Plug-and-Play-HOWTO

David S.Lawyer

dave (at) lafn.org

도운이

김현종

polsci (at) hitel.net

v1.03, August 2001

본 문서는 플러그 앤 플레이(PnP, Plug and Play)라는 난해한 주제의 이해를 돕는다. 현재 당신의 PC 에서 PnP 기능이 제대로 작동하지 않고 있다면 이를 동작시키는 방법도 다룬다. 본 문서는 소위 "유니버설 플러그 앤 플레이(UPnP, Universal Plug and Play)"라 불리는 내용은 다루지 않는다. 그 부분은 9.1절 를 참고하라.

고친 과정

고침 v1.032001-12-25고친이 김 현종

1.03으로 갱신 & 수정

고침 v0.11 고친이 남 상현

최초 번역

1. 소개

1.1. 저작권, 상표, 면책사항, 도와주신 분들

역주 : 독자의 편의를 위해 이 부분의 원문과 그 번역문을 같이 싣는다. 그 법적 효과에 대해 번역문을 신뢰해서는 절대 안되며 오로지 원문의 내용만을 신뢰해야한다.

1.1.1. 저작권

1.1.2. 면책사항

본의는 아니지만 문서중에 잘못된 부분이 있을 수 있습니다. 그런 부분이 있다면 부디 필자에게 알려주십시요. 본 문서는 자유 문서이기 때문에 필자는 문서내의 어떠한 잘못된 부분에 대해서도 법적인 책임을 질수 없음을 명확히 합니다.

1.1.3. 등록 상표

본문에 등장하는 브랜드 명칭은 상표명에 해당합니다. 그 상표들은 해당 소유자에 귀속됩니다.

1.1.4. 도와주신 분들

- 다니엘 스콧(Daniel Scott) 씨는 2000 년 3 월자 본 문서를 검토해서 많은 교정등을 해주셨습니다.
- 페테 바렛(Pete Barrett) 씨는 윈도우가 PCI IRQ 들을 비워버리지(zeroing) 못하게 하는 방법을 알려주셨습니다.

1.2. 앞으로의 계획; 당신이 도와줄 수 있는 방법

본 문서에 대해 틀린 부분, 의견, 논리적 구성, 오자, 문법적 잘못, 불명확한 부분, 링크, 기타 등등의 어떤 것이든 필자에게 알려주십시요. 하지만 그전에 먼저 본 문서의 날짜가 한달 이전의 것이라면 최신 버전의 문서가 나왔는지 우선 확인해 주십시요. 본 문서에 포함될만한 어떠한 정보라도 필자에게 보내주십시요.

필자는 각종 리눅스 드랑이버들이 PnP 를 구현하기 위해 사용한 코드들을 일일이 알아보지는 않았습니다. BIOS 들이 어떻게 PnP 를 설정하는지도 완전히 이해하지는 못하며(이는 BIOS 에 따라 다릅니다), 윈도우즈 9x 가 어떻게 ESCD 를 갱신하는지도 완전히는 모릅니다. 따라서 본 문서는 아직 불완전하며 어쩌면 부정확할 수도 있습니다(잘못된 부분은 필자에게 알려주십시요). 문서중에 때때로 ?? 라고 쓴 부분은 필자도 정확히는 모름을 나타냅니다.

1.3. 본 문서의 새로운 버전

본 Plug-and-Play-HOWTO 문서는 수개월마다 계속 갱신될 것이며 LDP 미러 사이트들에서 열람하거나 다운받을수 있을 것입니다. 미러 사이트들의 리스트는 다음과 같습니다 : <65533> (http://linuxdoc.org/mirrors.html). 여러가지 포맷이 준비되어 있습니다. 최신 문서의 날짜만 빨리 확인하고 싶다면 다음을 보십시요: <65533> (http://linuxdoc.org/HOWTO/Plug-and-Play-HOWTO.html). 당신이 지금 읽고있는 본 문서의 버전은 다음과 같습니다: v1.03, August 2001.

1.4. 최근의 버전에서 갱신된 내용들

v1.03 August 2001: 에러 메시지들, 부트 프롬프트 파라메터들. v1.02 July 2001: PCI 설정 레지스터들. v1.01 April 2001: 버스 리소스 부족문제가 최근에는 감소 추세. 2 절을 명확히 함. 윈도우즈 2000 은 (심지어 CMOS 에서 "PnP OS 없음" 로 설정한 경우에도) 잘 작동함.

1.0 버전(Nov. 2000)이 완성하기되기까지는 상당한 시일이 걸렸으며 디바이스 드라이버들이 PnP 를 설정하는데 있어 커널의 도움을 많이 받는다는 것을 알게됨. 커널 2.4 는 이 점에서 획기적으로 향상되었음. 본 문서는 물론이거니와 리눅스 역시 PnP 를 구현하는 방법에 있어서 아직 많은 개선이 필요함.

1.5. 일반적인 개요. 본 문서를 읽을 필요가 있는가?

PnP(Plug-and-Play)란 디스크, 사운드 카드, 이더넷 카드, 모뎀, 기타 등등의 PC 의 디바이스들을 자동으로 감지하는 시스템을 말한다. PnP 는 이 디바이스들을 저수준 설정하는 작업도 일부 담당한다. 어떤 디바이스가 PnP 에 의해 감지되기 위해서는 그 디바이스는 반드시 PnP 로 디자인되었어야 한다. PnP 아닌 디바이스는(혹은 이미 제대로 PnP 설정되어진 디바이스인 경우에도) PnP 아닌 방법을 쓰면 감지해낼수 있을 때가 많다.

리눅스 커널은 중앙집중식 PnP 시스템이 아니다. 각각의 드라이버들이 각자의 PnP 를 사용하는데 필요한 프로그램들을 리눅스 커널이 제공해 줄 뿐이다. 많은 드라이버들이 이를 이용해 PnP 디바이스들이 제대로 작동하는지 확인하게 된다. PC 의 BIOS 하드웨어도 PnP 작업의 일부를 담당할 수 있다. 따라서 시스템이 PnP 로서 다 잘 동작한다면 당신은 더이상 PnP 에 관해 공부할 필요없이 컴퓨터를 잘 사용할 수 있을 것이다. 하지만 PnP 가 발견해내지 못했거나 올바르게 설정해내지 못해서 리눅스를 지원하는 몇몇 디바이스들이 제대로 작동하지 않는 경우라면, 이 문서의 일부를 읽을 필요가 있을 것이다. PnP 에 관한 것뿐 아니라 컴퓨터 내부의 각 부분의 통신에 관한 내용도 배울 수 있을 것이다.

본 문서내에서 필자는 일어날 수 있는 많은 잘못들에 대해 언급하고 있기때문에 머피의 법칙(잘못될 가능성이 있는 일은 잘못되게 마련이다)을 믿는 독자이라면 상당히 겁을 먹게 될지 모르겠다. 하지만 또 역으로 생각해보면 비록 무언가 잘못될 가능성이 있다해도 실제로 그 잘못이 발생하는 경우는 드문 법이다. PnP 역시 대부분의 사람들에 있어서는 그러하다. 때때로 PnP 와 관련된 것처럼 보였던 문제들이 사실은 하드웨어 자체의 결함에 기인한 것이거나 그 하드웨어가 PnP 규격을 제대로 따르지 않았기 때문일 수 있음을 명심하라.

2. PnP 가 하는 일 : "버스리소스(Bus-Resources)" 의 할당

2.1. 플러그 앤드 플레이(PnP, Plug-and-Play)란 무엇인가?

이 절의 내용을 이해하지 못하겠다면 다음 절 2.2절 을 보라.

아주 간략히 말한다면, PnP 는 소프트웨어(디바이스 드라이버들)에게 어디에서 모뎀, 네트워크 카드, 사운드 카드등의 각종 하드웨어(디바이스)를 찾아야 하는지를 자동으로 알려주는 것이다(역주: PnP 를 쓰지 않는다면 사용자가 일일이 지정해 주어야만 한다). PnP 의 임무는 물리적 장치(디바이스)와 그를 조작하는 소프트웨어(디바이스 드라이버)를 서로 일치시키고, 각 디바이스와 그 드라이버 사이에 통신채널들을 만드는 것이다. 이를 실현하기 위해서 PnP 는 다음의 "버스리소스(bus-resources)" 라는 것을 드라이버와 하드웨어의 양쪽에 할당한다: I/O 어드레스, IRQ, DMA 채널(ISA 버스의 경우만 해당), 메모리 영역. 이 4 가지를 일차적 리소스라고 부르기도 한다. 이 4 개의 버스리소스들이 뭔지 잘 모르겠다면 뒤에 나올 I/O 어드레스, IRQ, DMA 채널, 메모리 영역의 절을 읽어보라. 또 이 버스리소스들 중 3 가지에 관한 Linux Gazette 의 기사가 Introduction to IRQs, DMAs and Base Addresses (http://www.linuxgazette.com/issue38/blanchard.html) 에 있다. 일단 이 버스리소스들이 할당되면 (그리고 올바른 드라이버가 설치되면), /dev 디렉토리에 있는 디바이스용 "파일"들을 사용할 수 있게 된다.

이렇게 PnP 가 버스리소스들을 할당하는 것을 "설정(configuring)" 한다라고 말하기도 하지만, 시실 이 PnP 설정이라는 것은 저수준 설정에 해당할 뿐이다. /etc 디렉토리 밑에는 많은 설정 화일들이 있다. 그 내용들의 대부분은 PnP 설정이 아니다. 하드웨어 디바이스들을 설정하는 작업의 상당부분은 PnP 와는 무관하다. PnP 설정은 전체 설정작업의 일부일 뿐인 것이다. 예를 들어, "초기화 문자열(init string)"을 보내 모뎀을 초기화시키거나 모뎀 속도를 설정하는 작업은 설정작업이긴 해도 PnP 설정작업은 아닌 것이다. 따라서, 앞으로 PnP 문제를 논할 때의 "설정(configure)" 이라는 용어는 어떤 특정한 종류의 설정작업만을 뜻하기로 한다. 다른 문서(예를 들면 MS 윈도우즈용의 문서등)에서는 버스 리소스를 "리소스(resources)" 라고 말하기도 하지만, 이 문서에서는 다른 종류의 많은 리소스들과 구별하기 위해서 "버스 리소스" 라는 용어를 사용하기로 한다.

2.2. 컴퓨터와 디바이스는 서로를 어떻게 찾아내는가?

컴퓨터는 계산을 담당하는 CPU(프로세서)와 프로그램 및 데이터를 저장하는 램 메모리(액세스속도가 빠르다)로 구성되어 있다. 여기에다 각종 디스크 드라이브들과 비디오 카드, 키보드, 네트웍 카드, 모뎀 카드, 사운드 카드, USB 버스, 시리얼 포트, 패러렐 포트 등등이 있고, 또 전력을 공급하는 전원부, 디바이스를 CPU 에 접속하기 위한 마더보드상의 각종 버스들, 그리고 이 모두를 담는 케이스가 있다.

옛날에는 대부분의 디바이스들은 각각 회로기판형(PCB, Printed Circuit Board) 카드형태로 제작되어 메인보드의 슬롯에 꽂히도록 되었었다. 그러던 것이 최근에는 많은 "디바이스" 들이 회로기판형태뿐 아니라 작은 반도체 칩 형태로도 집적되어 "마더보드" 상에 아예 영구적으로 실장되기도 한다. 또한 마더보드에 꽂는 하나의 카드위에 여러개의 디바이스가 탑재되는 경우도 있다. 메모리 칩들 역시 때로는 디바이스로 취급할 수도 있지만, 이 경우는 본 HOWTO 의 PnP 의 의미에는 해당하지 않는다.

컴퓨터 시스템이 제대로 동작하려면 각각의 디바이스는 해당 "디바이스 드라이버" 의 제어하에 있어야만 한다. 디바이스 드라이버는 운영체제의 일부인 소프트웨어로서 (대개는 모듈로서 로드된다), CPU 상에서 실행된다. 디바이스 드라이버들은 /dev 디렉토리에 있는 "특수 파일들"에 연관되어져 있는데, 이 파일들은 사실은 파일이 아니다. 이 파일들은 hda1(하드디스크 a 의 첫번째 파티션), ttyS0(첫번째 시리얼 포트), eth1(두번째의 이더넷 카드) 등의 이름을 가지고있다. 좀더 복잡하게는, 가령 eth1 용 디바이스 드라이버는 당신이 가진 이더넷 카드의 종류에 맞는것으로 선택해야 한다. 즉, 다짜고짜 아무 이더넷 드라이버에게나 eth1 을 지정해 주어서는 안되며, 반드시 당신이 설치한 이더넷 카드 종류에서 동작하는 특정 드라이버에게 eth1 을 지정해줘야 한다. 커널 모듈을 사용할 경우, 이러한 지정내용의 일부는 /etc/modules.conf 내의 alias 부분에 적히게 되며 그외 나머지 내용은 커널 내부의 테이블에 내재될 수도 있다.

어떤 디바이스를 제어하기 위해서는, 해당 디바이스 드라이버가 CPU 를 이용해 그 디바이스에 명령 및 데이터를 보내거나 디바이스로부터 정보를 읽어내야 한다. 이런 일을 하려면, 각 디바이스 드라이버들은 자신이 제어하는 디바이스의 어드레스를 알아야만 한다. 결국 그러한 어드레스를 안다는 것은 바로 디바이스와의 통신 채널을 설정한다는 것과 같은 의미인 것이다. 이 물리적인 "채널"은 실제로는 PC 내부의 데이터 버스로서, 디바이스뿐 아니라 다른 거의 모든 장치들에게도 공유되고 있다.

실제의 통신 채널은 위의 설명보다 조금 더 복잡하다. "어드레스" 란 실제로는 연속된 어드레스들로 이루어진 특정한 어드레스 범위를 뜻하는 것으로서 종종 "어드레스"란 말 대신 "어드레스 영역"이라고 말하기도 한다. 심지어는 한개의 단일 디바이스가 두개 이상의 어드레스 범위를 가질수도 있다(서로 중첩되지만 않으면 된다). 또한, 위와 반대방향의 채널에 해당하는 것으로 인터럽트가 있는데, 이를 통해서 이번에는 디바이스가 해당 디바이스 드라이버쪽으로 긴급 "구조요청" 을 보낼수 있다.

2.3. 각종 어드레스들

PC 에는 3 개의 어드레스 공간이 있다 : I/O , 메인 메모리(IO 메모리), 설정 어드레스(구형 ISA 버스는 이 "설정" 어드레스 공간이 없다). 이들 3 종류의 어드레스는 PC 내부의 동일 버스를 공유하고 있다. 어떤 어드레스가 I/O, 메인 메모리(2.5절), 설정 어드레스 셋 중의 어디에 속하는 것인지는 그 판별용으로 PC 의 버스상에 배선된 특정한 라인들의 전압으로 결정된다. 자세한 것은 9.2절 절을 참조하라. 이 세가지 어드레스 공간중 다음 두 가지만이 디바이스의 I/O 용도로 사용된다: I/O 와 메인 메모리.

2.4. I/O 어드레스와 그 할당

디바이스들은 원래 I/O 어드레스 공간에 위치했었으나, 오늘날에는 메인 메모리내의 공간을 사용할 수도 있다. I/O 어드레스는 때로는 그냥 "I/O", "IO", "i/o", "io" 라고 부르기도 한다. 또 "I/O포트" 나 "I/O 영역" 이라 부르기도 한다. 이 "IO 포트" 와 메인 메모리에 위치하는 "IO 메모리" 를 혼동해서는 안된다. I/O 어드레스(또는 인터럽트같은 다른 버스 리소스 자원들)를 할당하려면 다음 두 단계를 거쳐야한다.

1. I/O 어드레스나 그밖의 내용을 카드상의 레지스터에 셋팅한다.
2. 이 I/O 어드레스나 그밖의 내용을 디바이스 드라이버도 알게 한다.

위의 두 단계는 당신이 거리에서 어떤 이의 집의 번지수를 찾아내는 과정과 비슷하다. 먼저, 그 사람은 대문앞에 자기집의 번지수를 붙여놓아 외부에서 찾을수 있게 하여야 하고, 그러면 이를 통해 당신은 그 집 번지수를 얻어내 종이에 기록하면 된다. 컴퓨터의 경우, 먼저 디바이스 하드웨어가 자신이 사용할 어드레스를 특별한 레지스터에 기록해두어야 하고, 그러면 디바이스 드라이버는 그 어드레스를 알아내야만 한다. 이 두가지 작업은 소프트웨어로 자동으로 수행시키든, 아니면 특정한 파일에 손수 그 정보를 기록해두는 방법을 쓰든 반드시 완료되어야 한다. 두 작업중 어느 하나만을 시도하거나 완료시켰을 뿐이라면 문제가 발생한다.

수동으로 PnP 설정을 할 때 몇몇 사람들은 위의 두 작업중 하나만 해놓고는 왜 컴퓨터가 디바이스를 발견하지 못하는지 의아해한다. 예를 들면, 그들은 시리얼 포트에 특정 어드레스를 할당해준답시고 "setserial" 명령을 내리면서도, "setserial" 명령의 효과는 드라이버에게 어드레스를 알려줄 뿐임을 모르는 것이다. setserial 명령은 시리얼 포트 하드웨어내의 자체 레지스터에 어드레스를 셋팅하는 작업을 수행하지는 않는다. 만일 그 시리얼 포트 하드웨어가 당신이 setserial 명령으로 지시한 어드레스를 가지고 있지 않거나 아예 하드웨어에 어드레스가 셋팅되어있지 않은 상태라면 문제가 발생한다.

