하드웨어 제어 기법의 정복

(Mastering Hardware Control Techniques)

MS-DOS 시절에는 어플리케이션으로 기계를 직접 제어할 수 있었다. 비록 귀찮은 작업들이 많았지만, 프로그래머가 하드웨어에 직접 접근할 수 있었기에 빠른 속도를 낼 수 있었고, 동시에 많은 일들을 마음대로 할 수 있었다.
윈도우 3.1 로 넘어오면서 이러한 자유로움에 제한을 받기 시작했다. 더 이상 하드웨어에 직접 접근하는 것을 자유롭게 허용받지 못했다. 이는 어찌보면 당연하다고 말할 수 있다. 윈도우에서는 사용자가 수많은 어플리케이션을 사용할 수 있으며, 또한 같은 하드웨어를 동시에 접근하지 말라는 보장도 없다. 그리고, 윈도우는 멀티 태스킹 환경이기 때문에 다른 어플리케이션의 동작을 방해해서는 안되고, 이를 위해서 직접적인 하드웨어의 제어는 위험한 발상일 수도 있는 것이다. 그러나, 하드웨어를 직접 제어할 때의 장점도 무시할 수 없는 것이다.

하드웨어 포트 직접 제어의 문제점

이제는 Win32 (윈도우 NT, 윈도우 95)의 시대이다. 이들은 진정한 운영체제이며, 기본적으로 선점형(pre-emptive) 멀티 태스킹 환경이다. 그렇기 때문에, 각각의 쓰레드(실행단위)는 프로세서에 일정량의 시간을 할당 받는다. 시간이 되거나, 보다 높은 순위의 쓰레드가 있을 경우 시스템은 다른 쓰레드에 컨트롤을 넘긴다. 이러한 스위칭(switching)은 어떠한 어셈블리 코드 사이에서도 이루어지며, 하나의 쓰레드가 동작하는 도중에 이 쓰레드가 완료되기 전에 컨트롤이 다른 쓰레드로 넘어갔다가 얼마나 오랜 시간이 지난 후에 다음 코드가 진행될 수 있을지는 알 수 없다. 이런 점이 직접적인 하드웨어 제어를 할 때의 가장 큰 문제점이다.
가장 일반적인 형태의 I/O 포트의 입력 부분은 다음과 같은 몇 개의 어셈블리 코드로 이루어진다.

```
    mov    dx, AddressPort
    mov    al, Address
    out    dx, al
    jmp    Wait
Wait:
    mov    dx, DataPort
    in     al, dx
```

쓰레드가 변경될 때 모든 레지스터의 상태는 보존되더라도, I/O 포트의 상태는 보존되지 않는다. 그렇기 때문에, in, out 코드 사이에 자신의 I/O 포트 값을 보존할 수 있는 방법을 강구해야 한다.

표준적인 방법

뮤텍(mutex)을 이용하는 방법이다. 문제는 다른 어플리케이션에서도 뮤텍을 무시하지 않아야 한다는 것이다. 그 밖에도 몇 가지 문제점이 있는데, 예를 들어 App1, App2 라는 2 개의 어플리케이션이 있다고 하자. 이들이 모두 포트를 제어하려고 하는데, 이를 만든 제작자의 관점이 달라서 App1 은 일단 AddressPortMutex 을 먼저 얻으려고 하고, App2 는 DataPortMutex 을 먼저 얻으려고 한다고 하자. 그래서, 이들이 각각 뮤텍을 얻고 나서 시스템이 App1 에서 App2 로 쓰레드를 넘겼다고 하자. 그러면, App2 는 address port 를 얻을 수 없는 deadlock 상황이 되버린다. 또한, App1 은 data port 를 얻을 수 없게 되버린다. 이런 문제를 해결하기 위한 가장 좋은 방법은 port/memory area 를 제어하는 디바이스 드라이버를 만드는 것이다. 이때에는 API 를 사용하는데, 가장 전형적인 함수는 다음과 같다.

```
GetIOPortData(AddressPort, DataPort: word): Byte;
```

GetIOPortData 함수는 일단 양쪽 포트에 뮤텍을 걸고, 포트에 접근한다. 그리고, 호출자(caller)에게 결과 값을 돌려주기 전에 뮤텍을 해제한다. 만약 다른 쓰레드가 이 함수를 동시에 사용하면 한 쪽에서 먼저 동작하고, 다른 쪽에서는 대기하게 되므로 deadlock 은 발생하지 않는다.
문제는 디바이스 드라이버를 제작하는 것이 별로 쉽지 않다는 것이다. 보통 어셈블러나 C 로 제작하게 되는데, 디버깅하기도 어렵거니와 윈도우 NT 의 디바이스 드라이버(VDD, virtual device driver)와 윈도우 95 의 디바이스 드라이버(VxD) 사이의 호환성도 보장할 수 없다. 그러므로, 각각에 대해 다른 디바이스 드라이버를 제작해 주어야 한다.
디바이스 드라이버를 제작하는 방법에 대한 것은 이 책이 다룰 범위를 넘기 때문에, 여기서는 더 이상 다루지 않도록 하겠다.

