

4장. 시스템 프로그래밍을 위한 인터페이스 (Interfaces for System Programming)

필자가 전에 사용해본 운영체제들은 모두 다 응용프로그램이 시스템 서비스를 요청하는데 사용하기 위한 인터페이스를 제공하고 있었다. 필자는 IBM에서 만든 OS/360을 아주 어릴 때부터 익혔다. 이것은 방 크기 만한 컴퓨터를 제어했는데, 이 컴퓨터는 60, 70년대 기술의 최고를 대표하는 컴퓨터였다. OS/360은 그 서비스 - 슈퍼바이저가 호출(SVC; Supervisor Call)하는 기계 명령에 단일 파이프라인을 가지고 있었는데, 그 파이프라인으로 요청 사항을 입력하기 위한 어셈블리어로 만들어진 매우 정교한 매크로 명령들이 있었다. 또한 인터페이스 사용법을 설명한 굉장히 완벽하고 정확한 문서도 제공되었다. 경험 있는 프로그래머는 제공된 문서에 따라 친숙하지 않은 인터페이스도 호출할 수 있었고, 또 결과 프로그램이 원하는 대로 동작하리라 적당히 확신할 수 있었다는 것이 과장이 아니다. 그리고 우연하게도 원하는 결과가 나오지 않으면 시스템 프로그래머는 프로그램 로직 매뉴얼이나 시스템에 대한 완벽한 프로그램 코드가 포함된 마이크로피이서를 찾을 수 있다. IBM의 운영체제의 하나인 IBM/370은 시스템 프로그래머가 로컬 사용자 집단의 입맛에 맞도록 쉽게 수정할 수 있는 기계어 소스 코드로 들어 왔지만, 정식적인 명령어 수정 방법도 함께 제공되었다.

독자 여러분! 정연하고 정확도 있는 OS/360 시스템 프로그래머의 생활과 혼돈 되고 어리둥절한 윈도우즈 95 시스템 프로그래머의 생활을 비교해 보라. 인텔의 INT 명령(몇 개의 패밀리가 있음)은 시스템 서비스를 요구 아래에 놓여 있기 때문에 그것을 요청하는 방법에 각자의 규칙이 있다. 예를 들어, 파일 시스템 서비스를 요청하기 위해 INT 21h를 어떻게 사용해야 하는지를 배우는 것은 DPMI(Dos Protected Mode Interface)의 INT 31h를 어떻게 사용하는지에 대한 해답을 주지 못한다. DPMI를 사용하지 않을 때(다시 말하면 리얼모드에서)에만 INT 21h를 사용할 수 있는 것이다. 보통의 32비트 윈도우즈 프로그램에서 이러한 방법을 사용하면 당신의 프로그램을 제대로 동작하지 않을 것이다. 그래서 INT 21h 서비스가 제대로 동작하지 않는 그런 상황에서 리얼모드 MS-DOS의 기능 수행을 강제적으로 시키는 일(예를 들어, MS-DOS의 다국어 지원 서비스의 어떤 부분을 사용하기 위해서는 DPMI의 변환기능을 사용하는 경우)이든 아니면 다른 방법으로 이를 수행(예를 들어, 리얼모드 메모리를 할당해야 하는 때)해야 하는 때에든 DPMI를 사용해야 한다.

다른 시스템 인터페이스에서 완전히 다른 부분이 많지 않다면, 이 상품에 대하여 새로 배우려는 신참 시스템 프로그래머를 제외하고는 이 문서에 대한 걱정은 없어질 것이다. 앞 문단에서 하던 예를 계속하면, INT 21h 인터페이스는 이제 적당히 자세하게 문서화되었지만, 더 부지런한 분석 엔지니어들(reverse engineers)은 여기에 대해서 더 자세히 분석하고 문서화 하고 있다. MS-DOS 프로그래밍을 배우기 위해 초보 프로그래머는 서드 파티들에 의해 만들어진 책을 사야할 것이다. 왜냐하면 이런 책은 경험적인 결과나 모험적으로 분해한 결과에 대해 이야기한 다소 우화적이고 일화적으로 꾸며져 있기 때문이다. 필자가 인용할 수 있는 많은 예 중의 하나로써, 프로그래머가 MS-DOS 메모리 블록의 연결(chain)을 검사하는 코드를 만들 필요가 있는 상황이 있을 수 있다. 이러한 상황은 TSR에서 벌써 자신이 로드 되었는지를 체크하는데 필요하다. MS-DOS 메모리 블록의 구조는 메모리 블록의 링크드 리스트에서 헤더를 찾기 위한 첫 번째 단계로 사용하는 INT 21h, 함수 52h에 의해 잘 알려져 있다. (Ralf Brown과 Jim Kyle의 "PC Interrupts", [Addison-Wesley, 1991], 페이지 8-45) 그러나 마이크로소프트는 여기에 대한 어떠한 정보도 공개하지 않았다. 호기심 많고 광적인 프로그래머들이 이것을 알아냈고 법적인 위험 부담을 가지면서도 이를 공개했다.

물론 운영체제의 어떤 부분은 철저히 문서화 되어있다. 이 책에서 필자는 가상 디바이스 드라이버(VxD) 만드는 방법에 대하여 이야기 할 것이다. 마이크로소프트 윈도우즈 95 DDK는 VxD에서 시스템 인터페이스 호출하는 방법을 자세하게 설명한다. 그러나 그 이유는 없다. 여기에는 부작용에 대한 설명도 없으며, 서로간에 밀착된 프로그램으로부터 다른 시스템으로 어떻게 집어넣는지에 대한 설명도 없다. 필자가 경험했던 OS/360에서와는 달리, 다른 말로 하면, 필자가 경험한 윈도우즈에서는 공식적인 문서만 읽어서는 VxD를 작성할 수 없었다. 많은 시간동안 필자는 프로그램이 동작하도록 하는데 씨름했지만, VxD는 한번 대충 훑어본다거나 마구 이리저리 사용하는 방법만으로 배우기는 너무 늦어 버렸다.

지금까지 필자는 이 책을 쓰게 된 이유를 넘지시 설명했다. 이제는 윈도우즈 95를 어떻게 확장하고 사용하는

지에 대해 탐험을 시작할 시간이 되었다. 여기서 필자는 시스템 프로그래밍 프로젝트에서 사용할 수 있는 다양한 인터페이스에 대하여 종합적으로 설명한다. 여기에는 VxD 서비스 API가 포함되어 있으며, 직렬 포트나 병렬 포트 혹은 네트워크 파일 시스템 등과 같이 특정한 종류의 디바이스를 위한 조직화된 API 세트도 몇 가지 포함되어 있다. 물론 여기에는 이전 윈도우에서 사용하던 세트도 포함되어 있는데, 바로 윈도우 95의 호환성을 유지하기 위한 DPMI 같은 것이 여기에 속한다.

4.1 가상 디바이스 드라이버 (Virtual Device Drivers)

윈도우 95의 핵심은 가상 머신 관리자(Virtual Machine Manager; VMM) 밑에서 운영되는 가상 디바이스 드라이버들의 집합체이다. 그것은 그 자체로 VxD이다. VxD는 각자 3가지 방식을 이용하여 통신하는데, 시스템 제어 메시지, 서비스 API 호출, 콜백 함수가 이것이다.

VxD는 리얼모드와 보호모드 응용 프로그램에 의해 이용되는 API 엔트리 포인트를 쉽게 익스포트 할 수 있다. API를 액세스하기 위하여 응용 프로그램은 INT 2Fh, 함수 1684h를 사용하여 함수의 엔트리 포인트를 구한다. 응용 프로그램이 이 엔트리 포인트를 호출했을 때, VMM은 VxD에 대한 제어권과 노선 제어권을 얻는다. VxD는 응용 프로그램의 요청에 대한 응답으로 가상 머신의 3순위 권한 레지스터(ring-three register, 소위 *client register structure*)의 이미지를 바꾼다. 이러한 메커니즘은 16비트 윈도우 프로그램, 16비트 도스 프로그램, 도스 확장자 하에서 실행되는 16비트/32비트 프로그램에서 사용할 수 있다. Win32 프로그램은 대신 VxD로부터의 서비스 요청에 대하여 *DeviceIoControl* 함수를 사용한다.

호출 가능한 모든 VxD 서비스 호출에 관한 문서가 윈도우 95 DDK에 포함되어 있으니 참고하기 바란다. 이 책의 대부분은 언제 어떻게 이러한 메커니즘을 사용하는지에 대해 좀더 자세하게 제공하며, 이 장에서는 이러한 것이 종합적으로 설명된다.

4.1.1 시스템 제어 메시지 (System Control Messages)

VMM은 모든 로드 된 VxD들에게 미리 정해진 순서에 따라 시스템 제어 메시지를 보낸다. 각 VxD는 이러한 메시지를 받기 위해 *디바이스 콘트롤 프로시저(device control procedure)*를 익스포트 한다. 이 프로시저는 메시지 코드를 검사한 다음 VxD안에 포함된 핸들러 기능을 처리하거나 메시지 처리를 성공적으로 처리했음을 가리키기 위해 플래그를 지운다. 대략 50개의 시스템 제어 메시지가 현재 정의 되어있다. 몇 개의 메시지는 윈도우 95의 시작과 종료의 다양한 단계나 가상 머신과 쓰레드의 생성과 파괴를 나타낸다. 다양한 이벤트에 관계된 다른 메시지들도 많은 디바이스 드라이버에게는 중요하다.

