



Embedded Software

Device Driver

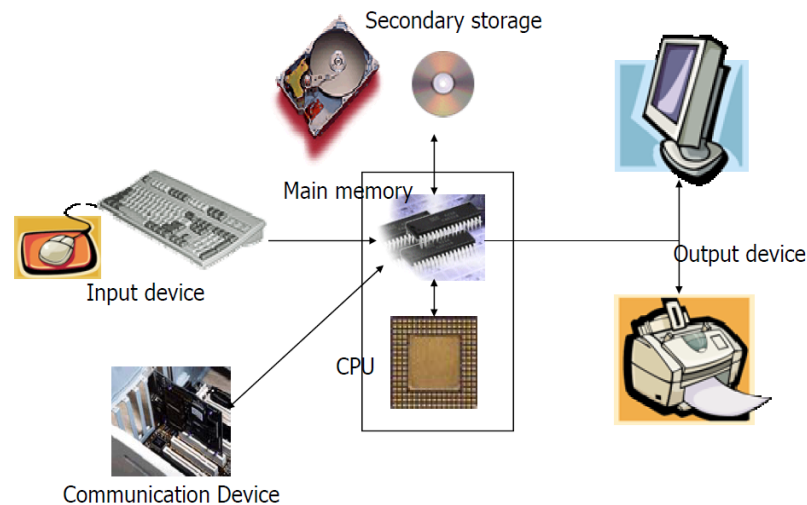
Young-woo Lee

System IC Design Technology Laboratory

Device Driver Overview

▪ Device Driver

- **디바이스(device)**: 하드 디스크(Hard Disk), 프린터(Printer), 스캐너(Scanner), 네트워크 어댑터(Network Adaptor), Touch Screen, LCD 디스플레이, Audio 등과 같이 컴퓨터 시스템 이외의 다른 주변 장치
- 디바이스를 구동하기 위해서는 당연히 디바이스 구동 프로그램인 디바이스 드라이버 프로그램 필요
- 디바이스 드라이버 프로그램은 리눅스 OS가 자체적으로 지원 하는 것도 있지만, 대부분 디바이스에 맞는 드라이버 프로그램을 찾아서 인스톨 시켜야 함



Device Driver Overview

▪ Device Driver

- 물리적인 hardware 장치를 다루고 관리하는 software
- user application이 driver에게 요청을 하면, driver는 hardware를 구동시켜 목적을 달성
- major number와 minor number를 이용하여 각각의 device들을 구분하여 사용
- device와 system memory 간에 data의 전달을 담당하는 kernel 내부 기능

▪ Device Driver의 주요 기능

- ① 디바이스와 시스템 사이에 데이터를 주고 받기 위한 인터페이스(Interface)
- ② 표준적으로 동일한 서비스 제공을 목적
- ③ 커널의 일부분으로 내장
- ④ 서브루틴과 데이터의 집합체
- ⑤ 디바이스의 고유한 특성을 내포

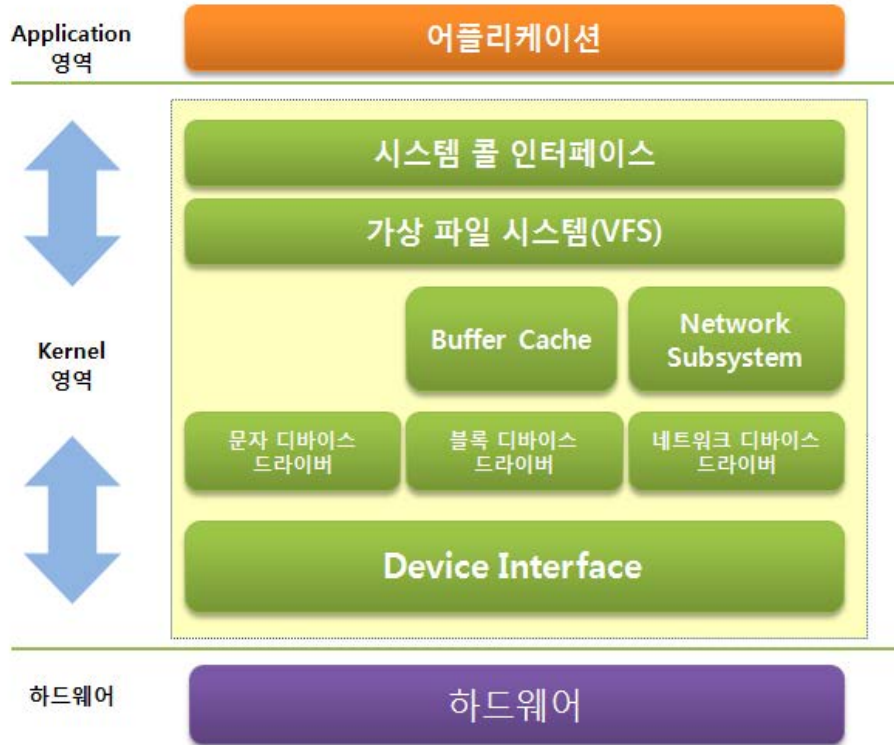
Device Driver Overview

임베디드 리눅스 시스템에서 디바이스를 다루는 방법은 디바이스에 대해서 하나의 파일처럼 파일을 통해 액세스가 가능하도록 하고 있다. 하나의 디바이스는 임베디드 리눅스 시스템에서 주번호 (Major Number)와 부 번호 (Minor Number)로서 표현된다. 디바이스 드라이버의 종류는 문자 디바이스 (Character Device), 블록 디바이스 (Block Device), 네트워크 디바이스 (Network Device)로 나뉜다.

