Kyle Loyka
Lab 8 Report
ECEN 449-503

Due: 11 April 2016

Introduction
The purpose of this lab was to implement an interrupt signal for the IR_demod circuit. This interrupt signal would notify the user when a new signal has been detected. Then a device driver was written for this peripheral.

Procedure
In the first part of the lab, an interrupt signal was added to the IR_demod peripheral. slv_reg2 was divided in half. The first half was for hardware writing, and the lower half was for software writing. With this implementation it was possible for hardware and software to communicate. The software could notify the hardware when it had read the new message.

In the second part of the lab, a Linux operating system was prepared to support the IR_demod peripheral.

In the third part of the lab, a device driver was written to support the specifications outlined in the lab manual.

Results / Q&A
The lab worked as expected.

- **(a)** The benefit of an interrupt signal is that the user doesn't have to continuously monitor the state of the peripheral. Instead, the user receives a signal only when there is new information to be read. This minimizes 'busy wait' time, where the process is waiting for new input. The implementation of an interrupt signal allows the process to do other tasks while it is waiting for new input.

- **(b)** One possible race condition is the queue may receive a new signal while at the same time a user is reading from the queue. In this case, writing to the queue should take precedence. After the writing is complete, the user can read from the queue again. This can be implemented using read-write-locks or semaphores.

- **(c)** When registering an interrupt handler as a 'fast' handler, you must be cautions with how much computing power/time your handler takes. The 'fast' designation is intended for handlers that can resolve interrupts quickly. 'Slow' interrupts takes more computing time and power. With a 'slow' handler, interrupts could be re-enabled while the 'slow' handler is handling it's signal. 'Fast' interrupts are also executed with all other interrupts disabled.

  When developing the interrupt for this lab, it is important to keep in mind the computation time for handling the interrupt, as well as the importance of the driver. If the IR_demod interrupt signal is critical for a system, then it should be implemented as a 'fast' interrupt.

- **(d)** If an incorrect IRQ number is specified, the driver will not work as intended. The interrupt handler may not be registered if the given interrupt line doesn't allow sharing and is already in use. If the interrupt line is open or can be shared, it is possible that the IR_demod interrupt may trigger other interrupt handlers. In this case, both affected drivers may behave unpredictably and cause system instability.

