Android Kernel Hacks

안드로이드를 위한 리눅스 커널 핵스

2013.7.16 ~ 2013.10.04

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

목차

CHAPTER 5

파워 관리(Power Management) 기법

본 장에서는 안드로이드(리눅스) 파워 관리(Power Management) 기법을 소개하고, 각각의 개념을 분석해 보고자 한다.

- Power Management 개요
- Old System Suspend 기법(wakelocks, early suspend/late resume)
- New System Suspend 기법(wakeup sources, autosleep)
- CPU PM
  - CPUFreq Framework, CPUIdle Framework
- I/O Runtime Power Management 기법
- Regulator Framework
- 배터리 충전 드라이버 분석

1. 파워 관리(Power Management) 개요

파워 관리는 배터리를 사용하는 스마트 폰, 태블릿 PC 등에서 간과해서는 안 되는 매우 중요한 영역이다. Linux 및 Android에서는 이미 오래 전부터 관련하여 다양한 기법이 연구되어 왔는데, 본 장에서는 이들에 관하여 집중적으로 분석해 보고자 한다.

파워 관리 기법은 크게 아래의 두 부류로 분류가 가능한데, 하나는 시스템 전체가 suspend나 low-power 상태로 진입하도록 하는 system sleep이고, 다른 하나는 필요한 특정 device만 suspend나 low-power 상태로 진입하게 만드는 runtime sleep이다.

1) System Suspend Power Management 기법

- Linux suspend system(PM core)
- Early suspend, Late resume, Wakelocks(Android 기법 – 구식 기법)
- Autosleep, Wakeup sources(신규 기법)

2) Runtime Power Management 기법

- CPUfreq(P states)
- CPUIdle(C states) + PM QoS
- CPU hotplug
- Clock framework(4장에서 이미 소개함)
- I/O Runtime PM framework

주요 파워 관리 기법의 용어를 정리하면 다음과 같다.

표 5-1 PM 기법 정리

PM 기법	설명
System suspend	시스템 전체를 sleep 상태로 만듦(User space forces system to sleep). User space로부터 명령이 내려감.
Auto sleep	Wakeup 주체가 더 이상 없으면, 시스템이 sleep으로 자동으로 진입하게 됨(kernel에 의해 진행됨)
CPU idle	CPU가 더 이상 할 일이 없을 때, Idle thread가 sleep 상태로 진입하도록 해 줌.
CPU freq	CPU 주파수(frequency)를 조정(줄이거나 늘임)하는 기법
I/O Runtime PM	장치 관련 파워 관리 기법
CPU hotplug	동작중인 시스템에서 CPU를 제거(내림)하는 기법

그림 5-1은 커널에서 제공하는 파워 관리 영역을 하나의 그림으로 표현한 것이다.

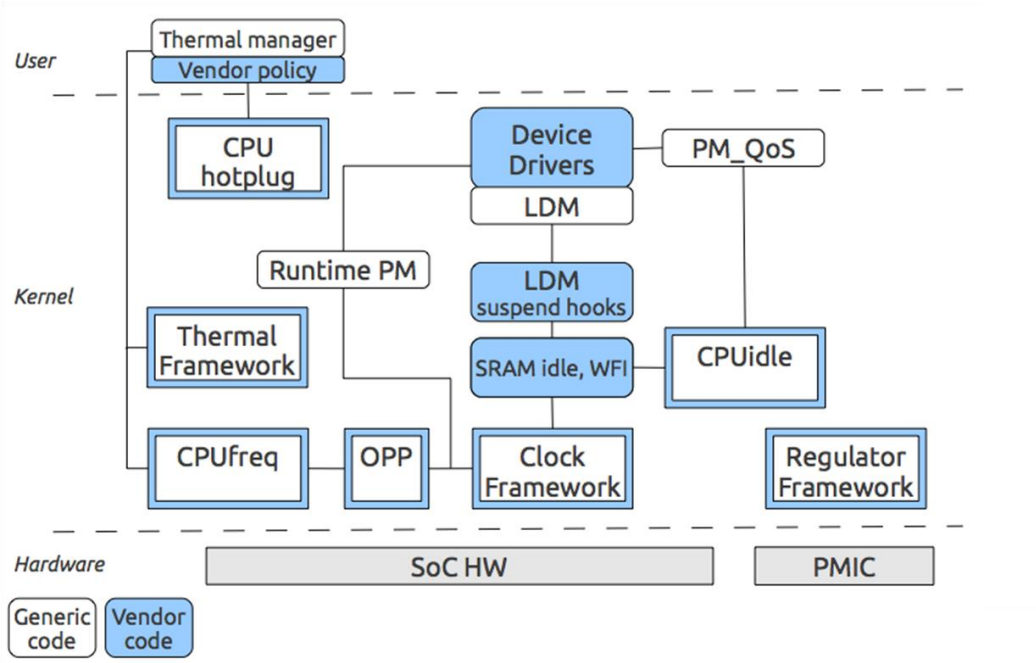

그림 5-1 Power Management 관련 영역 [출처 – 참고문헌 4]

아래에 BSP 작업 시 파워 관리 관련하여 고민해 보아야 할 사항을 정리해 보았다. 이를 통해 파워 관리의 범주를 다시 한번 재확인해 볼 수 있을 것이다.

1) clock을 정의하고, 각각의 장치와 연결시켜야 한다.

2) 보드에 장착된 각 디바이스 드라이버의 suspend/resume 관련 PM 핸들러(callback 함수)를 구현해야 한다.

3) 각각의 디바이스 드라이버 별로 runtime PM 핸들러를 구현해야 한다.

4) 배터리 관리 코드를 구현해야 한다.

5) 레귤레이터 framework 코드를 작성해야 한다.

6) 여기까지 작업하면 PM 관련 대부분의 영역이 작업되어 있어야 한다 – suspend/resume, cpuidle, cpu frequency, voltage scaling.

이 밖에도 대부분의 칩 제조사에서는 위의 내용으로는 부족하여, 전용 파워 관리 칩(PMIC)을 별도로 장착하여 사용하고 있는데, 이는 파워 관리가 그 만큼 중요하고도 어려운 영역에 속함을 암시한다.

2. System Suspend(Opportunistic Suspend)

이번 절에서는 시스템 전체에 영향을 주는 파워 관리 기법을 소개하고자 한다. 일명 System Suspend로 알려진 이 기법은 사용자 계층으로부터의 명령을 전달 받아 일순간에 전체 시스템을 sleep 모드로 전환시켜 주는 방법이다. 이 방법을 효과적으로 운용하기 위해서, 안드로이드에서는 wakelock(suspend blocker라고도 함)이라는 새로운 기법을 동원하여 일정 기간 상당한 효과를 보았다. 그러나, 메인 라인(mainline) 리눅스 커널에 통합되는 과정에서 문제가 대두(Linux 진영에서 문제를 제기함)되었고, 급기야는 Linux 커널이 버전 업되면서 다른 기능(autosleep, wakeup source)으로 교체되는 과정을 겪게 된다.

2.1 리눅스 PM Core

자 그럼, 우선 Linux 커널에서 제공하는 system suspend 파워 관리 기법을 살펴보기로 하자. 먼저, 커널에서 시스템의 파워 상태를 확인하고 이를 변경하는 방법을 그림으로 살펴보면 다음과 같다.

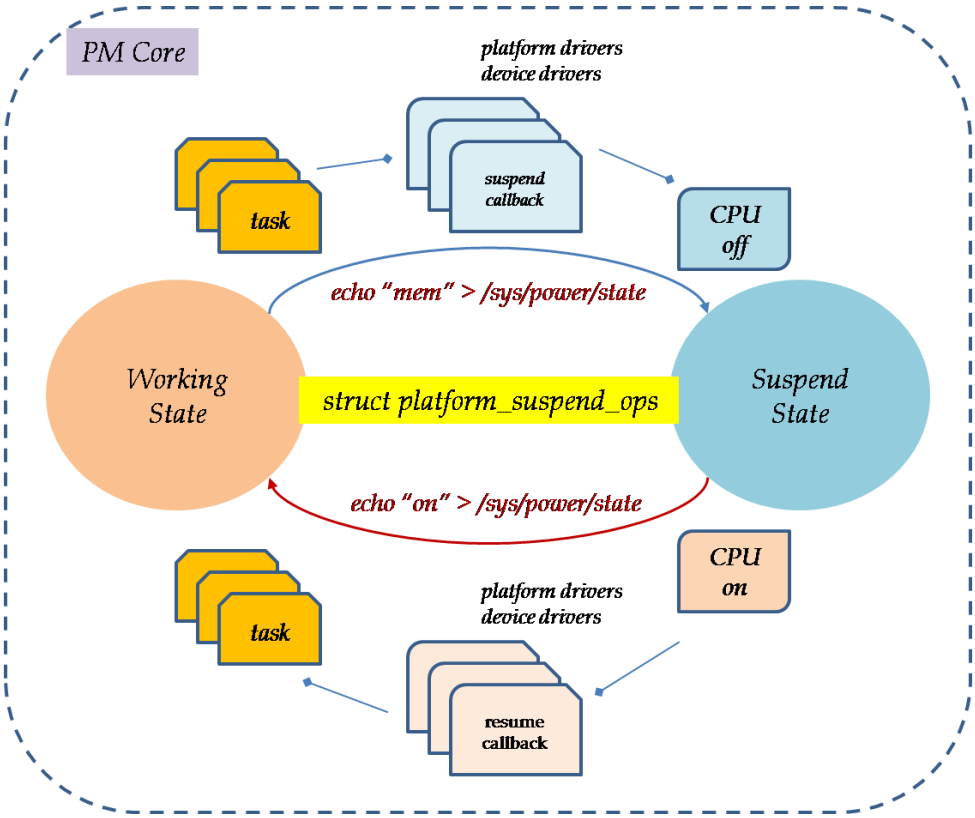

그림 5-2 Linux Power Management 개요 – PM Core 구조

그림에서 볼 수 있듯이, 시스템은 Working 상태와 Suspend 상태로 구분이 가능한데, 이 둘간의 상태 전환을 담당하는 것이 바로 PM(Power Management) Core이다. 먼저, 시스템이 Working 상태에서 Suspend 상태로 전환(줄여서 Suspend 절차라고 함)하기 위해서는 /sys/power/state에

"mem" 값을 써주어야 하며, 반대로 Suspend 상태에서 Working 상태로 전환(줄여서 Resume 절차라고 함)하기 위해서는 /sys/power/state에 "on" 값을 써주어야 한다.

Working 상태와 Suspend 상태 간의 상호 전환을 위해서는 실제로 응용 프로세스(task), 각종 주변 장치 드라이버 및 CPU가 관여하게 되며, platform_suspend_ops 구조체가 전체 흐름의 뼈대 역할을 담당한다.

지금부터는 PM Core의 구성 요소와 Suspend 및 Resume 절차를 하나씩 정리해 보도록 하겠다.

1) PM Core의 구성

a) PM Core 프레임워크

- Suspend, resume을 처리하는 메인 코드

b) struct platform_suspend_ops 수행 ..... 아래 (A) 내용 참조

- 여러 개의 callback으로 이루어진 데이터 구조
- .valid, .begin, .end, .prepare, .enter 등
- 보드 초기화 코드에서 정의하게 됨.

c) platform driver or device driver의 suspend/resume callback 수행 ..... 아래 (B) 내용 참조

- 각각의 버스 컨트롤러 및 주변 장치는 자신의 suspend, resume callback 함수를 사전에 정의해 두어야 함.

(A) struct platform_suspend_ops 선언

a) 보드 초기화 코드(arch/arm/mach-*/*) 내에서 선언해 줌.

```
static const struct platform_suspend_ops omap_pm_ops = {    //TI OMAP4의 예임.
    .begin      = omap_pm_begin,
    .end        = omap_pm_end,
    .enter      = omap_pm_enter,
    .finish     = omap_pm_finish,
    .valid      = suspend_valid_only_mem,
};
```

b) suspend_set_ops(&omap_pm_ops) 함수를 호출하여 등록해 줌(초기화해 줌).

(B) 버스 컨트롤러 및 주변 장치에서의 처리

a) Platform 드라이버(I^2C, SPI, USB bus controller 등이 이에 해당함)

- 아래 suspend, resume callback 함수를 구현하여야 함.

```
struct platform_driver {
    [...]
    int (*probe)(struct platform_device *);
    int (*remove)(struct platform_device *);
```

```
    int (*shutdown)(struct platform_device *);
    int (*suspend)(struct platform_device *, ...);
    int (*suspend_late)(struct platform_device *, ...);
    int (*resume_early)(struct platform_device *);
    int (*resume)(struct platform_device *);
    [...]
};
```

b) 일반 디바이스 드라이버(각각의 주변 장치가 이에 해당함)

- 아래 suspend, resume callback 함수를 구현하여야 함.

```
struct device_driver {
    [...]
    int (*probe) (struct device *dev);
    int (*remove) (struct device *dev);
    void (*shutdown) (struct device *dev);
    int (*suspend) (struct device *dev, pm_message_t state);
    int (*resume) (struct device *dev);
    [...]
};
```

2) Suspend 절차

- application으로 부터 "echo mem > /sys/power/state" 형태로 suspend 시작

a) struct platform_suspend_ops의 callback 중 valid() callback 함수를 호출한다.

- platform_suspend_ops data structure는 arch/arm/mach-xxx/pm.c에서 초기화됨.
- suspend_ops->valid() 함수 호출

b) 프로세스와 task를 freezing 시킨다.

c) 모든 device driver의 suspend callback 함수를 호출한다.

- 실제 각 주변 장치를 suspend 상태로 진입시켜 줌.

d) core device를 suspend 시킨다.

- 버스 컨트로러 등을 suspend로 만들어 줌.

e) CPU를 power off 시킨다.

- suspend_ops->enter() callback 함수를 호출함으로써, CPU power를 내림

Suspend 절차를 그림으로 그려 보면 다음과 같다.

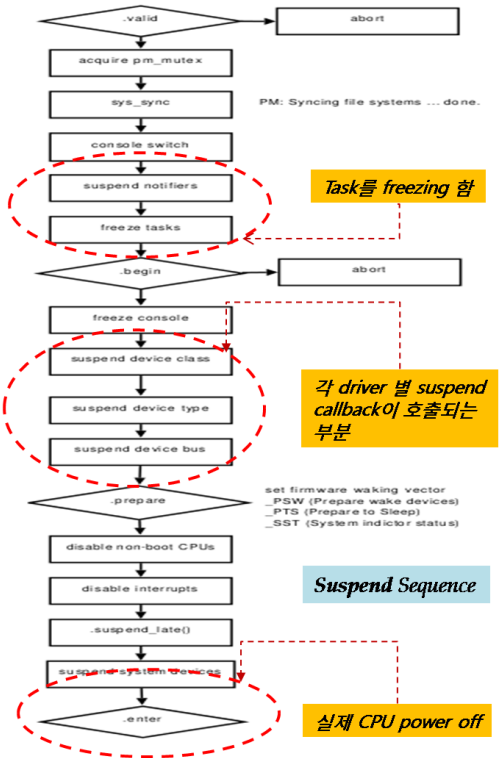

그림 5-3 PM Core의 Suspend 흐름 분석 [출처 – 참고 문헌 6]

3) Resume 절차

- application으로 부터 "echo on > /sys/power/state" 형태로 resume 시작
- interrupt 등에 의해 시작됨(예: power key 누름, 전화 걸려옴 ...). 따라서 이와 관련된 장치는 항상 wakeup 상태로 있어야 함.

a) CPU를 power on 시킨다.

b) System 장치(/sys/devices/system)를 먼저 깨운다.

- 버스 컨트로러 등을 깨워 줌.

c) suspend_ops->finish() callback 함수를 호출한다.

d) 나머지 모든 장치를 깨우다(resume callback 함수 호출).

- 실제 각 주변 장치를 resume 상태로 진입시켜 줌.

e) suspend_ops->end() callback 함수를 호출한다.

f) freezing되어 있는 프로세스와 task를 깨운다.

반대로 Resume 절차를 그림으로 표현하면 다음과 같다.

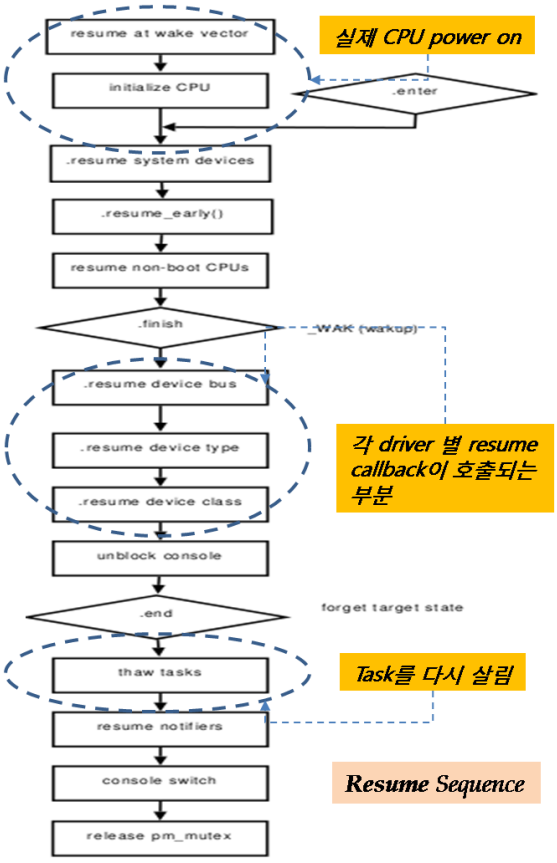

그림 5-4 PM Core의 Resume 흐름 분석 [출처 – 참고 문헌 6]

이 절에서 마지막으로 소개할 내용은, Suspend 절차를 실제로 수행하는 코드에 관한 것이다. Resume 절차는 suspend 절차와 반대 개념으로 보면 되므로, 여기서 별도로 정리하지는 않을 것이다. 자, 그럼 아래 코드 5-1의 Suspend 진입 절차 코드를 주의 깊게 살펴보기 바란다.

코드 5-1 리눅스 Suspend 진입 절차 코드

1) echo "mem" > /sys/power/state

- *state_store() 함수 호출됨.*
- *power_attr(state) 매크로 참조*

2) state_store( ) *[in kernel/power/main.c file]*

- *enter_state()*

*3) **enter_state**()* *[in kernel/power/suspend.c file]*

- *suspend_sys_sync_queue()*
- *suspend_prepare()*
- *suspend_devices_and_enter(state)*

4) suspend_devices_and_enter() *[in kernel/power/suspend.c file]*

- ***suspend_ops->begin()***
- *suspend_console()*
- *dpm_suspend_start(PMSG_SUSPEND)*
- *suspend_enter()*

5) suspend_enter() *[in kernel/power/suspend.c file]*

- ***suspend_ops->prepare()***
- *dpm_suspend_end(PMSG_SUSPEND)*
- ***suspend_ops->prepare_late()***
- *disable_nonboot_cpus()*
- *arch_suspend_disable_irqs()*
- *syscore_suspend()*
- ***suspend_ops->enter**(state)*
  - *여기까지 실행되면, 최종 적으로 CPU power를 내려 주게 됨*
  - *enter() callback 함수 관련해서는 아래 코드 참조*

위의 코드 5-1에서 중간 중간에 보이는 suspend_ops->begin(), prepare(), enter() 등의 함수는 어떻게 초기화되는지 궁금할 것이다. 이는 보드 초기화 코드(arch/arm/mach-*/*)를 수행하는 과정 중에 초기화되게 되는데, 코드 5-2에 이 과정을 정리해 보았다.

코드 5-2 suspend_ops->enter 흐름 분석 – OMAP4의 예

```
static int omap4_pm_suspend(void)      ⑥          [in arch/arm/mach-omap2/pm44xx.c file]
    // 이 함수 내에서 실제 OMAP4 CPU의 파워를 내리고 있음.
{
    [...]
    omap4_enter_lowpower(cpu_id, PWRDM_POWER_OFF);
    [...]
    return 0;
}

int __init omap4_pm_init(void)
{
    [...]
    omap_pm_suspend = omap4_pm_suspend;
    [...]
}

static int omap_pm_enter(suspend_state_t suspend_state)
```

```
{
        int ret = 0;

        if (!omap_pm_suspend)
                return -ENOENT; /* XXX doublecheck */

        switch (suspend_state) {
        case PM_SUSPEND_STANDBY:
        case PM_SUSPEND_MEM:
                ret = omap_pm_suspend();        ⑤
                break;
        default:
                ret = -EINVAL;
        }

        return ret;
}

static const struct platform_suspend_ops omap_pm_ops = {
        .begin          = omap_pm_begin,
        .end            = omap_pm_end,
        .enter          = omap_pm_enter,        ④
        .finish         = omap_pm_finish,
        .valid          = suspend_valid_only_mem,
};

//OMAP4 보드 초기화 과정 중에 아래 suspend_set_ops() 함수 호출
int __init omap2_common_pm_late_init(void)                    [in arch/arm/mach-omap2/pm.c file]
{
        [...]
        suspend_set_ops(&omap_pm_ops);      ①
        [...]
}

void suspend_set_ops(const struct platform_suspend_ops *ops)      [in kernel/power/suspend.c file]
{
        lock_system_sleep();
```

```
    suspend_ops = ops;    ②
        //앞서 설명한 suspend 커널 흐름 분석 코드 중, suspend_ops->begin/end/enter 등의
        // 함수가 이 과정에 의해 등록되는 것임.
    unlock_system_sleep();
}
EXPORT_SYMBOL_GPL(suspend_set_ops);

static int suspend_enter(suspend_state_t state, bool *wakeup)
{
        [...]
        error = suspend_ops->enter(state);    ③
        [...]
}
```

2.2 Android System Suspend 기법

Google은 안드로이드 작업을 하면서, System Suspend 기법을 위해 커널에는 Wakelock, Early suspend, Late resume 등을, 프레임워크에는 파워 매니저를 새롭게 추가하였다.

