Introduction to Microcontrollers

中山 大 学

数据科学与计算机学院

郭雪梅

Tel:39943108

Email:guoxuem@mail.sysu.edu.cn

URL1：http://human-robot.sysu.edu.cn/course

Introduction to Embedded Systems

Lecture 6: SysTick Timer, interrupt

Review

Agenda

- Recap
  - Subroutines and Parameter Passing
    - AAPCS Convention
  - Indexed Addressing and Pointers
    - In C: Address of (&), Pointer to (*)
  - Data Structures: Arrays, Strings
    - Length: hardcoded vs. embedded vs. sentinel
    - Array access: indexed vs. pointer arithmetic
  - Functional Debugging
- Outline
  - Review
  - SysTick Timer

Review

❿ Definitions (define in 16 words or less, choose the word, or multiple choice)

- volatile, nonvolatile, RAM, ROM, port
- structured program, call graph, data flow graph
- basis, nibble, precision, kibibyte, mebibyte
- signed/unsigned, 8-bit, 16-bit, 32-bit
- overflow, ceiling and floor, drop out
- bus, address bus, data bus
- memory-mapped, I/O mapped
- Harvard architecture, von Neumann
- ALU, D flip-flop, registers
- device driver, CISC, RISC
- friendly, mask, toggle, heartbeat, breakpoint
- Negative logic, positive logic, open collector
- Voltage, current, power, Ohm's Law

Review

- Number conversions - *convert one format to another*
  - alternatives, binary bits
  - signed decimal e.g., -56
  - unsigned decimal e.g., 200
  - binary e.g., 11001000_2
  - hexadecimal e.g., 0xC8
- Addressing modes *(book Sec 3.3.2)*
  - Immediate e.g., MOV R0,#0,
  - Indexed e.g., LDR R0,[R1] LDR R0,=123
  - PC-relative e.g., BL subroutine
  - Register list, e.g., PUSH {R1, R4-R6}

Review

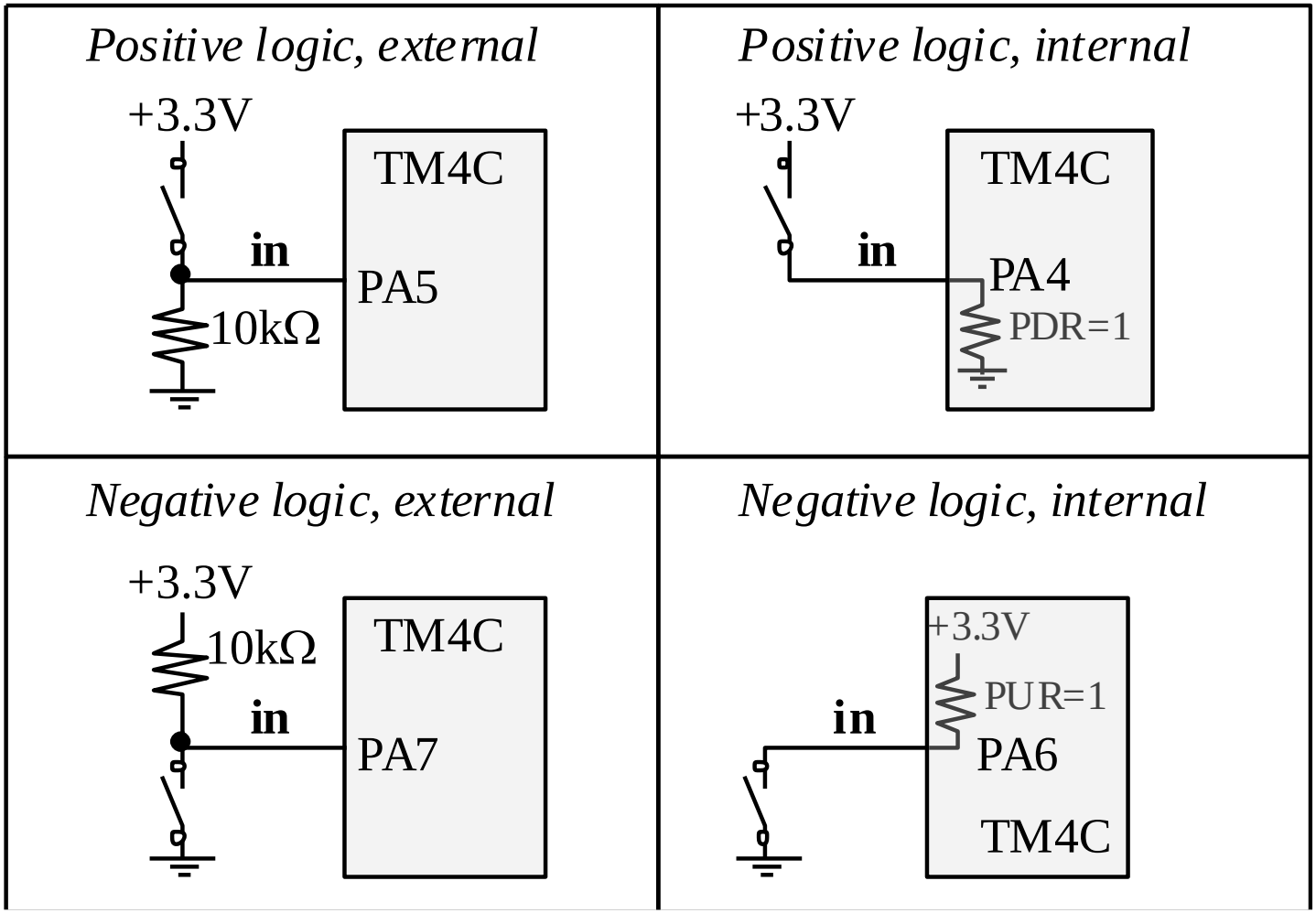
- Cortex-M4 operation & instructions
  - Definition of N,Z,V,C
    - What do they mean? How do we use them?
  - Thumb-2 instructions on reference sheet
  - Components in address space
  - Subroutine linkage
  - Stack operations
- Switch and LED interfaces
- C Programming
  - Declarations, expressions, control flow
    - Global variables, Conditionals and Loops

Review

- Simple programs (assembly and C)
  - create global variables
  - specify an I/O pin is an input
  - specify an I/O pin is an output
  - clear an I/O output pin to zero
  - set an I/O output pin to one
  - toggle an I/O output pin
  - check if an I/O input pin is high or low
  - add, sub, shift left, shift right, and, or, eor
  - subroutine linkage
- Use of the stack
  - Stack instructions and stack diagrams

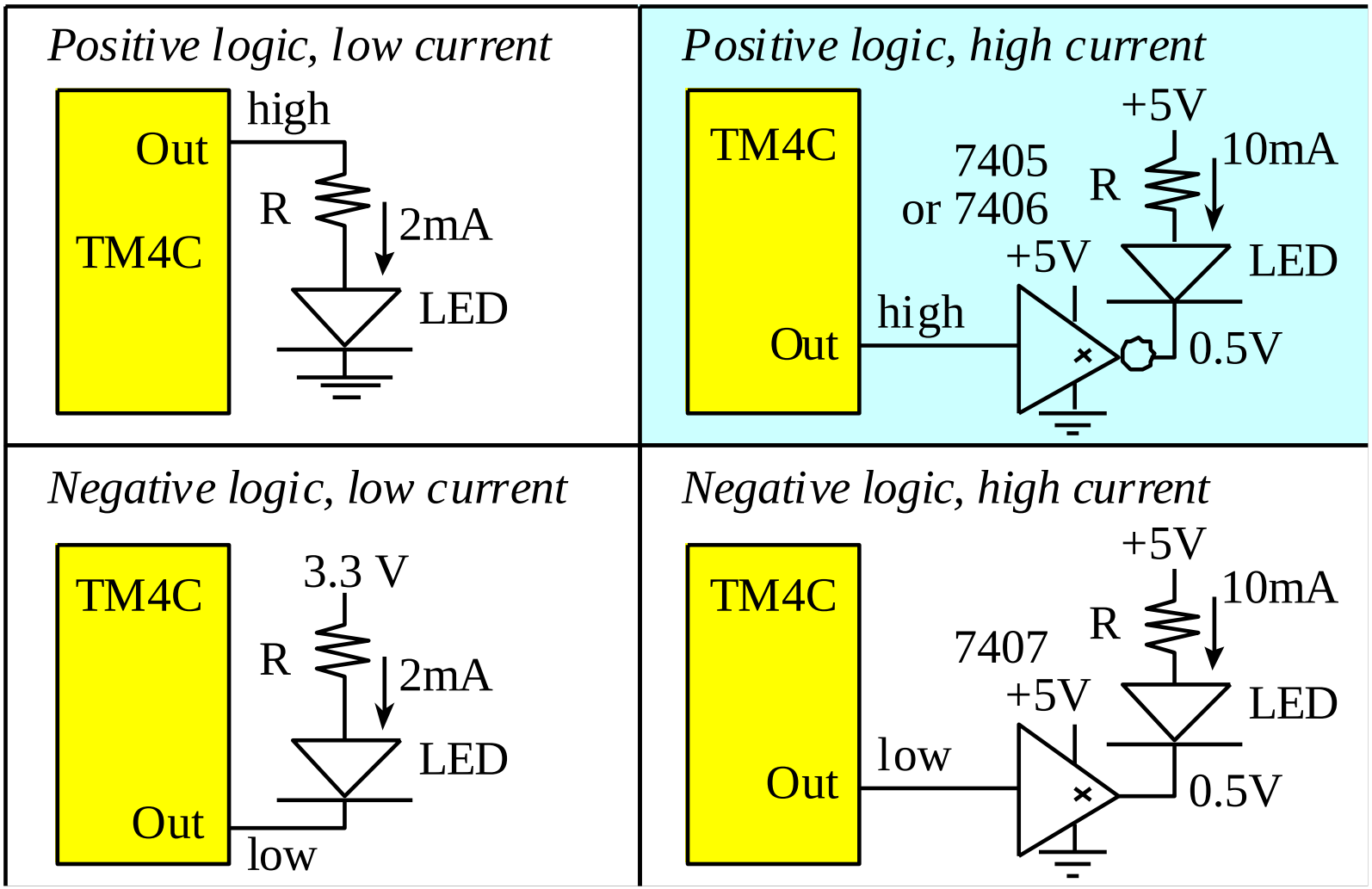
- Not on Exam
  - Pointers in C
  - Arrays, strings
  - Call by reference
  - Cycle by cycle execution

