Nathan Roe
EN.606.715.81.SP23
2023-02-05

Project 1 - Morse Code LED System

The first project for this class involves using an Arduino to blink Morse Code statements received from a user. A full demonstration video can be found on Youtube HERE and the source code for my project is included in the document, or available on Github HERE.

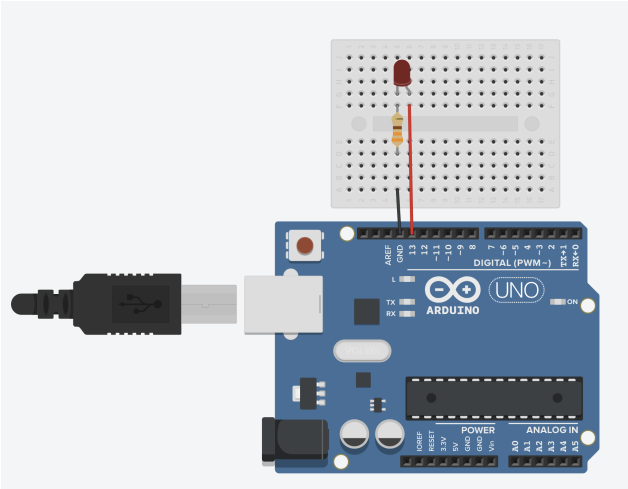
Requirements

- Application Displays user typed string as Morse Code on an LED (or several LEDs), or an LCD
    - Derived Requirement: Accepts user input
        - Met using Serial read to the Arduino, in my demonstration through the build-in Serial Monitor to send messages across the Serial interface
    - Derived Requirement: Morse Code on one or more LEDs
        - Met using the Arduino internal LED as well as an external LED that blink sequences of Morse
- Implement all of the Morse Code letters.
    - Derived Requirement: Characters Aa-Zz and 0-9 are valid characters as part of the input string
        - Met in the MorseCode class which contains all required blink sequences for these characters
    - Derived Requirement: Blink durations follow the Morse Code rules provided to the class: Dots are 1 unit, Dashes are 3 units, blink between dots/dashes is 1 unit, gap between letters is three units, and spaces are 7 units
        - Met in the MorseCode class, which uses a unit as 250 ms, and sets durations for all parts as required
- Design using a Round Robin Design where in a loop it waits for a string, displays it in Morse Code, and only exits the loop if a sentinel is entered, such as ctrl-z
    - Derived Requirement: Round Robin Design
        - Met in the main.ino Sketch, which has two functions that run in a loop: receiving input and blinking results
    - Derived Requirement: waiting for string until new-line character
        - Met in main.ino Sketch, which stores input characters until newline received, which exits input loop and begins blink loop
    - Derived Requirement: Sentinel value exits loop
        - Met in main.ino Sketch, as the main *loop* function does not exit until Ctrl+z or Ctrl+c are received

Nathan Roe
EN.606.715.81.SP23
2023-02-05

Hardware Design

The hardware design for my project is very simple, with an Arduino connected to my host Laptop over USB to receive Serial inputs. The digital pin 13 is then connected to the high side of an LED. I used pin 13 as it also has the Arduino built-in LED that I could use for testing without needing the additional electronic components. The low side of the LED then connects to a 330 Ω resistor, which, in turn, connects to the Arduino ground pin.



