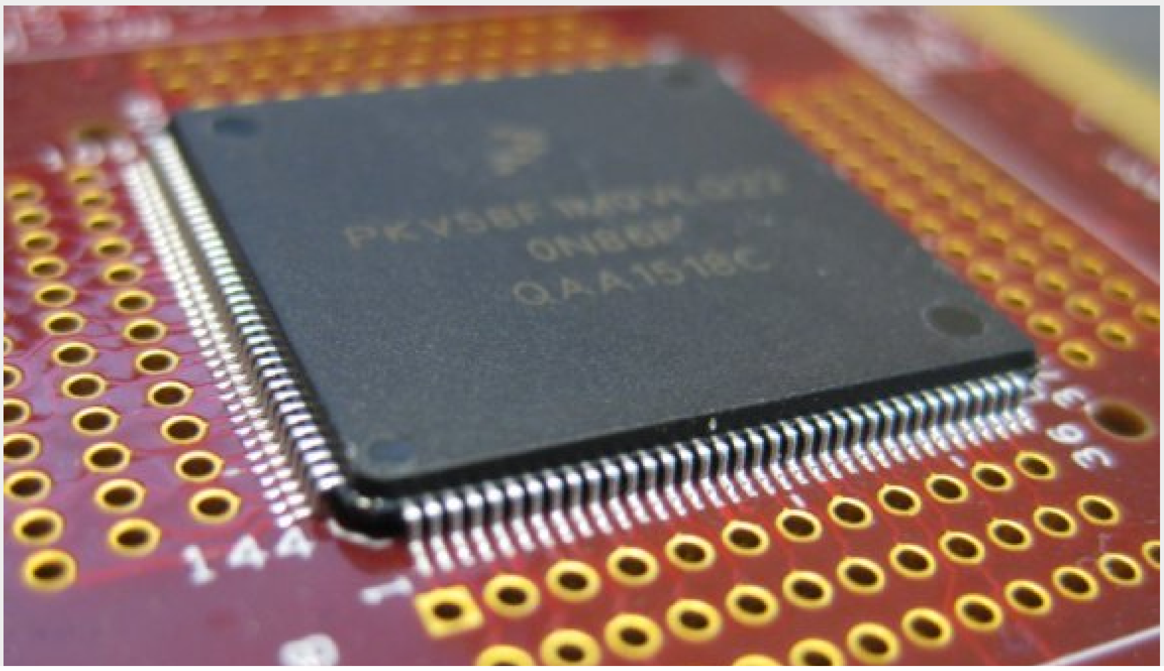
Posted on August 28, 2016 by Erich Styger

ARM Cortex-M Interrupts and FreeRTOS: Part 3

This is the third part about ARM Cortex-M and how the interrupts are used. In Part 1 I discussed the Cortex-M interrupt system and in Part 2 I showed nested interrupt examples. This part is about FreeRTOS and how it uses the Cortex-M interrupt system.



— NXP KV58F ARM Cortex-M7

Outline

FreeRTOS and any other RTOS I'm aware of uses the microcontroller interrupt system. It is critical to understand the interrupt system of the microcontroller for the application, that's why I wrote about this in the previous two parts. Using an RTOS like FreeRTOS means that I have to have an understanding of its usage of the interrupt system, because otherwise it can cause conflicts and wrong behavior of the application.

In this part I describe how FreeRTOS uses the ARM Cortex-M (0/0+/3/4/7) interrupts:

- Interrupts used by the RTOS
- Priorities of the RTOS interrupts
- Critical Section handling
- Partitioning of interrupt priorities/urgencies between the application and the RTOS
- Application startup and interrupts

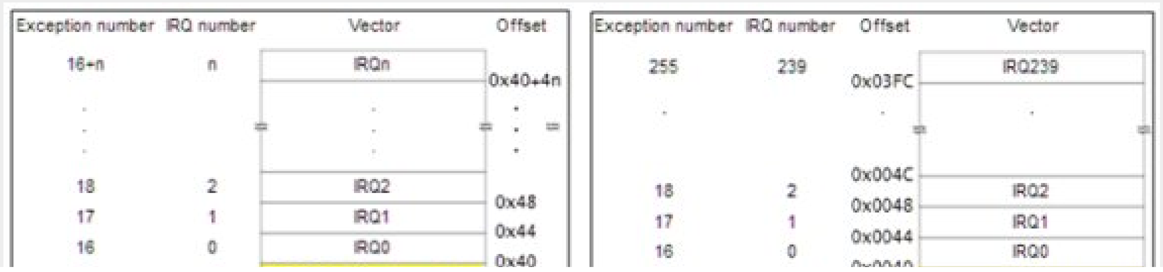
> 💡 *From the FreeRTOS perspective, Cortex-M0 and M0+ are the same, so I'm using M0 both for the M0+ and M0. The Cortex-M3/M4/M7 including their floating point variants are pretty much treated the same by FreeRTOS.*

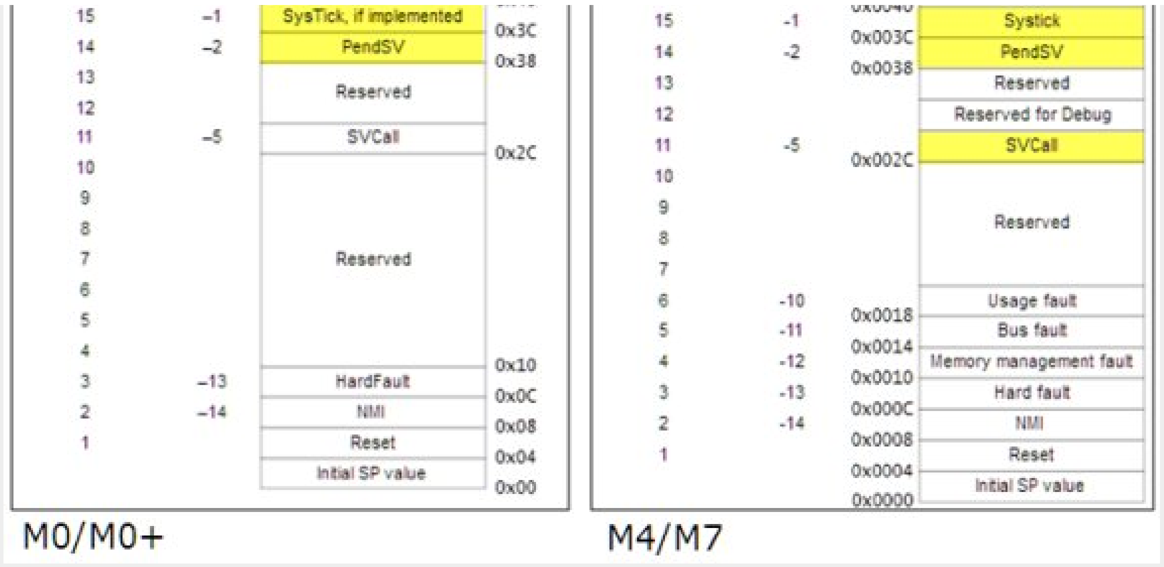
In this article, I'm using GNU assembly syntax to keep it simple. The McuOnEclipse FreeRTOS port covers GNU, IAR and Keil too.

Interrupts used by FreeRTOS

FreeRTOS on ARM Cortex-M uses the two or three interrupts, depending on the architecture and port used:

> 💡 *In FreeRTOS, a 'port' is the part of the Kernel which is microcontroller specific. This part deals with the low level hardware. Everything else in FreeRTOS is generic and written in C. The port part is written in a mix of C and assembly.*





— Interrupts Used by FreeRTOS (Source of tables: ARM Info Center)

Earlier ports for ARM Cortex-M0 did use SVCall too. Current port files provide that interrupt service routine for backward compatibility only.

- **SysTick**: this one is used as time base (timer interrupt) for the RTOS. Typical frequencies are 1 kHz or 100 Hz. In preemptive RTOS mode that interrupt provides a way for the RTOS to preempt a running task and to pass control to another task.
- **PendSV** (**Pen**dable **S**er**V**ice) is an interrupt request is used by the OS to force a context switch if no other interrupt is active.
- **SVCall** (**S**uper**V**isor **Call**) is triggered by the SVC instruction and is used by the FreeRTOS to start the scheduler. This one is not used for M0.

SysTick is the default time base. Of course any other timer interrupt can be used instead. For example for low power applications I'm using a special low power timer instead. And if enabling the FreeRTOS trace facility to measure task execution time, an extra timer might be needed.

When adding FreeRTOS to a 'bare-metal' (without RTOS) application, these three interrupts need to be routed to the FreeRTOS port.

Below is an example interrupt vector table for the NXP K20 (ARM Cortex-M4) with these three FreeRTOS interrupts highlighted (**vPortSVCHandler**, **vPortPendSVHandler** and **vPortTickHandler**):

