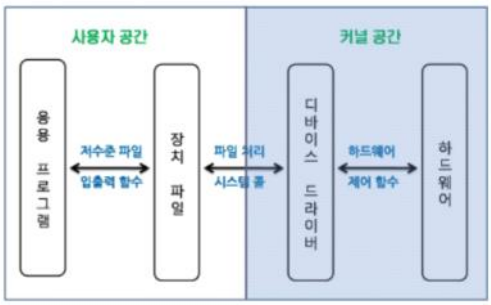
11

2024년 11월 11일 월요일 오후 12:23

장치 드라이버 인터페이스

커널 장치 드라이버란?
내부 하드웨어를 시스템의 다른 부분에서 접근하도록 하는 메커니즘

장치 드라이버의 역할



장치 드라이버는 위의 커널과 아래의 하드웨어 사이에서 중재하는 코드
리눅스에서는 모든 것을 파일로 관리
연결되어 있는 하드웨어 장치 또한, 장치 파일(Device file) 로 관리
이러한 장치 파일에 대한 컨트롤을 "**디바이스 드라이버(Device Driver)** 라는 프로그램" 을 통해 관리하는 것
이 때, 디바이스 드라이버는 "정해져 있는 하드웨어 제어함수" 를 통해 장치를 제어하도록 작성

장치 드라이브는 가능한 한 단순하게 만드는 것이 원칙

- **디바이스 드라이버의 종류**

a. 문자 디바이스 드라이버(chrdev)
버퍼링되지 않는 입출력을 위한 드라이버로 다양한 기능을 갖고 있으며 애플리케이션 코드와 드라이버 사이에 존재하는 얇은 레이어

b. 블록 디바이스 드라이버(blkdev)
이는 블록 입출력과 대용량 저장 장치를 위해 만들어진 인터페이스

c. 네트워크 디바이스 드라이버(netdev)
블록 장치와 유사하나 디스크 블록 보다는 네트워크 패킷을 전송하고 수신할 때 사용된다는 점에서 다름

문자 장치

문자 장치는 사용자 공간에서 장치 노드라는 특수 파일로 식별할 수 있음
해당 파일명은 주번호와 부번호를 이용해 장치 드라이버에 매핑
대체로 주번호는 장치 노드를 특정 장치 드라이버에 매핑하고, 부번호는 드라이버에 어떤 인터페이스가 액세스되는지 알려줌

```
user@raspberrypi:~ $ ls -l /dev/ttyAMA*
crw-rw---- 1 root dialout 204, 64 Oct 11 09:17 /dev/ttyAMA0
```

주번호 204, 부번호 64

장치 노드는 여러가지 방법으로 생성할 수 있음

devtmpfs	드라이버(ttyAMA)에서 제공하는 기본 이름과 인스턴스 번호를 사용해 장치 드라이버가 새로운 장치 인터페이스를 등록할 때 생성되는 노드
udev 나 mdev	기본적으로는 devtmpfs와 동일하지만, 사용자 공간 데몬 프로그램이 sysfs 에서 장치명을 추출해 노드를 생성해야 하는 점이 다름
mknod	정적 장치 노드는 mknod를 사용해 수동으로 생성해줘야 함

사용자가 문자 장치 노드를 열었을 때 커널은 주번호와 부번호가 문자 장치 드라이버에 등록된 범위에 해당하는지를 확인
주번호와 부번호가 해당 범위에 해당한다면 커널은 드라이버를 호출
장치 드라이버는 어떤 하드웨어 인터페이스를 사용할 것인지 확인하기 위해 부번호를 추출해낼 수 있음

블록 장치

블록 장치도 장치 노드와 관련이 있으며, 주번호와 부번호를 갖고 있음

```
user@raspberrypi:~ $ ls -l /dev/mmcblk*
brw-rw---- 1 root disk 179, 0 Oct 11 09:17 /dev/mmcblk0
brw-rw---- 1 root disk 179, 1 Oct 11 09:17 /dev/mmcblk0p1
brw-rw---- 1 root disk 179, 2 Oct 11 09:17 /dev/mmcblk0p2
```

주번호는 179, 부번호는 0~2
mmcblk0은 2개의 파티션이 있는 카드가 있는 마이크로 SD 카드 슬롯

MMC와 SCSI 블록 드라이버 모두 디스크의 시작 부분에서 파티션 테이블을 찾으며, 파티션 테이블은 fdisk, sfdisk, parted 와 같은 유틸리티를 사용해 생성할 수 있음

