ECEN 449: Microprocessor System Design
Department of Electrical and Computer Engineering
Texas A&M University

Prof. Sunil P Khatri
(Lab exercise created and tested by Ramu Endluri, He Zhou and Sunil P Khatri)

Laboratory Exercise #8

Interrupt-Based IR-remote Device Driver

Objective

The main objective of lab this week is to familiarize you with the use of interrupts in a Linux system. As an example, you will add an interrupt signal to the IR-remote hardware built in Lab 7 and write a Linux device driver for the IR-remote which utilizes this interrupt.

Procedure

The Linux compliant hardware system that you will create in lab this week is depicted in Figure 1.

1. In Lab 7 our software interfaced with the IR-remote by polling registers within the ir_demod peripheral. Polling wastes CPU time and overloads the bus by continually transferring redundant data from the peripheral registers to the PS. Using interrupts within the Linux OS provides us with a better method for handling asynchronous events from the IR-remote. The following steps will guide you through the process of adding an interrupt signal to your existing ir_demod hardware.

   (a) Make a copy of your Lab 7 folder along with the 'ir_demod' IP and rename the folder as Lab 8.

   (b) Open your Vivado project from Lab 8. Click on 'Project Settings' and select 'IP'. Remove the location of the old repository(lab7) and add the location of the new repository(lab8) which holds the 'ir_demod'ip repository of the Vivado project.

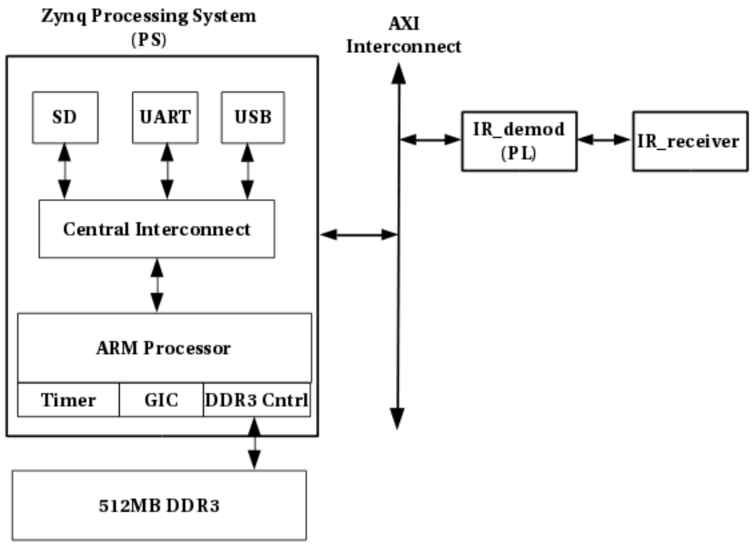

Figure 1: Zynq System Diagram

(c) Remove the 'ir_demod' IP from the design as it corresponds to old repository and add it again so that IP from the lab8 repository is added to project. Press 'OK'.

(d) Be prepared to edit the 'ir_demod' IP and modify the 'ir_demod_v1_SOO_AXI.v' and 'ir_demod_v1.v' source files. In Vivado open the block design, right click on 'ir_demod_0', and select 'Edit in IP Packager'.

(e) In 'ir_demod_v1_0_SOO_AXI.v' and 'ir_demod_v1_0.v', add a second user defined port called 'IR_interrupt'. Follow the procedure for creating the 'IR_signal' as outlined in Lab 7.
Please note that the 'Interrupt' signal must be defined as an output signal, rather than an input signal.

(f) Within the user logic portion of the peripheral source code, create logic that will set the 'IR_interrupt' high when a new message comes in.

(g) Now add logic that allows the user to read the status of the interrupt and clear the interrupt using the following guidelines.

- For interrupt status and control capabilities, you may use the third software accessible register that was available for debugging in Lab 7.
- The lower 16 bits of the status/control register should be writable via the PLB (i.e. for control), while the upper 16 bits should be writable only by logic internal to the peripheral (i.e. for status).

