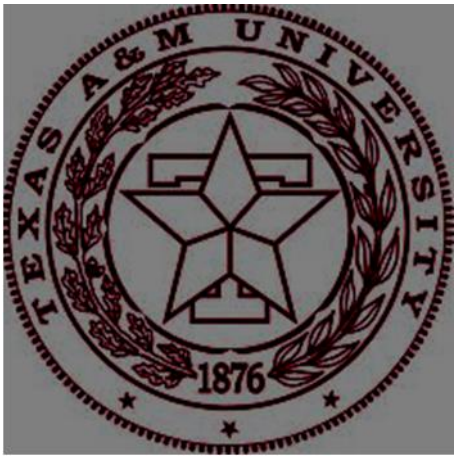
ECEN 449 - Microprocessor System Design

Section 503

Department of Electrical and Computer Engineering

Lab 9: AC '97 Codec Device Driver



Sridhar Mareguddi

UIN: 823000772

Date: 4/20/2014

Objective

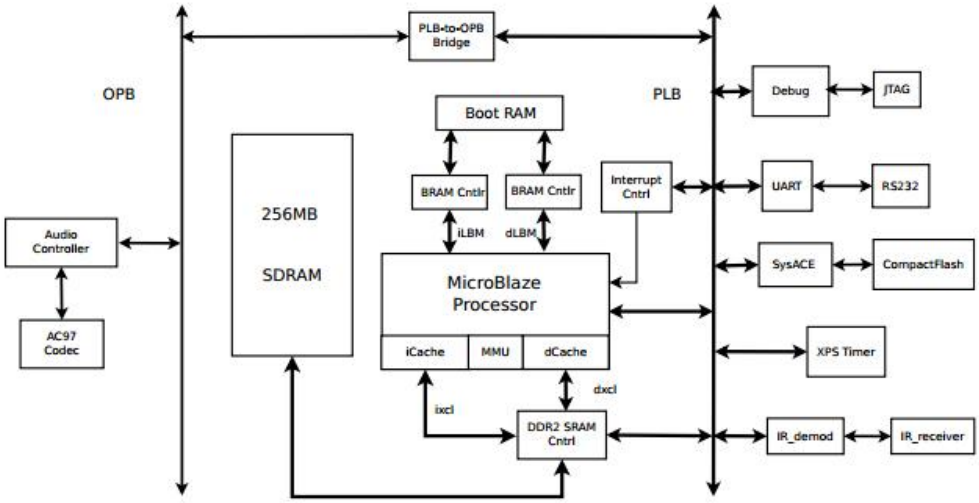
The objective of lab this Lab is to write a Linux device that handles buffering of real-time audio data and continuously play a short stream of audio while the device is open.

Resources Used

1. Xilinx FPGA for implementing Decoding hardware
2. Speaker
3. Programming Language – C, Verilog

Procedure

A simplified view of the system for lab8 is depicted below



1. Copy the template files “xac97.h”, “xac97.c”, and “audio samples.h” from ‘/homes/faculty/shared/ECEN449/’ to Lab9 ‘modules’ directory:
2. Write a character device driver, called “audio buffer.c with the following guidelines.
3. Perform the physical to virtual memory mapping in the initialization routine using ioremap().Use a semaphore within the open call to ensure only one device can open the driver at one time.
4. Within the open routine, initialize the audio codec and register the interrupt handler and set the PCM playback rate to 11025Hz
5. Within the close routine, clear the playback FIFO and perform a soft reset on audio codec and unregister the interrupt handler.
6. For both the read and write routines, print a kernel message stating that those operations are not supported.
7. The interrupt handler routine should refill the playback buffer through an interrupt

8. Which is triggered when the buffer is half empty. Play the samples defined in audio samples.h in a repeated mode, by reading samples from the 'audio samples' array in a circular fashion.
9. Compile the audio_buffer files by modifying the Makefile.
10. Create a devtest application that simply opens the device and remains in a while loop until the user enters 'q' in the terminal window to exit.
11. Implement an ioctl() routine that is used to transfer control and status information to and from the device driver respectively.
12. 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.
13. Copy the function prototype provided above into the device driver header file and the add the ioctl to the file operations structure.
14. Use the template code provided in the manual and complete missing functionality.
15. Modify the devtest such that it changes the AuxOut (headphone) volume and changes the play back rate.
16. Install the audio device driver into the XUP Linux system, and run the devtest application.

Results

The AC97 device driver was configured to implement the interrupt controlled audio playback through the speakers on the Microblaze processor system. The ioctl functionality was also verified by observing the output on making changes to the volume and playback rate for the audio.

Conclusion

The Xilinx FPGA – XUP board along with system.ace flash containing Linux, was used to implement the AC97 device driver using XPS as the platform and Verilog and C as the coding language. The correct functionality of the interrupt driven audio playback buffer/FIFO was observed using the device driver on the XUP board for audio playback using speakers connected to the aux out on the FPGA board

Additional Questions

1. **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.**

   The ioctl interface is mostly used by the custom programs, since it is difficult to transfer data through an ioctl connected to a pipeline of standard programs.

