Nate Ambrad and Pranav Jain

ECE2049-C24

14 February 2024

Lab2 – MSP430 Hero

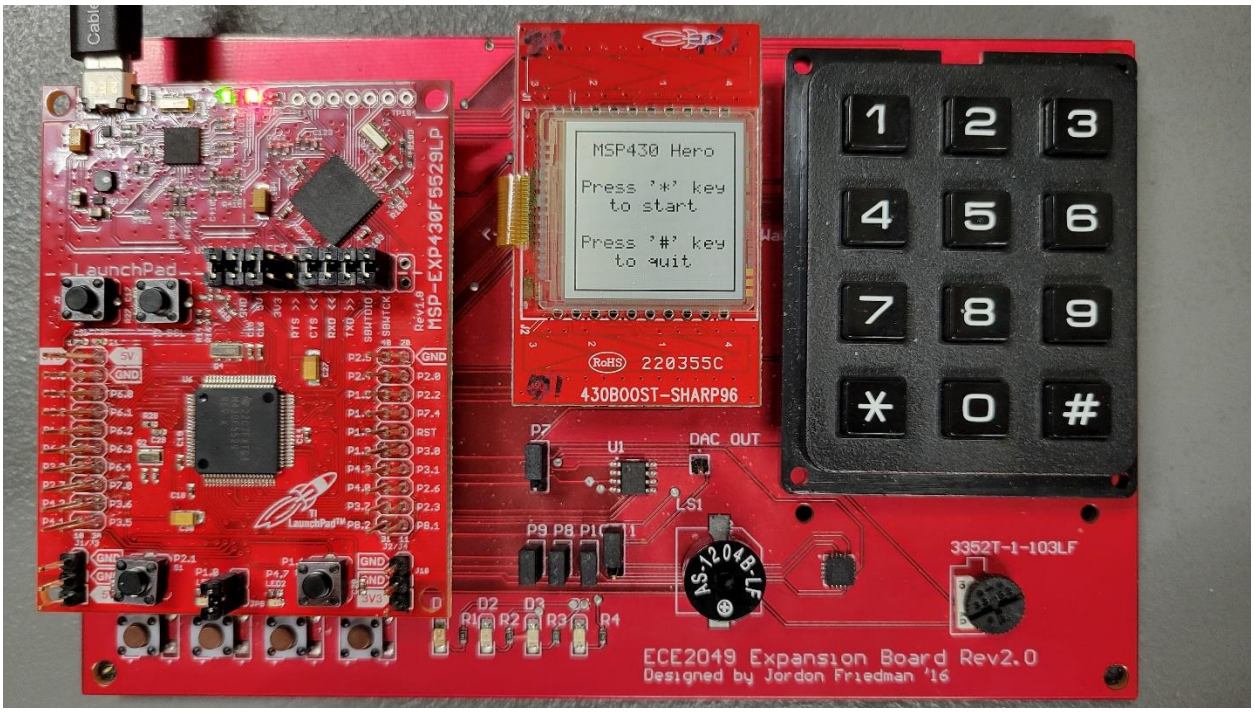

Figure 1 - MSP430F5529 Custom Lab Board displaying Welcome Screen for MSP430 Hero.

Introduction

The purpose of this lab was to create MSP430 Hero, a game which implements the mechanics of the popular video game Guitar Hero on the MSP430F5529 custom lab board. MSP430 Hero is a toned-down version of Guitar Hero, making use of peripherals such as the LCD screen, buzzer, launchpad user LEDs, buttons, custom board LEDs, and the keypad. This lab involved programming an internal timer on the MSP430 as well as a similar state machine structure to the one used in lab1.

The game begins at a welcome screen with a prompt that displays a start or reset option. When the player starts the game, a three second countdown is begun and shown on the LCD with the word “GO” displayed last. Once the countdown finishes, a song is played through the buzzer and the four LEDs are lit up for each corresponding note. The player must use the four buttons next to the LEDs to mirror the LED sequence while the song is running. A score is determined based on how quickly and accurately the player can replicate the LED sequence. After the song concludes, the score is shown, and the game is reset to the welcome screen. The reset button shown at the welcome screen can be pressed at any time during the game to return to the welcome screen, erasing the player’s progress.

