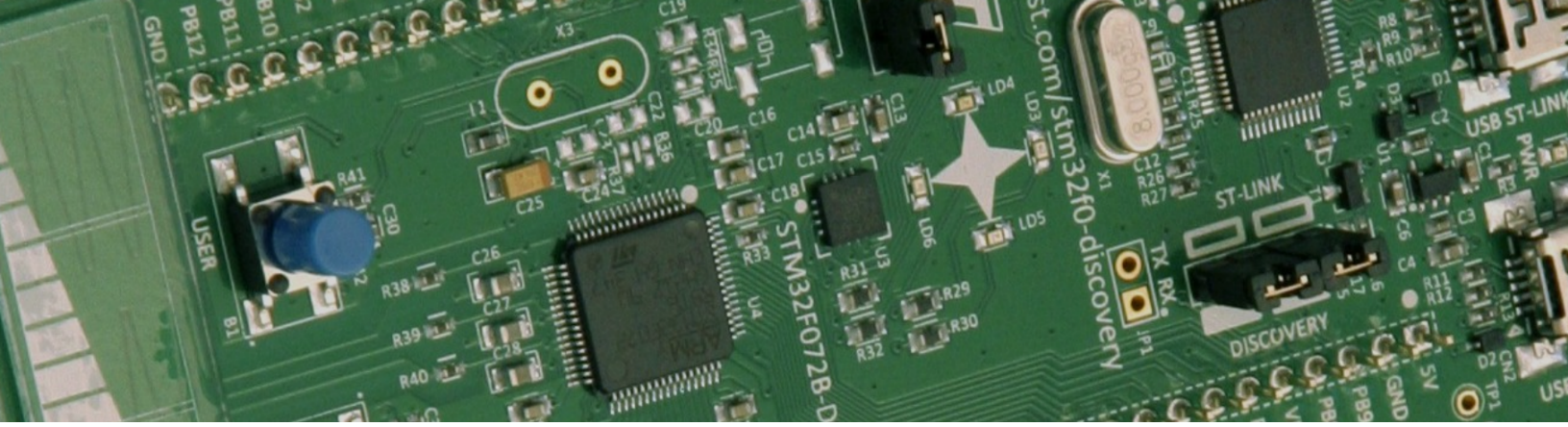
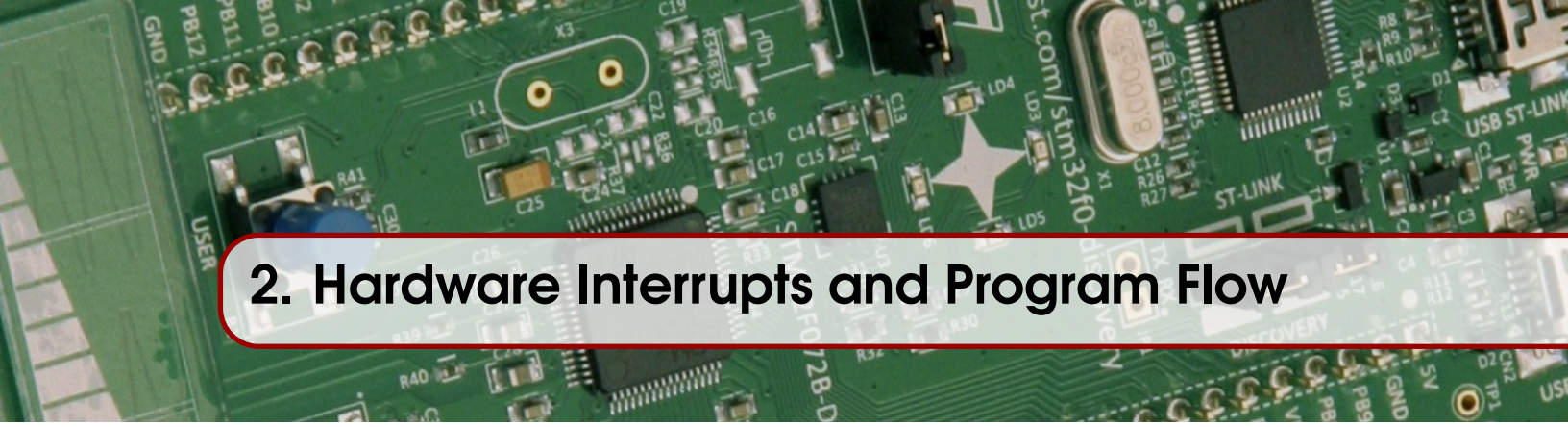



# Contents



# 2. Hardware Interrupts and Program Flow

## 2.1 Overview

This lab introduces the concept of interrupt-driven programming and guides through the configuration of interrupt-oriented peripherals; the exercises herein provide a foundation for utilizing interrupts in an embedded application. They introduce the practice of enabling, configuring parameters and writing handler routines to service peripheral interrupt requests. After completing this lab, you will understand how to use interrupts effectively without impacting the main application or each other.

## 2.2 Prelab Questions

**2.1 — Prelab 2.** Please answer the following questions and hand in as your prelab for Lab 2.

1. What is the purpose of the NVIC peripheral?
2. What is the difference between interrupt tail-chaining and nesting?
3. In what file are the CMSIS libraries that control the NVIC?
4. What is the purpose of the EXTI peripheral?
5. What is the purpose of the SYSCFG pin multiplexers?
6. What file has the defined names for interrupt numbers?
7. What file has the Vector table implementation?

