ECEN 749 LAB REPORT

Excercise #8:Interrupt-Based IR-remote Device Driver

Tanmay Verma

12.1.2017

INTRODUCTION

The objective of lab this week is to develop an interrupt based device driver to display the messages received by the demodulator designed in the previous lab

PROCEDURE

The following steps were carried out:

I. **Part A: Add an Interrupt signal to the demodulator**

1. Created a copy of the project from the Lab_7
2. Opened the “ir_demod” IP in ‘Edit in IP Manager’. And modified the AXI code to include IR_interrupt signal as the output.
3. Added the logic for sending and clearing the interrupt.
4. The IR_interrupt signal was made external for observational purpose.
5. A software test routine was designed in SDK after exporting the bitstream.
6. Correct operation was verified on the CRO unit.

II. **Part B: Connect the Interrupt to PS system and write the device driver to handle it.**

1. The IR_interrupt signal was removed from external ports and connected to the PS system.
2. The bitstream was created and exported to SDK.
3. A FSBL project was created from SDK and BOOT.bin image was created.
4. The device tree was modified to include the developed ir_demod module and dtb was created.
5. The irq_test driver code was taken and modified to function as per our requirements.
6. The module program was compiled and the kernel object was prepared.
7. A devtest.c file was written to test the functionality of our driver module by reading the messages continuously.
8. The test code was compiled.
9. All the required files were copied to the SD card and the SD card was connected to the Zybo Board and the jumper was set to boot from SD Card.
10. After the board successfully booted, the SD card was mounted in a directory point.
11. The irq_test.ko module was installed in the system using insmod command.

12. ./devtest was run and the proper operation of interrupt handler was verified.

RESULTS

Through this exercise, in the first step an interrupt was added in the ir_demod h/w and in second part a driver module was successfully written and compiled on the centos machine and copied to the SD card to be integrated with the Linux Operating System running on Zybo Board. The module successfully interacted with the ir_demod unit via interrupt handling mechanism.

CONCLUSION

In conclusion, at the end of the exercise, I successfully compiled a driver module to successfully handle interrupts.

Verilog Code

```
reg IR_signal_buf;

wire rising_edge;

wire falling_edge;

reg count_enable;

reg [31:0]counter;

wire clr_int_rst;

reg clkdiv;

reg [31:0]div_counter;

reg [C_S_AXI_DATA_WIDTH1:0] temp_reg;

reg [5:0] message_count;

// To detect the change in IR_signal level

assign rising_edge = IR_signal & (~IR_signal_buf);

assign falling_edge = (~IR_signal) & IR_signal_buf;

// Divides clock to an update at every pulse

3   always @(posedge S_AXI_ACLK) begin

        if ( S_AXI_ARESETN == 1'b0 ) begin

        clkdiv <= 0;

        div_counter <= 0;

        end else if (div_counter == 'd11250) begin

        clkdiv <= ~clkdiv;

        div_counter <= 0;

        end else begin
```

```
            div_counter <= div_counter + 1;
            end
   end
   // Updates the buffered value at every divided clock
   always @(posedge clkdiv) begin
            IR_signal_buf <= IR_signal;
   end

            // Updates the intermediate states until 12 bits are
received. After 12bits slv_reg0 and slv_reg1 are updated
   always @(posedge clkdiv) begin
            if ( S_AXI_ARESETN == 1'b0 ) begin
            count_enable <= 0;
            message_count <= 0;
            temp_reg <= 0;

            slv_reg0 <= 0;
            slv_reg1 <= 0;
            end else begin
            if(falling_edge) begin
            count_enable <= 1;
            end else if(rising_edge) begin
            count_enable <= 0;
            if(counter <= 32'd3) begin
```

```
message_count <= message_count + 1;
temp_reg <= {temp_reg[30:0], 1'b0};
end else if (counter <= 32'd5) begin
message_count <= message_count + 1;
temp_reg <= {temp_reg[30:0], 1'b1};
end else if (counter <= 32'hA) begin
temp_reg <= 0;
slv_reg0 <= temp_reg;
4 slv_reg1 <= slv_reg1 + 1;
end
end

if(message_count == 32'd12)
message_count <= 0;
end
end
/* Drives the slave 2 register for interrupt */
always@(posedge clkdiv) begin
if ( S_AXI_ARESETN == 1'b0 ) begin
slv_reg2[15:0] <= 16'd0;
end begin
if(slv_reg2[31:16] != 16'd0) begin
slv_reg2[15:0] <= 16'd0;
clr_int_rst <= 1;
end  else begin
```

```
        clr_int_rst <= 0;
        end

        if(message_count == 'd12) begin
        slv_reg2[15:0] <= 16'd1;
        end

end
end

assign IR_interrupt = slv_reg2[0];

//Runs the counter to detect the pulse width
  always @(posedge clkdiv) begin
        if ( S_AXI_ARESETN == 1'b0 ) begin
        counter <= 0;
        end else begin
        if(count_enable) begin
        counter <= counter + 1;
        end else begin
        counter <= 0;
        end
        end
  end
```

