

# 5장. 어셈블리어를 이용한 시스템 프로그래밍 (Systems Programming in Assembly Language)

이 책을 읽는 독자들은 벌써 어셈블리어 프로그램과 인텔 프로세서의 구조에 대하여 능숙해 있을 것이다. 레지스터를 다루는 것, 스택에 값을 넣어 보존하는 것 등등의 기술들은 의심할 여지없이 훌륭한 것이다. 그러나 필자가 만난 대부분의 프로그래머들처럼, C 같은 좀더 고급언어를 사용하기를 원할 것이고, 기계적인 특성보다는 프로그램 로직에 좀더 신경 쓰기를 원할지도 모르겠다. 여기에 좋은 소식이 있다. 그것은 앞으로 마이크로소프트 윈도우즈 DDK나 Vireo System의 VTOOLS에 C나 C++을 사용하여 윈도우즈 95 시스템 프로그래밍을 할 수 있다는 것이다.

나쁜 소식도 있다. 그것은 앞으로 시스템 프로그램 작업에 대하여 어셈블리어를 최소한은 읽을 수 있을 정도는 필요하다는 것이다. 가끔은 프로젝트에서 큰 부분이나 작은 부분이나 어셈블리어를 사용해야할지 모른다는 것이다. 이것을 하기 위해, 다음과 같이 80386이상의 프로세서에 대한 몇가지 어려운(비밀스런) 사양들에 대하여 이해할 필요가 있다.

- 리얼모드, 보호모드, 가상 8086(V86) 모드의 주소 지정방식의 차이점
- 16비트 레지스터와 32비트 레지스터의 사용법
- 인텔 프로세서의 인터럽트 처리 방법
- 윈도우즈 95의 다양한 시스템 수준의 레지스터와 데이터 구조체의 사용법 (디스크립터 테이블, 태스크 스테이트 세그먼트와 같은 것)

프로그래밍을 위한 어셈블리 언어에 대하여 기계 수준까지 알기 위한 모든 것을 설명하려면 이 책보다 더 많은 지면이 필요하다. 이 장에서는 인텔 프로세서의 32비트 보호모드 프로그래밍의 중요한 사항에 대하여 우선 알아보고, 윈도우즈 95에서 사용하는 프로세서의 사양에 대하여 좀더 자세히 설명한다. 이러한 주제에 대하여 인텔의 programmer's reference manual이 가장 좋은 참고서일 것이다. 필자는 i486 매뉴얼을 강력히 추천한다. 이 책은 이전 버전이나 이후 버전의 프로세서들에 대한 책 중에서 가장 체계화가 잘되어 있으며 가장 정확히 설명되어 있다. 공부하기로 좋은 책은 Igo Chebotko가 쓴 "Assembly Language Masetr Class" (XROX 출판, 1995)다. 이 책은 꼭 윈도우즈에 얽매이지 않고 몇 개의 시스템 프로그래밍의 주제에 대하여 설명하고 있다.

## 5.1 주소지정 모드 (Addressing Mode)

독자들도 알겠지만, 어떤 어셈블리 프로그램이든지 메모리 연산 명령이 있다. 인텔 프로세서는 세그먼트 레지스터와 한 개 혹은 두 개의 범용 레지스터에 들어있는 오프셋의 조합으로 메모리 어드레스 연산을 만든다. 80386은 리얼모드, 보호모드, V86 모드의 주소지정 모드를 제공함으로써 프로세서 호환을 유지하고 있는데, 프로세서가 메모리 어드레스의 세그먼트를 어떻게 해석하는가 하는 것은 어떤 모드에서 동작하고 있는가 하는가에 달려있다.

### 5.1.1 리얼모드 (Real Mode)

리얼모드(real-address mode, 이하 real mode)에서, 프로세서는 8086이나 8088이 했던 것과 같은 방법으로 주소를 만든다. 그것은 그림 5-1과 같이 세그먼트 레지스터에 있는 물리 페러그라프 주소와 명령어 오퍼랜드로부터 주어지는 오프셋을 더한다. 실제로는 옵션 카드나 BIOS 모듈에서 존재하는 메모리로 인하여 발생하는 예약영역은 1MB의 주소지정 영역을 640KB로 줄여 버린다.

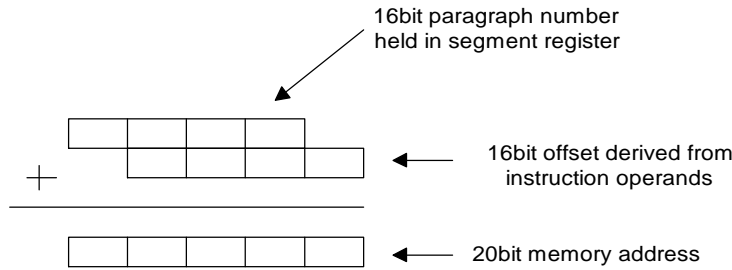


그림 5-1. 리얼모드에서의 어드레스 연산

컴퓨터는 켜져서 윈도우즈 95나 다른 보호모드 프로그램이 시작하기 전까지는 리얼모드에서 동작한다. EMM386과 같은 메모리 관리자는 보호모드 프로그램이다. 따라서 리얼모드에서의 전환은 생각보다 빨리 일어날 것이다. 그러나 BIOS의 POST(power-on self test)와 MS-DOS의 초기화 과정은 리얼모드에서 이루어진다.

응용 프로그램 프로그래머, 특히 게임 개발자는 가끔 리얼모드의 플랫폼을 더 좋아한다. 왜냐하면 프로그램이 할 수 있는 것에 대하여 구속이 없기 때문이다. 리얼모드 프로그램은 어떤 레지스터나 1MB이하의 어떤 메모리 주소든지 제한 없이 액세스 가능하며 어떤 명령어든지 거의 실행할 수 있다. 예를 들면, 이러한 자유는 그래픽 지향 게임에서는 최고의 성능을 이룰 수 있도록 해준다. 그러나 비록 게임 개발자라고 하더라도 그들의 마술과 같은 작업에도 메모리가 필요하며, 리얼모드의 640KB의 제한으로 인하여 보호모드에서 실행되는 소프트웨어를 작성하는 것에 대하여 매우 큰 유혹이 될 수밖에 없는 것이다.

### 5.1.2 보호모드 (Protected Mode)

다음의 그림 5-2와 같이, 보호모드에서 프로세서는 리얼모드와는 대체적으로 다른 방법으로 어드레스를 구성한다. 세그먼트 레지스터는 선택터를 가지게 되는데, 선택터는 디스크립터 테이블(descriptor table)에서 디스크립터를 선택하기 때문에 이렇게 불린다. 디스크립터 테이블에는 2가지가 있는데 GDT(global descriptor table)과 LDT(local descriptor table)이 그것이다. 선택터는 어느 테이블의 인덱스 인자를 나타내는 필드를 나타내고 있다. GDT는 운영체제나 모든 가상 머신을 포함하는 디스크립터들을 가지고 있으며, LDT는 현재의 가상 머신을 포함하는 디스크립터들을 가지고 있다.

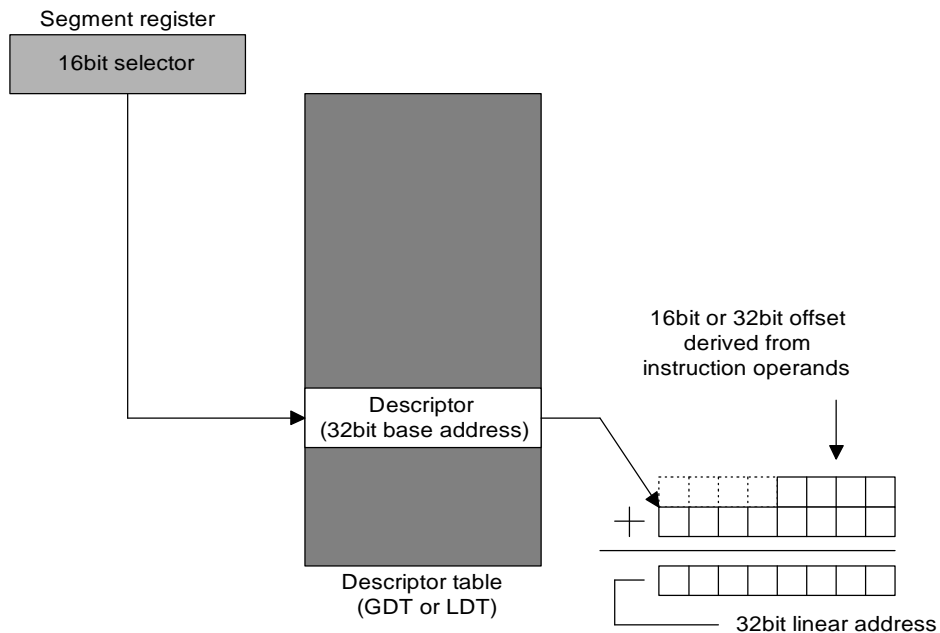


그림 5-2. 보호모드에서의 어드레스 연산

디스크립터는 그림 5-3과 같이, 베이스 어드레스, 세그먼트 리미트, 메모리 액세스 허용타입을 관리하는 액세스 콘트롤 플래그들을 가지고 있다. 명령어는 디스크립터의 베이스 주소와 명령어 오퍼랜드로부터 주어지는 오프셋을 더해서 가상 어드레스(간혹 선형주소라 부름)를 형성한다. 베이스 주소는 32비트의 크기를 가질 수 있기 때문에 보호모드 프로그램은 메모리의 4GB( $2^{32}$ 바이트)까지 액세스 할 수 있다.