많은 VxD가 로드 되어 있고 각각의 VxD는 대부분의 시스템 콘트롤 메시지에서 짧은 순간의 여유 밖에 없으므로, 많은 VxD와 관련된 진짜 중요한 이벤트에 대해서만 제한된 메시지만을 사용하는 것이 중요하다. 그래서 시스템 제어 메시지에서는 응용 프로그램 수준의 WM_MOUSEMOVE와 같은 윈도우 메시지를 볼 수 없을 것이다.

윈도우 95에서는 단일의 VxD에만 관계된 시스템 콘트롤 메시지는 아주 조금 밖에 없다. 이 메시지들이 시스템 제어 메시지로 정의된 이유는 이 디바이스 콘트롤 프로시저가 VMM이 초기에 위치시킬 수 있는 유일한 엔트리 포인트이기 때문이다. 이제 어떻게 하여 VxD는 자신만의 메시지를 정의하며, 어떻게 하여 그 뜻을 이해하는 다른 VxD로 전송하는지에 대하여 알아보자.

디바이스 콘트롤 프로시저는 어디에 있는가? (Where's the Device Control Procedure?)

본문에서도 언급했듯이, VMM이 초기에 찾을 수 있는 VxD의 유일한 진입점이 시스템 콘트롤 메시지를 처리하는 디바이스 콘트롤 프로시저이다. 스테틱 *디바이스 디스크립션 블록(Device Description Block; DDB)*의 구조체는 이 진입점을 가리킨다.

(1) 시작과 종료 메시지 (Startup and Shutdown Messages)

VMM은 윈도우즈 95가 시작하거나 종료하는 동안의 중요한 이벤트에 대하여 아래의 표 4-1에 나타난 시스템 제어 메시지를 이용하여 통신한다.

메시지	설명
Sys_Critical_Init	VxD에서 초기화의 첫 번째 단계임을 알린다. 인터럽트를 디스에이블 된다.
Device_Init	VxD에게 초기화의 두 번째 단계임을 알린다. 인터럽트는 인에이블 된다.
Init_Complete	VxD에게 초기화의 세 번째 단계임을 알린다.
System_Exit	VxD에게 종료의 첫 번째 단계임을 알린다. 인터럽트는 여전히 인에이블이다. System_Exit와 같지만 초기화 순서와 반대로 이루어진다.
System_Exit2	
Sys_Critical_Init	VxD에게 종료의 두 번째 단계임을 알린다. 인터럽트는 디스에이블 된다.
Sys_Critical_Init2	Sys_Critical_Exit와 같으나 초기화 순서와 반대로 이루어진다.
Reboot_Processor	만약 VxD가 프로세서 리부팅 방법을 알고 있다면 그렇게 하라고 명령을 내리는 것이다.
Device_Reboot_Notify	VxD에게 VMM이 새로 시작(restart)하려 한다는 것을 알린다. 인터럽트는 인에이블 되어 있다.
Device_Reboot_Notify2	Device_Reboot_Notify와 같으나 초기화 순서와 반대로 이루어진다.
Crit_Reboot_Notify	VxD에게 VMM이 새로 시작(restart)하려 한다는 것을 알린다. 인터럽트는 디스에이블 되어 있다.
Crit_Reboot_Notify2	Crit_Reboot_Notify와 같으나 초기화 순서와 반대로 이루어진다.

표 4-1. 시작과 종료 시스템 콘트롤 메시지 (startup and shutdown system control messages)

여기에서 열거된 메시지의 중요사항을 설명하려고 한다.

- 3개의 초기화 메시지가 있다. Sys_Critical_Init는 보호모드로 전환한 후 바로 발생하지만 그전에 인터럽트는 디스에이블 된다. 인터럽트가 인에이블 된 후 Device_init 메시지가 발생하며, Device_Init 동안 모든 디바이스가 초기화할 기회를 가진 후 Init_Complete가 발생한다.
- 2개의 종료 메시지가 있다. System_Exit는 윈도우즈가 종료하려고 한다는 것을 알리게 된다. Sys_Critical_Exit는 프로세서가 멈추기 전에 인터럽트가 디스에이블 된 후 발생한다 윈도우즈 3.1과 이전 버전의 윈도우즈는 이 메시지를 보낸 후 프로세서를 리얼모드로 전환하며, 이렇게 하여 DOS는 다시 제어권을 얻게 되었다. 윈도우즈 95에서 DOS는 윈도우즈가 종료된 후 절대로 다시 제어권을 갖지 못한다. (파워 유저들은 만약 윈도우즈 95가 MS-DOS 프롬프트에서 시작하였다면, "이제 컴퓨터를 꺼도 안전합니다."라는 메시지를 화면에 출력할 때 mode c80이라는 명령을 타이프 치면 MS-DOS로 되돌아간다는 것을 알고 있을 것이다. 필자도 이것을 Spencer Katt의 칼럼을 읽고 알게 되었다. 물론 마이크로소프트는 많은 사용자들이 이 시점에서 컴퓨터를 끄리라 기대한다.)
- Device_Reboot_Notify와 Crit_Reboot_Notify 메시지는 각각 System_exit와 Sys_Critical_Exit와 유사하다. 이들 메시지는 윈도우즈 95가 프로세서를 재시작(리부팅) 하기 위해 종료할 때 발생한다.
- 종료 메시지는 2개의 변종을 가지고 있는데 예를 들자면 System_Exit와 System_Exit2와 같은 것들이다.

VMM은 VxD들에 대한 초기화 순서(initialization order)를 유지하며 이것은 로드된 VxD에게 시스템 콘트롤 메시지 보내는 순서다. VMM은 초기화 순서와는 거꾸로 "2"라는 변종을 보낸다. 이 방법은 윈도우즈 시작시에 작업된 것들을 지원하기 위함이다. 어떤 VxD는 이보다 앞서 초기화 된 VxD에 의해서 제공되는 서비스를 사용할 수 있도록 하기 위함이며, 그래서 이것은 종료 메시지에서도 계속될 수 있도록 하기 위함이다.

(2) 가상 머신 메시지 (Virtual Machine Messages)

표 4-2에 가상 머신의 생성과 소멸에 관계된 중요한 시스템 제어 메시지를 나타내었다.

메시지	설명
Create_VM	VxD에게 MV 생성의 첫 번째 단계임을 알린다. (시스템 VM과는 다른것)
VM_Critical_Init	VxD에게 VM 생성의 두 번째 단계임을 알린다. (시스템 VM과는 다른것)
Sys_VM_Init	VxD에게 시스템 VM의 초기화라를 것을 알린다.
VM_Init	VxD에게 VM 생성의 세 번째 단계임을 알린다.
Begin_PM_App	DPMI 클라이언트가 보호모드로 전환했음을 알린다.
Kernel32_Initialized	VxD에게 KERNEL32.DLL이 초기화 되었음을 알린다.
VM_Suspend	VxD에게 VM이 임시적으로 실행하지 못하게 되었음을 알린다. VM_Suspend와 같으나 초기화 순서와 반대로 전달된다.
VM_Suspend2	
VM_Resume	VxD에게 VM이 다시 실행하게 되었음을 알린다.
Kernel32_Shutdown	VxD에게 KERNEL32.DLL이 종료했음을 알린다.
End_PM_App	VxD에게 보호모드 DPMI 클라이언트가 종료했음을 알린다. End_PM_App와 같으나 초기화 순서와 반대로 전달된다.
End_PM_App2	
Query_Destroy	VxD에게 특정한 VM이 파괴되어도 좋을지를 묻는다.
Close_VM_Notify	VxD에게 VM이 Close_VM_Service 호출에 의해 종료하려 한다는 것을 알린다.
Close_VM_Notify2	Close_VM_Notify와 같으나 초기화 순서와 반대로 전달된다.
Sys_VM_Terminate	VxD에게 시스템 VM의 정상적인 종료를 알린다.
Sys_VM_Terminate2	Sys_VM_Terminate와 같으나 초기화 순서와 반대로 전달된다. VxD에게 VM의 정상적인 종료를 알린다. (시스템 VM과는 다른 것)
VM_Terminate	VM_Terminate와 같으나 초기화 순서와 반대로 전달된다.
VM_Terminate2	
VM_Not_Executable	VxD에게 VM 파괴의 next-to-last 상태임을 알린다.
VM_Not_Executable2	VM_Not_Executable과 같으나 초기화 순서와 반대로 전달된다.
Destroy_VM	VxD에게 VM 파괴의 제일 마지막 상황을 알린다.
Destroy_VM2	Destroy_VM과 같으나 초기화 순서와 반대로 전달된다.

표4.2. 가상 머신 제어 메시지(Virtual machine control messages)

여기에서 열거된 메시지의 중요사항을 설명하려고 한다.

