CHAPTER 11

Interrupts

This chapter introduces the basic concepts of *interrupts* and *interrupt service routines*. We illustrate the programming of interrupts using the system timer (`SysTick`), external interrupts (`EXTI`), and software interrupts (`SVC`).

11.1 Introduction to Interrupts

An *interrupt* leverages a combination of software and hardware to force the processor to stop its current activity and begin to execute a particular piece of code called an ***interrupt service routine*** (ISR). An ISR responds to a specific event generated by either hardware or software. When an ISR completes, the processor automatically resumes the activity that had been halted. The halted process continues as if nothing had happened.

> ***An interrupt is simply a hardware-invoked function call.***

Interrupts are widely used to respond to both internal and external hardware requests efficiently. For example, interrupts can inform a program of some timely external events (such as pushing a button and receiving a message in a communication port). Interrupts allow a processor to gracefully shutdown when there are critical errors (such as memory access violations, and detection of undefined instructions).

Interrupts also allow a processor to perform multiple tasks simultaneously. At any given time, the microcontroller is serving only one program activity. However, interrupts enable the processor to serve multiple computation tasks alternately in a multiplexing fashion. Multiple tasks can be handled in a preemptive or non-preemptive manner.

- In the ***preemptive*** scenario, if a new task is more urgent than the current task, this new task can stop the current one without requiring any cooperation. The new

task will take over control of the processor. The processor resumes the old task after the new task completes.

- In the ***non-preemptive*** scenario, a new task cannot stop the current task until the current task voluntarily gives up control of the processor. A non-preemptive system often relies on the system timer, described in Chapter 23, to serve multiple tasks periodically in a round-robin fashion.

Interrupts enable a microcontroller to respond to human inputs or latency-sensitive events rapidly. An alternative to interrupts is busy-waiting or periodic ***polling***. In the polling scheme, the processor continually queries the I/O devices to check whether a specific event has happened, and handles the event. The latency of detecting the event is determined by the polling period. In the interrupt scheme, the processor provides a hardware mechanism that allows an internal or external device to generate a signal to immediately inform the processor of events that have occurred.

We use a telephone as an example to compare polling efficiency and interrupt efficiency. Suppose you are expecting a call. In the polling scheme, you pick up your telephone every 10 seconds to check whether there is anyone on the line calling you. In the interrupt scheme, you continue to perform whatever tasks you are supposed to complete while waiting for the telephone to ring. When the telephone finally rings (*i.e.*, you are interrupted), you can stop your current task and answer the phone. As you can see from this analogy, polling is much less efficient than using interrupts. With the polling scheme, you waste time that could be spent on other tasks picking up the telephone repeatedly without successfully receiving any calls.

11.2 Interrupt Numbers

Cortex-M processors support up to 256 types of interrupts. Each interrupt type, excluding the *reset* interrupt, is identified by a unique number, ranging from -15 to 240. Interrupt numbers are defined by ARM and chip manufacturers collectively. These numbers are fixed and software cannot re-define them. Interrupt numbers are divided into two groups.

- The first 16 interrupts are system interrupts, also called ***system exceptions***. Exceptions are the interrupts that come from the processor core. These interrupt numbers are defined by ARM. Specifically, the ARM CMSIS library defines all system exceptions by using negative values. CMSIS stands for Cortex Microcontroller Software Interface Standard.

- The remaining 240 interrupts are ***peripheral interrupts***, also called non-system exceptions. The peripheral interrupt numbers start at 0. Peripheral interrupts are defined by chip manufacturers. The total number of peripheral interrupts supported varies among chips.

This numbering scheme allows software to distinguish system exceptions and peripheral interrupts easily. Table 11-1 shows the definition of all interrupt numbers for STM32L4. Although Cortex-M processors support 256 interrupts, not all interrupt numbers are used on STM32L4.

Cortex-M4 Processor Exceptions Numbers		STM32L4 specific Interrupt Numbers															
-14	Non-maskable interrupt	0	WWDG	16	DMA1_CH6	32	I2C1_ER	48	FMC	64	COMP	80	RNG				
-13	Hard fault	1	PVD	17	DMA1_CH7	33	I2C2_EV	49	SDMMC1	65	LPTIM1	81	FPU				
-12	Memory management	2	TAMPER_STAMP	18	ADC1_ADC2	34	I2C2_ER	50	TIM5	66	LPTIM2						
-11	Bus fault	3	RTC_WKUP	19	CAN1_TX	35	SPI1	51	SPI3	67	OTG_FS						
-10	Usage fault	4	FLASH	20	CAN1_RX0	36	SPI2	52	UART4	68	DMA2_Channel6						
-5	Supervisor call (SVCall)	5	RCC	21	CAN1_RX1	37	USART1	53	UART5	69	DMA2_Channel7						
-4	Debug monitor	6	EXTI0	22	CAN1_SCE	38	USART2	54	TIM6_DAC	70	LPUART1						
-2	PendSV	7	EXTI1	23	EXTI9_5	39	USART3	55	TIM7	71	QUADSPI						
-1	SysTick	8	EXTI2	24	TIM1_BRK	40	EXTI15_10	56	DMA2_Channel1	72	I2C3_EV						
		9	EXTI3	25	TIM1_UP	41	RTC_Alarm	57	DMA2_Channel2	73	I2C3_ER						
		10	EXTI4	26	TIM1_TRG	42	DFSDM3	58	DMA2_Channel3	74	SAI1						
		11	DMA1_CH1	27	TIM1_CC	43	TIM8_BRK	59	DMA2_Channel4	75	SAI2						
		12	DMA1_CH2	28	TIM2	44	TIM8_UP	60	DMA2_Channel5	76	SWPMI1						
		13	DMA1_CH3	29	TIM3	45	TIM8_TRG	61	DFSDM0	77	TSC						
		14	DMA1_CH4	30	TIM4	46	TIM8_CC	62	DFSDM1	78	LCD						
		15	DMA1_CH5	31	I2C1_EV	47	ADC3	63	DFSDM2	79							

Table 11-1. CMSIS Definition of interrupt numbers for STM32L4

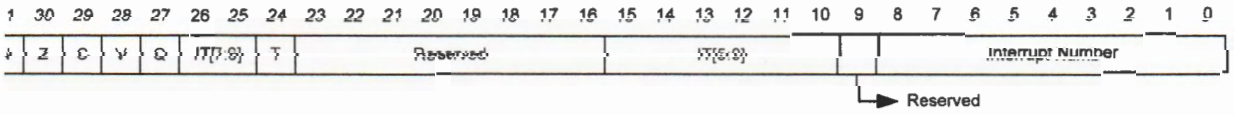


Figure 11-1. Program status register

When an interrupt is processed, the interrupt number is stored in the program status register (PSR), as shown in Figure 11-1. However, ARM Cortex-M does not store interrupt numbers in two's complement. Instead, the interrupt number in PSR adds a positive offset of 15 to the CMSIS interrupt number.

$$Interrupt\ number\ in\ PSR = CMSIS\ interrupt\ number + 15$$

In the rest of the book, unless specified otherwise, interrupt numbers are the ones defined by CMSIS. As will be introduced later, each interrupt number is used as an index into the interrupt vector table to search for the starting memory address of its corresponding interrupt service routine.

11.3 Interrupt Service Routines

An interrupt service routine (ISR), also called an ***interrupt handler***, is a special subroutine that hardware invokes automatically in response to an interrupt. Each ISR has a default implementation in the system startup code (such as the assembly file `startup_stm32xxxx.s`). The default implementation of most ISRs is simply a dead loop, such as the interrupt handler for the system timer shown below.

```
SysTick_Handler  PROC
                 EXPORT  SysTick_Handler  [WEAK]
                 B       .       ; dead loop
                 ENDP
```

Example 11-1. Default implementation of interrupt handler for the system timer (SysTick)

All ISRs are declared as `weak` in the system startup code. The keyword `weak` means that another non-weak subroutine with the same name defined elsewhere can override this one. Example 11-2 gives two implementations in C and assembly, respectively. ISRs do not return any values because they are called by hardware (there is no software caller). Furthermore, ISRs (excluding *SVC_Handler*) do not take any input arguments.

C Code	Assembly Code

`void SysTick_Handler (void) {`
 	
`  ...`
 	
`}`
	`SysTick_Handler  PROC`
	`  EXPORT  SysTick_Handler`
	
	`  ...`
	
	`  ENDP`

Example 11-2. User implementation of interrupt handler for the system timer (SysTick)

The *Reset_Handler* ISR, as shown below, is executed when the processor is reset or powered up. *Reset_Handler* eventually calls the *main* function. For a C program compiled in ARM Keil, *Reset_Handler* calls *__main*, which copies data segments from the instruction memory to the data memory, and then calls the user function *main*.

```
Reset_Handler PROC
              EXPORT  Reset_Handler  [WEAK]
              IMPORT  __main
              ; if your main program is written in assembly, make sure to
              ; to add code here to copy data segments to data memory
              LDR     R0, =__main
              BX      R0
              ENDP
```

Example 11-3. Default implementation of `Reset_Handler`

11.4 Interrupt Vector Table

There is an interrupt service routine (ISR) associated with each type of interrupt. Cortex-M stores the starting memory address of every ISR in a special array called the ***interrupt vector table.*** For a given interrupt number `i` defined in CMSIS, the memory address of its corresponding ISR is located at the $(i + 16)^{th}$ entry in the interrupt vector table. The interrupt vector table is stored at the memory address `0x00000004`. Because each entry in the table represents a memory address, each entry takes four bytes in memory.

