BIRZEIT UNIVERSITY

Department of Electrical and Computer Engineering

ENCS4110 | Computer Design LAB

Experiment No. 7: GPIO (General Purpose Input/Output) External Interrupts

Objectives

The main objective of this experiment is to introduce students to the two main methods of dealing with digital inputs (e.g., switches or push-buttons); polling and interrupt techniques. Students will learn how to read the state of the onboard switches with the two methods and control the onboard LED's.

How to use Switch as a digital Input?

Mechanical switches are commonly used to feed any parameters to the digital systems. The switches can be interfaced to a microcontroller using digital inputs. The software program for switch interfacing can be implemented using one of the following two methods.

- Polling based method
- Interrupt based method

We will discuss polling based switch interfacing in this tutorial. Before proceeding further, it is important to first make ourselves familiar with the physical behavior of switch and we will describe switch bouncing next, which is one of the critical attribute of its physical behavior.

Switch Bouncing

Electrical switches that use mechanical contacts to close or open a circuit are subject to bouncing of the contacts. Switch inputs are asynchronous and are not electrically clean. When a hardware switch is pressed the mechanical contact of the switch that is made to complete the electrical connection will start bouncing. This bouncing effect will read a single press in the software as multi presses of the switch.

The software will get confused about how many times the switch was pressed. There are software and hardware solutions to this problem. The hardware solution of this issue implies using a simple RC filter. The values of the resistor and capacitor are chosen, such that, the input will be captured after the bouncing period is over. The bouncing effect of a switch is shown in the figure below.

Figure 1

The above figure is the exact elaboration of the bouncing effect of the switch. At the beginning, the switch is at off (0) state. When the switch is activated, it will result in multiple bouncing is as is obvious from the figure, before actually coming to steady ON (1) state. The same is the case when the switch is deactivated. This issue may not cause any problem where you are working solely with hardware, but while working with the GPIO pins of the TIVA LaunchPad, one press will be interpreted as multiple presses, and the results may not be in accordance with the desired or required results.

This is one of the most important points to be considered while working with switches. If we are interested in using the built-in switch of the board, we must configure the corresponding pin of the board as an input pin. The pin will read data from the switch, and according to the data obtained from the switch, it will control the built-in LED of the board, which is configured as an output.

Controlling an LED with a push button using Tiva Launchpad

Firstly, let us see a simple example to control LED connected with PF1 pin using switch-one, which is connected with the PF4 pin of PORTF. That means, whenever a user presses the push button that is connected with the PF0 pin of TM4C123G6PM microcontroller, the LED will turn on. Moreover, as soon as the user releases the push button, the LED turns off.

Pin	Function
PF1	LED- Red
PF4	On-Board Switch-2

GPIO Pins as Digital Input Registers Configuration

In this experiment, we will use the register definition header file available in Keil that contains TM4C123G6PM microcontroller general purpose and peripheral register definitions instead of creating our own register definition file. In the previous experiment, we give you a general concept of how direct pointers dereferencing is used to update the microcontroller peripherals' registers values by using registers memory addresses. **TM4C123G6PM.h** header file contains a listing of all peripheral register's memory addresses. Hence, we can use this header file instead of creating our own. However, you should have an idea of how microcontroller peripheral registers are accessed through direct memory dereferencing via pointers.

Include Header File

Now let us write a code to control red LED with switch-2 of tiva LaunchPad. First, include the header file of TM4C123GH6PM microcontroller like this:

```
#include "TM4C123GH6PM.h"
```

After that, we initialize an unsigned integer variable with the name of “state”. It will be used to hold the state of the button.

```
unsigned int state;
```

GPIO Pins Clock Enable Register

As we discussed in the previous experiment, by default, the clock option is disabled for all GPIO ports of TM4C123GH6PM microcontroller to save power. However, we can enable the clock for each port using the **RCGCGPIO** register. Setting the 5th bit of the **RCGCGPIO** register enables the clock for PORTF.

```
SYSCTL->RCGCGPIO |= 0x20; // enable clock to GPIOF
```

GPIO Lock and Commit Registers

The next register is the **GPIOLOCK** register. It enables write access to the **GPIOCR** register. In order to unlock access to the GPIOCR register, we must initialize the GPIOLOCK register with **0x4C4F434B** value.

```
GPIOF->LOCK = 0x4C4F434B;   // unlockGPIOCR register
```

Pull-Up Resistor Register

Because we will be using an internal pull-up register with a PF4 pin which will be used as a digital input pin. GPIOPUR register is used to enable or disable internal pull-up register with any GPIO pin. But to enable write to GPIOPUR, we first need to enable GPIOCR register. Otherwise, write operation to GPIOPUR register will not commit.

```
GPIOF->CR = 0x01; // Enable GPIOPUR register enable to commit
GPIOF->PUR |= 0x10; // Enable Pull Up resistor PF4
```

Note: TM4C123GH6PM microcontroller also has support for internal pull-down resistor and we can use GPIOPDR to enable and disable pull-down resistor.