Discussion and Results

State Machine Structure

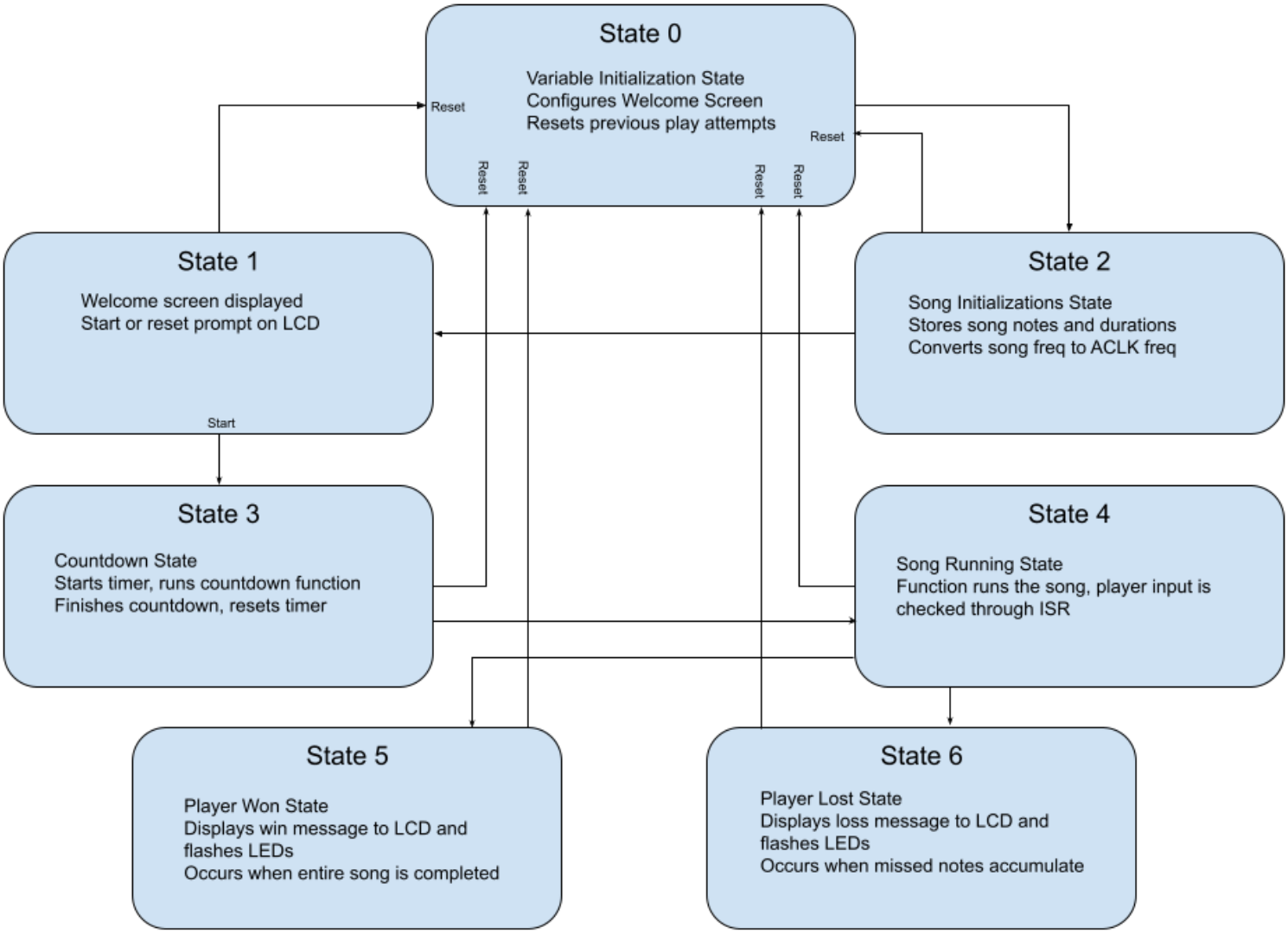

Figure 2 - State machine diagram for MSP430 Hero program. The program consists of seven different states. The ISR was used to check player input rather than having a separate state for it, allowing for the song to play during the checking.

