Prepared by: Quan Zhang, Xiaodan Tan
Group Number: 12
Date: 2015-12-3

ECE 224 Lab Report 2

Contribution & Self-reflection

Quan Zhang:
We built the hardware part and the first part of software together. For the hardware, My partner was building the circuit and I did the double check for the hardware. For the first part of software, we implemented it together since it is relatively straightforward. For the second part of the program, I was responsible for the half speed and reverse function and the implementation of interrupt handling for buttons. I did learn a lot from this lab. The experience of building hardware makes me more detail-oriented. What is more, I got a much better understanding of the concept of interrupt when implementing the interrupt handler. Through writing the code, I am more familiar with C, especially with the use of pointers. Also, I learnt how the data is managed in WAV file and how to read a piece of data and play it. I did encounter some problems when implementing the reverse function. After discussing with my partner and TAs, I found the correct way to find the last sector.

Xiaodan Tan:
In Part I, I was connecting up the pins to the controllers and typing out all the names, while my partner was responsible for double-checking them with the given schematic to make sure they matched correctly. We also implemented Part II together and played the audio in normal mode. In Part III, for software interfacing design, I was responsible for implementing a tight polling for the dip switches checking for play mode while waiting for the play button to be pressed. For audio playback mode, I chose to implement double speed and delay channel. However, my design for the delay channel did not quite work out and we ended up implementing this playback mode together after seeking help from the lab instructor.

Upon completing this lab, I got to learn how data is stored in FAT32 format and also how to read data in such a format from the SD card. Also, I learned how to manipulate an audio wav file in packet level to achieve different playback modes. I realized how the correct choice of data structures can significantly improve the software efficiency and performance.

Overview of system design

The system contains a single logic flow in an infinite loop. The flow starts off by enabling interrupts on all the buttons to allow users to do the maximum amount of controlling. Next, we tight poll the status of the PLAY button. While the PLAY button is not pressed, the system will check if a new play mode or a new song is selected and update the LCD accordingly. For the efficiency purpose, the system does not do any buffering or reading from the SD and will only show the cached file names onto the LCD. Once the PLAY button is pressed, the system will

immediately disable interrupts on the browsing buttons and PLAY buttons, so that the system only allows the user to stop playing audio but not change any other playing settings. Next, the program will start buffering the data from the SD card and proceed to play the audio in the selected play mode.

```
main(){
    initialize SD Card and FAT file system
    initialize push button interrupt vector
    initialize audio_codec

    fetch all the wav files and store their data_file into array data_info

    while(1){
        enable interrupt on button 3 down to 0
        Reset the edge capture register

        while(PLAY btn is not pressed){
            switches have been changed:
                set play mode based on switches
                update LCD
            browsing btns are pressed:
                update LCD
        }
            disable interrupts on buttons except STOP

            build cluster chain

            invoke corresponding method based on play mode
    }
}
```

The system interacts with the user interface through button interrupts. The interrupt handlers share the following global variables with the main program: STOP, PLAY, update_index and file_index. The first three are flags that can be polled by the main program to stop play, start playing and update selected file on LCD respectively. The last one, on the other hand, points to a given index in the file list which can be used to select the file to be played. This variable is incremented or decremented when the browsing buttons are pressed.

```
Push_Button_ISR(){
    reset button 3 to 0
    handle pressed buttons:
        btn 0: set STOP flag to 1 indicating the stop button is pushed
        btn 1: set Play flag to 1 indicating the resume button is pushed
        btn 2: increase file index & set UPDATE_INDEX flag to 1
        btn 3: decrease file index & set UPDATE_INDEX flag to 1
    reset edge capture register
}
```

The followings are brief description of how we implemented each playback mode and its associated pseudo-code.

In Normal() function, we fetch each sector in the buffer using get_real_section() method. After getting the 512 bytes data in each sector, we use an inner for loop to play each byte data stored in the current sector.

```
Normal(){
    for(each sector){
        buffer the data in the current sector

        for(each 2 bytes in current sector){
            play the sound
        }

        if(STOP){
            reset STOP to 0
            return
        }
    }
}
```

The Double_speed() function plays the audio in double speed. We use the same method to get each section. However, in order to play the audio in double speed, we need to loop the data at a step of 4 bytes instead of 2. This is because we need to skip every other 4 bytes of audio, which is one packet of audio containing sample pair for each channel. A on_off is toggled each time we step up by 4 bytes and it indicates whether we should play the current 4 bytes.

```
Double_speed(){
    on_off = 0

    for(each sector){
        buffer the data in the current sector

        for(each 4 bytes in current sector){
            if (on_off == 0) {
play the sound
}
else {
    skip
}

toggle on_off
        }

        if(STOP){
            reset STOP to 0;
            return;
        }
    }
}
```

