**University of Engineering and Technology Lahore**
**Department of Mechatronics and Control Engineering**

# Interrupts in TM4C123GH6PM CORTEX-M4 Microcontroller.

**Embedded Systems Lab**

## OBJECTIVE:

At the end of this lab, you should be able to:

- Understand What is Meaning of Interrupt in Microcontrollers.
- How Many Interrupts are in TM4C123GH6PM
- GPIO Interrupts in TM4C123GH6PM Cortex_M4
- Systick Interrupts in TM4C123GH6PM

## EVALUATION CRITERIA:

- Exp. Understanding
- Exp. Performance
- Accuracy\Precision
- Software Handling
- Equipment/Apparatus Handling

## APPARATUS:

- Laptop\PC with µVision Keil IDE installed
- Tiva C Series TM4C123G LaunchPad Evaluation Kit

## REFERENCES:

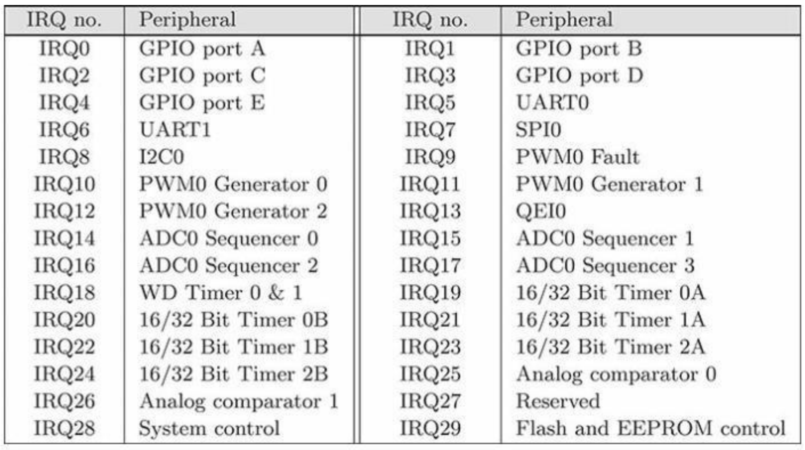
- Introduction to ARM Cortex-M Microcontrollers by Jonathan W. Valvano
- Real-Time Interfacing to ARM Cortex-M Microcontrollers by Jonathan W. Valvano
- ARM Microprocessor Systems – Cortex-M Architecture, Programming, and Interfacing by M Tahir
- Linker Script File of TM4C123GH6PM.
- STARTUP file of TM4C123GH6PM.
- Tiva™ C Series TM4C123GH6PM Microcontroller Data Sheet[1]
- Tiva™ C Series TM4C123G LaunchPad Evaluation Board User's Guide[2]
- Wikipedia.com
- https://www.tutorialspoint.com/embedded_systems/es_interrupts.htm
- www.microcontrollerslab.com

[1] TM4C123GH6PM Microcontroller Datasheet
[2] https://www.ti.com/lit/ug/spmu296/spmu296.pdf

## What is interrupt in microcontroller.

As its name suggests that interrupt is an event or a signal to the processor from software or hardware that suspends the execution of the main application program and starts executing the interrupt service routine. Whenever an event or exception happens, the corresponding peripheral or hardware requires a response from the processor. The response from the processor is implemented as a function call known as interrupt service routine (ISR). NVIC approves the interrupt request and compels the processor to execute the corresponding ISR or Exception Handler. ISR tells the processor or controller what to do when the interrupt occurs. In other words, whenever an interrupt occurs from a particular source, the processor executes a corresponding piece of code (function) or interrupt service routine. After successful Execution of the interrupt the processor start executing the statement where program was suspended cause of interrupt.

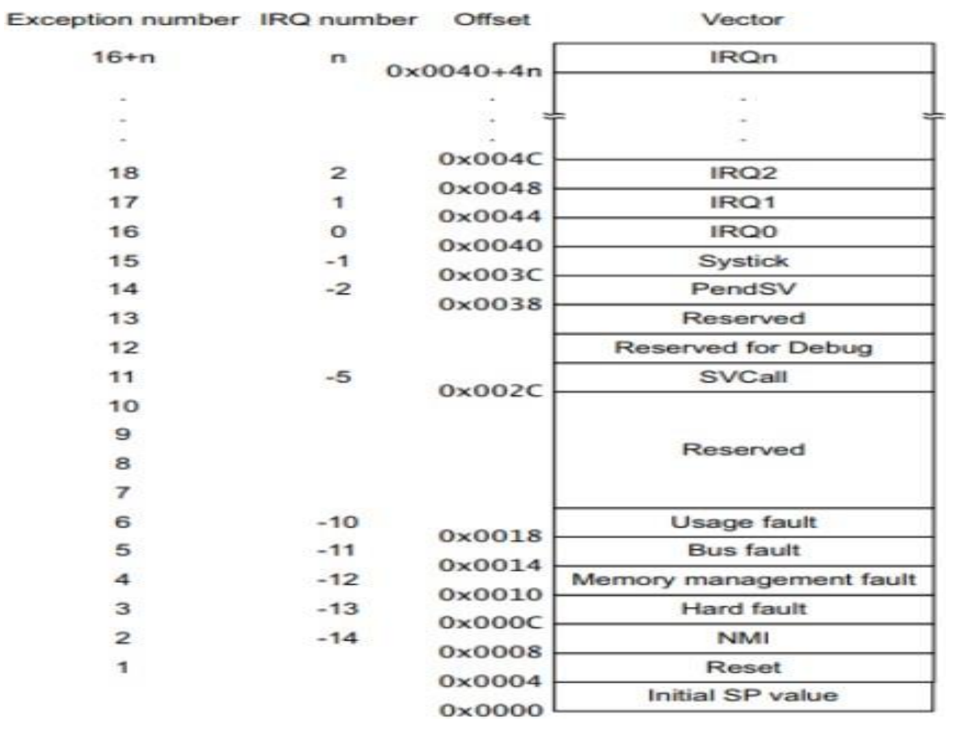
Cortex-M4 microcontroller supports 15 system exceptions and 240 peripheral or external interrupts**.** They are assigned to different peripherals including timers, communication interfaces (UART, SPI, i2c, CAN bus etc.), data converter modules, general purpose I/Os etc. A specific microcontroller can use an arbitrary number of external interrupts depending on its peripherals. A total of 139 interrupts are supported by the NVIC controller of Texas Instrument's TM4C123 microcontroller and are labeled as IRQ0 to IRQ138. Table lists the assignment of IRQ numbers to the peripheral devices. This unique IRQ number is used to determine the address of corresponding interrupt service routine (ISR) in the vector table. Assuming interrupt vector table base address is 0x00000000(By Default), then nth IRQ vector address is obtained as 4n + 64. This address when defined in terms of exception number m, is obtained as 4m, providing the relation between exception number and IRQ number as m = n +16.



| IRQ no. | Peripheral | IRQ no. | Peripheral |
|---|---|---|---|
| IRQ0 | GPIO port A | IRQ1 | GPIO port B |
| IRQ2 | GPIO port C | IRQ3 | GPIO port D |
| IRQ4 | GPIO port E | IRQ5 | UART0 |
| IRQ6 | UART1 | IRQ7 | SPI0 |
| IRQ8 | I2C0 | IRQ9 | PWM0 Fault |
| IRQ10 | PWM0 Generator 0 | IRQ11 | PWM0 Generator 1 |
| IRQ12 | PWM0 Generator 2 | IRQ13 | QEI0 |
| IRQ14 | ADC0 Sequencer 0 | IRQ15 | ADC0 Sequencer 1 |
| IRQ16 | ADC0 Sequencer 2 | IRQ17 | ADC0 Sequencer 3 |
| IRQ18 | WD Timer 0 & 1 | IRQ19 | 16/32 Bit Timer 0A |
| IRQ20 | 16/32 Bit Timer 0B | IRQ21 | 16/32 Bit Timer 1A |
| IRQ22 | 16/32 Bit Timer 1B | IRQ23 | 16/32 Bit Timer 2A |
| IRQ24 | 16/32 Bit Timer 2B | IRQ25 | Analog comparator 0 |
| IRQ26 | Analog comparator 1 | IRQ27 | Reserved |
| IRQ28 | System control | IRQ29 | Flash and EEPROM control |

