Lab 3: Interrupts

CMPT 328 Computer Architecture

Scott Overholser

WESTMINSTER

SALT LAKE CITY • UTAH

This lab was written for the Westminster College CMPT 328 Computer Architecture course Spring 2015 semester.

Last Modified: Sat Mar 7 12:59:15 2015 -0700

Table of Contents

1 Introduction

In Lab 2 we learned about fault conditions, the resulting processor exceptions, and how to handle them. Interrupts are similar in that program execution is abruptly changed and the location of the Interrupt Service Routine (ISR), like the fault handler, is specified in the *vector table*. The difference is that interrupts are more configurable to suit our purposes.

For Lab 3 we will examine an assembly language program that uses two different interrupts. An edge-triggered interrupt is configured for GPIO port F switch SW1. Another interrupt is configured for the system timer (SysTick) that is used in debouncing the switch.

Lab 3 also introduces the `startup.s` file that provides a complete *vector table* for our program. This file will be used with only minor modifications in all future assembly projects. You can consider it largely *boilerplate* code.

This lab also introduces a fully functional example of how to work with the GPIO ports. We will be working with GPIO port F. Even though each GPIO port offers different features and has it's own configuration, they are all very similar in terms of usage. The LaunchPad has 6 GPIO ports, A-F.

The program we will be working with in this lab activates one of the three LEDs when switch SW1 is pushed. Each push of the switch is handled by the two different ISRs which together cooperate to process the switch SW1 interrupt and rotate the active LED.

The switch SW1 is a mechanical switch and is imperfect—it bounces one or more times when pressed. These bounces are very short on the 1ms time scale. So humans do not notice this. We experience a crisp click as we press the button. But the processor at a clock speed of 80MHz cycles about one hundred thousand times each millisecond and notices each bounce and acts on it. The result is multiple LED rotations for each press of switch SW1.

The solution to this problem is to implement a debouncing algorithm. I chose the system timer to implement debouncing. The basic idea is to have the GPIO port F ISR turn off interrupts on GPIO port F and initialize the system timer with interrupts. The SysTick interrupt handler is activated and checks the switch SW1 value. If it is logic LOW, then the switch SW1 is pressed and the LED colors are rotated. If switch SW1 value is logic HIGH, then re-arm the GPIO port F interrupt and carry on as before. In either case, the SysTick ISR disables the system timer because we need it only once per SW1 activation.

Note the many places in the code where register LR is pushed onto the stack prior to branching. This saves the LR value so the code can return from a previous branch using `BX LR`.

2 Lab Objective

This lab will explore interrupts, what they are, and how the processor handles them. This encompases the *vector table*, *interrupt service routines*, and *context switching*.

Secondarily, you will learn how to use the General Purpose Input/Output (GPIO) system.

You will be introduced to the LaunchPad schematic in the TM4C123G LaunchPad Evaluation Board Users Guide. This will help you understand how to use the GPIO. You will continue learning how to use the Tiva TM4C123GH6PM Microcontroller Data Sheet as a reference.

3 Notes

You should now be familiar with your IAR Embedded Workbench development environment. You should also understand the *boot loading* process, the *vector table*, and both general and special purpose registers. You should also be comfortable referencing the TM4C datasheet.

3.1 Reference Material

Tiva TM4C123GH6PM Microcontroller Data Sheet will be referenced often in this and future labs. It is in the Canvas Files area filed under Tiva C Series Microcontroller. The filename is `tm4c123gh6pm.pdf`.

This lab exercise introduces the TM4C123G LaunchPad Evaluation Board Users Guide. It is in the Canvas Files area in the Tiva C Series Microcontroller folder. The filename is `spmu296.pdf`. Download this file to your computer.

The User's Guide is useful primarily for the schematics following page 19. The schematics provide information about the configuration of the GPIO pins, switches, LEDs, and jumper blocks. You will need to reference the schematic for this and future labs.

