Kyle Loyka

Lab 9 Report

ECEN 449-503

Due: 2 May 2016

Introduction
The purpose of this lab was to gain experience building device drivers. This lab also introduced the concept of the *ioctl* function.

Procedure
In the first part of the lab, the interrupt handler was programmed to refill the playback FIFO when needed. This handler function made the AC97 device output a looped audio clip. The interrupt handler made it so this audio clip would never stop playing.

In the second part of the lab, the device's *ioctl* function was implemented. This feature allows a user or programmer to alter certain parameters on the AC97 device. In this lab, the *ioctl* function gave the user the ability to adjust playback rate and output volume.

Results / Q&A
The lab worked as expected.

(a) *ioctl* is being phased out by kernel developers for more efficient functions. The issue with *ioctl* is that it blocks any other system calls from running in other threads of execution. This "feature" can create long latencies for unrelated processes. A similar function called *unlocked_ioctl* has been introduced to solve this problem. The *unlocked_ioctl* function allows the driver writer to choose which parts of the kernel to lock (as opposed to locking all of the kernel). This may allow other (unlocked) system functions to continue running (on multithreaded systems).

(b) By having a smaller buffer size, the buffer would have to be refilled more frequently. This would cause more calls to the interrupt handler, and the additional context switches would decrease overall performance of the computing system.

(c) With a lower trigger point, the interrupt handler will be called less frequently to write more data to the buffer. This results in less interrupts with more data written per function call. The driver programmer has to be careful though. If the buffer is emptied before the interrupt is triggered and handled, the audio will stop playing and the user will notice a pause in their music playback. If the trigger point was raised, the interrupt handler would be called more frequently to fill the buffer. This would result in more interrupts with less data written per function call. The driver programmer has to be careful not to make the trigger too sensitive, or the interrupt handler will be called too frequently, impacting unrelated process on the CPU.

(d) When an interrupt occurs, the CPU stops its current task and switches to the device's interrupt handler process. An interrupt handler with a significant amount of data transfer may take a long period of time to complete. This could possibly block other interrupts from being handled (from this same device, or from other devices).

Conclusion
This lab provided valuable experience in build device drivers and interrupt signals. This lab also exposed us to *ioctl* functions and their ability to modify device parameters.

