UNIVERSITY OF
WATERLOO

Faculty of Engineering

DESIGN OF AN MP3 SYSTEM

Lab 2 - Interfacing with SD card memory and FAT file system
MTE 325 Microprocessor Systems and Interfacing

Prepared by
Kevin Premrl & Pavel Shering
ID # 20517153 || 20523043

3A Mechatronics Engineering

June 21, 2016

1 System Design Overview

The flow chart (Figure 1 on page 2) illustrates the complete system design. The program starts off with initialization of SD card, the following with the master boot record, boot sector and LCD initializations. Audio codec is set up, then the button interrupts are set up, at which point the system enters a state machine based on the button state variable.

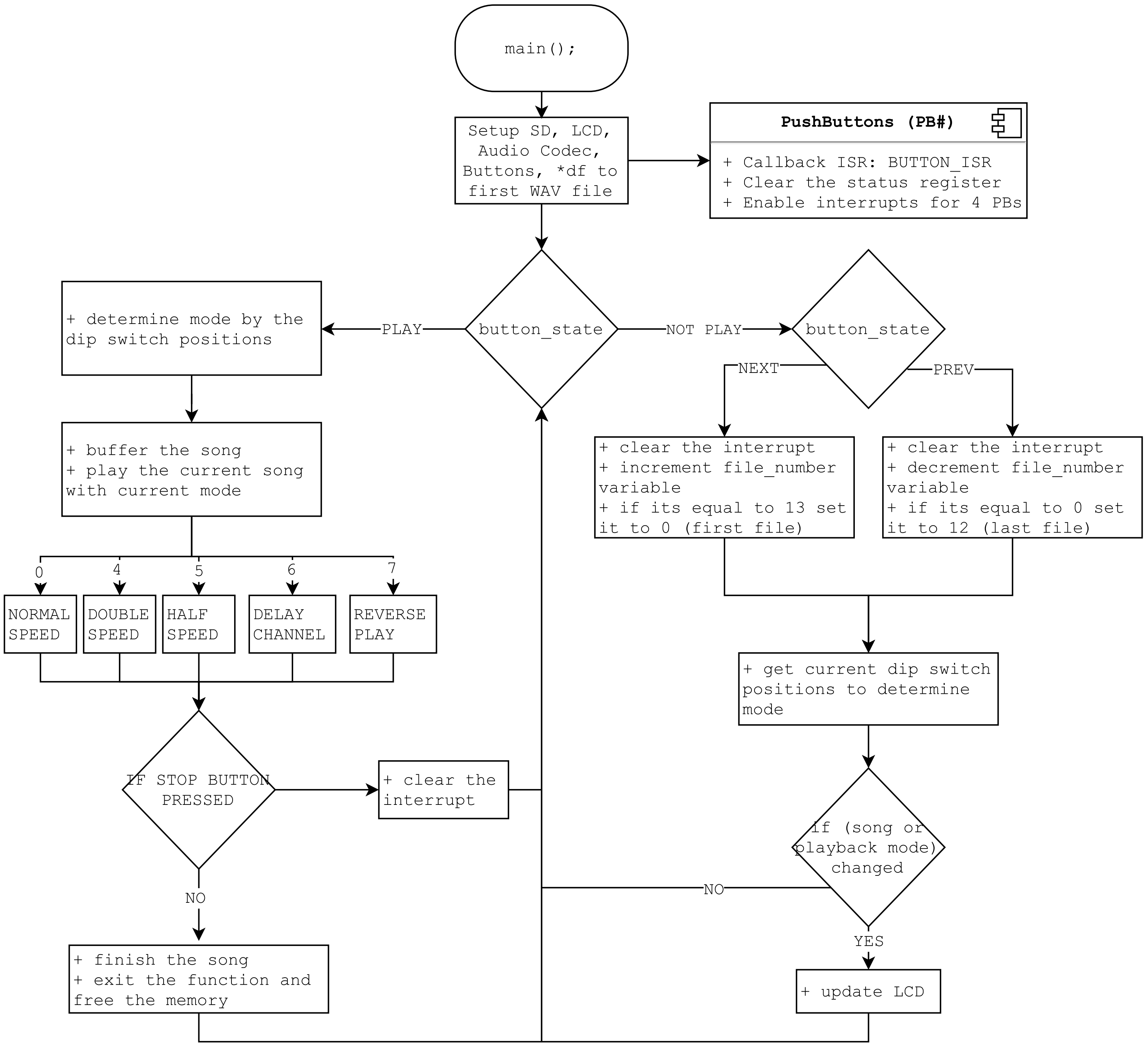
main();
Setup SD, LCD, Audio Codec, Buttons, *df to first WAV file
PushButtons (PB#)
+ Callback ISR: BUTTON_ISR
+ Clear the status register
+ Enable interrupts for 4 PBs
button_state
PLAY
NOT PLAY
+ determine mode by the dip switch positions
+ buffer the song
+ play the current song with current mode
0
4
5
6
7
NORMAL SPEED
DOUBLE SPEED
HALF SPEED
DELAY CHANNEL
REVERSE PLAY
IF STOP BUTTON PRESSED
+ clear the interrupt
NO
+ finish the song
+ exit the function and free the memory
button_state
NEXT
PREV
+ clear the interrupt
+ increment file_number variable
+ if its equal to 13 set it to 0 (first file)
+ clear the interrupt
+ decrement file_number variable
+ if its equal to 0 set it to 12 (last file)
+ get current dip switch positions to determine mode
if (song or playback mode) changed
NO
YES
+ update LCD

