모듈과 디바이스 드라이버

- Chapter 08 -

Contents

Module Programming

- 커널 모듈 프로그래밍의 필요성
  - 리눅스 커널
    - 모노리틱 커널(마이크로 커널 방식이 아님)
  - Monolithic Kernel
    - 모든 기능이 커널 내부에 구현
    - 사소한 커널 변경에도 커널 컴파일과 리부팅 필요
    - 커널의 크기가 너무 큼
    - 일부 사용되지 않는 기능이 메모리 상주
    - 예 : UNIX, SVR4, Solaris, Linux
  - Micro Kernel
    - 운영체제의 핵심적인 기능만을 kernel에 넣고, 나머지는 서비스 프로세스 형태로 사용자 애플리케이션처럼 동작시키는 구조
    - 예)VxWorks, pSOS, VRTX, QNX, SROS

Module Programming

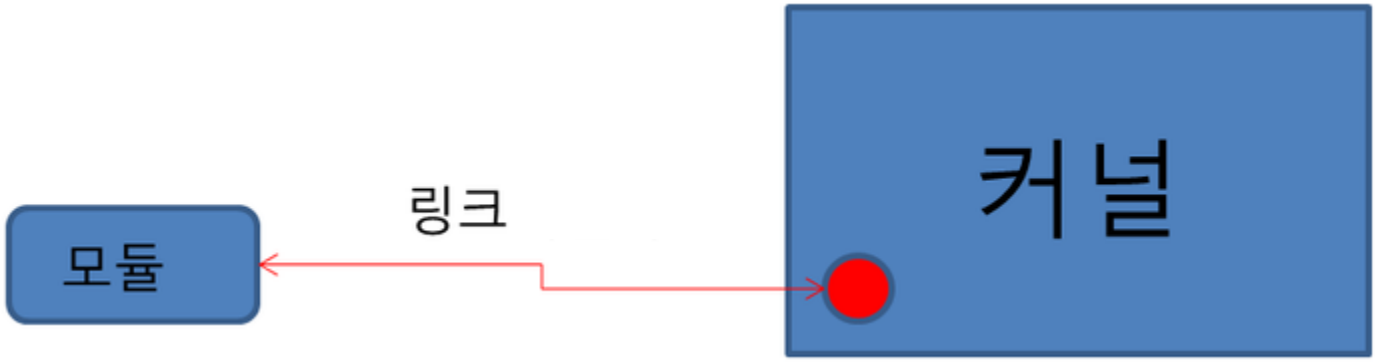
- 커널 모듈(Kernel Module)
  - 모듈은 리눅스 시스템이 부팅 된 후에 동적으로 load 및 unload 할 수 있는 커널의 구성 요소
    - 이러한 특징으로 커널을 다시 컴파일하거나 시스템을 재부팅하지 않고 커널의 일부분을 교체 할 수 있음
    - 파일 시스템, 디바이스 드라이버 등도 모듈로서 커널에 포함
  - 커널의 특징으로 의존성이 존재
    - 커널을 구성하는 모든 모듈에는 컴파일 한 커널 버전 정보가 포함
      - 이는 현재 실행되고 있는 커널 버전과 같아야 한다는 의미이며 다르면 에러 발생
      - 모듈의 버전 정보는 리눅스 커널 소스에 정의되어 있음
        - /include/linux/module.h

Module Programming

- 커널 모듈 프로그래밍의 역할
  - 모듈 프로그래밍으로 모노리틱 커널의 단점을 해결
  - 커널의 일부 기능을 빼고 모듈로 구현
  - 필요할 때만 적재시키므로 효과적인 메모리 사용
  - 커널 영역에서 프로그래밍 방법 제공
- 커널 모듈의 특징
  - 실행중인 커널에 동적으로 적재되거나 제거
  - 이벤트 처리(Event handling) 형태의 프로그램 방식(사건 구동형)
  - Main() 함수가 없다
  - Startup, cleanup 함수 존재

Module Programming

- 모듈 프로그램의 장점
  - 효과적인 메모리 사용
  - 커널 전체를 다시 컴파일하지 않고 커널의 일부분 또는 디바이스 드라이버를 교체
    - 시스템을 재부팅 하지 않고, 일부를 교체 가능
    - 모듈을 동적으로 링크하여 커널에 연결
    - 확장성, 재사용성이 높고 효율적

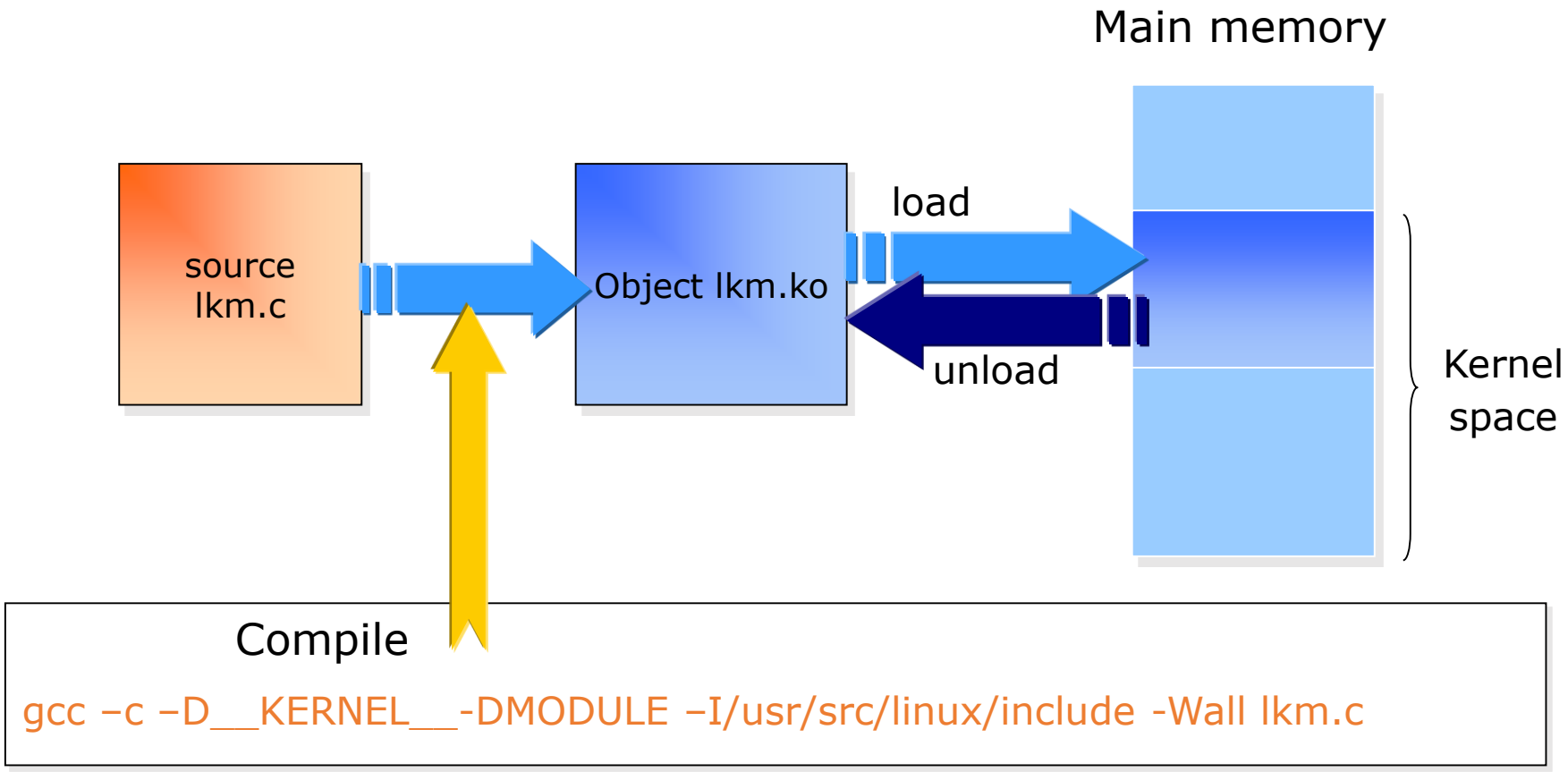


Module Programming

- 모듈 프로그램의 단점
  - namespace pollution
  - 모듈을 개발하다 보면 global 변수 혹은 function 명과 구별하기 어려움
  - 기존 global 변수 혹은 function 명이 충돌 가능

Module Programming

- Linux Kernel Module (LKM):
  - 동적으로 로딩, 언로딩



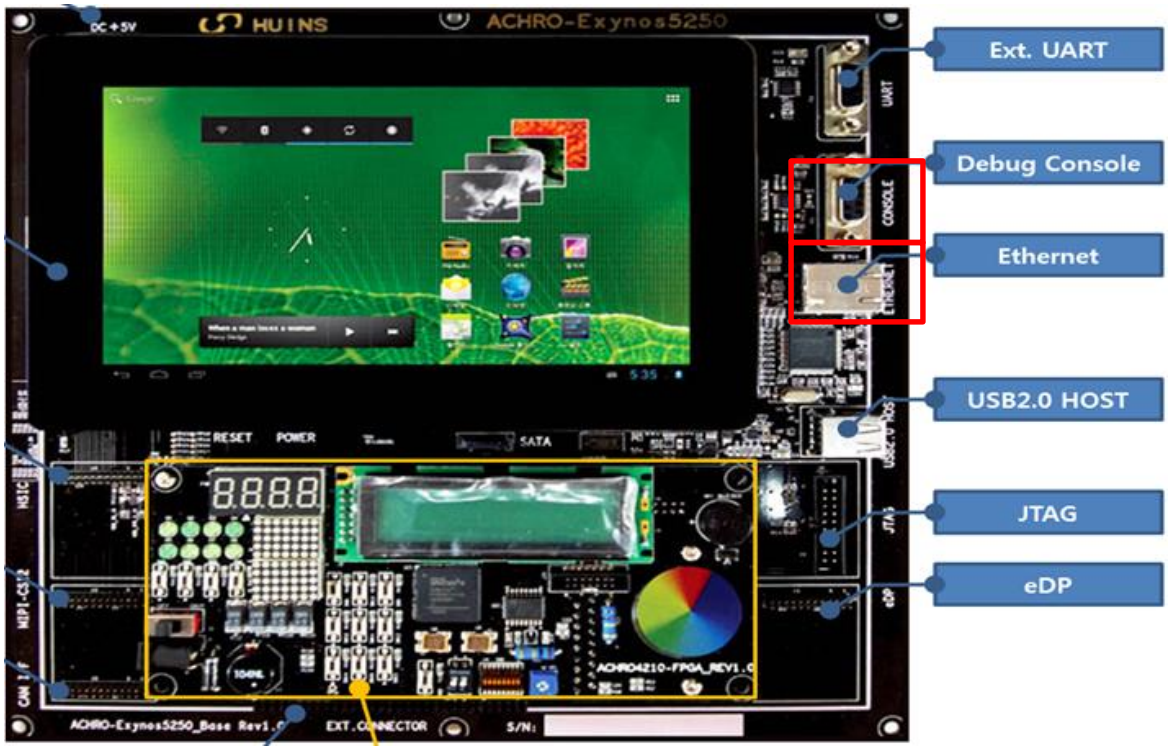
Module Programming

▶ 모듈 프로그래밍 명령어 요약

이 름	용 도
insmod	loadable kernel module을 설치(install)
rmmod	실행중인 loadable modules을 제거(unload)
lsmod	loaded module들의 정보를 표시
depmod	Module들의 symbol들을 이용하여 Makefile과 유사한 dependency file을 생성
modprobe	depmod명령으로 생성된 dependency를 이용하여 지정된 디렉토리의 module들과 연관된 module들을 자동으로 load.
modinfo	목적화일을 검사해서 관련된 정보를 표시