The Half_speed() function plays the audio in half speed. In order to play each channel pair twice, we used a third inner loop to do the repeating. It is worth mentioning that we ensured the audio data played on a given channel will be repeated on the same channel by repeating on 4 bytes instead of 2.

```
Half_speed(){
    for(each sector){
        buffer the data in the current sector

        for(each 4 bytes in current sector){
            repeat twice:
play the first 2 bytes
play the second 2 bytes
        }

        if(STOP){
            reset STOP to 0
            return
        }
    }
}
```

The Delay() function plays the audio in channel 1 at real-time and the audio in channel 2 in delay of one second. We calculated the number of bytes occupying 1 second of audio data and created a circular buffer with this size. The initial values of this circular buffer are 0, so that in the first second, Channel 2 will be playing blank audio data. While playing the audio in Channel 1 fetched fresh off the SD card, the interleaved Channel 2 data is pushed into the circular buffer. Since the reading point is always 2 bytes ahead of the writing point, we simply kept one pointer to traverse the array. However, we had to make sure we read the data off the circular buffer before we write new data onto the same slot to avoid overwriting data.

```
Delay(){
    initialize a circular buffer with a size of 88200(the amount of data to be played in 1s)

    for(each sector){

        buffer the data in the current sector

        for(each 4 bytes in current sector){
            play the first 2 bytes on Channel 1
        }

        for(2 bytes in circular buffer){
            play the delayed sound on Channel 2
        }

        push the second 2 bytes from buffer to circular buffer

        if(STOP){
            reset STOP to 0
            return
        }
    }

    play the last second of delayed channel
}
```

The Reverse() function plays the audio backwards. We first calculate the index of last sector of a song by dividing the file size in bytes by the constant "BytsPerSec". It is very possible that the last sector is not full and the size is returned by "get_real_sector" function. This returned value is our starting point of playing. One detail to notice is that we have to play each channel byte pair just like we do in normal mode, since the most significant bit of each channel byte pair should not be reversed to preserve the sound of each sample.

```
Reverse(){
    calculate the index of the last sector
    for(every sector backwards){
        buffer the data in the current sector

        for(each 2 bytes in current sector){
            play the audio pair in the ordinary way
        }

        if(STOP){
            reset STOP to 0
            return
        }
    }
}
```

Issues with hardware design/software interfacing

This lab includes the use of buttons and switches. The major hardware design is to determine the interface synchronization techniques (interrupt or polling) for both buttons and switches.

Final decision and discarded solutions to the above issue

Polling and interrupt both have their own advantages and disadvantages. Based on the experiment conclusion in Lab1, more background tasks can be done with the use of interrupt compared with polling. In the context of this lab, the goal of the main program is to play audio continuously at a fixed high frequency. That is, the system should not spend too much time or even all its time on polling devices to ensure a good quality of audio playback. Thus, interrupt is selected and polling is discarded for the buttons.

On the other hand, when the audio is stopped, the audio codec is at idle and it is the only time where play modes can be changed. In other words, no heavy-duty background task is performed and the system is doing nothing but synchronizing with the user interface under this circumstance. Therefore, polling is a better choice, due to its minimal latency and ease of implementation compared with interrupt. Thus, interrupt is discarded for the switches.

Testing/debugging strategy and discarded alternatives

Unit testing strategy is used the most frequently in this lab. To be more specific, hardware interfacing and audio playback software functionalities were tested separately. To isolate audio playback functionalities, we first wrote out the skeleton of the program including the main function with switch statement that jumps to different playback mode methods. The logics were simply placeholders with printf() that can print out the appropriate debug messages.

Following the rule of unit testing, we divided the hardware testing into two parts based on the system requirements: 1) the functionality of each button & switch while the audio is stopped; 2) Interface is disabled except for stop button while the audio is playing.

To test the first requirement, we first pressed stop button so that the system went into the tight polling stage. We made changes to different buttons and switches to observe if the correct debug messages were printed out to the console. In this way, any interfacing issues for requirement 1 were found and fixed at the early stage of development.

Moving to the second requirement, after pressing play button, only stop button should be enabled. That is, no other debug messages except for “stop button is pressed” should be displayed, even if other buttons and switches were pressed and switched. Also, we tested if we invoked the right playback mode function corresponding to the user-selected mode by checking the associated print statement in the console. Up to this point, we can confirm that the hardware was working properly and we could go ahead to implement the software part.

This test strategy made debugging the complete program much easier. For example, we got an issue in which the debug print “next song” or “previous song” were printed twice when pressing next/previous button. This effectively indicated pressing next/previous button would cause the system to jump over two songs instead of just one. Noticing this problem, we realized the button

handler was not written properly and the file index was somehow incremented twice. The reason for this was soon figured out - the interrupt service routine was invoked twice at rising and falling edges when the button was pressed. Therefore, extra logic was added to distinguish the rising and falling edges.

