ECEN 468 Lab Report

Tong Lu
UIN 621007982

January 29, 2019

Introduction

In this lab, we added an interrupt signal to the IR-remote hardware from Lab 7 and created a Linux device driver for the IR-remote which utilizes this interrupt.

Procedure

1. Add the interrupt signal output into the ir_demod ip code (see Appendix), make the signal external, add U20 in constraint file (see Appendix) and regenerate bitstream.
2. Program FPGA again in SDK and observe the interrupt signal from oscilloscope.
3. Demonstrate the first part result to TA.
4. Examine irq_test.c, irq_test.h in /home/faculty/shared/ECEN449/449NeededFiles/module_examples.
5. Based on the example source code and the source code from lab 6, create the device driver ir_dev.c and ir_dev.h (see Appendix)
6. To implement the queue structure in device driver, I implemented circular array structure in msg_queue.h (see Appendix) and included it in ir_demod.h
7. Modify the makefile and cross compile:

```
make ARCH=arm CROSS COMPILE=arm-xilinx-linux-gnueabi-
```

8. Based on the devtest.c code from lab 6, create the devtest.c for lab 8 (see Appendix):
9. Cross compile:

```
arm-xilinx-linux-gnueabi-gcc -o devtest devtest.c
```

10. Enable interrupt feature in ZYNQ processor system and connect the IR_interrupt signal to the interrupt port on ZYNQ block.
11. Recreate the HDL wrapper and regenerate bitstream for the modified block design.
12. Create FSBL based on the new bitstream and regenerate Boot.bin.
13. Copy ir_demod.ko, devtest and Boot.bin files to the SD card and mount the SD card on FPGA:

```
mount /dev/mmcb1k0p1 /mnt/
```

14. load the module into Linux kernel on ZYBO board and run devtest:

```
insmod /mnt/ir_demod.ko
mknod /dev/ir_demod c 245 0
```

15. run the devtest executable file and demo the result to TA:

```
./mnt/devtest
```

