UNIVERSITY OF CALIFORNIA, SANTA CRUZ
BOARD OF STUDIES IN COMPUTER ENGINEERING

CMPE-13/L: COMPUTER SYSTEMS AND "C" PROGRAMMING

Lab 8 - Morse Code

Commit ID Form: https://goo.gl/forms/eMSxwxVcf6LxOjNn1

17 Points

Introduction

This lab involves Morse code decoding and encoding using the terminal, OLED, and push buttons on the Basic I/O Shield. Working implementations of the OLED and Buttons libraries are provided for you. This lab builds on your previous experience with event-driven programming and state machines and expands that to the binary tree data structure and light recursion.

Reading

- Morse code article on Wikipedia

Concepts

- Pointers and malloc()
- Timers
- Finite state machines
- Abstract Data Type: Binary trees
- Recursion

Provided files

- Morse.h – This file contains the API that you will be implementing in the aptly named Morse.c file. It will use a Morse tree to decode Morse code strings. Encoding functionality is also provided. This library will entirely wrap the Buttons library, such that Lab8.c will only need to include Morse.h to do everything related to Morse code functionality.
- BOARD.c/h - Contains initialization code for the UNO32 along with standard #defines and system libraries used. Also includes the standard fixed-width datatypes and error return values.
- Tree.h – This file declares an interface for creating and working with binary trees. You will be implementing the corresponding Tree.c file.
- Lab8SupportLib.a (Buttons.h, Oled.h) – These files provide helper functions that you will need to use within your code. The header files contain comments that should clarify how the code is to be used.

- Lab8.c - This file includes `main()` where you will implement the bulk of this lab's functionality.s.

Assignment requirements

- **Binary tree library:** Implement the functions defined in Tree.h in a corresponding Tree.c.
  - All functions must use recursion when traversing the tree.
  - Any constants used must be declared as an enum type or #defined.
  - No user input/output should be done within this library!
- **Morse code library:** Implement all of the functions defined in Morse.h.
  - Morse code input must be decoded using a Morse tree implemented using the functionality of Tree.h.
  - It should decode all alphanumeric characters, representing alphabetic characters as only their upper-case representation.
  - All constants and enums in Morse.h must be used and cannot be redeclared in Morse.c.
  - Any additional constants used must be declared as an enum type or #defined.
  - This library ends up being a wrapper around the Buttons library, so that lab8.c only needs to run the functions listed in Morse.h, and no Buttons functions directly.
  - No user input/output should be done within this library!
- **main():** Your main() within lab8.c must implement the following functionality using the provided code and your Morse code library.
  - Initialize the Morse decoder and check its return value printing out an error message if and calling `FATAL_ERROR()` if it fails.
  - Use Timer2's ISR to check for Morse events at 100Hz.
  - The Buttons library should not be used directly within lab8.c, as all necessary Buttons functionality is contained entirely within the Morse library.
  - There should be no terminal input or output in your submitted program.
  - Create 3 static helper functions within lab8.c:
    - A static function that clears the top line of the OLED and updates it.
    - A static function that appends a character to the top line and updates the OLED.
    - A static function that appends a character to the bottom line and updates the OLED.
  - Use the Morse library to read Morse events, which are button presses on BTN4. Depending on if a DOT, DASH, END_CHAR, or a SPACE is input, the top and bottom OLED should be updated.

    - The user can input DOTs and DASHes by varying how long they press down BTN4 to specify the Morse code representation of an alphanumeric character. These are shown on the bottom line of the OLED as they're entered.
    - If the button is unpressed for more than .1 seconds, it signals the end of decoding an alphanumeric character. If the button is pressed again, the current character should be decoded and the top line cleared.
    - If the button continues to be unpressed for more than .2 seconds, the current character should be decoded and the top line cleared while also adding a space after the character.
    - `malloc()` (or `calloc()`) will be used in this lab to generate the Morse tree, so add a comment to your README.txt file indicating the necessary heap size for correct program functionality. If you're unsure how much to start with, try 1024 to start and move up from there.
- **Extra Credit (1 point):** When displaying decoded characters, once the 2nd line is full of characters, new characters should be appended to the 3rd row and then the 4th. Once there is no more room on the 4th row, new characters should appear at the last position on the 4th row, the first character on the 4^{th} row should be pushed onto the end of the third row, the first character on the 3^{rd} row should be pushed onto the end of the second, and the first character in the 2nd row should be pushed off to the left. These changes should only require modifications to your function for appending characters to the bottom line.
- **Code style:** Follow the standard style formatting procedures for syntax, variable names, and comments.
- **Readme:** Create a file named README.txt containing the following items. Note that spelling and grammar count as part of your grade so you'll want to proof-read this before submitting. This will follow the same rough outline as a lab report for a regular science class. It should be on the order of three paragraphs with several sentences in each paragraph.
    - First you should list your name and the names of anyone else who you have collaborated with.
        - NOTE: collaboration != copying
    - In the next section you should provide a summary of the lab in your own words. Highlight what you thought were the important aspects of the lab. If these differ from how the lab manual presents things, make a note of that.
    - The following section should describe your approach to the lab. What was your general approach to the lab? Did you read the manual first or what were your first steps? What went wrong as you worked through it? What worked well? How would you approach this lab differently if