audio_buffer.c

```
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/sched.h>
#include <asm/uaccess.h>
#include <linux/semaphore.h>
#include <linux/wait.h>
#include <linux/interrupt.h>
#include <asm/io.h>
#include <asm/ioctl.h>
#include <linux/ioctl.h>
#include <linux/slab.h>
#include "xac97.h"
#include "audio_samples.h"
#include "xparameters.h"

#define PHY_ADDR XPAR_OPB_AC97_CONTROLLER_REF_0_BASEADDR
#define MEM_SIZE XPAR_OPB_AC97_CONTROLLER_REF_0_HIGHADDR - PHY_ADDR + 1
#define DEVICE_NAME "ac97"
#define IRQ_NUM 1 // interrupt code

void* virt_addr;  // address mapped to beginning of the AC97 controller's
memory
static int Major; // major number assigned to device driver
static int Device_Open = 0; // flag to signify open device
static struct semaphore mutex; // semaphore to prevent more than one open
device

/* function declarations */
int my_init(void);
void my_cleanup(void);
static int device_open(struct inode *, struct file *);
static int device_release(struct inode *, struct file *);
static ssize_t device_read(struct file *, char *, size_t, loff_t *);
static ssize_t device_write(struct file *, const char *, size_t, loff_t *);
static irqreturn_t irq_handler(int irq, void* dev_id);
static int device_ioctl(struct inode *, struct file *, unsigned int, unsigned
int *);

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

/* This function is called when the module is loaded and registers a device
```

```
   for the driver to use */
int my_init(void){
      // explanation of code: (lab9-step.substep.bulletpoint)
      printk(KERN_ALERT "Allocating virtual memory\n");
      // (1.b.2) Virtual Memory Declaration
      virt_addr = ioremap(PHY_ADDR,MEM_SIZE);

      // (1.b.2) Initialize semaphore so only one user can open the device at
a time
      sema_init(&mutex,1);

      // (1.b.2) register character device
      Major = register_chrdev(0,DEVICE_NAME,&fops);
      if(Major < 0) {
            printk(KERN_ALERT "Registering char device failed with
%d\n",Major);
            iounmap( (void*)virt_addr); // unmap the io in case of device
failure
            return Major;
      }
      printk(KERN_INFO "Registered a device with dynamic Major number of
%d\n",Major);
      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);
      iounmap( (void*)virt_addr);
}


/* This function is called when a process tries to open the device file. */
static int device_open(struct inode *inode, struct file *file){

      printk(KERN_INFO "> starting device_open.\n");
      int irq_ret;
      // (1.b.3)
      /* lock mutex to prevent more multiple procs from reading/writing
                  the Device_Open object */
      if(down_interruptible(&mutex)) return -1;
      /* only allow 1 process to hold the device open at a time */
      if(Device_Open){
            up(&mutex);
            return -EBUSY;
      }
      Device_Open++;
      /* now past the critical section, release the mutex */
      up(&mutex);
```

```
      printk(KERN_INFO "> passed the critical section.\n");

      /* (1.b.4) time to configure the AC97 device */
      // the device acts as one large register.
      // different features/settings can be accessed through virt_addr +
offset.
      // initialize buffer.
      printk(KERN_INFO "> initializing buffer.\n");
      XAC97_InitAudio(virt_addr,0);
      // changing hardware sample rate
      printk(KERN_INFO "> initializing hardware.\n");
      XAC97_WriteReg(virt_addr, AC97_ExtendedAudioStat,
AC97_EXTENDED_AUDIO_CONTROL_VRA);
      XAC97_WriteReg(virt_addr, AC97_PCM_DAC_Rate, AC97_PCM_RATE_11025_HZ);
      // done configuring device
      printk(KERN_INFO "> registering IRQ.\n");
      /* request a fast IRQ and set handler */
      irq_ret = request_irq(IRQ_NUM, irq_handler, 0, 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); /* incriment the module use count */
      printk(KERN_INFO "end of 'device_open'\n");
      return 0;
}



/* called when a process closes the device file */
static int device_release(struct inode *inode, struct file *file){
      // clear the playback FIFO and perform a 'soft' reset of the audio
codec.
      // also unregister the interrupt handler.
      // (1.b.5)
      Device_Open--;
      XAC97_ClearFifos(virt_addr);
      XAC97_SoftReset(virt_addr);
      free_irq(IRQ_NUM,NULL);

      // decriment usage count
      module_put(THIS_MODULE);
      return 0;
}


/* called when a process, which already opened the dev file,
            attempts to read from it */
static int device_read(struct file *filp, char *buffer,
                                                                  size_t
length, loff_t * offset){
      //(1.b.6)
      printk(KERN_ALERT "operation not supported.\n");
```

```
	return -EINVAL;
}

/* called when a process writes to dev file */
static int device_write(struct file *filp,const char *buff,

	size_t len,loff_t* off){
	//(1.b.6)
	printk(KERN_ALERT "operation not supported.\n");
	return -EINVAL;
}

/* allows the user to provide control commands to the 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){
	u32 val; //temporary value

	get_user(val,(u32*)val_ptr); //get value from user space

	/* switch statement to execute commands */
	switch(cmd){
		/* adjust aux volume */
		case ADJUST_AUX_VOL:
			XAC97_WriteReg(virt_addr, AC97_AuxOutVol, val);
			//XAC97_WriteReg(virt_addr, AC97_AuxOutVol, val);
			break;
		case ADJUST_MAST_VOL:
			XAC97_WriteReg(virt_addr, AC97_MasterVol, val);
			//XAC97_WriteReg(virt_addr, AC97_MasterVol, val);
			break;
		case ADJUST_PLAYBACK_RATE:
			XAC97_WriteReg(virt_addr, AC97_PCM_DAC_Rate, val);
			//XAC97_WriteReg(virt_addr, AC97_PCM_DAC_Rate, val);
			break;
		/* if unknown command, error out */
		default:
			printk(KERN_INFO "Unsupported control command!\n");
			return -EINVAL;
	}
	return 0;
}

//handles interrupts
irqreturn_t irq_handler(int irq, void* dev_id){
	static unsigned int index = 0;
	//u32 sample;
	while(!XAC97_isInFIFOFull(virt_addr)){
		// if the FIFO isn't full, fill the FIFO
		//XAC97_WriteFifo(virt_addr,(u32)audio_samples[index]);
		//++index;
		//if (index == NUM_SAMPLES) index = 0;

		XAC97_WriteFifo(virt_addr,audio_samples[index%NUM_SAMPLES]);
```

```
            XAC97_WriteFifo(virt_addr,audio_samples[index%NUM_SAMPLES]);
            ++index;
      }
      return IRQ_HANDLED;
}


MODULE_LICENSE("GPL");
MODULE_AUTHOR("...");
MODULE_DESCRIPTION("Module which allows for the user to listen to audio using
the AC97 hardware");

module_init(my_init);
module_exit(my_cleanup);
```

devtest.c

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



int main(void){
      int fd = 0;
      fd = open("/dev/ac97", O_RDWR);
      if(fd < 0 ) return -1;

      printf("part '1' or '2'\n");
      char buffer[10];
      char *line;
      line = fgets(buffer,sizeof(buffer),stdin);

      if (line[0] == '1'){
            /* running code for part 1 */
            printf("playing music for part 1...\n");
```

```
		while(1) {
			line = fgets(buffer,sizeof(buffer), stdin);
			if((line == NULL) || (toupper(line[0]) == 'Q')) break;
		}
	}

	else {
		/* running code for part 2 */
		unsigned short int val = 0;
		printf("playing music for part 2...\n");
		usleep(2000000);

		printf("lowering aux volume\n");
		val = AC97_VOL_MIN;
		ioctl(fd,0x01DEAD,&val);
		usleep(4000000);

		printf("raising aux volume\n");
		val = AC97_VOL_MAX;
		ioctl(fd,ADJUST_AUX_VOL,&val);
		usleep(4000000);

		printf("lowering master volume\n");
		val = AC97_VOL_MUTE;
		ioctl(fd,ADJUST_MAST_VOL,&val);
		usleep(4000000);

		printf("raising master volume\n");
		val = AC97_VOL_MAX;
		ioctl(fd,ADJUST_MAST_VOL,&val);
		usleep(4000000);

		printf("decreasing playback rate\n");
		val = AC97_PCM_RATE_8000_HZ;
		ioctl(fd,ADJUST_PLAYBACK_RATE,&val);
		usleep(4000000);

		printf("increasing playback rate\n");
		val = AC97_PCM_RATE_48000_HZ;
		ioctl(fd,ADJUST_PLAYBACK_RATE,&val);
		usleep(4000000);

		printf("done!\n");
	}
	printf("closing device\n");
	close(fd);
	return 0;
}
```