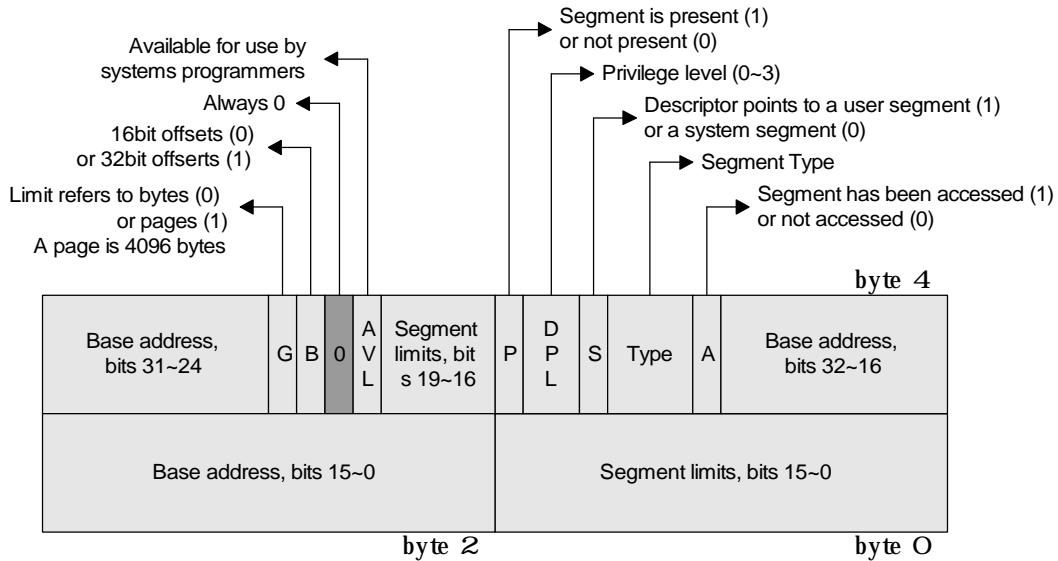


그림 5-3. 세그먼트 디스크립터 포맷

실제적으로 대개의 컴퓨터가 아직은 4GB의 물리 메모리를 보유할 수 없기 때문에, 운영체제는 물리 메모리가 없는 영역을 임시적으로 디스크 스왑 파일을 사용한다. 비록 컴퓨터가 8MB나 16MB의 메모리만 가지고 있다고 하더라도 응용 프로그램은 4GB까지의 가상 메모리를 액세스 할 수 있다. 그러나 이것은 물리 메모리가 어디에 있든간에 운영체제가 가상 메모리 페이지를 재배치할 수 있어야 한다. 가상주소(응용 프로그램에게는 변함없이 고정적이다)는 프로세서가 페이지 테이블을 이용하여 물리 주소로 변환하는데, 이 페이지가 물리 메모리의 어디에 위치해 있는지에 따라 변한다. 다음의 그림 5-4는 이 변환처리를 보여주고 있다.

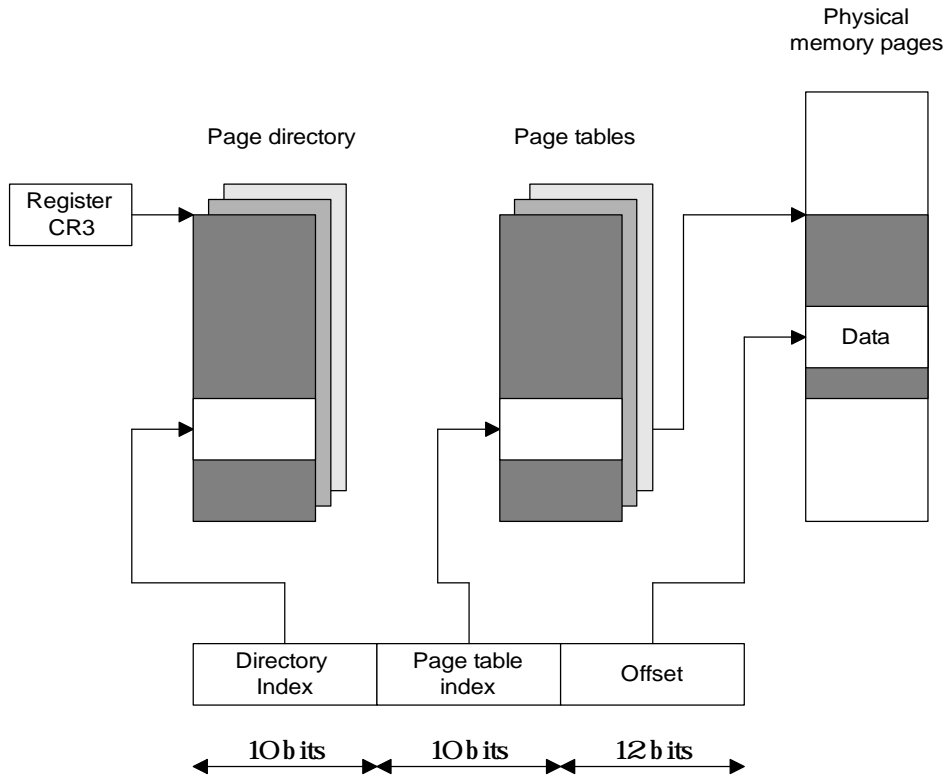


그림 5-4. 가상 어드레스에서 물리 어드레스로의 변환

**용어설명 (Note on Terminology)**

보호모드 어셈블리어 프로그램에 관계된 일을 할 때, 흔히 용어에 대하여 어물쩍 넘어가는 경우가 있다. 엄격히 말해서, 세그먼트 레지스터는 GDT나 LDT에 있는 세그먼트 디스크립터를 참조하는 세그먼트 선택터를 가진다. 따라서 정확하게 “선택터 리미트”와 같은 말은 절대로 사용하지 말아야 할 것이다. 왜냐하면 리미트는 디스크립터에 기록되어 있는 세그먼트 속성이기 때문이다. 어쨌든, 이제 이 책을 사서 보므로 인해서 탁상공론의 온상에서 비틀거리지 않게 되었음을 알리게 되어 기쁘다. 필자는 정확성을 요구하는 곳을 제외하고서라도 최대한 모호함을 없앨 것이다. 어떤 모든 경우에 있어서 선택터와 디스크립터는 일대일로 대응되며, 이 일대일 대응으로 정의된 유질동상(類質同像)이 대부분에 걸쳐 동일시된다는 것을 어릴 때 배웠다. 따라서 이것이 결국 선택터와 디스크립터를 혼동해서 생긴 일이 아니라는 것이다.

**(1) 세그먼트 액세스 제어 (Segment Access Controls)**

보호모드의 “보호”라는 말은 명령어를 실행하는 동안 프로세서가 자동으로 수행하는 몇 가지 비준절차로 온 것이다. 비준절차는 디스크립터에 포함된 세그먼트 속성을 주기적으로 검사하는 것을 말한다. 앞의 그림 5-3에서, 디스크립터의 S비트와 Type 필드는 이 컨트롤의 첫 번째 수준을 제공한다. “유저” 세그먼트(S비트=1)만이 소프트웨어에 의해 직접 액세스 가능하다. 다른 종류의 세그먼트는 “시스템” 세그먼트이다. 그러나 운영체제 내부의 핵심적인 요소를 만들지 않는다면 시스템 세그먼트에 대하여 자세히 알 필요는 없다. 유저 세그먼트는 그림 5-5와 같이 프로그램 명령이 포함된 코드 세그먼트와 프로그램이 사용하는 데이터가 포함된 데이터 세그먼트를 가질 수 있다. 어떤 프로그램도 코드 세그먼트의 내용을 변경할 수 없으며, 마찬가지로 어떤 프로그램도 데이터 세그먼트에 있는 명령을 실행시킬 수 없다. 단지 세그먼트가 읽기 속성을 가지고 있다면 프로그램은 코드 세그먼트로부터 데

이터를 읽을 수 있으며, 세그먼트가 쓰기 속성을 가지고 있다면 프로그램은 데이터 세그먼트에 기록할 수 있다. 윈도우즈 95는 보통의 경우 코드 세그먼트에 읽기 속성이 있으며, 데이터 세그먼트에 쓰기 속성이 있다. 이것은 비록 응용 프로그램이나 VxD가 이 속성을 빼먹었다 하더라도 이렇게 설정된다. Type 필드의 또 다른 속성은 컨퍼밍과 익스팬드다운인데, 이것은 이 장의 다른 곳에서 설명한다.

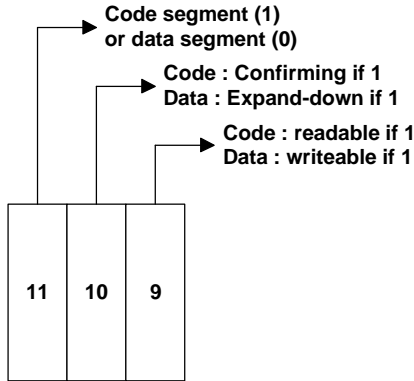


그림 5-5. 유저 세그먼트 디스크립터의 타입 필드

덧붙인다면, 프로세서는 액세스(읽기, 쓰기, 실행)를 시도한 명령의 타입이 대상 세그먼트에 적당함을 검사하며, 또한 액세스한 가상 메모리가 세그먼트로 정의되어 있는지를 확인한다. 앞의 그림 5-3에서와 같이, 디스크립터에 있는 세그먼트 리미트 필드는 20 비트의 숫자로서, 사용할 수 있는 세그먼트의 용량보다 1이 작은 숫자이다. 예를 들어 64KB의 세그먼트는 리미트로 0FFFFh의 값을 가지는 경우, 프로세서는 64KB를 초과하는 세그먼트의 데이터는 액세스할 수 없도록 한다. 세그먼트의 크기를 1M(220바이트)보다 크게 하기 위해서, 인텔은 그림 5-3과 같이 디스크립터의 그랜놀러티티 비트(granularity bit, 이하 G 비트)를 사용한다. 만약 G 비트가 1로 설정되어 있으면 세그먼트 리미트의 단위가 4096 바이트로 되고, 0으로 설정되어 있으면 세그먼트 리미트의 단위가 1 바이트가 된다. 따라서 리미트의 값이 FFFFFh의 경우, G 비트가 1로 되어 있으면 4G 바이트이며, G 비트가 0으로 되어 있으면 1MB이다.

세그먼트 리미트 체크라는 것은 참조하는 명령어가 오프셋 0과 세그먼트의 끝 사이에 있는지를 확인하는 것을 말한다. 그러나 그림 5-5와 같이 데이터 세그먼트는 익스팬드다운(Expand-down) 속성을 설정할 수 있는데, 익스팬드다운 세그먼트의 리미트는 1이다. 윈도우즈 95는 NULL 포인터나 NULL 포인터에 가까운 32비트 주소를 잡아내기 위해 이런 똑똑한 개념을 사용한다. 이러한 프로그램들이 익스팬드다운 속성이 설정되어 있고, 베이스 주소가 0이며 리미트가 0FFFFh인 데이터 셀렉터를 사용하면, 프로세서는 즉시 4096보다 숫자적으로 작은 데이터 포인터를 참조할 수 없게 한다. 이러한 방법은 프로그래머가 직접 NULL 포인터를 참조하는 것이나 어드레스가 NULL이거나 4096보다 작은 구조체의 필드를 액세스하는 것으로부터 보호할 수 있다.

---

### NULL 포인터에 대한 설명

NULL 포인터 참조에 대한 보호는, 윈도우즈 3.1의 Win32s 부속 시스템과 윈도우즈 95와는 다른 방법을 사용을 사용한다. Win32s에서 실행되는 32 비트 프로그램은 베이스 주소가 FFFF0000h이고 리미트가 FFFFFFFFh인 코드와 데이터 섹터를 사용한다. 어드레스 공간의 최후 64KB는 선형주소의 페이지 테이블에 대응되어 있지 않도록 해서, Win32는 NULL 포인터를 참조할 경우 페이지 폴트를 발생시킨다고 한다. 32비트의 10000h나 더 큰 포인터의 참조는 32 비트 주소의 잘림에 의해서 0에서 시작하는 유효한 선형 주소로 된다. 그러나 불행히도 페이지 폴트는 조용히 해결되지 않고, 이 NULL 포인터 참조를 올바르게 해석하는 것이 아니라, 초기의 Win32s는 이를 완전히 무시해 버렸다. 따라서 윈도우즈 95와 이전 버전의 윈도우즈와 호환되는 선형 주소를 32 비트 클라이언트에게 제공하는 VxD의 경우 다른 세그먼트의 베이스 주소까지 알아야 할 필요가 있다.