디바이스 드라이버가 어떤 어드레스를 사용하려면, 그전에 반드시 해당하는 물리적인 디바이스(가령 카드) 내에 그 어드레스가 셋팅되어있어야만 한다. 디바이스 드라이버들은 주로 컴퓨터가 시작되자마자 작동시작하는 것이 많기 때문에, 때때로 PnP 설정 프로그램이 카드내에 어드레스를 셋팅하기도 전에 드라이버가 먼저 카드에 엑세스를 시도하는 수가 있다(그 어드레스에 카드가 존재하는지 등등을 조사하려고). 이렇게 되면 카드가 있음에도 불구하고(하지만 카드가 어드레스를 갖고있지 않은 상태이므로), 카드를 찾을수 없다는 에러 메시지가 나타난다.

앞 문단들에서 설명한 I/O 어드레스에 관한 내용은 다른 자원(2.5절, 2.6절, 2.7절)에 대해서도 그대로 적용된다. 다음 세개 절에서 이들에 대해 설명한다.

2.5. 메모리 영역

메인 메모리내의 어드레스 공간을 할당받는 디바이스들도 많다. 때로는 이것을 "공유 메모리(shared memory)" 또는 "메모리 맵된 I/O(memory mapped I/O)" 또는 "IO 메모리" 라고 부르기도 한다. 이 메모리는 물리적으로는 디바이스상에 존재한다. 버스리소스를 논할 때는 그냥 "메모리"라고 부르기도 한다. 어떤 디바이스들은 이런 "메모리" 와 전통적인 I/O 어드레스공간을 둘다 사용하기도 한다.

이런 카드를 꽂으면 결과적으로 메인 메모리용 메모리 모듈을 꽂는 효과가 발생한다. 진짜 메인 메모리 칩들과 충돌하지 않도록 메모리맵 어드레스는 PnP 에 의해 상위 어드레스가 선택되어진다. 이 메모리는 ROM(Read Only Memory)일수도 있고 공유 메모리일 수도 있다. I/O 어드레스 공간이 디바이스와 CPU 에 공유되듯이, 공유 메모리도 (디바이스 드라이버를 구동시킴으로써) 디바이스와 CPU 가 공유하게된다. 이 공유 메모리는 디바이스와 메인 메모리간의 데이터 "전송" 수단으로 기능한다. 이것은 IO 지만 그 기능이 I/O 공간에서 이루어지는 것이 아니다. 카드와 디바이스 드라이버는 둘다 해당 공유 메모리의 어드레스를 알아야만 한다.

ROM 의 경우는 다르다. ROM 의 내용은 보통 그 디바이스가 사용할 프로그램(대개는 디바이스 드라이버)일 경우가 많다. 어쩌면 초기화코드일 뿐이라서 여전히 별도의 디바이스 드라이버가 필요할 수도 있다. 바람직스럽게도, 이 코드는 MS 윈도우즈 ?? 뿐만 아니라 리눅스에서도 동작할 것이다. 이것은 쉐도우(shadow) 될 필요가 있을지도 모른다. ROM 이 쉐도우된다는 것은 고속 동작을 위해 ROM의 내용을 메인 메모리로 복사해두는 것을 말한다. 일단 ROM 이 쉐도우되면, 그것은 더이상 "읽기 전용(read only)" 이 아니다.

2.6. IRQ --개요

이 설명을 읽은 뒤 더 자세한 것을 알고 싶다면 9.4절를 참고하라. 다음 내용은 일부러 극히 단순히 설명한 것이다: 어드레스 외에도, 인터럽트 번호(예를 들면 IRQ 5 등등)라는 것도 다루어야 한다. 이것은 IRQ(Interrupt ReQuest, 인터럽트 요구)번호라고 부른다. 디바이스 드라이버가 디바이스측에게 통신을 하기 위해서는 카드의 어드레스를 반드시 알고 있어야 함은 이미 설명하였다. 그렇다면 반대방향의 통신은 어떨까? 디바이스가 디바이스 드라이버에게 즉시 전해야 하는 내용이 있다면? 예를 들어, 디바이스가 메인 메모리를 목적지로 하는 대량의 바이트열을 외부에서 받았다고 치자. 이 경우, 디바이스는 거의 가득차가고있는 자신의 버퍼에서 바이트 열을 몽땅 퍼내어 메인메모리로 전송시켜줄 것을 디바이스 드라이버에게 요구할

필요가 있다. 또다른 예로, 외부로 전송해야할 바이트들을 다 전송한 디바이스가 디바이스 드라이버에게 작업을 끝마쳤음을 알리고 전송시킬 새로운 바이트들을 보내달라고 요청할 경우도 있다.

어떻게 하면 디바이스가 드라이버에게 즉각 신호를 보낼수 있을까? 메인 데이터버스는 뭔가가 이미 사용중이기 십상이므로 이를 사용할 수는 없다. 그대신, 디바이스는 인터럽트 전용으로만 사용되는 인터럽트 라인(버스의 일부이다)에 전압을 가한다. 이 라인은 보통 그 특정 디바이스 전용으로 예약되어져 있다. 이 전압신호를 인터럽트 요청(IRQ) 혹은 그냥 "인터럽트" 라고 부른다. PC 의 경우 인터럽트 라인은 16 개가 있어서, 이들은 각자 (간접적으로) 특정 디바이스 드라이버로 연결된다. 각 인터럽트 라인에는 고유의 IRQ(Interrupt ReQuest) 번호가 붙어 있다. 디바이스는 자신의 인터럽트를 그에 해당하는 올바른 라인으로 보내야 하고, 디바이스 드라이버는 올바른 라인에서 그 인터럽트 도착을 기다려야 한다. 특정 디바이스가 어느 라인을 통해 구조 요청을 보내는가는 그 디바이스에 기록저장되어있는 IRQ 번호에 의해 결정된다. 해당 디바이스 드라이버도 이 번호를 똑같이 알고있어야 한다. 그래야 해당번호의 IRQ 라인에서 구조 요청을 기다릴수 있기 때문이다.

일단 어떤 디바이스 드라이버가 해당 디바이스의 인터럽트를 받으면, 디바이스 드라이버는 인터럽트가 발생된 이유를 조사한 후, 그에따라 적절한 동작을 취해 인터럽트를 처리한다. ISA 버스의 경우, 각각의 디바이스는 대개 자신만의 유일무이한 IRQ 번호를 필요로한다. PCI 버스 및 기타 특별한 경우에는 IRQ 를 공유하는 것도 허용된다.

2.7. DMA 채널들

DMA 채널들은 전적으로 ISA 버스만을 위한 것이다. DMA 는 "Direct Memory Access(직접 메모리 억세스)" 를 의미한다. DMA 에서는 디바이스가 CPU 로부터 컴퓨터 메인 버스를 넘겨받아 이를 통해 바이트열을 메인 메모리에 직접 전송하는 것이 허용된다. DMA 를 쓰지않는 통상의 경우라면 CPU 는 이러한 전송작업을 다음 2 단계로 처리할 것이다:

1. 해당 디바이스의 I/O 메모리 공간으로부터 바이트열을 읽어 CPU 자신에게 보낸다.
2. 이 바이트열을 CPU 에서 메인 메모리로 보낸다.

1. DMA 를 사용하면 보통 위의 두 단계 과정이 디바이스에서 직접 메모리로 바이트열을 전송하는 한 단계 과정으로 처리된다.

디바이스의 하드웨어에 이 기능이 구비되어 있어야만 가능하므로, 모든 디바이스들이 다 DMA 를 사용할 수 있는것은 아니다. 또 DMA 가 진행되는 중에는 메인 버스가 DMA 전송에 사용되므로, 그동안 CPU 는 별다른 작업들은 할수없게된다.

PCI 버스는 사실 DMA 를 전혀 갖고있지 않지만, 그 대신 DMA 보다도 더 좋은 기능을 가지고 있다. 그것은 버스 마스터링(bus mastering)이라는 것이다. 버스 마스터링의 동작은 DMA 와 다소 유사하므로, 때로는 DMA 라고 불리기도 한다(예를 들면 하드디스크에서는 이를 "UltraDMA" 라고 부른다). 이 기능을 사용하면, 디바이스는 일시적으로 버스의 소유자(bus master)가 되어 마치 자신이 CPU 가 된 것처럼 바이트열을 전송할 수 있다. DMA 와는 달리 버스 마스터링은 채널 번호를 전혀 사용하지 않는다. 왜냐하면 PCI 버스의 구조상, PCI 하드웨

어는 어느 디바이스가 현재의 버스 마스터이고 어느 디바이스가 장차 버스 마스터가 되기를 요청하고있는지를 알수있기 때문이다. 따라서 PCI 버스에 DMA 채널을 할당하는 경우는 없다.

ISA 버스상의 어떤 디바이스가 DMA 를 하려고 할 때에는, 그 디바이스는 인터럽트 요구때와 유사한 개념으로서 DMA 요청용으로 배선된 라인들에다 DMA 요구(DMA-request)를 발하게 된다. 사실, 인터럽트를 이용해 DMA 를 처리토록 할수도 있었겠지만, 그랬었더라면 다소의 시간지연이 불가피했을 것이다. 따라서, DMA 요구라 불리는 특수한 종류의 인터럽트를 쓰는게 더 빠르다. 인터럽트에서와 마찬가지로 DMA 요구들 각각에도 번호가 붙여져 있어, 이를 통해 어느 디바이스가 DMA 를 요구하고있는지를 식별할 수 있다. 이 번호를 DMA 채널이라 부른다. DMA 전송은 모두 메인 버스를 사용하므로(따라서 한번에 한 개의 채널밖에 동작할 수 없다) 사실은 각 번호의 채널들이 모두 동일한 물리적 채널을 사용하는 것이지만, "DMA 채널" 번호를 이용해 현재 "채널"을 사용하고 있는 것이 어떤 디바이스인지 알수 있다. 마더보드상에는 각 "채널"의 현재 상태를 저장하고 있는 하드웨어 레지스터들이 있다. 따라서 어떤 디바이스가 DMA 요구를 발하려면, 그 디바이스는 자신의 DMA 채널 번호를 알고있어야만 하며, 이 번호는 그 물리적인 디바이스상의 특별한 레지스터내에 저장되어져 있어야만 한다.

2.8. 디바이스와 드라이버 양측에 대한 "리소스(resources)"

따라서, 디바이스 드라이버들은 자신이 제어하는 하드웨어와 어떤 방식으로든 연계되지 않으면 안된다. 이는 버스 리소스(I/O, 메모리, IRQ들, DMA 들)를 물리적인 디바이스와 디바이스 드라이버 소프트웨어 양쪽에 할당하는 것으로 이루어진다. 예를 들면, 하나의 시리얼 포트는 (가능한 4 가지 선택중에서) 단 2 개의 자원, 즉 IRQ 하나와 I/O 어드레스 하나만을 사용한다. 이 두개의 값은 디바이스 드라이버와 물리적 디바이스 양측에 모두 제공되어야 한다. 또한, 드라이버(그리고 그 디바이스)는 /dev 디렉토리밑에 특정한 이름을 제공받는다(예를 들면 ttyS1 등). 그 어드레스와 IRQ 번호는 물리적 디바이스에 의해 카드상의 설정 레지스터들 내에(마더보드에 실장된 경우는 마더보드상의 칩내에) 저장된다. 점퍼로 설정하는 경우에는, 점퍼의 조합 자체가 버스 리소스 설정을 디바이스의 하드웨어에 저장시키는 역할을 한다. 하지만 PnP 의 경우에는 대부분 PC 의 전원을 끄면 설정 레지스터의 데이터가 사라지게 되므로, PC 의 전원을 넣을 때마다 매번 버스 리소스 데이터를 각 디바이스에 새로 제공하여야 한다.

2.9. 문제점

PC 의 설계구조상 IRQ, DMA 채널, I/O 어드레스, 메모리 영역의 수에는 제한이 있다. 만일 PC 의 디바이스 수가 몇개 안되고 이들 모두가 표준화된 버스 리소스(가령 특정 디바이스는 특정한 I/O 어드레스와 IRQ 번호를 가진다는 식의 표준)를 가졌더라면, 디바이스 드라이버를 디바이스에 연계시킬 때 아무런 문제도 없었을 것이다. 그랬더라면 PC 상의 각 디바이스는 다른 디바이스와 충돌하지 않는 고정된 리소스를 가졌을테니 말이다. 서로 다른 두 디바이스가 같은 I/O 어드레스를 갖거나 같은 IRQ 를 갖는 일 등도 없었을 것이다. 각 디바이스 드라이버들은 각자의 고유의 어드레스들과 IRQ 등등을 아예 프로그램속에 새겨넣어버렸을 테고, 그렇게만 되었더라면 모든 것이 한결 간단했을 것이다.

그러나 현실은 그렇지 못하다. 오늘날에는 디바이스들의 종류도 무척 많아서 충돌도 빈번할 뿐더러, 같은 종류의 디바이스를 여러개 사용해야할 경우도 생긴다. 예를 들면, 서로 다른 디스크 드라이브를 여러대 달거나, 네트웍 카드도 몇 개씩이나 달아야 하는 등의 경우도 있는 것이다. 이러한 이유에서, 디바이스는 어떠한 어드레스나 IRQ 등으로도 세팅가능해서 충돌을 피할 수 있는 유연성을 갖추어야 한다. 하지만 몇몇 IRQ 들과 어드레스들은 사실상 표준이 된 것도 있다. 가령 클럭이나 키보드에 대한 IRQ 및 어드레스들이 그러하다. 이런 디바이스들은 리소스 설정에 있어 유연성이 필요없다.

버스 리소스 할당에서의 충돌 문제외에, 디바이스 드라이버에게 드라이버의 버스 리소스를 잘못 가르쳐줘도 문제가 생긴다. 예를 들면, 디바이스는 IRQ 5 로 설정되어 있는데 실수로 당신이 설정 파일에 IRQ 4 라고 적어놓는 경우이다. 이것도 버스 리소스 할당 에러의 한 유형이다.

버스 리소스들을 제대로 할당했다면, 물리적인 하드웨어와 그에 대응하는 디바이스 드라이버 사이에 통신 채널이 확립된다. 예를 들면, 어떤 범위의 I/O 어드레스 영역(리소스)이 디바이스 드라이버와 하드웨어 양측에 할당되어진 경우, 양자간에 일방통행의 통신 채널이 확립된다. 드라이버 측은 디바이스에게 명령어와 정보를 보낼 수 있다. 사실은 드라이버측에서 디바이스의 레지스터를 읽어 정보를 얻어오는 방법도 가능하므로, 일방통행보다는 조금 더 나은 것이라 할수 있다. 그러나, 이 방법으로는 디바이스측에서 먼저 통신을 시작할 수는 없다. 디바이스쪽에서 먼저 통신을 개시하려면 디바이스는 자신의 드라이버에게 인터럽트를 보낼 수 있는 IRQ 를 가져야 한다. 이 상태가 되어야 비로소 드라이버나 물리적 디바이스 어느 쪽이든 먼저 통신을 시작할 수 있는 양방향 통신 채널이 구축되는 것이다.

2.10. 시리얼 포트에 꽂은 디바이스를 PnP 가 찾아낸다

시리얼 포트에 케이블로 접속된 외부 디바이스들(예를 들면 외장형 모뎀)도 플러그 앤드 플레이라고 불리운다. 버스 리소스(IRQ 와 I/O 어드레스)를 필요로 하는 것은 시리얼 포트일 뿐으로서, 시리얼 포트에 꽂는 디바이스들에게는 버스 리소스가 할당되지 않는다. 그렇기 때문에 사실 이런 외부 디바이스에게는 PnP 가 필요없는 것이다. 그럼에도 불구하고, 그런 외부 시리얼 디바이스에 대해서도 PnP 명세서가 규정되어 있다.

PnP 운영체제는 이러한 외부 디바이스도 검출하여 그 디바이스의 모델 번호등을 읽어낸다. 그에 의해 해당 디바이스용 디바이스 드라이버를 찾아낼 수 있으므로, 예를 들면 /dev/ttyS1 상에 어떤 디바이스가 붙어있는지 당신이 직접 응용프로그램에게 알려주지 않아도 되게된다. 응용 프로그램에게 디바이스가 접속되어있는 시리얼 포트를 (설정 파일등을 사용해) 손수 알려주는 방법도 가능하므로 (디바이스의 모델 번호를 지정하는 것도 가능할지 모른다), PnP 의 이 "시리얼 포트" 기능이 반드시 필요한 것은 아니다. 리눅스가 이 기능을 지원하는지는 모르겠다. ??

3. PnP(Plug-and-Play) 해결책

3.1. PnP 에 대한 소개

PnP(Plug-and-Play) 라는 용어는 다양한 의미를 가지고 있다. 넓은 의미에서는 단순히 디바이스를 꼽기만 하면 스스로 자신을 설정하는 자동 설정기능을 가리킨다. 본 HOWTO 에서 사용하는 PnP 의 의미는 PnP 버스 리소스를 설정하고 디바이스 드라이버에게 그 설정 내용을 알리는 것만을 의미한다. 좀 더 좁은 의미로는 하드웨어 디바이스상에 버스 리소스를 셋팅하는 것만을 가리킨다. 리눅스의 경우에 한해 말한다면, PnP 는 대개는 그저 바이오스 등등에 의해 디바이스에 세팅된 버스 리소스의 내용을 드라이버가 감지해내는 것을 뜻한다. PnP 는 그저 PnP 규격명세를 뜻할수도 있다. ISA 버스 리소스에 관한 PnP 규격명세는 상당히 길다. 표준 PCI 규격명세(이것을 PnP 라고 부르지는 않는다)는 PnP 에 해당하는 기능을 PCI 버스상에서도 구현하고 있다.

