리눅스 환경에서 USB 디바이스 제작에 관한 연구

이양원

호남대학 정보통신공학부

목 차

Ⅰ. 서 론

USB는 1996년도에 인텔, 마이크로소프트, 컴팩, IBM, NEC, DEC, Nortel 등 정보통신의 선두 7개 업체가 협의하여 개발하였으며, 진정한 플러그 앤 플레이(PnP)를 위한 PC 주변장치의 Bus 규격으로서, 새로운 주변기기가 접속되었을 때 재부팅이나 셋업 과정 없이 자동인식으로 최대 127개의 장치를 연결할 수 있을 뿐더러 데이터 전송 속도도 빠르게 향상된 것을 목표로 시작하였다. 1998년도 USB1.1이 나왔으며 2000년도에는 USB 2.0이 나와서 기존의 USB1.0보다도 속도가 48배가 빨라서 현재는 거의 모든 PC에서 연동장치로 사용되고 있다. 실제로 2003년을 기점으로 국내외 모든 출하 PC는 USB2.0 포트를 지원하도록 하고 있어서 이제 USB는 PC 연동의 대표적인 표준이 되어가고 있다.

한편 리눅스 USB 서브시스템은 커널 2.2.7에서는 오직 두 가지의 형태의 디바이스 (즉 마우스와 키보드)만 지원이 가능하였으나 커널 2.4에 이르러서는 20가지가 넘는 형태의 디바이스를 지원할 정도로 성장하였다. 현재 리눅스는 거의 모든 USB 급 디바이스들(표준 형태의 디바이스인 키보드, 마우스, 모뎀, 프린터, 스키터)과 계속 증가하고 있는 벤더 사양 디바이스들(USB to 시리얼 컨버터, 디지털카메라, 이더넷 장치 및 MP3 플레이어)을 지원한다. 현재 지원되고 있는 여러 가지 형태의 USB 디바이스 전체 목록은 http://www.qbik.ch/usb/devices/에서 제공되고 있다.

리눅스에서 지원하지 않은 나머지 USB 디바이스의 종류는 거의 모두 벤더 사양 디바이스들이다. 각 벤더는 자기의 디바이스에 대해서 회사 고유의 프로토콜을 구현하여 사용하기 때문에 보통 제품마다 고유의 드라이버가 필요하다. 어떤 회사들은 USB 프로토콜을 공개하여서 리눅스 드라이버를 개발하는데 도움을 주고 있지만 대부분의 회사들은 프로토콜을 공개하지 않는다. 따라서 디바이스 개발자들은 리버스 엔지니어링(reverse engineering)을 강제로 수행할 수밖에 없다.

따라서 각기 다른 프로토콜은 새로운 드라이버를 개발해야 되는 문제가 발생한다. 본 연구에서는 이 같은 문제를 해결하기 위하여 리눅스 커널 소스 트리에 있는 것으로서 많은 PCI 네트워크 드라이버가 기반으로하고 있는 pci-skeleton.c를 이용하여 일반적인(generic) USB 드라이버 뼈대(driver skeleton)를 만드는 과정을 제안한다.

이 USB 뼈대는 /usr/src/linux-2.4.18-4/drivers/usb/usb-skeleton.c의 커널 소스 트리에서 찾을 수 있을 것이다.

본 논문에서는 뼈대 드라이버의 기본 사항들을 탐구하면서 이것과 각 회사에서 요구하는 구체적인 디바이스를 개발하는데 필요한 것들과 차이점들을 설명할 것이다.

II. 리눅스 USB 기본

만일 리눅스 USB 드라이버를 만들고자 원한다면 우선 USB 프로토콜 사양을 숙지 해야될 것이다. USB 사양은 USB 홈페이지인 http://www.usb.org에 기본 사양과 USB 관련 유용한 자료들을 제공하고 있다. USB 서브시스템에 대한 소개 사이트로서는 아주 우수한 것으로서 USB working devices list(http://www.qbik.ch/usb/devices)가 있다. 여기서는 리눅스 USB 서브시스템이 어떻게 구조화되어 있는가를 설명하며, 독자들에게 USB 드라이버에 필수적인 USB urbs(USB Request Block, or URB for short)의 개념을 제공하고 있다.

