

# 마이크로프로세서응용실험 LAB07 결과보고서

## Interrupts and exceptions

20181536 엄석훈

### 1. 목적

- 인터럽트의 동작원리를 전체적으로 이해한다.
- 발생한 인터럽트를 처리하는 과정에서 프로세서가 stack을 어떻게 사용하는지 이해한다.
- Vector table의 구성과 역할에 대해 이해한다.
- 외부 소자로부터 입력되는 신호에 의한 인터럽트의 처리과정에서 사용되는 레지스터들의 종류 및 역할을 이해한다.

### 2. 이론

#### 1) Interrupt types

먼저 인터럽트란 지금까지의 프로그램을 수행하는 도중 외부의 입력이나 특정한 조건에 따라 이벤트가 발생하면 프로그램이 해당 이벤트에 반응하는 동작을 수행하게 되는데 이를 exception과 interrupt라고 부른다. Exception은 일반적으로 예상치 못한 오류가 발생하면 발생하게 되고 interrupt는 외부 장치나 프로그래머의 의도에 따라서 동작하는 것을 말한다. 이를 묶어서 간단히 인터럽트라고 부른다면 프로세스가 명령어를 수행하던 도중 인터럽트가 발생하면 해당 인터럽트가 발생하였을 때의 정해진 동작을 하러 subroutine처럼 인터럽트 서비스 루틴을 수행하러 프로그램의 흐름이 바뀌게 된다. 하지만 보통 인터럽트는 언제 발생할지 예측하기 힘들기 때문에 다양한 방법을 이용해서 관리한다. 이에 대한 설명은 아래 이론부분과 실험에서 살펴본다. 그리고 인터럽트에는 다양한 종류가 존재한다. 또한 프로세스마다 다른 인터럽트가 존재한다. 우리가 실험에서 사용하는 cortex-m3 프로세서에서는 15가지의 exception과 그 외에 많은 수의 interrupt가 존재한다. 이에 대한 목록은 아래 2) vector tables에서 확인 가능하다.

## 2) Vector tables

인터럽트가 발생하게 되면 해당 인터럽트 서비스 루틴을 수행하러 프로그램의 흐름이 변경되어야 하는데 당연히 PC값이 해당 서비스 루틴이 저장되어 있는 곳으로 바뀌면 된다. 이를 위해서 인터럽트별로 서비스 루틴의 시작 주소를 저장하는 vector table이 존재한다. 이는 branch에서 사용하였던 jump table과 유사하다. Vector table은 기본적으로 메모리의 0번지부터 저장되어 있으며 인터럽트 넘버가 증가할수록 4byte씩 주소가 증가하며 모든 인터럽트 서비스 루틴의 시작 주소가 저장되어 있다. 하지만 필요에 따라서는 vector table의 위치 또한 변경할 수 있다. 아래 [그림 1]은 reference manual에서 가져온 STM32F10xxx 시리즈의 vector table의 일부 예시이다.

Table 63. Vector table for other STM32F10xxx devices

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
-	0	settable	MemManage	Memory management	0x0000_0010
-	1	settable	BusFault	Prefetch fault, memory access fault	0x0000_0014
-	2	settable	UsageFault	Undefined instruction or illegal state	0x0000_0018
-	-	-	-	Reserved	0x0000_001C - 0x0000_002B
-	3	settable	SVCall	System service call via SWI instruction	0x0000_002C
-	4	settable	Debug Monitor	Debug Monitor	0x0000_0030
-	-	-	-	Reserved	0x0000_0034
-	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	PVD through EXTI Line detection interrupt	0x0000_0044
2	9	settable	TAMPER	Tamper interrupt	0x0000_0048
3	10	settable	RTC	RTC global interrupt	0x0000_004C
4	11	settable	FLASH	Flash global interrupt	0x0000_0050
5	12	settable	RCC	RCC global interrupt	0x0000_0054
6	13	settable	EXTI0	EXTI Line0 interrupt	0x0000_0058

그림 1 STM32F10xxx Vector table

### 3) Interrupt inputs and pending behavior

인터럽트가 발생하면 가장 먼저 대기 상태인 pending상태가 된다. 그리고 인터럽트 핸들러가 수행되면 해당 인터럽트는 active상태가 되되 pending상태가 해제된다. 이때 만약 인터럽트가 계속 걸려 있다면 핸들러가 종료되고 다시 바로 pending되고 다시 active상태로 들어가게 된다. 또한 만약 핸들러 수행 중 인터럽트가 발생해도 다시 pending되게 되고 핸들러가 종료되고 다시 핸들러가 수행되게 된다. 그리고 pending은 각 인터럽트 별로 한 번만 걸릴 수 있다. 따라서 이미 pending상태인 인터럽트가 다시 발생해도 인터럽트 핸들러는 한 번만 수행된다.

### 4) Interrupt sequence

인터럽트는 언제 발생할지 예측하기 매우 힘들기 때문에 프로그램의 어떤 영역에서든 인터럽트 핸들러가 실행될 수 있다. 따라서 인터럽트가 발생하면 수행되는 정해진 루틴이 있는데 다음과 같다. 먼저 r0-r3, r12, lr, pc, psr 레지스터의 내용이 stack에 push된다. 이는 인터럽트 핸들러가 들어가기 수행되기 전에 이전 프로그램의 흐름을 유지하기 위해서 레지스터의 데이터를 stack에 저장하는 것이다. 그리고 다음으로 인터럽트 서비스 루틴의 시작주소를 vector table에서 가져온다. 마지막으로 sp는 스택에 레지스터를 저장한 후의 값으로 갱신되고, lr은 인터럽트 종료 후 어떤 방식으로 원래의 흐름으로 돌아갈지 저장하는 값을 저장한다. 예시는 아래 [그림 2]와 같다. 마지막으로 pc값을 인터럽트 서비스 루틴의 시작 주소로 업데이트한다.

마찬가지로 인터럽트 핸들러가 끝나고 원래의 프로그램 흐름으로 복귀할 때는 lr에 저장된 값을 pc로 가져와서 어떤 방식으로 원래의 프로그램 흐름으로 돌아갈지 파악하고 복귀 절차를 수행한다. 원래 stack에 넣어두었던 레지스터의 값을 원래 레지스터로 복구시키고 NVIC 레지스터에 저장된 인터럽트 active bit를 해제한다.

	Floating Point Unit was used before Interrupt (FPCA = 1)	Floating Point Unit was not used before Interrupt (FPCA = 0)
Return to Handler mode (always use Main Stack)	0xFFFFFE1	0xFFFFF1
Return to Thread mode and use the Main Stack for return	0xFFFFFE9	0xFFFFF9
Return to Thread mode and use the Process Stack for return	0xFFFFFED	0xFFFFFD

그림 2 EXC\_RETURN 값

## 5) NVIC and interrupt control

NVIC는 cortex-m3 프로세서에 내장되어 있는 인터럽트 동작을 도와주기 위한 레지스터이다.

NVIC는 다양한 종류가 있는데 다음과 같다.

NVIC\_ISERx : 해당 인터럽트를 사용할지 정하는 레지스터이다. 인터럽트 넘버 16번부터 순서대로 bit를 배정받아 있으며 해당 bit에 1을 set하면 해당 인터럽트를 사용한다는 뜻이다.

NVIC\_ICERx : 해당 인터럽트를 사용하지 않을 지 정하는 레지스터이다. 위와 마찬가지로 순서대로 bit를 배정받아 있으며 해당 bit에 1을 set하면 해당 인터럽트를 사용하지 않고 무시한다는 뜻이다.

NVIC\_ISPRx : 인터럽트의 pending상태를 기록하는 레지스터이다. 인터럽트 별로 순서대로 bit를 배정받아 있으며 해당 bit에 1을 쓰면 해당 인터럽트가 pending상태가 된다.

NVIC\_ICPRx : 인터럽트의 pending상태를 해제하는 레지스터이다. 인터럽트 별로 순서대로 bit를 배정받아 있으며 해당 bit에 1을 쓰면 해당 인터럽트가 pending상태가 해제된다.

NVIC\_IABRx : 인터럽트의 active상태를 기록하는 레지스터이다. 인터럽트 별로 순서대로 bit를 배정받아 있으며 해당 bit에 1이면 해당 인터럽트가 active상태라는 뜻이다.

NVIC\_IPRx : 인터럽트의 우선순위를 정하는 레지스터이다. 수행되는 중에도 새치기가 가능한 preempt 우선순위와 동시에 들어왔을 때의 우선순위를 지정하는 subpriority로 나뉘어서 사용되며 우리가 사용하는 프로세서는 4bit만 사용해서 priority를 지정한다.

## 3. 실험 과정

### 1) 실험 1

**STEP 1: Program 7.1을 이용해 프로젝트를 생성한다.**

```

1  RCC_APB2ENR equ 0x40021018
2  GPIOD_CRL equ 0x40011400
3  GPIDO_ODR equ 0x4001140C
4  EXTI_PR equ 0x40010414
5  EXTI_IMR equ 0x40010400
6  EXTI_RTISR equ 0x40010408
7  EXIT_FISR equ 0x4001040C
8  AFIO_EXTICR4 equ 0x40010014
9  NVIC_ISER0 equ 0xE000E100
10 NVIC_ISER1 equ 0xE000E104
11     export EXTI15_10_IRQHandler
12     area lab7_1, code
13     entry
14 __main proc
15     export __main [weak]
16 ; GPIO initialization
17     ldr r0,=RCC_APB2ENR
18     ldr r1,=0x00000021 ;IOPDEN, AFIOEN
19     str r1,[r0]
20     ldr r0,=GPIOD_CRL
21     ldr r1,=0x33333333
22     str r1,[r0]
23 ; AFIO
24     ldr r0,=AFIO_EXTICR4
25     ldr r1,=0x00000020
26     str r1,[r0]
27 ; EXTI initialization
28     ldr r1,=0x00002000
29     ldr r0,=EXIT_FISR
30     str r1,[r0]
31     ldr r0,=EXTI_IMR
32     str r1,[r0]
33 ; NVIC_ISER
34     ldr r0,=NVIC_ISER1
35     ldr r1,=0x00000100
36     str r1,[r0]
37 ;
38     ldr r0,=GPIDO_ODR
39     mov r5,#2_01001010
40     ldr r3,=sdata
41     str r5,[r3]
42 loop    ldr r4,[r3]
43         str r4,[r0]
44         b loop
45     endp
46 ;
47 EXTI15_10_IRQHandler proc
48     push {r4,r5}
49     ldr r4,=sdata
50     ldr r5,[r4]
51     add r5,#0x1
52     str r5,[r4]
53     ldr r4,=EXTI_PR
54     ldr r5,=0x00002000
55     str r5,[r4]
56     pop {r4,r5}
57     bx lr
58     endp
59     align
60     area lab7, data
61 sdata dcd 0x00
62     end

```

그림 3 직접 작성한 lab7\_1.s 코드

STEP 2: 본 과정은 다음 표와 같이 스위치와 LED들이 프로세서에 연결되었다고 가정한다. 즉, LED들은 GPIO의 PD0-PD7에 연결되며 스위치 USER\_BUTTON 신호는 GPIO의 PC13에 연결된다.