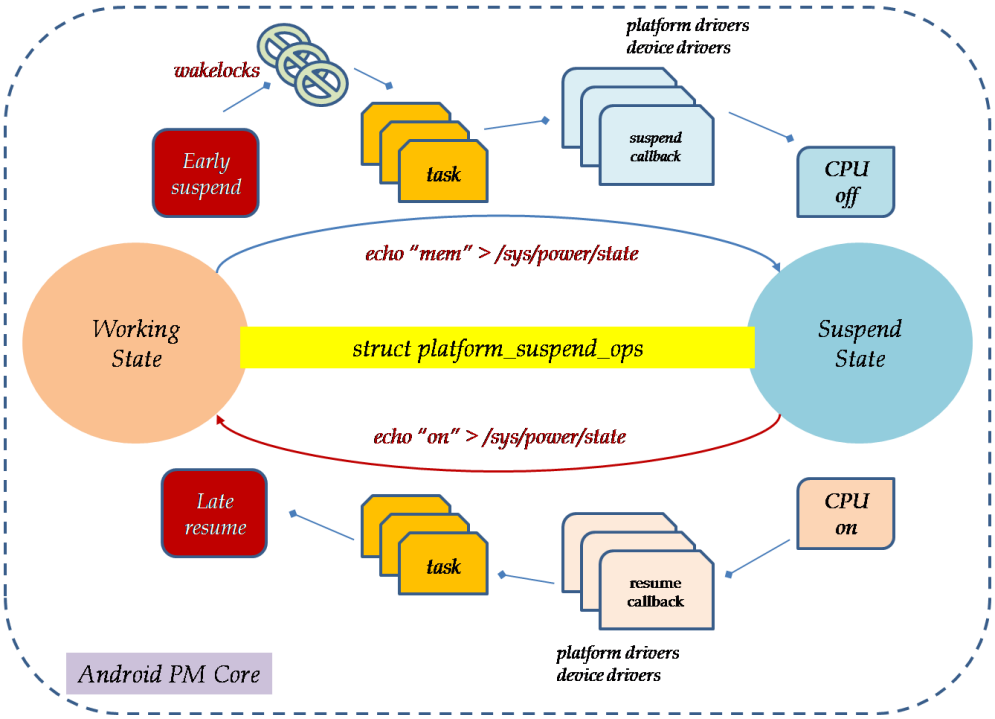

그림 5-5 안드로이드 PM 개요

Wakelock은 android 전원 관리 시스템의 핵심을 이루는 기능으로, 시스템이 suspend 혹은 low

power state로 가는 것을 막아주는 메카니즘이다. Smart Phone은 전류를 많이 소모하므로, 사용하지 않을 경우 항상 sleep mode로 빠질 수 있는 준비를 하고 있지만, 내부적으로 동작중인 디바이스 드라이버나, 응용 프로그램에서 선언된 wakelock에 막혀 sleep할 수가 없게 된다. 다시 말하면, sleep mode로 진입하기 위해서는 동작 중인 디바이스 드라이버나 응용 프로그램에서 wakelock을 스스로 해제해 주어야만 한다는 것이다.

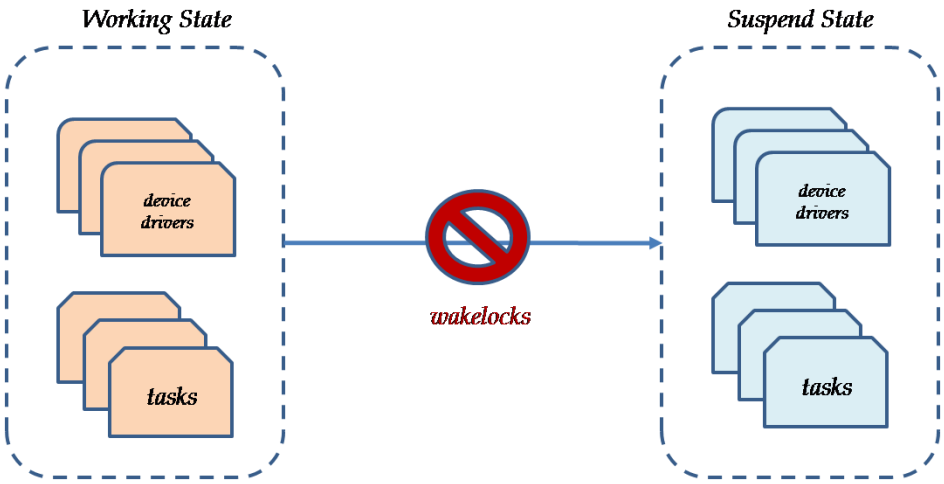

그림 5-6 Wakelock의 개념도

디바이스 드라이버 Wakelock

디바이스 드라이버에서 wakelock을 사용할 경우의 관련 API를 그림으로 정리해 보면 다음과 같다.

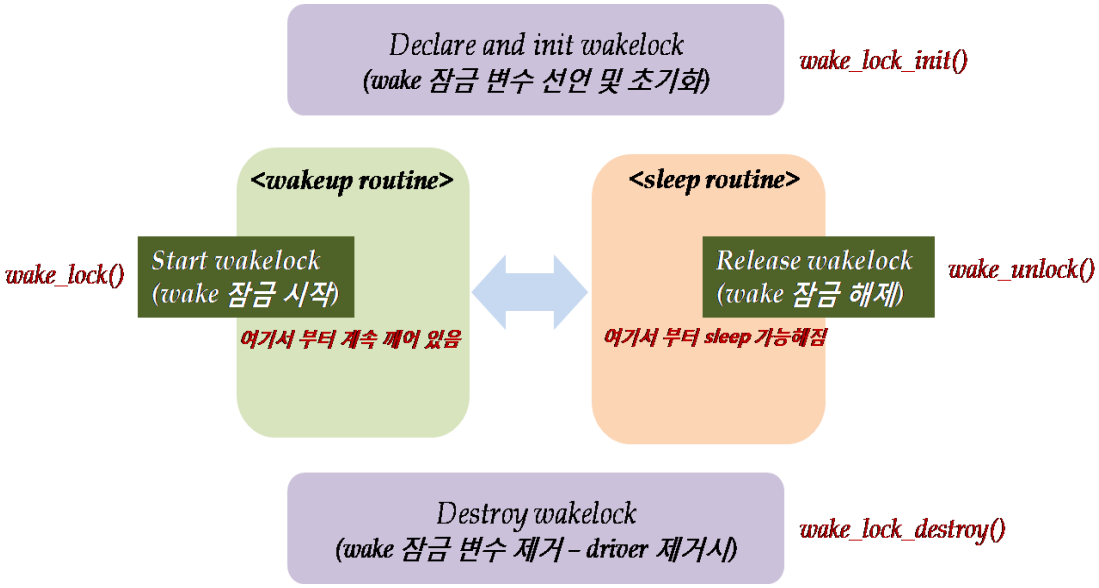

그림 5-7 Kernel wakelock API

[Kernel Wakelock 관련 주요 API 소개]

<변수 선언>

*struct **wakelock** mywakelock;*

<초기화>

***wake_lock_init**(&mywakelock, int type, "wakelock_name");*

- type :
  - WAKE_LOCK_SUSPEND: 시스템이 suspend상태(full system suspend)로 가는 것을 막음
  - WAKE_LOCK_IDLE: 시스템이 low-power idle 상태로 가는 것을 막음.

<wake 상태로 유지하기>

***wake_lock**(&mywakelock);*

<sleep 상태로 이동하기>

***wake_unlock**(&mywakelock);*

<일정 시간 후, sleep 상태로 이동하기>

***wake_lock_timeout**(&mywakelock, HZ);*

<wakelock 제거하기>

***wake_lock_destroy**(&mywakelock);*

<기타 API>

*long **has_wake_lock**(int type);*

- 한 개 이상의 wake lock이 사용 중이면 non-zero 값 return 함.

코드 5-3 예제 코드 – drivers/rtc/alarm.c [단, Code Aurora Forum에서 받은 코드]

```
static struct wake_lock alarm_rtc_wake_lock;      ① wakelock 선언


static void update_timer_locked(struct alarm_queue *base, bool head_removed)
{
        [...]
        wake_unlock(&alarm_rtc_wake_lock);      ④ wakelock 해제
        [...]
}

int alarm_set_rtc(struct timespec new_time)
```

```
{
        [...]
        wake_lock(&alarm_rtc_wake_lock);      ③ wakelock 지정
        [...]
}

static void alarm_triggered_func(void *p)
{
        [...]
        wake_lock_timeout(&alarm_rtc_wake_lock, 1 * HZ);  ③ timeout wakelock 지정
}

static int alarm_suspend(struct platform_device *pdev, pm_message_t state)
{
        [...]
        wake_lock_timeout(&alarm_rtc_wake_lock, 2 * HZ);  ③ timeout wakelock 지정
        [...]
}

static int __init alarm_driver_init(void)
{
        [...]
        wake_lock_init(&alarm_rtc_wake_lock, WAKE_LOCK_SUSPEND, "alarm_rtc");      ② 초기화
        [...]
}

static void   __exit alarm_exit(void)
{
        [...]
        wake_lock_destroy(&alarm_rtc_wake_lock);      ⑤ wakelock 삭제
        [...]
}
```

Wakelock은 특별한 사항은 없고, 아래에서 보는 것처럼 active_wake_locks linked list에 목록이 존재하지 않을 경우, suspend로 진입이 가능한 형태로 구성되어 있다. 뿐만 아니라, wake_lock 자체도 flags 및 expires 필드만을 가지고 운영되는 매우 간단한 구조체임을 알 수 있다 (kernel/power/wakelock.c 파일 참조).

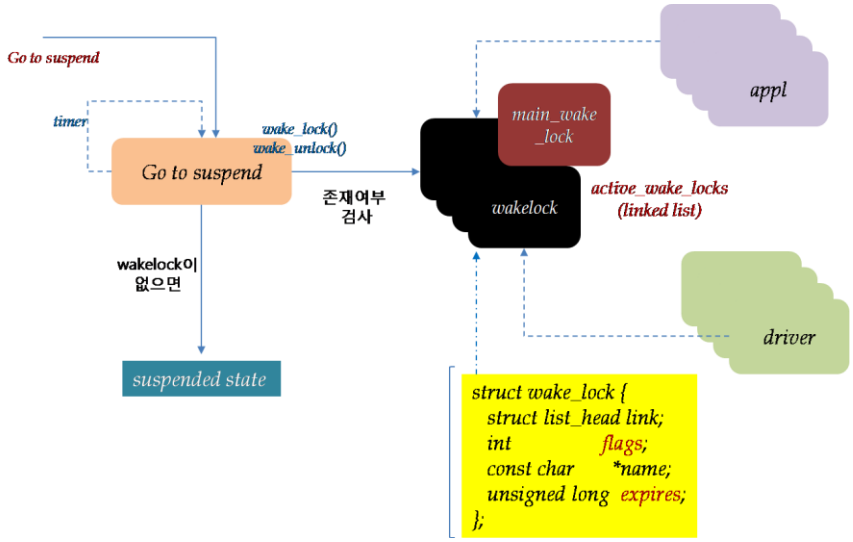

그림 5-8 Kernel wakelock의 동작 원리

응용 프로그램의 Wakelock

Wakelock은 커널 내에서뿐만 아니라, 응용 프로그램에서도 사용이 가능하다. 이 부분은 앞으로 5.3절에서 언급하게 될 wakelock이 가진 문제점에 해당하는 내용이기도 한데, 일단은 android 응용 프로그램에서 wakelock을 획득하고 해제하는 과정을 그림을 통해 간략히 살펴 보고 넘어기로 하자.

[참고] 안드로이드 파워 서비스 프레임워크 관련하여 보다 자세한 사항은 참고문헌 [16]을 참고하기 바란다.

코드 5-4 응용 프로그램에서 wakelock 사용

```
PowerManager pm = (PowerManager)getSystemService(Context.POWER_SERVICE);
PowerManager.WakeLock wl =
                    pm.newWakeLock(PowerManager.SCREEN_DIM_WAKE_LOCK, "My Tag");
wl.acquire();
/* screeen will stay on during this section ... */
wl.release();
```

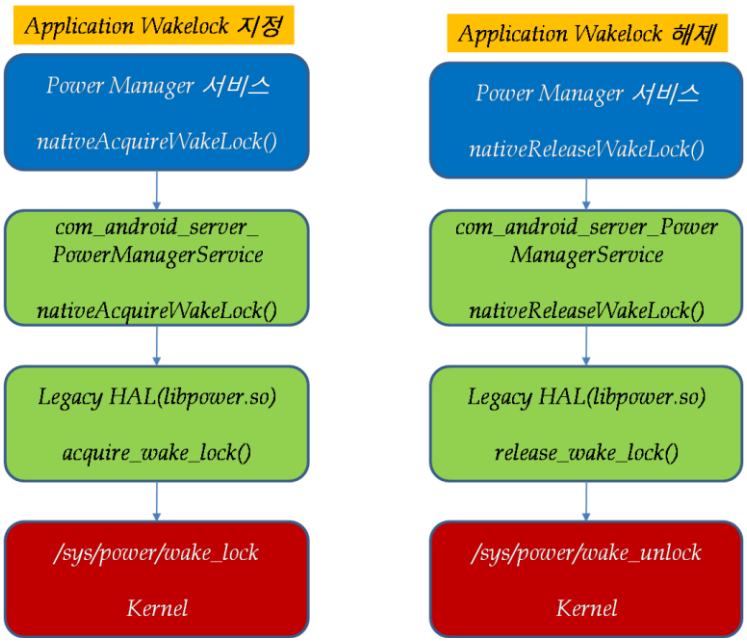

그림 5-9 응용 프로그램 wakelock 지정 및 해제 과정

Wakelock에 대한 소개를 마쳤으니, 다음으로 설명할 내용은 Early suspend, Late resume 기능에 관한 것이다.

<Early Suspend>

- Google에서 리눅스 커널에 새로 추가한 부분으로, 리눅스 커널의 original suspend 상태와 LCD screen off 사이에 존재하는 새로운 상태를 말한다. LCD를 끄면 배터리 수명과 몇몇 기능적인 요구 사항에 의해 LCD back-light나, G-sensor, touch screen 등이 멈추게 된다.

<Late Resume>

- Early Suspend와 쌍을 이루는 새로운 상태로, esume이 끝난 후 수행되며, early suspend 시 꺼진 장치들이 다시 켜지게 된다.

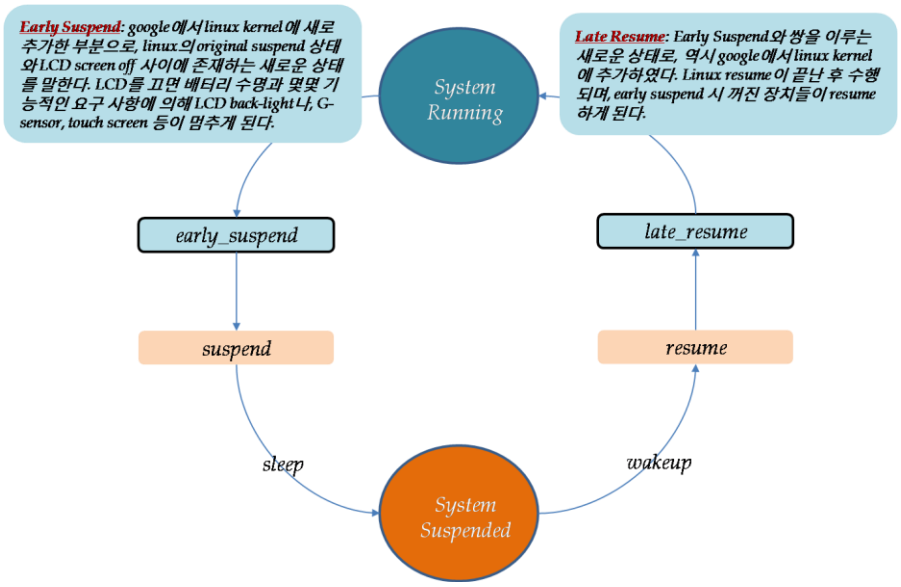

그림 5-10 안드로이드의 Early Suspend와 Late Resume 상태

Early suspend 상태를 거쳐 시스템이 suspend 상태로 진입하는 전체 절차를 요약하면 다음과 같다.

<Early Suspend & Suspend Step>

1) Android application에서 application 수행 중, 임의의 시점에서 wakelock을 건다.

2) Linux device driver에서 driver 동작 중, 임의의 시점에서 wakelock을 건다.

3) Sleep 요청(goToSleep)이 들어온다.

4) PowerManager -> PowerManagerService -> HAL(Power.c)을 거쳐 kernel로 suspend(=sleep) 요청이 내려간다.

- 화면이 off일 때, Power Manager Service는 early suspend를 요청한다.
- PARITAL_WAKE_LOCK을 제외한 wake lock이 잡혀 있을 경우에는 early suspend를 요청하지 않는다.
- "mem" 값을 "/sys/power/state"에 기록한다.

5) early_suspend를 처리하기 위한 work(early_suspend_work)을 <suspend> work queue에 전달한다.

<work queue 내부 처리>

- early_suspend( ) 함수 내에서는 사용자가 등록한 early_suspend callback 함수를 모두 호출한다.
- 마지막으로 main_wake_lock을 풀어준다.

6) 현재 동작중인 wake lock이 존재하는지 확인(세 군데)하여, 없을 경우 suspend를 처리하기 위한 work(suspend_work)을 <suspend> work queue에 전달한다.

- wakelock 존재 여부 검사는 timer를 이용하여 주기적으로 수행하는 것은 물론이고, wake_lock()/wake_unlock()함수 호출 시에도 행해진다.
- wakelock이 존재하지 않을 경우, 이 곳에서 suspend work을 <suspend> work queue로 던지게 됨.

- wakelock은 응용 프로그램 및 디바이스 드라이버에서 생성 가능하다.
- 또한, suspend/resume을 관리하기 위한 main_wake_lock이 존재한다.

7) 시스템이 마침내 suspend 상태로 바뀐다.

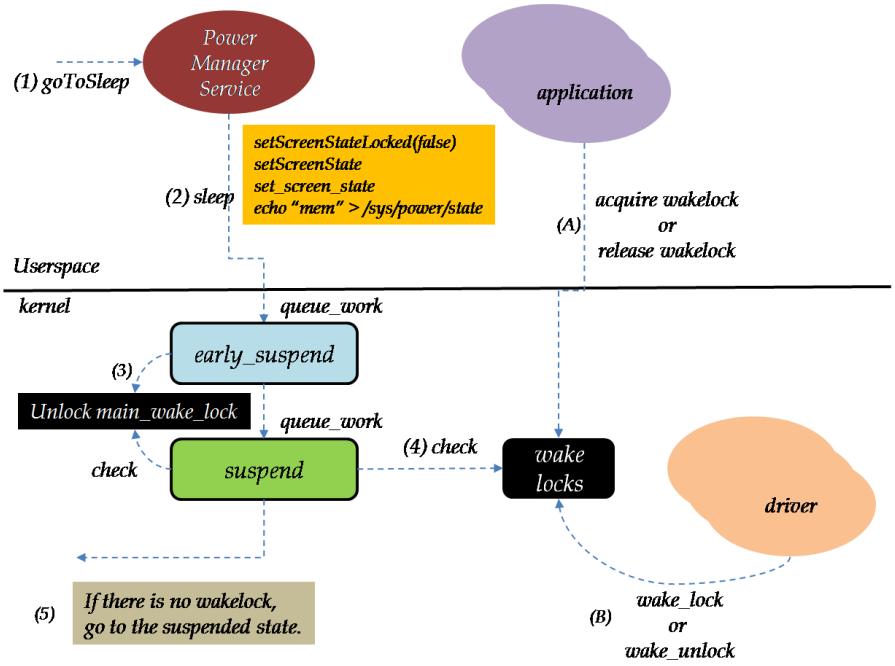

그림 5-11 Early Suspend & Suspend 상태 진입

이와는 반대로 Resume 상태를 거쳐 Late resume을 진입하는 과정을 소개하면 다음과 같다.

<Resume & Late Resume Step>

1) 사용자가 power key를 누른다.

- System을 resume 시키는 방법은 이 외에도 여러 가지가 있을 수 있음(ex: 전화가 온다).

2) kernel이 resume 루틴을 구동시킨다.

3) Power key는 input driver를 경유하여, android framework으로 전달된다.

4) Android framework으로 전달된 power key는 다시 (system_server 내의)EventHub -> Input Manager -> Window Manager 순으로 차례로 전달된다.

5) Window Manager는 power key를 인식하고, 이를 PowerManager -> PowerManagerService에게 전달한다.

- Power key가 아닌 경우에는 버린다.

6) PowerManagerService는 early suspend와 유사한 처리 과정을 거쳐 kernel에 late resume 요청을 한다.

7) 이 과정에서 "on" 값을 "/sys/power/state"에 기록한다.

8) late_resume을 처리하기에 앞서 "main_wake_lock"을 enable시킨다.

- 이 "main_wake_lock"은 early_suspend( ) 함수에서 다시 disable될 것이다.

(9) Late resume을 처리하기 위한 work(late_resume_work)을 <suspend> work queue에 전달한다.

- 이후, early_suspend 시에 suspend되었던, 장치들이 다시 깨어나게 된다.