타겟보드 NFS 설치

- 리눅스 실습 환경
  - USB-Serial + minicom 으로 진행상황 및 제어관련 정보 활용
  - 호스트 PC에서 타겟 보드로 NFS를 통해서 이미지를 다운로드
  - 이더넷 IP 설정이 필요



타겟보드 NFS 설치

- 타겟보드 IP 설정
  - 타겟보드에 Ethernet 케이블과 USB-Serial 케이블을 연결하고 전원 인가 ➔ root 로그인
  - 하기 내용은 호스트 리눅스에서 별도의 타겟용 minicom 창을 열어서 실시
  - 네트워크 관련 정의 파일 수정

```
#vi /etc/network/interfaces
```

```
# Configure Loopback
auto lo
iface lo inet loopback

# User Network setting
auto eth0
iface eth0 inet static
        address 10.40.1.(PC의 IP 번호 + 100)
        netmask 255.255.255.0
        broadcast 10.40.1.255
        gateway 10.40.1.254
```

  - 재실행

```
#reboot
```

타겟보드 NFS 설치

- 타겟보드 IP와 호스트(리눅스)와의 NFS 설정
  - 향후 모든 실습의 이미지 다운로드는 NFS를 통해서 실시
  - 따라서 실습 전에 마운트 상태의 확인이 필요
  - 하기 내용은 호스트 리눅스에서 별도의 타겟용 mnicom 창을 열어서 실시(USB-Serial 케이블)

```
#mount -t nfs 10.40.1.(리눅스 호스트번호):/nfsroot /mnt/nfs -o tcp,nolock
```

    - 리눅스 호스트 번호는 본인의 PC IP 번호 + 50
  - NFS 설정 확인

```
#ls /mnt/nfs
```

    - [… test1 …] 파일이 있는지 확인 ➔ 툴체인 설치 & 로컬 NFS 테스트시 작성된 파일

타겟보드 NFS 설치

- 타겟보드와 NFS 설정이 안되는 경우 (실습실 네트워트 상태에 따라 발생할 수도 있음)
  - 모든 NFS를 통한 데이터의 이동이 안된다.
  - 이럴 경우 다음과 같이 USB 리더를 이용해 NFS 없이 실습이 가능
    - Micro SD 카드를 PC에 연결
    - 우분투에서 SD 파일이 mount 되었는지 확인
      - #ls -l /media/Achro5250_System
    - 우분투에서 복사할 파일을 cp 명령으로 SD카드로 이동
      - 현재의 우분투 디렉토리에서 SD 카드의 root 디렉토리로 복사하는 경우의 예
      - cp <복사할 파일> /media/Achro5250_System/root
    - Micro SD 카드를 타겟에 연결하고 root에서 작업 시작

모듈 프로그래밍 실습

▶ 간단한 모듈프로그램 예 - 소스 프로그램 : hello.c

```
/* This program is module example */

#include <linux/kernel.h>                    /* 커널에서 수행될 때 필요한 헤더 파일 */
#include <linux/module.h>                    /* 모듈에 필요한 헤더 파일 */
#include <linux/init.h>                      /* module_init(), module_exit() 매크로 정의 */

MODULE_LICENSE("GPL");                       /* 모듈에 대한 라이선스를 명시 (GPL, GPL2...) */
MODULE_AUTHOR("Huins");                      /* 모듈 저작권자 */

static int module_begin(void) {
      printk(KERN_ALERT "Hello module, Achro 5250!!₩n");
      return 0;
}
static void module_end(void) {
      printk(KERN_ALERT "Goodbye module, Achro 5250!!₩n");
}
module_init(module_begin); /* 모듈이 설치될 때 초기화 수행, insmod 할 때 수행되는 함수 */
module_exit(module_end); /* 모듈이 제거될 때 반환 작업 수행, rmmod 할 때 수행되는 함수*/
```

모듈 프로그래밍 실습

- 간단한 모듈프로그램 예 - 소스 프로그램 동작 설명
  - printk()
    - Linux kernel에서 정의됨
    - Printf와 비슷하게 동작
    - printk는 커널 모드에서만 호출 가능, printf는 유저 모드에서만 호출 가능
    - printf는 내부적으로 버퍼에 출력할 것을 모아두었다가 방출시키는 구조(버퍼에 담은 내용을 방출하기도 전에 OS가 다운되어 출력 못할 가능성), printk는 버퍼 없이 바로 출력하는 구조
  - module_init() (init_module: Linux 2.4 version)
    - 모듈을 메모리 공간에 적재
    - 모듈이 사용하는 커널 내부 심볼 연결
  - module_exit() (cleanup_module: Linux 2.4 version)
    - 모듈을 메모리에서 삭제

모듈 프로그래밍 실습

- 간단한 모듈프로그램 예 - Makefile
  - obj-m : 만들어진 커널 오브젝트 이름
  - KDIR : 커널 소스 디렉터리
  - PWD : 현재 디렉터리

```
obj-m := hello_module.o
KDIR := /work/achro5250/kernel
PWD := $(shell pwd)

all: driver
driver:
	$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
clean:
	rm -rf *.ko
	rm -rf *.mod.*
	rm -rf *.o
```

- 간단한 모듈프로그램 예 - 컴파일

```
# make
```

모듈 프로그래밍 실습

- 모듈프로그램 실습 절차
  - 모듈 프로그래밍 예제 파일 복사 (리눅스 호스트)
    - # cd /mnt/hgfs/[CD FILE]/examples/linux/module
    - # cp -a module.tar.gz /work/achro5250
    - # cd /work/achro5250
    - # tar xvfz module.tar.gz
    - # cd module
    - # vi Makefile

Makefile
… KDIR := /work/achro5250/kernel …

    - # make
    - # cp hello_module.ko /nfsroot

모듈 프로그래밍 실습

모듈프로그램 실습 - 계속

타겟 보드에서 NFS 연결 (타겟 보드에서 실행)

▷NFS 연결 전, ping 을 통하여 네트워크가 연결 됐는지를 먼저 확인할 것

ping 10.40.1.(리눅스 호스트 번호)

mkdir /mnt/nfs (디렉토리가 없을 경우 생성)

mount -t nfs 10.40.1.(리눅스 호스트번호):/nfsroot /mnt/nfs -o tcp,nolock

cp /mnt/nfs/hello_module.ko /root

cd root

insmod hello_module.ko

[Hello, Welcome to module!!]

▷dmesg 혹은 lsmod로 모듈이 정상적으로 로드 되었는지 확인

rmmod hello_module

[GoodBye, Exit Module!!]

리눅스 디바이스 드라이버

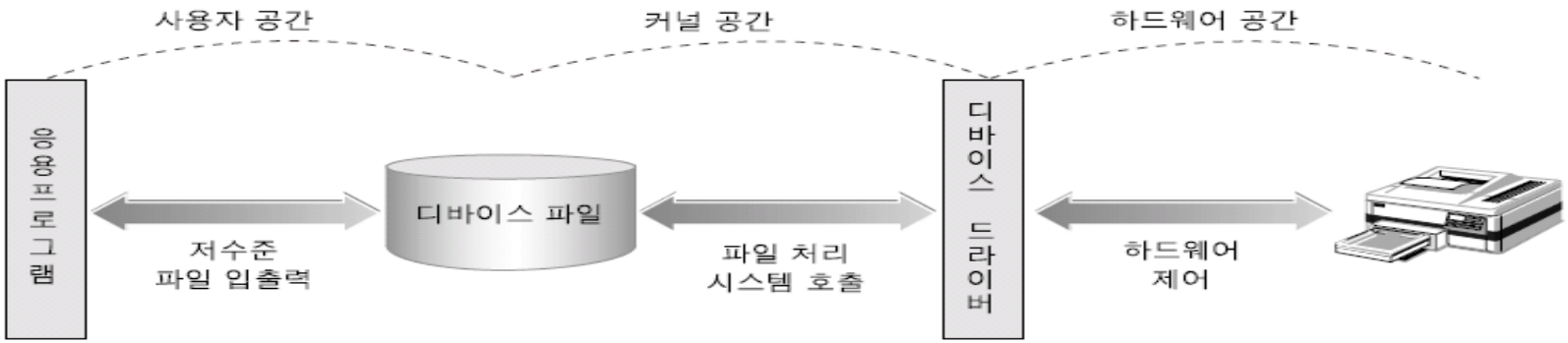
- 디바이스(Device)
  - 임베디스 시스템에서 디바이스는 LCD, USB, Ethernet, PCMCIA, AUDIO 등과 같이 시스템의 주변 장치를 지칭
  - 시스템 내부의 Flash Memory, SDRAM등은 제외
- 드라이버(Driver)
  - 디바이스와 같은 하드웨어 장치를 제어하고 관리하는 프로그램
    - 디바이스를 구동시키기 위하여 반드시 필요
  - 임베디드 시스템에서는 디바이스 드라이버라고 부르고 커널에서 지원하지 않는 경우 해당 드라이버를 찾아서 설치하거나 개발해야 함

리눅스 디바이스 드라이버

- 디바이스 드라이버(Device Driver) 개요
  - 임베디드 OS의 목적은 임베디드 장비의 구동
    - 임베디드 장비의 구동은 주변의 여러 장치가 정상적으로 동작하는 것
    - 이러한 주변 장치 제어를 책임지는 부분이 운영체제의 디바이스 드라이버
  - 주번호와 부번호를 이용하여 각각의 디바이스들을 구분하여 사용
    - 주번호: 커널에서 디바이스 드라이버를 구분하는데 사용
    - 부번호: 특정 디바이스 드라이버 내에서 필요한 경우 장치를 구분하기 위해 사용

리눅스 디바이스 드라이버

- 디바이스 드라이버(Device Driver) 개요
  - 물리적인 하드웨어 장치(디바이스)를 다루고 관리하는 소프트웨어 ➔ 커널의 일부로 취급
  - 응용프로그램이 H/W를 제어할 수 있도록 인터페이스를 제공해 주는 코드
    - 특성이 다른 디바이스라도 표준적으로 동일한 서비스 제공이 목적
  - 주변 장치들은 시스템 자원이며 사용자 태스크에서 사용하려면 시스템 콜을 사용
    - 표준 system call을 이용하여 디바이스를 제어