Direction Control Register

GPIODIR register sets the direction of the digital pin as either input or output. Writing 1 to a particular bit of GPIOPDR, sets the pin as a digital output pin and writing 0, sets the pin as a digital input.

```
GPIOF->DIR |= 0x02; //set PF1 as an output and PF4 as an input pin
GPIOF->DEN |= 0x12; // Enable PF1 and PF4 as a digital GPIO pins
```

Fourth bit will become 1 if the switch is pressed and otherwise remains 0.

```
state = GPIOF->DATA & 0x10;
```

Send the inverted value of the PF4 pin to the PF0 pin. The value is inverted. Because the switch is low active, the LED is high active.

```
GPIOF->DATA = (~state>>3);    //put it on red LED
```

Push Button Interfacing Complete Code

This is a complete code to control an LED with a push button. Copy this code and upload it to the TM4C123G LaunchPad. The red LED will turn on as you press the switch No. two.

```
//Program 1
#include "TM4C123GH6PM.h"

int main(void)
{
   unsigned int state;
   SYSCTL->RCGCGPIO |= 0x20;   /* enable clock to GPIOF */
   GPIOF->LOCK = 0x4C4F434B;   // unlockGPIOCR register
   GPIOF->CR = 0x01;           // Enable GPIOPUR register enable to commit
   GPIOF->PUR |= 0x10;        // Enable Pull Up resistor PF4
   GPIOF->DIR |= 0x02;          //set PF1 as an output and PF4 as an input pin
   GPIOF->DEN |= 0x12;         // Enable PF1 and PF4 as a digital GPIO pins

    while(1)
    {
        state = GPIOF->DATA & 0x10;
        GPIOF->DATA = (~state>>3);    /* put it on red LED */
    }
}
```

The following four steps are the main configuration steps of switch initialization:

1. Enabling the clock
2. Enabling the data register for pin0 or pin4
3. Enabling the direction register as GPIO input register
4. Enabling the PAD for digital operation and enabling the corresponding pull up register.

GPIO Interrupts

General-purpose input-output pins are the vital components of embedded systems. GPIO pins allow easy integration of external components with microcontrollers. Input pins allow microcontrollers to receive information from the external world and output pins are used to display information or control devices, such as, motors, etc.

Why do we need to use TM4C123 GPIO Interrupts?

In the last tutorial on controlling an LED with push button using TM4C123 Tiva C LaunchPad, we have seen an example to control an onboard LED of Tiva LaunchPad using onboard switches, such as, SW1 (PF0) and SW2 (PF4). In that tutorial, the TM4C123 microcontroller keeps checking the status of the push button by polling PF0 and PF4 bits of PORTF of TM4C123G microcontroller. But one of the main drawbacks of the polling method is that microcontroller will have to check the status of input switches on every sequential execution of the code

or keep monitoring continuously (polling method). Therefore, external or GPIO interrupts are used to synchronize external physical devices with microcontrollers.

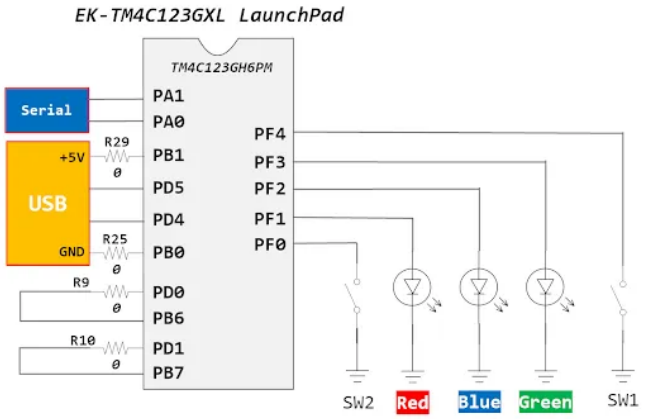
Hence, instead of checking the status of input switches continuously, a GPIO pin which is configured as digital input, can be initialized to produce an interrupt whenever the state of the switch changes. A source of interrupt trigger can be on falling edges, rising edges or both falling and rising edges, level triggered, as well.

