LAB6: Timer and automatic variables

Please demonstrate your work on Tasks 2 and 3 to a TA by Friday, August 2nd, and also upload your modified files to CourSys.

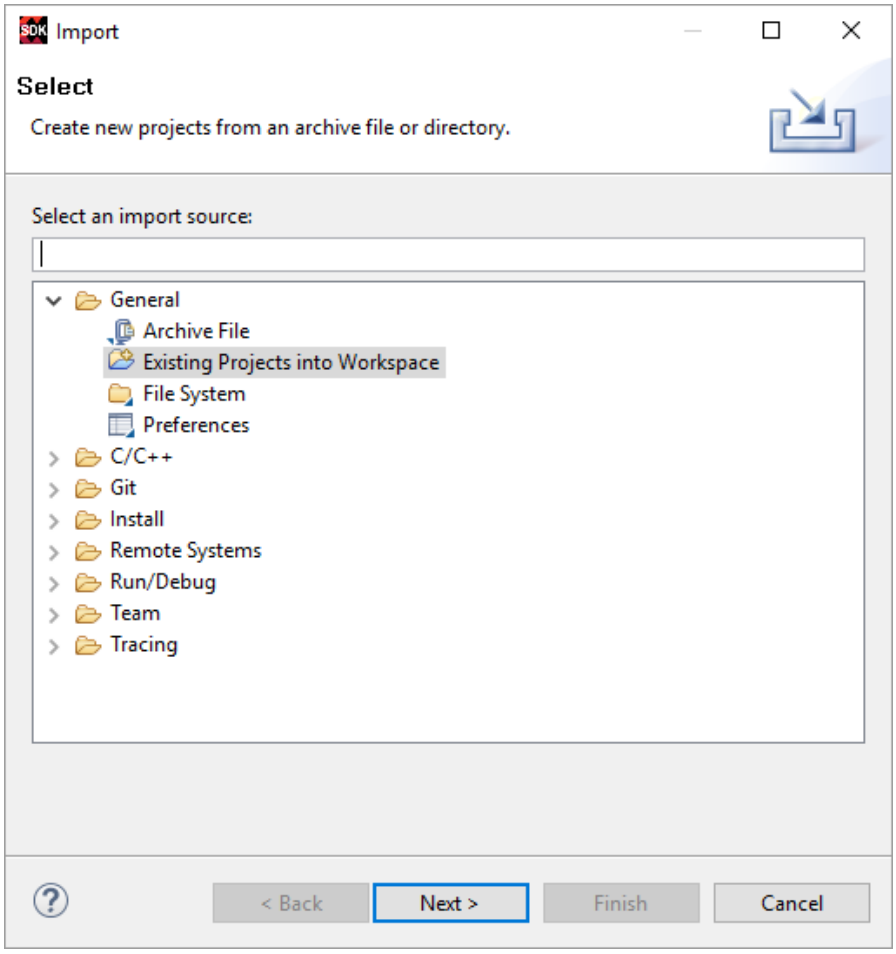
So far we've seen interrupts used to alert the CPU to a sporadic external event like button activity. In this lab we'll be looking at using interrupts for a second common task: helping the processor to keep track of time. We will also be looking at automatic variables and how they are sometimes more appropriate than variables with a static duration.

1. Importing the Lab6 Project

A Lab6 project has been created with some starting code included. Download the lab6.zip file from Canvas and import it into the *U:\XilinxWorkspace* workspace that you used for Labs 4 and 5 as follows:

File -> Import

Select 'Existing Projects into Workspace'



Choose ‘Select archive file’ and browse to your downloaded copy of lab6.zip. Select ‘Finish’.

A new Lab6 project should appear in the Project Explorer pane next to your previous projects. Inside you’ll find five source files: ‘main.c’, ‘asm_main.S’, ‘asuAdd.S’, ‘OLED.S’, and ‘fib.c’ . If you have any trouble importing refer to Lab 5 for more detailed instructions. The import process is the same with the exception of the project and file names.

