

# *Android **Device Driver Hacks***

*: platform driver, work queue, tasklet, wakelock ...*



ANDROID

*chunghan.yi@gmail.com, slowboot*

Revision	작성자	비고
0.1	이 충 한	최초 작성 11/11/2011
0.2	이 충 한	11/15/2011
0.3	이 충 한	11/16/2011
0.4	이 충 한	11/17/2011

## 목차

- 1. Platform Device & Driver
- 2. Task & Scheduling
- 3. Bottom Halves and Deferring Work
  - ➔ *Tasklet, Softirq, Work Queue, Kernel thread*
- 4. Timer
- 5. Wakelock
- 6. Synchronization
- 7. Notifier
- 8. Example Driver
  - ➔ *bluetooth sleep mode driver(TODO)*
- 9. Debug(TODO)
- **Appendix**
- **TODO**
- **References**

## 0. 이 문서에서 다루고자 하는 내용

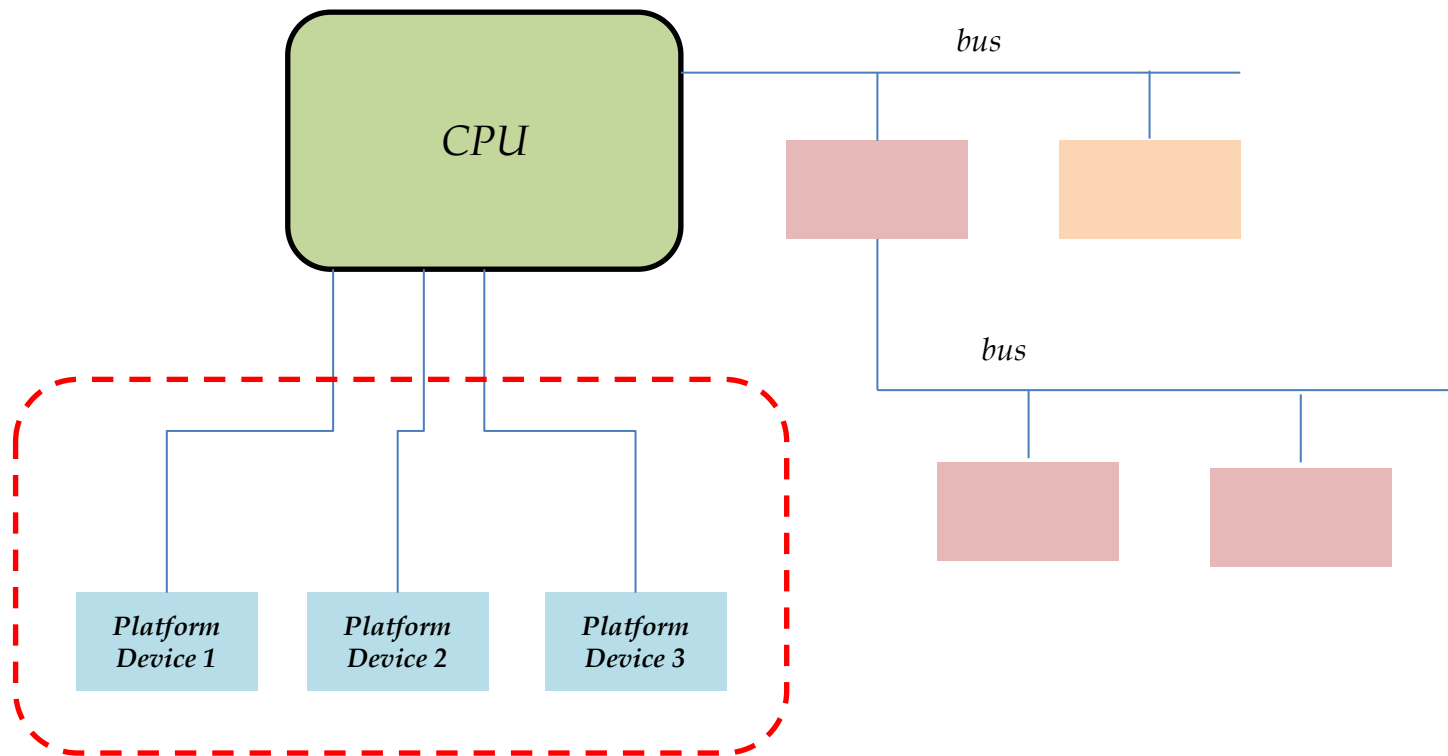
- 1) Platform driver
- 2) Interrupt handler, tasklet, work queue, kernel thread, timer
- 3) Wakelock, synchronization & wait queue issue
- 4) Kernel panic/Oops debug
- ➔ 평이하고 일반적인 내용은 빼고(이건 *device driver book*을 참조하시길...),
- ➔ *Qualcomm Device Driver*에서 자주 사용되는 방식 위주로 기술
- ➔ 단순 코드 나열 보다는 그림을 통해 분석하고 함 !!!

(\*) 본 문서는 *Gingerbread 2.3.4(linux kernel 2.6.35)*를 기준으로 작성하였으나, 일부 내용은 *kernel version*과 정확히 일치하지 않을 수 있음.

(\*) 본 문서에 기술되어 있는 *API* 혹은 *data structure* 내용 중에는 오타자가 포함되어 있을 수 있으며, 경우에 따라서는 의도적으로 생략한 부분도 있으니, 정확한 정보를 위해서는 관련 헤더 파일을 참조해 주시기 바람.

# 1. Platform Device & Driver(1) - 개념

- 1) Embedded system의 시스템의 경우, bus를 통해 device를 연결하지 않는 경우가 있음.  
→ bus는 확장성(enumeration), hot-plugging, unique identifier를 허용함에도 불구하고 ...
- 2) platform driver/platform device infrastructure를 사용하여 이를 해결할 수 있음.
- → platform device란, 별도의 bus를 거치지 않고, CPU에 직접 연결되는 장치를 일컫음.



# 1. Platform Device & Driver(1) - 개념

- *platform\_device* 정의 및 초기화
- *resource* 정의

(arch/arm/mach-msm/board-XXXX.c 파일에 위치함)

<예 - bluetooth sleep device>

```
struct platform_device my_bluesleep_device = {  
    .name = "bluesleep",  
    .id    = 0,  
    .num_resources = ARRAY_SIZE(bluesleep_resources),  
    .resource = bluesleep_resources,  
};
```

- *platform\_driver* 정의 및 초기화
- *probe/remove*

(drivers/XXXX/xxxx.c 등에 위치함)

*.name* 필드("bluesleep")로 상호 연결

```
struct platform_driver bluesleep_driver = {  
    .remove = bluesleep_remove,  
    .driver = {  
        .name = "bluesleep",  
        .owner = THIS_MODULE,  
    },  
};
```

## 1. Platform Device & Driver(2) – *platform driver*

- (\*) *drivers/serial/imx.c* file에 있는 *iMX serial port driver*를 예로써 소개하고자 함. 이 드라이버는 *platform\_driver structure*를 초기화함.

```
static struct platform_driver serial_imx_driver = {  
    .probe      = serial_imx_probe,  
    .remove     = serial_imx_remove,  
    .driver      = {  
        .name     = "imx-uart",  
        .owner    = THIS_MODULE,  
    },  
};
```

- (\*) *init/cleanup*ㄱ/, *register/unregister* 하기

```
static int __init imx_serial_init(void)  
{  
    platform_driver_register(&serial_imx_driver);  
}  
static void __exit imx_serial_cleanup(void)  
{  
    platform_driver_unregister(&serial_imx_driver);  
}
```

## 1. Platform Device & Driver(3) – *platform\_device*

- (\*) 플랫폼 디바이스는 동적으로 감지(detection)가 될 수 없으므로, static하게 지정해 주어야 함. static하게 지정하는 방식은 chip 마다 다를 수 있는데, ARM의 경우는 board specific code (arch/arm/mach-imx/mx1ads.c)에서 객체화 및 초기화(instantiation)를 진행하게 됨.
- (\*) Platform 디바이스와 Platform 드라이버를 matching시키기 위해서는 name(아래의 경우는 "imx-uart")을 이용함.

```
static struct platform_device imx_uart1_device = {
    .name          = "imx-uart",
    .id            = 0,
    .num_resources  = ARRAY_SIZE(imx_uart1_resources),
    .resource       = imx_uart1_resources,
    .dev = {
        .platform_data = &uart_pdata,
    }
};
```



## 1. Platform Device & Driver(4) - *platform\_device*(초기화)

- (\*) *platform device*는 아래 *list*에 추가되어야 하며,

```
static struct platform_device *devices[] __initdata = {
    &cs89x0_device,
    &imx_uart1_device,
    &imx_uart2_device,
};
```

- (\*) *platform\_add\_devices()* 함수를 통해서 실제로 시스템에 추가됨.

```
static void __init mx1ads_init(void)
{
    [...]
    platform_add_devices(devices, ARRAY_SIZE(devices));
    [...]
}

MACHINE_START(MX1ADS, "Freescale MX1ADS")
    [...]
    .init_machine    = mx1ads_init,
MACHINE_END
```

## 1. Platform Device & Driver(5) - *platform\_device(resource)*

- (\*) 특정 드라이버가 관리하는 각 장치(device)는 서로 다른 H/W 리소스를 사용하게 됨.

→ I/O 레지스터 주소, DMA 채널, IRQ line 등이 서로 상이함.

(\*) 이러한 정보는 *struct resource data structure*를 사용하여 표현되며, 이들 *resource* 배열은 *platform device* 정의 부분과 결합되어 있음.

- (\*) *platform driver*내에서 *platform\_device* 정보(pointer)를 이용하여 *resource*를 얻어 오기 위해서는 *platform\_get\_resource\_byname(...)* 함수가 사용될 수 있음.

```
static struct resource imx_uart1_resources[] = {
    [0] = {
        .start    = 0x00206000,
        .end      = 0x002060FF,
        .flags    = IORESOURCE_MEM,
    },
    [1] = {
        .start    = (UART1_MINT_RX),
        .end      = (UART1_MINT_RX),
        .flags    = IORESOURCE_IRQ,
    },
};
```

## 1. Platform Device & Driver(6) - *platform\_device(device specific data)*

- (\*) 앞서 설명한 *resource data structure* 외에도, 드라이버에 따라서는 자신만의 환경 혹은 데이터(*configuration*)을 원할 수 있음. 이는 *struct platform\_device* 내의 *platform\_data*를 사용하여 지정 가능함.  
(\*) *platform\_data*는 *void \* pointer*로 되어 있으므로, 드라이버에 임의의 형식의 데이터 전달이 가능함.  
(\*) *iMX* 드라이버의 경우는 *struct imxuart platform data*가 *platform\_data*로 사용되고 있음.

```
static struct imxuart_platform_data uart_pdata = {  
    .flags = IMXUART_HAVE_RTSCS,  
};
```

## 1. Platform Device & Driver(7) – *platform driver(probe, remove)*

- (\*) 보통의 *probe* 함수 처럼, 인자로 *platform\_device*에의 *pointer*를 넘겨 받으며, 관련 *resource*를 찾기 위해 다른 *utility* 함수를 사용하고, 상위 *layer*로 해당 디바이스를 등록함. 한편 별도의 그림으로 표현하지는 않았으나, *probe*의 반대 개념으로 드라이버 제거 시에는 *remove* 함수가 사용됨.

```
static int serial_imx_probe(struct platform_device *pdev)
{
    struct imx_port *sport;
    struct imxuart_platform_data *pdata;
    void __iomem *base;
    struct resource *res;

    sport = kzalloc(sizeof(*sport), GFP_KERNEL);
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    base = ioremap(res->start, PAGE_SIZE);

    sport->port.dev = &pdev->dev;
    sport->port.mapbase = res->start;
    sport->port.membase = base;
    sport->port.type = PORT_IMX,
    sport->port.iotype = UPIO_MEM;
    sport->port.irq = platform_get_irq(pdev, 0);
    sport->rxirq = platform_get_irq(pdev, 0);
    sport->txirq = platform_get_irq(pdev, 1);
    sport->rtsirq = platform_get_irq(pdev, 2);

    [...]
```

## 다음 장으로 넘어가기 전에 ...

### <work 정의>

- 1) task
- 2) some function routine  
: interrupt handler,  
softirq, tasklet,  
work queue, timer function

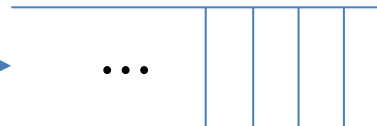
### <실행 요청>

- 1) schedule
- 2) Interrupt
- 3) it's own schedule func

(\*) 앞으로 설명할 task/scheduling, top/bottom halves, timer routine 등은 모두 아래와 같은 형태로 일반화시켜 생각해 볼 수 있을 듯하다.

→ 너무 일반화 시켰나 ^^

(\*) 한가지 재밌는 것은 이러한 구조는 kernel 내에서 뿐만 아니라, Android UI 내부 Message 전달 구조 및 media framework의 핵심인 stagefright event 전달 구조에서도 유사점을 찾을 수 있다는 것이다^^.



