LAB HW7:
Interrupts and the Timer

Expanded July 23rd

This week we will continue working with the simulated STR730 EVALBOARD, and will start programming more hardware on the board. In particular, we will be programming the Enhanced Interrupt Controller, a Timer, and learning more about low-level programming of the ARM processor.

First, every student should individually upload their work from HW6 to CourSys. This is just to have a record of your work. Your work for HW6, HW7, and HW8 will be evaluated during the Practical Exam at the end of the course.

Exit Keil uVision if it is open and make a copy of the directory holding the uVision project where you got HW6 working. Name the directories so that you know which is HW6 and in which directory you will start working on this HW7.

Replace the file BigFib.c in your directory for HW7 with the solution on Canvas (in the HardwareLabs folder), and uncomment the line in main.c that will calculate a Fibonacci number using the function FibCalc(). With these changes, the program calculates a large Fibonacci number before going on to the LoopFnc() that you worked on last week. While it is calculating the Fibonacci number, it is not doing anything else that is noticeable. Last week in the lab, you wrote some code to strobe simulated LEDs back and forth. The goal for this week's lab is to be able to efficiently, and as smoothly as possible, strobe the LEDs at the very same time that the processor is focused on calculating the Fibonacci number. We do this by making use of a Timer Circuit within the microcontroller that can periodically interrupt the microcontroller when it is time to "move" a LED.

Run the program and calculate the Fibonacci Number. You can calculate a smaller number if it takes too long for the number currently requested in the code. Before and after the Fibonacci number is calculated, heap statistics are collected and printed out. Make sure that the LEDs start strobing after the Fibonacci calculation has finished.

Next, you need to write code to initialize Timer0 (TIM0) and the Enhanced Interrupt Controller (EIC). We would like a fast FIQ interrupt to be triggered periodically starting as soon as the board has finished initializing. With this microcontroller, triggering FIQ with a Timer generally means that the

Timer must be Timer0 (TIM0). The other Timers are not routed to the FIQ interrupt. Please look at Chapter 7 on Interrupts and Chapter 14 on Timers from the "RM0001 Reference manual" for the STR73x microcontroller family in the ST reference files area on Canvas (the same file that was referenced in HW6). Section 7.1.2 informs you that there are actually two different types of interrupts that you can use with FIQ. Section 7.2 is also relevant, as is the top of page 86.

Timers like TIM0 are described in Chapter 14 of the manual. These TIM Timers can be configured in many ways. To generate periodic interrupts, I suggest that you use the Pulse Width Modulation (PWM) mode described in Sections 14.3.10 and 14.3.11. For our simple interrupts, we can actually use the PWM circuitry in one of the simplest ways possible. For example, when initializing TIM0, we can just leave the OCBR and OCAR registers in their default startup/reset state, where they both contain the value 0x8000 (if you want to make sure that these registers have that value then that is okay, and if you want to just trust that the startup/reset value will always be consistent then I am okay with that as well though you might want to indicate such in a comment). Also, we only need TIM0 to generate interrupts, so we don't need to worry about the OLVLx output levels or the OCAE bit.

We should not be using an external clock, so we need to reset the ECKEN bit (there seems to be a typo in the manual where they call it the ECK***G***EN bit in section 14.3.11 and perhaps a few other sections). You can adjust the prescaler division factor in bits (CC7 to CC0) to suit your needs. You might want to start around a value of 0x40. If you are working with a very slow computer, you might need to increase the prescaler factor. As mentioned above, I suggest that we will use OCBR == OCAR. A square waveform will be generated in this case according to Note 7 on page 168 and the middle portion of Figure 44. Note that when OCBR == OCAR, Figure 43 is no longer so accurate as the counter will be reset to FFFCh when the counter reaches OCAR as well as when the counter counts all the way back up to the identical value in OCBR.

All I/O registers for each type of peripheral on the chip are described in the reference manual. For each register for each type of peripheral, an Address Offset value from the base address of an instance of the type of peripheral is provided as well as the value taken on by the register when the microcontroller is Reset. See, for example, section 14.6. Note that a lower-case 'h' behind a number means hexadecimal. So for example, 8000h means 0x8000. This use of 'h' comes from older assemblers and such things. I believe that most newer tools and documentation follow the C-language convention of prepending a hexadecimal number with '0x'.

A handy way to see what is going on with the STR73x peripherals, and to experiment with them, is to use the Peripheral windows in the uVision simulator/debugger. You can close the "UART 0" and "Port 6" peripheral windows if you want, but strongly consider opening the "Timer 0" Timer window and the "Enhanced Interrupt Controller" Interrupt window. By experimenting with these windows, and setting and resetting certain bits, you might be able to figure out a good and minimal way to configure these peripherals.

Once you have decided how to initialize TIM0 and the EIC, you will need to make sure that your FIQ handling code is written and placed in the FIQ_Handler subroutine. The FIQ_Handler is called by the FIQHandler routine in the file FIQ.s in the set of files with the label "Library". Subroutine *FIQ_Handler* will have the job of "moving" a simulated LED to give a strobing effect. Sometimes FIQ_Handler will move the LED to the left when it is invoked, and sometimes the FIQ_Handler will need to move the LED to the right when it is invoked, so you will need to store somewhere the current direction of movement. Remember that the FIQ_Handler runs in the FIQ mode of the ARM Processor, and the FIQ mode is unique in that it has private copies of registers R8 through R12. You can use one or more bits in one of these registers to store the current direction of movement. If you want to initialize any of these 5 registers, you can do so in the subroutine FIQ_Init. FIQ_Init is called just after the startup code initializes the FIQ Stack Pointer and while the startup code is still running in the FIQ mode. Make sure that FIQ_Init leaves R0 untouched, as R0 is holding an important variable used in initializing the stack pointers.

One of the main things to remember about the FIQ handler is that you will need to clear the interrupt being serviced in a couple of places. You will need to clear the Output Compare Flag B (OCFB) in TIM0, and you will need to clear the appropriate FIQ Pending Bit in the EIC.

There are a number of include files, such as asm_include.h, 73x_tim_l.h, and 73x_eic_l.h, that contain definitions that will be helpful to you. We suggest you make use of these definitions.

Once you have written your FIQ handler, you can remove the code from the LoopFnc subroutine that deals with GPIO.

This lab is going to take significant patience to work on, and careful reading of the reference manual. Please consult Piazza as you are working on the lab. Piazza has a pretty good search function. If your question has not already been asked, feel free to ask the question yourself.

I might have a few tasks related to the above if you finish early and are bored!

Craig