Win32 쉘 확장 기법의 활용

(Using Win32 Shell Extensions)

윈도우 95 로 들어오면서, 각종 파일에 대한 접근과 운영체제에서 쉘의 역할을 하는 부분이 MS-DOS 시절과 많은 변화를 보여주고 있다.
이번 장에서는 COM 기술에 기반한 Shell 인터페이스의 사용 방법을 익히고, 이를 사용해서 유용한 어플리케이션을 제작해 보도록 한다.
Shell 확장 기법을 사용하기 위해서 델파이에서는 ShlObj.pas 유닛을 제공한다. 여러가지 쉘 확장 기법이 존재하지만 대표적인 것으로는 namespace 확장과 컨텍스트 메뉴 핸들러, 아이콘 핸들러와 파일 뷰어에 대한 확장 등을 들 수 있다.

폴더와 뷰 (Folder and View)

윈도우 탐색기는 2 개의 pane 으로 나뉘어진 창을 가지고 있다. 각각의 뷰는 객체와 그 내용에 대한 사용자 인터페이스를 표현한다.
좌측 창은 사용자에게 객체의 위치에 대해 Win32 트리뷰 컨트롤을 사용해서 Shell 에 있는 모든 다른 객체에 대한 상대적인 위치를 나타내 보여줌으로써 표현하고 있다. 사용자는 좌측 창을 보면서 현재 선택된 객체가 어떤 객체의 자손인지, 어떤 객체를 포함하고 있는지 한눈에 파악할 수 있다.
과거의 윈도우 3.1 의 파일 관리자를 생각해 보자. 이 때에는 좌측 창이 디렉토리 트리를 나타내는 것이었다. 보기에는 파일 관리자의 좌측 창과 크게 다르지 않아 보이지만, 윈도우 탐색기에서는 디렉토리 이외에 제어판 정보나 프린터 폴더 등의 수 많은 다른 객체 들에 대한 정보를 포함하고 있다.
우측 창에는 좌측 창에서 선택한 아이템에 대한 세부 내용을 표시하게 되어 있다. 현재 디렉토리를 탐색하고 있다면 디렉토리의 내용을 볼 수 있을 것이고, 제어판 등의 객체를 탐색하고 있다면 제어판에 속한 객체를 볼 수 있을 것이다.
이렇게 윈도우 탐색기를 이용해서 탐색을 할 수 있는 폴더 객체에는 몇 가지 종류가 있다. 가장 기본적이고 전통적인 하드 디스크의 디렉토리를 나타내는 폴더 객체가 있을 것이고, 제어판이나 네트워크 정보 등의 가상 폴더가 존재할 것이다. 전통적인 하드 디스크의 디렉토리에 대한 폴더는 쉘(shell) 자체에 의해서 구현되며, 다른 종류의 가상 폴더 들은 쉘 확장(Shell extension) COM 객체를 통해 구현된다.
이런 가상 폴더 들에 대한 올바른 명칭은 'namespace extensions' 이며, 개발자 들은 이들을 커스텀 쉘 확장 COM 객체를 제작함으로써 생성할 수 있다. 이렇게 만든 쉘 확장 객체는 반드시 DLL 로 포장해야 하며, 최소한의 IUnknown, IShellExtInit 인터페이스는 구현해

주어야 한다.
이런 namespace extension 객체는 두가지의 메인 컴포넌트로 구성되는데, 이들이 각각 폴더(folder)와 뷰(view) 객체이다. 이들 COM 객체 들은 각각 IShellFolder 와 IShellView 인터페이스를 반드시 구현해야 한다. 그러므로, 결국 커스텀 쉘 확장 COM 객체를 만들려면 폴더 객체는 IUnknown, IShellExtInit, IShellFolder 를 구현해야 하며, 뷰 객체는 IUnknown, IShellExtInit, IShellView 인터페이스를 구현해야 한다.

아이템 ID (Item Identifiers)

먼저 폴더 객체가 어떻게 윈도우의 쉘과 상호작용 하는지 알아 보자. 탐색기의 좌측 창은 쉘의 namespace 내에 있는 객체 들의 위치 정보를 대표하는 것이다. 그러므로, 각 객체 들은 자신의 위치를 정확하게 표현해야 하는데, 이를 위해 반드시 필요한 것이 아이템 ID(identifier) 이다.
이때 폴더가 나타내는 것은 디렉토리뿐 아니라 제어판, 네트워크 정보, 프린터 폴더 등일 수도 있다. 그렇기 때문에 폴더의 위치를 나타내는데 디렉토리의 경로를 사용할 수 없다.
마이크로소프트에서는 개발자가 개별적인 컴포넌트에 대해서 독특한 형식을 정의할 수 있도록 했는데, 이러한 구조체를 아이템 ID 라고 한다. 아이템 ID 의 구조는 다음과 같이 정의할 수 있다.

```
PSHItemID = ^TSHItemID;
TSHItemID = packed record
  cb: Word;                    //ID 의 Size
  abID: array[0..0] of Byte;   //아이템 ID (길이는 변화 가능)
end;
```

이 선언부를 살펴 보면 abID 에 대한 구체적인 정의가 없는 것을 알 수 있다. 그렇기 때문에, 자신의 객체에 대해 유일한 경로에 있는 다른 branch 를 적절하게 나타내기 위해서 어떠한 형태의 이진 데이터도 사용할 수 있는 것이다.
이러한 아이템 ID 는 독립적으로 사용하기 보다는, 아이템 ID 리스트(Item Identifier List)라고 불리는 데이터 구조로 사용한다. 보통 하나의 아이템 ID 리스트는 아이템 ID 세트 하나를 포함한다. 이때 목록의 마지막 아이템의 cb 파라미터는 0 을 값으로 가지게 된다. 이런 리스트를 이용해서 특정 객체를 윈도우 쉘의 namespace 에서 나타낼 수 있게 된다.
폴더 객체는 쉘의 namespace 내에서 어디에 존재하는지를 자신이 직접 추적할 필요가 없다. 객체가 구성되기 전에 쉘은 어디에서 이 객체를 찾을 것인지 미리 알아야 하고, 폴더 객체를 생성하고 나면, 쉘은 그 객체의 IPersistFolder 인터페이스를 호출해서 위치를 알아내고 이 값을 아이템 ID 리스트에 전달한다.

IShellFolder 인터페이스를 구현한 쉘 extension 이 아이템 ID 를 만들어낼 때, 아이템을 확인하기 위한 기본적인 데이터 뿐만 아니라 다른 기능을 구현할 때 도움이 되는 추가적인 정보를 기록한다. 예를 들어, 쉘의 파일 시스템 아이템들의 IShellFolder 구현 부분은 기본적으로 아이템 들을 확인하기 위해서 파일과 디렉토리에 대한 긴 이름을 저장하며, 동시에 짧은 이름과 크기, 날짜 등의 데이터도 기록한다.
아이템 ID 리스트가 SHGetPathFromIDList 등의 쉘 API 함수의 파라미터로 넘겨질 때에는 언제나 name space 의 root(데스크탑 폴더)에서부터의 경로를 사용해야 한다. 그에 비해서 IShellFolder 멤버 함수에 파라미터로 사용될 때에는 폴더로부터의 상대적 경로를 사용한다.

주요 인터페이스와 API

그렇다면 Win32 쉘에 의해 지원되는 주요 인터페이스와 API 에 대해서 간단히 알아보도록 하자.

- 작업 할당자 (Task allocator) API

모든 쉘 extension 은 메모리 객체(보통은 아이템 ID 리스트)를 할당하거나 해제할 때 반드시 작업 할당자를 사용해야 한다. 작업 할당자에 접근하는 방법에는 쉘 extension 이 Ole32.dll 과 링크되어 있는지 여부에 따라 두가지 방법이 존재한다.

1. 쉘 extension 이 OLE API 를 호출하는 경우(OLE32.DLL 과 링크되는 경우)에는 CoGetMalloc API 를 이용해서 Ole 의 작업 할당자를 호출해야 한다.
2. OLE API 를 호출하지 않는 경우에는 다음에 선언되어 있는 쉘 작업 할당자 API 를 호출해야 한다.

```
function SHGetMalloc(var ppMalloc: IMalloc): HResult; stdcall;
```

- IContextMenu 인터페이스

IContextMenu 인터페이스는 다음과 같은 세가지 경우에 사용된다.

1. 쉘이 컨텍스트 메뉴 extension 을 로드할 때

사용자가 쉘의 name space(파일, 디렉토리, 프린터, 워크 그룹 등)에서 오른쪽 마우스 버튼을 클릭하면 그 객체 형에 대한 디폴트 컨텍스트 메뉴가 생성되고, 레지스트리에 등록되어 있는 컨텍스트 메뉴 extension 이 로드된다. 여기에 추가적인 메뉴 아이템이 있으면 추가

된다. 이러한 컨텍스트 메뉴 extension 은 HKEY_CLASSES_ROOT₩beginProgIDend₩shellex₩ContextMenuHandlers 에 등록되어 있다.

