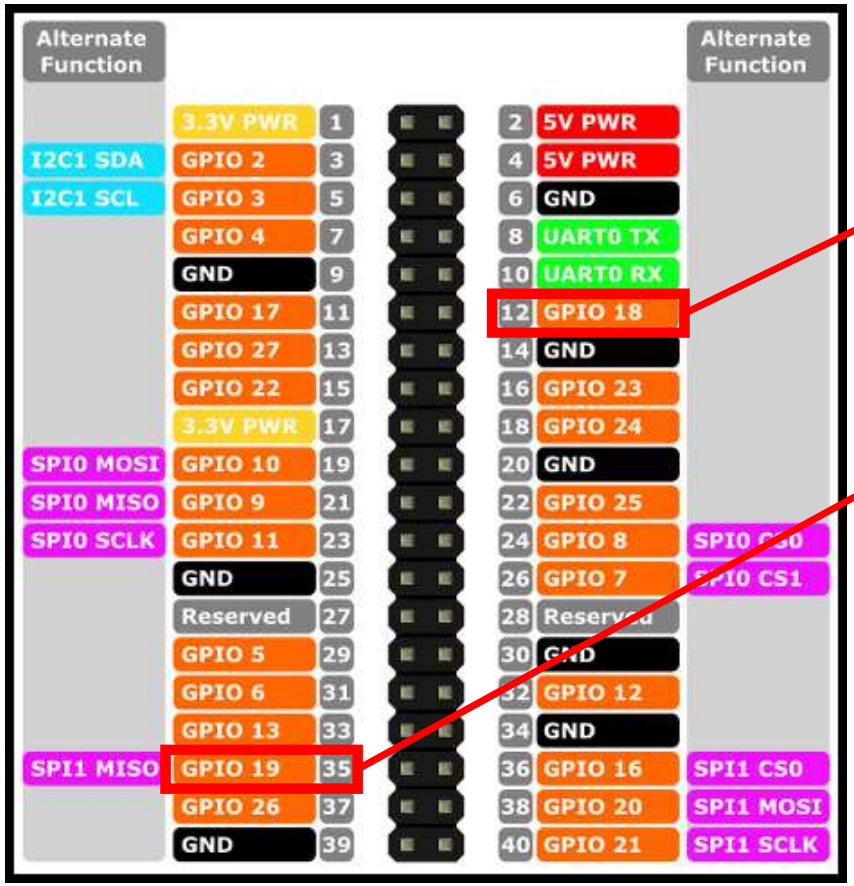
GPIO Driver

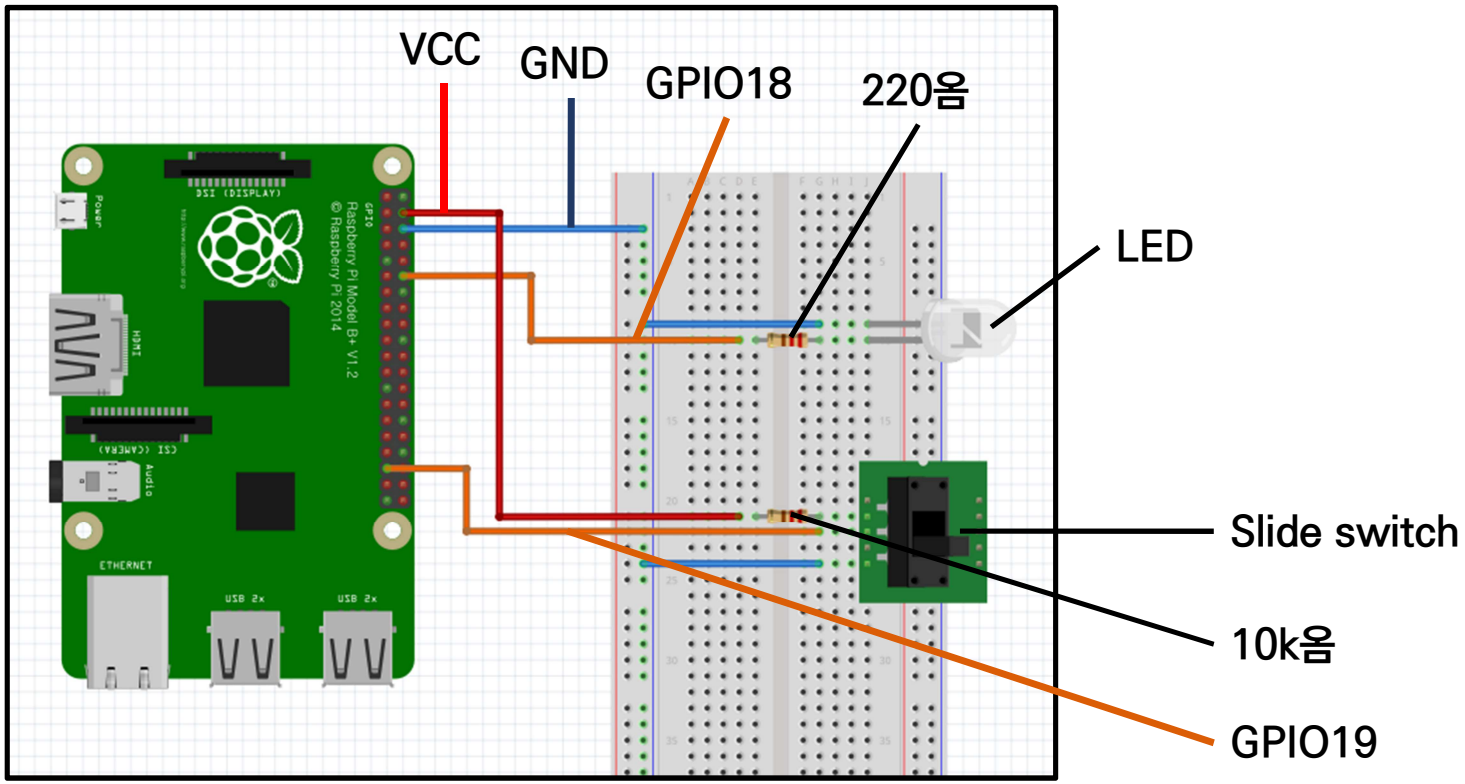
GPIO핀



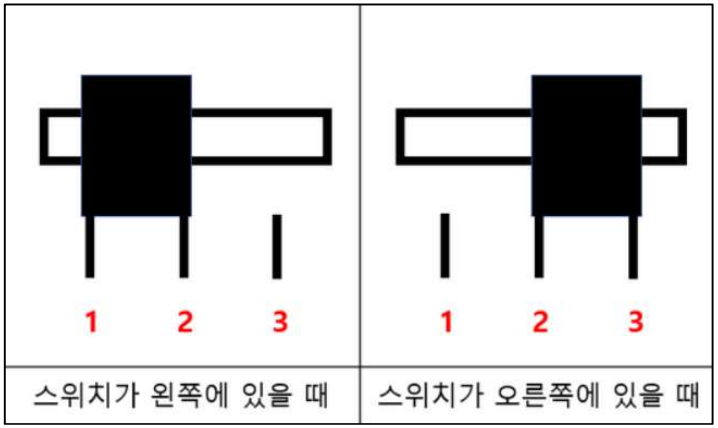
GPIO 18
: LED제어에 사용할 핀

GPIO 19
: Switch제어에 사용할 핀

회로구성도



Slide Switch 동작원리



슬라이드 스위치는 세 개의 단자가 존재합니다.

좌측부터 번호를 매겨 1번, 2번, 3번이라고 한다면 스위치를 좌측으로 이동시키면 1번과 2번이 서로 연결되고, 우측으로 이동시키면 2번과 3번이 서로 연결됩니다.

예를들어, 1번에 LED에 연결되는 전원 단자를 연결하고 2번에 그라운드를 연결했을 때 스위치가 좌측에 있을 경우 전류가 흘러 LED의 불빛이 켜지지만 스위치를 우측으로 이동시키면 전류가 흐르지 않아 LED의 불빛이 꺼집니다.

BCM2835 데이터시트

- Register View
  - ✓ 디바이스 드라이버에서는 GPIO를 제어하는 레지스터에 접근하여 직접 레지스터의 값을 바꿈으로써 특정 GPIO핀의 제어가 가능하다.
  - ✓ Raspberrypi B+의 Soc인 BCM2835의 데이터시트를 참조하여 필요한 레지스터들을 확인한다.
  - ✓ 표의 첫 필드인 Address는 각 레지스터들의 주소가 담긴 필드이다.
  - ✓ 이 실습에서 필요한 레지스터 필드의 이름은 GPFSEL1, GPSET0, GPCLR0, GPLEV0으로 각 4개의 레지스터 필드들을 다룬다.

6.1 Register View

The GPIO has 41 registers. All accesses are assumed to be 32-bit.

Address	Field Name	Description	Size	Read/ Write
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0004	GPFSEL1	GPIO Function Select 1	32	R/W
0x 7E20 0008	GPFSEL2	GPIO Function Select 2	32	R/W
0x 7E20 000C	GPFSEL3	GPIO Function Select 3	32	R/W
0x 7E20 0010	GPFSEL4	GPIO Function Select 4	32	R/W
0x 7E20 0014	GPFSEL5	GPIO Function Select 5	32	R/W
0x 7E20 0018	-	Reserved	-	-
0x 7E20 001C	GPSET0	GPIO Pin Output Set 0	32	W
0x 7E20 0020	GPSET1	GPIO Pin Output Set 1	32	W
0x 7E20 0024	-	Reserved	-	-
0x 7E20 0028	GPCLR0	GPIO Pin Output Clear 0	32	W
0x 7E20 002C	GPCLR1	GPIO Pin Output Clear 1	32	W
0x 7E20 0030	-	Reserved	-	-
0x 7E20 0034	GPLEV0	GPIO Pin Level 0	32	R
0x 7E20 0038	GPLEV1	GPIO Pin Level 1	32	R
0x 7E20 003C	-	Reserved	-	-