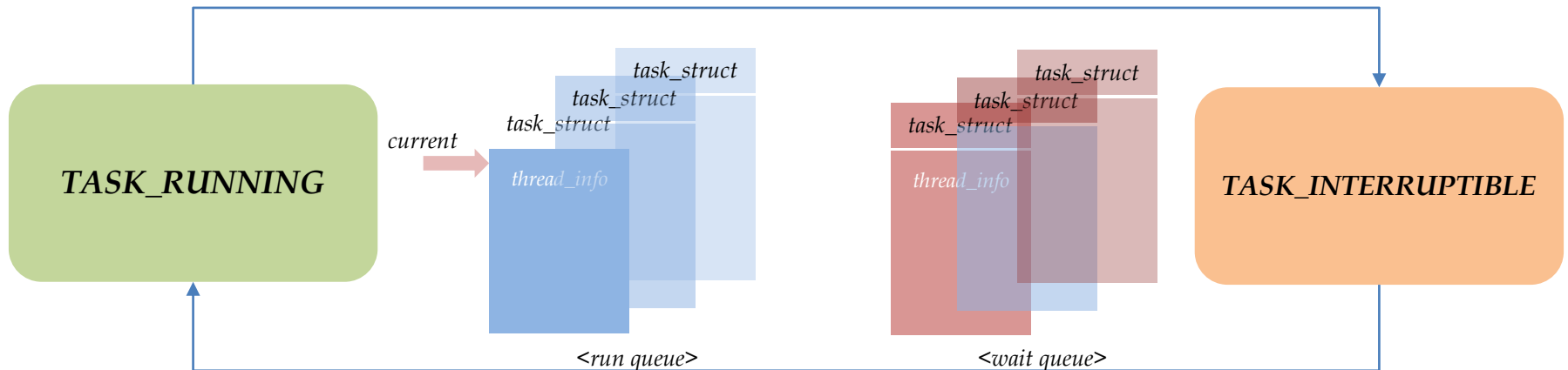
- 1) runqueue, waitqueue
- 2) work queue
- 3) **no queue** - interrupt, tasklet,  
timer function ...

### <처리 루틴>

- 1) scheduler
- 2) interrupt handling,  
tasklet processing,  
timer processing,
- 3) worker thread

## 2. Task & Scheduling(1)

- 1) `add_wait_queue`는 `task`를 `wait queue`에 추가하고, `task`의 상태를 `TASK_INTERRUPTIBLE`로 변경시킴.
- 2) 이어 호출되는 `schedule()` 함수는 `task`를 `runqueue`에서 제거해줌.

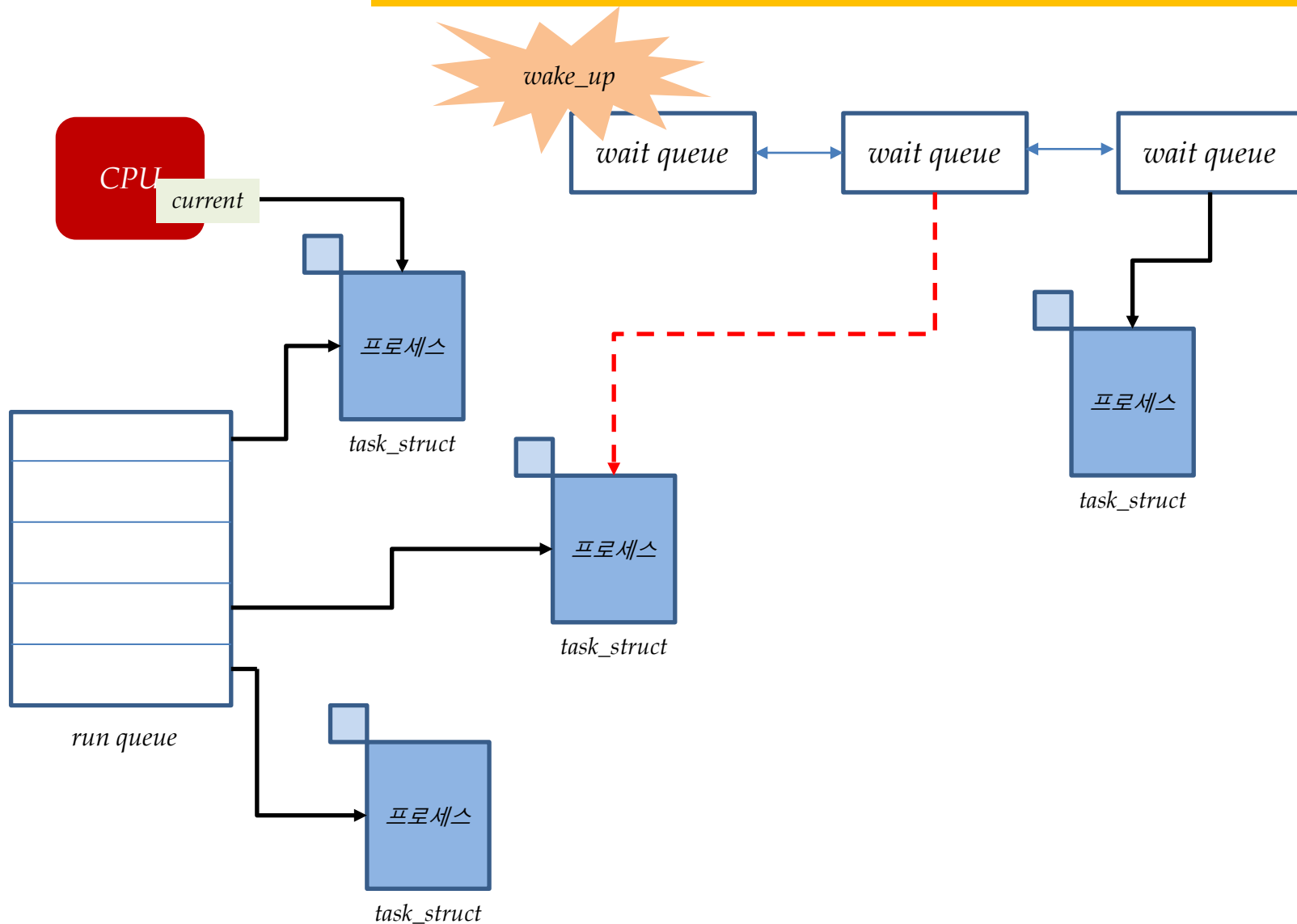


- 1) `Task`가 기다리던 `event`가 발생하면, `try_to_wake_up()` 함수는 `task`의 상태를 `TASK_RUNNING`으로 변경시키고, `activate_taks()` 함수를 호출하여 `task`를 `runqueue`에 추가시킴.
- 2) `_remove_wait_queue`는 `task`를 `wait queue`에서 제거함.

(\*) `task` 관련 `queue`로는 `wait queue`와 `run queue`가 있으며, `run queue`에 등록된 `task`는 실행되고, `wait queue`에 등록된 `task`는 특정 조건이 성립될 때까지 기다리게 된다.  
(\*) 위에서 언급된 특정 함수는 버전에 따라 차이가 있을 수 있음. 단, 전체적인 개념은 동일함.

## 2. Task & Scheduling(2)

(\*) task, wait queue, run queue 간의 관계를 다른 각도에서 보여 주는 그림으로, wake\_up 함수가 호출되면, 대기 중이던 해당 task가 run queue로 이동하여 CPU를 할당 받게 된다(Scheduler가 그 역할을 담당함).



## 2. Task & Scheduling(3) - *schedule* 함수

(\*) *schedule()*, *waitqueue*, *runqueue* 등의 개념을 정확히 이해하면 앞으로 설명하게 될 *bottom half & deferring work* 관련 코드를 이해하는데도 많은 도움이 될 수 있겠다^^

***schedule()*** : *runqueue*에서 *process(task)*를 하나 꺼내어, 그것을 CPU에게 할당해 주는 것을 의미(*scheduler*가 작업해 줌). 여러 *kernel routine*에 의해서 직/간접적으로 호출됨.

### <직접 호출 방법 요약>

- 1) Insert ***current*** in the proper wait queue.  
→ *current process(task)*를 *wait queue*에 넣어둠.
- 2) Changes the state of ***current*** either to *TASK\_INTERRUPTIBLE* or to *TASK\_UNINTERRUPTIBLE*.  
→ *current process*의 상태를 *TASK\_INTERRUPTIBLE* 혹은 *TASK\_UNINTERRUPTIBLE*로 지정함.
- 3) Invokes ***schedule()***.  
→ *schedule* 함수를 호출함. 즉, 다른 *process(task)*에게 CPU를 사용할 기회를 줌.
- 4) Checks if the resource is available; if not, goes to step 2.  
→ *current process*가 사용할 *resource*가 사용가능한지 체크하여, 없으면 2로 jump하여 대기함.
- 5) Once the resource is available, removes ***current*** from the wait queue.  
→ *current process*가 사용할 *resource*가 사용 가능하면, *current process*를 *wait queue*에서 빼냄(*runqueue*로 이동함).



## 2. Task & Scheduling(4) - *sleeping & waking up*

(\*) 아래 내용은 이전 page의 내용을 *Sleeping & Waking up* 관점에서 다시 정리한 것임.  
(\*) TODO: 현재 version에 맞게 수정해야 함.

```
/* 'q' is the wait queue we wish to sleep on */
```

```
DECLARE_WAITQUEUE(wait, current);
```

```
add_wait_queue(q, &wait);
```

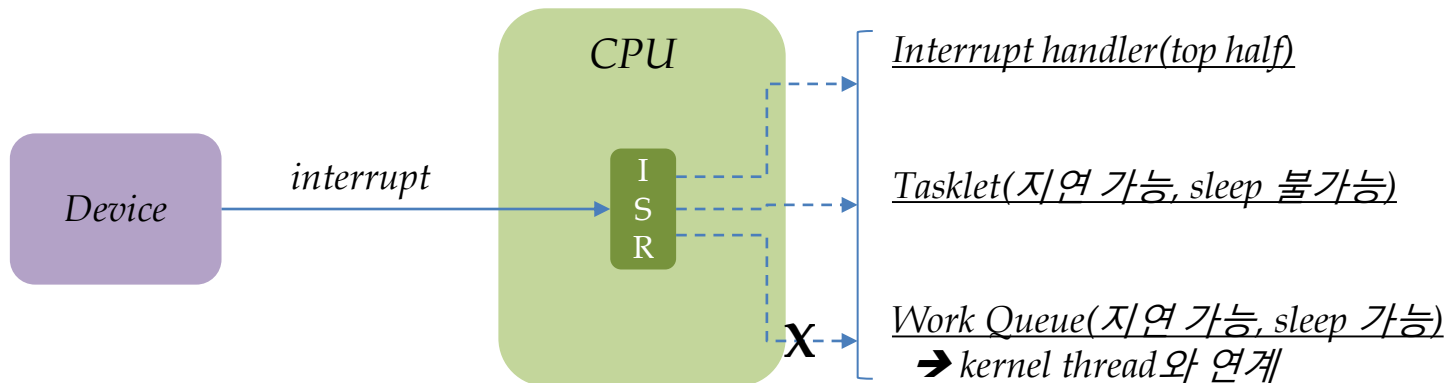
```
while (!condition) { /* condition is the event that we are waiting for */  
    set_current_state(TASK_INTERRUPTIBLE);  
    if (signal_pending(current)  
        /* handle signal */  
        schedule();  
}
```

```
set_current_state(TASK_RUNNING);
```

```
remove_wait_queue(q, &wait);
```

### 3. Bottom Halves and Deferring Work - 개념

- 0. Interrupt Handler를 top half라고 하며, 지연 처리(deferring work)가 가능한 루틴을 bottom half라고 함.
- 1. (bottom half 중에서도) 해당 작업이 sleep 가능하거나 sleep이 필요할 경우: work queue 사용  
→ *process context에서 실행*
- 2. 1의 조건에 해당하지 않으며 빠른 처리가 필수적인 경우: tasklet 사용  
→ *interrupt context에서 실행*  
→ *Softirq도 tasklet과 동일한 구조이나, 사용되는 내용이 정적으로 정해져 있음. 따라서 programming 시에는 동적인 등록이 가능한 tasklet이 사용된다고 이해하면 될 듯^^*
- 3. tasklet과 softirq의 관계와 마찬가지로, work queue는 kernel thread를 사용하여 구현되어 있음.



### 3. Bottom Halves and Deferring Work – *interrupt & process context*

- <interrupt context(=atomic context)의 제약 사항>

(\*) *user space*로의 접근이 불가능하다. *Process context*가 없으므로, 특정 *process*와 결합된 *user space*로 접근할 방법이 없다.

(\*) *current* 포인터(*현재 running 중인 task pointer*)는 *atomic mode*에서는 의미가 없으며, 관련 코드가 *interrupt* 걸린 *process*와 연결되지 않았으므로, 사용될 수 없다(*current pointer*에 대한 사용 불가).