편법적인 방법

보통 하드웨어를 직접 다루려고 하는 것은 특정 하드웨어에 접근하는 어플리케이션을 만들때이기 때문에, 디바이스 드라이버를 제작한다는 것은 사실 좀 비현실적이고, 낭비라고 할 수 있다.
다행히(?) 윈도우 95 는 윈도우 3.1 과 호환되게 만들어져 있어서 직접적인 I/O 제어가 일부

나마 가능하다 (이는 많은 수의 윈도우 3.1 프로그램이 직접 I/O 포트를 제어하기 때문이다.). 다음과 같은 함수를 이용하면 일단 포트에 접근하여 데이터를 주고 받을 수 있다.

```
function GetPort(p: Word): Byte; stdcall;
begin
  asm
    push   edx
    push   eax
    mov    dx, p
    in     al, dx
    mov    @result, al
    pop    eax
    pop    edx
  end;
end;

procedure SetPort(p: Word; b: Byte); stdcall;
begin
  asm
    push   edx
    push   eax
    mov    dx, p
    mov    al, b
    out    dx, al
    pop    eax
    pop    edx
  end;
end;
```

이와 비슷한 코드 들을 이용하게 되는데, 같은 기능을 하는 다른 코드를 살펴보자.

```
function PortIn(IOAddr: Word): Byte;
begin
  asm
    mov   dx, IOAddr
    in    al, dx
```

```
    mov   result, al
  end;
end;

procedure PortOut(IOAddr: Word; Data: Byte);
begin
  asm
    mov   dx, IOAddr
    mov   al, Data
    out    dx, al
  end;
end;
```

위의 두 코드를 비교해 보면 알겠지만, 핵심은 dx, al 레지스터에 각각 포트의 주소와 데이터의 값을 저장하고, 이를 이용해서 out, in 을 하면 된다.
이제 조금은 효율적이면서 여러모로 쓰일 수 있는 간단한 유닛을 만들어 보자

```
unit U_Port;

interface

function PortReadByte(Addr: Word): Byte;
function PortReadWord(Addr: Word): Word;
function PortReadWordLS(Addr: Word): Word;
procedure PortWriteByte(Addr: Word; Value: Byte);
procedure PortWriteWord(Addr: Word; Value: Word);
procedure PortWriteWordLS(Addr: Word; Value: Word);

implementation

function PortReadByte(Addr: Word): Byte; assembler; register;
asm
  mov   dx, ax
  in      al, dx
end;
```

```
function PortReadWord(Addr: Word): Word; assembler; register;
asm
  mov   dx, ax
  in      ax, dx
end;

function PortReadWordLS(Addr: Word): Word; assembler; register;
const
  Delay = 150;                  //CPU 속도와 카드의 속도에 따라 조절
asm
  mov   dx, ax
  in      al, dx                //LSB 포트 값을 읽는다.
  mov   ecx, Delay
@1:
  loop   @1                     //Delay ...
  xchg   ah, al
  inc     dx                    //포트 + 1
  in      al, dx                //MSB 포트 값을 읽는다.
  xchg   ah, al                 //바이트 순서를 바로 잡음
end;

procedure PortWriteByte(Addr: Word; Value: Byte): assembler; register;
asm
  xchg   ax, dx
  out     dx, al
end;

procedure PortWriteWord(Addr: Word; Value: Word); assembler; register;
asm
  xchg   ax, dx
  out     dx, ax
end;

procedure PortWriteWordLS(Addr: word; Value: Word): assembler; register;
const
  Delay = 150                   //CPU, 카드 속도에 따라서...
```

```
asm
  xchg   ax, dx
  out    dx, al              //LSB 포트에 쓰기
  mov    ecx, Delay
@1:
  loop   @1                  //Delay...
  xchg   ah, al
  inc    dx                  //포트 + 1
  out    dx, al              //MSB 포트에 쓰기
end;

end.
```

윈도우 NT 의 경우

앞의 코드는 윈도우 NT 하에서는 동작하지 않는다. 기본적으로 윈도우 NT 는 플랫폼에 구애받지 않기 때문에, 이런 방식의 I/O 포트 제어가 프로세서에 따라 다르게 되어야 하므로 앞의 방식은 통하지 않는다.

