ScienceDirect

Systick Interrupt

Related terms:

Control Register, Systick Timer

Low Power and System Control Features

Joseph Yiu, in The Definitive Guide to ARM® CORTEX®-M3 and CORTEX®-M4 Processors (Third Edition), 2014

9.5.3 Using the SysTick timer

If you only want to generate a periodic SysTick interrupt, the easiest way is to use a CMSIS-Core function called “SysTick_Config”:

```
uint32_t SysTick_Config(uint32_t ticks);
```

This function sets the SysTick interrupt interval to “ticks,” enables the counter using the processor clock, and enables the SysTick exception with the lowest exception priority.

For example, if you have a clock frequency of 30MHz and you want to trigger a SysTick exception of 1KHz, you can use:

```
SysTick_Config(SystemCoreClock / 1000);
```

The variable “SystemCoreClock” should hold the correct clock frequency value of 30x10.^{6} Alternatively, you can just use:

```
SysTick_Config(30000); // 30MHz / 1000 = 30000
```

The “SysTick_Handler(void)” will then be triggered at a rate of 1 kHz.

If the input parameter of SysTick_Config function cannot be fit into the 24-bit reload value register (larger than 0xFFFFFF), the SysTick_Config function returns 1; otherwise, it returns 0.

In many cases you might not want to use the SysTick_Config function because you might want to use the reference clock or you might not want to enable the SysTick interrupt. In these cases you need to program the SysTick registers directly, and the following sequence is recommended:

1. Disable the SysTick timer by writing 0 to SysTick->CTRL. This step is optional. It is recommended for reusable code because the SysTick could have been enabled previously.
2. Write the new reload value to SysTick->LOAD. The reload value should be the interval value –1.
3. Write to the SysTick Current Value register SysTick->VAL with any value to clear the current value to 0.
4. Write to the SysTick Control and Status register SysTick->CTRL to start the SysTick timer.

Since the SysTick timer counts down to 0, if you want to set the SysTick interval to 1000, you should set the reload value (SysTick->LOAD) to 999.

If you want to use the SysTick timer in polling mode, you can use the count flag in the SysTick Control and Status Register (SysTick->CTRL) to determine when the timer reaches zero. For example, you can create a timed delay by setting the SysTick timer to a certain value and waiting until it reaches zero:

```
 SysTick->CTRL = 0;    // Disable SysTick
 SysTick->LOAD = 0xFF; // Count from 255 to 0 (256 cycles)
 SysTick->VAL = 0;     // Clear current value as well as
count flag
 SysTick->CTRL = 5;    // Enable SysTick timer with processor
clock
 while ((SysTick->CTRL & 0x00010000)==0);// Wait until count
flag is set
 SysTick->CTRL = 0; // Disable SysTick
```

If you want to schedule the SysTick interrupt for one-shot operation, which triggers in a certain time, you can reduce the reload value by 12 cycles to compensate for the interrupt latency. For example, if we want to have the SysTick handler to execute in 300 clock cycle time:

```
volatile int SysTickFired; // A global software flag to
                           // indicate SysTickAlarm executed
…
SysTick->CTRL = 0;         // Disable SysTick
SysTick->LOAD = (300-12);  // Set Reload value
                           // Minus 12 because of exception
latency
SysTick->VAL = 0;          // Clear current value to 0
SysTickFired = 0;          // Setup software flag to zero
SysTick->CTRL = 0x7;       // Enable SysTick, enable SysTick
                           // exception and use processor
clock
while (SysTickFired == 0); // Wait until software flag is set
by
                           // SYSTICK handler
```

Inside the one-shot SysTick Handler, we need to disable the SysTick so that the SysTick exception only triggers once. We might also need to clear the SysTick pending status in case the pending status has been set again when the required processing task takes some time:

```
void SysTick_Handler(void) // SYSTICK exception handler
{
SysTick->CTRL = 0x0;       // Disable SysTick
…;                        // Execute required processing task
SCB->ICSR |= 1<<25;        // Clear SYSTICK pend bit
                           // in case it has been pended
again
SysTickFired++;            // Update software flag so that
the
                           // main program know that SysTick
alarm
                           // task has been carried out
return;
}
```

If there is another exception happening at the same time, the SysTick exception could be delayed.

The SysTick timer can be used for timing measurement. For example, you can measure the duration of a short function using the following code:

```
unsigned int start_time, stop_time, cycle_count;
SysTick->CTRL = 0;          // Disable SysTick
SysTick->LOAD = 0xFFFFFFFF; // Set Reload value to maximum
SysTick->VAL = 0;           // Clear current value to 0
SysTick->CTRL = 0x5;        // Enable SysTick, use processor
clock
while(SysTick->VAL != 0);   // Wait until SysTick reloaded
start_time = SysTick->VAL;  // Get start time
function();                 // Execute function to be
measured
stop_time = SysTick->VAL;   // Get stop time
cycle_count = start_time – stop_time;
```

Since the SysTick is a decrement counter, the value of start_time is larger than stop_time. You might want to include a check for the count_flag at the end of the timing measurement. If the count_flag is set, the duration being measured is longer than 0xFFFFFF clock cycles. In that case you will have to enable the SysTick exception and use the SysTick Handler to count how many times the SysTick counter underflows. The total number of clock cycles will then also include the SysTick exceptions.