■

## 2.3 Introduction to Hardware Interrupts

Many embedded processors—including the ARM Cortex-M0 STM32F0 family—are single-core, single-thread devices; however, many embedded applications are not a single, linear thread: these programs typically operate at low enough abstractions such that most operating system concepts—such as scheduling or multi-threading—simply do not exist. Instead, the processor hardware directly drives the program concurrency, using a method known as *interrupts*.

An interrupt is the process in which the hardware temporarily suspends the execution of the main single threaded program to execute specific regions of code at known locations in memory. We call them interrupts because these program jumps to "interrupt" the main program. Figure 2.1 demonstrates the basic operation of an interrupt in a system such as the STM32F0.

Peripherals within the embedded device typically generate these interrupts; these events signal a change in the peripheral state—such as receiving data from a communications interface. Other interrupts signal error conditions or recover from bad processor states. The user's code may use interrupts to perform operations with a higher priority than the main thread.

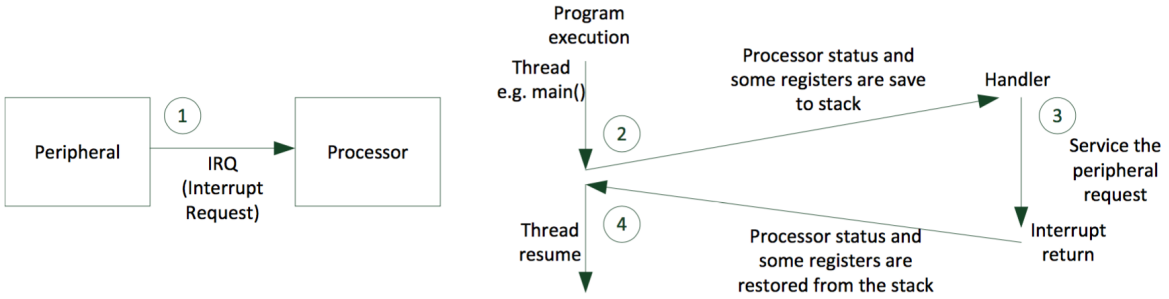



Figure 2.1: Operation of an Interrupt

Every interrupt has a hardware number designation: an interrupt's number indicates its hardware priority and indexes into the *Vector Table*. The Vector Table for a processor usually exists at the beginning of the system address space and is a list of memory addresses associated with the handling of a specific interrupt. For example, the location of the RESET vector (in actuality a RESET interrupt) in the Vector Table is right at the beginning. This indicates that the first few instructions that the processor executes after power-on are a load and branch to the reset handling code.

Whenever an interrupt triggers, the processor hardware uses the interrupt number to index into the Vector Table to find the memory address of the interrupt handling code. The processor hardware saves the current register and stack state before branching to the loaded handler address. After the routine completes, the processor restores the original state, and the main program executes almost as if no interrupt had occurred.

Figure 2.2 (peripheral reference manual pages 217-219) shows the documentation for the STM32F072 Vector Table. The Vector Table lies within the startup assembly code for the processor; it defines human-readable names used to designate functions as the appropriate interrupt handling code. When compiling and linking, the toolchain places the address of these functions within the Vector Table data.

The Kiel:MDK toolchain and HAL library have already defined a few interrupt handlers within the *stmstm32f0xx_it.c* file located under the *Application/User* µVision project folder. The device startup code and Vector Table implementation are located in the *startup_stm32f072xb.s* file within the *Application/MDK-ARM* directory. These files and how to use them will be discussed throughout the lab.

### 2.3.1 The SysTick Timer and Interrupt

The SysTick timer is an important but simple peripheral within most ARM processors. It consists of a 16-bit countdown timer which starts at a software configured value; this value decrements alongside the processor clock, and reaching zero triggers an interrupt, and the peripheral resets. The SysTick interrupt is often useful as a time reference for application code, an example of this is the `HAL_Delay()` function mentioned in the previous lab.

Figure 2.3 shows how the SysTick interrupt pauses the main application to execute a small section of interrupt handler code.Due to a constant delay between successive executions of the handler, you may calculate timing accurately with longer intervals that are multiples of the SysTick period by counting the number of interrupt executions.