Switch Interface



Know voltage, current, power

LED interfaces

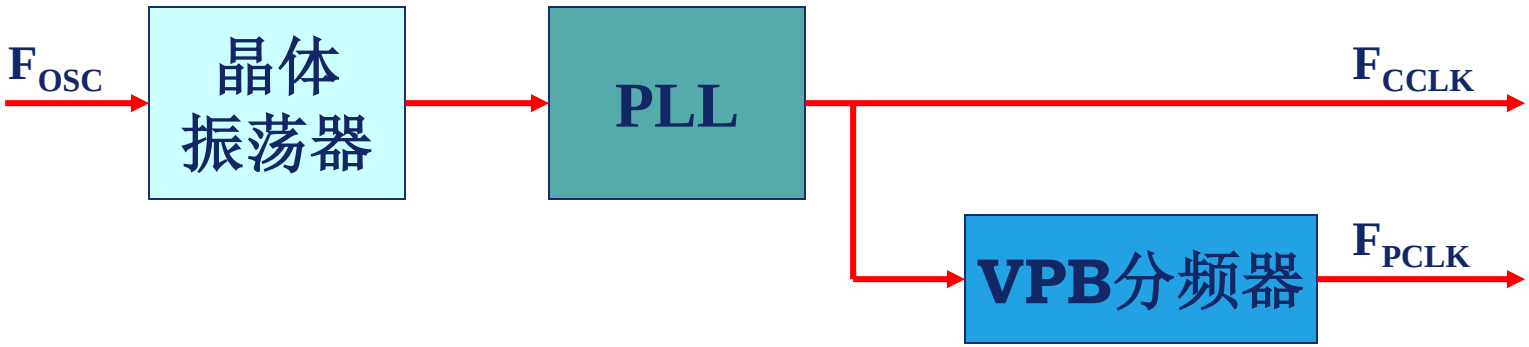


Know voltage, current, power, Ohm's Law

系统时钟

⑩ 时钟产生单元

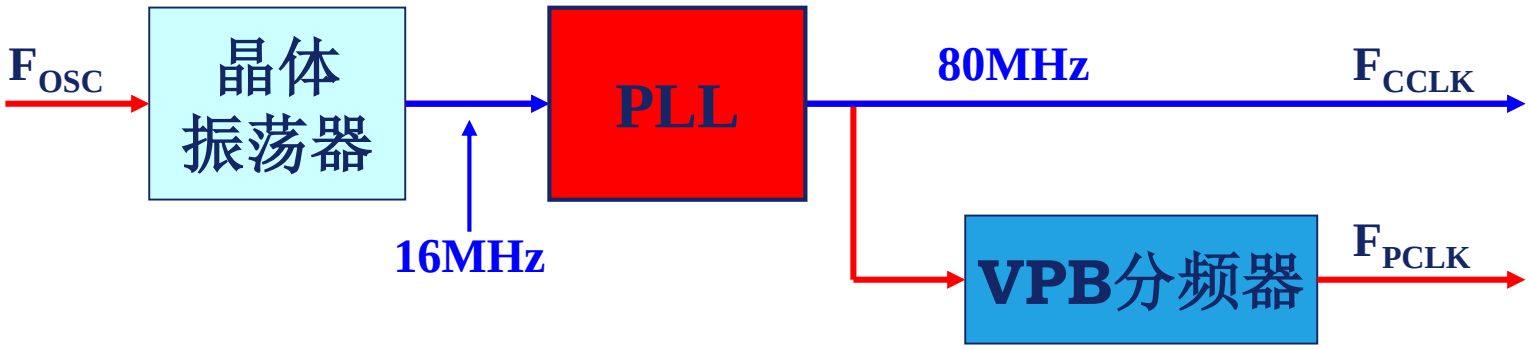
时钟产生单元包括晶体振荡器、锁相环振荡器（PLL）和VPB分频器。



系统时钟

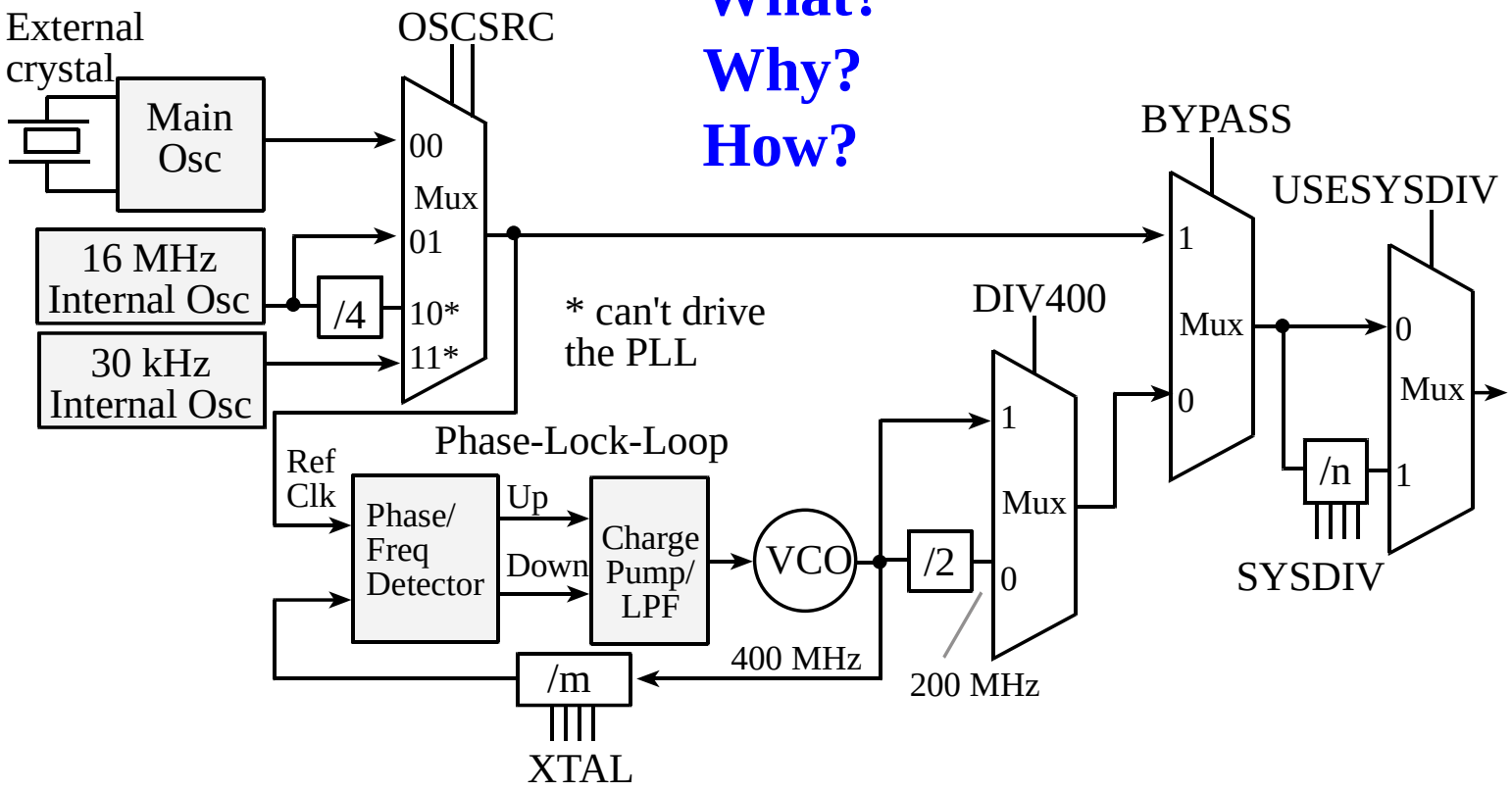
⑩ 锁相环（PLL）

由晶体振荡器输出的时钟信号，通过**PLL**升频，可以获得更高的系统时钟（**CCLK**）。



Phase-Lock-Loop

What?
Why?
How?



- Internal oscillator requires minimal power but is imprecise
- External crystal provides stable bus clock
- TM4C123 is equipped with 16 MHz crystal and bus clock can be set to a maximum of 80 MHz

Most popular parameters used to configure a device include:

1. The input crystal frequency
2. The oscillator to be used
3. Whether use of the PLL
4. The system clock divider

XTAL	Crystal Freq (MHz)		XTAL	Crystal Freq (MHz)
0x0	Reserved		0x10	10.0 MHz
0x1	Reserved		0x11	12.0 MHz
0x4	3.579545 MHz		0x14	14.31818 MHz
0x5	3.6864 MHz		0x15	16.0 MHz

To clock the system from the PLL, use SYSCTL_USE_PLL | SYSCTL_OSC_MAIN. For example, to configure the GPIO PORTF by using a 16-MHz crystal on the main oscillator with using the PLL whose output is 400 MHz. With a divisor in the clock path as above figure. To do this configuration as The following :

```
SYSCTL_RCC_R += SYSCTL_RCC_XTAL_16MHZ;// configure for 16 MHz crystal
SYSCTL_RCC2_R = (SYSCTL_RCC2_R&~0x1FC00000) // clear system clock divider field
                + (SYSDIV2<<22);      // configure for 80 MHz clock, SYSDIV2=4
SYSCTL_RCC2_R += (SYSDIV<<23)|(LSB<<22);  // divide by (2*SYSDIV+1+LSB)
#define SYSDIV 3    #define LSB 1
// bus frequency is 400MHz/(2*SYSDIV+1+LSB) = 400MHz/(2*3+1+1) = 50 MHz
```

TM4C123 16 MHz crystal and bus clock 80 MHz Configure

In file “pll.h”
#define SYSDIV2 4
void PLL_Init(void);
bus frequency is 400MHz/(SYSDIV2+1)
= 400MHz/(4+1) = 80 MHz
Example SYSDIV (n) values:
n=4 gives 400/(4+1) = 80 MHz
n=7 gives 400/(7+1) = 50MHz
n=9 gives 400/(9+1) = 40MHz
n=15 gives 400/(15+1) = 25MHz

```
#define SYSDIV2 4
void PLL_Init(void){
// 0) Use RCC2
SYSCTL_RCC2_R |= 0x80000000; // USERCC2
// 1) bypass PLL while initializing
SYSCTL_RCC2_R |= 0x00000800; // BYPASS2, PLL bypass
// 2) select the crystal value and oscillator source
SYSCTL_RCC_R = (SYSCTL_RCC_R & ~0x000007C0) // clear bits 10-6
+ 0x00000540; // 10101, configure for 16 MHz crystal
SYSCTL_RCC2_R &= ~0x00000070; // configure for main oscillator source
// 3) activate PLL by clearing PWRDN
SYSCTL_RCC2_R &= ~0x00002000;
// 4) set the desired system divider
SYSCTL_RCC2_R |= 0x40000000; // use 400 MHz PLL
SYSCTL_RCC2_R =
(SYSCTL_RCC2_R&~0x1FC00000)+(SYSDIV2<<22); // 80 MHz
// 5) wait for the PLL to lock by polling PLLLRIS
while((SYSCTL_RIS_R&0x00000040)==0){}; // wait for PLLRIS bit
// 6) enable use of PLL by clearing BYPASS
SYSCTL_RCC2_R &= ~0x00000800;
}
```

Program 4.6a. Activate the TM4C123 with a 16 MHz crystal to run at 80 MHz (PLL_xxx.zip).

SysTick Timer(系统定时器)

Cortex-M3集成了一个系统定时器，SysTick。 SysTick给一个简单的24位写清零、递减、计数到零时重装的计数器提供灵活的控制机制。 该计数器有几种使用方法，例如：

- 用作一个RTOS时钟节拍定时器，以编程设定的速率（例如，100Hz）启动，调用一个SysTick程序。
- 用作一个使用系统时钟的高速报警定时器。
- 用作一个速率可变的报警或信号定时器——它的工作时间取决于使用的参考时钟和计数器的动态范围。
- 用作一个简单的计数器。 软件可以使用这个定时器来测量完成操作的时间或操作用掉的时间。
- 一个根据未到达/到达的时间（missing/meeting durations）来控制的内部时钟。 作为动态时钟管理控制循环的一部分，控制和状态寄存器中的COUNTFLAG位域可以用来决定某项操作是否在设定的时间内完成。

SysTick Timer(系统定时器)

SysTickis a simple counter that we can use to create time delays and generate periodic interrupts. Table 4.10 shows some of the register definitions for SysTick.

The basis of SysTick is a 24-bit down counter that runs at the bus clock frequency. There are four steps to initialize the SysTick timer. First, we clear the **ENABLE** bit to turn off SysTick during initialization. Second, we set the **RELOAD**register(计数器的重装值). Third, we write to the **NVIC_ST_CURRENT_R(**计数器的当前值**)** value to clear the counter. Lastly, we write the desired mode to the control register, **NVIC_ST_CTRL_R**(控制和状态计数器用来配置其时钟、使能计数器、使能SysTick中断以及确定计数器状态)

Address	31-24	23-17	16	15-3	2	1	0	Name
$E000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	NVIC_ST_CTRL_R
$E000E014	0	24-bit RELOAD value						NVIC_ST_RELOAD_R
$E000E018	0	24-bit CURRENT value of SysTick counter						NVIC_ST_CURRENT_R

SysTick Timer

- **Timer/Counter operation**
  - **24-bit counter *decrements* at bus clock frequency**
    - With 80 MHz bus clock, decrements every 12.5 ns
  - **Counting is from *n* → 0**
    - Setting *n* appropriately will make the counter a modulo *n*+1 counter. That is:
      - **next_value = (current_value-1) mod (n+1)**
      - **Sequence: n,n-1,n-2,n-3… 2,1,0,n,n-1…**

SysTick Timer

Address	31-24	23-17	16	15-3	2	1	0	Name
$E000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	NVIC_ST_CTRL_R
$E000E014	0	24-bit RELOAD value						NVIC_ST_RELOAD_R
$E000E018	0	24-bit CURRENT value of SysTick counter						NVIC_ST_CURRENT_R

- **Initialization (4 steps)**
  - **Step1: Clear ENABLE to stop counter**
  - **Step2: Specify the RELOAD value**
  - **Step3: Clear the counter via NVIC_ST_CURRENT_R**
  - **Step4: Set NVIC_ST_CTRL_R**
    - **CLK_SRC = 1 (bus clock is the only option)**
    - **INTEN = 0 for no interrupts**
    - **ENABLE = 1 to enable**

When the **CURRENT** value counts down from 1 to 0, the **COUNT** flag is set.

SysTick Timer

24-bit Countdown Timer

```
SysTick_Init
; disable SysTick during setup
    LDR R1, =NVIC_ST_CTRL_R
    MOV R0, #0                  ; Clear Enable
    STR R0, [R1]
; set reload to maximum reload value
    LDR R1, =NVIC_ST_RELOAD_R
    LDR R0, =0x00FFFFFF;        ; Specify RELOAD value
    STR R0, [R1]                ; reload at maximum
; writing any value to CURRENT clears it
    LDR R1, =NVIC_ST_CURRENT_R
    MOV R0, #0
    STR R0, [R1]                ; clear counter
; enable SysTick with core clock
    LDR R1, =NVIC_ST_CTRL_R
    MOV R0, #0x0005     ; Enable but no interrupts (later)
    STR R0, [R1]        ; ENABLE and CLK_SRC bits set
    BX  LR
```

SysTick Timer

```
;------------SysTick_Wait------------
; Time delay using busy wait.
; Input: R0  delay parameter in units of the core clock
;        80 MHz(12.5 nsec each tick)
; Output: none
; Modifies: R1
SysTick_Wait
    SUB  R0, R0, #1    ; delay-1
    LDR  R1, =NVIC_ST_RELOAD_R
    STR  R0, [R1]      ; time to wait
    LDR  R1, =NVIC_ST_CURRENT_R
    STR  R0, [R1]      ; any value written to CURRENT clears
    LDR  R1, =NVIC_ST_CTRL_R
SysTick_Wait_loop
    LDR  R0, [R1]      ; read status
    ANDS R0, R0, #0x00010000 ; bit 16 is COUNT flag
    BEQ  SysTick_Wait_loop   ; repeat until flag set
    BX   LR
```

SysTick Timer

```
;------------SysTick_Wait10ms------------
; Call this routine to wait for R0*10 ms
; Time delay using busy wait. This assumes 80 MHz clock
; Input: R0  number of times to wait 10 ms before returning
; Output: none
; Modifies: R0
DELAY10MS  EQU 800000     ; clock cycles in 10 ms
SysTick_Wait10ms
    PUSH {R4, LR}                   ; save R4 and LR
    MOVS R4, R0                     ; R4 = R0 = remainingWaits
    BEQ SysTick_Wait10ms_done       ; R4 == 0, done
SysTick_Wait10ms_loop
    LDR R0, =DELAY10MS              ; R0 = DELAY10MS
    BL  SysTick_Wait                ; wait 10 ms
    SUBS R4, R4, #1                 ; remainingWaits--
    BHI SysTick_Wait10ms_loop       ; if(R4>0), wait another 10 ms
SysTick_Wait10ms_done
    POP {R4, PC}
```

