

임베디드 리눅스 시스템 프로그래밍

Chapter7

라즈베리파이

챕터의 포인트

- 라즈베리파이
- 개발환경
- GPIO

라즈베리파이

목표

라즈베리파이를 이용한 H/W 제어 방법에 대해 학습한다.
디지털 신호를 이용한 장치들을 제어한다.

라즈베리파이 활용

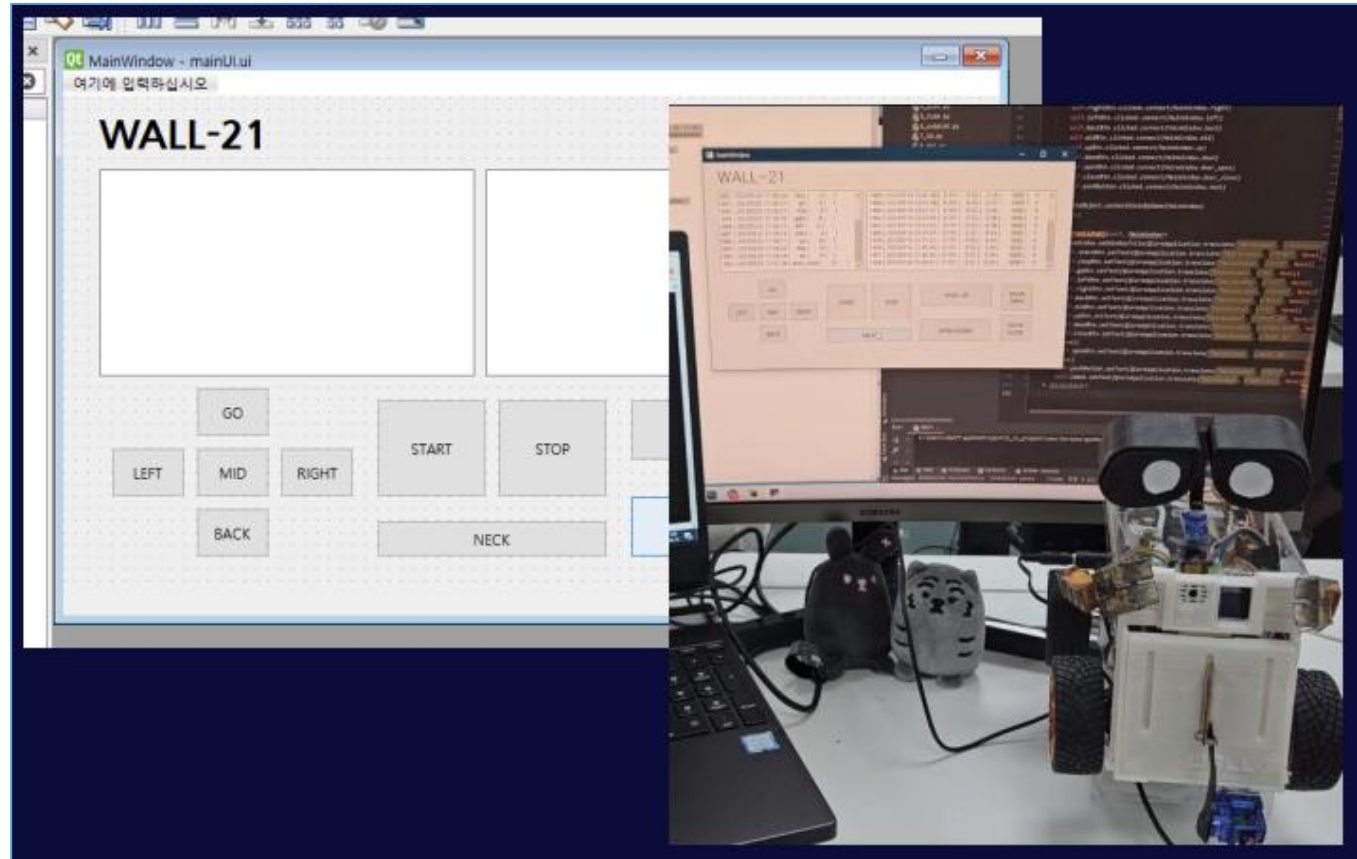
라즈베리파이는 다양한 프로젝트와 개발자들의 입문용 SBC 이다.

Single Board Computer

카메라를 활용한 OpenCV 프로젝트

통신을 이용한 IoT 프로젝트

저렴한 NAS용 보드 등



개발환경

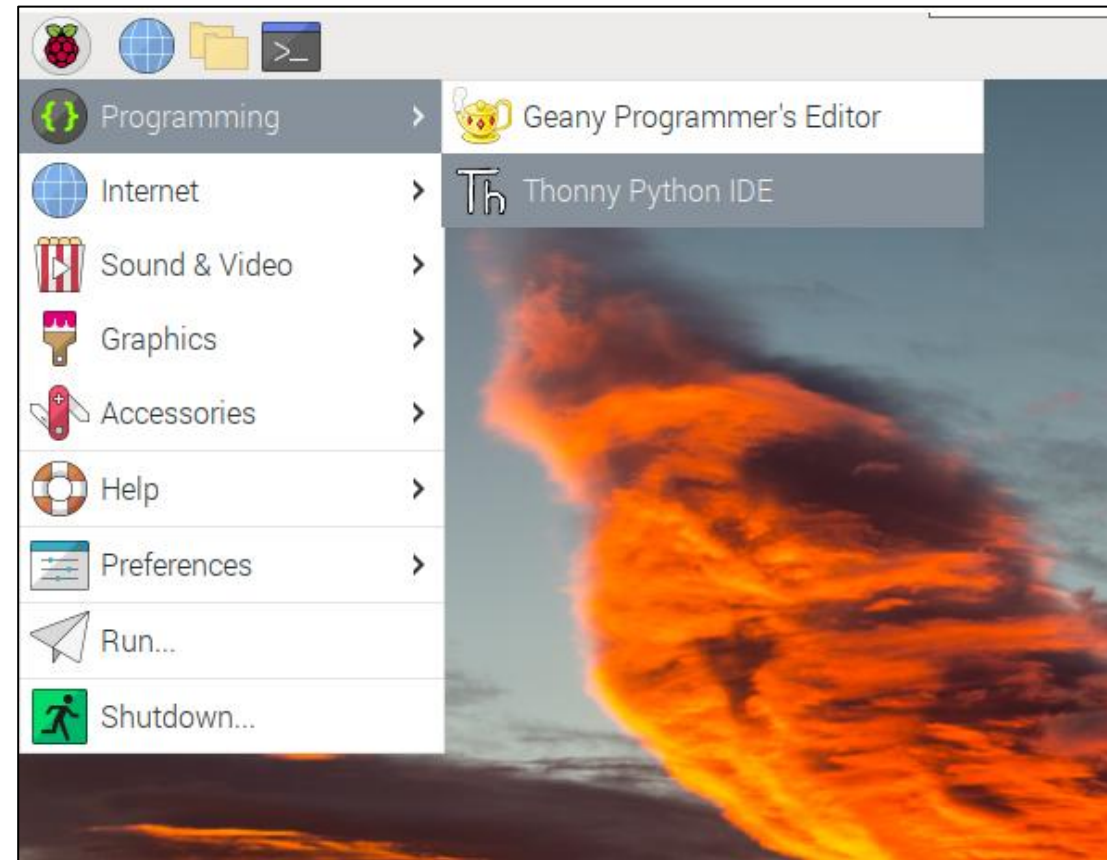
개발환경

수업에서는 Python 과 Thonny IDE 를 사용한다.

언어 : Python

IDE : Thonny Python IDE (RPI4 내포)

물론, MobaXterm 으로
.py 파일 만들어서
실행해도 된다.



라즈베리파이에서 H/W 제어를 하려면?

1. 쉽게 사용할 수 있는 Wrapping된 함수를 사용해서 GPIO 제어하기
2. 리눅스 Device Driver 사용해서, GPIO 제어하기

라즈베리파이의 공식 Library

1. gpiozero

<https://gpiozero.readthedocs.io/en/stable/>

본 교안은 GPIO Zero의 Document를 참고하여 제작

2. RPi.GPIO

gpiozero 보다 더 세부적인 설정 가능

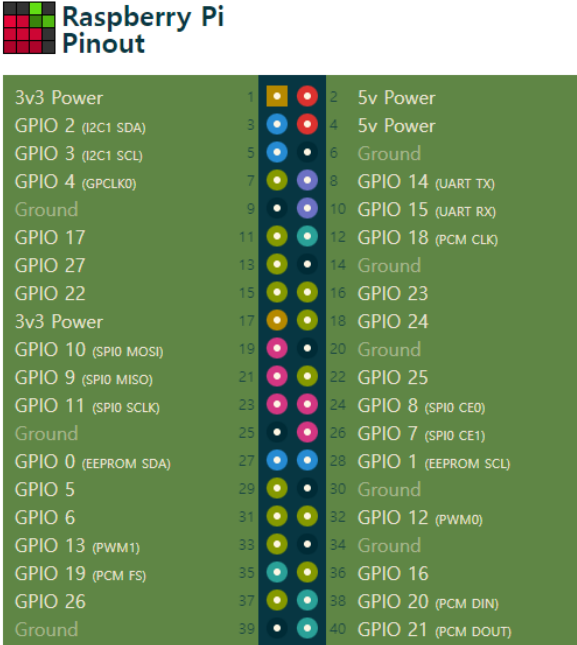
gpiozero 는 RPi.GPIO 를 Wrapping 한 Library

<https://sourceforge.net/p/raspberry-gpio-python/wiki/BasicUsage/>

라즈베리파이 GPIO Pin GUIDE

<https://pinout.xyz/>

이 사이트를 참고하여 배선한다.



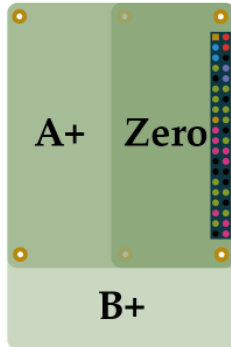
Raspberry Pi Pinout

Pin	Function	Pin	Function
1	3v3 Power	21	GPIO 9 (SPI MISO)
2	GPIO 2 (I2C1 SDA)	22	GPIO 25
3	GPIO 3 (I2C1 SCL)	23	GPIO 11 (SPI SCLK)
4	GPIO 4 (GPCLK0)	24	GPIO 8 (SPI CE0)
5	Ground	25	GPIO 7 (SPI CE1)
6	GPIO 17	26	GPIO 1 (EEPROM SCL)
7	GPIO 27	27	Ground
8	GPIO 22	28	Ground
9	3v3 Power	29	GPIO 5
10	GPIO 10 (SPI MOSI)	30	GPIO 6
11	GPIO 9 (SPI MISO)	31	GPIO 13 (PWM1)
12	GPIO 11 (SPI SCLK)	32	GPIO 19 (PCM FS)
13	Ground	33	GPIO 26
14	GPIO 0 (EEPROM SDA)	34	Ground
15	GPIO 5	35	GPIO 16
16	GPIO 6	36	GPIO 20 (PCM DIN)
17	GPIO 13 (PWM1)	37	GPIO 21 (PCM DOUT)
18	GPIO 19 (PCM FS)	38	
19	GPIO 26	39	
20	Ground	40	

Legend

Orientate your Pi with the GPIO on the right and the HDMI port(s) on the left.

- GPIO (General Purpose IO)
- SPI (Serial Peripheral Interface)
- I²C (Inter-integrated Circuit)
- UART (Universal Asynchronous Receiver/Transmitter)
- PCM (Pulse Code Modulation)
- Ground
- 5V (Power)
- 3.3V (Power)



A+ Zero B+

Pinout!

The Raspberry Pi GPIO pinout guide.

This GPIO Pinout is an interactive reference to the Raspberry Pi GPIO pins, and a guide to the Raspberry Pi's GPIO interfaces. Pinout also includes [hundreds of pinouts for Raspberry Pi add-on boards, HATs and pHATs.](#)

Support Pinout.xyz

If you love Pinout, please help me fund new features and improvements:

- via GitHub at [GitHub.com/sponsors/gadgetoid](https://github.com/sponsors/gadgetoid)
- via Patreon at [Patreon.com/gadgetoid](https://patreon.com/gadgetoid)

Every \$1 makes all the difference! Thank you.

pHAT Stack

Pinout has teamed up with Pimoroni to create a [prototype board compatibility tool](#), [check it out here!](#)

Explore HATs & pHATs

[Check out Pinout's board explorer!](#) Use it to find the pinout for your Raspberry Pi add-on board, or discover new boards. If you manufacture boards, we'd love to add yours too. [You can contribute to Pinout.xyz at GitHub.com.](#)

What do these numbers mean?

- GPIO - General Purpose Input/Output, aka "BCM" or "Broadcom". These are the big numbers, e.g. "GPIO 22". You'll use these with RPi.GPIO and GPIO Zero.
- Physical - or "Board" correspond to the pin's physical location on the header. These are the small numbers next to the header, e.g. "Physical Pin 15".

오늘 수업을 위한 장치

LED, Tact 스위치, 220옴 저항, 10k옴 저항
브레드보드와 점퍼케이블은 default 이다.



220옴

10k옴



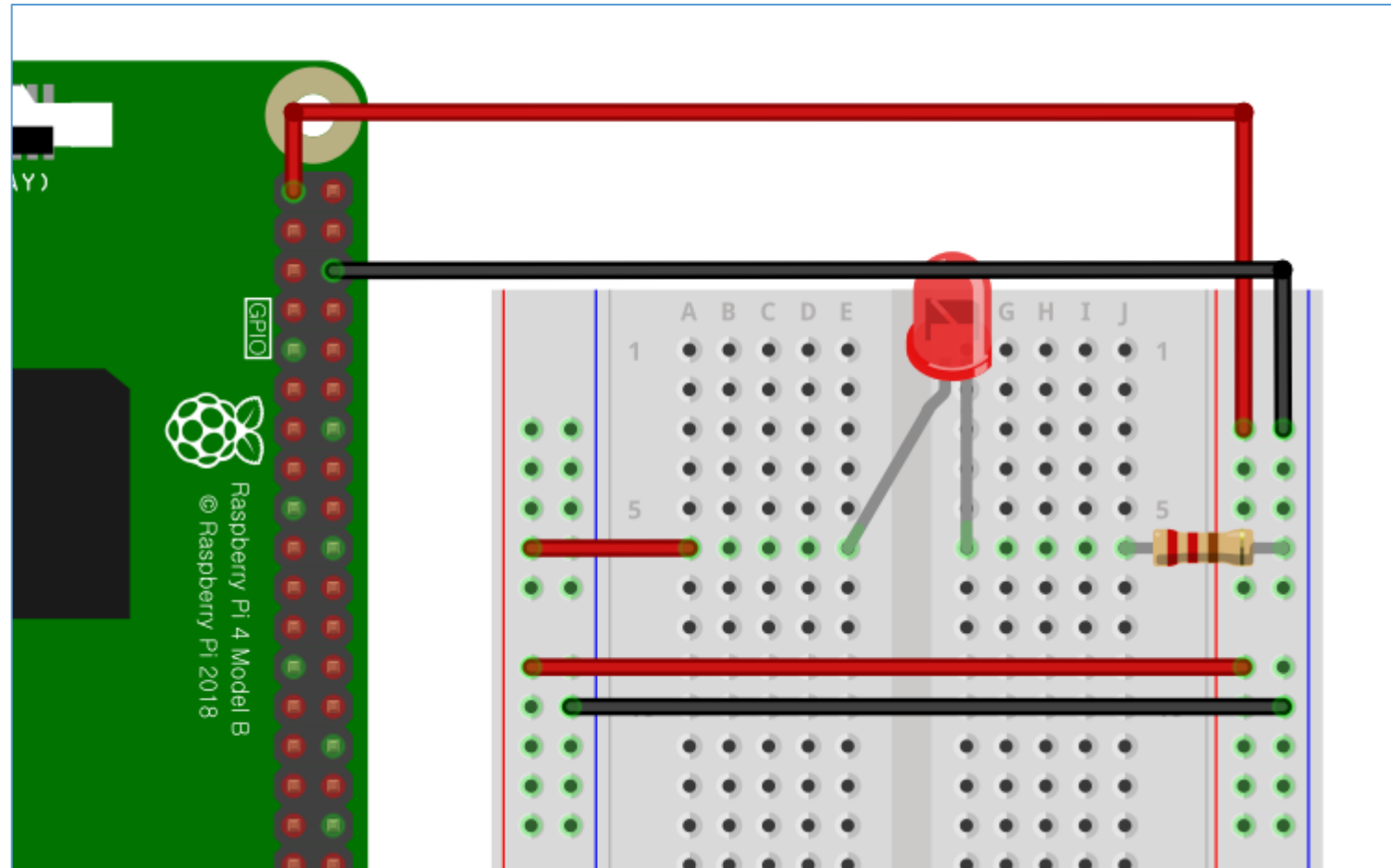
회로 연결하기

회로를 연결한다.

220옴 저항을 사용한다.

