**C12 Interrupts**

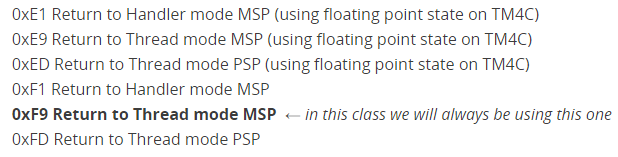
**C12.1 Interrupt Concepts**

Five conditions must be true for an interrupt to be generated:

1. Device arm
2. NCIV enable, (not needed for SysTick) | NVIC\_EN0 and enable one of the ports
3. Global enable. Bit 0 of the special register PRIMASK is the interrupt mask bit. If the bit is 1, most interrupts and exceptions are not allowed; we define this as disabled. 0 would enable it.
4. Interrupt priority level must be higher than current level executing. The BASEPRI register sets the priority. For example, if BASEPRI is 3, then requests with level 0, 1 and 2 can interrupt while requests at level 3 or higher will be postponed. If the BASEPRI is zero, all interrupts are allowed and the BASEPRI register is not active.
5. Hardware event trigger. For example, a hardware trigger is the count flag in the NVIC\_ST\_CTRL\_R register which is set periodically by SysTick.

After an interrupt, these steps occur, we call these steps the context switch:

1. Current instruction is finished
2. Current program is suspended and eight registers pushed on the stack (R0, R1, R2, R3, R12, LR, PC, PSR). This makes a record of where we were. The other registers R4-R11, these are not pushed to the stack because the ISR we write will not use them.
3. LR is set to 0XFFFFFFF9. This specific value signifies an interrupt service routine (ISR) is being run. If floating point registers were pushed it would be 0XFFFFFFe9. The bottom 8 bits signify how to return from interrupt:



In the case of a nested interrupt, the LR is set to 0xFFFFFF1, so it knows to return to handler mode.

1. IPSR is set to the interrupt number. It just contains the interrupt number.
2. PC is loaded with interrupt vector. The program counter (PC) is loaded with the ISR routine (called the interrupt vector).
3. To go back to the main control, BX LR will be called. This instruction pops the 8 registers out of the stack and return back to where the main is.

If the trigger flag is set but the interrupts are disabled (I=1), the flag is disarmed but it is held pending.

There are three essential mechanisms that are needed to utilize interrupts:

1. Ability for the hardware to request interrupt action from computer
2. Ability for the computer to determine the source. A vector interrupt system employs a separate interrupt vector address for each device. A polled interrupt system uses the interrupt software to poll each device and look for the device that requested the interrupt, in otherwords, some devices might share the same vector (for example, the GPIO ports share the same trigger flag, so if multiple pins are armed, the shared ISR must poll them.
3. Ability for the computer to acknowledge the interrupt.

**Definitions**

**Interrupt**: A hardware event (trigger) that is asynchronous with the current software execution and can suspend a task.

**Arm/Disarm**: Arming a device means to allow the hardware trigger to interrupt. Each trigger has its own arm bit.

**Enable/Disable**: Enable means to allow interrupts at this time. To disable interrupts, we set the I bit in PRIMASK or call EnableInterrupts()/DisableInterrupts().

In the example below, we see definitions in the Startup.s that allow the software to enable or disable interrupts. It can be called from either assembly or C code.

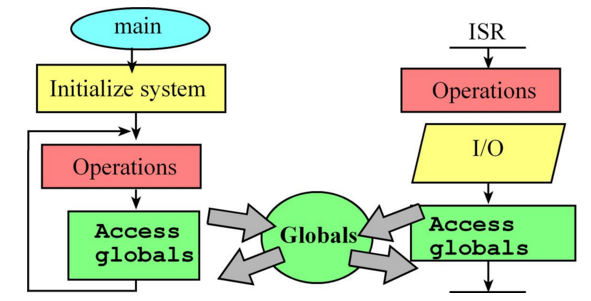
;\*\*\*\*\*\*\*\*\*\*\* DisableInterrupts \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
; disable interrupts                 
; inputs:  none          
; outputs: none                 
DisableInterrupts  CPSID  I     ;set I=1  
                   BX     LR

;\*\*\*\*\*\*\*\*\*\*\* EnableInterrupts \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
; enable interrupts                 
; inputs:  none          
; outputs: none                 
EnableInterrupts   CPSIE  I    ;set I=0  
                   BX     LR

;\*\*\*\*\*\*\*\*\*\*\* WaitForInterrupt \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
; go to low power mode while waiting for the next interrupt  
; inputs:  none  
; outputs: none  
WaitForInterrupt  
        WFI  
        BX     LR  
**Context Switch**: Occurs automatically in hardware as the context is switched from foreground thread to a background thread. We can also have a context switch from a lower priority ISR to a higher priority ISR.