Welcome Screen, Start, and Reset

When the game is first run, a welcome screen will appear on the LCD screen. This screen displays the title of the game “MSP430 HERO” and two prompts: “Press * to start” and “Press # to quit”. There is a function called configWelcomeLCD that writes all of the text to the LCD. The start button begins the countdown for the song by switching to another state and the quit (reset) button will return to the initialization state. Start and reset are both functions that do the same thing. They both scan for a button press and will return to the proper state if pressed. If the button is not pressed, then the current state is returned.

Countdown

The countdown is implemented as its own function. The function countdown takes the parameter called countdown and uses it in a switch case with each case being a number in the countdown and that number is written to the LCD screen. The parameter countdown is put into an if else statement to ensure that once countdown has fully incremented, the song will be played instead of continuing the countdown. Each time the function is called, the function configUserLED is called twice to create a blinking effect with the red launchpad user LED. This was necessary because the green launchpad user LED on the board was broken. If it had not been broken, the red and green LEDs would have alternated for 3, 2, 1 and then flashed together on GO. This would have been easier to implement as a part of each case in the switch statement because by turning on one LED, the other one can simultaneously turned off. Using only the red LED provided an early but solvable obstacle in the program.

Use of Buzzer

The buzzer was used to play the various notes of the song. There was a total of 9 notes, which also included a note for no sound.

Song Storage and Playing

The song and its duration were hard coded in two arrays. The duration was stored in the format of ticks instead of seconds.

Use of Timer

Calculations for TimerA2:

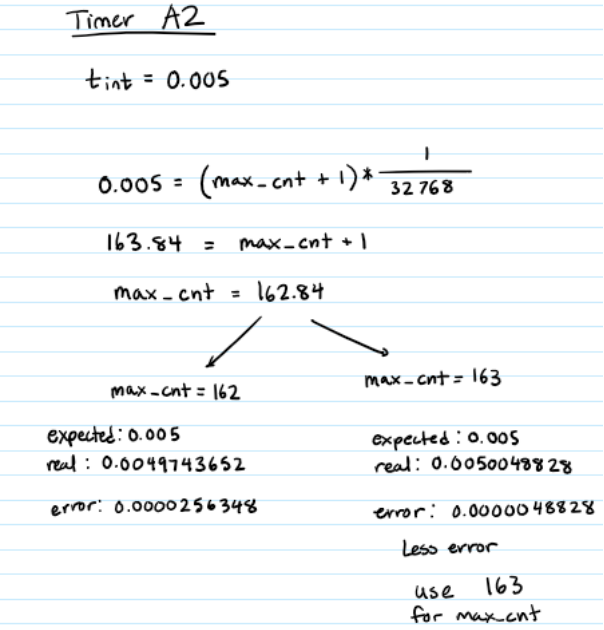

Figure 3 - Calculations done to find max_cnt for TimerA2. The value of 163 was used because it had the lowest error compared to the expected interval of 5ms.

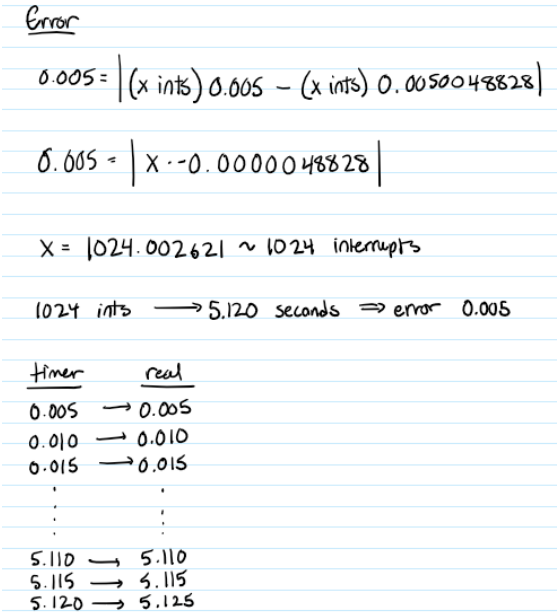

Figure 4 - Calculations done to find the number of interrupts before 5ms of error accumulates on the timer. The timer ran slow, so the error was +5ms rather than -5ms.

