Assignment 3 – Song Composer
CSCE 4114 – Embedded Systems – Fall 2018
Due Date – Friday, November 16th, 11:59 PM CT

Description: This assignment is designed to give you experience with microcontroller hardware, serial communication, and pulse width modulation. The assignment is to create a program that uses terminal-like operation over a serial communication to enter songs and play them back. [*]

Expected Operation: The following will define the behavior of the program you will develop:

1. On boot-up, a menu will be displayed that offer three options

   ```
   ====Main Menu====
   1: List Songs
   2: Play Song
   3: Create Song
   ```

   The user will input 1, 2, or 3 in the terminal with an enter return, moving to the chosen menu

2. Menu Option '1': List Songs
   If Option '1' is selected, a list of up to four stored songs will be shown. The list will appear as:

   ```
   ====Song List====
   1: Title: Title1
   2: Title: Title2
   3: Title: Title3
   4: Title: Title4
   ```

3. Menu Option '2': Play Song If option '2' is selected, the user is given a choice between playing a song by number, or searching for a song by title. e.g.

   ```
   ====Play Song Menu====
   1: Play By Number
   2: Search By Title
   =====================
   Please Enter Choice:
   ```

   If option 1 selected, user should be prompted for a song number corresponding to the number found in the list menu (Menu Option 1).
   If option 2 is selected, the user should be prompted for the title; e.g.

[*]This assignment is adapted from an assignment by Dr. Ryan Robucci of the University of Maryland, Baltimore County.

```
Enter title search string: mary had a
```

4. Search Behavior:
   For option 2, the song selection should be done as follows:

   - Treat user input string as a series of white-space separated tokens called the query string
   - Treat the stored titles as a series of whitespace separated tokens called the template strings
   - Determine if a matching token is found for each token in the query string
   - Count matches for repeated tokens in the query string, **but not repeated tokens in the template string**
   - Create a function to find the number of query tokens that match a token in the template string
   - Perform all matching as case-insensitive
   - Automatically select the song with the **BEST** match and continue by printing the song information and playing the song

   Example function MatchScore(char* query, char* template);

   ```
   MatchScore("hello there", "Hello") -- return 1
   MatchScore("hello there", "Hello hello") -- return 1
   MatchScore("Hello hello there", "Hello") -- return 2
   MatchScore("hello hello there", "hello hello") -- return 2
   MatchScore("hello hello there", "hello hello thERE") -- return 3
   ```

5. Menu Option '3': Create Song If option '3' selected from the menu, the user will be prompted to create a song. The user will be prompted with a number of a song to overwrite (1,2,3,4), a song title, and an ASCII representation of the song. e.g.

   ```
   Which song would you like to overwrite (1-4): 2
   Enter the Title of the Song: Hot Cross Buns
   Enter the song (A-G/R(est) followed by quarter seconds) : B2A2G3R1B2A2G3R1
   ```

   The user-input encoding for the song is as follows:

   ```
   <letter><quarter secs><letter><quarter secs>...<letter><quarter secs>
   ```

   - The letter may be uppercase or lowercase ABCDEFG or R (rest)
   - A-G are a tone to be played (e.g. A = 440 Hz)

- R is a rest (silence)
- Quarters should be a number from 0-31, and are the number of quarter seconds to play the note/rest
- The user presses enter to finish entering a song
- Once the user enters the song, it must be stored more compactly in order to save memory. Each letter-number pair should be stored as a single byte, using the following encoding:
  - A – 0b000
  - B – 0b001
  - C – 0b010
  - D – 0b011
  - E – 0b100
  - F – 0b101
  - G – 0b110
  - R – 0b111
- Letter is stored into the upper 3 bits of a byte
- Quarter seconds value stored in lower 5 bits of a byte
- End of song indicated by a zero-length rest – 0b11100000
- User does not need to provide R0 to indicate end of the song. Must append that to end of array to know when to stop playing

Example: User provides string: B2A2R1C2
Song should be stored as:
0b00100010
0b00000010
0b11100001
0b01000010
0b11100000 – Terminating note automatically appended
If user provides R0 in string, I don't care if you store any remaining notes or not.

Functions You MUST Write: The following functions must be written in your code.

- void StripEOL(char string[], int n);
  StripEOL looks through up to n characters to find '\n' or '\r'. Once it is found, the search should stop, and the character should be replaced with a null terminating character ('\0'). The user input string can be passed through this function and then further processed.

- uint8_t DisplayMenu(const char menu[]);
  DisplayMenu displays a menu. The parameter (menu) should be a string of '\n'-delimited substrings. The first one is the title, and the remaining ones correspond to a list of options. The function must check user input. If an invalid selection is made,

the user should be reprompted.

