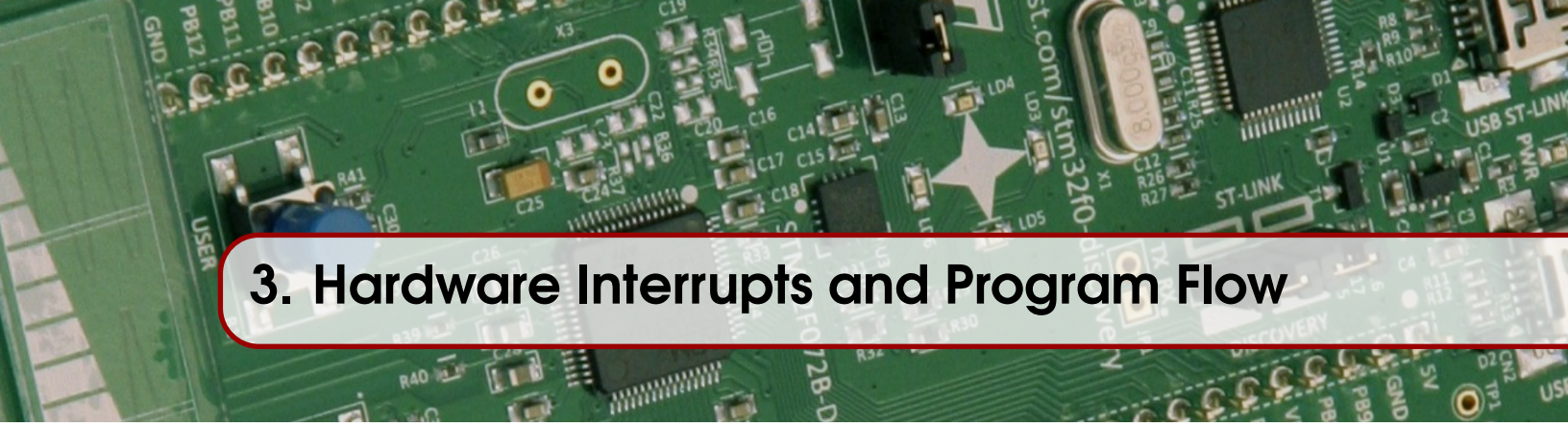

3. Hardware Interrupts and Program Flow

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.

