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/PpnROF3C77D65Cwa2

22 Points

Warning

This lab is much more involved than the previous ones (it is similar in scope to ToasterOven; conceptually more difficult, but less code), and will take you more time. You will need to start early and read this lab carefully. **Failure to plan is planning to fail**.

Introduction

Morse code is a very old standard for communication where each letter is a combination of dots (short pulse) and dashes (long pulse) that was used extensively in the mid-1800's onwards to connect towns and cities together. Proficient operators could achieve rates of up to 40 words a minute on both transmission and decoding.[1] It is no longer widely used, but has some special applications in Aviation and Maritime domains.

This lab involves Morse code decoding using 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
- Binary tree article on BTSmart (Sections 1-4)

Concepts

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

[1] Note that there is an international standard timing for trained Morse operators that is way too fast for you to be able to coherently test your code. In this lab we have slowed down the timings so that you can ensure you are doing the right decoding. These are controlled via #defines within Morse.h so they could be changed once you are sure your code works correctly.

- 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 binary tree (Morse tree) to decode Morse code strings. 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.
- BinaryTree.h – This file declares an interface for creating and working with binary trees. You will be implementing the corresponding BinaryTree.c file.
- BinaryTreeTest.c – provides a `main()` that is used for unit testing the BinaryTree functions that you have coded. Just like in previous labs, you will need to code your own test harness, and exclude it from the project when done testing.
- 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.

Assignment requirements

- **Binary tree library:** Implement the functions defined in BinaryTree.h in a corresponding BinaryTree.c.
    - All functions must use recursion when traversing the tree.
    - Any constants used must be declared as an enum type or #defined.
    - The only output should be the PrintTree function

- **Binary tree test harness**: Unit testing the implementation of functions defined in BinaryTree.c/h.
    - Create a test harness that tests your BinaryTree functions and ensures that they return the correct values. As in previous labs, at least two tests are required per function.
    - Once you are done testing the functionality of BinaryTree.c, you will need to exclude BinaryTreeTest.c from the project.
    - Be very careful to test the functions on a non-existent tree (i.e: NULL).

- **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 BinaryTree.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 are called 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 calling `FATAL_ERROR()` if it fails.
  - Use Timer2's ISR to check for Morse events at 100Hz. The results of `CheckMorseEvents()` should be returned to a module variable that is checked in the main loop. `Main()` should never call `CheckMorseEvents()`.
  - The Buttons library should not be used directly within lab8.c, as all necessary Buttons functionality is entirely contained 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 DASHs 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 second,[2] 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 an additional second (2 seconds total),[3] 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 4th row should be pushed onto the end of the third row, the first character on the 3rd 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.[4]
    - 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

[2] This is #defined in morse.h as ticks of the 100Hz timer, `MORSE_EVENT_LENGTH_UP_INTER_LETTER`

[3] This is also #defined in morse.h as `MORSE_EVENT_LENGTH_UP_INTER_LETTER_TIMEOUT`

[4] NOTE: collaboration != copying. If you worked with someone else, be DETAILED in your description of what you did together. If you get flagged by the code checker, this is what will save you.

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, BinaryTree.c, BinaryTreeTest.c, README.txt, and lab8.c before the deadline.