> ***An interrupt number is used as an index into the interrupt vector table to locate the corresponding interrupt service routine.***

$$Address\ of\ ISR = InterruptVectorTable[i + 15]$$

For example, the interrupt number of SysTick is `-1`, the memory address of `SysTick_Handler` can be founding by reading the word stored at the following address.

$$Address\ of\ \text{SysTick\_Handler} = 0x00000004 + 4 \times (-1 + 15) = 0x0000003C$$

The interrupt number of `reset` is `-15`. Thus, the memory address of `Reset_Handler` is

$$Address\ of\ \text{Reset\_Handler} = 0x00000004 + 4 \times (-15 + 15) = 0x00000004$$

The following describes the booting process of Cortex-M. When an ARM Cortex processor is turned on or reset, the processor fetches two words located at `0x00000000` and `0x00000004` in memory. The processor uses the word located at `0x00000000` to initialize the main stack pointer (`MSP`), and the other one at `0x00000004` to set up the program counter (`PC`). The word stored at `0x00000004` is the memory address of the *Reset_Handler*() procedure, which is determined by the compiler and link script. Typically, *Reset_Handler* calls *main*(), which is the user's application code. After `PC` is initialized, the program begins execution.

Booting process:

1. **`MSP` ← memory[0];**
2. **`PC` ← memory[4];**

While the very first word in memory stores the memory address used to initialize `MSP`, the following words starting at `0x00000004` represent a vector table. This vector table stores the memory addresses of all interrupt and exception handling routines.

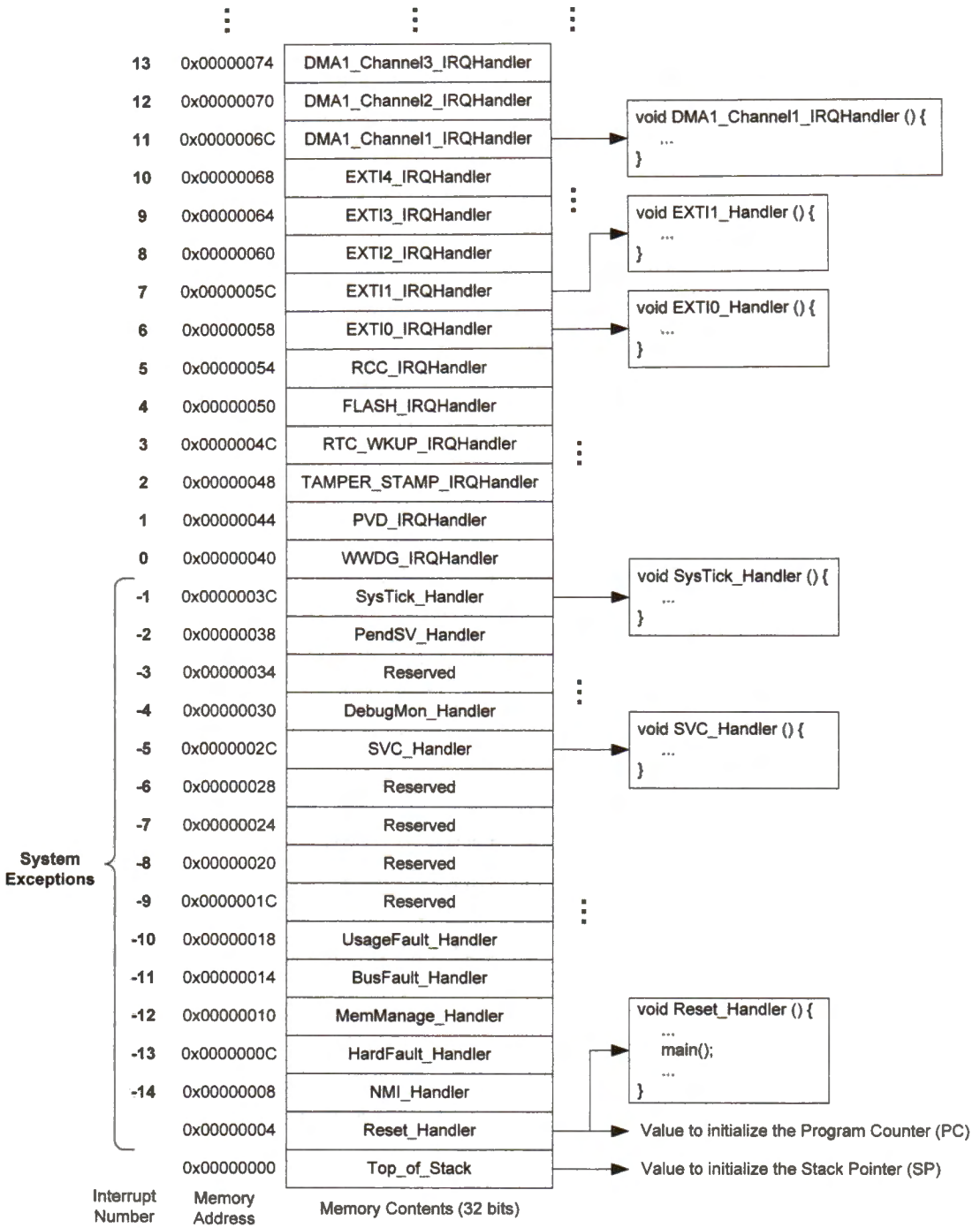


Figure 11-2. Interrupt vector table

Cortex-M processors use the ***nested vectored interrupt controller*** (NVIC) to manage interrupts. NVIC allows applications to enable specific interrupts and set their priority

levels. The processor serves all interrupts based on their priority levels. The processor stops the currently running interrupt handler if a new interrupt with a higher priority occurs. The new interrupt task preempts the current lower-priority task, and the processor resumes the low-priority task when the handler of the new interrupt completes. *A higher value of the interrupt priority number represents a lower priority (or urgency).* The *Reset_Handler*() has top priority, and its priority number is -3.

Among all interrupts defined in the interrupt vector table, the first 15 interrupts (including reserved ones) deal with system abnormalities. These are called ***system exceptions***. The remaining interrupts deal with the activities of peripherals. These are called ***external interrupts***.

Examples of system exceptions include supervisor call interrupts, system timer interrupts, and fault-handling interrupts. Faults include bus faults for prefetch and memory accesses, memory management faults, instruction usage faults, and hard faults.

Examples of external interrupts include ADC interrupts, USB interrupts, and serial communication interrupts (SPI, I²C, and USART). These are used to inform the microcontroller of external events efficiently. Without these interrupts, the microcontroller then would have to resort to inefficient polling to check peripherals repeatedly, wasting precious computation cycles.

The interrupt vector table can be relocated to different regions (SRAM or FLASH). Thus, the processor can boot from different memory devices.

The interrupt vector table is relocatable. While the interrupt vector table is located at the memory address `0x00000004`, this low memory address can be physically re-mapped to different regions, such as on-chip flash memory, on-chip RAM memory, or on-chip ROM memory. This allows the processor to boot from various memory regions. On STM32L4, the memory address `0x00000004` is an alias to `0x08000004` by default, which is in the address space assigned to the on-chip flash memory.

11.5 Interrupt Stacking and Unstacking

When serving an interrupt, the Cortex-M microcontroller performs automatic stacking and unstacking. Chapter 12.4.1.4 discusses automatic stacking and unstacking related to floating-point registers.

- **Interrupt Stacking**. Before executing the interrupt handler, the stacking process automatically pushes eight registers to preserve the running environment. These eight registers include the lowest four registers (`r0`, `r1`, `r2`, and `r3`) and four other registers (`r12`, `LR`, `PSR`, and `PC`).
- **Interrupt Unstacking**. After the interrupt handler completes, the unstacking process automatically pops the values of these eight registers off the stack. This recovers the environment that existed at the time instant immediately before the interrupt handler started to run. At the same time, the processor clears the corresponding active bits in the NVIC status registers.

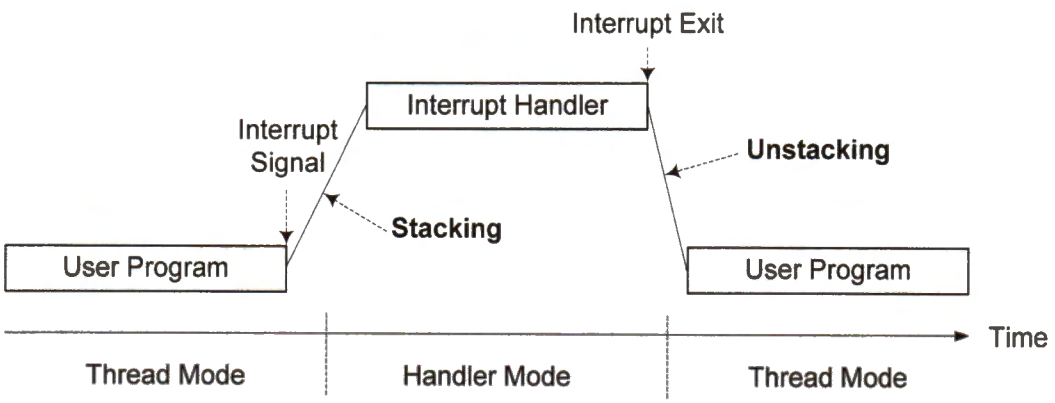


Figure 11-3. Automatic stacking and unstacking for interrupt handler

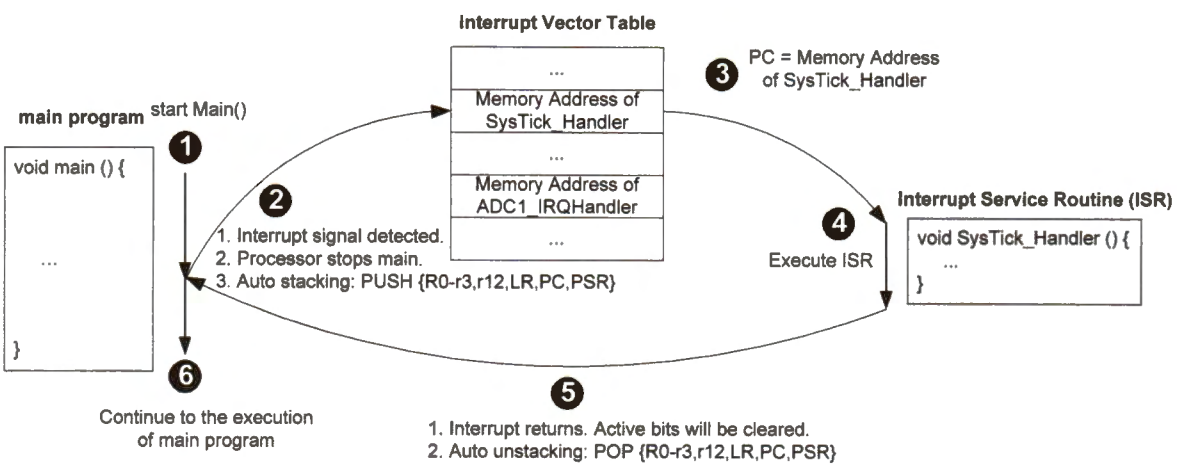


Figure 11-4. Steps of stacking and unstacking for interrupt handler

Since an interrupt may occur at any time, the program counter (`PC`) is preserved during interrupt stacking and then is recovered during interrupt unstacking. Thus, the processor can successfully continue executing the computation that had been interrupted.

