4주차 결과보고서

10조 (001분반)

과목명 : 임베디드시스템설계및실험

담당 교수 : 정상화 교수님

담당 조교 : 최호진 조교님

조원 : 201924603 하규승(조장)

201727102 강준혁

201924525 이광훈

202023139 박지원

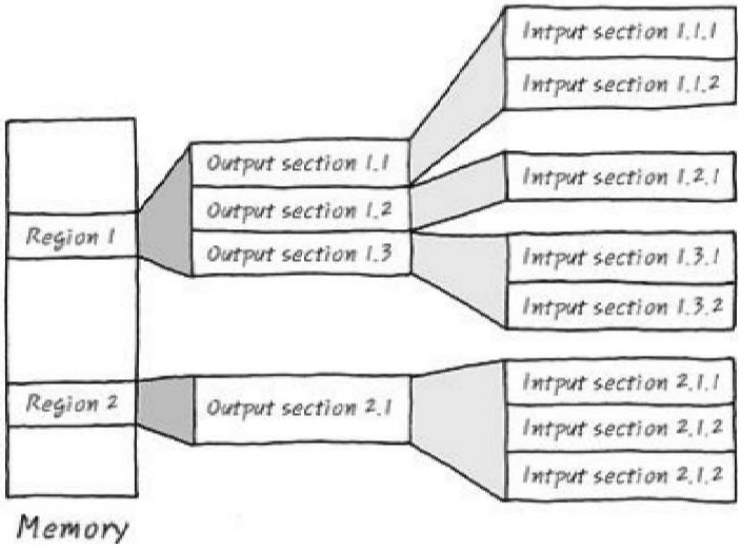
제출 날짜 : 2023.10.01

목차

1. 실험목표

1-2. 스캐터 파일의 이해 및 플래시 프로그래밍

스캐터 파일(Scatter File)은 실행될 바이너리 이미지가 메모리에 로드될 때, 그 바이너리 이미지의 어떤 영역이 어느 주소에 어느 크기만큼 배치되어야 할지 작성한 파일이다. 이러한 스캐터 파일은 바이너리의 여러 부분을 각각의 메모리 영역에 로드해야 될 때 필요하며, 특히 자주 사용되거나 빠른 실행을 요구하는 코드영역을 접근 시간이 빠른 메모리에 우선 배치하도록 설정할 수 있다. 스캐터 파일은 **그림 1.1**과 같이 구성된다 :



[그림 1.1] Scatter File의 구조

Input Section

RO (Read Only) : 읽기만 가능하고 수정이 불가한 symbol(ex) const형, 소스 코드)

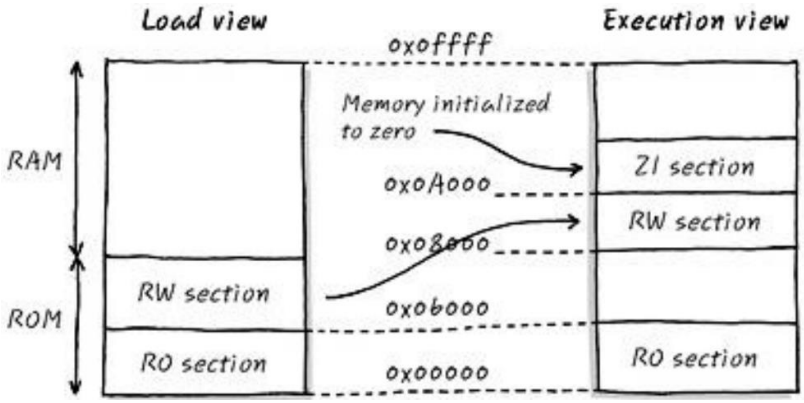
RW (Read Write) : 읽기/쓰기가 모두 가능한 symbol(ex) 초기화 값이 있는 전역 변수)

ZI (Zero Initialized) : 0으로 초기화된 symbol(ex) 초기화 값이 없는 전역 변수)