3.3V

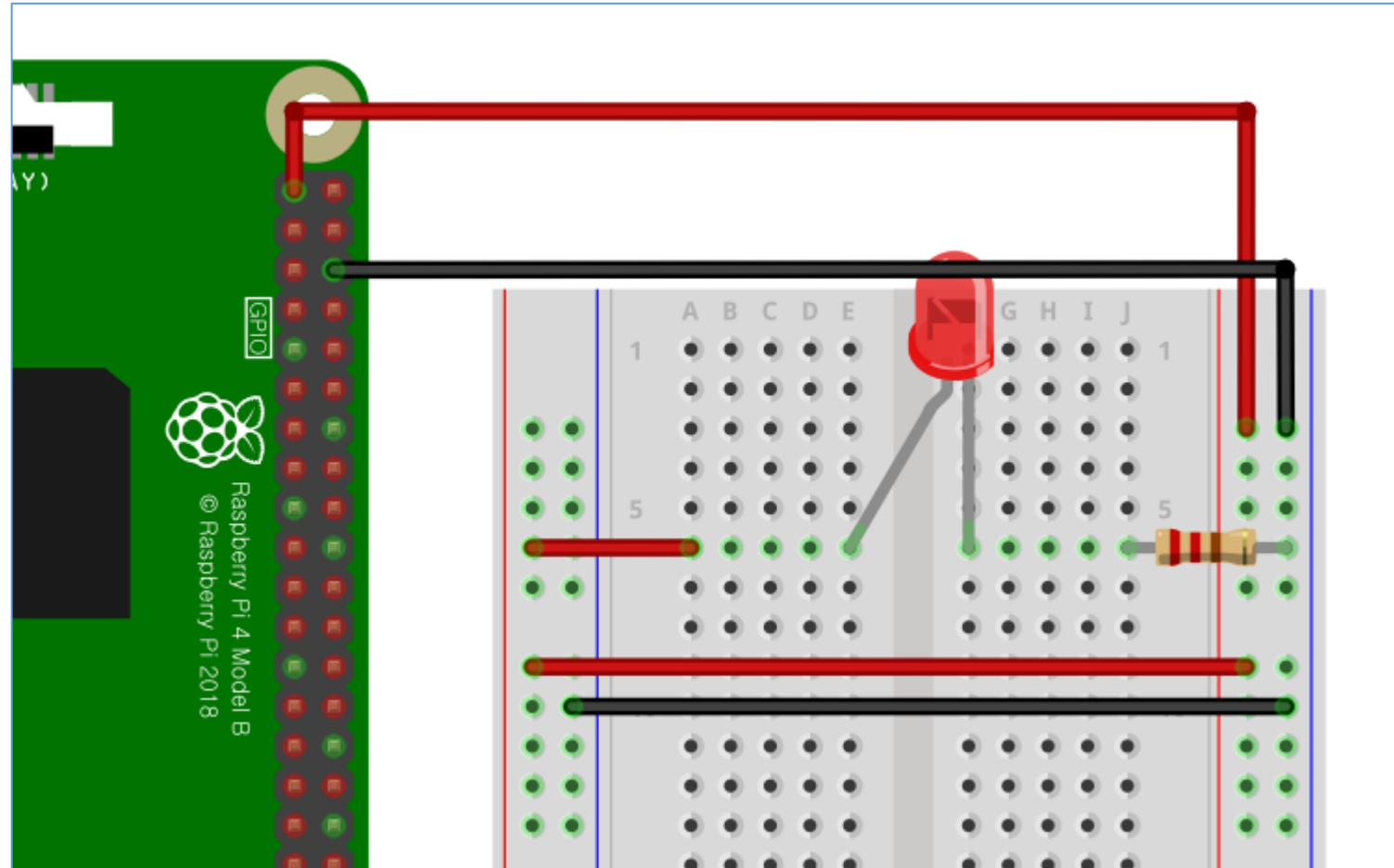
GND



LED 디버깅으로 활용하기

LED는 디버깅용도로 많이 사용 된다.

기기가 동작 중인지 User 에게 알려주는 역할을 한다.

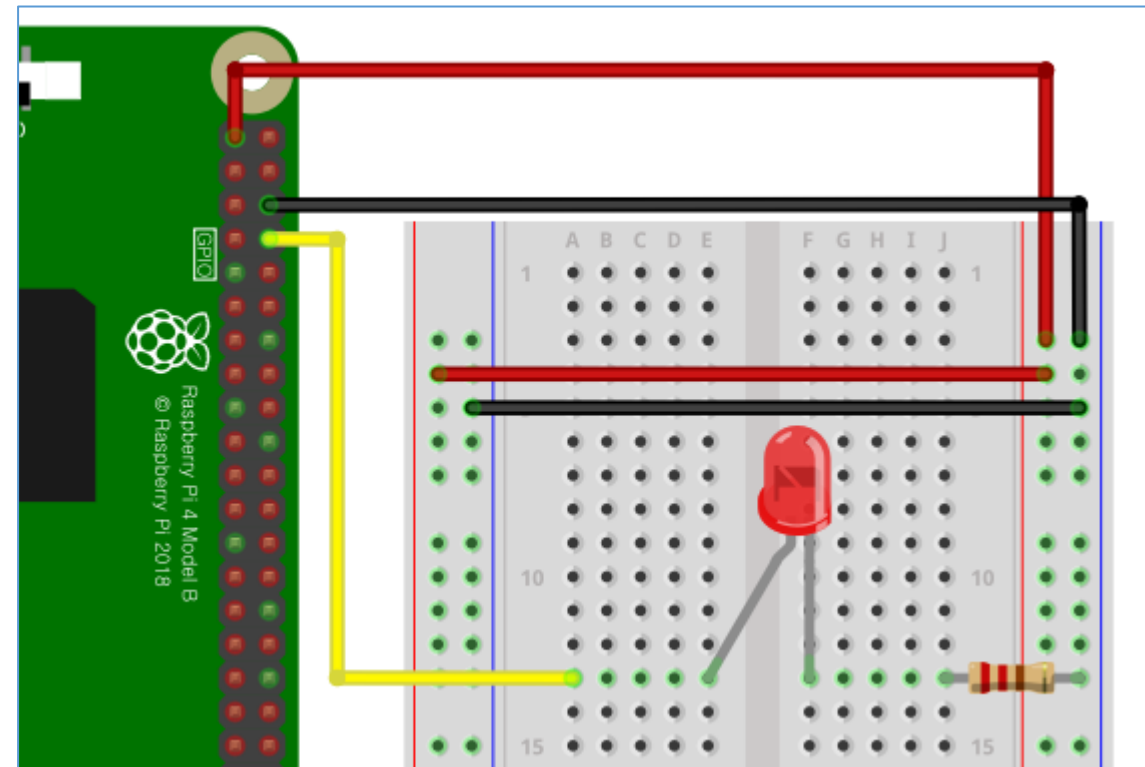


LED를 연결한다. (220옴 저항 필요)

(+) → GPIO14

(-) → GND

참고) LED의 + 극을 살짝 구부려서 연결하면, 회로를 디버깅하는 데 도움이 된다.



코드를 작성하고 빌드한다.

~/test 디렉토리 생성
led.py

LED 깜빡이는 코드

LED(pin) : LED 객체 생성
.on() : HIGH 신호를 보낸다.
.off() : LOW 신호를 보낸다.

```
1  from gpiozero import LED
2  from time import sleep
3
4  red = LED(14) #GPIO pin number
5
6  while True:
7      red.on()
8      sleep(1)
9      red.off()
10     sleep(1)
```

<https://gist.github.com/hoconoco/2e097c7f4a8bf3721a901a46254344db>

[참고] 회로 디버깅

항상 3.3V 와 GND는 먼저 연결한다.

모듈마다 전기가 통할 경우, LED에 불이 들어오는 장치들이 있다.

사용하지 않는 모듈은 분리한다. 또는 GND 핀을 빼둔다.

GND핀은 빼둬도, 전기를 내보내지 않으니 다른 장치에 영향을 주지 않는다.

스위치

전류의 흐름을 막거나 흐르게 하는 용도
다양한 종류가 있다.

1. Tact 스위치
2. 로커 스위치
3. Push 스위치



키트에 있는 스위치 종류

다양한 스위치가 있다.

1. 로터리 스위치 (회전식, 선택 스위치)
2. 조이스틱
3. Tact 스위치

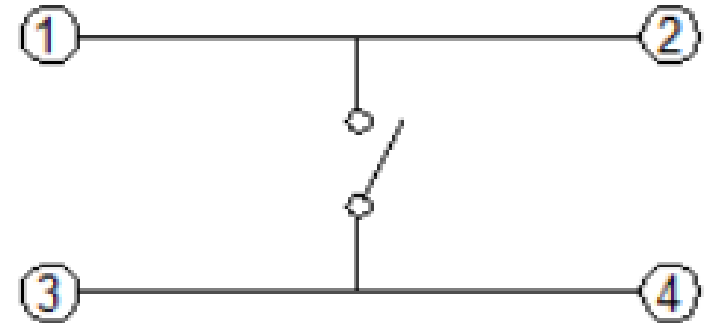


Tact 스위치 특징

Tact 스위치는 네 개의 다리가 있다.
극성은 없다.
가로로 두 개의 다리가 연결되어 있다.
누르면 모든 다리가 연결된다.



CIRCUIT DIAGRAM

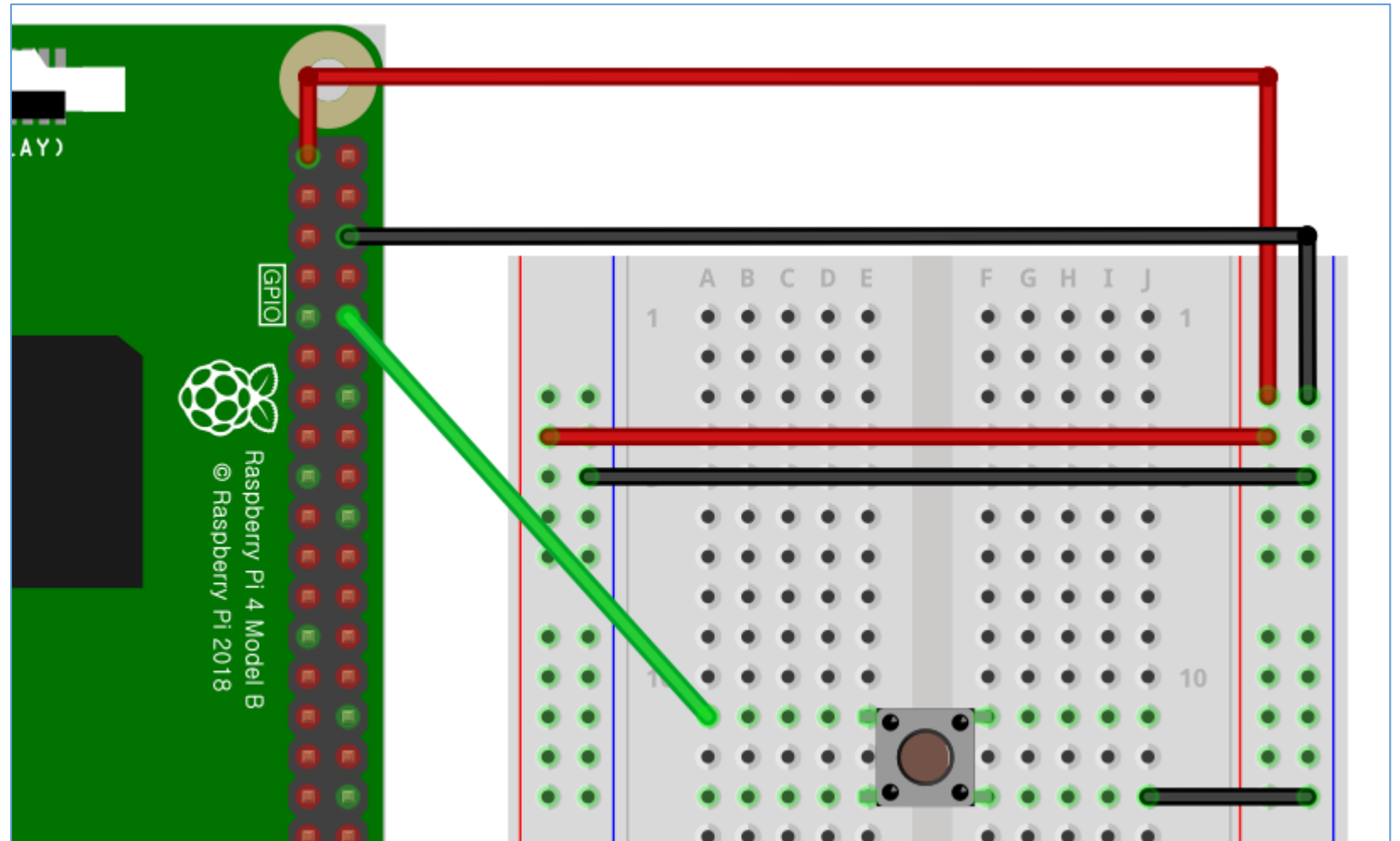


회로 연결

버튼을 연결한다.

GPIO15

GND



코드를 작성한 뒤 빌드한다.

~/test2 디렉토리 생성
switch_polling.py

버튼을 눌렀을 때, 동작하는 소스코드
polling 방식

Button(pin num) : 버튼 객체 생성

.is_pressed : 버튼 눌림 감지, True / False return

```
1  from gpiozero import Button
2
3  btn = Button(15)
4
5  while True:
6      if btn.is_pressed:
7          print('ON')
8      else:
9          print('OFF')
```

<https://gist.github.com/hoconoco/d977607ee222145a8245db086021825f>

코드를 작성한 뒤 빌드한다.

~/test3 디렉토리 생성
switch_interrupt.py

버튼을 눌렀을 때, 동작하는 소스코드

interrupt 방식 (이라 하지만, 내부적으로 Threading 동작)

.when_pressed : 버튼 눌리면, callback 함수 호출

.when_released : 버튼 눌리지 않으면, callback 함수 호출

pause() : 프로세스 종료 or 시그널 종료 받을 때까지 대기

Python Signal 라이브러리를 이용해서 인터럽트 방식 구현

- <https://docs.python.org/ko/3/library/signal.html>

```
1  from gpiozero import Button
2  from signal import pause
3
4  def press():
5      print("Btn Pressed!")
6
7  def release():
8      print("Btn Released!")
9
10 btn = Button(15)
11
12 btn.when_pressed = press
13 btn.when_released = release
14
15 pause()
```

<https://gist.github.com/hoconoco/eba87e0a92845fe1ce59f00fcc0d96bb>

Polling vs Interrupt 테스트 해보기

Polling / Interrupt 소스코드

```
from gpiozero import Button

button = Button(2)

while True:
    if button.is_pressed:
        print("Button is pressed")
    else:
        print("Button is not pressed")
```

```
from gpiozero import Button
from signal import pause

def say_hello():
    print("Hello!")

def say_goodbye():
    print("Goodbye!")

button = Button(2)

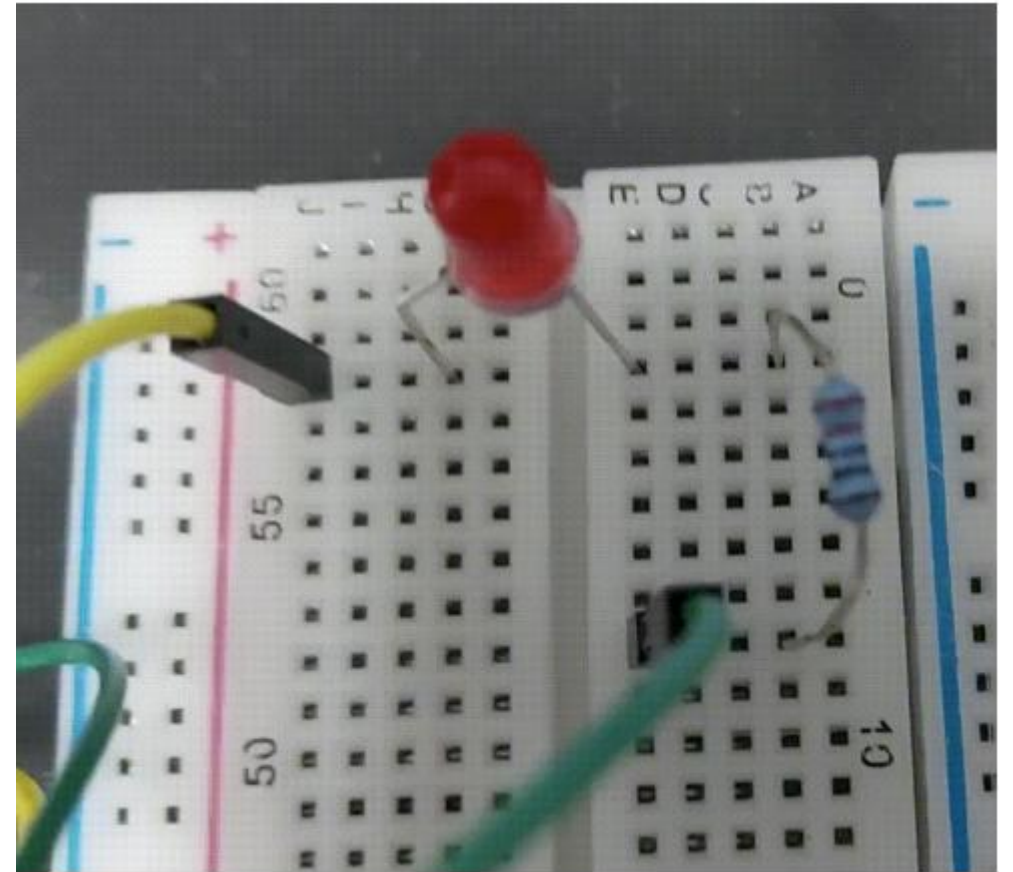
button.when_pressed = say_hello
button.when_released = say_goodbye

pause()
```


[도전1] 직접 짜보기

깜박임이 점차 빨라지도록 하기

sleep **1초 ~ 10ms** 까지 점차 빨라지도록



[도전 2] LED 제어 쉘 제작

다음 기능을 만족하는 LED 제어 쉘 제작

H/W : LED 3개, 220옴 저항 3개

만약 on 상태라면 off로, off 상태라면 on 으로 변경

- 1 입력 시 : 1번 LED ON 또는 OFF
- 2 입력 시 : 2번 LED ON 또는 OFF
- 3 입력 시 : 3번 LED ON 또는 OFF
- 그 외 숫자 입력 시 : 종료

```
INPUT>> 1
INPUT>> 1
INPUT>> 1
INPUT>> 1
INPUT>> 2
INPUT>> 2
INPUT>> 3
INPUT>> 3
INPUT>>
```

Chapter8

Linux Driver

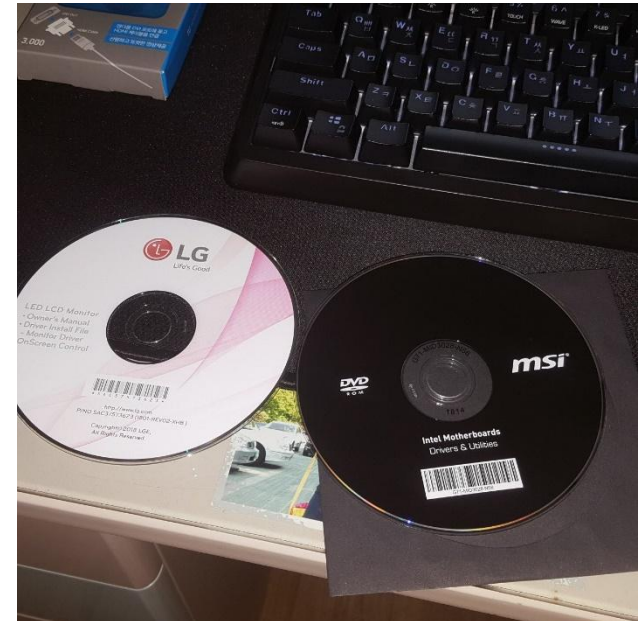
챕터의 포인트

- Device Driver 개념
- Device Driver 필요성
- Device Driver 개발과정
- 예시 코드

Device Driver 개념

디바이스 드라이버는 주변에 항상 있다.