2. 쉘이 확장 name space 에 있는 서브 폴더의 컨텍스트 메뉴를 가져올 때

탐색기의 name space 가 name space extension 에 의해 확장된 경우, 쉘은 IShellFolder.GetUIObjectOf 메소드를 호출하여 IContextMenu 객체를 불러온다.

3. 쉘이 디렉토리에 대한 논-디폴트 drag-and-drop 핸들러를 로드할 때

사용자가 논-디폴트 drag-and-drop 을 파일 시스템 폴더(디렉토리 등)에 수행할 때, 쉘은 레지스트리 HKEY_CLASSES_ROOT₩beginProgIDend₩shellex₩DragDropHandlers 키에 등록된 쉘 extension 을 로드한다.

- IFileViewer 인터페이스

FileViewer 컴포넌트 객체에 의해 구현되는 인터페이스이다. 파일 이름은 IPersistFile 인터페이스를 통해 뷰어에 전달된다.

- IShellBrowser/IShellView/IShellFolder 인터페이스

이들 인터페이스는 쉘이 name space extension 과 통신을 할 때 사용된다. 쉘은 IShellBrowser 인터페이스를 제공하며, extension 들은 IShellFolder 와 IShellView 인터페이스를 구현한다.

1. 커맨드/메뉴 아이템 ID

탐색기는 WM_COMMAND 메시지가 커맨드/메뉴 아이템 ID 들 범위 안에 있으면 이를 디스패치한다. 모든 메뉴 아이템의 ID 는 FCIDM_SHVIEWFIRST/LAST 범위에 있어야 하며, 그렇지 않으면 이들을 디스패치하지 못한다.

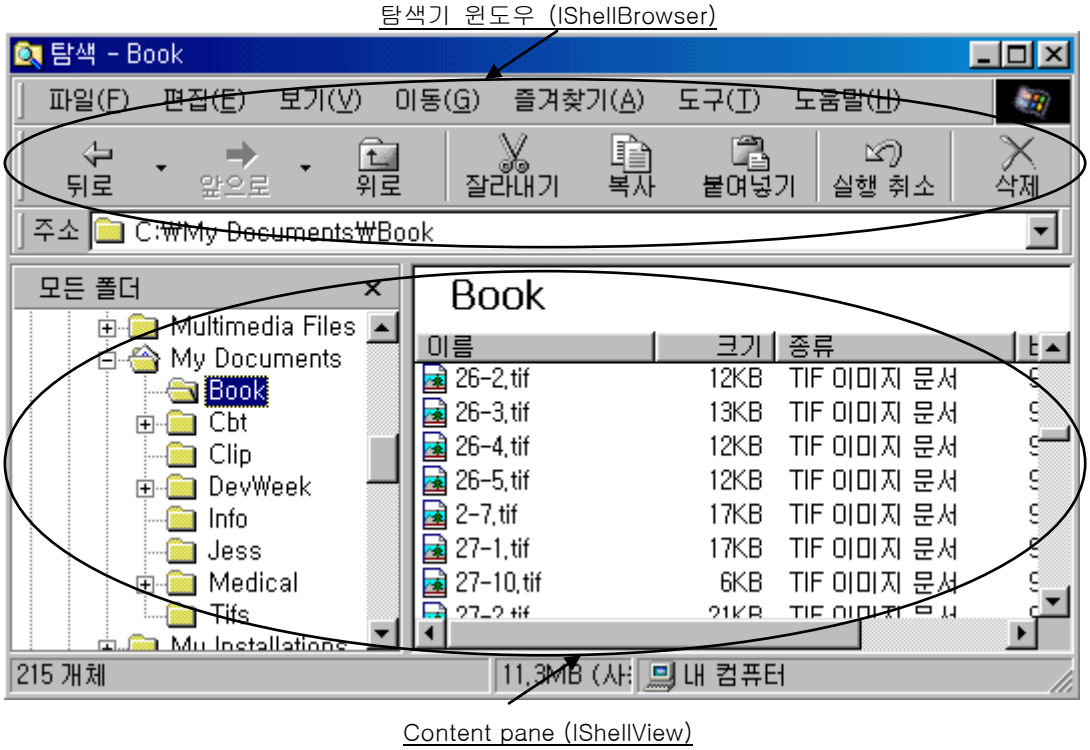
FCIDM_SHVIEWFIRST/LAST: IShellVIew 의 범위 (우측 pane)
FCIDM_BROWSERFIRST/LAST: 탐색기 프레임의 범위 (IShellBrowser)
FCIDM_GLOBAL/LAST: 탐색기 서브 메뉴 IDs

2. FOLDERSETTINGS

FOLDERSETTINGS 는 사용자가 브라우즈할 때에 탐색기가 하나의 폴더 뷰에서 다른 폴더 뷰로 넘어갈 때 전달되는 데이터 구조체이다. 현재의 설정을 얻기 위해서 ISV.GetCurrentInfo 메소드를 호출하며, 이를 ISV.CreateViewWindow 메소드에 넘겨서 다음 폴더 뷰가 이를 상속받을 수 있게 한다.

3. IShellBrowser 인터페이스

IShellBrowser 인터페이스는 쉘 탐색기/폴더 프레임 윈도우에 의해 제공된다. 이 인터페이스가 쉘 폴더에 대한 contents pane 을 생성할 때 (이것은 IShellFolder 인터페이스를 제공한다.), IShellView 객체를 생성하기 위해 IShellFolder 의 CreateViewObject 메소드를 호출하게 된다. 그리고 나서, IShellView 의 CreateViewWindow 메소드를 호출하여 contents pane 윈도우를 생성한다. IShellBrowser 인터페이스에 대한 포인터는 CreateViewWindow 를 호출할 때 IShellView 객체에 포인터로 넘겨진다.



Namespace

Namespace 에는 두 가지 종류가 있다. 하나는 쉘이 관리하는 표준 namespace 이고, 다른 하나는 개발자가 직접 생성한 커스텀 namespace 이다.

이때 커스텀 namespace 의 가장 상위 레벨의 객체 만이 표준 namespace 에 자동으로 나타난다. 이렇게 하려면 다음의 두 가지 방법 중 하나를 사용해야 한다.

- 표준 namespace 에서 디렉토리를 하나 생성하고 폴더 객체의 CLSID(클래스 ID)를 파일 이름 확장자의 형태로 부여한다. 예를 들면 Custom Namespace.{00000000-0000-0000-0000-000000000000}와 같은 형태가 된다.

- 레지스트리의 다음과 같은 키의 엔트리를 생성한다.
  HKEY_LOCAL_MACHINE₩Software₩Microsoft₩Windows₩CurrentVersion₩Explorer₩Desktop₩Namespace.

각각의 namespace 객체는 자신의 프로퍼티와 자식 객체 들이 프로퍼티 값을 어떻게 얻을 것인지에 대해서 알고 있어야 한다. 윈도우 탐색기는 사용자가 해당 namespace 확장자 폴더를 호출하면 (폴더 아이콘을 더블 클릭하는 등) 커스텀 namespace 에 있는 자식 객체를 보여주라고 요구하게 된다.
일단 커스텀 namespace 의 루트를 윈도우 탐색기에 추가하면 탐색기에 보이게 된다. 이 폴더를 탐색하려고 하면 탐색기는 쉘의 일반적인 인터페이스에 입각해서 사용자에게 내용을 보여주세 된다. 사용자가 폴더를 더블 클릭하거나 좌측의 '+' 기호를 클릭하면 탐색기는 선택된 폴더의 모든 자식 폴더의 목록을 보여주게 되어 있다. 이를 위해서 탐색기는 폴더 객체의 IShellFolder.EnumObjects 메소드를 호출한다.
또한, 사용자가 특정 폴더를 클릭하면 탐색기는 사용자에게 폴더 객체의 내용을 보여주게 되어 있다. 이를 위해서는 내용을 보여주기 전에 다음의 두 가지가 선행되어야 한다.

- 폴더 객체를 생성한다. 탐색기는 반드시 선택된 폴더 객체의 부모를 먼저 생성하고, 그 객체의 IShellFolder.BindToObject 메소드를 호출한다.
- 뷰 객체를 생성한다. 일단 선택된 폴더 객체가 생성되면 탐색기는 그 객체의 IShellFolder.CreateViewObject 메소드를 호출한다.