BCM2835 데이터시트

- GPFSEL1
  - ✓ 레지스터 주소 : 0x7E20 0004
  - ✓ GPFSEL레지스터 필드는 GPIO핀의 모드를 INPUT이나 OUTPUT으로 지정해주는 필드이다.
    - ➢ 예를 들어, 19번 GPIO를 OUTPUT모드로 지정 해주고 싶을 땐 총 32bit중 27~29번째 bit를 001로 설정해주면 된다.
  - ✓ 레지스터는 32bit로 구성되어 있음을 확인가능 하며, 32bit의 명령어형식을 사용하는 것을 알 수 있다.
  - ✓ 32bit임으로 이 필드는 10개의 GPIO에 특정 모드를 할당할 수 있다.

Bit(s)	Field Name	Description	Type	Reset
31-30	---	Reserved	R	0
29-27	FSEL19	FSEL19 - Function Select 19 000 = GPIO Pin 19 is an input 001 = GPIO Pin 19 is an output 100 = GPIO Pin 19 takes alternate function 0 101 = GPIO Pin 19 takes alternate function 1 110 = GPIO Pin 19 takes alternate function 2 111 = GPIO Pin 19 takes alternate function 3 011 = GPIO Pin 19 takes alternate function 4 010 = GPIO Pin 19 takes alternate function 5	R/W	0
26-24	FSEL18	FSEL18 - Function Select 18	R/W	0
23-21	FSEL17	FSEL17 - Function Select 17	R/W	0
20-18	FSEL16	FSEL16 - Function Select 16	R/W	0
17-15	FSEL15	FSEL15 - Function Select 15	R/W	0
14-12	FSEL14	FSEL14 - Function Select 14	R/W	0
11-9	FSEL13	FSEL13 - Function Select 13	R/W	0
8-6	FSEL12	FSEL12 - Function Select 12	R/W	0
5-3	FSEL11	FSEL11 - Function Select 11	R/W	0
2-0	FSEL10	FSEL10 - Function Select 10	R/W	0

BCM2835 데이터시트

- GPFSEL0
  - ✓ 레지스터 주소 : 0x7E20 001C
  - ✓ GPSET필드는 OUTPUT모드로 설정된 GPIO핀에 해당하는 비트를 SET을 시켜 HIGH값을 출력해내는 필드이다.
    - ➢ 예를 들어, GPFSEL1필드를 통해 19번 GPIO가 OUTPUT모드로 설정되고, GPSET0필드에서는 19번 GPIO에 해당하는 bit에 1이라는 값으로 설정하게되면 19번 GPIO에서는 HIGH값을 출력하게 된다. 19번 GPIO에 해당하는 bit는 32bit중 20번째에 해당하는 bit이다.
  - ✓ 이 필드는 32bit로 이루어져 있음으로 31번 GPIO까지 제어가 가능하다. (0~31번)

Bit(s)	Field Name	Description	Type	Reset
31-0	SETn (n=0..31)	0 = No effect 1 = Set GPIO pin *n*	R/W	0

BCM2835 데이터시트

- GPCLR0
  - ✓ 레지스터 주소 : 0x7E20 0028
  - ✓ GPCLR필드는 OUTPUT모드로 설정된 GPIO핀에 해당하는 비트를 SET시켜 LOW값을 출력해 내는 필드이다.
  - ✓ GPIO핀에 LOW를 출력해내는 방법은 이전의 GPSET필드의 방법과 동일하다.
  - ✓ 이 필드는 32bit로 이루어져 있음으로 31번 GPIO까지 제어가 가능하다.

Bit(s)	Field Name	Description	Type	Reset
31-0	CLRn (n=0..31)	0 = No effect 1 = Clear GPIO pin *n*	R/W	0

BCM2835 데이터시트

- GPLEV0
  - ✓ 레지스터 주소 : 0x7E20 0034
  - ✓ GPLEV필드는 INPUT모드로 설정된 GPIO핀에 입력되는 값을 읽어 들이는 필드이다.
    - ➢ 예를 들어, GPFSEL을 통해 INPUT모드로 설정된 GPIO 19번 핀에 스위치 등으로 인해서 HIGH값을 입력 받게 되면 GPLEV필드 내의 GPIO 19에 해당하는 비트가 0에서 1로 설정되고, 다시 LOW값이 입력 받게 되면 비트가 1에서 0으로 바뀌게 된다.
  - ✓ 이 필드는 32bit로 이루어져 있음으로 31번 GPIO까지의 값을 읽어 들일 수 있다.

Bit(s)	Field Name	Description	Type	Reset
31-0	LEVn (n=0..31)	0 = GPIO pin *n* is low 0 = GPIO pin *n* is high	R/W	0

레지스터주소가 매핑된 메모리의 물리적주소

1.2.3 ARM physical addresses

Physical addresses start at 0x00000000 for RAM.

