델파이 4 에서의 DLL 프로그래밍

(DLL Programming in Delphi 4)

윈도우에서 가장 표준적으로 다루게 되는 파일 형식 중에서 동적으로 사용하는 표준적인 실행 파일 형식이 바로 DLL 파일이다. 이번 장에서는 델파이 4 를 이용하여 DLL 파일을 이용하고, 제작하는 방법에 대해서 알아보도록 한다.

DLL 에 대한 기초적인 이해

동적 링킹(dynamic linking)이란 정적 링킹(static linking)과 상대되는 개념으로 실행 가능한 모듈 런타임에서 필요한 정보만을 포함할 수 있도록 하는 것이다. 이에 비해서 정적 링킹이란 라이브러리 함수의 실행 코드가 각각의 어플리케이션에 담겨져 있어야 한다.
DLL 은 프로세스가 시작되거나 프로세스의 쓰레드가 LoadLibrary 함수를 호출할 때 프로세스에 연결된다. 일단 DLL 이 프로세스에 연결되면 운영체제는 DLL 모듈을 프로세스의 주소 공간에 매핑을 하게 되고, 이 과정을 통해 프로세스가 DLL 의 실행 코드에 접근할 수 있다. 반대로 프로세스가 종료되거나 FreeLibrary 함수가 호출되면 DLL 모듈은 프로세스의 주소 공간의 매핑에서 벗어난다.
다른 함수들과 마찬가지로, DLL 함수 역시 쓰레드의 공간에서 실행된다. 그렇기 때문에 다음과 같은 조건을 항상 염두에 두어야 한다.

1. DLL 을 호출한 프로세스의 쓰레드는 DLL 함수에 의해 열린 핸들을 사용할 수 있다. 마찬가지로 호출하는 프로세스의 쓰레드가 연 핸들은 DLL 함수가 사용할 수 있다.
2. DLL 은 호출 쓰레드의 스택과 호출 프로세스의 주소 공간을 사용한다.
3. DLL 에 의해 할당된 메모리는 호출 프로세스의 주소 공간에 있다.
4. DLL 을 호출한 프로세스의 쓰레드들은 DLL 에서 전역으로 선언된 변수 값을 읽고 쓸 수 있다.

- 동적 링킹의 장단점

동적 링킹은 정적 링킹에 비해 몇 가지 장점을 가지고 있다. 이를 나열하면 다음과 같다.

1. 많은 수의 프로세스들이 하나의 DLL 을 동시에 사용할 수 있다. 이렇게 하면 메모리 공간을 절약할 수 있다.
2. DLL 에 있는 함수가 변하면 이를 사용하는 어플리케이션은 함수의 형태와 리턴 값이

바뀌지 않는 한에는 재컴파일이나 링크가 필요 없이 기능이 업그레이드될 수 있다.

3. DLL 로 쪼개 놓으면 나중에 업그레이드가 용이하다. 즉, 기능이 향상된 부분의 DLL 만 통신 등으로 보내면 해결이 될 수 있는 경우가 많다.
4. 서로 다른 프로그래밍 언어로 만들어진 프로그램 사이에서 함수의 호출 규칙(calling convention)만 동일하다면 함수의 공유와 통신이 가능하다.

DLL 을 사용할 때 단점 역시 존재한다. 쪼개져 있기 때문에 좋은 점도 있지만, 이들 중 하나만 없어도, 또는 하나만 비정상적으로 동작해도 전체 어플리케이션이 동작하지 않는 불상사를 맞을 수 있다.

- DLL 진입점(Entry-Point) 함수

DllEntryPoint 함수는 운영체제가 호출하는 함수로, 반드시 있을 필요는 없다. DllEntruPoint 는 전형적인 함수가 아니라 링커 커맨드 라인에 의해 지정되는 참조값(reference)이다. 그러므로, DLL 에 함상 진입점 함수가 있을 필요는 없다. 하지만, 쓰레드와 프로세스를 초기화하는 작업 등을 할 때 유용하게 사용할 수 있다.
일단 진입점 함수가 지정되면 운영체제는 프로세스에 DLL 이 연결되거나 연결이 끊길 때, 또는 쓰레드가 DLL 에 연결되거나 연결이 끊길 때 이 함수를 호출한다.
이렇게 호출되면 DLL 은 프로세스의 주소 공간에 메모리를 할당하거나 프로세스가 접근할 수 있는 핸들을 열 수가 있다. 또한, 새로운 쓰레드와 연결될 때에는 그 쓰레드에 대한 지역 저장소(thread local storage, TLS)를 할당할 수 있다.