Result

All the programs was finished and demonstrated to TA. The programs are working well and meet all the requirement on lab manual.

```
[lvtongtom305@lin04-424cvlb modules]$ picocom -b 115200 /dev/ttyUSB1
picocom v2.3a

port is        : /dev/ttyUSB1
flowcontrol    : none
baudrate is    : 115200
parity is      : none
databits are   : 8
stopbits are   : 1
escape is      : C-a
local echo is  : no
noinit is      : no
noreset is     : no
nolock is      : no
send_cmd is    : sz -vv
receive_cmd is : rz -vv -E
imap is        :
omap is        :
emap is        : crcrlf,delbs,
logfile is     : none

Type [C-a] [C-h] to see available commands

Terminal ready
 0
Device: zynq_sdhci
Manufacturer ID: 3
OEM: 5344
Name: SS08G
Tran Speed: 50000000
Rd Block Len: 512
SD version 3.0
High Capacity: Yes
Capacity: 7.4 GiB
Bus Width: 4-bit
reading uEnv.txt
** Unable to read file uEnv.txt **
Copying Linux from SD to RAM...
reading uImage
3447904 bytes read in 303 ms (10.9 MiB/s)
reading devicetree.dtb
7486 bytes read in 15 ms (487.3 KiB/s)
reading uramdisk.image.gz
3693174 bytes read in 323 ms (10.9 MiB/s)
## Booting kernel from Legacy Image at 03000000 ...
   Image Name:   Linux-3.18.0-xilinx
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    3447840 Bytes = 3.3 MiB
   Load Address: 00008000
   Entry Point:  00008000
   Verifying Checksum ... OK
## Loading init Ramdisk from Legacy Image at 02000000 ...
   Image Name:
   Image Type:   ARM Linux RAMDisk Image (gzip compressed)
```

```
   Data Size:    3693110 Bytes = 3.5 MiB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 02a00000
   Booting using the fdt blob at 0x2a00000
   Loading Kernel Image ... OK
   Loading Ramdisk to 1f7aa000, end 1fb2fa36 ... OK
   Loading Device Tree to 1f7a5000, end 1f7a9d3d ... OK

Starting kernel ...

Booting Linux on physical CPU 0x0
Linux version 3.18.0-xilinx (lvtongtom305@lin05-424cvlb.ece.tamu.edu) (gcc version
    4.9.1 (Sourcery CodeBench Lite 2014.11-30) ) #1 SMP PREEMPT Fri Sep 28 21:04:24 CDT
     2018
CPU: ARMv7 Processor [413fc090] revision 0 (ARMv7), cr=18c5387d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
Machine model: Xilinx Zynq
cma: Reserved 16 MiB at 0x1e400000
Memory policy: Data cache writealloc
PERCPU: Embedded 10 pages/cpu @5fbd3000 s8768 r8192 d24000 u40960
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 130048
Kernel command line: console=ttyPS0,115200 root=/dev/ram rw earlyprintk
PID hash table entries: 2048 (order: 1, 8192 bytes)
Dentry cache hash table entries: 65536 (order: 6, 262144 bytes)
Inode-cache hash table entries: 32768 (order: 5, 131072 bytes)
Memory: 492632K/524288K available (4650K kernel code, 258K rwdata, 1616K rodata, 212K
    init, 219K bss, 31656K reserved, 0K highmem)
Virtual kernel memory layout:
    vector  : 0xffff0000 - 0xffff1000   (   4 kB)
    fixmap  : 0xffc00000 - 0xffe00000   (2048 kB)
    vmalloc : 0x60800000 - 0xff000000   (2536 MB)
    lowmem  : 0x40000000 - 0x60000000   ( 512 MB)
    pkmap   : 0x3fe00000 - 0x40000000   (   2 MB)
    modules : 0x3f000000 - 0x3fe00000   (  14 MB)
      .text : 0x40008000 - 0x40626b1c   (6267 kB)
      .init : 0x40627000 - 0x4065c000   ( 212 kB)
      .data : 0x4065c000 - 0x4069cb60   ( 259 kB)
       .bss : 0x4069cb60 - 0x406d3a78   ( 220 kB)
Preemptible hierarchical RCU implementation.
    Dump stacks of tasks blocking RCU-preempt GP.
    RCU restricting CPUs from NR_CPUS=4 to nr_cpu_ids=2.
RCU: Adjusting geometry for rcu_fanout_leaf=16, nr_cpu_ids=2
NR_IRQS:16 nr_irqs:16 16
L2C-310 erratum 769419 enabled
L2C-310 enabling early BRESP for Cortex-A9
L2C-310 full line of zeros enabled for Cortex-A9
L2C-310 ID prefetch enabled, offset 1 lines
L2C-310 dynamic clock gating enabled, standby mode enabled
L2C-310 cache controller enabled, 8 ways, 512 kB
L2C-310: CACHE_ID 0x410000c8, AUX_CTRL 0x76360001
ps7-slcr mapped to 60804000
zynq_clock_init: clkc starts at 60804100
Zynq clock init
sched_clock: 64 bits at 325MHz, resolution 3ns, wraps every 3383112499200ns
ps7-ttc #0 at 60806000, irq=43
Console: colour dummy device 80x30
Calibrating delay loop... 1292.69 BogoMIPS (lpj=6463488)
```

```
pid_max: default: 32768 minimum: 301
Mount-cache hash table entries: 1024 (order: 0, 4096 bytes)
Mountpoint-cache hash table entries: 1024 (order: 0, 4096 bytes)
CPU: Testing write buffer coherency: ok
CPU0: thread -1, cpu 0, socket 0, mpidr 80000000
Setting up static identity map for 0x467598 - 0x4675f0
CPU1: Booted secondary processor
CPU1: thread -1, cpu 1, socket 0, mpidr 80000001
Brought up 2 CPUs
SMP: Total of 2 processors activated.
CPU: All CPU(s) started in SVC mode.
devtmpfs: initialized
VFP support v0.3: implementor 41 architecture 3 part 30 variant 9 rev 4
regulator-dummy: no parameters
NET: Registered protocol family 16
DMA: preallocated 256 KiB pool for atomic coherent allocations
cpuidle: using governor ladder
cpuidle: using governor menu
hw-breakpoint: found 5 (+1 reserved) breakpoint and 1 watchpoint registers.
hw-breakpoint: maximum watchpoint size is 4 bytes.
zynq-ocm f800c000.ps7-ocmc: ZYNQ OCM pool: 256 KiB @ 0x60880000
vgaarb: loaded
SCSI subsystem initialized
usbcore: registered new interface driver usbfs
usbcore: registered new interface driver hub
usbcore: registered new device driver usb
media: Linux media interface: v0.10
Linux video capture interface: v2.00
pps_core: LinuxPPS API ver. 1 registered
