13장 고난도 기법들

* 애플리케이션 메시지 처리의 고급 기법
* 애플리케이션의 중복 실행 방지
* 델파이의 내장 어셈블러
* 윈도우즈 훅의 활용
* C/C++ OBJ 파일 활용
* C++ 클래스 활용법
* 썽킹에 대해

델파이를 사용하다 보면 일반적인 기법으로는 해결할 수 없는 문제들을 만나곤 한다. 이번 장에서는 해결하기 어려운 문제들을 뚫고 나갈 수 있는 고난도의 기법들을 설명하겠다. 이번 장은 다른 어떤 장들보다 Win32 API들이 많이 등장한다. 이번 장에서는 VCL로는 해결하기 어려운 작업을 API로 해결하는 기법들을 제시한다. 이번 장을 통해서 윈도우 프로시저, 프로그램 인스턴스의 중복 실행, 윈도우 후킹, 델파이와 C++ 코드의 공유 등의 개념을 배우게 될 것이다.

애플리케이션 메시지 처리의 고급 기법

5장, "메시지의 이해"에서 이야기했듯이 윈도우 프로시저란, 윈도우즈가 보내는 메시지를 받는 함수를 뜻한다고 했다. 델파이 애플리케이션의 기본 윈도우 프로시저는 Application 객체가 가지고 있는 윈도우의 윈도우 프로시저이다. 그리고 TApplication 클래스는 개발자가 직업 메시지를 다룰 수 있도록 OnMessage 이벤트까지 제공하므로 윈도우즈가 애플리케이션에게 보내는 메시지를 개발자가 직접 처리할 수 있다.

그런데 문제는, TApplication의 OnMessage 이벤트란 것이 애플리케이션의 메시지 큐에서 꺼내온 메시지에 대해서만 발생한다는 점이다(5장을 참고해서 메시지 관련 용어들에 대해, 특히 Send와 Post의 차이를 다시 보기 바란다). 애플리케이션의 큐에 들어오는 메시지는 보통 윈도우 관리용 메시지(WM_PAINT나 WM_SIZE 등)이며, Win32 API 함수 PostMessage()나 PostAppMessage(), BroadcastSystemMessage() 같은 것들이 보낸 메시지들이다. SendMessage() API 함수처럼 윈도우 프로시저에게 직접 메시지가 도착하는 경우에는 TApplication.OnMessage 이벤트가 발생하지 않는다. 따라서, 델파이의 기본적인 클래스나 컴포넌트들로는 SendMessage()로 보낸 메시지를 처리할 수 없다.

서브클래싱

애플리케이션이 받는 메시지를 직접 처리하려면 Application 윈도우의 윈도우 프로시저를 개발자가 만든 프로시저로 변경해야 한다. 새로운 윈도우 프로시저 안에서 메시지를 적절히 처리한 후 메시지를 기본 윈도우 프로시저에 메시지를 넘겨서 나머지 처리를 맡기는 방식이 필요하다. 이처럼 메시지를 가로채서 기본 윈도우 프로시저의 행동을 변경하는 것을 서브클래싱(subclassing)이라고 한다.

어떤 윈도우의 윈도우 프로시저를 변경하려면 SetWindowLong() Win32 API 함수에 GWL_WNDPROC 상수와 윈도우 프로시저의 주소를 넣으면 된다. 윈도우 프로시저 함수에 사용할 수 있는 함수는 크게 두 가지로 나눌 수 있다. 하나는 API의 윈도우 프로시저 정의를 따르는 것이고, 또 하나는 델파이의 메시지 처리 지원 기능을 사용하는 윈도우 메소드(window method)라는 것이다. 그럼, 두 종류의 윈도우 프로시저를 차례로 알아보자.



VCL 윈도우의 윈도우 프로시저를 서브클래싱할 때 주의해야 할 점이 있다. VCL 윈도우 자체가 개발자 모르게 재생성될 수 있다는 점이다. 그런 일이 일어나면 애플리케이션이 의도한 대로 작동하지 않게 된다. 서브클래싱을 하려고 하는 윈도우의 윈도우 핸들이 실행 도중에 변경될 가능성이 있다면 서브클래싱을 적용할 때 신경을 많이 써야 한다. 안전한 방법은 Application.HookMainWindow()를 사용하는 것인데, 이에 대해서는 나중에 다시 이야기하겠다.

Win32 API의 정의를 따르는 윈도우 프로시저

API 윈도우 프로시저는 다음과 같은 형태로 선언되어야 한다.

```
function AWndProc(Handle: hWnd; Msg, wParam, lParam: Longint);
       Longint; stdcall;
```

Handle 속성은 메시지를 받는 윈도우의 핸들이고, Msg 인자는 윈도우 메시지이다. 그리고 wParam과 lParam 인자는 메시지에 관련된 추가 정보를 뜻한다. 이 함수가 돌려주는 리턴값의 의미는 메시지에 따라 달라진다. 그리고 API와 통신을 하는 함수이므로 호출 규약은 반드시 stdcall로 지정해야 한다.

위와 같은 형태의 윈도우 프로시저 함수를 만든 다음에는 그 함수를 Win32에 등록해야 하는데, 그때는 SetWindowLong() 함수를 사용한다. Application의 윈도우를 서브클래싱하는 경우는 다음과 같은 형태가 된다.

```
var
  WProc: Pointer;
begin
  WProc := Pointer(SetWindowLong(Application.Handle, GWL_WNDPROC,
      Integer(@NewWndProc)));
```

위의 코드를 수행하면 WProc에는 이전의 윈도우 프로시저를 가리키는 포인터가 저장된다. 새로운 윈도우 프로시저에서 모든 메시지를 일일이 다 처리한다면 문제가 없겠지만, 그렇지 않다면 저장한 포인터를 CallWindowProc() API에 넣어서 원래의 윈도우 프로시저에게 나머지 메시지 처리를 넘겨줘야 한다. 다음 코드를 보면 윈도우 프로시저가 어떤 형태를 지녀야 할지 감을 잡을 수 있을 것이다.

```
function NewWndProc(Handle: hWnd; Msg, wParam, lParam: Longint);
      Longint; stdcall;
begin
  { Msg값을 보고 어떤 메시지인지 판단한 다음, wParam과 lParam의 정보를 참고
      해서 적절한 작업을 수행한다. 이 프로시저 안에서 처리하지 않는 메시지는
      아래의 코드를 통해서 원래의 윈도우 프로시저에 넘겨줘야 한다. }
  Result := CallWindowProc(WProc, Application.Handle, Msg, wParam,
      lParam);
end;
```

목록 13.1은 Application의 윈도우 프로시저를 서브클래싱해서 DDGM_FOOMSG라는 사용자 정의 메시지를 처리하는 ScWndPrc.pas의 소스 코드이다.