PnP 는 디바이스들을 해당 디바이스 드라이버들에 결부시키고 양측의 통신 채널을 특정한다. PnP 가 등장하기 전의 ISA 버스에서는 점퍼를 이용해 사전에 미리 버스 리소스들을 하드웨어 디바이스상에 셋팅시켰었다. 소프트웨어 드라이버들은 설정화일(또는 그와 유사한 것)에 의해 버스 리소스들을 지정받거나, 아니면 특정 디바이스가 있을법한 어드레스들을 조사해보는 방법을 통해 버스 리소스를 지정받았다. PCI 버스는 애초부터 PnP 와 유사한 기능을 내재하고 있었으나 처음에는 이를 PnP 라 부르지는 않았다. PCI 버스 명세서에서는 PnP 라는 용어를 사용하지는 않지만 오늘날 PnP 라고 부르는 기능을 하드웨어적으로 지원하고 있다.

3.2. PnP 의 동작원리에 대한 간략한 설명

여기서는 PnP 동작원리의 개요를 아주 간략하게 설명하겠다. PnP 설정 프로그램(리눅스에서는 BIOS 만이 이 작업을 수행한다)은 모든 PnP 디바이스들을 검색한 후, 각 디바이스에게 어떤 버스 리소스를 원하는지를 묻는다. 그 다음, 분배해줄 버스 리소스들(IRQ 등등)이 무엇무엇인지를 체크한다. 물론, 이때 PnP 아닌 디바이스(Legacy 디바이스라고 한다)들이 사용하기로 예약된 버스 리소스들만큼은 (BIOS 가 그 버스 리소스들을 알고있는 한) 분배해주지 않아야한다. 그다음, 일정 기준(이 기준은 PnP 명세에 규정된 것은 아니다)에 의해 충돌이 없는 한도내에서 가능한한 각 디바이스들에게 원했던 버스리소스를 배정한다. 그다음, 각각의 물리적 디바이스들에게 어떤 버스 리소스가 배정되었는지를 알려주고, 이에 디바이스들은 그 버스 리소스대로 자신을 셋팅한다. 이렇게 되면, 이제 디바이스 드라이버들은 적당한 방법을 통해 자신의 디바이스가 사용하는 버스 리소스를 알아낸 후, 그를 통해 자신이 제어하는 디바이스와 통신할 수 있게 되는 것이다.

예를 들어 어떤 카드가 인터럽트(IRQ 번호) 하나와 1 MB 의 공유메모리를 필요로한다고 치자. PnP 프로그램은 그 카드로부터 이 요구사항을 읽어본 후, 그 카드에 IRQ 5 와 어드레스 0xe9000000 부터의 1 MB 메모리 어드레스 공간을 할당해준다. 그런데 일이 항상 이렇게 순조로운 것만은 아니다. 예를 들어 그 카드가 자기는 반드시 특정한 IRQ 번호만 써야한다고 주장하거나(ISA 경우에만 이런 일이 생길수 있다), 그 1 MB 메모리는 반드시 어떤 특정한 메모리 어드레스 범위내일 것을 주장하는 수가 있다. 자세한 사항은 PCI 버스냐 ISA 버스냐에 따라 각각 다른데, ISA 경우가 더 복잡하다.

이 경우 PnP 설정 프로그램이 취할수 있는 간편한 방법이 몇가지 있다. 그 하나는 컴퓨터를 마지막으로 사용했을 때의 최종적인 버스 리소스 설정상태를 기록해 두었다가 그것을 그대로 재사용하는 것이다. 윈도우즈 9X 와 PnP BIOS 들은 이 방법을 사용하지만 표준 리눅스는 이 방법을 사용하지 않는다. 윈도우즈 9x는 이 설정상태 정보를 자신의 하드디스크상의 "레지스트리"에 기록해 두며, PnP BIOS 는 이 정보를 PC 내의 비휘발성 메모리(ESCD 라고 부른다. 5.3.2절 를 보라)에 기록해 둔다.

MS 윈도우즈는 (윈도우즈 95 부터) PnP OS 인 반면, 리눅스는 PnP OS 가 아니다. 하지만 리눅스에 있어서도 BIOS 가 버스 리소스 설정을 해주고 디바이스 드라이버들이 (리눅스 커널이 제공하는 프로그램을 써서) BIOS 가 설정한 그 내용을 알아내게되므로 PnP 가 다소 역할을 한다고 할수있다. 또한, 드라이버들은 그런 프로그램들을 이용해 지정된 버스 리소스 값을 변경시켜버릴수도 있다(하지만 이때 다른 디바이스용의 버스 리소스를 침해할지도 모른다). 어떤 디바이스 드라이버들은 자신이 최종 사용한 설정내용을 저장했다가 다음번 컴퓨터 사용시에 그를 사용하기도 한다.

디바이스 하드웨어가 자신의 지난번 설정을 기억하고 있다면야 다음 부팅때에 하드웨어의 리소스 설정을 두고 고민할 필요가 전혀 없겠지만, 하드웨어들은 보통 전원이 꺼짐과 동시에 자신의 설정상태를 잊어버리게 된다. 일부 디바이스들은 디폴트 설정을 가지고 있을수도 있다(하지만 그 디폴트 설정상태가 지난번 최종사용시의 설정상태와 같으란 법도 없다). 따라서 PnP 설정 프로그램은 PC 를 켤때마다 실행되어야 한다. 또한, 만일 새로운 디바이스를 추가한다면 그 디바이스 역시 설정될 필요가 있다. 이 새로운 디바이스에 버스 리소스를 할당하는 작업은 어쩌면 현재의 디바이스로부터 버스리소스 일부를 빼앗아 새 디바이스에 주고 그대신 원래의 디바이스에게는 다른 버스 리소스를 할당해주는 작업이 될 수도 있다. 현재로서는 리눅스는 이런 작업은 할수 없다.

3.3. PC 의 기동

PC 를 처음 켜면, BIOS 칩은 컴퓨터를 시작시키기 위해서 BIOS 프로그램을 실행한다(첫 단계는 하드웨어의 체크이다). 운영체제가 하드 디스크에 저장되어 있는 경우(보통은 그렇다)라면, BIOS는 먼저 그 하드 디스크에 대한 사항을 알아야만 한다. 하드 디스크가 PnP 기기라면, BIOS 는 이를 찾아내기 위해 PnP 방법을 사용할 수도 있다. 또한, 컴퓨터 시작시에 사용자가 BIOS 의 CMOS 내용을 수동으로 설정할 수 있도록 하거나 컴퓨터 기동시의 에러 메세지에 대응할 수 있도록 하기 위해서는, 스크린(비디오 카드)와 키보드도 필요하다. 따라서, BIOS 는 이들 디바이스들만큼은 언제나 PnP 설정시켜야 하는 것이다.

일단 BIOS가 하드 디스크, 비디오 카드, 키보드를 인식하면, 이제 부팅(하드 디스크로부터 메모리로 운영체제를 로드하는 것)개시 준비는 완료된 것이다. 만일 BIOS 설정메뉴에서 PnP 운영체제(PnP OS)를 사용하는것으로 설정했다면, 앞서 말한 바대로 BIOS 가 PC의 부팅을 개시한 후, 이제 운영체제가 나머지 PnP 설정작업을 마무리짓게 된다. BIOS 를 PnP OS 사용않음으로 설정했다면 운영체제의 부팅에 앞서 PnP-BIOS 자신이 디바이스들의 PnP 설정을 끝까지 마무리지으려 할 것이다(하지만 BIOS 로서는 드라이버들에게 그 설정내용을 직접 알려주지는 못한다). 이상이 리눅스 기동시에 일반적으로 발생하는 일들이다.

3.4. 버스

ISA 는 구형 IBM PC 에서 사용하던 옛날 버스인 반면, PCI 는 인텔이 제안한 새롭고 빠른 버스이다. PCI 버스는 현재 우리가 PnP 라고 부르는 기능을 실현할 수 있도록 설계되었다. ISA 버스에 비해 PCI 버스에서는 하드웨어 디바이스들에 PnP 버스 리소스들이 어떻게 할당되었는지를 알아내기가 용이하다. 설정상태를 알아보려면 `lspci` 나 `scanpci`(X 윈도우용 명령어) 명령을 사용하거나, `/proc/pci` 또는 `/proc/bus/pci` 파일을 보면 된다. 부팅시에 화면에 표시되는 메시지(지나간 메시지를 보기위해서는 Shift-PageUp 을 사용한다)도 유용하다. 7.1절를 참조하라.

ISA 버스의 경우에는 PnP 를 구현하는데 있어 정말이지 어려운 문제가 있었다. 이러한 사태가 벌어진 이유는 ISA 버스를 설계할 당시에는 누구도 PnP를 고려하지 않았다는 점과, PnP 가 설정정보를 물리적 디바이스로 보내는데 사용할 수 있는 I/O 어드레스 영역이 거의 없다는 점때문이다. 결국, ISA 버스상에서 PnP 를 구현하는 방법은 매우 복잡한 것이 되었다. 이 주제에 대한 책이 몇권 있다. 5.8절 를 참고하라. 무엇보다도, ISA 버스에서의 PnP 구현시에는 각각의 PnP 디바이스는 PnP 프로그램에 의해 일시적인 "핸들(handle)"을 할당받아야만 한다. 그래야 이를 통해 그 디바이스에 PnP 설정을 위한 어드레스를 줄수있게 된다. 이 "핸들" 들을 할당하는 것을 "Isolation" 이라고 한다. 보다 복잡한 세부내용은 9.6절을 참조하라.

결국 ISA 버스는 언젠가 사라지게 될것이다. 그렇게되면 하드웨어를 BIOS 가 어떻게 설정했는지를 간단히 알아낼수 있게되므로 PnP 는 훨씬 쉬워질 것이다. 하지만 그래도 디바이스 드라이버와 디바이스를 일치시키는 작업과 PC 의 시작 및 실행시에 새로 추가된 디바이스를 설정하는 작업만큼은 언제까지나 필요할 것이다. 이러한 작업들은 리눅스가 PnP 운영체제가 된다면 보다 간단히 완료될수 있을 것이다.

3.5. 리눅스가 PnP 를 구현하는 방법

리눅스는 지금껏 PnP 를 다루는데 있어 심각한 문제점을 가지고 있었고 아직도 문제를 안고는 있다. 하지만 더이상 과거처럼 치명적인 문제는 아니다. 리눅스는 아직도 진정한 PnP 운영체제는 아니며, 디바이스용 버스리소스들을 설정하기위해 디바이스 드라이버들과 PnP BIOS 에 주로 의존한다고 볼수있다. 하지만 커널은 드라이버들의 요청이 있을 경우 PnP 프로그램을 제공하는 형식으로 드라이버들을 도와준다. 많은 경우, 필요한 설정작업은 디바이스 드라이버가 전부 해낸다. 그밖의 경우에는, BIOS 가 설정작업을 담당한 다음, 디바이스 드라이버는 바이오스가 설정한 내용을 찾아보는 방법을 사용할수도 있다. 커널은 드라이버에게 디바이스의 존재여부 확인, 설정상태 확인, 설정상태 변경 등의 몇가지 함수들(프로그램 코드이다)을 제공할 수도 있다. 커널 2.2 에서는 PCI 버스에 대해서만 이 기능이 있었으나, 커널 2.4 부터는 ISA 버스와 PCI 버스 둘다에 대해서도 이 기능이 가능하다(단, 커널 컴파일시에 해당하는 PNP 옵션들이 선택되어야 가능하다). 그렇지만 커널이 이런 기능을 가지고 있다고해서 모든 드라이버들이 이 기능을 정확하게, 또 완전히 활용하고 있다는 뜻은 아니다.

게다가, 커널은 두 디바이스들이 동시에 같은 버스 리소스를 사용하는 것을 허용치 않음으로써 리소스의 충돌이 일어나지 않게 도와준다. 원래 이것은 IRQ 들 간의 충돌이나 DMA 들간의 충돌만을 방지하기위한 기법이었으나 이제는 어드레스 리소스들 간의 충돌을 방지하기위해서도 사용되고있다.

커널 2.4 이전에는 독자적인 프로그램인 isapnp 를 이용해 ISA 버스상의 PnP 디바이스를 설정을 하거나 그 정보를 얻어내는 수가 많았다. 아직도 isapnp 는 ISA 버스상의 PnP 구현이 불완전한 디바이스 드라이버에게는 필요하다. 리눅스를 진짜 PnP 운영체제로 만들려는 시도가 적어도 한번은 있었다.<65533> (http://www.astarte.free-online.co.uk) 를 참고하라. 하지만 이 시도가 커널에 포함된 적은 없었다.

커널이 디바이스 드라이버들에게 제공할수 있는 도움에는 어떤 것들이 있는가는 커널 문서를 보라. 이 문서는 (당신이 가지고 있다면) /usr/.../.../Documentation 디렉토리에 있다. 앞의 ... 기호중에는 "kernel" 이라는 단어가 포함되어있다. "locate" 명령으로 찾을수 있다. 이 문서 디렉토리내의 pci.txt ("How to Write Linux PCI Drivers") 파일과 /usr/include/linux/pci.h 파일을 보라. 이 파일들은 상당히 간결하기 때문에 당신이 드라이버에 대해 도사 수준이 아니고 C 프로그래밍도 모른다면 드라이버 작성법을 배우는 데는 사실상 도움이 되지 못할 것이다. 하지만 드라이버가 이용할 수 있는 PnP 용 함수들에 어떤 것들이 있는지 다소 알수 있을 것이다. ISA 버스에 대한 것은 isapnp.txt 문서와 커널 2.4 의 경우라면 /usr/include/linux/isapnp.h 도 도움이 될 것이다.

PC 가 기동될때 화면에 나타나는 메시지중에는 몇몇 리눅스 디바이스 드라이버들이 자신의 하드웨어 디바이스를 (그리고 BIOS 가 그에 할당시킨 버스리소스들도) 발견했다는 내용도 있을수 있다. 하지만 진정한 PnP 운영체제라면 다음의 상태들까지도 더 잘 처리할 수 있을 것이다.

- 버스 리소스들이 부족한 경우의 할당문제
- 1 개의 물리적 디바이스에 대해 여러개의 드라이버들이 있는 경우의 처리
- (각 드라이버들이 디바이스를 찾아헤메는 대신) 검출해낸 디바이스에게 해당 드라이버를 찾아주는 일
- 디바이스 드라이버 프로그래머들의 수고를 덜어주는 일

"버스 리소스들의 부족" 문제는 다음 두가지 이유로 점차 문제거리가 안되어 가고있다 : 첫째, PCI 버스가 ISA 버스를 대체해가고 있다. PCI 에서는 IRQ 를 공유할 수 있기 때문에 비록 공유로 인해 효율이 떨어지기는 해도 IRQ 가 부족해지는 일은 없다. 또한, PCI 는 DMA 리소스를 사용하지 않는다(DMA 리소스 없이도 DMA 와 동등한 기능을 구현한다).

둘째, 점점 더 많은 물리적 디바이스들이 IO 어드레스 공간 대신 메인 메모리 어드레스 공간을 사용하는 쪽으로 가고있다. 32 비트 PC 들은 4GB 의 메모리 어드레스 공간을 쓸 수 있으므로, 당신 시스템이 4 GB 의 메인 메모리를 사용하지 않는 한, 이 메인 메모리 어드레스 버스 리소스의 상당 부분을 IO 디바이스로 사용할 수 있다. 이에 비해 IO 어드레스 공간은 겨우 64 KB 가 한계이다. 따라서 디바이스의 IO 용으로 쓸수 있는 메모리 공간은 (아직까지는?) 부족하지 않다.

4. PnP BIOS 의 설정

컴퓨터를 켜면 운영체제가 로드되기에 앞서 BIOS 가 실행된다. 최근의 BIOS 들은 PnP BIOS 라서 일부 또는 전부의 PnP 디바이스들을 설정할 수 있다. BIOS 의 CMOS 메뉴에서 다음 항목들을 선택할 수 있을 것이다 :

- 4.1절
- 4.2절
- 4.3절

4.1. PnP 운영체제 여부

이 항목의 가부에 상관없이 PnP BIOS 는 하드 디스크, 비디오 카드, 키보드에 대해서는 PnP 설정을 할것이다. 그래야 시스템을 부팅시킬수 있기 때문이다. 이 항목을 PnP OS 를 가진 것으로 설정했다면 PnP BIOS는 그 설정 작업의 마무리를 운영체제(또는 디바이스 드라이버들)에 맡긴다. PnP OS 없음으로 설정했다면 BIOS 가 설정작업을 전부 담당할 것이다. PC 에 리눅스만 깔았다면 순순히 PnP OS 없음으로 설정하면 될것이다. 허나, 만일 PC 에 MS 윈도우즈가 같이 깔려있고 그 윈도우즈 설치시에 PnP OS 있음으로 설정해서 설치한 것이라면, 윈도우즈 95/98 을 계속 기쁘게 하기 위해서는 PnP OS 있음으로 설정해야만 할 수도 있다. (하지만 PnP OS 있음으로 설정한다면 리눅스에서 문제가 생길수도 있다. 윈도우즈 2000 의 경우는 설사 당신이 PnP OS 없음으로 BIOS 설정을 바꾸어도 제대로 동작한다고 알려져있다. 이경우 윈도우즈 2000 은 새로운 하드웨어를 찾는 중이라는 메시지를 출력할 것이다(윈도우즈 2000 으로서는 이미 그 하드웨어에 대해서는 알고있지만, BIOS 가 그 하드웨어를 어떻게 PnP 설정했는지까지는 알지못하기 때문이다)).

