

Android Kernel Hacks

안드로이드를 위한 리눅스 커널 핵스



2013.7.16 ~ 2013.10.11

이 충 한(chunghan.yi@gmail.com, slowboot)

목차

머리말

Chapter 1. 안드로이드 소개

Chapter 2. 주요 커널 프로그래밍 기법/1

Chapter 3. 주요 커널 프로그래밍 기법/2

Chapter 4. ARM 보드 초기화 과정 분석

Chapter 5. 파워 관리(Power Management) 기법

Chapter 6. **주요 스마트폰 디바이스 드라이버 분석**

Chapter 7. 커널 디버깅 기법 소개

인덱스

CHAPTER 6

주요 스마트폰 디바이스 드라이버 분석



본 장에서는 안드로이드 기반 제품 개발 시 자주 등장하는 주요 디바이스 드라이버를 분석해 보고자 한다.

[본 장에서 소개할 내용]

- Peripheral 연결 Drivers
 - SD/MMC, USB, SPI, I2C, UART
- Input Device Drivers
 - touchscreen, keypad Drivers
 - 아날로그 Sensor Drivers
 - ◆ 가속도 감지 센서, 지자기 센서, 방향 감지 센서, 근접 센서, 자이로스코프 센서, 빛 감지 센서, 압력 감지 센서, 온도 감지 센서, 진동센서 ...
- Multimedia Drivers
 - Display Drivers(Framebuffer, LCD/backlight, Common Display Framework, HDMI, TVOut)
 - Camera Driver(V4L2, Camera sensor Driver, Camera flash driver)

- Audio Sound Driver
- Video Encoding/Decoding Driver
- 2D/3D Graphic Acceleration Drivers
- **Connectivity Drivers**
 - Wi-Fi Driver
 - Bluetooth Driver
 - NFC Driver
 - Virtual Ethernet Driver
 - GPS Driver

0 번 장에서는 스마트폰이나 태블릿 PC, Smart TV 등 개발 시 자주 등장하는 여러 디바이스 즉,

- 저장 장치(SD/MMC)
- 각종 주변 장치 인터페이스(USB, I2C, SPI, UART 등)
- Touchscreen, 각종 아날로그 센서(가속도, 근접, 조도, 지자기, 빛, 압력, 온도 센서)
- Display(LCD, HDMI), 2D/3D GPU
- Multimedia 장치(audio sound, video encoder/decoder, camera)
- 네트워크 장치(Wi-Fi, Bluetooth, NFC, Virtual Ethernet, Modem, GPS)

등을 살펴보고, 이를 디바이스 드라이버로 구현할 경우 고민해야 할 부분을 분석해 볼 것이다.

본 서에서는 ARM 기반의 SoC 혹은 보드에 관한 내용으로 설명을 제한하고 있는 바, 본 장에서 설명하는 주요 디바이스 드라이버는 이미 앞서 Chapter 4에서 설명한 바와 같이 아래의 두 가지 방식 중 하나로 표현될 수 있다.

1) Platform Device 방식

- 보드 및 SoC 파일에 platform device를 등록해 주어야 하며, platform driver의 probe 함수에서 인자 값으로 platform data를 전달 받아 사용한다.

2) Device Tree 방식

- 보드 및 SoC를 device tree 형태로 표현하고, 부트로더를 통해 부팅 시 메모리에 적재된다. 이후 platform driver의 probe 함수에서 of_XXX() 형태의 함수를 이용하여 platform data를 가져와 사용한다.

Platform device나 Device Tree가 필요한 이유는 SoC나 보드에 장착된 장치의 특성을 1차적으로 기술(초기화)하는 과정이 필요해서이며, 이후 platform driver의 probe 함수를 통해서 동적으로 구동하는 과정을 거치게 되는 것으로 보면 된다. 그림 6-1은 platform device 방식의 보드 초기화 과정을, 그림 6-2는 device tree 방식의 보드 초기화 과정을 표현하고 있다.

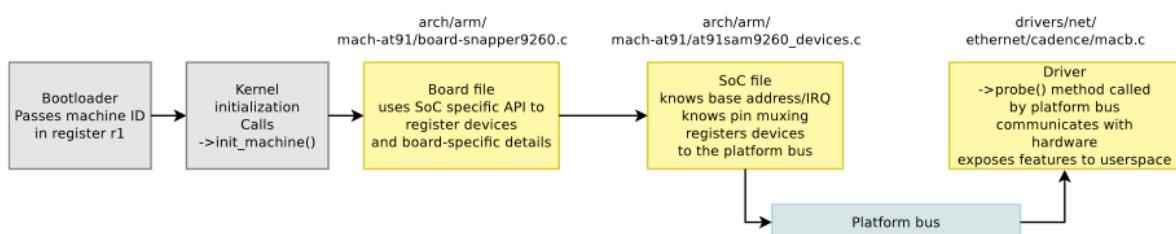


그림 6-1 Platform Device 방식 [출처 - 참고 문헌]

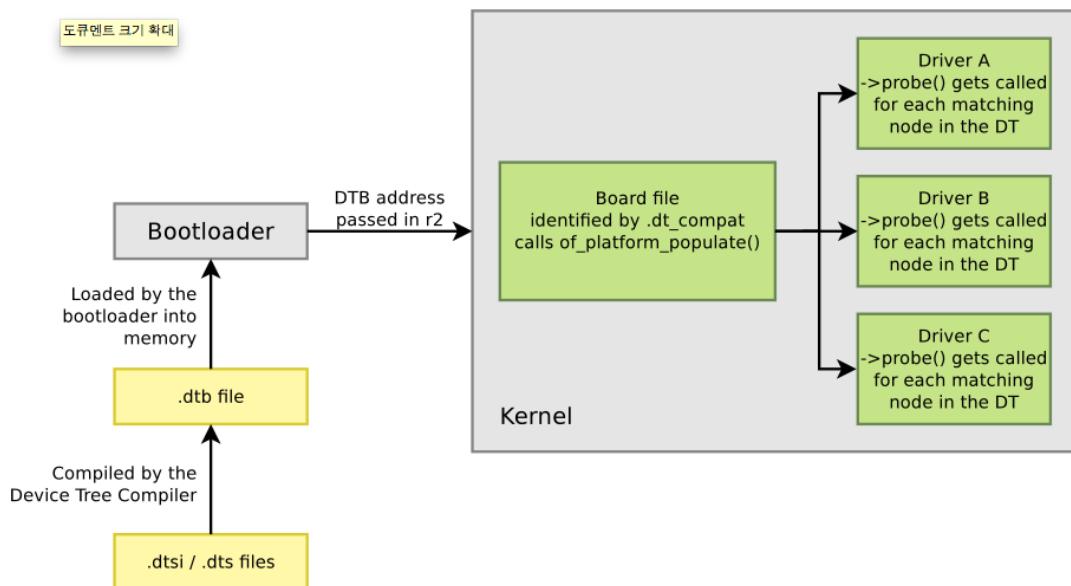


그림 6-2 Device Tree 방식 [출처 - 참고 문헌]

아직까지도 device tree 방식 보다는 platform device 방식이 많이 사용되고 있는 바, 본 장에서는 많은 부분에서 platform device & platform driver 형태로 드라이버 코드를 소개할 것이며, 필요 시 device tree를 이용한 방법을 언급할 것이다.

본 서에서 바라보는 리눅스 디바이스 드라이버 프로그래밍은 아래와 같은 특성이 있다고 볼 수 있다.

- 1) *Low-level* 코드를 직접 구현할 필요가 없이, 각 장치 별 *linux framework* 코드를 이해하고, 이를 활용할 수 있어야 한다.
 - SD/MMC framework, USB framework, I2C framework, SPI framework ...
 - 어떤 면에서는 매우 많은 *linux framework* 코드를 제대로 이해하는 것이 일일 수도 있겠다.
- 2) *SOC*를 기반으로 하는 경우만을 고려하고 있으므로, *platform device*(혹은 *device tree*) & *platform driver* 형태로 구현하여야 한다.
- 3) 대개의 경우, 유사한 드라이버 코드가 이미 구현되어 있으므로 이를 잘 활용하면, 손쉽게 새로운 드라이버를 구현할 수 있다.

본 장에서는 아래와 같은 형태로 개별 디바이스 드라이버 관련 내용을 기술하고자 한다. 즉, 본 장에서 역점을 두어 설명하고자 하는 부분은 상세한 배경 설명이 아니라, 실제로 디바이스 드라이버를 어떠한 형태로 구현하는지에 관하여 초점을 맞출 것이다.

<본 장의 서술 방향>

- 1) 디바이스의 특성을 개략적으로 소개한다.
- 2) 리눅스에서 해당 디바이스를 드라이버로 구현하기 위해 마련해 둔, 프레임워크(혹은 서브시스템) 부분을 간략히 소개한다.
- 3) 실제로 *platform device* & *platform driver* 혹은 *device tree* & *platform driver* 형태의 예제를 소개함으로써, 관련 드라이버를 제대로 이해할 수 있도록 한다.

1. SD/MMC 저장 장치 드라이버

e-MMC 장치 소개

e-MMC 장치는 저비용, 고성능 데이터 저장 장치로써, 무선이나 임베디드 영역에서 다양하게 사용할 목적으로 설계되었다. e-MMC는 데이터나 코드를 저장하는 용도로 사용되는 것은 물론이고, 부팅 용으로도 사용될 수 있다.

e-MMC는 NAND flash 메모리 장치와 e-MMC 컨트롤러를 하나의 패키지로 묶어 놓은 형태로 되어 있다. e-MMC 컨트롤러 내에는 FTL(flash translation layer), WL(wear leveling), BBM(bad block management) 같은 펌웨어 모듈이 동작 중에 있다. e-MMC 아키텍쳐는 표준 인터페이스 스펙을 제공함으로써 하드웨어 및 소프트웨어의 통합을 단순화시켜 주는 특징이 있다. 그림 6-3은 NAND flash와 e-MMC 메모리를 비교한 것이다.

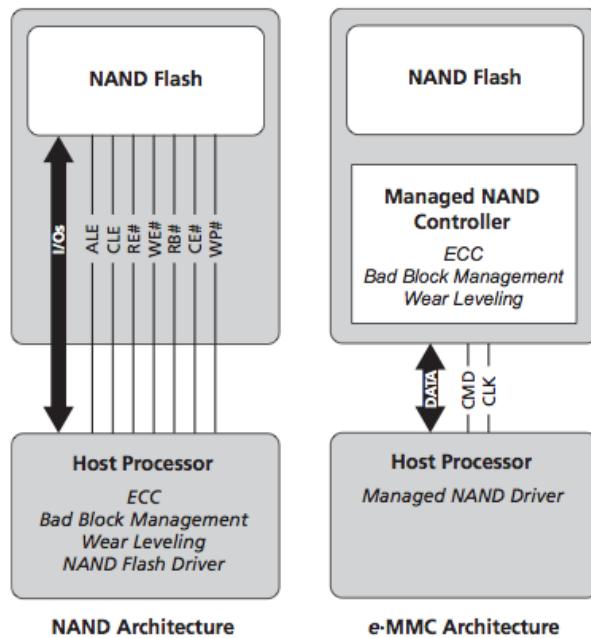


그림 6-3 NAND 플래쉬 메모리와 e-MMC 메모리 비교 [출처 - 참고 문헌 11]

리눅스 SD/MMC 서브시스템

리눅스의 e-MMC 서브시스템(drivers/mmc 디렉토리)은 아래의 세가지 계층으로 구성되어 있다.

1) card layer

- 외부 인터페이스로, block 디바이스 드라이버처럼 보인다.

2) core layer

- e-MMC 관련 핵심 기능을 구현한다. 즉, main operations, command management, host setting, management functions, and high-level functions 등 ...

3) host layer

- 플랫폼 종속적인 플랫폼 컨트롤러 드라이버를 구현한다.

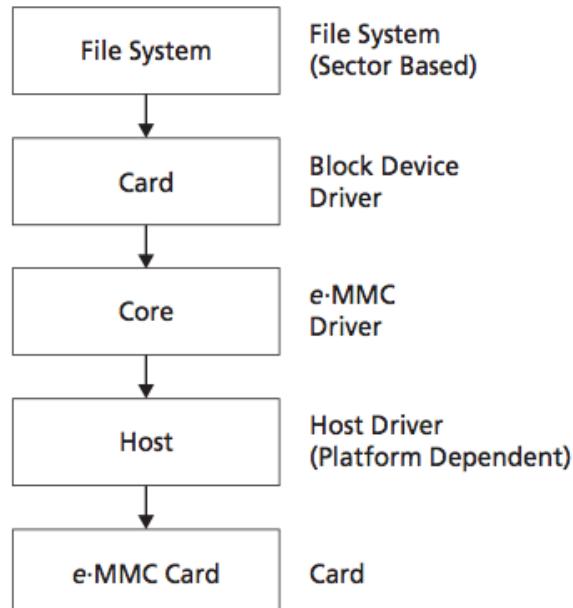


그림 6-4 리눅스 e-MMC 서브시스템(1)

실제로 e-MMC는 특정 파일 시스템 형태로 관리가 되어야 하기 때문에, 리눅스 Block Device Driver 계층과 연결되어야 한다. 그림 6-5는 이를 표현한 것으로, 그림의 맨 밑에 있는 Block Layer 계층 아래에 그림 6-6에 있는 e-MMC 서브시스템이 연결되게 된다.

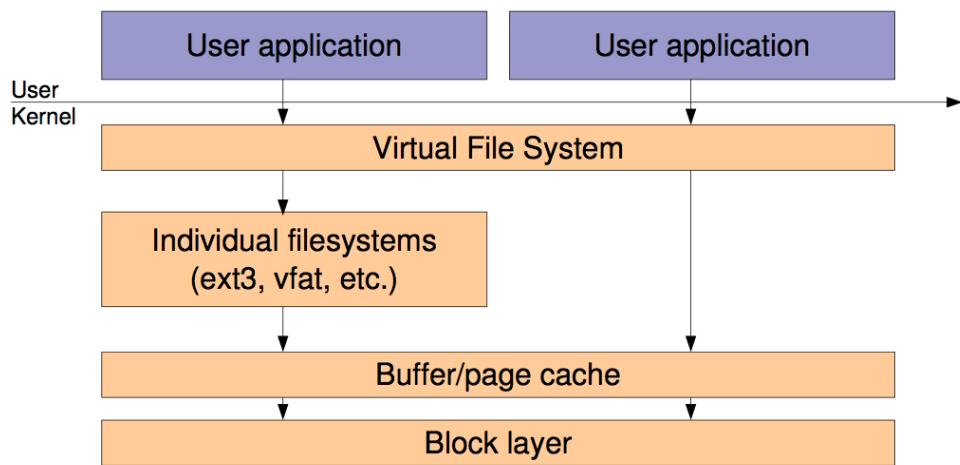


그림 6-5 리눅스 블록 드라이버 계층

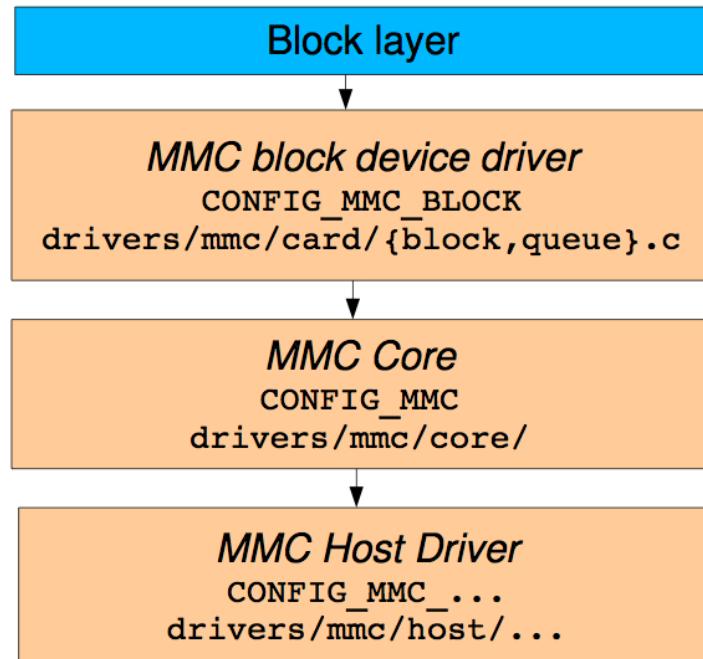


그림 6-6 리눅스 e-MMC 서브시스템(2)

그림 6-4 ~ 6까지를 다른 관점(e-MMC 장치와 호스트 컨트롤러)에서 그려 보면 그림 6-7과 같다.

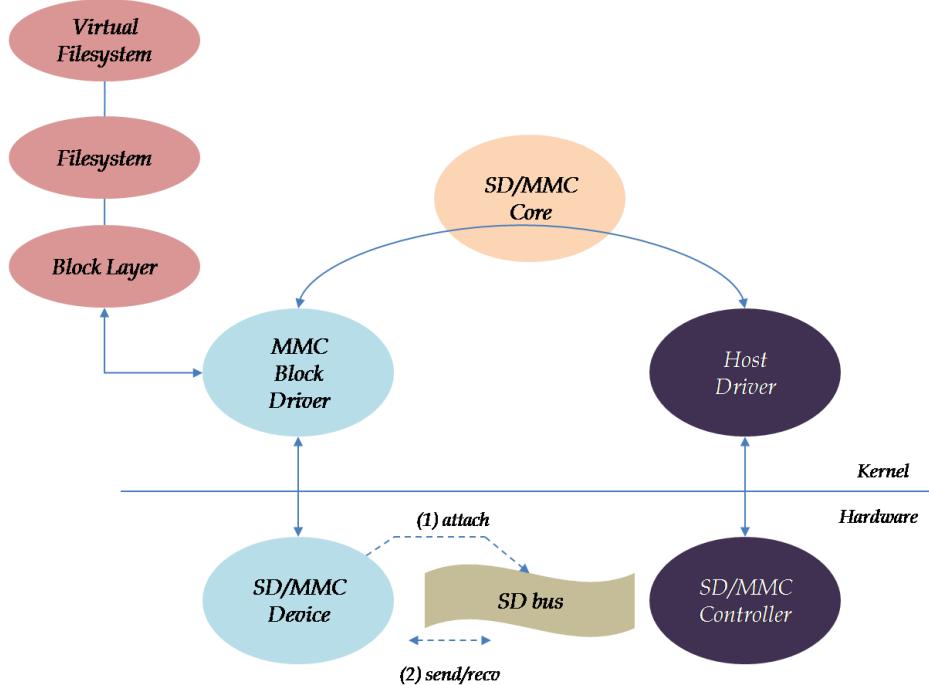


그림 6-7 리눅스 e-MMC 서브시스템(3)

MMC Host 드라이버의 동작 방식을 간략히 설명하면 다음과 같다.

<MMC Host 드라이버 동작 방식>

1) *Probe*시 *mmc_host* 및 *operation*을 할당하고, 몇 가지 *interrupt* 요청을 등록한 후, *mmc host*를 추가해준다.

2) 응용 프로그램으로부터 요청이 있을 경우, *mmc_host*의 *operation* 중 *request()* handler가 함수가 호출되면서, *command*, *data*등이 전송되기 시작한다. 이 부분은 SD/MMC core code에 구현되어 있으며, *struct mmc_host_ops*의 *request callback* 함수를 통해 실제 물리 장치로 데이터가 전송되는 구조로 되어 있다.

- *cmd* 전송 후, 전달할 *data*가 없으면, *mmc_request_done* 함수를 호출하여 *mmc request operation*을 종료한다.
- *cmd* 전송 후, 전달할 *data*가 있으면 이를 전송하고, 전송 완료 후에는 *mmc_request_done* 함수를 호출한다.

그림 6-8은 request operation의 대략적인 흐름을 나타내고 있다.

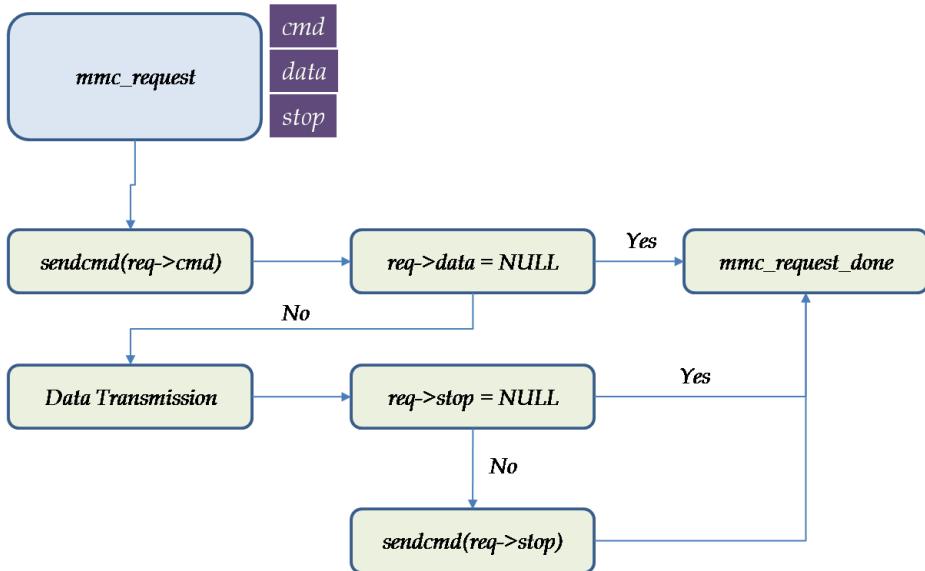


그림 6-8 MMC Request 흐름 분석

<Probe 단계 주요 API>

```
struct mmc_host *mmc_alloc_host(int extra, struct device *dev)
```

- `struct mmc_host` 의 각 필드를 초기화 시킨다.
- 즉, `caps`, `ops`, `max_phys_segs`, `max_hw_segs`, `max_blk_size`, `max_blk_count`, `max_req_size`

```
int mmc_add_host(struct mmc_host *host)
```

<Remove 단계 주요 API>

```
void mmc_remove_host(struct mmc_host *host)
```

```
void mmc_free_host(struct mmc_host *host)
```

- 참고로, `mmc_host->ops` 필드는 `mmc_host_ops` structure를 가리킨다.

<I/O request 처리 함수>

```
void (*request)(struct mmc_host *host, struct mmc_request *req);
```

<configuration setting 함수>

```
void (*set_ios)(struct mmc_host *host, struct mmc_ios *ios);
```

<read-only status 구하는 함수>

```
int (*get_ro)(struct mmc_host *host);
```

<card 존재 상태 확인 함수>

```
int (*get_cd)(struct mmc_host *host);
```

MMC Client 드라이버 & 호스트 드라이버 예제 소개

1) MMC 장치 추가하기(mmc/card/block.c)

아래 코드는 struct mmc_driver를 사용하여 mmcblk이라는 이름의 block 드라이버를 생성하는 예를 보여주고 있다.

코드 6-1

```
<TODO>
```

2) MMC 호스트 장치 드라이버

2.1) MMC 호스트 장치 선언하기(platform device)

아래 코드는 MMC 호스트 컨트롤러를 선언하는 부분에 관한 것이다.

코드 6-2

```
<TODO>
```

2.2) MMC 호스트 드라이버 추가하기

아래에는 OMAP4에서 사용하는 MMC 호스트 드라이버의 예제를 소개하였다.

코드 6-3 MMC 호스트 드라이버

From drivers/mmc/host/omap.c

```
static int mmc_omap_probe(struct platform_device *pdev)
{
    [...]
    for (i = 0; i < pdata->nr_slots; i++) {
        ret = mmc_omap_new_slot(host, i);
        if (ret < 0) {
            while (--i >= 0)
                mmc_omap_remove_slot(host->slots[i]);

            goto err_destroy_wq;
        }
    }
    [...]
}

static const struct mmc_host_ops mmc_omap_ops = {
    .request      = mmc_omap_request,
```

```

.set_ios      = mmc omap_set_ios,
};

static int mmc_omap_new_slot(struct mmc_omap_host *host, int id)
{
    struct mmc_omap_slot *slot = NULL;
    struct mmc_host *mmc;
    int r;

    mmc = mmc_alloc_host(sizeof(struct mmc_omap_slot), host->dev);
    if (mmc == NULL)
        return -ENOMEM;

    slot = mmc_priv(mmc);
    slot->host = host;
    slot->mmc = mmc;
    slot->id = id;
    slot->pdata = &host->pdata->slots[id];

    host->slots[id] = slot;
    [...]
    mmc->ops = &mmc_omap_ops;
    mmc->f_min = 400000;

    if (mmc_omap2())
        mmc->f_max = 48000000;
    else
        mmc->f_max = 24000000;
    [...]
    mmc->max_segs = 32;
    mmc->max_blk_size = 2048; /* BLEN is 11 bits (+1) */
    mmc->max_blk_count = 2048; /* NBLK is 11 bits (+1) */
    mmc->max_req_size = mmc->max_blk_size * mmc->max_blk_count;
    mmc->max_seg_size = mmc->max_req_size;

    r = mmc_add_host(mmc);
    [...]
}

```

[From drivers/mmc/core/core.c](#)

```

static void
mmc_start_request(struct mmc_host *host, struct mmc_request *mrq)
{
    [...]
    mmc_host_clk_hold(host);
    led_trigger_event(host->led, LED_FULL);
    host->ops->request(host, mrq);
}

static int mmc_wait_for_data_req_done(struct mmc_host *host,
                                      struct mmc_request *mrq,
                                      struct mmc_async_req *next_req)
{
    [...]
    host->ops->request(host, mrq);
    [...]
}

static void mmc_wait_for_req_done(struct mmc_host *host,
                                  struct mmc_request *mrq)
{
    struct mmc_command *cmd;

    while (1) {
        [...]
        host->ops->request(host, mrq);
        [...]
    }
}

```

참고로, 이번 절에서는 SD/MMC 관련 부분만을 언급하였으며, SDIO 관련 내용은 Wi-Fi 장에서 다시 설명할 것이다.

2. USB 가젯(gadget) 드라이버

USB 개요

리눅스는 현재 모바일 장치에서 폭넓게 사용되고 있다. 리눅스가 자랑하는 장점 중의 하나는 아마도 모든 종류의 USB 장치에 대한 지원일 것이다. 사용자들이 자신들의 모바일 장치에 많은 데이터를 저장하게 되면서, PC와 모바일 장치를 연결해야 할 필요성이 증가하게 되었다. 아마 어느 누구도 자신이 갖고 있는 모바일 장치 안의 몇 장의 사진을 PC로 옮기기 위하여 별도의 전용 케이블을 구매하기를 원하지는 않을 것이다. 일반 사용자들이 원하는 것은 단순히 USB 케이블을 연결하고, 파일을 PC로 바로 옮기는 것일 것이다. 지금부터는 리눅스에서 제공하는 USB의 기본 개념에 관하여 살펴 볼 것인데, USB 호스트 보다는 사용자 장치 즉, USB gadget 드라이버의 구현에 초점을 맞추어 설명을 진행할 것이다.

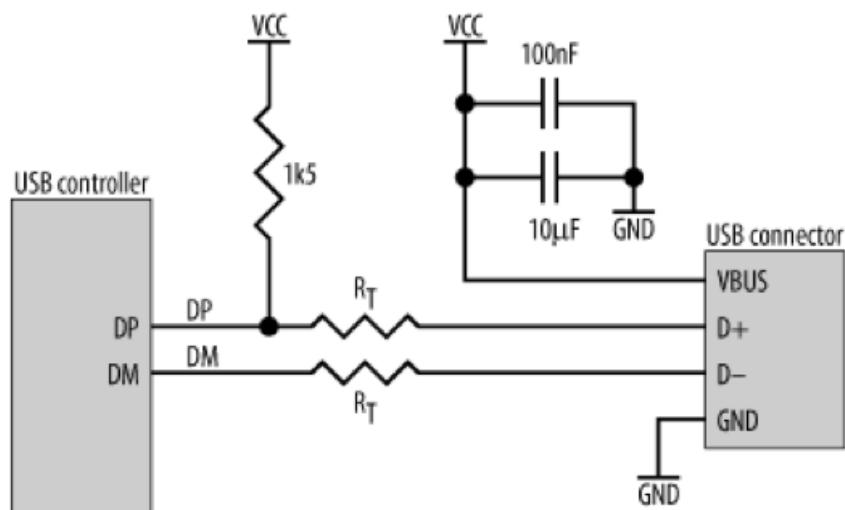


그림 6-9 USB(Universal Serial Bus) H/W [출처 - 참고 문헌]

그림 6-9에서 확인할 수 있듯이, USB는 4개의 단자로 구성되어 있으며, 각각의 의미를 설명하면 다음과 같다.

1) VBUS

- 적색으로 되어 있으며, USB 장치 전원(+5V)이다.

2) D+

- 녹색으로 되어 있으며, 동축 데이터 선이다.

3) D-

- 흰색으로 되어 있으며, 동축 데이터 선이다.

4) GND

- 검정색으로 되어 있으며, 전원과 신호 접지(ground)의 역할을 한다.

USB(Universal Serial Bus)는 master와 slave 장치 간의 시리얼 통신 프로토콜로써, 이는 한쪽에는 USB 케이블을 통해 데이터를 전송하는 객체(entity) 즉, master(혹은 host)가 있고, 반대쪽에는 master로 부터의 명령 요청을 처리하는 다른 객체 즉 slave(device 혹은 gadget이라고도 함)가 존재하는 방식을 말한다. Host(master)는 대개 데스크 탑 컴퓨터(PC)가 그 역할을 수행하며,

gadget(slave or device)은 마우스, 키보드, 핸드폰, 프린터 등이 그 역할을 한다고 이해하면 쉽다. 일반적으로 사람들이 생각하기에 리눅스가 올라간 시스템이 USB host(master)의 역할만을 할 것으로 생각하지만, 실제로는 리눅스 시스템이 gadget(slave)으로써의 역할을 하기도 하며, 경우에 따라서는 host와 gadget의 역할을 둘 다 수행하기도 한다(이런 경우를 OTG(On-The-Go)라고 부르며, Host Negotiation Protocol을 사용하여 자신의 역할을 결정짓게 된다).

USB 디바이스 디스크립터

USB 장치는 아래와 같이 여러 디스크립터(descriptor)로 구성되어 있다.

Device

- USB 버스에 연결된 장치를 나타낸다. 예: 볼륨 조절 버튼이 있는 USB 스피커

Configurations

- 장치의 상태를 나타낸다. 예: Active, Standby, Initialization

Interfaces

- 논리 장치를 뜻한다. 예: 스피커, 볼륨 조절 버튼

Endpoints

- 단 방향 통신 파이프로, IN(장치에서 호스트 방향) 혹은 OUT(호스트에서 장치 방향)으로 구분한다.

그림 6-10에서 확인 가능하듯이, endpoint는 여러 개를 모아 하나의 interface로 묶여지게 되며, interface는 다시 여러 개를 모아 configuration으로 통합된다. 서로 다른 configuration은 서로 다른 interface들을 포함할 수 있으며, 서로 다른 전력 요구 사항(power demands)를 가질 수 있다. 이러한 모든 정보는 host가 enumeration 과정 중에 요청한 다양한 descriptor에 저장되게 된다.

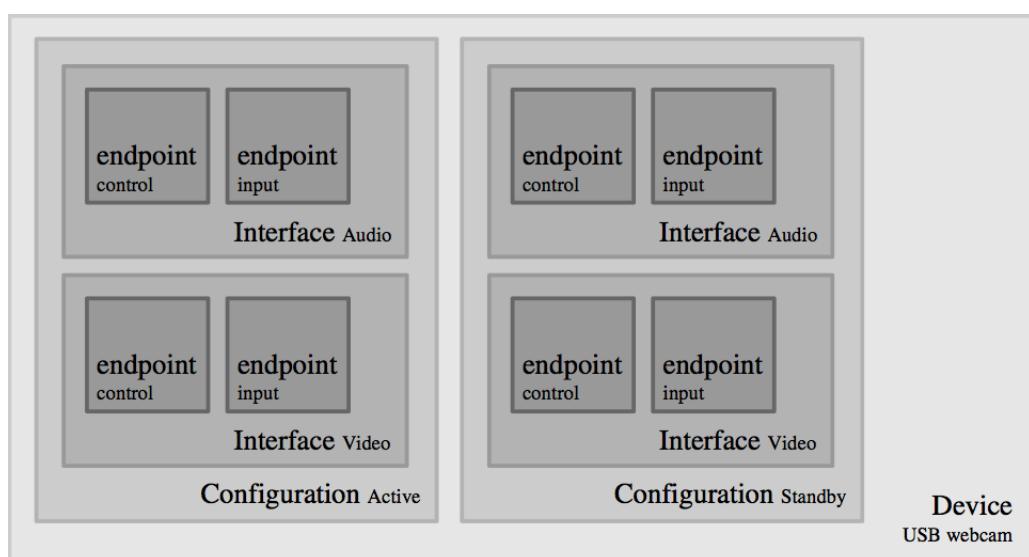


그림 6-10 USB 디바이스

USB Enumeration

장치 하나를 USB로 연결하면, enumeration 과정이 시작된다. 이 과정 동안에, 장치는 고유한 7bit의 이름(identifier)을 할당 받게 되는데, 결론적으로 127개(hub 포함)개의 slave가 하나의 host에 연결될 수 있다. USB 통신은 master와 slave의 여러 endpoint(EP라고도 함) 중 하나를 연결해주는 논리 파이프에 기초하고 있다. 장치마다 16개(0 ~ 15)의 endpoint가 존재할 수 있는데, endpoint 0(EPO)은 setup 요청(예를 들어, descriptor에 대한 질문, 특정 configuration을 설정하라는 요청 등)을 위해 사용된다. 파이프는 단일 방향(one-way)이며, 데이터는 IN endpoint를 통해 host로 들어 가거나, OUT endpoint를 통해 host로부터 나올 수 있다. Slave 관점에서 보면, IN endpoint는 host에 데이터를 보내는 쪽이 되며, OUT endpoint는 host로부터 데이터를 읽는 쪽에 해당한다. 즉, IN, OUT은 host 기준으로 설명하여야 한다.

USB 전송 모드 및 패킷 형식

USB 데이터 전송 모드에는 다음의 4가지 방법이 존재한다.

Bulk 전송 모드

- USB를 통해 비동기적으로 대량의 데이터를 전달하는데 사용한다. 시간에 민감하지 않은 데이터 전송 시 주로 사용된다.

Isochronous 전송 모드

- 단 방향 데이터 전송 시 사용되는 방법으로, 음성 데이터를 출력하는 경우와 같이 시간이 매우 중요한 실시간 데이터를 일정 간격으로 이동시키는데 사용된다.

Interrupt 전송 모드

- 시간에 민감한 소량의 데이터를 교환하는 용도로 사용된다.

Control 전송 모드

- USB 버스와 버스에 붙어 있는 장치의 환경을 설정하고, 상태 정보를 반환하는데 사용한다.

데이터는 패킷(packet)을 사용하여 USB 장치 사이를 오고 간다. 패킷은 Sync 바이트, PID(패킷 식별자), Data, CRC로 이루어져 있다.

Host가 descriptor에 정보를 얻어서, 어떠한 종류의 gadget들이 자신에게 연결되어 있는지를 알게 되는 순간, host는 하나의 configuration을 선택하게 되고, 이어 USB 통신이 시작되게 된다. 한번에 하나의 configuration만이 선택되고 사용될 수 있는 것이다.

리눅스 USB Composite Framework

예전에는 gadget 드라이버를 만들기 위해서, 하나의 단일 모듈에 모든 기능을 구현해 넣어야만 했다. 이러한 접근법은 아래와 같은 두 개의 커다란 문제점을 안고 있었다.

1) 대부분의 공통적인 USB 기능(즉, EPO를 통한 core device 설정 요청)이 각각의 드라이버 별로 구현이 되었다.

2) 이들을 하나로 묶어 새로운 gadget을 만드는 것이 매우 어려웠다.

이러한 이유 때문에, David Brownell은 새로운 장점을 갖고 있는 USB Composite Framework을 개발하게 되었다.