The ‘main.c’ code is identical to the code in Lab 5. The ‘asm.S’ code is where we’ll be starting in this lab. It contains the code necessary to output a counter variable that periodically increments. The incrementing of the variable is triggered by a counter/timer component that we have instantiated inside the FPGA.

The button code in ‘asm.S’ is similar to the button count code you worked on in Lab 5, but all buttons now decrement the counter variable. We’ll be modifying it later to use a timer counter to help ‘debounce’ the button activity.

The ‘fib.c’ code is, as in previous labs, used to give the processor something to do in its main loop so we can demonstrate the usefulness of interrupts in allowing occasional work to interrupt a background task.

Build and launch the project and verify that a counter appears on the OLED and LEDs and increments every second. If you then press a button, the counter should decrement according to the amount of button activity.

2. Exploring the Timer/Counter

Some CPUs and most microcontrollers contain one or more counter circuits that can be used for keeping track of time or even measuring the time between two external events. The counter circuits operate independently from the CPU and typically run at some fraction of the CPU clock. Since these counters run independently using a known clock frequency they can be used to calculate the passage of ‘real time’ simply by reading the current count values at two different times and calculating the difference. Multiplying the count difference by the counter clock period determines the time that has passed between the count sample events.

CPU counter circuits can also be configured to generate interrupts when they reach a desired count. This allows a software designer to have the CPU interrupted after the configured amount of time has passed. For example, a programmer could configure a counter to generate an interrupt after 1 second knowing that the CPU could then freely execute other tasks and be interrupted automatically at the requested time.

Timers can often be programmed to reload automatically as well. This allows the timer/counter to generate a ‘periodic interrupt’ at a specified time interval. This is what the timer/counter code in ‘asm.S’ does as we have given it to you. It configures a timer to generate an interrupt once a second so that an incrementing count can be displayed.

For this exercise we are using a timer/counter programmed into the FPGA. Documentation for the timer (as well as the gpio device used for switches and LEDs) can be found on Canvas under the Xilinx Reference files in the Files area. In addition to the timer Overview, you will want to read the “Register Space” section starting on page 11 of the .pdf file. Tables 2-5, 2-6, and 2-7 will be particularly relevant.

Timer auto-reload:

When the Timer Counter Register reaches zero, it can generate an interrupt if an interrupt is enabled.

If ‘Auto-Reload’ is enabled, it will then load the value in the Load register to the Timer Counter Register and restart the countdown again. If ‘Auto-Reload’ is disabled, the timer will stop upon reaching zero.

Your first task: read through the ‘asm.S file and ensure you understand how it works. Once you understand how the 1-second interrupt is being generated, experiment with modifying the counter interrupt to occur at different intervals (like half a second or 100 milliseconds). Finally, set the interrupt frequency for 50 milliseconds.

3. Using the Timer to Debounce the buttons.

Notice that a single button press/release can sometimes trigger multiple increments due to something called switch bouncing. If you don’t notice any bouncing, ask some of your lab neighbours if it is noticeable on their kits. Even if bouncing is infrequent on your kit, it is still a problem to be fixed because if you hit the button enough times it will most likely bounce eventually and the problem may get worse over time or possibly even due to changes in things like atmospheric conditions.

Figure 1 on the next page gives some examples of switch bouncing. The two parts of the figure are from supposedly identical switches.

Switch bouncing can be solved in hardware, software, or some combination. Our hardware has not been configured to help us with button bouncing, so we must solve the problem in software. One way to do this is to ignore any switch transitions from a certain button for a given period of time (up to 50ms or so for small poor-quality switches like the ones on our ZedBoards) after the switch first indicates that someone has started pressing it or releasing it. As we ask you to think about, it is best to use a timer to time the period during which transitions will be ignored from a switch.

