

8장. 가상 디바이스 드라이버의 초기화와 종료 (Initializing and Terminating Virtual Device Drivers)

윈도우즈 95의 가상 디바이스 드라이버는 언제 로드 되는가에 따라 이를 스테틱 드라이버와 다이내믹 드라이버로 분류된다. VMM은 스테틱 VxD를 윈도우즈 95가 시작할 때 로드하고, 윈도우즈 95 세션동안에는 가상 메모리에 있도록 유지 관리한다. VMM은 다이내믹 VxD를 응용 프로그램이나 다른 VxD가 요청할 때 동적으로(좀 다른 말었나?) 로드 한다. 다이내믹 VxD는 윈도우즈 95 세션동안 자주 언로드 하거나 재로드 할 수 없다. 일반적으로 볼 때, 특정한 하드웨어나 응용 프로그램과 관계된 VxD는 동적(다이내믹)이며, 반면 가상 머신 관리자의 핵심 요소들(VMM 그 자신도 포함)은 정적(스테틱)이다. VxD의 생명주기(life cycle)은 다음에 의해 결정된다. VMM은 스테틱 VxD를 로드하기 위해 레지스트리 데이터베이스나 SYSTEM.INI를 뒤진다. TSR 유틸리티나 MS-DOS 디바이스 드라이버 같은 리얼모드 상주 프로그램들은 특정 스테틱 VxD를 로드 하도록 VMM에게 요청하기 위해 인터럽트를 호크할 수 있다. 스테틱 VxD는 시스템 시작 신호로 세 개의 초기화 메시지를 받는데, 이것은 Sys_Critical_Init, Device_Init, Init_Complete이다. 이 메시지들은 스테틱 VxD가 시작하는 동안 VMM이 도달한 중요한 특정 상태를 나타내는 것이다. 스테틱 드라이버들은 VMM이 보호모드로 전환하기 전인 리얼모드에서 실행하는 초기화 코드를 포함하고 있다. 마지막에는, 스테틱 드라이버들은 VMM이 윈도우즈를 빠져나가서 리얼모드로 되돌아갈 준비를 하고 있다는 종료 메시지를 연속으로 받는다.

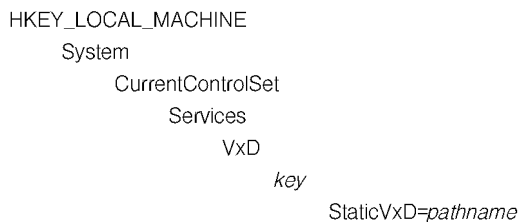
VMM은 그 자신이 다이내믹 드라이버를 찾지는 않는다. 구성 관리자(Configuration Manager)나 입출력 관리자(Input/Output Supervisor; IOS)와 같은 VxD나 응용 프로그램은 필요한 VxD들을 결정하고 이들을 로드하기 위해 VXDldr(이것은 자체적으로 스테틱 VxD이다)을 호출한다. 스테틱 VxD와는 대조적으로, 다이내믹 VxD는 단 한 개의 초기화 메시지와 한 개의 종료 메시지를 받는다. 스테틱 VxD와 다이내믹 VxD는 이러한 차이점들이 있음에도 불구하고, 둘 다 같은 틀과 같은 API를 사용한다. 사실, 스테틱 VxD와 다이내믹 VxD의 메커니즘 차이점이라고는, 다이내믹 VxD는 자신의 모듈정의 파일에 있는 VxD 문장에 자신이 다이내믹 VxD라는 것을 알려주고 있으며, 반면 스테틱 VxD는 그렇지 않다는 것이다.

8.1 스테틱 VxD (Static VxDs)

VMM에게 스테틱 VxD를 로드하게 하는 방법에는 세 가지가 있다. 대부분의 경우에는 레지스트리 데이터베이스나 윈도우즈 95 디렉토리에 있는 SYSTEM.INI 파일을 수정하는 것이다. (설치 프로그램이 할 수도 있다) 그러나 어떤 경우 VxD 없이는 동작할 수 없는 TSR 유틸리티나 다른 상주 프로그램이 있을 수 있는데, 이 경우 TSR 유틸리티는 소프트웨어 인터럽트 2Fh를 호크하고 시작 알림 메시지인 160Fh를 찾는다.

8.1.1 시스템 레지스트리 사용하는 것 (Using the System Registry)

윈도우즈 95에서 스테틱 VxD를 로드 하는데 좋은 방법은, 로컬 머신 경로에 있는 시스템 레지스트리 데이터베이스에 로딩 항목을 넣은 것이다.



이 다이어그램에서, 레지스트리 체계의 일부 중 *key*는 어떤 레지스트리 키이며, *pathname*은 정적으로 로드할 가상 디바이스 드라이버의 이름이다. VMM은 VxD라는 이름의 키에서 *StaticVxD*라는 이름의 값을 가진 서브키를 모두 나열하고, 이들을 각각 정적으로 로드 한다.

예를 들어, 구성관리자(Configuration Manager)는 그림 8-1에 나타낸 바와 같은 레지스트리 항목을 가지고 있다.

CONFIGMG 디바이스의 이름 앞에 붙은 *는 분리된 디스크 파일이 아니라 물리적으로 VMM32.VXD에 위치한 드라이버라는 것을 가리킨다. 물론 이것은 독자들이 만든 드라이버의 변환에는 사용할 수 없다. *Detect*와 같은 다양한 이름의 값들은 레지스트리가 디바이스에 지정된 인자들을 어떻게 담고 있는지를 보여주고 있다. 필자는 이 장의 뒷부분에서 독자들이 만든 드라이버를 초기화하는 동안 이를 구성하기 위해 *Detect*와 같은 인자를 알아내는 방법을 설명할 것이다. (역자 주 : 윈도우즈 정상적으로 설치했다라도 시작메뉴의 어디에도 레지스트리 편집기를 볼 수 없을 것이다. 윈도우즈 95 디렉토리를 잘 살펴보면 REGEDIT.EXE가 있다. 이것이 바로 레지스트리 편집기이다. 역자의 경우에는 “시작메뉴→프로그램(P)→보조프로그램”에 “레지스트리 편집기”라는 이름으로 등록해 놓고 사용한다.)

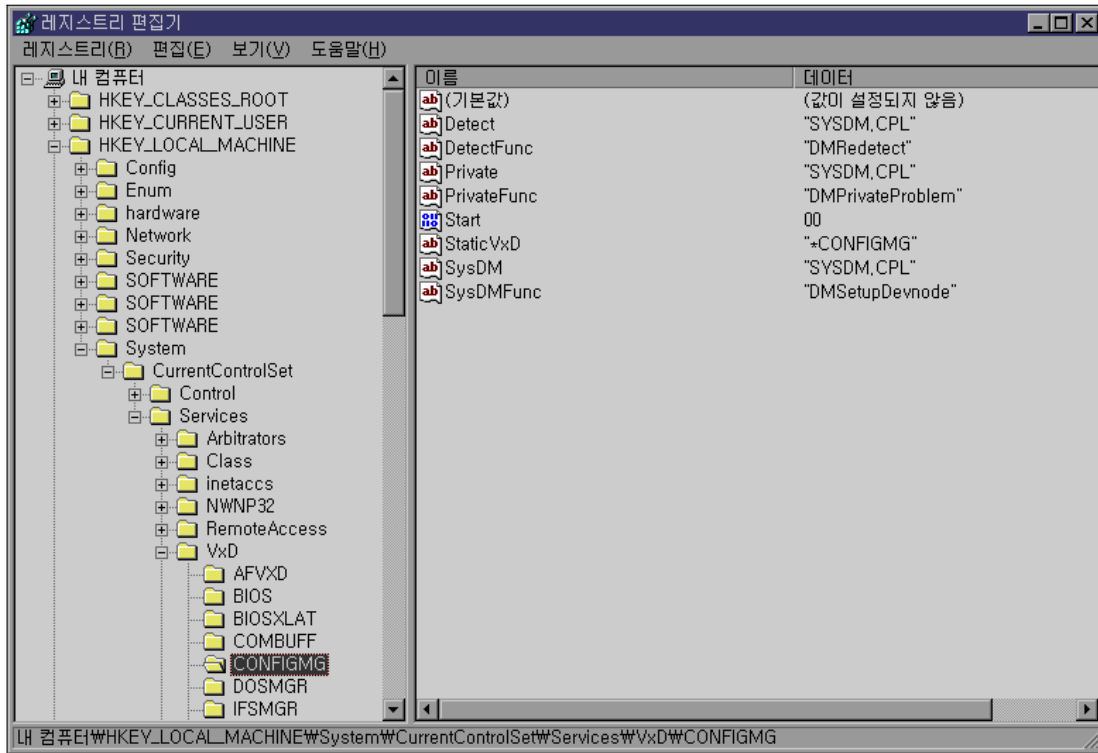


그림 8-1 CONFIGMG 디바이스의 레지스트리 항목들

8.1.2 SYSTEM.INI 파일을 이용하는 것 (Using the SYSTEM.INI File)

윈도우즈 3.1과 그 이전의 윈도우즈는 시스템 구성 파일인 SYSTEM.INI에 [386enh] 섹션이 있었는데, 이것은 많은 윈도우즈의 저수준 기능을 제어했었다. 또한 다음과 같은 문장을 써넣음으로 해서, 직접 윈도우즈가 가상 디바이스 드라이버를 로드 하도록 할 수 있었다.

```
device=pathname
```

예를 들어, 만약 독자들이 만든 VxD의 이름이 MYVXD.VXD이고 C 드라이브의 MYAPP 디렉토리에 설치되었다고 하면, SYSTEM.INI 파일은 다음과 같은 문장을 넣어야 할 것이다.

```
device=c:/MYAPP/MYVXD.VXD
```

물론 윈도우즈 95에서도 SYSTEM.INI 파일을 사용할 수 있지만, 앞서서도 설명한 것처럼 레지스트리를 수정하는 것이 더 바람직하다. 그럼에도 불구하고, 필자의 경우에는 스테틱 VxD를 개발하면서 SYSTEM.INI 파일을 수정하는 것이 더 쉬웠다. 깨진 VxD는 윈도우즈 95의 시작을 못하게 하기도 하는데, 이 깨진 VxD를 레지스트리에서 제거해야 하지만 윈도우즈 95 세션이 시작하지 않으면 이 레지스트리도 수정할 수 없기 때문에, 이것이 SYSTEM.INI 파일을 사용하게 되는 실제 이유이다.