SysTick Timer in C

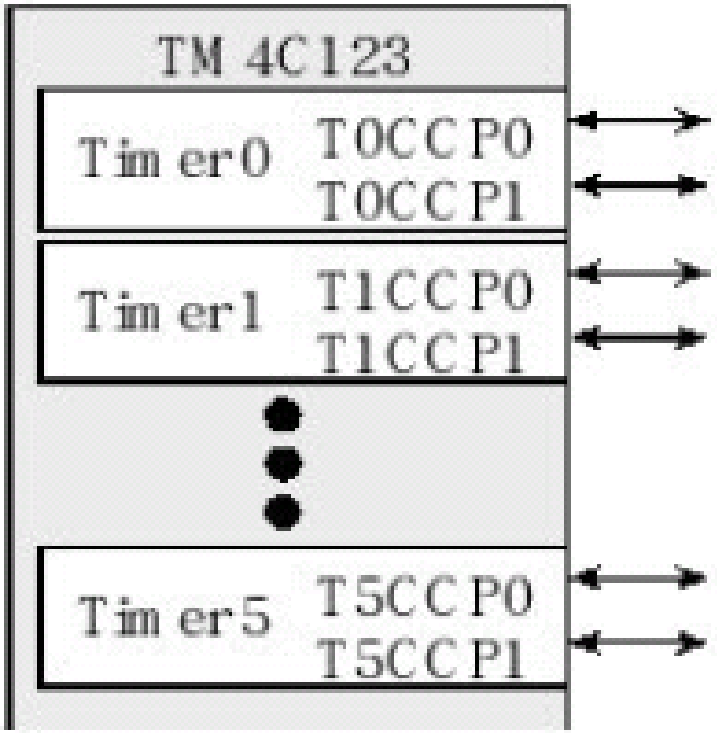
```
#define NVIC_ST_CTRL_R(*((volatile uint32_t *)0xE000E010))
#define NVIC_ST_RELOAD_R(*((volatile uint32_t *)0xE000E014))
#define NVIC_ST_CURRENT_R(*((volatile uint32_t *)0xE000E018))

void SysTick_Init(void){
 NVIC_ST_CTRL_R = 0; // 1) disable SysTick during setup
 NVIC_ST_RELOAD_R = 0x00FFFFFF; // 2) maximum reload value
 NVIC_ST_CURRENT_R = 0; // 3) any write to CURRENT clears it
 NVIC_ST_CTRL_R = 0x00000005; // 4) enable SysTick with core clock
}

// The delay parameter is in units of the 80 MHz core clock(12.5 ns)
void SysTick_Wait(uint32_t delay){
  NVIC_ST_RELOAD_R = delay-1;  // number of counts
  NVIC_ST_CURRENT_R = 0;       // any value written to CURRENT clears
  while((NVIC_ST_CTRL_R&0x00010000)==0){ // wait for flag
  }
}
// Call this routine to wait for delay*10ms
void SysTick_Wait10ms(uint32_t delay){
 unsigned long i;
 for(i=0; i<delay; i++){
   SysTick_Wait(800000); // wait 10ms
}
```

Timer in TM4C123

The TM4C123 has six timers, Output compare and input capture can also be combined to measure period and frequency over a wide range of ranges and resolutions. We may run the output compare modes with or without an external output pin attached.



```
void Timer0A_Init(unsigned short period){ volatile uint32_t delay;
SYSCTL_RCGCTIMER_R |= 0x01;      // 0) activate timer0
  delay = SYSCTL_RCGCTIMER_R;      // allow time to finish activating
  TIMER0_CTL_R &= ~0x00000001;     // 1) disable timer0A during setup
  TIMER0_CFG_R = 0x00000004;     // 2) configure for 16-bit timer mode
  TIMER0_TAMR_R = 0x00000002;      // 3) configure for periodic mode
  TIMER0_TAILR_R = period - 1;     // 4) reload value
  TIMER0_TAPR_R = 49;              // 5) 预分频1us timer0A=50MHZ/50
  TIMER0_ICR_R = 0x00000001;       // 6) clear timer0A timeout flag
  TIMER0_IMR_R |= 0x00000001;      // 7) arm timeout interrupt
  NVIC_PRI4_R = (NVIC_PRI4_R&0x00FFFFFF)|0x60000000; // 8)
priority 3
  NVIC_EN0_R = NVIC_EN0_INT19;     // 9) enable interrupt 19 in NVIC
  TIMER0_CTL_R |= 0x00000001;      // 10) enable timer0A
}
Timer0A_Init(5);          // 200 kHz, period=5
200KHZ=50MHZ/(49+1)/5
```

the start values in the GPTMTAILR and GPTMTAPR registers are loaded into the Timer A, Then the timer begins the count-down by first decrementing

For a periodic timer, the start values are reloaded from the GPTMTAILR and the GPTMTAPR registers into the timer and continue for the next cycle.

Input/Output Synchronization

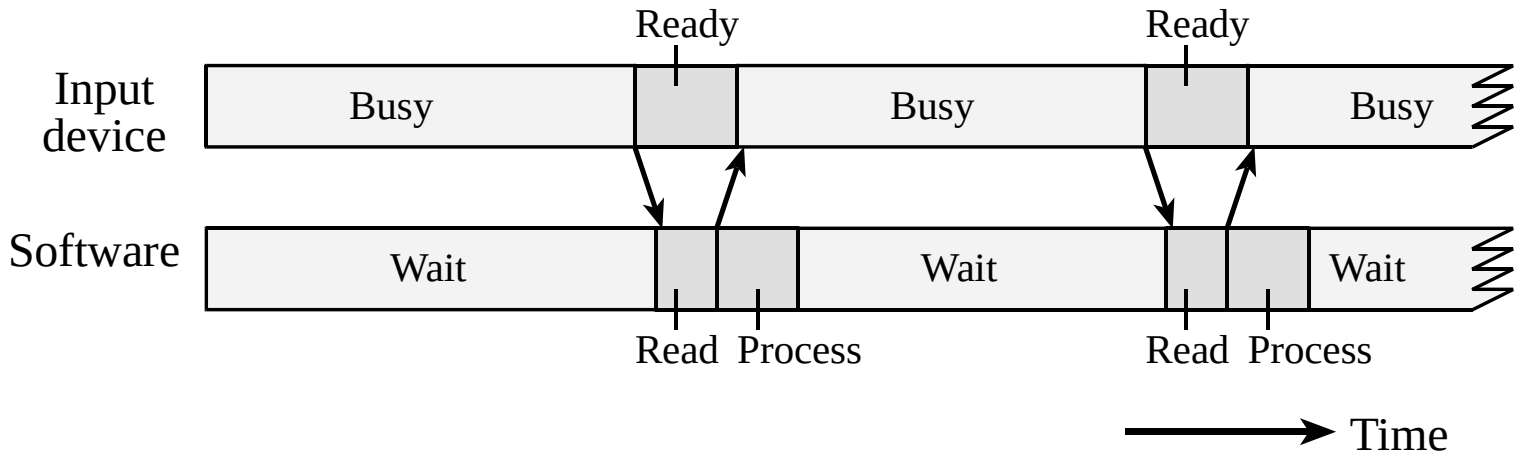
- Processor-Peripheral Timing Mismatch
  - Peripherals, *e.g.*, displays, sensors, switches, generally operate MUCH slower than processor instruction times
    - Processor ~ MHz
    - Peripheral ~ kHz or Hz
  - MANY instructions can be executed while peripheral processes information

Input/Output Sync. (cont.)

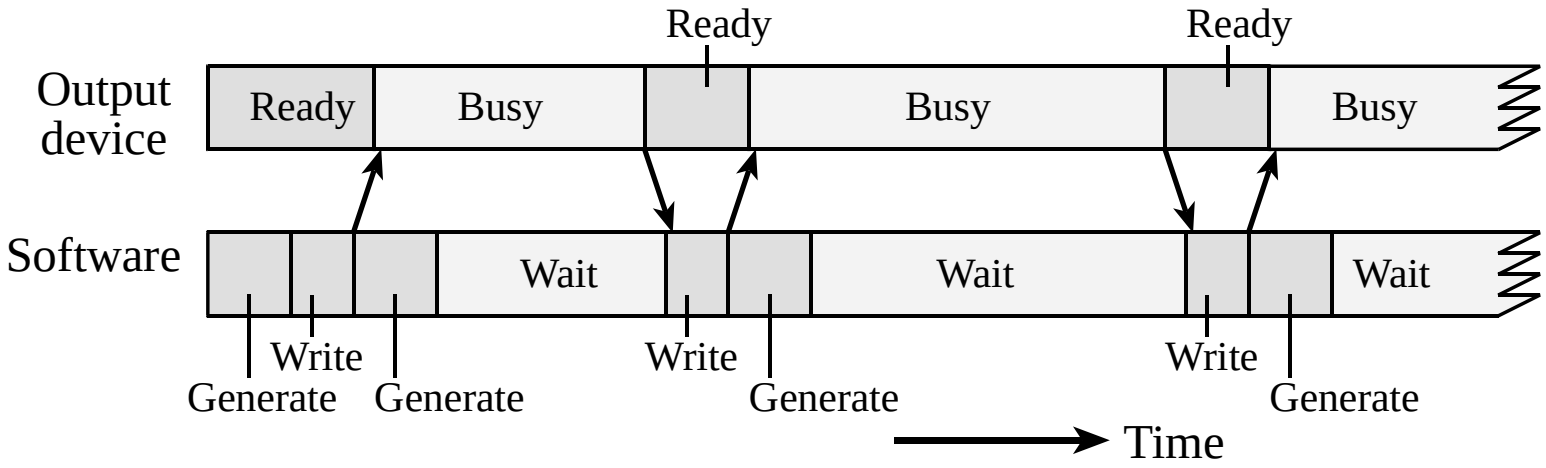
- Peripheral primitive states
  - READY
    - Peripheral is ready to initiate an I/O transfer
  - NOT READY
    - Peripheral is unable to perform I/O transfer
  - BUSY
    - READY peripheral becomes BUSY when I/O transfer initiated
    - Peripheral remains BUSY for duration of I/O transfer
    - Another transfer can NOT be initiated
  - NOT BUSY
    - READY peripheral is able to initiate another I/O operation

Input/Output Sync. (cont.)

INPUT



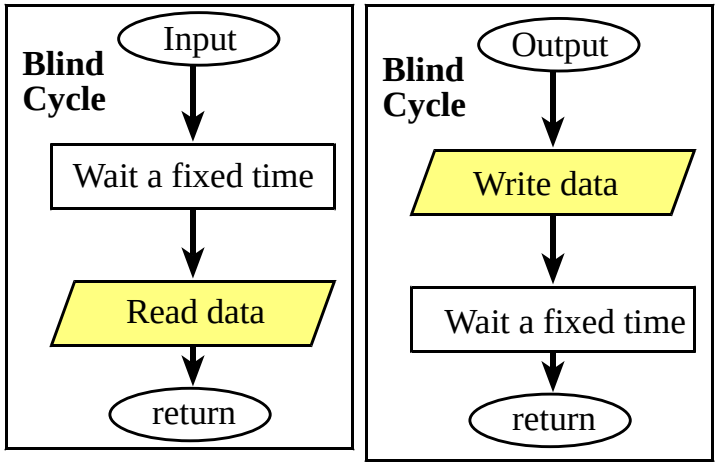
OUTPUT



I/O Sync Options (1)

What to do while the peripheral is BUSY?

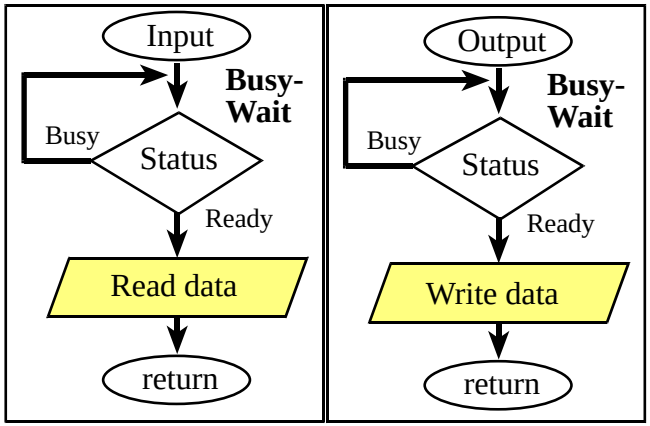
1. **BLIND CYCLE TRANSFER**
   - **Suppose that a BUSY control signal is not available**
   - **Perform I/O operation**
   - **Wait for a period of time that is guaranteed to be sufficient for operation to complete**
   - **Initiate next operation**



I/O Sync Options (2)

What to do while the peripheral is BUSY?