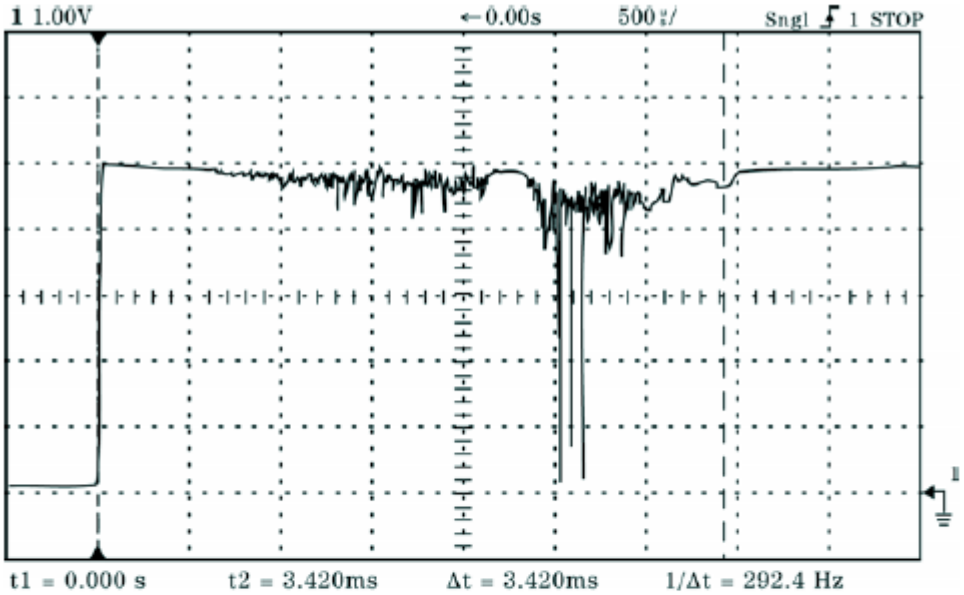


Figure 1- Switch bounce (going from off to on)

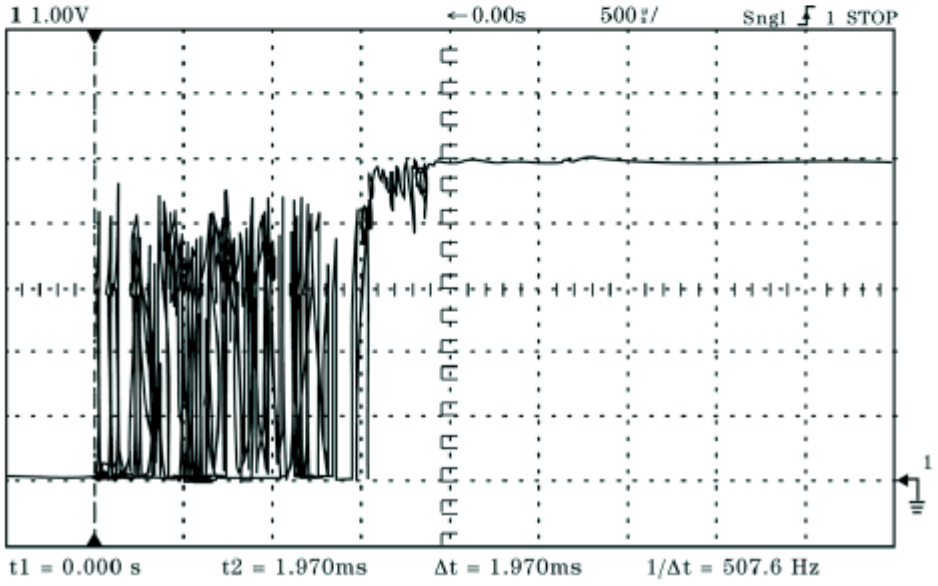


Figure 1 – Switch Bounce for Two "Identical" Switches

from http://www.educatorscorner.com/media/Exp17e.pdf

Your second task: modify 'asm.S' to use a timer interrupt to debounce button presses and releases. What is the advantage of using a timer for debouncing instead of a busy loop (e.g. a simple 'for' loop that does nothing significant and may loop many times) to create a delay?