| | | | |
|---|---|---|---|
| IRQ30 | GPIO port F | IRQ31 | Reserved |
| IRQ32 | Reserved | IRQ33 | UART2 |
| IRQ34 | SPI1 | IRQ35 | 16/32 Bit Timer 3A |
| IRQ36 | 16/32 Bit Timer 3B | IRQ37 | I2C1 |
| IRQ38 | QEI1 | IRQ39 | CAN0 |
| IRQ40 | CAN1 | IRQ41 | Reserved |
| IRQ42 | Reserved | IRQ43 | Hibernation module |
| IRQ44 | USB | IRQ45 | PWM0 Generator 3 |
| IRQ46 | uDMA Software | IRQ47 | uDMA error |
| IRQ48 | ADC1 Sequence 0 | IRQ49 | ADC1 Sequence 1 |
| IRQ50 | ADC1 Sequence 2 | IRQ51 | ADC1 Sequence 3 |
| IRQ52-56 | Reserved | IRQ57 | SPI2 |
| IRQ58 | SPI3 | IRQ59 | UART3 |
| IRQ60 | UART4 | IRQ61 | UART5 |
| IRQ62 | UART6 | IRQ63 | UART7 |
| IRQ64-67 | Reserved | IRQ68 | I2C2 |
| IRQ69 | I2C3 | IRQ70 | 16/32 Bit Timer 4A |
| IRQ71 | 16/32 Bit Timer 4B | IRQ72-91 | Reserved |
| IRQ92 | 16/32 Bit Timer 5A | IRQ93 | 16/32 Bit Timer 5B |
| IRQ94 | 32/64 Bit Timer 0A | IRQ95 | 32/64 Bit Timer 0B |
| IRQ96 | 32/64 Bit Timer 1A | IRQ97 | 32/64 Bit Timer 1B |
| IRQ98 | 32/64 Bit Timer 2A | IRQ99 | 32/64 Bit Timer 2B |
| IRQ100 | 32/64 Bit Timer 3A | IRQ101 | 32/64 Bit Timer 3B |
| IRQ102 | 32/64 Bit Timer 4A | IRQ103 | 32/64 Bit Timer 4B |
| IRQ104 | 32/64 Bit Timer 5A | IRQ105 | 32/64 Bit Timer 5B |
| IRQ106 | System Exception | IRQ107-133 | Reserved |
| IRQ134 | PWM1 Generator 0 | IRQ135 | PWM1 Generator 1 |
| IRQ136 | PWM1 Generator 2 | IRQ137 | PWM1 Generator 3 |
| IRQ138 | PWM1 Fault | | |

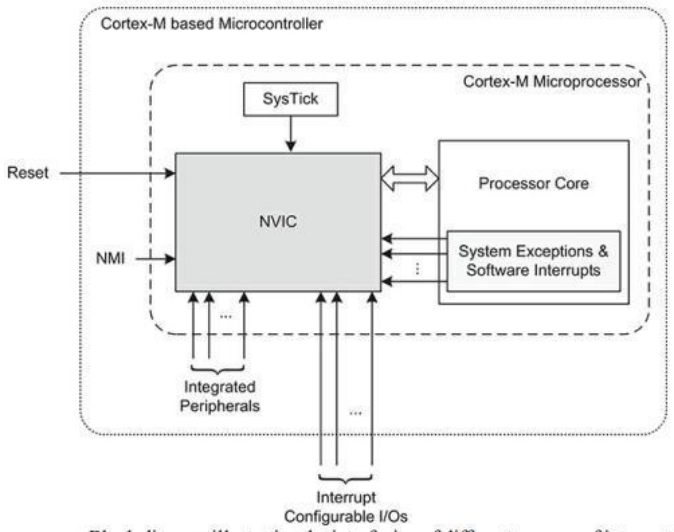
Table Showing Exception Number and IRQ number assigned to each ISR in IVT.



## Hardware and software Interrupts.

A hardware interrupt is an electronic alerting signal sent to the processor from an external device, like a disk controller or an external peripheral. For example, when we press a key on the keyboard or move the mouse, they trigger hardware interrupts which cause the processor to read the keystroke or mouse position.

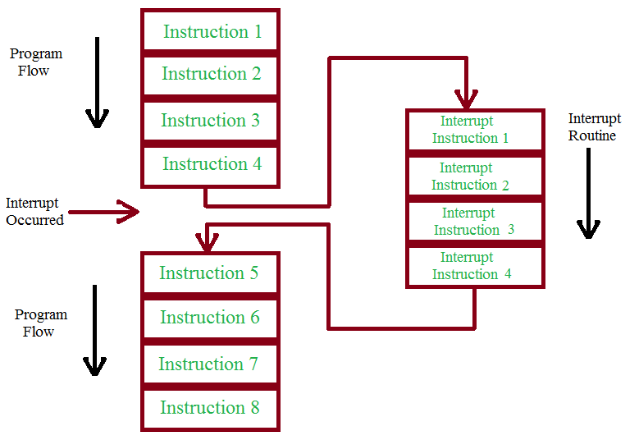
A software interrupt is caused either by an exceptional condition or a special instruction in the instruction set which causes an interrupt when it is executed by the processor. For example, if the processor's arithmetic logic unit runs a command to divide a number by zero, to cause a divide-by-zero exception, thus causing the computer to abandon the calculation or display an error message. Software interrupt instructions work similar to subroutine calls.


Cortex-M based Microcontroller
Cortex-M Microprocessor
SysTick
Reset
NVIC
NMI
Processor Core
System Exceptions & Software Interrupts
Integrated Peripherals
Interrupt Configurable I/Os

Block diagram illustrating the interfacing of different sources of interrupts with nested vectored interrupt controller.

## What happens when Interrupt Occurs?

Whenever a hardware or software exception occurs that specific peripheral or external/internal input requires a response of that interrupt. As a response, a function call occurs and the required response is executed in the form of a piece of code known as Interrupt Service Routine (ISR). After that set of instructions in the service, the routine is executed the control shifts back to the main program.



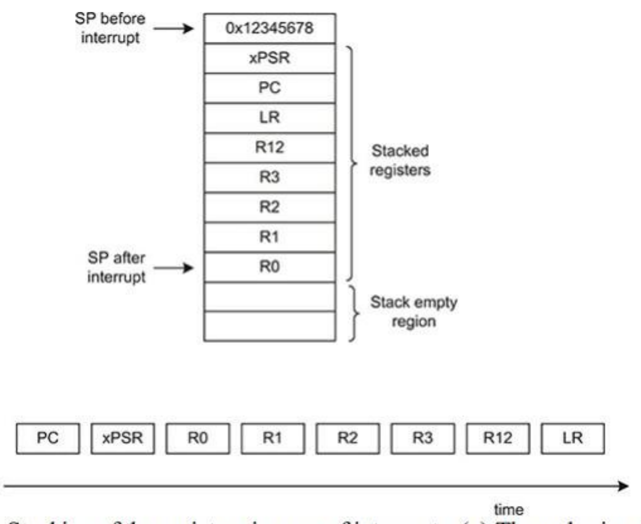
When an exception takes place, a sequence of operations is performed for proper handling of the exception. The following basic steps are involved in that sequence.

i. Execution of the current instruction is either completed or its execution is terminated without completing, depending on the value of interrupt continuable instruction (ICI) field in the PSR register.
ii. The current status is preserved by pushing registers R0-R3, R12, LR, PC, and PSR to the stack.
iii. The processor reads the exception number field from the updated value of PSR register and performs vector fetch by reading the exception handler starting address from the interrupt vector table.
iv. Before the execution of the interrupt service routine begins, the processor updates the stack pointer, link register (LR), and program counter (PC).
v. Specific interrupt service routine corresponding to the interrupt source is performed.
vi. The final step is to exit or return from the interrupt service routine, which also involves popping the registers from the stack

## Register Stacking in Response to Interrupt Occurrence

When an exception takes place, the instruction currently being executed is either finished or terminated (based on the value of ICI field in PSR register) and then registers R0-R3, R12, LR, PC, and program status register (PSR) are pushed on the stack. If the application program executing at the time of occurrence of interrupt or exception is using the process stack pointer, then the process stack will be used for storing the current status. On the other hand, if the application program was using the main stack pointer, then the registers are stacked to the main stack. The set of eight registers pushed to the stack is commonly termed stack frame. The PC and PSR registers are stacked first. This allows updating the exception number field in PSR register by the NVIC to let the processor know about the source of interrupt. Once the processor knows about the interrupt source and has stacked the PC register, it can start fetching the interrupt service routine, by updating the PC, in parallel to stacking of other registers. This is possible due to the fact that register stacking is done using the data bus, while fetching of ISR is performed using the instruction bus.