폴더와 뷰의 상호작용

사용자가 폴더를 클릭하면 탐색기는 폴더에 대한 뷰 객체를 생성해야 한다. 이 작업이 IShellFolder.CreateViewObject 를 호출하여 이루어 진다는 것은 앞에서 간단히 설명했다.
일단 뷰 객체가 생성되면 어떤 종류의 뷰 객체가 생성될 것인지 결정해야 한다.
뷰의 종류에는 크게 두 가지가 있다.
하나는 폴더 내부에 아이템을 포함하는 팝업 뷰 윈도우(popup view window)이다. 이 뷰는 사용자가 팝업 윈도우에서 폴더의 아이템을 더블 클릭 할 때 볼 수 있는 뷰이다. 또한, 사

용자가 탐색기에서 오른쪽 버튼을 클릭해서 해당 폴더에 대해서 '열기(O)'를 선택했을 때 볼 수 있다.
다른 하나는 탐색기에 의해서 만들어지는 것으로, 사용자가 탐색기의 좌측 창에서 폴더를 더블 클릭하면 폴더의 내용이 우측 창에 나타나게 되어 있다. 이런 동작은 좌측 창에서 오른쪽 버튼을 클릭해서 '탐색(E)'를 선택했을 때와 같은 것이다.
이 두 가지 경우에 있어서, 폴더 객체는 IShellView.CreateViewWindow 메소드를 호출해서 뷰 객체를 생성한다. 즉, 어떤 뷰 모드를 선택할 것인지 여부는 전적으로 뷰 객체가 CreateViewWindow 를 어떻게 구현할 것인지에 달려 있다.
마지막으로 폴더와 뷰의 관계에 있어서 알아야 할 것은 하나의 폴더 객체에 대해서 많은 수의 뷰 객체가 있을 수 있다는 것이다. 그러므로, 각각의 뷰 객체와 폴더 객체는 반드시 분리된 COM 객체로 구현되어야 한다. 이들에 대한 동기화 작업은 윈도우의 쉘이 알아서 하게 된다.
쉘 확장에 관한 내용 중에서 가장 복잡하고도 어려운 것이 namespace 와 파일 뷰어를 확장해서 구현하는 것이다. 여기에 대해서 구현하는 것은 이 책의 범위를 넘게 되므로 예제를 작성하지는 않는다. 그렇지만, 궁금한 독자들은 인터넷 상에 꽤 많은 컴포넌트와 예제가 소스와 함께 제공되므로 이들을 찾아보기 바란다.

참고: out 키워드

out 키워드는 델파이 3 에서 추가된 것으로 컴파일러에게 함수를 호출하기 전에 몇 가지 추가적인 코드를 생성하게 한다. 다음의 코드를 살펴보자.

```
var
  malloc: IMalloc;
begin
  GetAnotherMallocOut( malloc );
  GetAnotherMallocVar( malloc );
end;
```

앞에서 호출한 두가지 함수의 선언부는 다음과 같다.

```
procedure GetAnotherMallocOut( out malloc: IMalloc );
procedure GetAnotherMallocVar( var malloc: IMalloc );
```

즉, 하나는 out 키워드를 하나는 var 키워드를 사용한 것이다.
앞의 함수 호출을 디버거를 이용해서 컴파일러가 생성해낸 코드를 살펴보면 다음과 같다.

```
GetAnotherMallocOut( malloc );
        lea     eax,[ebp-04]
        call    @IntfClear
        mov     edx,eax
        mov     eax,ebx
        call    TForm1.GetAnotherMallocOut

GetAnotherMallocVar( malloc );
        lea     edx,[ebp-04]
        mov     eax,ebx
        call    TForm1.GetAnotherMallocVar
```

이를 살펴 보면 out 파라미터를 사용한 경우에는 컴파일러가 IntfClear 라는 함수를 호출하는 코드가 추가 되었다는 것을 알 수 있다. 이 함수는 System.pas 유닛에 선언되어 있는 것으로 현재 넘겨지는 변수에 COM 인터페이스가 대입되어 있는지 알아보고, 그렇다면 본 함수를 호출하기 전에 Release 를 호출하게 된다. 이렇게 하면 COM 인터페이스를 사용할 때 흔히 생기기 쉬운 참조 계수(reference count)의 혼란 문제를 해결할 수 있다. 비록 사소해 보이지만, 문제가 생겼을 때에는 대단히 어려울 수도 있는 부분을 해결해 준다.

쉘 링크 (Shell Link)

쉘 확장 인터페이스 중에서 이번에는 IShellLink 인터페이스를 이용해서 어플리케이션의 쉘 링크를 생성하는 방법에 대해서 알아보도록 하자.
다른 쉘 확장 기법을 사용할 때와는 달리 IShellLink 인터페이스는 윈도우 쉘에 의해 구현되므로 따로 구현할 필요는 없고, 단지 다음과 같이 CoCreateInstance 함수를 이용하여 인스턴스를 생성하면 된다.

```
var
  SL: IShellLink;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
    IShellLink, SL));
  ...
end;
```

참고로 기억해야할 것은, OLE 함수를 사용하기 전에 CoInitialize 함수를 호출하여 COM 라

이브러리를 초기화할 필요가 있으며, COM 을 모두 사용한 뒤에는 CoUninitialize 함수를 호출하여야 한다.
쉘 링크는 데스크 탑에서 오른쪽 버튼을 클릭해서 단축 아이콘을 생성할 수 있는데, 이렇게 생성된 쉘 링크는 실제로는 .lnk 확장자를 가지는 파일이다. 윈도우가 시작되면 .lnk 파일에 대한 디렉토리를 검사한다. 쉘은 링크와 폴더의 관계를 시스템 레지스트리에 기록하게 된다 (보통 HKEY_CURRENT_USER₩Software₩Microsoft₩Windows₩Current Version ₩Explorer₩Shell Folders 키에 위치한다).
쉽게 말해서 쉘 링크를 특정 폴더에 생성하는 것은 링크 파일을 특정 디렉토리에 위치시키는 것과 같은 의미이다. 이때 레지스트리에 직접 접근하지 않고, SHGetSpecialFolderPath 함수를 이용하여 특정 폴더에 대한 디렉토리 패스를 얻는 것이 중요하다. 이 함수의 선언부는 다음과 같다.

```
function SHGetSpecialFolderPath(hwndOwner: HWND; lpszPath: PChar;
  nFolder: Integer; fCreate: BOOL): BOOL; stdcall;
```

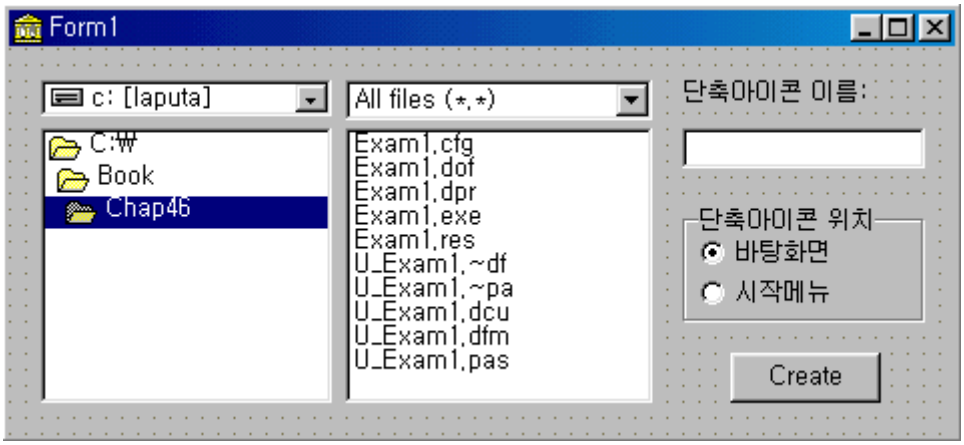
여기서 hwndOwner 파라미터는 윈도우의 핸들을 나타내며, lpszPath 파라미터에는 패스를 기록할 버퍼의 포인터를 나타낸다. nFolder 파라미터는 패스를 얻고자 하는 특정 폴더를 나타내며, 마지막으로 fCreate 파라미터는 폴더가 존재하지 않을 때 폴더를 생성할 것인지 여부를 결정한다.
여기서 nFolder 파라미터에 지정할 수 있는 값으로는 여러가지가 있는데, 이것에 대한 자세한 내용은 도움말이나 MSDN 등의 내용을 참고하기 바란다.
IShellLink 인터페이스를 구현하기 위해서는 IPersistFile 인터페이스를 지원해야 하는데, 이는 파일 접근이 필요하기 때문이다. IPersistFile 인터페이스는 디스크에 접근할 수 있는 메소드를 제공한다.
IShellLink 를 구현하는 클래스는 IPersistFile 인터페이스를 구현해야 하므로 as 연산자를 사용해서 다음과 같이 인스턴스를 얻을 수 있다.

```
var
  SL: IShellLink;
  PF: IPersistFile;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
    IShellLink, SL));
  PF := SL as IPersistFile;
  ...
end;
```

