Nested Vector Interrupt Controller (NVIC)

Objective

This tutorial demonstrates how to use interrupts on a processor. In general, you will understand the concept behind interrupts on any processor, but we will use the SJ-One board as an example.

What is an interrupt?

An interrupt is the hardware capability of a CPU to break the normal flow of software to attend an urgent request.

The science behind interrupts lies in the hardware that allows the CPU to be interrupted. Each peripheral in a microcontroller *may be* able to assert an interrupt to the CPU core, and then the CPU core would jump to the corresponding interrupt service routine (**ISR**) to service the interrupt.

ISR Procedure

The following steps demonstrate what happens when an interrupt occurs :

- CPU manipulates the PC (program counter) to jump to the ISR
- **IMPORTANT**: CPU will disable interrupts (or that priority level's interrupts until end of ISR)
- Registers are saved before running the ISR (pushed onto the stack)
- ISR is run
- Registers are restored (popped from stack)
- Interrupts are re-enabled (or that priority level's interrupt is re-enabled)

On some processors, the savings and restoring of registers is a manual step and the compiler would help you do it. You can google "GCC interrupt attribute" to study this topic further. On SJ-One board, which uses LPC17xx (ARM Cortex M3), this step is automatically taken care of by the CPU hardware.

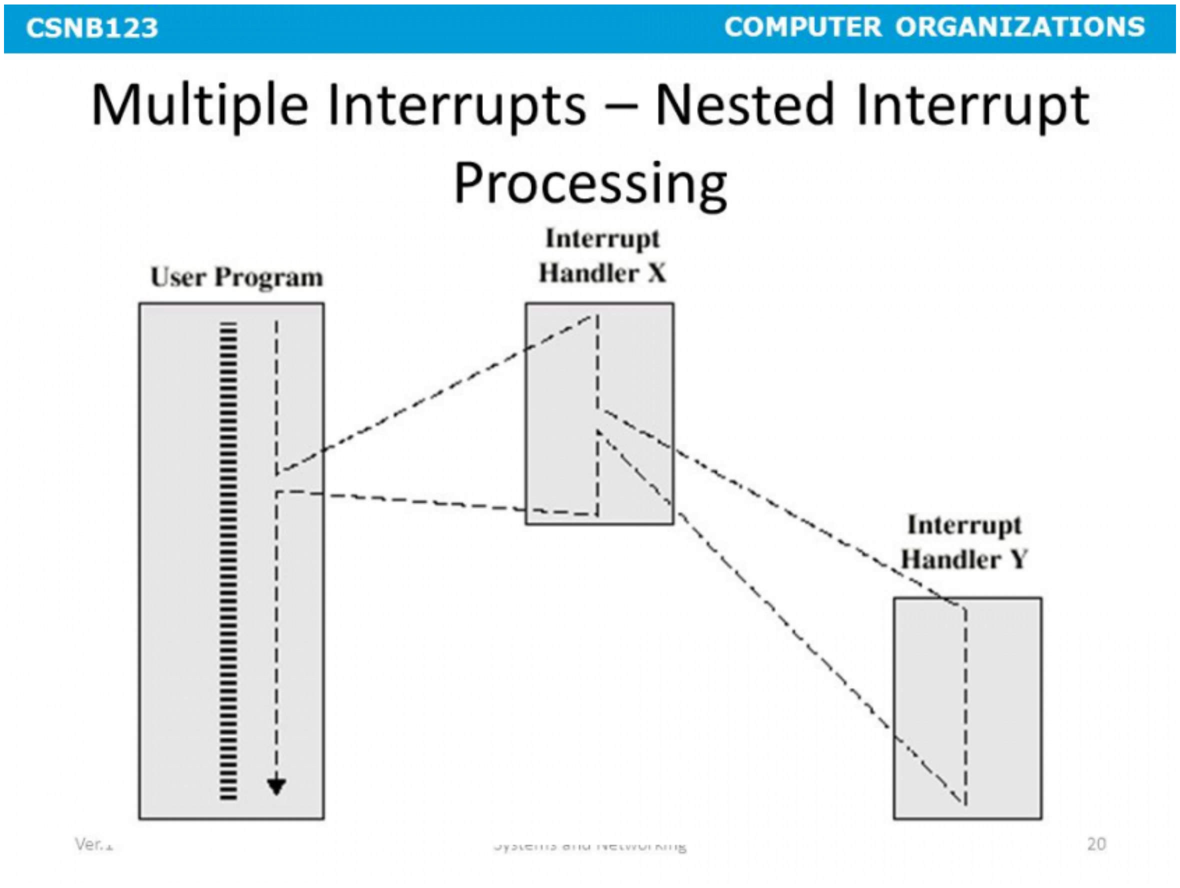