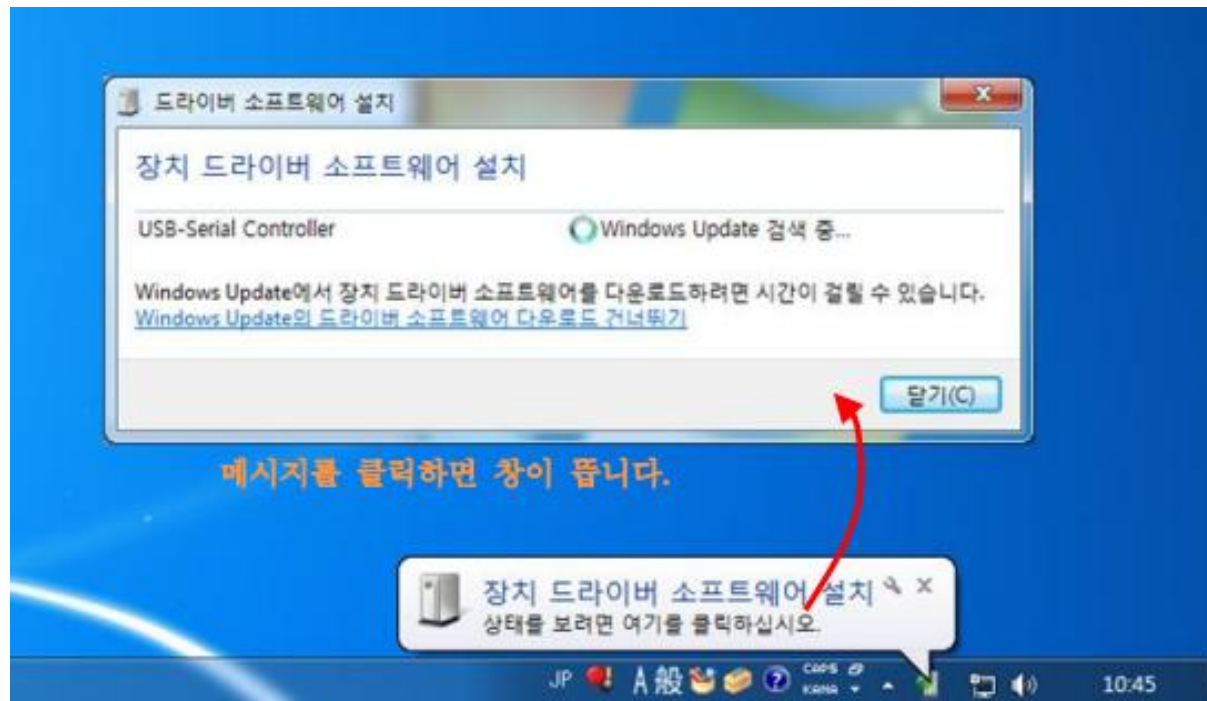
USB 저장장치를 쏘을 때 뜨는 메시지
모니터 / 프린터 / 공유기 구매시 항상 들어있는 CD



디바이스 드라이버는 주변에 항상 있다.

USB 저장장치를 꽂을 때 뜨는 메시지

모니터 / 프린터 / 공유기 구매시 항상 들어있는 CD



Device Driver 정의

사전적 정의

Program이 HW를 제어하기 위한 SW를 뜻함

Software Interface를 통해 Application이 HW Spec을 이해하지 않아도 되는 장점

개념적인 한 문장으로 Device Driver를 이해하기 힘들

이것이 필요했던 스토리를 알아야
디바이스드라이버의 정의를 이해할 수 있음

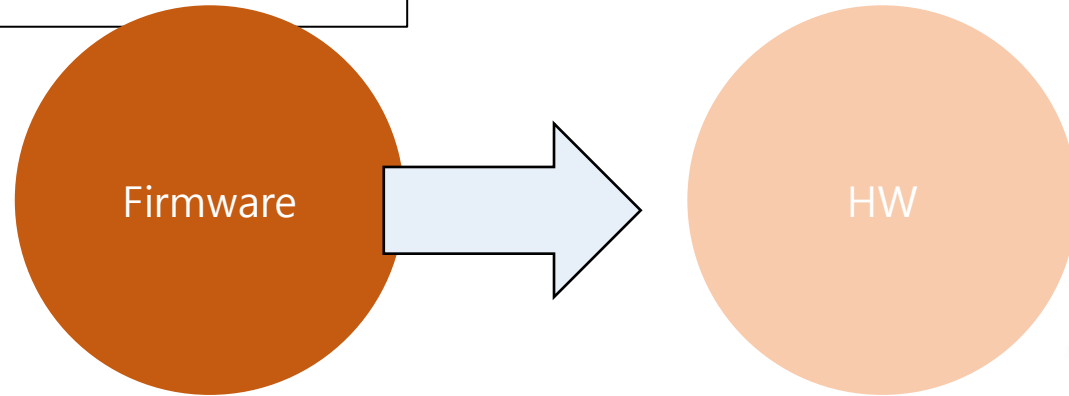
Device Driver 필요성

Firmware에서 임베디드 개발

HW 메모리 맵 Address에 직접 값 Access 가능

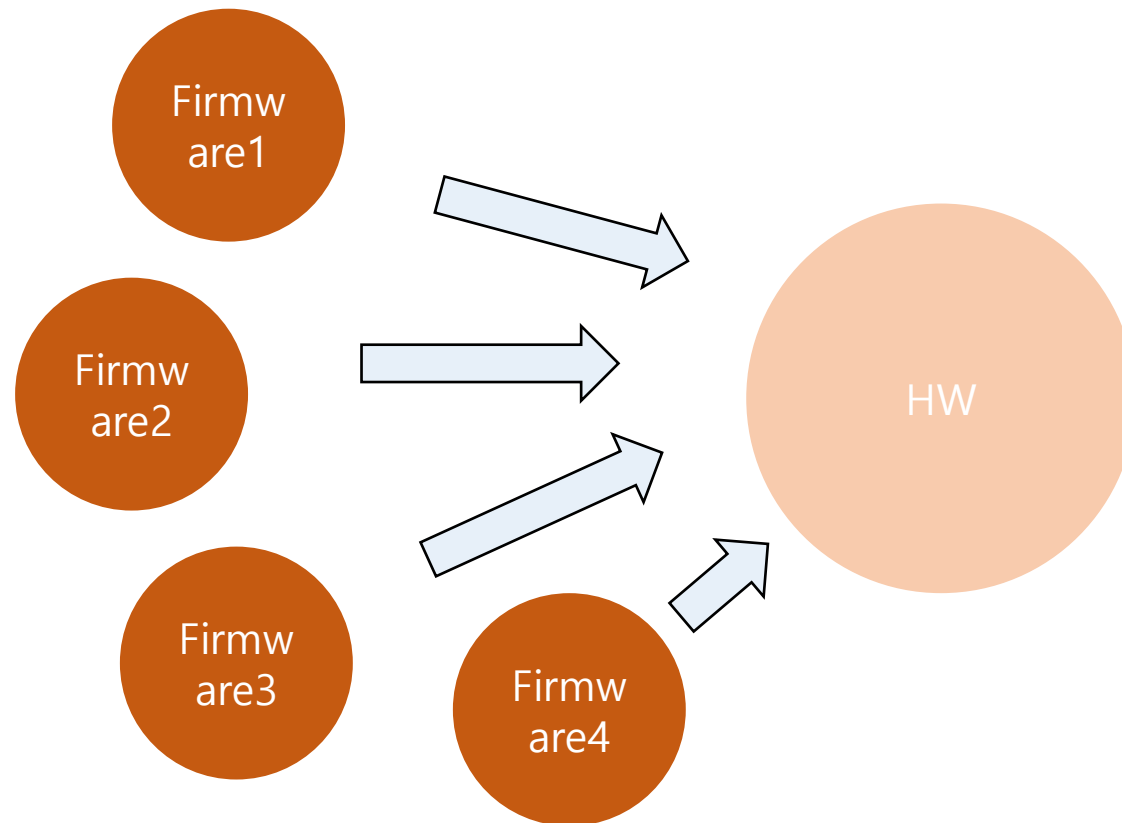
Application 이 HW를 직접 제어 한다. (중간 Layer가 없음)

```
int main(void)
{
    (*(volatile unsigned*)0x40021018) |= 0x8;
    (*(volatile unsigned*)0x40010C04) |= 0x10;
}
```



디바이스 드라이버 필요성

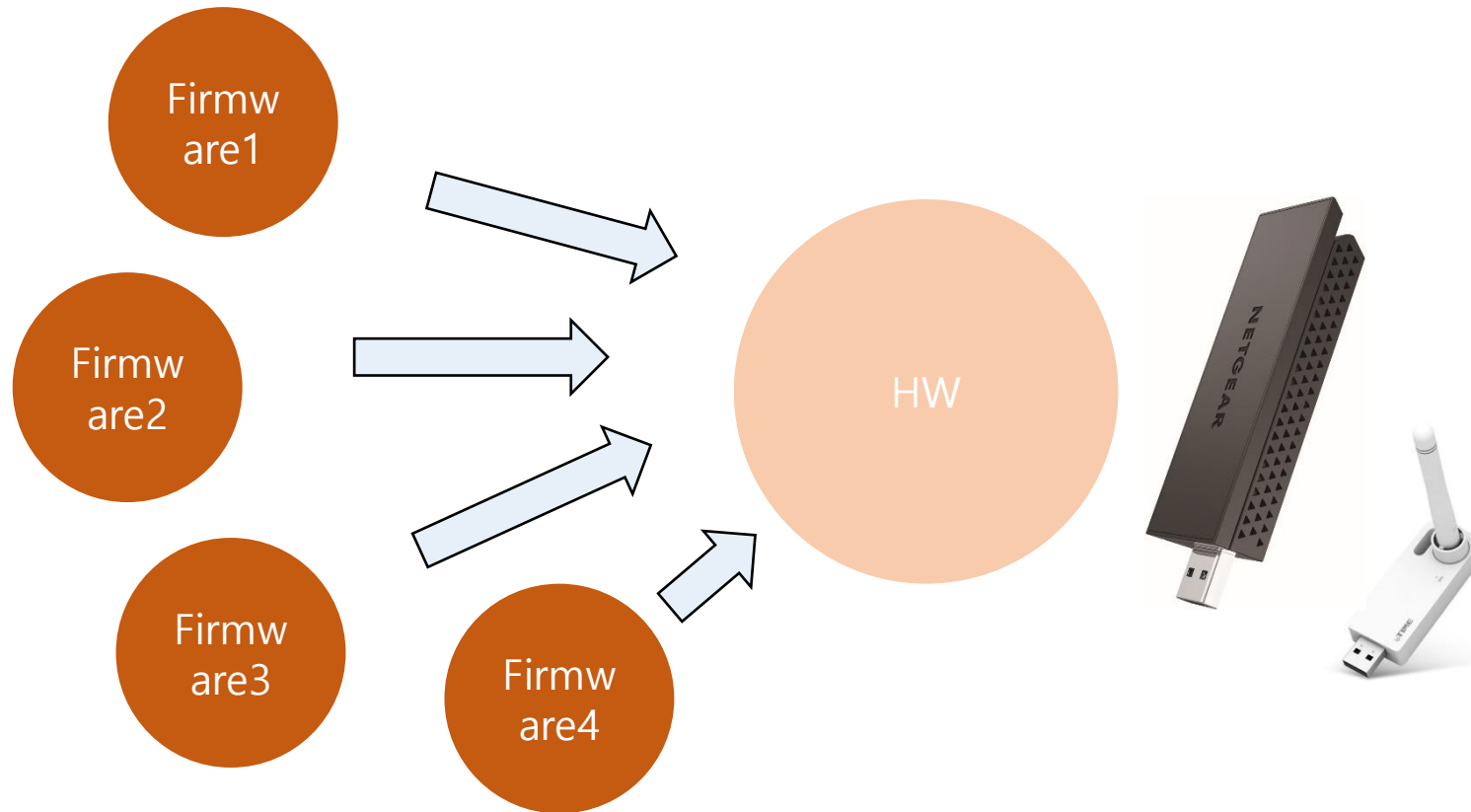
HW를 사용하는 여러개의 Firmware들을 개발



디바이스 드라이버 필요성

만약 HW 장치를 교체한다면 해야되는 일

모든 Firmware의 HW 관련 코드들을 수정하여, 모두 다시 Firmware 다운로드 후 실행해야 함

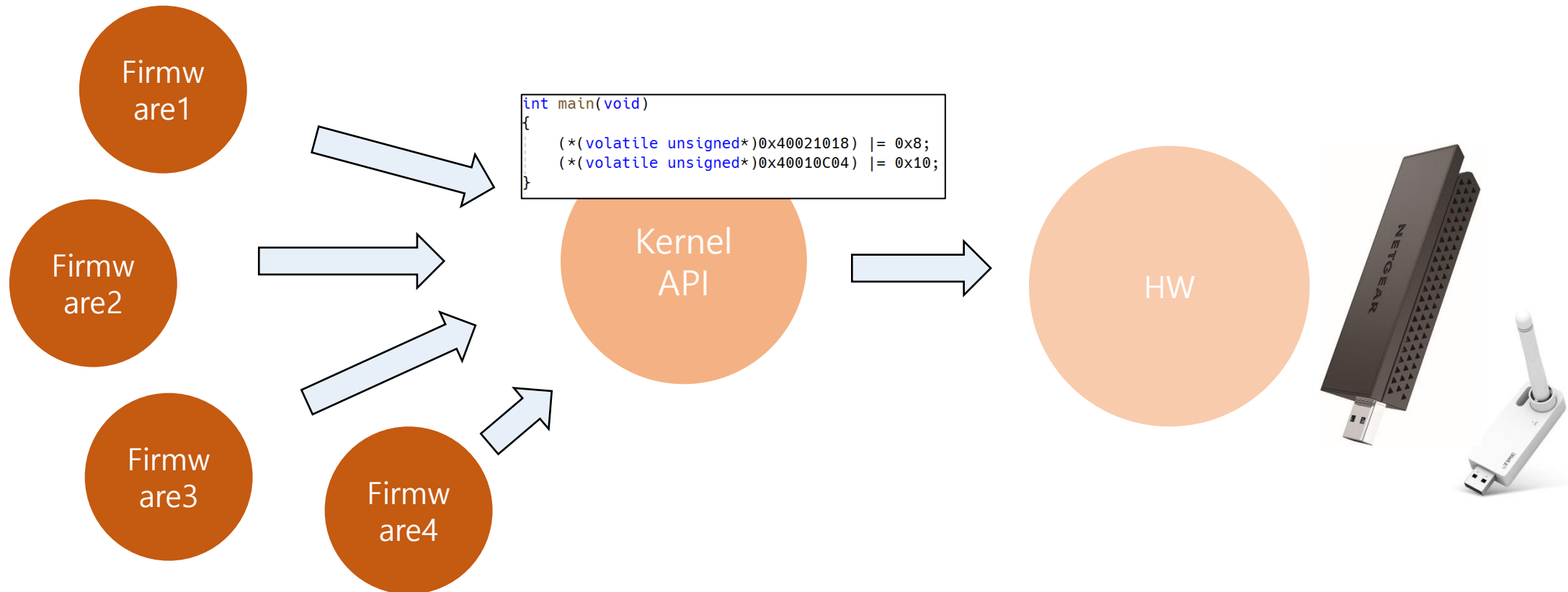


디바이스 드라이버 필요성

Layer의 편리함

Kernel은 공통적으로 쓰는 API를 제공

Kernel 소스코드만 새로운 HW가 동작되도록 수정하여 다시 Build하면, 다른 Firmware들 수정 필요 없음

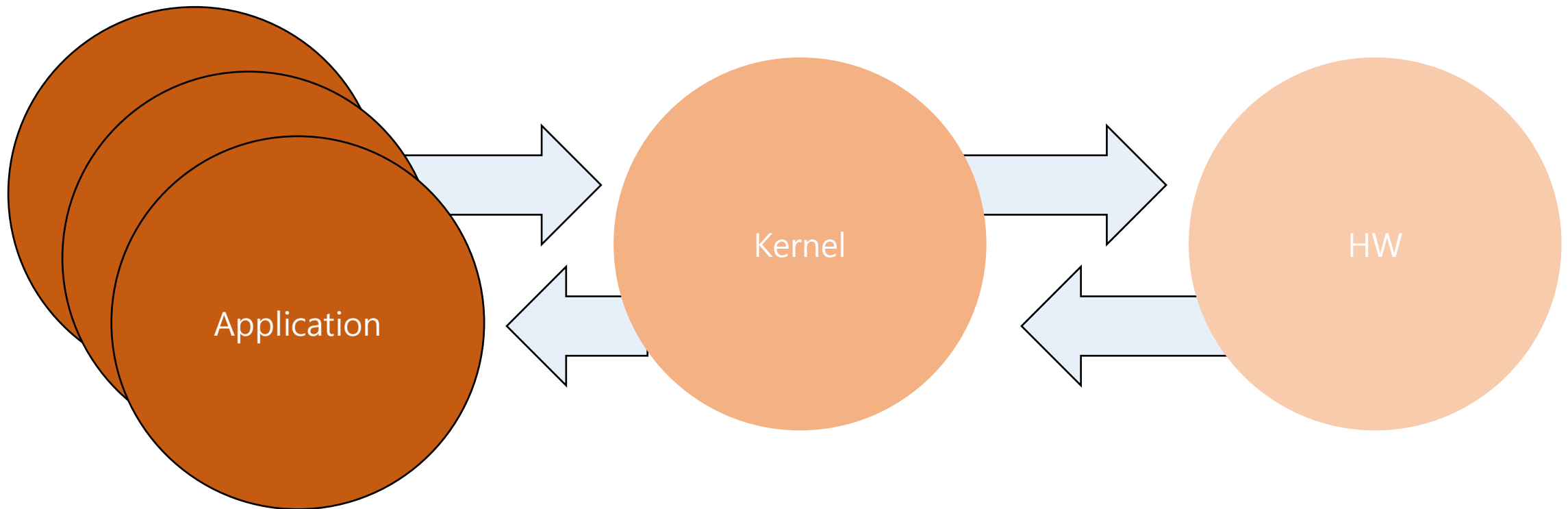


디바이스 드라이버 필요성

Application 과 HW 사이에 계층을 두었음

App은 커널의 API(System Call)을 통해 H/W 접근이 가능하다.