▪ Device Driver 개발방법

- OS가 없는 펌웨어(Firmware) 단계에서 주변장치의 물리주소를 직접 접근하는 방식
- 커널이 부팅된 후 메모리 디바이스의 파일 오퍼레이션인 mmap함수를 이용하여 사용자 애플리케이션에서 가상 메모리 번지를 매핑 하여 사용하는 방식
- 각각의 Device에 대한 디바이스 드라이버를 만들고 응용 프로그램에서 I/O디바이스를 오픈하여 사용하는 방식
- 커널에서 직접 I/O를 제어하는 시스템 함수를 만드는 방식

Device Driver 구조



영역	기능
Application area	사용자 모드에서 동작 하는 프로그램 영역
Kernel area	<p>시스템 콜 인터페이스 : 응용프로그램에게 커널서비스를 제공하는 인터페이스</p> <p>VFS : VFS는 여러 종류의 다양한 파일 시스템에 대해서 공통된 인터페이스를 제공한다.</p> <p>커널 : 프로세스 관리, 메모리 관리, IPC, 파일 시스템, 네트워킹, 디바이스 드라이버등</p>
Hardware	물리적인 하드웨어 영역

Device Driver 구조

▪ 커널 모듈(Kernel Module)

커널 모듈은 리눅스 시스템 부팅 후에 동적으로 로드, 언로드 할 수 있는 커널의 구성 요소이다. 따라서 커널을 다시 컴파일 하거나 시스템을 리부팅하지 않고도 커널의 일부분을 교체할 수 있다.

디바이스 드라이버, 파일 시스템, 네트워크 프로토콜 등이 모듈로 만들어져 있고, 모든 모듈에는 컴파일 한 커널 버전 정보가 들어가야 하며 이는 현재 실행되고 있는 커널 버전과 일치해야 한다. 즉, 모듈은 버전 의존성이 있다는 의미이며, 만일 다르다면 에러가 발생한다.

모듈 버전 정보는 <linux/module.h>에 정의되어져 있으며, 각 모듈 버전은 `char kernel_version[]`에 변수로 지정되어져 있다. 따라서 `insmod`는 모듈을 로딩시 현재 커널의 버전 정보와 항상 비교한다. 모듈의 버전 정보는 전체 모듈에서 하나만 존재해야 한다.

※ 일반 파일과 디바이스 파일

일반 파일이 데이터를 저장하는데 목적이 있다면 디바이스 파일은 시스템 혹은 하드웨어 정보(디바이스 형식, 주번호, 부번호)를 제공하는데 목적이 있다. 일반 파일에 데이터를 쓰면 보존되고 크기도 증가한다. 그러나 디바이스 파일은 데이터를 써도 보존되지 않고 디바이스로 데이터를 전달하게 된다.

Device Driver 구조

▪ 커널 심벌(Kernel Symbol)

커널에서 심벌(Symbol)이란 함수나 변수 이름을 의미한다. 심벌은 다른 곳에서 사용할 수 있도록 심벌을 제공할 (symbol export) 수도 있고, 다른 곳에 존재하는 심벌을 사용할(symbol import) 수도 있다. 또한 공개된 심벌들은 심벌 테이블(symbol table)이란 곳에 리스트로 관리되며, /proc/ksyms에 텍스트 형태로 제공되어 사용자가 읽어 볼 수도 있다. 사용자가 만든 모듈이 커널에 적재되었을 때, 그 안에 선언한 전역 심벌은 전부 커널 심벌의 일부가 되고 이것도 역시 /proc/ksyms에서 찾아볼 수 있다. 심벌 링킹(symbol linking)의 종류로는 static linking와 dynamic linking가 있다.

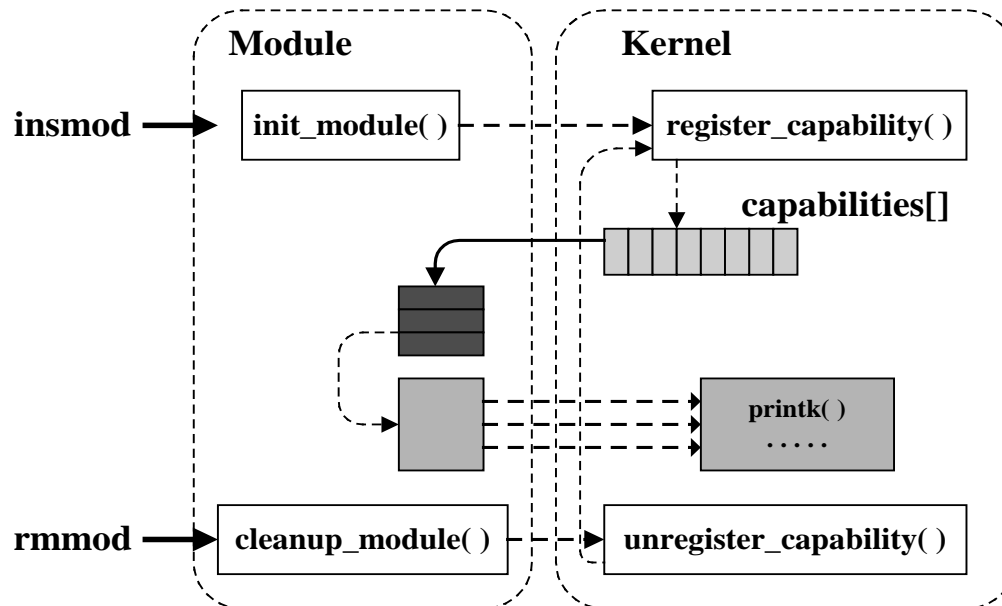
커널의 주 테이블에 모듈의 전체 테이블을 등록하는 함수는 register_symtab()이다. 그리고 심벌들에 버전 정보가 포함되어 있는 경우 같은 심벌이라도 버전 정보의 포함 유무에 따라 export할 때 마다 다르게 된다. 예를 들면 printk()에 커널이 버전 검사를 수행하지 않으면 그냥 printk()로 export되지만, 버전 검사를 수행하면 printk()뒤에 모듈 버전 정보가 추가되어 printk_xxxxxx 형태로 export되게 된다.

```
#ifdef CONFIG_MODVERSIONS
#define MODVERSIONS
#include <linux/modversions.h>
#endif
```

Device Driver 구조

▪ 커널 심벌(Kernel Symbol)

커널(Kernel)에 대한 모듈 프로그램(Module Program)이 일반적인 응용 프로그램과의 주된 차이점으로는 **main()함수가 없고**, 커널에 로딩(loading) 및 언로딩(unloading)할 때에 불리는 **int init_module(void)함수**[성공하면 "0", 실패하면 그 이외의 값을 리턴함]와 **void cleanup_module()함수**가 존재한다는 것이다. 즉, 한 개의 응용 프로그램은 처음부터 끝까지 한 개의 태스크를 수행하지만 한 개의 모듈들은 앞으로 들어올 요청에 대응하는 루틴들을 커널에 등록 후 주 함수는 신속하게 완료된다.