   The other ways of communicating with kernel drivers are:

- Using a /proc provides an interface to a device driver which is independent of any i/o through the regular driver mechanism.
- Directly through read() and/or write() system calls to cause driver action.

2. **What would be the effect of using a small, 16 word buffer as in the original AC'97 device?**
   We use an 8k buffer for playback and hence with a 16 word buffer, it will fill up quickly causing undesired audio playback since the interrupt handler will not be called correctly and prevent writing of sample values

3. **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?**
   If the interrupt trigger point is lower than half empty, then the interrupt will be triggered frequently. So it will result in this interrupt being serviced a lot of time than other interrupts. If the trigger point is raised, then the buffer will be full frequently. So data may not be written to the buffer resulting in delay in playing the file.

4. **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**
   Since the interrupt handler is executed only on an interrupt trigger, it does not run in the context of the process. So the value that was copied during that time may not be relevant to the value that was intended to be passed on. So these data transfer operation must be avoided in interrupt handler code.

Appendix

Audio device driver module 'audio_buffer.c'

```
#include "xac97.h"
#include "audio_samples.h"
#include "irq_test.h"

#include "/homes/grad/sridharm/ecen449/lab9/microblaze_0/include/xparameters.h"



#define PHY_ADDR XPAR_OPB_AC97_CONTROLLER_REF_0_BASEADDR
#define MEMSIZE XPAR_OPB_AC97_CONTROLLER_REF_0_HIGHADDR -
XPAR_OPB_AC97_CONTROLLER_REF_0_BASEADDR

void * virt_addr;
static struct file_operations fops = {
  .read = device_read,
  .write = device_write,
  .open = device_open,
  .release = device_release,
  .ioctl = device_ioctl
};


int my_init(void)
{
  printk(KERN_INFO "MAPPING VIRTUAL ADDRESS\n");
  virt_addr = ioremap(PHY_ADDR,MEMSIZE);
```

```
  init_waitqueue_head(&queue);           /* initialize the wait queue */

  /* Initialize the semaphor we will use to protect against multiple
     users opening the device  */
  sema_init(&sem, 1);

  Major = register_chrdev(0, DEVICE_NAME, &fops);
  if (Major < 0) {
    printk(KERN_ALERT "Registering char device failed with %d\n", Major);
    return Major;
  }
  printk(KERN_INFO "Registered a device with dynamic Major number of %d and virt_addr = %d\n", Major,(unsigned
int)virt_addr);
  printk(KERN_INFO "Create a device file for this device with this command:\n'mknod /dev/%s c %d 0'.\n", DEVICE_NAME,
Major);

  return 0;               /* success */
}

/*
 * This function is called when the module is unloaded, it releases
 * the device file.
 */
void my_cleanup(void)
{
  /*
   * Unregister the device
   */
  unregister_chrdev(Major, DEVICE_NAME);
  printk(KERN_ALERT "Unmapping Virtual Address Space...\n");
  iounmap((void*)virt_addr);
}


/*
 * Called when a process tries to open the device file, like "cat
 * /dev/irq_test".  Link to this function placed in file operations
 * structure for our device file.
 */
static int device_open(struct inode *inode, struct file *file)
{
  int irq_ret;

  if (down_interruptible (&sem))
              return -ERESTARTSYS;

  /* We are only allowing one process to hold the device file open at
     a time. */
  if (Device_Open){
    up(&sem);
    return -EBUSY;
  }
  Device_Open++;

  /* OK we are now past the critical section, we can release the
     semaphore and all will be well */
  up(&sem);


  /* Initialize the audio codec */
  XAC97_InitAudio(virt_addr,0);

  /** Enable VRA Mode **/
  XAC97_WriteReg(virt_addr, AC97_ExtendedAudioStat, AC97_EXTENDED_AUDIO_CONTROL_VRA);
   /* Set Playback rate */
  XAC97_WriteReg(virt_addr,AC97_PCM_DAC_Rate0, AC97_PCM_RATE_11025_HZ);
  XAC97_WriteReg(virt_addr, AC97_AuxOutVol, AC97_VOL_MAX);
```

```
  /* request a fast IRQ and set handler */
  irq_ret = request_irq(IRQ_NUM, irq_handler, 0 /*flags*/ , DEVICE_NAME, NULL);
  if (irq_ret < 0) {                /* handle errors */
    printk(KERN_ALERT "Registering IRQ failed with %d\n", irq_ret);
    return irq_ret;
  }
  try_module_get(THIS_MODULE);     /* increment the module use count
                                      (make sure this is accurate or you
                                      won't be able to remove the module
                                      later. */

  msg_Ptr = NULL;
  printk(KERN_ALERT "RANDOM PRINT \n");
  return 0;
}

/*
 * Called when a process closes the device file.
 */
static int device_release(struct inode *inode, struct file *file)
{
  Device_Open--;                /* We're now ready for our next caller */

  XAC97_ClearFifos(CLEAR_PLAYBACK_FIFO);
  XAC97_SoftReset(virt_addr);
  free_irq(IRQ_NUM, NULL);
  /*
   * Decrement the usage count, or else once you opened the file,
   * you'll never get get rid of the module.
   */
  //kfree(msg_queue);
  module_put(THIS_MODULE);

  return 0;
}

static ssize_t device_read(struct file *filp,           /* see include/linux/fs.h   */
                           char *buffer,        /* buffer to fill with data */
                           size_t length,       /* length of the buffer     */
                           loff_t * offset)
{

   /* not allowing writes for now, just printing a message in the
      kernel logs. */
  printk(KERN_ALERT "Sorry, READ operation isn't supported.\n");
  return -EINVAL;               /* Fail */
}

/*
 * Called when a process writes to dev file: echo "hi" > /dev/hello
 * Next time we'll make this one do something interesting.
 */
static ssize_t device_write(struct file *filp, const char *buff, size_t len, loff_t * off)
{

    /* not allowing writes for now, just printing a message in the
      kernel logs. */
  printk(KERN_ALERT "Sorry, WRITE operation isn't supported.\n");
  return -EINVAL;               /* Fail */
}

static int device_ioctl(struct inode *inode ,struct file * file, unsigned int cmd, unsigned int *val_ptr)
{

  u16 val;
  get_user(val,(u16 *)val_ptr);

  switch(cmd)
```

```
 {
            case ADJUST_AUX_VOL:
            printk("inside aux volume \n");
            XAC97_WriteReg(virt_addr, AC97_AuxOutVol, val);
            break;

            case ADJUST_MAST_VOL:
            printk("inside mast volume \n");
            XAC97_WriteReg(virt_addr, AC97_MasterVol, val);
            break;

            case ADJUST_PLAYBACK_RATE:
            XAC97_WriteReg(virt_addr, AC97_PCM_DAC_Rate, val);
            break;

            default:
            printk(KERN_INFO "Unsupported control command\n");
            return -EINVAL;
  }
  return 0;
}


irqreturn_t irq_handler(int irq, void *dev_id) {
  static int counter = 0;           /* keep track of the number of
                                              interrupts handled */
  static int i=0;

  while(XAC97_isInFIFOFull(virt_addr)!=1)
  {
            XAC97_WriteFifo(virt_addr, audio_samples[i]);
            XAC97_WriteFifo(virt_addr, audio_samples[i]);
            i++;
            if(i==NUM_SAMPLES)
                        i=0;
  }
  sprintf(msg, "IRQ Num %d called, interrupts processed %d times\n", irq, counter++);
  msg_Ptr = msg;

  wake_up_interruptible(&queue);   /* Just wake up anything waiting
                                              for the device */
  return IRQ_HANDLED;
}

/* These define info that can be displayed by modinfo */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Paul Gratz and Others");
MODULE_DESCRIPTION("Module which creates a audio device driver and allows user interaction with it");

/* Here we define which functions we want to use for initialization
   and cleanup */
module_init(my_init);
module_exit(my_cleanup);
```