[그림 10-3] 디바이스 드라이버와 디바이스 파일

리눅스 디바이스 드라이버

- 디바이스 드라이버 개발 프로세스

디바이스 드라이버를 커널에 올림

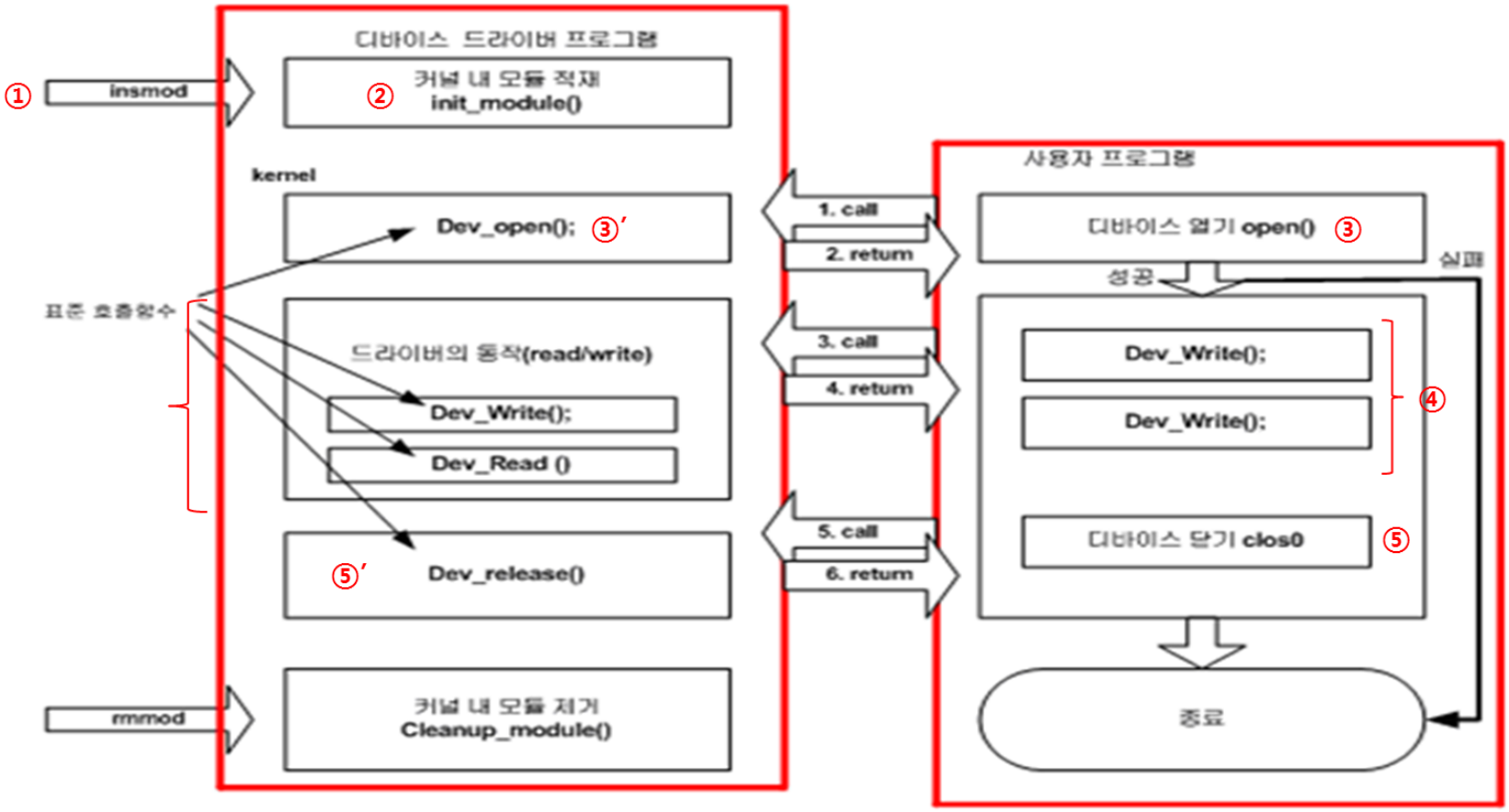
디바이스와 응용프로그램을 묶어주는 스페셜 파일(노드 파일) 생성

디바이스 드라이버를 커널에서 제거

- 리눅스 소스의 drivers 디렉터리에는 다양한 디바이스 드라이버를 이미 지원하기 때문에 실제로 특별히 새롭게 디바이스 드라이버를 개발하는 경우는 거의 없음

리눅스 디바이스 드라이버

- 디바이스 드라이버와 어플리케이션 간의 통신



리눅스 디바이스 드라이버

- 디바이스 드라이버와 어플리케이션 간의 통신 - 설명

① 디바이스 드라이버 로드 (insmod):

- 리눅스 시스템에서 모듈로 된 디바이스 드라이버를 insmod 명령을 통해서 커널에 등록

② 드라이버 내의 init_module()이 동작 하여 기본적인 하드웨어 부분을 초기화

③ 사용자 프로그램에서 open()을 호출하면 해당 디바이스 드라이버 내의 dev_open()이 구동

- 0~2를 제외한 양의 정수 값이 나오면 정상적으로 연결된 것이며, -1이 나오면 장치 연결실패

④ 사용자 프로그램에서 read(), write(), ioctl() 등을 호출하면, 매칭되는 시스템 콜이 발생

⑤ 디바이스 드라이버에서 호출된 함수에 맞는 구현 부분 실행 (File Operations Struct)

- read 및 write()의 리턴 값은 ssize_t

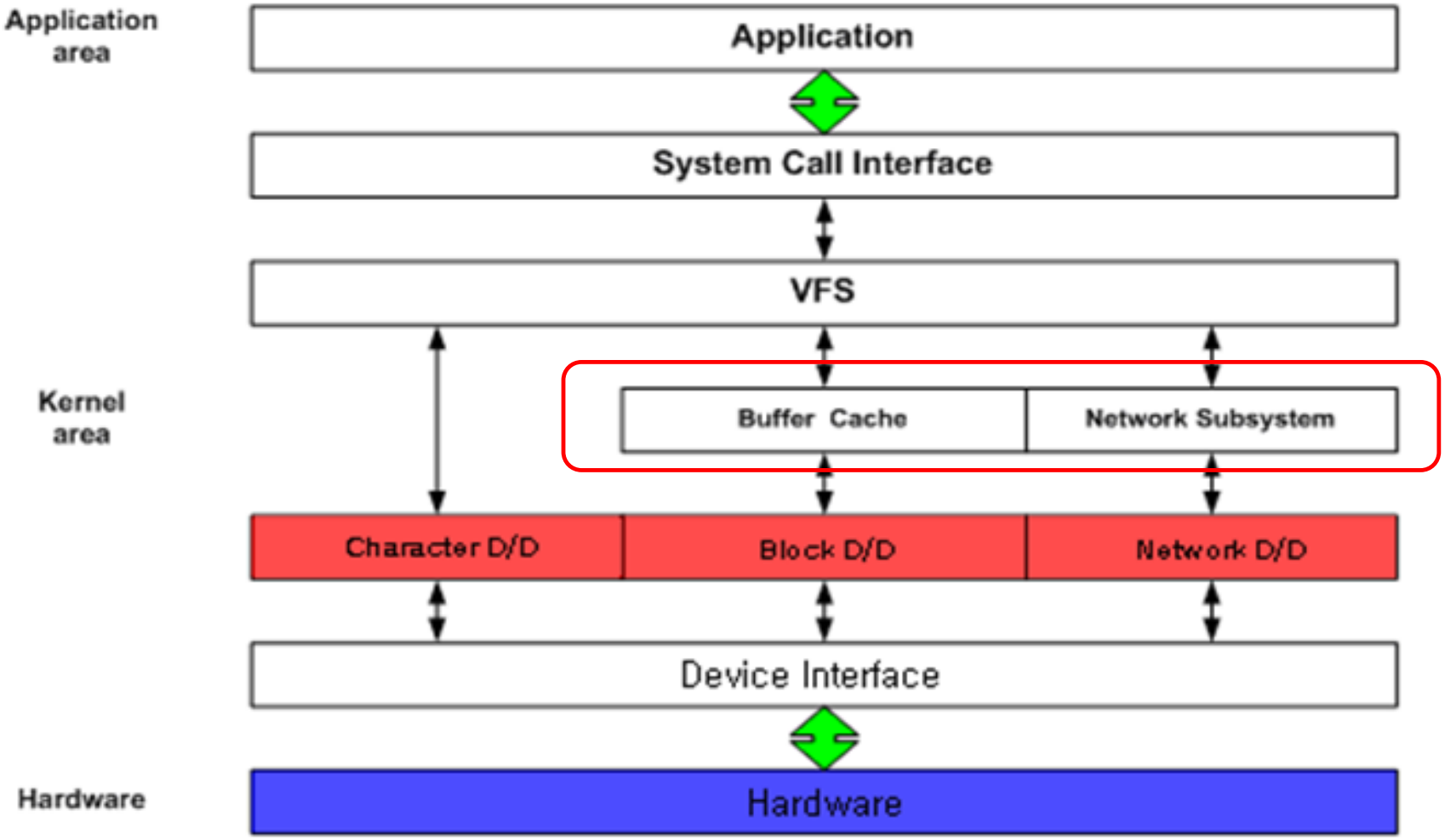
⑥ close()가 호출되면, 시스템 콜이 발생하여 디바이스 드라이버 내의 dev_release()가 구동

리눅스 디바이스 드라이버

- 디바이스 드라이버의 종류
  - 문자 디바이스 (Character device)
    - 파일시스템에서 하나의 노드 형태로 존재
    - 데이터를 문자 단위 또는 연속적 바이트 흐름으로 입출력 ➔ 순차성을 지닌 하드웨어
      - 터미널, 콘솔, 키보드, 사운드카드, 스캐너, 프린터, 직렬/병렬 포트, 마우스, 조이스틱 등
  - 블록 디바이스 (Block device)
    - 파일시스템에서 하나의 노드 형태로 존재
    - 버퍼 캐시를 통해 임의 접근이 가능하며 데이터를 블록 단위로 입출력 ➔ 버퍼(효율성)
      - 하드디스크, 플로피 디스크, 램디스크, 테이프, CD-ROM, DVD 등
  - 네트워크 디바이스 (Network device)
    - 파일시스템의 노드 형태가 아닌 특별한 인터페이스를 사용
    - 네트워크 통신을 통해 네트워크 패킷을 송수신할 수 있는 디바이스 ➔ 패킷 단위로 접근
      - 이더넷(eth0), PPP, SLIP, ATM, 네트워크 인터페이스 카드