If using CMSIS compliant interrupt names, then it would be SVC_Handler, PendSV_Handler and SysTick_Handler.

```c
__attribute__ ((section (".vectortable"))) const tVectorTable __vect_ta
```

```
  /* ISR name No. Address Pri Name Description */
  &__SP_INIT, /* 0x00 0x00000000 - ivINT_Initial_Stack_Pointer */
```

```
{
  (tIsrFunc)&__thumb_startup, /* 0x01 0x00000004 - ivINT_Initial_Program
  (tIsrFunc)&Cpu_INT_NMIInterrupt, /* 0x02 0x00000008 -2 ivINT_NMI */
  (tIsrFunc)&HF1_HardFaultHandler, /* 0x03 0x0000000C -1 ivINT_Hard_Faul
  (tIsrFunc)&Cpu_ivINT_Mem_Manage_Fault, /* 0x04 0x00000010 - ivINT_Mem_
  (tIsrFunc)&Cpu_ivINT_Bus_Fault, /* 0x05 0x00000014 - ivINT_Bus_Fault *
  (tIsrFunc)&Cpu_ivINT_Usage_Fault, /* 0x06 0x00000018 - ivINT_Usage_Fau
  (tIsrFunc)&Cpu_ivINT_Reserved7, /* 0x07 0x0000001C - ivINT_Reserved7 *
  (tIsrFunc)&Cpu_ivINT_Reserved8, /* 0x08 0x00000020 - ivINT_Reserved8 *
  (tIsrFunc)&Cpu_ivINT_Reserved9, /* 0x09 0x00000024 - ivINT_Reserved9 *
  (tIsrFunc)&Cpu_ivINT_Reserved10, /* 0x0A 0x00000028 - ivINT_Reserved10
  (tIsrFunc)&vPortSVCHandler, /* 0x0B 0x0000002C - ivINT_SVCall */
  (tIsrFunc)&Cpu_ivINT_DebugMonitor, /* 0x0C 0x00000030 - ivINT_DebugMon
  (tIsrFunc)&Cpu_ivINT_Reserved13, /* 0x0D 0x00000034 - ivINT_Reserved13
  (tIsrFunc)&vPortPendSVHandler, /* 0x0E 0x00000038 - ivINT_PendableSrvR
  (tIsrFunc)&vPortTickHandler, /* 0x0F 0x0000003C - ivINT_SysTick */
  (tIsrFunc)&Cpu_ivINT_DMA0, /* 0x10 0x00000040 - ivINT_DMA0 */
  (tIsrFunc)&Cpu_ivINT_DMA1, /* 0x11 0x00000044 - ivINT_DMA1 */
  (tIsrFunc)&Cpu_ivINT_DMA2, /* 0x12 0x00000048 - ivINT_DMA2 */
  (tIsrFunc)&Cpu_ivINT_DMA3, /* 0x13 0x0000004C - ivINT_DMA3 */
  (tIsrFunc)&Cpu_ivINT_DMA_Error, /* 0x14 0x00000050 - ivINT_DMA_Error *
  (tIsrFunc)&Cpu_ivINT_Reserved21, /* 0x15 0x00000054 - ivINT_Reserved21
  (tIsrFunc)&Cpu_ivINT_FTFL, /* 0x16 0x00000058 - ivINT_FTFL */
  (tIsrFunc)&Cpu_ivINT_Read_Collision, /* 0x17 0x0000005C - ivINT_Read_C
  (tIsrFunc)&Cpu_ivINT_LVD_LVW, /* 0x18 0x00000060 - ivINT_LVD_LVW */
  (tIsrFunc)&Cpu_ivINT_LLW, /* 0x19 0x00000064 - ivINT_LLW */
  (tIsrFunc)&Cpu_ivINT_Watchdog, /* 0x1A 0x00000068 - ivINT_Watchdog */
  (tIsrFunc)&Cpu_ivINT_I2C0, /* 0x1B 0x0000006C - ivINT_I2C0 */
  (tIsrFunc)&Cpu_ivINT_SPI0, /* 0x1C 0x00000070 - ivINT_SPI0 */
  (tIsrFunc)&Cpu_ivINT_I2S0_Tx, /* 0x1D 0x00000074 - ivINT_I2S0_Tx */
  (tIsrFunc)&Cpu_ivINT_I2S0_Rx, /* 0x1E 0x00000078 - ivINT_I2S0_Rx */
  (tIsrFunc)&Cpu_ivINT_UART0_LON, /* 0x1F 0x0000007C - ivINT_UART0_LON *
  (tIsrFunc)&ASerialLdd1_Interrupt, /* 0x20 0x00000080 8 ivINT_UART0_RX_
  (tIsrFunc)&ASerialLdd1_Interrupt, /* 0x21 0x00000084 8 ivINT_UART0_ERR
  (tIsrFunc)&Cpu_ivINT_UART1_RX_TX, /* 0x22 0x00000088 - ivINT_UART1_RX_
  (tIsrFunc)&Cpu_ivINT_UART1_ERR, /* 0x23 0x0000008C - ivINT_UART1_ERR *
  (tIsrFunc)&Cpu_ivINT_UART2_RX_TX, /* 0x24 0x00000090 - ivINT_UART2_RX_
  (tIsrFunc)&Cpu_ivINT_UART2_ERR, /* 0x25 0x00000094 - ivINT_UART2_ERR *
  (tIsrFunc)&Cpu_ivINT_ADC0, /* 0x26 0x00000098 - ivINT_ADC0 */
  (tIsrFunc)&Cpu_ivINT_CMP0, /* 0x27 0x0000009C - ivINT_CMP0 */
  (tIsrFunc)&Cpu_ivINT_CMP1, /* 0x28 0x000000A0 - ivINT_CMP1 */
  (tIsrFunc)&TU1_Interrupt, /* 0x29 0x000000A4 8 ivINT_FTM0 */
  (tIsrFunc)&Cpu_ivINT_FTM1, /* 0x2A 0x000000A8 - ivINT_FTM1 */
  (tIsrFunc)&Cpu_ivINT_CMT, /* 0x2B 0x000000AC - ivINT_CMT */
  (tIsrFunc)&Cpu_ivINT_RTC, /* 0x2C 0x000000B0 - ivINT_RTC */
  (tIsrFunc)&Cpu_ivINT_RTC_Seconds, /* 0x2D 0x000000B4 - ivINT_RTC_Secon
  (tIsrFunc)&Cpu_ivINT_PIT0, /* 0x2E 0x000000B8 - ivINT_PIT0 */
  (tIsrFunc)&Cpu_ivINT_PIT1, /* 0x2F 0x000000BC - ivINT_PIT1 */
  (tIsrFunc)&Cpu_ivINT_PIT2, /* 0x30 0x000000C0 - ivINT_PIT2 */
  (tIsrFunc)&Cpu_ivINT_PIT3, /* 0x31 0x000000C4 - ivINT_PIT3 */
  (tIsrFunc)&Cpu_ivINT_PDB0, /* 0x32 0x000000C8 - ivINT_PDB0 */
  (tIsrFunc)&USB_ISR, /* 0x33 0x000000CC 0 ivINT_USB0 */
  (tIsrFunc)&Cpu_ivINT_USBDCD, /* 0x34 0x000000D0 - ivINT_USBDCD */
  (tIsrFunc)&Cpu_ivINT_TSI0, /* 0x35 0x000000D4 - ivINT_TSI0 */
  (tIsrFunc)&Cpu_ivINT_MCG, /* 0x36 0x000000D8 - ivINT_MCG */
  (tIsrFunc)&Cpu_ivINT_LPTimer, /* 0x37 0x000000DC - ivINT_LPTimer */
  (tIsrFunc)&Cpu_ivINT_PORTA, /* 0x38 0x000000E0 - ivINT_PORTA */
  (tIsrFunc)&Cpu_ivINT_PORTB, /* 0x39 0x000000E4 - ivINT_PORTB */
  (tIsrFunc)&Cpu_ivINT_PORTC, /* 0x3A 0x000000E8 - ivINT_PORTC */
  (tIsrFunc)&Cpu_ivINT_PORTD, /* 0x3B 0x000000EC - ivINT_PORTD */
  (tIsrFunc)&Cpu_ivINT_PORTE, /* 0x3C 0x000000F0 - ivINT_PORTE */
  (tIsrFunc)&Cpu_ivINT_SWI /* 0x3D 0x000000F4 - ivINT_SWI */
 }
};
```

If using the NXP Processor Expert system, then the above interrupts get automatically added to vector table by the FreeRTOS component. If using an example from FreeRTOS.org, then the vector table already has these interrupts added.

SVCall

The M3/M4/M7 ports of FreeRTOS are using a SuperVisor to start the first task. The SVC (SuperVisor Call) instruction is designed by ARM to access OS Kernel functions and device drivers. The SVCall exception is raised by the SVC assembly instruction which takes an additional argument/number, e.g.

```
SVC <number>
```

For M3/M4/M7, FreeRTOS uses the SVC in a single place: when it starts the scheduler to run the first task. The function vPortStartFirstTask() gets called at the end when you do a call to the FreeRTOS vTaskStartScheduler():

```
void vPortStartFirstTask(void) {
#if configCPU_FAMILY_IS_ARM_M4_M7(configCPU_FAMILY) /* Cortex M4/M7 */
 __asm volatile (
 " ldr r0, =0xE000ED08 \n" /* Use the NVIC offset register to locate th
 " ldr r0, [r0] \n" /* load address of vector table */
 " ldr r0, [r0] \n" /* load first entry of vector table which is the re
 " msr msp, r0 \n" /* Set the msp back to the start of the stack. */
 " cpsie i \n" /* Globally enable interrupts. */
 " svc 0 \n" /* System call to start first task. */
 " nop \n"
 );
#elif configCPU_FAMILY_IS_ARM_M0(configCPU_FAMILY) /* Cortex M0+ */
 /* With the latest FreeRTOS, the port for M0+ does not use the SVC ins
 * and does not need vPortSVCHandler() any more.
 */
 /* The MSP stack is not reset as, unlike on M3/4 parts, there is no ve
 table offset register that can be used to locate the initial stack val
 Not all M0 parts have the application vector table at address 0. */
 __asm volatile(
 " ldr r2, pxCurrentTCBConst2 \n" /* Obtain location of pxCurrentTCB. *
 " ldr r3, [r2] \n"
 " ldr r0, [r3] \n" /* The first item in pxCurrentTCB is the task top c
 " add r0, #32 \n" /* Discard everything up to r0. */
 " msr psp, r0 \n" /* This is now the new top of stack to use in the ta
 " movs r0, #2 \n" /* Switch to the psp stack. */
 " msr CONTROL, r0 \n"
 " isb \n"
 " pop {r0-r5} \n" /* Pop the registers that are saved automatically. *
 " mov lr, r5 \n" /* lr is now in r5. */
 " pop {r3} \n" /* Return address is now in r3. */
 " pop {r2} \n" /* Pop and discard XPSR. */
 " cpsie i \n" /* The first task has its context and interrupts can be
 " bx r3 \n" /* Finally, jump to the user defined task code. */
 " \n"
 " .align 4 \n"
 "pxCurrentTCBConst2: .word pxCurrentTCB"
 );
#endif
}
}
```

SVC uses an argument which could be checked in the SVCall interrupt handler. Because FreeRTOS only uses the zero, no handling is needed. Below is an implementation of the SVCHandler:

> 💡 *Unlike the 'normal' FreeRTOS ports, the McuOnEcliopse FreeRTOS port covers multiple ARM architectures in a single port. For this the configCPU_FAMILY configuration macro is used. This makes sense as most of the port is the same for M0/M3/M4/M7.*

```
__attribute__ ((naked)) void vPortSVCHandler(void) {
#if configCPU_FAMILY_IS_ARM_M4_M7(configCPU_FAMILY)  /* Cortex M4 or M7
__asm volatile (
    " ldr r3, pxCurrentTCBConst2 \n" /* Restore the context. */
    " ldr r1, [r3]               \n" /* Use pxCurrentTCBConst to get th
    " ldr r0, [r1]               \n" /* The first item in pxCurrentTCB
    /* pop the core registers */
#if configCPU_FAMILY_IS_ARM_FPU(configCPU_FAMILY)
    " ldmia r0!, {r4-r11, r14}   \n"
#else
    " ldmia r0!, {r4-r11}        \n"
#endif
    " msr psp, r0                \n"
    " mov r0, #0                 \n"
    " msr basepri, r0            \n"
#if configCPU_FAMILY_IS_ARM_FPU(configCPU_FAMILY)
#else
    " orr r14, r14, #13          \n"
#endif
    " bx r14                     \n"
    "                            \n"
    " .align 2                   \n"
    "pxCurrentTCBConst2: .word pxCurrentTCB \n"
  );
#elif configCPU_FAMILY_IS_ARM_M0(configCPU_FAMILY) /* Cortex M0+ */
  /* This function is no longer used, but retained for backward
  compatibility. */
#endif
}
```

Don’t be worried about the (GNU) ARM assembly code above: basically what it does is performing a context switch: jumping from the vPortSVCHandler() interrupt handler/context to the task described by the pxCurrentTCB.

