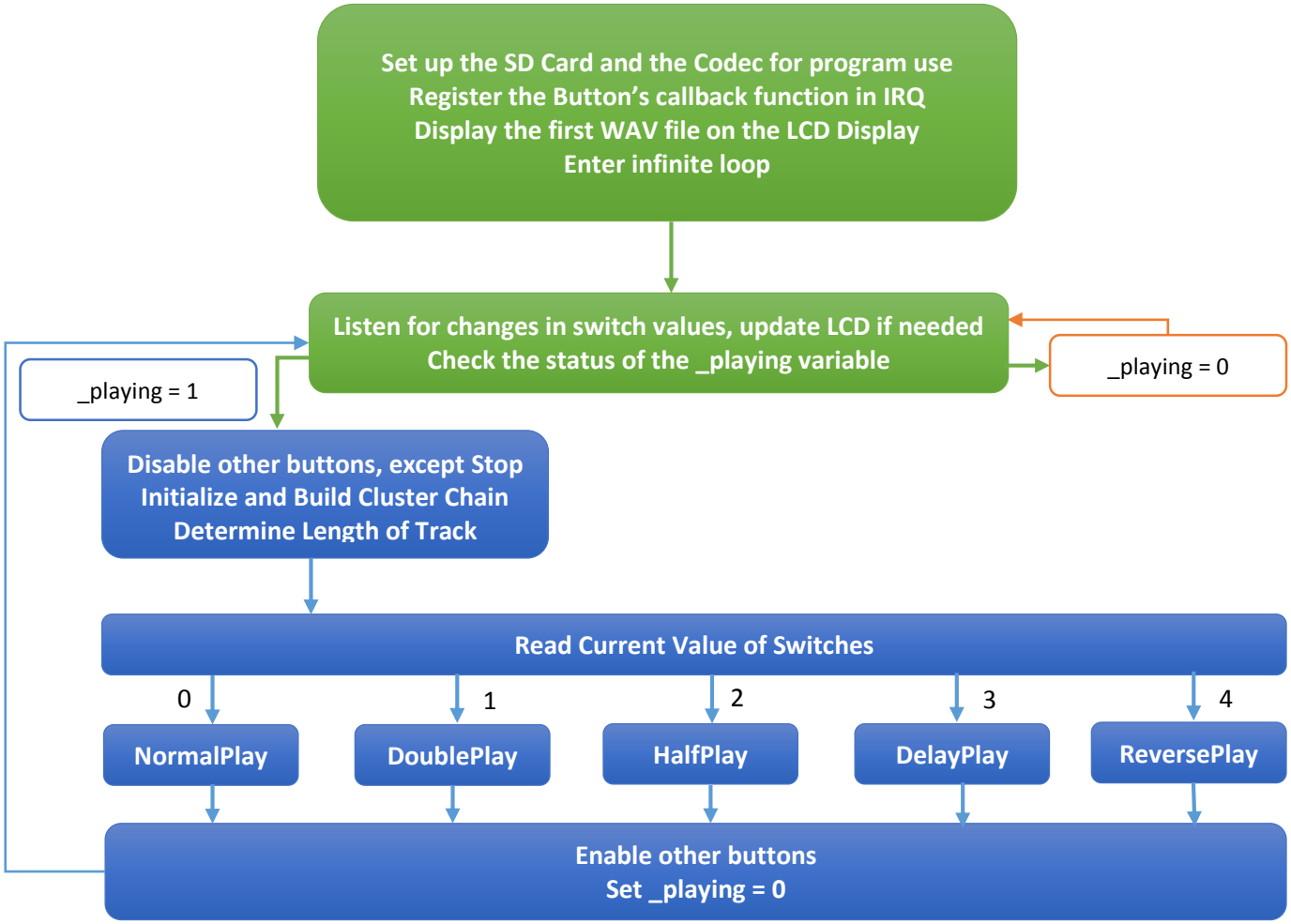
ECE 224: LAB 2

Ramie Raufdeen

Joey Keum

Overview of software design

Process Diagram (Main Method)



Switch values are read in binary format, and up to 3 switches are used. Example; if the switch pattern is 001, which translates to a value of 1, then the playback mode is mapped to the *DoublePlay* function.