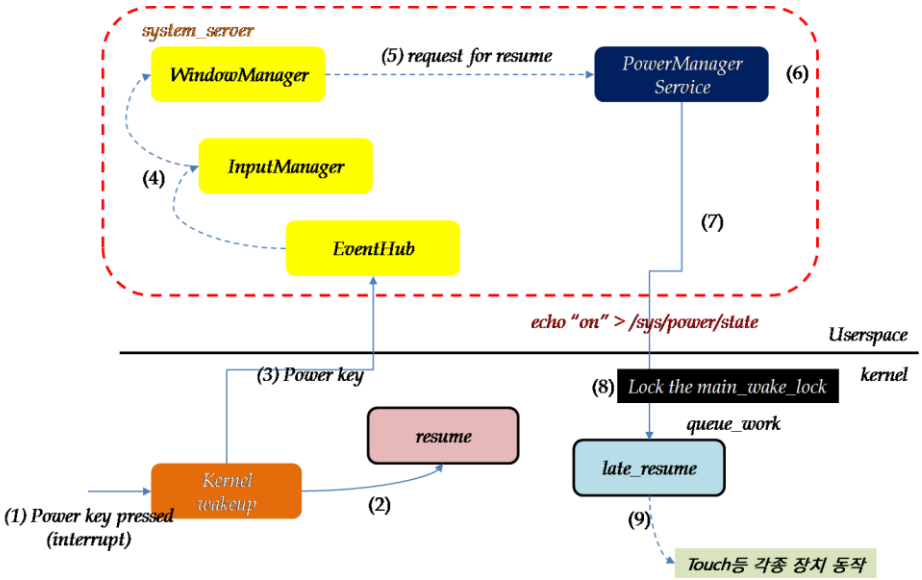

그림 5-12 Resume & Late resume 절차

앞서 절차 설명에서 언급한 바와 같이, early suspend와 late resume 처리를 위해 <suspend> work queue가 사용되고 있다. 또한, early suspend 후, 실행되는 suspend 루틴도 work queue형태로 처리되고 있다. 안드로이드 patch가 적용된 커널의 경우는 suspend 진입 시점이 아래와 같이 존재할 수 있다.

a) timer로 주기적인 검사 도중, wake_lock이 하나도 존재하지 않음을 발견한 시점

b) wake_lock(), wake_unlock() 함수에서 검사했을 때, wake_lock이 존재하지 않는 시점

파워 관리자 프로세스(안드로이드 Power Manager)에서 /sys/power/state에 "mem" 설정한 후, 커널 코드 내에서 earlysuspend 및 suspend에 진입하기 위해 수행하는 코드 절차 중 중요한 부분만을 요약해 보면 다음과 같다. 이 내용은 앞으로 2.3절에서 설명하게 될 <AutoSleep 커널 함수 흐름 분석>과 비교해 보기 바란다.

코드 5-5 안드로이드 Suspend 진입 절차

1) echo "mem" > /sys/power/state

- *Wakelock이 없을 때 마다 suspend로 진입 !!!)*
- *state_store() 함수 호출됨.*
- *power_attr(state) 매크로 참조*

<earlysuspend 단계>

2) state_store() *[in kernel/power/main.c file]*

3) request_suspend_state() *[in kernel/power/earlysuspend.c file]*

4) queue_work(early_suspend_work_queue, &early_suspend_work);

- *static DECLARE_WORK(early_suspend_work, early_suspend); 문 참조*
- *early_suspend() 함수 call*

5) pos->suspend(pos) in early_suspend()

- *early suspend callback 호출 !(return 값은 없음)*

6) suspend_sys_sync_queue() in early_suspend() *[in kernel/power/wakelock.c file]*

- *queue_work(suspend_sys_sync_work_queue, &suspend_sys_sync_work);*
- *suspend_sys_sync()* *[in kernel/power/wakelock.c file]*

7) sys_sync() in suspend_sys_sync() *[in kernel/power/wakelock.c file]*

- *system call(= sys_sync() 함수 호출로 마무리)*
- *여기까지 early_suspend work 완료 !!!*

<suspend 단계>

- *suspend() 함수는 early_suspend와는 무관하에 wake lock 상태를 보고, 진입하게 된다. 따라서, wakelock이 하나라도 살아 있으면, suspend() 함수로 진입할 수 없음.*
- *실제로 위의 7)번 code 흐름에서 8)번으로 이어지는 것이 아님에 주의 !*

8) DECLARE_WORK(suspend_work, suspend); *[in kernel/power/wakelock.c file]*

9) suspend_work_queue = create_singlethread_workqueue("suspend");

10) queue_work(suspend_work_queue, &suspend_work);

- *아래 3 함수 내에서 queue_work() 호출.*
- *expire_wake_locks()*
- *wake_lock_internal()*
- *wake_unlock()*

11) suspend() *[in kernel/power/wakelock.c file]*

11-1) suspend_sys_sync_queue();

- *queue_work(suspend_sys_sync_work_queue, &suspend_sys_sync_work);*
- *suspend_sys_sync()*
- *system call(= sys_sync() 함수 호출로 마무리) !*

11-2) pm_suspend(requested_suspend_state); *[in kernel/power/suspend.c file]*

- ***enter_state()***
- *여기서 linux의 suspend routine으로 진입*

12) ***enter_state()*** *[in kernel/power/suspend.c file]*

- *suspend_sys_sync_queue()*
- *suspend_prepare()*
- *suspend_devices_and_enter(state)*

13) suspend_devices_and_enter() *[in kernel/power/suspend.c file]*

- ***suspend_ops->begin()***
- *suspend_console()*
- *dpm_suspend_start(PMSG_SUSPEND)*
- *suspend_enter()*

14) suspend_enter() *[in kernel/power/suspend.c file]*

- ***suspend_ops->prepare()***
- *dpm_suspend_end(PMSG_SUSPEND)*
- ***suspend_ops->prepare_late()***
- *disable_nonboot_cpus()*
- *arch_suspend_disable_irqs()*
- *syscore_suspend()*
- ***suspend_ops->enter**(state)*
    - *여기까지 실행되면, 최종 적으로 CPU power를 내려 주게 됨*
    - *enter() callback 함수 관련해서는 아래 코드 참조*

이상으로 안드로이드에서 새롭게 추가한 System Suspend 기능에 관하여 살펴 보았다. 자, 그럼 이 부분이 최신 리눅스 커널에서는 어떤 식으로 변모하게 되는지 살펴보기로 하자.

2.3 Autosleep, Wakeup Sources

Android Wakelock 사용 여부에 대한 고민

본 절에서는 Rafael J. Wysocki가 새롭게 제안한 AutoSleep과 Wakeup Source를 기반으로 하는 새로운 System Suspend 기법에 관하여 소개하고자 한다. 먼저 구체적인 내용을 설명하기에 앞서 lwn.net에 기고된 아래 기사를 통해 안드로이드 wakelock의 문제점과 이를 대신할 새로운 system suspend 기법에 대해 고민해 보는 시간을 갖도록 하자.

<http://lwn.net/Articles/479841/ 기사 내용 발췌>

이 글은 2012 년 2 월 7 일, Jonathan Corbet 가 기고한 것임.

"Opportunistic suspend"는 손이 많이 가는 파워 관리 기법으로, 관심의 대상(device)이 더 이상 동작하지 않을 때, 시스템 전체가 suspend 로 진입하게 되는 기법이다. 안드로이드에서는 wakelock 이라는 효과적인 기법을 통해, 잘 못 만든 응용 프로그램이 배터리를 갉아 먹는 것을 막아 줄 수 있게 되었다. 다시 말해 wakelock 을 사용하는 kernel 과 권한이 부여된 몇몇 응용 프로그램으로 하여금 시스템 suspend 로 들어가는 것을 막을 수 있도록 하였다.

몇 가지 문제점이 인식되면서, wakelock(suspend blockers)이 mainline kernel 에 통합되지 못하고 있는 차에, 이를 대체할 만한 기법이 필요하게 되었다. 이때 거론된 내용이 wakeup events framework(2.6.36 에서 추가)와 wakeup sources(2.6.37 에서 추가)인데, 이때 까지만 해도 거의 많은 드라이버에서 이 기능을 사용하지 않고 있었다.

Rafael 은 wakeup events 및 wakeup sources 에 기반한 opportunistic suspend 를 autosleep 이라는 이름으로 새롭게

명명하고, 몇몇 기능을 개선한 패치를 내놓았다. Autosleep 은 /sys/power/autosleep 에 "mem" 값을 쓰게 되면, active 한 wakeup source 가 없을 때 마다 시스템이 suspend 로 진입하게 되는 기법이다.

...중략...

Wakeup source 는 커널 기반의 wakeup events 를 관리하는 좋은 기법이나, 응용 프로그램으로 하여금 시스템이 sleep 으로 들어가서는 안 된다는 사실을 알릴 방법을 제공하지 못했다. 따라서, Rafael 의 마지막 패치에서는 이를 위한 기능이 추가되게 되었는데, 응용 프로그램에서는 /sys/power/wake_lock 파일에 이름 혹은 timeout 값을 써 줌으로써 active wakeup source 를 생성하게 되었으며, timeout 값이 소모되거나 /sys/power/wake_unlock 에 이름을 써 줄 때가지 시스템이 suspend 로 진입하는 것을 막아 줄 수 있게 되었다.

이 기법(wakeup source)을 사용하게 되면, Android 의 opportunistic suspend(wakelock)를 대체할 수 있음을 쉽사리 알 수 있을 것이다. 예를 들어 wakeup event 를 받은 드라이버는 관련 wakeup source 를 active 로 표시해 둘 것이고, 이로 인해 시스템은 계속 동작(running) 상태에 있게 될 것이다. 해당 장치(source)는 사용자 영역(user space)에서 해당 event 를 처리할 때까지 active 상태로 있게 되는데, 사용자 영역의 응용 프로그램은 wake lock(여기서의 wakelock 은 wakeup source 를 기반으로 함)을 사용함으로써 이를 가능하게 만들 것이다.

지금까지 설명한 새로운 API 는 android API 와 매우 유사함을 알 수 있을 것이다. 이는 기존 안드로이드 시스템을 크게 훼손하지 않은 상태에서 새로운 방식으로 쉽게 전환할 수 있게 하려는 의도에 기인한 것이다.

...중략...

위의 기사 내용을 요약해 보면, wakelock 기반의 Android System Suspend가 문제가 있다고 판단하여 나름 새로운 방법을 고안된 것 처럼 보인다.

<안드로이드 wakelock의 문제점>

1) kernel은 물론이고, user space의 application을 수정해야만 한다. 특히 user space application에서 wakelock을 사용하도록 한 부분은 android에는 적합할지 몰라도, wakelock에 대해서 알지 못하는 다른 GNU/Linux 시스템 입장에서는 받아들이기 힘든 부분으로, Linux는 android 뿐만 아니라 다른 GNU/Linux 시스템도 고려해서 설계되어야 한다.

2) Wakelock은 runtime PM을 대신하는 기능일 뿐이다(단, 실제로 runtime PM은 wakelock 이후에 구현된 기능이기는 함). CPUIdle + I/O runtime PM framework으로도 완벽하지는 않지만 비슷한 효과를 누릴 수 있다. 따라서 wakelock의 의미가 그 만큼 없다.

3) Android wakelock은 device structure나 PM core와 연결되어 있지 않다. 따라서, 리눅스 진영에서 볼 때는 일관성을 유지하기가 쉽지 않다. 즉, 기존의 kernel data structure와 연계될 수 있도록 수정되어야 한다.

<android wakelock에 대한 대안 제시>

1) CPUIlde(기존에 존재하던 기법)과 I/O Runtime PM(2.6.32에 추가)으로 System Suspend에 상응하는 시스템을 만들고자 한 듯하다.

2) CPUIdle + I/O Runtime PM으로는 충분하지 않자, 기존의 linux code에 존재하던 wakeup events(wakeup sources)에 기반한 구조를 이용하여 새로운 system suspend를 구현하고자 하였다.

이는 안드로이드 이외의 다른 Linux 기반 시스템에서도 사용 가능한 범용 코드를 만들기 위한 노력에서 비롯된 것으로 보인다. 하지만, 궁극적으로 보면, 내부의 코드만 바뀌었지, 외관상으로는 android에서 초기에 제안한 구조가 어느 정도 유지되었다고 볼 수도 있을 듯 하다. 아무튼 기존의 커널 프로그래머가 작성해야 했던 wakelock 관련 코드는 사라졌으나, 유사한 개념의 wakeup source 관련 코드로 교체하는 작업이 여전히 필요하며, 응용 프로그래머 입장에서는 기존과 거의 유사한 방식으로 wakelock 관련 code를 사용할 수 있게 되었다.
필자의 견해로는 가장 큰 변화는 다름 아닌, (전혀 동떨어진 개념의) android wakelock을 리눅스 스타일(struct device, struct dev_pm_info 등과 연계)로 변형 및 통합한 것이라고 보여진다.

Autosleep과 Wakeup sources

I/O Runtime PM의 개발자이기도 한, Rafael J. Wysocki는 기존의 Opportunistic suspend 기법을 Autosleep이라는 새로운 이름으로 명명하고, Wakeup Event 프레임워크에 기초한 새로운 시스템 suspend 기법을 구현하기에 이르렀다. Rafael J. Wysocki가 말하는 새로운 시스템 suspend 기법을 요약하면 다음과 같다.

1) Wakeup source는 커널 내에서 wakeup event를 생성시켜 주는 객체를 말하며, wakeup 관련 통계 정보를 수집하는 역할을 한다. 그림 5-13에서 볼 수 있듯이, struct device와 연결되어 있으며, pm_wakeup_event(), pm_stay_awake(), pm_relax() 함수에 의해서 내부적으로 사용되는 형태로 되어 있다.

- 장치(struct device로 표현)에 wakeup_source(dev->power.wakeup으로 표현됨) 필드가 할당되어 있다는 얘기는 해당 장치는 동작 중이라는 뜻임.

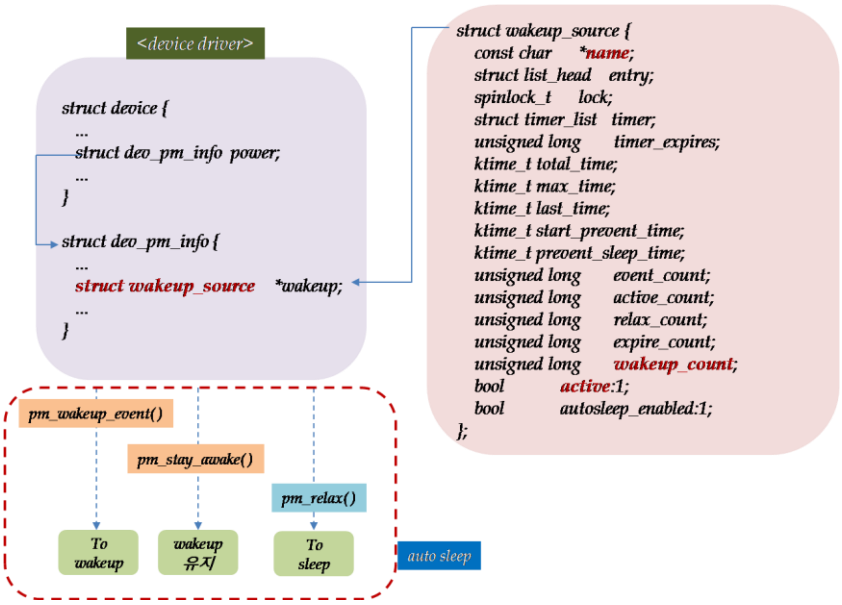

그림 5-13 Wakeup Sources

코드 5-6 struct wakeup_source

```
/**
 * struct wakeup_source - Representation of wakeup sources
 *
 * @total_time: Total time this wakeup source has been active.
 * @max_time: Maximum time this wakeup source has been continuously active.
 * @last_time: Monotonic clock when the wakeup source's was touched last time.
 * @prevent_sleep_time: Total time this source has been preventing autosleep.
 * @event_count: Number of signaled wakeup events.
 * @active_count: Number of times the wakeup sorce was activated.
 * @relax_count: Number of times the wakeup sorce was deactivated.
 * @expire_count: Number of times the wakeup source's timeout has expired.
 * @wakeup_count: Number of times the wakeup source might abort suspend.
 * @active: Status of the wakeup source.
 * @has_timeout: The wakeup source has been activated with a timeout.
 */
struct wakeup_source {
	const char		*name;
	struct list_head	entry;
	spinlock_t		lock;
	struct timer_list	timer;
	unsigned long		timer_expires;
	ktime_t total_time;
	ktime_t max_time;
	ktime_t last_time;
	ktime_t start_prevent_time;
	ktime_t prevent_sleep_time;
	unsigned long		event_count;
	unsigned long		active_count;
	unsigned long		relax_count;
	unsigned long		expire_count;
	unsigned long		wakeup_count;
	bool			active:1;
	bool			autosleep_enabled:1;
};
```

2) struct wakeup_source에 있는 event_count, wakeup_count라는 두 counter 값과 pm_stay_awake(), pm_wakeup_event(), pm_relax() 함수가 중요한 의미를 갖는데, 이를 설명하면 다

음과 같다.

- *event_count: 전송된 wakeup events의 개수*
- *wakeup_count: wakeup source가 suspend를 무시할 수 있는 시간*
- *pm_stay_awake() : PM core에게 wakeup event가 현재 처리 중임을 알려 줌. 시스템이 suspend로 진입하는 것을 막아주는 함수*
- *pm_wakeup_event() : PM core에 wakeup_event를 날려 줌. wakup event를 처리하는 동안에는 시스템이 깨어 있도록 해주는 함수*
- *pm_relax() : PM core에게 wakeup event가 처리 완료되었음을 알려 줌. 즉, device가 suspend로 진입할 수 있게 됨.*
- 자세한 사항은 아래 **Wakeup Source API** 절 내용 참조

3) 디바이스 드라이버에서 wakeup_source 정보를 참조하여 suspend, resume을 수행할 지 여부를 결정하는 내용을 아래에 정리해 보았다.

코드 5-7 wakeup source에 기반한 suspend/resume 코드 예

a) probe() 함수 내

device_init_wakeup()

b) suspend() 함수 내

가능한 한 장치를 low power 모드로 돌린다.

device_may_wakeup() 함수를 체크한다.

- *device_may_wakeup()는 wakeup_source가 할당되어 있는지를 체크하는 함수임. 즉, wakeup_source가 있다는 것은 장치는 suspend로 들어가면 안됨을 의미함.*

enable_irq_wake() 함수를 호출한다.

c) resume() 함수 내

장치를 정상 동작 모드로 전환한다.

device_may_wakeup() 함수를 체크한다.

disable_irq_wake() 함수를 호출한다.

[참고] device_may_wakeup() 함수 내용 – include/linux/pm_wakeup.h

```
static inline bool device_may_wakeup(struct device *dev)
{
    return dev->power.can_wakeup && !!dev->power.wakeup;
}
```

4) Autosleep은 active한 wakeup source가 없을 때 마다 시스템이 suspend로 진입하게 되는 기법으로, /sys/power/autosleep에 "mem" 값을 쓰게 되면 autosleep이 진행된다.

a) */sys/power/autosleep에 "mem" 값을 쓰게 되면 시스템 전체의 autosleep이 시작됨.*

b) *각각의 개별 디바이스 드라이버는 wakeup_event가 모두 처리 완료되면, 스스로 suspend로 진입하게 됨.*

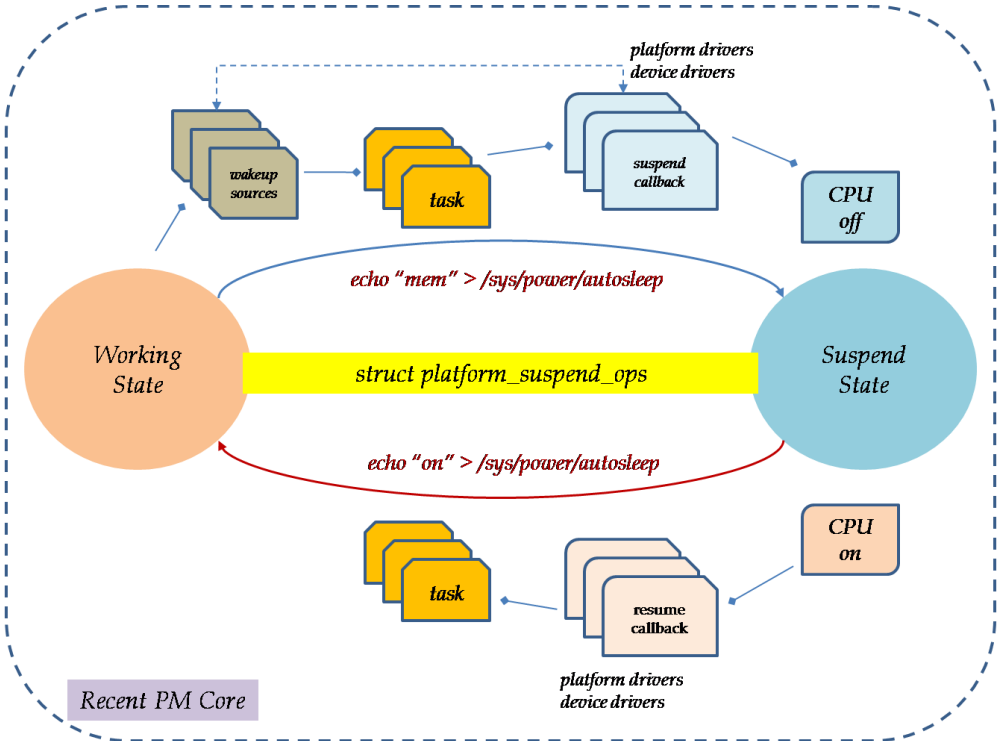

그림 5-14 Auto Sleep과 Wakeup Source기반의 새로운 PM Core 구조

5) 응용 프로그램에서는 /sys/power/wake_lock 파일에 이름 혹은 timeout 값을 써 줌으로써 active wakeup source를 생성하게 되었으며, timeout 값이 소모되거나 /sys/power/wake_unlock에 이름을 써 줄 때가지 시스템이 suspend로 진입하는 것을 막을 수 있게 되었다(그림 5-16 참조).

- 자세한 사항은 아래 **사용자 영역 Wakelock API** 절 내용 참조

6) 현재 event_count 값은 /sys/power/wakeup_count에서 읽을 수 있다. /sys/power/wakeup_count에 값을 쓰는 것도 물론 가능한데, 쓰려는 값이 event_count와 같을 경우에만 쓸 수 있게 된다. /sys/power/wakeup_count에 값을 쓰는 것이 성공할 경우, PM core는 시스템이 suspend로 진입하는 동안에 event_count 값이 바뀌었는지를 검사하게 된다.

위의 내용을 기반으로, 파워 관리자 프로세스(Android Power Manager)에서 하는 일(suspend 진입 절차)을 따라가 보면 다음과 같다:

<안드로이드 파워 매니저에서 하는 일 – Wakeup Source 기반>

Step 1) /sys/power/wakeup_count 값을 읽는다.

Step 2) Mutex lock을 얻어온다(critical section으로 진입함).

Step 3) Suspend counter 값이 0인지를 검사하여, 0이 아닐 경우, mutex lock을 해제하고, Step 2로 분기한다.

Step 4) 정상적으로 mutex lock을 해제하고, critical section을 빠져 나온다.