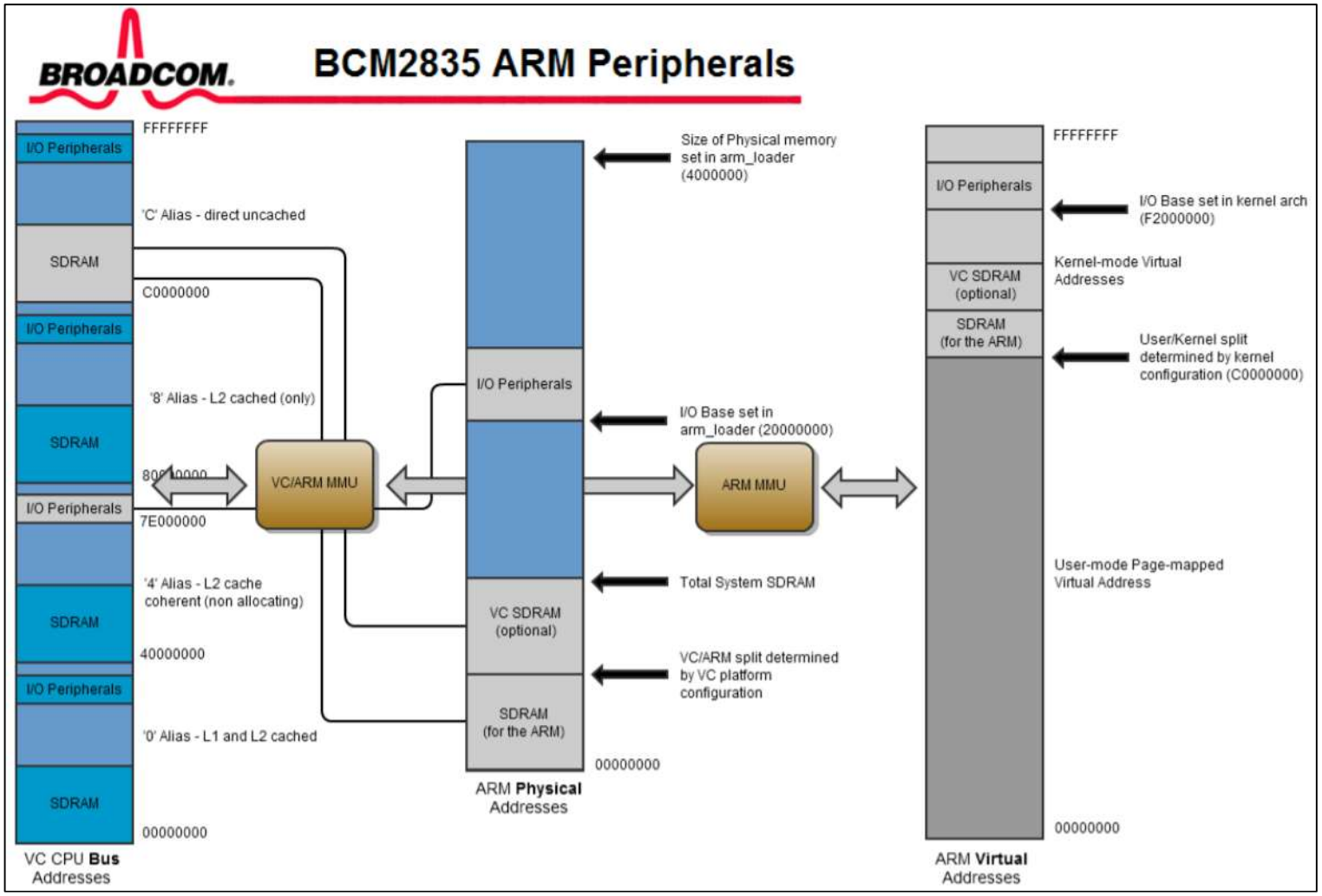
- The ARM section of the RAM starts at 0x00000000.
- The VideoCore section of the RAM is mapped in only if the system is configured to support a memory mapped display (this is the common case).

The VideoCore MMU maps the ARM physical address space to the bus address space seen by VideoCore (and VideoCore peripherals). The bus addresses for RAM are set up to map onto the uncached[1] bus address range on the VideoCore starting at 0xC0000000.

Physical addresses range from 0x20000000 to 0x20FFFFFF for peripherals. The bus addresses for peripherals are set up to map onto the peripheral bus address range starting at 0x7E000000. Thus a peripheral advertised here at bus address 0x7Ennnnnn is available at physical address 0x20nnnnnn.