다음의 코드는 메모장에 대한 데스크탑 쉘 링크를 생성한다.

```
procedure MakeNotepad;
const
  AppName = 'c:₩windows₩notepad.exe';
var
  SL: IShellLink;
  PF: IPersistFile;
  LnkName: WideString;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
    IShellLink, SL));
  PF := SL as IPersistFile;
  OleCheck(SL.SetPath(PChar(AppName)));        //링크 패스를 설정
  LnkName := GetFolderLocation('Desktop') + '₩' + ChangeExt(ExtractFileName(AppName), '.lnk');
  PF.Save(PWideChar(LnkName), True);           //링크 파일의 저장
end;
```

이 프로시저에서 IShellLink 인터페이스의 SetPath 메소드는 실행 파일이나 문서의 링크를 나타낼 때 사용된다. 그리고, 링크의 패스와 파일 이름은 GetFolderLocation 함수를 이용하여 얻어오게 되는데, ChangeFileExt 함수를 이용하여 .exe 확장자를 .lnk 확장자로 변경하여 사용한다. 마지막으로 Save 메소드는 디스크 파일에 대한 링크를 저장한다.
IShellLink 인터페이스의 정의를 보면 GetXXX, SetXXX 메소드가 여러 개 있는데, 이 메소드 들은 쉘 링크의 여러 필드의 값을 얻거나 설정하는데 사용된다.
그러면, 파일을 선택해서 그 파일을 바탕 화면이나 시작 메뉴에 등록하는 간단한 예제 어플리케이션을 만들어 보자. 시작 메뉴에는 'Sample'이라는 디렉토리를 만들어 여기에 단축 아이콘을 등록할 것이다.
먼저 폼에 TDriveComboBox, TDirectoryListBox, TFileListBox, TFilterComboBox 와 라벨과 에디트박스, 버튼 및 TRadioGroup 컴포넌트를 하나씩 폼에 얹고, 다음과 같이 디자인한다.



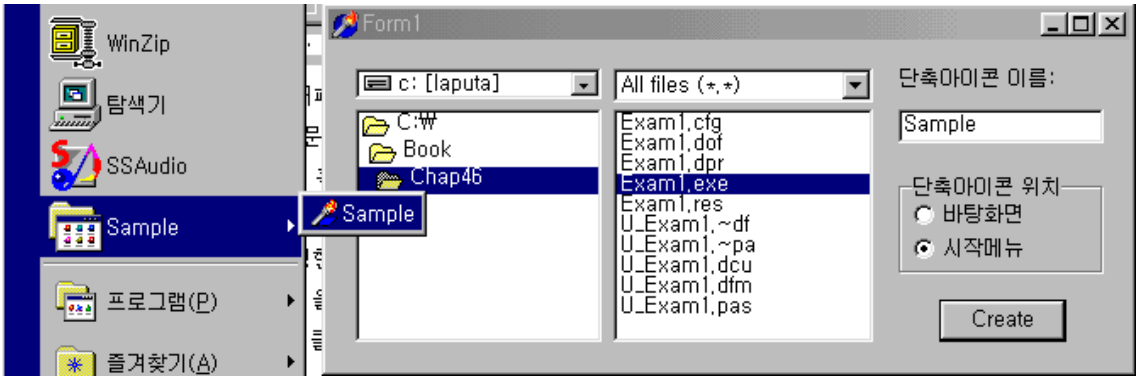
여기서 DriveComboBox1 의 DirList 프로퍼티는 DirectoryListBox1, DirectoryListBox1 과 FilterComboBox1 의 FileList 프로퍼티는 FileListBox1 으로 설정한다. 그리고, 필터로는 모든 파일, 실행 파일, DLL, 텍스트 파일의 4 종류로 설정한다.

이제 Button1 의 OnClick 이벤트 핸들러를 작성해서 Edit1 의 내용을 바탕화면이나 시작메뉴에 등록하도록 하면 된다. 이때 uses 절에 ShlObj.pas, ActiveX.pas, ComObj.pas, Registry.pas 유닛을 추가해야 한다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Unknown: IUnknown;
  ShellLink: IShellLink;
  PersistFile: IPersistFile;
  FileName, Directory: string;
  WideFileName: WideString;
  MyRegistry: TRegIniFile;
begin
  Unknown := CreateComObject(CLSID_ShellLink);
  ShellLink := Unknown as IShellLink;
  PersistFile := Unknown as IPersistFile;
  FileName := FileListBox1.FileName;
  with ShellLink do
  begin
    SetPath(PChar(FileName));
    SetWorkingDirectory(PChar(ExtractFilePath(FileName)));
  end;
  MyRegistry
```

```
    := TRegIniFile.Create('Software₩Microsoft₩Windows₩CurrentVersion₩Explorer');
  case RadioGroup1.ItemIndex of
    0: Directory := MyRegistry.ReadString('Shell Folders', 'Desktop', '');
    1:
    begin
      Directory := MyRegistry.ReadString('Shell Folders', 'Start Menu', '') + '₩Sample';
      CreateDir(Directory);
    end;
  end;
  WideFileName := Directory + '₩' + Edit1.Text + '.lnk';
  PersistFile.Save(PWChar(WideFileName), False);
  MyRegistry.Free;
end;
```

IShellLink 인터페이스와 IPersistFile 인터페이스는 이미 윈도우에 의해 구현되어 있으므로 특별히 따로 구현하지 않고, 이렇게 CreateComObject 함수를 이용하여 구현된 객체의 인스턴스를 이용하면 된다. CLSID_ShellLink 는 IShellLink 인터페이스를 구현한 CoClass 의 CLSID 이다. IShellLink 인터페이스를 구현한 CoClass 는 IPersistFile 인터페이스를 구현하고 있기 때문에 간단히 as 연산자를 이용하여 사용할 수 있다.
단축 아이콘을 지정하기 위해서 반드시 사용해야 하는 메소드로는 SetPath 메소드를 이용하여 단축 아이콘이 가리키게 될 파일의 패스를 지정하면 된다. SetWorkingDirectory 메소드를 이용해서는 단축 아이콘이 실행된 후의 디렉토리를 지정하므로 지정된 파일의 디렉토리로 설정하는 것이 좋다.
그리고, 레지스트리의 Software₩Microsoft₩Windows₩CurrentVersion₩Explorer 키의 내용이 중요한데, .lnk 파일을 저장하기 전에 실제로 바탕화면에 이를 저장할 것인지 아니면 시작메뉴에 저장할 것인지 여부는 'Shell Folders' 섹션에서 바탕화면의 경우는 'Desktop', 시작메뉴의 경우는 'Start Menu'의 값을 읽어와서 이를 이용하여 디렉토리를 결정하면 된다. IPersistFile 인터페이스의 Save 메소드는 PWideString 형의 문자열을 이용해야 하기 때문에, WideString 문자열로 선언된 WideFileName 변수에 디렉토리 이름과 에디트 박스의 내용을 이름으로 확장자 '.lnk'를 붙여서 저장한 후 이를 PWChar() 연산을 이용해 PWideString 문자열로 형변환하여 Save 메소드를 호출하면 된다.
컴파일하고, 실행한 뒤에 라디오 그룹에서 '시작메뉴'를 선택하고, 파일 리스트 박스에서 Exam1.exe 파일을 지정하고 단축 아이콘 이름으로 'Sample'을 지정하도록 하자. 그리고 'Create' 버튼을 클릭하면 다음과 같이 단축 아이콘이 만들어진 것을 볼 수 있을 것이다.



Copy Hook 핸들러

Copy hook 쉘 확장을 이용하면 폴더가 복사, 삭제, 이동, 변경될 때 이를 가로챌 수 있는 핸들러를 설치할 수 있다. 이를 이용하여 특정 작업을 할 수 없도록 할 수 있다.
Copy hook 핸들러를 만들기 위해서는 먼저 TComObject 클래스에서 상속한 클래스에 ICopyHook 인터페이스를 구현하도록 해야 한다. ICopyHook 인터페이스는 ShlObj.pas 유닛에 다음과 같이 정의되어 있다.