소자	소자신호	포트신호
LED	LED0	PD0
	LED1	PD1
	LED2	PD2
	LED3	PD3
	LED4	PD4
	LED5	PD5
	LED6	PD6
	LED7	PD7
스위치	USER_BUTTON	PC13
	WAKEUP	PA0

그림 4 실험 교재의 핀 연결

STEP 3:  $\mu$ Vision의 Peripherals 메뉴에서 General Purpose I/O를 선택한 후 GPIOC와 GPIOD를 각각 활성화한다. 그림 7.15는 활성화된 각 GPIO의 창들을 보여준다.

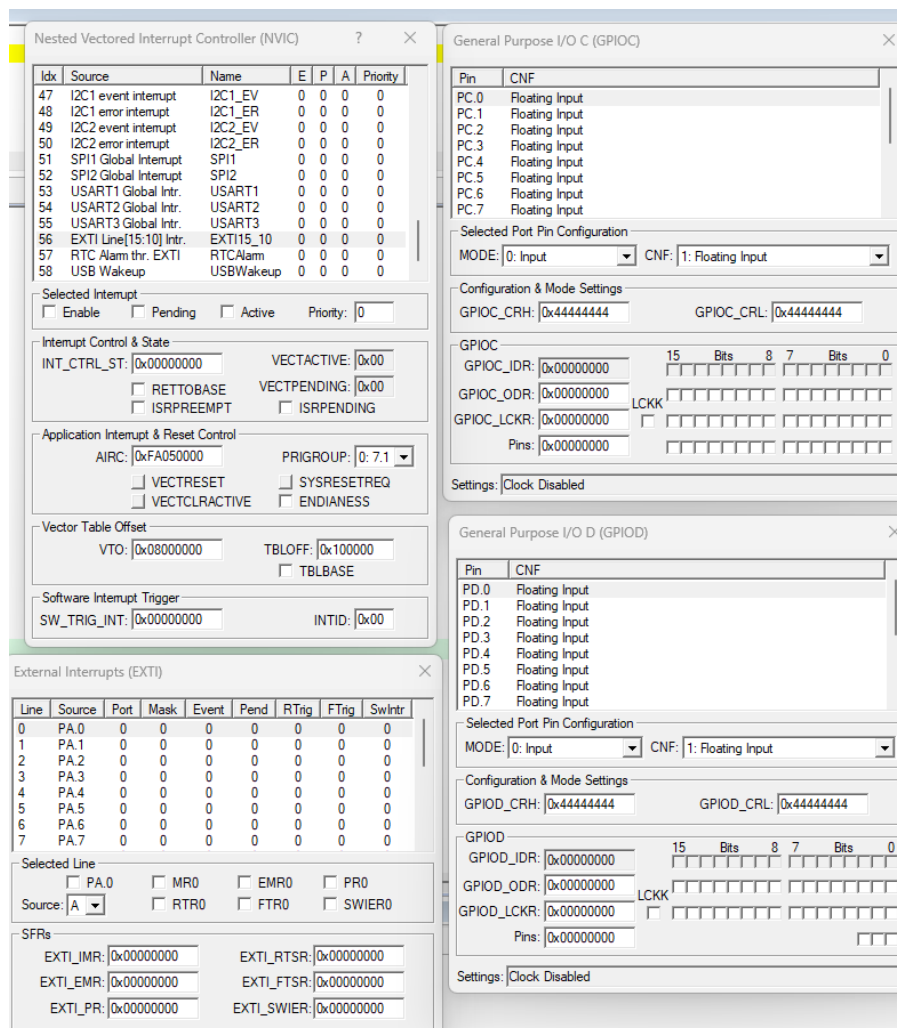


그림 5 실험을 위한 각종 창

**STEP 4: 프로그램을 break point의 설정없이 연속 수행한다.**

프로그램을 break point 없어 연속 수행하면 아래 [그림 6]의 line 42-44의 명령어만 계속해서 루프를 돌면서 r3의 주소에서 데이터를 가져와 r4에 저장하고 가져온 r4의 데이터를 r0의 주소에 저장하는 동작을 반복한다.

```

42: loop   ldr r4,[r3]
0x080002FE 681C      LDR      r4,[r3,#0x00]
43:        str r4,[r0]
0x08000300 6004      STR      r4,[r0,#0x00]
44:        b loop
45:        endp
46: ;
47: EXTI15_10_IRQHandler proc
0x08000302 E7FC      B        0x080002FE

```

그림 6 프로그램 무한 루프 수행

**STEP 5: STEP 2의 표에서 USER\_BUTTON 스위치가 연결되었다고 가정한 그림 7.15의 GPIOC 창의 13번 pin에 해당하는 사각형을 마우스로 click하기를 반복하면서 LED가 연결된 GPIOD 창의 변화를 살펴본다. 해당 GPIOC 핀에 연결된 인터럽트는 falling edge에서 발생하도록 설정하였다. 이 과정을 통해 관찰한 내용을 다음 단계들을 통해 구체적으로 이해해보자.**

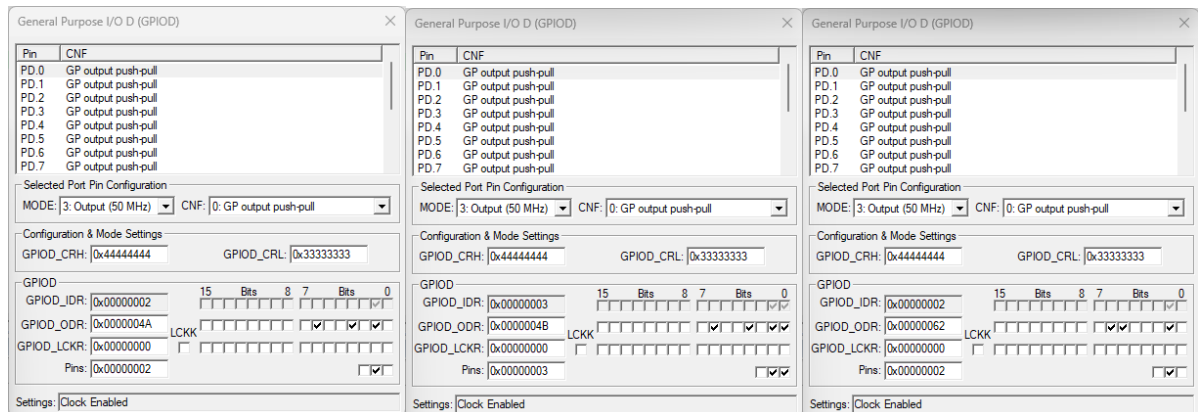


그림 7 13 pin 클릭 후 GPIOC 창 (좌측: 0회 클릭, 중간: 2회 클릭(set, reset), 우측: 수십번 클릭) GPIOC 창에서 13번 pin에 해당하는 사각형을 클릭하여 체크하였다가 풀면 인터럽트가 걸려서 인터럽트 핸들러로 들어가고 GPIOD에 저장된 값이 증가된다. 한 번 핸들러로 들어갈 때 마다 LED가 연결된 GPIOD\_ODR에 저장된 데이터가 1씩 증가하고 따라서 LED가 카운터가 증가하는 것처럼 하나씩 증가하며 켜졌다 꺼졌다 동작하는 것을 확인할 수 있다.

**STEP 6:** Line 22까지는 LED와 스위치가 연결되었다고 가정한 GPIO의 port D를 초기화하는 과정이다. 이와 관련해서는 8장에서 자세하게 설명한다.

Lines 24-26에서 수행되는 일의 의미를 7.2.6절의 설명과 자료를 이용해서 확인한다.

마찬가지로 lines 28-32에서 수행되는 일의 의미를 확인한다. USER\_BUTTON 스위치는 신호는 평소에 high이지만 누르면 low가 된다고 가정하고 falling edge trigger를 선택한다.

Line 24에서 AFIO\_EXTICR4는 GPIO의 5개의 포트 중에서 어떤 포트를 연결할 것인지에 대한 설정을 나타내는 부분으로 CR4는 EXTI12~15까지를 관리하는데 여기에 line 25-26을 통해서 0x00000020을 저장하였으므로 EXTI 13번에 0x2를 저장하였고 이는 PORT C를 연결한 것이다. 따라서 EXTI13은 PORT C가 연결되었다.

마찬가지로 line 28-32에서는 EXTI\_FTSR과 EXTI\_IMR에 0x00002000을 저장하였는데 이는 EXTI 13번을 falling edge trigger로 신호가 들어오면 인터럽트 요청이 pending해주는 동작이 가능하도록 설정을 해준 것이다.

따라서 line 24-26, line 28-32를 통해서 EXTI 13번에 PORT C를 연결해서 falling edge로 신호가 들어오면 인터럽트가 발생할 수 있도록 설정해주었다.

**STEP 7:** Lines 34-36에서 수행되는 일의 의미를 7.2.5절의 설명과 자료를 이용해서 확인한다.

Line 34-36을 통해서 NVIC\_ISER1에 0x00000100을 저장하였는데 NVIC\_ISER은 특정 인터럽트를 사용하고자 할 때 1로 bit를 set해서 해당 인터럽트를 사용할 수 있게 하는 역할을 한다. NVIC\_ISER1은 48번 인터럽트부터 79번 인터럽트까지 관리하는데 여기에 0x00000100을 저장하였으므로 56번 인터럽트를 사용하고자 한 것이고 이는 EXTI15\_10 인터럽트이다. 따라서 line 34-36 명령어를 통해서 EXTI15\_10 인터럽트를 사용하도록 설정해주었다.

**STEP 8:** µVision의 Peripherals 메뉴에서 Core peripherals – NVIC을 선택한다. 또한 Peripherals 메뉴에서 External interrupts를 선택하여 두 개의 확인창을 활성화 한다. 프로그램을 line 36까지 단계적으로 수행하면서 이 창들의 내용이 어떻게 변화하는지 관심부분을 확인한다.