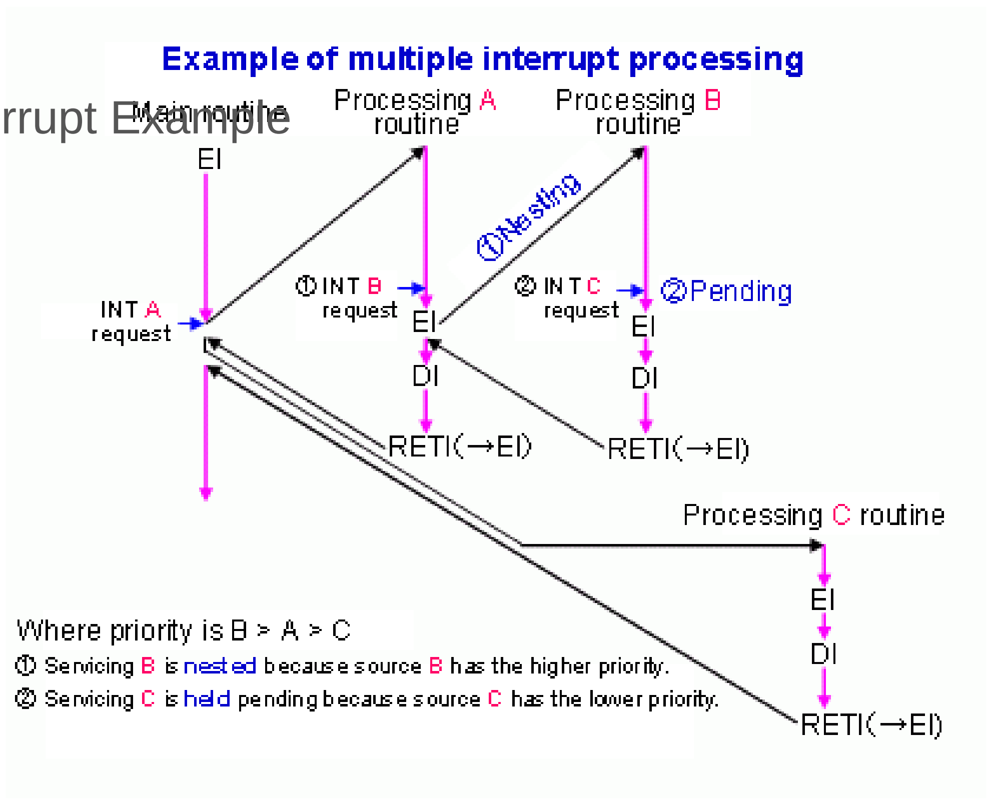
Figure 1. Nested Interrupt Processing

Nested Vector Interrupt Controller

Nested Vector Interrupt Controllers or NVIC for short, have two properties:

- Can handle multiple interrupts.
  - The number of interrupts implemented is device dependent.
- A programmable priority level for each interrupt.
  - A higher level corresponds to a lower priority, so level 0 is the highest interrupt priority.
- Level and pulse detection of interrupt signals.
- Grouping of priority values into group priority and sub-priority fields.
  - This means that interrupts of the same priority are grouped together and do not preempt each other.
  - Each interrupt also has a sub-priority field which is used to figure out the run order of pending interrupts of the same priority.
- Interrupt tail-chaining.
  - This enables back-to-back interrupt processing without the overhead of state saving and restoration between interrupts.
  - This saves us from the step of having to restore and then save the registers again.
- An external Non-maskable interrupt (NMI)

NVIC Interrupt Example



Example of multiple interrupt processing
Main routine
Processing A routine
Processing B routine
EI
①Nesting
INT A request
① INT B request
EI
② INT C request
②Pending
EI
DI
DI
RETI(→EI)
RETI(→EI)
Processing C routine
EI
DI
RETI(→EI)
Where priority is B > A > C
① Servicing B is nested because source B has the higher priority.
② Servicing C is held pending because source C has the lower priority.