Output Section

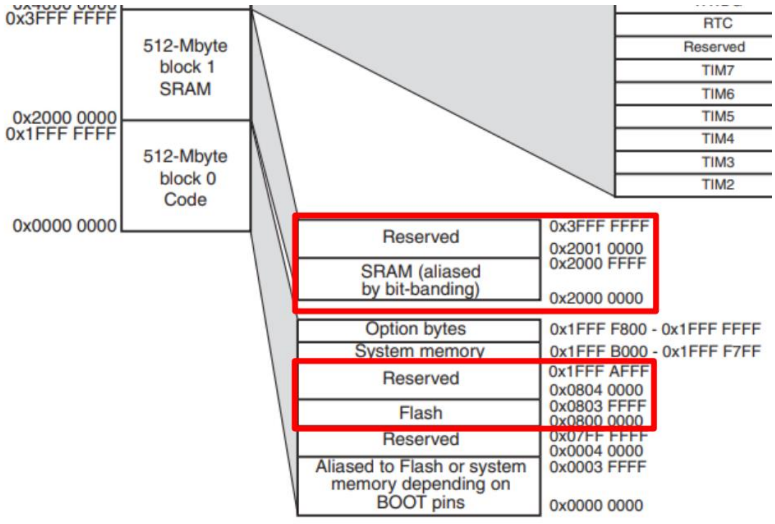
Input Section 들 중에 같은 속성값을 갖는 것들을 묶어 놓은 것

Region



[그림 1.2] Program의 Load/Execution View

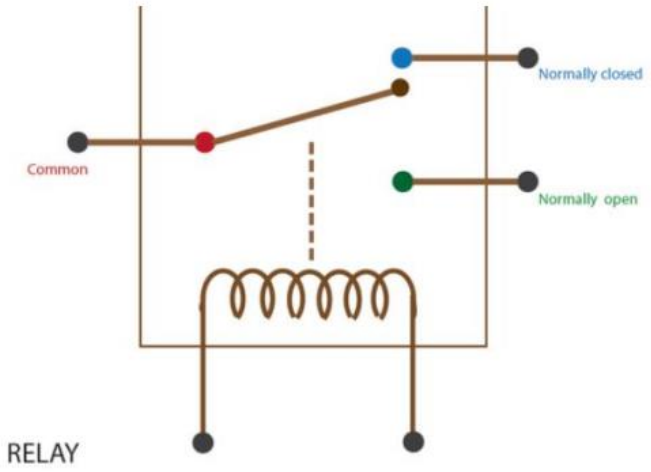
그림 1.2와 같이 구성된 Scatter File을 통해, Binary Image는 지정된 주소의 ROM에 로드되어 대기 중인 프로그램이 된다. 프로그램이 실행되게 되면 RO Section은 그대로 ROM에 남고, RW와 ZI Section은 지정된 주소의 RAM으로 이동하여 실행되게 된다.



[그림 1.3] RAM / ROM의 Memory Mapping

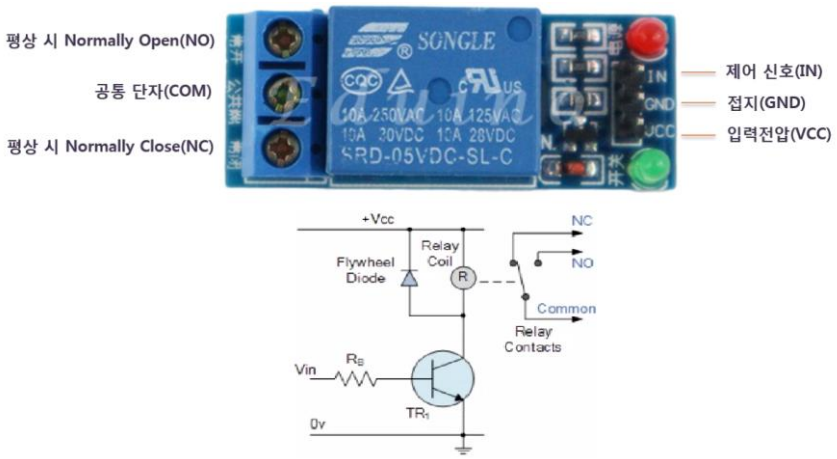
이번 실험에서는 RAM과 ROM 영역의 크기를 위의 남는 공간인 Reserved 공간을 활용하여 실험 조건을 충족하도록 늘리거나 줄인다.

1-2. 릴레이 모듈의 이해 및 임베디드 펌웨어를 통한 동작



[그림 1.4] Relay의 내부 회로도

릴레이는 입력되는 신호에 따라 작동하여 회로를 개폐하는 장치이다. 릴레이는 내부에 코일로 이루어진 전자석이 있으며 이 전자석에 전류가 흐르게 되면 전자기 유도현상에 의해 자석이 되어 옆의 철을 끌어당기게 되어 스위치가 ON 된다. 이후 전류가 끊어지면 다시 옆의 철이 원래 자리로 돌아가게 되어 스위치가 OFF 된다.