목록 13.1 Ch13\Wndproc\ScWndPrc.pas

```
unit Scwndprc;

interface

uses Forms, Messages;

const
  DDGM_FOOMSG = WM_USER;
```

```
implementation

uses Windows, SysUtils, Dialogs;

var
  WProc: Pointer;

function NewWndProc(Handle: hWnd; Msg, wParam, lParam: Longint): Longint;
  stdcall;
{ 이 함수는 Win32 API 수준의 윈도우 프로시저이다. 이 함수는 Application의
 윈도우가 받는 메시지를 처리한다. }
begin
  if Msg = DDGM_FOOMSG then
     { 도착한 메시지가 사용자 정의 메시지이면 사용자에게 알려준다. }
    ShowMessage(Format('WndProc에 의한 메시지! 값은: $%x', [Msg]));
 { 원래 윈도우 프로시저에게 메시지를 넘겨준다. }
  Result := CallWindowProc(WProc, Handle, Msg, wParam, lParam);
end;

initialization
 { Application의 윈도우 프로시저를 설정한다. }
  WProc := Pointer(SetWindowLong(Application.Handle, gwl_WndProc,
    Integer(@NewWndProc)));
 end.
```



GetWindowLong()이 돌려준 원래 윈도우 프로시저를 버리면 안된다. 새 윈도우 프로시저 안에서 처리하지 않은 메시지를 원래의 윈도우 프로시저에게 넘겨주지 않으면 애플리케이션은 물론이고, 운영체제 자체가 다운될 수 있다.

델파이의 윈도우 메소드

델파이는 API 윈도우 프로시저와 델파이의 메소드를 연결시켜 주는 MakeObject Instance()라는 함수를 제공한다. MakeObjectInstance()를 이용하면 TWndMethod형으로 만들어진 어떤 메소드를 윈도우 프로시저로 작동하도록 만들 수 있다. MakeObject Instance()는 Forms 유닛에 다음과 같이 선언되어 있다.

```
function MakeObjectInstance(Method: TWndMethod): Pointer;
```

그리고 TWndMethod 역시 Forms 유닛에 정의되어 있다. 다음과 같다.

```
type
  TWndMethod = procedure(var Message: TMessage) of object;
```

MakeObjectInstance()의 리턴값은 새로 만들어진 윈도우 프로시저를 가리키는 Pointer 이다. 이 리턴값을 SetWindowLong()에 넘겨주면 된다. 그리고 MakeObjectInstance()로 만든 윈도우 메소드를 사용한 후에는 반드시 FreeObjectInstance() 함수를 사용해서 그 윈도우 메소드를 해제시켜야 한다.

지금까지 이야기한 서브클래싱의 여러 가지 사용법은 WinProc.dpr이라는 예제를 보면 알 수 있을 것이다. 이 예제는 OnMessage 이벤트 처리 프로시저뿐만 아니라 Win32 API 수준의 윈도우 프로시저와 델파이의 윈도우 메소드를 모두 이용해서 Application의 윈도우를 서브클래싱한다. 메인 폼의 모습은 그림 13.1과 같다.

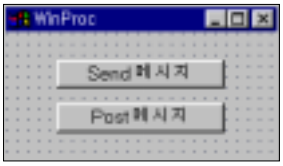


그림 13.1
WndProc 예제의 메인 폼CatchIt의 메인 폼

목록 13.2는 WndProc 프로젝트의 메인 폼 유닛인 Main.pas의 소스 코드이다.

목록 13.2 Ch13\Wndproc\Main.pas

```
unit Main;

interface

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

type
  TMainForm = class(TForm)
    SendBtn: TButton;
    PostBtn: TButton;
    procedure SendBtnClick(Sender: TObject);
    procedure PostBtnClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    OldWndProc: Pointer;
    WndProcPtr: Pointer;
    procedure WndMethod(var Msg: TMessage);
    procedure HandleAppMessage(var Msg: TMsg; var Handled: Boolean);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses ScWndPrc;


procedure TMainForm.HandleAppMessage(var Msg: TMsg; var Handled: Boolean);
```

```
{ Application의 OnMessage 이벤트 처리 프로시저 }
begin
  if Msg.Message = DDGM_FOOMSG then
    { 사용자 정의 메시지이면 사용자에게 알려준다. }
    ShowMessage(Format('OnMessage에 의한 메시지! 값은: $%x',
      [Msg.Message]));
end;

procedure TMainForm.WndMethod(var Msg: TMessage);
begin
  if Msg.Msg = DDGM_FOOMSG then
    { 사용자 정의 메시지이면 사용자에게 알려준다. }
    ShowMessage(Format('WndMethod에 의한 메시지! 값은: $%x',
              [Msg.Msg]));
  with Msg do
    { 메시지를 원래 윈도우 프로시저에게 넘긴다. }
    Result := CallWindowProc(OldWndProc, Application.Handle, Msg, wParam,
      lParam);
end;

procedure TMainForm.SendBtnClick(Sender: TObject);
begin
  SendMessage(Application.Handle, DDGM_FOOMSG, 0, 0);
end;

procedure TMainForm.PostBtnClick(Sender: TObject);
begin
  PostMessage(Application.Handle, DDGM_FOOMSG, 0, 0);
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  Application.OnMessage := HandleAppMessage; // OnMessage 처리 프로시저를 설정
  WndProcPtr := MakeObjectInstance(WndMethod);   // 윈도우 프로시저를 설정
 { Application의 윈도우의 윈도우 프로시저를 설정 }
  OldWndProc := Pointer(SetWindowLong(Application.Handle, GWL_WNDPROC,
    Integer(WndProcPtr)));
end;
```

```
procedure TMainForm.FormDestroy(Sender: TObject);
begin
  { Application 윈도우의 원래 윈도우 프로시저를 복구한다.}
  SetWindowLong(Application.Handle, GWL_WNDPROC, Longint(OldWndProc));
  { 직접 정의한 윈도우 프로시저를 해제한다. }
  FreeObjectInstance(WndProcPtr);
end;

end.
```

사용자가 SendBtn을 클릭하면 애플리케이션은 SendMessage() API 함수를 이용해서 Application의 윈도우 핸들에게 사용자 정의 메시지인 DDGM_FOOMSG를 직접 보낸다. 반면, PostBtn을 클릭하면 애플리케이션은 PostMessage() API 함수를 이용해서 동일한 메시지를 애플리케이션의 메시지 큐에 보낸다.