커널에 모듈을 링크하는 개념도

Device Driver 구조

▪ 사용자수 카운트(User Count)

리눅스 시스템은 각 모듈을 사용하고 있는 프로세스(process)의 수를 유지하고 있다. 이것은 모듈에 대한 응용 프로세스들을 액세스(access)정도와 모듈을 삭제시 안전하게 삭제 가능한지 여부를 결정하기 위함이다. 만일 모듈이 사용 중(busy)이라면 그 모듈을 삭제할 수 없기 때문에 리눅스 시스템은 이 값이 반드시 필요하다.

어떤 파일 시스템을 마운트 하는 도중이라면 그 파일 시스템의 타입 모듈을 제거할 수 없고 프로세스가 사용중인 문자 디바이스(character device)를 제거할 수도 없다. 사용자 카운트의 현재 값은 /proc/modules의 각 모듈 항목의 세 번째 필드에 나타나 있으며, 사용자 카운트 값이 "-1"인 경우는 그 모듈이 사용자 카운트를 가지고 있지 않음을 의미한다.

▪ User Count Macro

- MOD_INC_USE_COUNT : 현재 사용 중인 모듈의 카운트 횟수를 증가시킨다.
- MOD_DEC_USE_COUNT : 현재 사용 중인 모듈의 카운트 횟수를 감소시킨다.
- MOD_IN_USE : 현재 사용 중인 것으로 COUNT = 0이 아니면 TRUE이다.

Device Driver 종류

디바이스 드라이버는 문자 디바이스 드라이버, 블록 디바이스 드라이버, 네트워크 디바이스 드라이버 세 가지 타입으로 구분

▪ Character Device

- 데이터 자료의 순차성을 지닌 장치로서 버퍼 캐쉬(buffer cache)를 사용하지 않는다. 즉, 버퍼를 통하지 않고 데이터를 바로 읽고 쓸 수 있는 장치를 말한다. 문자 디바이스의 종류로는 Terminal, Console, Touch Screen, Keyboard, Sound Card, Scanner, Printer, Serial/Parallel, Mouse, Joystick등이 있다.
- 문자 디바이스는 파일 시스템의 이름(또는 노드)를 통해 액세스(access) 되며, 대개 이들의 디바이스는 /dev 디렉토리하에 있다. 블록 디바이스, 네트워크 디바이스도 마찬가지이다. 파일 시스템에 디바이스 노드를 만들려면 mknod명령을 사용해야 하며, 디바이스 파일을 만들려면 슈퍼유저로 접근해야 한다.
- mknod 명령은 디바이스 이름과 디바이스 종류, 주번호(Major Number), 부번호(Minor Number)등 세개의 추가 인자를 반드시 필요로 한다. 주번호는 커널에서 디바이스 드라이버를 구분하는 데 사용된다. 부 번호는 디바이스 드라이버 내에서 필요한 경우 장치를 구분하기 위해서 사용된다. 즉, 하나의 디바이스 드라이버가 여러 개의 디바이스를 제어할 수 있다. 당연히 새로운 디바이스 드라이버는 새로운 주번호(Major Number)를 가져야 한다.

Device Driver 종류

▪ Character Device

- Documentation/devices.txt에 모든 장치의 주번호(Major Number)가 정의되어 있다.
- /dev/hda1은 디바이스 이름이고, b는 블록 디바이스임을 의미한다. c이면 문자 디바이스임을 의미한다. 127은 주번호(major number)를 의미하고, 1은 부번호(minor number)를 뜻한다. 부번호 값은 0 – 255사이의 값을 가져야 한다. 이는 8bit로 제한되어 있기 때문이다.

```
$ mknod /dev/hda1 b 127 1
$ ls -al /dev/hda1
$ brw-rw---- 1 root disk 3 1 Mar 25 12:00 /dev/hda1
```

- /dev/hda1은 디바이스 이름, b는 블록 디바이스임을 의미, c이면 문자 디바이스임을 의미
- 127은 주번호(major number)를 의미하고, 1은 부번호(minor number)를 뜻한다. 부번호 값은 0 – 255사이의 값을 가져야 한다.

Device Driver 종류

▪ Character Device 등록과 해제

- 임베디드 리눅스 시스템에 새로운 디바이스 드라이버(Device Driver)를 추가하는 것은 하나의 주번호(Major Number)를 이 드라이버에 할당함을 의미한다.
- 주번호의 할당은 드라이버 초기화시에 `include/linux/fs.h`에 정의된 디바이스 등록 함수를 호출함으로서 이루어진다.