The SysTick timer provides a register to provide a calibration value. If this information is available, the lowest 24 bits of the SysTick->CALIB register provide the reload value required to get 10 msec SysTick intervals. However, many microcontrollers do not have this information and the TENMS bit field would read as zero. The CMSIS-Core approach of providing a software variable (SystemCoreClock) for clock frequency information is more flexible and supported by most microcontroller vendors.

You can use the bit 31 of the SysTick Calibration register to determine if a reference clock is available.

Read full chapter

URL: https://www.sciencedirect.com/science/article/pii/B9780124080829000099

Exceptions and Interrupts

Joseph Yiu, in The Definitive Guide to the ARM Cortex-M0, 2011

Exception Sequence Overview

Acceptance of Exception Request

The processor accepts an exception if the following conditions are satisfied:

- For interrupt and SysTick interrupt requests, the interrupt has to be enabled
- The processor is not running an exception handler of the same or a higher priority
- The exception is not blocked by the PRIMASK interrupt masking register

Note that for SVC exception, if the SVC instruction is accidentally used in an exception handler that has the same or a higher priority than the SVC exception itself, it will cause the hard fault exception handler to execute.

Stacking and Unstacking

To allow an interrupted program to be resumed correctly, some parts of the current state of the processor must be saved before the program execution switches to the exception handler that services the occurred exception. Different processor architectures have different ways to do this. In the Cortex-M0 processor, the architecture uses a mixture of automatic hardware arrangement and, only if necessary, additional software steps for saving and restoring processor status.

When an exception is accepted on the Cortex-M0 processor, some of the registers in the register banks (R0 to R3, R12, R14), the return address (PC), and the Program Status Register (xPSR) are pushed to the current active stack memory automatically. The Link Register (LR/R14) is then updated to a special value to be used during exception return (EXC_RETURN, to be introduced later in this chapter), and then the exception vector is automatically located and the exception handler starts to execute.

At the end of the exception handling process, the exception handler executes a return using the special value (EXC_RETURN, previously generated in LR) to trigger the exception return mechanism. The processor checks to determine if there is any other exception to be serviced. If not, the register values previously stored on the stack memory are restored and the interrupted program is resumed.

The actions of automatically saving and restoring of the register contents are called "stacking" and "unstacking" (Figure 8.4). These mechanisms allow exception handlers to be implemented as normal C functions, thereby reducing the software overhead of exception handling as well as the circuit size (no need to have extra banked registers), and hence lowering the power consumption of the design.

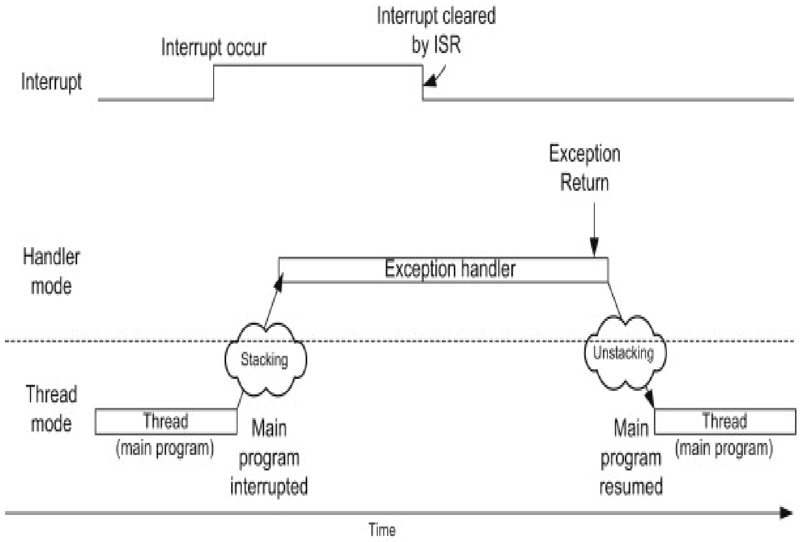


Figure 8.4. Stacking and unstacking of registers at exception entry and exit.

The registers not saved by the automatic stacking process will have to be saved and restored by software in the exception handler, if the exception handler had modified them. However, this does not affect the use of normal C functions as exception handlers, because it is a requirement for C compilers to save and restore the other registers (R4-R11) if they will be modified during the C function execution.

Exception Return Instruction

Unlike some other processors, there is no special return instruction for exception handlers. Instead, a normal return instruction is used and the value load into PC is used to trigger the exception return. This allows exception handlers to be implemented as a normal C function.

Two different instructions can be used for exception return:

> BX <Reg>; Load a register value into PC (e.g., "BX LR")

and

> POP {<Reg1>,< Reg2>,….,PC}; POP instruction with PC being one of the registers being updated

When one of these instructions is executed with a special value called EXC_RETURN being loaded into the program counter (PC), the exception return mechanism will be triggered. If the value being loaded into the PC does not match the EXC_RETURN pattern, then it will be executed as a normal BX or POP instruction.

