Writing udev rules
(udev rules 작성 위한 참고문서)

2015. 10. 23
jujakhana@gmail.com

Introduction

이 문서를 읽기 전에..

이 문서는 아래 문서를 번역하여 제공하는 것 입니다. 올바른 해석이 아닐 수도 잘못된 지식이 포함되어 있을 수 있으니 참고해 주시기 바랍니다. 내용 진행 상 불필요한 부분은 제거한 부분도 있으니 원문 또한 참고하면서 봐주시기 바랍니다. 잘못된 내용이나 수정이 필요한 내용이 있다면 jujakhana@gmail.com 로 메일로 해당 내용 보내주시면 감사하겠습니다. 많은 도움이 되었으면 좋겠습니다.

원문 : http://www.reactivated.net/writing_udev_rules.html

이 문서에 대하여..

udev는 Linux Kernel 버전 2.6에 등장하여 그 이상의 버전에서 모두 제공되고 있습니다.
User Space에서 동적으로 /dev 디렉토리의 장치 이름을 영구적으로 제공하기 위한 솔루션입니다. 이전의 Linux Kernel 버전에서는 devfs로 /dev가 구현되어 있었으나, 현재는 효율성과 여러 문제로 인해 사용되지 않고 있으며 udev가 그 자리를 대신하고 있습니다.
몇 년에 걸쳐 udev 규칙을 사용하는 방법이 변경되어 왔습니다. 그 뿐만 아니라 규칙 자체도 유연해졌습니다.

현재 시스템에서 udev는 몇몇 디바이스를 위한 사용자 정의 규칙의 필요성을 제거하고 일부 특별한 장치 유형에 대해 지속적이고 기본적인 네이밍을 제공하고 있습니다. 그러나 일부 사용자들은 여전히 장치들에 대해 사용자 정의를 추가할 수 있도록 요구하고 있는 상황입니다. 말 그대로 사용자가 원하는 규칙에 맞게 디바이스가 관리되고 네이밍 되었으면 좋겠다는 요구사항입니다.

이 문서는 여러분이 udev를 설치하고 기본 설정에 대해 동작하는 것을 확인한 것으로 고려합니다. udev의 기본 설정 동작은 현재 리눅스 배포판에서 자동적으로 이루어져 있습니다. 기본적인 설정이 되어 있다는 것입니다.

여기서는 규칙 작성에 대해 깊고 상세한 내용까지는 다루지 않을 것입니다. 하지만 주된 개념을 모두 소개할 예정입니다. dev 규칙 작성에 대해 더 상세한 내용은 man 페이지를 이용해 주시면 감사하겠습니다.

이제부터 말씀드릴 내용에서 여러분들의 이해를 돕기 위해 많은 예시를 사용할 예정입니다. 이 예시들은 여러분들의 이해를 위해 만들어 진 것이라 말이 되지 않는 부분도 많을 것입니다. 모든 구문이 첨부된 내용을 명확하게 설명하지는 못하지만 udev를 완전히 이해 하기 위해 예제들을 잘 살펴봐 주시기 바랍니다.

The concepts (기본 개념)

Terminology : devfs, sysfs, nodes, etc.

지금부터 설명할 각 용어의 기본 개념은 정확하지 않을 수 있음을 미리 말씀드립니다.

일반적으로 리눅스 시스템에서 /dev 디렉토리는 시스템의 특정 디바이스를 참조 파일과 같은 디바이스 노드 형태로 저장 및 관리하는 데 사용합니다. 이 노드들은 존재하거나 아직 존재하지 않는 시스템의 일부분(디바이스)을 가리키고 있습니다. 사용자 영역 어플리케이션들은 시스템의 하드웨어들을 사용하기 위해 디바이스 노드들을 인터페이스로 사용합니다. 예를 들어 리눅스 시스템에서 디스플레이를 담당하는 X server는 /dev/input mice를 인터페이스로 사용하고 있는데요, 사용자가 사용하는 입력 장치인 마우스의 움직임에 따라 마우스 포인터의 움직임을 X server에서 그려주기 위함입니다. 이렇게 사용자 영역에서 하드웨어에 쉽게 접근하여 사용할 수 있도록 /dev 디렉토리는 디바이스 노드 형태로 사용자에게 제공하는 것입니다.

원래 /dev 디렉토리는 시스템에서 나타낼 수 있는 모든 디바이스로 이루어져 있습니다. 그렇기에 /dev 디렉토리의 크기는 매우 클 수 밖에 없었습니다. devfs는 시스템에 연결되어 있는 하드웨어들을 /dev 디렉토리에서 더 쉽게 관리하게 위한 기능을 제공하였지만, devfs만으로는 쉽게 해결하지 못한 문제들이 있었습니다.

그래서 등장한 것이 **udev** 라는 새로운 /dev 디렉토리 관리 방법입니다. udev는 devfs가 해결하지 못한 이슈들을 깔끔하게 해결하였으며 강력한 기능을 제공하도록 설계되었습니다. 현재 시스템에서 표현되는 디바이스들에 대응하는 /dev 디바이스 노드를 만들거나 네이밍하기 위해서 udev는 사용자에 의해 제공되는 규칙(rules)을 **sysfs**에 의해 제공되는 정보를 이용하여 디바이스 노드와 매칭하는데 사용합니다. 이 문서는 udev 규칙을 어떻게 작성하는지에 대해 알려드리는 것을 목적으로 하고 있습니다. 규칙 작성은 사용자가 할 수 있는 udev와 관계된 task들 중 하나 일 뿐입니다.

sysfs는 kernel 버전 2.6에서 등장한 새로운 파일시스템입니다. 이 파일시스템은 커널에 의해 관리되고 현재 시스템에 연결된 디바이스들의 기본 정보를 관리하는 역할을 합니다. **udev**에서는 이 정보들을 이용하여 현재 연결된 하드웨어에 매칭되는 디바이스 노드를 만들 수 있습니다. sysfs는 /sys 디렉토리에 mount 되어있으며 누구든지 검색 및 확인할 수 있습니다. 여러분은 udev를 이해하기 전 이지만 /sys 디렉토리에 기록된 파일들을 확인할 수 있습니다.

이 문서 전반에 걸쳐 **/sys** 와 **sysfs**란 용어를 섞어 사용할 예정이니 유의하여 읽어주시기 바랍니다.

Why??

udev 규칙은 유연하고 강력한 기능을 제공합니다. 여러분들이 사용할 수 있는 규칙의 몇 가지 기능을 아래 소개합니다.