만일 PnP OS 있음으로 설정했다면, 이제 버스 리소스 설정을 다루기 위해서는 리눅스 디바이스 드라이버들 및 경우에 따라서는 isapnp 프로그램에 의지해야 한다. 대부분의 경우 이들이 제대로 작업해내겠지만 때로는 그렇지 못할수가 있다. 이때에는 아래에 나올 내용처럼 리눅스를 속여서 문제를 해결할 수도 있다. 이러한 속임수는 BIOS 가 자신의 작업을 제대로 해내지 않았어도 리눅스가 독자적으로 그 작업을 해냄으로써 가능한 것이다.

만일 PnP OS 없음으로 설정했다면, 이제 BIOS 가 설정작업을 하게된다. 새로운 PnP 디바이스를 추가하지 않는 이상, BIOS 는 자신의 비휘발성 메모리(ESCD)에 기록해두었던 설정내용을 사용할 것이다. 5.3.2절을 보라. 지난번 마지막에 사용했던 것이 리눅스였다면 BIOS 내의 메모리에 기록된 설정내용은 변함이 없다. 5.3절을 보라. 하지만 마지막으로 사용했던 것이 윈도우즈 9x(PnP) 이라면 윈도우가 ESCD 의 내용을 변경시켰을수 있다. 당신이 어떤 설정을 "강제"했거나 레가시(legacy) 디바이스를 설치한 경우라면 으레 ESCD 는 변경되게된다. 5.3.3절를 보라. 디바이스 드라이버들이 설정작업을 하면서 BIOS 가 작업한 내용을 변경시키는 수도 있다. isapnp 나 PCI 유틸리티 프로그램들도 그런 변경을 일으킬 수 있다.

4.1.1. 윈도우와의 이중 운영체제 시스템

Linux 와 윈도우즈를 같은 PC 상에서 사용하고 있는 경우, BIOS의 "PnP OS 를 사용하고 있는가" 라는 질문에 어떻게 답하면 좋을까? 상식적이고 정직한 답은 "Linux 에 대해서는 no, 윈도우즈에 대해서는 yes" 일 것이다. 당신이 정직하게 행동하기만 하면 세상 모든 일도 순조롭게

진행된다. 하지만, 다른 OS 를 쓰려고 할 때마다 BIOS 의 CMOS 메뉴를 일일이 수동으로 바꿔줘야 한다는 것은 너무도 귀찮은 일이다. 이 문제의 해결책은 리눅스나 윈도우즈 둘 중 한쪽에 거짓말을 하는 것이다. 한쪽을 무시함으로인해 다소 문제가 생길지 모르지만, 그 문제에는 또 적당한 해결책이 존재한다.

어느 한쪽을 무시하기로 작정했다면, 그 대상이 될 윈도우즈 혹은 리눅스에게 속인다는 사실을 알려주어야 하는가? 이는 그러한 무시에도 불구하고 리눅스나 윈도우가 얼마나 스스로를 잘 설정해내는지에 달렸다. 윈도우즈 2000 과 리눅스는 이렇게 자신이 무시되는 상황에 대해서도 점차 잘 대처해 나가고있다.

4.1.2. 리눅스쪽을 무시할 경우

PnP OS 있음으로 설정되면, 모든 드라이버들과 isapnp(만일 사용한다면)가 설정작업을 제대로 해낼 때에 한해 리눅스가 제대로 동작할 것이다. 아마도 리눅스 OS 나 드라이버들을 업데이트한다면 도움이 될 것이다.

4.1.3. 윈도우즈 9x 쪽을 무시할 경우

또다른 해결책은 CMOS 에서 PnP OS 없음으로 설정한 후 윈도우를 시작하는 것이다. 이제 윈도우즈 쪽이 무시되므로, 윈도우즈로서는 PnP 를 다루는 것이 훨씬 더 복잡해지게 된다. 혹자는 이제 하드웨어의 설정작업을 BIOS 가 완전히 처리해 버릴테니 윈도우즈 9x 로서는 그만큼 쉽게 하드웨어에 대응할수 있을 것이라 예상하겠지만, 실제로는 윈도우즈는 그렇게는 대응하지 못한다. 그러나 윈도우 2000 만큼은 제대로 대응하는 것으로 알려져있다. 혹자는 비록 하드웨어가 이미 BIOS 에 의해 설정완료되었다는 사실을 윈도우즈 9x 가 모른다고 하더라도, 윈도우즈는 자신의 레지스트리의 내용대로 물리적 디바이스들을 완전히 다시 설정해버릴 테니까 만사 OK 라고 생각할지도 모르겠다.

그러나 결과는 그런 식으로 진행되지는 않는듯 하다. 윈도우즈 9x 는 자신의 디바이스 드라이버들에게 윈도우의 레지스트리에 저장된 내용들을 그저 알려줄 뿐인 경우도 있는듯 하다. 하지만 BIOS 가 담당한 실제의 하드웨어 설정은 ESCD 에 저장되어 있던 내용대로이고, 따라서 레지스트리의 내용과는 다를수 있다. 이점이 문제인 것이다. 그러므로, 윈도우즈가 제대로 동작하기 위해서는 BIOS 가 ESCD 로부터 작성한 버스 리소스 설정내용을 레지스트리도 똑같이 가지게 만들 필요가 있다.

레지스트리와 ESCD 가 같은 내용을 가지도록 하는 한가지 방법은, BIOS 를 "PnP OS 없음" 으로 설정한 상태에서 윈도우를 설치(혹은 재설치)하는 것이다. 그러면 윈도우즈에게는 BIOS 가 설정완료시킨 하드웨어들이 주어지게 된다. 그 설정상태 자체에 충돌이 없는한, 바람직스럽게도 윈도우즈는 자신의 레지스트리에 하드웨어 설정상태를 변경없이 그대로 저장하게 된다. 이렇게해서 ESCD 와 레지스트리가 일치하게 되는 것이다. 이 방법이 제대로 먹힌다면(그리고 본 문서의 버전이 가장 최신버전이라면) 필자에게도 알려달라. 왜냐하면 필자는 이 방법이 성공했다는 보고를 단 한건 받아보았기 때문이다.

다른 방법은, 윈도우즈에서 문제를 일으키는 디바이스들을 디바이스 매니저(장치관리자)에서 "삭제"하는 것이다. 그다음, "PnP OS 없음" 상태로 PC를 재시작한다(부팅시에 CMOS 에서 세팅을 변경한다). 그러면 윈도우즈는 그 디바이스들을 재설치할 것이고, 기대한 바대로

BIOS가 설정한 버스 리소스 설정내용을 사용할 것이다. 이때 윈도우즈는 당신에게 윈도우 설치 CD 를 넣으라고 요구하기 십상이다. 아직 드라이버 파일들을 가지고 있음에도 불구하고 윈도우가 왕왕 그것을 찾지 못하는 수가 있기 때문이다. 이경우 "무시(skip file)"를 선택하고 계속 진행하는 것도 편법이 될 수 있다.

필자는 시험삼아 노벨 호환의 드라이버를 사용했던 NIC 카드를 "삭제" 했다. 재부팅해보니 윈도우는 노벨 대신 마이크로소프트 네트워킹으로 재설치해버렸다. 이런 경우는 노벨 클라이언트를 재설치해야 할 것이다. -- 상당히 귀찮은 작업들이다.

4.1.4. 윈도우즈 2000 쪽을 무시할 경우

윈도우즈 2000 의 경우에는, 처음에 BIOS 를 PnP OS 있음으로 설치한 후 나중에 PnP OS 없음으로 바꿔도 문제없이 동작한다고 알려져있다. PnP OS 없음으로 설정을 바꾸게 되면 윈도우즈 2000 은 자동적으로 자신의 디바이스들을 PnP 재설정하는 것으로 알려져있으며, 새로운 하드웨어를 발견됐고 새로운 디바이스를 설치하는 중이라고 당신에게 알려올 것이다. 이는 윈도우즈 2000 은 BIOS 에 의해 설정되지않은 하드웨어를 탐색하기에 앞서 먼저 BIOS 가 설정을 마친 하드웨어부터 탐색한다는 것을 의미한다. 윈도우즈 2000 은 아마도 종전과 다른 어드레스 및 IRQ(BIOS 가 셋팅한 것) 에서 어떤 하드웨어를 발견하게 되었으므로 그를 "새로운" 하드웨어라고 판단하는 듯 하다.

4.2. 버스 리소스의 조정

이 항목은 IRQ 와 DMA 버스 리소스들을 어떻게 할당할지 결정한다. BIOS 의 이 항목을 "자동" 으로 설정하면 BIOS가 할당작업을 수행한다. "수동" 으로 설정하면, 사용자가 직접 "legacy" (non-PnP) ISA 카드를 위해 IRQ를 몇개 예약할 수 있다. 사용자가 지정해주지 않는 경우, BIOS 는 그러한 legacy 카드에 대해 인식할수도 있고 못할수도 있다. BIOS 가 legacy 카드들에 대한 사항을 인식할 수 있는 경우란, 오로지 사용자가 윈도우즈상에서 ICU(또는 비슷한 것)을 실행시켜 BIOS에게 그 정보를 알려주는 경우 뿐이다. BIOS 가 legacy 카드들의 정보를 알고있다면 "자동" 으로 설정해보라. BIOS가 legacy 카드들에 대해 알지 못하는 경우에는 그 legacy ISA 카드들이 필요로하는 IRQ 들을 직접 수동으로 예약해준 후 나머지들은 BIOS PnP 가 할당하도록 맡겨두면 된다.

4.3. 설정내용의 리셋(reset)

이 항목은 legacy (non-PnP) 디바이스들의 설정 및 PnP 디바이스들의 설정상태에 대한 BIOS 의 ESCD 데이터베이스를 지워버릴 것이다. 이 데이타베이스가 잘못되서 재작성될 필요가 있다고 확신할 때 외에는 절대 이 항목을 건드리지 말라. 메뉴의 어딘가에 오직 사용자가 컴퓨터를 부트시킬 수 없을 때에만 이 항목을 사용해야 한다고 명시되어 있을 것이다. BIOS 가 legacy 디바이스들에 대한 데이터를 잃은 경우에는, DOS 나 윈도우즈에서 ICU 를 실행시켜 이 데이터를 새로 작성할 수 있을 것이다.

5. PnP 카드를 다루는 방법

5.1. PnP 카드를 다루는 방법에 대한 개요

오늘날 대부분의 신형 내장형 보드(카드)들은 PnP 이다. PnP 를 처리하는 방법은 아래처럼 다섯가지 다른 방법이 있다(하지만 당신 상황에 따라서는 일부 방법은 불가능할 것이다). 만일 디바이스 드라이버가 PnP 설정을 맡는다면 당신은 아무것도 손댈 필요가 없다. 만일 BIOS 가 PnP 설정을 맡는다면 당신으로서는 부디 해당 드라이버가 BIOS 가 설정한 내용을 발견해 주기를 기원할 수 밖에 없다(필요하다면 설정화일등을 이용해 당신이 직접 드라이버에게 설정내용을 알려주어야 한다).

- 5.2절
- 5.3절(PCI 버스의 경우라면 그저 PCI BIOS 만 있으면 되나, 그렇지 않다면 PnP BIOS 가 필요하다)
- 5.4절 점퍼나 도스/윈도우즈 소프트웨어를 사용한다(하지만 이렇게 할수 없는 카드들도 많다).
- 5.5절 이것은 언제나 ISA 버스상의 PnP 디바이스들을 설정하는데에만 사용되는 프로그램이다.
- 5.6절 이것은 PCI 버스를 설정하는데 사용되나 디바이스 드라이버가 이 프로그램을 사용한다.
- 5.7절 이 방법을 사용한 후 도스/윈도우즈로부터 리눅스를 부트시킨다. 최후의 방법으로 사용된다.

위의 어느 방법으로든 그 하드웨어의 버스 리소스를 설정할수 있지만 오직 첫번째 방법만이 드라이버측에게 설정내용을 알려준다. 드라이버가 어떻게 정보를 받아들이는가는 드라이버에 따라 다르다. 이를 위해서 따로 어떤 작업을 해주어야 할 수도 있다. 6절 를 보라.

5.2. 디바이스 드라이버가 설정작업을 담당하는 경우

많은 디바이스 드라이버들이 (커널이 제공하는 코드의 도움으로) PnP 방법을 써서 자신이 제어할 하드웨어의 버스 리소스를 셋팅할수 있다. 드라이버 스스로가 설정해버리므로 드라이버는 그 설정내용을 알수 있으며, 따라서 당신이 직접 그 정보를 가르쳐 줄 필요가 없다. 분명 이 방법이 가장 쉬운 방법이다. 드라이버가 다 해낸다면야 당신이 아무일도 할 필요가 없을테니까.

PCI 디바이스의 경우에는 대부분의 드라이버들이 PnP 설정을 해내지만, ISA 디바이스의 경우는 PnP 설정에 문제가 많다. 이는 (비록 과거에 PCI 에 대한 PnP 설정을 "PCI 설정" 이라고 불렀고 아직도 그렇게 부르기는 해도) 원래 PCI 에는 PnP 기능이 내재되어 있기 때문이다. ISA 의 경우, 2.4 버전 이전의 커널은 ISA 를 PnP 설정하기 위한 함수를 일체 제공하지 않았다. 따라서 만일 당신이 최신 버전의 커널과 드라이버를 가지고 있다면 그 드라이버는 PnP 버스

리소스 설정을 해낼 가능성이 더 높다. 하지만 만일 구버전을 가지고 있거나 그 드라이버의 담당자가 PnP 기능을 아직 넣지 않았다면 그 드라이버는 아마도 PnP 설정을 못할 것이다.

불운하게도, 어떤 드라이버는 다른 디바이스들이 필요로하는(그러나 아직 커널에 의해 할당받지는 못한) 버스 리소스를 점유하고 있을지도 모른다. 따라서, 모든 리소스 요청들을 받아들인 후 커널이 그 할당작업을 해낼수 있어야 비로소 진정한 PnP 리눅스 커널이라 할수있을 것이다. 3.5절 를 보라.

5.3. BIOS 가 PnP 를 설정

5.3.1. BIOS 로 PnP 를 설정하는 방법에 대한 개요

PnP BIOS 가 장착되어있다면, 그 PnP BIOS 가 하드웨어를 설정할 수 있다. 이는 BIOS 가 모든 디바이스들의 리소스 요구내용을 읽은 후 각각을 설정(즉, 디바이스들에게 버스 리소스를 할당)해준다는 의미이다. BIOS 로서는 드라이버들을 각각의 해당 디바이스들에게 연결시켜주지 못하고 드라이버들에게 BIOS 가 설정한 내용들을 알려주지도 못한다는 두가지만 빼고는, BIOS 가 PnP OS 의 기능을 대신하는 것이다. BIOS 는 보통 자신의 비휘발성 메모리(ESCD) 내에 저장해 두었던 설정내용을 이용하게된다. BIOS 가 새로운 디바이스를 발견하거나 충돌이 있음을 감지하게 되면, BIOS 는 설정내용에 필요한 변경을 가하게 되고, 이제 ESCD 에 기록되어있던 예전의 설정내용은 사용하지 않게 된다. 이경우 BIOS 는 새로운 상태를 반영하기 위해 ESCD 를 갱신시켜야한다.

당신의 BIOS 는 위에서 말한 것과 같은 설정작업을 수행해야 하지만, 간혹 이 작업을 올바르게 수행하지 못하거나 불완전하게 수행하는 BIOS 들도 있었다. BIOS 를 사용하는 이점은, 대부분의 경우 그밖의 다른 설정작업이 필요없어져(BIOS 의 CMOS 메뉴에서 PnP OS 없음으로 설정한 경우는 제외) 일이 간단해 진다는 점에 있다. 많은 디바이스 드라이버들이 자동으로 BIOS 가 해놓은 작업내용을 감지할 수 있는 반면, 일부의 경우에는 사용자가 드라이버들에게 그 내용을 정해줄 필요가 있을 수도 있다(이게 반드시 쉽지만은 않다). 7절 를 보라. 또다른 이점은 리눅스가 시작되기 전에 BIOS 가 설정작업을 하게되므로, 장차 디바이스 드라이버들이 시작되기 이전에 버스 리소스들은 디바이스 드라이버들에 의해 사용(즉, 발견될) 준비가 완료되어있다는 점이다.