Figure 1: Phase I software design

2 Hardware Design and Software Interfacing Issues

The following section describes the issues encountered in hardware design and software interfacing implementation for the MP3 system.

2.1 Hardware Design & Interfacing

One of the issues associated with the hardware that is given is that the LCD display seemed faded and incomplete. This issue is caused because the program is designed to constantly reprint the state of the music player while a song is not playing. Thus, if the song or playback mode is changed the LCD screen will reflect the change immediately. The LCD is unable to handle the constant reprinting of the screen.

2.2 Design Decisions

A software decision made to combat the issue of the LCD screen not displaying properly is to only refresh the screen when a change of the state of the music player happened. To do this, rather than reprinting the screen constantly, it simply checks to see if the current state is different from the previous state, and it reprints the screen if that was the case.

An alternative to that approach is to poll the current state of the music player periodically, and update on set intervals. This approach is discarded because it would have resulted in a delay in updating the LCD and is more difficult to implement.

2.3 Hardware & Software Interfacing Testing and Debugging Strategy

The testing and debugging strategy used for the button interfacing design is a rigorous and thorough one. The program is rebuilt and tested with each addition to the code to catch any bugs as they arose, rather than doing several parts at once and having to isolate bugs later. Intermediate stages of the design involved printing out information to the console, such as which song the hardware is going to play next. The printing stage is discarded when printing to the LCD display is functioning properly.

An alternative debugging strategy that is ultimately not used is to simply code the entire button interfacing section and isolate any bugs that arose upon completion of the initial code. The advantages to this method would be that time would not be wasted building and testing the code after each step. The disadvantages are that any bug is much harder to find and more time could potentially be wasted finding bugs than what is saved by not testing throughout the design process.

3 Audio Playback Software Issues

The following section describes the issues encountered in audio playback implementation for the MP3 system.

3.1 Audio Playback Issues

3.1.1 Normal Speed

There is no major issues with the normal playback mode, besides the implementation of the stop button. By the requirement, the stop button is to stop playing the music and if play is pressed on the same song, it will restart the song and not continue from the stopped point. Therefore, by our design the stop button is to exit the *play_song* function when pressed, which will allow for song restart. The exit must also happen with minimal delay such that the experience of the system is not degraded from the user's perspective.

The exit is implemented simply with a return statement when the interrupt changes the button state to "STOP". The returns are located inside the sector for loop as well as the sample for loop to minimize the response of the button press. In addition, by design the state machine is implemented in a way that normal speed playback is the default mode if a combination of switches is not recognized as one of the required playbacks.

3.1.2 Double Speed

The main issue with the double speed playback is retain the stereo functionality. The first approach taken to implement double speed is skipping every second sample. This however cannot be handled by the audio codec and the hardware, and the audio file is no longer played in distinct stereo, where the right and left individual channels creates interference and results in playing out of both channels with noise. The reason for outcome is because the signal is chopped at byte level which results in little parts of a sector combined into a signal wave. The second approach is to skip sectors which allows for smoother signal output as the chopping is at sector level instead of byte level. This results in stereo output signal and approximately the same signal duration as the first approach to achieve double the speed. For the final project demo, the first approach was used as the second one was forbidden by the lab requirement.