1. 시스템에 등록된 디바이스의 기본(default) 이름을 재 명명 할 수 있습니다.
2. 기본 디바이스노드에 심볼릭 링크를 만들어 줌으로써 디바이스 노드의 이름을 변경하거나 새롭게 명명할 수 있습니다.
3. 프로그램 출력(?)에 기초하여 장치 이름을 명명할 수 있습니다.
   (무슨 말인지 정확히 저도 이해가 되지 않습니다..)
4. 디바이스 노드의 권한과 소유권을 변경할 수 있습니다.
5. 디바이스 노드가 생성 또는 삭제 될 때 사용자가 작성한 스크립트를 실행할 수 있습니다.(일반적으로 장치가 plug 또는 unplug 될 때)

6. 네트워크 인터페이스들을 재 명명 할 수 있습니다.

udev 규칙을 작성하는 것은 특정 디바이스에 대한 디바이스 노드가 존재하지 않을 때 발생하는 문제의 해결책이 아닙니다. 해당 디바이스가 일치하는 규칙이 존재하지 않더라도 udev는 커널에서 제공하는 기본 이름으로 디바이스 노드를 만들게 됩니다.

우리가 지정한 이름을 지속적으로 가지게 되는 디바이스 노드를 만드는 것은 우리에게 큰 장점이 될 수 있습니다.
예를 들어 봅시다. 여러분들이 USB 저장 장치를 두 개 가지고 있다고 가정합니다. 하나는 디지털 카메라이고 하나는 USB flash disk 입니다. 이 두 장치를 시스템에 연결되게 되면 보통 /dev/sda , /dev/sdb란 이름으로 디바이스 노드가 생성됩니다. 하지만, 이때 명명되는 이름은 '시스템에 어떤 장치부터 연결되었나'에 의존 되기에 우리가 원하는 장치를 확인하기 쉽지 않습니다. 사용자는 바로 어떤 장치가 어떤 디바이스 노드로 연결되어 있는지 알고 싶지만 그렇지 못한다는 문제가 있습니다. 만약 각 디바이스 노드가 연결되는 순서와 상관없이 언제 어떤 순서로 연결되던간에 /dev/camera와 /dev/flashdisk라는 이름의 디바이스 노드가 만들어진다면 정말 편하게 사용할 수 있을겁니다.

Built-in persistent naming schemes

udev는 특별한 타입의 장치에게 영구적인 이름을 명명할 수 있다고 위에서 계속 설명드렸습니다. 이 기능은 매우 유용하며 여러분이 만약 규칙을 작성할 필요가 없이 udev의 개념만 알고 싶고 추가적인 내용이 필요하지 않다면 더 이상 이 문서를 읽지 않으셔도 됩니다.

udev는 /dev/disk 디렉토리 내의 저장 장치들을 위해 특별한 이름을 제공합니다. 당신의 저장 장치에 대해 생성된 이름들을 확인하기 위해서 아래 명령을 따라 쳐보세요.

```
# ls -lR /dev/disk
```

이 명령어는 모든 저장장치 타입에 적용됩니다.

Rule writing

Rule files and semantics

시스템에 장치가 연결되어서 추가적인 동작을 수행하거나 이름을 어떻게 정해야 할지 결정해야 할 때, udev는 규칙 파일을 읽어 들여 실행합니다. 이 파일들은 /etc/udev/rules.d 디렉토리 아래 저장되어 있으며 해당 파일들은 모두 .rules 이란 접미사가 붙어있습니다.

```
pi@raspberrypi /etc/udev/rules.d $ ls
40-scratch.rules  99-com.rules
```

위 사진처럼 /etc/udev/rules.d 디렉토리 아래 접미사 **.rules**이 붙은 파일들이 존재함을 알 수 있습니다. 위 파일들은 라즈베리 파이에 포팅된 OS인 라즈비안에서 확인하였습니다.

/etc/udev/rules.d/ 디렉토리 내의 파일, 즉 규칙 파일들을 어휘 순대로 실행됩니다. 이러한 실행순서는 어떤 환경에서는 매우 중요합니다. 일반적으로 여러분들은 리눅스 커널에서 제공하는 기본 udev

규칙보다 여러분이 작성 규칙이 먼저 실행되길 바랄 것입니다. 그런 경우에 파일 이름의 어휘 순대로 실행하기에 아래 예시와 같이 규칙 파일명을 생성하고 규칙을 해당 파일에 작성하는 것이 좋습니다.

/etc/udev/rules.d/10-local.rules

.rules파일 안에서 '#'로 시작하는 라인은 주석으로 취급되며 그 외 모든 라인은 규칙으로 판단됩니다. 여기서 한 가지 주의 할 것은 하나의 파일 내의 여러 규칙을 작성할 수 있으나 하나의 규칙은 무조건 한 라인에 작성해야 한다는 것입니다. 규칙이 길다고 개행(줄 바꿈)을 하는 것이 아니라 쭉 연결해서 작성해야만 하나의 규칙으로 시스템이 인식하고 실행할 것 입니다.

여기서 한 가지 의문점이 있을 수 있는데요, 만약 하나의 장치에 여러 규칙이 동시에 적용 된다면 어떻게 될까요? 가장 처음 적용되는 규칙만 실행하고 끝이 날까요?? **udev의 장점 중 하나는 하나 이상의 rule들이 하나의 디바이스에 모두 적용될 수 있다는 것입니다.** 이것은 udev의 매우 큰 장점 중의 하나입니다! 각 디바이스들은 각각 다른 이름들로 파일들이 생성이 될 것 입니다. 이 말인 즉 udev는 디바이스와 규칙이 중간에 매칭되더라도 거기서 해당 작업이 종료하지 않고 끝까지 규칙들을 매칭해 나간다는 것을 의미합니다. 그렇기에 모든 규칙에 대해 여러 개 적용되는 디바이스는 그에 맞게 여러 규칙이 적용됨을 알 수 있습니다.

Rule syntax

각 규칙은 ','로 분리된 key-value의 집합으로 이루어져 있습니다.

```
pi@raspberrypi /etc/udev/rules.d $ cat 40-scratch.rules
ATTRS{idVendor}=="0694", ATTRS{idProduct}=="0003", SUBSYSTEMS=="usb", ACTION=="add", MODE="0666", GROUP=
"plugdev"
```

match key는 규칙을 적용하기 위한 디바이스를 식별하기 위한 조건입니다. 규칙의 match key가 우리가 사용하려는 디바이스와 일치한다면 규칙이 적용되며 assignment key의 동작이 호출됩니다. 모든 규칙은 최소 하나의 match key와 assignment key를 가지고 있습니다.

아래 예시를 보겠습니다.

```
KERNEL=="hdb", NAME="my_spare_disk"
```

위의 규칙을 살펴보면 하나의 match key(KERNEL)과 하나의 assignment key(NAME)을 포함하고 있는 것을 알 수 있습니다. 이 키들은 각각 의미와 특성을 가지고 있지만 자세한 설명은 나중에 하겠습니다. 여기서 중요하게 보실 것은 match key의 값은 **'=='** 연산자와 연관이 있고 assignment key의 값은 **'='** 연산자와 관계 있다는 것만 주의해 주시면 됩니다.