Helloworld.c for Part1

```
#include "xil_io.h"

#include <stdio.h>

#include "platform.h"

#include "xparameters.h"

#include "ir_demod.h"


int main()

{

    init_platform();


    u32 reg0 = 0;

    int i;

    u32 reg1 = 0;

    u32 reg2 = 0;

    u32 mask = 0x00010000;


    u32 temp_reg1 = 0;


    while(1) {

    reg0 = IR_DEMOD_mReadReg((u32)XPAR_IR_DEMOD_0_S00_AXI_BASEADDR, 0);

    reg1 = IR_DEMOD_mReadReg((u32)XPAR_IR_DEMOD_0_S00_AXI_BASEADDR, 4);

    reg2 = IR_DEMOD_mReadReg((u32)XPAR_IR_DEMOD_0_S00_AXI_BASEADDR, 8);
```

```
	reg3 = IR_DEMOD_mReadReg((u32)XPAR_IR_DEMOD_0_S00_AXI_BASEADDR, 12);
	if(reg2 != 0) {

		xil_printf("Message register reg0 = 0x%x, Count register reg1 = 0x%x reg2 = 0x%x \n\r", reg0, reg1, reg2);
		reg2 = reg2 | mask;
		IR_DEMOD_mWriteReg((u32)XPAR_IR_DEMOD_0_S00_AXI_BASEADDR, 8, reg2);

	}
	temp_reg1 = reg1;
	}
	cleanup_platform();
	return 0;
}
```

irq_demod.c Module

```
/* Moved all prototypes and includes into the header file */
#include "ir_demod.h"

/* This structure defines the function pointers to our functions for
   opening, closing, reading and writing the device file.  There are
   lots of other pointers in this structure which we are not using,
   see the whole definition in linux/fs.h */
static struct file_operations fops = {
  .read = device_read,
  .write = device_write,
  .open = device_open,
  .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)
{

  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);


            printk(KERN_INFO "Mapping virtual address...\n");

            virt_addr = ioremap(PHY_ADDR, MEMSIZE);                    /* map virtual address to
multiplier physical address */

            /* print the virtual and physical addresses */

            printk(" %x Physical address of the ir_demod peripheral is mapped to  %p\n",
PHY_ADDR, virt_addr);

  Major = register_chrdev(0, DEVICE_NAME, &fops);

  if (Major < 0) {

            printk(KERN_ALERT "Registering char device failed with %d\n", Major);

                        printk(KERN_ALERT "unmapping virtual address space...\n");

                        iounmap((void*)virt_addr);

            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);

}



/*
 * 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);

  /* 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. */

  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 */

  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.
   */
  module_put(THIS_MODULE);

  return 0;
}

/*
 * Called when a process, which already opened the dev file, attempts to
 * read from it.
 */
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)
{
  int bytes_read = 0;

  /*
   * Actually put the data into the buffer
   */
  while (length && (tail_ptr != head_ptr)) {

          /*
```

```
        * The buffer is in the user data segment, not the kernel
        * segment so "*" assignment won't work.  We have to use
        * put_user which copies data from the kernel data segment to
        * the user data segment.
        */
        if(tail_ptr == (msg+BUF_LEN)){
        tail_ptr = NULL;
        }
        if(tail_ptr == NULL){
        tail_ptr = msg;
        }
        put_user(*(tail_ptr++), buffer++); /* one char at a time... */

        length--;
        bytes_read++;
  }

  /*
   * Most read functions return the number of bytes put into the buffer
   */
  return bytes_read;
}

/*
 * 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, this operation isn't supported.\n");
  return -EINVAL;       /* Fail */
}

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

    printk("IRQ Num %d called, interrupts processed %d times\n", irq, counter++);

  wake_up_interruptible(&queue);   /* Just wake up anything waiting
                                           for the device */
   if(head_ptr == NULL){
             head_ptr = msg;
    }
            *(head_ptr) = ioread8(virt_addr);
            head_ptr++;
            *(head_ptr) = ioread8(virt_addr+1);
            head_ptr++;
   iowrite8(1, virt_addr+10);
   if(head_ptr == (msg+BUF_LEN)){
```

```
    head_ptr = NULL;
   }
  return IRQ_HANDLED;
}
```

```
/* These define info that can be displayed by modinfo */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Paul V. Gratz (and others)");
MODULE_DESCRIPTION("Module which creates a character device 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);
```