그렇다고 전혀 방법이 없는 것은 아니다. NT 에서 윈도우 95/98 의 환경과 다른 점은 사용자 모드에 있는 코드가 직접 하드웨어에 접근할 수 없도록 막는다는 점이다. 즉, 모든 하드웨어 리소스는 운영체제에 의해 관리되는 것이다.

그렇지만, 이러한 기본적인 원칙에도 통하는 방법은 있기 마련이다. NT 커널은 I/O 포트 주소의 맵을 관리하고 있는데, 각각의 프로세스가 접근할 수 있도록 허용한다. 그러므로, 현재의 프로세스가 NT 에게 다른 IOPM(I/O Permission Map)에 접근한다고 알리고 포트에 접근할 수가 있는데, 이런 방법은 OS 의 측면에서 보면 그다지 바람직하지 못한 것이므로 절대로 남발해서는 안된다. 그렇지만, 특정 하드웨어에 접근해서 사용해야 하지만 NT 용 디바이스 드라이버를 새로 작성할 수 없는 특수한 경우에는 이런 방법을 쓸 수 밖에 없을 것이다.

여기서 가장 큰 문제는 앞에서도 언급했듯이 사용자 모드의 코드가 커널 함수를 이용하여 IOPM 을 변경할 수 없다는 점이다. 이를 해결하기 위해서 어플리케이션의 요구에 따라 IOPM 의 내용을 변경할 수 있는 NT 드라이버를 이용할 수 있다. 이 방법이 가능한 이유는 디바이스 드라이버는 IOPM 을 변경할 권한이 있기 때문이다. 즉, 사용자의 코드가 직접 포트에 접근할 수 없다면, 포트에 마음대로 접근할 수 있는 디바이스 드라이버를 거쳐서 이를 제어하는 것이다. 여기에 가장 흔히 사용되었던 디바이스 드라이버가 Dale Roberts 가 작성한 giveio.sys 디바이스 드라이버이다. Giveio.sys 디바이스 드라이버는 문서화되지 않은 커널 함수를 이용하여 모든 I/O 포트에 접근할 수 있도록 하였다. 여기서는 이 디바

이스 드라이버를 사용하지 않고, Graham Wideman 이 선택된 포트에만 접근할 수 있고 몇 가지 진단 함수를 추가하여 새롭게 작성한 gwiopm.sys 디바이스 드라이버를 사용한다. 이 디바이스 드라이버의 소스는 이번 장과 관련한 CD-ROM 디렉토리의 Gwiopm 서브 디렉토리에 C 로 작성된 소스가 담긴 Src 디렉토리에 포함되어 있으므로 관심이 있는 독자는 분석해보기 바란다. 그와 함께 Wideman 이 직접 예제로 제공한 PortTest 디렉토리도 같이 제공된다. gwiopm.sys 드라이버에 대한 델파이 인터페이스 유닛으로 gwiopm.pas 유닛 역시 Wideman 이 제공하는데, 이를 이용하여 프로그래밍을 하는 방법에 대해서 알아보도록 하자.

디바이스 드라이버 파일이 있다고 해서 이를 바로 사용할 수는 없다. 이를 제대로 사용하려면 드라이버를 설치해야 한다. NT 의 경우에는 레지스트리를 조작해야되는 문제가 있는데, 앞서 46 장에서도 언급했듯이 수동으로 레지스트리를 직접 조작하면 해결할 수 있는 것들이 많지만 의외로 이런 일을 할 수 있는 사람을 극소수이다. 그러므로, 이를 프로그램에서 해결할 수 있도록 해결책을 제시하는 것은 필수이다. Wideman 은 gwiopm.pas 유닛에 이런 설치를 담당하는 함수를 같이 제공하여 쉽게 드라이버를 설치해서 사용할 수 있도록 해주고 있다.

- gwiopm 유닛에 대한 고찰

gwiopm.pas 유닛에는 드라이버를 설치하고 IOPM 을 준비하는 등의 역할을 하는 GWIOPM_Driver 객체가 포함되어 있다. 하나의 IOPM 은 8KB 의 크기로 구성되어 있는데, 하나의 비트가 x86 의 64K I/O 포트 주소 공간 하나를 제어한다.

