# MSP430 Hero! Lab Report

Arthur Ames, Keelan Boyle

Work distribution:
Arthur: 90%
Keelan: 10%

Lab intro:

The goal of this lab is to create a guitar hero game using the MSP430 board. To accomplish this, we created a music player module to play songs that were generated by a custom python script. We also created a GUI to allow the player to select between a multitude of generated songs. Finally, we kept track of player score, and showed the player a win or defeat screen at the end of each song, depending on performance.

***Questions***

*In the final version, the countdown must be measured by Timer A2 and NOT implemented using software delays. Explain the difference between event (or interrupt) driven code and polling. Is your final code strictly event driven or will you use a mix of interrupts and polling? Explain in your report.*

Event driven code is code that is called upon an event, instead of code that is called from other code that constantly checks for a set of conditions. An example of this would be the use of Interrupt service routines (events) when a button is pressed vs the use of a while loop that constantly checks if a button was pressed. Our final code uses a combination of events and polling - all of the music and buzzer routines are implemented using event driven programming, while the buttons and keys are checked via polling.

*You will need to do some math to convert these frequencies to number of ACLK tics. Discuss your conversion of frequency in Hz to Timer B CCR0 settings in your report*

Since 32768 ticks is equal to 1 second, and we need to tick our interrupt x times per second, we can see that the math is simple to figure out - (32768 Ticks / 1 Second) * (1 Second / x Hz) = (32768 Ticks / x Hz) gives us a unit of Ticks per Hz. For example, 440Hz is equal to (32768 / 1) * (1 / 440) = ~74 Ticks/Hz. So, we need interrupt every 74 ticks of ACLK, and 74 is the value that goes into CCR0 to do this.

*Explain in you report why software delay would then no longer work and why you must implement note duration using the timer interrupts.*

Software delays are blocking - you cannot do any work on the cpu while they are running, due to the nature of how they work. If we implemented note duration using software delays, we would be unable to do anything while we were waiting for the note to be done playing - making it impossible to poll for button presses.

*Explain in your report how you setup Timer A2 and why Timer A2's resolution should be several times smaller than the duration of a note.*

We setup timer A2 using mostly the same settings that have been used in the previous examples and in class - we used ACLK as a clock source, put the timer in compare mode, and use CCR0 to trigger interrupts to play notes. Since a timer can only interrupt on a multiple value of it’s ticks, we need a resolution that is much smaller than the duration of a single note so that we can alter the note durations - if a note duration and a timer tick took the same amount of time, we would only be able to use whatever the timer tick’s duration was as the length of our note, making our musical pieces sound incorrect. The smaller the timer tick’s resolution, the better, as smaller timer ticks give us the ability to more accurately time each note’s duration, so that we can have a varied and accurate playback.

*Explain your rules for scoring and losing and how you implemented them in your report.*

The rules for scoring our game were fairly simple - if a user pressed an led corresponding to the proper note correctly, they would get 1 point. If they did not pressed an led correctly, they would get no points. At the end of the song, winner or loser status was calculated based on what percentage of the notes a user pressed correctly.

***Music Player***

The music player module was the module responsible for playing the correct notes at the correct times, and also had some helper functions to check if a user pressed the correct LED and to keep track of player score.

The two main data structures used by the music player were Song and Note. Song and Note are defined as follows:

```
typedef struct Song {
      char* name;
      const Note* notes;
      uint16_t length;
      uint8_t bpm;
} Song;

typedef struct Note {
      uint16_t freq;
      uint8_t duration; // Whole, half, quarter, eighth, 16th
      uint8_t leds; // Could compactify defintion by combining with
duration, but no need as of yet
} Note;
```

Songs are then played as follows: using the api function setSong(), the programmer selects a defined song for the music player to play. The programmer can then use the functions playSong(), pauseSong(), stopSong(), and restartSong() to play, pause, stop, or restart the currently active song. These functions are implemented by simply stopping or starting timer A2, as that is the timer that controls the song's playback.

The mechanics of playing the song itself are controlled using the ISR implemented in musicplayer.c. There are 2 modes, BEAT and HOLD mode. Beat mode plays a single ¾ beat per note, i.e. a note plays for ¾ a note duration and then there is a ¼ duration silence, to emulate something like a piano playing. HOLD mode simply holds a tone for the entire duration of a given note. Most songs sounds better in BEAT mode and that is how most of our songs were played. HOLD mode is implemented as such, in the timerA2 isr:

```
if(active_mode == HOLD) {
            note_idx++;
            SET_NOTE_DUR(NOTE_TO_MS(active_song->notes[note_idx].duration,
active_song->bpm));
            BuzzerOn(active_song->notes[note_idx].freq);
            setLeds(active_song->notes[note_idx].leds);
            CLR_TIMER;
            return;
}
```

NOTE_TO_MS is a helper macro that converts classical music notational notes - Whole, half, quarter, eighth, and 16th notes - into a millisecond duration, given a certain tempo, defined in BPM. SET_NOTE_DUR is a macro that converts milliseconds into ACLK timer ticks, and then sets timerA2's CCR0 register to that value.