MS-DOS로의 부팅 (BOOTing into MS-DOS)

다른 곳에서도 말했듯이, 컴퓨터를 켜면 윈도우즈 95는 그래픽 윈도우즈 환경으로 부팅 한다. (역자 주 : 이 박스의 내용은 3장에서 설명한 내용과 거의 중복된다. 왜 이런 똑 같은 내용을 적었을까 하는 의문이 생긴다.) MS-DOS 프롬프트로 시작하기 위한 손쉬운 방법을 벌써 한가지는 알고 있을 것이다. 예를 들어, 부팅 처리 중 F8을 누르면, 그래픽 윈도우즈 환경 대신 명령 프롬프트를 선택할 수 있는 메뉴가 나타난다. 명령 프롬프트에서는 리얼모드 네트워크 구성, DDK의 디버거나 작은 바이너리 설치, Soft-Ice/W 같은 디버거 시작과 같은 것을 포함하여 MS-DOS에서 할 수 있는 유용한 것이 있다. 또한 윈도우즈 95로 들어가기 위해 WIN 명령을 입력할 수도 있다.

필자의 경우에는 작업의 성격상 윈도우즈 95를 시작하기 전에 거의 항상 MS-DOS로 들어가게 된다는 것을 알게 되었으며, 그래서 부팅 처리에 있어 MS-DOS로 들어가게 하는 추가적인 방법을 두 가지 배웠다. 한가지 방법은 MS-DOS에서 AUTOEXEC.BAT를 처리한 다음 간단히 WIN 명령을 발생시키기 때문에 이로써 윈도우즈 95가 제어권을 얻는다는데 착안한 것이다. 만약 WIN.COM 파일을 가지고 있는 윈도우즈 디렉토리보다 패스 상에 앞에 있는 디렉토리에 WIN.BAT를 넣어 둔다면 윈도우즈 95 대신 만든 .BAT 파일이 실행하게 될 것이다.

MS-DOS 프롬프트로 부팅하기 위한 좀더 우아한 방법은 MSDOS.SYS를 수정하는 것이다. 이것은 신성 불가침 아닌가!! MSDOS.SYS는 MS-DOS를 가지고 있는 두 개의 파일중 하나(다른 하나는 IO.SYS)가 아닌가? 이전 버전의 MS-DOS에서, MSDOS.SYS는 프로그램 코드를 가지고 있는 커다란 바이너리 파일이었기 때문에 수정은 꿈도 꾸지 못했다. (이 책을 읽는 한두 명의 독자라면 가능할지 모르겠지만, 우리 대부분은 그렇지 못하다) 그러나 윈도우즈 95에 포함된 MS-DOS에서는 부팅 디스크의 루트 디렉토리에 있는 짧은 아스키 텍스트 파일이다. 여기에는 윈도우즈의 표준 .INI 파일과 비슷한 형식으로 된 윈도우즈 95의 시작 옵션을 가지고 있다. BootGUI 옵션은 MS-DOS가 묵시적으로 WIN 명령을 자동으로 발생시킬 것인지 아닌지를 제어한다. 그래서 다음에서 읽을 수 있는 것과 같이 수정할 수 있다.

```
[Options]
BootGUI=0
```

MSDOS.SYS 파일을 편집하기 위해서는 *Hidden, Read-Only, System* 속성을 제거해야 한다. 편집 후에는 이 속성을 복원할 수도 있고 그렇지 않을 수도 있는데, 이것은 MS-DOS 뿐만 아니라 윈도우즈 95도 이들 속성이 설정되었는지 그렇지 않은지 주목하지 않기 때문이다.

그런데 만약 이전 버전의 MS-DOS로 부팅하기 위한 멀티부팅 체제(MSDOS.SYS 파일의 옵션 섹션의 BootMulti=1로 제어된다)를 사용한다면 MSDOS.SYS라는 이름이라고 모두 편집하려고 하지 말라. 이 경우 MSDOS.SYS 파일은 진짜 MS-DOS를 가지고 있다. 윈도우즈 95는 BootMulti 체제를 사용하는 경우 파일 이름을 바꾸게 되는데, 따라서 정작 바꾸려고 하는 파일은 MSDOS.W40이라는 이름으로 되는 것이다.

8.1.3 시작 알림 INT2Fh 함수 1605를 사용하는 것

(Using the INT 2Fh, Function 1605h, Startup Broadcast)

윈도우즈 95에서 리얼모드 소프트웨어는 그 중요성이 많이 줄어들었으며, 시간이 지날수록 계속적으로 그렇게 되겠지만 아직은 여전히 많이 존재하고 있다. 호환성을 위해(아마 이 이유일 것이다), 윈도우즈 95는 리얼모드 드라이버와 램상주(TSR) 유틸리티가 함께 공존하기 위해 몇 가지 인터페이스를 계속적으로 지원한다. 시작알림을 하는 인터럽트 2Fh, 함수 1605h는 이제 필자가 설명하려고 하는 매우 중요한 인터페이스의 하나이다.

VMM은 시작할 때 AX 레지스터를 1605h로 설정하고 인터럽트 2Fh를 발생시킨다. 램상주 소프트웨어는 이 알림을 받아보기 위해 인터럽트 2Fh를 후크 한다. 우리가 만든 핸들러는 이 인터럽트의 다른 핸들러를 호출하고 시작 정보 구조체(startup information structure)의 링크드리스트를 가리키는 한 쌍의 ES:BX를 리턴 한다. 그 코드는 다음과 같을 것이다.

```
sis      db  3, 0          ; SIS_Version
         dd  0            ; SIS_Next_Dev_Ptr
         dd  vxdname      ; SIS_Virt_Dev_File_Ptr
```

```

                dd  refdata          ; SIS_Reference_Data
                dd  0                ; SIS_Instance_Data_Ptr
vxldname db 'pathname', 0
                assume ds:nothing, cs:@curseg

                align 4
org2f dd ?
int2f: cmp ax, 1605h
        je  @F
        jmp [org2f]
@@:    pushf
        call [org2f]
        mov word ptr sis+2, bx
        mov word ptr sis+4, es
        mov bx, cs
        mov es, bx
        mov bx, offset sis
        iret

```

이 코드조각에서 *pathname*은 가상 디바이스 드라이버의 이름을 나타내며 *refdata*는 이 드라이버의 리얼모드 초기화에 전달되는 32비트 데이터 항목을 나타낸다.

내가 만든 드라이버는 어디에 있는가? (Where's My Driver?)

만약 TSR에서 VxD를 제공하는 방법을 사용한다면, 사용자들에게 유연성을 제공하는 동시에 TSR과 **같은 디렉토리에 있는 VxD**와 TSR의 이름에 확장자만 다른 이름을 제공하여 코드를 간단하게 할 수 있으며, 제공하려는 VxD를 TSR과 항상 같은 디렉토리에 둘 수 있다. TSR은 다음의 코드와 같이 환경 블록의 끝까지 검사하여 자체의 완전한 경로 이름을 결정할 수 있다.

```

                mov es, es:[2Ch]    ; ES:야 -> 환경 블록
                xor di, di          ; ...
                cld                 ; 강제 전방 참조 (디렉션 플래그 지움)
                xor al, al          ; 비교하기 위하여 AL=0으로 설정
                xor cx, cx          ; CX=초기 카운트
                dec cx              ; ...
@@:    repnscasb                   ; 문자열의 끝을 찾는다
        scasb                      ; 다음의 바이트가 0인가?
        jne @B                      ; 만약 그렇지 않으면 다음의 환경 블록
        add di, 2                    ; ES:DI는 이제 우리 드라이버의 이름

```

필요한 VxD의 이름은 TSR 파일의 이름이 되며, 단지 파일 확장자만 바꾸기만 하면 된다. 예를 들어, .COM을 .VXD로 바꾸어야 한다.

대개의 프로그래머들이 자주 사용하는 기술로는, 리얼모드 TSR 코드를 .EXE 파일로 두고 VxD의 파일을 만들 때 이를 링크스텝으로 사용한다. (이것은 VxD의 .DEF 파일 STUB 문장에 이 TSR 실행 파일을 적어 넣는 것이다.) 이러한 방법은 한 개의 파일만 사용할 수 있도록 해준다. 물론 드라이버의 이름을 알아내기 위해 환경 블록을 검사해 하는 것은 여전히지만, 파일 확장자를 바꿀 필요는 없는 것이다. (역자 주 : 이러한 방법은 SMARTDRV.EXE등에서 사용한다.)

대개 *pathname*에는 드라이버의 완전한 이름을 사용한다. 예를 들어, 드라이버의 이름이 MYVXD.VXD이고 C 드라이브의 MYAPP 디렉토리에 위치해 있다면 다음과 같이 나타낼 것이다.

```
myvxdname db 'c:/MYAPP/MYVXD.VXD', 0
```

그러나 만약 드라이버가 윈도우즈 95 디렉토리나 시스템 디렉토리에 있는 경우와 드라이버가 있는 디렉토리가 현재 패스에 있다면, 단지 파일 이름과 확장자만 제공해도 된다.

간혹 램 상주 프로그램이 두 개 이상의 드라이버가 필요한 경우가 있을 수 있다. 이 경우 INT 2Fh 핸들러에서 리턴하기 전에 추가적인 시작 정보 구조체(startup information structure)를 생성하기만 하면 된다. (역자 주 : 물론 완전한 링크드리스트가 되도록 구조체의 값들도 세팅해야 할 것이다.)

8.4.1 로드 순서와 중복 (Load Order and Duplication)

VMM은 같은 드라이버를 두 번 로드 하면 안 된다. VMM은 드라이버를 만들 때 제공하는 `Declare_Virtual_Device` 매크로로 선언한 디스크립션 블록에 있는 식별자에 의해 드라이버를 인식한다. 그래서 드라이버가 다른 디스크에 있고, .DEF 파일에 다른 이름을 지정했고, `Declare_Virtual_Device` 매크로 호출에 다른 이름을 사용했다고 하더라도 중복된 드라이버로 간주한다. (역자 주 : 만약 식별자가 같은 드라이버인 경우) 여기에 한가지 예외가 있는데, 이것은 두 개의 드라이버가 완전히 똑같이 일치하더라도 0 식별자(`Undefined_Device_ID`)를 사용하면 절대 서로 같다고 간주하지 않는다. 정의되지 않은 식별자(undefined ID)를 사용하는 드라이버를 중복 로드 하는 것은 에러가 아니다.