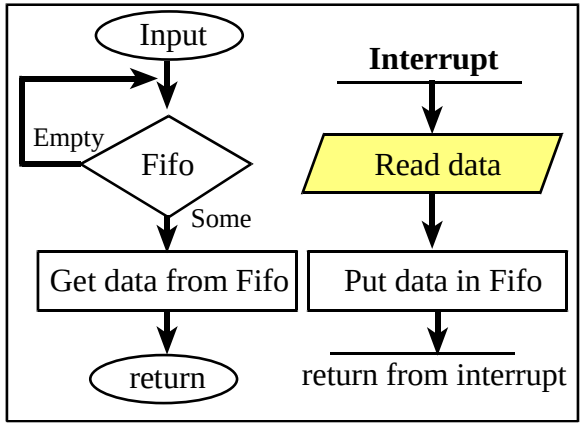
2. **BUSY-WAIT (e.g., ready-busy, test-transfer)**
   - **Poll peripheral status – wait for READY/NOT BUSY**
   - **Perform other tasks between polls**
   - **Unless timed correctly, *under/over run* possible**
     - **One solution: POLL CONTINUOUSLY**

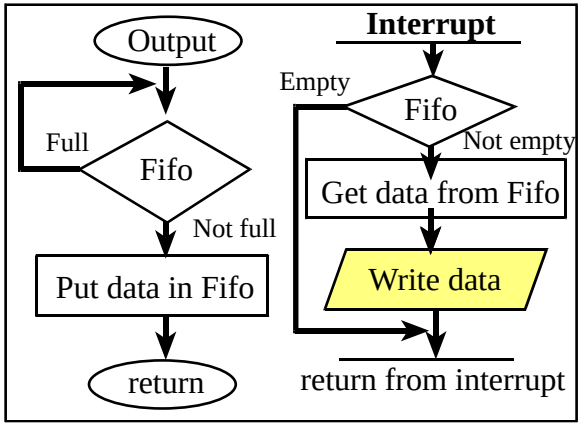


I/O Sync Options (3)

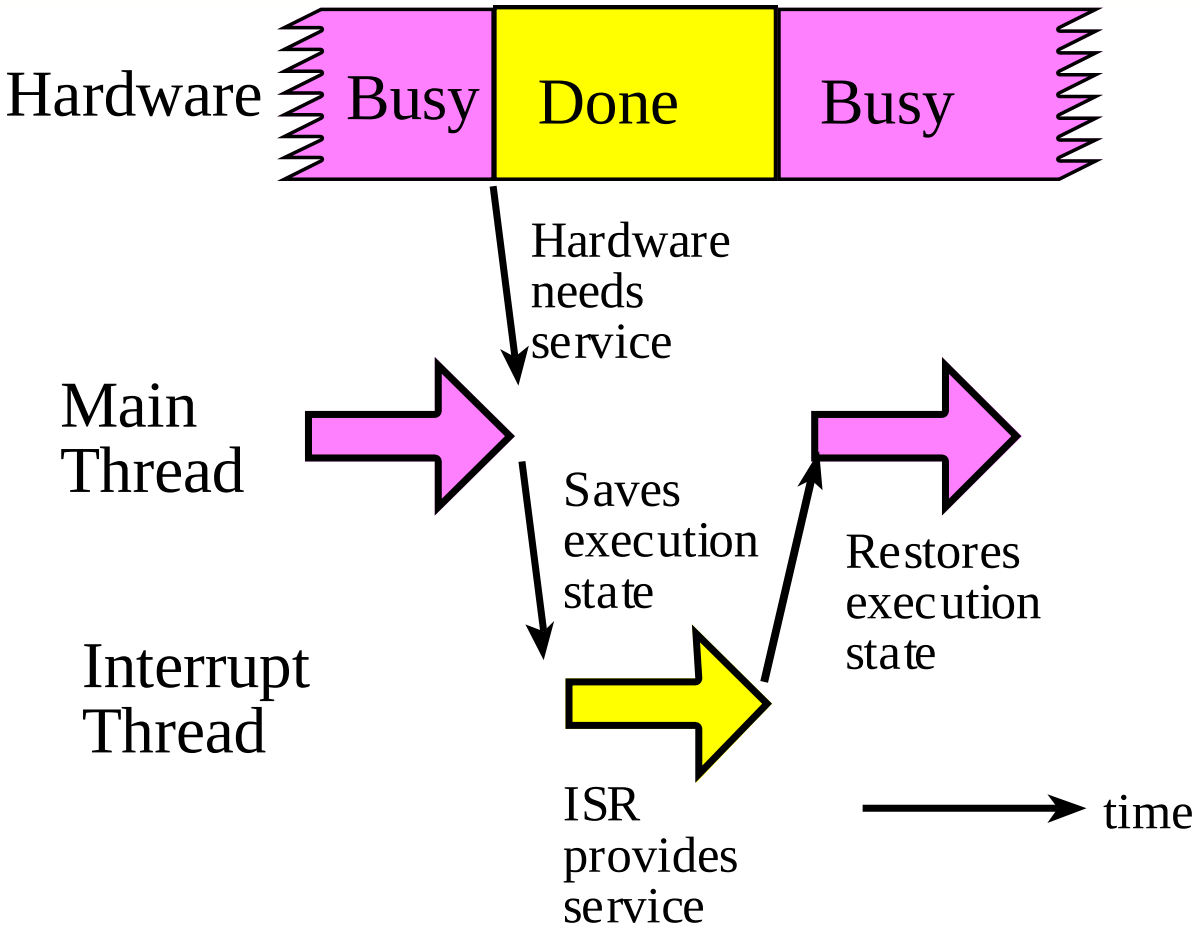
What to do while the peripheral is BUSY?

3. **INTERRUPT/TRANSFER**
   - **Hardware INTERRUPTS processor on condition of READY/NOT BUSY**
   - **Facilitates performing other – *background* - processing between I/O transfers**
     - **Processor changes context when current transfer complete**
     - **Requires program structure to process context change**





Interrupt Processing



Interrupts

- An **interrupt** is the automatic transfer of software execution in response to a hardware **event** that is **asynchronous** with the current software execution
- This hardware event is called **a trigger** and it breaks the execution flow of the **main thread** of the program
- The event causes the CPU to stop executing the current program and begin executing a special piece of code called an **interrupt handler** or **interrupt service routine (ISR)**
- Typically, the ISR does some work and then resumes the interrupted program

Internal

The hardware event can either be:

1) A **busy-to-ready transition** in an external I/O device. Caused by the external world

–Peripheral/device, e.g., UART input/output device

–Reset button, Timer expires, Power failure, System error

–Names: exception, interrupt, **external interrupt**

2) An **internal event**

–Bus fault, memory fault

–A periodic timer

–Div. by zero, illegal/unsupported instruction

–Names : exception, trap, **system exception**

- When the hardware needs service, signified by a busy to ready state transition, it will request an interrupt by setting its trigger flag

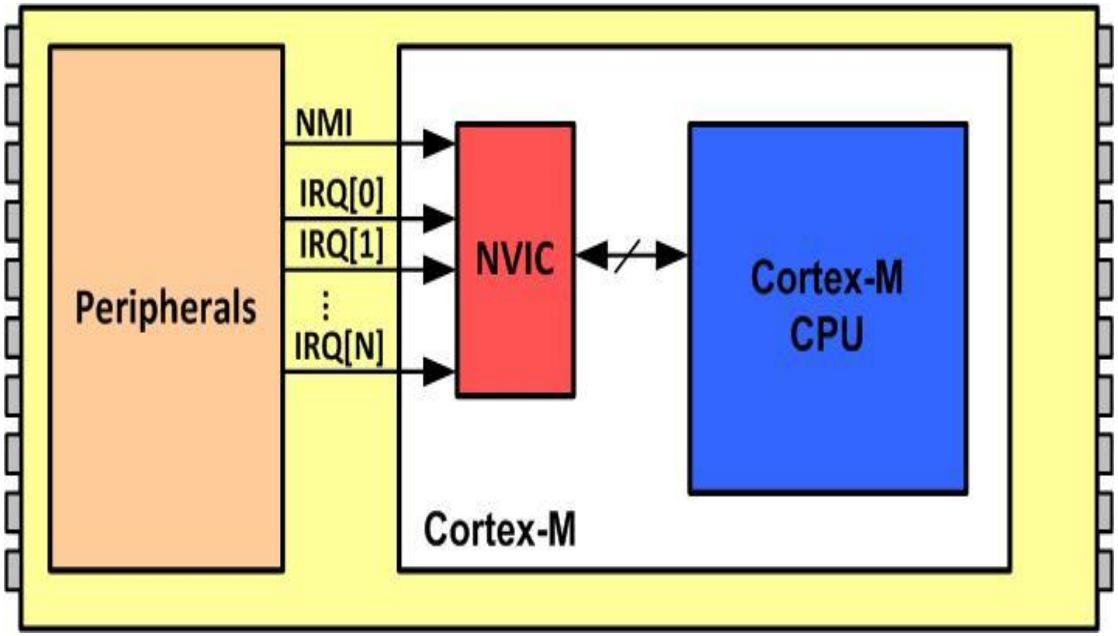
Cortex-M3 Interrupts

- Exceptions:
  - System exceptions: numbered 1 to 15
  - External interrupt inputs: numbered from 16 up

Different numbers of external interrupt inputs (from 1 to 240) and different numbers of priority levels

- Value of the current running exception is indicated by:
  - The special register Interrupt Program Status Register (IPSR) or
  - From the NVIC’ s Interrupt Control State Register (the VECTACTIVE field)

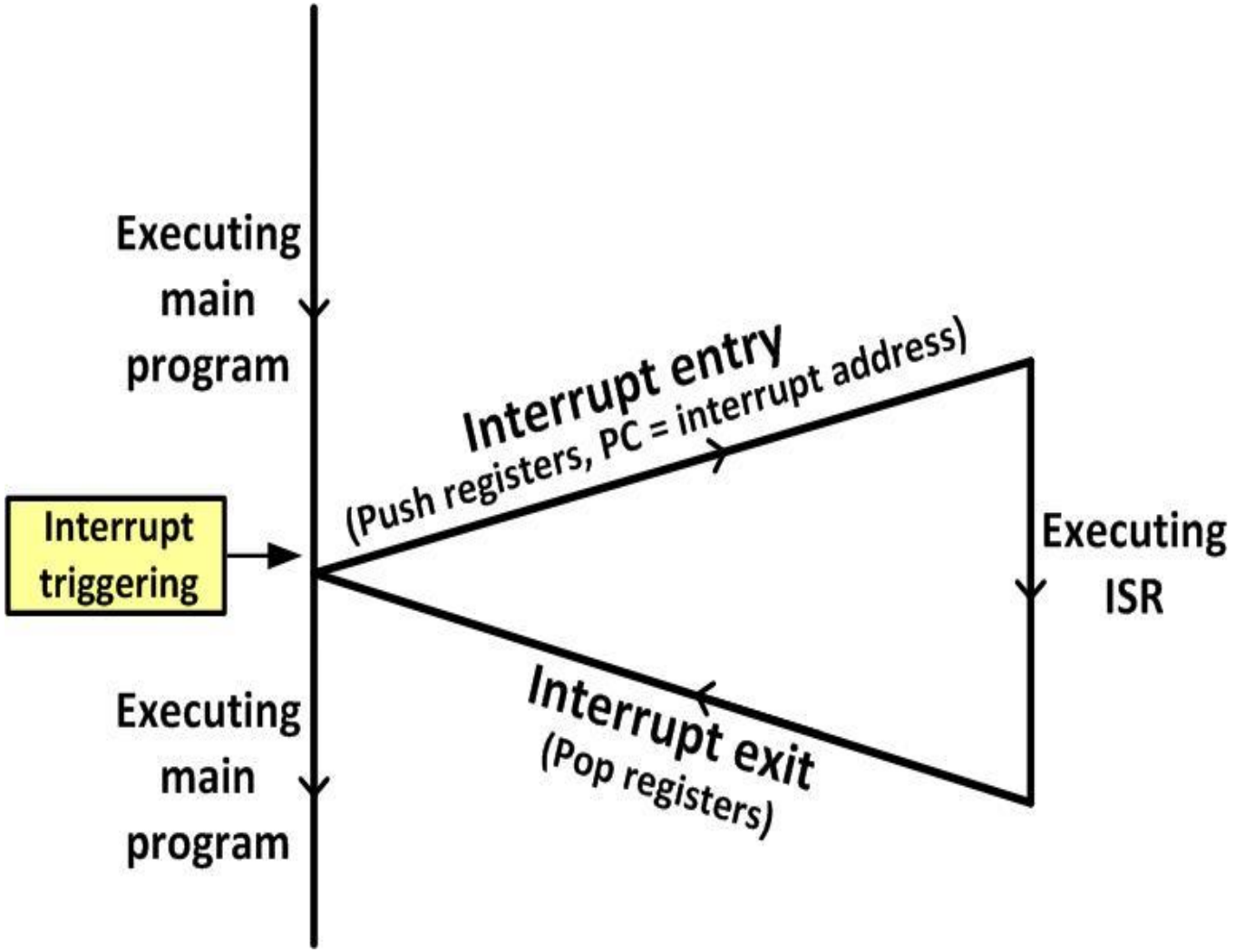
NVIC in ARM Cortex-M



Interrupt Vector Table for ARM Cortex-M

Interrupt #	Interrupt	Memory Location (Hex)
	Stack Pointer initial value	0x00000000
1	Reset	0x00000004
2	NMI	0x00000008
3	Hard Fault	0x0000000C
4	Memory Management Fault	0x00000010
5	Bus Fault	0x00000014
6	Usage Fault (undefined instructions, divide by zero, unaligned memory access,...)	0x00000018
7	Reserved	0x0000001C
8	Reserved	0x00000020
9	Reserved	0x00000024
10	Reserved	0x00000028
11	SVCall	0x0000002C
12	Debug Monitor	0x00000030
13	Reserved	0x00000034
14	PendSV	0x00000038
15	SysTick	0x0000003C
16	IRQ 0 for peripherals	0x00000040
17	IRQ 1 for peripherals	0x00000044
...	...	...
255	IRQ 239 for peripherals	0x000003FC