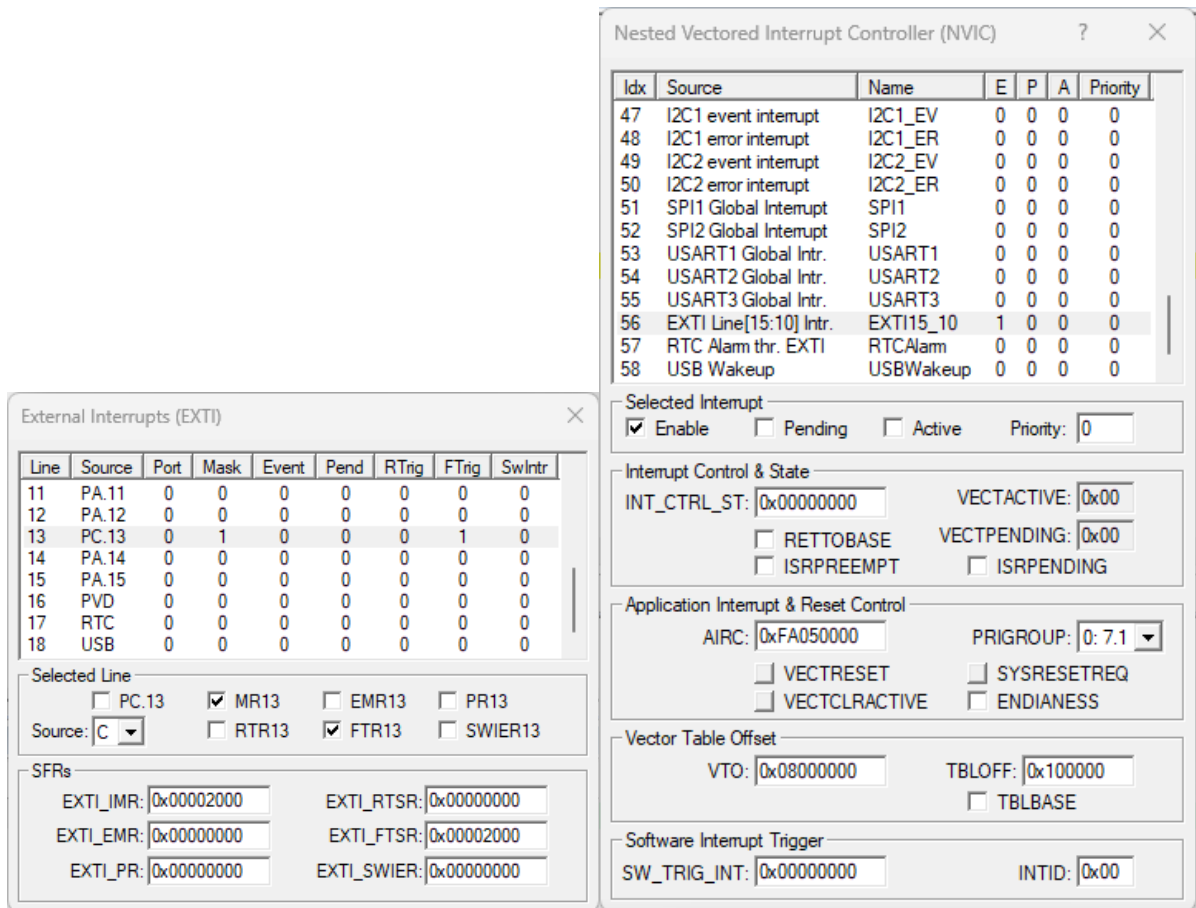


그림 8 Line 36 수행 후 EXTI, NVIC 창

Line 36까지 수행 후 EXTI 창을 보면 PC.13 13번 포트가 Mask와 FTrig가 1로 set된 것을 확인할 수 있다. 즉 PC13에 대해 falling edge로 신호가 오면 인터럽트가 발생할 수 있게 되었다. 또한 NVIC창에서 56번 EXTI Line[15:10]을 보면 E가 1로 set되어있다. 즉 enable로 56번 인터럽트가 동작할 수 있도록 활성화 되었다는 뜻이다.

**STEP 9: Lines 38-41을 통해 어떤 일이 이뤄지는지 확인한다.**

또한 line 43까지 단계별로 수행하면서 어떤 일이 수행되는지 확인한다. 특히 어느 단계에서 LED의 표시가 바뀌는지를 확인한다.

Lines 42-44로 구성되는 무한 loop에서 LED에 표시되는 내용을 변경하기 위해서는 어떤 조치가 필요한가? µVision의 Memory windows를 통해 메모리의 해당 부분을 변경하면서 수행하여 자신의 판단이 맞았는지를 확인한다.

Register	Value
Core	
R0	0x4001140C
R1	0x00000100
R2	0x40021000
R3	0x20000000
R4	0x00000000
R5	0x0000004A

Memory 1	
Address:	0x20000000
0x20000000:	4A 00 00 00 00 00 00 00
0x20000008:	00 00 00 00 00 00 00 00

그림 9 Line 41 수행 후 레지스터, 메모리 값

Line 38-41 명령어를 수행하면 r0에 GPIO\_ODR의 주소인 0x4001140C를 저장하고 r5에는 임의의 이진수 2\_01001010을 저장한다. 그리고 메모리의 sdata 공간의 주소인 0x20000000을 r3에 저장하고 해당 주소에 r5의 값을 저장한다. 그 결과 [그림 9]와 같이 레지스터 값이 들어가게 되었고 메모리에도 R5의 값이 정상적으로 들어간 것을 확인할 수 있다.

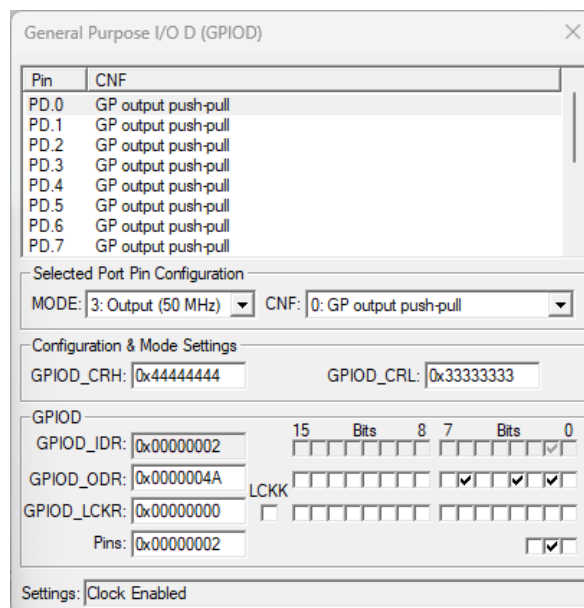


그림 10 Line 43 수행 후 LED 값

그리고 line 43을 수행한 결과 GPIO\_ODR에 메모리의 sdata 영역의 값이 그대로 저장된 것을 확인할 수 있다. 그리고 line 42-44는 무한 루프를 돌면서 계속해서 sdata 영역의 데이터를 r4로 가져와서 그대로 GPIO\_ODR에 저장해 LED를 켜게 된다. 이때 LED에 표시되는 내용을 바꾸기 위해서는 메모리의 sdata 영역의 데이터를 변경해야한다. 이를 확인하기 위해 [그림 11]과 같이 memory window에서 0x20000000에 저장된 값을 직접 바꿔준 뒤 [그림 12]와 같이 LED의 내용이 바뀐 것을 확인할 수 있다.

Memory 1	
Address:	0x20000000
0x20000000:	64 00 00 00 00 00 00 00
0x20000008:	00 00 00 00 00 00 00 00

그림 11 메모리 값 수정

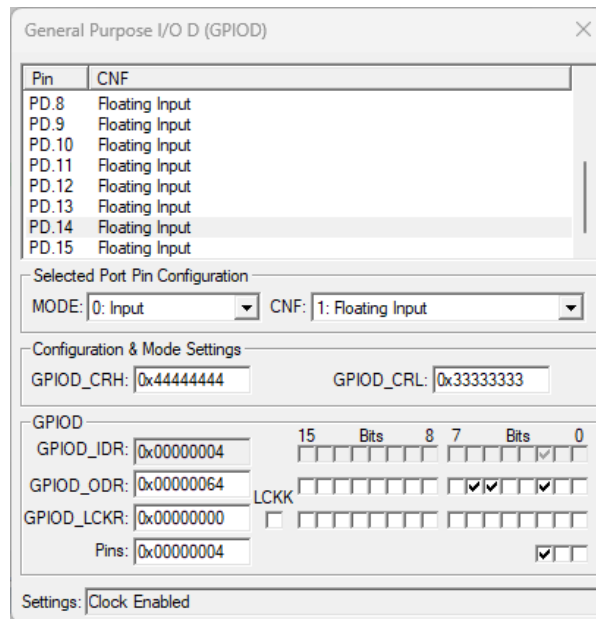


그림 12 변경된 LED 내용

**STEP 10:** Lines 48-52에서는 어떤 일이 이뤄지는가? 이 동작을 앞 단계에서 확인한 사실과 비교해서 설명해본다. 만일 EXTI15\_10IRQHandler가 위 무한 loop 내 어느 시점에서 subroutine처럼 호출되어 수행된다면 이는 LED에 어떻게 반영되겠는가?

Line 48-52 명령어는 인터럽트 핸들러로 56번 인터럽트인 EXTI15\_10인터럽트가 발생하면 해당 영역으로 jump해서 명령어들이 수행된다. 이 핸들러에서는 가장 먼저 r4, r5의 값을 stack에 푸쉬하고 sdata 주소에서 데이터를 가져와 1을 더한 뒤 다시 sdata 영역에 저장한다. 그리고 EXTI\_PR 레지스터에 0x00002000을 저장해 EXTI 13번의 pending 상태를 해제해준 뒤 처음에 push했던 r4, r5를 다시 pop하고 원래의 흐름으로 return한다. 따라서 앞 단계에서 EXTI 13번을 falling edge에서 동작하도록 설정하고 56번 인터럽트가 발생할 수 있도록 미리 세팅을 해두었기 때문에 이 명령어들이 수행될 수 있었다.

또한 이 인터럽트가 위의 line 42-44의 무한 루프 도중에 서브루틴처럼 호출되어 수행된다면 sdata의 값이 1 증가하게 되고 따라서 해당 sdata의 저장된 데이터를 가지고 LED가 동작하기 때문에 LED가 카운터처럼 하나 증가한 형태로 켜지게 될 것이다.

**STEP 11:** USER\_BUTTON 스위치는 그림 7.10에 의해 EXTI13으로 연결되며, 이 EXTI13은 표 7.2에 따르면 56번 인터럽트에 해당된다.

Vector table에서 이 인터럽트의 ISR의 시작번지를 확인해 본다. 이 주소가  $\mu$ Vision의 Disassembly window를 통해 확인 한 주소와 일치하는가? Startup code로 project에 기본적으로 추가되어 사용되고 있는 startup\_stm32f10x\_md.s를 살펴보고 인터럽트 vector table이 구성되는 방법을 파악해 본다.

Vector table에 의하면 56번 인터럽트의 주소는 0x000000E0이다. 그리고 메모리에서 해당 주소에 저장된 데이터를 살펴보면 [그림 13]에서 확인할 수 있듯이 0x08000305이다. 그리고 인터럽트 핸들러의 시작주소는 [그림 14]에서 확인할 수 있듯이 0x08000304로 [그림 13]에서 확인한 주소와 같은 것을 알 수 있다. LSB가 1차이 나는 것은 예전 branch 실험에서 확인하였듯 thumb 에러가 발생하지 않기 위해서이다.