Utility Functions

Setup()

The setup function primarily initializes the SD card functionalities in order to allow read and search operations with the SD Card. The audio coder/decoder is then initialized in order to play the audio files, finally we enable all the buttons and register the method to handle the button ISR.

```
//Setups up required aspects for the program to run.
void Setup(){
    SD_card_init();
    init_mbr();
    init_bs();
    init_audio_codec();

    IOWR(BUTTON_PIO_BASE, 2, 0xf);
    IOWR(BUTTON_PIO_BASE, 3, 0x0);
    alt_irq_register(BUTTON_PIO_IRQ, (void*)0, button_ISR);
}
```

The '_playing' Variable

A shared variable called *'_playing'* is used throughout the program to make decisions on whether to play the song. For example, it is used in the *main()* method to check whether the main program should even try to play the song. Furthermore, it is used in each individual method that buffers and plays the audio file; while playing, the *'_playing'* variable is checked to determine whether the method should break out of its song playing loop (see figure below).

This variable is set primarily by the user via the button ISR (user presses play or stop button), however it is also used in the main method and set to 0 when the song is done playing; self-explanatory.

```
//Play song in regular speed
int NormalPlay(data_file file_, int l_, int* clusters_)
{
    int i, j;
    BYTE playBuffer[BUF_SIZE] = {0};

    for (i = 0; i < l_ * BPB_SecPerClus; i++)
    {
        //If play flag is defaulted (user stops music), then stop playing from buffer
        if (_playing == 0)
            break;
```

button_ISR

The interrupt service routine for buttons are implemented using *button_ISR()*. The routine only handles one edge (falling edge); or else a duplicate request would be received in addition. The method consists of a simple switch statement to handle the *PLAY*, *STOP*, *NEXT*, and *PREV* functionality. It is primarily responsible for handling the value of the *_playing* variable and iterating through the WAV files.

DisplayStatusLCD

This method reads the switch values and displays the play mode and WAV file name on the LCD display.

```
//Get the current switch value and display the paly mode & song to LCD
static void DisplayStatusLCD()
{
    _swstate = _swstate & 0x07;
    LCD_Display(_fileinfo.Name, _swstate);
}
```

UINT16 BytePrep

This method is responsible for preparing a couple of bytes (left shift second byte, and OR it with first) from the buffer to prepare to push to the Audio player; this is called by the all the playback modes.

```
UINT16 BytePrep(int i, int j){
    //left shift j byte and or it with i, to prepare to write to audio
    return (UINT16)((j << 8) | i);
}
```

Playback Functions

All playback functions follow a standard set of parameters to accept:

- *data_file* (struct) → contains information about the audio file to be played/read
- *l_* (int) → the length of the clusters built in the main method calling the playback function
- *clusters_* (pointer) → a pointer to the cluster chain that has been built to play

Preface Notes

- The audio file consists of multiple clusters; and each cluster consists of multiple sectors. Each sector contains various bytes for channels which are 8 bits each.
- Buffered array is formatted {L, L, R, R, L, L, R, R….}, the highlighted green is how data is sent to the codec's FIFO buffer, and the overall highlighted data represents 1 sample.
- The sample rate to play audio at normal speed is 44.1K samples per second