MS 에 따르면 PnP BIOS 가 (MS 윈도우의 도움없이) 디바이스들을 독자적으로 PnP 설정할 수 있는 기능은 단지 옵션사항일 뿐 필수사항은 아니라고 되어있다. 하지만 대략 1996 ?? 이후에 제작된 대부분의 BIOS 들은 모두 이 기능이 가능한 것 같다. BIOS 들이 이 기능을 제대로 수행해낸다면 우리로서는 제작사에 감사편지라도 보낼 일이다. BIOS 는 PCI 버스와 ISA 버스 양자 모두에 대해 설정작업을 수행할 수 있다. 하지만 좀 오래된 BIOS 들은 오직 PCI 에 대해서만 설정작업할 수 있는 것으로 보고되고 있다. 당신이 가진 BIOS 에 대해서 더 자세히 알고 싶다면 웹페이지를 뒤져보라. 필자는 그에 대해서는 아는 바 없으므로 부디 필자에게 질문하지는 말아달라. BIOS 에 관한 자세한 내용은 찾기 힘들거나 이용불가능할지도 모른다. 일부 BIOS 들은 최소한의 PnP 기능만 가지고 있어서 운영체제가 나머지 설정작업을 완료해줄 것을 기대하는 수도 있다. 만약 그러한 경우라면, PnP BIOS 를 이용하는 방법이 아닌 다른 방법을 찾아보거나, BIOS 가 ESCD 데이터베이스를 가지고 있는 경우 그것을 셋업해보는 방법을 시도해보라. 다음 절의 내용을 참고하라.

5.3.2. BIOS 의 ESCD 데이터베이스

바이오스는 PnP 설정내용을 보관하는 비휘발성 데이터베이스를 유지하고 있으며 그 내용을 이용해 설정작업을 수행한다. 이를 ESCD(Extended System Configuration Data) 라고 부른다. 또한, ESCD 의 장착여부는 옵션사항이지만 대부분의 PnP BIOS 들은 이를 장착하고 있다. ESCD 는 충돌이 일어나는 것을 피하기 위해 PnP 디바이스들의 리소스 설정뿐 아니라 non-PnP 디바이스들의 설정정보까지도 저장한다(그리고 디바이스가 PnP 기기인지 여부도 기록하고 있다). ESCD 데이터는 보통 어떤 칩속에 저장되며 전원을 꺼도 그대로 보존된다. 하지만 때로는 하드디스크상에 저장되는 경우도 있다 ??

ESCD 는 최종사용된 설정정보를 저장하기 위한 것이지만, 만일 당신이 리눅스의 isapnp 나 pci 유틸리티등의 프로그램을 사용한다면(이들은 ESCD 를 갱신시켜주지는 않는다), ESCD 로서는 이들이 변경시킨 설정내용은 알리 없으므로, 그 정보를 ESCD 내에 저장하지 않을 것이다. 좋은 PnP OS 라면 이때에도 ESCD 를 갱신시킬 것이고, 그렇게만 된다면 후에 PnP 아닌 운영체제(예를 들면 표준 리눅스)도 이 갱신된 ESCD 정보를 이용할 수 있을 것이다. MS 윈도우는 오직 특별한 경우에만 ESCD 를 갱신시켜줄 뿐이다. 5.3.3절 를 보라.

ESCD 에 기록된 내용을 이용하려면 BIOS 의 CMOS 메뉴에서 반드시 "PnP OS 아님"으로 설정해야 한다. 그러면 (리눅스 운영체제가 로드되기 전에) BIOS 는 시작시에 다음과 같이 설정작업을 수행하게된다. 만일 BIOS 가 ESCD 에 없는 새로운 PnP 카드를 감지한다면, 버스리소스를 카드에 할당한 후 ESCD 를 갱신한다. 이때 심지어는 기존의 PnP 카드들에 할당됐던 버스 리소스들까지도 변경될 수 있으며, 그에따라 ESCD 를 수정한다.

만일 각 디바이스가 자신의 최후의 설정내용을 자체 하드웨어내에 저장해둘수만 있다면, PC 를 시작할 때마다 매번 설정작업을 해줄 필요도 없었을 것이다. 그러나 현실은 그렇지 못하다. 따라서 PnP 를 위해 BIOS 를 사용하겠다면 ESCD 의 데이터는 항상 올바르게 유지되어야 한다. 어떤 BIOS 들은 비록 ESCD 기능은 아니지만 non-PnP 카드용으로 예약해둔 버스리소스들에 대한 정보를 소량의 비휘발성 메모리속에 저장해두기도 한다. 많은 BIOS 들은 이 두가지 기능을 다 가지고 있다.

5.3.3. 윈도우즈를 사용해서 ESCD 를 설정하기

BIOS 가 당신이 원하는 대로 ESCD 를 설정하지 못거나 올바르게 ESCD 를 설정하지 못할 경우, 리눅스 유틸리티를 써서 ESCD 를 세팅할수 있다면 좋을 것이다. 하지만 그런 리눅스 유틸리티는 1999 년초에는 없었으며 2001 현재에도 필자는 그런 것이 있다는 말을 들어보지 못했다. 따라서 (같은 PC 상에 윈도우즈도 깔려있다면) 윈도우즈를 써서 ESCD 를 설정하는 방법을 시도해 볼 수 있다.

윈도우즈를 써서 ESCD 를 설정/수정하는 방법은 3 가지가 있다. 하나는 도스나 윈도우즈 3.x 용으로 디자인된 ICU 유틸리티를 사용하는 것이다. 이 방법은 윈도우즈 9x/2k 에서도 가능할 것이다 ??. 또다른 방법은 윈도우즈 9x/2k 상에서 수동으로("강제로") 디바이스들을 설정해서 정상 종료시에 윈도우즈가 이 정보를 ESCD 내에 기록하게 만드는 것이다. 세번째 방법은 PnP 가 아닌 legacy 디바이스들에게만 해당된다. 윈도우즈가 legacy 디바이스의 존재와 그들이 사용하는 버스 리소스가 무엇인지를 파악한다면 윈도우즈는 그 정보를 ESCD 에 기록할 것이다.

만일 PnP 디바이스들이 사용자의 "강제" 적인 설정변경없이 윈도우즈에 의해 자동으로 설정되도록 된 경우라면, 그러한 설정내용은 아마도 ESCD 로 기록되지 않을 것이다. 물론 ESCD

에 기록된 내용대로 디바이스들을 설정할 지 여부는 윈도우즈가 스스로 결정하기 때문에, 디바이스들이 과연 ESCD 에 기록된 설정내용에 따라 기동될지는 우연적이라 할수 있다.

윈도우즈 9x 는 PnP 운영체제라서 디바이스들을 자동으로 PnP 설정한다. 윈도우즈 9x 는 자체의 PnP 데이터베이스를 이진 윈도우즈 파일들로 저장해서 레지스트리내에 깊숙히 보관하고 있다. 그 레지스트리 내에는 PnP 버스 리소스들 외에도 다른 많은 설정정보가 담겨있다. 현재의 PnP 리소스 설정상태는 메모리상에 하나가 저장되고, 또한 (아마도 동일한 내용으로서) 하드디스크에도 또 하나가 저장된다. 윈도우즈 98 에서 이 내용을 보거나 그 설정을 강제로 바꾸려면 디바이스 매니저(장치관리자)를 이용한다.

윈도우즈 98 에서 디바이스 매니저(장치관리자) 창을 여는 방법은 2 가지가 있다 : 1. My computer(내 컴퓨터) --> Control Panel(조종판) --> System Properties(시스템 속성) --> Device Manager(장치관리자) 2. My Computer(내 컴퓨터) 에서 오른쪽 클릭 --> Properties(속성) --> Device Manager(장치관리자). 그런 다음, Device Manager(장치관리자) 창 내에서 디바이스를 하나 선택한다(같은 종류 디바이스들이 여러개 있을 때는 아래로 몇 단계를 거쳐야 할수도 있다). 그다음, "Properties(속성)" 을 클릭한 후 "Resources(자원)"을 클릭한다. 리소스 설정을 수동으로 바꾸려면 "Use automatic settings(자동설정)" 에 체크된 것을 해제한 후 "Change Settings(설정변경)" 을 클릭한다. 이제 설정을 변경해본다. 이때 변경하는 것이 허용되지 않을 수도 있다. 만일 변경이 허용된다면 이제 강제설정이 완료된 것이다. 강제로 설정되었다는 메시지가 뜰 것이다. 만일 윈도우즈가 가졌던 현재의 세팅내용을 그대로 유지하면서도 단지 그 것을 "자동설정"이 아닌 "강제설정"으로 만들고 싶은 것이라면, 일단 뭔가 다른 내용으로 강제설정한 후에 다시 원래의 셋팅으로 바꾸어 강제설정해야만 한다.

윈도우즈 98 에서 강제설정된 것들을 보려면 "강제설정된 하드웨어들" 리스트를 보면 된다 : --> Programs(프로그램) --> Accessories(악세서리) --> System Tools(시스템 도구) --&62; System Information(시스템 정보) --> Hardware Resources(하드웨어 자원) --> Forced Hardware(강제설정된 하드웨어). 윈도우의 버스리소스들을 강제로 변경시켰다면 윈도우는 그 변경을 ESCD 에 기록할 것이다(정상종료시켜야만 한다). "System Information(시스템 정보)" 창을 보면 IRQ 와 IO 포트들이 윈도우즈하에서 어떻게 할당되었는지를 조사할 수 있다.

윈도우즈에서는 버스 리소스의 충돌이 없는데 리눅스에서는 충돌이 일어나는 수가 있다. 이는 윈도우즈는 ESCD 의 내용과 다르게 버스 리소스를 할당할 수 있기 때문이다. 드문 경우라 하겠지만 윈도우하의 모든 디바이스들이 legacy 디바이스이거나 강제설정된 상태라면 윈도우즈와 ESCD 의 설정은 동일하게된다.

5.3.4. 새로운 디바이스를 윈도우즈나 리눅스에 추가하기

새로운 PnP 디바이스가 추가되고 BIOS 는 "PnP OS 없음" 상태라면, BIOS 는 자동적으로 디바이스를 설정한 후 그 설정내용을 ESCD 에 저장한다. 만일 추가되는 디바이스가 non-PnP legacy 디바이스라면 (혹은 디바이스의 점퍼설정등을 통해 legacy 디바이스로 만들었다면), 이를 처리하기위해 다음과 같은 몇가지 방법을 택할수 있다.

CMOS 설정메뉴를 이용해 직접 BIOS 에게 그 디바이스가 사용할 특정한 버스 리소스(가령 IRQ)를 예약해두도록하여, PnP 가 그 리소스를 할당시켜버리지 못하게 한다. 이 방법은 그 정보를 ESCD 에 기록시키지는 못한다. 하지만 CMOS 설정메뉴를 보면 충돌 발생시 이 정보를 ESCD 의 내용에 우선시킬 것인지 여부를 정하는 메뉴가 있을것이다. 또다른 방법은 도스/윈도우즈에서 ICU 를 실행시키는 것이다. 그밖의 방법으로는 윈도우즈 9x/2k 하에서 수동으로

설치한 후 그 설정내용을 "강제" 하는 방법이다(앞절에서 설명했다). 강제설정 후 PC 가 정상 종료될 때 윈도우즈가 ESCD 를 갱신시킬 것이다.

5.4. PnP 기능을 끄려면(disable)?

PnP 기능을 아예 끌수 없는 PnP 전용의 디바이스들이 많이 있다. 하지만 일부 디바이스의 경우, 점퍼를 쓰든지 아니면 디바이스 구입시 함께 따라온 윈도우즈 프로그램을 실행시켜(jumperless configuration) PnP 기능을 끌 수 있다. 디바이스 드라이버가 디바이스를 제대로 설정하지 못하는 경우에는, PnP 기능을 끔으로써 아예 PnP 설정이라는 복잡한 작업을 피해갈 수가 있다. 이경우 BIOS 에게 이 버스 리소스들이 예약되어있음을 알리는 것을 잊어서는 안된다. 하지만, 또한 다음과 같은 이유로 디바이스의 PnP 기능을 꺼버리기 싫을수도 있다 :

1. 같은 컴퓨터상에 MS 윈도우즈가 설치되어있는 경우, PnP 로 하여금 디바이스를 윈도우에서와 리눅스에서 각기 다르게 설정케할 필요가 있을 수 있다.
2. IRQ 번호나 포트 어드레스 등등의 선택범위가 PnP 를 사용하지 않는 경우 상당히 제한될 수 있다.
3. 당신이 사용하는 어떤 리눅스 디바이스 드라이버가 디바이스 탐색시에 PnP 방법을 이용하는 것일수 있다.
4. 장래에 설정을 변경할 필요가 생길 경우, 설정이 PnP 로 되어있다면 일이 훨씬 쉬워진다(점퍼를 설정할 필요도 없고 도스/윈도우즈용 프로그램을 실행시킬 필요도 없으므로).
5. 현재 또는 장래라도 다른 PnP 디바이스들을 설정해야할 필요가 생길 것이다. 그렇다면 어쨌든 PnP 디바이스에 대해 알아둘 필요가 있을 것이다.

일단 non-PnP 디바이스로 설정했다면, 그 디바이스는 (점퍼 설정을 바꾸거나 도스/윈도우즈용 설정 소프트웨어를 다시 실행시키기 전에는) PnP-BIOS 나 PnP 소프트웨어에 의해서는 설정작업이 이루어지지 않게된다.

5.5. Isapnp (isapnptools 라는 프로그램 패키지중 하나이다)

운나쁘게도, 당신이 PnP 의 기본적인 내용들을 모른다면 isapnp 에 대한 문서들의 대부분은 이해하기 힘들 것이다. 본 HOWTO 와 isapnp 패키지에 딸려오는 FAQ 를 보면 그 내용을 이해하는데 도움이 될 것이다. `isapnp` 는 오직 ISA 버스상의 PnP 디바이스들을 위한 프로그램으로서 PCI 버스용 디바이스에는 해당없다. 부팅시에 리눅스 프로그램인 "isapnp" 를 실행시키면 /etc/isapnp.conf 에 적혀있는 리소스 값대로 디바이스들을 설정할 것이다. 이 설정 파일을 자동으로 생성시키는 것도 가능하지만, 생성시킨 후에는 직접 수동으로 다양한 옵션들을 골라 편집해 주어야만 한다.

isapnp 를 사용할 경우, isapnp 가 하드웨어 내에 어드레스 등등을 셋팅하기도 전에, 커널에 내장된 디바이스 드라이버가 너무 일찍 실행되는 수가 있다. 이렇게 되면 그 디바이스 드라이버는 디바이스를 발견할 수 없다는 결론을 내버린다. 드라이버는 올바른 어드레스를 검색했으나 아직 하드웨어상에는 그 어드레스가 셋팅되지 않았기 때문이다.

만일 당신의 리눅스 배포본이 자동으로 isapnptools 를 설치했다면, 아마도 isapnp 는 이미 컴퓨터 시동시부터 작동하고 있을 것이다. 이 경우, 당신은 `man isapnp.conf` 의 내용을 참고해 /etc/isapnp.conf 를 편집해주기만 하면 된다. 명심할 점은, 설정화일을 편집하여 어떻게 설정할지를 지정해주는 것이므로, 이는 수동으로 PnP 를 설정하는 것과 같은 효과를 가진다는 것이다. "pnpdump" 프로그램을 쓰면 이 설정화일을 작성하는데 도움이 될 것이다. 이 프로그램은 설정화일 거의 전부를 만들어주지만 이를 사용하기전에는 반드시 그 내용 일부를 요령있게 편집해주어야만 한다. 이 화일에는 주석이 일부 붙어있으므로 편집시에 도움이 될것이다. 당신이 설정작업에 "isapnp" 를 이용하고 PnP BIOS 도 가지고 있다면, BIOS 설정 메뉴에서 PnP OS 없음으로 설정하고플 것이다. BIOS 가 PCI 디바이스 설정작업을 담당해주기를 원할테니 말이다. BIOS 가 ISA 디바이스들까지 설정해버린다 해도 isapnp 는 ISA 디바이스들의 설정작업을 새로 다시하게된다.

/etc/isapnp.conf 파일에 사용된 용어들은 처음에는 낯설어 보일지도 모르겠다. 예를 들어 0x3e8 이라는 IO 어드레스는 대신 "(IO 0 (BASE 0x3e8))" 이라고 쓰여있을 것이다. "IO 0" 는 이 디바이스가 사용하는 첫번째(0 번) IO 어드레스 영역임을 뜻한다. 전체 내용을 다음처럼 표현할 수도 있을 것이다 : "IO[0] = 0x3e8". 하지만 isapnp 는 그런 표현은 사용하지 않는다. "IO 1" 은 그것이 디바이스가 사용하는 두번째 IO 어드레스 영역임을 의미한다. "INT 0" 도 유사한 의미이지만 이것은 IRQ(인터럽트) 에 해당된다. 카드 한개에 여러개의 물리적 디바이스들이 탑재되어있는 경우도 있는데, 그 경우 위의 설명은 그중 일개 디바이스에만 해당하는 것이다.

5.6. PCI 유틸리티

PCI 유틸리티(= pciutils, "pcitools" 라는 잘못된 명칭으로 불리기도 한다)라는 패키지는 당신이 수동으로 PCI 버스를 PnP 설정할수 있게 해준다. "lspci" 나 "scanpci"(X 윈도우용) 프로그램은 버스 리소스들을 출력해주며, "setpci" 는 하드웨어 디바이스 내의 리소스 할당을 담당한다. setpci 는 주로 스크립트 프로그램들이 사용할 용도로 제작된 듯하며, 현재로서는 이걸 사용하려면 PCI 설정 레지스터의 상세한 내용에 대해 잘 알고 있어야 한다. 이 주제는 본 문서에서는 설명하지 않을 것이며 setpci 에 대한 맨페이지에도 잘 나와있지 않다.