| Position | Priority | Type of priority | Acronym | Description | Address |
|---|---|---|---|---|---|
| - | - | - | - | Reserved | 0x0000 0000 |
| - | -3 | fixed | Reset | Reset | 0x0000 0004 |
| - | -2 | fixed | NMI | Non maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector. | 0x0000 0008 |
| - | -1 | fixed | HardFault | All class of fault | 0x0000 000C |
| - | 3 | settable | SVCall | System service call via SWI instruction | 0x0000 002C |
| - | 5 | settable | PendSV | Pendable request for system service | 0x0000 0038 |
| - | 6 | settable | SysTick | System tick timer | 0x0000 003C |
| 0 | 7 | settable | WWDG | Window watchdog interrupt | 0x0000 0040 |
| 1 | 8 | settable | PVD_VDDIO2 | PVD and $V_{DDIO2}$ supply comparator interrupt (combined EXTI lines 16 and 31) | 0x0000 0044 |
| 2 | 9 | settable | RTC | RTC interrupts (combined EXTI lines 17, 19 and 20) | 0x0000 0048 |
| 3 | 10 | settable | FLASH | Flash global interrupt | 0x0000 004C |
| 4 | 11 | settable | RCC_CRS | RCC and CRS global interrupts | 0x0000 0050 |
| 5 | 12 | settable | EXTI0_1 | EXTI Line[1:0] interrupts | 0x0000 0054 |
| 6 | 13 | settable | EXTI2_3 | EXTI Line[3:2] interrupts | 0x0000 0058 |
| 7 | 14 | settable | EXTI4_15 | EXTI Line[15:4] interrupts | 0x0000 005C |
| 8 | 15 | settable | TSC | Touch sensing interrupt | 0x0000 0060 |
| 9 | 16 | settable | DMA_CH1 | DMA channel 1 interrupt | 0x0000 0064 |
| 10 | 17 | settable | DMA_CH2_3<br>DMA2_CH1_2 | DMA channel 2 and 3 interrupts<br>DMA2 channel 1 and 2 interrupts | 0x0000 0068 |
| 11 | 18 | settable | DMA_CH4_5_6_7<br>DMA2_CH3_4_5 | DMA channel 4, 5, 6 and 7 interrupts<br>DMA2 channel 3, 4 and 5 interrupts | 0x0000 006C |
| 12 | 19 | settable | ADC_COMP | ADC and COMP interrupts (ADC interrupt combined with EXTI lines 21 and 22) | 0x0000 0070 |
| 13 | 20 | settable | TIM1_BRK_UP_ TRG_COM | TIM1 break, update, trigger and commutation interrupt | 0x0000 0074 |
| 14 | 21 | settable | TIM1_CC | TIM1 capture compare interrupt | 0x0000 0078 |
| 15 | 22 | settable | TIM2 | TIM2 global interrupt | 0x0000 007C |
| 16 | 23 | settable | TIM3 | TIM3 global interrupt | 0x0000 0080 |
| 17 | 24 | settable | TIM6_DAC | TIM6 global interrupt and DAC underrun interrupt | 0x0000 0084 |
| 18 | 25 | settable | TIM7 | TIM7 global interrupt | 0x0000 0088 |
| 19 | 26 | settable | TIM14 | TIM14 global interrupt | 0x0000 008C |
| 20 | 27 | settable | TIM15 | TIM15 global interrupt | 0x0000 0090 |
| 21 | 28 | settable | TIM16 | TIM16 global interrupt | 0x0000 0094 |
| 22 | 29 | settable | TIM17 | TIM17 global interrupt | 0x0000 0098 |
| 23 | 30 | settable | I2C1 | $I^2C1$ global interrupt (combined with EXTI line 23) | 0x0000 009C |
| 24 | 31 | settable | I2C2 | $I^2C2$ global interrupt | 0x0000 00A0 |
| 25 | 32 | settable | SPI1 | SPI1 global interrupt | 0x0000 00A4 |
| 26 | 33 | settable | SPI2 | SPI2 global interrupt | 0x0000 00A8 |
| 27 | 34 | settable | USART1 | USART1 global interrupt (combined with EXTI line 25) | 0x0000 00AC |
| 28 | 35 | settable | USART2 | USART2 global interrupt (combined with EXTI line 26) | 0x0000 00B0 |
| 29 | 36 | settable | USART3_4_5_6_7_ 8 | USART3, USART4, USART5, USART6, USART7, USART8 global interrupts (combined with EXTI line 28) | 0x0000 00B4 |
| 30 | 37 | settable | CEC_CAN | CEC and CAN global interrupts (combined with EXTI line 27) | 0x0000 00B8 |
| 31 | 38 | settable | USB | USB global interrupt (combined with EXTI line 18) | 0x0000 00BC |

Figure 2.2: STM32F072 Vector Table

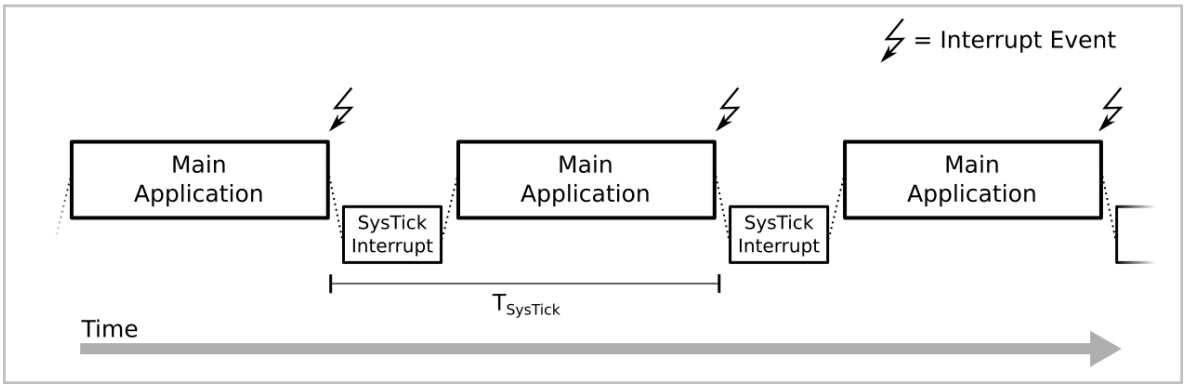



Figure 2.3: Effect of the SysTick interrupt on application execution.

By default, the HAL library configures the SysTick timer to a rate of 1000 Hz, giving a 1 ms period between each interrupt. The `HAL_Delay()` function operates by monitoring a global variable which increments within the SysTick interrupt handler; since the variable increases by one each millisecond, the delay function can stall the processor for a desired time by watching and waiting until the variable equals a calculated target amount.