Grading

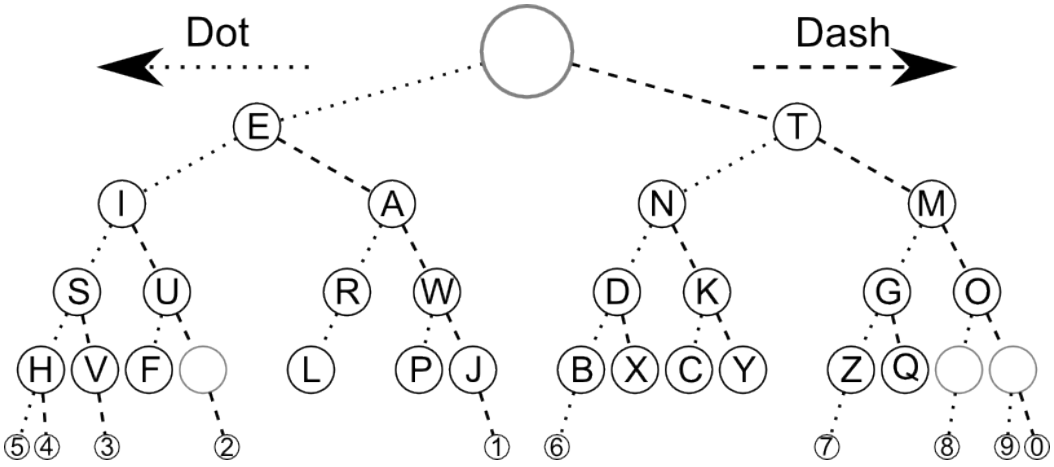
This assignment consists of 22 points:

- 4 points – Binary tree functionality as outlined in BinaryTree.h
- 2 points – Binary tree unit testing functionality in BinaryTreeTest.c
- 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 – GIT hygiene
- 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.[5] 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 Binary Tree library within your Morse library.

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.

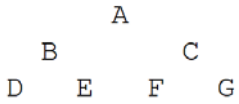
Wikipedia has a quite thorough description of recursion that would be a good starting point, though we recommend jumping directly to the examples section.

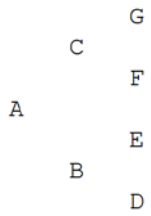
Printing the Binary Tree

BinaryTree.h includes functions to return the left and right child of a given node of the binary tree. A full implementation of a binary tree would include insertions, deletions, and various traversals; however these are beyond the scope of this class.[6]

One function which will be quite useful for this lab is the PrintTree function, which prints out the tree (rotated CCW 90°) so that you can immediately see the tree structure. This function is recursive and has some subtlety involved. The first and most important thing is for PrintTree to be robust to being called with a NULL pointer.[7]

If the tree looks like: Then calling TreePrint results in:





Hence the description that it has rotated the tree CCW 90°.

---

[5] Remember that when using binary, 2^n is (1<<n). Do **NOT** use the math library to raise 2 to power of n.

[6] This is not an Abstract Data Types (ADT) class, and in this class we introduce some of the concepts as they are useful. There are other classes who focus exclusively on ADTs and how to use them.

[7] It is very important to NEVER dereference the NULL pointer. That is, if root is NULL, you cannot call the child (root->left) as you will be attempting to follow the NULL pointer. Deferencing the NULL pointer causes very strange crashes and weird things to happen. Do NOT do it.

The basic idea is to increment the number of spaces to print for each level in, then go right down the tree and do it again. Drop a line, print the spaces and the character (remembering to substitute something for unprintable characters), then go left down the tree and do it again. That is, each call to TreePrint has two calls to TreePrint inside it, one for the right child, and one for the left child. This is the recursion.

The pseudocode for the recursive TreePrint call is:

```
If root is NULL then return
Increment space by LEVEL_SPACES
Recurse right child, space
Print '\n' + ' ' x space + data
Recurse left child, space
```

By following this algorithm, the rightmost node data gets printed on the first line with a number of spaces that is equivalent to LEVEL_SPACES x levels deep that it is. Make sure you understand what is happening by going through the algorithm yourself by hand on a simple tree.