Tail Chaining

If an exception is in a pending state when another exception handler has been completed, instead of returning to the interrupted program and then entering the exception sequence again, a tail-chain scenario will occur. When tail chain occurs, the processor will not have to restore all register values from the stack and push them back to the stack again. The tail chaining of exceptions allows lower exception processing overhead and hence better energy efficiency (Figure 8.5).

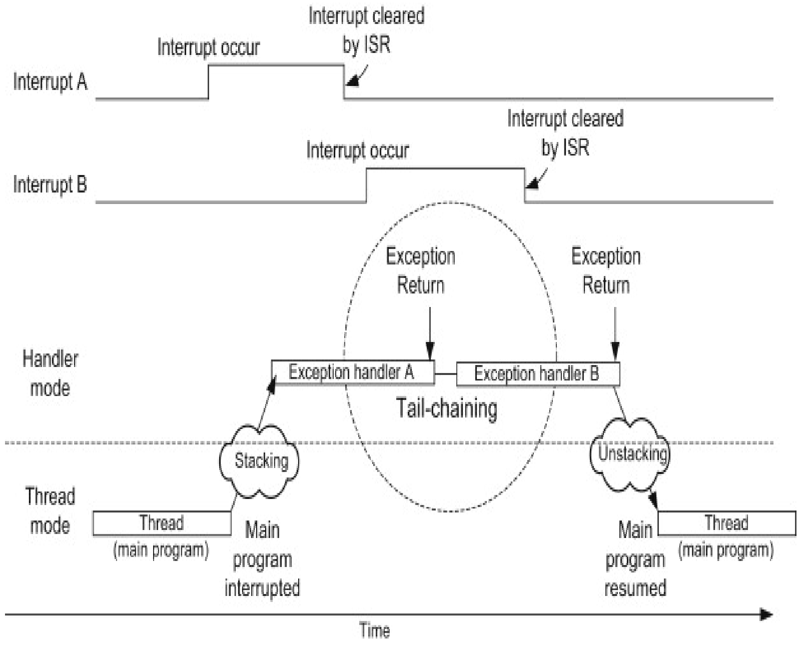


Figure 8.5. Tail chaining of an interrupt service routine.

Late Arrival

Late arrival is an optimization mechanism in the Cortex-M0 to speed up the processing of higher-priority exceptions. If a higher priority exception occurs during the stacking process of a lower-priority exception, the processor switches to handle the higher-priority exception first (Figure 8.6).

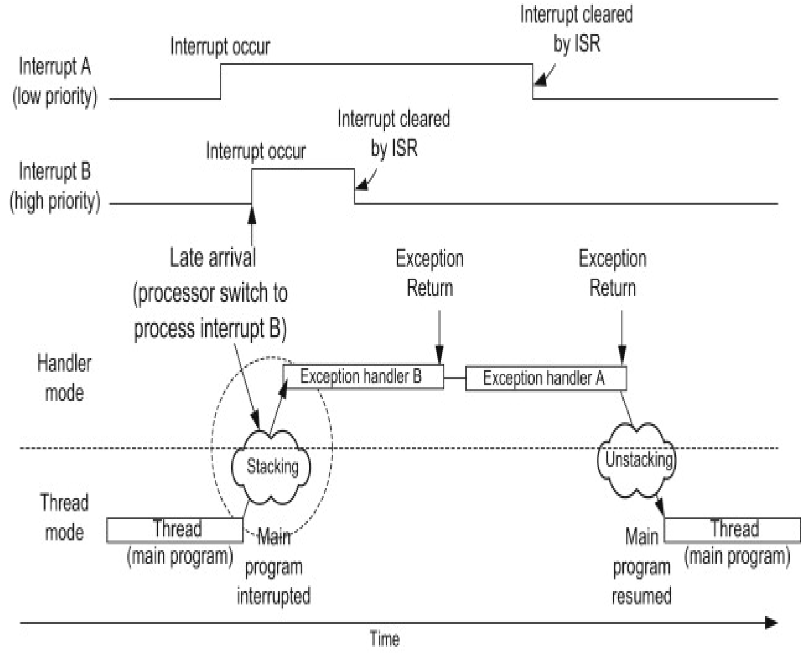


Figure 8.6. Late arrival optimization.

Because processing of either interrupt requires the same stacking operation, the stacking process continues as normal when the late-arriving, higher-priority interrupt occurs. At the end of the stacking process, the vector for the higher-priority exception is fetched instead of the lower-priority one.

Without the late arrival optimization, a processor will have to preempt and enter the exception entry sequence again at the beginning of the lower-priority exception handler. This results in longer latency as well as larger stack memory usage.

