Group 5 Microcontroller Code Documentation

Drive Link

Wave Sampling and Analogue to Digital Conversion

Decoding Morse Code Inputs

Decoding Morse code inputs can be implemented and represented using various data structures, but one of the most efficient would be using a binary tree due to the binary (“.” or “-”) nature of Morse code. This representation optimizes the receiver for the LPC4088 microcontroller through fast lookup, as decoding the input to find its character takes place in O(n) time, where n is the number of dots and dashes in a Morse code character string. The initial implementation relied on dynamic memory allocation, but it was shifted to static memory allocation to avoid relying on functionality from external libraries. As a result, a null pointer is manually defined.

```
#define MAX_NODES 100
#define NULL_PTR ((void*)0)  // NULL replacement

typedef struct Node {
    char letter;
    struct Node* dot;
    struct Node* dash;
} Node;
```

Allowing the tree to have at most 100 nodes gives more than enough space to decode the characters as outlined in the International Morse code document. This memory usage does not strain the microcontroller, but if need be, the number can be adjusted for optimal memory allocation. This implementation is accompanied by a header file, “MorseTree.h” for seamless integration with all code pertaining to the Morse code receiver, providing an accessible way to test the tree with other modules.

Creating a Node

The tree is implemented as an array of Node structs with the given maximum size. To create a node or vertex in the tree, *createNode* checks the index to ensure it remains within the array, then by case analysis on the struct, initializes the new node and returns it.

```
// Static memory pool
static Node nodePool[MAX_NODES];
static int nodeIndex = 0;

// Create a new node using static pool
static Node* createNode() {
```

```
    if (nodeIndex >= MAX_NODES) {
        return NULL_PTR;  // Pool exhausted
    }

    Node* newNode = &nodePool[nodeIndex++];
    newNode->letter = '\0';  // Empty by default
    newNode->dot = NULL_PTR;
    newNode->dash = NULL_PTR;
    return newNode;
}
```

Inserting a Node

To insert a node for a character in the Morse tree, *insertMorse* takes 3 parameters: a pointer for the root node of the tree, a Morse code string consisting of dots and dashes, and the letter to assign that node. It iterates over the string and creates left children for dot inputs and right children for dash inputs. Once the end of the string is reached, the character is stored in the node.

```
// Function to insert a character into the Morse tree
static void insertMorse(Node* root, const char* morse, char letter) {
    Node* current = root;

    while (*morse) {
        if (*morse == '.') {
            if (current->dot == NULL_PTR) current->dot = createNode();  // Create node if
missing
            current = current->dot;
        } else if (*morse == '-') {
            if (current->dash == NULL_PTR) current->dash = createNode();
            current = current->dash;
        }
        morse++;
    }

    current->letter = letter;  // Store the character at the leaf node
}
```

Building the Morse Tree

Building the tree is a simple matter of using the *insertMorse* function in accordance with international Morse code standards, assigning characters to nodes in the way they would appear in a breadth first search, which is why they appear in order of how many symbols make up a given character. The rules are fully outlined in comments, and a simplified version is shown here for brevity.

```
Node* buildMorseTree() {
    nodeIndex = 0;  // Reset pool
    Node* root = createNode();  // Root is empty

    /* ... */

    // Letters

    // 1 symbol characters, 2 characters
    insertMorse(root, ".", 'e');
    insertMorse(root, "-", 't');

    // 2 symbol characters, 4 characters
    insertMorse(root, ". .", 'i');
    insertMorse(root, ". -", 'a');
    insertMorse(root, "- .", 'n');
    insertMorse(root, "- -", 'm');

    ...rest of the tree...

    // 8 symbol characters, 1 character
    insertMorse(root, ". . . . . . . .", '^');  // error (8 dots)

    return root;
}
```

Decoding Dots and Dashes

This is the essence of the tree implementation, where performance can really be observed in testing. The decoding function takes the root, Morse code string, and performs a simple guided walk of the tree, checking at each step if the character is in the tree. At the end of the walk, *decodeMorse* checks via the ternary operator that the final node is not empty or null, returning the letter (character more accurately) stored in that node and a question mark otherwise.

```
// Function to decode a Morse code sequence
char decodeMorse(Node* root, const char* morse) {
    Node* current = root;

    while (*morse) {

        // Invalid Morse sequence check
        if (*morse == '.') {
            if (current->dot == NULL_PTR) return '?';
            current = current->dot;
        } else if (*morse == '-') {
            if (current->dash == NULL_PTR) return '?';
            current = current->dash;
```

```
        }

        morse++;
    }

    return (current && current->letter != '\0') ? current->letter : '?';  // Return character if valid
}
```

Testing

A straightforward test of the tree was performed as shown with the following code and outputs of that code. Note that print statements use stdio.h, an external library, but purely for testing purposes as outlined, and functionality is not dependent on external libraries. This test can be done for any and all characters stored in *buildMorseTree*.

```
#include <stdio.h>
#include "morsetree.h"

int main() {

    Node* morseTree = buildMorseTree();

    printf("Morse '. - - - -' -> %c\n", decodeMorse(morseTree, ". - - - -"));       // 1
    printf("Morse '. . - - -' -> %c\n", decodeMorse(morseTree, ". . - - -"));       // 2
    printf("Morse '. . . - -' -> %c\n", decodeMorse(morseTree, ". . . - -"));       // 3

    return 0;
}
```

Output:

Morse '. - - - -' -> 1

Morse '. . - - -' -> 2

Morse '. . . - -' -> 3

A more robust unit test is presented below, where assert.h is clearly only included for testing.

```
#include <stdio.h>
#include "morsetree.h"
#include <assert.h>

void unitTestDecodeMorse() {
    // Test on small Morse tree
    Node* root = createNode();
```

```
    insertMorse(root, ".", 'E');
    insertMorse(root, "-", 'T');

    char result;

    // Valid test case: Decoding a single Morse code symbol
    result = decodeMorse(root, ".");
    assert(result == 'E');  // Expecting 'E' for Morse code "."

    result = decodeMorse(root, "-");
    assert(result == 'T');  // Expecting 'T' for Morse code "-"

    // Invalid test case: Decoding an invalid Morse code
    result = decodeMorse(root, "..");  // This should reach a dead end in the tree, would be "i"
in a full tree
    assert(result == '?');  // Expecting '?' because ".." is not mapped

    result = decodeMorse(root, "-.-.");  // Invalid Morse sequence, would be "c" in a full tree
    assert(result == '?');  // Expecting '?' due to no mapping

    printf("All test cases passed!\n");
}

int main() {
    unitTestDecodeMorse();
    return 0;
}
```

Output:
All test cases passed!

Displaying Outputs on LCD

Scheduling