Designing with a Microcontroller – Interrupt Handling

Safety: In this lab, voltages are less than 15 volts and this is not normally dangerous to humans. However, you should assemble or modify a circuit when power is disconnected and don't touch a live circuit if you have a cut or break in the skin.

Objective: This lab provides an opportunity to learn about single and multiple interrupt handling in a Stellaris® microcontroller (MCU). It uses the Stellaris® LM4F120 LaunchPad [1] that has an ARM Cortex M4F (LM4F120H5QR, datasheet [2]) microcontroller in it.

The lab has two parts. Part I of this lab uses a single internal interrupt where a stop-watch will be designed using the SysTick timer. In Part II, the students will use multiple external interrupts to design a customer counter of a store.

LaunchPad Interface: The pin-out diagram of the LaunchPad is shown in Fig. 1. Although the LM4F120H5QR MCU has 43 GPIO, only 35 of them are available through the LaunchPad. They are: 6 pins of Port A (PA2-PA7), 8 pins of Port B (PB0-PB7), 4 pins of Port C (PC4-PC7), 6 pins of Port D (PD0-PD3, PD6-PD7), 6 pins of Port E (PE0-PE5), and 5 pins of Port F (PF0-PF4). In addition, there are two ground, one 3.3V, one 5V (VBUS), and one reset pins available on the LaunchPad.

Pins PC0-PC3 are left off as they are used for JTAG debugging. Pins PA0-PA1 are also left off as they are used to create a virtual COM port to connect the LaunchPad to PC. These pins should not be used for regular I/O purpose.

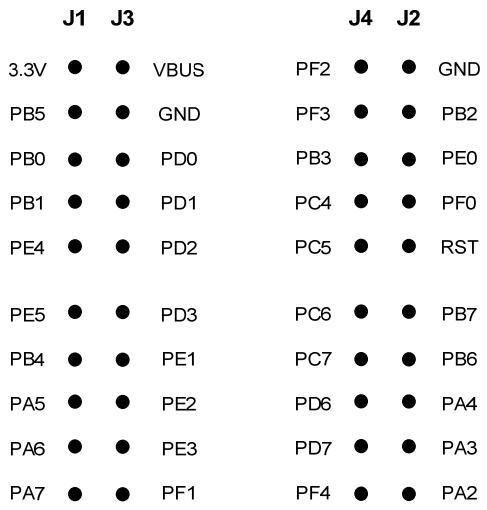

Fig. 1 Header pins on the LaunchPad (EK-LM4F120XL)

Interrupt Control in Cortex-M: Interrupts on the LM4F120H5QR MCU are controlled by the Nested Vectored Interrupt Controller (NVIC) (p.47 of [2]). The NVIC and Cortex-M4F prioritize and handle all exceptions in Handler Mode. The processor state is automatically stored to the stack on an exception and automatically restored from the stack at the end of the Interrupt

Service Routine (ISR). The interrupt vector is fetched in parallel to the state saving, enabling efficient interrupt entry. The processor supports tail-chaining, meaning that back-to-back interrupts can be performed without the overhead of state saving and restoration.

Software can set **eight priority levels** (0 to 7; a higher level corresponds to a lower priority, i.e., level 0 is the highest interrupt priority) on **seven exceptions** (such as, reset, software interrupt, hardware interrupt, bus fault, etc.; more on p.97) and **65 interrupts**. More information on NVIC can be found on p.119 of [2].

Each exception has an associated 32-bit vector that points to the memory location where the ISR that handles the exception is located. These vectors are stored in ROM at the beginning of the program memory. The internal ROM of the Stellaris device is located at address 0x0100.0000 of the device memory map (p.483). The vector addresses in ROM of these exceptions can be found on p.98-101 (Table 2-8 and 2-9 in [2]). These definitions are added in the **Startup.s** file provided on the course website. It can also be downloaded from [3]. <u>You must use this file in your project</u>. (**Do not make any changes to the Startup.s file**)

For example, ROM address 0x0000.0004 contains the initial program counter which is also called **reset vector**. It points a function called Reset_Handler which is the first thing executed following reset. Similar, ROM addresses 0x0000.003C and 0x0000.00B8 point to SysTick_Handler and GPIOPortF_Handler respectively. It means that if we want to use the interrupt triggered by SysTick timer, we have to use the "SysTick_Handler ()" function as the ISR in our main code.