- 1) 모든 핵심 USB 요청(request)은 프레임워크 내에서 개발한다.
- 2) 단일 기능 혹은 USB composite function은 별도로 개발한다. 나중에, 이 function 들은 composite gadget을 만들기 위해 composite function을 사용하여 통합한다.

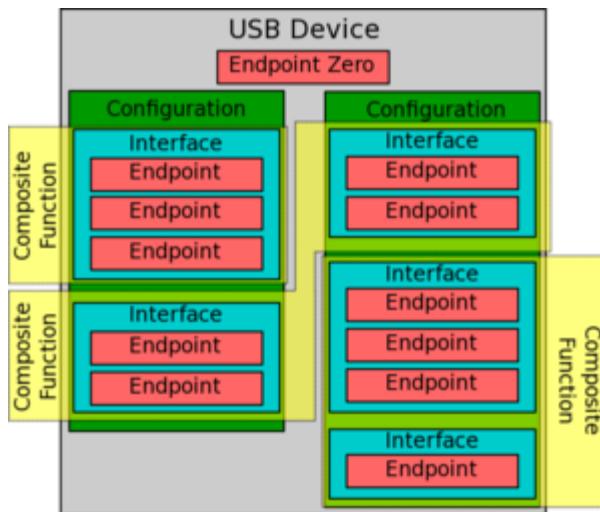


그림 6-11 USB Composite Gadget 디스크립터 구조

Composite gadget의 관점에서 보면, 각각의 장치는 여러 function을 가지게 되며, 이들은 여러 configuration으로 묶일 수 있다. 즉, 하나의 function은 여러 개의 configuration 내에 존재할 수 있는 것이다. 각각의 function은 다시 여러 개의 interface와 서로 다른 descriptor를 가질 수 있다. 그리고, composite 프레임워크를 사용하게 되면, USB 관련 세세한 구현을 피할 수가 있다. 즉, USB gadget 드라이버 개발자는 복잡한 endpoint나 interface 등에 대해 고민할 필요가 없이, 오로지 자신의 function에 대해서만 생각하면 된다.

리눅스 Gadget 드라이버 구조 이해

여기서부터는 ethernet gadget 드라이버를 만드는 방법을 소개해 보고자 한다. 리눅스의 좋은 점 중의 하나는 이미 많은 부분이 구현되어 있다는 점이다. 따라서, 적은 노력으로도 원하는 결과를 쉽게 얻을 수가 있다.

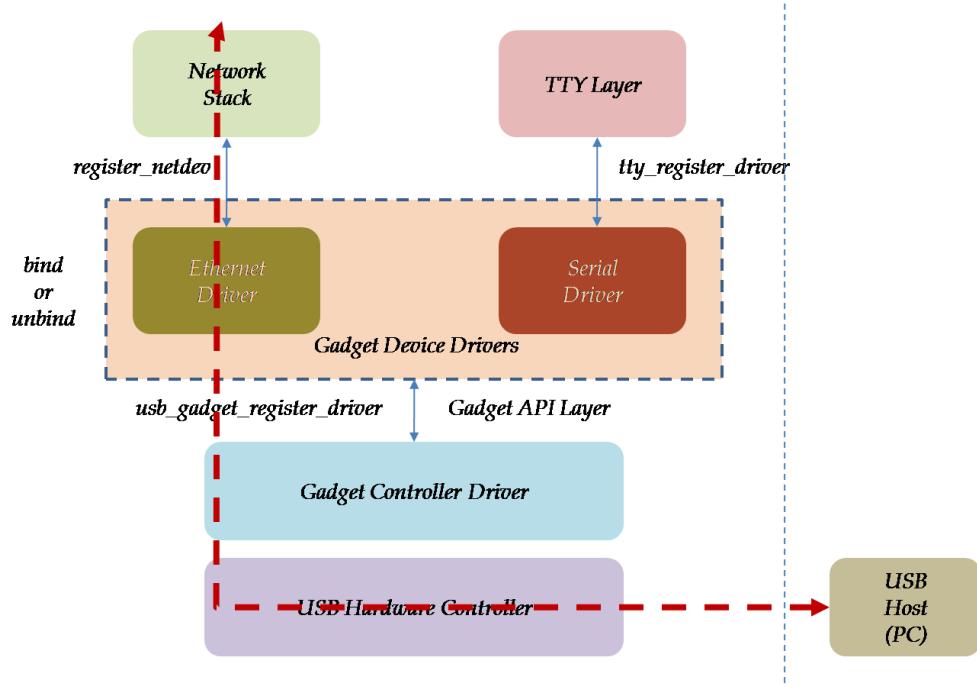


그림 6-12 USB Ethernet Gadget 드라이버 구조

제일 먼저 해야 할 일은 아래와 같이 장치 descriptor를 정의하는 것이다. 이는 gadget 관련하여 몇 가지 기본적인 정보를 담고 있게 된다.

코드 6-4

From [drivers/usb/gadget/ether.c](#)

```

153 static struct usb_device_descriptor device_desc = {
154     .bLength =           sizeof(device_desc),
155     .bDescriptorType =   USB_DT_DEVICE,
156
157     .bcdUSB =            cpu_to_le16(0x0200),
158
159     .bDeviceClass =       USB_CLASS_COMM,
160     .bDeviceSubClass =    0,
161     .bDeviceProtocol =   0,
162     /* .bMaxPacketSize0 = f(hardware) */
163
164     /* Vendor and product id defaults change according to what configs
165      * we support. (As does bNumConfigurations.) These values can
166      * also be overridden by module parameters.
167      */
168     .idVendor =           cpu_to_le16(CDC_VENDOR_NUM),

```

```

169     .idProduct =             cpu_to_le16 (CDC_PRODUCT_NUM),
170     /* .bcdDevice = f(hardware) */
171     /* .iManufacturer = DYNAMIC */
172     /* .iProduct = DYNAMIC */
173     /* NO SERIAL NUMBER */
174     .bNumConfigurations =    1,
175 };

```

실제로 `usb_device_descriptor` structure는 더 많은 필드를 갖고 있으나, 여기서는 필요치 않다.

bLength and bDescriptorType

- 각각의 descriptor가 가지게 되는 표준 필드

bcdUSB

- 장치가 지원하는 USB 스펙 버전으로, BCD 형식으로 인코딩되어 있음. 그래서, 0x200은 2.00 버전을 뜻함.

idVendor and idProduct

- 각각의 장치는 고유의 vendor 및 product ID 값을 쌍을 가지고 있게 됨.

다음 단계로는 `usb_configuration` structure를 사용하여 드라이버가 제공하는 USB configuration을 정의하는 것이다.

코드 6-5

From `drivers/usb/gadget/ether.c`

```

230 static struct usb_configuration rndis_config_driver = {
231     .label                 = "RNDIS",
232     .bConfigurationValue   = 2,
233     /* .iConfiguration = DYNAMIC */
234     .bmAttributes          = USB_CONFIG_ATT_SELFPOWER,
235 };

267 static struct usb_configuration eth_config_driver = {
268     /* .label = f(hardware) */
269     .bConfigurationValue   = 1,
270     /* .iConfiguration = DYNAMIC */
271     .bmAttributes          = USB_CONFIG_ATT_SELFPOWER,
272 };

```

`eth_config_driver` 오브젝트(객체)는 label 값, bind callback(위의 경우는 없음), configuration

number, device 속성(USB_CONFIG_ATT_SELFPOWER) 등을 지정한다. 이러한 configuration 정보는 아래에서 설명할 bind callback(eth_bind) 함수에 의해 장치에 연결(usb_add_config() 함수)되게 된다.

eth_bind() 함수가 하는 역할은 다음과 같다.

- composite function을 설정하고,
- descriptor를 준비하며,
- 장치가 지원하는 모든 configuration을 추가해 준다.

코드 6-6

From drivers/usb/gadget/ether.c

```
276 static int __init eth_bind(struct usb_composite_dev *cdev)
277 {
278     struct usb_gadget      *gadget = cdev->gadget;
279     int                      status;
280
281     /* set up network link layer */
282     the_dev = gether_setup(cdev->gadget, hostaddr);
283     if (IS_ERR(the_dev))
284         return PTR_ERR(the_dev);
285
286     /* set up main config label and device descriptor */
287     if (use_eem) {
288         /* EEM */
289         eth_config_driver.label = "CDC Ethernet (EEM)";
290         device_desc.idVendor = cpu_to_le16(EEM_VENDOR_NUM);
291         device_desc.idProduct = cpu_to_le16(EEM_PRODUCT_NUM);
292     } else if (can_support_ecm(cdev->gadget)) {
293         /* ECM */
294         eth_config_driver.label = "CDC Ethernet (ECM)";
295     } else {
296         /* CDC Subset */
297         eth_config_driver.label = "CDC Subset/SAFE";
298
299         device_desc.idVendor = cpu_to_le16(SIMPLE_VENDOR_NUM);
300         device_desc.idProduct = cpu_to_le16(SIMPLE_PRODUCT_NUM);
301         if (!has_rndis())
302             device_desc.bDeviceClass = USB_CLASS_VENDOR_SPEC;
```

```

303     }
304
305     if (has_rndis()) {
306         /* RNDIS plus ECM-or-Subset */
307         device_desc.idVendor = cpu_to_le16(RNDIS_VENDOR_NUM);
308         device_desc.idProduct = cpu_to_le16(RNDIS_PRODUCT_NUM);
309         device_desc.bNumConfigurations = 2;
310     }
311
312     /* Allocate string descriptor numbers ... note that string
313      * contents can be overridden by the composite_dev glue.
314      */
315
316     status = usb_string_ids_tab(cdev, strings_dev);
317     if (status < 0)
318         goto fail;
319     device_desc.iManufacturer = strings_dev[USB_GADGET_MANUFACTURER_IDX].id;
320     device_desc.iProduct = strings_dev[USB_GADGET_PRODUCT_IDX].id;
321
322     /* register our configuration(s); RNDIS first, if it's used */
323     if (has_rndis()) {
324         status = usb_add_config(cdev, &rndis_config_driver,
325                                rndis_do_config);
326         if (status < 0)
327             goto fail;
328     }
329
330     status = usb_add_config(cdev, &eth_config_driver, eth_do_config);
331     if (status < 0)
332         goto fail;
333
334     usb_composite_overwrite_options(cdev, &coverwrite);
335     dev_info(&gadget->dev, "%s, version: " DRIVER_VERSION "%n",
336              DRIVER_DESC);
337
338     return 0;
339
340 fail:
341     gether_cleanup(the_dev);

```

```
342         return status;
343 }
```

이제 composite 장치를 정의할 수 있는 준비가 모두 끝났다. `usb_composite_driver` structure를 사용하여 아래와 같이 composite 장치를 정의해 보자. 여기서 눈 여겨 보아야 할 부분은 device descriptor 부분과 bind callback 함수 부분이다.

코드 6-7

From `drivers/usb/gadget/ether.c`

```
351 static __refdata struct usb_composite_driver eth_driver = {
352     .name          = "g_ether",
353     .dev           = &device_desc,
354     .strings       = dev_strings,
355     .max_speed    = USB_SPEED_SUPER,
356     .bind          = eth_bind,
357     .unbind        = _exit_p(eth_unbind),
358 };
```

여기까지 작업하게 되면, 이제 남은 것은 `init()`과 `exit()` 모듈 함수를 정의하는 일일 것이다.

코드 6-8

From `drivers/usb/gadget/ether.c`

```
364 static int __init init(void)
365 {
366     return usb_composite_probe(&eth_driver);
367 }
368 module_init(init);
369
370 static void __exit cleanup(void)
371 {
372     usb_composite_unregister(&eth_driver);
373 }
374 module_exit(cleanup);
375
```

리눅스 USB composite framework은 꽤 직관적으로 USB 장치를 추가하는 방법을 제공하고 있다. 앞서도 이미 말했다시피, composite framework이 있기 전에는, 개발자들이 모든 USB 요청 사항을 구현해야만 했다. Composite framework은 기본적인 USB 요청을 처리해주고, 각각의 composite function으로 분기시켜 주는 일을 담당한다. 따라서, 이는 gadget 드라이버 개발자들이 USB 관련 low-level의 인터페이스나, 통신 관련 코드를 작성할 필요 없이, 자신들의 함수에 관해서만 고민할 수 있도록 범위를 좁혀 주게 되었다.

3. Input 디바이스 드라이버

본 장에서는 터치스크린, keypad, 각종 아날로그 센서 류(가속도 감지 센서, 지자기 센서, 방향 감지 센서, 근접 센서, 자이로스코프 센서, 빛 감지 센서, 압력 감지 센서, 온도 감지 센서 등)에 관하여 소개하고자 한다. 이들은 모두 리눅스 Input 서브시스템(프레임워크)을 사용하며, I²C 혹은 SPI 등의 버스 인터페이스를 통해 제어되는 형태를 띤다.

Input 서브시스템

아래 그림은 리눅스 Input 서브시스템에 관한 전체 구조를 표현한 것으로, Input 시스템에 해당하는 장치로는 PS/2, USB 혹은 Bluetooth 키보드/마우스, SPI Touch controller, 가속도 센서 (Accelerometer) 등이 존재한다. 각각의 Input 장치는 자신을 Input Device로 등록한 상태에서 사용자로부터의 입력이 있을 경우, 이를 Input Event Driver를 통해 userspace의 application으로 전달해 주게 된다.

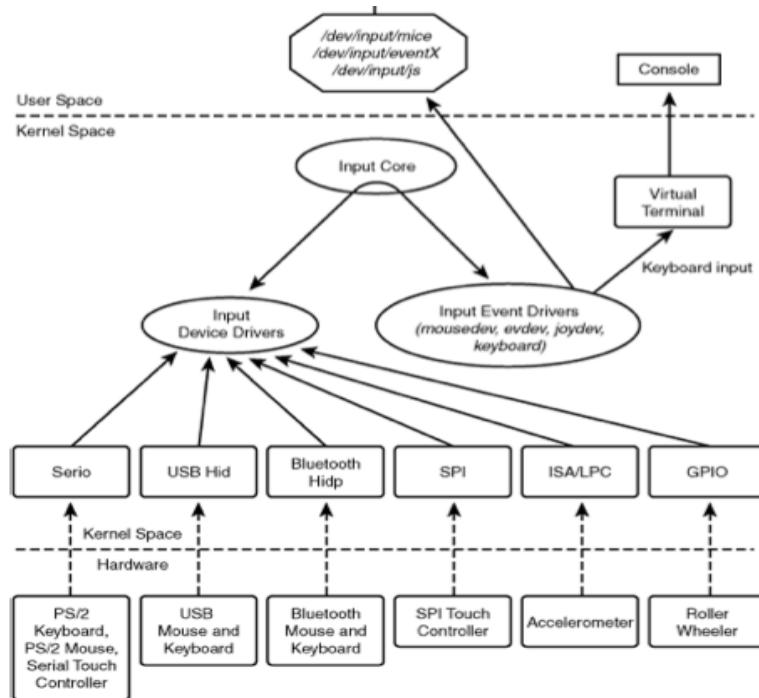


그림 6-13 리눅스 Input 서브시스템(1) [출처 - 참고 문헌]

그림 6-14는 입력 장치가 Touch controller인 경우의 데이터 전달 과정을 data structure 내용과 함께 표현한 것이다.

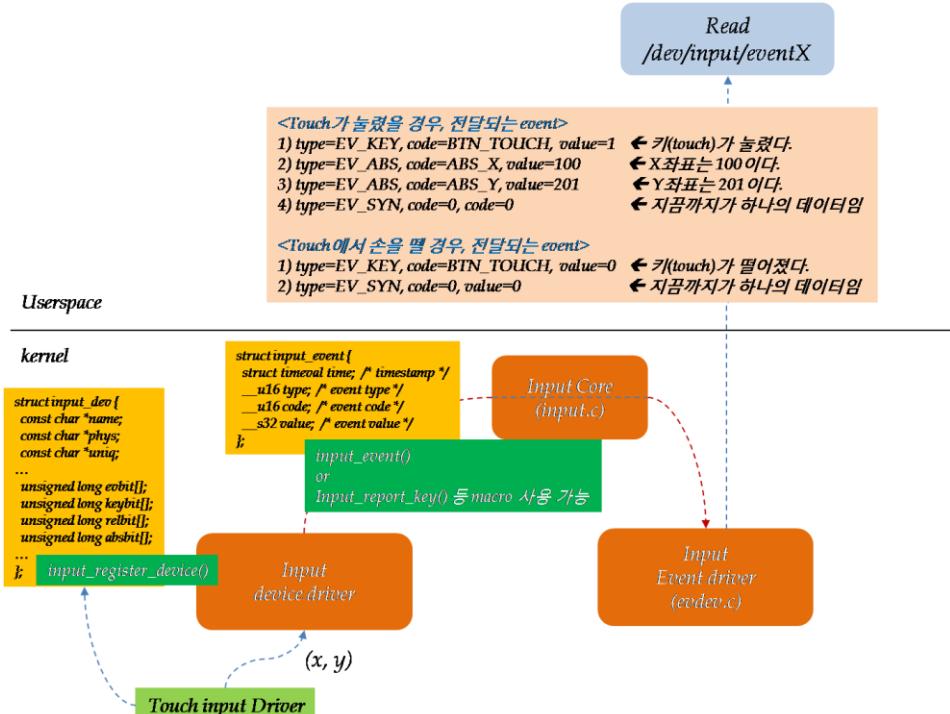


그림 6-14 리눅스 Input 서브시스템(1)

Input 서브시스템 주요 API

Input 서브시스템에서 사용하는 주요 API를 정리해 보면 다음과 같다.

<Touch가 눌린 경우 event 전달 함수 호출 예>

- 1) `input_report_key(&my_touch_drv, BTN_TOUCH, 1);`
- 2) `input_report_abs(&my_touch_drv, ABS_X, 100);`
- 3) `input_report_abs(&my_touch_drv, ABS_Y, 201);`
- 4) `input_sync(&my_touch_drv);`

<Touch에서 손을 뗄 경우 event 전달 함수 호출 예>

- 1) `input_report_key(&my_touch_drv, BTN_TOUCH, 0);`
- 2) `input_sync(&my_touch_drv);`

<TODO> 이 밖에도 더 많은 API가 있음. 이를 정리해야 함.

아래 그림 6-15의 `input_polled_dev`는 user space에서 주기적으로 값을 읽어가는 것과는 달리, work queue를 이용하여 지정된 시간(`polled_interval`)마다 값을 읽어 user space로 던져주는 방식이다.

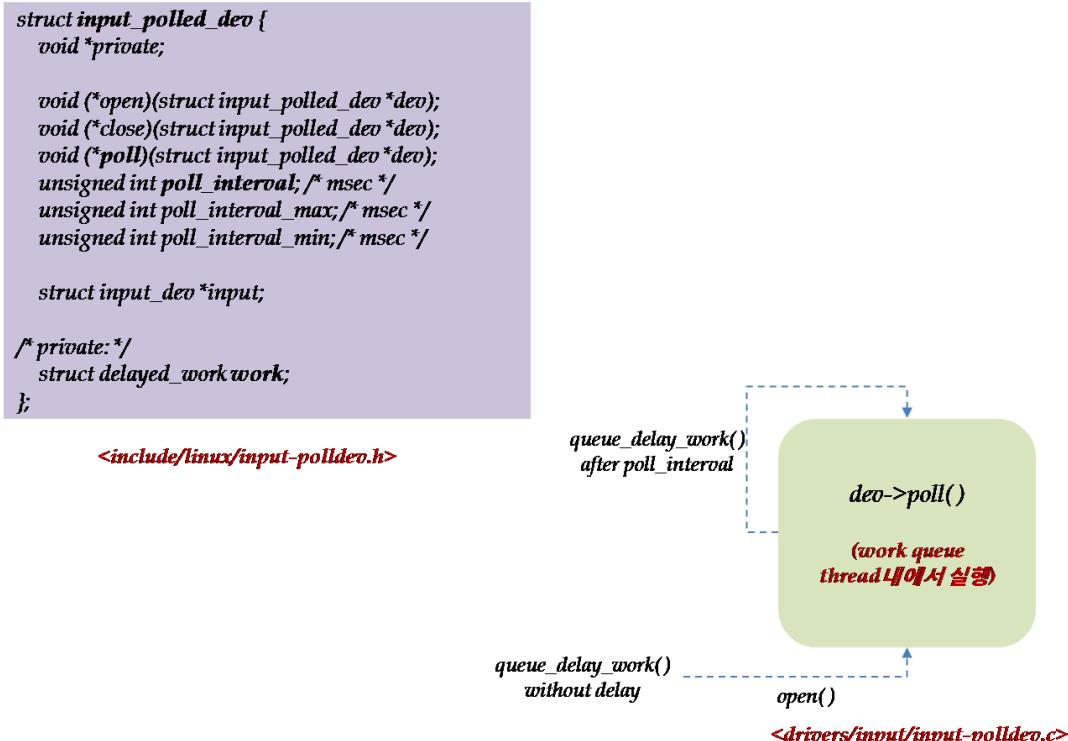


그림 6-15 Input Polled Device

3.1 터치 스크린 드라이버

터치스크린 장치 소개

아래 그림 6-16은 Touchscreen 장치로부터 사용자 입력이 발생할 경우, 이를 Input 서브시스템을 통해 userspace로 전달하는 과정으로 개략적으로 표현한 것이다.

<TODO> - Touchscreen의 hardware 적인 특성 기술

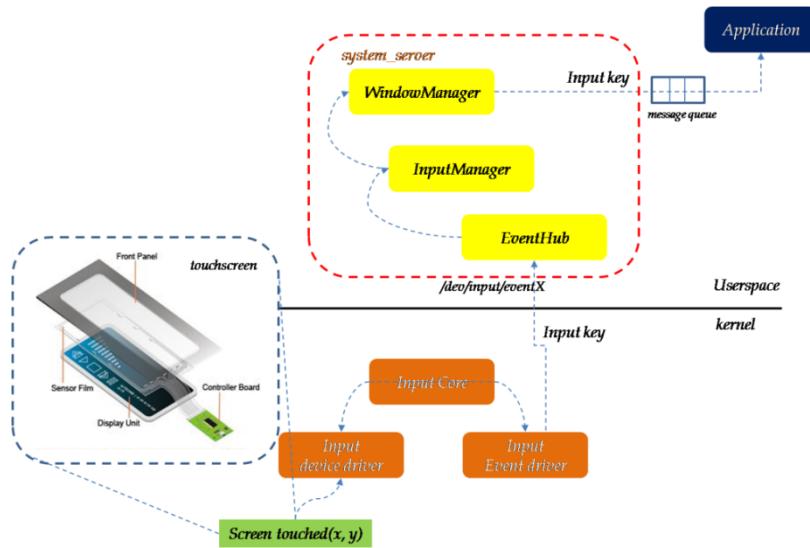


그림 6-16 Touchscreen 장치

SPI 서브시스템

SPI 서브시스템은 I²C 서브시스템과 유사하게 그림 6-17과 같은 형태로 구성되어 있다. 먼저 Master Controller Driver는 SPI Host Controller 장치에 대한 드라이버를 나타내며, SPI Client Driver는 SPI slave device에 대한 드라이버를 말한다. I²C와는 달리, SPI에서는 아직까지는 userspace를 위한 인터페이스를 제공하지는 않고 있다(즉, userspace에서 SPI를 제어할 수 있는 방법이 없다).

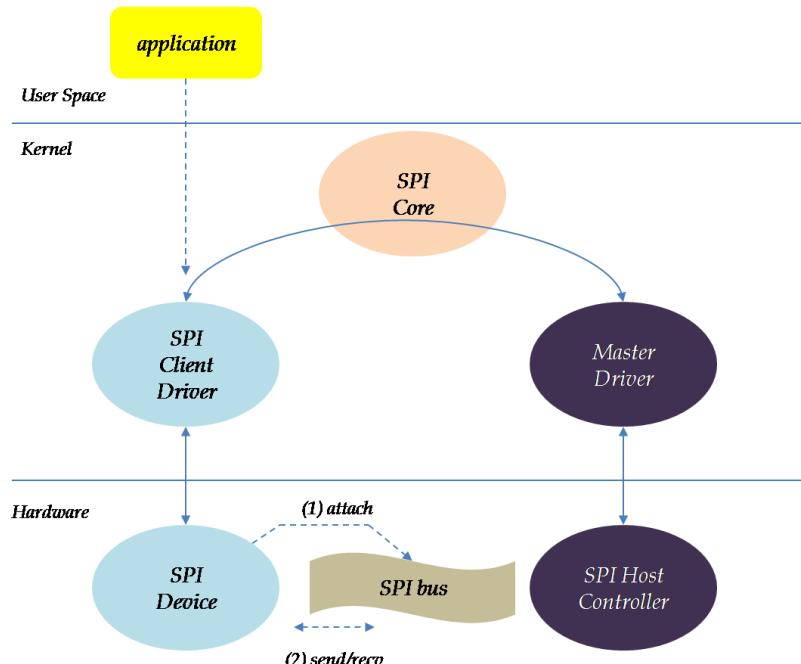


그림 6-17 SPI 서브시스템

SPI 서브시스템에서 정의하는 두 가지 드라이버를 소개하면 다음과 같다.

Controller drivers

- SPI 컨트롤러는 SoC 내에 포함되어 있으며, Master 혹은 Slave로서의 역할을 수행한다. SPI 컨트롤러는 하드웨어 레지스터를 제어하고, DMA를 사용할 수도 있다.

Protocol drivers

- SPI 연결의 반대쪽에 있는 Slave 혹은 Master와 통신하기 위하여 SPI 컨트롤러 드라이버를 통해 메시지를 전송하는 드라이버를 Protocol Driver라고 한다.

그림 6-18은 SPI Core를 경유하여, SPI slave 디바이스 드라이버로부터 SPI controller 드라이버로 데이터를 전송하는 과정을 보여준다.

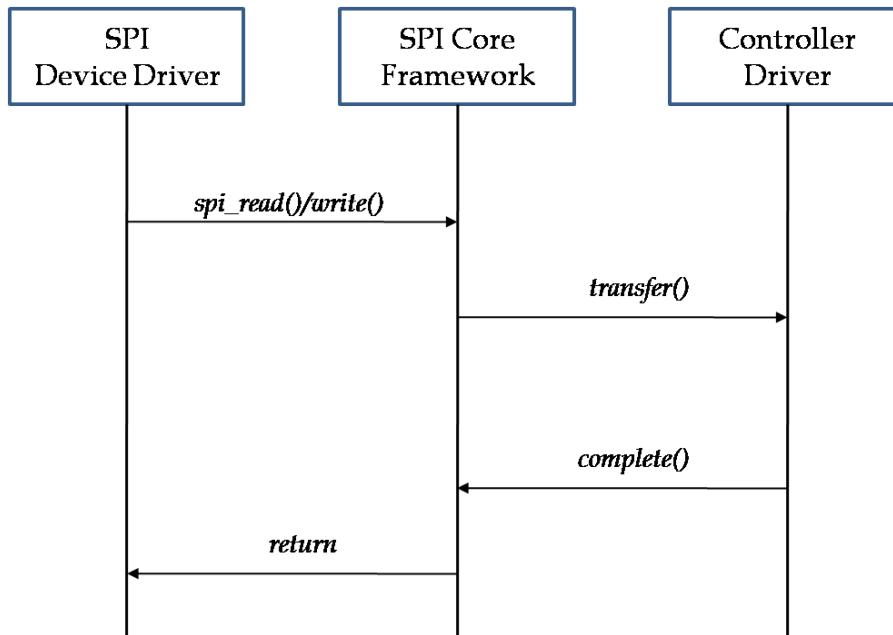


그림 6-18 SPI 데이터 전송

터치스크린 드라이버 소개

Touchscreen 디바이스 드라이버에서 실제로 수행하는 내용을 정리하면 다음과 같다.

<Touch Input Flow>

- 0) touchscreen driver의 경우 전송되는 data의 양이 많지 않으므로 i2c driver 형태로 구현함(이건 장치마다 서로 상이함. SPI로 구현하기도 함).
- 1) 사용자의 touch 입력에 대해 interrupt가 들어온다.
- 2) Interrupt handler의 내부는 지역 처리가 가능한 work queue 형태로 구현하였으며, 따라서 interrupt가 들어올 경우 work queue routine이 동작하도록 schedule해준다.
- 3) Work queue routine은 `i2c_read()` 함수를 이용하여 i2c로 입력된 정보(touch 좌표 정보)를 읽어 들인다.
- 4) 읽어 들인 i2c data를 input subsystem에서 인식할 수 있는 정보로 변형하여 input subsystem을 전달한다.
- 5) Application은 `/dev/input/eventX` 장치 파일로부터 touch 정보를 읽어 들인다.

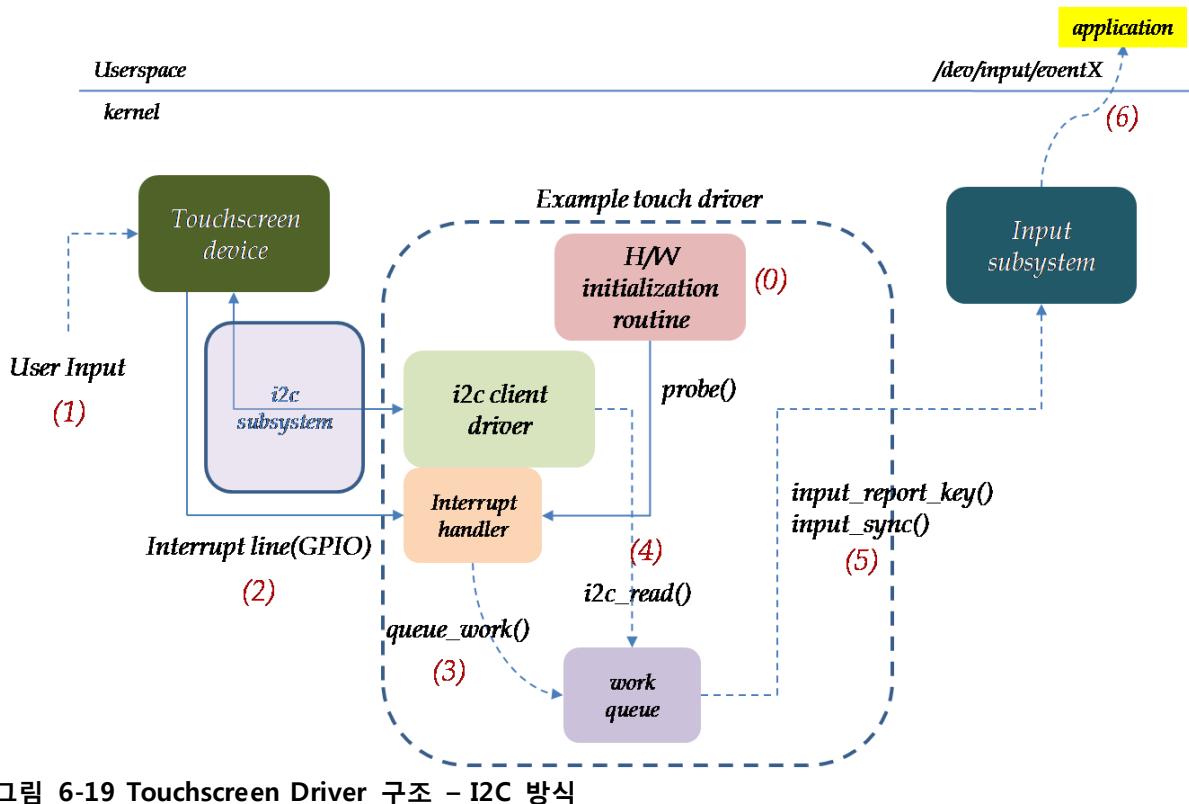


그림 6-19 Touchscreen Driver 구조 – I2C 방식

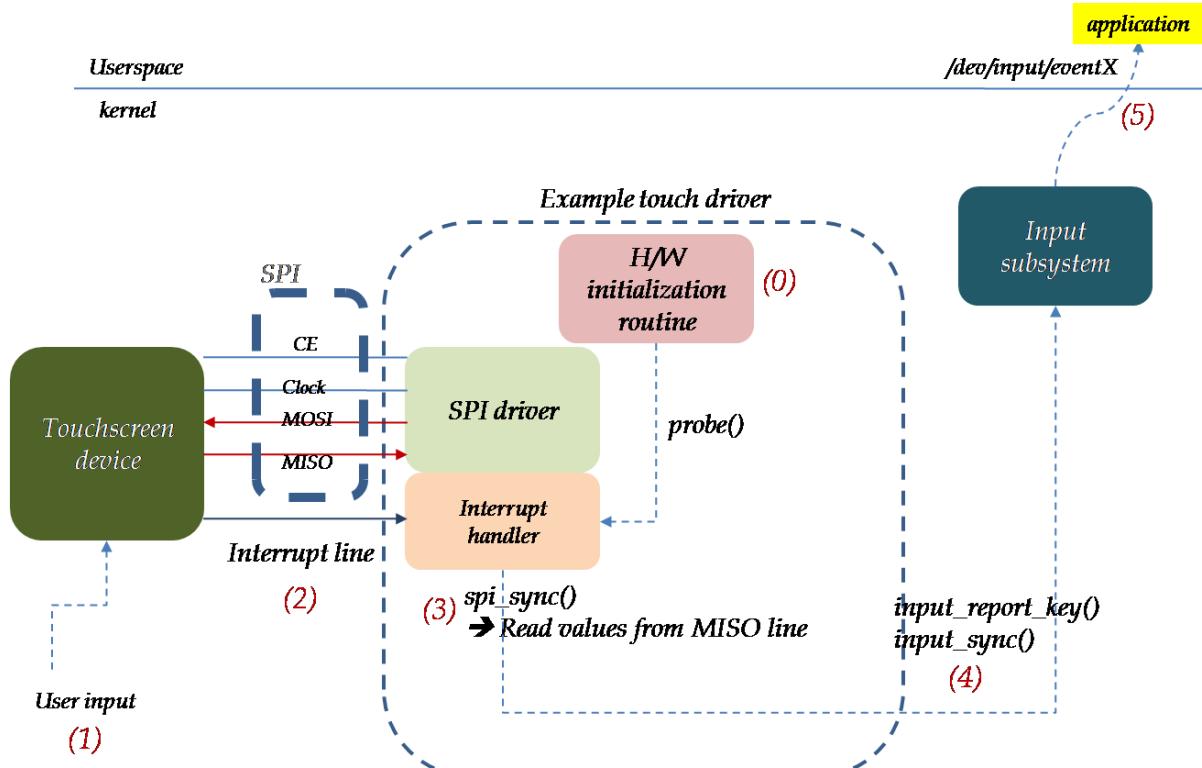


그림 6-20 Touchscreen Driver 구조 – SPI 방식

Touchscreen 드라이버 예제

코드 6-9

<TODO>

3.2 아날로그 센서 드라이버

아날로그 센서 하드웨어

SmartPhone에서 자주 사용되는 아날로그 센서 장치를 열거하면 다음과 같다.

표 6-1 아날로그 센서의 종류

센서 명	기능(역할)
<i>Accelerometer sensor</i>	가속도 감지(흔들림 감지) 센서
<i>Geomagnetic sensor</i>	주변의 자기장을 감지하는 센서
<i>Orientation sensor</i>	기기의 방향을 감지하는 센서
<i>Proximity sensor</i>	특정 물체와 근접한 정도를 감지하는 센서
<i>Gyroscope sensor</i>	모션 센서의 정밀한 교정을 위해 쓰이는 자이로스코프 센서
<i>Light sensor</i>	주변의 빛을 감지하는 센서
<i>Pressure sensor</i>	기기에 적용되는 압력을 감지하는 센서
<i>Temperature sensor</i>	기기 근처의 온도를 감지하는 센서

그림 6-21에는 가속도 센서 칩의 예를 제시하고 있다.