(\*) *sleeping*이 불가하며, *scheduling*도 할 수 없다. *Atomic code*는 *schedule()* 함수를 *call*해서는 안되며, *wait\_event*나 *sleep*으로 갈 수 있는 어떠한 형태의 함수를 호출해서도 안된다. 예를 들어, *kmalloc(..., GFP\_KERNEL)* 을 호출해서는 안된다. *Semaphone*도 사용할 수 없다.

(\*) 위의 내용은 앞으로 설명할 *interrupt handler*와 *tasklet*에 모두 적용되는 내용임.

(\*) 반대로, *work queue*는 *process context*에서 동작하므로 위에서 제약한 사항을 모두 사용할 수 있음^^.

### 3. Bottom Halves and Deferring Work - Interrupt Handler(top half)

(\*) Bottom half를 설명하기에 앞서, top half(interrupt handler)를 먼저 언급할 필요가 있어, 이곳에 정리하였음.

```
int request_irq(unsigned int irq,  
               irqreturn_t (*handler)(int, void *),  
               unsigned long flags,  
               const char *devices,  
               void *dev_id);
```

→ Interrupt handler 등록

(\*) /proc/interrupts에서 인터럽트 상태를 확인할 수 있음 !

#### Interrupt handler

H/W interrupt가 발생할 때마다 호출됨

\*) 인터럽트 처리 중에 또 다른 인터럽트가 들어 올 수 있으니, 최대한 빠른 처리가 가능한 코드로 구성하게 됨.  
\*) 이 부분을 tasklet 코드로도 채울 수 있음 !

#### synchronize\_irq()

→ free\_irq를 호출하기 전에 호출하는 함수로, 현재 처리 중인 interrupt handler가 동작을 완료하기를 기다려 줌.

#### free\_irq()

→ 인터럽트 handler 등록 해제 함수

#### disable\_irq()

→ 해당 IRQ 라인에 대한 interrupt 리포팅을 못하도록 함.

#### disable\_irq\_nosync()

→ Interrupt handler가 처리를 끝내도록 기다리지 않고 바로 return 함

#### enable\_irq()

→ 해당 IRQ 라인에 대한 interrupt 리포팅을 하도록 함.

### 3. Bottom Halves and Deferring Work - *Interrupt Handler(top half)*

#### <Interrupt handler 사용 예>

```
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
#include <linux/interrupt.h>

#define SHARED_IRQ 17
static int irq = SHARED_IRQ, my_dev_id, irq_counter = 0;
module_param(irq, int, S_IRUGO);

static irqreturn_t my_interrupt(int irq, void *dev_id)
{
    irq_counter++;
    printk(KERN_INFO "In the ISR: counter = %d\n", irq_counter);
    return IRQ_NONE; /* we return IRQ_NONE because we are just observing */
}

static int __init my_init(void)
{
    if (request_irq
        (irq, my_interrupt, IRQF_SHARED, "my_interrupt", &my_dev_id))
        return -1;
    printk(KERN_INFO "Successfully loading ISR handler\n");
    return 0;
}

static void __exit my_exit(void)
{
    synchronize_irq(irq);
    free_irq(irq, &my_dev_id);
    printk(KERN_INFO "Successfully unloading, irq_counter = %d\n",
        irq_counter);
}

module_init(my_init);
module_exit(my_exit);
```

### 3. Bottom Halves and Deferring Work - Tasklet

(\*) tasklet과 softirq의 동작 원리는 동일함. 다만, softirq는 compile-time에 이미 내용(action)이 정해져 있으며, tasklet은 dynamic하게 등록할 수 있는 형태임.  
(\*) tasklet은 동시에 하나씩만 실행됨(count와 state 값을 활용)  
→ 이는 multi-processor 환경에서도 동일하게 적용됨.  
(\*) tasklet은 task 개념과는 전혀 무관하며, 또한 work queue와는 달리 Kernel thread를 필요로 하지 않음(그 만큼 간단한 작업을 처리한다고 보아야 할 듯^^).

tasklet list

my\_tasklet

my  
data

my\_tasklet\_handler  
{  
}  
}

reference count, state

tasklet\_init(&my\_tasklet, my\_tasklet\_handler)

or

DECLARE\_TASKLET(my\_tasklet, my\_tasklet\_handler, my\_data)

→ 초기화

tasklet\_schedule(&my\_tasklet)

→ 이것이 호출되면 tasklet handler 실행됨

my\_tasklet\_handler(my\_data) will be run !

→ 애는 빠르게 처리되는 코드이어야 함 !

### 3. Bottom Halves and Deferring Work - *Tasklet*

<data structure 일부 발췌 – include/linux/interrupt.h>

```
struct tasklet_struct
{
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    void (*func)(unsigned long);
    unsigned long data;
};

#define DECLARE_TASKLET(name, func, data) \
struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(0), func, data }

#define DECLARE_TASKLET_DISABLED(name, func, data) \
struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(1), func, data }

enum
{
    TASKLET_STATE_SCHED,    /* Tasklet is scheduled for execution */
    TASKLET_STATE_RUN       /* Tasklet is running (SMP only) */
};

#ifdef CONFIG_SMP
static inline int tasklet_trylock(struct tasklet_struct *t)
{
    return !test_and_set_bit(TASKLET_STATE_RUN, &(t)->state);
}

static inline void tasklet_unlock(struct tasklet_struct *t)
{
    smp_mb__before_clear_bit();
    clear_bit(TASKLET_STATE_RUN, &(t)->state);
}
```

### 3. Bottom Halves and Deferring Work - *Tasklet*

#### <tasklet 사용 예>

파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)

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

static void t_fun(unsigned long t_arg)
{
    struct my_dat *data = (struct my_dat *)t_arg;
    atomic_inc(&counter_bh);
    printk(KERN_INFO
           "In BH: counter_th = %d, counter_bh = %d, jiffies=%ld, %ld\n",
           atomic_read(&counter_th), atomic_read(&counter_bh),
           data->jiffies, jiffies);
}

static DECLARE_TASKLET(t_name, t_fun, (unsigned long)&my_data);

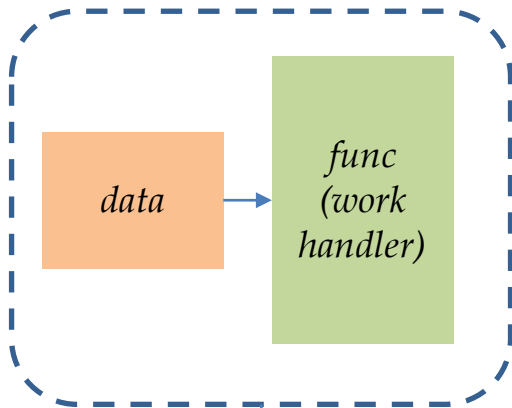
/* initialize tasklet */
static irqreturn_t my_interrupt(int irq, void *dev_id)
{
    struct my_dat *data = (struct my_dat *)dev_id;
    atomic_inc(&counter_th);
    data->jiffies = jiffies;
    tasklet_schedule(&t_name);
    mdelay(delay); /* hoke up a delay to try to cause pileup */
    return IRQ_NONE; /* we return IRQ_NONE because we are just observing */
}

module_init(my_generic_init);
module_exit(my_generic_exit);
```

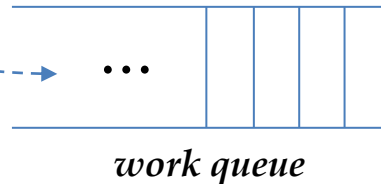


### 3. Bottom Halves and Deferring Work - Work Queue(default)

<my work - *work\_struct*>



*schedule\_work*  
or  
*schedule\_delayed\_work*



<기 정의된 *work thread*를 사용하는 경우>

*schedule\_work()*

→ 엔트리를 *work queue*에 추가함

*schedule\_delayed\_work()*

→ 엔트리를 *work queue*에 추가하고, 처리를 지연시킴

*flush\_scheduled\_work()*

→ *work queue*의 모든 엔트리를 처리(비움)

*cancel\_delayed\_work()*

→ *delayed work*(엔트리)를 취소함.

- (\*) *work queue*라고 하면, *work*, *queue*, *worker thread*의 세가지 요소를 통칭함.
- (\*) *work queue*를 위해서는 반드시 *worker thread*가 필요함.
- (\*) 기 정의된 *worker thread*(*events/0*)를 사용하는 방식과 새로운 *worker thread* 및 *queue*를 만드는 두 가지 방법이 존재함.
- (\*) *work queue*에서 사용하는 *work*는 *sleep*이 가능하다.
- (\*) *worker thread* 관련 보다 자세한 사항은 *kernel/workqueue.c* 파일 참조

<*worker thread*>

Set task state to *TASK\_INTERRUPTIBLE*

Add itself(thread) to wait queue  
(→ *sleep*)

If work list is empty then  
schedule  
Else  
set task state to *TASK\_RUNNING*

Remove itself from wait queue  
(→ *wakeup*)

Run work queue  
→ 여기서 *func* 함수 호출됨

loop

### 3. Bottom Halves and Deferring Work - Work Queue(사용자 정의)

#### <사용자 정의 work queue 관련 API 모음>

```
struct workqueue_struct *create_workqueue(const char  
                                     *name);
```

→ 사용자 정의 워크 큐 및 worker thread를 생성시켜 줌.

```
int queue_work(struct workqueue_struct *wq, struct  
              work_struct *work);
```

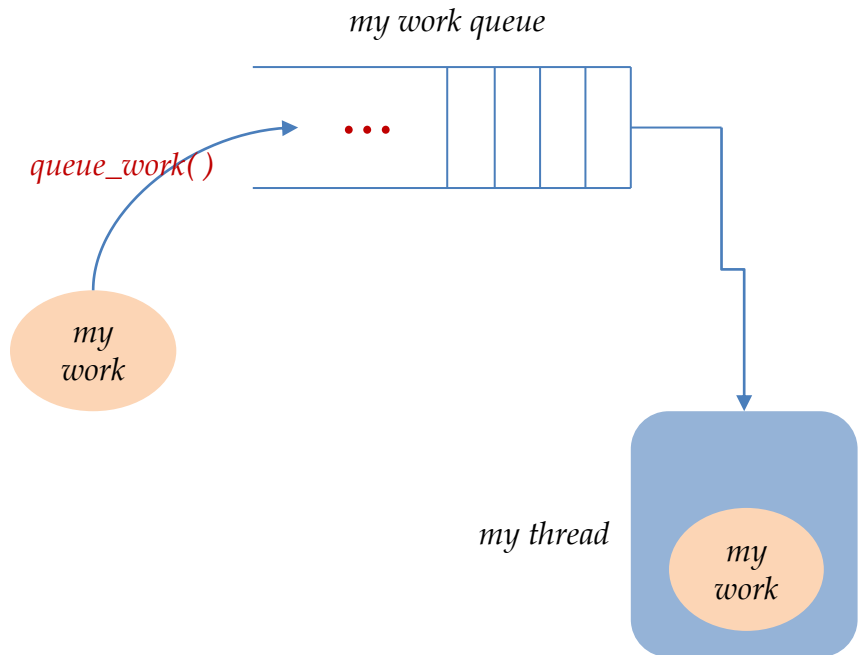
→ 사용자 정의 work을 사용자 정의 work queue에 넣고,  
schedule 요청함.

```
int queue_delayed_work(struct workqueue_struct *wq,  
                      struct work_struct *work,  
                      unsigned long delay);
```

→ queue\_work과 동일하나, delay 값을 주어, delay 후에  
schedule하도록 요청

```
void flush_workqueue(struct workqueue_struct *wq);
```

→ 사용자 정의 work queue에 있는 모든 work을 처리하여,  
queue를 비우도록 요청



(create\_workqueue에 인수로 넘겨준  
name 값이 thread name이 됨 - ps 명령으로  
확인 가능)

(\*) 사용자 정의 work queue를 생성하기 위해서는 create\_workqueue()를 호출하여야 하며,  
queue\_work() 함수를 사용하여 work을 queue에 추가해 주어야 한다.

(\*) 보통은 기 정의된 work queue를 많이 활용하나, 이는 시스템의 많은 driver 들이 공동으로  
사용하고 있으므로, 경우에 따라서는 원하는 결과(성능)를 얻지 못할 수도 있다. 따라서 이러한  
경우에는 자신만의 독자적인 work queue를 만드는 것도 고려해 보아야 한다.

(\*) 보다 자세한 사항은 include/linux/workqueue.h 파일 참조

### 3. Bottom Halves and Deferring Work - Work Queue

<data structure 일부 발췌 – include/linux/workqueue.h>

```
struct workqueue_struct;

struct work_struct;
typedef void (*work_func_t)(struct work_struct *work);

/*
 * The first word is the work queue pointer and the flags rolled into
 * one
 */
#define work_data_bits(work) ((unsigned long *)(&(work)->data))

struct work_struct {
    atomic_long_t data;
#define WORK_STRUCT_PENDING 0          /* T if work item pending execution */
#define WORK_STRUCT_STATIC 1          /* static initializer (debugobjects) */
#define WORK_STRUCT_FLAG_MASK (3UL)
#define WORK_STRUCT_WQ_DATA_MASK (~WORK_STRUCT_FLAG_MASK)
    struct list_head entry;
    work_func_t func;
#ifdef CONFIG_LOCKDEP
    struct lockdep_map lockdep_map;
#endif
};

#define WORK_DATA_INIT()    ATOMIC_LONG_INIT(0)
#define WORK_DATA_STATIC_INIT() ATOMIC_LONG_INIT(2)

struct delayed_work {
    struct work_struct work;
    struct timer_list timer;
};

static inline struct delayed_work *to_delayed_work(struct work_struct *work)
```

### 3. Bottom Halves and Deferring Work - Work Queue

#### <work queue 사용 예>

(\*) 아래 예는 default(system) work queue의 사용 예이며, 사용자 정의 work queue의 사용 예는 별도로 정리하여야 함^^.