NormalPlay

This is the fundamental playback function which sets the base standards for how other playback functions are programmed and implemented. The requirement of this is to play the audio file at normal speed. The function is implemented by iterating through all of the sectors for the clusters; this is noticeable in the first for loop which utilizes the *'get_rel_sector'*. This helps retrieve a specific sector and load it into the play buffer. After loading the buffer with data to play, we iterate through the buffer and write to the Audio Codec. Data is written to the audio codec by waiting to ensure the FIFO buffer isn't full. When space is available, 16 bits of data is prepared using the *BytePrep* function. In general the buffer loop sends 16 bits of left-channel data first, and then 16 bits of right-channel data in the next iteration. The time difference in sending the data to different channels each iteration isn't noticeable to the human ear, so it sounds like everything is in sync. During an iteration of each sector, a check is done to see if the *_playing* value is set to 0, which means the music should stop and we break out of the playback loop.

```
//Play song in regular speed
int NormalPlay(data_file file_, int l_, int* clusters_)
{
    int i, j;
    BYTE playBuffer[BUF_SIZE] = {0};

    for (i = 0; i < l_ * BPB_SecPerClus; i++)
    {
        //If play flag is defaulted (user stops music), then stop playing from buffer
        if (_playing == 0)
            break;

        //Buffer current sector (i) from cluster into the playBuffer
        get_rel_sector(&file_, playBuffer, clusters_, i);
        for (j = 0; j < BUF_SIZE; j+=2){

            //Wait for signal to write for audio, then write to it with prepared bytes
            while(IORD(AUD_FULL_BASE, 0)){}
            IOWR(AUDIO_0_BASE, 0 , BytePrep(playBuffer[j], playBuffer[j + 1]));
        }
    }
}
```

DoublePlay

Similar logic to *NormalPlay*, except we skip every other sample from the buffer since the requirement is to play double the speed of *NormalPlay* (double the sample rate). Since a sample consists of 4 bytes of data, every time we hit a 4^{th} byte, we skip the next 4 bytes of the buffer (using the value of the iterator *j*).

HalfPlay

Similar logic to *NormalPlay*, except we duplicate every other sample from the buffer since the requirement is to play half the speed of *NormalPlay* (Half the original sample rate). Just like *DoublePlay* we mod *j* by the number of bytes in a sample, however instead of skipping; we decrement it by 4 to play the sample again. We use the value *r* so that we don't end up decrementing and incrementing repeatedly causing an infinite loop!

ReversePlay

Same procedures as *NormalPlay*, except we play from the last sector and iterate backwards by sector. In addition, we play the buffer from the last byte to the first byte. The requirement is to play the audio file in the reverse order (by samples). One important catch is that the buffer decrements by 6, the reason for this is so that we can play backwards by sample and not each individual byte.

DelayPlay

This method varies mostly from Normal play because a duplicate buffer is used to delay the samples that are being sent to the Codec. This method skips the values on the right side of the sample and stores it in a buffer, and plays it in the next time it reaches the same code block. The *idxDelay* variable is used to keep track of where the function is currently in the delay buffer. In essence, the processed values of the right side of the sample are stored in a separate buffer and played during the next iteration of the right side of the sample, delaying it by 1 sample. Upon completing the original buffer, the remaining delayed values are iterated and played using 2 additional for loops within the sector.

Software/Hardware Design

Issues and Resolutions

A primary issue we had which wasn't in the requirements was the file browser getting stuck at a specific file when browsing previous. This was extremely annoying as we would have to restart the whole program to get a different file to play. We fixed this issue by adding a check inside the ISR for the previous button case to prevent the *file_number* variable from going below 0 upon decrementing. The knowledge was gained by browsing through the *search_for_filetype* method and seeing that it increments by 1 within it, so in order to decrement the *file_number* by 1 we had to subtract 2 from it to make up for the incremental 1 inside the method provided to us.

Updating the playback mode onto the LCD display on demand based on switch values was another issue we faced in this lab. The LCD screen was not updating whenever the switch for the feature were changed (i.e normal -> double, delay -> reverse, etc.). It would only change when we pressed the play button or the previous/next button. It would not change dynamically with just the switches changing (except when it is playing of course). Initially we thought we had to implement a separate ISR for the switches, however we realized that would be a bit of an overhead. As a result, we called the *DisplayStatusLCD()* inside the while loop within main. However, we also noticed that the LCD display

would flicker; to fix this, we implemented a state variable to keep track of the previous switch values and only update if changed (within the while loop), and store the current into the previous. What this has allowed is that it frees up the program whenever the switch state is the same. In the method before, it would constantly re-display the LCD regardless of the state which could have tied up the program unnecessarily.