HandleAppMessage()는 Application.OnMessage 이벤트를 처리한다. 이 프로시저는 사용자 정의 메시지가 들어오면 간단한 메시지 상자를 표시한다. 이 프로시저는 메인 폼의 OnCreate 이벤트 처리 프로시저에서 OnMessage 이벤트에 배정된다.

메인 폼의 OnDestroy 이벤트 처리 프로시저를 보면 FreeObjectInstance()를 호출해서 MakeProcInstance()로 만든 프로시저를 해제하기 전에, 우선 Application의 윈도우 프로시저를 원래대로(OldWndProc) 돌려 놓는다는 점을 알 수 있다. 원래 윈도우 프로시저를 먼저 복구하지 않고 FreeObjectInstance()를 호출해 버리면 그 윈도우에는 윈도우 프로시저가 없어져 버리기 때문이다. 윈도우에 윈도우 프로시저가 없으면 그 애플리케이션 뿐만 아니라, 운영체제까지 다운될 수 있다.

implementation부의 uses절을 보면 앞에서 설명한 ScWndProc가 포함되어 있다. 즉, 이 애플리케이션의 윈도우 프로시저는 두 번 서브클래싱된다. 한 번은 ScWndProc가 API 기법을 이용해서 서브클래싱을 하고, 또 한 번은 Main이 윈도우 메소드 기법을 이용해서 서브클래싱한다. 그러나 윈도우 프로시저 안에서 CallWindowProc()를 통해서 메시지를 원래 윈도우 프로시저에게 넘겨주기만 한다면 이처럼 한 번 이상 서브클래싱을 해도 문제가 생기지 않는다.

이 애플리케이션을 실행시켜서 버튼들을 눌러보면 두 윈도우 프로시저가 모두 메시지 상자를 표시할 것이다. 그리고 Application.OnMessage는 윈도우의 메시지 큐에 들어온(post) 메시지에만 반응한다는 것도 알 수 있을 것이다.

HookMainWindow()

VCL의 메시지를 가로챌 때 주로 사용하는 것이 TApplication의 HookMainWindow()라는 메소드이다. 이 메소드는 TApplication의 WndProc() 메소드가 받을 메시지를 가로채는 메시지 처리 함수를 설치하는 역할을 한다. HookMainWindow()의 선언은 다음과 같다.

```
procedure HookMainWindow(Hook: TWindowHook);
```

이 메소드의 인자는 TWindowHook형으로, TWindowHook형의 정의는 같다.

```
type
  TWindowHook = function (Var Message: TMessage): Boolean of object;
```

HookMainWindow()의 사용법은 간단하다. 메시지 처리 함수를 인자에 넣어서 호출하면 된다. 그러면 그 함수는 TApplication.WndProc()가 메시지를 받기 전에, 먼저 메시지를 받게 된다. 이러한 과정을 "윈도우훅(hook)을 설치한다, 또는 윈도우 훅 목록에 훅을 추가한다."고 말하며, 그러한 함수를 윈도우 훅 메소드 또는 그냥 훅이라고 한다. 윈도우 훅 메소드가 True를 돌려주면 TApplication의 WndProc()는 그 메시지를 윈도우 훅 메소드가 처리한 것으로 간주하고, 자신은 메시지를 처리하지 않는다.

윈도우 훅 메소드의 사용이 끝난 이후에는 UnhookMainWinow()를 호출해서 훅을 제거해야 한다. UnhookMainWinow()의 선언은 다음과 같다.

```
procedure UnhookMainWindow(Hook: TWindowHook);
```

목록 13.3은 애플리케이션이 받는 메시지들을 가로채서 메시지의 내용을 목록 상자에 추가하는 간단한 예제 프로젝트의 소스 코드이다. 그리고 그림 13.2는 이 예제 프로젝트의 실행 모습이다.

목록 13.3 Ch13\HookWnd\HookMain.pas

```
unit HookMain;

interface

uses
```

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

type
  THookForm = class(TForm)
    SendBtn: TButton;
    GroupBox1: TGroupBox;
    LogList: TListBox;
    DoLog: TCheckBox;
    ExitBtn: TButton;
    procedure SendBtnClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure ExitBtnClick(Sender: TObject);
  private
    function AppWindowHook(var Message: TMessage): Boolean;
  end;

var
  HookForm: THookForm;

implementation

{$R *.DFM}

procedure THookForm.FormCreate(Sender: TObject);
begin
  Application.HookMainWindow(AppWindowHook);
end;

procedure THookForm.FormDestroy(Sender: TObject);
begin
  Application.UnhookMainWindow(AppWindowHook);
end;

function THookForm.AppWindowHook(var Message: TMessage): Boolean;
const
  LogStr = 'Message ID: $%x, WParam: $%x, LParam: $%x';
begin
```