**Acknowledgement**: When you clear a trigger flag. Each trigger flag has a specific action software must perform to clear that flag.

**Interrupt Service Routine**: Is the software module that is executed when the hardware requests an interrupt. The ISR is responsible for clearing the trigger flag. When the ISR is done, it executes BX LR. Because LR contains the special value (ex 0Xffffff9), this instruction pops the 8 registers from stack and returns control to the main program.



**Latency**: For inputs, it is the time between when new input is available and the time when the software reads the input data. For outputs, it is the time between when the output device is idle and the time when the software writes new data.

**Bandwidth**: Amount of data/sec being processed.

**Atomic** **Operation**: Sequence that once started will always finish and cannot be interrupted. To make an instruction atomic, disable the interrupts

**C12.2 Inter-thread Communication and Synchronization**

For regular function calls, we use the registers and stack to pass parameters, but the interrupt threads have logically separate registers and stack. Thus, all parameter passing must occur through global memory, one cannot pass data from the main program to the ISR using registers or the stack.

We have the main thread and many background ISR threads. We can communicate using global memory which can be:

1. Flag: Set by the background thread to tell the main thread we need an interrupt. The main thread would poll for the flag. The main will need to acknowledge the flag when done.
2. Mailbox: There is a flag and a data field in the mailbox. Ie. The ISR gets some UART data and the UART\_data is written into the data field of the mailbox. It then sets the flag to 1. The main thread is polling for the flag, when it sees that flag = 1, it gets the data form the mailbox. The main will need to acknowledge the flag when done. Here data flows from ISR to main but it can flow the other way. For example, the main gets the data and sets the flag, then the ISR does something with the data.
3. FIFO Buffer (Pipeline): ISR reads data and puts the data into a fifo. The main thread calls Get() to get the data in the FIFO.
4. Binary Semaphore: Use it to wait and signal either the main or ISR thread about new data.

**C12.3 NVIC on the ARM Cortex-M Processor**

The Nested Vector Interrupt Controller (NVIC) controls the interrupts on the Cortex-M. Each exception (resets, software/hardware interrupts) has an associated 32-bit vector that points to the memory location where the ISR that handles the exception is located. These vectors are stored in ROM.

The vectors are defined in Startup.s. DCD is an assembler pseudo-op that defines a 32-bit constant. ROM location 0x0000.0000 has the initial stack pointer, and location 0x0000.0004 contains the initial program counter, which is also called the reset vector. This points to a function called the reset handler which is the first thing executed following reset. In C, the reset handler initializes global variables and then calls your main() program.

Here is the software syntax to set the interrupt vectors for the TM4C:

 EXPORT  \_\_Vectors  
\_\_Vectors                        ; address    interrupt  
   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  
   DCD     SSI0\_Handler          ; 0x0000005C SSI  
   DCD     I2C0\_Handler          ; 0x00000060 I2C  
   DCD     PWM0Fault\_Handler     ; 0x00000064 PWM Fault  
   DCD     PWM0Generator0\_Handler  ; 0x00000068 PWM 0 Generator 0  
   DCD     PWM0Generator1\_Handler  ; 0x0000006C PWM 0 Generator 1  
   DCD     PWM0Generator2\_Handler  ; 0x00000070 PWM 0 Generator 2  
   DCD     Quadrature0\_Handler   ; 0x00000074 Quadrature Encoder 0  
   DCD     ADC0Seq0\_Handler      ; 0x00000078 ADC0 Sequence 0  
   DCD     ADC0Seq1\_Handler      ; 0x0000007C ADC0 Sequence 1  
   DCD     ADC0Seq2\_Handler      ; 0x00000080 ADC0 Sequence 2  
   DCD     ADC0Seq3\_Handler      ; 0x00000084 ADC0 Sequence 3  
   DCD     WDT\_Handler           ; 0x00000088 Watchdog  
   DCD     Timer0A\_Handler       ; 0x0000008C Timer 0 subtimer A  
   DCD     Timer0B\_Handler       ; 0x00000090 Timer 0 subtimer B  
   DCD     Timer1A\_Handler       ; 0x00000094 Timer 1 subtimer A  
   DCD     Timer1B\_Handler       ; 0x00000098 Timer 1 subtimer B  
   DCD     Timer2A\_Handler       ; 0x0000009C Timer 2 subtimer A  
   DCD     Timer2B\_Handler       ; 0x000000A0 Timer 2 subtimer B  
   DCD     Comp0\_Handler         ; 0x000000A4 Analog Comp 0  
   DCD     Comp1\_Handler         ; 0x000000A8 Analog Comp 1  
   DCD     Comp2\_Handler         ; 0x000000AC Analog Comp 2  
   DCD     SysCtl\_Handler        ; 0x000000B0 System Control  
   DCD     FlashCtl\_Handler      ; 0x000000B4 Flash Control  
   DCD     GPIOPortF\_Handler     ; 0x000000B8 GPIO Port F

A typical ISR function looks like this:

void GPIOPortF\_Handler(void){  
  GPIO\_PORTF\_ICR\_R = 0x10; // ack, clear interrupt flag4  
  // stuff  
}

There are five enable registers **NVIC\_EN0\_R** through **NVIC\_EN4\_R**. The **NVIC\_EN0\_R** controls the IRQ (32 bits) numbers 0 to 31 (interrupt numbers 16-47). For example, UART0 is IRQ=5, so to enable UART0 we set bit 5 in **NVIC\_EN0\_R**. The 32 bits in **NVIC\_EN1\_R** control the IRQ numbers 32-63 (interrupt numbers 48-79. Writing 0 to the **EN** registers has no effect. Instead to disable interrupts, write ones to the corresponding bit in **NVIC\_DIS0\_R** through **NVIC\_DIS4\_R**.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Address | 31 | 30 | 29-7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Name |
| 0xE000E100 | G | F | … | UART1 | UART0 | E | D | C | B | A | NVIC\_EN0\_R |
| 0xE000E104 |  |  | … |  |  |  |  |  | UART2 | H | NVIC\_EN1\_R |

*Table 12.3. Some of the TM4C NVIC interrupt enable registers. There are five such registers defining 139 interrupt enable bits.*

Each interrupt has an interrupt number and an IRQ number. Let x be the interrupt number and let n be the IRQ number. The interrupt number is:

The interrupt number defines the position in the vector table. The vector address is:

For example, SysTick is interrupt number 15 and IRQ number -1. Therefore, the SysTick vector is in memory location 60, which is ROM location 0x0000003C.

For those interrupts with IRQ numbers greater and or equal to 0, we can find its priority register by dividing n by 4. Let m=n/4 (integer divide). The priority register number will be m. We can find the bit field for that IRQ by looking at the remainder, p = n%4, where p = 0, 1, 2, or 3. The three bits will be 8\*p+7, 8\*p+6, and 8\*p+5.

For example, Timer3A is interrupt number 51 and IRQ number 35. n=35, so m=35/4 = 8. p=35%4=3. So the Timer3A priority register is 8 (NVIC\_PRI8\_R), and the three priority bits are be 8\*3+7, 8\*3+6, and 8\*3+5, which are 31, 30, and 29.

For those interrupts with IRQ numbers greater and or equal to 0, we can find its enable register by dividing n by 32. Let a = n/32 (integer divide). The enable register number will be a. We can find the bit field for that IRQ by looking the remainder, b = n%32, where b = 0, 1, ... or 31. The one bit will be used to enable the interrupt.

Again, Timer3A is interrupt number 51 and IRQ number 35. n=35, so a=35/32 = 1. b=35%32=3. So the Timer3A enable register is 1 (NVIC\_EN1\_R), and the enable bit is bit 3.