pps_core: Software ver. 5.3.6 - Copyright 2005-2007 Rodolfo Giometti <giometti@linux.
    it>
PTP clock support registered
EDAC MC: Ver: 3.0.0
Advanced Linux Sound Architecture Driver Initialized.
Switched to clocksource arm_global_timer
NET: Registered protocol family 2
TCP established hash table entries: 4096 (order: 2, 16384 bytes)
TCP bind hash table entries: 4096 (order: 3, 32768 bytes)
TCP: Hash tables configured (established 4096 bind 4096)
TCP: reno registered
UDP hash table entries: 256 (order: 1, 8192 bytes)
UDP-Lite hash table entries: 256 (order: 1, 8192 bytes)
NET: Registered protocol family 1
RPC: Registered named UNIX socket transport module.
RPC: Registered udp transport module.
RPC: Registered tcp transport module.
RPC: Registered tcp NFSv4.1 backchannel transport module.
Trying to unpack rootfs image as initramfs...
rootfs image is not initramfs (no cpio magic); looks like an initrd
Freeing initrd memory: 3608K (5f7aa000 - 5fb30000)
hw perfevents: enabled with armv7_cortex_a9 PMU driver, 7 counters available
futex hash table entries: 512 (order: 3, 32768 bytes)
jffs2: version 2.2. (NAND) (SUMMARY)  © 2001-2006 Red Hat, Inc.
msgmni has been set to 1001
io scheduler noop registered
io scheduler deadline registered
io scheduler cfq registered (default)
dma-pl330 f8003000.ps7-dma: Loaded driver for PL330 DMAC-241330
dma-pl330 f8003000.ps7-dma:     DBUFF-128x8bytes Num_Chans-8 Num_Peri-4 Num_Events-16
```

```
xuartps e0001000.serial: ttyPS0 at MMIO 0xe0001000 (irq = 82, base_baud = 3125000) is
    a xuartps
console [ttyPS0] enabled
xdevcfg f8007000.ps7-dev-cfg: ioremap 0xf8007000 to 6086c000
[drm] Initialized drm 1.1.0 20060810
brd: module loaded
loop: module loaded
CAN device driver interface
e1000e: Intel(R) PRO/1000 Network Driver - 2.3.2-k
e1000e: Copyright(c) 1999 - 2014 Intel Corporation.
libphy: XEMACPS mii bus: probed
xemacps e000b000.ps7-ethernet: invalid address, use random
xemacps e000b000.ps7-ethernet: MAC updated 8a:0b:61:1b:32:7e
xemacps e000b000.ps7-ethernet: pdev->id -1, baseaddr 0xe000b000, irq 54
ehci_hcd: USB 2.0 'Enhanced' Host Controller (EHCI) Driver
ehci-pci: EHCI PCI platform driver
ULPI transceiver vendor/product ID 0x0424/0x0007
Found SMSC USB3320 ULPI transceiver.
ULPI integrity check: passed.
zynq-ehci zynq-ehci.0: Xilinx Zynq USB EHCI Host Controller
zynq-ehci zynq-ehci.0: new USB bus registered, assigned bus number 1
zynq-ehci zynq-ehci.0: irq 53, io mem 0x00000000
zynq-ehci zynq-ehci.0: USB 2.0 started, EHCI 1.00
hub 1-0:1.0: USB hub found
hub 1-0:1.0: 1 port detected
usbcore: registered new interface driver usb-storage
mousedev: PS/2 mouse device common for all mice
i2c /dev entries driver
Xilinx Zynq CpuIdle Driver started
sdhci: Secure Digital Host Controller Interface driver
sdhci: Copyright(c) Pierre Ossman
sdhci-pltfm: SDHCI platform and OF driver helper
sdhci-arasan e0100000.ps7-sdio: No vmmc regulator found
sdhci-arasan e0100000.ps7-sdio: No vqmmc regulator found
mmc0: SDHCI controller on e0100000.ps7-sdio [e0100000.ps7-sdio] using ADMA
ledtrig-cpu: registered to indicate activity on CPUs
usbcore: registered new interface driver usbhid
usbhid: USB HID core driver
TCP: cubic registered
NET: Registered protocol family 17
can: controller area network core (rev 20120528 abi 9)
NET: Registered protocol family 29
can: raw protocol (rev 20120528)
can: broadcast manager protocol (rev 20120528 t)
can: netlink gateway (rev 20130117) max_hops=1
zynq_pm_ioremap: no compatible node found for 'xlnx,zynq-ddrc-a05'
zynq_pm_late_init: Unable to map DDRC IO memory.
Registering SWP/SWPB emulation handler
drivers/rtc/hctosys.c: unable to open rtc device (rtc0)
ALSA device list:
  No soundcards found.
RAMDISK: gzip image found at block 0
mmc0: new high speed SDHC card at address aaaa
mmcblk0: mmc0:aaaa SS08G 7.40 GiB
 mmcblk0: p1
EXT2-fs (ram0): warning: mounting unchecked fs, running e2fsck is recommended
VFS: Mounted root (ext2 filesystem) on device 1:0.
devtmpfs: mounted
Freeing unused kernel memory: 212K (40627000 - 4065c000)
```

```
Starting rcS...
++ Mounting filesystem
++ Setting up mdev
++ Starting telnet daemon
++ Starting http daemon
++ Starting ftp daemon
++ Starting dropbear (ssh) daemon
random: dropbear urandom read with 1 bits of entropy available
rcS Complete
zynq> mount /dev/mmcblk0p1 /mnt/
FAT-fs (mmcblk0p1): Volume was not properly unmounted. Some data may be corrupt.
    Please run fsck.
zynq> insmod /mnt/ir_dev.ko
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> ./mnt/devtest
message Reading...1th interrupt: raw_data = 490
2th interrupt: raw_data = 490
3th interrupt: raw_data = 490
4th interrupt: raw_data = 490
5th interrupt: raw_data = 490
6th interrupt: raw_data = 490
7th interrupt: raw_data = 490
8th interrupt: raw_data = 490
9th interrupt: raw_data = 490
10th interrupt: raw_data = 490

message Reading...message = 0x490

message Reading...message = 0x490

message Reading...message = 0x490

message Reading...message = 0x490
11th interrupt: raw_data = 610
12th interrupt: raw_data = 610
13th interrupt: raw_data = 610
14th interrupt: raw_data = 610
15th interrupt: raw_data = 610
16th interrupt: raw_data = 610
17th interrupt: raw_data = 610
18th interrupt: raw_data = 610
19th interrupt: raw_data = 610

message Reading...message = 0x490

message Reading...message = 0x490

message Reading...message = 0x490

message Reading...message = 0x490
```

