ECEN 449: Microprocessor System Design
Department of Electrical and Computer Engineering
Texas A&M University

Prof. Sunil P. Khatri
TA: Ayan Mandal
(Lab exercise created by A Targhetta and P Gratz)

Laboratory Exercise #10

Audio Player Application with IR-remote and AC '97 Codec

Objective

The objective of lab this week is to provide experience developing an application which utilizes the custom device drivers written in previous labs for the IR-remote and AC '97 audio device. You will create an application, which plays wav files stored on the compact flash, controlled by IR-remote input. Before developing your wav player, you will extend your audio device driver to have write capabilities from user space.

System Overview

The system hardware for this lab is depicted in Figure 1. This is the same hardware system created in Lab 8 and updated in Lab 9. As a review, some of the notable features in this hardware system include a Linux compatible MicroBlaze processor with cache, 256 MB of DDR2, Compact Flash, a Uart, an Audio controller, and an IR peripheral. The software system for this lab is shown in Figure 2, the components in gray are those which you will write or modify in this lab. The wav player (i.e. your user application) interacts directly with the Linux kernel via file system calls, while the file system interacts with the hardware through the device drivers installed in the kernel. As a whole, your user application will receive input from the IR-remote via the device driver you created in Lab 8 and the demodulation hardware you developed in

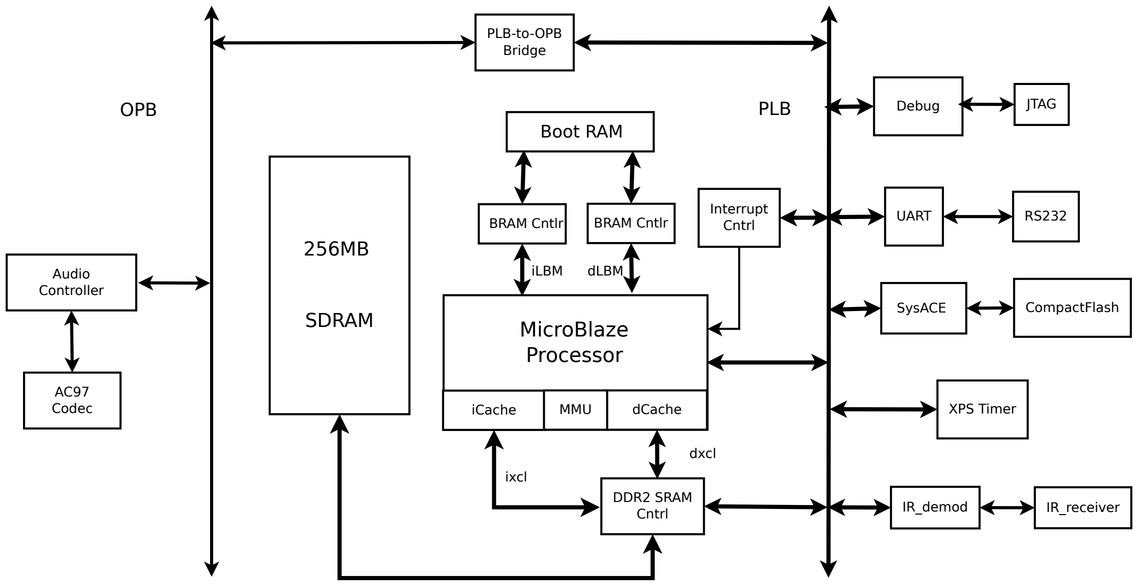

Figure 1: Hardware/Software System Diagram

Lab 7. Based on the IR remote input, your user application will transfer audio data from wav files on the compact flash to the audio codec via the audio device driver you began development on in Lab 9.

One important difference between the hardware you will use in this lab and the hardware you built last lab is not visible in the block diagram above but has to do with the trigger sensitivity of the audio controller interrupt. The trigger sensitivity is "positive edge" triggered in this lab, whereas in Lab 9, it was "high level" triggered. In interrupt based systems, an interrupt is said to be level triggered when the processor is interrupted based on the level of the interrupt signal coming from hardware. For example, when an interrupt is "high level" triggered, it means the processor will continue to service the interrupt as long as the level of the interrupt signal remains high. In addition to being level sensitive, interrupts can be edge triggered. For example, if an interrupt is said to be "positive edge" triggered, it means the processor will respond to the interrupt only when it receivers a rising edge of the interrupt signal. Similarly, "negative edge" triggered interrupts exist. You already have experience working with edge triggered interrupts since the IR-remote hardware built in Labs 7 and 8 has a "positive edge" triggered interrupt.