- 런타임 동적 링킹 (Run-Time Dynamic Linking)

런타임 동적 링킹은 프로세스가 DLL 의 이름을 지정해서 LoadLibrary 함수를 호출하고, DLL 의 함수에 대한 시작 주소를 얻어오는 GetProcAddress 함수를 호출함으로써 이루어진다. LoadLibrary 를 호출할 때 DLL 모듈이 이미 호출 프로세스의 주소 공간에 매핑되어 있는 경우에는 DLL 에 대한 핸들을 반환하고, 모듈의 참조 계수(reference count)를 하나 증가시킨다.
시스템이 DLL 을 찾지 못하거나 진입점 함수가 False 를 반환하면 LoadLibrary 는 NULL 값을 리턴하며, 성공적으로 수행된 경우에는 DLL 모듈의 핸들을 돌려 준다. 프로세스는 이 핸들을 이용해서 GetProcAddress, FreeLibrary 등의 함수를 호출할 수 있게 된다.
GetModuleHandle 함수도 GetProcAddress 나 FreeLibrary 함수가 사용할 수 있는 핸들을 반환하지만, DLL 모듈이 이미 프로세스의 주소 공간에 매핑이 된 후에야 동작하며, 모듈의 참조 계수를 증가시키지 않는다.
LoadLibrary 나 GetModuleHandle 함수에 의해 DLL 의 모듈 핸들을 얻었으면 프로세스는

GetProcAddress 함수를 이용해서 함수의 시작점을 알아낼 수 있다.
더 이상 DLL 모듈이 필요하지 않으면 프로세스는 FreeLibrary 함수를 호출하여 모듈의 참조 계수를 하나 감소시키고, 만약 참조 계수가 0 이 되면 주소 공간의 매핑을 해제한다.

- DLL 데이터

윈도우는 다음의 DLL 데이터를 지원한다.

1. DLL 을 사용하는 각각의 프로세스 전용의 전역 또는 정적 변수
2. DLL 을 사용하는 모든 프로세스가 공유하는 전역 또는 정적 변수
3. DLL 을 사용하는 각각의 프로세스 전용의 동적 할당 메모리
4. 여러 프로세스에서 공유할 수 있는 동적 할당 메모리
5. 멀티 쓰레드 프로세스의 각각의 쓰레드 전용의 정적 저장소(static storage)

DLL 변수의 디폴트 범위는 1 의 경우로 각각의 프로세스에 대해 전역, 정적 변수로 설정되어 있다.
DLL 이 메모리 할당 함수(GetMem, New, GlobalAlloc, LocalAlloc, HeapAlloc, VirtualAlloc)를 써서 메모리를 할당할 경우, 메모리는 DLL 을 호출한 프로세스의 주소 공간에 할당되므로 그 프로세스의 쓰레드만 접근할 수 있다. 그러므로, DLL 에서 데이터를 공유하기 위해서는 파일 매핑을 사용해서 메모리를 할당해야 한다.