사용자 공간 프로그램은 장치 노드를 통해 블록 장치를 직접 열고 조작할 수 있음

네트워크 장치

네트워크 장치는 장치 노드를 통해 접근할 수 없으며 주번호와 부번호를 갖고 있지 않음
대신 네트워크 장치는 문자열과 인스턴스 번호를 기반으로 커널이 할당하는 이름을 갖고 있음

```
my_netdev = alloc_netdev(0, "net%d", NET_NAME_UNKNOWN, netdev_
setup);
ret = register_netdev(my_netdev);
```

1. **alloc_netdev(0, "net%d", NET_NAME_UNKNOWN, netdev_setup);**
   새로운 네트워크 장치를 생성하는 코드

- **0:**
  기본 값 세팅
- **"net%d":**
  네트워크 장치의 이름을 지정하는 포맷
  %d는 자동으로 숫자가 들어가게 되면서, net0, net1 이렇게 지정됨
- **NET_NAME_UNKNOWN:**
  alloc_netdev() 함수에서 자동으로 이름을 지정 (정해져 있지 않아서)
- **netdev_setup:**
  네트워크 장치를 설정하는 함수 포인터, 네트워크 장치의 초기화 작업을 수행하는 데 사용

=> alloc_netdev()를 호출하여 새로운 네트워크 장치를 메모리에서 할당하고 초기화

2. **Ret = register_netdev(my_netdev)**
   생성한 네트워크 장치를 시스템에 등록하는 코드

=> register_netdev()를 호출하여 해당 네트워크 장치를 커널 네트워크 시스템에 등록하고 활성화

(하드웨어 주소를 얻기 위해 드라이버를 조회하는 SIOCGIFHWADDR을 사용하는 예제)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/sockios.h>
#include <net/if.h>

int main(int argc, char *argv[])
{
    int s;
    int ret;
    struct ifreq ifr;
    int i;
    if (argc != 2) {
        printf("Usage %s [network interface]\n", argv[0]);
        return 1;
    }
    s = socket(PF_INET, SOCK_DGRAM, 0);
    if (s < 0) {
        perror("socket");
        return 1;
    }
    strcpy(ifr.ifr_name, argv[1]);
    ret = ioctl(s, SIOCGIFHWADDR, &ifr);
    if (ret < 0) {
        perror("ioctl");
        return 1;
    }
    for (i = 0; i < 6; i++)
        printf("%02x:", (unsigned char)ifr.ifr_hwaddr.sa_data[i]);
    printf("\n");
    close(s);
    return 0;
}
```

이 프로그램은 네트워크 인터페이스 이름을 인수로 사용
소켓을 연 후 인터페이스 이름을 구조체에 복사하고 해당 구조체를 소켓의 ioctl 호출에 전달해 결과 MAC 주소를 출력

실행 시 드라이버 찾기

리눅스 시스템을 사용한다면 어떤 장치 드라이버가 로드돼 있고 그들이 어떤 상태에 있는지를 아는 것이 유용
(/proc과 /sys 에 있는 파일 확인)

(/proc/devices)

```
user@raspberrypi:~ $ cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  5 ttyprintk
  7 vcs
 10 misc
 13 input
 14 sound
 29 fb
 81 video4linux
116 alsa
128 ptm
136 pts
153 spi
180 usb
189 usb_device
204 ttyAMA
226 drm
237 cec
238 media
239 uio
240 binder
241 hidraw
242 rpmb
243 nvme-generic
244 nvme
245 bcm2835-gpiomem
246 vc-mem
247 bsg
248 watchdog
249 ptp
250 pps
251 lirc
252 rtc
253 dma_heap
254 gpiochip
```

```
Block devices:
  1 ramdisk
  7 loop
  8 sd
 65 sd
 66 sd
 67 sd
 68 sd
 69 sd
 70 sd
 71 sd
128 sd
129 sd
130 sd
131 sd
132 sd
133 sd
134 sd
135 sd
179 mmc
259 blkext
```

문자 장치와 블록 장치 목록을 확인할 수 있음

(ifconfig나 ip 명령어를 통해 네트워크 장치 목록 확인 가능)

```
user@raspberrypi:~ $ ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group default qlen 1000
    link/ether d8:3a:dd:25:34:6c brd ff:ff:ff:ff:ff:ff
3: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DORMANT group default qlen 1000
    link/ether d8:3a:dd:25:34:6d brd ff:ff:ff:ff:ff:ff
4: can0: <NOARP,UP,LOWER_UP,ECHO> mtu 72 qdisc pfifo_fast state UP mode DEFAULT group default qlen 20000
    link/can
user@raspberrypi:~ $
```

(lsusb와 lspci 명령어를 사용해 USB와 PCI 버스에 연결된 장치 확인 가능)