src/terminal_printout

.

Conclusion

In this lab, in this lab, I used the all the knowledge and experience from previous labs to create the the IR-remote hardware with interrupt feature and a Linux device driver for the IR-remote which utilizes this interrupt. This lab took me a lot of time but in the end I learned a lot.

Answer to Questions

(a) **Contrast the use of an interrupt based device driver with the polling method used in the previous lab.**
According to the lecture slides, interrupts are better if the processor has other work to do and the time to respond to events aren't absolutely critical.
On the other hand, polling can be better if the processor has nothing better to do and has to respond to an event ASAP.

(b) **Are there any race conditions that your device driver does not address? If so, what are they and how would you fix them?**
Yes. The race condition can happen when both devtest and irq_handler are accessing the message queue at same time. One possible solution is to use semaphore and allow only one thread to access the queue at any time.

(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?**
"Fast" means the interrupt should be handled very quickly, compared to the so called "slow interrupt". To make this work properly, we should think about the time consumption when we develop the interrupt handler. I will delete all the printk statement in the interrupt handler and remove all of the debug variables. This will make the handler less time-consuming.

(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?**
If we specified an incorrect IRQ number, then the system will not function properly, because the interrupt handler will not be register correctly,and device driver will not work.

Appendix

```
1
2  `timescale 1 ns / 1 ps
3
4      module ir_demod_v2_0_S00_AXI #
5      (
6          // Users to add parameters here
7
8          // User parameters ends
9          // Do not modify the parameters beyond this line
10
11         // Width of S_AXI data bus
12         parameter integer C_S_AXI_DATA_WIDTH    = 32,
13         // Width of S_AXI address bus
14         parameter integer C_S_AXI_ADDR_WIDTH    = 4
15     )
16     (
17         // Users to add ports here
18         input wire IR_signal, // IR_signal from amplifier circuit
19         output reg IR_interrupt, // interrupt signal for ZYNQ system
20         /*output reg [C_S_AXI_DATA_WIDTH-1:0]   slv_reg0,
21         output reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg1,
22         output reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg2,
23         output reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg3,
24         */
25         // User ports ends
26         // Do not modify the ports beyond this line
27
28         // Global Clock Signal
29         input wire  S_AXI_ACLK,
30         // Global Reset Signal. This Signal is Active LOW
31         input wire  S_AXI_ARESETN,
32         // Write address (issued by master, acceped by Slave)
33         input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_AWADDR,
34         // Write channel Protection type. This signal indicates the
35             // privilege and security level of the transaction, and whether
36             // the transaction is a data access or an instruction access.
37         input wire [2 : 0] S_AXI_AWPROT,
38         // Write address valid. This signal indicates that the master signaling
39             // valid write address and control information.
40         input wire  S_AXI_AWVALID,
41         // Write address ready. This signal indicates that the slave is ready
42             // to accept an address and associated control signals.
43         output wire  S_AXI_AWREADY,
44         // Write data (issued by master, acceped by Slave)
45         input wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_WDATA,
46         // Write strobes. This signal indicates which byte lanes hold
47             // valid data. There is one write strobe bit for each eight
48             // bits of the write data bus.
49         input wire [(C_S_AXI_DATA_WIDTH/8)-1 : 0] S_AXI_WSTRB,
50         // Write valid. This signal indicates that valid write
51             // data and strobes are available.
52         input wire  S_AXI_WVALID,
53         // Write ready. This signal indicates that the slave
54             // can accept the write data.
55         output wire  S_AXI_WREADY,
56         // Write response. This signal indicates the status
```

```
57              // of the write transaction.
58          output wire [1 : 0] S_AXI_BRESP,
59          // Write response valid. This signal indicates that the channel
60              // is signaling a valid write response.
61          output wire  S_AXI_BVALID,
62          // Response ready. This signal indicates that the master
63              // can accept a write response.
64          input wire  S_AXI_BREADY,
65          // Read address (issued by master, acceped by Slave)
66          input wire [C_S_AXI_ADDR_WIDTH-1 : 0] S_AXI_ARADDR,
67          // Protection type. This signal indicates the privilege
68              // and security level of the transaction, and whether the
69              // transaction is a data access or an instruction access.
70          input wire [2 : 0] S_AXI_ARPROT,
71          // Read address valid. This signal indicates that the channel
72              // is signaling valid read address and control information.
73          input wire  S_AXI_ARVALID,
74          // Read address ready. This signal indicates that the slave is
75              // ready to accept an address and associated control signals.
76          output wire  S_AXI_ARREADY,
77          // Read data (issued by slave)
78          output wire [C_S_AXI_DATA_WIDTH-1 : 0] S_AXI_RDATA,
79          // Read response. This signal indicates the status of the
80              // read transfer.
81          output wire [1 : 0] S_AXI_RRESP,
82          // Read valid. This signal indicates that the channel is
83              // signaling the required read data.
84          output wire  S_AXI_RVALID,
85          // Read ready. This signal indicates that the master can
86              // accept the read data and response information.
87          input wire  S_AXI_RREADY
88      );
89
90          // AXI4LITE signals
91          reg [C_S_AXI_ADDR_WIDTH-1 : 0]  axi_awaddr;
92          reg     axi_awready;
93          reg     axi_wready;
94          reg [1 : 0]     axi_bresp;
95          reg     axi_bvalid;
96          reg [C_S_AXI_ADDR_WIDTH-1 : 0]  axi_araddr;
97          reg     axi_arready;
98          reg [C_S_AXI_DATA_WIDTH-1 : 0]  axi_rdata;
99          reg [1 : 0]     axi_rresp;
100         reg     axi_rvalid;
101
102         // Example-specific design signals
103         // local parameter for addressing 32 bit / 64 bit C_S_AXI_DATA_WIDTH
104         // ADDR_LSB is used for addressing 32/64 bit registers/memories
105         // ADDR_LSB = 2 for 32 bits (n downto 2)
106         // ADDR_LSB = 3 for 64 bits (n downto 3)
107         localparam integer ADDR_LSB = (C_S_AXI_DATA_WIDTH/32) + 1;
108         localparam integer OPT_MEM_ADDR_BITS = 1;
109         //------------------------------------------------
110         //-- Signals for user logic register space example
111         //------------------------------------------------
112         //-- Number of Slave Registers 4
113         reg [C_S_AXI_DATA_WIDTH-1:0]    slv_reg0;
114         reg [C_S_AXI_DATA_WIDTH-1:0]    slv_reg1;
115         reg [C_S_AXI_DATA_WIDTH-1:0]    slv_reg2;
```

```
116         reg [C_S_AXI_DATA_WIDTH-1:0]    slv_reg3;
117         wire     slv_reg_rden;
118         wire     slv_reg_wren;
119         reg [C_S_AXI_DATA_WIDTH-1:0]     reg_data_out;
120         integer  byte_index;
121
122         // I/O Connections assignments
123
124         assign S_AXI_AWREADY    = axi_awready;
125         assign S_AXI_WREADY = axi_wready;
126         assign S_AXI_BRESP  = axi_bresp;
127         assign S_AXI_BVALID = axi_bvalid;
128         assign S_AXI_ARREADY    = axi_arready;
129         assign S_AXI_RDATA  = axi_rdata;
130         assign S_AXI_RRESP  = axi_rresp;
131         assign S_AXI_RVALID = axi_rvalid;
132         // Implement axi_awready generation
133         // axi_awready is asserted for one S_AXI_ACLK clock cycle when both
134         // S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_awready is
135         // de-asserted when reset is low.
136
137         always @( posedge S_AXI_ACLK )
138         begin
139           if ( S_AXI_ARESETN == 1'b0 )
140             begin
141               axi_awready <= 1'b0;
142             end
143           else
144             begin
145               if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID)
146                 begin
147                   // slave is ready to accept write address when
148                   // there is a valid write address and write data
149                   // on the write address and data bus. This design
150                   // expects no outstanding transactions.
151                   axi_awready <= 1'b1;
152                 end
153               else
154                 begin
155                   axi_awready <= 1'b0;
156                 end
157             end
158         end
159
160         // Implement axi_awaddr latching
161         // This process is used to latch the address when both
162         // S_AXI_AWVALID and S_AXI_WVALID are valid.
163
164         always @( posedge S_AXI_ACLK )
165         begin
166           if ( S_AXI_ARESETN == 1'b0 )
167             begin
168               axi_awaddr <= 0;
169             end
170           else
171             begin
172               if (~axi_awready && S_AXI_AWVALID && S_AXI_WVALID)
173                 begin
174                   // Write Address latching
```

```
175                 axi_awaddr <= S_AXI_AWADDR;
176               end
177           end
178       end
179
180       // Implement axi_wready generation
181       // axi_wready is asserted for one S_AXI_ACLK clock cycle when both
182       // S_AXI_AWVALID and S_AXI_WVALID are asserted. axi_wready is
183       // de-asserted when reset is low.
184
185       always @( posedge S_AXI_ACLK )
186       begin
187         if ( S_AXI_ARESETN == 1'b0 )
188           begin
189             axi_wready <= 1'b0;
190           end
191         else
192           begin
193             if (~axi_wready && S_AXI_WVALID && S_AXI_AWVALID)
194               begin
195                 // slave is ready to accept write data when
196                 // there is a valid write address and write data
197                 // on the write address and data bus. This design
198                 // expects no outstanding transactions.
199                 axi_wready <= 1'b1;
200               end
201             else
202               begin
203                 axi_wready <= 1'b0;
204               end
205           end
206       end
207
208       // Implement memory mapped register select and write logic generation
209       // The write data is accepted and written to memory mapped registers when
210       // axi_awready, S_AXI_WVALID, axi_wready and S_AXI_WVALID are asserted. Write
    strobes are used to
211       // select byte enables of slave registers while writing.
212       // These registers are cleared when reset (active low) is applied.
213       // Slave register write enable is asserted when valid address and data are
    available
214       // and the slave is ready to accept the write address and write data.