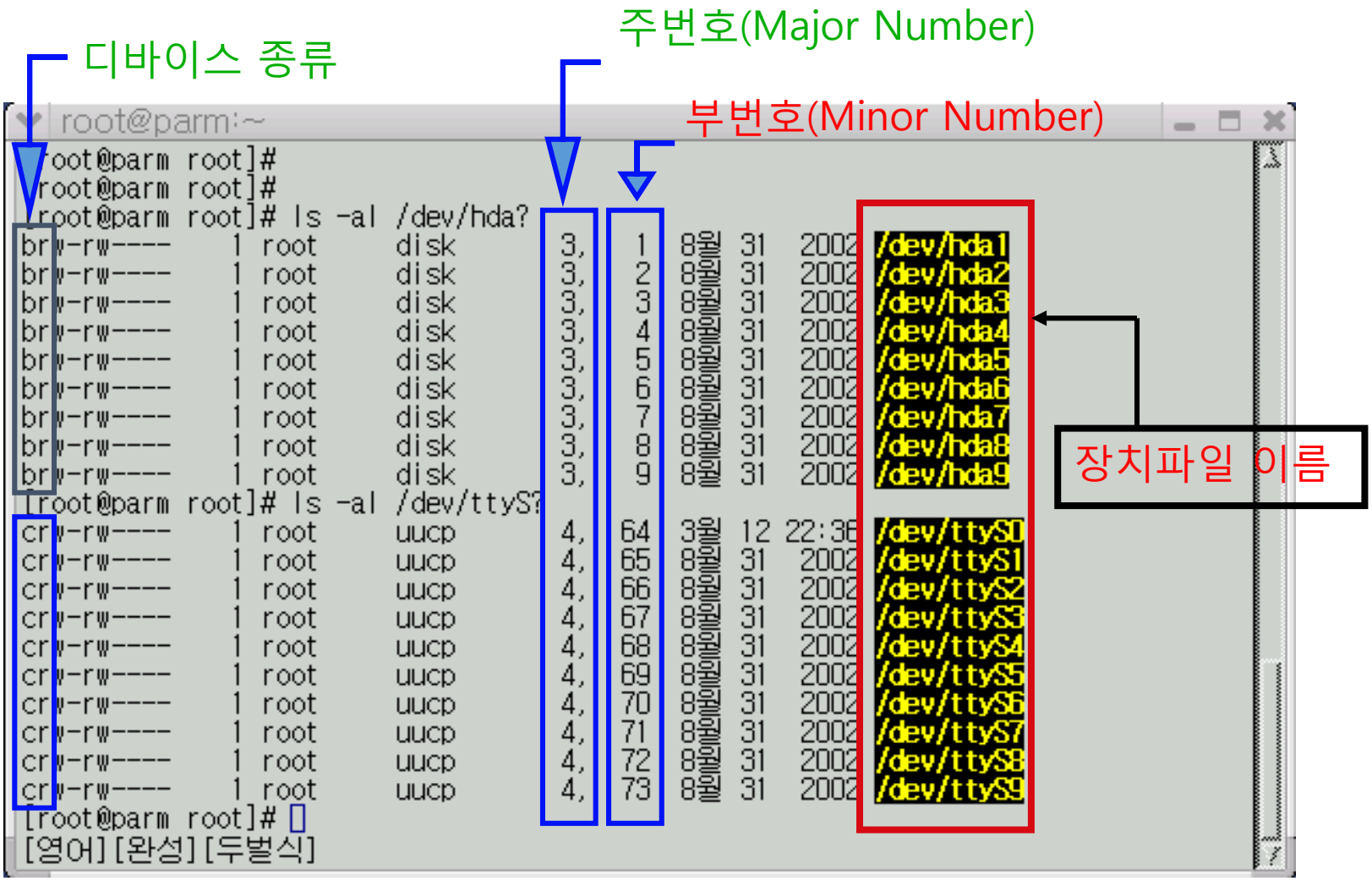
리눅스 디바이스 드라이버

- Linux Kernel Layout



리눅스 디바이스 드라이버

▶ 디바이스 장치 파일 예제



리눅스 디바이스 드라이버

- 일반적인 디바이스 드라이버 내부 구성
  - 초기화 인터페이스 : init() 함수, init_module() 함수
  - 파일 시스템 인터페이스 부분
    - 잘 정의된 인터페이스 : 문자와 블록의 경우 file_operations 이용
    - 문자 드라이버 : open, release, read, write, ioctl
    - 블록 드라이버 : open, release, request, ioctl
    - 네트워크 드라이버 : open, close, transmit, receive, ioctl
  - 하드웨어 인터페이스
    - in(), out()

문자 디바이스 드라이버

- 문자 디바이스 드라이버
  - 문자(char) 디바이스는 (파일처럼) 바이트 스트림 형태로 참조
  - 최소한 open, close, read, write 시스템 호출을 구현
  - 일반적으로 순차적 접근만 허용
  - /dev/ttyS0, /dev/lp1과 같은 파일 시스템 노드로 접근

```
[root@uclibc /dev]#
[root@uclibc /dev]#
[root@uclibc /dev]# ll ttyS*
crw-rw-rw-    1 default  default     4,  64 Jan  1  1970 ttyS0
crw-rw-rw-    1 default  default     4,  65 Jan  1  1970 ttyS1
crw-rw-rw-    1 default  default     4,  66 Jan  1  1970 ttyS2
crw-rw-rw-    1 default  default     4,  67 Jan  1  1970 ttyS3
crw-------    1 root     root      204,  64 Dec 23 09:46 ttySAC0
crw-rw-rw-    1 default  default   204,  65 Jan  1  1970 ttySAC1
crw-rw-rw-    1 default  default   204,  66 Jan  1  1970 ttySAC2
crw-rw-rw-    1 default  default   204,  67 Jan  1  1970 ttySAC3
[root@uclibc /dev]#
```

  - 위의 그림에서 /dev/ttySAC0는 다음과 같은 의미
    - c는 Character Device를 의미하고, 주 번호는 204번, 부 번호는 64, 디바이스 이름은 ttySAC0

문자 디바이스 드라이버

디바이스 드라이버의 커널 등록

- insmod 실행하면 디바이스 드라이버 내부 프로그램인 init_module 함수에서 실행
- Extern int register_chrdev(unsigned int, const char *, struct file_operations *)

  ○ 첫 번째 인수 : Major 번호로 만약 '0'값을 주면 Major 번호 중 사용하지 않는 번호로 동적 할당
  ○ 두 번째 인수 : 디바이스 name 즉 장치 이름이며, /proc/devices에 나타난다.
  ○ 세 번째 인수 : 파일연산 함수 (다음에 자세히 설명할 것이다.)
  ○ 리턴하는 값 : 0이나 양수이면 정상, 음수이면 실패.

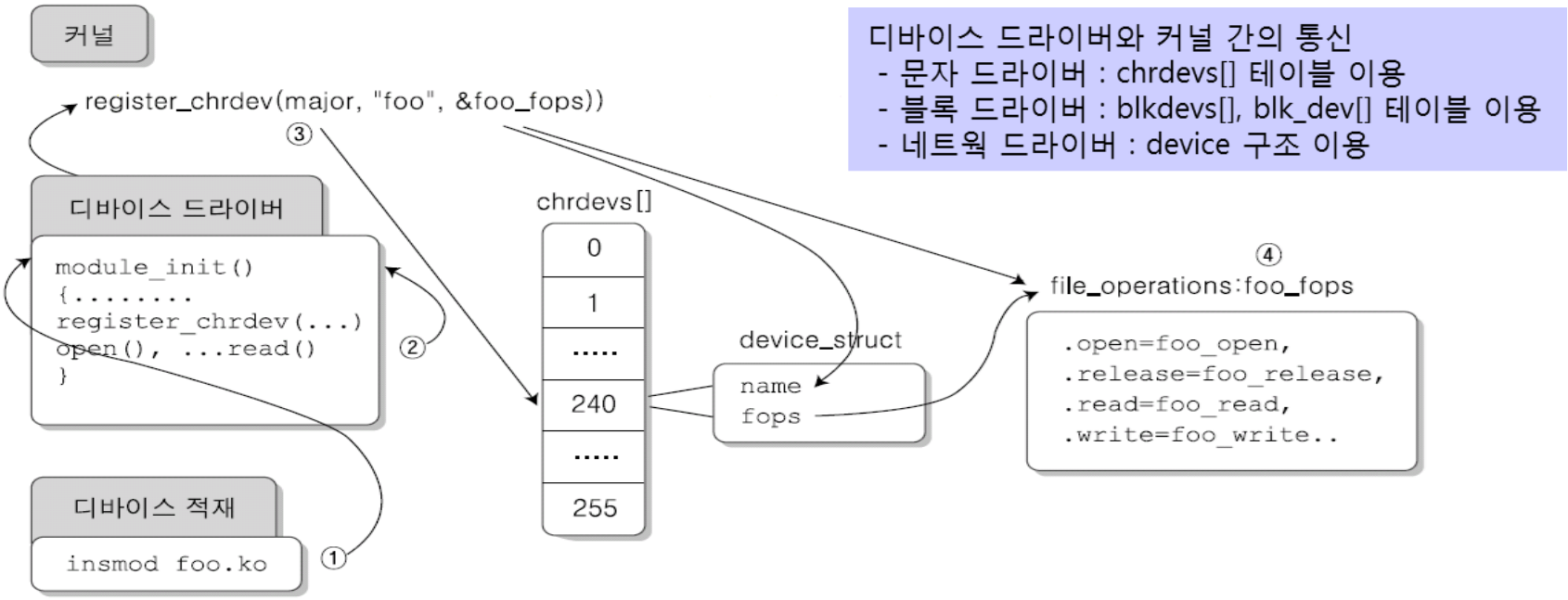
디바이스 파일 해제

- rmmod 실행하면 디바이스 드라이버 내부 프로그램인 cleanup_module 함수에서 실행
- extern int unregister_chrdev(unsigned int, const char *)

  ○ 첫 번째 인자 : 해제하고자 하는 장치 주 번호(Major number)
  ○ 두 번째 인자 : 해제하고자 하는 장치 이름(Device name)

문자 디바이스 드라이버

- 디바이스 드라이버의 커널 등록 과정
  - 쉘모드에서 insmod 실행
  - insmod 명령은 디바이스 드라이버 파일 내의 module_init()을 호출
  - module_init() 내에서는 chrdsvs[] 구조체에 등록을 하기 위해 resister_chrdev() 함수를 호출
  - resister_chrdev() 함수는 주번호, 파일 이름, file_operation 구조체를 포함해 등록을 실행
    - 주번호를 0으로 할 경우 커널에서 자동으로 지정