In summary, the use of external GPIO interrupts makes the embedded system event driven, responsive and they make use of microcontroller's processing time and resources efficiently.

TM4C123GH6PM Microcontroller GPIO Interrupts

TM4C123GH6PM microcontroller has six GPIO ports such as PORTA, PORTB, PORTC, PORD, PORTE, and PORTF. Each pin of every GPIO port can be configured as a source of external interrupt. In this tutorial, we will see how to configure PF0 and PF4 pins as an external interrupt source. However, the procedure to configure other GPIO interrupts will remain the same.

TM4C123 Tiva C LaunchPad has two onboard switches SW1 and SW2 which are connected with PF0 and PF4 GPIO pins. These input switches will be used to demonstrate GPIO interrupt programming examples.



Find GPO Interrupt Number

TM4C123 microcontroller has an integrated Nested Vectored Interrupt Controller (NVIC), which manages all interrupt requests which are issued either by a processor (exceptions) or peripherals (IRQs). TM4C123GH6PM microcontroller supports 76 peripheral interrupts (some are reserved) and each interrupt has a unique number assigned to it. This interrupt number is defined inside the startup file and header file of TM4C123GH6PM.

NVIC identifies each exception or peripheral interrupt by its numbers. If you see table 2.9 in the datasheet of TM4C123GH6PM MCU, you will find the unique number assigned to each exception and peripheral interrupt. As you can see in the second column of the figure shown below, the interrupt number of GPIO PORTF is 30.

Vector Number	Interrupt Number (Bit in Interrupt Registers)	Vector Address or Offset	Description
45	29	0x0000.00B4	Flash Memory Control and EEPROM Control
46	30	0x0000.00B8	GPIO Port F
47-48	31-32	-	Reserved
49	33	0x0000.00C4	UART2
50	34	0x0000.00C8	SSI1
51	35	0x0000.00CC	16/32-Bit Timer 3A
52	36	0x0000.00D0	16/32-Bit Timer 3B
53	37	0x0000.00D4	I²C1
54	38	0x0000.00D8	QEI1
55	39	0x0000.00DC	CAN0
56	40	0x0000.00E0	CAN1
57-58	41-42	-	Reserved
59	43	0x0000.00EC	Hibernation Module
60	44	0x0000.00F0	USB
61	45	0x0000.00F4	PWM Generator 3
62	46	0x0000.00F8	µDMA Software
63	47	0x0000.00FC	µDMA Error
64	48	0x0000.0100	ADC1 Sequence 0
65	49	0x0000.0104	ADC1 Sequence 1
66	50	0x0000.0108	ADC1 Sequence 2

How to Configure External Interrupts of TM4C123?

In this section, we will learn how to configure TM4C123 microcontroller GPIO interrupts using their respective configuration registers.

However, one important thing to note here is that PORTF has eight pins and only one interrupt number is assigned to all the pins of PORTF. This is because the separate interrupt register is used to distinguish which pin of PORTF caused the interrupt.

Enabling GPIO Interrupts

Before enabling any peripheral interrupt, we must enable the source of interrupt requests for this particular interrupt source using the NVIC interrupt control register. In the interrupt vector table, there is an interrupt enable bit number for each interrupt source. We can enable these interrupt bits using NVIC interrupt enable registers. There are four NVIC interrupts enable registers such as EN0 (ISER [0]), EN1 (ISER [1]), EN2 (ISER [2]) and EN3 (ISER [3]).

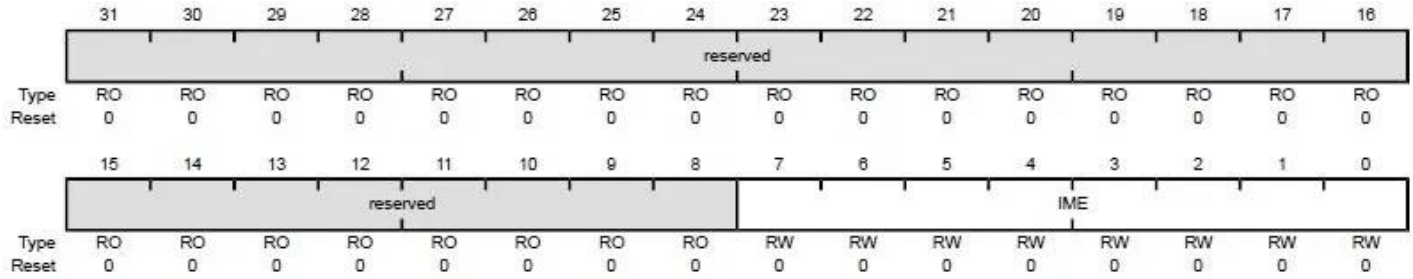
The ENn (ISER[n]) registers enable interrupts and shows which interrupts are enabled. Bit0 of EN0 corresponds to Interrupt 0; bit31 corresponds to Interrupt 31. Bit 0 of EN1 (ISER[1]) corresponds to Interrupt 32; bit31 corresponds to Interrupt 6, and so on other two registers.