Figure 2. Multiple Interrupt Processing

The SW to HW Connection

Now that we understand how the CPU hardware services interrupts, we need to define how we inform the CPU WHERE our ISR function is located at.

Interrupt Vector Table

This table is nothing but addresses of functions that correspond to the microcontroller interrupts. Specific interrupts use specific "slots" in this table, and we have to populate these spots with our software functions that service the interrupts.

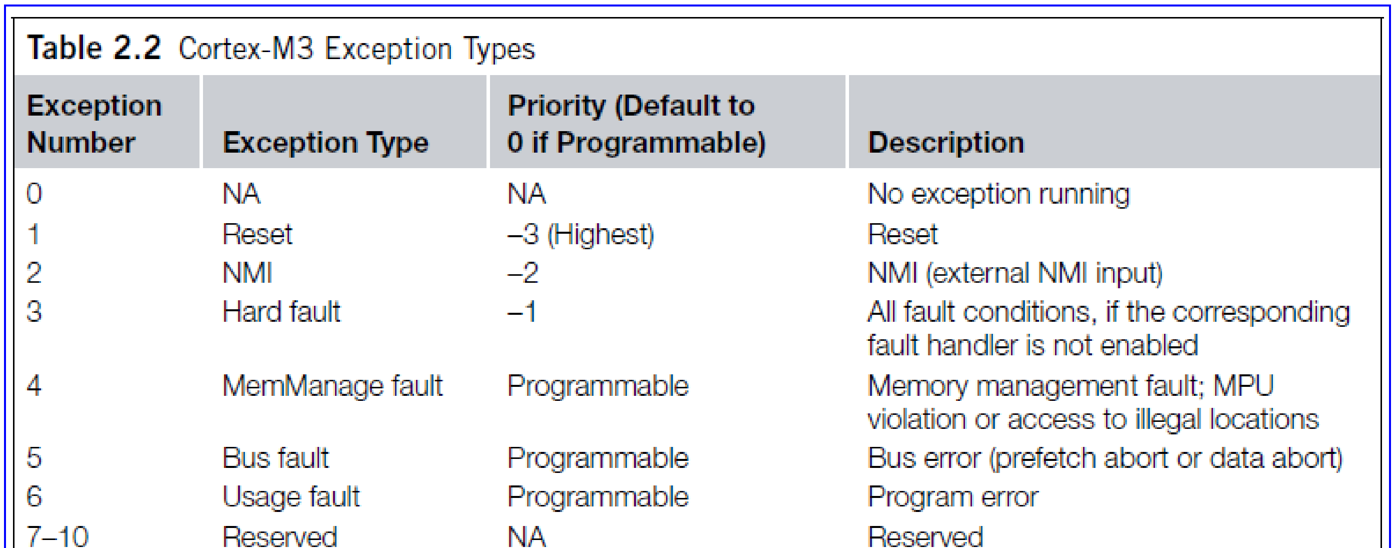


Table 2.2 Cortex-M3 Exception Types

Exception Number	Exception Type	Priority (Default to 0 if Programmable)	Description
0	NA	NA	No exception running
1	Reset	–3 (Highest)	Reset
2	NMI	–2	NMI (external NMI input)
3	Hard fault	–1	All fault conditions, if the corresponding fault handler is not enabled
4	MemManage fault	Programmable	Memory management fault; MPU violation or access to illegal locations
5	Bus fault	Programmable	Bus error (prefetch abort or data abort)
6	Usage fault	Programmable	Program error
7–10	Reserved	NA	Reserved

