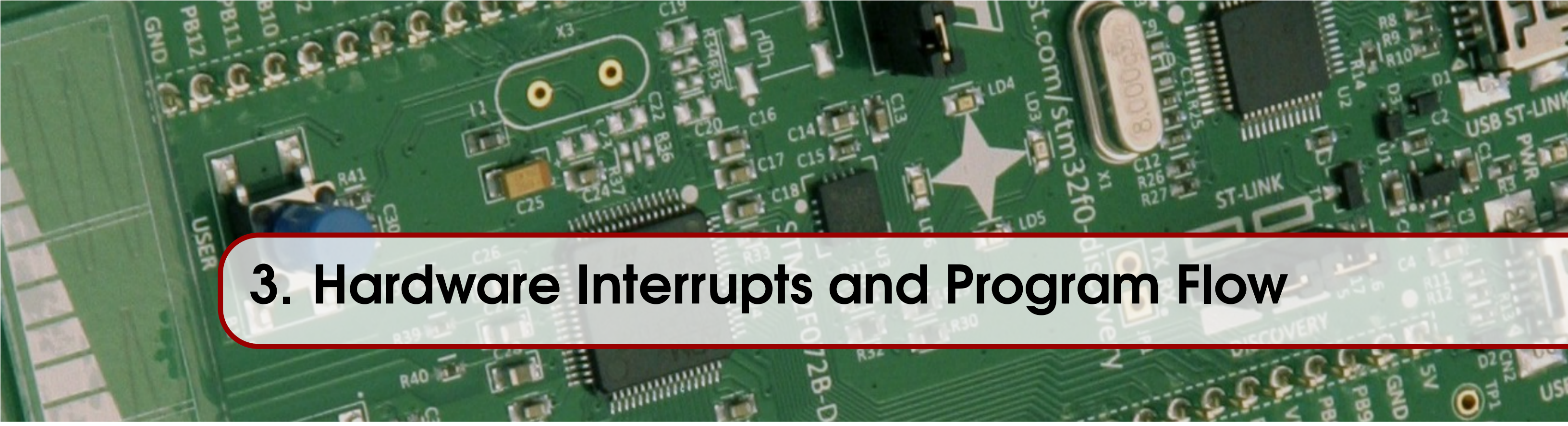


3. Hardware Interrupts and Program Flow

3.1 Overview

This lab introduces the concept of interrupt-driven programming and guides through the configuration of interrupt-oriented peripherals. The exercises in this lab 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 and each other.

3.2 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 written as 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, program concurrency is directly driven by the processor hardware. The method by which this happens are called *interrupts*.
An interrupt is the process by which the hardware temporarily suspends the execution of the main single threaded program to execute specific regions of code at known locations in memory. Interrupts are named such because these program jumps "interrupt" the main program. Figure 3.1 demonstrates the basic operation of an interrupt in a system such as the STM32F0.

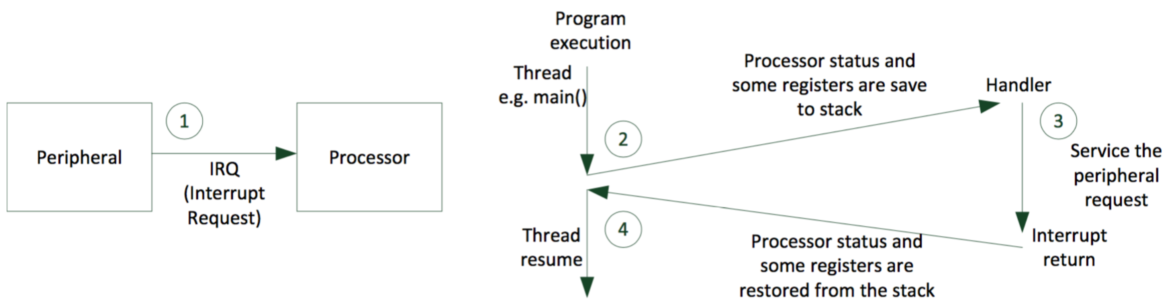

Peripheral
1
IRQ (Interrupt Request)
Processor
Program execution
Thread e.g. main()
2
Processor status and some registers are save to stack
Handler
3
Service the peripheral request
Interrupt return
4
Processor status and some registers are restored from the stack
Thread resume

Figure 3.1: Operation of an Interrupt

Interrupts are usually generated by peripherals within the embedded device. These events are used to signal a change in the peripheral state, such as receiving data from a communications interface. Other interrupts signal error conditions or are used to recover from bad processor states caused by disallowed operations within the main program. Many interrupts can be directly triggered by the user's code to perform operations with a higher priority than the main thread.