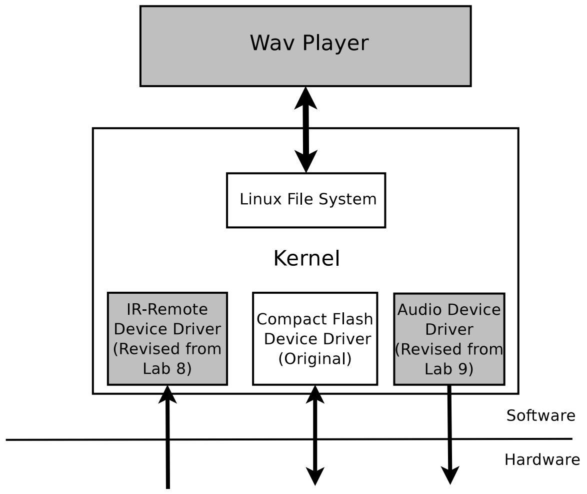
Wav Player
Linux File System
Kernel
IR-Remote Device Driver (Revised from Lab 8)
Compact Flash Device Driver (Original)
Audio Device Driver (Revised from Lab 9)
Software
Hardware

Figure 2: Wav Player Block Diagram

In Lab 9, the playback FIFOs were filled in the interrupt handler. Thus, a “high level” interrupt worked quite well. Once the playback FIFO became half empty, the interrupt signal went high and the processor stopped what it was doing and called the appropriate handler. Once in the handler, interrupts were disabled, and the MicroBlaze filled the playback FIFO. The hardware responded by lowering the interrupt signal when the FIFO was greater than half full. When the MicroBlaze exited the interrupt handler, interrupts were once again enabled, but the playback interrupt signal was already low, which means the MicroBlaze was not interrupted until the playback FIFO was half empty again. This must change, however, when the filling of the FIFO is handled outside of the interrupt handler. In this lab, the interrupt handler is responsible for merely waking processes in the wait queue in the *write* method. Therefore, if the playback FIFO is level triggered, the playback interrupt does not get cleared while the interrupt handler is executing, and the MicroBlaze will continue to re-enter the playback interrupt handler, never yielding for other processes including the one that called the *write* method.

To address the issue of recurrent interrupts you will make the playback interrupt “positive edge” triggered. In this case, the interrupt handler will only execute once when the playback FIFO level dips below the half way mark. At which time, the process that called the *write* method of your audio device driver will be woken up and scheduled to run at the processor’s latest convenience. The handler will not be called again until the next rising edge of the interrupt allowing time for the process to fill the buffer and lower the interrupt line.

Procedure

1. Before development of the wav player application, you must allow user applications to write data to the audio codec. Remember in Lab 9 you wrote a device driver that continually transferred data from a buffer in kernel space to an audio buffer in hardware within the interrupt handler. What you need now is for data to be transferred from a user space buffer to the hardware. One issue with Lab 9's implementation is that you were performing a significant amount of data movement within the interrupt handler while interrupts were disabled. Additionally, in this lab, you must transfer data from user space to kernel space. This should not be done in the interrupt handler. The next few steps will guide you through the process of setting up the data transfer in the *write* routine while still utilizing the playback interrupt.

   (a) The playback interrupt in last lab was level sensitive because the data transfer was handled in the interrupt handler. However, in this lab, you are required to use a "rising edge" triggered interrupt since data will be transferred in the *write* method. Re-import the audio controller peripheral setting both the playback and record interrupts as rising-edge triggered. Have the TA inspect your system before continuing.

   (b) Start the hardware re-build process while you modify your device driver as outlined below.