```
user@raspberrypi:~ $ lsusb
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 002: ID 2109:3431 VIA Labs, Inc. Hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
user@raspberrypi:~ $ lspci
00:00.0 PCI bridge: Broadcom Inc. and subsidiaries BCM2711 PCIe Bridge (rev 20)
01:00.0 USB controller: VIA Technologies, Inc. VL805 USB 3.0 Host Controller (rev 01)
user@raspberrypi:~ $
```

<sysfs 에서 정보 가져오기>

sysfs는 커널 객체, 속성, 관계를 대표하는 것
커널 객체 = 디렉터리, 속성 = 파일, 관계 = 심볼릭 링크를 듯함

장치 - /sys/devices

```
user@raspberrypi:~ $ ls /sys
block  bus  class  dev  devices  firmware  fs  kernel  module  power
user@raspberrypi:~ $ ls sys/devices
ls: cannot access 'sys/devices': No such file or directory
user@raspberrypi:~ $ ls /sys/devices
armv8_cortex_a72  breakpoint  kprobe  platform  software  system  tracepoint  virtual
user@raspberrypi:~ $
```

- **armv8_cortex_a72**: ARMv8 Cortex-A72 CPU 아키텍처와 관련된 설정이나 코드가 담길 가능성이 있는 디렉토리
- **breakpoint**: 디버깅에서 breakpoints와 관련된 파일이나 설정을 담고 있을 수 있는 디렉토리
- **kprobe**: 리눅스 커널의 함수 추적을 위한 설정 파일이나 데이터가 포함된 디렉토리
- **platform**: 하드웨어 플랫폼 관련 설정
- **software**: 소프트웨어 관련 설정
- **system**: 시스템 전반과 관련된 설정
- **tracepoint**: 커널 추적 관련 파일
- **virtual**: 가상 파일 시스템 관련 파일들

모든 시스템이 공통적으로 갖고 있는 디렉터리

System/	CPU 와 클럭을 비롯한 시스템의 핵심 장치들에 대한 정보를 포함
Virtual/	메모리 기반의 장치에 대한 정보를 포함 Virtual/mem 디렉터리에서 /dev/null, /dev/random, /dev/zero 로 나타나는 메모리 장치들에 대한 정보를 찾을 수 있으며, 루프백 장치에 대한 정보는 virtual/net에서 찾아볼 수 있음
Platform/	기존의 하드웨어 버스에 연결되지 않은 모든 장치에 대한 정보를 보여줌

드라이버 - /sys/class

장치 드라이버의 형태에 따라 정리 (하드웨어적인 관점에서의 목록)
각 서브 디렉터리는 각각의 드라이버를 클래스로 나타내고 있으며, 드라이버 프레임워크 컴포넌트가 구현한 것

```
user@raspberrypi:~ $ ls -d /sys/class/tty/ttyAMA*
/sys/class/tty/ttyAMA0
user@raspberrypi:~ $
```

```
user@raspberrypi:~ $ ls /sys/class/tty/ttyAMA0
close_delay  closing_wait  console  custom_divisor  dev  device  flags  hci0  iomem_base  iomem_reg_shift  io_type  irq  line  port  power  subsystem  type  uartclk  uevent  xmit_fifo_size
user@raspberrypi:~ $
```

device라는 이름의 링크는 장치의 하드웨어 객체를 가리키고 있으며, subsystem이라는 이름의 링크는 부모 디렉터리인 /sys/class/tty를 가리키고 있음

(dev 속성)

```
user@raspberrypi:~ $ cat /sys/class/tty/ttyAMA0/dev
204:64
user@raspberrypi:~ $
```

장치의 주번호와 부번호를 나타냄

드라이버가 해당 인터페이스를 등록할 때 생성되며, 이 파일에서 udev와 mdev가 장치 드라이버의 주번호와 부번호를 찾음

블록 드라이버 - /sys/block

각 블록장치의 서브디렉터리가 존재