```
  Result := True;
  if DoLog.Checked then
    with Message do
      LogList.Items.Add(Format(LogStr, [Msg, WParam, LParam]));
end;

procedure THookForm.SendBtnClick(Sender: TObject);
begin
  SendMessage(Application.Handle, WM_NULL, 0, 0);
end;

procedure THookForm.ExitBtnClick(Sender: TObject);
begin
  Close;
end;

end.
```

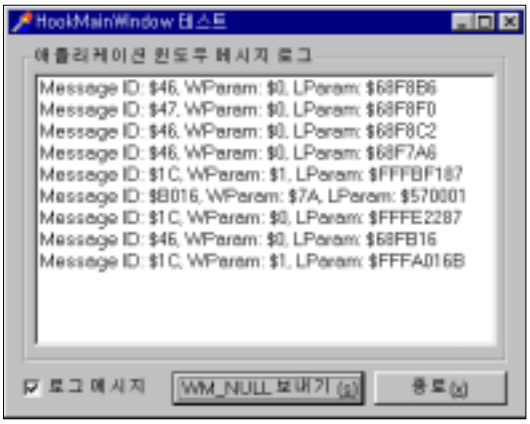


그림 13.2
HookWnd 프로젝트가 Application이 받은 메시지들을 표시한 모습

애플리케이션의 중복 실행 방지

중복 실행이란, 한 애플리케이션의 인스턴스가 하나 이상 존재하는 것을 말한다. Win32 운영체제는 멀티태스킹 환경이므로 한 애플리케이션을 한 번 이상 띄워서 여러 개의 복사본, 즉 인스턴스들을 동시에 실행하는 것이 가능하다. 그러나 애플리케이션의 기능이나 성격에 따라서는 한 시점에서 단 하나의 인스턴스만 존재하도록 해야 하는 경우가 있다.

16비트 윈도우즈에서는 중복 실행 방지가 무척 간단했다. 중복 실행을 방지하려면 이미 시스템에 자신과 동일한 애플리케이션의 인스턴스가 실행중인지만 알면 되는데, 16비트 윈도우즈의 경우에는 시스템 변수 hPrevInst로 즉시 알아낼 수 있었다. 16비트 윈도우즈에서 hPrevInst의 값이 0이 아니면 같은 애플리케이션의 다른 인스턴스가 이미 존재한다는 뜻이다. 그러나 3장, "Win32 API"에서도 이야기했듯이 Win32 환경에서는 각각의 프로세스가 완전히 독립된 주소 공간에서 실행되므로 hPrevInst는 다른 인스턴스의 존재 여부에 상관 없이 항상 0이 된다. 따라서, Win32에서는 hPrevInst를 중복 실행 방지에 써먹을 수 없다.

16비트나 32비트 윈도우즈에서 공히 사용할 수 있는 기법으로는 API 함수 FindWindow()를 사용해서 애플리케이션의 윈도우를 찾아 판단하는 방법이 있다. 그러나 이 기법에는 두 가지 중요한 단점이 있다. 우선, FindWindow()는 윈도우의 클래스 이름과 윈도우 이름(제목)으로 윈도우를 찾는다. 그런데 서로 다른 애플리케이션이 동일한 클래스 이름이나 윈도우의 이름을 가지지 않는다고 보장할 방법이 없다. 또한, 윈도우 이름은 실행중에도 바뀌는 경우가 많으므로(델파이의 메인 윈도우의 제목에는 현재 프로젝트의 이름이 포함된다) 윈도우 이름으로 윈도우를 찾는 것은 정확한 결과를 기대할 수 없다. 두 번째로, FindWindow()는 시스템에 활성화된 모든 최상위 윈도우들을 검사해야 하므로 수행 속도가 느리다.

Win32 환경에서 최적의 해결책은 프로세스들의 경계를 넘을 수 있는 전역 API 객체들을 이용하는 것이다. 11장, "멀티스레드 애플리케이션의 작성"에 나왔던 뮤텍스나 세마포어 같은 것들이 프로세스들의 경계를 넘을 수 있는 전역 API 객체들이다. 그들 중 가장 사용하기 간편한 것이 뮤텍스이기 때문에 중복 실행 방지에는 뮤텍스를 주로 사용한다.

그럼, 뮤텍스를 이용한 중복 실행 방지의 논리적인 과정을 따라가 보자. 애플리케이션의 첫 번째 인스턴스가 처음 실행될 때 CreateMutex() API 함수를 이용해서 뮤텍스를 하나 만든다. 이때 lpName 인자에 고유한 문자열을 넣어서 뮤텍스의 이름을 지정한다. 나중에 다른 인스턴스가 실행될 때 그 인스턴스는 그 뮤텍스의 이름을

OpenMutex() 함수의 인자에 넣어서 뮤텍스를 연다. 만일 OpenMutex()가 성공한다면 그 뮤텍스가 존재한다는 뜻이며, 따라서 이미 다른 인스턴스가 실행중이라는 뜻이 된다.

뮤텍스가 존재하면 즉시 실행을 종료하는 것도 좋겠지만, 제대로 된 애플리케이션들은 첫 번째 인스턴스의 메인 폼에 포커스를 옮겨준다. 이전 인스턴스의 메인 폼에 포커스를 옮기는 방법은 여러 가지이겠지만, 가장 깔끔한 방법이라면 RegisterWindowMessage() 함수를 이용해서 애플리케이션만의 고유한 메시지를 윈도우즈에 등록하고, 그 메시지를 보내서 이전 인스턴스가 포커스를 가지도록 하는 것이다. 두 번째 인스턴스가 윈도우즈에 등록된 고유한 메시지를 윈도우즈 전체에 뿌리면(메시지를 받을 윈도우를 알 수 없으므로) 첫 번째 인스턴스는 그 메시지를 받아서 자신에게 포커스를 설정하면 된다. 이와 같은 과정을 실제로 구현한 예제가 부록 CD의 OneInst.dpr이다. 목록 13.4는 중복 실행 방지를 구현하는 코드가 들어 있는 MultInst.pas의 소스 코드이고, 목록 13.5는 이 예제의 메인 폼 유닛인 OIMain.pas의 소스 코드이다. 이 예제의 실행 모습은 그림 13.3에 나와 있다.

목록 13.4 Ch13\Oneinst\ultInst.pas

```
unit multinst;

interface

uses Forms, Windows, Dialogs, SysUtils;

// Windows 유닛에 있는 BroadcastSystemMessage()의 선언이 정확하지 않기
// 때문에 여기서 직접 정의한다.
function BroadcastSystemMessage(Flags: DWORD; Recipients: PDWORD;
  uiMessage: UINT; wParam: WPARAM; lParam: LPARAM): Longint; stdcall;
  external 'user32.dll';


const
  MI_NO_ERROR          = 0;
  MI_FAIL_SUBCLASS     = 1;
  MI_FAIL_CREATE_MUTEX = 2;