- 윈도우즈 체제의 특권 위치를 나타내는 것으로, 시스템 VM은 자체적인 초기화와 종료 메시지가 있다.
- 새로운 가상 머신을 생성할 때 윈도우즈는 3개의 콘트롤 메시지를 보내는데 Create_VM, VM_Critical_Init, VM_Init가 그것이다. 이들은 시스템 VM이 완성된 후(Sys_VM_Init 콘트롤 메시지)에 이루어 지는데 왜냐하면, 시스템 VM을 생성하는 것은 다른 VxD를 호출하기 전에 VMM이 자동으로 행하는 것중의 하나이기 때문이다.
- 만약 VM이 정상적으로 종료되었다면, 윈도우즈는 VM_Terminate 메시지를 보낸다. VxD는 Nuke_VM 서비스나 Crash_Cur_VM 서비스 같은 것을 호출해서 정상적인 방법으로 가상 머신을 파괴할 수 있다. VM이 어떻게 종료하든지 그 종료에 VM_Not_Executable [sic]와 Destroy_VM 메시지를 동반한다. (역자 주 : [sic]는 틀린 원문을 그대로 옮길 때 사용하는 것으로 DDK 문서에는 VM_Not_Executable이라 설명되었겠지만 맞는 철자라면 아니라 철자 VM_Not_Executable이 될 것이다.)
- 앞에서 시스템 시작과 종료 메시지에서 설명한 바와 마찬가지로 종료 메시지에는 2개의 변종이 있다. VMM은 각 메시지에 있어 보통의 초기화 순서와는 반대로 "2" 변종을 보낸다.

(3) 다른 시스템 콘트롤 메시지들 (Other System Control Messages)

다음의 표 4-3은 나머지의 시스템 콘트롤 메시지들을 나타내었다.

메 시 지	설 명
Set_Device_Focus	VxD에게 한 개 혹은 여러개의 디바이스의 소유권을 가질 것을 VxD에게 지시한다.
Begin_Message_Mode	SHELL이 메시지를 디스플레이 하려고 한다는 것을 VxD에게 알린다.
End_Message_Mode End_Message_Mode2	SHELL이 메시지를 디스플레이 했음을 알린다. End_Message_Mode와 같으나, 메시지는 초기화 순서와는 반대로 전달된다.
Debug_Query	VxD에게 디버깅 출력 발생을 지시한다.
Power_Event	VxD에게 시스템의 전기 파워 전환 처리를 지시한다.
Sys_Dynamic_Device_Init	VxD에게ダイナ믹하게 로드된 VxD를 초기화 할 것을 지시 한다.
Sys_Dynamic_DeviceExit	VxD에게 다이내믹하게 로드된 VxD를 언로드 할것을 준비하도록 지시한다.
Create_Thread	VxD에게 새로운 쓰레드 생성의 첫번째 단계임을 알린다.
Thread_Init	VxD에게 새로운 쓰레드 생성의 두번째 단계임을 알린다.
Terminate_Thread	쓰레드 파괴의 첫번째 단계임을 VxD에게 알린다.
Thread_Not_Executable	쓰레드 파괴의 두번째 단계임을 VxD에게 알린다.
Destroy_Thread	쓰레드 파괴의 세번째 단계임을 VxD에게 알린다.
PNP_New_Devnode	새로운 디바이스 노드가 생성되었음을 VxD에게 알린다.
W32_DeviceIoControl	Win32 프로그램이 DeviceIoControl 호출을 냈음을 VxD에게 알린다.
Get_Contention_Handler	VCOMM이 contention handler의 주소를 원함을 VxD에게 알린다.

표 4-3. 다른 시스템 제어 메시지들(Other System Control Messages)

이 메시지들의 중요한 사항은 아래에 기술되었다.

- Set_Device_Focus는 윈도우즈가 마우스 같은 디바이스의 소유권을 바꾸는 기본적인 방법이다. 윈도우즈가 명시된 디바이스에 대하여 이 메시지를 발행하면, 이 경우 만약 메시지가 자신의 디바이스를 포함하고 있다면 각 VxD는 메시지를 받아볼 수 있다. 만약 모든 디바이스에 대하여 이 메시지를 발생시키면, 이 경우 복수의 VxD등이 답을 보내올 것이다.
- Begin_Message_Mode와 End_Message_Mode는 각각 SHELL 디바이스에 의해서 시스템 모달 메시지를 나타내는 처음과 끝을 나타낸다. 흔히 이 메시지는 “파란 메시지 박스”를 나타낸다. 이 메시지가 보여지는 동안에 메시지 인터페이스나 사용자에 대한 응답 같은 것 등에 대해서 디바이스 드라이버는 아무 것도 할 수 없다.

- `Debug_Query`는 디버거에서 프로그래머가 특정한 명령을 사용했을 때 발생한다. 예를 들어, `.V86MMGR` 명령을 사용하면 `V86MMGR`에게 이 메시지를 보낸다. 이 메시지는 코드를 디버깅하는데 유용하다.
- `Power_Event`는 컴퓨터에 공급되는 전기 파워 공급이 바뀔 것이라는 것을 이에 흥미를 보이는 `VxD`에게 알린다. 필자의 경험으로는 비행기에서 랩탑 컴퓨터에게 이 파워 전환은 치명적이었다.
- `Sys_Dynamic_Device_Init`와 `Sys_Dynamic_Device_Exit`는 윈도우즈에서 다이내믹하게 로드된 `VxD`의 초기화와 종료에 사용된다.
- 5개의 쓰레드 라이프 타임 메시지, 즉 `Create_Thread`, `Thread_Init`, `Treminate_Thread`, `Thread_Not_Executable [sic]`, `Destroy_Thread`는 Win32 쓰레드의 생존과 관계 있다.
- `PNP_New_Devnode`는 `VxD`에게 구성 관리자(Configuration Manager)가 물리적인 디바이스를 표현하는 새로운 디바이스 노드와 컨트롤 블록을 생성했다는 것을 알린다.
- Win32 프로그램은 디바이스 드라이버 핸들을 얻기 위해 `CreateFile` 함수를 사용할 수 있으며, 이 핸들은 `DeviceIoControl`의 인자로 제공될 수 있다. `DeviceIoControl` 호출은 `W32_DeviceIoControl` 컨트롤 메시지로 전환된다.
- 직렬 포트와 병렬 포트를 관리하는 `VCOMM VxD`는 디바이스 쟁탈을 처리하기 위한 콜백 함수의 어드레스를 얻기 위해 `Get_Contention_Handler`를 사용한다.

(역자주 : 위의 내용에서 “`Thread_Not_Executable [sic]`”란 부분이 있을 것이다. 여기서 `[sic]`라고 적은 이유는 원래의 영문철자에서 `Thread_Not_Executebale`이 아니라 `Thread_Not_Executable`이라고 해야 맞을 것이다)

(4) 자체적인 컨트롤 메시지 (Private Control Messages)

함께 작업하도록 설계된 `VxD`는 다른 `VxD`에게 자체적인 메시지를 보내기 위해 `Direct_Sys_Control` API 호출을 사용할 수 있다. 이러한 프로토콜을 위한 시스템 제어 메시지의 영역은 `70000000h ~ 7FFFFFFh` (`BEGIN_RESERVED_PRIVATE_SYSTEM_CONTROL ~ END_RESERVED_PRIVATE_SYSTEM_CONTROL`)이다. `VMM`은 이러한 자체적인 메시지를 위한 어떤 레지스트리도 지원하지 않는 대신 이러한 방법의 통신 상황하에서는 그 메시지 코드가 뜻하는 것이 무엇인가를 `VxD`는 알고 있어야 한다.

4.1.2 VxD 서비스 API (The VxD Service API)

`VxD`는 `VxD` 서비스 API를 이용하여 `VMM`과 다른 `VxD`와 통신한다. 이러한 API는 `VxD`와 다이내믹하게 링크 되어있으며 소프트웨어 인터럽트 `20h`와 `VxD` 서비스 류 세트를 사용한다. 이 서비스 류는 특정한 `VxD`와 그 `VxD`에 포함된 서비스 기능을 유일하게 인식한다. `VMM`은 많은 `VxD` 클라이언트들에게 일반적인 유틸리티를 대략 400개정도 공급하며, 다른 `VxD`는 특정한 목적을 위한 좀더 제한된 서비스 API를 익스포트한다.

서비스 엔트리 포인트를 익스포트 한 각 `VxD`는 유일한 16비트 인식정수 (Identifying Integer, `VxD ID`)와 브랜치 테이블(Branch Table)을 가지고 있다. 브랜치 테이블의 주소는 드라이버의 디바이스 디스크립션 블록에 나타난다. 브랜치 테이블의 기재 사항은 외부에서 호출 가능한 서비스 함수들이다. 드라이버 제작자는 `VxD_Define_Hot_Key`와 같은 니모닉과 `000D0001h`와 같은 값을 가지고 있는 32비트 서비스 테이블 시리지가 정의된 헤더 화일을 공급한다. 각 상수의 상위 반은 `VxD ID`이며 하위 반은 특별한 서비스를 위한 브랜치 테이블의 진입점 인덱스이다. (그림 4-1 참조) 그래서 `000D0001h`는 가상 키보드 디바이스(Virtual Keyboard Device; `VKD`)에 포함된 첫 번째 서비스(`Define_Hot_Key`)를 나타내며 그의 ID는 `000Dh`이다.