Every interrupt has a hardware number designation. An interrupt's number indicates its hardware priority and is used to index 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 RESET vector's (in actuality a RESET interrupt) location in the Vector Table is directly at the beginning. This results 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 is triggered, 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, any original state is restored, and the main program executes almost as if it was never interrupted.

Figure 3.2 (peripheral reference manual pages 217-219) shows the documentation for the STM32F072 Vector Table. The Vector Table is located within the startup assembly code for the processor. It defines human-friendly 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.

3.3 Managing System Interrupts

Figure 2.3 in the previous lab showed a block diagram of the peripherals within an STM32F072 device. Considering that many of these peripherals can generate interrupts, there are a large number of possible sources that the system must recognize and manage. Some peripherals share interrupts, and within a single peripheral there may be multiple trigger conditions.

Because of the large number of possible interrupt sources, there must be a way to enable, sort and otherwise manage them. Because interrupts are tightly bound to the operation of the actual processor core, within the ARM Cortex-M0 itself, exists the Nested Vectored Interrupt Controller (NVIC).

3.3.1 The Nested Vectored Interrupt Controller

The primary responsibilities of the NVIC are to enable and disable interrupts, indicate requests waiting to be serviced, cancel pending interrupt requests, and set how multiple interrupts interact through configurable priorities. A simplified block diagram of the NVIC is shown in figure 3.3.

Depending on the type of ARM core present 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. Because the NVIC is a ARM-core peripheral it is documented in the ARM core and programming manual not the STM32F0 peripheral reference manual. Additionally the structure and register definitions are located in the *core_cm0.h* file and not in the *stm32f072xb.h* like the other peripherals.

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 DMA2_CH1_2	DMA channel 2 and 3 interrupts DMA2 channel 1 and 2 interrupts	0x0000 0068
11	18	settable	DMA_CH4_5_6_7 DMA2_CH3_4_5	DMA channel 4, 5, 6 and 7 interrupts 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 3.2: STM32F072 Vector Table

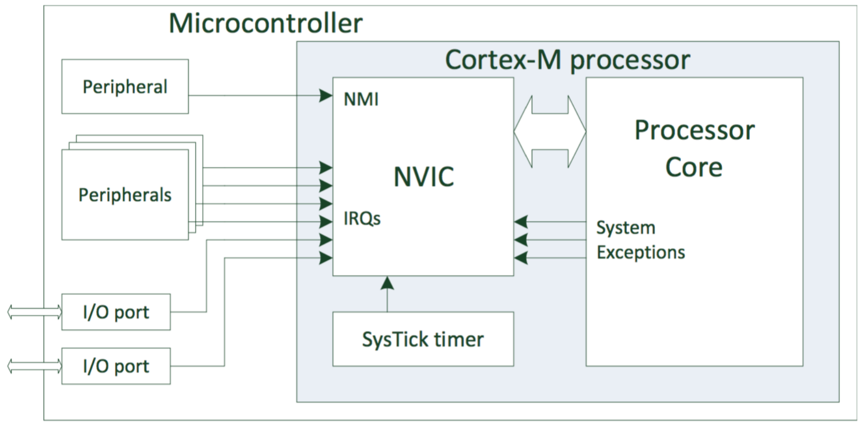
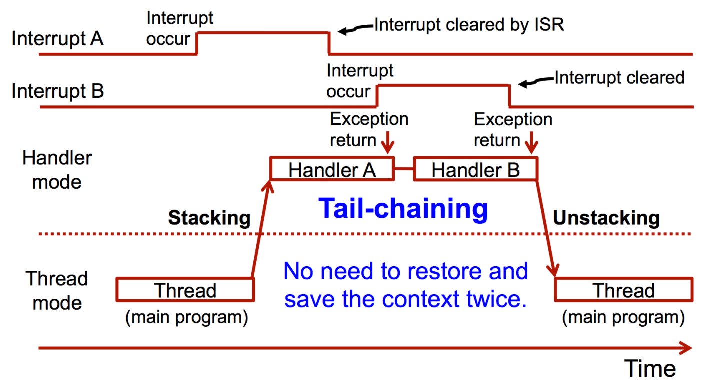


Figure 8.2
The NVIC in the Cortex®-M0 and Cortex-M0+ processors can deal with up to 32 IRQ inputs, an NMI, and a number of system exceptions.