HW에 대한 지식이 없어도, 커널 API를 통해 H/W 제어하는 Application 제작 가능

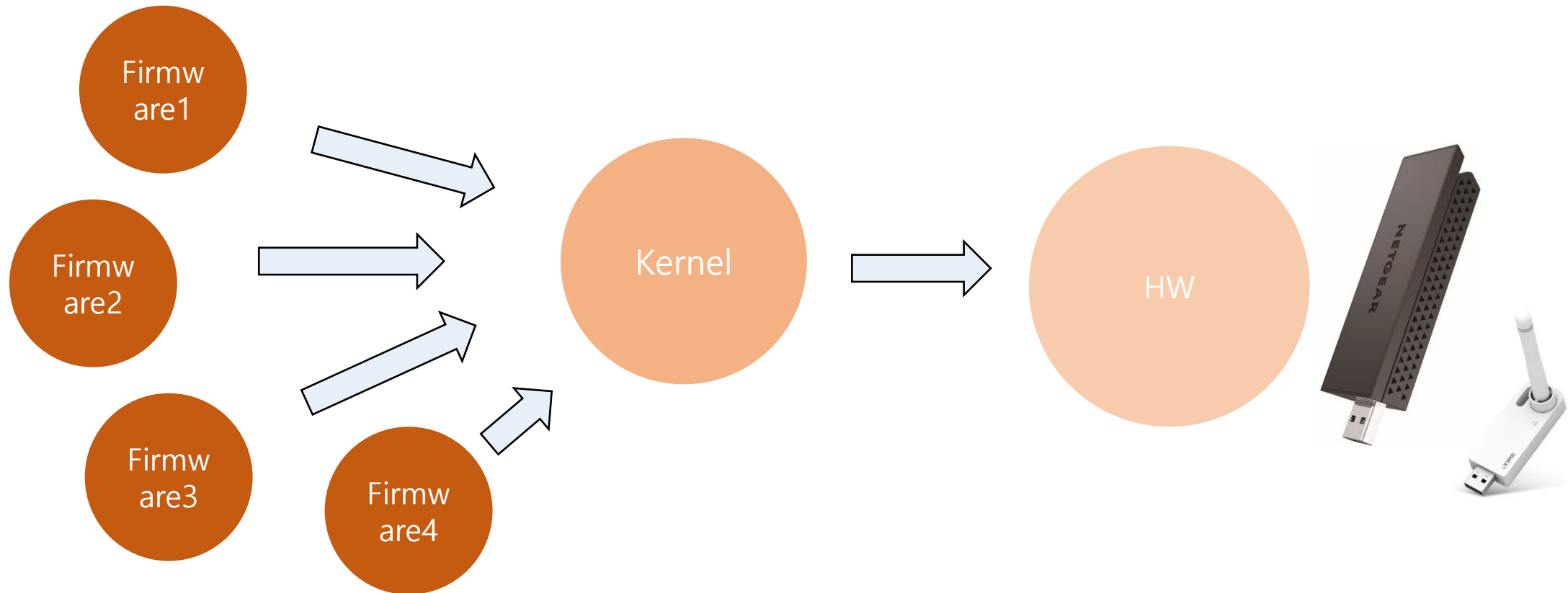


디바이스 드라이버 필요성

새로운 HW 추가를 위해 커널 소스코드 수정시 재 빌드 필요

Kernel 만 재 빌드하면 된다.

그런데 Build 시간이 너무 오래걸린다.



디바이스 드라이버 필요성

커널 빌드에 걸리는 시간 (개발 PC 기준)

2010 년도 : 한국에서 미국까지 비행기 타고 걸리는 시간

2020 년도 : 3분, 라면 익는 시간

AMD의 64코어 : 20초

Kernel



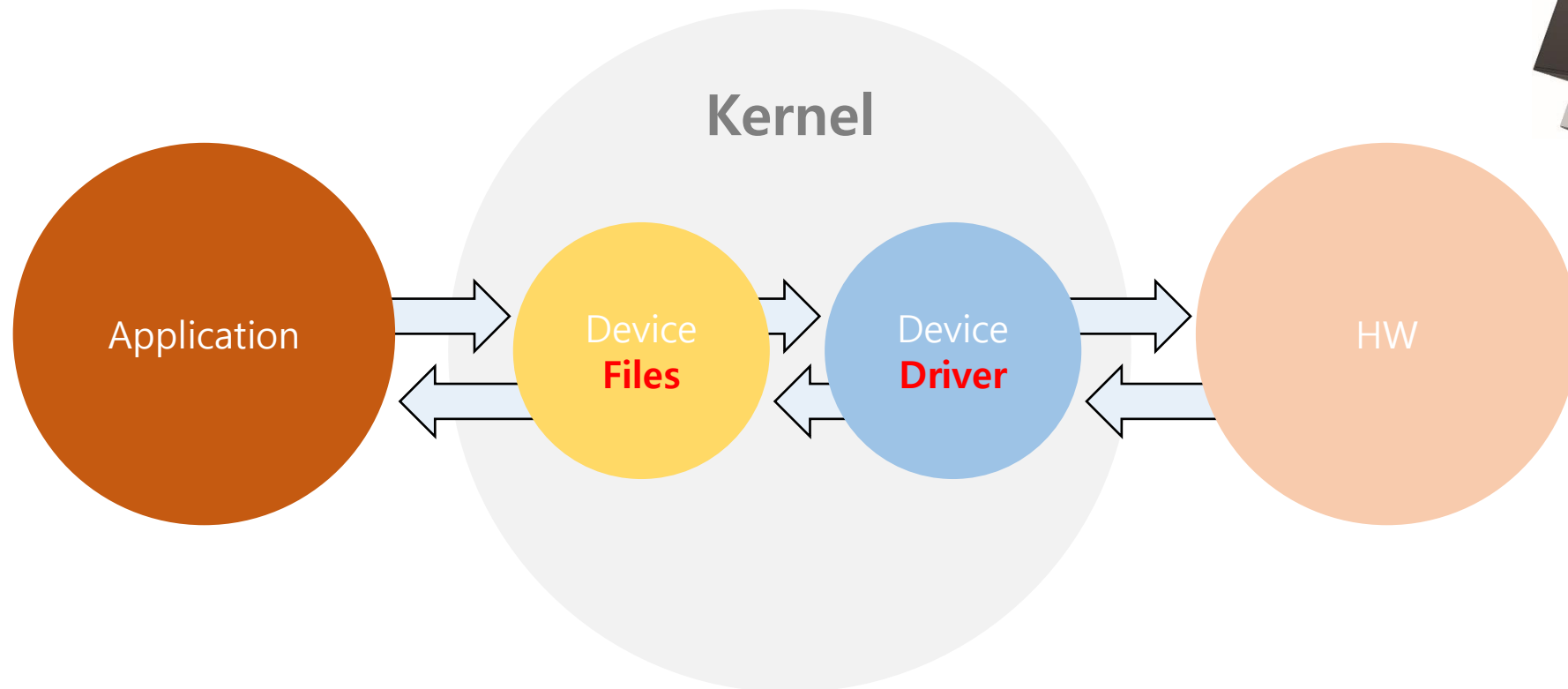
디바이스 드라이버 필요성

커널 안에서도 2 개의 Layer로 나눔

커널을 다시 Build하지 않도록 모듈 방식 사용

Device Files 에게 API 를 던진다.

Device Driver만 재 Build를 하여 커널에 넣고 뺀다.



디바이스 드라이버 필요성

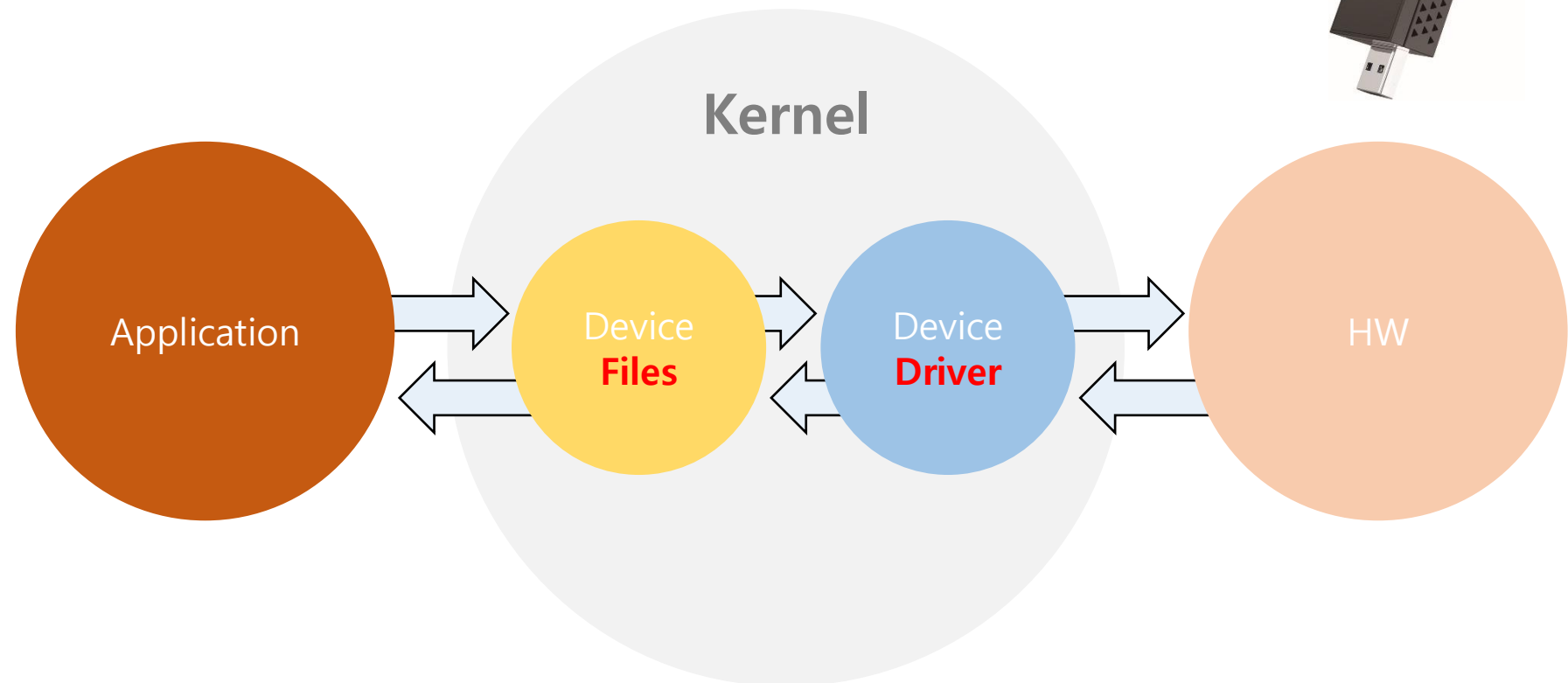
Device Driver Module 사용하기

Kernel에 Device Driver Module을 넣는다. (insmod 명령어)

- 넣으면 Kernel이 해당 모듈을 관리 하기 시작한다.

필요 없으면 Module을 제거한다. (rmmod 명령어)

- 빼는 것이 아니라, 메모리상 모두 Remove 하는 것이다.



Device Driver 개발과정 - 실습 1

[실습 1] module

디바이스 드라이버 모듈 만들어보기

main 함수가 없다.

라이선스 설정 해주어야 경고 메시지가 안뜬다.

두 가지 권장사항

- 커널 내부 모듈끼리 함수 중복을 피하기 위해 함수명을 "모듈이름_역할" 형태로 사용
- 모든 함수에 static 을 붙이기

파일명 : hi.c

```
#include <linux/module.h>

MODULE_LICENSE("GPL");

static int hi_init(void)
{
    printk( KERN_INFO "OK HELLO KFC\n");
    return 0;
}

static void hi_exit(void)
{
    printk( KERN_INFO "BYE BYE\n\n");
}

module_init(hi_init);
module_exit(hi_exit);
```

[실습 1] module

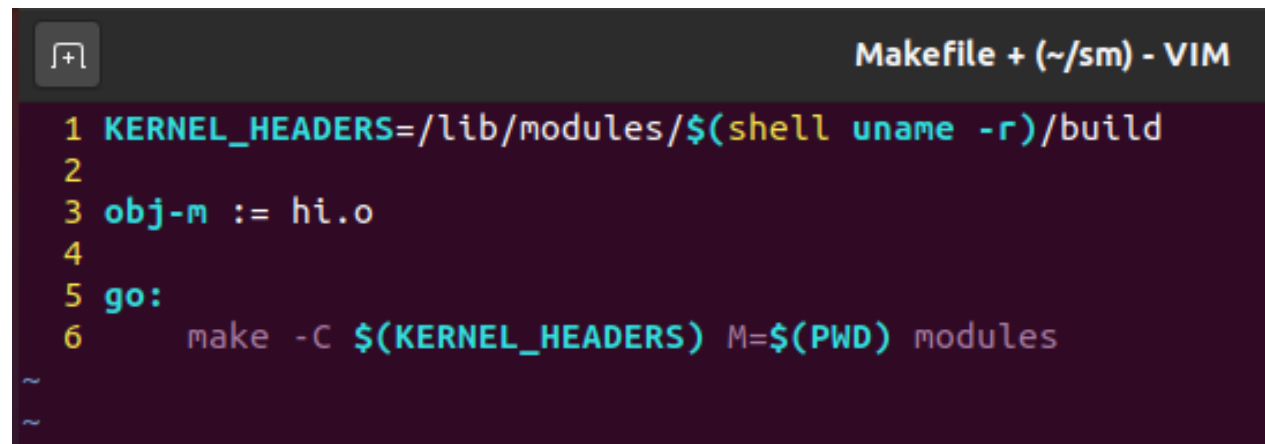
Makefile 만들기

target 밑에 들여쓰기는 반드시 "TAB" 으로 한다.

- C 옵션 : 해당 디렉토리로 이동해서 make를 수행
 - 해당 디렉토리에 있는 Makefile이 수행된다.

M=\$PWD

- 결과물이 현재 디렉토리에 생성된다.



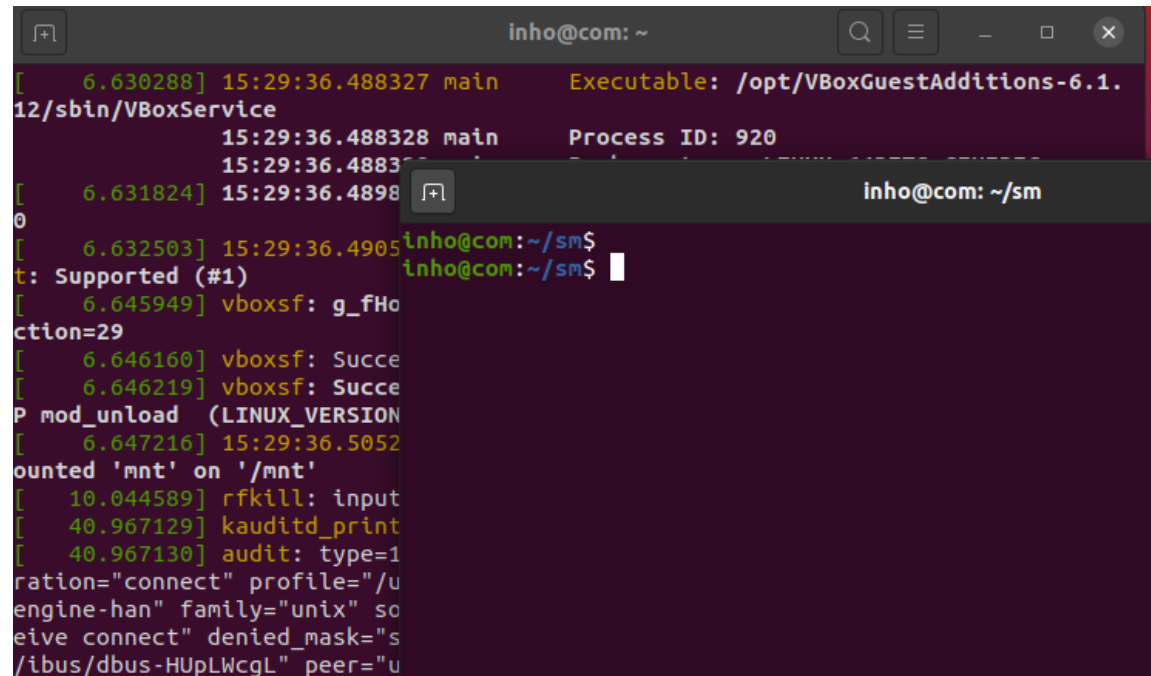
```
1 KERNEL_HEADERS=/lib/modules/$(shell uname -r)/build
2
3 obj-m := hi.o
4
5 go:
6     make -C $(KERNEL_HEADERS) M=$(PWD) modules
~
~
```

[실습 1] module

kernel log로 동작 상태 파악

`dmesg` : 최근 커널로그 출력

`dmesg -w` : 커널로그 모니터링



```
inho@com: ~  
[ 6.630288] 15:29:36.488327 main Executable: /opt/VBoxGuestAdditions-6.1.  
12/sbin/VBoxService  
15:29:36.488328 main Process ID: 920  
15:29:36.488328 main  
[ 6.631824] 15:29:36.489800  
[ 6.632503] 15:29:36.490500 inho@com:~/sm$  
t: Supported (#1) inho@com:~/sm$  
[ 6.645949] vboxsf: g_fHo  
ction=29  
[ 6.646160] vboxsf: Succe  
[ 6.646219] vboxsf: Succe  
P mod_unload (LINUX_VERSION  
[ 6.647216] 15:29:36.505200  
ounted 'mnt' on '/mnt'  
[ 10.044589] rfkill: input  
[ 40.967129] kauditd_print  
[ 40.967130] audit: type=1  
ration="connect" profile="/u  
engine-han" family="unix" so  
eive connect" denied_mask="s  
/ibus/dbus-HUpLWcgl" peer="u
```

창을 2개 띄어두고 있으면 개발이 편리하다.