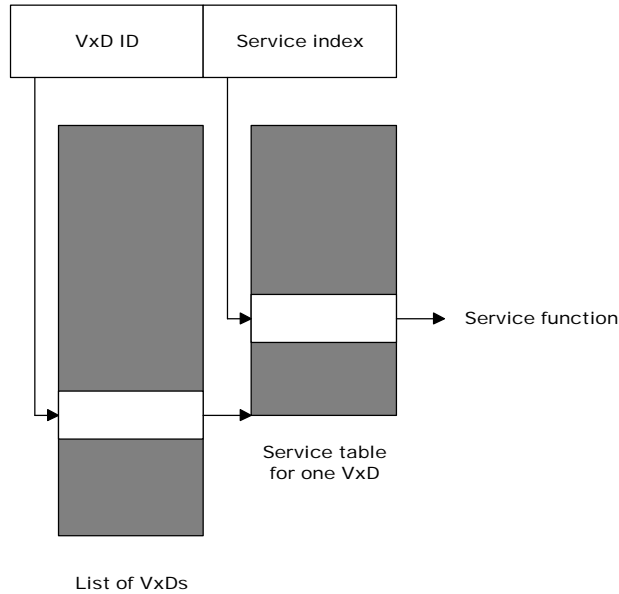


그림 4-1. VxD 서비스 인식자.

서비스 함수 호출을 원하는 VxD는 INT 20h 명령을 발생시키는데, 이 명령 뒤에는 여기에 알맞는 적절한 서비스 테이블의 상수에 해당하는 인라인 데이터 값을 정의한 메모리가 와야 한다. VMM은 그 서비스 함수를 포함한 VxD에 위치시키기 위해 상수의 ID 부분을 사용한다. 그리고 상수의 인덱스 부분은 브랜치 테이블에 위치시키기 위해 사용한다. VMM은 서비스 함수를 호출하고 나서 호출한 VxD의 인라인 데이터 값 뒤로 제어권을 넘겨주기 위한 조정 작업을 한다. 어셈블리어로 만든 드라이버는 자동적으로 INT 20h 명령과 뒤에 나오는 데이터 값을 생성하기 위해 VMCall과 VxDCall 매크로를 사용한다. 예를 들어 다음과 같다.

```
include vkd.inc
...
mov     al, scancode
mov     ah, scantype
mov     ebx, shiftstate
mov     cl, operationflags
mov     esi, offset32 callback
mov     edi, delay
VxDCall VKD_Define_Hot_Key
```

명백히 이것은 안에서 VxD의 서비스 테이블과 인덱스를 살펴보는 것에 비하면 너무 비싼 댓가다. 그래서 VMM은 그들을 처음 만났을때 서비스 링크를 슬쩍 바꿔 놓는다. 즉, VxDCall 매크로는 초기에 다음과 같이 나타낼 수 있다.

```
CD 20          INT 20h
01 00 0D 00   DD  VKD_Define_Hot_Key
```

이 명령을 최초로 실행한 뒤 VMM은 이 6 바이트를 서비스 직접 진입점을 사용한 간접 CALL 명령으로 바꾼다. 이것은 아래와 같다.(x는 서비스의 진입 주소를 나타낸다.)

```
FF 15 xx xx xx xx   CALL [$VKD_Define_Hot_Key]
```

이렇게 링크를 바꾸는 것은 원래 코드보다 절대적으로 빠르다. 이것은 어떤것에 대해서 매우 효과적인데, 예

를 들면 VMM은 *Get_Cur_VM_Handle* 서비스 호출을 현재 VM 핸들이 존재하는 메모리의 위치를 직접 참조한 MOV 명령으로 바꾼다. MOV 명령은 INT 20h 명령과 서비스 식별자를 함께 사용한것 처럼 6 바이트를 점유하기 때문이다.

이러한 함수 연결 방법은 몇 가지 이점을 제공한다. 무엇보다도, 서비스 브랜치 테이블에 많은 엔트리를 저장할 수 있고, 다른 VxD에 의한 호출을 가로챌 수 있다는 것이다. 이러한 역할을 하는 *Hook_Device_Service*라고 하는 VMM 서비스가 있다. 또 다른 이점은 다른 VxD의 엔트리 포인트를 참조하는 VxD를 다이내믹하게 합칠 수 있다. 이것은 윈도우즈 응용프로그램의 DLL의 다이내믹 연결과 비슷하다. 그러나 이런 연결 방법은 호출자가 실제적으로 VxD 호출을 실행할 때 발생하기 때문에 단순히 존재하지 않는 심볼을 참조하는 코드가 발생하는데 대한 대책이 없다. 이것은 로드 할 수 없는 VxD에 대한 걱정 없이 런타임에서 발견한다는 조건을 바탕으로 하여 어떤 서비스를 조건적으로 호출하는 VxD를 만들 수 있는 것이다. 윈도우즈 DLL과 닮은 다이내믹한 연결을 이루기 위해서는 *GetProcAddress*를 사용하여 스스로의 참조를 열심히 합쳐야 한다.

이러한 서비스 연결 구조의 마지막 이점은 어떤 서비스의 호출에 의한 어떤 특정한 드라이버가 로드 되었는지를 쉽게 알 수 있다. 전통적으로 서비스를 익스포트한 모든 VxD는 서비스 인덱스 0에 *Get_Version* 서비스를 제공하는데, 이것을 호출하면 된다. 만약 호출이 실패하면 드라이버는 로드 되지 않는 것이다.

```
VxDCall  FOO_Get_Version
jc       L_not_loaded
...
```

이러한 목적으로 *Get_Version* 서비스를 호출하는 이유는 이것은 정상적이라면 실패할 수가 없기 때문이다.. 만약 실패했다면 VxD가 없어서 서비스 호출에 응답할 수가 없다는 뜻이 된다.

다이내믹한 VxD에 대한 서비스 (Services in Dynamic VxDs)

Get_Version 테스트는 정적으로 로드된 VxD에 대해서는 잘 작동한다. 이 VxD가 이 요청에 의하여 처음 로드 되었을수도 있고, 전혀 그렇지 않을 수도 있다. 비록 다이내믹하게 로드된 VxD에서 서비스들을 익스포트 하지 않았다고 하더라도 그렇게 동작하는 것이 가능하다. 그러면 누군가가 그것을 호출하려 했던 제일 처음 *Get_Version* 서비스가 존재하지 않았을 시나리오가 있을 수 있다. 설사 나중에 VxD가 나타난다고 하더라도, 호출자는 항상 "Nobody home"이라는 같은 대답만 얻게 될 것이다. 왜냐하면 VMM이 벌써 다이내믹 연결을 슬쩍 바꿔치기 해 놓았기 때문에 이와는 전혀 상관없이 되기 때문이다. 이러한 낭패를 피하기 위해, 다이내믹하게 로드된 디바이스의 존재를 묻는 *Get_DDB* 서비스를 사용하라. 그리고, 그것과 통신하기 위하여 *Direct_Sys_Control* 서비스 및 이와 통신하기 위한 몇 종류의 교차등록(cross-registration) 프로토콜을 사용하라.

4.1.3 콜백 함수(Callback Functions)

VxD는 또한 콜백함수를 이용하여 통신한다. 여기서 말하는 "콜백(callback)"은 한 VxD에 있는 프로시저로서, 이 VxD에 의해서 제공되는 포인트를 통해 다른 VxD가 호출할 수 있는 프로시저이다. 전형적인 시나리오에서, VMM은 *Sys_Dynamic_Device_Init* 같은 시스템 콘트롤 메시지를 처리하기 위해 VxD의 디바이스 콘트롤 프로시저를 호출한다. VxD의 메시지 핸들러는 콜백 함수를 등록하기 위해 *_VCOMM_Register_Port_Driver* 같은 VxD 서비스 호출을 사용하여 다른 중앙 서비스 공급자(central service provider)에 등록한다. 그 이후부터 중앙 공급자(central provider)는 다른 활동을 수행하기 위해 등록된 함수를 호출한다.

다른 전형적인 시나리오에서, VxD는 시스템 콘트롤 메시지를 처리하는 동안 인터럽트를 폭킹(인터럽트를 위한 새로운 핸들러 주소를 설치하는 것)한다. VMM이나 다른 제 1수준의 인터럽트 핸들러는 폭크 프로시저를 호출한다. 그래서 인터럽트를 처리한다.

4.2 윈32 API (The WIN32 API)

(역자 주 : 이 절을 읽기 전에 이 절은 VxD에 관한 API가 아니라 Win32 응용 프로그램의 API에 관한 설명임을 기억하라) WIN32 API는 윈도우 NT에서 실행되는 32비트 프로그램을 위해 만들어졌다. (윈도우 95에서 실행되는 응용 프로그램도 손쉽게 사용 가능하다) 윈도우 NT에서 그렇게 하듯이, WIN32 API를 이용하여 32비트 프로그램을 생성한다. 32비트 윈도우 응용프로그램과 DLL, 표준 커맨드 라인 작업과 같은 32비트 콘솔(console) 응용 프로그램도 만들 수 있다. 몇 개 부류의 WIN32 API는 시스템 프로그래밍에 유용하며, 여기서 종합적으로 살펴보자.

