9장 DLL의 활용

* DLL의 정확한 개념
* 정적 링크와 동적 링크
* DLL을 사용하는 이유
* DLL의 작성과 사용
* DLL의 폼을 모듈리스로 띄우기
* 델파이 애플리케이션에서 DLL을 사용하려면
* DLL의 진입/종료 함수
* DLL의 예외 처리
* 콜백 함수
* 콜백 함수를 호출하는 DLL 만들기
* 프로세스들의 DLL 데이터 공유
* DLL에서 객체를 뽑아내려면
* DLL로 구현하는 로컬라이즈
* CPL(Control Panel Applet) 마법사

이번 장은 Win32의 동적 연결 라이브러리(Dynamic link library, 줄여서 DLL)에 대해서 알아보겠다. DLL은 윈도우즈 애플리케이션을 작성할 때 핵심적인 요소이다. 이번 장은 DLL의 작성과 활용에서 중요한 몇 가지 측면들을 이야기한다. DLL을 적재하는 여러 가지 방법들이나, DLL이 제공하는 프로시저와 함수들을 사용하는 방법을 배우게 될 것이다. 또한, 콜백 함수의 사용법이나, 서로 다른 프로세스들에서 DLL의 데이터를 공유하는 방법도 이야기하겠다.

DLL의 정확한 개념

동적 연결 라이브러리란, 여러 윈도우즈 애플리케이션들이 공유할 수 있는 코드나 데이터, 리소스 등을 담고 있는 프로그램 모듈이다. DLL의 주된 장점 중 하나는, 컴파일 시점이 아니라 실행 시점에서 외부 코드를 애플리케이션에 동적으로 연결시킬 수 있다는 점이다. 따라서, 하나 이상의 애플리케이션들이 DLL이 제공하는 동일한 코드를 동시에 사용할 수 있다. 사실, Win32 운영체제의 핵심 기능들은 KERNEL32, USER32.DLL, 그리고 GDI32.DLL에 들어 있다. KERNEL32.DLL은 메모리, 프로세스, 스레드 관리를 담당하며, USER32.DLL에는 윈도우의 생성과 Win32 메시지를 처리하는 사용자 인터페이스 루틴들이 들어 있다. 그리고 GDI32.DLL은 그래픽 처리를 담당한다. 그 이외에도, 객체의 보안/레지스트리 조작과 공용 대화상자를 처리하는 ADVAPI32.DLL이나 COMDLG32.DLL 같은 시스템 DLL들이 있다.

DLL의 또 다른 장점은, 애플리케이션을 좀더 모듈화시킬 수 있다는 것이다. 즉, 애플리케이션의 여러 기능들을 모두 하나의 실행 파일에 넣는 대신, 여러 개의 DLL들로 분리시켜 놓으면 나중에 특정 기능을 개선했을 때 애플리케이션 전체를 교체할 필요 없이 그 기능을 담당하는 DLL만 교체하면 된다. Win32 운영체제 자체가 그런 기법을 사용한다. 예를 들어서, 사운드 카드를 바꿨다고 해서 윈도우즈 자체를 다시 설치하지는 않는다. 사운드 카드 드라이버만 바꾸면 되는 것이다.

DLL의 구체적인 구조는 윈도우즈 EXE 파일과 같다. DLL과 EXE의 가장 큰 차이점이라면, DLL도 실행 코드를 가지고 있긴 하지만 자체적으로 실행되지는 않는다는 것이다. DLL 파일은 대부분 .dll이라는 확장자를 가지지만, 장치 드라이버일 경우는 .drv, 시스템 파일일 경우는 .sys라는 확장자를 가지기도 하며, 폰트 리소스일 경우는 .fon이라는 확장자를 사용한다. 단, .fon 파일에는 실행 코드가 없다.

DLL은 이름 그대로 동적으로 애플리케이션에 연결된다. 동적 링크에 대해서는 나중에 다시 이야기하겠다. 일반적으로, 어떤 애플리케이션이 DLL을 사용할 때 Win32 운영체제는 메모리에 DLL의 복사본(원본은 디스크에 있는 .dll 파일)을 단 하나만 올라오도록 한다. 그때 사용하는 기법이 메모리 맵 방식의 파일(memory-mapped file)이라고 한다. DLL은 우선 Win32 운영체제의 전역 힙 메모리 영역에 들어간다. 그런 다음, DLL을 호출한 프로세스의 주소 공간에 맵핑(mapping)된다. Win32 운영체제에서 각각의 프로세스는 개별적으로 4GB의 선형 메모리 공간을 부여받는다. 여러 프로세스가 하나의 DLL을 적재한 경우, 각각의 프로세스는 자신만의 DLL 이미지를 가진다. 따라서, 16비트 윈도우즈와는 달리 Win32에서는 DLL의 코드나 데이터, 리소스가 공유되지 않는다.

각각의 프로세스가 자신만의 DLL 이미지를 가진다고 했는데, 그 앞에서는 Win32가 메모리에 단 하나의 복사본만 존재하도록 한다고 했었다. 두 말이 모순되는 것은 아니다. x86의 메모리 관리 구조에 대한 기초 지식이 전혀 없는 독자라면 이해하기가 좀 힘들겠지만, 어쨌든 파일 상태의 DLL의 복사본은 운영체제의 전역 힙에 존재하며, 그 복사본이 각 프로세스의 주소 공간에 이미지로 맵핑되는 것이다. 따라서, 하나의 DLL을 여러 프로세스가 적재한다고 해서 물리적인 메모리가 여러 번 소비되는 것은 아니다(물론, 이상적인 상황에서의 이야기이고, 애플리케이션의 상황에 따라서는 실제로 물리적 메모리를 여러 번 소비할 수도 있다. 아래 박스를 참고).

DLL의 우선적 기준 주소 설정

DLL의 복사본이 물리적 메모리 공간을 한 번만 차지할 수 있으려면 그 DLL이 각 프로세스의 주소 공간 중 우선적 기준 주소에 적재될 수 있어야 한다. 우선적 기준 주소와 DLL의 주소 범위에 해당하는 주소 공간이 이미 할당되어 있으면 Win32의 로더(loader)는 전체 DLL 이미지를 다른 기준 주소로 옮겨야 하며, 그렇게 되면 그 프로세스는 전역 힙의 DLL 복사본을 다른 프로세스와 공유할 수 없게 된다. 결과적으로, 그 프로세스의 DLL 이미지는 물리적 메모리와 스왑 파일을 실제로 소비한다.
따라서, DLL을 작성하는 경우 그 DLL의 우선적 기준 주소가 애플리케이션이 사용하는 다른 주소 공간과 겹치지 않도록 하는 것이 중요하다.
우선적 기준 주소는 $IMAGEBASE 지시자를 이용해서 지정할 수 있다.

주소 공간이 겹치지 않게 하려면 애플리케이션이 할당하는 메모리는 주로 아래쪽 끝부터 올라오며, 공용 DLL들(VCL 패키지 등)은 윗쪽 끝에서부터 내려오므로 그 사이의 공간을 선택하는 것이 좋다. EXE와 DLL의 기본 기준 주소는 둘 다 $40000이므로 한쪽을 바꾸지 않는 한 DLL의 공유는 불가능하다.
기준 주소 로딩에는 부차적인 이점도 있다. 일반적으로, DLL은 메모리 재배치나 수정이 필요 없으며, 지역 하드 디스크에 존재하므로 DLL의 메모리 페이지는 디스크의 DLL 파일에 직접 맵핑할 수 있다. 그렇게 되면 DLL 코드는 시스템의 페이지 파일(즉, 스왑 파일)을 소비하지 않는다. 운영체제가 소비한 페이지의 크기가 시스템의 스왑 파일과 물리적 램의 크기를 더한 값보다 클 수 있는 이유도 바로 그것이다.

다음은 DLL에 관련된 몇 가지 용어들이다.

- 애플리케이션(application) : exe 파일로 실행되는 윈도우즈 프로그램
- 실행 파일(excutable) : 실행 가능한 코드를 담고 있는 파일. .dll과 .exe 등
- 인스턴스(instance) : 디스크에 있는 DLL 파일이 하나의 틀이라면, 메모리에 존재하는 DLL 코드는 그 틀로 찍어낸 인스턴스이다. 원본 또는 틀의 구체적 실현물을 인스턴스라고 한다. Win32는 각각의 인스턴스마다 고유한 인스턴스 핸들을 부여한다. 예를 들어서, 하나의 애플리케이션을 두 번 실행시키면 두 개의 애플리케이션 인스턴스가 존재하며, 따라서 인스턴스 핸들도 두 개다. DLL이 적재되면 역시 DLL 인스턴스와 인스턴스 핸들이 생긴다. 이번 장에서 인스턴스는 기본적으로 애플리케이션 또는 DLL의 인스턴스를 의미하는 것이니 클래스나 객체의 인스턴스와 혼동하지 말기 바란다.
- 모듈(module) : 16비트 윈도우즈에서와는 달리, 32비트 윈도우즈에서 모듈과 인스턴스는 동일한 의미로 쓰인다. 16비트 윈도우즈는 운영체제가 모듈을 관리하는 데이터베이스를 유지하며 각각의 모듈에 모듈 핸들을 부여한다. Win32의 경우, 한 애플리케이션의 각 인스턴스들은 자신만의 주소 공간을 가지므로 개별적인 모듈 핸들은 필요치 않다. 그러나 마이크로소프트는 아직도 모듈이라는 용어를 쓰고 있다. Win32에서 모듈은 인스턴스와 같은 의미라는 점을 기억하기 바란다.
- 작업(task) : 작업이란, 윈도우즈가 시스템 자원과 시간을 할당하는 기본 요소이다. 윈도우즈는 멀티태스킹, 즉 다중 작업(또는 작업 전환) 환경이다. 멀티태스킹이 가능하려면 운영체제는 여러 인스턴스들에 시스템 자원과 시간을 할당할 수 있어야 한다. Win32는 인스턴스 핸들이나 기타 작업 전환에 필요한 정보들을 관리하는 작업 데이터베이스를 통해서 멀티태스킹을 구현한다.

정적 링크와 동적 링크

고전적인 개념에서 링크라는 용어는 소스 코드를 컴파일할 때 함수 라이브러리에 있는 외부 코드(메소드, 프로시저, 변수, 형 지정 상수 등등)를 실행 파일에 포함시키는 작업을 말한다. 그것이 바로 정적 링크이다. 다시 말해, 정적 링크는 컴파일 시점에서 외부 코드를 애플리케이션의 실행 파일에 포함시키는 것을 말하며, 동적 링크는 개별적인 파일로 존재하는 DLL을 실행 시점에서 애플리케이션에 링크시키는 것을 말한다. 특별한 지정이 없는 한 어떤 유닛의 uses절에 있는 다른 유닛의 코드는 정적 링크로 그 유닛의 실행 파일에 포함된다.

정적 링크는 단순히 외부 코드를 애플리케이션의 실행 파일에 집어넣을 뿐만 아니라, 함수(메소드나 프로시저)의 주소도 애플리케이션에 맞게 적절히 변환시킨다. 함수를 호출한다는 것은 그 함수의 주소로 프로그램의 제어권을 옮겨서 함수의 코드를 실행한 다음, 다시 함수를 호출한 곳에 제어권을 돌려주는 과정을 통틀어서 말하는 것이다(물론, 인자를 넘기고 리턴값을 받는 과정도 포함된다). 라이브러리의 함수 주소를 그대로 사용할 수는 없으므로 링크 과정에서 함수의 주소가 적절히 변환된다.

물론, 컴파일러 관련 기술은 소프트웨어에서 가장 어렵고 복잡한 분야들 중 하나이므로 실제적인 링크 과정은 이보다 훨씬 더 복잡하다. 어쨌든, 정적 링크는 함수의 주소를 변환하는 작업과 함수의 코드를 애플리케이션의 실행 파일에 포함시키는 작업을 수행한다는 점만 기억하기 바란다.



델파이의 링커(링크를 수행하는 프로그램)는 유닛 단위로 작업하지 않는다. 다시 말해서, 같은 유닛에 있는 함수라도 프로젝트에서 실제로 쓰이지 않으면 실행 파일에 포함되지 않는다. 따라서, 필요 없이 실행 파일이 커지는 일은 없다.

예를 들어서, 두 개의 애플리케이션이 하나의 유닛에 있는 어떤 함수를 정적 링크를 통해 각각의 실행 파일에 포함시켰다고 가정하자. 물론, 애플리케이션은 모두 uses절에 그 유닛의 이름을 포함하고 있다. 두 애플리케이션이 동시에 실행중이라면 그 함수는 메모리에 두 번 존재한다. 만일 또 다른 하나의 애플리케이션을 실행시킨다면 동일한 함수가 메모리에 세 번 존재하게 되므로 낭비이다. 동적 링크의 주된 목적들

중 하나가 이러한 단점을 극복하는 것이다. 동적 링크의 경우, 함수는 DLL에 존재한다. 애플리케이션이 함수를 사용하기 위해 DLL을 메모리에 적재하면 DLL은 운영체제의 전역 힙에 적재되며, 애플리케이션의 고유한 주소 공간에 DLL의 이미지가 맵핑된다. 다른 애플리케이션이 그 함수를 사용하기 위해서는 DLL의 이미지를 자신의 주소 공간에 맵핑하기만 하면 되므로 결과적으로 하나의 함수는 물리적 메모리에 단 한 번만 존재한다(물론, 이상적인 경우의 이야기이다).

동적 링크에서 함수 호출과 실행 파일 코드의 연결은 DLL의 함수에 대한 외부 참조를 통해 실행 시점에서 일어난다. 외부 참조는 애플리케이션 안에서 선언하기도 하지만, 개별적인 도입(import) 유닛에서 선언하는 것이 일반적이다.

예를 들어서, MAXLIB.DLL이란 DLL에 다음과 같은 함수가 있다고 하자.

```
function Max(i1, I2: Integer): Integer;
```

이 함수는 인자로 받은 두 개의 정수 중 큰 값을 리턴해 준다. 위의 함수를 다른 애플리케이션에서 사용한다고 했을 때 도입 유닛은 다음과 같은 형태가 될 것이다.

```
unit MaxUnit;
interface
 function Max(i1, I2, Integer): Integer;
implementaion
 function Max; external 'MAXLIB';
end.
```

도입 유닛 역시 일반적인 오브젝트 파스칼 유닛의 형태이지만, Max()라는 함수를 구체적으로 정의하고 있지는 않다. 대신, external이라는 키워드를 이용해서 Max()라는 함수가 MAXLIB.DLL에 있다는 것을 알려주기만 하는 것이다. 이 도입 유닛을 애플리케이션이 사용하려면 그냥 uses절에 이 유닛의 이름인 MaxUnit을 포함시키기만 하면 된다. 애플리케이션을 실행하면 DLL이 자동으로 메모리에 적재되며, Max()를 호출하면 DLL에 있는 Max() 함수에 "링크"된다.

external 키워드를 사용하는 방식은 DLL을 적재하는 두 가지 방법 중 하나로, 흔히 묵시적 적재(implicit loading)라고 한다. 즉, 도입 유닛에만 명시적으로 DLL의 이름이 들어 있을 뿐, 나머지 코드에서는 Max()가 DLL에 있는 것인지 아닌지를 알 필요가 없다. 또 하나의 방법은 명시적 적재(explicit load)이다. 이에 대해서는 잠시 후에 이야기하겠다.

DLL을 사용하는 이유

DLL을 사용하는 이유와 장점에 대해서는 앞에서도 조금 이야기했다. 일반적으로 말해서, DLL은 코드나 시스템 자원을 공유하기 위해서, 그리고 코드의 구현이나 저수준 시스템 루틴들을 은폐하기 위해서, 또는 사용자 컨트롤을 설계하기 위해서 DLL을 사용한다. 하나씩 구체적으로 알아보자.

코드, 리소스, 데이터를 여러 애플리케이션이 공유

앞에서 DLL의 가장 큰 장점은 코드를 공유할 수 있다는 점이라고 했다. 유닛은 델파이 애플리케이션에만 의미가 있는 반면, DLL은 윈도우즈 표준 형식이므로 다른 언어로 개발된 윈도우즈 애플리케이션들도 사용할 수 있다.