We were looking to modularize the code further by implementing a base method which would somewhat resemble *NormalPlay* and allow other playback methods to call it while handling their buffer play different. Initially we started planning this out, but we realized there were way too many state variables, indices, lengths, and logic to handle to satisfy this; especially since C wasn't a strong programming language for both of us. The tradeoff for elegant code in exchange for the time wasn't worth it so we decided not to implement a base method for playback.

Debugging Strategies

Primarily for debugging we used breakpoints and ran the program in debug mode; although the audio was not clear. We toggled breakpoints at specific pieces of the code that we had problems with or that we wanted to test, and then stepped through it using the NIOS IDE.

One debugging strategy we used was the "*printf* technique". This is a debugging strategy where you would place *printf* statements strategically throughout the code to ensure that that part of the code was reached and successfully ran. This technique was surprisingly useful at the very early stages of our lab. For example, we had them placed in all of our initialization processes to ensure that they were properly getting there and running. We also placed them on each of our button press and switch cases to determine if the correct button and switch were read correctly. However once all that was done, the *printf* debugging strategy started to show its limitation. For example, it could show us when the song was planning but not if the song that was being played was the desirable effect. We had to discard this strategy as we moved to the end of the lab because of this huge limitation.

The next strategy we used was to manually play through the songs. This debugging strategy helped us greatly on the latter half of the lab. There were particular songs within the list of songs provided that were great ways to determine whether or not our certain play-modes were working correctly or not. For example, the song titled *thanku.wav* was a great way to test the delay play-mode. Songs like *WINDOW.wav* was used to easily determine whether or not the reverse play-mode was working or not. Since we knew how the songs should sound in their respective play-mode, it was easily known if our play-mode was working or not.

From a high-level testing perspective, we utilized regression testing to ensure our previous functionalities were working. After any modular changes to the program (global variables, main method changes, LCD update changes, etc…) we would test all the playback modes and buttons again to make sure they're still working.

Audio Playback Performance

Issues and Resolutions

Implementation of the various playback modes was the initial problem; we just didn't know how to do it at first. This required a bit of research into audio CODECs, the structure inside a WAV file, and how the FIFO buffer wanted the data. In addition, for each playback mode we had to research ways to present the audio file in the requested playback mode.

Normal Play

Fairly straightforward method, we didn't run into issues.

Double Play

The requirement for this was to play twice as fast as normal play. Initially, we planned to alternate in skipping samples; {L, L, R, R, L, L, R, R, || L, L, R, R, L, L, R, R, || L, L, R, R …} (skip the red ones). However, that gave us a distorted/weird sound, and we realized the solution is to deal with skipping samples instead of modifying samples themselves. As a result, we implemented counting logic to skip every other sample (32 bits); which provided us with the right sound.

Half Play

From the previous implementation (*DoublePlay*), we learned that we had to solely deal with samples. From research, we also knew that playing a sample repeatedly would ensure that the audio was slowed down. However, an issue we had was at what frequency to play the samples at (every sample, every 2 samples, etc…). Initially we programmed it to replay every 16 bits;{L1, L1, R1, R1, L1, L1, R1, R1, L2, L2, R2, R2, L2, L2, R2, R2} where (Li = 8 bits of Li << 1 | next 8 bits of Li). However the audio wasn't smooth and sounded like it was corrupt. After some debugging, we concluded it wasn't a problem with the way we were playing the audio, but rather a performance issue. Deciding to repeat at a frequency of samples (32 bits) instead of each 16 bit output to a channel was the more optimal idea {L1, R1, L1, R1, L1, R1, L1, R1, L2, R2, L2, R2, L2, R2, L2, R2}. This ended up producing the desired output and played the data at half the original sample rate without any performance issues.

Reverse Play

This was a fairly straight-forward method, and there were no issues that impacted the audio playback performance of this requirement.

