ECEN 749- Microprocessor System Design

Section 603

TA: Mr. Kunal Bharathi

LAB 8

Interrupt-Based IR-remote Device Driver

Date of Performance: 10/31/2019

Student Name: Dhiraj Dinesh Kudva

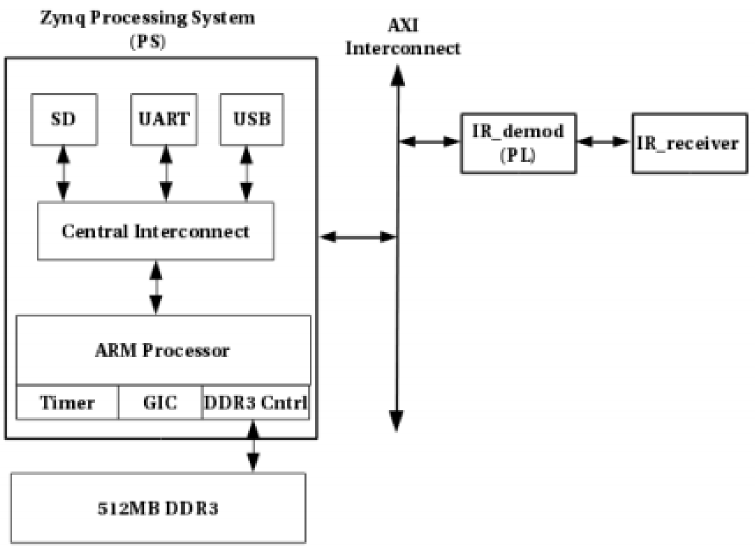
UIN: 829009538

Introduction

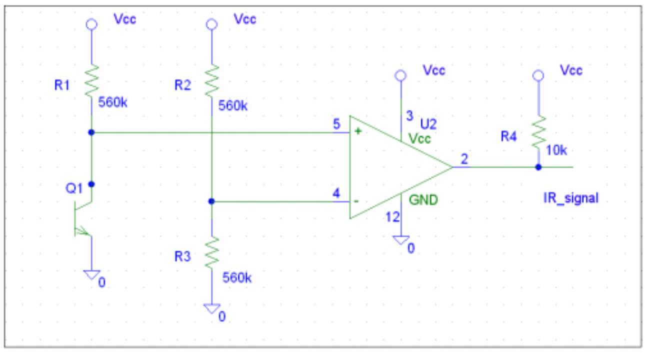
The purpose of this lab to understand use of interrupts by implementing an interrupt driven circuit. This is done by adding an interrupt signal in the IR demodulator circuit and using that to provide interrupts to drive the circuit.The demodulation hardware is to be designed using FPGA and Verilog and the demodulated message must be software accessible (to show on Picocom console).

Design:

The below diagram represents the overall IR Demodulator to be designed in this lab. The Zynq processing system and IR demodulator will be configured on Vivado, while the IR receiver circuit would be hardware connected on the breadboard. The signal received form the breadboard will be decoded and shown on the terminal output. Demodulation logic is written inside the IR demodulator IP peripheral that is connected to the Zynq Processing System.



The connection for IR receiver signal is similar like to that done in LAB 7. The output from the hardware is checked on the Oscilloscope. (Vcc=+3.3 V.)



This lab an interrupt signal would also be created, so that the central processing unit need not continuously poll the input signal, but only process when an interrupt is ready.

Part 1: Adding interrupt signal in the Design Block

1. In this lab, as per the design of the IR receiver circuit shown above, the components must be connected on breadboard. Check the output (similar to LAB7)
2. Copy the lab 7 files in lab 8 directory. Update the IP repository for the IR demodulator so that the address is of the lab 8 directory.
3. Open "Edit in IP packager" by clicking on the IR demodulator, then open the axi.v file.
4. As the interrupt method is to be implemented instead of polling, an additional Interrupt signal is to be added. This is an output signal.
5. This interrupt signal is mapped to slv_reg2. The lower half of the bit is used to indicate the arrival of an interrupt (for control) and is writeable by the Verilog code. The upper half of the 32 bit slv_reg2 (for status) would be used by the user application to indicate that message is received and read.
6. Initially in order to test the interrupt, keep the signal as external and connect it to W15 pin as per the lab manual.
7. The logic must be added in the Verilog code indicating that when the signal gets high (after receiving the 12 bit message from the remote).
8. Update the ports and interfaces tab and merge changes.
9. Select the IR_interrupt under the ports and select interrupt under the 'Auto Infer Single Bit Interface'. Click on re-package IP.
10. Check the output of IR_interrupt and IR_signal on the oscilloscope.
11. Use picocom to view the output based on the modified C code to display the output.

