Lab 3: Interrupts

Kyle Swanson

April 4, 2015

For this lab, the main focus was on interrupts and how the ARM architecture handles them. We continued using the vector table, learned what interrupts were, and how they related to hardware buttons. We kept learning about the different registers and the stack, and how they may be affected by interrupts.

The vector table again played a major part in this lab. For completeness, the vector table addresses were listed in the *statup.s* file, see the example below. Make sure to note the SysTick_Handler as it comes up later in the program.

```
DC32    Stack                   ; 0x00000000   0-Stack Pointer
DC32    __iar_program_start     ; 0x00000004   1-Reset Handler
DC32    NMI_Handler             ; 0x00000008   2-NMI Handler
DC32    HardFault_Handler       ; 0x0000000C   3-Hard Fault Handler
DC32    MemManage_Handler       ; 0x00000010   4-MPU Fault Handler
DC32    BusFault_Handler        ; 0x00000014   5-Bus Fault Handler
DC32    UsageFault_Handler      ; 0x00000018   6-Usage Fault Handler
DC32    SVC_Handler             ; 0x0000002C   11-SVCall Handler
DC32    DebugMon_Handler        ; 0x00000030   12-Debug Monitor Handler
DC32    PendSV_Handler          ; 0x00000038   14-PendSV Handler
DC32    SysTick_Handler         ; 0x0000003C   15-SysTick Handler
```

Figure 1: A selection of the vector table values.

To start, we had a program that simply looped for ever.

```
B       .       ; Pretend the processor is gainfully occupied.
```

As the comment implies, this is to simulate a computer performing actions, like running another program, or receiving input from a keyboard.

Before the infinite loop, the program did a few other tasks, like initializing the GPIO ports, and the interrupts associated with those ports.

To see how ports are initialized, we inspected the *GPIOF_Init* branch. It starts by using the *LDR* instruction to move SYSCTL_GPIOHBCTL_R into R0.

Continuing through the rest of this branch, it continues to set different values which are essentially defaults to set up the Input Output controller for the switch we want.

Next, we needed the Interrupts for our button to be configured. A hardware interrupt is a signal from a device, in this case a button, to the processor that it needs immediate attention. The most obvious example of an interrupt on a normal computer is a keyboard, when you tap a key, it sends a signal to the processor that it needs attention, which then communicates with the OS to handle the interrupt. In our experiments, the interrupts were handled by functions in our code. The configuration for our interrupt happened in the *GPIOF_Interrupt_Init* branch.

Like above, this used the *LDR, MOV, and ORR* commands among others, to set different values. Part of the process is setting certain bits to 1 to enable

the GPIO port we need for the led to function.

While there are more parts to initializing the GPIO ports, those were the main functions. Now that we have a brief explanation about initializing the interrupts, back to the vector table. The vector table is like a directory. I think the ARM docs say it best, "The Cortex-M3 vector table contains the address of the exception handlers and ISR (Interrupt Service Routine)". So, as we would later found with our GPIO and SysTick handlers, the vector table tells the hardware what address to run in the event of the corresponding interrupt. See figure 2 for an example of our SysTick Handler.

```
PUSH    {LR}                    ; Will need this later.
BL      SysTick_Disable         ; Only needed one SysTick

; Read SW1 position. SW1 depressed will be R0 bit 4 = 0.
LDR     R0, =GPIO_PORTF_AHB_DATA_BITS_R
ADD     R0, R0, #0x40   ; SW1 data bits address offset
LDR     R1, [R0]        ; SW1 state will be 1 when not pressed.
ANDS    R1, R1, #0x10   ; Test bit 4. Is it set?
CBNZ    R1, ReArm       ; Bit set? Re-arm GPIOF interrupt.

; Read, modify, write LED color bits
LDR     R0, =GPIO_PORTF_AHB_DATA_BITS_R
ADD     R0, R0, #111B << 3      ; GPIOF LED Bits offset PF123
```

```
LDR     R1, [R0]        ; Fetch current LED color
LSR     R1, R1, #1      ; Right shift LED color bit
BIC     R1, R1, #1      ; Clear SW2 bit
CBNZ    R1, SetLED      ; Branch to SetLED if result non-zero.
MOV     R1, #0x08       ; Hit zero, re-init LED color to green.
```

Figure 2: The SysTick_Handler.

```
; Clear the interrupt
LDR     R0, =GPIO_PORTF_AHB_ICR_R       ; See datasheet p670
LDR     R1, [R0]        ; GPIO_PORTF_AHB_ICR_R register value to R1
ORR     R1, R1, #0x10   ; Clear bit 4 interrupt
STR     R1, [R0]        ; Ack
BX      LR
```

Figure 3: The GPIOF_PF4_Interrupt_Clear function.

One part in the process to handle button hits was acknowledging the ISR. See figure 3 for the code on how to do this. We learned with the experiments, that if you don't acknowledge this interrupt (STR R1, [R0]), it will keep being raised. Note that we didn't do this for the SysTick, I assume that's since it's a periodic timer, and not related to a user or system input. It's expected that SysTick happens in a timely manner, where a button press is not.

Throughout the program, you see PUSH {LR} and POP {LR}. {LR} is the link register, in simple terms, this register contains information about where to return to when a function finishes. If my understanding is correct, in the *GPIOPortF_Handler* it pushes the LR onto the stack to store it for when this interrupt is complete. Once it's done, it pops it off the stack, and branches to LR. If my understanding of this lab and architecture is true, the stack is used with interrupts to store where to return to after an interrupt occurs and is processed. This is to ensure that the computer resumes processing from where it left off when the interrupt occurred. Since, as the name implies, it interrupted normal execution.

As you may of picked up on, interrupts allow a computer to practically do multiple things at once, even with only one processing core. While not technically at once, the process of Context switching is what enables this. Without context switching, your computer could only do one thing at a time, it could process a thing, then respond to input, etc. This sounds like a pretty miserable way to use a computer. Luckily, computers can do small task very quickly, so context switching has a computer perform a small task, then switch to another small task, and with enough of these switches, you have the illusion of a multitasking computer. Above is a form of context switching, the handler interrupts the normal flow of the hardware, the hardware gives control to the handler, which then using the LR register stores it, does it's own actions, and switches context back to the previous program by popping the LR register.

I found this lab to be quite interesting. It was fun to run through the code and see exactly how it is affecting the hardware, in this case, the LED. Since we were talking about and experimenting with the SysTick, Registers, Interrupts, and Context Switching, it was a good refresher from things I learned in a past Operating System class and simpler Architecture classes.