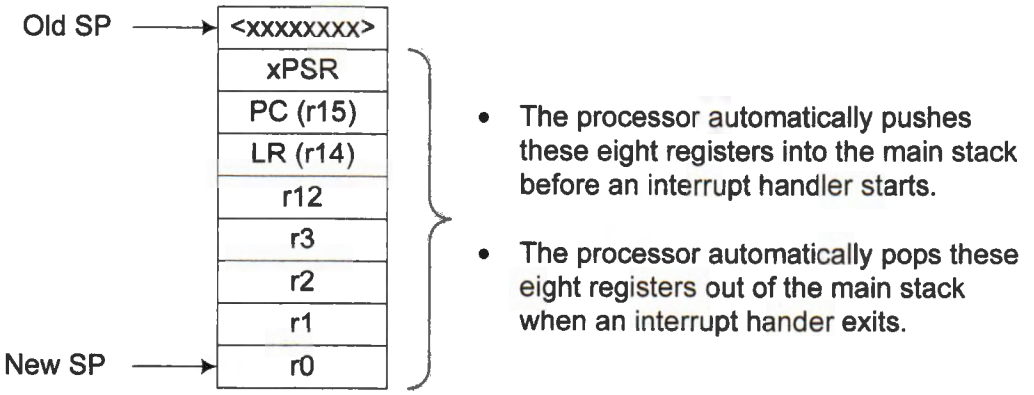


Figure 11-5. Stacking and unstacking when an interrupt handler starts or exits (Assume there is no FPU. See Chapter 12.4.1.4 for stacking and unstacking of floating-point registers.)

The interrupt service routine exits by running "`BX LR`". Note that `LR` in an interrupt service routine has a meaning different from `LR` in a normal subroutine.

- `LR` in a regular subroutine represents the return address to the caller. When a regular subroutine is called, `LR` holds the memory address of the instruction to be processed after exiting the subroutine. The value of `LR` is copied to `PC` when a regular service routine exits.
- `LR` in an interrupt service routine indicates whether the processor uses the main stack or the process stack in the push and pop operations. Since the interrupt service routine preserves and recovers `PC` via stacking and unstacking, the `LR` register is not copied to set `PC` when an interrupt service routine exits. Instead, the `LR` register shall be fixed to a special value (see Chapter 23.1.2), to indicate whether the processor should unstack data out of the main stack (`MSP`) or the process stack (`PSP`). Chapter 23.1 gives a detailed explanation of `LR` for interrupts.

11.6 Nested Vectored Interrupt Controller (NVIC)

The nested vectored interrupt controller (NVIC) is built into Cortex-M cores to manage all interrupts. It offers three key functions:

1. Enable and disable interrupts
2. Configure the preemption priority and sub-priority of a specific interrupt
3. Set and clear the handing bit of a specific interrupt

The Cortex-M microcontroller supports up to 256 interrupts, in which the first 16 interrupts are system exceptions, and the rest 240 interrupts are peripheral interrupts.

The number of interrupts supported by a specific microcontroller, stored in ICTR (interrupt controller type register), differs among different manufacturers.

Each interrupt has six control bits, as listed in Table 11-2.

Interrupt control bit	Corresponding register (32 bits)
Enable bit	Interrupt set enable register (ISER)
Disable bit	Interrupt clear enable register (ICER)
Pending bit	Interrupt set pending register (ISPR)
Un-pending bit	Interrupt clear pending register (ICPR)
Active bit	Interrupt active bit register (IABR)
Software trigger bit	Software trigger interrupt register (STIR)

Table 11-2. Interrupt control bits

Each interrupt has an interrupt number, ranging from 0 to 256 for Cortex-M processors. A Cortex-M processor includes 16 system interrupts (0-15) defined by ARM. System interrupts are also called *system exceptions* or processor exceptions. The rest 240 interrupts are peripheral interrupts (also known as non-system interrupts), which are defined by chip manufacturers. Note that in the Cortex-M code library, the interrupt number of peripherals starts with 0.

Cortex-M defines eight registers for each control bit. For example, there are ISER0, ISER1, ..., and ISER7, which can enable 256 interrupts.

1. ***Enable and disable an interrupt***. Writing an enable bit to 1 can enable the corresponding interrupt. Writing an enable bit to 0 does not turn off the corresponding interrupt. Write a disable bit to 1 can disable the interrupt. Writing a disable bit to 0 has no impacts on the related interrupt. Separating enable bits and disable bits allows us to disable an interrupt conveniently without affecting the other interrupts.
2. ***Pend and clear an interrupt***. If an interrupt occurs, the corresponding pending bit is set if the microcontroller cannot process this interrupt immediately. Writing the clear pending bit to 1 removes the corresponding interrupt from the pending list. When an interrupt is disabled but its pending bit has already been set, this interrupt instance remains active, and it is serviced before it is disabled.
3. ***Trigger an interrupt***. Setting an active bit by software or hardware activates the related interrupt, and the microcontroller starts the corresponding interrupt handler. If software writes a trigger bit of the software trigger interrupt register (STIR) to 1, the related interrupt is also activated. Most system exceptions can only be activated by hardware.

11.6.1 Enable and Disable Peripheral Interrupts

Cortex-M has eight 32-bit Interrupt Set-Enable Registers (`ISER`), `ISER0` – `ISER7`, and eight 32-bit Interrupt Clear-enable Register (`ICER`), `ICER0` – `ICER7`. Writing a bit to 1 in `ISER` and `ICER` enables and disables the corresponding interrupt, respectively.

Figure 11-6 shows `ISER0`, `ISER1` and `ISER2` registers for enabling a peripheral interrupt for STM32L4 processors.

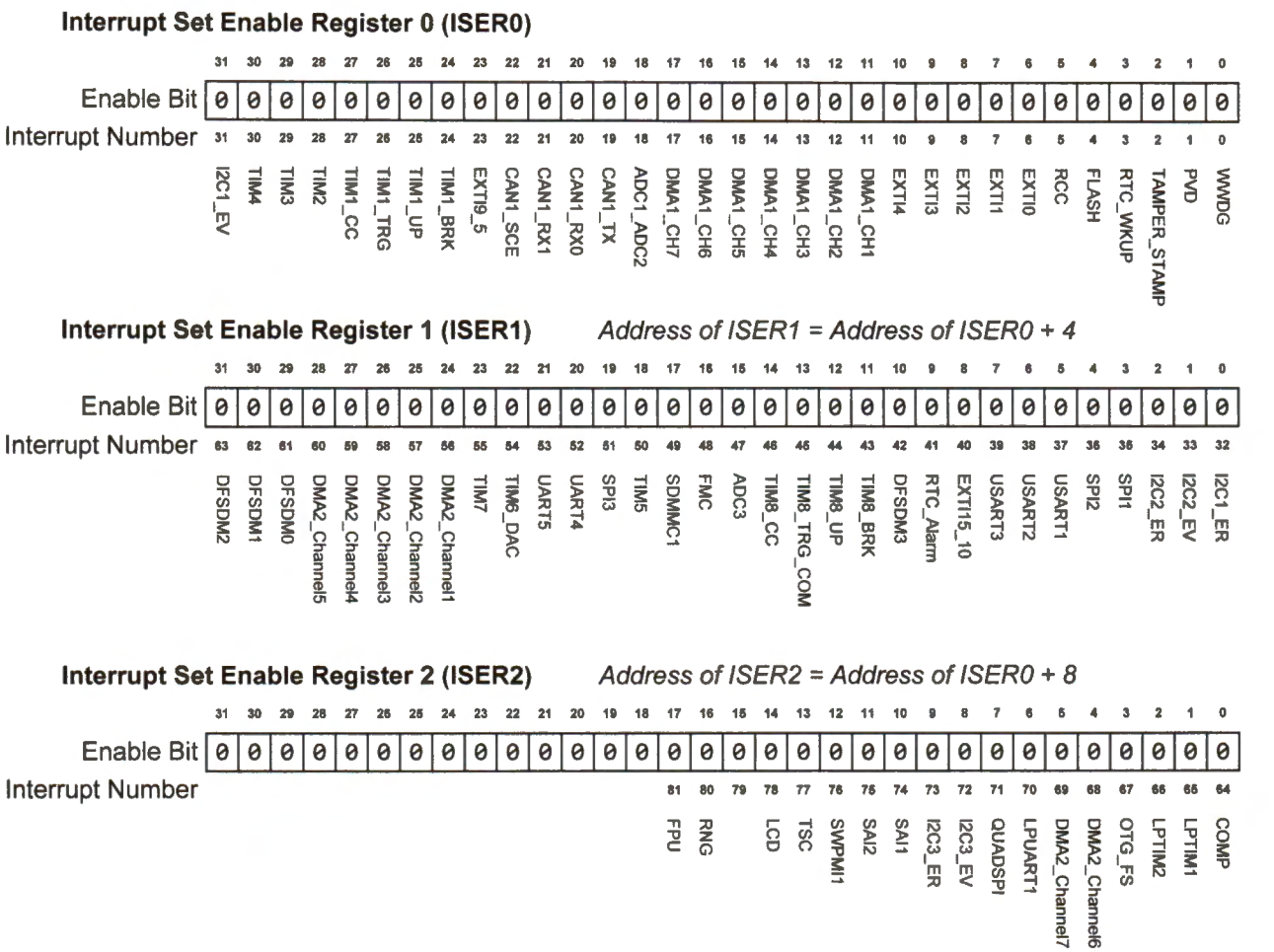


Figure 11-6. Interrupt Set Enable Registers for Peripheral Interrupts in STM32L

Each bit in an `ISER` register can enable one peripheral interrupt. The interrupt number of peripheral interrupts ranges from `0` to `240`. Not all peripheral interrupts are used. For example, STM32L has only 45 peripheral interrupts. Note that all NVIC registers are little endian regardless of whether the processor deploys big endian or little endian.