절대 하나의 규칙을 작성할 때 여러 줄로 작성해서는 안됩니다. 만약 규칙 작성 중간에 개행된다면 여러분들의 규칙은 하나가 아닌 여러 개의 규칙으로 인식될 것이고 사용자가 원하는 동작을 하지 않을 것 입니다. 다시 한번 말씀드리지만, 절대 여러 줄로 하나의 규칙을 작성하시면 안됩니다.

Basic Rules

udev는 매우 정확하게 장치와 매칭되는 규칙을 작성하는데 사용할 수 있는 여러 match key를 제공합니다. 여러 key 중 가장 흔히 사용되는 key들을 아래 소개 할텐데요, 그 외 다른 key는 이 문서 뒤쪽에 설명할 예정이며 전체 key 리스트를 보고 싶으시다면 udev man 페이지에서 확인하시기 바랍니다.

- **KERNEL** - 디바이스가 커널 이름에 대해 일치하는지 확인
- **SUBSYSTEM** - 디바이스의 서브 시스템에 대해 일치하는지 확인
- **DRIVER** - 디바이스를 백업 드라이버의 이름에 대해 일치하는지 확인

여러분이 디바이스와 정확히 매치되는 일련의 match key를 사용한 후, udev는 다양한 assignment key들을 통해 다음에 실행할 동작을 여러분이 제어할 수 있게 해줍니다. 여러분이 사용 가능한 전체 assignment key 리스트는 man 페이지를 참고해 주시기 바랍니다. 아래는 가장 기본적으로 자주 사용되는 assignment key들을 소개하고 그 외의 assignment key들은 문서 후반부에서 설명 드리겠습니다.

- **NAME** - 디바이스 노드에 사용하려는 이름
- **SYMLINK** - 디바이스 노드를 대체하여 동작할 수 있는 심볼릭 링크 리스트
-

위 설명처럼 udev는 오직 하나의 디바이스를 위한 하나의 디바이스 노드를 생성합니다. 만약 여러분이 디바이스 노드에 대해 대체할 수 있는 이름을 사용하고 싶다면 심볼릭 링크 기능을 이용하면 됩니다. 여러분은 실제로 **SYMLINK assignment key**를 사용하여 심볼릭 링크를 생성 할 수 있습니다. 이 모든 것들이 실제 디바이스와 연결되어 있습니다. 위와 같은 링크들을 다루기 위해 우리는 새로운 operator(연산자)를 이용하여 심볼릭 링크 목록에 추가합니다. 그 operator는 '**+=**'입니다. 여러분은 생성할 심볼릭 링크를 공백으로 구분함으로써 하나의 규칙에 여러 개의 심볼릭 링크를 생성할 수 있습니다.
그럼 아래 예시들을 보시죠.

```
KERNEL=="hdb", NAME="my_spare_disk"
```

위의 규칙에 대해 설명하자면 "hdb"로 커널에 의해 명명된 디바이스를 매치하고 "hdb"를 호출하는 "my_spare_disk"라는 이름의 디바이스 노드를 만들어 매치시킵니다. 그리하여 디바이스 노드는 /dev/my_spare_disk로 나타내게 됩니다.

```
KERNEL=="hdb", DRIVER=="ide-disk", SYMLINK+="sparedisk"
```

다음으로 위의 예시를 보겠습니다. 위의 규칙은 커널에 의해 "hdb"란 이름으로 명명되어 있고 드라이버가 "ide-disk"인 디바이스를 매치합니다. 해당 디바이스는 NAME key가 없으므로 커널에 의해 명명된 기본 이름을 가지며 디바이스 노드는 "sparedisk"란 이름으로 심볼릭 링크가 생성됩니다. 우리가 디바이스 노드의 이름을 지정하지 않는다면 udev는 default name을 사용하게 됩니다. 표준 /dev 레이아웃을 유지하기 위해 여러분들의 규칙들은 일반적으로 NAME을 사용하지 않고 SYMLINK를 사용하여 심볼릭 링크를 생성하고 다른 작업을 수행합니다.

```
KERNEL=="hdc", SYMLINK+="cdrom cdrom0"
```

위의 rule은 아마도 여러분이 작성하려는 규칙의 일반적인 형태일 것입니다. 위의 규칙은 "/dev/cdrom"과 "/dev/cdrom0"란 두 개의 심볼릭 링크를 생성할텐데요, 두 링크 모두 "/dev/hdc"를 연결하고 있습니다. 위처럼 NAME key가 지정되지 않았다면 default kernel name(hdc)으로 지정됩니다.

Matching sysfs attributes

지금까지 위에서 소개했던 match key는 제한된 기능만 제공했습니다. 하지만 우리는 훨씬 더 많은 기능(제어)를 필요로 하는데요, 예를 들어 공급 업체코드, 제품 번호, 일련 번호, 저장 용량, 파티션 수 와 같은 고급 속성을 이용하여 디바이스를 확인 및 판별하기 원합니다.

많은 드라이버들은 sysfs로 정보를 보내고 udev는 약간 다른 구문인 ATTR key를 이용해 우리가 작성한 규칙들을 sysfs와 매칭시킵니다.

여기 sysfs로부터 단일 속성을 매칭하는 규칙 예제가 있습니다. 추가적으로 sysfs 속성을 기반으로 한 규칙을 작성하는 자세한 내용은 문서 뒤 쪽에서 설명 드리겠습니다.

```
SUBSYSTEM=="block", ATTR{size}=="234441648", SYMLINK+="my_disk"
```

Device hierarchy (디바이스 체계/구조)

리눅스 커널은 실제로 디바이스들을 트리 형태의 구조로 표현하고 이 정보들을 sysfs을 통해 나타내며 규칙을 작성할 때 매우 유용합니다. 예를 들어 내 하드 디스크 디바이스의 표현은 차례로 PCI 버스 디바이스의 직렬 ATA 컨트롤러 디바이스의 SCSI 디스크 디바이스의 자식 디바이스입니다. 위처럼 여러분은 문제의 디바이스의 부모로부터 정보를 확인해 볼 수 있습니다. 예를 들어 저의 하드디스크의 시리얼 번호는 디바이스 레벨에서는 확인할 수 없습니다. 하지만 부모 디바이스인 SCSI 디스크 레벨에서는 확인 할 수 있습니다.

지금까지 여러분들이 사용하려는 해당 디바이스와는 일치하지만 부모 디바이스의 속성 값과는 일치하지 않는 4개의 메인 match key(KERNEL/SUBSYSTEM/DRIVER/ATTR)들을 소개했습니다. udev는 디바이스 트리 구조에서 상위 레벨을 검색할 수 있는(parent) 다양한 match key를 제공합니다. (자세히 보시면 위의 match key와는 다른 key들입니다.)