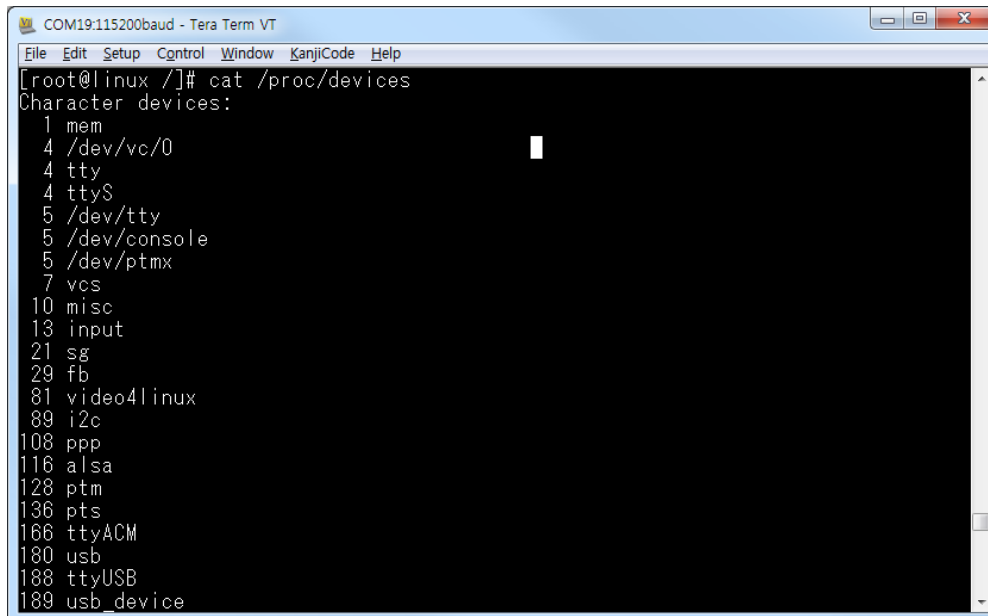
```
static inline int register_chrdev(unsigned int major, const char *name, const struct file_operations *fops)
```

Device Driver 종류

▪ Character Device 등록과 해제

- 디바이스 장치의 등록 함수 인자로서 major 은 등록할 주번호(Major Number)이며, 만일 "0"이면 사용하지 않는 주 번호 중에서 선택하여 동적으로 할당된다.
- name은 디바이스 장치의 이름이며 /proc/devices에 나타나 있다.

```
# cat /proc/devices
```



```
COM19:115200baud - Tera Term VT
File Edit Setup Control Window KanjiCode Help
[root@linux ~]# cat /proc/devices
Character devices:
1 mem
4 /dev/vc/0
4 tty
4 ttyS
5 /dev/tty
5 /dev/console
5 /dev/ptmx
7 vcs
10 misc
13 input
21 sg
29 fb
81 video4linux
89 i2c
108 ppp
116 alsa
128 ptm
136 pts
166 ttyACM
180 usb
188 ttyUSB
189 usb_device
```

실습 키트에서 등록된 디바이스 확인
(/proc/devices 내용)

Device Driver 종류

▪ Character Device 등록과 해제

- 리눅스 시스템에서 모듈을 삭제할 때에는 주번호도 제거하여야 한다.
- `cleanup_module`에서 호출하는 디바이스 장치 해제 함수인 다음의 함수를 사용하면 된다.

```
int unregister_chrdev(unsigned int major, const char *name)
```

- 함수 인자는 제거하고자 하는 주번호(Major Number)와 디바이스 이름(Device Name)이다. 커널은 이 디바이스 이름과 주번호 할당시 등록한 이름을 비교한다. 이름이 서로 다르거나 또는 지정된 드라이버가 없거나 주번호 경계를 벗어난 값에 대해서는 에러 코드인 `EINVAL`을 리턴한다.

Device Driver 종류

▪ Character Device 형태

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

```
int device_open( )
int device_release( )
ssize_t device_write( )
ssize_t device_read( )
```

```
static struct file_operations device_fops = {

    ssize_t (*read)
    ssize_t (*write)

    int (*open)
    int (*release)

}
```

```
static struct fiint init_module(void)
void cleanup_module(void)
```

- Header Files : 디바이스 드라이버에서 사용할 헤더 정의
- Function Prototypes : 디바이스 드라이버에서 사용될 함수
디바이스 특징에 따라 file_operation의 구조체 함수 사용
- File Operation : 함수들을 각각의 특성에 맞게 file_operation 구조체에 등록
- 디바이스 드라이버 커널 삽입/제거

Device Driver 종류

▪ Character Device File Operations

- 사용자 프로그램과 디바이스 드라이버는 파일 인터페이스(file interface)를 통해 액세스된다. 디바이스 드라이버는 자신이 커널에 등록할 때 register_chrdev()함수를 사용하며, 등록 함수의 인자로서 file_operations 구조체를 사용하고 있다.
- File_operations구조체는 include/linux/fs.h에 정의된 파일 함수 포인터 테이블이다.

```
root@ubuntu: /4412_Linux/kernel_4412/include/linux
* the big kernel lock held in all filesystems.
*/
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 (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, 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 (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
```

file구조체

Device Driver 종류

▪ Character Device File Operations

- `loff_t (*lseek) (struct file *, loff_t, int);` : 파일에서 현재 읽고 (read), 쓰는(write) 위치를 이동한다.
- `ssize_t (*read) (struct file *, char *, size_t, loff_t *);` : 디바이스 장치에서 데이터를 읽어들인다.
- `ssize_t (*write) (struct file *, const char *, size_t, loff_t *);` : 디바이스 장치에 데이터를 쓴다.
- `int (*readdir) (struct file *, void *, filldir_t);` : 디바이스 장치에서는 사용하지 않고 (즉, 디바이스 노드에서는 NULL을 리턴함), 디렉토리에서만 사용하는 함수
- `unsigned int (*poll) (struct file *, struct poll_table_struct *);` : 현 프로세스를 대기 큐에 넣는다.
- `int (*mmap) (struct file *, struct vm_area_struct *);` : 디바이스 장치의 메모리를 프로세스의 메모리에 매핑시키는 데 사용한다.
- `int (*open) (struct inode *, struct file *);` : 디바이스 장치를 오픈한다.
- `int (*release) (struct inode *, struct file *);` : 디바이스 장치를 종료한다.
- `int (*fsync) (struct file *, struct dentry *);` : 디바이스 장치를 flush한다. 즉, 데이터 중에서 디바이스에 있는 것은 모두 디바이스에 쓴다.
- `int (*fasync) (int, struct file *, int);` : 비동기 통지(Asynchronous notification)를 알려주는 FLAG인 FASYNC flag에 변화가 있는 디바이스를 확인하기 위해 사용한다.
- `int (*lock) (struct file *, int, struct file_lock *);` : 파일에 lock를 걸기 위해서 사용한다.