```
ICopyHookA = interface(IUnknown) { sl }
  [SID_IShellCopyHookA]
  function CopyCallback(Wnd: HWND; wFunc, wFlags: UINT; pszSrcFile: PAnsiChar;
    dwSrcAttribs: DWORD; pszDestFile: PAnsiChar; dwDestAttribs: DWORD): UINT; stdcall;
end;
```

즉, CopyCallback 메소드만 구현하면 되는 것이다. 이 함수는 쉘 폴더를 이용하여 작업을 할 때마다 호출되는 콜백 함수의 역할을 한다. 이 함수의 파라미터를 이해하는 것이 중요한데, Wnd 파라미터는 copy hook 핸들러가 사용할 부모 윈도우의 핸들을 지정하며, wFunc 파라미터는 폴더에 수행되는 작업의 종류를 지정한다. 여기에는 다음과 같은 것들이 있다.

값	의 미
FO_COPY ($2)	pszSrcFile 에 지정된 파일을 pszDestFile 에 지정된 위치에 복사한다.
FO_DELETE ($3)	pszSrcFile 에 지정된 파일을 삭제한다.
FO_MOVE ($1)	pszSrcFile 에 지정된 파일을 pszDestFile 에 지정된 위치로 이동한다.
FO_RENAME ($4)	pszSrcFile 에 지정된 파일의 이름을 변경한다.
PO_DELETE ($13)	pszSrcFile 에 지정된 프린터를 삭제한다.
PO_PORTCHANGE ($20)	프린터 포트를 변경한다. pszSrcFile 과 pszDestFile 파라미터는 문자열의 목록을 가지는데, 각각의 문자열은 포트와 프린터 이름이 연결되어

	있다. pszSrcFile 파라미터는 현재 프린터 포트를, pszDestFile 파라미터에는 새로운 프린터 포트를 지정한다.
PO_RENAME ($14)	pszSrcFile 에 지정된 프린터의 이름을 변경한다.
PO_REN_PORT ($34)	PO_RENAME 과 PO_PORTCHANGE 의 결합

wFlags 파라미터는 작업에 대한 추가적인 내용을 지정한다. 이 파라미터에는 다음 값의 조합으로 지정할 수 있다.

값	의 미
FOF_ALLOWUNDO ($40)	Undo 정보를 저장한다.
FOF_MULTIDESTFILES ($1)	SHFileOperation 함수에서 여러 파일을 선택할 수 있다.
FOF_NOCONFIRMATION ($10)	확인을 하지 않는다.
FOF_NOCONFIRMMKDIR ($200)	새로운 디렉토리를 생성해야 할 때 확인을 하지 않는다.
FOF_RENAMEONCOLLISION ($8)	이미 존재하는 파일에 작업을 할 경우 새로운 이름을 이용한다.
FOF_SILENT ($4)	파일 작업이 진행되는 대화 상자를 보여주지 않는다.
FOF_SIMPLEPROGRESS ($100)	작업 상황을 간단히 보여주지만, 파일 이름을 표시하지 않는다.

pszSrcFile 파라미터는 소스 폴더를 지정하며, dwSrcAttribs 파라미터는 소스 폴더의 속성을 지정한다. 마찬가지로 pszDestFile 파라미터는 목적 폴더의 이름을 나타내며, dwDestAttribs 파라미터는 목적 폴더의 속성을 나타낸다.
다른 메소드 들과는 달리 이 메소드는 OLE 결과 코드를 반환하지 않고, 작업을 허용할 때에는 IDYES, 이 파일에 대한 작업만 허용하지 않을 경우에는 IDNO, 전체 작업을 취소할 때에는 IDCANCEL 을 반환한다.

컨텍스트 메뉴 핸들러 (Context Menu Handlers)

컨텍스트 메뉴 핸들러는 쉘에서 파일 객체와 연관되어 있는 로컬 메뉴에 아이템을 추가할 수 있도록 해주는 역할을 한다. 컨텍스트 메뉴 쉘 확장을 지원하기 위해서는 IShellExtInit 인터페이스와 IContextMenu 인터페이스를 지원해야 한다.
Copy hook 핸들러와 마찬가지로 컨텍스트 메뉴 핸들러를 작성하기 위해서도 TComObject 클래스에서 상속받아서 CoClass 를 작성해야 한다.
IShellExtInit 인터페이스는 쉘 확장을 초기화하는데 사용된다. 이 인터페이스에는 다음과 같은 Initialize 메소드 하나로 정의되어 있다.

```
function Initialize(pidlFolder: PItemIDList; lpdobj: IDataObject; hKeyProgID: HKEY): HResult; stdcall;
```

Initialize 메소드는 컨텍스트 메뉴 핸들러를 초기화하는 역할을 한다. pidlFolder 파라미터는 컨텍스트 메뉴를 디스플레이할 아이템을 포함하는 폴더에 대한 PItemIDList 구조체의 포인터이다. PItemIDList 에 대해서는 앞에서 자세히 설명한 바 있으므로 자세한 설명은 생략한다. lpdobj 파라미터는 실행할 객체를 가져오는데 사용할 IDataObject 인터페이스 객체를 지정한다. hkeyProgID 파라미터는 파일 객체나 폴더 형에 대한 레지스트리 키를 포함한다.
Initialize 메소드를 구현하기 위해서는 lpdobj 파라미터에서 IDataObject 인터페이스의 GetData 메소드를 이용하여 데이터를 가져온 뒤에 DragQueryFile 메소드를 이용하여 파일의 수와 파일 이름을 가져와야 한다. 자세한 내용은 예제를 작성하면서 설명하도록 하겠다.
IContextMenu 인터페이스는 쉘의 파일과 연관되어 있는 팝업 메뉴를 관리하는 기능을 하는 것으로, 다음과 같이 정의되어 있다.

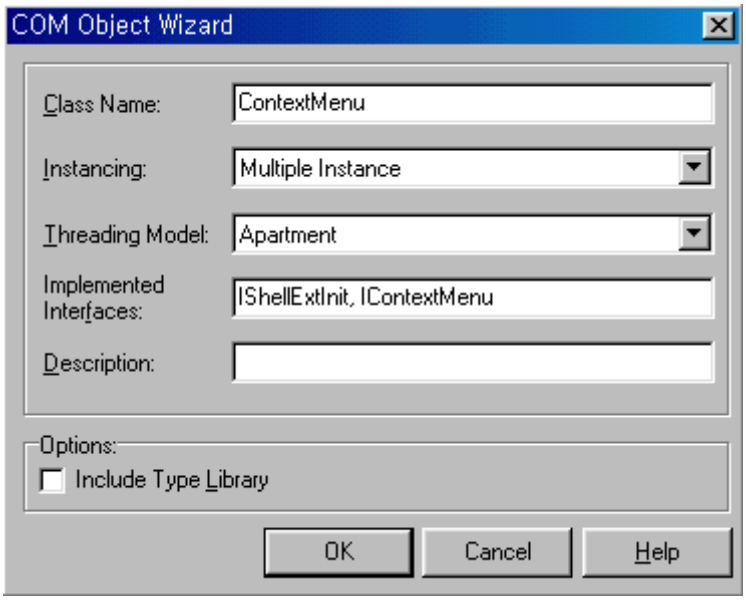
```
IContextMenu = interface(IUnknown)
  [SID_IContextMenu]
  function QueryContextMenu(Menu: HMENU;
    indexMenu, idCmdFirst, idCmdLast, uFlags: UINT): HResult; stdcall;
  function InvokeCommand(var lpici: TCMInvokeCommandInfo): HResult; stdcall;
  function GetCommandString(idCmd, uType: UINT; pwReserved: PUINT;
    pszName: LPSTR; cchMax: UINT): HResult; stdcall;
end;
```

일단 컨텍스트 핸들러가 IShellExtInit 인터페이스에 의해 초기화되면, 그 다음에는 IContextMenu 인터페이스의 QueryContextMenu 메소드가 호출된다. 이 메소드의 파라미터에는 메뉴의 핸들과 삽입될 메뉴 아이템이 첫번째 메뉴 아이템에서 몇 번째에 위치할 것인지를 지정할 인덱스, 그리고 메뉴 아이템 ID의 최소값과 최대값을 지정한다. 마지막으로 메뉴의 속성을 지정할 플래그를 파라미터로 넘겨주면 된다.
그 다음에는 GetCommandString 메소드가 쉘에 의해 호출되는데, 이 메소드는 특정 메뉴 아이템에 대한 언어 독립적인 커맨드 문자열(language-independent command string)이나 도움 문자열(help string)을 반환하는 역할을 한다. 이 메소드의 파라미터로는 메뉴 아이템의 옵셋과 받아올 정보의 종류를 나타내는 플래그, 예약된 파라미터와 문자열 버퍼 그리고 버퍼의 크기를 지정하게 된다.
컨텍스트 메뉴에서 사용자가 새로운 아이템을 클릭하면 쉘은 InvokeCommand 메소드를 호출한다. 이 메소드는 TCMInvokeCommandInfo 구조체를 파라미터로 이용하는데, 이 구조체는 ShlObj.pas 유닛에 다음과 같이 정의되어 있다.