- BCM2835데이터 시트에 따르면 RAM의 0x2000 0000 ~ 0x20FF FFFF에 해당하는 물리주소에 주변기기들을 다루는 레지스터의 주소(0x7E00 0000 ~ 0x7Enn nnnn)들이 MMU를 통해 매핑되어 관리되고있다.
- 우리는 레지스터의 주소에 직접접근하여 레지스터를 제어하는 것이 아니라 RAM의 물리주소를 통해 레지스터의 주소를 가상주소로 매핑하여 레지스터를 제어하는 실습을 진행한다.
- 즉, 우리가 알아야 될 RAM의 물리주소인 0x2000 0000을 알고있어야한다. //rpi4는 0xfe00 0000
  주변기기의 물리주소(0x2000 0000)에 0x0020 0000의 주소를 더하게되면 0x2020 0000이 되는데, 이 주소가 RAM내의 GPIO물리주소가 된다. //rpi4는 0xfe20 0000

MMU를 통해 매핑된 상태



실습시에 사용하는 프로그램

- gpio_dev.c : GPIO 디바이스 드라이버를 생성
- gpio_app.c : GPIO 디바이스 드라이버를 활용하는 User의 App

- 두 개의 소스코드를 사용하여 GPIO 18번에는 LED를 GPIO 19번에는 Switch를 연결하여 GPIO 디바이스 드라이버에서 각 GPIO에 레지스터를 통해 접근하여 제어하는 실습을 진행 할 것이다.

gpio_dev.c

- 헤더파일

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/io.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
```

module.h, kernel.h, init.h : 모듈프로그래밍을 할 시 필수적인 헤더파일

io.h : RAM의 물리주소를 가상주소로 매핑시켜주는 함수인 ioremap()을 참조하는 헤더파일

fs.h : 캐릭터 디바이스 드라이버를 제작할 시 필요로한 헤더파일

uaccess.h : 커널공간과 유저공간의 데이터 송수신을 위한 헤더파일
copy_to_user()함수를 참조하는 헤더파일

gpio_dev.c

- **전역변수로 선언한 변수 및 상수**

GPIO_BASE

: GPIO의 물리주소 (보드의 버전마다 물리주소가 다름)

GPFSEL1, GPSET0, GPCLR0, GPLEV0

: GPIO핀을 제어하는 레지스터들의 주소

*gpio_base

: GPIO의 물리주소를 가상주소로 매핑시켜 줄 시 매핑된 주소를 가르키는 포인터변수

*gpfsel1, *gpset0, *gpclr0, *gplev0

: GPIO핀을 제어하는 가상주소를 담는 포인터 변수

```
/*
 * 라즈베리파이B+ 이하 : 0x20200000
 * 라즈베리파이2 이상 : 0x3f200000
 * 라즈베리파이4 이상 : 0xfe200000
 *
 * 라즈베리파이2의 Soc는 BCM2836
 * BCM2836 이상의 버전부터는 GPIO레지스터를
 * 0x3f200000으로 맵핑
 * 라즈베리파이4의 Soc는 BCM2711
 * BCM2711 이상의 버전부터는 GPIO레지스터를
 * 0xfe200000으로 맵핑
 */
#define GPIO_BASE 0xFE200000

#define GPFSEL1 0x4
#define GPSET0 0x1C
#define GPCLR0 0x28
#define GPLEV0 0x34

static void __iomem *gpio_base;
volatile unsigned *gpfsel1;
volatile unsigned *gpset0;
volatile unsigned *gpclr0;
volatile unsigned *gplev0;
```

gpio_dev.c

- init() : 레지스터 가상주소 매핑, GPIO모드설정, 캐릭터 디바이스 드라이버 생성

```
int __init dev_init(void){
        int result;

        printk("%s", __FUNCTION__);

        /* # ioremap()
         * 첫번째 인자로 GPIO의 물리주소를 입력하면
         * 가상주소로 매핑된 주소를 반환시켜준다.
         * 두번째 인자는 Page_size로 0x60으로 설정한다. */
        gpio_base = ioremap(GPIO_BASE, 0x60);

        /* gpio_base에 각 레지스터를 제어하는 주소들을 더하게되면
         * GPFSEL, GPSET등의 필드에 접근가능하다. */
        gpfsel1 = (volatile unsigned *)(gpio_base + GPFSEL1);
        gpset0 = (volatile unsigned *)(gpio_base + GPSET0);
        gpclr0 = (volatile unsigned *)(gpio_base + GPCLR0);
        gplev0 = (volatile unsigned *)(gpio_base + GPLEV0);

        /* GPIO 19는 INPUT, GPIO 18은 OUTPUT으로 설정 */
        *gpfsel1 = 0b 00 000 001 000 000 000 000 000 000 000 000;

        result = register_chrdev(300, "gpio_dev", &dev_fops);
        if(result < 0)
                printk("[gpio_dev]register_chrdev failed..");
        else
                printk("[gpio_dev]register_chrdev successful..");

        return 0;
}
```

gpio_dev.c

- open() : GPIO 18 LED ON
- close() : GPIO 18 LED OFF

```
int dev_open(struct inode *inode, struct file *filp){

        /* GPSET0필드의 19번째에 해당하는 18번 GPIO를 1로 set시켜 LED ON */
        *gpset0 = 0b00000000000001000000000000000000;

