

CPRE 488X

LAB02

Lab Conventions

The following conventions are used throughout the labs:



Location of additional information or tips.



An important error or step that you should be aware of.



A reference to a chapter in the book.

Lab Overview

For this lab, we will be expanding your project from the first lab to include interrupts. Interrupts, as you learned in earlier classes, can be used to optimize system performance. If a system doesn't have to poll a hardware device for its status, it can do other tasks instead.

Prelab



Please make sure you are familiar with sections **3.2.4** and **4.4.1** of the book, as many concepts from this section will be used in the lab.

Read through the lab and any necessary documentation. Look over the given `system.c` so you have a reference as to what functions will be used. Write down which functions you will be using and a short description of each in your lab notebook. Try to determine ahead of time which values you will be passing to each function, especially the arguments that are defined constants.

Also, several questions are posed in the lab. Try to answer the ones you can before lab and place the answers in your lab notebook. Final answers to the questions can be turned in once the lab is complete, but make sure you have at least some of the questions answered before coming to lab.

I/O and Interrupt system

New functionality is required of your system from Lab 1. Specifically, you need to display the time the program has been running down to the second. Features for which actions need to be very quick or very accurate usually require the use of interrupts.

Think about how you would design a piece of software that displayed the time accurately – perhaps you would use a `sleep()` function. However, this method is not accurate because it does not account for the time it takes to display the message over the serial port – if your baud rate was slow, it may be significant and noticeable immediately. If you think about it, even if your display rate was very fast, using a simple `sleep()` function will always result in a time drift. Now think about if you implemented the poor timer with a `sleep()` function and then were asked to add more time-dependent functionality; the software gets very complicated very quickly. With interrupts, the solution is much simpler and very accurate.

To improve our LED flasher from Lab 1, we will be adding an interrupt controller and several hardware cores. The LEDs will flash at a rate set by a timer. A timer built into the PPC440 CPU core, the Periodic Interrupt Timer (PIT), will be used to display the time accurately. Lastly, we will use an interrupt to handle the push-button input.

In fact, the only code in the `main()` while loop will be to display characters over the serial port (TeraTerm)! All the actions besides writing to the serial port will be driven by interrupts. A global boolean flag, called `terminal_update`, is used by the various interrupt handlers to communicate with `main()`. If a task needs to refresh the display, it sets `terminal_update` to 1.

In general, interrupt handlers should be as short (in execution time) as possible. **Why might writing to the serial port in the interrupt handler be a bad idea?**

Part I: Add hardware cores

In order to complete this design, we will first need to talk about how the Xilinx XPS tool handles the addition of hardware to a system. Adding hardware actually implies a hardware component and a software component that your program uses to interface to the hardware component.

In the Xilinx documentation, you will see the word “core”, “pcore”, or “IP”. This refers to a hardware module, usually written in VHDL, that can be added to a system through XPS. Each hardware core also has an associated “driver”, which refers to a library of software functions made specifically to interact with a particular hardware core.