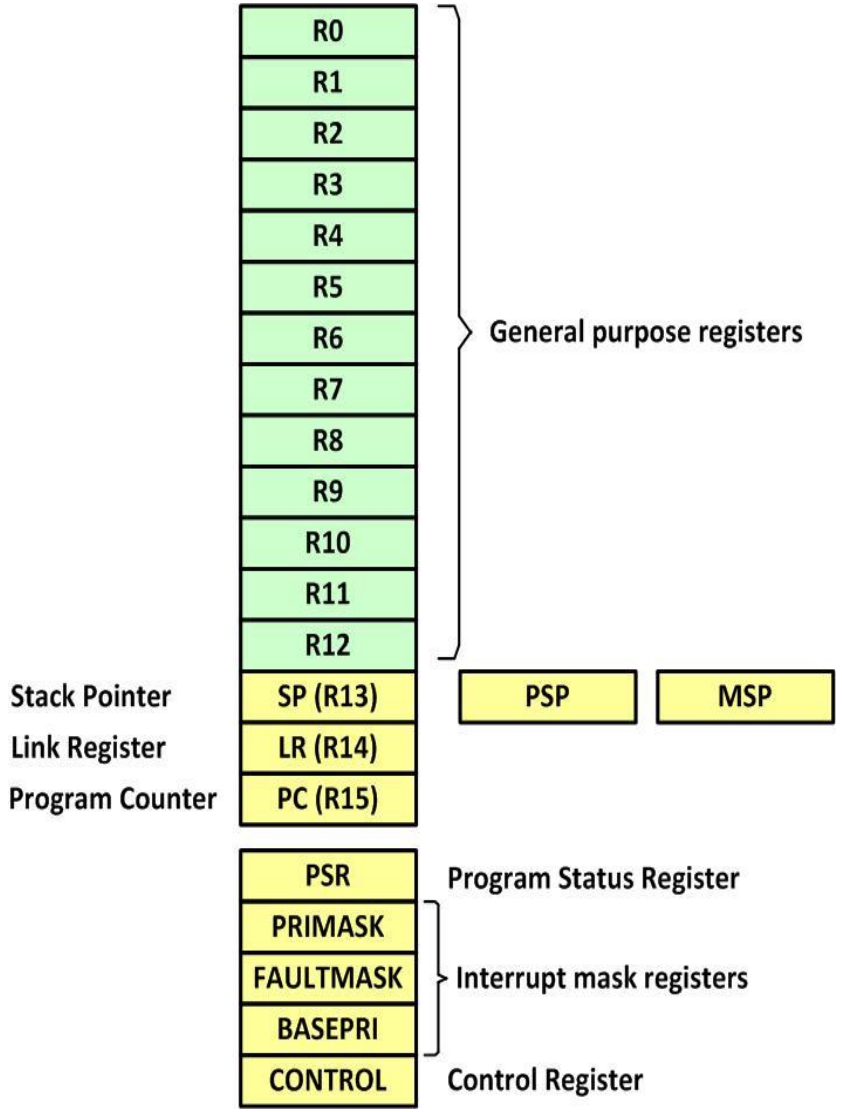
Main Program gets interrupted



Interrupt Priority for ARM Cortex-M

Interrupt #	Interrupt	Priority Level
0	Stack Pointer initial value	
1	Reset	-3 Highest
2	NMI	-2
3	Hard Fault	-1
4	Memory Management Fault	Programmable
5	Bus Fault	Programmable
6	Usage Fault (undefined instructions, divide by zero, unaligned memory access,....)	Programmable
7	Reserved	Programmable
8	Reserved	Programmable
9	Reserved	Programmable
10	Reserved	Programmable
11	SVCall	Programmable
12	Debug Monitor	Programmable
13	Reserved	Programmable
14	PendSV	Programmable
15	SysTick	Programmable
16	IRQ 0 for peripherals	Programmable
17	IRQ 1 for peripherals	Programmable
...	...	Programmable
255	IRQ 239 for peripherals	Programmable

ARM Cortex-M Registers



IRQ assignment in Tiva ARM TM123GH6PM

INT#	IRQ#	Vector location	Device
1-15	none	0000 0000 to 0000 003C	CPU Exception (set by ARM)
16	0	0000 0040	GPIO PORT A
17	1	0000 0044	GPIO PORT B
18	2	0000 0048	GPIO PORT C
19	3	0000 004C	GPIO PORT D
20	4	0000 0050	GPIO PORT E
21	5	0000 0054	UART0
22	6	0000 0058	UART1
23	7	0000 005C	SSI0
24	8	0000 0060	I2C0
25	9	0000 0064	PWM0 Fault
26	10	0000 0068	PWM0 Generator 0
27	11	0000 006C	PWM0 Generator 1
28	12	0000 0070	PWM0 Generator 2

IRQ assignment in Tiva ARM TM123GH6PM(Cont.)

INT#	IRQ#	Vector location	Device
29	13	0000 0074	QEI0
30	14	0000 0078	ADC0 Sequence 0
31	15	0000 007C	ADC0 Sequence 1
32	16	0000 0080	ADC0 Sequence 2
33	17	0000 0084	ADC0 Sequence 3
34	18	0000 0088	Watchdog Timers 0 and 1
35	19	0000 008C	16/32-Bit Timer 0A
36	20	0000 0090	16/32-Bit Timer 0B
37	21	0000 0094	16/32-Bit Timer 1A
38	22	0000 0098	16/32-Bit Timer 1B
39	23	0000 009C	16/32-Bit Timer 2A
40	24	0000 00A0	16/32-Bit Timer 2B
41	25	0000 00A4	Analog Comparator 0
42	26	0000 00A8	Analog Comparator 1

IRQ assignment in Tiva ARM TM123GH6PM(Cont.)

INT#	IRQ#	Vector location	Device
43	27	-	Reserved
44	28	0000 00B0	System Control
45	29	0000 00B4	Flash Memory Control and EEPROM Control
46	30	0000 00B8	GPIO Port F
47-48	31-32	-	Reserved
49	33	0000 00C4	UART2
50	34	0000 00C8	SSI1
51	35	0000 00CC	16/32-Bit Timer 3A
52	36	0000 00D0	16-32-Bit Timer 3B
53	37	0000 00D4	I2C1
54	38	0000 00D8	QEI1
55	39	0000 00DC	CAN0
56	40	0000 00E0	CAN1
57-58	41-42	-	Reserved

IRQ assignment in Tiva ARM TM123GH6PM(Cont.)

INT#	IRQ#	Vector location	Device
59	43	0000 00EC	Hibernation Module
60	44	0000 00F0	USB
61	45	0000 00F4	PWM Generator 3
62	46	0000 00F8	µDMA Software
63	47	0000 00FC	µDMA Error
64	48	0000 0100	ADC1 Sequence 0
65	49	0000 0104	ADC1 Sequence 1
66	50	0000 0108	ADC1 Sequence 2
67	51	0000 010C	ADC1 Sequence 3
68-72	52-56	-	Reserved
73	57	0000 0124	SSI2
74	58	0000 0128	SSI3
75	59	0000 012C	UART3
76	60	0000 0130	UART4

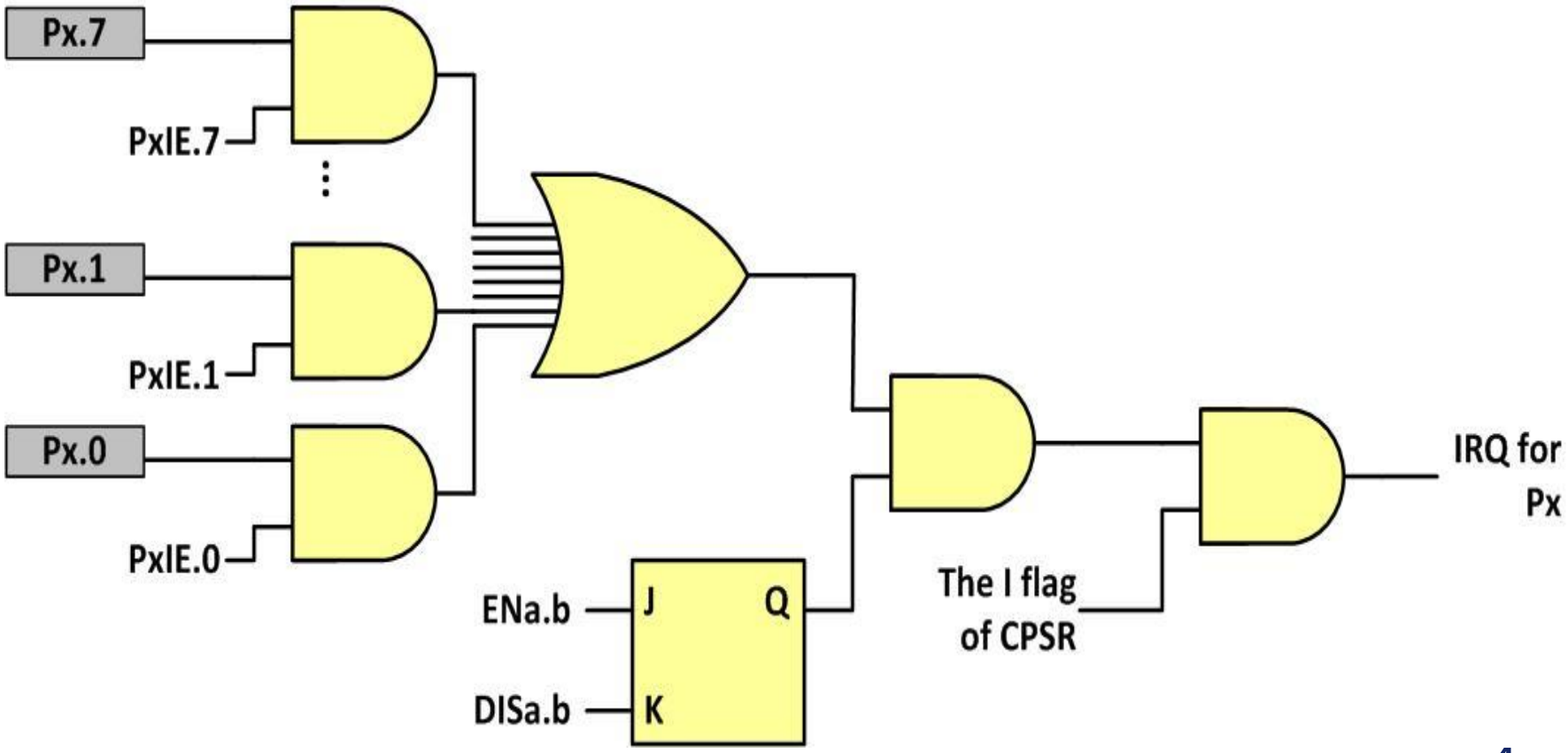
IRQ assignment in Tiva ARM TM123GH6PM(Cont.)

INT#	IRQ#	Vector location	Device
77	61	0000 0134	UART5
78	62	0000 0138	UART6
79	63	0000 013C	UART7
80-83	64-67	-	Reserved
84	68	0000 0150	I2C2
85	69	0000 0154	I2C3
86	70	0000 0158	16/32-Bit Timer 4A
87	71	0000 015C	16/32-Bit Timer 4B
88-107	72-91	-	Reserved
108	92	0000 01B0	16/32-Bit Timer 5A
109	93	0000 01B4	16/32-Bit Timer 5B
110	94	0000 01B8	32/64-Bit Timer 0A
111	95	0000 01BC	32/64-Bit Timer 0B
112	96	0000 01C0	32/64-Bit Timer 1A

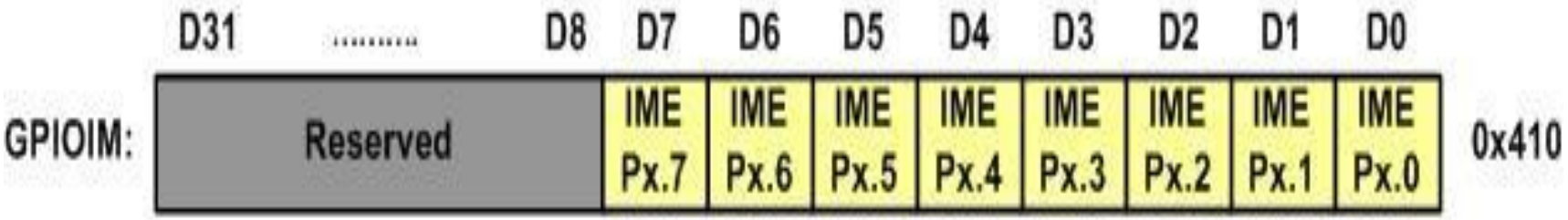
IRQ assignment in Tiva ARM TM123GH6PM(Cont.)

INT#	IRQ#	Vector location	Device
113	97	0000 01C4	32/64-Bit Timer 1B
114	98	0000 01C8	32/64-Bit Timer 2A
115	99	0000 01CC	32/64-Bit Timer 2B
116	100	0000 01D0	32/64-Bit Timer 3A
117	101	0000 01D4	32/64-Bit Timer 3B
118	102	0000 01D8	32/64-Bit Timer 4A
119	103	0000 01DC	32/64-Bit Timer 4B
120	104	0000 01E0	32/64-Bit Timer 5A
121	105	0000 01E4	32/64-Bit Timer 5B
122	106	0000 01E8	System Exception (imprecise)
123-149	107-133	-	Reserved
150	134	0000 0258	PWM Generator 0
151	135	0000 025C	PWM Generator 1
152	136	0000 0260	PWM Generator 2
153	137	0000 0264	PWM Generator 3
154	138	0000 0268	PWM1 Fault

Interrupt enabling with all 3 levels



GPIO Interrupt Mask (GPIOIM)

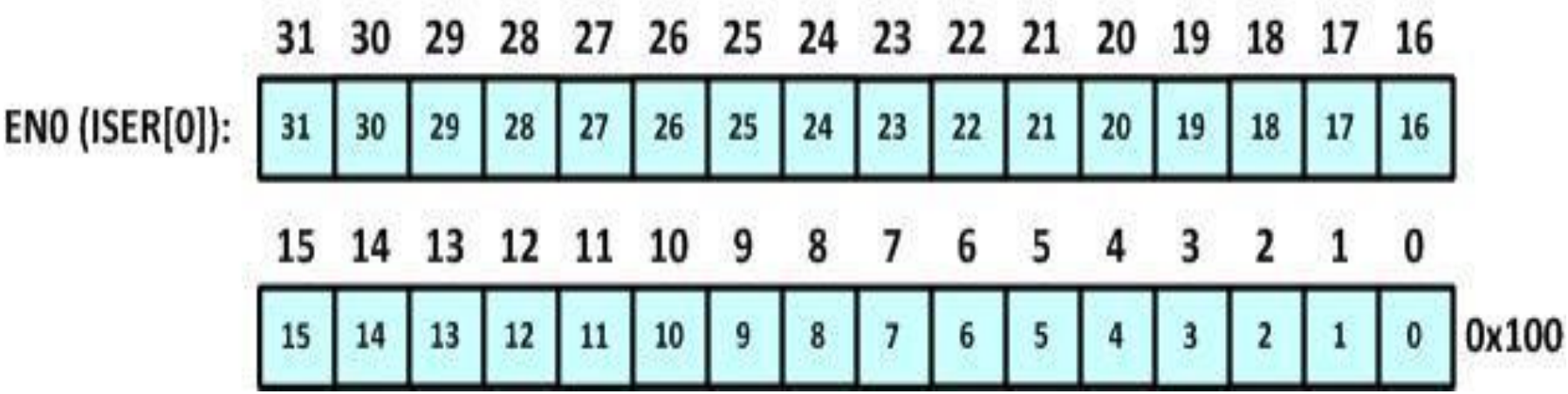


Note: D0 to D7 are used to enable/disable the interrupt for pins 0 to 7 of the port.