For example, the interrupt number of PORTF is IRQ30. Hence, the corresponding NVIC interrupt enable register of PORTF is ISER [0] and setting bit 30 of ISER [0] will enable the PORTF interrupt.

```
NVIC->ISER [0] |= (1<<30); /*Enable PORTF Interrupt IRQ30 */
```

After enabling interrupt in NVIC register, enable the interrupt of the peripheral, which you want to use. For example, we will be using GPIOF pins interrupt. In order to enable GPIO interrupt GPIO interrupt mask enable

register is used. First eight bits of GPIOIM register enable or disable interrupt functionality for each pin as shown in figure below:



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IME	RW	0x00	GPIO Interrupt Mask Enable

Value	Description
0	The interrupt from the corresponding pin is masked.
1	The interrupt from the corresponding pin is sent to the interrupt controller.

For example, we will use PF0 and PF4 pins of PORTF to get SW1 and SW2 status on interrupt. This line enables interrupt for PF0 and PF4 pins.

```
GPIOF->IM |= (1<<4)| (1<<0);
```

GPIO Interrupt Edge or Level Triggered Setting (GPIOIS)

As we mentioned earlier, external GPIO interrupts of TM4C123G microcontroller can be configured in four modes:

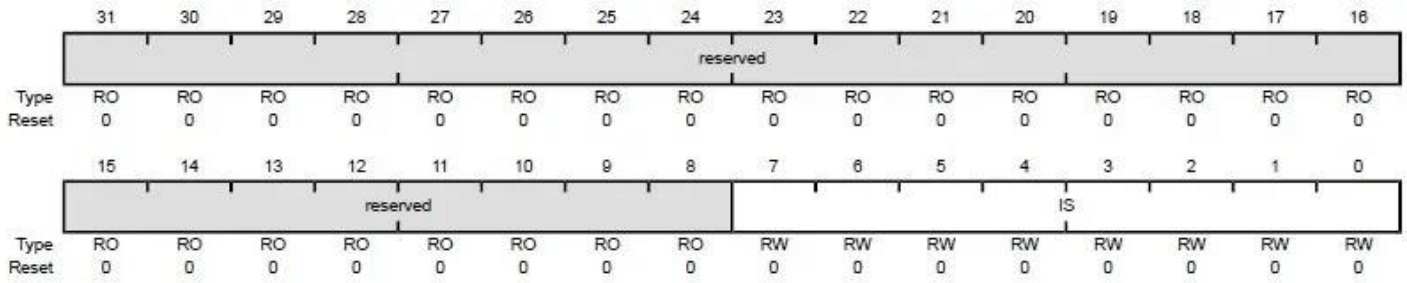
1-Positive edge triggered

2-Negative edge triggered

3-Positive Level (active high)

4-Negative Level (active low)

GPIO interrupt sense register is used to configure respective pins either as a level or edge triggered. Setting a bit in the GPIOIS register configures the corresponding pin to detect levels while clearing a bit configures the corresponding pin to detect edges.



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IS	RW	0x00	GPIO Interrupt Sense

Value	Description
0	The edge on the corresponding pin is detected (edge-sensitive).
1	The level on the corresponding pin is detected (level-sensitive).

For example, this line initializes the PF0 and PF4 pins as edge triggers.

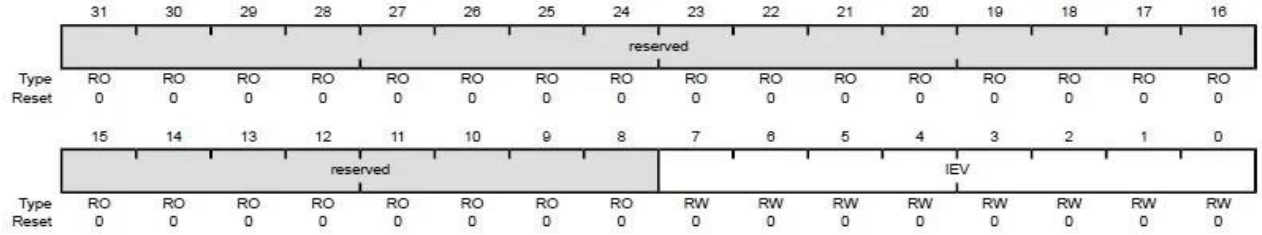
GPIOF->IS &= ~ (1<<4)|~ (1<<0); / make bit 4, 0 edge sensitive */*

