Quantum Leaps, LLC
103 Cobble Ridge Drive
Chapel Hill, NC 27516
www.state-machine.com

Setting ARM Cortex-M Interrupt Priorities in QP 5.1

This Application Note describes how to set the ARM Cortex-M interrupt priorities in QP™ version **5.1.0** or higher.

The interrupt disabling policy for ARM-Cortex-M3/M4 has changed in QP **5.1**. Interrupts are now disabled more **selectively** using the BASEPRI register, which disables only interrupts with numerical value of priority higher than the current value written to BASEPRI register, and **never disables interrupts with priority values below this level** ("zero interrupt latency"). Among others, this means that in QP 5.1 and higher versions, you should always set all interrupt priorities **explicitly**. Leaving the interrupt priorities at zero, which is the default out of reset, is most likely **incorrect**, because interrupts with priority zero are never disabled and **cannot** call any QP services.

> **NOTE:** Calling QP services, such as `QF_TICK()`, `QF_PUBLISH()`, `Q_NEW()`, or `QACTIVE_POST()`, from interrupts that are never disabled is **unsafe**, because it can cause data corruption within the framework.

1 Cortex-M Interrupt Management Hardware and Conventions

The ARM Cortex-M processor offers very versatile interrupt priority management in the Nested Vectored Interrupt Controller (NVIC), but unfortunately, the priority numbering conventions and layout of hardware registers used in managing the interrupt priorities are often counter-intuitive and confusing. This section describes briefly the relevant priority numbering conventions and hardware registers to clarify the terminology and avoid confusion.

1.1 The Inverse Relationship Between Priority Numbers and Urgency of the Interrupts

The most important fact to know is that ARM Cortex-M uses the "reversed" priority numbering scheme for interrupts, where priority zero corresponds to the highest urgency[1] interrupt and higher numerical values of priority correspond to lower urgency. This numbering scheme poses a constant threat of confusion, because any use of the terms "higher" or "lower" priority immediately requires clarification, whether they represent the numerical value of priority, or perhaps, the urgency of an interrupt.

> **NOTE:** To avoid this confusion, in the rest of this document, the term "**priority**" means the numerical value of interrupt priority in the ARM Cortex-M convention. The term "**urgency**" means the capability of an interrupt to preempt other interrupts. A higher-urgency interrupt (lower priority number) can preempt a lower-urgency interrupt (higher priority number).

1.2 Interrupt Priority Configuration Registers in the NVIC

The number of priority levels in the ARM Cortex-M core is configurable, meaning that various silicon vendors can implement different number of priority bits in their chips. However, there is a minimum number of interrupt priority bits that need to be implemented, which is 2 bits in ARM Cortex-M0 and 3 bits in ARM Cortex-M3/M4. But here again, the most confusing fact is that the priority bits are implemented in the **most-significant** bits of the priority configuration registers in the NVIC. The following Figure 1 illustrates the bit assignment in a priority configuration register for 3-bit implementations, such as TI Tiva MCUs, and 4-bit implementations, such as the NXP LPC17xx ARM Cortex-M3 MCUs.

1 The ARM Cortex-M documentation uses also **negative** priority levels for hardware exceptions of even higher urgency than zero. However, these priority levels are not configurable and cannot be used for regular interrupts.

Quantum Leaps, LLC
103 Cobble Ridge Drive
Chapel Hill, NC 27516
www.state-machine.com

Figure 1: Interrupt priory registers with 3 bits of priority (A), and 4 bits of priority (B)

(A)

Bit7	**Bit6**	**Bit5**	Bit4	Bit3	Bit2	Bit1	Bit0
Implemented 8 priority levels			Not implemented				

Example: TI Tiva Cortex-M4F

(B)

Bit7	**Bit6**	**Bit5**	**Bit4**	Bit3	Bit2	Bit1	Bit0
Implemented 16 priority levels				Not implemented			

Example: NXP LPC17xx Cortex-M3

The relevance of the bit representation in the NVIC priority register is that this creates **another priority numbering scheme**, in which the numerical value of the priority is shifted to the left by the number of unimplemented priority bits. If you ever write directly to the priority registers in the NVIC, you must remember to use this convention.