Read full chapter

URL: https://www.sciencedirect.com/science/article/pii/B9780123854773100084

RTOS Techniques

Trevor Martin, in The Designer's Guide to the Cortex-M Processor Family (Second Edition), 2016

Exercise 10.1 RTOS Interrupt Exercise Handling

CMSIS-RTOS does not introduce any latency in serving interrupts generated by user peripherals. However, operation of the RTOS may be disturbed if you lock out the SysTick interrupt for a long period of time. This exercise demonstrates a technique of signaling, a thread from an interrupt and servicing the peripheral interrupt with a thread rather than a standard ISR.

Open the Pack Installer.

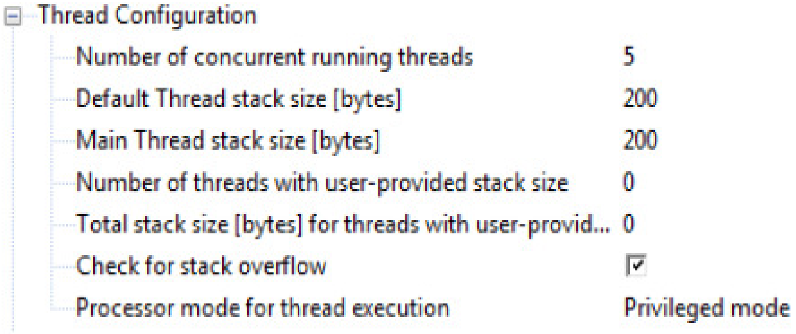
Select the Boards::Designers Guide Tutorial.

Select the Example tab and Copy "EX 10.1 RTOS Interrupt Handling."

In the main function, we initialize the ADC and create an ADC thread which has a higher priority than all the other threads.

```
osThreadDef(adc_Thread, osPriorityAboveNormal, 1, 0);

int main (void) {

LED_Init ();

init_ADC ();

T_led_ID1 = osThreadCreate(osThread(led_Thread1), NULL);

T_led_ID2 = osThreadCreate(osThread(led_Thread2), NULL);

T_adc_ID = osThreadCreate(osThread(adc_Thread), NULL);
```

However, there is a problem when we enter main(): the code is running in unprivileged mode, so we cannot access the NVIC registers without causing a fault exception. There are several ways round this; the simplest is to give the threads privileged access by changing the setting in the RTX_Conf_CM.c (Fig. 10.4).



⊟ Thread Configuration

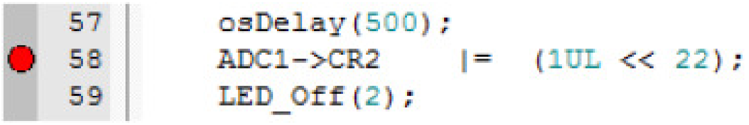
Number of concurrent running threads	5
Default Thread stack size [bytes]	200
Main Thread stack size [bytes]	200
Number of threads with user-provided stack size	0
Total stack size [bytes] for threads with user-provid...	0
Check for stack overflow	☑
Processor mode for thread execution	Privileged mode

Figure 10.4. Configure the RTOS threads to have privileged access to the Cortex-M processor registers.

Here, we have switched the "Processor Mode for Thread Execution" to privileged which gives the threads full access to the Cortex-M processor. As we have added a thread, we also need to increase the number of concurrent running threads.

Build the code and start the debugger.

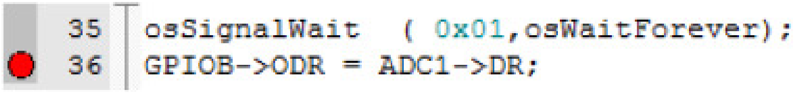
Set breakpoints in led_Thread2, ADC_Thread, and ADC1_2_IRQHandler (Fig. 10.5).



```
    osDelay(500);
    ADC1->CR2     |=  (1UL << 22);
    LED_Off(2);
```

Figure 10.5. Breakpoint on "start of ADC conversion".

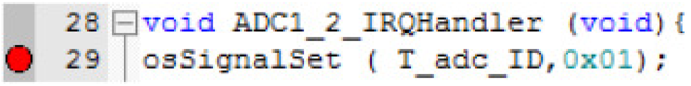
And in adc_Thread() (Fig. 10.6)



```
osSignalWait  ( 0x01,osWaitForever);
GPIOB->ODR = ADC1->DR;
```

Figure 10.6. Breakpoint in the ADC Thread.

And in ADC1_2_IRQ Handler (Fig. 10.7)



```
void ADC1_2_IRQHandler (void){
osSignalSet ( T_adc_ID,0x01);
```

Figure 10.7. Breakpoint in the ADC interrupt.

Run the code.

You will hit the first breakpoint which starts the ADC conversion, then run the code again and you will enter the ADC interrupt handler. The handler sets the adc_thread signal and quits. Setting the signal will cause the ADC thread to preempt any other running task, run the ADC service code, and then block waiting for the next signal.

Read full chapter

URL: https://www.sciencedirect.com/science/article/pii/B9780081006290000104

Cortex-M Architecture

Trevor Martin, in The Designer's Guide to the Cortex-M Processor Family (Second Edition), 2016

Exercise 3.3 Working with Multiple Interrupts

This exercise extends our original SysTick exception exercise to enable a second ADC interrupt. We can use these two interrupts to examine the behavior of the NVIC when it has multiple interrupt sources.