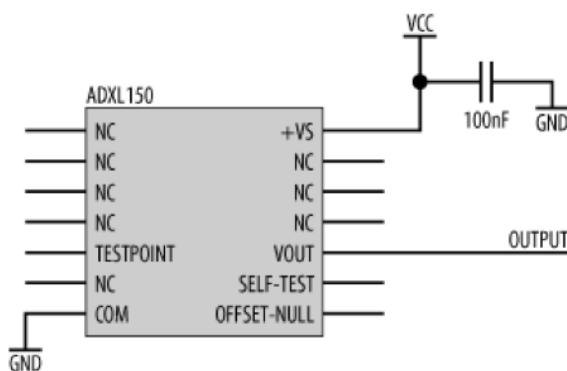


그림 6-21 Accelerometer 센서

I²C 서브시스템(혹은 프레임워크)

리눅스 커널에서는 그림 6-22와 같은 I²C 서브시스템을 구현해 두었다. 이 그림에서 I²C Client 드라이버는 slave 장치에 대한 드라이버로 보아야 하며, I²C Adapter/Algo Driver는 master 장치인

I²C Host Controller로 이해하면 될 것이다. 대개의 경우는 I²C Client 드라이버 즉, slave 장치를 추가하는 경우가 많겠으나, master 장치를 추가하는 경우도 충분히 있을 수 있다.

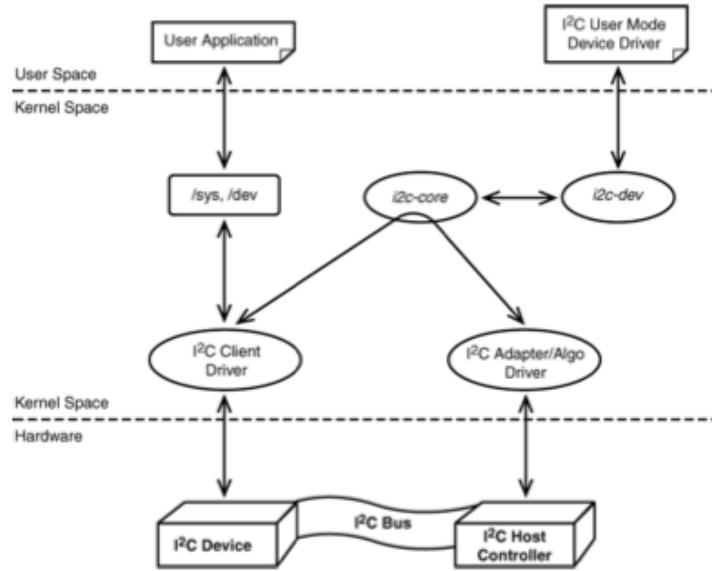


그림 6-22 리눅스 I²C 서브시스템 [출처 - 참고 문헌]

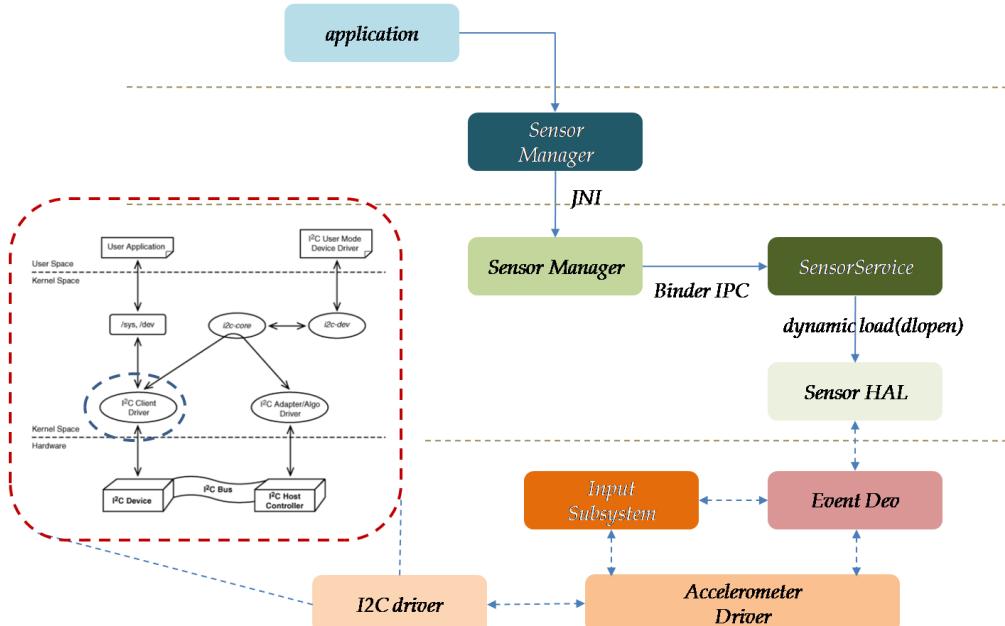


그림 6-23 안드로이드 센서 아키텍처

I²C device 및 Controller 추가 예제

여기서부터는 그림 6-22에서 말하는 I²C Client Driver와 I²C Adapter/Algo Driver를 각각 하나씩 추가하는 방법에 관하여 소개해 보고자 한다. 새로 추가하는 slave 장치와 Host Controller는 각각

다음과 같다.

1) I²C Device

- LIS3LV02DL accelerometer sensor 장치
- 유사한 내용을 drivers/misc/lis3lv0sd/lis3lv02d_i2c.c에서 확인 가능함

2) I²C Host Controller

- STn8815
- 유사한 내용을 drivers/i2c/busses/i2c-nomadik.c에서 확인 가능함

1) I²C device 추가하기

1.1) I²C Client 드라이버 등록하기

struct i2c_driver 객체를 선언하고, 이를 i2c_add_driver() 함수의 argument로 전달함으로써 I2C client를 간단하게 등록할 수 있다.

코드 6-10 I²C Client 드라이버

```
/* Device and driver names */  
# define DEVICE_NAME "lis3lv02d"  
  
/* I2C client structure */  
static struct i2c_device_id lis3lv02d_idtable [] = {  
    { DEVICE_NAME , 0 },  
    {}  
};  
MODULE_DEVICE_TABLE (i2c , lis3lv02d_idtable );  
  
static struct i2c_driver lis3lv02d_driver = {  
    .driver = {  
        .name = DRIVER_NAME  
    },  
    .probe = lis3lv02d_probe,  
    .remove = __devexit_p ( lis3lv02d_remove ),  
    .id_table = lis3lv02d_idtable,  
};  
  
/* Module init */  
static int __init lis3lv02d_init ( void )  
{  
    return i2c_add_driver (& lis3lv02d_driver );  
}
```

1.2) I²C 디바이스(slave) 등록

장치와 장치를 제어하는 드라이버 코드를 서로 묶어 주는 절차가 필요한데, 사전에 I²C 버스와 버스에 연결된 장치의 개수를 알고 있다면, 미리 이를 기술해 주어야 한다. 이는 보드 파일 안에서 기술해 주게 된다.

nhk8815_platform_init() 함수는 board 초기화 과정에서 호출되는데, 이때 i2c_register_board_info() 함수를 호출해 줌으로써 I²C slave 장치를 등록해 주게 된다. struct i2c_board_info 배열은 여러 가지 slave 장치 정보를 담을 수 있는데, 여기에 포함될 수 있는 내용으로는 device name(이는 1.1에서 기술한 driver의 name과 동일한 값이어야 함), device address(I²C slave 주소), IRQ number 등이 있다. 또한, .platform_data 필드를 사용할 경우, 장치 고유(특유)의 정보를 지정할 수도 있다.

코드 6-11 I²C 디바이스

From board-nhk8815.c

```
/* I2C devices */

static struct i2c_board_info nhk8815_i2c_devices [] = {
    {
        I2C_BOARD_INFO ("lis3lv02d", 0x1D),
        .irq = NOMADIK_GPIO_TO_IRQ (82),
        /* No platform data : use driver defaults */
    },
};

static void __init nhk8815_platform_init ( void )
{
    [...]
    /* Register I2C devices on bus #0 (scl0 , sda0 ) */
    i2c_register_board_info (0, nhk8815_i2c_devices ,
        ARRAY_SIZE ( nhk8815_i2c_devices ));
    [...]
}
```

< LIS3LV02DL i2c device probe 함수 요약 >

1. *lis3lv02d_priv private data structure*를 위한 메모리를 할당한다.
2. *device* 설정 정보를 가져온다.
3. *LIS3LV02DL chip*을 명명한다.
4. 장치 하드웨어의 특성을 설정한다.
5. *sysfs* 노드를 만든다.
6. *IRQ handler*를 등록한다.

7. 장치(*accelerometer*)를 *input* 서브시스템에 추가한다.

2) I²C controller 추가하기

2.1) I²C Bus 드라이버 등록(platform device 형태)

STn8815 I²C bus driver를 초기화 및 probing하는 작업은 앞서 소개한 LIS3LV02DL client driver에서의 절차와 유사하다. 다만, 차이가 있다면, STn8815 I²C bus driver는 platform bus를 사용한다는 점이다(주의: I²C Host Controller가 SoC 내에 포함되어 있는 것이며, I²C slave 장치는 이 host controller와 연결되는 것일 뿐이므로, platform bus를 통해 연결되는 것은 아님). Platform bus를 사용한다는 것은 I²C adapter(STn8815)를 platform_device structure를 사용하여 등록해 주어야 함을 뜻한다. platform_device structure는 bus adapter를 표현해 주며, device name, device resource, adaptor number 같은 정보를 bus driver(platform driver)에게 제공해 주어야 한다. I²C adapter를 platform bus 형태로 등록해 주는 작업은 보드 초기화 파일(board-*.c 혹은 *devices*.c 파일)에서 해주게 되며, 이는 보드에 특화된 내용들이다.

아래 코드에서는 STn8815가 두 개의 I²C adapter를 가지고 있으므로, 두 개의 platform device가 정의되어야 한다. 즉, 하나는 bus 0, 다른 하나는 bus 1. 아래 예에서는 첫 번째 platform device의 경우는 platform data를 정의하여 사용하였으나, 두 번째 platform device의 경우는 사용하지 않고 있다. resource와 num_resources 필드는 메모리 영역(base address, size), 인터럽트 번호 등 장치 관련 리소스를 정의하는데 사용된다.

코드 6-12 I²C 버스 드라이버 – platform device

from i2c-8815nhk.c

```
/* first bus : i2c0 */
static struct platform_device nhk8815_i2c_dev0 = {
    .name = "stn8815_i2c",
    .id = 0,
    .resource = & nhk8815_i2c_resources [0],
    .num_resources = 2,
    .dev = {
        .platform_data = & nhk8815_i2c_dev0_data,
    },
};

/* second bus: i2c1 */
static struct platform_device nhk8815_i2c_dev1 = {
    .name = "stn8815_i2c",
    .id = 1,
    .resource = & nhk8815_i2c_resources [2],
    .num_resources = 2,
```

```

/* No platform data : use driver defaults */
};

static int __init nhk8815_i2c_init ( void )
{
    [...]
    platform_device_register (& nhk8815_i2c_dev0 );
    [...]
    platform_device_register (& nhk8815_i2c_dev1 );
    [...]
}
arch_initcall ( nhk8815_i2c_init );

```

2.2) I²C Bus 드라이버 등록하기(platform driver 형태)

한편, platform bus 드라이버는 struct platform_driver를 사용하여 등록할 수 있다. Platform bus는 driver.name과 platform_device structure에 정의되어 있는 각 장치의 이름을 단순히 비교해 본 후, 일치할 경우, 해당 장치를 driver에 연결시킨다.

코드 6-13 I²C 버스 드라이버 – platform driver

```

from i2c-stn8815.c

#define DRIVER_NAME "stn8815_i2c"

static const struct dev_pm_ops stn8815_i2c_pm_ops = {
    SET_RUNTIME_PM_OPS ( stn8815_i2c_runtime_suspend, stn8815_i2c_runtime_resume,
NULL )
};

static struct platform_driver stn8815_i2c_driver = {
    .probe = stn8815_i2c_probe ,
    .remove = __devexit_p ( stn8815_i2c_remove ),
    .driver = {
        .name = DRIVER_NAME ,
        .owner = THIS_MODULE ,
        .pm = & stn8815_i2c_pm_ops ,
    },
};

module_platform_driver ( stn8815_i2c_driver );

```

< I²C Bus Driver Probe 과정 >

1. device resource 정의 내용을 가져온다.
2. 적당한 메모리를 할당하고, 이를 커널에서 접근 가능한 가상 주소로 다시 맵핑한다.
3. 장치 설정 정보를 가져온다.
4. 장치 하드웨어의 특성을 설정한다.
5. 파워 관리 부분을 등록해 준다.
6. sysfs 노드를 생성한다.
7. IRQ handler를 등록해 준다.
8. struct i2c_adapter를 설정하고, I²C core에 이를 추가(등록)해 준다.

2.3) Data Transfer 관련

리눅스 I²C 서브시스템에서는 bus driver가 adapter driver와 algorithm driver로 구성되어 있다. 이는 소프트웨어의 재사용성과 포팅을 용이하게 하기 위한 목적에 기인하고 있다. 실제로 algorithm driver는 i2c adapter가 사용하는 일반적인 코드를 담고 있으며, 반면에 adapter driver는 algorithm driver에 의존하거나, 자신만의 고유의 코드로 이루어져 있다. Bus driver는 probe() 함수에서 struct i2c_adapter를 설정하고, I²C core에 이를 추가(등록)해 주게 된다. i2c_adapter structure는 algo 필드가 struct i2c_algorithm를 가리키도록 설정해 주게 되는데, struct i2c_algorithm에는 아래의 두 pointer가 들어 있다.

master_xfer – 실제 I²C 데이터 송신 및 수신 알고리즘을 구현하는 함수를 가리킨다.

functionality – I²C adapter가 지원하는 속성(feature)을 리턴하는 함수를 가리킨다.

I²C client와 데이터를 주고 받기 위해서, I²C 서브시스템은 원래 아래 두 종류의 함수를 제공하고 있으나, 아래 예제에서는 master_xfer가 가리키는 함수(stn8815_i2c_xfer)가 이를 대신하여 호출될 것이다.

- 1) i2c_master_send(), i2c_master_recv(), i2c_transfer()
- 2) SMBus commands

코드 6-14 I²C Data Transfer

```
from i2cstn8815.c

/* I2C algorithm structure */

static struct i2c_algorithm stn8815_i2c_algo = {
    .master_xfer = stn8815_i2c_xfer ,
    .functionality = stn8815_i2c_func ,
};

/* Probe function */
static int __devinit stn8815_i2c_probe( struct platform_device * pdev )
```

```

{
    [...]
    adap = kzalloc ( sizeof ( struct i2c_adapter ), GFP_KERNEL );
    [...]
    adap -> algo = & stn8815_i2c_algo ;
    [...]
    err = i2c_add_numbered_adapter ( adap );
    [...]
}

```

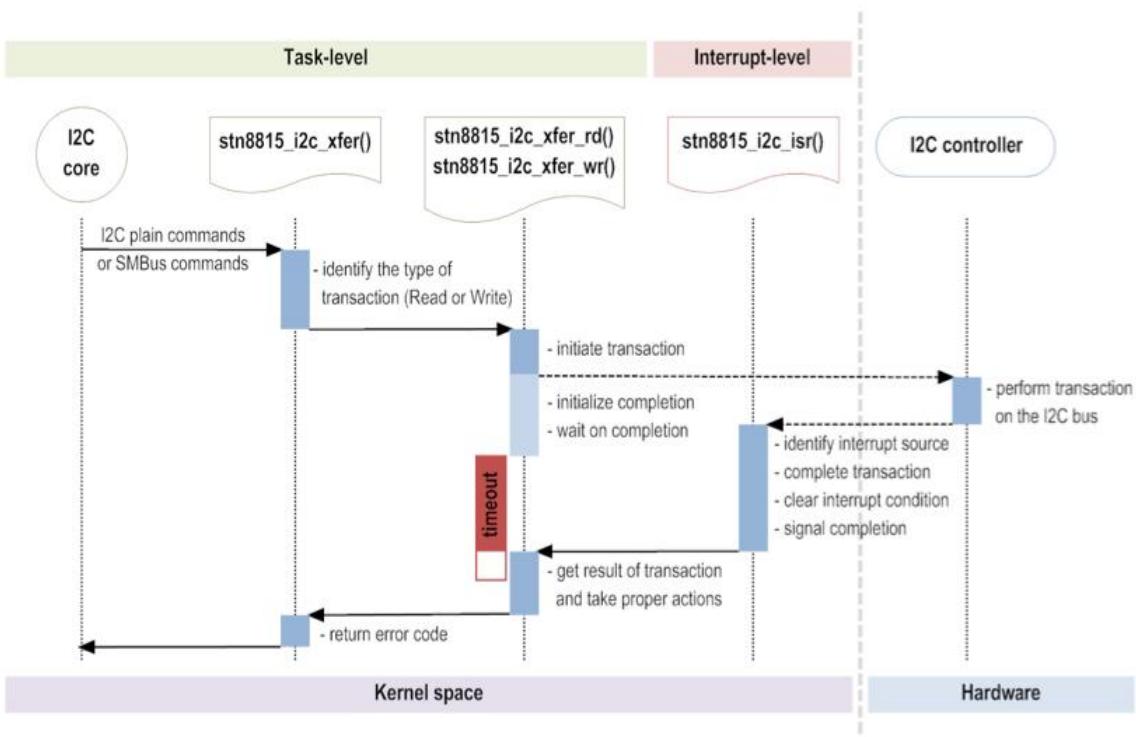


그림 6-24 I²C Transaction 시퀀스 다이어그램

Accelerometer 드라이버 소개

Sensor driver의 경우는 대부분은 i2c client driver 형태로 구현되며, input subsystem을 이용하여 정보를 user space로 전달하는 형태로 구성되어 있다. 아래 그림은 I²C 절에서 소개했던, LI2SLV02DL 센서(accelerometer) 드라이버의 구조를 내부 구조를 표현한 것이다.

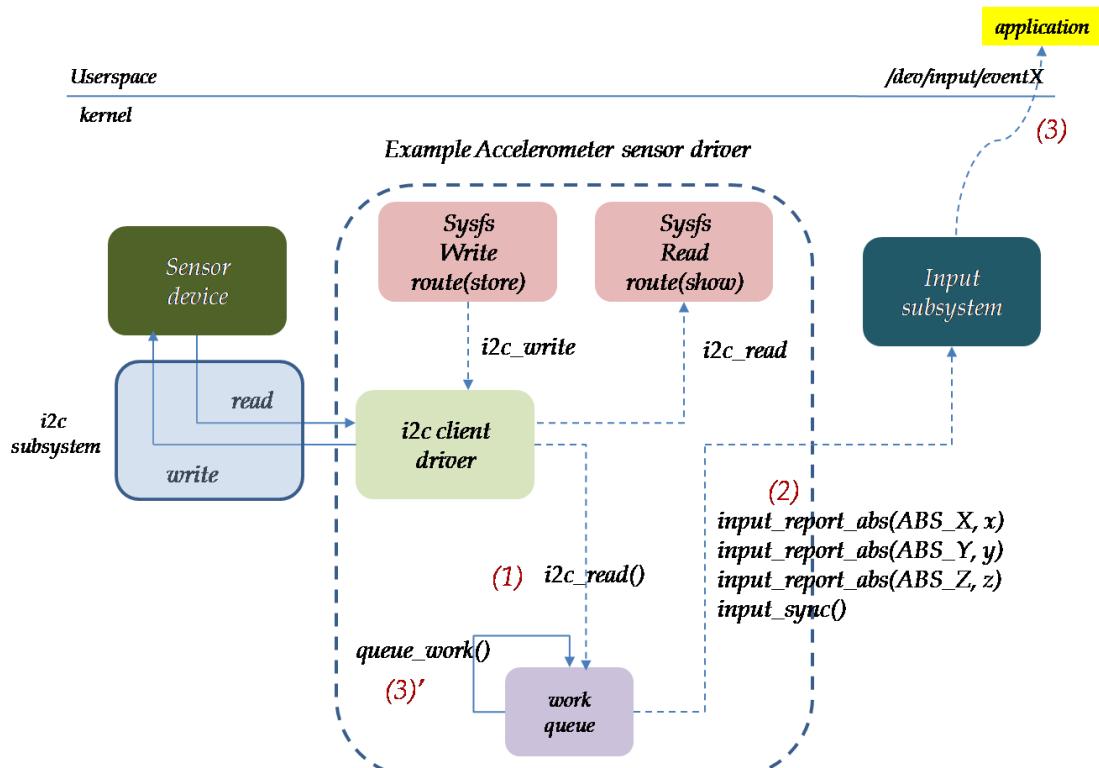


그림 6-25 Accelerometer 센서 드라이버 구조

Accelerometer 센서는 장치 제어를 위해 I²C를 사용하며, 3개의 축(axe) 중 하나에서 가속 증상(programmable acceleration threshold)이 발생할 경우, 이를 알리는 인터럽트가 발생하게 되고, 이 인터럽트 핸들러(여기서는 kernel thread로 구현함) 내에서 input 서브시스템을 통해 가속도 정보를 userspace(/dev/input/eventX)로 전달하게 되는 형태로 되어 있다. 또한, input event 정보를 전달하는 방식은 앞서 소개한 input polled device 방식을 사용하여 구현하였다.

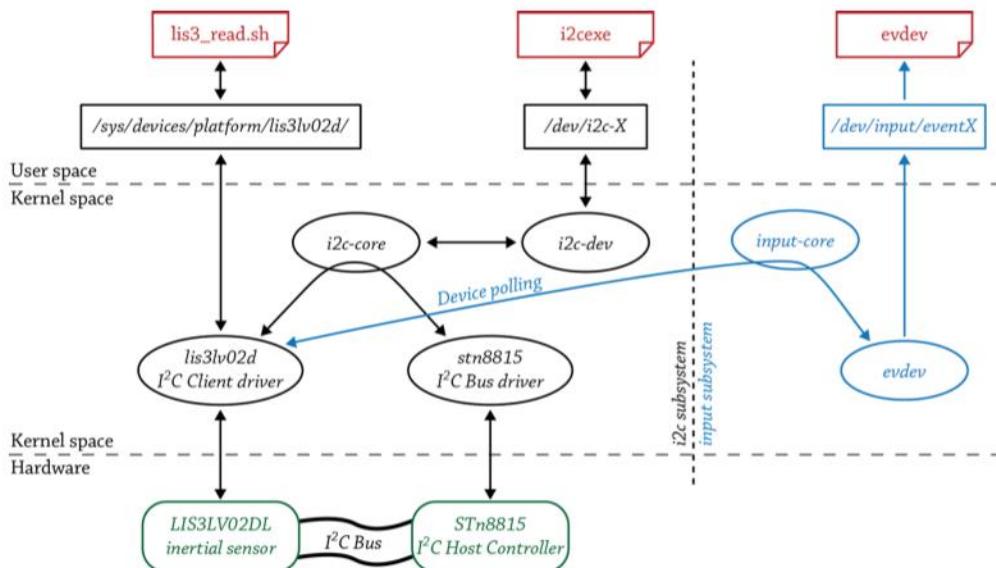


그림 6-26 I²C Device(accelerometer sensor)와 I²C Host Controller 추가 예

코드 6-15

From lis3lv02dnhk8815.c

```
/* Input polled 장치로 등록하는 코드 */
struct input_polled_dev * input_polled ;
struct input_dev * input ;

/* Allocate memory for the input device */
input_polled = input_allocate_polled_device ();

/* Setup input parameters */
input_polled -> open = lis3lv02d_open ;
input_polled -> close = lis3lv02d_close ;
input_polled -> poll = lis3lv02d_poll ;
input = input_polled -> input ;

/* Setup ABS input events */
set_bit ( EV_ABS , input -> evbit );
set_bit ( ABS_X , input -> absbit );
set_bit ( ABS_Y , input -> absbit );
set_bit ( ABS_Z , input -> absbit );

/* Setup KEY event for free - fall ( only if enabled ) */
if (priv -> ff_enabled ) {
    set_bit ( EV_KEY , input -> evbit );
    set_bit ( KEY_FREE_FALL , input -> keybit );
}

/* Register input polled device */
input_register_polled_device ( input_polled );
```

/* 인터럽트 쓰레드 핸들러 등록 – 가속도 관련 인터럽트 발생 시 핸들러 호출됨 */

```
if (pdata -> free_fall_cfg & LIS3_FF_ALL ) {
    [...]
    /* Register IRQ */
    err = request_threaded_irq (client ->irq, NULL, lis3lv02d_isr_thread,
                               IRQF_TRIGGER_RISING | IRQF_ONESHOT, DEVICE_NAME, priv);
    [...]
```

```

priv -> ff_enabled = true;
}

[...]

/* lis3lv02d_isr_thread 인터럽트 쓰레드 내에서 event 전달 */
input_report_key (input , KEY_FREE_FALL , true );
input_report_key (input , KEY_FREE_FALL , false );
input_sync ( input );

```

4. 디스플레이 드라이버

본 장에서는 디스플레이 관련 주요 구성요소인 프레임버퍼(Framebuffer), MIPI DSI 컨트롤러(controller), LCD 패널, LCD 백라이트(backlight) 장치 등을 위주로 설명을 진행하고자 한다. 이 밖에도 HDMI, TV 출력(TVOut), Mixer 등이 있으나, 내용이 너무 광범위한 듯하여 소개에서 제외하였다. 실제로 디스플레이 부분은 매우 복잡한 구조로 되어 있으며, 벤더(vendor) 간에도 구현 방식에 많은 차이가 있어 이를 하나로 통합하려는 움직임(Common Display Framework)이 최근에 진행 중에 있다. 아직 공식적으로 완성되지는 않았으나, 이 부분 관련해서도 일부 소개하도록 하겠다.

4.1 프레임버퍼 드라이버

프레임버퍼 개요

프레임버퍼는 LCD 등의 디스플레이 출력 장치로 화면을 내보내기 위해 메모리 내에 할당되는 일정 영역을 의미한다. 대개의 경우, 커널은 물론이고 응용 프로그램에서도 접근이 가능한 구조로 되어 있으며, 아래 그림에서도 볼 수 있듯이 그래픽 가속 장치와도 밀접한 연관이 있다.

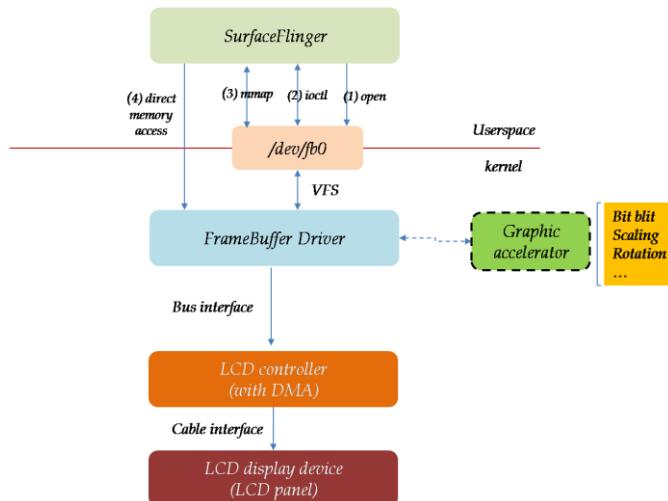


그림 6-27 프레임버퍼 개요

프레임버퍼는 아래 그림 6-28 처럼, fb_info, fb_fix_screeninfo, fb_var_screeninfo, fb_ops 등의 데이터 구조(data structure)를 가지고 있는데, 각각의 내용을 정리해 보기로 하겠다.

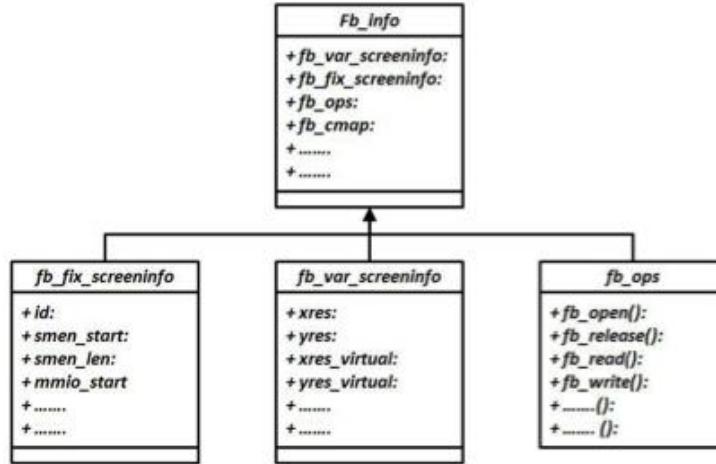


그림 6-28 프레임버퍼 데이터 구조 [다시 그려야 함]

코드 6-16 struct fb_info

fb_info – framebuffer 정보 저장

```

struct fb_info {
    int node;
    int flags;
    struct fb_var_screeninfo var; /* Current var */
    struct fb_fix_screeninfo fix; /* Current fix */
    struct fb_monspecs monspecs; /* Current Monitor specs */
    struct work_struct queue; /* Framebuffer event queue */
    struct fb_pixmap pixmap; /* Image hardware mapper */
    struct fb_pixmap sprite; /* Cursor hardware mapper */
    struct fb_cmap cmap; /* Current cmap */
    struct list_head modelist; /* mode list */
    struct fb_videomode *mode; /* current mode */
    struct backlight_device *bl_dev; /* assigned backlight device */
    struct mutex bl_curve_mutex; /* Backlight level curve */
}
  
```

코드 6-17 struct fb_var_screeninfo

fb_var_screeninfo - 사용자 정의 video card나 LCD controller의 특징을 기술하는 용도로 사용됨.

```
struct fb_var_screeninfo {
```

```

__u32 xres;           /* visible resolution */
__u32 yres;
__u32 xres_virtual;  /* virtual resolution */
__u32 yres_virtual;
__u32 xoffset;        /* offset from virtual to visible */
__u32 yoffset;        /* resolution */
__u32 bits_per_pixel; /* guess what */
__u32 grayscale;      /* != 0 Graylevels instead of colors */
struct fb_bitfield red; /* bitfield in fb mem if true color, */
struct fb_bitfield green; /* else only length is significant */
struct fb_bitfield blue;
struct fb_bitfield transp; /* transparency */
[...]
__u32 pixclock;       /* pixel clock in ps (pico seconds) */
__u32 left_margin;    /* time from sync to picture */
__u32 right_margin;   /* time from picture to sync */
__u32 upper_margin;   /* time from sync to picture */
__u32 lower_margin;
__u32 hsync_len;       /* length of horizontal sync */
__u32 vsync_len;       /* length of vertical sync */
__u32 sync;            /* see FB_SYNC_* */
__u32 vmode;           /* see FB_VMODE_* */
__u32 rotate;          /* angle we rotate counter clockwise */
__u32 reserved[5];     /* Reserved for future compatibility */
}

```

코드 6-18 struct fb_fix_screeninfo

[fb_fix_screeninfo](#) - 프레임버퍼의 시작 주소, 크기 등과 같이 video 장치 관련 고정된 정보를 저장

```

struct fb_fix_screeninfo {
    char id[16];           /* identification string eg "TT Builtin" */
    unsigned long smem_start; /* Start of frame buffer mem */
                           /* (physical address) */
    __u32 smem_len;        /* Length of frame buffer mem */
    __u32 type;             /* see FB_TYPE_* */
    __u32 type_aux;         /* Interleave for interleaved Planes */
    __u32 visual;            /* see FB_VISUAL_* */
    __u16 xpanstep;         /* zero if no hardware panning */
    __u16 ypanstep;         /* zero if no hardware panning */
}

```

```

__u16 ywrapstep;           /* zero if no hardware ywrap */
__u32 line_length;         /* length of a line in bytes */
unsigned long mmio_start;  /* Start of Memory Mapped I/O */
                           /* (physical address) */
__u32 mmio_len;           /* Length of Memory Mapped I/O */
__u32 accel;              /* Indicate to driver which */
                           /* specific chip/card we have */
__u16 reserved[3];         /* Reserved for future compatibility */

};

```

코드 6-19 struct fb_ops – include/linux/fb.h

fb_ops – 프레임버퍼 관련 operations callback 함수 정의

```

struct fb_ops {
    /* open/release and usage marking */
    struct module *owner;
    int (*fb_open)(struct fb_info *info, int user);
    int (*fb_release)(struct fb_info *info, int user);

    /* For framebuffers with strange non linear layouts or that do not
     * work with normal memory mapped access
     */
    ssize_t (*fb_read)(struct fb_info *info, char __user *buf,
                      size_t count, loff_t *ppos);
    ssize_t (*fb_write)(struct fb_info *info, const char __user *buf,
                       size_t count, loff_t *ppos);

    /* checks var and eventually tweaks it to something supported,
     * DO NOT MODIFY PAR */
    int (*fb_check_var)(struct fb_var_screeninfo *var, struct fb_info *info);

    /* set the video mode according to info->var */
    int (*fb_set_par)(struct fb_info *info);

    /* set color register */
    int (*fb_setcolreg)(unsigned regno, unsigned red, unsigned green,
                        unsigned blue, unsigned transp, struct fb_info *info);

    /* set color registers in batch */

```

```
int (*fb_setcmap)(struct fb_cmap *cmap, struct fb_info *info);

/* blank display */
int (*fb_blank)(int blank, struct fb_info *info);

/* pan display */
int (*fb_pan_display)(struct fb_var_screeninfo *var, struct fb_info *info);

/* Draws a rectangle */
void (*fb_fillrect) (struct fb_info *info, const struct fb_fillrect *rect);
/* Copy data from area to another */
void (*fb_copyarea) (struct fb_info *info, const struct fb_copyarea *region);
/* Draws a image to the display */
void (*fb_imageblit) (struct fb_info *info, const struct fb_image *image);

/* Draws cursor */
int (*fb_cursor) (struct fb_info *info, struct fb_cursor *cursor);

/* Rotates the display */
void (*fb_rotate)(struct fb_info *info, int angle);

/* wait for blit idle, optional */
int (*fb_sync)(struct fb_info *info);

/* perform fb specific ioctl (optional) */
int (*fb_ioctl)(struct fb_info *info, unsigned int cmd,
                unsigned long arg);

/* Handle 32bit compat ioctl (optional) */
int (*fb_compat_ioctl)(struct fb_info *info, unsigned cmd,
                       unsigned long arg);

/* perform fb specific mmap */
int (*fb_mmap)(struct fb_info *info, struct vm_area_struct *vma);

/* get capability given var */
void (*fb_get_caps)(struct fb_info *info, struct fb_blit_caps *caps,
                    struct fb_var_screeninfo *var);
```

```

/* teardown any resources to do with this framebuffer */
void (*fb_destroy)(struct fb_info *info);

/* called at KDB enter and leave time to prepare the console */
int (*fb_debug_enter)(struct fb_info *info);
int (*fb_debug_leave)(struct fb_info *info);
};

```

프레임버퍼 응용 프로그램과 디바이스 드라이버 예제

1) 프레임버퍼를 사용하는 응용 프로그램 예제

아래 코드는 응용 프로그램에서 mmap() 함수를 사용하여 프레임버퍼를 사용하는 간단한 예제를 보여준다.