[그림 1.5] Relay Module의 외관과 내부 회로도

릴레이 모듈은 릴레이를 이용해 구성되었으며, Vcc - GND를 통해 3.3V의 전원이 인가되어야 한다. 릴레이의 상태는 IN에 따라 결정되며, 평소(IN이 Low)에는 NC와 COM이 연결된 상태, In이 High 일 땐 NO와 COM이 연결된 상태가 된다.

3. 폴링 방식의 이해

CPU와 입/출력 장치 사이에는 통신이 필요하고, 주된 통신 방법이 2가지가 있는데, 크게 인터럽트(Interrupt)와 폴링(Polling) 방식으로 나뉜다.

일단 각 방식에 관해 설명하자면, Interrupt 방식은 **H/W가 CPU에** 예외 상황이 발생하여 처리가 필요하다고 알려주는 방식이다. 반대로 Polling 방식은 **CPU가 H/W를** 주기적으로 감시하면서 예외 상황 발생 시 처리하는 방식이다.

각 방식에 대해 차이점은 다음과 같다. Interrupt는 CPU 외부에서 신호가 들어오는 방식이고, H/W로부터 신호가 들어오게 되면 CPU는 진행 중인 작업을 잠시 멈추고 Interrupt 처리 루틴을 실행하여 신호를 처리하게 된다. 반대로 Polling은 CPU가 H/W의 변화를 주기적으로 체크해야 하므로 CPU가 다른 일을 하는 도중에 신호를 읽을 수 없고, 이에 따라 문제가 생기더라도 CPU가 일단 작업을 시작했다면, 그 작업이 끝난 이후 오류를 발견하게 된다.

이러한 Interrupt 방식과 Polling 방식의 차이점으로 인해, Polling 방식은 오류가 일어난 시점에서 즉시 반응이 힘들고, 계속 해당 H/W에 대해 상태를 체크해야 하므로 레지스터를 계속 확인해 봐야 하므로 CPU의 점유율이 높다는 단점이 있다. 반면, Interrupt 방식에 비해 Polling 방식이 좀 더 구현이 쉽다는 장점이 있다.

2. 세부 목표

2-1. Datasheet 및 Reference Manual을 참고하여 해당 레지스터 및 주소에 대한 설정 이해

2-2. 스캐터 파일을 통해 플래시 메모리에 프로그램 다운로드

2-3. 플래시 메모리에 올려진 프로그램 정상적인 동작 확인

3. 실험 기구

- STM32F107 보드
- 브레드보드, 점프선
- 릴레이 모듈 2개
- 모터 1개
- IAR 프로그램

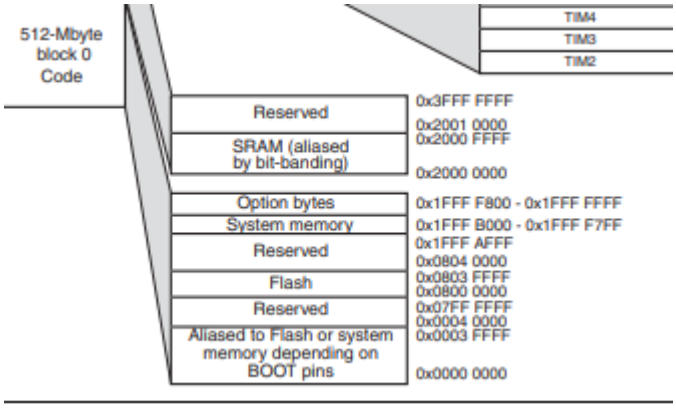
4. 실험 과정

4-1. Scatter file 업로드

```
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__ = // TODO
define symbol __ICFEDIT_region_ROM_end__   = // TODO
define symbol __ICFEDIT_region_RAM_start__ = // TODO
define symbol __ICFEDIT_region_RAM_end__   = // TODO
```

[그림 4.1] 수정 전 myicf.icf 파일의 Region 메모리 주소

지난 주와 동일한 방식으로 프로젝트를 생성한 후, 제공된 myicf.icf 스캐터 파일을 수정한다.