215       assign slv_reg_wren = axi_wready && S_AXI_WVALID && axi_awready &&
    S_AXI_AWVALID;
216       /*
217       always @( posedge S_AXI_ACLK )
218       begin
219         if ( S_AXI_ARESETN == 1'b0 )
220           begin
221             slv_reg0 <= 0;
222             slv_reg1 <= 0;
223             slv_reg2 <= 0;
224             slv_reg3 <= 0;
225           end
226         else begin
227           if (slv_reg_wren)
228             begin
229               case ( axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
230                 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 = 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 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
```

```
282           else
283             begin
284               if (axi_awready && S_AXI_AWVALID && ~axi_bvalid && axi_wready &&
      S_AXI_WVALID)
285                 begin
286                   // indicates a valid write response is available
287                   axi_bvalid <= 1'b1;
288                   axi_bresp  <= 2'b0; // 'OKAY' response
289                 end                   // work error responses in future
290               else
291                 begin
292                   if (S_AXI_BREADY && axi_bvalid)
293                     //check if bready is asserted while bvalid is high)
294                     //(there is a possibility that bready is always asserted high)
295                     begin
296                       axi_bvalid <= 1'b0;
297                     end
298                 end
299             end
300         end
301
302         // Implement axi_arready generation
303         // axi_arready is asserted for one S_AXI_ACLK clock cycle when
304         // S_AXI_ARVALID is asserted. axi_awready is
305         // de-asserted when reset (active low) is asserted.
306         // The read address is also latched when S_AXI_ARVALID is
307         // asserted. axi_araddr is reset to zero on reset assertion.
308
309         always @( posedge S_AXI_ACLK )
310         begin
311           if ( S_AXI_ARESETN == 1'b0 )
312             begin
313               axi_arready <= 1'b0;
314               axi_araddr  <= 32'b0;
315             end
316           else
317             begin
318               if (~axi_arready && S_AXI_ARVALID)
319                 begin
320                   // indicates that the slave has acceped the valid read address
321                   axi_arready <= 1'b1;
322                   // Read address latching
323                   axi_araddr  <= S_AXI_ARADDR;
324                 end
325               else
326                 begin
327                   axi_arready <= 1'b0;
328                 end
329             end
330         end
331
332         // Implement axi_arvalid generation
333         // axi_rvalid is asserted for one S_AXI_ACLK clock cycle when both
334         // S_AXI_ARVALID and axi_arready are asserted. The slave registers
335         // data are available on the axi_rdata bus at this instance. The
336         // assertion of axi_rvalid marks the validity of read data on the
337         // bus and axi_rresp indicates the status of read transaction.axi_rvalid
338         // is deasserted on reset (active low). axi_rresp and axi_rdata are
339         // cleared to zero on reset (active low).
```

```
340         always @( posedge S_AXI_ACLK )
341         begin
342           if ( S_AXI_ARESETN == 1'b0 )
343             begin
344               axi_rvalid <= 0;
345               axi_rresp  <= 0;
346             end
347           else
348             begin
349               if (axi_arready && S_AXI_ARVALID && ~axi_rvalid)
350                 begin
351                   // Valid read data is available at the read data bus
352                   axi_rvalid <= 1'b1;
353                   axi_rresp  <= 2'b0; // 'OKAY' response
354                 end
355               else if (axi_rvalid && S_AXI_RREADY)
356                 begin
357                   // Read data is accepted by the master
358                   axi_rvalid <= 1'b0;
359                 end
360             end
361         end
362
363         // Implement memory mapped register select and read logic generation
364         // Slave register read enable is asserted when valid address is available
365         // and the slave is ready to accept the read address.
366         assign slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid;
367         always @(*)
368         begin
369               // Address decoding for reading registers
370               case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
371                 2'h0   : reg_data_out <= slv_reg0;
372                 2'h1   : reg_data_out <= slv_reg1;
373                 2'h2   : reg_data_out <= slv_reg2;
374                 2'h3   : reg_data_out <= slv_reg3;
375                 default : reg_data_out <= 0;
376               endcase
377         end
378
379         // Output register or memory read data
380         always @( posedge S_AXI_ACLK )
381         begin
382           if ( S_AXI_ARESETN == 1'b0 )
383             begin
384               axi_rdata  <= 0;
385             end
386           else
387             begin
388               // When there is a valid read address (S_AXI_ARVALID) with
389               // acceptance of read address by the slave (axi_arready),
390               // output the read dada
391               if (slv_reg_rden)
392                 begin
393                   axi_rdata <= reg_data_out;     // register read data
394                 end
395             end
396         end
397
398         // Add user logic here
```

```
399         reg [31:0] pulse_length_counter; // measure the length of each pulse
400         reg [1:0] state; // state varible for FSM
401         reg [3:0] pulse_idx; // measure the number of pulses for decoding
402         reg [11:0] temp_reg0; // store temporary value
403         parameter START = 2'b00, LEN0 = 2'b01, LEN1 = 2'b10, OTHER = 2'b11;
404         reg IR_signal_prev; // IR signal from previous clock cycle
405         reg [31:0] slv_reg1_prev; // slv_reg1 value from previous clock cycle
406         always @(posedge S_AXI_ACLK) begin // set up counter
407             if ( S_AXI_ARESETN == 1'b0 ) begin // reset
408                 pulse_length_counter <= 0; //reset counter
409                 state <= OTHER; // reset state
410             end
411             else begin
412                 if (~IR_signal) begin // if IR_signal is 0
413                     pulse_length_counter <= pulse_length_counter + 1; //increment
414                 end
415                 else begin // if IR_signal is 1
416                     pulse_length_counter <= 0; // counter set to zero
417                 end
418 
419                 if (pulse_length_counter > 40_000 && pulse_length_counter <= 50_000)
    begin
420                     state <= LEN0; // around 0.6ms, or 45_000 cycles
421                 end
422                 else if (pulse_length_counter > 85_000 && pulse_length_counter <= 95
    _000) begin
423                     state <= LEN1; // around 1.2ms, or 90_000 cycles
424                 end
425                 else if (pulse_length_counter > 175_000 && pulse_length_counter <= 185
    _000) begin
426                     state <= START; // around 2.4ms, or 180_000 cycles
427                 end
428                 else begin
429                     state <= OTHER; // Other
430                 end
431             end
432         end
433 
434         always @(posedge IR_signal) begin // when IR signal rises, send out state
435             case(state)
436                 START: begin // when start is received
437                     //slv_reg0  <= 0;
438                     temp_reg0 <= 0; // set temp to 0
439                     pulse_idx <= 0; // index varible reset
440                 end
441                 LEN0: begin // when 0 is received
442                     if (pulse_idx < 11) begin // for the first 11 bits
443                         temp_reg0[11 - pulse_idx] <= 0; // push 0
444                         pulse_idx <= pulse_idx + 1; // increment idx
445                     end
446                     if (pulse_idx == 11) begin // last bit
447                         slv_reg0 <= temp_reg0; // push temp to slv_reg0
448                         if (slv_reg1 >= 1) begin
449                             slv_reg1 <= slv_reg1 +1; // slv_reg1 increment
450                         end
451                         else begin // initalize reg1
452                             slv_reg1 <= 1;
453                         end
454                     end
```

```
455                 end
456                 LEN1: begin // when 1 is received
457                     if (pulse_idx < 11) begin // for the first 11 bits
458                         temp_reg0[11 - pulse_idx] <= 1; // push 1
459                         pulse_idx <= pulse_idx + 1; // increment idx
460                     end
461                     if (pulse_idx == 11) begin // last bit
462                         slv_reg0 <= temp_reg0 + 1; // push temp to slv_reg0, with the
    last "1"
463                         if (slv_reg1 >= 1) begin
464                             slv_reg1 <= slv_reg1 +1; // slv_reg1 increment
465                         end
466                         else begin // initalize reg1
467                             slv_reg1 <= 1;
468                         end
469                     end
470                 end
471                 OTHER: begin
472                     pulse_idx <= 4'b1111; // if wrong signal received, set pulse index
     to a impossible value
473                 end
474                 default: ;
475             endcase
476         end
477 
478         always @( posedge S_AXI_ACLK ) begin // interrupt block
479             if ( S_AXI_ARESETN == 1'b0 ) begin // when reset
480                 IR_interrupt <= 0; // reset interrput varible
481                 slv_reg2 <= 0; // reset slv_reg2
482             end
483             else begin
484                 slv_reg2[31] <= IR_interrupt; // set the highest bit to interrupt
485                 slv_reg1_prev <= slv_reg1; // store the current vaule
486                 if (slv_reg_wren) begin // enable the write feature for lower 16 bits
     of reg2
487                     if( axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] == 2'h2) begin
488                         for ( byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/16)-1;
     byte_index = byte_index+1 ) begin
489                             if ( S_AXI_WSTRB[byte_index] == 1 ) begin
490                                 // Respective byte enables are asserted as per write
    strobes
491                                 // Slave register 2
492                                 slv_reg2[(byte_index*8) +: 8] <= S_AXI_WDATA[(
    byte_index*8) +: 8];
493                             end
494                         end
495                     end
496                     else slv_reg2[15:0] <= slv_reg2[15:0]; // if nothing is written in
    , keep the previous value
497                 end
498                 if (slv_reg2[0]) begin // status/ control
499                     IR_interrupt <= 0; // reset interrupt
500                     //slv_reg2[0] <= 0;
501                 end
502                 else if (slv_reg1 != slv_reg1_prev) begin // if counter register
    changed
503                     IR_interrupt <= 1; // pull up interrupt signal
504                 end
505             end
```

```
506         end
507
508         // User logic ends
509
510     endmodule
```