Device Driver 종류

▪ Block Device

버퍼 캐쉬(buffer cache)를 통해 Random Access가 가능한 장치. 데이터를 버퍼 캐쉬를 통해 블록(Block)단위로 입출력하며, 파일 시스템을 구축할 수 있다. 그 종류로는 Hard Disk, Floppy Disk, RAM Disk, CD-ROM등이 있다.

블록 디바이스는 문자 디바이스와 같이 외부 사용자 응용 프로그램과는 파일 인터페이스를 통해 액세스되며, 파일 연산(file operations)을 제공한다. 블록 디바이스 드라이버는 주번호(Major Number)로 구별되며, 문자 디바이스 드라이버와는 별개로 관리됨으로 문자 디바이스 드라이버와는 주번호가 중첩되어도 관계 없다. 파일 시스템은 버퍼 캐쉬(buffer cache)를 통해서 블록 디바이스에 액세스 된다. 버퍼 캐쉬를 통한 작업은 일반적인 read/write 동작과는 다르기 때문에 파일 연산의 read/write와 별도로 버퍼 캐쉬를 통한 read/write를 수행하기 위하여 추가적인 구조체 `blk_dev_struct{ }`가 존재한다. 이 구조체내에서는 버퍼 캐쉬를 통해 실질적인 read/write를 수행하는 `request_fn()`함수를 구현해야 한다. 또한, 블록의 크기, 섹터(sector)의 크기, 전체 크기 등을 따로 블록 장치를 관리하는 전역 변수에 설정해 관리해야한다

Device Driver 종류

▪ Block Device 등록과 해제

임베디드 리눅스 시스템에서 문자 디바이스 드라이버처럼 블록 디바이스 드라이버는 커널에서 주번호(Major Number)로 식별한다.

```
int register_blkdev(unsigned int major, const char *name, struct file_operations *fops)
int unregister_blkdev(unsigned int major, const char *name)
```

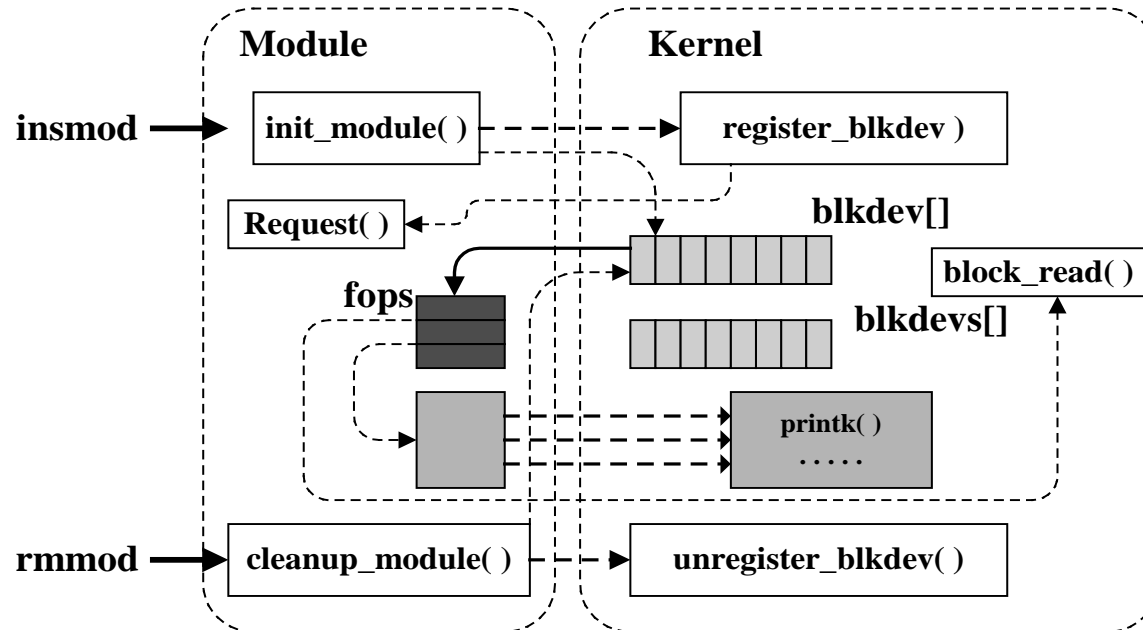
블록 디바이스 드라이버를 등록하고 해제 하기 위해 사용하는 함수

- 문자 디바이스 드라이버에서와 마찬가지로 블록 디바이스 장치의 등록 함수 인자는 major 로서 등록할 주번호 (Major Number)이며, 만일 “0”이면 사용하지 않는 주번호 중에서 선택하여 동적으로 할당된다. name은 블록 디바이스 장치의 이름이다. fops 는 블록 디바이스 장치에 대한 파일 연산 함수이다. 리턴 값은 에러 코드이며 리턴 값으로 “0”이나 양수의 값이면 성공적으로 수행되었음을 나타내고, 음수 값이면 에러가 발생했음을 나타낸다

Device Driver 종류

▪ Block Device 등록과 해제

- 블록 디바이스 드라이버 모듈과 커널 사이의 관계



Device Driver 종류

▪ Block Device File Operation

사용자 프로그램과 블록 디바이스 드라이버는 파일 인터페이스를 통해 액세스된다. 그러나 블록 디바이스 드라이버의 파일 연산은 문자 디바이스 드라이버와 조금 차이가 있다. read/write/fsync 함수는 따로 구현할 필요가 없으며, 일반적으로 block_read(), block_write(), block_fsync() 함수를 그대로 사용한다. 그리고 open, release 함수도 문자 디바이스 드라이버에서와 마찬가지로 구현한다. 그러나 ioctl에서는 많이 다르다. 대부분의 블록 디바이스 드라이버가 지원할 것으로 예상되는 수 많은 공통 ioctl 명령이 있으며, 따라서 이들의 대부분은 반드시 지원해 주어야 한다.

