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

Prof. Peng Li
TA: Andrew Targhetta
(Lab exercise created by A Targhetta and P Gratz)

Laboratory Exercise #9

AC '97 Codec Device Driver

Objective

The objective of lab this week is to provide exposure to a Linux device driver and hardware with real-time constraints, and the features of a standard AC'97 device. You will write a partial device driver that utilizes interrupts to continuously play a short stream of audio while the device is open. This will provide you with experience developing drivers that handle buffering of real-time data and lead you into the next lab where you will add user space interaction.

System Overview

The hardware system that you created in Lab 8 will be used again in this lab and is depicted in Figure 1. At this point, you have successfully designed the IR demodulation peripheral with interrupt capabilities. Thus, the focus of this lab is on the audio controller. For audio playback and record on the XUPV5 board, Xilinx provides the source code for version 1.00.a of the OPB AC '97 sound controller. This IP provides the link between the OPB and the AC '97 Audio Codec (Analog Devices AD1981B) on the XUP board. Documentation for the audio peripheral can be found in the ML40x EDK Processor Reference Design from Xilinx website. Version 1.00.a of the audio controller from Xilinx contained only 16 deep playback and record FIFOs. To increase the buffering capabilities of the audio controller, the design has been modified to include 8K deep playback and record FIFOs. The improved design is what you built into your system

last lab and is labeled version 1.01.a. As seen in Figure 2, serial communication is used to transfer audio samples and control/status information in and out of the audio codec. Thus, the Xilinx sound controller must handle the serialization/deserialization of data to and from the codec respectively. Additionally, the controller provides a significant amount of temporary storage for outgoing and incoming audio samples via the playback and record First In First Out (FIFO) buffers. The playback and record FIFOs are each 8K samples deep and are software accessible from the OPB through two 16-bit data registers. The audio controller has its own status register which provides the software with information regarding the status of the FIFOs (i.e. full or empty) and the status of the codec itself (i.e. ready or not ready). The audio controller control register lets the software clear the FIFOs. Three keyhole registers (i.e. address, data_in, data_out) provide the software with access to the control/status registers within the audio codec.

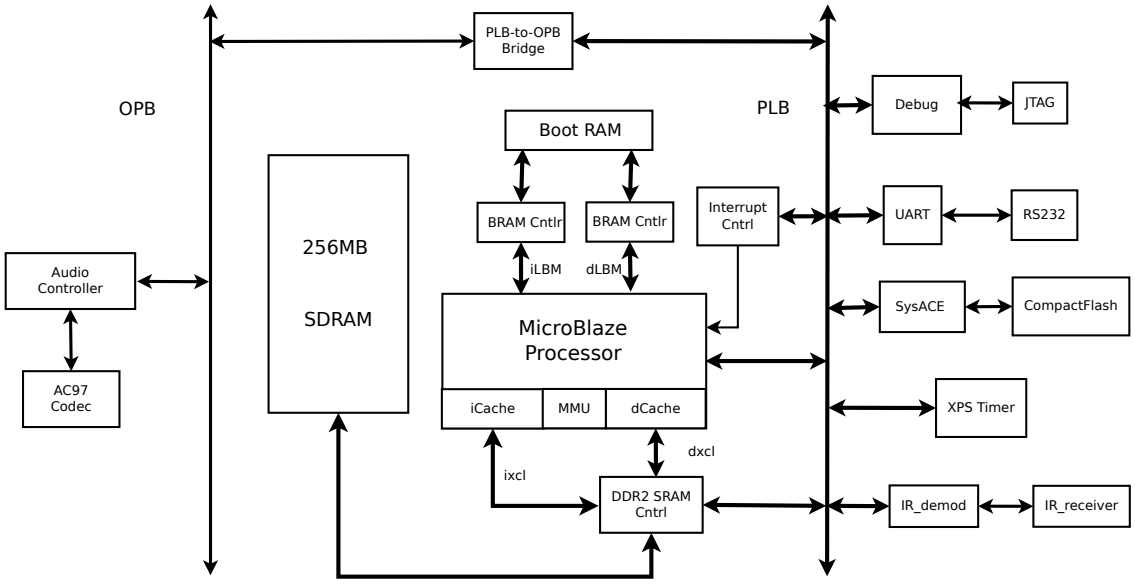

Figure 1: Hardware/Software System Diagram

An oversimplified block diagram of the AC '97 codec is shown in Figure 3. The two main components within the codec are the Analog to Digital Converter (ADC) and the Digital to Analog Converter (ADC)