**Various Interrupt Vectors**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Vector address | Number | IRQ | ISR name in **Startup.s** | NVIC | Priority bits |
| 0x00000038 | 14 | -2 | **PendSV\_Handler** | **NVIC\_SYS\_PRI3\_R** | 23 – 21 |
| 0x0000003C | 15 | -1 | **SysTick\_Handler** | **NVIC\_SYS\_PRI3\_R** | 31 – 29 |
| 0x00000040 | 16 | 0 | **GPIOPortA\_Handler** | **NVIC\_PRI0\_R** | 7 – 5 |
| 0x00000044 | 17 | 1 | **GPIOPortB\_Handler** | **NVIC\_PRI0\_R** | 15 – 13 |
| 0x00000048 | 18 | 2 | **GPIOPortC\_Handler** | **NVIC\_PRI0\_R** | 23 – 21 |
| 0x0000004C | 19 | 3 | **GPIOPortD\_Handler** | **NVIC\_PRI0\_R** | 31 – 29 |
| 0x00000050 | 20 | 4 | **GPIOPortE\_Handler** | **NVIC\_PRI1\_R** | 7 – 5 |
| 0x00000054 | 21 | 5 | **UART0\_Handler** | **NVIC\_PRI1\_R** | 15 – 13 |
| 0x00000058 | 22 | 6 | **UART1\_Handler** | **NVIC\_PRI1\_R** | 23 – 21 |
| 0x0000005C | 23 | 7 | **SSI0\_Handler** | **NVIC\_PRI1\_R** | 31 – 29 |
| 0x00000060 | 24 | 8 | **I2C0\_Handler** | **NVIC\_PRI2\_R** | 7 – 5 |
| 0x00000064 | 25 | 9 | **PWM0Fault\_Handler** | **NVIC\_PRI2\_R** | 15 – 13 |
| 0x00000068 | 26 | 10 | **PWM0\_Handler** | **NVIC\_PRI2\_R** | 23 – 21 |
| 0x0000006C | 27 | 11 | **PWM1\_Handler** | **NVIC\_PRI2\_R** | 31 – 29 |
| 0x00000070 | 28 | 12 | **PWM2\_Handler** | **NVIC\_PRI3\_R** | 7 – 5 |
| 0x00000074 | 29 | 13 | **Quadrature0\_Handler** | **NVIC\_PRI3\_R** | 15 – 13 |
| 0x00000078 | 30 | 14 | **ADC0\_Handler** | **NVIC\_PRI3\_R** | 23 – 21 |
| 0x0000007C | 31 | 15 | **ADC1\_Handler** | **NVIC\_PRI3\_R** | 31 – 29 |
| 0x00000080 | 32 | 16 | **ADC2\_Handler** | **NVIC\_PRI4\_R** | 7 – 5 |
| 0x00000084 | 33 | 17 | **ADC3\_Handler** | **NVIC\_PRI4\_R** | 15 – 13 |
| 0x00000088 | 34 | 18 | **WDT\_Handler** | **NVIC\_PRI4\_R** | 23 – 21 |
| 0x0000008C | 35 | 19 | **Timer0A\_Handler** | **NVIC\_PRI4\_R** | 31 – 29 |
| 0x00000090 | 36 | 20 | **Timer0B\_Handler** | **NVIC\_PRI5\_R** | 7 – 5 |
| 0x00000094 | 37 | 21 | **Timer1A\_Handler** | **NVIC\_PRI5\_R** | 15 – 13 |
| 0x00000098 | 38 | 22 | **Timer1B\_Handler** | **NVIC\_PRI5\_R** | 23 – 21 |
| 0x0000009C | 39 | 23 | **Timer2A\_Handler** | **NVIC\_PRI5\_R** | 31 – 29 |
| 0x000000A0 | 40 | 24 | **Timer2B\_Handler** | **NVIC\_PRI6\_R** | 7 – 5 |
| 0x000000A4 | 41 | 25 | **Comp0\_Handler** | **NVIC\_PRI6\_R** | 15 – 13 |
| 0x000000A8 | 42 | 26 | **Comp1\_Handler** | **NVIC\_PRI6\_R** | 23 – 21 |
| 0x000000AC | 43 | 27 | **Comp2\_Handler** | **NVIC\_PRI6\_R** | 31 – 29 |
| 0x000000B0 | 44 | 28 | **SysCtl\_Handler** | **NVIC\_PRI7\_R** | 7 – 5 |
| 0x000000B4 | 45 | 29 | **FlashCtl\_Handler** | **NVIC\_PRI7\_R** | 15 – 13 |
| 0x000000B8 | 46 | 30 | **GPIOPortF\_Handler** | **NVIC\_PRI7\_R** | 23 – 21 |
| 0x000000BC | 47 | 31 | **GPIOPortG\_Handler** | **NVIC\_PRI7\_R** | 31 – 29 |
| 0x000000C0 | 48 | 32 | **GPIOPortH\_Handler** | **NVIC\_PRI8\_R** | 7 – 5 |
| 0x000000C4 | 49 | 33 | **UART2\_Handler** | **NVIC\_PRI8\_R** | 15 – 13 |