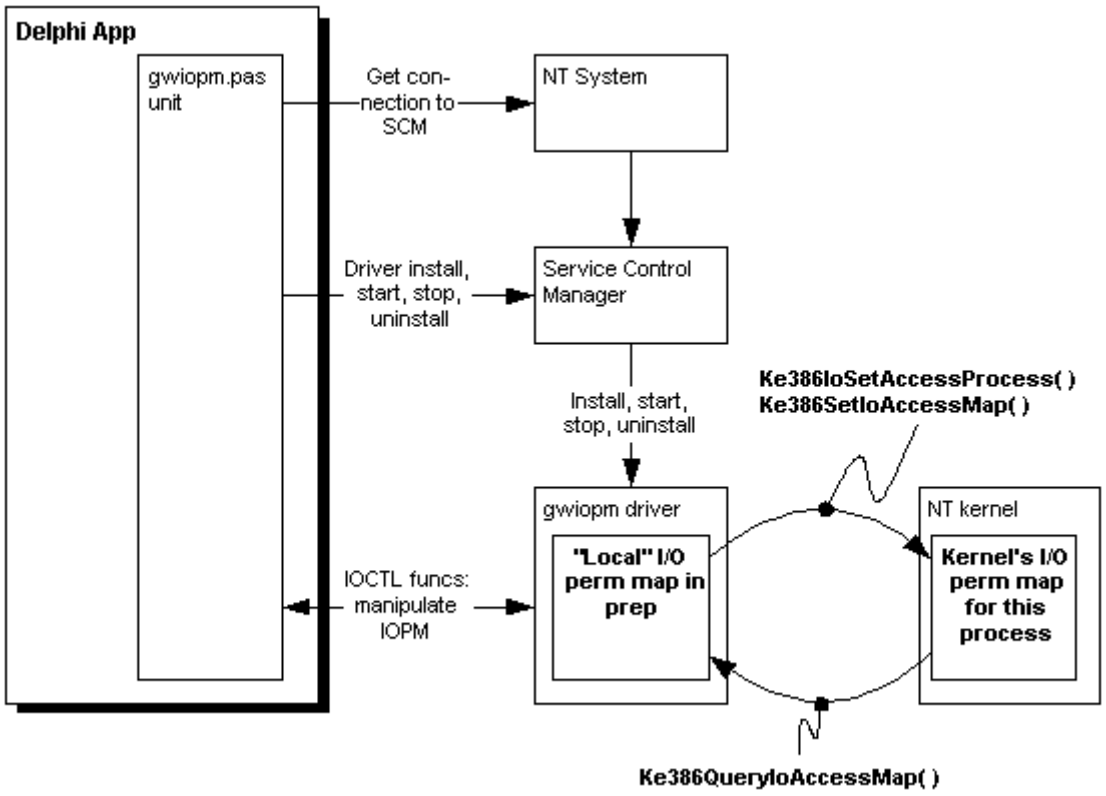
gwiopm.pas 유닛의 역할은 윈도우 NT 의 SCM(Service Control Manager)에 접근하여 접속을 한다. 그리고 드라이버를 설치, 시작, 중지, 제거할 수 있는 방법을 제공하며, IOCTL 함수를 이용하여 IOPM 을 관리할 수 있도록 해준다.

실제로 커널의 IOPM 과 드라이버의 IOPM 을 상호작용할 수 있도록 매핑하는 역할을 하는 것이 gwiopm.sys 디바이스 드라이버의 구현 방법이다. 이를 위해서는 커널의 함수로 Ke386IoSetAccessProcess, Ke386SetIoAccessMap, Ke386QueryIoAccessMap 함수를 사용하는데, 이들의 역할은 다음과 같다고 추정된다 (추정된다는 표현을 쓴 이유는 이들 함수가 모두 문서화되지 않은 함수이기 때문이다). 사용법을 익히고자 하는 독자는 CD-ROM 에 제공되는 C 소스 코드를 참고하기 바란다.

커널 함수 (void 형)	설 명
Ke386IoSetAccessProcess(PEPROCESS, int);	커널에게 현재 프로세스를 제공하여, 이 프로세스가 특정 IOPM 을 사용하기 원한다는 것을 알리는 역할을 한다. 이 함수의 결과로 커널은 프로세스에 대한 주소 공간을 확보한다. int 파라미터의 값이 1 이면 특

	정 맵이 사용 가능한 것이다.
Ke386SetIoAccessMap(int, IOPM *);	드라이버가 8KB 크기의 새로운 맵을 커널에게 전달하는 함수이다. int 파라미터의 값이 1 이면 현재의 맵을 복사하는 것이며, 0 이면 커널의 맵은 $FF 로 채운다. 참고로 맵의 1 비트가 특정 포트를 담당하므로 int=0 을 파라미터로 사용하면 커널의 맵이 모두 clear 된다.
Ke386QueryIoAccessMap(int, IOPM *);	커널의 맵을 다시 드라이버의 버퍼로 복사하는 역할을 한다. int 파라미터는 1 로 설정한다.