irq_demod.h Driver

```
/* All of our linux kernel includes. */
#include <linux/module.h>  /* Needed by all modules */
#include <linux/moduleparam.h>  /* Needed for module parameters */
#include <linux/kernel.h>  /* Needed for printk and KERN_* */
#include <linux/init.h>        /* Need for __init macros  */
#include <linux/fs.h>          /* Provides file ops structure */
#include <linux/sched.h>   /* Provides access to the "current" process
```

```
                              task structure */
#include <asm/uaccess.h>   /* Provides utilities to bring user space
                              data into kernel space.  Note, it is
                              processor arch specific. */
#include <asm/io.h>                  /* Needed for IO reads and writes */
#include <linux/semaphore.h>    /* Provides semaphore support */
#include <linux/wait.h>        /* For wait_event and wake_up */
#include <linux/interrupt.h>    /* Provide irq support functions (2.6
                                        only) */
#include "xparameters.h"     /* Needed for physical address of multiplier */

/* Some defines */
#define DEVICE_NAME "ir_demod"
#define BUF_LEN 200
#define IRQ_NUM 61

/* from xparameters.h file */
#define PHY_ADDR XPAR_MULTIPLY_0_S00_AXI_BASEADDR  // physical address of
multiplier

/* size of physical address range for multiply */
#define MEMSIZE XPAR_MULTIPLY_0_S00_AXI_HIGHADDR -
XPAR_MULTIPLY_0_S00_AXI_BASEADDR+1
/* Function prototypes, so we can setup the function pointers for dev
   file access correctly. */
int init_module(void);
void cleanup_module(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);

/*
 * Global variables are declared as static, so are global but only
 * accessible within the file.
 */
static int Major;          /* Major number assigned to our device driver */
static int Device_Open = 0;    /* Flag to signify open device */
static char msg[BUF_LEN];    /* The msg the device will give when asked */
static char *tail_ptr;
static char *head_ptr;
static void *virt_addr;
static struct semaphore sem;  /* mutual exclusion semaphore for race
                                on file open  */
static wait_queue_head_t queue; /* wait queue used by driver for
                                        blocking I/O */
```

DEVTEST.C: The C application to test out driver

```
#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>


int main(){

   int fd;                        /* file descriptor */


   char * input_val;

   input_val = (char *)malloc(200*sizeof(char));


   fd = open("/dev/ir_demod", O_RDWR);


   /*handle error opening file */

   if(fd == -1){

          printf("Failed to open device file!\n");

          return -1;

   }

   while(1){
```

```
        if(read(fd, input_val, 2) != 0){
                printf("Received value = 0x%x%x\n", *(input_val+1),*(input_val));
        }
    }
    close(fd);
    return 0;

}
```

Makefile

```
obj-m += irq_demod.o


all:
	make -C /home/grads/t/tanmay2592/ECEN749/lab_5/linux-3.14 M=$(PWD) modules


clean:
	make -C /home/grads/t/tanmay2592/ECEN749/lab_5/linux-3.14 M=$(PWD) clean
```

PICOCOM OUTPUT

The First Part :

Message register reg0 = 0x490, Count register reg1 = 0x6E reg2 = 0x1
Message register reg0 = 0x490, Count register reg1 = 0x6F reg2 = 0x1
Message register reg0 = 0x490, Count register reg1 = 0x70 reg2 = 0x1
Message register reg0 = 0x490, Count register reg1 = 0x71 reg2 = 0x1
Message register reg0 = 0x490, Count register reg1 = 0x72 reg2 = 0x1
Message register reg0 = 0x490, Count register reg1 = 0x73 reg2 = 0x1
Message register reg0 = 0x490, Count register reg1 = 0x74 reg2 = 0x1
Message register reg0 = 0x490, Count register reg1 = 0x75 reg2 = 0x1
Message register reg0 = 0x490, Count register reg1 = 0x76 reg2 = 0x1
Message register reg0 = 0x10, Count register reg1 = 0x77 reg2 = 0x1
Message register reg0 = 0x10, Count register reg1 = 0x78 reg2 = 0x1
Message register reg0 = 0x10, Count register reg1 = 0x79 reg2 = 0x1
Message register reg0 = 0x10, Count register reg1 = 0x7A reg2 = 0x1
Message register reg0 = 0x10, Count register reg1 = 0x7B reg2 = 0x1
Message register reg0 = 0x10, Count register reg1 = 0x7C reg2 = 0x1
Message register reg0 = 0x10, Count register reg1 = 0x7D reg2 = 0x1
Message register reg0 = 0x10, Count register reg1 = 0x7E reg2 = 0x1
Message register reg0 = 0x10, Count register reg1 = 0x7F reg2 = 0x1
Message register reg0 = 0x10, Count register reg1 = 0x80 reg2 = 0x1
Message register reg0 = 0x10, Count register reg1 = 0x81 reg2 = 0x1
Message register reg0 = 0x10, Count register reg1 = 0x82 reg2 = 0x1
Message register reg0 = 0x10, Count register reg1 = 0x83 reg2 = 0x1
Message register reg0 = 0x10, Count register reg1 = 0x84 reg2 = 0x1
Message register reg0 = 0x10, Count register reg1 = 0x85 reg2 = 0x1
Message register reg0 = 0x10, Count register reg1 = 0x86 reg2 = 0x1
Message register reg0 = 0x10, Count register reg1 = 0x87 reg2 = 0x1
Message register reg0 = 0x10, Count register reg1 = 0x88 reg2 = 0x1
Message register reg0 = 0x10, Count register reg1 = 0x89 reg2 = 0x1
Message register reg0 = 0x10, Count register reg1 = 0x8A reg2 = 0x1
Message register reg0 = 0x10, Count register reg1 = 0x8B reg2 = 0x1
Message register reg0 = 0x490, Count register reg1 = 0x8C reg2 = 0x1
Message register reg0 = 0x490, Count register reg1 = 0x8D reg2 = 0x1
Message register reg0 = 0x490, Count register reg1 = 0x8E reg2 = 0x1
Message register reg0 = 0x490, Count register reg1 = 0x8F reg2 = 0x1
Message register reg0 = 0xC90, Count register reg1 = 0x90 reg2 = 0x1
Message register reg0 = 0xC90, Count register reg1 = 0x91 reg2 = 0x1
Message register reg0 = 0xC90, Count register reg1 = 0x92 reg2 = 0x1