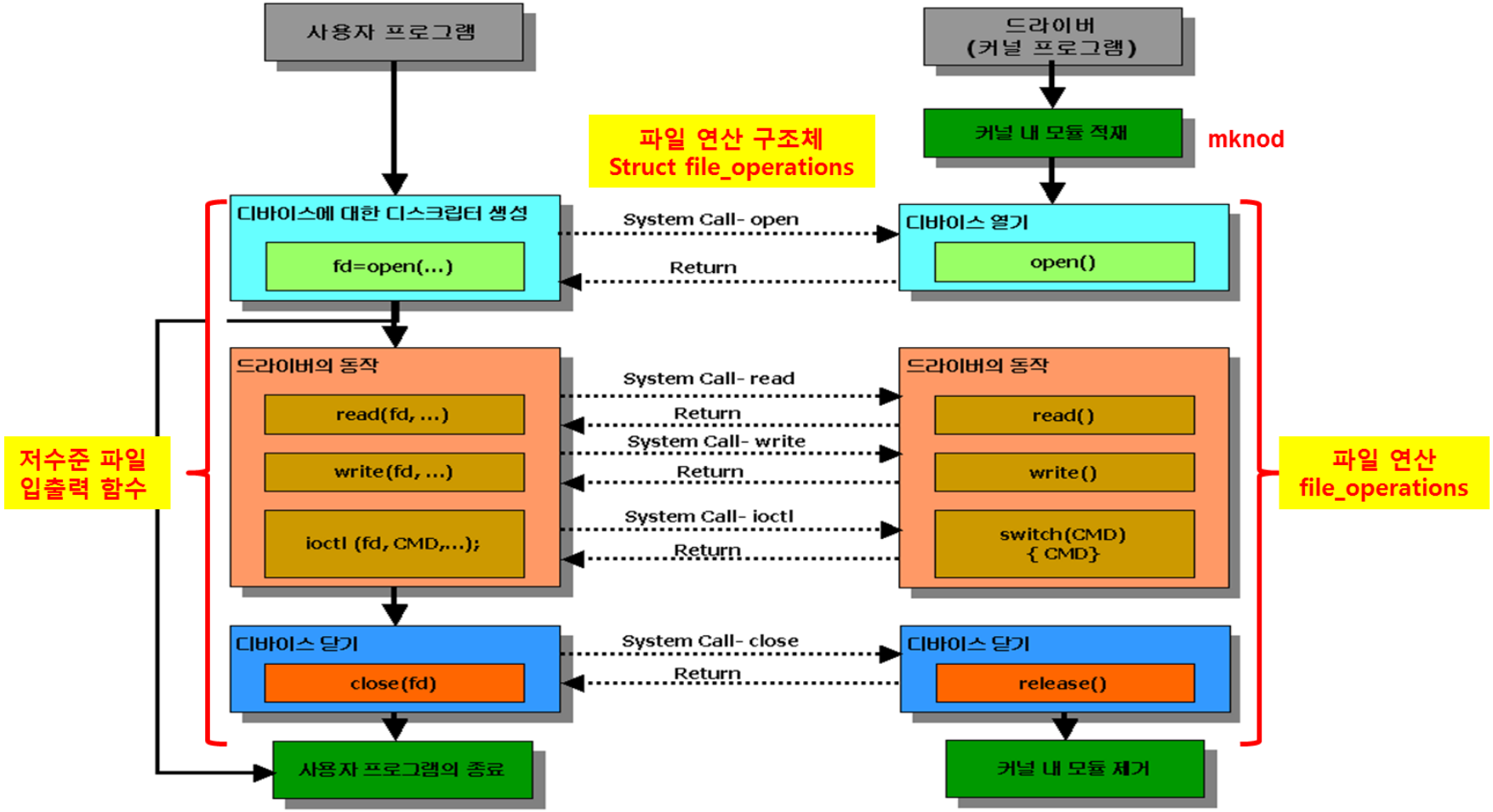


문자 디바이스 드라이버

- 디바이스 파일의 생성 (mknod)
  - 디바이스 드라이버에 접근할 수 있는 장치 파일(디바이스 파일)의 생성
  - 리눅스는 모든 자원을 파일 형식으로 표현하며 이런 파일들은 /dev/ 디렉터리에 존재
  - 리눅스에서 동작하는 응용 프로그램은 하드웨어를 다루기 위해 해당 디바이스 파일에 저수준 파일 입출력 함수를 이용한다
  - 생성 방법
    - mknod [device file name] [타입] [주번호] [부번호]
      - Type : c= 문자 디바이스 드라이버, b - 블록 디바이스 드라이버
    - 예): mknod /dev/fpga_led c 260 0
  - 디바이스 파일의 생성(mknod)과 디바이스 드라이버의 커널 등록(insmod)의 순서는 무관

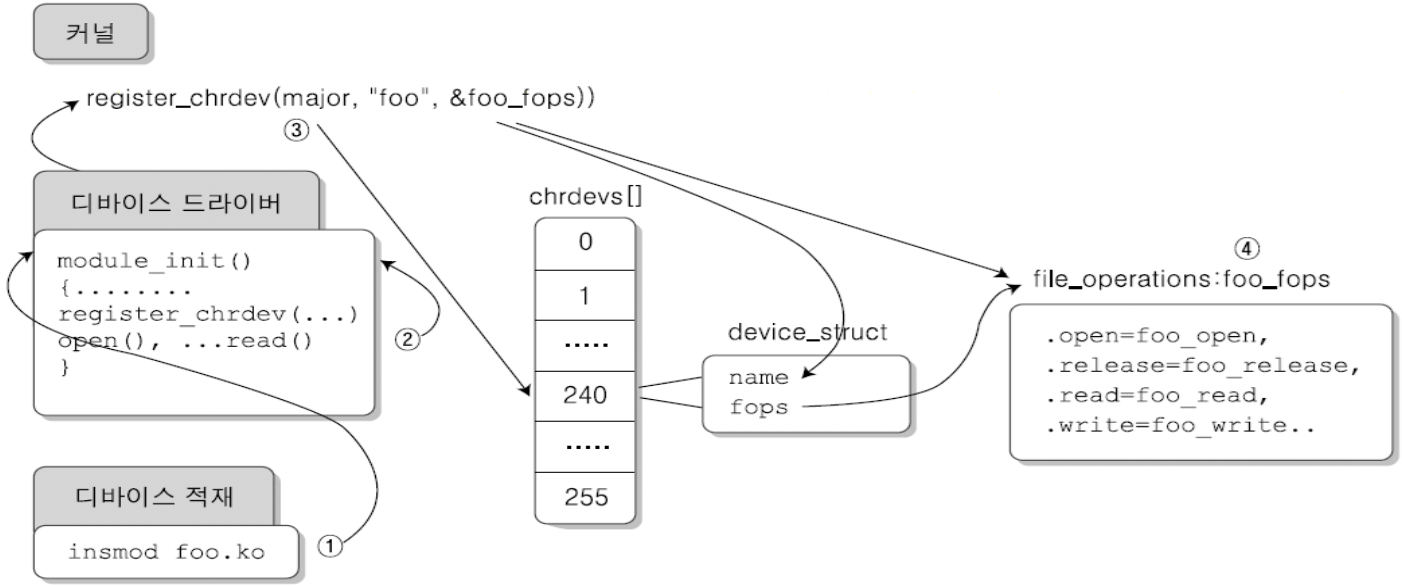
문자 디바이스 드라이버

▶ 디바이스 드라이버 동작 과정



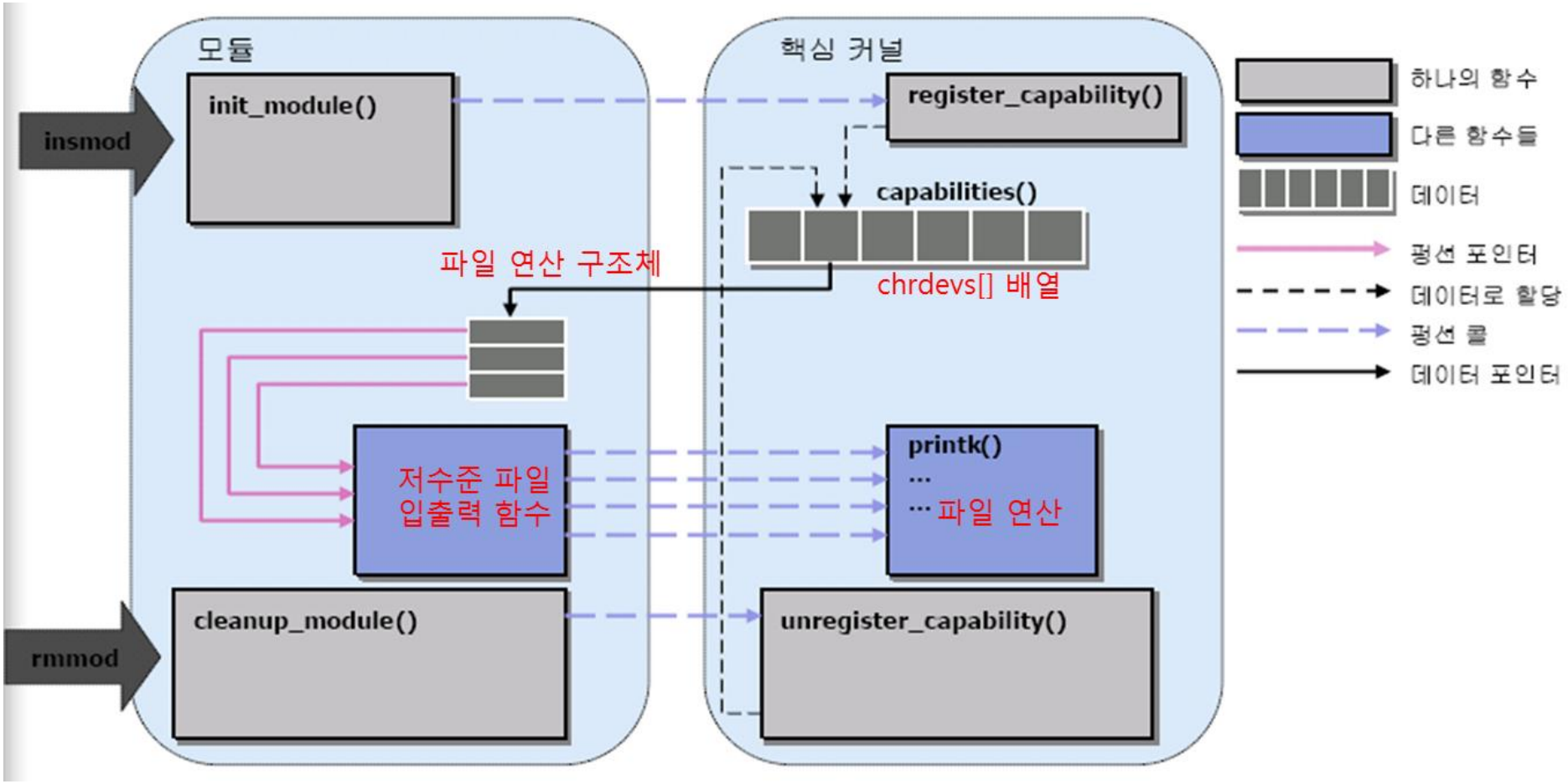
문자 디바이스 드라이버

- 디바이스 드라이버 동작 과정
  - 응용 프로그램에서 open() 함수로 디바이스 파일을 열어 타입정보와 주 번호를 얻는다
  - 이 정보를 이용해 chrdevs[] 배열내에 등록된 디바이스 드라이버 인덱스를 얻는다
  - 이 인덱스를 이용해 chrdsvs[] 변수에 등록된 파일 연산 구조체(struct file_operations)의 주소를 얻는다 ➔ 이 구조체에는 저수준 파일 입출력 함수를 설정한 내용이 있다
  - 저수준 파일 입출력 함수는 디바이스 드라이버의 함수와 1:1로 매핑이 되어 있으므로 응용 프로그램이 디바이스 파일에 저수준 입출력 함수를 적용하면 그에 해당하는 함수가 호출



문자 디바이스 드라이버

▶ 디바이스 드라이버 동작 과정

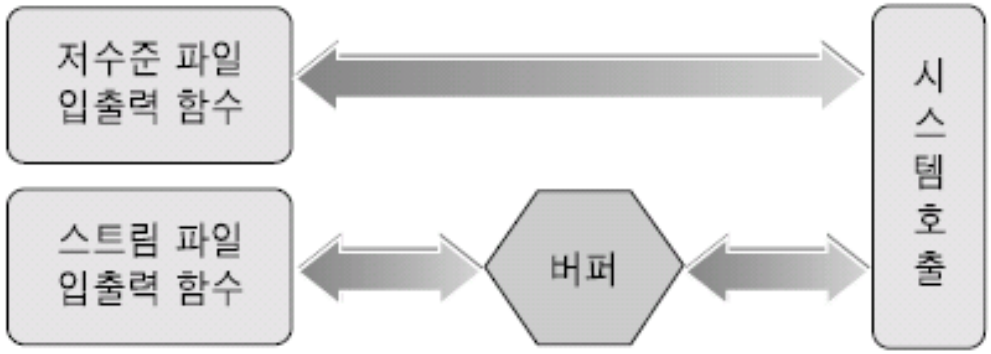


문자 디바이스 드라이버

- 저수준 파일 입출력 함수
  - 리눅스 커널에서 제공하는 파일 관련 시스템 콜을 라이브러리 함수로 만든 것
  - 응용프로그램은 단순화된 저수준 파일 입출력 함수를 이용하여 디바이스 파일에 연관된 디바이스 드라이버 함수가 동작하게 됨
  - 종류
    1. open(): 파일이나 장치를 연다.
    2. close(): 열린 파일을 닫는다.
    3. read(): 파일에서 데이터를 읽어온다.
    4. write(): 파일에 데이터를 쓴다.
    5. lseek(): 파일의 쓰거나 일기 위치를 변경한다.
    6. ioctl(): read(), write()로 다루지 않는 특수한 제어를 한다.
    7. fsync(): 파일에 쓴 데이터를 실제 하드웨어의 동기를 맞춘다.