Step 5) Step 1 단계에서 읽었던 값(/sys/power/wakeup_count)을 다시 /sys/power/wakeup_count 파일에 쓴다. 만일 쓰기가 실패한다면, Step 1로 돌아간다.

Step 6) 이제 suspend 작업을 시작할 수 있게 되었다.

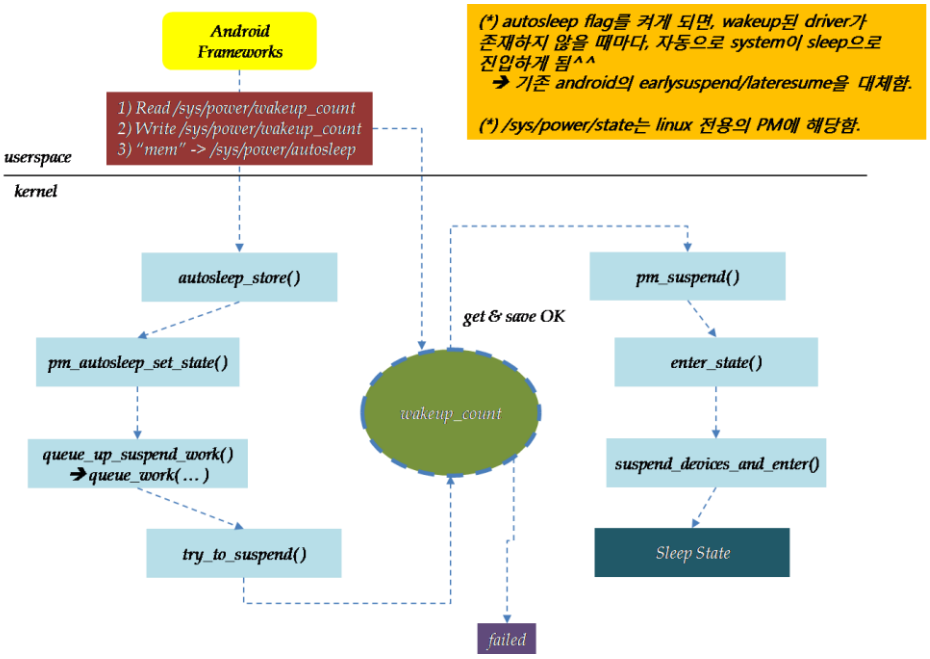

그림 5-16 새로운 System Suspend(= Autosleep) 절차

파워 관리자 프로세스(안드로이드 Power Manager)에서 /sys/power/autosleep에 "mem" 설정한 후, 커널 코드 내에서 suspend에 진입하기 위해 수행하는 코드 절차 중 중요한 부분만을 요약해 보면 다음과 같다.

코드 5-8 AutoSleep에 기초한 Suspend 절차

1) echo "mem" > /sys/power/autosleep

- *active wakeup source가 없을 때 마다 suspend로 진입 !!!)*
- *autosleep_store() 함수 호출됨.*
- *power_attr(autosleep) 매크로 참조* *[in kernel/power/power.h file]*

2) autosleep_store() *[in kernel/power/main.c file]*

3) pm_autosleep_set_state(state) *[in kernel/power/autosleep.c file]*

4) queue_up_suspend_work() *[in kernel/power/autosleep.c file]*

- *queue_work(autosleep_wq, &suspend_work);*
- *DECLARE_WORK(suspend_work, try_to_suspend)*

5) try_to_suspend() *[in kernel/power/autosleep.c file]*

- *pm_get_wakeup_count()*
- *pm_save_wakeup_count()*
- *pm_suspend()*

6) pm_suspend(autosleep_state) *[in kernel/power/suspend.c file]*

- *enter_state()*

7) enter_state() *[in kernel/power/suspend.c file]*

- *freeze_begin()*
- *sys_sync()*
- *suspend_prepare()*
- *suspend_devices_and_enter()*

8) suspend_devices_and_enter() *[in kernel/power/suspend.c file]*

- *suspend_ops->**begin**()*
- *suspend_console()*
- *dpm_suspend_start(PMSG_SUSPEND)*
- ***suspend_enter**()*

9) suspend_enter() *[in kernel/power/suspend.c file]*

- *suspend_ops->**prepare**()*
- *dpm_suspend_end(PMSG_SUSPEND)*
- *suspend_ops->**prepare_late**()*
- *disable_noboot_cpus()*
- *arch_suspend_disable_irqs()*
- *syscore_suspend()*
- ***suspend_ops->enter**()* ← *omap_pm_enter()* *[in arch/arm/mach-omap2/pm.c file]*
  - 여기까지 실행되면, 최종 적으로 CPU power를 내려 주게 됨
  - enter() callback 함수 관련해서는 아래 코드 참조

사용자 영역 Wakelock API

기존 android wakelock과의 호환을 위하여 user space용 wakelock API를 마련해 두었다.

1) 내부 루틴은 wakeup source를 통하여 구현하였으며,

2) Android area에는 partial wake_lock acquire & release만 존재하도록 변경되어야 한다.

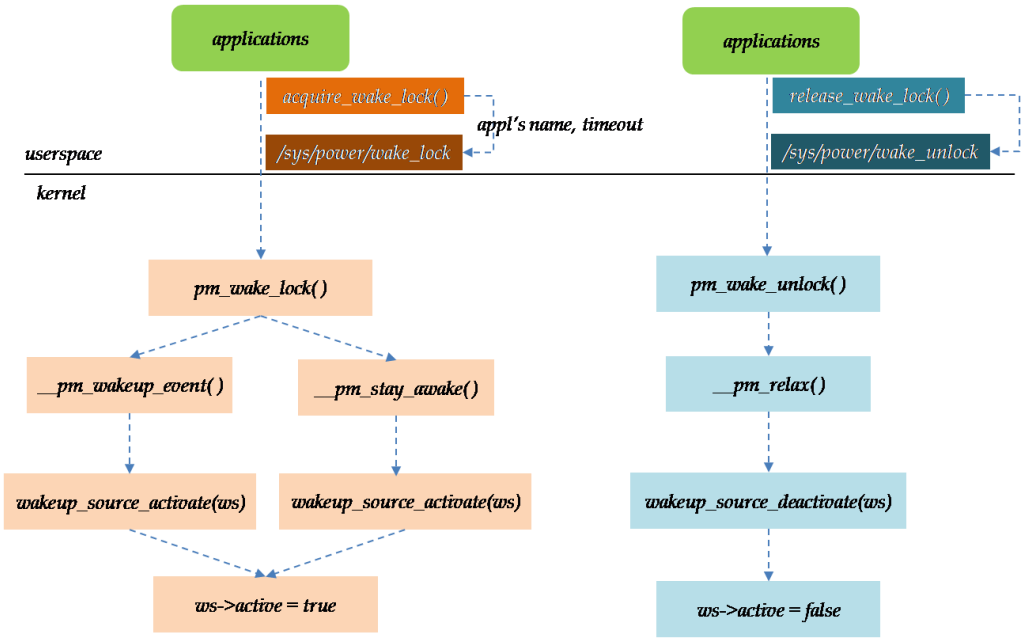

그림 5-17 사용자 영역 용 wakelock의 진화

코드 5-9 wake_lock(), wake_unlock() 커널 함수

```
#define power_attr(_name)                    ₩                    [in kernel/power/power.h file]
static struct kobj_attribute _name##_attr = {    ₩
    .attr    = {                             ₩
        .name = __stringify(_name),          ₩
        .mode = 0644,                        ₩
    },                                       ₩
    .show    = _name##_show,                 ₩
    .store   = _name##_store,                ₩
}

//사용자 영역에서 /sys/power/wake_lock하여 내용 읽어갈 때 호출됨
//wake_lock은 wakelock 획득과 관계됨.
static ssize_t wake_lock_show(struct kobject *kobj,                    [in kernel/power/main.c file]
                              struct kobj_attribute *attr,
                              char *buf)
{
    return pm_show_wakelocks(buf, true);
}

//사용자 영역에서 /sys/power/wake_lock에 내용을 쓸 때 호출됨
//wake_lock은 wakelock 획득과 관계됨.
static ssize_t wake_lock_store(struct kobject *kobj, struct kobj_attribute *attr, const char *buf,
```

```
size_t n)
{
	int error = pm_wake_lock(buf);
	return error ? error : n;
}

power_attr(wake_lock);

//사용자 영역에서 /sys/power/wake_unlock하여 내용 읽어갈 때 호출됨
//wake_unlock은 wakelock 해제와 관계됨.
static ssize_t wake_unlock_show(struct kobject *kobj, struct kobj_attribute *attr, char *buf)
{
	return pm_show_wakelocks(buf, false);
}

//사용자 영역에서 /sys/power/wake_unlock에 내용을 쓸 때 호출됨
//wake_unlock은 wakelock 해제와 관계됨.
static ssize_t wake_unlock_store(struct kobject *kobj, struct kobj_attribute *attr, const char *buf,
size_t n)
{
	int error = pm_wake_unlock(buf);
	return error ? error : n;
}

power_attr(wake_unlock);
```

Wakeup Source API

기존 방식인 kernel wakelock은 그림 5-18과 같이 wakeup source API로 교체되었다. device_init_wakeup() 호출 후, sleep 혹은 wakeup 진입 시점에 pm_relax(), pm_wakeup_event() 함수를 각각 호출해 주면 되는데, 이는 기존에 wakelock 함수를 사용하던 위치와 동일하다. 한편 pm_stay_awake() 함수는 wakeup 상태를 계속 유지하고자 할 경우에 사용한다. 자, 새롭게 소개된 wakeup source API를 기반으로 기존의 wakelock 기반의 디바이스 드라이버를 수정해 보기 바란다. 아래 그림에서도 확인할 수 있는 것처럼, 이는 그리 어려운 작업이 아니다.

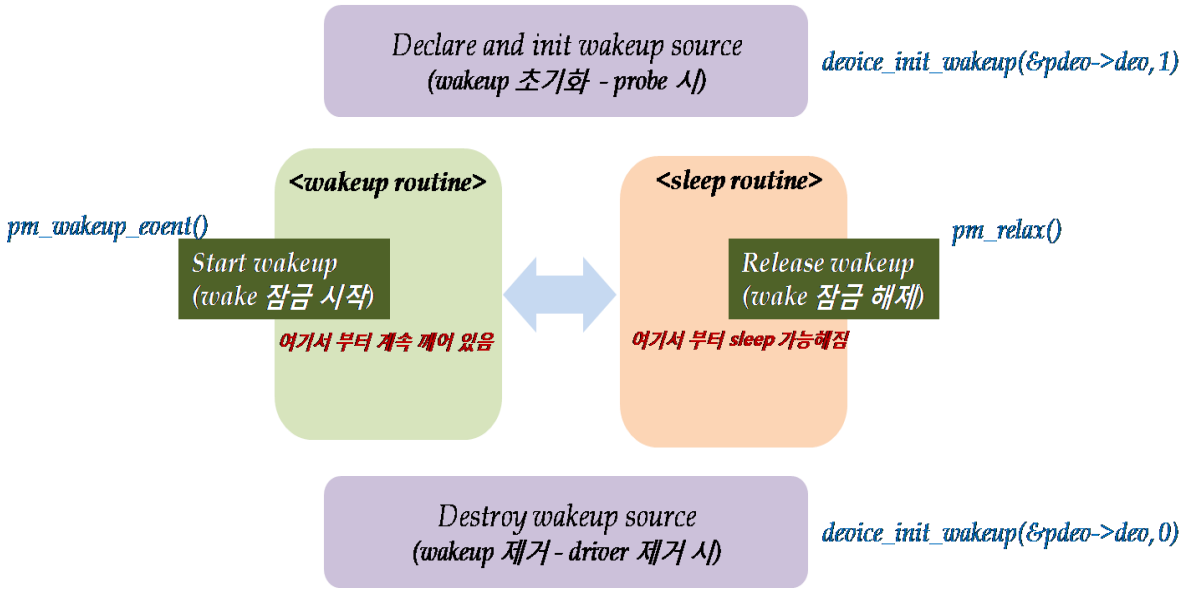

그림 5-18 Kernel wakelock API의 진화

Wakeup source 관련 API를 정리해 보면 다음과 같다.

[Wakeup Source 관련 주요 API 소개 – include/linux/pm_wakeup.h 참조]

*void **wakeup_source_prepare**(struct wakeup_source *ws, const char *name);*
*struct wakeup_source ***wakeup_source_create**(const char *name);*
*void **wakeup_source_drop**(struct wakeup_source *ws);*
*void **wakeup_source_destroy**(struct wakeup_source *ws);*
*void **wakeup_source_add**(struct wakeup_source *ws);*
*void **wakeup_source_remove**(struct wakeup_source *ws);*
*struct wakeup_source ***wakeup_source_register**(const char *name);*
*void **wakeup_source_unregister**(struct wakeup_source *ws);*

- wakeup source internal 함수

*int **device_wakeup_enable**(struct device *dev);*
*int **device_wakeup_disable**(struct device *dev);*
*void **device_set_wakeup_capable**(struct device *dev, bool capable);*
*int **device_init_wakeup**(struct device *dev, bool val);*
*int **device_set_wakeup_enable**(struct device *dev, bool enable);*

- 초기화 함수

*void **__pm_stay_awake**(struct wakeup_source *ws);*
*void **pm_stay_awake**(struct device *dev);*

- *PM core에게 wakeup event가 현재 처리 중임을 알려 줌. 시스템이 suspend로 진입하는 것을 막아주는 함수*

*void **__pm_relax**(struct wakeup_source *ws);*

*void **pm_relax**(struct device *dev);*

- *PM core에게 wakeup event가 처리 완료되었음을 알려 줌. 즉, device가 suspend로 진입할 수 있게 됨.*

*void **__pm_wakeup_event**(struct wakeup_source *ws, unsigned int msec);*

*void **pm_wakeup_event**(struct device *dev, unsigned int msec);*

- *PM core에 wakeup_event를 날려 줌. wakup event를 처리하는 동안에는 시스템이 깨어 있도록 해주는 함수*

Wakeup source API를 사용하는 예제 코드를 살펴 보면 다음과 같다.

코드 5-10 예제 코드 - drivers/input/keyboard/gpio_keys.c

```
static void gpio_keys_gpio_work_func(struct work_struct *work)
{
    struct gpio_button_data *bdata = container_of(work, struct gpio_button_data, work);

    gpio_keys_gpio_report_event(bdata);

    if (bdata->button->wakeup)
        pm_relax(bdata->input->dev.parent);
            //PM core에게 wakeup event가 처리 완료되었음을 알려 줌.
            //즉, device가 suspend로 진입할 수 있게 됨.
}

static irqreturn_t gpio_keys_gpio_isr(int irq, void *dev_id)
{
    struct gpio_button_data *bdata = dev_id;

    BUG_ON(irq != bdata->irq);

    if (bdata->button->wakeup)
        pm_stay_awake(bdata->input->dev.parent);
          // PM core에게 wakeup event가 현재 처리 중임을 알려 줌.
          //시스템이 suspend로 진입하는 것을 막아줌.

    if (bdata->timer_debounce)
        mod_timer(&bdata->timer, jiffies + msecs_to_jiffies(bdata->timer_debounce));
```

```
    else
        schedule_work(&bdata->work);
    return IRQ_HANDLED;
}

static irqreturn_t gpio_keys_irq_isr(int irq, void *dev_id)
{
    struct gpio_button_data *bdata = dev_id;
    const struct gpio_keys_button *button = bdata->button;
    struct input_dev *input = bdata->input;
    unsigned long flags;

    BUG_ON(irq != bdata->irq);

    spin_lock_irqsave(&bdata->lock, flags);

    if (!bdata->key_pressed) {
        if (bdata->button->wakeup)
            pm_wakeup_event(bdata->input->dev.parent, 0);
              // PM core에 wakeup_event를 날려 줌.
              // wakup event를 처리하는 동안에는 시스템이 깨어 있도록 해줌.
        input_event(input, EV_KEY, button->code, 1);
        input_sync(input);
        [...]
    }
    [...]
}
```

마지막으로, 지금까지 2장에서 설명한 System Suspend 기법의 변천 과정을 표로 정리해 보는 것으로 이장을 마치고자 한다.

버전	내용
안드로이드 이전 리눅스	PM Core + suspend
안드로이드(~ kernel 3.4)	PM Core + earlysuspend + suspend + wakelock
리눅스 최신(kernel 3.5 ~)	PM Core + autosleep + suspend + wakelock(wakeup sources)

[주의 사항] 주요 스마트폰 칩 벤더(예: Qualcomm, Samsung, TI)의 경우, 안드로이드 Jelly Bean에서 kernel 3.4 버전을 사용함에도 불구하고, 아직까지는 Autosleep, Wakeup sources 개념을 적용한 PM Core를 활용하지 않고, 기존의 방식(wakelock, earlysuspend, lateresume)을 고수하고 있다.

3. CPU Power Management

3.1 CPUFreq Framework(CPU Frequency Scaling)

CPU Frequency Scaling이란 시스템 부하를 고려하여 CPU 클럭의 속도를 동적으로 변화시키는 것을 뜻한다. 이렇게 하는 이유는 전류 소비와 발열을 줄이기 위해서인데, 주파수(frequency)와 전압(voltage) 값이 동적으로 감소하는 것을 P-State라고 한다. CPU core의 주파수를 조정하기 위해서는 칩에서 기본적으로 이를 지원(예전 CPU의 경우는 이러한 기능이 없었음) 해 줄 수 있어야 하며, 바꾸고자 하는 주파수 상태에서의 latency(overhead) 정보를 알고 있어야 한다.
여기서 한가지 구분해야 할 것은, 실제로 주파수 변경을 담당하는 부분(CPU frequency scaling drivr)과 정책을 결정하는 부분(governor)에 관한 것이다. CPUFreq 기법은 governor를 통해 정책 설정을 하게 되는데, 프로세서의 부하가 늘어나면, 주파수 값을 키워주고, 부하가 줄어들면, 주파수 값을 줄여주는 과정을 반복하게 된다.

그림 5-21은 CPUFreq 기법의 전체 동작 과정을 한눈에 알아 볼 수 있도록 표현한 것인데, 그림에서 볼 수 있듯이 CPUFreq 기법을 구현하기 위해서는 아래의 4가지 기본 요소가 필요하다.

<CPUFreq 기법의 구성 요소>

1) CPUFreq core

- CPUFreq의 핵심 부분으로, 관련 코드는 drivers/cpufreq/cpufreq.c에 위치하고 있다. 이 코드의 대략적인 구성을 보면, 실제 frequency 전환을 담당하는 부분(CPUFreq 아키텍쳐 드라이버 인터페이스)과 notifier로 되어 있음을 알 수 있다.

2) cpufreq_driver structure

- CPU 별로 지정 가능한 cpufreq driver로, init, exit, verify, setpolicy, target 등 CPU에 맞는 실제 frequency 관련 함수로 구성되어 있다.

3) cpufreq_policy structure

- CPU frequency scaling 관련 정책 내용을 담고 있다.

4) cpufreq_governor structure

- 정책을 결정해 주는 주체로, 아래와 같이 여러 가지 governor가 존재한다.

<CPUFreq – 사용 가능한 governor의 종류>

1) powersave

- 항상 가장 낮은 frequency 값을 선택함.

2) performance

- 항상 가장 높은 frequency 값을 선택함.

3) ondemand

- CPU 사용량에 맞추어 frequency 값을 조정함. 즉, CPU idle 상태가 전체 시간 중 20%

미만이면(즉, 매우 바삐 움직이면), frequency 값을 최고로 올려 줌. 그리고, 만일 CPU idle 상태가 30% 이상(조금씩 쉬는 율이 커지면)이면, frequency 값을 5% 씩 줄여줌.

4) conservative

- ondemand와 동일하게 동작함. 단, CPU idle 상태가 전체 시간 중 20% 미만일 경우, 바로 frequency 값을 최고로 올려주는 대신에, 5% 씩 점진적으로 증가시킴.

5) userspace

- frequency 값을 userspace application이 설정하도록 하는 방식.

[참고 사항] 이 밖에도 안드로이드에서 새롭게 추가한 governor가 여럿 존재하지만, 여기에서는 별도로 정리하지 않았다.

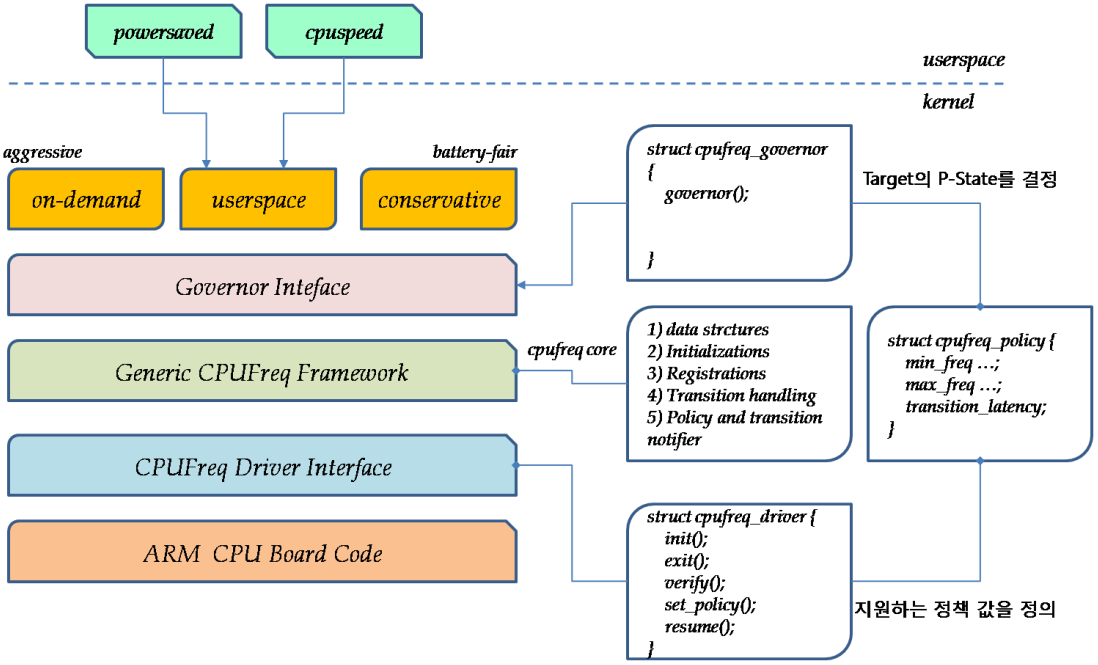

그림 5-21 CPUFreq 파워 관리 기법

cpufreq_driver structure의 구성 요소를 살펴보면 다음과 같다.