- Win16 과 Win32 DLL

Win32 환경에서의 근본적인 Win16 과의 차이점은 세그먼트로 나뉘어진 메모리 모델을 플랫 메모리 프로세스로 변경한다는 것이다. 16 비트 윈도우에서는 DLL 이 운영체제와 통합된 전역으로 접근 가능한 라이브러리로 취급되기 때문에, 여러 어플리케이션에서 공유할 수 있다.
16 비트 DLL 은 어플리케이션을 적재할 때 스택을 사용하기 때문에, 메모리의 64K 데이터 세그먼트 제한을 가질 수 밖에 없다. 이들 DLL 을 사용할 때에는 LoadLibrary 함수를 사용하는데, LoadLibrary 함수가 호출되면 윈도우가 관리하는 참조 계수가 하나 증가한다. 어플리케이션이 DLL 을 사용하고 나면, FreeLibrary 함수를 호출하게 되는데 이 함수에 의해 참조 계수가 하나 감소하며 이 값이 0 이되면 DLL 이 메모리에서 해제된다.
16 비트 DLL 은 자신의 데이터 세그먼트를 따로 가지고 있기 때문에, 전역 변수와 상수 들이 어플리케이션 사이에서 서로 통신을 할 수 있는 수단으로 사용이 가능하다. 어플리케이션은 DLL 에 메모리 블록을 글로벌 힙에 할당하고, 포인터를 다른 어플리케이션에서 사용하도록 넘기는 등의 작업을 할 수 있다.

Win32 모델에서는 어플리케이션이 자신의 메모리 공간에 분리되어 존재한다. 운영체제가 어플리케이션을 적재하게 되면, 프로세스(process)라고 하는 커널 프로세스 객체(kernel process object)를 생성한다. 하나의 프로세스는 최대 4GB 까지의 크기를 가질 수 있으며, 시스템 페이징 파일의 메모리 지역을 주소 관리를 위해 할당된다. 프로세스가 생성되면, 커널 파일 매핑 객체(kernel file-mapping object)가 생성되어 실행 파일을 프로세스의 주소 공간에 매핑하게 된다. 이로 인해 실행 파일은 하드 디스크의 특정 위치에 자리잡게 되므로 전체 이미지를 RAM 으로 적재할 필요가 없다.
실행 파일이 메모리에 매핑되면, 운영체제가 EXE 파일에 의해 요구되는 모든 DLL 의 이름들을 가져오기 위해 이미지의 오프셋으로 이동한 뒤에 요구되는 DLL 을 찾아서 EXE 파일과 같은 프로세스로 매핑한다. 커널 파일 매핑 객체는 각각의 DLL 에 대해서 생성된다. 어플리케이션의 실행이 종료되면 각각의 DLL 의 매핑이 해제되고 파일 매핑 객체가 파괴된다. 이런 프로세스는 EXE 파일의 매핑이 해제되고 커널 프로세스 객체가 파괴될 때까지 지속된다. Win32 환경에서 DLL 은 이와 같이 각각의 어플리케이션에 의해 독자적으로 사용된다. 다시 말해 여러 인스턴스가 생성되어 각각의 DLL 이 독자적인 주소 공간을 가지게 되는 것이다.

DLL 호출하기

DLL 에 정의되어 있는 루틴을 호출하기 전에, 이들을 반드시 import 해야 한다. 이렇게 DLL 의 함수를 import 하는 방법에는 2 가지가 있다. 첫번째는 외부 프로시저나 함수를 선언하는 것이고, 다른 하나는 윈도우 API 함수를 직접 호출하는 것이다.
어떤 방법을 사용하든 런타임이 될 때까지는 어플리케이션에 링크되지 않는다. 이는 DLL 이 프로그램을 컴파일할 때에는 필요없다는 것이며, 루틴을 import 할 때 컴파일 시에는 특별한 타당성 검사를 하지 않는다는 것을 의미한다.

- 정적 로딩 (Static loading)

DLL 함수를 import 하여 사용하는 가장 간단한 방법은 외부 지시어를 이용하여 선언해 사용한 것이다. 예를 들어, 다음과 같이 하면 된다.

```
procedure DoSomething; external 'MYLIB.DLL';
```

이런 선언부를 프로그램에 위치시키면, MYLIB.DLL 이 프로그램이 시작될 때 메모리에 적재된다. 프로그램이 실행되는 동안 DoSomething 을 호출하는 것은 이는 언제나 DLL 의 진입점(entry point)를 호출하는 것을 의미한다.
이런 선언부는 프로그램에 직접 위치시킬 수도 있고, 따로 유닛을 추가해서 선언한 뒤에 이

유닛을 use 절에 추가하여 사용할 수도 있다. 관리의 편리성이라는 측면에서 이렇게 따로 유닛을 선언해서 사용하는 것이 좋은데, 델파이의 Windows.pas 유닛도 이런 대표적인 DLL 의 API 함수를 선언한 유닛이다.