코드 6-20 프레임버퍼 사용 응용 프로그램 예제

```

#include <sys/mman.h>
#include <sys/ioctl.h>
#include <linux/fb.h>

int main(void)
{
    int fb;
    unsigned char* fb_mem;

    fb = open("/dev/fb0", O_RDWR);
    //프레임버퍼 장치(/dev/fb0)을 오픈함.

    fb_mem = mmap (NULL, 320*240*2, PROT_READ|PROT_WRITE, MAP_SHARED, fb, 0);
    //mmap() 함수를 사용하여 커널 메모리를 응용 프로그램에서 공유할 수 있도록 함.

    memset(fb_mem, 0, 320*240*2);
    memcpy(fb_mem, (unsigned char *)ScreenBitmap, 320*240*2);
    //fb_mem이 가리키는 곳이 프레임버퍼에 해당함.
    //이 곳에 그림(화면)을 출력하면 화면(LCD)에 출력될 것임.

    close(fb);
}

```

2) 프레임버퍼 드라이버 예제

코드 6-21 arch/arm/mach-omap2/fb.c – platform device

```
static struct omapfb_platform_data omapfb_config;

static struct platform_device omap_fb_device = { //omapfb platform device 선언
    .name      = "omapfb",
    .id        = -1,
    .dev = {
        .dma_mask      = &omap_fb_dma_mask,
        .coherent_dma_mask = DMA_BIT_MASK(32),
        .platform_data = &omapfb_config,
    },
    .num_resources = 0,
};

static int __init omap_init_fb(void)
{
    return platform_device_register(&omap_fb_device); //omapfb platform device 등록
}

omap_arch_initcall(omap_init_fb);
```

코드 6-22 drivers/video/omap2/omapfb/omapfb-main.c – platform driver

```
static int omapfb_probe(struct platform_device *pdev) //probe() 함수 호출
{
    struct omapfb2_device *fbdev = NULL;
    int r = 0;
    int i;
    struct omap_dss_device *def_display;
    struct omap_dss_device *dssdev;

    DBG("omapfb_probe\n");

    if (omapdss_is_initialized() == false)
        return -EPROBE_DEFER;

    if (pdev->num_resources != 0) {
        dev_err(&pdev->dev, "probed for an unknown device\n");
        r = -ENODEV;
```

```
        goto err0;
    }

fbdev = devm_kzalloc(&pdev->dev, sizeof(struct omapfb2_device),
                     GFP_KERNEL);
if (fbdev == NULL) {
    r = -ENOMEM;
    goto err0;
}

if (def_vrfb && !omap_vrfb_supported()) {
    def_vrfb = 0;
    dev_warn(&pdev->dev, "VRFB is not supported on this hardware, "
             "ignoring the module parameter vrfb=y\n");
}

r = omapdss_compat_init();
if (r)
    goto err0;

mutex_init(&fbdev->mtx);

fbdev->dev = &pdev->dev;
platform_set_drvdata(pdev, fbdev);

fbdev->num_displays = 0;
dssdev = NULL;
for_each_dss_dev(dssdev) {
    struct omapfb_display_data *d;

    omap_dss_get_device(dssdev);

    if (!dssdev->driver) {
        dev_warn(&pdev->dev, "no driver for display: %s\n",
                 dssdev->name);
        omap_dss_put_device(dssdev);
        continue;
    }
    [...]
```

```

    }
    [...]
}

[...]

static struct platform_driver omapfb_driver = { //omapfb platform driver 선언
    .probe      = omapfb_probe,
    .remove     = __exit_p(omapfb_remove),
    .driver     = {
        .name   = "omapfb",
        .owner  = THIS_MODULE,
    },
};

[...]
module_platform_driver(omapfb_driver);

```

4.2 MIPI DSI & LCD 패널

MIPI DSI 개요

MIPI D-PHY DSI(Display Serial Interface)와 CSI(Camera Serial Interface)는 D-PHY를 기반으로 한 디스플레이 및 카메라에 관한 프로토콜 표준 스펙(Specification)이다. D-PHY의 트랜스미터(Transmitter)와 리시버(Receiver)는 각각 8비트 데이터를 SerDes를 통해 주고 받는 구조로 되어 있다. PHY는 1개의 클록과 4개까지의 데이터 레인(lane)으로 구성 될 수 있으며, 레인 별로 각각 데이터를 분배하여 Tx 및 Rx로 신호 처리를 하고 있다. 예를 들어, DSI 트랜스미터의 경우 병렬 데이터, 시그널 이벤트 그리고 커맨드(command)들이 패킷으로 변환 되어지고, 프로토콜 레이어는 여기에 패킷-프로토콜 정보와 헤더 등을 붙여서 PHY로 전달한다. 그리고 PHY는 이를 시리얼화 시켜 전송하며 DSI 리시버에서는 Tx 단과는 반대로 신호처리를 하게 되고 패킷을 해부하여 병렬 데이터, 시그널 이벤트 등과 커맨드 등을 처리하게 된다. 그림 6-29는 MIPI DSI 계층 구조를 표현한 것이며, 그림 6-30은 이 중에서 PHY 계층 부분만을 좀 더 구체적으로 표현해 놓은 것이다.

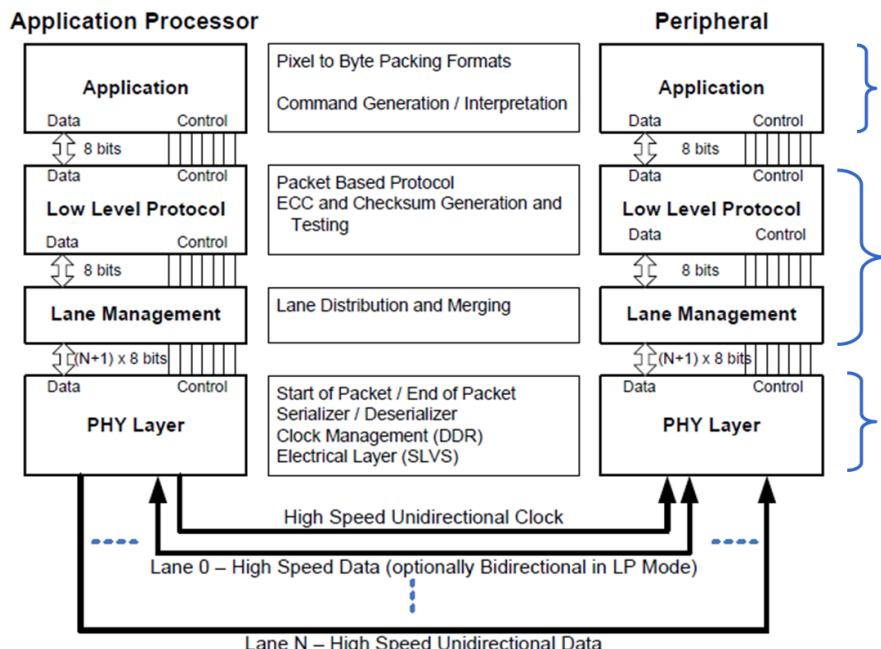


그림 6-29 DSI 계층 구조

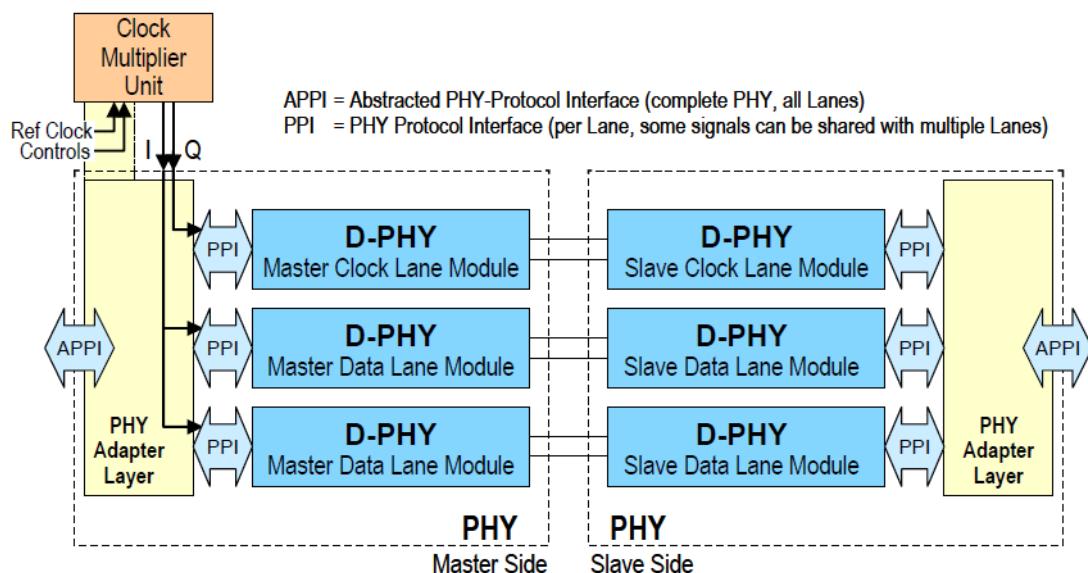


그림 6-30 2 Data 레인 PHY Configuration

[참고 사항] MIPI DSI 관련 보다 자세한 사항은 스펙 문서(참고 문서 18)를 참고하기 바란다.

디스플레이 서브시스템

디스플레이 서브시스템은 벤더(vendor)마다 상당한 차이를 보이고 있으며, 아직 이렇다 할 표준이 정해지지 않고 있어(실제로는 어느 정의 표준이 정해져 있기는 하나 잘 지켜지지 않고 있음), 한마디로 전체 구조를 표현하기가 매우 어려운 것이 사실이다.

먼저 그림 6-31은 디스플레이 서브시스템을 개략적으로 표현한 것으로, 상단의 FBDEV(Framebuffer Device) 및 DRM(Direct Rendering Manager)과 중간의 Display Interface 드라이버(DSI, DBI, DPI, HDMI/DVI로 구성), 그리고 하단의 Panel 드라이버로 구성되어 있음을 알 수 있다.

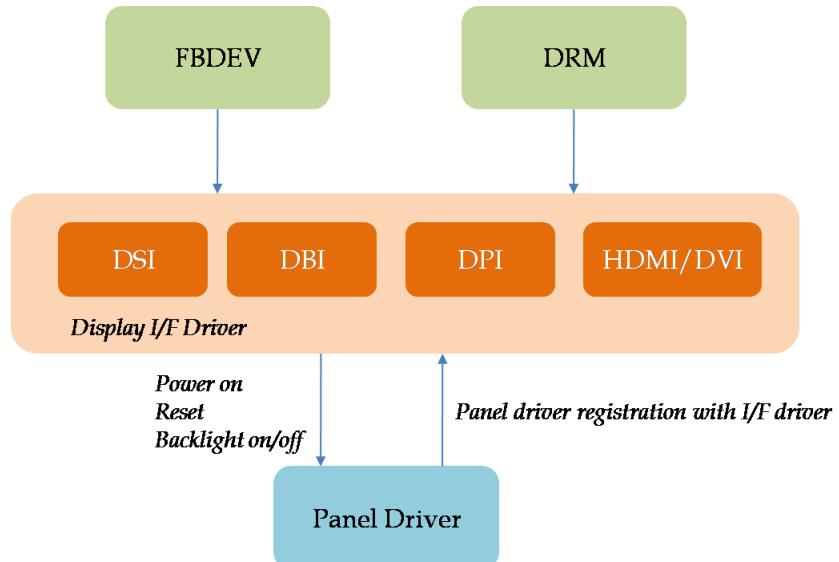


그림 6-31 디스플레이 서브시스템 – 기존 방식

이 상태에서 MIPI DSI 프레임워크를 이용하여 LCD Panel 드라이버를 초기화하는 과정을 살펴 보기로 하자.

<LCD 패널 드라이버 초기화 과정 – Legacy 방식>

- 1) 보드 초기화 코드(`arch/arm/mach-*/board-*.c`)에서 MIPI 프레임워크 내에 `LCD panel device`를 등록한다.
- 2) `LCD panel driver` 초기화 코드에서 자신을 MIPI DSI 프레임워크 드라이버로 등록한다.
- 3) MIPI DSI 프레임워크 드라이버 `probe()` 함수 내에서 1)에서 등록한 LCD 패널 장치를 검색하여, `panel data structure`와 `control ops(operations)`를 추가한다.
- 4) `LCD panel driver`의 `probe()` 함수를 호출한다.
- 5) `LCD panel`의 `power`를 켜고, 3)에서 추가한 `control ops`를 사용하여 LCD 패널 초기화 과정을 진행한다.
- 6) 마침내 LCD 패널이 동작하게 된다.

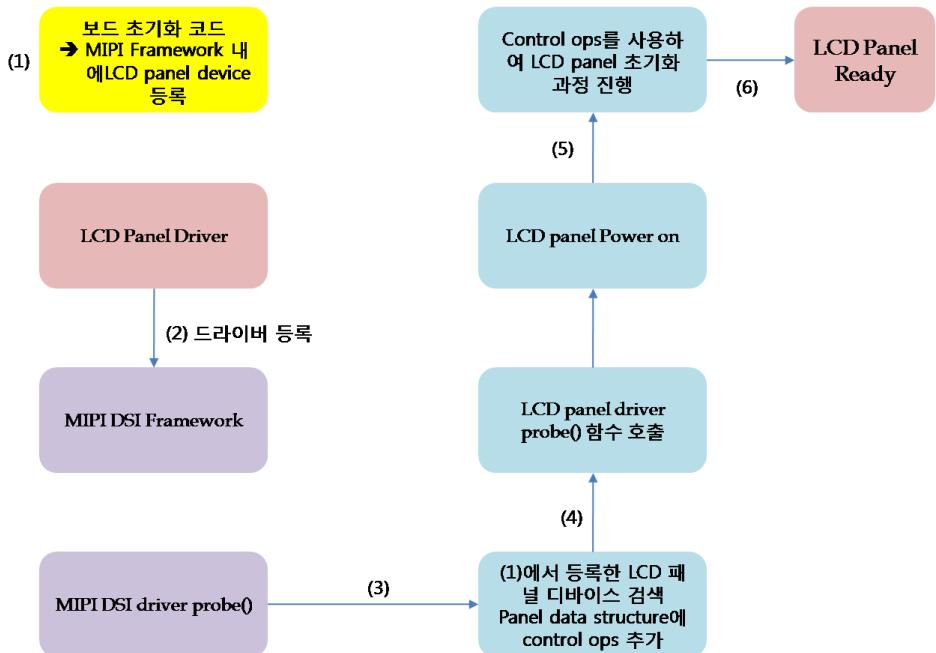


그림 6-32 MIPI DSI 드라이버 모델 – 기존 방식

Laurent Pinchart(참고 문헌 [21] 참조)는 기존의 디스플레이 서브시스템에 문제(아마도 일반화 시키는데 문제가 있다고 판단한 듯 보임)가 있음을 인식하고, Common Display Framework(줄여서 CDF라고 함)을 새롭게 제안하였다. (이 글을 쓰고 있을 당시까지도) 아직 공식 release가 나온 것은 아니지만, 앞으로 이 방식이 디스플레이 서브시스템의 표준으로 자리 잡을 것이 예상되어, 여기에서 간략히 소개하고자 한다.

CDF는 Entity Model에 기초하여, 디스플레이 시스템을 Video Source와 여러 개의 Display Entity가 구성된 것으로 형상화하였다. CDF의 구성 요소를 정리해 보면 다음과 같다.

<CDF의 구성 요소>

1) Entity

- Display controller 이후 단의 디스플레이 장치를 뜻함. 예: LCD panel, TV
- 그림 6-33 참조

2) Link

- Entity 간의 연결
- 그림 6-33 참조

3) Port

- Entity의 input과 output을 의미함(그림 6-33 참조)
- Device Tree로 표현해 주어야 함(그림 6-34 참조)

4) Set/Get operations

- Entity의 port의 속성 값을 얻거나, 변경하는 것을 말함.
- 그림 6-35 참조

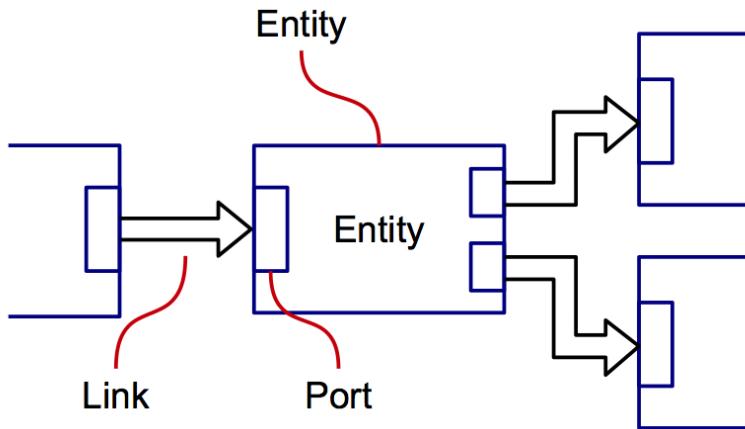


그림 6-33 Entity Model [출처 – 참고 문헌]

```
hdmi_encoder {
    ports {
        #address-cells = <1>;
        #size-cells = <0>

        port@0 {
            hdmi_input: endpoint@0 {
                remote = <&display_output>;
            };
        };
        port@1 {
            endpoint@0 { ... };
            endpoint@1 { ... };
        };
    };
};
```

그림 6-34 Port에 대한 Device Tree 표현 예 [출처 – 참고 문헌]

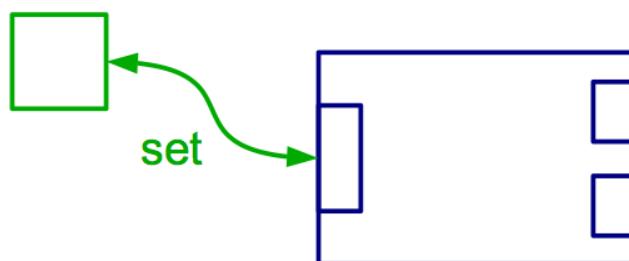


그림 6-35 Get/Set operations [출처 – 참고 문헌]

1개의 display controller와 복수개의 entity가 존재하고, 이를 중간에서 통제하는 pipeline controller가 있을 경우를 그림 6-36과 같이 표현해 보았다. Display controller로부터 전달된 video

데이터는 각각의 entity를 거치게 되며, 각각의 entity에 대한 속성은 pipeline controller를 통해 제어되고 있다.

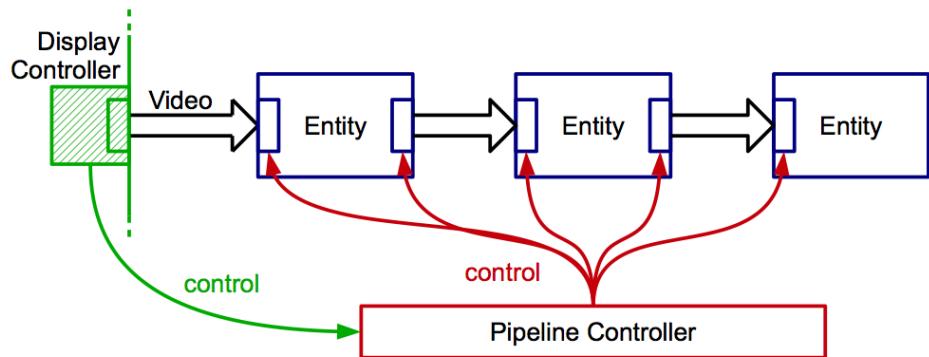


그림 6-36 CDF – Video Source와 Entity 모델 [출처 – 참고 문헌]

이상의 내용을 토대로 새로운 디스플레이 서브시스템을 그림으로 표현해 보면 다음과 같다.

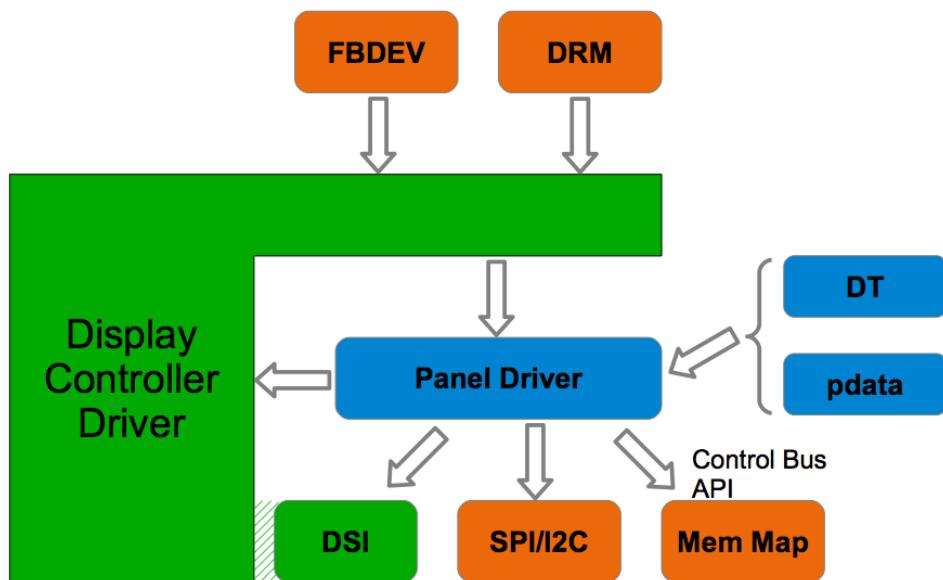


그림 6-37 CDF 전체 구조 [출처 – 참고 문헌]

자 그럼, 마지막으로 CDF가 추가된 상태에서 MIPI DSI 프레임워크를 이용하여 LCD Panel 드라이버를 초기화하는 과정을 통해 앞서 설명했던 내용과 비교해 보기로 하자.

<LCD 패널 드라이버 초기화 과정 – CDF 방식>

1) MIPI DSI Platform 드라이버를 Video Source로 등록한다.

2) LCD Panel Platform 드라이버를 Display Entity로 등록한다.

3) LCD Panel의 power를 켠다.

- 4) CDF를 통해 Video Source를 찾는다(name O/용).
- 5) CDF는 Video Source와 control ops(operations)를 알려 준다.
- 6) Video Source ops를 사용하여 LCD panel 초기화 과정을 진행하면, LCD가 동작하게 된다.

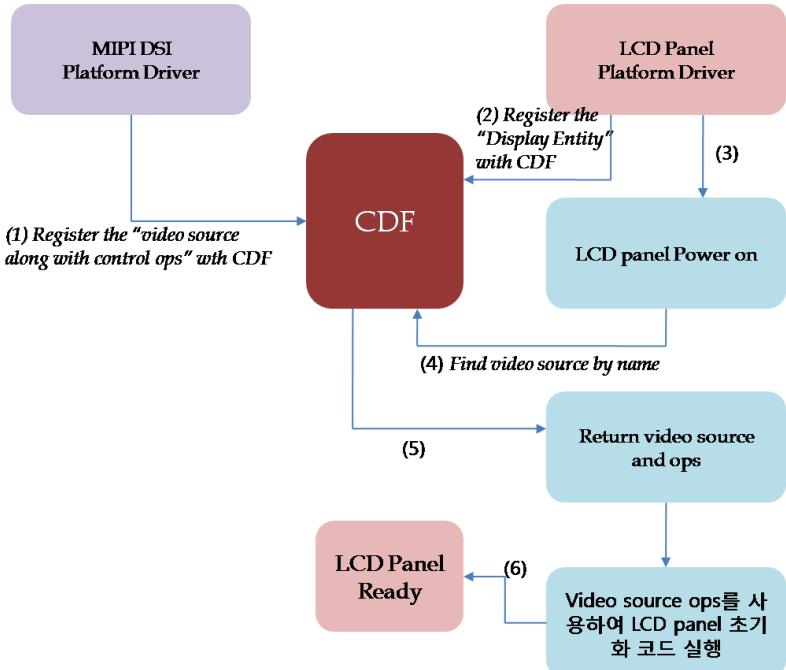


그림 6-38 MIPI DSI 드라이버 모델 – 새로운 방식(Common Display Framework)

MIPI DSI 프레임워크 및 LCD 패널 예제

1) OMAP2 MIP DSI 프레임워크 드라이버 소개

아래 코드는 omapdss_dsi라는 이름의 OMAP2용 MIPI DSI 플랫폼 디바이스와 이를 구동시키는 플랫폼 드라이버 중, 초기화 부분만을 발췌한 것이다.

코드 6-23 arch/arm/mach-omap2/display.c, drivers/video/omap2/dss/dsi.c

From arch/arm/mach-omap2/display.c
 ⇒ omapdss_dsi platform device 등록

```
static const struct omap_dss_hwmod_data omap4_dss_hwmod_data[] __initconst = { ①
  { "dss_core", "omapdss_dss", -1 },
  { "dss_dispc", "omapdss_dispc", -1 },
  { "dss_rfb", "omapdss_rfb", -1 },
```

```

    { "dss_dsi1", "omapdss_dsi", 0 },
    { "dss_dsi2", "omapdss_dsi", 1 },
    { "dss_hdmi", "omapdss_hdmi", -1 },
};

[...]

int __init omap_display_init(struct omap_dss_board_info *board_data) ②
{
    [...]

    /* create devices for dss hwmods */
    if (cpu_is_omap24xx()) {
        curr_dss_hwmod = omap2_dss_hwmod_data;
        oh_count = ARRAY_SIZE(omap2_dss_hwmod_data);
    } else if (cpu_is_omap34xx()) {
        curr_dss_hwmod = omap3_dss_hwmod_data;
        oh_count = ARRAY_SIZE(omap3_dss_hwmod_data);
    } else {
        curr_dss_hwmod = omap4_dss_hwmod_data;
        oh_count = ARRAY_SIZE(omap4_dss_hwmod_data);
    }

    dss_pdev = create_dss_pdev(curr_dss_hwmod[0].dev_name, ③
                               curr_dss_hwmod[0].id,
                               curr_dss_hwmod[0].oh_name,
                               board_data, sizeof(*board_data),
                               NULL);
}

[...]
}

static struct platform_device *create_dss_pdev(const char *pdev_name, ④
                                              int pdev_id, const char *oh_name, void *pdata, int pdata_len,
                                              struct platform_device *parent)
{
    struct platform_device *pdev;
    struct omap_device *od;
    struct omap_hwmod *ohs[1];
    struct omap_hwmod *oh;

```

```

int r;

oh = omap_hwmod_lookup(oh_name);
if (!oh) {
    pr_err("Could not look up %s\n", oh_name);
    r = -ENODEV;
    goto err;
}

pdev = platform_device_alloc(pdev_name, pdev_id);    (5) //platform device 등록 준비
if (!pdev) {
    pr_err("Could not create pdev for %s\n", pdev_name);
    r = -ENOMEM;
    goto err
}

if (parent != NULL)
    pdev->dev.parent = &parent->dev;

if (pdev->id != -1)
    dev_set_name(&pdev->dev, "%s.%d", pdev->name, pdev->id);
else
    dev_set_name(&pdev->dev, "%s", pdev->name);

ohs[0] = oh;
od = omap_device_alloc(pdev, ohs, 1);    (6) //platform device로 등록
[...]
}

```

[From drivers/video/omap2/dss/dsi.c](#)

⇒ [omapdss_dsi](#) platform driver 등록 & 구동

```

/* DSI1 HW IP initialisation */

static int omap_dsihw_probe(struct platform_device *dsidev)    (9) //probe() 함수 호출
{
    u32 rev;
    int r, i;
    struct resource *dsi_mem;

```

```

struct dsi_data *dsi;

dsi = devm_kzalloc(&dsidev->dev, sizeof(*dsi), GFP_KERNEL);
if (!dsi)
    return -ENOMEM;

dsi->module_id = dsidev->id;
dsi->pdev = dsidev;
dev_set_drvdata(&dsidev->dev, dsi);

spin_lock_init(&dsi->irq_lock);
spin_lock_init(&dsi->errors_lock);
dsi->errors = 0;

#ifndef CONFIG OMAP2_DSS_COLLECT_IRQ_STATS
    spin_lock_init(&dsi->irq_stats_lock);
    dsi->irq_stats.last_reset = jiffies;
#endif

mutex_init(&dsi->lock);
sema_init(&dsi->bus_lock, 1);

INIT_DEFERRABLE_WORK(&dsi->framedone_timeout_work,
                     dsi_framedone_timeout_work_callback);

[...]
}

static struct platform_driver omap_dsihw_driver = { ⑦ //platform driver 선언
    .probe      = omap_dsihw_probe,
    .remove     = __exit_p(omap_dsihw_remove),
    .driver     = {
        .name      = "omapdss_dsi",
        .owner     = THIS_MODULE,
        .pm        = &dsi_pm_ops,
    },
};

int __init dsi_init_platform_driver(void)

```

```
{
    return platform_driver_register(&omap_dsihw_driver);    ⑧ //platform driver로 등록
}
```

2) LCD Panel 드라이버 소개

아래 코드는 Taal DSI command mode panel 드라이버 코드 중, omap_dss_driver로 등록하는 과정만을 발췌해본 것이다. 아래 코드에서 볼 수 있듯이, LCD panel 드라이버는 플랫폼 디바이스/드라이버라기 보다는 OMAP DSS 프레임워크 용 드라이버로 이해해야 할 것이다.

코드 6-24 drivers/video/omap2/displays/panel-taal.c

From include/video/omapdss.h

```
struct omap_dss_driver {    ① //omap_dss_driver structure 정의
    struct device_driver driver;

    int (*probe)(struct omap_dss_device *); //아래 코드에서 taal_probe()로 채워지게 됨.
    void (*remove)(struct omap_dss_device *);

    int (*enable)(struct omap_dss_device *display);
    void (*disable)(struct omap_dss_device *display);
    int (*run_test)(struct omap_dss_device *display, int test);

    int (*update)(struct omap_dss_device *dssdev,
                  u16 x, u16 y, u16 w, u16 h);
    int (*sync)(struct omap_dss_device *dssdev);

    int (*enable_te)(struct omap_dss_device *dssdev, bool enable);
    int (*get_te)(struct omap_dss_device *dssdev);

    u8 (*get_rotate)(struct omap_dss_device *dssdev);
    int (*set_rotate)(struct omap_dss_device *dssdev, u8 rotate);

    bool (*get_mirror)(struct omap_dss_device *dssdev);
    int (*set_mirror)(struct omap_dss_device *dssdev, bool enable);

    int (*memory_read)(struct omap_dss_device *dssdev,
                       void *buf, size_t size,
                       u16 x, u16 y, u16 w, u16 h);
```

```

void (*get_resolution)(struct omap_dss_device *dssdev,
                      u16 *xres, u16 *yres);
void (*get_dimensions)(struct omap_dss_device *dssdev,
                      u32 *width, u32 *height);
int (*get_recommended_bpp)(struct omap_dss_device *dssdev);

int (*check_timings)(struct omap_dss_device *dssdev,
                     struct omap_video_timings *timings);
void (*set_timings)(struct omap_dss_device *dssdev,
                    struct omap_video_timings *timings);
void (*get_timings)(struct omap_dss_device *dssdev,
                    struct omap_video_timings *timings);

int (*set_wss)(struct omap_dss_device *dssdev, u32 wss);
u32 (*get_wss)(struct omap_dss_device *dssdev);

int (*read_edid)(struct omap_dss_device *dssdev, u8 *buf, int len);
bool (*detect)(struct omap_dss_device *dssdev);

/*
 * For display drivers that support audio. This encompasses
 * HDMI and DisplayPort at the moment.
 */
/*
 * Note: These functions might sleep. Do not call while
 * holding a spinlock/readlock.
 */
int (*audio_enable)(struct omap_dss_device *dssdev);
void (*audio_disable)(struct omap_dss_device *dssdev);
bool (*audio_supported)(struct omap_dss_device *dssdev);
int (*audio_config)(struct omap_dss_device *dssdev,
                    struct omap_dss_audio *audio);
/* Note: These functions may not sleep */
int (*audio_start)(struct omap_dss_device *dssdev);
void (*audio_stop)(struct omap_dss_device *dssdev);
};


```

[From drivers/video/omap2/displays/panel-taal.c](#)

```

static int taal_probe(struct omap_dss_device *dssdev) ④ //probe() 함수 호출
{
    struct backlight_properties props;
    struct taal_data *td;
    struct backlight_device *bldev = NULL;
    int r;

    dev_dbg(&dssdev->dev, "probe\n");

    td = devm_kzalloc(&dssdev->dev, sizeof(*td), GFP_KERNEL);
    if (!td)
        return -ENOMEM;

    dev_set_drvdata(&dssdev->dev, td);
    td->dssdev = dssdev;

    if (dssdev->data) {
        const struct nokia_dsi_panel_data *pdata = dssdev->data;

        taal_probe_pdata(td, pdata);
    } else {
        return -ENODEV;
    }

    dssdev->panel.timings.x_res = 864;
    dssdev->panel.timings.y_res = 480;
    dssdev->panel.pixel_clock = DIV_ROUND_UP(864 * 480 * 60, 1000);
    dssdev->panel.dsi_pix_fmt = OMAP_DSS_DSI_FMT_RGB888;
    dssdev->caps = OMAP_DSS_DISPLAY_CAP_MANUAL_UPDATE |
        OMAP_DSS_DISPLAY_CAP_TEAR_ELIM;
    [...]
}

static struct omap_dss_driver taal_driver = { ② //omap_dss_driver로 선언
    .probe      = taal_probe,
    .remove     = __exit_p(taal_remove),
    .enable     = taal_enable,
}

```

```

.disable      = taal_disable,
.update      = taal_update,
.sync        = taal_sync,
.get_resolution = taal_get_resolution,
.get_recommended_bpp = omapdss_default_get_recommended_bpp,
.enable_te   = taal_enable_te,
.get_te      = taal_get_te,
.run_test    = taal_run_test,
.memory_read = taal_memory_read,
.driver      = {
    .name      = "taal",
    .owner     = THIS_MODULE,
},
};

static int __init taal_init(void)
{
    omap_dss_register_driver(&taal_driver);    ③
    //LCD panel 드라이버를 omap dss 드라이버로 등록함

    return 0;
}

```

LCD Backlight 프레임워크 소개

아래 그림은 시리얼(혹은 직렬) 및 병렬 방식의 LED 하드웨어 구조를 그림으로 표현한 것이다. 특별히 이 절에서 LED를 언급하는 이유는 LCD 디스플레이 장치로 출력된 내용이 LED의 밝기에 따라서 보일 수도 혹은 안 보일 수도 있게 되기 때문이다.

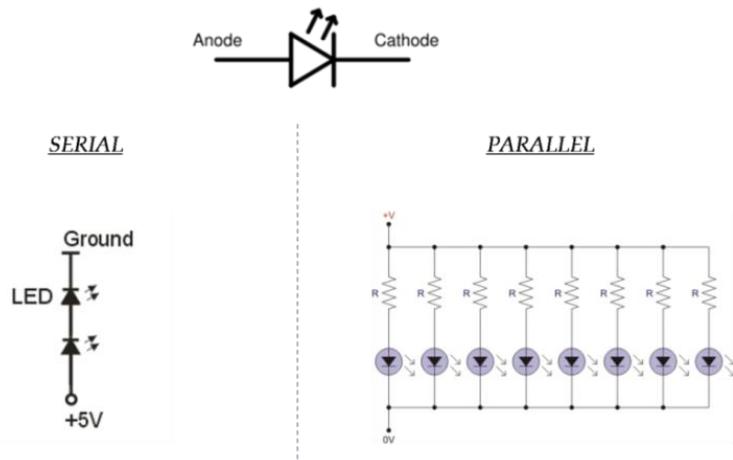


그림 6-39 LED 디바이스

LCD 화면 출력과 깊은 연관이 있는 LCD backlight의 밝기를 조정하는 절차를 간략히 살펴보기로 하자.

<LCD backlight 밝기 조정>

1) 자신만의 *struct backlight_ops*를 선언한다(*backlight get, set* 함수 선언해 줌).

- 최종적으로 LED 제어는 GPIO를 통한다.
- (TODO) GPIO framework 소개

2) *LCD panel* 장치를 *probe()* 함수 내에서 *backlight_device_register()* 함수를 사용하여 *backlight* 장치로 등록한다. 이때 앞서 선언한 *struct backlight_ops* 객체를 파라미터로 전달한다.

3) 사용자 계층에서 LED 밝기 조절을 위한 인터페이스는 */sys/class/leds/lcd-backlight/brightness*이며, 이를 통해 LED 밝기 조정 명령을 내리면, 앞서 2단계에서 등록한 *struct backlight_ops* 객체의 *backlight set* 함수가 호출되어 LED 밝기가 조절된다.

