7장. 인터럽트와 인터럽트 처리 (Interrupt and Interrupt Handling)

이 장에서는 리눅스 커널이 인터럽트를 어떻게 처리하는지 살펴본다. 커널이 인터럽트를 처 리하는 데는 일반적인 메커니즘과 인터페이스가 있지만, 인터럽트를 처리하는 세세한 내용 은 아키텍처마다 다르다.

리눅스는 서로 다른 일을 하는 수많은 하드웨어를 사용한다. 비디오 장치는 모니터를 구동 하며, IDE 장치는 디스크를 구동하는 식이다. 이런 장치들은 동기적으로 구동할 수 있다, 즉 어떤 동작을 요청하고 (예를 들면 메모리 블럭을 디스크에 저장하는 것과 같은) 그것이 완 료될 때까지 기다리는 것이다. 하지만 이 방법은 동작하기는 하지만 매우 비효율적이어서 운영체제는 각각의 동작이 완료될 때까지 기다려야 하므로 "아무것도 하지 않으면서 바쁜 상태(busy doing nothing)"로 많은 시간을 소비할 것이다. 이보다 더 좋고 더욱 효율적인 방법 은 요청을 한 뒤 다른 더 유용한 작업을 하고 요청한 작업이 끝나면 장치로부터 인터럽트를 받는 것이다. 이런 설계를 사용하면 여러 장치에 동시에 작업을 요청하는 것이 가능하다.