4.2.1 프로세스와 스레드 관리 (Process and Thread Mangement)

윈도우 95는 WIN32 프로그램에서 윈도우 NT와 같은 프로세스와 스레드 모델을 지원한다. 이러한 모델에서, 각 응용 프로그램은 자체적으로 프로세스를 가지고 있는데 이 프로세스는 한 개 이상의 실행 스레드를 가지고 있다. 시스템의 모든 스레드들은 프로세서 시간에 대해서 서로 경쟁하는데, 이는 적절한 할당을 위해 만든 우선권과 다른 알고리즘 인자를 바탕으로 한다. 각 프로세스는 각자 별개의 메모리 컨텍스트(memory context)를 소유하고 있는데, 이 메모리 컨텍스트는 선형 주소의 00400000h ~ 80000000h에 위치한다. 한 프로세스에 속한 스레드들은 이 선형 어드레스에 의해 맵핑된 물리 메모리를 서로 공유하지만, 다른 프로세스에 있는 스레드는 이 영역을 액세스 할 수 없다.

새로운 스레드는 *CreateThread*를 호출해서 생성한다. 이 *CreateThread*의 인자중 하나는 이 스레드가 실행할 함수의 어드레스이며, 이 스레드는 이 함수가 리턴을 하거나 *ExitThread*를 호출했을 때 종료한다. 또다른 인자는 이 스레드가 정지된 채로 생성되는지 실행 상태로 생성되는지를 나타내는데, 이 정지된 스레드를 해제하고 계속 실행하기 위해서는 *ResumeThread*를 호출한다. *CreateThread*는 스레드의 핸들을 리턴하며, 이 핸들은 다양한 용도로 사용할 수 있다. 예를들어 강제적으로 스레드를 종료시키기 위해 *TerminateThread*를 호출한다든지, 종료 코드를 얻기 위해 *GetExitCodeThread*를 사용한다든지, 이 스레드를 기다리게 하기 위해 *WaitForSingleObject*와 같은 동기화 프리미티브를 사용한다든지 하는 등등이다.

한 프로세스에 속한 모든 스레드는 프로그램 모듈에 속한 스택틱한 저장 공간을 포함하여 같은 가상 메모리 컨텍스트를 공유한다. 어떤 스레드에 속한 한 함수에서 다른 스레드가 소유한 데이터와는 다른 영속적인 데이터를 생성하기 위해서는 *스레드 로컬 스토리지(thread local storage)* 메커니즘을 사용한다. 마이크로소프트 비주얼 C++을 사용한다면, *_declspec(thread)* 지시어를 사용하여 스레드 로컬 오브젝트를 선언할 수 있다. 실행 시 마스터 스레드는 그 자신이나 다른 스레드에서 사용하는 *TLS* 인덱스를 할당하기 위해 *TlsAlloc*를 사용한다. 보통은 *TlsGetValue*와 *TlsSetValue*를 사용하는 서브루틴에서 액세스 가능하게 하기 위해 이 인덱스를 전역 변수로 만들어 저장한다.

가상 머신 관리자(Virtual Machine Manager)는 소위 "2차 스케줄러(secondary scheduler)"라 불리는 VxD를 가지고 있는데, 이 VxD는 Win32 스레드 우선권 모델을 실행한다. 이 스케줄러는 "1차 스케줄러(primary scheduler)" 클라이언트 중의 하나인데, 1차 스케줄러는 순전히 우선권을 바탕으로 하여 동작한다. *GetThreadPriority*와 *SetThreadPriority*를 호출하여 기본 우선권 값을 제어할 수 있다. 2차 스케줄러는 1차 스케줄러의 선택에 영향을 미치기 위해 이 기본 우선권에 "우선권 부양(priority boosts)"을 적용한다. 또한 다른 VxD도 스레드의 부양 값을 더하거나 빼는데, 이 부양 값은 보통 2차 스케줄러가 사용하는 값보다 숫자적으로 매우 크다. 따라서 Win32 API를 호출해서 설정한 이 우선권 값은 스레드가 얻는 실행 시간의 한 인자일 뿐이다.

프로세스와 스레드에 대하여 좀더 자세히 알고 싶다면 "Microsoft Win32 Programmer's Reference, Vol. 2 (Microsoft Press, 1993)"의 43장이거나 이 책의 6장에 보면 스레드 스케줄러에 대하여 좀더 자세히 설명되어 있다.

스레드의 동기화 (Thread Synchronization)

윈도우 95는 스레드의 동작 순서를 조정하기 위한 몇 종류의 동기화 오브젝트를 사용하는데, 뮤텍스 오브젝트(mutex object), 세마포어 오브젝트(semaphore object), 이벤트 오브젝트(event object)가 그것이다. 이들 오브젝트를 생성하고 핸들을 얻기 위해 오브젝트마다의 특정 API를 사용하며, *CloseHandle*을 호출하여 이 객체를 파괴한다. 또한 한 개 또는 다수의 동기화 오브젝트가 "signalled" 상태에 도달할 때까지 스레드를 기다리게 하기 위하여

WaitForSingleObject, *WaitForSingleObjectEx*, *WaitForMultipleObjects*, *WaitForMultipleObjectsEx*를 사용한다. (또한 이러한 호출에 다른 타입의 핸들을 사용할 수 있다. 예를 들어, 어떤 쓰레드 핸들은 이와 관련된 쓰레드가 제거 되었을 때 "signalled"로 간주된다)

뮤텍스 오브젝트(mutex는 "상호 배제인 mutual exclusion"을 의미함)는 한 시점에 한 쓰레드에만 소유될 수 있으며, *CreateMutex*를 사용해 뮤텍스를 생성할 수 있다. 어떤 쓰레드가 뮤텍스를 소유하고 있지 않을 때 "signalled"이고, 소유하고 있을 때 "unsignalled"로 간주된다. 그래서 상호배제 구간에 진입하기 위해 위해서 쓰레드는 앞에서 설명한 대기 프리미티브중의 하나를 사용한다. 대기 프리미티브가 정상적으로 완료되었다는 것은, 뮤텍스는 대기하던 쓰레드에 소유되고 뮤텍스상에 대기하던 다른 쓰레드는 막히게 된다는 것을 의미한다. 상호 배제 구간을 빠져나가서 뮤텍스를 신호 받은 상태에게 리턴하기 위해서 쓰레드는 *ReleaseMutex*를 호출한다. 그러면 뮤텍스를 기다리던 쓰레드는 제어권과 실행권을 얻는 기회를 가지게 될 것이다.

세마포어 오브젝트는 근본적으로 동기화 의미와 관계된 양의 정수(unsigned integer)이다. 세마포어는 그 카운터가 0보다 클 때 "signalled"이며, 카운터가 0일 때 "unsignalled"이다. *CreateSemaphore*를 호출해서 세마포어를 생성한다. 이전에서 설명한 대기 프리미티브중의 하나를 호출해서 세마포어가 0이 아닌 카운트를 가지기를 기다린다. 어떤 쓰레드든지 세마포어의 카운터를 증가시키기 위해 *ReleaseSemaphore*를 호출할 수 있다. (세마포어 상에서 성공적으로 기다린 단 하나의 것이 아니라). 이러한 특징은 여러 동시 사용자를 지원하는 오브젝트가 최대 리미트의 지배를 받도록 하는 동기화된 액세스에 세마포어를 사용할 수 있도록 한다.

이벤트 오브젝트는 간단한 문(gate)이라 볼 수 있으며, *CreateEvent*를 호출해서 이벤트를 생성한다. 이벤트가 신호 받은 상태에 도달하기를 기다리기 위해 앞에서 언급한 프리미티브들을 사용한다. 이벤트가 "signalled"로 되거나 "unsignalled"로 되거나, 얼마나 길건 간에 *SetEvent*를 사용했는지 *PlusEvent*를 사용했는지, 그리고 이벤트가 수동 리셋속성 인지 자동리셋 속성인지에 달려있다. *CreateEvent*의 인자는 수동 리셋 속성인지 자동 리셋 속성인지를 나타낸다. *SetEvent* 호출은 "signalled" 상태로 수동 리셋 이벤트로 만들어서, 무엇인가 *ResetEvent*를 호출할 때까지 머물도록 한다. 그러면, 수동 리셋 이벤트가 신호 받은 상태인 동안 기다린 쓰레드는 즉시 해제된다. 그러나 한 쓰레드가 해제되자마자 "unsigned" 상태로 되돌아간다. (어떤 쓰레드가 벌써 기다리고 있다면 해제는 즉시 일어난다.) 현재 이벤트 상에 대기하던 쓰레드가 없다면 *PlusEvent*는 아무런 영향이 없다. *PlusEvent* 호출은 현재 대기 중인 모든 수동 리셋 이벤트를 해제하기 위함이며 자동 리셋 이벤트는 단 한 개만이 쓰레드를 해제하기 위해 호출한다. 어떤 경우이든 간에 *PlusEvent*가 완료 되었을 때 이벤트는 아직도 "unsigned"이다.

확장된 형식의 대기 프리미티브들(*WaitForSingleObjectEx*와 *WaitForMultipleObjectsEx*)은 경보 가능한 대기상태(alerable wait state)를 호출하는 것을 허용한다. 윈도우즈 NT에서는, 같은 쓰레드에서 비동기 I/O 수행 루틴 호출하는 시스템을 허용하기 위해 경보 가능한 대기 상태를 사용한다. 윈도우즈 95에서는 VxD에 의해 스케줄 된 비동기 프로시저 호출을 기다리는데 경보 가능한 대기 상태를 사용한다.