Open the Pack Installer.

Select the Boards tab and "The Designers Guide Tutorial Examples."

Select the Example tab and Copy "Ex 3.3 Multiple Interrupts."

Open main.c and locate the main() function.

```
unsigned char BACKGROUND =0;unsigned char ADC =0;unsigned
char SYSTICK=0;

int main (void){

int i;

GPIOB->CRH = 0x33333333;    //Configure the Port B LED pins

SysTick->VAL   = 0x9000;      //Start value for the sys Tick
counter

SysTick->LOAD  = 0x9000;       //Reload value

SysTick->CTRL  = SYSTICK_INTERRUPT_ENABLE |
SYSTICK_COUNT_ENABLE;

init_ADC();                   //setup the ADC peripheral

ADC1->CR1    |= (1UL <<  5);      // enable for EOC Interrupt

NVIC->ISER[0] = (1UL << 18);  // enable ADC Interrupt

ADC1->CR2    |= (1UL <<  0);   // ADC enable

while(1){

 BACKGROUND = 1;

}}
```

We initialize the SysTick timer same as before. In addition, the ADC peripheral is also configured. To enable a peripheral interrupt, it is necessary to enable the interrupt source in the peripheral and also enable its interrupt channel in the NVIC by setting the correct bit in the NVIC ISER registers. In this example, we are writing directly to the NVIC->ISER[0] register. In the next chapter, we will see a better more “standard” way to do this.

We have also added three variables: BACKGROUND, ADC, and SYSTICK. These will be set to logic one when the matching region of code is executing and zero at other times. This allows us to track execution of each region of code using the debugger Logic Analyzer.

```
void ADC_IRQHandler (void){
int i;
BACKGROUND  = 0;
SYSTICK   = 0;
for (i=0;i<0x1000;i++){
 ADC =1;
}
ADC1->SR &= ~(1 << 1);         /* clear EOC interrupt      */
ADC =0;
}
```

The ADC interrupt handler sets the execution region variables then sits in a delay loop. Before exiting it also writes to the ADC status register to clear the end of conversion flag. This deasserts the ADC interrupt request to the NVIC.

```
 void SysTick_Handler (void){
int i;
BACKGROUND = 0;
ADC = 0;
ADC1->CR2    |=  (1UL << 22);
for (i=0;i<0x1000;i++){
 SYSTICK = 1;
}
SYSTICK =0;
}
```

The SysTick interrupt handler is similar to the ADC handler. It sets the region execution variables and sits in a delay loop before exiting. It also writes to the ADC control register to trigger a single ADC conversion.

Build the project and start the simulator.

Add each of the execution variables to the Logic Analyzer and start the code running (Fig. 3.40).

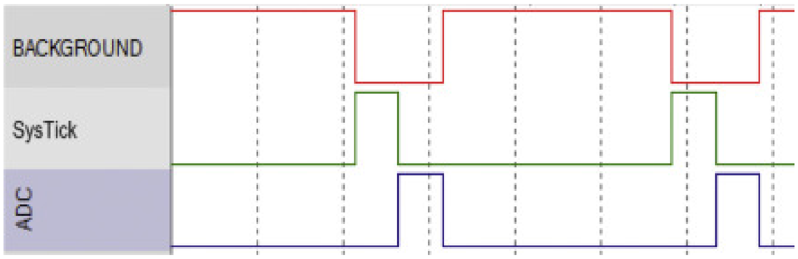


Figure 3.40. The SysTick interrupt is executed (logic high) then the ADC interrupt is tail chained and will run when the SysTick ends.

The SysTick interrupt is raised which starts the ADC conversion. The ADC finishes conversion and raises its interrupt before the SysTick interrupt completes so it enters a Pending state. When the SysTick interrupt completes, the ADC interrupt is tail chained and begins execution without returning to the background code.

Exit the debugger and comment out the line of code that clears the ADC end of conversion flag.

```
//ADC1->SR &= ~(1 << 1);
```

Build the code and restart the debugger and observe the execution of the interrupts in the Logic Analyzer window (Fig. 3.41).

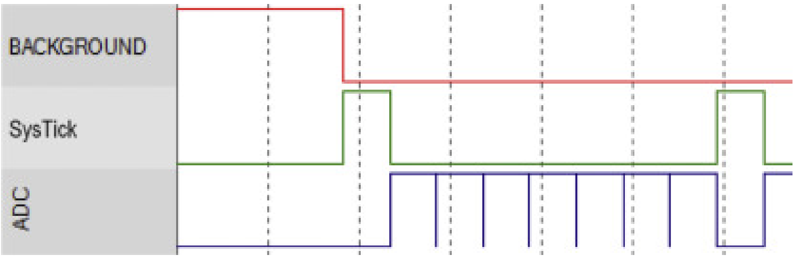


Figure 3.41. The ADC tail chains the SysTick and runs multiple times because the ADC status flag has not been cleared.