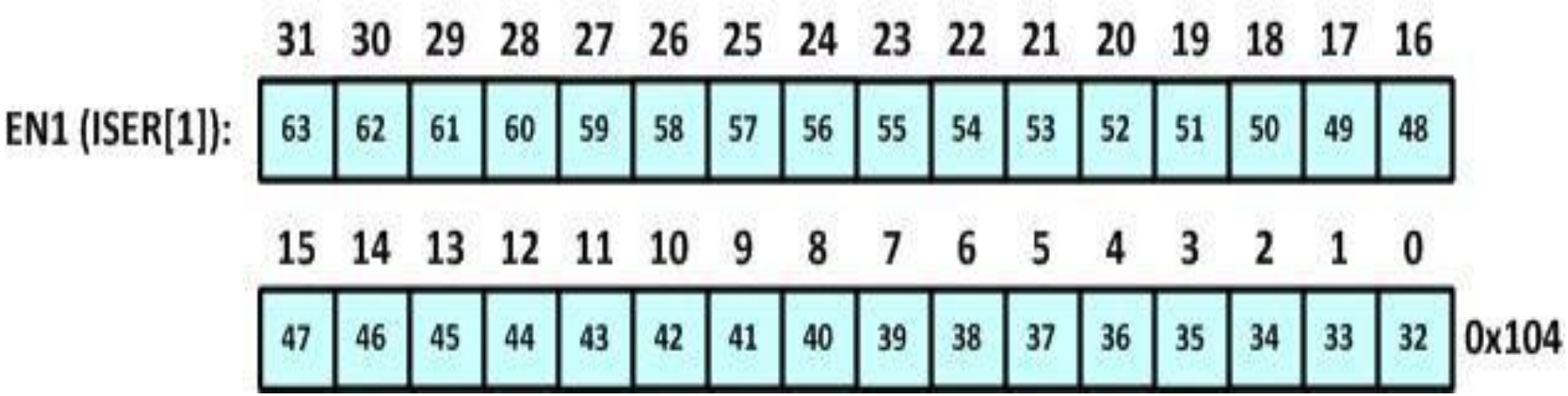
1: Enable interrupt

0: Disable interrupt (mask the interrupt)

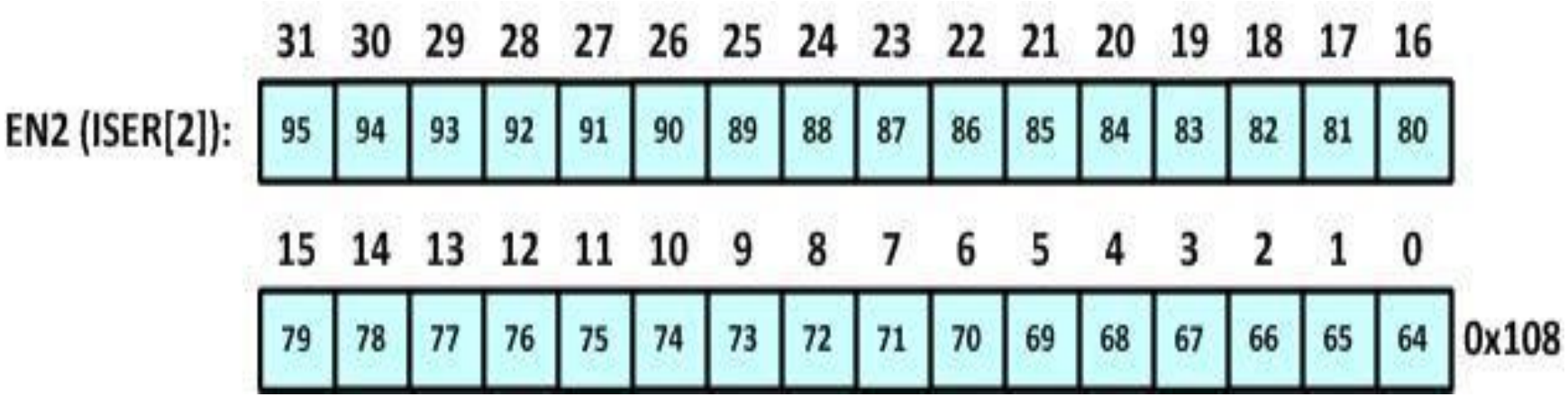
Interrupts 0–31 Set Enable (EN0)



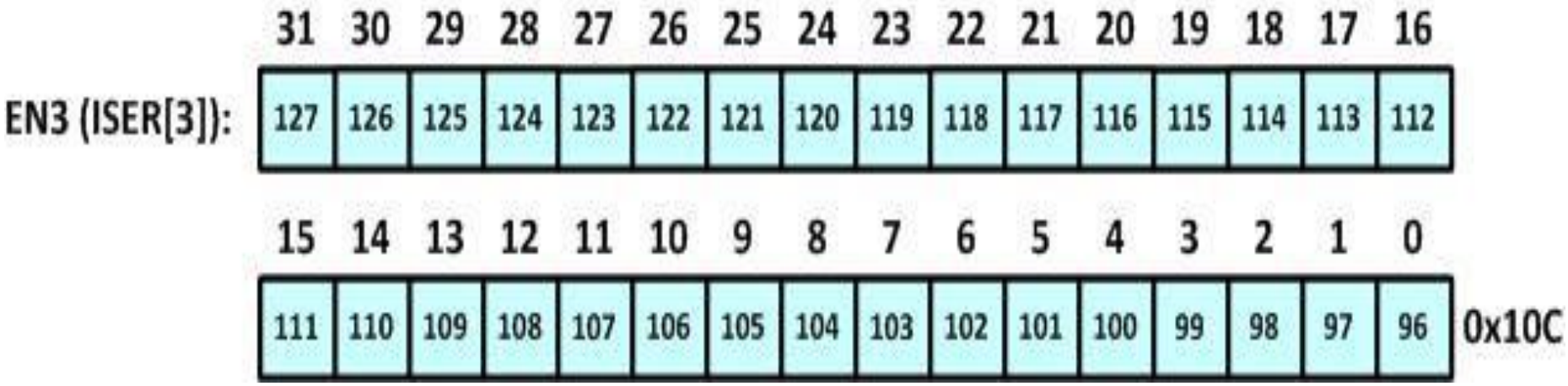
Interrupts 32–63 Set Enable (EN1)



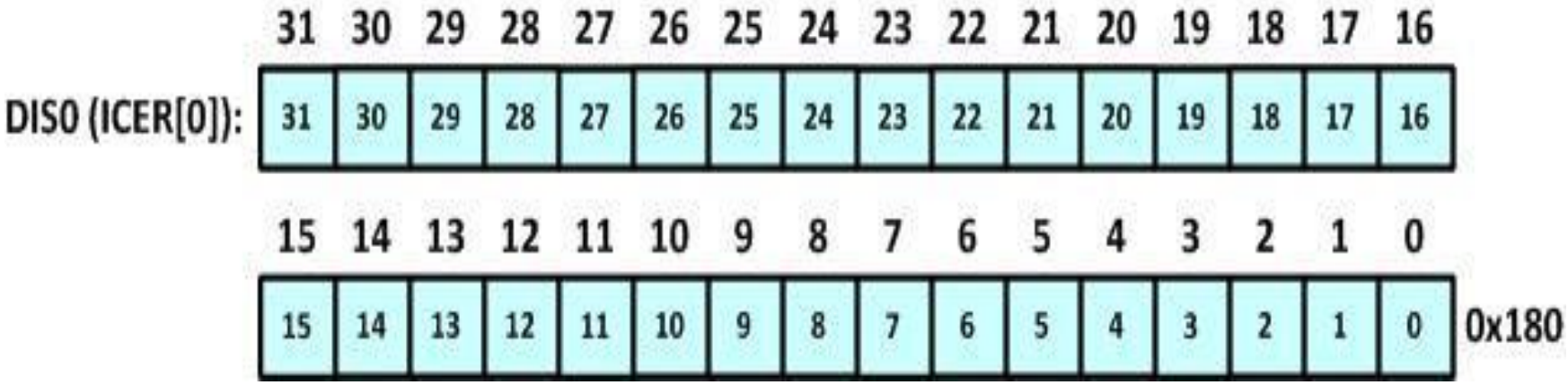
Interrupts 64–95 Set Enable (EN2)



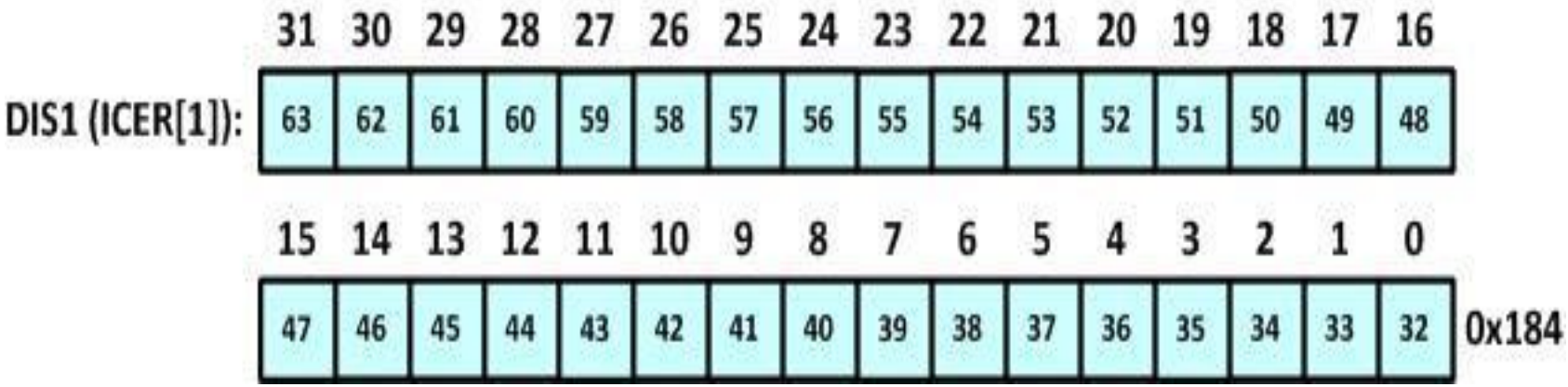
Interrupts 94–127 Set Enable (EN3)



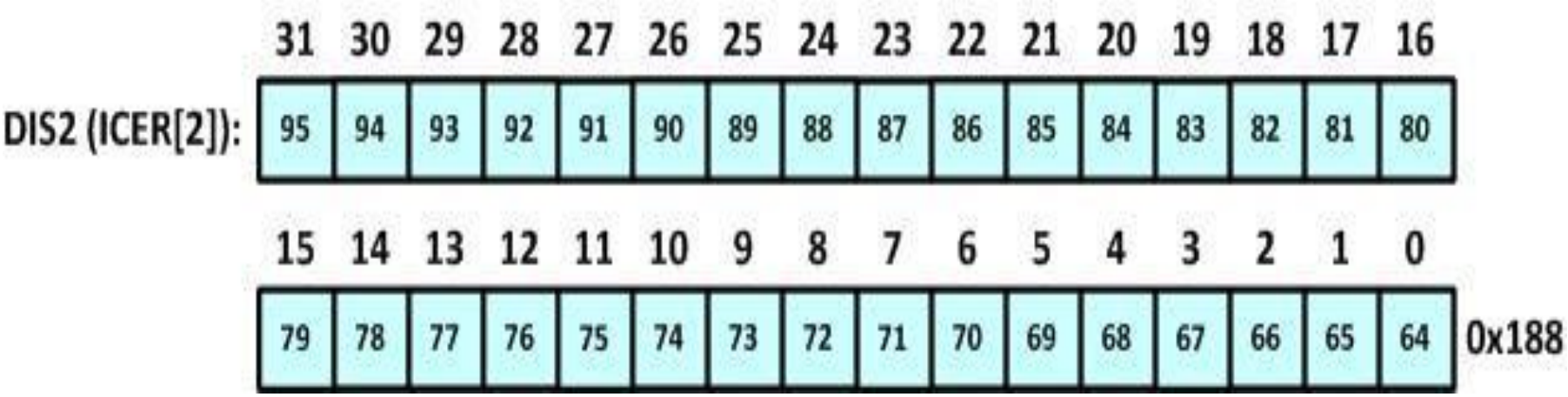
Interrupts 0–31 Clear Enable (DIS0)