**2.1 — Modifying an Existing Interrupt.** In this exercise, you will modify the SysTick timer interrupt to flash the blue LED on the Discovery board; you should begin this lab by generating a blank project within STMCube.

### Preparing the Main Application

1. Initialize all of the LED pins in the main function.
   - Run-once code such as initializations should execute before the main infinite loop; never use an interrupt handler to do this!
2. Set the green LED (PC9) high (we will use this later in the lab)
3. Toggle the red LED (PC6) with a moderately-slow delay (400-600ms) in the infinite loop.
   - This LED indicates whether the main loop is executing; use it to determine if the system is stuck in an interrupt.

### Modifying the SysTick Interrupt

1. Find the `void SysTick_Handler(void)` function in the *stm32f0xx_it.c* file.
   - Look under the *Application/User* µVision project folder.
   - This file contains pre-generated interrupt handlers.
2. Modify the SysTick handler so that it toggles the blue LED (PC7) every 200ms.
   - The HAL library uses the SysTick timer. Do not remove any pre-generated code in the handler.
3. The modified SysTick handler interrupts every millisecond; you can use this periodicity to count the number of iterations as a timing mechanism.

- For example, toggling an LED every 100th execution of the interrupt results in 100 ms between blinks.
- You will need to use either a volatile global or local-static variable to store interrupt count.
- **Do not use delay functions in the interrupt handler!** See warning below.

4. Compile and load your application onto the Discovery board. Remember to press the **RESET** button after it loads!

Both the red and blue LEDs should be flashing at different rates. If the blue LED appears to stay constant and not toggle, check to ensure that you are only toggling it every 200ms. ■

**Never use any sort of delay within an interrupt handler!** Handler functions should perform work quickly and then return—the HAL delay functions will deadlock within an interrupt with the same or higher priority than the SysTick.

## 2.4 Triggering Interrupts With External Signals

In the previous lab, we used a button press on the Discovery board to toggle between two LEDs. We did this by repeatedly checking the button state in the infinite loop of the main application; we call this method *polling*. Polling has the advantage that the repetitive and periodic check enables tricks like software debouncing; it has the disadvantage, however, of using a significant number of processor cycles even when the device could otherwise sit idle.

In some embedded systems—such as the Discovery board—wasting energy on polling is not a significant challenge as we have a constant supply of power; many battery-powered systems, however, must reduce power consumption to prolong the battery life and therefore field service.

One method of avoiding continuous polling is to leverage the interrupt system to monitor and detect changes in a pin's state; this makes it possible to place the device into a low-power mode when no other processing is necessary.

The exercises in this lab uses the "Wait for Interrupt" (WFI) assembly instruction, where the processor enters into "sleep" mode (without changing other registers). This is the least drastic of the low-power modes that the STM32F0 offers. In this state the ARM processor stops, but all memory and peripherals operate normally. Any hardware interrupt has the capability to start the processor again; once the interrupt handler exits, the main program will continue the main application thread.

Other low-power modes selectively shut down additional peripherals, system oscillators, and power circuitry. These modes are more limited in the methods available to wake them up—some of them may even lose device state! It is therefore imperative that you use the correct low-power mode for your application.

### 2.4.1 Extended Interrupts and Events Controller

The *Extended Interrupts and Events Controller* (EXTI) peripheral allows non-peripheral sources to trigger interrupts. While its typical use is to generate interrupts from the GPIO pins of the device, it may also monitor various internal signals such as the brownout protection circuitry (low-voltage shutdown).

The EXTI documentation begins on page 219 (Section 12.2) of the peripheral reference manual. Similar to the NVIC, bits within the EXTI registers do not feature names suggesting the signals they control. The documentation within the *functional description* section on the peripheral describes the mapping between EXTI event "lines"—input sources—and the control bits.

- **Interrupt mask register (EXTI_IMR)**
  - The IMR register "unmasks" or enables an input signal to generate one of the EXTI interrupts.
- **Event mask register (EXTI_EMR)**
  - Processor events are similar in design to interrupts but do not cause program execution to branch to separate handler code; events generally wake the processor from low-power modes. The EMR enables input signals to generate processor events.
- **Rising trigger selection register (EXTI_RTSR)**
  - All external (pin) interrupts are edge-sensitive, which means that they only generate interrupt requests at the transitions from one logic state to another. The RTSR enables a rising/positive-edge trigger for a pin.
- **Falling trigger selection register (EXTI_FTSR)**
  - The FTSR enables a negative/falling-edge trigger for a pin. The EXTI allows enabling both rising and falling triggers for inputs.
- **Software interrupt event register (EXTI_SWIER)**
  - The SWIER register allows the user to trigger any of the interrupt or event conditions within the EXTI (as long as the matching bits in the IMR or EMR registers are also set).
- **Pending register (EXTI_PR)**
  - The pending register indicates whether an event has occurred on an input signal since the last clear. If set, the EXTI interrupt handler will trigger repeatedly until the corresponding pending flags clear. Note that some events will self-clear this bit, so manual clearing is unnecessary.

**2.2 — Configuring the EXTI.** This next exercise uses the EXTI peripheral to generate interrupts on the rising-edge of the user button (PA0). Place this code in the main function **before** the infinite loop.

1. Configure the button pin (PA0) to input-mode at low-speed, with the internal pull-down resistor enabled.
2. Pin PA0 connects to the EXTI input line 0 (EXTI0).
   - We will explore the relationship between pins and the EXTI input lines in the next section.
   - The first 16 inputs to the EXTI are for external interrupts; for example, EXTI3 is the 3rd input line.
3. Enable/unmask interrupt generation on EXTI input line 0 (EXTI0).
4. Configure the EXTI input line 0 to have a rising-edge trigger.
   - The peripheral reference manual documents EXTI in section 12.2 (page 219).