The order followed to stack the registers in the stack memory region is shown in Figure



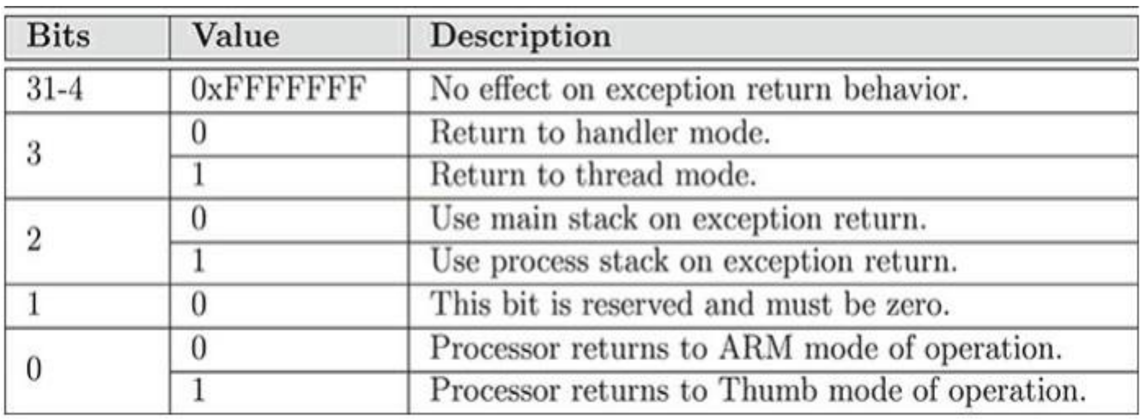
Stacking of the registers in case of interrupts. (a) The order in which eight registers are stacked in memory and (b) time sequence of stacking the registers.

## Updating Registers

After stacking of the registers is completed, few of the stacked registers are updated to their new values. Specifically, PSR, SP, and PC registers are updated during the process of stacking and vector fetching, while the LR register is updated before the execution of ISR. A brief description of this updating of registers is provided below.

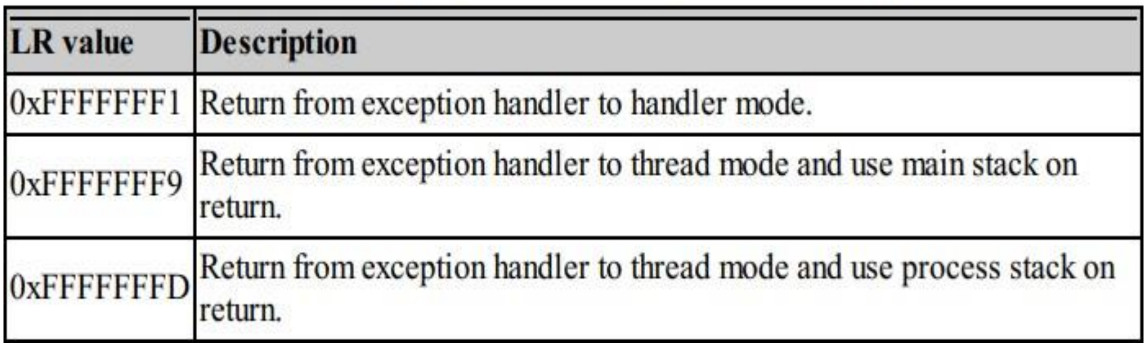
- SP: The stack pointer (either the MSP or the PSP) will be updated to a new value during register stacking. During execution of the interrupt service routine, the MSP will be used if any further stacking of some registers is required.
- PSR: The least significant nine bits of PSR (corresponding to the interrupt program status register, IPSR) will be updated to the new exception number.
- PC: The program counter will be updated to the exception handler and starts fetching the service routine instructions from the location of the exception handler.
- LR: The LR will be updated to a special value called EXC-RETURN. This special value controls the exception or interrupt return type, when returning from the service routine. The least significant 4 bits (5 bits in case of Cortex-M4F processors) of LR are used to provide exception return information.

Description of bit fields on LR register for exception.



| Bits | Value | Description |
|---|---|---|
| 31-4 | 0xFFFFFFF | No effect on exception return behavior. |
| 3 | 0 | Return to handler mode. |
| | 1 | Return to thread mode. |
| 2 | 0 | Use main stack on exception return. |
| | 1 | Use process stack on exception return. |
| 1 | 0 | This bit is reserved and must be zero. |
| 0 | 0 | Processor returns to ARM mode of operation. |
| | 1 | Processor returns to Thumb mode of operation. |

Possible exception returns based on LR register.



| LR value | Description |
|---|---|
| 0xFFFFFFF1 | Return from exception handler to handler mode. |
| 0xFFFFFFF9 | Return from exception handler to thread mode and use main stack on return. |
| 0xFFFFFFFD | Return from exception handler to thread mode and use process stack on return. |

## Exception Return.

After completing the execution of exception or interrupt handler, an exception or interrupt return is performed to restore the system status, by unstacking the registers to resume normal execution of the interrupted program.

- Unstacking: All the registers that have been pushed to the stack are restored during this process. The sequence of POP operation will match the one followed during register stacking. The stack pointer register is also updated during this process.
- NVIC register update: The active bit corresponding to current exception is cleared. If the interrupt input, for external interrupts, is still asserted, the pending bit is set again. As a result, the program execution leads to reentering the interrupt service routine.

## Type difference between interrupt and exception.

Exception and interrupt both terms can be used interchangeably. Although there is a minor difference between interrupt and exception. Interrupts are special types of exceptions which are caused by peripherals or external interrupts such as Timers, GPIO, UART, I2C, etc. On the contrary, exceptions or system exceptions are generated by processor or system.

Exceptions numbered 1-15 are assigned to system exceptions while exceptions numbered from 16 onward are allocated for external interrupts. Some of the system exceptions are assigned fixed priority, while the external interrupts have programmable priority.

## What is interrupt vector table?

The interrupt vector table is a table that contains addresses (function pointers) of interrupt service /routines and exception handler functions. Here vector stand for the addresses of function pointers. In other words, IVT defines where the code of a particular interrupt/exception routine is located in microcontroller memory.

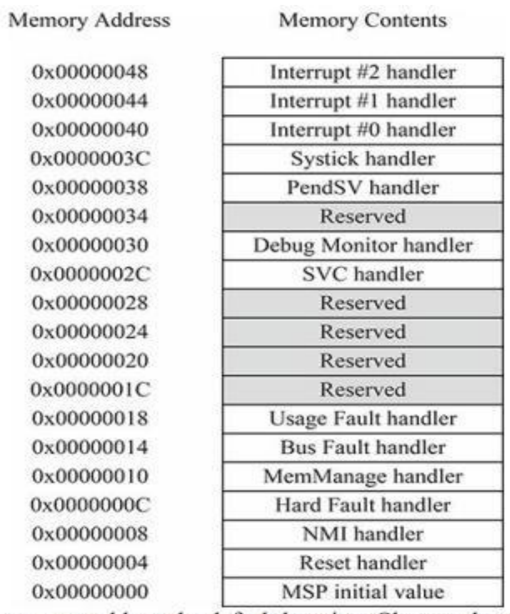
- **Location of the IVT in Microcontroller Memory.**

If you explore the datasheet of TM4C123GH6PM microcontroller (page 107), the interrupt vector table stores at the starting addresses of code memory (starting from 0x0000_0000).

The startup file and a linker script file define the way to store the interrupt vector table at the starting 256 locations of the microcontroller's code memory. The first two entries of the vector table are the initial value of the stack pointer and the address of the reset handler function. Because whenever a microcontroller resets, it performs hardware initialization steps. (i.e. Stack initialization, Memory Allocation etc.)

The vector table and interrupt service routines/exception handlers are defined inside the startup file of a microcontroller. Programmer need not worry about this startup file. This startup file is given in the IDE and added into the application program by default through the compiler.

The figure below shows the interrupt vector table along with their memory addresses and memory contents. Each memory address contains the address of exception handlers. For example, the address 0x0000_003C contains the address location of the systick timer interrupt handler.



| Memory Address | Memory Contents |
|---|---|
| 0x00000048 | Interrupt #2 handler |
| 0x00000044 | Interrupt #1 handler |
| 0x00000040 | Interrupt #0 handler |
| 0x0000003C | Systick handler |
| 0x00000038 | PendSV handler |
| 0x00000034 | Reserved |
| 0x00000030 | Debug Monitor handler |
| 0x0000002C | SVC handler |
| 0x00000028 | Reserved |
| 0x00000024 | Reserved |
| 0x00000020 | Reserved |
| 0x0000001C | Reserved |
| 0x00000018 | Usage Fault handler |
| 0x00000014 | Bus Fault handler |
| 0x00000010 | MemManage handler |
| 0x0000000C | Hard Fault handler |
| 0x00000008 | NMI handler |
| 0x00000004 | Reset handler |
| 0x00000000 | MSP initial value |

Interrupt vector table at the default location. Observe that some of the unused entries are also included to maintain the location of each entry in the vector table.