코드 5-8 struct cpufreq_driver

```
struct cpufreq_driver {                                    [in include/linux/cpufreq.h]
    struct module     *owner;
    char              name[CPUFREQ_NAME_LEN];   //드라이버 명
    u8                flags;
    bool              have_governor_per_policy;

    /* needed by all drivers */
    int (*init) (struct cpufreq_policy *policy);   // CPU별 초기화 함수 포인터
```

```
    int (*verify) (struct cpufreq_policy *policy);
      // 사용자가 선택한 정책(policy)을 검증하는 함수 포인터

    /* define one out of two */
    int (*setpolicy) (struct cpufreq_policy *policy);
    int (*target) (struct cpufreq_policy *policy, unsigned int target_freq, unsigned int relation);
        //실제로 frequency를 바꿀 때 호출되는 함수 포인터 – setpolicy() 혹은 target()
        //CPU가 오직 한 개의 주파수로만 설정 가능한 경우는 target() 함수 사용함.
        //그렇지 않을 경우에는 setpolicy() 함수를 사용.

    /* should be defined, if possible */
    unsigned int (*get) (unsigned int cpu);

    /* optional */
    unsigned int (*getavg) (struct cpufreq_policy *policy, unsigned int cpu);
    int (*bios_limit) (int cpu, unsigned int *limit);

    int (*exit) (struct cpufreq_policy *policy);
    int (*suspend) (struct cpufreq_policy *policy);
    int (*resume) (struct cpufreq_policy *policy);
    struct freq_attr        **attr;
};
```

CPU frequency 드라이버를 등록/해제하기 위해서는 아래 함수를 사용해야 한다.

*int cpufreq_register_driver(struct cpufreq_driver *driver_data);*
*int cpufreq_unregister_driver(struct cpufreq_driver *driver_data);*

다음으로 cpufreq_governor structure를 살펴보기로 하자.

코드 5-9 struct cpufreq_governor

```
struct cpufreq_governor {                                    [in include/linux/cpufreq.h]
        char    name[CPUFREQ_NAME_LEN];
        int initialized;
        int (*governor) (struct cpufreq_policy *policy, unsigned int event);
          //governor에서 가장 중요한 부분은 이 callback 함수를 구현하는 것임.
        ssize_t (*show_setspeed) (struct cpufreq_policy *policy, char *buf);
        int (*store_setspeed) (struct cpufreq_policy *policy, unsigned int freq);
        unsigned int max_transition_latency; /* HW must be able to switch to
```

```
                                        next freq faster than this value in nano secs or we
                                        will fallback to performance governor */
    struct list_head    governor_list;
    struct module       *owner;
};

struct cpufreq_governor cpufreq_gov_performance = {   [in drivers/cpufreq/cpufreq_performance.c]
    .name       = "performance",
    .governor   = cpufreq_governor_performance,
    .owner      = THIS_MODULE,
};
```

Governor를 등록/해제하기 위해서는 아래 함수를 사용해야 한다.

*int cpufreq_register_governor(struct cpufreq_governor *governor);*
*void cpufreq_unregister_governor(struct cpufreq_governor *governor);*

cpufreq_governor structure에서 가장 중요한 필드는 (*governor) callback 함수인데, governor callback 함수는 최종적으로 아래 함수를 호출하여, CPU frequency 드라이버로 하여금 주파수를 변경하도록 유도한다.

*int cpufreq_driver_target(struct cpufreq_policy *policy, unsigned int target_freq, unsigned int relation);*

CPUFreq 기법을 설명함에 있어, 또 한가지 빼먹지 말아야 할 내용은 아래 두 가지 notifier에 관한 것이다.

1) transition notifier

- CPUFREQ_TRANSITION_NOTIFIER
- Transition notifier callback 함수는 주파수가 바뀔 때마다 두 번씩 호출된다. 각각 CPUFREQ_PRECHANGE(바뀌기 전 event), CPUFREQ_POSTCHANGE(바뀐 후 event)
- 또한, 시스템이 suspend 혹은 resume될 경우에도 한번씩 호출된다.

2) policy notifier

- CPUFREQ_POLICY_NOTIFIER
- Policy notifier callback 함수는 정책이 결정될 때 마다 세 번씩 호출된다. 즉, CPUFREQ_ADJUST, CPUFREQ_INCOMPATIBLE, CPUFREQ_NOTIFY

지금까지 설명한 4가지 기본 구성 요소를 바탕으로, cpufreq driver의 동작 과정을 실제 예제 코드(OMAP 4430 SDP 보드)를 통해 분석해 보기로 한다. cpufreq driver(예: omap-cpufreq.c)는

cpufreq_driver structure의 오퍼레이션(operations)을 구현하고, cpufreq_register_driver() 함수를 사용하여 이를 등록해 주어야 한다. 이후 현재 지정된 정책을 기반으로 frequency를 변경하는 조건 발생(device driver code내에서 발생 혹은 user space application에서 발생) 시, 실제 주파수 값을 변경해 주게 된다.

코드 5-10 예제 코드 - CPUFreq 드라이버 동작 과정

```
MACHINE_START(OMAP_4430SDP, "OMAP4430 4430SDP board")
                                                    [in arch/arm/mach-omap2/board-4430sdp.c file]
    /* Maintainer: Santosh Shilimkar - Texas Instruments Inc */
    .atag_offset    = 0x100,
    .smp            = smp_ops(omap4_smp_ops),
    .reserve        = omap_reserve,
    .map_io         = omap4_map_io,
    .init_early = omap4430_init_early,
    .init_irq     = gic_init_irq,
    .init_machine     = omap_4430sdp_init,
    .init_late    = omap4430_init_late,      ①
                    //arch/arm/kernel/setup.c의 init_machine_late() 함수에서 호출함.
    .init_time    = omap4_local_timer_init,
    .restart      = omap44xx_restart,
MACHINE_END

void __init omap4430_init_late(void)                              [in arch/arm/mach-omap2/io.c file]
{
    omap_common_late_init();      ②
    omap4_pm_init();
    omap2_clk_enable_autoidle_all();
}

static void __init omap_common_late_init(void)
{
    omap_mux_late_init();
    omap2_common_pm_late_init();      ③
    omap_soc_device_init();
}

int __init omap2_common_pm_late_init(void)                        [in arch/arm/mach-omap2/pm.c file]
{
```

```
	/*
	 * In the case of DT, the PMIC and SR initialization will be done using
	 * a completely different mechanism.
	 * Disable this part if a DT blob is available.
	 */
	if (!of_have_populated_dt()) {

		/* Init the voltage layer */
		omap_pmic_late_init();
		omap_voltage_late_init();

		/* Initialize the voltages */
		omap3_init_voltages();
		omap4_init_voltages();

		/* Smartreflex device init */
		omap_devinit_smartreflex();

		/* cpufreq dummy device instantiation */
		omap_init_cpufreq();	④
	}

#ifdef CONFIG_SUSPEND
	suspend_set_ops(&omap_pm_ops);
#endif

	return 0;
}

static inline void omap_init_cpufreq(void)
{
	struct platform_device_info devinfo = { .name = "omap-cpufreq", };
	platform_device_register_full(&devinfo);	⑤ //omap-cpufreq platform device 등록
}

// omap-cpufreq platform driver registration & probe
static struct platform_driver omap_cpufreq_platdrv = {		[in drivers/cpufreq/omap-cpufreq.c file]
	.driver = {
```

```
        .name     = "omap-cpufreq",
        .owner    = THIS_MODULE,
    },
    .probe        = omap_cpufreq_probe,     ⑥   //probe() 함수 선언
    .remove       = omap_cpufreq_remove,
};

static int omap_cpufreq_probe(struct platform_device *pdev)     ⑦   //probe() 함수 호출
{
    mpu_dev = get_cpu_device(0);
    if (!mpu_dev) {
        pr_warning("%s: unable to get the mpu device₩n", __func__);
        return -EINVAL;
    }

    mpu_reg = regulator_get(mpu_dev, "vcc");
    if (IS_ERR(mpu_reg)) {
        pr_warning("%s: unable to get MPU regulator₩n", __func__);
        mpu_reg = NULL;
    } else {
        /*
         * Ensure physical regulator is present.
         * (e.g. could be dummy regulator.)
         */
        if (regulator_get_voltage(mpu_reg) < 0) {
            pr_warn("%s: physical regulator not present for MPU₩n",
                __func__);
            regulator_put(mpu_reg);
            mpu_reg = NULL;
        }
    }

    return cpufreq_register_driver(&omap_driver);   //cpufreq_driver를 등록     ⑧
}

static struct cpufreq_driver omap_driver = {                    [in drivers/cpufreq/omap-cpufreq.c file]
    .flags        = CPUFREQ_STICKY,
    .verify       = omap_verify_speed,
```

```
    .target     = omap_target,      //실제로 frequency를 바꿀 때 호출되는 함수 포인터
    .get        = omap_getspeed,
    .init       = omap_cpu_init,
    .exit       = omap_cpu_exit,
    .name       = "omap",
    .attr       = omap_cpufreq_attr,
};

struct cpufreq_governor cpufreq_gov_performance = {    [in drivers/cpufreq/cpufreq_performance.c]
    .name       = "performance",
    .governor   = cpufreq_governor_performance,     ⑨ //governor callback 정의
    .owner      = THIS_MODULE,
};

static int cpufreq_governor_performance(struct cpufreq_policy *policy, unsigned int event)
{
    switch (event) {
    case CPUFREQ_GOV_START:
    case CPUFREQ_GOV_LIMITS:
        __cpufreq_driver_target(policy, policy->max, CPUFREQ_RELATION_H);     ⑩
        break;
    default:
        break;
    }
    return 0;
}

int __cpufreq_driver_target(struct cpufreq_policy *policy,          [in drivers/cpufreq/cpufreq.c file]
                unsigned int target_freq,
                unsigned int relation)
{
  [...]
  if (cpufreq_driver->target)
        retval = cpufreq_driver->target(policy, target_freq, relation);     ⑪
          //cpufreq driver의 target callback 함수 호출
  [...]
}
```

```
//실제로 frequency를 바꿔 줌.  ⑫
static int omap_target(struct cpufreq_policy *policy, unsigned int target_freq, unsigned int relation)  [in drivers/cpufreq/omap-cpufreq.c file]
{
    [..]
    struct cpufreq_freqs freqs;
    [...]
    ret = cpufreq_frequency_table_target(policy, freq_table, target_freq, relation, &i);
    [...]
    freqs.new = freq_table[i].frequency;
    [...]
    ret = clk_set_rate(mpu_clk, freqs.new * 1000);
    [...]
    freqs.new = omap_getspeed(policy->cpu);
    [...]
}
```

위의 코드 5-10의 주요 코드 흐름을 정리하면 다음과 같다.

1) .init_late callback 정의(선언)
   - .init_late = omap4430_init_late
2) omap4430_init_late() 함수에서 omap_common_late_init() 함수 호출
3) omap_common_late_init() 함수에서 omap2_common_pm_late_init() 함수 호출
4) omap2_common_pm_late_init() 함수에서 omap_init_cpufreq() 함수 호출
5) omap_init_cpufreq() 함수에서 omap-cpufreq platform device 등록
   - platform_device_register_full(&devinfo)
6) omap_cpufreq_platdrv structure 내에서 probe 함수 선언
   - .probe = omap_cpufreq_probe
7) 드라이버 probing 과정에서 omap_cpufreq_probe() 함수 호출
8) omap_cpufreq_probe() 함수에서 cpufreq_driver 등록
   - cpufreq_register_driver(&omap_driver)
   - 등록하기 위해서는 이미 struct cpufreq_driver omap_driver를 선언해 주어야 함
9) cpufreq_gov_performance structure 내에서 governor callback 정의
   - .governor = cpufreq_governor_performance
10) cpufreq_governor_performance() 함수에서 __cpufreq_driver_target() 함수 호출
11) __cpufreq_driver_target() 함수에서 cpufreq_driver->target() 함수 호출
12) cpufreq_driver->target()은 omap_target()을 의미함. 이 함수 내에서 실제 주파수 변경 작업 시작함.

3.2 CPUIdle Framework

대부분의 시스템에서 프로세서는 실제로 idle 상태(할 일이 없는 상태)로 보내는 시간이 많다. 할 일이 없을 때, 프로세서는 자신이 다시 필요해 질 때까지 idle 상태로 빠지게 되는데, idle 상태는 우선 순위가 낮은 thread(swapper)를 실행하는 상태를 말하며, 이 상태에서는 실행해야 할 다른 task가 존재할 때가지 아무 하는 일 없이 무한 loop을 돌게 된다.

그런데, 아무 하는 일 없이 무한 대기 loop(busy-wait loop)에 빠지는 것 역시 파워를 소비하는 일이다. 따라서 CPU 설계자들은 프로세서가 아무런 할 일이 없을 때, 프로세서를 저 전력(lower-power) 상태로 빠질 수 있게끔 하는 방법을 마련하였다. 일반적으로 이러한 상태로 빠질 경우에 CPU는 clock을 멈추고, 새로운 인터럽트가 걸릴 때까지 CPU 회로(circuitry) 일부 혹은 전체를 내려, 파워 소비를 막을 수 있도록 한다.

실제로, 대부분의 CPU는 idle 상태를 효과적으로 보내는 방법을 가지고 있는데, 이러한 idle 상태를 "C states"라고 부르며, 절약할 수 있는 파워 량, 잃어 버릴 지도 모르는 부가 정보의 량, 혹은 정상 동작 모드로 돌아오는데 걸리는 시간 등에 따라서 아래와 같은 3가지 모드로 분류하게 된다.

	C1	C2	C3
종료 지연시간 (μs)	1	1	57
소모 전류(mW)	1000	500	100

이 중, C1은 프로세서 clock을 끄기만 하는 상태이며, C2는 시스템 내의 다른 clock도 함께 내리는 상태를 말한다. 마지막으로 C3는 CPU의 일부분을 완전히 내리는 상태를 말한다. 따라서 가능한 한 C3 상태에서 오래 머물러 있는 것이 파워 소비를 줄이는데 유리하다고 볼 수 있다.

그렇다면, C3 상태만을 사용하면 될 것인데, 왜 다른 두 개의 상태가 필요한 것일까 ? 그 이유는 C3 상태를 유지하기 위해서는 추가 비용(cost)이 들기 때문이다. 57μs 의 종료 지연시간 (exit latency)이 든다는 것은 꽤 긴 시간 동안 시스템이 아무 하는 일 없이 보내야 한다는 것을 의미한다(낭비라는 얘기). 프로세서를 정상으로 복귀시키기 위해서는 그 자체적으로도 파워를 소비하게 되며, L2 cache를 비워야(flushing)할 수도 있으므로, 정보 손실을 가져올 수도 있다. 따라서 C3로 진입하기 위해서는 확실히 파워를 절약할 수 있다고 판단되는 경우와 57μs 내에 프로세서가 처리해야 할 아무런 일로 발생하지 않을 것이라는 것을 알고 있을 경우이다. 이러한 조건이 충족되지 않을 경우, C1이나 C2 상태로 진입하여야 하며, 그러한 결정은 CPUIdle 서브시스템에서 판단하게 된다.
각각의 프로세서 마다 idle상태를 규정하는 방법이 서로 다르며, 따라서 idle 상태에 진입하거나 빠져 나오기 위한 서로 다른 방법이 존재하게 된다. CPUIdle 코드는 이러한 복잡함을 해결하기 위해 별도의 드라이버 계층(cpuidle drivers)을 마련해 두었다. 한편 현재 상태가 어느 idle 상태에서 적합한지를 결정하는 것은 정책(policy) 설정을 어떻게 하느냐에 따라 달라지게 되는데, 이를 cpuidle governor라고 한다.

그림 5-22는 CPUIdle 프레임워크의 전체 구조를 하나의 그림으로 표현한 것이다.

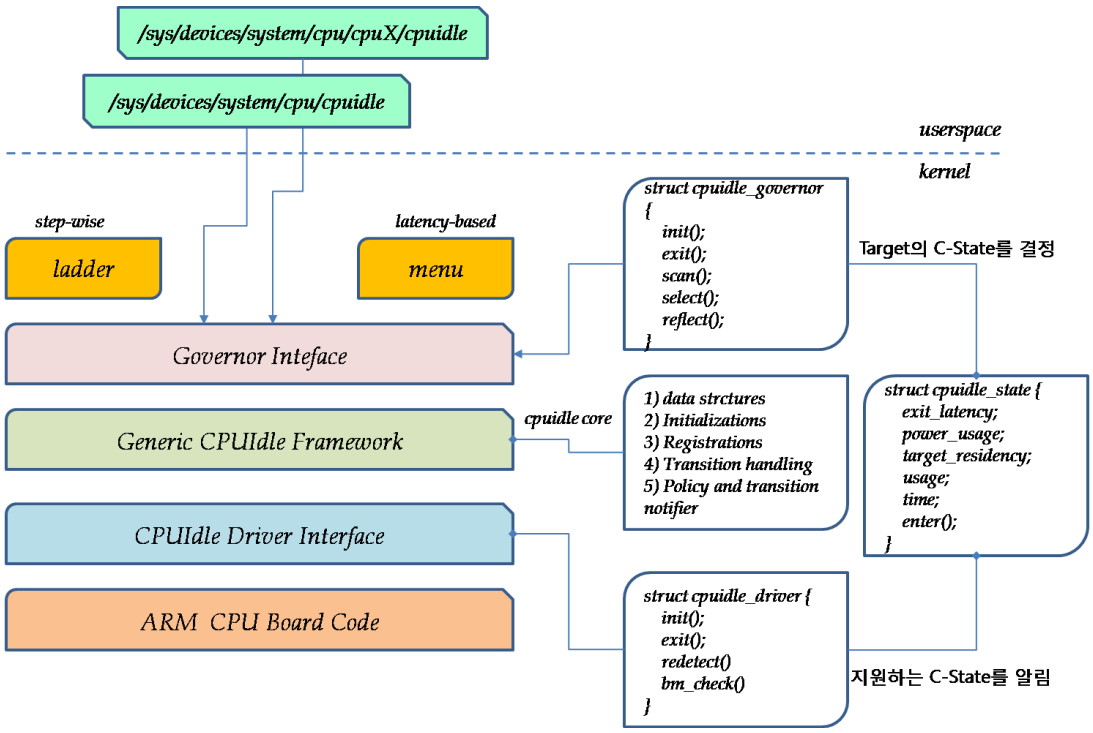

그림 5-22 CPUIdle 파워 관리 기법

이제부터는 CPUIdle 드라이버에 관하여 자세히 살펴 볼 것이다. CPUIdle 드라이버 인터페이스는 꽤나 간단하다. 즉, 아래의 함수를 써서 드라이버를 등록해 주기만 하면 된다.

코드 5-10 struct cpuidle_driver

```
struct cpuidle_driver {                                    [in include/linux/cpuidle.h file]
    char                name[CPUIDLE_NAME_LEN];
    struct module       *owner;
};

int cpuidle_register_driver(struct cpuidle_driver *drv);
```

이렇게 하면 드라이버 이름이 sysfs에 생성되며, CPUIdle 드라이버가 시스템 내에 추가되게 된다. 일단, CPUIdle 드라이버가 존재하게 되면, 각각의 CPU에 맞는 cpuidle device를 등록해 줄 수가 있다. CPU별로 서로 다른 설정이 가능한데, 제일 먼저 해야 할 일은 cpu idle state를 기술하는 것이며, 이는 아래의 struct cpuidle_state를 통해 이루어진다.

코드 5-11 struct cpuidle_state

```
struct cpuidle_state {                                     [in include/linux/cpuidle.h file]
    char                name[CPUIDLE_NAME_LEN];
```

```
        char                    desc[CPUIDLE_DESC_LEN];
        void                    *driver_data;

        unsigned int flags;
        unsigned int exit_latency; /* in US */
        unsigned int power_usage; /* in mW */
        unsigned int target_residency; /* in US */

        unsigned long long      usage;
        unsigned long long      time; /* in US */

        int (*enter)    (struct cpuidle_device *dev, struct cpuidle_state *state);
        [...]
};
```

위의 structure에서 name과 desc 필드는 cpuidle 상태를 묘사해주며, sysfs 항목으로 나타나게 된다. 또한 driver_data 필드는 드라이버에서 필요한 private data를 위해 사용되며, flags 이하의 4개의 필드(flags, exit_latency, power_usage, target_residency)는 sleep 상태의 특징을 표현해 준다. flags에서 사용 가능한 값을 정리하면 다음과 같다.

- ***CPUIDLE_FLAG_TIME_VALID***
- ***CPUIDLE_FLAG_CHECK_BM***
- ***CPUIDLE_FLAG_POLL***
- ***CPUIDLE_FLAG_SHALLOW***
- ***CPUIDLE_FLAG_BALANCED***
- ***CPUIDLE_FLAG_DEEP***

Sleep 상태의 깊이(depth)는 나머지 필드에 의해 표현이 되는데, 먼저 exit_latency는 CPU가 정상 동작 상태(fully functional state)로 바뀌는데 걸리는 시간을 나타내며, power_usage는 현재 상태에 있을 때 CPU가 소비하는 파워 량을 나타낸다. 마지막으로 target_residency는 상태 전환이 의미를 가질 수 있도록 CPU가 소비하는 시간의 최소량을 뜻한다.

마지막으로 enter() 함수는 governor에 의하여 CPU의 상태가 결정되는 순간에 호출되는 함수로, 이 함수를 통해 실제 상태 변화(state transition)가 이루어지게 된다. usage 필드는 상태 진입(enter) 횟수를 기록하는데 사용되며, time 필드는 현재 상태에서 사용한 시간을 기록하는데 쓰인다.

CPUIdle 드라이버는 각각의 CPU에 대해 cpuidle_device structure의 각 필드를 채워 넣어 주어야 한다.