윈도우즈 95는 Win32 이벤트 오브젝트를 위한 0순위 권한 핸들을 생성하는 *OpenVxDHandle*라는 함수를 수행한다. 우리는 *DeviceControl*이란 함수를 통해 이 핸들을 VxD에 넘겨준다. 그러면 VxD는 이 핸들을 이용하여 Win32함수와 쌍으로 대응되어 있는 비슷한 함수들을 사용할 수 있다. 예를 들면, VxD는 3순위 권한의 *SetEvent*와 유사한 것을 이루기 위해 *_VWIN32_SetWin32Event*를 사용한다.

"*Microsoft Win32 Programmer's Reference, Vol2*"의 44장과 이 책의 9장, 10장에서 좀더 자세한 동기화에 대하여 찾을 수 있다. 여기에는 VxD의 0순위 권한 동기화 호출과 비동기 프로시저의 호출에 관한 사용법에 대하여 기술하고 있다.

4.2.2 시스템 레지스트리(The System Registry)

윈도우즈 95 레지스트리는 키들과 값들로 이루어진 전용 데이터베이스이다. 모든 키는 무한정의 서브키를 가질 수 있다. 또한 모든 키는 한 개의 이름 없는 값과 어떤 개수에 관계없는 이름 있는 값을 가질 수 있다. 시스템 요소나 응용 프로그램과 같은 것은 설치된 하드웨어나 소프트웨어에 대하여 영속적인 데이터를 저장하기 위해 레지스트리를 사용한다.

시스템 레지스트리의 두 가지 최상위 레벨 브랜치는 시스템 프로그램에 특별한 중요성을 가지고 있다.

HKEY_LOCAL_MACHINE(이하 HKLM) 브랜치와 HKEY_DYN_DATA 브랜치가 그것이다. 윈도우즈 95 구성관리자 VxD는 시스템 구성과 디바이스 드라이버를 선택하는데 HKLM 엔트리를 사용한다. 또한 구성 관리자는 HKEY_DYN_DATA 브랜치에 있는 동적인 키들에 실행 시간 정보를 기록한다.

Win32 프로그램은 레지스트리 액세스하거나 고치기 위해 표준 Win32 API를 사용할 수 있다. 흔히, 최상위 수준 키(HKLM 같은것)중 하나에서 이름을 가지고 있는 서브키에 대한 핸들을 열기 위해 *RegOpenKey*를 호출하며 새로운 서브키를 만들기 위해 *RegCreateKey*를 호출한다. 어떤 키에 대하여, 이름을 가지지 않은 단 한개의값을 물어보기 위해 *RegQueryValue*를 호출하며, 이름을 가지고 있는 값들은 검사하기 위해 *RegQueryValueEx*를 호출한다. 또한 *RegSetValue*와 *RegSetValueEx* 호출을 통해서 이 값들을 수정할 수도 있다. 키에 대하여 어떤 서브 키와 값들을 가지고 있는지를 알기 위해서는 각각 *RegEnumKey*와 *RegEnumValue*를 호출한다.

Win16 프로그램도 윈도우즈 95 버전의 16 비트 windows.h 헤더 파일에 선언된 함수들을 사용하여 레지스트리를 액세스 할 수 있다. (DDK나 어쩌면 16 비트 컴파일러 회사에서 제공된 것에서) 레지스트리에 대한 좀더 자세한 것은 "Microsoft Win32 Programmer's Reference, Vol2"의 52장이나 이 책의 11장과 12장을 참조하기 바란다. 여기에는 윈도우즈 95의 플러그 앤 플레이 서브 시스템과 레지스트리의 사용을 힘들게 만드는 구성 관리자의 요소들이 설명된다.

4.3 인터페이스 호환성 (Compatibility Interface)

이전 버전의 윈도우즈에서는 몇 개의 인터페이스가 중요했다. 이것은 DPMI(DOS Protected Mode Interface), 익스텐디드 메모리와 익스팬디드 메모리 관리를 상징하는 XMS 와 EMS, 가상 DMA 서비스들을 포함한다. 윈도우즈 95는 응용프로그램에 대하여 호환성을 제공하기 위해 이런 인터페이스를 계속 지원한다.

4.3.1 DPMI

DPMI는 보호모드 프로그램에게 윈도우즈 95 시스템 서비스의 일부분을 액세스하도록 해 준다. DPMI의 원래 목적은 상용 도스 확장자가 윈도우즈 3.0의 도스 가상 머신에서 동작할 수 있게 하는 것이었다. 윈도우즈 95는 아직도 DPMI에 의존하는 응용프로그램이 정지하는 것을 막기 위해 이것을 지원한다.

DPMI는 가상 머신에서 실행되는 도스 확장자를 돕는 일을 한다. 프로그램은 V86 모드에서 실행되고, 모드 전환 함수의 주소를 얻기 위해 INT 21h, 함수 1687h를 호출한다. 그리고 나서 프로그램은 "호스트"로 부터 추가적인 서비스를 요청하기 위해 INT 31h를 사용한다. 실행이 끝나면, 프로그램은 즉시 보호모드를 끝내고 도스 프롬프트로 돌아가기 위해 INT 21h, 함수 4Ch를 실행한다. 흥미로운 점은 윈도우즈 95 KERNEL 요소는 시스템 VM에서 윈도우즈 요소들을 실행시키기 위해 이와 똑 같은 메커니즘을 사용한다는 것이다.

DPMI를 사용하는 것은 어셈블리어를 사용하는 것이 실제적이다. DPMI호출은 수십개의 서비스중 하나를 지시하기 위해 AX 레지스터에 함수 코드를 사용한다. 어떤 것은 추가적인 레지스터를 사용하기 위해 다른 범용 레지스터와 세그먼트 레지스터를 이용한다. 결과는 항상 범용 레지스터를 통해 전달되며 캐리 플래그는 성공적인 수행(캐리플래그를 지움)과 실패(캐리 플래그를 세팅)를 나타낸다. 그러나 레지스터가 리턴 값을 제외하고는 다른 레지스터의 값들이 호출 전에 비해 변경되지 않으리라는 것은 문서에는 설명되지 않았다.

DPMI 함수 코드에서 상위 바이트는 서비스의 넓은 의미를 가리키는데 사용되며, 하위 바이트는 넓은 류의 서비스에 있는 특정한 서비스를 가리키는데 사용된다. 다음의 표 4-4는 DPMI 버전 0.9와 0Exxh인 추가 서비스를 나타내고 있다. DPMI 위원회는 일시 버전인 0.9보다 개발이 중지된 1.0 버전을 실제 스펙으로 하려고 하고 있다. 그러나 마이크로소프트는 코프로세서 관리를 위한 기능 0E00h를 제외하고 추가적인 규약을 전혀 이행하지 않는다. 그래서, 윈도우즈 95는 윈도우즈 NT가 오늘날 그랬던 것처럼 윈도우즈 3.0과 3.1의 DPMI 레벨 DPMI 0.9만을 계속 지원할 것이다.

함수 분류	설 명
00xxh	셀렉터와 디스크립터 관리 서비스
01xxh	V86 메모리 관리 (하위 1MB) 서비스
02xxh	인터럽트 후킹 서비스
03xxh	보호모드에서 리얼모드로, 혹은 리얼모드에서 보호모드로의 호출
04xxh	버전을 알려주는 서비스 (단지 0400h 하나만 제공한다)
05xxh	익스텐디드 메모리 관리 서비스
06xxh	페이지 락 서비스
07xxh	페이지 성능 튜닝 서비스
08xxh	물리 메모리 맵핑 서비스 (단지 0800h 하나만 제공한다)
09xxh	가상 인터럽트 상태 관리 서비스
0Axxh	제작자가 제공하는 엔트리 포인트를 위치시키기 위한 진부한 함수 (0A00h)
0Bxxh	디버그 레지스터 관리 서비스
0Exxh	부동 소숫점 코프로세서 관리 서비스

표 4-1. DPMI 서비스 분류

만약 DPMI가 제공하는 서비스의 종류를 좀더 자세히 살펴보면 도스확장자에 의해 제공되는 함수들에 깊은 관심을 가지게 될 것이다. 따라서 이를 설명하기 위해 도스 확장자에 대한 간단한 개요를 설명한다. 내가 알고 있는 도스 확장자는 기본적으로 표준 MS-DOS와 바이오스 소프트웨어 인터럽트 연결을 지원하기 위한 런타임 라이브러리를 가지고 있는 프로그램 로더이다. 도스 확장자는 보호모드로 전환후 실행 화일을 열고 읽어야 한다. 이러한 목적에는 경제적인 코딩 효율을 위해 표준 INT 21h를 사용을 권고한다. 그러나, 도스 확장자는 리얼모드 도스로 이것을 전해 주기 위해 DPMI 함수 0205h를 사용해 보호모드 인터럽트를 후크해야 할 필요가 있다.

```

mov ax, 0205h      ; 함수 0205h : PM의 인터럽트 벡터 설정
mov al, 21h       ; BL = 인터럽트 번호
mov cx, cs        ; CX:DX -> 인터럽트 핸들러
mov dx, offset int21 ; ..
int 31h          ; 프로텍티드 모드에서 인터럽트를 후크함