<u>Part 2: Plugging the interrupt as a soft signal.</u>

1. In this part, the Interrupt signal must be connected to the PS.
2. Edit the PS IP and select 'Interrupts', check 'Fabric Interrupts', expand 'Fabric Interrupts', expand'PL-PS Interrupt Ports', check 'IRQ F2P[15:0]' and click 'OK'. Note: 61 is the interrupt id of the 'ir demod' IP.
3. Remove the external 'IR interrupt' signal port and connect the 'IR interrupt' on 'ir demod 0' to 'IRQ F2P[0:0]' of the Zynq PS IP using a wire. This will connect the interrupt signal to the PS.
4. Implement the device driver including this interrupt signal. The buffer must have capacity to store 100 messages. Semaphore condition must also be implemented to ensure that only a single process will opens the device file.
5. Create BOOT.bin file and update the device tree (.dts file) as per the IR demod and then update the .dtb file.
6. Create the devtest application to test the operation of device driver.

Program:

<u>Verilog Code (IR_demod):</u>

```
`timescale 1 ns / 1 ps

     module ir_demod_v1_0_S00_AXI #
     (
          // Users to add parameters here

          // User parameters ends
          // Do not modify the parameters beyond this line

          // Width of S_AXI data bus
          parameter integer C_S_AXI_DATA_WIDTH  = 32,
          // Width of S_AXI address bus
          parameter integer C_S_AXI_ADDR_WIDTH  = 4
     )
     (
          // Users to add ports here
        output wire IR_interrupt,
  input wire IR_signal,
          // User ports ends
          // Do not modify the ports beyond this line

          // Global Clock Signal
          input wire  S_AXI_ACLK,
          // Global Reset Signal. This Signal is Active LOW
          input wire  S_AXI_ARESETN,
          // Write address (issued by master, acceped by Slave)
          input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_AWADDR,
          // Write channel Protection type. This signal
indicates the
          // privilege and security level of the transaction,
and whether
          // the transaction is a data access or an instruction
access.
          input wire [2 : 0] S_AXI_AWPROT,
          // Write address valid. This signal indicates that
the master signaling
          // valid write address and control information.
          input wire  S_AXI_AWVALID,
          // Write address ready. This signal indicates that
the slave is ready
          // to accept an address and associated control
signals.
          output wire  S_AXI_AWREADY,
          // Write data (issued by master, acceped by Slave)
          input wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_WDATA,
		// Write strobes. This signal indicates which byte 
lanes hold
		// valid data. There is one write strobe bit for each 
eight
		// bits of the write data bus.    
		input wire [(C_S_AXI_DATA_WIDTH/8)-1 : 0] 
S_AXI_WSTRB,
		// Write valid. This signal indicates that valid 
write
		// data and strobes are available.
		input wire  S_AXI_WVALID,
		// Write ready. This signal indicates that the slave
		// can accept the write data.
		output wire  S_AXI_WREADY,
		// Write response. This signal indicates the status
		// of the write transaction.
		output wire [1 : 0] S_AXI_BRESP,
		// Write response valid. This signal indicates that 
the channel
		// is signaling a valid write response.
		output wire  S_AXI_BVALID,
		// Response ready. This signal indicates that the 
master
		// can accept a write response.
		input wire  S_AXI_BREADY,
		// Read address (issued by master, acceped by Slave)
		input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_ARADDR,
		// Protection type. This signal indicates the 
privilege
		// and security level of the transaction, and whether 
the
		// transaction is a data access or an instruction 
access.
		input wire [2 : 0] S_AXI_ARPROT,
		// Read address valid. This signal indicates that the 
channel
		// is signaling valid read address and control 
information.
		input wire  S_AXI_ARVALID,
		// Read address ready. This signal indicates that the 
slave is
		// ready to accept an address and associated control 
signals.
		output wire  S_AXI_ARREADY,
		// Read data (issued by slave)
		output wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_RDATA,
		// Read response. This signal indicates the status of 
the
		// read transfer.
		output wire [1 : 0] S_AXI_RRESP,
		// Read valid. This signal indicates that the channel
is
		// signaling the required read data.
		output wire  S_AXI_RVALID,
		// Read ready. This signal indicates that the master
can
		// accept the read data and response information.
		input wire  S_AXI_RREADY
	);

	// AXI4LITE signals
	reg [C_S_AXI_ADDR_WIDTH-1 : 0] 	axi_awaddr;
	reg  	axi_awready;
	reg  	axi_wready;
	reg [1 : 0] 	axi_bresp;
	reg  	axi_bvalid;