Figure 3.3: The Nested Vectored Interrupt Controller

Open the core programming manual and go to page 71. Beginning here is 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 enabled. Writing a ‘1’ to a bit enables the matching interrupt, this register is “read and set only,” meaning that attempts to clear bits are ignored.
- **Interrupt clear-enable register (ICER)**
  - The ICER register disables interrupts. This register uses a write to clear scheme. Writing a ‘1’ to a bit disables the matching interrupt. This register is “read and write-one-clear only,” meaning that attempts to clear bits are ignored.
- **Interrupt set-pending register (ISPR)**
  - The ISPR shows which interrupts are pending, and can manually force interrupts into a pending state.
- **Interrupt clear-pending register (ICPR)**
  - The ICPR shows which interrupts are pending, and can manually clear pending status for an interrupt. This can be used to cancel an interrupt request before the interrupt handler is launched.
- **Interrupt priority registers (IPR0-IPR7)**
  - These registers configure the priorities for each interrupt.
  - Each IPR register contains four 8-bit regions dedicated to configuring the priority of a specific interrupt. The NVIC within the STM32F0s only has the uppermost two bits from these regions implemented, giving four possible configurable priority levels.



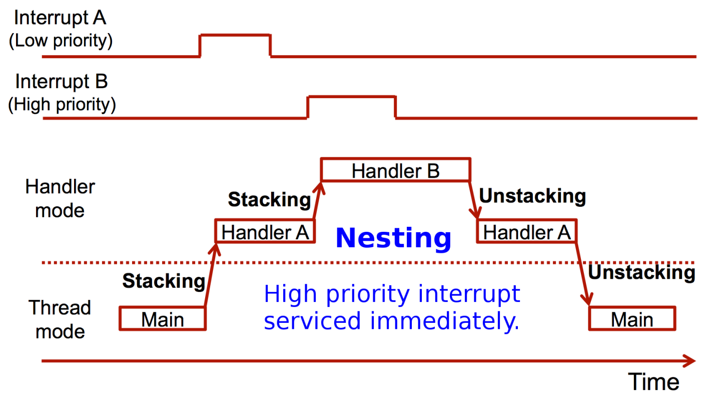


Figure 3.4: Multi-Interrupt Ordering Modes

3.3.2 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 likely. Because of this, there must be a deterministic way of dealing 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 there are two possible outcomes for multi-interrupt conditions. Figure 3.4 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 are executed one after each other according to the hardware priority. In this mode known as *tail-chaining*, interrupt handlers are never interrupted. Tail-chaining can use a simple save and restore mechanism for transitioning from the main thread but 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.
The NVIC will tail-chain interrupts configured to the same software priority within the IPR registers. If multiple interrupts with the same software priority become pending at the same time, the built-in hardware ordering will determine the next handler to launch.

3.3.3 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 can not be interrupted without losing or corrupting data. An example of this are interrupts which move data between communication peripherals. Many of these have limited buffer space and will overwrite data if the interrupt execution is delayed or paused for too long.

Much of this can be handled by setting the priorities between interrupts properly. However, in some cases, it may be appropriate to *mask* or temporarily disable 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 an interrupt is masked, it is still able to enter the pending state; this allows the NVIC to evaluate and launch the appropriate handlers once the masks has been removed.

3.3.4 Using CMSIS Libraries to Configure the NVIC

Similar to ST Microelectronics which publishes the HAL library for STM32F0 peripherals, 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 modifying interrupts can become a complicated task to do safely. One of the main issues with directly modifying NVIC registers is that if an interrupt were to occur during the process, there is a possibility of corrupting or overwriting the register state.

Because the NVIC is within the ARM core its interface remains consistent across multiple vendors devices. This is beneficial because the CMSIS library functions are usually 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.

3.4 Triggering Interrupts With External Signals

In the previous lab, we used a button press on the Discovery board to toggle between two LEDs. To do this, we repeatedly checked the button state in the infinite loop of the main application. This method of detection is called *polling*. Polling has the advantage that the repetitive and periodic checking enables tricks such as software debouncing. However it has the disadvantage of using a significant amount of processor cycles even when the device could otherwise be idle.