Software Design

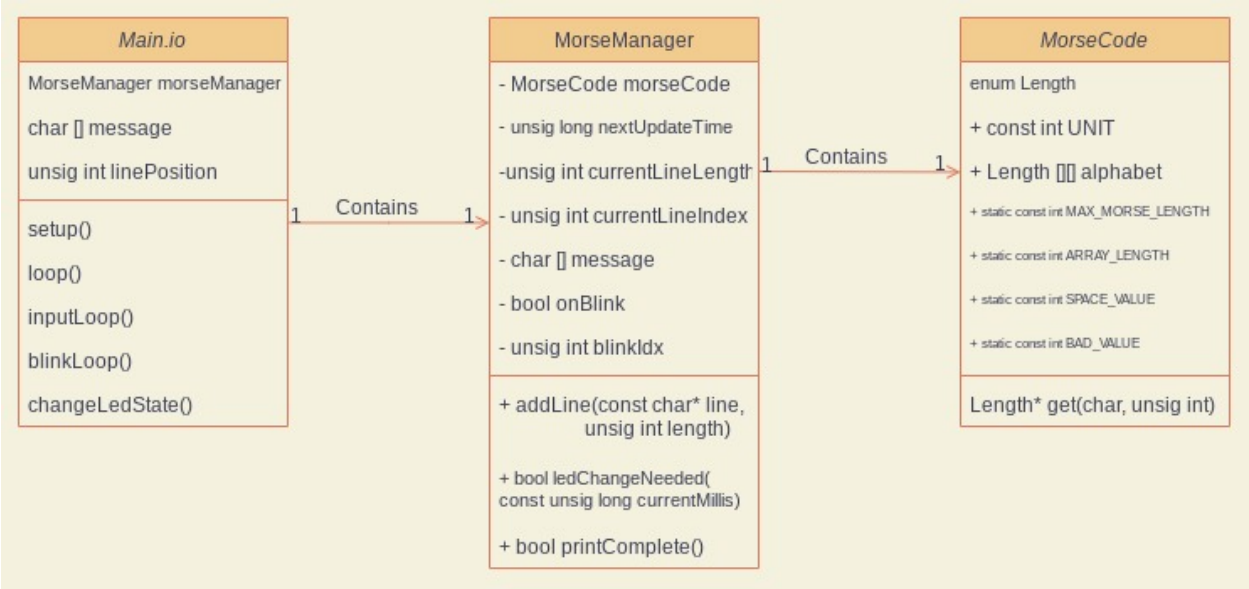
For my software implementation, I have three main components: the Sketch (main.io) file that runs on an Arduino, a MorseManager class that handles the timing, and a MorseCode object that handles constants related to Morse Code.

The Sketch will contain the two standard Arduino functions, *setup* and *loop*, where *setup* runs once and initializes variables, and *loop* runs until exit is called or the Arduino power resets. For this project exit is called if the Serial reader gets a Ctrl+z or Ctrl+c value. The *loop* function will hop between two other loops in round robin fashion, *inputLoop* and *blinkLoop*. The *inputLoop* runs until a line end or exit character is received, and then sends the last line of text to MorseManager, then will exit the loop. Once the *inputLoop* exits, the *blinkLoop* begins, and repeatedly checks with MorseManager to determine whether the LED status should be changed and whether the morse code blink sequence has completed. Once *blinkLoop* completes, *loop* resets and enters *inputLoop* once again.

Nathan Roe
EN.606.715.81.SP23
2023-02-05

The MorseManager class will control timing for the morse code, by receiving data about blink durations and sequences from a MorseCode object. The code will operate based on a function *ledChangeNeeded* that takes a current time in milliseconds (long) and determines whether the next update time has been passed. This assumes chronological calls to the function, but keeps the class from having to maintain information about hardware states, which will all be managed by the Sketch.