To activate an interrupt source, we need to perform two tasks: (a) enable the source from the corresponding NVIC and (b) set its priority. The register map is given on p.129 (see Table 3-8 in [2]). To better understand the activation procedure, an example is given below: Say, you want to use Port F interrupt. First of all, find the interrupt number (i.e., bit position in interrupt register) from Table 2-9 (2^{nd} column) on p.99 corresponding to GPIO Port F. It is "30". Now find the interrupt register that is needed to enable IRQ30 from Table 3-8 (p.129). It is NVIC_EN0. So, it tells you that you need to set bit 30 of NVIC_EN0 to '1' to enable interrupt on Port F.

From Table 3-8, you will also need to find the register needed to set the priority of IRQ30. It is NVIC_PRI7. To set a priority value, say 5, you may use the following statement in C:

NVIC_PRI7 = (NVIC_PRI7 & 0xFF00FFFF) | 0x00A00000;

For both NVIC_EN0 and NVIC_PRI7, go to respective pages in [2] to find more instructions on how to use them. However, it should be noted that the interrupt triggered by the SysTick timer is known as "processor exception", and thus cannot be controlled by any NVIC_EN register. Instead, NVIC_ST_CTRL is used to enable/disable it (p.133).

Part 1 – Building a stop-watch

The objective is to build a stop-watch using SysTick timer. The on-board switch, SW1 will be used to start/stop the watch. The time (mm:ss) will be displayed on the 7-segment display board.

Using SysTick timer: Cortex-M4 has an integrated system timer (24-bit down counter), known as, SysTick, which provides a simple, 24-bit clear-on-write, decrementing, wrap-on-zero counter with a flexible control mechanism. The counter can be used in several different ways. More information on SysTick can be found on p.118 (section 3.1.1 of [2]). It can be used to create time delays and generate periodic interrupts. The key registers used to control SysTick as a follows:

NVIC_ST_CTRL – to enable timer features (p. 133)
NVIC_ST_RELOAD – specifies the start value to load into the timer (p.135)
NVIC_ST_CURRENT – contains the current value of the timer (p.135). Note that this register is write-clear. Writing to it with any value clears the register.
NVIC_SYS_PRI3 – used to configure the priority level of SysTick exception (p.167)

Interrupt Control for SysTick: The timer needs to be turned off during configuration (clear ENABLE). In NVIC_ST_CTRL, make sure to set CLK_SRC to '1' (for internal clock source) and INTEN to '1' (to enable interrupt). When the value of CURRENT register counts down from 1 to 0, the COUNT flag (bit 16 in NVIC_ST_CTRL) is set. On the next clock, the CURRENT register is loaded with the RELOAD value. In this way, the SysTick counter continuously decrements and operates as modulo *N*+1 if the RELOAD value is *N*. When INTEN is set to '1', the COUNT flag triggers a hardware interrupt. Once properly configures, the SysTick timer is turned on (set ENABLE). It should be noted that the timer can be stopped at any time by disabling the ENABLE bit.

The circuit diagram is given in Fig. 2. Open a new project in Keil uVision. Make sure to use the Startup.s file provided on the course website. Write a C-program to accomplish the objectives. In your main program, configure Ports A, B, and F accordingly. Make sure to enable pull-up resistors on Port F (PF4 for SW1). Once the program is fully written and successfully compiled, load it into the LaunchPad and demonstrate the stop-watch to the instructor or lab TA.