또한, DLL은 실행 코드뿐만 아니라 비트맵이나 폰트, 아이콘 같은 리소스들도 담을 수 있으며, 그 리소스들 역시 실행 시점에서 애플리케이션에 링크된다. 따라서, 자주 쓰이는 아이콘들은 DLL로 만들어 놓으면 매번 실행 파일에 포함시킬 필요가 없다. 윈도우즈의 Moricons.dll이 대표적인 예이다.

16비트 윈도우즈의 경우, DLL은 자신의 데이터 세그먼트를 가지기 때문에 그 DLL을 사용하는 모든 애플리케이션들은 DLL의 메모리 공간을 통해서 전역 또는 정적 데이터를 공유할 수 있었다. 그러나 Win32의 경우는 그렇지 않다. DLL의 이미지가 각 프로세스의 고유한 주소 공간에 맵핑되기 때문에 DLL에 속한 모든 데이터는 그 프로세스에만 속한다. 그런데 프로세스가 다르면 DLL의 데이터를 공유할 수 없지만, 한 프로세스에 속한 여러 스레드들은 DLL의 데이터를 공유할 수 있다는 점을 기억하기 바란다. 스레드는 개별적으로 그리고 병렬적으로 실행되는 반면, DLL의 전역 데이터를 공유할 수 있으므로 한 스레드가 다른 스레드의 작업을 고려하지 않고 DLL의 데이터를 수정하면 문제가 생긴다. 멀티스레딩을 구현할 때 골치아픈 부분 중 하나이다.

그러나 DLL을 통해 데이터를 공유할 수 있는 길이 완전히 없어진 것은 아니다. DLL에 메모리 맵 파일을 이용해서 공유 메모리 영역을 만드는 방법이 있다. DLL을 사용하는 각각의 프로세스(애플리케이션)들은 공유 메모리 영역에 데이터를 쓰거나 읽으면 되는데, 이에 대해서는 잠시 후에 이야기하겠다.

구체적인 구현을 은폐

DLL을 통해서 함수나 프로시저의 구체적인 구현을 외부로부터 숨길 수 있다는 것도 DLL의 장점이다. DLL을 사용하는 애플리케이션은 DLL이 제공하는 기능이 어떻게 이루어지는지에 대해서는 알 필요가 없으며, 인터페이스, 즉 함수의 이름이나 인자의 구성만 알면 된다. 또한, 인터페이스가 변하지 않는 한 DLL 자체의 구체적인 구현을 바꿔도 애플리케이션들을 일일이 바꿀 필요가 없다. 물론, 은폐화(캡슐화, 모듈화)는 델파이의 DCU로도 충분히 구현할 수 있는 것이지만, DCU의 경우는 같은 버전의 델파이로 만들어진 델파이 애플리케이션에서만 사용할 수 있다는 단점이 있다. DLL은 언어에 독립적이므로 인터페이스만 알면 C++나 VB 같은 다른 개발 도구에서도 그 DLL을 사용하는 애플리케이션을 만들 수 있다.

델파이의 Windows 유닛은 Win32의 시스템 DLL들에 대한 인터페이스 정의가 모여 있는 유닛이다. 델파이 5에는 Win32 API 유닛 소스 파일들이 포함되어 있는데, 그 중 WINDOWS.PAS가 Windows 유닛의 소스 파일이다. WINDOWS.PAS의 interface절을 보면 다음과 같은 함수 정의가 있다.

```
function ClientToScreen(hWnd: HWND; var lpPoint: TPoint): BOOL; stdcall;
```

위의 함수는 implementation절에 다음과 같이 구현되어 있다.

```
function ClientToScreen; external user32 name 'ClientToScreen';
```

위의 코드는 ClientToScreen()이 윈도우즈의 시스템 DLL들 중 하나인 USER32.DLL에 있는 ClientToScreen이라는 함수라는 뜻이다.

사용자 컨트롤

사용자 컨트롤들은 보통 DLL의 형태를 가진다. 사용자 컨트롤은 델파이의 사용자 정의 컴포넌트와는 다른 형태이다. 사용자 컨트롤은 윈도우즈에 등록되며, 언어에 상관 없이 사용할 수 있다. 사용자 컨트롤 역시 여러 애플리케이션들이 동시에 사용하는 경우가 많기 때문에 DLL로 만들어지는 것이 당연하다.



이전의 사용자 컨트롤 DLL(vbx 등)의 구조는 상당히 거칠고 융통성이 없었기 때문에 마이크로소프트는 이제 OLE 컨트롤 또는 ActiveX 컨트롤을 사용한다. 이전의 사용자 컨트롤들은 현재 거의 사용되지 않는다.

DLL의 작성과 사용

그럼, 실제로 델파이에서 DLL을 작성하고 사용하는 방법을 알아보자. DLL을 다른 애플리케이션들이 사용할 수 있도록 인터페이스 유닛을 만드는 방법도 나오고, 또 델파이 폼을 DLL에 집어넣는 방법도 나오니 찬찬히 읽어보기 바란다.

동전 계산 예제 DLL

전산학 관련 서적이나 강의를 보면 "100원, 50원, 10원짜리 동전을 최소의 개수만 사용해서 270원을 만들어라" 같은 문제가 나오는데, 그런 문제를 푸는 루틴을 DLL로 만들어보자. 페니(1/100달러, 즉 1센트) 단위의 돈을 최소의 니켈(nickel, 5센트)과 다임(dime, 10센트), 쿼터(quater, 25센트)로 쪼개는 루틴이다(역주 : 미국 화폐 단위를 사용했지만, 원 단위로 바꾸는 것도 어렵지 않을 것이다).

기본 DLL

이 DLL에는 PeniesToCoins() 메소드가 들어 있다. 목록 9.1이 DLL 프로젝트의 전체 코드이다(폼을 사용하지 않으므로 dpr 파일에 코드를 넣는다).



Ch9\Pennies\PenniesLip.bpr

```
library PenniesLib;
{$DEFINE PENNIESLIB}
uses
  SysUtils,
  Classes,
```

```
  PenniesInt;

function PenniesToCoins(TotPennies: word; CoinsRec: PCoinsRec): word;
 StdCall;
begin
  Result := TotPennies;  // Result에 값을 배정한다.
  { 동전들의 개수를 구한다.}
  with CoinsRec^ do
  begin
    Quarters    := TotPennies div 25;
    TotPennies  := TotPennies - Quarters * 25;
    Dimes       := TotPennies div 10;
    TotPennies  := TotPennies - Dimes * 10;
    Nickels     := TotPennies div 5;
    TotPennies  := TotPennies - Nickels * 5;
    Pennies     := TotPennies;
  end;
end;

{ 함수 이름을 노출(export)시킨다. }
exports
  PenniesToCoins;
 end.
```

이 라이브러리는 PenniesInt라는 유닛을 사용한다는 점을 주목하기 바란다. 이에 대해서는 다음 절에서 이야기하겠다.

export절은 DLL이 다른 애플리케이션들에게 제공할 함수나 프로시저를 지정하는 곳이다.

인터페이스 유닛의 정의

다른 애플리케이션들이 uses절을 통해서 DLL의 루틴을 묵시적으로 적재하려면 인터페이스 유닛, 즉 이전에 말한 도입(import) 유닛이 있어야 한다. 또한, DLL과 애플리케이션이 모두 사용하는 범용 구조체 등을 정의하는 용도로도 쓸 수 있다. 목록 9.2는 PenniesLib의 인터페이스 유닛인 PenniesInt.pas이다.

목록 9.2 Ch9\PePennies\PenniesInt.pas

```
unit PenniesInt;
{ PENNIES.DLL의 인터페이스 루틴 }

interface
type

 { 변환이 끝난 후 각 동전들의 개수들이 이 레코드에 저장된다. }
  PCoinsRec = ^TCoinsRec;
  TCoinsRec = record
    Quarters,
    Dimes,
    Nickels,
    Pennies: word;
  end;

{$IFNDEF PENNIESLIB}
{ export 키워드로 함수를 선언한다. }

function PenniesToCoins(TotPennies: word; CoinsRec: PCoinsRec): word;
StdCall;
{$ENDIF}

implementation

{$IFNDEF PENNIESLIB}
{ 도입할 함수를 정의한다. }
function PenniesToCoins; external 'PENNIESLIB.DLL' name 'PenniesToCoins';
{$ENDIF}

 end.
```

이 유닛의 type 섹션에는 TCoinsRec라는 레코드와 그 레코드에 대한 포인터가 선언되어 있다. PenniesToCoins()에 이 레코드를 넣어서 호출하면 변환이 끝난 후의 결과, 즉 각 동전들의 개수가 레코드의 각 필드에 들어간다. PenniesToCoins() 함수는 전체

금액과 TCoinsRec 변수의 포인터를 인자로 받는다. TCoinsRec의 각 필드를 조사하면 각 동전의 개수를 알아낼 수 있다.

PENNIESINT.PAS의 interface절에는 PENNIESLIB.DLL이 노출시킨 함수인 PenniesToCoins()가 선언되어 있다. PenniesToCoins()의 정의는 implementation절에 나와 있는데, 실제 코드가 있는 것이 아니라 external 키워드를 통해서 실제 코드가 있는 DLL의 이름이 명시되어 있다.

즉, 애플리케이션에서 PenniesToCoins()를 호출하면 PENNIESLIB.DLL에 있는 함수가 실행되는 것이다. 그런데 조건부 컴파일 지시문을 통해서 PENNIESLIB 지시자가 정의되어 있지 않은 경우에만 함수의 선언과 정의가 컴파일된다는 점을 주목하기 바란다. PENNIESLIB 지시자는 PenniesLib.dpr 파일에 정의되어 있다. PenniesLib.dpr을 컴파일해서 DLL을 만들 때에는 DLL에 구조체에 대한 정의만 포함시키면 되지, DLL의 함수에 대한 인터페이스는 필요가 없기 때문에 조건부 지시문을 쓴 것이다.



애플리케이션 전역 조건부 지시자는 Project | Options 대화상자의 Directories/Conditional 페이지에서 지정하면 된다. 조건부 지시자를 변경한 경우, 빌드(Build)를 다시 해야 효과가 나타난다는 점을 기억하기 바란다. Make는 변경된 지시자를 적용시키지 않는다.



다음은 DLL 함수를 도입하는 또 다른 예이다.

```
function PenniestoCoins; external 'PENNIESLIB.DLL' index 1;
```

위의 코드는 숫자(서수)를 이용해서 함수를 지정했다. 이전에 설명한 방식은 다음과 같은 것이었다.

```
function PenniestoCoins; external 'PENNIESLIB.DLL' name
'PenniesToCoins';
```

이 경우는 name 키워드를 써서 함수의 이름을 지정했다.
숫자를 사용하는 경우, DLL을 적재할 때 이름을 찾는 과정이 없기 때문에 적재 시간이 더 짧다. 그러나 Win32에서는 숫자를 사용하는 것을 권장하지 않는다. 왜냐하면, 이름을 이용하면 DLL의 진입점을 재배치하는 작업에 융통성이 생기기 때문이다. 숫자를 사용하면 DLL에서 그 함수의 위치가 중요하지만, 이름을 사용하면 이름과 실제 함수를 Win32가 알아서 일치시켜 주기 때문에 좀더 융통성이 있다.

만일 이번 프로젝트의 DLL이 개발자를 위한 것이라면 PenniesLib.dll뿐만 아니라, PenniesInt.pas도 배포해야 한다. 그러면 그 개발자들은 PenniesInt.pas에 정의된 레코드와 함수를 이용해서 DLL을 쉽게 사용할 수 있다. 또한, C++ 같은 다른 언어를 사용하는 개발자들도 사용할 수 있도록 하려면 PenniesInt.pas를 해당 언어에 맞게 변환해서 제공해야 한다.

DLL의 폼을 모달로 띄우기

이번에는 폼을 포함하고 있는 DLL로부터 폼을 모달로 띄우는 방법을 알아보자. 폼을 DLL로 만들면 델파이뿐만 아니라, C++나 비주얼 베이직, 그리고 Object PAL 같은 다른 윈도우즈 애플리케이션 개발 언어에서도 폼을 사용할 수 있다.

DLL 기반의 폼은 자동으로 생성되지 않도록 해야 한다(auto-create forms 목록에서 옵션 선택 제거).

이번 예제는 TCalendar 컴포넌트가 있는 폼을 DLL로 만드는 것이다. 폼을 띄우는 함수는 DLL에 포함되어 있으며, 애플리케이션이 그 함수를 호출하면 달력 모양의 폼이 나타난다. 사용자가 달력에서 어떤 날짜를 선택하면 DLL은 그 날짜를 호출한 애플리케이션에게 돌려준다.

목록 9.3은 DLL의 폼을 모달로 띄우는 예제 프로젝트 파일인 CalendarTest.dpr의 메인 폼 유닛이다. 그리고 목록 9.4는 DLL에 포함시킬 폼의 유닛인 DllFrm.pas의 소스 코드이다.

목록 9.3 Ch9\ModalDLL\DLLFrm.pas

```
unit DLLFrm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Grids, Calendar;

type

  TDLLForm = class(TForm)
    calDllCalendar: TCalendar;
    procedure calDllCalendarDblClick(Sender: TObject);
  end;

{ 공개할 함수를 선언 }
function ShowCalendar(AHandle: THandle; ACaption: String): TDateTime;
StdCall;

implementation
{$R *.DFM}

function ShowCalendar(AHandle: THandle; ACaption: String): TDateTime;
var
  DLLForm: TDllForm;
begin
  // 애플리케이션의 핸들을 DLL의 TApplication.Handle 속성에 넣는다.
  Application.Handle := AHandle;
  DLLForm := TDLLForm.Create(Application);
  try
    DLLForm.Caption := ACaption;
    DLLForm.ShowModal;
    Result := DLLForm.calDLLCalendar.CalendarDate;
    // 선택된 날짜를 돌려준다.
  finally
    DLLForm.Free;
```

```
  end;
end;

procedure TDLLForm.calDllCalendarDblClick(Sender: TObject);
begin
  Close;
 end;


 end.
```

이 DLL을 사용하는 예제 프로젝트의 메인 폼 유닛(CalendarTest.dpr의 MainFrm.pas)을 보면 함수 포인터와 LoadLibrayr() DLL을 이용해서 명시적으로 DLL을 적재하는데, 명시적 적재에 대해서는 나중에 다시 이야기하겠다.

DLL의 ShowCalendar() 함수는 우선 Application.Handle 속성에 인자로 받은 애플리케이션의 핸들을 넣는다. 4장에서 라이브러리 프로젝트를 포함한 델파이 프로젝트들은 전역 Application 객체를 가지고 있다는 이야기를 했었다. DLL의 Application 객체는 DLL을 호출하는 애플리케이션의 Application 객체와는 다른 것이다. DLL의 폼이 그 DLL을 호출하는 애플리케이션의 모달 폼이 되려면 호출하는 애플리케이션의 핸들을 DLL의 Application.Handle 속성에 넣어야 한다. 그렇게 하지 않으면 DLL의 폼이 제대로 동작하지 않는다. 특히, DLL의 폼을 최소화시키면 문제가 생긴다. 그리고 또 한 가지, DLL의 폼을 생성할 때 소유자를 nil로 해서는 안된다. 위의 코드는 호출한 애플리케이션을 DLL 폼의 소유자로 지정했다.

폼을 생성한 후에는 인자로 받은 ACaption을 DLL의 폼의 Caption에 넣고, DLL 폼을 모달로 띄운다. 폼이 닫히면 사용자가 TCalendar 컴포넌트에서 선택한 날짜를 호출한 애플리케이션에 돌려준다. 사용자가 TCalendar 컴포넌트를 더블클릭하면 폼이 닫힌다.



DLL이 델파이 문자열(string형 등)이나 동적 배열을 인자나 리턴값으로 사용하는 함수를 노출시키는 경우, 그 DLL 프로젝트의 프로젝트 유닛 소스(View | Project Source로 볼 수 있다)나 함수가 있는 유닛의 uses절에는 반드시 ShareMem이 첫 번째로 와야 한다. 어떤 형태로든 문자열을 주고 받아야 한다면 반드시 그렇게 해야 한다. 즉, 인자나 리턴값으로 주고 받은 구조체나 클래스 안에 문자열이나 동적 배열이 끼어 있으면 ShareMem을 반드시 첫 번째 자리에 놓아야 한다.