Device Test Application ‘dev_test.c’

```
# include <sys/types.h>
# include <sys/stat.h>
# include <fcntl.h>
# include <sys/ioctl.h>
# include <stdio.h>
# include <unistd.h>
# include <stdlib.h>
# include "sound.h"
```

```
int main()
{

char ch;
int fd,option;
fd=open("/dev/irq_test",O_RDWR);

if(fd==-1)
{
  printf("Failed to open device! \n");
  return -1;
}

while(1)
{

  printf("Enter your option :\n");
  printf(" 1. MUTE \n");
  printf(" 2. Audio Min \n");
  printf(" 3. Audio Mid \n");
  printf(" 4. Audio Max \n");
  printf(" 5. Adjust PlayBack Rate to 48 KHz \n");
  printf(" 6. Adjust PlayBack Rate to 1.1025 KHz \n");
  scanf("%d", &option);

              //while(1);


  switch (option)
              {

              case 1:
                          ioctl(fd, ADJUST_AUX_VOL, AC97_VOL_MUTE);
                          sleep(5);
                          break;
              case 2:
                          ioctl(fd, ADJUST_AUX_VOL, AC97_VOL_MIN);
                          sleep(5);
                          break;
              case 3:
                          ioctl(fd, ADJUST_AUX_VOL, AC97_VOL_MID);
                          sleep(5);
                          break;
              case 4:
                          ioctl(fd, ADJUST_AUX_VOL, AC97_VOL_MAX);
                          sleep(5);
                          break;
              case 5:
                          ioctl(fd, ADJUST_PLAYBACK_RATE, AC97_PCM_RATE_48000_HZ);
                          sleep(10);
                          break;
              case 6:
                          ioctl(fd, ADJUST_PLAYBACK_RATE, AC97_PCM_RATE_11025_HZ);
                          sleep(10);
                          break;
              default:
                          printf(" Invalid option - Try again ");
                          break;
              }
}

close(fd);
return 0;
}
```