[3] == '-' &&
morse[4] == '-'){
            letter[counterL] = '1';

```

```

        } else if(morse[0] == '-' && morse[1] == '-' && morse[2] == '-' && morse[3] == '-' &&
morse[4] == '-'){
        letter[counterL] = '0';
        }

        if(letter[counterL] == ""){ /* If the morse code does not match a letter, do not
increment the counterL */

        } else{
        counterL = counterL + 1; /* If the morse code does match, continue on*/
        }

        morse[0] = 0; /* Clear the morse code variable for the next letter*/
        morse[1] = 0;
        morse[2] = 0;
        morse[3] = 0;
        morse[4] = 0;
        counter = 0;

        OSFlagPost(mfFlagGrp, 0x01, OS_FLAG_CLR, &err); /* Unlock the global variables */
    }
}
OSTimeDlyHMSM(0, 0, 0,100); /* Read the keypad every 100ms, makes
it less sensitive when pressing on the keypad */
}
}

```