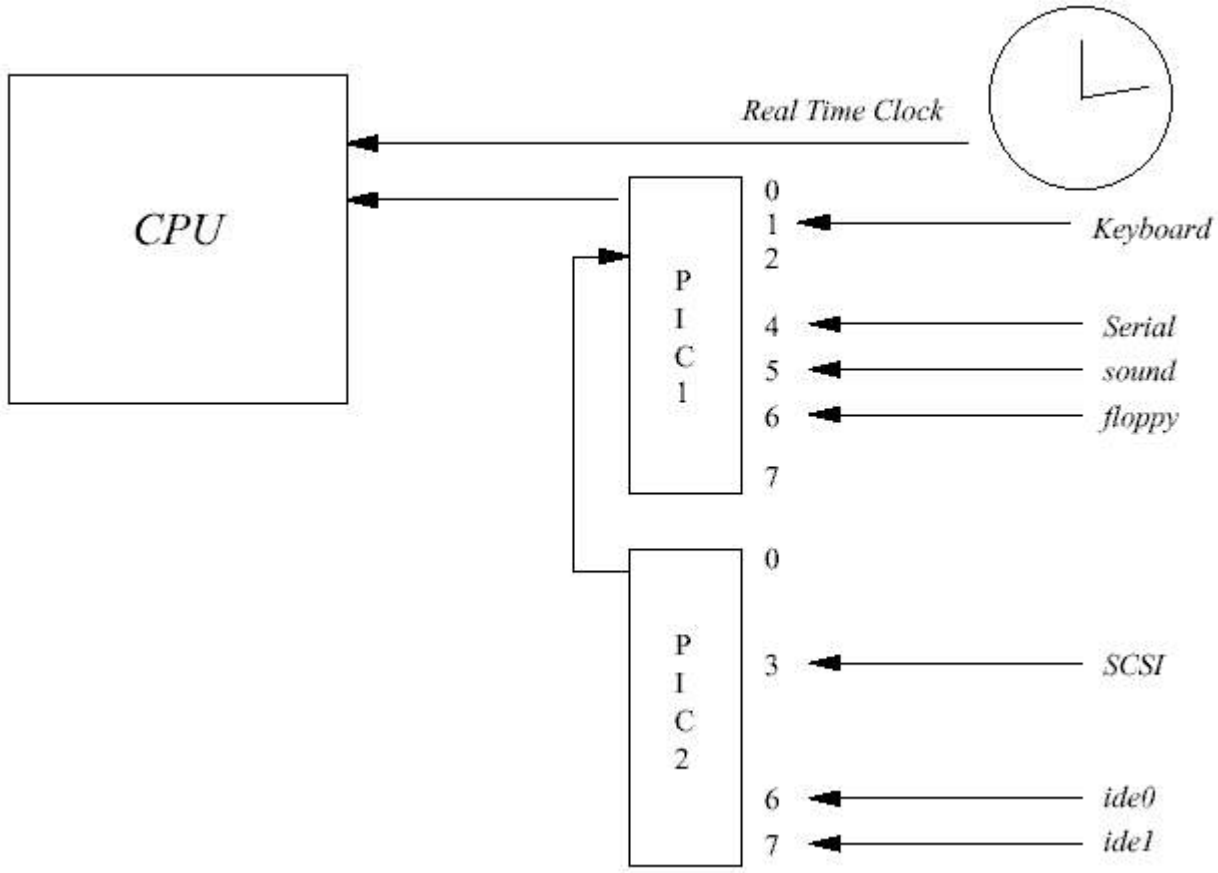

그림 7.1 : 인터럽트 전달 과정에 대한 논리도

CPU가 무엇을 하고있건 간에 장치가 인터럽트를 걸 수 있으려면 하드웨어에서 지원해야 한 다. 모두는 아니더라도 알파 AXP와 같은 대부분의 범용 프로세서들은 대개 비슷한 방법을 사용한다. CPU의 특정한 핀의 전압이 바뀌면 (예를 들어 +5볼트에서 -5볼트로), CPU는 하던 일을 멈추고 인터럽트 처리 코드라는 인터럽트를 다루는 특별한 코드를 수행하기 시작한다. 이들 핀 중 어떤 핀은 간격 타이머에 연결되어 있어 1000분의 1초마다 인터럽트를 받으며, SCSI 컨트롤러와 같은 다른 장치에 연결된 핀들도 있을 것이다.

대체로 시스템은 인터럽트 컨트롤러를 사용하여 CPU의 인터럽트 핀으로 인터럽트를 1:1로 전달하지 않고, 장치 인터럽트를 그룹으로 묶어준다. 이렇게 하면 CPU에 있는 인터럽트 핀 을 줄일 수 있을 뿐 아니라 시스템을 유연하게 디자인할 수 있다. 인터럽트 컨트롤러에는 인터럽트를 조정하는 마스크 레지스터와 상태 레지스터가 있다. 마스크 레지스터의 비트들 을 켜거나 껴서 인터럽트를 가능하게 하거나 불가능하게 만들 수 있으며, 상태 레지스터는 시스템에 현재 발생한 인터럽트를 돌려준다.

시스템의 일부 인터럽트는 하드웨어적으로 연결되어 있다. 예를 들어 실시간 클럭의 간격 타이머는 영구적으로 인터럽트 컨트롤러의 3번 핀에 연결되어 있다. 그러나 어떤 핀들은 특 정한 ISA 또는 PCI 슬롯에 어떤 컨트롤러 카드가 꽂혀 있는지에 따라 어떤 장치에 연결되 는지 결정된다. 예를 들어 인터럽트 컨트롤러의 4번 핀이 PCI 슬롯 0번에 연결되어, 여기에 이더넷 카드를 꽂을 수도 있지만 뒤에 SCSI 컨트롤러로 바꿔 끼울 수도 있다는 것이다. 인 터럽트 처리에 있어서 기본적인 사항은, 각 시스템은 독자적인 인터럽트 전달 방식을 가지 고 있으며, 운영체제는 이에 대처할 수 있도록 유연하게 만들어져야 한다는 것이다.

대부분의 현대 범용 마이크로프로세서들은 인터럽트를 똑같은 방식으로 처리한다. 하드웨어 인터럽트가 발생하면 CPU는 지금 수행하고 있던 명령어의 실행을 중단하고 인터럽트 처리 코드가 있거나 인터럽트 처리 코드로 분기하는 명령어가 있는 메모리 번지로 점프한다. 이 코드들은 일반적으로 인터럽트 모드(interrupt mode)라고 하는 CPU의 특별한 모드에서 수행 되는데, 보통 이 모드에서는 다른 인터럽트가 발생할 수 없다. 물론 예외가 있다. 어떤 CPU 에서는 인터럽트에 우선순위를 매겨 더 높은 우선순위의 인터럽트가 발생할 수 있게 한다. 이런 경우 가장 높은 순위의 인터럽트 처리 코드는 아주 주의해서 작성해야 하며, 종종 자 신의 스택을 가지고 있어 인터럽트를 처리하기 전에 여기에 CPU의 수행상태(즉, CPU의 일 반 레지스터와 컨텍스트 모두)를 저장하는데 사용한다. 어떤 CPU에는 인터럽트 모드에서만 존재하는 특별한 레지스터 세트가 있어, 인터럽트 코드는 필요한 컨텍스트를 저장하는데 이 레지스터들을 사용할 수 있다.

