Alan Achtenberg

# Lab 8 Interrupt Based IR remote Device Driver

ECEN 449 Sec:505

Due: 12/5/2014

Introduction:

Lab 8 was a much more complicated version of Lab 5. We created an interrupt based Device Driver for our IR remote. This required us to add support for the interrupt in our user_logic.v as well as in our Xilinx system builder.

Procedure:

The first step in Lab 8 was to add support for our Interrupt in our demodulator hardware. To do this we added an interrupt signal to the netlist, as well as we set up our slv_reg2 to function as a control/status register. By preventing the software from writing to the half the register and preventing the hardware from writing to the other half, we can safely ensure that slv_reg2 can be used as a communication device between the software and the hardware. When the hardware detects a new message it sets a bit in the status part of the register. This allows the software to recognize and handle the interrupt. Once the interrupt has been handled. We set a bit in the control part of the register, which is then interpreted by the hardware and sets the interrupt back to low. This allows the register to handle the interrupt before receiving another one.

The second step in Lab 8 was to set up all the hardware required for the operating system as well as an AC97 controller. Many of the steps were the same as Lab 5 except for the specific hardware signals and peripherals that were added. We added a on peripheral bus that communicates between the AC97 hardware and the processor through the mplb. We also added our hardware decoder. We made all of the connections except for the interrupts external by connecting them to hardware pins on the FPGA.

The 3rd step of this lab was to write a device driver and user test file that will fully implement the interrupt driven driver. The device driver uses a buffer of 100 messages which are printed out by the

user test program.  The hardest part about this device driver is the need to synchronize both the acquisition of the device as well as synchronize the buffer.

Results: C Source Files

Conclusion:

Lab 8 was the most difficult one yet, however it is astonishing how simple the user program is. Getting this set up now will make many applications in the future much easier to implement.

Questions:

(a) The interrupt driver requires much more hardware capabilities than the original device, but it also gains several useful features. Number one being that it greatly reduces the cpu overhead in terms of polling the sensors.

(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 it does not address the race condition on weather the reader reads from the queue before the interrupt handler fills it up. Simply use a counting semaphore to prevent over filling the buffer.

(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? You are expected to complete the handler function code in a much smaller amount of time. Knowing this I would remove all print statements as well as unnecessary bookkeeping for debugging.

(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?
Whenever an IR interrupt would occur it would trigger a handler for another interrupt, such as the

system timer. Not only will the IR device driver not function, but it could potentially cause errors. For example if another handler relied on some hardware to allocate memory for it, before sending the interrupt signal.