Figure 3. HW Interrupt Vector Table

SJOne (LPC17xx) Example

The using a linker script and compiler directives (commands for the compiler), the compiler is able to place the software interrupt vector table at a specific location that the CPU expects the interrupt vector table to be located at. This connects the dots about how the CPU is able to determine WHERE your interrupt service routines are located at. From there on, anytime a specific interrupt occurs, the CPU is able to fetch the address and make the JUMP.

```
/**
 * CPU interrupt vector table that is loaded at the beginning of the CPU start
 * location by using the linker script that will place it at the isr_vector location.
 * CPU loads the stack pointer and begins execution from Reset vector.
 */
extern void (* const g_pfnVectors[])(void);
__attribute__ ((section(".isr_vector")))
void (* const g_pfnVectors[])(void) =
{
    // Core Level - CM3
    &_vStackTop,        // The initial stack pointer
    isr_reset,          // The reset handler
    isr_nmi,            // The NMI handler
    isr_hard_fault,     // The hard fault handler
    isr_mem_fault,      // The MPU fault handler
    isr_bus_fault,      // The bus fault handler
    isr_usage_fault,    // The usage fault handler
    0,                  // Reserved
    0,                  // Reserved
    0,                  // Reserved
    0,                  // Reserved
    vPortSVCHandler,    // FreeRTOS SVC-call handler (naked function so needs direct call - not a wrap
    isr_debug_mon,      // Debug monitor handler
    0,                  // Reserved
    xPortPendSVHandler, // FreeRTOS PendSV handler (naked function so needs direct call - not a wrappe
    isr_sys_tick,       // FreeRTOS SysTick handler (we enclose inside a wrapper to track OS overhead)
    // Chip Level - LPC17xx - common ISR that will call the real ISR
    isr_forwarder_routine,      // 16, 0x40 - WDT
    isr_forwarder_routine,      // 17, 0x44 - TIMER0
    isr_forwarder_routine,      // 18, 0x48 - TIMER1
    isr_forwarder_routine,      // 19, 0x4c - TIMER2
    isr_forwarder_routine,      // 20, 0x50 - TIMER3
    isr_forwarder_routine,      // 21, 0x54 - UART0
    isr_forwarder_routine,      // 22, 0x58 - UART1
    isr_forwarder_routine,      // 23, 0x5c - UART2
    isr_forwarder_routine,      // 24, 0x60 - UART3
    isr_forwarder_routine,      // 25, 0x64 - PWM1
    isr_forwarder_routine,      // 26, 0x68 - I2C0
    isr_forwarder_routine,      // 27, 0x6c - I2C1
    isr_forwarder_routine,      // 28, 0x70 - I2C2
    isr_forwarder_routine,      // 29, 0x74 - SPI
    isr_forwarder_routine,      // 30, 0x78 - SSP0
    isr_forwarder_routine,      // 31, 0x7c - SSP1
    isr_forwarder_routine,      // 32, 0x80 - PLL0 (Main PLL)
    isr_forwarder_routine,      // 33, 0x84 - RTC
    isr_forwarder_routine,      // 34, 0x88 - EINT0
    isr_forwarder_routine,      // 35, 0x8c - EINT1
    isr_forwarder_routine,      // 36, 0x90 - EINT2
    isr_forwarder_routine,      // 37, 0x94 - EINT3
    isr_forwarder_routine,      // 38, 0x98 - ADC
    isr_forwarder_routine,      // 39, 0x9c - BOD
    isr_forwarder_routine,      // 40, 0xA0 - USB
    isr_forwarder_routine,      // 41, 0xa4 - CAN
    isr_forwarder_routine,      // 42, 0xa8 - GP DMA
    isr_forwarder_routine,      // 43, 0xac - I2S
    isr_forwarder_routine,      // 44, 0xb0 - Ethernet
    isr_forwarder_routine,      // 45, 0xb4 - RITINT
    isr_forwarder_routine,      // 46, 0xb8 - Motor Control PWM
    isr_forwarder_routine,      // 47, 0xbc - Quadrature Encoder
    isr_forwarder_routine,      // 48, 0xc0 - PLL1 (USB PLL)
    isr_forwarder_routine,      // 49, 0xc4 - USB Activity interrupt to wakeup
    isr_forwarder_routine,      // 50, 0xc8 - CAN Activity interrupt to wakeup};
```

Code Block 1. Software Interrupt Vector Table

NOTE: that a vector table is really just a lookup table that hardware utilizes.