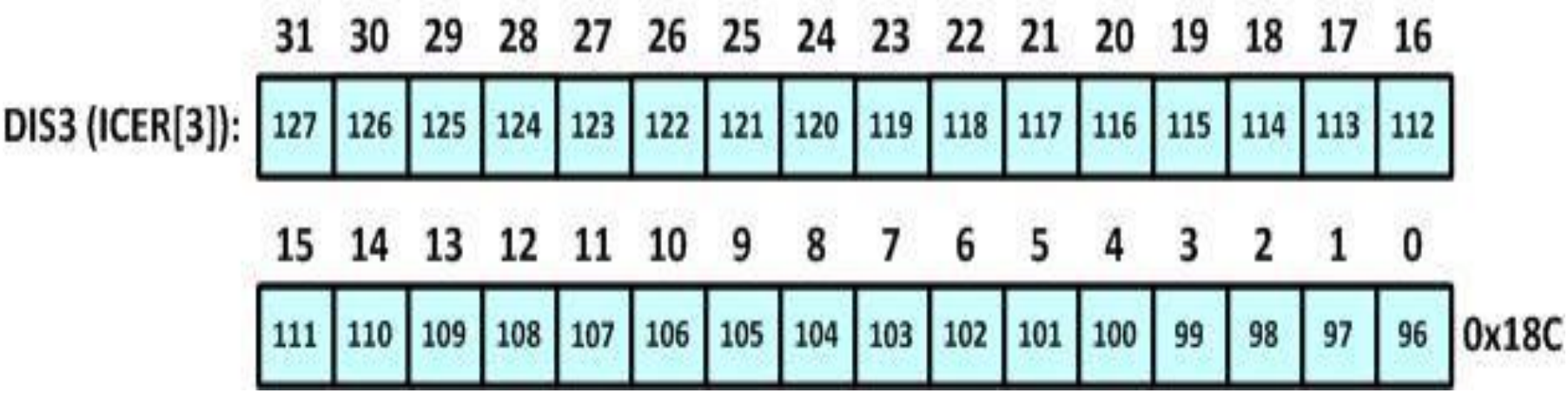
Interrupts 32–63 Clear Enable (DIS1)



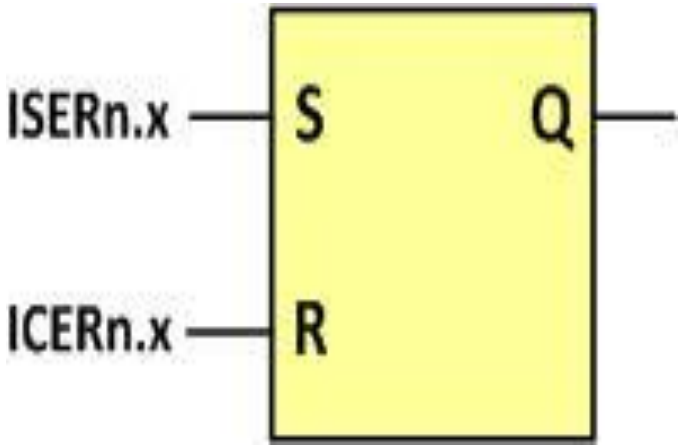
Interrupts 64–95 Clear Enable (DIS2)



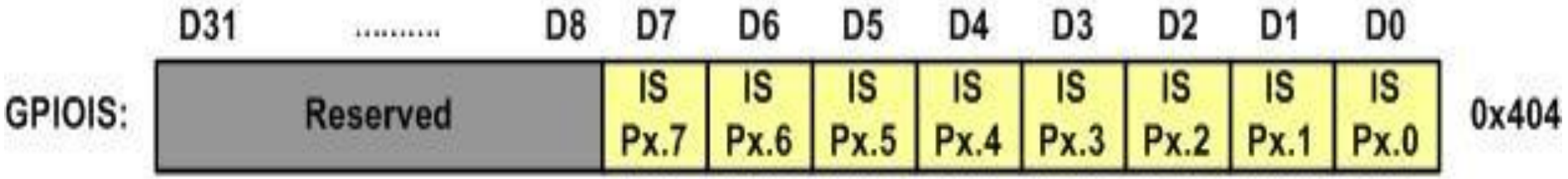
Interrupts 96–127 Clear Enable (DIS3)



Enabling and Disabling an Interrupt



GPIO Interrupt Sense (GPIOIS)

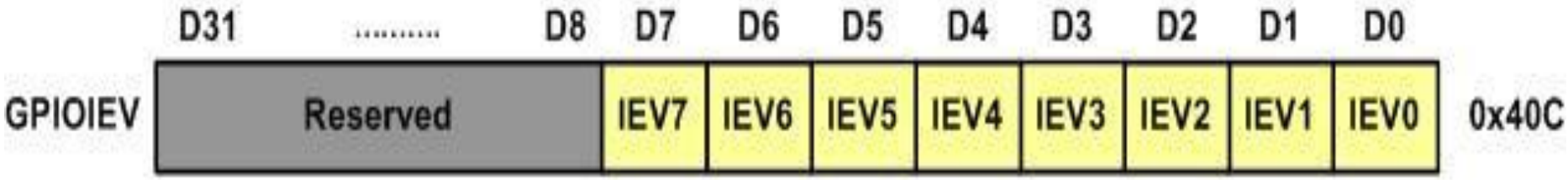


Note: D0 to D7 are used to choose between edge and level sense for pins 0 to 7 of the port.

0: Edge-sensitive

1: Level-sensitive

GPIOIEV

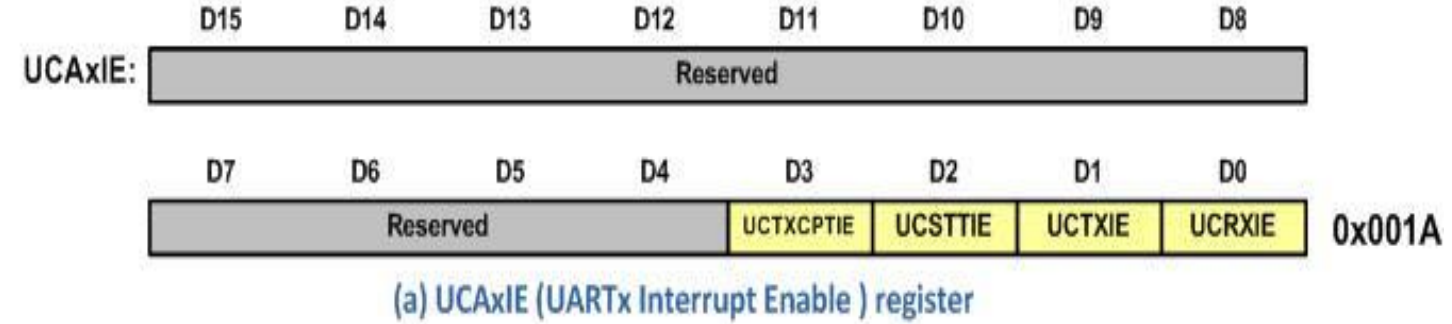


Note: D0 to D7 are used to choose between edge and level sense for pins 0 to 7 of the port.

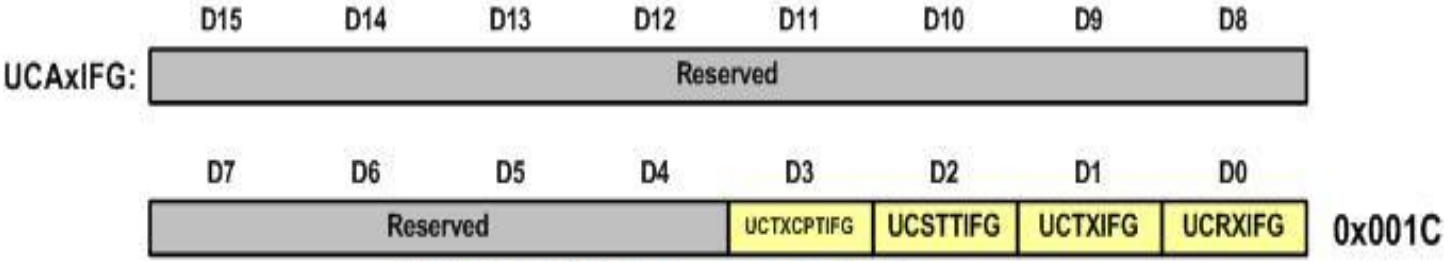
0: A falling edge or a Low level on the corresponding pin triggers an interrupt

1: A rising edge or a High level on the corresponding pin triggers an interrupt

UART Interrupt Registers



(a) UCAxIE (UARTx Interrupt Enable) register



(b) UCAxIFG (UARTx Interrupt flag) register

Using GPIOIM and GPIOIEV Registers

IS.n (interrupt sense)	IEV.n (Interrupt Event)	
0	0	Falling edge
0	1	Rising edge
1	0	Low level
1	1	High level

Interrupt Programming

Interrupts on the Cortex-M are controlled by the **Nested Vectored Interrupt Controller** (NVIC)

- To activate an "interrupt source" we need to set its priority and enable that source in the NVIC:

Activate = Set priority + Enable source in NVIC

NVIC supports 1 to 240 external interrupt inputs (commonly known as IRQs)

- NVIC control registers are accessible as memory-mapped devices
- NVIC can be accessed as memory location **0xE000E000**

Basic Interrupt Configuration

- Each external interrupt has several registers associated with it:
  - Enable and clear enable registers
  - Set-pending and clear-pending registers
  - Active status
  - Priority level
- In addition, a number of other registers can also affect the interrupt processing:
  - Exception-masking registers (PRIMASK, FAULTMASK, and BASEPRI)
  - Vector Table Offset register
  - Software Trigger Interrupt register
  - Priority Group

Interrupt Enable and Clear Enable

The Interrupt Enable register is programmed via two addresses

–To set the enable bit, we write to the **SETENA** register address

–To clear the enable bit, you need to write to the **CLRENA** register address

欲使能一个中断，你需要写1 到对应SETENA 的位中；欲除能一个中断，你需要写1 到对应的CLRENA 位中；如果往它们中写0，不会有任何效果。通过这种方式，使能 / 除能中断时只需把“当事位”写成1，其它的位可以全部为零。再也不用像以前那样，害怕有些位被写入0 而破坏其对应的中断设置（写0 没有效果），从而实现每个中断都可以自顾地设置，而互不侵犯——只需单一的写指令，不再需要读-改-写。

Definitions of Priority

异常的优先级:

一个高优先级的异常可以抢占一个低优先级的异常.

复位, NMI, 和硬件错误有特定的优先级.

Cortex-M3 支持256级可编程异常.

减少优先级数可以通过减少优先级配置寄存器的一些低位来实现。

每个外部中断都有一个对应的优先级寄存器，每个寄存器占用8 位，但是允许最少只使用最高3 位

Active Status

每个外部中断都有一个活动状态位。在处理器执行了其ISR 的第一条指令后，它的活动位就被置1，并且直到ISR返回时才硬件清零。由于支持嵌套，允许高优先级异常抢占某个ISR。然而，哪怕一个中断被抢占，其活动状态也依然为1（请仔细琢磨前文讲到的“直到ISR返回时才清零）。活动状态寄存器的定义，与前面讲的使能/除能和悬起/解悬寄存器相同，只是不再成对出现。它们也能按字 / 半字 / 字节访问，但他们是只读的，如表8.4 所示。

Program Status Register

- Accessed separately or all at once

Figure 3. APSR, IPSR and EPSR bit assignments

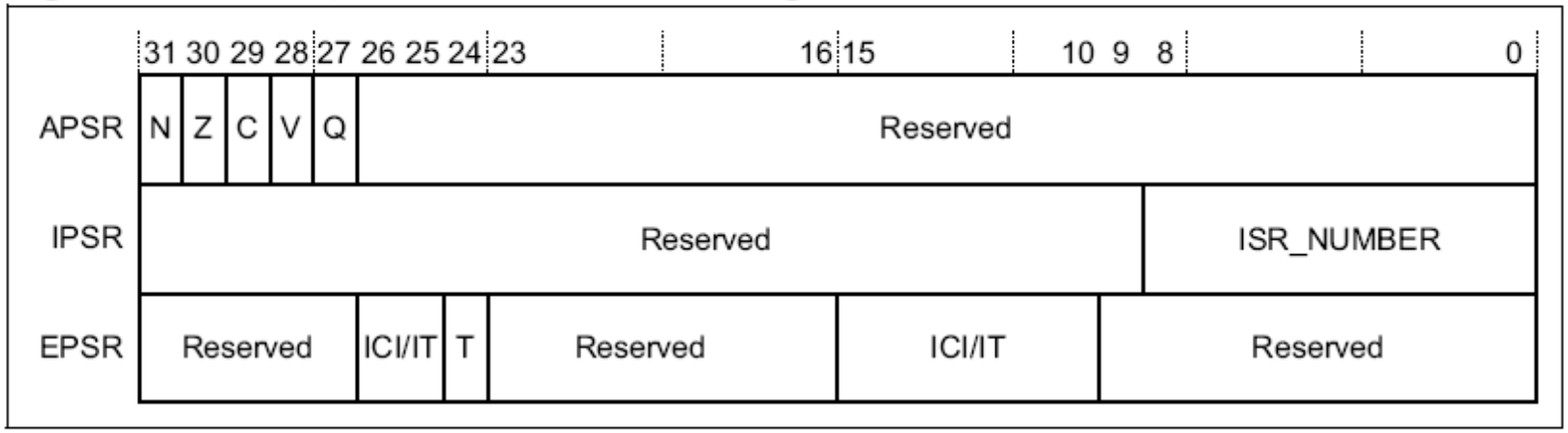
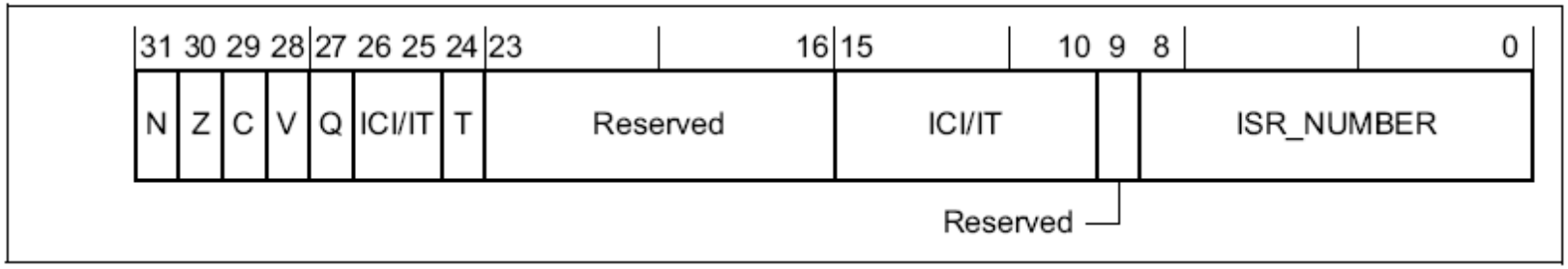