이를 도식화한 그림이 다음과 같다.



그러면 gwiopm.pas 유닛에 의해 제공되는 함수에 대해서 알아보도록 하자. 이들 함수는 모두 GWIOPM_Driver 객체의 메소드로서 제공된다.

1. SCM 에 접속, 해제를 담당하는 함수

```
function OpenSCM: DWORD;
```

function CloseSCM: DWORD;

이 함수들은 윈도우 NT 의 SCM(Service Control manager)을 열고, 닫는 역할을 한다. OpenSCM 에 의해 SCM 의 핸들을 얻을 수 있으며, 이 핸들은 GWIOPM_Driver 객체에 의해 사용된다.

2. 드라이버 설치에 관련한 함수

function Install(newdriverpath: string): DWORD;
function Start: DWORD;
function Stop: DWORD;
function Remove: DWORD;

이 함수들은 각각 드라이버를 설치, 시작, 중지, 제거하는 역할을 담당한다. Install 함수는 드라이버의 패스를 지정하거나 널 문자열(‘’, 디폴트 값)을 지정한다. 널 문자열이 지정되면 드라이버가 어플리케이션과 같은 디렉토리에 있는 것으로 간주한다.

3. 디바이스 함수

function DeviceOpen: DWORD;
function DeviceClose: DWORD;

디바이스를 사용하기 위해서는 먼저 디바이스를 열면 GWIOPM_Driver 의 다른 드라이버 함수를 사용할 수 있는 디바이스 핸들을 가져올 수 있다. 드라이버 함수를 사용한 뒤에 드라이버를 제거하려면 먼저 디바이스를 닫아야 한다.

4. 테스트 함수

function IOCTL_IOPMD_READ_TEST(var RetVal: DWORD): DWORD;
function IOCTL_IOPMD_READ_VERSION(var RetVal: DWORD): DWORD;

드라이버가 잘 동작하는지 알아보기 위한 진단 함수이다. READ_VERSION 의 RetVal 파라미터 값은 보통 100 을 넘으며, READ_TEST 의 RetVal 값은 $123 이다.

5. 드라이버의 로컬 IOPM(LIOPM)의 조작 함수

```
function IOCTL_IOPMD_CLEAR_LIOPM:  DWORD;
function IOCTL_IOPMD_SET_LIOPM(Addr: Word; B: byte): DWORD;
```

드라이버의 맵 배열을 Clear 하고, 특정 값을 주소에 설정하는 함수이다.

```
function IOCTL_IOPMD_GET_LIOPMB(Addr: Word; var B: byte): DWORD;
function IOCTL_IOPMD_GET_LIOPMA(var A: TIOPM): DWORD;
```

이 함수 들은 특정 바이트 값을 가져오거나, 드라이버에서 맵 전체를 가져오도록 하는 함수이다.

```
function LIOPM_Set_Ports(BeginPort: word; EndPort: word; Enable: Boolean): DWORD;
```

특정 포트에 대한 비트를 Enable/disable 시키는 함수이다.

6. 커널 IOPM(KIOPM)과의 상호 작용을 담당하는 함수

```
function IOCTL_IOPMD_ACTIVATE_KIOPM: DWORD;          //드라이버 맵을 커널에 전달
function IOCTL_IOPMD_DEACTIVATE_KIOPM: DWORD;        //커널이 현재의 맵을 무시하도록 지정
function IOCTL_IOPMD_QUERY_KIOPM: DWORD;             //커널 맵을 사용할 수 있게 한다.
```

- 예제 어플리케이션

예제로 제공된 PortTest 어플리케이션을 잘 분석해보면 쉽게 사용방법을 익힐 수 있을 것이다. 비교적 자세하게 작성된 좋은 예제이므로 이를 참고하기 바란다.