you were to do it again? Did you work with anyone else in the class? How did you work with them and what did you find helpful/unhelpful?

- The final section should describe the results of you implementing the lab. How did it end up finally? How many hours did you end up spending on it? What'd you like about it? What did you dislike? Was this a worthwhile lab? Do you have any suggestions for altering it to make it better? What were the hardest parts of it? Did the points distribution for the grading seem appropriate? Did the lab manual cover the material in enough detail to start you off? Did examples or discussions during class help you understand this lab or would more teaching on the concepts in this lab help?

- **Submission:**
  - Submit Morse.c, Tree.c, README.txt, and lab8.c before the deadline.

Grading

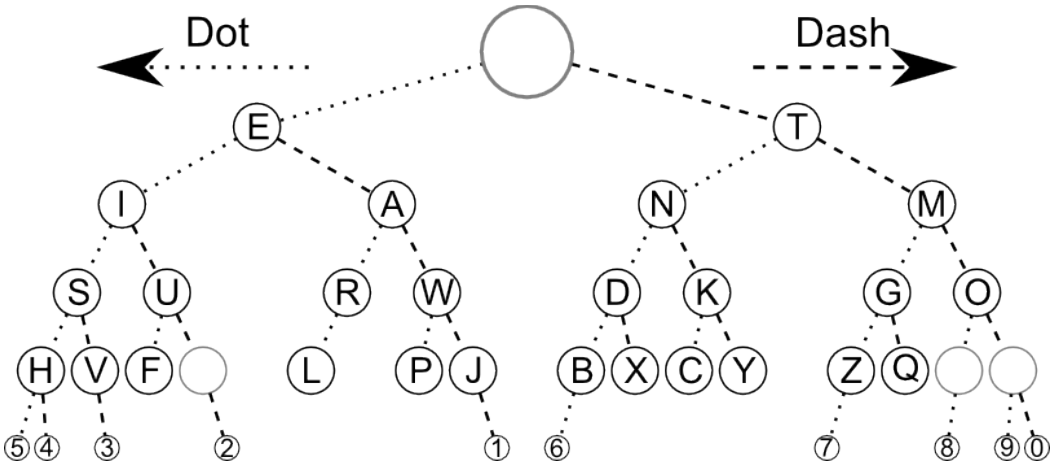
This assignment consists of 17 points:

- 2 points – Binary tree functionality as outlines in Tree.h
- 7 points – Morse code library functionality as outlined in Morse.h
- 5 points – Input/output functionality in lab8.c
- 1 point - Timer implemented correctly
- 1 point - Provided a README.txt
- 1 point - Named partner for the next lab.
- 1 point – Coding and style guidelines were followed with < 6 errors in all source files.

You will lose points for the following:

- NO CREDIT for sections where required code didn't compile
- -2 points: Compilation produces warnings
- -2 points: gotos were used
- -1 point: The used heap size for your project was not specified at the top of lab8.c

Morse tree



Morse code is a representation of the standard English alphabet via a sequence of dots and dashes. Because there can only be dots and dashes for representing a single character, a binary tree can store the patterns for all the characters that can be encoded by Morse code. While there are Morse code representations for many characters in many languages, this lab focuses on only the alphanumeric characters within the ASCII character set. A Morse tree with those characters in them is shown above.