> 💡 *pxCurrentTCB points to an RTOS structure which contains a descriptor of the task (TCB stands for Task Control Block).*

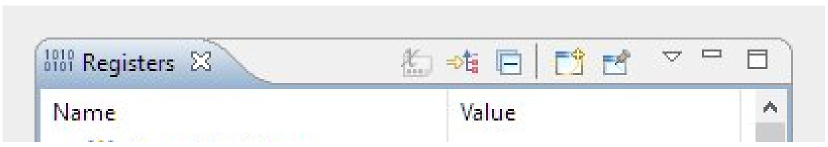
MSP and PSP

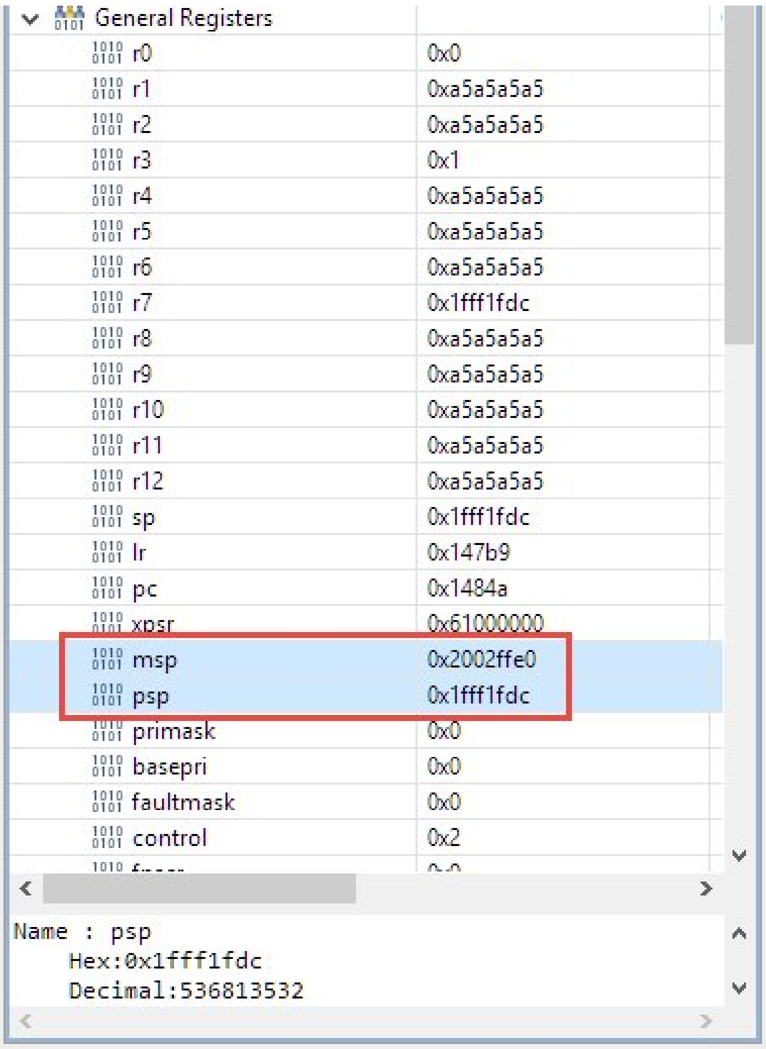
You might notice that the above handler sets the PSP (Process Stack Pointer):

```
"msr psp, r0 \n" /* Remember the new top of stack for the task. */
```

The Cortex-M has two different stack pointers:

- **MSP** (**M**ain **S**tack **P**ointer) is used during startup and during main(). Interrupts are using this stack.
- **PSP** (**P**rocess **S**tack **P**ointer) is used by the process/tasks.





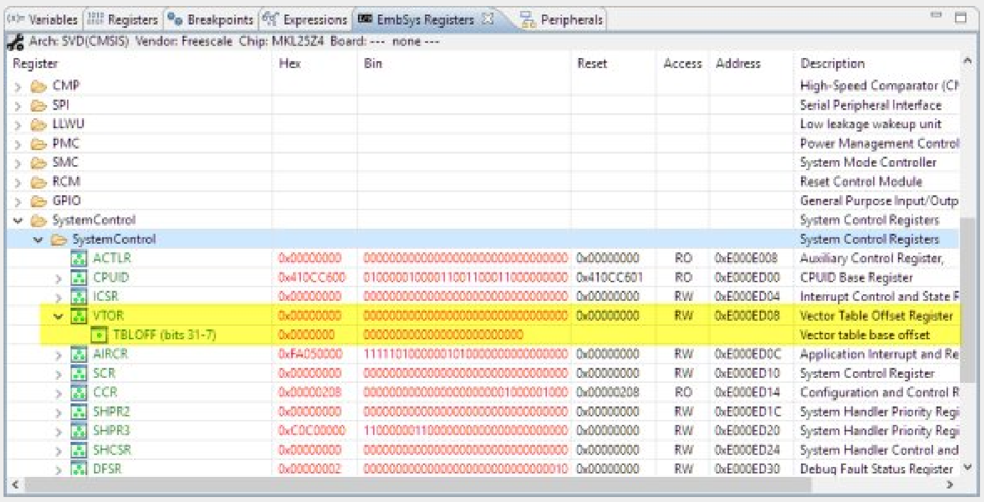
```
General Registers
r0          0x0
r1          0xa5a5a5a5
r2          0xa5a5a5a5
r3          0x1
r4          0xa5a5a5a5
r5          0xa5a5a5a5
r6          0xa5a5a5a5
r7          0x1fff1fdc
r8          0xa5a5a5a5
r9          0xa5a5a5a5
r10         0xa5a5a5a5
r11         0xa5a5a5a5
r12         0xa5a5a5a5
sp          0x1fff1fdc
lr          0x147b9
pc          0x1484a
xpsr        0x61000000
msp         0x2002ffe0
psp         0x1fff1fdc
primask     0x0
basepri     0x0
faultmask   0x0
control     0x2

Name : psp
    Hex:0x1fff1fdc
    Decimal:536813532
```

— ARM Cortex MSP and PSP Registers

The good thing with this is that the task stacks only need to count for the task stack/variables, and not include the stack needed for the interrupts. On the other side it means that the MSP stack needs to hold all the (nested) interrupt stacks.

In the above port for Cortex-M4/M7, the MSP stack pointer is reset to the reset vector stack pointer: This gives some extra stack bytes for the interrupt service routines. Because not all Cortex-M0 have the vector table at address zero and there is no dedicated register to locate the initial stack pointer, this is not used for the Cortex-M0 port.

> 💡 *To my knowledge, all NXP Cortex-M0+ parts do have the default vector table at address 0x0000'0000. It is true that the NVIC for Cortex-M0 does not implement a VTOR (Vector Table Offset Register) at 0xE000ED08. However, all the NXP Cortex-M0+ parts I'm aware of have it implemented. Not as part of NVIC, but as part of the SystemControl. So for the NXP Cortex-M0+, the same approach as for M4/M7 with VTOR and SVC could be used.*


Variables | Registers | Breakpoints | Expressions | EmbSys Registers | Peripherals

Arch: SVD(CMSIS) Vendor: Freescale Chip: MKL25Z4 Board: --- none ---

Register	Hex	Bin	Reset	Access	Address	Description
CMP						High-Speed Comparator (CI
SPI						Serial Peripheral Interface
LLWU						Low leakage wakeup unit
PMC						Power Management Control
SMC						System Mode Controller
RCM						Reset Control Module
GPIO						General Purpose Input/Outp
SystemControl						System Control Registers
SystemControl						System Control Registers
ACTLR	0x00000000	00000000000000000000000000000000	0x00000000	RO	0xE000E008	Auxiliary Control Register,
CPUID	0x410CC600	01000001000011001100011000000000	0x410CC601	RO	0xE000ED00	CPUID Base Register
ICSR	0x00000000	00000000000000000000000000000000	0x00000000	RW	0xE000ED04	Interrupt Control and State F
VTOR	0x00000000	00000000000000000000000000000000	0x00000000	RW	0xE000ED08	Vector Table Offset Register
TBLOFF (bits 31-7)	0x0000000	0000000000000000000000000				Vector table base offset
AIRCR	0xFA050000	11111010000001010000000000000000	0x00000000	RW	0xE000ED0C	Application Interrupt and Re
SCR	0x00000000	00000000000000000000000000000000	0x00000000	RW	0xE000ED10	System Control Register
CCR	0x00000208	00000000000000000000001000001000	0x00000208	RO	0xE000ED14	Configuration and Control R
SHPR2	0x00000000	00000000000000000000000000000000	0x00000000	RW	0xE000ED1C	System Handler Priority Regi
SHPR3	0xC0C00000	11000000110000000000000000000000	0x00000000	RW	0xE000ED20	System Handler Priority Regi
SHCSR	0x00000000	00000000000000000000000000000000	0x00000000	RW	0xE000ED24	System Handler Control and
DFSR	0x00000002	00000000000000000000000000000010	0x00000000	RW	0xE000ED30	Debug Fault Status Register

— *VTOR on NXP Kinetis L Family (ARM Cortex-M0+)*

PendSV

The Pendable Service interrupt is used by the RTOS to perform a context switch. Such a context switch can be originated by the RTOS or by the application with a 'yield'.

The vPortPendSVHandler() It is similar to the vPortSVCHandler(). But it does not switch from the MSP to the PSP: it performs a task context switch between different PSP values. Additionally it calls the FreeRTOS vTaskSwitchContext() which selects the highest (RTOS!) priority ready task:

> 💡 *Don't get confused by the RTOS priorities. FreeRTOS uses a system where numerically lower values mean lower task priorities. Zero is the lowest task priority in FreeRTOS. Don't get this confused with the ARM Cortex-M interrupt priority levels where zero is the *highest* urgency!*

```c
__attribute__ ((naked)) void vPortPendSVHandler(void) {
```

```
#if configCPU_FAMILY_IS_ARM_M4_M7(configCPU_FAMILY) /* Cortex M4 or M7*
```

```
    __asm volatile (
" mrs r0, psp \n"
" ldr r3, pxCurrentTCBConst \n" /* Get the location of the current TCB
```

```
    " ldr r2, [r3] \n"
```

```
#if configCPU_FAMILY_IS_ARM_FPU(configCPU_FAMILY)
```

```
" tst r14, #0x10 \n" /* Is the task using the FPU context? If so, push
```

```
    " it eq \n"
```

```
" vstmdbeq r0!, {s16-s31} \n"
```

```
  " stmdb r0!, {r4-r11, r14} \n" /* save remaining core registers */
```

```
#else
```

```
    " stmdb r0!, {r4-r11} \n" /* Save the core registers. */
```

```
#endif
```

```
    " str r0, [r2] \n" /* Save the new top of stack into the first member
    " stmdb sp!, {r3, r14} \n"
```

```
" mov r0, %0 \n"
```

```
    " msr basepri, r0 \n"
```

```
  " bl vTaskSwitchContext \n"
  " mov r0, #0 \n"
  " msr basepri, r0 \n"
  " ldmia sp!, {r3, r14} \n"
  " ldr r1, [r3] \n" /* The first item in pxCurrentTCB is the task top c
  " ldr r0, [r1] \n"
 #if configCPU_FAMILY_IS_ARM_FPU(configCPU_FAMILY)
  " ldmia r0!, {r4-r11, r14} \n" /* Pop the core registers */
  " tst r14, #0x10 \n" /* Is the task using the FPU context? If so, pop
  " it eq \n"
  " vldmiaeq r0!, {s16-s31} \n"
 #else
  " ldmia r0!, {r4-r11} \n" /* Pop the core registers. */
 #endif
  " msr psp, r0 \n"
  " bx r14 \n"
  " \n"
  " .align 2 \n"
  "pxCurrentTCBConst: .word pxCurrentTCB \n"
  ::"i"(configMAX_SYSCALL_INTERRUPT_PRIORITY)
  );
 #else /* Cortex M0+ */
  __asm volatile (
  " mrs r0, psp \n"
  " \n"
  " ldr r3, pxCurrentTCBConst \n" /* Get the location of the current TCB
  " ldr r2, [r3] \n"
  " \n"
  " sub r0, r0, #32 \n" /* Make space for the remaining low registers. *
  " str r0, [r2] \n" /* Save the new top of stack. */
  " stmia r0!, {r4-r7} \n" /* Store the low registers that are not saved
  " mov r4, r8 \n" /* Store the high registers. */
  " mov r5, r9 \n"
  " mov r6, r10 \n"
  " mov r7, r11 \n"
  " stmia r0!, {r4-r7} \n"
  " \n"
  " push {r3, r14} \n"
  " cpsid i \n"
  " bl vTaskSwitchContext \n"
  " cpsie i \n"
  " pop {r2, r3} \n" /* lr goes in r3. r2 now holds tcb pointer. */
  " \n"
  " ldr r1, [r2] \n"
  " ldr r0, [r1] \n" /* The first item in pxCurrentTCB is the task top c
  " add r0, r0, #16 \n" /* Move to the high registers. */
  " ldmia r0!, {r4-r7} \n" /* Pop the high registers. */
  " mov r8, r4 \n"
  " mov r9, r5 \n"
  " mov r10, r6 \n"
  " mov r11, r7 \n"
  " \n"
  " msr psp, r0 \n" /* Remember the new top of stack for the task. */
  " \n"
  " sub r0, r0, #32 \n" /* Go back for the low registers that are not au
  " ldmia r0!, {r4-r7} \n" /* Pop low registers. */
  " \n"
  " bx r3 \n"
  " \n"
  ".align 2 \n"
  "pxCurrentTCBConst: .word pxCurrentTCB"
  );
 #endif
 }
```

Task Yielding

As mentioned above, a task context switch can be requested by a task. For this taskYIELD() us used. This is actually a macro:

```
/**
 * task. h
 *
 * Macro for forcing a context switch.
 *
 * \defgroup taskYIELD taskYIELD
 * \ingroup SchedulerControl
 */
#define taskYIELD() portYIELD()
```

which is yet another macro:

```
#define portYIELD() vPortYieldFromISR()
```

which is implemented as

```
void vPortYieldFromISR(void) {
 /* Set a PendSV to request a context switch. */
 *(portNVIC_INT_CTRL) = portNVIC_PENDSVSET_BIT;
 /* Barriers are normally not required but do ensure the code is complet
 within the specified behavior for the architecture. */
 __asm volatile("dsb");
 __asm volatile("isb");
}
```

It sets the portNVIC_PENDSVSET_BIT to trigger the PendSV interrupt in the first statement of the function.