하드웨어 제어와 다이렉트 X

지금까지 포트 제어를 이용한 하드웨어를 접근하는 방법에 대해서 알아보았다. 여기까지 소개한 방법은 비교적 과거의 도스 때와 마찬가지로 low-level 에서 접근하여 하드웨어를 제어하는 방법 들이다.
윈도우 프로그래밍 환경에서 아직도 이런 방식으로 하드웨어에 직접 접근하고자 하는 필요성이 대두되는 것은 아마도 속도의 문제와 윈도우라는 운영체제의 특성을 깰 수 없다는 점이 가장 문제가 될 것이다. 예를 들어, 조이스틱을 제어하려면 MMSystem.pas 유닛을 uses 절에 추가하고 적당한 조이스틱 제어 API 함수를 호출하여 사용할 수 있지만 윈도우라는 제한에 걸리게 되며, 개발자가 원하는 정도로 폭넓은 지원을 API 가 제공하지 않을 경

우 난관에 빠질 수 밖에 없다. 보통 하드웨어를 직접 제어할 때에는 도스에서와 마찬가지로 현재의 프로그램이 모든 권한을 쥐고서 사용할 수 있기를 바라지만, 윈도우 API 는 여러 윈도우의 공유 관계를 우선시 하기 때문에 이런 프로그래머의 욕구와는 배치되는 면이 많다. 그렇지만, 이런 방식으로 직접 하드웨어에 접근하는 것은 그다지 바람직하지 못한 방법이다. 이런 문제를 해결하기 위해서 마이크로소프트에서 해결책으로 제시한 것이 바로 다이렉트 X 이다. 물론 모든 종류의 하드웨어에 다 적용되는 것은 아니지만, 가장 흔히 사용되는 하드웨어를 바탕으로 하여 비교적 성공적으로 정착되어 나가고 있다.
델파이에서도 게임 프로그래밍을 하는 많은 그룹들이 생겨나고 있으며, 이들을 중심으로 새로운 다이렉트 X 버전이 발표될 때마다 파스칼 버전의 유닛을 번역하여 제공하는 여러 사람들도 생기고 있으며, 다이렉트 X 를 클래스로 잘 포장하여 쉽게 사용할 수 있도록 제공하는 프리웨어 컴포넌트도 많이 등장하고 있다.
그중에서도 DGC(Delphi Game Creator)와 델파이 X 가 가장 유명하다. CD-ROM 에 가장 최신 버전의 델파이 X 컴포넌트와 다이렉트 X 유닛을 같이 제공하므로 이를 이용하여 다이렉트 X 의 기능을 즐겨보기 바란다. 좋은 예제도 많이 실려있으므로 이를 이해하는 것은 그다지 어렵지 않을 것이다.
다이렉트 X 에 대한 부분 역시 따로 책을 한 권 구성하여 설명해도 모자랄 정도로 다루고 있는 범위가 광범위하다. 그렇기 때문에, 여기서는 다이렉트 X 에 대한 간단한 길잡이 정도로 다루고 넘어가고자 한다. 기회가 닿는다면 웹이나 하이텔 등의 통신망을 통해 다이렉트 X 강좌를 게시할 계획을 가지고 있으므로 많은 기대 바란다.
다이렉트 X 는 현재 6.0 버전이 출시될 준비를 하고 있으며, 5.0 버전이 가장 일반화되어 많이 사용되고 있다. 다이렉트 X 는 프로그래머가 윈도우 환경에서 하드웨어에 직접 접근할 수 있도록 해주는 API 의 세트이다. 다이렉트 X 의 강점은 윈도우 환경에서의 표준 장치에 접근하는 장점을 채용하여 각각의 하드웨어 별로 따로 프로그래밍이 필요하지 않도록 공통된 API 를 제공하면서도 여기에 최적화된 하드웨어를 제조업체에서 제작하도록 유도함으로써 그 수행 성능의 향상을 같이 유도했다는 점이다.

- 다이렉트 X 의 목적

다이렉트 X 가 처음 등장한 이유는 WinG 의 후속으로 윈도우 95 를 게임 플랫폼으로도 확장시키고자하는 의도에서 였다. 돌이켜 보면 게임에 PC 산업에 미친 영향은 막대한데, 그 중에서도 비디오와 사운드를 비롯한 각종 하드웨어의 고급화를 부추긴 결정적인 역할을 했다. 에를 들어, EGA 에서 VGA 이상급으로 비디오 카드가 고급화하고 사운드 블라스터를 위시한 각종 사운드 카드가 일반화한 결정적인 계기가 바로 게임이다. 아마도 IBM PC 의 고전 게임들을 기억하는 독자들은 페르시아 왕자라는 게임이 사운드 블라스터를 전세계적인 히트 상품으로 이끈 장본인이라는 것 정도는 잘 알고 있을 것이다.
그런데, 윈도우 95 의 초기에는 이러한 게임 들의 높은 시스템 요구사항을 따라가기에 윈도