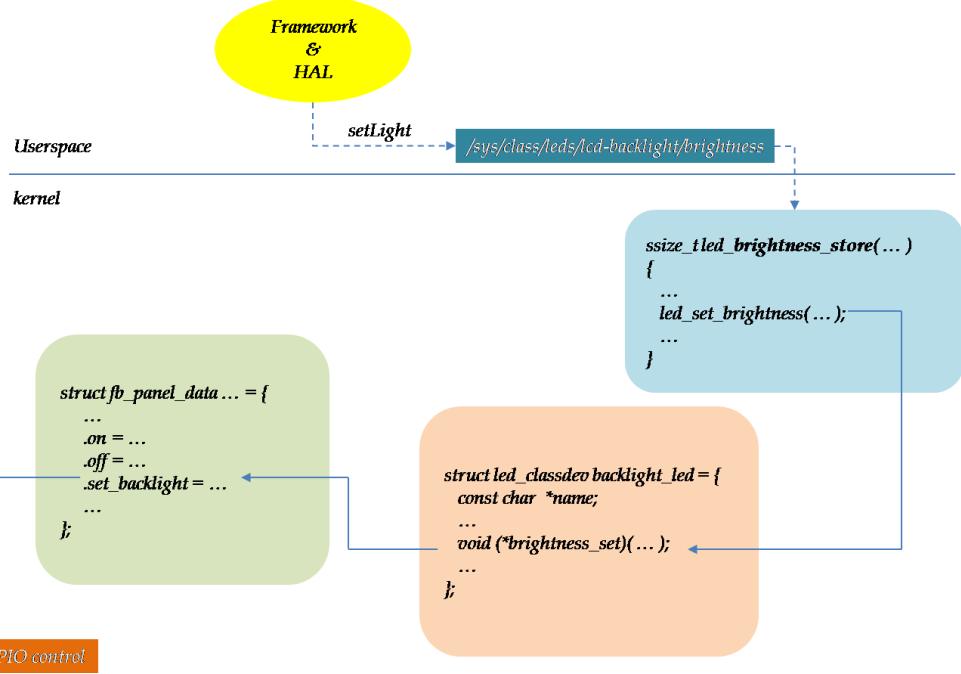


그림 6-40 LCD Backlight 드라이버

5. 카메라 드라이버

이번 절에서는 Camera 센서 연결을 위한 MIPI-CSI2 인터페이스와 Video Capture Application Interface인 V4L2에 대하여 소개하고자 한다.

5.1 MIPI CSI & 카메라 센서 드라이버

현재 5M 픽셀 이상을 지원하는 고화소 이미지 센서는 MIPI 인터페이스를 채용하는 추세이다. CCIR601/656 방식과 같은 기존 병렬 인터페이스로는 전송 속도, 전력 소모, 잡음 및 신호 무결성 등의 문제로 고화소 센서에 대한 대응이 어려워지기 때문에 이러한 문제점을 해결한 고속 직렬 버스 방식인 MIPI 인터페이스 채용이 늘고 있다. 또한 고화소 센서가 MIPI 방식을 채용하면서, 화상 통신을 위해 2개의 카메라를 지원하는 휴대폰에서는 저화소 센서까지도 MIPI 인터페이스를 요구하고 있는 상황이다.

그림 6-41은 카메라 센서와 MIPI CSI-2 인터페이스 관련 개략적인 구조를 표현한 것이다.

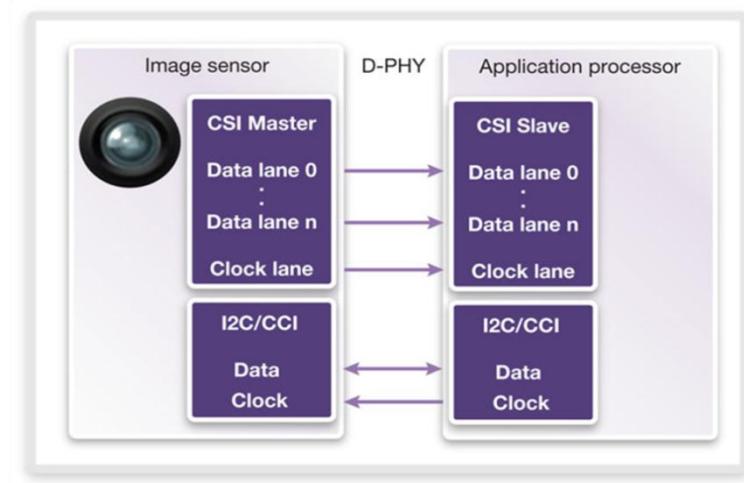


그림 6-41 MIPI-CSI2와 Camera 센서 개요

카메라 센서(혹은 모듈)와 인터페이스(혹은 카메라 Core)

일반적으로 카메라는 외부 센서 모듈과 내부 카메라 코어 인터페이스로 구성되어 있다고 볼 수 있다. 그림 6-42에 카메라 센서와 내부 코어 인터페이스의 상관 관계를 그려 보았는데, 이들의 관계를 좀 더 살펴 보기로 하자.

< Camera 모듈(센서)과 Camera Core 인터페이스 >

1) Camera 센서의 제어는 대개 I²C를 통하여 이루어진다. 카메라 UI를 통해 이루어지는 각종 설정이나 부분과 관련이 있다고 보면 된다.

- Exposure control
- Lens control
- White balance control
- Effect control
- Face detect
- Zoom control
- Strobe control 등

2) Camera 센서로부터 들어오는 raw data는 포맷 전환 과정(YUV)을 거쳐 파일(JPEG)로 저장되거나 서피스플링어(surfaceflinger)를 거쳐 화면으로 출력(RGB)된다. 이 과정은 다시 아래와 같이 3가지 유형으로 세분화할 수 있다.

- Camera Preview 흐름(그림 6-43 참조)
- Still shot 흐름(그림 6-44 참조)
- Motion Picture 흐름(그림 6-45 참조)

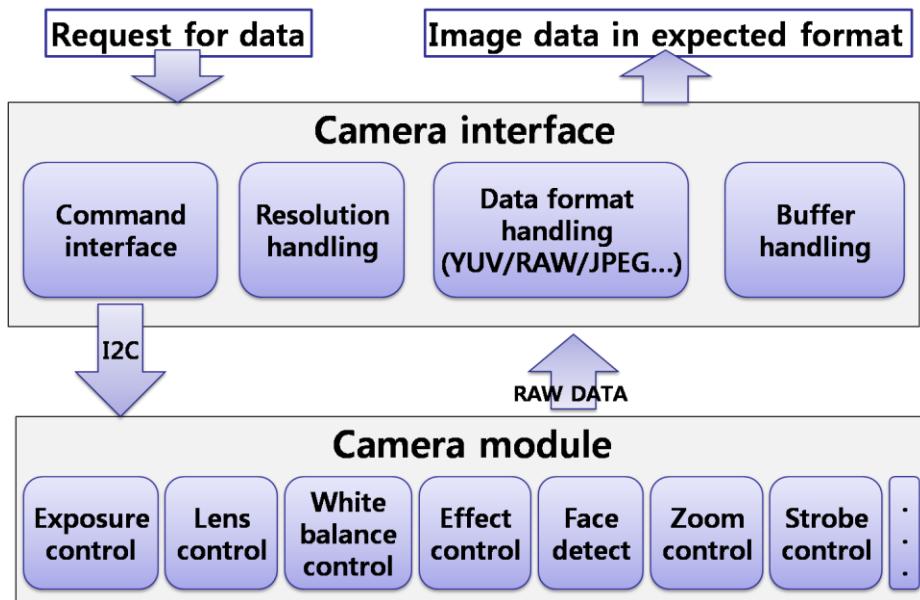


그림 6-42 Camera 모듈(센서)과 Camera Core 인터페이스 개요 [다시 그려야 함]

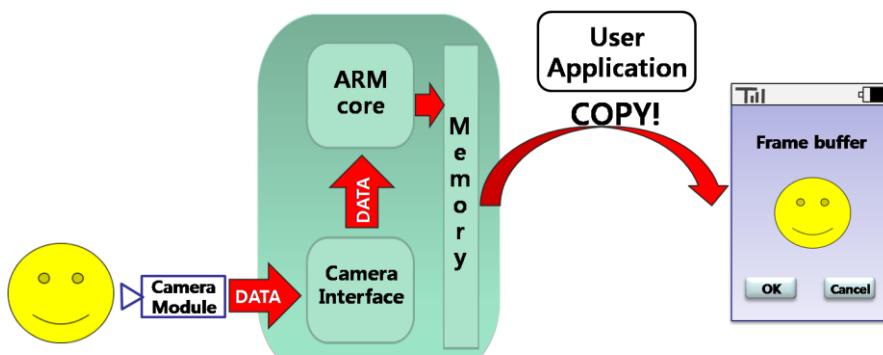


그림 6-43 Camera Preview 흐름도 [다시 그려야 함]

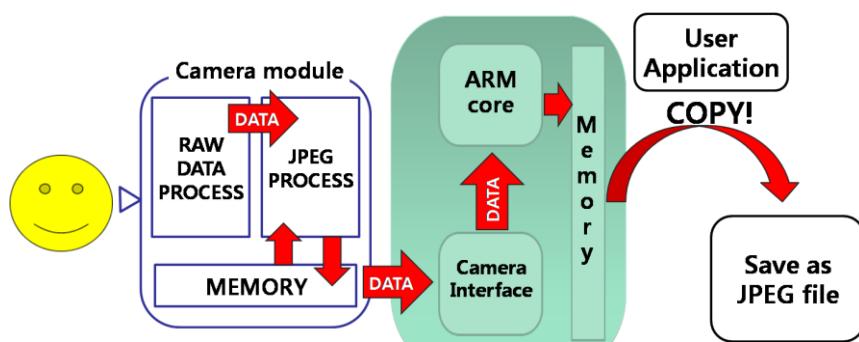


그림 6-44 Still shot 흐름도 - JPEG 파일 포맷 [다시 그려야 함]

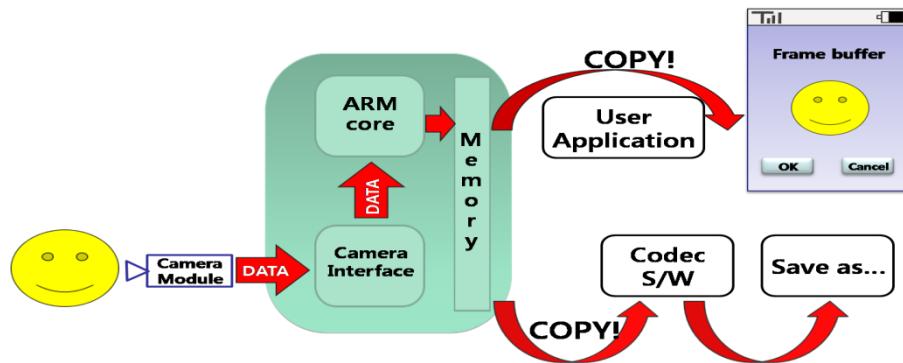


그림 6-45 Motion Picture 흐름도 [다시 그려야 함]

지금까지 설명한 Camera 센서(혹은 모듈)와 Core Interface(혹은 Camera Core)의 관계를 데이터 흐름의 관점에서 다시 표현해 보면 다음과 같다. 아래 그림에서 처음 언급되는 V4L2에 대해서는 다음 절에서 상세히 설명하도록 하겠다.

<Camera 데이터 흐름>

- 1) /dev/video0 파일을 오픈한다.
- 2) Camera 센서를 통해 데이터를 캡처한다.
- 3) 2에서 캡처한 데이터를 가공한다.
 - scaler, rotator, colorspace
- 3') 2에서 캡처한 데이터를 YUV 포맷으로 형식 변환하여, JPEG 파일로 저장한다.
- 4) 3에서 가공된 데이터를 서피스플링어를 거쳐 RGB 포맷으로 화면에 출력한다.

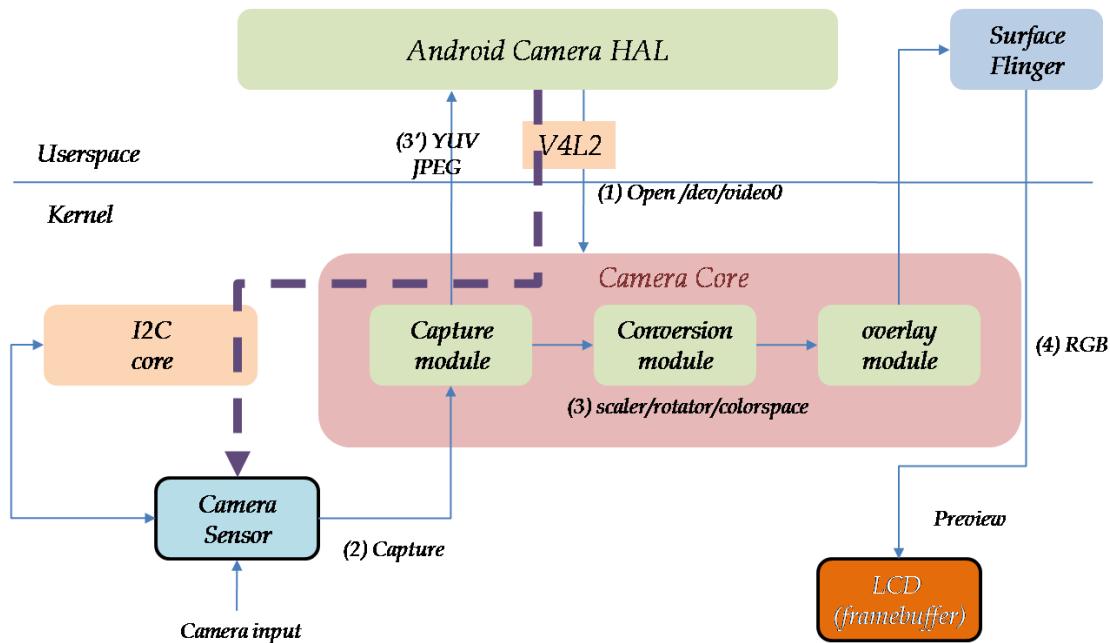


그림 6-46 카메라 처리 드라이버 개관

5.2 V4L2 서브시스템

V4L(Video4Linux)은 Video Capture 응용 프로그래밍 인터페이스로 Alan Cox에 의해 제안되었다. V4L은 USB webcams, TV tuners 등의 일부 영역에서 사용되었으며, 리눅스 커널과 밀접히 연결되어 있어 다양한 장치로 확장하는데 어려움이 있었다. 한편, Bill Dirks는 기존의 V4L이 안고 있는 문제점을 개선한 새로운 버전인 V4L2를 설계하게 되었는데, 기존의 V4L과 호환되지는 않았지만, 더 많은 캡처(capture) 장치를 지원할 수 있도록 기능이 확장되었다. V4L2가 지원하는 장치로는 aMSN, Gstreamer, Mplayer, MythTV, OpenCV, Skype, VLC media player 등 그 범위가 매우 다양하다.



그림 6-47 V4L2 개요

V4L2를 카메라 센서(모듈)와 Core 인터페이스 관점에서 다시 그려 보면, 그림 6-48과 같다고 볼 수 있다. V4L2는 Camera Core 인터페이스는 물론이고, 센서 모듈에도 영향을 주고 있음을 알 수 있다. 뿐만 아니라, 아래 그림에는 표현되어 있지 않지만, V4L2가 사용자 영역의 프로그램을 위한 인터페이스인 만큼, 당연히 사용자 영역의 프로그램도 V4L2 API를 사용하여야만 한다.

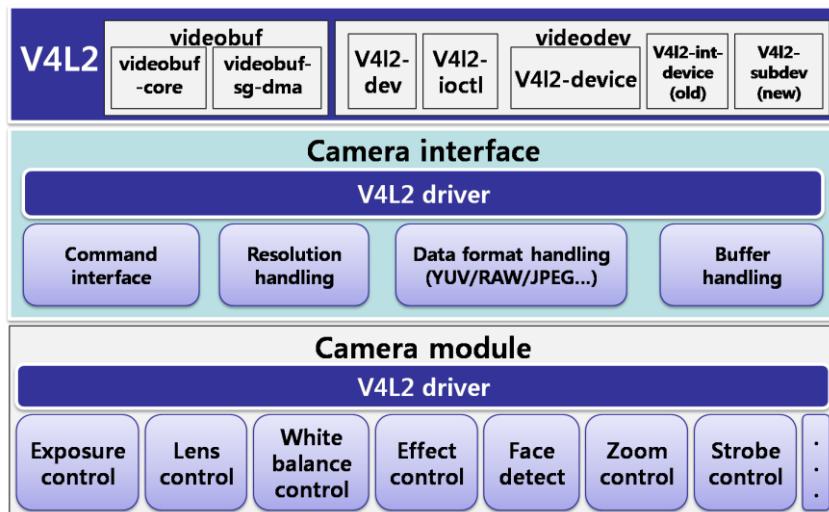


그림 6-48 Camera 모듈(센서), Camera 인터페이스, V4L2와의 관계도 [다시 그려야 함]

V4L2 사용자 영역 인터페이스

지금부터는 사용자 영역의 프로그램으로부터 V4L2 드라이버에 이르기까지의 흐름을 상세히 분석

해 봄으로써, V4L2에 대한 이해를 돋고자 한다.

<V4L2를 사용하는 응용 프로그램 코드 흐름>

- 1) 장치를 연다(*Opening the device*)
- 2) 장치의 속성을 변경한다(*video, audio* 입력, *video* 표준, *picture* 밝기 등 선택).
- 3) 데이터 포맷을 협상하여 결정한다.
- 4) 입/출력 방법을 협상하여 결정한다.
- 5) 실제 *loop*를 돌며 입/출력을 처리한다.
- 6) 장치를 닫는다.

이상의 과정을 그림으로 정리해 보면 다음과 같다.

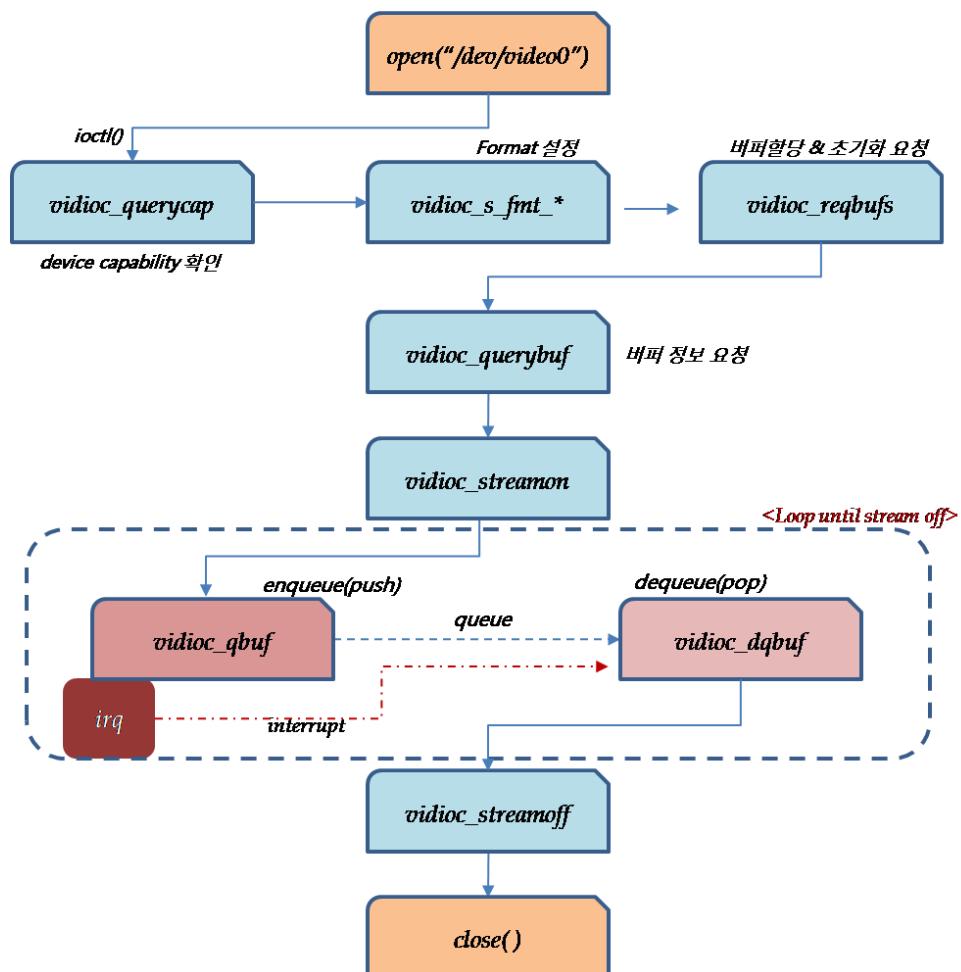


그림 6-49 V4L2를 사용하는 응용 프로그램 전체 흐름도

V4L2 드라이버 구조

V4L2를 사용하는 응용 프로그램 코드를 살펴 보았으니, 이제는 V4L2 드라이버의 내부 구조를 살펴보아야 할 차례이다. V4L2 드라이버는 상당히 복잡한 구조(여러 개의 구조체를 사용함)로 이루-

어져 있는데, 이를 구성하는 몇가지 요소를 먼저 정리해 보면 다음과 같다.

<V4L2 드라이버 구성 요소>

1) 장치의 상태 정보를 포함한 각 장치 인스턴스를 위한 구조체

- 관련 구조체: struct v4l2_device

2) 하위 장치를 초기화하고, 제어하는 방법

- 관련 구조체: struct v4l2_subdev
- 여기서 하위 장치(sub-device)는 카메라 센서에 해당함.

3) V4L2 장치 노드(/dev/videoX, /dev/vbiX and /dev/radioX)를 생성하고, 장치 노드 관련 전용 데이터를 유지하기

- 관련 구조체: struct video_device 구조체
- 이 구조체 안의 const struct v4l2_ioctl_ops 멤버가 사용자 영역과 연결되는 부분임.

4) 파일 핸들 관련 구조체

- 관련 구조체: v4l2_fh

5) 비디오 버퍼 처리 관련 코드

V4L2 프레임워크는 흡사 드라이버 구조를 닮았다. 먼저, 장치 인스턴스(instance) 데이터를 위해 v4l2_device 구조체를 가지고 있으며, v4l2_subdev 구조체를 통해 하위 장치(sub-device) 인스턴스를 표현하고 있다. 또한 video_device 구조체는 장치 노드(사용자 영역에 표출) 데이터를 저장하며, v4l2_fh 구조체는 파일 핸들 인스턴스를 유지한다. 이들간의 관계를 그림으로 표현해 보면 다음과 같다.

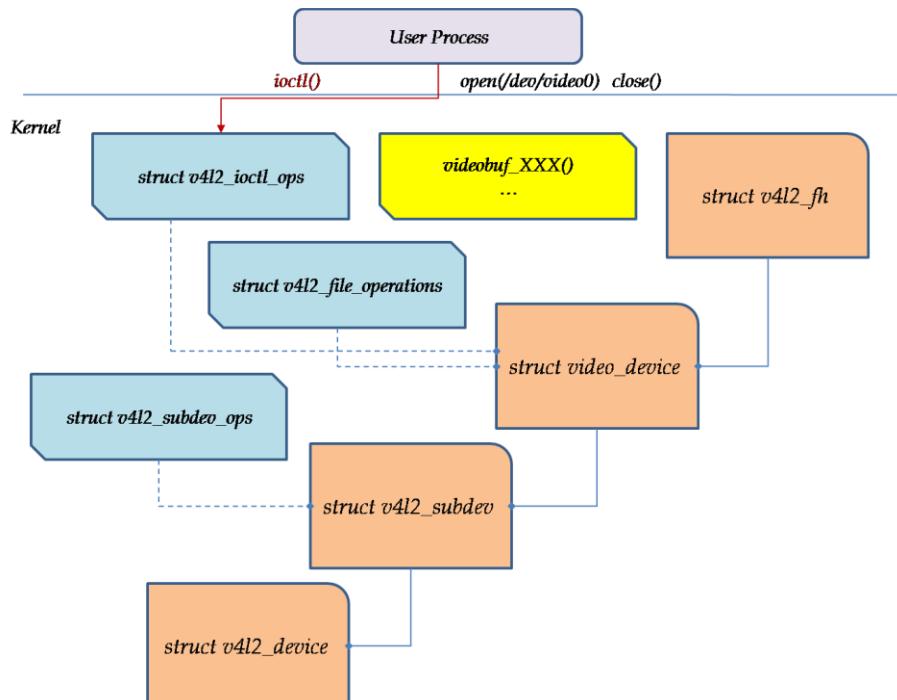


그림 6-50 V4L2 드라이버 구조

지금부터는 위에서 언급한 주요 구조체를 하나씩 살펴보기로 하겠다. 먼저, 각각의 장치 인스턴스는 v4l2_device 구조체로 표현되는데, v4l2_device 구조체를 살펴 보면 다음과 같다.

코드 6-25 struct v4l2_device(include/media/v4l2-device.h)

```
struct v4l2_device {
    /* dev->driver_data points to this struct.

    Note: dev might be NULL if there is no parent device
    as is the case with e.g. ISA devices. */

    struct device *dev;

#if defined(CONFIG_MEDIA_CONTROLLER)
    struct media_device *mdev;
#endif

    /* used to keep track of the registered subdevs */
    struct list_head subdevs;

    /* lock this struct; can be used by the driver as well if this
     struct is embedded into a larger struct. */
    spinlock_t lock;

    /* unique device name, by default the driver name + bus ID */
    char name[V4L2_DEVICE_NAME_SIZE];

    /* notify callback called by some sub-devices. */
    void (*notify)(struct v4l2_subdev *sd,
                   unsigned int notification, void *arg);

    /* The control handler. May be NULL. */
    struct v4l2_ctrl_handler *ctrl_handler;

    /* Device's priority state */
    struct v4l2_prio_state prio;

    /* BKL replacement mutex. Temporary solution only. */
    struct mutex ioctl_lock;

    /* Keep track of the references to this struct. */
    struct kref ref;

    /* Release function that is called when the ref count goes to 0. */
    void (*release)(struct v4l2_device *v4l2_dev);
};
```

v4l2_device를 등록 및 해제하기 위해서는 아래의 API를 사용해야 한다.

```
v4l2_device_register(struct device *dev, struct v4l2_device *v4l2_dev);
v4l2_device_unregister(struct v4l2_device *v4l2_dev);
```

많은 드라이버들은 하위 장치들과 통신할 필요가 있다. 이를 표현하는 v4l2_subdev 구조체를 살펴보면 다음과 같다.

코드 6-26 struct v4l2_subdev(include/media/v4l2-subdev.h)

```
struct v4l2_subdev {
#ifndef CONFIG_MEDIA_CONTROLLER
    struct media_entity entity;
#endif
    struct list_head list;
    struct module *owner;
    u32 flags;
    struct v4l2_device *v4l2_dev;
    const struct v4l2_subdev_ops *ops;
    /* Never call these internal ops from within a driver! */
    const struct v4l2_subdev_internal_ops *internal_ops;
    /* The control handler of this subdev. May be NULL. */
    struct v4l2_ctrl_handler *ctrl_handler;
    /* name must be unique */
    char name[V4L2_SUBDEV_NAME_SIZE];
    /* can be used to group similar subdevs, value is driver-specific */
    u32 grp_id;
    /* pointer to private data */
    void *dev_priv;
    void *host_priv;
    /* subdev device node */
    struct video_device *devnode;
};

struct v4l2_subdev_ops {
    const struct v4l2_subdev_core_ops    *core;
    const struct v4l2_subdev_tuner_ops   *tuner;
    const struct v4l2_subdev_audio_ops   *audio;
    const struct v4l2_subdev_video_ops   *video;
    const struct v4l2_subdev_vbi_ops     *vbi;
    const struct v4l2_subdev_ir_ops      *ir;
    const struct v4l2_subdev_sensor_ops  *sensor;
    const struct v4l2_subdev_pad_ops     *pad;
```

```

};

struct v4l2_subdev_core_ops {
    int (*g_chip_ident)(struct v4l2_subdev *sd, struct v4l2_dbg_chip_ident *chip);
    int (*log_status)(struct v4l2_subdev *sd);
    int (*s_io_pin_config)(struct v4l2_subdev *sd, size_t n,
                           struct v4l2_subdev_io_pin_config *pincfg);
    int (*init)(struct v4l2_subdev *sd, u32 val);
    int (*load_fw)(struct v4l2_subdev *sd);
    int (*reset)(struct v4l2_subdev *sd, u32 val);
    int (*s_gpio)(struct v4l2_subdev *sd, u32 val);
    int (*queryctrl)(struct v4l2_subdev *sd, struct v4l2_queryctrl *qc);
    int (*g_ctrl)(struct v4l2_subdev *sd, struct v4l2_control *ctrl);
    int (*s_ctrl)(struct v4l2_subdev *sd, struct v4l2_control *ctrl);
    int (*g_ext_ctrls)(struct v4l2_subdev *sd, struct v4l2_ext_controls *ctrls);
    int (*s_ext_ctrls)(struct v4l2_subdev *sd, struct v4l2_ext_controls *ctrls);
    int (*try_ext_ctrls)(struct v4l2_subdev *sd, struct v4l2_ext_controls *ctrls);
    int (*querymenu)(struct v4l2_subdev *sd, struct v4l2_querymenu *qm);
    int (*g_std)(struct v4l2_subdev *sd, v4l2_std_id *norm);
    int (*s_std)(struct v4l2_subdev *sd, v4l2_std_id norm);
    long (*ioctl)(struct v4l2_subdev *sd, unsigned int cmd, void *arg);
#define CONFIG_VIDEO_ADV_DEBUG
    int (*g_register)(struct v4l2_subdev *sd, struct v4l2_dbg_register *reg);
    int (*s_register)(struct v4l2_subdev *sd, const struct v4l2_dbg_register *reg);
#endif
    int (*s_power)(struct v4l2_subdev *sd, int on);
    int (*interrupt_service_routine)(struct v4l2_subdev *sd,
                                    u32 status, bool *handled);
    int (*subscribe_event)(struct v4l2_subdev *sd, struct v4l2_fh *fh,
                          struct v4l2_event_subscription *sub);
    int (*unsubscribe_event)(struct v4l2_subdev *sd, struct v4l2_fh *fh,
                           struct v4l2_event_subscription *sub);
};

}

```

V4L2 하위 장치를 등록 및 해제하는 함수는 아래와 같다.

```

int err = v4l2_device_register_subdev(v4l2_dev, sd);
v4l2_device_unregister_subdev(sd);

```

/dev 디렉토리에 존재하는 장치 노드(파일)는 video_device 구조체를 사용하여 만들어지게 된다. 이 구조체 내용을 살펴보면 다음과 같다.

코드 6-27 struct video_device(include/media/v4l2-dev.h)

```
struct video_device
{
    #if defined(CONFIG_MEDIA_CONTROLLER)
        struct media_entity entity;
    #endif

    /* device ops */
    const struct v4l2_file_operations *fops;

    /* sysfs */
    struct device dev;          /* v4l device */
    struct cdev *cdev;          /* character device */

    /* Set either parent or v4l2_dev if your driver uses v4l2_device */
    struct device *parent;      /* device parent */
    struct v4l2_device *v4l2_dev; /* v4l2_device parent */

    /* Control handler associated with this device node. May be NULL. */
    struct v4l2_ctrl_handler *ctrl_handler;

    /* vb2_queue associated with this device node. May be NULL. */
    struct vb2_queue *queue;

    /* Priority state. If NULL, then v4l2_dev->prio will be used. */
    struct v4l2_prio_state *prio;

    /* device info */
    char name[32];
    int vfl_type;   /* device type */
    int vfl_dir;    /* receiver, transmitter or m2m */
    /* 'minor' is set to -1 if the registration failed */
    int minor;
    u16 num;
    /* use bitops to set/clear/test flags */
    unsigned long flags;
```

```

/* attribute to differentiate multiple indices on one physical device */
int index;

/* V4L2 file handles */
spinlock_t      fh_lock; /* Lock for all v4l2_fhs */
struct list_head    fh_list; /* List of struct v4l2_fh */

int debug;           /* Activates debug level*/

/* Video standard vars */
v4l2_std_id tvnorms;        /* Supported tv norms */
v4l2_std_id current_norm;   /* Current tnorm */

/* callbacks */
void (*release)(struct video_device *vdev);

/* ioctl callbacks */
const struct v4l2_ioctl_ops *ioctl_ops;
DECLARE_BITMAP(valid_ioctls, BASE_VIDIOC_PRIVATE);

/* serialization lock */
DECLARE_BITMAP(disable_locking, BASE_VIDIOC_PRIVATE);
struct mutex *lock;
};

```

//이 구조체가 최종적으로 사용자 영역의 ioctl() 호출 시, 연결되는 부분임.

```

struct v4l2_ioctl_ops {                                     [in include/media/v4l2-ioctl.h file]
    /* ioctl callbacks */

    /* VIDIOC_QUERYCAP handler */
    int (*vidioc_querycap)(struct file *file, void *fh, struct v4l2_capability *cap);

    /* Priority handling */
    int (*vidioc_g_priority)  (struct file *file, void *fh,
                               enum v4l2_priority *p);
    int (*vidioc_s_priority)  (struct file *file, void *fh,
                               enum v4l2_priority p);

    /* VIDIOC_ENUM_FMT handlers */

```

```

int (*vidioc_enum_fmt_vid_cap)      (struct file *file, void *fh,
                                    struct v4l2_fmtdesc *f);
int (*vidioc_enum_fmt_vid_overlay) (struct file *file, void *fh,
                                    struct v4l2_fmtdesc *f);
int (*vidioc_enum_fmt_vid_out)      (struct file *file, void *fh,
                                    struct v4l2_fmtdesc *f);
int (*vidioc_enum_fmt_vid_cap_mplane)(struct file *file, void *fh,
                                      struct v4l2_fmtdesc *f);
int (*vidioc_enum_fmt_vid_out_mplane)(struct file *file, void *fh,
                                      struct v4l2_fmtdesc *f);

/* VIDIOC_G_FMT handlers */
int (*vidioc_g_fmt_vid_cap)      (struct file *file, void *fh,
                                 struct v4l2_format *f);
int (*vidioc_g_fmt_vid_overlay)(struct file *file, void *fh,
                                struct v4l2_format *f);
int (*vidioc_g_fmt_vid_out)      (struct file *file, void *fh,
                                 struct v4l2_format *f);
int (*vidioc_g_fmt_vid_out_overlay)(struct file *file, void *fh,
                                    struct v4l2_format *f);
int (*vidioc_g_fmt_vbi_cap)      (struct file *file, void *fh,
                                 struct v4l2_format *f);
[...]
};

//v4l2_ioctl_ops 구조체 사용 예: drivers/media/platform omap3isp/ispvideo.c
//사용자 영역에서 ioctl(VIDIOC_XXX...) 형태의 함수 호출 시, 아래의 callback 함수가 호출될 것
임.

//그림 6-50과 관련된 부분이기도 함.

static const struct v4l2_ioctl_ops isp_video_ioctl_ops = {
    .vidioc_querycap      = isp_video_querycap,
    .vidioc_g_fmt_vid_cap = isp_video_get_format,
    .vidioc_s_fmt_vid_cap = isp_video_set_format,
    .vidioc_try_fmt_vid_cap = isp_video_try_format,
    .vidioc_g_fmt_vid_out = isp_video_get_format,
    .vidioc_s_fmt_vid_out = isp_video_set_format,
    .vidioc_try_fmt_vid_out = isp_video_try_format,
    .vidioc_cropcap       = isp_video_cropcap,
    .vidioc_g_crop        = isp_video_get_crop,

```

//v4l2_ioctl_ops 구조체 사용 예: drivers/media/platform omap3isp/ispvideo.c
//사용자 영역에서 ioctl(VIDIOC_XXX...) 형태의 함수 호출 시, 아래의 callback 함수가 호출될 것
임.

//그림 6-50과 관련된 부분이기도 함.

```

static const struct v4l2_ioctl_ops isp_video_ioctl_ops = {
    .vidioc_querycap      = isp_video_querycap,
    .vidioc_g_fmt_vid_cap = isp_video_get_format,
    .vidioc_s_fmt_vid_cap = isp_video_set_format,
    .vidioc_try_fmt_vid_cap = isp_video_try_format,
    .vidioc_g_fmt_vid_out = isp_video_get_format,
    .vidioc_s_fmt_vid_out = isp_video_set_format,
    .vidioc_try_fmt_vid_out = isp_video_try_format,
    .vidioc_cropcap       = isp_video_cropcap,
    .vidioc_g_crop        = isp_video_get_crop,

```

```

.vidioc_s_crop          = isp_video_set_crop,
.vidioc_g_parm          = isp_video_get_param,
.vidioc_s_parm          = isp_video_set_param,
.vidioc_reqbufs         = isp_video_reqbufs,
.vidioc_querybuf        = isp_video_querybuf,
.vidioc_qbuf            = isp_video_qbuf,
.vidioc_dqbuf           = isp_video_dqbuf,
.vidioc_streamon        = isp_video_streamon,
.vidioc_streamoff       = isp_video_streamoff,
.vidioc_enum_input      = isp_video_enum_input,
.vidioc_g_input          = isp_video_g_input,
.vidioc_s_input          = isp_video_s_input,
};


```

video_device를 등록 및 해제하기 위해서는 아래 함수를 사용하여야 한다.

```

err = video_register_device(vdev, VFL_TYPE_GRABBER, -1);
video_unregister_device(vdev);


```

v4l2_fh 구조체는 V4L2 프레임워크에서 사용하는 파일 핸들 전용 데이터를 유지하는 용도로 사용되는데, 이를 살펴 보면 다음과 같다.