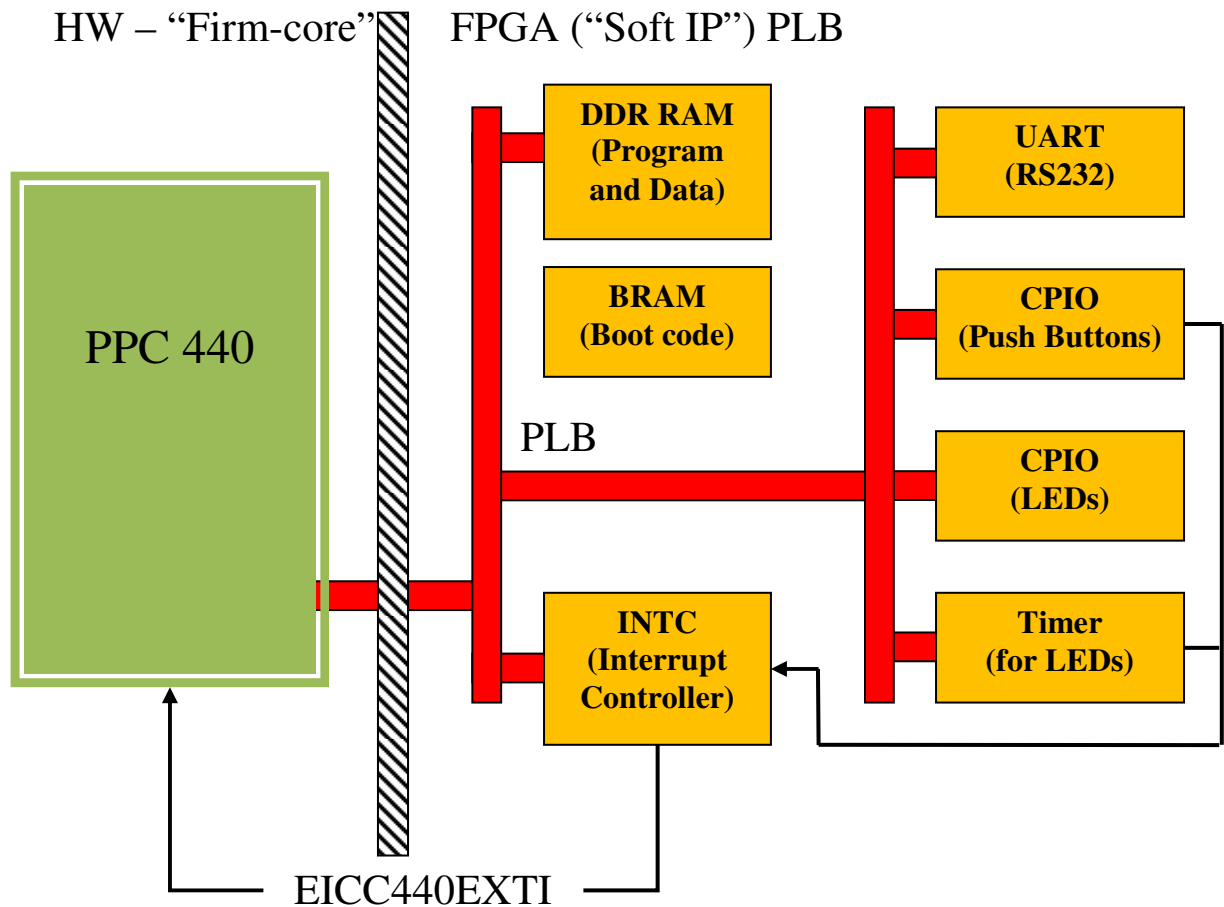
In this lab we will be using three new hardware cores and their associated software drivers. The following is a short description of each type of module:

Name	Function Prefix	Description
Gpio	XGpio	Software drivers to interact with a “gpio” hardware module. GPIO stands for General Purpose Input/Output. This type of hardware module is used to access the LEDs, push buttons, and user DIP switches. It could also be used to interface with simple hardware you create. It provides only a simple interface of reading and writing, along with optionally providing an interrupt signal.
Intc	XIntc	Software drivers to interact with a “intc” hardware module. Intc is an abbreviation of Interrupt Controller. The interrupt controller hardware module consists of up to 32 interrupt inputs and a single interrupt output. Interrupts can be configured as rising/falling edge or high/low level triggered. Read more about interrupts in the book in section 3.2.4 .
Tmrctr	XTmrCtr	Software drivers to interact with a “tmrctr” hardware module. TmrCtr is an abbreviation for Timer/Counter. This hardware module is a simple counter which can count up or down and generate an interrupt when the count reaches user-defined values. The uses of timers are briefly discussed in the book in section 4.4.1 .



The first step in adding interrupts to your project is to add an external interrupt controller.

To understand why an external interrupt controller is necessary, we must look into the architecture of the PowerPC 440 core within the Virtex-5 FPGA.



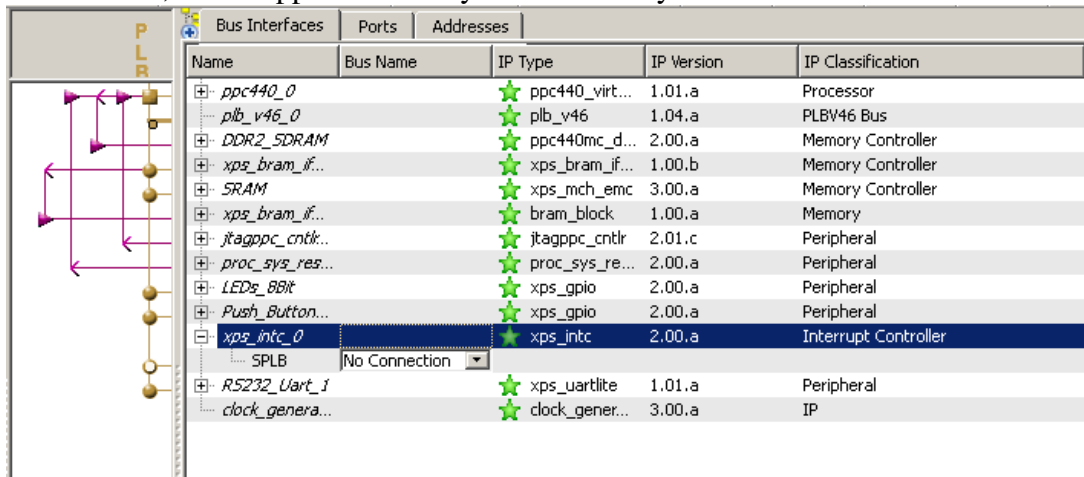
The PowerPC 440 block has two external interrupt signals: one is called the external or non-critical interrupt, and the other is called the critical interrupt. A critical interrupt can interrupt a non-critical interrupt; however a non-critical interrupt cannot interrupt a critical interrupt. This allows for a 2-level hierarchy of interrupts. We will only be using the non-critical interrupt, but some real-time or deterministic systems may need to use this 2-level scheme to achieve strict timing requirements for interrupt latency.