- 동적 로딩 (Dynamic loading)

DLL 에 직접 접근하여 사용하려면 윈도우의 API 함수를 호출해서 사용할 수 있다. 대표적인 함수가 LoadLibrary, FreeLibrary, GetProcAddress 등이다. 이들은 Windows.pas 유닛에 선언되어 있으므로 uses 절에 Windows.pas 유닛을 추가하고 다음과 같은 식으로 사용하면 된다.

```
uses Windows, ...;
type
  TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
  end;
  TGetTime = procedure(var Time: TTimeRec);
  THandle = Integer;

var
  Time: TTimeRec;
  Handle: THandle;
  GetTime: TGetTime;
  ...
begin
  Handle := LoadLibrary('DATETIME.DLL');
  if Handle <> 0 then
  begin
    @GetTime := GetProcAddress(Handle, 'GetTime');
    if @GetTime <> nil then
    begin
      GetTime(Time);
      with Time do
        WriteLn('The time is ', Hour, ':', Minute, ':', Second);
```

```
    end;
    FreeLibrary(Handle);
  end;
end;
```

이런 식으로 루틴을 import 하면 DLL 은 LoadLibrary 함수가 실행되기 전까지는 DLL 이 메모리에 적재되지 않는다. 그리고 LoadLibrary 에 의해 적재된 DLL 은 사용이 끝난 뒤 FreeLibrary 를 호출해야 메모리에서 해제된다. 그러므로, 꼭 필요할 때에만 DLL 을 메모리에 올려서 사용하게 되므로 메모리를 절약할 수 있는 장점이 있다.

DLL 제작하기

델파이를 이용해서 쉽게 DLL 을 만들 수 있다. 새로운 DLL 프로젝트는 하나의 DPR 파일로 이루어지며, 디폴트 폼이나 추가적인 유닛을 필요로 하지 않는다. DLL 을 만들려면 먼저 File|New 메뉴를 선택하여 객체 저장소를 띄우고 DLL 을 더블 클릭하면 껍데기 코드가 만들어진다.

```
library Exam1;

...

uses
  SysUtils,
  Classes;

begin
end.
```

DLL 의 구조는 기본적으로 다른 프로그램과 다를 바가 없다. 다만 DLL 프로젝트는 library 키워드로 시작된다. 다음 코드는 Min, Max 라는 2 개의 export 함수를 가지는 DLL 을 제작한 것이다.

```
library Exam1;

function Min(X, Y: Integer): Integer; stdcall;
begin
```

```
  if X < Y then Min := X else Min := Y;
end;

function Max(X, Y: Integer): Integer; stdcall;
begin
  if X > Y then Max := X else Max := Y;
end;

exports
  Min index 1,
  Max index 2;

begin

end.
```

다른 언어로 제작하는 어플리케이션에 DLL 을 사용할 수 있게 하려면 이와 같이 stdcall 호출 규칙을 사용하는 것이 좋다. 여기에 대해서는 다시 다루게 될 것이다. 이렇게 만든 DLL 을 사용할 때에는 앞에서도 설명했듯이 사용할 어플리케이션의 type 선언부나 새로운 유닛을 추가한 뒤에 Min, Max 를 다음과 같이 선언하고 간단히 이 함수를 호출하면 된다.

```
function Min(X, Y: Integer): Integer; stdcall; external 'Exam1.DLL';
function Max(X, Y: Integer): Integer; stdcall; external 'Exam1.DLL';

ShowMessage(IntToStr(Min(1, 2));        //결과는 '1'을 메시지 박스로 보여 준다.
```