```
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)

static struct my_dat {
    int irq;
    struct work_struct work;
} my_data;

static void w_fun(struct work_struct *w_arg)
{
    struct my_dat *data = container_of(w_arg, struct my_dat, work);
    atomic_inc(&bhs[data->irq]);
}

static irqreturn_t my_interrupt(int irq, void *dev_id)
{
    struct my_dat *data = (struct my_dat *)&my_data;
    data->irq = irq;
    atomic_inc(&interrupts[irq]);
    schedule_work(&data->work);
    mdelay(delay);
    /* we return IRQ_NONE because we are just observing */
    return IRQ_NONE;
}

static int my_init(void)
{
    struct my_dat *data = (struct my_dat *)&my_data;
    INIT_WORK(&data->work, w_fun);
    return my_generic_init();
}

module_init(my_init);
module_exit(my_generic_exit);
```

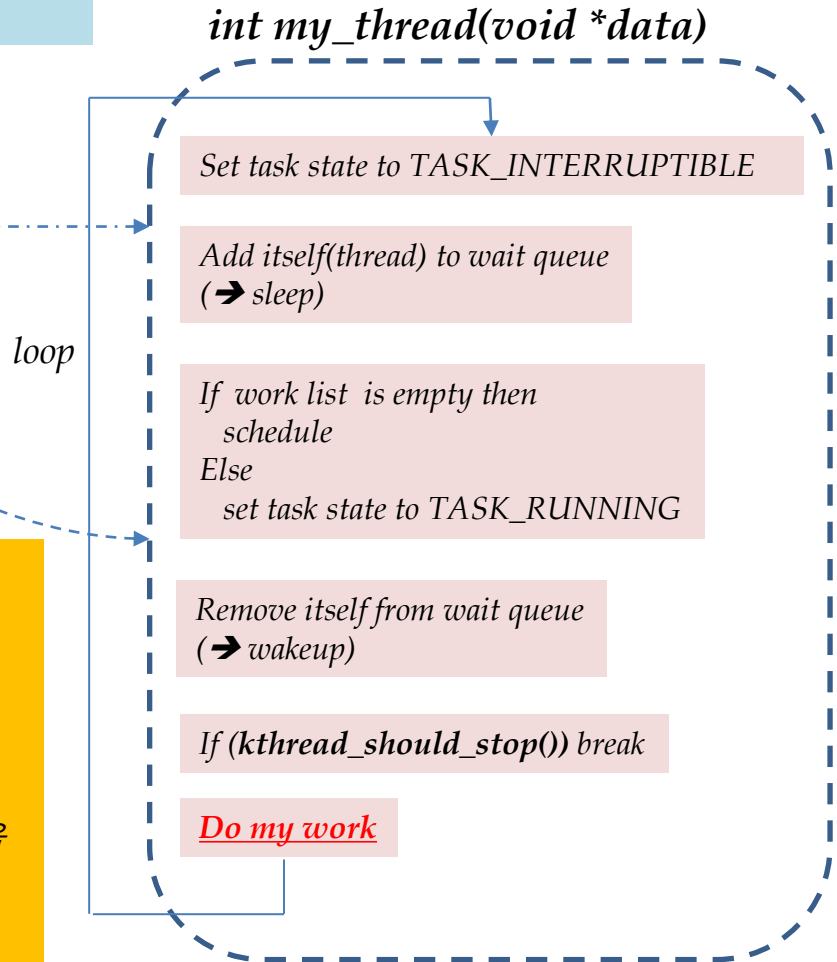
29,0-1 87%

### 3. Bottom Halves and Deferring Work - Kernel Threads

```
struct task_struct *kthread_create(my_thread, data, ...);  
→ Kernel thread 생성
```

(\*) 옆의 kernel thread 내부는 앞서 설명한 worker thread를 거의 그대로 복사한 것이라, 다음 다음 페이지의 코드 예를 보는 것이 보다 정확할 듯^^

```
kthread_run()  
→ kernel thread를 만들고, thread를 깨워줌  
kthread_create()  
→ kernel thread를 만들(sleeping 상태로 있음)  
kthread_bind()  
→ thread를 특정 CPU에 bind 시킬 때 사용함.  
kthread_stop()  
→ thread를 중지할 때 사용함. Kthread_should_stop을  
위한 조건을 설정해 줌.  
kthread_should_stop()  
→ kernel thread 루틴을 멈추기 위한 조건 검사 함수.
```



(\*) work queue가 kernel thread를 기반으로 하고 있으므로, kernel thread를 직접 만들 필요 없이, Work queue를 이용하는 것이 보다 간편할 수 있다.

(\*) 보다 자세한 사항은 include/linux/kthread.h 파일 참조

### 3. Bottom Halves and Deferring Work - *Kernel Threads*

<data structure 일부 발췌 – include/linux/kthread.h>

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

struct task_struct *kthread_create(int (*threadfn)(void *data),
                                   void *data,
                                   const char namefmt[], ...)
    __attribute__((format(printf, 3, 4)));

/*
 * kthread_run - create and wake a thread.
 * @threadfn: the function to run until signal_pending(current).
 * @data: data ptr for @threadfn.
 * @namefmt: printf-style name for the thread.
 *
 * Description: Convenient wrapper for kthread_create() followed by
 * wake_up_process(). Returns the kthread or ERR_PTR(-ENOMEM).
 */
#define kthread_run(threadfn, data, namefmt, ...) \
({ \
    struct task_struct *__k \
        = kthread_create(threadfn, data, namefmt, ## __VA_ARGS__); \
    if (!IS_ERR(__k)) \
        wake_up_process(__k); \
    __k; \
})

void kthread_bind(struct task_struct *k, unsigned int cpu);
int kthread_stop(struct task_struct *k);
int kthread_should_stop(void);

int kthreadd(void *unused);
extern struct task_struct *kthreadd_task;

#endif /* _LINUX_KTHREAD_H */
```

### 3. Bottom Halves and Deferring Work - Kernel Threads

#### <kernel thread 사용 예>

파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)

```
static int thr_fun(void *thr_arg)
{
    struct my_dat *data = (struct my_dat *)thr_arg;

    /* go into a loop and deal with events as they come */

    do {
        atomic_set(&cond, 0);
        wait_event_interruptible(wq, kthread_should_stop()
                                || atomic_read(&cond));
        if (atomic_read(&cond))
            atomic_inc(&counter_bh);
        printk
            (KERN_INFO
             "In BH: counter_th = %d, counter_bh = %d, jiffies=%ld, %ld\n",
             atomic_read(&counter_th), atomic_read(&counter_bh),
             data->jiffies, jiffies);
    } while (!kthread_should_stop());
    return 0;
}

static int __init my_init(void)
{
    atomic_set(&cond, 1);
    if (!(tsk = kthread_run(thr_fun, (void *)&my_data, "thr_fun"))) {
        printk(KERN_INFO "Failed to generate a kernel thread\n");
        return -1;
    }
    return my_generic_init();
}
```

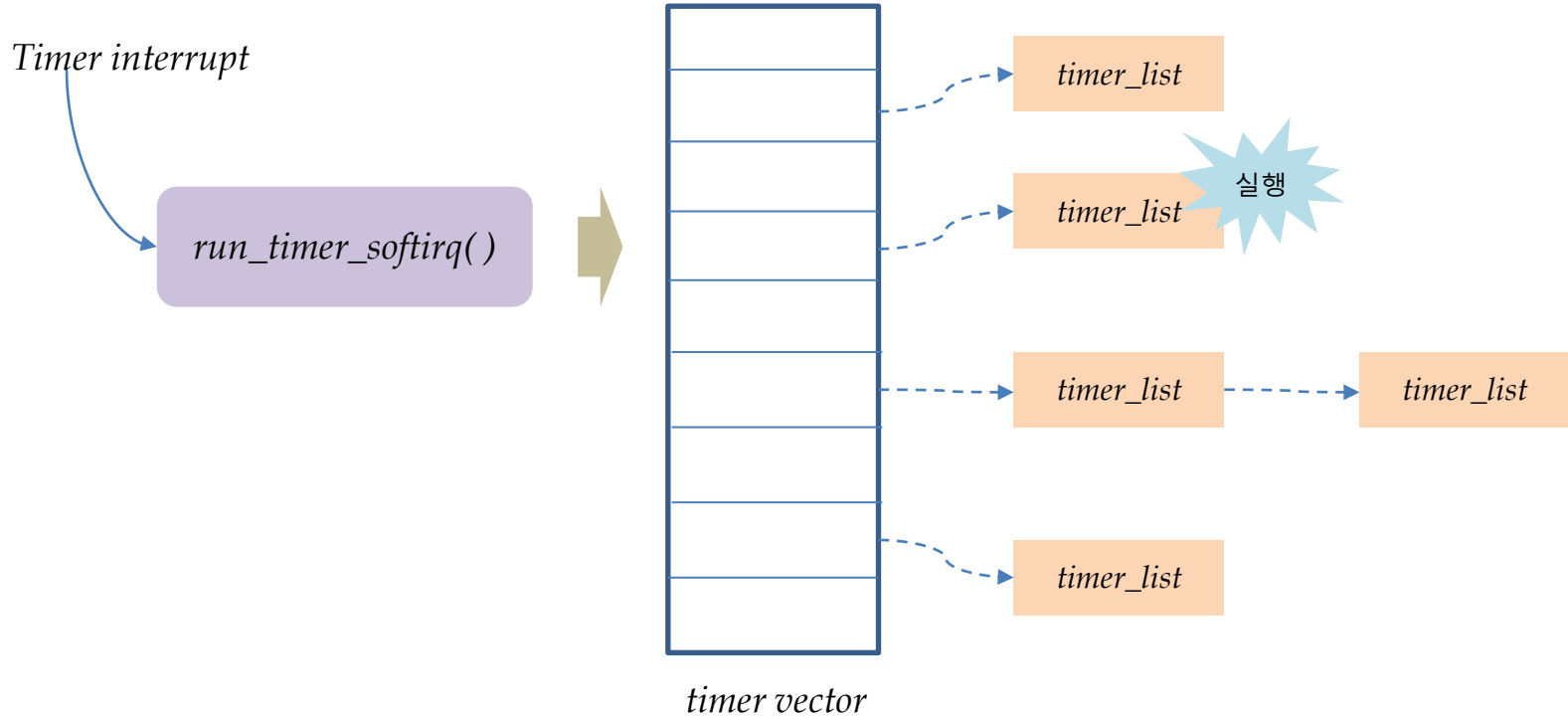
## 4. Timer(1)

(\*) 앞서 설명한 *bottom half*의 목적은 *work*을 단순히 *delay*시키는데 있는 것이 아니라, 지금 당장 *work*을 실행하지 않는데 있음. 한편 *timer*는 일정한 시간 만큼 *work*을 *delay*시키는데 목적이 있음 !

(\*) *timer*는 *timer interrupt*를 통해 동작하는 방식을 취함(*software interrupt*).