NOTE: The interrupt priorities don't need to be uniquely assigned, so it is perfectly legal to assign the same interrupt priority to many interrupts in the system. That means that your application can service many more interrupts than the number of interrupt priority levels.

NOTE: Out of reset, all interrupts and exceptions with configurable priority have the same default priority of **zero**. This priority number represents the highest-possible interrupt urgency.

1.3 Interrupt Priority Numbering in the CMSIS

The Cortex Microcontroller Software Interface Standard CMSIS provided by ARM Ltd. is the recommended way of way of programming Cortex-M microcontrollers the in a portable way. The CMSIS standard provides the function `NVIC_SetPriority(IRQn, priority)` for setting the interrupts priorities.

However, it is very important to note that the '`priority`' argument of this function must **not** be shifted by the number of unimplemented bits, because the function performs the shifting by the number of unimplemented bits `(8 - __NVIC_PRIO_BITS)` internally, before writing the value to the appropriate priority configuration register in the NVIC. The number of implemented priority bits `__NVIC_PRIO_BITS` is defined in CMSIS for each ARM Cortex-M device.

For example, calling `NVIC_SetPriority(7, 6)` will set the priority configuration register corresponding to IRQ#7 to 1100,0000 binary on ARM Cortex-M with 3-bits of interrupt priority and it will set 0110,0000 binary on ARM Cortex-M with 4-bits of priority.

NOTE: The confusion about the priority numbering scheme used in the `NVIC_SetPriority()` is further promulgated by various code examples on the Internet and even in reputable books. For example the book *"The Definitive Guide to ARM Cortex-M3, Second Edition"*, ISBN 979-0-12-382091-4, Section 8.3 on page 138 includes a call `NVIC_SetPriority(7, 0xC0)` with the intent to set priority of IR#7 to 6. This call is incorrect and will set the priority of IR#7 to zero.

Quantum Leaps, LLC
103 Cobble Ridge Drive
Chapel Hill, NC 27516
www.state-machine.com

1.4 Preempt Priority and Subpriority

The interrupt priority registers for each interrupt is further divided into two parts. The upper part (most-significant bits) is the **preempt priority**, and the lower part (least-significant bits) is the **subpriority**. The number of bits in each part of the priority registers is configurable via the Application Interrupt and Reset Control Register (AIRC, at address 0xE000ED0C).

The **preempt priority** level defines whether an interrupt can be serviced when the processor is already running another interrupt handler. In other words, preempt priority determines if one interrupt can preempt another.

The **subpriority** level value is used only when two exceptions with the same preempt priority level are pending (because interrupts are disabled, for example). When the interrupts are re-enabled, the exception with lower subpriority (higher urgency) will be handled first.

In most QP applications, it is recommended to assign all the interrupt priority bits to the preempt priority group, leaving no priority bits as subpriority bits, which is the default setting out of reset. Any other configuration complicates the otherwise direct relationship between the interrupt priority number and interrupt urgency.

NOTE: Some third-party code libraries (e.g., the STM32 driver library) changes the priority grouping configuration to non-standard. Therefore, it is **highly recommended** to explicitly re-set the priority grouping to the default by calling the CMSIS function `NVIC_SetPriorityGrouping(0U)`.

1.5 Disabling Interrupts with PRIMASK and BASEPRI Registers

The real-time schedulers included in the QP framework, as all other real-time kernels, need to perform certain operations atomically to prevent data corruption. QP achieves the atomicity by briefly disabling and re-enabling interrupts.

The ARM Cortex-M offers two methods of disabling and re-enabling interrupts. The simplest method is to set and clear the interrupt bit in the **PRIMASK** register. Specifically, disabling interrupts can be achieved with the "`CPSID i`" instruction and enabling interrupts with the "`CPSIE i`" instruction. This method is simple and fast, but it disables all interrupt levels indiscriminately. This is the only method available in the ARMv6-M architecture (Cortex-M0/M0+).

However, the more advance ARMv7-M (Cortex-M3/M4/M4F) provides additionally the **BASEPRI** special register, which allows you to disable interrupts more selectively. Specifically, you can disable interrupts only with urgency lower than a certain level and leave the higher-urgency interrupts not disabled at all.