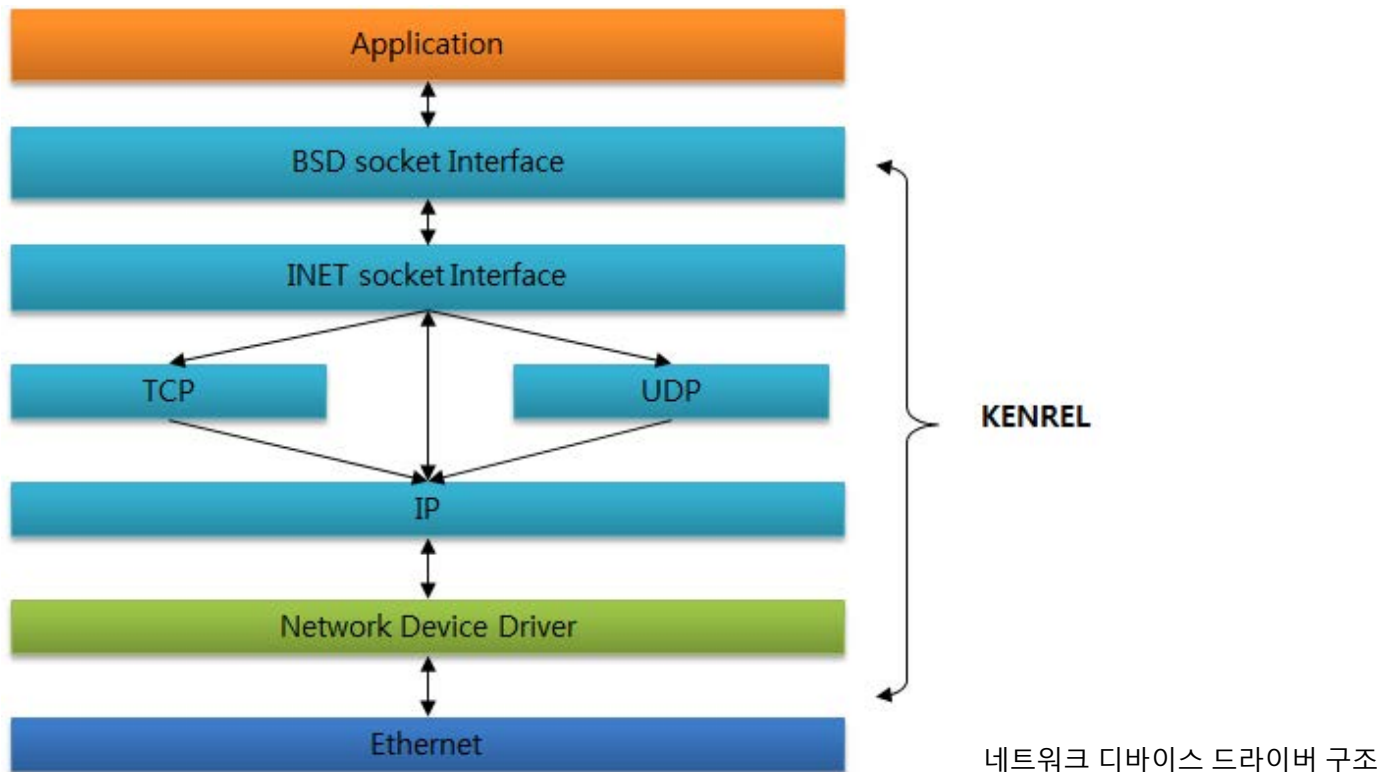
내용으로는 먼저 장치의 크기를 섹터의 개수로 리턴하는 BLKGETSIZE, 버퍼 캐시를 flush하는 BLKFLSBUF, 얼마나 미리 읽기(read-ahead)를 할 것인지 미리 읽기 값을 획득하는 BLKRAGET, 미리 읽기 값을 세트하는 BLKRASET, 블록 디바이스 장치의 읽기 전용 FLAG을 얻어오는 BLKROGET, 변경하는 BLKROSET, 파티션이 있는 장치에서 파티션 테이블을 다시 읽어오는 BLKRRPART, 하드 디스크의 구조를 읽어오는 HDIO_GETGEO가 있다.

그리고 check_media_change는 미디어(media)가 변경될 수 있는 블록 디바이스 장치의 경우 미디어가 변경되었는지 여부를 조사하는 데 사용된다. 미디어가 변경된 경우 "1"을, 변경되지 않은 경우 "0"을 리턴 한다. Revalidate는 미디어가 변경된 경우 호출되는 함수이다.

Device Driver 종류

▪ Network Device Driver

네트워크 디바이스(Network Device)는 기본적으로 네트워크 통신을 통해 네트워크 패킷을 송수신할 수 있는 장치를 말한다. 그 예로는 Ethernet, PPP, Slip, ATM, ISDN NIC(Network Interface Card)등이 있다.



Device Driver 종류

▪ Network Device Driver

- Socket

네트워크를 소켓이라는 단일한 인터페이스로 묶어서 사용하며, 이를 통하여 통신의 요소를 가상화 한다. Network을 초기화하여 다양한 통신 수단을 사용할 수 있으며, Network의 초기화는 통신이라는 것을 file system의 체제로 초기화하는 것이다. 그리고 하나의 연결이란 하나의 file을 등록하고 초기화하여 file descriptor에 연결한 후 file operation을 규정하는 행위와 같다. 이것은 `init_module()`과 `open()`의 과정으로 볼 수 있으며 소켓에서는 이를 `socket()`과 `bind()`함으로써 수행한다.

- Association

소켓 프로그램에 있어서 로컬 Host에서 실행되는 프로세스와 원격 Host에서 실행되는 프로세스간에 데이터를 교환하기 위해 요구되는 정보이다. 프로토콜 식별자, 로컬 인터넷 주소, 로컬 포트 번호, 원격 인터넷 주소, 원격 포트번호로 구성된다.

