Lab 2: Using Interrupts

Objectives

- Use GPIO Interrupts to read the status of push buttons
- Use SysTick Interrupts to drive a recurring action (LED flashes)
- Configure ARM Cortex NVIC
- Use the datasheet to lookup I/O registers

Experiment 1: GPIO Interrupts

E.1.1 Introduction

SW1 and SW2 on TI LaunchPad are switches connected to the PF4 and PF0 pins, respectively. We will use these two switches to show examples of interrupt programming. Interrupt numbers 16 to 255 are assigned to the peripherals. In TM4C123 microcontrollers, **interrupt #30** (interrupt vector #46) is assigned to The **GPIO port F** (PORTF). Although PORTF has 5 pins, there is only **one Interrupt Service Routine (ISR) assigned to the entire PORTF**. It is the job of our ISR to find out which pin caused the interrupt.

Upon Reset, all the interrupts are disabled. To enable any interrupt, we need three steps (Figure 1):

1) Enable the interrupt for a specific peripheral module.
2) Enable the interrupts at the NVIC module.
3) Enable interrupts globally.

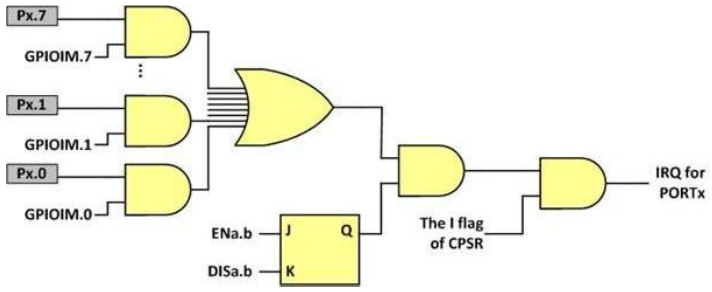

Figure 1. Interrupt Enabling for a PORT

When an input pin is connected to an external device to be used for interrupt, we have 5 choices for trigger point. They are:

1) Low-level trigger[1] (active Low level),
2) High-level trigger (active High level),
3) Rising-edge trigger[2] (positive-edge going from Low to High),
4) Falling-Edge trigger (negative-edge going from High to Low),
5) Both edge (rising and falling) trigger.

[1] A level-triggered interrupt module always generates an interrupt whenever the level of the interrupt source is asserted.
[2] An edge-triggered interrupt module generates an interrupt only when it detects an asserting edge of the interrupt

Use the **GPIO Interrupt Sense (GPIOIS)** register and the **GPIO Interrupt Event (GPIOIEV)** to set the desired option.
With Keil µVision IDE, a new project will get an assembly startup code `startup_TM4C123.s` created by the project wizard. For each interrupt, there is a **dummy interrupt handler** written that does not perform any thing and will never return from the handler. The addresses of these interrupt handlers are listed in the interrupt vector table named `__Vectors` in the file. To write an interrupt handler, one has to find out the name of the dummy interrupt handler in the interrupt vector table and reuse the name of the dummy interrupt handler. The linker will overwrite the interrupt vector table with the new interrupt handler. In the case of PORTF interrupt, the interrupt handler name is **`GPIOF_Handler`**. The interrupt handler is written with a format of a function in C language.

It is critical to have the interrupt handler **clear the interrupt flag** before returning from interrupt handler. Otherwise, the interrupt appears as if it is still pending and the interrupt handler will be executed again and again forever and the program hangs. The PORTF interrupt flag is cleared by writing a 1 to the bit that corresponds to the pin that triggered the interrupt in the **GPIO Interrupt Clear Register (GPIOICR)**. In the case of SW1 (PF4) and SW2 (PF0), the interrupt flags are cleared by:

```
GPIOF->ICR |= 0x11;
```

The **ARM Cortex writes** are **buffered**. That means a write does not take effect immediately. It may take many clock cycles before the write to ICR to clear the interrupt flags. When the program returns from the interrupt handler and the interrupt flag is still pending, the interrupt handler will be executed again. One way to ensure that interrupt flags are cleared before returning from interrupt handler is to **perform a read** to force the write to take effect.

