Beverly Yee

u0770041

Lab 02

Prelab 02 questions

1. What is the purpose of the NVIC peripheral?

**To handle interrupts, from enabling them to configuring priorities of the interrupts**

1. What is the difference between interrupt tail-chaining and nesting?

**Tail-chaining handles interrupts in succession based on priority while Nesting allows higher priority interrupts to be handled first, even if an interrupt is already being handled.**

1. In what file are the CMSIS libraries that control the NVIC?

**core\_cm0.h file, after the peripheral structure and register definitions.**

1. What is the purpose of the EXTI peripheral?

**Allows non-peripheral sources to trigger interrupts.**

1. What is the purpose of the SYSCFG pin multiplexers?

**Controls which pins connect to the EXTI inputs, which only has 16 input lines.**

1. What file has the defined names for interrupt numbers?

**stm32f072xb.h file, in the IRQn\_Type enumeration.**

1. What file has the Vector table implementation?

**startup\_stm32f072xb.s**

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!

1. Set the green LED (PC9) high (we will use this later in the lab).
2. 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.

Follow the blinky example from Lab01 to execute these instructions.

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.

1. 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.

The blue LED is already initialized from the previous exercise, so using the HAL toggle function will work without previous declaration.

1. 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.

Global variables are typically not a good idea. So, use a local static variable.

* **Do not use delay functions in the interrupt handler!**

A delay loop can be a for loop or while loop. While the local variable is under 200 (each count is a millisecond, so this is 200ms), increment the count. Once it hits 200, toggle PC7 and reset count.

1. 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.

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.

Can follow the blinky example from Lab01 to do this using the HAL functions. For some reason, it doesn’t work with putting all the parameters on one line, so branch it out like accessing class properties.

GPIO\_PIN\_0, GPIO\_MODE\_INPUT, GPIO\_SPEED\_FREQ\_LOW, GPIO\_PULLDOWN

1. 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.

1. Enable/unmask interrupt generation on EXTI input line 0 (EXTI0).

EXTI\_IMR\_IM0

1. Configure the EXTI input line 0 to have a rising-edge trigger.

* The peripheral reference manual documents EXTI in section 12.2 (page 219).

EXTI\_RTSR\_RT0

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.

There is nothing to show that the interrupt worked in this exercise.

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.

HAL function mirroring the other clock enables in the function.

1. 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.

The last exercise mentions that PA0 is connected to EXTI0 multiplexer. The lab manual has a figure that shows the multiplexer.

1. Each of the EXTICRx registers control multiple pin multiplexers. Find which register contains the configuration bits for the required multiplexer.

* When accessing the EXTICRx registers in your application, you will find that they are arrays defined in stm32f0xb.h.

For EXTI0, it is SYSCFG->EXTICR[0]—it accesses the EXTI0 register.

1. Configure the multiplexer to route PA0 to the EXTI input line 0 (EXTI0).

SYSCFG\_EXTICR1\_EXTI0\_PA

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.

1. Select the entry that references the EXTI input line 0.
2. Enable the selected EXTI interrupt by passing its defined name to the NVIC\_EnableIRQ() function. (located in core\_cm0.h)

EXTI0\_1\_IRQn

1. Set the priority for the interrupt to 1 (high-priority) with the NVIC\_SetPriority() function.

NVIC\_SetPriority(EXTI0\_1\_IRQn, 1);

2.5 — 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.

EXTI0\_1\_IRQHandler

1. 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!

Forward declare and start filling in the function found. Placed in main.c as to minimize the number of windows to switch between.

1. Toggle both the green and orange LEDs (PC8 & PC9) in the EXTI interrupt handler.
2. 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.

EXTI\_PR\_PR0—don’t use NVIC functions

1. Compile and load your application onto the Discovery board.

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.

2.6 — 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.

Use another while loop and a count variable.

1. Add a second LED toggle so that the green and orange LEDs should exchange once before and after the delay loop.
2. Compile and load your application onto the Discovery board.
3. Use the logic analyzer to find the time delay from the button press to the start of the interrupt routine and the LEDs changing.

* We will be using the Digilent Analog Discovery 2 logic analyzer and Waveforms software; you can find the User Guide here. If you haven’t purchased a Digilent Analog Discovery 2, there are several available for checkout from the stockroom.

1. Take a screenshot of the result from your logic analyzer.

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.