ShareMem은 공유 메모리 관리를 담당하는 BorIndmm.dll의 인터페이스 유닛이며, ShareMem을 사용하는 프로젝트를 배포할 때에는 BorIndmm.dll도 함께 배포해야 한다. string 대신 PChar나 ShortString을 사용하면 ShareMem이 필요 없으므로 BorIndmm.dll을 함께 배포할 필요도 없다.

ShareMem은 힙 메모리에 할당된 문자열이나 동적 배열을 서로 다른 모듈들이 주고 받을 때에만 필요한 것이다. 그러한 전송이 일어날 경우, 문자열뿐만 아니라 그 문자열의 소유권도 전송된다. 문자열을 PChar로 형 변환시킨 다음, PChar를 다른 모듈에 넘겨주면 소유권은 넘어가지 않으므로 ShareMem이 필요 없다.

이 ShareMem 문제는 델파이/C++ 빌더로 만든 DLL이 문자열이나 동적 배열을 다른 델파이/C++ 빌더 DLL이나 EXE로 넘길 때에만 적용되는 것이다. 델파이 고유의 문자열을 델파이가 아닌 언어로 만든 DLL이나 EXE와 주고 받아서는 안된다. PChar를 사용하는 것이 좋다.

또한, ShareMem은 하나의 패키지로 만들어진 모듈들 사이에는 필요하지 않다. 패키지 안에 포함된 모듈들의 메모리 공유는 델파이가 알아서 처리하기 때문이다.

이렇게 해서 모달 폼을 DLL에 넣는 방법에 대한 설명이 끝난다. 이 DLL을 사용하는 예제 프로젝트는 잠시 후에 설명하겠다.

DLL의 폼을 모들리스로 띄우기

DLL의 폼을 모들리스로 띄우는 것도 모달 폼과 거의 비슷하지만, 한 가지 중요한 차이가 있다.

DLL의 폼을 모들리스(modeless)로 띄우려면 그 DLL은 폼을 생성하고 표시하는 작업을 처리하는 루틴뿐만 아니라, 폼을 파괴/해제하는 작업을 처리하는 루틴도 제공해야 한다. 모들리스 폼의 파괴는 DLL 안에서 결정하는 것이 아니라, 호출한 애플리케이션이 결정하기 때문이다.


목록 9.4 Ch9\ModalessDll\DLLFrm.pas

```
unit DLLFrm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Grids, Calendar;

type

  TDLLForm = class(TForm)
    calDllCalendar: TCalendar;
  end;

{ 공개할 루틴들을 선언 }
function ShowCalendar(AHandle: THandle; ACaption: String): Longint;
stdCall;
procedure CloseCalendar(AFormRef: Longint); stdcall;

implementation
{$R *.DFM}

function ShowCalendar(AHandle: THandle; ACaption: String): Longint;
var
  DLLForm: TDllForm;
begin
  // 애플리케이션의 핸들을 DLL의 TApplication.Handle 속성에 넣는다.
```

```
  Application.Handle := AHandle;
  DLLForm := TDLLForm.Create(Application);
 Result := Longint(DLLForm);
  DLLForm.Caption := ACaption;
 DLLForm.Show;
end;

procedure CloseCalendar(AFormRef: Longint);
begin
  if AFormRef > 0 then
    TDLLForm(AFormRef).Free;
end;

 end.
```

이 DLL은 ShowCalendar()와 CloseCalendar()라는 루틴을 노출시킨다.

ShowCalendar()는 모달 폼 예제의 것과 마찬가지로, 호출한 애플리케이션의 핸들을 DLL의 Application.Handle에 넣는다. 그러나 ShowModal()로 폼을 띄우는 대신 Show()로 띄우며, 폼을 해제하지 않는다는 점이 틀리다. 또한, 이 함수의 리턴값은 선택된 날짜가 아니라 DLLForm의 인스턴스를 뜻하는 Longint값이다. 생성된 DLL 폼을 나중에 해제하기 위해서는 DLL 폼에 대한 참조가 필요한데, DLL 내부에서 관리하는 것보다는 호출한 애플리케이션이 관리하는 것이 편하기 때문에 리턴값으로 DLLForm의 인스턴스 참조를 돌려주는 것이다. 호출한 애플리케이션은 돌려받은 참조를 보존하고 있다가 나중에 DLL 폼을 파괴할 때 사용한다.

CloseCalendar() 프로시저는 인자로 넘겨 받은 참조가 유효한지 검사한 후 Free 메소드로 DLL 폼을 해제한다. 애플리케이션은 반드시 ShowCalendar()의 리턴값으로 받은 폼 참조를 인자로 넣어서 이 프로시저를 호출해야 한다.

모들리스 폼을 다룰 때에는 DLL이 호출한 애플리케이션과는 상관 없이 함부로 폼을 해제시키면 안된다. 만일 그렇게 하면(예를 들어서, CanClose()에서 caFree를 돌려준다든지) 나중에 애플리케이션이 CloseCalendar()를 호출했을 때 이미 해제된 폼을 다시 해제시키려고 하기 때문에 에러가 난다.

델파이 애플리케이션에서 DLL을 사용하려면

이번 장의 앞부분에서 DLL을 적재, 즉 도입하는 방법에는 묵시적 적재와 명시적 적재가 있다고 했다. 이제, 앞에서 만든 DLL들을 이용해서 두 방법에 대해 설명하겠다.

이번 장에서 첫 번째로 만들었던 DLL 프로젝트에는 인터페이스 유닛이 포함되어 있다. 그럼, 인터페이스 유닛을 사용해서 묵시적으로 DLL을 적재하는 예제를 만들어 보자. 이번 예제 프로젝트의 메인 폼은 하나의 TMaskEdit와 TButton, 그리고 아홉 개의 TLabel로 구성된다.

사용자가 TMaskEdit에 페니 단위의 금액을 입력하고 버튼을 클릭하면 그 금액을 구성하는 동전들의 개수가 TLabel들에 표시된다. 금액과 동전 개수의 변환은 PenniesLib.dll의 PenniesToCoins() 함수로 얻는다.

목록 9.5는 이번 프로젝트의 메인 폼 유닛(MainFrm.pas)이다.

목록 9.5 Ch9\Pennies\MainFrm.pas

```
unit MainFrm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Mask;

type

  TMainForm = class(TForm)
    lblTotal: TLabel;
    lblQlbl: TLabel;
    lblDlbl: TLabel;
    lblNlbl: TLabel;
    lblPlbl: TLabel;
    lblQuarters: TLabel;
    lblDimes: TLabel;
    lblNickels: TLabel;
    lblPennies: TLabel;
    btnMakeChange: TButton;
```

```
    meTotalPennies: TMaskEdit;
    procedure btnMakeChangeClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation
uses PenniesInt;  // 인터페이스 유닛을 사용

{$R *.DFM}

procedure TMainForm.btnMakeChangeClick(Sender: TObject);
var
  CoinsRec: TCoinsRec;
  TotPennies: word;
begin
  { DLL의 함수를 호출해서 사용자가 입력한 금액을 구성하는 동전들의 최소 개수를
    얻는다. }
  TotPennies := PenniesToCoins(StrToInt(meTotalPennies.Text), @CoinsRec);
  with CoinsRec do
  begin
    { 동전 개수들을 표시한다. }
    lblQuarters.Caption := IntToStr(Quarters);
    lblDimes.Caption    := IntToStr(Dimes);
    lblNickels.Caption  := IntToStr(Nickels);
    lblPennies.Caption  := IntToStr(Pennies);
  end
end;

end.
```

PenniesToCoins()라는 함수의 선언은 인터페이스 유닛인 PenniesInt.pas에 들어 있으므로 uses절에 PanniesInt를 포함시켰다. 이 애플리케이션을 실행시키면 Win32는 자동적으로, 즉 묵시적으로 PenniesLib.dll을 적재하고, DLL의 이미지를 애플리케이션의 프로세스 주소 공간에 맵핑한다.

물론, 인터페이스 유닛을 사용하지 않을 수도 있다. 일단 uses절에서 PenniesInt를 지운 다음, implementation절에 다음과 같이 PenniesToCoins()를 선언할 때 external 지시자와 DLL 이름을 써준다.

```
implementation

function PenniesToCoins(TotPennies: word; ChangeRec: PChangeRec): word;
  stdCall external 'PENNIESLIB.DLL';
```

물론, PChangeRec와 TChangeRec의 정의가 PenniesInt.pas에 들어 있으므로 그 부분도 MainFrm.pas에 복사해야 제대로 컴파일할 수 있다. 그러나 DLL의 함수가 많거나, DLL을 여러 프로젝트에서 사용한다면 위의 코드처럼 external을 이용해서 직접 함수를 선언하는 것은 짜증나는 일이다. 인터페이스 유닛을 만들어두면 uses절에 인터페이스 유닛 이름만 추가하면 되기 때문에 훨씬 편하다.



델파이로 만든 것이 아닌 DLL을 사용하는 경우, 그 DLL은 델파이 인터페이스 유닛을 제공하지 않으며, 대신 C/C++ 임포트 라이브러리를 제공하는 경우가 대부분이다. 그런 경우에는 임포트 라이브러리를 델파이 인터페이스 유닛으로 변환해야 한다.

명시적 DLL 적재

DLL을 묵시적으로 적재하는 것이 편하긴 해도, 효율성이 떨어지는 경우가 있다. 예를 들어서, 앞의 예제에서 사용자가 한 번도 버튼을 클릭하지 않고 그냥 종료시키면 DLL의 함수는 한 번도 사용되지 않는다. 그러나 묵시적으로 DLL을 적재했으므로 DLL은 항상 메모리에 올라온다. 만일 애플리케이션이 여러 개의 DLL을 사용하거나 DLL의 덩치가 크다면 메모리의 낭비가 더욱 심해질 것이다. 따라서, DLL의 함수를 실제로 사용할 필요가 있을 때만 DLL을 명시적으로 적재하는 것이 더 효율적이다. 많은 종류의 그래픽 파일 포맷을 지원하는 그래픽 프로그램의 경우가 대표적인 예이다. 즉, 사용자가 JPEG 파일을 선택하면 그때서야 JPEG 해석 DLL을 적재하는 것이다. 이러한 방식을 명시적 적재라고 한다.

그럼, 모달 폼 DLL을 명시적으로 적재하는 예제를 살펴보자. 목록 9.6은 모달 폼 DLL을 명시적으로 적재하는 예제 프로젝트의 메인 폼 유닛이다. 이 프로젝트는 부록 CD의 \ModalDLL 디렉토리에 있는 CalendarTest.dpr에 해당한다.

목록 9.6 Ch9\ModalDLL\MainFrm.pas

```
unit MainFfm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  { 우선, DLL이 제공하는 함수와 동일한 형태의 함수 형을 선언한다. }
  TShowCalendar = function (AHandle: THandle; ACaption: String):
      TDateTime; StdCall;

  { DLL의 적재가 실패하면 발생시킬 예외를 정의한다. }
  EDLLLoadError = class(Exception);

  TMainForm = class(TForm)
    lblDate: TLabel;
    btnGetCalendar: TButton;
    procedure btnGetCalendarClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.btnGetCalendarClick(Sender: TObject);
var
  LibHandle   : THandle;
```

```
  ShowCalendar: TShowCalendar;
begin

  { DLL 적재 시도 }
  LibHandle := LoadLibrary('CALENDARLIB.DLL');
  try
    { LibHandle이 0이면 실패한 것이다. 실패했으므로 예외를 발생시킨다. }
    if LibHandle = 0 then
      raise EDLLLoadError.Create('Unable to Load DLL');
      { 0이 아니면 성공이다. 이제, DLL의 함수를 이 유닛에 링크한다. }
    @ShowCalendar := GetProcAddress(LibHandle, 'ShowCalendar');
      { 링크가 성공했으면 ShowCalendar 함수를 호출하고, 함수의 리턴값을
        lblDate.Caption에 넣어서 화면에 표시한다. 성공이 아니면 예외를
        발생시킨다. }
    if not (@ShowCalendar = nil) then
      lblDate.Caption := DateToStr(ShowCalendar(Application.Handle,
Caption))
    else
      RaiseLastWin32Error;

  finally
    FreeLibrary(LibHandle); // DLL을 해제한다.
  end;
end;

end.
```

이 유닛은 우선 CalendarLib.dll에 있는 ShowCalendar()와 동일한 형태의 함수 형인 TShowCalendar를 정의한다. 그런 다음, DLL의 적재가 실패했을 때 발생시킬 예외도 정의한다. btnGetCalendarClick() 이벤트 처리 프로시저에서는 Win32 API 함수 Load Library()와 FreeLibrary(), GetProcAddress()를 사용해서 DLL을 다룬다.

LoadLibrary()는 다음과 같이 선언되어 있다.

```
function LoadLibrary(lpLibFileName: PChar): HMODULE; stdcall;
```

이 함수는 인자로 지정한 파일 이름에 해당하는 DLL 모듈을 적재하고, 그것을 호출한 프로세스의 주소 공간에 맵핑한다. 함수가 성공하면 DLL 모듈의 핸들을 돌려준

다. 리턴값이 0이면 적재가 실패한 것이므로 메인 유닛은 예외를 발생시킨다. LoadLibrary()의 자세한 사용법이나 주의 사항은 온라인 도움말을 참고하기 바란다.

FreeLibrary()는 다음과 같이 선언되어 있다.

```
function FreeLibrary(hLibModule: HMODULE): BOOL; stdcall;
```

FreeLibrary()는 hLibModule 인자로 지정된 DLL의 참조 횟수(인스턴스 개수)를 하나 감소시킨다. 만일 인스턴스 개수가 0이면 DLL을 메모리에서 제거한다. 인스턴스 개수는 DLL을 사용하는 작업의 개수를 뜻한다.

그리고 다음은 GetProcAddress()의 선언이다.

```
fuction GetProcAddress(hModule: HMODULE; lpProcName: LPCSTR); FARPROC;
stdcall
```

GetProcAddress()는 첫 번째 인자 hModule로 지정된 DLL에서 lpProcName으로 지정된 이름의 함수를 찾아서 그 함수의 주소를 돌려준다. hModule에는 LoadLibrary()로 얻은 핸들을 넣어야 한다. 만일 GetProcAddress()가 실패하면 nil을 돌려준다. 리턴값이 nil인 경우, GetLastError()를 호출하면 자세한 에러 정보를 얻을 수 있다.

사용자가 btnGetCalendar 버튼을 클릭하면 그 버튼의 OnClick 이벤트 처리 프로시저는 LoadLibrary()를 호출해서 CALENDARLIB.DLL을 적재한다. 호출이 성공하면 GetProcAddress()를 호출해서 DLL의 ShowCalendar() 함수의 주소를 얻고, 그 주소를 함수 형 TShowCalendar의 변수인 ShowCalendar에 넣는다. 이때 주소 연산자 @를 사용했음을 주목하기 바란다. ShowCalendar가 nil이면 주소를 얻지 못한 것이므로 예외를 발생시키고, nil이 아니면 ShowCalendar() 함수, 즉 ShowCalendar의 주소에 있는 함수를 호출한다. 마지막으로, FreeLibrary()를 호출해서 DLL의 사용을 끝내는데, 예외가 발생해도 FreeLibrary()가 반드시 수행되도록 finally 블록에서 FreeLibrary()를 호출한다는 점도 주목하기 바란다.

CALENDARLIB.DLL은 버튼이 클릭될 때마다 적재/해제를 반복한다. 사용자가 버튼을 단 한 번만 클릭하고 애플리케이션을 끝낸다면 명시적 적재의 장점이 최대한 살아나는 것이다. 그러나 사용자가 버튼을 여러 번 클릭한다면, 즉 DLL의 적재/해제가 여러 번 일어난다면 묵시적 적재보다 비효율적이다. 결론적으로 말하면, DLL의 명시적 적재와 묵시적 적재를 결정할 때에는 그 DLL이 얼마나 자주 사용될 것인지를 기준으로 삼아야 한다.