Due to the low-level wakeup functions it provides, the EXTI peripheral always connects to the peripheral clock, so it is not necessary to enable via the RCC. ■

### 2.4.2 Pin Multiplexing with the SYSCFG

Although the STM32F0 family has the ability to generate external interrupts on almost any pin, only 16 available input lines connect to the EXTI; therefore, a series of pin multiplexers are necessary to select the pins that connect to the limited EXTI inputs.

The *System Configuration Controller* (SYSCFG) peripheral controls these multiplexers; the SYSCFG deals primarily with signal routing, and controls data transfer between peripherals and memory, remapping portions of memory, and some high-power communication modes.

Figure 2.4 shows the SYSCFG pin multiplexers that the EXTI uses; these multiplexers group external pins by their orderings within the GPIO peripherals—for example, PA0, PB0 ... PF0 are on a single multiplexer with the output routed to the EXTI0 input.

Since only a single pin from a group may be in use, select pins such that they do not conflict with each other when using multiple external interrupts. Careful planning and forethought is necessary to avoid such conflicts in your designs, but it is much simpler to address those issues before you begin programming than if you choose to address them in the later design stages.

Use the EXTIx registers within the SYSCFG peripheral to configure the multiplexers. The register maps for these begin on page 177 (Section 10.1.2) of the peripheral reference manual.

**2.3 — Setting the SYSCFG Pin Multiplexer.** In the previous exercise you configured the EXTI to generate an interrupt on the rising edge of its input line 0. In order to get the external interrupt working with the actual button pin (PA0), you must configure the SYSCFG pin multiplexers to connect the two signals together.

1. Use the RCC to enable the peripheral clock to the SYSCFG peripheral.
2. Determine which SYSCFG multiplexer can route PA0 to the EXTI peripheral.
   - Information about the SYSCFG is in section 10 of the peripheral reference manual (page 173).
   - Each multiplexer indicates the input line/signal of the EXTI to which they connect.
3. Each of the EXTICRx registers control multiple pin multiplexers. Find which register contains the configuration bits for the required multiplexer.
4. Configure the multiplexer to route PA0 to the EXTI input line 0 (EXTI0).
   - When accessing the EXTICRx registers in your application, you will find that they are arrays defined in *stm32f0xb.h*.

```
SYSCFG->EXTICR[3] |= ... // Accesses the EXTICR4 register
```

■

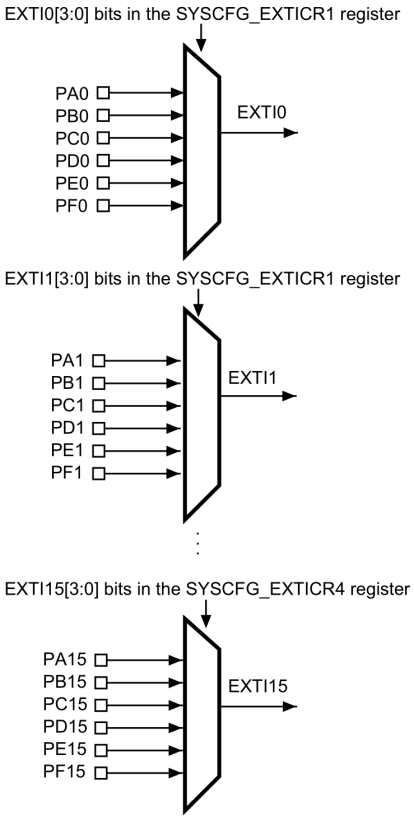



Figure 2.4: SYSCFG/EXTI Pin Multiplexers

## 2.5 Working With Interrupts

Figure 1.3 in the previous lab showed a block diagram of the peripherals within an STM32F072 device. Considering that many of these peripherals can generate interrupts, the system must recognize and manage a large number of possible sources. Some peripherals share interrupts, and a single peripheral may have multiple trigger conditions.

The large number of possible interrupt sources necessitates a way to enable, sort, and otherwise manage these sources. Because of the tight binding of the interrupts to the processor core within the ARM Cortex-M0 itself, we make use of the Nested Vectored Interrupt Controller (NVIC).

### 2.5.1 The Nested Vectored Interrupt Controller

The primary responsibilities of the NVIC are enabling and disabling interrupts, indicating requests waiting for servicing, canceling pending interrupt requests, and establishing how multiple interrupts interact through configurable priorities. Figure 2.5 shows a simplified block diagram of the NVIC, also available from *The Definitive Guide to Arm Cortex-M0 and Cortex-M0+ Processors*.