After the first ADC interrupt has been raised, the interrupt status flag has not been cleared and the ADC interrupt line to the NVIC stays asserted. This causes continuous ADC interrupts to be raised by the NVIC blocking the activity of the background code. The SysTick interrupt has the same priority as the ADC so it will be tail chained to run after the current ADC interrupt has finished. Neglecting to clear interrupt status flags is the most common mistake made when first starting to work with interrupts and the Cortex-M processors.

Exit the debugger and uncomment the end of conversion code.

```
ADC1->SR &= ~(1 << 1);
```

Add the following lines to the background initializing code.

```
NVIC->IP[18] = (2 << 4);
```

```
SCB->SHP[11] = (3 << 4);
```

This programs the user peripheral NVIC interrupt priority registers to set the ADC priority level and the “System Handler Priority” registers to set the SysTick priority level. These are both byte arrays which cover the 8-bit priority field for each exception source. However, on this microcontroller the manufacturer has implemented four priority bits out of the possible eight. The priority bits are located in the upper nibble of each byte. On reset, the PRIGROUP is set to zero which creates a 7-bit preemption field and 1-bit priority field (Fig. 3.42).

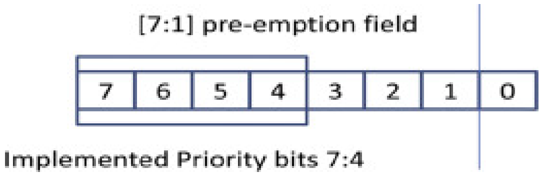


Figure 3.42. After reset a microcontroller with four implemented priority bits will have 16 levels of preemption.

On our device all of the available priority bits are located in the preemption field giving us 16 levels of priority preemption.

Build the code, restart the debugger, and observe the execution of the interrupts in the Logic Analyzer window (Fig. 3.43).

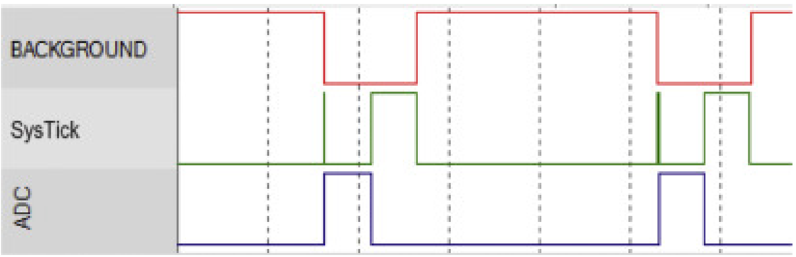


Figure 3.43. The ADC is at a higher priority than the SysTick so it pre-empts the SysTick interrupt.

The ADC now has the highest per emption value so as soon as its interrupt is raised it will preempt the SysTick interrupt. When it completes, the SysTick interrupt will resume and complete before returning to the background code.

Exit the debugger and uncomment the following lines in the background initialization code.

The AIRC register cannot be written to freely. It is protected by a key field which must be programmed with the value 0x5FA before a write is successful.

```
temp   =    SCB->AIRCR;
temp & = ~(SCB_AIRCR_VECTKEY_Msk | SCB_AIRCR_PRIGROUP_Msk);
temp   =    (temp|((uint32_t)0x5FA << 16) | (0x05 << 8));
SCB->AIRCR = temp;
```

This programs the PRIGROUP field in the AIRC register to a value of 5, which means a 2-bit preemption field and a 6-bit priority field. This maps onto the available 4-bit priority field giving four levels of preemption each with four levels of priority (Fig. 3.44).

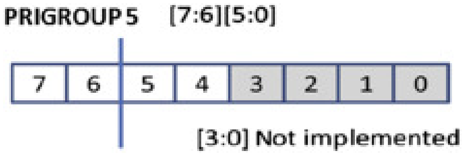


Figure 3.44. The PRI Group is set to define a 2-bit preemption field and a 2-bit priority field.

Build the code and restart the debugger and observe the execution of the interrupts in the Logic Analyzer window (Fig. 3.45).

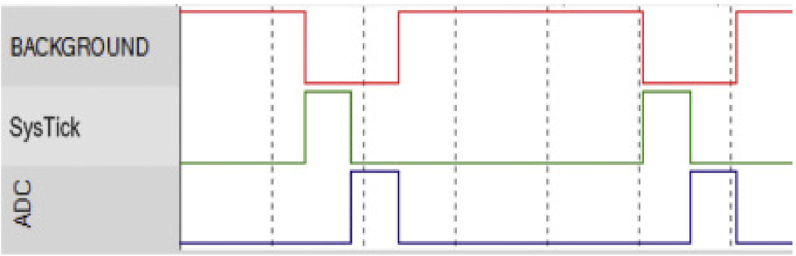


Figure 3.45. The SysTick and ADC have different priority levels but the same preempt level. Now, the ADC cannot preempt the SysTick.

The ADC interrupt is no longer preempting the SysTick timer despite them having different values in their priority registers. This is because they now have different values in the priority field but the same preempt value.