- The interrupt signal should be connected to one bit in the status portion of the status/control register.
- Setting a certain bit within the control portion of the status/control register should reset 'Interrupt'.

(h) Once your source code modifications are complete, select the 'Ports and Interfaces' tab in the Package IP window. Click on the 'Merge changes from the Ports and Interfaces Wizard'.

(i) You should see the 'IR_interrupt' listed under ports. Right click on 'IR_interrupt' and select 'Interrupt' under 'Auto Infer Single Bit Interface'. The signal will be recognized as an interrupt when we connect it to the processor. When finished, make sure to re-package the IP.

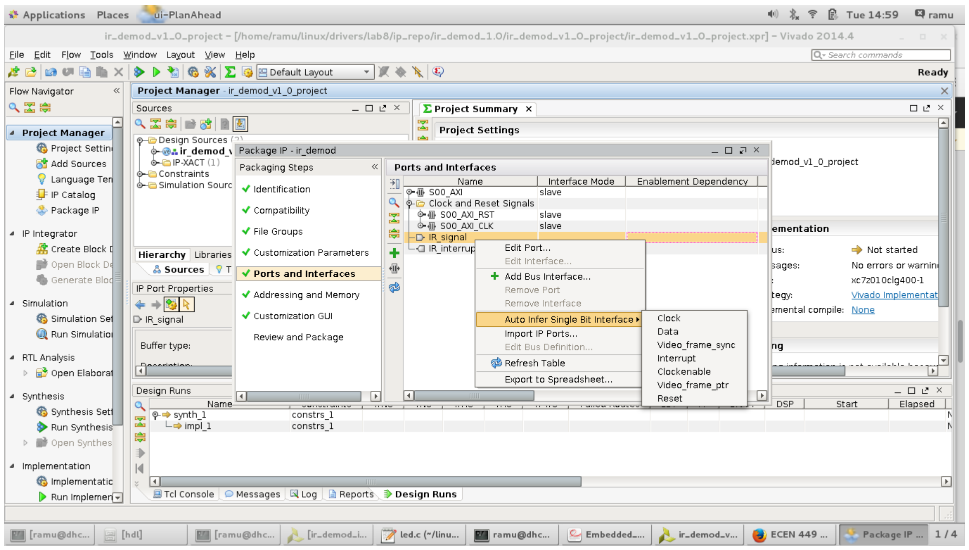

Figure 2: IR_interrupt

(j) The interrupt signal should be recognized by the processor when a positive edge is triggered and it will be specified in the device tree entry.

(k) Remove the current instance of 'ir_demod' from your system and add the updated 'ir_demod'. You should see the 'IR_interrupt' signal as an output in the block diagram in the list of ports within the system assemble view.

(l) Ensure the 'IR_signal' is externally connected as outlined in Lab 7 and make the 'IR_interrupt' external as well.

(m) Do not worry about connecting the 'IR_interrupt' signal to the interrupt controller for the moment. We want to test the operation of the signal without actually using it.

(n) Refer to the ZYBO reference manual provided on the course website to determine which FPGA pin is connected to pin U20 on the ZYBO board. Modify your XDC such that the 'IR_interrupt' signal is connected to this pin.

(o) Since you have copied the files from Lab 7, the bitstream and the SDK projects from Lab 7 are copied as well. Generating a bitstream in Lab 8 will create a new hardware file. Make sure to note the file name when you export your hardware and use the same hardware in the SDK.

(p) Modify your C program from Lab 7 such that the PS prints the value of the 'ir_demod' status/control register to the screen when it determines a new message has come in. Additionally, make your software write a one to the appropriate bit of the status/control register after your printf statements.

(q) Use the oscilloscope to view both 'IR_signal' and 'IR_interrupt'. Use Picocom to view the terminal output of your software.