코드 5-12 struct cpuidle_device

```
struct cpuidle_device {                                    [in include/linux/cpuidle.h file]
        unsigned int            cpu;

        int                     last_residency;
        int                     state_count;
        struct cpuidle_state    states[CPUIDLE_STATE_MAX];
        struct cpuidle_state    *last_state;

        void                    *governor_data;
        struct cpuidle_state    *safe_state;
        [...]
};
```

위의 structure에서 state_count 필드에는 유효한 상태의 개수를 넣어 주어야 하며, cpu 필드에는 CPU의 개수를 넣어 주어야 한다. 또한 safe_state 필드는 DMA 수행 중에도 안전하게 진입할 수 있는 sleep 상태를 가리키는 값(포인터)으로 채워 주어야 한다. cpuidle device는 아래 함수를 사용하여 등록해 주어야 한다. 이 함수는 성공 시 0을, 에러 발생 시 음수 값을 되돌려 준다.

*int cpuidle_register_device(struct cpuidle_device *dev);*

드라이버가 해야 할 필요가 있는 다른 것으로는 상태 전환 코드를 구현하는 일일 것이다. 위에서 본 것처럼, 이는 각각의 상태와 결합된 enter() 함수를 통해 가능해 진다.

*int (*enter)(struct cpuidle_device *dev, struct cpuidle_state *state);*

CPUIdle 드라이버를 등록하는 과정을 다음 예제로 알아보기로 하자.

코드 5-14 예제 코드 - CPUIdle

```
static int omap_enter_idle_simple(struct cpuidle_device *dev, struct cpuidle_driver *drv, int index)
                                                    [in arch/arm/mach-omap2/cpuidle44xx.c file]
{
      omap_do_wfi();
      return index;
}

static int omap_enter_idle_coupled(struct cpuidle_device *dev, struct cpuidle_driver *drv, int index)
{
```

```
    struct idle_statedata *cx = state_ptr + index;
    [...]
    cpu_pm_enter();

    omap4_enter_lowpower(dev->cpu, cx->cpu_state);
    cpu_done[dev->cpu] = true;
    [...]
}

static struct cpuidle_driver omap4_idle_driver = {      ①    //cpuidle driver 선언
      .name                  = "omap4_idle",
      .owner                 = THIS_MODULE,
      .states = {
          {
              /* C1 - CPU0 ON + CPU1 ON + MPU ON */
              .exit_latency = 2 + 2,
              .target_residency = 5,
              .flags = CPUIDLE_FLAG_TIME_VALID,
              .enter = omap_enter_idle_simple,      Ⓓ
              .name = "C1",
              .desc = "CPUx ON, MPUSS ON"
          },
          {
              /* C2 - CPU0 OFF + CPU1 OFF + MPU CSWR */
              .exit_latency = 328 + 440,
              .target_residency = 960,
              .flags = CPUIDLE_FLAG_TIME_VALID | CPUIDLE_FLAG_COUPLED |
                       CPUIDLE_FLAG_TIMER_STOP,
              .enter = omap_enter_idle_coupled,      Ⓓ
              .name = "C2",
              .desc = "CPUx OFF, MPUSS CSWR",
          },
          {
              /* C3 - CPU0 OFF + CPU1 OFF + MPU OSWR */
              .exit_latency = 460 + 518,
              .target_residency = 1100,
              .flags = CPUIDLE_FLAG_TIME_VALID | CPUIDLE_FLAG_COUPLED |
                       CPUIDLE_FLAG_TIMER_STOP,
```

```
                .enter = omap_enter_idle_coupled,      Ⓓ
                .name = "C3",
                .desc = "CPUx OFF, MPUSS OSWR",
            },
        },
        .state_count = ARRAY_SIZE(omap4_idle_data),
        .safe_state_index = 0,
};

int __init omap4_idle_init(void)
{
    [...]
    return cpuidle_register(&omap4_idle_driver, cpu_online_mask); //cpuidle 드라이버 등록   ②
}

================================================================

void arch_cpu_idle(void)                                    [in arch/arm/kernel/process.c]
{
      if (cpuidle_idle_call())      Ⓐ     //kernel idle loop에서 호출
            default_idle();
}

int cpuidle_idle_call(void)
{
    [...]
    if (cpuidle_state_is_coupled(dev, drv, next_state))
        entered_state = cpuidle_enter_state_coupled(dev, drv, next_state);
    else
        entered_state = cpuidle_enter_state(dev, drv, next_state);      Ⓑ
    [...]
}

int cpuidle_enter_state(struct cpuidle_device *dev, struct cpuidle_driver *drv, int index)
                                                        [in drivers/cpuidle/cpuidle.c file]
{
    struct cpuidle_state *target_state = &drv->states[index];
```

```
    [...]
    entered_state = target_state->enter(dev, drv, index);    ©
    [...]
}
```

위의 코드 5-14의 주요 코드 흐름을 정리하면 다음과 같다.

<cpuidle 드라이버 정의 및 등록>

1) cpuidle driver 선언
   - struct cpuidle_driver omap4_idle_driver
   - 이 structure 내에서 enter() callback 함수를 사전에 정의해 둠.

2) omap4_idle_init() 함수에서 cpuidle 드라이버 등록
   - cpuidle_register(&omap4_idle_driver, cpu_online_mask)

<Kernel idle loop – main routine>

A) arch_cpu_idle() 함수에서 cpuidle_idle_call() 함수 호출

B) cpuidle_idle_call() 함수에서 cpuidle_enter_state() 함수 호출

C) cpuidle_enter_state() 함수에서 target_state->enter() 함수 호출

D) 위의 1) 단계에서 선언한 cpuidle driver structure 내의 enter() callback 함수 호출됨
   - .enter = omap_enter_idle_simple
   - .enter = omap_enter_idle_coupled
   - .enter = omap_enter_idle_coupled

4. I/O Runtime 파워 관리 기법

I/O Runtime PM은 Kevin Hilman과 Paul Walmsley가 초기에 많은 논의를 하였으며, 최종적으로는 Rafael Wysocki에 의해 kernel 2.6.32에 통합(android wakelock 보다 나중에 추가된 기법임) 되었다. I/O Runtime PM이 필요한 이유는 앞서 2장에서 소개한 system suspend(정해진 순간에 일괄적으로 suspend로 진입)의 한계 즉, system suspend 중에 특정 device가 suspend 조건에 부합하지 못하여, system이 suspend 상태로 진입하지 못하는 문제를 해결하기 위해서이다. I/O Runtime 파워 관리의 필요성을 요약해 보면 다음과 같다.

1) System sleep 만으로는 실시간 에너지 소비를 줄이는데 한계가 있다.

2) Device 간에는 상호 의존 관계가 있을 수 있다.

3) Idle 조건을 해결하는데 도움이 된다.

4) CPUIdle 시에, I/O runtime PM을 처리할 수 있는 것은 아니다. 즉, CPU PM과 I/O device PM이

서로 다르게 운용될 수 있는 상황이 존재한다.

I/O Runtime PM은 CPU(이 경우는 cpuidle 프레임워크가 있음)를 위한 것이 아니라, I/O device를 위해 마련된 기법으로 이를 사용하는 드라이버를 예로 들어보면 다음과 같다.

- *Usb subsystem HUB, usb mass storage, UVC, HID, CDC, serial, usb net, ...*
- *PCI subsystem e1000e, rtl8169, ehci-pci, uhci-pci, ...*
- *SCSI subsystem sd*
- *I2C subsystem*
- *MMC subsystem*
- *Serial devices*
- *Misc device(gpio, key,.....)*
- *ARCHs(RPM via dev_pm_domain)*

그림 5-23은 I/O Runtime PM을 표현한 그림으로, I/O Runtime PM의 주체가 각각의 device임을 보여준다.

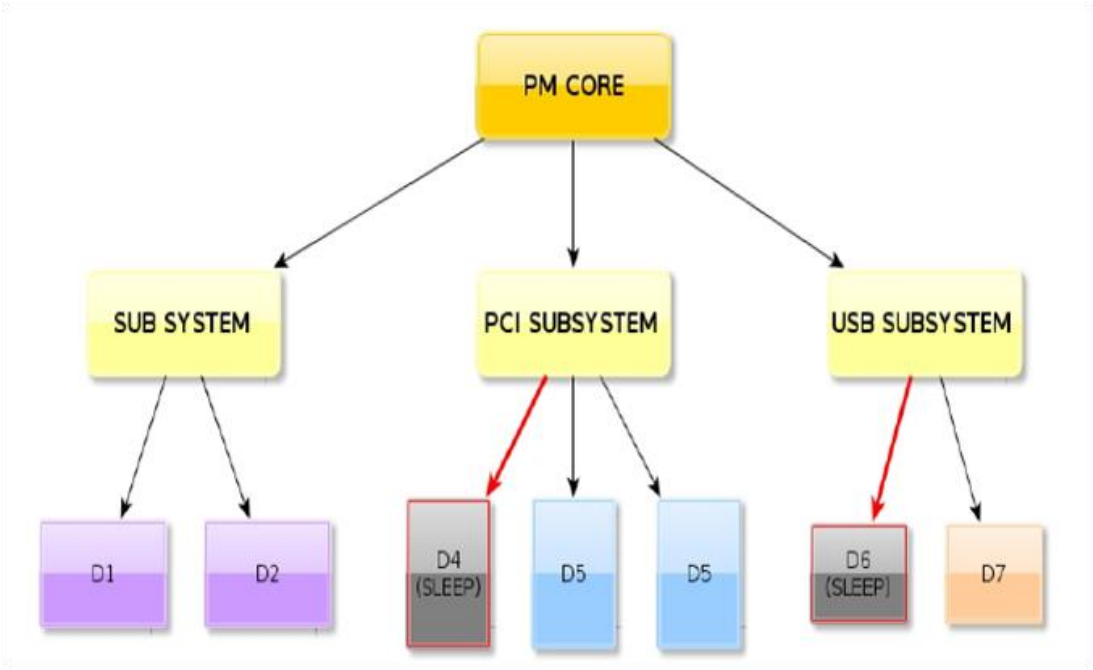

그림 5-23 I/O Run-time PM의 개요 – 디바이스 드라이버 별로 Sleep [출처 – 참고문헌 11]

I/O Runtime PM framework에서 사용하는 파워 상태를 정리해 보면 다음과 같다.

1) ACTIVE *– Full-power 상태로, 장치는 I/O가 가능하다.*

2) SUSPENDED *– Low-power 상태이며, 장치는 I/O를 진행할 수 없다.*

3) SUSPENDING *– ACTIVE에서 SUSPENDED로 전이 중인 상태를 말한다.*

4) RESUMING *- SUSPENDED에서 ACTIVE로 전이 중인 상태를 말한다.*

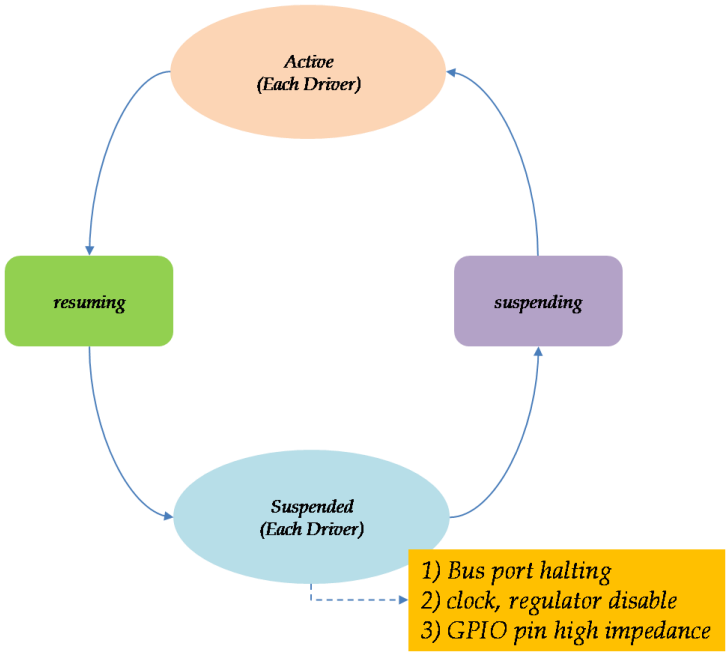

그림 5-24 I/O Run-time PM의 상태 전이도

IO Runtime PM 관련 sysfs 인터페이스를 정리해 보면 다음과 같다.

/sys/devices/.../power/control

on – device가 항상 ACTIVE 상태이다(default).

auto – device의 상태를 바꿀 수 있다.

/sys/devices/.../power/runtime status (read-only, 2.6.36 material)

active – device가 ACTIVE 상태이다.

suspended – device가 SUSPENDED 상태이다.

suspending – device 상태가 ACTIVE에서 SUSPENDED로 바뀌는 중이다.

resuming – device 상태가 SUSPENDED에서 ACTIVE로 바뀌는 중이다.

error – runtime PM 동작이 실패했다.

unsupported – device에 대한 runtime PM이 설정되어 있지 않다.

다시 한번 말하지만, I/O Runtime power management란 I/O 장치들에 대해 사용 중(runtime)에 low-power state로 만들거나, wake-up 시키는 것을 말한다. PM core는 아래의 callback 함수를 정의(등록)한 드라이버에 대해 작업을 수행한다.

코드 5-15 struct dev_pm_ops

```
struct dev_pm_ops {                                    [in include/linux/pm.h file]
    [...]
    int (*runtime_suspend)(struct device *dev);
    int (*runtime_resume)(struct device *dev);
    int (*runtime_idle)(struct device *dev);
```

```
};

struct device_driver {                                    [in include/linux/device.h file]
        [...]
        const struct dev_pm_ops *pm;
        ...
};

struct struct bus_type {
        [...]
        const struct dev_pm_ops *pm;
        [...]
};
```

I/O Run-time PM 관련 주요 API를 소개하면 다음과 같다.

1) 장치의 상태(suspend or resume)를 바꾸는 함수

<Suspend 함수>

*int **pm_runtime_suspend**(struct device *dev);*

*int **pm_schedule_suspend**(struct device *dev, unsigned int delay);*

<Resume 함수>

*int **pm_runtime_resume**(struct device *dev);*

*int **pm_request_resume**(struct device *dev);*

<idle 상태 통보(Notification) 함수>

*int **pm_runtime_idle**(struct device *dev);*

*int **pm_request_idle**(struct device *dev);*

2) device를 사용 중임을 나타내는 참조 카운트를 지정하는 함수(참조 카운트 값이 설정된 경우에는 suspend 상태로 진입할 수 없음)

<참조(reference) 카운트 값 증가 함수(Taking a reference)>

*int **pm_runtime_get**(struct device *dev); /* + resume request */*

*int **pm_runtime_get_sync**(struct device *dev); /* + sync resume */*

*int **pm_runtime_get_noresume**(struct device *dev);*

<참조(reference) 카운트 값 감소 함수(Dropping a reference)>

*int **pm_runtime_put**(struct device *dev); /* + idle request */*

*int **pm_runtime_put_sync**(struct device *dev); /* + sync idle */*

*int **pm_runtime_put_noidle**(struct device *dev);*

3) 주요 API 사용법

a) Probe 시

pm_runtime_enable()

/ probe/configure hardware */*

pm_runtime_suspend()

(참고) remove() 함수에서는 pm_runtime_disable() 함수를 호출함.

b) Activity(실제 코드 내) 안에서

pm_runtime_get() / reference count 증가 */*

/ Do work */*

pm_runtime_put() / reference count 감소 */*

c) Done(device 사용을 마치고, suspend 진입 시)

pm_runtime_suspend()

4) 지금까지 소개한 I/O Runtime PM API를 정리하면 다음과 같다.

From include/linux/pm_runtime.h

*static inline int pm_runtime_idle(struct device *dev)*
*static inline int pm_runtime_suspend(struct device *dev)*
*static inline int pm_runtime_autosuspend(struct device *dev)*
*static inline int pm_runtime_resume(struct device *dev)*
*static inline int pm_request_idle(struct device *dev)*
*static inline int pm_request_resume(struct device *dev)*
*static inline int pm_request_autosuspend(struct device *dev)*
*static inline int pm_runtime_get(struct device *dev)*
*static inline int pm_runtime_get_sync(struct device *dev)*
*static inline int pm_runtime_put(struct device *dev)*
*static inline int pm_runtime_put_autosuspend(struct device *dev)*
*static inline int pm_runtime_put_sync(struct device *dev)*
*static inline int pm_runtime_put_sync_suspend(struct device *dev)*
*static inline int pm_runtime_put_sync_autosuspend(struct device *dev)*
*static inline int pm_runtime_set_active(struct device *dev)*
*static inline void pm_runtime_set_suspended(struct device *dev)*
*static inline void pm_runtime_disable(struct device *dev)*
*static inline void pm_runtime_use_autosuspend(struct device *dev)*

*static inline void pm_runtime_dont_use_autosuspend(struct device *dev)*

아래 코드는 I/O Runtime PM API를 사용하는 keypad 드라이버를 예를 든 것이다.

코드 5-16 예제 코드 - input/keyboard/samsung-keypad.c

```
static int samsung_keypad_probe(struct platform_device *pdev)
                                        [in input/keyboard/samsung-keypad.c file]
{
        const struct samsung_keypad_platdata *pdata;
        const struct matrix_keymap_data *keymap_data;
        struct samsung_keypad *keypad;
        struct resource *res;
        struct input_dev *input_dev;
        unsigned int row_shift;
        unsigned int keymap_size;
        int error;
        [...]
        device_init_wakeup(&pdev->dev, pdata->wakeup);
        platform_set_drvdata(pdev, keypad);
        pm_runtime_enable(&pdev->dev);
        [...]
err_disable_runtime_pm:
        pm_runtime_disable(&pdev->dev);
        device_init_wakeup(&pdev->dev, 0);
        platform_set_drvdata(pdev, NULL);
err_unprepare_clk:
        clk_unprepare(keypad->clk);
        return error;
}

static irqreturn_t samsung_keypad_irq(int irq, void *dev_id)
{
        struct samsung_keypad *keypad = dev_id;
        unsigned int row_state[SAMSUNG_MAX_COLS];
        unsigned int val;
        bool key_down;

        pm_runtime_get_sync(&keypad->pdev->dev);
```

```
    do {
        val = readl(keypad->base + SAMSUNG_KEYIFSTSCLR);
        /* Clear interrupt. */
        writel(~0x0, keypad->base + SAMSUNG_KEYIFSTSCLR);

        samsung_keypad_scan(keypad, row_state);

        key_down = samsung_keypad_report(keypad, row_state);
        if (key_down)
            wait_event_timeout(keypad->wait, keypad->stopped, msecs_to_jiffies(50));

    } while (key_down && !keypad->stopped);

    pm_runtime_put(&keypad->pdev->dev);

    return IRQ_HANDLED;
}
```

5. Regulator Framework

지금까지는 Runtime PM과 System Suspend의 개념을 소개하였다. 여기서 한가지 집고 넘어가야 할 사실은 이들 기법은 suspend 혹은 resume으로 전환될 때 호출되어야 할 callback 함수를 정의하는 일과, 실제로 상태 변화가 일어나기 위해서 호출되어야 하는 API를 제공하는 것(커널 프로그래머는 이 API를 드라이버의 적당한 곳에서 호출해 주어야 함)이 주 임무라고 볼 수 있다. 다시 말해, 실제로 suspend로 진입하여 시스템의 소모 전류를 줄이기 위해서는 실질적으로 하드웨어를 제어하는 action(callback 함수 등에서 이를 구현해야 함)이 이루어져야 하는데, 이러한 내용에 해당하는 것들을 정리해 보면 다음과 같다. 본 절에서는 이 중, Regulator와 관련한 내용을 상세히 소개해 보고자 한다.

1) Clock 조정(Chapter 4에서 이미 소개하였음)
2) Regulator 조정
3) PMIC(Power Management IC)
- *무수히 많은 regulator와 다른 subsystem을 포함하고 있으며, vendor 마다 고유의 칩 사용함)*

4) GPIO 제어
…

본론으로 들어가기에 앞서 regulator를 쉽게 이해할 수 있도록 몇 가지 중요한 용어를 정리하고 넘어가고자 한다.

표 5-2 Regulator 관련 주요 용어 정리

용어	내용 설명
Regulator	다른 장치(device)에 파워를 제공(공급)하는 장치를 말함. 입력 전압 -> Regulator -> 출력 전압
PMIC	Power Management IC. 내부에는 수많은 regulator와 다른 subsystem으로 구성되어 있음.
Consumer	Regulator로부터 파워를 공급받는 장치를 말함. Consumer는 아래의 두 가지 유형으로 분류가 됨. **Static**: consumer가 직접 자신의 supply voltage나 current limit 값을 바꿀 수 없음. 오로지 자신의 power supply를 enable 혹은 disable하기만 함. Supply voltage는 하드웨어, 부트로더, 펌웨어 혹은 커널 보드 초기화 코드에 의해서 설정됨. **Dynamic**: consumer는 자신의 supply voltage나 current limit 값을 요구 사항에 맞게 바꿀 수 있음.
Power Domain	자신의 입력 파워를 regulator나 switch 혹은 다른 power domain의 출력 파워에 의해 제공받은 전자 회로를 말함. Supply regulator가 switch의 뒤에 배치되는 경우도 있다. 즉, Regulator -+-> Switch-1 -+-> Switch-2 --> [Consumer A] | | | +-> [Consumer B], [Consumer C] | +-> [Consumer D], [Consumer E] 위의 배치도는 1개의 regulator와 3개의 power domain으로 구성되어 있다: Domain 1: Switch-1, Consumers D & E. Domain 2: Switch-2, Consumers B & C. Domain 3: Consumer A. 이들간의 파워 공급의 관계는 아래와 같다: Domain-1 --> Domain-2 --> Domain-3. Power domain이 다른 regulator로부터 파워를 공급받는 regulator를 포함하는 경우도 있다. 즉, Regulator-1 -+-> Regulator-2 -+-> [Consumer A]

	| +-> [Consumer B] 이 경우는 2개의 regulator와 2개의 power domain으로 구성된 예이다: Domain 1: Regulator-2, Consumer B. Domain 2: Consumer A. 이 경우의 파워 공급 관계는 아래와 같다: Domain-1 --> Domain-2
Constraints	Constraints는 성능이나, 하드웨어 보호를 위한 목적으로 power level을 정의하기 위해 사용되는 개념으로, 3가지 power level이 존재한다. **Regulator Level**: regulator datasheet에 명시되어 있는 regulator 동작 파라미터에 의해 정의된 수준을 말한다. 예를 들어, - voltage 출력은 800mV ~ 3500mV 범위에 있다. - regulator current output limit은 20mA @ 5V ~ 10mA @ 10V 이다. **Power Domain Level**: 이 값은 커널 보드 초기화 코드에서 소프트웨어 적으로 정의되며, 특정 파워 범위로 power domain을 제한하기 위해 사용된다. 예를 들어, - Domain-1 voltage은 3300mV 이다. - Domain-2 voltage는 1400mV ~ 1600mV 이다. - Domain-3 current limit은 0mA ~ 20mA 이다. **Consumer Level**: 이 값은 동적으로 voltage나 current limit 레벨을 정하는 consumer 드라이버에 의해 정의된다. 예를 들어, consumer backlight 드라이버 LCD 밝기를 크게 하기 위하여 전류를 5mA 에서 10mA로 늘릴 것을 요청한다고 하자. 이 요청은 다음과 같은 흐름을 타고 level로 전달된다. *Consumer: need to increase LCD brightness. Lookup and request next current mA value in brightness table (the consumer driver could be used on several different personalities based upon the same reference device).* *Power Domain: is the new current limit within the domain operating limits for this domain and system state (e.g. battery power, USB power)* *Regulator Domains: is the new current limit within the regulator operating*

	parameters for input/output voltage. Regulator 요청 사항이 모든 constraint 테스트를 통과하게 되면, 새로운 regulator 값이 적용되게 된다.