```
PCMInvokeCommandInfo = ^TCMInvokeCommandInfo;
_CMINVOKECOMMANDINFO = record
  cbSize: DWORD;
  fMask: DWORD;
  hwnd: HWND;
  lpVerb: LPCSTR;
  lpParameters: LPCSTR;
  lpDirectory: LPCSTR;
  nShow: Integer;
  dwHotKey: DWORD;
  hIcon: THandle;
end;
TCMInvokeCommandInfo = _CMINVOKECOMMANDINFO;
```

여기에서 lpVerb 필드의 LoWord 에 선택된 메뉴 아이템의 인덱스가 저장되어 있다.
컨텍스트 메뉴 핸들러는 반드시 시스템 레지스트리의 HKEY_CLASSES_ROOT₩<File Type>₩shellex₩ContextMenuHandlers 에 등록되어야 한다. TComObjectFactory 클래스에서 상속받아 레지스트리에 등록하는 과정을 추가한 새로운 클래스 팩토리를 이용하여 이 문제를 해결할 것이다.
그러면, 꽤 쓸모 있는 예제를 만들어 보자.
델피언이라면 폴더를 열고 파일을 열어서 팝업 메뉴를 사용할 때, .pas 확장자를 가진 델파이 유닛의 소스 코드를 보고자 하면 지금까지는 .pas 파일을 더블 클릭하여 덩치 큰 델파이 IDE 를 띄워서 보거나, 아니면 메모장과 같은 에디터를 일단 띄운 뒤에 '열기' 메뉴를 이용하여 .pas 파일을 읽도록 해야 소스 코드를 볼 수 있었을 것이다. 아마도, 필자를 포함해서 많은 독자들이 .pas 유닛 파일에 커서를 두고 오른쪽 버튼을 클릭한 뒤에, 여기서 파스칼 소스 코드를 표시하고 편집할 수 있는 에디터를 바로 띄워서 사용할 수 있으면 좋겠다는 생각을 모두들 가졌을 것이라고 생각한다. 실제로 델파이 2, 3 의 (지금쯤은 델파이 4 버전도 나왔을 지 모르겠다.) Merlin 이라는 쉘 확장 프로그램을 설치하면 델파이 유닛 전용 에디터를 팝업 메뉴에서 직접 띄워서 이를 편집할 수 있도록 제공하였다.
이번에 제작할 컨텍스트 메뉴 핸들러는 .pas 확장자를 가진 델파이 유닛 파일에서 오른쪽 버튼을 클릭하면 '내용 보기'라는 메뉴 아이템이 추가된 팝업 메뉴가 나타나며, '내용 보기'를 선택하면 메모장에 해당 델파이 유닛 파일을 열어서 보여주게 되는 역할을 하는 것이다. 이와 같이 쉘 확장 기법의 활용에서도 파일 뷰어('간략히 보기')를 제작하는 것과 함께 컨텍스트 메뉴 핸들러를 만드는 것은 가장 그 유용성이 높은 것 중에 하나이다.
컨텍스트 메뉴 핸들러를 제작하기 위해서 먼저 File|New 메뉴를 선택하고 ActiveX 탭의 ActiveX Library 아이콘을 더블 클릭하여 액티브 X DLL 을 생성하도록 한다. 앞에서도 간

단히 설명했듯이 앞에서 구현한 IShellLink 인터페이스와는 달리 IContextMenu 와 IShellExtInit 인터페이스에 대한 내용은 윈도우가 구현하고 있는 객체를 활용하는 것이 아니기 때문에, 이들 인터페이스를 구현하는 CoClass 를 직접 구현해야 한다. 참고로 여기서 설명하는 용어들을 이해하기 어렵다면 제 4 부의 액티브 X 에 대한 내용들을 보다 정확하게 이해하고 다시 읽어보기를 권장한다. 그러므로, TComObject 에서 상속한 객체에 IContextMenu 와 IShellExtInit 인터페이스를 구현하도록 선언할 것이다. 이를 위해서 델파이 4 에서 새롭게 제공되는 ComObject 위저드를 이용하도록 하자. File|New 메뉴를 선택하고 ActiveX 탭에서 ComObject 아이콘을 더블 클릭한다. 그리고, 나타나는 대화상자의 내용을 다음과 같이 채운다.



OK 버튼을 클릭하면 아마도 다음과 같은 뼈대 코드가 만들어질 것이다.

```
unit U_Exam2;

interface
uses
  Windows, ActiveX, ComObj;

type
  TContextMenu = class(TComObject, IShellExtInit, IContextMenu )
  protected
    // IShellExtInit methods
    // IContextMenu methods
```

```
  end;

const
  Class_ContextMenu: TGUID = '{B722EC20-3D21-11D2-A345-0000E8364868}';

implementation
uses ComServ

initialization
  TComObjectFactory.Create(ComServer, TContextMenu, Class_ContextMenu,
    'ContextMenu', '', ciMultiInstance, tmApartment);
end.
```

여기에서 먼저 IShellExtInit, IContextMenu 인터페이스의 선언부가 있는 ShlObj.pas 유닛을 interface 섹션의 uses 절에 추가한다. 그리고 이들 인터페이스의 메소드를 다음과 같이 TContextMenu 클래스의 메소드로 구현하기 위해 선언하도록 한다.

```
uses
  Windows, ActiveX, ComObj, ShlObj;

type
  TContextMenu = class(TComObject, IShellExtInit, IContextMenu )
  private
    FFileName: array[0..MAX_PATH] of Char;
    FMenuIndex: UINT;
  protected
    // IShellExtInit method
    function Initialize(pidlFolder: PItemIDList; lpdobj: IDataObject;
      hKeyProgID: HKEY): HResult; reintroduce; stdcall;
    // IContextMenu methods
    function QueryContextMenu(Menu: HMENU; indexMenu, idCmdFirst, idCmdLast,
      uFlags: UINT): HResult; stdcall;
    function InvokeCommand(var lpici: TCMInvokeCommandInfo): HResult; stdcall;
    function GetCommandString(idCmd, uType: UINT; pwReserved: PUINT;
      pszName: LPSTR; cchMax: UINT): HResult; stdcall;
  end;
```

먼저, 구현에 필요한 유닛인 SysUtils.pas, ShellAPI.pas, Registry.pas 유닛을 uses 절에 추가한다. 그리고, IShellExtInit 인터페이스의 Initialize 메소드를 다음과 같이 구현한다.

```
function TContextMenu.Initialize(pidlFolder: PItemIDList; lpdobj: IDataObject;
  hKeyProgID: HKEY): HResult;
var
  StgMedium: TStgMedium;
  FormatEtc: TFormatEtc;
begin
  try
    if lpdobj = nil then
    begin
      Result := E_FAIL;
      Exit;
    end;
    with FormatEtc do
    begin
      cfFormat := CF_HDROP;
      ptd := nil;
      dwAspect := DVASPECT_CONTENT;
      lindex := -1;
      tymed := TYMED_HGLOBAL;
    end;
    Result := lpdobj.GetData(FormatEtc, StgMedium);
    if Failed(Result) then Exit;
    try
      if DragQueryFile(StgMedium.hGlobal, $FFFFFFFF, nil, 0) = 1 then
      begin
        DragQueryFile(StgMedium.hGlobal, 0, FFileName, SizeOf(FFileName));
        Result := NOERROR;
      end
      else
        Result := E_FAIL;
    finally
      ReleaseStgMedium(StgMedium);
```