같은 드라이버를 중복 로드 하는 것이 가능(사실 매우 쉽다)하기 때문에, VMM이 드라이버를 로드하려고 하는 순서를 알아둘 필요가 있다.

- ① VMM은 INT 2Fh 함수 1605h의 시작 정보 구조체(Startup Information Structure)를 통해 구한 드라이버를 우선 로드 한다. 순서는 이 구조체의 링크드리스트에 나타난 순서이다. 링크드리스트에 나타난 순서는 램상주 프로그램이 INT 2Fh를 후크한 순서의 역순이어서, 이 인터럽트를 최후로 후크한 프로그램이 거꾸로 최고의 우선권을 가진다.
- ② 그리고 난 뒤, VMM은 앞에서 설명한 것처럼 레지스트리 데이터베이스의 `StaticVxD`라고 하는 이름의 값을 찾는다.
- ③ 마지막으로, VMM은 SYSTEM.INI 파일의 `device` 문장에 나타낸 이름의 드라이버를 로드 한다.

8.2 스태틱 VxD의 초기화와 종료

(Initializing and Terminating a Static VxD)

윈도우즈 95는 정적으로(스태틱하게) 로드한 드라이버를 네 단계에 걸쳐 초기화를 진행한다. 프로세서가 보호모드로 전환되기 전, VMM은 VxD가 만약 *리얼모드 초기화 함수(real-mode initialization function)*를 가지고 있다면 이를 호출한다. 보호모드로 전환한 후, 그러나 프로세서의 인터럽트가 디스에이블 된 상태에서 VMM은 `Sys_Critical_Init` 메시지를 보내며, 인터럽트가 인에이블 된 후 `Device_Init` 메시지를 보낸다. 대부분의 드라이버가 대부분 이 시점에서 초기화를 수행한다. (역자 주 : `Device_Init` 메시지에서)

8.2.1 리얼모드에서의 초기화 (Real-Mode Initialization)

컴퓨터를 부팅 시켰을 때, 프로세서는 리얼모드에서 시작한다. 프로세서는 MS-DOS를 초기화하는 동안, MS-DOS가 CONFIG.SYS에 지정한 드라이버를 로드 하는 동안, MS-DOS가 AUTOEXEC.BAT 파일을 처리하는 동안 리얼모드로 남아 있다. AUTOEXEC.BAT 처리를 마쳤을 때, MS-DOS는 윈도우즈 95를 로드하기 위해 자동으로 WIN 명령을 발생시킨다. 따라서 윈도우즈 95는 리얼모드에서 동작을 시작한다. 윈도우즈 95는 로드 할 스태틱 가상 디바이스 드라이버를 결정한 직후 보호모드로 전환한다.

설명 (Note)

이 설명란에서 대부분 많이 아는 체를 했다. 그러나 이것은 꼭 언급해야 하겠다. 만약 AUTOEXEC.BAT 과 일을 처리하는데 EMM386 같은 익스텐디드 메모리 관리자 로드 하도록 했다면, 윈도우즈 95가 시작할 때 컴퓨터는 실제 V86모드에 있을 것이다. 이것은 초기화하는 동안 선형 어드레스 공간의 하위 1MB를 액세스하는데 영향을 미칠 수 있다. 왜냐하면 윈도우즈 V86MMGR 디바이스가 메모리 관리자에서 상위 메모리 블록(upper memory block)의 관리를 인수받아야 하기 때문이다. 이 인수 작업은 V86MMGR의 Sys_Critical_Init 핸들러의 동작에서 일어난다.

초기화 개시 직후, 보호모드로 전환하기 전, VMM은 리얼모드 초기화를 수행한다. 이 초기화 단계의 한 부분이 INT 2Fh 함수 1605h인 시작알림(startup broadcast)이며, 이것은 램 상주 프로그램에게 이들이 필요로 하는 드라이버를 지정할 수 있는 기회를 제공한다. 이 단계 과정에서 VMM은 또한 레지스트리와 SYSTEM.INI의 [386enh] 섹션의 device= 문장을 읽어 들인다. VMM은 옵션널인 리얼모드 초기화 함수를 호출하기 위해 각 드라이버 파일을 검사한다. 리얼모드 초기화 함수는 다음의 것 중 어떤 것이라도 혹은 모두 다를 수행할 수 있다.

- 같은 드라이버가 여러 개 로드 되는 것을 견제하고 로드 할 드라이버를 복사할 것을 선택한다.
- 인스턴스 데이터 영역(instance data)을 제공한다.
- 가상 디바이스에 포함된 고정 메모리 페이지를 식별한다.
- 다른 리얼모드 프로그램과 통신할 수 있다.
- 윈도우즈 95의 시작을 막을 수 있다.

인스턴스 데이터(instance data)

모든 가상 머신의 똑같은 어드레스를 점유하는 데이터, 그러나 이것은 각 가상 머신마다 서로 다른 데이터를 가질 수 있다. 윈도우즈 95에서 이 인스턴스 데이터란 하드웨어 디바이스에 속하지 않은 하위 1MB의 주소 공간에 초기에 위치된 데이터로 간주한다. 예를 들어, DOSKEY의 명령어 저장 버퍼(command recall buffer)가 인스턴스 데이터로 간주될 수 있으나, 비디오 램은 그렇지 않다. 비록 양쪽 영역이 서로 같은 주소를 가지고 있다하더라도 다른 VM에서는 다른 내용을 가진다.

어쨌든 모든 가상 디바이스 드라이버에 리얼모드 초기화 함수가 필요한 것은 아니다. 이를 제공하는 일반적인 이유는 리얼모드 코드를 호출해야만 얻을 수 있는 정보를 리얼모드가 임시적으로 액세스 불가능하게 되는 Sys_Critical_Init 초기화 단계에서 필요하게 되기 때문이다. 다른 이유는 로드 하도록 요구된 드라이버가 중복되었는지를 감지해 내기 위함이다.

(1) 프로그램 구조 (Program Structure)

리얼모드 초기화 함수 및 여기에 사용되는 데이터를 VxD의 한 개 16비트 세그먼트로 묶고 싶을 경우가 있다.

```
VxD_REAL_INIT_SEG      ; _RCODE 세그먼트의 시작
...
VxD_REAL_INIT_ENDS    ; _RCODE 세그먼트의 끝
```

이를 위해서는 초기화 세그먼트의 영역을 정하는 VxD_REAL_INIT_xxx 세그먼트 매크로를 사용한다. 만약 윈도우즈 3.1 DDK툴과 LINK386을 사용하고 있다면, END 문장에 VxD 진입점의 이름과 같은 리얼모드 초기화 함수의 시작점을 지정해야 한다. 그러나 비주얼 C++이나 윈도우즈 95 DDK의 링커를 사용한다면 END 문장에 아무 것도 넣지 않아도 된다.

(2) 진입조항과 종료상황(Entry and Exit Conditions)

리얼모드 초기화 함수에 진입할 때, 레지스트리는 다음의 표 8-1과 같은 값을 가지고 있다. 레지스터 AX에 있는 버

전이란 윈도우즈 95에서는 0400h, 윈도우즈 3.1에서는 030Ah 등등이다. 레지스터 ECX에는 서비스 루틴의 어드레스가 있는데, 이 서비스 루틴은 레지스트리와 SYSTEM.INI 파일에 있는 항목 검사나 리얼모드 로더와 통신하기 위해 호출할 수 있다. 레지스터 EDX에 있는 참조 데이터는 SYSTEM.INI 파일의 device= 문장이나 레지스트리를 통해 로드 되었다면 0일 것이다.

레지스터	내 용
AX	VMM 버전 번호(예를들어, 0400h나 030Ah이다)
BX	플래그는 다음과 같다. 비트 0 : Duplicate_Device_ID - 만약 설정되어 있다면, 이미 같은 ID를 가진 다른 VxD가 로드되었다. 비트 1 : Duplicate_From_INT2F - 만약 설정되어 있다면, INT2Fh 함수 1605h에 의해 이전에 드라이버가 벌써 로드되었다. 비트 2 : Loading_From_INT2F - 이 드라이버는 INT 2Fh 함수 1605h에 의해 로드 되고 있다.
ECX	초기화 서비스 진입점의 세그먼트:오프셋 주소 (윈도우즈 3.1 이상)
EDX	0이거나 INT2Fh 함수 1605h의 시작 정보 구조체의 참조 데이터
SI	MS-DOS 환경영역의 세그먼트
CS, DS, ES	_RCODE 세그먼트를 가지고 있는 패러그래프

표 8-1. 리얼모드 초기화 진입에 있어서 레지스터의 내용

버전 번호(Version Numbers)

윈도우즈 95는 버전번호로써 Win32 프로그램에게는 0004(다른 말로, 버전 4.00), VxD에게는 0400이라 알려 준다. 그러나 Win16 응용 프로그램은 *GetVersion*을 호출하면 5F03(다른 말로, 버전 3.95)라는 값을 얻게 될 것이다. 윈도우즈 95가 Win16 프로그램에게 이런 거짓말을 하는 이유는 너무 많은 Win16 프로그램이 잘못된 방법으로 *GetVersion*을 사용하기 때문이다. 어떤 프로그램은 “버전 2보다 이후 버전인가?”라는 뜻으로 “버전 3인가?”라는 식으로 물어본다. 또다른 어떤 프로그램은 *GetVersion*의 리턴 값을 비교하기 전에 상위 바이트와 하위 바이트를 바꾸지 않기 때문에, 0004h(4.00)이 0A03h(3.10)보다 작은 값으로 잘못 판단된다.