As shown in the picture above, it is very possible you will want to connect multiple HW resources to the PowerPC core interrupt input. However since we are restricting ourselves to the non-critical input and there is only one input, we need some external hardware. The INTC core allows up to 32 devices to connect their interrupt signals to it, and it generates a master interrupt signal to send to the PowerPC.

Open your project in Xilinx Platform Studio.

First, we need to add an **Interrupt Controller**.

On the *Project Information Area* select the “**IP Catalog**” tab. Click the plus sign next to “Clock, Reset and Interrupt” to see all available IPs of under this category. We are looking for the *XPS Interrupt Controller*. To add it right click and select “Add IP”. Once the IP is added, it will appear on the System Assembly View.

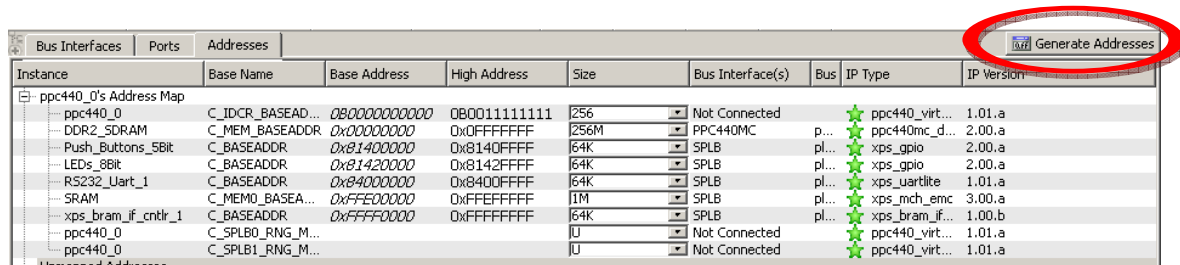


Notice that the new IP is not connected to the bus. Hence the empty bubble under PLB to the left. To connect the new IP to the bus click on the empty bubble, or you may select “plb_v46_0” from the drop down menu on the SPLB port.

The PowerPC accesses all devices through memory-mapped I/O. Click on the “**Addresses**” tab to view the current memory map for the system.

**Where is the SDRAM located in memory (this is where our program will be stored)?
Where is the BRAM (memory internal to the FPGA)?**

To update the memory map so the opb_intc core is included, click on the “**Addresses**” tab and click on “**Generate Addresses**”. This will automatically generate a valid memory map for your system, although if you needed to you could still edit it manually.