{ 시동 과정에서 에러가 생겼는지를 알려주는 함수.
```

```
 이 함수의 리턴값은 MI_* 에러 플래그들 중 하나나 그 이상이 결합된 값이다. }
function GetMIError: Integer;

implementation

const
  UniqueAppStr : PChar = 'I am the Eggman!';

var
  MessageId: Integer;
  WProc: TFNWndProc = Nil;
  MutHandle: THandle = 0;
  MIError: Integer = 0;

function GetMIError: Integer;
begin
  Result := MIError;
end;

function NewWndProc(Handle: HWND; Msg: Integer; wParam, lParam: Longint):
  Longint; stdcall;
begin

 { 등록된 메시지이면... }
  if Msg = MessageID then
  begin
     { 메인 폼이 최소화 상태이면 이전 크기로 되돌린 후 애플리케이션에 포커스를
       설정한다. }
   if IsIconic(Application.Handle) then
    begin
      Application.MainForm.WindowState := wsNormal;
      Application.Restore;
    end;
    SetForegroundWindow(Application.MainForm.Handle);
    Result := 0;
  end
  { 등록된 메시지가 아니면 원래의 윈도우 프로시저에게 메시지를 넘긴다. }
  else
    Result := CallWindowProc(WProc, Handle, Msg, wParam, lParam);
```

```
end;

procedure SubClassApplication;
begin
 { 이 유닛을 사용할 독자를 위해 Application.OnMessage는 건드리지
     않겠다. 대신, 애플리케이션의 윈도우 프로시저를 서브클래싱한다. }
  WProc := TFNWndProc(SetWindowLong(Application.Handle, GWL_WNDPROC,
                                    Longint(@NewWndProc)));
  { 에러가 발생하면 에러 플래그를 적절히 설정한다. }
  if WProc = Nil then
    MIError := MIError or MI_FAIL_SUBCLASS;
end;

procedure DoFirstInstance;
begin
  SubClassApplication;
  MutHandle := CreateMutex(Nil, False, UniqueAppStr);
  if MutHandle = 0 then
    MIError := MIError or MI_FAIL_CREATE_MUTEX;
end;

procedure BroadcastFocusMessage;
{ 이미 다른 인스턴스가 실행중이면 이 메소드가 실행된다. }
var
  BSMRecipients: DWORD;
begin
  { 다른 인스턴스가 실행중이므로 메인 폼을 표시하지 않는다. }
  Application.ShowMainForm := False;
 { 다른 인스턴스에게 포커스를 설정하라는 메시지를 뿌린다. }
  BSMRecipients := BSM_APPLICATIONS;
  BroadCastSystemMessage(BSF_IGNORECURRENTTASK or BSF_POSTMESSAGE,
    @BSMRecipients, MessageID, 0, 0);
  Application.Terminate;
end;

procedure InitInstance;
begin
  MutHandle := OpenMutex(MUTEX_ALL_ACCESS, False, UniqueAppStr);
  if MutHandle = 0 then
     { 지정한 뮤텍스 객체가 존재하지 않으므로 실행중인 다른 인스턴스가
```

```
        없는 것이다. }
    DoFirstInstance
  else
    BroadcastFocusMessage;
end;

initialization
  MessageID := RegisterWindowMessage(UniqueAppStr);
  InitInstance;
finalization
  if WProc <> Nil then
    { 원래의 윈도우 프로시저를 복구한다. }
    SetWindowLong(Application.Handle, GWL_WNDPROC, LongInt(WProc));
end.
```

목록 13.5 Ch13\Oneinst\OIMain.pas

```
unit OIMain;

interface

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

type
  TMainForm = class(TForm)
    Label1: TLabel;
    CloseBtn: TButton;
    procedure CloseBtnClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  MainForm: TMainForm;

implementation

uses MultInst;

{$R *.DFM}

procedure TMainForm.CloseBtnClick(Sender: TObject);
begin
  Close;
end;

end.
```

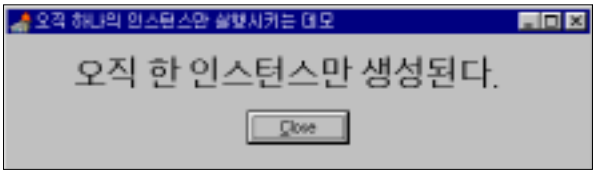


그림 13.3
OneInst 프로젝트의 메인 폼

델파이의 내장 어셈블러

델파이는 인터프리터 방식이 아닌 진짜 컴파일러를 기반으로 하는 도구이다. 그래서 오브젝트 파스칼 코드 중간에 어셈블리 코드를 집어넣을 수 있다. 어셈블리 코드는 델파이의 내장 어셈블러(BASM)가 처리한다. BASM에 대해 이야기하기 전에, 어셈블리 코드를 델파이 코드에 집어넣을 때 주의해야 할 점부터 알아보자.

"약 좋다고 남용 말자"라는 말도 있듯이 어셈블리 언어는 매우 강력하지만, 함부로 쓰는 것은 불필요한 에러나 버그를 불러온다. 다음은 어셈블리 언어를 사용할 때 주의해야 할 점들이다.

- 오브젝트 파스칼로 할 수 있는 일을 굳이 어셈블리로 하지는 말아야 한다. 예를 들어서, 직렬 포트 통신은 Win32 API로도 충분히 할 수 있으므로 어셈블리를 사용하는 것은 시간 낭비이다.
- 어셈블리 언어로 애플리케이션을 최적화하는 데 너무 많은 노력을 들이는 것은 좋지 않다. 물론, 어셈블리를 사용하면 델파이의 오브젝트 파스칼 코드보다 훨씬 더 빠르고 작은 실행 파일을 만들 수 있지만, 코드를 읽거나 유지/보수하는 것 역시 훨씬 더 힘들어진다. 그리고 사실, 델파이의 컴파일러 성능도 무시 못할 정도로 뛰어나다. 경우에 따라서는 직접 만든 어셈블리 코드보다 더 빠르고 간략한 코드를 델파이의 컴파일러가 만들 수도 있다.
- 어셈블리 코드에 상세한 주석을 붙여야 한다. 돌아서면 잊는 것이 사람이다. 어셈블리 코드는 주석이 없으면 정말 이해하기 힘들다.
- 어셈블리 코드로 하드웨어를 직접 건드려서는 안된다. 윈도우즈 9x에서는 그냥 넘어가는 수도 있지만, 윈도우즈 NT는 절대로 그냥 넘어가지 않는다.
- 될 수 있는 한 어셈블리 코드를 오브젝트 파스칼이 호출할 수 있는 형태, 즉 함수나 프로시저로 분리해서 만들어둬야 한다. 그러면 유닛 전체를 이해하기도 쉬울 뿐만 아니라, 다른 플랫폼으로 이식할 때에도 편하다.



이 책에서는 어셈블리 언어 자체에 대해서 설명하는 것이 아니라, 델파이에서 어셈블리를 사용하는 방법에 대해서만 설명한다. 어셈블리에 대해 잘 모르는 독자라면 이해하지 못하는 부분은 그냥 건너뛰어도 된다.
그리고 델파이 1에서 BASM을 다뤄본 독자라면 32비트 델파이의 BASM은 전혀 다른 물건이라는 점을 명심하기 바란다. 16비트 어셈블리 코드와 32비트 어셈블리 코드는 상당히 다르기 때문이다. 델파이 1에 사용한 어셈블리 코드를 32비트 BASM에 맞게 이식하느니, 새로 다시 쓰는 것이 나을 정도이다.

BASM의 작동 방식

어셈블리 언어 자체가 어렵지, 델파이 애플리케이션 안에 어셈블리 코드를 집어넣는 것은 사실 매우 간단하다. 너무 간단해서 문제일 정도이다. asm 키워드를 써준 다음

어셈블리 코드를 쓰고 end로 마감하면 끝이다. 다음은 인라인 어셈블리 코드를 오브젝트 파스칼 코드에 포함시킨 예이다.