De-bouncing method. The push-button switches on the ZedBoard seem of particularly poor quality. If you press down on a button and continue pressing, and then vary pressure or move your finger, additional electrical glitches can occur. The switches seem to be "normally open". Had "normally closed" switches been installed on the ZedBoards, then moving one's finger while pressing on a button should not have caused such glitching. For our lab, we will generally avoid holding a switch down, though such holding might be useful if stepping through code such as an interrupt service routine in the debugger. For debouncing, we recommend that you at least start with the following very simple scheme. Consider that only one button will be pushed at a time, and that a button will be released immediately after pressing it. When an initial button interrupt occurs, process that button interrupt but temporarily dismiss further button interrupts and start a "one shot" timer. If the button was pressed, **increment** and display a button-press count. After the timer expires, don't dismiss the next button interrupt. Dismissing a button interrupt means acknowledging the interrupt (i.e. resetting the interrupt) but not actually processing the button activity.

To complete this task you will likely need to:

- Modify timer initialization code in asm_main. For example, we will no longer need auto-reload as we will now be using the timer for debouncing instead of for generating periodic interrupts.
- Modify our 'FIQHandler' in asm.S as described above.

The original timer delay value provided was 1 second, which is much longer than desired (i.e. if you press a button twice within 1 second, the counter only increments once). After possibly confirming this, try reducing the delay time and see how long of a delay is enough to stop the button from bouncing while not limiting the speed of pressing a particular button.

All the timer "ingredients" that you will need, and more, have been demonstrated in the one-second periodic timer interrupt code that was given out. You will need to rearrange and discard these ingredients as appropriate. Definitions of the form XTC_* can be found in the file xtmrctr_l.h, included via asm_include.h. Definitions of the form XGPIO_* can be found in the file xgpio_l.h.

Note: I have noticed some mysterious and spurious button interrupts while button interrupts are disabled. This includes while debugging the FIQHandler in some solutions to this lab. If you get the same and they are of concern to you, consider asking Craig or Remy about it. We have investigated this issue for considerable time, and haven't been able to get rid of the spurious button interrupts when button interrupts are disabled. Though it may be elegant to disable button interrupts in a debouncing period, it seems we will have to rely on simply dismissing button interrupts instead of temporarily disabling them.

4. Automatic Variables and re-entrant code.

You will notice that I am modifying the SP near the top and bottom of FIQHandler. At the top of the routine, I subtract some amount from the SP to allocate space on the stack for automatic variables local to *FIQHandler*. At the bottom of the *FIQHandler*, I add the amount back to the SP to remove the automatic variables from the stack. The addresses of the two word-length automatic variables, initialized to zero, can be given to the *asuAdd* subroutine to add two zeros together.

You will notice that I ended up with a variable used by *asuAdd* in the data section of the file asuAdd.S. This variable was not labelled as global in the assembly file, so I guess we should refer to it as a file-scope static

variable. If the truth be told, I forced that variable into the solution for *asuAdd*. A smarter use of registers allows the subroutine to avoid having a variable stored in memory. Still, some subroutines with many variables will exhaust the ability to store variables in registers. If a subroutine needs to have a variable stored in memory, note that a static duration variable, like we currently have in *asuAdd*, can render the subroutine non-re-entrant. A re-entrant subroutine can be "re-entered" while one activation of the subroutine is in progress. Calling a subroutine from both inside and outside of an interrupt service routine can result in a subroutine being re-entered. Remove the comment symbol and allow *asuAdd* to be called from near the top of *FIQHandler*. Now build and run the program and generate a sequence of interrupts. Do you eventually end up in an error condition near the bottom of *asm_main*? If so, this is because *fib_main* produced an incorrect answer. Why exactly do you think that *fib_main* produced an incorrect answer? (Hint: we can't blame the Xilinx Tools or Digilent buttons for this particular problem.)

Your third task: modelling after the use of automatic variables in *FIQHandler*, in *asuAdd* convert the variable *holdMaxN0Size* from having static duration to being an automatic variable. Now build the project again, run, and generate more interrupts. Are you able to avoid getting into that error loop with this change to the *asuAdd* code?