인터럽트가 처리되고 나면 CPU의 상태는 인터럽트 이전으로 복구되고 인터럽트는 해제된다. 그러면 CPU는 인터럽트가 발생하기 전에 수행하던 것을 계속 실행하게 된다. 중요한 것은 인터럽트를 처리하는 코드는 가능한 효율적이어야 하며 운영체제는 인터럽트를 너무 자주 또는 너무 오래 막고 있지 않아야 한다는 점이다.

7.1 프로그램 가능 인터럽트 컨트롤러(Programmable Interrupt Controller, PIC)

시스템 디자이너는 자신이 원하는 어떤 인터럽트 구조라도 사용할 수 있지만, IBM PC는 인 텔 82C59A-2 CMOS PIC [6, 인텔 주변 장치]나 그 유사형을 사용한다. 이 컨트롤러는 PC의 초창기때부터 널리 사용된 것으로, ISA 주소공간에 있는 컨트롤러의 레지스터를 이용해 (이 레지스터의 위치는 고정되어 이미 알려져 있다) 프로그래밍을 할 수 있다. 가장 최근의 로 직 칩 세트도 ISA 메모리의 같은 위치에 동등한 레지스터를 가지고 있다. 알파 AXP 기반 PC와 같이 인텔에 기반하지 않은 시스템들은 이러한 구조적 제약으로부터 자유로우며, 대개 다른 인터럽트 컨트롤러를 사용한다.

그림 7.1에서 8비트 컨트롤러 PIC1과 PIC2가 같이 연결되어 있으며, 각각 마스크 레지스터 와 인터럽트 상태 레지스터 하나씩을 가지고 있는 것을 볼 수 있다. 마스크 레지스터는 주 소 0x21과 0xA1에 있으며 상태 레지스터는 0x20과 0xA0에 있다. 마스크 레지스터의 특정한 비트에 1을 쓰면 해당 인터럽트를 가능하게 하며, 0을 쓰면 인터럽트를 불가능하게 한다. 즉, 세번째 비트에 1을 쓰면 인터럽트 3번을 가능하게 하는 것이며, 0을 쓰면 불가능하게 하는 것이다. 불행하게도 (또한 귀찮게도), 인터럽트 마스크 레지스터는 쓸 수만 있으며, 거기에 써 놓은 값을 읽어올 수는 없다. 따라서 리눅스는 마스크 레지스터에 어떤 것을 설정하였는 지를 따로 복사하여 보관하여야만 한다. 리눅스는 인터럽트 허용 루틴과 인터럽트 금지 루 틴에서, 이 보관된 마스크를 변경하고 매번 레지스터에 전체 마스크를 쓴다.

인터럽트가 발생하면, 인터럽트 처리 코드는 두 인터럽트 상태 레지스터(Interrupt Status Register, ISR)를 읽는다. 인터럽트 처리 루틴은 0x20에 있는 ISR을 16비트 인터럽트 레지스터 의 하위 여덟 비트로, 0xA0에 있는 ISR을 상위 여덟 비트로 처리한다. 따라서 0xA0에 있는 ISR의 첫번째 비트에 해당하는 인터럽트는 시스템 인터럽트 9로 취급하게 된다. PIC1에 있 는 두번째 비트는 PIC2에서 발생하는 인터럽트를 연결하는데 사용하기 때문에 사용할 수 없다. PIC2에 발생하는 어떤 인터럽트든지 PIC1의 두번째 비트를 설정하게 된다.