```

도스 확장자는 인터럽트 디스크립터 테이블을 쉽게 액세스 할 수 없기 때문에 DPMI 호스트로부터의 이러한 원조가 필요하다. 이제, 도스 확장자가 INT 21h 호출을 발생 시켰을 때 int21 이라는 이름의 프로그램이 제어권을 얻는다. (여전히 보호모드에 있음). 이것은 리얼모드로 인터럽트를 전해 주기 위해 DPMI 함수 0300h(리얼모드 인터럽트 시뮬레이트)와 같은 것을 사용할 것이다. 시스템 소프트웨어로부터의 도움이 한번 더 필요한데, 왜냐하면 보호모드에서 리얼모드로 전환하고 또 되돌아오기 위해서는 특권레벨 0인 슈퍼바이저 프로그램만이 액세스 할 수 있는 프로세서 레지스터의 변경이 필요하기 때문이다. 또한 도스 확장자의 작업 부분은 보호모드 프로그램에 의해 사용되는 selector:offset 형식의 포인터로부터 리얼모드에서 사용하는 segment:offset 형식으로 변환하는 작업이다. 또한, 변환작업은 확장 메모리 위치에서 가상 메모리의 첫 1M 바이트 위치로 복사하는 작업이 필요하다. 왜냐하면 V86 프로그램은 첫 1M 바이트 이하에 있는 것만 액세스 가능하기 때문이다. DPMI 함수 0100h, 그 메모리 주소 지정을 위한 셀렉터의 할당을 하는 0000h, 리얼모드 메모리의 주소를 셀렉터의 베이스 주소로 하는데 0007h의 사용이 필요하다.

비록 이러한 사실들이 널리 알려진 것은 아니지만, 이러한 어지간한 도스 확장자가 윈도우즈 3.0에는 벌써 있

있고 윈도우즈 95에는 여전히 있다는 것이다. 만약 당신이 프로그램을 로드할 수 있고 보호모드 선택터를 사용하는 것을 제배치 할 수 있다면 윈도우즈는 그것이 사용하는 모든 소프트웨어 인터럽트를 핸들링 할 것이다. 그래서 도스 확장자가 하는 INT21h 호출은 필요하지 않다. 왜냐하면, INT 21h의 윈도우즈 디폴트 핸들링에 있어서 포인터 인자를 변환하여 리얼모드 인터럽트로 전달한다. 윈도우즈는 마우스 드라이버 호출(INT 33h), 비디오 바이오스 호출(INT 10h), 기타의 적절한 핸들링을 제공한다. 따라서 윈도우즈 3.0을 사용한 도스 확장자는 보호모드의 스위칭과 표준화일 시스템을 사용하여 클라이언트 프로그램을 익스텐디드 메모리로 로드를 간단하게 한다. (이것은 표준 런타임 라이브러리가 보호모드에서 동작하는데 아주 적은 부분만이 문제가 있다는 것이고, 이것도 아주 쉽게 고칠 수 있다. 그러나 대부분의 상업용 도스 확장자가 이것에 비해 매우 심하다.)

윈도우즈 95는 확장 도스 프로그램에 매우 유용한 DPMI를 제공하지만 이것의 주요 목적은 어디까지나 이들의 위한 호환성을 유지하는데 있다. 하지만 Win32 API를 이용해 32비트 콘솔 응용 프로그램을 만들 수 있기 때문에 굳이 상업용 도스 확장자를 살 이유가 없을 것이다. 만약 확장 도스 응용 프로그램을 작성해 보고 싶다면, Win32 콘솔 응용 프로그램으로 새로 작성하는 대신 특정 도스 확장자로부터 벗어나 다른 32 비트 플랫폼으로 포팅 가능한 방법을 익혀야 할 것이다.

아직은 DPMI를 16비트 윈도우즈 프로그램에 사용하는 데는 몇 가지 한계점을 가지고 있다. 물론 32비트 윈도우즈 프로그램은 DPMI를 호출이나 소프트웨어 인터럽트를 기반으로 하는 어떤 인터페이스도 호출할 수 없다. 응용 프로그램이 비표준 리얼모드 프로그램 호출할 때는 도스 확장자가 필요로 하는 포인터를 변환해 주지 못하기 때문에 DPMI 0300h 함수를 사용해야 할 필요가 있다. DPMI 함수 09xxh는 가상 인터럽트 플래그를 제어할 수 있도록 해주는데, 가상 인터럽트 플래그는 VMM이 인터럽트를 VM으로 전달할 것인지에 대하여 문 역할을 수행한다.

DPMI 함수 0800h는 액세스하려고 하는 물리 메모리를 가상 주소로 변환 해 주는 역할을 한다. 이런 기능은 확장 카드 같은 것에 유리한데, 보통 이런 카드들의 물리 주소는 알려져 있기 때문이다. 또한 다양한 DPMI 호출을 사용하여 필요한 선택터를 가진 포인터를 만들 수 있다. 윈도우즈의 *AllocSelector*, *SetSelectorBase*, *SetSelectorLimit* API 함수는 이런 것을 편리하게 한다. 이러한 선택터 관리 함수들이 윈도우즈 3.1 SDK에 문서화 되기 전에, 많은 프로그래머들은 이러한 방법이 있었는지 몰랐기 때문에 DPMI 서비스에 의존했다.

마지막으로, DPMI 호출은 각각 메모리를 나누어 쓰기 위한 구식 응용 프로그램(MS-DOS용이나 이전 버전의 윈도우즈용으로 작성된 응용 프로그램)을 위해 아직도 지원된다. 한 개의 가상 머신에서 운영되고 있는 확장 도스 프로그램은 80000000h ~ C0000000h의 어드레스 공간 영역을 나누어 사용하는 선형주소의 확장 메모리 블록을 할당받기 위해 함수 0501h를 사용할 수 있다. 다른 가상 머신에서 운영되고 있는 확장된 도스 프로그램은 같은 선형 주소를 액세스 할 수 있다. 마이크로소프트는 아직도 이러한 방법으로 작업하는 응용 프로그램보다는 파일 맵핑이라 불리는 방법으로 대화하는 Win32 프로그램으로 재구성되는 것을 오히려 좋아한다.

DPMI에 대한 정보는 인텔을 포함한 몇 개의 공인된 명세서에 보면 잘 나타나 있다. 이 책의 17장에서 DPMI에 대해 좀더 자세히 논의한다.

4.3.2 익스텐디드 메모리 관리 (Extended Memory Management)

XMS(Extended Memory Specification)은 MS-DOS 프로그램이 물리 주소 공간의 첫 1M 바이트와 익스텐디드 메모리를 어떻게 액세스하고 사용하는지에 대한 내용이 담겨 있다. 모든 윈도우즈 95는 윈도우즈 95 버전의 HIMEM.SYS 같은 XMS 제공자를 포함하고 있다. 리얼모드 프로그램인 인터럽트 2Fh 함수 4301h를 사용해서 XMS 서비스 루틴의 주소를 얻는다. 이 인터럽트는 AH 레지스터에 사용하는 함수 코드에 따라 메모리 할당 등의 다양한 동작을 하는 리얼모드 함수를 리턴 한다. 그러나 보호모드에서는 XMS를 직접 사용할 수 없음을 기억하라. 그래서 도스 확장자 같은 프로그램은 XMS 서비스 함수 호출을 위해서 우선 리얼모드로 전환할 필요가 있다.

XMS 서비스는 선형 주소 00100000h에서 시작되는 "high memory block"의 영역을 예약하고 해제하는 호출자를 허용한다. 이것은 익스텐디드 메모리 블록의 할당과 해제, A20 하드웨어 라인의 제어를 위함이다. (A20 하드웨어 라인은 물리 주소의 21번째 비트를 의미한다. 왜냐하면, 8086 프로세서는 단지 20비트의 물리 주소를 지원하므로, 80286이나 그 이후의 프로세서에서 프로그래머들이 A20 하드웨어 라인을 디스에이블 해서 하위 호환을 제공

한다.) MS-DOS 프로그램은 직접 XMS 서비스를 잘 사용하지 않으며, 도스 확장자와 같은 프로그램 상에서 실행되는 응용 프로그램이 XMS 제공자 인터페이스에 더 유용하다.

XMS는 윈도우즈 95에 2가지 작동을 한다. 첫째, 윈도우즈 95는 시스템 시작시 시스템에 존재하는 남은 익스텐디드 메모리의 제어권을 갖기 위해 사용한다. 둘째, 윈도우즈 95는 XMS를 사용하는 리얼모드 소프트웨어에 대하여 호환성을 유지하기 위해 XMS서비스를 시뮬레이트 한다. 예를 들어, 다음의 DEBUG 세션은 XMS 함수 01h를 사용하고 있는데, 이것은 XMS 진입점을 구하고 XMS 제공자의 버전 번호를 알아보기 위한 것이다.

```
C:\>debug
-a 100
2DBE:0100    mov ax, 4301
2DBE:0103    int 2f
2DBE:0105    mov [200], bx
2DBE:0109    mov [202], es
2DBE:010D    mov ah, 0
2DBE:010F    call far [200]
2DBE:0103    int 3h5 2DBE:0100
2DBE:0104
-g

AX=0300 BX=035F CX=0000 DX=0001
...
2DBE:0113    CC INT 3
-q
```

AX 레지스터의 리턴 값은 XMS 제공자가 버전 3.00을 지원한다는 것을 가리킨다. BX 레지스터의 값은 XMS 제공자의 내부 개정 수준이 3.95임을 가리킨다. XMS에 대한 추가적인 정보는 MSDN(Microsoft Developer Network) 디스크에 있는 “*Extend Memory Specification (XMS) 3.0*”의 “Specifications” 절을 참조하기 바란다.