DLL의 진입/종료 함수

DLL의 초기화나 해제시에 특별한 작업들이 필요하다면 DLL의 추가적인 진입/종료 함수를 정의하면 된다. 그러한 작업들은 프로세스나 스레드의 초기화/종료 과정에 집어넣는다.

프로세스/스레드의 초기화와 종료 루틴들

초기화 작업에는 보통 윈도우즈 클래스 등록이나 전역 변수들의 초기화, 또는 진입/종료 함수의 초기화 등이 포함된다. DLL의 최초의 초기화 코드는 프로젝트 소스, 즉 dpr 파일의 begin..end. 블록의 코드이다. 그리고 프로세스와 DLL의 적재/제거 관련 작업이 일어날 때 실행될 루틴을 직접 정의할 수 있는데, 그런 루틴을 DLL의 진입점 또는 진입/종료 루틴(보통, DLLEntryPoint()라는 이름을 쓴다)이라고 한다. DLLEntryPoint()는 프로세스가 DLL을 적재하거나 제거할 때 실행되는(Win32의 내부적인 처리를 제외하고) 루틴이며, 구체적인 정의는 개발자가 하면 된다. DLLEntryPoint는 반드시 DWord형의 인자 하나만을 받아들이도록 정의해야 한다.

DLL 프로젝트의 진입점 루틴을 가리키는 전역 변수가 DLLProc이다. 이 변수는 프로시저형 포인터 변수이며, 특별히 설정하지 않는 한 이 변수의 값은 nil이다. 이 변수에 어떤 프로시저를 설정하면 DLL이 초기화/해제될 때 그 프로시저가 호출된다. 이때 DWord값을 인자로 받는데, DWord 인자(dwReason)는 일종의 이벤트 메시지라고 생각하면 된다. DWord 인자에 들어갈 수 있는 값은 표 9.1과 같다.

표 9.1 DLL 진입/종료 이벤트

이벤트	의 미
DLL_PROCESS_ATTACH	프로세스가 시작되었거나, 프로세스가 LoadLibrary() 함수를 호출 DLL의 인스턴스 데이터를 초기화하는 작업을 수행하면 된다.
DLL_PROCESS_DETACH	호출한 프로세스의 주소 공간으로부터 DLL이 제거됨. 프로세스 자체가 종료되거나 프로세스에서 FreeLibrary()를 호출한 경우 발생한다. 이 이벤트가 오면 인스턴스 데이터의 종료 처리를 하면 된다.
DLL_TREAD_ATTACH	현재 프로세스가 새 스레드를 생성한 경우이다. 프로세스가 새 스레드를 생성하면 운영체제는 그 프로세스에 맵핑된 모든 DLL들의 진입점 함수를 호출한다. 진입점 함수 호출은 새 스레드의 문맥하에서 일어나므로 스레드 관련 데이터의 할당이나 초기화는 이 이벤트에서 수행하면 된다.
DLL_THREAD_DETACH	스레드가 종료되면 발생한다. 스레드 관련 종료 처리를 하면 된다.



TerminateThread()를 호출했거나, 운영체제의 오류로 인해 스레드가 비정상적으로 종료되는 경우에는 DLL_THREAD_DETACH가 발생하지 않을 수 있다.

DLL의 진입/종료 예제

목록 9.7은 DLL의 전역 변수 DLLProc에 진입/종료 프로시저를 설정하는 예이다.



목록 9.7 Ch9\DLLEntryLib.dpr

```
library DLLEntryLib;
uses
  SysUtils,
  Windows,
  Dialogs,
  Classes;
procedure DLLEntryPoint(dwReason: DWord);
begin
  case dwReason of
    DLL_PROCESS_ATTACH: ShowMessage('Attaching to process');
    DLL_PROCESS_DETACH: ShowMessage('Detaching from process');
    DLL_THREAD_ATTACH:  MessageBeep(0);
    DLL_THREAD_DETACH:  MessageBeep(0);
  end;
end;

begin
  { DLLProc 변수에 진입/종료 프로시저를 설정한다. }
  DllProc := @DLLEntryPoint;
  { DLL이 프로시저에 맵핑됨을 진입/종료 프로시저에 알린다. }
  DLLEntryPoint(DLL_PROCESS_ATTACH);
 end.
```

.dll 프로젝트 파일의 begin..end 블록에서 진입/종료 프로시저를 DLL의 DLLProc 변수에 넣었음을 알 수 있다. 진입/종료 프로시저인 DLLEntryPoint()는 DWord 인자의 값을 보고 어떤 상황인지를 판단한 다음, 적절한 처리를 한다. DLL이 적재되거나 해제되면 그에 해당하는 메시지 상자가 뜨고, DLL을 호출한 애플리케이션이 스레드를 생성/해제하면 '삐' 소리를 내도록 했다. 실제 DLL에서는 DLL 데이터의 초기화/해제 작업을 적절히 수행해야 할 것이다.

목록 9.8은 이 DLL이 어떻게 동작하는지 확인하기 위한 예제 프로젝트이다.

목록 9.8 Ch9\Dllentry\ManiFrm.pas

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ComCtrls, Gauges;

type

  { TThread를 상속하는 클래스 }
  TTestThread = class(TThread)
    procedure Execute; override;
  end;

  TMainForm = class(TForm)
    btnLoadLib: TButton;
    btnFreeLib: TButton;
    btnCreateThread: TButton;
    btnFreeThread: TButton;
    lblCount: TLabel;
    procedure btnLoadLibClick(Sender: TObject);
    procedure btnFreeLibClick(Sender: TObject);
    procedure btnCreateThreadClick(Sender: TObject);
    procedure btnFreeThreadClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
```

```
    LibHandle   : THandle;
    TestThread  : TTestThread;
    Counter     : Integer;
    GoThread    : Boolean;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TTestThread.Execute;
begin
  while MainForm.GoThread do
  begin
    MainForm.lblCount.Caption := IntToStr(MainForm.Counter);
    Inc(MainForm.Counter);
  end;
end;

procedure TMainForm.btnLoadLibClick(Sender: TObject);
{ DllEntryLib.DLL을 적재한다. }
begin
  if LibHandle = 0 then
  begin
    LibHandle := LoadLibrary('DLLENTRYLIB.DLL');
    if LibHandle = 0 then
      raise Exception.Create('DLL을 적재할 수 없습니다.');
  end
  else
    MessageDlg('라이브러리가 이미 적재되었습니다.', mtWarning, [mbok], 0);
end;

procedure TMainForm.btnFreeLibClick(Sender: TObject);
{ DLL을 해제한다. }
begin
  if not (LibHandle = 0) then
```

```
  begin
    FreeLibrary(LibHandle);
    LibHandle := 0;
  end;
end;

procedure TMainForm.btnCreateThreadClick(Sender: TObject);
{ TThread 인스턴스를 생성한다. DLL이 적재되어 있으면 '삐' 소리가 날 것이다. }
begin
  if TestThread = nil then
  begin
    GoThread   := True;
    TestThread := TTestThread.Create(False);
  end;
end;

procedure TMainForm.btnFreeThreadClick(Sender: TObject);
{ TThread를 해제하는 프로시저. DLL이 적재되어 있다면 '삐' 소리가 날 것이다. }
begin
  if not (TestThread = nil) then
  begin
    GoThread   := False;
    TestThread.Free;
    TestThread := nil;
    Counter    := 0;
  end;

end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  LibHandle  := 0;
  TestThread := nil;
end;

 end.
```

이 프로젝트의 메인 폼에는 TButton 컴포넌트가 네 개 있다. btnLoadLib 버튼을 클릭하면 DLLEntryLib.dll이 적재된다. btnFreeLib를 클릭하면 프로세스로부터 DLL을 제거한다. btnCreateThread는 TThread를 상속하는 객체를 생성해서 스레드를 하나 생성하며, btnFreeThread는 그 TThread 객체를 파괴한다. lblCount에는 스레드 수행 상태가 표시된다.

btnLoadLibClick() 이벤트 처리 프로시저는 LoadLibrary()를 호출해서 DLlEntryLib.dll을 적재한다. 그러면 DLL이 프로세스의 주소 공간에 맵핑된다. 또한, DLL의 초기화 코드가 실행된다. 초기화 코드란, DLL 프로젝트 파일의 begin..end 블록을 말한다. 그 초기화 코드에서는 다음과 같이 DLL의 진입/종료 프로시저를 설정했었다.

```
begin
  { DLLProc 변수에 진입/종료 프로시저를 설정한다. }
  DllProc := @DLLEntryPoint;
  { DLL이 프로시저에 맵핑됨을 진입/종료 프로시저에 알린다. }
  DLLEntryPoint(DLL_PROCESS_ATTACH);
end.
```

위의 초기화 코드는 프로세스당 한 번만 호출된다. 다른 프로세스가 이 DLL을 적재하면 위의 코드는 그 프로세스의 문맥하에서 일어난다. 프로세스들은 DLL 인스턴스를 공유하지 않기 때문이다.

btnFreeLibClick() 이벤트 처리 프로시저는 FreeLibrary()를 호출해서 DLL을 제거하는데, 그러면 DLL의 전역 변수 DLLProc이 가리키는 진입점 프로시저인 DLLEntryPoint()가 호출된다. 이때 전달되는 인자의 값은 DLL_PROCESS_DETACH이다.

btnCreateThreadClick() 이벤트 처리 프로시저는 TThread를 상속하는 객체를 생성해서 프로세스에 스레드를 하나 생성한다. 그러면 DLL_THREAD_ATTACH를 인자로 해서 DLLEntryPoint()가 호출된다. 그리고 btnFreeThreadClick() 이벤트 처리 프로시저는 생성된 스레드를 제거하며, 이때는 DLL_THREAD_DETACH를 인자로 해서 DLLEntry Point()가 호출된다.

DLLEntryPoint()는 넘겨 받은 인자에 따라 메시지 상자나 '삐' 소리를 내지만, 실제적인 DLL에서라면 프로세스나 스레드 관련 데이터의 초기화/해제 작업을 수행해야 할 것이다. 나중에 진입/종료 루틴을 통해서 프로세스들이 공유하는 DLL의 전역 데이터를 관리하는 기법을 제시하겠다. 이번 예제는 부록 CD의 DLLEntryTest.dpr에 해당한다.

DLL의 예외 처리

이번에는 DLL과 Win32 예외에 대해 알아보자.

16비트 델파이에서의 DLL 예외 처리

16비트 델파이, 즉 델파이 1에서는 델파이의 예외가 델파이 고유의 기능이었다. 따라서, 델파이로 만든 DLL 안에서 예외가 발생하면 그 예외를 DLL 바깥, 즉 호출한 프로세스로 넘길 수 없었다. 그래서 DLL의 모든 루틴들에 다음과 같은 형태로 예외 처리 구조를 설치해야 했다.

```
procedure SomeDLLProc;
begin
  try
      { 작업을 수행 }
  except
      on Exception do
          { 예외를 깨끗이 처리해야 하며, 재발생시키면 안된다. }
  end;
end;
```

32비트로 넘어오면서 상황이 바뀌었다. 델파이 5의 예외는 Win32의 예외에 그대로 대응되기 때문이다. 따라서, DLL에서 발생한 예외를 델파이 고유의 기능이 아니라 Win32 운영체제의 기능을 통해 처리할 수 있다.

그런데 Win32가 델파이로 만든 DLL의 예외를 인식할 수 있으려면 DLL의 uses절에 SysUtils가 있어야 한다. uses절에 SysUtils를 포함시키지 않으면 DLL 안에서의 델파이의 예외 지원이 활성화되지 않는다.



대부분의 Win32 애플리케이션은 예외를 처리할 수 있도록 만들어지지 않았으므로, 델파이 언어의 예외를 Win32 예외로 대응시킬 수 있다고 해도 DLL의 예외 때문에 애플리케이션이 다운될 가능성이 있다.
델파이나 C++ 빌더로 만든 애플리케이션이라면 그리 큰 문제가 되지 않지만, 순수한 C/C++와 Win32 API만으로 윈도우즈 애플리케이션을 만드는 사람도 많이 있으며, 그런 애플리케이션들은 예외라는 것을 알지 못한다.
따라서, 좀더 안전한 DLL을 만들기 위해서는 16비트 시절에 했던 것처럼 try..except 구조를 이용해서 DLL 안에서 예외를 확실하게 처리해 주는 것이 좋다.



델파이로 만든 것이 아닌 애플리케이션이 델파이로 만든 DLL을 사용하는 경우, 델파이 고유의 예외 클래스를 사용하지 못한다. 다만, 델파이의 예외 클래스는 Win32 시스템 예외로 처리될 수는 있는데, 이때 Win32의 예외 코드는 $0EEDFACE이다. 그리고 예외의 주소는 Win32의 EXCEPTION_RECORD 구조체 배열 ExceptionInformation의 첫 번째 항목이다. 이 배열의 두 번째 항목에는 델파이 예외 객체에 대한 참조(reference)가 담겨 있으므로 오브젝트 파스칼 이외의 언어로도 델파이의 예외 객체를 참조할 수 있다. 델파이 도움말의 EXCEPTION_RECORD 항목을 참고하기 바란다.

예외와 Safecall 지시자

COM이나 예외 처리에 관련된 DLL의 루틴은 safecall 지시자를 사용해야 한다. Safecall 지시자로 루틴을 선언하면 예외를 호출한 프로세스에 전달할 수 있다. Safecall 지시자는 델파이 예외를 HResult 리턴값으로 변환하며, 묵시적으로 StdCall 호출 규약을 지정한다. 예를 들어, 다음과 같은 함수 선언을 보자.

```
fuction Foo(i: Integer): string; Safecall;
```

컴파일러는 위의 선언을 다음과 같은 형태로 이해한다.

```
fuction Foo(i: Integer): string; HResult; StdCall;
```

그리고 컴파일러는 묵시적으로 함수 전체를 try..except 블록으로 감싸서 예외를 잡아낸다. 또한, except 블록 안에 SafecallExceptionHandler()를 호출해서 예외를 HResult값으로 변환하는 코드를 집어넣는다. 16비트 시절, 개발자가 직접 예외를 잡아내고 에러 번호를 돌려주던 것을 컴파일러가 대신 해주는 셈이다.

콜백 함수

콜백 함수(callback function)는 Win32 DLL이나 다른 DLL이 호출하는 함수이다. 한 프로세스 안에 있는 함수는 그 프로세스가 일반적이지만, 콜백 함수는 다른 프로세스, 즉 다른 애플리케이션이나 다른 DLL에게 자신의 주소를 넘겨주고, 그쪽에서 자신을 호출하도록 하는 것이다. 전화번호를 알려주고 전화를 기다리는 것과 같은 개념이며, 그래서 콜백이란 이름이 붙었다. 콜백 함수와 DLL이 어떤 관계인지 궁금한 독자도 있을텐데, 콜백 함수는 대부분 Win32 API와 관련이 있으며, Win32 API들은 모두 Win32 시스템 DLL에 들어 있기 때문에 DLL을 설명하는 이번 장에서 콜백 함수를 설명하는 것이다.

콜백 함수를 요구하는 Win32 API들 중 대표적인 것이 EnumWindows() 함수이다. 이 함수는 최상위 윈도우들에 대한 정보를 얻을 때 사용한다. 이 함수에 콜백 함수의 주소를 넘겨주면 이 함수는 현재 활성화되어 있는 윈도우의 핸들을 그 콜백 함수의 인자에 넣어서 콜백 함수를 호출하는데, 한 윈도우당 한 번 콜백 함수를 호출한다. EnumWindows()에 넘겨줄 콜백 함수는 다음과 같은 인자와 리턴값으로 이루어져야 한다.