[그림 4.2] Datasheet의 memory map

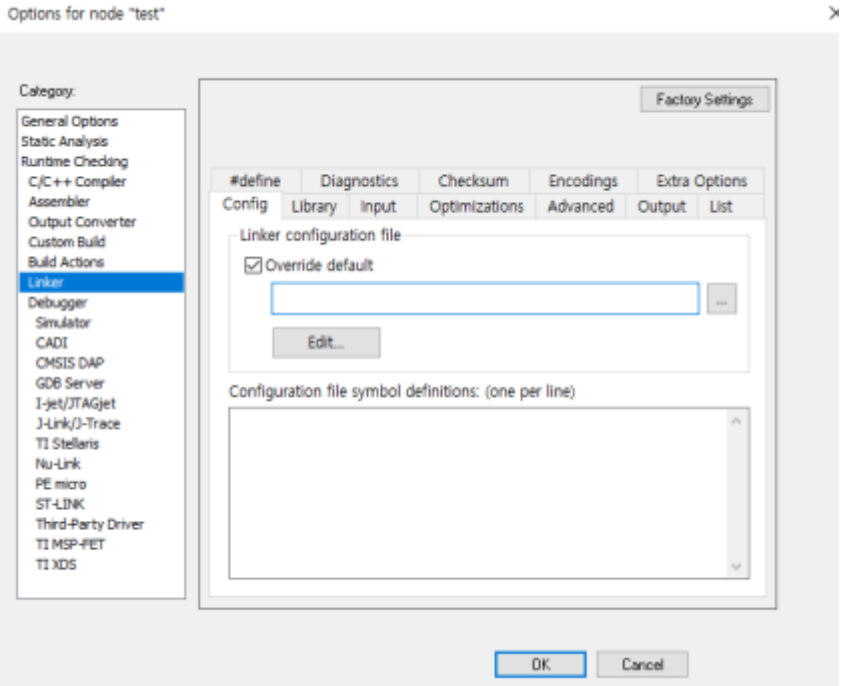
이 때, 주소의 시작 위치는 datasheet의 memory map을 참조한다.

```
define symbol __ICFEDIT_region_ROM_start__ = 0x08000000;
define symbol __ICFEDIT_region_ROM_end__   = 0x08080000;
define symbol __ICFEDIT_region_RAM_start__ = 0x200000000;
define symbol __ICFEDIT_region_RAM_end__   = 0x20008000;
```

[그림 4.3] 수정 후 myicf.icf 파일의 Region 메모리 주소

Memory region을 설정할 때, reserved memory를 초과할 수 없음에 유의하여 원하는 크기만큼 icf파일을 수정한다.

Memory 시작 주소에 필요한 크기 값만큼 더하기 연산을 하면 된다.



[그림 1.4] icf 파일 업로드

수정한 myicf.icf 파일을 업로드하면 scatter file 업로드가 완료된다.

4-2. Main.c 구현

```
3   #define RCC_APB2ENR *((volatile unsigned int *)0x40021018)
4
5   #define GPIOA_CRL *(volatile unsigned int *)0x40010800 //switch Configuration Regiester Low
6   #define GPIOA_IDR *(volatile unsigned int *)0x40010808 //switch Input Data Register
7
8   #define GPIOB_CRH *(volatile unsigned int *)0x40010C04 //switch Configuration Regiester high
9   #define GPIOB_IDR *(volatile unsigned int *)0x40010C08 //switch Input Data Register
10
11  #define GPIOC_CRL *(volatile unsigned int *)0x40011000 //switch Configuration Regiester Low
12  #define GPIOC_CRH *(volatile unsigned int *)0x40011004 //switchConfiguration Regiester Low
13  #define GPIOC_IDR *(volatile unsigned int *)0x40011008 //switch Input Data Register
14
15  #define GPIOD_CRL *(volatile unsigned int *)0x40011400 //Output Configuration Regiester Low
16  #define GPIOD_BSRR *(volatile unsigned int *)0x40011410 //Output Bit Set/Reset Register
```