Figure 4. PSR bit assignments



Q = Saturation, T = Thumb bit

Vector Tables

- When an exception takes place and is being handled by the Cortex-M3, the processor will need to locate the starting address of the exception handler
- This information is stored in the vector table
- Each exception has an associated 32-bit vector that points to the memory location where the ISR that handles the exception is located
- Vectors are stored in ROM at the beginning of the memory

Vector Tables

- Exception vector table after power-up is located at address 0x00000000:
- ROM location 0x00000000 has the initial stack pointer
- Location 0x00000004 contains the initial program counter (PC), which is called the reset vector
- Reset vector points to a function called reset handler, which is the first thing executed following reset
- Vector table can be relocated to change interrupt handlers at runtime (vector table offset register)

Address	Exception Number	Value (Word Size)
0x00000000	–	MSP initial value
0x00000004	1	Reset vector (program counter initial value)
0x00000008	2	NMI handler starting address
0x0000000C	3	Hard fault handler starting address
…	…	Other handler starting address

```
EXPORT __Vectors
__Vectors ; address ISR
DCD StackMem + Stack ; 0x00000000 Top of Stack
DCD Reset_Handler ; 0x00000004 Reset Handler
DCD NMI_Handler ; 0x00000008 NMI Handler
DCD HardFault_Handler ; 0x0000000C Hard Fault Handler
DCD MemManage_Handler ; 0x00000010 MPU Fault Handler
DCD BusFault_Handler ; 0x00000014 Bus Fault Handler
DCD UsageFault_Handler ; 0x00000018 Usage Fault Handler
DCD 0 ; 0x0000001C Reserved
DCD 0 ; 0x00000020 Reserved
DCD 0 ; 0x00000024 Reserved
DCD 0 ; 0x00000028 Reserved
DCD SVC_Handler ; 0x0000002C SVCall Handler
DCD DebugMon_Handler ; 0x00000030 Debug Monitor
Handler
DCD 0 ; 0x00000034 Reserved
DCD PendSV_Handler ; 0x00000038 PendSV Handler
DCD SysTick_Handler ; 0x0000003C SysTick Handler
DCD GPIOPortA_Handler ; 0x00000040 GPIO Port A
DCD GPIOPortB_Handler ; 0x00000044 GPIO Port B
DCD GPIOPortC_Handler ; 0x00000048 GPIO Port C
DCD GPIOPortD_Handler ; 0x0000004C GPIO Port D
DCD GPIOPortE_Handler ; 0x00000050 GPIO Port E
DCD UART0_Handler ; 0x00000054 UART0
DCD UART1_Handler ; 0x00000058 UART1
```

Each exception has an associated 32-bit vector that points to the memory location where the ISR that handles the exception is located. Vectors are stored in ROM at the beginning of memory.

Startup.sfile

```
volatile uint32_t Counts;
#define PD0 (*((volatile uint32_t *)0x40007004))
void SysTick_Init(uint32_t period){
SYSCTL_RCGCGPIO_R |= 0x08; // activate port D
Counts = 0;
GPIO_PORTD_AMSEL_R &= ~0x01; // no analog
GPIO_PORTD_PCTL_R &= ~0x0000000F; // regular GPIO function
GPIO_PORTD_DIR_R |= 0x01; // make PD0 out
GPIO_PORTD_AFSEL_R &= ~0x01; // disable alt funct on PD0
GPIO_PORTD_DEN_R |= 0x01; // enable digital I/O on PD0
NVIC_ST_CTRL_R = 0; // disable SysTick during setup
NVIC_ST_RELOAD_R = period - 1;// reload value
NVIC_ST_CURRENT_R = 0; // any write to current clears it
NVIC_SYS_PRI3_R = (NVIC_SYS_PRI3_R&0x00FFFFFF)|0x40000000;
//priority 2
NVIC_ST_CTRL_R = 0x00000007; // enable with core clock and interrupts
EnableInterrupts();
}
void SysTick_Handler(void){
PD0 ^= 0x01; // toggle PD0
Counts = Counts + 1;
}
```

Program 9.7. Implementation of a periodic interrupt using SysTick (PeriodicSysTickInts_xxx.zip).

Simplified procedure for setting up an interrupt

If the application is stored in ROM and there is no need to change the exception handlers, we can have the whole vector table coded in the beginning of ROM in the Code region (0x00000000)

- This way, the vector table offset will always be 0 and the interrupt vector is already in ROM
- The only steps required to set up an interrupt are:

1) Set up the priority group, if needed
2) Set up the priority of the interrupt
3) Enable the interrupt

Interrupt Service Routines (ISRs)

- When an interrupt/exception takes place, a number of things happen:

1. Stacking (automatic pushing of eight registers' contents to stack)

- PC, PSR, R0–R3, R12, and LR

2. Vector fetch (reading the exception handler starting address from the vector table)

3. Exception vector starts to execute. On the entry of the exception handler, a number of regs are updated:

- stack pointer (SP) to new location
- IPSR (low part of PSR) with new exception number
- program counter (PC) to vector handler
- link register (LR) to special value EXC_RETURN

- Several other registers get updated
- Latency: as short as 12 cycles

Interrupt/Exception Exits

- At the end of the exception handler, an exception exit (a.k.a interrupt return in some processors) is required to restore the system status so that the interrupted program can resume normal execution
- There are three ways to trigger the interrupt return sequence; all of them use the special value stored in the LR in the beginning of the handler:

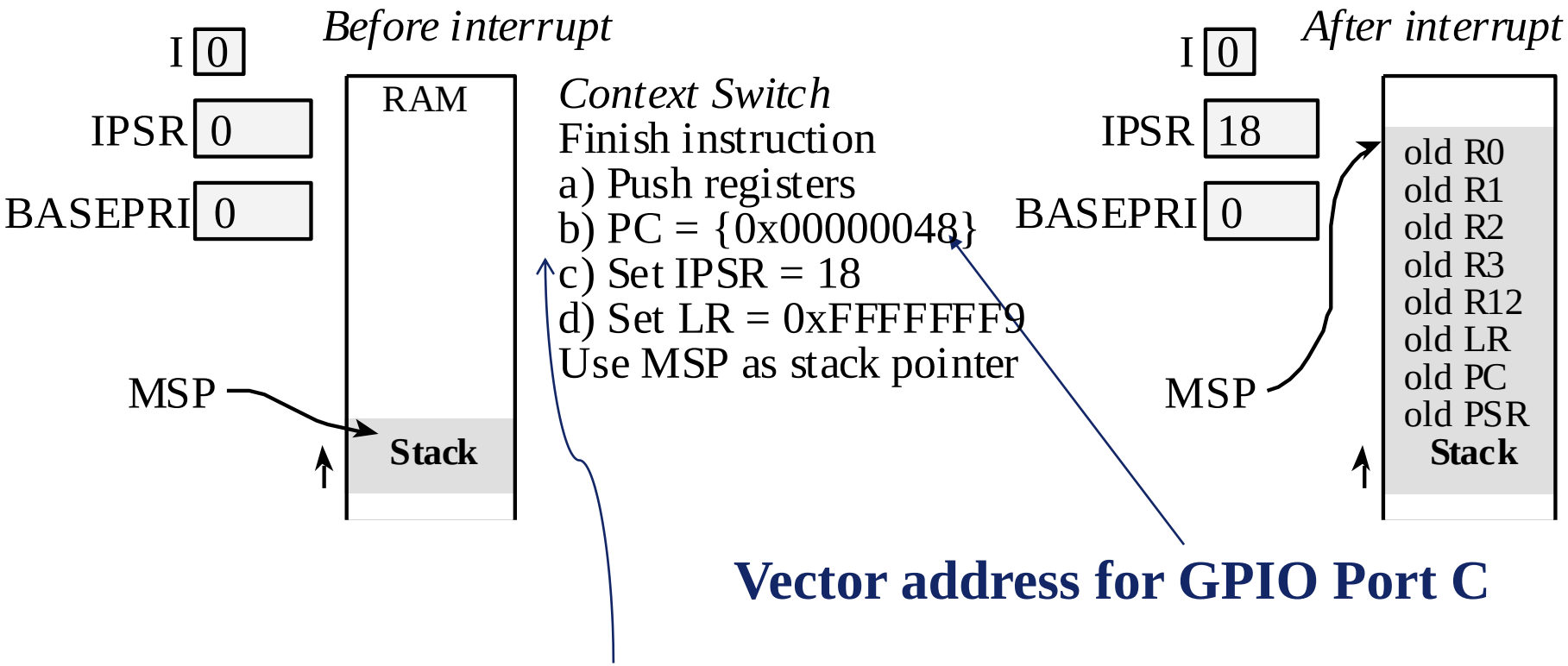
Table 9.2 Instructions that Can be Used for Triggering Exception Return

Return Instruction	Description
BX <*reg*>	If the EXC_RETURN value is still in LR, we can use the *BX LR* instruction to perform the interrupt return.
POP {PC}, or POP {...., PC}	Very often the value of LR is pushed to the stack after entering the exception handler. We can use the POP instruction, either a single POP or multiple POPs, to put the EXC_RETURN value to the program counter. This will cause the processor to perform the interrupt return.
LDR, or LDM	It is possible to produce an interrupt return using the LDR instruction with PC as the destination register.

Interrupt Processing

1. The execution of the main program is suspended
    1. the current instruction is finished,
    2. suspend execution and push 8 registers (R0-R3, R12, LR, PC, PSR) on the stack
    3. LR set to 0xFFFFFFF9 (indicates interrupt return)
    4. IPSR set to interrupt number
    5. sets PC to ISR address
2. The interrupt service routine (ISR) is executed
    - clears the flag that requested the interrupt
    - performs necessary operations
    - communicates using global variables
3. The main program is resumed when ISR executes **BX LR**
    - pulls the 8 registers from the stack

Interrupt Context Switch



To **return from an interrupt**, the ISR executes the typical function return **BX LR** . However, since the top 24 bits of **LR** are 0xFFFFFF, it knows to return from interrupt by popping the eight registers off the stack.

```
GPIOPortC_Handler
  LDR R0,=GPIO_PORTC_ICR_R
  MOV R1,#0x10
  STR R1,[R0]  ; ack
  ;stuff
  BX  LR    ;return from interrupt
```

```
void GPIOPortC_Handler(void){
  GPIO_PORTC_ICR_R = 0x10; //
ack
  // stuff
}
```

```
// user button connected to PF4 (increment counter on falling edge)

#include <stdint.h>
#include "inc/tm4c123gh6pm.h"

void DisableInterrupts(void); // Disable interrupts
void EnableInterrupts(void);  // Enable interrupts
long StartCritical (void);    // previous I bit, disable interrupts
void EndCritical(long sr);    // restore I bit to previous value
void WaitForInterrupt(void);  // low power mode

// global variable visible in Watch window of debugger
// increments at least once per button press
volatile uint32_t FallingEdges = 0;
void EdgeCounter_Init(void){
  SYSCTL_RCGCGPIO_R |= 0x00000020; // (a) activate clock for port F
  FallingEdges = 0;             // (b) initialize counter
  GPIO_PORTF_DIR_R |= 0x0E;   // make PF3-1 output (PF3-1 built-in LEDs)
  GPIO_PORTF_AFSEL_R &= ~0x0E;     // disable alt funct on PF3-1
  GPIO_PORTF_DEN_R |= 0x0E;       // enable digital I/O on PF3-1
```

```
 // configure PF3-1 as GPIO

GPIO_PORTF_DIR_R &= ~0x10;    // (c) make PF4 in (built-in button)
  GPIO_PORTF_AFSEL_R &= ~0x10;  //     disable alt funct on PF4
  GPIO_PORTF_DEN_R |= 0x10;     //     enable digital I/O on PF4
  GPIO_PORTF_PCTL_R &= ~0x000F0000; // configure PF4 as GPIO
  GPIO_PORTF_AMSEL_R = 0;       //  disable analog functionality on PF
  GPIO_PORTF_PUR_R |= 0x10;     //     enable weak pull-up on PF4
  GPIO_PORTF_IS_R &= ~0x10;     // (d) PF4 is edge-sensitive
  GPIO_PORTF_IBE_R &= ~0x10;    //     PF4 is not both edges
  GPIO_PORTF_IEV_R &= ~0x10;    //     PF4 falling edge event
  GPIO_PORTF_ICR_R = 0x10;      // (e) clear flag4
  GPIO_PORTF_IM_R |= 0x10;      // (f) arm interrupt on PF4 *** No
IME bit as mentioned in Book ***
  NVIC_PRI7_R = (NVIC_PRI7_R&0xFF00FFFF)|0x00A00000; // (g)
priority 5
  NVIC_EN0_R = 0x40000000;      // (h) enable interrupt 30 in NVIC
  EnableInterrupts();           // (i) Clears the I bit
}
```

```
void GPIOPortF_Handler(void){
  GPIO_PORTF_ICR_R = 0x10;    // acknowledge flag4
  FallingEdges = FallingEdges + 1;
 GPIO_PORTF_DATA_R^= 0x02;           // turn on LED
}

//debug code
int main(void){
  EdgeCounter_Init();    // initialize GPIO Port F interrupt
  while(1){
    WaitForInterrupt();
  }
}
```