   (c) Take a moment to review the "irq_test.h" and "irq_test.c" source files from module_examples directory in the laboratory directory. Notice how this example device driver sleeps within the read function using 'wait_event_interruptible' and wakes up from the read function using 'wake_up_interruptible' from within the interrupt handler.

   (d) Modify your audio device driver from Lab 9 such that data is transferred from user space to kernel space within the *write* method call using the following guidelines:

      - Your *write* method should block until all data is transfer from the user buffer to the hardware buffer. The user shall be allowed to request that an arbitrary amount of data is transferred. Thus, larger requests will cause *write* to block for longer.
      - From within the *write* method, transfer data one audio sample at a time into the hardware FIFO using the 'get_user' and 'iowrite' system calls. This code should be similar to the code within your interrupt handler from last lab.
      - Use the 'wait_event_interruptible' call to put *write* to sleep when the audio FIFO is full. You may use the 'isInFIFOFull()' macro defined in "xac97.h" to determine when the playback FIFO is full.
      - Your interrupt handler should only call 'wake_up_interruptible' to wake up the *write* method so that it can continue to transfer data.

   (e) Create a devtest application which calls *write* to transfer data from the predefined buffer found in "audio_samples.h" to the audio codec. Note: you may have to modify the declaration of audio samples to use **unsigned short** instead of the **u16** data type.

(f) Compile your modified source code on the CentOS workstation and install the resultant kernel module into the XUP Linux system. Execute your devtest and ensure it plays audio correctly. Demonstrate your progress to the TA.

2. You will now enable support for audio files which store either one channel (mono) or two channels (stereo). Because, the playback FIFO interleaves left and right channel data, in the Lab 9 you had to perform two writes to the playback FIFO for every sample stored in the 'audio_samples' data array. You will incorporate support for either mono or stereo in your device driver following the next few steps.

   (a) Define a *static* global variable in the header file of your audio device driver called *mono*. In the *open* method initialize that variable to 0.

   (b) Now within your *ioctl* method, add another command to the switch statement called 'ENABLE_DISABLE_MONO' such that when *val* is true, *mono* is set to true. Otherwise, *mono* should be set to false. The predefine for this new command can be found in the "xac97.h" and "sound.h" header files.

   (c) With the ability to set *mono* to true or false from a user application, you can now add code within the *write* method which utilizes *mono*. Modify the *write* method such that when *mono* is set to true, duplicate samples are written to the playback FIFO.

   (d) Change your devtest to test the functionality you just added to your device driver. When *mono* is set to false with the audio samples provided in lab, the playback rate will seem to be twice that of when *mono* is set to true. Demonstrate this effect to the TA.

3. The next step is to create a test application that is able to read wav files directly from the compact flash card. Thus, your application must open an audio file, parse the header within the wav file to determine the format of the file, initialize the audio codec accordingly, and finally, copy data from the wav file to the audio device driver file. The following steps will guide you through the development process.

   (a) Copy the "audio_samples.wav" file from '/homes/faculty/shared/ECEN449' onto the compact flash card.

   (b) Start with a new source file called "wav_play.c". Add code which properly opens and closes the your audio device file at the beginning and end of the program respectively. Add code to open and close the wav file as well. Be sure to include the proper error handling as this may reduce the amount of debugging necessary.

   (c) Parsing the wav file requires an understanding of the wav file format. The "parse_wav.h" and "parse_wav.c" files from '/homes/faculty/shared/ECEN449' provide you with code to perform the parsing. Read through these files to understand the wav file format and the interface to the parsing code so that you may integrate the parsing functions into your new "wav_play.c" source file in the next step.

Note that the provided code is written to support little endian processors as well as big endian. The microblaze is big endian; however, the wav file is formatted using little endian (this file format was originally used on X86, a little endian implementation). Thus wav files require the reversal of the byte order performed by the 'reverse_endian()' function found in "parse_wav.c" when used on the microblaze processor.

(d) Use 'mmap' to map the contents of the wav file into a user space buffer. Use the provided parsing function to parse the wav file's header. Be sure to declare a 'wav_properties_t' structure properly prior to calling the 'parse_wav()' function and handle erroneous returns from 'parse_wav'. Examine the "parse_wav.c" file to understand what would cause it to return a negative value.