Also refer to `lm4f120h5qr.h` in the `arm/inc/TexasInstruments` folder on your development system. This is a C header file that contains useful C macro definitions for the TM4C. These macro names have been used in the assembly project for this lab.

The following list of citations is relevant to the assembly project written for this lab exercise:

- *Schematic Sheet 1 of 3*—user's guide, p20.
- *Figure 1-1. Tiva TM4C123GH6PM Microcontroller High-Level Block Diagram*—datasheet, p48.
- *2.3.2 Stacks*—datasheet, p74.
- *2.5.4 Vector Table*—datasheet, p106.
- *2.5.7.1 Exception Entry*—datasheet, p109.
- *2.5.7.2 Exception Return*—datasheet, p110.
- *Figure 2-7. Exception Stack Frame*—datasheet, p110.
- *Register 1: SysTick Control and Status Register (STCTRL), offset 0x010*—datasheet, p138.
- *Register 2: SysTick Reload Value Register (STRELOAD), offset 0x014*—datasheet, p140.
- *Register 3: SysTick Current Value Register (STCURRENT), offset 0x018*—datasheet, p141.
- *Register 4: Interrupt 0-31 Set Enable (EN0), offset 0x100*—datasheet, p142.
- *Register 32: Interrupt 12-15 Priority (PRI3), offset 0x40C*—datasheet, p152.
- *Register 36: Interrupt 28-31 Priority (PRI7), offset 0x41C*—datasheet, p152.
- *Register 9: GPIO High-Performance Bus Control (GPIOHBCTL), offset 0x06C*—datasheet, p258.
- *Register 60: General-Purpose Input/Output Run Mode Clock Gating Control (RCGCGPIO), offset 0x608*—datasheet, p340.

- *10.4 Register Map*—datasheet, p658.
- *Table 10-6. GPIO Register Map*—datasheet, p660.
- *Register 2: GPIO Direction (GPIODIR), offset 0x400*—datasheet, p663.
- *Register 6: GPIO Interrupt Mask (GPIOIM), offset 0x410*—datasheet, p667.
- *Register 9: GPIO Interrupt Clear (GPIOICR), offset 0x41C*—datasheet, p670.
- *Register 15: GPIO Pull-Up Select (GPIOPUR), offset 0x510*—datasheet, p677.
- *Register 18: GPIO Digital Enable (GPIODEN), offset 0x51C*—datasheet, p682

4 Experiments

Download and extract the Lab 3 project zip file. Double-click on `workspace.eww` to start IAR Embedded Workbench.

Review the code. Look at `startup.s`. The complete *vector table* is implemented there. The only exceptions are `GPIOPortF_Handler` and `SysTick_Handler`. They are included in `main.s`.

All of the logic for this project is implemented in `main.s`. The `main` function simply initializes GPIO Port F and GPIO Port F pin 4 (SW1) interrupts. Incidentally, it also loads some junk data into registers `R0`, `R1`, `R2`, `R3`, and `R12` to make the *stack frame* easier to see. But this step is for purposes of illustration only. It does not contribute to program function at all.

After initialization, the `main` function just loops forever. This is just to simulate other activity that the processor might be doing as interrupts occur. The other activity might be reading from a UART, writing to an ethernet port, or reading keyboard input.

4.1 Start Program in Debugger

Compile the program. Then Download and Debug. The debugger will break at `main` as usual. Click the "Go" button on the debug toolbar.[1] The program will begin executing. Confirm program operation by pressing the button labeled SW1 on your LaunchPad board. Press SW1 repeatedly and observe the active LED rotate through the colors green, blue, and red.

4.2 GPIO Port F Interrupt Handler

Stop debugging. In the `main.s` source view, set breakpoints `main` at line 70, line 84 in CheckSW1Value, and in `GPIOPortF_Handler` at lines 257 and 262,

Download and Debug.