레지스터 BX에 있는 드라이버 중복 플래그(duplicate-driver-flag, 비트 0과 1)는 이 드라이버가 로드 된 드라이버와 같은 드라이버 식별자를 사용하고 있다는 것을 가리킨다. 비트 1 (*Duplicate_From_INT2F*)은 같은 식별자(ID)를 가지고 있는 첫 번째 드라이버를 이것으로 간주한다. 비트 2(*Loading_From_INT2F*)를 검사해서 드라이버가 INT 2Fh 함수 1605h를 통해 로드 되었는지를 알아낼 수 있다.

리얼모드 초기화 함수를 끝낼 때는 표 8-2에 나타낸 것과 같이 레지스터를 설정하고 근거리 리턴(near return) 명령을 수행한다. 초기화 함수에서의 정상 리턴을 나타낼 때는 AX 레지스터를 0(*Device_load_OK*)로 설정한다.

레지스터	내용
AX	다음과 같은 값을 가지고 있는 리턴 코드이다. 비트 0 : Abort_Device_Load - 만약 설정되면, 이 VxD를 로드하지 않는다는 것을 알린다. (그러나 같은 ID를 가지고 있으나 다른 방법으로 로드가 요청된 VxD는 로드될 수 있다.) 비트 1 : Abort_Win386_Load - 만약 설정되면, 윈도우즈 95가 시작되지 않도록 한다. 비트 15 : No_Fail_Message - 만약 설정되면, VMM에게 이 드라이버를 로드하지 않는다는 에러 메시지를 표시하지 않도록 한다.
BX	소유 페이지의 리스트 포인터
ED	보호모드 초기화를 위한 참조 데이터
SI	인스턴스 데이터 항목의 리스트 포인터

표 8-2. 리얼모드 초기화 함수를 끝낼 때에 설정되어야 하는 레지스터의 내용

필자는 근거리 리턴(near return) 명령을 사용해야 한다는 것이 놀라웠다. 이것은 실행 시 VMM이 리얼모드 초기화 세그먼트(real-mode initialization segment)에 약간의 코드를 추가하는 것 같다. 그리고 나서 VMM은 이 추가된 코드를 원거리 호출(far call)하고, 이 추가된 코드는 우리가 만든 초기화 함수를 근거리 호출(near call)하는 것이다. 그래서 결국 근거리 리턴은 VMM으로 제어권을 되돌려 주기 위해 원거리 리턴을 하는 명령에 도달하는 것이다.

소유 페이지 리스트(owned pages list)는 0으로 끝나는 단정수(short integer)의 벡터이다. 인스턴스 데이터 항목 리스트(instance data item list)는 32비트 0으로 끝나는 Instance_Item_Struct 구조체의 벡터이다. (그림 8-2 참조) 소유 페이지 벡터와 인스턴스 항목 벡터 둘 다 리얼모드 함수의 코드와 데이터가 있는 _RCODE 세그먼트에 위치한다. 소유 페이지의 개념은 이 절의 (4)에서 논의하며, 인스턴스 데이터는 10장에서 설명할 것이다.

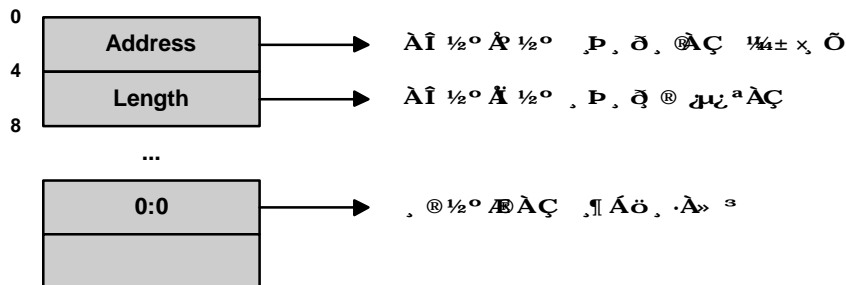


그림 8-2. 인스턴스 데이터 항목 구조체의 리스트

(3) 중복 검사(Checking for Duplication)

만약 디바이스 드라이버 로드 요청에 중복이 발생했으리라 의심되면, 리얼모드 초기화 함수에서 중복 감지를 할 수 있다. 예를 들면, 다음과 같다.

```

rminit: test bx, Duplicate_Device_ID      ; 중복되었는가?
        setnz al                          ; 그렇다면 1로 설정
        cbw                               ; ...
        xor bx, bx                        ; 소유 페이지는 없다.
        xor si, si                        ; 인스턴스 데이터도 없다
        ret

```

SETNZ 명령은 만약 이전의 TEST 명령이 0이 아니면 AL 레지스터를 1로 설정한다. AL 레지스터에 있는 이 값은 Abort_Device_Load 플래그에 해당된다.

왜 윈도우즈 95는 자동으로 드라이버 중복을 감지해 내고 첫 번째로 지정된 드라이버만을 로드하지 않는지 궁금히 여길지 모르겠다. 왜 윈도우즈 95는 리얼모드 초기화 함수를 호출하며 여기를 통해 이를 결정하도록 했을까? 왜냐하면 각각의 드라이버 요청은 소유 페이지를 선언을 해야 하거나 컨벤셔널 메모리 영역에 인스턴스 데이터를 지정해야 함을 가

리킬 수도 있기 때문이다. 윈도우즈 95는 이런 가능성들을 수용하기 위해 초기화 함수를 호출하는 것이다. 그러나 이것은 또한 `Abort_Device_Load` 플래그를 리턴 한다고 하더라도 BX 레지스터와 SI 레지스터를 통하여 소유페이지와 인스턴스 데이터 영역을 설정해야 한다는 것을 의미한다. (역자주 : 드라이버는 로드하지 않더라도 소유페이지와 인스턴스 데이터 영역은 사용될 수 있기 때문이다.)

(4) 소유 페이지 (Owned Pages)

윈도우즈 95는 새로운 가상 머신마다 컨벤셔널 메모리 영역에 대하여 페이지 테이블을 만든다. 이러한 각각의 테이블은 직접 물리 어드레스에 해당하는 가상 어드레스를 맵핑하기도 하고 익스텐디드 메모리의 페이지에 가상 어드레스를 맵핑하기도 한다. 매우 우발적이기는 하지만 가상 어드레스 맵핑의 디폴트 선택을 바꾸고 싶다면 소유 페이지 리스트 (owned page list)에서는 이것도 할 수 있다.

윈도우즈 95의 EBIOS 드라이버를 설명하면 이 소유 페이지를 쉽게 설명할 수 있을 것 같다. PS/2 컴퓨터에서는 BIOS가 임시적으로 읽기나 쓰기를 위한 장소를 위해서 확장 BIOS 데이터(extended BIOS data; EBIOS) 영역을 할당한다. 호환성 때문에 0040h:0000h에 있는 EBIOS 영역의 크기와 사용에 제한을 받는다. 따라서 PS/2의 BIOS는 추가적인 물리 메모리 영역을 사용해야 했다. 각 가상 머신은 리얼모드 어드레스와 똑같이 동작하는 가상 어드레스에 EBIOS 영역을 지정해야 하며 그렇지 않으면 가상 머신에서의 EBIOS는 제대로 동작하지 않을 것이다. 윈도우즈 95의 다른 요소들은 EBIOS 영역이 존재한다고 알지 못할 것이다. 왜냐하면 BIOS는 물리 메모리의 끝을 알려주는데 있어서(소프트웨어 인터럽트 12h를 통해서) EBIOS의 시작 앞쪽을 알려주기 때문이다. 따라서 EBIOS 어드레스 영역을 익스텐디드 메모리 페이지에 맵핑하는 소유 페이지의 디폴트 동작은 올바르지 않기 때문에 EBIOS 드라이버는 소유 페이지의 디폴트 선택을 바꿔야 한다.

EBIOS 드라이버의 리얼모드 초기화 함수는 640KB 컨벤셔널 메모리의 뒤쪽 40KB를 위치 시킬 것인지를 결정하고 만약 그렇다면 A000h:0000h까지의 EBIOS 영역 시작점을 선언한다. 이것은 리턴되는 레지스터 BX가 0으로 끝나는 16비트 정수 배열을 가리키도록 해서 이를 수행한다. 예를들어, EBIOS 영역이 9E80h:0000h에서 시작한다면 배열은 다음과 같이 3개의 항목을 가지고 있을 것이다.

```
exc_bios_page dw 9Eh, 9Fh, 0
```

나중에, 각각 세 가상머신이 생성될 때, EBIOS 드라이버는 물리 페이지 9Eh와 9Fh를 가상머신의 가상 페이지인 9Eh와 9Fh로 각각 맵핑 되기 위해 `_PhyIntoV86` 서비스를 사용한다.

(5) 디바이스 지령 프로토콜 (Device Callout Protocol)

리얼모드 초기화 함수에서, 일반적인 방법과 같이 BIOS와 MS-DOS 기능들을 사용할 수 있다. 만약 MS-DOS 디바이스 드라이버나 TSR과 같은 상주 프로그램과 통신하기를 원한다면 기계어 어떠한 방법이라도 사용할 수 있다. 그러나 마이크로소프트는 이 통신을 표준화하기 위해 소프트웨어 인터럽트 2Fh를 기반으로 하는 **디바이스 지령(device callout)** 변환을 정의하였다.

디바이스 지령을 사용하기 위해서는 다음과 같이 범용 레지스터를 적재시킨다. 레지스터 AX에는 1607h를 설정하고 레지스터 BX에는 VxD의 유일한 식별자를 설정한다. 그리고 나서 인터럽트 2Fh를 발생시킨다. 이 지령을 기다리는 상주 소프트웨어는 이 인터럽트를 후크 해 있었을 것이고, 함수 코드로 1607h와 BX 레지스터가 디바이스 ID와 같은 것을 찾을 것이다. 호출에 사용되는 다른 레지스터의 내용과 INT 2Fh에서 리턴 되는 모든 레지스터의 내용은 전적으로 우리 맘대로 정할 수 있다.

(6) 리얼모드 초기화 서비스 (Real-Mode Initialization Service)

리얼모드 초기화 함수 진입시 ECX 레지스터는 레지스트리와 SYSTEM.INI파일의 디코딩 항목에 대한 몇 가지 유틸리티 함수를 제공하는 서비스 루틴의 주소인 페리그래프:오프셋을 가지고 있다. 이런 서비스 중의 하나를 사용하려면, 우선 DWORD 메모리에 이 서비스 루틴의 주소를 저장해야 한다.

```
rminit: mov     _ServiceEntry, ecx
...
_ServiceEntry dd 0
```

이 서비스 루틴의 주소가 저장된 변수인 `_ServiceEntry`를 호출하는 것은 `VMMREG.H`에 있는 기능의 레지스터 액세스가 훨씬 쉽게 된다.