	reg [C_S_AXI_ADDR_WIDTH-1 : 0] 	axi_araddr;
	reg  	axi_arready;
	reg [C_S_AXI_DATA_WIDTH-1 : 0] 	axi_rdata;
	reg [1 : 0] 	axi_rresp;
	reg  	axi_rvalid;

	// Example-specific design signals
	// local parameter for addressing 32 bit / 64 bit
C_S_AXI_DATA_WIDTH
	// ADDR_LSB is used for addressing 32/64 bit
registers/memories
	// ADDR_LSB = 2 for 32 bits (n downto 2)
	// ADDR_LSB = 3 for 64 bits (n downto 3)
	localparam integer ADDR_LSB = (C_S_AXI_DATA_WIDTH/32) + 1;
	localparam integer OPT_MEM_ADDR_BITS = 1;
	//----------------------------------------------
	//-- Signals for user logic register space example
	//------------------------------------------------
	//-- Number of Slave Registers 4
	reg [C_S_AXI_DATA_WIDTH-1:0]	slv_reg0;
	reg [C_S_AXI_DATA_WIDTH-1:0]	slv_reg1;
	reg [C_S_AXI_DATA_WIDTH-1:0]	slv_reg2;
	reg [C_S_AXI_DATA_WIDTH-1:0]	slv_reg3;
	wire	 slv_reg_rden;
	wire	 slv_reg_wren;
	reg [C_S_AXI_DATA_WIDTH-1:0]	 reg_data_out;
	integer	 byte_index;

	// I/O Connections assignments

	assign S_AXI_AWREADY	= axi_awready;
	assign S_AXI_WREADY	= axi_wready;
	assign S_AXI_BRESP	= axi_bresp;
	assign S_AXI_BVALID	= axi_bvalid;
	assign S_AXI_ARREADY	= axi_arready;
```

```
      assign S_AXI_RDATA    = axi_rdata;
      assign S_AXI_RRESP    = axi_rresp;
      assign S_AXI_RVALID   = axi_rvalid;
      // Implement axi_awready generation
      // axi_awready is asserted for one S_AXI_ACLK clock cycle
when both
      // S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready
is
      // de-asserted when reset is low.

      always @( posedge S_AXI_ACLK )
      begin
        if ( S_AXI_ARESETN == 1'b0 )
          begin
            axi_awready <= 1'b0;
          end
        else
          begin
            if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID)
              begin
                // slave is ready to accept write address when
                // there is a valid write address and write data
                // on the write address and data bus. This design
                // expects no outstanding transactions.
                axi_awready <= 1'b1;
              end
            else
              begin
                axi_awready <= 1'b0;
              end
          end
      end

      // Implement axi_awaddr latching
      // This process is used to latch the address when both
      // S_AXI_AWVALID and S_AXI_WVALID are valid.

      always @( posedge S_AXI_ACLK )
      begin
        if ( S_AXI_ARESETN == 1'b0 )
          begin
            axi_awaddr <= 0;
          end
        else
          begin
            if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID)
              begin
                // Write Address latching
                axi_awaddr <= S_AXI_AWADDR;
              end
          end
      end

      // Implement axi_wready generation
      // axi_wready is asserted for one S_AXI_ACLK clock cycle
when both
      // S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_wready
is
      // de-asserted when reset is low.

      always @( posedge S_AXI_ACLK )
      begin
        if ( S_AXI_ARESETN == 1'b0 )
          begin
            axi_wready <= 1'b0;
          end
        else
          begin
            if (~axi_wready && S_AXI_WVALID && S_AXI_AWVALID)
              begin
                // slave is ready to accept write data when
                // there is a valid write address and write data
                // on the write address and data bus. This design
                // expects no outstanding transactions.
                axi_wready <= 1'b1;
              end
            else
              begin
                axi_wready <= 1'b0;
              end
          end
      end

      // Implement memory mapped register select and write logic
generation
      // The write data is accepted and written to memory mapped
registers when
      // axi_awready, S_AXI_WVALID, axi_wready and S_AXI_WVALID
are asserted. Write strobes are used to
      // select byte enables of slave registers while writing.
      // These registers are cleared when reset (active low) is
applied.
      // Slave register write enable is asserted when valid
address and data are available
      // and the slave is ready to accept the write address and
write data.
      assign slv_reg_wren = axi_wready && S_AXI_WVALID &&
axi_awready && S_AXI_AWVALID;
      reg instruction_reset;
      always @( posedge S_AXI_ACLK )
      begin
```

```
        if ( (S_AXI_ARESETN == 1'b0) || (instruction_reset==1'b1)
)
          begin
           // slv_reg0 <= 0;
           // slv_reg1 <= 0;
            slv_reg2[31:16] <= 0;
            //slv_reg3 <= 0;
          end
        else begin
          if (slv_reg_wren)
            begin
              case (
axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