Conclusion

This lab provided valuable experience in designing and implementing hardware interrupts. This lab also expanded our skills in writing device drivers and supporting interrupt signals.

```
module user_logic
(
  // -- ADD USER PORTS BELOW THIS LINE ---------------
  IR_signal,
  Interrupt,
  // -- ADD USER PORTS ABOVE THIS LINE ---------------

  // -- DO NOT EDIT BELOW THIS LINE ------------------
  // -- Bus protocol ports, do not add to or delete
  Bus2IP_Clk,                     // Bus to IP clock
  Bus2IP_Reset,                   // Bus to IP reset
  Bus2IP_Data,                    // Bus to IP data bus
  Bus2IP_BE,                      // Bus to IP byte enables
  Bus2IP_RdCE,                    // Bus to IP read chip enable
  Bus2IP_WrCE,                    // Bus to IP write chip enable
  IP2Bus_Data,                    // IP to Bus data bus
  IP2Bus_RdAck,                   // IP to Bus read transfer acknowledgement
  IP2Bus_WrAck,                   // IP to Bus write transfer acknowledgement
  IP2Bus_Error                    // IP to Bus error response
  // -- DO NOT EDIT ABOVE THIS LINE ------------------
); // user_logic

// -- ADD USER PARAMETERS BELOW THIS LINE ------------
// --USER parameters added here
// -- ADD USER PARAMETERS ABOVE THIS LINE ------------

// -- DO NOT EDIT BELOW THIS LINE --------------------
// -- Bus protocol parameters, do not add to or delete
parameter C_SLV_DWIDTH                   = 32;
parameter C_NUM_REG                      = 3;
// -- DO NOT EDIT ABOVE THIS LINE --------------------

// -- ADD USER PORTS BELOW THIS LINE -----------------
input IR_signal;
output Interrupt;
// -- ADD USER PORTS ABOVE THIS LINE -----------------

// -- DO NOT EDIT BELOW THIS LINE --------------------
// -- Bus protocol ports, do not add to or delete
input                                     Bus2IP_Clk;
input                                     Bus2IP_Reset;
input      [0 : C_SLV_DWIDTH-1]           Bus2IP_Data;
input      [0 : C_SLV_DWIDTH/8-1]         Bus2IP_BE;
input      [0 : C_NUM_REG-1]              Bus2IP_RdCE;
input      [0 : C_NUM_REG-1]              Bus2IP_WrCE;
output     [0 : C_SLV_DWIDTH-1]           IP2Bus_Data;
output                                    IP2Bus_RdAck;
output                                    IP2Bus_WrAck;
output                                    IP2Bus_Error;
// -- DO NOT EDIT ABOVE THIS LINE --------------------

//----------------------------------------------------------------------------
// Implementation
//----------------------------------------------------------------------------

  // --USER nets declarations added here, as needed for user logic
```

```
// Nets for user logic slave model s/w accessible register example
reg       [0 : C_SLV_DWIDTH-1]          slv_reg0;
reg       [0 : C_SLV_DWIDTH-1]          slv_reg1;
reg       [0 : C_SLV_DWIDTH-1]          slv_reg2;
wire      [0 : 2]                       slv_reg_write_sel;
wire      [0 : 2]                       slv_reg_read_sel;
reg       [0 : C_SLV_DWIDTH-1]          slv_ip2bus_data;
wire                                    slv_read_ack;
wire                                    slv_write_ack;
integer                                 byte_index, bit_index;

// --USER logic implementation added here

// ------------------------------------------------------
// Example code to read/write user logic slave model s/w accessible registers
//
// Note:
// The example code presented here is to show you one way of reading/writing
// software accessible registers implemented in the user logic slave model.
// Each bit of the Bus2IP_WrCE/Bus2IP_RdCE signals is configured to correspond
// to one software accessible register by the top level template. For example,
// if you have four 32 bit software accessible registers in the user logic,
// you are basically operating on the following memory mapped registers:
//
//    Bus2IP_WrCE/Bus2IP_RdCE   Memory Mapped Register
//                     "1000"   C_BASEADDR + 0x0
//                     "0100"   C_BASEADDR + 0x4
//                     "0010"   C_BASEADDR + 0x8
//                     "0001"   C_BASEADDR + 0xC
//
// ------------------------------------------------------
reg flag = 0;