리눅스 USB 드라이버가 해야 하는 첫 번째 사항은 그 자신을 리눅스 USB 서브시스템에 등록(register)하는 것이다. 이것은 그 드라이버가 지원하는 디바이스가 어떤 것인가 하는 정보를 제공하여 주며, 또한 드라이버에 의해서 지원되는 디바이스가 시스템으로부터 삽입되거나 제거될 때 호출되는 함수가 무엇인가를 알려주는 것이다. 이 같은 모든 정보는 usb_driver structure안의 USB 서브시스템으로 전달된다. 뼈대 드라이버는 usb_driver를 다음과 같이 선언한다: [source 183-191]

```
static struct usb_driver skel_driver = {
        name: "skeleton",
        probe: skel_probe,
        disconnect: skel_disconnect,
        fops: &skel_fops,
        minor: USB_SKEL_MINOR_BASE,
        id_table: skel_table,
};
```

변수에서 name은 드라이버를 설명하는 문자열이다. 이것은 시스템 로그(System Log)에 프린트되는 정보 메시지로서 사용된다. probe 와 disconnect 함수 포인터들은 id_table 변수에서 제공되는 정보와 일치하는 디바이스가 보이거나 제거될 때 호출된다.

fops와 minor 변수는 옵션이다. 대부분의 USB 드라이버는 SCSI, 네트워크 혹은 TTY 서브시스템과 같은 다른 커널 서브시스템을 가로챈다. 이런 형태의 드라이버는 다른 커널 서브시스템과 함께 스스로 등록한다. 그리고 어떤 유저 공간의 상호작용은 그 인터페이스를 통하여 제공된다. 그러나 이 같은 매칭 커널 서브시스템을 가지고 있지 않은 드라이버 즉 MP3 플레이어, 스캐너는 유저 공간과 상호작용하는 별도의 방법이 요구된다. 따라서 USB 서브시스템은 minor 디바이스 번호와 유저 공간 상호작용을 가능하게 해주는 file_operations 함수 포인터 집합을 등록하는 방법을 제공하고 있다. 뼈대 드라이버는 이런 종류의 별도 인터페이스를 요구한다. 그래서 뼈대 드라이버는 minor 시작 번호와 file_operations 함수의 포인터를 제공한다.

다음으로 USB 드라이버는 usb_register의 호출과 함께 등록되어지는데 보통 아래와 같은 드라이버의 init 함수에서 이루어진다: [source 659-675, 688]

```
static int __init usb_skel_init(void)
{
        int result;
        /* register this driver with the USB
 subsystem */
        result = usb_register(&skel_driver);
        if (result < 0) {
          err("usb_register failed for the
 "__FILE__ "driver."
                "Error number %d", result);
                return -1;
        }
        return 0;
}
module_init(usb_skel_init);
```

드라이버가 시스템으로부터 해제(unload)되었을 때 USB 서브시스템에서 그 자신을 등록말소시킬 필요가 있다.

이것은 usb_deregister 함수를 가지고 다음과 같이 수행된다:[source 678-687, 689]

```
static void __exit usb_skel_exit(void)
{
        /* deregister this driver with the
USB subsystem */
        usb_deregister(&skel_driver);
}
module_exit(usb_skel_exit);
```

디바이스를 끼웠을 때 드라이버가 자동으로 로드가 되는 리눅스 hotplug 시스템이 가능하게 하기 위해서는 별도로 MODULE_DEVICE_TABLE를 만들어야한다. 다음 코드는 이 모듈이 하나의 구체적인 벤더 ID와 제품 ID를 가진 단일 디바이스를 지원하는 hotplug 스크립트를 말한다: [source 79-85]

```
/* table of devices that work with this driver */
static struct usb_device_id skel_table [] = {
        {
USB_DEVICE(USB_SKEL_VENDOR_ID,
 USB_SKEL_PRODUCT_ID) },
        { } /* Terminating entry */
};
MODULE_DEVICE_TABLE (usb, skel_table);
```

USB 드라이버 전체 클래스를 지원하는 드라이버에 대한 usb_device_id를 설명하는데 사용할 수 있는 다른 매크로들이 있는데 이것은 /usr/src/linux-2.4.18-4/include/linux/usb.h에 더 많은 정보가 들어있다.

III. 디바이스 동작(Device Operation)

어떤 디바이스를 USB 버스에 꽂았을 때 만일 이 디바이스가 USB 코어(USB core)에 등록된 드라이버의 디바이스 ID 패턴과 일치하는 것이라면 probe 함수가 호출된다. usb_device structure, 인터페이스 번호, 인터페이스 ID 들은 probe 함수로 전달된다 : [source 132, 505-620]

```
static void * skel_probe(struct usb_device *dev,
unsigned int ifnum, const struct usb_device_id *id)
```

이제 드라이버는 이 디바이스가 실질적으로 받아들일 수 있는 것인가를 증명하는 것을 필요로 한다. 만일 받아들일 수 없다거나 혹은 초기화 과정에 어떤 에러가 발생하면 NULL 값을 probe 함수에 되돌려준다. 그렇지 않으면 이 디바이스에 대한 드라이버의 상태를 포함하고 있는 private data structure의 포인터를 되돌려 준다. 이 포인터는 usb_device structure에 저장되어지며, 드라이버에 대한 모든 callback들은 이 포인터를 통과한다.