문자 디바이스 드라이버

- 참고 : 파일 입출력 함수
  - 파일 입출력 함수의 종류
    - 저수준 파일 입출력 함수
      - 시스템에서 제공하는 파일 입출력 함수
    - 스트림 파일 입출력함수
      - 버퍼를 사용하여 스트림 처리가 가능 ➔ 디바이스 파일에 직접 접근 불가
      - 저수준 파일 입출력 함수를 사용하기 쉽게 만든 함수



	저수준 파일 입출력 함수	스트림 파일 입출력함수
접근 형태	직접 접근	형식화된 형태로 접근, 사용하기 쉬움
버퍼 사용 유무	사용 않음	버퍼를 사용
디바이스 파일 다루기	가능	곤란
디바이스 드라이버 구현	사용	사용 불가
종류	open(), read(), write(), close(), fsync(),ioctl(), poll(), mmap() 등	fopen(), fread(), fwrite(), fclose() 등

문자 디바이스 드라이버

- 파일 연산 구조체 (struct file_operations)
  - 문자 디바이스 드라이버와 응용 프로그램을 연결하는 고리
  - 응용 프로그램이 저수준 파일 입출력 함수를 사용하여 디바이스 파일에 접근하면, 커널은 등록된 문자 디바이스 드라이버의 오퍼레이션 구조체 정보를 참고하여 디바이스 파일에 접근하여 대응하는 함수를 호출한다.
  - 자주 사용하는 파일 오퍼레이션(파일 연산)
    1) open: 디바이스를 open
    2) read: 장치에서 데이터를 읽는다
    3) write: 장치에 데이터를 쓴다.
    4) ioctl:
       - 디바이스에 종속적인 함수나 커맨드를 만드는데 사용한다.
       - 각 디바이스 장치에 대한 고유한 커맨드를 프로그래머가 임의로 만들 수 있다.
    5) release: 디바이스 해제

문자 디바이스 드라이버

파일 연산 구조체 - 계속

file_operations의 구조체 원형 -<linux/fs.h>참조

```
struct file_operations {
 struct module *owner;
 loff_t (*llseek) (struct file *, loff_t, int);
 ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
 ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
 ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
 ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
 int (*readdir) (struct file *, void *, filldir_t);
 unsigned int (*poll) (struct file *, struct poll_table_struct *);
 int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
 int (*mmap) (struct file *, struct vm_area_struct *);
 int (*open) (struct inode *, struct file *);
 int (*flush) (struct file *);
 int (*release) (struct inode *, struct file *);
 int (*fsync) (struct file *, struct dentry *, int datasync);
 int (*aio_fsync) (struct kiocb *, int datasync);
 int (*fasync) (int, struct file *, int);
 int (*lock) (struct file *, int, struct file_lock *);
 ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
 ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
 ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void __user *);
 ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
 unsigned long (*get_unmapped_area) (struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
};
```

문자 디바이스 드라이버

- 파일 연산 구조체 - 계속
  - onwer: 어떤 모듈로 올라간 디바이스 드라이버와 연관을 가지는지를 나타내는 포인터
  - lseek: 파일에서 현재의 read/write 위치를 옮기며 새로운 위치가 양수 값으로 return된다. error는 음수가 return 된다.
  - read: 디바이스에서 데이터를 가져오기 위해 사용된다.
  - write: 디바이스에 데이터를 보낸다.
  - readdir : 디바이스 노드에서는 NULL이어야 하며 디렉토리에 대해서 사용한다.
  - poll: 현재 프로세스에 대기 큐를 넣는다 (2.0에서 select_wait과 같다)
  - ioctl: 디바이스에 종속적인 명령을 만들 수 있도록 한다.(플로피의 한 트랙을 포맷하는 명령)
  - mmap: 디바이스 메모리를 프로세스 메모리에 맵핑 시키도록 요청에 사용
  - open: 디바이스 노드에 대해 수행되는 첫 번째 동작,application의해 장치가 열릴 때 호출
  - flush: 열린 디바이스를 닫기 전에 모든 데이터를 쓰도록 하기 위해 사용
  - release: 노드를 닫을 때 수행된다
  - fsync: 디바이스에 대한 모든 연산의 결과를 지연하지 않고 즉시 일어나도록 한다.
  - fasync: FASYNC 플래그에 변화가 있는 디바이스를 확인하기 위해 사용한다.
  - check_media_change: 블록 디바이스에 사용되는데 플로피처럼 제거가능 미디어에 사용
  - revalidate: check_media_change와 관련 버퍼 캐쉬의 관리에 상관이 있다.
  - lock: 파일체계에서만 유효하며 사용자가 파일을 잠그게 하기 위해 사용 한다.

문자 디바이스 드라이버

- 디바이스 드라이버 내에서의 파일 연산 구조체 사용
  - 파일 연산 구조체 중 사용하지 않는 것은 NULL 값으로 초기화
  - 내부적으로 필요한 파일 연산만 만들고 이를 구조체에서 필요한 필드만 선언
  - 우리가 앞으로 실험할 LED 제어용 디바이스 드라이버의 소스 일부
    - Open, write, read, release의 4개의 파일 연산만 사용

```
// define file_operations structure
struct file_operations iom_led_fops = {
    .owner = THIS_MODULE,
    .open = iom_led_open,
    .write = iom_led_write,
    .read = iom_led_read,
    .release = iom_led_release,
};
```

문자 디바이스 드라이버

- 디바이스 드라이버 내에서의 파일 연산 구조체 사용 - 계속
  - 가장 많이 사용하는 파일 연산

API(파일 연산)	설명
int open(struct inode *, struct file *);	응용 프로그램에서 장치 파일을 int open (const char *pathname, int flasgs) system call을 호출 해서 열 때, 커널이 호출하는 함수이다. 즉 장치 파일을 열 때, 호출된다.
int release(struct inode *, struct file *);	응용 프로그램에서 장치 파일을 int close (int fd) system call을 호출 해서 닫을 때, 커널이 호출하는 함수이다. 즉 장치 파일을 닫을 때, 호출된다.
ssize_t read(struct file *, char *, size_t, loff_t *);	응용 프로그램에서 장치 파일을 ssize_t read (int fd, void *buf, size_t count) system call을 호출해서 읽을 때, 커널이 호출하는 함수이다. 즉 장치 파일을 읽을 때, 호출된다.
ssize_t write(struct file *, const char *, size_t, loff_t *);	응용 프로그램에서 장치 파일을 ssize_t write (int fd, const void *buf, size_t count) system call을 호출 해서 쓸 때, 커널이 호출하는 함수이다. 즉 장치 파일에 쓸 때, 호출된다.
int ioctl (struct inode *, struct file *, unsigned int, unsigned long)	응용 프로그램에서 장치 파일에 대해서 int ioctl(int fd, int request, . . .) system call을 호출할 때, 커널이 호출하는 함수이다.

문자 디바이스 드라이버

- 디바이스 열기 (user side) - open()
  - 가장 먼저 수행됨

```
#include <fcntl.h>
#include <unistd.h>
...
int open(const char  *PathName , int flags);
```

  - Open - 장치 파일을 여는 요구를 할 때 사용한다.
  - *PathName - 열고자 하는 장치 파일 이름을 지정한다.
  - flags - 접근허가
    - O_RDONLY(읽기만 허용), O_WRONLY(쓰기만 허용), O_RDWR(읽기, 쓰기 모두 허용) 값의 조합

문자 디바이스 드라이버

- 디바이스 열기 (kernel side) - *open()

```
int  (*open) ( struct  inode *, struct file *) ;
/* int open(const char  *PathName , int flags); */
```

  - 성공 : 파일 기술자를 리턴
  - 실패 : 음수 값 리턴
  - inode
    - 구조체에 장치를 구분하기 위한 주, 부 번호 정보가 포함되어 있기 때문에 특정 장치를 구분할 수 있다
  - open
    - 포인터 함수에 대응된 모듈 내부의 실제 함수는 보통 초기화 작업을 담당
    - 장치가 다른 사용자 프로그램에 의해 이미 사용되고 있는지 확인
    - 장치가 사용중임을 명시
    - 사용 중에 다른 사용자 프로그램의 방해를 받지 않기 위해 모듈 사용 카운트 수를 증가시킴.
    - 하드웨어 초기화가 필요한 경우 초기화 루틴 포함

문자 디바이스 드라이버

- 디바이스 닫기 (user side)

  #include <unistd.h>

  int close (int FD);

  - FD : open 에 의해 반환 받은 파일 기술자를 인자로 넣어 줌

- 디바이스 닫기 (kernel side)

  int (* release) (struct inode *, struct file *) ;

  - 자신이 종료하였음을 명시
    - release에 명시된 실제 함수는 open 경우와 반대로 모듈 사용 수 카운터를 하나 감소시킴
  - 모듈 사용수 카운트 ≠ 0 은 다른 사용자가 존재함을 의미.