특정 서비스를 사용하기 위해서는 AX 레지스터에 함수 인덱스를 넣고 이 저장된 포인터를 이용하여 원거리 호출을 실행한다.

```
mov     ax, n           ; AX = 함수의 인덱스
call   [_ServiceEntry] ; 서비스 루틴을 호출
```

사용 가능한 서비스 루틴을 표 8-3에 나타내었다. 필자는 이들 루틴에 대한 참고 문서를 찾기 힘들다는 것을 알았다. 처음의 여섯 가지 서비스(0h에서 6h까지는 윈도우즈 95 DDK의 "Programmer's Guide"에 "Reference"라고 하는 제목부에 "Real-Mode Initialization"절에 문서화 되어있다. 이들에 대한 내용을 찾기 위해 MSDN에서 `LDRSRV_COPY_EXTENDED_MEMORY`라는 키워드를 찾아보아라. 이 레지스트리 서비스는 윈도우즈 95 DDK의 "Kernel Service Guide"에 "Real-Mode Function"이라는 제목의 "Registry Service Reference"절에 있다. 그러나 `LDR_RegCloseKey`라는 키워드로 MSDN을 찾아보아라.

AX 내용	함수 이름
00h	LDRSRV_GET_PROFILE_STRING
01h	LDRSRV_GET_NEXT_PROFILE_STRING
03h	LDRSRV_GET_PROFILE_BOOLEAN
04h	LDRSRV_GET_PROFILE_DECIMAL_INT
05h	LDRSRV_GET_PROFILE_HEX_INT
06h	LDRSRV_COPY_EXTENDED_MEMORY
100h	LDR_RegOpenKey
102h	LDR_RegCloseKey
105h	LDR_RegQueryValue
106h	LDR_RegEnumKey
108h	LDR_RegEnumValue
109h	LDR_RegQueryValueEx

표 8-3. 리얼모드 초기화 서비스 호출

예를 들어, 다음과 같이 디바이스의 I/O 포트 주소를 나타내는 `SYSTEM.INI` 파일에 `[myvxd]` 섹션을 가지고 있다고 하자.

```
[myvxd]
port=1234h
```

이 값을 읽기 위한 코드는 다음과 같을 것이다.

```
rminit:  mov     _ServiceEntry, ecx      ; 서비스 어드레스 저장
...
mov     ax, LDRSRV_GET_PROFILE_HEX_INT ; 즉, 05h
mov     ecx, -1                        ; ECX = 디폴트 값
mov     si, offset secname             ; DS:SI -> 섹션 이름
mov     di, offset varname             ; DS:SI -> 셋팅 이름
call   [_ServiceEntry]                 ; get SYSTEM.INI 셋팅
mov     ioaddr, ecx                    ; (리턴 값은 ECX에 있다)
...

_ServiceEntry dd 0
secname       db 'myvxd', 0
varname       db 'port', 0
```

```
ioaddr          dd 0
```

레지스트리를 액세스하는 코드는 좀 복잡하다. 왜냐하면 드라이버가 레지스트리 키를 로드 하려고 하면 엄청난 양의 레지스트리 키의 구조에 대하여 알고 있어야 하기 때문이다. 여기에 리얼모드 초기화 동안 레지스트리에서 포트 주소를 얻기 위한 근사한 코드가 있다. (이 코드는 부록 CD의 /CHAP8/REALMODEINIT 디렉토리에 포함되어 있다.

```
; RMREG.ASM -- Test of real-mode registry access

.386p
include vmm.inc
include vmmreg.inc
include regstr.inc

Declare_Virtual_Device RMREG, 1, 0, rmreg_control,\
    Undefined_Device_Id, Undefined_Init_Order

Begin_Control_Dispatch RMREG
End_Control_Dispatch RMREG

VxD_REAL_INIT_SEG

; INT 3

    mov     _ServiceEntry, ecx

LDR_RegOpenKey HKEY_LOCAL_MACHINE, <offset namevxd>, ds,\
    <offset hvxd>, ds

    test   ax, ax
    jnz   fail1

LDR_RegOpenKey hvxd, <offset myname>, ds, <offset hme>, ds

    test   ax, ax
    jnz   fail2

Ldr_RegQueryValueEx hme, <offset portname>, ds, 0, 0, 0, <offset port>, ds, <offset portsize>, ds

    test   ax, ax
    jnz   fail3
    ...           ; "port"의 값을 가지고 어떤 작업을 한다.
fail3:
    LDR_RegCloseKey hme
fail2:
    LDR_RegCloseKey hvxd
fail1:

alldone:
    xor    ax, ax
    xor    bx, bx
    xor    si, si
    ret

_ServiceEntry dd 0

namevxd        db  REGSTR_PATH_VXD, 0
hvxd           dd  0
```

```

myname      db  'RMREG', 0
hme        dd  0
portname    db  'port', 0
port        dd  0
portsize    dd  size port
VxD_REAL_INIT_ENDS

```

end

필자는 이 예제에서 VMMREG.INC에 있는 몇 가지 매크로를 사용했으며, 이들 매크로는 모두 초기화 서비스 주소를 가지고 있는 *_ServiceEntry*라는 이름의 DWORD 변수를 참조한다.

해당 디바이스의 레지스트리 키를 열기 위해서는 우선 그 키의 이름을 결정해야 한다. REGSTR.INC에 정의되어 있는 *REGSTR_PATH_VXD*라는 상수는 /System/CurrentControlSet/Services/VxD 인데, RMREG (역자 주 : 위의 예제에서 VxD의 이름)의 자체적인 키를 여는 가장 손쉬운 방법은 우선 이 패스로 된 키를 연 다음 그 밑에 있는 RMREG라고 하는 서브키를 여는 것이다. 여기에는 *LDR_RegOpenKey*를 두 번 호출하고 이 열려진 키에서 특정 설정을 얻기 위해서는 *LDR_RegQueryValueEx*를 사용한다.

앞에서 보여준 리얼모드 예제에 내재해 있는 주요 문제는 레지스트리 항목을 넣을 곳에 대하여 결정하는 것이 컴파일할 때라는 것이다. 나중에 보여주겠지만 VxD의 보호모드 초기화 함수에서 적당한 레지스트리 키를 액세스하는 것이 좀 더 쉽고 매우 강인한 방법이다. 이것은 VxD가 로드 해야 할 레지스트리 패스를 *_GetRegistryPath* 서비스가 계산해 주기 때문이다. 그래서 레지스트리에 대하여 알아두어야 하는 양이 줄어들게 된다.

(7) 보호모드 초기화 함수와의 통신 (Communicating with Protected-Mode Initializers)

리얼모드 초기화 함수는 리턴 할 때 EDX 레지스터에 임의의 32비트 참조 데이터(reference data)를 설정하는 방법으로 드라이버의 보호모드 부분과 정보를 통신할 수 있다. 이 값은 윈도우즈 95가 드라이버에게 보호모드 초기화 메시지를 보낼 때 EDX에 담기게 된다. 윈도우즈 95는 또한 *Declare_Virtual_Device* 매크로로 생성된 드라이버 디스크립션 블록의 *DDB_Reference_Data* 필드에도 저장되게 된다.

윈도우즈 95에서, 익스텐디드 메모리의 블록을 할당하고 초기화하기 위해 *LDRSVR_Copy_Extended_Memory* 서비스를 사용한다. 이 메모리 블록의 주소는 결국 드라이버의 보호모드 부분에서 사용되도록 레지스터 EDX를 통해 전달된다. 예를 들어, 윈도우즈 95 구성 관리자의 리얼모드 초기화 함수는 자체 드라이버의 보호모드 초기화 함수로 다소 긴 데이터 영역을 전달하기 위해 이 서비스를 사용한다. 이 데이터 영역에는 현재 하드웨어의 프로필이 들어 있다. *LDRSVR_Copy_Extended_Memory* 서비스가 존재하기 전인 이전 버전의 윈도우즈에서는 이를 리얼모드 메모리에 할당해야 했으며 이에 대한 어드레스를 레지스터 EDX를 통해 보호모드 초기화 함수로 전달된다. 이 경우 윈도우즈가 종료하기 전까지는 아무도 이 메모리를 해제하지 않기 때문에 모든 MS-DOS 가상 메모리에 있어 영구적인 외판점으로 존재하게 된다. *LDRSVR_Copy_Extended_Memory*를 사용하는 것은 리얼모드 메모리가 분할되는 것을 방지할 뿐만 아니라 보호모드로 전환되기 전인 리얼모드 초기화 함수에서 할당한 어떤 메모리라도 윈도우즈 95가 해제할 수 있기 때문이다.

8.2.2 보호모드 초기화 (Protected-Mode Initialization)

어떤 드라이버가 로드 되어야 할지 결정된 후, VMM은 프로세서를 리얼모드에서 보호모드로 전환한다. 이 전환은 프로세서 인터럽트가 디스에이블 된 채로 일어난다. 그리고 나서 VMM은 각 디바이스 드라이버의 콘트롤 프로시저에 *Sys_Critical_Init* 메시지를 보낸다. 모든 드라이버가 이 메시지를 처리했을 때, 프로세서의 인터럽트를 인에이블 시키고 각 드라이버에게 *Device_Init* 메시지를 보낸다. 그리고 나서 초기화 처리를 끝내기 위해 각 드라이버에게 *Init_Complete* 메시지를 보낸다.