이것은 16비트 프로그램에서 NULL 포인터를 참조하기 때문이지만, 윈도우즈 95는 NULL 포인터를 잡아내기 위해 다른 방법을 사용한다. 또한 0 섹터를 사용한 메모리 참조도 할 수 없다. 16:16 NULL 포인터는 0000h:0000h인데, NULL 포인터를 우회 참조하거나 주소가 NULL인 구조체의 필드를 참조하는 것은 폴트를 발생시킨다. (16:16의 주소는 16비트 페러그래프나 섹터 요소와 오프셋 요소를 가리킨다.)

---

## (2) 일반 보호 폴트(The General Protection Fault)

지금까지는 보호모드 프로그램에 대하여 프로세서가 어떻게 액세스 제한을 하는지 일부러 자세하게 설명하지 않았다. 액세스 규칙을 위반하게 되면 일반 보호 폴트(GP Fault)가 발생한다. 일반 보호 폴트는 인텔 프로세서에 있어서는 당나라 군대와 같다. 프로세서가 GP 폴트를 일으키는 경우가 매우 다양하기 때문이다. 많은 경우 프로그램 버그로 인해 발생하지만 보통의 윈도우즈 프로그램에서도 많이 발생한다. 아래에 GP 폴트가 발생하는 경우를 간단히 나타내었다.

- DS, ES, FS, GS에 유효하지 않은 섹터를 로드 하려고 시도하거나, 그 디스크립터의 특권 수준이 현재 프로그램의 특권 수준보다 더 높은 섹터를 로드 하려고 시도하는 경우.
- 섹터가 0인 16:16 NULL 포인터를 참조하려고 시도하는 경우. 그러나 데이터를 참조하지 않는다면 NULL 섹터를 로드 하는 것은 가능하다.
- 다른 특권 수준의 프로그램을 호출하는 경우
- 디스크립터에 기록된 세그먼트 리미트를 초과하는 코드나 데이터를 참조하려고 시도하는 경우.
- 읽기 전용 데이터 세그먼트에 기록하려고 하거나, 실행 전용 코드 세그먼트를 읽으려고 하거나, 데이터 세그먼트를 실행하려고 하거나, 코드 세그먼트에 기록하려고 시도하는 경우.
- IDT(Interrupt Descriptor Table)의 리미트를 넘어서는 인터럽트를 호출하려고 시도하는 경우. 프로그램이 소프트웨어 인터럽트 60h나 그 이상의 인터럽트를 발생시키는 경우 항상 GP폴트가 발생하므로 반드시 프로그램 에러는 아니다.
- 보호모드 응용 프로그램이 CLI나 STI와 같은 머신 명령을 실행하려고 시도하는 경우. 이 폴트는 빈번히 발생하는데, 윈도우즈는 이 폴트를 잡은 다음, 영향이 미치는 가상 머신에게 적당하게 가상 인터럽트 플래그를 세팅한다.

---

### 수 논리에 관한 글 (Numerologic Note)

비록 부적절할지도 모르겠지만, 일반 보호 폴트가 프로세서 예외 인터럽트 0Dh라는 것은 매우 재미 있는 사실이다. 이것은 이 제앙의 수가 IBM OS/360의 비정상 종료 슈퍼바이저 호출(ABEND, SVC 13)과 같다는 것이다.

이와 비슷하게, 어떤 시스템 프로그래머가 내가 어디 살고 무슨 일을 하는지에 대한 슬랭코드를 넣는 것과 같이 GP 폴트로 인해 프로그램이 죽을 경우 Oh Dees라는 문장이 포함되어 있는 것을 관찰할 수 있다.

---

### (3) 특권 링(Privilege Rings)

프로세서는 특권 링이라 불리는 4개의 특권 수준을 제공하는데, 이것은 시스템 자원을 필요로 하는 제어 등급에 따라 소프트웨어를 나누기 위함이다. 이들 중에서 윈도우즈는 2개의 특권 수준을 사용한다. 운영체제 슈퍼바이저는 신임도가 제일 높은 0순위 권한(ring zero)에서 실행한다. 0순위 권한 코드는 모든 메모리와 레지스터의 값을 바꿀 수 있다. 응용 프로그램은 신임도가 가장 낮은 3순위 권한(ring three)에서 실행된다. 3순위 권한의 프로그램은 시스템 제어용 레지스터를 액세스 할 수 없으며, 운영체제가 보호 되도록 설계한 메모리 영역은 읽거나 기록할 수 없다. 또한 프로세서는 3순위 권한의 프로그램이 앞에서 언급한 CLI나 STI와 같은 특정 명령을 실행시키면, 이 명령을 에뮬레이션 하거나 응용 프로그램이 그것들을 수행하는 때와 방법을 제어하기 위해서 이를 가로챈다.

특권수준에 대하여 좀더 자세히 알아보기 위해, 복잡한 용어들에 대하여 확실히 알아 볼 것이다. 우선 윈도우즈 95는 프로세서에서 제공하는 특권수준의 특징들을 사용하는데, 윈도우즈 95에서는 이들 중 많은 부분을 무시할 수 있다. 그림 5-6과 같이, 선택터의 하위 두 비트는 RPL(Requested Privilege level)이며, 이론적으로는 대응되는 DPL(Descriptor Privilege Level)과 서로 다를 수 있다. CS 세그먼트에 있는 선택터의 RPL을 CPL(Current Privilege Level)이라 하며, 실행되는 프로그램이 가지고 있는 특권과 같다.

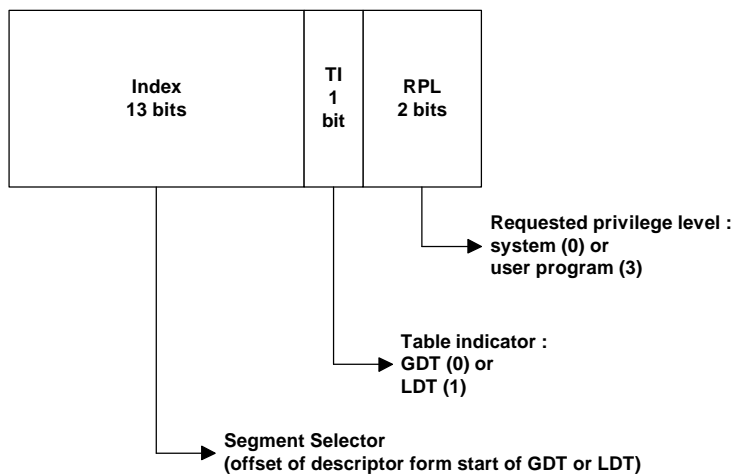


그림 5-6. 선택터의 구조

RPL, DPL과 같거나 혹은 다른 권한의 CPL을 가진 프로그램을 관리하는 방법에는 정교한 규칙이 있다.(이전에 서로 다른 선택터와 디스크립터에 대해서 잔뜩 얘기한 것에 대하여 기억하고 있는가? RPL은 선택터의 속성이며, DPL은 디스크립터의 속성을 알고 있을 것이다. 그러나 저자는 이것의 결과가 무엇을 의미하는지 완전히 이해하지 못하고 있다.) 앞서서도 말했듯이, 윈도우즈 95에서는 이러한 규칙을 대부분 무시할 수 있는데, 0순위 권한 프로그램은 CPL, RPL, DPL이 모두 0이며, 3순위 권한 프로그램은 CPL, RPL, DPL이 모두 3이기 때문이다. 0순위 권한 프로그램은 대개 GDT 선택터를 사용하는 반면, 응용 프로그램은 LDT 선택터를 사용한다. 따라서 시스템 소

프트웨어는 코드 선택터로 28h(0순위 권한의 GDT 선택터)를 사용하는 것을 볼 수 있으며, 응용 프로그램은 코드와 데이터 선택터로 xxx7h나 xxxFh(3순위 권한의 LDT 선택터)를 사용하는 것을 볼 수 있다.

프로세서는 다른 권한에서 실행되는 프로그램의 코드나 데이터의 액세스에 대하여 엄격히 제한하고 있어서, 0순위 프로그램을 직접 호출할 수 없다. 따라서 운영체제와 통신하기 위해, 인터럽트를 이용한 권한 변경을 해야 한다. 인텔 아키텍처는 게이트 디스크립터(Gate Descriptor)와 컨포밍 코드 세그먼트(Conforming code segment)를 제공한다. 게이트 디스크립터는 시스템 세그먼트의 일종이며, 컨포밍 코드 세그먼트는 디스크립터 타입 필드의 컨포밍 속성을 설정하여 사용한다. 이들을 사용하면 권한에 관계없이 직접 코드를 호출할 수 있다. 그러나 윈도우즈는 이들 중 아무 것도 사용하지 않는다.

0순위 권한 프로그램은 모든 특권 레벨에 있는 데이터를 액세스할 수 있으나, 다른 순위에 있는 코드는 직접 실행할 수 없다. 따라서 사용자 코드를 실행하기 위해, 0순위 권한 슈퍼바이저는 다른 권한 프로그램의 모든 상태가 저장된 특정한 스택에서 IRET 머신 명령을 실행해야 한다.

### 5.1.3 가상 8086 모드 (Virtual 8086 Mode)

가상 8086 모드(Virtual 8086 Mode, 이하 V86 모드)는 도스나 다른 리얼모드 응용 프로그램을 위해 만들어진 일종의 보호모드이다. V86모드에서 세그먼트 레지스터의 패러그래프 숫자와 명령어에서 주어지는 오프셋을 합치는 형식으로 동작함으로써 리얼모드의 어드레스 만드는 방법과 동일하게 동작한다. 리얼모드와의 차이점은 이 20비트의 주소가 물리 주소가 아니라 가상 어드레스라는 것이다. 따라서 V86 어드레스는 다른 모든 보호모드 주소와 같은 방법으로 페이지 변환을 한다. 이러한 사실은 윈도우즈나 다른 시스템 소프트웨어가 익스텐디드 메모리를 리얼모드 프로그램이 액세스할 수 있는 1MB이하 주소로 맵핑할 수 있다는 것이다. QEMM, 386Max나 다른 80386 메모리 관리자는 확장 메모리를 프로그램이 액세스할 수 있는 1MB이하 주소로 만들기 위해 이러한 방법을 사용하며, 윈도우즈 95도 MS-DOS 가상 머신의 개념에 같은 방법을 도입하고 있다.