```
var
  i: Integer;
begin
  i := 0;
  asm
      mov eax, i
      inc eax
      mov i, eax
  end;
  { i가 1 증가했음. }
```

위의 코드는 오브젝트 파스칼로 변수 i를 선언한 다음 0으로 초기화한다. 그런 다음, i의 값을 eax 레지스터에 옮기고, eax 레지스터를 1 증가시킨 다음, 다시 eax 레지스터의 값을 i에 넣는다. 위의 코드에서처럼 어셈블리 코드를 델파이 코드에 집어넣는 것도 쉬울 뿐만 아니라, 오브젝트 파스칼의 변수를 어셈블리 코드에서 참조하는 것도 매우 쉽다.

간단한 인자 다루기

전역 변수나 프로시저의 전역 변수뿐만 아니라, 프로시저에 넘어온 인자를 어셈블리 코드로 다루는 것도 매우 간단하다. 다음이 그 예이다.

```
procedure Foo(I: Integer);
begin
  { 파스칼 코드를 여기에 }
  asm
      mov eax, I
      inc eax
      mov I, eax
  end;
  { I가 1 증가했음. }
  { 그리고 다른 파스칼 코드 }
end;
```

순수 어셈블리 프로시저에서는 스택 기준 포인터(ebp) 레지스터를 통해서 프로시저의 인자를 다뤄야 하지만, 위에서처럼 델파이의 인라인 어셈블리 코드에서는 인자의 이름을 직접 사용했다. 순수 어셈블리 언어의 프로시저라면 I의 값을 [ebp+4], 즉 스택의 기준 포인터를 이용해서 얻어야 했을 것이다.



프로시저에 넘어온 인자를 BASM으로 얻을 때 ebp 레지스터를 사용할 필요가 없다. 인자 이름으로 인자의 값을 얻을 수 있다는 점을 기억하기 바란다. ebp 레지스터와 오프셋을 이용하면 코드를 읽기가 더욱 어려워진다.

참조 전달 방식의 var 인자

함수를 선언할 때 인자를 var로 선언하면 그 인자는 값이 아니라 참조, 즉 포인터가 전달된다. BASM 코드에서 그런 인자를 다룰 때에는 그 인자가 값이 아니라 32비트 짜리 포인터라는 점을 고려해야 한다. 앞에서 나온 프로시저를 var 인자를 받아들이도록 수정했을 때 어셈블리 코드는 다음과 같은 형태가 된다.

```
procedure Foo(var I: Integer);
begin
  { 파스칼 코드를 여기에 }
  asm
      mov eax, I
      inc dword ptr [eax]
  end;
  { I가 1 증가했음. }
  { 그리고 다른 파스칼 코드 }
end;
```

레지스터 호출 규약

오브젝트 파스칼 함수와 프로시저의 기본 호출 규약은 레지스터(register)이다. 컴파일을 최적화하기 위해서 레지스터 호출 규약을 쓴다. 레지스터 호출 규약에 따르면 인자들 중 처음 세 개의 32비트 인자들은 각각 eax, edx, ecx 레지스터에 들어간다. 다음 함수 선언을 보자.

```
function BlahBlah(I1, I2, I3: Integer): Integer;
```

이 함수를 호출하면 I1의 값은 eax에 들어가고, I2는 edx에, 그리고 I3은 ecx에 들어간다. 또 다른 예를 보자.

```
procedure TSomeObject.SomeProc(S1, S2: PChar);
```

이 경우, S1은 ecx에, S2는 edx에 들어가며, eax에는 묵시적으로 Self 인자가 들어간다.

어셈블리로만 된 프로시저

프로시저나 함수를 시작할 때 begin 대신 asm을 쓰면 어셈블리 코드로만 된 프로시저나 함수를 만들 수 있다. 다음은 어셈블리 코드로만 이루어진 함수의 예이다.