In some embedded systems such as the Discovery board, wasting energy on polling is not a significant challenge as continuous power is readily available. However, many battery-powered systems need to reduce the power consumption by any means possible to prolong the battery life.

One method of avoiding continuous polling is to utilize the interrupt system of the processor to monitor and detect changes in a pin's state. With the ability to do this it becomes possible to place the device into a low-power mode when no other processing is required.

The exercises in this lab will be using the "Wait for Interrupt" (WFI) assembly instruction which puts the processor into "sleep" mode. This is the least drastic of the low-power modes that the STM32F0 offers. In this state, the ARM processor is stopped, 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 shutdown additional peripherals, system oscillators, and power circuitry. These modes are more limited in the methods available to wake them up, and some of them lose device state.

3.4.1 Extended Interrupts and Events Controller

The *Extended Interrupts and Events Controller* (EXTI) is the peripheral that allows non-peripheral sources to trigger interrupts. While typically used to generate interrupts from the GPIO pins of the device, it also has the ability to monitor various internal signals such as the brownout protection circuitry. (low-voltage shutdown)
The EXTI documentation begins on page 219 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 *functional description* section on the peripheral describes the mapping between EXTI event "lines" or 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 are typically used to 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. This 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 both rising and falling triggers to be enabled for an input.
- **Software interrupt event register (EXTI_SWIER)**
  - The SWIER register allows the user to manually 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 trigger event has occurred on an input signal since the pending flag was last cleared. As long as the corresponding interrupt is enabled the EXTI interrupt handler will be repeatedly called unless the corresponding pending flags are cleared.

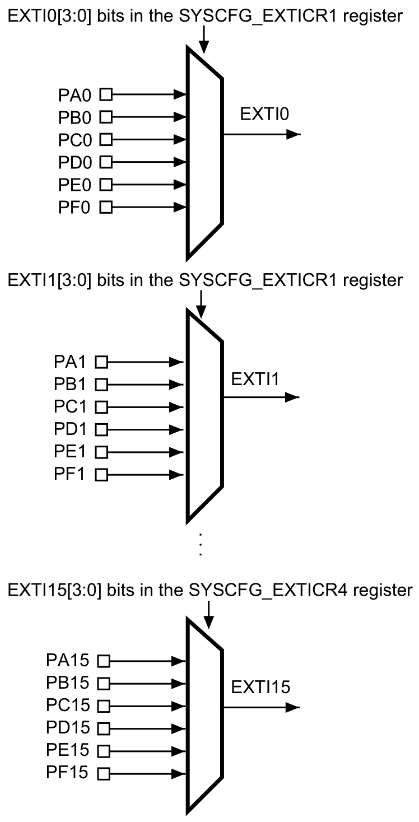

EXTI0[3:0] bits in the SYSCFG_EXTICR1 register
PA0
PB0
PC0
PD0
PE0
PF0
EXTI0
EXTI1[3:0] bits in the SYSCFG_EXTICR1 register
PA1
PB1
PC1
PD1
PE1
PF1
EXTI1
EXTI15[3:0] bits in the SYSCFG_EXTICR4 register
PA15
PB15
PC15
PD15
PE15
PF15
EXTI15

Figure 3.5: SYSCFG/EXTI Pin Multiplexers

3.4.2 Pin Multiplexing with the SYSCFG

Although the STM32F0s have the ability to generate external interrupts on almost any pin, there are only 16 available input lines to the EXTI. Because of this, there are a series of pin multiplexers selecting the pins that connect to the limited EXTI inputs.

These multiplexers are controlled by the *System Configuration Controller* (SYSCFG) peripheral. 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 3.5 shows the SYSCFG pin multiplexers used by the EXTI. These multiplexers group external pins by their orderings within the GPIO peripherals. For example, PA0, PB0 ... PF0 are grouped on a single multiplexer with the output routed to the EXTI0 input. Because only a single pin from a group can be used, pins need to chosen such that they do not conflict with each other when using multiple external interrupts.

The multiplexers are configured by the EXTICRx registers within the SYSCFG peripheral. The register maps for these begin on page 177 of the peripheral reference manual.

3.5 Working With Interrupts

The examples in this section of the lab demonstrate the process of setting up an interrupt for the *Universal-Synchronous-Asynchronous-Transmitter* (USART). You are not expected to know how the USART operates, that will be the topic for a later lab, but understand that it is a communications peripheral with has the capability of generating an interrupt after receiving a full byte of data.