assign
  slv_reg_write_sel = Bus2IP_WrCE[0:2],
  slv_reg_read_sel  = Bus2IP_RdCE[0:2],
  slv_write_ack     = Bus2IP_WrCE[0] || Bus2IP_WrCE[1] || Bus2IP_WrCE[2],
  slv_read_ack      = Bus2IP_RdCE[0] || Bus2IP_RdCE[1] || Bus2IP_RdCE[2];

// implement slave model register read mux
always @( slv_reg_read_sel or slv_reg0 or slv_reg1 or slv_reg2 )
  begin: SLAVE_REG_READ_PROC

    case ( slv_reg_read_sel )
      3'b100 : slv_ip2bus_data <= slv_reg0;
      3'b010 : slv_ip2bus_data <= slv_reg1;
      3'b001 : slv_ip2bus_data <= slv_reg2;
      default : slv_ip2bus_data <= 0;
    endcase

  end // SLAVE_REG_READ_PROC

always @( posedge Bus2IP_Clk )
  begin: SLAVE_REG_WRITE_PROC
```

```
    if ( Bus2IP_Reset == 1 || flag == 1)
      begin
        //slv_reg0 <= 0;
        //slv_reg1 <= 0;
        slv_reg2[16:31] <= 0;
      end
    else
      case ( slv_reg_write_sel )
        3'b001 :
          for ( byte_index = 2; byte_index <= 3; byte_index = byte_index+1 )
            if ( Bus2IP_BE[byte_index] == 1 )
              for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index = bit_index+1 )
                slv_reg2[bit_index] <= Bus2IP_Data[bit_index];
        default : ;
      endcase

  end // SLAVE_REG_WRITE_PROC


// ------------------------------------------------------------
// IR_DEMOD HARDWARE LOGIC

// slv_reg0 holds latest demodulated message
// slv_reg1 holds running count of # of messages recieved **TA won't check this value
// slv_reg2 debugging

reg [17:0] counter = 0;     // clock cycle counter to determine IR signal type: Start, 1, or 0
reg [0:11] msg = 12'b0;
reg [0:11] prev_msg = 12'b1;        // message repeat signal on initial start
reg [3:0] index = 0;
reg enableCounting = 0;
reg state = 0;
reg startFlag = 0;
reg [9:0] slowClkCounter = 0;
reg slowClk = 0;
reg buffer = 0;


always@(posedge Bus2IP_Clk) begin
	// this will create a posedge clock signal.
	slowClkCounter <= slowClkCounter + 1;
	if (slowClkCounter == 1000) begin
		slowClk <= 1;
		slowClkCounter <= 0;
	end
	else slowClk <= 0;
end

assign Interrupt = slv_reg2[0];


/////////////////////////////////
always@(posedge slowClk) begin

	buffer <= IR_signal;
```

```
        // this is to check if the software has recieved the interrupt notification
        // software modifies slv_reg2[31], hardware modifies slv_reg2[0]
        // slv_reg2          (hardware) MSB: 0---------------|---------------31 :LSB (software)
        if (slv_reg2[31]) begin
                flag <= 1;
                slv_reg2[0] <= 0;
        end
        else flag <= 0;


        if (buffer && !IR_signal) enableCounting <= 1;              //negedge
        else if (!buffer && IR_signal) begin                        //posedge
                // writing logic
                enableCounting <= 0;
                counter <= 0;
                index <= index + 1;
                if (startFlag) begin
                        if (index < 12) begin
                                if (index != 0) msg[index-1] <= state;
                        end
                        else begin
                                slv_reg0 <= msg;
                                slv_reg2[0] <= 1;
                                index <= 0;
                                startFlag <= 0;
                        end
                end
        end
        // counting logic
        if(enableCounting && !IR_signal) begin
                counter <= counter + 1;
                if( (counter < 59) && (counter > 29) ) begin
                        state <= 0;
                end
                else if( (counter < 104) && (counter > 74) ) begin
                        state <= 1;
                end
                else if( (counter < 300) && (counter > 150) ) begin
                        startFlag <= 1;
                        index <= 0;
                end

        end
    end

    // ------------------------------------------------------------
    // Example code to drive IP to Bus signals
    // ------------------------------------------------------------

    assign IP2Bus_Data    = slv_ip2bus_data;
    assign IP2Bus_WrAck   = slv_write_ack;
    assign IP2Bus_RdAck   = slv_read_ack;
    assign IP2Bus_Error   = 0;

endmodule
```