```
function IncAnInt(I: Integer): Integer;
asm
  mov eax, I
  inc eax
end;
```



델파이 1의 assembler 지시자는 이제 필요 없다. 델파이 5의 컴파일러는 assembler 지시자를 그냥 무시해 버린다.

이 함수는 정수 인자 I를 받아서 1을 증가시킨다. 레지스터 호출 규약에서 Integer형의 리턴값은 eax 레지스터이므로 제대로 된 리턴값을 돌려준다. 표 13.1은 델파이의 함수가 돌려주는 리턴값의 데이터형이 어떻게 해석되는지를 나타낸 것이다.



표 13.1 델파이 함수의 리턴값이 전달되는 방식

리턴값의 데이터형	전달 방식
Char, Byte	al 레지스터
SmallInt, Word	ax 레지스터
Integer, LongWord, AnsiString	eax 레지스터
Pointer, class	eax 레지스터
Real48	스택의 데이터를 가리키는 포인터가 eax에 들어간다.
Int64	edx:eax 레지스터 쌍
Single, Double, Extended, Comp	8087 레지스터 스택의 ST(0)



ShortString 리턴값은 스택의 임시 문자열을 가리키는 포인터로 전달된다.

레코드

BASM은 오브젝트 파스칼의 레코드도 인식할 수 있다. 레지스터.레코드형.필드 형식으로 레코드의 필드를 참조하면 된다. 다음과 같은 레코드가 있다고 하자.

```
type
  TDumbRec = record
      i: Integer;
      c: Char;
  end;
```

다음은 TDumbRec형의 인자를 받아들이는 어셈블리 프로시저이다.

```
procedure ManipulateRec(var DR: TDumbRec);
asm
  mov [eax].TDumbRec.i, 24
  mov [eax].TDumbRec.c, 'c'
end;
```

인자로 받은 레코드를 가리키는 포인터가 바로 eax 레지스터이기 때문에 위와 같은 코드가 되었다. 마침표를 통해서 레코드의 필드를 직접 참조할 수 있다는 점을 주목하기 바란다. 물론, 직접 오프셋을 계산해도 되겠지만, 위의 코드처럼 하는 것이 작성하기도 쉽고, 나중에 이해하기도 쉽다.

윈도우즈 훅의 활용

윈도우즈 훅이란, 특정 윈도우에 도착하는 메시지만을 가로채는 것이 아니라, 윈도우즈 전체의 메시지를 가로채는 것을 말한다. 윈도우즈 훅을 이용하면 시스템이 보내는 모든 메시지와 이벤트를 가로챌 수 있으므로 윈도우즈를 거의 완전히 장악할 수 있다고 해도 과언이 아니다.

훅의 설치

윈도우즈에 훅을 설치할 때에는 API 함수 SetWindowsHookEx()를 사용한다.

```
function SetWindowsHookEx(idHook: Integer, lpfn: TFNHookProc; hmod: HINST;
dwThreadID: DWORD): HHOOK; stdcall;
```



32비트 애플리케이션에서는 SetWindowsHook()이 아니라 Ex가 붙은 SetWindowsHookEx()를 사용해야 한다는 점을 주의하기 바란다. SetWindowsHook()은 윈도우즈 3.1에 존재하던 것으로, Win32 API에는 구현되어 있지 않다.

idHook 인자는 설치할 훅의 종류를 뜻한다. 이 인자에 사용할 수 있는 값들이 표 13.2에 나와 있다.



표 13.2 윈도우즈 훅의 종류

훅 상수	의미
WH_CALLWNDPROC	윈도우 프로시저 필터. 메시지를 윈도우 프로시저에 보낼 때마다 훅 프로시저가 실행된다.
WH_CALLWNDPROCRET	대상 윈도우가 메시지를 처리한 다음에 훅 프로시저가 실행된다.
WH_CBT	컴퓨터 기반 트레이닝 필터. 대부분의 윈도우 관리, 마우스, 키보드 메시지를 처리하기 전에 훅 프로시저가 호출된다.
WH_DEBUG	디버깅 필터. 훅 프로시저는 다른 윈도우즈 훅보다 먼저 실행된다.
WH_GETMESSAGE	메시지 필터. 애플리케이션의 메시지 큐에서 메시지를 꺼낼 때 훅 프로시저가 실행된다.
WH_HARDWARE	하드웨어 메시지 필터. 애플리케이션의 메시지 큐에서 하드웨어 메시지를 꺼낼 때 훅 프로시저가 실행된다.
WH_JOURNALPLAYBACK	시스템 큐에서 메시지를 꺼낼 때마다 훅 프로시저가 실행된다. 주로 시스템 이벤트를 큐에 삽입할 때 사용한다.
WH_JOURNALRECORD	시스템 큐로부터 이벤트가 요청될 때마다 훅 프로시저가 실행된다. 보통, 시스템 이벤트를 "기록"할 때 사용한다.
WH_KEYBOARD	키보드 필터. 애플리케이션의 큐에서 wm_KeyDown이나 wm_KeyUp 메시지를 꺼낼 때마다 훅 프로시저가 실행된다.
WH_MOUSE	마우스 메시지 필터. 애플리케이션의 큐에서 마우스 메시지를 꺼낼 때마다 훅 프로시저가 실행된다.
WH_MSGFILTER	특별한 메시지 필터. 애플리케이션의 대화상자나 메뉴, 메시지 상자가 메시지를 처리하려고 할 때 훅 프로시저가 실행된다.
WH_SHELL	쉘 애플리케이션 필터. 최상위 윈도우가 생성되거나 파괴될 때, 그리고 쉘 애플리케이션이 활성화되려고 할 때 훅 프로시저가 실행된다.
WH_SYSMSGFILTER	시스템 메시지 필터. WH_MSGFILTER와 같지만, 시스템 전반에 대해 작동한다.

lpfn 인자는 윈도우즈 훅 프로시저로 사용할 콜백 함수의 주소이다. 그 함수의 형은 TFNHookProc이어야 하는데, TFNHookProc의 정의는 다음과 같다.