5.7. 윈도우를 이용한 설정

같은 PC 상에 윈도우즈 9x (또는 윈도우 2000)도 설치되어있다면, 윈도우즈를 시작시켜서 윈도우즈가 PnP 를 설정토록 하라. 그다음 윈도우즈(혹은 도스) 상태에서 리눅스를 시작시키면 된다. 하지만 이경우 PCI 디바이스들 용의 IRQ 에서 문제가 생길 수 있다. 리눅스에게 자리를 양보하기 위해 윈도즈우가 종료될 때에, PCI 디바이스의 설정 레지스터에 저장되어있는 IRQ 번호를 지워버리는(zero, 0 으로 만들어버리는) 수가 있는 것이다. 이 경우 리눅스는 IRQ 값이 0 인 것을 발견했다고 불평을 늘어놓을 것이다.

위의 현상은 리눅스를 바로가기(shortcut) 파일(PIF)로 시작시킬 경우에 발생하는 것으로 보고되고 있다. 하지만 바로가기 PIF 파일을 사용하면서도 이런 문제를 피할수 있는 방법이 알려져 있다. 바로가기 파일이란 리눅스로 치면 심볼릭 링크와 유사하나 그보다 좀 발전된 것이다. PIF 파일은 "설정" 할 수 있기 때문이다. 도스로부터 리눅스를 시작시키려면 리눅스를 시작시키는 배치파일(스크립트)을 만들어야 한다(리눅스를 시작시키는 프로그램은 "loadlin" 이

라는 패키지에 들어있다). 그다음 그 배치파일을 가리키는 PIF 바로가기 파일을 만든 후, 그 바로가기 파일의 "Properties(속성)" 창을 연다. "Advanced(고급)" 을 선택한 다음 "MS-DOS 모드" 를 체크해놓으면 이제 리눅스는 진짜 MS-DOS 모드에서 시작될 것이다.

이제 PCI IRQ 들이 0 이 되버리는 것을 막기위해 술수를 부려보자. "Specify a new MS-DOS configuration" 을 체크하라. 그다음 당신에게 제시될 디폴트 설정을 그대로 받아들이든지 아니면 "Configuration" 을 클릭해서 그 내용을 변경하라. 이제 바로가기 파일을 클릭해서 리눅스를 시작시키면 당신의 새로운 설정에 따라 새로운 설정 파일들(Config.sys 와 Autoexec.bat)이 생성될 것이다.

그전에 존재하던 같은 이름의 파일들은 "Config.wos" 와 "Autoexec.wos" 라는 이름으로 저장될 것이다. 리눅스 사용을 끝내고 PC 를 끌 경우, 장차 PC 를 사용하기 위해서는 또 도스를 실행시켜야 하며, 그러기 위해서는 원래의 파일들이 또 필요하다. 따라서 당신은 이름이 바뀌어버린 이 원래의 화일들을 *.sys 와 *.bat 의 확장자로 반드시 다시 회복시켜두어야 한다. 윈도우즈/도스 상태로부터 리눅스가 시작되는 시점에서는, 윈도우즈는 당신이 리눅스를 다 사용한 후 다시 윈도우즈로 돌아오면 이 화일들을 원래의 이름으로 자동복구시켜 놓겠다고 벼르고 있다. 하지만 리눅스를 다 사용하면 PC 를 꺼버리지 다시 윈도우로 돌아가는 일은 절대 없으므로, 이 화일들은 원래의 이름으로 돌아가지 못하게 된다. 그럼 어떻게 해야 이 화일들을 다시 원래이름으로 돌려놓을 수 있을까? 그건 간단하다. 당신의 "리눅스 시작" 배치파일내에 이 화일들을 원래이름인 *.bat 와 *.sys 로 복구시키는 명령어를 넣어두면 되는 것이다. 이름을 변경시키는 명령어를 당신의 배치파일내의 리눅스를 로드하는 부분 직전에 넣어두어라.

또한, 당신의 바로가기 화일의 "Property(속성)" 창의 "General(일반)" 탭을 클릭해서 "Read-only(읽기 전용)" 항목을 체크해두어야 하는 것으로 알려져 있다. 그렇게 하지 않으면 윈도우즈가 "Advanced Settings(고급 설정)" 부분을 "Use current MS-DOS configuration" 으로 리셋시켜버려서 PCI IRQ 들을 0 으로 만들어버리는 수가 있다. 결론적으로, 윈도우즈는 당신이 현재의 MS-DOS 설정상태를 이용할 때는 그 IRQ 들을 지워버리지만, 새로운 설정상태(내용적으로는 예전의 설정상태와 동일한 값들이어도 상관없다)를 사용할 때는 지우지 않는 것이다. 이점만 놓고보자면 윈도우가 아주 일관성있다고 말하기는 힘들겠다.

5.8. PnP 소프트웨어와 관련문서들

- Isapnptools homepage (http://www.roestock.demon.co.uk/isapnptools/)
- Patch to make the Linux kernel PnP (http://www.astarte.free-online.co.uk)
- PnP driver project (http://www.io.com/~cdb/mirrors/lpsg/pnp-linux.html)
- PnP Specs. from Microsoft (http://www.microsoft.com/hwdev/respec/pnpspecs.htm)
- Book: PCI System Architecture, 3rd ed. by Tom Shanley +, MindShare 1995. PCI 버스가 가진 PnP 와 유사한 기능들을 다루고있다.
- Book: Plug and Play System Architecture, by Tom Shanley, Mind Share 1995. ISA 버스상의 PnP 에 관한 세부사항을 다루고있다. PCI 버스상의 PnP 는 간단한 개요만 적고있다.

- Book: Programming Plug and Play, by James Kelsey, Sams 1995. PnP BIOS 와의 통신 프로그래밍을 자세히 다룬다. ISA, PnP, PCMCIA 버스들도 다루고 있다.

6. 드라이버에게 설정상태를 알려주기

6.1. 개요

이 작업이 어떻게 이루어지는가는 순전히 그 드라이버에 달려있다. 일부 드라이버들은 자신의 물리적 디바이스의 설정상태를 파악하는데 있어 둘 이상의 방법을 가질수도 있다. 아주 극단적인 방법으로는 아예 버스 리소스 값들을 커널(또는 모듈) 소스코드에 직접 적어넣은 후 다시 컴파일하는 방법이 있겠고, 그에 반대되는 극단적인 방법으로는 드라이버가 자동적으로 모든 것을 처리해서 당신이 해줄 일이 전혀 없는 경우도 있을수 있다. 심지어는 PnP 방법들을 사용해서 버스 리소스 값들을 하드웨어에 셋팅시키는 방법도 가능할 수 있다.

중간적인 방법이라면 어떤 프로그램을 수행시켜 리소스 정보를 드라이버에게 전달하거나 리소스 정보를 파일에 넣어두는 것 정도겠다. 어떤 경우에는 드라이버가 디바이스가 있음직한 어드레스들을 뒤져 해당 디바이스를 찾아내는 수도 있다. 이때는 여러 IRQ 들을 테스트해서 어느 것이 동작하는지 확인하게된다. 드라이버가 이 작업을 자동으로 해낼수도 있지만 그렇지 못할 수도 있다. 그밖에, 드라이버가 PnP 방법들을 사용해 디바이스를 찾아내어 그 버스 리소스 설정상태를 알아낼 수도 있지만, 이때 그 디바이스를 셋팅해주지는 않을 것이다. 또한, 드라이버는 /proc 디렉토리의 일부 화일들을 살펴볼수도 있다.

때로는 사용자가 직접 "수동으로" 드라이버에게 너는 어떠어떠한 버스 리소스를 사용해야만 한다고 알려줘야 할 때도 있다. 그러한 버스 리소스 값들은 커널 또는 로딩될 모듈에게 파라메터로 전달하게 된다. 드라이버가 커널에 내장되어 있는 경우에는 "부트 프롬프트" 를 통해 커널에게 전해주어야 한다. Boot-Prompt-HOWTO 문서에는 몇몇 버스 리소스들과 그밖의 파라메터들이 설명되어있다. 일단 어떤 파라메터들을 커널에 전달해야 할지 알고있다면, 그것을 lilo.conf 파일속에 append="..." 형식으로 기록해두는 방법도 가능하다. 그러면 lilo 프로그램이 실행될 때 이 정보를 읽어 커널 로더에게 전달해준다.

만일 드라이버가 모듈로서 로드된다면 버스 리소스 값들을 그 모듈에게 파라메터로 전달해줄 필요가 있을것이다. 일부 버전의 리눅스에서는 /usr/lib/modules_help/descr.gz 파일에 사용가능한 모듈 파라메터들의 리스트가 담겨있을 것이다. 모듈에 대한 파라메터들은 (자동적으로 로드되는 것들도 포함해서) /etc/modules.conf 에 적혀있을 수 있다. 보통은 이 화일을 수정시켜주는 도구들이 있기는 하나, 배포판에 따라 제공되는 그 도구들이 각기 다르다. 이 파일내에 적혀있는 주석은 파일을 수정하는데 도움이 될 것이다. 또한, 여기 적힌 파라메터들의 내용에 따라 /lib/modules 디렉토리에 밑에 넣어둔 모듈들이 로드될 것이다.

드라이버들이 버스 리소스들을 찾아내는 방법들은 엄청나게 일관성이 없지만 그 최종 목적만큼은 다들 같다. 드라이버에 대해 뭔가 문제가 있다면 그 드라이버 문서를 봐야한다(커널의 문서 디렉토리를 살펴보라). 몇몇 드라이버들에 대한 간략한 정보가 아래 절들에 나와있다.

6.2. 시리얼 포트 드라이버: setserial

표준 시리얼 포트 드라이버의 경우(멀티포트 카드는 해당되지 않는다), setserial 명령을 써서 그 드라이버를 설정하게된다. 이 명령어는 대개 시스템 시동 파일의 실행시 수행된다. 최신 버전부터는 단순히 통상의 방식으로 setserial 명령을 실행시킬 경우, /etc/serial.conf 라는 파일이 "편집" 되게 되었다. 따라서 `setserial` 명령으로 세팅시킨 내용은 `serial.conf` 설정파일속에 저장되는 것이다. 시스템 시동파일에 의해 수행될 때의 `setserial` 명령은 `serial.conf` 파일을 참조하게 된다. 이와같은 설정방법은 당신의 배포본에 따라서는 해당없을 수도 있다.

당신이 옵션을 어떻게 주느냐에 따라 `setserial` 을 두가지 다른 식으로 사용할 수 있다. 하나는 수동으로 드라이버에게 설정내용을 알려주는 것이다. 다른 하나는 어떤 주어진 어드레스를 검색케해서 거기에 시리얼 포트가 있는지 여부를 보고하게하는 것이다. 이때, 그 어드레스를 조사해서 해당 포트에 사용되는 IRQ 가 뭔지까지도 알아보도록 할수있다. 시스템 시동시에 드라이버는 `setserial` 비슷한 명령을 실행시킨다. 이로서 IRQ 를 알아내지 못한 경우에는 그냥 "표준" IRQ 값을 할당하게 되며, 따라서 이는 잘못된 값일수도 있다. 포트의 존재여부도 조사해줄 것이다. 더 자세한 것은 Serial-HOWTO 문서를 보라.

PCI 시리얼의 경우에는, 그 시리얼 드라이버가 특정한 모뎀들을 감지해내고 버스 리소스를 설정할 수 있다.

6.3. 사운드 카드 드라이버들

6.3.1. OSS-Lite

IO, IRQ, DMA 값들을 파라메터로 모듈에게 알려주거나, 그 값들을 커널에 넣어 컴파일해야만 한다. 하지만 일부 PCI 카드들은 자동적으로 감지될수도 있다(가령 lspci 명령 따위를 사용할 경우). 레드햇은 "sndconfig" 라는 프로그램을 제공하는데, 이것은 ISA PnP 카드를 감지한 후, 그 감지된 버스 리소스에 맞춰 자동으로 모듈이 로딩되게한다.

6.3.2. OSS(Open Sound System) 과 ALSA

이 프로그램들은 PnP 방법을 써서 카드를 감지한 후 적절한 드라이버를 택해 로드한다. 또한 ISA PnP 카드상에 버스 리소스를 셋팅해줄 수도 있다. 충돌을 피하기 위해서는 당신이 수동으로 간섭해주어야할 수도 있다. ALSA 드라이버의 경우 ISA-PnP 지원은 옵션사항이므로, 원한다면 isapnp 를 사용할 수도 있을 것이다.

7. 나의 현재의 설정상태는 무엇인가?

여기서의 "설정" 은 PnP 버스 리소스들(어드레스, IRQ, DMA)을 할당하는 것을 의미한다. 각 디바이스마다 이 질문에 대한 답은 두 부분으로 구성된다. 이 두 부분은 반드시 서로 일치해

야만 한다.

1. 그 디바이스 드라이버 소프트웨어의 설정상태는 무엇인가? 즉, 드라이버는 하드웨어의 설정상태를 무엇이라고 생각하고 있는가?
2. 디바이스 하드웨어상의 설정상태(만일 설정되어있다면)는 무엇인가?

당연한 말이지만 디바이스 하드웨어의 설정내용과 그 드라이버의 내용은 일치해야만 한다(또 보통은 일치한다). 만일 뭔가 제대로 동작하지 않는다면, 그 원인은 두 내용이 일치하지 않기 때문일 수도 있다. 이는 드라이버가 디바이스의 실제의 하드웨어 설정내용과는 다른 잘못된 정보를 가지고 있음을 뜻한다. 이런 상황은 문제를 야기한다. 당신이 사용하는 소프트웨어가 어디가 잘못된 것인지에 대한 적절한 답을 제시해주지 못한다면(혹은 자동으로 설정을 바로잡아주지 못한다면), 당신이 직접 하드웨어 디바이스의 설정상태와 드라이버의 설정상태를 조사하여야 한다. 리눅스 디바이스 드라이버들은 몇몇 경우 자신의 모든 내용을 알려주지만, 그에 반해 손수 하드웨어에 세팅된 내용들을 알아내는 것은 쉬운 일이 아니다.

또다른 문제는, 화면에 설정상태에 대한 메시지들이 나타날 경우 그것이 디바이스 드라이버의 설정상태를 의미하는 것인지, 아니면 디바이스 하드웨어의 상태를 뜻하는 것이지, 아니면 양자 모두인지가 분명치 않을수 있다는 점이다. 디바이스 드라이버가 하드웨어의 설정을 담당했거나 아니면 하드웨어의 설정을 체크해본 후라면 드라이버는 올바른 정보를 가지고 있을 것이다.

하지만 때로는 드라이버가 스크립트로부터 잘못된 리소스 정보를 제공받거나, 잘못된 리소스 파라메터들이 모듈에 전달되거나, 아예 리소스 정보를 전달받지 못해서 잘못된 디폴트 리소스 상태를 사용해버리는 경우가 있다. 예를 들어, "setserial" 사용시 드라이버에게 잘못된 리소스 설정 정보를 전해준다면 드라이버는 아무 의심없이 그 정보를 받아들일 것이다.

7.1. 부트시의 메시지

설정내용에 관한 어떤 정보들은 컴퓨터 시작시 BIOS 나 리눅스가 화면에 출력시키는 메시지를 통해 얻을 수도 있다. 이 메시지들은 번개처럼 지나가버려 읽기가 힘들지만 일단 스크롤이 멈추면 Shift-PageUP 키를 눌러서 뒤로 돌려볼수 있다. 앞쪽으로 스크롤 시키려면 Shift-PageDown 키를 누른다. 쉘 상태에서 언제든 "dmesg" 라고 입력하면 리눅스 커널이 출력시켰던 메시지를 볼수 있지만, 이때는 아주 중요한 몇가지 내용들과 BIOS 가 출력한 내용들은 볼수 없다. 리눅스가 출력시킨 메시지들은, 때로는 단지 디바이스 드라이버가 생각하고 있는, 즉 잘못된 설정파일을 참조해 알아낸 설정내용일 수도 있다.

BIOS 메시지는 부팅 당시의 실제의 하드웨어 설정내용을 보여주지만, 나중에 isapnp 나 pci 유틸리티들이나 디바이스드라이버가 이 설정을 바꿔버릴 수가 있다. BIOS 메시지는 리눅스가 출력하는 정보보다도 앞서 출력된다. Shift-PageUp 을 사용해서 메시지를 읽는 대신, "Pause" 키를 눌러서 동작의 진행을 멈추게 해버리는 방법도 있다. 아무 키나 누르면 진행이 재개된다. 하지만 일단 리눅스가 메시지를 출력하기 시작하면 더이상 "Pause" 키는 먹지 않는다. 리눅스의 메시지 출력은 중간에 멈출수가 없기 때문이다.

7.2. 나의 디바이스 드라이버들의 설정상태는?

명령행 상태에서 실행시켜 디바이스 드라이버를 설정시키는 프로그램들이 있다(가령 시리얼 포트에 대한 "setserial" 명령). /proc 디렉토리의 내용이 도움이 될 것이다. 이 디렉토리 밑의 많은 화일들은 버스 리소스 정보를 담고 있다. 그중 몇가지만 여기서 언급할 것이다. /proc/ioports 는 그 드라이버가 사용하는(혹은 사용하는 것으로 잘못 알고 있는) I/O 어드레스를 보여줄 것이다. 하드웨어가 반드시 이 정보대로 세팅되어있지 않을수도 있다.