```
    end;
  except
    Result := E_UNEXPECTED;
  end;
end;
```

이 구현 부분에서 핵심이 되는 것은 IDataObject 인터페이스인 lpdobj 파라미터이다. 이 파라미터의 GetData 메소드를 이용하여 파일의 데이터를 가져오는 부분이다. 이를 제대로 이해하기 위해서는 TStgMedium 과 TFormatEtc 구조체의 내용을 이해해야 한다.

TStgMedium 구조체는 IDataObject, IAdviseSink 인터페이스의 데이터 전송 작업에 사용되는 전역 메모리 핸들로 사용된다. tymed 필드는 저장되는 매체의 종류를 나타내는 것으로 hBitmap(비트맵), hGlobal(전역 메모리 핸들) 등의 값을 가질 수 있다. 이 필드는 공용체(union)과 같은 형태로 사용될 수 있다. 그러므로 StgMedium.hGlobal 은 전역 메모리에 대한 핸들을 나타낸다. 더 자세한 내용은 Win32 SDK 의 도움말을 참고하기 바란다.

TFormatEtc 구조체는 일반화된 클립보드 포맷이다. 그러므로 목적 디바이스와 데이터의 뷰, 저장 매체 등에 따라 다른 값들을 지정할 수 있다. OLE 에서는 이 구조체를 사용하여 클립보드의 정보를 이용한다. cfFormat 멤버는 특정 클립보드 포맷을 나타내는 것으로 OLE 에 의해 인식되는 형식은 CF_TEXT 와 같은 표준 포맷과 특정 어플리케이션에 의해서만 지원되는 포맷, 그리고 객체의 연결과 임베딩이 가능한 OLE 포맷 등이 있다. 여기서 사용한 CF_HDROP 포맷은 TDropFiles 구조체를 포함한 전역 메모리 객체로, 이 객체는 클립보드에서 드래그-드롭 작업을 할 때 같이 복사된다. 어플리케이션은 데이터 객체에 대한 정보를 DragQueryFile 이나 DragQueryPoint 함수에 객체의 핸들을 파라미터로 호출하여 얻을 수 있다.

ptd 멤버는 목적 디바이스에 대한 정보를 포함하는 구조체를 가리키는 멤버로 이 값이 nil 이면 지정된 데이터 포맷이 목적 디바이스의 종류에 상관없이 적용된다는 의미이다. dwAspect 멤버는 특정 클립보드 포맷의 여러 뷰 중에서 어떤 것을 선택할 것인지를 결정한다. DVASPECT_CONTENT 는 객체를 컨테이너 내부에 임베딩된 형태로 표시한다는 의미이다. linindex 멤버는 데이터가 페이지 경계에 걸렸을 때 어떤 aspect 를 보여줄 것인지를 결정하는 것으로 보통 -1 을 지정하는데, 이것은 데이터 모두를 보여주라는 의미이다. 마지막으로 tymed 멤버는 TStgMedium 의 내용과 같다. 더 자세한 내용은 Win32 SDK 의 도움말을 참고하기 바란다.

어쨌든 이 코드의 의미는 드래그-드롭이 가능한 파일의 내용을 초기화하고, IDataObject 인터페이스의 GetData 메소드를 이용하여 StgMedium, FormatEtc 구조체에 의해 지정된 형태로 데이터를 가져오는 것이다. 그리고 나서, DragQueryFile 함수에서 2 번째 파라미터를 $FFFFFFFF 로 지정하면 드롭된 파일의 수를 알아낼 수 있다. 여기서는 파일이 하나일 때만 열 수 있으므로 그 값이 1 인 경우에만 다음으로 진행한다. 그 다음에는

DragQueryFile 을 다시 호출하여 FFileName 변수에 드롭된 파일의 버퍼를 지정한다.
이렇게 Initialize 메소드를 구현함으로써 FFileName 변수를 이용해 파일에 접근할 수 있도록 초기화를 완료하였다.
이제 IContextMenu 인터페이스의 QueryContextMenu 메소드를 구현하도록 하자. 이 메소드는 메뉴 인덱스를 지정하고, 메뉴 아이템을 삽입하면 되므로 다음과 같이 간단히 구현이 가능하다.

```
function TContextMenu.QueryContextMenu(Menu: HMENU; indexMenu, idCmdFirst,
  idCmdLast, uFlags: UINT): HResult;
begin
  FMenuIndex := indexMenu;
  InsertMenu(Menu, FMenuIndex, MF_STRING or MF_BYPOSITION, idCmdFirst,
    '내용 보기');
  Result := FMenuIndex + 1;
end;
```

이 메소드에 의해 팝업 메뉴에 '내용 보기'라는 메뉴 아이템이 추가된다.
그러면, 팝업 메뉴에서 이 메뉴 아이템을 선택했을 때 호출되는 InvokeCommand 메소드를 다음과 같이 구현한다.

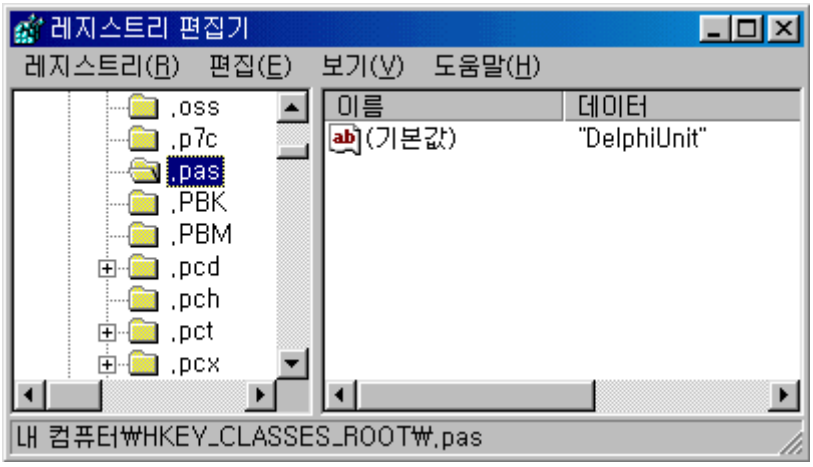
```
function TContextMenu.InvokeCommand(var lpici: TCMInvokeCommandInfo): HResult;
begin
  Result := S_OK;
  try
    if HiWord(Integer(lpici.lpVerb)) <> 0 then
    begin
      Result := E_FAIL;
      Exit;
    end;
    if LoWord(lpici.lpVerb) = FMenuIndex then
      WinExec(PChar('Notepad.exe ' + FFileName), SW_SHOW)
    else
      Result := E_INVALIDARG;
  except
    MessageBox(lpici.hwnd, '파일을 볼 수 없습니다 !', 'Error',
      MB_OK or MB_ICONERROR);
```

```
      Result := E_FAIL;
    end;
  end;
```

먼저 lpici 구조체의 lpVerb 멤버의 HiWord 를 검사하여 이 값이 0 인 경우에 제대로 선택된 경우이므로 계속 진행한다. 그리고, 이 멤버의 LoWord 값은 선택된 메뉴의 인덱스를 담고 있으므로 이 값이 FMenuIndex 값과 일치한다면 추가한 메뉴 아이템을 선택한 경우이므로 WinExec 함수를 이용하여 메모장에 해당 델파이 유닛을 열도록 실행한다.
마지막으로 GetCommandString 메소드는 다음과 같이 구현하여 도움말 문자열을 제공하도록 한다.

```
function TContextMenu.GetCommandString(idCmd, uType: UINT; pwReserved: PUINT;
  pszName: LPSTR; cchMax: UINT): HRESULT;
begin
  Result := S_OK;
  try
    if (idCmd = FMenuIndex) and ((uType and GCS_HELPTEXT) <> 0) then
      StrLCopy(pszName, '파스칼 소스 코드를 메모장에 보여줍니다.', cchMax)
    else
      Result := E_INVALIDARG;
  except
    Result := E_UNEXPECTED;
  end;
end;
```

이것으로 IShellExtInit 인터페이스와 IContextMenu 인터페이스는 모두 구현되었다. 이제부터는 .pas 파일에 대한 쉘 확장 부분이 있음을 레지스트리에 등록해야한다. 이런 등록 작업을 하지 않으면 해당 확장자에 대한 정보가 없기 때문에 핸들러가 작동하지 않는다.
우리가 작업할 확장자는 .pas 이므로 .pas 확장자가 과연 어떤 시스템 객체를 가리키고 있는지를 먼저 알아야 한다. 윈도우 디렉토리의 RegEdit.exe 파일을 실행하면 시스템 레지스트릴 직접 표시하고, 편집할 수 있는데 HKEY_CLASSES_ROOT 키를 열면 먼저 파일의 확장자들이 나열될 것이다. 여기서 우리가 쉘 확장을 시키려고 하는 .pas 확장자에 대한 키를 선택하면 오른쪽 pane 에 다음과 같이 해당 확장자에 대한 값을 보여줄 것이다. 여기서는 'DelphiUnit'이 바로 .pas 파일에 대한 객체이다.



참고로 여기에 등록되어 있지 않은 확장자에 대한 컨텍스트 메뉴 핸들러를 제작하는 경우라면 확장자에 대한 키를 새로 등록하고, 등록한 값에 대해서 다음에 설명하는 과정을 같은 방법으로 사용해야 한다.

확장자들의 키들의 아래에 계속해서 나타나는 객체들의 키 중에서 DelphiUnit 키를 찾아보자, 이 키에는 서브 키로 DefaultIcon 과 Shell 이 존재할 것이다. 여기에 쉘 확장 부분이 있다면 shellex 서브 키를 추가하면 되고, 컨텍스트 메뉴 핸들러가 존재하면 shellex 서브 키에 ContextMenuHandlers 서브 키를 추가하고 그 값으로 컨텍스트 메뉴 핸들러를 구현한 CoClass 의 CLSID 를 등록하면 된다.

이런 모든 과정은 직접 손으로 RegEdit.exe 프로그램을 이용해서 직접 입력해도 좋지만, 프로그램으로 처리하는 것이 더 좋다는 것은 당연한다.

이를 등록하기 위해서 가장 좋은 방법은 CoClass 가 처음 등록될 때, 레지스트리를 변경하는 과정에서 이들 내용을 같이 변경하도록 하는 것이다. 가장 세련된 형태로 이를 구현하는 것은 TComObjectFactory 클래스를 상속한 새로운 클래스 팩토리의 UpdateRegistry 메소드를 오버라이드하여 이를 구현하는 방법이다. 이때 컨텍스트 메뉴 핸들러는 ProgID 를 가지지 않으므로 GetProgID 메소드를 같이 오버라이드하여 ProgID 를 널 문자열로 지정하도록 하자.

그리고, 또 하나 고려해야 하는 것은 윈도우 NT 의 경우로 모든 쉘 확장 부분에 대해 HKEY_LOCAL_MACHINE₩SOFTWARE₩Microsoft₩Windows₩CurrentVersion₩Shell Extensions₩Approved 키에 그 내용을 기록해야 한다. 이를 위해 메소드를 하나 더 추가하도록 한다.

그러면 TContextMenuFactory 클래스를 다음과 같이 선언하고, initializatoin 섹션의 클래스 팩토리를 TContextMenuFactory 로 교체한다.