- **KERNELS** - 상위 디바이스 중 임의의 장치에 대한 커널 이름 또는 디바이스의 커널 장치 이름 매치
- **SUBSYSTEMS** - 해당 디바이스 또는 상위 디바이스의 임의의 서브시스템과 매치
- **DRIVERS** - 디바이스를 지원하는 드라이버 이름 또는 상위 디바이스 중의 드라이버 이름과 매치
- **ATTRS** - 디바이스의 sysfs 속성 또는 상위 디바이스 중 임의의 디바이스 sysfs 속성과 매치

트리 형태의 디바이스 구조를 고려하면 규칙을 작성하는 것이 조금은 복잡하게 느껴질 수 있습니다. 여러분들을 도와줄 툴(Tool)들이 있으니 안심하시고 그 내용은 추후에 설명 드리겠습니다.

String substitutions (String 치환 / 변경)

여러분이 잠재적으로 유사한 많은 디바이스들을 다루는 규칙을 작성할 때, udev의 문자열 치환 연산자들은 매우 유용합니다. 여러분들은 단순히 규칙을 만들어 어떤 assignment에 이러한 연산자들을 포함할 수 있으며 udev는 그 assignment가 실행될 때 그것들을 판단(평가)할 것입니다(?).

가장 흔하게 사용되는 연산자들은 '**%k'** 와 '**%n'** 입니다. 먼저 %k는 디바이스의 커널 이름을 판단합니다. 예를 들어 /dev/sda3에 나타나듯 "**sda3**"를 의미합니다. 그리고 %n은 디바이스의 커널 넘버를 판단합니다. 이것은 저장 장치의 파티션 넘버 같은 것인데요, 예를 들어 /dev/sda3의 "**3**"을 의미하죠.

또한 udev는 추가적인 기능의 치환 연산자들을 제공하는데요, 이 문서의 남은 부분을 읽은 후 udev man 페이지를 참조하세요.
위에서 말한 %k와 %n 연산자를 대체할 수 있는 또 다른 구문이 있습니다. 바로 **$kernel** 과 **$number**인데요, 그럼 여기서 '$'와 '%'는 연산자이기에 문자 자체로 사용하지 못한다는 건가요? 이런 이유로 여러분이 규칙 내에서 문자 그대로 '$'를 사용하려면 **'$$'**를 사용해야 하며 '%'를 사용하려면 **'%%'**를 사용하면 됩니다. 즉, 규칙 작성에 필요한 이스케이프 시퀀스들입니다.

string 치환의 개념을 설명하기 위해 아래 예제를 살펴보겠습니다.

```
KERNEL=="mice", NAME="input/%k"
KERNEL=="loop0", NAME="loop/%n", SYMLINK+="%k"
```

첫 번째 규칙은 mice 디바이스 노드(기본적으로 /dev/mice 인)를 /dev/input 디렉토리에 표시되게 합니다. 두 번째 규칙은 loop0란 이름의 디바이스 노드를 /dev/loop/0에 만들고 추가적으로 /dev/loop0에 심볼릭 링크 또한 만들어줍니다.

위 규칙들을 보면 의심스러운 부분이 있습니다. string 치환의 개념을 설명하기 위한 예제인데 실제 치환 연산자들을 사용하지 않고 모든 규칙을 작성할 수 있는 예제들 입니다.
진정한 치환 연산자의 힘은 다음 섹션에서 보여드리겠습니다.

String matching

정확하게 문자열과 일치할 수 있도록 udev는 쉘 스타일의 패턴 매칭을 사용할 수 있습니다. 아래 udev에서 지원하는 3가지 패턴에 설명드립니다.

- ***** - 0회 이상의 어떤 문자라도 매치
- **?** - 정확히 하나의 어떤 문자라도 매치
- **[]** - 대괄호로 지정된 임의의 문자 또는 허용 범위의 문자열 매치

아래는 위의 패턴들을 포함하고 있는 예제들입니다. 문자열 치환 연산자를 어떻게 사용하고 있는지 유의해서 살펴봐주세요.

```
KERNEL=="fd[0-9]*", NAME="floppy/%n", SYMLINK+="%k"
KERNEL=="hiddev*", NAME="usb/%k"
```

첫 번째 규칙은 floppy 디스크 드라이브들과 일치하는 디바이스 노드들에 관한 것입니다. default name으로 된 심볼릭 링크가 /dev/floppy 디렉토리에 생성되는 규칙입니다.

두 번째 규칙은 hiddev란 접두사 뒤에 어떤 문자 또는 문자열로 이뤄진 이름의 장치는 /dev/usb 디렉토리에만 존재하는 것을 보장하는 규칙입니다.

Finding information from sysfs

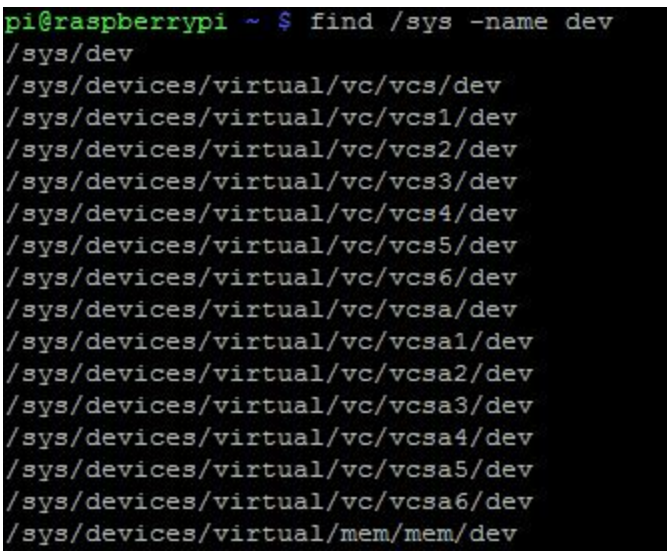
The sysfs tree

sysfs로부터 우리가 관심있는 정보를 사용하기 위한 방법은 위에서 간단히 다루었습니다. 위에서 다루었던 내용을 기반으로 규칙을 작성하기 위해서 attributes(속성)의 이름과 그들의 현재 value(값/정보)들에 대해 알아야 합니다.

sysfs는 매우 단순한 구조입니다. 논리적인 디렉토리 형태로 나뉘어져 있죠. 각 디렉토리는 하나의 정보를 포함하는 수많은 파일(속성)들을 포함합니다. 몇몇 심볼릭 링크들은 부모에 디바이스를 연결합니다.

일부 디렉토리는 최상위 디바이스 경로들입니다. 이 디렉토리들은 디바이스 노드들에 해당하는 실제 디바이스들을 나타냅니다. 최상위 디바이스 경로들은 dev 파일을 포함하고 있는 sysfs 디렉토리들로 구분됩니다. 그 목록들을 확인 하려면 아래 명령어를 실행하시면 됩니다.

```
# find /sys -name dev
```



예를 들어, 나의 시스템에서 /sys/block/sda 디렉토리는 나의 하드디스크에 대한 디바이스 경로입니다. 이것은 부모 디바이스인 SCSI 디스크 디바이스와 심볼릭 링크인 /sys/block/sda/device로 연결되어있습니다.