3.5.1 Enabling and Setting Priorities for an Interrupt

Configuring a Peripheral to Generate Interrupts

Most peripherals have multiple conditions that can trigger an interrupt. These conditions may signal different events or error states that may occur in the operation of the peripheral. Usually all interrupt-based features within a peripheral are disabled by default, this allows the user to enable only the conditions that they wish to manage in their interrupt handler.

■ **Example 3.1 — Enabling the USART RXNE Interrupt.** In this example we'll be enabling the *receive register not empty interrupt* (RXNE) which "fires" or triggers whenever new data arrives and is waiting to be processed.
Open the peripheral reference manual to page 720 and examine table 26.7 *USART Interrupts.* This table lists all of the events that can trigger the USART interrupt. Because these events must share a single interrupt handler, they set status bits which are used to determine what event needs to be managed. The table also lists the control bits that need to be set to enable the interrupt for each specific event.
From table 26.7, we can see that we need to set the *RXNEIE* bit to enable the receive interrupt condition. This bit is located within the *Control Register 1* (CR1) of the USART peripheral. The following line of code configures USART1 to signal an interrupt request whenever data is received

```
USART1->CR1 |= USART_CR1_RXNEIE; // Enable RX interrupt in USART
```

■

Enabling the Interrupt within the NVIC

In the previous example we configured the USART to generate an interrupt request whenever new data arrives. However, unless the NVIC is also configured to allow the interrupt it will simply ignore the request.
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, there are four software priority levels available. The CMSIS library functions accept a numeric value in the range of [0-3] as allowed priority levels. The lower the priority value given, the higher the actual priority assigned to the interrupt by the NVIC.

! 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.

■ **Example 3.2 — Configuring the NVIC.** First we need to look up the appropriate interrupt number, preferably by a defined name in the *stm32f072xb.h* file. Afterwards we can pass it to the `NVIC_EnableIRQ()` function to enable the interrupt.

```
NVIC_EnableIRQ( USART1_IRQn )
```

After enabling the interrupt, we need to set the priority. Since the USART may be receiving a stream of data, we will need some buffer unless it is possible to process each byte as it arrives. Unfortunately the USART's receive register can only hold a single byte at a time, and if new data arrives before we have read the previous byte, it will be overwritten. This means that we will have to do the buffering ourselves, and depending on the speed that the USART is configured to use, we may not have time to wait around until it becomes convenient to move the data.
This probably means that we will want to give the USART a higher priority than many of the other interrupts. The following code snippet configures the USART interrupt to high priority.

```
NVIC_SetPriority(USART1_IRQn, 1 ); // Configure to high priority
```

■

3.5.2 Setting up the Interrupt Handler

Once the interrupt has been enabled 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. These are automatically referenced by the Vector table when compiling and linking. Declaring a function using one of these defined names automatically makes it into an interrupt handler.
These names are defined with the Vector table in *startup_stm32f072xb.s*. Your interrupt handlers must be declared 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 used to generate interrupt requests. Typically you will need to manually clear the matching status bit for the interrupt condition you are handling. Otherwise, the interrupt will continuously repeat because the request is never acknowledged as completed.

> Always check the conditions for clearing status flags in the reference manual!
>
> Many status registers are cleared by writing a one to the bit position. Others are read-only and must be cleared through other methods.
>
> Some peripherals such as the USART automatically clear some status flags. For example, explicitly clearing the receive interrupt flag in a USART is unnecessary since it self-clears whenever the receive register is read.

■ **Example 3.3 — Writing the USART Interrupt Handler.** In the *startup_stm32f072xb.s* file, we can see the implementation of the Vector table. This table lists a series of (mostly) unimplemented function names that are linked by the toolchain whenever the appropriate interrupt request is signaled from the NVIC.
Looking down the table, we can find the name for the USART1 handler to be "USART1_IRQHandler" We can use this name to define a function anywhere within the code project. Typically, interrupt handlers are placed either within the interrupt specific code file *stm32f0xx_it.c*, main.c, or files containing the peripheral's driver.
Figure 3.6 shows a completed interrupt handler for the USART1. It begins by checking all enabled conditions to find the one triggering the interrupt, clears the condition flag, and performs some action.