## What is Nested Vectored Interrupt Controller?

In Cortex-M microcontrollers, a nested vectored interrupt controller usually known as NVIC is used to handle all the interrupts and exceptions that Cortex-M supports.

The nested vectored interrupt controller is basically an integrated part of Cortex-M. We can also configure the interrupt controller according to our needs using specific registers. The mode of operation of most of the interrupt registers is privileged i.e. They can only be accessed in privileged mode, but if the interrupt is a software interrupt than these registers can be accessed in user mode also. The role of NVIC is to manage all low and high priority interrupts in such a way that the higher priority interrupts always gets to execute before a lower priority interrupts even if the lower priority interrupts occurs earlier. For example, if a low priority interrupt has occurred and being executed. While a low priority interrupt is still executing, a higher priority interrupt occurs. ARM CPU will pause the low priority interrupt and starts to execute high priority interrupt

Nesting of interrupts is the major concept when talking about nested vectored interrupt controller. This concept is somewhat similar to nested for-loops, i.e. Processing an interrupt (with higher priority) with in another interrupt (with lower priority). This can be implemented using NVIC as NVIC allows us to set the priority of every interrupt and the interrupts with higher priorities can preempt the interrupts with lower priority resulting in interrupts with in an interrupt and this preemption of interrupts is known as interrupt nesting. As it is mentioned earlier, ARM Cortex-M microcontrollers have 0-255 exceptions/interrupts and each exception has a priority and some exceptions are user-programmable. That means some interrupt will have higher priority than others during program execution. Furthermore, some interrupts can be configured as critical interrupts or non-maskable. That means they cannot be disabled.

Followings are the main responsibilities of NVIC:

- Interrupts handling
- Programmable interrupt feature
- Interrupt tail chaining and interrupt latency.

## NVIC Role in Interrupt Latency control.

Interrupt *latency* is defined as the time interval from the time of interrupt occurrence and the time at which the 1st statement of interrupt service routine is executed. NVIC helps in reducing the interrupt latency by providing a faster response to the application. This in turn executes the interrupts quickly and resumes the program flow.

## NVIC Interrupt Tail-Chaining

This is another name for nesting of interrupts, and it helps in executing the interrupts back to back without the problem of context switching (Register Stacking and Unstacking). Without nested interrupt controller the coming interrupt of same or lesser priority goes in the pending state if an interrupt is already being executed unless that running interrupt completes its service routine and go back to the program and do complete context switch. In nested interrupts, however, we do not have to do this, and the next interrupts got executed within the first interrupt before giving the control back to the calling program.

Cortex-M4 architecture avoids this unnecessary register unstacking/stacking cycle to improve both interrupt latency as well as its power efficiency (due to reduced number of memory read/write operations). This performance improvement in Cortex-M4 processor is achieved by implementing tail-chaining in the NVIC. Tail-chaining allows NVIC to switch from pending interrupt to active interrupt without unstacking/stacking of the registers. Tail-chaining reduces the latency between two interrupt handlers to 6 cycles compared to 22 (10+12) cycles. The 6-cycle latency in tail- chaining is due to the vector fetch and interrupt handler fetch corresponding to the second ISR. Comparison between tail-chaining and conventional interrupt handling is illustrated.

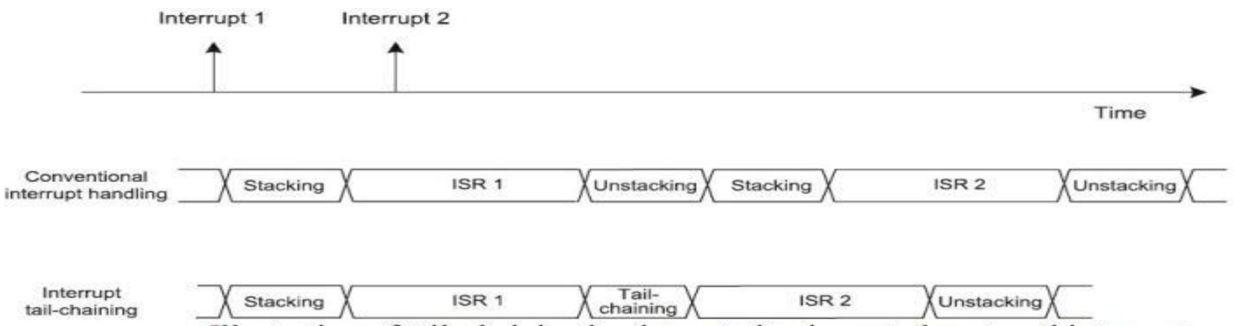


Illustration of tail-chaining implementation in nested vectored interrupt controller.

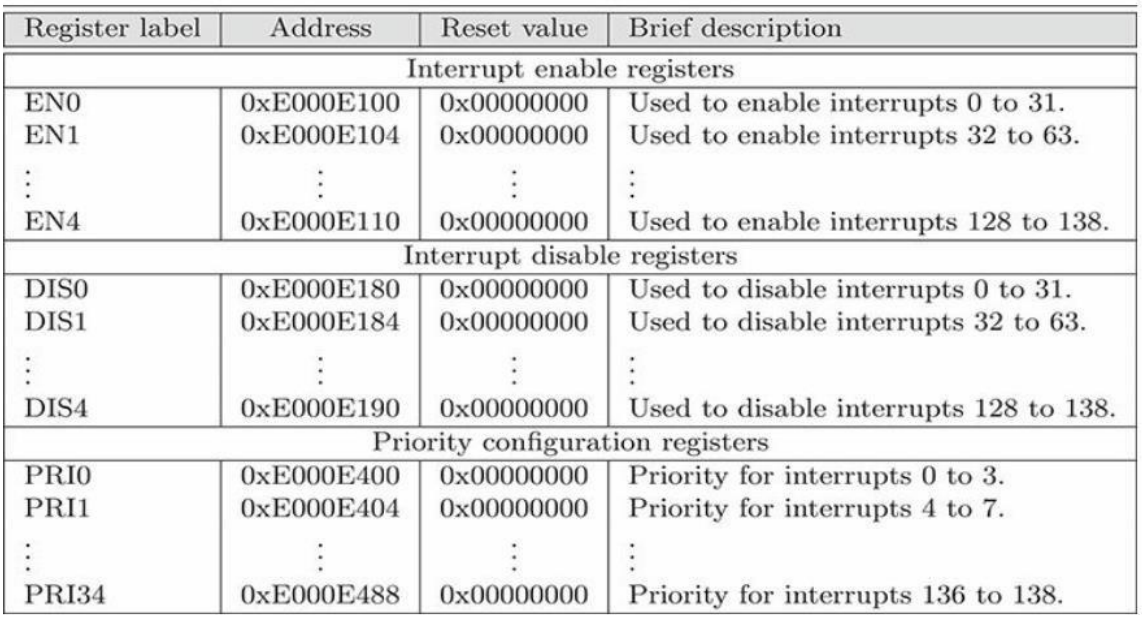
## Configuring Interrupts for Cortex-M Peripherals.

### Processor Interrupt Configuration on TM4C123.

There are three processor registers, FAULTMASK, PRIMASK and BASEPRI that are used for interrupt masking. The default values of these registers are such that all interrupts are enabled. However, in case of critical code execution, that should not be interrupted, global interrupt enabling and disabling can be performed. For that purpose, either the PRIMASK or FAULTMASK register can be used. Register is used for enabling or disabling interrupts selectively, depending on their priorities.