src/ir_demod_v2_0_S00_AXI.v

```
1
2  `timescale 1 ns / 1 ps
3
4      module ir_demod_v2_0 #
5      (
6          // Users to add parameters here
7
8          // User parameters ends
9          // Do not modify the parameters beyond this line
10
11
12         // Parameters of Axi Slave Bus Interface S00_AXI
13         parameter integer C_S00_AXI_DATA_WIDTH  = 32,
14         parameter integer C_S00_AXI_ADDR_WIDTH  = 4
15     )
16     (
17         // Users to add ports here
18         input wire IR_signal, // IR_signal from amplifier circuit
19         output wire IR_interrupt, // interrupt signal for ZYNQ system
20         // User ports ends
21         // Do not modify the ports beyond this line
22
23
24         // Ports of Axi Slave Bus Interface S00_AXI
25         input wire  s00_axi_aclk,
26         input wire  s00_axi_aresetn,
27         input wire [C_S00_AXI_ADDR_WIDTH-1 : 0] s00_axi_awaddr,
28         input wire [2 : 0] s00_axi_awprot,
29         input wire  s00_axi_awvalid,
30         output wire  s00_axi_awready,
31         input wire [C_S00_AXI_DATA_WIDTH-1 : 0] s00_axi_wdata,
32         input wire [(C_S00_AXI_DATA_WIDTH/8)-1 : 0] s00_axi_wstrb,
33         input wire  s00_axi_wvalid,
34         output wire  s00_axi_wready,
35         output wire [1 : 0] s00_axi_bresp,
36         output wire  s00_axi_bvalid,
37         input wire  s00_axi_bready,
38         input wire [C_S00_AXI_ADDR_WIDTH-1 : 0] s00_axi_araddr,
39         input wire [2 : 0] s00_axi_arprot,
40         input wire  s00_axi_arvalid,
41         output wire  s00_axi_arready,
42         output wire [C_S00_AXI_DATA_WIDTH-1 : 0] s00_axi_rdata,
43         output wire [1 : 0] s00_axi_rresp,
44         output wire  s00_axi_rvalid,
45         input wire  s00_axi_rready
46     );
47 // Instantiation of Axi Bus Interface S00_AXI
48     ir_demod_v2_0_S00_AXI # (
49         .C_S_AXI_DATA_WIDTH(C_S00_AXI_DATA_WIDTH),
50         .C_S_AXI_ADDR_WIDTH(C_S00_AXI_ADDR_WIDTH)
51     ) ir_demod_v2_0_S00_AXI_inst (
```

```
        .IR_signal(IR_signal),
        .IR_interrupt(IR_interrupt),
        .S_AXI_ACLK(s00_axi_aclk),
        .S_AXI_ARESETN(s00_axi_aresetn),
        .S_AXI_AWADDR(s00_axi_awaddr),
        .S_AXI_AWPROT(s00_axi_awprot),
        .S_AXI_AWVALID(s00_axi_awvalid),
        .S_AXI_AWREADY(s00_axi_awready),
        .S_AXI_WDATA(s00_axi_wdata),
        .S_AXI_WSTRB(s00_axi_wstrb),
        .S_AXI_WVALID(s00_axi_wvalid),
        .S_AXI_WREADY(s00_axi_wready),
        .S_AXI_BRESP(s00_axi_bresp),
        .S_AXI_BVALID(s00_axi_bvalid),
        .S_AXI_BREADY(s00_axi_bready),
        .S_AXI_ARADDR(s00_axi_araddr),
        .S_AXI_ARPROT(s00_axi_arprot),
        .S_AXI_ARVALID(s00_axi_arvalid),
        .S_AXI_ARREADY(s00_axi_arready),
        .S_AXI_RDATA(s00_axi_rdata),
        .S_AXI_RRESP(s00_axi_rresp),
        .S_AXI_RVALID(s00_axi_rvalid),
        .S_AXI_RREADY(s00_axi_rready)
    );

    // Add user logic here

    // User logic ends

    endmodule
```