                2'h0:
                  for ( byte_index = 0; byte_index <=
(C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
                    if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                      // Respective byte enables are asserted as
per write strobes
                      // Slave register 0
                      //slv_reg0[(byte_index*8) +: 8] <=
S_AXI_WDATA[(byte_index*8) +: 8];
                    end
                2'h1:
                  for ( byte_index = 0; byte_index <=
(C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
                    if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                      // Respective byte enables are asserted as
per write strobes
                      // Slave register 1
                      //slv_reg1[(byte_index*8) +: 8] <=
S_AXI_WDATA[(byte_index*8) +: 8];
                    end
                2'h2:
                  for ( byte_index = 2; byte_index <=
(C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
                    if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                      // Respective byte enables are asserted as
per write strobes
                      // Slave register 2
                     slv_reg2[(byte_index*8) +: 8] <=
S_AXI_WDATA[(byte_index*8) +: 8];
                    end
                2'h3:
                  for ( byte_index = 0; byte_index <=
(C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1 )
                    if ( S_AXI_WSTRB[byte_index] == 1 ) begin
                      // Respective byte enables are asserted as
per write strobes
                      // Slave register 3
```

```
                   //  slv_reg3[(byte_index*8) +: 8] <=
S_AXI_WDATA[(byte_index*8) +: 8];
                   end
               default : begin
                         //  slv_reg0 <= slv_reg0;
                         //  slv_reg1 <= slv_reg1;
                        //   slv_reg2 <= slv_reg2;
                         //  slv_reg3 <= slv_reg3;
                         end
             endcase
           end
       end
     end

     // Implement write response logic generation
     // The write response and response valid signals are
asserted by the slave
     // when axi_wready, S_AXI_WVALID, axi_wready and
S_AXI_WVALID are asserted.
     // This marks the acceptance of address and indicates the
status of
     // write transaction.

     always @( posedge S_AXI_ACLK )
     begin
       if ( S_AXI_ARESETN == 1'b0 )
         begin
           axi_bvalid  <= 0;
           axi_bresp   <= 2'b0;
         end
       else
         begin
           if (axi_awready && S_AXI_AWVALID && ~axi_bvalid &&
axi_wready && S_AXI_WVALID)
             begin
               // indicates a valid write response is available
               axi_bvalid <= 1'b1;
               axi_bresp  <= 2'b0; // 'OKAY' response
             end                   // work error responses in
future
           else
             begin
               if (S_AXI_BREADY && axi_bvalid)
                 //check if bready is asserted while bvalid is
high)
                 //(there is a possibility that bready is always
asserted high)
                 begin
                   axi_bvalid <= 1'b0;
                 end
             end
```

```
            end
        end

        // Implement axi_arready generation
        // axi_arready is asserted for one S_AXI_ACLK clock cycle
when
        // S_AXI_ARVALID is asserted. axi_awready is
        // de-asserted when reset (active low) is asserted.
        // The read address is also latched when S_AXI_ARVALID is
        // asserted. axi_araddr is reset to zero on reset
assertion.

        always @( posedge S_AXI_ACLK )
        begin
          if ( S_AXI_ARESETN == 1'b0 )
            begin
              axi_arready <= 1'b0;
              axi_araddr  <= 32'b0;
            end
          else
            begin
              if (~axi_arready && S_AXI_ARVALID)
                begin
                  // indicates that the slave has acceped the valid
read address
                  axi_arready <= 1'b1;
                  // Read address latching
                  axi_araddr  <= S_AXI_ARADDR;
                end
              else
                begin
                  axi_arready <= 1'b0;
                end
            end
        end

        // Implement axi_arvalid generation
        // axi_rvalid is asserted for one S_AXI_ACLK clock cycle
when both
        // S_AXI_ARVALID and axi_arready are asserted. The slave
registers
        // data are available on the axi_rdata bus at this
instance. The
        // assertion of axi_rvalid marks the validity of read data
on the
        // bus and axi_rresp indicates the status of read
transaction.axi_rvalid