E1.2 Description

Using the TIVA C launchpad board, we will develop an application that toggles the green LED of PF3 continuously. When an interrupt comes from SW1 or SW2, the LED color changes to the violet color of PF2,1 instead. And so on **between green and violet colors**.

```
#include "TM4C123GH6PM.h"
void delayMs(int n);
unsigned int col = 0;
unsigned int colors[] = {0x08, 0x06};

int main(void)
{
    SYSCTL->RCGCGPIO |= 0x20;   /* enable clock to PORTF */
    /* PORTF0 has special function, need to unlock to modify */
    GPIOF->LOCK = 0x4C4F434B;   /* unlock commit register */
    GPIOF->CR |= 0x01;           /* make PORTF0 configurable */
    GPIOF->LOCK = 0;            /* lock commit register */

    /* configure PORTF for switch input and LED output */
    GPIOF->DIR &= ~0x11;        /* make PORTF4,0 input for switch */
    GPIOF->DIR |= 0x0E;         /* make PORTF3, 2, 1 output for LEDs */
    GPIOF->DEN |= 0x1F;         /* make PORTF4-0 digital pins */
    GPIOF->PUR |= 0x11;         /* enable pull up for PORTF4,0 */
```

```
    /* configure PORTF4, 0 for falling edge trigger interrupt */
    GPIOF->IS  &= ~0x11;        /* make bit 4, 0 edge sensitive */
    GPIOF->IBE &= ~0x11;        /* trigger is controlled by IEV */
    GPIOF->IEV &= ~0x11;        /* falling edge trigger */
    GPIOF->ICR |= 0x11;         /* clear any prior interrupt */
    GPIOF->IM  |= 0x11;         /* unmask interrupt for PF4,PF0 */

    /* enable interrupt in NVIC and set priority to 3 */
    NVIC->IP[30] = 3 << 5;        /* set interrupt priority to 3 */
    NVIC->ISER[0] |= 0x40000000;  /* enable IRQ30 (D30 of ISER[0]) */

    __enable_irq(); /* global enable IRQs */

    /* toggle the green/violet LED colors continuously */
    while(1)
    {
        GPIOF->DATA |= colors[col];
        delayMs(500);
        GPIOF->DATA &= ~ 0x0e;   /* turn all LEDs off */
        delayMs(500);
    }
}

/* SW1 is connected to PF4 pin, SW2 is connected to PF0. */
/* Both of them trigger PORTF interrupt */
void GPIOF_Handler(void)
{
    volatile int readback;
    col = ! col;
    GPIOF->ICR |= 0x11;     /* clear the interrupt flag before return */
    readback = GPIOF->ICR;  /* a read to force clearing of interrupt flag */
}

/* delay n milliseconds (16 MHz CPU clock) */
void delayMs(int n)
{
      SysTick->LOAD = 15999*n;
      SysTick->CTRL = 0x5; /*Enable the timer and choose sysclk */
      while((SysTick->CTRL & 0x10000) == 0) /*wait till Count flag is set */
      { }
      SysTick->CTRL = 0; /*Stop the timer (Enable = 0) */
}
```

Note: The Port Pins C0-3, D7 and F0/E7 are locked pins for specific functionality of JTAG, NMI and NMI respectively. To use them in GPIO or any other function they need to be unlocked and commit register be set.

Experiment 2: SysTick Interrupt

E.2.1 Introduction

Another useful interrupt in ARM is the SysTick. The SysTick timer (Figure 2) was used in Lab 1 to implement the delay function. In this experiment we will learn how to use the SysTick interrupt.

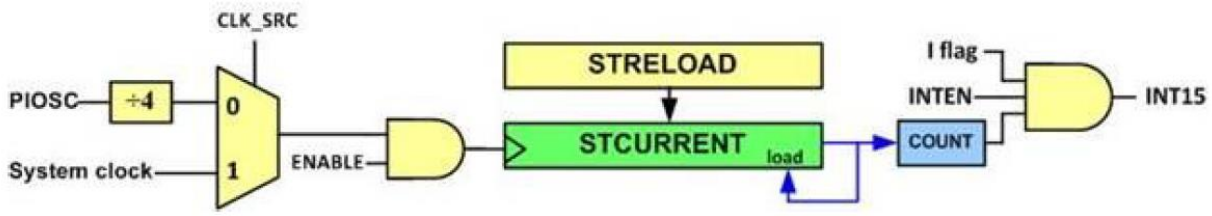

Figure 2. SysTick Timer

If INTEN=1 (bit 1 of the `STCTRL` register), when the `COUNT` flag is set, it causes an interrupt via the NVIC. The SysTick interrupt can be used to initiate an action on a periodic basis. This action is performed internally at a fixed rate without external signal. For example, in a given application we can use SysTick to read a sensor every 100 msec. Examining interrupt vector table for ARM Cortex, we see the SysTick is the interrupt **#15** assigned to the system itself.

E2.2 Description

In this experiment we will use the SysTick to toggle the red LED of PF1 every second. We need the RELOAD value of 16,000,000-1. We assume the CPU clock is 16MHz (default frequency; 1/16MHz=62.5ns). The COUNT flag is raised every 16,000,000 clocks and an interrupt occurs. Then the RELOAD register is loaded with 16,000,000-1 automatically.

```
#include "TM4C123GH6PM.h"

int main (void)
{
    /* enable clock to GPIOF at clock gating control register */
    SYSCTL->RCGCGPIO |= 0x20;
    /* enable the GPIO pins for the LED (PF3, 2, 1) as output */
    GPIOF->DIR |= 0x0e;
    /* enable the GPIO pins for digital function */
    GPIOF->DEN |= 0x0e;

    /* Configure SysTick */
    SysTick->LOAD = 16000000-1; /* reload with number of clocks per second */
    SysTick->CTRL = 7;       /* enable SysTick interrupt, use system clock */

    __enable_irq();             /* global enable interrupt */

    while (1)
    {
    }
}

void SysTick_Handler(void)
{
    GPIOF->DATA ^= 0x02;           /* toggle the red LED */
}
```

Experiment 3: GPIO and SysTick Interrupts

Update Experiment 1 to handle the time delays in the same way as Experiment 2 does, using SysTick interrupt instead of polling. Also give PORTF the highest interrupt priority.

Lab Report [10 pts]
(Deadline: Monday of next week 11:59 pm) (Individual submission)

1. [1 pts] Provide your C code of experiment 3 conducted in the lab.
2. [4.5 pts] Develop an application same as the one in lab report 1 Q2 using interrupts only (no polling allowed). Provide your code along with a small sized video of the application running on the Launchpad.
3. [4.5 pts] Develop an application same as the one in lab report 1 Q3 using interrupts only (no polling allowed). Provide your code along with a small sized video of the application running on the Launchpad.