[실습 1] module

커널 모듈을 커널에 적재하는 방법

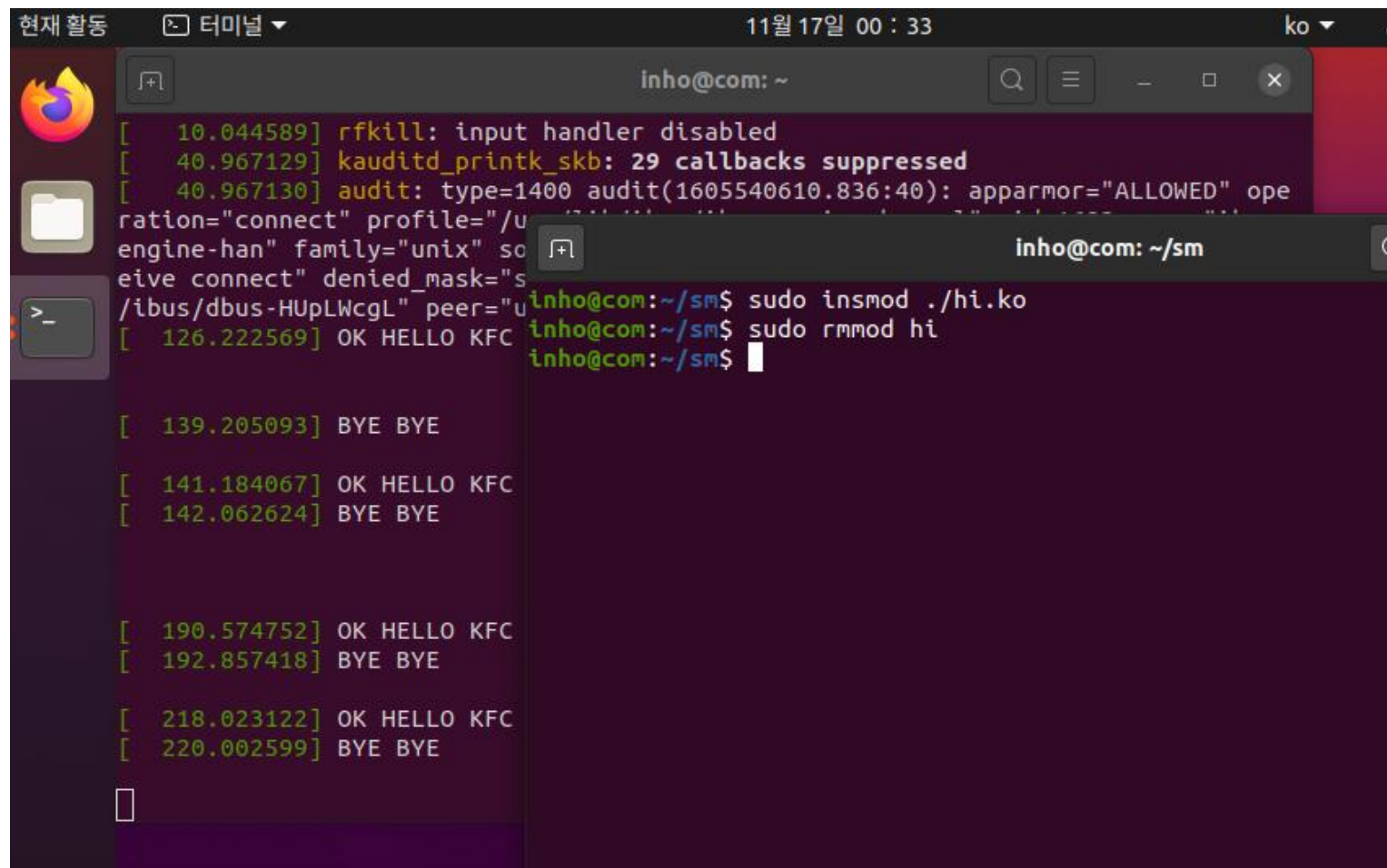
Device Driver를 Build 후 생성되는 파일 : .ko
ex) min.ko

`sudo insmod hi.ko` : 커널에 모듈 적재

`sudo rmmod hi` : 커널에 모듈 제거

[실습 1] module

동작 결과



The screenshot shows a terminal window with two panes. The top pane displays kernel logs with timestamps and messages from rfkill, kauditd, and audit. The bottom pane shows the user in the directory ~/sm performing module operations: loading hi.ko, listing modules, and removing hi.

```
현재 활동  터미널  11월 17일 00:33  ko
incho@com: ~
[ 10.044589] rfkill: input handler disabled
[ 40.967129] kauditd_printk_skb: 29 callbacks suppressed
[ 40.967130] audit: type=1400 audit(1605540610.836:40): apparmor="ALLOWED" operation="connect" profile="/usr/share/apparmor/profiles/abx-engine-han" family="unix" source="eive connect" denied_mask="source" /ibus/dbus-HUpLWcgl" peer="u
[ 126.222569] OK HELLO KFC
[ 139.205093] BYE BYE
[ 141.184067] OK HELLO KFC
[ 142.062624] BYE BYE
[ 190.574752] OK HELLO KFC
[ 192.857418] BYE BYE
[ 218.023122] OK HELLO KFC
[ 220.002599] BYE BYE
incho@com:~/sm$ sudo insmod ./hi.ko
incho@com:~/sm$ sudo lsmod
incho@com:~/sm$ sudo rmmod hi
incho@com:~/sm$
```


Device Driver의 장점

1> Firmware로 게임을 제작했다.

그래픽 카드만 바뀌었을 때, 게임을 다시 Build 해야할까?

2> Embedded Linux용 게임을 제작했다.

그래픽 카드만 바뀌었을 때, 게임을 다시 Build 할 필요없다.

어떤 것을 다시 교체해야 할까?

Windows / MacOS / Linux 모두
Device Driver는 필수적으로 사용된다.

Device Driver의 장점

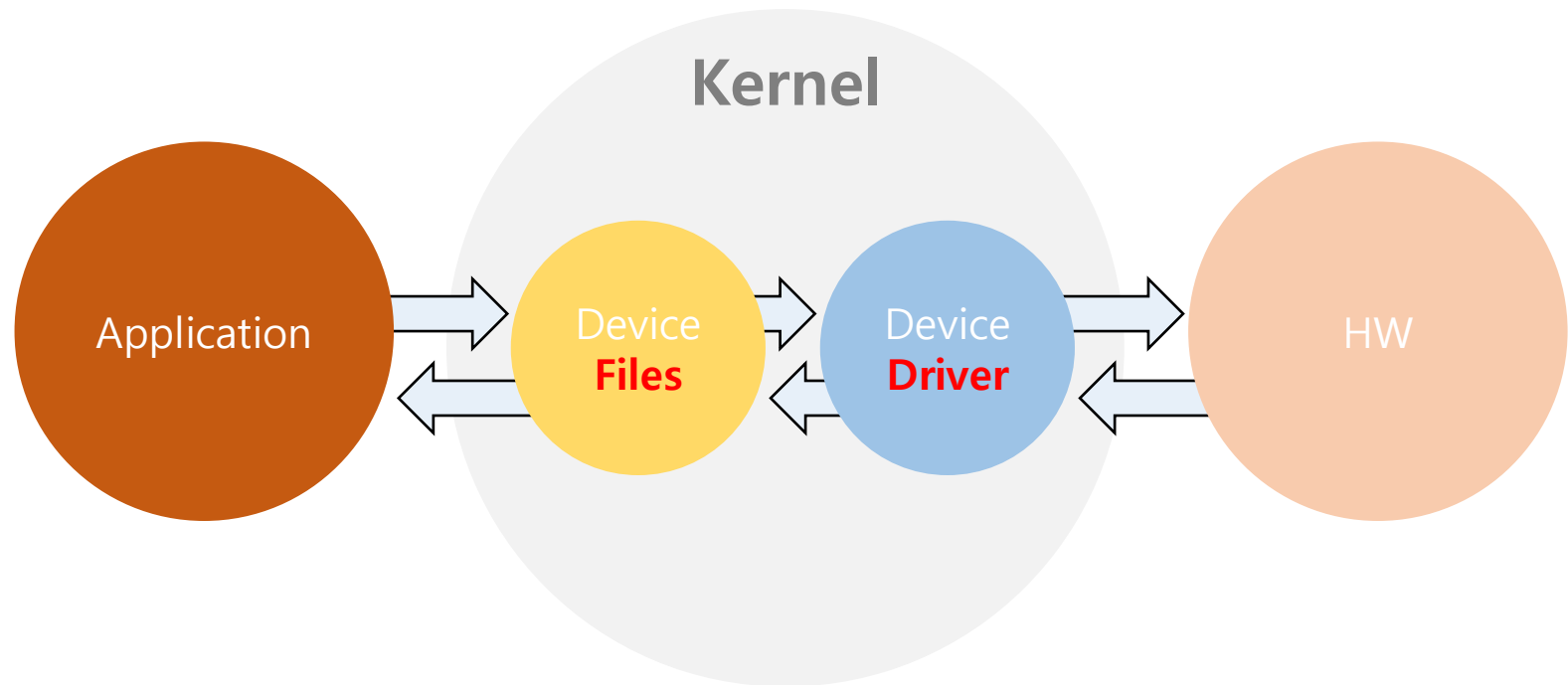
App / Driver 역할 분담 가능

Application 개발자

- HW를 제어할 수 있는 API 사용 방법 익히기
- API가 건내준 Data 처리
- API를 이용하여 HW 제어
- Log 남기기
- Network / UI 작업

Device Driver 개발자

- HW 제어 API 제작
- HW 불량 분석
- Latency 측정



Device Driver 개발과정 – 실습 2. device file

실습2 소스코드 - open과 release만 있는 커널모듈

다음은 실습2에 대한 소스코드이다.

코드 다운로드

<https://gist.github.com/mincoding1/9bc943df5255fbef6d8a15e067207ece>

app.c : Application

nobrand.c : Device Driver

Makefile : Build

```
0=1000 0000=0 LD [M] /home/ubuntu/test1/nobrand.ko
[ 2812.230067] hi make[1]: Leaving directory '/usr/src/linux
[ 3062.578091] bye gcc -o app app.c
[ 3083.657966] hi ubuntu@ubuntu:~/test1$ ./app
[ 3228.078329] welcome GO
[ 3228.078547] release ubuntu@ubuntu:~/test1$
```

동작결과 : dmesg -w 로 확인 가능

[실습 2] device File

디바이스 파일 제작

```
sudo mknod /dev/nobrand c 100 0
```

- c는 캐릭터 디바이스라는 뜻, 메이저번호 100, 마이너번호 0

권한 설정하기 (666 권한)

```
sudo chmod 666 /dev/nobrand
```

```
lnho@com:~/hi$  
lnho@com:~/hi$ sudo mknod /dev/nobrand c 100 0  
lnho@com:~/hi$  
lnho@com:~/hi$ ls -al /dev/nobrand  
crw-r--r-- 1 root root 100, 0 11월 16 15:10 /dev/nobrand  
lnho@com:~/hi$  
lnho@com:~/hi$
```

[실습 2] device File

Application 제작 (파일명 : app.c)

Device File 을 open, close 하는 system call 사용

ERROR 출력을 반드시 해 주자

fcntl.h : open 을 위해 추가

unistd.h : close 를 위해 추가

```
app.c (~/hi) - VIM
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/stat.h>
4 #include <fcntl.h>
5 #include <unistd.h>
6
7 int main()
8 {
9     int fd = open("/dev/nobrand", O_RDWR);
10    if (fd < 0) printf("ERROR\n");
11
12    printf("GO\n");
13
14    close(fd);
15
16    return 0;
17 }
```

[실습 2] device File

MAJOR / NAME 지정
되어있음을
확인할 수 있다.

```
1 #include <linux/module.h>
2 #include <linux/fs.h>
3
4 #define NOD_MAJOR 100
5 #define NOD_NAME "nobraud"
6
7 MODULE_LICENSE("GPL");
8
```

```
1 #include <linux/module.h>
2 #include <linux/fs.h>
3
4 #define NOD_MAJOR 100
5 #define NOD_NAME "nobraud"
6
7 MODULE_LICENSE("GPL");
8
9 static int nobrand_open(struct inode *inode, struct file *filp)
10 {
11     printk( KERN_INFO "welcome\n");
12     return 0;
13 }
14
15 static int nobrand_release(struct inode *inode, struct file *filp)
16 {
17     printk( KERN_INFO "release\n");
18     return 0;
19 }
20
21 static struct file_operations fops = {
22     .open = nobrand_open,
23     .release = nobrand_release,
24 };
25
26 static int nobrand_init(void)
27 {
28     if (register_chrdev(NOD_MAJOR, NOD_NAME, &fops) < 0) {
29         printk("INIT FAIL\n");
30     }
31     printk( KERN_INFO "hi\n");
32     return 0;
33 }
34
35 static void nobrand_exit(void)
36 {
37     unregister_chrdev(NOD_MAJOR, NOD_NAME);
38     printk( KERN_INFO "bye\n");
39 }
40
41 module_init(nobrand_init);
42 module_exit(nobrand_exit);
43
44
45
```

[실습 2] device File

file_operation 구조체 변수 만들기

app level 에서 "open" system call → minco_open 함수가 호출

app level에서 "close" system call → minco_release 함수가 호출
커널에 chrdev를 등록

- Major Number / 모듈 이름 필요
- connect 정보가 담긴 fops 구조체 필요

```
21 static struct file_operations fops = {
22     .open    = nobrand_open,
23     .release = nobrand_release,
24 };
25
26 static int nobrand_init(void)
27 {
28     if (register_chrdev(MAJOR, NAME, &fops) < 0) {
29         printk("INIT FAIL\n");
30     }
31
32     printk( KERN_INFO "hi\n");
33     return 0;
34 }
35
36 static void nobrand_exit(void)
37 {
38     unregister_chrdev(MAJOR, NAME);
39     printk( KERN_INFO "bye\n");
40 }
```

```
1 #include <linux/module.h>
2 #include <linux/fs.h>
3
4 #define NOD_MAJOR 100
5 #define NOD_NAME "nobrand"
6
7 MODULE_LICENSE("GPL");
8
9 static int nobrand_open(struct inode *inode, struct file *filp)
10 {
11     printk( KERN_INFO "welcome\n");
12     return 0;
13 }
14
15 static int nobrand_release(struct inode *inode, struct file *filp)
16 {
17     printk( KERN_INFO "release\n");
18     return 0;
19 }
20
21 static struct file_operations fops = {
22     .open    = nobrand_open,
23     .release = nobrand_release,
24 };
25
26 static int nobrand_init(void)
27 {
28     if (register_chrdev(NOD_MAJOR, NOD_NAME, &fops) < 0) {
29         printk("INIT FAIL\n");
30     }
31
32     printk( KERN_INFO "hi\n");
33     return 0;
34 }
35
36 static void nobrand_exit(void)
37 {
38     unregister_chrdev(NOD_MAJOR, NOD_NAME);
39     printk( KERN_INFO "bye\n");
40 }
41
42 module_init(nobrand_init);
43 module_exit(nobrand_exit);
44
45
```


[실습 2] device File

- open, release 함수 (inode, filp)

inode : Device File의 inode 정보를 갖는다.
이 디바이스 파일의 정보를 읽을 수 있다.
insmod시 inode에는 주번호 / 부번호 정보도 적힌다.

filp : 디바이스 파일이 App에서 어떤 형태로 읽혔는지
정보가 들어있다.
우리 예제에서는 O_RDWR 형태로 읽었음

```
9 static int nobrand_open(struct inode *inode, struct file *filp)
10 {
11     printk( KERN_INFO "welcome\n");
12     return 0;
13 }
14
15 static int nobrand_release(struct inode *inode, struct file *filp)
16 {
17     printk( KERN_INFO "release\n");
18     return 0;
19 }
20
```

```
1 #include <linux/module.h>
2 #include <linux/fs.h>
3
4 #define NOD_MAJOR 100
5 #define NOD_NAME "nbrand"
6
7 MODULE_LICENSE("GPL");
8
9 static int nobrand_open(struct inode *inode, struct file *filp)
10 {
11     printk( KERN_INFO "welcome\n");
12     return 0;
13 }
14
15 static int nobrand_release(struct inode *inode, struct file *filp)
16 {
17     printk( KERN_INFO "release\n");
18     return 0;
19 }
20
21 static struct file_operations fops = {
22     .open = nobrand_open,
23     .release = nobrand_release,
24 };
25
26 static int nobrand_init(void)
27 {
28     if (register_chrdev(NOD_MAJOR, NOD_NAME, &fops) < 0) {
29         printk("INIT FAIL\n");
30     }
31
32     printk( KERN_INFO "hi\n");
33     return 0;
34 }
35
36 static void nobrand_exit(void)
37 {
38     unregister_chrdev(NOD_MAJOR, NOD_NAME);
39     printk( KERN_INFO "bye\n");
40 }
41
42 module_init(nobrand_init);
43 module_exit(nobrand_exit);
44
45
```

```
fd = open("/dev/device", O_RDWR);
```

```
int device_open ( struct inode *inode, struct file *filp )
{
    return 0;
}
```

```
ret = close ( fd );
```

```
int device_release ( struct inode *inode, struct file *filp )
{
    return ret;
}
```