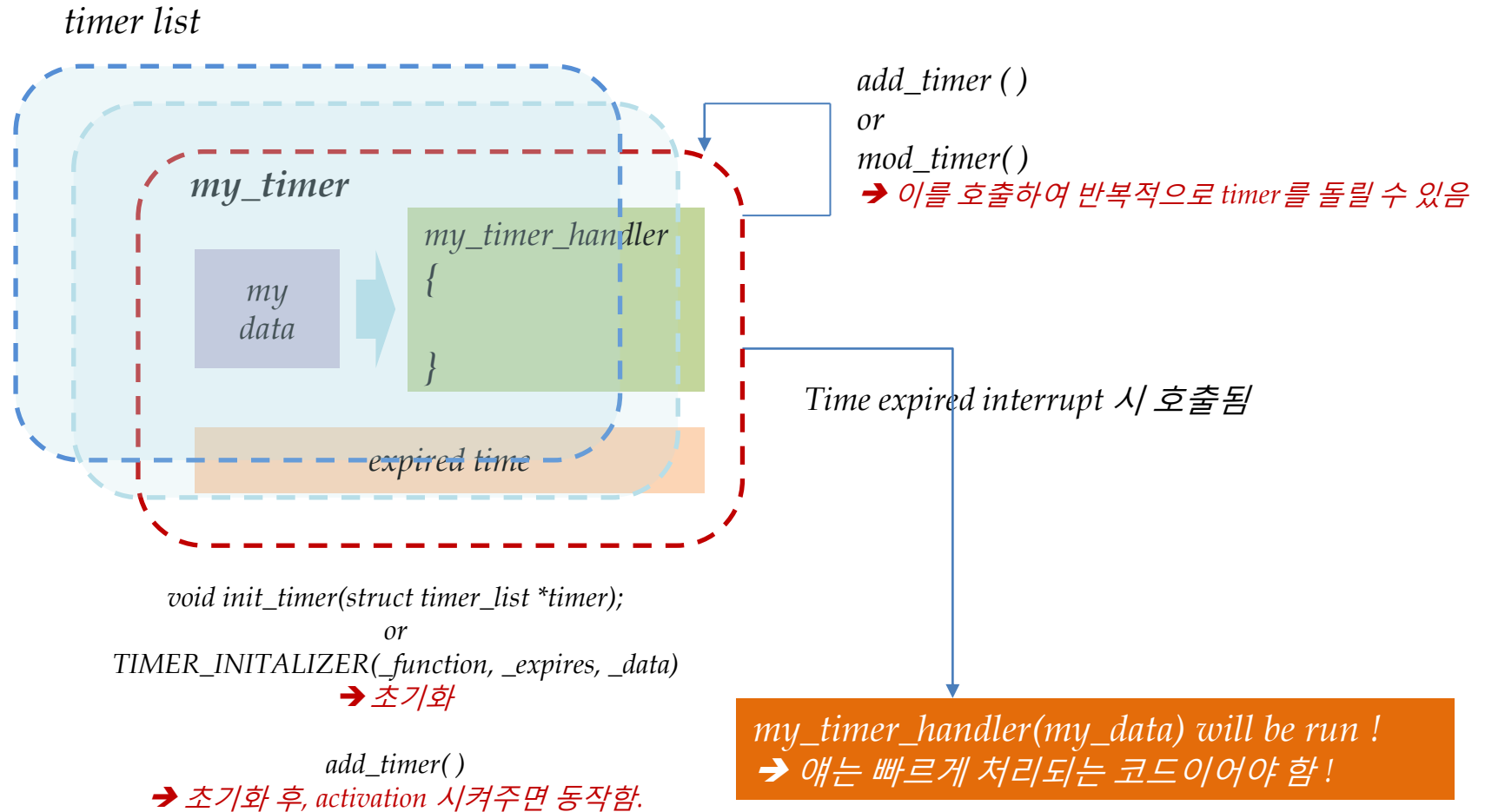
즉, 처리하려는 *function*을 준비한 후, 정해진 시간이 지나면 *timer interrupt*가 발생하여 해당 *function*을 처리하는 구조임.

(\*) *timer*는 *cyclic*(무한 반복) 구조가 아니므로, *time*이 경과하면 *timer function*이 실행되고, 해당 *timer*는 제거된다.





## 4. Timer(2)



## 4. Timer(3)

***schedule\_timeout(timeout)*** : 현재 실행 중인 task에 대해 delay를 줄 수 있는 보다 효과적인 방법. 이 방법을 사용하면 현재 실행 중인 task를 지정된 시간이 경과할 때까지 sleep 상태(wait queue에 넣어 줌)로 만들어 주고, 시간 경과 후에는 다시 runqueue에 가져다 놓게 함. *schedule\_timeout()*의 내부는 timer와 *schedule* 함수로 구성되어 있음.

```
schedule_timeout (signed long timeout)
{
    timer_t timer;
    unsigned long expire;

    ...
    expire = timeout * jiffies;

    init_timer(&timer);
    timer.expires = expire;
    timer.data = (unsigned long)current;
    timer.function = process_timeout;

    add_timer(&timer);
    schedule();
    del_timer_sync(&timer);

    timeout = expire - jiffies;
    ...
}
```

(\*) *schedule\_timeout* 말고도, process scheduling과 조합한 타이머 리스트 관련 함수로는 아래와 같은 것들이 있다.  
*process\_timeout, sleep\_on\_timeout, interruptible\_sleep\_on\_timeout*

## 4. Timer(4)

<data structure 일부 발췌 – include/linux/timer.h>

```
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)

struct timer_list {
    /*
     * All fields that change during normal runtime grouped to the
     * same cacheline
     */
    struct list_head entry;
    unsigned long expires;
    struct tvec_base *base;

    void (*function)(unsigned long);
    unsigned long data;

    int slack;

#ifdef CONFIG_TIMER_STATS
    void *start_site;
    char start_comm[16];
    int start_pid;
#endif
#ifdef CONFIG_LOCKDEP
    struct lockdep_map lockdep_map;
#endif
};

extern struct tvec_base boot_tvec_bases;

#ifdef CONFIG_LOCKDEP
/*
 * NB: because we have to copy the lockdep_map, setting the lockdep_map key
    40,1

```

## 4. Timer(5)

### <timer 사용 예>

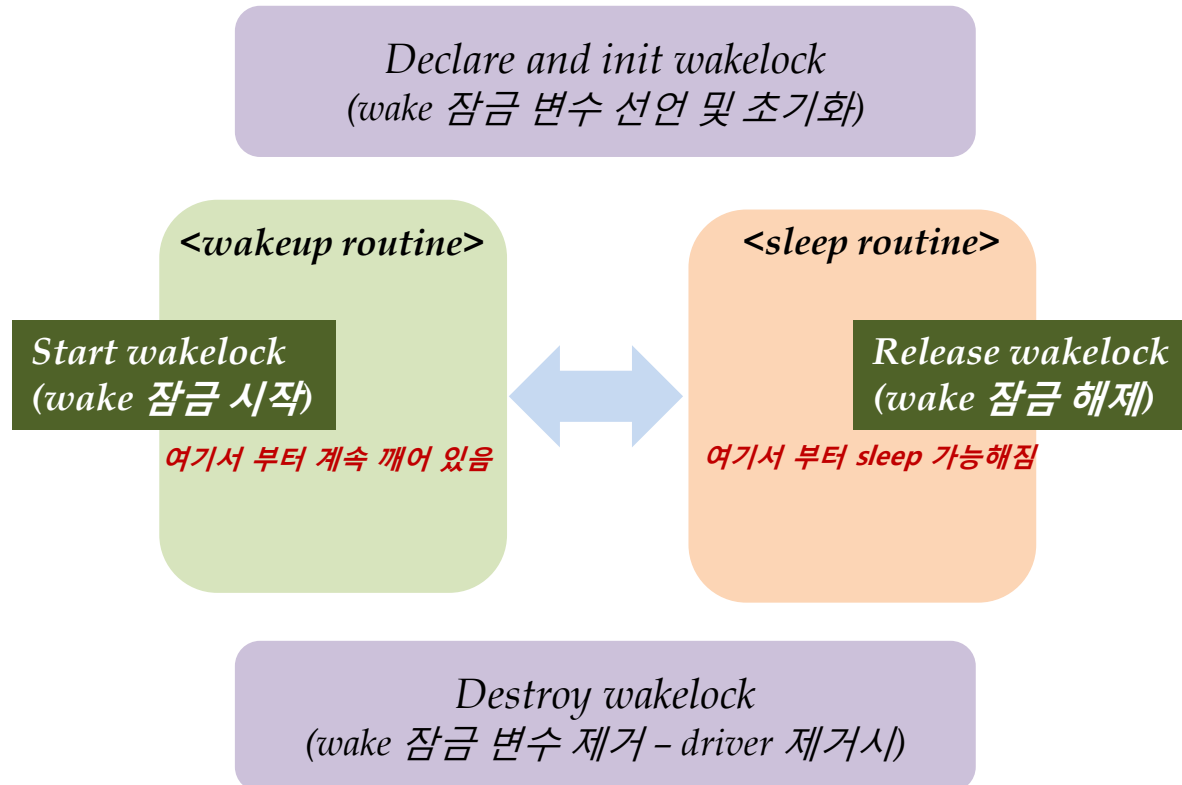
```
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
static struct timer_list my_timer;

static void my_timer_function(unsigned long ptr)
{
    printk(KERN_INFO "I am in my_timer_fun, jiffies = %ld\n", jiffies);
    printk(KERN_INFO " I think my current task pid is %d\n",
        (int)current->pid);
    printk(KERN_INFO " my data is: %d\n", (int)ptr);
}

static ssize_t
mycdrv_write(struct file *file, const char __user * buf, size_t lbuf,
    loff_t * ppos)
{
    static int len = 100;
    printk(KERN_INFO " Entering the WRITE function\n");
printk(KERN_INFO "my current task pid is %d\n", (int)current->pid);
    init_timer(&my_timer); /* intialize */
    my_timer.function = my_timer_function;
    my_timer.expires = jiffies + HZ; /* one second delay */
    /* my_timer.data = (void *) len; */
    my_timer.data = len;
    printk(KERN_INFO "Adding timer at jiffies = %ld\n", jiffies);
    add_timer(&my_timer);
len += 100;
    return lbuf;
}
```

## 5. Wakelock(1)

- (\*) 시스템이 low power state로 가는 것을 막아주는 메카니즘(google에서 만듦).
- (\*) Smart Phone은 전류를 많이 소모하므로, 항상 sleep mode로 빠질 준비를 해야 한다.
- (\*) wake\_lock\_init의 인자로 넘겨준, name 값은 /proc/wakelocks에서 확인 가능함.



## 5. Wakelock(2)

### <Wakelock 관련 API 모음>

**[변수 선언]** `struct wakelock mywakelock;`

**[초기화]** `wake_lock_init(&mywakelock, int type, "wakelock_name");`

→ type :

= WAKE\_LOCK\_SUSPEND: 시스템이 suspending 상태로 가는 것을 막음

= WAKE\_LOCK\_IDLE: 시스템이 low-power idle 상태로 가는 것을 막음.

**[To hold(wake 상태로 유지)]** `wake_lock(&mywakelock);`

**[To release(sleep 상태로 이동)]** `wake_unlock(&mywakelock);`

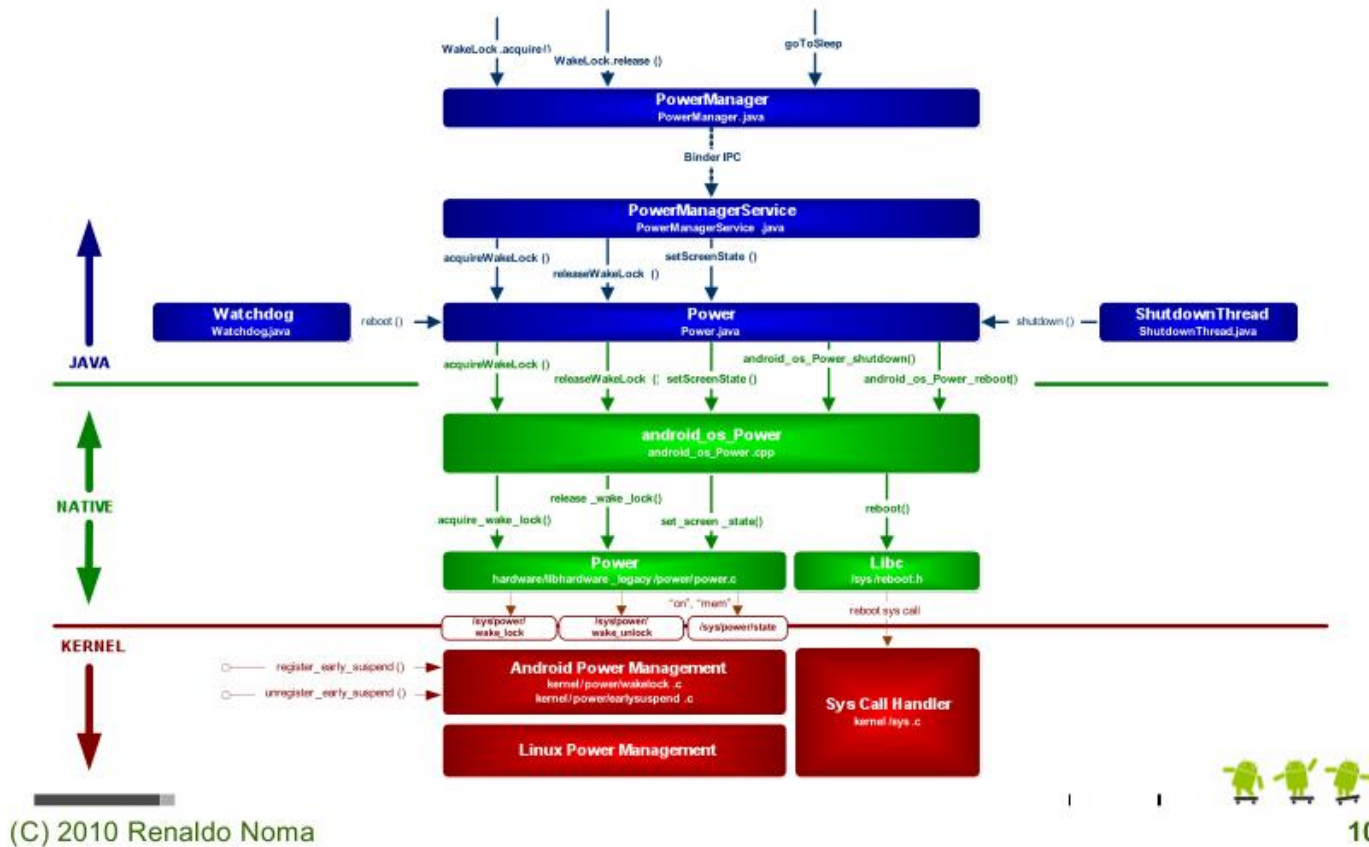
**[To release(sleep 상태로 이동)]** `wake_lock_timeout(&mywakelock, HZ);`

**[제거]** `wake_lock_destroy (&mywakelock);`

## 5. Wakelock(3)

(\*) 이 그림은 kernel wakelock을 설명하기에는 많이 부족하나, Android 관점에서 power manangement를 이해하기 위해 첨부하였다.