(r) Ensure that the 'IR_interrupt' goes high when a new message comes in and that your software is able to clear the signal appropriately. Demonstrate your progress to the TA.

2. At this point, you should have a working 'ir_demod' peripheral with interrupt capabilities and you have tested the interrupt signal.

3. We need to connect the interrupt signal to the PS so that it can recognize it and execute the interrupt service routine.

4. 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.

5. 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.

6. With the hardware system from Figure 1 built, it is now time to write a device driver for our IR-remote demodulation hardware. Unlike the device driver you created in Lab 6, this device driver will utilize the interrupt signal added in Part 1 of the lab procedure. When using the Linux OS, most of the work for handling the interrupt is done for you. As a driver developer, you are only responsible for providing the usual read, write, open, close, init, and exit functions along with an additional routine referred to as an interrupt handler. The interrupt handler routine, as the name suggests, is the routine the OS calls when your hardware interrupt triggers.

   (a) Use the following guidelines and suggestions to create your device driver:

   - Use the 'irq_test' source code as a starting point for your device driver (you will find this source file in the 'module_examples' directory within the laboratory directory).

- Your device driver shall maintain a message queue large enough to hold 100 incoming messages (assume each message is 16-bits in length). The interrupt handler is responsible for placing incoming messages into this queue.
- Your interrupt handler is responsible for clearing the interrupt in your IR demodulation hardware.
- The memory allocation for the message queue must happen dynamically when the device is opened to avoid using unnecessary resources.
- Likewise, your device driver shall not register the interrupt handler until the device is opened.
- The device driver must only allow one process to open the device at a time, and it must do so using a semaphore to avoid race conditions.
- Writes to the device are not supported from user space. Your driver must print a message when a process attempts to write to it.
- Reads to the device must move messages from the queue to the user space buffer. Use the count variable as it was intended, not transferring more than the requested number of bytes. Return the number of bytes actually transferred. Reads to an empty queue should be allowed but return 0.
- For debugging purposes, you may want to program the read routine such that the last 4 bytes transferred to user space represent the message count.

(b) Before we load modules in Linux we need to create BOOT.bin, and also provide a hardware description in the device tree file.

(c) Create boot.bin as shown in Lab 4, and edit the .dts file. Include the following lines under the axi interconect and recreate the .dtb file.

```
ir_demod {
        compatible = "ecen449,ir_demod";
        interrupt-parent = <&ps7_scugic_0>;
        interrupts = <0 61 1>;
        reg = <0x43C00000 0x10000>;
} ;
```

(d) ‘scugic’ is short for Snoop Control Unit Generic Interrupt Controller which controls the interrupts and 61 is the interrupt id of the ‘ir_demod’ IP and 1 represents a positive edge triggered detection. Update ‘reg’ entry with the IP base address from the ‘Address Editor’ tab.

(e) Use ‘insmod’ and ‘mknod’ to properly install your device driver into the Linux System.

(f) Create a devtest application to test the operation of your device driver. Your devtest shall print the IR-remote messages to the screen similar to the test application in Lab 7. Use the ‘sleep()’ system call to avoid continually sampling of the device driver. Play around with various delay lengths to find the maximum amount of delay that could be tolerated by the user. Note that reducing the delay makes the system more responsive, but requires more calls to 'read'.

(g) Run the devtest application on the ZYBO board and demonstrate your progress to the TA when you have successfully completed parts (a) and (b).

Deliverables

1. [5 points.] Demo the working IR device driver to the TA.

   Submit a lab report with the following items:

2. [5 points.] Correct format including an Introduction, Procedure, Results, and Conclusion.
3. [4 points.] Commented Verilog and C files.
4. [2 points.] The output of the Picocom terminal.
5. [4 points.] Answers to the following questions:
   (a) Contrast the use of an interrupt based device driver with the polling method used in the previous lab.
   (b) Are there any race conditions that your device driver does not address? If so, what are they and how would you fix them?
   (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?
   (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?