Embedded Systems
Postlab 6 Interrupts

By Rajendra Singh (111601017), final year, CSE, IIT PALAKKAD

20 September 2019

Objective

To learn how to setup and use interrupts and exceptions in the KL25Z Microcontroller using ARM Cortex M0+.

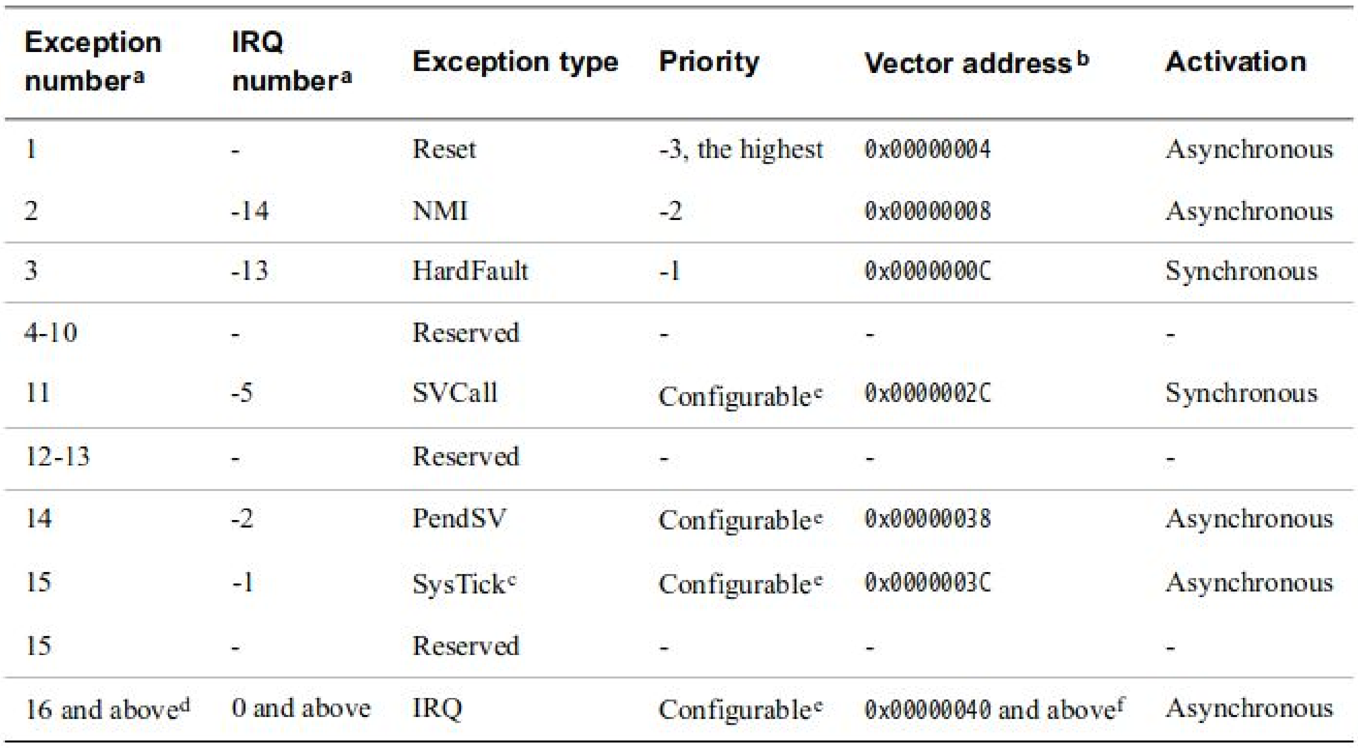
Abstract

The Cortex-M0+ processor supports interrupts and system exceptions. The processor and the Nested Vectored Interrupt Controller (NVIC) prioritize and handle all exceptions. Some of the exceptions generated by the Cortex M0+ processor are Reset, Non-Maskable Interrupt (NMI), HardFault interrupt, SysTick timer interrupt and so on. All exceptions have an associated priority, with a lower priority value indicating a higher priority or configurable priorities for all exceptions except Reset, HardFault, and NMI.