뼈대 드라이버(skeleton driver)에서 우리는 end point가 bulk-in과 bulk-out으로 표시된다는 것을 결정하였다. 우리는 디바이스로 보내거나 디바이스로부터 받을 데이터를 유지하고 있는 버퍼들을 만들었다. 그리고 디바이스에 데이터를 쓰기 위한 USB urb를 초기화하였다. 또한 우리는 디바이스를 devfs 서브시스템에 등록하여서 devfs의 유저들이 디바이스에 접근이 가능하도록 하였다. 이 같은 등록 과정은 다음과 같다: [source 599-607]

```
/* initialize the devfs node for this device and
register it */
sprintf(name, "skel%d", skel->minor);
skel->devfs = devfs_register (usb_devfs_handle,
                              name,

DEVFS_FL_DEFAULT,
        USB_MAJOR,

USB_SKEL_MINOR_BASE+ skel->minor,
        S_IFCHR | S_IRUSR | S_IWUSR |
        S_IRGRP | S_IWGRP | S_IROTH,
        &skel_fops, NULL);
```

만일 devfs_register 함수가 실패하면, 우리는 개의치 않는다. 왜냐면 devfs 서브시스템이 이 사실을 유저에게 보고할 것이기 때문이다.

역으로 하면 디바이스가 USB 버스로부터 제거

될 때, disconnect 함수가 디바이스 포인터와 함께 호출된다. 드라이버는 할당된 어떤 private data를 깨끗이 청소하는 것과 USB 시스템에 있던 대기하고 있는 urbs를 shutdown 시키는 것을 요구한다. 또한 드라이버는 호출(call)과 함께 devfs 서브시스템으로부터 자기 자신을 등록 해제한다: [source 640-641]

```
/* remove our devfs node */
devfs_unregister(skel->devfs);
```

이제 디바이스가 시스템에 물려지고(plugged into) 드라이버가 디바이스의 경계안에 있게 되면 USB 서브시스템으로 통과되어지는 file_operations structure내의 어떤 함수들은 디바이스와 대화를 시도하고자 하는 유저 프로그램으로부터 호출되어질 수 있다. 프로그램이 I/O을 위해서 디바이스를 열기를 시도할 때, 호출되는 최초의 함수는 open 함수일 것이다. 만일 MODULE_INC_USE_COUNT에 대한 호출을 가진 모듈이다면 뼈대 드라이버의 open 함수 내에서 우리는 드라이버의 사용 횟수(driver's usage count)를 증가시킨다. 이 매크로 호출 즉 MODU LE_INC_USE_COUNT를 가지고서, 만일 드라이버가 모듈(module)로서 컴파일되었다면 드라이버는 MODULE_DEC_USE_COUNT 매크로가 호출되기 전까지는 unload 할 수 없다. 우리는 또한 private usage count도 증가시키고 file structure 내의 내부 structure에 대한 포인터를 저장해둔다. 이것은 file operations에 대한 장차 있을 호출 시에 드라이버가 사용자가 어떤 디바이스를 addressing하는지를 결정하는 것이 가능하게 하기 위함이다. 이 같은 모든 행위는 다음 코드로서 이루어진다:[source 254, 271-275][1)]

```
/* increment our usage count for the module */
MOD_INC_USE_COUNT;
++skel->open_count;

/* save our object in the file's private structure */
file->private_data = skel;
```