[실습 2] device File

Make 파일 변경하기 (탭키 주의하기 - 들여쓰기를 띄어쓰기로 하면 에러 발생)

기존 Make file에 C언어 빌드도 추가

make all : 전체 빌드

make driver : 드라이버 빌드

make app : app 빌드

make clean : 드라이버 + app 모두 제거

```
Makefile
1 KERNEL_HEADERS=/lib/modules/$(shell uname -r)/build
2 CC = gcc
3
4 TARGET := app
5 obj-m := nobrand.o
6
7 all: driver app
8
9 driver:
10     make -C $(KERNEL_HEADERS) M=$(PWD) modules
11
12 app:
13     $(CC) -o $@ $@.c
14
15 clean:
16     make -C $(KERNEL_HEADERS) M=$(PWD) clean
17     rm -f *.o $(TARGET)
~
```

[실습 2] device File

동작 순서

커널로그 켜두기

> `dmesg -w`

```
[15458.934120] hi  
[15459.745177] welcome  
[15459.745728] release  
[15461.063935] bye
```

Make / insmod

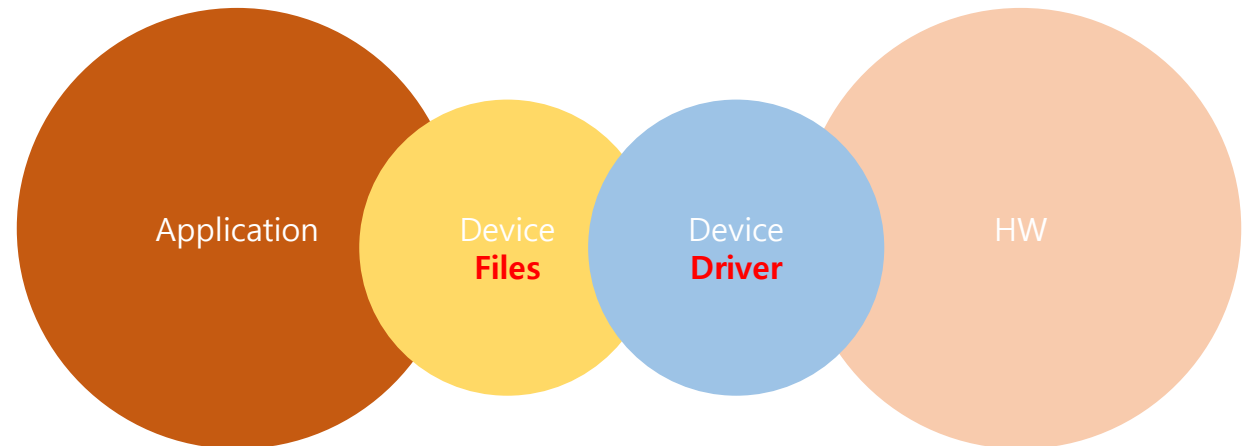
`sudo mknod /dev/nobrand c 100 0`

`sudo chmod 666 /dev/nobrand.c`

`sudo Insmod ./nobrand.ko`

Application �행

> `./app`



Device Driver 개발과정 – 실습 3. ioctl

ioctl 이란?

ioctl (input output control)

<sys/ioctl.h> 필요

하드웨어를 제어하기 위한 함수

예약된 동작 번호 지정

int ioctl(int fd, unsigned long request, arg)

fd : 파일 디스크립터

request : cmd parameter 규칙에 맞춰서 작성

arg : 포인터 전달

ex) ioctl(fd, _IO(0,3), 0);

ioctl 을 사용하는 이유

리눅스는 모든 것을 파일로 관리한다.

장치에 대응하는 장치 파일을 open() / read() / write() close() 하면 된다?!

read() / write() 는 실제로 많이 사용하지 않는다.

ioctl() 을 사용한다.

대부분의 장치는 통신 메커니즘이 필요하다.

SPI / I2C 등등..

read/write 만으로는 장치와 통신을 할 수 없다.

대규모의 데이터 전송도 힘들다.

그래서 ioctl() 을 사용한다.

[실습 3] 기존 코드에 ioctl 코드를 추가

코드 다운로드

<https://gist.github.com/mincoding1/ec1d36d95881c08baa458684498ef1a1>

변경사항

app.c : ioctl 호출 코드 추가

Nobrand.c : ioctl 처리 코드 추가

```
LD [M] /home/ubuntu/test1/nobrand.ko [ 3953.044580] welcome
make[1]: Leaving directory '/usr/src/linux-headers-5.15.0-46-generic' [ 3953.044698] 3
gcc -o app app.c [ 3953.044703] HI - ioctl 3
ubuntu@ubuntu:~/test1$ sudo insmod nobrand.ko [ 3953.044705] 4
ubuntu@ubuntu:~/test1$ ./app [ 3953.044707] HI - ioctl 4
GO [ 3953.044709] 5
ubuntu@ubuntu:~/test1$ [ 3953.044724] HI - ioctl 5
[ 3953.044728] release
```

동작결과 : dmesg -w 로 확인 가능

코드 설명 – app.c

`#include <sys/ioctl.h>`
ioctl 사용을 위한 헤더

ioctl() 이용해서 arg 값 전달

_IO(type, num) : cmd parameter 매크로
arg : 10진수, 2진수, 16진수 전달 가능

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>

int main()
{
    int fd = open("/dev/nobrand", O_RDWR);
    if (fd < 0) printf("ERROR\n");

    printf("GO\n");

    //ioctl
    ioctl(fd, _IO(0, 3), 0);
    ioctl(fd, _IO(0, 4), 0);
    ioctl(fd, _IO(0, 5), 0);

    close(fd);

    return 0;
}
```


App 에서 보낸 ioctl() syscall에 대응할 API 제작
cmd argument로 전송된 값을 출력한다.

```
static long nobrand_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    printk( KERN_ALERT "%d\n", cmd);
    switch (cmd) {
        case _IO(0, 3) :
            printk( KERN_INFO "HI - ioctl 3 \n");
            break;

        case _IO(0, 4) :
            printk( KERN_INFO "HI - ioctl 4 \n");
            break;

        case _IO(0, 5) :
            printk( KERN_INFO "HI - ioctl 5 \n");
            break;
    }

    return 0;
}
```

코드 설명 – nobrand.c

switch 문을 이용하여, cmd 값에 따라 동작한다.

```
static long nobrand_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    printk( KERN_ALERT "%d\n", cmd);
    switch (cmd) {
        case _IO(0, 3) :
            printk( KERN_INFO "HI - ioctl 3 \n");
            break;

        case _IO(0, 4) :
            printk( KERN_INFO "HI - ioctl 4 \n");
            break;

        case _IO(0, 5) :
            printk( KERN_INFO "HI - ioctl 5 \n");
            break;
    }

    return 0;
}
```

fops 에 등록해서 ioctl() syscall 에 동작하도록 한다.

.unlocked_ioctl : kernel 이 ioctl 사용 시 lock 이 걸리는 데, 이를 방지하도록 한다.

```
static struct file_operations fops = {  
    .open    = nobrand_open,  
    .release = nobrand_release,  
    .unlocked_ioctl = nobrand_ioctl,  
};
```

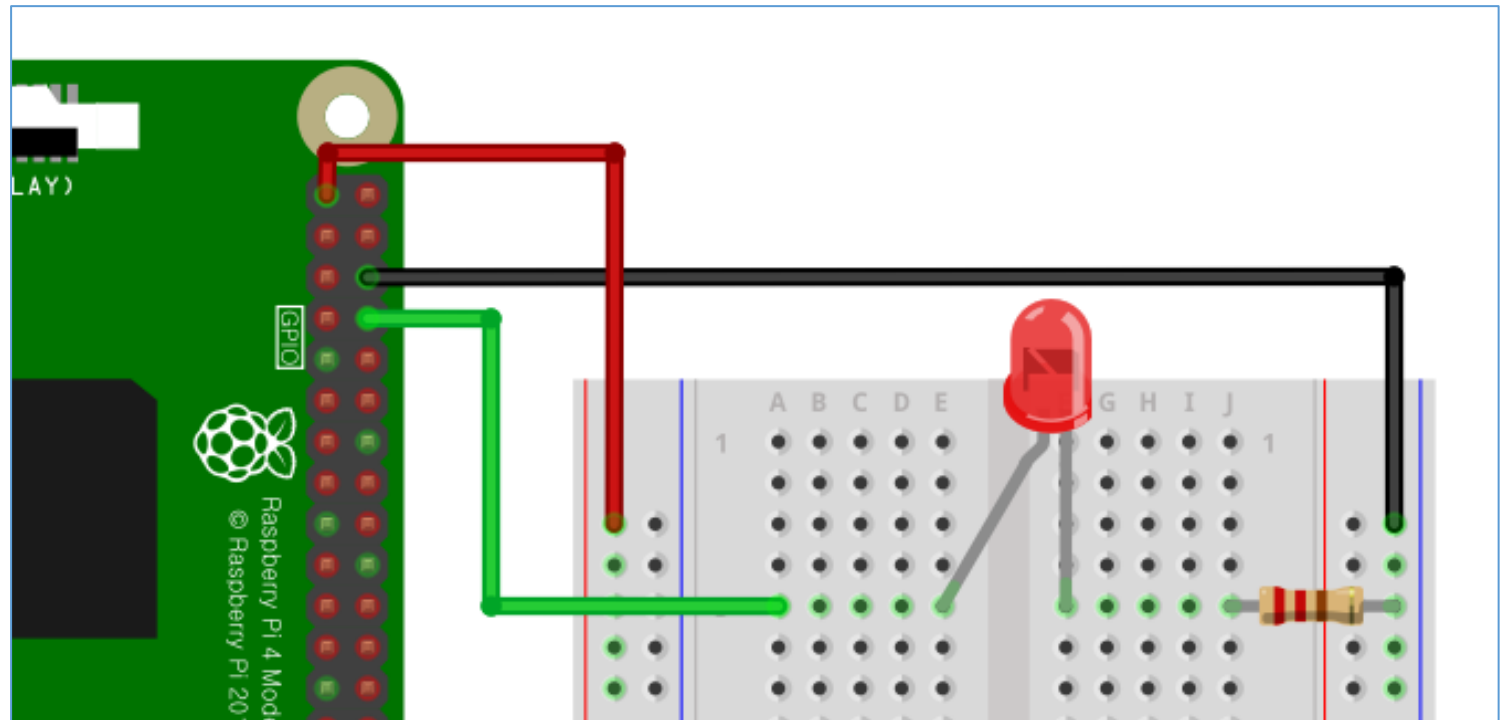
Device Driver 개발과정 – 실습 4. GPIO 제어

회로 연결

회로를 연결한다.

LED (+) → GPIO14

LED (-) → 220옴 → GND



코드를 작성하고 make 한다.

~/test8 디렉토리 생성
LED 를 깜빡이는 샘플 코드

이 코드에는
Device File을
생성해주는 코드도 추가되어있다.
mknod를 따로 해주지 않아도 된다.

```
case _IO(0,3):  
    pr_info("LED ON\n");  
    ledon();  
    break;  
case _IO(0,4):  
    pr_info("LED OFF\n");  
    ledoff();  
    break;  
default :  
    return -EINVAL;
```

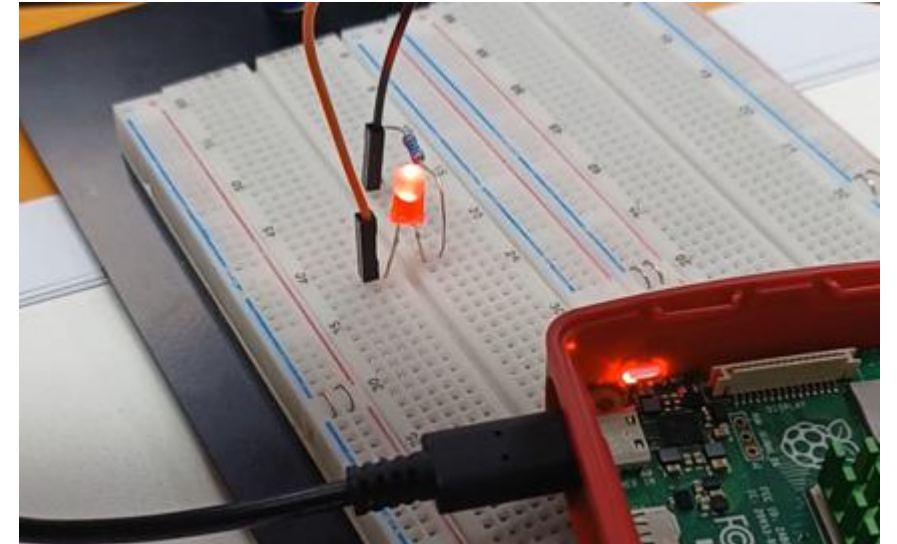
<https://gist.github.com/hoconoco/68f90af8277b238d2d12a5713f94e6c1>

동작 확인

app을 실행해서 숫자를 입력한다.

3 → LED ON

4 → LED OFF



```
pi@raspberrypi:~/test $ sudo insmod devicedriver.ko
pi@raspberrypi:~/test $ sudo chmod 666 /dev/deviceFile
pi@raspberrypi:~/test $ ./app
>>3
>>4
>>3
>>4
>>5
command invalid!
pi@raspberrypi:~/test $ sudo rmmod devicedriver
pi@raspberrypi:~/test $
```

```
[ 7067.927356] Insmod Module
[ 7067.930261] Major number 236
[ 7067.930289] Device file : /dev/deviceFile
[ 7076.742153] Open Device
[ 7077.793477] LED ON
[ 7078.289628] LED OFF
[ 7078.809162] LED ON
[ 7079.216749] LED OFF
[ 7079.578440] Close Device
[ 7086.748659] Unload Module
```

<linux/gpio.h>

GPIO 가 리눅스에서 표준화 되어 있다.
커널에서 쓸 수 있는 API가 지원된다.

LED 핀 번호

```
7  #include <linux/gpio.h>
8
9  #define NOD_NAME "deviceFile"
10
11  MODULE_LICENSE("GPL");
12
13  static int NOD_MAJOR;
14  static struct class *cls;
15
16  #define LED 14
```


gpio_set_value(unsigned int gpio, int value)

해당 gpio 핀에 출력 값 설정

```
18  static void ledon(void){  
19      gpio_set_value(LED,1);  
20  }  
21  static void ledoff(void){  
22      gpio_set_value(LED,0);  
23  }
```

`int gpio_request(unsigned gpio, const char* label)`

해당 gpio 핀 사용
라벨 : 디버깅 용도

`int gpio_direction_output(unsigned gpio, int value)`

해당 gpio 핀을 출력 모드로 설정
초기 값