Player Input

The player input is checked at every interrupt. If the player has pressed a button, it compares it to the one stored in it memory and lights up the user led to approve if the correct button is clicked.

Victory, Defeat, and Scoring

The player can win if they can play through the whole song without lagging behind. The player would lose if they missed 8 notes. The player can earn up to 3 points for every correct input. Each note has a specific duration that it plays. That duration is divided into 3 sections and based on which section the player has the correct input, they can earn up to 3 points.

Highlighted Questions

Explain the difference between event (or interrupt) driven code and polling.

Event driven code is determined by events such as user input onto a peripheral such as the push of a button. This code is written to have a protocol or callback for each event of interest. Event driven code is efficient because it only reacts when events occur. Polling involves repeatedly checking a condition or state at a regular interval for changes. Polling is less efficient than event driven code because CPU resources are wasted checking events even if they do not occur.

Is your final code strictly event driven, or does it use a mix of interrupts and polling?

It uses a mix of interrupts and polling. Polling occurs when the code goes through if statements because they will be checked for the proper conditions regardless of if those events are happening. For example, we use polling to check whether the hash “#” key has been pressed to

know when to reset the game. Here, polling is important as we need to continuously check whether the player has clicked the hash key which would then lead to resetting the game.

Discuss your conversion of frequency in Hz to Timer B CCR0 settings.

For each of the possible notes, the reciprocal of the frequencies were found to get the period. This was then multiplied by the frequency of the ACLK which is 32768 Hz. The final value of these calculations was given to Timer B CCR0 settings based on which note needed to be played.

How did you control the duration of your notes?

The durations of the notes were hard coded in an array in the form of ticks. Timer A2 was used to get the current time and hence know when the duration of the current note ended and the next note had to be played.

Did you do this within the buzzer function or within the main game loop?

The duration of the notes were controlled within the main game loop with the Timer A2 interrupt allowing MSP430 to know when to play the next note.

Why don't software delays work when you need to be checking button presses and why must note duration be implemented using the timer interrupts.

Software delays do not work when you need to check for button presses because software delays are blocking, meaning when the software delay occurs, it is the only thing being run, so inputs such as a button press, or the reset button will be blocked for the duration of the delay.

The note duration must be implemented using timer interrupts as we need the program to keep running. This would also be crucial to know if the player has pressed a button while the note was playing, which would be impossible with software delays as software delays are blocking functions.

Explain how you setup Timer A2 and why Timer A2's resolution is several times smaller than the duration of a note.

TimerA2 is used in run timer function and is configured to use ALCK, 16-bit, up mode, with 1 divider. The max_cnt is set to 163 to give a interrupt interval of very near 5ms. Interruptions are enabled. There is also a stop timer function that stops the timer, resets the count, and disables interrupts if a reset parameter is met.

Timer A2's resolution needs to be several times smaller than the duration of the note for the program to understand the length of each note. A higher resolution would make the program/ timing errors too high for the song to be played.

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

The player can win if they can play through the whole song without lagging behind. The player would lose if they missed 8 notes. The player can earn up to 3 points for every correct input. Each note has a specific duration that it plays. That duration is divided into 3 sections and based on which section the player has the correct input, they can earn up to 3 points.