코드 6-28 struct v4l2_fh(include/media/v4l2-fh.h)

```

struct v4l2_fh {
    struct list_head    list;
    struct video_device *vdev;
    struct v4l2_ctrl_handler *ctrl_handler;
    enum v4l2_priority   prio;

    /* Events */
    wait_queue_head_t    wait;
    struct list_head    subscribed; /* Subscribed events */
    struct list_head    available; /* Dequeueable event */
    unsigned int         navailable;
    u32                 sequence;
};


```

Camera 센서 예제 소개

아래 코드는 Renesas Solutions Corp에서 구현한 mt9t112camera 센서 드라이버의 주요 부분(v4l2 sub system과 i2c driver 관점)만을 발췌한 것이다.

코드 6-29 arch/arm/mach-shmobile/board-armadillo800eva.c

From arch/arm/mach-shmobile/board-armadillo800eva.c

```
static struct i2c_board_info i2c_camera_mt9t111 = {
    I2C_BOARD_INFO("mt9t112", 0x3d),
};

static struct mt9t112_camera_info mt9t111_info = {
    .divider = { 16, 0, 0, 7, 0, 10, 14, 7, 7 },
};

static struct soc_camera_link mt9t111_link = { ① //platform data 선언
    .i2c_adapter_id = 0,
    .bus_id = 0,
    .board_info = &i2c_camera_mt9t111,
    .power = mt9t111_power,
    .priv = &mt9t111_info,
};

static struct platform_device camera_device = { ② //platform device 선언
    .name = "soc-camera-pdrv",
    .id = 0,
    .dev = {
        .platform_data = &mt9t111_link,
    },
};

[...]
/*
 * board devices
 */

static struct platform_device *eva_devices[] __initdata = { ③ //초기화할 platform device 배열 선언
    &lcdc0_device,
    &gpio_keys_device,
    &sh_eth_device,
```

```

&vcc_sdhi0,
&vccq_sdhi0,
&sdhi0_device,
&sh_mmcif_device,
&hdmi_device,
&hdmi_lcdc_device,
&camera_device,
&ceu0_device,
&fsi_device,
&fsi_wm8978_device,
&fsi_hdmi_device,
&i2c_gpio_device,
};

[...]
static void __init eva_init(void)
{
    [...]
    platform_add_devices(eva_devices, ARRAY_SIZE(eva_devices)); ④
    // soc camera를 platform device로 등록
    [...]
}

```

코드 6-30 drivers/media/i2c/soc_camera/mt9t112.c

From [drivers/media/i2c/soc_camera/mt9t112.c](#)

```

static struct v4l2_subdev_core_ops mt9t112_subdev_core_ops = {
    .g_chip_ident    = mt9t112_g_chip_ident,
#ifdef CONFIG_VIDEO_ADV_DEBUG
    .g_register      = mt9t112_g_register,
    .s_register      = mt9t112_s_register,
#endif
    .s_power         = mt9t112_s_power,
};

[...]
static struct v4l2_subdev_video_ops mt9t112_subdev_video_ops = {
    //V4L2 video operations 정의
    .s_stream        = mt9t112_s_stream,
    .g_mbus_fmt     = mt9t112_g_fmt,
}

```

```

.s_mbus_fmt = mt9t112_s_fmt,
.try_mbus_fmt = mt9t112_try_fmt,
.cropcap = mt9t112_cropcap,
.g_crop = mt9t112_g_crop,
.s_crop = mt9t112_s_crop,
.enum_mbus_fmt = mt9t112_enum_fmt,
.g_mbus_config = mt9t112_g_mbus_config,
.s_mbus_config = mt9t112_s_mbus_config,
};

static struct v4l2_subdev_ops mt9t112_subdev_ops = {
    .core = &mt9t112_subdev_core_ops,
    .video = &mt9t112_subdev_video_ops,
};

[...]

static int mt9t112_probe(struct i2c_client *client, const struct i2c_device_id *did)
//i2c driver probe() 함수 호출
{
    struct mt9t112_priv *priv;
    struct soc_camera_subdev_desc *ssdd = soc_camera_i2c_to_desc(client);
    struct v4l2_rect rect = {
        .width = VGA_WIDTH,
        .height = VGA_HEIGHT,
        .left = (MAX_WIDTH - VGA_WIDTH) / 2,
        .top = (MAX_HEIGHT - VGA_HEIGHT) / 2,
    };
    int ret;

    if (!ssdd || !ssdd->drv_priv) {
        dev_err(&client->dev, "mt9t112: missing platform data!\n");
        return -EINVAL;
    }

    priv = devm_kzalloc(&client->dev, sizeof(*priv), GFP_KERNEL);
    if (!priv)
        return -ENOMEM;

    priv->info = ssdd->drv_priv;
}

```

```

v4l2_i2c_subdev_init(&priv->subdev, client, &mt9t112_subdev_ops);

    ret = mt9t112_camera_probe(client);
    if (ret)
        return ret;

    /* Cannot fail: using the default supported pixel code */
    mt9t112_set_params(priv, &rect, V4L2_MBUS_FMT_UYVY8_2X8);

    return ret;
}

[...]

static struct i2c_driver mt9t112_i2c_driver = { //i2c driver로 선언
    .driver = {
        .name = "mt9t112",
    },
    .probe     = mt9t112_probe,
    .remove    = mt9t112_remove,
    .id_table = mt9t112_id,
};

module_i2c_driver(mt9t112_i2c_driver);

```

6. 오디오 사운드 드라이버

본 절에서는 ALSA SoC 기반의 오디오 사운드 장치 및 관련 드라이버를 소개하고자 한다.

오디오 사운드 장치

일반적으로 SoC 형태의 Audio 장치는 그림 6-51과 같이 Audio Back-End 장치를 포함하고 있는데, 이 장치에서 수행하는 역할을 정리하면 다음과 같다.

- 1) CPU나 DSP 혹은 Audio Codec(아래 그림의 경우는 TWL6040 codec)으로부터 voice나 audio sample을 수신한다.
- 2) 1에서 수신한 audio data를 가공 후, 메모리나 analog 파트(codec 포함)로 송신한다.

일반적으로 Audio Back-End와 Codec간의 오디오 데이터는 I²S나 PCM 인터페이스를 통해 전달되

며, Audio Codec의 제어는 I²C(아래 그림에서는 McPDM 사용)를 통해 이루어진다고 말할 수 있다.

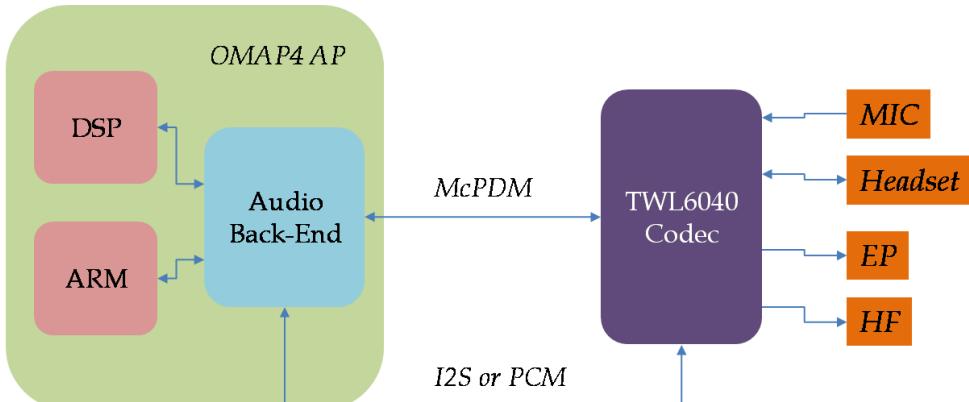


그림 6-51 OMAP4 Audio 장치 개요

오디오 사운드 서브시스템(ALSA SoC 계층)

ASoC(ALSA System on Chip) 계층을 둔 궁극적인 이유는 SoC 기반의 embedded system에서의 보다 좋은 ALSA 지원을 제공하기 위해서이다. 과거의 codec 드라이버는 SoC cpu에 종속된 형태로 구현되었다. 이는 코드 중복 등 이상적인 방향이 아니었다. 또한, 사용자 계층으로부터 시작되는 audio events 즉, Headphone/Mic 삽입 및 감지 event를 알릴 수 있는 표준화된 방법이 마련되어 있지 않았다. 마지막으로, 과거의 방식은 audio 재생 혹은 녹화 중, 전체 codec를 켜두는 경향이 있었다. 이러한 방식은 PC에는 적합할지 몰라도, embedded 장치에서는 전력의 낭비를 가져오게 된다. ASoC 계층은 이러한 문제를 해결하기 위해 설계되었으며, 다음과 같은 특징을 제공한다.

- 1) *Codec의 독립성*. 다른 플랫폼이나 머신에서 codec 드라이버를 재사용할 수 있도록 해준다.
- 2) *Codec과 SoC 간에 보다 쉬운 I2S/PCM 오디오 인터페이스 설정이 가능하도록 해 줌*. 각각의 SoC 인터페이스 및 code은 자신의 audio interface를 core에 등록해 주며, HW 파라미터를 알게 되는 시점에 설정이 이루어진다.
- 3) *동적 오디오 파워 관리(Audio Power Management - DAPM)*. DAPM은 자동으로 codec를 파워를 최소한으로 사용하는 상태로 만들어 준다. 이는 내부 codec audio routing이나 active stream에 따라서 power block을 내리거나 올려줌으로써 가능하다.

ALSA SoC Sound card 드라이버를 구성하는 3가지 요소를 열거하면 다음과 같다.

- 1) **Codec driver**: codec 드라이버는 플랫폼 독립적이며, audio control, audio interface, codec dpm 정의 및 codec IO 함수 관련 사항을 포함하고 있다.
- 2) **Platform driver**: platform 드라이버는 audio dma 엔진과 audio interface 드라이버(예를 들어, I2S, AC97, PCM)를 포함한다. [주의 사항] 여기서 말하는 Platform driver와 리눅스 장치 모델에서

말하는 *platform driver*는 아무런 연관 관계가 없다.

3) Machine driver: machine 드라이버는 마シン 전용 코드와 audio event를 처리한다.

ALSA Sound card 드라이버는 통상적으로 DMA driver(빠질 수도 있음), I²S driver, Codec driver 3개의 드라이버가 링크된 플랫폼 드라이버 형태를 띤다. DMA 드라이버는 Data 전송, I²S 드라이버는 I²S 시그널 생성, Codec 드라이버는 Digital to Analog 전환, In/Out device 믹싱(mixing) 등의 역할을 수행하게 된다. 동작 방식을 살펴 보면 각각의 드라이버들이 등록된 후, ALSA Soc Core에서 각 드라이버들 간의 링크가 수행되는 형태인데, 그림 6-52에 이들의 상관 관계를 표현해 보았다.

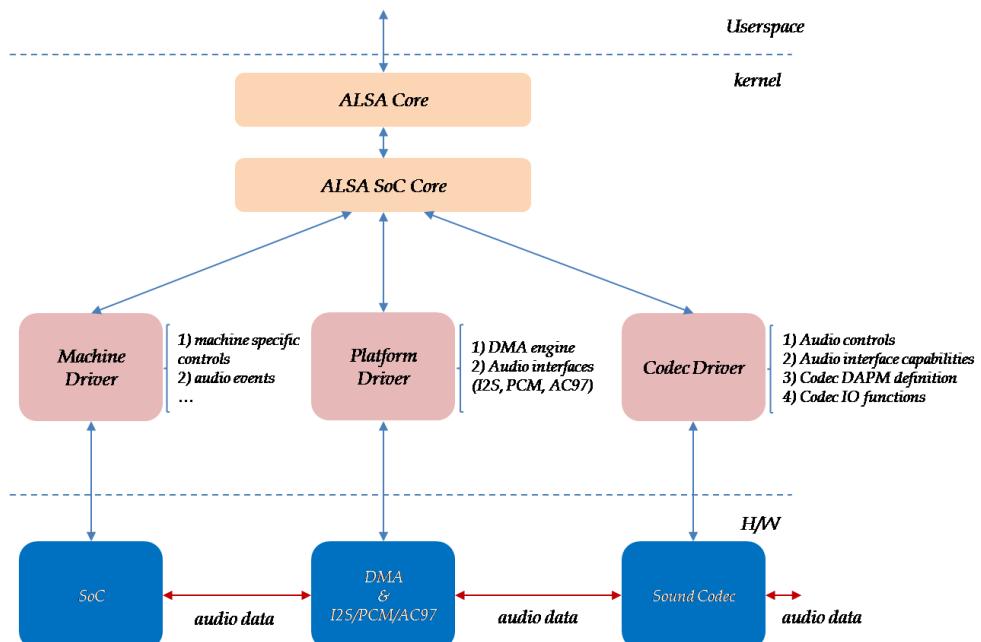


그림 6-52 Audio Sound 드라이버 구조 – ALSA 기반(ASoC)

DAI는 Digital Audio Interface를 뜻하며, I2S, PCM, AC97 등의 인터페이스를 사용하여 audio codec과 CPU를 연결시켜주는 개념을 뜻한다. DAI link가 확립된 상태에서, 그림 6-53은 MIC로부터 입력된 audio data가 다른 곳으로 전달되는 세가지 경로를 그림으로 표현해 주고 있다.

1) Modem을 타고 상대방 전화로 전달

- MIC input -> Codec -> I2S or PCM or AC97 interface -> CPU(audioflinger) -> Modem

2) 파일로 저장되었다가 재생되는 경우

- MIC input -> Codec -> I2S or PCM or AC97 interface -> CPU(audioflinger) -> Local File(i.e. PCM file)
- Local File -> CPU(audioflinger) -> I2S or PCM or AC97 interface -> Codec -> Earpiece or Speaker

3) 자신의 Earpiece나 Speaker로 바로 전달

- MIC input -> Codec bypass function -> Earpiece or Speaker

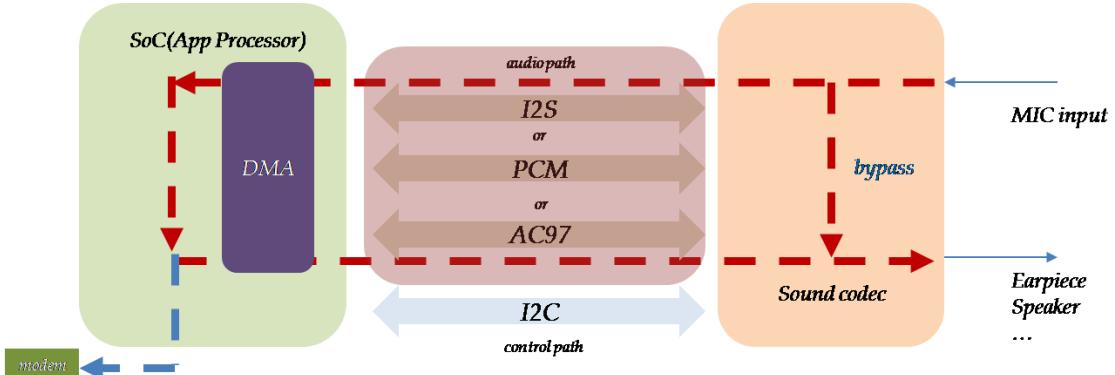


그림 6-53 Audio 장치와 Audio Sound Path(경로) - DAI(Digital audio interface)

그림 6-54는 안드로이드 AudioFlinger로부터 출발한 PCM data가 ASoC 계층을 거쳐 스피커(speaker)로 전달되는 과정을 표현한 것이다. 이를 위해서는 앞서 설명한 바와 같이 사전에 Codec과 CPU를 연결하는 과정(DAI Link)이 선행되어야 하며, 오디오 데이터 전송을 위한 인터페이스(I²S 혹은 PCM 등)가 정해져 있어야 한다.

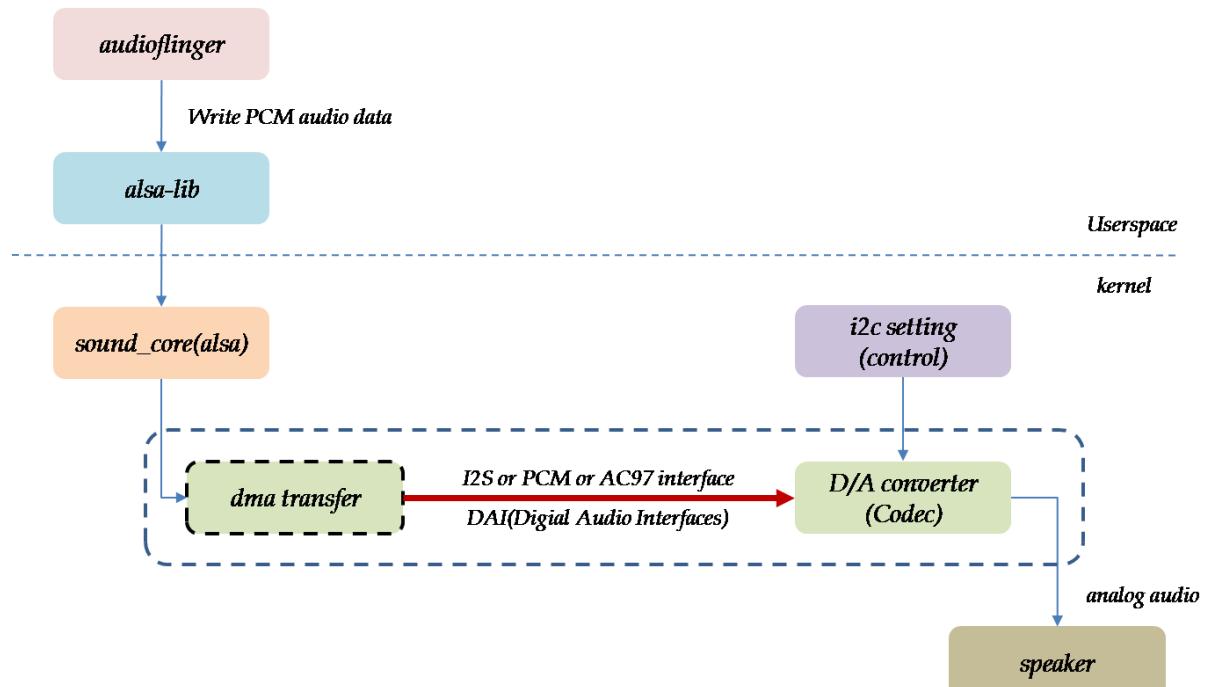


그림 6-54 PCM Data 출력 Path(경로)

Audio Sound 드라이버 코드 소개

1) Machine Driver 예

ASoC Machine Driver는 Codec과 SoC를 연결시켜 주는 역할을 하는 부분으로 머신 의존적인 코드를 말한다.

**코드 6-31 arch/arm/mach-omap2/board-omap4panda.c &
sound/soc/omap/omap-abe-tw16040.c**

From arch/arm/mach-omap2/board-omap4panda.c

⇒ Machine 드라이버

```
static struct omap_abe_tw16040_data panda_abe_audio_data = {  
    /* Audio out */  
    .has_hs      = ABE_TW16040_LEFT | ABE_TW16040_RIGHT,  
    /* HandsFree through expansion connector */  
    .has_hf      = ABE_TW16040_LEFT | ABE_TW16040_RIGHT,  
    /* PandaBoard: FM TX, PandaBoardES: can be connected to audio out */  
    .has_aux     = ABE_TW16040_LEFT | ABE_TW16040_RIGHT,  
    /* PandaBoard: FM RX, PandaBoardES: audio in */  
    .has_afm     = ABE_TW16040_LEFT | ABE_TW16040_RIGHT,  
    /* No jack detection. */  
    .jack_detection = 0,  
    /* MCLK input is 38.4MHz */  
    .mclk_freq   = 38400000,  
  
};  
  
static struct platform_device panda_abe_audio = {  
    .name        = "omap-abe-tw16040",  
    .id         = -1,  
    .dev = {  
        .platform_data = &panda_abe_audio_data,  
    },  
};  
[...]  
static struct platform_device *panda_devices[] __initdata = {  
    &leds_gpio,  
    &wl1271_device,  
    &panda_abe_audio,  
    &panda_hdmi_audio_codec,  
    &btwilink_device,  
};
```

```
[..]

static void __init omap4_panda_init(void)
{
    int package = OMAP_PACKAGE_CBS;
    int ret;

    if (omap_rev() == OMAP4430_REV_ES1_0)
        package = OMAP_PACKAGE_CBL;
    omap4_mux_init(board_mux, NULL, package);

    omap_panda_wlan_data.irq = gpio_to_irq(GPIO_WIFI_IRQ);
    ret = wl12xx_set_platform_data(&omap_panda_wlan_data);
    if (ret)
        pr_err("error setting wl12xx data: %d\n", ret);

    omap4_panda_init_rev();
    omap4_panda_i2c_init();
    platform_add_devices(panda_devices, ARRAY_SIZE(panda_devices));
    platform_device_register(&omap_vwlan_device);
    omap_serial_init();
    omap_sdrc_init(NULL, NULL);
    omap4_twl6030_hsmmc_init(mmc);
    omap4_ehci_init();
    usb_bind_phy("musb-hdrc.2.auto", 0, "omap-usb2.3.auto");
    usb_musb_init(&musb_board_data);
    omap4_panda_display_init();
}

}
```

From [sound/soc/omap/omap-abe-tw16040.c](#)

- ⇒ Machine 드라이버
- ⇒ SoC audio for TI OMAP based boards with ABE and twl6040 codec

/ DAI(Digital audio interface) glue – codec과 CPU를 연결시켜주는 역할 수행 */*

```
static struct snd_soc_dai_link abe_tw16040_dai_links[] = {
{
    .name = "TWL6040",
    .stream_name = "TWL6040",
    .cpu_dai_name = "omap-mcpdm",
    .codec_dai_name = "twl6040-legacy",
```

```

.platform_name = "omap-pcm-audio", /* ASoC platform driver와 연결 */
.codec_name = "twl6040-codec",      /* ASoC codec driver와 연결 */
.init = omap_abe_twl6040_init,
.ops = &omap_abe_ops,
},
{
    .name = "DMIC",
    .stream_name = "DMIC Capture",
    .cpu_dai_name = "omap-dmic",
    .codec_dai_name = "dmic-hifi",
    .platform_name = "omap-pcm-audio",
    .codec_name = "dmic-codec",
    .init = omap_abe_dmic_init,
    .ops = &omap_abe_dmic_ops,
},
};

/* Audio machine driver */
static struct snd_soc_card omap_abe_card = {
    .owner = THIS_MODULE,

    .dapm_widgets = twl6040_dapm_widgets,
    .num_dapm_widgets = ARRAY_SIZE(twl6040_dapm_widgets),
    .dapm_routes = audio_map,
    .num_dapm_routes = ARRAY_SIZE(audio_map),
};

[...]
static struct platform_driver omap_abe_driver = {
    .driver = {
        .name = "omap-abe-twl6040",
        .owner = THIS_MODULE,
        .pm = &snd_soc_pm_ops,
        .of_match_table = omap_abe_of_match,
    },
    .probe = omap_abe_probe,
    .remove = omap_abe_remove,
};

```

```
module_platform_driver(omap_abe_driver);
```

2) Platform Driver 예

ASoC Platform Driver는 Codec과 SoC 간의 오디오 데이터 전송을 관리하는 PCM, I²S, AC97 등의 인터페이스를 말한다. 아래 예에서는 PCM 인터페이스를 사용한 경우를 보여주고 있다.

코드 6-32 sound/soc/omap/omap-pcm.c

From [sound/soc/omap/omap-pcm.c](#)

- ⇒ Platform Driver
- ⇒ ALSA PCM interface for the OMAP SoC

```
static struct snd_pcm_ops omap_pcm_ops = {  
    .open      = omap_pcm_open,  
    .close     = snd_dmaengine_pcm_close_release_chan,  
    .ioctl     = snd_pcm_lib_ioctl,  
    .hw_params = omap_pcm_hw_params,  
    .hw_free   = omap_pcm_hw_free,  
    .trigger   = snd_dmaengine_pcm_trigger,  
    .pointer   = omap_pcm_pointer,  
    .mmap      = omap_pcm_mmap,  
};  
[...]  
static struct snd_soc_platform_driver omap_soc_platform = {  
    .ops      = &omap_pcm_ops,  
    .pcm_new  = omap_pcm_new,  
    .pcm_free = omap_pcm_free_dma_buffers,  
};  
  
static int omap_pcm_probe(struct platform_device *pdev)  
{  
    return snd_soc_register_platform(&pdev->dev,  
                                    &omap_soc_platform);  
}  
  
static int omap_pcm_remove(struct platform_device *pdev)  
{  
    snd_soc_unregister_platform(&pdev->dev);  
    return 0;  
}
```

```

static struct platform_driver omap_pcm_driver = {
    .driver = {
        .name = "omap-pcm-audio",
        .owner = THIS_MODULE,
    },
    .probe = omap_pcm_probe,
    .remove = omap_pcm_remove,
};

module_platform_driver(omap_pcm_driver);

```

3) Codec Driver 예

Codec 드라이버의 경우, 보통은 I²C로 제어하는 경우가 많지만 아래 예제(twl6040 codec)의 경우는 OMAP4 McPDM(Multi-channel Pulse Density Modulation)을 이용하고 있음을 알 수 있다. 참고로, I²C를 사용한 예는 sound/soc/codecs 디렉토리 아래에서 쉽게 찾아볼 수가 있다.

코드 6-33 sound/soc/codecs/twl6040.c

From [sound/soc/codecs/twl6040.c](#)

⇒ ALSA SoC TWL6040 codec driver

```

static struct snd_soc_codec_driver soc_codec_dev_twl6040 = {
    .probe = twl6040_probe,
    .remove = twl6040_remove,
    .suspend = twl6040_suspend,
    .resume = twl6040_resume,
    .read = twl6040_read_reg_cache,
    .write = twl6040_write,
    .set_bias_level = twl6040_set_bias_level,
    .reg_cache_size = ARRAY_SIZE(twl6040_reg),
    .reg_word_size = sizeof(u8),
    .reg_cache_default = twl6040_reg,
    .ignore_pmdown_time = true,

    .controls = twl6040_snd_controls,
    .num_controls = ARRAY_SIZE(twl6040_snd_controls),
    .dapm_widgets = twl6040_dapm_widgets,
    .num_dapm_widgets = ARRAY_SIZE(twl6040_dapm_widgets),
};

```

```

.dapm_routes = intercon,
.num_dapm_routes = ARRAY_SIZE(intercon),
};

static int twl6040_codec_probe(struct platform_device *pdev)
{
    return snd_soc_register_codec(&pdev->dev, &soc_codec_dev_twl6040,
                                twl6040_dai, ARRAY_SIZE(twl6040_dai));
}

[...]

static struct platform_driver twl6040_codec_driver = {
    .driver = {
        .name = "twl6040-codec",
        .owner = THIS_MODULE,
    },
    .probe = twl6040_codec_probe,
    .remove = twl6040_codec_remove,
};

module_platform_driver(twl6040_codec_driver);

```

7. Wi-Fi 드라이버

Wi-Fi 장치 소개

이번 장에서는 Wi-Fi와 Bluetooth를 위한 칩(chip)인 Broadcom BCM4329 Combo chip을 소개하고자 한다. BCM4329는 내부적으로 WL(Wi-Fi), BT(Bluetooth), FM(radio)의 세 부분으로 다시 나뉘어지게 되며, 공통적으로는 2.4Ghz RF 수신 부를 서로 공유하고 있다. 또한 Wi-Fi, BT의 경우는 각각 CORTEX-M3, ARM7TDMIS를 CPU로 하고 있으며, 자체 ROM 내에 간단한 OS(firmware)를 탑재하여 운용하는 형태로 구성되어 있어서, Linux 상에서 각각의 드라이버를 구동 시, Wi-Fi, BT 용 firmware를 함께 구동해 주어야만, 정상적으로 동작할 수 있다.

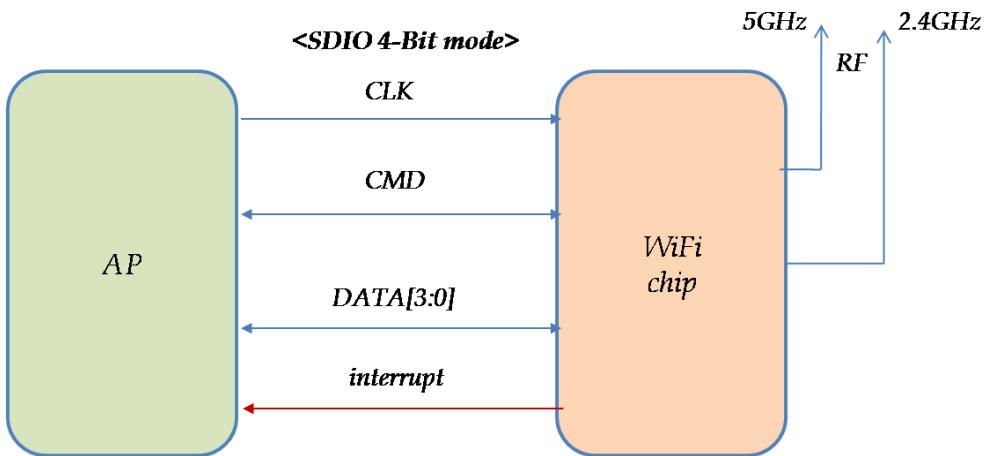


그림 6-55 BCM4329 Combo chip – Wi-Fi

Wi-Fi 드라이버 소개

그림 6-56은 Wi-Fi 드라이버의 개념을 주변 모듈(kernel TCP/IP stack, bcm4329 firmware 등)과의 관점에서 정리한 것이다. 그림 6-56에서 알 수 있는 것처럼, Wi-Fi 드라이버는 크게 다음의 4가지 코드로 구성되어 있는데, 실제 RF에 관련 처리 등 복잡한 부분은 bcm4329 chipset에서 담당하고 있어서, 그 구조(Wi-Fi 드라이버)가 간단할 것 같지만, 실제로는 SDIO 등의 인터페이스가 매우 난해한 형태를 띠고 있음을 알 수 있다.

<Wi-Fi 드라이버의 구성>

- 1) *net_device interface*
- 2) *SDIO interface*
- 3) *Chipset specific codes*
- 4) *Wireless extenions*

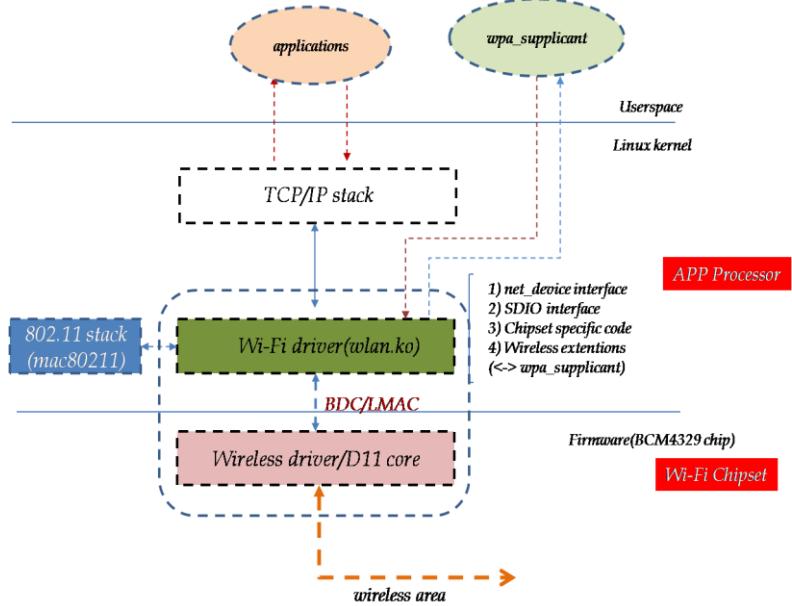


그림 6-56 Wi-Fi 드라이버 구조

그림 6-57은 `wpa_supplicant`에서 `ioctl`로 wifi 관련 명령(wireless extension 명령)을 wifi 드라이버에 전달하는 과정에서 정리한 것이다. 그림에서 유추할 수 있는 것처럼, `wpa_supplicant`는 wifi driver를 제어하여, 무선 AP(access point)와의 인증(authentication) 및 최종 연결을 확립(association)하는 중요한 역할을 담당한다. 또한 처리 중간 중간에 wifi driver로부터 올라오는 각종 event를 수신하여, 이를 android framework으로 전달해 주기도 한다.

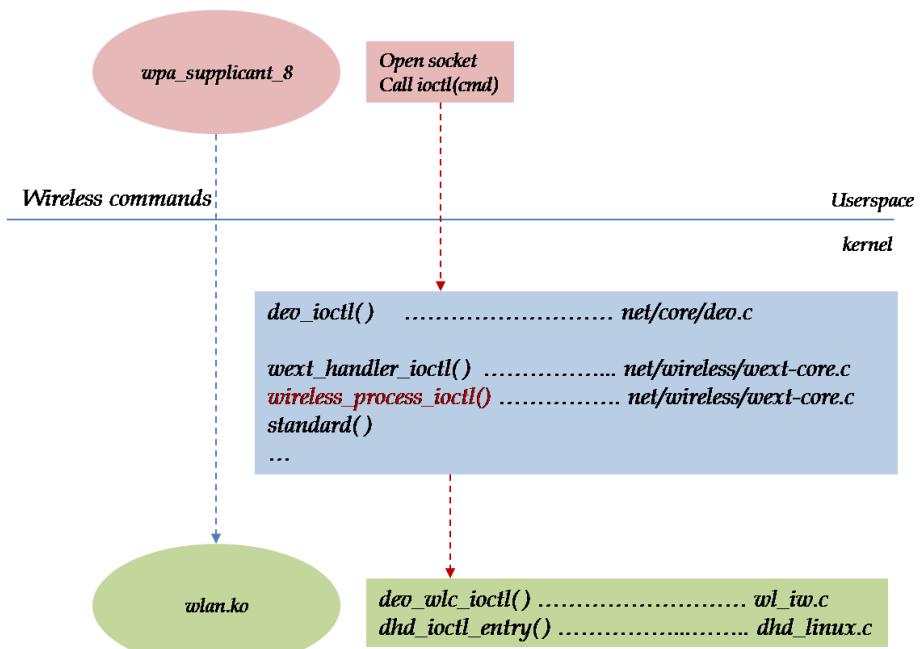


그림 6-57 Wi-Fi 드라이버와 wpa_supplicant와의 통신

그림 6-58은 Wi-Fi HotSpot을 그림으로 표현한 것이다. Wi-Fi HotSpot을 한마디로 설명하면, 내부 망을 Wi-Fi로 하고, 외부 망을 3G 혹은 4G망으로 하여, Smart Phone을 인터넷 공유기처럼 사용하는 것이라고 할 수 있겠다. (당연히) 공유기 기능을 위해서는 Linux에서 지원하는 netfilter(iptables)를 사용한다.

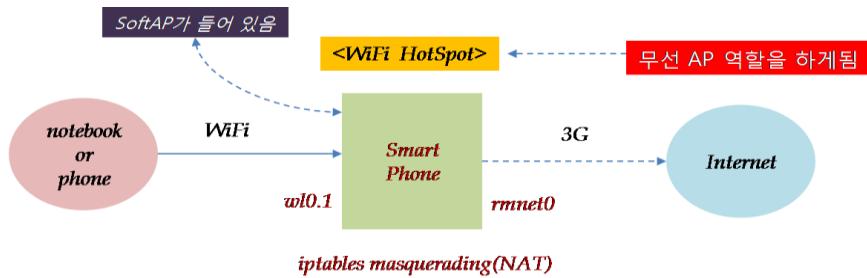


그림 6-58 WiFi Hotspot 개념도

<HotSpot을 위한 iptables 규칙>

```
iptables -t nat -A POSTROUTING -o rmnet0 -j MASQUERADE
```

Wi-Fi도 계속 발전을 거듭하여 최근에는 Bluetooth에서나 가능해 보였던, 1 대 1 통신을 지원한다. Wi-Fi Direct(or P2P라고도 함)로 불리는 이 기법의 의미 및 특징을 간략히 정리해 보면 다음과 같다.