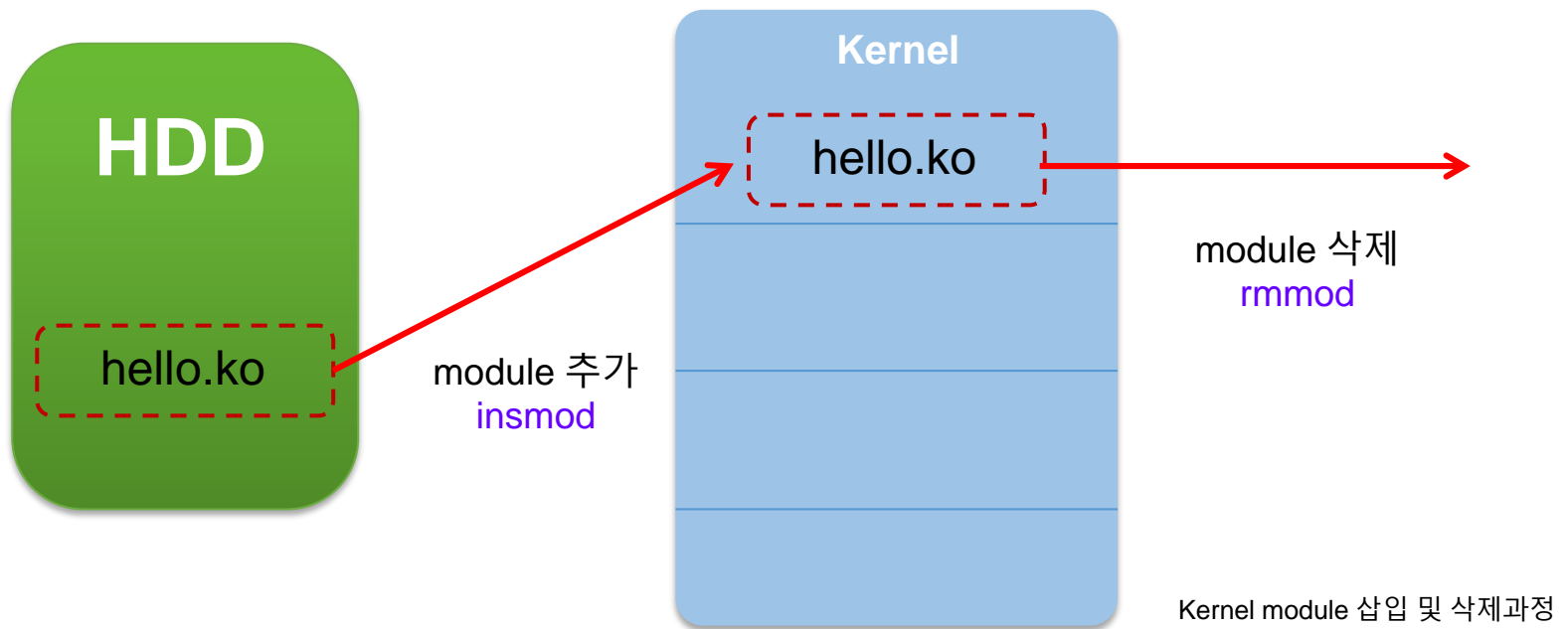
- Filesystem과 socket의 관계

File System에서는 file을 식별하기 위해서 file descriptor를 사용한다. Socket은 하나의 connection (or association)을 식별하기 위한 수단이다. 프로토콜 식별자, 포트 번호, 인터넷 주소, 사용안함으로 구성된다. File system에는 file descriptor와 연결되고, TCP/IP에서는 port 번호와 IP에 할당된다. File System과 Socket과의 관계는 사용자에게 간편함을 제공하기 위하여 connection을 하나의 file descriptor와 연결한다. 따라서 사용자는 통신을 일반 file처럼 사용할 수 있다.

모듈 프로그래밍

Kernel 2.x 버전이 진화하면서 가장 큰 변화 중 하나는 커널 모듈 개념이 추가된 것이다. 구조적 문제 때문에 커널 크기를 줄이기 위한 방법 중의 하나로 등장한 커널 모듈 개념은 단일화 구조의 Linux 커널에 동적으로 기능의 추가/제거를 가능하게 해주었다.

- 커널 메모리 영역에 module을 삽입(추가)하거나 삭제하기 위한 예제 소스 코드 작성



모듈 프로그래밍

■ 예제소스 작성 "hello.c"

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

MODULE_LICENSE("GPL");

int module_start(void)
{
    printk ("Hello World!!\n");
    return 0;
}

void module_end(void)
{
    printk("Goodbye World!! \n");
}

module_init(module_start);
module_exit(module_end);
```

.h : 커널 모듈과 관련된 헤더파일

- **module_start(void)** : 커널 모듈이 적재될 때 호출되는 함수
insmod 명령어를 실행하면 호출
- **module_end** : 커널 모듈이 삭제될 때 호출되는 함수
rmmod 명령어를 실행하면 호출
- **printk()** : printf() 함수와 유사한 함수로 문자열 출력 함수
linux/kernel.h에 정의되어 있으며 Kernel Module을 리눅스 커널이
실행시킬때 관련 정보들을 출력, 출력정보의 중요도에 따라 메시지의
우선순위를 지정(8가지 레벨)

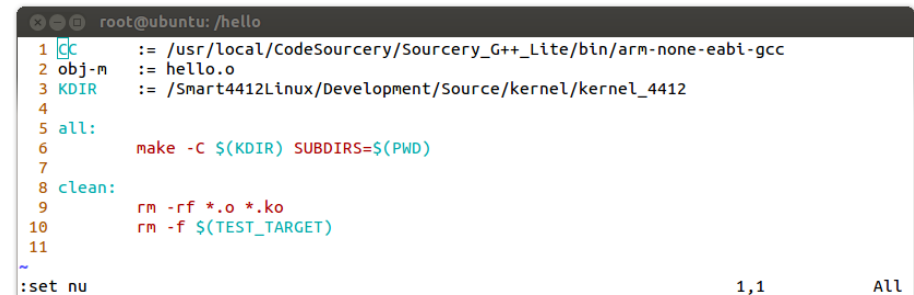
모듈 프로그래밍

▪ Makefile 작성