여러분이 sysfs 정보를 기반으로 규칙을 작성할 때, 단순하게 체인(트리구조에서 연결된 형태)의 한 부분의 어느 파일들의 속성 내용들과 매치만 하면 된다. 그 말인 즉 디바이스 정보 파일 내의 디바이스의 속성을 맞춰 규칙을 작성하면 됩니다. 예를 들어 내 하드디스크의 사이즈를 알고 싶다면 아래와 같이 명령어를 치면 됩니다. (체인이란 말이 디바이스에 연결된 정보들의 파일들을 의미하는 것 같습니다. 이해가 안되시면 원문을 참고해 보세요.)

```
# cat /sys/block/sda/size
234441648
```

udev 규칙에서 디스크 사이즈인 “234441648”을 확인하기 위해 ATTR{size}를 사용할 수 있습니다. udev는 전체 디바이스 체인을 통해 반복하며 또 다른 체인 부분의 속성들을 ATTR 속성을 사용하여 매치할 수 있습니다.(ex. /sys/class/block/sda/device/의 속성들)
그러나 이 후에 설명할 다른 체인 부분을 다룰 때 여러분들이 따라야 할 몇 개의 특정 절차가 있습니다.

비록 이것이 sysfs의 구조의 유용한 안내서 역할을 하고 정확히 udev가 값들을 매치 하더라도 sysfs를 통해 수동적으로 확인하는 것은 시간 소모가 크고 불필요한 일이 될 것입니다.

udevinfo

여러분들이 규칙을 만드는데 가장 쉽게 사용할 수 있는 툴(Tool)이 **udevinfo**일 것 입니다. 이것을 사용하는데 여러분들이 알아야 하는 것은 우리가 알고 싶어하는 장치의 sysfs 디바이스 경로 입니다. 해당 경로만 안다면 쉽게 툴을 사용할 수 있습니다. 아래는 사용 예시입니다.

```
# udevinfo -a -p /sys/block/sda

  looking at device '/block/sda':
    KERNEL=="sda"
    SUBSYSTEM=="block"
     ATTR{stat}==" 128535     2246  2788977   766188    73998   317300  3132216   5735004       0
516516  6503316"
    ATTR{size}=="234441648"
    ATTR{removable}=="0"
    ATTR{range}=="16"
    ATTR{dev}=="8:0"

  looking at parent device '/devices/pci0000:00/0000:00:07.0/host0/target0:0:0/0:0:0:0':
    KERNELS=="0:0:0:0"
    SUBSYSTEMS=="scsi"
    DRIVERS=="sd"
    ATTRS{ioerr_cnt}=="0x0"
    ATTRS{iodone_cnt}=="0x31737"
    ATTRS{iorequest_cnt}=="0x31737"
    ATTRS{iocounterbits}=="32"
    ATTRS{timeout}=="30"
    ATTRS{state}=="running"
    ATTRS{rev}=="3.42"
    ATTRS{model}=="ST3120827AS     "
    ATTRS{vendor}=="ATA     "
    ATTRS{scsi_level}=="6"
    ATTRS{type}=="0"
    ATTRS{queue_type}=="none"
```

```
  ATTRS{queue_depth}=="1"
  ATTRS{device_blocked}=="0"

looking at parent device '/devices/pci0000:00/0000:00:07.0':
  KERNELS=="0000:00:07.0"
  SUBSYSTEMS=="pci"
  DRIVERS=="sata_nv"
  ATTRS{vendor}=="0x10de"
  ATTRS{device}=="0x037f"
```

위 내용을 보면 알 수 있듯이, **udevinfo**는 단순히 여러분들이 규칙을 작성할 때 있는 그대로 match key들로 사용할 수 있는 속성들을 나열해 줍니다. 위 예시로부터 두 개 정도의 예제 규칙을 만들어 보겠습니다.

```
SUBSYSTEM=="block", ATTR{size}=="234441648", NAME="my_hard_disk"
SUBSYSTEM=="block", SUBSYSTEMS=="scsi", ATTRS{model}=="ST3120827AS",
NAME="my_hard_disk"
```

위 예제 중 파란 글을 주의 깊게 봐주세요. 임의의 장치로부터 속성(attribute)들을 결합한 올바른 문법의 규칙이지만 부모가 오직 하나, 단일 디바이스 일 때만 해당됩니다. 만약 부모 디바이스가 여러 개라면 대한 속성에는 사용할 수 없고 여러분의 규칙은 제대로 동작하지 않을 수 있습니다.

아래 규칙을 보면 두 개의 부모 디바이스로부터는 속성(attribute)을 매치 하지 못합니다.

```
SUBSYSTEM=="block", ATTRS{model}=="ST3120827AS", DRIVERS=="sata_nv",
NAME="my_hard_disk"
```

여러분들은 많은 속성(attribute)을 사용할 수 있으며 그 중 몇 가지를 선택하여 규칙을 작성해야 합니다. 많은 사람들이 규칙을 작성할 때 지속적이고 인간이 인식 할 수 있는 방식으로 디바이스를 식별하는 속성들을 선택합니다. 위에 예제에서는 내 디스크의 크기와 모델번호를 속성으로 선택하였습니다. ATTRS{iodone}==“0x31373”과 같은 의미없는 속성을 사용하지는 않았습니다.

udevinfo 출력을 확인해 보세요. 임의의 장치에 대해 녹색 부분은 KERNEL과 ATTR 같은 표준 match key들입니다. 파랑과 빨강 부분은 SUBSYSTEMS와 ATTRS와 같은 속성들로 부모 디바이스와 연결되어 사용될 수 있습니다. udevinfo의 출력, 즉 계층적인 구조의 복잡함은 udevinfo에서 제공하는 값들을 정확하게 다룰 줄 안다면 매우 쉽게 사용할 수 있습니다.

주목해야 할 또 다른 점은(udevinfo 출력리스트 중 ST31208027AS(굵은 글씨) 참조) udevinfo 출력에서 나타난 속성들을 표시하는 텍스트 중 공백으로 이뤄진 부분입니다. 여러분들의 규칙에서 여분의 공간을 지정하거나 아니면 바로 위의 예시처럼 공백을 없앤 후 작성할 수 있습니다.

udevinfo를 사용할 때 유일한 문제는 여러분들이 알고자 하는 장치의 최상위 디바이스 경로를 알아야 한다는 것입니다.(ex. /sys/block/sda) 하지만 여러분들은 이미 존재하는 디바이스 노드에 대해 규칙을 작성하는거나 해당 디바이스의 경로를 찾기 위해서도 udevinfo를 사용할 수 있습니다.

```
# udevinfo -a -p $(udevinfo -q path -n /dev/sda)
```

Alternative methods

udevinfo가 여러분이 작성하려는 규칙의 속성을 확인하기 위해 사용되는 가장 흔히 사용되는 방법이긴 하지만, 모든 사용자를 만족 시킬 수는 없습니다. 그래서 또 다른 툴의 도움을 받을 수 있으며 대표적인 툴이 usbview입니다. usbview와 같은 tool을 이용하면 규칙을 만들기 위한 수많은 정보들을 얻을 수 있습니다.