In this lab, we will try to trigger interrupts such as **SysTick timer**.. If the device implements the SysTick timer, a SysTick exception is an exception the system timer generates when it reaches zero. Software can also generate a SysTick exception. In an OS environment, the device can use this exception as system tick.

Introduction

Each exception can be possibly in four states i.e. Inactive, Pending, Active, Active and Pending. The different types of exceptions are Reset, NMI, HardFault, SV Call, PendSV, SysTick, Interrupt (IRQ). The properties of different exception types are given in Table 4.1



Exception number[a]	IRQ number[a]	Exception type	Priority	Vector address[b]	Activation
1	-	Reset	-3, the highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	HardFault	-1	0x0000000C	Synchronous
4-10	-	Reserved	-	-	-
11	-5	SVCall	Configurable[c]	0x0000002C	Synchronous
12-13	-	Reserved	-	-	-
14	-2	PendSV	Configurable[c]	0x00000038	Asynchronous
15	-1	SysTick[c]	Configurable[c]	0x0000003C	Asynchronous
15	-	Reserved	-	-	-
16 and above[d]	0 and above	IRQ	Configurable[c]	0x00000040 and above[f]	Asynchronous

Table 4.1

The processor handles exceptions using ISRs, Fault handler, System Handlers. Out of which, HardFault is the only exception handled by Fault handler and System Handlers can handle both SysTick and HardFault.

Exception entry occurs when there is a pending exception with sufficient priority and either:

- the processor is in Thread mode
- the new exception is of higher priority than the exception being handled, in which case the new exception preempts the exception being handled.

The stack frame includes the return address. This is the address of the next instruction in the interrupted program. This value is restored to the PC at exception return so that the interrupted program resumes.

The processor performs a vector fetch that reads the exception handler start address from the vector table. When stacking is complete, the processor starts executing the exception handler. At the same time, the processor writes an EXC_RETURN value to the LR. This indicates the stack pointer corresponding to the stack frame and the operation mode the processor was in before the entry occurred. The stack frame contains the following information:

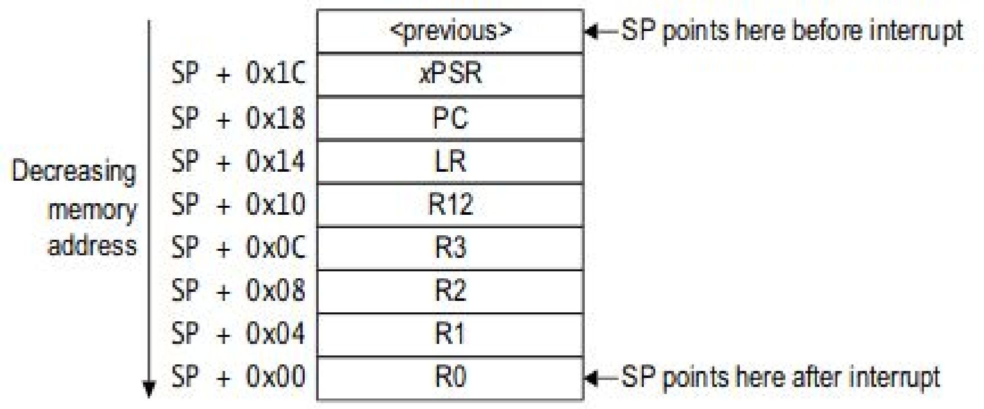


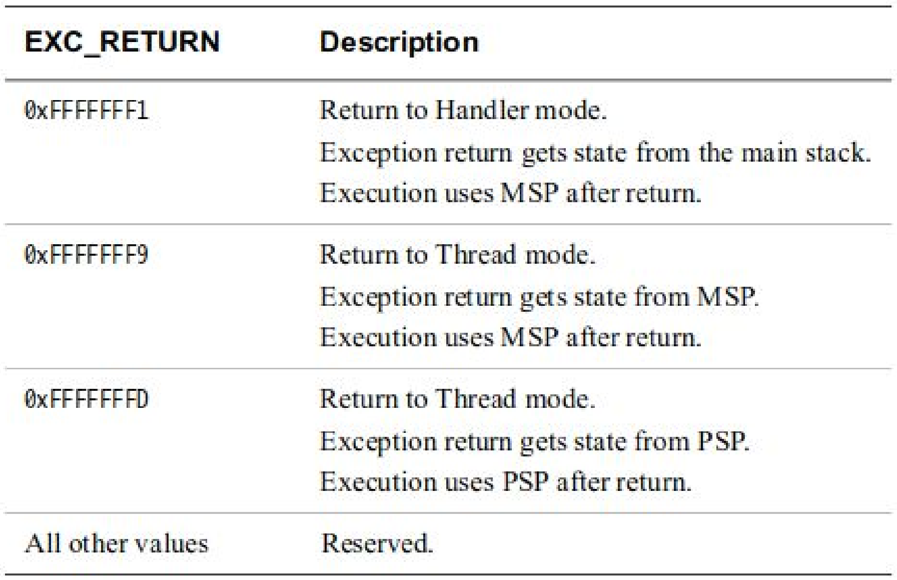
Table 4.2

If no higher priority exception occurs during exception entry, the processor starts executing the exception handler and automatically changes the status of the corresponding pending interrupt to **active**. If another higher priority exception occurs during exception entry, the processor starts executing the exception handler for this exception and does not change the pending status of the earlier exception. This is the **late arrival case**.

Exception return occurs when the processor is in Handler mode and execution of one of the following instructions attempts to set the PC to an EXC_RETURN value:

- a POP instruction that loads the PC
- a BX instruction using any register.

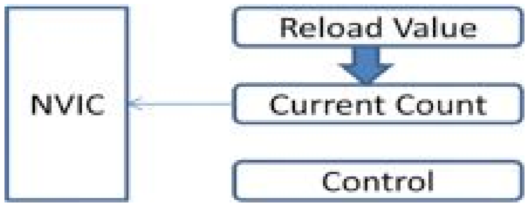
The processor saves an **EXC_RETURN** value to the LR on exception entry. The exception mechanism relies on this value to detect when the processor has completed an exception handler. Bits[31:4] of an EXC_RETURN value are 0xFFFFFFF. When the processor loads a value matching this pattern to the PC it detects that the operation is a not a normal branch operation and, instead, that the exception is complete. Therefore, it starts the exception return sequence. Bits[3:0] of the EXC_RETURN value indicate the required return stack and processor mode, as shown in Table 4.3



EXC_RETURN	Description
0xFFFFFFF1	Return to Handler mode. Exception return gets state from the main stack. Execution uses MSP after return.
0xFFFFFFF9	Return to Thread mode. Exception return gets state from MSP. Execution uses MSP after return.
0xFFFFFFFD	Return to Thread mode. Exception return gets state from PSP. Execution uses PSP after return.
All other values	Reserved.

Table 4.3

SysTick timer is a 24-bit countdown timer with auto reload . Once started, the SysTick timer will countdown from its initial value, once it reaches zero it will raise an interrupt and a new count value will be loaded from the reload register. The main purpose of this timer is to generate a periodic interrupt for a real-time operating system (RTOS) or other event driven software. If you are not running an RTOS, you can also use it as a simple timer peripheral.



All faults result in the HardFault exception being taken or may cause lockup. Faults related to memory access instructions can be caused by:

- Invalid address - check the address value
- Data alignment issue (the processor has attempted to carried an unaligned data accesses)
- For Cortex-M0+ processor, please check for memory access permission (e.g. unprivileged access to the NVIC register), or MPU permission violations.
- Bus components or peripheral returned an error response for other reason.

You can also get a HardFault exception if you executed SVC instruction in an exception handler with same or higher priority than the SVC priority level. The fault happened because the current context does not have the right priority level for the SVC.