> 💡 *The above implementation uses data and instruction barrier instructions. The Cortex-M has several special instructions to flush the internal pipelines/caches/bus registers. Check the ARM Infocenter for more details about the need for barrier instructions.*

SysTick

The SysTick interrupt is a 24bit timer which is designed to be used by the RTOS as time base.Typical time bases are 10ms or 1ms. Basically the tick interrupt preempts any running task and passes control to the scheduler so it can increment its internal tick counter and perform a context switch as needed. Below is a (simplified) version of the tick handler interrupt routine:

```
void vPortTickHandler(void) {
  /* The SysTick runs at the lowest interrupt priority, so when this in
    executes all interrupts must be unmasked.  There is therefore no ne
    save and then restore the interrupt mask value as its value is alre
    known. */
  portDISABLE_INTERRUPTS();   /* disable interrupts */
  if (xTaskIncrementTick()!=pdFALSE) { /* increment tick count */
    traceISR_EXIT_TO_SCHEDULER();
    taskYIELD();
  }
  portENABLE_INTERRUPTS(); /* re-enable interrupts */
}
```

💡 *As a side note: In FreeRTOS non-preemptive (cooperative, configUSE_PREEMPTION set to zero) mode the call to xTaskIncrementTick() will always return pdFALSE, so the tick interrupt will not cause a context switch.*

Notice the pair of *portDISABLE_INTERRUPTS()* and *portENABLE_INTERRUPTS()*: some ports will not completely turn off interrupts, more about later.

configKERNEL_INTERRUPT_PRIORITY

You might have noticed that comment inside vPortTickHandler() from above:

```
/* The SysTick runs at the lowest interrupt priority, ... */
```

In FreeRTOS, the kernel and its interrupts are running with the lowest interrupt priority/urgency.

FreeRTOS uses the macro **configKERNEL_INTERRUPT_PRIORITY** in FreeRTOSConfig.h for the interrupts used by the kernel port layer. Below is an implementation for a system with 4 interrupt priority bits.

```
#define configPRIO_BITS 4 /* 4 bits/16 priority levels on ARM Cortex M4
/* The lowest interrupt priority that can be used in a call to a "set pr
#define configKERNEL_INTERRUPT_PRIORITY (configLIBRARY_LOWEST_INTERRUPT_
```

Note that configKERNEL_INTERRUPT_PRIORITY is in the form the NVIC expects it: already *shifted* to the left.

As noted above, the important point to note is that configKERNEL_INTERRUPT_PRIORITY is set to the *lowest* urgency interrupt level in the system. So this means the interrupts of the kernel (SysTick and context switches) run with the lowest possible urgency. This makes sense as the kernel should not block or interrupt the other interrupts in the system. The other reason is that this approach keeps the kernel simple, as it does not have to deal with nested interrupts ‘below’ the urgency level of the kernel.

PendSV and **SysTick** interrupt priorities are configured by the FreeRTOS at scheduler start:

```
/* Interrupt priorities used by the kernel port layer itself. These are
 to all Cortex-M ports, and do not rely on any particular library funct
#define configKERNEL_INTERRUPT_PRIORITY (configLIBRARY_LOWEST_INTERRUPT

#define portNVIC_SYSPRI3 ((volatile unsigned long*)0xe000ed20) /* syste
#define portNVIC_SYSTICK_PRI (((unsigned long)configKERNEL_INTERRUPT_PR
#define portNVIC_PENDSV_PRI (((unsigned long)configKERNEL_INTERRUPT_PRI

*(portNVIC_SYSPRI3) |= portNVIC_PENDSV_PRI; /* set priority of PendSV i
*(portNVIC_SYSPRI3) |= portNVIC_SYSTICK_PRI; /* set priority of SysTick
```

The **SVCall** priority (only used on M4/M7) is *not* touched. Out of reset, this priority is zero (highest urgency for the ARM core). The SVCall interrupt is only used once at the startup of the RTOS (see above).

> 💡 *The reason why the SVCall priority is still at the highest urgency (0 value) is when executing the SVC instruction, the SVCall interrupt shall not be masked (disabled). If SVCall would be at the same priority of the SysTick, then a SysTick might happen earlier, and then using SVC would cause a hard fault. Leaving it at 0 (highest interrupt urgency) ensures that it is not masked by a running PendSV or SysTick interrupt, and that it is not masked by the BASEPRI register.*

Calling RTOS API Functions from ISR

It is not uncommon to call RTOS functions (e.g. to set/clear a semaphore) from an interrupt service routine (ISR). There is one very important rule with FreeRTOS:

> *Only RTOS API functions ending with “FromISR” are allowed to be called from an interrupt service routine.*

The “FromISR” functions are ‘interrupt safe, for example xSemaphoreGiveFromISR().

```
xSemaphoreGiveFromISR
 (
 SemaphoreHandle_t xSemaphore,
 signed BaseType_t *pxHigherPriorityTaskWoken
 )
```

If pxHigherPriorityTaskWoken returns pdTRUE, then the API call caused a higher priority task to be unlocked. In that case the interrupt should call a yield at the end of the service routine to perform a context switch.

The following example implements a timer interrupt which gives a semaphore:

```
/* Timer ISR */
void vTimerISR(void) {
  signed BaseType_t xHigherPriorityTaskWoken;

  xHigherPriorityTaskWoken = pdFALSE;
  xSemaphoreGiveFromISR(xSemaphore, &xHigherPriorityTaskWoken);
  /* If xHigherPriorityTaskWoken was set to true you we should yield. */
  portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Kernel Critical Sections

FreeRTOS stores internally information, things like the list of tasks and other data used by the kernel. Because this data is accessed both by tasks through FreeRTOS API calls (e.g. accessing a semaphore) and by the kernel itself (trigged through a SysTick timer interrupt), access to this data needs to be reentrant and protected with a critical section.

Remember the simple critical section we saw earlier for the tick handler:

```
void vPortTickHandler(void) {
  /* The SysTick runs at the lowest interrupt priority, so when this in
    executes all interrupts must be unmasked.  There is therefore no ne
    save and then restore the interrupt mask value as its value is alre
    known. */
  portDISABLE_INTERRUPTS();   /* disable interrupts */
  if (xTaskIncrementTick()!=pdFALSE) { /* increment tick count */
    traceISR_EXIT_TO_SCHEDULER();
    taskYIELD();
  }
  portENABLE_INTERRUPTS(); /* re-enable interrupts */
}
```

Inside xTaskIncrementTick() the kernel is running: checking if there is another task to schedule in preemptive mode. In cooperative mode it will return pdFALSE

So now you could think that the RTOS is disabling all interrupts during its critical sections. This is actually true for some FreeRTOS ports (e.g. for ARM Cortex-M0/M0+). Disabling interrupts for critical sections increases interrupt latency and should be avoided as much as possible, and such a critical section should be as small as possible. While FreeRTOS makes every effort to keep such critical sections as small and fast as possible, they are certainly longer than a few CPU instructions. The good news is that for the Cortex-M3/M4/M7 ports, not all interrupts are disabled: FreeRTOS is taking advantage of the BASEPRI register (see Part 1).

They are behind yet another macro as below:

```
#define portDISABLE_INTERRUPTS() portSET_INTERRUPT_MASK()
#define portENABLE_INTERRUPTS()  portCLEAR_INTERRUPT_MASK()
```

The implementation depends on the Cortex-M core used:

```
#if configCPU_FAMILY_IS_ARM_M4_M7(configCPU_FAMILY) /* Cortex M4/M7 */
  /*
  * Set basepri to portMAX_SYSCALL_INTERRUPT_PRIORITY without effecting
  * registers. r0 is clobbered.
  */
  #define portSET_INTERRUPT_MASK() \
   __asm volatile \
   ( \
   " mov r0, %0 \n" \
   " msr basepri, r0 \n" \
   : /* no output operands */ \
   :"i"(configMAX_SYSCALL_INTERRUPT_PRIORITY) /* input */\
   :"r0" /* clobber */ \
   )
  /*
  * Set basepri back to 0 without effective other registers.
  * r0 is clobbered.
  */
  #define portCLEAR_INTERRUPT_MASK() \
   __asm volatile \
   ( \
   " mov r0, #0 \n" \
   " msr basepri, r0 \n" \
   : /* no output */ \
   : /* no input */ \
   :"r0" /* clobber */ \
   )
#elif configCPU_FAMILY_IS_ARM_M0(configCPU_FAMILY) /* Cortex-M0+ */
  #define portSET_INTERRUPT_MASK()   __asm volatile("cpsid i")
  #define portCLEAR_INTERRUPT_MASK() __asm volatile("cpsie i")
#endif
```

You might wonder why in above vPortTickHandler() it enables all interrupts with portENABLE_INTERRUPTS(), and why it does not need to store the current BASEPRI level? Because the tick interrupt has the lowest interrupt value, it only can run if all interrupts are enabled. So BASEPRI must be zero, so it does not make sense to save and restore a known value :-).

So on Cortex-M0 it globally disables all interrupts, while on M4/M7 FreeRTOS only disables the interrupts up to the

```
configMAX_SYSCALL_INTERRUPT_PRIORITY
```

level (more about this FreeRTOS configuration setting later). So using an ARM Cortex-M3/M4/M7 not only makes sense from the performance point of view: with the BASEPRI NVIC hardware it is possible that some interrupts are *not* blocked by the Kernel and their interrupt latency is not impacted by the RTOS. However, the range of interrupt levels need to be carefully selected, more about this later as well.

Task Critical Sections

So far we had a look a the port critical section handling for the tick interrupt. Another aspect is how the Kernel implements critical sections inside the RTOS API.

For example below is an example how FreeRTOS implements a critical section inside:

```
    void* xQueueGetMutexHolder( QueueHandle_t xSemaphore )
    {
    void *pxReturn;

        /* This function is called by xSemaphoreGetMutexHolder(), and s
        be called directly.  Note:  This is a good way of determining i
        calling task is the mutex holder, but not a good way of determi
        identity of the mutex holder, as the holder may change between
        following critical section exiting and the function returning.
        taskENTER_CRITICAL();
        {
            if( ( ( Queue_t * ) xSemaphore )->uxQueueType == queueQUEUE
            {
                pxReturn = ( void * ) ( ( Queue_t * ) xSemaphore )->pxM
            }
            else
            {
                pxReturn = NULL;
            }
        }
        taskEXIT_CRITICAL();

        return pxReturn;
    } /*lint !e818 xSemaphore cannot be a pointer to const because it i
```

As shown above, FreeRTOS uses the following:

```
taskENTER_CRITICAL();
/* critical section here */
taskEXIT_CRITICAL();
```

They are macros, and the implementation is part of the port for a microcontroller:

```
#define portENTER_CRITICAL() vPortEnterCritical()
#define portEXIT_CRITICAL() vPortExitCritical()
```

Below is the implementation for ARM Cortex-M (simplified):

```
void vPortEnterCritical(void) {
/*
 * Disable interrupts before incrementing the count of critical section
 * The nesting count is maintained so we know when interrupts should be
 * re-enabled. Once interrupts are disabled the nesting count can be ac
 * directly. Each task maintains its own nesting count.
 */
 portDISABLE_INTERRUPTS();
 uxCriticalNesting++;
}
/*-----------------------------------------------------------*/
void vPortExitCritical(void) {
  /* Interrupts are disabled so we can access the nesting count directl
  * nesting is found to be 0 (no nesting) then we are leaving the criti
  * section and interrupts can be re-enabled.
  */
  uxCriticalNesting--;
  if (uxCriticalNesting == 0) {
    portENABLE_INTERRUPTS();
  }
}
```

The two functions are using a nesting counter to allow nesting of critical sections. They are using

```
portDISABLE_INTERRUPTS();
```

and

```
portENABLE_INTERRUPTS();
```

to disable/enable the interrupts we have seen above. So here again: FreeRTOS is masking all interrupts on Cortex-M0, but leaves some interrupt levels enabled based on the configMAX_SYSCALL_INTERRUPT_PRIORITY.

configMAX_SYSCALL_INTERRUPT_PRIORITY

We already saw earlier the configMAX_SYSCALL_INTERRUPT_PRIORITY setting of FreeRTOS in FreeRTOSConfig.h. On Cortex-M3/M4/M7 it is used to mask interrupts using the BASEPRI register.

In FreeRTOSConfig.h it present as a macro:

```
/* !!!! configMAX_SYSCALL_INTERRUPT_PRIORITY must not be set to zero !!!
See http://www.FreeRTOS.org/RTOS-Cortex-M3-M4.html. */
#define configMAX_SYSCALL_INTERRUPT_PRIORITY (configLIBRARY_MAX_SYSCALL_
```

> 💡 *because the MAX_SYSCALL can be confusing, there is a alias configMAX_API_CALL_INTERRUPT_PRIORITY which much better reflect up to which interrupt level the FreeRTOS API can be called from an interrupt. configMAX_API_CALL_INTERRUPT_PRIORITY is a new name for configMAX_SYSCALL_INTERRUPT_PRIORITY that is used by newer ports only. The two are equivalent.*

The macro *configMAX_SYSCALL_INTERRUPT_PRIORITY* uses *configPRIO_BITS* (the number of piority bits available, see Part 1), and *configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY:*

```
/* The highest interrupt priority that can be used by any interrupt serv
routine that makes calls to interrupt safe FreeRTOS API functions. DO NO
INTERRUPT SAFE FREERTOS API FUNCTIONS FROM ANY INTERRUPT THAT HAS A HIGH
PRIORITY THAN THIS! (higher priorities are lower numeric values. */
#define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY 5
```

It is used in the port to mask interrupts as below:

```
#define portSET_INTERRUPT_MASK() \
 __asm volatile \
 ( \
 " mov r0, %0 \n" \
 " msr basepri, r0 \n" \
 : /* no output operands */ \
 :"i"(configMAX_SYSCALL_INTERRUPT_PRIORITY) /* input */\
 :"r0" /* clobber */ \
 )
/*
* Set basepri back to 0 without effective other registers.
* r0 is clobbered.
*/
#define portCLEAR_INTERRUPT_MASK() \
 __asm volatile \
 ( \
 " mov r0, #0 \n" \
 " msr basepri, r0 \n" \
 : /* no output */ \
 : /* no input */ \
 :"r0" /* clobber */ \
 )
```

Because the kernel only masks out interrupts up and equal to configMAX_SYSCALL_INTERRUPT_PRIORITY numerical value, FreeRTOS API calls from interrupts can only be called from interrupts *numerically equal or higher* (lower urgency) than configMAX_SYSCALL_INTERRUPT_PRIORITY (remember: ARM NVIC uses zero as the highest urgency level).

Let’s assume a system with 2 priority bits on a Cortex-M3/M4/M7:

```
#define configPRIO_BITS 2 /* prios: 0, 1, 2, and 3 */
```

The lowest priority is 3

```
#define configLIBRARY_LOWEST_INTERRUPT_PRIORITY   3
```

So the FreeRTOS Kernel (PendSV and Systick) will run on that level which is 0xC0:

```
#define configKERNEL_INTERRUPT_PRIORITY    (configLIBRARY_LOWEST_INTERRU
```

Let’s set the ‘Max API Call Interrupt level’ to 2:

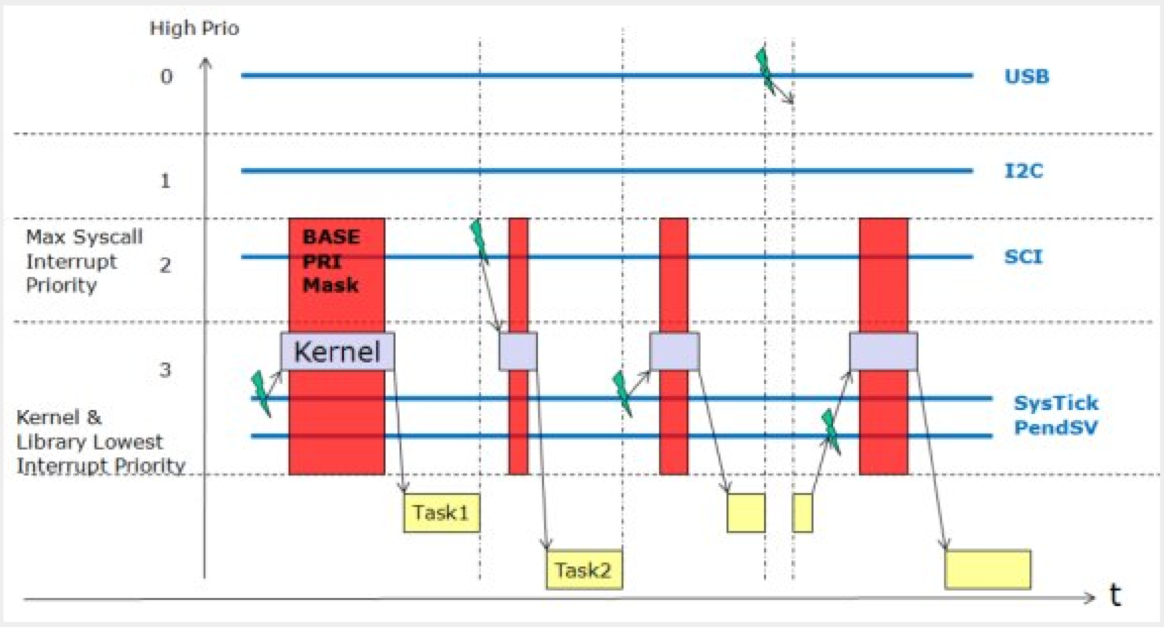
```
#define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY     2
```

That means the kernel critical section (BASEPRI) is set to 0x80:

```
#define configMAX_SYSCALL_INTERRUPT_PRIORITY      (configLIBRARY_MAX_SYS
```

I configure the rest of my interrupts to, and get the following NVIC priorities:

- USB: 0 (0x00)
- I2C: 1 (0x40)
- BASEPRI: 0x80 (all interrupts with NVIC values >= 0x80 are masked)
- SCI: 2 (0x80)
- SysTick: 3 (0xC0)
- PendSV: 3 (0xC0)


High Prio
0
1
Max Syscall Interrupt Priority
2
3
Kernel & Library Lowest Interrupt Priority
BASE PRI Mask
Kernel
Task1
Task2
USB
I2C
SCI
SysTick
PendSV
t

— FreeRTOS and Interrupts

The FreeRTOS kernel through SysTick and PendSV runs on the lowest interrupt urgency. Whenever the FreeRTOS is creating a critical section, it writes 0x80 to the BASEPRI mask register, effectively blocking any interrupts with NVIC value 0x80 or greater.

Kernel and ISRs calling RTOS API

Any interrupt (SCI in the above case) with a priority value equal or higher than BASEPRI can call FromISR() FreeRTOS API functions. The ability of an interrupt service routine to use FreeRTOS API functions comes with the price that the RTOS is blocking that interrupt in the kernel critical sections. All the interrupts with NVIC priority values lower (higher urgency) than the BASEPRI value are not blocked by the RTOS.

Extract from http://www.freertos.org/a00110.html#kernel_priority: which applies to the Cortex-M3/M4/M7:

> *"configKERNEL_INTERRUPT_PRIORITY sets the interrupt priority used by the RTOS kernel itself. configMAX_SYSCALL_INTERRUPT_PRIORITY sets the highest interrupt priority from which interrupt safe FreeRTOS API functions can be called.*
>
> *A full interrupt nesting model is achieved by setting configMAX_SYSCALL_INTERRUPT_PRIORITY above (that is, at a higher priority level) than configKERNEL_INTERRUPT_PRIORITY. This means the FreeRTOS kernel does not completely disable interrupts, even inside critical sections. Further, this is achieved without the disadvantages of a segmented kernel architecture. Note however, certain microcontroller architectures will (in hardware) disable interrupts when a new interrupt is accepted – meaning interrupts are unavoidably disabled for the short period between the hardware accepting the interrupt, and the FreeRTOS code re-enabling interrupts.*
>
> *Interrupts that do not call API functions can execute at priorities above configMAX_SYSCALL_INTERRUPT_PRIORITY and therefore never be delayed by the RTOS kernel execution."*

Of course not every interrupt service routine needs FreeRTOS API functions: they can use a NVIC priority numerically lower than the BASEPRI value and are not affected by the RTOS critical section. However, if an interrupt service routine *is* using RTOS API calls (e.g. dealing with a semaphore or using FreeRTOS queues or events), then the NVIC priority of it has to be numerically equal or higher (lower urgency) than the BASEPRI value.

> 💡 *It is highly recommended to keep the configASSERT() in FreeRTOS enabled, as these checks are catching many wrong settings. Failure to configure the Cortex-M priorities in alignment how the RTOS is using it can cause strange and sporadic failure of the system!*

What about systems without BASEPRI NVIC hardware support as the Cortex-M0+? http://www.freertos.org/a00110.html#kernel_priority says:

> *"For ports that only implement configKERNEL_INTERRUPT_PRIORITY configKERNEL_INTERRUPT_PRIORITY sets the interrupt priority used by the RTOS kernel itself. Interrupts that call API functions must also execute at this priority. Interrupts that do not call API functions can execute at higher priorities and therefore never have their execution delayed by the RTOS kernel activity (within the limits of the hardware itself)."*

Here I think this is not correct for the Cortex-M0+ or maybe misleading: As the port for Cortex-M0 blocks all interrupts in the RTOS critical section, they are indeed delayed by the RTOS kernel activity. On the other side as they are all blocked and affected, they can use any priority. But any ISR using FreeRTOS API calls has to use FromISR() API variants.

System Startup and Interrupts

In FreeRTOS, the scheduler gets started with a call to vTaskStartScheduler:

```
void vTaskStartScheduler(void);
```

By default on ARM Cortex-M, interrupts are disabled out of reset. Typically they get enabled by the startup code on some systems. In most system they get enabled in or after main().

With using FreeRTOS, I recommend that interrupts are disabled during system initialization. FreeRTOS will enable interrupts inside vTaskStartScheduler().

It is safe to call FreeRTOS API functions before the scheduler is started, see point 3 in http://www.freertos.org/FAQHelp.html:

> *"If a FreeRTOS API function is called before the scheduler has been started then interrupts will deliberately be left disabled and not re-enable again until the first task starts to execute. This is done to protect the system from crashes caused by interrupts attempting to use FreeRTOS API functions during system initialization, before the scheduler has been started, and while the scheduler may be in an inconsistent state.*

> *Do not alter the microcontroller interrupt enable bits or priority flags using any method other than calls to taskENTER_CRITICAL() and taskEXIT_CRITICAL(). These macros keep a count of their call nesting depth to ensure interrupts become enabled again only when the call nesting has unwound completely to zero. Be aware that some library functions may themselves enable and disable interrupts."*

FreeRTOS and Subpriorities?

I have not (yet) used FreeRTOS in a system with subpriorites. http://www.freertos.org/RTOS-Cortex-M3-M4.html says:

> *"It is recommended to assign all the priority bits to be preempt priority bits, leaving no priority bits as subpriority bits. Any other configuration complicates the otherwise direct relationship between the configMAX_SYSCALL_INTERRUPT_PRIORITY setting and the priority assigned to individual peripheral interrupts. "*

I have made some experiments, and with careful settings of priority values and macros with a few tweaks it seems to be possible to run FreeRTOS in a system with subpriorities. I have not made extensive tests, and currently I don't have the need to run it with subpriorities. For sure it is (not yet?) supported out of the box, but certainly doable. And I'm sure if there is demand for it, it can get added to FreeRTOS.

Summary

What I started writing on the topic of Cortex-M interrupts and FreeRTOS, I thought I could cover it in one article. Now it ended up in three, with part three much larger than I thought. While collecting all the information and reading many other articles, I learned a lot new things and even ended up tweaking the current McuOnEclipse Processor Expert port for Cortex-M0+ (I was still using SVCall, now I removed the need for it :-)).

The key takeaways are:

- The FreeRTOS kernel uses 2-3 interrupts, depending on the core: **SysTick** is used as time base, **PendSV** for context switches and **SVCall** on Cortex-M3/4/7 to start the scheduler.
- SysTick and PendSV are configured for lowest urgency: the **RTOS runs with the lowest urgency level**.
- FreeRTOS task priorities start with **0 as the lowest urgent RTOS task priority**, while the **ARM NVIC is using zero as the highest urgent interrupt priority**.
- The tasks are using the **PSP** (Process Stack Pointer), while the interrupts are using the

MSP (Main Stack Pointer). On the Cortex-M3/4/7 port, the MSP is set back to the reset stack pointer at scheduler start.
- The FreeRTOS kernel needs to use a **critical section** to protect its internal data structures.
- On **Cortex-M0**, the critical section of the Kernel masks **all** interrupts
- On **Cortex-M3/4/7**, the critical section of the kernel **does not mask all interrupts**: only masks interrupts up and equal to **configMAX_SYSCALL_INTERRUPT_PRIORITY** using the NVIC BASEPRI. Interrupts with NVIC values lower than BASEPRI (higher urgency) are not affected by the Kernel.
- **Only interrupts with NVIC numerical values greater-equal than configMAX_SYSCALL_INTERRUPT_PRIORITY/BASEPRI are allowed to use the RTOS API.**
- Interrupt service routines calling the RTOS API have to use the interrupt safe **FromISR()** variants.
- **Subpriorities** would be possible, but not supported out-of-the-box in FreeRTOS V9.0.0.

Let me know if I have missed anything or if something needs further clarification. I hope with this it makes it easier for you to use FreeRTOS with the ARM Cortex-M interrupt system.

Happy FreeRTOSing 🙂

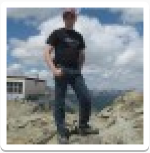
LINKS

- ARM Cortex-M, Interrupts and FreeRTOS: Part 1
- ARM Cortex-M, Interrupts and FreeRTOS: Part 2
- ARM Cortex-M, Interrupts and FreeRTOS: Part 3
- FreeRTOS operating system: http://www.freertos.org/
- Running FreeRTOS on ARM Cortex-M core: http://www.freertos.org/RTOS-Cortex-M3-M4.html
- Wikipedia on ARM Cortex-M: https://en.wikipedia.org/wiki/ARM_Cortex-M
- *The Definitive Guide to the ARM Cortex-M0 and Cortex-M0+ Processors*; 2nd Edition; Joseph Yiu; Newnes; 784 pages; 2015; ISBN 978-0128032770
- *The Definitive Guide to the ARM Cortex-M3 and Cortex-M4 Processors*; 3rd Edition; Joseph Yiu; Newnes; 600 pages; 2013; ISBN 978-0124080829
- Interrupt Latency on Cortex-M: https://community.arm.com/docs/DOC-2607
- Cutting throught the Confusion: http://embeddedgurus.com/state-space/2014/02/cutting-through-the-confusion-with-arm-cortex-m-interrupt-priorities/

Pinterest Pocket WhatsApp Telegram

Loading...

This entry was posted in **ARM**, **CPU's**, **Embedded**, **FreeRTOS**, **Freescale**, **gcc**, **Kinetis**, **LPC**, **LPC**, **NXP** and tagged **arm gcc**, **Assembly**, **FRDM-K64F**, **FreeRTOS**, **Freescale**, **gnu gcc**, **Interrupts**, **software**, **technology** by **Erich Styger**. Bookmark the **permalink**.



About Erich Styger

Embedded is my passion....

View all posts by Erich Styger →

46 THOUGHTS ON "ARM CORTEX-M INTERRUPTS AND FREERTOS: PART 3"

Pingback: ARM Cortex-M, Interrupts and FreeRTOS: Part 1 | MCU on Eclipse

Pingback: ARM Cortex-M, Interrupts and FreeRTOS: Part 2 | MCU on Eclipse

Liviu Ionescu (ilg) on **August 28, 2016 at 23:56** said:

For the record, one major comment related to FreeRTOS interrupt critical sections: the current implementation, which uses a counter to keep track of nested critical sections, is unsafe in certain conditions, which mean the critical section is shortened and part of it runs with interrupts enabled.

The scenario is the following: a binary library, which uses its own critical section management code (without calling FreeRTOS code), disables interrupts or sets BASEPRI to a certain value, then invokes a callback, and on this callback the user invokes the critical section macros; when the critical section ends, because the FreeRTOS counter returns to zero, the interrupts are prematurely enabled, end the rest of the callback plus possible some code in the library, runs unprotected.

The only safe solution to implement nested critical sections, that do not interfere with other libraries or applications, is to save the current status when entering the critical section and to restore it when the critical section exits.

Unfortunately the FreeRTOS author does not acknowledge this problem and insists his implementation has no problems.

The µOS++/CMSIS++ implementation does not suffer from this problem:

http://micro-os-plus.github.io/reference/cmsis-plus/classos_1_1rtos_1_1interrupts_1_1critical__section.html

(I know I already said this with other occasions, but I consider this a “critical” aspect of RTOSes, which must be clearly understood).

★ Like

Reply ↓

Erich Styger
on **August 29, 2016 at 08:21** said:

Hi Liviu,
mixing critical section handling or interrupt grouping (as the default STM libraries do) is always problematic. And using a binary library which I cannot rebuild is something I avoid too as it exactly can cause such problems. To me the current FreeRTOS implementation is ok: for performance reasons and there is still the option to change that by the user if desired (I hope my article might help others on this). And I agree with you: these are ‘critical’ aspects of an RTOS (or any application dealing with interrupts) and must be clearly understood.
Erich

★ Like

Reply ↓

Liviu Ionescu (ilg)
on **August 29, 2016 at 08:45** said:

> To me the current FreeRTOS implementation is ok:
… and there is still the option to change that by the user if desired

well, yes and no.

yes because technically the solution is closer than we think, the famous macros are already there, being used by the xXxxFromISR() functions:

```
UBaseType_t uxSavedInterruptStatus;
uxSavedInterruptStatus = portSET_INTERRUP
{
        ...
}
portCLEAR_INTERRUPT_MASK_FROM_ISR( uxSave
```

no because bad habits die hard; once used to complacent solutions like the one in FreeRTOS, it is harder to convince developers to write even slightly more code, for the simple reason that "it works for me". 🙁

and here is where teachers role intervene, in trying to educate developers to get the "good habits".

★ Like

Reply ↓

Liviu Ionescu (ilg) on **August 29, 2016 at 00:09** said:

My second comment is about the PendSV_Handler: it does not need to be that complicated. And it does not need to be written in assembly at that large extent; the only parts that really need to be in assembly are the parts used to save/restore the registers, the rest can be very well be written in C/C++.

As such, another variant of the PendSV_Handler may look like:

```
void
__attribute__ ((section(".after_vectors"), naked, us
PendSV_Handler (void)
{
  // The naked attribute and the push/pop are used t
  // the function entry/exit code; be sure other reg
  // used in the assembly parts.
  asm volatile ("push {lr}");

  // The whole mystery of context switching, in one 
  port::scheduler::restore_from_stack (
      port::scheduler::switch_stacks (port::schedule
  