src/ir_demod_v2_0.v

```
## IR_signal
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets IR_signal_IBUF]
set_property PACKAGE_PIN T20 [get_ports IR_signal]
set_property IOSTANDARD LVCMOS33 [get_ports IR_signal]

## IR_interrupt
set_property PACKAGE_PIN U20 [get_ports IR_interrupt]
set_property IOSTANDARD LVCMOS33 [get_ports IR_interrupt]
```

src/ir2_xdc.xdc

```
/*  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 "ir_dev.h"
```

```
16 //reset; make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi-
17
18
19 /* This structure defines the function pointers to our functions for
20    opening, closing, reading and writing the device file.  There are
21    lots of other pointers in this structure which we are not using,
22    see the whole definition in linux/fs.h */
23 static struct file_operations fops = {
24   .read = device_read,
25   .write = device_write,
26   .open = device_open,
27   .release = device_release
28 };
29
30 /*
31  * This function is called when the module is loaded and registers a
32  * device for the driver to use.
33  */
34 int my_init(void)
35 {
36   virt_addr = (void*)ioremap(PHY_ADDR, MEMSIZE); // map virtual address to physical
      address
37   init_waitqueue_head(&queue);  /* initialize the wait queue */
38
39   /* Initialize the semaphor we will use to protect against multiple
40      users opening the device  */
41   sema_init(&sem, 1);
42
43   Major = register_chrdev(0, DEVICE_NAME, &fops); // register character device
44   if (Major < 0) { // error handling
45     printk(KERN_ALERT "Registering char device failed with %d\n", Major);
46     return Major;
47   }
48   printk(KERN_INFO "Registered a device with dynamic Major number of %d\n", Major); //
       print Major number
49   printk(KERN_INFO "Create a device file for this device with this command:\n'mknod /
      dev/%s c %d 0'.\n", DEVICE_NAME, Major);
50
51   return 0;     /* success */
52 }
53
54 /*
55  * This function is called when the module is unloaded, it releases
56  * the device file.
57  */
58 void my_cleanup(void)
59 {
60   /*
61    * Unregister the device
62    */
63   printk(KERN_ALERT "unmapping virtual address space...\n");
64   unregister_chrdev(Major, DEVICE_NAME); // unregister character device
65   iounmap((void*)virt_addr); // unmap virtual address
66 }
67
68
69 /*
70  * Called when a process tries to open the device file, like "cat
71  * /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_queue_Ptr = msg_queue_new(QUEUE_LEN); // allocate message queue
  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);
  msg_queue_destroy(msg_queue_Ptr); // free message queue
  /*
   * 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.
 */
```

```
131 static ssize_t device_read(struct file *filp,   /* see include/linux/fs.h   */
132                 char *buffer,    /* buffer to fill with data */
133                 size_t length,   /* length of the buffer     */
134                 loff_t * offset)
135 {
136   int bytes_read = 0;
137   ir_msg temp_msg; // temporary message varible
138   /* In this driver msg_Ptr is NULL until an interrupt occurs */
139   //wait_event_interruptible(queue, (msg_Ptr != NULL)); /* sleep until interrupted */
140
141   if (DEBUG) printk(KERN_INFO "device_read\n"); // print debug statement
142   /*
143    * If we're at the end of the message,
144    * return 0 signifying end of file
145    */
146   /*if (*msg_Ptr == 0) {
147     msg_Ptr = NULL; // completed interrupt servicing reset pointer to wait for another
       interrupt
148     if (DEBUG) printk(KERN_INFO "device_read: completed interrupt servicing\n");
149     return 0;
150   }*/
151
152   /*
153    * Actually put the data into the buffer
154    */
155   while (length && dequeue(msg_queue_Ptr, &temp_msg) >= 0) {
156
157     /*
158      * The buffer is in the user data segment, not the kernel
159      * segment so "*" assignment won't work.  We have to use
160      * put_user which copies data from the kernel data segment to
161      * the user data segment.
162      */
163      if (DEBUG) printk(KERN_INFO "Read: loop\n"); // debug statement
164       put_user(temp_msg.byte_low, buffer++); /* one char at a time... */
165       put_user(temp_msg.byte_high, buffer++); /* one char at a time... */
166       length--; // decrement length
167       bytes_read++; // increment bytes_read
168
169   }
170
171   /*
172    * Most read functions return the number of bytes put into the buffer
173    */
174   return bytes_read;
175 }
176
177 /*
178  * Called when a process writes to dev file: echo "hi" > /dev/hello
179  * Next time we'll make this one do something interesting.
180  */
181 static ssize_t
182 device_write(struct file *filp, const char *buff, size_t len, loff_t * off)
183 {
184
185   /* not allowing writes for now, just printing a message in the
186      kernel logs. */
187   printk(KERN_ALERT "Sorry, this operation isn't supported.\n");
188   return -EINVAL;       /* Fail */
```

```
}

irqreturn_t irq_handler(int irq, void *dev_id) {
  static int counter = 0;   /* keep track of the number of
                   interrupts handled */
  int raw_data, raw_count; // temporary varible for data and count
  ir_msg* temp_msg; // temporary pointer for message

  //sprintf(msg, "IRQ Num %d called, interrupts processed %d times\n", irq, counter++)
    ;
  //msg_Ptr = msg;
  interrupt_counter += 1; // interrupt counter increment
  raw_data = ioread32(virt_addr); // read slv_reg0
  temp_msg = ir_msg_new((raw_data >> 8) & 0xff, raw_data & 0xff); // convert to ir_msg
     type
  enqueue(msg_queue_Ptr, temp_msg); // put message into msg_queue
  if (DEBUG) printk(KERN_INFO "irq_handler: raw_data = %x\n", raw_data); // debug
    statement
  printk(KERN_INFO "%dth interrupt: raw_data = %x\n", interrupt_counter, raw_data); //
     print out interrput
  iowrite16(1, virt_addr + 8); // reset interrput
  while (ioread32(virt_addr + 8)&0x8000); // wait till interrupt
  iowrite16(0, virt_addr + 8); // clear reset bit

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



/* These define info that can be displayed by modinfo */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Paul 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);
```

src/ir_dev.c

```
/* 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 <linux/slab.h>    /* Provide kmalloc()/kfree() */
#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>          // Needed for IO reads and writes
#include "xparameters.h"     // Needed for IO reads and writes
#include "xparameters_ps.h"     // Needed for IO reads and writes
#include "msg_queue.h"
//#include "msg_queue.h"
/* Some defines */
#define DEVICE_NAME "ir_demod"

#define BUF_LEN 80
#define IRQ_NUM 61
// From xparameters.h, physical address of multiplier
#define PHY_ADDR XPAR_IR_DEMOD_0_S00_AXI_BASEADDR
// Size of physical address range for multiply
#define MEMSIZE XPAR_IR_DEMOD_0_S00_AXI_HIGHADDR - XPAR_IR_DEMOD_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_Ptr;
static char msg[BUF_LEN];   /* The msg the device will give when asked */
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 */
static void* virt_addr;
// Message queue related
static int interrupt_counter=0; // initalize interrupt counter

msg_queue *msg_queue_Ptr; // initalize message queue pointer
```