이때 주의할 점은 Exma1.DLL 이 윈도우, 시스템이나 어플리케이션의 디렉토리와 같이 검색하는 디렉토리 안에 위치해야 한다는 점이다. 아니면 정확한 패스를 external 뒤에 적어주어야 한다. 또한, 앞에서 설명한 것과 같이 LoadLibrary 함수를 이용하여 동적으로 루틴을 호출할 수도 있다.
DLL 은 여러 개의 유닛으로 구성할 수도 있다. 이런 경우에는 라이브러리의 소스 파일은 uses 절에 실제로 루틴을 구현한 유닛을 적어주고, 여기에는 단순히 export 문장과 DLL 초기화 코드만 적어주면 된다. 다음 코드를 살펴 보자.

```
library Editors;
uses EdInit, EdInOut, EdFormat, EdPrint;
```

```
exports
  InitEditors index 1,
  DoneEditors index 2,
  InsertText index 3,
  DeleteSelection index 4,
  FormatSelection index 5,
  PrintSelection index 6,
   ...
  SetErrorHandler index 53;

begin

end.
```

이와 같이 DLL 프로젝트 파일은 명시적으로 루틴을 export 하는 부분 만을 선언하고, 나머지 구현 부분은 uses 절에 추가된 유닛에 위치한다.

- exports 절

루틴을 exports 하는 방법은 다음 형식을 가지고 선언하면 된다.

```
exports entry1, ..., entryn;
```

각각의 entry 는 exports 절 이전에 선언된 프로시저나 함수의 이름과 옵션으로 인덱스와 옵션 이름을 추가할 수 있다. 인덱스는 1~2,147,483,647 까지의 값을 가질 수 있다. 그렇지만 효율적인 프로그램이 되려면 낮은 수를 사용하는 것이 좋다. Entry 의 인덱스는 옵션이기 때문에, 이것이 없는 경우에는 DLL export 테이블에 자동으로 인덱스를 부여하게 된다. 또한, 이름을 지정하지 않은 경우에는 유닛에서 선언한 이름을 그대로 사용하게 된다. 그러므로, name 지시어는 다음과 같이 루틴의 원래 이름과 다른 이름으로 export 할 때 사용하게 된다.

```
exports
  DoSomethingABC index 1 name 'DoSomething';
```

- DLL 초기화 코드

델파이 프로젝트에서 DLL 의 진입점은 처음의 begin ... end. 사이에 위치한다. 로더 코드가 실행될 때 여기에 위치한 모든 문장이 가장 먼저 실행된다. 그러므로, 여기에는 초기화 루틴과 메모리 처리 루틴 등을 위치시키기 좋다. DLLEntryPoint 는 DLLMain 으로 불리기도 하는데, 옵션으로 제공되며 이 함수가 export 되는 경우에는 프로세스나 쓰레드가 DLL 을 적재하거나 해제할 때 호출된다. DLLEntryPoint 는 C++에서 다음과 같이 선언되어 있다.

```
BOOL WINAPI DLLEntryPoint(HINSTANCE hInst, DWORD dwReason, LPVOID lpvReserved);
```

델파이는 C++에서의 DLLMain 처럼 동작하는 DLLMain 루틴을 지원하는데, 이를 위해서 다음과 같은 추가적인 코딩이 필요하다.

```
library Exam1;

uses
  ShareMem, Windows, SysUtils, Classes;

procedure DLLMain(dwReason: DWORD);
begin
  case dwReason of
    dll_Process_Attach: ;
    dll_Process_Detach: ;
    dll_Thread_Attach: ;
    dll_Thread_Detach: ;
  end;
end;

begin
  DLLProc := @DLLMain;
  DLLMain(dll_Process_Attach);
end.
```

uses 절의 처음에 ShareMem 유닛을 추가한 것에 주의하라. 델파이에서 사용하는 긴 문자열을 제대로 지원하기 위해서는 이와 같이 DLL 의 첫번째 uses 절에 ShareMem.pas 유닛을 추가해야 하고, 배포할 때 BORLANDMM.DLL 파일을 같이 배포해야 한다. 물론 PChar, ShorString 데이터 형만 사용하는 경우에는 필요가 없다.

uses 절에 Windows.pas 유닛을 추가한 이유는 dll_ 상수에 대한 선언부가 있기 때문으로 이들의 의미는 다음과 같다.