3.1.3 Half Speed

There was no particular issues with this playback, however out of curiosity two different approaches are attempted to implement the half speed mode. The first approach is to play the same sample twice consecutively while incrementing through the sector. The second approach is to interpolate between two samples to create another sample and attempt to smooth out the output signal (see Listing 1 in A). This requires playing the first sample and storing it, then averaging the next sample with the stored first and playing it, then storing the second. This however sounded distorted and noisy. Thus the final demo included the first approach as the half speed implementation.

3.1.4 Delay Channel

The delay of channel playback is the most difficult one to implement, and only one issue occurred on the audio playback which was static in the first second of signal output. The design for this playback was carefully thought through. The implementation in the current system, delays the right channel playback by a full second. This is done by allocating a buffer that is exactly a second long by taking into account the sample rate of WAV files. The first channel (left) is played for the first second which is measured by a counter while the right channel is stored and an empty right buffer is played at the counter location. When the counter reaches the full second, the right empty buffer is now filled with signal data from a second before. The counter is then reset and thus acts like a circular buffer. Lastly, the last second of the right buffer is played when the file ends and empty samples are sent for the left channel. The problem with the static was the fact that *malloc* does not initialize the allocated memory to zero, and *calloc* function was not available, thus *memset* to zero was used right after buffer memory allocation.

3.1.5 Reverse Play

Minor issue was encountered for the reverse play mode was the mistake of starting at the last channel sample instead of second last channel which will keep the left and right in order. That was a simple code refactoring mistake that was caught by reading over the code.

3.2 Efficient Software Issues

There are a couple of issues with writing efficient code for this system. First, the LCD display code is not to be edited thereby making twice the amount of playback mode defines. Second, it is hard to reuse code for multiple playback modes which leads to a decision of whether to split each playback mode into a function or have all the playback modes in one function. This also effects the implementation the stop button interrupt. Lastly, the *data_file file_number* variable is changed by other functions in the background which created unexpected behaviour.

3.3 Software Design Decisions

The solution to the LDC display issues was to create defines for both the buttons modes from the dip switches as well as the LDC display defines that are used in the non play state machine (that updates the song titles and playback modes on the LCD). This allows for much more readable code, as well as clean switch case implementation where every dip switch mode is explicitly defined.

To reuse as much of the same code as possible all the playback modes are dealt with in the same function *play_song*. This allows for less lines of code, while maintaining documented source code using the if statements with mode defines mentioned above. This approach is simple to implement and efficient since there is minimal repeated code. Further more the main only allocated *data_file* pointer **df* once and passes it into the *play_song* function. The cluster chain pointer and the buffer are allocated inside the function and free upon stop press or song finish to have memory available for possible future features. This implementation then makes the stop press to be a simple return statement if the STOP flag is set by the button interrupt.

Alternatively, the calculation of the length into the main function, and if the system design is final, the allocation of space for the buffer can be moved as well and passed into the function as parameters. This saves the amount of operations the processor has to complete and avoids mallocing and freeing operations every time a new song is played which slows the system down. However, that means there is less available memory on the stack to be used for other memory required operations. This is discarded to have more available stack memory for the system, even thought testing this is the same difficulty as the first approach.

Each of the playback mode issue solutions are mentioned in section 3.1 and the approach to the solution is described in section 3.4.

3.4 Testing and Debugging of Audio Playback Software

Although all the playback modes are done through one function, *play_song*, the development and integration of each individual mode was done in small increments to make sure all parts of the system work. Initially the *play_song* function only implemented the normal mode which is the most basic mode but is the building block for all others. The next mode is then build upon that, and implemented as an isolated if statement at first. As soon as its working according to specification, the code is refactored for efficiency. The defines make it easy to isolate playback modes when the function is called with the mode as a parameter. This strategy also involved visual confirmation by using the profiling technique with print statements of mode passed in and the mode chosen in the *play_song* function. Profiling involves sending feedback about variable values and system state (mode) to the console and visually verifying that it correct state is entered and its the expected behaviour. Profiling allowed for the developers to catch simple boolean operation bugs, if the state chosen on the dip switches did not match the state chosen in the *play_song* function.

An alternative strategy is complete isolation and test driven development. This technique is much more in depth in terms of time taken to develop, however it allows for strong confirmation of the system correct operation. The essence is to create test cases that will validate the signal output or sector/file traversal of the output signal but feeding the output into a buffer and running identity checks for expected output. This is done for each mode individually, however, can further be expanded to full system testing in terms of state transitions which then use the smaller test cases that test the expected output. The test cases are then run and failed. Based on the test output and test cases that act as documentation the developer then writes the implementation of the system in small parts consistently updates the test cases list and running the system against it. The cycle is repeated until all features are implemented and the test cases all pass.