The following C program can enable a peripheral interrupt whose interrupt number is IRQn. Bit j in register ISER i enables interrupt $IRQn = j + 32 \times i$.

```
WordOffset = IRQn >> 5;                          // Word Offset = IRQn/32
BitOffset = IRQn & 0x1F;                         // Bit Offset = IRQn mod 32
NVIC->ISER[WordOffset]  =  1 << BitOffset;       // Enable interrupt
```

Since each ISER register has 32 bits, the ISER register array index is obtained by shifting right the interrupt number IRQn by five bits. Note that the peripheral interrupt number starts with 0. For example, the interrupt number of Timer 5 is 50. Therefore, the following code can enable the Timer 5 interrupt:

```
NVIC->ISER[1] = 1 << 18;                         // Enable Timer 5 interrupt
```

Similarly, the following C program can disable the interrupt IRQn.

```
WordOffset = IRQn >> 5;                          // WordOffset = IRQn/32
BitOffset = IRQn & 0x1F;                         // BitOffset = IRQn mod 32
NVIC->ICER[WordOffset]  =  1 << BitOffset;       // Disable interrupt
```

```
; Input arguments:
;    r0: interrupt number of a peripheral interrupt
;    r1: 1 = Enable, 0 = Disable
Peripheral_Interrupt_Enable   PROC
   PUSH  {r4, lr}
   AND   r2, r0, #0x1F      ; Bit offset in a word
   MOV   r3, #1
   LSL   r3, r3, r2         ; r3 = 1 << (IRQn & 0x1F)
   LDR   r4, =NVIC_BASE

   CMP   r1, #0             ; Check whether enable or disable
   LDRNE r1, =NVIC_ISER0    ; Enable register base address
   LDREQ r1, =NVIC_ICER0    ; Disable register base address

   ADD   r1, r4, r1         ; r1 = addr. of NVIC->ISER0 or NVIC->ICER0
   LSR   r2, r0, #5         ; Memory offset (in words): IRQn >> 5
   LSL   r2, r2, #2         ; Calculate byte offset
   STR   r3, [r1, r2]       ; Enable/Disable interrupt
   POP   {r4, pc}
   ENDP
```

Example 11-4. Enabling/disabling a peripheral interrupt

Example 11-4 shows the assembly implementation of enabling or disabling a peripheral interrupt. The program takes two arguments: an interrupt number held in register r0 and a Boolean option of enabling and disabling stored in register r1.

The memory address [r1,r2] in the STR instruction is the memory address of NVIC->ISER[i] or NVIC->ICER[i], depending on whether the interrupt is to be enabled or disabled. Because each ISER or ICER register controls 32 interrupts, the program uses the following two instructions to calculate the address:

```
LSR   r2, r0, #5      ; Memory offset in words = IRQn >> 5
LSL   r2, r2, #2      ; Offset in bytes = 4 × Offset in words
```

These two operations are equivalent to:

$$i = floor\left(\frac{Interrupt\ number}{32}\right)$$

$$Byte\ Offset = 4 \times i$$

For example, the program is to enable the interrupt numbered 77. The corresponding register is NVIC->ISER[2], which controls all interrupts from 64 to 95.

$$Byte\ Offset = floor\left(\frac{77}{32}\right) \times 4 = 2 \times 4 = 8$$

Thus, the memory address offset between the NVIC base address and NVIC->ISER[2] is 8. In most computer systems, memory is byte addressable, and thus the offset is in bytes. The registers NVIC->ISER[0] and NVIC->ISER[1] take 4 bytes each in memory. Accordingly, the offset of NVIC->ISER[2] from the NVIC base is 8 bytes.

11.6.2 Interrupt Priority

Priority determines the order of interrupts to be serviced. Each interrupt has an interrupt priority register (IP), which has a width of 8 bits. Each consists of two fields: the preemption priority number and the sub-priority number. *A lower value of a priority number represents a higher priority or a higher urgency.* Priority value 0 has the highest priority (or the highest urgency).

Lower priority value means higher urgency.

Usually, the peripheral interrupts have a positive interrupt level while a microcontroller core interrupt can have negative priority numbers, not changeable by software. When there are multiple pending interrupts, the interrupt that has the lowest interrupt number is serviced first by the processor.

Preemption is a widely-used technique that allows a time-sensitive and urgent computation task to take control of the processor from a relatively less urgent computation task. The preemption priority number defines the priority for preemption.

If the processor receives a new interrupt that has a lower preemption priority number than the preemption priority number of the current interrupt in progress, the current interrupt is stopped, and the processor starts to serve the new interrupt. The preempted interrupt is resumed after the new interrupt handler routine completes.

> *"Don't interrupt me while I'm interrupting."*
>
> Winston Churchill
> Former British Prime Minister

While Cortex-M processors use eight bits to store the priority number, STM32L processors only implement four bits. Thus, the STM32L microcontroller only supports 16 interrupt priority levels, ranging from 0 to 15. For a different Cortex-M processor, the interrupt priority byte might be different.

STM32L processors allow five different schemes to split the four-bit priority number. If we use n bits for the preempt priority number, then the sub-priority number has $4 - n$ bits, where $n = 0, 1, 2, 3,$ or 4. By default, two bits are used for the preempt priority number, and two bits are used for the sub-priority number, as shown in Figure 11-7.

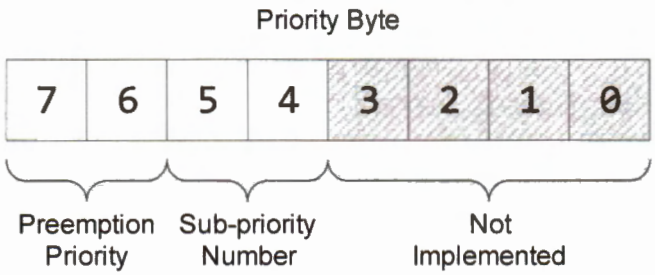


Figure 11-7. Interrupt Priority Byte of STM32L by default

For a system interrupt IRQn, its priority can be set as follows. Note that SHP (System Handler Priority) is defined as a byte array, instead of a word array.

```
typedef struct {
  ...
  volatile uint8_t SHP[12];  // System handler priority array
  ...
} SCB_Type;

// Set the priority of a system interrupt IRQn
SCB->SHP[((uint8_t)IRQn) & 0xF) - 4] = (priority << 4) & 0xFF;
```

SCB is the base memory address of the System Control Block (SCB). SHP is the base memory address of the System Handler Priority registers. Cortex-M has three SHP

registers, as shown in Figure 11-8. For example, the interrupt number of SysTick is -1. The byte offset in the system handler priority array is as follows

```
((uint8_t)IRQn) & 0xF) - 4 = ((uint8_t)-1) & 0xF) - 4
                           = (0xFF & 0xF) - 4
                           = 11
```

Therefore, the following code sets the priority of SysTick to 1:

```
SCB->SHP[11] = (1UL << 4) & 0xFF;
```

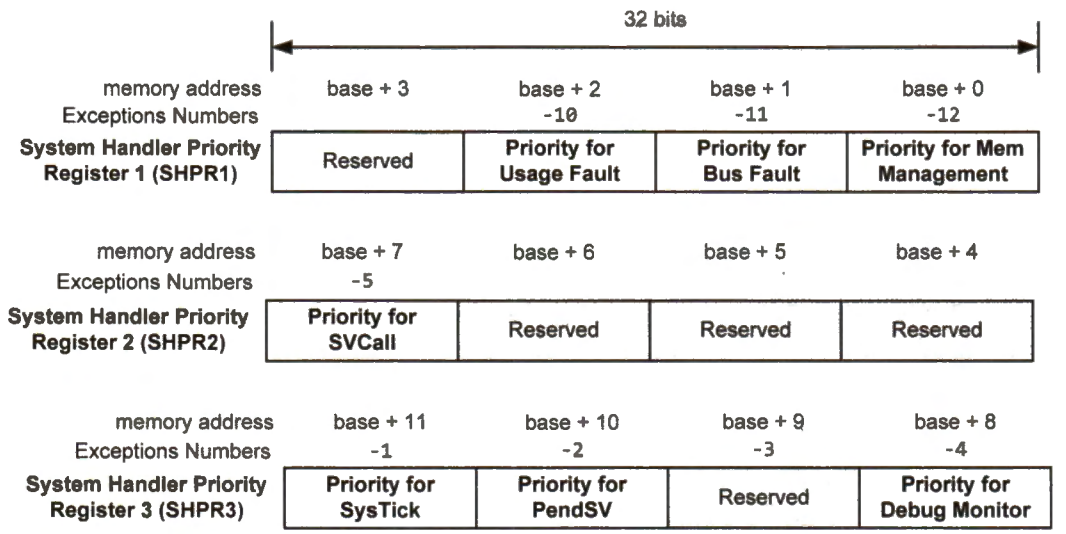


Figure 11-8. Priority registers for system interrupts

For a peripheral interrupt `IRQn`, its priority can be set as follows. Note that the interrupt priority (`IP`) array is defined as an array of bytes, not words.

```
typedef struct {
  ...
  volatile uint8_t  IP[240]; // Interrupt Priority Register
  ...
} NVIC_Type;

// Set the priority of a peripheral interrupt IRQn
NVIC->IP[IRQn] = (priority << 4) & 0xFF;
```

Figure 11-9 shows the memory layout of interrupt priority registers for the first 16 peripheral interrupt. For example, the following code sets the priority of EXTI Line `0`, whose interrupt number is 6, to the lowest priority. Both the preemption priority and the sub-priority number are `0b11` (*i.e.*, 3).

```
// Set the priority for EXTI 0 (Interrupt number 6)
NVIC->IP[6] = 0xF0;
```