상 수	값	의 미	용 도
dll_Process_Attach	1	프로세스가 DLL 에 매핑된다. 프로세스당 한번만 호출된다.	전역 객체나 변수를 초기화 하거나, 배치 프로세스를 처리한다.
dll_Thread_Attach	2	프로세스가 자식 쓰레드를 생성할 때 호출된다.	쓰레드에 대한 코드를 처리할 때 사용된다.
dll_Thread_Detach	3	쓰레드가 종료될 때 호출된다.	DLL 이 쓰레드 당 자원을 처리할 때 이를 해제하는 역할
dll_Process_Detach	0	프로세스에서 DLL 을 해제할 때 호출된다.	전역 객체나 변수, 파일, 포트 등을 해제하고 닫는다.

DLLMain 프로시저는 여러 차례 호출될 수 있으나, 이들이 호출되는 원인은 이렇게 4 가지이다. 프로세스는 DLL 을 여러 차례 적재할 수 있는데, 이들은 다중 쓰레드로 실행된다. 그런데 이때 DLL 이 주소에 매핑되는 것은 첫번째 호출에 의해서 결정되며, 그 이후의 호출은 참조 계수를 증가시키게 된다.
델파이는 암시적으로 시스템 유닛에 정의된 모든 기능을 포함하고 있다. 시스템 유닛은 _StartLib 라는 어셈블리 언어 함수를 정의하고 있다. 이 루틴은 DLLProc 라는 시스템 레벨 변수에 대입된 모든 코드를 실행할 책임이 있다. DLLProc 변수는 일종의 포인터 변수로 디폴트 값은 nil 이다. 여기에 DLLMain 함수의 주소를 대입하면 DLLMain 함수가 자동으로 호출된다. 또한, 델파이 프로젝트 파일의 begin ... end. 사이의 부분은 초기화를 담당하는 부분으로 프로세스가 처음 DLL 을 불러들일 때 한번만 호출되기 때문에 dll_Process_Attahc 와 같은 역할을 하게 된다.
DLLMain 함수는 DLL 프로젝트에 꼭 사용해야 하는 것은 아니고, 옵션으로 이용하는 것이다. 그렇지만, 전역 변수를 초기화하고 사용할 객체를 인스턴스화 하는 등의 초기 작업을 하는데 유용하게 사용할 수 있다. 다음과 같이 어플리케이션의 전역 예외 처리를 위한 객체를 생성하여 사용하는 것이 대표적인 예가 된다.

```
library Exam2;

uses
  ShareMem, Windows, SysUtils, Classes, Forms;

type
  TErrorHandler = class
```

```
    procedure ErrorHandle(Sender: TObject; E: Exception);
  end;

procedure TErrorHandler.ErrorHandle(Sender: TObject; E: Exception);
begin
  //전역 에러 처리 루틴
end;

var
  ErrorHandler: TErrorHandler;

procedure DLLMain(dwReason: DWORD);
begin
  case dwReason of
    dll_Process_Attach:
    begin
      ErrorHandler := TErrorHandler.Create;
      Application.OnException := ErrorHandler.ErrorHandle;
    end;
    dll_Process_Detach:
    begin
      Application.OnException := nil;
      ErrorHandler.Free;
    end;
    dll_Thread_Attach: ;
    dll_Thread_Detach: ;
  end;
end;

begin
  DLLProc := @DLLMain;
  DLLMain(dll_Process_Attach);
end.
```

먼저 Application 객체를 사용하기 위해 Forms.pas 유닛을 uses 절에 추가한다. 그리고, Application 객체의 OnException 이벤트 핸들러에 TErrorHandler 클래스의 ErrorHandle

프로시저로 대입한다. 여기에서 ErrorHandle 프로시저에서 에러에 대해 아무런 작업을 하지 않았지만, 에러 메시지를 보여주는 등의 작업을 하도록 정의할 수 있다.
이 밖에도 라이브러리 초기화 코드로 ExitProc 변수를 이용하여 DLL 이 메모리에서 해제될 때 실행되는 Exit 프로시저를 설치할 수도 있다. 다음의 코드를 살펴 보자.

```
library Test;

var
  SaveExit: Pointer;

procedure LibExit;
begin
  ...
  ExitProc := SaveExit;          //원래의 내용을 복구 한다.
end;

begin
  ...
  SaveExit := ExitProc;          //복구를 위해 위치를 저장한다.
  ExitProc := @LibExit;          //LibExit 프로시저를 ExitProc 로 저장한다.
end.
```

이 코드에 의해 DLL 이 메모리에서 해제될 때 라이브러리의 exit 프로시저가 ExitProc 변수가 nil 이 될 때까지 ExitProc 에 저장된 주소를 반복적으로 호출하게 된다.