  asm volatile ("pop {pc}");
}
```

Also the use of the SVC_Handler is not mandatory, the first thread can be invoked using simpler code.

For those interested in details, the full Cortex-M specific context switching code is available from:

https://github.com/micro-os-plus/micro-os-plus-iii/blob/xpack/src/rtos/os-core.cpp

★ Like

Reply ↓

Erich Styger
on **August 29, 2016 at 08:35** said:

Hi Liviu,
thanks, I was scratching my head as well why the FreeRTOS ports for M3/M4/M7 still use SVCall. About how much is in assembly and if things can be mostly in C: I prefer C too, on the other hand using assembly code in such small (and critical) parts help to rule out possible problems between different compilers: using assembly is less portable, but avoids that one of the many compilers break things (at least this has been my experience dealing with many non-GNU compilers in the past).

★ Like

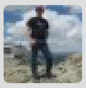
Reply ↓

Andreas Gustafsson on September 2, 2016 at 15:14 said:

Thank you for a very helpful article. This part confused me, though: “FreeRTOS API calls from interrupts can only be called from interrupts *numerically equal or lower* than configMAX_SYSCALL_INTERRUPT_PRIORITY”. Shouldn’t this be “numerically equal or higher” (i.e., of lower urgency)?

★ Liked by 1 person

Reply ↓



Erich Styger
on September 2, 2016 at 16:06 said:

Hi Andreas,
yes, you are absolutely right, I have fixed it now.
Thanks!
Erich

★ Like

Reply ↓

HIROSHI SAKURAI on September 8, 2016 at 04:23 said:

Great article. It clarified my confusion about FreeRTOS interrupt levels.

There are some typos.
– But it does not does not switch
– I that case
– BASEPRI can call call
– It is save to call

★ Like

Reply ↓

Erich Styger
on **September 8, 2016 at 05:47** said:

Thank you very much for finding these, they are fixed now.

★ Like

Reply ↓

Pingback: McuOnEclipse Components: 30-Oct-2016 Release | MCU on Eclipse

jm on **August 23, 2017 at 11:08** said:

Hi Eric big fan of yours but I am a bit confused

So imagine our interrupt is an UART interrupt that whatever reason it blocks. If you have the interrupt separated from the RTOS kernel this means it would block the entire RTOS. Then if you have the interrupt inside the kernel so that RTOS is aware of it does that means if the interrupt blocks the RTOS can still switch tasks and do stuff? I say this because of the Kernel interrupt being the lowest priority one so this is where I am confused. If the kernel interrupt is the lowest priority one does that means the RTOS will block too ?

What is the point of having an interrupt inside the RTOS (RTOS awareness) instead of having the interrupt separate from the kernel?

★ Like

Reply ↓

Erich Styger
on **August 23, 2017 at 11:19** said:

If your UART interrupt blocks (for whatever time), it will block any other interrupts with the same or lower urgency (higher NVIC numbers). So only interrupts with higher urgency will be able to interrupt your UART interrrupt.

As the RTOS interrupt (SysTick) has the lowest priority, no context switch/etc will happen, and the RTOS is blocked too: no task switch/etc will happen.
Having the RTOS running with the lowest priority sounds confusing, but makes a lot of sense: the RTOS itself is not the most important thing in the application, it is actually the least important thing: your interrupts (e.g. UART) are more important and needs to be handled first.
Good question, I usually ask that at exams 🙂
The point of the RTOS to use configMAX_SYSCALL_INTERRUPT_PRIORITY is to separate the interrupts which can interrupt the Kernel from the ones which cannot do this. It greatly simplifies the kernel and the way it has to protect its own data structures with a critical section. Creating a critical section does not come for free and takes runtime. The advantage of this is that the interrupts 'outside' that Kernel band do not need to create a critical section and are not affected by the latency caused by the kernel, so they can run with full speed and with the least overhead.
The advantage of the interrupts in that 'kernel range' is that they can use RTOS functions, but using these functions need some overhead by the kernel critical sections. And to make the critical section by the kernel simpler and faster, it will create a critical which ensures it cannot be interrupted by interrupts within that kernel interrupt band.
I hope this helps?
Erich

★ Like

Reply ↓

jm
on **August 24, 2017 at 09:05** said:

Nice I understand now. It all comes to play; where when why and how you will design critical sections; to be placed outside kernel (having almost no overhead to process) or inside the kernel playing with the RTOS functions as an advantage, having to process kernel overhead as a disadvantage.

Thank you.

★ Like

Reply ↓

Andy Blue on **October 15, 2017 at 17:34** said:

Hi Erich,
I don't know why that I saw the great article so late. Thanks for contribution.
Here are some comments.
1. Partioning of interrupt priorities/urgencies between the application and the RTOS
Would it is typo of Partitioning ?
2. It seems that lot of "M4/M7 ports" referred in the article really means of "M3/M4/M7 ports"
3. "As a side note: In FreeRTOS non-preemptive (cooperative, configUSE_PREEMPTION set to zero) mode the call to xTaskIncrementTick() will always return pdTRUE, so the tick interrupt will not cause a context switch."
In FreeRTOS non-preemptive mode the call to xTaskIncrementTick() will always return pdFALSE, instead of pdTRUE.
4. "and gete the following NVIC priorities:"
gete ? looks another typo of get
5.
USB: 0 (0x00)
I2C: 1 (0x40)
SCI: 2 (0x80)
BASEPRI: 0x80
SysTick: 3 (0xC0)
PendSV: 3 (0xC0)

suggest to swap the line for
SCI: 2 (0x80)
BASEPRI: 0x80
as "BASEPRI: 0x80" would make the boundary for the interrupts can be preempted or not.

★ Like

Reply ↓

Erich Styger
on **October 15, 2017 at 19:13** said:

Hi Andy,
and many thanks for providing such detailed feedback. You have read that article in deep details, very well done!!!
1) Yes, typo, fixed.
2) I have not used much the M3, so I appologize for that. Added M3 in several places.
3) Correct, yet again I had my memory in ~memory mode 🙂
4) yes, typo, fixed.
5) Absolutely, makes more sense, changed now.

thanks again!
Erich

★ Like

Reply ↓

Andy Blue on **October 15, 2017 at 18:11** said:

Regarding implementation of critical section.
Certainly I agree Liviu Ionescu more. The implementation of enter and exit critical section in FreeRTOS is really bad. In general it prevents the developer to set BASEPRI to any other value than configMAX_SYSCALL_INTERRUPT_PRIORITY. Following Liviu Ionescu's prompt, I more like to implement the critical section this way:

#define Open_Critical_Section() { BaseType_t uxSavedMaskLevel = portGET_INTERRUPT_MASK_LEVEL();

#define Enter_Critical() portRAISE_INTERRUPT_MASK_LEVEL(configMAX_SYSCALL_INTERRUPT_PRIORITY);

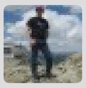
#define Exit_Critical() portLOWER_INTERRUPT_MASK_LEVEL(uxSavedMaskLevel);

#define Close_Critical_Section() }

portRAISE_INTERRUPT_MASK_LEVEL(newLevel) effect only when newLevel be urgent than current level ,and portLOWER_INTERRUPT_MASK_LEVEL(newLevel) effect only when currentLevel be more urgent then newLevel.

★ Like

Reply ↓



Erich Styger
on **October 15, 2017 at 19:15** said:

I agree that the current way is not the most flexible way in FreeRTOS. On the other side I think changing dynamically the BASEPRI needs real good analyis and design of the system to make it work correctly.

★ Like

Reply ↓

Andy Blue on **October 15, 2017 at 19:20** said:

I'm be more interested to improve the FreeRTOS implementation on Cortex-M0.
One is to save MainStack as the similar way on M3/M4/M7. Usually on M0/M0+, the RAM space is more limited than on M3/M4/M7, so the effort to save RAM would be more valuable.
I think one easy way to do is add a config Option of configM0plusWithVTOR, when configured to 1, allow M0+ do the same procedure to reset MSP as M3/M4/M7 does when startFirstTask.
Another way is to add a config Option to specify the location of Vector Table, or even the absolute const value for MSP to reset when startFirstTask.

★ Like

Reply ↓

Andy Blue on **October 15, 2017 at 22:34** said:

The another improvement on RTOS with Cortex-M0 is the way of implementation of critical section.
As you explained, Enter_Critical on Cortex-M0 actually disabled all interrupts, this certainly increased the latency of interrupt response, and give no chance to get rid of it, for event an ISR doesn't invoke Kernel_API_FromISR. This could be a big impact to an engineering work in the real world.
I have some quick thinking to tackle the issue, not sure would be a right approach.
First, the needs of Enter_Critical/Exit_Critical can be divided from ISR-Handler-Context and Thread-Execution-Context. Let's first to consider in ISR-Handler-Context. All the purpose of Enter_Critical is to mask/block certain interrupt urgency level, to avoid the current execution sequence to be preempted by interrupt with higher urgency level. Instead of manipulate BASEPRI (M0 has no BASEPRI register), it is possible to temporary change SHPRx or NVIC_IPRx which mapping to current executing ISR to raise the current ISR urgency level to protect itself?
Let me give an example, assume a Cortex-M0 have configPRIO_BITS 2 . So interrupt priority number could be 0, 1, 2, and 3. PenSV and SysTick will have interrupt priority number 3 assigned. Within ISR code for PenSV and SysTick, when need to Enter_Critical, changing PRI_15 or PRI_14 to 0x40, aka raise the running ISR priority to number 1. I hope by doing this, the current running ISR will be able to prevent itself to be preempted by an interrupt with priority 1 or higher number, but still be possible to preempted by interrupt with priority number 0.
Would this be a viable approaching?
Of course, even this is viable, it only solve the problem in ISR-Handler-Context, we still need to find solution for Thread-Execution-Context.
By looking current FreeRTOS implementation, there is no easy way. But theoretically, it is possible to wrap the critical section in Kernel_API function to through a "svc xx", once in SVC handler, it comes to be in ISR-Handler-Context again, then all tricks I suggested in ISR-Handler-

Context could come to play.
Do I have big mistake in my proposal? Thanks for your time to read.

★ Like

Reply ↓

Erich Styger
on **October 20, 2017 at 12:22** said:

I think you would have to use the special barrier instructions with your approach too. But my recommendation is not to change the interrupt priorities in such a dynamic way. And in that place where you are changing the priorities, you have to be very, very sure that you are not getting interrupted by another interrupt. To me, changing the RTOS for an M0 is not worth it that way, better move on to an M3 or M4 which has better capabilities?

★ Like

Reply ↓

Andy Blue
on **October 22, 2017 at 04:14** said:

yep, I might have to use barrier instructions when make the interrupt priority changes. The big concern is that whether change the PRI_xx while the corresponding interrupt is already in processing would effect immediately? Of course the best way to answer is to write an experimental program and test it. I will try it once I get time.
Personally if I chose, I will prefer to use M3/M4 instead of the low end M0, as it is be much better on performance and only cost few dollars more; but for industry/manufacture, the M0 micro-controller could cost down to 50 cents, that really means a lot to mass production; I wish I can do something to get more juicy from the cheap M0 🙂

★ Like

Reply ↓

Erich Styger
on **October 24, 2017 at 20:42** said:

We pretty much get what we pay for, right? Plus the M3/M4 has a hardware divide instruction too, making it a much better choice in many applications too.
Anyway, I have not had much problems with M0 running FreeRTOS and the interrupt system. The M0 is used more in low power and lower performance systems, and in these typically in such applications it is not a problem to deal with some interrupt latency time. And the RTOS really does its best to keep the latency low. What really helps is turning on compiler optimizations, that makes a big difference too.

★ Like

Andy Blue
on **October 22, 2017 at 04:23** said:

There is another topic which I'm be very interested. The optional MPU on M0+/M3/M4 uses in FreeRTOS environment. Could you write an article to cover it?

★ Like

Reply ↓

Erich Styger
on **October 24, 2017 at 20:38** said:

Hi Andy,

yes, I have started working on an article on this, featuring the NXP FRDM-K64F with an M4F which supports the MPU. But I have not had the time to finish it, but it is on my (growing) list of backlog items.
Thanks for the reminder and the interest, so it really looks like I should move that article up to the priority list?
Erich

★ Like

Andy Blue
on **October 27, 2017 at 21:02** said:

Just want to give a brief/quick feedback.
I tried in the running ISR context to change the ISR priority in SHPRx or NVIC_IPRx; It effect immediately to itself the running ISR; Which means this method could be used for implementation of OS_ISR_Critical section. The immediate benefit is that I have Interrupt with urgency higher than configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY will NOT be blocked by OS_ISR_Critical on Cortext-M0 port.
There is still a lot to do to complete the changes. Need to move most part of OS_API function through SVC to run in handler context in privilege level, but test result shows this is a valid approach.
And thank to Erich for reminder to use barrier instruction. It needed indeed. 🙂

★ Like

Reply ↓

Andy Blue on **October 26, 2017 at 15:17** said:

Yes, I like to read it soon.
While I had spent time to study the MPU support in FreeRTOS, I saw the way they using MPU in FreeRTOS likely a very late added patch, inelegant. And because of using MPU, they have to support unprivilege level thread (user mode thread than privilege mode thread, by FreeRTOS's term) . Supporting unprivilege level thread is not necessary to depend on supporting MPU, but seems that they were been lazy and do things only when no way to around. 🙂
And again, I like to read it soon. It does not have to be perfect at the first publish, readers may help by feedback and reveal what is more interested part to peoples. 🙂

★ Like

Reply ↓

Scott Miller on **October 9, 2018 at 20:21** said:

Hi Erich,

I stumbled across the version of this post on DZone and it seems to be incorrect – that one says that interrupts must be numerically equal or *lower* than the max syscall priority. I don't know if you're responsible for that version or if they just scraped it from here, but in any case I'm assuming that was from an earlier revision.

I'm still tracking down my usage fault problem and I've determined that (at least in the instance I finally caught in the instruction trace buffer) my FTM ISR with priority 5 interrupted vTaskSwitchContext just as it was taking pxCurrentTCB from a list. The ISR sent data to a queue and set a direct task notification using the ISR-safe calls, and I can't see any sign of a stack overflow or memory corruption, but when it returns to vTaskSwitchContext, pxCurrentTCB is null.

Any idea why this is happening?

★ Like

Reply ↓

Erich Styger

on **October 9, 2018 at 20:50** said:

Hi Scott,
yes, DZone copies stuff from here (with my permission), and they do not update their copy. And yes, I had some mistakes in that original/first version, so this article is the latest and greatest.
About your problem: what is your BASEPRI setting/value? If your FTM ISR is using RTOS functions (what it does), then the FTM interrupt priority must numerically higher than BASEPRI. If your BASEPRI is 5, then your FTM ISR shall be 5, 6, 7 or higher. Is this the case? I assume you are not on a Cortex-M0+?
Erich

★ Like

Reply ↓

Scott Miller
on **October 9, 2018 at 21:41** said:

configMAX_SYSCALL_INTERRUPT_PRIORITY is 5, configLIBRARY_LOWEST_INTERRUPT_PRIORITY is 15, and the FTM ISR is 5. It's on a Cortex M4 (K22F).

As far as I can tell, the vTaskSwitchContext code that's being interrupted is not in a critical section, so my assumption would be that it's accessing the task list in a safe way, but I haven't thoroughly dissected it yet. The trace goes like this:

vQueueWaitForMessageRestricted
prvProcessTimerOrBlockTask
xTaskResumeAll
vPortEnterCritical
xTaskResumeAll
vPortExitCritical
xTaskResumeAll
prvProcessTimerOrBlockTask
vPortYieldFromISR

vPortPendSVHandler
vTaskSwitchContext
xTaskGetTickCountFromISR
vPortValidateInterruptPriority
xTaskGetTickCountFromISR
vTaskSwitchContext
ftm0_irq
vTaskSwitchContext
(crashes because pxCurrentTCB is now null)

I'm skipping a bunch of stuff in ftm0_irq, but it does call xTaskGenericNotifyFromISR, which calls uxListRemove and vListInsertEnd.

I'm trying to increase the rates of both the timer task and the FTM IRQ to increase the frequency of the problem, which really does look like a race condition to me. Right now I usually have to wait anywhere from 2-20 hours to see if it's going to crash.

★ Like

Reply ↓

Scott Miller on **October 10, 2018 at 00:23** said:

I've narrowed it down and simplified my setup. It's definitely acting like a race condition. To highlight the problem, I added a for loop to listGET_OWNER_OF_NEXT_ENTRY, just before the final line that sets (pxTCB), that counts down from 2000. That's enough to make the fault happen in seconds rather than hours. It happens when the ISR uses xTaskNotifyFromISR. SysTick, PendSV, and FTM0 priorities are all correct. I'm going to check the latest FreeRTOS code and see if there have been any updates since 10.0.1 that might be related.

★ Like

Reply ↓

Scott Miller on October 10, 2018 at 04:57 said:

Erich, I'm in touch with Richard Barry at FreeRTOS trying to sort this out. My debugger keeps hanging while trying to step through the PendSV handler, which isn't helping things. Can you tell me where this port.c came from? It says v10.0.1 in the header, and 'FreeRTOS for 56800EX port by Richy Ye in Jan. 2013' below that.

★ Like

Reply ↓

Erich Styger
on October 10, 2018 at 07:23 said:

Hi Scott,
I commented on that FreeRTOS thread (https://sourceforge.net/p/freertos/discussion/382005/thread/0fd89fda36). That port file is a combination of many ports, based on the origional FreeRTOS ports. I hope I have not missed anything, as I have not seen the same problem on my side yet. I try to have a look today, but if you could send me an example to try, that would be great. Otherwise I try to replicate things with the information you have provided).

Thanks,
Erich

★ Like

Reply ↓

Scott Miller
on October 10, 2018 at 19:51 said:

It's a reasonably large project that won't run on any of the standard development boards without modification, but I can try to pack it up if you want, or let me know what pieces you need to take a look at.

★ Like

Reply ↓

Erich Styger
on October 10, 2018 at 19:53 said:

I'm trying to replicate this on my side. What is the frequency of your timer interrupt?

★ Like

Scott Miller
on October 10, 2018 at 20:51 said:

Should be around 57 Hz I think. It's the overflow interrupt on a 16-bit timer with a modulo of 65535, prescaler of 16, bus clock 60 MHz. It doesn't generate a task notification every time, just when it receives a full line of GPS data, so a few times per second. For testing purposes I did have it set to generate a notification every time and that was enough to show the problem. The SysTick interrupt is 300 Hz.

★ Like

Pingback: FreeRTOS: how to End and Restart the Scheduler | MCU on Eclipse

Pingback: Tutorial: How to Optimize Code and RAM Size | MCU on Eclipse

Manuj Agrawal on May 27, 2020 at 15:21 said:

Dear Erich,

Thanks for this informative article.

Since, all interrupts with numerical value of priority greater than and equal to configMAX_SYSCALL_INTERRUPT_PRIORITY are blocked/masked by the critical section, what really happens to such blocked interrupt after FreeRTOS exits the critical section? Will this interrupt still enter the ISR function after the the critical section has been exited? -> Is this interrupt completely missed or just delayed?

If this interrupt is completely missed, flag reset operation of that particular interrupt generator peripheral may not be performed and the application may start exhibiting unpredictable behavior.

Please help be build clarity on this issue.

Regards
Manuj

★ Like

Reply ↓

Erich Styger
on **May 27, 2020 at 20:02** said:

Hi Manuj,
What happens if the interrupt is masked (regardless of BASEPRI or PRIMASK): it gets registered (pending bit set).
Should the interrupt happen multiple times while it is masked, it only gets registered once (because only one bit).
Once interrupts are unmasked, the interrupt controller selects the one with the highest priority to be executed first.

I hope this helps,
Erich

★ Like

Reply ↓

Justin on **June 30, 2020 at 17:54** said:

Hi Erich,

How would you go about nested enter/exit critical sections that do not block all interrupts, but instead use the BASEPRI to decide what interrupt priorities to block. I mean something outside of FreeRTOS, on the developer side on an M4 or M7. Additionally, if the application was also using FreeRTOS at the same time, do we need to be worry of something as FreeRTOS also changes the BASEPRI mask?

It looks like FreeRTOS sets the BASEPRI mask to configMAX_SYSCALL_INTERRUPT_PRIORITY on enter every time, and sets it back to 0 on exit. What if we wanted to allow enter to change the BASEPRI mask to a wider range of priorities. Would we allow deeper calls in the nest to change the BASEPRI mask as well or only the first call? And what should happen on exit?

Thank you!

★ Like

Reply ↓

Erich Styger
on **June 30, 2020 at 20:36** said:

You can use the way FreeRTOS does the nesting with the BASEPRI for a non-FreeRTOS application too. And you don't need to worry about setting the BASEPRI as it is done with FreeRTOS, because it is implemented in a reentrant way. I would not consider to change the BASEPRI level at runtime as you suggest: that can be very tricky in some cases, and makes a system more complex than it needs to be.
You would need to push the current value of the BASEPRI on the stack and restore it properly. In addition to checking the active value. That introduces extra interrupt latency too.

★ Like

Reply ↓

Andy Blue
on **July 12, 2020 at 21:22** said:

To this problem I'm be with Liviu on his comments of https://mcuoneclipse.com/2016/08/28/arm-cortex-m-interrupts-and-freertos-part-3/#comment-85261.
The problem with the implementation of FreeRTOS is that it uses internal counter to handle reentry of enter and exit critical section, this can be a trouble to an ISR want to have their own intention to adjust BASEPRI level in a period at runtime.
Let's look an example. Assume on a Contex-M4 with 4 priority bits, and all 16 priority level to be configured as preempt priority and none for subpriority, configMAX_SYSCALL_INTERRUPT_PRIORITY is configured to 8. If an X_ISR with its default priority pre-configured to 12, and in the middle of ISR processing, want to have a period to block interrupts with priority number equal or higher than 4. With FreeRTOS implementation to critical section, it is very difficult to get things right. Refer to my previous comment on https://mcuoneclipse.com/2016/08/28/arm-cortex-m-interrupts-and-freertos-part-3/#comment-99344
In my own RTOS implementation, I invented and introduced a concept of SystemCriticalLevel. For the Contex-M4 with 4 priority bits, the valid SystemCriticalLevel is ranging from 0 to 15, and the dynamic changing SystemCriticalLevel will be suggest to a paradigm as this:
// save current SystemCriticalLevel to stack.
// rise SystemCriticalLevel to desired level if current is lower.
// do the work needed
// restore SystemCriticalLevel to the original level which have been saved at beginning.

★ Like

Reply ↓

Liviu Ionescu (ilg)
on **July 13, 2020 at 12:20** said:

Hi Andy,

In a previous version of µOS++, I also experimented with assembly code for saving the interrupt status on stack. Unfortunately GCC proved unsafe when the applications handle SP on their own, and I wouldn't be surprised that other compilers have bugs related to this too.

For you to understand the issue, compilers reserve stack space for local variables generally by decreasing SP and saving it in a separate register that is not touched during the function. For unknown reasons, presumably bugs, sometimes the compiler refers to some local variables directly via SP, not the separate register. This is ok as long as you do not touch SP in assembly code. Once you do it, you obviously end up referring to the wrong local variables, and the behaviour can be very hard to debug.

Thus my conclusion that the only portable and safe way to implement critical sections is with an explicit variable where to save the interrupt status (be it the simple enable/disable flag, or the priority threshold, or whatever).

In C++ the Resource acquisition is initialization (RAII) paradigm provides an even more convenient solution, like