만약 원한다면, 운영체제는 V86 프로그램을 깊숙이 관리할 수 있다. V86 프로그램은 모두 3 순위 권한에서 동작한다. 이것은 특권 명령, 특권 레지스터, 특권 데이터 영역의 액세스 시도는 일반 보호 폴트를 발생시킨다는 것을 의미한다. 또한 슈퍼바이저는 3보다 작은 값을 EFLAGS 레지스터의 IOPL(IO privilege level) 비트로 설정해서, INT, PUSHF, POPF, IRET, CLI, STI등의 불순한 명령에 의해 일반 보호 폴트를 발생시킬 수 있다. (이런 명령들은 흔히 인터럽트 플래그의 저장과 복구에 사용된다). 인텔의 의도는 운영체제에게 이런 가상화된 인터럽트 플래그를 제공하는 것이었다. 그러나 마이크로소프트는 이런 수행을 하게 되면 MS-DOS 프로그램이 15% 정도 느려지는 것을 발견했다. 그래서 윈도우즈 95는 IRET 가상 머신 명령을 잡아내려고 하는 경우를 제외하고는 보통 IOPL을 3으로 설정한다. 그렇기 때문에, MS-DOS 프로그램이 인터럽트를 디스에이블 시켜놓고 무한 루프를 돈다면 컴퓨터를 정지시킬 수 있다. PS/2 아키텍처에서는 와치독 타이머(Watchdog Timer)를 제공해서, 이런 경우 인터럽트를 발생하도록 할 수 있다. 그러나 표준 PC는 이러한 기능이 없으며, 이러한 종류의 프로그램 실수에 노출되어 있다.

### V86 모드 검출(Detecting V86 Mode)

만약, 작성하는 프로그램이 분명 MS-DOS에서 실행되는 것이라면, 잘 알려지지 않는 SMSW 명령을 사용하여 리얼모드와 V86 모드를 검출할 수 있다. 이 명령은 80286 명령인데, 80386과 이후의 프로세서에서도 제공하고 있다. 이 명령은 CR0 레지스터의 하위 2바이트를 얻는 것이며, 이 값의 하위 비트는 보호모드가 운영 중인것을 가리킨다.

```
pe_mask equ 1 ; CR0레지스터에서 PE(protected-execution)비트

smswax ; CR0의 하위부를 읽음
test ax, pe_mask ; PE 비트를 체크함
jnz InV86Mode ; 보호모드에 있으면 점프
```

호환성을 이유로, SMSW 명령은 특권 명령으로 되지 않았기 때문에, 3순위 권한 프로그램이 이 명령을 실행할 수 있다. 이것의 숨겨진 단편적인 로직은 이렇다: 우리는 프로그램이 MS-DOS에서 실행된다고 가정하면, 이것은 리얼모드나 V86 모드에서 실행됨을 암시하는 것이다. 만약 PE 비트가 설정되어 있으면 프로세서는 리얼모드가 아니기 때문에 V86 모드이어야 한다. 만약 작성하는 프로그램이 윈도우즈에서 실행되는 경우, V86 모드인지 보호모드인지를 검출하고 싶다면, 이러한 방법은 사용할 수 없다. 왜냐하면 SMSW 명령은 두 가지 경우 모두다 보호모드라 알려주기 때문이다. 이런 경우 두 모드를 구별하고 싶다면 INT 2Fh, 함수 1686h를 사용하면 된다.

```
mov ax, 1686h ; 함수 1686h
int 2Fh ; 모드 검출
test ax, ax ; 어떤 비트라도 설정되어있는가?
jz ProtectedMode ; 만약 아니면 보호모드임
```

인텔은 V86 모드가 프로그램에게 가능한 한 보이지 않도록 디자인했다. 사실, 프로그램이 V86에서 동작하고 있는지 직접적으로 알아내는 방법은 없다. 비록 EFLAGS 레지스터의 VM 비트가 V86 모드인지 아닌지를 제어하지만 이 말은 사실이다. EFLAGS 레지스터는 그림 5-8을 참고하기 바람 다음 32비트 프로그래밍 절에서 논의하기로 한다. 인텔은 PUSHFD 명령을 수행 수 있는데, EFLAGS 레지스터를 다음의 어셈블리 코드는 좋은 섹션 제대로 동작하지 않을 것이다. 만약, 16비트 프로그램에서 32 비트 레지스터의 사용에 익숙하지 않다면 다음절의 32비트 프로그래밍을 참조하기 바란다.

```
pushfd ; EFLAGS 레지스터를 스택에 넣음
pop eax ; EFLAGS의 이미지를 EAX에 저장
bt eax, 17 ; EFLAGS 이미지의 VM 비트를 테스트
jc InV86Mode ; 이것은 절대 발생하지 않는다.
```

비록 V86 모드가 응용 프로그램에 보이지 않도록 만들어 졌다고 하더라도, 윈도우즈 95에서 제공하는 서비스를 사용하면 정보를 얻을 수 있다. 이것은 V86 프로그램에서 ARPL(Adjust Requested Privilege Level) 명령을 실행하면 할 수 있다. 이 명령은 프로세서 예외(Inavliad operation processor exception)의 원인이 되며, 가상 머신 관리자는 예외처리의 인자로 넘겨오는 세그먼트:오프셋을 검사한 다음 VxD 서비스 루틴을 제어한다. IBM은 CP-67과 VM/370에 같은 방법을 사용하는데, 가상 머신 코드는 제어 프로그램과 통신하기 위해 특권 진단 명령을 실행한다.

## 5.2 삼십이 비트 프로그래밍

## (Thirty-Two-Bit Programming)

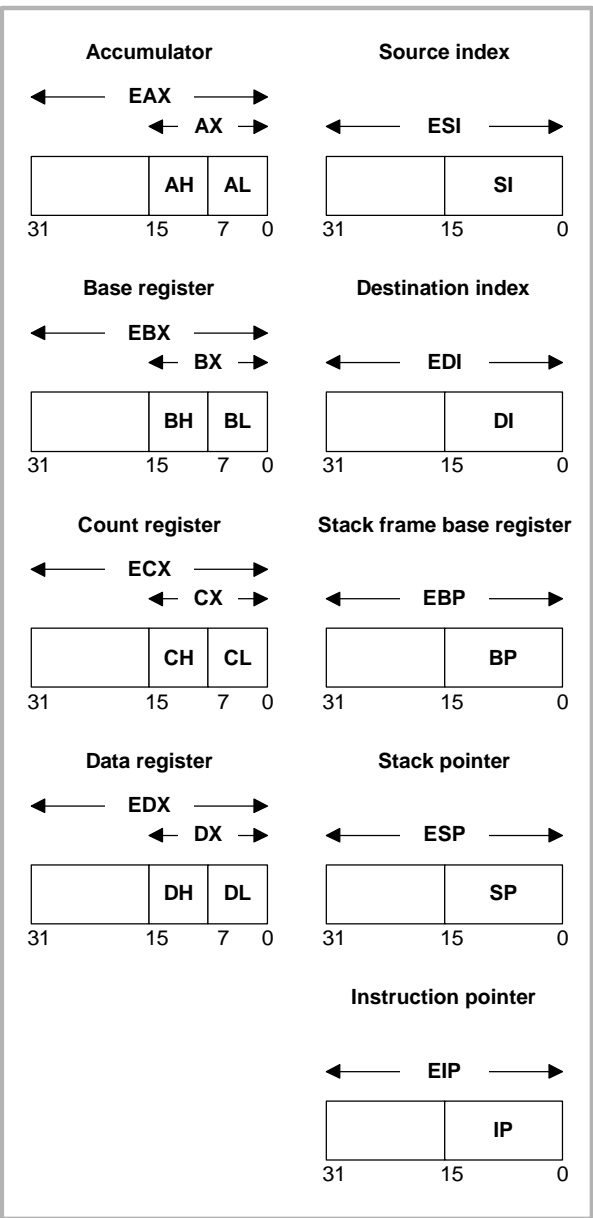
인텔의 80386 이상의 프로세서와 윈도우즈 95와 같은 진보된 운영체제의 중요한 점은 32비트 프로그램으로 작성할 때 많은 이익이 있다는 것이다. 이것은 향상된 성능을 제공하여 사용자들(End-user)에게 직접 이익이 되며, 프로그래머를 편리하게 하여 더 좋은 프로그램을 때에 맞춰 구할 수 있기 때문에 사용자들에게 간접적인 이익이 된다. 이전의 프로세서와 여기서 운영되던 이전의 운영체제에서 사용하던 16비트 프로그래밍과 32비트 프로그래밍과의 차이점은 확장 레지스터의 사용, 32비트 명령과 주소지정을 사용할 수 있다는데 있다. 더욱이, 16비트 프로그램은 코드나 데이터가 64KB를 넘느냐 넘지 않느냐에 따라 4가지 메모리 모델(small, compact, medium, large) 중 하나를 사용했지만, 32비트 프로그램은 다중 세그먼트가 없는 플랫 어드레스 모델(Flat address model)을 사용하기 때문에 세그먼트 레지스터의 변경 없이 코드나 데이터를 4GB까지 쉽게 액세스 할 수 있다.

### 5.2.1 확장 레지스터(Extended Register)

그림 5-7, 5-8과 같이, 80386 이상의 프로세서는 프로그램이 8개의 범용 목적 레지스터(줄여서 범용 레지스터라 부름)와 어떤 모드에서나 사용할 수 있는 6개의 세그먼트 레지스터와 플래그 레지스터를 제공하고 있다. 범용 레지스터와 플래그 레지스터는 32비트의 크기를 가지고 있으며, 어셈블리어에서 레지스터의 이름 앞에 E(Extend의 뜻)를 붙여 사용한다. 예를 들어 32비트의 EAX 레지스터는 다음과 같이 코딩한다.

```
xor  eax, eax      ; 32 비트의 확장 AX 레지스터를 지움
```

**General registers**



**Segment registers**

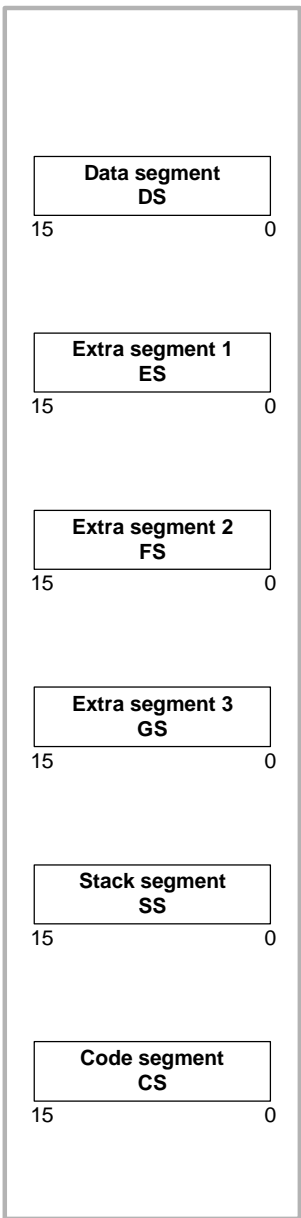


그림 5-7. 범용 레지스터와 세그먼트 레지스터



```

mov     eax, dividend
cdq                                ; 피젯수의 부호확장
idiv    divisor                    ; divisor를 이용한 나눗셈
mov     quotient, eax              ; 결과 저장