The CMSIS provides the function `__set_BASEPRI(priority)` for changing the value of the BASEPRI register. The function uses the hardware convention for the '`priority`' argument, which means that the priority must be shifted left by the number of unimplemented bits `(8 - __NVIC_PRIO_BITS)`.

NOTE: The priority numbering convention used in `__set_BASEPRI(priority)` is thus different than in the `NVIC_SetPriority(priority)` function, which expects the "`priority`" argument **not** shifted.

Quantum Leaps, LLC
103 Cobble Ridge Drive
Chapel Hill, NC 27516
www.state-machine.com

2 "Kernel-Aware" and "Kernel-Unaware" Interrupts

Starting from QP **5.1.0**, the QP port to ARM Cortex-M3/M4 **never completely disables interrupts**, even inside the critical sections. On Cortex-M3/M4 (ARMv7-M architectures), the QP port disables interrupts selectively using the BASEPRI register. As shown in Figure 2 and Figure 3, this policy divides interrupts into "kernel-unaware" interrupts, which are never disabled, and "kernel-aware" interrupts, which are disabled in the QP critical sections.

Only "kernel-aware" interrupts are allowed to call QP services. "Kernel-unaware" interrupts are **not allowed** to call any QP services and they can communicate with QP only by triggering a "kernel-aware" interrupt (which can post or publish events).

NOTE: The BASEPRI register is not implemented in the ARMv6-M architecture (**Cortex-M0**/M0+), so Cortex-M0/M0+ need to use the PRIMASK register to disable interrupts globally. In other words, in Cortex-M0/M0+ ports, all interrupts are "kernel-aware".

Figure 2: Kernel-aware and kernel-unaware interrupts with 3 priority bits implemented in NVIC