Similarly, this line initializes the PF0 and PF4 pins as level triggers

```
GPIOF->IS |= (1<<4)| (1<<0); /* make bit 4, 0 level sensitive */
```

GPIO Interrupt Event Register (GPIOIEV)

GPIOIS register only configures the pins as level or edge triggered. However, we should also configure the pin either as a positive/negative edge or positive/negative level triggered. Setting the respective bit in GPIOIEV register configures the corresponding pin to detect positive edge or positive level depending on in which mode pin is configured in GPIOIS register. Similarly, clearing the respective bit in GPIOIEV register configures the corresponding pin to detect negative edge or negative level.



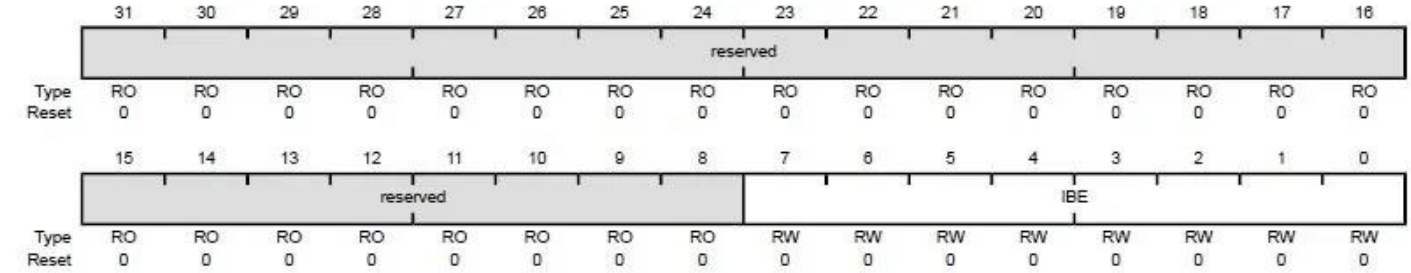
Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IEV	RW	0x00	GPIO Interrupt Event

Value	Description
0	A falling edge or a Low level on the corresponding pin triggers an interrupt.
1	A rising edge or a High level on the corresponding pin triggers an interrupt.

```
GPIOF->IEV &= ~(1<<4)|~(1<<0); /*PF0, PF4 falling edge trigger */
```

GPIO Interrupt Both Edges (GPIOIBE)

We can also configure GPIO pins to cause interrupt on both positive and negative edges. GPIOIBE register is used to configure each pin to cause an interrupt on both edges. Setting a bit in the GPIOIBE register configures the corresponding pin to detect both rising and falling edges, regardless of the corresponding bit in the GPIO Interrupt Event (GPIOIEV) register (see page 666). Clearing a bit configures the pin to be controlled by the GPIOIEV register.



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IBE	RW	0x00	GPIO Interrupt Both Edges

Value	Description
0	Interrupt generation is controlled by the GPIO **Interrupt Event (GPIOIEV)** register (see page 666).
1	Both edges on the corresponding pin trigger an interrupt.