VMM은 각각의 *Declare_Virtual_Device*에 나타낸 것과 같이 오름차순의 초기화 순서(ascending initialization order)로 이러한 메시지들을 보낸다. 마이크로소프트에 의해 제공되는 각 드라이버들은 VMM.H 파일에 정의된 초기화 순서 상수를 가지고 있다. 다음에는 이러한 리스트를 간략히 인용했다.

```

#define VMPOLL_INIT_ORDER      0x06400000
#define UNDEFINED_INIT_ORDER   0x08000000
#define WINDEBUG_INIT_ORDER    0x08100000

```

```
#define VDMAD_INIT_ORDER 0x09000000
```

즉, WINDEBUB는 VMPOLL 이후와 VMMAD 이전에 초기화된다. 인터럽트에 있어서 이를 나중에 후크 하는 프로그램이 우선권을 가지고 있어서 MS-DOS 드라이버나 TSR의 로드 순서가 종종 중요한 것과 같은 이유로 이 초기화 순서는 가끔 중요하다. 또는 다른 VxD에 의해서 제공되는 서비스에 의존적인 경우 이 다른 VxD 뒤에 초기화되어야 한다. 만약 드라이버의 초기화 순서가 상관없다면 *Undefined_Init_Order*를 사용한다. 만약 드라이버가 특정 드라이버의 바로 앞 혹은 바로 뒤에 초기화되기를 원한다면 그 특정 드라이버의 초기화 순서보다 좀 적거나 큰 값을 선택한다. 우리가 만든 드라이버와 이웃한 드라이버 사이에 다른 사람들이 사용할 수 있는 영역을 남겨놓기 위하여 인접한 초기화 순서 상수간에는 많은 공간이 있다. 그래서 우리가 만든 드라이버와의 순서에 있어 주목의 대상이 되는 드라이버간에는 대략 1000h 정도의 간격을 띄워라.

(1) 호출 순서 (Calling Sequences)

VMM은 세 개의 보호모드 초기화 메시지에 있어서 같은 호출 방식을 사용하고 있다. 표 8-4는 이들 메시지의 호출에 대한 범용 레지스터의 내용을 보여준다.

레지스터	내용
EAX	0(Sys_Critical_Init), 1(Device_Init), or 2(Init_Complete)
EBX	시스템 VM의 핸들 (VM 컨트롤 블록 어드레스)
EDX	리얼모드 초기화 함수에서 보낸 참조 데이터인데, 리얼모드 초기화 함수가 없는 경우에는 0이 온다.
ESI	VEMM386.EXE의 커맨드 명령어 라인의 주소이다. 이것은 PSP(Program Segment Prefix)에 있는 커맨드 명령어 라인의 뒷부분인데, 앞에는 바이트의 수가 오며, 뒤에는 0Dh로 끝난다.

표 8-4. 보호모드 초기화 컨트롤 메시지의 레지스터 내용

만약 이 세 개의 메시지들 중 하나에서라도 VxD가 캐리 플래그를 세팅한 채로 (혹은 C언어에서 리턴 값이 FALSE 인 경우) 리턴 한다면, VMM은 이 드라이버의 로드를 중단한다. VxD는 이러한 수행이 정상적으로 완료되었음을 나타내기 위해 캐리 플래그를 지운 채로 리턴 해야한다. (혹은 C언어에서라면 TRUE를 리턴 해야 한다.)

디바이스 컨트롤 프로시저는 어셈블리어로 쓰여지고 나머지는 C로 작성된 VxD에서 이 세 개의 초기화 메시지를 처리하는 대체적인 골격은 다음과 같다. (이 코드는 부록 CD의 /CHAP08/C-D아 디렉토리에 있다.)

DEVDECL.ASM

; DEVDECL.ASM -- Assembly language interfaces for Sample VxD

```
.386p
include vmm.inc
include debug.inc

Declare_Virtual_Device MYVXD, 1, 0, MYVXD_control,\
    Undefined_Device_ID, Undefined_Init_Order

Begin_Control_Dispatch MYVXD

Control_Dispatch Sys_Critical_Init, OnDeviceInit,      sCall, <ebx, edx>
Control_Dispatch Device_Init,      OnSysCriticalInit, sCall, <ebx, edx>
Control_Dispatch Init_Complete,    OnInitComplete,   sCall, <ebx, edx>

End_Control_Dispatch MYVXD

end
```

MYVXD.VXD

// MYVXD.C -- Sample Virtual Device Driver

```
#define WANTVXDWRAPS

#include <basedef.h>
#include <vmm.h>
#include <debug.h>
#include <vxdwraps.h>

#pragma VxD_ICODE_SEG
#pragma VxD_IDATA_SEG

BOOL _stdcall OnDeviceInit(PVMMCB hVM, DWORD refdata)
{
    // OnDeviceInit
    return TRUE;
}

BOOL _stdcall OnSysCriticalInit(PVMMCB hVM, DWORD refdata)
{
    // OnSysCriticalInit
    return TRUE;
}

BOOL _stdcall OnInitComplete(PVMMCB hVM, DWORD refdata)
{
    // OnInitComplete
    return TRUE;
}
```

(2) Sys_Critical_Init

VMM은 보호모드로 전환한 후 디바이스 드라이버에게 Sys_Critical_Init 메시지를 보내지만 인터럽트는 디스에이블되어있다. 대부분의 드라이버는 이 메시지를 처리할 필요가 없다. 왜냐하면 여기서는 주로 다음의 것들을 처리할 것이기 때문이다.

- 우리가 만든 드라이버는 소프트웨어 인터럽트에 있어 도스화장자처럼 동작한다. 드라이버는 MS-DOS 인터럽트와 BIOS 소프트웨어 인터럽트의 보호모드 호출을 처리해야할 책임이 있으며 이 인터럽트를 가상 86모드에서 실행한다. 이것은 아마 보호모드에서 호출한 인터럽트의 인자 중 포인터 인자를 리얼모드 포인터로 변환하는 것이다. Sys_Critical_Init동안 인터럽트를 후크(Set_PM_Int_Vector를 사용하여)하는데, 이것은 다른 VxD들이 Device_Init 호출을 받았을 때부터 Exec_VxD_Int를 통해 이 인터럽트를 자유롭게 호출할 수 있게 하기 위함이다.
- 우리가 만든 드라이버는 Device_Init 동안에 다른 VxD에서 사용하는 몇 개의 서비스를 제공하기도 한다. 예를 들어, V86MMGR 디바이스는 Sys_Critical_Init 동안 상위 메모리 블록(upper memory block)의 관리를 장악해서 윈도우즈 95가 시작하기 전 리얼모드 프로그램이 사용했던 모든 메모리를 VxD가 액세스할 수 있도록 해준다.
- 최후로 불러질 디폴트 인터럽트 핸들러를 제공하고 싶다면, Sys_Critical_Init 동안 인터럽트를 후크해서 VMM이 설치한 디폴트 핸들러를 얻는다. (왜냐하면 VMM이 우리가 만든 드라이버보다 먼저 초기화되기 때문이다.) 그러나 이 인터럽트도 Device_Init나 그 이후에 인터럽트를 후크 한 디바이스 드라이버에 의해 조종된다.
- 우리가 만든 드라이버에서 이 드라이버보다 더 먼저 초기화되는 드라이버에게 VxD 서비스를 익스포트 한다. 따라서 이 경우 Device_Init 동안 이 서비스를 제공하기 위해 몇 개의 데이터를 초기화해야 한다.

이 메시지를 처리하는 동안 인터럽트를 인에이블 할 수 없다는 것은 말할 것도 없다. 물론 리얼모드 코드를 실행하기 위한 서비스들(Exec_Init 같은 것)을 사용하지 말라. 마지막으로, 이 메시지를 처리하는 동안 과도한 지연으로 하드웨어 인터럽트를 놓치지 않도록 작업량을 최소화하라.

(3) Device_Init

윈도우즈 95는 인터럽트를 인에이블 한 후 Device_Init 메시지를 보낸다. 많은 디바이스 드라이버가 이 메시지 처리에서 대부분의 초기화를 수행한다. 왜냐하면 인터럽트가 인에이블 되어있기 때문에, 시스템에 부작용 없이 시간이 많이 소비되는 작업을 수행할 수 있으며, 리얼모드 코드도 수행할 수 있기 때문이다. Device_Init 동안에 무엇을 해야 할지는 전적으로 VxD의 목적에 달려 있으므로, 일부러 분명히 얘기하기 못하고 있다.

(4) Init_Complete

모든 디바이스 드라이버를 초기화한 후에, 초기화 세그먼트를 해제하고 10장에서 설명하는 인스턴스 스냅 샷(역자 주 : 역자는 아직 이 뜻을 이해하지 못하고 있다. 10장에서 보도록 하자)을 얻기 전에, VMM은 Init_Complete 메시지를 보낸다.

몇 개의 디바이스 드라이버는 Init_Complete 메시지를 처리할 필요가 있다. 이렇게 하는 한가지 이유는 DOSMGR 디바이스에서 다음과 같은 것이 일어나기 때문이다. 즉, 많은 리얼모드 MS-DOS 디바이스 드라이버는 전부의 메모리 혹은 그 드라이버가 소유한 일부 메모리를 인스턴스화 해야 한다. 요즘의 드라이버는 이를 수행하기 위해 INT 2Fh를 후크한 다음 INT 2Fh 함수 1605h 알림을 사용하지만, 모든 드라이버가 이렇게 하는 것이 아니므로 VxD는 윈도우즈 95임을 알아채지 못하는 리얼모드 드라이버를 위해 이에 대한 처리를 세심하게 수행해야 한다. (역자 주 : 혹시 윈도우즈 95의 도스창에서 볼랜드 C/C++의 램 상주용 티보 헬프를 사용해 본적이 있는가. 이런 것을 본다면 이 동작들이 아주 잘 이루어지는 것은 아닌 것 같다.) DOS_Instance_Device 서비스는 이런 것을 수행하지만 이것은 오로지 Init_Complete 동안에만 유효하다.

(5) 초기화에 유용한 서비스들 (Services That Can be Useful During Initialization)

초기화 동안에 레지스트리나 SYSTEM.INI 파일의 항목을 알아내야 할 필요가 있을 때가 있다. 이러한 목적에 맞는 몇 가지 서비스가 있다.

- 레지스트리 서비스(_RegQueryValue 같은 것)는 레지스트리 데이터베이스를 액세스할 수 있도록 해준다.
- 프로파일 서비스를 알아내는 서비스(Get_Profile_Decimal_Int 같은 것)는 SYSTEM.INI 파일로부터 어떤 값을 읽어 들인다.
- 변환 서비스(Convert_Boolean_String 같은 것)는 문자열을 숫자 값으로 변환해 준다. 대개 이 문자열이란 Get_Profile_String을 통해 SYSTEM.INI 파일에서 읽어들이는 것이다.

