Lab 4

Interrupts

Purpose:

This lab will familiarize the student with the operation of interrupts using the STM32F10x.

Equipment/Software:

- STM32F10x Development Board
- USB Cable
- PC with Keil UVision4
- Reference Manual, Data Sheets as required

Background:

The SYSTICK Timer is part of the ARM Cortex M3 core. This means that it is available to programmers regardless of the variant of device that is being used. SYSTICK can be used to generate interrupts at regular intervals to aid in scheduling tasks within a program.

EXTI0 is an external interrupt source which can be connected to various input pins on the STM32F10x. This interrupt will allow the user to detect and respond to external logic level transitions.

In order to use an interrupt, a number of considerations must be addressed. The programmer needs to create an Interrupt Service Routine (ISR) which will perform tasks when the interrupt occurs. The programmer must also provide the address of the ISR to the system Vector Table, so that the controller is able to find the ISR when the interrupt occurs. The system generating the interrupt must also be configured properly.

Some important things to know:

A number of registers are automatically pushed when the processor enters an interrupt and popped when it exits. (Section 9.1 of the text).

ISRs should perform their tasks and exit in the least possible time. Performing delays, calling other routines, waiting for flags, are all bad practices in ISR programming.

Information about the registers associated with SYSTICK can be found in Section 8.5 of the text.

Procedure:

1) Identify the SYSTICK registers using the Cortex M3 Programming Manual. Configure the SYSTICK Reload value to 0xB71B00 and enable the SYSTICK timer with SYSCLK as its clock source.

2) Create an interrupt service routine for SYSTICK which will toggle the state of any available output on your development board when it executes.

3) Use the ADALM to measure the frequency of the signal on your selected output. From this measurement, and the Reload value from Step 1, what is the frequency of the system clock?

4) Configure the EXTI0 interrupt to use PB0 as its interrupt source and generate interrupts on the falling edge of logic changes on PB0.

5) Create an interrupt service routine for EXTI0 which will set an available output HIGH when a HIGH to LOW transition appears on PB0.

6) In the while(1) loop of your main function, test the state of the output used in the previous step. If the output becomes HIGH, enter a delay loop of 100 ms, and then change the output state to LOW after the delay has occurred.

7) Use your ADALM to generate HIGH and LOW logic values on PB0 and to monitor the state of the output used in step 6. Debug and test your program.

Using the interrupt system with the Keil startup files:

The Keil startup files do a lot of the heavy lifting for us. They configure the vector table based on the processor variant we are using. They also establish default names for the interrupt handlers. We must use these default names in order for everything to work.

Let's look at SYSTICK first:

The SYSTICK system has 4 registers that might need to be considered: CTRL, LOAD, VAL, and CALIB.

The order of configuration operations will be:

1) Disable SYSTICK by writing 0 into the CTRL register

2) Clear the SYSTICK counter by writing 0 into the VAL register

3) Configure the interval for the interrupt by writing a value into the LOAD register
The value is the number of counts of the 24 MHZ clock that occur before the interrupt is generated.
The value is also automatically reloaded into the counter after each SYSTICK event

4) Select the clock source, enable the interrupt, and enable the counter by writing a 7 into the CTRL register

SYSTICK will also require a handler or Interrupt Service Routine to respond to the interrupt. The handler will appear as a function in your main.c file and must use the name SysTick_Handler. Your function will look like this:

```
void SysTick_Handler(void)
{
do something
}
```

That's it, that's all! Each time the SYSTICK interrupt occurs, the instructions in SysTick_Handler will be executed.

Now for EXTI. This is a little more complicated because EXTI is part of the Nested Vectored Interrupt Controller (NVIC). The process is similar but we have more registers to configure in many different subsystems.

We're going to use PB0 as our EXTI0 inputs, so the steps will be:

1) Turn on the clocks for AFIO and PortB (RCC-> APB2ENR)

2) Select Port B pins as the source for EXTI 0 events (AFIO->EXTICR)

3) Unmask PB0 as the interrupt source (EXTI->IMR)

4) Select the falling edge of PB0 events as the trigger (EXTI->FTSR)

5) Unmask EXTI0 as an interrupt source in the NVIC (NVIC->ISER[0])

And then we also need an interrupt handler that uses the name EXTI0_IRQHandler. An important part of the interrupt handler is code to acknowledge that we've acted on the interrupt and to clear it as an interrupt source. This is done by overwriting the appropriate "interrupt pending" bit.

```
Void EXTI0_IRQHandler(void)
{

EXTI->PR |= EXTI_PR_PR0;        // Clear the pending interrupt bit

and then do something useful
}
```

And now a falling edge on PB0 should cause the code in EXTI0_IRQHandler to execute.