4.2.1 SW1 Pull-up Resistor

The first observation you will make is the value of switch SW1 before and after the pull-up resistor is configured. Click the "Go" button to run the program. It will break in the

[1] It is the button with "+++" on it.

`CheckSW1Value` function at line 84 that is called from the `GPIOF_Init` function[2]. Make sure you have selected *Binary Format* by right clicking the value of register `R1` in the Watch window.

Single-step through the next few instructions until you reach the `LDR R1, [R0]` instruction. Execute that and inspect register `R1` in the watch window. Observe that every bit position is 0 or logic LOW. The pull-up resistor is not yet configured. *Make a note of this observation.*

Click the "Go" button again. The program will again break. Repeat the steps from the last paragraph. This time you will notice that bit 4[3] position of R1 is 1. This is because the pull-up resistor is configured and is pulling up the value of SW1 to logic HIGH. *Make a note of this observation.*

See the LaunchPad schematic sheet 1 of 3 for details on SW1. The switch circuit is in the middle of the sheet. It shows that SW1 connects GPIO port F pin 4 to GND. The pull-up resistor must be configured in order to arrange for a logic level change. Otherwise, SW1 would do nothing. *This is significant. You should discuss this in your lab report.*

4.2.2 Load Conspicuous Values in Registers

Click the "Go" button once again. The program will stop at the first breakpoint on line 70. Single-step[4] through the `LDR` instructions that load the *junk* values into the registers. *Note the register values as you do this.* You will see each update in turn. Stop single stepping when you reach the `B .` infinite loop.

4.2.3 SW1 Interrupt

Click the "Go" button in the debug toolbar. The program will run freely. Then push the button SW1 on the LaunchPad board. The program will stop at the breakpoint set on line 262. The GPIO Port F SW1 interrupt was triggered by the button push.

Note the value of the SP register. In the memory view click in the "Go to" field and enter the SP register value using the `0x` hexadecimal notation. Compare the 4-byte words on the stack with the register values noted above. Review *Figure 2-7. Exception Stack Frame* on page 110 of the datasheet. *Note the contents of the general purpose registers R0-R3, R12.* They are the same as before. Even though they have been pushed onto the stack, their values are unchanged.

Note the value of register IPSR. The value in register `IPSR` will indicate the interrupt number being processed. This will correspond to the address contained in the *vector table* at the location corresponding to the specific interrupt number. *Examine the vector table and find the decimal value of the interrupt number in register IPSR.*

The memory address in the *vector table* corresponding to the interrupt number in register `IPSR` is the location of the executable code the processor is to execute—it is the *interrupt service routine* or ISR. In the disassembly window, compare the memory address in the *vector table* and the beginning of the `GPIOPortF_Handler`.

At this point, note that `GPIOPortF_Handler` performs only three tasks: It clears the interrupt. If this is not done, the interrupt will assert over and over *ad infinitum.* Then it

[2] See lines 122 and 127.

[3] Counting from 0.

[4] Click the "Step Into" button on the debug toolbar.

disables the SW1 interrupt. Finally it initializes the system timer with interrupt. Program execution then resumes in the `main` infinite loop. Click the "Go" button in the debug toolbar.

Program execution will stop at the breakpoint set on line 262. *Note the contents of the general purpose registers.* They have changed because they are modified by the `GPIOF_PF4_Interrupt_Clear`, `GPIOF_PF4_Interrupt_Disable`, and `SysTick_Init` functions.

4.2.4 Return From Interrupt Handler

Click the "Step Into" button on the debug toolbar. Program execution will now be stopped at the `B .` infinite loop instruction in `main`. *Note both the general purpose and special purpose registers.* They now contain their original values once again.[5]

You are ready to proceed with the next section.

4.3 SysTick Interrupt Handler

The system timer will interrupt almost immediately as we click the "Go" button on the debug toolbar. Before we do that, in the `main.s` source view, set a breakpoint in `SysTick_Handler` at line 281. Now click the "Go" button on the debug toolbar.