- 호출 규칙 (Calling Conventions)

호출 규칙은 메소드나 함수에 변수를 넘기는 프로토콜을 정의하는 것이다. 호출 규칙은 함수에 넘겨 줄 파라미터의 순서와, 파라미터를 넘길 때 CPU 레지스터 사용 여부 등을 지정한다. 또한, 스택의 내용을 정리하는 책임을 호출한(caller) 함수가 가지는 지 아니면 호출된(callee) 함수가 가지는 지 결정한다.
델파이에서 사용되는 호출 규칙에는 다음과 같은 것들이 있다.

호출 규칙	파라미터 순서	스택 청소 책임
Fastcall(Register)	좌측에서 우측으로	호출된 루틴
Stdcall	우측에서 좌측으로	호출된 루틴

Pascal	좌측에서 우측으로	호출된 루틴
Cdecl	우측에서 좌측으로	호출하는 루틴
Safecall	우측에서 좌측으로	호출된 루틴

디폴트 호출 규칙은 Fastcall 이다. 가장 효과적인 프로토콜이며, CPU 레지스터를 이용하여 처음 3 개의 파라미터를 넘긴다. 그렇기 때문에, 수행속도가 빠르다. Pascal 호출 규칙은 델파이 1.0 에서 사용하던 호출 규칙으로 윈도우 3.1 의 표준 호출 규칙이다. Win32 에서는 Stdcall 로 표준 호출 규칙을 변경하였다.
그러므로, 작성한 DLL 을 델파이에서 사용할 것이라면 지정한 호출 규칙만 알면 어떤 것을 사용해도 상관이 없다. 그러나, C++ 을 비롯한 다른 언어와의 호환성을 위해서는 Win32 의 표준 호출 규칙인 Stdcall 을 사용하는 것이 좋다.

- DLL 의 예외와 런타임 에러

예외가 발생하지만 DLL 에서 처리하지 못한 경우에는 예외가 DLL 을 호출한 어플리케이션으로 전달된다. 그러므로, DLL 에서 발생한 에러라 할 지라도 이를 호출한 어플리케이션 코드에서 try ... except 문을 사용하여 처리가 가능하다. 만약, DLL 이 오브젝트 파스칼이 아닌 다른 언어에서 사용될 때에는 예외 코드 $0EEDFACE 를 처리하면 된다. 운영체제의 예외 레코드의 배열인 ExceptionInformation 의 첫번째 entry 에는 예외가 발생한 주소가를 담고 있으며, 두번째 entry 에는 오브젝트 파스칼 예외 객체의 레퍼런스를 저장하고 있다.
DLL 이 SysUtils.pas 유닛을 사용하지 않는 경우에는 델파이가 예외 처리를 제대로 하지 못한다. 이런 경우에는 런타임 에러가 DLL 에서 발생한 경우, 이를 호출한 어플리케이션이 중단된다. DLL 은 오브젝트 파스칼 프로그램에서 호출될 지를 알 수 있는 방법이 없기 때문에, 어플리케이션의 exit 프로시저를 호출할 수도 없다. 그러므로, 어플리케이션의 동작이 중지되면서 메모리에서 삭제된다.

정 리 (Summary)

이번 장에서는 DLL 에 대한 정보를 알 때, 이를 선언하여 델파이에서 사용하는 방법과 간단한 DLL 파일의 제작 방법에 대해서 알아 보았다.
물론 DLL 을 이용하여 리소스를 저장하거나, 폼을 DLL 로 제작하는 등의 다소 고급스러운 내용에 대해서는 다루지 않았지만 여기에 대해서는 비교적 많은 책들에서 다루고 있고, 또한 Inprise 의 TI, FAQ 등에서도 자주 나오는 내용이기 때문에 생략하였다