```
function EnumWindowsProc(Hw: Hwnd; lp: lParam): Boolean; stdcall;
```

그럼, EnumWindows()와 콜백 함수를 사용하는 예제를 설명하겠다. 이 예제는 부록 CD의 CallBack.dpr에 해당하며, 목록 9.9는 이번 예제 프로젝트의 메인 폼의 소스이다.

목록 9.9 Ch9\CallBack\MainFrm.pas

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ComCtrls;

type

 { 윈도우 이름과 클래스 이름을 담을 클래스를 정의한다. 이 클래스의 인스턴스들이
   목록 상자의 Objects 배열에 추가된다. }
  TWindowInfo = class
    WindowName,         // 윈도우 이름
    WindowClass: String; // 윈도우의 클래스 이름
  end;

 TMainForm = class(TForm)
    lbWinInfo: TListBox;
    btnGetWinInfo: TButton;
    hdWinInfo: THeaderControl;
    procedure btnGetWinInfoClick(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure lbWinInfoDrawItem(Control: TWinControl; Index: Integer;
      Rect: TRect; State: TOwnerDrawState);
    procedure hdWinInfoSectionResize(HeaderControl: THeaderControl;
      Section: THeaderSection);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}
function EnumWindowsProc(Hw: HWnd; AMainForm: TMainForm): Boolean; stdcall;
```

```
{ 이 프로시저는 User32.DLL에 있는 Win32 API EnumWindows()가 최상위 윈도우의
  정보를 넘겨주는 콜백 함수이다. }
var
  WinName, CName: array[0..144] of char;
  WindowInfo: TWindowInfo;
begin
 { 리턴값이 True라는 것은 EnumWindows()에게 계속 이 콜백 함수를 호출해서
   윈도우 정보를 넘겨달라는 뜻이다. }
  Result := True;
  GetWindowText(Hw, WinName, 144); // 현재 윈도우 이름(제목)을 얻는다.
  GetClassName(Hw, CName, 144);    // 현재 윈도우의 클래스 이름을 얻는다.
 { TWindowInfo의 인스턴스를 생성하고, 윈도우 이름과 클래스 이름을 넣는다. 그런
   다음, 인스턴스를 목록 상자 lbWinInfo의 Objects 배열에 추가한다. 나중에 목록
   상자를 그릴 때 이 인스턴스의 필드들을 이용해서 윈도우 이름과 클래스 이름을
   표시할 것이다. }
  WindowInfo := TWindowInfo.Create;
  with WindowInfo do
  begin
    SetLength(WindowName, strlen(WinName));
    SetLength(WindowClass, StrLen(CName));
    WindowName := StrPas(WinName);
    WindowClass := StrPas(CName);
  end;
  MainForm.lbWinInfo.Items.AddObject('', WindowInfo); // Objects 배열에 추가
end;

procedure TMainForm.btnGetWinInfoClick(Sender: TObject);
begin
 { EnumWinodws()에 EnumWindowsProc 콜백 함수의 주소를 넘겨준다. 그러면
   EnumWindows()는 활성화된 모든 최상위 윈도우들을 열거해 준다. }
  EnumWindows(@EnumWindowsProc, 0);
end;

procedure TMainForm.FormDestroy(Sender: TObject);
var
  i: integer;
begin
  { TWindowInfo 인스턴스들을 모두 해제한다. }
  for i := 0 to lbWinInfo.Items.Count - 1 do
```

```
    TWindowInfo(lbWinInfo.Items.Objects[i]).Free
end;

procedure TMainForm.lbWinInfoDrawItem(Control: TWinControl; Index:
 Integer;  Rect: TRect; State: TOwnerDrawState);
begin
 { 우선, 텍스트들이 출력될 영역을 지운다. }
  lbWinInfo.Canvas.FillRect(Rect);
 { 그런 다음, 목록 상자의 Objects 배열에 있는 TWindowInfo 인스턴스들에 담긴
   윈도우 이름과 클래스 이름을 출력한다. 이때 컬럼 형식으로 윈도우 이름과
   클래스 이름을 배치하기 위해서 HeaderControl 컴포넌트 hdWinInfo의 컬럼
   너비에 맞게 텍스트들을 출력한다. }
  with TWindowInfo(lbWinInfo.Items.Objects[Index]) do
  begin
    DrawText(lbWinInfo.Canvas.Handle, PChar(WindowName),
      Length(WindowName), Rect,dt_Left or dt_VCenter);
      { hdWinInfo의 섹션 너비를 이용해서 다음 텍스트의 출력 영역을 조정한다. }
    Rect.Left := Rect.Left + hdWinInfo.Sections[0].Width;
    DrawText(lbWinInfo.Canvas.Handle, PChar(WindowClass),
      Length(WindowClass), Rect, dt_Left or dt_VCenter);
  end;
end;

procedure TMainForm.hdWinInfoSectionResize(HeaderControl:
   THeaderControl; Section: THeaderSection);
begin
  lbWinInfo.Invalidate; // 목록 상자를 다시 그린다.
end;

 end.
```

이 애플리케이션은 EnumWindows() 함수를 이용해서 현재 활성화되어 있는 모든 최상위 윈도우들의 윈도우 이름과 클래스 이름을 얻은 다음, 그것들을 메인 폼의 목록 상자에 추가한다. 메인 폼의 목록 상자는 오너 드로(owner-draw) 목록 상자, 즉 목록 상자의 표시를 코드에서 직접 수행하는 목록 상자이다. 윈도우 이름과 클래스 이름을 컬럼 형식으로 나타내기 위해서 오너 드로 기법을 사용했는데, 이에 대해서는 잠시 후에 설명하겠다.

콜백 함수의 사용

EnumWindowsProc()이 이번 예제의 콜백 함수이다. 이 함수는 EnumWindow() API 함수에 사용할 콜백 함수이므로 첫 번째 인자의 데이터형은 반드시 윈도우 핸들이어야 하며, 두 번째 인자는 Integer 데이터형과 길이가 같은 형이어야 한다.

EnumWindowsProc()은 Win32 API에 대한 콜백 함수이므로 반드시 StdCall 지시자를 써서 Win32의 표준 호출 규약을 만족하도록 선언해야 한다. 이 함수를 EnumWindows()에 넘겨주면 EnumWindows()는 최상위 윈도우들을 찾고, 한 윈도우에 한 번씩 그 윈도우의 핸들을 인자로 해서 EnumWindowsProc()을 호출한다. EnumWindowsProc()에서는 넘겨 받은 윈도우 핸들을 이용해서 그 윈도우의 제목(윈도우 이름)과 클래스 이름을 얻는다. 두 이름을 얻은 다음에는 TWinInfo 클래스의 인스턴스를 하나 생성해서 그 인스턴스의 필드들에 이름을 넣은 다음, 그 인스턴스를 목록 상자 lbWinInfo의 Objects 배열에 추가한다. 목록 상자에 추가된 인스턴스들은 나중에 목록 상자를 그릴 때 사용된다.

TWinInfo 클래스의 인스턴스들을 동적으로 생성했으므로 프로그램을 종료하기 전에 그 인스턴스들을 해제하는 것을 잊어서는 안된다. 인스턴스들의 해제는 폼의 OnDestroy 이벤트 처리 프로시저에서 수행한다.

그리고 Win32 API 함수 EnumWindows()에 콜백 함수 EnumWindowsProc()의 주소를 넘겨주는 작업은 btnGetWinInfoClick() 이벤트에서 수행한다.

이 애플리케이션을 실행시키고 버튼을 클릭하면 최상위 윈도우들의 이름과 클래스 이름이 목록 상자에 나타날 것이다.

컬럼 형식의 목록 상자 그리기

일반적인 목록 상자와는 달리, 이 프로젝트의 메인 폼에 있는 목록 상자(TListBox)에는 윈도우 이름과 클래스 이름이 컬럼 형식으로 나타난다. 이처럼 목록 상자의 표시 방식을 직접 바꿀 수 있는데, 그러한 기법을 오너드로(owner-draw)라고 한다. 오너 드로를 적용하려면 우선 목록 상자의 Style 속성을 lbOwnerDraw로 설정해야 한다. Style이 lbOwnerDraw로 되어 있으면 목록 상자의 항목이 화면에 표시되려고 할 때 TListBox.OnDrawItem 이벤트가 발생한다. 그 이벤트의 처리 프로시저에서 목록 상자의 표시를 개발자가 직접 정의하는 것이다.

목록 9.9의 lbWinInfoDrawItem()이 목록 상자의 항목을 표시하는 이벤트 처리 프로시저이다. 이 프로시저는 lbWinInfo.Objects 배열에 담긴 TWinInfo 클래스 인스턴스들의 정보를 목록 상자에 표시하는데, 이때 Canvas와 TextOut()을 이용해서 텍스트들의 출력 위치를 직접 지정한다. 자세한 설명은 주석을 참고하기 바란다.

콜백 함수를 호출하는 DLL 만들기

앞의 예제는 Win32 API에게 넘겨줄 콜백 함수를 사용하는 예제였다. 그럼, 이번에는 다른 애플리케이션이나 DLL로부터 콜백 함수의 주소를 넘겨 받고, 그 콜백 함수를 호출해 주는 DLL을 작성해 보자. 목록 9.10은 이번 예제 DLL의 소스 코드이다.

목록 9.10 Ch9\DllCallback\StrSrchLib.dpr

```
library StrSrchLib;

uses
  Wintypes,
  WinProcs,
  SysUtils,
  Dialogs;

type
 { 콜백 함수의 함수형을 정의 }
 TFoundStrProc = procedure(StrPos: PChar); stdcall;

procedure SearchStr(ASrcStr, ASearchStr: PChar; AProc: TFarProc); stdcall;
{ 이 함수는 문자열 ASrcStr에서 문자열 ASearchStr을 찾는다. ASearchStr을 찾으면
  AProc으로 지정된 콜백 함수에 결과를 알려준다. AProc이 nil이면 그냥 종료한다. }
var
  FindStr: PChar;
begin
  FindStr := ASrcStr;
  FindStr := StrPos(FindStr, ASearchStr);
  while FindStr <> nil do
  begin
```

```
    if AProc <> nil then
      TFoundStrProc(AProc)(FindStr);
    FindStr := FindStr + 1;
    FindStr := StrPos(FindStr, ASearchStr);
  end;
end;

exports
  SearchStr;
begin

end.
```

콜백 함수를 요구하는 DLL은 자신이 호출할 콜백 함수의 형을 정의해야 한다. TFoundStrProc은 이 DLL이 호출할 콜백 함수(실제로는 프로시저)의 인자를 정의한 프로시저형이다.

콜백 함수를 실제로 호출하는 프로시저는 SearchStr()이다. 이 프로시저에 대해서는 주석만으로도 충분히 설명이 될 것이다.

이 DLL을 사용하는 예제는 부록 CD의 \DLLCallBack 디렉토리에 있는 CallBackDemo.dpr이다. 다음의 목록 9.11은 CallBackDemo.dpr의 메인 폼의 소스이다.

목록 9.11 Ch9\DllCallback\MainFrm.pas

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TMainForm = class(TForm)
    btnCallDLLFunc: TButton;
    edtSearchStr: TEdit;
    lblSrchWrd: TLabel;
```

```
    memStr: TMemo;
    procedure btnCallDLLFuncClick(Sender: TObject);
  end;

var
  MainForm: TMainForm;
  Count: Integer;

implementation

{$R *.DFM}

{ 사용할 DLL의 프로시저를 정의한다. }
function SearchStr(ASrcStr, ASearchStr: PChar; AProc: TFarProc): Integer;
 StdCall external  'STRSRCHLIB.DLL';

{ 콜백 프로시저를 정의한다. 콜백 함수에는 반드시 StrCall 지시자를 써줘야 한다. }
procedure StrPosProc(AStrPsn: PChar); StdCall;
begin
  inc(Count); // Count 변수를 증가시킨다.
end;

procedure TMainForm.btnCallDLLFuncClick(Sender: TObject);
var
  S: String;
  S2: String;
begin
  Count := 0; // Count를 0으로 초기화
  { 검색할 텍스트를 넣을 S의 메모리를 할당한다. }
  SetLength(S, memStr.GetTextLen);
  { 검색할 텍스트를 S에 복사한다. }
  memStr.GetTextBuf(PChar(S), memStr.GetTextLen);
 { 검색 단어가 있는 에디트 컨트롤의 Text를 string 변수에 넣는다. Text 속성은
DLL 함수에 직접 넘길 수 없다. }
  S2 := edtSearchStr.Text;
  { DLL 함수를 호출한다. }
  SearchStr(PChar(S), PChar(S2), @StrPosProc);
 { 검색할 텍스트에 검색 단어가 몇 번 나오는지를 표시한다. 검색 단어의 출현
   횟수는 Count 변수에 담겨 있다. DLL이 콜백 함수(StrPosProc)를 호출하면 콜백
```

```
    함수는 Count를 증가시키므로 Count는 검색 단어의 출현 횟수를 의미하게 된다. }
  ShowMessage(Format('%s %d %s', [edtSearchStr.Text, Count,
  '번 발생.']));
end;

end.
```

이 애플리케이션의 메인 폼에는 버튼 하나와 메모 컨트롤, 그리고 에디트 컨트롤이 있다. 메모 컨트롤에는 검색할 텍스트가 담겨 있으며, 에디트 컨트롤에는 검색 단어가 담겨 있다. 두 텍스트를 DLL 함수 SerchStr()의 인자로 사용하는데, 이때 문자열의 메모리 할당이나 변환이 중요하다. 소스의 주석을 참고하기 바란다.

StrPosProc()이 콜백 함수인데, 이 함수는 전역 변수 Count를 증가시킨다. 따라서, Count는 검색할 텍스트 안에서 검색 단어가 나오는 횟수가 된다.

프로세스들의 DLL 데이터 공유

16비트 윈도우즈는 Win32와는 다른 방식으로 DLL의 메모리를 관리했었다. 16비트 윈도우즈에서는 DLL의 전역 메모리를 여러 애플리케이션들이 공유할 수 있었다. 다시 말해서, 16비트 DLL에서 선언한 전역 변수는 그 DLL을 사용하는 모든 애플리케이션들이 사용할 수 있었으며, 한 애플리케이션이 그 변수의 값을 변경하면 변경된 값은 다른 애플리케이션들에도 적용이 되었던 것이다.

그러한 점이 편리할 수도 있고, 또 위험할 수도 있다. 그 전역 변수를 애플리케이션들 사이의 일종의 정보 교환 수단으로 사용한다면 편리하겠지만, 다른 애플리케이션들이 언제 어떻게 전역 변수를 변경하는지 알 수가 없으므로 위험하기도 한 것이다.

Win32의 경우, DLL의 전역 데이터를 더 이상 공유할 수 없다. 각각의 애플리케이션 프로세스는 DLL을 자신의 고유한 주소 공간에 맵핑시키기 때문에 DLL의 전역 변수 역시 그 프로세스 주소 공간에서만 의미가 있는 것이다. 열 개의 애플리케이션이 DLL을 적재하면 전역 변수 역시 각각 10개가 생기는 셈이다. 따라서, 한 애플리케이션이 전역 변수를 변경하는 것은 그 애플리케이션에게만 의미가 있을 뿐, 다른 애플리케이션의 프로세스에게는 아무런 의미가 없다.

만일 DLL을 통한 데이터 공유를 사용하는 16비트 애플리케이션을 32비트로 포팅한다고 하면 뭔가 다른 방법을 찾아야 한다. 전역 변수를 읽고 쓰는 것만으로 자동적으로 데이터를 공유할 수는 없게 된 것이다. 프로세스들 사이의 데이터 공유 기법은 여러 가지가 있지만, 가장 일반적인 것은 메모리 맵 파일(memory-mapped file)이다. 메모리 맵 파일은 12장, "파일 다루기"에서 자세히 이야기하니 참고하기 바란다. 이번 예제는 메모리 맵 파일을 통해서 프로세스들 사이에서 데이터를 공유한다.

데이터를 공유하기 위한 DLL 작성

목록 9.12는 자신을 사용하는 애플리케이션들이 자신의 전역 데이터를 공유할 수 있도록 하는 DLL의 예제이다. 공유하고자 하는 데이터는 GlobalData라는 변수에 저장된다.



Ch9\SharedLib\SharedLib.dpr

```
library ShareLib;

uses
  ShareMem,
  Windows,
  SysUtils,
  Classes;
const

  cMMFileName: PChar = 'SharedMapData';

{$I DLLDATA.INC}

var
  GlobalData : PGlobalDLLData;
  MapHandle  : THandle;

{ DLL이 공개하는 프로시저 GetDLLData를 정의한다. }
procedure GetDLLData(var AGlobalData: PGlobalDLLData); StdCall;
begin
  { AGlobalData에 GlobalData가 가리키는 메모리 주소를 설정한다. }
  AGlobalData := GlobalData;
```