4.3.1 SysTick Interrupt

Program execution will break at line 281 in the `SysTick_Handler`. Note that the first action taken by the `SysTick_Handler` is to disable the system timer. We are done with with the timer. It is used only for SW1 debounce.

Note again the SP register. The *stack frame* has again been pushed onto the stack. Since the *stack frame* it is the same as before it will appear unchanged. *Note the value of register IPSR.*

The value in register `IPSR` will indicate the interrupt number being processed. This will correspond to the address contained in the *vector table* at the location corresponding to the specific interrupt number. *Examine the vector table and find the decimal value of the interrupt number in register IPSR.*

4.3.2 Check SW1 Value

This is where the real action occurs. The first step is to check the value of SW1. If it is still pressed, then it is a real button push. Keep in mind that at full speed, only a few milliseconds have elapsed since you pushed the button. But the switch bounces have been avoided.

The second action will depend on whether SW1 is logic LOW or logic HIGH. See? Binary. If SW1 is logic LOW, it is a button push and the LED bit will be shifted into the next position. If SW1 is logic HIGH, the LED bit rotation will be skipped. In both cases the GPIO port F SW1 interrupt will be re-enabled

[5] Note the memory view at the top of the stack. The stack frame has not been erased, only copied back to the registers.

4.3.3 Rotate the LED Bit

Since we did push the button, single-step through the code and follow what happens. On the *View* menu select *Watch->Watch 1.* Arrange the Watch window conveniently. In the *Expression* field enter "R1". In the *Value* field, right-click on the value and select "Binary Format." Make the field wide enough to view all 32 binary digits in the *Value* field. *Since you downloaded this project you should already see the Watch window. The instructions are for completeness only.*

On the first button push, register R1 will be zero. Step through the code anyway to see how it is initialized with the green bit which is `0x8`. At this point you should see a single bit set in register R1 corresponding to `0x8`.

Clear all breakpoints except the one in `SysTick_Handler` at line 281. Click the "Go" button on the debug toolbar. Now each time you press SW1, program execution will break in `SysTick_Handler`.

Press the SW1 button repeatedly. Each time program execution breaks in `SysTick_Handler`, single-step through the code. Make note of the *Watch* window and the binary value of register R1. Watch when the `LSR` instruction is executed. This instruction is Logical Shift Right. Bit position 0 is for SW2. After the red LED is set, the bit will be shifted into the SW2 position. But we do not have any interest in SW2 and we do have an interest in testing for end of rotation. So on each bit shift, a `BIC` or bit clear instruction is used to clear bit position 0. Then if register R1 is non-zero, `CBNZ`, Compare Branch Non-Zero, is done to `SetLED` to set the shifted bit value. Otherwise, register R1 is zero and is reinitialized with `0x8`.

4.4 GPIO Port F Initialization

Review sheet 1 of the LaunchPad schematics in the LaunchPad User's Guide. The physical GPIO pins are presented in detail there. In the upper left is a partial representation of chip U1 labeled U1-A. Each GPIO pin is labeled with it's port designator and a pin number. For example, USR SW1 is labeled PF4 to indicate Port F pin 4. Similarly, the LEDs are labeled PF1, PF2, and PF3 respectively for green, blue, and red.

Note also the representation of SW1 in the middle of page 1 of the schematic. It shows a switch that will short to GND when pressed. In the lower left on the same page see LED_R, LED_G, and LED_B.

Review the High-Level Block Diagram listed in the References section. It is on page 48 of the datasheet. Note that there are two busses used for communicating with the peripherals. The Advanced Peripheral Bus (APB) is the legacy bus and is used for compatability. It is the default. We will be using the faster Advanced High Performance Bus (AHB).

Turn your attention now to the code in the IAR Embedded Workbench. See the function `GPIOF_Init`. This section of code initializes GPIO port F and the specific GPIO port F pins we want to use. This is done very simply by writing to various control registers.