So given this tree, decoding a Morse string involves traversing the tree starting at its root. Either left or right branch is taken depending on what character is next in the string until all the characters have been read. The node that you end up at is the character that the Morse string represented.

For example, to decode "`-..`" you first start at the root node. The first character in the Morse string is a dash, so you branch to the right to the T node. The next two characters are then both dots and so you branch left twice going through the N node and ending on 'D', which is the alphanumeric character represented by the Morse string '`-..`'.

The function `MorseInit()` is where you will initialize your own Morse tree. Make sure that you only initialize just the necessary number of nodes, specifically only those indicated by the above Morse tree diagram.

Binary trees

A binary tree is a common data structure used in programming, with the Morse tree depicted early is shown using a binary tree. The term "binary" refers to the number of children allowed for each node, in this case 2 (ternary trees are also common and they have 3 children per node).

The binary tree library that you will implement stores a single character in each node within the tree and only allows for perfect binary trees. Perfect binary trees are trees in which all levels of the tree are full. This means that the tree has $2^{levels} - 1$ nodes. It should be noted that a Morse tree is not a perfect tree, it isn't even a packed, full, or complete

tree. This means that you will need to account for this discrepancy when using the Tree library within your Morse one.

Recursion

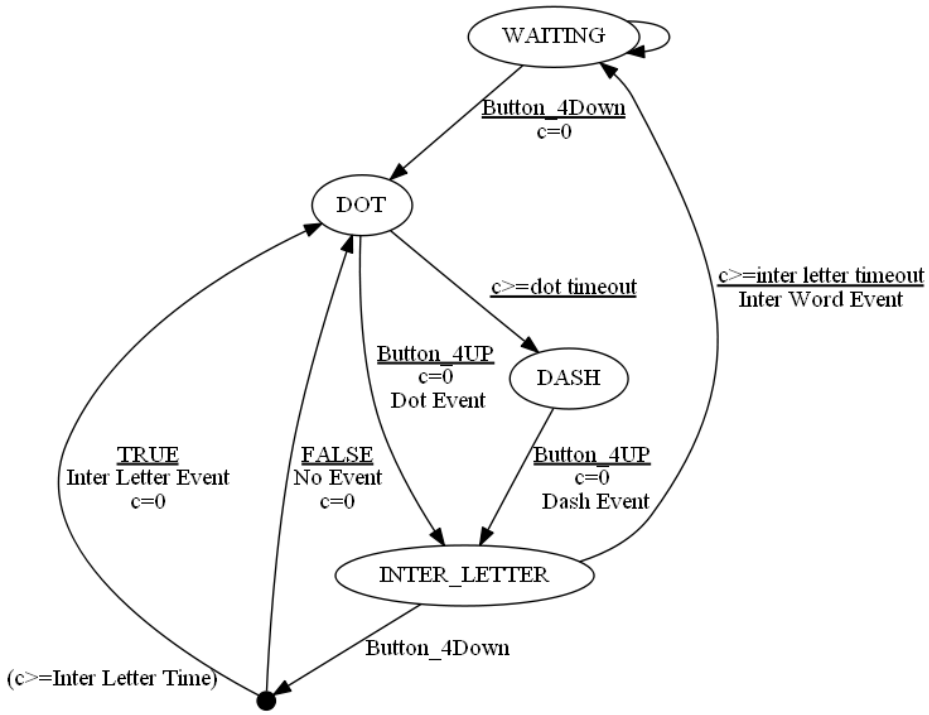
With binary trees, or in fact trees of any type, recursion is a concept that generally comes up. The concept of recursion relies on an action that can be reduced to a base case and an operation that can reduce all other cases to this base case. What this means generally in programming is that you can have a function call itself with slightly different parameters to perform all desired actions necessary.
Wikipedia has a quite thorough description of recursion that would be a good starting point, though I recommend jumping directly to the examples section.