Summary and Conclusion

To summarize, this lab's objective was to develop MSP430 Hero, an adaptation of Guitar Hero for the MSP430F5529 custom lab board. The game utilizes various peripherals to create a functional, dynamic, and immersive game. The structure of the program follows a state machine model but makes use of TimerA2 and tracks user input as the program is being run. The use of a buzzer enables the playing of songs, with note durations controlled through timer interrupts. The game requires players to mimic LED sequences using buttons, with scoring based on accuracy and speed. The implementation involves a mix of interrupts and polling, with timer configurations for precise timing. Scoring and win/lose conditions are also defined within the program logic. The game is tied together by a C program which controls each aspect mentioned above in harmony, creating one while package.

In conclusion, this lab achieved its objective of emulating the Guitar Hero experience. The visual and audible cues of the various colored LEDs and changing pitched buzzer make MSP430 Hero reminiscent of its inspiration. The current product is a good building block to improve and make into an even more refined version of the classic game. As it currently is, MSP430 Hero demonstrates a strong utilization of the board's limited resources and peripherals and knowledge of important programming techniques.

Appendices

```
// Nate Ambrad
// ECE2049-C24
// Lab2 Prelab
// 30 January 2024

// 1) Read entire lab assignment
// 2) void configButtons();
// 3) char buttonState();
// 5) void configUserLED(char inbits);
// 4) Possible data structure for notes
        typedef struct {
                int pitch;    // note pitch
                int duration; // note duration
                char led;     // note LED mapping
        } Note;
        // minimun song length is 28 notes
        // 2 * int + 1 * char = 3 bytes per note
        // 28 * 3 = 72 bytes per song minimum

#include <msp430.h>
#include <stdio.h>
#include "peripherals.h"

// function prototypes
void configButtons();
char buttonState();
void configUserLED(char inbits);

// functions
void configButtons() {

        // Select P7.0 (S1), P3.6 (S2), P2.2 (S3), P7.4 (S4) for digital I/O
        P7SEL &= ~(BIT4|BIT0); // xxx0 xxx0
        P3SEL &= ~BIT6;         // x0xx xxxx
        P2SEL &= ~BIT2;         // xxxx x0xx

        // Set as inputs
        P7DIR &= ~(BIT4|BIT0); // xxx0 xxx0
        P3DIR &= ~BIT6;         // x0xx xxxx
        P2DIR &= ~BIT2;         // xxxx x0xx

        // Push-up/down Resistor enabled
        P7DIR |= (BIT4|BIT0); // xxx1 xxx1
        P3DIR |= BIT6;        // x1xx xxxx
        P2DIR |= BIT2;        // xxxx x1xx

        // Push-up selected
        P7OUT |= (BIT4|BIT0); // xxx1 xxx1
        P3OUT |= BIT6;        // x1xx xxxx
        P2OUT |= BIT2;        // xxxx x1xx
}

char buttonState() {

        // state stores which buttons are pressed, initialized to 0x00
        char state = ~(BIT7|BIT6|BIT5|BIT4|BIT3|BIT2|BIT1|BIT0);

        // check which buttons are pressed and update state
        if ((P7IN & BIT0) == 0)
                state |= BIT0;
        if ((P3IN & BIT6) == 0)
                state |= BIT1;
        if ((P2IN & BIT2) == 0)
                state |= BIT2;
        if ((P7IN & BIT4) == 0)
                state |= BIT3;

        // return updated version of state
        return state;
}


void configUserLED(char inbits) {

        // Select P1.0 and P4.7 (2 User LEDs) for digital I/O
        P1SEL &= ~BIT0;
        P4SEL &= ~BIT7;

        // Set P1.0 and P4.7 as outputs
        P1DIR |= BIT0;
        P4DIR |= BIT7;

        // Check if LED1 should be on
        if ((inbits & BIT0) == 1) {
                P1OUT |= BIT0; // LED on
        } else {
                P1OUT &= ~(BIT0); // LED off
        }

        // Check if LED2 should be on
        if ((inbits & BIT7) == 1) {
                P4OUT |= BIT0; // LED on
        } else {
                P4OUT &= ~(BIT7); // LED off
        }
}
```