```
void
func(void)
{
// Do something
{
interrupts::critical_section ics; // Critical section
```

begins here.
// Inside the critical section.
} // Critical section ends here.
// Do something else.
}

★ Like

Andy Blue
on **July 14, 2020 at 18:11** said:

Hi Liviu,
Thanks for your comments and sharing.
Personally I would avoid to touch SP in assembly. The only situation I had done that is in RTOS starting part.
What I had mentioned to save current SystemCriticalLevel to stack is by preserve it to an auto (local) variable within a C function. It could be like

void aFunction()
{
SystemCriticalLevel_t originalSystemCriticalLevel;

originalSystemCriticalLevel = riseSystemCriticalLevel(DESIRED_LEVEL);

// from this point the System Critical Level will be guaranteed as newlevel or higher.
// do things

dropSystemCriticalLevel(originalSystemCriticalLevel);
// from this point the System Critical Level restored to originalSystemCriticalLevel.
}

Behind riseSystemCriticalLevel() and

dropSystemCriticalLevel() it update global interrupt enable/diable or BASEPRI as the logic needed.
There is an alternative API can perform the same procedure while revealing little more internal logic.

```
void bFunction()
{
SystemCriticalLevel_t
originalSystemCriticalLevel;

originalSystemCriticalLevel =
getSystemCriticalLevel();

if (DESIRED_LEVEL higher_than
originalSystemCriticalLevel)
{
setSystemCriticalLevel(DESIRED_LEVEL);
}

// from this point the System Critical Level will
be guaranteed as newlevel or higher.

// do things

if (DESIRED_LEVEL higher_than
originalSystemCriticalLevel)
{
setSystemCriticalLevel(originalSystemCriticalL
evel);
}
// from this point the System Critical Level
restored to originalSystemCriticalLevel.
}
```

Where the valid range of System Critical Level depends on the CPU architecture and implementation. Global Interrupt disabled is considered to be highest System Critical Level; global interrupt enabled with 0 of BASEPRI is considered to be lowest (or none) System

Critical Level; global interrupt enabled with non-zero BASEPRI is considered to be middle System Critical Levels in between.

Then, Critical Section is just one of System_Critical_Level select and configured, using consistently through the project. Application code can leverage on the implementation of System Critical Level to easily get what they want.

★ Like

What do you think?