EXERCISE

First part : This aim of this exercise is to illustrate the use of a SysTick timer and trigger an interrupt that flashes the **single** on-board LED green.

Second part : This aim of this exercise is to illustrate the use of a SysTick timer and trigger an interrupt that flashes **three** led one after other.

Third part : This aim of this exercise is to illustrate the **uart** based interrupt that flashes led on-board.

CODE:

Below code can also be found at github.

```
/*

*   Author : Rajendra singh

*   Roll no: 111601017

*/



#include<MKL25Z4.h> //INCLUDING LIBRARY



//global

static unsigned volatile int delayCtr=0;

int ticket[4];



//==============================green B19==============================//

//=======init========//
```

```
void led_green_init(){

    SIM->SCGC5 |=(1<<10); //TO ACTIVATE PORT B

    PORTB->PCR[19]|=(1<<8); //SETTING 8TH BIT TO 1

    PORTB->PCR[19]&=0xFFFFF9FF; //SETTING 9TH, 10TH BIT TO 0, OTHER UNCHANGED

    PTB->PDDR |= (1<<19); //18TH BIT = 1

}

//=======ON========//

void led_green_on(){

    PTB->PCOR |= (1<<19 ); //CLEAR 18PIN VALUE

}

//=======OFF========//

void led_green_off(){

    PTB->PDOR |= (1<<19); //CLEAR 18PIN VALUE

}

//=======TOGGLE========//

void led_green_toggle(){

    PTB->PTOR |= (1<<19); //CLEAR 18PIN VALUE

}
```

```
//=============================blue D1=============================//

//=======init========//

void led_blue_init(){

    SIM->SCGC5 |=(1<<12); //TO ACTIVATE PORT B

    PORTD->PCR[1]|=(1<<8); //SETTING 8TH BIT TO 1

    PORTD->PCR[1]&=0xFFFFF9FF; //SETTING 9TH, 10TH BIT TO 0, OTHER UNCHANGED

    PTD->PDDR |= (1<<1); //18TH BIT = 1

}

//=======ON========//

void led_blue_on(){

    PTD->PCOR |= (1<<1 ); //CLEAR 18PIN VALUE

}

//=======OFF========//

void led_blue_off(){

    PTD->PDOR |= (1<<1); //CLEAR 18PIN VALUE

}

//=======TOGGLE========//

void led_blue_toggle(){

    PTD->PTOR |= (1<<1); //CLEAR 18PIN VALUE
```

```
}


//==============================DELAY old==============================//

void delay_old(long long int d){

    while(d--);

}


//==============================DELAY better==============================//

void delay_better(unsigned int d){

    delayCtr=0;

    while(delayCtr<d);

}


//==============================SysTick_Handler==============================//

void SysTick_Handler(void){

    int i;

    delayCtr++;

    for(i=0; i<4; i++){

        ticket[i]++;
    }

}


//============================SysTick_init==============================//

void SysTick_init(){

    SysTick->LOAD = 20971 - 1;//load the RVR reg of systick

    SysTick->VAL = 0x00;

    SysTick->CTRL = 0x07;//enable the timer and interrupt

}


void UART0_init(void)

{

    SIM->SCGC4 |= (1<<10); // set 10th index(index start from 0) bit = 1, enable
clock for UART0 by 1<<10 or 0x400

    SIM->SOPT2 |= (1<<26); // set 26th index bit = 1, Selecting MCGFLLCLK clock or
MCGPLLCLK/2 as clock source 1<<26 or 0x04000000

    SIM->SOPT2 &= 0xF7FFFFFF; // set 27th index bit = 0, other undisturbed,
F(0111)FFFFFF

    UART0->C2 = 0x00; // Transmitter, Receiver disabled

    UART0->BDH = 0x00; // Baudrate updated

    UART0->BDL = 0x18; //00001101, to write 24, SBR = (clock freq/(OSR*))
    UART0->C4 = 0x0F; //00001111, for OCR of 16, Setting OverSampling Ratio 01111

    UART0->C1 = 0x00; //00000000, no parity

    //UART0->C2 = 0x04; //set 3rd index bit = 1, 00001000, Transmitter disabled &
Receiver enabled - old

    UART0->C2 = 0x36; //set 5rd index bit = 1, 00001000, Transmitter disabled &
Receiver enabled with interrupt - new