```

삼십이 비트 주소 지정은 훨씬 더 쉽다. 왜냐하면 64KB 경계를 지날 때 세그먼트를 다시 로드 해야할 필요가 없어 졌기 때문이다. 예를 들어, 윈도우즈에서 몇몇의 데이터 구조는 64KB보다 클 수 있다. (특히 그래픽 비트맵) 또한 16비트 프로그램에서 사용하던 거대(huge) 포인터들도 사용할 수 있는데, 윈도우즈의 *GlobalAlloc* 함수는 메모리 영역을 채우기 위해 7E7h, 7EFh, ...와 같은 8간격의 셀렉터를 연속적으로 대입할 것이다. 따라서 데이터 액세스에 대한 셀렉터 연산은 올바르게 동작한다. 아마 어떤 이유로 256 컬러의 비트맵에서 검정 픽셀 수를 세는 프로그램 작성을 원하게 될지도 모른다. 이때 픽셀의 값이 0인 경우를 검은색으로 가정한다면(실제로도 그렇다), 다음과 같은 코드가 될 것이다.

```

LONG CountBlack(BYTE __huge *lpBits, LONG nbits)
{
    // CountBlack
    LONG result = 0;
    LONG i;
    for( i = 0; i < nbits; ++i)
        if( !lpBits[i])
            ++result;
    return result;
}
// CountBlack

```

16비트 플랫폼에서 *lpBits[i]*의 참조를 C 컴파일러가 어셈블리 코드로 어떻게 만드는지 모른다면, 다음의 코드가 끔찍하게 느껴질지도 모르겠다.

```

mov     ax, WORD PTR i
mov     dx, WORD PTR i+2
mov     cx, WORD PTR lpBits
mov     bx, WORD PTR lpBits+2
add     ax, cx
adc     dx, 0
mov     cx, OFFSET __AHSIFT
shl     dx, cl
add     dx, bx
mov     bx, ax
mov     es, bx
mov     al, BYTE PTR es:[bx]

```

이 코드에서, *\_\_AHSIFT*는 외부 상수(실제로는 임포트 된 심볼)로써 3의 값을 가진다. 이 복잡하게 얽힌 코드는 *lpBits*의 제일 처음 셀렉터에 대하여  $(i/65536) * 8$ 을 더하며, 오프셋은  $i\%65535$ 를 사용하게 되는 것이다. 만약, 이전 버전의 윈도우즈에서 *BitBlit*호출이 왜 그렇게 느리던가의 이유를 생각해 보면 이제 알게 되었을 것이다.

같은 프로그램을 32비트 버전으로 만들면 더 작고 더 빠르게 되었을 것이다.

```

mov     eax, DWORD PTR I
mov     ecx, DWORD PTR lpBits
mov     edx, edx
mov     dl, BYTE PTR [eax+ecx]

```

32비트 주소지정은 큰 데이터의 조작을 하게 될 때뿐만 아니라, 배열의 관리에도 도움을 준다. 원래의 808x 프로세서는 베이스 레지스터와 인덱스 레지스터를 조합해서 어드레스를 만드는데 매우 제한적이었다. 이와는 대조

적으로 32비트 주소지정에서는 베이스 레지스터로서 어떤 레지스터든지 사용할 수 있으며, 인덱스 레지스터로 ESP를 제외한 어떠한 레지스터도 사용할 수 있다. 더구나, 인덱스 값에 1,2,4,8의 값을 곱해서 사용할 수 있다. 다음의 코드는 어드레스 지정에 있어 레지스터의 조합을 보여주고 있으며, 인덱스에 4바이트씩 곱해서 사용하는 것을 보여주고 있다.

```

mov     ecx, array      ; ECX는 long[] array의 주소를 가리킴
mov     edx, i          ; EDX = loop 횟수 (0부터 n까지)
mov     eax, [ecx + 4*edx] ; EAX <- i번째 array의 요소

```

## 5.2.2. USE32와 USE16 세그먼트 (USE32 and USE16 Segments)

비록 범용 레지스터가 32비트이더라도 프로그램이 32비트 전부를 액세스해야 하는 것은 아니다. 현재 코드 세그먼트의 디스크립터에 포함된 B비트는 오퍼랜드와 명령어의 주소 값이 디폴트로 32비트인지 16비트인지를 결정한다. USE32 세그먼트라 불리는 세그먼트에서의 명령은 16:32 어드레스(16비트 셀렉터와 32비트 오프셋)를 사용하며, 어드레스의 연산 결과가 32비트의 폭을 가진다. 마찬가지로 USE16 세그먼트는 16:16 어드레스(16비트 셀렉터와 16비트 오프셋)를 사용하여 주소의 연산 결과가 16비트 값을 가진다. 다른 한 경우, 세그먼트 디스크립터에 있어서 베이스 주소의 크기가 32비트이면, 가상 어드레스는 명령에 대하여 32비트의 크기를 가진다. 80386이상 프로세서의 보호모드에서 32비트 주소지정과 16비트 주소지정의 차이는 마지막 가상 주소까지 형성하는데 베이스 주소에 얼마나 큰 오프셋 값이 더해져야 하는가에 달려있다.

코드 세그먼트의 B비트의 중요성을 자세히 나타내기 위해, 16진수로 표현될 수 있는 8B 07의 기계어에 대하여 생각해 보자. 비록 인텔 명령의 인코딩을 메모리에 맡긴다고 하더라도, 이 코드를 보는 것만으로도 무엇을 하는 것인지 단정적으로 얘기할 수 없다. USE16 세그먼트에서 어떤 명령이 다음과 같다고 할 때,

```

mov ax, [bx]          ; 16비트 어드레스, 16비트 오퍼랜드

```

USE32 세그먼트에서는 똑같은 인코딩임에도 불구하고 실제로는 다른 명령이 된다.

```

mov eax, [edi]       ; 32비트 어드레스, 32비트 오퍼랜드

```

## 5.2.3 어드레스와 오퍼랜드 크기의 재지정 접두어

### (Address and Operand Size Override Prefixes)

재지정 접두어는 프로그래머에게 한 명령어가 실행되는 동안에 오퍼랜드나 주소 크기를 재지정할 수 있도록 해준다. 그래서 오퍼랜드 재지정 접두어 66h는 이어지는 명령의 오퍼랜드가 반대의 크기를 가지도록 하며(역자주 : 16비트는 32비트로, 32비트는 16비트로), 주소크기 재지정 접두어 67h는 주소 크기가 반대의 크기를 가지도록 한다. USE16 세그먼트에서 8B 07은 앞서서도 설명했듯이, 재지정 접두어가 있고 없음에 따라 다음의 네가지가 된다.

```

8B 07      mov     ax, [bx]      ; 재지정 없음
66 8B 07   mov     eax, [bx]    ; 오퍼랜드 크기 재지정
67 8B 07   mov     ax, [edi]    ; 어드레스 크기 재지정
67 66 8B 07 mov     eax, [edi]  ; 둘다의 크기 재지정

```

USE32 세그먼트에서는 다음의 네가지 가능성이 있을 것이다.

```

8B 07      mov     eax, [edi]   ; 재지정 없음
66 8B 07   mov     ax, [edi]   ; 오퍼랜드 크기 재지정
67 8B 07   mov     eax, [bx]   ; 어드레스 크기 재지정
67 66 8B 07 mov     ax, [bx]   ; 둘다의 크기 재지정

```

이것은 과연 디폴트 주소나 오퍼랜드 크기를 바꾸기 위해서는 재지정 접두어를 꼭 삽입해야 한다는 것을 의미하는가? 운 좋게 그건 아니다. 편리하게도 어셈블러는 어디에 재지정 접두어를 넣어야 하는지를 알고 있기 때문이다. 그러기 위해서는 우선 USE16이나 USE32 지시어를 사용하여 코드 세그먼트의 B비트를 선언해야 한다.

```
_TEXT SEGMENT BYTE PUBLIC USE16 CODE
```

혹은

```
_TEXT SEGMENT BYTE PUBLIC USE32 CODE
```

### 16비트 코드에서의 32비트 오퍼랜드와 어드레스 (32-bit Operands and Addresses in 16-bit Code)

16비트 프로그램에서 32비트 레지스터를 사용하는 것이 유용하며 이것이 가능하다는 것을 배웠을 때 매우 놀랐을지도 모르겠다. 예를 들어 프로그램이 32비트 프로세서에서 운영될 것이 확실한 경우, 몇몇의 C 컴파일러는 long 오퍼랜드의 정수에 대하여 32비트 레지스터를 사용한다. 또한 보호모드용 윈도우즈 프로그램에서 32비트 주소의 사용은 비트맵과 같은 \_\_huge 데이터에 대하여 높은 성능을 나타낸다.

리얼모드에서 32비트 주소를 사용하는 것(어떤 사람들은 이것은 big real mode라고 부르기도 함)은 무의미한 것처럼 보인다. 그러나 실제로는 그렇지 않다. 프로세서는 각 세그먼트의 디스크립터를 캐쉬한다. 리얼모드 운영중일 때 이러한 캐시는 일반적으로 64KB의 리미트와 세그먼트의 20비트 물리 패러그라프 주소와 같은 베이스 어드레스를 가리킨다. 만약, 보호모드로 전환하여, 몇 개의 세그먼트 레지스터에 선택터를 로드하고 나서, 세그먼트 레지스터의 복구 없이 리얼모드로 되돌아오면, 프로세서의 디스크립터 캐시는 물리 메모리의 어느 곳이나 가리킬 수 있는 베이스 어드레스를 가진 새로운 디스크립터를 계속 가지고 있게 된다. 이렇게 되면 1MB를 벗어난 영역을 액세스 할 수 있다. 즉, 보호모드 선택터가 가리키는 메모리는 어디든 액세스 할 수 있는 것이다. 그러나, 잠시 후 무작위로 인터럽트가 걸리므로(역자주 : 특히 타임 인터럽트), 프로세서는 베이스 주소를 다시 로드하게 되고 이러한 자유는 곧 끝나게 된다.

지금까지 앞의 예에서처럼, 확장 레지스터와 확장되지 않은 레지스터에 대한 오퍼랜드와 어드레스에 대하여 간단한 코드를 이용하여 알아보았다. 어셈블러는 실행하는데 알맞도록 필요한 접두어를 자동으로 집어넣는다.

여기서 한가지 중요한 주의점이 있다. PUSHF, PUSHA, MOVS 등의 몇 개 명령어는 명시적인 레지스터 오퍼랜드를 가지고 있다. 만약 이러한 명령의 32비트 버전에 대하여 올바르게 동작하는 어셈블러를 원한다면, 명령어의 뒤에 D 문자를 붙인다(PUSHFD, PUSHAD, MOVSD 등등). 비록 USE32 세그먼트로 코딩했어도 말이다. 난 전에, 아마 PUSHAD/POPAD로 의도했는데, PUSHA/POPA로 된 어처구니없는 실수를 한 상용 VxD를 본적이 있다. 이러한 종류의 풋내기 실수를 하지 말기를 바란다.