```
user@raspberrypi:~ $ ls /sys/block
loop0  loop1  loop2  loop3  loop4  loop5  loop6  loop7  mmcblk0  ram0  ram1  ram10  ram11  ram12  ram13  ram14  ram15  ram2  ram3  ram4  ram5  ram6  ram7  ram8  ram9
user@raspberrypi:~ $ ls /sys/block/mmcblk1
ls: cannot access '/sys/block/mmcblk1': No such file or directory
user@raspberrypi:~ $ ls /sys/block/mmcblk0
alignment_offset  capability  device          diskseq  events_async       ext_range  hidden   inflight   mmcblk0p1  mq     queue  removable  size   stat       trace
bdi               dev         discard_alignment  events   events_poll_msecs  force_ro   holders  integrity  mmcblk0p2  power  range  ro         slaves  subsystem  uevent
user@raspberrypi:~ $
```

Mmcblk을 살펴보면 해당 인터페이스의 속성과 내부 파티션을 확인할 수 있음
시스템에 존재하는 장치(하드웨어)와 드라이버(소프트웨어)에 대해서는 sysfs를 읽어보면 많은 것을 배울 수 있음

사용자 공간의 장치 드라이버

사용자 공간의 장치 드라이버는 크게 두가지로 나뉨

- GPIO와 LED 처럼 sysfs의 파일을 통해 제어하는 방식
- I2C처럼 장치 노드를 통해 공통 인터페이스를 노출시키는 시리얼 버스를 통해 제어하는 방식

<GPIO>

- GPIO는 2개의 상태값 중 하나를 갖는 하드웨어 핀에서 직접 접근할 수 있는 가장 단순한 형태의 디지털 인터페이스
  GPIO 레지스터는 32비트 레지스터로 구성되어 있음

- GPIO는 디지털 입력/출력을 위해 사용되는 하드웨어 핀
- 비트 뱅잉을 통해 I2C, SPI 등 상위 레벨 통신 프로토콜을 구현할 수 있음

- **온칩(On-Chip) GPIO**: 칩 내부에 내장된 GPIO 핀으로, 핀 먹스(Pin Mux) 기술을 이용해 하나의 핀이 여러 기능을 수행
- **오프칩(Off-Chip) GPIO**: 외부 GPIO 확장기나 장치를 I2C, SPI 버스를 통해 연결하여 추가적인 GPIO 기능을 사용하는 방식

- gpiolib는 리눅스 커널에서 GPIO를 관리하는 서브시스템, 일관된 방식으로 GPIO를 노출하고 제어

```
user@raspberrypi:~ $ ls /sys/class/gpio
export  gpiochip0  gpiochip502  gpiochip504  unexport
user@raspberrypi:~ $
```

비트 뱅잉(Bit Banging) 이란?

하드웨어의 특정 기능을 구현하기 위해 소프트웨어적으로 디지털 신호를 직접 제어하는 기술
이 방식은 특정 하드웨어가 필요한 기능을 하드웨어적으로 구현하지 않고, 마이크로컨트롤러나 프로세서의 GPIO(General Purpose Input/Output) 핀을 이용해 직접 신호를 제어함으로써 특정 프로토콜을 처리하는 데 사용

Gpiochip0~gpiochip504 까지의 이름을 가진 디렉터리는 각각 32개의 GPIO 비트를 가진 총 4개의 GPIO 레지스터를 나타냄

```
user@raspberrypi:~ $ ls /sys/class/gpio/gpiochip
gpiochip0/   gpiochip502/ gpiochip504/
user@raspberrypi:~ $ ls /sys/class/gpio/gpiochip504
base  device  label  ngpio  power  subsystem  uevent
user@raspberrypi:~ $
```

base:
레지스터의 첫 번째 GPIO 핀의 숫자 값을 갖고 있음
ngpio:
레지스터의 비트 개수 정보를 갖고 있음

사용자 공간에서 GPIO 비트를 제어하려면 먼저 GPIO 번호를 /sys/class/gpio/export 에 작성해 커널에서 추출해야 함

(LED0에 연결된 GPIO 53에서 진행)

```
user@raspberrypi:~ $ echo 53 > /sys/class/gpio/export
user@raspberrypi:~ $ ls /sys/class/gpio
export  gpio53  gpiochip0  gpiochip502  gpiochip504  unexport
user@raspberrypi:~ $
```

=> 원하는 핀을 제어하기 위해 필요한 파일을 갖고 있는 gpio53이라는 새로운 디렉터리 생성

(gpio53 디렉터리는 다음과 같은 파일을 갖고 있음)