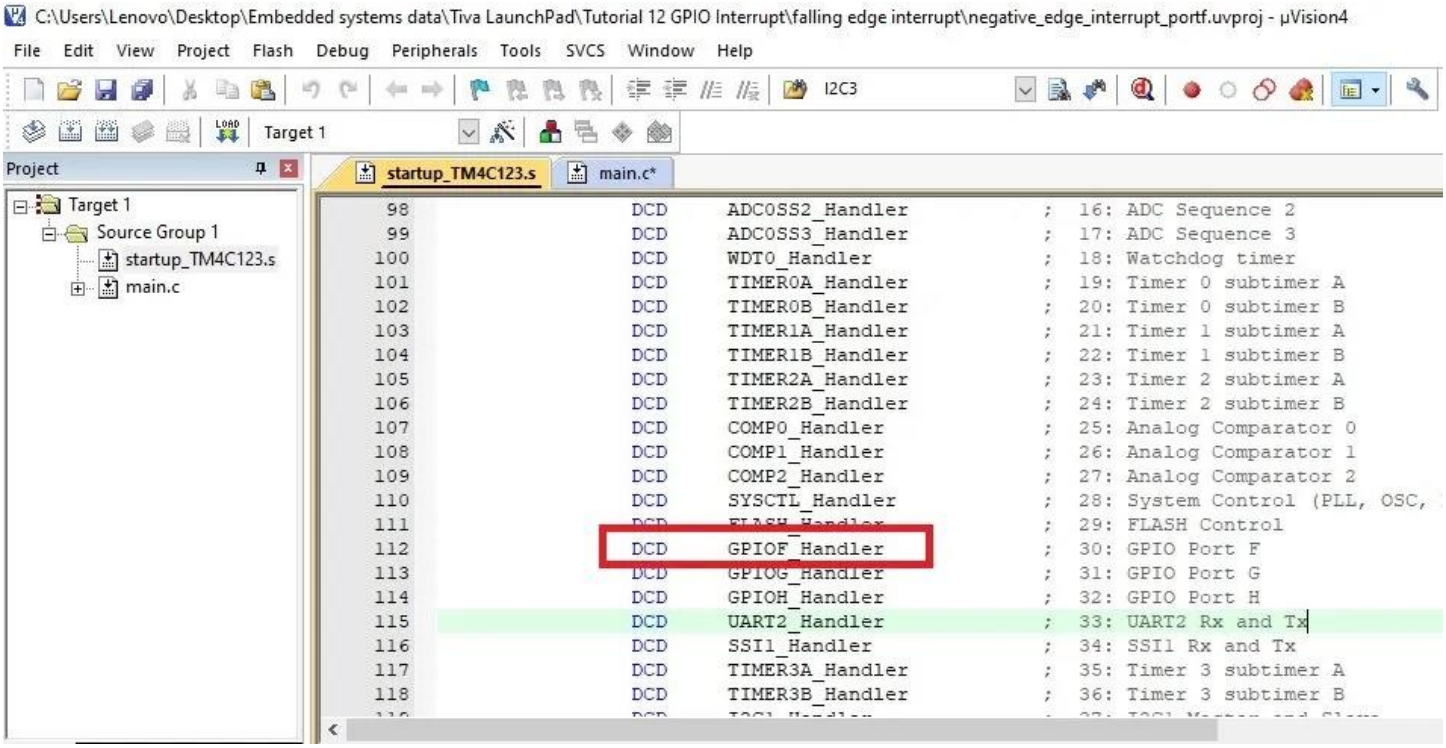
These lines configure the PF0 and PF4 pins to detect positive edges.

```
GPIOF->IS &= ~(1<<4)|~(1<<0); /* make bit 4, 0 edge sensitive */
GPIOF->IBE &=~(1<<4)|~(1<<0); /* trigger is controlled by IEV */
GPIOF->IEV &= ~(1<<4)|~(1<<0); /* falling edge trigger */
```

GPIO Interrupt Handler Name

In the startup file of TM4C123G microcontroller, there is a vectored mapped table, which contains the starting addresses of all system exceptions and peripheral interrupt service routines. This file also contains the dummy implementations of exception Handlers or interrupt Handlers. These dummy implementations do nothing except an infinite loop and we can modify these dummy interrupt handlers inside our main code.