# Android Power Management Design



## 5. Wakelock(4)

### <data structure 일부 발췌 – include/linux/wakelock.h>

```
enum {
    WAKE_LOCK_SUSPEND, /* Prevent suspend */
    WAKE_LOCK_IDLE,    /* Prevent low power idle */
    WAKE_LOCK_TYPE_COUNT
};
```

```
struct wake_lock {
#ifdef CONFIG_HAS_WAKELOCK
    struct list_head    link;
    int                 flags;
    const char          *name;
    unsigned long        expires;
#ifdef CONFIG_WAKELOCK_STAT
    struct {
        int             count;
        int             expire_count;
        int             wakeup_count;
        ktime_t          total_time;
        ktime_t          prevent_suspend_time;
        ktime_t          max_time;
        ktime_t          last_time;
    } stat;
#endif
#endif
};
```

```
#ifdef CONFIG_HAS_WAKELOCK
```

```
void wake_lock_init(struct wake_lock *lock, int type, const char *name);
void wake_lock_destroy(struct wake_lock *lock);
void wake_lock(struct wake_lock *lock);
void wake_lock_timeout(struct wake_lock *lock, long timeout);
void wake_unlock(struct wake_lock *lock);
```



## 5. Wakelock(5)

### <wakelock 사용 예>

```
static struct workqueue_struct *workqueue;  
static struct wake_lock mmc_delayed_work_wake_lock;
```

```
static int __init mmc_init(void)  
{  
    int ret;  
  
    wake_lock_init(&mmc_delayed_work_wake_lock, WAKE_LOCK_SUSPEND, "mmc_delayed_work");  
  
    workqueue = create_freezeable_workqueue("kmmcd");  
    if (!workqueue)  
        return -ENOMEM;  
}
```

```
static int mmc_schedule_delayed_work(struct delayed_work *work,  
                                     unsigned long delay)  
{  
    wake_lock(&mmc_delayed_work_wake_lock);  
    return queue_delayed_work(workqueue, work, delay);  
}
```

(\*) 위의 내용은 *drivers/mmc/core/core.c* 파일에서 발췌한 것임.

## 5. Wakelock(5)

### <wakelock 사용 예(계속)>

```
void mmc_host_deeper_disable(struct work_struct *work)
{
    struct mmc_host *host =
        container_of(work, struct mmc_host, disable.work);

    /* If the host is claimed then we do not want to disable it anymore */
    if (!mmc_try_claim_host(host))
        goto out;
    mmc_host_do_disable(host, 1);
    mmc_do_release_host(host);

out:
    wake_unlock(&mmc_delayed_work_wake_lock);
}
```

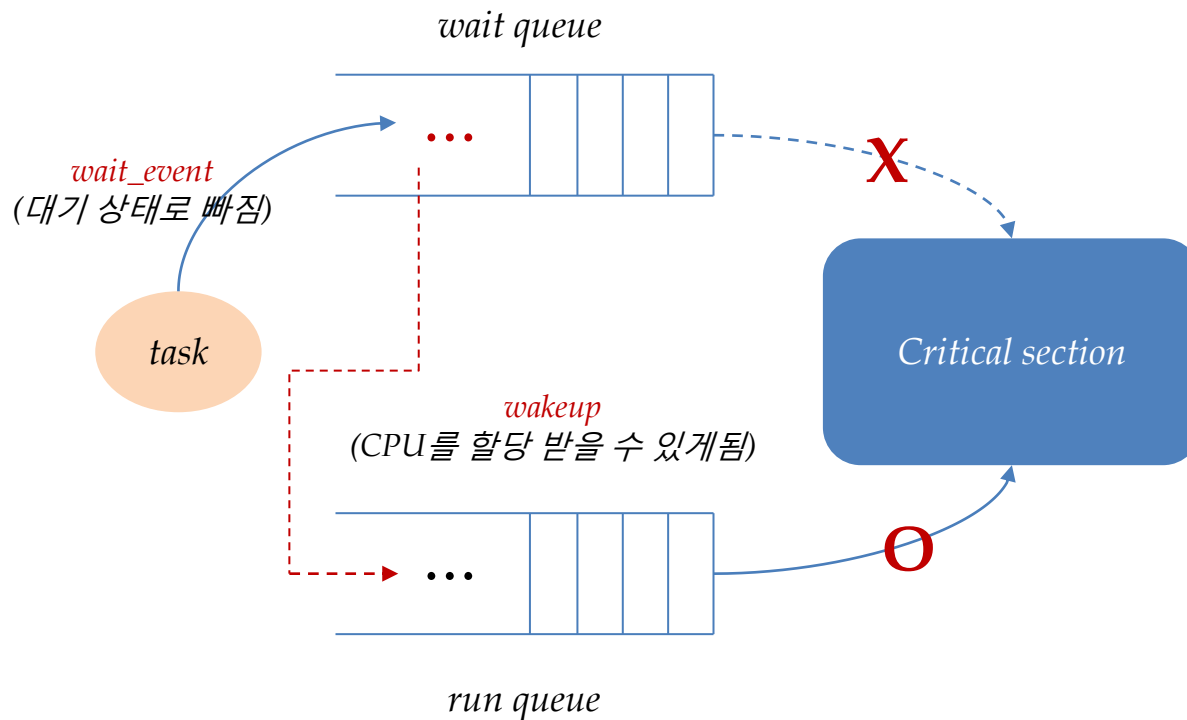
```
    mmc_release_host(host);
    mmc_power_off(host);

out:
    if (extend_wakelock)
        wake_lock_timeout(&mmc_delayed_work_wake_lock, HZ / 2);
    else
        wake_unlock(&mmc_delayed_work_wake_lock);

    if (host->caps & MMC_CAP_NEEDS_POLL)
        mmc_schedule_delayed_work(&host->detect, HZ);
```

## 6. Synchronization(1) - *Sleeping & Wait Queue*

(\*) *Spinlock, big kernel lock, mutex, semaphore* 등 일반적인 내용은 제외하고, *wait queue* 및 *Completion function*만 언급하고자 함.



## 6. Synchronization(2) - Sleeping & Wait Queue

(\*) *wait queue*는 *kernel mode*에서 *running* 중인 *task*가 특정 조건이 만족될 때까지 기다려야 할 때 사용된다.

(\*) *task*가 필요로 하는 특정 조건이나 *resource*가 준비될 때까지, 해당 *task*는 *Sleep* 상태에 있어야 한다.

### <변수 선언 및 초기화>

```
wait_queue_head_t wq;
```

```
init_waitqueue_head(&wq);
```

or

```
DECLARE_WAIT_QUEUE_HEAD(wq);
```

### <Going to Sleep → critical section으로 들어가기 전에 대기 상태로 빠짐>

```
wait_event (wait_queue_head_t wq, int condition)
```

```
wait_event_interruptible (wait_queue_head_t wq, int condition)
```

```
wait_event_killable (wait_queue_head_t wq, int condition)
```

```
wait_event_timeout (wait_queue_head_t wq, int condition, long timeout)
```

```
wait_event_interruptible (wait_queue_head_t wq, int condition, long timeout)
```

...

### <Waking Up → critical section으로 들어가는 조건을 만들어줌(풀어줌)>

```
void wake_up (wait_queue_head_t *wq);
```

```
void wake_up_interruptible (wait_queue_head_t *wq);
```

```
void wake_up_interruptible_sync (wait_queue_head_t *wq);
```

...

## 6. Synchronization(3) - *Sleeping & Wait Queue*

### <Wait Queue 사용 예>

```
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
#include "lab_miscdev.h"

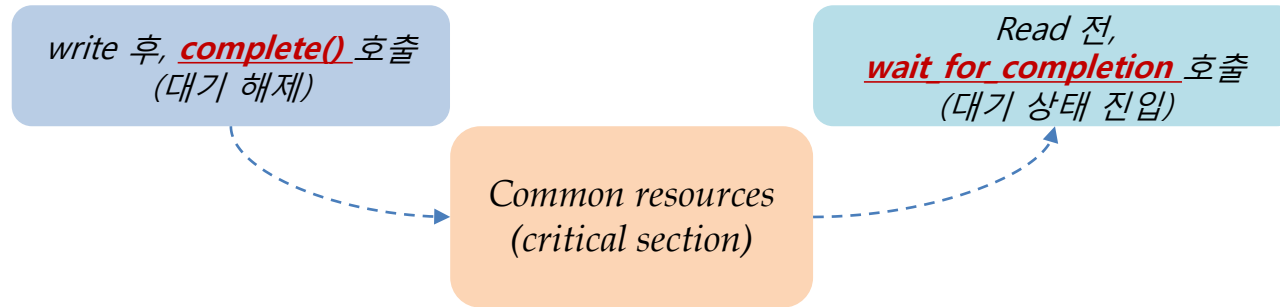
static DECLARE_WAIT_QUEUE_HEAD(wq);

static atomic_t data_ready;

static ssize_t
mycdrv_read(struct file *file, char __user * buf, size_t lbuf, loff_t * ppos)
{
    printk(KERN_INFO "process %i (%s) going to sleep\n", current->pid,
           current->comm);
    wait_event_interruptible(wq, (atomic_read(&data_ready)));
    printk(KERN_INFO "process %i (%s) awakening\n", current->pid,
           current->comm);
    return mycdrv_generic_read(file, buf, lbuf, ppos);
}

static ssize_t
mycdrv_write(struct file *file, const char __user * buf, size_t lbuf,
             loff_t * ppos)
{
    int nbytes = mycdrv_generic_write(file, buf, lbuf, ppos);
    printk(KERN_INFO "process %i (%s) awakening the readers...\n",
           current->pid, current->comm);
    atomic_set(&data_ready, 1);
    wake_up_interruptible(&wq);
    return nbytes;
}
```

## 6. Synchronization(4) - Completion



```
struct completion {  
    unsigned int done;  
    wait_queue_head_t wait;  
};
```

```
void init_completion(struct completion *c); /* DECLARE_COMPLETION(x)도 사용 가능 */
```

→ completion 초기화

```
void wait_for_completion(struct completion *c); /* timeout 함수도 있음 */
```

→ critical section에 들어갈 때 호출(대기를 의미함)

```
int wait_for_completion_interruptible(struct completion *c); /* timeout 함수도 있음 */
```

→ critical section에 들어갈 때 호출(대기를 의미함). 이 함수 호출 동안에 Interrupt 가능함.

```
void complete(struct completion *c);
```

→ critical section에 들어갈 수 있도록 해줌(대기 조건을 해지해 줌)

```
void complete_and_exit(struct completion *c, long code);
```

## 6. Synchronization(5) - Completion

### <completion 사용 예>

```
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)

static DECLARE_COMPLETION(my_wait);

static ssize_t
mycdrv_read(struct file *file, char __user * buf, size_t lbuf, loff_t * ppos)
{
    printk(KERN_INFO "process %i (%s) going to sleep\n", current->pid,
        current->comm;);
    wait_for_completion(&my_wait);
    printk(KERN_INFO "process %i (%s) awakening\n", current->pid,
        current->comm);
    return mycdrv_generic_read(file, buf, lbuf, ppos);
}

static ssize_t
mycdrv_write(struct file *file, const char __user * buf, size_t lbuf,
    loff_t * ppos)
{
    int nbytes = mycdrv_generic_write(file, buf, lbuf, ppos);

    printk(KERN_INFO "process %i (%s) awakening the readers...\n",
        current->pid, current->comm;);
    complete(&my_wait);
    return nbytes;
}


```

## 7. Notifier(1)

(\*) *notifier*는 서로 다른 곳에 위치한 *kernel code* 간에 *event*를 전송하기 위한 *mechanism*으로 *callback* 함수 개념으로 생각하면 이해가 쉽다^^.