반도체의 파워 소비 관련해서는 아래와 같은 공식을 써서 표현할 수 있다.

$$Power(Total) = P(static) + P(dynamic)$$

정적 파워(static power)는 누설 전류(leakage current)를 말하며, 이는 시스템이 active 상태일 때에는 동적 파워(dynamic power) 보다 작은 값으로, 시스템이 standby 상태일 때에 주 파워 소비 요인(main power drain)이라고 말할 수 있다. 반면, 동적 파워(dynamic power)는 active 전류를 말하는데, clock과 같은 signal switching이나, 아날로그 회로의 상태 변환(예: audio playback)이 여기에 해당한다. Regulator는 이러한 정적 및 동적 파워를 절약하기 위해 사용된다. Regulator는 글자 그대로, 입력 파워(전원)을 출력 파워로 변환하는 장치를 말하며, 구체적으로 아래의 세가지 동작이 가능하다.

1) 전압(Voltage) 제어(control) – "input is 5V output is 1.8V"
2) 전류(Current) 제한(limiting) – "limit output current to 20mA max"
3) 단순 스위치 역할(Simple switch) – "switch output power on/off"

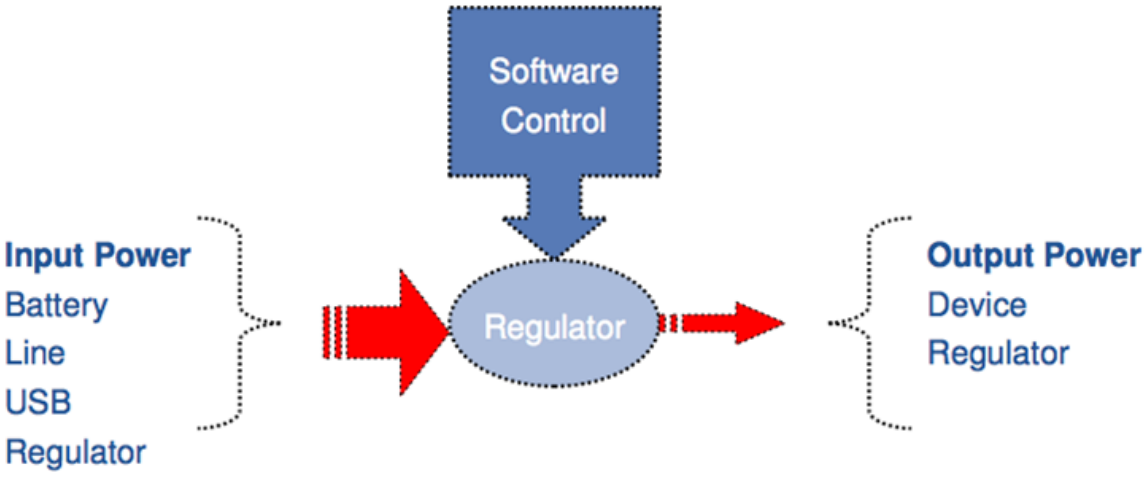

그림 5-25 Regulator 장치 소개 [출처 – 참고 문헌 14]

아래 그림은 파워 도메인(power domain)과 regulator의 관계를 표현한 것으로, Regulator-1은 Power Domain-1에 파워를 공급해 주고 있으며, Regulator-2는 Power Domain-2에, 마지막으로 Regulator-3는 Power Domain-3에 파워를 공급해 주고 있음을 알 수 있다.

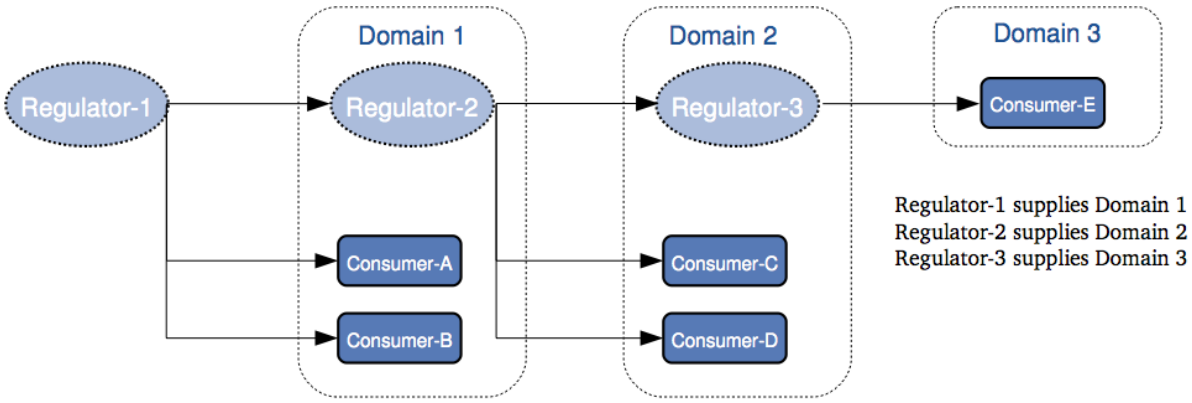

그림 5-26 파워 도메인(power domain)과 regulator와의 관계 [출처 – 참고 문헌 14]

그림 5-27은 CPU를 포함하여 여러 장치와 이들의 입출력 파워를 제어하는 regulator와의 관계를 하나의 그림으로 표현해 본 것이다. 그림에서 볼 수 있는 것처럼, 현대(?) 보드에서 regulator는 clock 만큼 중요한 역할을 담당하는 것을 짐작할 수 있다.

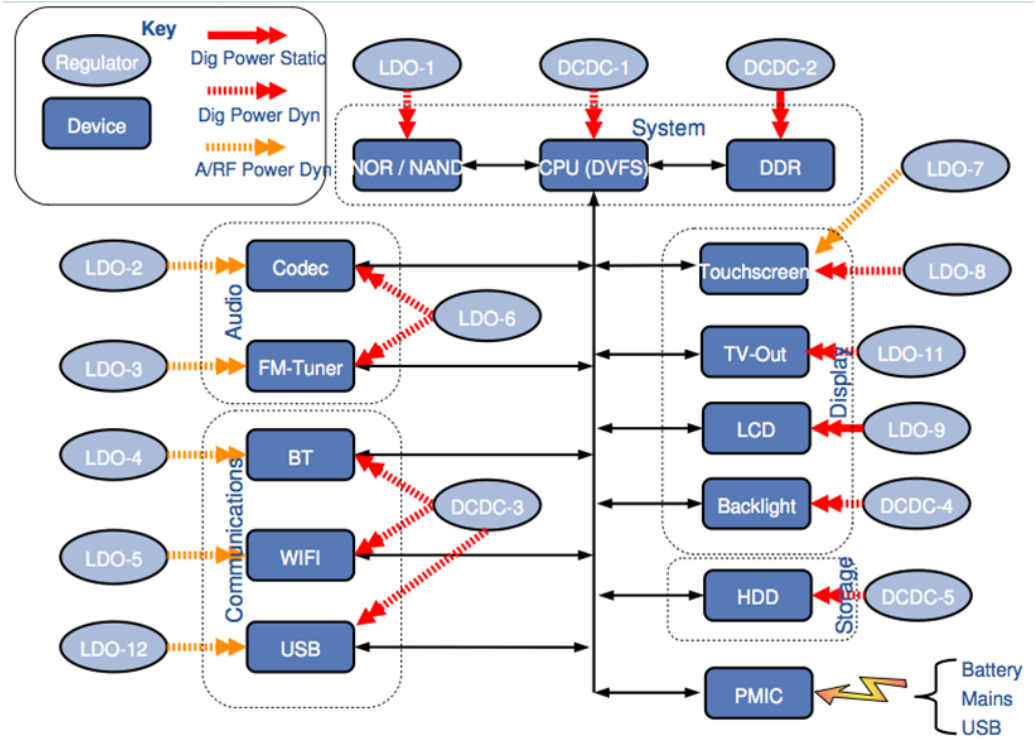

그림 5-27 각각의 장치와 레귤레이터 [출처 – 참고 문헌 14]

이제부터는 그림 5-27에서 본 것과 같이 각 장치에 붙어 있는 regulator를 제어하기 위하여, 커널에서 제공하는 regulator 프레임워크(Liam Girdwood에 의해 구현됨)를 소개해 보려고 한다.

Regulator 프레임워크는 전압(voltage)과 전류 regulator를 제어하는 표준 커널 인터페이스를

제공할 목적으로 설계되었다. 따라서 regulator 프레임워크를 이용하여 파워를 절약하고 배터리의 수명을 연장하기 위하여 regulator의 파워 출력을 동적으로 제어하는 것이 가능해졌다. Regulator 프레임워크는 아래 두 군데에 적용이 가능하다.

1) voltage regulators (전압 출력을 제어할 수 있는 곳 - where voltage output is controllable)
2) current sinks (전류 한계를 조절할 수 있는 곳 - where current limit is controllable)

Regulator 프레임워크는 구조적으로 아래의 4가지 인터페이스로 나뉘어져 있다.
1) 각각의 디바이스 드라이버를 위한 consumer 인터페이스(Consumer interface for device drivers)
2) Regulator 드라이버를 위한 regulator 드라이버 인터페이스(Regulator drivers interface for regulator drivers)
3) 보드 배치를 위한 머신 인터페이스(Machine interface for board configuration)
4) 사용자 영역에서의 사용을 위한 sysfs 인터페이스(sysfs interface for userspace)

Regulator Consumer Interface

Consumer는 파워를 제어하기 위해 regulator를 사용하는 디바이스 드라이버를 말하며, 아래의 두 가지 유형으로 분류할 수 있다.
1) Static (only need to enable/disable)
2) Dynamic (need to change voltage/ current limit)

각각의 디바이스 드라이버에서 사용하는 Consumer 인터페이스를 열거해 보면 아래와 같다.

Regulator 접근 함수
*regulator = **regulator_get**(dev, "Vcc");*
***regulator_put**(regulator);*

Regulator enable, disable 함수
*int **regulator_enable**(regulator);*
*int **regulator_disable**(regulator);*
*int **regulator_force_disable**(regulator);*

Regulator Status 확인 함수
*int **regulator_is_enabled**(regulator);*

전압 값을 설정하는 함수
*int **regulator_set_voltage**(struct regulator *regulator, int min_uV, int max_uV);*

전압 값을 얻어 오는 함수

*int **regulator_get_voltage**(struct regulator *regulator);*

전류 한계 값을 설정하는 함수
*int **regulator_set_current_limit**(struct regulator *regulator, int min_uA, int max_uA);*

전류 한계 값을 얻어 오는 함수
*int **regulator_get_current_limit**(struct regulator *regulator);*

최적의 효율을 설정하는 함수
***regulator_set_optimum_mode**(regulator, 10000); // 10mA*
[참고 사항] regulator는 100% 효율을 내지는 못하며, 효율은 부하에 영향을 받는다. Regulator는 효율을 높이기 위해 최적화(optimum) 모드를 변경할 수 있다.

Regulator 장치(hardware)는 아래와 같은 특수한 상황이 발생할 경우, 이를 알릴 수 있어야 하고, consumer(디바이스 드라이버)는 이를 받아 처리할 수 있어야 한다.
1) Regulator 실패 상황(Regulator failures)
2) 과열된 상태(Over temperature)

지금까지 설명한 consumer 인터페이스(API)를 정리해 보면 다음과 같다.

<Consumer registration>
regulator_get(), regulator_put()

<Regulator output power control and status>
regulator_enable(), regulator_disable(), regulator_force_disable(), regulator_is_enabled()

<Regulator output voltage control and status>
regulator_set_voltage(), regulator_get_voltage()

<Regulator output current limit control and status>
regulator_set_current_limit(), regulator_get_current_limit()

<Regulator operating mode control and status>
regulator_set_mode(), regulator_get_mode(), regulator_set_optimum_mode()

<Regulator events>
regulator_register_notifier(), regulator_unregister_notifier()

Regulator Driver Interface

Regulator 드라이버는 consumer 드라이버에서 사용할 수 있도록 미리 등록을 해주어야 한다.

*struct regulator_dev **regulator_register**(struct regulator_desc *regulator_desc, void *reg_data);*
*void **regulator_unregister**(struct regulator_dev *rdev);*

Consumer 드라이버로 event를 전달하는 함수는 아래의 notifier 체인을 통해 사전에 등록이 되어야 한다.

*int **regulator_notifier_call_chain**(struct regulator_dev *rdev, unsigned long event, void *data);*

Regulator 드라이버는 아래의 struct regulator_ops를 구현해야 한다.

```
struct regulator_ops {
	/* get/set regulator voltage */
	int (*set_voltage) (struct regulator_dev *, int min_uV, int max_uV);
	int (*get_voltage) (struct regulator_dev *);
	/* get/set regulator current */
	int (*set_current_limit) (struct regulator_dev *,
	int min_uA, int max_uA);
	int (*get_current_limit) (struct regulator_dev *);
	/* enable/disable regulator */
	int (*enable) (struct regulator_dev *);
	int (*disable) (struct regulator_dev *);
	int (*is_enabled) (struct regulator_dev *);
	/* get/set regulator operating mode (defined in regulator.h) */
	int (*set_mode) (struct regulator_dev *, unsigned int mode);
	unsigned int (*get_mode) (struct regulator_dev *);
	/* get most efficient regulator operating mode for load */
	unsigned int (*get_optimum_mode) (struct regulator_dev *, int input_uV,
	int output_uV, int load_uA);
};
```

Machine Driver Interface

Regulator 머신 드라이버 인터페이스는 글자 그대로 보드 초기화 과정에서 Regulator를 어떠한 식으로 기술할 것인지를 정의해 주는 부분으로, 아래의 예를 통해 어떠한 형태로 코드를 작성해야 하는지를 살펴보도록 하자.

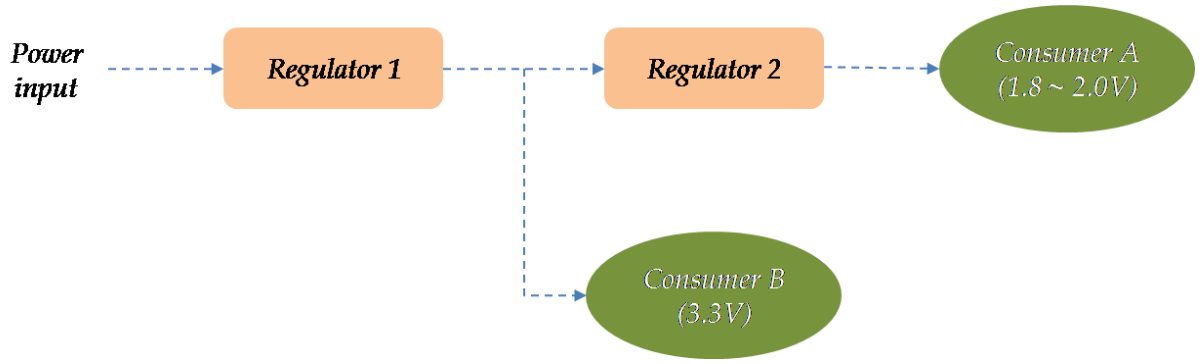

그림 5-28 Regulator 장치를 플랫폼 디바이스 등록하는 절차를 소개하기 위한 Regulator 예

먼저 consumer 드라이버 A와 B는 정확한 regulator에 매핑을 시켜야 한다. 이는 각각의 regulator에 대해 struct regulator_consumer_supply를 만들어 주는 과정을 통해 가능하다.

코드 5-17 regulator_consumer_supply structure

```
struct regulator_consumer_supply {                    [in include/regulator/machine.h file]
      const char *dev_name;     /* consumer dev_name() */
      const char *supply; /* consumer supply - e.g. "vcc" */
};

<Regulator1의 경우>
static struct regulator_consumer_supply regulator1_consumers[] = {
{
      .dev_name     = "dev_name(consumer B)",
      .supply       = "Vcc",
};
[설명] Regulator1을 Consumer B의 Vcc 파워 supply로 매핑시킨다는 뜻임.

<Regulator2의 경우>
static struct regulator_consumer_supply regulator2_consumers[] = {
{
      .dev      = "dev_name(consumer A"),
      .supply = "Vcc",
};
[설명] Regulator2을 Consumer A의 Vcc 파워 supply로 매핑시킨다는 뜻임.
```

다음으로 각각의 power domain에 대해 struct regulator_init_data를 정의해 줌으로써 constraints 값을 등록해 주어야 한다.

코드 5-18 regulator_init_data structure

```
struct regulator_init_data {                                    [in include/regulator/machine.h file]
      const char *supply_regulator;          /* or NULL for system supply */

      struct regulation_constraints constraints;

      int num_consumer_supplies;
      struct regulator_consumer_supply *consumer_supplies;

      /* optional regulator machine specific init */
      int (*regulator_init)(void *driver_data);
      void *driver_data;   /* core does not touch this */
};

static struct regulator_init_data regulator1_data = {
      .constraints = {
            .name = "Regulator-1",
            .min_uV = 3300000,
            .max_uV = 3300000,
            .valid_modes_mask = REGULATOR_MODE_NORMAL,
      },
      .num_consumer_supplies = ARRAY_SIZE(regulator1_consumers),
      .consumer_supplies = regulator1_consumers,
};
```

[설명] name field를 지정하지 않을 경우, 시스템이 알아서 적당한 이름을 만들어주게 된다. 하지만, 아래의 예에서 처럼 다른 부분에서 참조가 필요한 경우가 있으므로 필요에 따라 적당한 이름으로 지정해 줄 필요가 있을 것이다.

```
static struct regulator_init_data regulator2_data = {
      .supply_regulator = "Regulator-1",
      .constraints = {
            .min_uV = 1800000,
            .max_uV = 2000000,
            .valid_ops_mask = REGULATOR_CHANGE_VOLTAGE,
            .valid_modes_mask = REGULATOR_MODE_NORMAL,
      },
      .num_consumer_supplies = ARRAY_SIZE(regulator2_consumers),
      .consumer_supplies = regulator2_consumers,
};
```

[설명] 현재 Regulator-1은 Regulator-2에 파워를 공급하고 있다. 따라서 Consumer A가 자신의 power supply인 Regulator-2를 enable 시킬 때, Regulator-1도 함께 enable이 되어야 한다. 이는 .supply_regulator 필드에 "Regulator-1"을 설정하므로써 가능해진다.

마지막으로 regulator 장치를 platform device 형태로 등록해 주어야 한다(참고로, 6장에서 설명하는 Device Tree를 사용할 경우에는 platform device 형태로 등록할 필요가 없다).

코드 5-19 regulator device 등록

```
static struct platform_device regulator_devices[] = {
    {
        .name = "regulator",
        .id = DCDC_1,
        .dev = {
            .platform_data = &regulator1_data,
        },
    },
    {
        .name = "regulator",
        .id = DCDC_2,
        .dev = {
            .platform_data = &regulator2_data,
        },
    },
};
```

[설명] .platform_data 필드에 들어가는 내용은 앞서 설명한 constraints 관련 부분(struct regulator_init_data)이다.

```
/* regulator 1 device를 등록 */
platform_device_register(&regulator_devices[0]);

/* regulator 2 device를 등록 */
platform_device_register(&regulator_devices[1]);
```

6. 배터리 충전 드라이버

이번 장에서는 대부분 전용 PMIC(Power Management Integrated Circuit) 칩 내에 내장되어 있는 충전 장치(charger device)와 이에 대한 디바이스 드라이버(charger driver)를 소개하고자 한다.

충전 장치

PMIC는 시스템 전체(주로 CPU 외부)의 파워를 조절하는 역할을 하는 장치이지만, 대개의 경우 배터리에 대한 충전 작업도 병행하는 경향이 있다.