src/ir_dev.h

```
#include <linux/slab.h>    /* Provide kmalloc()/kfree() */
// debug flag
#define DEBUG 0
#define QUEUE_LEN 100 // set the length of queue to 100
typedef struct ir_msg { // define a 16 bits data type to store message
    unsigned char byte_low; // lower bits
    unsigned char byte_high; // upper bits
} ir_msg;

//typedef unsigned short int ir_msg; // use 2-byte data type for message
ir_msg* ir_msg_new(unsigned char high, unsigned char low) { // initalize ir_message
    ir_msg* new_ir_msg = (ir_msg*)kmalloc(sizeof(ir_msg), GFP_KERNEL); // allocate
    message
```

```
    new_ir_msg -> byte_low = low; // assign lower bits
    new_ir_msg -> byte_high = high; // assign upper bits
    return new_ir_msg; // return pointer
}

typedef struct msg_queue { // define a message queue structure using circular array
    int head_idx, tail_idx, size, capacity; // parameters
    ir_msg* msg_array; // circular array pointer
} msg_queue;

msg_queue* msg_queue_new (int capacity) { // initalize msg_queue
    msg_queue* queue = (msg_queue*)kmalloc(sizeof(msg_queue), GFP_KERNEL); // allocate
     queue
    queue -> capacity = capacity; // assign capacity
    queue -> head_idx = 0; // initalize head index
    queue -> tail_idx = 0; // initalize tail index
    queue -> size = 0; // initalize queue size
    queue -> msg_array = (ir_msg*)kmalloc(QUEUE_LEN*sizeof(ir_msg), GFP_KERNEL); //
    allocate array
    return queue;
}

void msg_queue_destroy (msg_queue* queue) { // free queue
    if (queue == NULL) { // handle null pointer
        printk(KERN_INFO "Queue is NULL!\n");
        return;
    }
    else {
        kfree(queue -> msg_array);  // free array first
        kfree(queue); // then free queue
        return;
    }
}

int is_full (msg_queue* queue) { // check if the queue is full
    if (queue == NULL) { // handle null pointer
        printk(KERN_INFO "Queue is NULL!\n");
        return -2;
    }
    if (queue -> size == queue -> capacity) { // if queue size reach the capacity
        if (DEBUG) printk(KERN_INFO "is_full: queue is full"); // then queue is full
        return 1; // return 1 for full
    }
    else return 0; // 0: not full yet
}

int is_empty (msg_queue* queue) { // check if the queue is full
    if (queue == NULL) { // handle null pointer
        printk(KERN_INFO "Queue is NULL!\n");
        return -2;
    }
    if (queue -> size == 0) { // if queue size is zero
        if (DEBUG) printk(KERN_INFO "is_empty: queue is empty"); // then queue is
    empty
        return 1; // return 1 for empty
    }
    else return 0; // 0: not empty yet
}

```

```
69 void print_queue (msg_queue* queue) { // print function for debug
70     if (queue == NULL) { // handle null pointer
71         printk(KERN_INFO "Queue is NULL!\n");
72         return;
73     }
74     if (is_empty(queue)) return; // if queue is empty then do nothing
75     else { // if not empty
76         int i; // for loop varible
77         //printk(KERN_INFO "print_queue: ");
78         if (queue -> head_idx <= queue -> tail_idx) { // if head index is less than
    tail index
79             for (i = queue -> head_idx; i <= queue -> tail_idx; i++) { // then iterate
     from head to tail
80                 printk(KERN_INFO "%x%x, ", queue->msg_array[i].byte_high, queue->
    msg_array[i].byte_low); // print messages from array
81             }
82         }
83         else { // if tail index is less than head index
84             for (i = queue -> tail_idx; i <= queue -> head_idx; i++) { // then iterate
     from tail to head
85                 printk(KERN_INFO "%x%x, ", queue->msg_array[i].byte_high, queue->
    msg_array[i].byte_low); // print messages from array
86             }
87         }
88         printk(KERN_INFO "\n"); // new line
89         return;
90     }
91 }
92
93 int enqueue (msg_queue* queue, ir_msg* msg_in) { // put message into queue
94     if (queue == NULL) { // handle null pointer
95         printk(KERN_INFO "Queue is NULL!\n");
96         return -2;
97     }
98     if (is_full(queue)) return -1; // if full then don't enqueue
99     else { // if not full yet
100         queue -> tail_idx = (queue -> tail_idx + 1) % queue -> capacity; // tail index
     increment in circular manner
101         queue -> msg_array[queue -> tail_idx] = *msg_in; // put the message in the
    array
102         queue -> size++; // increment the size
103         if (DEBUG) print_queue(queue); // debug statement
104         return queue -> size; // return the current queue size
105     }
106 }
107
108 int dequeue (msg_queue* queue, ir_msg* msg_out) { // push message out of queue
109     if (queue == NULL) { // handle null pointer
110         printk(KERN_INFO "Queue is NULL!\n");
111         return -2;
112     }
113     if (is_empty(queue)) return -1; // if empty do nothing
114     else { // if not empty
115         *msg_out = queue -> msg_array[queue -> head_idx + 1]; // pass message from
    array
116         queue -> head_idx = (queue -> head_idx + 1) % queue -> capacity; // head index
     increment in circular manner
117         queue -> size--; // decrement size
118         if (DEBUG) { // debug statement
```

```
            printk(KERN_INFO "dequeue!\n");
            print_queue(queue);
        }
        return queue -> size; // return current size
    }
}
```

src/msg_queue.h

```
obj-m += ir_dev.o

all:
    make -C /home/grads/l/lvtongtom305/ecen749/lab5/linux-3.14 M=$(PWD) modules

clean:
    make -C /home/grads/l/lvtongtom305/ecen749/lab5/linux-3.14 M=$(PWD) clean
```

src/Makefile

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
// arm-xilinx-linux-gnueabi-gcc -o devtest devtest.c
#define MAX_STRING_SIZE 100
int main() {
    unsigned short read_0, read_1;
    int fd;     // File descriptor
    unsigned char* rd_buf = (unsigned char*) malloc(MAX_STRING_SIZE*2*sizeof(char));
    char input[3] = "0";
    int data;
    int i, buf_read_len;
    unsigned short* msg;
    int input_num = 0;
    // Open device file for reading and writing
    // Use 'open' to open '/dev/multiplier'
    fd = open("/dev/ir_demod", O_RDWR);
    // Handle error opening file
    if(fd == -1) {
        printf("Failed to open device file!\n");
        return -1;
    }
    while (input != "q") { // quit when typing in "q"
        printf("message Reading...\n");
        input_num = atoi(input); // change input type from string to integer, if
    possible
        if (input_num) { // if input varible is valid
            buf_read_len = read(fd, rd_buf, input_num); // read input_num bits from
    ir_demod
            if (buf_read_len) { // if read succeed
                msg=(unsigned short*)rd_buf; // convert read buffer to two byte type
                for (i = 0; i < buf_read_len; i++) { // iterate through the buffer
                    data=msg[i]&0xfff; // read out only the lower 12 bits
                    printf("message %d = 0x%x\n", (i+1), (unsigned int)data); // print
     out the data
                }
            }
```

```
38          }
39          // Read from terminal
40          printf("Enter number of message you want to read:"); // print statement for
        user interface.
41          fgets(input, MAX_STRING_SIZE, stdin); // get user input for the number of
        message user want to read
42          // Continue unless user entered 'q'
43      }
44      close(fd);
45      free(rd_buf);
46      return 0;
47  }
```

src/devtest.c