The first approach is taken in this project as it takes less time and the system is not at large enough scale to justify the use of testing frameworks in conjunction with the signal output for validation. The project is small enough to test by profiling technique which is faster in small system development, however will be harder is the system is scaled.

4 Future Extension

Given enough time, Kevin would like to make a music player that allows a user to play music along with it. A song is to be loaded from the SD card to be played from the speakers. The buttons that are unused while a song is playing (next, previous, start) are then used to play notes. The switches can be used to change the modes partway through, instead of only being able to change it in between songs. In the end, a user will have a music player that allows to create a remix and play along with it.

Pavel likes this project as it involves signal processing, however given enough time he would like to try image processing and the possibility to learn about machine learning and artificial intelligence. The same hardware can still be used to load the images from the SD card and determine specific features found on the images, or perhaps using edge detection identifying objects on the image. The reason for the project would be introduction to image processing and AI algorithms, which can serve as initial building blocks for upper year courses. The current project should still be part of the curriculum as it builds on from SYDE 252 signal processing course taken in 2B.

5 Contribution and Reflection

Kevin's primary contribution to the lab is getting the button functionality to work. This involves determining how to interact with the SD card to change and stop songs via button interrupts. Another large contribution was the LCD functionality. This includes getting the proper song and mode to be displayed, and for it to update immediately when there is a change of state.

The most important learning outcome from Kevin's lab experience was learning how to handle interrupts in a highly efficient manner, and learning how to interact with external memory.

Pavel's primary contribution to the lab is implementation of the playback modes. This involves meeting the specifications of each mode (described in the lab manual) and retrieving the expected output.

The most important learning outcome from Pavel's lab experience was signal processing and interfacing with the FAT file system on the SD card.

Appendix A Interpolation Implementation of Half Speed Playback Mode

```c
for(i = 0; i < BPB_SecPerClus * length; i++) {
    if (button_state == STOP) return;

    get_rel_sector(df, buffer, cc, i);
    for(j = 0; j < BPB_BytsPerSec; j += speed) {
        if (button_state == STOP) return;

        while(IORD( AUD_FULL_BASE, 0 )) {} //wait until the FIFO is not full
        tmpL = ( buffer[ j + 1 ] << 8 ) | ( buffer[ j ] ); //Package 2 8-bit bytes from the sector buffer array into the
        IOWR(AUDIO_0_BASE, 0, tmpL); //Write the 16-bit variable tmp to the FIFO where it will be processed by the audio CODEC

        while(IORD( AUD_FULL_BASE, 0 )) {}
        tmpR = ( buffer[ j + 3 ] << 8 ) | ( buffer[ j + 2 ] );
        IOWR(AUDIO_0_BASE, 0, tmpR);

        if(MODE == HALF_SPEED) {
            UINT16 tmp2L, tmp2R;
            UINT16 tmp_half_speed_L16, tmp_half_speed_R16;
            if( j + 7 < BPB_BytsPerSec) {
                tmp2L = ( buffer[ j + 5 ] << 8 ) | ( buffer[ j + 4 ] );
                tmp2R = ( buffer[ j + 7 ] << 8 ) | ( buffer[ j + 6 ] );

                UINT32 tmp_half_speed_L = tmpL;
                tmp_half_speed_L += tmp2L;
                tmp_half_speed_L >>= 1;
                tmp_half_speed_L16 = tmp_half_speed_L;
                UINT32 tmp_half_speed_R = tmpR;
                tmp_half_speed_R += tmp2R;
                tmp_half_speed_R >>= 1;
                tmp_half_speed_R16 = tmp_half_speed_R;
            } else {
                    tmp_half_speed_L16 = ( buffer[ j + 1 ] << 8
                        ) | ( buffer[ j ] );
                    tmp_half_speed_R16 = ( buffer[ j + 3 ] << 8
                        ) | ( buffer[ j + 2 ] );
                }
                while(IORD( AUD_FULL_BASE , 0 )) {};
                IOWR(AUDIO_0_BASE , 0, tmp_half_speed_L16);
                while(IORD( AUD_FULL_BASE , 0 )) {};
                IOWR(AUDIO_0_BASE , 0, tmp_half_speed_R16);
            }
        }
}
```

Listing 1: Interpolation for half speed playback