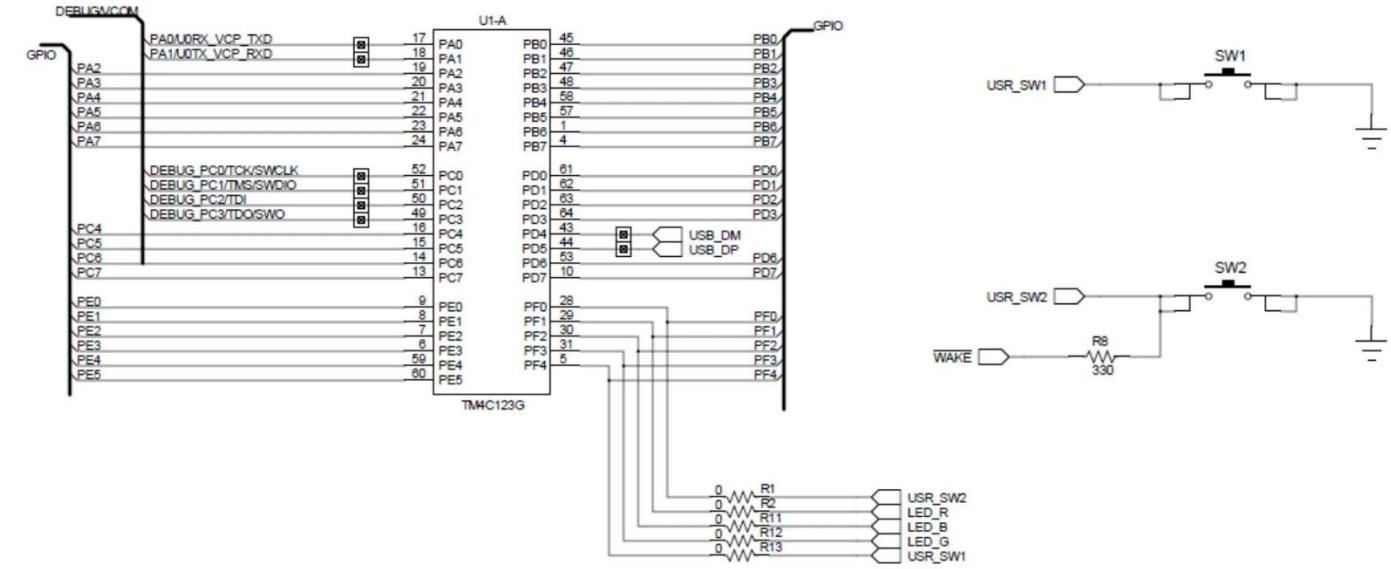
Thus, in order to implement a specific functionality inside the respective interrupt handler function, we should find the name of the corresponding peripheral interrupt in the startup file. For example, if you open the startup file of TM4C123 and explore it, you will find that the name of PORTF interrupt handler is GPIOF_Handler. Inside our main code, we will use GPIOF_Handler as a function name for GPIOF interrupt service routine.



The linker script file will automatically replace the dummy implementation of GPIOF_Handler inside the startup file with our defined implementation.

TM4C123 Tiva C GPIO Interrupt Example

For demonstration purposes, in this example, we will configure PF0 and PF1 pins to cause interrupts on negative edges. On TM4C123 Tiva C LaunchPad, there are two onboard switches, namely, SW1 and SW2. SW1 is connected with the PF4 pin, and SW2 is connected with PF0 pin as shown in figure below.



One end of both input switches is connected to the ground and the other end to PF0 and PF4 pins. TM4C123GH6PM microcontroller provides internal pull-up resistors, which can be configured using GPIOPUR register. This line enables the pull-up resistor for PF4 and PF0 pin.

```
GPIOF->PUR |= (1<<4)|(1<<0);  /* enable pull up for PORTF4, 0 */
```

By enabling pull-up resistors, active high signals will appear on PRF4 and PF0 pins when switches are not pressed. However, when we press the switch button, the transition of the signal occurs from active high to active low, which means a negative edge will occur. We will configure PF0 and PF4 pins to detect negative edges and cause an interrupt. Hence, whenever the user presses either of the push buttons, an interrupt will be generated and TM4C123 microcontroller executes the interrupt handler function of PORTF.

Inside the PORTF interrupt handler function, the onboard green LED of TM4C123 Tiva C LaunchPad is turned on, when SW1 is pressed. On the other hand, it is turned off when if SW2 is pressed.

External Interrupt Code TM4C123 MCU

This code controls the green LED of the TM4C123 Tiva LaunchPad based on SW1 and SW2 states. Both switches are used to generate external interrupt signals on negative edges (falling edge). If the interrupt is caused by SW1 (PF4), the LED will turn on. On the other hand, if the interrupt is caused by SW2(PF0), the LED will turn off.

```
//Program 2
/*PORTF PF0 and PF4 fall edge interrupt example*/
/*This GPIO interrupt example code controls green LED with switches SW1 and SW2 external interrupts */

#include "TM4C123.h"                // Device header

int main(void)
{
```

```
        SYSCTL->RCGCGPIO |= (1<<5);   /* Set bit5 of RCGCGPIO to 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 */


  /*Initialize PF3 as a digital output, PF0 and PF4 as digital input pins */

  GPIOF->DIR &= ~(1<<4)|~(1<<0);  /* Set PF4 and PF0 as a digital input pins */
  GPIOF->DIR |= (1<<3);         /* Set PF3 as digital output to control green LED */
  GPIOF->DEN |= (1<<4)|(1<<3)|(1<<0);          /* make PORTF4-0 digital pins */
  GPIOF->PUR |= (1<<4)|(1<<0);          /* enable pull up for PORTF4, 0 */

  /* configure PORTF4, 0 for falling edge trigger interrupt */
  GPIOF->IS  &= ~(1<<4)|~(1<<0);       /* make bit 4, 0 edge sensitive */
  GPIOF->IBE &=~(1<<4)|~(1<<0);        /* trigger is controlled by IEV */
  GPIOF->IEV &= ~(1<<4)|~(1<<0);       /* falling edge trigger */
  GPIOF->ICR |= (1<<4)|(1<<0);        /* clear any prior interrupt */
  GPIOF->IM  |= (1<<4)|(1<<0);        /* unmask interrupt */

  /* enable interrupt in NVIC and set priority to 3 */
  NVIC->IP[30] = 3 << 5;     /* set interrupt priority to 3 */
  NVIC->ISER[0] |= (1<<30);  /* enable IRQ30 (D30 of ISER[0]) */

  while(1)
  {
                        // do nothing and wait for the interrupt to occur
  }
}

/* SW1 is connected to PF4 pin, SW2 is connected to PF0. */
/* Both of them trigger PORTF falling edge interrupt */
void GPIOF_Handler(void)
{
 if (GPIOF->MIS & 0x10) /* check if interrupt causes by PF4/SW1*/
  {
    GPIOF->DATA |= (1<<3);
    GPIOF->ICR |= 0x10; /* clear the interrupt flag */
   }
  else if (GPIOF->MIS & 0x01) /* check if interrupt causes by PF0/SW2 */
  {
   GPIOF->DATA &= ~0x08;
   GPIOF->ICR |= 0x01; /* clear the interrupt flag */
  }
}
```