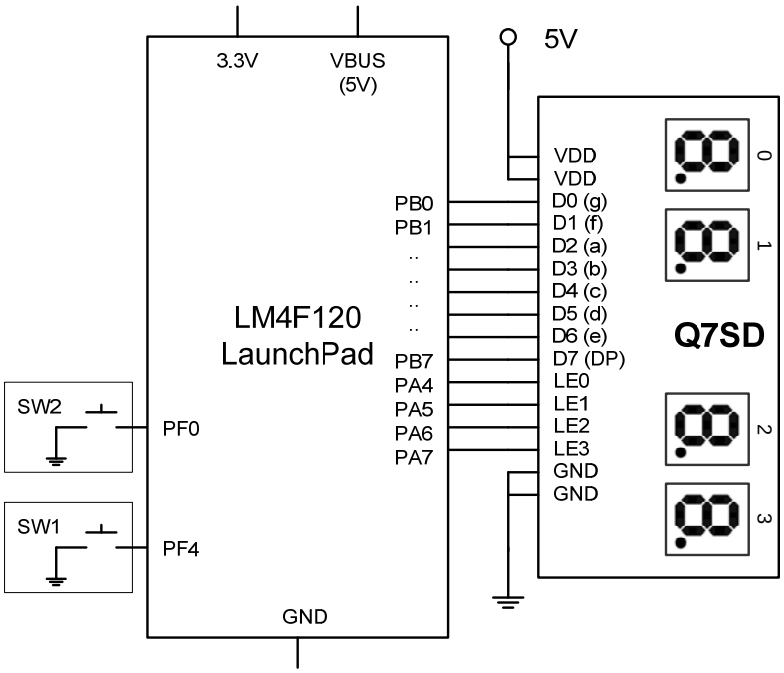

Fig. 2 Circuit diagram of the lab

Part 2 – Building a customer counter

In this part of the lab, we will design a customer counter of a store. You will use hardware interrupt triggered by GPIO Port F (external event). You will also learn how to handle more than one interrupt.

Problem Description: If the number of customer inside the store at a given time is 30 or more, the system will flash a red light. If the number is between 20 and 29, it will turn on a blue warning light. And if the number is below 20, it will turn on a green light indicating normal operation. You may use the onboard LEDs on the LaunchPad for this purpose. There are two doors in the store – one for entry, one for exit. We will use two onboard switches (SW1 and SW2) to simulate the entry and exit action of customers. For example, pressing SW1 once means that a customer has just entered into the store, while pressing SW2 means that a customer has just left the store. These switches are active low. You will need to use two 7-segment displays to display the number of customer.

Unlocking SW2 (PF0): By now, you know how to configure Port F to use SW1 (PF4). However, SW2 (PF0) is locked as it is a non-maskable interrupt (NMI) (p.213, section 5.2.3.1). A special procedure is required to unlock it. As NMI pin, the Alternate Function Select, Pull-Up Resistor, Pull-Down Resistor, and Digital Enable are all locked for PF0 until a value of 0x4C4F434B is written to the Port F GPIO Lock Register (GPIO_PORTF_LOCK, p.637). After Port F is unlocked, bit 0 of the Port F GPIO Commit Register (GPIO_PORTF_CR, p.638) must be set to '1' to allow access to PF0's four control registers. On the LM4F120, the other bits of the Port F GPIO Commit Register are hard-wired to '1', meaning that the rest of Port F can always be freely re-configured at any time using respective control registers. Requiring this procedure makes it unlikely to accidentally re-configure the JTAG and NMI pins as GPIO, which can lock the debugger out of the processor and make it permanently unable to be debugged or re-programmed [3].

Interrupt Control for GPIO: The interrupt capabilities of each GPIO port are controlled by a set of **seven registers**. These registers are used to select the source of the interrupt, its polarity, and the edge properties. When one or more GPIO inputs cause an interrupt, a single interrupt output is sent to the interrupt controller for the entire GPIO port. For edge-triggered interrupts, software must clear the interrupt to enable any further interrupts. For a level-sensitive interrupt, the external source must hold the level constant for the interrupt to be recognized by the controller. Carefully review section 10.2.2 on p.608 for the complete procedure of handling an interrupt event.