The MorseCode object will be a mostly static object that just sets information about Morse. It will contain an array with all allowable character blink sequences for the characters 0-9 and Aa-Zz, with capital and lowercase letters considered the same. It also sets how long Short blinks, Long blinks, and spaces between characters and words will last.



```ino
#include "MorseManager.hpp"

// Initialize Required Variables
const int ledPin = LED_BUILTIN;
int ledState = LOW;

const unsigned int MAX_LINE_LENGTH = 20;
unsigned int linePosition = 0;
static char message[MAX_LINE_LENGTH];
const char *m = message;

const int CTRL_C = 3;
const int CTRL_Z = 26;

MorseManager *morseManager;

// Arduino setup, first function called
void setup()
{
    pinMode(ledPin, OUTPUT);
    Serial.begin(9600);
    morseManager = new MorseManager();
}

// Arduino loop, runs continuously until exit()
void loop()
{
    inputLoop();
    blinkLoop();
}

// First step in loop; receives user input from Serial
void inputLoop()
{
    Serial.println("Please Provide Input");
    while (true)
    {
        char readByte = Serial.read();
        if (readByte == '\n')
        {
            morseManager->addLine(m, linePosition);
            linePosition = 0;
            Serial.println(" Playing");
            break;
        }
        else if (readByte == CTRL_Z || readByte == CTRL_C)
        {
            exit(0);
        }
        else if (linePosition < MAX_LINE_LENGTH && readByte >= 0)
        {
            message[linePosition++] = readByte;
            Serial.print(message[linePosition - 1]);
        }
    }
```

```
}

// Second step in loop; blinks Morse code translation of input
void blinkLoop()
{
    while (true)
    {
        if (morseManager->ledChangeNeeded(millis()))
        {
            changeLedState();
        }

        if (morseManager->printComplete())
        {
            break;
        }
    }
}

// Flips current LED state
void changeLedState()
{
    if (ledState == LOW)
    {
        ledState = HIGH;
    }
    else
    {
        ledState = LOW;
    }
    digitalWrite(ledPin, ledState);
}
```

```cpp
#pragma once

#include "MorseCode.hpp"

using namespace std;

// MorseManager for tracking Morse Code sequence durations
//
// Starts blink time tracking when line is added using
// addLine; status updates can be tracked using ledChangeNeeded
// and using printComplete to verify all characters blinked.
class MorseManager
{

public:
    MorseManager(){};

    // Copies char array to blink in morse, and reset all tracking vars
    void addLine(const char *line, unsigned int length);

    // Checks whether next LED update change time is exceeded
    // based on current Arduino time in milliseconds
    bool ledChangeNeeded(const unsigned long currentMillis);

    // Checks wether current line blink sequence is complete
    bool printComplete();

    // Getters for internal variables, used for debugging Morse sequence timing
    unsigned long getNextUpdateTime() { return this->nextUpdateTime; };
    unsigned int getLineLength() { return sizeof(this->message) / sizeof(char); };
    unsigned int getCurrentLineLength() { return this->currentLineLength; };
    unsigned int getCurrentLineIndex() { return this->currentLineIndex; };
    bool getOnBlink() { return this->onBlink; };
    unsigned int getBlinkIdx() { return this->blinkIdx; };

private:
    MorseCode *morseCode = new MorseCode();
    unsigned long nextUpdateTime = 0;

    static const unsigned int MAX_LINE_LENGTH = 20;
    unsigned int currentLineLength = 0;
    unsigned int currentLineIndex = 0;
    char message[MAX_LINE_LENGTH];

    bool onBlink = true;

    unsigned int blinkIdx = 0;
};
```

```cpp
#include "MorseManager.hpp"
#include "MorseCode.hpp"

// Checks whether next LED state change time is met
// based on current Arduino time in milliseconds
bool MorseManager::ledChangeNeeded(const unsigned long currentMillis)
{
    // No LED change needed if next update time is not exceeded
    bool changeNeeded = false;
    // Next update time has been met, check whether conditions require LED update
    if (currentMillis > this->nextUpdateTime)
    {
        // Enter state between Dashes/Dots
        if (!onBlink)
        {
            this->onBlink = true;
            this->nextUpdateTime = currentMillis + MorseCode::length::BLINK;
            changeNeeded = true;
        }
        // Enter state for current character's next Dash or Dot
        else if (onBlink && this->blinkIdx < MorseCode::MAX_MORSE_LENGTH - 1)
        {
            this->onBlink = false;
            if (this->nextUpdateTime > 0)
            {
                ++this->blinkIdx;
            }
            this->nextUpdateTime = currentMillis + *morseCode-
>get(message[currentLineIndex], blinkIdx);
            if (currentMillis == this->nextUpdateTime)
            {
                this->onBlink = true;
                changeNeeded = false;
            }
            else
            {
                changeNeeded = true;
            }
        }
        // Enter state for Gap between characters
        else if (this->blinkIdx == MorseCode::MAX_MORSE_LENGTH - 1)
        {
            this->onBlink = true;
            ++this->currentLineIndex;
            ++this->blinkIdx;
            this->nextUpdateTime = currentMillis + MorseCode::length::GAP;