[그림 4.5] 레지스터 정의부

3주차와 동일하게 RCC를 설정하고, 사용할 button과 relay module에 대해 Port configuration resister를 설정한다.

```
RCC_APB2ENR |=  0x0000003C;

GPIOA_CRL &= 0xFFFFFFF0; // reset
GPIOA_CRL |= 0x00000008; // switch PA 0 input mode
GPIOA_IDR |= 0x00000000; // switch Input Data Register reset

GPIOB_CRH &= 0xFFFFF0FF ; // reset
GPIOB_CRH |= 0x00000800; // switch PB 10 input mode
GPIOB_IDR |= 0x00000000; // switch Input Data Register reset

GPIOC_CRL &= 0xFF0FFFFF; // reset
GPIOC_CRL |= 0x00800000; // switch PC 4 input mode
GPIOC_IDR |= 0x00000000; // switch Input Data Register reset

GPIOC_CRH &= 0xFFF0FFFF; // reset
GPIOC_CRH |= 0x00080000; // switch PC 13 input mode
GPIOC_IDR |= 0x00000000; // switch Input Data Register reset

GPIOD_CRL &= 0x0FF000FF; // reset
GPIOD_CRL |= 0x30033300; // PD 2,3,4,7 output push-pull
```

[그림 4.2]reset and mode setting

이후, 각 GPIO에 대하여 input, output mode를 설정한다.

```
void pd2off() {
  GPIOD_BSRR |= 0x00000004;
}

void pd3off() {
  GPIOD_BSRR |= 0x00000008;
}

void pd2on() {
  GPIOD_BSRR |= 0x00040000;
}

void pd3on() {
  GPIOD_BSRR |= 0x00080000;
}

void delay(){
  int i;
  for(i = 0; i<10000000;i++){

  }
}
```

[그림 4.7] 함수 구현부

PD2와 PD3에 대해 output을 켜고 끄는 각 함수들을 구현한다.

또한, 일정 시간 동안 코드 동작을 지연하는 delay() 함수 역시 구현한다.

이 때, 실험에서 요구하는 지연시간에 따라 적절히 for문 내 조건을 수정한다.

```
while(1){
  if(~GPIOC_IDR & 0x10){     //0100
    pd2on();
    pd3off();// switch 1번 누르면 PD2가 켜지고, PD3가 꺼짐.
  }
  else if(~GPIOB_IDR & 0x400){  //10 0000 0000
    pd2off();
    pd3on(); //switch 2번 누르면 PD2가 꺼지고, PD3가 켜짐.
  }
  else if(~GPIOC_IDR & 0x2000) {  //10 0000 0000 0000
    pd2on();
    pd3off();

    delay();

    pd2off();
    pd3on();

    delay();

    pd3off();
    pd2off();

    delay();
  }
  else if(~GPIOA_IDR & 0x1){  //0000 0001
    pd2off();
    pd3off();// switch 4 누르면 PD3, PD7 LED 가 꺼짐
  }
}
```

[그림 4.8] 작동부

각 port의 input에 따라 **그림 4.7**의 함수를 호출한다.

스위치에 할당된 기능은 다음과 같다.

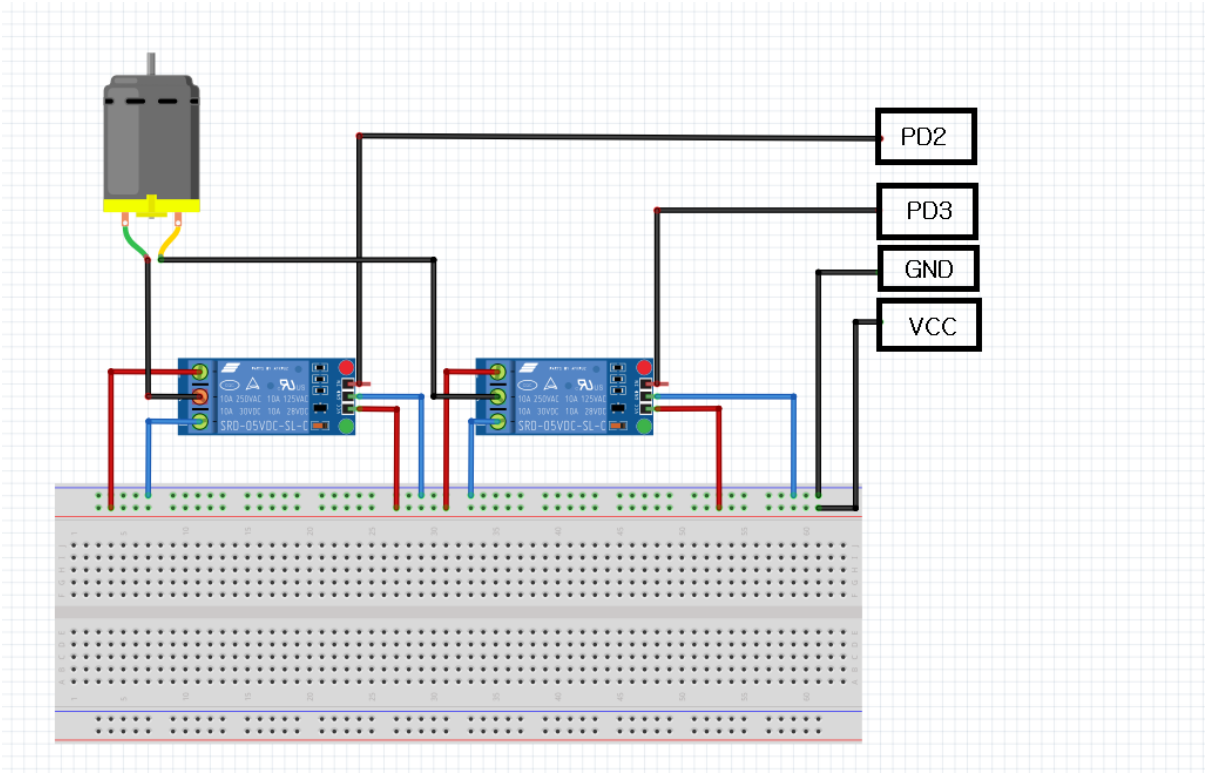
A. 버튼1(KEY1) : 모터 정방향 회전 (전진)