```
user@raspberrypi:~ $ ls /sys/class/gpio/gpio53
active_low  device  direction  edge  power  subsystem  uevent  value
user@raspberrypi:~ $
```

- **direction**: GPIO 핀의 입력/출력 방향을 설정
- **value**: GPIO 핀의 현재 상태(High/Low)를 읽거나 설정
- **edge**: GPIO 핀의 인터럽트 트리거 조건을 설정
- **active_low**: 액티브 로우 상태 여부 설정
- **device**: 핀이 속한 디바이스에 대한 정보
- **subsystem**: 핀이 속한 서브시스템 정보
- **uevent**: 이벤트 알림 시스템
- **power**: 전력 소비 및 관리

핀은 기본으로 입력 모드로 설정 (이를 출력으로 변경하고 싶다면 direction파일에 out이라고 적어야 함)
Value 파일은 현재 핀의 상태 값을 갖고 있으며 0인 경우 low, 1인 경우 high를 나타냄
(가끔 반대인 경우가 있으므로, active_low 파일에 1을 기록하면 의미가 반전돼 low전압은 1로, high 전압은 0으로 표시)
/sys/class/gpio/unexport 에 해당하는 GPIO 번호를 작성하면 사용자 공간에서 GPIO 를 제거할 수 있음
(ex. echo 53 > /sys/class/gpio/unexport)

- **GPIO에서의 인터럽트 처리**

  GPIO 입력은 상태 변경 시 인터럽트를 생성하도록 구성해, 소프트웨어를 비효율적으로 반복적으로 폴링하는 대신 인터럽트를 기다려서 처리
  만약 GPIO 비트가 인터럽트를 생성할 수 있는 상태라면 edge라는 이름의 파일이 존재할 것 (초기값 none, 인터럽트를 생성하지 않는다는 의미)

  인터럽트가 생성되도록 하려면 이 파일 값을 다음중 하나로 지정

rising	상승 천이 시 인터럽트 발생
falling	하강 천이 시 인터럽트 발생
both	상승/하강 천이 시 인터럽트 발생
none	인터럽트 없음

  GPIO 48에서 하강 천이가 나타나는 것을 기다리려면 먼저 인터럽트 활성화

```
user@raspberrypi:~ $ echo 48 > /sys/class/gpio/export
user@raspberrypi:~ $ echo falling > /sys/class/gpio/gpio48/edge
user@raspberrypi:~ $
```

```
user@raspberrypi:~ $ ls /sys/class/gpio
export  gpio48  gpio53  gpiochip0  gpiochip502  gpiochip504  unexport
user@raspberrypi:~ $
```

GPIO에서 인터럽트 신호를 기다리는 방식

1. 먼저 epoll_create를 호출해 epoll 알림 기능을 생성
2. 다음으로는 GPIO 를 열고 초기 값을 읽음
3. Epoll_ctl 을 호출해 POLLPRI를 이벤트로 GPIO의 파일 디스크립터를 등록
4. 마지막으로, epoll_wait 함수를 사용해 인터럽트를 기다림

<LED>

- **LED 제어**: GPIO 핀으로 LED를 제어
- Leds 커널 보조 시스템을 사용하면 밝기 조정 등 더 세밀한 제어가 가능하고, 여러 방식으로 연결된 LED도 다룰 수 있음
- **sysfs 인터페이스**: LED는 **/sys/class/leds** 경로를 통해 제어
- **셸 경로 처리**: 경로명에 콜론(:)을 포함하려면 백슬래시()를 사용해야 함

<I2C>

I2C (Inter-Integrated Circuit)

- **개요**: 저속 2선식 버스로, **임베디드 보드**에서 흔히 사용
  (디스플레이, 카메라 센서, GPIO 확장 모듈 등 SoC 보드에 없는 주변 기기와 통신할 때 사용)
- **관련 표준**: I2C의 변형인 **SMBus**(System Management Bus)는 주로 PC에서 온도 및 전압 센서에 접근할 때 사용
- **구조**: **마스터-슬레이브** 형식
  - **마스터**: 하나 이상의 SoC 호스트 컨트롤러
  - **슬레이브**: 각 기기는 고유한 **7비트 주소**를 가지며, 버스당 최대 128개 노드 연결 가능 (실제 사용 가능한 노드는 112개)
- **통신**: 마스터는 슬레이브와 데이터를 읽거나 쓸 때 트랜잭션을 시작
  - 첫 번째 바이트: 슬레이브의 **레지스터**를 지정
  - 이후 바이트: 레지스터로부터 **데이터 읽기/쓰기**

```
master@build-master:~$ ls -l /dev/i2c*
crw------- 1 root root 89,  0 11월  6 20:32 /dev/i2c-0
crw------- 1 root root 89,  1 11월  6 20:32 /dev/i2c-1
crw------- 1 root root 89, 10 11월  6 20:32 /dev/i2c-10
crw------- 1 root root 89, 11 11월  6 20:32 /dev/i2c-11
crw------- 1 root root 89, 12 11월  6 20:32 /dev/i2c-12
crw------- 1 root root 89, 13 11월  6 20:32 /dev/i2c-13
crw------- 1 root root 89,  2 11월  6 20:32 /dev/i2c-2
crw------- 1 root root 89,  3 11월  6 20:32 /dev/i2c-3
crw------- 1 root root 89,  4 11월  6 20:32 /dev/i2c-4
crw------- 1 root root 89,  5 11월  6 20:32 /dev/i2c-5
crw------- 1 root root 89,  6 11월  6 20:32 /dev/i2c-6
crw------- 1 root root 89,  7 11월  6 20:32 /dev/i2c-7
crw------- 1 root root 89,  8 11월  6 20:32 /dev/i2c-8
crw------- 1 root root 89,  9 11월  6 20:32 /dev/i2c-9
```

i2cdetect	I2C 어댑터 목록을 보여주고 버스를 검사
i2cdump	I2C 주변 기기의 모든 레지스터로부터 데이터를 덤프
i2cget	I2C 슬레이브로부터 데이터를 읽어옴
i2cset	I2C 슬레이브에 데이터를 작성

i2c-tool 패키지는 buildroot와 yocto 프로젝트를 포함한 대부분의 주요 배포판에서 사용 가능

(I2C 프로토콜을 사용하여 EEPROM으로부터 데이터를 읽는 간단한 예제)

슬레이브 주소는 0x50