Event-driven programming

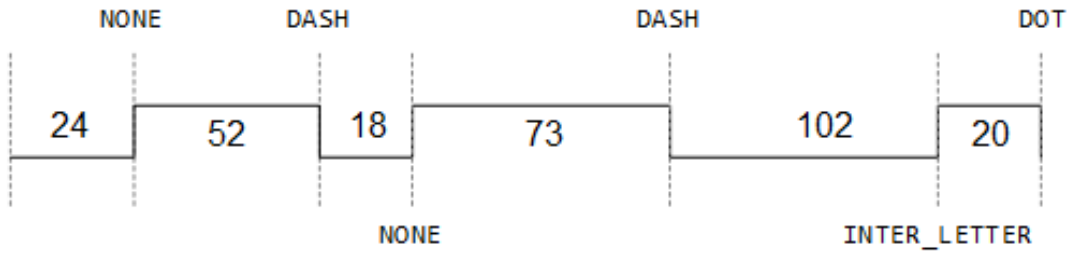
This lab again relies on the event-driven programming paradigm. The events that are important in this lab will be those generated by the Morse library, which in turn relies on the Buttons library. An interrupt for Timer 2, which triggers every 100Hz, has been provided as a way to repeatedly call the MorseEvents event checkers in the background.

Morse Events

In this lab you will again implement a state machine. The state machine diagram below illustrates how to implement the `MorseCheckEvents()` function to convert button events into Morse events.



To understand this state machine better, it is important to understand how the various button events for BTN4 are translated into Morse events based on how long it has been in between the button events. The following timeline shows the button state and how it translates into Morse events. The units are calls to `MorseCheckEvents()`, so the first `NO_EVENT` occurs at the 24th call to `MorseCheckEvents()` (which should be at t=0.24s given that the 100Hz timer calls it).



Now it will be much easier to test that your state machine works correctly here, which you can do easily in the simulator (NOT on the real hardware!), but just calling `MorseCheckEvents()` manually and checking every return value. To do that, you must first set the pins for the buttons to be outputs with the following code, which will come before all of your test code:

```
TRISD &= ~0x00E0;
TRISF &= ~0x0002;
```

Now you can use the BUTTON4_STATE_FLAG constant in lab8.c to set the LATD variable, which will in turn set the pin that corresponds to BTN4 and trick the Buttons library (which is used by `MorseCheckEvents()`) into generating button events. Just remember to account for the 4-timestep delay introduced by `ButtonsCheckEvents()` when writing your test code!

Here is how you would write test code to induce the first two events in the above timeline:

```
TRISD &= ~0x00E0;
TRISF &= ~0x0002;

uint8_t event;
int i;
LATD = 0; // Make sure that BTN4 is unpressed

// Wait for 24 timesteps (only need to wait 20 bc of debouncing)
for (i = 0; i < 20; ++i) {
     event = MorseCheckEvents();
     if (event != MORSE_EVENT_NONE) {
         while (1);
     }
}
```

```
// Now set the button high so that a DOWN_EVENT occurs at t=24
LATD = BUTTON4_STATE_FLAG;
for (i = 0; i < BUTTONS_DEBOUNCE_PERIOD - 1; ++i) {
     event = MorseCheckEvents();
     if (event != MORSE_EVENT_NONE) {
         while (1);
     }
}

// Now this next check will be a NO_EVENT when the button event occurs.
event = MorseCheckEvents();
if (event != MORSE_EVENT_NONE) {
    while (1);
}

// Now change the button state to trigger an UP_EVENT at t=60
for (i = 0; i < 32; ++i) {
    event = MorseCheckEvents();
    if (event != MORSE_EVENT_NONE) {
        while (1);
    }
}
LATD = 0;
for (i = 0; i < BUTTONS_DEBOUNCE_PERIOD - 1; ++i) {
     event = MorseCheckEvents();
     if (event != MORSE_EVENT_NONE) {
         while (1);
     }
}

// Now this next check will be a DASH when the button event occurs.
event = MorseCheckEvents();
if (event != MORSE_EVENT_DASH) {
    while (1);
}
```

Note that the failure of any part will result in an infinite loop being reached. So this code should be run in the debugger and then paused. If the end of the test case is reached without hitting an infinite loop, the test succeeded, otherwise the test failed wherever the loop was hit. Using the Variables window in the debugger, you can see the index of the for-loops used to help in debugging.

Iterative Design Plan

Since you will be implementing from a state machine, the general way to implement this is to implement one state transition at a time. By starting with the display functionality

in this lab, being able to confirm state transitions will be simple because then you can just view the OLED.

We have not provided an iterative design plan because you should be comfortable approaching these labs and getting started. If you are not, please see the professor or TA/tutor for help in planning on how to start this lab. Making an iterative design plan is worthwhile as these labs can be complicated and focusing on a specific feature to implement will make sure you can still move forward, one feature at a time.