(\*) 즉, *kernel routine A*에서는 호출되기를 원하는 *callback* 함수를 기술 및 등록하고, *event* 발생 시점을 아는 *kernel routine B*에서 해당 함수를 호출해 주는 것으로 설명할 수 있겠음 !

**<kernel routine A>**

```
{  
    notifier callback 함수 정의/  
    notifier callback 함수 등록  
  
}
```

```
my_callback_func()  
{  
  
}
```

**<event 발생 시점 예 - bluetooth>**

- 1) *HCI\_DEV\_UP*(link up 시)
- 2) *HCI\_DEV\_DOWN*(linux down 시)
- 3) *HCI\_DEV\_REG*
- 4) *HCI\_DEV\_UNREG*
- 5) *HCI\_DEV\_WRITE*(패킷 전송 시)

**<kernel routine B>**

```
{  
    notifier chain register  
  
    ...  
    notifier chain unregister  
}
```

```
some_func()  
{  
    call notifier_callback func  
    ➔ 특정 event 발생 시점에서 호출  
}
```



## 7. Notifier(2)

(\*) 자세한 사항은 `include/linux/notifier.h` 파일 참조 !

```
struct notifier_block {
    int (*notifier_call)(struct notifier_block *, unsigned long, void *);
    struct notifier_block *next;
    int priority;
};

extern int atomic_notifier_chain_register(struct atomic_notifier_head *nh,
    struct notifier_block *nb);
extern int blocking_notifier_chain_register(struct blocking_notifier_head *nh,
    struct notifier_block *nb);
struct atomic_notifier_head {
    spinlock_t lock;
    struct notifier_block *head;
};
extern int raw_notifier_chain_register(struct raw_notifier_head *nh,
    struct notifier_block *nb);
extern int srcu_notifier_chain_register(struct srcu_notifier_head *nh,
    struct notifier_block *nb);

struct blocking_notifier_head {
    struct rw_semaphore rwsem;
    struct notifier_block *head;
};
extern int blocking_notifier_chain_cond_register(
    struct blocking_notifier_head *nh,
    struct notifier_block *nb);
extern int atomic_notifier_chain_unregister(struct atomic_notifier_head *nh,
    struct notifier_block *nb);
extern int blocking_notifier_chain_unregister(struct blocking_notifier_head *nh,
    struct notifier_block *nb);
extern int raw_notifier_chain_unregister(struct raw_notifier_head *nh,
    struct notifier_block *nb);
extern int srcu_notifier_chain_unregister(struct srcu_notifier_head *nh,
    struct notifier_block *nb);

extern int atomic_notifier_call_chain(struct atomic_notifier_head *nh,
    unsigned long val, void *v);
extern int __atomic_notifier_call_chain(struct atomic_notifier_head *nh,
    unsigned long val, void *v, int nr_to_call, int *nr_calls);
extern int blocking_notifier_call_chain(struct blocking_notifier_head *nh,
    unsigned long val, void *v);
extern int __blocking_notifier_call_chain(struct blocking_notifier_head *nh,
    unsigned long val, void *v, int nr_to_call, int *nr_calls);
```

## 7. Notifier(3)

### <notifier 사용 예>

파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)

```
static BLOCKING_NOTIFIER_HEAD(my_nh);
static int my_notifier_call(struct notifier_block *b, unsigned long event,
                           void *data)
{
    long *c = (long *)data;
    *c += 100;
    printk(KERN_INFO "\n ..... I was called with event = %ld data=%ld\n",
           event, *c);
    return NOTIFY_OK;
}

static struct notifier_block my_nh_block = {
    .notifier_call = my_notifier_call,
    .priority = 0,
};

static long counter = 0;

static int __init my_init(void)
{
    int rc;
    if (blocking_notifier_chain_register(&my_nh, &my_nh_block)) {
        printk(KERN_INFO "Failed to register with notifier\n");
        return -1;
    }

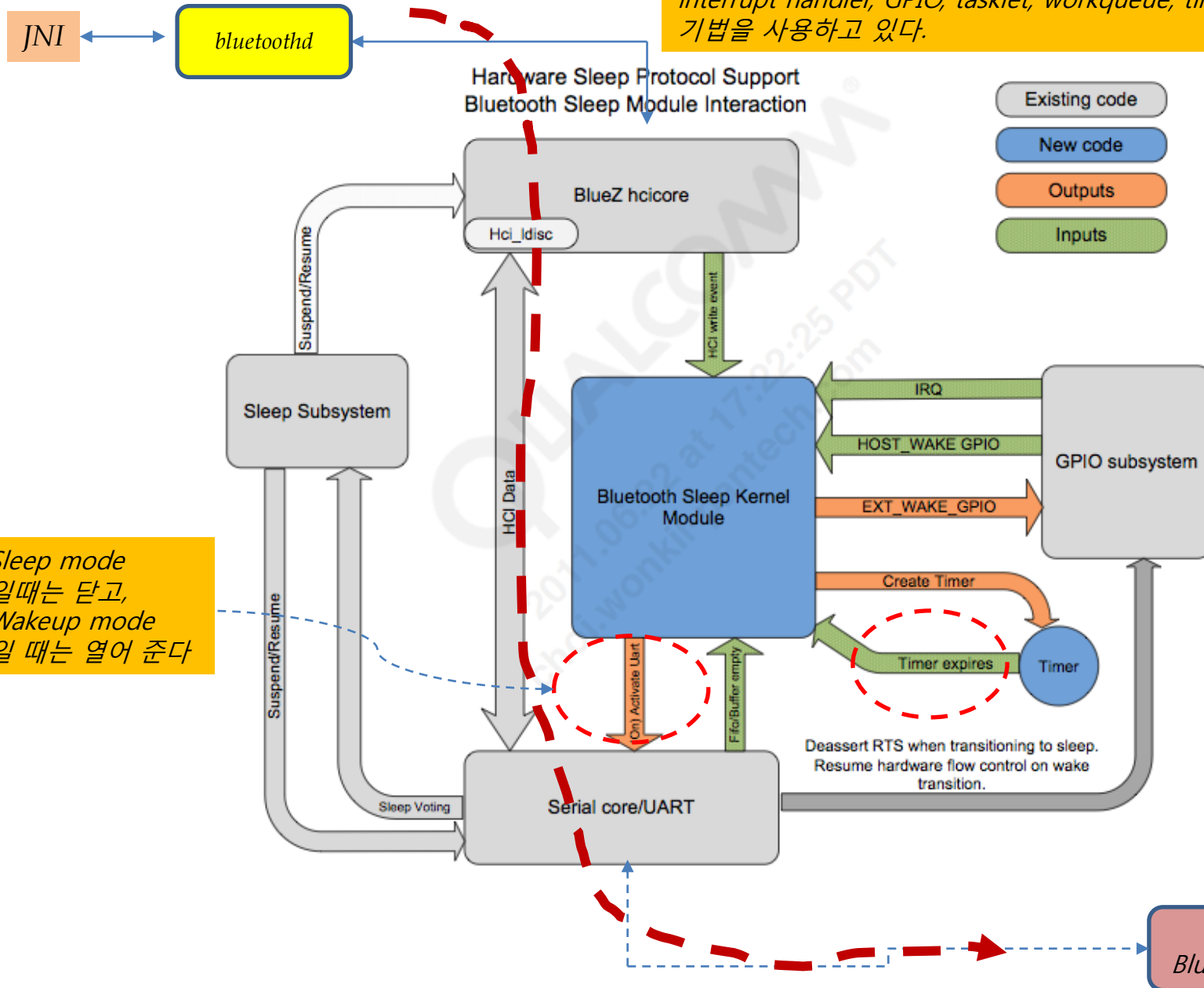
    rc = blocking_notifier_call_chain(&my_nh, 1000, &counter);
}
```

37,0-1

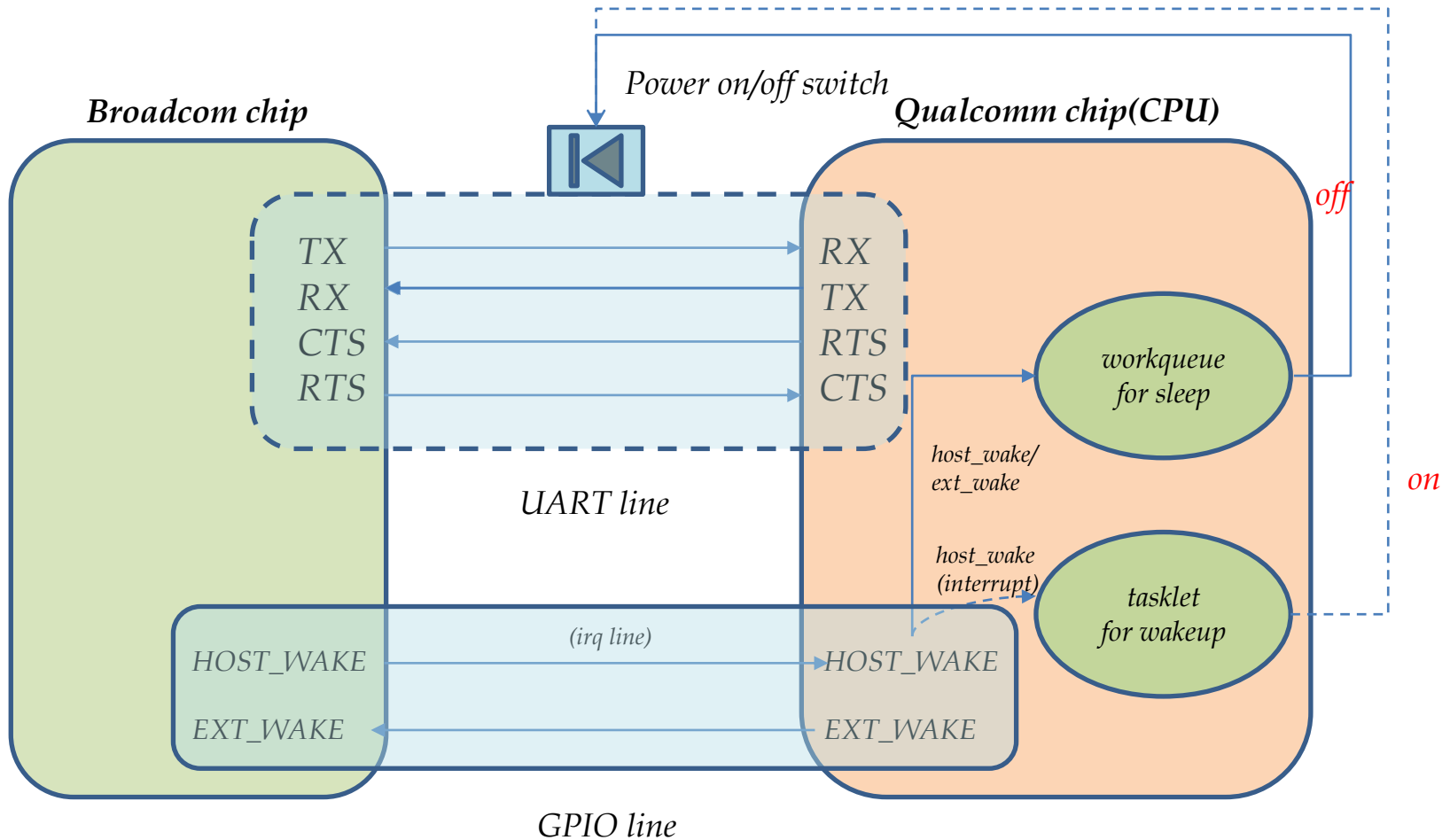
62%

## 8. Example Driver : *bluetooth sleep mode driver(1)*

(\*) 아래 그림은 *bluetooth sleep driver*의 개념도로써, *platform\_driver*, *interrupt handler*, *GPIO*, *tasklet*, *workqueue*, *timer*, *wakelock* 등의 기법을 사용하고 있다.



## 8. Example Driver : *bluetooth sleep mode driver(2)*



(\*) bluetooth가 wakeup되는 조건은 위의 HOST\_WAKE가 enable(interrupt)되는 것 이외에도 실제로 bluetooth packet이 나가고 나서 발생하는 HCI event(callback)에 기인하기도 한다.

(\*) 전력 소모를 최소화 하기 위해, 틈만 나면(?) sleep mode로 진입해야 하며, HOST\_WAKE 및 EXT\_WAKE GPIO pin이 모두 사용중이지 않을 때(deasserted), sleep으로 들어가게 된다.

## 8. Example Driver : *bluetooth sleep mode driver*(3)

- *<tasklet routine>*

*TODO*

## 8. Example Driver : *bluetooth sleep mode driver(4)*

- *<work queue routine>*

TODO

## 9. Debug

- TODO

➔ *Debug* 관련해서는 일단은 *Android\_Debug\_Guide4.pdf* 파일을 참조하기 바람(추후, 좀 더 보강 예정임).