```c
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/i2c-dev.h>

/* Address of the EEPROM on the BeagleBone Black board */
#define I2C_ADDRESS 0x50

int main(void)
{
    int f;
    int n;
    char buf[10];

    /* Open the adapter and set the address of the I2C device */
    f = open("/dev/i2c-0", O_RDWR);
    if (f < 0) {
        perror("/dev/i2c-0:");
        return 1;
    }

    /* Set the address of the i2c slave device */
    if (ioctl(f, I2C_SLAVE, I2C_ADDRESS) == -1) {
        perror("ioctl I2C_SLAVE");
        return 1;
    }

    /* Set the 16-bit address to read (0) */
    buf[0] = 0; /* address byte 1 */
    buf[1] = 0; /* address byte 2 */
    n = write(f, buf, 2);
    if (n == -1) {
        perror("write");
        return 1;
    }

    /* Now read 4 bytes from that address */
    n = read(f, buf, 4);
    if (n == -1) {
        perror("read");
        return 1;
    }
    printf("0x%x 0x%x0 0x%x 0x%x\n",
        buf[0], buf[1], buf[2], buf[3]);

    close(f);
    return 0;
}
```

<SPI>

- SPI 버스는 I2C와 비슷하지만 최대 수십 MHz 정도의 속도
- 4개의 회선을 이용해 전송과 쉰 회선을 개별적으로 사용하기 때문에 완벽하게 양방향 통신 가능
- 마스터-슬레이브 방식
- 하나 이상의 마스터 호스트 컨트롤러가 구현돼 있음
- CONFIG_SPI_SPIDEV라는 커널을 이용하면 기본 SPI 장치 드라이버를 사용할 수 있음
- 이 드라이버는 사용자 공간에서 SPI 칩에 접근할 수 있도록 해주는 SPI 컨트롤러의 장치 노드를 생성

```
user@raspberrypi:~ $ ls -l /dev/spi*
crw-rw---- 1 root spi 153, 0 Oct 11 09:17 /dev/spidev0.1
```

커널 장치 드라이버 작성

<캐릭터 드라이버 인터페이스 설계>

주 문자 드라이버 인터페이스는 시리얼 포트에서 처럼 바이트 스트림을 기반으로 하고 있음
read와 write만으로 하드웨어를 제어할 수 없는 경우도 종종 발생
=> 그래서 리눅스 커널은 ioctl이라는 방식을 제공

ioctl	드라이버와 애플리케이션 간의 유연한 통신을 가능하게 하며, 구조체를 통해 복잡한 명령을 처리 단점: 커널과 애플리케이션 간의 강한 의존성으로 인해 유지보수가 어려움
sysfs	시스템 상태와 하드웨어 정보를 텍스트 파일 형식으로 제공하여 자동화와 스크립팅을 용이 단점: 각 파일은 한 개의 값만 저장할 수 있어, 여러 값을 동시에 변경하려면 원자성(atomicity)을 유지하기 어려움
mmap	하드웨어 메모리나 커널 버퍼에 직접 접근할 수 있게 해주며, 빠른 데이터 처리에 유리 인터럽트와 DMA 를 제어하려면 커널 코드를 작성해야 할 필요가 있지만, 이를 캡슐화 한 서브시스템인 uio 가 존재
digio	디지털 입출력을 제어하며, 인터럽트나 이벤트 신호를 애플리케이션에 전달할 수 있는 방법을 제공 커널의 함수인 kill_fasync()를 이용하면, 입력 장치가 준비되거나 인터럽트를 받는 등의 이벤트 때 드라이버에서 애플리케이션으로 신호를 보내 이를 알릴 수 있음
debugfs	커널의 디버깅 정보를 파일 형태로 제공하는 디버깅 전용 파일 시스템
proc	시스템과 프로세스에 관한 **실시간 정보**를 제공하지만, 최근에는 사용이 줄어들고 있음 (non-GPL에서 사용 가능)
netlink	커널과 사용자 공간 간 네트워크 통신을 위한 소켓 인터페이스로, 네트워크 상태나 라우팅 테이블 등을 관리

DMA(Direct Memory Access) 란?

중앙 처리 장치(CPU)에서 독립한 입출력 장치를 갖고
주 기억 장치와 직접 데이터를 주고받을 수 있는 방식

놀람 최소화 원칙
사용자가 놀라거나 혼란스럽게 하는 요소를 최소화

<장치 드라이버의 구조>

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

#define DEVICE_NAME "dummy"
#define MAJOR_NUM 42
#define NUM_DEVICES 4

static struct class *dummy_class;

static int dummy_open(struct inode *inode, struct file *file)
{
    pr_info("%s\n", __func__);
    return 0;
}

static int dummy_release(struct inode *inode, struct file *file)
{
    pr_info("%s\n", __func__);
    return 0;
}

static ssize_t dummy_read(struct file *file,
            char *buffer, size_t length, loff_t * offset)
{
    pr_info("%s %u\n", __func__, length);
    return 0;
}

static ssize_t dummy_write(struct file *file,
            const char *buffer, size_t length, loff_t * offset)
{
    pr_info("%s %u\n", __func__, length);
    return length;
}

struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .open = dummy_open,
    .release = dummy_release,
    .read = dummy_read,
    .write = dummy_write,
};
```