We discarded the strategy of implementing the entire program including both hardware interfacing and software audio playback and testing the system as a whole. This is because by the point where the full system was coded, a number of issues could potentially stack up and interleave with each other, making it difficult to debug. Doing unit testing will make sure we have a working system before making more changes and make it easier to tackle issues coming from different functionalities one by one.

Issues impacted on the audio playback performance

One playback performance issue we solved was in Reverse playback mode, whether the order of each byte from the audio or the order of channel packet should be reversed. The goal of the reverse playback mode is to create an effect of playing audio backwards and yet preserve the audio quality. Thus, we needed to try implementing both designs and determine which design could create the effect we expected.

Issues with writing efficient software

In Delay playback mode, a circular buffer was used to store audio samples from the delayed channel (channel 2). Therefore, a write/read index pointer was incremented and wrapped around back to zero whenever reaching to the end. A software efficiency issue rose: an efficient way was required to update the index pointer to the circular buffer for reading and writing the delayed channel data. If the buffer can not be read or written to at a speed that catches up with the audio play frequency, the audio will sound stalling due to insufficient audio data in the FIFO.

Software design decision for the above two performance issues

Regarding the issue for the Reverse mode, the final design was to reverse the order of channel packet only. The discarded design, reversing the order of each byte, produced soft noise on the speakers instead of reversed audio. In this implementation, the order of the 2 bytes in each audio channel packet was also reversed. As a result, the least significant bits were magnified and played before the most significant bits of each audio channel packet. However, the most significant bits of each packet carry the most important information of the audio packet and the least significant bits are usually noise. Thus, the most significant bits should alway get magnified and played before the least significant bits. In other words, the order of the 2 bytes in each audio channel packet should be preserved and remain un-reversed.

For the software efficiency issue, a simple “if statement” was used to reset the index back to 0 in cases where the incremented index is greater than the buffer size. The discarded alternative was to calculate the updated index by taking modulo of circular buffer size. This is due to the fact that value comparisons are generally fast operations on processors. However, the NIOS II processor designed in the project does not have a native hardware support for modulo operator. Consequently, a significant amount of time will be spent on the execution of modulo and the audio data cannot be pushed into the FIFO fast enough to keep up the audio playback frequency.

Thus, a highly distorted audio signal was produced when modulo operator was used. Also, we discovered that the read pointer was always two bytes ahead of the write pointer. As a result, only one index pointer was needed, as long as we made sure the program read off the buffer before writing new data into the same location. Thus, the final design only had to update one pointer instead of two, reducing the number of operations and increasing the system efficiency.

Testing/debugging strategy with regard to audio playback

The testing strategy used to debug the audio playback quality was simply listening to the audio repeatedly and examining the code directly. For example, in Delay mode, we listened to the audio and found out the last second from the delayed channel was missing. Then, we added an extra loop to play the leftover audio from the circular buffer one more time. After adding this extra loop, the last second of audio was played but in a distorted way. The common reasons for distorted audio were either the audio was not processed fast enough or the channels were messed up. Then, we went back to examine this loop in the code. We discovered we effectively split the second channel audio data onto two channels, since the codec was always expecting channel 1 and channel 2 interleaved. Thus, we stuffed zeros in between the second channel audio so that channel 1 was playing empty data while channel 2 was playing the last second. Next, the audio was played correctly but the last second was repeated, which indicated the counter of the last loop was doubled. Thus, the loop was fixed again with the right counter and the audio was played as expected.

The testing strategies discarded for playback performance were using printf() functions to print out debug messages or using Debugger to set breakpoints. The main reason for this is the audio is played in real time at 44.3 kHz. Any delay introduced by printf() or debuggers to the program will definitely cause the system to malfunction, defeating the purpose of debugging.

Possible future extension

Possibility 1 - Discrete progress bar

A series of consecutive LEDs can be used to show the progress of the playing song. LEDs can be lit up one by one from the left to the right to indicate the song is playing forward. Thus, each LED will point to a certain point in the song. While playing a selected song, pushing up the dip switch right below any LED and pressing the PLAY button will jump to the corresponding position within the song.

Possibility 2 - Cutting audio

Implement a “cutting audio” function to make a short clip of audio out of a given audio file. Designate a button x to receive the “cutting point” position. The user can let the song play and when the button x is pressed the first time, the system memorizes the position within the song and sets it as the starting point. When button x is pressed the second time, the system sets this point in the song to be the end point of the clip and stops playing the song in the meantime. Now, a new audio file containing this short clip of audio will be created and saved onto the SD card.