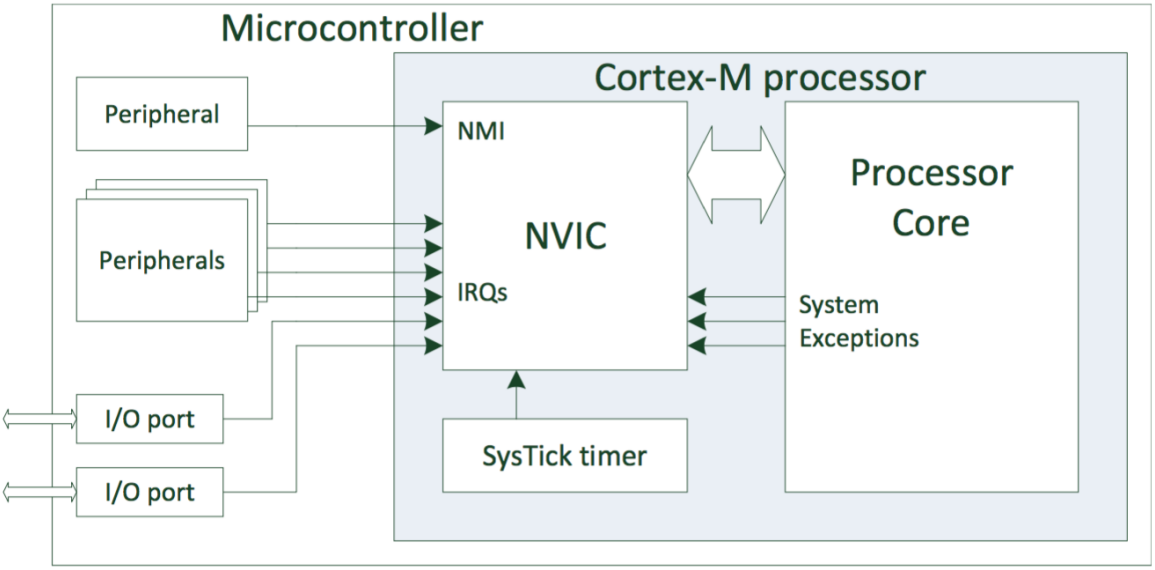



Figure 2.5: The Nested Vectored Interrupt Controller

Depending on the type of ARM core within a device, the NVIC features different capabilities. Within a Cortex M0 device such as the STM32F0, the peripheral only contains the few types of control registers. Since the NVIC is a ARM-core peripheral, its documentation exists in the ARM core and programming manual—not the STM32F0 peripheral reference manual. Similarly the structure and register definitions lie within the *core_cm0.h* file, and not in the *stm32f072xb.h* with the other peripherals.

Open the core programming manual and go to page 70 (Section 4.2) which begins the register documentation for the NVIC peripheral. A summary of the NVIC registers is as follows:

- **Interrupt set-enable register (ISER)**

  - The ISER register enables interrupts and indicates which are active. Setting a bit enables the corresponding interrupt. This register is "read and set only," meaning that it ignores attempts to clear bits.

- **Interrupt clear-enable register (ICER)**
  - The ICER register disables interrupts. This register uses a write to clear scheme: setting a bit **disables** the matching interrupt. This register is "read and write-one-clear only," meaning that it ignores attempts to clear bits (that is, enable interrupts).
- **Interrupt set-pending register (ISPR)**
  - The ISPR shows which interrupts are pending currently; you may also set a bit in order to force a manual interrupt.
- **Interrupt clear-pending register (ICPR)**
  - The ICPR shows which interrupts are pending currently, which you may manually clear; this is useful to cancel an interrupt request before the interrupt handler launches.
- **Interrupt priority registers (IPR0-IPR7)**
  - These registers configure the priorities for each interrupt.
  - Each IPR register contains four 8-bit regions to set the priority of an interrupt; the NVIC within the STM32F0 only has the uppermost two bits from these regions implemented, giving four possible configurable priority levels (0-3). More bits are available on other chipsets.

### 2.5.2 Using CMSIS Libraries to Configure the NVIC

Similar to ST Microelectronics, which publishes the HAL library for STM32F0 peripheral, ARM Ltd. provides the *cortex microcontroller software interface standard* (CMSIS) library which controls Cortex M0 peripherals.

Although the NVIC has a fairly simple register interface, safely modifying interrupts can become a complicated task. One of the main issues with directly modifying NVIC registers is that if an interrupt occurs during the process, it may corrupt or overwrite the register state.

The NVIC lies within the ARM core; its interface therefore remains consistent across multiple vendors devices. This is beneficial since the CMSIS library functions usually are available regardless of the specific chip manufacturer.

Within the exercises in these labs you have the choice of controlling the NVIC through the CMSIS library or register access. These CMSIS functions are located after the peripheral structure and register definitions in the *core_cm0.h* file.

#### Enabling and Setting Priorities for an Interrupt

Using the CMSIS library functions in *core_cm0.h* simplifies configuring the NVIC. These functions identify the interrupt to be modified by a number representing its index in the Vector table. These numbers have conveniently been given defined names in the *IRQn_Type* enumeration within the *stm32f072xb.h* file.

Because the NVIC within the STM32F0 has two configuration bits for each interrupt's priority, four software priority levels are available. The CMSIS library functions accept a numeric value in the range of [0-3] as allowed priority levels. The **lower** the numerical value given, the **higher** the precedence assigned to the interrupt by the NVIC.

Think of these values as the place in line they would stand according to importance, with 0 being VIP, and 3 being a B-list celebrity (the main program code being regular people with no celebrity status).

Negative priority values for interrupts are possible, but only for specialized priority interrupts like exception and reset interrupts; this way, they **always** override user-defined interrupts.

Remember that the highest software priority level for the NVIC is 0; the lowest is 3. The hardware priorities also follow a similar scheme with lower indexes in the Vector table having higher priority.

**2.4 — Enable and Set Priority of the EXTI Interrupt.** Unlike most peripherals, the EXTI has multiple interrupts assigned to it. Each interrupt is bound to a selection of the EXTI input lines, and you must choose the correct to match the input line desired.