```
end;

procedure OpenSharedData;
var
   Size: Integer;
begin
  { 맵핑할 데이터의 크기를 구한다. }
  Size := SizeOf(TGlobalDLLData);

 { 메모리 맵 파일 객체를 생성한다. 첫 번째 인자는 $FFFFFFFF, 즉 DWord(-1)이다.
   이 값은 시스템의 페이징 파일로부터 공간을 할당하라는 의미를 갖는다. 마지막
   인자에는 이 메모리 맵 파일의 이름을 넣는다. }

  MapHandle := CreateFileMapping(DWord(-1), nil, PAGE_READWRITE, 0, Size,
     cMMFileName);

  if MapHandle = 0 then
    RaiseLastWin32Error;
 { 호출한 프로세스의 주소 공간에 전역 데이터를 맵핑하고, 맵핑 영역의 시작
   주소를 가리키는 포인터를 얻는다. }
  GlobalData := MapViewOfFile(MapHandle, FILE_MAP_ALL_ACCESS, 0, 0, Size);
  { 전역 데이터를 초기화한다. }
  GlobalData^.S := 'ShareLib';
  GlobalData^.I := 1;
  if GlobalData = nil then
  begin
    CloseHandle(MapHandle);
    RaiseLastWin32Error;
  end;
end;

procedure CloseSharedData;
{ 메모리 맵 파일의 맵핑을 해제하고, 메모리 맵 파일 핸들을 해제한다. }
begin
  UnmapViewOfFile(GlobalData);
  CloseHandle(MapHandle);
end;

procedure DLLEntryPoint(dwReason: DWord);
```

```
begin
  case dwReason of
    DLL_PROCESS_ATTACH: OpenSharedData;
    DLL_PROCESS_DETACH: CloseSharedData;
  end;
end;

exports
  GetDLLData;

begin
  { DLLProc 변수에 진입/종료 프로시저를 설정한다. }
  DllProc := @DLLEntryPoint;
 { 진입/종료 프로시저에 DLL_PROCESS_ATTACH를 넣고 호출해서 DLL이 프로시저에
   부착됨을 알린다. }
  DLLEntryPoint(DLL_PROCESS_ATTACH);
 end.
```

GlobalData의 형은 PGlobalDLLData로, PGlobalDLLData는 포함 파일 DllData.inc에 정의되어 있다. DllData.inc는 다음과 같은 정의를 담고 있다. 포함 파일(include file, C의 헤더 파일과 같은 의미)은 지시자 $I를 통해 프로젝트에 연결된다.

```
type

  PGlobalDLLData = ^TGlobalDLLData;
  TGlobalDLLData = record
    S: String[50];
    I: Integer;
  end;
```

이 DLL 역시 진입/종료 프로시저를 가지고 있다. DLLEntryPoint()가 이 DLL의 진입/종료 프로시저이다. 프로세스가 DLL을 적재하면 DLLEntryPoint()는 OpenSharedData() 메소드를 호출하며, 프로세스가 DLL을 제거하면 DLLEntryPoint()는 CloseSharedData()를 호출한다.

메모리 맵 파일의 사용법에 대해서는 12장에 자세히 나오므로 여기서 일일이 설명하지는 않겠다. 이 DLL이 메모리 맵 파일을 어떻게 사용하는지에 대해서만 이야기하겠다.

메모리 맵 파일이란, 메모리의 일정 영역을 파일처럼 쓸 수 있도록 만드는 것이라고 생각하면 된다. 디스크의 파일을 사용할 때에는 파일 이름을 통해서 파일 핸들을 얻고, 그 파일 핸들을 통해서 데이터를 읽거나 쓰고, 작업이 끝나면 파일 핸들을 닫아서 파일을 닫는다. 메모리 맵 파일은 메모리를 다룬다는 점에서는 포인터와 비슷하지만, 디스크의 파일을 다루는 것과 거의 비슷한 방식으로 메모리를 다룰 수 있으며, 핸들을 얻을 수만 있으면 여러 프로세스가 메모리 맵 파일을 읽거나 쓸 수 있으므로 결과적으로 프로세스들 사이에서의 메모리 공유가 가능해진다.

메모리 맵 파일은 기본적으로 디스크에 있는 파일을 메모리에 맵핑(mapping), 즉 사상시켜서 만든다. 맵핑이란, 어떤 것을 다른 어떤 것에 대응시킨다는 뜻이다. 디스크에 있는 파일이 메모리의 일정한 영역에 대응되도록 만드는 것이 바로 메모리 맵 파일이다. 이번 장의 앞부분에서 Win32에서는 프로세스마다 DLL의 인스턴스를 갖지만, 그렇다고 해서 DLL의 복사본이 프로세스 개수만큼 생기는 것은 아니라는 점이 언급되었다. 파일 상태의 DLL은 Win32 운영체제의 전역 힙 메모리에 한 번만 올라오고, 각각의 프로세스는 DLL의 복사본이 있는 메모리 영역을 자신의 주소 공간에 대응시키는데, 이때에도 메모리 맵 파일이 사용된다.

메모리 맵 파일을 데이터 공유에 사용하는 예를 들겠다. 두 애플리케이션이 디스크의 파일을 이용해서 데이터를 공유한다고 하자. 한 애플리케이션이 자신의 처리 결과를 디스크의 파일에 저장하면 다른 애플리케이션은 그 디스크의 파일을 읽어서 그 데이터로 작업을 할 것이다. 그러나 디스크는 느린 매체이므로 그 파일을 메모리의 일정 영역에 맵핑시켜서 메모리 맵 파일을 만들고, 그 영역에서 데이터를 쓰거나 읽을 수 있도록 만들면 속도가 훨씬 빨라질 것이다. App1이라는 애플리케이션이 메모리 맵 파일을 만들어서 디스크의 파일에 맵핑시키고, App2라는 애플리케이션도 그 메모리 맵 파일을 동일한 파일에 맵핑시키기만 하면 두 애플리케이션은 메모리 맵 파일을 통해서 자유롭게 데이터를 공유할 수 있다. 메모리 맵 파일의 읽기/쓰기 설정에는 여러 가지가 있지만, 데이터 공유를 위해서라면 한 쪽이 변경한 내용이 즉시 메모리 맵 파일에 적용되도록 해야 할 것이다.

데이터를 공유하는 것이 주목적이라면 실제로 디스크에 있는 파일로부터 메모리 맵 파일을 만들 필요는 없다. 그래서 Win32는 페이징 파일로부터 메모리 맵 파일을 만들 수 있도록 한다. 굳이 임시 파일을 만들고 지울 필요가 없는 것이다. 페이징 파일

은 Win32가 직접 관리하므로 메모리 맵 파일의 사용이 끝나면 페이징 파일에 할당된 메모리도 해제된다.

앞에서 메모리 맵 파일을 이용해서 두 애플리케이션이 데이터를 공유하는 예를 들었는데, 동일한 기법을 애플리케이션들과 DLL 사이에도 적용할 수 있다. 다음과 같은 상황을 생각하면 된다. 애플리케이션이 DLL을 적재하면 DLL은 메모리 맵 파일을 만든다. 다른 애플리케이션이 그 DLL을 적재하면 역시 메모리 맵 파일을 만드는데, 이때 사용하는 파일 이름이 같으므로 결국은 같은 메모리 맵 파일을 얻게 된다. 프로세스마다 다른 DLL 인스턴스가 생기는 것이지만, 두 인스턴스가 가지고 있는 메모리 맵 파일은 동일하다. 프로세스들은 그 메모리 맵 파일을 통해서 DLL의 데이터를 공유하게 된다.

목록 9.12에서 메모리 맵 파일을 만드는 작업은 OpenSharedData()가 담당한다. 우선, CreateFileMapping() 함수를 이용해서 파일 맵핑 객체를 생성한 다음, 그것을 MapViewOfFile() 함수에 넘겨준다. MapViewOfFile() 함수는 그 파일의 뷰를 호출한 프로세스의 주소 공간에 맵핑한다. MapViewOfFile()의 리턴값은 맵핑된 주소 공간의 시작 주소이다. 이 주소는 호출한 프로세스의 주소 공간에 속한 것임을 기억하기 바란다. 서로 다른 프로세스가 이 DLL을 호출한 경우, 맵핑된 주소 공간의 시작 주소는 다르겠지만, 그 주소가 가리키고 있는 데이터는 동일하다. 맵핑의 의미를 잘 이해하기 바란다.



CreateFileMapping()의 첫 번째 인자는 메모리 맵 파일이 맵핑될 디스크 파일의 핸들이다. 그러나 데이터 공유를 위해서 Win32의 페이징 파일에 메모리 맵 파일을 맵핑하는 경우에는 실제 파일의 핸들 대신 $FFFFFFFF(이는 Dword(-1)과 같다)를 넣는다. 또한, 마지막 인자에는 파일 맵핑 객체의 이름을 넣어야 한다. 이 이름은 운영체제가 파일 맵핑을 식별하기 위해 사용한다. 하나 이상의 프로세스가 동일한 이름을 이용해서 메모리 맵 파일을 생성하면 그 메모리 맵 파일들은 모두 동일한 시스템 메모리 영역을 가리키게 된다.

MapViewOfFile()을 호출한 다음에는 GlobalData 포인터에 메모리 맵 파일의 주소를 넣는다. 애플리케이션들은 DLL이 제공하는 GetDLLData() 함수를 통해서 GlobalData

가 가리키고 있는 주소를 얻으므로 애플리케이션 안에서 DLL의 GlobalData가 가리키는 곳의 값을 변경하거나 읽을 수 있다.

CloseSharedData() 프로시저는 호출한 프로세스로부터 파일 뷰의 맵핑을 해제하고, 파일 맵핑 객체를 해제하는 역할을 한다. DLL의 인스턴스 자체는 공유되지 않으므로 한 프로세스에서 파일 맵핑 객체를 해제한다고 해서 메모리 맵 파일이 실제로 위치하고 있는 메모리 영역 자체가 해제되는 것은 아니다.

메모리 맵 파일 DLL을 통한 데이터 공유 예제

그럼, 실제로 이 DLL을 사용해서 데이터를 공유하는 예를 살펴보자. 두 개의 애플리케이션이 DLL의 데이터를 공유하는데, 첫 번째 애플리케이션은 App1.dpr이고, 이 애플리케이션은 DLL의 데이터를 수정한다. 두 번째 애플리케이션은 App2.dpr이며, DLL의 데이터를 레이블 컨트롤에 표시하는데, 타이머를 통해서 계속 표시하므로 App1이 DLL의 데이터를 변경하면 그것이 App2에도 나타난다.

목록 9.13은 App1.dpr 프로젝트의 소스 코드이다.



Ch9\SharedLib\MainFrmA1.pas

```
unit MainFrmA1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls, Mask;

{$I DLLDATA.INC}

type

  TMainForm = class(TForm)
    edtGlobDataStr: TEdit;
    btnGetDllData: TButton;
    meGlobDataInt: TMaskEdit;
    procedure btnGetDllDataClick(Sender: TObject);
    procedure edtGlobDataStrChange(Sender: TObject);
```

```
    procedure meGlobDataIntChange(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  public
    GlobalData: PGlobalDLLData;
  end;

var
  MainForm: TMainForm;

{ 사용할 DLL의 프로시저를 정의한다. }
procedure GetDLLData(var AGlobalData: PGlobalDLLData); StdCall External
 'SHARELIB.DLL';

implementation

{$R *.DFM}

procedure TMainForm.btnGetDllDataClick(Sender: TObject);
begin
  { DLL의 데이터를 가리키는 포인터를 얻는다. }
  GetDLLData(GlobalData);
 { GlobalData의 필드들을 폼에 표시한다. }
  edtGlobDataStr.Text := GlobalData^.S;
  meGlobDataInt.Text  := IntToStr(GlobalData^.I);
end;

procedure TMainForm.edtGlobDataStrChange(Sender: TObject);
begin
  { DLL의 데이터를 변경한다. }
  GlobalData^.S := edtGlobDataStr.Text;
end;

procedure TMainForm.meGlobDataIntChange(Sender: TObject);
begin
  { DLL의 데이터를 변경한다. }
  if meGlobDataInt.Text = EmptyStr then
    meGlobDataInt.Text := '0';
  GlobalData^.I := StrToInt(meGlobDataInt.Text);
end;
```

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  btnGetDllDataClick(nil);
end;

 end.
```

이 애플리케이션 역시 TGlobalDLLData와 그에 대한 포인터가 정의되어 있는 포함 파일 DLLData.inc를 사용한다. btnGetDllDataClick() 이벤트 처리 프로시저는 GetDLLData() 함수를 호출해서 DLL의 데이터를 가리키는 포인터를 얻는다. 그런 다음, 그 포인터의 값들을 폼의 컨트롤들에 표시한다. 에디트 컨트롤의 값이 변해서 OnChange 이벤트가 발생하면 GlobalData에 변경된 값을 넣는다. GlobalData는 DLL의 데이터를 가리키고 있으므로 GlobalData의 값을 변경하면 DLL의 데이터, 즉 DLL의 메모리 맵 파일의 내용이 변한다.

목록 9.14는 두 번째 애플리케이션인 App2.dpr의 소스 코드이다.

목록 9.14 Ch9\SharedLib\MainFrmA2.pas

```
unit MainFrmA2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, StdCtrls;

{$I DLLDATA.INC}

type

  TMainForm = class(TForm)
    lblGlobDataStr: TLabel;
    tmTimer: TTimer;
    lblGlobDataInt: TLabel;
    procedure tmTimerTimer(Sender: TObject);
```

```
  public
    GlobalData: PGlobalDLLData;
  end;

{ 사용할 DLL의 프로시저를 정의한다.}
procedure GetDLLData(var AGlobalData: PGlobalDLLData); StdCall External
 'SHARELIB.DLL';

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.tmTimerTimer(Sender: TObject);
begin
  GetDllData(GlobalData);  // DLL의 데이터에 대한 포인터를 얻는다.
 { GlobalData의 필드들을 폼에 표시한다. }
  lblGlobDataStr.Caption := GlobalData^.S;
  lblGlobDataInt.Caption := IntToStr(GlobalData^.I);
end;

 end.
```

이 폼은 두 개의 TLabel 컴포넌트가 있는데, 두 레이블은 타이머의 OnTimer 이벤트를 통해서 일정 간격마다 갱신된다. App1에서 사용자가 DLL의 데이터를 변경하면 그것이 App2에도 나타난다.

이 두 예제 프로그램 역시 부록 CD에 들어 있다.

DLL에서 객체를 뽑아내려면

DLL에는 함수뿐만 아니라 객체도 담을 수 있다. 즉, 다른 애플리케이션에서 그 DLL의 객체를 뽑아내서 그 객체의 메소드를 사용할 수 있는 것이다. 그러나 DLL 안에 객체를 집어넣거나 그것을 사용하려면 몇 가지 필수 조건이나 제약이 따른다. 이번에 설명할 예제는 매우 특수한 조건에서만 유용한 것이다. 일반적으로, 객체를 은폐화시키는 것은 델파이의 패키지나 인터페이스만으로도 충분하다.

다음은 DLL에서 객체를 뽑아낼 때 적용되는 일반적인 조건이나 제약 사항이다.

1. 호출하는 애플리케이션은 그 객체의 가상(virtual) 메소드만을 사용할 수 있다.
2. 그 객체의 인스턴스는 DLL 안에서만 생성될 수 있다.
3. DLL뿐만 아니라 애플리케이션도 그 객체를 정의해야 하며, 메소드의 순서도 동일하게 정의해야 한다.
4. DLL에서 뽑아낸 객체를 상속할 수는 없다.

이 외에도 다른 제약들이 있지만, 일단은 이 정도로 충분할 것이다.

그럼, DLL이 제공하는 객체를 사용하는 간단한 예제를 살펴보자. 그 객체에는 문자열의 대·소문자를 변환하는 메소드가 하나 들어 있다. 목록 9.15는 DLL이 제공할 객체의 정의이다.