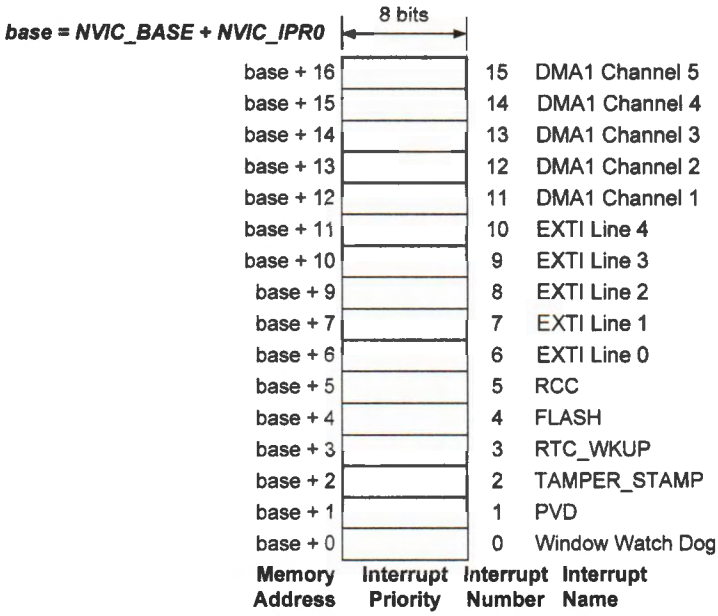


Figure 11-9. Example of interrupt priority (IP) registers for the first 16 peripheral interrupts.

Example 11-5 and Example 11-6 give two example assembly subroutines that set the priority of a system exception and a peripheral interrupt, respectively.

- Both functions use the STRB (store a byte) instruction, instead of STR (store a word). As discussed previously, the priority array is a byte array, instead of an integer array.
- Also, the subroutine shifts the priority left by four bits because the lower four bits of each priority register are not used in STM32L processors (see Figure 11-7).

```
; Input arguments:
;    r0: Interrupt Number IRQn
;    r1: Interrupt Priority
Set_System_Exception_Priority PROC
   PUSH {r4, lr}
   LSL  r2, r1, #4          ; r2 = priority << 4
   LDR  r3, =SCB_BASE       ; System control block base address
   LDR  r4, =SCB_HP_1_12    ; System handler priority registers
   ADD  r3, r3, r4
   AND  r4, r0, #0x0F
   SUB  r4, r4, #4
   STRB r2, [r3, r4]        ; Save priority; Do not use STR
   POP  {r4, pc}
   ENDP
```

Example 11-5. Setting priority of system interrupts

```
; Input arguments:
;    r0: Interrupt Number IRQn
;    r1: Interrupt Priority

Set_Peripheral_Interrupt_Priority PROC

   PUSH {r4, lr}
   LSL  r2, r1, #4       ; r2 = priority << 4
   LDR  r3, =NVIC_BASE   ; NVIC base address
   LDR  r4, =NVIC_IP0    ; Interrupt priority register
   ADD  r3, r3, r4
   STRB r2, [r3, r0]     ; Save priority; Don't use STR
   POP  {r4, pc}

   ENDP
```

Example 11-6. Setting priority of peripheral interrupts

11.6.3 Global Interrupt Enable and Disable

Besides using NVIC to configure individual interrupts, the Cortex-M processors also allow us to enable and disable a group of interrupts by using change processor state (CPS) instructions.

We use the priority mask register (PRIMASK) to enable or disable interrupts excluding hard faults and non-maskable interrupts (NMI). Also, we use the fault mask register (FAULTMASK) to enable or disable interrupts except for non-maskable interrupts (NMI).

Instruction	Action	Equivalent
CPSID i	Disable interrupts & configurable fault handlers	MOVS r0, #0 MSR PRIMASK, r0
CPSID f	Disable interrupts and all fault handlers	MOVS r0, #1 MSR FAULTMASK, r0
CPSIE i	Enable interrupts and configurable fault handlers	MOVS r0, #0 MSR PRIMASK, r0
CPSIE f	Enable interrupts and fault handlers	MOVS r0, #1 MSR FAULTMASK, r0
N/A	Disable interrupts with priority 0x05 - 0xFF	MOVS r0, #5 MSR BASEPRI, r0

Table 11-3. Instructions for enabling or disabling interrupts (excluding hard faults and NMI)

When the base priority mask register (BASEPRI) is non-zero, all interrupts with a priority value higher than or equal to BASEPRI are disabled. In this case, we also say that interrupts with a priority value lower than BASEPRI are unmasked (*i.e.*, enabled). A larger priority value represents lower priority (*i.e.*, lower urgency).

In the equivalent instructions given in Table 11-3, MSR transfers the content of a general-purpose register to a special-purpose register. Note that MOV or MOVS cannot access these special registers.

11.7 System Timer

The system tick timer (SysTick) is a simple 24-bit down counter to produce a small fixed time quantum. Software uses SysTick to create time delays or generate periodic interrupts to execute a task repeatedly.

- The timer counts down from `N-1` to `0`, and the processor generates a SysTick interrupt once the counter reaches zero.
- After reaching zero, the SysTick counter loads the value held in a special register named the SysTick Reload register and counts down again.
- The SysTick timer does not stop counting down when the processor is halted. The processor still generates SysTick interrupts during the process of debugging.

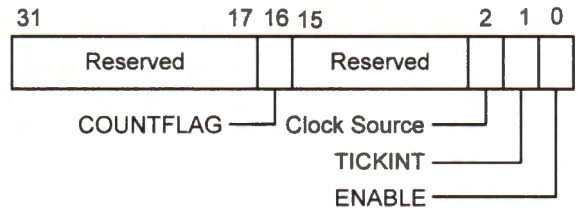
Another usage of SysTick timer is to create a useful hardware timer for the CPU scheduler in real-time operating systems (RTOS). When multiple tasks run concurrently, the processor allocates a time slot to each task according to some scheduling policy, such as the round robin. To achieve that, the processor utilizes a hardware timer to generate interrupts at regular time intervals. These interrupts inform the processor to stop the current task, save the context registers of the present task to the stack, and then select a new task in the job waiting queue to serve. Chapter 23 gives a detailed discussion. When SysTick timers are used as a system level function, processors often protect SysTick timers from being modified by software running in the unprivileged mode.

There are four 32-bit registers for configuring system timers. Their memory addresses are listed in Table 11-4. Chip manufacturers may have different names for these registers, but their memory addresses are the same for each ARM Cortex-M family.

```
SysTick_CTRL  EQU (0xE000E010)   ; SysTick control and status register
SysTick_LOAD  EQU (0xE000E014)   ; SysTick reload value register
SysTick_VAL   EQU (0xE000E018)   ; SysTick current value register
SysTick_CALIB EQU (0xE000E01C)   ; SysTick calibration register
```

Table 11-4. Memory address of control registers for SysTick timer

(1) SysTick control and status register (`SysTick_CTRL`)



- **CLKSOURCE** indicates the clock source:
  0 = External clock. The frequency of SysTick clock is the frequency of the AHB clock divided by 8.
  1 = Processor clock

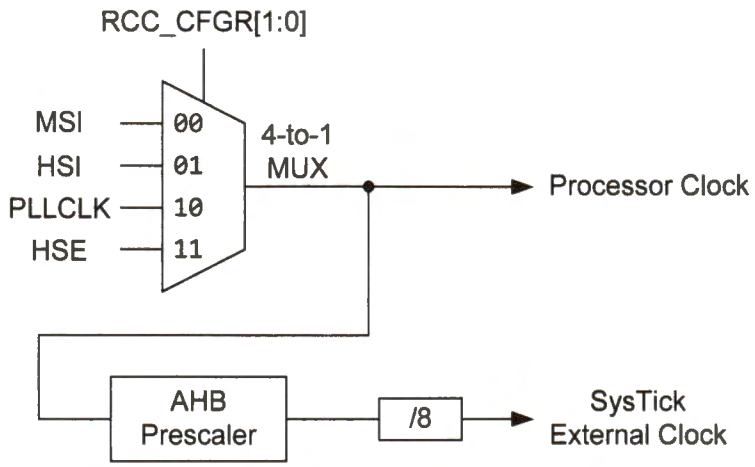


Figure 11-10. Two different clock sources for SysTick on STM32 processor. The default AHB prescaler is 1. RCC_CFGR register selects MSI, HSI, PLLCLK, or HSE.

Figure 11-10 shows the clock selection configured by the RCC_CFGR register. A different ARM Cortex processor might use a different clocking scheme.

- **TICKINT** enables SysTick interrupt request:
  0 = Counting down to zero does not assert the SysTick interrupt request
  1 = Counting down to zero asserts the SysTick interrupt request

- **ENABLE** enables the counter:
  0 = Counter disabled
  1 = Counter enabled

  To enable SysTick interrupt, the program needs to set up three bits:
  1. Set bit TICKINT in SysTick_CTRL to enable SysTick interrupt.
  2. Enable SysTick interrupt in the NVIC vector. Note that SysTick interrupt is enabled by default in the NVIC vector.
  3. Set bit ENABLE in SysTick_CTRL to enable the SysTick timer.

- **COUNTFLAG** indicates whether a special event has occurred.
  1 = Counter has transitioned from 1 to 0 since the last read of SysTick_CTRL
  0 = COUNTFLAG is cleared by reading SysTick_CTRL or by writing to SysTick_VAL.

(2) SysTick reload value register (SysTick_LOAD)

As the SysTick counter wraps on zero, the `SysTick_LOAD` register provides the wrap-around value. After the counter counts down to zero, the counter restarts from the value in `SysTick_LOAD`.

If the SysTick interrupt is required once every N clock pulses, software should set `SysTick_LOAD` to $N - 1$. `SysTick_LOAD` supports any 24-bit values between `1` and `0x00FFFFFF` or `16,777,215` in decimal. For example, if an application needs to generate a SysTick interrupt for each 100 clock pulses sent to the timer, `SysTick_LOAD` should be set to 99.