        /* GPCLR0필드는 모든 GPIO들을 set하지 않은 상태로 초기화 */
        *gpclr0 = 0b00000000000000000000000000000000;

        printk(KERN_INFO "[gpio_dev]%s : LED_ON", __FUNCTION__);
        return 0;
}

int dev_close(struct inode *inode, struct file *filp){

        /* GPSET0필드의 모든 GPIO들을 set하지 않은 상태로 초기화 */
        *gpset0 = 0b00000000000000000000000000000000;

        /* GPCLR0필드의 19번째에 해당하는 18번 GPIO를 1로 set시켜 LED OFF */
        *gpclr0 = 0b00000000000001000000000000000000;

        printk(KERN_INFO "[gpio_dev]%s : LED_OFF", __FUNCTION__);

        return 0;
}
```

gpio_dev.c

- read() : GPIO 19 READ

```
ssize_t dev_read(struct file *filp, char *buf, size_t count, loff_t *f_pos){
        int result;
        static const int HIGH = 1; /* 5v에 해당하는 HIGH(1)  */
        static const int LOW = 0; /* GND에 해당하는 LOW(0) */

        /* 20번째 bit를 1로 set시켜 GPLEV0의 20번째 bit와 &연산하여 결과값을 input에 저장 */
        unsigned int input = *gplev0 & 0b00000000000010000000000000000000;

        /* input이 0을 초과하는 값이면 5v에 해당하는 HIGH를 유저(App)으로 전달
         * input이 0이하일 시에는 GND에 해당하는 LOW를 유저(App)으로 전달 */
        if(input > 0){
                result = copy_to_user(buf, &HIGH, sizeof(HIGH));
        } else {
                result = copy_to_user(buf, &LOW, sizeof(LOW));
        }

        /* 유저영역으로 값이 잘 전달됐는지 확인하는 if */
        if(result == 0){
                printk(KERN_INFO "[gpio_dev]%s : SWITCH_READ", __FUNCTION__);
                printk(KERN_INFO "[gpio_dev]input : %d", input);
        } else {
                printk(KERN_INFO "[gpio_dev]%s : SWITCH_READ Failed(%d) ", __FUNCTION__, result);
        }

        return count;
}
```

gpio_dev.c

- file_operations : 디바이스 파일이 수행할 함수들을 각 시스템 콜에 매핑
- exit() : 캐릭터 디바이스 드라이버를 커널에서 제거

```
/* 디바이스 파일에 시스템 콜이 될 시 각 시스템 콜에 해당하는  함수들을 매핑 */
static struct file_operations dev_fops = {
        .read = dev_read,
        .open = dev_open,
        .release = dev_close
};


/* 단지 캐릭터 디바이스 드라이버를 커널에서 제거하는 동작을 수행 */
void __exit dev_exit(void){
        printk("[gpio_dev]%s", __FUNCTION__);
        unregister_chrdev(300, "gpio_dev");
}
```

gpio_app.c

- gpio_app.c : GPIO 디바이스 드라이버를 활용하여 GPIO를 제어하는 App

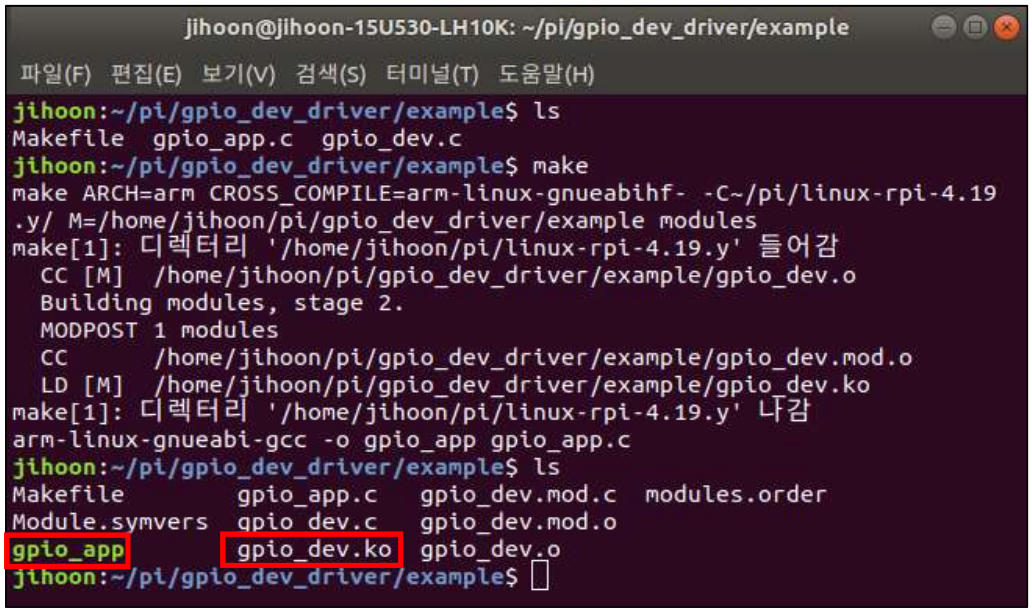
```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>

int main(void) {
        int dev;
        int value;