목록 9.15 Ch9\ExpObject\StrConvert.Inc(DLL이 제공하는 객체)

```
type

  TConvertType = (ctUpper, ctLower);

  TStringConvert = class(TObject)
{$IFDEF STRINGCONVERTLIB}
  private
    FPrepend: String;
    FAppend : String;
{$ENDIF}
  public
    function ConvertString(AConvertType: TConvertType; AString: String):
```

```
          String; virtual; stdcall; {$IFNDEF STRINGCONVERTLIB} abstract;
{$ENDIF}
{$IFDEF STRINGCONVERTLIB}
    constructor Create(APrepend, AAppend: String);
    destructor Destroy; override;
{$ENDIF}
  end;

{ 조건부 컴파일 지시자들이 많이 나왔는데, 이 클래스를 사용하는 애플리케이션에는
  STRINGCONVERTLIB 지시자가 정의되어 있지 않다. 애플리케이션에서는 결국 다음과
  같은 정의가 된다. }

  TStringConvert = class(TObject)
  public
    function ConvertString(AConvertType: TConvertType; AString: String):
String;
      virtual; stdcall; abstract;
  end;
 }
```

목록 9.15는 StrConvert.inc라는 포함 파일이다. 포함 파일로 객체를 정의한 이유는 위에 나온 3번 조건, 즉 DLL과 호출하는 애플리케이션이 이 객체를 모두 정의하고 있어야 한다는 조건을 만족시키기 위한 것이다. DLL이나 호출하는 애플리케이션이 이 포함 파일만 포함시키면 그 조건은 간단하게 만족된다. 동일한 정의를 두 번 입력하면 지겹기도 하거니와 오타가 날 수도 있기 때문에 이렇게 포함 파일을 이용하는 것이다.

그럼, ConvertString() 메소드의 정의를 살펴보자.

```
function ConvertString(AConvertType: TConvertType; AString: String):
  String;  virtual; stdcall;
```

ConvertString() 메소드를 virtual로 선언한 이유는, 이 객체를 상속하는 객체가 이 메소드를 재정의(Override)하라는 뜻이 결코 아니다. virtual로 한 이유는 가상 메소드 테이블(VMT)에 ConvertString() 메소드에 대한 항목이 포함되어야 한다는 뜻이다. VMT에 대해서는 13장에서 자세히 설명하겠다. 일단은 VMT라는 것을 객체의 가상

메소드들을 가리키는 포인터들이 모여 있는 테이블이라고만 생각하기 바란다. 호출하는 애플리케이션이 객체의 메소드를 호출할 수 있으려면 VMT의 도움을 받아야 한다. 메소드를 선언할 때 virtual 지시자를 써주지 않으면 VMT에 포인터가 등록되지 않으므로 호출하는 애플리케이션은 그 메소드에 접근할 수가 없다. DLL이 제공하는 객체를 사용하는 애플리케이션이 얻는 것은 사실 메소드에 대한 포인터이다. 포인터는 메소드의 시작 주소를 가리키며, 그 메소드의 인자나 리턴값은 객체에 정의되어 있으므로 델파이는 그에 기반해서 적절히 인자를 넘겨주고 메소드를 호출한다.

그리고 STRINGCONVERTLIB 지시자를 통한 조건부 컴파일을 설명하겠다. 애플리케이션에서는 객체의 모든 정의를 알 필요가 없다. 애플리케이션이 사용할 부분만 알면 된다. 또한 애플리케이션을 컴파일할 때 컴파일 에러가 나지 않게 하려면 객체의 메소드를 abstract 지시자로 선언해서 추상 메소드로 만들어야 한다. 애플리케이션이 객체의 메소드를 실제로 구현하고 있는 것은 아니기 때문이다. 조건부 컴파일 지시자들은 그러한 점을 자동적으로 조정하기 위한 것이다. 애플리케이션을 컴파일할 때에는 STRINGCONVERTLIB가 정의되어 있지 않으므로 객체의 정의 중 일부분만 컴파일된다.

목록 9.16 Ch9\ExpObject\StringConvertImp.pas

```
unit StringConvertImp;
{$DEFINE STRINGCONVERTLIB}

interface
uses SysUtils;
{$I StrConvert.inc}

function InitStrConvert(APrepend, AAppend: String): TStringConvert;
stdcall;

implementation

constructor TStringConvert.Create(APrepend, AAppend: String);
begin
  inherited Create;
  FPrepend := APrepend;
  FAppend  := AAppend;
end;
```

```
destructor TStringConvert.Destroy;
begin
  inherited Destroy;
end;

function TStringConvert.ConvertString(AConvertType: TConvertType; AString:
 String): String;
begin
  case AConvertType of
    ctUpper: Result := Format('%s%s%s', [FPrepend, UpperCase(AString),
 FAppend]);
    ctLower: Result := Format('%s%s%s', [FPrepend, LowerCase(AString),
 FAppend]);
  end;
end;

function InitStrConvert(APrepend, AAppend: String): TStringConvert;
begin
  Result := TStringConvert.Create(APrepend, AAppend);
end;

 end.
```

앞에서도 이야기했듯이 DLL이 제공하는 객체는 반드시 DLL 안에서 생성되어야 한다. 객체의 생성은 DLL이 제공하는 함수 InitStrConvert()에서 수행한다. InitStrConvert()는 두 개의 인자를 받아서 그 인자들로 객체의 생성자를 호출해서 객체를 생성한다. 애플리케이션이 생성자를 직접 호출할 수 없으므로 이렇게 중간 단계의 함수를 사용했다.

또한, 이 유닛에는 STRINGCONVERTLIB 지시자가 정의되어 있음을 주목하기 바란다. 나머지 부분은 굳이 설명하지 않아도 될 것이다. 목록 9.17은 DLL의 프로젝트 파일이다.

```
library StringConvertLib;
uses
  ShareMem,
  SysUtils,
  Classes,
  StringConvertImp in 'StringConvertImp.pas';

exports
  InitStrConvert;
end.
```

이 DLL의 프로젝트 파일에 대해서는 특별히 설명할 만한 것이 없다. 그래도 다시 한 번 ShareMem의 중요성에 대해 강조해야 할 것 같다. 앞에서도 이야기했지만, DLL과 애플리케이션이 문자열이나 동적 배열을 주고 받을 때에는 DLL과 애플리케이션의 프로젝트 파일 모두 uses절의 제일 처음에 ShareMem을 넣어야 한다.

목록 9.17은 DLL이 제공하는 객체를 사용하는 예제 애플리케이션의 메인 폼이다. 에디트 컨트롤에 텍스트를 입력하고, 두 버튼 중 하나를 클릭하면 버튼에 따라 텍스트가 모두 대문자 또는 소문자로 변하는 간단한 예제로, 부록 CD의 StrConvertTest.dpr에 해당한다.

목록 9.17 DLL이 제공하는 객체를 사용하는 예제 프로젝트

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

{$I strconvert.inc}

type


  TMainForm = class(TForm)
```

```
    btnUpper: TButton;
    edtConvertStr: TEdit;
    btnLower: TButton;
    procedure btnUpperClick(Sender: TObject);
    procedure btnLowerClick(Sender: TObject);
  private
  public
  end;

var
  MainForm: TMainForm;

function InitStrConvert(APrepend, AAppend: String): TStringConvert;
 stdcall; external 'STRINGCONVERTLIB.DLL';

implementation

{$R *.DFM}

procedure TMainForm.btnUpperClick(Sender: TObject);
var
  ConvStr: String;
  FStrConvert: TStringConvert;
begin
  FStrConvert := InitStrConvert('대문자 ', ' 끝');
  try
      ConvStr := edtConvertStr.Text;
      if ConvStr <> EmptyStr then
        edtConvertStr.Text := FStrConvert.ConvertString(ctUpper, ConvStr);
  finally
    FStrConvert.Free;
  end;
end;

procedure TMainForm.btnLowerClick(Sender: TObject);
var
  ConvStr: String;
  FStrConvert: TStringConvert;
begin
```

```
    FStrConvert := InitStrConvert('소문자 ', ' 끝');
    try
        ConvStr := edtConvertStr.Text;
        if ConvStr <> EmptyStr then
          edtConvertStr.Text := FStrConvert.ConvertString(ctLower, ConvStr);
    finally
      FStrConvert.Free;
    end;
  end;

  end.
```

DLL로 구현하는 로컬라이즈

애플리케이션을 로컬라이제이션(각 나라 언어 번역, 우리나라의 경우 한글화)하는 데 있어 가장 힘든 일 중의 하나는 유저 인터페이스에서 나타나는 문자열들을 번역하는 것이다. 모든 코드를 변경하지 않고서도 문자열이 번역되려면 반드시 하나의 단독 모듈로 분리되어야만 한다. 델파이는 자동적으로 메뉴, 대화상자, 비트맵을 포함하는 .DFM 파일을 만든다. 이것들뿐만 아니라, 에러 메시지, 문자열 같은 것도 분리시킬 수 있다. 스트링 리소스들은 .DFM 파일에 포함되어 있기는 하지만, .RC 파일로 분리시킬 수 있다.

리소스를 독립시키는 것은 번역 과정을 단순화시켜 준다. 리소스 분리 과정이 끝난 후에는 리소스 모듈을 생성하는 것이다. 리소스 모듈은 .DLL 파일이고, 이 파일 안에는 프로그램에 필요한 모든 리소스를 포함하고 있다. 리소스 DLL은 리소스 모듈을 단지 바꿔줌으로써 많은 로컬라이즈화된 제품을 만들 수 있도록 도와준다. 프로그램의 이러한 로컬라이즈화된 리소스 모듈을 만들기 위해 델파이에서는 ITE를 제공한다.

통합 번역 환경(ITE, Integrated Translation Environment, DCLITE50.BPL)은 소프트웨어의 로컬리제이션(우리나라의 경우, 한글화)이나 다른 언어를 가지고서 동시에 애플리케이션을 개발할 경우에 필요한 툴이다. 델파이 IDE에 통합되어 있기 때문에 ITE는 다중의 언어를 가진 애플리케이션을 하나의 델파이 프로젝트의 일부분으로 관리할 수 있다.

ITE는 개발 내에 각각의 로컬화된 애플리케이션의 버전별 분리된 리소스 DLL을 생성하고 유지시켜 준다. 번역된 스트링 리소스를 보여주고 편집할 수 있게끔 하나의 데이터베이스 그리드를 제공해 준다. 알맞는 언어로 번역한 다음, 하나의 메뉴 커맨드로 프로젝트 리소스 DLL들을 모두 업데이트할 수 있다. 또한, 다른 프로젝트 내에서도 재사용할 수 있게끔 중앙 저장소를 만들 수 있다.

ITE는 3가지의 툴을 구성하고 있다.

1. Translation Manager는 번역된 스트링과 다른 리소스들을 보여주고, 수정할 수 있는 그리드를 보여준다.
2. Translation Repository는 다른 개발자들에 의해 프로젝트를 공유할 수 있게끔 번역 리소스들에 대한 중앙 데이터베이스를 제공한다.
3. Resource DLL 마법사는 선택한 언어에 대한 리소스 DLL들을 생성하고 관리해 줄 수 있다.

리소스 모듈을 만들 때 한 가지 주의할 사항은 리소스 모듈의 디렉토리는 반드시 로컬화된 이름을 가진 디렉토리명을 가져야만 한다. 가령, 한국의 경우 KOR이라는 디렉토리명을 가진다. 이 이름이 변경될 경우, 델파이는 인식하지 못함을 주의하기 바란다.

실행 파일, DLL, 그리고 패키지들은 모든 필요한 리소스들을 포함하고 있다. 하지만, 로컬라이즈 버전으로 변경하고자 할 경우 반드시 로컬화시킨 실행 파일, DLL, BPL을 함께 배포해야만 한다.

애플리케이션이 시작할 때 로컬 시스템의 로컬화를 먼저 체크하고, 만약에 리소스 모듈이 시스템 로컬의 언어와 나라에 부합된다면 실행 파일, DLL, 패키지에 있는 리소스를 사용하는 대신, 리소스 모듈에 있는 리소스를 사용하게 될 것이다.

예제는 Demos\RichEdit\의 예제를 약간만 수정해서(한국어를 사용 가능하게끔만 수정) 만든 것이다. 여기서 korean을 누르면 한글로 메뉴와 버튼이 바뀌고, english를 누르면 영어로 바뀌는 그런 폼을 만들고자 한다. 지금까지 얘기한 것이 바로 이렇게 다국적 애플리케이션을 제작하기 위해 리소스 DLL 마법사를 사용한다고 얘기한 것이다.

제일 먼저 메인 소스 코드의 변경 사항은 다음과 같다(변경되는 부분만 요약해 놓았다).

목록 9.18 Ch9\Localize\REMain.pas

```
unit REMain;

interface

...중간 생략...

var
  MainForm: TMainForm;

implementation

uses REAbout, RichEdit, ShellAPI, ReInit;

const
  RulerAdj = 4/3;
  GutterWid = 6;

  ENGLISH = (SUBLANG_ENGLISH_US shl 10) or LANG_ENGLISH;
  KOREAN  = (SUBLANG_KOREAN shl 10) or LANG_KOREAN;     // 한국어 상수 정의

{$R *.DFM}

...중간 생략...

{ Event Handlers }

procedure TMainForm.FormCreate(Sender: TObject);
var
  ScreenLogPixels: Integer;
  DC: HDC;
begin
  Application.OnHint := ShowHint;
  OpenDialog.InitialDir := ExtractFilePath(ParamStr(0));
  SaveDialog.InitialDir := OpenDialog.InitialDir;
  SetFileName('Untitled');
  GetFontNames;
  SetupRuler;
  SelectionChange(Self);
```

```
  CurrText.Name := DefFontData.Name;
  DC := GetDC(0);
  ScreenLogPixels := GetDeviceCaps(DC, LOGPIXELSY);
  CurrText.Size := -MulDiv(DefFontData.Height, 72, ScreenLogPixels);

  LanguageKorean.Checked := TRUE;
  LanguageEnglish.Tag := ENGLISH;
  LanguageKorean.Tag := KOREAN;  // 한국어 태그
end;

...중간 생략...

procedure TMainForm.SwitchLanguage(Sender: TObject);
var
  Name : String;
  Size : Integer;
begin
  if LoadNewResourceModule(TComponent(Sender).Tag) <> 0 then
  begin
    Name := FontName.Text;
    Size := StrToInt(FontSize.Text);
    ReinitializeForms;
    LanguageEnglish.Checked := LanguageEnglish = Sender;
    LanguageKorean.Checked  := LanguageKorean  = Sender;  // 한국어 추가

    CurrText.Name := Name;
    CurrText.Size := Size;
    SelectionChange(Self);
    FontName.SelLength := 0;

    SetupRuler;
    Editor.SetFocus;
  end;
end;

...중간 생략...