        // is deasserted on reset (active low). axi_rresp and
axi_rdata are
        // cleared to zero on reset (active low).
        always @( posedge S_AXI_ACLK )
```

```
      begin
        if ( S_AXI_ARESETN == 1'b0 )
          begin
            axi_rvalid <= 0;
            axi_rresp  <= 0;
          end
        else
          begin
            if (axi_arready && S_AXI_ARVALID && ~axi_rvalid)
              begin
                // Valid read data is available at the read data
bus
                axi_rvalid <= 1'b1;
                axi_rresp  <= 2'b0; // 'OKAY' response
              end
            else if (axi_rvalid && S_AXI_RREADY)
              begin
                // Read data is accepted by the master
                axi_rvalid <= 1'b0;
              end
          end
      end

      // Implement memory mapped register select and read logic
generation
      // Slave register read enable is asserted when valid
address is available
      // and the slave is ready to accept the read address.
      assign slv_reg_rden = axi_arready & S_AXI_ARVALID &
~axi_rvalid;
      always @(*)
      begin
            // Address decoding for reading registers
            case (
axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
              2'h0   : reg_data_out <= slv_reg0;
              2'h1   : reg_data_out <= slv_reg1;
              2'h2   : reg_data_out <= slv_reg2;
              2'h3   : reg_data_out <= slv_reg3;
              default : reg_data_out <= 0;
            endcase
      end

      // Output register or memory read data
      always @( posedge S_AXI_ACLK )
      begin
        if ( S_AXI_ARESETN == 1'b0 )
          begin
            axi_rdata  <= 0;
          end
        else
```

```
          begin
            // When there is a valid read address (S_AXI_ARVALID)
with
            // acceptance of read address by the slave
(axi_arready),
            // output the read dada
            if (slv_reg_rden)
              begin
                axi_rdata <= reg_data_out;     // register read
data
              end
          end
      end

      // Add user logic here
reg [31:0] Main_clock_count; // Clock counter
    reg reduced_clock; // Device counter
    assign reset = ~S_AXI_ARESETN; // Active Low Reset
    assign mainClock = S_AXI_ACLK; // System clock
    // Modifying mainClock for Counting
    always@(posedge mainClock) begin
    if (Main_clock_count == 1000 && ~reset) begin
    reduced_clock <= 1;
    Main_clock_count <= 0;
    end
    else if (reset) begin
    Main_clock_count <= 0;
    reduced_clock <= 0;
    end
    else begin
    Main_clock_count <= Main_clock_count + 1;
    reduced_clock <= 0;
    end
    end
    reg prev_main_signal; // detect edge
    reg [31:0] msg_select; // msg selector start, 1 or zero
    reg State; // msg bit
    reg [11:0] final_msg; // 12 bit msg
    reg signal_start; // start signal flag
    reg [31:0] number_bits; // max 12 bits per msg
    reg flag_count; // flag to start counting

    assign IR_interrupt = slv_reg2[0]; //assiging the status of
interrupt to the flag/


    always@(posedge reduced_clock) begin

      if (slv_reg2[31])begin     //used to take the input from
the user space and clear the slv reg2
      slv_reg2[15:0]<=0;
      instruction_reset <= 1'b1;
      end
      else if (instruction_reset)begin
      instruction_reset <=1'b0;
      end

else begin
    prev_main_signal <= IR_signal;
    if (prev_main_signal && ~IR_signal) begin
    // neg edge found, set the flag to start counting of start
bit
    flag_count <= 1'b1;
    end
    else if (~prev_main_signal && IR_signal) begin // if pos
edge then 1 bit is counted

    number_bits <= number_bits + 1;
    flag_count <= 1'b0;
    msg_select <= 0;

    if (signal_start && number_bits >= 12) begin // once 12 bits
found, send it to the output
    slv_reg0 <= final_msg;
    slv_reg1 <= slv_reg1 + 1;
   slv_reg2[0] <=  1;//interrupt is 1 for new messages.
    number_bits <= 0;
    signal_start <= 0;
    end

    else if (signal_start && number_bits < 12 && number_bits !=
0) begin //if 12 bits not over, keeep reading the messages
    final_msg[11 - (number_bits - 1)] <= State;
    end
    end

    if (flag_count && ~IR_signal) begin
    msg_select <= msg_select + 1;
    // Start signal time of 2.3 ms, 0 signal time 0.6ms, 1
signal time 1.19ms.
    // Start signal low for 180 cycles
    // Zero Signal = 45 cycles
    // One Signal = 90 cycles

    if (msg_select >= 20 && msg_select <= 68) begin
    State <=0;
    end
    else if (msg_select >= 69 && msg_select <= 134) begin
    State <= 1;
    end
    else if (msg_select >= 135 && msg_select <= 250) begin
    signal_start <= 1;
    number_bits <= 0;
    end
    else begin
    State <= 0;
    end
    end
    end
end
     // User logic ends

endmodule
```