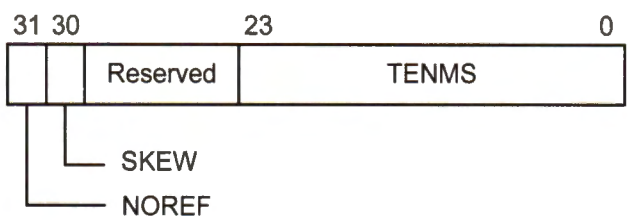
(3) SysTick current value register (SysTick_VAL)

When SysTick is enabled, the 24-bit current counter value in the `SysTick_VAL` register is copied from the `SysTick_LOAD` register initially. However, this value in `SysTick_VAL` is arbitrary on reset.

The processor automatically decrements `SysTick_VAL` by one at each clock pulse sent to the timer. Writing any value to `SysTick_VAL` resets it to zero, making the counter restart from `SysTick_LOAD` on the next clock pulse. Reading `SysTick_VAL` returns the current timer value.

(4) SysTick calibration register (SysTick_CALIB)

- The `TENMS` value in the calibration register (stored in the `TENMS` field) is the required preload value for generating a time interval of 10 *ms* (*i.e.*, a timer of 100 Hz). Software can use an external reference clock with high accuracy to calibrate the system timer. If `TENMS` is zero, we can calculate its value from the clock frequency that drives the timer's counter. The `TENMS` value provides a convenient way to generate a specific time interval. For example, to generate a SysTick interrupt every 1 *ms*, we can set `SysTick_LOAD` to `TENMS/10`.



- The SKEW indicates whether the 10 *ms* calibration is exact or inexact. If it is 0, the TENMS field cannot generate exactly 10 *ms* due to small variations in clock frequency.
- The NOREF indicates whether the processor chip has implemented a reference clock. If TENMS is 1, the reference clock has not been applied by the chip manufacturer.

Example of calculating the system timer interval

Figure 11-11 shows an example of how to calculate the time interval between two consecutive SysTick interrupts. In this example, SysTick_LOAD is 6. If the processor clock is 1 MHz and the SysTick counter takes it as the input clock, then we can calculate the SysTick interrupt period as follows:

$$SysTick\ Interrupt\ Period = (1 + SysTick\_LOAD) \times \frac{1}{SysTick\ Counter\ Clock\ Frequency}$$

$$= (1 + 6) \times \frac{1}{1MHz}$$

$$= 7\mu s$$

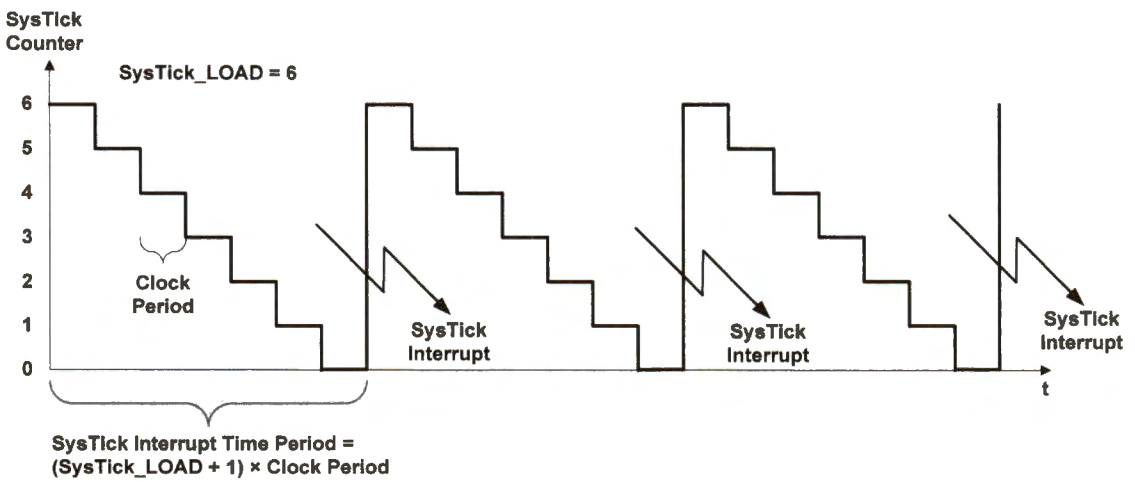


Figure 11-11. Example of SysTick interrupt when SysTick_LOAD is 6.

Figure 11-12 shows the flowchart of the SysTick initialization and an endless loop that use the delay function. The processor executes concurrently the SysTick interrupt handler (shown in Example 11-8) and the delay function (shown in Example 11-9).

The interrupt handler decrements the *ticks* variable by one whenever a SysTick interrupt is generated. The delay function uses polling to check constantly whether the SysTick handler decreases *ticks* to zero. The delay function exits when *ticks* reaches zero. To delay one second, the program should initialize *ticks* to 1000 if SysTick generates an interrupt every 1 *ms*. Note that *SysTick_Initialize()* function sets SysTick_LOAD to *ticks* - 1.

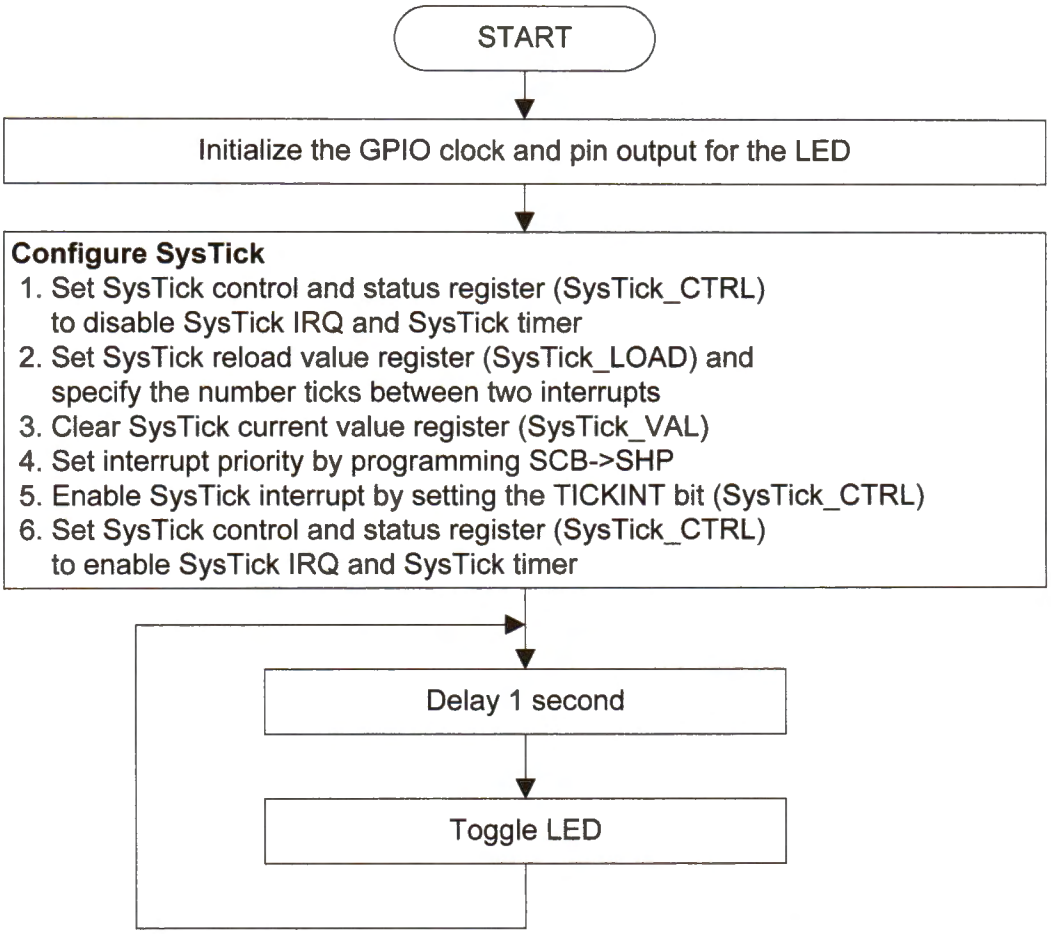


Figure 11-12 Flowchart of the main program to toggle an LED periodically

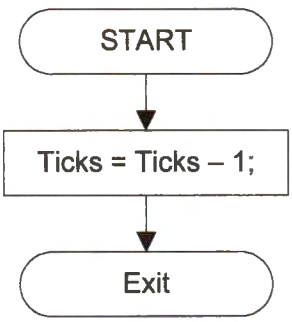


Figure 11-13. SysTick interrupt handler

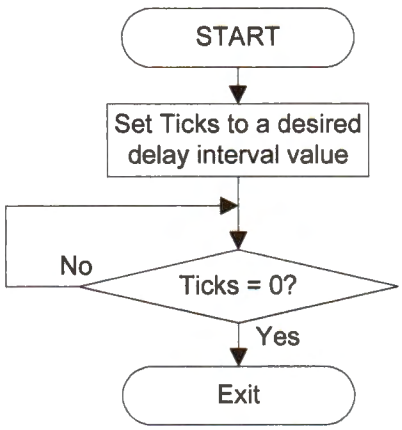


Figure 11-14. Delay function

To enable the SysTick interrupt request, we can use the following C statement to set bit 2 of SysTick_CTRL to 1 by using bitwise OR.

```
// Set bit 2 in SysTick_CTRL
*((volatile uint32_t *) 0xE000E010) |= 1UL<<2;
```

This statement first casts the memory address of SysTick_CTRL to a memory pointer, which points to a 32-bit unsigned integer. Then, it uses dereferencing to access the memory. Dereferencing a pointer mean, getting the value that is stored in the memory location pointed by the pointer.

However, directly dereferencing a numeric memory address is inconvenient to use in practice and its code is hard to read. To improve the code readability and create convenience in programming, we can cast a contiguous block of physical memory to a data structure.