■

```c
void USART1_IRQHandler(void) {

    /* Test status flags to determine condition(s) that triggered interrupt
     * Only the "receive register not empty" event is shown in this example
     */
    if( USART1->ISR & USART_ISR_RXNE) {
        /* Clear the appropriate status flag in the USART
         * Technically this isn't necessary because the RXNE bit is
         * automatically cleared by reading the RDR register.
         */
        USART1->RQR |= USART_RQR_RXFRQ;   // Clears the RXNE status bit

        rxbuf_push( USART1->RDR );        // Save the data in the RX register
    }

    /* Additional status flag checks can follow, can operate on multiple
     * events in a single interrupt as long as the total execution
     * time of the handler is short.
     */
}
```

Figure 3.6: Example USART RXNE Interrupt Handler

3.6 Lab Assignment: Writing Interrupt-Based Code

The following exercises explore basic concepts of interrupt-driven programming, peripheral-interrupt configuration, and how priorities order multiple interrupts. After completing these tasks, make sure to show the lab assistant! Most of your points for the lab will come from demonstrating your solutions.

3.6.1 Modifying an Existing Interrupt

In this exercise, you will modify the SysTick timer interrupt to perform some user operations. In the project template that STMCube generates, the SysTick timer is already configured as part of the HAL library initialization.
The SysTick peripheral is a simple countdown timer which begins at a software set value. Once the value of the timer reaches zero, an interrupt is triggered, and the timer resets. By default, the HAL configures the SysTick interrupt to occur once every millisecond. The interrupt handler is located within the *stm32f0xx_it.c* file.

Adding Code to the SysTick Interrupt

1. Locate the SysTick interrupt handler and write a simple application which blinks or flashes LEDs in the handler.
    - Choose something reasonable, but have fun. Your code can be a single flashing LED or a complex multi-color pattern.
2. Configure used GPIO pins in the main application. Don't place run-once code such as initializations in the handler.

3. Your modified SysTick handler will be called every millisecond. Because of this periodicity, you can count the number of iterations as a timing basis.
   - For example, toggling an LED every 200th time the handler is called results in 200 ms between blinks.
   - You will need to use either a volatile global variable or local-static variable to store interrupt count.
   - **Do not use delay functions in the interrupt handler.**
4. Remove any existing code within the infinite loop of the main function.
   - We want the processor to go to sleep when not executing an interrupt handler.
   - Place a call to the `__WFI()` function in the infinite loop.
     - This function executes an assembly *wait for interrupt* (WFI) instruction, and causes the device to enter into "sleep" mode.

! **Never use any sort of delay within an interrupt handler!** Handler functions are intended to quickly perform work and then return. The HAL delay functions will deadlock if used in interrupts with the same or higher priority than the SysTick.

Changing the Interrupt Configuration

Even though the HAL library has already configured the SysTick and enabled the interrupt, we are going to override the default settings in the application.

1. Use the CMSIS `SysTick_Config()` function in your initialization code to set the SysTick interrupt rate.
   - The SysTick CMSIS functions are located with the NVIC control library.
   - The single argument to the function is the number of processor cycles the timer should count between interrupts.
     - This value is calculated by taking the processor clock frequency and dividing by the desired SysTick interrupt frequency. (Use 1 kHz interrupt frequency)
     - Since you aren't familiar with the clock system of the STM32F0, use the `HAL_RCC_GetHCLKFreq()` library function to get the value of the active clock source.
2. Use the CMSIS functions to enable and set the priority of the SysTick interrupt in the NVIC.
   - See the example in section 3.5.1.

3.6.2 Using External Interrupt Sources

Writing a simple application based around the SysTick interrupt introduced the concepts of basic interrupt-driven programming. However, modifying the existing interrupt handler of a peripheral which requires minimal configuration doesn't give an accurate representation of the full process.
In this exercise you will be enabling an external interrupt on the rising-edge of the user button (PA0) pin.

1. Configure the button pin (PA0) to input-mode, low-speed and with the internal pull-down resistor enabled.
2. Use the RCC to enable the peripheral clock to the SYSCFG peripheral.

   - Because of the low-level wakeup functions it can provide, the EXTI peripheral always is connected to the peripheral clock.