디바이스 초기화 동안에만 사용할 수 있는 서비스들도 있다. 예를 들어, Get_Name_Of_Ugly_TSR은 디바이스 드라이버가 올바르게 동작하는데 방해가 될 수 있는 “비협동(uncooperative)” TSR을 알아낼 수 있도록 해 준다. 그러나 그렇다면

이것은 드라이버가 초기화되는 동안에만 유용할 것이다.

레지스트리 서비스를 제외하고 여기서 설명한 모든 서비스는 물리적으로 초기화 코드 세그먼트에 위치해 있어서 Init_Complete 이후에는 사용할 수 없다.

레지스트리 서비스를 사용하는 것. 다음의 문단은 드라이버를 구성하기 위해 초기화 서비스의 사용을 실제로 보여주고 있다. MYVXD.VXD라는 이름으로 작성된 I/O 포트 가상화 드라이버가 있다고 가정하자. 문제를 간단히 하기 위해 MYVXD는 윈도우즈 95의 플러그 앤 플레이와는 전혀 관계가 없는 고전적인 디바이스 드라이버라고 한다. 지금까지 지정된 연습을 잘 따라 했다고 하고 드라이버를 로드하기 위해 시스템 레지스트리를 사용한다고 하자. //HKLM/System/CurrentControlSet/Services/VxD/MYVXD의 키가 다음과 같은 값을 가지고 있다.

```
StaticVxD=C:/MyProd/Myvxd.vxd
Port=1234h
```

Static VxD 항목은 로드 할 드라이버로서 우리가 만든 드라이버이다. Port 항목은 바이너리 데이터를 가져야 하며 이것은 레지스트리 서비스를 이용하여 드라이버에게 중요한 구성 데이터를 제공한다.

```
extern VxD_Desc_Block MYVXD_DDB;

BOOL __stdcall OnDeviceInit(PVMMCB hVM, DWORD refdata)
{
    // OnDeviceInit
    char regpath[256];
    char *p;

    HKEY hkey;
    DWORD port;
    DWORD cbdata = sizeof(port);

    _GetRegistryPath(&MYVXD_DDB, regpath, sizeof(regpath));

    p = regpath + strlen(regpath);
    while(p > regpath && p[-1] == ' ')
        --p; // 이름의 끝에 붙어있는 공백 문자를 제거한다.
    *p = 0;

    _RegOpenKey(HKEY_LOCAL_MACHINE, regpath, &hkey);
    _RegQueryValueEx(hkey, "Port", NULL, NULL, (PBYTE)&port, &cbdata);
    _RegCloseKey(hkey);
    return TRUE;
} // OnDeviceInit
```

_GetRegistryPath는 로드 할 MYVXD의 위치가 저장된 알맞은 레지스트리 경로를 알려준다. 여기서는 다음과 같이 될 것이다.

```
System/CurrentControlSet/Services/VxD/Myvxd
```

_GetRegistryPath에 의해 리턴 되는 경로는 DDB로부터 직접 구한 것이며, 8바이트의 디바이스 이름과 끝에 공백이 더해져 있다. _RegOpenKey를 호출하기 전에 경로문자열의 끝에 있는 공백을 제거해야 한다. 키를 연 후라면 _RegQueryValueEx를 호출하여 Port 같은 구성 설정 값을 알아낼 수 있다. 동작이 끝난 후에 _RegCloseKey를 호출해서 레지스트리 키를 닫아야 한다.

초기화하는 동안에는 JKEY_LOCAL_MACHINE의 서브키만을 액세스 할 수 있으며, Device_Init후는 레지스트리 전체를 액세스할 수 있다.

프로파일 서비스를 사용하는 것. 드라이버를 로드하기 위해 SYSTEM.INI 파일의 device= 문장을 사용하는 것은 좀 구식 방법이다. 이 구식 방법에서 구성 데이터를 기록하는 것은 전통적인 .INI 파일을 사용하는 것과 동일하다. 예를 들어,

SYSTEM.INI에 다음과 같은 항목들이 들어 있다면

```
[386enh]
device=c:\MyProd/myvxd.vxd
...
[myvxd]
port=1234h
```

이전에 이를 등록하는 코드는 대략 다음과 같을 것이다.

```
BOOL __stdcall OnDeviceInit(PVMMCB hVM, DWORD refdata)
{
    // OnDeviceInit
    DWORD port = Get_Profile_Hex_Int(-1, "myvxd", "port");
    return TRUE;
} // OnDeviceInit
```

위의 코드가 간단하기는 하지만, 튼튼하지도 않고 디바이스를 구성하는데 대한 표준 사용자 인터페이스에도 적합하지 않다. 그러나 윈도우즈 3.1에서는 잘 동작할 것이다.

8.2.3 종료 (Termination)

시스템이 시작할 때 스택틱 VxD에게 큰 변화가 일어난다고 설명했으며, 또한 스택틱 VxD는 윈도우즈 95가 리얼모드 MS-DOS로 되돌아 가기 위해 시스템을 종료하는 때에도 큰 변화가 일어난다. VMM은 종료 시점에 존재하는 모든 VxD에게 시스템 콘트를 메시지를 보낸다.

- System_Exit는 윈도우즈 95가 종료하려고 한다는 것을 알린다. 이때 프로세서는 여전히 보호모드에 남아 있으며, 여전히 시스템 VM에 있는 코드를 실행해도 안전하다. 대부분의 다른 VM은 이 시점에서 종료된다. 그러나 윈도우즈 KERNEL 모듈은 남아 있다.
- System_Exit2는 System_Exit 바로 뒤에 발생한다. 이 두 개의 메시지는 모두 같은 의미를 가지고 있다. 그러나 초기화 순서에 입각해서 제공되는 서비스에 의존하는 VxD를 위해 VMM은 초기화 순서의 역순으로 System_Exit2 메시지를 보낸다. 어떤 것이든 새로운 VxD에서는 System_Exit 대신 System_Exit2를 처리해야 한다.
- 모든 VxD들이 두 개의 시스템 종료 메시지를 처리한 후, VMM은 인터럽트를 디스에이블하고 각 VxD에게 Sys_Critical_Exit 메시지를 보낸다.
- 마지막으로 VMM은 각 VxD에게 Sys_Critical_Exit2 메시지를 보낸다. System_Exit2와 마찬가지로 Sys_Critical_Exit2는 초기화 순서의 역순으로 보내진다.

대부분의 VxD는 이러한 시스템 종료 메시지를 모두 처리할 필요는 없다. 그러나 만약 리얼모드 이미지를 사용하지 못하도록 한 것이 있다면 이들 메시지 중 하나는 처리해야 할 것이다. 예를 들어, V86MMGR 디바이스는 윈도우즈 95가 시작할 때 XMS 서버를 중단시키고 진입점을 덮어쓰게 되는데, 윈도우즈 95가 종료할 때 XMS가 정상적으로 동작하기 위해 진입점을 복구한다.

8.3 다이내믹 VxD (Dynamic VxDs)

윈도우즈 95는 디바이스 드라이버를 동적으로 로드하고 해제할 수 있다. 이렇게 하는 큰 목적은 동적으로 하드웨어를 재구성하도록 지원하기 위함이었지만, 특정 응용 프로그램만을 지원할 목적으로 드라이버를 로드하고 해제할 수도 있다.

디바이스 드라이버가 동적으로 로드 될 수 있도록 하기 위해서는 다음의 두 가지 것을 해야 한다.

- 윈도우즈 95용으로만 드라이버를 만들어야 한다. 윈도우즈 3.1 드라이버는 동적으로 로드 될 수 없다. (이것은 어셈블이나 컴파일에 WIN31COMPAT 전처리 매크로를 사용하지 말라는 것이다.)
- 드라이버 모듈정의 파일(.DEF)의 VxD 문장에 다음과 같이 DYNAMIC 키워드를 추가시킨다.

```
VXD MYVXD DYNAMIC
```

이 장의 나머지 부분에서는, 응용 프로그램이 드라이버를 동적으로 로드하는 방법과 윈도우즈 95가 동적으로 로드된 드라이버를 초기화하고 종료하는 방법을 설명한다.

(1) 삼십이 비트 응용 프로그램 (Thirty-Two-bit Applications)

32비트 응용 프로그램의 다이내믹 로드 인터페이스는 *CreateFile*이라고 하는 일반적인 Win32 API를 사용한다. 드라이버를 로드하기 위해서는 *CreateFile*에 다음과 같이 특이한 인자들을 사용한다.

```
HANDLE hDevice=CreateFile("\\\\.\\pathname", 0, 0,
NULL, 0, FILE_FLAG_DELETE_ON_CLOSE, NULL);
```

이러한 방법으로 *CreateFile*을 사용하는데 대부분의 인자들은 관계없기 때문에 0으로 놓을수 있다. 그러나 이 이상한 이름의 접두사 `\\.\\` (C에서는 백슬래시 두 개가 한 개로 되는 것을 상기하라)는 윈도우즈 95가 디바이스 드라이버를 로드하고 이 디바이스 핸들을 되돌려 달라는 것을 가리킨다. 물론 *CreateFile*은 일반적으로 파일을 읽고 쓰기 위해 연다. 필자는 드라이버를 로드 하는 이 인터페이스를 설계한 사람이 누구든지 간에 그를 화나게 하려고 하는 것은 아니지만, 필자는 이러한 방법이 범용 API에 과부하를 준다고 생각한다. (역자 주 : Win32에서는 많은 부분에서 이 *CreateFile*을 사용하고 있다. 예를 들면 직렬통신과 같은 것을 들 수 있다. 역자 생각으로는 이것이 UNIX의 형태를 따르려고 했다는 생각이 들기는 하지만 이것이 굳이 Win32에서만 아니고 도스에도 그런 흔적이 많다는 것을 기억하기 바란다) 다이내믹 VxD를 로드 하는 샘플 코드가 부록 CD의 /CHAP08/DYNLOAD-DDK와 /CHAP08/DYNLOAD-VTOOLS.D 디렉토리에 있다.