---

#### 설명 (Note)

마이크로소프트의 매크로 어셈블러(MASM assembler)는 .286, .386처럼 선언된 프로세서 타입을 기초로 하여 디폴트 세그먼트 B비트를 대입한다. 만약 .286으로 지정하면 80286 프로세서에서 제공하는 16비트 범용 레지스터와 세그먼트 레지스터, 명령어를 쓰도록 하게 해준다. 모든 세그먼트는 USE16으로 되는데 80286에서는 이것만 사용할 수 있기 때문이다. 또한 .386이라 지정하면 묵시적으로 디폴트 세그먼트 B비트는 USE32로 되는데, 그것은 아마 사용자가 원하는 것이길 때문이다. 이러한 디폴트 선택은 간편하지만, 애기치 않은 곤란한 일이 될 수도 있다. 만약 오퍼랜드로 확장된 레지스터를 사용하기 위해, 프로그램의 시작부분에 .386 지정자를 넣고, 라지 모델 윈도우즈 프로그램을 작성하기 위해 .model large, c로 지시어를 넣으면 결국 세그먼트는 USE32로 잘못된다. 이것을 피하기 위해 .386 지정자 전에 .model 지시어를 넣으면 된다.

---

### 5.2.4. 플랫 메모리 모델(The Flat Memory Model)

삼십이 비트 프로그램들은 보통 플랫 메모리 모델(flat memory model)을 사용한다. 플랫 메모리 모델에서는 CS 레지스터에서 베이스 주소는 0이고 리미트가 4GB인 코드 섹터를 가지며, DS, ES, SS는 베이스 주소가 0이고 리미트가 4GB인 다른 섹터를 가진다.(여기서 다르다고 한 것은 코드 섹터라고 보기 보다 데이터 섹터를 요구하기 때문이다. 필자는 앞에서 윈도우즈 95가 3순위 권한 프로그램에 대해 리미트가 4KB인 익스팬드다운(Expand-down) 데이터 세그먼트를 사용한다고 했던 것을 기억하고 있다. 절대 이 복잡하고 특별한 논의를 중복해서 하고 싶지 않다.) 따라서, 32비트 오프셋만으로도 4GB 가상 주소 공간의 어느 곳이나 위치할 수 있기 때문에 이것 자체로 충분한 정보를 제공할 수 있다. 32비트 플랫 모델 주소 계산에서 오프셋은 세그먼트 섹터의 역할이 최소화됨을 강조하기 위하여 가끔 0:32 포인터로 불리기도 하며, 간단히 플랫 포인터나 선형 포인터로 불리기도 한다. 따라서 플랫 모델 프로그램은 세그먼트 레지스터의 내용을 바꿀 필요가 없다. 그러나 십육비트 프로그램은 흠어진 메모리 조각을 액세스하기 위해 계속적으로 세그먼트 레지스터를 재로드 할 필요가 있다. 실제로 세그먼트 디스크립터의 로드와 보호모드와 관계된 절차들을 수행하는데 대한 비용은 대단히 높아서 플랫 포인터의 로드예 비해 최대 7개까지 된다. 따라서, 라지 모델의 16비트 프로그램은 플랫 모델 32비트 프로그램보다 대단히 느리다.

## 5.3 시스템 프로그래밍의 구성요소

### (Systems Programming Features)

80386 이상 프로세서의 몇 가지 특징은 운영체제 슈퍼바이저의 배타적 사용을 위한 것이다. 이러한 특징은 콘트롤 레지스터, 디버깅 레지스터, 글로벌 디스크립터, 로컬 디스크립터, 인터럽트 디스크립터 등의 시스템 테이블에 있다. 독자는 아마 리얼모드에서 프로세서의 인터럽트 처리법과 인터럽트가 걸리기 이전 루틴으로 되돌아가기 위해 IRET 명령에 사용법에 익숙할 것이다. 그러나 인터럽트 프로그래밍에 대해 리얼모드와 보호모드 사이에는 중대한 차이점이 약간 있다. 이제 이야기하려고 하는 것에 대하여 알 필요 없이 쉬러 같지도 모르겠지만, 그러나 이것을 알게 된다면 윈도우즈 95 가상 머신 관리자(Virtual Machine Manager)을 더 잘 이해하게 될 것이다.

#### 5.3.1 콘트롤 레지스터 (Control Registers)

프로세서는 CR0부터 CR3까지 4개의 콘트롤 레지스터를 가지고 있다. 콘트롤 레지스터는 0순위 권한 프로그램(ring-zero program)에 의해서만 액세스 가능하다. 그림 5-9와 같이 CR0(Control Register zero) 레지스터의 비트들은 프로세서 모드(리얼모드와 보호모드)운영, 페이징 기능의 사용여부, 코프로세서 명령을 잡아내서 에뮬레이션할 것인지 말 것인지 등등을 결정한다.

CR3(Control Register three)는 그림 5-4와 같이 페이지 디렉토리의 물리 주소를 가지고 있다. 이것은 가상 주소를 물리 주소로 변환할 때 사용하는 활성화된 페이지 테이블을 가지고 있다. CR2(Control Register two)는 페이지 폴트에 대한 선행주소를 가지고 있다. 페이지 폴트는 필요로 하는 가상 메모리의 페이지가 물리 메모리에 존재하지 않을 때 발생한다. 다른 콘트롤 레지스터의 수와 용도는 CPU 모델에 관계되며, 윈도우즈는 이들을 특별한 용도로 사용하지 않는다. 다양한 MOV 명령을 이용하여 콘트롤 레지스터를 액세스 할 수 있으며, 여기에 두 가지 예를 나타내었다.

```
mov cr0, eax    ; eax를 cr0로 복사
mov eax, cr2    ; cr2를 eax로 복사
```

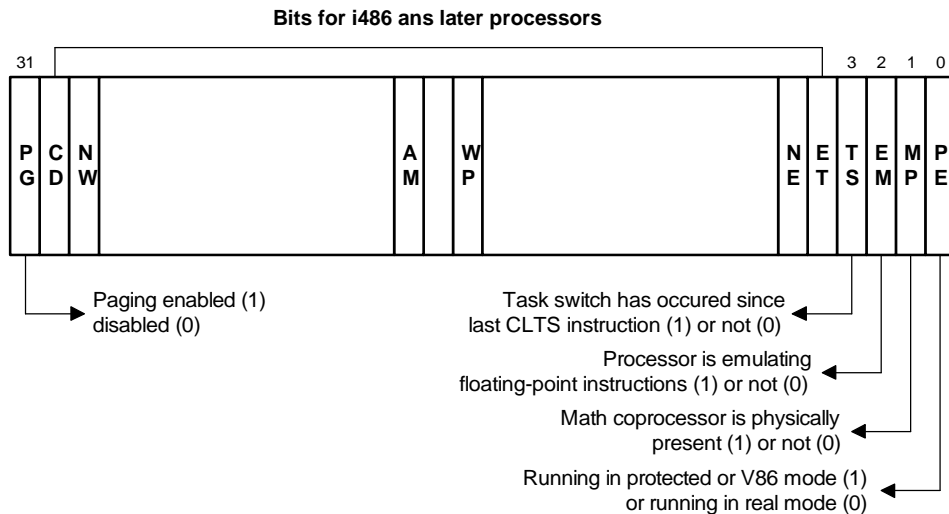


그림 5-9. 레지스터 CR0

만약, 마이크로소프트 매크로 어셈블러(MASM) 프로그램에서 이들 콘트롤 레지스터를 코딩하거나 다른 0순위 권한 명령을 사용하고 싶다면, 다음과 같이 타입 지정자의 뒤에 P문자를 붙여 특권 명령을 사용할 것이라는 것을 어셈블러에게 알려 주어야 한다.

```
.386P
mov     eax, cr0
```

만약 3순위 권한 프로그램이 이 콘트롤 레지스터 중의 하나를 읽거나 쓰려고 하면 일반 보호 폴트(**general protection fault**)가 뒤이어 일어난다. 3순위 권한 프로그램에게 CR0 레지스터의 읽기를 금지한 것은 약간 어리석은 것이다. 왜냐하면 특권 명령이 아닌 SMSW는 cr0의 하위 16비트를 저장하는 명령이기 때문이다(역자 주 : 앞에서도 논의되었지만 SMSW는 3순위 권한 프로그램도 실행할 수 있기 때문이다.)

비록 프로세서가 3순위 권한 프로그램에 CR0 레지스터의 변경을 막는다고 하더라도 윈도우는 이 변경에 관대하기 때문에 이 폴트를 잡은 다음 이를 에뮬레이트 해 준다. 예를 들어, 실제로 코프로세서를 가진 컴퓨터에서 V86 모드 프로그램이 코프로세서 에뮬레이션을 가능하게 하기 위한 방법으로 CR0의 EM 비트를 설정하는 것이다.

```
em_mask equ 4           ; 코프로세서 에뮬레이션 비트

mov     eax, cr0        ; 보통 특권 명령이지만, 실행됨
or      eax, em_mask    ; 코프로세서 에뮬레이션이 가능하게 설정
mov     cr0, eax        ; 보통 특권 명령이지만, 실행됨
```

윈도우즈 95에서 이러한 작업을 하는 이유는, 위의 두개 MOV 명령에 의해 발생하는 일반 보호 폴트를 VMM이 분석하기 때문이다. CR0에서 EAX로 MOV하는 것은 안전하다. 그래서 윈도우는 3순위 권한 프로그램에게 이를 허용한다. EM 비트를 켜는 것도 또한 안전하므로, 윈도우는 CR0로 MOV하는 것을 잘 에뮬레이트 해 준다.

(역자 주 : CR1 레지스터에 대해서는 어떠한 책에도 언급이 없다. 사실 있는지 없는지도 확실하지 않지만 말이다. 역자의 생각으로 CR1 레지스터는 인텔이 특수한 용도로 사용하거나 아니면 예약해 둔 것이 아닌가 한다.)

### 5.3.2 디버깅 레지스터(Debugging Registers)

또한 프로세서는 8개의 디버깅 레지스터를 가지고 있다. DR0 ~ DR3의 4개 레지스터는 브레이크 포인트 주소를 가지고 있으며, 나머지 레지스터는 브레이크 포인트 위치에서 프로그램이 메모리를 액세스할 때 프로세서가 디버깅 예외를 발생시키는 방법을 제어한다. 이 레지스터들은 지정된 4개의 어드레스에서 로드, 저장, 실행 등에 대하여 정지할 수 있도록 해준다. 펜티엄은 I/O 동작에서도 이런 방법을 사용할 수 있는데, 이것은 사이비 포트에 대한 OUT 명령에 대하여 유용하다. 시스템 소프트웨어는 MOV 명령으로 디버그 레지스터를 액세스 할 수 있으며 두 가지 예를 여기에 나타내었다.

```
mov     eax, dr0        ; DR0를 EAX로 복사
mov     dr7, eax        ; EAX를 DR7로 복사
```

윈도우는 디버거에서 DPMI 호출을 통해 이런 레지스터를 사용하도록 하고 있다. WDEB386과 Soft-Ice/W 같은 커널 디버거는 나나 독자 같은 시스템 프로그래머를 도와주기 위해 디버깅 레지스터를 사용한다. (역자 주 : 사실 일반적으로 사용하는 디버거는 브레이크 포인트를 거의 무한대로? 설정할 수 있다. 위에서 설명한 것처럼 프로세서는 4개의 브레이크만을 지원한다면 이를 어떻게 구현하겠는가? 독자들은 혹시라도 이것에 대하여 궁금하게 생각해 본적이 있는가? 이 책을 공들여 읽어 본다면 해답을 얻을 수 있을 것이다.)