/proc/interrupts 는 단지 현재 사용중인 인터럽트들만을 보여준다. 따라서 드라이버들에 할당되기는 했으나 현재 사용중은 아닌 많은 인터럽트들은 나타나있지 않을 것이다. 예를 들어, 당신의 플로피 드라이브에 플로피 디스크가 들어있고 또한 사용가능한 상태라 해도, 실제 사용중인 상태가 아니라면 그에 해당하는 인터럽트는 나타나있지 않을것이다. 또한, 앞서도 설명했듯이 여기에 인터럽트 번호가 나와있다고 해서, 하드웨어에도 그 정보 그대로 셋팅됐음을 의미하는 것은 아니다. 하드웨어가 이 정보대로 셋팅되어있지 않았음을 알수있는 한가지 실마리는, 이 인터럽트에 의해 생성된 인터럽트 번호가 0 인 경우이다. 더구나 어떤 인터럽트들이 발해졌다해서 그것이 거기에 적힌 특정 디바이스에 의해 발생한 것이라고 보증할 수도 없다. 현재 사용중이 아닌 다른 디바이스가 예전에 발생시켰던 인터럽트일수도 있기 때문이다. (커널이) 현재 사용중이지 않은 디바이스라도 여러가지 원인으로 인해 계속 인터럽트를 발생시키는 수가 있다.

7.3. 나의 하드웨어 디바이스들의 설정상태는?

PCI 버스상의 디바이스들에 어떤 버스 리소스들이 할당되었는지는 "lspci" 나 "scanpci" 명령을 사용하면 쉽게 알아낼 수 있다. 커널 2.2 의 경우라면 `/proc/pci` 을 보고, 그 이후의 커널이라면 `/proc/bus/pci` 를 보라. `/proc/pci` 에 나타나는 IRQ 들은 16 진수로 표현된다. `/proc/bus/pci/devices` 을 해독하느라 고생할 필요는 없다. "lspci" 가 알아서 해독해주기 때문이다.

PCI 에 대해서는 대부분의 경우 그 하드웨어가 어떻게 설정되었는지만 볼수 있을 뿐 어떤 리소스를 필요로하고 있는지는 볼수 없을 것이다. 어떤 경우에는 단지 베이스 어드레스(어드레스 영역의 최초번지)만을 볼수 있을 뿐 그 끝번지는 볼수 없을수도 있다. 영역 전체를 볼수있는 경우라면 얼마나 많은 바이트의 어드레스 리소스가 필요한지를 알 수 있을 것이다. 따라서 어떤 경우에는 요구되는 리소스양을 계산해서 필요하다면 다른 어드레스 영역에 그만큼을 셋팅(setpci -- 사용법이 어렵다)시켜줘야 할수도 있다. 디바이스는 스스로 어드레스 영역의 크기를 알고 있으므로 당신은 그저 그 디바이스에게 새로운 베이스 어드레스가 뭔지만 가르쳐주면 된다.

ISA 버스의 경우에는 `pnpdump --dumpregs` 명령을 시도해 볼수는 있지만 이것은 확실한 수단은 아니다. 그 출력도 암호문처럼 난해하긴 하지만 해독해볼수는 있다. `pnpdump` 가 "시도한" (그래서 거기있는 어떤 디바이스로부터 알아낸) read-port 어드레스와, 그 발견된 디바이스가 가진 I/O 어드레스를 혼동해서는 안된다. 이 둘은 같지않다. ISA 버스상의 실종된 하드웨어(PnP 및 legacy)를 찾으려면 "scanport" 라는 프로그램을 사용해보라. 하지만 이 프로그램은 PC 를 먹통으로 만들수도 있음에 유의하라. 이 프로그램은 IRQ 를 알려주지도 않을 것이고 발견된 하드웨어가 뭔지 친절히 확인해주지도 않을 것이다.

부팅시에 BIOS 가 출력하는 메시지는 그 순간의 하드웨어 설정상태를 알려줄것이다. 만일 BIOS 만이 설정을 담당한다면 이 설정상태는 계속 유지될 것이다. 리눅스가 출력하는 메시지 내용중 일부는 드라이버들이 커널의 PnP 함수를 사용해 버스 리소스값들을 조사및 설정했음을 알리는 것일수도 있다. 이 값들은 올바른 값이어야 한다. 다만, 디바이스가 보내온 메시지의 내용은 어떤 설정화일내의 값(즉, 드라이버가 생각하고 있는 값)을 그대로 출력한 것일 뿐임에 주의하라. 물론 디바이스가 제대로 동작한다면 디바이스의 설정상태와 드라이버의 설정상태는 동일하게 될 것이다.

어떤 사람들은 윈도우즈를 써서 버스 리소스가 어떻게 설정되어있는지 알아보기도 한다. 불운하게도, 전원을 끄면 하드웨어는 자신의 버스 리소스 설정을 잊어버리기 때문에 그 설정내용은 리눅스때의 것과는 다를수 있다. 때로는 똑같은 값일 수도 있는데, 이는 BIOS 가 설정한 값을 윈도우즈도 리눅스도 그대로 인정하고 받아들이는 수가 많기 때문이다. 하지만 윈도우즈나 리눅스가 설정을 담당하는 경우에는 두 운영체제는 서로 다른 값으로 설정시킬 수도 있다. 따라서 디바이스들이 양 운영체제에서 똑같은 값으로 설정되리라 기대해서는 안된다.

8. 에러 메시지

8.1. 예기치않은 인터럽트

이는 특정 인터럽트가 발생했으나 어떤 드라이버도 이 인터럽트를 기대하고 있지 않은 경우를 말한다. 하드웨어가 실수로 인터럽트를 발생시키는 경우는 드물다. 그보다는 그 소프트웨어에 사소한 버그가 있어서 어떤 소프트웨어가 인터럽트를 발생시키는 무슨 작업을 했음을 알아차리지 못하기 때문이기가 쉽다. 많은 경우 안심하고 이 에러메시지를 무시해도 상관없으며, 특히 부팅시에 한두번 이런 메시지가 뜰 뿐일 경우가 그러하다. 부팅 메시지에서 해당 부분 근처의 메시지를 찾아 무슨일이 일어나고 있는지를 살펴보라. 예를 들어, 하드웨어 탐색이 진행중인 때라면 아마도 물리적인 디바이스를 찾는 작업 와중에 아직 드라이버가 대응할 수 없는 인터럽트를 디바이스가 발생시켜버렸을 것이다. 아마도 그 드라이버는 아직 올바른 IRQ 번호에 귀기울이지 못하는 상태일 것이다.

9. 부록

9.1. 유니버설 플러그 엔드 플레이 (UPnP, Universal Plug and Play)

이것은 사실상 마이크로소프트사에 의해 개발된 네트웍 PnP 의 일종이지만 리눅스에서도 사용가능하다. UPnP 하드웨어를 네트웍에 꽂아 네트웍상의 다른 UPnP 디바이스들과 자동으로 통신이 이루어지기만 하면, 따로 별도의 설정작업이 필요없는 것이다. 여기서의 "설정" 은 넓은 의미이며 버스리소스의 할당만을 의미하는 것이 아니다. 또다른 목적은 네트웍이나 설정작업에 문외한인 사람들도 라우터, 게이트웨이, 네트웍 프린터등을 설치할 수 있게 하는데

있다. UPnP 는 무선 네트워킹에서 주로 사용될 것이다.

UPnP 에서 사용되는 것들

- 디바이스들을 탐색하기 위한 Simple Service Discovery Protocol
- General Event Notification Architecture
- 디바이스들을 제어하기위한 Simple Object Access Protocol

본 문서는 UPnP 를 다루지 않는다. 리눅스에서의 UPnP 는 인텔이 지원하고있는데, 이 회사는 이를 위한 소프트웨어를 개발해왔다. UPnP 와 같은 기능을 하는 다른 프로그램들도 있다. 이들에 대한 몇가지 비교가 다음에 나와있다. <65533> (http://www.cs.umbc.edu/~dchakr1/papers/mcommerce.html)

9.2. 어드레스에 대한 세부사항

어드레스에는 세 종류가 있다: 메인 메모리 어드레스, I/O 어드레스, 설정 어드레스가 그것이다. PCI 버스상에서는 설정 어드레스가 I/O 어드레스의 경우처럼 별도의 어드레스 공간을 형성한다. ISA 설정 어드레스같은 복잡한 경우말고는, 버스상의 어떤 어드레스가 메모리 어드레스인지, I/O 어드레스인지, 설정 어드레스인지 여부는 버스상의 특정 라인의 전압에 의해 결정된다. ISA 설정 어드레스의 경우라면 자세한 사항은 9.3절 를 참고하라.

9.2.1. 어드레스 영역들

본 문서에서 "어드레스" 라는 용어는 때로는 연속된 어드레스 범위를 뜻하기도 한다. 어드레스들은 단위 바이트들로 구성되므로, 개별 어드레스 하나는 단지 어떤 바이트 하나의 위치를 뜻하지만, I/O (혹은 메인 메모리) 어드레스들이라고 말할때는 이는 복수개의 바이트들로 구성된 것이다. 따라서, 가령 8 바이트로 구성된 어드레스 범위 하나가 I/O 어드레스들로 사용되기도 하며, 디바이스에 할당되는 메인 메모리 어드레스들의 범위로 이보다 훨씬 큰 영역이 할당되기도 한다. 한개의 시리얼 포트(I/O 디바이스에 해당한다)에는 그 디바이스의 I/O 어드레스 시작번지만 배당하면 충분하다(가령 3F8). 왜냐하면 시리얼 포트에 대한 어드레스 범위는 단지 8 바이트뿐임이 잘 알려져있기 때문이다. 시작 어드레스는 "베이스 어드레스(base address)" 라고도 부른다. 때로는 "어드레스 범위(address range)" 를 그냥 "범위(range)"이라고 부르기도 한다.

9.2.2. 어드레스 공간

ISA 의 경우 동일한 어드레스 버스를 이용해 I/O 및 (메인) 메모리 어드레스 "공간(space)" 각각을 액세스한다(어드레스용으로 사용되는 회로배선들이 물리적으로 공유된다). 디바이스는 어드레스 버스상에 나타난 어드레스가 메모리 어드레스인지 I/O 어드레스인지를 어떻게 판별하는 것일까? 그 판별을 위한 정보및 기타 추가정보의 전송을 위해, 버스상에는 별도의 라인이 4 선 배선되어있다. 이 4 선중 특정한 한 선에 신호가 들어온면, 이는 CPU 가 I/O 어드레스로부터 읽기를 원한다는 것이며 따라서 메인 메모리는 그 버스상의 어드레스를 무시하

게 된다. 다른 3 선들도 이런 식의 용도로 사용된다. 따라서, 메인 메모리와 I/O 어드레스 각자는 읽기용도의 라인과 쓰기 용도의 라인을 가진 것이다(그래서 배선이 총 4 개가 된다)

PCI 버스의 경우에도 위와 똑같은 아이디어를 사용하지만(역시 배선이 4 개다), 약간 다르게 되어있다. 4 선중 한 선만 이용해 상태를 나타내는게 아니라, 4 개의 배선을 다 이용해 2진법 숫자를 표현한다(16 가지의 다른 표현이 가능하다). 따라서 더 많은 정보를 전할수 있다. 이 16 가지 숫자중 4 개는 윗 문단에서 말한 I/O 와 메모리 공간용으로 사용된다. 더불어 그 다음 2 개의 숫자는 설정 어드레스 공간용으로 사용한다. 나머지 10 개의 숫자는 다른 용도를 위해 남겨져 있다.

9.2.3. 범위(range) 체크 (ISA 에서의 IO 어드레스 충돌여부를 테스트)

ISA 버스상에서는, 각 PnP 카드내에는 다른 카드들이 자기와 같은 I/O 어드레스를 사용하는지를 체크해 볼 수 있는 방법이 내장되어있다. 만일 둘 이상의 카드들이 같은 IO 어드레스를 사용한다면 대개는 어느 것도 제대로 동작하지 않을 것이다(전혀 동작하지 않을 수도 있다). 좋은 PnP 소프트웨어는 이런 충돌을 피해서 버스 리소스를 할당하겠지만, 심지어는 그런 경우에도 이번에는 어딘가에 숨어있던 legacy 카드가 같은 어드레스를 사용해 버릴지 모른다.

테스트는 카드 자체의 IO 레지스터들 속에 테스트용 번호를 넣어준 후, PnP 소프트웨어가 이를 읽게 한 다음, 제대로 그 테스트 번호를 읽어냈는지 확인함으로써 이루어진다. 제대로 읽지 못한다면 무언가 잘못된 것이다(예를 들면 다른 카드가 같은 어드레스를 사용했기 때문일 수도 있다). 다른 테스트용 번호를 택해서 같은 작업을 반복해본다. 이는 결국 카드에 할당된 IO 어드레스의 범위를 체크하는것과 마찬가지이므로, 이 방법을 "범위 체크(range check)"라 부른다. 어드레스 충돌 테스트라고 부르는게 더 타당했을것이다. 만일 어드레스 충돌이 있다면 에러메시지를 받게 될테고, 그러면 문제를 해결하면 된다.

9.2.4. 메모리를 통한 직접 통신

전통적으로 대부분의 I/O 디바이스들은 I/O 메모리만 사용해서 CPU 와 통신했었다. 예를 들어, 시리얼 포트가 그러하다. 이때 디바이스 드라이버는 CPU 상에서 실행되면서 I/O 어드레스 공간및 메인 메모리에 데이터를 읽고 쓴다. 더 빠르게 동작하고자한다면 디바이스가 직접 메인 메모리로 데이터를 보내면 될 것이다. 그런 방법의 하나로 2.7절 나 버스 마스터링을 이용하는 방법이 있다. 또다른 방법으로는 메인 메모리상의 일부 공간을 디바이스용으로 할당해버리는 것이다. 그러면 디바이스는 DMA 나 버스 마스터링 때문에 골치썩일 필요없이 직접 메인 메모리에 읽고 쓸수 있다. 또한, 디바이스는 이와함께 IO 어드레스들을 사용할 수도 있다.

9.3. ISA 버스의 설정 어드레스들 (읽기포트 등등)

이 어드레스들을 "자동 설정 포트(Auto-configuration ports)"라고도 한다. ISA 버스의 경우, 기술적으로 볼때 설정 어드레스 공간이 없으나, 특별한 방법을 이용하면 CPU 가 PnP 카드상의 PnP 설정 레지스터들을 액세스할 수 있다. 이 용도로 3 개의 I/O 어드레스들이 할당되어 있으

며 각각은 겨우 한 바이트 짜리다(그러므로 어드레스 "범위" 라고 할수는 없다). 카드 하나당 3 개씩의 어드레스가 아니라, 어드레스 3 개를 모든 ISA-PnP 카드들이 공유하는 것이다.

이 세개의 어드레스들을 읽기포트(read-port), 쓰기포트(write-port), 어드레스포트(address-port)라고 부른다. 각 포트는 딱 1 바이트 크기다. 각각의 PnP 카드들상에는 설정 레지스터들이 많이 있으므로, 이 3 개의 어드레스들만으로는 단지 카드 하나의 설정 레지스터들용으로도 충분치 않다. 이 문제를 해결하기 위해서, "isolation" 이라 부르는 기법을 사용해 각 카드에 카드번호(handle)란 것을 부여한다. 자세한 사항은 9.6절 을 보라.

이제 특정 카드를 설정하기 위해서는, 쓰기포트 어드레스를 통해 해당 카드번호(handle)를 보내어 그 카드가 어드레스포트에 귀기울이게 만든다. 다른 모든 카드들은 자기 카드번호가 아님을 알고는 어드레스포트의 내용에 귀기울이지 않게 된다. 그 다음, 원하는 설정 어드레스를 어드레스포트로 보내면 해당 카드는 이를 알게 된다(다른 모든 카드들도 이를 전달받기는 하나 무시해버리며, 단지 해당 카드만이 이를 받아들이는 것이다). 이제, 읽기포트에서 읽거나 쓰기포트에 씀으로써 그 카드상의 설정 레지스터와 데이터 통신을 할수 있는 것이다.

쓰기포트는 언제나 A79 이고 어드레스포트는 언제나 279 이다(16진수). 읽기포트는 고정되어있지 않으나, 다른 ISA 카드와 충돌하지 않는 한, 설정 소프트웨어를 써서 203-3FF 범위내의 어떤 어드레스로 지정할 수 있다. 만일 충돌이 있는 경우에는 설정 소프트웨어는 다른 어드레스로 바꿀 것이다. 모든 PnP 카드들은 이 어드레스로 "프로그램" 된다. 따라서, 가령 isapnp 를 사용해서 설정 데이터를 지정하거나 체크하려면, 이 읽기포트 어드레스를 반드시 정해주야만 한다.