`GPIOF_Init` takes 4 steps:

- Select the AHB.
- Enable GPIO port F clock.
- Set the direction of the LED pins to output. The default direction is input.

- Set PF1-4 as digital.

Note how the register values are set. First, the register address is loaded into register R0. Then the value of the register is loaded into register R1. Then the `ORR` instruction is used to toggle only 1 bit. Recall from the Boolean Null Element theorem (T2') `B + 1 = 1` that any binary digit OR 1 is equal to 1. The resulting value is stored back into the register at the address in register R0.

This is an important approach. Using the `ORR` instruction does not disturb any other bits in the register that are already set and we do not need to test them individually. Nor do we stomp on them by using `LDR R1, 0x20` that would set bit 5 but would also *clear* every other bit!

Single-step through `GPIOF_Init` to familiarize yourself with it's operation.

4.5 GPIO Port F Interrupt Initialization

The code in `GPIOF_Interrupt_Init` configures the interrupt on SW1. GPIO interrupts must be configured in two steps. First, configure the Nested Vector Interrupt Controller (NVIC) to process the interrupt. Second, configure the GPIO interrupt.

Configuring the NVIC is easy. Just set the priority of the specific interrupt that you want to handle. In our case we are selecting the highest priority of 0. Next, set the enable bit in the memory mapped register NVIC_EN0.

Configuring the interrupt on SW1 is also easy. Edge-triggered interrupts[6] are the default. We need to select pull up on PF4 so the logic level is HIGH by default. Recall from the schematic that a SW1 press connects PF4 to GND. If we did not pull PF4 HIGH, there would be no logic level change on SW1 press. Last, we need to set the GPIO port F interrupt mask to select interrupts on PF4.

Now interrupts are fully configured in the NVIC and GPIO port F. Single-step through `GPIOF_Interrupt_Init` to familiarize yourself with it's operation.

4.6 System Timer Initialization

The system timer implements a simple countdown timer. The 24-bit countdown or RELOAD value is loaded into the NVIC_ST_RELOAD register. Once enabled, the system timer counts down from this value once per clock cycle. The current value of the timer is stored in NVIC_ST_CURRENT. When the count down reaches zero, time is up. In our case, we setup an interrupt to be triggered when the count down reaches zero.

The system timer is useful because we do not need to implement it in our code. It is an automatic mechanism of the processor. If we want to implement a delay loop, we can just loop and wait until the NVIC_ST_CURRENT value reaches zero. Or, we can setup the interrupt and forget about it until the interrupt is triggered. The latter is the approach taken in this project.

Configuring the NVIC is the same as described earlier in GPIO Port F Interrupt Initialization. To configure the system timer, we need to first disable the timer so it is not counting down as we are configuring it. Then the countdown value is loaded into the

[6] As opposed to level-triggered interrupts.

NVIC_ST_RELOAD register. The NVIC_ST_CURRENT register is cleared. Once fully configured, we enable the system timer with interrupts.

In the source view clear all breakpoints. Set a breakpoint in `main.s` at line 201. Click the "Go" button on the debug toolbar. Press button SW1 on the LaunchPad. Program execution will stop at your breakpoint. Single-step through the `SysTick_Init` code to familiarize yourself with it's operation.

5 Lab Report

Lab report will focus on the general purpose of interrupts as well as their specific use in this lab project. It will also cover the roles played by the vector table and the stack. Observations made during the experiments chapter will be integrated with the report.

Explain how ARM interrupts work.

- The vector table and it's purpose.
- The purpose of interrupts.
- The Nested Vector Interrupt Controller
- The meaning of the value in register IPSR.
- The stack and how it is used when an interrupt occurs.
- The reason for acknowledging the interrupt from the ISR—SysTick being the notable exception.
- The process of context switching and why it matters.
- Why do some of the *functions* `PUSH` register LR onto the stack?

As you write your lab report, include your observations to illustrate and support your explanations.