*Table 12.1. Some of the interrupt vectors for the TM4C. The TM4C123 has over 100 possible interrupt sources.*

**Definitions**

Thread Mode: Running the main program, ie. IPSR register is 0.

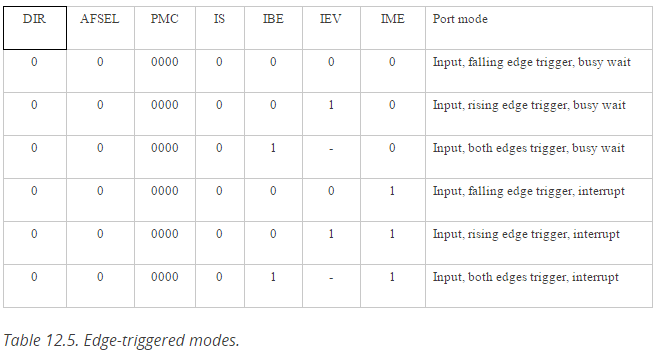
Handler Mode: Running an ISR which means the IPSR value is nonzero.

**C12.4 Edge-Triggered Interrupts**

Each of the digital I/O pins on the TM4C family can be configured for edge triggering. Edge triggering is where the interface is configured to set a flag on the rising (or falling) edge of the input.

To configure an edge triggered pin, we first enable the clock on the port and configure the pin as a regular digital input. Clearing the IS (Interrupt Sense) bit configures the bit for edge triggering. Next, write to the IBE (Interrupt Both Edges) and the IEV (Interrupt Event) bits to define the active edge. We can trigger on the rising, falling, or both edges. We clear the IME (Interrupt Mask Enable) bits if we are using busy-wait synchronization and we set the IME bits to use interrupt synchronization.

So for example, make it an input, disable (set to 0) AFSEL (alt func), PCTL, AMSEL (analog). Set DIR to 0. IS to 0 which means edge. IBE to 0 because we don’t want both. IEV 0 would give falling edge triggered. IME is 1, which says once you see the trigger, cause an interrupt.



Interrupts are vectors, vectors are addresses which tells the systems where to get the software to execute.

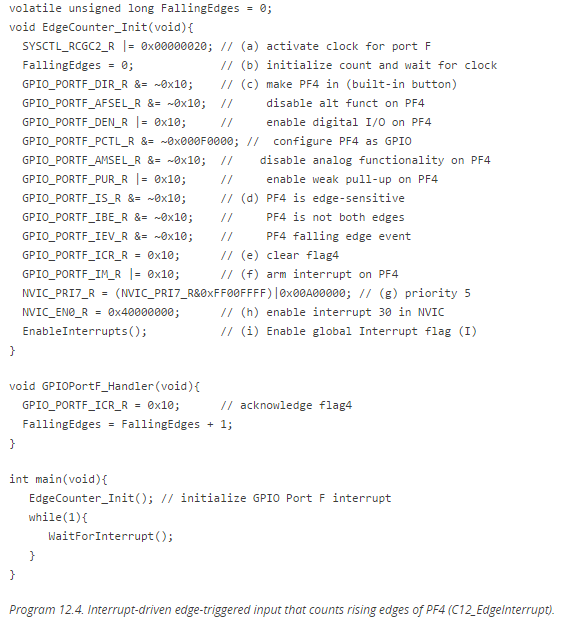
The hardware sets an RIS (Raw Interrupt Status) bit (Called the trigger) and the software clears it (called the acknowledgement). To acknowledge, write a 1 to the ICR bit to clear the trigger. The RIS is set every time you have a falling or rising edge depending on how it was configured.