### 5.3.3. 글로벌 디스크립터 테이블 (The Global Descriptor Table)

GDT 레지스터는 앞에서 필자가 언급했던 두 개의 디스크립터 테이블 중의 하나인 GDT(global descriptor table)의 가상 위치와 길이를 가지고 있다. GDT는 세그먼트 디스크립터를 포함하고 있는데, 세그먼트 디스크립터는 그림 5-3과 같이 시작주소, 길이, 타입, 액세스 권한 등을 나타내고 있다. 윈도우는 초기의 부팅 처리를 하는 동안 GDT를 생성하며, 여기에는 시스템 세그먼트인 0순위 권한 셀렉터가 위치한다. 특별한 경우 예외로써 셀렉터 40h(물리 주소에 있어 BIOS 데이터 영역은 0400h)를 제외하고, GDT 셀렉터는 보통 DPL이 0인 값을 가진다. 따라

서 응용 프로그램은 보통 GDT 디스크립터에 있는 세그먼트를 액세스 할 수 없다. 그러나, 어떤 특권 수준에 있는 소프트웨어든지 이 레지스터의 내용을 알아내기 위해 SGDT 명령을 사용할 수 있다.

```

gdtlimit  dw  ?           ; GDT의 길이 - 1
gdtaddr   dd  ?           ; GDT의 선형 베이스 주소
...
sgdt      fword ptr gdtlimit ; 어떤 순위 권한도 가능
lgdt      fword ptr gdtlimit ; 0순위 권한만 가능

```

SGDT와 LGDT는 오퍼랜드로 6바이트 메모리를 사용함을 기억하기 바란다. 이 코드에서 SGDT는 *gdtlimit*에 GDT의 리미트가 저장되고 그 다음에 이어지는 *gdtaddr*에 GDT의 가상 주소가 각각 저장된다. LGDT는 이 두 변수의 6바이트가 로드 된다.

### 5.3.4. 인터럽트 디스크립터 테이블 (The Interrupt Descriptor Table)

IDT 레지스터는 IDT(interrupt descriptor table)의 길이 및 가상 위치를 가지고 있다. 그림 5-10과 같이, IDT는 프로세서 인터럽트에 대한 핸들러를 나타내는 게이트(gate)를 가지고 있다. 윈도우즈의 IDT는 인터럽트 0 ~ 5Fh의 엔트리만을 가지고 있으며, 인터럽트 60h ~ FFh는 소프트웨어가 INT 명령을 실행해서 발생할 수 있는데 실제로 이것은 IDT 세그먼트의 리미트 때문에 일반 보호 폴트를 발생시킨다. IDT 레지스터의 액세스는 GDT 레지스터의 액세스와 비슷해서, 0순위 권한 코드나 리얼모드 코드만이 LIDT 명령을 통해 이 레지스터를 변경시킬 수 있다. 그러나 어떤 순위 권한 코드라도 IDT 레지스터의 내용을 알기 위해 SIDT명령을 사용할 수 있다.

```

idtlimit  dw  ?           ; IDT의 길이 - 1
idtaddr   dw  ?           ; IDT의 선형 베이스 주소
...
sidt      fword ptr idtlimit ; 어떤 순위 권한도 가능
lidt      fword ptr idtlimit ; 0순위 권한만 가능

```

SIDT와 LIDT 명령은 SGDT나 LGDT와 같이 6바이트 오퍼랜드를 사용한다.

#### 보호 전복시키기 (Subverting Protection)

어설프게 아는 것은 도리어 위험이 되는 법이다. 어떤 독자들은 기계를 접수하기 위해 특권 명령이 아닌 SGDT와 SIDT를 어떻게 사용하는가에 대한 방침을 따라 실험을 할 계획을 세우고 있을 것이다. 필자는 그것들에 대한 명백한 결과를 설명해서 이 실험에 대한 도전을 집어치우기를 바란다. SGDT와 SIDT는 정말로 GDT와 IDT 시스템 테이블의 선형 주소와 길이를 알려주며, 맘먹은 대로 테이블을 쉽게 변경할 수 있다. 이것은 매우 쉽지만 재미있기까지는 않다. (그러나 SIDT를 통해 얻은 IDT의 주소는 윈도우즈가 아닌 디버거의 것일지 모르니 주의하라!) 만약 원한다면, 우리 프로그램을 0순위 권한으로 만들어 주어서 황홀하게 해줄 게이트 디스크립터를 조작할 수 있지만, 우리는 올바른 인터럽트 처리를 하지 못할 것이기 때문에 깨지고 불타버릴 것이다. 음~ 이건 당신 컴퓨터이다. 윈도우즈는 이 중대한 테이블을 가지고 있는 페이지를 보호할 수 있다. 그러나 윈도우즈는 이 페이지를 보호하려 하지 않는다. 결국 당신은 개인용 컴퓨터를 다루고 있다는 것이다. 만약 차를 샀다면, 기름을 다 빼버리고 엔진이 어떻게 동작하는가를 보는 것은 자유다. 그러나 나중에 이 호기심에 대한 댓가를 지불하게 될 것이다. 윈도우즈에서도 같은 이치다. 일을 하기 위해 프로그램을 작성하는 곳에 이 역엔지니어링(reverse engineering)을 사용한다면 이를 허락할 것이지만, 기계를 관할하는 이 유혈 항목에 대해서는 그것을 어떻게 하는지 벌써 알고 있는 시스템 소프트웨어에게 말기는 편이 낫다.

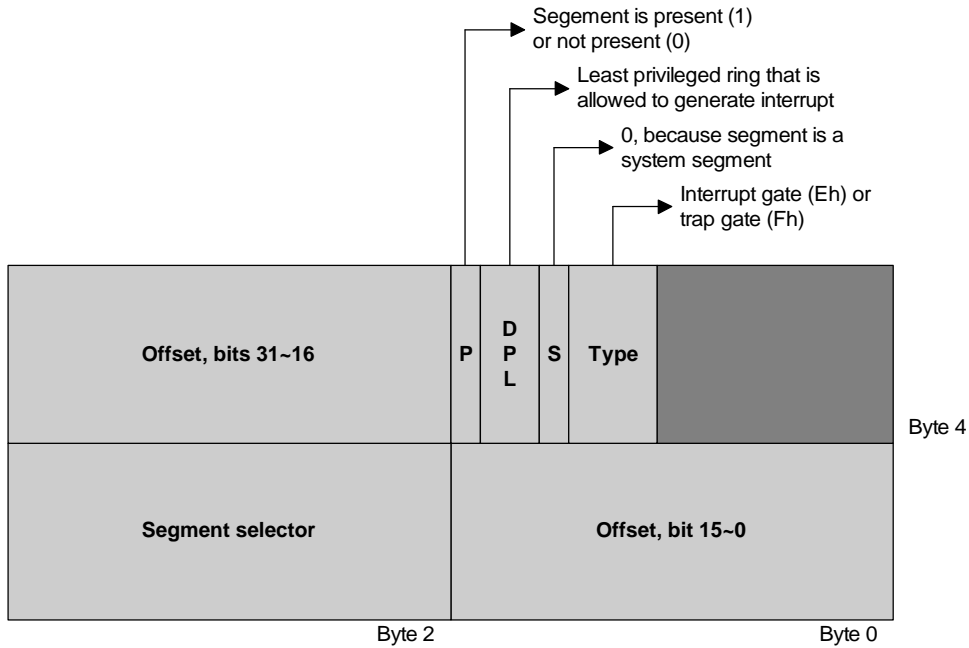


그림 5-10. 인터럽트 디스크립터 테이블에 있는 게이트 엔트리

### 5.3.5 로컬 디스크립터 테이블 (The Local Descriptor Table)

LDT 레지스터는 LDT(local descriptor table)의 셀렉터를 가지고 있다. 윈도우즈 95는 각 가상 머신마다 분리된 LDT를 생성한다. LDT 레지스터를 로드하기 위한 LLDT 명령은 특권화 되어 있으나, 저장하기 위한 SLDT 명령은 그렇지 않다. LDT 세그먼트 디스크립터의 타입 필드는 LDT가 시스템 세그먼트라는 것을 가리키고 있다. LDT의 셀렉터를 세그먼트 레지스터로 로드 할 수 없는데, 다른 말로 하면 (특권을 가지고 있다면) LDT의 셀렉터는 LDT 레지스터에만 로드 할 수 있다는 것이다. (메모리에 있는 디스크립터 테이블의 어드레스와 길이는 LDT의 세그먼트 디스크립터에 있다. 자신이 이 테이블을 액세스하기 위해서는 표준 사용자 데이터 세그먼트와 같은 다른 셀렉터가 필요할 것이다.) LDT는 GDT에 있는 디스크립터와 같은 형식의 디스크립터를 가지고 있다. 앞에서 언급한 바와 같이, 셀렉터의 비트2(3번째 비트)는 그것이 LDT를 선택할 것인지(비트2 = 1), GDT를 선택할 것인지(비트2 = 0)을 결정한다.

### 5.3.6 인터럽트 (Interrupts)

3종류의 이벤트는 응용 프로그램의 보통의 흐름을 가로채서 운영체제에 제어권을 넘긴다. 하드웨어 인터럽트(hardware interrupt), 소프트웨어 인터럽트(software interrupt), 프로세서 예외(processor exception)가 그것이다.

- 하드웨어 인터럽트는 컴퓨터의 PIC(Programmable Interrupt Controller)를 통해서 인터럽트 요청(interrupt request; IRQ)을 나타내는 I/O 장치에서 발생한다.
- 소프트웨어 인터럽트는 프로그램이 시스템 소프트웨어와 통신하기 위해 INT n 명령을 실행함으로써 발생한다.
- 프로세서 예외는 프로세서가 문제를 인식하거나, 다른 예외 조건(페이지 폴트나 일반 보호 폴트)의 발생으로 운영체제의 중재를 요청하는 경우 발생한다.

이러한 3종류의 인터럽트는 항상 일어난다. 여기에 한가지 더 NMI(non-maskable interrupt)가 있기는 하지만 정상적으로 운영되고 있는 시스템에서는 일반적으로 발생하지 않는다.

모든 인터럽트는 그 원인에 관계없이, 현재 실행되는 프로그램의 컨텍스트를 스택에 저장한 다음, 제어권을 IDT 게이트 엔트리의 핸들러로 전환한다. 컨텍스트를 저장한다는 정확한 방법은 인터럽트에 의해 정지된 프로그램과 핸들러간의 특권 수준의 관계에 따라 다르다.