```
CC := /usr/local/CodeSourcery/Sourcery_G++_Lite/bin/arm-none-eabi-gcc
obj-m := hello.o
KDIR := /Smart4412Linux/Development/Source/kernel/kernel_4412

all:
    make -C $(KDIR) SUBDIRS=$(PWD)
clean:
    rm -rf *.o *.ko
    rm -f $(TEST_TARGET)
```

- CC : arm-linux-gcc 컴파일러가 설치되어있는 경로
- obj-m : 목적파일 이름 설정
- KDIR : 커널 경로
- all : 목적파일을 만들기 위해 hell.c 파일의 위치 설정
- clean : make clean 명령을 통해 목적파일 삭제 명령



```
root@ubuntu: /hello
1 CC := /usr/local/CodeSourcery/Sourcery_G++_Lite/bin/arm-none-eabi-gcc
2 obj-m := hello.o
3 KDIR := /Smart4412Linux/Development/Source/kernel/kernel_4412
4
5 all:
6     make -C $(KDIR) SUBDIRS=$(PWD)
7
8 clean:
9     rm -rf *.o *.ko
10    rm -f $(TEST_TARGET)
11
~
:set nu                                1,1      All
```

모듈 프로그래밍

- 모듈 컴파일

실행 결과로 hello.ko 파일이 생성. hello.ko파일을 Target Board에 다운로드하여 insmod명령을 사용하여 로딩

```
# make
```

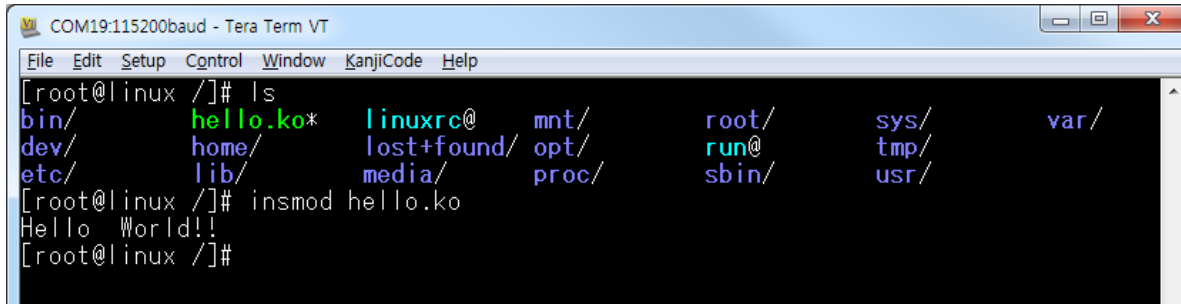
```
root@ubuntu: /hello
root@ubuntu:/hello# ls
hello.c  Makefile
root@ubuntu:/hello# make
make -C /Smart4412Linux/Development/Source/kernel/kernel_4412 SUBDIRS=/hello
make[1]: Entering directory `/Smart4412Linux/Development/Source/kernel/kernel_4412'
  LD      /hello/built-in.o
  CC [M]  /hello/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /hello/hello.mod.o
  LD [M]  /hello/hello.ko
make[1]: Leaving directory `/Smart4412Linux/Development/Source/kernel/kernel_4412'
root@ubuntu:/hello# ls
built-in.o  hello.ko    hello.mod.o  Makefile      Module.symvers
hello.c     hello.mod.c  hello.o      modules.order
root@ubuntu:/hello#
```

모듈 프로그래밍

- 모듈 적재(삽입) [실습 키트가 없으므로 생략]

insmod응용프로그램에 의해서 커널 모듈이 적재되면, 목적파일의 내용이 커널 영역으로 복사 된다.

```
# insmod hello.ko
```



```
COM19:115200baud - Tera Term VT
File Edit Setup Control Window KanjiCode Help
[root@linux ~]# ls
bin/      hello.ko*  linuxrc@  mnt/      root/      sys/      var/
dev/      home/      lost+found/ opt/      run@      tmp/
etc/      lib/       media/    proc/     sbin/     usr/
[root@linux ~]# insmod hello.ko
Hello World!!
[root@linux ~]#
```

※ 커널 모듈 관련 유틸리티

insmod : 커널 모듈을 적재한다.

rmmod : 커널 모듈을 제거한다.

lsmod : 현재 적재된 커널 모듈들의 이름을 출력한다.

modprobe : insmod와 유사하지만, 모듈간의 의존성을 검사해서 필요한 모든 커널 모듈을 적재한다.

depmod : 모듈 간의 의존성을 검사한다.

kemeld : kernel thread로 돌아가며, 커널에서 필요한 모듈을 자동으로 적재하고 일정 시간동안 사용하지 않는 모듈을 제거하는 역할을 한다.

모듈 프로그래밍

- 모듈 목록 확인 [실습 키트가 없으므로 생략]

적재되어있는 모듈을 확인하기 위해서 모듈 내부에 작성한 `printk()`의 실행결과를 `dmesg`명령을 이용하여 확인한다. 마지막 라인에 "Hello World" 라는 메시지가 나타난다.

```
# dmesg
```

```
readAdapterInfo_8192CU(): REPLACEMENT = 1
<=== ReadAdapterInfo8192C in 305 ms
rtw_register_early_suspend
rtw_macaddr_cfg MAC Address = 08:10:77:05:89:83
MAC Address from pnetdev->dev_addr= 08:10:77:05:89:83
bDriverStopped:1, bSurpriseRemoved:0, bup:0, hw_init_completed:0
usbcore: registered new interface driver rtl8192cu
ttyS3: LSR safety check engaged!
Hello World!!
[root@linux /]#
```

- 모듈 목록 확인

적재되어있는 `hello.ko` 모듈파일을 해제하기 위해 `rmmod`명령을 사용한다.

```
# rmmod hello
```

```
ttyS3: LSR safety check engaged!
Hello World!!
[root@linux /]#
[root@linux /]# rmmod hello
Goodbye World!!
[root@linux /]#
```

Any Questions?

Young-woo Lee
E-mail. ylee@jnu.ac.kr