Advanced topics

Controlling permissions and ownership

udev는 여러분들이 작성하는 규칙에 추가적인 할당 키(assignment key)를 사용하여 각 디바이스의 권한과 소유권을 제어할 수 있도록 해줍니다.

GROUP 할당(assignment)은 여러분이 Unix 그룹의 디바이스 노드 소유권을 정의 할 수 있도록 해줍니다. 아래 예제 규칙을 보면 video 그룹이 framebuffer 디바이스 노드들을 소유하도록 정의해주는 것을 확인할 수 있습니다.

```
KERNEL=="fb[0-9]*", NAME="fb/%n", SYMLINK+="%k", GROUP="video"
```

OWNER key(assignment 중 하나)는 여러분들이 Unix 사용자가 어떤 디바이스 노드에 대해 소유권을 가지게 정의하는데 사용하기에 그렇게 유용하게 사용되진 않습니다. 여러분은 아래처럼 “john”이란 소유자가 여러분의 floppy 디바이스들을 소유하길 원하는 이상한 상황을 원한다면 아래와 같이 사용하시면 됩니다.

```
KERNEL=="fd[0-9]*", OWNER="john"
```

udev는 0660(소유자와 그룹이 readable/writable 권한을 가지는)과 같은 Unix 권한을 가진 노드들을 생성해주진 않습니다. 만약 위처럼 권한을 기본으로 설정해주고 싶다면 MODE assignment를 규칙에 포함하시면 됩니다. 예를 들어 “**inotify**” 노드가 모든 사용자에게 읽고 쓸 수 있는 권한을 부여하고 싶다면 아래와 같이 규칙을 정의하면 됩니다.

```
KERNEL=="inotify", NAME="misc/%k", SYMLINK+="%k", MODE="0666"
```

*inotify : 파일 시스템 이벤트를 통보해주는 기능을 가진 리눅스 커널 서브시스템 중 하나

Using external programs to name devices

어떤 경우에는 표준 udev 규칙들에서 제공하는 것보다 더 많은 것을 필요로 할 수 있습니다. 이럴 때 여러분들은 udev를 사용하여 다른 프로그램을 실행하거나 디바이스의 이름을 정하기 위한 프로그램의 표준 출력을 사용할 수도 있습니다. 이 기능을 사용하기 위해선 규칙 속성 중 PROGRAM assignment에 실행 할 프로그램의 절대 경로를 명시하거나 NAME/SYMLINK assignment에 %c 치환을 조금 변형하여 사용할 수 있습니다.

아래 예제는 "/bin/device_namer" 이라는 가상의 프로그램을 언급하고 있습니다. device_namer는 디바이스에 대한 커널의 이름을 하나의 command line 인자로 사용합니다. 이 커널 이름을 기반으로 device_namer는 여러 부분으로 나뉘어 표준 출력 파이프로 출력을 내보내는 일을 합니다. 각 부분은 하나의 단어이고 각 파트들은 공백으로 나뉘어집니다.

첫 번째 예제에서 우리는 device_namer의 출력들을 이용해 임의의 디바이스들의 심볼릭 링크를 만들기 위한 것입니다.

```
KERNEL=="hda", PROGRAM="/bin/device_namer %k", SYMLINK+="%c"
```

다음 예제는 device_namer의 출력을 두 부분으로 나눈 것 입니다. 첫 째는 디바이스 네임이고, 두 번째 부분은 추가적인 심볼릭 링크입니다. %c{N} 치환법을 이용하여 N개의 출력이 가능함을 볼 수 있습니다.

```
KERNEL=="hda", PROGRAM="/bin/device_namer %k", NAME="%c{1}", SYMLINK+="%c{2}"
```

다음 예제는 device_namer의 출력이 디바이스 이름에 대해 추가적인 심볼릭 링크를 형성 할 부분의 수가 계속되는 예제입니다. 현재 출력이 끝날 때 까지 %c{N+} 치환법에 의해 N개의 부분으로 형성된다.

```
KERNEL=="hda", PROGRAM="/bin/device_namer %k", NAME="%c{1}", SYMLINK+="%c{2+}"
```

출력 부분은 NAME, SYMLINK 뿐만 아니라 다른 assignment key를 사용하여도 출력할 수 있습니다. 다음 예제는 디바이스를 소유하고 있는 Unix 그룹을 확인하는 가상의 프로그램을 사용하기 위한 예제입니다.

```
KERNEL=="hda", PROGRAM="/bin/who_owns_device %k", GROUP="%c"
```

Running external programs upon certain events

udev 규칙들을 작성하는 또 다른 이유는 디바이스가 연결 또는 연결 해제 되었을 때 특정 프로그램을 실행할 수 있기 때문입니다. 예를 들어 여러분들의 디지털 카메라가 연결되었을 때 자동적으로 모든 사진들을 다운받는 스크립트를 실행할 수 있습니다.

위에서 설명한 PROGRAM과 혼동하시면 안됩니다. PROGRAM은 디바이스 이름을 명명하기 위해 실행되는 프로그램이며 그 외에는 아무런 기능을 가지고 있지 않습니다. 프로그램들이 실행될 때 디바이스 노드가 아직 만들어지지 않았다면 어떤 상황이던지 프로그램을 실행할 수 없습니다.

여기서 소개하는 기능은 항상 디바이스 노드가 생성된 이후 프로그램을 실행해야 합니다. 이 프로그램은 디바이스에서 동작할 수 있으나, 프로그램이 실행되는 동안 udev가 효율을 위해 프로그램을 정지 시키기 때문에 장시간 실행은 되지 않습니다. 이 제한사항의 한 가지 해결책은 여러분의 프로그램이 즉시 분리(? 원문에는 detach라고 되었습니다.)되는지 확인하는 것입니다.

아래 예제는 규칙에 RUN assignment를 사용하는 방법을 보여줍니다.

```
KERNEL=="sdb", RUN+="/usr/bin/my_program"
```

/usr/bin/my_program이 실행 되었을 때, udev 환경의 다양한 부분들이 SUBSYSTEM처럼 key value를 포함하고 있는 환경 변수로써 사용 할 수 있습니다. 또한 여러분들은 ACTION 환경 변수를 이용하여 디바이스가 연결되었는지 연결 해제되었는지 확인할 수 있습니다. ACTION 환경 은 “추가” 또는 “제거” value를 가지기 때문입니다.

udev는 프로그램들을 활성화된 terminal과 쉘 컨텍스트에서는 실행하지 않습니다. 하지만 만약 프로그램이 쉘 스크립트이고 shebang(e.g #!/bin/sh)으로 시작된다면 그 프로그램은 실행될 수 있습니다. 하지만 터미널에는 어떠한 표준 출력도 표시되지 않을 것 입니다.

(*shebang : 스크립트의 가장 윗 줄에 이 스크립트를 실행할 인터프리터와 실행 옵션을 작성하게 되는데 그 가장앞에 붙는 #! 를 shebang이라고 합니다.)