Figure 5 - Nate Ambrad's prelab assignment which includes the functions configButtons, buttonState, configUserLED, and a possible data structure for song notes.

```
void buttonConfig(){
    P7SEL &= ~(BIT0|BIT4);
    P7DIR &= ~(BIT0|BIT4);
    P7REN |= (BIT0|BIT4);
    P7OUT |= (BIT0|BIT4);

    P3SEL &= ~BIT6;
    P3DIR &= ~BIT6;
    P3REN |= BIT6;
    P3OUT |= BIT6;

    P2SEL &= ~BIT2;
    P2DIR &= ~BIT2;
    P2REN |= BIT2;
    P2OUT |= BIT2;
}

void configUserLED(char inbits){
    P1SEL &= ~(BIT0);
    P1DIR |= (BIT0);

    P4SEL &= ~(BIT7);
    P4DIR |= (BIT7);

    if((inbits & BIT0) == 1){
        P1OUT |= BIT0;
    }
    else if((inbits & BIT0) == 0){
        P1OUT &= ~(BIT0);
    }

    if((inbits & BIT1) == 1){
        P4OUT |= BIT1;
    }
    else if((inbits & BIT1) == 0){
        P4OUT &= ~(BIT1);
    }
}

char buttonState(){
    char bState = ~(BIT7|BIT6|BIT5|BIT4|BIT3|BIT2|BIT1|BIT0);

    if((P7IN & BIT0) == 0){
        bState |= BIT0;
    }
    if((P3IN & BIT6) == 0){
        bState |= BIT1;
    }
    if((P2IN & BIT2) == 0){
        bState |= BIT2;
    }
    if((P7IN & BIT4) == 0){
        bState |= BIT3;
    }

    return state;
}
```

Figure 6 – Pranav Jain's prelab assignment which includes the functions configButtons, buttonState, and configUserLED.

Other Items

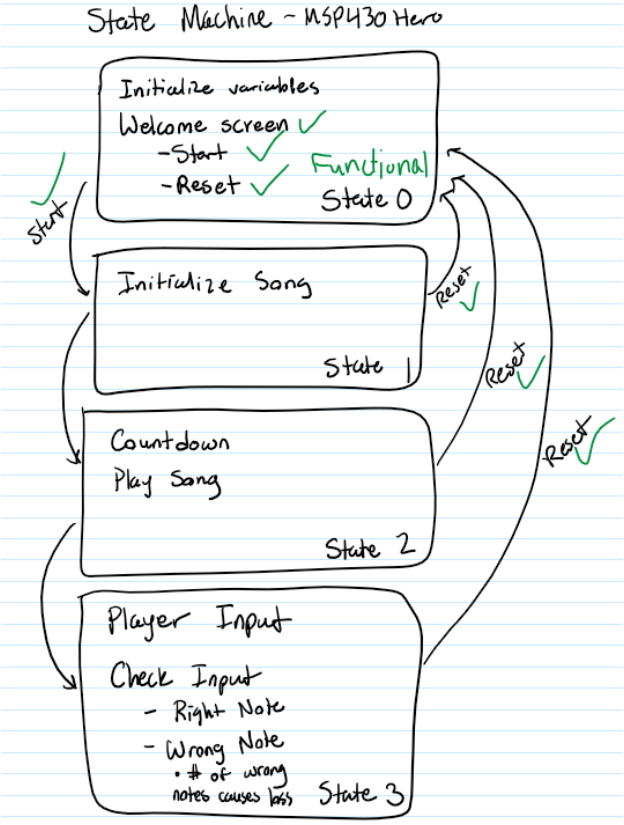

Figure 7 - Original hand-drawn state machine diagram for MSP430. Included four states: Initialize/welcome, song initialize, countdown/play song, and player input/check.

Note	Hz
G2	784
F2	698
D2	587
C	523
Bb	466
G1	391
F1	349
D1	294
rest	0

Figure 8 - Hand-drawn chart of notes used on song. The song "Money for Nothing" is in a G minor scale, so the notes given in the lab instructions were not all necessary or sufficient.