B. 버튼2(KEY2) : 모터 역방향 회전 (후진)

C. 버튼3(KEY3) : 모터 정방향 2초 역방향 2초 회전 후 정지

D. 버튼4(KEY4) : 모터 정지

4-3. 릴레이 모듈 및 회로 연결



[그림 4.9] Schematic

그림 4.9와 같이 회로를 연결한다.

이 때 3.3 전원에 릴레이 모듈을 연결함에 유의한다.

4-4. 동작 확인

스위치 1, 2, 3, 4 번을 차례로 눌러 각 스위치에 할당된 동작대로 모터가 동작하는지 확인한다.

이후 3 번 스위치를 누른 후 작동하는 동안 4 번 스위치를 눌러 모터가 꺼지는지 확인한다.

5. 실험 결과

1, 2, 3, 4 번에 해당하는 스위치를 눌렀을 때 각 스위치에 할당한 동작이 정상적으로 작동하였다.

또한, 3 번 스위치를 누른 후 모터가 동작하는 동안 4 번 스위치를 눌렀을 때 모터가 꺼지지 않았다.

6. 결과에 대한 논의

각 스위치에 할당된 동작이 정상적으로 수행됨을 보아, 릴레이 모듈과 모터가 적절히 연결됐으며 레지스터에 직접 접근하여 설정한 보드의 각 포트들이 정상적으로 설정되었음을 알 수 있다.

또한 3 번 스위치를 누르고 모터가 동작하는 동안, 4 번 스위치를 눌렀을 때 모터가 꺼지지 않음을 확인했다. 이는 실험 과정에서 3 번에 할당된 delay 함수가 동작하는 동안, 4 번 스위치를 눌렀을 때 발생하는 input 을 보드가 인식하지 않아 발생한 현상으로 추측한다. 따라서 실험 목표 문단에서 설명한 폴링 방식의 정의에 따라 보드가 폴링 방식으로 동작하고 있음을 알 수 있다.

7. 결론

보드에 스캐터 파일을 업로드하여 RAM 과 ROM 의 메모리 영역을 설정하였다. 또한, 릴레이 모듈의 원리를 이용해 양방향으로 모터를 제어할 수 있도록 회로를 구성하였다. 레지스터에 직접 접근하여 각 port 를 설정하고, 원하는 동작이 수행되도록 코드를 구현하였다.

보드에 구현한 코드를 업로드 한 후, 각 스위치를 눌렀을 때 할당한 동작이 오류 없이 모터로 수행됨을 확인하였다. 또한, 특정 스위치를 눌러 delay() 메서드를 실행시키는 동시에 보드가 다른 input 을 인식하거나, output 을 내보낼 수 없음을 확인하였다.

위 결과에 따라 향후 동시에 여러 동작이 수행되는 회로를 만들고자 할 때엔 delay 메서드의 사용을 지양해야 할 것으로 추측한다. delay 메서드를 사용할 경우 다른 메서드의 동작이 중지되기 때문이다.