Table 11-5 shows the key data structure of SysTick. The SysTick data structure includes four registers introduced previously. The memory address pre-defined is converted to a pointer variable pointing to the SysTick structure. Chapter 10.2.2 discusses the keyword "volatile".

```
// Permission definitions
#define __I   volatile const // defines as read only
#define __O   volatile       // defines as write only
#define __IO  volatile       // allows both read and write

// Memory mapping structure for SysTick
typedef struct {
  __IO uint32_t CTRL;        // SysTick control and status register
  __IO uint32_t LOAD;        // SysTick reload value register
  __IO uint32_t VAL;         // SysTick current value register
  __I  uint32_t CALIB;       // SysTick calibration register
} SysTick_Type;

// Memory address pre-defined by the chip manufacturer
#define SysTick_BASE  0xE000E010

// Cast a pointer to the SysTick struct
#define SysTick       ((SysTick_Type *) SysTick_BASE)
```

Table 11-5. Data structure of SysTick

The following gives a few example functions in C and assembly to show how to set up the SysTick timer and perform time delay. The *SysTick_Initialize()* function sets the SysTick to generate interrupts at a fixed-time interval. The input parameter *ticks* equals the time interval divided by the clock period.

```
// Input:  ticks = number of ticks between two interrupts
void SysTick_Initialize (uint32_t ticks) {

  // Disable SysTick IRQ and SysTick counter
  SysTick->CTRL = 0;

  // Set reload register
  SysTick->LOAD = ticks - 1;

  // Set interrupt priority of SysTick
  // Make SysTick least urgent (i.e., highest priority number)
  // __NVIC_PRIO_BITS: number of bits for priority levels, defined in CMSIS
  NVIC_SetPriority (SysTick_IRQn, (1<<__NVIC_PRIO_BITS) - 1);

  // Reset the SysTick counter value
  SysTick->VAL = 0;

  // Select processor clock
  // 1 = processor clock;  0 = external clock
  SysTick->CTRL |= SysTick_CTRL_CLKSOURCE_Msk;

  // Enables SysTick exception request
  // 1 = counting down to zero asserts the SysTick exception request
  // 0 = counting down to zero does not assert the SysTick exception request
  SysTick->CTRL |= SysTick_CTRL_TICKINT_Msk;

  // Enable SysTick timer
  SysTick->CTRL |= SysTick_CTRL_ENABLE_Msk;
}
```

Example 11-7. Generating an interrupt periodically with a fixed-time interval in C

The SysTick interrupt handler decrements the *TimeDelay* variable, as shown below.

```
void SysTick_Handler (void) { // SysTick interrupt service routine
  // TimeDelay is a global variable declared as volatile
  if (TimeDelay > 0)          // Prevent it from being negative
    TimeDelay--;              // TimeDelay is a global volatile variable
}
```

Example 11-8. SysTick Interrupt Handler in C

The Delay function initializes the *TimeDelay* variable and waits until *TimeDelay* is decremented to zero by *SysTick_Handler()*.

```
void Delay(uint32_t nTime) {
  // nTime: specifies the delay time length
  TimeDelay = nTime;          // TimeDelay must be declared as volatile
  while(TimeDelay != 0);      // Busy wait
}
```

Example 11-9. Delay function in C

The following shows the complete codes in assembly to initialize the system timer.

```
SysTick_Initialize PROC
  EXPORT SysTick_Initialize

  ; Set SysTick_CTRL to disable SysTick IRQ and SysTick timer
  LDR r0, =SysTick_BASE

  ; Disable SysTick IRQ and SysTick counter, select external clock
  MOV r1, #0
  STR r1, [r0, #SysTick_CTRL]

  ; Specify the number of clock cycles between two interrupts
  LDR r2, =262                    ; Change it based on interrupt interval
  STR r2, [r0, #SysTick_LOAD]     ; Save to SysTick reload register

  ; Clear SysTick current value register (SysTick_VAL)
  MOV r1, #0
  STR r1, [r0, #SysTick_VAL]      ; Write 0 to SysTick value register

  ; Set interrupt priority for SysTick
  LDR  r2, =SCB_BASE
  ADD  r2, r2, #SCB_SHP
  MOV  r3, #1<<4                  ; Set priority as 1, see Figure 11-7
  STRB r3, [r2, #11]              ; SCB->SHP[11], see Figure 11-8

  ; Set SysTick_CTRL to enable SysTick timer and SysTick interrupt
  LDR r1, [r0, #SysTick_CTRL]
  ORR r1, r1, #3                  ; Enable SysTick counter & interrupt
  STR r1, [r0, #SysTick_CTRL]

  BX  lr  ; Exit
  ENDP
```

Example 11-10. Configuring SysTick timer in assembly

We can implement *SysTick_Handler()* in assembly as follows. It decreases the *TimeDelay* variable (assume it is saved in register r10) by one when a SysTick interrupt is generated, *i.e.*, the SysTick counter counts down to zero.

```
SysTick_Handler  PROC
  EXPORT SysTick_Handler

  ; NVIC automatically stacks eight registers: r0 - r3, r12, LR, PSR and PC
  SUB r10, r10, #1     ; Decrement TimeDelay
  BX  lr               ; Exit and trigger auto-unstacking

  ENDP
```

Example 11-11. SysTick interrupt handler in assembly

The *delay* function given in Example 11-12. Register `r0` is the input argument, representing the amount of delay in time units set by the *SysTick_Handler*. The function deploys a busy-waiting loop, which exits when the *TimeDelay* variable has been decreased to zero by the SysTick interrupt handler *SysTick_Handler*().

```
delay  PROC
       EXPORT delay

       ; r0 is the TimeDelay input
       MOV  r10, r0        ; Make a copy of TimeDelay
loop   CMP  r10, #0        ; Wait for TimeDelay = 0
       BNE  loop           ; r10 is decreased periodically by SysTick_Handler
       BX   lr             ; Exit
       ENDP
```

Example 11-12. Delay subroutine in assembly

A common mistake in Example 11-11 and Example 11-12 is that a programmer might choose register `r0`, `r1`, `r2`, `r3`, or `r12`, instead of `r10` to hold *TimeDelay*. The processor automatically pushes these registers onto the stack when `SysTick_Handler` starts, and then automatically pops them off the stack. Therefore, `SysTick_Handler` would fail to change *TimeDelay* if `SysTick_Handler` uses one of these registers to represent *TimeDelay*.

11.8 External Interrupt

External interrupts are interrupts triggered by peripherals or devices, external to the microprocessor core, such as push buttons and keypads. External interrupts are very useful because they allow the microcontroller to monitor external signals efficiently and promptly response to external events.

The external interrupt controller supports 16 external interrupts, named `EXTI0`, `EXTI1`, ..., `EXTI15`. Each of these interrupts is only associated with one specific GPIO pin. However, a microcontroller has more than 16 GPIO pins. How does the microcontroller map GPIO pins to external interrupts?

The GPIO pins with the same pin number in all GPIO ports are assigned to the same external interrupt, as shown in Figure 11-15. In other words, only pins with the pin number *k* can be the source of external interrupt EXTI *k*. For example, the processor can map GPIO pin PA 0 to EXTI 0, PA 1 to EXTI 1, PA 2 to EXTI 2, and so on.

Also, there is only one external interrupt on all pins with the same number out of all GPIO ports. For example, if the pin PA 3 has an external interrupt on it, we cannot use the pins PB 3, PC 3, PD 3, or PE 3 as the external interrupt source.

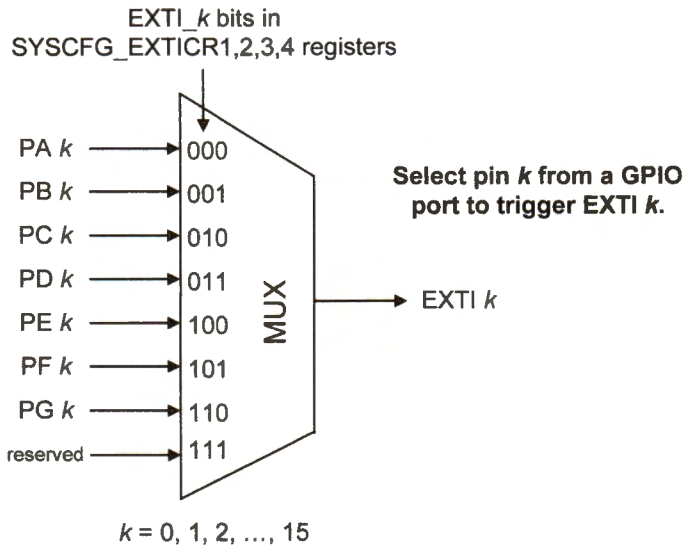


Figure 11-15. Mapping between external interrupt (EXTI) and GPIO pins. A multiplexer (MUX) is a circuit that selects one of its inputs and forwards the selected input to the output.

Figure 11-16 shows an example in which a button is connected to pin PA 3. When the button is pressed, the voltage on PA 3 goes high. Software should configure Pin PA 3 to be pulled down internally so that PA 3 remains low when the button is not pressed.

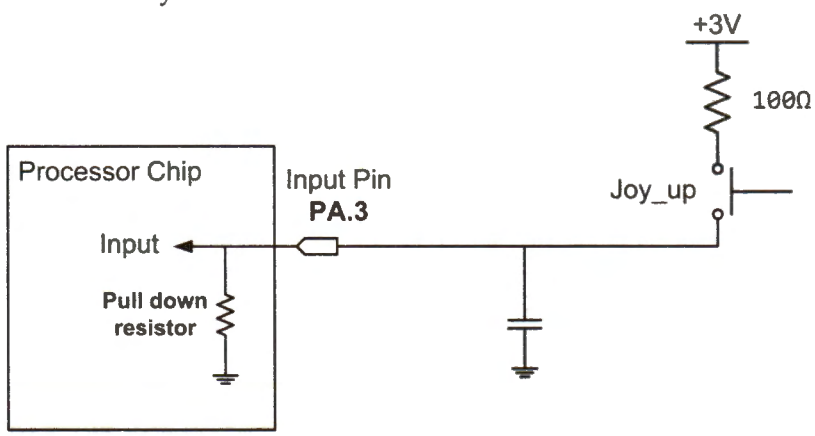


Figure 11-16. Interfacing an external button via external interrupt.