우 95 는 너무나 느린 플랫폼이었다. 이렇게 된 가장 주된 원인이 바로 하드웨어에 대한 직접 접근이 봉쇄된 것이다. 어쩔 수 없이 많은 게임 개발자들은 도스 환경에서의 게임 개발을 계속할 수 밖에 없었고, 불과 1 년전만 해도 대부분의 게임이 도스 환경에서 돌아가도록 제작되었다. 이로 인해 게임을 즐기는 많은 윈도우 95 의 사용자들은 게임을 위해서 도스 환경을 버릴 수 없는 상태가 되었고, 마이크로소프트 역시 향후 운영체제 시장의 발전을 위해서 이러한 형태는 결코 자사에 이익이 되지 않는다는 것을 간파하였다.
다이렉트 X 는 이렇게 탄생했지만, 멀티미디어가 어플리케이션의 주류로 등장하면서 다이렉트 X 의 필요성은 더욱 커져만 간다. 그리고, IE 4.0 으로 대별되는 후기 윈도우 95 와 윈도우 98 에서는 다이렉트 X 가 기본적인 컴포넌트로 운영체제에 설치되면서 다이렉트 X 는 기본 운영체제 환경으로 자리를 잡았다.
현재의 IE 4.0 이상 환경에서 제공되는 다이렉트 X 컴포넌트는 DirectDraw, DirectSound, Direct3D, DirectShow, DirectAnimation 의 5 가지로 구성된다.

- 다이렉트 X 의 기초

다이렉트 X 는 기본적으로 서로 다른 하드웨어 종류에 따라 다른 컴포넌트로 구성되었다. 하드웨어와의 인터페이스는 크게 2 가지인데, 소프트웨어와 하드웨어 사이에 위치하여 개발자가 하드웨어의 종류에 관계없이 사용할 수 있는 중간층 역할을 하는 HAL(Hardware Abstraction Layer)와 하드웨어적으로 지원되지 않는 기능을 소프트웨어적으로 보완하여 구현하는 HEL(Hardware Emulation Layer)로 나누어볼 수 있다. HEL 이 적용되는 가장 흔한 부분은 3D 를 지원하는 부분이다.
또한, 다이렉트 X 는 기본적으로 액티브 X 와 같은 컴포넌트 기술에 기반하고 있기 때문에 쉽게 접근하여 사용할 수 있다.

- DirectDraw

DirectDraw 는 그래픽 카드에 직접 접근하는 방법을 개발자들에게 열어 줌으로써 게임 프로그래밍과 같은 다이렉트 X 를 사용하는 어플리케이션의 활성화를 가져온 가장 대표적인 컴포넌트이다. DirectDraw 를 이용해 그래픽의 속도가 빨라졌으며, 이로 인해 윈도우 95 를 본격적인 게임 플랫폼으로 사용할 수 있는 길이 열렸다.
DirectDraw 는 HAL 과 HEL 을 모두 이용하여 실제로 하드웨어를 직접 접근할 때 HAL 을 사용하고, 비록 지원되지 않는 기능을 사용하더라도 HEL 을 이용하여 소프트웨어 적으로 이를 보완하기 때문에 개발자는 표준적인 개발환경을 이용할 수 있게 되었다.
DirectDraw 의 기능을 단적으로 설명하자면 비디오 메모리 관리자라고 할 수 있다. DirectDraw 는 비디오 메모리를 DirectDraw, DirectDrawSurface, DirectDrawPalette, DirectDrawClipper 라는 4 개의 객체를 통해 인터페이스한다.

DirectDraw 객체는 하드웨어 장치를 대표하는 기본적인 객체로 어플리케이션을 전체화면이나 윈도우 모드로 실행될 수 있도록 설정이 가능하며, 경우에 따라서는 다른 프로그램이 접근하지 못하도록 하드웨어 장치를 배타적으로 사용할 수 있도록 지정할 수도 있다. 또한, 이 객체를 이용하여 비디오 해상도의 조절이 가능하다.
DirectDrawSurface 객체는 개발자가 메모리에 저장된 그래픽에 접근할 때 사용된다. Surface 라는 것은 현재 화면에 나타나는 주 surface(primary surface)와 다음에 화면에 나타날 백 버퍼(back buffer), 큐에 들어가려고 대기하는 off-screen surface 로 나눌 수 있다. 화면이 움직이는 것은 off-screen-surface 들이 백 버퍼로 들어가고, 백 버퍼의 surface 가 주 surface 로 전환되면서 이루어지는 것이다.
DirectDrawPalette 객체는 각 surface 와 surface 사이에 독자적인 256-색상의 팔레트를 적용하거나, 공유 팔레트를 사용하도록 지정할 수 있으며, DirectDrawClipper 객체는 GDI 를 건너뛰어 그래픽을 처리함으로써 그래픽 처리 속도의 향상을 가져올 수 있다.