(e) For debugging purposes, add code that prints the wav properties defined in the 'wav_properties_t' structure to the terminal. Load your application and the provided wav file onto the XUP board and demonstrate your progress to the TA.

(f) After the file parse code is complete, add code which ensures that 'format' field within the 'wav_properties_t' structure is equal to '1'. Any other value signifies that the file contains compressed audio data of which you need not support. Be sure to print an error message accordingly.

(g) For this lab, you will only support 8-bit audio samples. Therefore, you must add code which ensures the 'bits_per_sample' is equal to 8. Your application should error out if this particular field equals anything else.

(h) Your next goal is to support various sampling rates. Add code which utilizes the *ioctl* functionality of your audio device driver to adjust the playback rate according to the playback rate defined within the 'wav_properties_t' structure.

(i) Within the provided wav file structure, there is a void pointer labeled 'audio_samples', which points to the beginning of the data section within the memory mapped region of the wav file. Use this pointer along with the other wav file properties to play the audio in the "audio_samples.wav" file. The following hints should get you started:

- Use only the provided wav file for now. Larger wav files require additional care and will be discussed later in lab.
- For initial testing, you may write the entire contents of the wav file data section at once (a file-at-once mode). The 'wav_properties_t' structure populated by the 'parse_wav' method contains a 'num_bytes' and a 'bits_per_sample' field. You should use these appropriately.
- Once file-at-once mode works properly, modify your program to write audio samples to the audio device driver in blocks. Create a define within your header file that allows you to set the size of data block at compile time. Note that the playback FIFO has space for 4k samples when the playback interrupt triggers.
- The last 'block' will be a partial block. Be sure not to overstep the bounds of your mmapped region with *write*.

(j) To prepare for the next section, ensure your application plays the provided wav file in repeat mode. This allows us to do testing with a small wav file.

(k) Demonstrate your current progress to the TA.

4. With wav playback capabilities in place, you will now add use of the IR-remote to control the wav player application. Specifically, you will add use of the 'Volume-Up' and 'Volume-Down' buttons to adjust the AuxOut volume and the 'Play', 'Pause', and 'Stop' buttons to control playback. This means you must periodically call *read* on your IR-remote device driver in order to receive input from the user. Currently, your IR-remote device driver maintains a queue of the most recent unread messages. This allows your user application to do other things without missing incoming messages. The audio device driver, however, blocks the user application until the number of samples specified by the *length* parameter have been written to the playback FIFO.

   (a) Modify your application such that it requests the entire message queue from your IR-remote device driver in between calls to your audio driver *write* routine. This means your application must allocate a 200 byte buffer in user space and call the *read* routine of your IR-remote device driver with the *length* parameter set to 200. Your software must keep track of the number of bytes that were actually read.

   (b) After reading in the IR-remote messages, process each message, updating your system accordingly. The following suggestions should help you get started:

   - Use your notes from Lab 7 and 8 for the mapping of the buttons on the IR-remote.
   - For volume control, review the "xac97.h". The AC '97 allows for independent left and right channel adjustments, where the first byte of the volume control register adjusts the right channel and the second byte adjusts the left channel. For your simple application, adjust the two channels together. Note that the 0x1f1f hex code is used for minimum volume for both channels, while 0x001f is for maximum volume on the left channel and minimum on the right. 0x0000 sets both channels to maximum.
   - If the left and right channels are adjusted together, there are 32 discrete volume levels available. One simple solution for volume adjustment is to use a *static* variable to keep track of the current volume level and increment or decrement the variable accordingly for every volume message you receive. Also update the codec with *ioctl* and your volume variable when the volume changes. If you find the volume changes too rapidly, you may employ counters to reduce the amount of effect each message has on changing the volume.
   - The functionality for 'pause' and 'stop' are similar in that each should inhibit your application from calling *write* and the user must press the 'play' button to start playing again. Pause, however, must keep track of the last sample written such that your application can start where it left off when 'play' is pressed, while 'stop' will reset to the beginning of the wav file. One solution is to maintain a 'play_audio' variable which is set to '1' when 'play' is pressed and set to '0' when 'stop' or 'pause' are pressed. You may then use an if-statement

that checks the value of 'play_audio' prior to writing samples to the audio codec. To avoid continually calling read, when not writing samples to codec, your application should sleep for approximately the same time it would have spent in the *write* routine. You may use the 'usleep' system call for this.