C Code (dev test):

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

int main() {
     printf("IR Remote Device started\n");
     unsigned int message;
     char* msg_Ptr = (char*)&message; //used to write 1 byte at a
time to message
     int fd; //file descriptor
     char input = 0;
     char* outputBuffer = (char*)malloc(200 * sizeof(char));
//enough to read all 100 messages because each message will be of
2 bytes

                              //open device file for reading and
writing
                              //user open to open 'dev/ir_demod'/
     fd = open("/dev/driver", O_RDWR);

     //handle error opening file
     if (fd == -1) {
          printf("Failed to open device file!\n");
          return -1;
     }
     int i = 0;
     for (;;) {
            //printf("Display messages since previous
iteration...\n\n");
            int bytesRead = read(fd, outputBuffer, 200); //read up
to 100 messages at a time

            for (i = 0; i < bytesRead / 2; i++) { //print all 100
messages
                  msg_Ptr[0] = outputBuffer[i * 2];
                  msg_Ptr[1] = outputBuffer[i * 2 + 1];
                  msg_Ptr[2] = 0;
                  msg_Ptr[3] = 0;
                  printf("IR Demodulated Message Received: 0x%x\n",
message);
                  if (message==0x490)
                  {printf("Button pressed is VOLUME + \n");}
                  else if (message==0xc90)
                  {printf("Button pressed is VOLUME - \n");}
                  else if (message==0x90)
                  {printf("Button pressed is CHANNEL + \n");}
                  else if (message==0x890)
                  {printf("Button pressed is CHANNEL - \n");}
                  else
                  {printf("New button except from predefined ones
\n");}

            }
            printf("\n");
            sleep(1); //sleep for () seconds
      }
      close(fd);
      free(outputBuffer);
      return 0;
}
```

C code (Character Driver)

```
#include "irq_test.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
};

void* virt_addr; //virtual address pointing to ir peripheral
```

```
                       /*
                       * This function is called when the module is
loaded and registers a
                       * device for the driver to use.
                       */
int my_init(void)
{
     printk(KERN_INFO "Mapping virutal address...\n");
     //map virtual address to multiplier physical address//use ioremap
     virt_addr = ioremap(PHY_ADDR, MEMSIZE);
     printk("Physical Address: 0x%x\n", PHY_ADDR);
     printk("Virtual Address: 0x%x\n", virt_addr);
     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\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);
     printk(KERN_ALERT "unmapping virtual address space...\n");
     iounmap((void*)virt_addr);
}
```

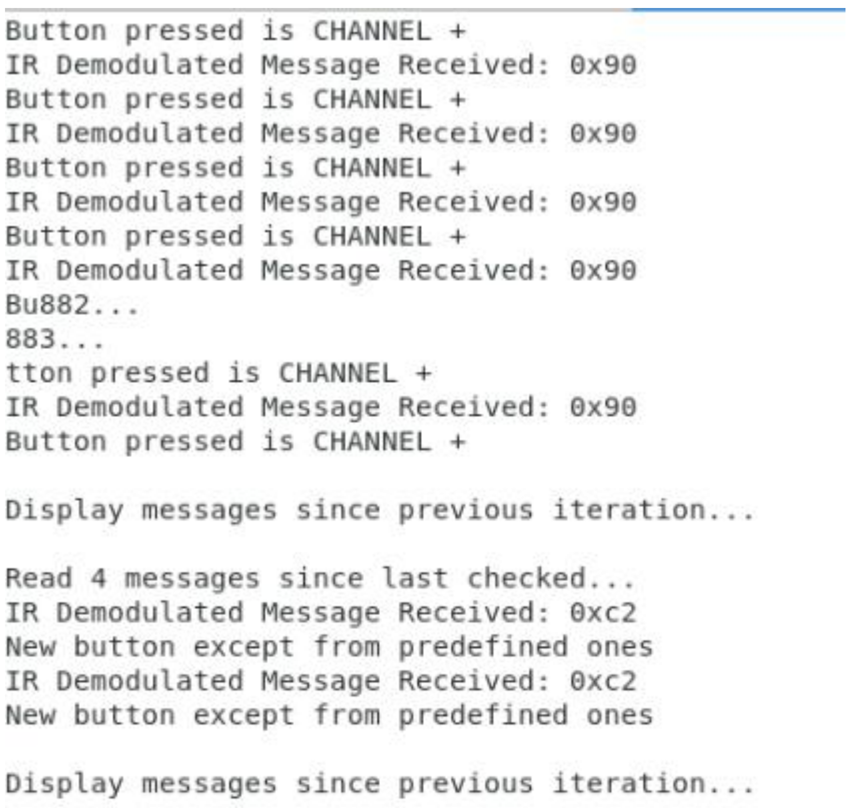
```
/*
* Called when a process tries to open the device file, like "cat
* /dev/irq_test".  Link to this function placed in file operations
* structure for our device file.
*/
static int device_open(struct inode *inode, struct file *file)
{
      int irq_ret;