## NVIC Configuration on TM4C123

The NVIC configuration has two main components, namely enabling/disabling device interrupts and assigning priority to the interrupts. In addition, interrupt pending behavior can also be configured using NVIC configuration. Separate NVIC registers are allocated for enabling and disabling of interrupts (IRQ0-IRQ138 or correspondingly exceptions 16 to 154). On the other hand, system exceptions (i.e., exceptions 1 to 15) are configured using system control block (SCB) registers (see SCB register descriptions for address range 0xe000ed04-0xe000ed2c). Some of these system exceptions have programmable priority. The priority for system exceptions can be configured using system priority (SYSPRI) registers (see registers in address space 0xe000ed18-0xe000ed20).
The NVIC registers used for enabling and disabling of interrupts are labeled as ENx and DISx in table given below. Where x can take values from 0 to 4 depending on the specific peripheral interrupt allocation in the vector table. Similarly, priorities can be assigned using priority registers PRIy, where y can take values from 0 to 34 for TM4C123 microcontroller. Table also provides memory address allocation for registers as well as their reset values for enable, disable and priority registers. NVIC has various register for priority and pending, enabling disabling interrupt registers. They are used in case of multiple interrupt Configurations. These registers are available in datasheet Page104.



| Register label | Address | Reset value | Brief description |
|---|---|---|---|
| | | Interrupt enable registers | |
| EN0 | 0xE000E100 | 0x00000000 | Used to enable interrupts 0 to 31. |
| EN1 | 0xE000E104 | 0x00000000 | Used to enable interrupts 32 to 63. |
| ⋮ | ⋮ | ⋮ | ⋮ |
| EN4 | 0xE000E110 | 0x00000000 | Used to enable interrupts 128 to 138. |
| | | Interrupt disable registers | |
| DIS0 | 0xE000E180 | 0x00000000 | Used to disable interrupts 0 to 31. |
| DIS1 | 0xE000E184 | 0x00000000 | Used to disable interrupts 32 to 63. |
| ⋮ | ⋮ | ⋮ | ⋮ |
| DIS4 | 0xE000E190 | 0x00000000 | Used to disable interrupts 128 to 138. |
| | | Priority configuration registers | |
| PRI0 | 0xE000E400 | 0x00000000 | Priority for interrupts 0 to 3. |
| PRI1 | 0xE000E404 | 0x00000000 | Priority for interrupts 4 to 7. |
| ⋮ | ⋮ | ⋮ | ⋮ |
| PRI34 | 0xE000E488 | 0x00000000 | Priority for interrupts 136 to 138. |

**Interrupt Sense** ***(IS)*** register is used to configure interrupt triggering as edge or level. When a bit in this register is cleared to 0 the corresponding GPIO port pin is configured as edge triggered interrupt and writing 1 corresponds to the level triggered. It has offset value of 0x404

**Interrupt Event (IEV**) register configures the type of the interrupt event. Setting to 1 configures the port pin for rising edge interrupt (when edge type interrupt is configured in IS register) or a logic level high triggers the interrupt when IS register is configured for level sensitive interrupt. Clearing it to 0 configures either the falling edge or logic low as the interrupt condition depending on the IS register. It has offset value of 0x40c

**Interrupt Both Edges (IBE),** when set to 1, configures both rising as well as falling edges as the source of interrupt. For this to be valid, IS register should be configured for edge type interrupt. Configuring IBE register with a value of 1 overrides the IEV register configuration. It has offset value of 0x408

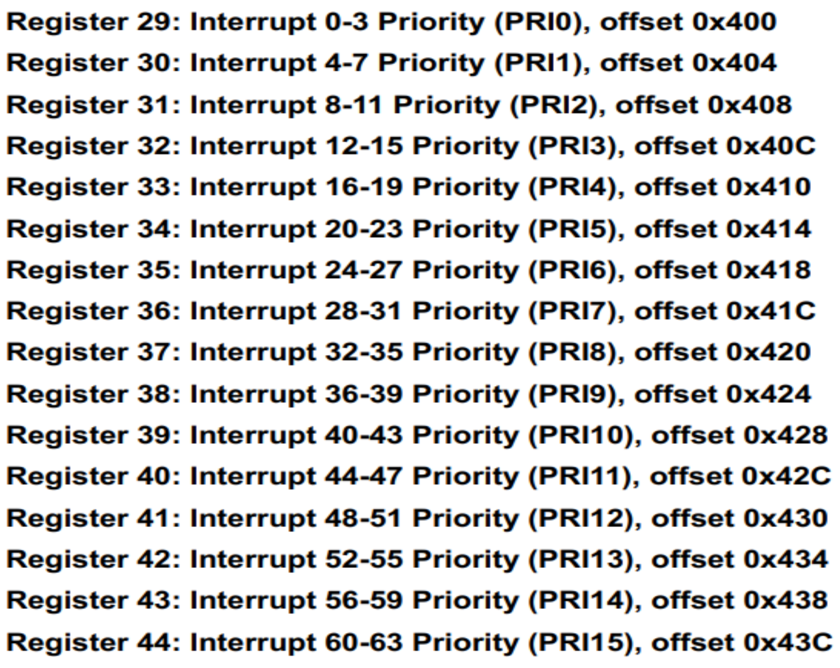
**Interrupt Mask (IM)** register is used to enable or disable individual port pin interrupts. Writing 1 to this register enables the corresponding port pin interrupt. It has offset value of 0x410

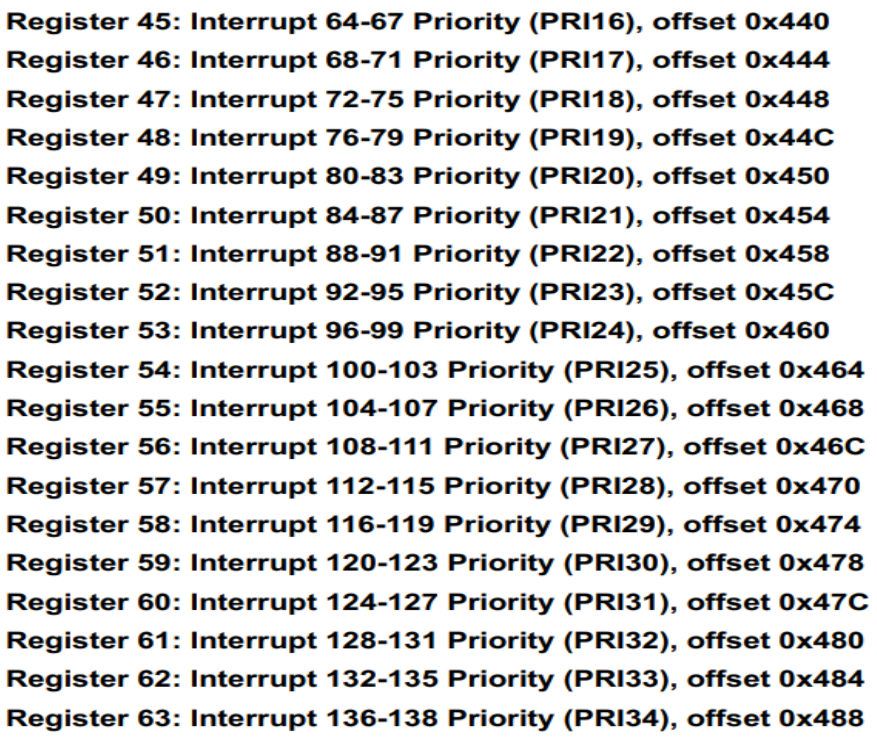
**Raw Interrupt Status (RIS) and Masked Interrupt Status (MIS)** are the two registers, which indicate the occurrence of an interrupt condition. The RIS register indicates that a GPIO pin satisfies the requirements for an interrupt, but whether the interrupt is sent to NVIC or not depends on the IM register. On the other hand, the MIS register only indicates those GPIO pin interrupts that are enabled and sent to NVIC. They have offset value of 0x414 and 0x418 respectively.

**Interrupt Clear (ICR)** register is used to clear an interrupt flag in the status registers. To clear an interrupt status flag in RIS as well as MIS registers, it is required to write '1' to the corresponding bit in the ICR register. If an interrupt is not cleared using ICR before exiting the corresponding interrupt service routine, then the ISR is immediately entered again, provided no other interrupt of higher priority has occurred in the meantime. It has offset value of 0x41c.