9.4. 인터럽트에 대한 자세한 사항

인터럽트를 이용해서도 다량의 정보를 전송할 수 있으나 이는 오직 간접적인 방법일 뿐이다. 인터럽트 요청 신호(특정 라인에 전압을 가하는 것이다)는 그저 인터럽트 컨트롤러라는 어떤 칩에게 현재 특정 디바이스가 뭔가를 요청하고 있음을 알릴 뿐이다. 인터럽트 요청을 받은 인터럽트 컨트롤러는 CPU 에게 신호를 보낸다. 그러면 CPU 는 현재 하고있는 작업을 중단하고 해당 디바이스용의 드라이버를 찾아 그안의 "인터럽트 서비스 루틴"(인터럽트 핸들러라고도 한다) 부분을 실행시킨다. 이 "루틴"은 무슨 일이 일어났는지 파악한 후 문제를 처리한다. 예를 들면, 디바이스로 바이트 열을 보내거나 반대로 받아와야할 경우일수도 있다. 이 프로그램(루틴)은 무슨 일이 일어났는지를 쉽게 발견해낼 수 있다. 왜냐하면 그 디바이스의 레지스터들이 어느 어드레스에 있는지를 드라이버 소프트웨어가 이미 알고있기 때문이다(물론 디바이스의 IRQ 번호와 I/O 어드레스가 제대로 세팅되어있어야 한다). 이 레지스터들은 그 디바이스에 대한 상태 정보를 담고있다. 소프트웨어는 이 레지스터들의 내용을 읽어 살펴본 후, 무슨 일이 일어났는지를 파악해서 적절한 조치를 취하게된다.

따라서 디바이스 드라이버들은 각자 어떤 인터럽트 번호(IRQ)에 귀기울여야할 지를 알고 있어야만 한다. PCI 버스에서는(그리고 ISA 버스에서도 일부 특별한 경우에는) 둘 이상의 디바이스들이 같은 IRQ 번호를 공유하는 것이 가능하다. 이러한 공유 인터럽트가 발생하면 CPU 는 그 인터럽트를 사용하는 모든 디바이스들의 인터럽트 서비스 루틴들을 전부 동작시킨다. 최초 실행된 서비스 루틴이 제일 먼저 하는 일은 그 디바이스 레지스터들을 체크해서 정말 자신의 디바이스를 위해 그 인터럽트가 발생한 것인지를 확인하는 것이다. 확인 결과 자신의 디바이스가 인터럽트를 발생시킨게 아닌 것으로 판단되면 그 루틴은 그대로 종료하고, 같은 인

터럽트를 공유하는 두번째 디바이스의 서비스 루틴이 시작된다. 인터럽트를 발생시킨 디바이스를 발견할 때까지 이런식으로 계속 진행된다.

IRQ 라인에 전압을 가하는 것은 단지 CPU 로 하여금 하던 일을 멈추고 디바이스 드라이버를 실행시킬 것을 요구하는 것이다. 거의 대부분의 경우, CPU 는 요청대로 멈추게 될것이다. 하지만 때로는 일시적으로 인터럽트를 요청할 수 없다거나 인터럽트들이 우선순위순으로 처리되는 수가 있으며, 따라서 아주 드문 경우지만 인터럽트가 발생되지 않거나 지연되는 수가 있다. 그러므로 위에서 "인터럽트" 라고 한것은 더 정확히 말하면 인터럽트 요청이라고 말해야 맞을 것이다. IRQ 가 Interrupt Request 의 약자인 것은 이런 이유 때문이다.

9.5. PCI 인터럽트

최근 PCI 인터럽트에서는 두가지가 새로이 개발되었으며 이는 여기서 다루지 않는다. 이 개발된 사항들은 특히 둘이상의 CPU 를 사용하는 컴퓨터에서 중요하다. 하나는 Advanced Programmable Interrupt Controller(APIC) 라는 것이다. 다른 하나는 Message Signalled Interrupts(MSI) 라는 것으로, (인터럽트 라인이 필요없이) 직접 메인 컴퓨터 버스상의 특별한 어드레스로 보내진 어떤 메시지 자체가 바로 인터럽트가 되는 것이다. 하지만 그런 메시지를 보내는 디바이스는 반드시 먼저 메인버스의 제어권을 획득해야 인터럽트 메시지를 보낼수 있다. 이런 메시지에는 "나 인터럽트 보내고 있어요" 라는 것 이상의 정보가 포함된다.

원래 PCI 인터럽트들은 ISA 인터럽트들과는 다르지만, 보통은 ISA 인터럽트처럼도 동작할 수도 있도록 IRQ 번호들로 매핑되어있다. 한가지 주요한 차이점은 BIOS 가 이 매핑을 담당한다는 점이다. CMOS 메뉴가 허용하지 않는한 리눅스에서 이 매핑을 변경하는 것은 불가능한 듯 하다 ??. 또다른 주요한 차이점은 PCI 인터럽트들은 공유될수 있다는 것이다. 예를 들면 IRQ5 를 두 PCI 디바이스가 공유하는 경우이다. 이러한 공유능력은 하드웨어에 내장되어 있으며, 모든 디바이스 드라이버들은 이 기능을 지원하기로 되어있다. 주의할 것은 PCI 와 ISA 버스간에는 같은 인터럽트를 공유할 수 없다는 것이다. 하지만 충돌상태에 있는 디바이스들이라도 같은 순간에 사용중이지만 않으면, 이러한 불법적인 공유도 제대로 동작할 것이다. 여기서 "사용중" 이라는 뜻은, 실행중인 어떤 프로그램이 그 C 프로그래밍 코드내에서 디바이스를 "개방(open)" 했다라는 의미이다.

여기서는 PCI 인터럽트 시스템에 대한 세부사항을 조금 언급한다. 각 PCI 카드(그리고 마더보드상에 탑재된 디바이스)는 4 개의 가능한 인트럽트들을 가진다; INTA#, INTB#, INTC#, INTD#. 지금부터는 그냥 A, B, C, D 라 부르겠다. 각각마다 카드의 커넥터 부분의 핀들중 하나씩을 차지한다. 따라서 7 개의 슬롯을 가진 시스템이라면 카드들 용으로 7 x 4 = 28 개의 서로 다른 인터럽트 라인들이 가능할것이다. 하지만 사양상의 제약으로 인해 이 인터럽트 라인들중 몇개만 허용되며, 많은 PCI 버스들은 단지 4개의 인터럽트 라인들만을 가지도록 제작되는 듯 하다. 하지만 인터럽트들은 공유될수 있으므로 4 개라도 지나치게 적은 것은 아니다. 이 라인(회로 배선)들을 W, X, Y, Z 이라고 부르자. 슬롯 3 으로부터의 B 인터럽트를 인터럽트 3B 라는 식으로 표시하기로 하자.

이 라인들을 인터럽트와 결합시키는 간단한 방식은 모든 A 인터럽트들(INTA#)은 라인 W 에 연결, 모든 B 인터럽트들은 라인 X 에 연결, ... 이런 방식일 것이다. 실제로 수년 전에는 이 방식이 사용되기도 했지만, 이는 좋은 해결책이 아니다. 이유는 다음과 같다. 위와 같은 방식이라면, 만약 어떤 카드가 인터럽트 한개만 필요로 할 경우 A 를 사용할 것이다. 만일 두개의 인터럽트를 필요로 한다면 A 와 B 를 사용해야만 할 것이다. 이런 식이라면 결국 INTA# 는

INTD# 보다 훨씬 자주 사용되게된다. 따라서 첫번째 라인을 공유하는 인터럽트들의 수가 너무 과도해지는 문제가 생긴다(모든 INTA# 들은 W 에 연결되어 있다). 이 문제를 극복하려면 보다 복잡한 방식을 택해 네개의 인터럽트 라인들(W, X, Y, Z)이 다들 대략 같은 수만큼의 인터럽트들을 공유케 하면 된다.

그 한가지 방법은 W 라인이 인터럽트 1A, 2B, 3C, 4D, 5A, 6B, 7C 들을 공유하게 하는 것이다. 이는 라인 W 를 물리적으로 1A, 2B, ... 들과 연결시키면 된다. 같은 식으로 라인 X 는 1B, 2C, 3D, 4A, 5B, 6C, 7C 들과 연결시킨다. 나머지도 같은 식이다. 이렇게 해두면 시스템 시작시에 BIOS 가 X, W, Y, Z 을 IRQ 들에 매핑시킨다. 그다음, BIOS 는 디바이스들이 사용할 IRQ 값들을 각각의 디바이스상의 하드웨어 설정 레지스터에 써넣는다. 여기까지 완료되었다면 이제 어떤 레지스터를 조사한 프로그램은 그 디바이스가 무슨 IRQ 를 사용하는지 알아낼 수 있게된다.

슬롯에 꽂힌 하나의 카드는 그 카드상에 디바이스를 8 개까지 가질수 있다. 하지만 그 카드가 쓸수있는 인터럽트는 단지 4 개 뿐이다(A, B, C, D). 인터럽트는 공유될 수 있으므로 8 개의 디바이스들(만약 8 개 까지 탑재되어 있다면) 각각이 인터럽트를 하나씩 가지는 것도 가능하다. 한 디바이스의 PCI 인터럽트 문자는 보통 고정되어 있으며 그 디바이스에 하드웨어적으로 배선되어있다. 인터럽트들을 할당하는 것은 앞서 언급한 바와 같이 BIOS 가 ISA 인터럽트들을 PCI 인터럽트들에 매핑시킴으로써 이루어진다. 만일 위의 예에서처럼 단지 4 개의 라인(W, X, Y, Z)만 있다면 PCI BIOS 가 선택의 폭은 그 수에 제한된다. 어떤 마더보드들은 더 많은 라인들을 사용할 수도 있으며, 따라서 이경우 선택도 더 다양해진다. BIOS 는 어떻게 결선되어 있는지에 대해 알고있다.

BIOS 는 자신이 ISA 버스용으로 파악하고 있는 IRQ 들과 충돌하지 않도록하면서 PCI 버스상의 IRQ 들을 할당해 나간다. 때로는 CMOS BIOS 메뉴를 통해 사용자가 PCI 카드들에게 IRQ 를 할당해줄 수도 있다. 또한, PnP 운영체제(가령 MS 윈도우즈)라면 우선 BIOS 가 해놓은 작업을 파악한 후 IRQ 들을 할당하려고 시도할 수도 있다. 이러한 할당내역을 "라우팅 테이블" 이라고 부른다. MS 윈도우즈가 IRQ 할당작업을 동적으로 해내는 경우(가령 도킹 이벤트), 이를 "IRQ 스티어링(steering)" 이라고 부른다. BIOS 는 BIOS 자체의 IRQ 스티어링 기능을 가지고 있을수도 있다(리눅스가 이를 활용할 수도 있다). 만일 윈도우즈로 부팅한 후 PC 를 끄지 않은 채 그 상태에서 리눅스를 시작시킨다면, 그 IRQ 들은 BIOS 가 셋팅시킨 것과 다를 수도 있는 것이다.

PCI 가 ISA 버스상의 IRQ 들을 사용할 경우, 당신은 ISA 버스의 속도는 느리니까 시스템이 느려질거라고 생각할 수도 있겠다. 실제로는 그렇지 않다. ISA 인터럽트 컨트롤러 칩(들)은 CPU 로 향하는 직접적인 인터럽트 배선을 가지기때문에 CPU 는 즉각 신호를 알아챌 수 있다. ISA 어드레스상의 신호나 데이터 버스들상의 신호라면 CPU 에의 도달 시간이 느릴 수도 있지만, IRQ 인터럽트 신호들만큼은 거의 즉각적으로 CPU 에 도달한다.

9.6. ISA Isolation

이것은 ISA 버스에만 해당되는 내용이다. Isolation 이란 일시적인 핸들(id 번호, 즉, 카드 선택번호(CSN, Card Select Number))을 ISA 버스상의 PnP 디바이스들 각각에 할당시키는 복잡한 방법을 말한다. 일부 사람들은 이를 위해 더 효율적인(그러나 더 복잡한) 방법을 사용할 수 있으므로 isolation 는 간단한 방법이라고 주장하기도 한다. 모든 PnP 디바이스들에 대한 PnP 쓰기용으로 오직 하나의 쓰기 어드레스만이 사용되므로, 이 어드레스에 무언가를 쓰면 모든

PnP 디바이스들이 이에 귀기울이게 된다. 이 쓰기 어드레스는 각 PnP 디바이스들에게 그 디바이스 고유의 유일무이한 핸들을 보내는데(즉, 할당하는데) 사용된다. 이 핸들을 할당하려면, 핸들이 모든 디바이스들에 공유되고있는 그 쓰기 어드레스로 보내질 때(즉, 쓰여질 때) 오로지 하나의 디바이스만이 이에 귀기울이고 있을것이 요구된다. 모든 PnP 디바이스들은 isolation 과정에서 사용될 고유의 유일무이한 시리얼 넘버를 가지고 있다. isolation 을 행하는 것은 일종의 게임과 비슷하다. PnP 디바이스들 전부가 공유하고있는 단 하나의 버스라인을 isolation 프로그램과 결부시키는 게임이다.

"게임" 의 제 1 회전에서는 모든 PnP 디바이스들이 이 라인에 귀기울이고 있는 상태이며, 또한 각 디바이스들은 동시에 이 라인으로 비트열을 보내게 된다. 허용되는 비트들은 1 (양전압) 이거나 전압이 걸리지 않은 상태인 "open 0" 전압(회로가 개방되었거나 tri-state 상태)이다. 이를 위해 각 PnP 디바이스는 자신의 시리얼 넘버를 이 라인상으로 높은 비트부터 비트단위로 순서대로 전송한다. 만약 어떤 디바이스가 1 을 보내면 다른 모든 디바이스들은 그 라인상에서 1 이 왔음을 듣게될 것이다. 만일 모든 디바이스들이 "open 0" 를 보낸다면 라인에서는 아무것도 들리지 않는 것이다. 목적은 (일회전이 끝났을 때) 가장 높은 시리얼 넘버의 디바이스만 남기고 나머지는 전부 제거시키는 것이다. "제거" 는 게임의 이번 회전에서 탈락하는 것을 말하며, 따라서 일시적으로 더이상 쓰기 라인에 귀기울이는 것을 중단하게 된다. (모든 시리얼 넘버들은 같은 길이임에 주목하라.) 만일 단 하나의 디바이스가 남아서 아직도 귀기울이고 있다면, 그 디바이스는 핸들(카드 번호)을 받게된다.

먼저 시리얼 넘버의 가장 최상위 비트만 고려하자. 아직 핸들을 받지 못한 상태인 모든 디바이스들이 각자의 최상위 비트들을 이 라인으로 처음 보내오면, 이들은 중첩되어 하나의 비트가 된다. 만일 어떤 PnP 디바이스가 0(open 0) 을 보내고서 1 을 듣게 된다면, 이는 어떤 다른 PnP 디바이스가 더 높은 시리얼 넘버를 가지고 있다는 뜻이고, 따라서 이 디바이스는 일시적으로 이번 라운드에서 탈락한다. 이제 게임에서 살아남은(이번 회전에서만임) 디바이스들은 모두 같은 숫자(가령 1)의 앞자리 숫자를 가진 것이며, 따라서 우리는 이제 이 숫자를 떼버린 후 남는 "한 비트 벗겨낸 시리얼 넘버"를 가지고 이번 회전에 계속 참가하게 된다. 그다음 본 문단의 첫단계를 다시 반복하고, 이런 식으로 각 디바이스에 대한 모든 시리얼 넘버가 시험될 때까지 반복하는 것이다(다음 문단의 모두 0 인 경우를 보라).

따라서 이번 라운드에서는 낮은 시리얼 넘버를 가진 카드들은 모두 탈락될 것이 분명하다. 하지만 만약 게임에 참가한 모든 디바이스들이 모두 자신의 최상위 비트로서 0 을 내보내온 경우는 어떻게 될까? 이 경우 "open 0" 가 라인상으로 보내지게 되므로 모든 참가자들이 다 게임에 남게된다. 이들 모두가 최상위 비트로 0 을 가지고 있으므로 이는 비긴 것이며, 따라서 위의 문단에서 1 이 벗겨졌었던 것과 마찬가지로 최상위 비트인 0 은 벗겨내지게 된다. 이제 게임은 시리얼 넘버의 그다음 숫자가 보내지면서 계속 진행된다.

이번 회전의 마지막 단계(즉, 시리얼 넘버의 최하위 비트가 보내진 다음)에서는 가장 높은 시리얼 넘버를 가진 PnP 디바이스 하나만이 게임에서 살아남게 된다. 그러면 이 디바이스는 핸들을 할당받고 게임에서 영구히 빠지게 된다. 그다음, 이전 회전에서의 모든 탈락자들이(아직 핸들을 못받고 있다) 다시 게임에 참가하고, 이제는 참가자 하나가 줄어든 상태에서 새로운 회전이 시작된다. 결국 모든 PnP 디바이스들은 핸들을 할당받게 된다. 이 알고리즘이 제대로 동작한다는 것은 쉽게 증명할 수 있다. 실제의 알고리즘은 이보다 다소 복잡하다. 왜냐하면 신뢰성을 보장하기 위해 각 단계를 두번씩 반복시키며 그 반복도 다소 달라지기 때문이다(하지만 기본적으로 같은 아이디어이다).

일단 모든 핸들들이 할당되면, 이 핸들들은 각 PnP 디바이스가 설정 데이터를 읽고 쓰는데

사용할 어드레스를 할당하는데 사용된다. 이 핸들들은 오직 PnP 설정용으로만 사용될 뿐, PnP 디바이스와의 통상의 통신용으로는 사용되지 않음에 유의하라. 컴퓨터가 PnP BIOS 를 시작할 때는 보통 이런 isolation 이 수행된 후에 PnP 설정이 수행된다. 여기까지 완료되면 모든 핸들들은 "폐기" 되며, 따라서 만일 사용자가 다시 설정을 바꾸려면(혹은 조사하려면), isolation 이 또다시 수행되어야만 한다.

끝