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

Prof. Peng Li
TA: Andrew Targhetta
(Lab exercise created by A Targhetta and P Gratz)

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. Part of lab this week will involve the instantiation of an AC97 audio controller for next week's lab.

System Overview

The Linux compliant hardware system that you will create in lab this week is depicted in Figure 1. One of the notable features of this system is the use of two peripheral buses. In order to make use of the existing Xilinx audio controller, our system must include an On-chip Peripheral Bus (OPB). The OPB is a standard interconnect for low-bandwidth peripherals. As there is no direct connection between the MicroBlaze processor and the OPB bus, your system must include a PLB-to-OPB bridge. The bus bridge allows the MicroBlaze to address peripherals on the OPB seamlessly as though they were attached directly to the PLB. Thus, XPS must ensure peripheral addresses on the OPB do not overlap with those on the PLB.

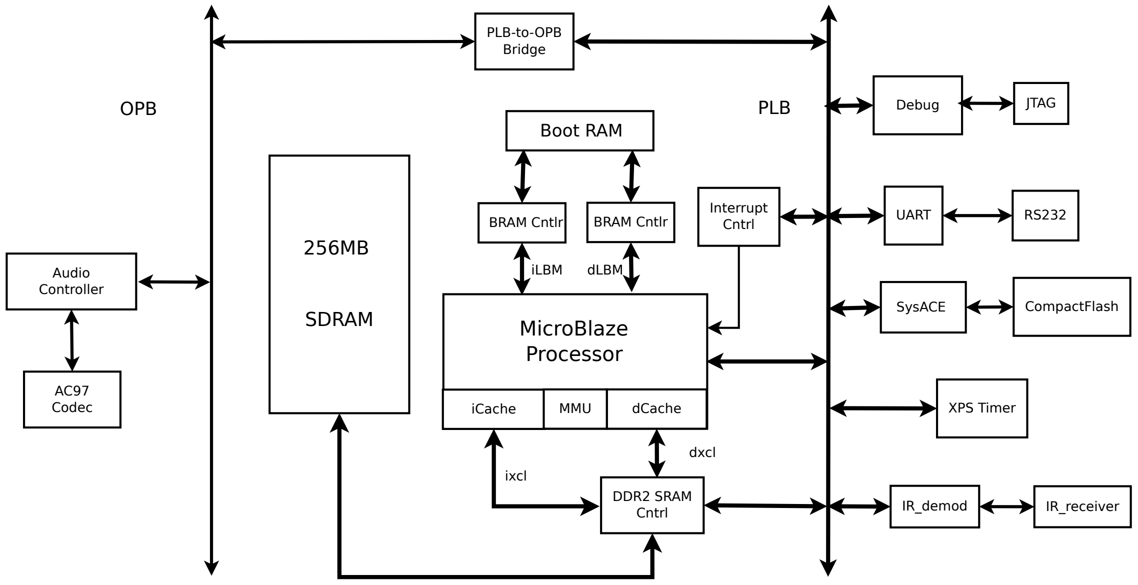

Figure 1: Hardware/Software System Diagram

Procedure

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 MicroBlaze. 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) Open your XPS project from Lab 7. Be prepared to modify the "ir_demod.vhd" and "user_logic.v" source files located under the 'pcores/ir_demod_v1_00_a/hdl' directory.

   (b) From within "ir_demod.vhd" and "user_logic.v", add a second user defined port called 'Interrupt'. Follow the procedure for creating '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.

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

(d) 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'.

(e) Once your source code modifications are complete, re-import your peripheral into your Lab 7 project. Be sure to specify 'Interrupt' as an interrupt signal. Also, set it up to be positive edge triggered. This means the interrupt controller will force the MicroBlaze to respond to an event only on the positive edge of 'interrupt'.

(f) Remove the current instance of ir_demod from your system and add the updated ir_demod. You should see the 'Interrupt' signal in the list of ports within the system assemble view.

(g) Ensure 'IR_signal' is externally connected as outlined in Lab 7 and make 'interrupt' external as well.

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

(i) Reference the ML505 board schematics provided on the course website to determine which FPGA pin is connected to pin 10 of J6 on the XUP board. Modify your UCF such that the 'Interrupt' signal is connected to this pin.

(j) Modify your C program from Lab 7 such that the MicroBlaze 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 <u>after</u> your printf statements.

(k) Use the oscilloscope to view both 'IR_signal' and 'Interrupt'. Use kermit to view the terminal output of your software.

(l) Ensure that '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. The following steps with help you create the system depicted in Figure 1.

(a) Create a copy of your Lab 4 directory, calling it Lab 8, and copy the '/homes/faculty/shared/E-CEN449/pcores' directory into it.