1. In the *stm32f0xb.h* file, locate the *IRQn_Type* enumeration values that reference the EXTI peripheral.
   - These are defined names for the interrupt numbers used with the CMSIS NVIC control functions.
   - Each of the defined names for the EXTI interrupt numbers includes the range of input lines that can trigger it.
2. Select the entry that references the EXTI input line 0.
3. Enable the selected EXTI interrupt by passing its defined name to the `NVIC_EnableIRQ()` function. (located in *core_cm0.h*)
4. Set the priority for the interrupt to 1 (high-priority) with the `NVIC_SetPriority()` function.

■

### 2.5.3 Lab Assignment: Setting up the Interrupt Handler

Once you have enabled the interrupt within both the peripheral and NVIC, it is time to define a region of code as the appropriate handler.

The MDK:ARM toolchain includes a set of function names used for interrupt handlers; the Vector table automatically reference these names when compiling and linking. Declaring a function using one of these defined names automatically makes it into an interrupt handler.

The Vector table in *startup_stm32f072xb.s* defines the names; you must declare your interrupt handlers to accept no arguments and have no return value.

Most peripherals have a status register containing flag bits for pending interrupt requests; however, even in those without dedicated registers, most interrupts set status flags within their peripheral. These flags are necessary to generate interrupt requests. Typically you will need to clear the matching status bit manually for the interrupt condition that you are handling; otherwise, the interrupt will repeat continuously because the request never acknowledges as complete.

Always check the conditions for clearing status flags in the reference manual!
Many status registers clear by writing a one to the bit position; others are read-only and must clear through other methods. Some peripherals such as the USART automatically clear some status flags, so clearing the flag manually is unnecessary.

**2.1 — Writing the EXTI Interrupt Handler.** This is the final step in preparing an interrupt for use. One of the biggest difficulties with using interrupts is the number of steps that you must complete correctly before getting a functioning handler.

1. The file *startup_stm32f072xb.s* contains the names of interrupt handlers. Find the handler name that matches the named interrupt number you found in the previous exercise.
2. Use the handler name to declare the handler function in either *main.c* or *stm32f0xx_it.h*.
   - Although pre-generated interrupt handlers exist in *stm32f0xx_it.h*, they can be anywhere within the project.
   - Remember that interrupt handler function declarations accept no arguments and have no return value!
3. Toggle both the green and orange LEDs (PC8 & PC9) in the EXTI interrupt handler.
4. Clear the appropriate flag for input line 0 in the EXTI pending register within the handler.
   - Otherwise the handler will loop because the interrupt request never acknowledged.
   - Read the bit description of the pending flags underneath the register map: these bits require a different action to clear them.
5. Compile and load your application onto the Discovery board.
6. Pass off this portion with the TA.

If you completed all the previous exercises successfully, the red and blue LEDs should continue to blink while the green and orange LEDs toggle between each other whenever the user button is pressed.

If the green and orange LEDs do not toggle, check the configuration of the EXTI and NVIC peripherals. If the red LED stops flashing and the green and orange LEDs appear to light up consistently, you are not properly clearing the pending flag in the EXTI, and your application is stuck in the EXTI interrupt handler.

■

### 2.5.4 Interactions Between Multiple Interrupts

As the number of enabled interrupts within a system increases, the possibility of an interrupt request occurring during the handler of another becomes more likely; we therefore must have a deterministic way to deal with inter-interrupt interactions. The NVIC solves this problem with a system of both software configurable and fixed hardware interrupt priorities. Depending on these priority settings two outcomes are possible for multi-interrupt conditions. Figure 2.6 shows a graphical representation of each mode of operation.

#### Tail-Chaining

Some embedded processors have only a built-in hardware ordering between interrupts. In these systems, if multiple interrupt trigger concurrently or during a handler, they execute one after each other in succession according to the hardware priority. In this mode known as *tail-chaining*, interrupt handlers do not interrupt each other. Tail-chaining may use a simple save and restore mechanism for transitioning from the main thread, but it has the disadvantage of allowing a rapidly-triggering or long-running interrupt high on the hardware priority to “starve”, or prevent lower interrupts from executing.

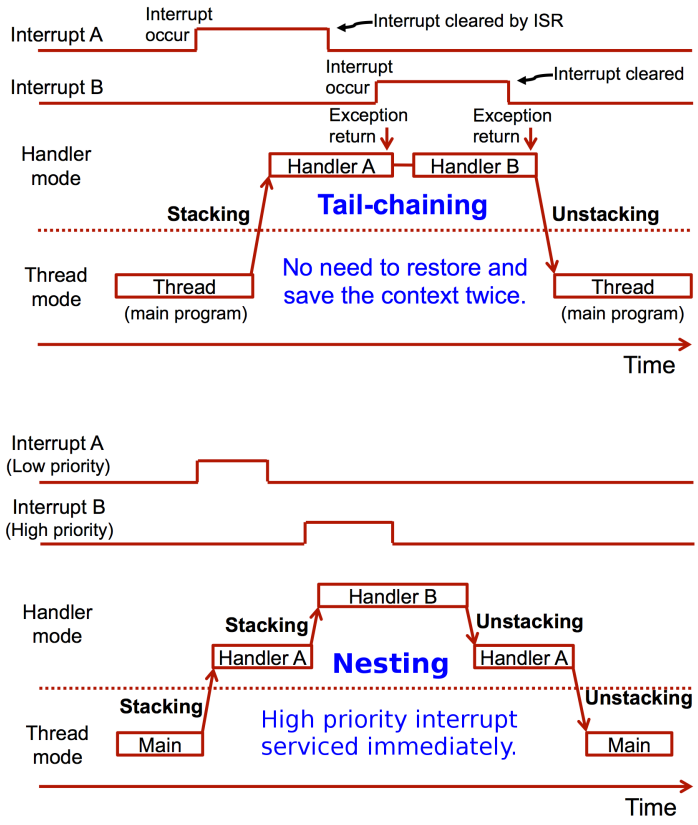



Figure 2.6: Multi-Interrupt Ordering Modes

The NVIC will tail-chain interrupts configured to the same software priority within the IPR registers. If multiple interrupts with the same software priority trigger simultaneously, the built-in hardware ordering determines the next handler to launch.