Example: userChoice = DisplayMenu("Main Menu\nCreateSong\nPlay Song\nListSongs\n);
In this case, the userChoice should be assigned a value from 1-3.

- List Songs Function
  void ListSongs(char songTitle[NUM_SONGS][MAX_TITLE_LENGTH]);
  ListSongs prints an organized list of songs.

- Play Song Function
  void PlaySong(uint8_t song[]);
  PlaySong plays a stored song with ending indicated by R0. Each byte should be defined using the above structure. The PlaySong function uses the Piezo Buzzer element to play a tones.

- Pack/Unpack Note Functions
  uint8_t PackNote(char letterASCII, uint8_t duration);
  PackNote takes input provided from a songString and packs into bytes as described above. Note that the letter is NOT stored in ASCII.
  uint8_t UnpackNoteLetterASCII(uint8_t packedNote);
  UnpackNoteLetterASCII returns ASCII character of a packed note.
  uint8_t UnpackNoteDuration(uint8_t packedNote);
  UnpackNoteDuration returns the number of quarter seconds a note will be played.

- Store Song Function
  void StoreSong(uint8_t song[], const char songString[]);
  StoreSong takes ASCII input provided in songString and packs information into song bytes. SongString should be note-duration pairs such as "B2A2G3R1B10R0A20G30". Lower-case notes should also be accepted. R corresponds to silence. Any zero-length-silences R0 indicate the end of the song. Notes following an R0 should not be stored. Storage format must be compatible with PlaySong function and adhere to other descriptions in this document.

- Play Note Functions
  void PlayNote(uint8_t letterASCII, uint8_t quarters);
  You will need functions to play notes for variable duration. Here are example constants for a G note:

  ```
  #define FREQ_G4_HZ 392
  #define HALFPERIOD_G4_US 1276 // delay in microseconds
  ```

  See `https://pages.mtu.edu/~suits/notefreqs.html` for note frequencies.
  Pseudo-code for the playNote function:

  1. Declare integers numIterations and halfPeriodMicroseconds

2. Based on the ASCII letter, assign halfPeriodMicroseconds and numIterations. The number of iterations can be computed from the period and delay as (freq/4)*quarters.
3. Repeat this for numIterations times:

   ```
   {
       set piezoPin low
       delay for halfPeriodMicroseconds
       set piezoPin high
       delay for halfPeriodMicroseconds
   }
   ```

- Matching Function
  int MatchScore(const char countQueryString[], const char templates[]);
  MatchScore treats contents of countQueryString and templates as whitespace-delimited tokens. It returns a count of the number of tokens from countQueryString that can be found at least once in a template string. All queries should be case-insensitive. Queries are defined above.

Useful Stuff or Top of Code

```
#define USER_LINE_MAX 128
char userLine[USER_LINE_MAX];

#define NUMBER_OF_SONGS 4
#define MAX_SONG_LENGTH 64
#define NOTE_A 0
#define NOTE_B 1
#define NOTE_C 2
#define NOTE_D 3
#define NOTE_E 4
#define NOTE_F 5
#define NOTE_G 6
#define NOTE_R 7

char songTitle [NUMBER_OF_SONGS][STR_LENGTH]={"Title1","Title2","Title3","Title4"};
char song[NUMBER_OF_SONGS][MAX_SONG_LENGTH] =
{{(NOTE_B<<5)+2 ,(NOTE_A<<5) +2,(NOTE_G<<5) +2 },{NOTE_R<<5},{NOTE_R<<5},{NOTE_R<<5}};
// some initial song with notes
const char menuMain[] = "Main Menu\nCreate Song\nPlay Song\nList Songs\n";
const char menuPlay[] = "Play Menu\nSearch By Title\nNumber\n";
```

Rubric: The project will be graded according to the following rubric:

These documents should be uploaded through Blackboard to the TA before the beginning of the next lab. Documents turned in after this deadline are subject to the course late policy.

Category	Description	Percentage
Implement Behavior	The behavior of the system should be implemented as defined by the expected behavior. Each of the 5 behaviors needed to operate correctly.	50%
State Machine Diagram	Make a diagram of the state machine(s) that compose the system. The state machine should clearly note the states, transitions, and actions of each state. Shared variables between state machines should be shown, with what SM controls the variable, and what SMs read from the variable.	20%
Coding Comments & Style	Use appropriate coding styles. Comment functions with a description about their behavior, any parameters, any return values, and any shared variables which it manipulates.	15%
Report	A simple, one- or two-page report. The report should have the project name (e.g. Binary Mastermind), a short description of what you did, and the outcomes (e.g. Did it pass all example tests, if not, why not, etc...).	15%

Table 1: Grading Rubric