Environment interaction

udev는 matching 과 assignment에 사용할 수 있는 환경 변수인 ENV key를 제공합니다.

할당(assignment)의 경우, 여러분은 환경 변수를 나중에 매칭할 수 있도록 세팅할 수 있습니다. 또한 위에서 언급된 기술들을 사용, 적용된 외부 프로그램들을 사용할 수 있도록 환경 변수를 세팅할 수 있습니다. 아래 예제는 환경 변수를 설정하는 예제입니다.

```
KERNEL=="fd0", SYMLINK+="floppy", ENV{some_var}="value"
```

매칭(matching)의 경우, 오직 환경 변수의 값에 의해 동작하도록 규칙을 적용할 수 있습니다.
udev는 여러분들이 콘솔에서 얻을 수 있는 환경과 같은 사용자 환경과 동일하지 않음을 유의해야 합니다.
아래 예제는 환경 매칭에 관한 예제입니다.

```
KERNEL=="fd0", ENV{an_env_var}=="yes", SYMLINK+="floppy"
```

위 규칙은 오직 udev의 환경변수 “$an_env_var”이 “yes”로 세팅되어있을때만 /dev/floppy 링크를 만듭니다.

Additional options

여러분에게 유용할 수 있는 또 다른 assignment OPTION 목록입니다.

- **all_partitions** - 초기에 발견된 것 뿐만 아니라 block 디바이스의 가능한 모든 파티션들을 만듭니다.
- **ignore_device** - 모든 이벤트를 무시합니다.
- **last_rule** - 더 이상 규칙의 어떤 영향도 무시합니다.

예를 들어, 아래 규칙은 하드디스크 노드의 그룹 소유권을 세팅하고 더 이상 어떤 규칙도 영향을 끼치지 못하게 설정하는 예제 입니다.

```
KERNEL=="sda", GROUP="disk", OPTIONS+="last_rule"
```

Examples

USB Printer

프린터 전원을 켜고 /dev/lp0란 디바이스 노드에 디바이스를 할당했습니다. 하지만 이 단조롭고 특징도 없는 이름이 마음에 들지 않습니다. 그래서 여러분은 대체 이름을 제공하는 규칙을 작성하기 위해 udevinfo를 사용하기로 결정하였습니다.

```
# udevinfo -a -p $(udevinfo -q path -n /dev/lp0)
  looking at device '/class/usb/lp0':
    KERNEL=="lp0"
    SUBSYSTEM=="usb"
    DRIVER==""
    ATTR{dev}=="180:0"

  looking at parent device '/devices/pci0000:00/0000:00:1d.0/usb1/1-1':
    SUBSYSTEMS=="usb"
    ATTRS{manufacturer}=="EPSON"
    ATTRS{product}=="USB Printer"
    ATTRS{serial}=="L72010011070626380"
```

위 내용을 보고 규칙을 작성하였습니다.

```
SUBSYSTEM=="usb", ATTRS{serial}=="L72010011070626380", SYMLINK+="epson_680"
```

USB Camera

대부분의 카메라는 외장 하드디스크로 SCSI 방식의 USB 버스로 연결됩니다. 저장된 내 사진을 보기 위해 drive를 마운트하고 사진을 나의 하드디스크로 옮깁니다.

하지만, 모든 카메라들이 위와 같은 방식으로 작동하지는 않습니다. gphoto2가 지원되는 카메라 같은 경우 non-storage 프로토콜을 사용하기 때문에 이 경우 여러분은 규칙을 작성할 수가 없습니다. gphoto같은 경우 특정 커널 드라이버가 아닌 사용자 영역에서 제어해야 합니다.

보통의 USB 카메라의 문제는 /dev/sdb, /dev/sdb1과 같이 카메라 스스로 단일 파티션을 나누고 식별되는 것인데요. sdb 노드는 여러분들에게 쓸모없는 노드지만, sdb1는 여러분들이 마운트 할 수 있기 때문에 sdb1만 관심을 가지면 됩니다. 하지만, 여기서 문제는 sysfs로 연결되어 있다는 것인데요, devinfo로 확인할 수 있는 /dev/sdb1의 유용한 속성(attribute)들이 /dev/sdb와 동일하다는 것입니다. raw disk와 파티션 둘 다 여러분들의 규칙에 매칭될 수 있기 때문에 여러분들이 원하는 것을 얻을 수 없습니다. 그렇기에 여러분들의 규칙은 특별해야 합니다.

이 문제를 해결하려면 여러분은 sdb와 sdb1 사이에 무엇이 다른지 생각해 볼 필요가 있습니다. 매우 단순하게 해결될 수 있는 문제이죠. 자세히 볼 필요도 없이 두 가지는 이름 부터 다릅니다!. 그렇기 때문에 우리는 NAME의 단순한 패턴 매칭만으로도 문제를 해결 할 수 있습니다.

```
# udevinfo -a -p $(udevinfo -q path -n /dev/sdb1)
  looking at device '/block/sdb/sdb1':
    KERNEL=="sdb1"
    SUBSYSTEM=="block"

                                looking at parent device
'/devices/pci0000:00/0000:00:02.1/usb1/1-1/1-1:1.0/host6/target6:0:0/6:0:0:0':
    KERNELS=="6:0:0:0"
    SUBSYSTEMS=="scsi"
    DRIVERS=="sd"
    ATTRS{rev}=="1.00"
    ATTRS{model}=="X250,D560Z,C350Z"
    ATTRS{vendor}=="OLYMPUS "
    ATTRS{scsi_level}=="3"
    ATTRS{type}=="0"
```

위 내용을 기반으로 규칙을 작성해 봅시다.

```
KERNEL=="sd?1", SUBSYSTEMS=="scsi", ATTRS{model}=="X250,D560Z,C350Z"
SYMLINK+="camera"
```

USB Hard Disk

USB 하드 디스크는 위에서 설명한 USB 카메라와 매우 유사합니다. 하지만 일반적인 문법 패턴들이 다릅니다. 카메라 예제에서 sdb 노드는 관심이 없는 노드라고 설명드렸을 텐데요,
이것은 실제 파티셔닝을 위해 사용되어 집니다.(예를 들어 fdisk) 하지만 누가 자신의 카메라 파티션을 나누길 바라겠습니까?

물론, 만약 여러분이 100GB짜리 USB 하드디스크를 가지고 있다면 여러분이 파티션을 나누고 싶다는 것을 이해할 수 있습니다. 만약 정말 100GB짜리 하드 디스크가 파티션이 나뉘어져 있는 경우 아래와 같이 udev의 문자열 치환을 이용할 수 있습니다.

```
KERNEL=="sd*", SUBSYSTEMS=="scsi", ATTRS{model}=="USB 2.0 Storage Device",
SYMLINK+="usbhd%n"
```

위의 규칙은 아래와 같이 심볼릭 링크를 만들어 줍니다.