```
TContextMenuFactory = class(TComObjectFactory)
protected
```

```
    function GetProgID: string; override;
    procedure ApproveShellExtension(Register: Boolean; const CLSID: string);
      virtual;
  public
    procedure UpdateRegistry(Register: Boolean); override;
  end;

... (중략)

initialization
  TContextMenuFactory.Create(ComServer, TContextMenu, Class_ContextMenu,
    'ContextMenu', '', ciMultiInstance, tmApartment);
end.
```

GetProgID 메소드는 구현하기 쉽다. 단순히 ProgID 로 널 문자열을 넘겨주도록 하면 된다.

```
function TContextMenuFactory.GetProgID: string;
begin
  Result := '';
end;
```

UpdateRegistry 메소드는 앞에서 설명한 바대로 DelphiUnit 키에 대해 새로운 쉘 확장에 대한 서브 키를 추가하고, 그 값으로 컨텍스트 메뉴 핸들러 객체의 CLSID 를 추가한다. 여기서 중간에 ApproveShellExtension 메소드를 호출하여 윈도우 NT 의 경우에 추가적인 레지스트리 등록 작업을 수행하도록 하면 된다.

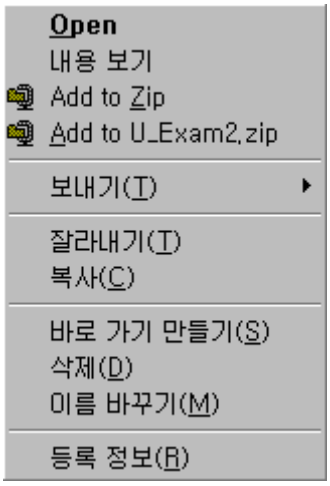
```
procedure TContextMenuFactory.UpdateRegistry(Register: Boolean);
var
  CLSID: string;
begin
  CLSID := GUIDToString(ClassID);
  inherited UpdateRegistry(Register);
  ApproveShellExtension(Register, CLSID);
  if Register then
  begin
    CreateRegKey('DelphiUnit₩shellex₩ContextMenuHandlers₩' +
```

```
        ClassName, '', CLSID);
    end
    else begin
      DeleteRegKey('DelphiUnit₩shellex₩ContextMenuHandlers₩' +
        ClassName);
    end;
end;
```

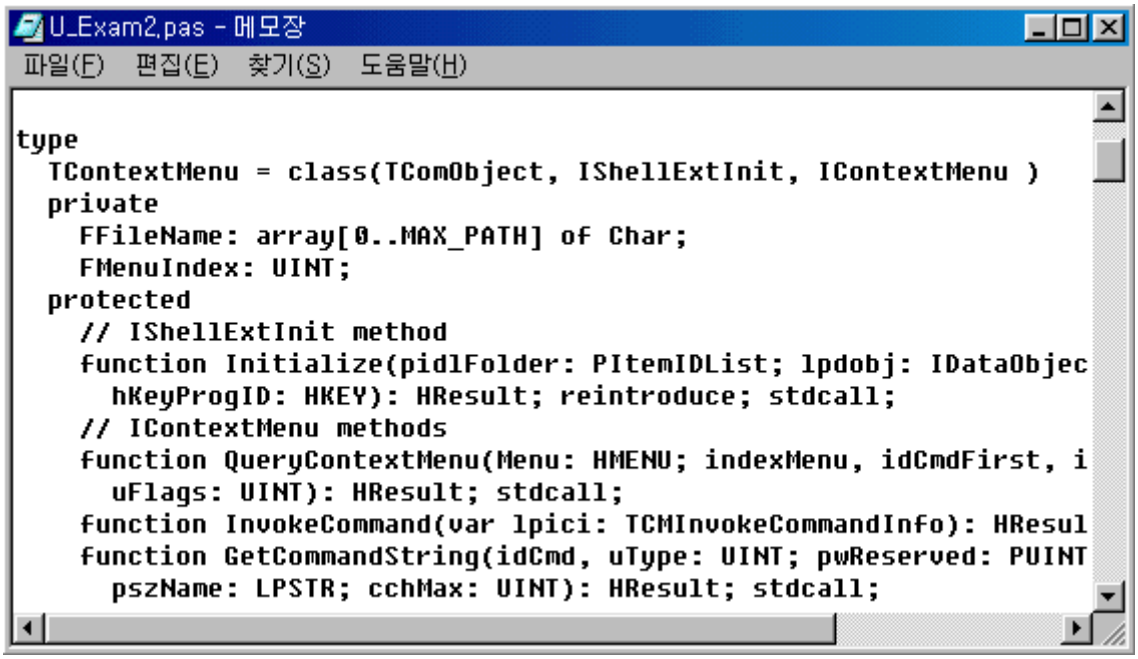
마지막으로 ApproveShellExtension 메소드는 다음과 같이 구현하여 해당 키에 값을 등록하도록 하면 된다.

```
procedure TContextMenuFactory.ApproveShellExtension(Register: Boolean;
  const CLSID: string);
const
  SApproveKey = 'SOFTWARE₩Microsoft₩Windows₩CurrentVersion₩Shell Extensions₩Approved';
begin
  with TRegistry.Create do
    try
      RootKey := HKEY_LOCAL_MACHINE;
      if not OpenKey(SApproveKey, True) then Exit;
      if Register then WriteString(CLSID, Description)
      else DeleteValue(CLSID);
    finally
      Free;
    end;
end;
```

이것으로 컨텍스트 메뉴 핸들러를 구현하였다. 컴파일하고 Run|Register ActiveX Server 메뉴를 선택하여 이를 등록하도록 하자. 그리고, 탐색기를 실행하고 .pas 확장자를 가진 파일에 커서를 위치시킨후 오른쪽 버튼을 클릭하면 다음과 같이 추가된 메뉴 아이템을 볼 수 있을 것이다.



'내용 보기' 메뉴 아이템을 선택하면 다음과 같이 메모장에 해당 파스칼 소스 파일이 열린다.



정 리 (Summary)

이번 장에서는 델파이를 이용해 각종 쉘 확장 기법을 이용하는 방법에 대해서 몇 가지 알아보았다. 여기에 다룬 내용 이외에도 쉘 확장 기법에는 시스템 아이콘을 이용하는 방법도 있고, 파일 뷰어를 구현하고 OLE 객체의 진정한 드래그-드롭을 구현하는 등의 여러가지 기법을 구현할 수 있다. 이들을 자유자재로 구현하는 좋은 쉘 유틸리티를 개발하기 위해서는 COM 과 OLE 인터페이스에 대한 지식이 필수적이라는 것은 두말할 나위도 없다.

아마도, 필자가 나름대로 쉽게 풀어썼음에도 불구하고 내용이 난해하다고 느낀 독자가 많을

것이다. 그런 독자들은 제 4 부의 내용을 다시 한번 읽어보고 COM 에 대한 내용을 보다 확실하게 익힌 후에 다시한번 도전해 보기 바란다. 기본적인 원리만 이해한다면 쉘 확장 기법을 이용한 환상적인 유틸리티를 작성하는 것은 이미 누워서 떡먹기와도 다름 없다.

참고로, 잘된 쉘 확장 유틸리티를 작성하기 위해서는 Win32 SDK 를 비롯하여 마이크로소프트에서 제공하는 MSDN 과 같은 문서자료를 많이 참고하는 것이 필수적이다.

아무쪼록, 우리나라에서도 외국에서 나오는 윈도우 95/98 전용의 쉘 확장 유틸리티를 능가하는 한국형 쉘 확장 유틸리티가 나오기를 기대한다. 도스의 MDir 은 그 기능과 편리성에서 세계적인 수준이었음에도 불구하고 윈도우 시장에는 그런 작품이 국내에서 나오지 못한 것이 매우 안타깝다고 생각된다.

필자의 개인적인 생각으로는 비주얼 C++ 보다 델파이가 이런 쉘 확장 유틸리티를 작성하는데 훨씬 유리하다고 생각하고 있다. 이번 장의 내용이 한국형 쉘 확장 유틸리티 탄생의 초석이 되기를 간절히 바라면서 이번 장을 마무리하고자 한다.