Delay Play

This was the most difficult playback mode to implement, and we ran into multiple issues related to the performance of this playback. Through lab sessions and help from TAs we were able to get an idea of how to implement it; which was still wrong, because we thought we had to delay every single half sample that is outputted to a channel. This did not output the desired *DelayPlay* affect, and instead just shifted the playback time by a few seconds for each sample. We then understood we were to delay only the output to the right channel; and once we figured that out, programming it was easy.

Initially we had set the size of the *delayBuffer* to double the sample rate and we stored bytes in there. We noticed that our delay was a bit longer than usual, and may have been a performance issue. This was fixed by pre-processing the sample using *BytePrep* and storing it into the *delayBuffer* as a UINT16. This may have increased the performance slightly as the delay sounded well, and there was no need to call *BytePrep* twice.

Software Efficiency

Embedded System Considerations

A software efficiency issue that we faced apart from those mentioned above with the playback was the pre-processing of packet values for samples. In each of the playback methods, after we populate the *playBuffer* with values using *get_rel_sector(&file_, playBuffer, clusters_, i)*, we intended to create another array of type UINT16 and process the values using *BytePrep* and store them into the array. This would call for an additional for loop to just process the raw sample data, which would lead to add an additional *O(n)* runtime for each sector. However, it would require minimal operations when outputting the value to the FIFO buffer as it is just a simple lookup in the UINT16 array of processed samples. Space is crucial in an embedded system, along with runtime; we came to realize the *BytePrep* method didn't require much operations anyways so optimizing it in this way would be nearly pointless to performance benefit (it wasn't worth the space as well, especially when you run big audio files such as LONG.WAV). As a result, we ended up modifying our code to prepare the bytes for the FIFO buffer dynamically inside the loop; and discarded the idea of pre-processing the byte values into a UINT16 array prior to entering the buffer loop

Testing Efficiency & Playback

We initially had a lot of problems with the delay functionality. We had to go through a lot different debugging techniques in order to be sure that it was correctly implemented. At first we tried to implement a *printf* statement with a counter to indicate the time difference between the first channel and the second channel. However, it quickly got extremely complicated and messy. We ditched the idea and decided it would be easier to just manual test the functionality. Through our peers and TA we found that one of the track (specifically thanku.wav) was extremely helpful in determining if our delay was working out not. This is because whenever there are pauses in the first audio, which is when the second channel should be playing and alternate like that. If there was any overlap of audio then that would easily identify that the delay wasn't functioning as we expected to. Through this method we figured out that our audio was actually overlapping in the undesirable places and was able to pinpoint and fix the error accordingly.

Future Extensions of this Lab

In a previous course (ECE205) the concept of Fourier transforms and its usefulness in our modern day society was presented. We learned that this was the key principle idea in things like auto-tune. One possible extension to this lab is to implement an additional feature that allows pitch correction or alteration to the songs. If the formula or function is given, it would be simple to just apply the changes to the wav files and play it normally. It would just be a cool additional feature that further allows the students to develop a stronger understanding of the whole process without making the original lab that much more complicated and difficult. Additionally, I think by allowing students to add and play with this feature, it fills up some loose ends in what we been learning up to this point.

Contribution

Ramie Raufdeen

I was able to experience a full scope of this lab from design to implementation to testing to report writing. Joey and I both started the lab early and planned how we were to implement the functionalities required for it. I was finally able to understand the basics of Audio Codecs and how various playback modes were implemented. This lab was a lot more challenging and fun than the previous lab due to the fact that there could be multiple implementations to illusion the various playback modes. This was a lot more relatable to the real world because we deal with Audio players and MP3 players on a daily basis, and it's really cool how they work inside. Just like last lab, I also picked up more knowledge of the NIOS IDE, and was able to integrate GIT into the Eclipse IDE and push our lab changes directly to my GitHub account! This lab has gotten me comfortable with C programming and I look forward to expanding those skills in the future.

Joey Keum

Similarity to the first lab, I took charge of the hardware side of the lab. When it came to the software side, Ramie took the lead, and I assisted him in finishing the code. We spit the report in half. The most important thing I learned through this lab is that, we should first fully understand each process and how they worked before going ahead and start coding. By doing so, it could have saved us from spending unnecessary time trying to debug the code.