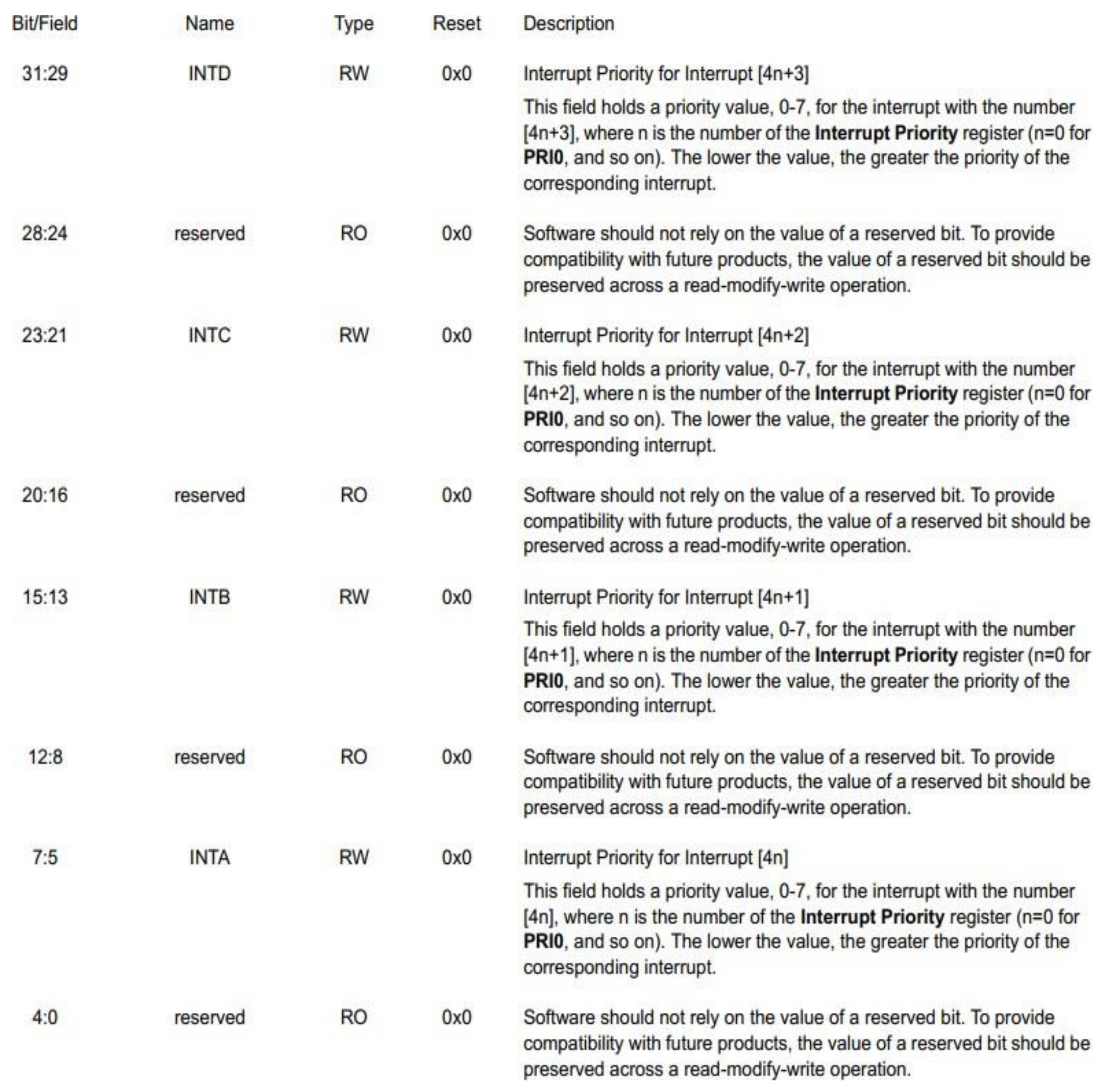
## Priority Registers.

Register Given Below are used to set the priority for the specific interrupt.


**Register 29: Interrupt 0-3 Priority (PRI0), offset 0x400**
**Register 30: Interrupt 4-7 Priority (PRI1), offset 0x404**
**Register 31: Interrupt 8-11 Priority (PRI2), offset 0x408**
**Register 32: Interrupt 12-15 Priority (PRI3), offset 0x40C**
**Register 33: Interrupt 16-19 Priority (PRI4), offset 0x410**
**Register 34: Interrupt 20-23 Priority (PRI5), offset 0x414**
**Register 35: Interrupt 24-27 Priority (PRI6), offset 0x418**
**Register 36: Interrupt 28-31 Priority (PRI7), offset 0x41C**
**Register 37: Interrupt 32-35 Priority (PRI8), offset 0x420**
**Register 38: Interrupt 36-39 Priority (PRI9), offset 0x424**
**Register 39: Interrupt 40-43 Priority (PRI10), offset 0x428**
**Register 40: Interrupt 44-47 Priority (PRI11), offset 0x42C**
**Register 41: Interrupt 48-51 Priority (PRI12), offset 0x430**
**Register 42: Interrupt 52-55 Priority (PRI13), offset 0x434**
**Register 43: Interrupt 56-59 Priority (PRI14), offset 0x438**
**Register 44: Interrupt 60-63 Priority (PRI15), offset 0x43C**


**Register 45: Interrupt 64-67 Priority (PRI16), offset 0x440**
**Register 46: Interrupt 68-71 Priority (PRI17), offset 0x444**
**Register 47: Interrupt 72-75 Priority (PRI18), offset 0x448**
**Register 48: Interrupt 76-79 Priority (PRI19), offset 0x44C**
**Register 49: Interrupt 80-83 Priority (PRI20), offset 0x450**
**Register 50: Interrupt 84-87 Priority (PRI21), offset 0x454**
**Register 51: Interrupt 88-91 Priority (PRI22), offset 0x458**
**Register 52: Interrupt 92-95 Priority (PRI23), offset 0x45C**
**Register 53: Interrupt 96-99 Priority (PRI24), offset 0x460**
**Register 54: Interrupt 100-103 Priority (PRI25), offset 0x464**
**Register 55: Interrupt 104-107 Priority (PRI26), offset 0x468**
**Register 56: Interrupt 108-111 Priority (PRI27), offset 0x46C**
**Register 57: Interrupt 112-115 Priority (PRI28), offset 0x470**
**Register 58: Interrupt 116-119 Priority (PRI29), offset 0x474**
**Register 59: Interrupt 120-123 Priority (PRI30), offset 0x478**
**Register 60: Interrupt 124-127 Priority (PRI31), offset 0x47C**
**Register 61: Interrupt 128-131 Priority (PRI32), offset 0x480**
**Register 62: Interrupt 132-135 Priority (PRI33), offset 0x484**
**Register 63: Interrupt 136-138 Priority (PRI34), offset 0x488**

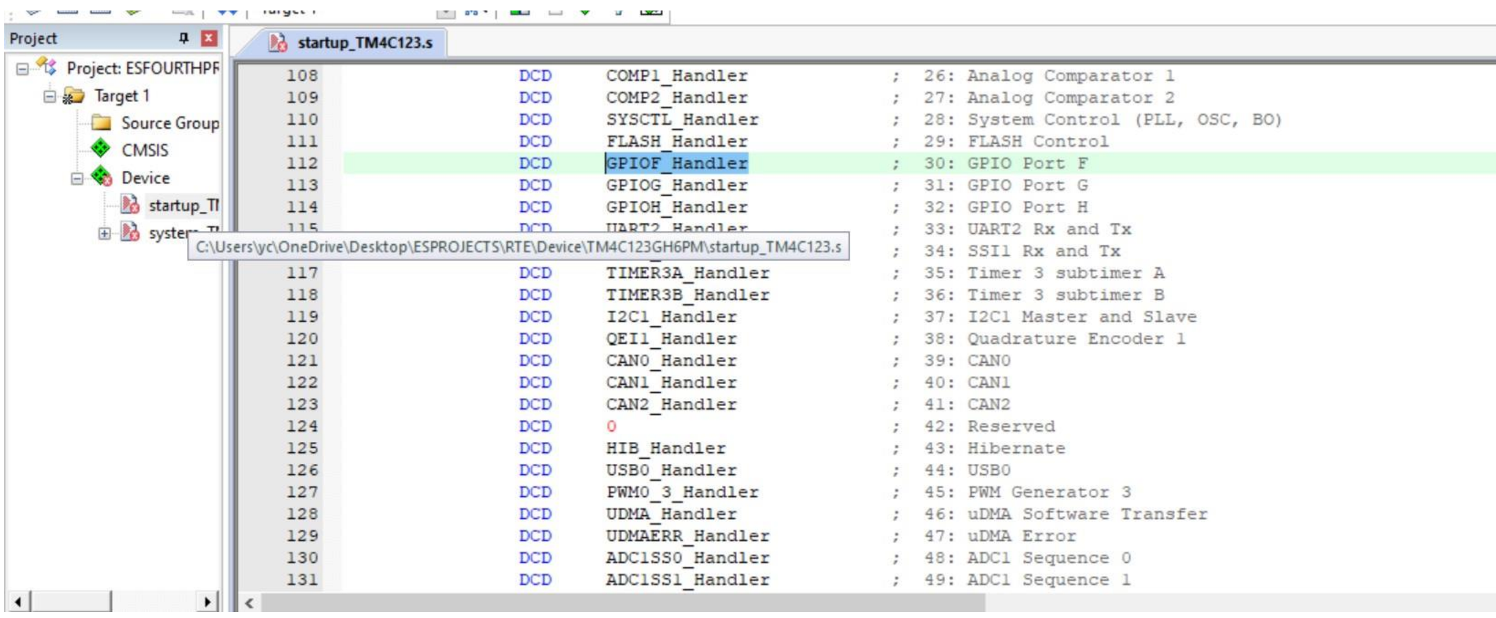
Each Priority Regiters have cpability for Four interrupts using Three bits for Each interrupt.The way to use these three bits is as follows.