<P2P 기능>

1) P2P Discovery

- 디바이스 발견(device discovery), 서비스 발견(service discovery), 그룹 형성(group formation), P2P 초대(P2P invitation) 등 담당

2) P2P Group Operation

- P2P Group 형성과 종료, P2P 그룹으로의 연결, P2P 그룹 내의 통신, P2P client 발견을 위한 서비스, 지속적 P2P 그룹의 동작 규정

3) P2P Power Management

- P2P 디바이스의 전력 관리 방법과 절전 모드 시점의 신호 처리 방법 규정(P2P GO sleep mode 진입 관련)

4) Managed P2P Device

- 한 개의 P2P 디바이스에서 P2P 그룹을 형성하고, 동시에 WLAN AP에 접속하는 방법 규정

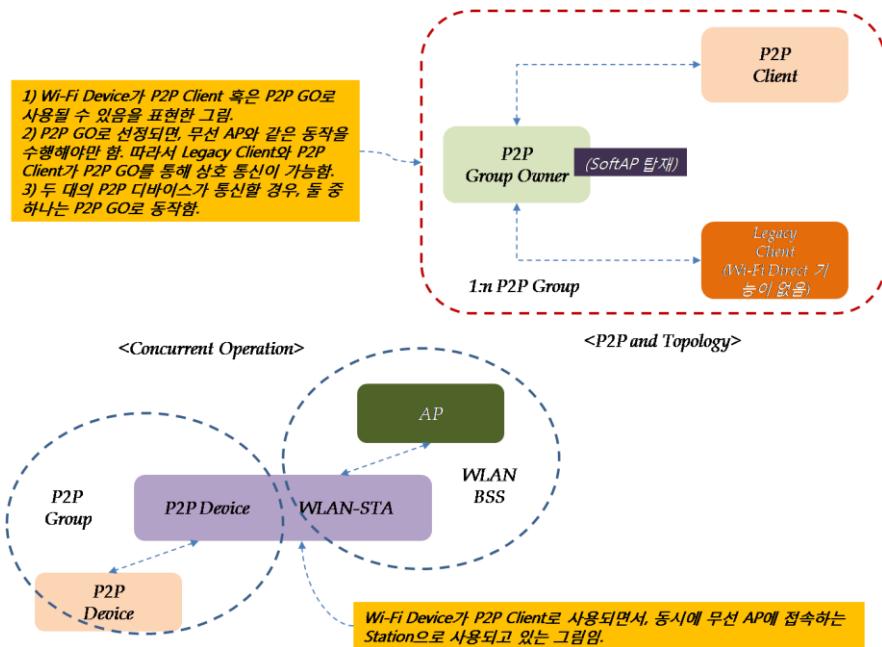


그림 6-59 Wi-Fi Direct 개요

BCM4329 Wi-Fi 드라이버 구현

이번 절에서는 Wi-Fi 장치 연결을 위한 SDIO 서브시스템을 분석해 보고, 실제 Wi-Fi 드라이버를 작성하는 것에 관하여 정리해 보고자 한다.

BCM4329 드라이버는 다시 아래의 4가지 세부 드라이버로 구성되어 있다.

- 1) Platform driver
- 2) net_device(network) driver
- 3) sdio function driver
- 4) sdio bus driver

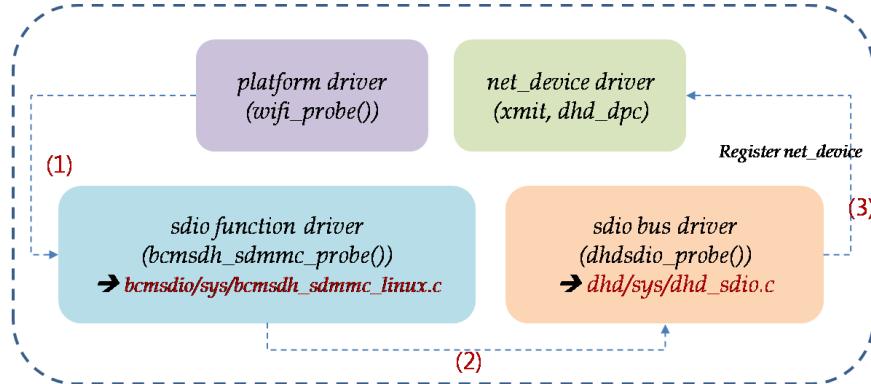


그림 6-60 BCM4329 드라이버 구조(1)

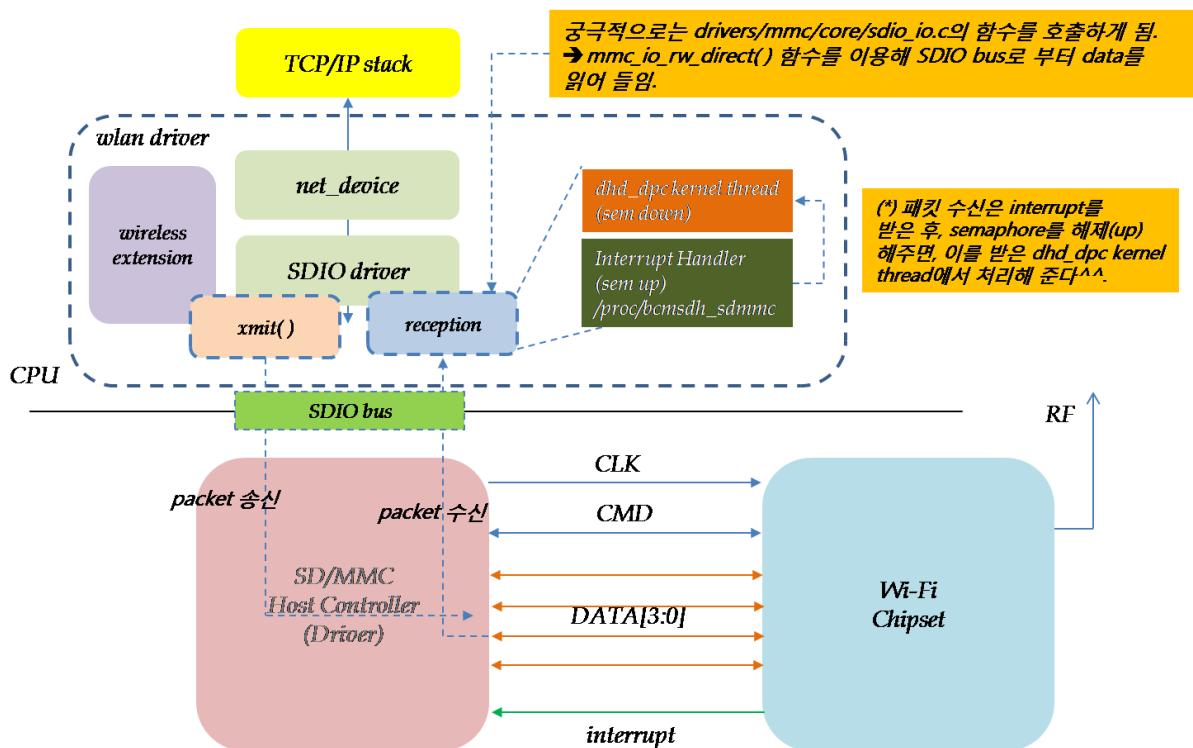


그림 6-61 Wi-Fi 드라이버 구조(2)

지금까지 설명한 내용을 토대로, SDIO 드라이버 관점에서 BCM4329 드라이버를 다시 그려보면 다음과 같다.

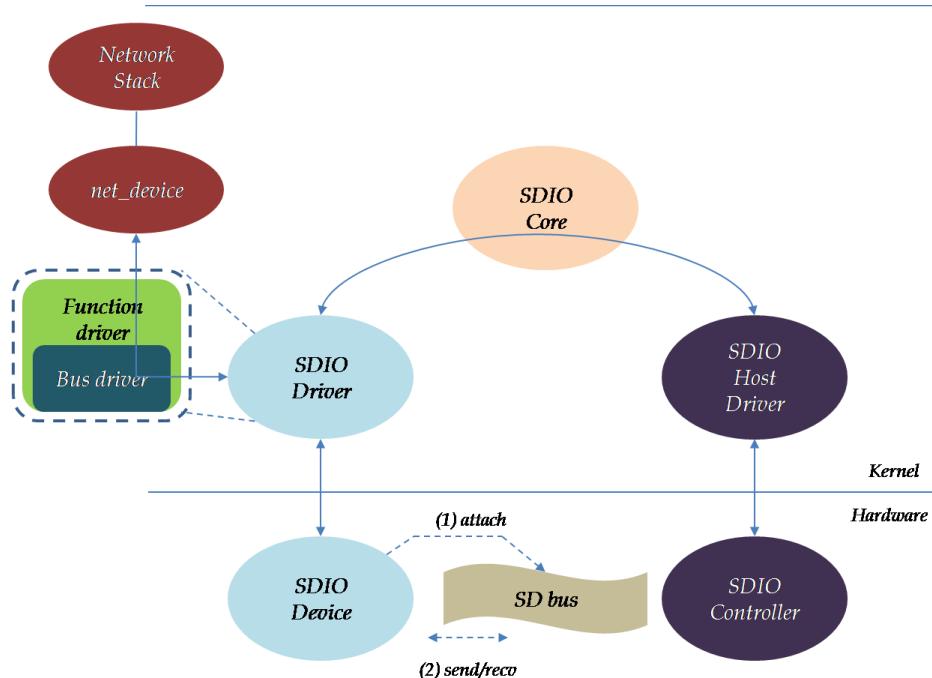


그림 6-62 Wi-Fi 연결을 위한 SDIO 서브시스템

SDIO 드라이버의 작성 절차를 간략히 요약해 보면 다음과 같다.

코드 6-34 초기화 단계

```
static const struct sdio_device_id xxx_ids[] = {
{
    SDIO_DEVICE(0xvendor, 0xproduct)
},
{
    /* end: all zeroes */
},
};

static struct sdio_driver xxx_driver = {
    .probe      = xxx_probe,
    .remove     = xxx_remove,
    .name       = "xxx",
    .id_table   = xxx_ids,
};

static int __init xxx_init(void) {
    [...]
    ret = sdio_register_driver(&xxx_driver);
    [...]
}
```

코드 6-35 Probe와 removal 단계

```
// This is internal driver storage. Not necessary, but useful.
struct xxx_port {
    [...]
    struct sdio_func    *func;
};

static int xxx_probe(struct sdio_func *func, const struct sdio_device_id *id)
{
    struct xxx_port *port;
    port = kzalloc(sizeof(struct xxx_port), GFP_KERNEL);

    port->func = func;
    sdio_set_drvdata(func, port);
```

```

}

static void xxx_remove(struct sdio_func *func)
{
    struct xxx_port *port = sdio_get_drvdata(func);
    port->func = NULL;
}

```

코드 6-36 Start operations 단계

```

sdio_claim_host(port->func);
ret = sdio_enable_func(port->func);
ret = sdio_claim_irq(port->func, xxx_irq);
sdio_release_host(port->func);

```

코드 6-37 Stop operations 단계

```

sdio_claim_host(port->func);
sdio_release_irq(port->func);
sdio_disable_func(port->func);
sdio_release_host(port->func);

```

SDIO bus로 부터 데이터를 읽거나 쓰고자 할 경우, 궁극적으로 아래와 같은 함수 호출 순서를 따르고 있다.

- 1) `sdio_claim_host()`
- 2) `sdio_readb()` or `sdio_writeb()`
 - `drivers/mmc/core/sdio_io.c` // 있는 함수
- 3) `sdio_release_host()`

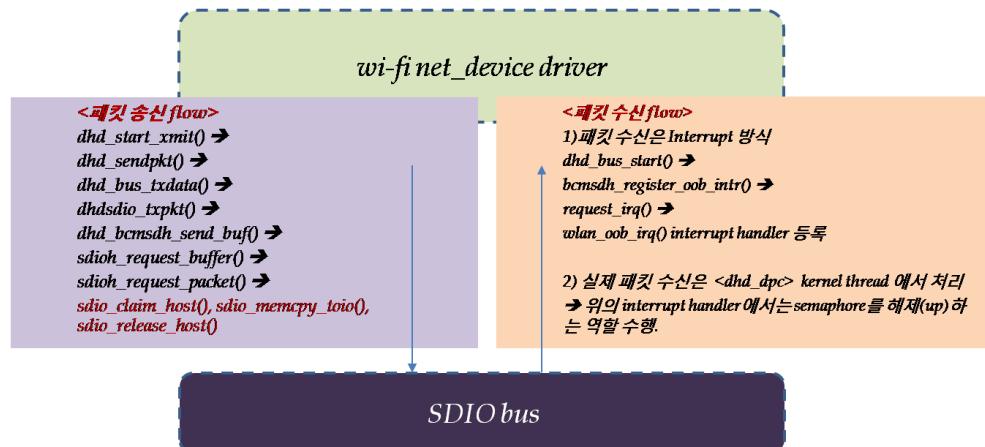


그림 6-63 BCM4329 드라이버 패킷 송수신 개요

코드 6-38 Actual I/O 단계

```
extern u8 sdio_readb(struct sdio_func *func, unsigned int addr, int *err_ret);
extern u16 sdio_readw(struct sdio_func *func, unsigned int addr, int *err_ret);
extern u32 sdio_readl(struct sdio_func *func, unsigned int addr, int *err_ret);

extern int sdio_memcpy_fromio(struct sdio_func *func, void *dst, unsigned int addr, int count);
extern int sdio_readsbs(struct sdio_func *func, void *dst, unsigned int addr, int count);

extern void sdio_writeb(struct sdio_func *func, u8 b, unsigned int addr, int *err_ret);
extern void sdio_writew(struct sdio_func *func, u16 b, unsigned int addr, int *err_ret);
extern void sdio_writel(struct sdio_func *func, u32 b, unsigned int addr, int *err_ret);

extern u8 sdio_writeb_readb(struct sdio_func *func, u8 write_byte, unsigned int addr, int *err_ret);

extern int sdio_memcpy_toio(struct sdio_func *func, unsigned int addr, void *src, int count);
extern int sdio_writesbs(struct sdio_func *func, unsigned int addr, void *src, int count);

extern unsigned char sdio_f0_readb(struct sdio_func *func, unsigned int addr, int *err_ret);
extern void sdio_f0_writeb(struct sdio_func *func, unsigned char b, unsigned int addr, int *err_ret);
[...]
```

Wi-Fi 드라이버 예제

코드 6-39 BCM4329 Init 코드

```
static struct platform_driver wifi_device = {(1)
    .probe      = wifi_probe,
    .remove     = wifi_remove,
    .suspend    = wifi_suspend,
    .resume     = wifi_resume,
    .driver     = {
        .name   = "bcm4329_wlan",
    }
};

struct sdio_driver bcmsdh_sdmmc_driver = { /* function driver */(2)
    .probe      = bcmsdh_sdmmc_probe,
```

```

    .remove    = bcmsdh_sdmmc_remove,
    .name      = "bcmsdh_sdmmc",
    .id_table  = bcmsdh_sdmmc_ids,
};

bcmsdh_driver_t dhd_sdio = { (3)
    dhdsdio_probe,
    dhdsdio_disconnect
};

<bm4329 module_init>
1) gpio를 이용하여 wi-fi chipset을 on한다.

2) platform_driver로 등록한다. (1)

3) SDIO/MMC driver로 등록한다.

    3-1) dhd_bus_register( )
    3-2) bcmsdh_register( )
    3-3) sdio_function_init( )
    3-4) sdio_register_driver( )   ↪ drivers/mmc/core/sdio_bus.c
        ● sdio function driver로 등록함. (2)
        ● bcmsdh_sdmmc_probe() 함수가 이 sdio function driver의 probe 함수임.

```

코드 6-40 bcmsdh_sdmmc_probe() 함수 흐름

```

struct sdio_driver bcmsdh_sdmmc_driver = {
    .probe      = bcmsdh_sdmmc_probe,
    .remove     = bcmsdh_sdmmc_remove,
    .name       = "bcmsdh_sdmmc",
    .id_table   = bcmsdh_sdmmc_ids,
};

1) bcmsdh_probe
2-1) bcmsdh_attach
2-2) drvinfo.attach (= dhdsdio_probe)
    ■ dhd_bus_register() 함수 안에서 아래 함수 호출 시, argument로 dhd_sdio가 전달되고, 이
      것의 probe 함수가 여기서 다시 호출 됨.
    ■ bcmsdh_register(&dhd_sdio); (3)

3) sdioh_attach  ↪ 2-1) bcmsdh_attach() 가 호출함.
    ■ sdioh_sdmmc_osinit(), sdio_set_block_size(), sdioh_sdmmc_card_enablefuncs() 함수 등

```

출하고 마무리

주-1) 이 probe 함수에서는 sdio function driver 형태로 동작하기 위한 기본 처리 작업을 진행하고, 나머지는 sdio bus driver를 등록시켜 주는 역할을 수행함.

주-2) sdio function driver는 실제 SDIO 장치에 data를 read/write하기 위한 용도로 사용됨.

코드 6-41 dhdsdio_probe() 함수 흐름

```
bcmisdh_driver_t dhd_sdio = {  
    dhdsdio_probe, /* 이 함수가 실제로 sdio bus에 attach 시키는 루틴임 */  
    dhdsdio_disconnect  
};
```

1-1) dhd_common_init

- firmware path 초기화

1-2) dhdsdio_probe_attach

- dongle에 attach 시도

1-3) *dhd_attach

- dhd driver의 main routine
- dhd/OS/network interface에 attach
- dhd_info_t data structure 변수 선언 및 초기화
- net_device 추가(dhd_add_if)
- dhd_prot_attach() 함수 호출
- dhd_watchdog_thread kernel thread 생성
- dhd_dpc_thread kernel thread 생성
 - frame 송수신 관련 thread로 보임
- _dhd_sysioc_thread kernel thread 생성
- 몇 개의 wakelock 생성
 - dhd_wake_lock, dhd_wake_lock_link_dw_event,
 - dhd_wake_lock_link_pno_find_event

1-4) dhdsdio_probe_init

1-5) bcmisdh_intr_reg

1-6) *dhd_bus_start

- firmware download 하고, bus를 start 시킨다.
- 이 안에서 bcmisdh_register_oob_intr() 함수 호출하여 request_irq() interrupt handler 등록

1-7) *dhd_net_attach

- register_netdev() 호출하여, net_device로 등록함.

8. Bluetooth 드라이버

이번 장에서 살펴볼 내용은 근거리 통신의 대표 주자인 블루투스이다. 블루투스는 Wi-Fi, NFC 등과의 경쟁에서 살아남기 위해 현재도 계속 진화를 거듭하고 있는데, 버전별 특징을 간략히 살펴보면 다음과 같다.

<Bluetooth 발전 과정>

1) BR 1.1 (2002)

- Basic Rate (1 Mbit/s)

2) EDR 2.0 (2004)

- Enhanced Data Rate (2 and 3 Mbit/s)
- 2.4 GHz ISM 밴드, 1M symbols/s, GFSK, 4PSK, 8PSK 1MHz 채널 공간(Wi-Fi와 공존)

3) HS 3.0 (2009)

- 고속(Alternate MAC/PHY = AMP)
- Bluetooth 3.0 AMP(Alternate MAC/PHY)는 추가 radio 즉, IEEE 802.11n을 사용한다. 이는 Wi-Fi에서 속도를 향상시킨 것과 동일한 내용에 해당한다.

4) LE 4.0 (2010)

- Low Energy (1 Mbit/s ultra low power)
- Bluetooth 4.0 버전은 3.0 버전을 계승(속도 향상)했다가 보다는, 저전력에 초점을 맞춘 새로운 표준으로 보아야 한다.

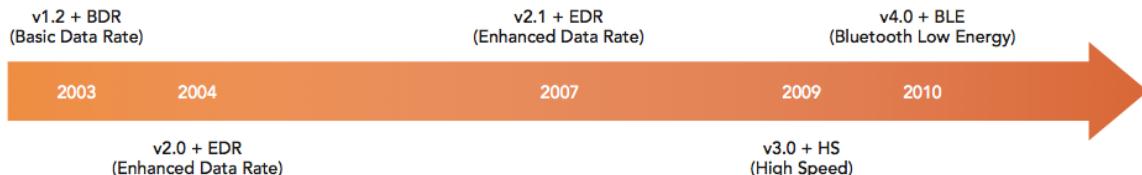


그림 6-64 블루투스 변천사

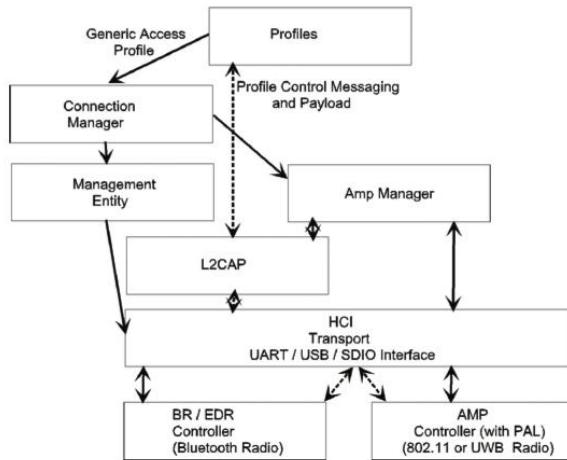


그림 6-65 블루투스 3.0 HS 아키텍처 [출처 - 참고 문헌]

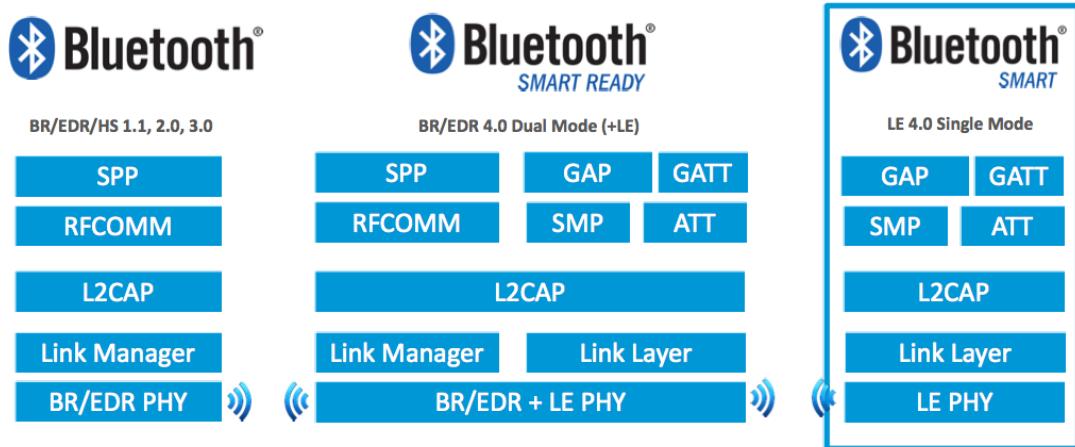


그림 6-66 블루투스 4.0 LE(Low Energy) 개요 [출처 - 참고 문헌]

<블루투스 4.0 - 저전력(BLE : Bluetooth Low Energy)>

블루투스 4.0은 전송속도가 1Mbps로 다소 느린 대신 전력 소비량을 최대 10분의 1 수준으로 줄인 새로운 기술이다. 기존 3.0+HS(High Speed, 최대속도 24Mbps)의 업그레이드 버전이 아닌 서로 다른 특징을 갖는 새로운 규격으로 저전력의 장점을 이용해 모바일 헬스, 스포츠, 보안, 홈 엔터테인먼트 등 여러 분야에서 이용된다. 특히 저전력 기술 이전의 전력소모량 때문에 짧은 시간 밖에 이용 못하던 각종 입력장치(마우스, 키보드, 트랙패드 등)를 더욱 더 장기간 이용할 수 있게 되었으며, 만보계나 혈당 체크기 등 장시간 몸에 지니고 다녀야 하는 휴대용 기기라도 배터리 걱정없이 이용이 가능하게 되었다.

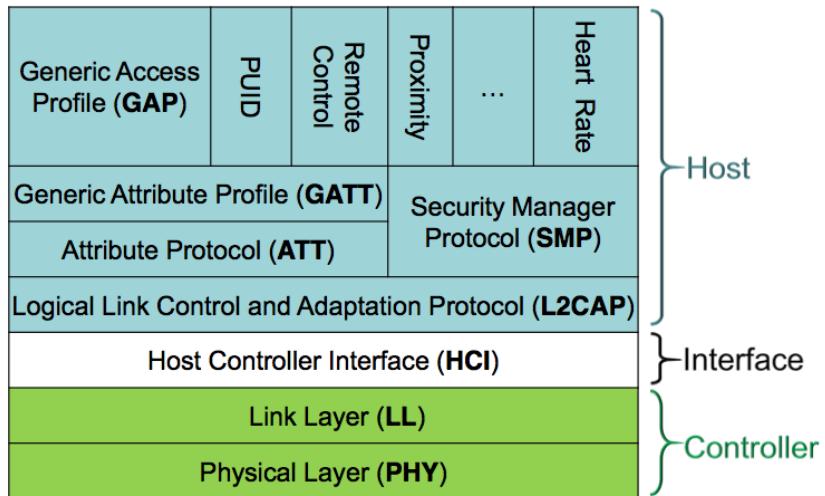


그림 6-67 블루투스 4.0(Bluetooth Low Energy) 아키텍처 [출처 - 참고 문헌]

BlueZ 기반의 블루투스 S/W 스택

앞서 소개한 블루투스 스펙의 변화 못지 않게, 안드로이드에서 사용하는 블루투스 아키텍처도 계속 변화하고 있는데, 먼저 안드로이드 4.1까지 사용되던 BlueZ 기반의 블루투스 S/W 스택을 살펴보기로 하자.

<BlueZ 기반의 안드로이드 블루투스 스택 구조>

1) 블루투스 리눅스 커널 드라이버(BlueZ 커널 스택)

- HCI, L2CAP, SCO, RFCOMM 등으로 구성되어 있음.

2) bluetoothd 데몬 프로세스

- dbus를 통해 프레임워크로부터 명령 혹은 오디오 데이터를 수신하여 블루투스 커널로 전달해주는 역할 수행
- 또한, 블루투스 커널에서 올라온 각종 이벤트를 프레임워크로 전달해 주는 역할도 수행 함.

3) dbus-daemon

- Domain socket을 기반으로 설계된 process간 통신 프로토콜
- Linux desktop 환경에서 (거의) 통신 표준임

4) JNI

5) Java framework

- Bluetooth Device, Bluetooth A2DP, Bluetooth Event Loop 등으로 구성됨.
- 아래 그림에서는 system_server로 표기되어 있음.

6) 다양한 블루투스 applications

- HSP, AVRCP, A2DP 등 관련한 media player, HFP 등 관련한 phone appl, OPP/OBEX 등 관련한 appl 등

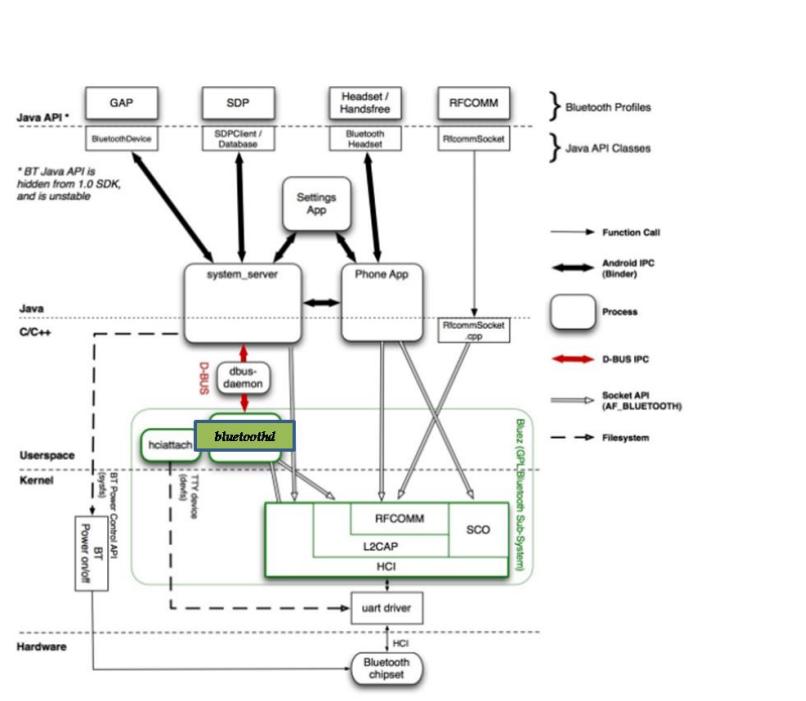


그림 6-68 BlueZ 기반의 안드로이드 블루투스 스택 구조 [출처 - 참고 문헌]

Bluedroid 기반의 블루투스 S/W 스택

Bluedroid는 브로드컴에서 개발한 내용으로 안드로이드 4.2(Jelly Bean)에서부터 추가되기 시작하였다. 기존의 BlueZ 기반의 블루투스 스택과 비교해 볼 때, bluetoothd(dbus-daemon 경유)가 제거되고, HAL의 기능을 확장하여 BluetoothService에서 직접 HAL을 사용하도록 구조가 단순화되었다. dbus-daemon, bluetoothd를 사용하던 기존의 구조를 경험해 본 독자라면, bluedroid의 구조가 훨씬 단순하고, 문제 발생의 소지를 줄일 수 있는 방법임을 쉽게 짐작할 수 있을 것이다. 솔직히 기존의 방식은 데스크탑에서 사용하던 BlueZ 스택을 안드로이드 프레임워크에 구겨 넣은 듯한 인상을 주었었는데, Bluedroid 기반으로 가면서 이러한 부분이 깔끔하게 정리가 되었다고 볼 수 있다.

아래 그림은 Bluedroid 기반의 블루투스 스택을 그림으로 정리한 것이다.

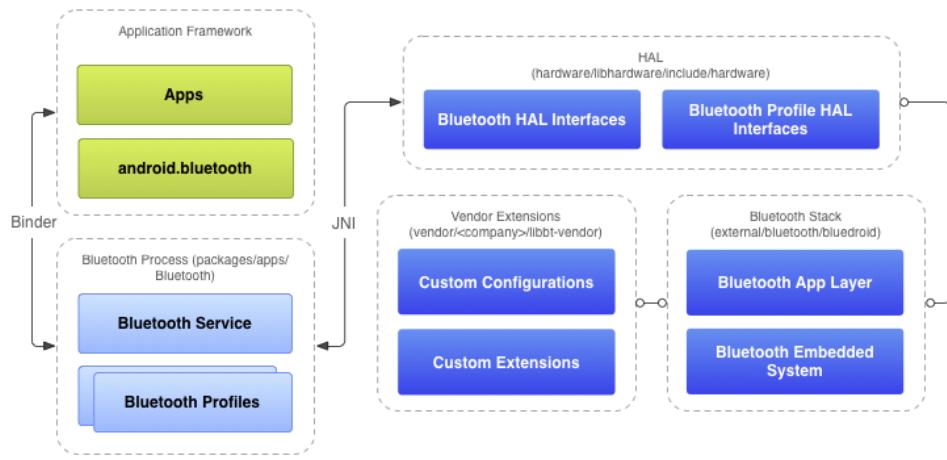


그림 6-69 Bluedroid 기반의 안드로이드 블루투스 스택(1) [출처 - 참고 문헌]

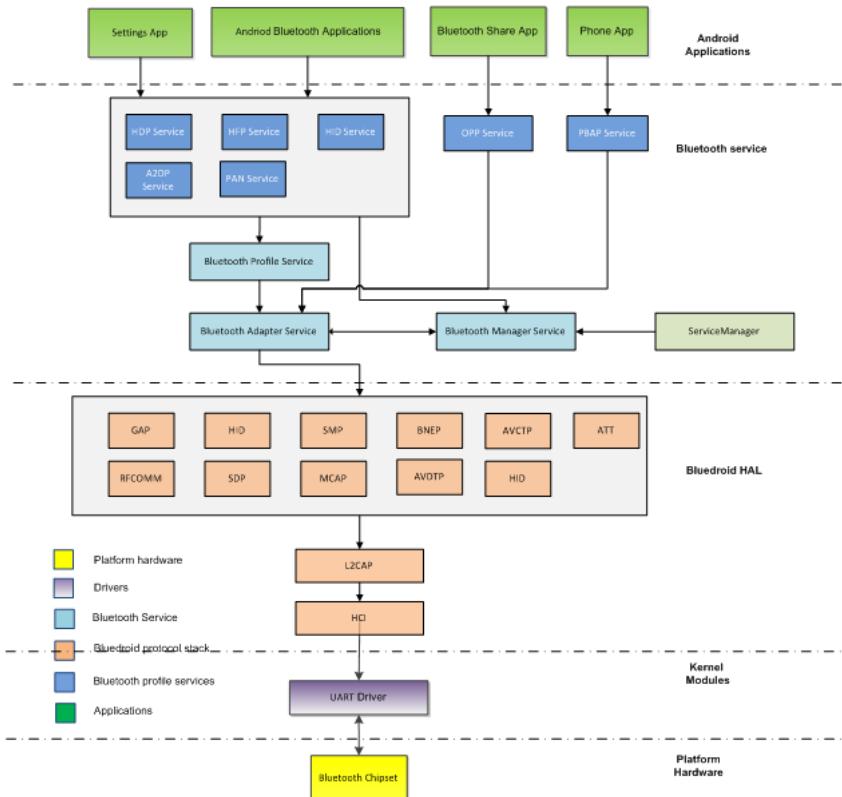


그림 6-70 Bluedroid 기반의 안드로이드 블루투스 스택(2) [출처 - 참고 문헌]

BlueZ 및 Bluedroid 환경에서 공통으로 사용되는 Bluetooth 커널 모듈을 정리해 보면 다음과 같다.

<Bluetooth 커널 모듈>

1) *hci_uart.ko*

- H4 UART 드라이버

- 이 밖에도 USB, PCI, SDIO 등의 인터페이스를 사용할 수 있음.

2) *sco.ko*

- SCO audio 드라이버(phone 음성 송/수신시 사용)

3) *l2cap.ko*

- Logical Link Control and Adaptation Protocol 드라이버 (Session 계층)

4) *rfcomm.ko*

- Radio Frequency Communication 드라이버 (Presentation 계층)

5) *bnep.ko*

- Network Encapsulation 드라이버(TCP/IP와 연결됨)

6) *hidp.ko*

- Human Interface Device 드라이버(mouse, keyboard 등과 연관)

7) *bluetooth.ko*

- Core module

블루투스 프로파일(Profile) 소개

블루투스 프로파일은 블루투스를 사용하는 응용 계층 프로그램간의 통신 스펙으로 설명할 수 있다. 이말은 블루투스 프로파일을 준수하여 S/W를 구현하여야만 상호간 통신에 문제가 없음을 뜻 한다. 전통적인 방식에서부터 최근 4.0 버전에서 새롭게 추가된 내용까지 매우 다양한 프로파일이 존재함을 알 수 있다.

<Bluetooth Profile>

1) *SPP/SDAP*

- 기본 profile & BT 장치 검색 관련
- SPP: Serial Port Profile(DUN, FAX, HSP, AVRCP의 기초가 되는 profile임)
- SDAP: Service Discovery Application Profile

2) *HSP/HFP*

- phone app/headset과 연관됨
- HSP: Headset Profile
- HFP: Hands-Free Profile
- Headset을 사용하여 음악과 전화를 ...

3) *GAVDP/AVRCP/A2DP*

- media player app과 연관됨
- GAVDP: Generic Audio/Video Distribution Profile
- AVRCP: AV Remote Control Profile
- A2DP: Advanced Audio Distribution Profile
- Audio/Video playback 제어

4) OPP

- 파일 전송과 연관됨
- OPP: Object Push Profile
- File 전송 관련 ... OBEX와 연관됨

5) PBAP

- 전화번호부 전송 관련
- PBAP: Phone Book Access Profile
- Phone book object 교환(car kit <-> mobile phone)

6) HID

- BT 무선 키보드/마우스등 관련
- Human Interface Device profile
- Bluetooth keyboard/mouse/joypad ...