devtest.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <linux/unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define DEVICE_FILE "/dev/irq_test"
#define WAIT_VAL 0x0100000/2 // iterations to wait in delay
int main()
{
	printf("Beginning DEV TEST\n");

	int fd;
	printf("about to open file \n");
      fd = open(DEVICE_FILE, O_RDWR);
	printf("finished opening file \n");


	if(fd == -1) {
		printf("Device open error\n");
		return -1;
	}
	char* buffer=(char*)malloc(2*10);//each message is only 2 bytes
	char* buffer_1 = (char*) malloc(20); //second buffer
	memset(buffer,'0',20);
	//memset(buffer_1,'0',20);
	int message;
	char* msg_ptr=(char*)&message;//char buffer for copy of int
	int i = 0;
	while (i < 200)
	{

		delay();
		printf("about to read \n");
		read(fd, (char *) buffer,4, 0);
		//read(fd, (char *) buffer_1, 2, 0);
		printf("finished reading \n");
		printf("The message was: %x\n", *buffer);
		memset(buffer,0,20);
		//memset(buffer_1,'0',20);
		++i;

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

```
int delay (void)
{
	volatile int delay_count=0; // volatile prevents compiler optimization
	while(delay_count<WAIT_VAL)
		delay_count++;// each iteration of while loop is 2 clock cycles
	return(0);
}
```

irq_test.h

```
/* 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 <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 <asm/io.h>
#include <linux/slab.h>
/* Some defines */
#define DEVICE_NAME "irq_test"
#define BUF_LEN 80
#define IRQ_NUM 3

/* 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);
void mem_write (int value);
char mem_read();


/*
 * 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 *msg_Ptr;
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 */
```

irq_test.c

```
/*  irq_test.c - Simple character device module
 *
 *  Demonstrates interrupt driven character device.  Note: Assumption
 *  here is some hardware will strobe a given hard coded IRQ number
 *  (200 in this case).  This hardware is not implemented, hence reads
 *  will block forever, consider this a non-working example.  Could be
 *  tied to some device to make it work as expected.
 *
 *  (Adapted from various example modules including those found in the
 *  Linux Kernel Programming Guide, Linux Device Drivers book and
 *  FSM's device driver tutorial)
 */

/* Moved all prototypes and includes into the headerfile */
#include "irq_test.h"
#include "xparameters.h"
#define MEMSIZE XPAR_IR_DEMOD_0_HIGHADDR - XPAR_IR_DEMOD_0_BASEADDR + 1
#define PHY_ADDR XPAR_IR_DEMOD_0_BASEADDR

/* 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 */


char* mem;
void* virt_addr;
static int message = 0;
int m_index = 0;
int c_index = 0;
static short int r_index = 0;
static short int w_index = 0;
char* r_queue;
static struct file_operations fops = {
  .read = device_read,
  .write = device_write,
  .open = device_open,
  .release = device_release
};

////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
//memory functions
            //
////////////////////////////////////////////////////////////////////////////////////////////////////

void mem_write (int value)
{
            if ((m_index + 1) == 100)
            {
                        m_index = 0;
                        if (c_index == 0)
                        {
                                    ++c_index;
                        }
            }
            mem[m_index] = value;
            ++m_index;
}

char mem_read()
{
            char temp;
            temp = mem[c_index];
            ++c_index;
            if (c_index == 100)
            {
                        c_index = 0;
                        if (m_index == 0)
                        {
                                    ++m_index;
                        }
            }
            if (c_index == m_index)
            {
                        c_index = 0;
                        m_index = 0;
            }
            return temp;
}



/*
 * This function is called when the module is loaded and registers a
 * device for the driver to use.
 */
int my_init(void)
{

  //Virtual Memory Declaration
  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);
    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);
}


/*
 * 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)
{
  printk(KERN_INFO "> starting device_open.\n");
  int irq_ret;
  if (down_interruptible (&sem))
            return -ERESTARTSYS;
   //Initialize memorymsg = ioread16(virt_addr + 2);
  //mem_write((char) msg);
  //printk(KERN_INFO "attempting malloc...\n");
  //mem = kmalloc(200, GFP_KERNEL);      //16*100 = 1600bits => 200bytes
  //printk(KERN_INFO "malloc complete.\n");


  /* We are only allowing one process to hold the device file open at
     a time. */
  if (Device_Open){
    up(&sem);
    return -EBUSY;
  }
  Device_Open++;
  //wait_event_interruptible(queue, (msg_Ptr != NULL));
   /* OK we are now past the critical section, we can release the
     semaphore and all will be well */
```

```
  up(&sem);
  printk(KERN_INFO "> passed the critical section.\n");
  r_queue = kmalloc(200, GFP_KERNEL);
  /* 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. */
  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)
{
  Device_Open--;              /* We're now ready for our next caller */
  kfree(mem);
  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)
{
  printk(KERN_INFO "reading from device\n");
  int bytes_read = 0;
  int result;

  while(length > 0) {
        result = put_user(r_queue[r_index*2], buffer++);
        result = put_user(r_queue[r_index*2 + 1],buffer++);
        if (result < 0){
                printk(KERN_ALERT "read error\n");
                break;
        }
```

```
            length = length - 2;
            r_index++;
            if(r_index == w_index){
                        r_index = 0;
                        w_index = 0;
                        if(length > 0)
                                    return bytes_read;
            }
  }

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

  printk(KERN_INFO "end of 'device_read'\n");
  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) {
  r_queue[w_index*2] = ioread8(virt_addr+2);
  r_queue[w_index*2 + 1] = ioread8(virt_addr+3);
  if(w_index < 99)
            ++w_index;
  iowrite16(0x01,virt_addr+10);
  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);
```