            changeNeeded = false;
        }
        // Move to next character and begin first Dash/Dot
        else if (this->currentLineIndex < currentLineLength)
        {
            this->onBlink = false;
            this->blinkIdx = 0;
            this->nextUpdateTime = currentMillis + *morseCode-
>get(message[currentLineIndex], blinkIdx);
            if (currentMillis == this->nextUpdateTime)
```

```
            {
                this->onBlink = true;
                changeNeeded = false;
            }
            else
            {
                changeNeeded = true;
            }
        }
    }
    // Leave LED alone if Space is active
    if (message[currentLineIndex] == ' ')
    {
        changeNeeded = false;
    }
    return changeNeeded;
}

// Copies char array to blink in morse, and reset all tracking vars
void MorseManager::addLine(const char *line, unsigned int length)
{
    this->nextUpdateTime = 0;
    this->onBlink = true;
    this->currentLineIndex = 0;
    this->blinkIdx = 0;

    const char *linePointer = line;
    unsigned int idx = 0;
    while (idx < length)
    {
        this->message[idx] = *line;
        ++line;
        ++idx;
    }
    this->currentLineLength = length;
}

// Checks wether current line blink sequence is complete
bool MorseManager::printComplete()
{
    bool letterComplete = this->blinkIdx >= MorseCode::MAX_MORSE_LENGTH;
    bool lineComplete = this->currentLineIndex >= this->currentLineLength;
    return letterComplete && lineComplete;
}
```

```cpp
#pragma once

// Define timing and blink sequences for Morse Code
//
// Defines timing base off of variable UNIT, and provides
// blink sequences in sets of 5 (longest Morse Code pattern).
// the function 'get' provides the blink duration for
// a character and a sequence position (0-4), where
// an N/A (0) means the Morse sequence is complete
class MorseCode
{
public:
    // Defines single Morse Unit in ms
    static const int UNIT = 250;
    // Define Morse Code durations based on Unit
    enum length
    {
        NA = 0,
        BLINK = UNIT,
        SHORT = UNIT,
        GAP = 3 * UNIT,
        LONG = 3 * UNIT,
        SPACE = 7 * UNIT
    };

    // Define aspects of the alphabet
    static const int MAX_MORSE_LENGTH = 5;
    static const int ARRAY_LENGTH = 38;
    static const int SPACE_VALUE = 36;
    static const int BAD_VALUE = 37;

    // Array defining Morse Code Sequences for each Character
    MorseCode::length alphabet[ARRAY_LENGTH][MAX_MORSE_LENGTH] = {
        {LONG, LONG, LONG, LONG, LONG},      // 0
        {SHORT, LONG, LONG, LONG, LONG},     // 1
        {SHORT, SHORT, LONG, LONG, LONG},    // 2
        {SHORT, SHORT, SHORT, LONG, LONG},   // 3
        {SHORT, SHORT, SHORT, SHORT, LONG},  // 4
        {SHORT, SHORT, SHORT, SHORT, SHORT}, // 5
        {LONG, SHORT, SHORT, SHORT, SHORT},  // 6
        {LONG, LONG, SHORT, SHORT, SHORT},   // 7
        {LONG, LONG, LONG, SHORT, SHORT},    // 8
        {LONG, LONG, LONG, LONG, SHORT},     // 9
        {SHORT, LONG, NA, NA, NA},           // A
        {LONG, SHORT, SHORT, SHORT, NA},     // B
        {LONG, SHORT, LONG, SHORT, NA},      // C
        {LONG, SHORT, SHORT, NA, NA},        // D
        {SHORT, NA, NA, NA, NA},             // E
        {SHORT, SHORT, LONG, SHORT, NA},     // F
        {LONG, LONG, SHORT, NA, NA},         // G
        {SHORT, SHORT, SHORT, SHORT, NA},    // H
        {SHORT, SHORT, NA, NA, NA},          // I
        {SHORT, LONG, LONG, LONG, NA},       // J
        {LONG, SHORT, LONG, NA, NA},         // K
        {SHORT, LONG, SHORT, SHORT, NA},     // L
        {LONG, LONG, NA, NA, NA},            // M
```

```
        {LONG, SHORT, NA, NA, NA},          // N
        {LONG, LONG, LONG, NA, NA},         // O
        {SHORT, LONG, LONG, SHORT, NA},     // P
        {LONG, LONG, SHORT, LONG, NA},      // Q
        {SHORT, LONG, SHORT, NA, NA},       // R
        {SHORT, SHORT, SHORT, NA, NA},      // S
        {LONG, NA, NA, NA, NA},             // T
        {SHORT, SHORT, LONG, NA, NA},       // U
        {SHORT, SHORT, SHORT, LONG, NA},    // V
        {SHORT, LONG, LONG, NA, NA},        // W
        {LONG, SHORT, SHORT, LONG, NA},     // X
        {LONG, SHORT, LONG, LONG, NA},      // Y
        {LONG, LONG, SHORT, SHORT, NA},     // Z
        {SPACE, NA, NA, NA, NA},            // SPACE
        {NA, NA, NA, NA, NA}                // BAD VALUE
    };

    // Function for getting Morse Code blink duration for
    // a Character and sequence position
    MorseCode::length *get(char letter, unsigned int idx)
    {
        if (idx >= MAX_MORSE_LENGTH)
        {
            return &alphabet[BAD_VALUE][idx];
        }

        int charVal = int(letter);
        if (letter >= '0' && charVal <= '9')
        {
            charVal = charVal - int('0');
        }
        else if (letter >= 'A' && letter <= 'Z')
        {
            charVal = charVal - int('A') + 10;
        }
        else if (letter >= 'a' && letter <= 'z')
        {
            charVal = charVal - int('a') + 10;
        }
        else if (letter == ' ')
        {
            charVal = SPACE_VALUE;
        }
        else
        {
            charVal = BAD_VALUE;
        }

        return &alphabet[charVal][idx];
    };
};
```