# *Appendix*



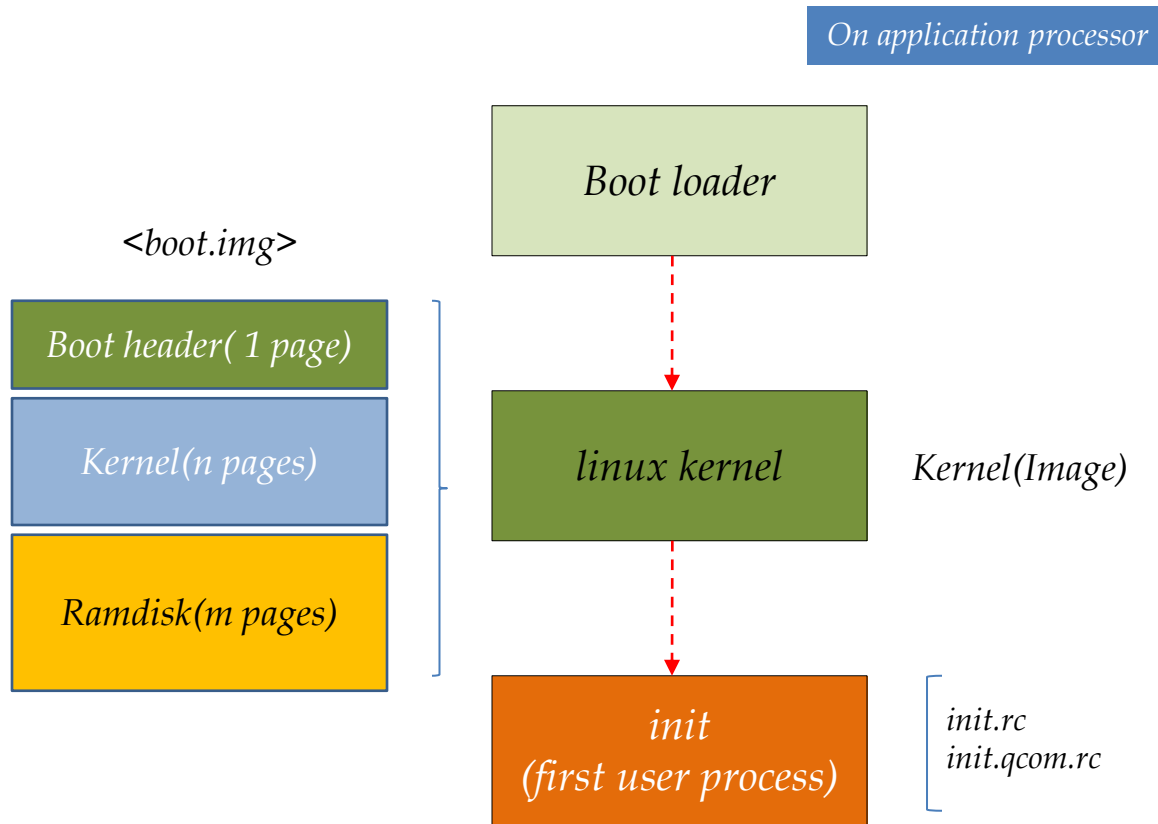
## Appendix

- 부록 1: *kernel(만) build 방법*
- 부록 2: *android boot.img 분해 및 kernel 교체 방법*

## 부록 1: Kernel(만) Build 하기

- `# make ARCH=arm YOUR_MACHINE_ARCH_CONFIG` ← *arch/arm/configs*  
→ 사용중인 machine type으로 configuratiobn 설정(조정)
- `# make ARCH=arm menuconfig`  
→ Configuration을 원하는대로 변경하고자 할 경우
- `# make -j3 ARCH=arm zImage`  
→ Build 결과: *arch/arm/boot/zImage*
- `# make -j3 ARCH=arm modules`  
→ wi-fi 등 kernel module build 함.
- [TIP] menuconfig로 수정한 파일을, 이후에도 재 사용하고자 할 경우에는, 아래와 같이 하면 된다.  
→ `cp -f .config arch/arm/configs/YOUR_MACHINE_ARCH_CONFIG`

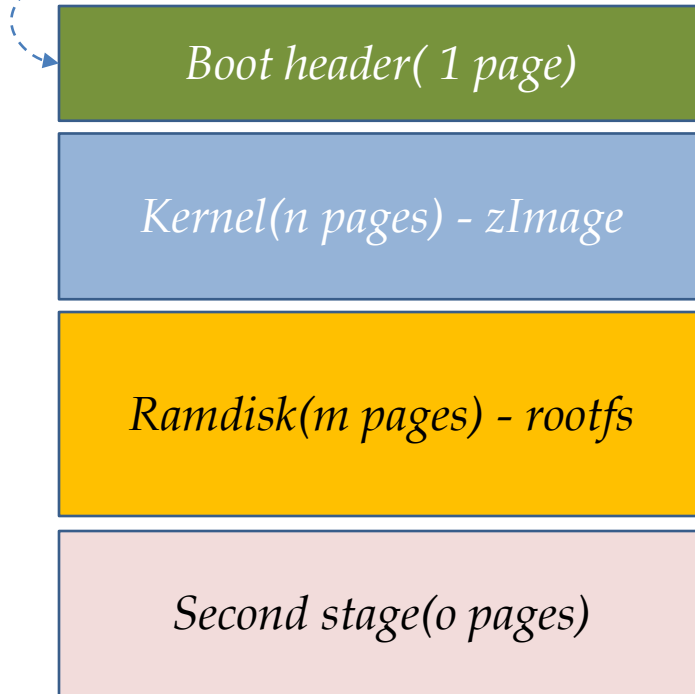
## 부록 2: Android boot.img 분해 및 kernel 교체 방법(1)



## 부록 2: Android boot.img 분해 및 kernel 교체 방법(2)

```
mkbootimg --cmdline "console=ttyHS1 pmem_kernel_ebi1_size=0x200000  
androidboot.hardware=qcom"  
--kernel zImage --ramdisk ramdisk.gz --base 0x20000000 -o boot.img
```

**ANDROID!**



$$n = (\text{kernel\_size} + \text{page\_size} - 1) / \text{page\_size}$$
$$m = (\text{ramdisk\_size} + \text{page\_size} - 1) / \text{page\_size}$$
$$o = (\text{second\_size} + \text{page\_size} - 1) / \text{page\_size}$$

(\*) 주의 : 위의 내용은 qualcomm chip을 기준으로 작성한 것이므로, 다른 chip에 적용하기 위해서는 빨간 글씨 부분이 적절히 수정되어야 할 것임. 따라서 개념을 이해하는 용도로만 이해하시기 바람^^

(Example)

Page size: 2048 (0x00000800)

Kernel size: 4915808 (0x004b0260)

Ramdisk size: 310311 (0x0004bc27)

Second size: 0 (0x00000000)

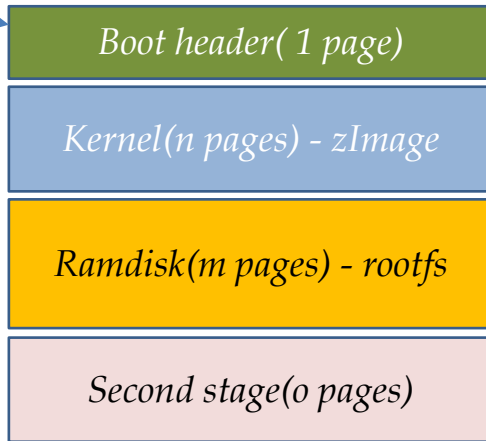
## 부록 2: *Android boot.img* 분해 및 *kernel* 교체 방법(3)

[TIP] *split\_bootimg.pl* 은 인터넷에서 구할 수 있음.

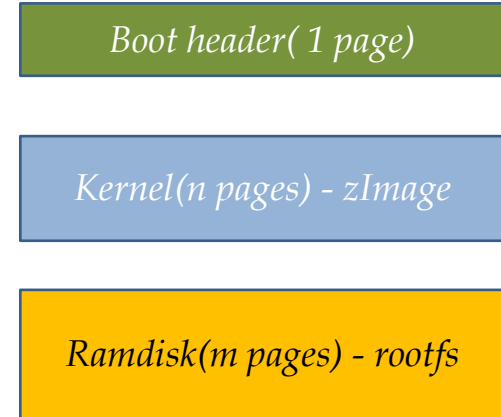
[TIP] 앞서 설명한 *mkbootimg* 와 위에서 설명한 내용을 토대로, 새로 build 한 *kernel image* 를 적용한 새로운 *boot.img* 를 만들 수 있게 됨.

[TIP] *fastboot* 명령을 사용하여 새로 만든 *boot.img* 를 flash 에 write 하면 됨 ^^.

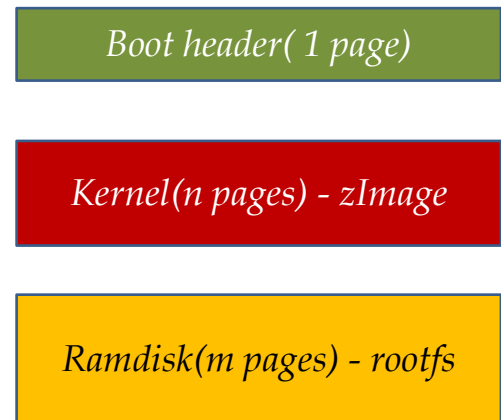
**ANDROID!**



*split\_bootimg.pl*  
(*boot.img* 분해)

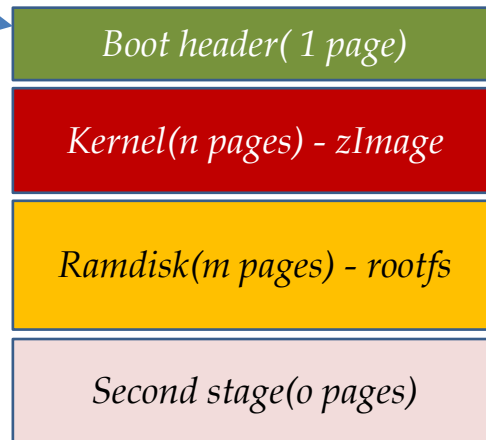


새로 build한 zImage로 교체



*mkbootimg*  
(*boot.img* 재생성)

**ANDROID!**



*fastboot*



## 부록 2: Android boot.img 분해 및 kernel 교체 방법(4)

- **<boot.img 파일 분해하기>**

- # ./split\_bootimg.pl boot.img
- -> boot header
- -> boot.img-kernel            ← kernel
- -> boot.img-ramdisk.gz        ← ramdisk root file system

- **<boot.img 파일 재생성하기>**

- # mkbootimg --cmdline "console=ttyHS1 pmem\_kernel\_ebi1\_size=0x200000 androidboot.hardware=qcom"
- --kernel zImage --ramdisk ramdisk.gz --base 0x20000000 -o boot.img
- ➔ 앞서 설명한 것 처럼, 빨간색 표시 부분은 시스템마다 다르니, 주의 요망(잘못하면, 영영 부팅 안됨^^)

- **<fastboot으로 boot.img write 하기>**

- # adb reboot-bootloader           ← fastboot mode로 전환(혹은 시스템에서 정의한 key 조합 선택하여)
- # fastboot flash boot ./boot.img

- -----

- **<기타 참고 사항1: ramdisk rootfs 파일 해부하기>**

- # gzip -d boot.img-ramdisk.gz
- # cpio -i < boot.img-ramdisk  
    ← 현재 디렉토리에 ramdisk file system을 구성하는 파일이 풀리게 됨.

- **<기타 참고 사항2: 새로운 ramdisk file 만들기 - init.rc, init.qcom.rc 등을 수정 후 테스트 사>**

- # find . | cpio -o -H newc | gzip > ../newramdisk.cpio.gz  
    <= 위에서 cpio로 파일을 풀어둔 디렉토리에서 명령을 실행.

## TODO

- 1) Android Specific Driver를 일반화하여 정리하기
  - ➔ *PMIC, charger, SDIO, USB, Video/Audio, Camera, T-DMB, WiFi, Bluetooth, Telephony, GPS, Sensors/Input ...*

*Thanks a lot !*



## References

- 1) *Linux Kernel Development* ..... [Robert Love]
- 2) *Writing Linux Device Drivers* ..... [Jerry Cooperstein]
- 3) *Essential Linux Device Drivers* ..... [Sreekrishnan Venkateswaran]
- 4) *Linux kernel 2.6 구조와 원리* ..... [이영희 역, 한빛미디어]
- 5) *Linux Kernel architecture for device drivers* .....  
[Thomas Petazzoni Free Electronics([thomas.petazzoni@free-electronics.com](mailto:thomas.petazzoni@free-electronics.com))]
- 6) *Some Internet Articles* ...