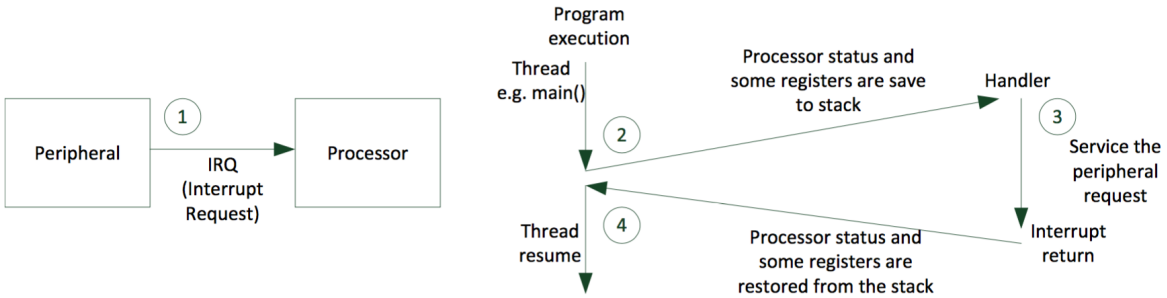

Figure 3.1: Operation of an Interrupt

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

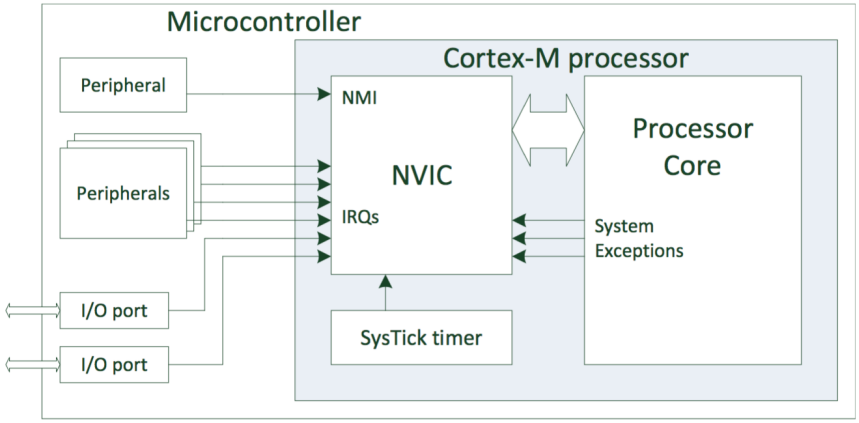

Figure 8.2
The NVIC in the Cortex®-M0 and Cortex-M0+ processors can deal with up to 32 IRQ inputs, an NMI, and a number of system exceptions.

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.

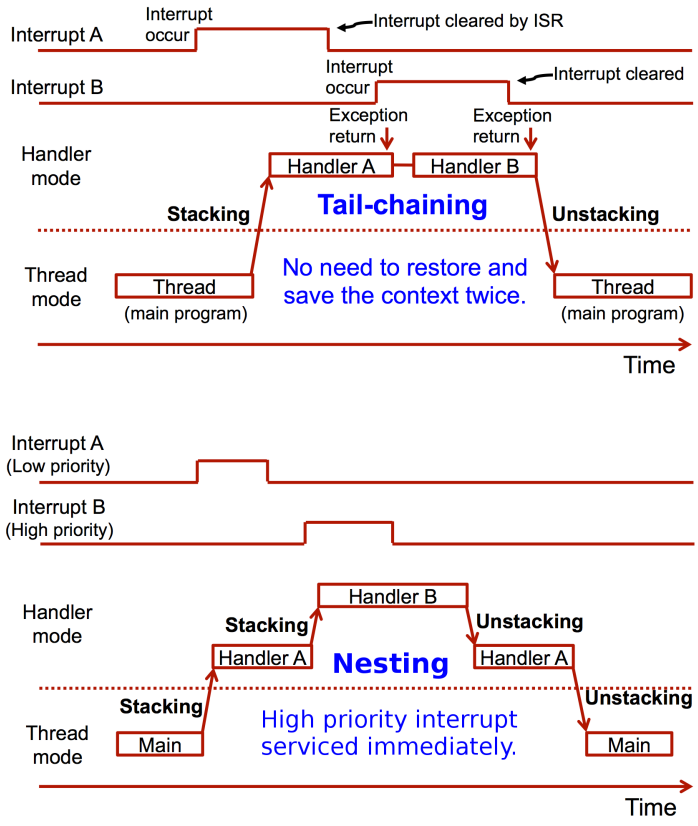

Figure 3.4: Multi-Interrupt Ordering Modes

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 priories 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.

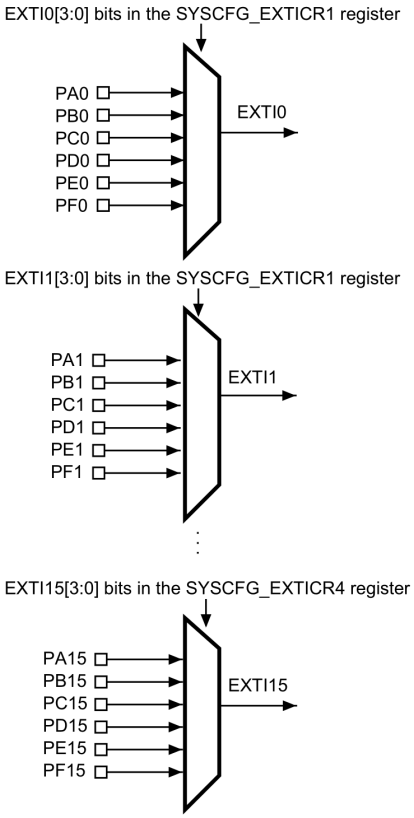

Figure 3.5: SYSCFG/EXTI Pin Multiplexers

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.

■

```
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

This manual will be updated with the lab assignment after a few issues have been resolved.