CreateFile 호출의 마지막 인자는 속성 및 플래그(attributes-and-flags) 인자이다. *FILE_FLAG_DELETE_ON_CLOSE*는 다음과 같이 *CloseHandle*을 호출하였을 때 윈도우즈 95가 자동으로 이 드라이버를 해제(언로드)하도록 하기 위함이다.

```
CloseHandle(hDevice);
```

일반적으로 *CreateFile*은 드라이버를 로드하지 못하면 *INVALID_HANDLE_VALUE* (값은 -1)을 리턴한다.

이러한 방법으로 드라이버를 동적으로 로드하기 위해서는 한가지 더 해주어야 하는 것이 있는데, 이것은 VxD에서 *W32_DEVICEIOCONTROL* 메시지를 처리해 주어야 한다는 것이다. *VWIN32 VxD*는 Win32 응용 프로그램에서 호출하는 *DeviceIoControl* 호출을 처리하는데 이 콘트롤 메시지를 사용한다. 또한 이 메시지는 응용 프로그램이 *CreateFile* 호출을 통해 드라이버가 동적으로 로드 되었을 때도 보내진다. 간단한 핸들러는 다음과 같을 것이다.

```
#include <vwin32.h>
DWORD __stdcall OnW32DeviceIoControl(PDIOPARAMETERS p)
{
    // OnW32DeviceIoControl
    switch(p->dwIoControlCode)
    {
        // 콘트롤 호출을 처리
        case DIOC_GETVERSION:
            return 0;
    }
    // 콘트롤 호출을 처리
    // OnW32DeviceIoControl
}
```

필자는 드라이버를 로드하기 위해 *CreateFile*을 수행하는 도중 왜 *W32_DEVICEIOCONTROL* 메시지를 보내는지 그 이유를 찾지 못했다. 윈도우즈의 많은 것들이 이 모양인 것처럼 이것도 바로 그렇다.

(2) 십육비트 응용 프로그램 (Sixteen-bit Applications)

16비트 응용 프로그램(리얼모드와 보호모드 모두다)은 모두다 가상 디바이스 드라이버를 동적으로 로드하는데 *VXD.LDR* 드라이버 API 호출을 사용한다. 이것은 32비트 응용 프로그램에서 했던 것과 아주 다르고 매우 복잡하다. 첫 번째 단계는 *INT 2Fh* 함수 *1684h* 인터페이스를 사용하여 API 진입점을 구하는 것이다. (이 중요한 인터페이스는 10장에서 좀더 포괄적으로 설명할 것이다) 그 코드는 다음과 같을 것이다.

```
include vmm.inc ; VXD.LDR_DEVICE_ID를 사용하기 위하여
```

```
...
```

```

vxdldr    dd    0                ; VXD LDR API의 진입점 주소
...
mov      ax, 1684h              ; 함수 1684h : VxD의 진입점을 구한다.
xor      di, di                 ; 진입점을 처음으로 구하므로 ES:DI를 지움
mov      es, di                 ; ...
mov      bx, VXD LDR_DEVICE_ID
int      2Fh                    ; 로더의 API 진입점을 구한다.
mov      ax, es                 ; 진입점을 찾았는가?
or       ax, di                 ; ...
jz       fail                   ; 만약 찾지 못하면 동적 로딩을 실패

mov      word ptr vxdldr, di
mov      word ptr vxdldr+2, es

```

위의 예제에서 VXD LDR의 API 어드레스를 구했고 이를 저장해 두었다. 이제 AX가 함수 코드를 가지도록 설정하고 가상 디바이스를 로드하고 해제하는데 이 진입점을 호출하는 것이다. 드라이버를 로드 하는데 VXD LDR.H나 VXD LDR.INC를 편입하고 다음과 같은 코드를 작성한다.

```

include vxdldr.inc             ; VXD LDR 서비스를 사용하기 위하여
...
mov      ax, VXD LDR_APIFUNC_LOADDEVICE ; 즉, 1
mov      dx, offset filename         ; DS:DX -> 파일이름 + 0
call     [vxdldr]                   ; 드라이버를 로드하도록 한다.
jc       fail                       ; 만약 로드하지 못하면 캐리 플래그가 설정된다.

```

이 샘플 코드에서 *filename*은 로드 하려고 하는 드라이버의 경로명을 가지고 있으며 널로 끝나는 문자열이다. 만약 드라이버 이름이 MYVXD.VXD라면 그 코드는 다음과 같을 것이다.

```
filename db 'myvxd.vxd', 0
```

드라이버를 해제하기 위한 코드는 다음과 같을 것이다.

```

mov      ax, VXD LDR_APIFUNC_UNLOADDEVICE ; 즉, 2
mov      dx, offset name                 ; DS:DX -> driver name + 0
call     [vxdldr]                       ; 드라이버를 해제하도록 한다.
jc       fail                           ; 만약 해제하지 못하면 캐리 플래그가 설정된다.

```

이 경우 *name*은 드라이버의 내부이름을 가지고 있으며 널로 끝나는 문자열이다. 드라이버 이름이 MYVXD라고 되어 있으면 그 코드는 다음과 같을 것이다.

```
namedb 'MYVXD', 0
```

로드 서비스와 해제(언로드) 서비스에 사용되는 이름이 **서로 다른 형식**이라는 것을 명심해야 한다. LOADDEVICE 함수는 경로명을 사용해야 하며, UNLOADDEVICE 함수는 VxD의 Declare_Virtual_Device 매크로에 있는 문자열인 디바이스 이름을 사용해야 한다.

8.4 다이내믹 VxD의 초기화와 종료

(Initializing and Terminating a Dynamic VxD)

윈도우즈 95는 동적으로 로드 된 디바이스 드라이버를 초기화하는데 Sys_Dynamic_Device_Init 메시지를 보낸다. 다이내믹하게 로드 된 드라이버를 해제하기 전에 Sys_Dynamic_Device_Exit 메시지를 보낸다.

윈도우즈 95는 다이내믹 드라이버가 로드 되기 오래 전에 초기화 세그먼트를 해제하고 인스턴스 데이터를 복사하기

때문에, 동적으로 로드된 드라이버는 다음과 같은 초기화용만의 서비스는 사용할 수 없다.

_Add_Gloabl_V86_Data_Area	Get_Next_Area
_AddFreePhysPage	Get_Next_Profile_String
_AddInstanceltem	Get_Profile_Boolean
_Allocate_Global_V86_Data_Area	Get_Profile_Decimal_int
_Allocate_Temp_V86_Data_Area	Get_Profile_Fixed_Point
_Free_Temp_V86_Data_Area	Get_Profile_Hex_Int
_GetGblRng0V86IntBase	Get_Profile_String
_SetFreePhysRegCalBk	GetDOSVectors
_SetLastV86Page	Locate_Byte_In_ROM
Allocate_PM_App_CB_Area	MMGR_SetNULPageAddr
Convert_Boolean_String	OpenFile
Convert_Decimal_String	PageFile_Init_File
Convert_Fixed_Point_String	Set_Physical_HMA_Alias
Convert_Hex_String	V86MMGR_NoUMBInitCalls
DOSMGR_BackFill_Allowed	V86MMGR_Set_Mapping_Info
DOSMGR_Enable_Indos_Polling	V86MMGR_SetAvailMapPgs
DOSMGR_Instance_Device	V86MMGR_SetLocalA20
Get_Name_Of_Ugly_TSR	VDMAD_Reserve_Buffer_Space

동적으로 로드된 드라이버는 이미 후크된 인터럽트는 후크하지 않도록 하며, 계속적으로 부적절한 효과를 가져올 수 있는 동작은 하지 않도록 해야 한다.

동적으로 로드된 디바이스의 권고 사항중의 예외 한가지는 *Allocate_V86_Call_Back*에 의한 V86 모드 콜백이나 *Allocate_PM_Call_Back*에 의한 보호모드 콜백과 관련된 것은 끝나기 전에 깨끗하게 마무리해야 한다는 것이다. 왜냐하면 이러한 것은 해제하는 서비스가 없기 때문이다. 만약 동적으로 로드될 때마다 콜백을 다시 할당한다면 이것은 리소스를 낭비하게 된다. 이러한 문제는 스택 코드와 스택 데이터 세그먼트를 사용하여 처리한다.

```

VxD_STATIC_DATA_SEG
mycallback dd 0
loaded dd 0
VxD_STATIC_DATA_ENDS

BeginProc myfunc, static
    cmp loaded, 1 ; 이전에 이미 로드된 VxD가 있는가?
    je @f ; 만약 있다면, 좋다!
    [스택 세그먼트를 빠져나가지 않도록 실패를 안전하게 한다.]
@@: jmp realcallback ; realcallback으로 간다
    ...
EndProc myfunction

BeginProc OnSysDynamicDeviceInit, init
    mov loaded, 1
    cmp mycallback, 0
    jne @f
    mov esi, offset32 myfunction
    VMMCall Allocate_PM_Call_Back
    mov mycallback, eax
    @@: ...
EndProc OnSysDynamicDeicelnit

BeginProc OnSysDynamicDeviceExit, locked
    mov loaded, 0

```

```
...
EndProc OnSysDynamicDeviceExit

BeginProc realcallback, locked

...
EndProc realcallback
```

(필자는 이 샘플에 어셈블리어를 사용했는데 이러한 콜백코드에 C를 사용하는 것은 목에 걸리 가치처럼 생각시다.)

마지막으로 강조할 것은 이 장에서도 말했던 것처럼, 다이내믹 VxD가 스택 VxD와는 그 처리에 있어서 정말 다르다는 것이다. 다이내믹 VxD는 스택 VxD가 받지 않는 두 개의 시스템 콘트롤 메시지(Sys_Dynamic_Device_Init와 Sys_Dynamic_Device_Exit)를 받는다. 앞에서 Init_Complete 메시지 이후에는 호출할 수 없는 서비스에 대하여 강조해서 설명한 것을 제외한다면, 스택 VxD에 넣을 수 있는 코드는 다이내믹 VxD에도 넣을 수 있으며, 그 반대도 마찬가지이다. 더하여 다이내믹 VxD는 로드 되는 중에서도 발생하는 모든 시스템 콘트롤 메시지를 받는다. 대부분의 다이내믹 VxD는 이러한 메시지를 무시하지만 원한다면 처리할 수도 있다.