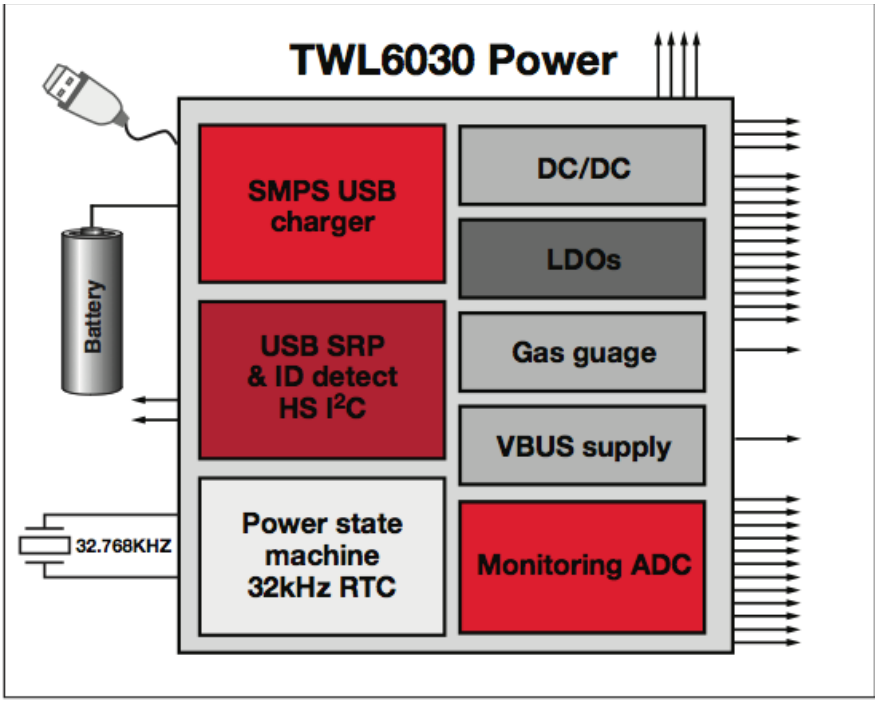

그림 5-29 OMAP4 TWL6030 PMIC 칩 [출처 – 참고문헌 17]

여기서 잠깐 ! PMIC란 ?

PMIC (Power Management IC)는 말 그대로 파워를 조정해주는 IC를 의미

주로 모바일이나 배터리로 동작해야 하는 장비들에서 배터리 구동시간을 늘리기 위해 많이 사용함. 기능은 CPU 등에서 처리해야 하는 Load에 맞추어 그에 따른 인터페이스 신호를 PMIC 에 주면 PMIC가 CPU에 공급하는 Core 전압을 그에 맞게 조정을 해서 항시 최소한의 전력으로 구동할수 있도록 함.

PMIC 회로기술의 주 목적은 한정적인 배터리 전원을 다양한 부하 변동에 능동적으로 대처하여 배터리 전원을 효율적으로 관리하여 배터리 수명을 연장하는 것이고, 최근에는 주변의 온도, 열, 진동, 압전 등 환경에서 발생하는 에너지를 수확 및 저장하여 사용하는 에너지 하베스팅 및 태양광, 풍력 등의 신재생용 에너지 전력반도체 회로 기술에 대한 연구가 활발히 진행되고 있다.

LDO는 (Low Dropout) Regulator를 뜻함.

입력전압과 출력전압 차이가 크지 않은데 사용하면서 근래에는 초기 출력이 강하여 아주 빠른 시간 내에 정상 출력 값이 되는 제품들도 많이 나오고 있음. 제품은 여러 벤더에서 나오는데… Micrel , Sipex 등등… 국내에서 구하기 쉬운 제품들도 많음. 데이터 시트의 스펙을 보시면 일반 Regulator 와 차이점을 아실 수 있을 것으로 보임 …

일반적인 충전 절차를 소개하면 다음과 같다.

<충전 절차>

1) Trickle charging

- 상당히 많이 떨어져 있는 battery 전압을 끌어 올려, 다음 단계인 fast charging을 하기에 충분한 상태로 만드는 단계
- 높은 전류를 공급하는 fast charging 과정을 방전된 battery에 대해 시도하게 되면, 과전류가 흘러 핸드폰이 손상될 수 있어, trickle charging 단계를 두어 낮은 전류를 공급하여 충전을 시도하게 됨.

2) Constant Current charging(= fast charging)

- 일정 수준의 전류를 유지하면서 충전하는 단계
- 3.2V ~ 4.1V 사이에서 구동
- CC(constant current) -> CV(constant voltage)로 전이 조건은 voltage detector가 담당. Signal을 state machine에 전달
- 최대 512min간 지속

3) Constant Voltage charging

- 충전의 마지막 단계로, 일정 수준의 전압을 유지하면서 충전
- 4.1V ~
- 최대 90분간 지속

*) 기타 참고 사항

- Vbat 값이 원하는 정도에 도달하면, 저항을 올려서 battery로 들어가는 current가 줄어들게 됨.
- 상태를 관장하는 중심에는 SM(State Machine)이 있음.

그림 5-30에 위에서 설명한 충전 과정을 대략적으로 그림으로 표현해 보면 다음과 같다.

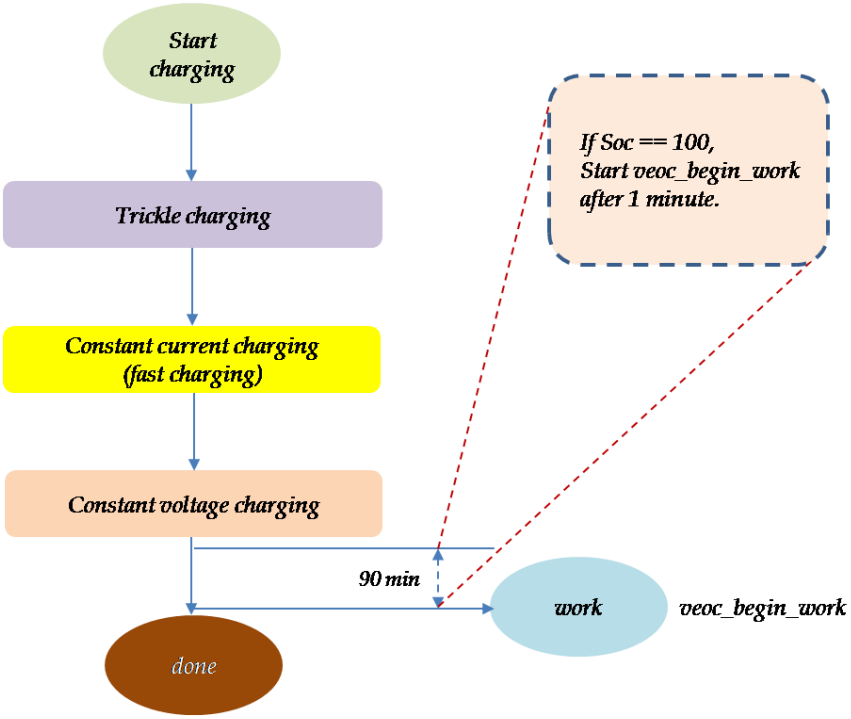

그림 5-30 충전 드라이버(Charger)의 충전 절차

충전 드라이버와 Power Supply 프레임워크

충전 드라이버(Charger driver)는 충전 과정 중 발생하는 인터럽트(interrupt)에 따라 충전 상태를 파악한 후, 안드로이드 프레임워크로 배터리(battery) 상태 정보를 전달하는 역할을 수행한다. 일반적으로 배터리 정보는 연료 게이지(Fuel Gauge) 드라이버를 통해 이루어진다.

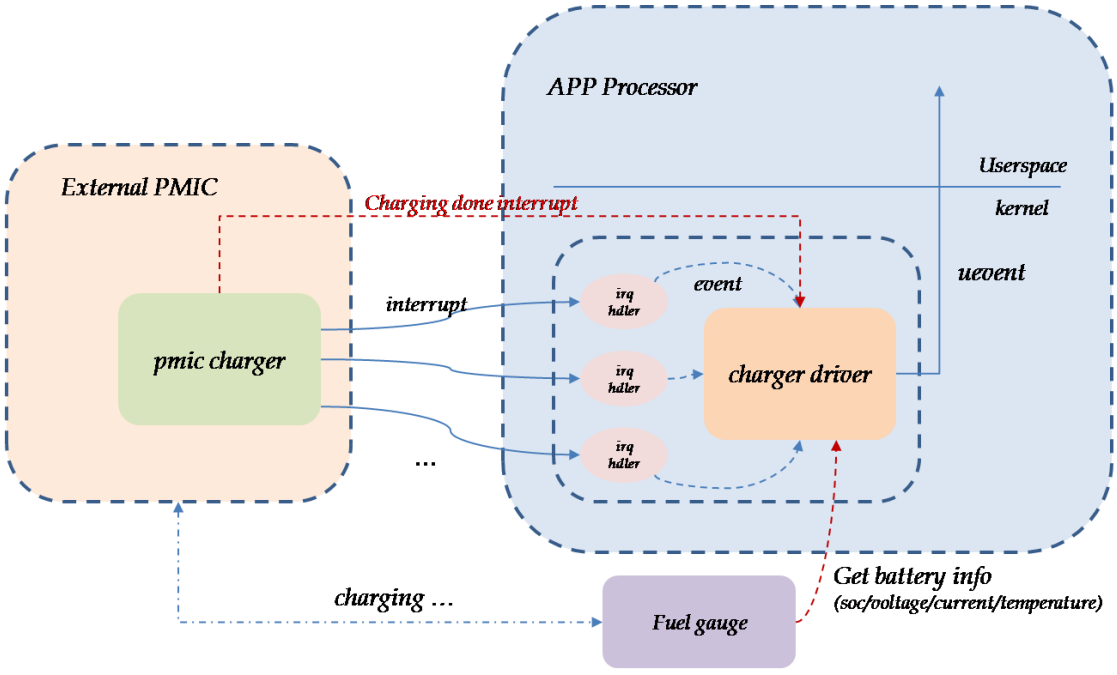

그림 5-31 충전 및 Fuel Gauge 드라이버의 상관 관계

Fuel Gauge는 SOC(State of Charge), 전압 값, 전류 값, 온도 값 등의 정보를 알려 주는 장치로, 아래와 같이 충전 드라이버에서 주기적으로 Fuel Gauge에 그 값을 요청하는 형태로 이해하면 될 것이다.

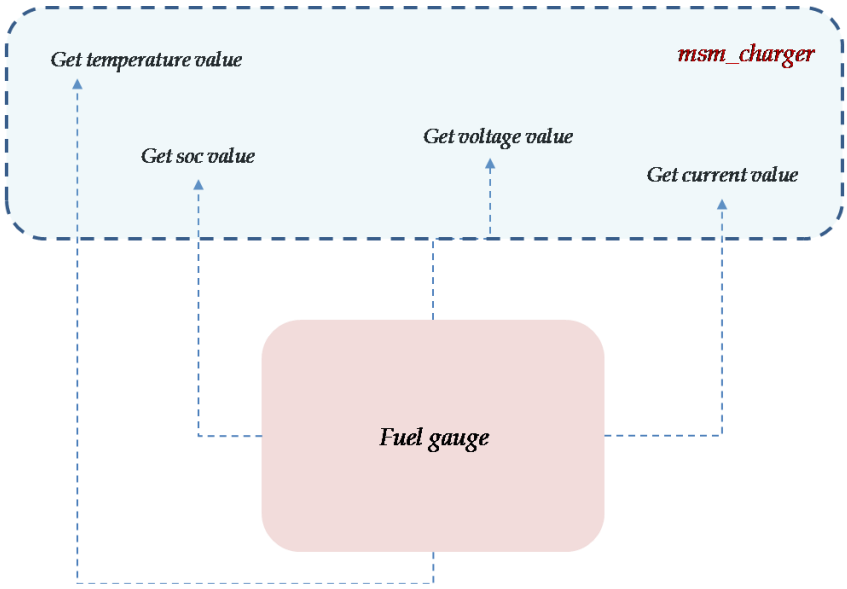

그림 5-32 Fuel Gauge 드라이버 개요 [다시 그려야 함]

충전 드라이버에서 주기적으로 배터리 상태를 검사하여 변화가 발생한 경우, power_supply_changed() 함수를 사용하여, 이를 사용자 영역으로 전달하게 된다. 이때 사용되는 power_supply_changed() 함수는 내부적으로 kobject_uevet() 함수를 사용하여 구현되어 있으며, 최종적으로는 netlink socket을 사용하여 사용자 영역으로 배터리 정보를 전달하도록 구현되어 있음을 알 수 있다. 그림 5-33에 이들의 관계를 표현해 보았다.

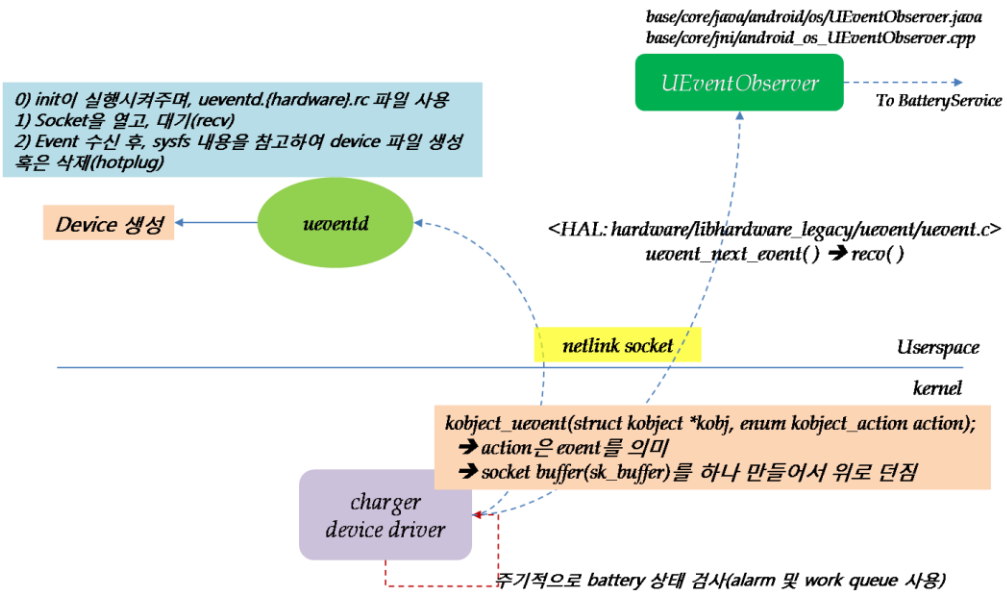

그림 5-33 Power Supply 프레임워크

Charger 드라이버 코드 소개

아래 예제는 OMAP3에서 사용되는 TWL4030/TPS65950 BCI (Battery Charger Interface) 드라이버를 정리한 것이다.

코드 5-20 drivers/power/twl4030_charger.c

```
1) TWL4030/TPS65950 BCI (Battery Charger Interface) driver
2) Platform device는 arch/arm/mach-omap2/board*.c 파일이 아니라, drivers/mfd/twl-core.c에서 등록해 주고 있음.

struct twl4030_bci {                                    [in drivers/power/twl4030_charger.c file]
        struct device           *dev;
        struct power_supply ac;
        struct power_supply usb;
        struct usb_phy          *transceiver;
        struct notifier_block     usb_nb;
        struct work_struct    work;
        int             irq_chg;
        int             irq_bci;
        struct regulator       *usb_reg;
        int             usb_enabled;

        unsigned long           event;
};
[...]
static irqreturn_t twl4030_charger_interrupt(int irq, void *arg)
{
        struct twl4030_bci *bci = arg;

        dev_dbg(bci->dev, "CHG_PRES irq₩n");
        power_supply_changed(&bci->ac);     //배터리 상태 변화를 사용자 영역으로 전달
        power_supply_changed(&bci->usb);

        return IRQ_HANDLED;
}
[...]
static irqreturn_t twl4030_bci_interrupt(int irq, void *arg)
{
```

```
struct twl4030_bci *bci = arg;
u8 irqs1, irqs2;
int ret;

ret = twl_i2c_read_u8(TWL4030_MODULE_INTERRUPTS, &irqs1,
                TWL4030_INTERRUPTS_BCIISR1A);
if (ret < 0)
    return IRQ_HANDLED;

ret = twl_i2c_read_u8(TWL4030_MODULE_INTERRUPTS, &irqs2,
                TWL4030_INTERRUPTS_BCIISR2A);
if (ret < 0)
    return IRQ_HANDLED;

dev_dbg(bci->dev, "BCI irq %02x %02x₩n", irqs2, irqs1);

if (irqs1 & (TWL4030_ICHGLOW | TWL4030_ICHGEOC)) {
    /* charger state change, inform the core */
    power_supply_changed(&bci->ac); //배터리 상태 변화를 사용자 영역으로 전달
    power_supply_changed(&bci->usb);
}

/* various monitoring events, for now we just log them here */
if (irqs1 & (TWL4030_TBATOR2 | TWL4030_TBATOR1))
    dev_warn(bci->dev, "battery temperature out of range₩n");

if (irqs1 & TWL4030_BATSTS)
    dev_crit(bci->dev, "battery disconnected₩n");

if (irqs2 & TWL4030_VBATOV)
    dev_crit(bci->dev, "VBAT overvoltage₩n");

if (irqs2 & TWL4030_VBUSOV)
    dev_crit(bci->dev, "VBUS overvoltage₩n");

if (irqs2 & TWL4030_ACCHGOV)
    dev_crit(bci->dev, "Ac charger overvoltage₩n");

return IRQ_HANDLED;
```

```
}
[...]
static int __init twl4030_bci_probe(struct platform_device *pdev)
{
	struct twl4030_bci *bci;
	struct twl4030_bci_platform_data *pdata = pdev->dev.platform_data;
	int ret;
	u32 reg;

	bci = kzalloc(sizeof(*bci), GFP_KERNEL);
	if (bci == NULL)
		return -ENOMEM;

	bci->dev = &pdev->dev;
	bci->irq_chg = platform_get_irq(pdev, 0);
	bci->irq_bci = platform_get_irq(pdev, 1);

	platform_set_drvdata(pdev, bci);
	bci->ac.name = "twl4030_ac";
	bci->ac.type = POWER_SUPPLY_TYPE_MAINS;
	bci->ac.properties = twl4030_charger_props;
	bci->ac.num_properties = ARRAY_SIZE(twl4030_charger_props);
	bci->ac.get_property = twl4030_bci_get_property;

	ret = power_supply_register(&pdev->dev, &bci->ac);
	if (ret) {
		dev_err(&pdev->dev, "failed to register ac: %d₩n", ret);
		goto fail_register_ac;
	}
	bci->usb.name = "twl4030_usb";
	bci->usb.type = POWER_SUPPLY_TYPE_USB;
	bci->usb.properties = twl4030_charger_props;
	bci->usb.num_properties = ARRAY_SIZE(twl4030_charger_props);
	bci->usb.get_property = twl4030_bci_get_property;

	bci->usb_reg = regulator_get(bci->dev, "bci3v1");

	ret = power_supply_register(&pdev->dev, &bci->usb);
	if (ret) {
```

```
            dev_err(&pdev->dev, "failed to register usb: %d₩n", ret);
            goto fail_register_usb;
        }

        ret = request_threaded_irq(bci->irq_chg, NULL,
                twl4030_charger_interrupt, IRQF_ONESHOT, pdev->name,
                bci);
        if (ret < 0) {
            dev_err(&pdev->dev, "could not request irq %d, status %d₩n",
                bci->irq_chg, ret);
            goto fail_chg_irq;
        }
        ret = request_threaded_irq(bci->irq_bci, NULL,
                twl4030_bci_interrupt, IRQF_ONESHOT, pdev->name, bci);
        if (ret < 0) {
            dev_err(&pdev->dev, "could not request irq %d, status %d₩n",
                bci->irq_bci, ret);
            goto fail_bci_irq;
        }

        INIT_WORK(&bci->work, twl4030_bci_usb_work);
        [...]
        twl4030_charger_enable_ac(true);
        twl4030_charger_enable_usb(bci, true);
        twl4030_charger_enable_backup(pdata->bb_uvolt, pdata->bb_uamp);
        [...]
}
[...]
static struct platform_driver twl4030_bci_driver = {
        .driver = {
            .name     = "twl4030_bci",
            .owner    = THIS_MODULE,
        },
        .remove = __exit_p(twl4030_bci_remove),
};

module_platform_driver_probe(twl4030_bci_driver, twl4030_bci_probe);
```

7. 그 밖의 주제

지금까지 설명한 내용 이외에도 파원 관리 관련하여 그 밖의 중요한 주제를 정리해 보면 다음과 같다.

1) CPU Hotplug
2) Thermal Framework
3) PM QoS
4) Clock Domain, Power Domain
5) PMIC
...

References

[1] Android PM Guide10.pdf, Chunghan Yi, www.kandroid.org

[2] Power Management, Michael Opdenacker, Thomas Petazzoni, Free Electrons

[3] Power saving in Linux devices, Chris Simmonds, 2net Limited, Class 5.5 Embedded Systems Conference UK. 2009

[4] Powerdebugging inside Linaro, Amit Kucheria amit.kucheria@linaro.org, Tech Lead Power Management Working Group

[5] %233.GTUG-Android-Power Management.pdf, Renaldo Noma, 2010

[6] Suspend-to-RAM in Linux, A. Leonard Brown, Intel Open Source Technology Center, len.brown@intel.com, Rafael J. Wysocki, Institute of Theoretical Physics, University of Warsaw, rjw@sisk.pl

[7] Mainline Kernel Support for Opportunistic Suspend, Rafael J. Wysocki, Renesas Electronics / SUSE Labs August 29, 2012

[8] Technical Background of the Android Suspend Blockers Controversy, Rafael J. Wysocki, rjw@sisk.pl, Faculty of Physics, University of Warsaw / SUSE Labs, Novell Inc., November 12, 2010

[9] Runtime Power Management Framework for I/O Devices in the Linux Kernel, Rafael J. Wysocki, Faculty of Physics UW / SUSE Labs / Renesas, June 10, 2011

[10] Runtime Power Management, Kevin Hilman, Deep Root Systems, LLC, khilman@deeprootsystems.com

[11] Power Management Framework in Linux, Pavitrakumar K.M, Vayavya Labs Pvt. Ltd., Bangalore, India, Pavitra@vayavyalabs.com

[12] Linux clock management framework, Siarhei Yermalayeu, GertVervoort, ShankarMahadevan, BoudewijnBecking, Embedded Linux Conference, November 2 & 3, 2007

[13] A Generic Clock Framework in the Kernel: Why We Need It & Why We Still Don't Have It, ELC Europe 2011, Sascha Hauer s.hauer@pengutronix.de

[14] A Dynamic Voltage and Current Regulator Control Interface for the Linux Kernel, Liam Girdwood

[15] Kernel documentations under linux/Documentation directory.

[16] 안드로이드 하드웨어 서비스, 김대우/박재영/문병원, 개발자가 행복한 세상

[17] swpt034b.pdf - OMAPTM 4 mobile applications platform brochure, Texas Instruments