In essence, the music player works as follows: The timer triggers the ISR -> Timer A2's CCR0 is set to the note's duration, in milliseconds -> The buzzer is set to the corresponding frequency -> -> The timer is cleared -> The ISR returns. This means that music playback happens completely asynchronously from the main program's logic, making the music player very easy to work with - there is no need to poll in a main loop. Music is entirely controlled through the aforementioned api functions, which can be called at any point, allowing the music player to be controlled from the main GUI without having to worry about note timings.

BEAT mode works very similarly to HOLD mode, with one major difference: a boolean variable called "onBeat" is tracked, which alternates every interrupt tick - allowing a note to be played for ¾ time and then toggled off for ¼ time, while still maintaining the asynchronicity of the music player as a whole.

Songs themselves were generated from downloaded sheet music online - we wrote a python script (included with the lab code) to convert online music into the proper format. There is a common sheet music format called musicXML - these xml files were simply parsed and converted into a single generated C header file which could then be copied into the project and compiled with the rest of the program.

Scoring was calculated as follows (excerpt from function handleMusicButtons()):

```
if(buttons && (buttons ^ leds)) {
      LIGHT_RED;
      STOP_GREEN;
      return;
}

if(buttons & leds) {
      score++;
      LIGHT_GREEN;
      STOP_RED;
      return;
}
```

The ‘buttons’ variable was the currently pressed button bitmap, and the ‘leds’ variable was the currently lit leds. The first xor comparison ensures that no buttons are pressed other than the one corresponding to the correct LED, and the second and comparison checks if the current LED’s button is pressed. This function was called from a loop in main, which polled the buttons and passed them into the function. Because the music played asynchronously from the main function, there was no advanced logic required to check note durations or timings - the ‘leds’ variable was automatically updated by the isr when a new note played, and the time since the note began was always contained in Timer A2’s CCR0 register.

LIGHT_GREEN, STOP_GREEN, LIGHT_RED, and STOP_RED are simple macros that light up the user leds on the launchpad board to give the player feedback as to whether they are doing a good job or not.

***GUI***

We also implemented a basic GUI for our game. The GUI had a very similar implementation to Lab 1’s game gui. The gui consisted of a number of states, and a function to switch between them. There were then other api functions to check the keypad and execute the correct gui operation for each state. For example, one of the main states of the program was the *SONGLIST* state - it simply drew a list of all the songs currently available on the device, and allowed the user to scroll between different pages of the song list, to select whichever song they wanted to play.

The main function that handled the input was the handleGuiKey function - this was a function called from the main game loop, and was passed in the polled keypad key. It then called the input handler function corresponding to whatever state the GUI was in; For example, the handleSonglistKey function worked as follows:

```
void handleSonglistKey(char key) {
      if(key >= '0' && key <= '9') {
      int numSongs = sizeof(songList)/sizeof(Song*);
          if((currentPage * 10) + (key - '0') < numSongs) {
                setSong(songList[(currentPage * 10) + (key - '0')]);
                setGuiState(STARTING);
          }
      }

      if(key == '*' && currentPage != 0) {
      setCurrentPage(currentPage - 1);
      }

      if(key == '#' && currentPage != 255) {
      setCurrentPage(currentPage + 1);
      }
}
```

setSong is the music player API function to set the currently active song, songList is the global list of all the songs on the device, and setGuiState is a function that sets the state of the GUI, and then triggers a redraw of the gui based on whatever state it was switched into.

All of the draw functions were simple functions that use the grlib api functions to draw to the screen.

There are currently 16 songs in the header file included with the lab submission, however, the page scheme implemented in the gui supports up to 255 pages of 10 songs each. Song amount is limited by ROM space of course, however, the songs are fairly compact - the current program uses roughly 25Kb of ROM for the entire program, including the 16 songs, some of which are fairly long - Flight of the Bumblebees, for example, is 700 notes long.

Conclusion:

In this lab, we created a robust system for creating, playing, and scoring songs. Songs were generated by a python script. Songs were then played with a music player module that used interrupt service routines to play notes of different durations and pitches. Players could score by pressing the button corresponding to an LED which lit up with each note. We also added a track selection GUI, which allowed the play to select between a multitude of different tracks. Player performance was tracked throughout each song, and the player was told whether they achieved a sufficient score at the end of each song.