| Bit/Field | Name | Type | Reset | Description |
|---|---|---|---|---|
| 31:29 | INTD | RW | 0x0 | Interrupt Priority for Interrupt [4n+3]<br>This field holds a priority value, 0-7, for the interrupt with the number [4n+3], where n is the number of the **Interrupt Priority** register (n=0 for **PRI0**, and so on). The lower the value, the greater the priority of the corresponding interrupt. |
| 28:24 | reserved | RO | 0x0 | 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. |
| 23:21 | INTC | RW | 0x0 | Interrupt Priority for Interrupt [4n+2]<br>This field holds a priority value, 0-7, for the interrupt with the number [4n+2], where n is the number of the **Interrupt Priority** register (n=0 for **PRI0**, and so on). The lower the value, the greater the priority of the corresponding interrupt. |
| 20:16 | reserved | RO | 0x0 | 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. |
| 15:13 | INTB | RW | 0x0 | Interrupt Priority for Interrupt [4n+1]<br>This field holds a priority value, 0-7, for the interrupt with the number [4n+1], where n is the number of the **Interrupt Priority** register (n=0 for **PRI0**, and so on). The lower the value, the greater the priority of the corresponding interrupt. |
| 12:8 | reserved | RO | 0x0 | 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. |
| 7:5 | INTA | RW | 0x0 | Interrupt Priority for Interrupt [4n]<br>This field holds a priority value, 0-7, for the interrupt with the number [4n], where n is the number of the **Interrupt Priority** register (n=0 for **PRI0**, and so on). The lower the value, the greater the priority of the corresponding interrupt. |
| 4:0 | reserved | RO | 0x0 | 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. |

## Handler Function:

Name of the handler Function Should Be Similar to the name defined for that specific Interrupt in Startup File. Startup file can be Obtained from Opened Kiel project by clicking on the device then on Startup file tab. As shown below.



C:\Users\yc\OneDrive\Desktop\ESPROJECTS\RTE\Device\TM4C123GH6PM\startup_TM4C123.s

```
                DCD     COMP1_Handler             ;  26: Analog Comparator 1
                DCD     COMP2_Handler             ;  27: Analog Comparator 2
                DCD     SYSCTL_Handler            ;  28: System Control (PLL, OSC, BO)
                DCD     FLASH_Handler             ;  29: FLASH Control
                DCD     GPIOF_Handler             ;  30: GPIO Port F
                DCD     GPIOG_Handler             ;  31: GPIO Port G
                DCD     GPIOH_Handler             ;  32: GPIO Port H
                DCD     UART2_Handler             ;  33: UART2 Rx and Tx
                                                  ;  34: SSI1 Rx and Tx
                DCD     TIMER3A_Handler           ;  35: Timer 3 subtimer A
                DCD     TIMER3B_Handler           ;  36: Timer 3 subtimer B
                DCD     I2C1_Handler              ;  37: I2C1 Master and Slave
                DCD     QEI1_Handler              ;  38: Quadrature Encoder 1
                DCD     CAN0_Handler              ;  39: CAN0
                DCD     CAN1_Handler              ;  40: CAN1
                DCD     CAN2_Handler              ;  41: CAN2
                DCD     0                         ;  42: Reserved
                DCD     HIB_Handler               ;  43: Hibernate
                DCD     USB0_Handler              ;  44: USB0
                DCD     PWM0_3_Handler            ;  45: PWM Generator 3
                DCD     UDMA_Handler              ;  46: uDMA Software Transfer
                DCD     UDMAERR_Handler           ;  47: uDMA Error
                DCD     ADC1SS0_Handler           ;  48: ADC1 Sequence 0
                DCD     ADC1SS1_Handler           ;  49: ADC1 Sequence 1
```

**Interrupt Program.**

**This C program interfaces the on-board user switch SW1 connected to PF4 by configuring PF4 as an external interrupt. The interrupt is generated on the falling edge, corresponding to the pressing of the switch. On every key press an on-board LED (green color) connected to PF3 is toggled.**

| **Example 1: Toggle On-Board Green LED (Assembly Program)** |
|---|
| //clock, Data, Direction, Digital Enable, Pull Up resistor _Register Enabling for the portf<br><br># define SYSCTL_RCGCGPIO_R (*((volatile unsigned long *)0x400fe608))<br><br># define GPIO_PORTF_DATA_RD (*((volatile unsigned long *)0x40025040))<br><br># define GPIO_PORTF_DATA_WR (*((volatile unsigned long *)0x40025020))<br><br># define GPIO_PORTF_DIR_R (*((volatile unsigned long *)0x40025400))<br><br># define GPIO_PORTF_DEN_R (*((volatile unsigned long *)0x4002551c))<br><br># define GPIO_PORTF_PUR_R (*((volatile unsigned long *)0x40025510)) |

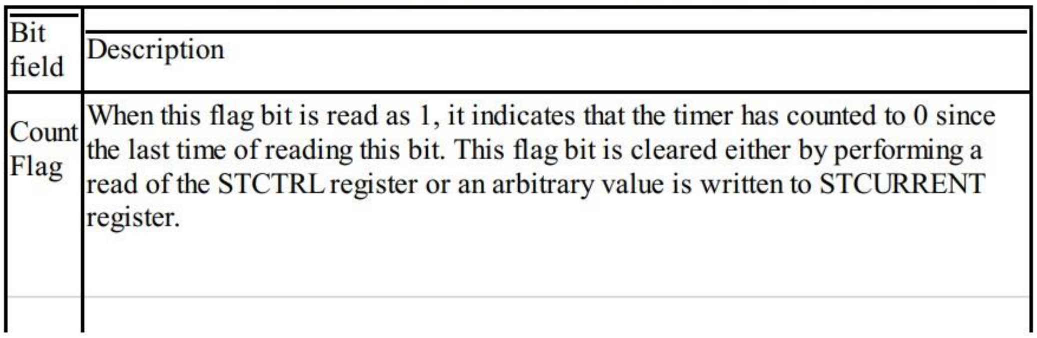
```
// IRQ 0 to 31 Set Enable Register

# define NVIC_EN0_R *(( volatile unsigned long *)0xe000e100 )

# define NVIC_EN0_INT30 0x40000000 // Interrupt 30 enable.

//Interrupt Resiters Configuration For portf.

# define NVIC_PRI7_R *(( volatile unsigned long *)0xe000e41c ) // IRQ 28 to 31 Priority Register
NVIC_PRI7_R.

# define GPIO_PORTF_IS_R *(( volatile unsigned long *)0x40025404 ) // interrupt service routine register
enabling.

# define GPIO_PORTF_IBE_R *(( volatile unsigned long *)0x40025408 ) //enabling both edge sensitive, (not
compulsory.)

# define GPIO_PORTF_IEV_R *(( volatile unsigned long *)0x4002540c ) // event register enabling to make
sure the occurrence of interrupt.

# define GPIO_PORTF_IM_R *(( volatile unsigned long *)0x40025410 ) // interrupt mask register.

# define GPIO_PORTF_ICR_R *(( volatile unsigned long *)0x4002541c )

#define GPIO_PORTF_MIS_R  *(( volatile unsigned long *)0x40025418 )

// portf pin definitions.

# define SYSCTL_RCGC2_GPIOF 0x0020

# define GPIO_PORTF_PIN3_EN 0x08

# define GPIO_PORTF_PIN4_EN 0x10


// Default clock frequency and delay definition

# define SYSTEM_CLOC_FREQUENCY  16000000

#define DELAY_DEBOUNCE                    SYSTEM_CLOC_FREQUENCY/10000

Void Delay(unsigned long counter) // delay function body.

{

    Unsigned long i = 0;

    For(i = 0; i<counter; i++);

}
```