3. Determine and configure the SYSCFG multiplexer routing PA0 to the EXTI peripheral.
   - See section 3.4.2 in the lab manual.
   - The SYSCFG is documented in section 10 of the peripheral reference manual. (page 173)
     - Future labs won't provide page numbers into the documentation. You will want to get familiar with navigating the table of contents of each manual.
   - Each multiplexer indicates the input line/signal of the EXTI to which they connect.
4. Enable the external interrupt within the EXTI peripheral.
   - See sections 3.4.1 and 3.5.1 in the lab manual.
   - You will need to enable/unmask the specific input line and set a rising-edge trigger.
     - Remember that the first 16 inputs to the EXTI are used for external interrupts. For example, *EXTI3* is the 3rd input line.
   - The EXTI is documented in section 12.2 of the peripheral reference manual. (page 219, under Interrupts and Events)
5. Find the EXTI interrupt number that matches the enabled input.
   - See section 3.5.1 in the lab manual.
   - The EXTI has multiple interrupts, each of these handle a subset of its input lines.
   - The defined names in the Vector table and interrupt number definitions suggest which input lines the interrupt handles.
6. Enable and configure the interrupt's priority in the NVIC.
   - See section 3.5.1 in the lab manual.
   - Set the priority to be 1. (high-priority)
7. Define the interrupt handler function.
   - See section 3.5.2 in the lab manual.
   - Remember to clear the appropriate flag in the EXTI pending register within the handler.
     - Otherwise the handler will loop because the interrupt request was never acknowledged.
   - Place code that blinks or modifies the LEDs on the Discovery board within the handler.
     - Keep in mind that the handler will trigger depending on the conditions you set in the EXTI.

One of the biggest difficulties with interrupts is the number of steps that need to be completed correctly before getting any positive result.
Assuming that you set a rising-edge trigger in the EXTI, you should expect to see your handler called once per button press. However, there is a good chance that you see multiple rapid transitions instead. If this occurs when pressing or releasing the button, don't worry because it's not your code.
These rapid multiple triggers of the external interrupt are caused by button bounce. (see section 2.5.5 in the previous lab) The EXTI sees multiple transitions during the bouncing period and repeats the interrupt multiple times. Unfortunately, it's hard to debounce interrupts in software. Typically all interrupt lines connected to mechanical switches or buttons need hardware debouncing filters.
For this lab, just ignore the bounce.

3.6.3 Measuring Interrupt Latency

In this exercise, you will be using a debugging tool known as a logic analyzer to help estimate the latency between an interrupt request and the execution of its handler. Although it isn't possible to accurately measure the individual causes, we can measure the total delay caused by the processor flushing its pipeline, loading a new address to branch, and performing operation which indicate that we've entered the interrupt handler.

If you don't you have your own Saleae logic analyzer, you can check one out from the stockroom window in the lab. You will need your U-Card as they will want to keep it until you return the device.

About Logic Analyzers

The basic idea behind a logic analyzer is that it is a multi-channel digital capture tool, similar to how an oscilloscope is used to view analog signals. Unlike an oscilloscope, most logic analyzers don't capture and display data in real-time. However, even the inexpensive Saleae devices have at least four input channels and professional units may have hundreds.

Logic analyzers allow the user to capture sections of digital communications between devices. Typically you configure the device to start capturing data automatically when certain conditions are met. Afterward, the data is displayed and protocol decoders are run to annotate the data with information about what was captured.

Rather than duplicate a large portion of the Saleae user manual, this lab will suggest reading portions of the online documentation throughout this exercise. The entire manual is linked here: **Saleae Users Guide**

Preparing the Interrupt Handler

In order to detect when the interrupt handler begins executing, we need to output some sort of physical signal that the logic analyzer can detect. The easiest method to do this is to set a pin high. When doing this we want to make sure to use an efficient method that won't waste too much time waiting for other instructions or loading and storing values from memory.

Because of this, place the provided line of code directly at the beginning of the interrupt handler before anything else.

```
GPIOC->BSRR = 0x000003C0; // Set all LED pins high
```

Connecting and Measuring Latency

1. Familiarize yourself with the connectors on the Saleae device by reviewing the **Wire Harness & Test Clips** section of the user guide.
2. Connect one of the *GND* wires on the logic analyzer to a ground pin on the Discovery board.
   - Because the Discovery board has male pin headers, you can simply push the connectors of the analyzer wires onto the pins.
3. Connect the *black* signal wire (not a black ground wire) to pin PA0 on the discovery board. This will be our indicator that the interrupt signal was requested.
4. Connect the *brown* signal wire to any of the LED pins on the discovery board. (PC6-PC9) .
   - These are our indicator that the interrupt handler has been entered.
   - The chosen LED needs to start cleared and should not be modified by any other interrupts.
5. Review the basics of capturing data with the **Collecting Data & Device Settings** section of the user guide.