Part B

zynq> insmod ir_demod.ko
Mapping virtual address…
43c00000 Physical address of the ir_demod peripheral is mapped to 608e0000
Registered a device with dynamic Major number of 245
Create a device file for this device with this command:
'mknod /dev/ir_demod c 245 0'.
zynq> mknod /dev/ir_demod c 245 0
zynq> ./devtest
IRQ Num 61 called, interrupts processed 1 times
IRQ Num 61 called, interrupts processed 2 times
Received value = 0x490
Received value = 0x490
IRQ Num 61 called, interrupts processed 3 times
Received value = 0x490
IRQ Num 61 called, interrupts processed 4 times
IRQ Num 61 called, interrupts processed 5 times
Received value = 0x490
IRQ Num 61 called, interrupts processed 6 times
Received value = 0x490
Received value = 0x490

QUESTIONS

5. [4 points.] Answers to the following questions:

(a) Contrast the use of an interrupt based device driver with the polling method used in the previous lab.

Unlike polling, when the device needs to continuously transfer data to User so that we can compare whether the value changed, interrupt sends the signal only when device has the valid set of new data. This minimizes the data transfer(bandwidth requirement) between PS and AXI device and also frees up CPU cycle to do any other meaningful task while there is no data reception.

(b) Are there any race conditions that your device driver does not address? If so, what are they and how would you fix them?

The data race might arise when the buffer (200 Bytes) gets completely filled and the next 2 Bytes are overwriting the previously stored but not read bytes. And at the same time user performs a read operation. Now there would be a race condition between the older value and the new incoming value.

We can use semaphores based locking mechanism to hold user read request when processor is serving interrupt. The priority of interrupt routine should be higher to avoid losing messages.

(c) If you register your interrupt handler as a 'fast' interrupt (i.e. with the SA INTERRUPT flag set), what precautions must you take when developing your interrupt handler routine? Why is this so? Taking this into consideration, what modifications would you make to your existing IR-remote device driver?

By definition, a 'fast interrupt' can not be interrupted by other interrupts in the course of their execution. Using fast interrupts should be done only for the performance sensitive tasks as otherwise the system might drop other essential interrupts. So, my interrupt should be designed in such a way to take the least number of clock cycles for its execution.

As part of the modification, I will remove lines putting additional information like and
-->IRQ Num 61 called, interrupts processed 3 times
just copy the contents of slv_reg2 to the kernel memory. This will reduce the number of clock cycles considerably. Also in my current implementation I am reading the buffer byte by byte(character driver). I can modify the code to do an ioread16 and maintain a uint16_t queue.

(d) What would happen if you specified an incorrect IRQ number when registering your interrupt handler? Would your system still function properly? Why or why not?

The IRQ number refers to the interrupt id of our peripheral systems. If we specify a wrong id then the interrupt handler routine would not be executed when there is an interrupt in our concerned signal. This will happen because it isn't associated with the signal. Upon the reception of signal, the kernel looks for the matching handler. And also as a matter of fact, if it matches with any other interrupt id, there will be two handlers for same interrupt. This is definitely an unintended behaviour.