    SIM->SCGC5 |= (1<<9); //set 9th index bit = 0, Clock for PORT A Enabled

    PORTA->PCR[1] = (1<<9); //set 9th index bit = 0, MUXing PORT A to use as UART

    PORTA->PCR[1] = 0xFFFFFAFF; //reset

    NVIC->ISER[0] = 0x00001000; //- new

}

void LED_init(void)//Initiating GREEN LED

{

    SIM->SCGC5 |= (1<<10); // enable clock to Port B

    PORTB->PCR[19] |= (1<<8); // MUXing PORT B to use as (PCR19 - 001)

    PORTB->PCR[19] &= 0xFFFFF9FF;

    PTB->PDDR |= (1<<19); //Setting Pin 19 as input and taking XOR

    PTB->PDOR |= (1<<19); //initially off, Corresponding bit 19 in PDORB is set to
logic 1.

}

void LED_set(char value)
```

```
{
    if (value =='g') //Green LED ON
    {
        PTB->PCOR |= (1<<19); //Corresponding bit 19 in PDORB is cleared to logic 0
    }else if(value =='o') //Green LED OFF
    {
        PTB->PDOR |= (1<<19); //Corresponding bit 19 in PDORB is set to logic 1.
    }
}



void UART0_IQRHandler(){
    char c=UART0->D;
    if (c =='g') //Green LED ON
    {
        led_green_on();
        //delay_better(1000);
        delay_old(100000);
        led_green_off();
    }else if(c=='b') //blue LED OFF

    {

        led_blue_on();

        delay_better(1000);

        delay_old(100000);

        led_blue_off();

    }

}



//=============================MAIN==============================//

int main(void){



    SystemCoreClockUpdate(); //updating clock from PLL

    long long int n; //NUMBER OF BLINK



    //INIT ALL LED

    led_green_init();

    led_blue_init();

    led_green_off();
    led_blue_off();


    //INIT SysTick

    SysTick_init();




    //==================================== first and second experiment
=========================

    /* ------------------------------------ Blinking led using interrupt
-------------------------*/

    led_green_on();

    led_blue_on();

    while(1){

        if(ticket[0]>500){//used for second experiment of blinking multiple led

            ticket[0]=0;

            led_green_toggle();

        }

        if(ticket[1]>1000){

            ticket[1]=0;

            led_blue_toggle();
        }

        //led_green_toggle(); //used for very first simple experiment of blinking 
single led

        //delay_old(1e6); //normal delay function

        //delay_better(1000); //improved delay function

    }

    led_green_off();


//=====================================================================================
=======



    //===================================== third experiment 
=====================================

    /* ----------------------------------- interrupt with uart 
--------------------------------*/

    /*

    UART0_init(); // Initiating UART0 as receiver

    while (1);

    */



}
```

Analysis:

The SysTick interrupt service routine simply decrements a counter (SysTickCounter) at each interrupt. When the count reaches 0, it toggles the green led on the FRDM-KL25Z board. The result is that the green led changes state depending on the Delay value. The interrupt request takes place when the current value switches from 1 to 0, but reloading only takes place at the next rising edge of the clock. This process will go on periodically as the code ends up in an infinite loop after initialising GPIO Pin PTB19 and SysTick Timer.

Conclusion

- The SysTick counter reload and current value are not initialized by hardware. This means the correct initialization sequence for the SysTick counter is:
    - Program the reload value.
    - Clear the current value.
    - Program the Control and Status register.

Reference

1) Google
2) **Cortex M0+ Generic User's Guide**
3) **Cortex M0+ Technical Reference Manual**