7) DUN

- BT로 인터넷 사용 관련
- DUN: Dial-up Networking Profile
- PC -> Phone -> Internet !!!

8) GAP

- Generic Access Profile

9) GATT

- Generic Attribute Profile

10) HID over GATT: 무선 휴먼 인터페이스 장치

- 호스트 (PC, tablet, phone)
- 장치 (keyboard, mouse, trackpad, ...)

11) Heart Rate: 스포츠 및 의료 심박수 전송

- 수집기 (PC, tablet, phone)
- 센서 (Heart Rate belt 등)

12) Proximity / Find Me: 장치 존재 감지

- Monitor (PC, tablet, phone)
- Reporter (keyfob, phone)

...

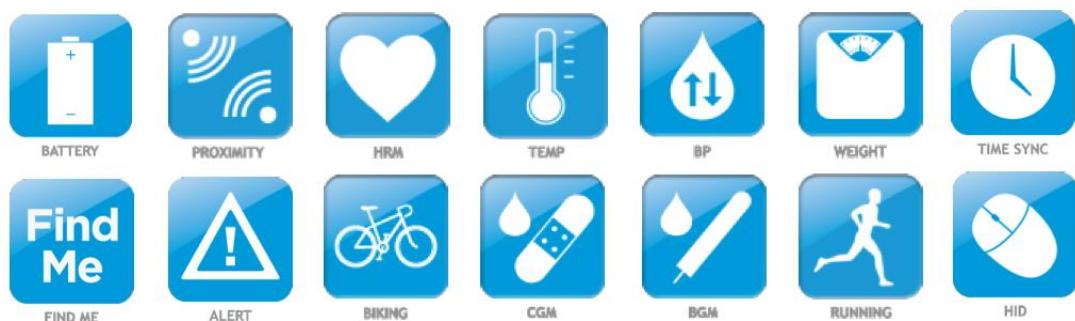


그림 6-71 블루투스 4.0 프로파일 모음 [출처 - 참고 문헌]

블루투스 HCI & Power On/Off

블루투스 드라이버에서 커널 프로그래머가 신경써야 할 부분은 블루투스 장치와 CPU 간을 연결해주는 HCI(Host Control Interface)와 관련된 부분일 것이다. HCI에서 담당하는 일을 정리하면 다음과 같다.

<HCI의 역할>

- 1) TX/RX 패킷 송수신
- 2) UART, USB, SDIO 등의 인터페이스 연결
- 3) Power On/Off
- 4) ...

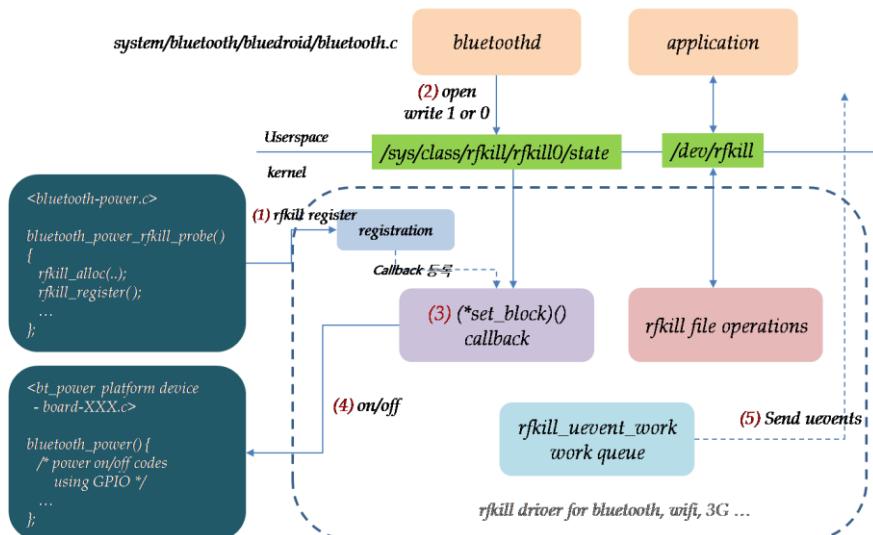


그림 6-72 Bluetooth 파워 드라이버

<rfkill driver 개요 및 flow>

0) Wi-Fi, Bluetooth, 3G 등 전파 송수신 장치들에 대해 질의하고, 활성화하고, 비활성화할 수 있도록 해주는 interface를 제공해주는 kernel subsystem을 rfkill 드라이버라고 함.

- net/rfkill 디렉토리에 코드 있음.

1) rfkill을 사용하고자 하는 드라이버는 driver probe 단계에서 자신을 rfkill driver로 등록해 주어야 한다.

- 등록 단계에서 자신을 power on 혹은 off할 수 있는 callback 함수를 등록해 주어야 함(이때 rfkill_alloc() 함수를 사용함).
- bluetooth의 경우는 board-XXX.c 파일에 bt_power_platform_device를 선언하면서 등록한 함수(platform_data = &bluetooth_power)를 위의 callback 함수로 사용하고 있음.

2) Application에서는 "/sys/class/rfkill/rfkillX/state" 파일을 open 한 후, power on의 경우 1을, power off의 경우 0을 write해 준다.

3) rfkill driver는 2)에서 요청한 값을 토대로, 1)에서 기 등록한 callback 함수(*set_block)를 호출해

준다.

- 4) 3) 단계에 의해, 1)에서 등록한 실제 드라이버의 *power control* 함수가 호출되어, 실제 전파 송수신 장치(wi-fi, bluetooth, 3G ...)의 *power*가 *on* 혹은 *off* 된다.
- 5) 상태 변화가 있을 때마다 이를 uevent 형태로 application에 알린다.

코드 6-42

<TODO>

Drivers/Bluetooth/btwilink.c

Net/Bluetooth/hci_core.c

9. NFC(Near Field Communication) 드라이버

NFC 장치 소개

NFC는 13.56MHz contactless 표준을 기반으로 하는 통신 기술로써, Near Field Communication을 뜻한다. 13.56MHz의 기존 표준을 포괄(ISO 14443A/B, Sony Felica)하며, 106, 212 및 424Kbps 데이터 전송이 가능하다. 또한 3가지의 동작 모드로 구분할 수가 있는데, 이를 정리하면 다음과 같다.

<NFC 3가지 동작 모드>

1) Card Emulation mode

- Google Wallet이 필요함.

2) Reader Writer mode

- Gingerbread 버전부터 들어간 기능. Handset을 reader 혹은 writer로도 사용 가능하도록 해 주는 기능(RFID와 차이점이기도 함).

3) Peer-2-Peer mode

- ICS에 추가된 Android Beam으로 알려진 기능(ISO 18092)
- 상호 데이터 교환 가능.



그림 6-73 NFC 3가지 동작 모드 [출처 - 참고 문헌]

다음 그림은 NFC 칩이 장착된 스마트 폰의 내부구조를 표현한 것이다. 이 그림을 통해 몇 가지 정보를 얻을 수 있는데, 이를 정리해 보면 다음과 같다.

<NFC 칩>

1) AP(Application Processor)와 NFC 칩은 UART, I₂C, SPI 등의 인터페이스를 통해 상호 연결된다.

- HCI, NCI

2) AP는 ISO-7816 표준을 따르며 UICC, eSE, microSD 등의 Secure Element와 통신한다. 또한 NFC 칩과 Secure Element 간에는 SWP/S2C를 기준으로 통신한다.

- Secure Element는 중요 정보를 안전하게 저장하는 장치로 보면 됨.

3) NFC 칩은 ISO-14443을 준수하며, 13.56MHz 주파수를 사용한다.

4) NFC 칩은 Tag로부터 최대 4cm 내에서만 동작 가능하다.

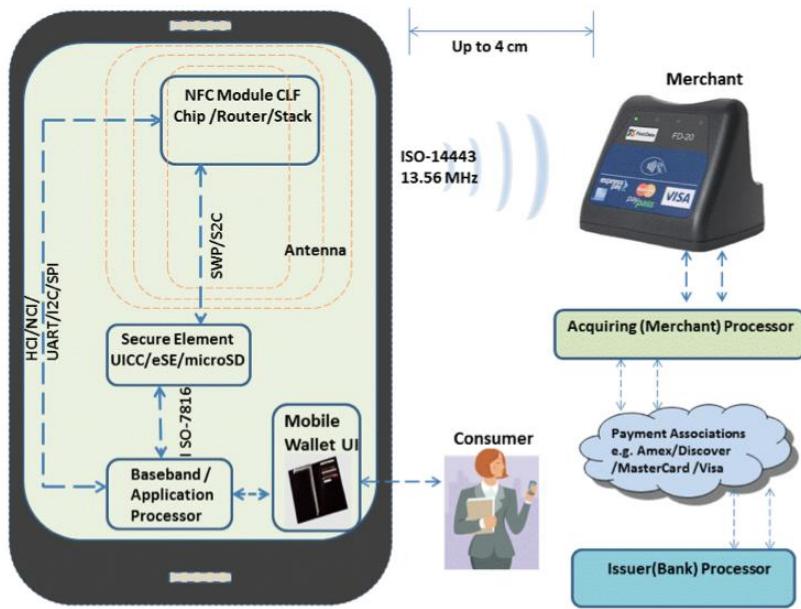


그림 6-74 NFC Phone 구조 [출처 - 참고 문헌]

안드로이드 NFC 프레임워크 소개

블루투스와 유사하게 NFC의 경우도 매우 과도기적인 상태에 있든 듯 보인다. 현재 두가지의 존재하는데, 이를 정리해 보면 다음과 같다.

1)

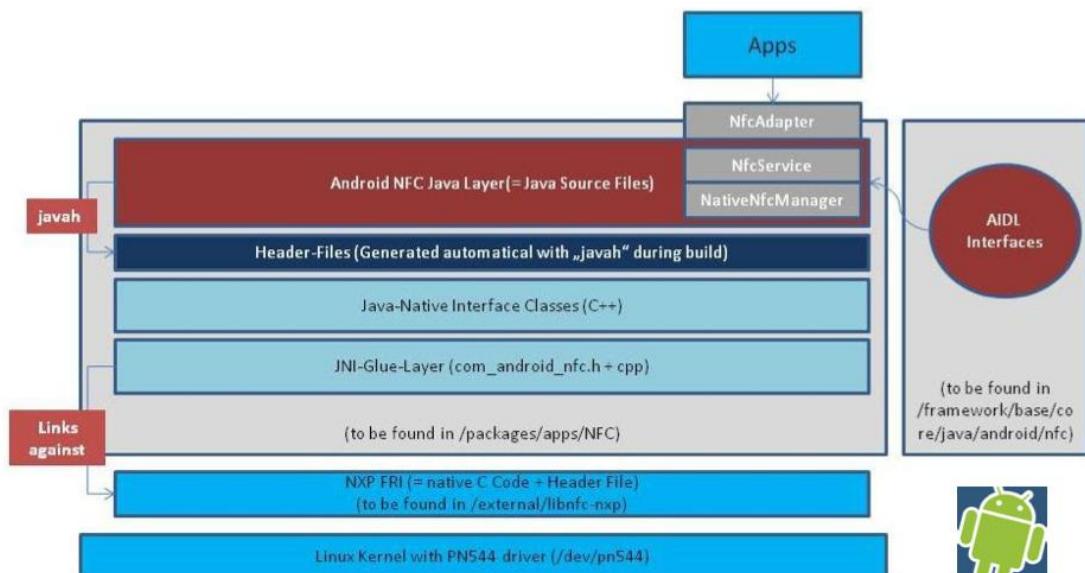


그림 6-75 Android NFC 아키텍처 – libnfc-nxp 기반

리눅스 NFC 프레임워크 소개

overall picture

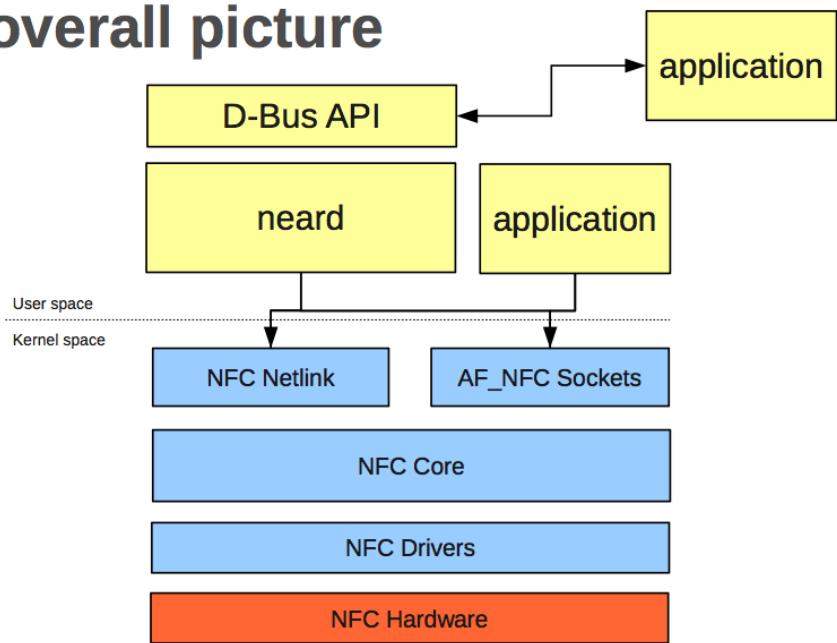


그림 6-76 Linux 공식 NFC 아키텍처

<NDEF 개요>

- 1) NDEF은 NFC Data Exchange Format의 줄임말이다.
- 2) NDEF을 통해 응용 프로그램에서 정의한 임의의 형식 및 크기를 가진 데이터(payload)를 한 개의 단일화된 틀에 넣어 전송하는 것이 가능해 진다.
- 3) 각각의 payload는 형(type), 길이, 선택 가능한 구분자(optional identifier)로 구성된다.
- 4) NFC Tag Read/Write 및 P2P (LLCP)를 사용할 때, NDEF를 교환하는 것이 가능하다.

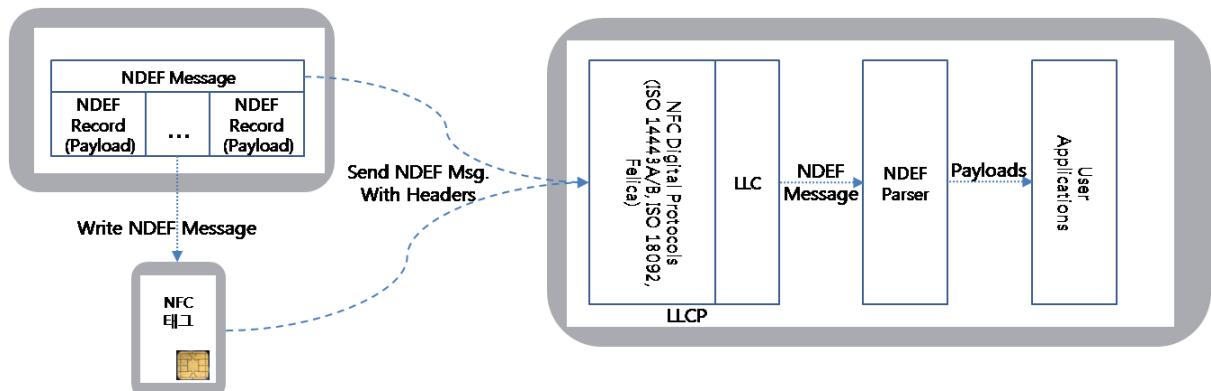


그림 6-77 NDEF 메시지 처리

NFC 드라이버 예제

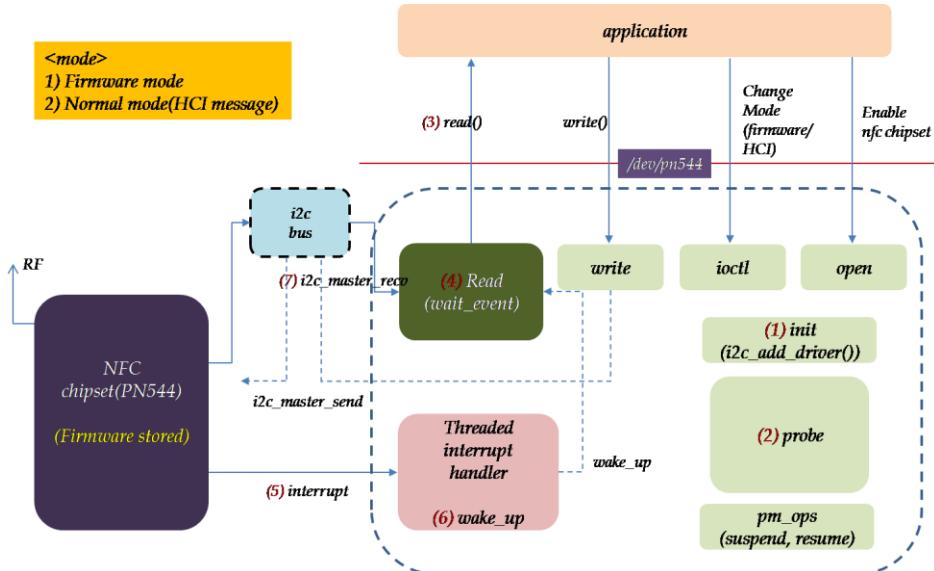


그림 6-78 NFC 드라이버 개관

<NFC 드라이버 코드 흐름 분석>

1) *pn544 NFC driver*는 *i2c client driver*이다. 따라서 *pn544_init()* 함수에서 *i2c_add_driver()* 함수를 호출하여 *i2c client*로 등록한다.

2) *pn544_probe()* 함수

- *pn544 driver*에서 사용하는 *pn544_info* data structure를 위한 buffer를 할당하고, 각각의 field를 초기화 한다.
- 'read_wait' wait queue를 초기화한다(read 함수 <-> interrupt handler 간의 sync를 맞추기 위해 사용됨)
- *i2c_set_clientdata()* 함수를 호출하여 *i2c client*를 위한 정보를 초기화한다.
- nfc 장치로부터 들어오는 interrupt 요청을 받아 처리하는 interrupt handler를 등록한다. *pn544 driver*는 이를 위해 특별히 *threaded_interrupt_handler* 형태로 등록하고 있음.
- 테스트 목적으로 sysfs에 파일을 생성함(*pn544_attr*).
- misc driver 형태로 자신을 등록함.

3) *application*에서 *read* 함수를 호출할 경우, *nfc chip*으로부터 *i2c_master_recv()* 함수를 사용하여 *data*를 읽어 들인다.

- 단, 이때 *nfc chip*에 읽어 들일 *data*가 준비되지 않았을 수 있으므로, 읽기 작업 수행 전에 *wait_event_interruptible()* 함수를 호출하여 대기 상태로 들어간다.
- *probe* 함수에서 등록한 interrupt handler routine에서는 interrupt가 발생(*nfc chip*으로 부

터 data 수신 가능 의미)할 경우, `wake_up_interruptible()` 함수를 호출하여, 대기 상태로 빠진 read 작업이 재개될 수 있도록 만들어 준다.

4) Application에서는 `ioctl` 함수를 사용하여, nfc chipset의 동작 방식을 *firmware update mode*와 *normal mode(HCI mode)*로 바꾸어 준다.

- firmware update mode에서는 application에서 read 혹은 write 함수 호출시 firmware read 및 write 관련 작업이 수행되며,
- normal HCI mode에서는 HCI message에 대한 송/수신이 가능하게 된다. HCI message(8bit header + body)는 최대 33bytes이며, firmware message의 최대 길이는 1024bytes이다.

5) 자세한 것은 알 수 없으나, 무선 통신은 nfc chip 자체에서 수행하며, 이를 담당하는 firmware 를 user application에서 교체할 수 있는 것으로 보인다.

코드 6-43

```
<TODO>
```

10. 가상 이더넷 드라이버 – 3G, 4G

net_device 소개

```

struct net_device {
    /* 
     * This is the first field of the "visible" part of this
     * structure
     * (i.e. as seen by users in the "Space.c" file). It is the
     * name
     * of the interface.
     */
    char      name[IFNAMSIZ];
    struct pm_qos_request_list pm_qos_req;
    /* device name hash chain */
    struct hlist_node name_hlist;
    /* smp alias */
    char      *alias;
    ...
}

struct net_device_ops {
    int      (*ndo_init)(struct net_device *dev);
    void    (*ndo_uninit)(struct net_device *dev);
    int      (*ndo_open)(struct net_device *dev);
    int      (*ndo_stop)(struct net_device *dev);
    netdev_tx_t (*ndo_start_xmit)(struct sk_buff *skb,
                                  struct net_device *dev);
    u16    (*ndo_select_queue)(struct net_device *dev,
                               struct sk_buff *skb);
    void    (*ndo_change_rx_flags)(struct net_device
                                   *dev,
                                   int flags);
    void    (*ndo_set_rx_mode)(struct net_device
                               *dev);
    void    (*ndo_set_multicast_list)(struct net_device *dev);
    ...
}

struct sk_buff {
    /* These two members must be first. */
    struct sk_buff    *next;
    struct sk_buff    *prev;
    ktime_t        timestamp;
    struct sock      *sk;
    struct net_device *dev;
    char      cb[48] __aligned(8);
    unsigned long    skb_refdst;
    ...
}

```

그림 6-79 net_device 개요

<net_device 주요 API>

`int register_netdev(struct net_device *);`

- network device를 구동시킨다.

`void unregister_netdev(struct net_device *);`

- network device를 내린다.

`struct net_device *alloc_netdev(int sizeof_priv, const char *name, void (*setup)(struct net_device *));`

- netdevice data structure를 위한 버퍼를 동적으로 할당한다.
- **sizeof_priv**: priv data field의 크기
- **name**: device 명. 예: mynet0, mynet1 ...
- **setup**: net_device data structure의 나머지 field를 초기화시키기 위해 사용하는 initialization 함수

`void free_netdev(struct net_device *dev);`

- 할당된 netdevice data structure 버퍼를 해제한다.

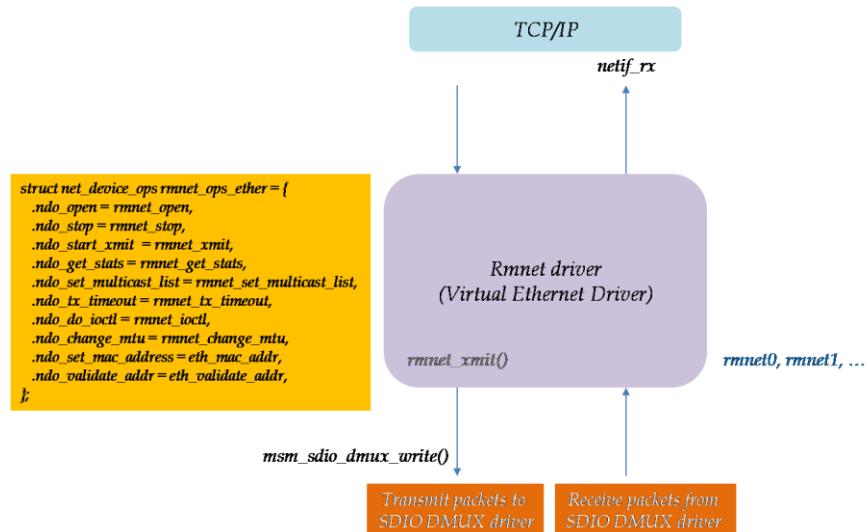


그림 6-80 가상 이더넷 드라이버 개관(1)

<rmnet_init() 함수 분석>

0) 변수 선언

- `struct device *d;`
- `struct net_device *dev;`
- `struct rmnet_private *p;`

1) `alloc_netdev()` 함수 호출하여 `net_device` 할당.

- 이 줄을 포함하여 아래 step을 RMNET_DEVICE_COUNT(=8) 만큼 반복!

2) `rmnet_private pointer(p)` 및 초기화 및 몇개의 field 및 채움.

3) tasklet 하나 초기화

- `_rmnet_resume_flow()` 함수가 나중에 호출될 것임.

4) `wake_lock_init`

5) `completion` 초기화

6) `p->pdrv.probe = msm_rmnet_smd_probe;`

7) `ret = platform_driver_register(&p->pdrv);`

8) `ret = register_netdev(dev);`

9) sysfs entry 생성

- `device_create_file(d, &dev_attr_timeout)`
- `device_create_file(d, &dev_attr_wakeups_xmit)`
- `device_create_file(d, &dev_attr_wakeups_rcv)`
- `device_create_file(d, &dev_attr_timeout_suspend)`

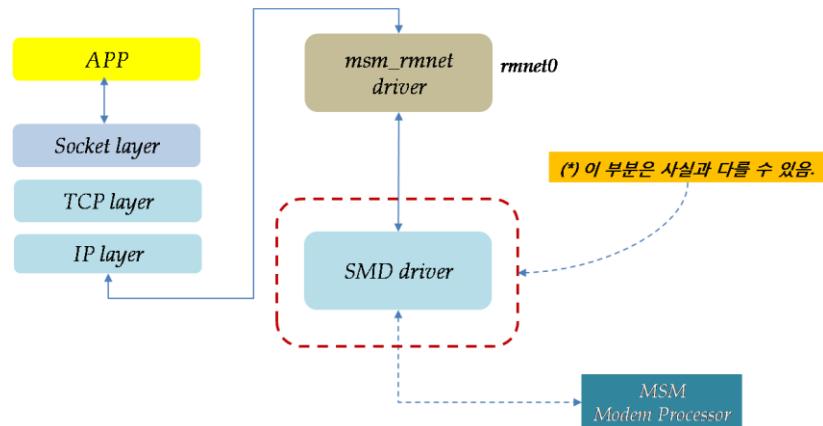


그림 6-81 가상 이더넷 드라이버 개관(2)

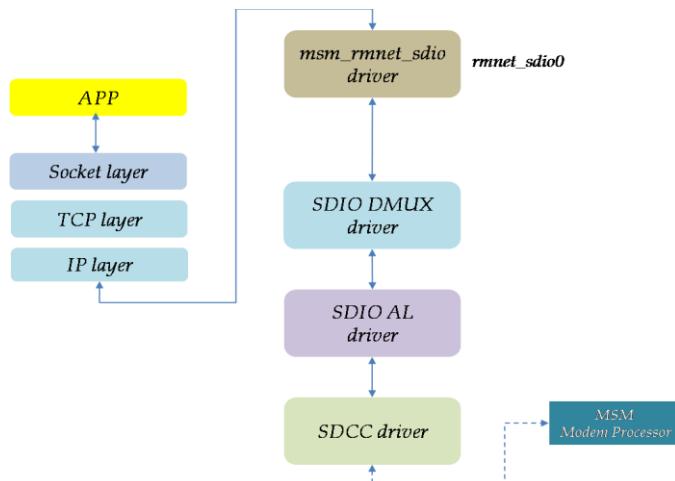


그림 6-82 가상 이더넷 드라이버 개관(3)

11. 그 밖의 중요한 주제들

지면 관계상 본서에서 자세히 소개하지는 못하지만, 그 밖의 중요 장치 드라이버로는 아래와 같은 것들이 있다.

- 1) HDMI, TVout 드라이버
- 2) 2D/3D 그래픽 가속 드라이버(그림 6-57 참조)
- 3) 비디오 인코딩/디코딩 드라이버(그림 6-58, 6-59 참조)
- 4) Modem Interface 드라이버
- 5) GPS 드라이버 등

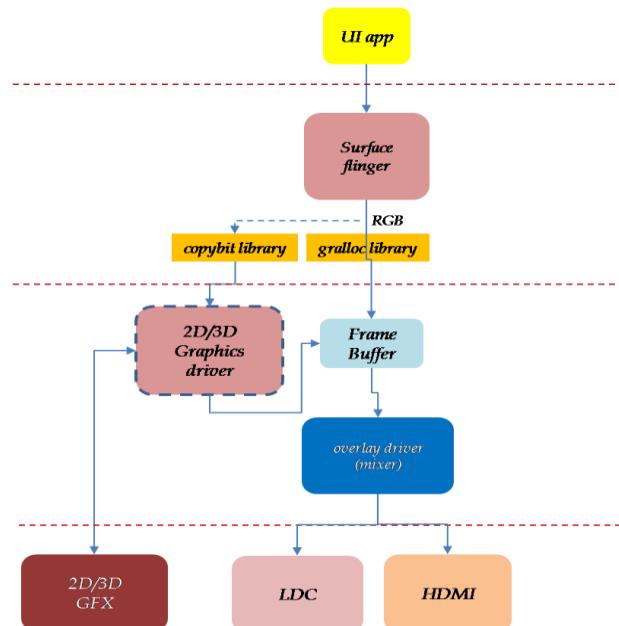


그림 6-83 2D/3D 그래픽 가속 드라이버 개관

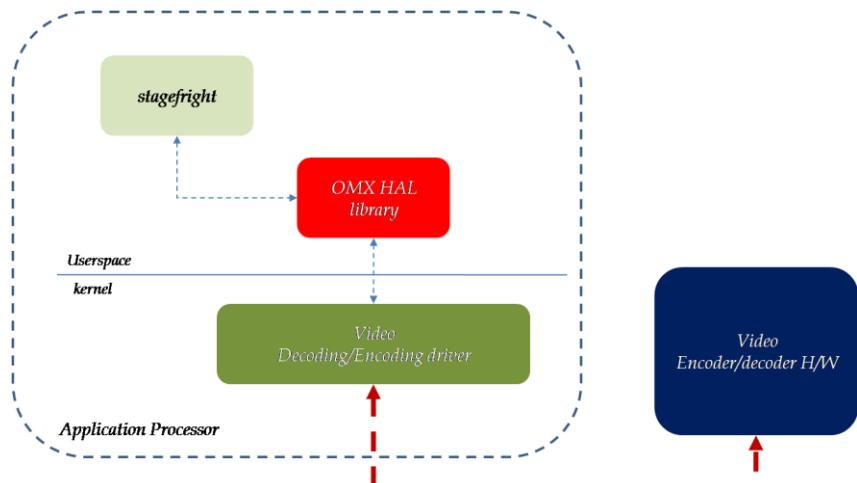


그림 6-84 Video Encoding/Decoding 드라이버 개관(1)

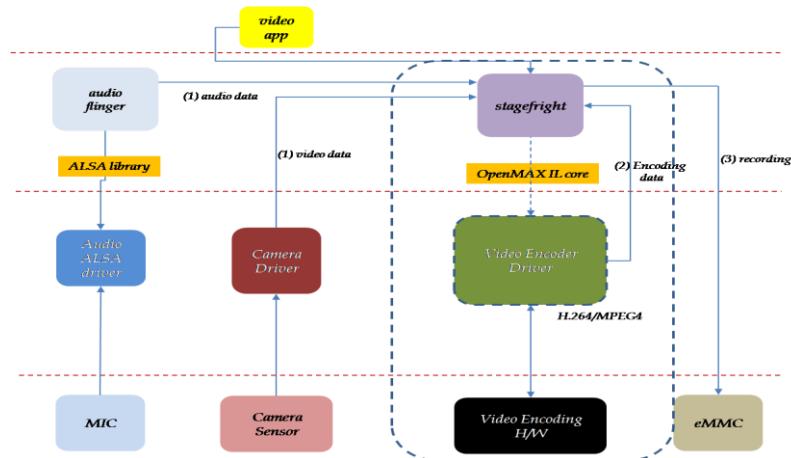


그림 6-85 Video Encoding/Decoding 드라이버 개관(2)

References

<Generic Reference>

[1] *Essential Linux Device Drivers*, Sreekrishnan Venkateswaran, Prentice Hall.

[2] *Designing Embedded Hardware, Second Edition*, John Catsoulis, O'Reilly.

[3] *Embedded Linux Primer, Second Edition*, Christopher Hallinan, Prentice Hall.

[4] *Embedded Linux System Design and Development*, P.Raghavan, Auerbach Publications.

[5] *Android Device Driver Collage2*, Chunghan Yi, www.kandroid.org

<OMAP reference>

[6] *Android OMAP Summary*, Chunghan Yi.

[7] *OMAPTM 4 PandaBoard System Reference Manual*, pandaboard.org.

[8] *OMAP4430 Multimedia DeviceSilicon Revision 2.x(Rev. AN)*, Texas Instruments.

[9] www.omappedia.org

<Pin Control, GPIO reference>

[10] *Pin Control Sysytem – Building Pins and GPIO from the ground up*, Linus Walleij Linaro Kernel Workgroup ST-Ericsson

<e-MMC reference>

[11] *TN-52-05: e-MMC Linux Enablement*, Micron.

<I2C reference>

[12] *I2C on a Linux based embedded system Design of a bus driver and a client driver for the Nomadik NHK8815 platform*, Ghiringhelli Fabrizio Matr. 753368.

<SPI reference>

[13] *GPIO, SPI and I2C from Userspace, the True Linux Way*, Baruch Siach, baruch@tkos.co.il, Tk Open Systems, June 27, 2011

[14] *Linux Configuration for Spansion® SPI, Application Note*

<USB reference>

[15] *Linux USB drivers*, Michael Opdenacker Free Electrons.

[16] *Useful USB Gadgets on Linux* February, 2012 Gary Bisson, Adeneo Embedded, Embedded Linux Conference 2012

[17] *Bootstrap Yourself with Linux-USB Stack: Design, Develop, Debug, and Validate Embedded USB*, Rajaram Regupathy, Course Technology

<MIPI reference>

[18] *MIPI Alliance Specification for D-PHY Version 1.00.00 – 14 May 2009*, MIPI Board Approved 22 September 2009

[19] *MIPI D-PHY Interface Test*, Jack Lee, Teradyne

<Display & LCD reference>

[20] *Common Display Framework(V2) – Introduction*, Tom Gall, Graphics Working Group

[21] *Common Display Framework*, Linaro Connect Europe Dublin – 2013/07/08, Laurent Pinchart

[22] *Board Bringup: LCD and Display Interfaces, Slides and Resources* at <http://www.elinux.org/BoardBringupLCD>

<V4L2 & Camera reference>

[23] *Linux Video Driver Architecture*, Chin-Feng Lai, Assistant Professor, institute of CSIE, National Ilan University, Oct 6th 2012

[24] *SP 1.20 DaVinci Linux V4L2 Display Driver User's Guide*, Literature Number: SPRUEL3 March 2008

[25] *Framework for digital camera in Linux*, Dongsoo Kim, Heungjun Kim, Samsung Electronics

[26] *Exposing the Android Camera Stack*, Balwinder Kaur, Joe Rickson, The San Francisco Android User Group

[27] *Documentation/video4linux/v4l2-framework.txt*

<ALSA & Audio reference>

[28] *Audio in embedded Linux systems, Free Electrons*

<Wi-Fi reference>

[29] *AR6003 for Android Platform, Wilson Loi/ Date [20110324]*

<NFC reference>

[30] *Near_Field_Communication_with_Linux.pdf, Samuel Ortiz Intel Open Source Technology Center*

[31] *Linux NFC Subsystem, Lauro Ramos, Venancio, Samuel Ortiz, 2011, October 26th*

[32] *NFC - NEAR FIELD COMMUNICATIONS, ubho Halder and Aditya Gupta*

[33] *NFC: caught between hype and compromise, Stephan Hartwig, STC*

[34] *Open NFC - Android Gingerbread 2.3.2 – SDK, Inside Secure*

[35] *NFC 기술 및 소개, Mobile Platform 개발본부 Tag기술개발팀 강기천 매니저*

[36] *Mobile/NFC Security Fundamentals Secure Elements 101, Smart Card Alliance Webinar □ March 28, 2013*

<Bluetooth reference>

[37] *Introduction to Bluetooth® low energy, Nordic Semiconductor*

[38] *Bluetooth Low Energy on Android, Getting it done, Szymon Janc Łukasz, Rymanowski*

[39] *BLUETOOTH LOW ENERGY, Flavia Martelli, flavia.martelli@unibo.it*

[40] *Bluetooth® Standardization, 2010. 4. 30 BQTF / Network Testing Center*