> 4개의 장치를 생성하는 dummy라는 이름을 가진 장치 드라이버

문자 장치 인터페이스에 대한 dummy_open(), dummy_release(), dummy_read(), dummy_write() 함수를 정의
=> Open(2), read(2), write(2), close(2) 를 사용할 때 호출됨

```c
struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .open = dummy_open,
    .release = dummy_release,
    .read = dummy_read,
    .write = dummy_write,
};

int __init dummy_init(void)
{
    int ret;
    int i;

    printk("Dummy loaded\n");
    ret = register_chrdev(MAJOR_NUM, DEVICE_NAME, &dummy_fops);
    if (ret != 0)
        return ret;

    dummy_class = class_create(THIS_MODULE, DEVICE_NAME);
    for (i = 0; i < NUM_DEVICES; i++) {
        device_create(dummy_class, NULL,
                      MKDEV(MAJOR_NUM, i), NULL, "dummy%d", i);
    }

    return 0;
}

void __exit dummy_exit(void)
{
    int i;

    for (i = 0; i < NUM_DEVICES; i++) {
        device_destroy(dummy_class, MKDEV(MAJOR_NUM, i));
    }
    class_destroy(dummy_class);

    unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
    printk("Dummy unloaded\n");
}

module_init(dummy_init);
module_exit(dummy_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Chris Simmonds");
MODULE_DESCRIPTION("A dummy driver");
```

dummy_init(): 드라이버가 로드될 때 장치 등록과 초기화를 담당
dummy_exit(): 드라이버가 언로드될 때 리소스를 해제
file_operations 구조체: 문자 장치가 수행할 연산들을 정의
module_init과 module_exit 매크로: 드라이버의 초기화와 종료 함수를 지정

마지막에 있는 MODULE_*라는 이름의 세 매크로는 모듈에 대한 몇가지 기본 정보를 추가한 것
Modinfo 명령을 사용하면 컴파일된 커널 모듈에서 정보 확인 가능

=> 드라이버가 구현한 4개의 함수의 포인터를 갖고 있는 struct file_operations 로 포인터를 넘기면서 register_chrdev 를 호출할 때 dummy 가 문자 장치가 되는 것을 확인할 수 있음

<커널 모듈 컴파일하기>

- **소스 트리 외부에서 모듈로 컴파일 하는 방법**

  커널 빌드 시스템을 이용한 간단한 makefile 작성

```
# make

LINUXDIR ?= $(SDKTARGETSYSROOT)/usr/src/kernel

obj-m := dummy.o

all:
    make -C $(LINUXDIR) M=$(shell pwd)

clean:
    make -C $(LINUXDIR) M=$(shell pwd) clean
```

  LINUXDIR을 먼저 모듈을 실행할 장치의 커널 디렉터리에 지정
  obj-m := dummy.o 코드: 커널 빌드 규칙을 호출해 **dummy.c** 라는 소스 파일을 가져온 후 **dummy.ko** 라는 커널 모듈 생성

- **커널 소스 트리 내에서 드라이버 빌드하는 방법**

  갖고 있는 드라이버의 유형에 맞는 디렉터리 고르기

  - **문자 장치 드라이버**: drivers/char 디렉터리에 .c 파일 넣기
  - **블록 장치 드라이버**: drivers/block 디렉터리에 파일 넣기
  - **네트워크 장치 드라이버**: drivers/net 디렉터리에 파일 넣기