문자 디바이스 드라이버

- 디바이스 읽기 (user side)

```
#include <unistd.h>

ssize_t read(int  fd , void * buf , size_t  count);
```

  - 디바이스의 읽기 동작을 위해 read 함수를 호출
  - fd
    - open에서 얻었던 파일 기술자 fd 가 지시하는 장치를 참조.
  - *buf
    - 위치에 해당되는 데이터를 count 갯 수 만큼 읽어 들인다.
  - Return값
    - 자신이 실제 읽은 바이트 수를 반환 (음수는 에러 의미)
    - EOF(end of file)인 경우 0을 리턴
    - 에러가 발생하면 음수 리턴

문자 디바이스 드라이버

- 디바이스 읽기 (kernel side)

ssize_t (*read) (struct file * , char * , size_t , loff_t *);

- 실제 함수는 해당 장치로부터 데이터를 읽어 오는 코드가 구현됨.
- loff_t - 파일의 읽고 쓰는 위치
- sizt_t - 읽을 데이터의 크기

문자 디바이스 드라이버

- 디바이스 쓰기 (user)

```
#include <unistd.h>

ssize_t  write(int fd , const void * buf , size_t count);
```

  - 파일 기술자 fd 가 지시하는 장치에서, *buf가 가리키는 데이터를count 크기 만큼 쓴다.
  - Return 값
    - 실제 쓰여진 바이트 수
    - 에러 발생시 : 음수
    - 하나도 쓰지 못한 경우 : 0
  - 디바이스 쓰기 (kernel)

```
ssize_t (*write) (struct file * , const  char * , size_t , loff_t *);
```

문자 디바이스 드라이버

- ioctl
  - 읽기/쓰기 이외의 부가적인 연산을 위한 인터페이스
  - 디바이스 설정 및 하드웨어 제어 (향상된 문자 드라이버 작성 가능)
  - ioctl 시스템 콜은 디바이스에 특화된 명령을 나타내는 방법을 제공한다.
    - 예) /linux/driver/cdrom/cdrom.c
      - reading이나 writing이 아닌 CDROMEJECT, CDROMSTART, CDROMSTOP 등 것과 같은 명령
- Ioctl에 전달되는 cmd와 관련 매크로 함수

  xxx_ioctl(, , unsigned int cmd,)
    - 단순한 구별 숫자 이외에 처리에 도움을 주는 몇 가지 정보를 포함한 형태
  - 구분 번호: 명령을 구분하는 명령어의 순차 번호
  - 매직 번호: 다른 디바이스 드라이버의 ioctl 명령과 구분하기 위한 값
  - 데이터 크기: 매개변수 arg를 통해 전달되는 메모리의 크기
  - 읽기 쓰기 구분: 읽기 명령인지, 쓰기 명령인지 구분하는 속성

문자 디바이스 드라이버

- loctl에 전달되는 cmd와 관련 매크로 함수
  - 매크로 함수
    - 앞에서 본 cmd의 4가지 필드 형식에 맞춰 cmd 상수값을 만드는 매크로함수와 cmd 상수값에서 필요한 필드값을 추출하는 매크로 함수를 제공
  - cmd 명령을 만드는 매크로 함수
    - _IO: 부가적인 데이터가 없는 명령을 만드는 매크로
    - _IOR: 디바이스 드라이버에서 데이터를 읽어오기 위한 명령을 만드는 매크로
    - _IOW: 디바이스 드라이버에서 데이터를 써넣기 위한 명령을 만드는 매크로
    - _IOWR: 읽고 쓰기를 수행하기 위한 명령을 만드는 매크로

문자 디바이스 드라이버

- loctl에 전달되는 cmd와 관련 매크로 함수
  - 매크로의 형태
  - _IO(매직 번호, 구분 번호)
  - _IOR(매직 번호, 구분 번호, 변수형)
  - _IOW(매직 번호, 구분 번호, 변수형)
  - _IOWR(매직 번호, 구분 번호, 변수형)
  - 매직번호: 0~255 사이값, 특별한 의미는 없다. 매직번호는 잘못된 사용을 막는 간단한 보안 장치
  - 구분번호: 각 명령을 구분하기 위해 사용. 보통 0부터 순서대로 붙여나간다.
  - 변수형: 변수형은 arg매개변수가 가리키는 데이터의 전달 크기를 지정하는데 사용. 숫자가 아닌 변수형을 넣는다.

문자 디바이스 드라이버

- loctl에 전달되는 cmd와 관련 매크로 함수
  - 매크로 함수
    - 앞에서 본 cmd의 4가지 필드 형식에 맞춰 cmd상수값을 만드는 매크로함수와 cmd상수값에서 필요한 필드값을 추출하는 매크로 함수를 제공
    - cmd명령을 만드는 매크로 함수
    - _IO: 부가적인 데이터가 없는 명령을 만드는 매크로
    - _IOR: 디바이스 드라이버에서 데이터를 읽어오기 위한 명령을 만드는 매크로
    - _IOW: 디바이스 드라이버에서 데이터를 써넣기 위한 명령을 만드는 매크로
    - _IOWR: 읽고 쓰기를 수행하기 위한 명령을 만드는 매크로

문자 디바이스 드라이버

- loctl에 전달되는 cmd와 관련 매크로 함수
  - _IO 매크로 함수
    - 전달되는 매개변수가 없고, 단순히 명령만 전달할 때 사용.
    - #define TEST_DRV_RESET_IO(12, 0)
  - _IOR 매크로 함수
    - 디바이스에서 데이터를 읽어오는 명령을 만들 때 사용.
    - #define TEST_DRV_READ_IOR(12,1,int)
    - 디바이스에서 응용프로그램이 읽어올 데이터의 크기가 int 크기 만큼이라는 의미

문자 디바이스 드라이버

- Kernel 영역과 User 영역 사이의 데이터 교환 방식
  - 사용자 영역에서 커널 영역으로 데이터를 이동할 때 ; 데이터 쓰기
    - void copy_from_user(void *to, const void *from, unsigned long count);
    - void get_user(datum ptr);
  - 모듈로부터 사용자 프로그램으로 데이터를 전달할 때 ; 읽기.
    - void copy_to_user(void *to, const void *from, unsigned long count);
    - void put_user(datum ptr);
  - copy_to_user / copy_from_user
    - 데이터 크기는 count 인자로 조정
  - get_user / put_user
    - ptr 변수가 참조하는 데이터 형에 따라 1, 2, 4. 8 byte가 전송된다.

문자 디바이스 드라이버

- 디바이스 드라이버 내부의 구성
  1) 커널 소스 헤더 파일
  2) 디바이스 파일 이름과 주 번호(major number)
  3) 저수준 파일 입출력에 대응하는 file_operations 구조체에 등록할 함수
  4) file_operations 함수
     - 디바이스 드라이버 내부에서 실질적인 동작을 하는 함수들
  5) 문자 디바이스 드라이버를 등록하는 모듈 초기화 함수
     - register_chrdev 함수 호출
  6) 문자 디바이스 드라이버를 제거하는 모듈 마무리 함수
     - unregister_chrdev 함수 호출

문자 디바이스 드라이버

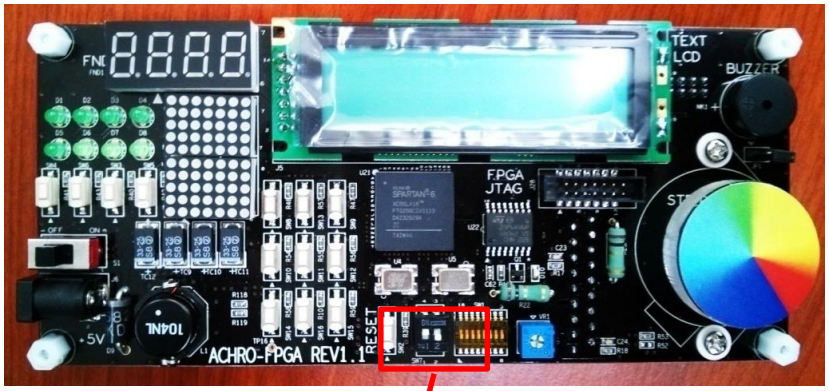
디바이스 드라이버 내부의 구성 예

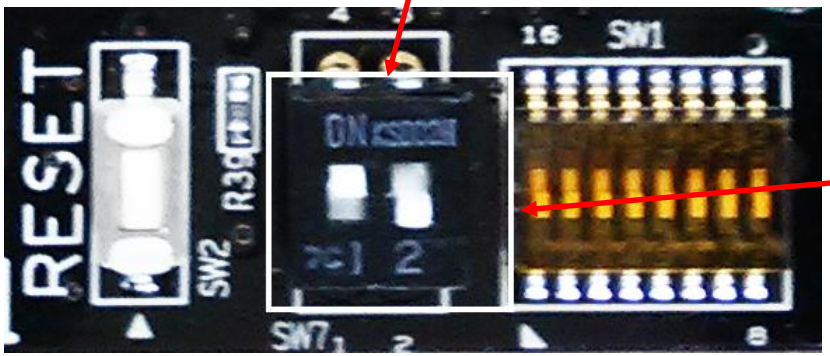
```
01  #include <linux/module.h>
02  #include <linux/kernel.h>
03  #include <linux/fs.h>
04  #include <linux/init.h>
05
06  #define TEST_DEV_NAME "test_dev"
   // 디바이스 파일 이름
07  #define TEST_DEV_MAJOR 240
   // 디바이스 파일의 주번호
08
09  int test_open(struct inode *inode, struct
   file *filp)
10  {
11    ……(open 처리)……
12    return 0;
13  }
14
15  int test_release(struct inode *inode,
   struct file *filp)
16  {
17    ……(close 처리)……
18    return 0;
19  }
20
```

```
21  struct file_operations test_fops =
22  {
23    .owner    = THIS_MODULE,
24    .open     = test_open,
25    .release = test_release,
26  };
27
28  int test_init(void)
   // 디바이스를 커널에 모듈로 적재시 수행되는 함수
29  {
30    register_chrdev(TEST_DEV_MAJOR,
   TEST_DEV_NAME, &test_fops);
31    ……(초기화 처리)……
32    return 0;
33  }
34
35  void test_exit(void)
   // 커널에서 디바이스를 제거할 때 수행되는 함수
36  {
37   ……(마무리 처리)……
38    unregister_chrdev(TEST_DEV_MAJOR,
   TEST_DEV_NAME);
39  }
40
41  MODULE_LICENSE("GPL");
42  module_init(test_init);
43  module_exit(test_exit);
```

디바이스 드라이버 작성 예 - LED Driver

▶ 파일 연산 구조체 - 계속





SW7의 1번 ON, 2번 OFF

디바이스 드라이버 작성 예 - LED Driver

- FPGA의 각 장치별 제어 어드레스 및 노드, 주번호 정보
  - Achro-FPGA 보드는 0x07000000에 매핑 되어있으므로, 베이스 어드레스로부터 해당 장치에 지정된 장데이터가 전송

번호	장치	어드레스	Node	Major
1	LED	0x0700_0016	/dev/fpga_led	260
2	Seven Segment (FND)	0x0700_0004	/dev/fpga_fnd	261
3	Dot Matrix	0x0700_0210	/dev/fpga_dot	262
4	Text LCD	0x0700_0100	/dev/fpga_text_lcd	263
5	Buzzer	0x0700_0020	/dev/fpga_buzzer	264
6	Push Switch	0x0700_0017	/dev/fpga_push_switch	265
7	3Dip Switch	0x0700_0000	/dev/fpga_dip_switch	266
8	Step Motor	0x0700_000C	/dev/fpga_step_motor	267
EN	Demo Register	0x0700_0300	N/A	N/A

디바이스 드라이버 작성 예 - LED Driver

- 개발보드용 디바이스 드라이버 소스 및 Test 파일 가져오기
  - 작업 폴더 생성

    # mkdir /work/achro5250/device_driver

    # cd /work/achro5250/device_driver
  - 디바이스 드라이버 소스 및 test 파일 복사

    # cp -a ~/ACHRO-5250-1.5.0.2/examples/linux/fpga_driver/* .
  - 실습 디바이스 드라이버 리스트

LED	fpga_led.tar.gz
Seven Segment (FND)	fpga_fnd.tar.gz
Dot Matrix	fpga_dot.tar.gz
Text LCD	fpga_text_lcd.tar.gz
Buzzer	fpga_buzzer.tar.gz
Push Switch	fpga_push_switch.tar.gz
3Dip Switch	fpga_dip_switch.tar.gz
Step Motor	fpga_step_motor.tar.gz

디바이스 드라이버 작성 예 - LED Driver

- 개발보드용 디바이스 드라이버 및 Test 파일 Makefile 수정
  - 해당 디바이스용 압축을 푼 후 다음과 같이 작업환경을 변경
  - Makefile의 수정 (vi)