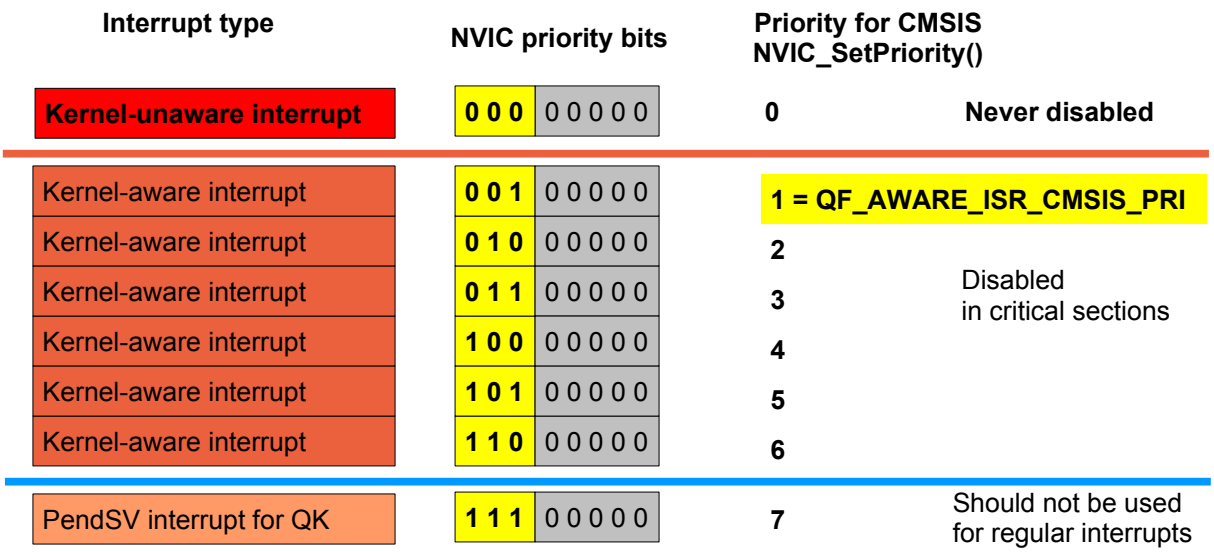
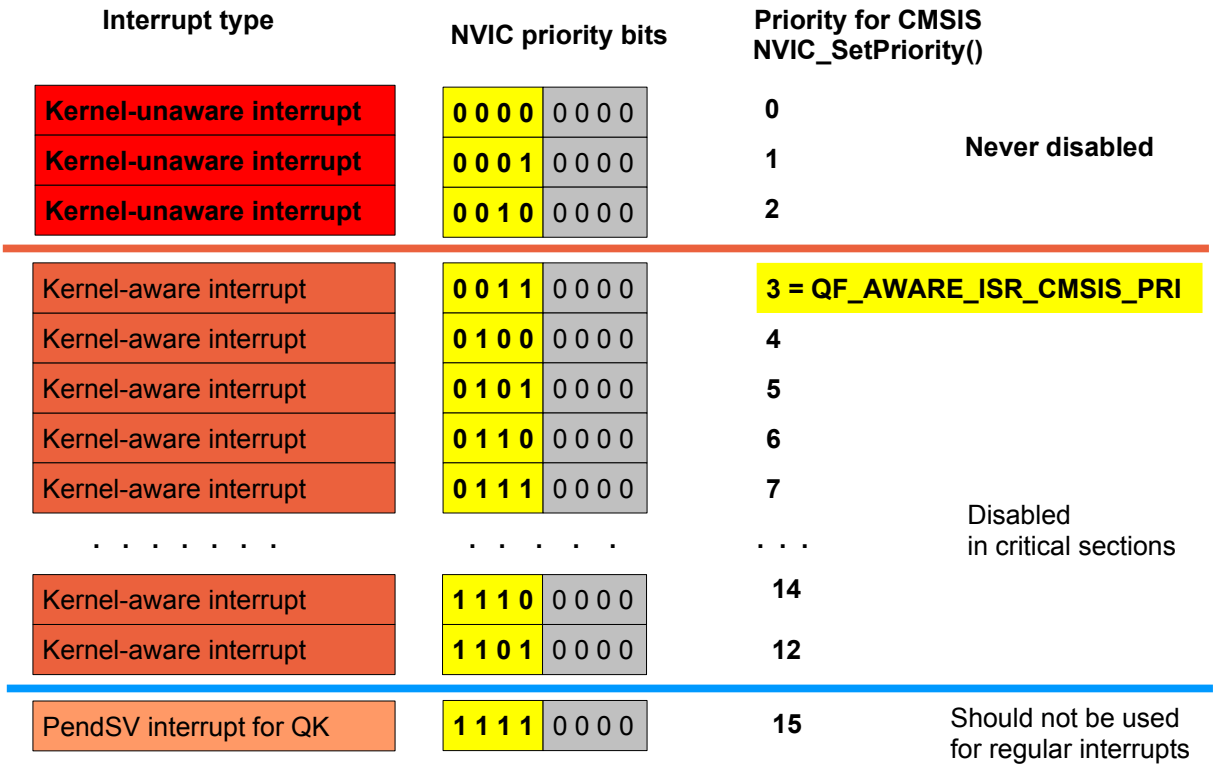


Figure 3: Kernel-aware and kernel-unaware interrupts with 4 priority bits implemented in NVIC



Quantum Leaps, LLC
103 Cobble Ridge Drive
Chapel Hill, NC 27516
www.state-machine.com

3 Assigning Interrupt Priorities

The example projects accompanying this Application Note demonstrate the recommended way of assigning interrupt priorities in your applications. The initialization consist of two steps: (1) you enumerate the "kernel-unaware" and "kernel-aware" interrupt priorities, and (2) you assign the priorities by calling the `NVIC_SetPriority()` CMSIS function. Listing 1 illustrates these steps with the explanation section following immediately after the code.