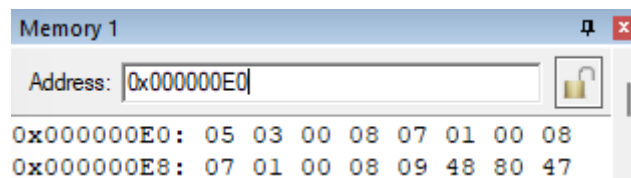


그림 13 메모리에 저장된 vector table

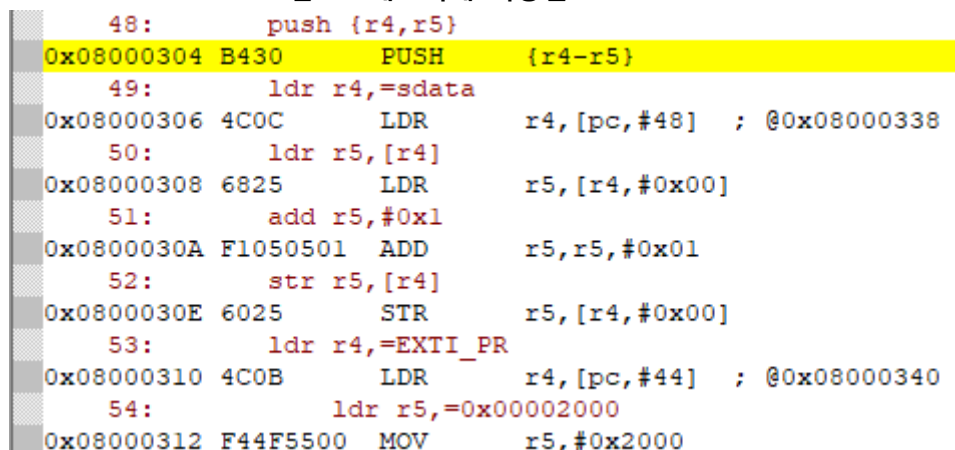


그림 14 ISR 부분의 disassembly window

그리고 마지막으로 아래 [그림 15]의 startup\_stm32f10x\_md.s 파일을 살펴보면 vector table의 구성을 확인할 수 있다. \_\_Vectors라는 readonly 데이터가 주소 0번부터 차례대로 나열되어 있는 것을 확인할 수 있다. 따라서 인터럽트가 발생하면 이 벡터테이블을 살펴보고 EXTI15\_10\_IRQHandler 함수의 위치로 점프를 하게 된다. 이를 간단히 확인하기 위해서 startup\_stm32f10x\_md.s파일에서 EXTI15\_10\_IRQHandler대신 My\_Handler로 이름을 바꾸고 우리가 작성한 lab7\_1.s 파일에서도 My\_Handler로 이름을 바꾼 결과 정상적으로 핸들러가 호출되는 것을 [그림 16]에서 확인 가능하다.

```

57 ; Vector Table Mapped to Address 0 at Reset
58 AREA RESET, DATA, READONLY
59 EXPORT __Vectors
60 EXPORT __Vectors_End
61 EXPORT __Vectors_Size

63 __Vectors DCD __initial_sp ; Top of Stack
64 DCD Reset_Handler ; Reset Handler
65 DCD NMI_Handler ; NMI Handler
66 DCD HardFault_Handler ; Hard Fault Handler
67 DCD MemManage_Handler ; MPU Fault Handler
68 DCD BusFault_Handler ; Bus Fault Handler
69 DCD UsageFault_Handler ; Usage Fault Handler
70 DCD 0 ; Reserved
71 DCD 0 ; Reserved
72 DCD 0 ; Reserved
73 DCD 0 ; Reserved
74 DCD SVC_Handler ; SVC_Handler
75 DCD DebugMon_Handler ; Debug Monitor Handler
76 DCD 0 ; Reserved
77 DCD PendSV_Handler ; PendSV_Handler
78 DCD SysTick_Handler ; SysTick_Handler
79
80 ; External Interrupts
81 DCD WWDG_IRQHandler ; Window Watchdog
82 DCD PVD_IRQHandler ; PVD through EXTI Line detect
83 DCD TAMPER_IRQHandler ; Tamper
84 DCD RTC_IRQHandler ; RTC
85 DCD FLASH_IRQHandler ; Flash
86 DCD RCC_IRQHandler ; RCC
87 DCD EXTI0_IRQHandler ; EXTI Line 0
88 DCD EXTI1_IRQHandler ; EXTI Line 1
89 DCD EXTI2_IRQHandler ; EXTI Line 2
90 DCD EXTI3_IRQHandler ; EXTI Line 3
91 DCD EXTI4_IRQHandler ; EXTI Line 4
92 DCD DMA1_Channel1_IRQHandler ; DMA1 Channel 1
93 DCD DMA1_Channel2_IRQHandler ; DMA1 Channel 2
94 DCD DMA1_Channel3_IRQHandler ; DMA1 Channel 3
95 DCD DMA1_Channel4_IRQHandler ; DMA1 Channel 4
96 DCD DMA1_Channel5_IRQHandler ; DMA1 Channel 5
97 DCD DMA1_Channel6_IRQHandler ; DMA1 Channel 6
98 DCD DMA1_Channel7_IRQHandler ; DMA1 Channel 7
99 DCD ADC1_2_IRQHandler ; ADC1_2
100 DCD USB_HP_CAN1_TX_IRQHandler ; USB High Priority or CAN1 TX
101 DCD USB_LP_CAN1_RX0_IRQHandler ; USB Low Priority or CAN1 RX0
102 DCD CAN1_RX1_IRQHandler ; CAN1 RX1
103 DCD CAN1_SCE_IRQHandler ; CAN1 SCE
104 DCD EXTI9_5_IRQHandler ; EXTI Line 9..5
105 DCD TIM1_BRK_IRQHandler ; TIM1 Break
106 DCD TIM1_UP_IRQHandler ; TIM1 Update
107 DCD TIM1_TRG_COM_IRQHandler ; TIM1 Trigger and Commutation
108 DCD TIM1_CC_IRQHandler ; TIM1 Capture Compare
109 DCD TIM2_IRQHandler ; TIM2
110 DCD TIM3_IRQHandler ; TIM3
111 DCD TIM4_IRQHandler ; TIM4
112 DCD I2C1_EV_IRQHandler ; I2C1 Event
113 DCD I2C1_ER_IRQHandler ; I2C1 Error
114 DCD I2C2_EV_IRQHandler ; I2C2 Event
115 DCD I2C2_ER_IRQHandler ; I2C2 Error
116 DCD SPI1_IRQHandler ; SPI1
117 DCD SPI2_IRQHandler ; SPI2
118 DCD USART1_IRQHandler ; USART1
119 DCD USART2_IRQHandler ; USART2
120 DCD USART3_IRQHandler ; USART3
121 DCD EXTI15_10_IRQHandler ; EXTI Line 15..10
122 DCD RTCAlarm_IRQHandler ; RTC Alarm through EXTI Line
123 DCD USBWakeUp_IRQHandler ; USB Wakeup from suspend

```

그림 15 startup\_stm32f10x\_md.s 파일의 vector table

```

49  My_Handler proc
50      push {r4,r5}
51      ldr r4,=sdata
52      ldr r5,[r4]
53      add r5,#0x1
54      str r5,[r4]
55      ldr r4,=EXTI_PR
56      ldr r5,=0x00002000
57      str r5,[r4]
58      pop {r4,r5}
59      bx lr
60  endp

```

그림 16 My\_Handler가 호출된 모습

STEP 12: 앞서의 과정에서 설정된 break-point들을 모두 제거한 후 line 48 명령어에만 break-point를 설정한다. ISR의 처음에 위치한 이 명령어는 인터럽트가 발생해야 수행된다.

Run icon을 이용하여 전체 프로그램을 수행한다.

USER\_BUTTON 스위치에 해당하는 GPIO port C 13번 pin을 click하여 인터럽트를 발생시킨 후 이 명령어에서 프로그램에서 프로그램이 정지하는 것을 확인한다.

그림 7.8을 참조하여 stack에 저장된 내용을 확인한다. 또한 SP, LR, PC의 값을 확인한다. 이 과정을 통해 인터럽트가 발생했을 때 CPU가 취하는 조치를 어떻게 정리할 수 있는가?

또한 어떤 명령어를 수행하는 중에 인터럽트가 발생했다고 판단했는가? 그 근거는 무엇인가?

Line 48에만 breakpoint를 설정하고 프로그램을 run한 상태에서 GPIO port C의 13번 핀을 2번 클릭하여 falling edge에서 인터럽트를 발생시켰다. 그 결과 [그림 17]처럼 인터럽트 핸들러로 들어 가서 프로그램이 정지하는 것을 확인할 수 있다.

```

47: EXTI15_10_IRQHandler proc
0x08000302 E7FC      B      0x080002FE
48:      push {r4,r5}
0x08000304 B430      PUSH     {r4-r5}
49:      ldr r4,=sdata
0x08000306 4C0C      LDR      r4,[pc,#48] ; @0x08000338
50:      ldr r5,[r4]
0x08000308 6825      LDR      r5,[r4,#0x00]
51:      add r5,#0x1
0x0800030A F1050501 ADD      r5,r5,#0x01

```

그림 17 Line 48에서 멈춘 상황

이 상황에서 SP, LR, PC의 레지스터 값과 스택에 저장된 데이터를 확인해보면 아래 [그림 18], [그림 19]처럼 확인할 수 있다.

Register	Value
<b>Core</b>	
R0	0x4001140C
R1	0x00000100
R2	0x40021000
R3	0x20000000
R4	0x0000004A
R5	0x0000004A
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x200005E8
R14 (LR)	0xFFFFFFFF9
R15 (PC)	0x08000304
xPSR	0x61000038
N	0
Z	1
C	1
V	0
Q	0
T	1
IT	Disabled
ISR	56

그림 18 Line 48 수행 전 레지스터 값

그림 19 Line 48 수행 전 stack 값

먼저 스택에 저장된 데이터를 살펴보면 메모리가 낮은 부분부터 0x4001140C, 0x00000100, 0x40021000, 0x20000000, 0x00000000, 0x0800013B, 0x08000302, 61000000이 저장되어 있는 것을 확인할 수 있다. 이는 이론에서 살펴본 대로 인터럽트 핸들러로 들어가면서 자동으로 r0-r3, r12, lr, pc, psr의 값이 스택에 저장된 것을 확인할 수 있다. 그 후 sp는 현재 스택의 탑인 0x200005E8으로 바뀌고, LR은 원래의 흐름으로 main stack을 가지고 돌아간다는 의미의 0xFFFFFFFF9이 저장되고 PC는 vector table에서 가져온 0x08000304로 업데이트 된 것을 확인할 수 있다. 즉 인터럽트가 발생하자 cpu는 자동적으로 일부 레지스터의 내용을 stack에 저장하고 vector table에서 핸들러의 주소를 가져오고 SP, LR, PC의 값을 핸들러에 적합하도록 업데이트 한 것을 확인할 수 있다.

그리고 스택에 저장된 내용에서 PC값을 살펴보면 0x08000302인 것을 확인할 수 있다. 즉 원래대로라면 다음에 수행되어야 할 명령어는 0x08000302임으로 마지막으로 실행된 명령어는

0x08000300이었고 해당 명령어가 수행되던 중에 인터럽트가 발생하였다고 예상할 수 있다.

**STEP 13: ISR을 명령어별로 수행하면서 동작내용을 확인한다. 마지막 부분에서 interrupt return 과정을 확인해 본다. SP, LR, PC변화를 통해 이 과정을 STEP 12의 관찰 내용과 비교하며 설명해 보자.**

ISR의 동작을 살펴보면 먼저 핸들러에서 사용할 레지스터인 r4, r5의 값을 스택에 저장해두고 r4에는 sdata 공간의 주소를 가져온다. 그리고 해당 공간에서 데이터를 r5로 로드한다. 그리고 r5에 1을 더한 뒤 다시 r4의 공간 즉 sdata 공간에 r5값을 다시 저장한다. 그리고 r4에 EXTI\_PR의 주소를 가져오고 r5에는 0x00002000을 저장한 뒤 r5의 값을 r4에 저장한다. 이를 통해 EXTI 13의 pending을 지워준다. 마지막으로 스택에 저장하였던 r4, r5를 pop해서 원래의 값으로 복구해준다. 그리고 마지막으로 bx lr을 통해 원래의 흐름으로 복귀한다.