        /* gpio_dev파일을 Read Write옵션으로 open한다.
         * GPIO드라이버의 open()함수가 실행되어 GPIO 18번의 LED가 ON된다. */
        dev = open("/dev/gpio_dev", O_RDWR);
        printf("%d\n", dev);

        while(1){
                /* gpio_dev파일을 read한다.
                 * GPIO드라이버의 read()함수가 실행되어 GPIO 19번의
                 * 데이터 값을 읽어서 value에 저장한다. */
                read(dev, &value, sizeof(value));
                printf("%d\n", value);

                sleep(1);
        }

        /* gpio_dev파일을 close한다.
         * GPIO드라이버의 close()함수가 실행되어 GPIO 18번의 LED가 OFF된다. */
        close(dev);
        return 0;
}
```

Makefile

```
/* obj-m : 커널모듈(.ko)로 GPIO 디바이스 드라이버를 생성 */
obj-m := gpio_dev.o

/* KDIR에는 라즈베리파이의 커널소스 경로를 넣어주면 된다. */
KDIR := # Please, insert the Rpi kernel directory path

/* gpio_app을 생성하기 위한 컴파일러를 지정
 * 라즈베리파이 B+ 이하 : arm-linux-gnueabi-gcc
 * 라즈베리파이 2 이상 : arm-linux-gnueabihf-gcc  */
CC := arm-linux-gnueabi-gcc
OBJECT := gpio_app

/* default : make시 수행할 명령어모음 */
default:
        make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -C$(KDIR) M=$(shell pwd)
modules
        $(CC) -o $(OBJECT) $(OBJECT).c

/* clean : make clean시 수행할 명령어모음 */
clean:
        make -C$(KDIR) M=$(shell pwd) clean
        rm $(OBJECT)
```

GPIO 디바이스 드라이버 실습 실행

- make 실행



- gpio_dev_driver.zip파일을 zip명령어를 사용하여 압축을 푼 뒤example디렉터리에서 make명령어를 수행시켜서 gpio_app파일과 gpio_dev.ko파일을 생성시킨다.

- 만약, 라즈베리파이2이상 버전의 보드를 사용하여 실습을 진행할 경우에는 Makefile내용 중 컴파일러(CC)를 arm-linux-gnueabihf-gcc로 변경하고 make를 실시.

GPIO 디바이스 드라이버 실습 실행

- ssh을 이용하여 Rpi에 전송