### 2.5.5 Lab Assignment: Interrupt Nesting

Unlike systems having only hardware interrupt priorities, the NVIC allows important interrupts to interrupt lower priority handlers; this process, called *nesting*, requires a more complex context-switch mechanism but otherwise works identically to how interrupts pause execution of the main application thread.

Allowing nested interrupts introduces some complications: some interrupt tasks require uninterrupted processing without losing or corrupting data (e.g., interrupts which move data between communication peripherals). Many of these have limited buffer space and will overwrite data if the interrupt execution delays or pauses for too long.

Properly establishing the priorities between interrupts resolves much of the problem; in some cases, however, it may be appropriate to *mask*. Masking temporarily disables other interrupts during critical sections of code. The NVIC has capabilities to mask specific interrupts, and larger relatives such as those in the Cortex M3 devices can mask interrupts by priority level. When the NVIC masks an interrupt, it may still enter the pending state; this allows the NVIC to evaluate and launch the appropriate handlers once the NVIC removes the mask.

**2.2 — Long-Running Interrupts.** Although in some cases it may be infeasible, normally you want to keep interrupt handlers as short as possible to avoid starving parts of your program. This exercise demonstrates how a long running interrupt impacts the main application loop.

In this exercise we shall create a long-running interrupt by adding a delay loop to the EXTI handler you wrote earlier. **Remember, adding delay to an interrupt is typically a bad idea!**

1. Add a delay loop of roughly 1-2 seconds to the EXTI interrupt handler.
   - These exercises will temporarily break the operation of the HAL delay library.
   - A count to 1,500,000 should be sufficient.
2. Add a second LED toggle so that the green and orange LEDs should exchange once before and after the delay loop.
3. Compile and load your application onto the Discovery board.
4. Pass off this section with a TA.

When you press the user button, you should see the red LED stop flashing while the EXTI interrupt hander is in the delay loop. This indicates that the main application has stopped working since the processor is stuck in the long interrupt.

Although the main application stops whenever the system is in an interrupt handler, the SysTick handler controlling the blue LED should continue to operate normally. This is due to the priority of the SysTick handler being higher than the EXTI and therefore has the ability to interrupt the long running hander. ■

**2.3 — Exploring Interrupt Priorities.** In situations where you have a mix of short and long-running interrupts, pay special attention to their priorities. This exercise demonstrates how a long running interrupt can prevent others from executing properly.

**"Starving" Interrupts**

This exercise shows how having poor priority choices can prevent the SysTick handler from operating properly.

1. The HAL library initializes the SysTick timer to have the highest priority possible (again, lowest numerical priority level). Add code to your main application that changes the SysTick interrupt priority to 2 (medium priority).
   - You will have to look up the defined name for the SysTick interrupt number and pass it to the `NVIC_SetPriority()` function.
2. Your EXTI interrupt should already have its priority set to 1 (high priority) in the NVIC.
3. Compile and load your application onto the Discovery board.

Watch the SysTick interrupt operate normally and blink the blue LED until the button is pressed and the EXTI external interrupt launches. If you set the priorities as described above, the SysTick interrupt will stop while the external interrupt operates.

What is happening in this situation? The EXTI interrupt is "starving" the SysTick interrupt. Determine why this is happening.

**Fixing with NVIC Priorities**

As shown, a long running interrupt will prevent all others of a lower or equal priority from executing until it finishes. By setting priorities properly, both interrupts can coexist.

1. Change the EXTI interrupt to have priority 3. (lowest priority)
2. Program the board and observe the LEDs, both interrupts should be working properly once more.
3. Pass off this section with a TA to show that changing the priorities changes the way the LEDs function.

In this case, it is trivial to decide on the proper priorities for the two interrupts. However, in real systems the process can become complicated. As a general rule interrupts used for timing need the highest priority, interrupts from communications peripherals need high priority, many others can be set to medium, and anything long-running should be on low.

If you must have a long-running, high-priority interrupt, verify that the others will function properly; similarly, even the lowest priority interrupts interrupt the main application thread. If your main thread performs a task, make sure you don't starve it as well.

■

## 2.6 Postlab Questions

**2.2 — Postlab 2.** Please answer the following questions and hand in as your postlab for Lab 2.

1. Why can't you use both pins PA0 and PC0 for external interrupts at the same time?

2. What software priority level gives the highest priority? What level gives the lowest?
3. How many bits does the NVIC have reserved in its priority (IPR) registers for each interrupt (including non-implemented bits)? Which bits in the group are implemented?
4. What was the latency between pushing the Discovery board button and the LED change (interrupt handler start) that you measured with the logic analyzer? Make sure to include a screenshot in the post-lab submission.
5. Why do you need to clear status flag bits in peripherals when servicing their interrupts?

■