7.2 인터럽트 처리용 자료구조의 초기화

커널의 인터럽트 처리용 자료구조는 디바이스 드라이버들이 시스템의 인터럽트에 대한 제어 권을 요청하면서 셋업된다. 이를 위해 디바이스 드라이버는 일련의 리눅스 커널 서비스를 사용함으로써, 인터럽트를 요청하고, 인터럽트를 가능하게 하거나, 불가능하게 만든다. 개별 디바이스 드라이버는 이런 루틴들을 불러 자신의 인터럽트 처리 루틴의 주소를 등록한다.

PC 아키텍처의 관례상 몇몇 인터럽트들은 (특정한 장치가 사용하도록) 지정되어 있는데, 이 경우에 해당 디바이스 드라이버는 초기화될 때 간단하게 그 지정된 인터럽트를 요청하면 된 다. 플로피 디스크 디바이스 드라이버가 이와 같이 동작하는데, 항상 IRQ 6번을 요청한다. 하지만 장치가 어떤 인터럽트를 사용하게 될 것인지 디바이스 드라이버가 모르는 경우도 있 다. PCI 디바이스 드라이버의 경우에는 장치가 어떤 인터럽트를 사용하는지 항상 알고 있기 때문에 문제가 되지 않지만, 불행하게도 ISA 디바이스 드라이버의 경우에는 자신이 사용할 인터럽트 번호를 쉽게 찾을수 있는 방법이 없다[1]. 리눅스에서는 이런 문제를 해결하기 위해 디바이스 드라이버가 자신이 사용할 인터럽트를 탐사 (probe)할 수 있도록 허용하고 있다.

먼저 디바이스 드라이버는 장치에 무엇인가를 해서 인터럽트가 발생하도록 한다. 그런 후 다른 장치에 할당되지 않은 시스템의 모든 인터럽트들을 가능하게 한다. 이렇게 하면 처음 에 발생시켰던 장치의 인터럽트가 PIC를 통해 전달될 것이다. 리눅스는 인터럽트 상태 레지 스터를 읽어 그 내용을 디바이스 드라이버에게 돌려준다. 이 값이 0이 아니라면 탐사 중에 하나 이상의 인터럽트가 발생한 것이다. 드라이버는 탐사를 종료하고 다른 장치에 할당되지 않은 인터럽트를 모두 불가능하게 한다[2]. 탐사를 통해 ISA 디바이스 드라이버가 자신이 사 용할 IRQ 번호를 찾았다면, 정상적으로 이에 대한 통제권을 요청할 수 있다.

ISA 기반 시스템에 비해 PCI 기반 시스템은 훨씬 더 동적이다. ISA 장치가 사용하는 인터럽 트 핀은 대개 하드웨어 장치 위에 있는 점퍼를 사용해 설정하고, 디바이스 드라이버에 이 값이 지정되어 있다. 반면에, PCI 장치가 사용할 인터럽트는 시스템이 부팅하면서 PCI를 초 기화 할 때 PCI BIOS나 PCI 서브시스템이 할당해 준다. 각각의 PCI 장치는 A, B, C, D의 4개 의 인터럽트 핀을 사용할 수 있다. 어떤 핀을 사용할 지는 장치를 만들 때 결정되는데, 대부 분은 기본적으로 A 핀에 있는 인터럽트로 설정한다. 각 PCI 슬롯에 있는 PCI 인터럽트 라 인 A, B, C, D는 인터럽트 컨트롤러에 연결되어 있다. 예를 들어 PCI 슬롯 4의 A 핀은 인터 럽트 컨트롤러의 6번 핀에 연결하고, PCI 슬롯 4의 B 핀은 인터럽트 컨트롤러의 7번 핀에 연결하는 식으로 되어 있다.