Exit the debugger and change the interrupt priorities as shown below.

```
NVIC->IP[18] = (2<<6 | 2<<4);
```

```
SCB->SHP[11] = (1<<6 | 3<<4);
```

Set the base priority register to block the ADC preempt group.

```
 __set_BASEPRI (2<<6);
```

Build the code, restart the debugger, and observe the execution of the interrupts in the Logic Analyzer window (Fig. 3.46).

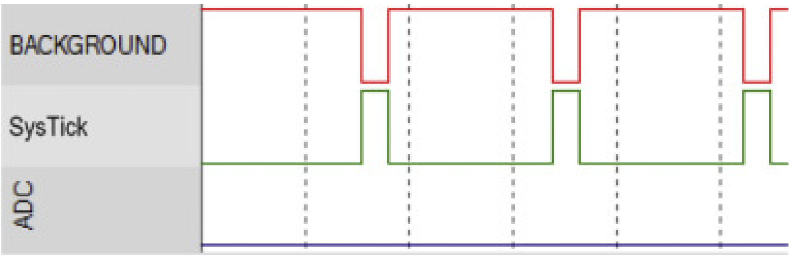


Figure 3.46. The setting the Base priority register disables the ADC interrupt.

Setting the BASEPRI register has disabled the ADC interrupt and any other interrupts that are on the same level preempt group or lower.

Read full chapter

URL: https://www.sciencedirect.com/science/article/pii/B9780081006290000037

Developing QP Applications

Miro Samek, in Practical UML Statecharts in C/C++ (Second Edition), 2009

9.3.2 “Vanilla” Kernel on Cortex-M3

The code for the DPP port to Cortex-M3 with the “vanilla” kernel is located in the directory `<qp>\qpc\examples\cortex-m3\vanilla\iar\dpp\`. The directory contains the IAR EWARM v5.11 project files to build the application and download it to the EV-LM3S811 board. Figure 9.6 shows the display of the board while it is executing the application. Listing 9.5 shows the BSP for this version of DPP.

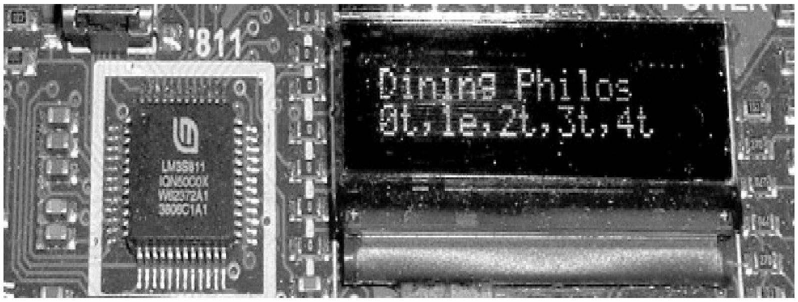


Figure 9.6. DPP test application running on the EV-LM3S811 board (Cortex-M3). The status of each Philosopher is displayed as “t” (thinking), “e” (eating), or “h” (hungry).