- */dev/usbhd* - The fdiskable node
- */dev/usbhd1* - The first partition (mountable)
- */dev/usbhd2* - The second partition (mountable)

USB Card Reader

USB 카드 리더기(CompactFlash, SmartMedia 등)은 아직 다른 조건이 필요한 USB 저장 디바이스입니다. 즉 보통의 USB 저장 디바이스들의 일반적인 범위에 속하지 않죠.

그 디바이스들은 일반적으로 일반적으로 미디어 변경 시 호스트 컴퓨터에 어떠한 정보도 알려주지 않습니다. 그래서 여러분이 미디어가 없는 상태로 디바이스를 연결하고 카드를 삽입하면, 컴퓨터는 인식하지 못합니다. 해당 미디어를 위한 마운트 할 수 있는 sdb1 파티션 노드가 없기 때문입니다.

한 가지 해결 방법은 규칙과 일치하는 모든 블록 장치들을 위한 16개의 파티션 노드들을 만드는 방법입니다.

```
KERNEL="sd*", SUBSYSTEMS=="scsi", ATTRS{model}=="USB 2.0 CompactFlash Reader" SYMLINK+="cfrdr%n", OPTIONS+="all_partitions"
```

여러분은 cfrdr, cfrdr1, cfrdr2, …. , cfrdr15의 이름을 가진 노드를 만들 수 있습니다.

USB Palm Pilot

현재는 사용자가 없으므로.. 설명은 생략하겠습니다.

CD/DVD drives

현재 자신이 가진 컴퓨터에 두 개의 광학 드라이버가 있다고 가정합니다. DVD reader(hdc) 와 DVDrewriter(hdd) 라고 가정합시다. 두 개의 디바이스들이 내 PC에 다시 연결됐을 때 그 디바이스 노드들이 변경됐는지 예상할 수가 없습니다. 그래서 많은 사용자들이 편의성을 위해 /dev/DVD등의 디바이스 노드를 가지고 싶어 합니다.

그 디바이스들의 KERNEL 이름을 안다면 규칙을 만드는 것은 매우 간단합니다.

```
SUBSYSTEM=="block", KERNEL=="hdc", SYMLINK+="dvd", GROUP="cdrom"
SUBSYSTEM=="block", KERNEL=="hdd", SYMLINK+="dvdrw", GROUP="cdrom"
```

Network interfaces

네트워크 인터페이스들은 이름으로 참조 되더라도 일반적으로 그와 관련된 디바이스 노드들이 없습니다. 그렇다 하더라도 규칙을 작성하는 방법은 거의 동일합니다.

인터페이스에 대한 규칙을 작성할 때 MAC address를 일치시키는 것은 당연하게 느끼는 것입니다. udevinfo를 이용해 MAC address를 정확히 확인해야 여러분의 규칙이 동작하지 않는 상황이 발생하지 않을 것입니다.

```
# udevinfo -a -p /sys/class/net/eth0
  looking at class device '/sys/class/net/eth0':
    KERNEL=="eth0"
    ATTR{address}=="00:52:8b:d5:04:48"
```

규칙은 아래와 같습니다.

```
KERNEL=="eth*", ATTR{address}=="00:52:8b:d5:04:48", NAME="lan"
```

이 규칙을 적용하려면 net driver를 다시 로드 해야합니다. 다시 로드하는 방법은 시스템을 리부트 하거나 모듈을 제거 후 다시 로드하면 됩니다. 그리고 “eth0”가 아닌 “lan으로 다시 시스템을 재 설정해줘야 합니다. eth0를 참조하고 있는 모든 것들이 완전히 드랍될 때 까지 몇 가지 문제가 발생할 수 있습니다. 예를 들어 인터페이스 이름이 재 정의되지 않을 수도 있습니다. 여러분은 ifconfig나 유사한 프로그램에서 “eth0” 대신에 “lan”을 사용할 수 있어야 합니다.

Testing and debugging

Putting your rules into action

여러분이 inotify를 지원하는 최신 커널을 사용한다면 udev는 자동적으로 udev 규칙 디렉토리를 확인하고 자동적으로 만들어놓은 규칙 파일들의 변경사항을 확인할 것 입니다.

하지만 udev가 모든 디바이스를 재처리하여 새로운 규칙을 적용하는 것은 아닙니다. 만약 여러분의 카메라가 연결되어 있는 상태에서 카메라에 대한 추가적인 심볼릭 링크를 만드는 규칙을 작성한다면 원하는 심볼릭 링크를 확인할 수 없을 것입니다.

심볼릭 링크가 만들어졌음을 확인하기 위해서는 카메라를 제거한 후 새로 연결하거나 디바이스를 제거하지 않고 확인하고 싶다면 **udevtrigger**를 사용하면 됩니다.

만약 inotify를 지원하지 않는 커널을 사용 중이라면 새로 작성한 규칙을 자동적으로 확인하지 못합니다. 이런 경우에 여러분은 규칙을 만들거나 수정한 후에 해당 규칙을 적용하기 위해 **udevcontrol reload_rules**이란 툴을 사용하여 해당 문제를 해결할 수 있습니다.

udevtest

만약 sysfs에서 디바이스의 최상위 경로를 알고 있다면, **udevtest**를 이용하여 udev가 실행하는 작업들을 확인할 수 있습니다. **udevtest**의 기능을 이용한다면 여러분들이 만든 규칙을 매우 편하게 디버깅 할 수

있습니다. 예를 들어 여러분이 /sys/class/sound/dsp에 대한 규칙을 디버깅하길 원한다면 아래와 같이 확인할 수 있습니다.

```
# udevtest /class/sound/dsp
main: looking at device '/class/sound/dsp' from subsystem 'sound'
udev_rules_get_name: add symlink 'dsp'
udev_rules_get_name: rule applied, 'dsp' becomes 'sound/dsp'
udev_device_event: device '/class/sound/dsp' already known, remove possible symlinks
udev_node_add: creating device node '/dev/sound/dsp', major = '14', minor = '3', mode = '0660', uid = '0', gid = '18'
udev_node_add: creating symlink '/dev/dsp' to 'sound/dsp'
```

udevtest를 이용할 때 명령에서 경로 입력 시 /sys 접두사가 생략되는 것에 주의하셔야 하는데요, udevtest 자체가 디바이스 경로에서 동작하기 때문입니다. 추가로 udevtest는 테스트/디버깅 툴이므로 어떤 출력을 제공하더라도 디바이스 노드를 생성하지는 않는다는 것에 주의하셔야 합니다.

Author and contact

이 문서는 Daniel Drake <dan@readctivated.net>에 의해 작성되었으며,
Son Seungha<jujakhana@gmail.com>에 의해 번역 및 이해를 돕기 위한 부분을 추가하여 새로 구성하였습니다.

피드백 주시면 감사하겠습니다.

Korean translation of Writing udev rules
Date: 2015. 10. 23

E-mail: jujakhana@gmail.com