Lab #12 – Timer Interrupts

Goals:

Write an assembly code function on the Discovery Board to handle update interrupt requests from Timer 3

Simulate a rudimentary multitasking operating system with an adjustable task switching time slice

For each step enter the necessary responses into the Canvas assignment text entry tool to let the TA follow your work.

Refer to the lecture slides for the following:

Part 1:

a) Download Lab12-PrimeRotateMain.c and Lab11-clock.s from the Canvas files area, and in the LabCode folder. Our goal is to set up Timer 3 on the STM32F429ZI to interrupt the main program thread periodically and cause a switch between its two computational tasks. First study the main program code. Find the two functions defined near the top that perform different example computation tasks, one a search for a prime number and the other a graphical rotation a triangle. Then trace in the main program how the value of a variable '`task`' is used as a task index, so that the selection of which computational task in the functions that is allowed to use processor execution time is determined by the value of the task index variable, either 0 or 1. Then locate the declaration of the variable '`task`' as an ***external*** variable at the top of the main C code, and how the variable is defined in the .bss segment in the assembly code. (The linker will provide the memory location of this variable in the assembly module to the C module at link time.) Note that this variable is also declared in the C code as 'volatile', which means the C compiler will assume its value may be changed by the external interrupt handler at any time. Also locate the call in the main program to the function `setupTimer` that is defined in the assembly source file.

b) Study the Week 15 Timers class slides 8 through 18 for the memory-mapped registers for Timer 3 that need to be configured in the assembly code. Trace through the `setupTimer` function and note how it sets these various memory-mapped registers controlling Timer 3 operation. Enter into the Canvas text entry the source file line numbers in the supplied assembly code that set the:

Timer 3 peripheral clock enable
Timer 3 prescaler value
Timer 3 auto-reload value
Timer 3 update interrupt enable
Timer 3 global interrupt vector enable

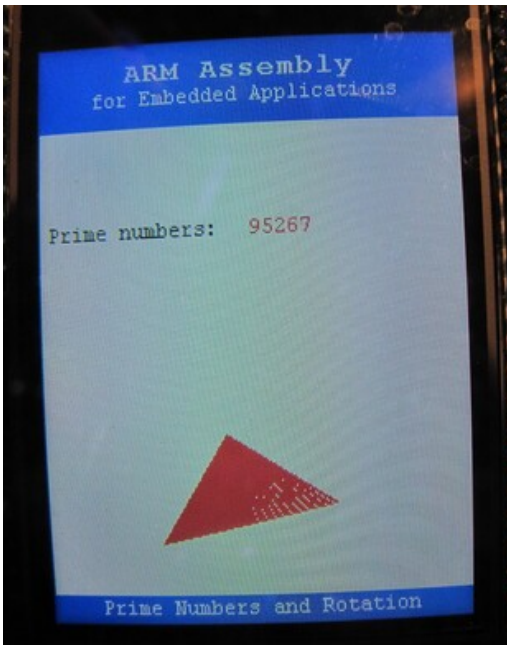
Where does the value that is stored into the Timer 3 auto-reload register originate?

c) Write the assembly code for the Timer 3 interrupt handler routine. This is the function at the bottom of the source code file with the required name '`TIM3_IRQHandler`'. The linker will ensure the address of this function is filled into the vector for Timer 3 in the system interrupt vector table. This short function must accomplish two items before returning from the interrupt with its '`BX LR`' instruction:

Clear the UIF flag (bit 0) in the `TIM3_SR` status register to 0, as shown in class slide 14. This flag bit is set to a '1' by hardware when the timer "updates", or when its counter has reached the value programmed into the auto-reload register. The flag must be reset to '0' by your assembly code to prevent the interrupt from being triggered again immediately after finishing the handler routine, and to therefore require the timer to count another update interval before issuing its next interrupt.

Invert bit 0 in the '`task`' global variable. That is, turn a '1' bit to '0' or a '0' bit to '1'. This variable is read in memory by the main C program whenever a portion of its task is complete, and the value of this variable will determine which of its tasks the processor will spend cycles working on.

You can of course make use of the symbolic address constants defined by the `.equ` directives at the top of the assembly source file in making memory-mapped references in your handler code. Refer to the code samples in the Week 14 lecture slides on configuring I/O pins for hints on how to code these bit operations on these memory words.



d) Design a value for the Timer 3 auto-reload register that will cause interrupts to occur every second, and use that value by editing the C main program. Refer to the class slide 8 to see how the rate of "update events" that trigger the interrupts is determined. Build and run your code. Observe the two computational tasks report their results to the Discovery Board screen, and alternate with one-second time slices allocated by this "operating system". Upload your code and a cell phone video clip.

e) Repeat part d, but with a different value for the Timer 3 auto-reload register designed for time slices of 100 milliseconds, or 0.1 second.

f) Repeat part d, but with a different value for the Timer 3 auto-reload register designed for time slices of 10 milliseconds, or 0.01 second.

g) Adjust the Timer 3 auto-reload register value to give what you consider to be the longest time slice that gives smooth "multitasking" in your "operating system". That is, the two tasks should both appear to run simultaneously at constant rates, with no jerky switching from one task to the other. Report your time slice value.

Part 2: Time to work on the current programming project assignment.