      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);

      /* 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("Device has been opened\n");

      //allocating Head_message with enough bytes to store 100 of
MESSAGE_STRUC
      Head_message = (MESSAGE_STRUC*)kmalloc(100 *
sizeof(MESSAGE_STRUC), GFP_KERNEL);//MESSAGE_STRUC is a structure of
two characters. We need two characters as it can have multiple bytes
like 0x490 || passing the head pointer of that to Head_message
```

```
	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);
	printk("Device has been closed\n");
	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;

	/* In this driver msg_Ptr is NULL until an interrupt occurs */
	//wait_event_interruptible(queue, (msg_Ptr != NULL)); /* sleep
until

	//interrupted */

	/*

	* Actually put the data into the buffer

	*/
	int i = 0;
	//if we go past the amount of messages we've written
	/*if (length > counter * 2 || length > 200) {
		length = writeIndex * 2;
	}*/

	length = writeIndex * 2;
```

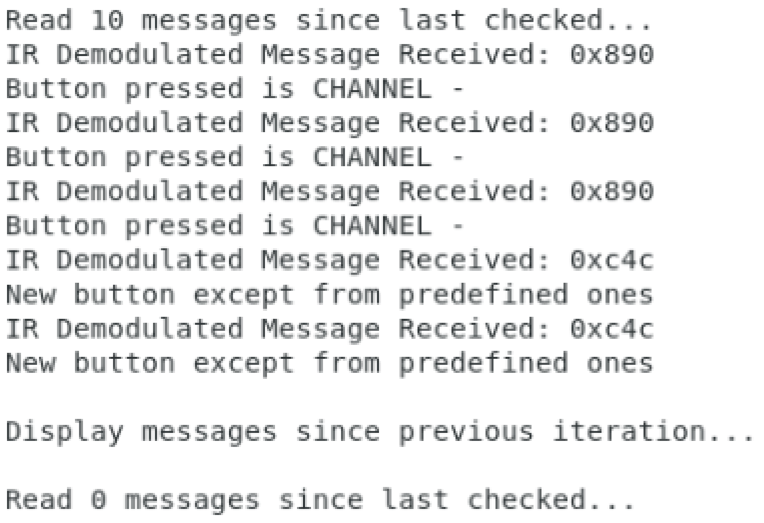
```
      printk("Read %d messages since last checked...\n", length);
      writeIndex = 0;

      msg_Ptr = (char*)Head_message;
      for (i = 0; i < length; i++) {
            /*
            * 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.
            */
            put_user(*(msg_Ptr++), buffer++); /* one char at a time...
*/
            bytes_read++;
      }

      /* completed interrupt servicing reset
      pointer to wait for another
      interrupt */
      msg_Ptr = NULL;

      /*
      * 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) {
      //sprintf(msg, "IRQ Num %d called, interrupts processed %d
times\n", irq, counter++);
      printk(msg, "IRQ Num %d called, interrupts processed %d times\n",
irq, counter++);
```

```
	//printk("%d...\n", counter);
	msg_Ptr = (char*)Head_message; //pointer array to the start of the
queue
	message = ioread32(virt_addr + 0);// getting the 32 bit
demodulated data from the verilog module

	if (writeIndex == 100) {//every 100 messages we send a wake signal
					//reset writeIndex when it
becomes large
					/* Just wake up anything waiting
					for the device */
		//wake_up_interruptible(&queue);
		writeIndex = 0;
	}

	//head_message[write-index].byte0 = (char *)(message);
	//head_message[write-index].byte0 = ((char *)(message)+1);
	Head_message[writeIndex].byte0 = byteBuff[0]; //write to the
message queue
	Head_message[writeIndex].byte1 = byteBuff[1];
	writeIndex++;
	iowrite32(0x80000000, virt_addr + 8); //clear the interrupt