Register	Value
<b>Core</b>	
R0	0x4001140C
R1	0x00000100
R2	0x40021000
R3	0x20000000
R4	0x0000004A
R5	0x0000004A
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000608
R14 (LR)	0x0800013B
R15 (PC)	0x08000302
xPSR	0x61000000
N	0
Z	1
C	1
V	0
Q	0
T	1
IT	Disabled
ISR	0

**그림 20 ISR 리턴 직후 레지스터 값**

[그림 20]을 통해서 ISR이 종료되고 리턴한 직후의 레지스터 값을 확인할 수 있다. r0-r3와 r12의 값은 이번 핸들러로 인해 바뀐 적은 없지만 원상태로 복구되었고 SP도 원래 ISR로 들어가기 전의 값인 0x20000608로 돌아가고 LR도 원래의 값인 0x0800013B로 복구되고 PC값도 0x08000302로 정상적으로 복구된 것을 확인할 수 있다. 즉 핸들러로 들어가기 전의 R0-R3, R12, SP, LR, PC값을



스택에 저장하였다가 그대로 다 정상적으로 복구된 것을 확인할 수 있다. 또한 [그림 21]의 56번 인터럽트를 보면 ISR을 수행 하는 도중에는 해당 인터럽트가 active 상태이고 해당 ISR이 종료되고 return되면 자동으로 NVIC의 active bit가 0으로 reset되는 것을 확인할 수 있다.

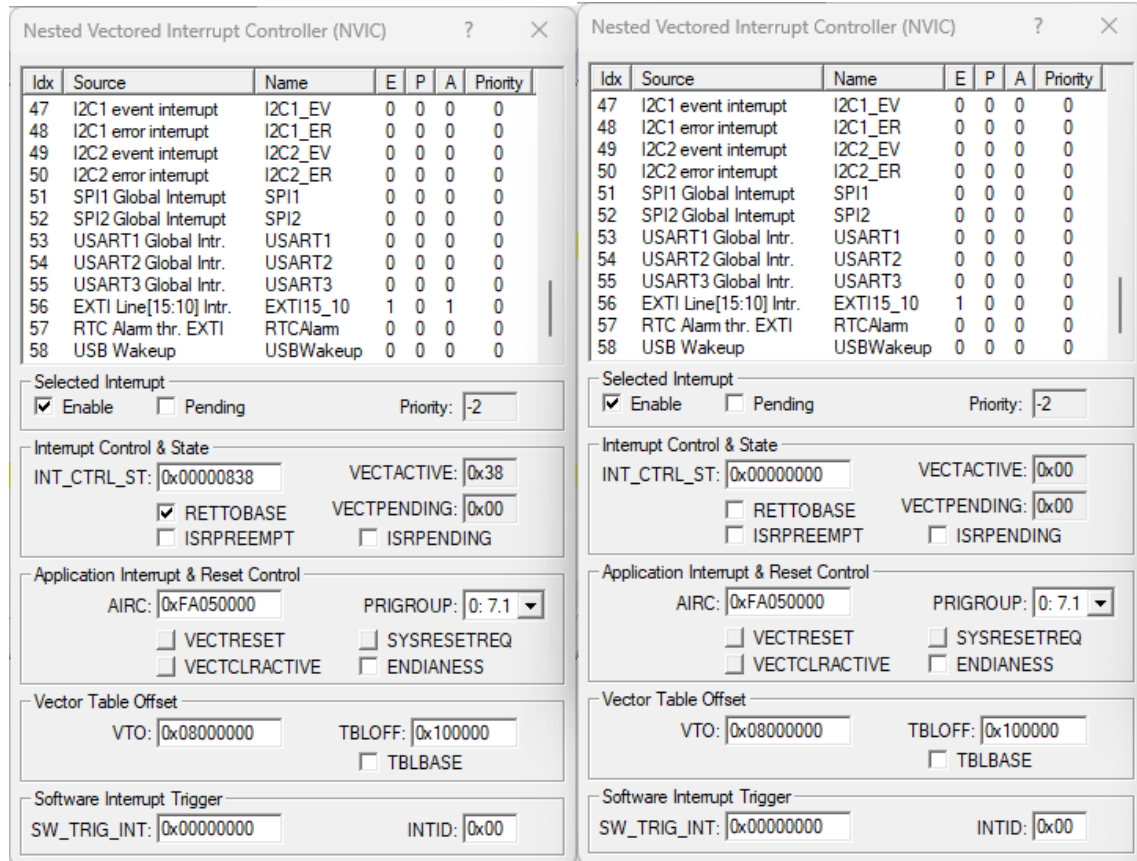


그림 21 ISR 수행 후 NVIC 레지스터의 변화(좌측 : ISR 수행 중, 우측 : ISR 수행 후)

STEP 14: Run icon을 통해 프로그램을 연속 수행하면서 위 두 과정을 반복해 보자. 인터럽트가 발생할 때 수행 중이던 명령어(interrupted instruction)가 매번 다른지 확인한다.

이를 통해 프로세서가 ISR을 처리하는 일반적인 과정 서술해보자.

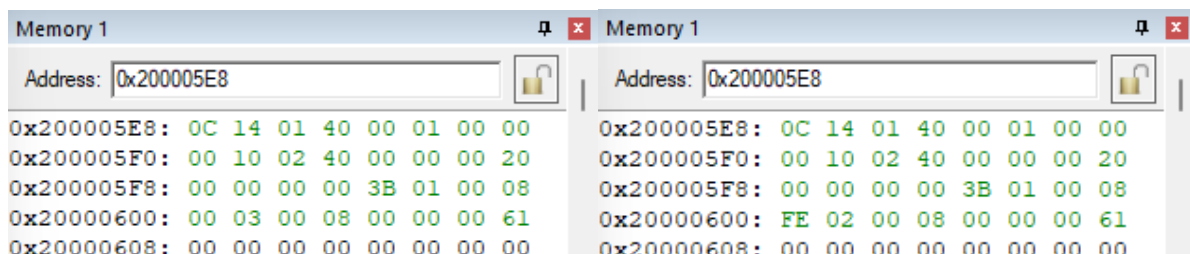


그림 22 인터럽트 발생 시 스택에 저장된 값

[그림 22]에서 확인할 수 있는 것처럼 ISR로 들어가서 stack에 저장되는 값 중에서 PC값을 보면 0x08000300, 0x080002FE, 그리고 아까 [그림 19]의 0x08000302처럼 여러가지 값이 모두 확인된다.

이는 어떠한 명령어를 수행중이더라도 인터럽트가 발생하면 해당 명령어만 마치고 바로 인터럽트 핸들러로 들어온다는 것을 의미한다.

즉 일반적으로 ISR은 인터럽트가 걸리고 프로세서가 수행중이던 명령어를 마치고 바로 stack에 r0-r3, r12, sp, lr, pc의 값을 stack에 저장하고 vector table에서 ISR의 시작주소를 가져온 뒤 sp, lr, pc의 값을 업데이트 하고 ISR의 동작이 수행된다. 그리고 다시 ISR이 끝나면 스택에 저장하였던 값을 복구하고 NVIC의 active bit를 해제하고 원래의 흐름으로 돌아가는 방식으로 진행된다.

## 2) 실험 2

STEP 15: Program 7.1을 계속 이용한다.

STEP 2의 표에서 WAKEUP 스위치를 통해 발생하는 인터럽트를 처리하기 위한 ISR을 추가해보자.

WAKEUP 스위치는 PA0에 연결된다고 가정한다.

```
59  EXTI0_IRQHandler proc
60      push {r4,r5}
61      ldr r4,=sdata
62      ldr r5,[r4]
63      sub r5,#0x1
64      str r5,[r4]
65      ldr r4,=EXTI_PR
66      ldr r5,=0x00000001
67      str r5,[r4]
68      pop {r4,r5}
69      bx lr
70  endp
```

그림 23 lab7\_2.s 에 추가한 코드

PA0인터럽트가 발생하면 EXTI0 인터럽트가 발생하기 때문에 이에 해당하는 EXTI\_IRQHandler ISR을 추가하였다. 동작은 EXTI15\_10\_IRQHandler에서 add부분만 sub로 바뀌서 ISR이 수행될 때 sdata영역의 데이터가 1감소하도록 하였다.

STEP 16: Line 28 명령어의 operand를 0x00002001로 변경한다(7.2.6절 참조).

Line 37의 주석처리 부분에 아래 내용을 추가한다(7.2.5참조).

ldr r0, =NVIC\_ISER0

ldr r1, =0x00000040

str r1, [r0]

**이 변경의 의미를 교재에서 찾아서 확인한다.**

먼저 Line 28을 `ldr r1,=0x00002000`에서 `ldr r1,=0x00002001`로 바꾼 이유는 이 다음 명령어에서 `EXTI_FTSR`과 `EXTI_IMR`에 해당 `r1`의 값을 저장하는데 이때 `EXTI13`뿐만 아니라 `EXTI0`도 falling edge에서 `EXTI0`가 인터럽트를 발생할 수 있게 하기 위해서 `0x00002001`로 바꾸었다.

또한 line 37뒤에 STEP 16에서 주어진 명령어를 추가하는 것은 `EXTI0` 인터럽트는 22번 인터럽트인데 이 인터럽트를 사용하기 위해서는 `NVIC_ISER0`에서  $22-15=7$ 번째 bit를 1로 set해야 하는데 이것을 수행하는 명령어가 새로 추가한 명령어이다.

**STEP 17: Line 11 다음에 `export EXTI0_IRQHandler`를 추가한다.**

기존의 lines 47-58을 line 58와 line 59 사이에 복사한 후 procedure 이름을 `export EXTI0_IRQHandler`로 변경하고, line 51에 해당하는 명령어를 `add`에서 `sub`로 변경한다. 또한 line 54에 해당하는 명령어의 operand를 `0x00000001`로 변경한다. 이 변경들의 의미를 확인한다.

[그림 23]에서 확인할 수 있는 것처럼 코드를 작성하여 `EXTI0`에 해당하는 ISR을 작성하였다. Line 51에 해당하는 명령어를 `add`에서 `sub`로 변경한 것은 `EXTI0` 인터럽트가 발생하였을 때 LED가 카운터에서 1 감소한 형태로 동작하기 위해서 `sub`로 바꾸었다. 또한 line 54에 해당하는 명령어를 `ldr r5,=0x00000001`로 바꾼 것은 인터럽트에서 필요한 동작을 수행하고 마지막에 `EXTI_PR`에서 1번째 bit를 1로 set해서 `EXTI0` 인터럽트의 pending상태를 풀어주기 위해서 `0x00000001`로 바꿔주었다.

**STEP 18: 이 프로그램을 break point 없이 연속으로 수행하면서 `USER_BUTTON` 스위치와 `WAKEUP` 스위치에 해당하는 pin을 해당 GPIO port 창을 통해 번갈아 눌러보자.**

**LED가 연결된 Port D의 변화가 변경된 코드의 의도와 일치하는가?**

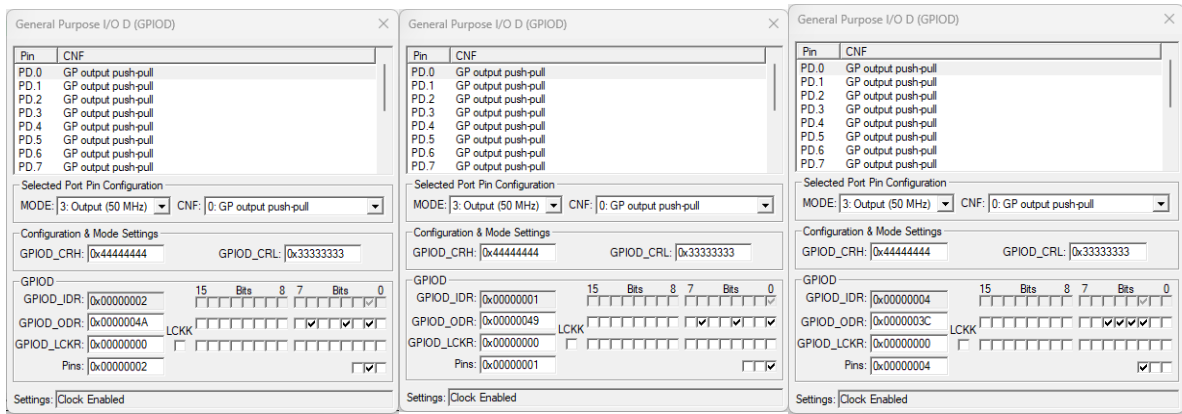


그림 24 PA0 pin 클릭 후 GPIOD 창 (좌측: 0회 클릭, 중간: 2회 클릭(set, reset), 우측: 수십번 클릭) [그림 24]에서 확인할 수 있는 것처럼 WAKEUP 스위치에 PA0를 클릭해본 결과를 GPIOD 창을 통해 확인해보면 LED가 카운터에서 1 감소하는 형태로 동작하는 것을 확인할 수 있다. 또한 USER\_BUTTON 스위치를 통해서 [그림 25]처럼 다시 USER\_BUTTON을 통해서 카운트 업 시킬 수도 있는 것을 확인할 수 있었다.

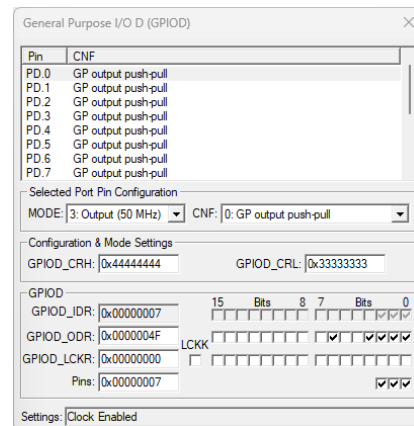


그림 25 PA13 pin 클릭 후 GPIOD 창

STEP 19: 만일 WAKEUP 스위치가 USER\_BUTTON 스위치와 달리 누르지 않을 때 low 상태이고 누르면 high 상태가 된다고 가정한다면 (즉, rising edge에서 인터럽트가 발생한다면) 어떤 조치가 필요한지 교재의 자료(7.2.6절)를 찾아 변경하고 확인해보자.

```
28 ; EXTI initialization
29     ldr r1,=0x00002000
30     ldr r0,=EXTI_FTSR
31     str r1,[r0]
32     ldr r1,=0x00000001
33     ldr r0,=EXTI_RTSR
34     str r1,[r0]
35     ldr r1,=0x00002001
36     ldr r0,=EXTI_IMR
37     str r1,[r0]
```

그림 26 lab7\_2.s 에서 수정한 코드

[그림 26]에서 확인할 수 있는 것처럼 WAKEUP 스위치는 rising edge에서 동작해야 함으로 EXTI\_FTSR이 아닌 EXTI\_RTSR에서 1번째 bit를 1로 set해주어야 한다. 이를 위해서 line 32-34를 추가하였다. 그리고 falling edge에서는 동작하지 않기 위해서 line 29-31은 0x00002001을 다시 0x00002000으로 수정하였다. 그리고 EXTI\_IMR은 동일하게 1로 set해야 하기 때문에 동일하게 EXTI\_IMR에는 0x00002001을 저장하였다.

#### 4. Exercises

**1) 본 실험과정에서는 vector table이 flash memory 영역에 위치한다. 이 table을 RAM 영역에 위치하기 위한 프로세서의 기능(relocate)을 교재(131페이지)에서 언급하고 있는데, 어떤 경우에 이러한 table relocation이 필요하겠는가?**

ROM과 RAM의 가장 큰 차이점은 ROM은 프로그램 수행 도중 변경될 수 없다는 것이고 RAM의 경우는 프로그램 수행 도중에도 수정이 될 수 있다. 따라서 만약 프로그램 수행 도중 인터럽트에 대해 여러가지 핸들러를 사용하고 싶은 경우에는 vector table을 RAM영역에 relocation한 뒤 필요에 따라 다른 핸들러의 주소를 적어서 프로그램 수행 도중에도 변경이 되도록 할 수 있다.

**2) 프로세서를 이용한 연산 중에 어떤 변수를 0으로 나누는 것과 같은 동작(divide by 0), 또는 프로세서에서 정의되지 않는 명령어(undefined instruction)를 실행할 때 프로세서는 어떤 조치를 취하는가? 이를 본 장에서 다루는 인터럽트(또는 exception)와 연관지어 설명해보자.**

프로세서에서 0을 나누는 것과 같은 동작이나 정의되지 않은 명령어를 실행하는 경우에는 usage faults가 발생하게 된다. Usage faults는 인터럽트와 비슷하지만 차이점이라면 외부에서 신호가 들어오는 것이 아니라 내부 동작에 의해 특정 상황이 되면 발생하는 오류를 의미한다. 기본적인 동작은 인터럽트처럼 fault가 발생하면 fault handler로 이동하여 handler의 동작을 수행하게 된다. 다만 기본적으로는 fault handler는 disable되어 있는데 SCB\_SHCSR 레지스터에서 원하는 fault를 enable 시켜주어야만 원하는 fault handler가 수행된다. 다만 0을 나누는 행위는 경우에 따라 fault가 발생하지 않을 수 있다. 아래 [사진 27]에서 확인할 수 있는 것처럼 DIV\_0\_TRP라는 bit가

SCB\_CCR 레지스터에 존재한다. 이 bit를 0으로 reset하면 0으로 나누었을 때 fault가 발생하지 않고 그냥 0으로 처리하고 1로 set되어 있으면 0으로 나누었을 때 fault 발생하여 fault handler가 수행되도록 한다.

#### DIV\_0\_TRP

Enables faulting or halting when the processor executes an SDIV or UDIV instruction with a divisor of 0:

0: Do not trap divide by 0

1: Trap divide by 0.

When this bit is set to 0, a divide by zero returns a quotient of 0.

그림 27 Cortex-M3 Programming Manual의 DIV\_0\_TRP 설명

**3) 이 장은 maskable interrupt를 중심으로 구성되었다. NMI(non-maskable interrupt)는 어떤 목적으로 사용되는지 알아보자.**

NMI는 실험에서 사용하였던 인터럽트와 다르게 인터럽트를 enable시켜줄 필요도 없고 인터럽트인 발생하였을 때 마스킹을 통해 무시하거나 할 수도 없는 인터럽트이다. NMI는 reset 바로 다음의 우선순위를 가지고 있을 정도로 매우 중요한 인터럽트로 매우 중요한 오류를 처리할 때 사용된다. 예를 들어 하드웨어나 메모리에 오류가 있거나 시스템에 심각한 문제가 발생한 경우에 최우선으로 NMI를 발생시켜 문제를 해결하고자 한다. 또는 NMI를 이용해서 시스템을 강제로 리셋할 수 있다.

**4) 표 7.2에 의하면 reset도 인터럽트와 같은 방식으로 다뤄지고 있음을 알 수 있다. 프로세서가 reset되었을 때 처음 동작하는 프로그램의 시작주소를 어떻게 확인할 수 있겠는가?**

Reset도 인터럽트와 같은 방식으로 reset에 해당하는 handler가 vector table에 저장되어 있다. 이때 vector table의 주소는 0x00000000부터 시작되는데 아래 [그림 28]은 cortex-m3 programming manual에서 가져온 사진으로 cortex-m3 프로세서의 vector table을 볼 수 있다. 여기서도 0x00000000에는 Initial SP value가 저장되어 있고 바로 다음 0x00000004에 reset\_handler의 시작 주소가 적혀있다. 따라서 프로세서가 reset되었을 때 바로 0x00000000에서 SP를 가져오고 0x00000004를 참조해서 reset\_handler의 주소로 이동해 수행되게 된다.

Exception number	IRQ number	Offset	Vector
83	67	0x014C	IRQ67
.	.	.	.
.	.	.	.
.	.	.	.
18	2	0x004C	IRQ2
17	1	0x0048	IRQ1
16	0	0x0044	IRQ0
15	-1	0x0040	Systick
14	-2	0x003C	PendSV
13		0x0038	Reserved
12			Reserved for Debug
11	-5	0x002C	SVCall
10			Reserved
9			
8			
7			
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
		0x0000	Initial SP value

그림 28 Cortex-M3 vector table

## 5) 일반적으로 software interrupt를 어떻게 정의하는가? Software interrupt는 주로 어떤 목적으로 사용되는가?

소프트웨어 인터럽트는 실험에서 사용한 인터럽트처럼 하드웨어에서 발생하는 신호로 발생하는 인터럽트가 아니라 소프트웨어에서 프로그래머의 의도에 따라서 발생시키는 인터럽트를 말한다. 보통 시스템콜을 사용하기 위해서 소프트웨어 인터럽트를 사용한다. 프로그램이 동작할 때 보통의 경우는 유저모드와 커널모드로 구분하여 동작하는데 일반적으로 커널모드에서 시스템을 다루는 작업을 하고 유저모드에서는 일반적인 프로그램을 수행한다. 이때 유저모드에서 커널모드로 진입해 시스템을 다루는 동작을 하기 위해서 시스템콜을 사용하는데 이를 위해서 소프트웨어 인터럽트를 사용한다.

6) 3.3.4절에서 bit-banding을 통한 메모리 access과정이 atomic하게 동작하기 때문에 interrupt handler의 동작으로 인한 문제점을 피할 수 있다고 설명하는 부분을 다시 한번 살펴보자. (그림 3.10 참조)

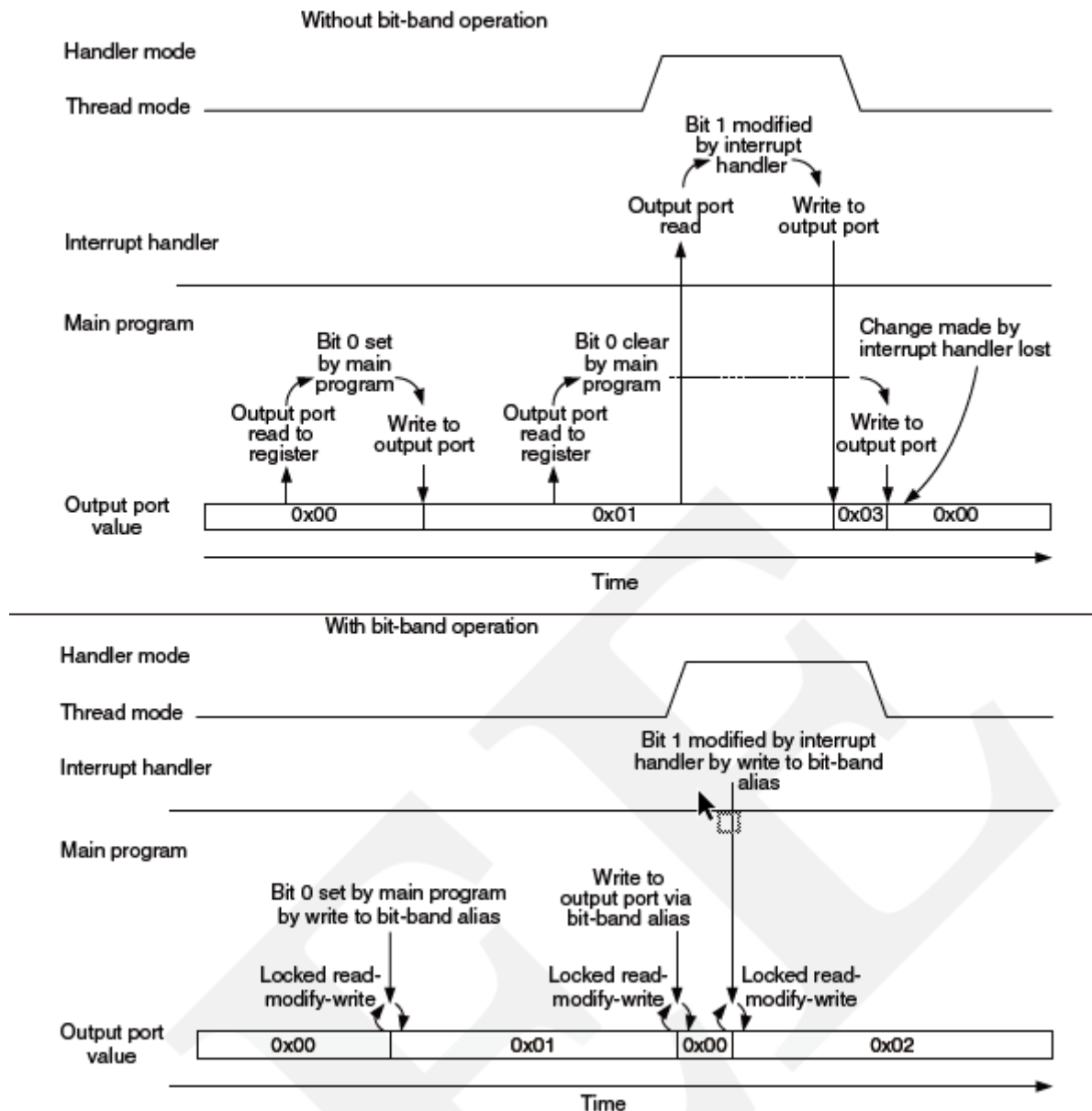


그림 29 교재 그림 3.10 중 일부

[그림 29]의 위쪽에서 확인할 수 있는 것처럼 bit-banding을 사용하지 않은 경우에는 메모리의 데이터에 접근해서 수정하는 과정에서 인터럽트가 발생하게 되면 인터럽트 핸들러가 메모리의 데이터를 수정하더라도 다시 원래의 흐름으로 돌아왔을 때 인터럽트 핸들러에 의해 수정된 데이터를



바탕으로 데이터를 수정하는 것이 아니라 기존에 진행되던 흐름에서 데이터가 수정되기 때문에 인터럽트 핸들러의 결과를 반영하지 못하는 경우가 발생할 수 있다.

하지만 [그림 29]의 아래쪽처럼 bit-banding을 하는 경우에는 atomic하게 메모리에 접근하기 때문에 중간에 인터럽트가 발생할 수 없고 하나하나의 중요한 동작이 다 수행되고 인터럽트가 발생하기 때문에 인터럽트에 의해 변경된 경우에도 잘 반영하여 메모리에 접근할 수 있다는 장점이 있다.

## 5. 추가 실험

### 1) Exception Catch

```
1 SCB_SHCSR equ 0xE000ED24
2 SCB_CCR equ 0xE000ED14
3     export UsageFault_Handler
4     area lab7_3, code
5     entry
6     __main proc
7         export __main [weak]
8         ldr r0,=SCB_SHCSR
9         ldr r1,=0x00040000
10        str r1,[r0]
11        ldr r0,=SCB_CCR
12        ldr r1,=0x00000010
13        str r1,[r0]
14
15        mov r0,#10
16        mov r1,#0
17        udiv r0,r1
18        b .
19    endp
20 UsageFault_Handler proc
21     mov r0,#0
22     bx lr
23     endp
24     end
```

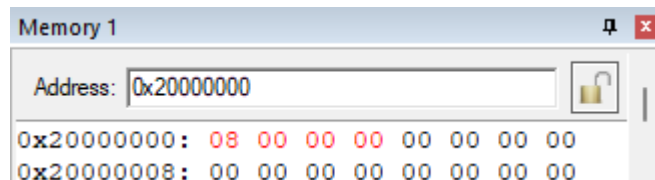
그림 30 직접 작성한 lab7\_3.s 코드

위 [그림 30]과 같이 divide by 0에 대한 exception catch를 위해서 직접 코드를 작성해보았다. 먼저 line 8-13을 통해서 usage fault를 enable해주고 divide by 0가 발생하였을 때 usage fault가 발생하도록 해주었다. 그리고 r0,r1에 각각 10,0을 넣고 10/0을 해주었다. 그 결과 아래 [그림 31]처럼 핸들러로 들어간 뒤 r0의 값을 0으로 설정하고 원래의 흐름으로 돌아가도록 해주었다.

Register	Value
<b>Core</b>	
R0	0x00000000
R1	0x00000000

그림 31 Handler 이후 레지스터 값

## 2) Nested Interrupt and Race Condition



Memory 1	
Address:	0x20000000
0x20000000:	08 00 00 00 00 00 00 00
0x20000008:	00 00 00 00 00 00 00 00

그림 32 Bit banding 하지 않은 경우의 결과

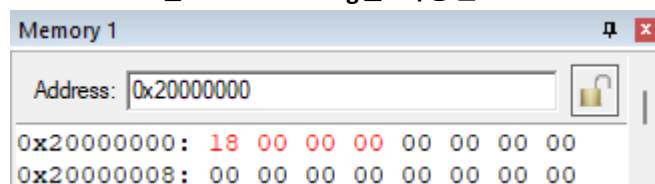
인터럽트가 중첩되는 경우 메모리에 접근할 때 주의를 하지 않으면 원치 않는 동작을 하는 경우가 많다. 예를 들어 이번 주어진 코드에서 인터럽트가 중첩되어 실행되는 경우 인터럽트 13 핸들러에 의해 변경된 메모리의 값이 반영되지 않는 것을 [그림 32]에서 확인할 수 있다. 이를 해결하기 위해서는 atomic하게 동작하는 bit-banding을 사용하는 것이 좋다. 아래 [그림 33]은 bit-banding을 사용해서 메모리에 접근하도록 작성한 코드이다. 이 결과는 [그림 34]와 같이 인터럽트 13의 결과에는 영향을 미치지 않고 인터럽트 14가 원하는 동작만을 수행한 것을 확인할 수 있다.

```

37 ; 인터럽트 14 핸들러
38 EXTI1_IRQHandler proc
39
40 ; bit 3을 설정
41 LDR R5, =0x2200000C
42 MOV R4, #1
43
44 ; 인터럽트 13 호출
45 LDR R0, =0xE000E200 ; NVIC의 STIR 레지스터 주소를 R0에 로드
46 MOV R1, #(1<<6) ; 인터럽트 13의 번호를 R1에 로드
47 STR R1, [R0] ; STIR 레지스터에 인터럽트 번호를 쓰기
48
49 STR R4, [R5]
50
51 BX LR ; 인터럽트 핸들러 종료
52 endp

```

그림 33 Bit-banding을 사용한 코드



Memory 1	
Address:	0x20000000
0x20000000:	18 00 00 00 00 00 00 00
0x20000008:	00 00 00 00 00 00 00 00

그림 34 Bit banding 사용한 경우의 결과

Line 41-42에서 bit-band영역을 접근하고 r4에 저장하려는 데이터 1을 로드한 뒤 인터럽트가 걸려서 인터럽트 13에 의해 메모리의 bit 3가 1로 set되었고 그 후에 인터럽트 14는 line 49를 통해서 bit 4영역만 1로 set해서 바꾸기 때문에 인터럽트 13에 의해 바뀐 결과 또한 잘 반영되어 메모리에 저장된 것을 알 수 있다. 이를 통해서 인터럽트에서 메모리에 접근할 때는 다른 인터럽트에 의해 중첩되어 race condition이 발생할 수 있기 때문에 atomic하게 동작하는 bit-banding을 통해 메모리에 접근하는 것이 안전하다.

### 3) Interrupt Priority

인터럽트 사이에는 우선순위가 존재한다. 우선순위에는 preempt priority와 subpriority가 존재한다. Preempt priority가 높은 인터럽트는 다른 인터럽트 Preempt priority가 높은 인터럽트는 다른 인터럽트보다 늦게 발생하더라도 새치기해서 인터럽트 핸들러가 수행된다. Subpriority는 동시에 여러 인터럽트가 발생하였을 때의 우선순위이다. 우리가 사용하는 STM32F103RB 보드는 NVIC\_IIRPx 레지스터를 통해서 우선순위를 다음 [그림 35]와 같이 지정할 수 있다. [그림 35]가 포함된 추가실험을 위한 코드는 우리가 실험2에서 사용하였던 코드에 우선순위를 추가한 코드이다. 그리고 [그림 36]을 통해서 인터럽트의 우선순위가 8과 7로 세팅된 것을 확인할 수 있다.

```

41      ldr r0,=0xE000E404
42      mov r1,#0x00800000
43      str r1,[r0]
44      ldr r0,=0xE000E428
45      mov r1,#0x00000070
46      str r1,[r0]

```

그림 35 lab7\_5.s에 추가한 우선순위 코드

Idx	Source	Name	E	P	A	Priority	Idx	Source	Name	E	P	A	Priority
22	EXTI Line0 Interrupt	EXTI0	0	0	0	8	47	I2C1 event interrupt	I2C1_EV	0	0	0	0
23	EXTI Line1 Interrupt	EXTI1	0	0	0	0	48	I2C1 error interrupt	I2C1_ER	0	0	0	0
24	EXTI Line2 Interrupt	EXTI2	0	0	0	0	49	I2C2 event interrupt	I2C2_EV	0	0	0	0
25	EXTI Line3 Interrupt	EXTI3	0	0	0	0	50	I2C2 error interrupt	I2C2_ER	0	0	0	0
26	EXTI Line4 Interrupt	EXTI4	0	0	0	0	51	SPI1 Global Interrupt	SPI1	0	0	0	0
27	DMA Channel 1	DMA Ch1	0	0	0	0	52	SPI2 Global Interrupt	SPI2	0	0	0	0
28	DMA Channel 2	DMA Ch2	0	0	0	0	53	USART1 Global Intr.	USART1	0	0	0	0
29	DMA Channel 3	DMA Ch3	0	0	0	0	54	USART2 Global Intr.	USART2	0	0	0	0
30	DMA Channel 4	DMA Ch4	0	0	0	0	55	USART3 Global Intr.	USART3	0	0	0	0
31	DMA Channel 5	DMA Ch5	0	0	0	0	56	EXTI Line[15:10] Intr.	EXTI15_10	0	0	0	7
32	DMA Channel 6	DMA Ch6	0	0	0	0	57	RTC Alarm thr. EXTI	RTCAlarm	0	0	0	0
33	DMA Channel 7	DMA Ch7	0	0	0	0	58	USB Wakeup	USBWakeup	0	0	0	0

그림 36 인터럽트 우선순위

이 상태에서 인터럽트를 발생시키면 EXTI0 핸들러가 동작하고 있었을 때 EXTI15\_10 인터럽트가 발생하면 바로 EXTI15\_10 인터럽트로 이동하고 반면에 EXTI15\_10 핸들러가 동작하고 있을 때 EXTI0 인터럽트가 발생하면 EXTI15\_10 핸들러의 동작이 끝나고 EXTI0 핸들러가 동작한다. 이를

확인하기 위해 디버거 모드에서 call stack을 보면 EXTI0 핸들러 위에 EXTI15\_10 핸들러가 호출된 것을 확인할 수 있다.

Call Stack + Locals		
Name	Location/Value	Type
EXTI15_10_IRQHandler	0x0800031C	void f()
EXTI0_IRQHandler	0x08000336	void f()

그림 37 인터럽트 핸들러의 우선순위 증거

#### 4) Interrupt Blocking

코드를 작성하다 보면 인터럽트가 발생하지 못하도록 막아야 하는 경우가 많다. 예를 들어 멀티쓰레드로 동시에 여러 쓰레드를 통해서 작업을 수행하고자 할 때 동기화 문제를 해결하기 인터럽트를 임시로 막아야 하는 경우가 많다. 이를 위해서 간단하게 다음 [그림 38]과 같이 코드를 작성하였다.

```

59  loop  cpsid I
60      mov r4, #10000
61  loop2 cmp r4, #0
62      beq end2
63      sub r4, #1
64      b loop2
65  end2  cpsie I
66      ldr r4, [r3]
67      str r4, [r0]
68      b loop

```

그림 38 직접 작성한 lab7\_6.s 코드 일부

위 코드에서 반복문을 수행할 때 우선 cpsid I 명령어를 통해서 모든 인터럽트를 block한다. 그리고 일정 시간동안 기다리도록 10000번동안 loop2를 돌게 된다. 그리고 난 뒤 cpsie I를 통해서 인터럽트를 허용해준다. 그렇게 되면 인터럽트가 발생하여 인터럽트 핸들러가 수행될 수 있다. 아래 [그림 39]는 인터럽트를 블럭하고 나서 인터럽트가 발생하였을 때 해당 인터럽트가 pending되어 있는 모습이다. 아무래도 인터럽트가 한순간에 블록되고 풀리고 핸들러가 수행되다 보니 사진으로는 인터럽트가 해당 모습을 전부 담기는 어렵지만 블록되어 인터럽트 핸들러가 수행되지 못하는 모습이다.

Nested Vectored Interrupt Controller (NVIC) ? X

Idx	Source	Name	E	P	A	Priority
47	I2C1 event interrupt	I2C1_EV	0	0	0	0
48	I2C1 error interrupt	I2C1_ER	0	0	0	0
49	I2C2 event interrupt	I2C2_EV	0	0	0	0
50	I2C2 error interrupt	I2C2_ER	0	0	0	0
51	SPI1 Global Interrupt	SPI1	0	0	0	0
52	SPI2 Global Interrupt	SPI2	0	0	0	0
53	USART1 Global Intr.	USART1	0	0	0	0
54	USART2 Global Intr.	USART2	0	0	0	0
55	USART3 Global Intr.	USART3	0	0	0	0
56	EXTI Line[15:10] Intr.	EXTI15_10	1	1	0	7
57	RTC Alarm thr. EXTI	RTCAlarm	0	0	0	0
58	USB Wakeup	USBWakeup	0	0	0	0

Selected Interrupt  
☒ Enable ☒ Pending ☐ Active Priority: 7

Interrupt Control & State  
 INT\_CTRL\_ST: 0x00438000 VECTACTIVE: 0x00  
☐ RETTOBASE VECTPENDING: 0x38  
☐ ISRPREEMPT ☒ ISRSPENDING

Application Interrupt & Reset Control  
 AIRC: 0xFA050000 PRIGROUP: 0: 7.1  
☐ VECTRESET ☐ SYSRESETREQ  
☐ VECTCLRACTIVE ☐ ENDIANESS

Vector Table Offset  
 VTO: 0x08000000 TBLOFF: 0x100000  
☐ TBLBASE

Software Interrupt Trigger  
 SW\_TRIG\_INT: 0x00000000 INTID: 0x00

그림 39 인터럽트가 block되어 pending 되어 있는 상태

## 5) Interrupt Clear Pending

```

35  ldr r0,=NVIC_ISER1
36  ldr r1,=0x00000100
37  str r1,[r0]
38
39  ldr r0,=GPIO_ODR
40  mov r5,#2_01001010
41  ldr r3,=sdata
42  str r5,[r3]
43  loop ldr r4,[r3]
44  str r4,[r0]
45  b loop
46  endp
47
48  EXTI15_10_IRQHandler proc
49  push {r4,r5}
50  ldr r4,=sdata
51  ldr r5,[r4]
52  add r5,#0x1
53  str r5,[r4]
54  ldr r4,=EXTI_PR
55  ldr r5,=0x00002000
56  str r5,[r4]
57  pop {r4,r5}
58  ldr r0,=NVIC_ICPR1
59  ldr r1,=0x00000100
60  str r1,[r0]
61  bx lr
62  endp
63  align
64  area lab7, data
65  sdata dcd 0x00
66  end

```

Nested Vectored Interrupt Controller (NVIC) ? X

Idx	Source	Name	E	P	A	Priority
47	I2C1 event interrupt	I2C1_EV	0	0	0	0
48	I2C1 error interrupt	I2C1_ER	0	0	0	0
49	I2C2 event interrupt	I2C2_EV	0	0	0	0
50	I2C2 error interrupt	I2C2_ER	0	0	0	0
51	SPI1 Global Interrupt	SPI1	0	0	0	0
52	SPI2 Global Interrupt	SPI2	0	0	0	0
53	USART1 Global Intr.	USART1	0	0	0	0
54	USART2 Global Intr.	USART2	0	0	0	0
55	USART3 Global Intr.	USART3	0	0	0	0
56	EXTI Line[15:10] Intr.	EXTI15_10	1	1	0	7
57	RTC Alarm thr. EXTI	RTCAlarm	0	0	0	0
58	USB Wakeup	USBWakeup	0	0	0	0

Selected Interrupt  
☒ Enable ☒ Pending ☒ Active Priority: 0

Interrupt Control & State  
 INT\_CTRL\_ST: 0x00438000 VECTACTIVE: 0x38  
☒ RETTOBASE VECTPENDING: 0x38  
☐ ISRPREEMPT ☒ ISRSPENDING

Application Interrupt & Reset Control  
 AIRC: 0xFA050000 PRIGROUP: 0: 7.1  
☐ VECTRESET ☐ SYSRESETREQ  
☐ VECTCLRACTIVE ☐ ENDIANESS

Vector Table Offset  
 VTO: 0x08000000 TBLOFF: 0x100000  
☐ TBLBASE

Software Interrupt Trigger  
 SW\_TRIG\_INT: 0x00000000 INTID: 0x00

Nested Vectored Interrupt Controller (NVIC) ? X

Idx	Source	Name	E	P	A	Priority
47	I2C1 event interrupt	I2C1_EV	0	0	0	0
48	I2C1 error interrupt	I2C1_ER	0	0	0	0
49	I2C2 event interrupt	I2C2_EV	0	0	0	0
50	I2C2 error interrupt	I2C2_ER	0	0	0	0
51	SPI1 Global Interrupt	SPI1	0	0	0	0
52	SPI2 Global Interrupt	SPI2	0	0	0	0
53	USART1 Global Intr.	USART1	0	0	0	0
54	USART2 Global Intr.	USART2	0	0	0	0
55	USART3 Global Intr.	USART3	0	0	0	0
56	EXTI Line[15:10] Intr.	EXTI15_10	1	1	0	7
57	RTC Alarm thr. EXTI	RTCAlarm	0	0	0	0
58	USB Wakeup	USBWakeup	0	0	0	0

Selected Interrupt  
☒ Enable ☐ Pending ☒ Active Priority: 0

Interrupt Control & State  
 INT\_CTRL\_ST: 0x00000838 VECTACTIVE: 0x38  
☒ RETTOBASE VECTPENDING: 0x00  
☐ ISRPREEMPT ☐ ISRSPENDING

Application Interrupt & Reset Control  
 AIRC: 0xFA050000 PRIGROUP: 0: 7.1  
☐ VECTRESET ☐ SYSRESETREQ  
☐ VECTCLRACTIVE ☐ ENDIANESS

Vector Table Offset  
 VTO: 0x08000000 TBLOFF: 0x100000  
☐ TBLBASE

Software Interrupt Trigger  
 SW\_TRIG\_INT: 0x00000000 INTID: 0x00

그림 40 인터럽트 pending clear

[그림 40]에서 확인할 수 있듯이 line 58-60에 새로운 코드를 추가하였다. 이 코드의 기능은 인터럽트 핸들러를 끝내기 전에 만약 인터럽트가 걸려있는 것이 있으면 해제를 하도록 하였다. [그림 40]의 좌측은 인터럽트 수행도중 인터럽트가 하나 더 pending된 상태이고 우측은 NVIC\_ICPR1에서 56번 인터럽트의 pending 상태를 clear하기 위해서 1을 set하였다. 그 결과 우측 그림처럼 인터럽트의 pending 상태가 해제된 것을 확인할 수 있다. 이를 이용하면 다른 인터럽트 핸들러에서 필요에 따라서 다른 인터럽트의 pending상태를 해제하는 등 다방면으로 활용할 수 있다.

## 6. 결론

이번 실험을 통해서 기존의 프로그래밍에서는 잘 다루지 않았던 인터럽트라는 것에 대해 공부할 수 있었다. 인터럽트가 발생하였을 때 이를 받고 인터럽트를 허용하는 방법부터 핸들러를 작성하고 수행시키는 부분까지 모든 부분을 어셈블리어로 직접 다루어 봄으로써 인터럽트를 더욱 자세히 이해할 수 있었다. 그리고 인터럽트와 관련된 여러가지 레지스터들도 사용해보고 매뉴얼도 읽어보고 하는 과정에서 매뉴얼을 보고 해당 기능을 이해하는 능력 또한 키울 수 있었다.

그리고 추가실험을 통해서 기본 실험에서 진행하였던 인터럽트의 기본적인 흐름에 더해 추가적으로 인터럽트와 관련된 여러 동작을 수행해보고 응용하여 수행해 봄으로써 인터럽트와 관련된 동작을 대부분 한 번씩 경험해볼 수 있었다.

## 7. 참고문헌

서강대학교 전자공학과 (2023). 마이크로프로세서개론.

STMicroelectronics (2017). STM32F10xxx/20xxx/21xxx/L1xxx Cortex-M3 Programming Manual (Rev 6)

STMicroelectronics (2021). STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced ARM-based 32-bit MUCs Reference Manual (Rev 21).

Yiu, J. (2010). The Definitive Guide To The ARM Cortex-M3 (2<sup>nd</sup> Edition)

Yiu, J. (2014). The Definitive Guide To The ARM Cortex-M3 and Cortex-M4 (3<sup>rd</sup> Edition)

히연. (2021). EMBEDDED RECIPES.