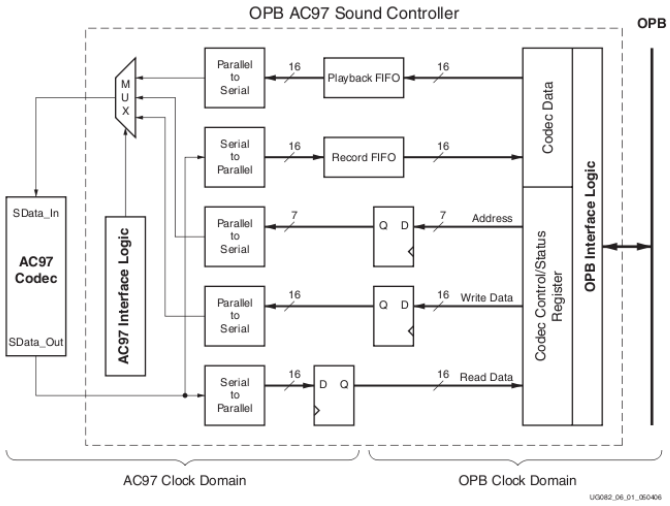

Figure 2: OPB AC97 Sound Controller Block Diagram

for record and playback respectively. In addition, the codec provides a certain amount of audio mixer and volume control along with playback/record rate control. For this lab, you will be provided with a header file, "xac97.h", which contains the control and status register definitions for setting up the codec to play audio. Additionally, a source file, "xac97.c", which contains example code for reseting and initializing the codec along with writing to the playback FIFO, will be provided. Please consult the Analog Devices AD1981B datasheet for a more detailed description.

Procedure

1. Please take a moment to look over the "xac97.h" and "xac97.c" source files found in the laboratory directory. Everything you need to interface with the AC '97 codec is provided in these files. Notice the code is commented so that you can understand how to use the provided functions. Also take a look at the "audio_samples.h" header file, which statically declares an array with 5 seconds worth of audio samples. The following steps will guide you through the process of creating a device driver that continually fills the audio controller playback buffer when the device driver is opened.

   (a) Copy the following files from '/homes/faculty/shared/ECEN449/' to your 'modules' directory: "xac97.h", "xac97.c", and "audio_samples.h".

   (b) Write a character device driver, "audio_buffer.c", which follows the guidelines provided below:

   - Include the "xac97.h" and "audio_samples.h" files in your device driver header file.

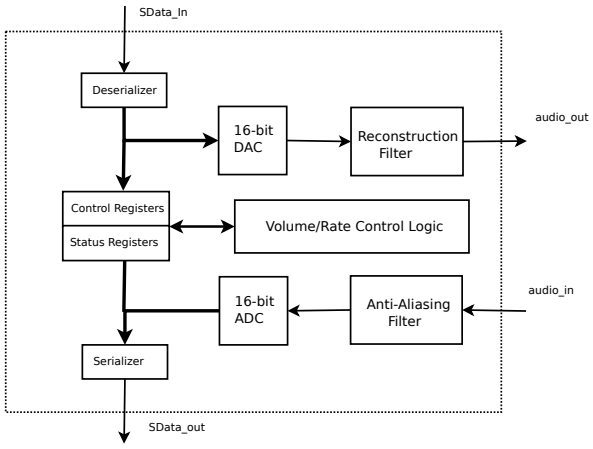

Figure 3: Simplified AC '97 Codec Block Diagram

- Perform the physical to virtual memory mapping in the initialization routine along with the initialization of any semaphores and the character device registration. Be sure to perform those operations in the proper order and handle errors appropriately as you will be graded on this.
- Use a semaphore within the open call to ensure only one device can open the driver at one time.
- Within the open routine, initialize the audio codec using the functions provided in the "xac97.c" source file and register the interrupt handler. The audio in "audio_samples.h" was sampled at 11025Hz so be sure to set the PCM playback rate accordingly. Consult the provided source files if this is unclear.
- The close routine should clear the playback FIFO and perform a 'soft' reset on the audio codec. In addition, it must unregister the interrupt handler.
- For both the read and write routines, print a kernel message stating that those operations are not supported. We will add support for the write system call in the next lab.
- Your interrupt handler should simply refill the playback buffer when an interrupt comes in. The interrupt, by default, will trigger when the buffer is half empty. A macro is provided for you in "xac97.h" which returns true when the playback buffer is full. The audio in "audio_samples.h" should play in repeat mode, meaning you must use a pointer to read from the 'audio_samples' array in a circular fashion.

