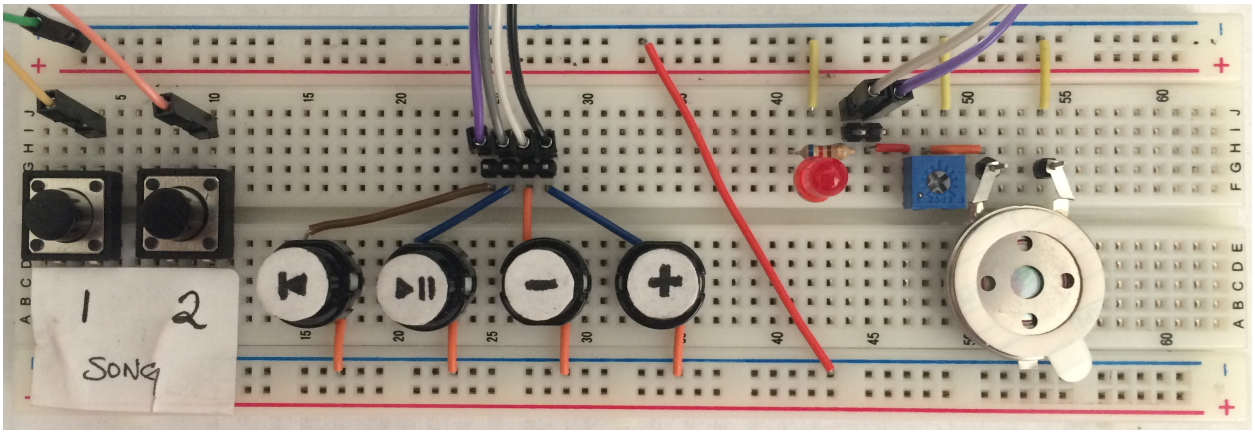
Carlton Duffett
EC450 HW5
Professor Giles
3/25/2015

EC450 Homework 5

Hardware Design:

I chose to implement the music player with the following components:



1. Restart Button
   Allows the user to restart the currently selected song. Also resets the tempo.

2. Play/Pause Button
   Pressing the button toggles between playback and pause modes.

3. Increase/Decrease Tempo Buttons
   '-' lowers the rate at which notes are played. '+' raises the rate at which notes are played. The lower bound yields a playback rate of several seconds per measure. The upper bound yields a playback rate of several milliseconds per measure.

4. Select Song Buttons
   Song 1 is "Joy to the World." Song 2 is the Super Mario Theme. Pressing each button selects the respective song and queues it for play. Song 1 is queued at startup by default.

5. LED Indicator
   Indicates the state of the system. A solid light indicates the start or the end of a song (before playback begins or after it finishes). A flashing light indicates that the song is paused during playback. The light is off during normal playback.

6. Potentiometer/Speaker
   Allows fine adjustment of the speaker's volume.

Collectively, these 8 devices use the entirety of port 1. The port 1 interrupt vector handles all 6 buttons.

Software Design:

Each song is stored in a character array. Each note is stored in a separate 8-bit character. The most significant 3 bits store the note's duration. The least significant 5 bits store the note's frequency. For example:

Note:	0	1	0	0	1	0	1	1
	Duration			Frequency				

Given 3 bits, there are 8 durations to select from, stored in the array `durations[]`:

0. Break (quick pause between notes of the same frequency)
1. $1/16^{th}$ note
2. $1/8^{th}$ note
3. Dotted $1/8^{th}$ note
4. $1/4^{th}$ note
5. Dotted $1/4^{th}$ note
6. 1/2 note
7. Dotted 1/2 note

The "break" is inserted into the music score between adjacent notes of the same frequency. It is equivalent to a $1/64^{th}$ note of silence. A whole note must be represented by two 1/2 notes of the same frequency without a break in between them.

Given 5 bits, there are 32 notes to choose from of which is use 26, stored in the array `notes[]`. Frequency 0 is a "rest" frequency that is inaudible to the user. This avoids stopping the timer during normal playback. The remaining 25 notes represent 2 full octaves from C4 to C6.

The "frequencies" stored in the notes array are actually half-period values. Given a 1MHz clock, and a desired frequency *f*, I calculated each half period *p* using the relationship:

$$\frac{1Mhz}{2p} = f$$

To make writing each score easier, I defined constants for all of indices into the `durations[]` and `notes[]` arrays. For example, I defined the durations as:

```
#define BREAK       0x00    // break between adjacent notes
#define SIXTEENTH   0x20
#define EIGHTH      0x40
#define D_EIGHTH    0x60    // dotted eighth
#define QUARTER     0x80
#define D_QUARTER   0xA0    // dotted quarter
#define HALF        0xC0
#define D_HALF      0xE0    // dotted half
```

And some of the notes as:

```
#define C4      0x01
#define C4s     0x02    // sharp
#define D4      0x03
#define E4b     0x04    // flat
#define E4      0x05
#define F4      0x06
#define F4s     0x07
#define G4      0x08
```

This allowed me to write each score using the following compact syntax. For example, the first two measures of "Joy to the World":

```
(QUARTER + C5),     // 1st measure
(D_EIGTH + B4),
(SIXTEENTH + A4),
(D_QUARTER + G4),   // 2nd measure
(EIGTH + F4),
```

Declared as `const unsigned char[]` arrays, the two songs I wrote occupy exactly 500 characters, or 500 bytes of memory. This leaves ample room for additional or longer songs.

Playback:

When a new song is queued, two counters `duration_counter` and `score_counter` are reset to 0. `score_counter` keeps track of which note in the song is currently being played. As the song progresses, `duration_counter` is incremented to count the number of WDT interrupts per note. After a note's full duration is reached, the `score_counter` is incremented and the `duration_counter` is reset to 0.

For example, a default 1/8th note occupies 36 WDT interrupts. This 36 interrupts is a combination of a set value for an 1/8th note (16) and the variable `tempo` that scales this value by some multiple. The default tempo value is 2.25.

1/8th note: `16 * 2.25 = 36` WDT interrupts

`tempo` is a float that is incremented/decremented by 0.125 at a time. All durations in the `durations[]` array are a multiple of 8, making this scaling possible. The one exception that is not scaled is `durations[0]`, the break. This remains a constant 1, or 1 WDT interrupt (~8ms) per break.

To obtain the frequency and duration of the next note in the song during playback, I use the following two operations:

```
next_freq     = notes[(song[score_counter] & NOTE_MASK)]; // 0x1F
next_duration = durations[(song[score_counter] >> 5)]; // bit shift
```

Virtues and Limitations:

Overall this is a very robust implementation. Since each note in the song is stored in a single byte, the limits on song capacity are exactly the number of free bytes in memory. I estimate that I can store upwards of 10 minutes of music.

Since only 5 bits are used to index the frequency, the number of distinct tones my system can play is a little over two octaves. Unfortunately, any song that spans more than two octaves cannot be played without modification.

Since only 3 bits are used to index the duration, the granularity of my durations is $1/16^{th}$ notes. If a song contains $1/32^{nd}$ notes, dotted $1/16^{th}$ notes, or a triplet of $1/8^{th}$ notes (each equivalent to a dotted $1/16^{th}$ note), the song also cannot be played without modification. An example of this is my version of the "Super Mario Theme" where several triplets must be approximated with $1/16^{th}$ and $1/8^{th}$ notes.

As the buttons are not debounced, they depend entirely on the length of the WDT interval. Depending on the exact moment they're pressed, button presses are occasionally detected twice or not at all. This is especially common with the lower-quality buttons used for the playback functionality.