```
70     pr_info("Major number %d\n", NOD_MAJOR);
71     pr_info("Device file : /dev/%s\n", NOD_NAME);
72
73     gpio_request(LED, "LED");
74     gpio_direction_output(LED, 0);
75
76     return 0;
77 }
```

void gpio_free(unsigned gpio)

해당 gpio 핀 사용 해제

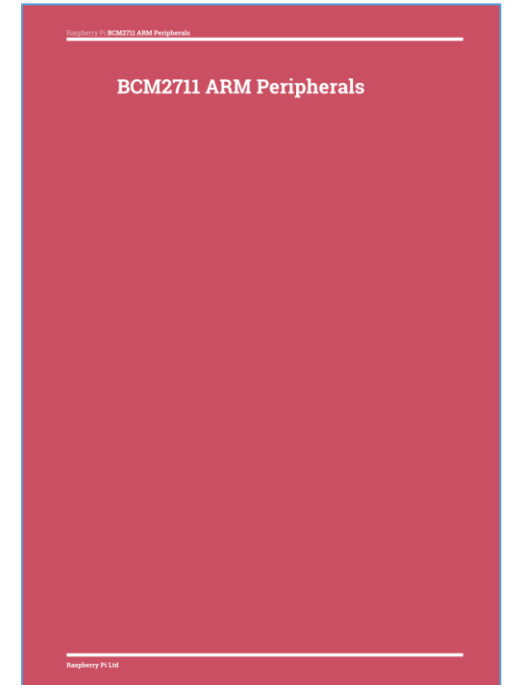
```
79  static void __exit deviceFile_exit(void)
80  {
81      gpio_set_value(LED,0);
82      gpio_free(LED);
83      device_destroy(cls, MKDEV(NOD_MAJOR, 0));
84      class_destroy(cls);
```

linux/gpio.h 가 없었다면,

우리는 Datasheet 를 보고

특정 register 에 값을 넣는 방식으로 H/W 제어를 해야 한다!

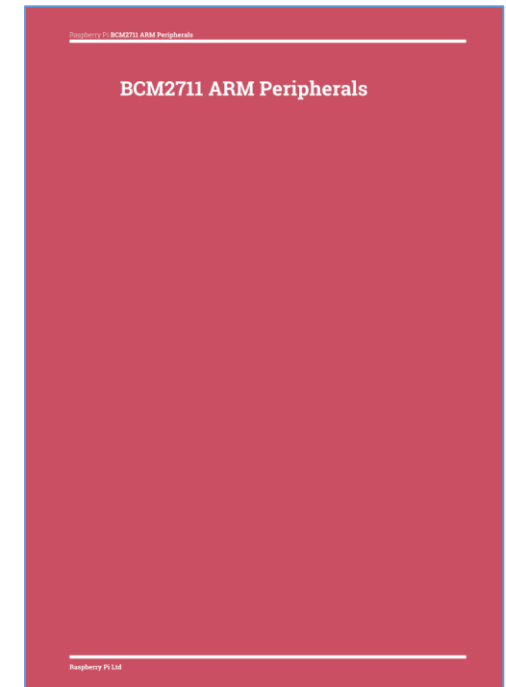
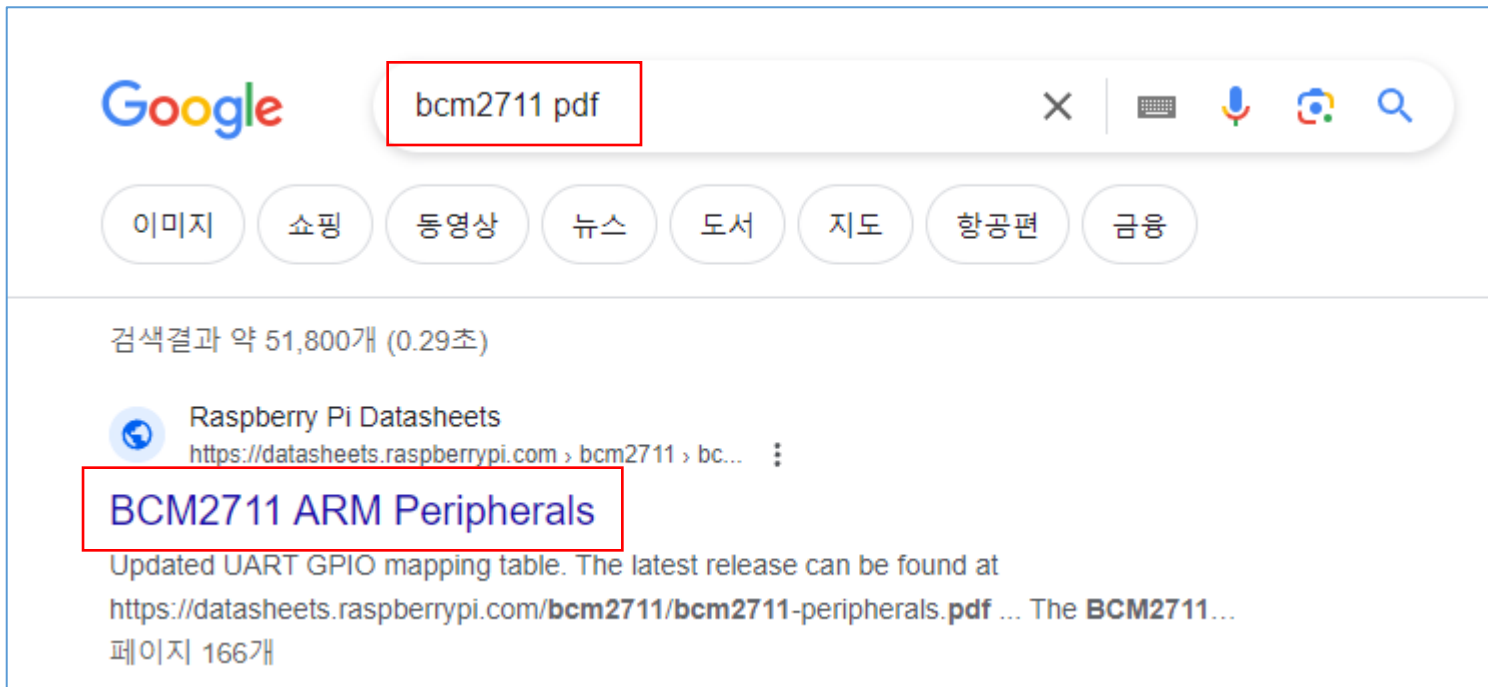
다음 챕터는 Datasheet를 보고 직접 H/W 제어하는 방법을 소개한다.
참고만하자. (gpio.h가 있어서 다행이다 ^^)



[참고자료] HW 직접 제어하기 - Memory Map 분석

bcm2711 datasheet 다운로드

bcm2711 pdf 을 검색해서 datasheet 를 다운 받는다.



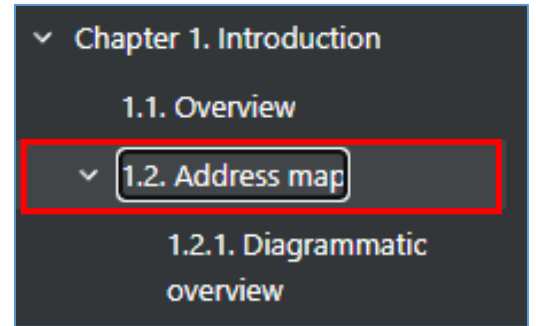
1.2. Address map 으로 간다.

라즈베리파이의 Memory Map을 살펴보자.

bcm2711 은 두 개의 main 주소 체계가 있다.

“Full 35-bit” / “Legacy Master”

Figure1 에서 address 간 관계를 알 수 있다고 한다.



1.2. Address map

1.2.1. Diagrammatic overview

The BCM2711 has two main addressing schemes: a "Full" 35-bit address bus and a 32-bit "Legacy Master" view as seen by the peripherals (except for "large address" masters). There's also a "Low Peripherals" mode which modifies the ARM's view of the peripheral addresses. [Figure 1](#) shows how these address maps inter-relate. Note that the relative sizes of the address blocks in the diagram are definitely **not** to scale! (The PCIe address range covers 8GB, but the Main peripherals address range only covers 64MB.)

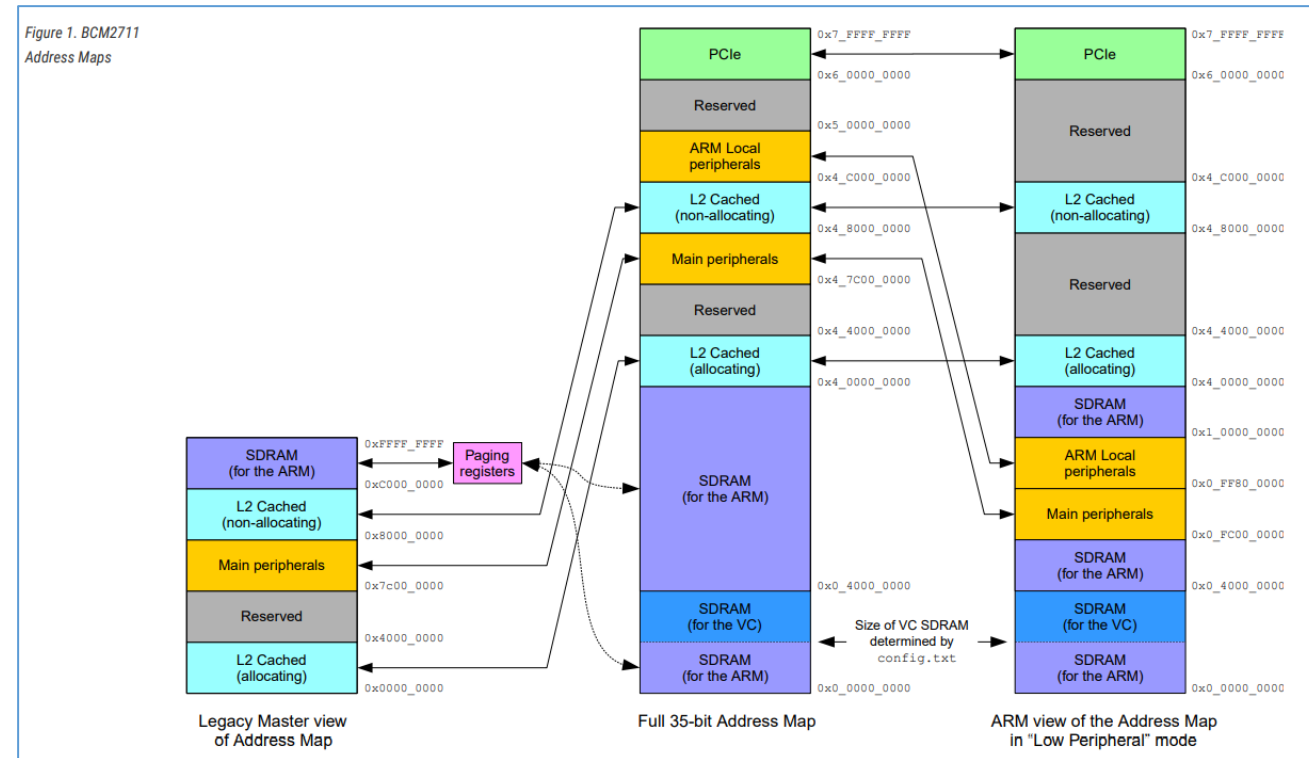
Figure 1

Main address 사이의 관계를 나타내는 그림 1

Legacy : 공식문서 default 메모리 주소

Full 35-bit Address : high performance 용 주소

Low peripheral : ARM 프로세서의 요구에 맞게 구성된 주소



우리가 할 일

내가 사용하는 장치가 어떤 메모리 맵을 사용하는 지 먼저 파악한다.

lscpu

aarch64 아키텍처 사용 중 → 64bit 아키텍처

호환 문제로 64bit 를 지원하는 하드웨어에 64bit OS를 설치하였는 데도 불구하고,
32bit 주소체계를 사용한다고 한다!!!

```
pi@raspberrypi:~ $ lscpu
Architecture: aarch64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 4
On-line CPU(s) list: 0-3
Thread(s) per core: 1
Core(s) per socket: 4
Socket(s): 1
Vendor ID: ARM
Model: 3
Model name: Cortex-A72
Stepping: r0p3
CPU max MHz: 1800.0000
```

그리고 Datasheet 에 있는 주소 정보는 Legacy가 기준!

Datasheet 에 있는 모든 주소 정보가 Legacy Address 기준이다!

오른쪽과 같은 주소를 보고 그대로 사용하면 안된다는 것!!

우리가 보고 있는 주소에 맞게 변환해서 사용해야 한다!

5.2. Register View

The GPIO has the following registers. All ac
0x7e200000.

Table 63. GPIO
Register Assignment

Offset	Name
0x00	GPFSEL0
0x04	GPESEL 1

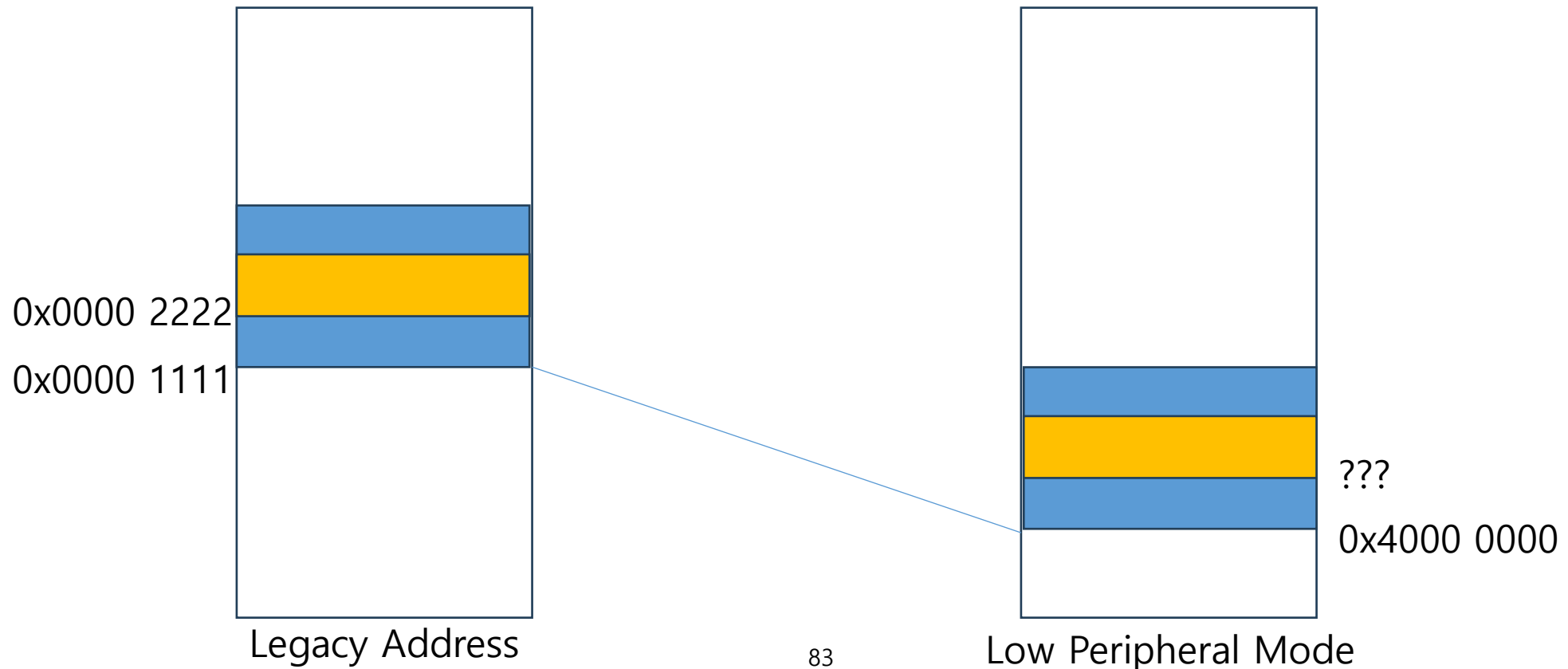
1.2.4. Legacy master addresses

The peripheral addresses specified in this document are legacy master addresses. Software accessing peripherals using the DMA engines must use 32-bit legacy master addresses. The Main peripherals are available from 0x7C00_0000 to 0x7FFF_FFFF. Behind the scenes, the VideoCore transparently translates these addresses to the 35-bit 0x4_7nnn_nnnn addresses.

Legacy Address 의 0x00002222를 코드에 적용하려면

$0x0000\ 2222 - 0x0000\ 1111 = 0x0000\ 1111$ (offset)

$0x4000\ 0000 + 0x0000\ 1111 = 0x4000\ 1111$ 을 적용해야 한다.



GPIO 주소를 확인하자

Page 67, Register View

GPIO의 Base 주소 : 0x7E20 0000

GPIO는 Main Peripherals 에 속해 있다.

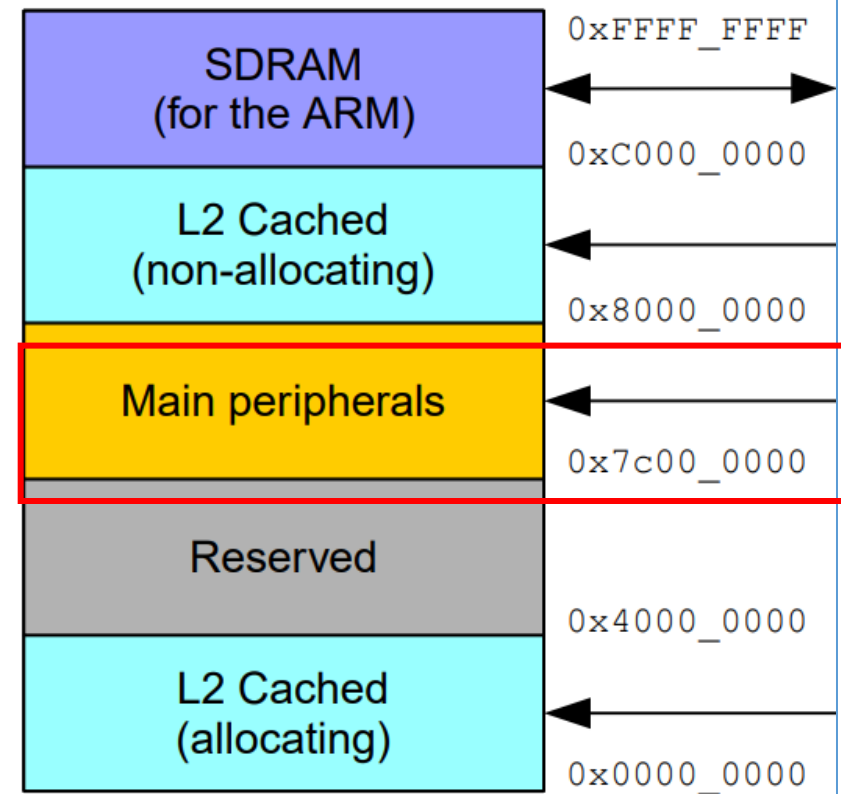
5.2. Register View

The GPIO has the following registers. All accesses are to the base address

0x7e200000.

Table 63. GPIO
Register Assignment

Offset	Name
0x00	GPFSSEL0



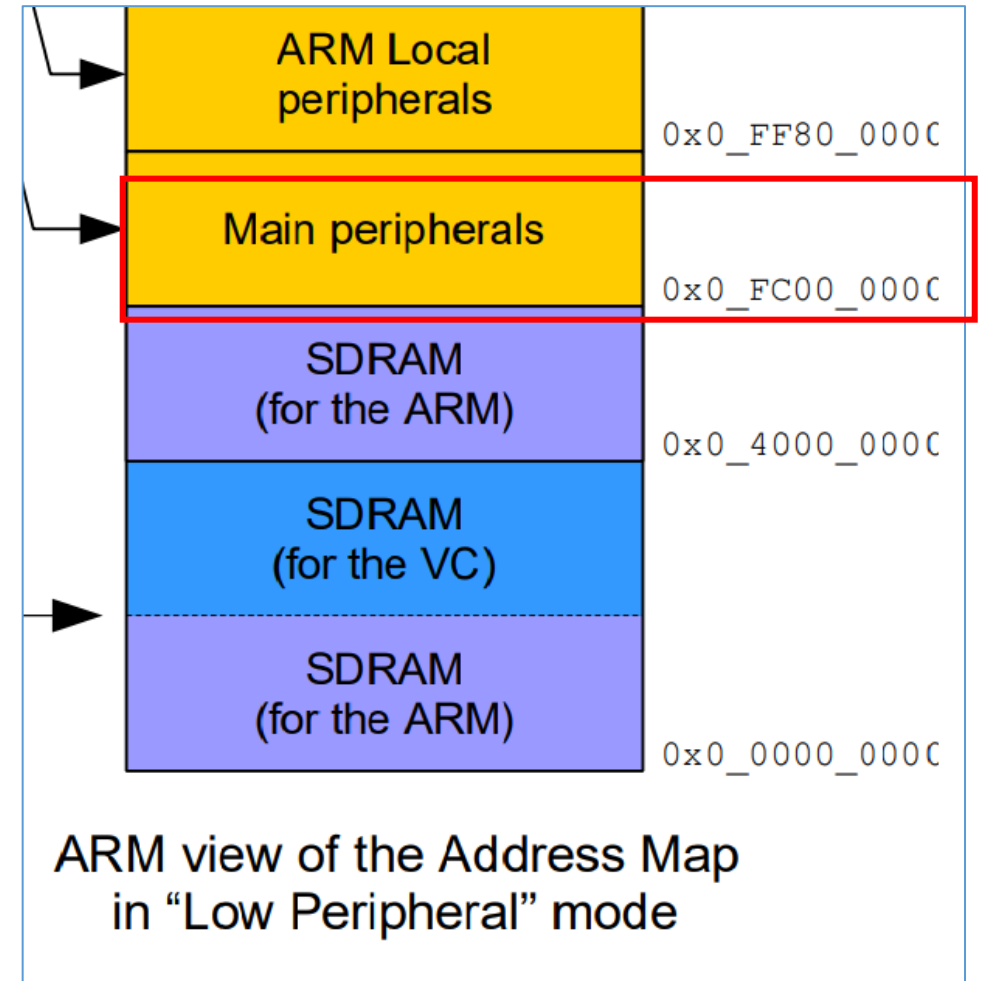
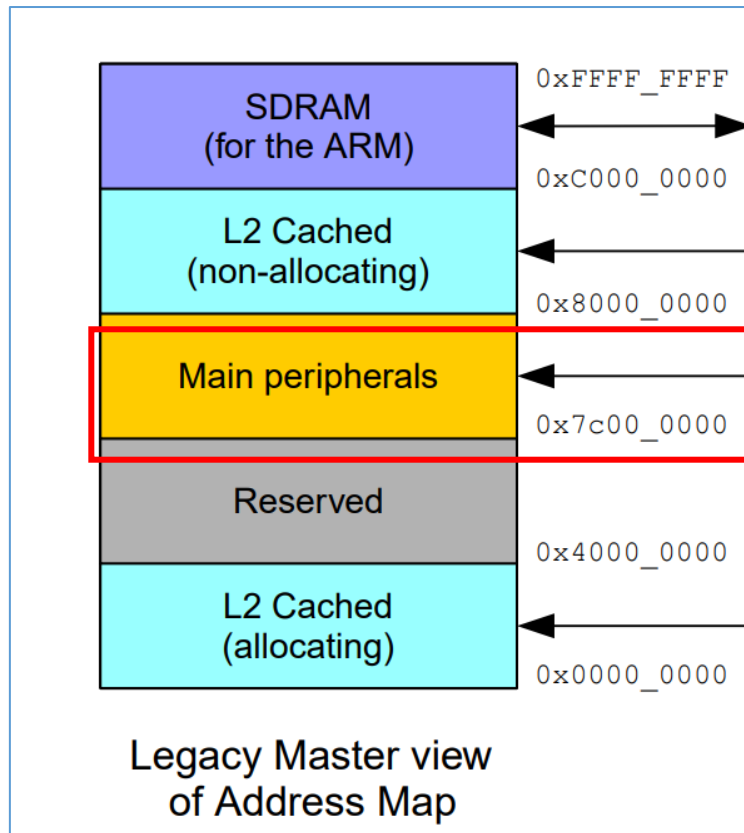
Legacy Master view
of Address Map

[QUIZ] Address 맞추기

Low Peripheral Mode 기준으로 GPIO 의 Base Address 는?

Legacy GPIO의 Base 주소 : 0x7E20 0000

Legacy Main Peripherals Base 주소 : 0x7C00 0000



BASE ADDRESS 계산

Legacy 기준 Main peripherals 시작 Address = 0x7C00 0000

datasheet 기준 GPIO Base Address = 0x7E20 0000

offset : $0x7E20\ 0000 - 0x7C00\ 0000 = 0x0220\ 0000$

Low Peripheral 기준 Main peripherals 시작 Address = 0xFC00 0000

$0xFC00\ 0000 + 0x0220\ 0000 = 0xFE20\ 0000$

즉, GPIO BASE ADDRESS 는 0xFE20 0000 이다.

참고: LPAE (Large Physical Address Extention)

ARM 칩셋에서 제공하는 LPAE 기능

32bit system에선 4GB 까지의 메모리만 사용가능하다.

- LAST ADDRESS 0xFFFF FFFF (32bit)
- 총 1Byte x 0xFFFF FFFF = 약 4GB

만약 그 이상을 쓰고자 한다면, 64bit system 사용해야 한다!

- 그러나 우리 raspbian os 는 32bit system 으로 LPAE 기능을 이용해 더 넓은 주소 공간을 사용할 수 있다!

LPAE 기능: 32비트가 아니라 실제로 **35비트를 사용**

- 가상 주소 공간을 이용해 더 넓게 사용 가능

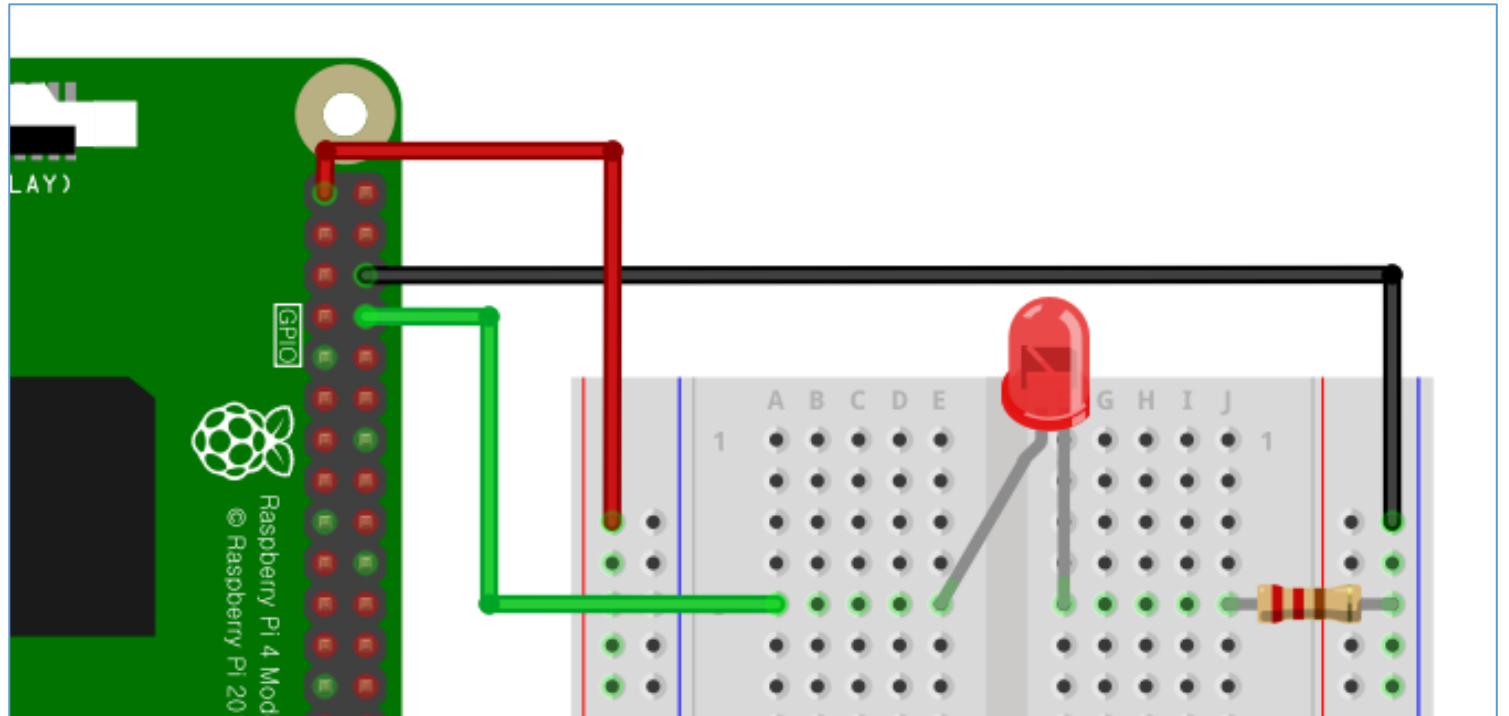
<https://developer.arm.com/documentation/ddi0464/f/Programmers-Model/Large-Physical-Address-Extension-architecture?lang=en>

회로 연결

회로를 연결한다.

LED (+) → GPIO14

LED (-) → 220옴 → GND



코드를 작성하고 make 한다.

~/test20 디렉토리 생성

datasheet 를 보고 개발하는 LED 제어 샘플 코드