The external interrupt controller has a voltage monitor module that has a programmable edge detector. This hardware monitor module can monitor the voltage signal on GPIO pins. Software can select the rising edge, the falling edge, or both edges of the voltage signal on PA 3 to trigger an interrupt request named EXTI3. The EXTI3 interrupt request is then sent to the NVIC controller. Finally, the microcontroller responds to the interrupt request and executes the interrupt service routine *EXTI3_IRQHandler()*.

The following shows the software configuration to select GPIO pin *k* as the trigger source for external interrupt EXTI *k*.

1. Enable the clock of SYSCFG and corresponding GPIO port.
2. Configure the GPIO pin *k* as input.
3. Set up the SYSCFG external interrupt configuration register (`SYSCFG_EXTICR`) to map the GPIO pin *k* to the external interrupt input line *k*.
4. Select the active edge that can trigger EXTI *k*. The signal can be a rising edge, a falling edge or both. This is programmed via the rising edge trigger selection register (`EXTI_RTSR1` or `EXTI_RTSR2`) and the falling edge trigger selection register (`EXTI_FTSR1` or `EXTI_FTSR2`).
5. Enable EXTI *k* by setting the k^{th} bit in EXTI interrupt mask register (`EXTI_IMR1` or `EXTI_IMR2`). An interrupt can only be generated if the corresponding bit in the interrupt mask register is 1 (or called unmasked).
6. Enable interrupt EXTI *k* on NVIC controller via `NVIC_EnableIRQ`.
7. Write the interrupt handler for EXTI *k*. The EXTI pending register (`EXTI_PR1` or `EXTI_PR2`) records the source of the interrupt. The function name of the interrupt handler is given by the startup assembly file *startup_stm32l476xx.s*. For example, the handler for EXTI 3 is called *EXTI3_IRQHandler()*.
8. In the interrupt handler, software needs to clear the corresponding pending bit to indicate the current request has been handled. Surprisingly, writing it to 1 clears a pending bit.

The following shows an example program that uses the external interrupt to light up the LED when the user push button is pressed. Example 11-13 illustrates the initialization for connecting EXTI 3 to pin 3 of GPIO port A.

```
void EXTI_Init(void) {

  // Enable SYSCFG clock
  RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;

  // Select PA.3 as the trigger source of EXTI 3
  SYSCFG->EXTICR[0] &= ~SYSCFG_EXTICR1_EXTI3;
  SYSCFG->EXTICR[0] |=  SYSCFG_EXTICR1_EXTI3_PA;
  SYSCFG->EXTICR[0] &= ~(0x000F);

  // Enable rising edge trigger for EXTI 3
  // Rising trigger selection register (RSTR)
  // 0 = disabled, 1 = enabled
  EXTI->RTSR |= EXTI_RTSR_RT3;
```

```
  // Disable falling edge trigger for EXTI 3
  // Falling trigger selection register (FSTR)
  // 0 = disabled, 1 = enabled
  EXTI->FTSR &= ~EXTI_FTSR_RT3;

  // Enable EXTI 3 interrupt
  // Interrupt mask register: 0 = masked, 1 = unmasked
  // "Masked" means that processor ignores the corresponding interrupt.
  EXTI->IMR1 |= EXTI_IMR1_IM3;          // Enable EXTI line 3

  // Set EXTI 3 priority to 1
  NVIC_SetPriority(EXTI3_IRQn, 1);

  // Enable EXTI 3 interrupt
  NVIC_EnableIRQ(EXTI3_IRQn);
}
```

Example 11-13. Initialize and enable EXTI 3 for pin PA.3

Example 11-14 shows the code of the external interrupt handler of EXTI 3, which toggles an LED when PA 3 triggers an external interrupt.

One mistake that new programmers often make is that interrupt handlers do not clear interrupt pending flags. If the interrupt handler does not clear an interrupt pending flag after processing the interrupt, the microcontroller would mistakenly think another interrupt request has arrived and then repeatedly execute the interrupt handler.

```
void EXTI3_IRQHandler(void) {
  // Check for EXTI 3 interrupt flag
  if ((EXTI->PR1 & EXTI_PR1_PIF3) == EXTI_PR1_PIF3) {

    // Toggle LED
    GPIOB->ODR ^= 1<<8;           // Toggle PB.8 output

    // Clear interrupt pending request
    EXTI->PR1 |= EXTI_PR1_PIF3;  // Write 1 to clear
  }
}
```

Example 11-14. External interrupt handler for EXTI 3

External interrupts can not only monitor the external voltage applied to GPIO pins, but also monitor internal events, such as RTC alarm, COMP outputs, or internal wakeup events. Also, software can trigger EXTI interrupts by writing to the EXTI software interrupt event register (`EXTI_SWIER`).

11.9 Software Interrupt

Interrupt signals can be generated by hardware, such as hardware timers and peripheral hardware components. Software can also generate interrupt signals by setting the interrupt pending registers or by using special instructions. There are two major usages of software interrupt: exception handling and privileged hardware access.

Exception Handling: When exceptional conditions occur during execution, such as division by zero, illegal opcode, and invalid memory access, the processors should handle these abnormal situations to potentially correct software errors. The processor can capture two software faults, including division by error and unaligned memory access, if software enables these fault capture features. A software interrupt invoked by software faults is often referred to as a ***trap***. Example 11-15 enables the trap of dividing by zero.

```
    LDR  r2, =SCB_Base        ; Base address of system control block (SCB)
    LDR  r3, [r2, #SCB_CCR]   ; Read Configuration and Control Register
    ORR  r3, r3, #16          ; Enable trap on dividing by 0
    STR  r3, [r2, #SCB_CCR]   ; Write Configuration and Control Register

    MOV  r0, #0
    MOV  r1, #1
    UDIV r1, r1, r0           ; Invoke hard fault
```

Example 11-15. A trap that handles abnormal situations

If the software enables the dividing-by-zero trap and a division instruction (`UDIV` or `SDIV`) generates such a trap, the processor halts and invokes the hard fault handler shown in Example 11-16. The fault handler may print out the error message or reboot the processor.

```
HardFault_Handler PROC
    EXPORT HardFault_Handler

    ; Handle the error of division by 0
    ; For example, force the processor to reboot

    BL NVIC_SystemReset     ; Reboot the system using AIRCR register
    ENDP
```

Example 11-16. Hard fault interrupt handler

Privileged Hardware Access: When a user application runs in unprivileged mode and needs to access a hardware resource that is only accessible in privileged mode, a special instruction (supervisor call) generates a software interrupt and makes the processor switch from the unprivileged mode to the privilege mode. Chapter 23.2 introduces the supervisor call (`SVC`).

11.10 Exercises

1. Compare two methods of responding to external events: polling and interrupts. Discuss the advantages and disadvantages of each approach.

2. Give two example instructions that make an interrupt service routine exit.

3. The MSI (multi-speed internal) oscillator clock is selected as system clock source after startup from Reset, wakeup from Stop or Standby low power modes. The MSI clock has seven optional frequency ranges available: 100 KHz, 200 KHz, 400 KHz, 800 KHz, 1 MHz, 2 MHz, 4 MHz (default value), 8 MHz, 16 MHz, 24 MHz, 32 MHz, and 48 MHz. Write an assembly program that selects MSI 8 MHz as the system clock.

4. If the MSI 4.094 MHz clock is selected as the system clock and the SysTick selects it as the clock, what should the `SysTick_LOAD` register be to generate a SysTick interrupt every microsecond? What is the `SysTick_LOAD` value to produce a SysTick interrupt every millisecond?

5. Suppose software selects the default MSI (4 MHz) to drive the system timer (SysTick). Can you use this MSI to generate a SysTick interrupt every minute? If yes, show how do you set up the system timer registers. If not, give a solution to solve this problem.

6. Is it possible to use the SysTick Timer to generate an interrupt once every 12 seconds if there is only a clock of 2.097 MHz? If not, name two ways that you can solve this problem. If yes, how to set up the timer registers?

7. Suppose register i ($i \leq 12$) is initialized to have a value of i (e.g. `r0 = 0`, `r1= 1`, `r2 = 2`, `r3 = 3`, etc.). Assume the main stack (`MSP`) is used. Also, in the interrupt handler, if `LR = 0xFFFFFFF9`, then the main stack (`MSP`) is used. If `LR = 0xFFFFFFFD`, then the process stack (`PSP`) is used. The program status register (`PSR`) = `0x00000020`, `PC = 0x08000020`, and `LR = 0x20008020`, when the interrupt occurs.

   (1) Show the stack content immediately before the `PUSH` instruction runs. Suppose the stack pointer `SP` (*i.e.*, `MSP` in this case) was `0x20000600` immediately before the system timer interrupt occurs.

```
SysTick_Handler PROC
    PUSH {r4, r5, r6}
    ADD  r0, r0, #1
    ADD  r1, r1, #1
    ADD  r2, r2, #1
    ADD  r3, r3, #1
    ADD  r4, r4, #1
    ADD  r5, r5, #1
    ADD  r6, r6, #1
    ADD  r7, r7, #1
    ADD  r8, r8, #1
    ADD  r9, r9, #1
    ADD  r10, r10, #1
    ADD  r11, r11, #1
    ADD  r12, r12, #1
    POP  {r4, r5, r6}
    BX   LR
    ENDP
```

(2) What are the values of these registers (R0-R12, LR, SP, and PC) immediately after the interrupt exits?

8. Suppose the SysTick interrupt occurs when PC = 0x08000044, XPSR = 0x00000020, SP = 0x20000200, LR = 0x08001000, and register $R_i = i$, $i = 0, 1, 2, \ldots, 12$.

Memory Address	Instruction
	__main PROC
	…
0x08000044	MOV r3, #0
	…
	ENDP
	SysTick_Handler PROC
	EXPORT SysTick_Handler
0x0800001C	ADD r3, #1
0x0800001E	ADD r4, #1
0x08000020	BX lr
	ENDP

(1) Show the stack contents and the value of PC and SP when immediately entering the SysTick interrupt service routine.

(2) When executing the instruction "BX LR", how does the processor know whether it is exiting a standard subroutine or an interrupt service routine? What operations does the processor perform when a standard subroutine exits? What operations does the processor perform when an interrupt service routine exits?