	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);
```

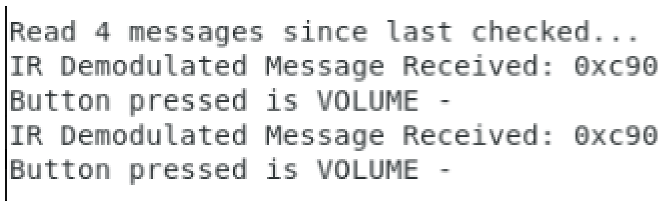
Output:



```
Button pressed is CHANNEL +
IR Demodulated Message Received: 0x90
Button pressed is CHANNEL +
IR Demodulated Message Received: 0x90
Button pressed is CHANNEL +
IR Demodulated Message Received: 0x90
Button pressed is CHANNEL +
IR Demodulated Message Received: 0x90
Bu882...
883...
tton pressed is CHANNEL +
IR Demodulated Message Received: 0x90
Button pressed is CHANNEL +

Display messages since previous iteration...

Read 4 messages since last checked...
IR Demodulated Message Received: 0xc2
New button except from predefined ones
IR Demodulated Message Received: 0xc2
New button except from predefined ones

Display messages since previous iteration...
```



```
Read 10 messages since last checked...
IR Demodulated Message Received: 0x890
Button pressed is CHANNEL -
IR Demodulated Message Received: 0x890
Button pressed is CHANNEL -
IR Demodulated Message Received: 0x890
Button pressed is CHANNEL -
IR Demodulated Message Received: 0xc4c
New button except from predefined ones
IR Demodulated Message Received: 0xc4c
New button except from predefined ones

Display messages since previous iteration...

Read 0 messages since last checked...
```



```
Read 4 messages since last checked...
IR Demodulated Message Received: 0xc90
Button pressed is VOLUME -
IR Demodulated Message Received: 0xc90
Button pressed is VOLUME -
```

Result:

The interrupt was configured and measured using the hardware created in LAB 7 and by just adding a single interrupt signal in the block design. This interrupt signal then was soft configured and connected to the PS of Zynq board and then the messages were read when the interrupt flag was high and not by polling the input IR signal continuously. This offloads the CPU from continuous amount of polling work and saves a lot of CPU time. Slv_reg2 was used to store the interrupt information. The demodulated signal was then displayed on the screen.

Conclusion:

This lab helped in understanding the concept of interrupts using IR demodulator and the use of semaphore to avoid problem of multiple processes sharing the same resource.

Questions:

1. **Contrast the use of an interrupt-based device driver with the polling method used in the previous lab**.

In the interrupt method, an additional signal for interrupts is used. As soon as the message is received by the IR modulator, interrupt is generated, and the CPU is interrupted. This causes the process to jump to interrupt service routine to handle the message event interrupt. This reduces the overhead but is not fast enough as it has a higher response time.
In the polling method (performed in lab 7), the CPU continuously monitors the state of the peripheral to check if a new message has arrived. This wastes a lot of CPU cycles as the CPU is busy polling and can't be used for different cycles. However, the response time is small, that is it can respond faster to any sudden changes in the signal than the interrupt method as the interrupt method first generates the interrupt and then sends it to the CPU.

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

There are multiple race conditions possible with the device driver. The first condition is when two different programs are trying to access the same device driver or resources. In this case it would be the program attempting to get access to the received data.
Another race condition that might occur is when new message arrives while user is still reading previous data from the queue. To fix such a case we need to use semaphores.

3. 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?

Fast Interrupt Requests (FIQs) are a specialized type of Interrupt Request. FIQs are specific to the ARM CPU architecture, which supports two types of interrupts; FIQs for fast, low latency interrupt handling and Interrupt Requests (IRQs), for more general interrupts. An FIQ takes

priority over an IRQ in an ARM system. Also, only one FIQ source at a time is supported. This helps reduce interrupt latency as the interrupt service routine can be executed directly without determining the source of the interrupt. A context save is not required for servicing an FIQ since it has its own set of banked registers. This reduces the overhead of context switching.
While developing a fast interrupt handler for this lab, we should make sure that the handler takes less computational power and time. This can be done by removing all printk statements and debugging variable storages.

4. 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?

If an incorrect IRQ is specified when registering the interrupt, the CPU would jump to that incorrect handler when the interrupt occurs. Since this ISR is not designed to handle the actual interrupt, it wouldn't clear the interrupt status bit. This would cause the peripheral to not accept new data (peripheral design dependent). Also, since we are now executing a handler for some other IRQ, we may write to some other peripheral which may also cause unexpected results. Thus, both IR device driver and other Interrupt which is triggered may behave unexpectedly and cause potential errors.