The following three registers are used to define the edge or sense that causes interrupts:

GPIO_PORTF_IS – define interrupt sensing type; 0 = edge, 1 = level (p.617)
GPIO_PORTF_IBE – define single edge or both edges (p.618)
GPIO_PORTF_IEV – define interrupt event; 0 = falling, 1 = rising (p.619)

Controlling interrupts and checking the status are done using the following four registers:

GPIO_PORTF_IM – interrupt mask to individually enable/disable interrupt (p.620)

GPIO_PORTF_RIS – raw interrupt status (p.621)
GPIO_PORTF_MIS – masked interrupt status (p.622)
GPIO_PORTF_ICR – acknowledge interrupt flag, must be cleared in software by writing '1' to the appropriate bit of GPIO (p.623)

The circuit diagram is given in Fig. 2. Open a new project in Keil uVision. Make sure to use the Startup.s file provided on the course website. Write a C-program to accomplish the objectives. In your main program, configure Ports A, B, and F accordingly. Make sure to unlock PF0. Now carefully configure the interrupt event for GPIO Port F (especially, for PF0 and PF4). Make sure to enable pull-up resistors on Port F. Use "GPIOPortF_Handler ()" function as the interrupt service routine (ISR). Inside the ISR, check the source of interrupt by checking the bits of GPIO_PORTF_MIS and perform actions accordingly. In this problem, you may not need to use GPIO_PORTF_RIS at all. Make sure to clear the interrupt flag before exiting the ISR.

Once the program is fully written and successfully compiled, load it into the LaunchPad. Press the switches to demonstrate the counting-alarm action. Use two 7-segment displays to display the number of customer.

Now, collect one sensor circuit from the tech office (already built for you, shown in Fig. 3). Connect it to PF0 and PF4 ports of the LaunchPad. Demonstrate the customer counter.

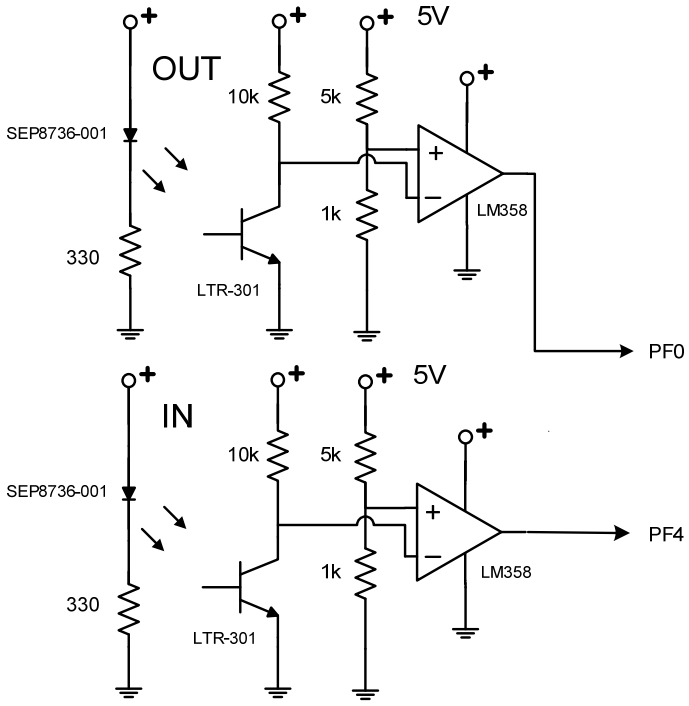

Fig. 3 Sensor circuit for customer counter (already built by James)

References

1. Stellaris® LM4F120 LaunchPad User Manual (SPMU289A–Revised December 2012): http://www.ti.com/tool/sw-ek-lm4f120xl
2. LM4F120H5QR Datasheet (Aug 29, 2012): http://www.ti.com/lit/gpn/lm4f120h5qr
3. Helpful resources: http://users.ece.utexas.edu/~valvano/arm/index.htm

END