2.7 — 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.

NVIC\_SetPriority(SysTick\_IRQn, 2);

Remember that these functions ask for IRQn as an argument

1. Your EXTI interrupt should already have its priority set to 1 (high priority) in the NVIC.
2. 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.

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.

(part one code)

/\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @file : main.c

\* @brief : Main program body

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* @attention

\*

\* Copyright (c) 2024 STMicroelectronics.

\* All rights reserved.

\*

\* This software is licensed under terms that can be found in the LICENSE file

\* in the root directory of this software component.

\* If no LICENSE file comes with this software, it is provided AS-IS.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

/\* Includes ------------------------------------------------------------------\*/

#include "main.h"

/\* Private function prototypes -----------------------------------------------\*/

void SystemClock\_Config(void);

void SetPins();

/\*\*

\* @brief The application entry point.

\* @retval int

\*/

int main(void)

{

HAL\_Init(); // Reset of all peripherals, init the Flash and Systick

SystemClock\_Config(); //Configure the system clock

SetPins();

// set green LED (PC9) to high

GPIOC->ODR |= GPIO\_ODR\_6;

GPIOC->ODR &= ~(GPIO\_ODR\_7);

GPIOC->ODR &= ~(GPIO\_ODR\_8);

GPIOC->ODR |= GPIO\_ODR\_9;

// now work with EXTI0, interrupt connected to PA0

// enable/unmask = EXTI->IMR

EXTI->IMR |= EXTI\_IMR\_IM0;

// rising edge trigger = EXTI\_RTSR

EXTI->RTSR |= EXTI\_RTSR\_RT0;

// use RCC to enable the peripheral for the system clock

RCC->APB2ENR |= RCC\_APB2ENR\_SYSCFGEN;

// find the multiplexer that connects PA0 to EXTI0

// EXTICR[0] accesses the EXTICR0 register

SYSCFG->EXTICR[0] |= SYSCFG\_EXTICR1\_EXTI0\_PA;

// find the IRQn\_Type for EXTI0 = EXTI0\_1\_IRQn

// enable it

NVIC\_EnableIRQ(EXTI0\_1\_IRQn);

// set interrupt priority (intially set to 1)

NVIC\_SetPriority(EXTI0\_1\_IRQn, 1);

NVIC\_SetPriority(SysTick\_IRQn, 2);

while (1)

{

HAL\_Delay(400); // delay in ms

// toggle red LED

GPIOC->ODR ^= GPIO\_ODR\_6;

}

}

/\*

\* Interrupt handler for EXTI0

\*/

void EXTI0\_1\_IRQHandler(void){

// toggle green and orange LEDs when the interrupt is triggered

GPIOC->ODR ^= GPIO\_ODR\_8;

GPIOC->ODR ^= GPIO\_ODR\_9;

// clear the flag for EXTI0

EXTI->PR = EXTI\_PR\_PR0;

}

/\*\*

\* Set the pins according to the lab instructions

\* Helper method so the main method doesn't look overcrowded

\*/

void SetPins(){

// enable the GPIOC peripheral clock

RCC->AHBENR |= RCC\_AHBENR\_GPIOCEN;

// enable the GPIOA clock for the user button

RCC->AHBENR |= RCC\_AHBENR\_GPIOAEN;

// set the general purpose output for the LEDs

// bits = 01

GPIOC->MODER |= GPIO\_MODER\_MODER6\_0;

GPIOC->MODER |= GPIO\_MODER\_MODER7\_0;

GPIOC->MODER |= GPIO\_MODER\_MODER8\_0;

GPIOC->MODER |= GPIO\_MODER\_MODER9\_0;

// LEDs have push-pull output type = both bits cleared

GPIOC->OTYPER &= ~(GPIO\_OTYPER\_OT\_6);

GPIOC->OTYPER &= ~(GPIO\_OTYPER\_OT\_7);

GPIOC->OTYPER &= ~(GPIO\_OTYPER\_OT\_8);

GPIOC->OTYPER &= ~(GPIO\_OTYPER\_OT\_9);

// low speed = both bits cleared

GPIOC->OSPEEDR &= ~(GPIO\_OSPEEDER\_OSPEEDR6);

GPIOC->OSPEEDR &= ~(GPIO\_OSPEEDER\_OSPEEDR7);

GPIOC->OSPEEDR &= ~(GPIO\_OSPEEDER\_OSPEEDR8);