PCI 인터럽트가 어떻게 전달되는지는 시스템마다 다르므로, PCI 인터럽트 전달 구조를 이해 할 수 있는 셋업 코드가 필요하다. 인텔 칩을 사용하는 PC에서는 시스템이 부팅할 때 실행 되는 시스템 BIOS가 이 역할을 하는데, 알파 AXP를 사용하는 시스템과 같이 BIOS가 없는 시스템의 경우에는 리눅스 커널이 이러한 설정을 한다. PCI 셋업 코드는 각 장치별로 인터 럽트 컨트롤러의 핀 번호를 PCI 설정 헤더에 쓴다. 그리고 장치가 사용하는 PCI 슬롯 번호 와 PCI 인터럽트 핀 번호 및 PCI 인터럽트 전달 구조를 이용하여 인터럽트 핀 (또는 IRQ) 번호를 결정한다. 이러한 방법으로 장치가 사용할 인터럽트 핀 번호가 고정되고, 인터럽트 핀 번호는 이 장치를 관리하는 PCI 설정 헤더에 있는 항목에 저장된다. 셋업 코드는 이 정 보를 이러한 목적으로 마련된 인터럽트 라인 항목에 적어 넣는다. 디바이스 드라이버는 이 정보를 읽어다 리눅스 커널에게 인터럽트에 대한 제어권을 요청할 때 사용한다.

PCI-PCI 브릿지를 사용할 때와 같이 시스템에 PCI 인터럽트를 일으키는 장치가 많은 경우가 있다. 인터럽트를 일으키는 장치가 시스템의 PIC에 있는 핀 수보다 많을 수 있다. 이 경우 PCI 장치라면, 인터럽트를 공유하여 여러 PCI 장치의 인터럽트가 인터럽트 컨트롤러의 핀 하나에 발생하게 할 수 있다[3].

이런 인터럽트 공유를 지원하기 위해 리눅스는 해당 인터럽 트의 제어권을 처음으로 요청하는 디바이스 드라이버가 인터럽트를 공유할 수 있는지를 밝 히도록 하고 있다[4]. 인터럽트를 공유하기 위해 irq_action 벡터에 irqaction 자료구조를 여러 개 담게 된다. 공유 인터럽트가 발생하면, 리눅스는 그 인터럽트를 사용하는 장치의 인 터럽트 핸들러를 모두 불러준다. 인터럽트를 공유할 수 있는 모든 디바이스 드라이버(모든 PCI 디바이스 드라이버겠지만)는 서비스할 인터럽트가 없는 경우라 하더라도 인터럽트 핸들 러가 불릴 수 있으므로 이에 대비해야 한다[5].