open 함수가 호출된 후에 read와 write 함수들이 디바이스에 데이터를 보내거나 읽기 위해서 호출된다. skel_write 함수에서 우리는 유저가 디바이스에 보내기를 원하는 어떤 데이터와 데이터의 size에 대한 포인터를 수신한다. 그 함수는 write urb[2)]의 크기에 근거로 하여 얼마나 많은 량의 데이터를 디바이스로 보낼 수 있는 가를 결정한다(이 사이즈는 디바이스가 가지고 있는 bulk out end point의 size에 의존한다). 그리고 나서 유저 공간으로부터 커널 공간(kernel space)으로 데이터를 복사하고서, urb가 데이터를 지시하게 하고서 urb를 USB 서브시스템에 전송한다. 이 같은 과정은 다음 코드로서 수행된다: [source 416-443]

```
/* we can only write as much as 1 urb will hold
*/
bytes_written = (count > skel->bulk_out_size) ?
skel->bulk_out_size : count;
/* copy the data from user space into our urb */
copy_from_user(skel->write_urb->transfer_buffer,
 buffer, bytes_written);
/* set up our urb */
usb_fill_bulk_urb(skel->write_urb,
            skel->dev,
            usb_sndbulkpipe(skel->dev,
 skel->bulk_out_endpointAddr),
            skel->write_urb->transfer_buffer,
            bytes_written,
            skel_write_bulk_callback,
            skel);

/* send the data out the bulk port */
result = usb_submit_urb(skel->write_urb);
if (result) {
            err("Failed submitting write urb, error
 %d", result);
}
```

write urb가 FILL_BULK_URB 함수를 이용하여 적절한 정보로서 채워졌을 때 우리는 urb's completion callback(urb의 완성 콜백)이 우리 자

1) 원래 source에는 skel이 dev로 되어 있음

2) 생성한 것임

신의 skel_write_bulk_callback 함수를 호출하도록 지시한다. 이 함수는 urb가 USB 서브시스템에 의해서 종료되었을 때 호출된다. callback 함수는 인터럽트 context(전후관계)에서 호출된다. 따라서 이때는 많은 프로세싱을 행하는 것은 주의를 요한다. 우리가 여기서 구현하는 skel_write_bulk_callback 은 단지 urb가 성공적으로 완수되었는가 혹은 실패했는가를 보고하는 것이다.

read 함수는 wirte 함수와는 약간 다르게 작동한다. 즉 이것은 디바이스로부터 드라이버로 데이터를 전송하는데 urb를 사용하지 않는다. 대신에 우리는 usb_bulk_msg 함수를 호출한다. usb_bulk_msg 함수는 urbs의 생성과 urb completion callback을 다루는 것을 하지 않고서도 디바이스로부터 데이터를 보내거나 받을 수 있다. 우리는 usb_bulk_message 함수를 호출할 때는 디바이스로부터 수신된 어떤 데이터와 timeout 값을 저장할 버퍼를 제공한다. 만일 timeout 주기가 디바이스로부터 어떤 데이터를 받는 것 없이 만료되면 그 함수는 실패하고 에러메시지를 보내주게 된다. 이 같은 행위는 다음과 같은 코드로서 구현된다: [source 361-374]

```
/* do an immediate bulk read to get data from the
device */
retval = usb_bulk_msg (skel->dev,
                       usb_rcvbulkpipe (skel->dev,
                       skel->bulk_in_endpointAddr),
                       skel->bulk_in_buffer,
                       skel->bulk_in_size,
                       &count, HZ*10);
/* if the read was successful, copy the data to user
space */
if (!retval) {
  if (copy_to_user (buffer, skel->bulk_in_buffer,
count))
  retval = -EFAULT;
 else
 retval = count;
}
```

usb_bulk_msg 함수는 하나의 디바이스에 한번 읽고 쓰는 경우에는 매우 유용할 수 있다. 따라서 만일 당신이 디바이스에 계속해서 읽고 쓰기를 원한다면 이때는 자기자신의 urb를 설정하여서 urb을 USB 서브시스템에 전송하여 사용하는 것을 권고한다.

유저 프로그램이 디바이스에게 대화하는데 사용하는 file handle 을 해제하면 드라이버의 해제 함수(release function)가 호출된다. 이 함수에서 우리는 module usage count를 MOD_DEC_USE_COUNT에 대한 호출과 함께 감소시킨다.(이것은 앞에서 우리가 사용하였던 MOD_INC_USE_COUNT의 호출과 대응되는 것임). 우리는 또한 어떤 다른 프로그램이 현재 디바이스에게 대화를 하고 있는지를 결정한다.(이것은 디바이스는 한번에 하나 이상의 프로그램에 open 되어질 수 있을 수도 있기 때문이다.) 만일 이것이 디바이스의 마지막 유저라면 현재 발생하고 있을 수 있는 어떤 가능한 쓰고 있는 중이라도 shutdown 시킨다. 이것은 다음과 같은 코드로서 행해진다: [source 321-330]

```
/* decrement our usage count for the device */
--skel->open_count;
if (skel->open_count <= 0) {
        /* shutdown any bulk writes that might
be going on */
        usb_unlink_urb (skel->write_urb);
        skel->open_count = 0;
}
/* decrement our usage count for the module */
MOD_DEC_USE_COUNT;
```