GPIOC->OSPEEDR &= ~(GPIO\_OSPEEDER\_OSPEEDR9);

// no pull-up/down resistors = both bits cleared

GPIOC->PUPDR &= ~(GPIO\_PUPDR\_PUPDR6);

GPIOC->PUPDR &= ~(GPIO\_PUPDR\_PUPDR7);

GPIOC->PUPDR &= ~(GPIO\_PUPDR\_PUPDR8);

GPIOC->PUPDR &= ~(GPIO\_PUPDR\_PUPDR9);

// configure the user button, input-mode @ low speed and pull down resistor

GPIOA->MODER &= ~(GPIO\_MODER\_MODER0);

GPIOA->OSPEEDR &= ~(GPIO\_OSPEEDER\_OSPEEDR0);

GPIOA->PUPDR |= GPIO\_PUPDR\_PUPDR0\_1;

}

/\*\*

\* @brief System Clock Configuration

\* @retval None

\*/

void SystemClock\_Config(void)

{

RCC\_OscInitTypeDef RCC\_OscInitStruct = {0};

RCC\_ClkInitTypeDef RCC\_ClkInitStruct = {0};

/\*\* Initializes the RCC Oscillators according to the specified parameters

\* in the RCC\_OscInitTypeDef structure.

\*/

RCC\_OscInitStruct.OscillatorType = RCC\_OSCILLATORTYPE\_HSI;

RCC\_OscInitStruct.HSIState = RCC\_HSI\_ON;

RCC\_OscInitStruct.HSICalibrationValue = RCC\_HSICALIBRATION\_DEFAULT;

RCC\_OscInitStruct.PLL.PLLState = RCC\_PLL\_NONE;

if (HAL\_RCC\_OscConfig(&RCC\_OscInitStruct) != HAL\_OK)

{

Error\_Handler();

}

/\*\* Initializes the CPU, AHB and APB buses clocks

\*/

RCC\_ClkInitStruct.ClockType = RCC\_CLOCKTYPE\_HCLK|RCC\_CLOCKTYPE\_SYSCLK

|RCC\_CLOCKTYPE\_PCLK1;

RCC\_ClkInitStruct.SYSCLKSource = RCC\_SYSCLKSOURCE\_HSI;

RCC\_ClkInitStruct.AHBCLKDivider = RCC\_SYSCLK\_DIV1;

RCC\_ClkInitStruct.APB1CLKDivider = RCC\_HCLK\_DIV1;

if (HAL\_RCC\_ClockConfig(&RCC\_ClkInitStruct, FLASH\_LATENCY\_0) != HAL\_OK)

{

Error\_Handler();

}

}

/\* USER CODE BEGIN 4 \*/

/\* USER CODE END 4 \*/

/\*\*

\* @brief This function is executed in case of error occurrence.

\* @retval None

\*/

void Error\_Handler(void)

{

/\* USER CODE BEGIN Error\_Handler\_Debug \*/

/\* User can add his own implementation to report the HAL error return state \*/

\_\_disable\_irq();

while (1)

{

}

/\* USER CODE END Error\_Handler\_Debug \*/

}

#ifdef USE\_FULL\_ASSERT

/\*\*

\* @brief Reports the name of the source file and the source line number

\* where the assert\_param error has occurred.

\* @param file: pointer to the source file name

\* @param line: assert\_param error line source number

\* @retval None

\*/

void assert\_failed(uint8\_t \*file, uint32\_t line)

{

/\* USER CODE BEGIN 6 \*/

/\* User can add his own implementation to report the file name and line number,

ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) \*/

/\* USER CODE END 6 \*/

}

#endif /\* USE\_FULL\_ASSERT \*/

Postlab 02

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?

**They use the same multiplexer. A multiplexer allows the programmer to control which array of inputs it receives can be used as an output. If the 2 inputs are using the same multiplexer, only one of the 2 can be used.**

1. What software priority level gives the highest priority? What level gives the lowest?

**Lowest = higher priority. 0 is the highest priority. 3 is the lowest.**

1. 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 8-bit regions, 32-bits total.**

**The NVIC uses the uppermost 2 bits to set priority.**

1. 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.
2. Why do you need to clear status flag bits in peripherals when servicing their interrupts?

**So they don’t get stuck in the interrupt and allow the application to continue running.**