In particular, five conditions must be simultaneously true for an edge-triggered interrupt to be requested:

1. The trigger flag bit is set (RIS)
2. The arm bit is set (IME)
3. The level of the edge-triggered interrupt must be less than BASEPRI
4. The edge-triggered interrupt must be enable din the NVIC\_EN0\_R
5. The 1 bit, bi 0 of the special register PRIMASK, is 0

**Example**

We will begin with a simple example that counts the number of rising edges on Port F bit 4 (Program 12.4). The initialization requires many steps. (a) The clock for the port must be enabled. (b) The global variables should be initialized. (c) The appropriate pins must be enabled as inputs. (d) We must specify whether to trigger on the rise, the fall, or both edges. In this case we will trigger on the rise of PF4. (e) It is good design to clear the trigger flag during initialization so that the first interrupt occurs due to the first rising edge after the initialization has been run. We do not wish to count a rising edge that might have occurred during the power up phase of the system. (f) We arm the edge-trigger by setting the corresponding bits in the IM register. (g) We establish the priority of Port F by setting bits 23 – 21 in the NVIC\_PRI7\_R register as listed in Table 9.2. We activate Port F interrupts in the NVIC by setting bit 30 in the NVIC\_EN0\_R register, Table 12.3. There is no need to unlock PF4.

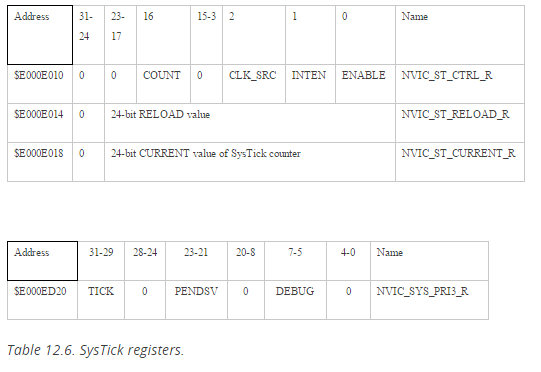


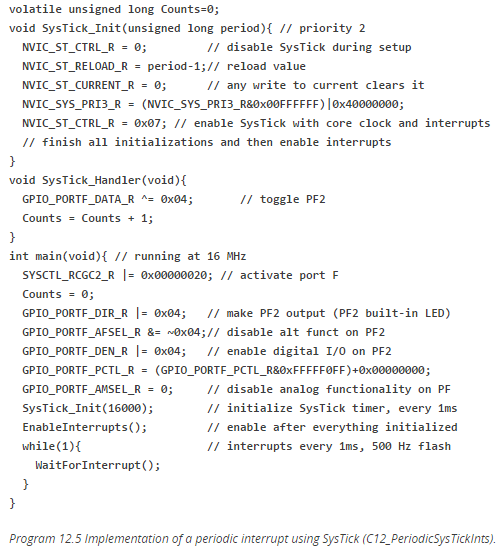
**C12.5 SysTick Periodic Interrupts**

Using the SysTick timer we can periodically interrupt the microcontroller to perform a task.

First, we clear the ENABLE bit to turn off SysTick during initialization. Second, we set the RELOAD register. Third, we write any value to NVIC\_ST\_CURRENT\_R to clear the counter. Lastly, we write the desired mode to the control register, NVIC\_ST\_CTRL\_R. We must set CLK\_SRC=1, because CLK\_SRC=0 external clock mode is not implemented on the TM4C microcontroller family. We set INTEN to enable interrupts. We need to set the ENABLE bit so the counter will run. In summary, we will write a 7 to the NVIC\_ST\_CTRL\_R register.

We establish the priority of the SysTick interrupts using the TICK field in the NVIC\_SYS\_PRI3\_R register (bits 31-29). When the CURRENT value counts down from 1 to 0, the COUNT flag is set. On the next clock, the CURRENT is loaded with the RELOAD value. In this way, the SysTick counter (CURRENT) is continuously decrementing.





Example calculation for RELOAD:

