

마이크로프로세서응용실험 LAB08 결과보고서

General Purpose Inputs/Outputs (GPIOs)

20181536 엄석훈

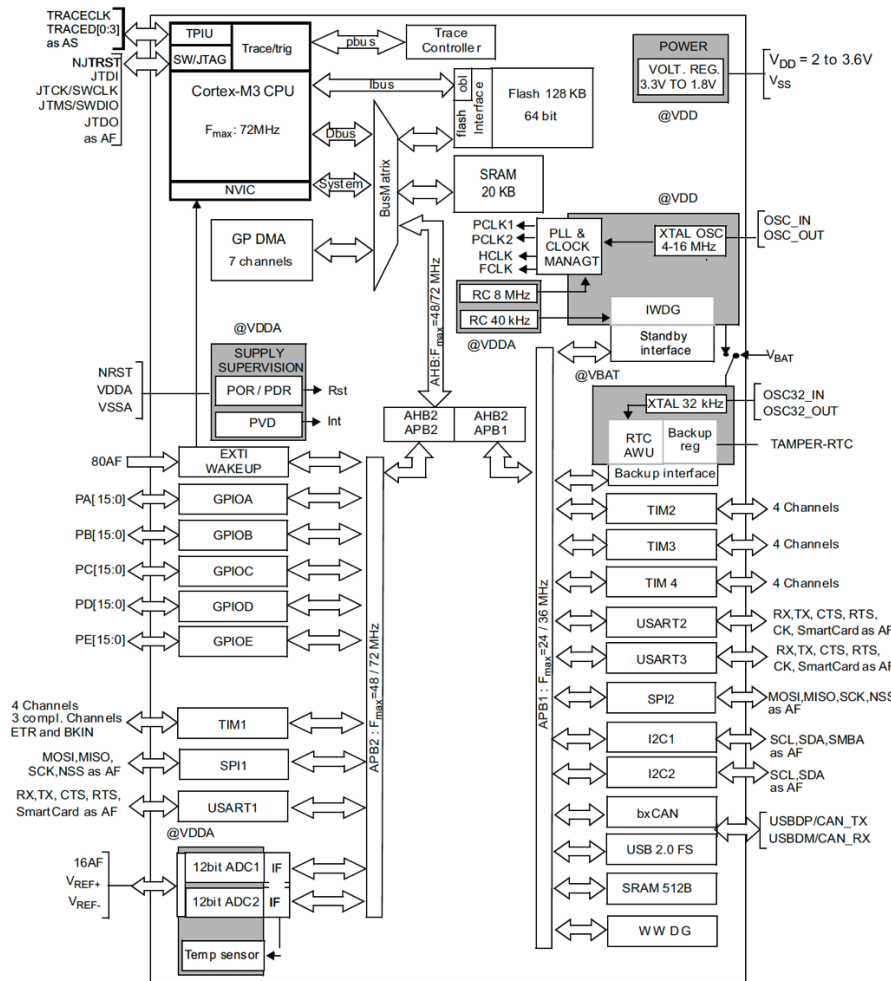
1. 목적

- GPIO의 동작모드 설정에 사용되는 레지스터들의 구성과 초기화 방법을 이해한다.
- GPIO의 일부 포트 신호들을 외부 소자/장치와 어떻게 연결하여 사용하는지 hardware/software 관점에서 이해한다.
- C program을 통한 포트와 외부 소자 access 방법을 익힌다.
- Array 형태로 배열된 소자들을 multiplexing을 통해 access하는 방법에 대해 이해하고 프로그램을 통해 구현한다. (dot matrix display, key matrix)

2. 이론

1) Peripherals in STM32F103x8/STM32F103xB

우리가 실험에서 사용하는 보드에 다양한 주변 장치를 연결할 수 있다. 이러한 주변장치들은 bus를 통해서 연결되어 있는데 아래 [그림 1]을 통해서 주변 장치가 어떤 버스를 통해서 CPU와 연결되어 있는지 확인할 수 있다. 우리가 이번 실험에서 사용할 GPIO는 APB2와 AHB를 통해서 CPU와 연결되어 있다. 그리고 [그림 2]를 통해서 GPIO를 포함한 일부 주변 장치의 주소와 그 범위를 확인할 수 있다.



ai14390d

그림 1 STM32F103xx block diagram

Boundary address	Peripheral	Bus
0x4001 5800 - 0x4001 7FFF	Reserved	
0x4001 5400 - 0x4001 57FF	TIM11 timer	
0x4001 5000 - 0x4001 53FF	TIM10 timer	
0x4001 4C00 - 0x4001 4FFF	TIM9 timer	
0x4001 4000 - 0x4001 4BFF	Reserved	
0x4001 3C00 - 0x4001 3FFF	ADC3	
0x4001 3800 - 0x4001 3BFF	USART1	
0x4001 3400 - 0x4001 37FF	TIM8 timer	
0x4001 3000 - 0x4001 33FF	SPI1	
0x4001 2C00 - 0x4001 2FFF	TIM1 timer	
0x4001 2800 - 0x4001 2BFF	ADC2	APB2
0x4001 2400 - 0x4001 27FF	ADC1	
0x4001 2000 - 0x4001 23FF	GPIO Port G	
0x4001 1C00 - 0x4001 1FFF	GPIO Port F	
0x4001 1800 - 0x4001 1BFF	GPIO Port E	
0x4001 1400 - 0x4001 17FF	GPIO Port D	
0x4001 1000 - 0x4001 13FF	GPIO Port C	
0x4001 0C00 - 0x4001 0FFF	GPIO Port B	
0x4001 0800 - 0x4001 0BFF	GPIO Port A	
0x4001 0400 - 0x4001 07FF	EXTI	
0x4001 0000 - 0x4001 03FF	AFIO	

그림 2 APB2에 연결된 주변장치의 address

2) GPIO functional description

우리가 사용하는 보드에서는 GPIOA 부터 GPIOE까지 5개의 포트와 각 포트는 16개의 신호로 구성된다. 따라서 총 80개의 신호를 사용할 수 있으며 각 핀은 input floating, input pull-up, input pull-down, analog, output open-drain, output push-pull, alternate function push-pull, alternate function open-drain으로 상요할 수 있다. 그리고 GPIO에 속한 핀은 설정을 통해서 다양한 방법으로 입력 또는 출력을 할 수 있다. 또한 각 핀마다 설정하여 GPIO를 사용할 수 있도록 각 포트마다 설정을 위한 GPIOx_CRL, GPIOx_CRH, 데이터를 위한 GPIOx_IDR, GPIOx_ODR, set/reset을 위한 GPIOx_BSRR, reset을 위한 GPIOx_BRR, lock을 위한 GPIOx_LCKR가 존재한다.

기본적으로 초기화 되는 도중이나 직후는 AFIO는 연결되지 않으며 input floating mode로 초기화된다. 또한 출력모드일때 GPIOx_ODR의 값이 출력이 되고 push-pull, open-drain 모드를 선택할 수 있다. 또한 GPIOx_IDR은 매 APB2 클럭 사이클마다 입출력으로 나타나는 신호를 저장한다. 이때 핀의 설정은 아래 [그림 3]에 따라서 결정된다.

Configuration mode		CNF1	CNF0	MODE1	MODE0	PxODR register	
General purpose output	Push-pull	0	0	01 10 11 see <i>Table 21</i>		0 or 1	
	Open-drain		1			0 or 1	
Alternate Function output	Push-pull	1	0				Don't care
	Open-drain		1				Don't care
Input	Analog	0	0	00		Don't care	
	Input floating		1			Don't care	
	Input pull-down	1	0			0	
	Input pull-up					1	

Table 21. Output MODE bits

MODE[1:0]	Meaning
00	Reserved
01	Maximum output speed 10 MHz
10	Maximum output speed 2 MHz
11	Maximum output speed 50 MHz

그림 3 Port pin configuration

다음으로 GPIO의 핀을 신호연결을 목적으로 사용하고자 할 때 이를 AF(alternate function)라고 부르고 GPIOx_CRL, GPIOx_CRH를 통해서 AF로 사용하도록 설정할 수 있다. 이때 AF input으로 사용하려면 핀을 input모드로 설정하고 입력신호를 외부에서 연결하여야 한다. 또한 AF output으로 사용하려면 핀을 alternate function output모드로 설정해야 하고 push-pull, open-drain 중 선택해야 한다. 마지막으로 AF bidirectional로 사용하려면 핀을 alternate function output모드로 설정해야 하고 이때 입력버퍼는 input floating 모드로 동작한다.

다음으로 GPIO의 동작을 고정해두기 위해서는 locking을 사용할 수 있다. GPIOx_LCKR 레지스터의 원하는 핀의 bit에 1을 set하게 되면 다음 reset때 까지 해당 핀의 값을 수정할 수 없게 된다. 또한 GPIO의 데이터를 변경할 때 중간에 인터럽트가 발생하는 것을 방지하기 위해서 atomic하게 데이터를 접근해야 하는 경우가 있다. 이때는 GPIOx_BSRR 레지스터를 사용해서 1을 쓰면 GPIOx_ODR에 데이터가 set (또는 reset)되게 된다. 또는 GPIOx_BRR 레지스터에 1을 쓰면 reset이 되게 된다.

그리고 [그림 3]의 설정에 따라 일부 동작에 대한 설명은 다음과 같다. 가장 먼저 input configuration의 경우 [그림 4]와 같이 회로가 구성된다. 특징으로는 output buffer가 사용되지 않고, Schmitt trigger input이 활성화되고 설정에 따라 pull-up, pull-down 저항이 연결되고, APB2 클럭 사이클마다 핀의 데이터가 저장되며 input data 레지스터를 통해서 입력을 읽을 수 있다.

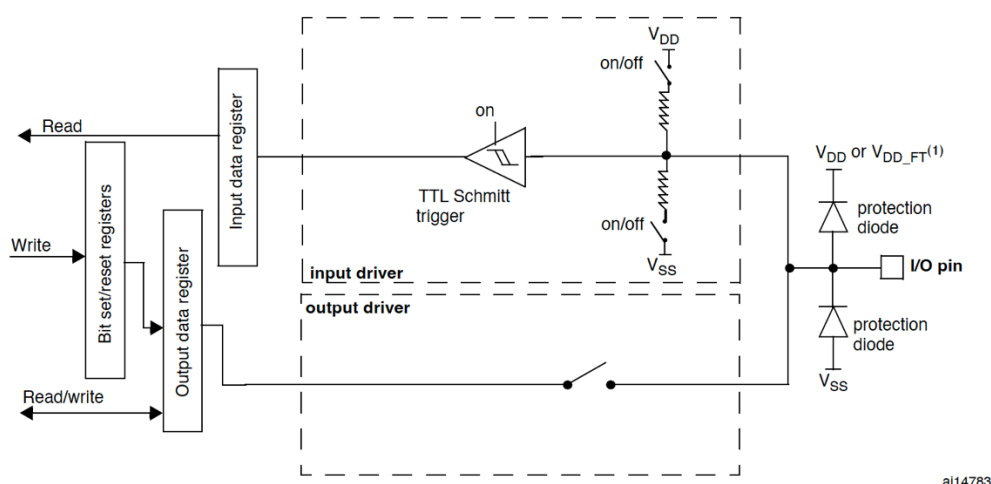
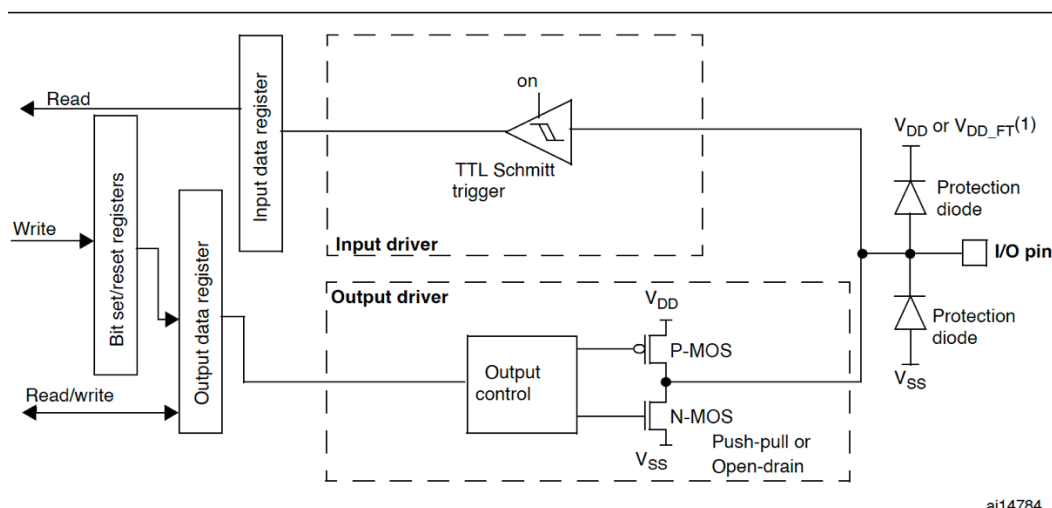


그림 4 Input configuration에서의 회로

다음으로 output configuration에서의 회로는 [그림 5]와 같다. 특징으로는 output buffer가 활성화 되고 Schmitt trigger input이 활성화되고 pull-up, pull-down 저항이 연결되지 않고, APB2 클럭 사

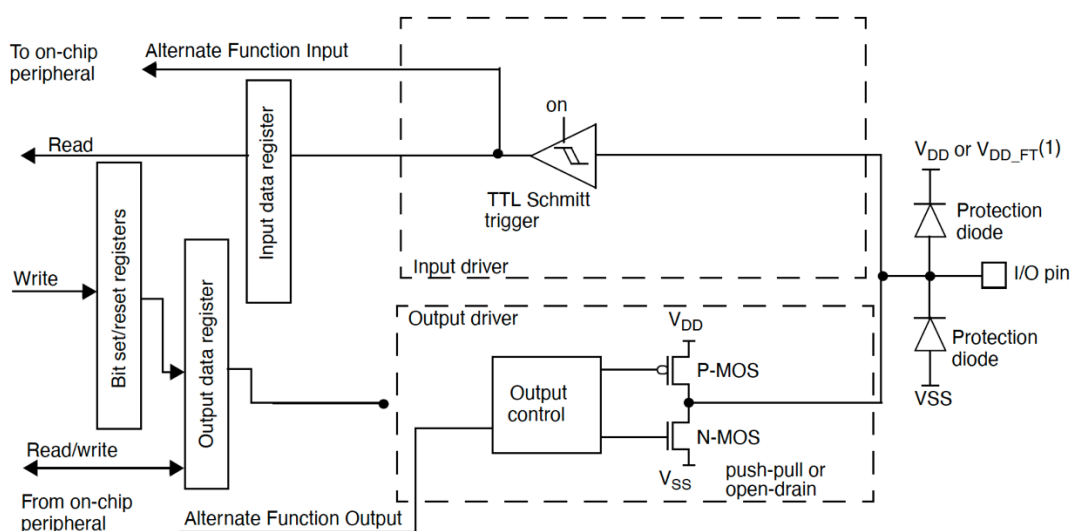
이클마다 핀의 신호가 input data 레지스터에 저장되고 open-drain 모드의 경우 input data 레지스터에서 데이터를 읽을 수 있고 push-pull 모드의 경우는 output data 레지스터에서 데이터를 읽을 수 있다.



ai14784

그림 5 Output configuration에서의 회로

다음으로 alternate function configuration에서의 회로는 [그림 6]과 같다. 특징으로는 output buffer가 open-drain 또는 push-pull 모드에서 켜지고 주변장치의 출력이 output buffer에 연결되고 Schmitt trigger input이 활성화 되고 pull-up, pull-down 저항이 연결되지 않고, APB2 클럭 사이클마다 I/O핀의 데이터가 input data 레지스터에 저장되고 open-drain 모드의 경우는 input data 레지스터에서 데이터를 읽을 수 있고 push-pull 모드의 경우는 output data 레지스터에서 데이터를 읽을 수 있다.



ai14785

그림 6 alternate function configuration에서의 회로

마지막으로 analog configuration에서의 회로는 [그림 7]과 같다. 특징으로는 output buffer가 사용되지 않고 Schmitt trigger input이 비활성화되고 Schmitt trigger output이 0으로 고정되며 pull-up, pull-down 저항이 연결되지 않고 input data 레지스터에서는 0이 읽혀진다.

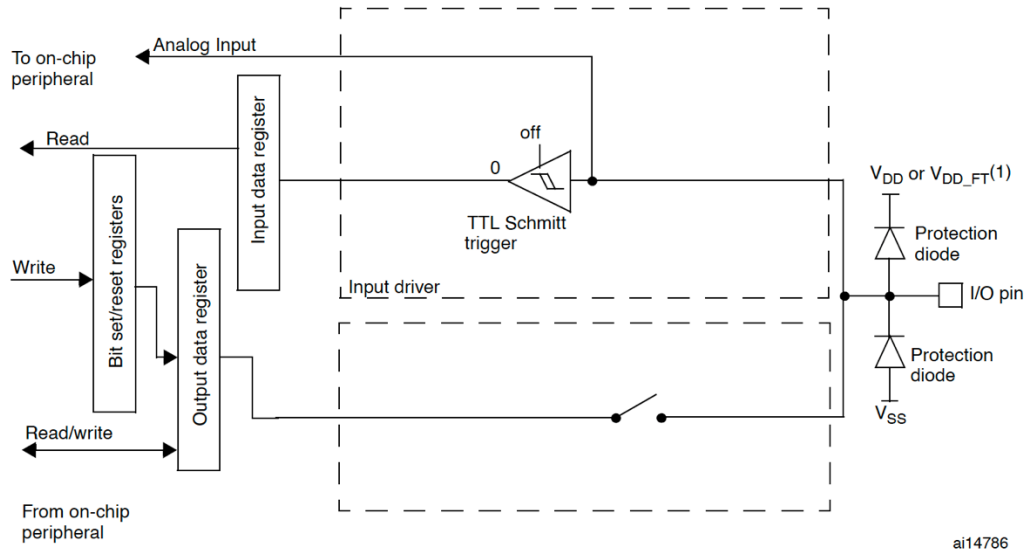


그림 7 analog configuration에서의 회로

3) GPIO registers

GPIO의 모드와 데이터를 위한 레지스터의 구성과 목록, 초기값은 아래 [그림 8]과 같다.

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	GPIOx_CRL	CNF 7 [1:0]	MODE 7 [1:0]	CNF 6 [1:0]	MODE 6 [1:0]	CNF 5 [1:0]	MODE 5 [1:0]	CNF 4 [1:0]	MODE 4 [1:0]	CNF 3 [1:0]	MODE 3 [1:0]	CNF 2 [1:0]	MODE 2 [1:0]	CNF 1 [1:0]	MODE 1 [1:0]	CNF 0 [1:0]	MODE 0 [1:0]																
	Reset value	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	0	0	1	0	0
0x04	GPIOx_CRH	CNF 15 [1:0]	MODE 15 [1:0]	CNF 14 [1:0]	MODE 14 [1:0]	CNF 13 [1:0]	MODE 13 [1:0]	CNF 12 [1:0]	MODE 12 [1:0]	CNF 11 [1:0]	MODE 11 [1:0]	CNF 10 [1:0]	MODE 10 [1:0]	CNF 9 [1:0]	MODE 9 [1:0]	CNF 8 [1:0]	MODE 8 [1:0]																
	Reset value	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	0	0	1	0	0
0x08	GPIOx_IDR	Reserved																IDRy															
	Reset value																	0															
0x0C	GPIOx_ODR	Reserved																ODRy															
	Reset value																	0															
0x10	GPIOx_BSRR	BR[15:0]																BSR[15:0]															
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0x14	GPIOx_BRR	Reserved																BR[15:0]															
	Reset value																	0															
0x18	GPIOx_LCKR	Reserved																LCKK	LCK[15:0]														
	Reset value																	0	0														

그림 8 GPIO 관련 레지스터 구성

4) Alternate function I/O (AFIO)

동일한 프로세서더라도 사용 목적에 따라서 pin의 수에 변화를 주는 등 다양한 패키지가 존재한다. 우리가 사용하는 보드는 64pin을 가지고 있는데 위에서 설명한 대로 GPIO핀만 해도 80개가 필요하다. 따라서 하나의 핀을 여러개의 기능을 할 수 있도록 핀에 대해서 alternate function을 수행할 수 있도록 해준다. 기본적으로 주어진 default 기능과 alternate function 그리고 이에 추가적으로 remapping도 가능하게 칩을 설계하였다. 이러한 각 핀에 대한 AFIO의 배치 예시 다음 [그림 9]와 같다.

Pins		Pin	Main	Alternate function	
100	64	Name	Function	Default	Remap
1	-	PE2	PE2	TRACECK	
2	-	PE3	PE3	TRACED0	
3	-	PE4	PE4	TRACED1	
4	-	PE5	PE5	TRACED2	
5	-	PE6	PE6	TRACED3	
6	1	VBAT	VBAT		
7	2	PC13-TAMPERRTC	PC13	TAMPER-RTC	
8	3	PC14-OSC32_IN	PC14	OSC32_IN	
9	4	PC15-OSC32_OUT	PC15	OSC32_OUT	
10	-	VSS_5	VSS_5		
11	-	VDD_5	VDD_5		
12	5	OSC_IN	OSC_IN		
13	6	OSC_OUT	OSC_OUT		
14	7	NRST	NRST		
15	8	PC0	PC0	ADC12_IN10	
16	9	PC1	PC1	ADC12_IN11	
17	10	PC2	PC2	ADC12_IN12	
18	11	PC3	PC3	ADC12_IN13	
19	12	VSSA	VSSA		
20	-	VREF-	VREF-		
21	-	VREF+	VREF+		
22	13	VDDA	VDDA		
23	14	PA0-WKUP	PA0	WKUP/ USART2_CTS/ ADC12_IN0/ TIM2_CH1_ETR	
24	15	PA1	PA1	USART2_RTS/ ADC12_IN1/ TIM2_CH2	
25	16	PA2	PA2	USART2_TX/ ADC12_IN2/ TIM2_CH3	
26	17	PA3	PA3	USART2_RX/ ADC12_IN3/ TIM2_CH4	

그림 9 일부 pin에 대한 AFIO 배치

그리고 아래 [그림 10]은 AFIO 사용을 위해 필요한 레지스터와 그 초기값이다. AFIO_EVCR, AFIO_MAPR 레지스터를 통해서 어떤 AFIO를 사용할지에 대해 설정하고 remapping을 설정할 수 있으며 AFIO_EXTICR 레지스터를 통해서 핀과 EXTI를 연결해주는 역할을 한다.

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0x00	AFIO_EVCR	Reserved																										EVOE	PORT[2:0]			PIN[3:0]		
	Reset value																											0	0	0	0	0	0	
0x04	AFIO_MAPR low-, medium-, high- and XL- density devices	Reserved				SWJ_CFG[2]	SWJ_CFG[1]	SWJ_CFG[0]	Reserved				ADC2_ETRGREG_REMAP	ADC2_ETRGINJ_REMAP	ADC1_ETRGREG_REMAP	ADC1_ETRGINJ_REMAP	TIM5CH4_REMAP	PD01_REMAP	CAN1_REMAP[1]	CAN1_REMAP[0]	TIM4_REMAP	TIM3_REMAP[1]	TIM3_REMAP[0]	TIM2_REMAP[1]	TIM2_REMAP[0]	TIM1_REMAP[1]	TIM1_REMAP[0]	USART3_REMAP[1]	USART3_REMAP[0]	USART2_REMAP	USART1_REMAP	I2C1_REMAP	SPI1_REMAP	
	Reset value												0	0	0															1	0			
0x08	AFIO_EXTICR1	Reserved												EXTI3[3:0]			EXTI2[3:0]			EXTI1[3:0]			EXTI0[3:0]											
	Reset value													0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0C	AFIO_EXTICR2	Reserved												EXTI7[3:0]			EXTI6[3:0]			EXTI5[3:0]			EXTI4[3:0]											
	Reset value													0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x10	AFIO_EXTICR3	Reserved												EXTI11[3:0]			EXTI10[3:0]			EXTI9[3:0]			EXTI8[3:0]											
	Reset value													0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14	AFIO_EXTICR4	Reserved												EXTI15[3:0]			EXTI14[3:0]			EXTI13[3:0]			EXTI12[3:0]											
	Reset value													0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

그림 10 AFIO 관련 레지스터

3. 실험 과정

1) 실험 1

STEP 1: Program 8.1은 NUCLEO-F103RB board에 장착된 스위치와 LED를 대상으로 한다. 스위치와 LED는 각각 PC13과 PA5에 연결되었다.

Program 8.1을 이용하여 프로젝트를 생성한다.


```

1  #include <stm32f10x.h>
2
3  static char sw;
4
5  int main(void) {
6
7      RCC->APB2ENR = 0x00000015;
8      GPIOC->CRH = 0x00800000;
9      GPIOA->CRL = 0x00300000;
10
11     EXTI->FTSR = 0x2000;
12     EXTI->IMR = 0x2000;
13     AFIO->EXTICR[3] = 0x20;
14     NVIC->ISER[1] = (1<<8);
15
16     GPIOA->ODR = 0x0000;
17     sw = 0;
18
19     while (1) { }
20
21 } /*end main*/
22
23 void EXTI15_10_IRQHandler(void) {
24     if(EXTI->PR == 0x2000) {
25         if(sw) {
26             GPIOA->ODR = 0x0;
27             sw = 0;
28         }
29         else {
30             GPIOA->ODR = 0x20;
31             sw = 1;
32         }
33         EXTI->PR = 0x2000;
34     }
35 }

```

그림 11 직접 작성한 lab8_1.c 코드

STEP 2: Lines 8-9는 GPIO의 configuration과정을 보여준다. 이처럼 초기화하는 이유를 스위치와 LED가 연결된 port와 연관지어 설명하시오.

Line8의 GPIOC->CRH = 0x00800000; 코드를 통해서 GPIOC의 configuration을 설정할 수 있다. GPIOC의 13번 포트를 input pull-down으로 설정하였다. 다음으로 line 9의 GPIOA->CRL = 0x00300000; 코드를 통해서 GPIOA의 5번포트를 push-pull output으로 설정하였다. GPIOC의 13번 포트는 버튼에 연결되어 있으므로 input으로 연결하고 GPIOA의 5번 포트는 LED에 연결할 것이기 때문에 output으로 연결하였다.

STEP 3: Lines 11-14는 무엇을 위함인지 7장에서 배운 내용을 근거로 설명하시오.

Lines 11-14를 하나하나 분석해보면 다음과 같다. 먼저 EXTI->FTSR = 0x2000; 은 EXTI13이 falling trigger edge에서 이벤트를 감지하도록 설정한 것이다. 다음으로 EXTI->IMR = 0x2000; 은 EXTI13을 사용하도록 설정한 것이다. 다음으로 AFIO->EXTICR[3] = 0x20; 은 EXTICR4에 0x20을 저장한 것으로 EXTI13을 GPIOC의 pin13의 input과 연결을 해준다. 마지막으로 NVIC->ISER[1] = (1<<8); 은 56번 인터럽트인 EXTI Line[15:10]을 enable시켜서 해당 인터럽트가 발생할 수 있도록 해준다.

STEP 4: EXTI15_10_IRQHandler에서 수행되는 동작에 대해 설명하시오.

Lines 26, 30은 각각 어떤 동작을 구현한 것인가?

Line 33은 무엇을 위함인가?

인터럽트 핸들러에서는 EXTI13이 pending되어 있는지 확인하고 sw가 1이라면 0으로 바꾸고 GPIOA->ODR에 0을 넣어서 LED를 끈다. 그리고 sw가 0이라면 1로 바꾸고 GPIOA->ODR에 0x20을 넣어서 LED를 켜다. 즉 LED가 켜져있다면 끄고 꺼져있다면 켜는 LED를 토글하는 동작을 한다. Line 26은 GPIOA->ODR에 0을 넣어서 GPIOA의 5번핀에도 0이 들어있기 때문에 LED가 꺼지게 된다. 다음으로 line 30은 GPIOA->ODR에 0x20을 넣어서 5번핀에만 1을 넣어 LED가 켜지도록 하였다. 마지막으로 line 33의 EXTI->PR = 0x2000을 통해서 EXTI13의 펜딩된 인터럽트를 clear시켜준다.

STEP 5: Program을 borad에서 수행하면서 파란색 스위치를 누를 때마다 녹색 LED에 어떤 변화가 일어나는지 확인해본다. 관찰한 내용을 이전 과정에서 해석한 내용과 비교하며 설명해보자.

파란색 스위치를 누를 때 마다 아래 [그림 12]처럼 녹색 LED가 켜졌다 꺼졌다 상태를 바꾸게 된다. 즉 파란색 버튼이 눌러 GPIOC의 13핀에 입력이 들어가 인터럽트가 발생하고 GPIOA의 5번핀의 LED의 값이 토글되어 꺼졌다 켜졌다 동작하게 된다. 이를 통해서 이전 과정에서 해석한 결과와 동일하게 동작하는 것을 알 수 있다.

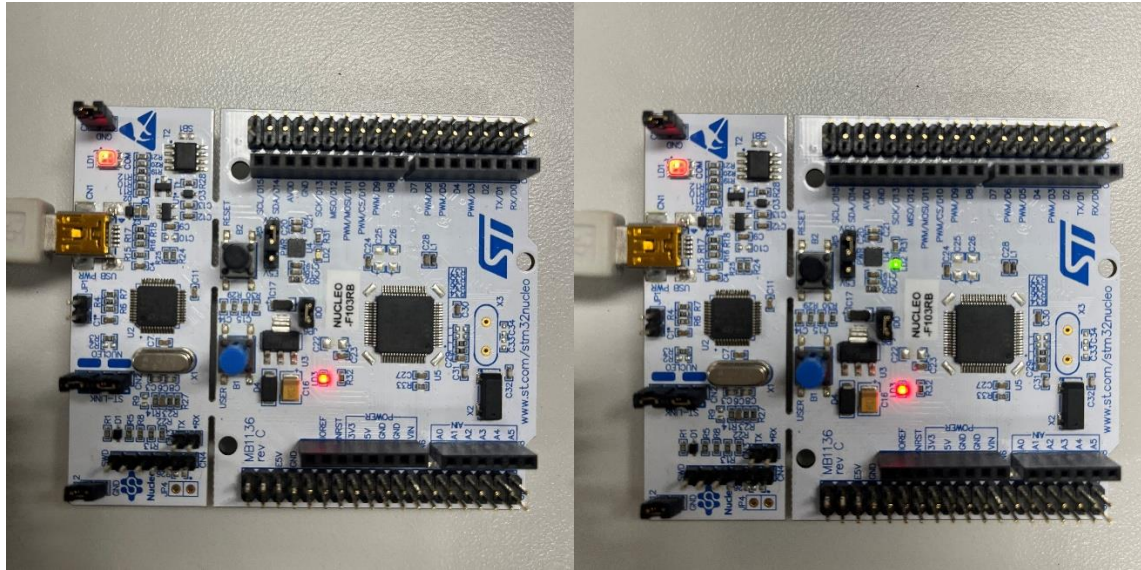


그림 12 LED의 변화

STEP 6: Line 19에서 CPU는 어떤 동작을 수행하는가? 이 부분을 `while (1) { _wfi(); }`로 교체하면 CPU는 어떤 상태에서 어떤 동작을 수행하는지, 그리고 이와 같이 처리하면 어떤 이점이 있는지 알아보자.

Line 19에서 CPU는 계속해서 현재의 명령어로 branch하는 동작을 수행한다. 따라서 보기에는 프로세서에서 아무 변화가 없지만 계속해서 동일한 branch명령어를 수행하고 있다. 이때 이 부분을 `while(1) { _WFI(); }`로 교체하면 CPU는 인터럽트가 발생할 때 까지 sleep mode 상태에서 인터럽트는 wait하고 프로세서는 어떠한 동작도 수행하지 않는다. 이와 같이 처리하면 프로세서가 아무런 동작도 하지 않기 때문에 불필요한 명령어를 수행하지 않아도 되기 때문에 전력효율이 좋아진다.

2) 실험 2

STEP 7: 그림 8.7은 별도로 제공된 8 x 8 dot matrix display의 구성을 보여준다. 그림의 caption을 통해 핀번호와 내부 연결관계를 “반드시” 확인한다. 또한 이러한 dot matrix display에 임의의 패턴을 표시하기 위해 software적으로 어떤 고려가 필요한지 생각해보자.

Dot matrix를 이용해서 임의의 패턴을 표시하기 위해서는 row 1부터 8까지 하나씩 선택해서 해당 row에 LED를 켜고 싶은 column에 1을 입력해서 켜주어야 한다. 즉 임의의 패턴을 표시하기 위해서는 각 row별로 어떤 column을 켜야할지 배열의 형태로 저장해두고 배열을 참조하는 방식으로 원하는 임의의 패턴을 그릴 수 있다.

STEP 8: NUCLEO-F103RB의 GPIO port 신호와 dot matrix를 별도로 제공된 jump wire를 이용하여 다음 표와 같이 연결한다.

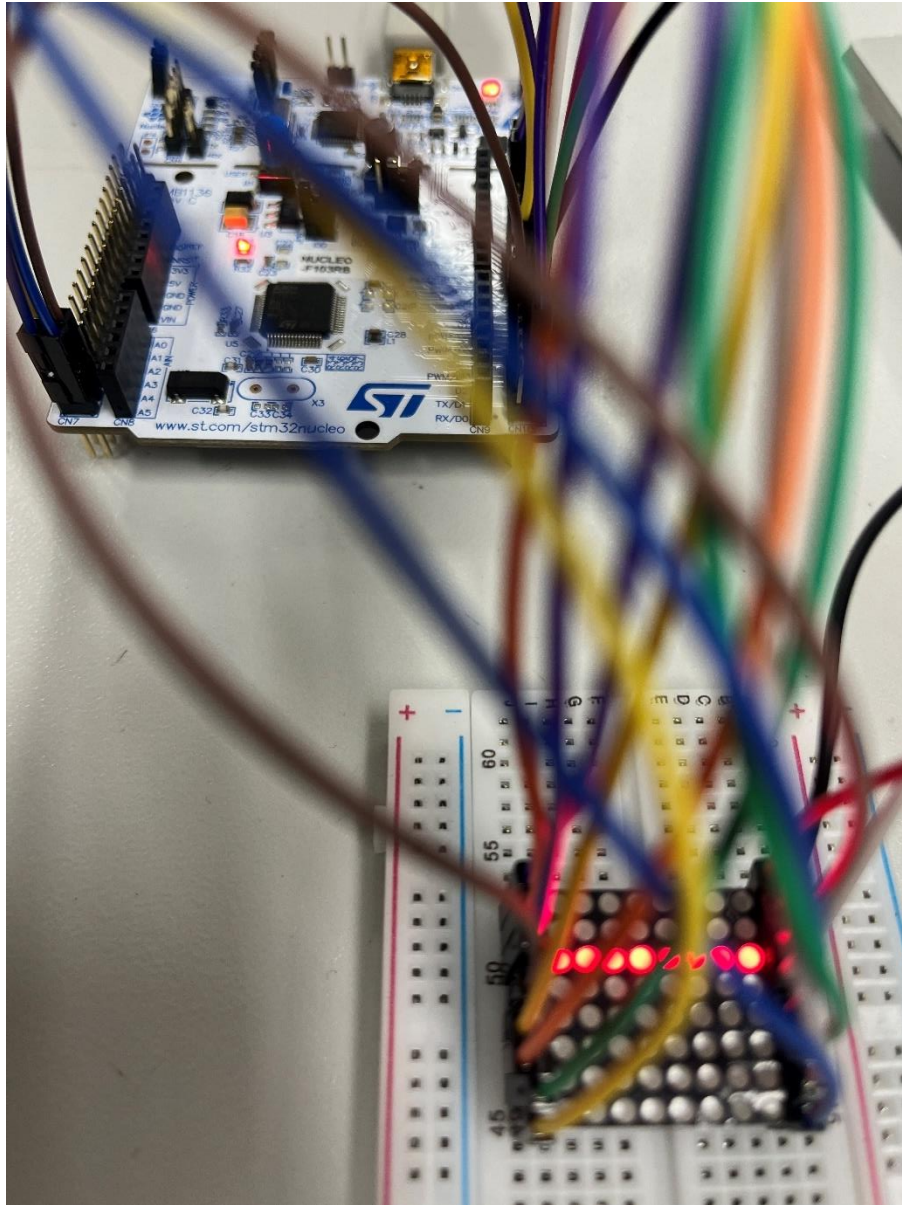


그림 13 직접 연결한 lab8_2 보드

STEP 9: Program 8.2를 이용하여 프로젝트를 생성한다.

```

1  #include <stm32f10x.h>
2
3  u32 i, row, col;
4
5  int main(void) {
6
7      RCC->APB2ENR = 0x0000001D;
8      GPIOC->CRL = 0x33333333;
9      GPIOB->CRH = 0x33333333;
10
11     row = 0x0000;
12     col = 0x0001;
13
14     while(1) {
15         GPIOC->ODR = row;
16         GPIOB->ODR = col << 8;
17
18         for(i = 0; i < 300000; i++){;}
19
20         col = col << 1;
21         if(col == 0x0100) col = 0x0001;
22     }
23 }

```

그림 14 직접 작성한 lab8_2.c 코드

STEP 10: Lines 7-9에서 수행되는 초기화 과정의 내용을 위 step들에서 파악한 내용과 연관지어 확인해 본다.

Line 7-9에서 수행되는 명령어를 해석하면 다음과 같다. 먼저 line 7의 `RCC->APB2ENR = 0x0000001D;` 는 APB2를 통해서 주변장치가 연결될 수 있도록 허용해주는 동작을 수행하는데 GPIOA, GPIOB, GPIOC, AFIO를 APB2를 이용해서 데이터가 이동할 수 있도록 enable해준다. 다음으로 line 8의 `GPIOC->CRL = 0x33333333;` 은 GPIOC의 포트0부터 포트7을 push-pull output으로 사용할 수 있도록 해준다. 마지막으로 `GPIOB->CRH = 0x33333333;` 도 동일하게 GPIOB의 포트8부터 포트15까지 push-pull output으로 사용할 수 있도록 해준다. 즉 dot matrix의 row에 연결된 PC0 – PC7와 column에 연결된 PB8 – PB15까지 모두 push-pull output으로 사용할 수 있도록 초기화해 주었다.

STEP 11: Lines 14-22에서 수행되는 무한 반복 동작의 내용을 파악한다.

- 이와 같이 무한 반복하는 이유는 무엇인가?
- Lines 15-16는 무엇을 위함인가?

- **Line 18의 역할은 무엇인가? Line 18에서 300000을 증가 또는 감소시키면 dot matrix에서 어떤 변화가 있을 것으로 예상되는가? 변경하면서 확인해 본다.**

Lines 14-22을 무한 반복하는 이유는 계속해서 dot matrix에 반복적으로 출력을 해주기 위해서 lines 14-22를 무한 반복한다. 그리고 line 15의 `GPIOC->ODR = row;`를 통해서 PC0 – PC8까지 모두 0으로 reset해준다. 그리고 line 16의 `GPIOB->ODR = col << 8;`을 통해서 PB8 – PB15에 while 문을 돌면서 순서대로 PB8, PB9 ... PB15까지 하나씩 1로 set해준다. 즉 하나씩 column을 증가시키면서 모든 row에 0으로 해서 해당 column의 모든 row에 해당하는 LED를 켜준다. 마지막으로 line 18의 300000을 증가시키거나 감소시키면 dot matrix에서 막대가 움직이는 속도가 감소하거나 증가한다. 즉 300000보다 큰 값을 넣으면 for문을 수행하는데 소요되는 시간이 증가하고 따라서 하나의 막대를 더 오랫동안 보여주어 막대가 움직이는 속도가 감소한다. 반대로 300000보다 작은 값을 넣으면 for문을 수행하는데 소요되는 시간이 감소하고 따라서 하나의 막대를 더 짧게 보여주어 막대가 움직이는 속도가 증가한다. 그리고 코드를 수정하여 실행한 결과 막대의 사진에서 보기는 어렵지만 [그림 15]처럼 속도가 바뀐 것을 확인할 수 있다.

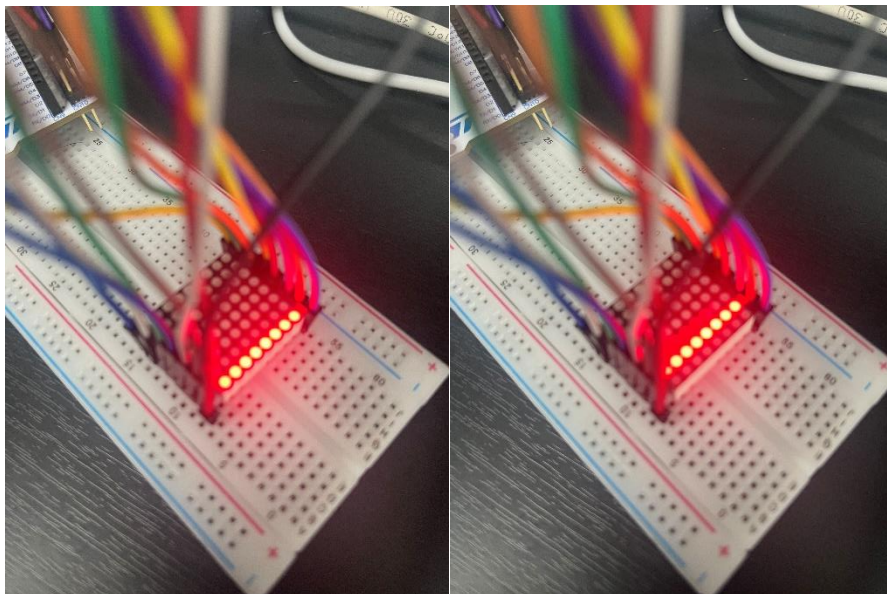


그림 15 LED의 이동 사진

STEP 12: 프로그램을 수행할 때 움직이는 막대의 방향을 반대로 변경하기 위해서는 프로그램의 어떻게 수정해야 하는가? 막대가 상하로 움직이도록 프로그램을 변경해본다. 또한, 막대의 두께를 변경하기 위해서는 프로그램의 어떤 부분을 수정해야 하겠는가?

막대의 방향을 반대로 하기 위해서 수정한 코드는 [그림 16]과 같다. 원래의 코드와 다르게 column을 2b10000000인 0x0080부터 시작해서 오른쪽으로 shift하면서 0x0000이 되면 다시 0x0080으로 업데이트해준다. Column의 이동 방향만 바뀌서 간단하게 막대의 이동 방향을 바꿀 수 있다. 이에 대해 [그림 17]과 같이 확인할 수 있다.

```
1  #include <stm32f10x.h>
2
3  u32 i, row, col;
4
5  int main(void) {
6
7      RCC->APB2ENR = 0x0000001D;
8      GPIOC->CRL = 0x33333333;
9      GPIOB->CRH = 0x33333333;
10
11     row = 0x0000;
12     col = 0x0080;
13
14     while(1) {
15         GPIOC->ODR = row;
16         GPIOB->ODR = col << 8;
17
18         for(i = 0; i < 300000; i++){;}
19
20         col = col >> 1;
21         if(col == 0x0000) col = 0x0080;
22     }
23 }
```

그림 16 막대의 방향 반대로 하기 위한 코드

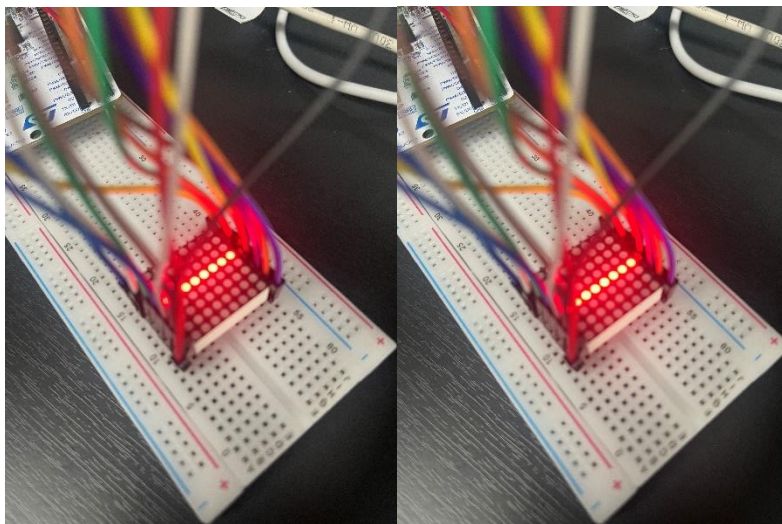


그림 17 반대로 움직이는 LED

다음으로 아래 [그림 18]과 같이 코드를 수정해서 막대가 상하로 움직이도록 코드를 작성할 수 있다. 그리고 이에 대한 동작은 [그림 19]를 통해서 확인 가능하다. 모든 특정 row에서 모든 column에 대해서 LED를 켜야하기 때문에 column에는 0x00FF를 저장하고 row를 0x0001부터

0x0080까지 shift하면서 반복하도록 하였다. 이때 row를 0으로 해야 해당 row를 감지하기 때문에 GPIOC->ODR = ~row를 통해서 하나의 column만 0이 되도록 해주었다.

```
1  #include <stm32f10x.h>
2
3  u32 i, row, col;
4
5  int main(void) {
6
7      RCC->APB2ENR = 0x0000001D;
8      GPIOC->CRL = 0x33333333;
9      GPIOB->CRH = 0x33333333;
10
11     row = 0x0001;
12     col = 0x00FF;
13
14     while(1) {
15         GPIOC->ODR = ~row;
16         GPIOB->ODR = col << 8;
17
18         for(i = 0; i < 300000; i++){;}
19
20         row = row << 1;
21         if(row == 0x0100) row = 0x0001;
22     }
23 }
```

그림 18 막대의 방향을 상하로 하기 위한 코드

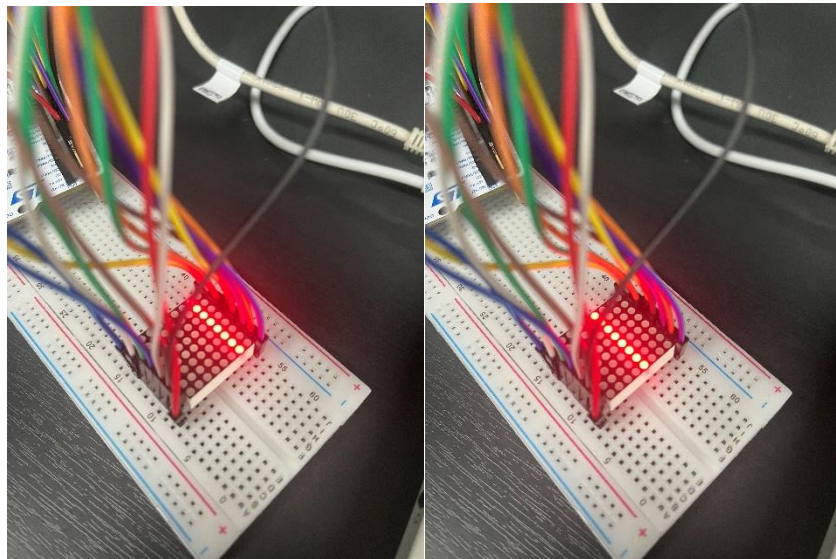


그림 19 상하로 움직이는 LED

마지막으로 막대의 두께를 변경하기 위해서는 [그림 20]과 같이 코드를 작성하였다. 이에 대한 동작은 [그림 21]을 통해서 확인 가능하다. 간단히 col을 0x0001에서 0x0003으로 바꿔서 2개의 column이 1이 되도록 해주었다. 그리고 2씩 shift하면서 반복하도록 코드를 작성하였다. 그리고 원한다면 1씩 shift하면서 겹치는 부분이 있도록 LED가 이동하는 효과를 만들 수 있다.


```

1  #include <stm32f10x.h>
2
3  u32 i, row, col;
4
5  int main(void) {
6
7      RCC->APB2ENR = 0x0000001D;
8      GPIOC->CRL = 0x33333333;
9      GPIOB->CRH = 0x33333333;
10
11     row = 0x0000;
12     col = 0x0003;
13
14     while(1) {
15         GPIOC->ODR = row;
16         GPIOB->ODR = col << 8;
17
18         for(i = 0; i < 300000; i++){;}
19
20         col = col << 2;
21         if(col == 0x0300) col = 0x0003;
22     }
23 }

```

그림 20 막대의 두께를 바꾼 코드

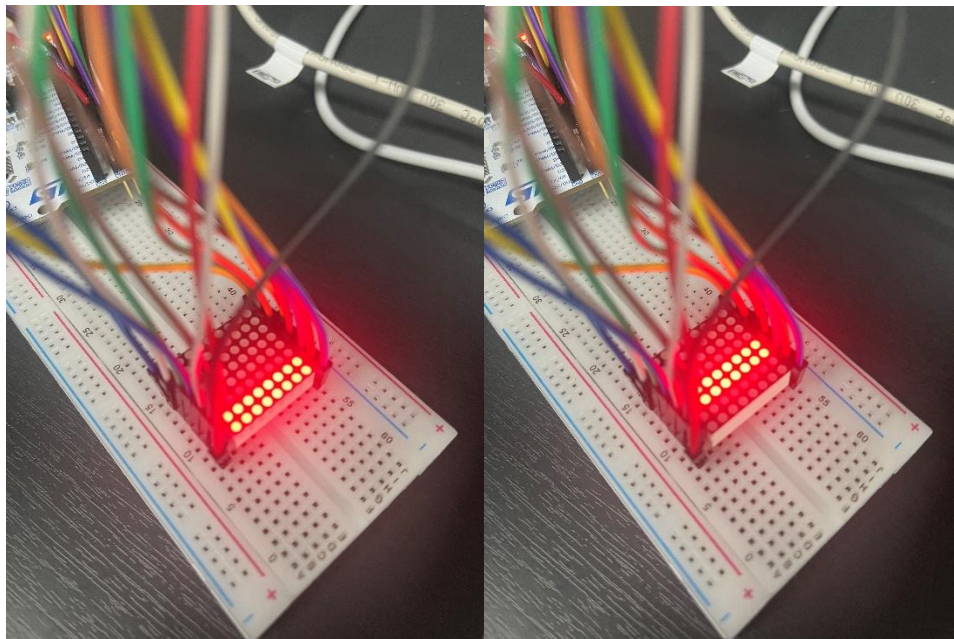


그림 21 막대의 두께를 바꾼 LED

STEP 13: Dot matrix에 임의의 패턴(예를 들어, 삼각형 모양)을 표시하기 위한 프로그램을 구현해 보자.

```

1  #include <stm32f10x.h>
2
3  u32 i, j, row, col;
4  u8 heart[8] = {0x00, 0x66, 0xff, 0xff, 0x7e, 0x3c, 0x18, 0x00};
5
6  int main(void) {
7
8      RCC->APB2ENR = 0x0000001D;
9      GPIOC->CRL = 0x33333333;
10     GPIOB->CRH = 0x33333333;
11
12     row = 0x0001;
13     col = 0x0000;
14     j = 0;
15
16     while(1) {
17         GPIOC->ODR = ~row;
18         row = row << 1;
19         col = heart[j++];
20         GPIOB->ODR = col << 8;
21         if(row == 0x0100) {row = 0x0001; j = 0; }
22     }
23 }

```

그림 22 하트 모양 LED를 위한 코드

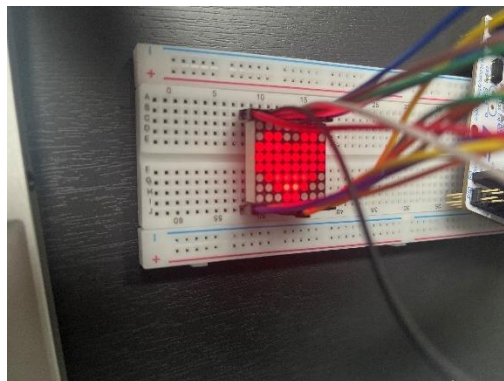


그림 23 하트 모양 LED

위 [그림 22]와 같이 코드를 작성하여 [그림 23]과 같이 하트 모양으로 LED를 켜주었다. 이를 위해서 하트 모양을 만들기 위해 heart라는 배열을 만들어 각 row별로 켜줘야 할 column에 대한 정보를 미리 저장해두었다. 그리고 row를 하나씩 증가시켜 가면서 각 row에 켜줘야 할 column정보를 heart배열에서 col로 가져와서 LED를 켜주었다. 그리고 row를 0x01부터 0x80까지 반복해서 무한루프를 실행하였다. 이를 통해 실제로는 LED가 각 row별로 켜지는 것 이지만 눈에는 전부 동시에 켜져있는 것 처럼 보이게 해주었다.

STEP 14: 다음 프로그램을 위해 dot matrix의 연결을 유지한다.

3) 실험 3

STEP 15: 그림 8.8은 별도로 제공되는 4 x 4 keypad의 구성을 보여준다. 이 그림을 통해 핀번호와 내부 연결관계를 확인한다.

이러한 형태의 keypad에서 어떤 버튼이 눌렸는지 여부를 확인하기 위해 software 측면에서 어떤 고려가 필요한지 생각해보자.

어떤 keypad가 눌렸는지 확인하기 위해서는 keypad의 row에 해당하는 K4 - K7을 하나씩 0으로 해주고 나머지는 1로 세팅한 상태에서 버튼이 눌리면 K0 - K3이 0이 된다. K0 - K3에서 0이 확인되면 K4 - K7에서 0이 되어있는 부분과 K0 - K3에서 0이된 부분의 교집합에 해당하는 버튼이 눌렸다고 확인할 수 있다. 이를 구현하기 위해서 소프트웨어 측면에서 K4 - K7 중에서 단 하나만 0이 되도록 주의해서 코드를 작성해야 하고 빠른 속도로 K0 - K3에 0이 된 것이 있나 빠르게 무한 반복하면서 확인해주어야 한다. 또한 하드웨어적으로 debouncing 문제가 해결되어 있지 않은 경우에는 software 측면에서 debouncing도 해주어야 한다.

STEP 16: NUCLEO-F103RB의 GPIO port 신호와 key matrix를 별도로 제공된 jump wire를 이용하여 다음 표와 같이 (추가로) 연결한다.

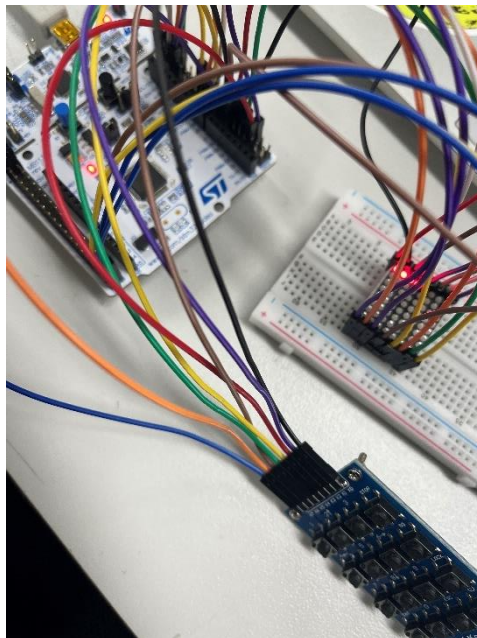


그림 24 직접 연결한 lab8_3 보드

STEP 17: Program 8.3을 이용하여 프로젝트를 생성한다.

```

1  #include <stm32f10x.h>
2
3  u32 i, key_index, key_row, key_col, col_scan;
4
5  int main(void) {
6
7      RCC->APB2ENR = 0x0000001D;
8
9      GPIOC->CRH = 0x00003333;
10     GPIOA->CRH = 0x00008888;
11     GPIOA->ODR = 0x0F00;
12
13     GPIOC->CRL = 0x33333333;
14     GPIOB->CRH = 0x33333333;
15
16     GPIOC->BSRR = 0x0FE;
17
18     key_index = 0;
19     key_row = 0x01;
20
21     while(1) {
22         GPIOC->BSRR = ~(key_row << 8) & 0x0F00 | (key_row << 24);
23         for(i = 0; i < 1000; i++) { ; }
24         key_col = GPIOA->IDR;
25         key_col = (key_col >> 8) & 0x0F;
26         col_scan = 0x01;
27         for (i = 0; i < 4; i++) {
28             if((key_col & col_scan) == 0)
29                 GPIOB->ODR = key_index << 8;
30             col_scan = col_scan << 1;
31             key_index = key_index + 1;
32         }
33         key_row = key_row << 1;
34         if (key_row == 0x10) {
35             key_row = 0x01;
36             key_index = 0;
37         }
38     }
39 }

```

그림 25 직접 작성한 lab8_3.c 코드

STEP 18: Lines 7-14에서 수행되는 초기화 과정의 내용을 이전 단계들에서 파악한 내용과 연관지어 확인해 보자. 이 중에서 Line 11의 역할이 keypad의 연결에 있어서 어떤 의미를 갖는지 설명해보자.

Lines 7-14는 GPIO와 관련된 초기화 과정이다. 먼저 line 7의 `RCC -> APB2ENR = 0x0000001D;` 는 STEP10에서 설명하였듯 GPIOA, GPIOB, GPIOC, AFIO의 데이터가 APB2를 통해서 이동할 수 있도록 enable해준 것이다. 다음으로 line 9의 `GPIOC->CRH = 0x00003333;` 는 PC8 - PC11을 push-pull output모드로 설정한 것이고 line 10의 `GPIOA->CRH = 0x00008888;` 은 PA8 - PA11을 input 모드

로 설정한 것이다. 이때 line 11의 GPIOA->ODR = 0x0F00를 통해서 PC8 – PC11의 input을 pull-up input이 되도록 설정하였다. 이렇게 설정하는 이유는 우리가 사용하는 keypad에서 아무 버튼이 눌리지 않은 경우는 keypad에서 출력이 floating되어 있다. 하지만 디지털에서는 0또는 1로 값이 무조건 결정되기 때문에 혹시나 오류가 발생하는 것을 막기 위해서 pull-up을 달아서 default로 high인 1이 되도록 한 것이다. 왜냐하면 우리가 사용하는 keypad에서는 0이 되어야 눌렸다고 판단하기 때문이다. 추가적으로 line 13과 line 14의 GPIOA->CRL = 0x33333333; 과 GPIOB->CRH = 0x33333333; 은 STEP10에서 설명하였듯 dot matrix를 위해서 각각 GPIOC의 포트0부터 포트7을 push-pull output으로 사용할 수 있도록 하고 GPIOB의 포트8부터 포트15까지 push-pull output으로 사용할 수 있도록 해준다.

STEP 19: Lines 21-38에서 수행되는 무한 반복 동작의 내용을 파악한다.

- Line 22는 무엇을 위함인가? GPIOC->ODR을 사용할 수도 있는데 BSRR을 사용하는 이유는 무엇이라고 생각하는가?
- Keypad에서 어떤 key가 눌렸는지를 어떻게 파악하고 있는가?
- 만일 한 row의 두 키가 동시에 눌리면 어떻게 처리되는가?
- 선택된 key는 dot matrix의 어떤 부분에서 어떤 방식으로 표시되는가? Lines16,29와 연관지어 설명해보자.

먼저 lines 21-38의 코드는 row를 바꿔가며 무한 루프를 돌다가 keypad에 의해 버튼이 눌렸을 때 keypad에서 어떤 column이 눌렸는지 확인하고 row와 column의 조합을 통해서 어떤 버튼이 눌렸는지 버튼을 확인한 뒤 해당 버튼을 이용해서 dot matrix에 2진법으로 버튼의 숫자를 출력을 하는 동작을 수행한다. 이에 대한 일부 결과는 [그림 26]과 [그림 27]을 통해서 확인 가능하다.

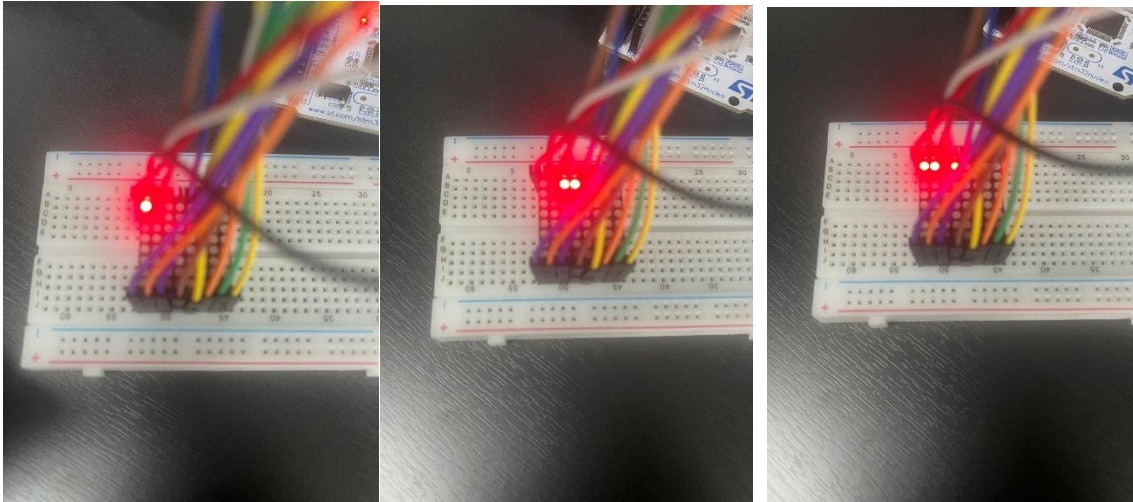


그림 26 S1 클릭 시 LED 출력 / S6 클릭 시 출력 / S11 클릭 시 출력

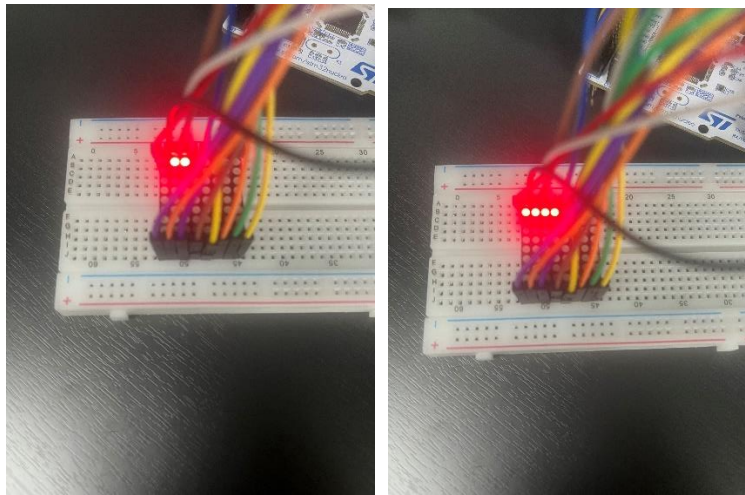


그림 27 S12 클릭 시 LED 출력 / S15 클릭 시 LED 출력

다음으로 keypad가 어떤 key가 눌렸는지 파악하는 방법은 다음과 같다. 먼저 line 22의 코드를 통해서 keypad에서 K4 – K7 중에서 하나만 0으로 reset하고 나머지는 1로 set을 한다. 그리고 line 27-32의 반복문을 통해서 어떤 K0 – K3에서 0이 감지되면 어떤 column에서 버튼이 눌렸는지 알 수 있다. 이를 통해서 col과 row의 교집합인 버튼이 눌렸음을 감지할 수 있다.

이때 만일 한 row에서 두 키가 동시에 눌리면 두 버튼이 모두 감지될 수 있다. 하지만 이에 대한 결과는 dot matrix에서 확인할 수 있는데 col_scan이 순차적으로 증가하면서 버튼이 눌렸는지 파악하기 때문에 더 높은 column 즉 더 큰 숫자를 가진 버튼이 최종적으로 감지된 것처럼 dot matrix에 출력된다. 예를 들면 S0와 S3가 같이 눌리면 S3가 눌렸을 때와 동일한 출력이 나오게 된다.

특정 key가 눌리게 되면 이에 대한 결과는 dot matrix에서 확인할 수 있다. 먼저 line 16의 `GPIOC->BSRR = 0xFE;` 를 통해서 PC1 – PC7이 모두 1로 set되고 PC0은 기본값인 0을 가지고 있

다. 이때 PC0 – PC7은 dot matrix의 row에 해당하는 부분으로 PC0만 0이기 때문에 dot matrix에서 1번 row만 LED로 출력할 수 있는 상황이 되고 나머지 row에서는 LED가 켜질 수 없다. 그리고 이 상태에서 line 29의 GPIOB->ODR = key_index << 8; 에서 key_index는 while문 안에서 keypad의 버튼숫자가 증가할 때 같이 증가하게 되고 이를 왼쪽으로 8칸 shift해서 GPIOB->ODR에 저장함으로 dot matrix에서 column에 해당하는 PB8 – PB15에 key_index의 값이 저장된다. 따라서 dot matrix에서 1번 row에서 key_index, 즉 버튼의 숫자대로 이진법으로 LED가 켜지게 된다. 예를 들어 [그림 26]에서 볼 수 있는 것처럼 S11의 경우 11의 이진수인 2b1011이 LED의 첫번째 row에 켜진 것을 확인할 수 있다. 이와 마찬가지로 [그림 26]과 [그림 27]의 나머지 경우도 모두 동일하다.

4. Exercises

1) C program을 사용하면서 각 포트의 주소를 직접 지정하는 대신 data structure에 의해 정의된 표기를 사용하고 있는데(예를들어, GPIOC->ODR), 전체 프로그램의 어느 부분에서 실제 주소와 이 표기들이 연관을 갖게 되는지 확인해 보자. (프로그램에서 내제적으로 include 되는 .h 파일을 살펴본다.)

우리가 실험에서 사용한 c코드에서 stm32f10x.h 파일을 include하였다. 해당 파일을 살펴보면 [그림 28]처럼 GPIO에 대한 구조체가 선언되어 있는 것을 알 수 있다. 또한 Peripheral memory map이라는 주석 다음으로 [그림 29]와 같이 GPIO를 포함한 각 주변장치별로의 주소가 선언되어 있다. 그리고 이를 바탕으로 [그림 30]과 같이 다양한 주변장치들이 define되어 있다. 이를 바탕으로 c 프로그램에서 각 포트를 GPIOC->ODR과 같이 우리가 원하는 주소에 편하게 접근할 수 있다.

```

1005  /**
1006   * @brief General Purpose I/O
1007   */
1008
1009  typedef struct
1010  {
1011      __IO uint32_t CRL;
1012      __IO uint32_t CRH;
1013      __IO uint32_t IDR;
1014      __IO uint32_t ODR;
1015      __IO uint32_t BSRR;
1016      __IO uint32_t BRR;
1017      __IO uint32_t LCKR;
1018  } GPIO_TypeDef;

```

그림 28 GPIO 구조체

```

1289  /*!< Peripheral memory map */
1290  #define APB1PERIPH_BASE      PERIPH_BASE
1291  #define APB2PERIPH_BASE      (PERIPH_BASE + 0x10000)
1292  #define AHBPERIPH_BASE       (PERIPH_BASE + 0x20000)
1293
1294  #define TIM2_BASE             (APB1PERIPH_BASE + 0x0000)
1295  #define TIM3_BASE             (APB1PERIPH_BASE + 0x0400)
1296  #define TIM4_BASE             (APB1PERIPH_BASE + 0x0800)
1297  #define TIM5_BASE             (APB1PERIPH_BASE + 0x0C00)
1298  #define TIM6_BASE             (APB1PERIPH_BASE + 0x1000)
1299  #define TIM7_BASE             (APB1PERIPH_BASE + 0x1400)
1300  #define TIM12_BASE            (APB1PERIPH_BASE + 0x1800)
1301  #define TIM13_BASE            (APB1PERIPH_BASE + 0x1C00)
1302  #define TIM14_BASE            (APB1PERIPH_BASE + 0x2000)
1303  #define RTC_BASE              (APB1PERIPH_BASE + 0x2800)
1304  #define WWDG_BASE             (APB1PERIPH_BASE + 0x2C00)
1305  #define IWDG_BASE             (APB1PERIPH_BASE + 0x3000)
1306  #define SPI2_BASE             (APB1PERIPH_BASE + 0x3800)
1307  #define SPI3_BASE             (APB1PERIPH_BASE + 0x3C00)
1308  #define USART2_BASE           (APB1PERIPH_BASE + 0x4400)
1309  #define USART3_BASE           (APB1PERIPH_BASE + 0x4800)
1310  #define UART4_BASE            (APB1PERIPH_BASE + 0x4C00)
1311  #define UART5_BASE            (APB1PERIPH_BASE + 0x5000)
1312  #define I2C1_BASE             (APB1PERIPH_BASE + 0x5400)
1313  #define I2C2_BASE             (APB1PERIPH_BASE + 0x5800)
1314  #define CAN1_BASE             (APB1PERIPH_BASE + 0x6400)
1315  #define CAN2_BASE             (APB1PERIPH_BASE + 0x6800)
1316  #define BKP_BASE              (APB1PERIPH_BASE + 0x6C00)
1317  #define PWR_BASE              (APB1PERIPH_BASE + 0x7000)
1318  #define DAC_BASE              (APB1PERIPH_BASE + 0x7400)
1319  #define CEC_BASE              (APB1PERIPH_BASE + 0x7800)
1320
1321  #define AFIO_BASE             (APB2PERIPH_BASE + 0x0000)
1322  #define EXTI_BASE             (APB2PERIPH_BASE + 0x0400)
1323  #define GPIOA_BASE            (APB2PERIPH_BASE + 0x0800)
1324  #define GPIOB_BASE            (APB2PERIPH_BASE + 0x0C00)
1325  #define GPIOC_BASE            (APB2PERIPH_BASE + 0x1000)
1326  #define GPIOD_BASE            (APB2PERIPH_BASE + 0x1400)
1327  #define GPIOE_BASE            (APB2PERIPH_BASE + 0x1800)
1328  #define GPIOF_BASE            (APB2PERIPH_BASE + 0x1C00)
1329  #define GPIOG_BASE            (APB2PERIPH_BASE + 0x2000)
1330  #define ADC1_BASE             (APB2PERIPH_BASE + 0x2400)
1331  #define ADC2_BASE             (APB2PERIPH_BASE + 0x2800)
1332  #define TIM1_BASE             (APB2PERIPH_BASE + 0x2C00)

```

그림 29 일부 Peripheral memory map

```

1414  #define AFIO ((AFIO_TypeDef *) AFIO_BASE)
1415  #define EXTI ((EXTI_TypeDef *) EXTI_BASE)
1416  #define GPIOA ((GPIO_TypeDef *) GPIOA_BASE)
1417  #define GPIOB ((GPIO_TypeDef *) GPIOB_BASE)
1418  #define GPIOC ((GPIO_TypeDef *) GPIOC_BASE)
1419  #define GPIOD ((GPIO_TypeDef *) GPIOD_BASE)
1420  #define GPIOE ((GPIO_TypeDef *) GPIOE_BASE)
1421  #define GPIOF ((GPIO_TypeDef *) GPIOF_BASE)
1422  #define GPIOG ((GPIO_TypeDef *) GPIOG_BASE)
1423  #define ADC1 ((ADC_TypeDef *) ADC1_BASE)
1424  #define ADC2 ((ADC_TypeDef *) ADC2_BASE)
1425  #define TIM1 ((TIM_TypeDef *) TIM1_BASE)
1426  #define SPI1 ((SPI_TypeDef *) SPI1_BASE)

```

그림 30 일부 Peripheral 선언

2) 본 장에서 제시된 프로그램들의 main()에서는 무한 loop를 수행하면서 관련 입출력 소자/장치들을 access하고 있다. 만일 다수의 입력장치와 출력장치를 포함하는 시스템의 구현을 위해 이와 같은 형태(즉 반복적인 scan을 통해 입력과 출력을 확인)하는 프로그램을 작성한다면 프로세서는 대부분의 시간을 입출력 장치의 관리에 할당하여야 할 것이다. 이런 문제를 완화시킬 수 있는 방법에 대해서 알아보자.

무한루프를 통해서 입력과 출력을 반복적인 scan을 하지 않고 인터럽트를 사용하면 된다. 입력이 들어온 경우에 인터럽트가 발생하도록 하여서 인터럽트 핸들러에서 입력을 처리하도록 하거나 타

이며 인터럽트 등을 사용해서 원하는 시간마다 출력을 하게 하는 등 인터럽트를 사용하면 장치의 입력과 출력을 위해서 계속해서 자원을 낭비할 필요가 없다. 또는 멀티 쓰레드로 프로세서가 동작하도록 해서 하나의 작업만 수행하는 것이 아니라 여러 장치의 입출력 외에도 프로세서가 돌아가면서 작업을 할 수 있도록 해주면 된다.

3) Dot matrix display에 단순 형태의 도형이 아닌 일반적인 형태의 문자/도형을 표시하기 위해 table lookup을 이용해서 프로그램을 작성할 수 있겠는가? 컴퓨터에서 주로 사용되는 표시장치의 font는 어떻게 설계되어 적용될지 생각해보자. 또한, 광고 등에 많이 사용되는 dot matrix 전광판에서와 같이 어떤 패턴을 좌우로 흐르는 것처럼(또는 움직이는 것처럼) 보이게 하려면 어떤 고려가 필요하겠는가?

이번 실험에서 STEP13에서 하트 모양을 출력하기 위해서 배열에 미리 하트 모양을 위한 값을 저장하고 불러와서 사용하였다. 이처럼 문자/도형을 그릴 수 있도록 미리 table을 만들어 두고 필요할 때 table lookup을 통해서 원하는 모양을 편하게 출력하도록 프로그램을 작성할 수 있다. 컴퓨터에서 주로 사용되는 표시장치인 모니터에 나타나는 font는 벡터 또는 래스터 이미지의 형태로 나타낼 수 있다. 각 font별로 글자에 해당하는 데이터가 저장되어 있고 해당 데이터를 불러와서 화면에 출력하는 형태로 사용될 것이다. 또한 dot matrix 전광판에서 패턴이 움직이는 것처럼 보이기 위해서는 패턴에 대한 여러가지 방법이 있겠지만 간단하게는 움직이는 모든 경우에 대해서 패턴을 저장해두고 불러오거나, 패턴 데이터에 수학적 계산을 통해서 dot matrix상에서 좌표 값을 바꿔주는 식으로 구현할 수 있다.

4) Dot matrix에 임의의 패턴을 출력하거나 KIT 또는 컴퓨터 키보드에서 사용되는 key matrix에서 어떤 키가 눌렸는지를 확인하기 위해 기본적으로 row와 column을 순차적으로 scan하는 방식을 취한다. Scan 주기가 dot matrix 또는 key matrix의 성능에 어떤 영향을 미치겠는지 생각해보자.

Dot matrix의 scan주기가 빨라지더라도 큰 변화는 없다. 하지만 너무 빨라지게 되면 LED소자에 불빛이 들어오기도 전에 다음 row로 넘어가게 됨으로 LED가 켜지지 않는 문제가 발생할 수 있다.

또한 scan주기가 느려지면 사람의 눈에 dot matrix가 한 번에 켜진 것처럼 보이지 않고 한 줄씩 켜지는 것처럼 보일 수 있다는 단점이 있다. 따라서 적절한 scan주기를 가지도록 설계해야한다. 다음으로 key matrix의 scan주기가 빨라지면 debouncing문제가 잡히지 않을 수도 있어 사용자가 누른 키가 인식이 되지 않을 수 있다. 또한 scan주기를 느리게 하면 키를 누르고 나서 인식되는 속도가 느려져 사용자가 딜레이를 느껴 불편할 수 있다. 따라서 key matrix와 dot matrix모두 적당한 scan주기를 가져야 한다.

5) Keypad의 구성에 사용되는 기계적인 스위치는 bouncing 문제를 피하기 어렵다. 이 문제를 완화하기 위한 software 및 hardware적인 조치에 대해 알아보자.

Keypad에서 디바운싱문제를 소프트웨어 또는 하드웨어적으로 해결할 수 있다. 먼저 소프트웨어적으로는 신호가 들어오고 나서 안정화를 위해서 일정시간을 기다리고 신호를 감지하는 방법으로 디바운싱을 할 수 있다. 또는 하드웨어적으로는 캐패시터사용해 RC회로를 구성해서 신호의 bouncing을 해결할 수 있다.

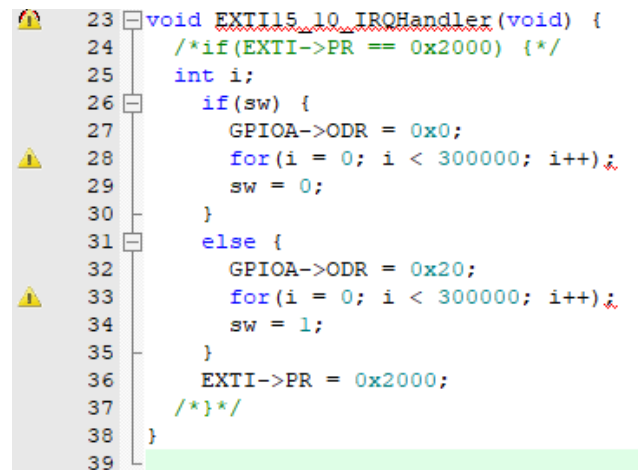
5. 추가 실험

1) GPIO Interrupt Latency

우선 실험 1에서 사용한 코드에서 line 24, 33, 34를 지운 상태에서 프로그램을 수행해보면 처음 버튼을 눌렀을 때는 LED가 한 번 켜지고 다시 꺼지지 않는 문제가 발생하였다. 이는 인터럽트 핸들러로 들어가고 나서 LED를 처음 한 번 토글시켜 켜 둔 뒤에 인터럽트의 pending상태를 해제하지 않아 ISR이 끝나고 나서도 아직도 pending되어 있음으로 다시 ISR로 들어가 계속해서 켜다 껴다를 반복하는 문제가 발생한다.

다음으로 이를 해결해 보기 위해서 line 33만을 다시 작성한 상태로 프로그램을 수행해보면 버튼을 눌러도 LED가 토글되지 않고 아주 가끔만 토글되는 문제가 발생한다. 이로 미루어 봤을 때 우선 버튼을 눌렀을 때 인터럽트 핸들러가 수행된다고 예상할 수 있다. 이 경우를 확인하기 위해 LED가 켜지고 나서 불이 눈에 보이는 시간동안 켜지도록 다음 [그림 31]과 같이 코드를 작성하였다. 그 결과 버튼을 한 번 누를 때 마다 LED가 켜졌다 꺼졌다 하는 것을 확인할 수 있다. 즉 버

튼을 한 번 누르는데 인터럽트 핸들러는 두 번 호출된 것이다. 이는 EXTI->PR=0x2000을 통해서 EXTI의 펜딩상태는 해제하였지만 인터럽트를 관리하는 NVIC에서의 펜딩 상태는 핸들러를 나가고 나서 자동으로 해제되게 된다. 따라서 NVIC의 펜딩상태가 해제되는 시간보다 다시 인터럽트가 호출되는 시간이 더 짧게 되면 NVIC의 펜딩상태가 해제되기 전에 인터럽트 핸들러가 한 번 더 호출되는 문제가 발생할 수 있다.



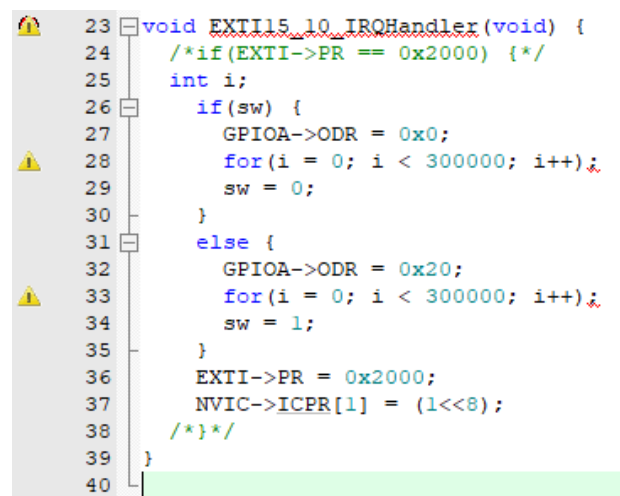
```

23 void EXTI15_10_IRQHandler(void) {
24     /*if(EXTI->PR == 0x2000) {*/
25     int i;
26     if(sw) {
27         GPIOA->ODR = 0x0;
28         for(i = 0; i < 300000; i++);
29         sw = 0;
30     }
31     else {
32         GPIOA->ODR = 0x20;
33         for(i = 0; i < 300000; i++);
34         sw = 1;
35     }
36     EXTI->PR = 0x2000;
37     /*}*/
38 }
39

```

그림 31 문제 확인을 위해 수정한 핸들러 코드

이를 해결하기 위해서는 실험에서처럼 핸들러 내부에서 if(EXTI->PR == 0x2000)를 추가해 핸들러에 진입하더라도 EXTI도 펜딩되어 있어야만 동작이 수행되도록 하거나 핸들러 내부에서 아래 [그림 32]처럼 NVIC_ICPR에 접근하여 인터럽트의 펜딩을 강제로 해제하는 방법이 있다. 다만 이 경우에는 인터럽트 핸들러가 수행되는 도중에 버튼이 한 번 더 눌러서 인터럽트가 펜딩된 경우 제한한다는 단점이 있지만 이번 실험에서는 사람의 속도로 컴퓨터가 핸들러를 처리하는 속도보다 빠르게 버튼을 더 누를 수는 없기 때문에 문제가 없다.



```

23 void EXTI15_10_IRQHandler(void) {
24     /*if(EXTI->PR == 0x2000) {*/
25     int i;
26     if(sw) {
27         GPIOA->ODR = 0x0;
28         for(i = 0; i < 300000; i++);
29         sw = 0;
30     }
31     else {
32         GPIOA->ODR = 0x20;
33         for(i = 0; i < 300000; i++);
34         sw = 1;
35     }
36     EXTI->PR = 0x2000;
37     NVIC->ICPR[1] = (1<<8);
38     /*}*/
39 }
40

```

그림 32 문제를 해결한 핸들러 코드

2) GPIO Configuration Abstraction

```
#include <stm32f10x.h>

#define GPIOAA 1
#define GPIOBB 2
#define GPIOCC 3
#define GPIODD 4
#define GPIOEE 5

#define PIN0 0x0001
#define PIN1 0x0002
#define PIN2 0x0004
#define PIN3 0x0008
#define PIN4 0x0010
#define PIN5 0x0020
#define PIN6 0x0040
#define PIN7 0x0080
#define PIN8 0x0100
#define PIN9 0x0200
#define PIN10 0x0400
#define PIN11 0x0800
#define PIN12 0x1000
#define PIN13 0x2000
#define PIN14 0x4000
#define PIN15 0x8000
```

```
void GPIO_PUSH_PULL_OUTPUT(u32 abcde, u32 start, u32 end) {
```

```
    u32 n;
```

```
    for(n = start; n <= end; n++) {
```

```
        switch(abcde) {
```

```
            case GPIOAA :
```

```
                if(n == start) {
```

```
                    RCC->APB2ENR |= ((u32)1 << 2);
```

```
                    if(n < 8) {
```

```
                        GPIOA->CRL = ((u32)3 << (4 * n));
```

```
                    }
```

```
                    else {
```

```
                        GPIOA->CRH = ((u32)3 << (4 * (n - 8)));
```

```
                    }
```

```
                }
```

```
                if(n < 8) {
```

```
                    GPIOA->CRL |= ((u32)3 << (4 * n));
```

```
                }
```

```
                else {
```

```
                    GPIOA->CRH |= ((u32)3 << (4 * (n - 8)));
```

```
                }
```

```
                break;
```

```
            case GPIOBB :
```

```
                if(n == start) {
```

```
                    RCC->APB2ENR |= ((u32)1 << 3);
```

```
                    if(n < 8) {
```

```

        GPIOB->CRL = ((u32)3 << (4 * n));

    }

    else {

        GPIOB->CRH = ((u32)3 << (4 * (n - 8)));

    }

}

if(n < 8) {

    GPIOB->CRL |= ((u32)3 << (4 * n));

}

else {

    GPIOB->CRH |= ((u32)3 << (4 * (n - 8)));

}

break;

case GPIOCC :

    if(n == start) {

        RCC->APB2ENR |= ((u32)1 << 4);

        if(n < 8) {

            GPIOC->CRL = ((u32)3 << (4 * n));

        }

        else {

            GPIOC->CRH = ((u32)3 << (4 * (n - 8)));

        }

    }

    if(n < 8) {

        GPIOC->CRL |= ((u32)3 << (4 * n));

```

```

    }

    else {

        GPIOC->CRH |= ((u32)3 << (4 * (n - 8)));

    }

    break;

case GPIODD :

    if(n == start) {

        RCC->APB2ENR |= ((u32)1 << 5);

        if(n < 8) {

            GPIOD->CRL = ((u32)3 << (4 * n));

        }

        else {

            GPIOD->CRH = ((u32)3 << (4 * (n - 8)));

        }

    }

    if(n < 8) {

        GPIOD->CRL |= ((u32)3 << (4 * n));

    }

    else {

        GPIOD->CRH |= ((u32)3 << (4 * (n - 8)));

    }

    break;

case GPIOEE :

    if(n == start) {

```

```

RCC->APB2ENR |= ((u32)1 << 6);

if(n < 8) {

    GPIOE->CRL = ((u32)3 << (4 * n));

}

else {

    GPIOE->CRH = ((u32)3 << (4 * (n - 8)));

}

}

if(n < 8) {

    GPIOE->CRL |= ((u32)3 << (4 * n));

}

else {

    GPIOE->CRH |= ((u32)3 << (4 * (n - 8)));

}

break;

}

}

}

void GPIO_PULL_UP_INPUT(u32 abcde, u32 start, u32 end) {

    u32 n;

    for(n = start; n <= end; n++) {

        switch(abcde) {

            case GPIOAA :

                if(n == start) {

                    RCC->APB2ENR |= ((u32)1 << 2);

                    if(n < 8) {

```



```

        GPIOA->CRL = ((u32)8 << (4 * n));

    }

    else {

        GPIOA->CRH = ((u32)8 << (4 * (n - 8)));

    }

}

if(n < 8) {

    GPIOA->CRL |= ((u32)8 << (4 * n));

}

else {

    GPIOA->CRH |= ((u32)8 << (4 * (n - 8)));

}

GPIOA->ODR |= ((u32)1 << n);

break;

case GPIOBB :

    if(n == start) {

        RCC->APB2ENR |= ((u32)1 << 3);

        if(n < 8) {

            GPIOB->CRL = ((u32)8 << (4 * n));

        }

        else {

            GPIOB->CRH = ((u32)8 << (4 * (n - 8)));

        }

    }

    if(n < 8) {

```

```

        GPIOB->CRL |= ((u32)8 << (4 * n));

    }

    else {

        GPIOB->CRH |= ((u32)8 << (4 * (n - 8)));

    }

    GPIOB->ODR |= ((u32)1 << n);

    break;

case GPIOCC :

    if(n == start) {

        RCC->APB2ENR |= ((u32)1 << 4);

        if(n < 8) {

            GPIOC->CRL = ((u32)8 << (4 * n));

        }

        else {

            GPIOC->CRH = ((u32)8 << (4 * (n - 8)));

        }

    }

    if(n < 8) {

        GPIOC->CRL |= ((u32)8 << (4 * n));

    }

    else {

        GPIOC->CRH |= ((u32)8 << (4 * (n - 8)));

    }

    GPIOC->ODR |= ((u32)1 << n);

    break;

```

case GPIODD :

```
if(n == start) {  
    RCC->APB2ENR |= ((u32)1 << 5);  
  
    if(n < 8) {  
        GPIOD->CRL = ((u32)8 << (4 * n));  
    }  
  
    else {  
        GPIOD->CRH = ((u32)8 << (4 * (n - 8)));  
    }  
}  
  
if(n < 8) {  
    GPIOD->CRL |= ((u32)8 << (4 * n));  
}  
  
else {  
    GPIOD->CRH |= ((u32)8 << (4 * (n - 8)));  
}  
  
GPIOD->ODR |= ((u32)1 << n);  
  
break;
```

case GPIOEE :

```
if(n == start) {  
    RCC->APB2ENR |= ((u32)1 << 6);  
  
    if(n < 8) {  
        GPIOE->CRL = ((u32)8 << (4 * n));  
    }  
}
```

```

        else {

            GPIOE->CRH = ((u32)8 << (4 * (n - 8)));

        }

    }

    if(n < 8) {

        GPIOE->CRL |= ((u32)8 << (4 * n));

    }

    else {

        GPIOE->CRH |= ((u32)8 << (4 * (n - 8)));

    }

    GPIOE->ODR |= ((u32)1 << n);

    break;

}

}

}

```

```

void GPIO_SET(u32 abcde, u32 n) {

```

```

    switch(abcde) {

```

```

        case GPIOAA :

```

```

            GPIOA->BSRR = n;

```

```

            break;

```

```

        case GPIOBB :

```

```

            GPIOB->BSRR = n;

```

```

            break;

```

```

        case GPIOCC :

```

```
        GPIOC->BSRR = n;

        break;

    case GPIODD :

        GPIOD->BSRR = n;

        break;

    case GPIOEE :

        GPIOE->BSRR = n;

        break;

    }
}

void GPIO_RESET(u32 abcde, u32 n) {

    switch(abcde) {

        case GPIOAA :

            GPIOA->BRR |= n;

            break;

        case GPIOBB :

            GPIOB->BRR |= n;

            break;

        case GPIOCC :

            GPIOC->BRR |= n;

            break;
```

```
case GPIODD :

    GPIOD->BRR |= n;

    break;


case GPIOEE :

    GPIOE->BRR |= n;

    break;

}

}

void GPIO_WRITE_ODR(u32 abcde, u32 n) {

    switch(abcde) {

        case GPIOAA :

            GPIOA->ODR = n;

            break;


        case GPIOBB :

            GPIOB->ODR = n;

            break;


        case GPIOCC :

            GPIOC->ODR = n;

            break;


        case GPIODD :

            GPIOD->ODR = n;

            break;
```

```
case GPIOEE :  
    GPIOE->ODR = n;  
    break;  
}  
}  
u32 GPIO_READ_IDR(u32 abcde) {  
    switch(abcde) {  
        case GPIOAA :  
            return GPIOA->IDR;  
            break;  
  
        case GPIOBB :  
            return GPIOB->IDR;  
            break;  
  
        case GPIOCC :  
            return GPIOC->IDR;  
            break;  
  
        case GPIODD :  
            return GPIOD->IDR;  
            break;  
  
        case GPIOEE :  
            return GPIOE->IDR;
```

```

        break;

    }

}

u32 i, key_index, key_row, key_col, col_scan;

int main(void) {

    //RCC->APB2ENR = 0x0000001D;

    GPIO_PUSH_PULL_OUTPUT(GPIOCC, 8, 11);
    GPIO_PULL_UP_INPUT(GPIOAA, 8, 11);
    //GPIOC->CRH = 0x00003333;
    //GPIOA->CRH = 0x00008888;
    //GPIOA->ODR = 0x0F00;

    GPIO_PUSH_PULL_OUTPUT(GPIOCC, 0, 7);
    GPIO_PUSH_PULL_OUTPUT(GPIOBB, 8, 15);
    //GPIOC->CRL = 0x33333333;
    //GPIOB->CRH = 0x33333333;

    GPIO_SET(GPIOCC, PIN1|PIN2|PIN3|PIN4|PIN5|PIN6|PIN7);
    //GPIOC->BSRR = 0x0FE;

    key_index = 0;

    key_row = 0x01;

```



```

while(1) {

    GPIO_SET(GPIOCC, (~(key_row << 8) & 0x0F00) | (key_row << 24));

    //GPIOC->BSRR = (~(key_row << 8) & 0x0F00) | (key_row << 24);

    for(i = 0; i < 1000; i++) { ; }

    key_col = GPIO_READ_IDR(GPIOAA);

    //key_col = GPIOA->IDR;

    key_col = (key_col >> 8) & 0x0F;

    col_scan = 0x01;

    for (i = 0; i < 4; i++) {

        if((key_col & col_scan) == 0)

            GPIO_WRITE_ODR(GPIOBB, (key_index << 8));

            //GPIOB->ODR = key_index << 8;

        col_scan = col_scan << 1;

        key_index = key_index + 1;

    }

    key_row = key_row << 1;

    if (key_row == 0x10) {

        key_row = 0x01;

        key_index = 0;

    }

}

}

```

위의 코드와 같이 우리가 작성한 코드를 추상화하였다. GPIO의 이름과 PIN을 모두 #define을 통해서 인자로 전달하도록 하였고 GPIO_PUSH_PULL_OUTPUT, GPIO_PULL_UP_INPUT, GPIO_SET, GPIO_RESET, GPIO_WRITE_ODR, GPIO_READ_IDR 함수를 작성하여 해당 함수 내에서 case문을 사용해서 어떤 GPIO인지 파악한 뒤 핀에 따라서 관련 레지스터들의 값을 할당하도록 해주었다. 그

결과 실험 3에서와 동일하게 동작하는 것을 확인할 수 있었다. 물론 처음 코드를 추상화하는 과정에서 작성해야 할 코드가 훨씬 많지만 나중에는 훨씬 편하게 활용할 수 있다는 장점이 있다.

3) 4x3 Keyboard Implement

```
#include <stm32f10x.h>

void keyboard(void);

u32 i, key_index, key_row, key_col, col_scan, new_button, old_button, ch, dot_row, dot_col, j = 0,
k;

u32 font[26][8] = {
    {0x18, 0x24, 0x42, 0x42, 0x7e, 0x42, 0x42, 0x00},
    {0x7c, 0x22, 0x22, 0x3c, 0x22, 0x22, 0x7c, 0x00},
    {0x18,0x24,0x40,0x40,0x40,0x40,0x24,0x18},
    {0x78,0x44,0x42,0x42,0x42,0x42,0x44,0x78},
    {0x7E,0x40,0x40,0x7E,0x40,0x40,0x40,0x7E},
    {0x7E,0x40,0x40,0x7C,0x40,0x40,0x40,0x40},
    {0x18,0x24,0x40,0x40,0x46,0x42,0x24,0x18},
    {0x42,0x42,0x42,0x7E,0x42,0x42,0x42,0x42},
    {0x08,0x08,0x08,0x08,0x08,0x08,0x08,0x08},
    {0x02,0x02,0x02,0x02,0x02,0x42,0x24,0x18},
    {0x44,0x48,0x50,0x60,0x50,0x48,0x44,0x42},
    {0x40,0x40,0x40,0x40,0x40,0x40,0x40,0x7E},
    {0x66,0x5A,0x5A,0x5A,0x5A,0x5A,0x42,0x42},
    {0x42,0x62,0x52,0x52,0x4A,0x4A,0x46,0x42},
```

```
{0x18,0x24,0x42,0x42,0x42,0x42,0x24,0x18},  
  
{0x7C,0x42,0x42,0x42,0x7C,0x40,0x40,0x40},  
  
{0x18,0x24,0x42,0x42,0x42,0x4E,0x26,0x1A},  
  
{0x7C,0x42,0x42,0x42,0x7C,0x48,0x44,0x42},  
  
{0x3C,0x42,0x40,0x20,0x1C,0x02,0x02,0x3C},  
  
{0x3E,0x08,0x08,0x08,0x08,0x08,0x08,0x08},  
  
{0x42,0x42,0x42,0x42,0x42,0x42,0x24,0x18},  
  
{0x42,0x42,0x42,0x42,0x42,0x42,0x24,0x18},  
  
{0x44,0x44,0x54,0x54,0x54,0x28,0x28,0x28},  
  
{0x42,0x24,0x18,0x18,0x18,0x24,0x42,0x42},  
  
{0x44,0x44,0x44,0x28,0x10,0x10,0x10,0x10},  
  
{0x7E,0x02,0x04,0x08,0x10,0x20,0x40,0x7E}  
};
```

```
int main(void) {
```

```
    RCC->APB2ENR = 0x0000001D;
```

```
    GPIOC->CRH = 0x00003333;
```

```
    GPIOA->CRH = 0x00008888;
```

```
    GPIOA->ODR = 0x0F00;
```

```
    GPIOC->CRL = 0x33333333;
```

```
    GPIOB->CRH = 0x33333333;
```

```
    dot_row = 0x0001;
```

```

dot_col = 0x0000;

key_index = 0;

key_row = 0x01;

while(1) {

    GPIOC->BSRR = (~(key_row << 8) & 0x0F00) | (key_row << 24);

    for(i = 0; i < 1000; i++) { ; }

    key_col = GPIOA->IDR;

    key_col = (key_col >> 8) & 0x0F;

    col_scan = 0x01;

    for (i = 0; i < 4; i++) {

        if((key_col & col_scan) == 0) {

            for(k = 0; k < 300000; k++);

            new_button = key_index;

            keyboard();

        }

        col_scan = col_scan << 1;

        key_index = key_index + 1;

    }

    key_row = key_row << 1;

    if (key_row == 0x10) {

        key_row = 0x01;

        key_index = 0;

    }
}

```

```

GPIOC->ODR = ~dot_row;

    dot_row = dot_row << 1;

    dot_col = font[ch - 'a'][j++];

    GPIOB->ODR = dot_col << 8;

    if(dot_row == 0x0100) {

        dot_row = 0x0001;

        j = 0;

    }

}

}

```

```

void keyboard(void) {

    if(new_button == old_button) {

    switch(new_button) {

    case 8: //2: abc

        if(ch != 'c') {

            ch++;

        }

        break;

    case 4: //3: def

        if(ch != 'f') {

            ch++;

        }

        break;

    case 13: //4: ghi

        if(ch != 'i') {

```

```
        ch++;  
    }  
  
    break;  
  
case 9: //5: jkl  
  
    if(ch != 'l') {  
  
        ch++;  
  
    }  
  
    break;  
  
case 5: //6: mno  
  
    if(ch != 'o') {  
  
        ch++;  
  
    }  
  
    break;  
  
case 14: //7: pqrt  
  
    if(ch != 't') {  
  
        ch++;  
  
    }  
  
    break;  
  
case 10: //8: tuv  
  
    if(ch != 'v') {  
  
        ch++;  
  
    }  
  
    break;  
  
case 6: //9: wxyz  
  
    if(ch != 'z') {  
  
        ch++;  
  
    }
```

```
    }

    break;

default:

    return;

}

}

else if(new_button != 9999){

    switch(new_button) {

        case 8: //2: abc

            ch = 'a';

            break;

        case 4: //3: def

            ch = 'd';

            break;

        case 13: //4: ghi

            ch = 'g';

            break;

        case 9: //5: jkl

            ch = 'j';

            break;

        case 5: //6: mno

            ch = 'm';

            break;

        case 14: //7: pqrt

            ch = 'p';

            break;
```

```

        case 10: //8: tuv

            ch = 't';

            break;

        case 6: //9: wxyz

            ch = 'w';

            break;

        case 3: //enter

            break;

        default:

            return;

    }

    if(new_button != 9999) {

        old_button = new_button;

    }

    new_button = 9999;

}
}

```

예전 핸드폰을 보면 4x3 키패드를 통해서 문자를 입력하였다. 이번 추가실험에서는 이에 아이디어를 얻어서 아래 [그림 33]과 같은 키보드 배치로 설계하였다. 키패드에서 알파벳을 입력받으면 해당 알파벳을 dot matrix를 통해서 출력하도록 하였다. 간단하게 코드를 설명하면 실험 3에서 상요한 코드를 바탕으로 key pad에서 눌린 값을 받아와서 처음 눌렀으면 해당 키패드의 첫번째 알파벳으로 값을 설정하였다. 그리고 같은 값이 다시 한번 눌리면 다음 알파벳으로 넘어가도록 keypad()함수를 작성해주었다. 그리고 미리 선언해둔 배열을 통해서 dot matrix에 LED를 출력하도록 하였다.



그림 33 키보드 배치

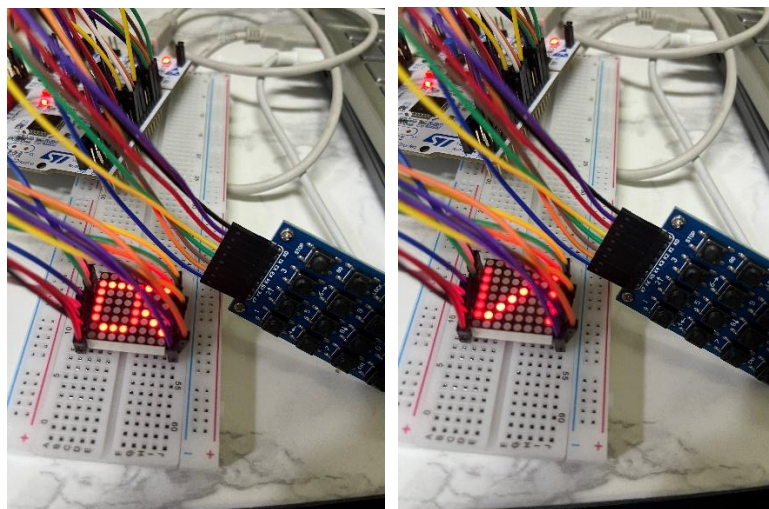
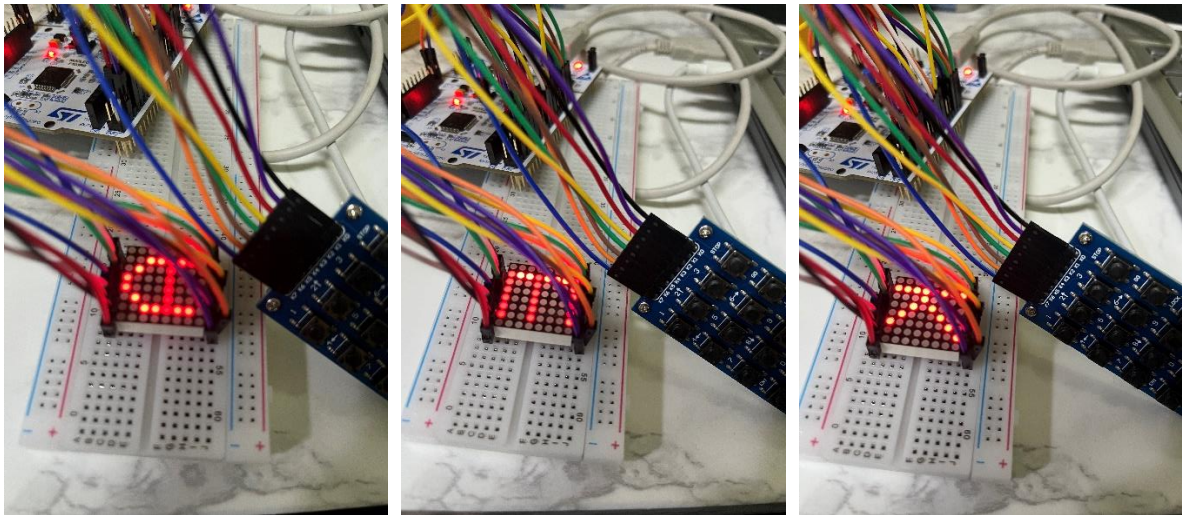


그림 34 알파벳 출력 예시

4) 7-Segment Application

```
#include <stm32f10x.h>

void print_num(u32 n);

int main(void) {
    u32 i, j;

    RCC->APB2ENR = 0x0000001D;

    GPIOA->CRL = 0x33333333;
    GPIOA->CRH = 0x33330000;

    while(1) {
        for(i = 0; i < 9; i++) {
            print_num(i);
            for(j = 0; j < 3000000; j++);
        }
    }

    void print_num(u32 n) {
        switch(n) {
            case 0:
                GPIOA->ODR = 0xFF03;
                break;
```

case 1:

GPIOA->ODR = 0xFF9F;

break;

case 2:

GPIOA->ODR = 0xFF25;

break;

case 3:

GPIOA->ODR = 0xFF0D;

break;

case 4:

GPIOA->ODR = 0xFF99;

break;

case 5:

GPIOA->ODR = 0xFF49;

break;

case 6:

GPIOA->ODR = 0xFF41;

break;

case 7:

GPIOA->ODR = 0xFF1B;

break;

case 8:

GPIOA->ODR = 0xFF01;

break;

case 9:

GPIOA->ODR = 0xFF09;

```
break;

}

}
```

Anode 타입의 7-segment display를 위한 코드를 작성하였다. 사용한 소자의 회로도에는 아래 [그림 35]와 같다. A-DP에 해당하는 핀은 PA0 – PA7에 할당하고 DIG1 – DIG4에 해당하는 PIN은 PA12 – PA15에 할당하였다. 그리고 display의 동작을 확인하기 위해서 카운터 형식으로 동작하도록 코드를 작성한 뒤 print_num()함수를 통해서 원하는 숫자를 첫번째 화면에 출력하도록 코드를 작성하였다. 그 결과에 대한 예시는 아래 [그림 36]을 통해서 확인 가능하다.

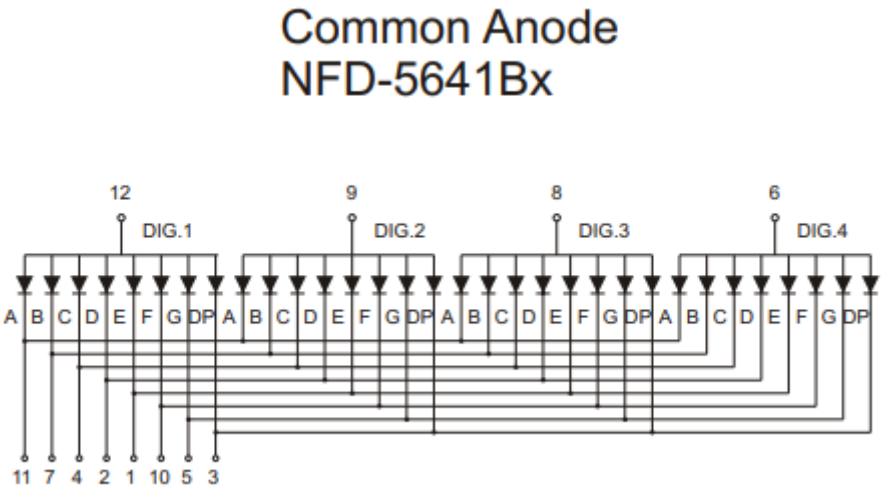


그림 35 Anode 타입의 7 segment display

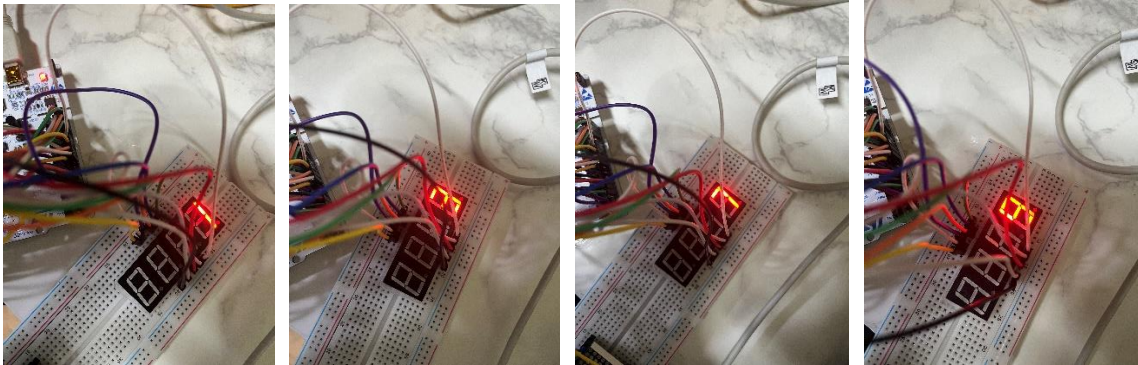


그림 36 7-segment display 동작 예시

5) Keypad Interrupt

```
#include <stm32f10x.h>
```

```
u32 i, key_index, key_row, key_col, col_scan;
```

```
int main(void) {
```

```
    RCC->APB2ENR = 0x0000001D;
```

```
    GPIOC->CRH = 0x00003333;
```

```
    GPIOA->CRH = 0x00008888;
```

```
    GPIOA->ODR = 0x0F00;
```

```
    EXTI->FTSR = 0xF00;
```

```
    EXTI->IMR = 0xF00;
```

```
    AFIO->EXTICR[2] = 0x0000;
```

```
    NVIC->ISER[0] = (1 << 23);
```

```
    NVIC->ISER[1] = (1 << 8);
```

```
    GPIOC->CRL = 0x33333333;
```

```
    GPIOB->CRH = 0x33333333
```

```
    GPIOC->BSRR = 0x0FE;
```

```
    key_index = 0;
```

```
    key_row = 0x01;
```

```
    while(1) {
```

```

        __WFI();

    }

}

void EXTI15_10_IRQHandler(void) {

    GPIOC->BSRR = (~(key_row << 8) & 0x0F00) | (key_row << 24);

    for(i = 0; i < 1000; i++) { ; }

    key_col = GPIOA->IDR;

    key_col = (key_col >> 8) & 0x0F;

    col_scan = 0x01;

    for (i = 0; i < 4; i++) {

        if((key_col & col_scan) == 0)

            GPIOB->ODR = key_index << 8;

        col_scan = col_scan << 1;

        key_index = key_index + 1;

    }

    key_row = key_row << 1;

    if (key_row == 0x10) {

        key_row = 0x01;

        key_index = 0;

    }

}

void EXTI9_5_IRQHandler(void) {

    GPIOC->BSRR = (~(key_row << 8) & 0x0F00) | (key_row << 24);

```

```

for(i = 0; i < 1000; i++) { ; }

key_col = GPIOA->IDR;

key_col = (key_col >> 8) & 0x0F;

col_scan = 0x01;

for (i = 0; i < 4; i++) {

    if((key_col & col_scan) == 0)

        GPIOB->ODR = key_index << 8;

    col_scan = col_scan << 1;

    key_index = key_index + 1;

}

key_row = key_row << 1;

if (key_row == 0x10) {

    key_row = 0x01;

    key_index = 0;

}

}

```

실험 3에서 key matrix를 감지하기 위해서 계속해서 무한 루프를 돌고 있다. 이는 CPU의 자원을 소모하는 작업이기 때문에 `_WFI()`함수를 사용해서 인터럽트가 발생하기 전까지 CPU가 wait하도록 할 수 있다. 이를 활용하여 실험3 코드를 기반으로 GPIO를 EXTI로 연결하고 EXTI를 통해 인터럽트가 발생하도록 설정하고 `while(1)`안에서 `_WFI()`함수를 호출해서 인터럽트가 발생하기 전까지는 계속 wait하도록 하였다. 이 코드에서는 dot matrix에서도 하나의 row만 출력하기 때문에 눌린 버튼에 따라서 한 번만 출력을 설정해두고 dot matrix를 위해서 무한 루프도 돌 필요가 없기 때문에 계속해서 wait하며 CPU의 자원을 아낄 수 있다. 그리고 인터럽트가 발생하면 각 핸들러로 들어가 어떤 버튼이 눌렸는지 확인하고 그에 따라 dot matrix를 키도록 코드를 작성하였다. 그 결과는 아래 [그림 37]과 같이 기존 실험과 동일하게 잘 출력되는 것을 확인할 수 있다.

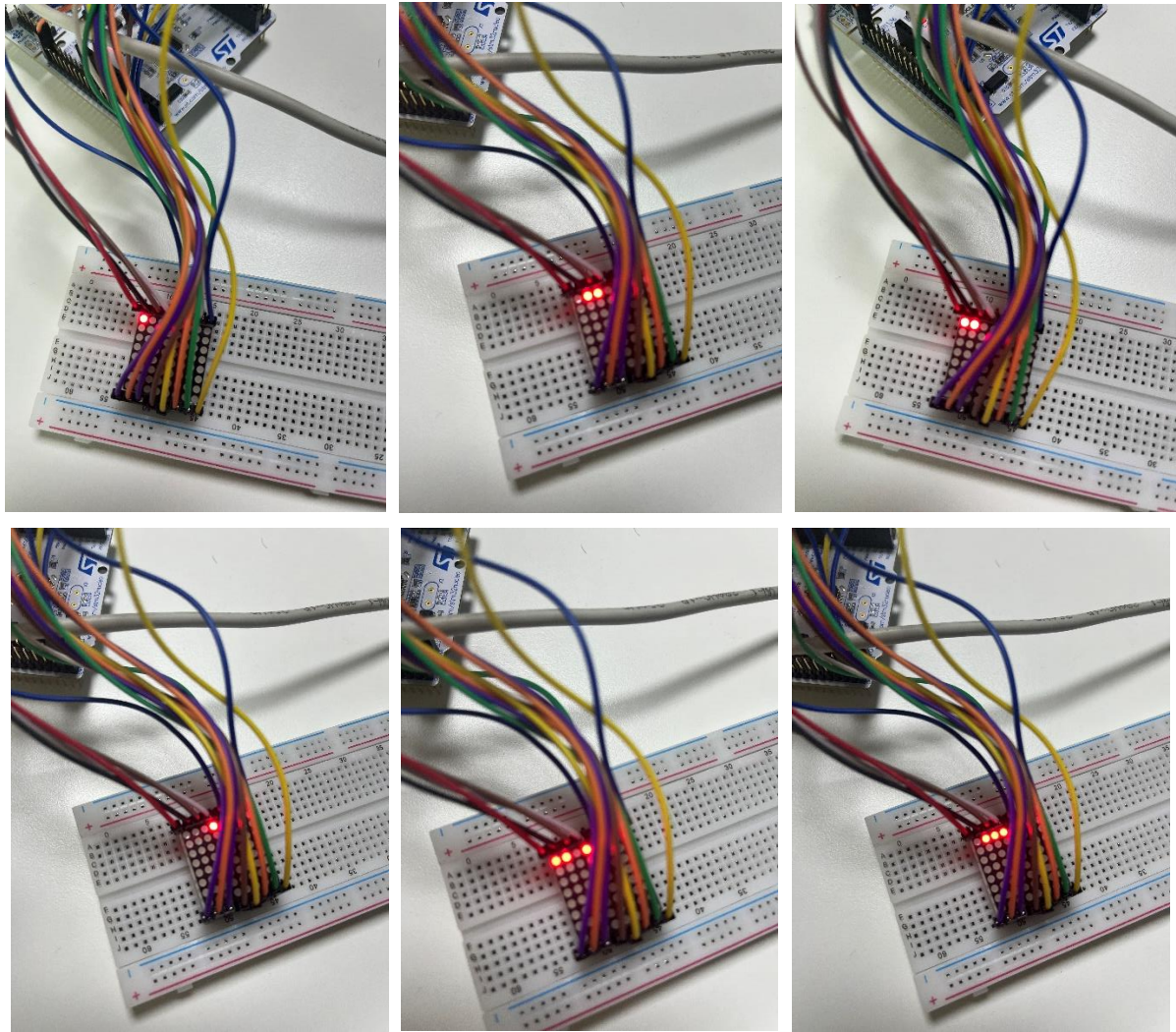


그림 37 추가실험5 출력 예시

6. 결론

이번 실험을 통해서 GPIO를 이용해서 외부 소자와 보드를 연결해서 입력과 출력을 사용하는 방법을 익힐 수 있었다. 이 과정에서 어셈블리가 아니라 C언어로 작성함으로써 미리 data structure로 정의된 것을 사용함으로써 high level 언어의 장점 또한 알 수 있었다. 그리고 GPIO를 초기화하는 과정부터 configuration설정까지 직접 수행하고 소자와 연결하고 동작하는 것을 확인할 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).

STMicroelectronics (2022). STM32F103x8, STM32F103xB Data Sheet (Rev 18)

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

히연. (2021). EMBEDDED RECIPES.