Notes for TA

We incorporated a bit of the software optimization question into the playback performance section of the report. Apologies, we tried our best to split them apart so that the answers would be more direct, but some software efficiencies/optimization problems overlapped. The same goes with our testing strategies for software efficiencies; as some concepts might have been repeated from the testing of the design.

Source Code Appendix

```
#include "alt_types.h"
#include <stdio.h>
#include "system.h"
#include "sys/alt_irq.h"
#include <io.h>
#include <math.h>

#include "SD_Card.h"
#include "fat.h"
#include "LCD.h"
#include "wm8731.h"
#include "basic_io.h"

//Size of sector buffers and sample rate
#define BUF_SIZE 512
#define SAMPLE_RATE 44100

//Numerical values for switches
#define SW_NORMALPLAY 0
#define SW_DOUBLEPLAY 1
#define SW_HALFPLAY 2
#define SW_DELAYPLAY 3
#define SW_REVERSEPLAY 4

//Numerical values for buttons
#define BTN_STOP 1
#define BTN_PLAY 2
#define BTN_NEXT 4
#define BTN_PREV 8

volatile static int _playing = 0;
volatile static int _isredge = 0;

volatile static data_file _fileinfo;
volatile static alt_u8 _swstate = 0x0;

UINT16 BytePrep(int i, int j){
    //left shift j byte and or it with i, to prepare to write to audio
    return (UINT16)((j << 8) | i);
}

//Play song in regular speed
int NormalPlay(data_file file_, int l_, int* clusters_)
{
    int i, j;
    BYTE playBuffer[BUF_SIZE] = {0};

    for (i = 0; i < l_ * BPB_SecPerClus; i++)
    {
        //If play flag is defaulted (user stops music), then stop playing from buffer
        if (_playing == 0)
            break;

        //Buffer current sector (i) from cluster into the playBuffer
        get_rel_sector(&file_, playBuffer, clusters_, i);
        for (j = 0; j < BUF_SIZE; j+=2){

            //Wait for signal to write for audio, then write to it with prepared bytes
            while(IORD(AUD_FULL_BASE, 0)){}
            IOWR(AUDIO_0_BASE, 0 , BytePrep(playBuffer[j], playBuffer[j + 1]));
        }
    }
}

//Skips every other sample to make it seem like its playing double speed.
int DoublePlay(data_file file_, int l_, int* clusters_)
{
    int i, j = 0;
    BYTE playBuffer[BUF_SIZE] = {0};

    for (i = 0; i < l_ * BPB_SecPerClus; i++)
    {
        if (_playing == 0)
            break;

        //Buffer current sector (i) from cluster into the playBuffer
        get_rel_sector(&file_, playBuffer, clusters_, i);
        for (j = 0; j < BUF_SIZE; j+=2)
        {
            //Wait for signal to write for audio, then write to it with prepared bytes
            while(IORD(AUD_FULL_BASE, 0)){}
            IOWR(AUDIO_0_BASE, 0 , BytePrep(playBuffer[j], playBuffer[j + 1]));

            //Skip 2 lefts and 2 rights every 2 lefts and 2 rights... get it? HAHAHAHAA
            if (j % 4 == 0)
                j += 4;
        }
    }
}

//Duplicate every other sample to make it seem like its half speed
int HalfPlay(data_file file_, int l_, int* clusters_)
{
    int i, j, r = 0;
    BYTE playBuffer[BUF_SIZE] = {0};

    for (i = 0; i < l_ * BPB_SecPerClus; i++)
    {
        if (_playing == 0)
                break;

            //Buffer current sector (i) from cluster into the playBuffer
            get_rel_sector(&file_, playBuffer, clusters_, i);
            for (j = 0; j < BUF_SIZE; j+=2)
            {
                //Wait for signal to write for audio, then write to it with prepared bytes
                while(IORD(AUD_FULL_BASE, 0)){}
                IOWR(AUDIO_0_BASE, 0 , BytePrep(playBuffer[j], playBuffer[j + 1]));

                //Logic to play the bytes twice (repeat every 4 bits)
                if (j % 4 == 0)
                {
                    if (r == 1){
                        r = 0;
                    }else{
                        j -= 4;
                        r = 1;
                    }
                }
            }
        }
    }

    //Plays the audio backwards by iterating from end of buffer to the beginning
    int ReversePlay(data_file file_, int l_, int* clusters_)
    {