- 같은 특권 수준으로부터의 인터럽트(Interrupts from the Same Privilege Level) : 많은 경우에, 핸들러는 인터럽트된 프로그램과 같은 특권 수준에 있다. 이런 경우, 프로세서는 현재 스택에 플래그와 명령 카운터(current instruction counter)를 저장한 다음 제어권을 핸들러로 전환한다. 핸들러는 나중에 인터럽트된 지점으로 되돌아가기 위해 IRET나 IRETD 명령을 실행한다. (IRET는 16비트 인터럽트 서비스 루틴을 위한 것이며, IRETD는 32 비트 인터럽트 서비스 루틴을 위한 것이다.) 이러한 절차는 리얼모드 인터럽트가 IDT의 게이트 엔트리를 통하는 대신 물리 메모리의 0:0에 있는 인터럽트 벡터를 통해 전달된다는 것을 제외하면 리얼모드에서 행해지는 것과 매우 유사하다.
- 낮은 특권 수준으로부터의 인터럽트 (Interrupts from an Outer(Lesser) Privilege Level) : IDT게이트에 지정된 핸들러가 0순위 권한이고, 인터럽트는 3순위 권한으로부터 올 때 좀더 흥미로운 일이 일어난다. 우선, 프로세서는 TSS(task state segment)라 불리는 특별한 시스템 세그먼트를 검사한다. (인텔은 모든 다른 쓰레드는 그 자체의 TSS를 가지게 할 의도였으며, TSS는 멀티태스킹에서 기본적 구조체로 될 수 있었다. 그러나 TSS를 사용한 태스크 전환의 대가는 너무 비싸기 때문에, 윈도우즈는 이러한 방법을 사용하지 않는다. 대신 윈도우즈는 좀더 폭력적인 명령을 사용한다.) 프로세서는 권한이 교차되는 인터럽트 핸들링을 하기 위해 TSS를 살펴볼 필요가 있다. 왜냐하면 TSS는 컨텍스트를 저장할 0순위 스택의 톱 포인터를 가지고 있기 때문이다. 이 포인터는 0순위 권한으로 전환하기 위한 인터럽트가 발생하는 때에는 항상 같은 값을 가지는 스택 톱 포인터이다. 0순위 권한 스택으로 전환한 후, 그림 5-11과 같이 3순위 권한의 스택 포인트(SS:ESP), EFLAGS 레지스터와 CS:EIP 명령포인터를 포함하고 있는 표준 인터럽트 프레임(standard interrupt frame)을 푸시한다. 나중에 슈퍼바이저가 IRETD 명령을 실행하면, 프로세서는 표준 인터럽트 프레임을 꺼내고 난 다음, 새로운 코드 섹터의 RPL이 되돌아갈 곳의 낮은 특권 수준을 요구하게 되는 것을 관찰한다.

	SS	10
ESP		C
EFLAGS		8
	CS	4
EIP		0

그림 5-11. 낮은 수준으로부터 발생한 인터럽트에 대한 스택 프레임

- V86 모드로부터의 인터럽트 (Interrupts from V86 Mode) : 인터럽트가 V86 모드에서 발생하면 더 복잡한 처리가 일어난다. 인터럽트가 발생한 시점에, 세그먼트 레지스터는 보호모드에서는 무의미한 리얼모드의 패러그래프 숫자를 가지고 있다. 0순위 권한 스택으로 전환한 후, 프로세서는 GS, FS, DS, ES를 저장하고 그 값들을 0으로 설정한다. 세그먼트 레지스터를 0으로 만드는 것은, 인터럽트 핸들러가 어떤 데이터를 액세스 시도 전에 유효한 섹터의 로드 하려고 하는 작용에서, 회복 불가능한 일반 보호 폴트의 발생 가능성에 대하여 미리 손을 쓰는 것이다. 그리고 나서 앞에서 설명한 것과 같이 프로세서는 SS:ESP, EFLAGS, CS:IP를 푸시 한다. 그림 5-12는 이 스택 프레임을 도해한 것이다. 언젠가는 IRETD 명령이 실행될 것이며, IRETD는 모든 것의 최후에 세그먼트 레지스터를 복구한다. 나중에 VxD의 프로그래밍에 사용되는 클라이언트 레지스터 구조체(client register structure)를 논의할 때, 그림 5-12를 살펴볼 것임을 상기시켜 주겠다. 왜냐하면 두개의 구조가 아주 비슷하기 때문이다.

	<b>GS</b>	<b>20</b>
	<b>FS</b>	<b>1C</b>
	<b>DS</b>	<b>18</b>
	<b>ES</b>	<b>14</b>
	<b>SS</b>	<b>10</b>
<b>ESP</b>		<b>C</b>
<b>EFLAGS</b>		<b>8</b>
	<b>CS</b>	<b>4</b>
<b>EIP</b>		<b>0</b>

그림 5-12. V86모드로부터 발생한 인터럽트에 대한 스택 프레임

#### 인터럽트 핸들링 (Handling Interrupts)

윈도우즈 95에는 인터럽트 핸들링에 알아야 할 것이 4가지 항목이 있다. 이런 항목을 알아볼 때, 시스템 프로 그래머가 윈도우즈 95의 내부와 이에 관계된 어셈블리어에 대하여 알 필요가 있는 것에 대하여도 논의할 것이다.

첫째 항목은 **하드웨어 인터럽트를 명료하게 하는 것(disambiguating hardware interrupt)**이다. 리얼모드 BIOS는 IRQ(interrupt request) 0h ~ 7h를 인터럽트 8h ~ Fh, IRQ 8h~ Fh를 인터럽트 70h ~ 77h가 되도록 PIC를 프로그 램한다. 그러나 IRQ 5h는 하드웨어에 의해 발생하는 INT 0Dh, 프로세서에서 발생하는 일반 보호 폴트, 실없는 사 용자 프로그램에 의해서 실행되는 INT 0Dh 명령 등에 의해서 발생된다는 것을 걱정해야 한다. 제일 마지막 경우 인 소프트웨어 INT 0Dh는 탈선 행위이며, 아마 혼동이 발생하는 것은 마땅할 것이다. 그러나 처음의 두 경우는 일 반적으로 작업중인 시스템에 발생할 수 있는 경우이다. 그래서 인터럽트 핸들러는 PIC의 입력 서비스 레지스터 (in-service register)를 검사해서 왜 인터럽트가 입력되었는지를 결정해야 한다.

윈도우즈는 하드웨어 인터럽트를 일반적인 영역으로부터 옮김으로서 이 인터럽트를 명료하게 할 필요성을 없 애고 있다. 윈도우즈 하에서는 IRQ 0h ~ Fh를 인터럽트 50h ~ 5Fh에 존재한다. 그래서 프로세서 예외와 하드웨어 인터럽트의 혼동 가능성은 없다. (리얼모드 드라이버와 TSR이 갑자기 새로운 인터럽트 핸들러를 흑크해야 할 것이 라는 걱정은 없다. VPICD(Virtual PIC Device)는 **가상 머신**에 대하여 기존의 장소에 있는 곳에 하드웨어 인터럽트 를 전달하기 때문이다)

비록 PIC가 다시 프로그램 된다고 하더라도, IRQ 0Ch(보통 버스 마우스 인터럽트)와 NetBIOS 인터럽트 사이 의 혼동이 내재해 있다. 왜냐하면 둘 다 INT 5Ch를 사용하기 때문이다. 인터럽트 핸들링에 대한 두 번째 항목이 바로 윈도우즈가 이런 혼란스런 인터럽트를 어떻게 처리하는가 이다. 실제적으로 IDT 게이트에는 두개의 다른 특 권 수준이 결합되어 있다. 지금까지는 핸들러 자체의 특권 수준에 대해서만 논의해 왔으며, 이 특권수준은 프로세 서의 인터럽트 처리에 있어서 권한 변경의 필요성을 결정한다. 하지만 또한, 게이트 디스크립터에는 DPL(Descriptor Privilege Level) 필드가 있으며, 이것은 INT n 명령의 발생을 허용할 것인지 그렇지 않을 것인지를 결정한다. 인터럽트 5Ch에 대한 게이트 엔트리는 DPL이 0이다. 만약 3순위 권한 프로그램이 INT 5Ch를 발생 시키면, 프로세서는 특권 수준이 맞지 않기 때문에 GP 폴트를 발생시킨다. 그러면 GP 폴트 핸들러는 NetBIOS의 진로를 제어할 수 있다.

그 다음으로 알아야 할 세 번째 항목은, 프로세서의 어떤 예외(전부는 아님)는 에러코드를 스택에 푸시 한다. 이것은 다른 모든 작업을 수행한 후에 행해진다. 이 코드는 윈도우즈에서는 아무쓸모 없다. 왜냐하면 에러코드를 통해 운반된 정보의 양이 매우 작기 때문이다. 또 다른 이유는 예외가 가끔 에러코드를 푸시하기 때문에 여기에 맞는 정확한 1차 핸들러를 만들어야 한다는 것이다. 예를 들어, 에러코드를 푸시하지 않는 인터럽트와 예외 1차 핸 들러는 모든 핸들러에서 사용하는 균일한 형식의 스택 레이아웃으로 처리할 수 있도록 하기 위해 더미 값을 푸시

해야하기 때문이다. 그래서 시스템은 인터럽트 된 프로그램으로 되돌아가기 전 레지스터 복구 부분처럼 에러코드를 지나쳐 버린다.

마지막으로, 인터럽트 게이트(interrupt gate)는 보통 친절하기 때문에, 인터럽트 핸들러가 제어권을 얻을 때 인터럽트가 디스에이블 된 상태로 된다. 하지만 인터럽트가 디스에이블로 만드는 인터럽트 게이트와는 달리 인에이블 된 채로 남겨두는 트랩 게이트(trap gate)가 있다. 윈도우즈는 사용자 특권 코드에 직접 전달되는 INT 21h 같은 소프트웨어 인터럽트에 트랩 게이트를 사용한다. 다른 핸들러들에게도 인터럽트를 허용해 주기 위해 STI 명령을 사용하면 값비싼 대가인 GP 폴트를 지불하게 될 것이다. 나아가 게이트는 16비트와 32비트 종류로 나누어진다. 게이트의 B비트는 핸들러가 USE16 세그먼트에서 동작하던지 USE32 세그먼트에 동작하던지 관계없이 스택에 저장될 컨텍스트의 폭을 제어한다. 윈도우즈에서 하드웨어 인터럽트는 항상 32비트 게이트를 통해서 전달된다. 왜냐하면 16비트 코드가 실행되고 있는지 32비트 코드가 실행되고 있는지 알 수 없기 때문이다. 그러나 시스템 VM에서 많은 소프트웨어 인터럽트는 16비트 게이트를 통해 전달된다. 그리고 이러한 사실로 인해 Win32 프로그램에 의해 사용되는데 방해가 된다.