   end.
```

이와 같이 작성하고, 우선 컴파일하여 실행 파일을 만든다. 컴파일이 되어야만(실행 파일이 만들어져야만) 리소스 DLL 마법사를 만들거나 수정할 수 있다. 다음으로, File | New에서 New 탭에 있는 Resource DLL Wizard 아이콘을 선택한다.

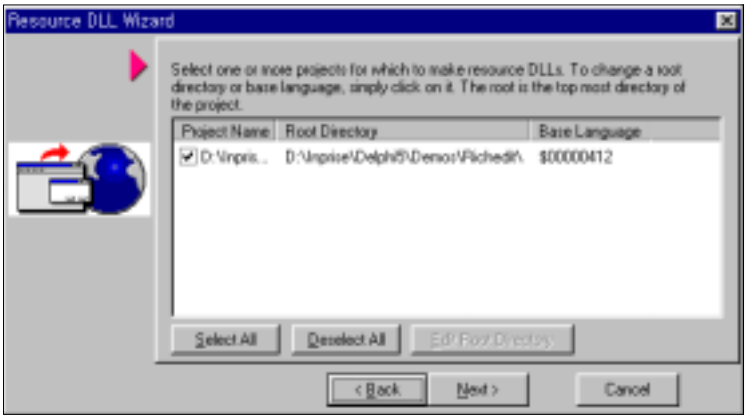


그림 9.1
리소스 Dll 1

리소스 DLL을 만들기 위해 하나 이상의 프로젝트를 선택한다. 디렉토리나 기준 클래스를 변경할 경우에는 단순히 마우스로 클릭하고 변경하면 된다.

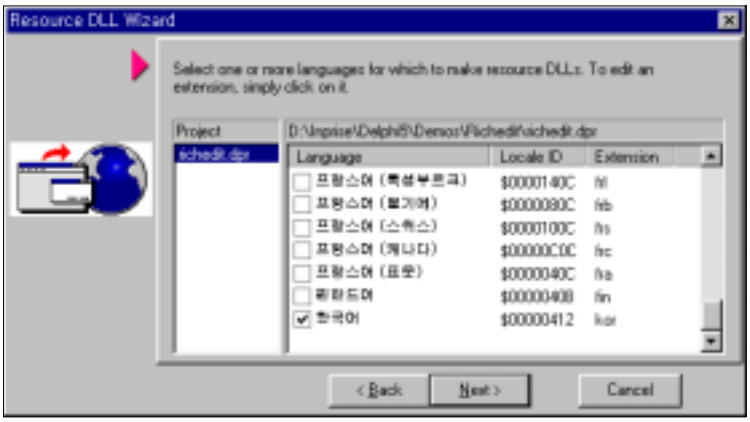


그림 9.2
리소스 Dll 2

리소스 DLL로 만들기 위한 언어를 다중 선택할 수 있다. 이 과정에서 영어(미국)와 한국어를 선택한다. 이 대화상자에서는 새로 DLL을 만들 경우에는 그 언어에 맞는 디렉토리와 파일들을 만들어주지만, 업데이트할 경우는 기존의 디렉토리에 그대로 덧붙여 적용된다.

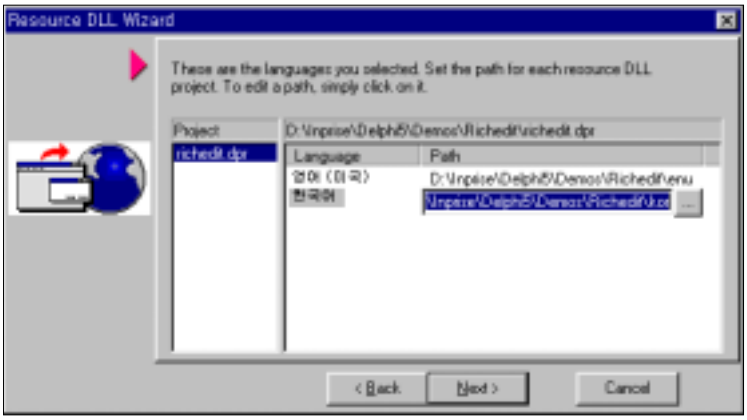


그림 9.3
리소스 Dll 3

선택된 언어와 경로명이 올바른지 확인한다.

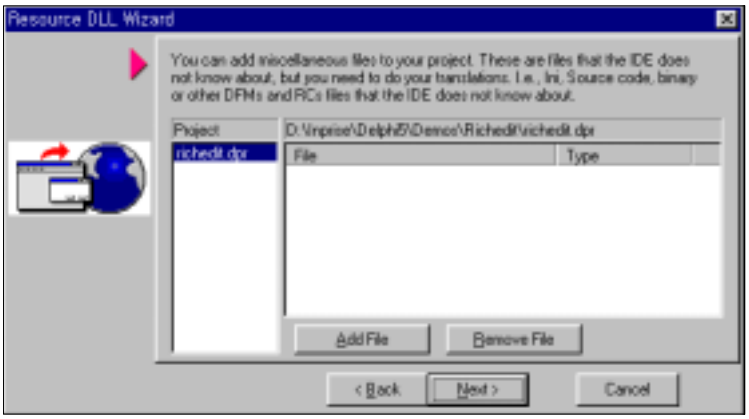


그림 9.4
리소스 Dll 4

프로젝트에 추가할 파일이 있는지 확인한다. 델파이 IDE에서는 추가적으로 할당하는 파일에 대해서는 인식하지 못하므로 직접 수동으로 파일들을 추가해야만 한다.

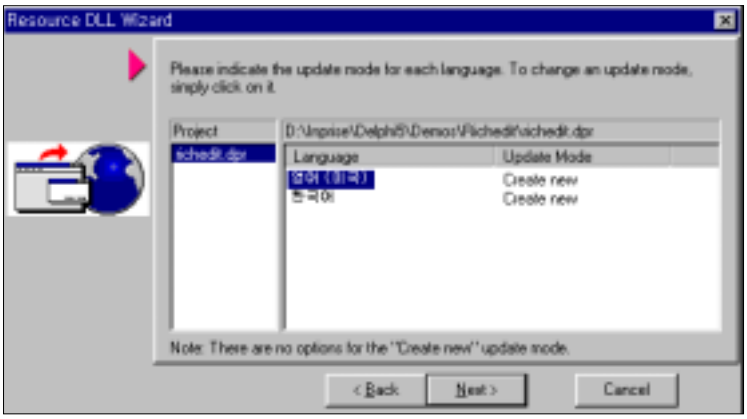


그림 9.5
리소스 Dll 5

선택된 리소스 DLL들이 새로 생성되는 것들인지, 아니면 기존의 것을 업데이트하는 것인지 확인한다.

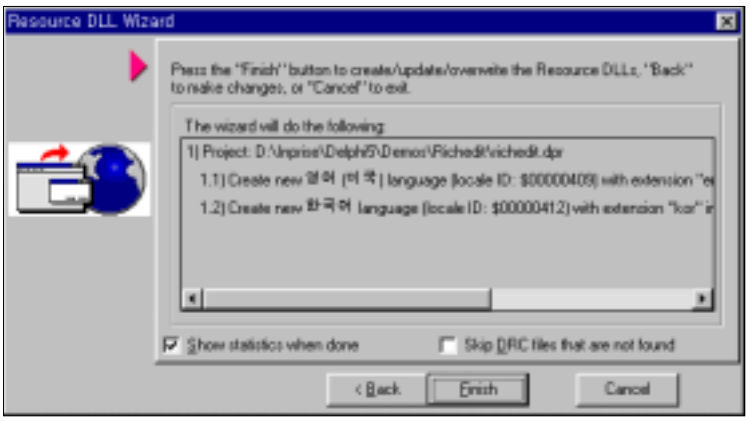


그림 9.6
리소스 Dll 6

최종적으로, 각각에 대해 최종 점검한다.

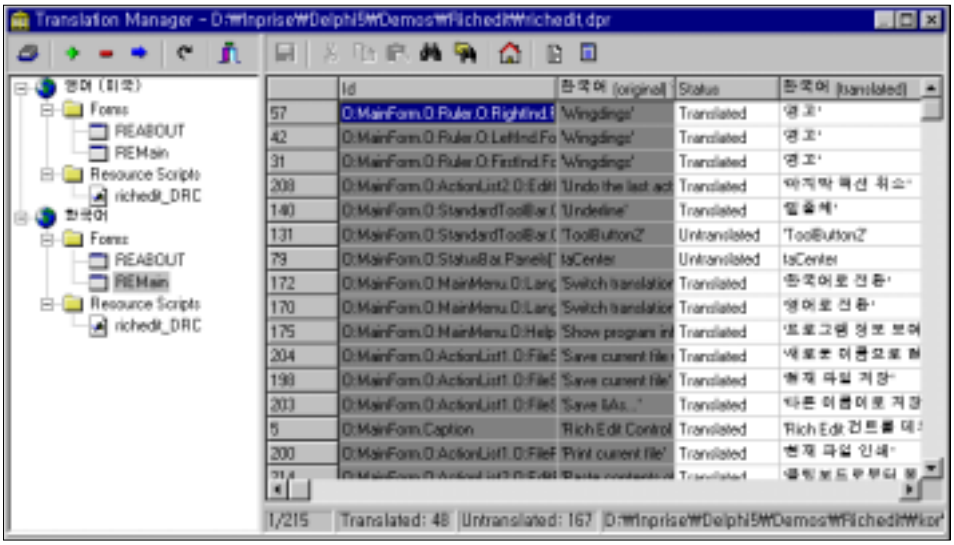


그림 9.7
번역 매니저

이렇게 만들어진 리소스 DLL들은 번역 매니저(Translation Manager)에서 리소스 문자열들을 각국에 맞는 언어로 고칠 수 있다. 화면으로 보기에는 리소스 워크샵과 비슷한 모양새를 지닌다. 그러나 문자열들 보기 좋게 그리드 형태로 나열되어 있기 때문에 쉽게 한글화로 구현할 수 있다.

프로젝트를 컴파일하게 되면 자동적으로 현재 작업 디렉토리 밑에 RichEdit.kor과 RichEdit.enu가 생성된다. 이 파일들이 바로 리소스 DLL 파일들인 것이다. 마지막으로, 애플리케이션을 실행해 보면 다음과 같은 화면을 얻게 된다.

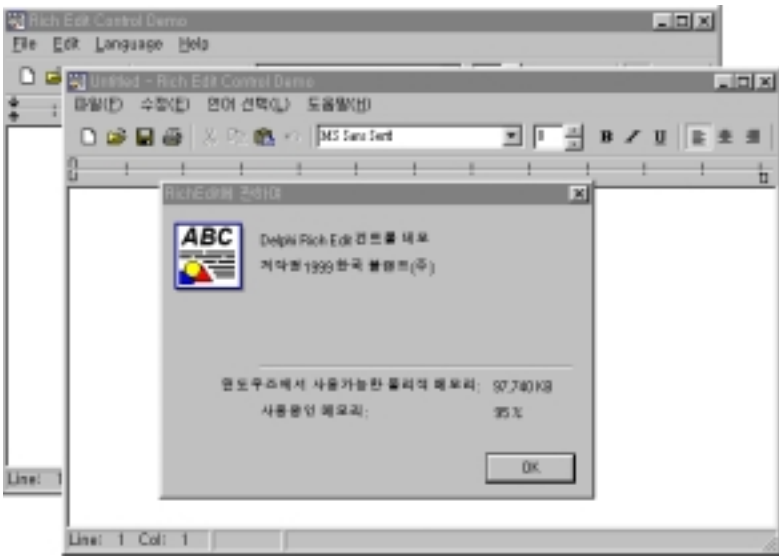


그림 9.8
한/영 전환 리소스 DLL 실행 화면

CPL(Control Panel Applet) 마법사

제어판(Control Panel)에 보면 윈도우의 다양한 윈도우즈 운영에 관한 제어 프로그램들이 등록되어 있어서 쉽게 시스템을 제어할 수 있는 것을 확인할 수 있다. 이 제어판의 컨트롤들은 Windows의 일부분으로 동작되기 때문에 스탠드 얼론(홀로 실행 가능한) 애플리케이션과는 달리, DLL과 마찬가지로 동작한다. 마이크로소프트에서는 제어판의 컨트롤들을 애플릿(Applet)이라고 표현한다.

컨트롤 패널(Control Panel) 애플릿은 하나의 DLL(Dynamically-Linked Library)로, 이것은 확장명이 단지 CPL로 바뀐 것뿐, 다른 점은 거의 없다.

일반적으로, CPL을 처리하기 위해서는 CPL 유닛에 정의되어 있는 다음과 같은 API 함수를 이용하는 것이 일반적이었으나, 지금의 델파이 5에는 이러한 API들을 모두 커버할 수 있는 마법사가 있다.

```
function CPlApplet(hwndCPL : HWND; Msg : Word; lParam1, lParam2 : LongINt)
 : LongInt;stdcall;
```

마법사를 이용할 경우, CPL 애플릿을 만들 수 있게끔 기본 프레임웍은 생성된다. 그러면 실제로 어떻게 제어판의 컨트롤을 만드는지 알아보자.

우선, File | New | Control Panel Application 아이콘을 선택해 새로운 CPL 프로젝트를 연다. 객체 조사기에 보면 AppletIcon과 Caption 속성이 보인다. 이 속성들을 적당한 값들로 할당한다. 그리고 OnActivate 이벤트에서 다음과 같이 작성한다.

목록 9.19 Ch9\cpl_demo\MainFrm.pas

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  CtlPanel;

type
  TAppletModule1 = class(TAppletModule)
    procedure AppletModule1Activate(Sender: TObject; Data: Integer);
  private
  protected
  public
  end;

var
  AppletModule1: TAppletModule1;
  h : HIcon;
  i : TIcon;

implementation

uses CalcFrm;

{$R *.DFM}

procedure TAppletModule1.AppletModule1Activate(Sender: TObject;
  Data: Integer);
```

```
var
  SysTime: TSystemTime;
  DateTime: TDateTime;
begin
  with TCalcForm.Create(Application) do
  begin
    if ShowModal=mrOK then
    begin
        // 시스템 날짜와 시간을 바꾼다.
      DateTime := Now;
      ReplaceDate(DateTime, calDLLCalendar.CalendarDate);
      DateTimeToSystemTime(DateTime, SysTime);
      SetLocalTime(SysTime);
    end;
    Free;
  end;
end;

end.
```

컨트롤 애플릿에서 호출하여 실행될 폼은 단순히 폼에 달력 컴포넌트와 버튼 하나만 첨가한 것이다. 데모로 확인하기 위해 현재 달력 이외만 제공하는 폼을 만들었다. 이 프로젝트를 Build하게 되면 cplappl.cpl이라는 파일로 생성되는데, 이 파일을 Windows\System 디렉토리로 옮겨주면 모든 과정은 마무리된다(그러나 델파이는 이 과정까지도 할 필요 없이 애플릿 모듈 그 자체에서 AppletModule1 클래스를 마우스로 클릭하고, 마우스 오른쪽 버튼을 클릭하면 Install Control Panel, UnInstall Control Panel, Launch Control Panel의 세 가지 옵션으로 애플릿을 설치, 제거, 제어판 활성화할 수 있다).

다음 그림은 이 제어판 컨트롤을 실행시킨 화면이다.

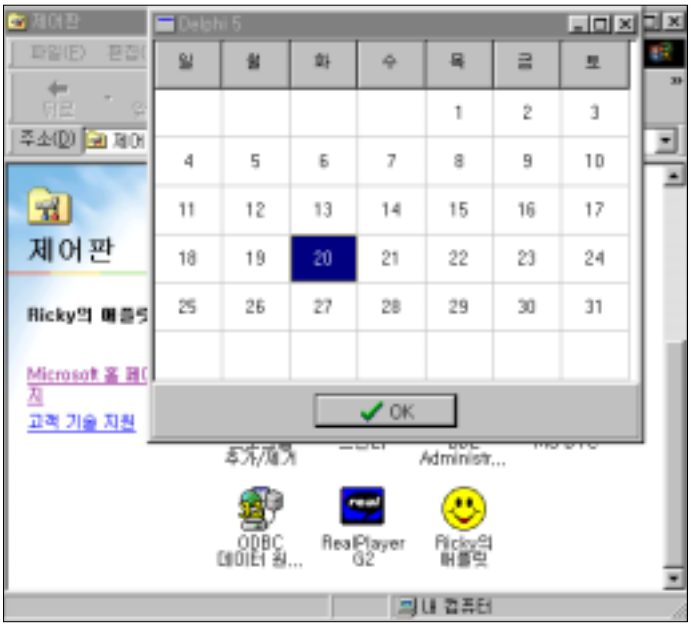


그림 9.9
컨트롤 패널 애플릿

요약

DLL은 윈도우즈 애플리케이션의 핵심적인 요소이다. 특히, 코드의 재사용이나 애플리케이션 구DLL은 윈도우즈 애플리케이션의 핵심적인 요소이다. 특히, 코드의 재사용이나 애플리케이션 구조의 모듈화에 큰 역할을 한다. 이번 장에서는 DLL을 사용하는 이유에 대해서 이야기하고, 델파이로 DLL을 만들거나 사용하는 여러 가지 방법도 설명했다. 또한, 여러 애플리케이션들이 DLL의 전역 데이터를 공유하는 기법에 대해서도 이야기했다.

이번 장을 공부한 독자는 이제 DLL을 만들거나 사용하는 데 큰 어려움이 없을 것이다. 이후에도 DLL을 사용하거나 작성하는 부분이 나오니 이번 장에서 잘 이해가 되지 않는 부분은 확실히 이해하고 넘어가기 바란다.