Two Methods to setup an ISR on the SJOne

All of the methods require that you run this function to allow the NVIC to accept a particular interrupt request.

NVIC_EnableIRQ(EINT3_IRQn);

Where the input is the IRQ number. This can be found in the LCP17xx.h file. Search for **enum IRQn**.

Method 1. Modify IVT

DO NOT DO THIS, unless you really know what you are doing. The ISR forwarder works with FreeRTOS to distinguish CPU utilization between ISRs and tasks.

I highly discourage modifying the **startup.cpp** and modifying the vector tables directly. Its not dynamic is less manageable in that, if you switch projects and the ISR doesn't exist, the compiler will through an error.

IVT modify

```
/* You will need to include the header file that holds the ISR for this to work */
#include "my_isr.h"
extern void (* const g_pfnVectors[])(void);
__attribute__ ((section(".isr_vector")))
void (* const g_pfnVectors[])(void) =
{
    // Core Level - CM3
    &_vStackTop,        // The initial stack pointer
    isr_reset,          // The reset handler
    isr_nmi,            // The NMI handler
    isr_hard_fault,     // The hard fault handler
    isr_mem_fault,      // The MPU fault handler
    isr_bus_fault,      // The bus fault handler
    isr_usage_fault,    // The usage fault handler
    0,                  // Reserved
    0,                  // Reserved
    0,                  // Reserved
    0,                  // Reserved
    vPortSVCHandler,    // FreeRTOS SVC-call handler (naked function so needs direct call - not a wrapp
    isr_debug_mon,      // Debug monitor handler
    0,                  // Reserved
    xPortPendSVHandler, // FreeRTOS PendSV handler (naked function so needs direct call - not a wrappe
    isr_sys_tick,       // FreeRTOS SysTick handler (we enclose inside a wrapper to track OS overhead)
    // Chip Level - LPC17xx - common ISR that will call the real ISR
    isr_forwarder_routine,      // 16, 0x40 - WDT
    isr_forwarder_routine,      // 17, 0x44 - TIMER0
    isr_forwarder_routine,      // 18, 0x48 - TIMER1
    isr_forwarder_routine,      // 19, 0x4c - TIMER2
    isr_forwarder_routine,      // 20, 0x50 - TIMER3
    isr_forwarder_routine,      // 21, 0x54 - UART0
    isr_forwarder_routine,      // 22, 0x58 - UART1
    isr_forwarder_routine,      // 23, 0x5c - UART2
    isr_forwarder_routine,      // 24, 0x60 - UART3
    isr_forwarder_routine,      // 25, 0x64 - PWM1
    isr_forwarder_routine,      // 26, 0x68 - I2C0
    isr_forwarder_routine,      // 27, 0x6c - I2C1
    isr_forwarder_routine,      // 28, 0x70 - I2C2
    isr_forwarder_routine,      // 29, 0x74 - SPI
    isr_forwarder_routine,      // 30, 0x78 - SSP0
    isr_forwarder_routine,      // 31, 0x7c - SSP1
    isr_forwarder_routine,      // 32, 0x80 - PLL0 (Main PLL)
    isr_forwarder_routine,      // 33, 0x84 - RTC
    isr_forwarder_routine,      // 34, 0x88 - EINT0
    isr_forwarder_routine,      // 35, 0x8c - EINT1
    isr_forwarder_routine,      // 36, 0x90 - EINT2
    runMyISR,                   // 37, 0x94 - EINT3 <---- NOTICE how I changed the name here
    isr_forwarder_routine,      // 38, 0x98 - ADC
    isr_forwarder_routine,      // 39, 0x9c - BOD
    isr_forwarder_routine,      // 40, 0xA0 - USB
    isr_forwarder_routine,      // 41, 0xa4 - CAN
    isr_forwarder_routine,      // 42, 0xa8 - GP DMA
    isr_forwarder_routine,      // 43, 0xac - I2S
    isr_forwarder_routine,      // 44, 0xb0 - Ethernet
    isr_forwarder_routine,      // 45, 0xb4 - RITINT
    isr_forwarder_routine,      // 46, 0xb8 - Motor Control PWM
    isr_forwarder_routine,      // 47, 0xbc - Quadrature Encoder
    isr_forwarder_routine,      // 48, 0xc0 - PLL1 (USB PLL)
    isr_forwarder_routine,      // 49, 0xc4 - USB Activity interrupt to wakeup
    isr_forwarder_routine,      // 50, 0xc8 - CAN Activity interrupt to wakeup};
```