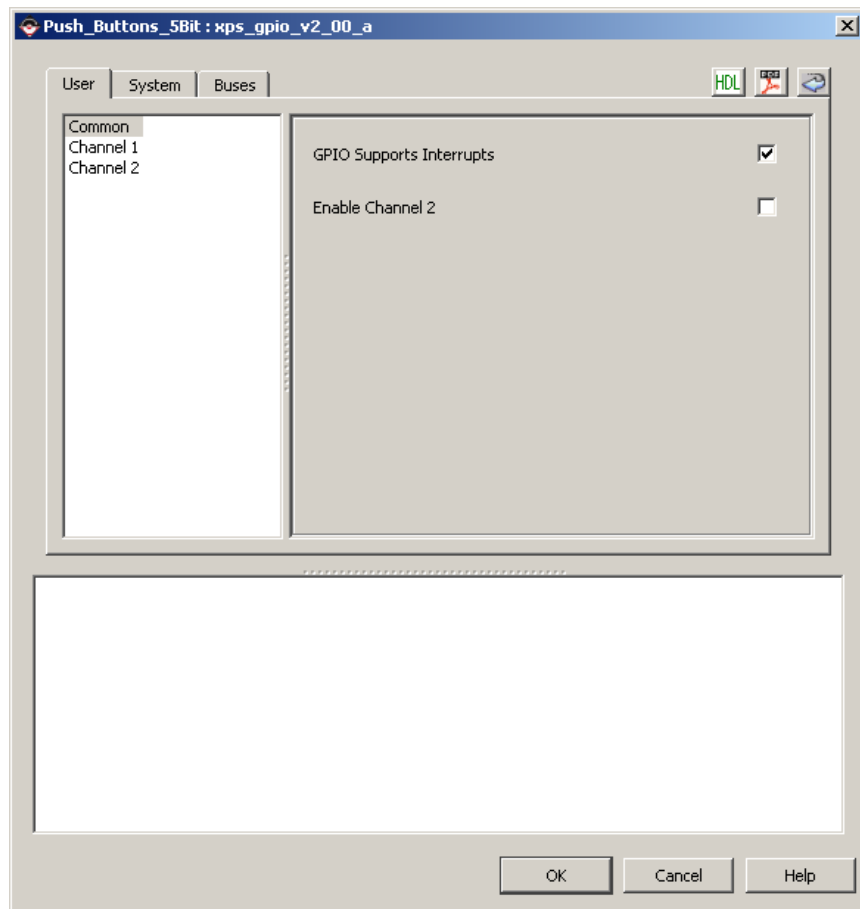


After generating the address space, click on the “**Ports**” tab. Here we will connect specific ports for the **xps_intc_0** core, and we will also need to connect the interrupt line for the Push Buttons. Refer to the architectural diagram at the beginning of the lab if you'd like to follow along as we connect the interrupts to the processor.

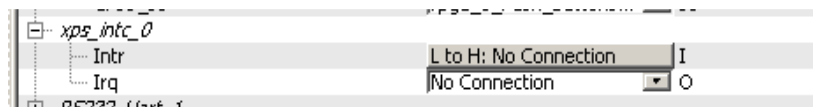
Inspect the ports on the new **xps_intc_0** core. The two ports on the INTC core are:

Intr	A 32-bit wide input to the INTC core, where each bit is an interrupt from a core
Irq	The master IRQ (Interrupt ReQuest) that is connected to the EICC405EXTINPUTIRQ input on the PowerPC core

Let's connect the **xps_intc_0** and Push Button ports. The first step is to enable interrupt support for the Push Buttons. To do this, right click on the **Push_Buttons_5Bit** core and select "Configure IP...". In the new window that comes up check "GPIO Supports Interrupts" and click OK. Now back on the System Assembly View you should see a new port for the push buttons called "IP2INTC_Irpt".



Now to connect the push buttons to the interrupt controller select the "Intr" port on **xps_intc_0**, by clicking on the box labeled "L to H: No Connection".



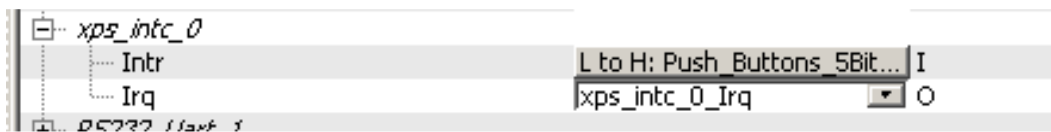
A new window comes up where you see the list of IP blocks that are available for connecting to the interrupt controller. From this list select the push button and move it to

the right using the arrow in the middle. On the left is the list of IPs connected to the interrupt controller in order of priority.

Click **OK**.

Now the port “IP2INTC_Irpt” in the push buttons should appear connected in the System Assembly View.

Now the INTC and the input to it are configured, we just need to connect the output of the INTC to the PowerPC non-critical (external) interrupt input. To do this, open the drop down menu on the “Irq” port of **xps_intc_0** and select “New Connection”. Immediately after you select new connection a new signal is created and the port value updated. You should have something similar to:



The name of the signal may be different.

Now look at the ports of the ppc440_0 IP and find EICC440EXTIRQ. Open the drop down menu of this port and select the signal you just created.

We are now finished connecting the push buttons to the interrupt timer. Using the previous process, do the following:

- Add an **XPS Timer/Counter** core, under DMA and Timer, to the system and connect it to the bus. Change the instance name to “**xps_timer_leds**” to avoid confusing it with other timers.
- Connect the **Interrupt** output port of the timer to the **INTC** device.

After you are finished with the hardware changes select the “**Project**” tab in *Project Information Area*. Open the Microprocessor Hardware Specification (**MHS**) file. The MHS file is automatically generate by in Base System Builder and updated every time you add a new core. It lists, gives parameters to, and connects the different hardware components in the system. The MHS file also defines external ports.