```
76     BASE = (uint32_t*)ioremap(0xFE200000, 256);
77     GPFSEL1 = BASE + (0x04 / 4);
78     GPSET0 = BASE + (0x1C / 4);
79     GPCLR0 = BASE + (0x28 / 4);
80
81     *GPFSEL1 &= ~(0x7 << 12);
82     *GPFSEL1 |= (1 << 12);
```

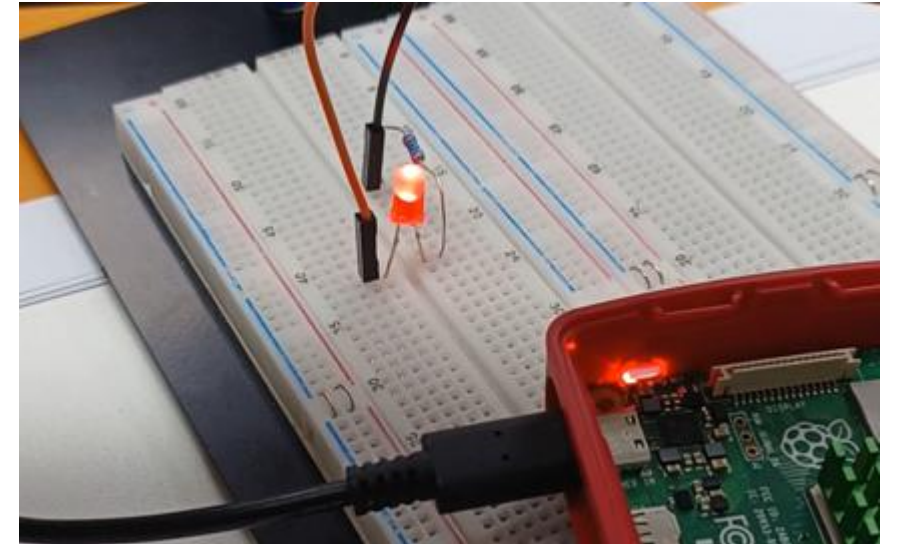
<https://gist.github.com/hoconoco/49530a44b67cb2a43f38300f2a11cac4>

동작 확인

app을 실행해서 숫자를 입력한다.

3 → LED ON

4 → LED OFF



```
pi@raspberrypi:~/test $ sudo insmod devicedriver.ko
pi@raspberrypi:~/test $ sudo chmod 666 /dev/deviceFile
pi@raspberrypi:~/test $ ./app
>>3
>>4
>>3
>>4
>>5
command invalid!
pi@raspberrypi:~/test $ sudo rmmod devicedriver
pi@raspberrypi:~/test $
```

```
[ 7067.927356] Insmod Module
[ 7067.930261] Major number 236
[ 7067.930289] Device file : /dev/deviceFile
[ 7076.742153] Open Device
[ 7077.793477] LED ON
[ 7078.289628] LED OFF
[ 7078.809162] LED ON
[ 7079.216749] LED OFF
[ 7079.578440] Close Device
[ 7086.748659] Unload Module
```

GPIO PIN Setup 순서

1. GPFSELx : GPIO Input 또는 Output을 결정 (핀 선택 후 OUTPUT)
2. GPSETx : GPIO Set (LED ON)
3. GPCLR x : GPIO Clear (LED OFF)

찾아야 하는 Address

1. GPFSEL 에서 14번 PIN
2. GPSET 에서 14번 PIN
3. GPCLR 에서 14번 PIN

5.2. Register View

The GPIO has the following registers. All accesses are assumed to be 32-bit. The GPIO register base address is **0x7e200000**.

Offset	Name	Description
0x00	GPFSEL0	GPIO Function Select 0
0x04	GPFSEL1	GPIO Function Select 1
0x08	GPFSEL2	GPIO Function Select 2
0x0c	GPFSEL3	GPIO Function Select 3
0x10	GPFSEL4	GPIO Function Select 4
0x14	GPFSEL5	GPIO Function Select 5
0x1c	GPSET0	GPIO Pin Output Set 0
0x20	GPSET1	GPIO Pin Output Set 1
0x28	GPCLR0	GPIO Pin Output Clear 0
0x2c	GPCLR1	GPIO Pin Output Clear 1

p.68 GPFSEL 레지스터

해당 gpio 핀을 출력 모드로 사용할 지, 입력 모드로 사용할 지 결정

000 → input

001 → output

GPFSEL0

GPIO pin : 0~9 까지 담당

GPFSEL1

GPIO pin : 10~19 까지 담당

Description

The function select registers are used to define the operation of the general-purpose I/O pins. Each of the 58 GPIO pins has at least two alternative functions as defined in [Section 5.3](#). The FSEL_n field determines the functionality of the *n*th GPIO pin. All unused alternative function lines are tied to ground and will output a "0" if selected. All pins reset to normal GPIO input operation.

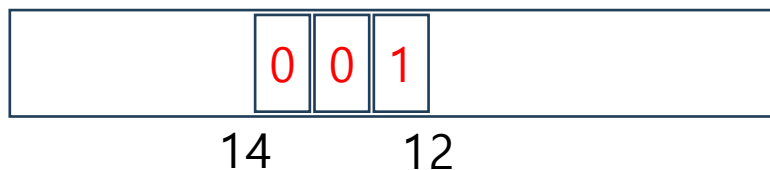
Bits	Name	Description	Type	Reset
31:30	Reserved.	-	-	-
29:27	FSEL9	FSEL9 - Function Select 9 000 = GPIO Pin 9 is an input 001 = GPIO Pin 9 is an output 100 = GPIO Pin 9 takes alternate function 0 101 = GPIO Pin 9 takes alternate function 1 110 = GPIO Pin 9 takes alternate function 2 111 = GPIO Pin 9 takes alternate function 3 011 = GPIO Pin 9 takes alternate function 4 010 = GPIO Pin 9 takes alternate function 5	RW	0x0

GPFSEL1 Register

GPFSEL1 Register

gpio pin : 10~19 까지 담당
GPIO14 → [14:12] bit
001 을 넣으면 output 사용!

GPFSEL1



```
81      *GPFSEL1 &= ~(0x7 << 12);  
82      *GPFSEL1 |= (1 << 12);
```

GPFSEL1 Register

Bits	Name	Description	Type	Reset
31:30	Reserved.	-	-	-
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	RW	0x0
26:24	FSEL18	FSEL18 - Function Select 18	RW	0x0
23:21	FSEL17	FSEL17 - Function Select 17	RW	0x0
20:18	FSEL16	FSEL16 - Function Select 16	RW	0x0
17:15	FSEL15	FSEL15 - Function Select 15	RW	0x0
14:12	FSEL14	FSEL14 - Function Select 14	RW	0x0
11:9	FSEL13	FSEL13 - Function Select 13	RW	0x0

p.71 GPSET 레지스터

해당 GPIO 핀에 HIGH 신호를 보내줄 때 사용

0 → 효과 없음

1 → set GPIO pin

GPSET0

GPIO pin : 0~31 담당

14bit 에 1 넣으면 됨

```
21 static void ledon(void){  
22     *GPSET0 = (1<<14);  
23 }
```

GPSET0 Register

Description

The output set registers are used to set a GPIO pin. The SET n field defines the respective GPIO pin to set, writing a "0" to the field has no effect. If the GPIO pin is being used as an input (by default) then the value in the SET n field is ignored. However, if the pin is subsequently defined as an output then the bit will be set according to the last set/clear operation. Separating the set and clear functions removes the need for read-modify-write operations

Bits	Name	Description	Type	Reset
31:0	SET n ($n=0..31$)	0 = No effect 1 = Set GPIO pin n	WO	0x00000000

p.71 GPCLR 레지스터

해당 GPIO 핀에 LOW 신호를 보내줄 때 사용

0 → 아무 효과 없음

1 → clear GPIO pin

GPCLR0

GPIO pin : 0~31 담당

14bit 에 1 넣으면 됨

```
24 static void ledoff(void){  
25     *GPCLR0 = (1<<14);  
26 }
```

GPCLR0 Register

Description

The output clear registers are used to clear a GPIO pin. The CLR n field defines the respective GPIO pin to clear, writing a "0" to the field has no effect. If the GPIO pin is being used as an input (by default) then the value in the CLR n field is ignored. However, if the pin is subsequently defined as an output then the bit will be set according to the last set/clear operation. Separating the set and clear functions removes the need for read-modify-write operations.

Bits	Name	Description	Type	Reset
31:0	CLR n ($n=0..31$)	0 = No effect 1 = Clear GPIO pin n	WO	0x00000000

ioremap() API

특정 메모리 공간을 커널 주소 공간에 매핑한다.

기준 주소 : 0xFE20 0000

mapping range : 256 bit

<asm/io.h> 필요

```
76      BASE = (uint32_t*)ioremap(0xFE200000, 256);  
77      GPFSEL1 = BASE + (0x04 / 4);  
78      GPSET0 = BASE + (0x1C / 4);  
79      GPCLR0 = BASE + (0x28 / 4);
```


GPIO 제어를 위한 레지스터 offset

GPIO Base Address : 0x7E20 0000

GPFSSEL1 : 0x04 offset

GPSET0 : 0x1c offset

GPCLR0 0x28 offset

/4 는 Base Address 자료형이 uint32t(=4byte) 이므로 나눠준다.

```
77      GPFSSEL1 = BASE + (0x04 / 4);  
78      GPSET0 = BASE + (0x1C / 4);  
79      GPCLR0 = BASE + (0x28 / 4);
```

5.2. Register View

The GPIO has the following registers:

0x7e200000.

Offset	Name
0x00	GPFSSEL0
0x04	GPFSSEL1
0x08	GPFSSEL2
0x0c	GPFSSEL3
0x10	GPFSSEL4
0x14	GPFSSEL5
0x1c	GPSET0
0x20	GPSET1
0x28	GPCLR0
0x2c	GPCLR1

감사합니다.