(c) Once complete, demonstrate your progress to the TA.

5. The final capability you will add to your player application is the ability to select various wav files on the compact flash to play using the 'Channel-up' and 'Channel-down' buttons on the remote. Doing so means that your application must read the contents of the current directory to determine which wav files exist. For this lab, you are provided with skeleton code which uses 'scandir' to read the directory, filter the contents for '.wav' files, and sort the remaining filenames in alphabetical order. The end result is an array of filenames which you can index according the button events from the remote. The call to 'scandir' returns the number of filename entries in the array, and our application is able to use that to read the files in a circular fashion. It is still up to our application to ensure the '.wav' files provided by 'scandir' are valid, but our application can make use of the 'parse_wav' function used earlier in lab. The next few steps will guide you through the process of adding the finishing touches to your wav player application.

   (a) Start by copying the 'wav_play_skeleton.c' file from the '/homes/faculty/shared/ECEN449/' directory. Rename this file to another name of your choosing.

   (b) Look over the code you just copied and work through what is going on. Note that the skeleton code provides some additional functionality not requested in the lab. This code is there for example sake so you should use it to get an idea of how to implement what is requested in the lab.

   (c) Merge your code from the previous sections with the skeleton code provided.

   (d) Use the following guidelines to complete your application:

   - Initially, your applications should open the first filename entry pointed to by 'namelist' (index 0). Opening a wav file involves mmapping its contents, parsing the header of the file, and initializing the pointers used to access the data.
   - Each subsequent time a file is opened, your application must properly munmap the contents of the previously opened file and close that file prior to opening another file.
   - When 'channel-up' is pressed, you should open the next file. Do this in a circular fashion such that when you reach the end of the filename list, you start back at the beginning.
   - Likewise, when 'channel-down' is pressed, you should open the previous file. Do this in a circular fashion such that when you reach the beginning of the filename list, you wrap back around to the end of the list.
   - For 'channel-up' and 'channel-down' it is important that your application respond only once to a single button event. Remember that the IR-remote sends multiple copies of the same

message. One way to do this is to respond to the first message of a given type and ignore repeats until a certain amount of time has passed or another message type is received. The 'mute' functionality in the code provided does this by maintaining a 'wait_count' variable. This variable is incremented between calls to *write* when greater than 0 and reset to 0 when past a particular threshold. Because the *write* method blocks for a deterministic amount of time, this has the same effect as using a timer to wait.

(e) Once your code is complete, load the provided wav files from the '/homes/facult/shared/E-CEN449/' directory onto the compact flash. Install the audio and IR-remote device drivers and test your final application. Note that the additional wav files provided have been recorded using various playback rates and some even include stereo audio. Your application should be able to play all of them correctly.

(f) Demonstrate your application's functionality to the TA.

Deliverables

1. [7 points.] Demo the working wav player to the TA.

   Submit a lab report with the following items:

2. [5 points.] Correct format including an Introduction, Procedure, Results, and Conclusion.
3. [4 points.] Commented C files.
4. [4 points.] Answers to the following questions:

   (a) In your wav player application, we suggested the use of mmap for reading the contents of the wav file. What other ways are there to accomplished the same thing? Compare and contrast these methods. Some of the things to consider include ease of use, code readability, efficiency, and blocking time. Hint: You can find more information on mmap in the O'Reilly's *Linux System Programming* book available for free on-line.

   (b) What is the purpose of the call to 'madvise'? What might be the effect of removing this call? You can test this with the 'audio_samples7.wav' which is a full length song recorded at 44100 Hz with two channels (stereo).

   (c) Why are wav files stored using little-endian byte representation even though the MicroBlaze is a big-endian processor? What are the costs associated with the conversion from little-endian to big-endian in your current application?

   (d) If you wanted to support 16-bit audio in your wav player application, what changes to your existing code would you have to make? Would your audio device driver have to change? If so, how? Hint: Research how 16-bit audio samples are stored in wav files.