Code Block 3. Weak Function Override Template

Method 2. ISR Register Function

The **EINT3_IRQn** symbol is defined in an enumeration in LPC17xx.h. All you need to do is specify the IRQ number and the function you want to act as an ISR. This will then swap out the previous ISR with your function.

> This is the best option! Please use this option almost always!

```
/**
 * Just your run-of-the-mill function
 */
void myEINT3ISR(void)
{
    doSomething();
    clearInterruptFlag();
}
int main()
{
    /**
     * Find the IRQ number for the interrupt you want to define.
     * In this case, we want to override IRQ 0x98 EINT3
     * Then specify a function pointer that will act as your ISR
     */
    isr_register(EINT3_IRQn, myEINT3ISR);
    NVIC_EnableIRQ(EINT3_IRQn);}
```

Code Block 5. Weak Function Override Template

PROS	CONS
• Can dynamically change ISR during runtime. • Does not disturb core library files in the process of adding/changing ISRs. ◦ Always try to prevent changes to the core libraries. • Does not cause compiler errors. • Your ISR cpu utilization is tracked.	• Must wait until **main** is called before ISR is registered ◦ Interrupt events could happen before main begins.

What to do inside an ISR

Do very little inside an ISR. When you are inside an ISR, the whole system is blocked (other than higher priority interrupts). If you spend too much time inside the ISR, then you are destroying the real-time operating system principle and everything gets clogged.

With that said, here is the general guideline:

Short as possible

DO NOT POLL FOR ANYTHING! Try to keep loops as small as possible. Note that printing data over UART can freeze the entire system, including the RTOS for that duration. For instance, printing 4 chars may take 1ms at 38400bps.

FreeRTOS API calls

If you are using FreeRTOS API, you must use **FromISR** functions only! If a FromISR function does not exist, then don't use that API.

Clear Interrupt Sources

Clear the source of the interrupt. For example, if interrupt was for rising edge of a pin, clear the "rising edge" bit such that you will not re-enter into the same interrupt function.

> If you don't do this, your interrupt will get stuck in an infinite ISR call loop. For the Port interrupts, this can be done by writing to the IntClr registers.

?

ISR processing inside a FreeRTOS Task

It is a popular scheme to have an ISR quickly exit, and then resume a task or thread to process the event. For example, if we wanted to write a file upon a button press, we don't want to do that inside an ISR because it would take too long and block the system. What we can utilize a **wait on semaphore** design pattern.

What you may argue with the example below is that we do not process the ISR immediately, and therefore delay the processing. But you can tackle this scenario by resuming a HIGHEST priority task. Immediately, after the ISR exits, due to the ISR "yield", FreeRTOS will resume the high priority task immediately rather than servicing another task

```
/* Create the semaphore in main() */
SemaphoreHandle_t button_press_semaphore = NULL;
void myButtonPressISR(void)
{
    long yield = 0;
    xSemaphoreGiveFromISR(button_press_semaphore, &yield);
    portYIELD_FROM_ISR(yield);
}
void vButtonPressTask(void *pvParameter)
{
    while(1)
    {
        if (xSemaphoreTake(button_press_semaphore, portMAX_DELAY))
        {
            /* Process the interrupt */
        }
    }
}
void main(void)
{
    button_press_semaphore = xSemaphoreCreateBinary();
    /* TODO: Hook up myButtonPressISR() using eint.h */
    /* TODO: Create vButtonPressTask() and start FreeRTOS scheduler */}
```

Code Block 6. Wait on Semaphore ISR design pattern example

Resources

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0489b/CACDDJHB.html

---

Revision #17
Created 1 year ago by Admin

Updated 11 months ago by Preet Kang