Listing 9.5

BSP for the DPP application with the “vanilla” kernel on bare metal Cortex-M3 (file `&lt;qp&gt;\qpc\examples\cortex-m3\vanilla\iar\dpp-ev-lm3s811\bsp.c`)

```
     #include &quot;qp_port.h&quot;
     #include &quot;dpp.h&quot;
    #include &quot;bsp.h&quot;
(1) #include &quot;hw_ints.h&quot;
     . . .     /* other Luminary Micro driver library
include files */
     /* Local-scope objects -------------------------
-------------------------*/
     static uint32_t l_delay = 0UL; /* limit for the
loop counter in busyDelay() */
     /*..............................................
.......................*/
(2) void ISR_SysTick(void) {
             QF_tick();                            /*
process all armed time events */
            /* add any application-specific clock-tick
processing, as needed */
     }
     . . .
 /*..................................................
...................*/
    void BSP_init(int argc, char *argv []) {
           (void)argc;                  /* unused:
avoid the complier warning */
           (void)argv;               /* unused: avoid
the compiler warning */
           /* Set the clocking to run at 20MHz from
the PLL. */
(3)
```

```
           SysCtlClockSet(SYSCTL_SYSDIV_10  |
     SYSCTL_USE_PLL
                                            |
     SYSCTL_OSC_MAIN | SYSCTL_XTAL_6MHZ);
                    /* Enable the peripherals used by
     the application. */
(4)        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
              SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOC);
               /* Configure the LED, push button, and UART
     GPIOs as required. */
(5)         GPIODirModeSet(GPIO_PORTA_BASE, GPIO_PIN_0 |
     GPIO_PIN_1,

     GPIO_DIR_MODE_HW);
                GPIODirModeSet(GPIO_PORTC_BASE,
     PUSH_BUTTON, GPIO_DIR_MODE_IN);
                GPIODirModeSet(GPIO_PORTC_BASE, USER_LED,
     GPIO_DIR_MODE_OUT);
                GPIOPinWrite(GPIO_PORTC_BASE, USER_LED,
     0);
               /* Initialize the OSRAM OLED display. */
(6)         OSRAMInit(1);
(7)         OSRAMStringDraw(&quot;Dining Philos&quot;, 0,
     0);
(8)         OSRAMStringDraw(&quot;0 ,1 ,2 ,3 ,4&quot;, 0,
     1);
          }
      /*.................................................
     ...................*/
          void BSP_displyPhilStat(uint8_t n, char const
     *stat) {
                 char str [2];
                 str [0] = stat [0];
                 str [1] = ‘\0’;
(9)         OSRAMStringDraw(str, (3*6*n + 6), 1);
          }
      /*.................................................
     ...................*/
          void BSP_busyDelay(void) {
                 uint32_t volatile i = l_delay;
                 while (i-- &gt; 0UL)
     {                                   /* busy-wait loop
     */
                 }
          }
      /*..................................................
     ...................*/
          void QF_onStartup(void) {
                 /* Set up and enable the SysTick timer.
     It will be used as a reference
                  * for delay loops in the interrupt
     handlers.  The SysTick timer period
                  * will be set up for BSP_TICKS_PER_SEC.
                  */
 (10)      SysTickPeriodSet(SysCtlClockGet() /
     BSP_TICKS_PER_SEC);
 (11)            SysTickEnable();
 (12)            IntPrioritySet(FAULT_SYSTICK,
     0xC0);     /* set the priority of SysTick */
 (13)            SysTickIntEnable();        /* Enable the
     SysTick interrupts */
 (14)            QF_INT_UNLOCK(dummy);              /* set
     the interrupt flag in PRIMASK */
          }
      /*..................................................
     ...................*/
          void QF_onCleanup(void) {
 (15)}
      /*..................................................
     ...................*/
          void QF_onIdle(void) {        /* entered with
     interrupts LOCKED, see NOTE01 */
                 /* toggle the User LED on and then off,
     see NOTE02 */
                 GPIOPinWrite(GPIO_PORTC_BASE, USER_LED,
     USER_LED);        /* User LED on */
                 GPIOPinWrite(GPIO_PORTC_BASE, USER_LED,
     0);                      /* User LED off */
 (16)#ifdef NDEBUG
                 /* Put the CPU and peripherals to the low-
     power mode.
                 * you might need to customize the clock
     management for your application,
                 * see the datasheet for your particular
     Cortex-M3 MCU.
                 */
 (17)
     __asm(&quot;WFI&quot;);
           /* Wait-For-Interrupt */
          #endif
   (18)     QF_INT_UNLOCK(dummy);    /* always unlock the
     interrupts */
          }
      /*..................................................
     ...................*/
          void Q_onAssert(char const Q_ROM * const
     Q_ROM_VAR file, int line) {

     (void)file;                                  /* avoid
     compiler warning */

     (void)line;                                  /* avoid
     compiler warning */
                 QF_INT_LOCK(dummy);       /* make sure
     that all interrupts are disabled */
   (19)      for (;;) {   /* NOTE: replace the loop with
     reset for the final version */
                  }
          }
          /* error routine that is called if the Luminary
     library encounters an error */
   (20)void __error__(char *pcFilename, unsigned long ulLine)
     {
                  Q_onAssert(pcFilename, ulLine);
          }
```

(1) The BSP for Cortex-M3 relies on the driver library provided by Luminary Micro with the EV-LM3S811 board.

(2) As described in Section 8.2.3 in Chapter 8, ISRs in Cortex-M3 are just regular C functions. The system clock tick is implemented with the Cortex-M3 SysTick interrupt, specifically designed for that purpose. Note that the Cortex-M3 enters ISRs with interrupts unlocked, so there is no need to unlock interrupts before calling QF services, such as `QF_tick()`.

(3-5)The board initialization includes enabling all peripherals used in the DPP application.

(6-8)The graphic OLED display driver is initialized and the screen is prepared for the DPP application.

(9) The output of the philosopher status is implemented as drawing a single letter on the screen (see Figure 9.6). Note that the output occurs only from the context of the Table active object.

(10)Upon startup, the hardware system clock tick rate is set.

(11)The system clock tick hardware is enabled.

(12)The Cortex-M3 performs prioritization of all interrupts in hardware, and it is highly recommended to explicitly set the priority of every interrupt used by the application. The Cortex-M3 represents an ISR priority in the three most significant bits of a byte, whereas 0xE0 is the lowest and 0x00 is the highest hardware priority. Priority 0xC0 corresponds to the second-lowest priority in the system.

(13)The system clock tick interrupt is enabled in hardware.

(14)The interrupts are enabled.

(15)The DPP application running on the EV-LM3S811 board operates on "bare metal" and has no operating system to return to. The cleanup callback is not used in this case.

(16-18)In Section 8.2.4, I have already discussed idle processing for the "vanilla" kernel running on Cortex-M3.

(19)The assertion handler enters a forever loop in the DPP application. You need to replace this loop with the fail-safe shutdown, followed perhaps by a reset in the production version of your application.

(20)The function `__error__()` is used inside the Luminary Micro driver library. This function has the same purpose and signature as `Q_onAssert()`.

Read full chapter

URL: https://www.sciencedirect.com/science/article/pii/B978075068706500009X