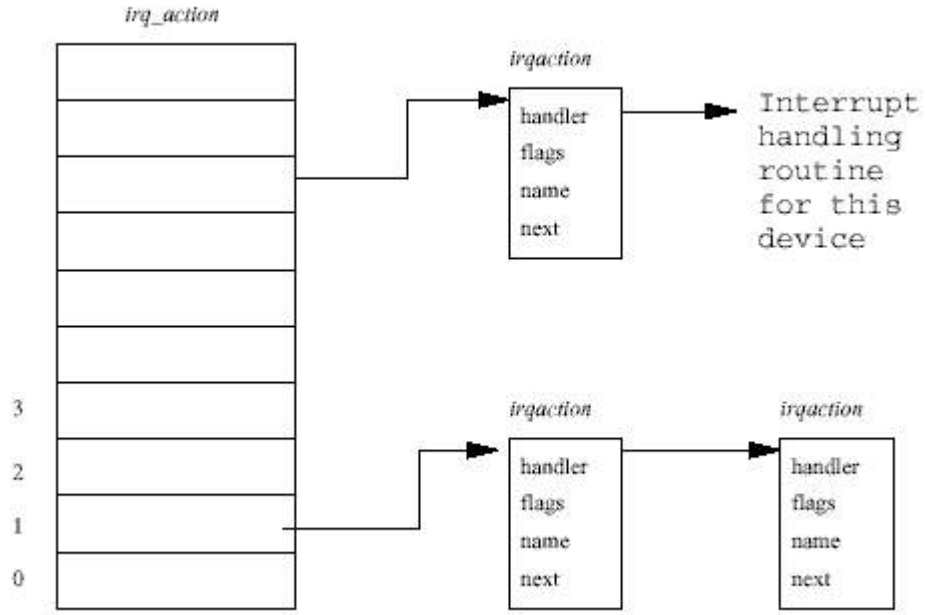

그림 7.2 : 리눅스의 인터럽트 처리 자료구조

7.3 인터럽트 처리

리눅스의 인터럽트 처리 서브시스템의 주요한 임무중 하나는 인터럽트를 올바른 인터럽트 처리 코드로 전달하는 것이다. 따라서 인터럽트 처리 서브시스템은 시스템의 인터럽트 전달 구조를 파악하고 있어야만 한다. 예를 들어 플로피 컨트롤러가 인터럽트 컨트롤러의 6번 핀 에 인터럽트를 일으킨다면[6], 리눅스 커널의 인터럽트 처리 서브시스템은 이 인터럽트가 플 로피에서 발생할 것임을 인지하고 이것을 플로피 디바이스 드라이버의 인터럽트 처리 코드 로 전달해야 한다. 이를 위해 리눅스는 시스템의 인터럽트를 처리하는 루틴들의 주소를 가 지고 있는 자료구조에 대한 일련의 포인터를 사용한다. 이 루틴들은 해당 디바이스 드라이 버에 있는 것이며, 드라이버가 초기화될 때 자신이 사용할 인터럽트를 요청하는 것은 각 드 바이스 드라이버의 책임이다. 그림 7.2는 irqaction 자료 구조를 가리키고 있는 포인터들 의 벡터인 irq_action을 보여주고 있다. 각 irqaction 자료구조는 인터럽트 처리 루틴 의 주소를 포함해 해당 인터럽트에 대한 핸들러 정보를 담고 있다. 인터럽트의 수와 이들이 어떻게 처리되는지는 아키텍처마다, 때때로 시스템마다 다르기 때문에 리눅스의 인터럽트 처리 코드는 아키텍처 종속적이다. 즉, irq_action 벡터의 크기는 시스템에 있을 수 있는 인터럽트를 일으키는 장치의 숫자에 따라 달라진다.

인터럽트가 발생하면, 리눅스는 먼저 시스템에 있는 PIC의 인터럽트 상태 레지스터(ISR)를 읽어 어느 장치가 인터럽트가 일으켰는지 알아낸다. 그런 후 리눅스는 그 장치를 irq_action 벡터의 오프셋으로

변환한다. 예를 들어, 플로피 컨트롤러가 일으키는 인터럽 트 컨트롤러 6번 핀에 발생한 인터럽트는 인터럽트 핸들러 벡터의 일곱번째 포인터로 변환 된다. 인터럽트가 발생하였는데 이를 처리할 인터럽 트 핸들러가 없다면 리눅스 커널은 오류 를 기록할 것이다. 핸들러가 있다면 이 인터럽트를 일으키는 모든 장치에 대해 irqaction 자료구조에 있는 인터럽트 처리 루틴을 부를 것이다.

리눅스 커널이 디바이스 드라이버의 인터럽트 처리 루틴을 부르면, 이 루틴은 왜 인터럽트 가 발생하였지를 파악하여 이에 효율적으로 반응해야 한다. 왜 인터럽트가 발생하였지를 파 악하기 위해 디바이스 드라이버는 인터럽트가 발생한 장치의 상태 레지스터를 읽을 수 있을 것이다. 그 이유는 오류였을 수도 있고 요청한 작업이 완료됐다고 보고한 것일 수도 있다. 예를 들어 플로피 컨트롤러는 플로피 디스크의 정확한 섹터 위에 플로피의 헤드를 올려 놓 았다고 보고할 수 있다. 인터럽트가 발생한 이유를 알아 냈다면, 디바이스 드라이버는 인터 럽트를 처리하기 위해 더 많은 작업을 해야 할 필요가 있을 있다. 그런 경우 리눅스 커널에 는 디바이스 드라이버가 그 작업을 뒤로 연기할 수 있는 메커니즘이 있다[7]. 이것은 CPU가 너무 오래동안 인터럽트 모드에 있는 것을 피하려는 것이다[8]. 더욱 자세한 내용은 디바이스 드라이버 장(8장)을 보라.