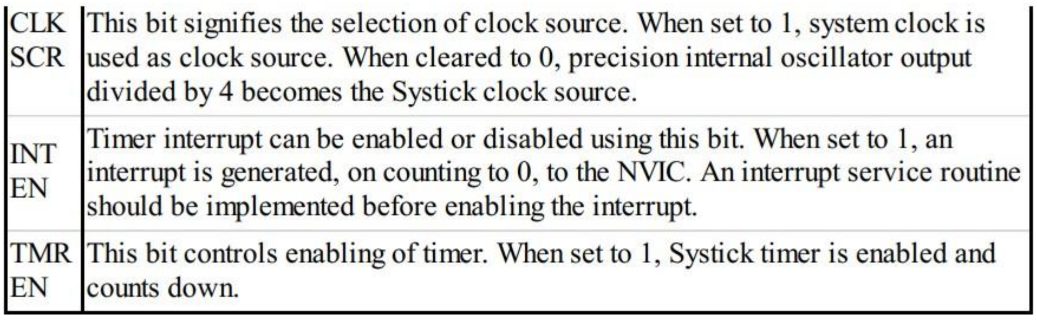
```
// main User Application Program

Int main ()

{

// Enable the clock for port F

SYSCTL_RCGCGPIO_R |= SYSCTL_RCGC2_GPIOF;

GPIO_PORTF_DEN_R|= GPIO_PORTF_PIN3_EN+ GPIO_PORTF_PIN4_EN ; //Digitally Enabling both the
Pins Of portf.

GPIO_PORTF_DIR_R |=GPIO_PORTF_PIN3_EN; // PF3 as output.

GPIO_PORTF_DIR_R &= (~GPIO_PORTF_PIN4_EN); // PF4 as input.

GPIO_PORTF_PUR_R |= GPIO_PORTF_PIN4_EN ; // Enable pulp resistor for portfp4.

    //configure PORTF4, for falling edge trigger interrupt

GPIO_PORTF_IS_R &= (~GPIO_PORTF_PIN4_EN); // PF4 is edge triggered

GPIO_PORTF_IBE_R &= (~GPIO_PORTF_PIN4_EN); // PF4 is not both edge sensitive

GPIO_PORTF_IEV_R &= (~GPIO_PORTF_PIN4_EN); // PF4 falling edge sensitive

GPIO_PORTF_ICR_R |= GPIO_PORTF_PIN4_EN ; // Clearing interrupt Flag in status Register

GPIO_PORTF_IM_R |= GPIO_PORTF_PIN4_EN ; // Masking the interrupt for portf4

NVIC_PRI7_R |= 0x00300000 ; // Priority is 3 (0000 0000 0011 0000 0000 0000 0000 0000)

NVIC_EN0_R |= NVIC_EN0_INT30; // Enable INT 30 in NVIC

While (1) // wait to occur interrupt.

{

    // Do Nothing.

}

}

 // Interrupt service routine for Port F

Void GPIOF_Handler(void) // definig interrupt function. Name from the file as shown above.

{

// Insert delay for switch debouncing.
Delay (DELAY_DEBOUNCE);

If(GPIO_PORTF_MIS_R &= 0x10) // condition to check interrupt caused by portf4.

{

GPIO_PORTF_DATA_WR ^= GPIO_PORTF_PIN3_EN; // turning onboard led on.

GPIO_PORTF_ICR_R = 0x10 ; // acknowledge flag4.

}

}// End Of The Program.
```

## Systick Interrupt Introduction.

Systick timer has three configuration and control registers. But STCTRL (systick control and status register is the main register which is used to configure the timer. It is a 32-bit register, but only four bits are used such as ENABLE, INTEN, CLK_SCR and COUNT bits.

Bit field descriptions for Systick control and status (STCTRL) register.

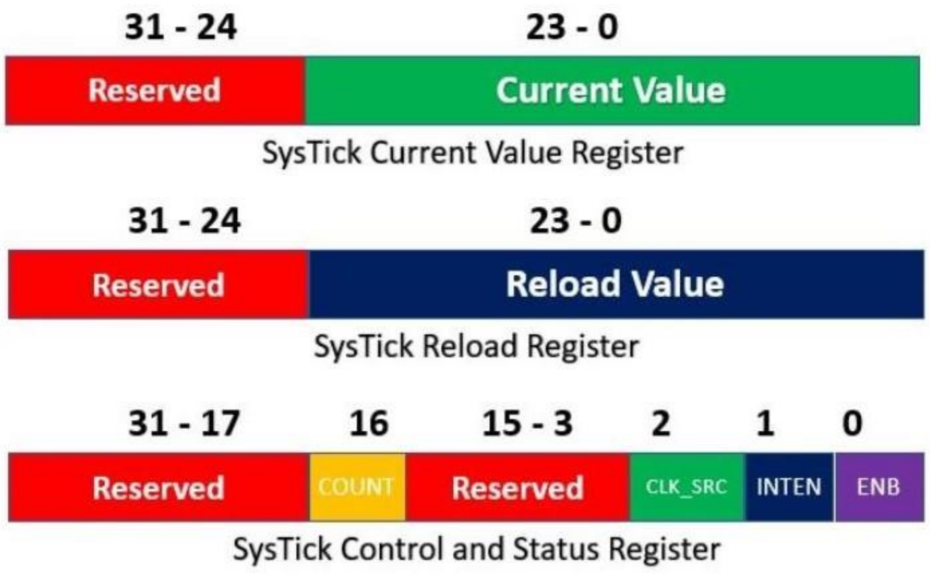




| Bit field | Description |
|---|---|
| Count Flag | When this flag bit is read as 1, it indicates that the timer has counted to 0 since the last time of reading this bit. This flag bit is cleared either by performing a read of the STCTRL register or an arbitrary value is written to STCURRENT register. |
| | |
| CLK SCR | This bit signifies the selection of clock source. When set to 1, system clock is used as clock source. When cleared to 0, precision internal oscillator output divided by 4 becomes the Systick clock source. |
| INT EN | Timer interrupt can be enabled or disabled using this bit. When set to 1, an interrupt is generated, on counting to 0, to the NVIC. An interrupt service routine should be implemented before enabling the interrupt. |
| TMR EN | This bit controls enabling of timer. When set to 1, Systick timer is enabled and counts down. |

INTEN bit enables or disables the interrupt functionality of systick timer. If we set the INTEN bit (INTEN=1), it enables the systick interrupt. That means, whenever the counter reaches zero from a reload value, the systick timer requests the interrupt signal to the nested interrupt vector controller (NVIC) of TM4C123 microcontroller.

For example, if we load the number 100 to reload the register and the clock frequency of TM4C123 microcontroller is 16mhz. That means counter value will decrement by 1 after every 1/16mhz or 62.5 nanoseconds. Therefore, the counter value will reach to zero after 101 x 62.5ns



To generate interrupt after every second, the value of reload register will be.

**Reload = (require delay / clock time period) - 1**

**Reload = (1/16mhz) - 1 = 16000000 - 1 = 1599999**

**C Language Program**

**Example 2: Toggle On-Board Green LED BY generating interrupt from Systic.**

```
//Register Configuration for portf.

# define SYSCTL_RCGCGPIO_R (*(( volatile unsigned long *)0x400fe608))

# define GPIO_PORTF_DATA_WR (*(( volatile unsigned long *)0x40025020))

# define GPIO_PORTF_DIR_R (*(( volatile unsigned long *)0x40025400))

# define GPIO_PORTF_DEN_R (*(( volatile unsigned long *)0x4002551c))

//systic Register Definition.

#define ST_CTRL_R (*((volatile unsigned long*)0xe000e010)) //Systick control and status register

#define ST_RELOAD_R (*((volatile unsigned long*)0xe000e014)) // Systick reload value register

#define ST_CURRENT_R (*((volatile unsigned long *)0xe000e018)) // Systick reload Current register
```

```
Int main()

{

 SYSCTL_RCGCGPIO_R |= 0x20; // turn on bus clock for GPIOF

  GPIO_PORTF_DIR_R |= 0x08; //set GREEN pin as a digital output pin

  GPIO_PORTF_DEN_R  |= 0x08; // Enable PF3 pin as a digital pin

            ST_RELOAD_R = 15999999; // one second delay relaod value.

            ST_CTRL_R  = 7 ; // enable counter, interrupt and select system bus clock

            ST_CURRENT_R = 0; //initialize current value register

            While (1)

            {

                        //do nothing here

            }

}

// this routine will execute after every one second

Void systick_Handler(void)

{

   GPIO_PORTF_DATA_WR  ^= 8; //toggle PF3 pin


}
```

## TASKS.

### Task_1.

**Repeat the Example 1 and Make addition of 7_Segment Display to Count and to visualize the occurrence of Interrupts.**

### Task_2.

**Write a Program that 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) LED will turn on and if the interrupt is caused by SW2(PF0) LED will turn off and Use 7_segment Display to count No of interrupts and to make sure weather interrupt takes place or not.**

### Task_3.

**Write a Program that controls the Green and Red 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) Green LED will turn on and if the interrupt is caused by SW2(PF0) Red LED Will turn On.**