Now create a new project using Keil uvision and upload this code to TM4C123G Tiva C Launchpad. After that, press the reset button of the LaunchPad. Finally, check whether the code is working, by pressing SW1 and SW2.

Differentiating which GPIO pin causes Interrupt

As we mentioned earlier, there is only one interrupt service routine for each GPIO port. For example, PORTF has one interrupt handler function that is GPIOF_Handler(). Interrupt caused by all the pins of PORTF executes the same GPIOF_Handler() interrupt service routine. Now the question is how to differentiate which of the pins of PORTF forces the GPIOF_Handler() function to execute?

Well, that is pretty easy, TM4C123GH6PM microcontroller has GPIO masked interrupt status register (GPIOMIS). This register provides the status of the interrupt caused by each pin. The first eight bits of this register corresponds to the PIN0 to PIN7 of each GPIO interrupt status.

For example, if the interrupt service routine of PORTF is called due to PF0 pin, the 0th bit of GPIOMIS register will be 1, and if it is caused by PF4 pin, the 4th bit of GPIOMIS register will set automatically. Hence, by checking the value of each bit of GPIOMIS register inside the PORTF interrupt handler function, we can identify which pin causes this particular interrupt.

In this code, we used GPIOMIS register to differentiate either PF0 or PF4 causing the external interrupt, and based on this, the code executes only the true condition block of if-else.

```
/* SW1 is connected to PF4 pin, SW2 is connected to PF0. */

/* Both of them trigger PORTF falling edge interrupt */

void GPIOF_Handler(void)

{

  if (GPIOF->MIS & 0x10) /* check if interrupt causes by PF4/SW1*/

    {

      GPIOF->DATA |= (1<<3);

      GPIOF->ICR |= 0x10; /* clear the interrupt flag */

                          }

    else if (GPIOF->MIS & 0x01) /* check if interrupt causes by PF0/SW2 */

    {

     GPIOF->DATA &= ~0x08;

     GPIOF->ICR |= 0x01; /* clear the interrupt flag */

    }

}
```

In-Lab Exercises:

1-Modify program 1 (using polling technique) to toggle the green LED with the press of a switch SW1. The switch on PF4 will be configured as input and LED on PF3 (green LED) will be used as output. At each press of the switch, the LED will toggle its present state i.e., the LED will turn ON if it was OFF previously.

Note: The needed time for the user to press the push button is approximately 200 ms. The speed of the TM4C123G controller is very high, so 'while true' loop will be executed a lot of times through this period (200 ms), which means that the button has been pressed many times, but in fact the user did it just once! Therefore, you need some delay after button clicking detection.

2- Modify program 2 (using Interrupt technique) so the onboard LEDs light on in the following sequence R→B→G, when the onboard SW1 is pressed. On the other hand, the LEDs light on in reverse sequence G→B→R, when SW2 is pressed. Each LED should ON for the same amount of time, e.g. around 0.5 sec.

3- Write a program that changes the color of the onboard LED by using the two onboard push button keys. When the board is turned on, only the RED LED should is on. When the right-side key is pressed, the green LED should be ON (alone), and when the left-side key is pressed, the blue LED should be ON alone.