REVIEW NOTE : Fast interrupt와 slow interrupt는 인텔에 있는 개념인가?[9]

번역 : 김성룡, 홍경선
정리 : 이호

역주 1) ISA 장치는 인터럽트를 하드웨어에 있는 점퍼로 설정하거나 EPROM에 값을 기록함 으로써 지정하도록 되어 있다. 하지만 이들 값을 운영체제에서 알아낼 수 있는 방법은 없다. (flyduck)

역주 2) 커널에는 이를 위해 probe_irq_on()과 probe_irq_off() 함수가 있다. 잎의 함수 를 불러 할당되지 않은 인터럽트를 가능하게 한 후, 인터럽트를 발생한 후, 뒤의 함수를 부르면 발생한 인터럽트를 돌려주고, 인터럽트 상태를 원상태로 복구한다. probe_irq_off() 함수는 인터럽트가 발생하지 않으면 0을, 하나의 인터럽트가 발생하 면 해당 인터럽트 번호를, 둘 이상의 인터럽트가 발생하여 모호한 경우 음수를 돌려준다. (flyduck)

역주 3) 사실 PCI에서는 규약으로 인터럽트 공유가 가능하도록 되어 있지만, ISA라고 해서 인터럽트를 공유할 수 없는 것은 아니다. 물론 ISA 규약에는 인터럽트 공유에 대한 규정 이 없고 초창기에 나온 카드는 전기적인 문제로 인터럽트 공유에 문제가 있었지만 지금 있는 대부분의 ISA 카드는 하드웨어적으로 인터럽트 공유에 문제가 없다. 따라서 인터럽 트 공유의 문제는 대부분 소프트웨어 문제이며, 리눅스는 ISA 디바이스 드라이버라고 하 더라도 인터럽트 핸들러를 등록할 때 인터럽트를 공유할 수 있는지 지정할 수 있다. (flyduck)

역주 4) 물론 인터럽트를 인터럽트를 공유하지 않는 인터럽트 핸들러가 설치되어 있다면 인 터럽트를 공유하는 핸들러를 등록할 수 없을 것이며, 반대의 경우도 마찬가지다. (flyduck)

역주 5) 즉 자신이 처리하는 장치에서 인터럽트가 발생하지 않았더라도 인터럽트를 공유하 는 다른 장치에서 발생한 인터럽트 때문에 자신의 인터럽트 핸들러가 불릴 수 있다는 것 이다. 이는 자신이 제어하는 장치에 있는 인터럽트 상태 레지스터를 읽어서 인터럽트가 발생한 경우 이를 처리하고, 그렇지 않은 경우에는 그냥 무시하면 된다. 그러면 실제로 인터럽트가 발생한 장치의 디바이스 드라이버가 이를 처리할 것이다. (flyduck)

6) 사실 플로피 컨트롤러는 관례상 PC 시스템에서 인터럽트가 고정된 장치 중 하나이다. 플 로피 컨트롤러는 항상 인터럽트 6번에 연결된다.

역주 7) 이런 방법으로 하반부 핸들러(bottom half handler)와 작업큐(task queue)가 있다. (flyduck)

역주 8) 인터럽트를 처리하는 도중에는 다른 인터럽트를 발생하지 못하도록 하기 때문에 (모든 인터럽트이든, 우선순위가 낮은 인터럽트이든), 인터럽트 처리 상태에 오래 있는 것은 좋지 않다. (flyduck)

역주 9) 빠른 인터럽트(fast interrupt)와 느린 인터럽트(slow interrupt)는 인터럽트 처리 방식의 차이이다. 빠른 인터럽트는 인터럽트 처리가 원자적으로(atomic) 이루어지는 경우이고, 느 린 인터럽트는 그렇지 않다. 실질적인 차이는 빠른 인터럽트의 경우 프로세서에서 인터 럽트를 금지시켜 처리중 다른 인터럽트의 방해를 받지 않지만, 느린 인터럽트는 다른 인 터럽트에 의해 중지될 수 있다. 그리고 빠른 인터럽트는 인터럽트 핸들러는 앞뒤에 하는 일이 적이 보다 빠르다. 알파나 Sparc에서는 이런 차이는 없으며, 인텔에서도 2.1.37 버전 이후에 이 차이는 없어졌다. (flyduck)