Write down the lines that connect the pushbuttons and the led timer interrupt to the interrupt controller, and the interrupt controller to the processor. Also write down the baud rate of the serial connection for TeraTerm. Put these in you Lab Notebook.

Another noteworthy file generated in Base System Builder is the Universal Constraints File (**UCF**). This file connects all the external ports to actual pins on the FPGA. It will add constraints to certain pins and settings of the system. In other words, all pin

assignments are automatically chosen unless the pin is listed in the UCF, then it is given its assigned value.

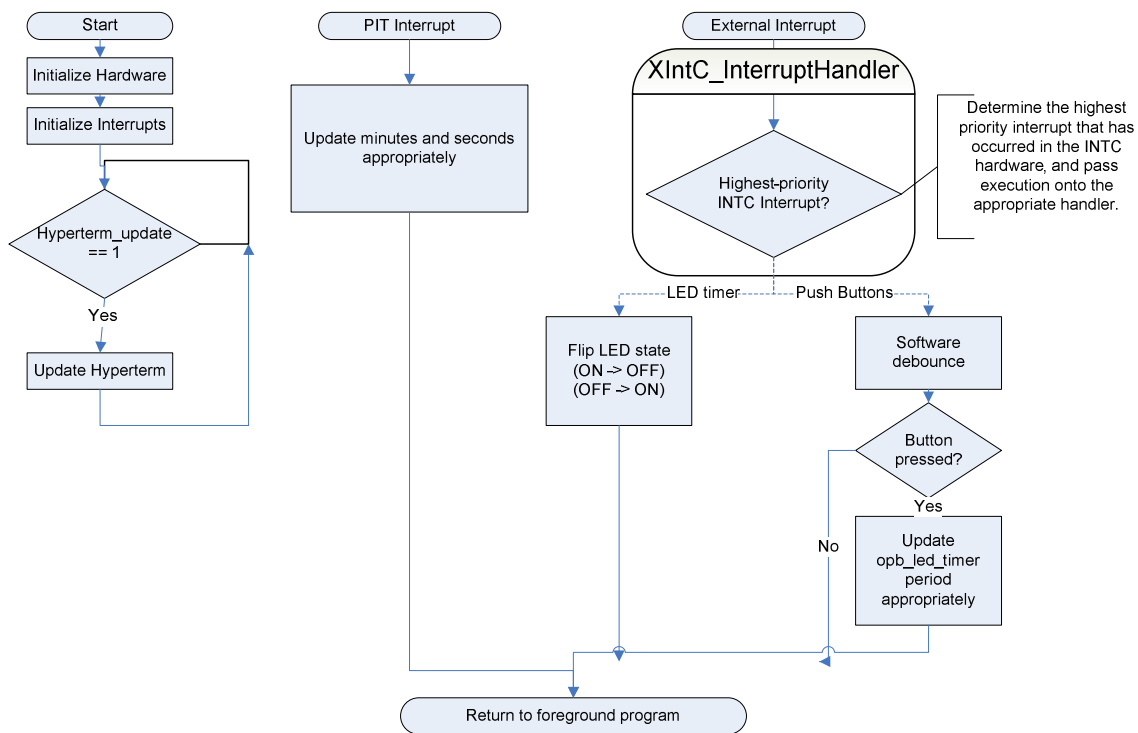
When you make hardware changes you will want to be sure that all of the addresses for your system do not overlap. As a final step, go to **Add/Edit Cores...** and **Generate Addresses...** one more time.



Make sure you re-synthesize your hardware before continuing, and download the new hardware to your board. You will also need to generate new libraries for the new hardware (Software -> Generate Libraries and BSPs). If you do not do one or both of these, you will not be able to proceed with adding functionality to the system.

The architecture for our system is now complete. Now we must write the software code to run our system. A sample system.c is provided with the basic structure of the code you will need.

The basic structure of system.c is:



Implement the system so the time is displayed and updated every second, and the flashing delay is displayed and updated when changed on the HyperTerminal. The LEDs should flash at the current “delay” rate (in ms), and all actions should not affect any other action occurring on the board. The portions of the code which you need to update are marked with a “TODO:” comment.

Remember: If you need information on a particular software function, instructions on how to find help are found in the class website for Lab 1.

Draw a UML sequence diagram of our system. See the examples shown on page 118 or page 123 of the textbook. It should show a typical user interaction with the system. Turn this diagram in with your lab notebook.

Demonstrate the complete system to your TA. Print off a copy of your source code and turn it in with your lab notebook.