        int i, j;
        BYTE playBuffer[BUF_SIZE] = {0};

        for (i = l_ * BPB_SecPerClus; i > 0; i--)
        {
            if (_playing == 0)
                break;

            get_rel_sector(&file_, playBuffer, clusters_, i);

            //Starts from the end and works backwards
            for (j = 508; j > 0; j-=6)
            {
```

```
                    //Wait for signal to write for audio, then write to it with prepared bytes
                    while(IORD(AUD_FULL_BASE, 0)){}
                    IOWR(AUDIO_0_BASE, 0 , BytePrep(playBuffer[j], playBuffer[j + 1]));

                    //To play the next 2 bits of reverse batch (right side)
                    j+=2;
                    while(IORD(AUD_FULL_BASE, 0)){}
                    IOWR(AUDIO_0_BASE, 0 , BytePrep(playBuffer[j], playBuffer[j + 1]));
                }
            }
        }


        int DelayPlay(data_file file_, int l_, int* clusters_)
        {
            int i, j;

            //Create an additional delay buffer with the size of sample rate
            BYTE playBuffer[BUF_SIZE] = {0};
            UINT16 delayBuffer[SAMPLE_RATE] = {0};

            int idxDelay, flag = 0;

            //Iterate through the audio
            for (i = 0; i < l_ * BPB_SecPerClus; i++)
            {
                if (_playing == 0)
                    break;

                get_rel_sector(&file_, playBuffer, clusters_, i);

                for (j = 0; j < BUF_SIZE; j+=2)
                {
                    //Wait for signal to write for audio, then write to it with prepared bytes
                    while(IORD(AUD_FULL_BASE, 0)){}

                    //Populate delayBuffer using idxDelay with what's in playBuffer on an edge
                    if (flag == 0){
                        IOWR(AUDIO_0_BASE, 0 , BytePrep(playBuffer[j], playBuffer[j + 1]));
                        flag = 1;
                    }else{
                        IOWR(AUDIO_0_BASE, 0, delayBuffer[idxDelay]);
                        delayBuffer[idxDelay] = BytePrep(playBuffer[j], playBuffer[j + 1]);
                        flag = 0;
                    }

                    //If we exceed the buffer size, loop around to beginning of array
                    idxDelay++;
                    if (idxDelay > SAMPLE_RATE)
                        idxDelay = idxDelay % SAMPLE_RATE;
                }
            }

            //Finish from current delay index to end of delay array
            for (i = idxDelay; i < SAMPLE_RATE; i++)
            {
                if (_playing == 0)
                    break;

                while(IORD(AUD_FULL_BASE, 0)){}
                IOWR(AUDIO_0_BASE, 0, delayBuffer[i]);
            }

            //Finish last remaining audio data from beginning to delayIdx
            for (i = 0; i < idxDelay; i++)
            {
                if (_playing == 0)
                    break;

                while(IORD(AUD_FULL_BASE, 0)){}
                IOWR(AUDIO_0_BASE, 0, delayBuffer[i]);
            }
        }

        //Get the current switch value and display the paly mode & song to LCD
        static void DisplayStatusLCD()
        {
            _swstate = _swstate & 0x07;
            LCD_Display(_fileinfo.Name, _swstate);
        }

        //This is used to set up the buttons to do their required roles.
        static void button_ISR(void* context, alt_u32 id)
        {
            //Listen to ISR on a specific edge, so that we don't receive 2 interrupts at once
            if(_isredge == 0){
                alt_u8 btnPressed = IORD(BUTTON_PIO_BASE, 3) & 0xf;

                switch(btnPressed){
                    //Set playing flag to default value and update the status to LCD
                    case BTN_STOP:
                        _playing = 0;
                        IOWR(BUTTON_PIO_BASE, 2, 0xf);
                        DisplayStatusLCD();
                    break;

                    //Set play flag to high and let while loop handle audio play (disable other buttons)
                    case BTN_PLAY:
                        _playing = 1;
                        IOWR(BUTTON_PIO_BASE, 2, 0xf);
                    break;
                    case BTN_NEXT:
                        if(search_for_filetype("WAV", &_fileinfo, 0, 1) == 0)
                            DisplayStatusLCD();
                    break;
                    case BTN_PREV:
                        //Prevent file iterator from getting stuck on the first file when going previous
                        if(file_number < 0)
                            file_number = 0;
                        else
                            file_number = file_number - 2;

                        if(search_for_filetype("WAV", &_fileinfo, 0, 1) == 0)
                            DisplayStatusLCD();
                    break;
                }
                _isredge = 1;
            }else{
                _isredge = 0;
            }

            IOWR(BUTTON_PIO_BASE, 3, 0x0);
        }

        int main()
        {
            alt_u8 _prevswstate = 0x0;

            Setup();

            //Set up the initial file on SD card
            search_for_filetype("WAV", &_fileinfo, 0, 1);
            _swstate = IORD(SWITCH_PIO_BASE, 0);
            _prevswstate = 8;

            while(1)
            {
                //Listen for any changes in the switches, update if so
                _swstate = IORD(SWITCH_PIO_BASE, 0);
                if(_swstate != _prevswstate){
                    DisplayStatusLCD();
                    _prevswstate = _swstate & 0x07;
                }
```

```
283.
284.                if (_playing == 1)
285.                {
286.                    DisplayStatusLCD();
287.                    printf("Playing audio file: %s\n", _fileinfo.Name);
288.
289.                    //Disable other buttons
290.                    IOWR(BUTTON_PIO_BASE, 2, 0x01);
291.
292.                    //Initialise cluster chain and build it from the audio file
293.                    int clusterChain[100000];
294.                    int tracklength = 1 + ceil(_fileinfo.FileSize/(BPB_BytsPerSec*BPB_SecPerClus));
295.
296.                    LCD_File_Buffering(_fileinfo.Name);
297.                    build_cluster_chain(clusterChain, tracklength, &_fileinfo);
298.
299.                    //Update LCD and play audio dependent on user-selected mode
300.                    DisplayStatusLCD();
301.                    switch(_swstate){
302.                        case SW_NORMALPLAY:
303.                            NormalPlay(_fileinfo, tracklength, clusterChain);
304.                            break;
305.                        case SW_DOUBLEPLAY:
306.                            DoublePlay(_fileinfo, tracklength, clusterChain);
307.                            break;
308.                        case SW_HALFPLAY:
309.                            HalfPlay(_fileinfo, tracklength, clusterChain);
310.                            break;
311.                        case SW_DELAYPLAY:
312.                            DelayPlay(_fileinfo, tracklength, clusterChain);
313.                            break;
314.                        case SW_REVERSEPLAY:
315.                            ReversePlay(_fileinfo, tracklength, clusterChain);
316.                            break;
317.                        default:
318.                            NormalPlay(_fileinfo, tracklength, clusterChain);
319.                            break;
320.                    }
321.
322.                    //Enable buttons, and reset play flag
323.                    IOWR(BUTTON_PIO_BASE, 2, 0xf);
324.                    _playing = 0;
325.                }
326.            }
327.
328.            return 0;
329.        }
330.
331.        //Setups up required aspects for the program to run.
332.        void Setup(){
333.            SD_card_init();
334.            init_mbr();
335.            init_bs();
336.            init_audio_codec();
337.
338.            IOWR(BUTTON_PIO_BASE, 2, 0xf);
339.            IOWR(BUTTON_PIO_BASE, 3, 0x0);
340.            alt_irq_register(BUTTON_PIO_IRQ, (void*)0, button_ISR);
341.        }
```