(b) Copy your 'ir_demod_v1_00_a' directory from Lab 7 into the 'pcores' directory of Lab 8.

(c) From within XPS, open your Lab 8 project and ensure version **1.01.a** of the OPB AC '97 controller and version **1.00.a** of your IR demodulation peripheral exist under the 'USER' category of the IP catalog.

An important note about version numbers in Xilinx: V1.01.a signifies a major version of 1 with a minor version of 01. Major versions imply significant design changes that usually influence the interface to the device, while minor version numbers typically imply enhancements to the device or bug fixes that do not effect the interface. The 'a' in the complete version number is referred to as a hardware/software compatibility designator and might be used to specify architectural variation in the design.

(d) Add the 'ir_demod' and 'opb_ac97_controller_ref' peripherals to your system.

(e) From the XPS main menu, click on **Edit→Preference...→IP Calalog and IP Config Dialog**. Then, enable "Display Available IP cores (including legacy PLB/OPB cores) in IP Catalog".

(f) Now add the 'On-chip Peripheral Bus (OPB) 2.0' and the 'PLBV46 to OPB Bridge' to your system. These IP blocks can be found in the IP catalog under the 'Bus and Bridge' category.

(g) In the System Assembly window, connect both bus connection to the PLB-to-OPB bridge. Also, connect your ir_demod peripheral to the PLB and the AC97 controller to the OPB.

(h) Assign a 64K address range to each of the peripherals just added. Regenerate the system addresses. Observe the addresses assigned to the AC97 controller and the bus bridge.

(i) Select the 'Ports' tab at the top of the system assembly window and expand the 'opb_ac97_controller_ref_0'. Select the 'Playback_Interrupt' and then click 'make external'.

(j) Also make the Bit_Clk, SDatat_In, SData_Out, and Sync signals of the audio controller external and add the following lines to your UCF.

```
  ### AC97 audio codec ###
Net opb_ac97_controller_ref_0_Bit_Clk_pin LOC = AF18;
Net opb_ac97_controller_ref_0_Bit_Clk_pin IOSTANDARD = LVCMOS33;
Net opb_ac97_controller_ref_0_Bit_Clk_pin PERIOD = 80; #12.5 MHz serial clock from codec
Net opb_ac97_controller_ref_0_SData_In_pin LOC = AE18;
Net opb_ac97_controller_ref_0_SData_In_pin IOSTANDARD = LVCMOS33;
Net opb_ac97_controller_ref_0_SData_Out_pin LOC = AG16;
Net opb_ac97_controller_ref_0_SData_Out_pin IOSTANDARD = LVCMOS33;
Net opb_ac97_controller_ref_0_Sync_pin LOC = AF19;
Net opb_ac97_controller_ref_0_Sync_pin IOSTANDARD = LVCMOS33;
```

(k) Likewise add a couple more lines to the UCF such that 'Playback_Interrupt is mapped to pin 'G32' of the FPGA using LVTTL IO standard.

(l) From the 'Ports' tab in the system assembly window, connect the interrupts to the interrupt controller using the following order starting with the lowest priority first:
RS232_Uart_1,
SysACE_CompactFlash,
ir_demod_0,
xps_timer_0,
opb_ac97_controller_ref_0_Playback,
opb_ac97_controller_ref_0_Record

(m) For the On-chip Peripheral Bus, connect the 'SYS_Rst' to 'sys_bus_reset' and 'OPB_Clk' to 'sys_clk_s'.

(n) Make the 'IR_signal' external and modify the UCF such that it connects to 'H32' as done in Lab 7. Do not make the ir_demod 'interrupt' signal external.

(o) Have the TA inspect your current system prior to performing a complete system rebuild.

(p) Update the system bitstream, and regenerate the software libraries. Consult Lab 4 if this is unclear.

(q) Recompile the 2.6.35.7 Linux kernel. Be sure to copy the new device tree generated in the previous step into the appropriate subdirectory from within the 'linux-2.6.35.7' directory.

(r) Recreate the system ace file and boot the Linux system to ensure everything is working so far.

(s) Run the following command in the XUP terminal window:
```
>cat /proc/interrupts
```

(t) Base on what you see from the above command and the order in which your interrupts are connected to the interrupt controller in "system.mhs", determine the IRQ number that your IR-remote device driver should use.

3. 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 Section 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) Use ‘insmod’ and ‘mknod’ to properly install your device driver into the XUP Linux System.

(c) Create a devtest application to test the operation of your device driver. Your devtest shall print the IR-remote messages to the screen similarly 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’.

(d) 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 XUP terminal (kermit).

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 the incorrect IRQ number when registering your interrupt handler? Would your system still function properly? Why or why not?