Listing 1: Assigning the interrupt priorities (see file bsp.c in the example projects).

```
    /*!!!!!!!!!!!!!!!!!!!!!!!!!!!! CAUTION !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    * Assign a priority to EVERY ISR explicitly by calling NVIC_SetPriority().
    * DO NOT LEAVE THE ISR PRIORITIES AT THE DEFAULT VALUE!
    */
(1) enum KernelUnawareISRs {                                        /* see NOTE00 */
        /* ... */
(2)     MAX_KERNEL_UNAWARE_CMSIS_PRI                        /* keep always last */
    };
    /* "kernel-unaware" interrupts can't overlap "kernel-aware" interrupts */
(3) Q_ASSERT_COMPILE(MAX_KERNEL_UNAWARE_CMSIS_PRI <= QF_AWARE_ISR_CMSIS_PRI);

(4) enum KernelAwareISRs {
(5)     GPIOPORTA_PRI = QF_AWARE_ISR_CMSIS_PRI,                  /* see NOTE00 */
        SYSTICK_PRIO,
        /* ... */
(6)     MAX_KERNEL_AWARE_CMSIS_PRI                          /* keep always last */
    };
    /* "kernel-aware" interrupts should not overlap the PendSV priority */
(7) Q_ASSERT_COMPILE(MAX_KERNEL_AWARE_CMSIS_PRI <= (0xFF>>(8-__NVIC_PRIO_BITS)));

    ~~~~~

(8) void QF_onStartup(void) {
                /* set up the SysTick timer to fire at BSP_TICKS_PER_SEC rate */
        SysTick_Config(ROM_SysCtlClockGet() / BSP_TICKS_PER_SEC);

        /* assing all priority bits for preemption-prio. and none to sub-prio. */
(9)     NVIC_SetPriorityGrouping(0U);

        /* set priorities of ALL ISRs used in the system, see NOTE00
        *
        * !!!!!!!!!!!!!!!!!!!!!!!!!!!! CAUTION !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
        * Assign a priority to EVERY ISR explicitly by calling NVIC_SetPriority().
        * DO NOT LEAVE THE ISR PRIORITIES AT THE DEFAULT VALUE!
        */
(10)    NVIC_SetPriority(SysTick_IRQn,   SYSTICK_PRIO);
(11)    NVIC_SetPriority(GPIOPortA_IRQn, GPIOPORTA_PRIO);
        /* ... */
                                                            /* enable IRQs... */
(12)    NVIC_EnableIRQ(GPIOPortA_IRQn);
    }
```

(1) The enumeration `KernelUnawareISRs` lists the priority numbers for the "kernel-unaware" interrupts. These priorities start with zero (highest possible). The priorities are suitable as the argument for the `NVC_SetPriority()` CMSIS function.

NOTE: The NVIC allows you to assign the same priority level to multiple interrupts, so you can have more ISRs than priority levels running as "kernel-unaware" or "kernel-aware" interrupts.

(2) The last value in the enumeration `MAX_KERNEL_UNAWARE_CMSIS_PRI` keeps track of the maximum priority used for a "kernel-unaware" interrupt.

(3) The compile-time assertion ensures that the "kernel-unaware" interrupt priorities do not overlap the "kernel-aware" interrupts, which start at `QF_AWARE_ISR_CMSIS_PRI`.

(4) The enumeration `KernelAwareISRs` lists the priority numbers for the "kernel-aware" interrupts.

(5) The "kernel-aware" interrupt priorities start with the **`QF_AWARE_ISR_CMSIS_PRI`** offset, which is provided in the `qf_port.h` header file.

(6) The last value in the enumeration `MAX_KERNEL_AWARE_CMSIS_PRI` keeps track of the maximum priority used for a "kernel-aware" interrupt.

(7) The compile-time assertion ensures that the "kernel-aware" interrupt priorities do not overlap the lowest priority level reserved for the PendSV exception.

(8) The `QF_onStartup()` callback function is where you set up the interrupts.

(9) This call to the CMIS function `NVIC_SetPriorityGrouping()` assigns all the priority bits to be preempt priority bits, leaving no priority bits as subpriority bits to preserve the direct relationship between the interrupt priorities and the ISR preemption rules. This is the default configuration out of reset for the ARM Cortex-M3/M4 cores, but it can be changed by some vendor-supplied startup code. To avoid any surprises, the call to `NVIC_SetPriorityGrouping(0U)` is recommended.

(10-11) The interrupt priories fall **all** interrupts ("kernel-unaware" and "kernel-aware" alike) are set explicitly by calls to the CMSIS function `NVIC_SetPriority()`.

(12) All used IRQ interrupts need to be explicitly enabled by calling the CMSIS function `NVIC_EnableIRQ()`.