(c) Modify your Makefile such that both "xac97.c" and "audio_buffer.c" will be compiled into a single kernel module called "audio_test.ko" and then run 'make ARCH=microblaze'. See below:

```
obj-m := audio_test.o
audio_test-objs := xac97.o audio_buffer.o
```

(d) Create a devtest that simple opens the device and remains in a while loop until the user enters 'q' in the terminal window to exit.

(e) Copy "audio_test.ko" and "devtest" to the compact flash, install the kernel module and run './devtest' to test your driver.

(f) Plug the speaker provided for you in lab into the headphone jack on the XUP board. Do you hear audio? If so, demonstrate your progress to the TA.

2. In the previous section, we enabled audio playback using a simple device driver that initializes the volume and playback rate when the device is opened. However, we would like to be able to change the various codec settings in the middle of playback. In the next few steps, we will add this functionality via the implementation of an ioctl() driver method. Just like read and write, we can implement an ioctl() routine that defines an additional way for the user to interface with our device. Specifically, ioctl() is used to transfer control and status information to and from the device driver respectively. The prototype for ioctl() is provided below:

```
static int device_ioctl(struct inode *, struct file *, unsigned
int, unsigned int*);
```

The first two parameters correspond to the file descriptor and are found in the other driver methods we have already implemented. For the last two parameters, no real standard usage exists. This gives the device driver developer freedom to implement ioctl() however they would like, but it can cause some confusion in the process.

Review Chapter 6 in the Linux Device Drivers Book, 3rd edition, for more details on the ioctl() interface.

For this lab, we will use the last two parameters as follows:

For control, the second to last parameter is used to specify a particular command, while the last parameter will be used in a pass by reference manner. This implies that the control value will be pointed to by the last parameter. For status, the second to last parameter specifies the requested status information, while the last parameter is an address of the variable where ioctl will place the requested information.

(a) First, copy the function prototype provided above into your device driver header file.

(b) From within your source file, locate the file operations structure and add the following line:

```
.ioctl = device_ioctl,
```

(c) Copy the code provided below into your device driver source file:

```
/*This function allows the user process to provide control
commands to our device driver and read status from the
device*/
static int device_ioctl(struct inode *inode,
                        struct file *file,
                        unsigned int cmd,
                        unsigned int *val_ptr)
{
        u16 val;//temporary value

        get_user(val,(u16*)val_ptr);//grab value from user space

        /*switch statement to execute commands*/
        switch(cmd)
        {
                /*adjust aux volume*/
                case ADJUST_AUX_VOL:
                  /*add code to adjust aux volume*/
                  break;

                /*adjust Master volume*/
                  case ADJUST_MAST_VOL:
                  XAC97_WriteReg(virt_addr, AC97_MasterVol, val);
                  break;

                /*adjust playback rate*/
                case ADJUST_PLAYBACK_RATE:
                  /*add code to adjust playback rate*/
                  break;

                /*if unknown command, error out*/
                default:
                  printk(KERN_INFO "Unsupported control commad!\n");
                  return -EINVAL;
        }

        return 0;
}
```

(d) Fill in the missing functionality described in the comments above.

(e) Modify your devtest such that it changes the AuxOut (headphone) volume and changes the playback rate. A "sound.h" file has been provided for you which contains a subset of the defines

found within the "xac97.h" file. That file is more appropriate for user applications that utilize your audio device driver, and it may be copied from the '/homes/faculty/shared/ECEN449' directory.

(f) Install your audio device driver into the XUP Linux system, and run your devtest application. Demonstrate your progress to the TA.

Deliverables

1. [7 points.] Demo the working Audio driver 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) The ioctl() interface is slowly becoming depreciated in the Linux kernel. Why are kernel developers moving away from it? Outline a different method to achieve the same end.
   (b) What would be the effect of using a small, 16 word buffer as in the original AC'97 device?
   (c) Currently the interrupt trigger point is set to half empty on the playback buffer. What might be the consequences of lowering the trigger point? What would be the effects of raising the trigger point?
   (d) In your current audio device driver, a significant amount of data transfer is being performed within the interrupt handler. What problems could this cause? Skim through chapter 10 of *Linux Device Drivers, Third Edition* and provide a solution to the existing problem.