```
jihoon:~/pi/gpio_dev_driver/example$ ls
Makefile        gpio_app.c   gpio_dev.mod.c  modules.order
Module.symvers  gpio_dev.c   gpio_dev.mod.o
jihoon:~/pi/gpio_dev_driver/example$ scp gpio_app gpio_dev.ko pi@192.255.255.2:/home/pi
pi@192.255.255.2's password:
gpio_app                                      100% 8336     1.3MB/s   00:00
gpio_dev.ko                                   100% 6900     1.4MB/s   00:00
```

- gpio_app, gpio_dev.ko파일들을 scp명령어를 이용하여 라즈베리파이로 전송시킨다.

- scp명령어 형식

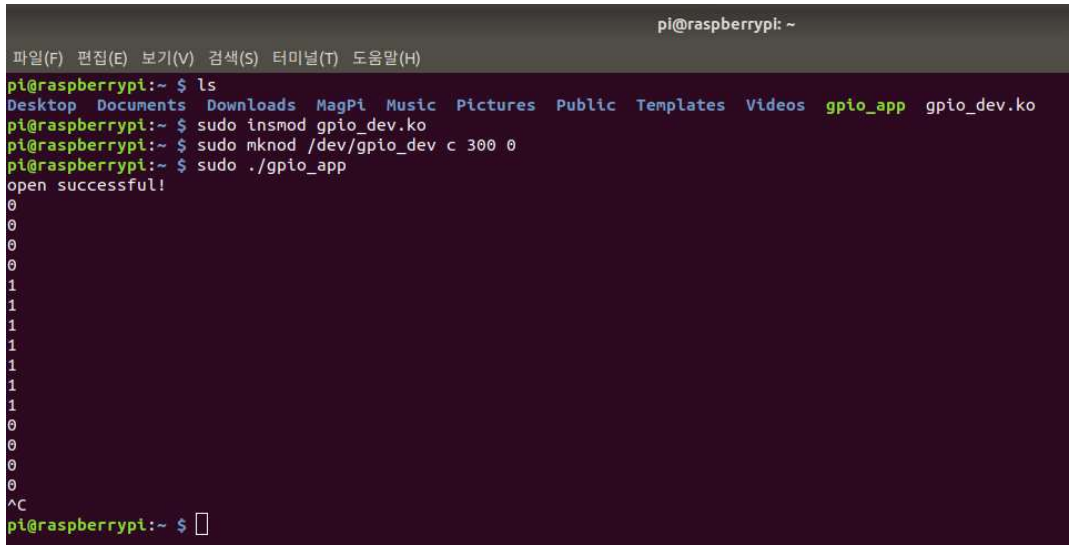
  scp [파일명] [target_user명]@[target_user_ip]:[target_user_디렉터리]

- ssh로 Host PC에서 라즈베리파이에 원격제어를 실시
- ssh명령어 형식

  ssh [target_user명]@[target_user_ip]

GPIO 디바이스 드라이버 실습 실행

- 모듈을 커널에 등록, 디바이스 파일 생성, gpio_app실행



✓ 이 후의 명령어는 라즈베리파이에서 실행

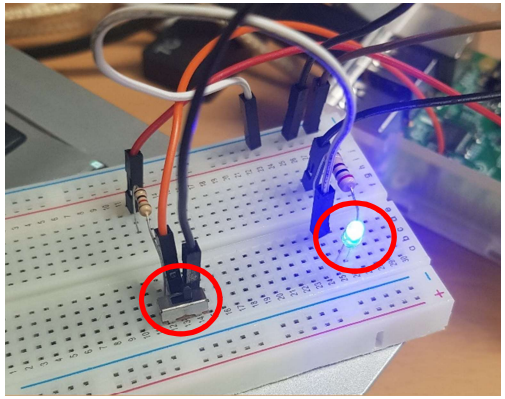
✓ $ sudo insmod gpio_dev.ko : 커널에 GPIO 디바이스 드라이버 모듈을 적재

✓ $ sudo mknod /dev/gpio_dev c 300 0 : GPIO 디바이스 드라이버가 사용할 디바이스 파일 생성

✓ $ sudo ./gpio_app : gpio_app프로그램을 실행하여 GPIO 18, 19핀들을 제어

GPIO 디바이스 드라이버 실습 실행

✓ 동작사진1



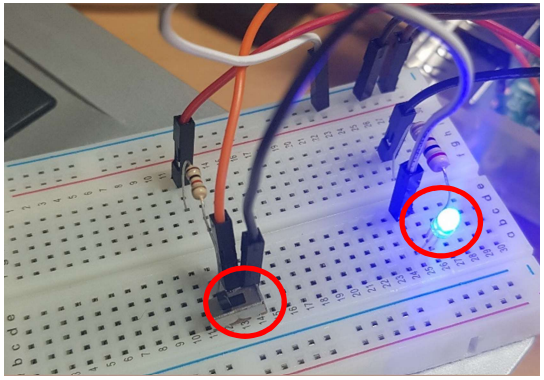
✓ 터미널상태

```
pi@raspberrypi:~ $ sudo ./gpio_app
open successful!
0
0
0
0
```

- gpio_app실행직후 모습
- 디바이스 파일이 open되어 GPIO 18번에 HIGH값을 출력하여 LED ON이 된 상태
- 스위치는 오른쪽을 향해 있으며 GPIO 19번에서는 GND(LOW)가 입력되어 터미널에 0이 출력되는 상태

GPIO 디바이스 드라이버 실습 실행

✓ 동작사진2



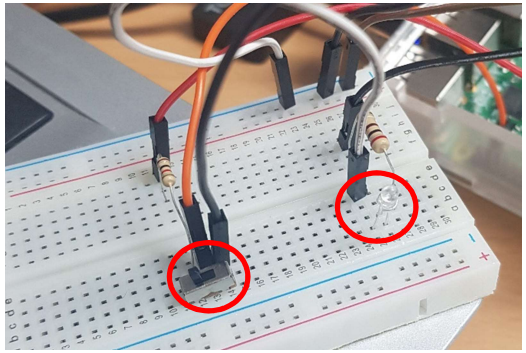
✓ 터미널상태

```
pi@raspberrypi:~ $ sudo ./gpio_app
open successful!
0
0
0
0
1
1
1
1
1
1
```

- 스위치가 왼쪽을 향하여 GPIO 19번에 5v(HIGH)가 입력되고 있는 모습
- 계속해서 gpio_app은 디바이스 파일에 read를 실행하고 있으며, 계속해서 GPIO 19번에서 값을 읽어오고 있다.

GPIO 디바이스 드라이버 실습 실행

✓ 동작사진3



✓ 터미널상태

```
pi@raspberrypi:~ $ sudo ./gpio_app
open successful!
0
0
0
0
1
1
1
1
1
1
1
^C
```

- 터미널에 Ctrl+C를 통해 SIGINT를 발생시켜 gpio_app프로세스를 종료
- 종료 이 후에는 디바이스 파일이 close됨으로 GPIO 18번에 연결된 LED OFF
- 스위치는 왼쪽으로 향해있는 상태이지만 프로세스가 종료되어 더 이상 드라이버가 값을 입력받지 못하는 상태가 됨

E N D