4.3.3 익스팬디드 메모리 관리 (Expanded Memory Management)

소위 말하는 “익스팬디드” 메모리는 PC 표준 하드웨어와 소프트웨어의 성장을 그대로 보여주는 역사를 가지고 있다. 일찍이 MS-DOS 응용 프로그램은 좀처럼 진보하지 못했다. 왜냐하면 이들은 고작 640KB의 메모리만 사용할 수 있었기 때문이다. 대용량의 램에 대한 초기의 해결책은 로터스-인텔-마이크로소프트(Lotus-Intel-Microsoft; LIM) 익스팬디드 메모리 규약이었다. 이 대용량의 램은 수 메가바이트의 메모리가 있는 확장 카드로 PC에 꽂혀 사용되었다. 이 카드는 원래 롬 영역(A000h:0000h)에서부터 BIOS 영역의 시작점인 F000h:0000h)인 곳에 세그먼트 주소를 구성(*page frame*)하여 일반 램처럼 동작하였다. 소프트웨어는 이 익스팬디드 메모리를 16KB의 조각으로 만든다음, 페이지 프레임의 주소를 통해 이 메모리를 사용할 수 있도록 카드를 프로그램 할 수 있었다. 응용 프로그램은 소프트웨어 인터럽트 INT 16h를 이용하여 드라이버를 액세스하였는데, 이것은 서로 다른 카드를 사용하기 위함이었다.

80386 프로세서의 출현으로 소프트웨어에서 익스팬디드 메모리를 시뮬레이트 할 수 있게 되었다. EMM386, QEMM, 386Max와 같은 익스팬디드 메모리 관리자는 386의 가상 86모드에서 동작했으며 386의 페이지 기능을 사용했다. 요약하면, 이 익스팬디드 메모리 관리자는 컴퓨터를 가상 86모드로 전환하고, 페이지 테이블이 가상 메모리의 하위 1MB가 물리 주소의 위치와 동일하도록 한후 페이지 기능을 인에블 한다. 익스팬디드 메모리 관리자는 지정한 V86 페이지 프레임 어드레스에 익스팬디드 메모리의 페이지의 주소를 맵핑하는 방법으로 익스팬디드 메모리를 제공한다.

근래의 익스팬디드 메모리 관리자는 물리 주소의 빈 공백 (램이 존재하지 않는 영역)을 익스텐디드 메모리를 이용하여 채운다. 일반적인 PC의 0000h:0000h ~ A0000h:0000h에 해당되는 00000h ~ A0000h 영역은 램이 존재한다. 나머지의 384KB에 확장 카드와 BIOS가 제공하는 롬과 램이 다양한 위치에 존재한다. 그러나 그곳에는 메모리가 존재하지 않는 공백이 있다. 메모리 관리자는 이 공백에 소위 말하는 *upper memory block*을 만들고 여기에

MS-DOS가 디바이스나 TSR을 위해 사용할 수 있도록 한다.

불행히도 V86모드에서 실행되는 PC에는 부작용이 있었다. 리얼모드에서 일반적으로 잘 실행되던 명령어가 V86모드에서는 특권 명령으로 되어 갑자기 일반 보호 폴트(general protection fault; GP fault)를 발생시킨다. 이러한 문제를 제일 처음 겪은 프로그램은 도스 확장자였다. 32비트 도스 확장자와 메모리 관리자의 선두 업체들(파랩이나 퀴터텍)은 VCPI(Virtual Control Program Interface) 표준을 정하는데 협력했다. 이 VCPI는 도스 확장자와 메모리 관리자가 함께 공존할 수 있는 방법을 기술하고 있다. 또한 보호모드로의 진입과 탈출, 물리 메모리 관리, 특권 레지스터의 액세스를 지원하기 위해 소프트웨어 인터럽트 67h를 제공했다.

윈도우즈 95는 VCPI 메모리 관리자와 공존할 없다. (역자주 : 실제로 마이크로소프트는 뒤늦게 VCPI와 경쟁 관계에 있는 DPMI를 만들었다. 기술적으로 본다면 VCPI는 보호모드에서의 충돌을 방지하기 위한 최소한의 규약과 기능이 있지만 DPMI는 실제로 서버와 클라이언트와의 관계로 규정될 만큼 복잡다단하고 많은 기능이 제공된다. 소문에 의하면 마이크로소프트는 업체들을 VCPI에서 DPMI로 돌리기 위해 많은 노력을 했다고 한다. 물론 그때의 제조업체들은 마이크로소프트가 윈도우즈를 띄우기 위해 DPMI가 필요했다는 사실을 꿈에도 몰랐을 것이다. 지금은 DPMI에 대한 마이크로소프트의 입장이 대단히 거만해서 규약의 버전 업이나 그 상세 사양에 대해서는 거의 함구하고 있는 실정이다.) 실제로 윈도우즈 95의 도스 창에서 VCPI를 검사하기 위해 인터럽트 67h 함수 DE00h를 사용하면, VCPI가 존재하지 않는다는 결과를 얻게 될 것이다. 이것은 윈도우즈 95를 실행하기 전에 VCPI 메모리 관리자를 로드 했더라도 마찬가지일 것이다. 윈도우즈 95는 시작할 때, 윈도우즈 95가 머신을 장악하려고 한다는 것을 모든 램상주 소프트웨어들에게 알리기 위해 INT 2Fh 함수 1605h를 발생시킨다. VCPI 메모리 관리자는 나중에 윈도우즈 95가 V86 모드에서 리얼모드로 전환할 때 호출할 수 있도록 모드 전환 루틴의 주소를 제공한다. 또한 이 메모리 관리자는 윈도우즈 95에게 페이지 맵핑에 관한 정보를 제공하지만 이것은 문서화 되어있지 않다. 이 맵핑 정보는 윈도우즈 95가 시작할 때 V86 메모리 관리 VxD가 이미 사용중인 상위 메모리 블록(upper memory block)의 중복을 피하기 위함이다. 따라서 윈도우즈 95가 운영 중일 때는 어떠한 확장 TSR도 VxD의 도움 없이 실행할 수 없다.

좀더 추가적인 정보는 Ralf Brown과 Kyle가 쓴 "PC Interrupts", (Addison-Wesley, 1991)의 10에서 얻을 수 있다.

4.3.4 가상 DMA 서비스 (Virtual DMA Services)

DMA(direct memory access)는 메모리로부터(혹은 메모리로) 데이터의 빠른 전송을 위한 하드웨어 장치이다. 그러나 불운하게도, DMA 컨트롤러는 세 가지의 한계점이 있다. 바로 DMA는 오직 물리 주소만 사용할 수 있고, 메모리 경계 교차(memory boundary crossing)가 금지되어 있고, (사용한 하드웨어 버스에 따라 다르지만) 대개 16MB이하의 물리 주소에만 사용할 수 있다는 것이다. 윈도우즈 95에서 실행되는 프로그램들은 가상 메모리를 사용하는데, 이 선형 주소는 DMA에게 의미가 없다는 것이며, 컴퓨터에는 16MB보다 더 많은 메모리가 있을 수 있다는 것이다.

그러나 이런 문제에 대하여 리얼모드 프로그램은 기존 방식대로 DMA를 프로그램해서 사용할 수 있으며, 윈도우즈 95 프로그램은 이러한 문제점들을 완전히 무시하고 사용할 수 있다. 이것은 가상 DMA 디바이스(Virtual DMA Device; VDMAD)가 모든 일을 대신하는 방식으로써 DMA 제어 포트를 가상화하기 때문이다. 따라서 리얼모드 플로피 디스크 드라이버는 V86 메모리 관리자가 메모리를 16MB 이상의 메모리로 할당했는지 모른다는 걱정 없이 지정된 V86 메모리로 전송하기 위해 DMA를 초기화할 수 있다.

DMA 전송을 원하는 가상 디바이스 드라이버 프로그래머는 DMA 제어 포트를 직접 액세스하는 대신 VDMAD 서비스를 호출하면 DMA 제어 능력에 대하여 자세히 신경 쓰지 않아도 된다.

보호모드 프로그램(3순위 권한 디바이스 드라이버를 포함하여)은 VDMAD의 가상화 영역밖에 있거나 이에 대한 서비스를 사용하지 않는 종류의 프로그램이다. 무슨 이유에선지, VDMAD는 3순위 권한의 보호모드 프로그램에 의해 직접 초기화되지 않는다. 대신 이런 프로그램은 가상 DMA 서비스(Virtual DMA Services; VDS) 호출을 통해서 요청할 수 있다. 이 VDS의 기본 목적은 보호모드 응용 프로그램에게 소프트웨어 인터럽트 4Bh를 이용하여 VDMAD를 액세스하는 방법을 제공하기 위한 것이다.

가상 DMA 서비스에 대한 좀더 자세한 것은 "Virtual DMA Services(VDS) Sepcification 1.0"의

“Specifications” 절을 참조하기 바란다. 이는 MSDN(Microsoft DeveloperNetwork) 디스크에서 찾을 수 있다.