6. Set a rising-edge trigger on the *black* signal wire, and begin a capture session.
    - The analyzer should open a window indicating that it is waiting for the trigger signal.
7. Push the Discovery board button, the LEDs should appear to light instantly.
8. Once the capture has finished, review the section on **Measurements, Timing Markers, and Bookmarks**.
9. Measure the delay between the *black* signal rising-edge and the *brown* signal rising-edge.
    - This is the estimated latency between the interrupt signal and handler execution.
    - Write this value down and take a screenshot of the analyzer window, you will need them for your post-lab assignment.

3.6.4 Exploring Interrupt Priority

Normally you want to keep interrupt handlers as short as possible. However, there are some cases where this simply isn't feasible. In situations where a long-running interrupt exists, you must pay special attention to the priorities of the others in the system. This exercise demonstrates how a long running interrupt can prevent another from executing properly. By setting appropriate priorities in the NVIC, the two interrupts can be adjusted so that they coexist without interfering with each other's operation. To do this we are going to use both the SysTick and EXTI interrupts. Before you begin, you should have both interrupts configured and enabled in your main application.

Normal Interrupt Operation

This section demonstrates how interrupts of differing priorities can operate normally with each other.

1. In each interrupt handler, write code that toggles two of the board LEDs.
    - Make sure to use different LEDs in each handler to prevent conflicts.
    - Leave the priorities of the interrupt handlers the same as configured in the previous exercises.
2. Program the board and observe the LEDs.

If both interrupts are configured correctly, the LEDs modified in the SysTick interrupt should be flashing repeatedly. If the Discovery board button is pressed, you should see the LEDs controlled by the EXTI handler update. Both interrupts should seem to be coexisting without interfering with each other. Even though they have different priorities because they run for short periods of time they don't often conflict with each other. Even if they both were to trigger at the same moment, because they are short running the impact of being delayed or paused is minor.

"Starving" Interrupts

Now lets change the EXTI external interrupt to simulate a long-running interrupt handler. We are going to do this by putting an infinite loop in its code.

1. Remove the previous interrupt-style LED code in the EXTI handler.
2. Add an infinite loop to the EXTI handler.
3. Write new LED code in the infinite loop as if the interrupt hander were a normal thread.
    - The infinite loop makes the interrupt behave like an ordinary thread that never returns.
    - You'll need to introduce some delay into the loop to slow down the LED transitions until they are visible.

- The HAL delay functions will not work within the interrupt, you must use delay loops.(section 2.5.3 in previous lab)
- If you are feeling lazy you can simply turn on an LED to signal that the interrupt occurred.

4. Set the priority of the EXTI interrupt to be 1 (high priority) in the NVIC.
5. Set the priority of the SysTick interrupt to be 2 (medium priority) in the NVIC.
6. Program the board and observe the LEDs.

After programming, watch the SysTick interrupt operate normally and blink its LEDs. Now press the button and launch the EXTI external interrupt. If you set the priorities as described above the SysTick interrupt should have frozen, while the external interrupt is operating.
What is happening in this situation? The answer is that the EXTI interrupt is "starving" the SysTick one. Because the EXTI interrupt runs for an extended period of time the only way the SysTick handler can occur is if it can interrupt the EXTI handler. Unfortunately, because we changed the SysTick interrupt to have a lower priority than the EXTI, it is unable to do so.
Technically if the EXTI handler were to finish eventually and not continue in an infinite loop, the SysTick would eventually have the opportunity to execute, but very late.

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. Even though the EXTI interrupt contains an infinite loop, it is still possible to allow the SysTick to operate normally.

1. Set the priority of the SysTick interrupt to be 0 (highest priority) in the NVIC.
2. Program the board and observe the LEDs, both interrupts should be working properly.

In this case, it was trivial to decide on the proper priorities for the two interrupts. However, in a real system, this 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 you need to be very careful to determine that the others will function properly. Likewise, even the lowest priority interrupts have higher priority than the main application thread. If your main thread performs a task, make sure it isn't getting starved as well.
Even with good priority settings, too frequent or too many interrupts can starve parts of your application. In this case, the system remains busy attempting to keep up with the flood of requests. This can cause the lowest priority portions to be perpetually pushed down on the waiting list.