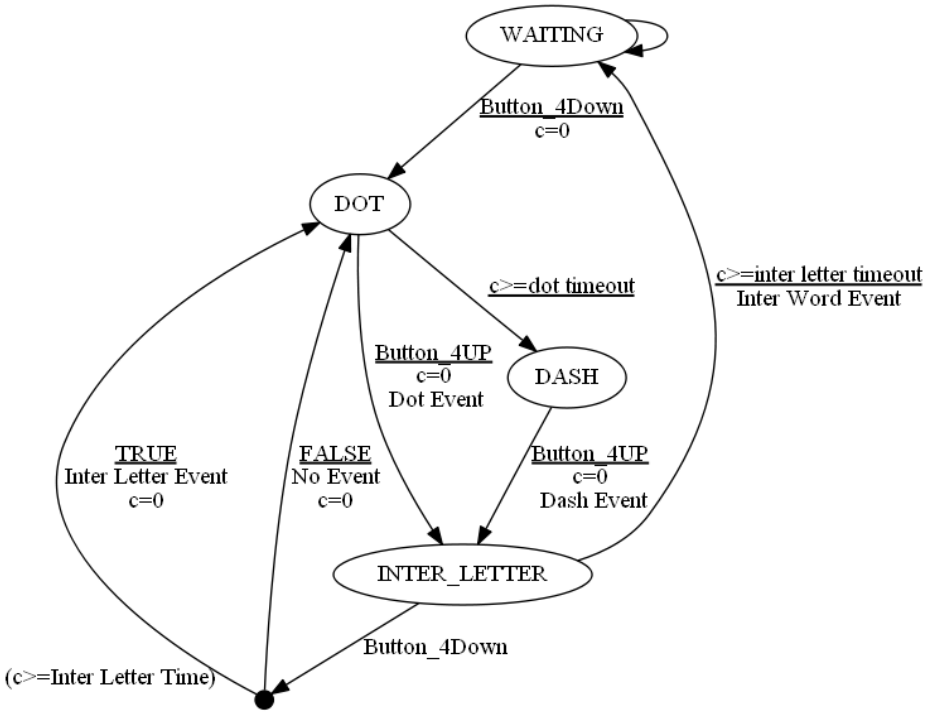
Using PrintTree and some experimentation will show you exactly how to construct your Morse Tree for this lab. Try to set up some examples that will show you how the tree is constructed from your input data array.

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. The returned event must be stored in a module level variable so that the rest of the program can access the events.

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.[8] The way to properly figure out how the state machine works and to understand it is to "walk" your way through the states imagining inputs and checking the output. `MorseCheckEvents()` should be checked inside the 100Hz timer interrupt (which increments a counter) and nominally return `MORSE_EVENT_NONE` as the default.

For example, you start with BTN4 unpressed, and then press BTN4 for 0.5 seconds and then release it and leave it unpressed again. You will get `MORSE_EVENT_NONE` until the 4th call to `MorseCheckEvents()` after releasing the button (for debouncing) and then you will get a single `MORSE_EVENT_DOT` followed by `MORSE_EVENT_NONE`s until two seconds go by and then you will get a single `MORSE_EVENT_INTER_WORD`. This should cause your code to decode "." and print out "`E` ".

See if you can figure out the timing for printing out "CAT ".

Once you have figured out how the state machine works, it is much easier to test that your state machine works correctly, which you can do easily by pressing buttons and having your code print out the number of time steps and what the event was.

[8] Note that the you are really operating on button events and timing, the button events come from a call to `ButtonsCheckEvents()` inside MorseCheckEvents() and the timing is done through incrementing a timer variable each time it is call inside the repeated 100Hz interrupt.

You will have to add some lines to your MorseCheckEvents() that only prints out the state and the number of ticks of the 100Hz counter when the state changes (or else you will spam the output with `MORSE_EVENT_NONE`. Remember to comment out this test section of code before you submit.

Iterative Design Plan

The first step of the Lab is to get the BinaryTree functions working and tested. Use the binary tree functions to experiment (and print the output) to figure out how to encode the Morse Tree.

Since you will be implementing from a state machine, the general way to implement this is to implement one state transition at a time. You can printf inside MorseCheckEvents() to check the states are transitioning the way they should. Ensure that you comment those lines out before you submit your code. 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.