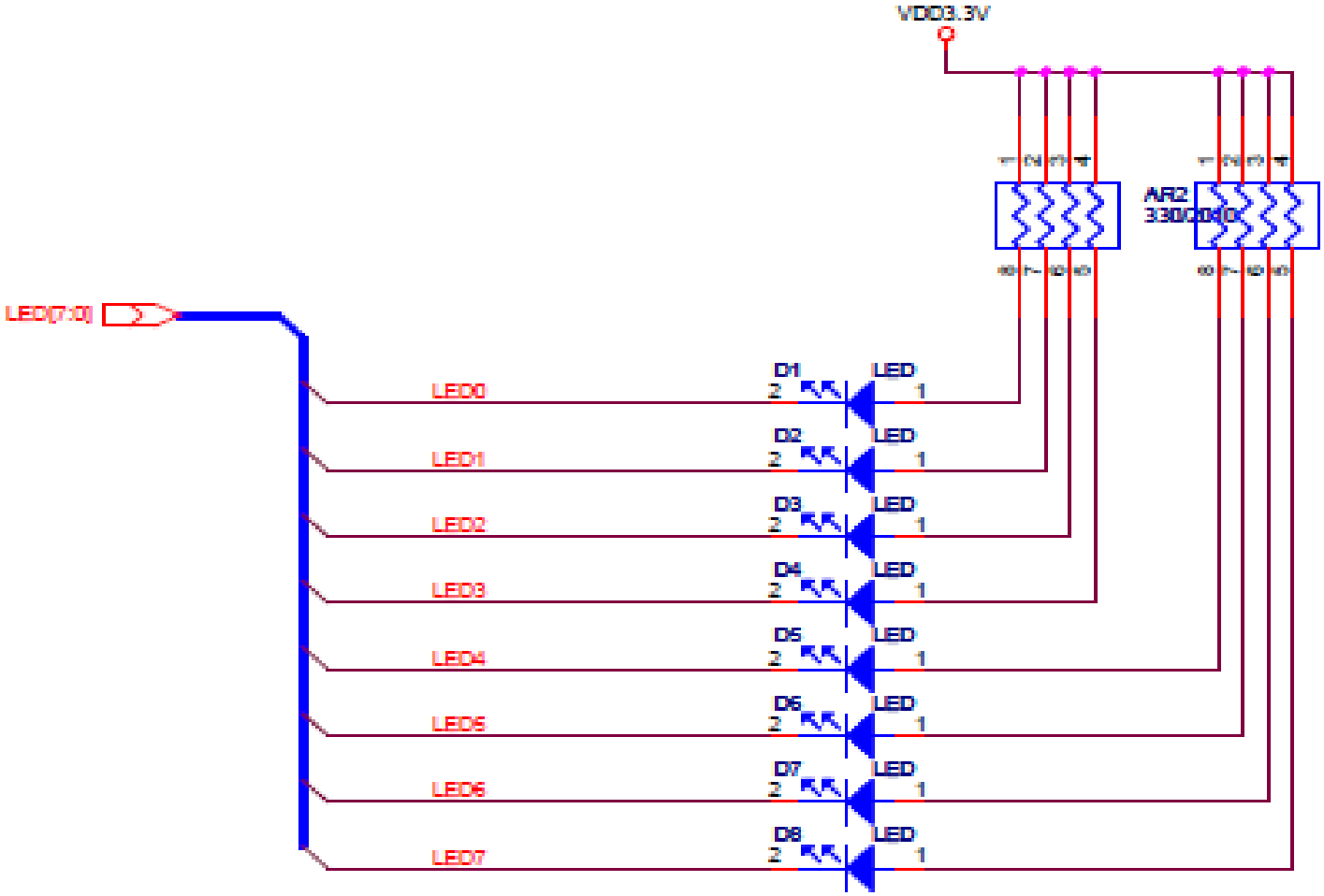
```
.....
KDIR :=/work/achro5250/kernel
....
app:
    arm-none-linux-gnueabi-gcc –static ..................
....
```

  - 상기와 같이 일일이 수정이 불편할 경우 다음과 같은 방법으로 일괄적으로 변경이 가능

```
# cd /work/achro5250
# ln -s kernel kernel-20130306          /* Makefile 내의 kernel-20130306을 kernel로 대체
# cd /opt/toolchains/arm-2010q1/bin
# ftp computer.kpu.ac.kr
Name (computer.kpu.ac.kr: root): anonymous
Passwd: anonymous
ftp> cd achro5250
ftp> get cross-symlink.sh
ftp> bye
# sh cross-symlink.sh                   /* Makefile 내의 app 내용을 일괄 대체
```

디바이스 드라이버 작성 예 - LED Driver

- FPGA LED Driver
  - VDD로부터 전원을 공급 받고 있기 때문에 LED I/O측 값에 따라서 LED가 점등



디바이스 드라이버 작성 예 - LED Driver

- LED 제어 드라이버 소스 (fpga_led_driver.c)

```
/* FPGA LED Ioremap Control
FILE : fpga_led_driver.c
AUTH : largest@huins.com */
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <mach/gpio.h>
#include <mach/regs-gpio.h>
#include <plat/gpio-cfg.h>
#include <linux/platform_device.h>
#include <linux/delay.h>
#include <asm/io.h>
#include <asm/uaccess.h>
#include <linux/kernel.h>
#include <linux/ioport.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/version.h>

#define IOM_LED_MAJOR 260 // ioboard led device major number
#define IOM_LED_NAME "fpga_led" // ioboard led device name
#define IOM_LED_ADDRESS 0x07000016 // physical address
#define IOM_DEMO_ADDRESS 0x07000300
#define UON 0x00 // IOM
#define UOFF 0x01 // IOM
```

디바이스 드라이버 작성 예 - LED Driver

LED 제어 드라이버 소스 - 계속

```
//Global variable
static int ledport_usage = 0;
static unsigned char *iom_fpga_led_addr;
static unsigned char *iom_demo_addr;

// define functions...
ssize_t iom_led_write(struct file *inode, const char *gdata, size_t length, loff_t *off_what);
ssize_t iom_led_read(struct file *inode, char *gdata, size_t length, loff_t *off_what);
int iom_led_open(struct inode *minode, struct file *mfile);
int iom_led_release(struct inode *minode, struct file *mfile);

// define file_operations structure
struct file_operations iom_led_fops = {
    .owner = THIS_MODULE,
    .open = iom_led_open,
    .write = iom_led_write,
    .read = iom_led_read,
    .release = iom_led_release,
};

// when led device open ,call this function
int iom_led_open(struct inode *minode, struct file *mfile)  {
    if(ledport_usage != 0) return -EBUSY;
    ledport_usage = 1;
    return 0;
}
```

디바이스 드라이버 작성 예 - LED Driver

LED 제어 드라이버 소스 - 계속

```
// when led device close ,call this function
int iom_led_release(struct inode *minode, struct file *mfile)  {
    ledport_usage = 0;
    return 0;
}

// when write to led device ,call this function
ssize_t iom_led_write(struct file *inode, const char *gdata, size_t length, loff_t *off_what) {
    unsigned char value;
    const char *tmp = gdata;
    if (copy_from_user(&value, tmp, 1))
    return -EFAULT;
    outb(value,(unsigned int)iom_fpga_led_addr);
    return length;
}

// when read to led device ,call this function
ssize_t iom_led_read(struct file *inode, char *gdata, size_t length, loff_t *off_what) {
    unsigned char value;
    char *tmp = gdata;
    value = inb((unsigned int)iom_fpga_led_addr);
    if (copy_to_user(tmp, &value, 1))
        return -EFAULT;
    return length;
}
```

디바이스 드라이버 작성 예 - LED Driver

▶LED 제어 드라이버 소스 - 계속

```
int __init iom_led_init(void)  {
    int result;
    result = register_chrdev(IOM_LED_MAJOR, IOM_LED_NAME, &iom_led_fops);
    if(result < 0) {
        printk(KERN_WARNING"Can't get any major₩n");
        return result;
    }

    iom_fpga_led_addr = ioremap(IOM_LED_ADDRESS, 0x1);
    iom_demo_addr = ioremap(IOM_DEMO_ADDRESS, 0x1);
    outb(UON,(unsigned int)iom_demo_addr);
    printk("init module, %s major number : %d₩n", IOM_LED_NAME, IOM_LED_MAJOR);
    return 0;
}

void __exit iom_led_exit(void) {
    iounmap(iom_fpga_led_addr);
    iounmap(iom_demo_addr);
    unregister_chrdev(IOM_LED_MAJOR, IOM_LED_NAME);
}

module_init(iom_led_init);
module_exit(iom_led_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Huins");
```

디바이스 드라이버 작성 예 - LED Driver

Linux LED Test Code 소스

```
/* FPGA LED Test Application
File : fpga_test_led.c
Auth : largest@huins.com */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define LED_DEVICE "/dev/fpga_led"

int main(int argc, char **argv) {
    int dev;
    unsigned char data;
    unsigned char retval;
    if(argc!=2) {
        printf("please input the parameter! ₩n");
        printf("ex)./test_led a1₩n");
        return -1;
    }

    data = atoi(argv[1]);
    if((data<0)||(data>255))
    {
        printf("Invalid range! (0~255).₩n");
        exit(1);
    }
    dev = open(LED_DEVICE, O_RDWR);
    if (dev<0) {
        printf("Device open error : %s₩n",LED_DEVICE);
        exit(1);
    }

    retval=write(dev,&data,1);
    if(retval<0) {
        printf("Write Error!₩n");
        return -1;
    }

    sleep(1);
    data=0;

    retval=read(dev,&data,1);
    if(retval<0) {
        printf("Read Error!₩n");
        return -1;
    }

    printf("Current LED Value : %d₩n",data);
    close(dev);
    return(0);
}
```

디바이스 드라이버 작성 예 - LED Driver

▶ 컴파일 스크립트 (Makefile)

```
#Makefile for a basic kernel module
obj-m := fpga_led_driver.o
KDIR :=/work/achro5250/kernel
PWD :=$(shell pwd)
all: driver app
#all: driver
driver:
$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
app:
arm-none-linux-gnueabi-gcc -o fpga_test_led fpga_test_led.c
install:
cp -a fpga_led_driver.ko /nfsroot      // device driver 복사 → nfs 이용해 타겟 보드로 복사
cp -a fpga_test_led /nfsroot           // 테스트 프로그램 복사 → nfs 이용해 타겟 보드로 복사
clean:
rm -rf *.ko
rm -rf *.mod.*
rm -rf *.o
rm -rf fpga_test_led
rm -rf Module.symvers
rm -rf modules.order
rm -rf .led*
```

디바이스 드라이버 작성 예 - LED Driver

- LED 드라이버 압축해제 및 컴파일


```
# cd /work/achro5250/device_driver
# tar xvfz fpga_led.tar.gz
# cd fpga_led
```

  - Makefile의 수정은 symbolic link 설정 시에는 안 해도 됨

```
# make
```

make install ➔ 드라이버 파일 fpga_led_driver.ko, test 파일 fpga_test_led가 nfsroot로 복사

- 타겟 보드에서 실행

```
# cd /mnt/nfs                    // 그냥 nfs 디렉토리에서 실행해도 무방
# mount -t nfs 10.40.1.x:/nfsroot /mnt/nfs -o rw,rsize=1024,nolock
# insmod fpga_led_driver.ko
# mknod /dev/fpga_led c 260 0 // 디바이스와 응용을 묶어주는 노드 생성
# ./fpga_test_led 1
```

Q & A