- DirectSound, DirectInput

DirectSound 는 사운드와 음악 하드웨어에 대한 인터페이스를 담당한다. 보통 .wav 파일이 표준으로 이용되는데 이때 1 차, 2 차, 정적, 스트림 버퍼를 이용하게 된다. 1 차 버퍼(primary buffer)는 현재 컴퓨터에 의해 사용되고 있는 것으로 2 차 버퍼(secondary buffer)에서 사운드 파일을 가져온다. 작은 파일 들은 정적 버퍼(static buffer)에 위치했다가 빠르게 접근이 가능하며, 큰 파일들은 스트림 버퍼(stream buffer)를 이용하여 그때 그때 부분적으로 접근하여 사용하게 된다.
DirectInput 은 조이스틱을 비롯한 각종 게임용 입력 장치를 다룰 수 있는 인터페이스를 제공한다. DirectInput 을 이용하여 최고 16 개의 조이스틱과 32 버튼, 6 가지 축을 지원할 수 있다. 조이스틱 이외에도 그래픽 타블렛과 같은 장치를 입력장치로 사용할 수 있다.
원래 이번 장을 계획하면서 MMSystem.pas 유닛에 정의된 API 를 이용하여 조이스틱을 사용하는 방법에 대해서 다루려고 했으나, 현재의 대세는 다이렉트 X 의 DirectInput 을 이용하여 해결하는 것이기 때문에 따로 다루지 않았다.

- 미디어와 컴포넌트 레이어

앞에서 설명한 DirectDraw, DirectSound, DirectInput 을 기초 레이어라고 한다면, 이 위에는 미디어 레이어가 위치한다. 미디어 레이어에서는 개발자가 기초 레이어에서의 여러 컴포넌트들을 이용하여 진정한 멀티미디어를 구현할 수 있도록 해준다. 미디어 레이어보다 더 상위 레벨의 레이어가 바로 컴포넌트 레이어인데, 컴포넌트 레이어는 미디어 레이어와 기초 레이어의 컴포넌트를 조합하여 실제로 사용될 수 있는 컴포넌트 형태로 제공된다. 가장 대표적인 컴포넌트 레이어로는 NetMeeting, NetShow, ActiveMovie 등을 들 수 있겠다.

미디어 레이어에는 DirectShow, DirectModel, DirectAnimation, DirectPlay, Direct3D Retained Mode, VRML 등이 있다. DirectShow 는 퀵타임, avi, wav, mpeg 등의 오디오와 비디오 스트림을 재생할 수 있도록 해준다. 이를 구현하기 위해 많은 필터를 필터 그래프 관리자(filter graph manager)를 통해 데이터 스트림에 연결하는데, 이를 구현하여 새로운 형식의 비디오 스트림을 만들어볼 수도 있다.
DirectModel 은 커다란 3 차원 그래픽 객체에 대한 렌더링과 상호작용 수단을 제공하며 DirectAnimation 은 개발자가 사운드, 동영상, 텍스트를 조합하여 이들을 시간에 맞도록 통합하여 관리할 수 있는 애니메이션 작업을 지원한다. DirectPlay 는 온라인 게임과 네트워크에 대한 지원을 강화하여 플레이어 간의 메시지를 주고 받는 등의 상호작용이 가능하다. 최근의 네트워크 지원 상황을 감안할 때 많은 개발자 들이 접근하고자 하는 부분 중에 하나이다. 마이크로소프트의 인터넷 게임존(www.zone.com)에서는 DirectPlay 를 완벽하게 지원하고 있다.

정 리 (Summary)

이번 장에서는 하드웨어에 접근하여 여러가지 작업을 하는 방법에 대해서 알아보았다. 최근의 윈도우 환경에서는 다이렉트 X 를 이용하여 하드웨어를 다루는 방법이 거의 표준으로 굳어지고 있으며, 예제도 많고 비교적 향상된 수행 성능을 보여주고 있기 때문에 그 활용 가능성은 밝다고 말할 수 있겠다.
델파이를 이용해서도 다이렉트 X 를 마음대로 조작할 수 있으며, 잘 만들어진 클래스도 많이 제공되고 있으므로 많은 개발자들이 여기에 관심을 가지고 다이렉트 X 의 발전된 기능을 어플리케이션에 도입하여 사용할 수 있기를 고대한다.