  Make 파일 수정
  무조건 모듈로 드라이버를 빌드하는 한 줄을 추가 (m)

```
obj-m += dummy.o
```

무조건 내장 요소로 빌드하고 싶다면 다음 행 추가 (y)

```
obj-y += dummy.o
```

<커널 모듈 로딩>

- 모듈의 유틸리티

insmod [모듈명].ko : 모듈의 커널 적재
rmmod [모듈명] : 커널에 적재된 모듈 제거
lsmod [모듈명] : 커널에 적재된 모듈 확인
modprobe / depmod : 모듈의 의존성 검사

하드웨어 구성 정보 찾기

장치 드라이버는 보통 하드웨어와 상호작용 하기 위해 쓰이는 경우가 대부분이지만 각 하드웨어는 각각 다른 구조와 다른 주소 값을 갖고 있으므로
장치 드라이버의 일부는 처음에 어떤 하드웨어인지 파악하기 위해 사용
리눅스의 표준 드라이버 모델에서 장치 드라이버는 PCI, USB, 오픈 펌웨어, 플랫폼 장치 등의 보조 시스템 중 적절한 것을 찾아 자신을 등록
이러한 등록 과정에는 고유 식별자와 드라이버의 ID와 하드웨어의 ID 가 일치할 경우 호출되는 probe 함수를 부르는 콜백 함수가 포함돼 있음

<장치 트리, 플랫폼 데이터> & <장치 드라이버와 하드웨어의 연동>

항목	장치 트리 (Device Tree)	플랫폼 데이터 (Platform Data)
하드웨어 정의 위치	**DTS (Device Tree Source)** 파일에 정의	**C 구조체**로 정의 (struct platform_device)
하드웨어 리소스 정의	- reg (메모리 주소 등) - interrupts (인터럽트 번호)	- IORESOURCE_MEM (메모리) - IORESOURCE_IO (I/O 포트) - IORESOURCE_IRQ (인터럽트 번호)
연동 방식	compatible 속성을 통해 **드라이버**와 연동	driver.name을 통해 **드라이버**와 연동
드라이버 등록	compatible이 일치하면 **probe** 함수 호출	name이 일치하면 **probe** 함수 호출
주요 특징	유연하고 동적으로 하드웨어 변경 가능	하드웨어 정보가 보드 초기화 시 코드에 고정됨

플랫폼 데이터는 보통 보드가 초기화될 때 커널에 등록돼야만 함

Probe 함수는 인터페이스에 대한 정보를 갖고 옴

- 가장 중요하게 기억해야 할 점
- 드라이버가 probe 함수를 등록해야 함
- 커널이 알고 있는 하드웨어와 일치하는 항목을 찾을 때 probe 함수를 호출할 수 있는 충분한 정보를 등록해야 함

리소스 유형은 다음과 같은 플래그로 결정

IORESOURCE_MEM	메모리 영역의 물리 주소
IORESOURCE_IO	I/O 레지스터의 포트 번호나 물리 주소
IORESOURCE_IRQ	인터럽트 번호

```
static const struct of_device_id smc91x_match[] = {
    { .compatible = "smsc,lan91c94", },
    { .compatible = "smsc,lan91c111", },
    {},
};
MODULE_DEVICE_TABLE(of, smc91x_match);
static struct platform_driver smc_driver = {
    .probe = smc_drv_probe,
    .remove = smc_drv_remove,
    .driver = {
        .name = "smc91x",
        .of_match_table = of_match_ptr(smc91x_match),
    },
};
static int __init smc_driver_init(void)
{
    return platform_driver_register(&smc_driver);
}
static void __exit smc_driver_exit(void)
{
    platform_driver_unregister(&smc_driver);
}
module_init(smc_driver_init);
module_exit(smc_driver_exit);
```

```
static int smc_drv_probe(struct platform_device *pdev)
{
    struct smc91x_platdata *pd = dev_get_platdata(&pdev->dev);
    const struct of_device_id *match = NULL;
    struct resource *res, *ires;
    int irq;   .

    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    ires platform_get_resource(pdev, IORESOURCE_IRQ, 0);
    [...]
    addr = ioremap(res->start, SMC_IO_EXTENT);
    irq = ires->start;
    [...]
}
```

```
match = of_match_device(of_match_ptr(smc91x_match), &pdev->dev);
if (match) {
    struct device_node *np = pdev->dev.of_node;
    u32 val;
    [...]
    of_property_read_u32(np, "reg-io-width", &val);
    [...]
}
```