USB 드라이버들이 부드럽게 조정되게 할 수 있게 하는데 가장 어려운 문제들 중의 하나는 현재 프로그램이 디바이스와 대화를 하고 있는 중간에도 USB 디바이스가 어떤 임의의 시점에서 시스템으로부터 제거되어질 수 있냐는 사실이다. 현재하고 있는 어떤 read와 write를 shutdown 시킬 수 있고 유저 공간 프로그램에게 디바이스가 더 이상 유효하지 않다고 알릴 수 있는 것이 필요하다. 이 같은 행위는 다음과 같은 코드로서 행해질 수 있다:[source 643-654, sub 219-229]

```
/* if the device is not opened, then we clean right
now */
if (skel->open_count) {
        minor_table[skel->minor] = NULL;
        if (skel->bulk_in_buffer != NULL)
                kfree (skel->bulk_in_buffer);
        if (skel->bulk_out_buffer != NULL)
                kfree (skel->bulk_out_buffer);
        if (skel->write_urb != NULL)
                usb_free_urb(skel->write_urb);
        kfree (skel);
} else {
        skel->dev = NULL;
        up (&skel->sem);
}
```

만일 한 프로그램이 현재 디바이스에 대하여 open handle를 가지고 있다면, 이제 디바이스는 멀리 가버렸으므로 우리는 단지 우리의 local structure에서 usb_device structure를 null로 한다. 디바이스가 존재할 때 기대할 수 있는 모든 read, write, release 등의 함수들에 대해서 드라이버는 처음에 이 usb_device structure가 아직도 존재하는지 여부를 점검한다. 만일 존재하지 않으면 디바이스가 사라진 것을 해제하고 -ENODEV 에러를 유저 공간 프로그램에게 전달한다. release 함수가 궁극적으로 호출되면 usb_device structure가 없는지 아닌지를 결정한다. 그리고 skel_disconnect function이 보통 디바이스위에 open file들이 없는가를 보고서 청소한다: [source 312-319]

```
if (skel->dev == NULL) {
/* the device was unplugged before the file was
released */
        minor_table[skel->minor] = NULL;
        if (skel->bulk_in_buffer != NULL)
                kfree (skel->bulk_in_buffer);
        if (skel->bulk_out_buffer != NULL)
                kfree (skel->bulk_out_buffer);
        if (skel->write_urb != NULL)
                usb_free_urb
(skel->write_urb);
        kfree (skel);
        goto exit;
}
```

IV. Isochronous data

이 usb-skeleton 드라이버는 디바이스로부터 데이터를 읽거나 디바이스로 데이터를 쓰는데 있어서 어떤 인터럽트나 isochronous 방식의 데이터 전송의 예를 가지고 있지 않다. 인터럽트 데이터는 몇 개의 미미한 차이점을 제외하고는 거의 bulk 데이터와 같은 방식으로 행해진다. Isochronous 데이터는 디바이스로 데이터를 주고 받는데 있어서 데이터의 연속 스트림과는 다르게 작동한다. 오디오와 비디오 카메라 드라이버는 isochronous data를 다루는데 드라이버의 좋은 예가 된다.

V. 결 론

리눅스 USB 디바이스 드라이버는 usb-skeleton 드라이버에서 보였듯이 어려운 작업은 아니다. 이 드라이버가 현재 다른 USB 드라이버와 결합하면 최소한의 시간에 작동하는 드라이버를 만들 수 있게 하는 좋은 예가 될 것이다. linux-usb-devel mailing list archives는 또한 많은 유용한 정보를 포함하고 있다.

참고문헌

[1] The Linux USB Project: http://www.linux-usb.org/(http://www.linux-usb.org)

[2] Linux Hotplug Project: http://linux-hotplug.sourceforge.net/(http://linux-hotplug.sourceforge.net)

[3] Linux USB Working Devices List: http://www.qbik.ch/usb/devices/

[4] linux-usb-devel Mailing List Archives: http://marc.theaimsgroup.com/?l=linux-usb-devel

[5] Programming Guide for Linux USB Device Drivers: http://usb.cs.tum.edu/usbdoc

[6] USB Home Page: http://www.usb.org

저자소개

이양원

1982년 중앙대학교 전자공학과(공학사)
1989년 서울대학교 제어계측공학과(공학석사)
1996년 포항공과대학교 전자전기공학과(공학박사)
1982년 : 국방과학연구소 선임연구원
1996년 3월~현재 : 호남대학 정보통신공학과 부교수
※관심분야 : 디지털신호처리, 표적 추적 필터, 인터넷 응용 제어, 임베디드 시스템 등