```
TFNHookProc = function (code: Integer; wparam: WPARAM; lparam: LPARAM):
LRESULT stdcall;
```

훅 함수의 인자들의 의미는 설치된 훅의 종류에 따라 다르다. 자세한 내용은 Win32 API 도움말을 참고하기 바란다.

hMod 인자는 훅 프로시저가 있는 모듈의 핸들로, 설치할 훅 콜백 함수를 담고 있는 EXE나 DLL의 hInstance값을 넣어야 한다.

dwThreadID 인자는 훅이 연결될 스레드를 지정하는 것으로, 이 값이 0이면 훅은 모든 스레드들에 연결된다.

SetWindowsHookEx()의 리턴값은 이전 훅의 핸들이며, 나중에 사용해야 하므로 전역 변수에 담아두어야 한다.

윈도우즈에는 하나 이상의 훅을 설치할 수 있으며, 동일한 종류의 훅도 하나 이상 설치할 수 있다.

또한, 어떤 훅들은 오직 DLL로만 구현해야 한다는 점도 기억하기 바란다. 그런 훅들에 대해서는 Win32 API 도움말이나 문서를 참고하기 바란다.



훅 DLL의 인스턴스는 각 프로세스들의 주소 공간에 개별적으로 맵핑된다. 따라서, 훅 DLL은 자신을 설치한 호스트 애플리케이션과 직접 통신할 수 없다. 메시지나 공유 메모리 영역(12장, "파일 다루기"에서 설명한 메모리 맵 파일 등)을 통해서 정보를 주고 받을 수밖에 없다.

훅 함수의 사용

훅 함수의 Code, wParam, lParam 인자의 값은 설치된 훅의 종류에 따라 다르다. 그에 대한 자세한 내용은 Win32 API 도움말을 참고하기 바란다. 그러나 단 한 가지 공통점이 있는데, 인자의 값이 어떻든지 간에 윈도우즈에 현재 설치된 다음 훅에게 인자들을 넘겨줘야 한다는 것이다.

다음 훅을 호출할 때에는 API 함수 CallNextHookEx()를 사용한다.

```
Result := CallNextHookEx(HookHandle, Code, wParam, lParam);
```



다음 훅을 호출할 때 DefHookProc()를 사용하면 안된다. 이 역시 Win32에는 구현되어 있지 않은 윈도우즈 3.x만의 함수이다.

훅의 제거

윈도우즈 훅을 제거할 때에는 다음과 같이 UnhookWindowsHookEx()를 사용한다.

```
UnhookWindowsHookEx(HookHandle);
```

JournalPlayback 훅을 이용한 SendKeys() 구현

비주얼 베이직이나 Pardox for Windows로 개발한 경험이 있는 독자라면 SendKeys()라는 함수를 알 것이다. SendKey()는 활성화된 윈도우에 대해 키보드 입력을 흉내내는 함수이다. 즉, SendKeys()로 보낸 문자열은 마치 사용자가 키보드로 입력한 것과 같은 효과를 낸다. 델파이에는 그에 해당하는 함수나 기능이 없기 때문에 한 번 만들어보았다. 이제, 이 책이 제공하는 SendKeys() 함수에 대해 알아보자. SendKeys()의 기능을 구현하려면 윈도우즈에 wh_JournalPlayBack 훅을 설치해야 한다.

SendKey()의 용도

어떤 키 입력을 발생시키려고 한다면 wm_KeyDown이나 wm_KeyUp 메시지만으로도 충분하다고 생각하는 독자도 있을 것이다. 그러나 그 메시지를 보낼 윈도우의 핸들을 모른다면 메시지를 보낼 수 없다. 반면, SendKeys()는 실제로 키보드의 키들이 눌린 것과 동일한 사건들을 발생시키므로 핸들과는 상관이 없다. 또한, 어떤 애플리케이션들은 키보드의 상태를 직접 점검해서 정보를 얻기도 한다. SendKeys()는 그런 경우까지도 처리할 수 있다.

SendKeys()의 작동 방식

SendKeys() 함수의 선언은 다음과 같다.

```
function SendKeys(S: String): TSendKeyError; export;
```

SendKeys()의 리턴값은 에러 조건을 뜻하는 열거형이다. 이 열거형에는 다음과 같은 값들이 있다.

값	의 미
sk_None	함수가 성공적으로 수행됨.
sk_FailSetHook	윈도우즈 훅을 설치할 수 없음.
sk_InvalidToken	문자열 안에 잘못된 토큰(token)이 있음.
sk_UnknownError	알지 못하는 심각한 에러가 발생했음.
sk_AlreadyPlaying	훅이 이미 설치되어 있으며, 키 입력이 진행중임.

S에 들어갈 문자열 중 키보드의 일반 영문자와 소문자는 그대로 넣어서 사용하면 되고, <Alt>키는 @, <Ctrl>키는 ^, <Shift>키는 ~로 표시한다. 기타 특수키나 기능키들은 중괄호({}) 안에 키 이름을 지정하면 되는데, 키 이름들은 목록 13.6과 같이 KeyDefs.pas 유닛에 정의되어 있다.



Ch13\SendKeys\KeyDefs.pas

```
unit Keydefs;

interface

uses Windows;

const
  MaxKeys = 24;
  ControlKey = '^';
  AltKey = '@';
  ShiftKey = '~';
  KeyGroupOpen = '{';
  KeyGroupClose = '}';

type
```

```
  TKeyString = String[7];

  TKeyDef = record
    Key: TKeyString;
    vkCode: Byte;
  end;

const
  KeyDefArray : array[1..MaxKeys] of TKeyDef = (
    (Key: 'F1';     vkCode: vk_F1),
    (Key: 'F2';     vkCode: vk_F2),
    (Key: 'F3';     vkCode: vk_F3),
    (Key: 'F4';     vkCode: vk_F4),
    (Key: 'F5';     vkCode: vk_F5),
    (Key: 'F6';     vkCode: vk_F6),
    (Key: 'F7';     vkCode: vk_F7),
    (Key: 'F8';     vkCode: vk_F8),
    (Key: 'F9';     vkCode: vk_F9),
    (Key: 'F10';    vkCode: vk_F10),
    (Key: 'F11';    vkCode: vk_F11),
    (Key: 'F12';    vkCode: vk_F12),
    (Key: 'INSERT'; vkCode: vk_Insert),
    (Key: 'DELETE'; vkCode: vk_Delete),
    (Key: 'HOME';   vkCode: vk_Home),
    (Key: 'END';    vkCode: vk_End),
    (Key: 'PGUP';   vkCode: vk_Prior),
    (Key: 'PGDN';   vkCode: vk_Next),
    (Key: 'TAB';    vkCode: vk_Tab),
    (Key: 'ENTER';  vkCode: vk_Return),
    (Key: 'BKSP';   vkCode: vk_Back),
    (Key: 'PRTSC';  vkCode: vk_SnapShot),
    (Key: 'SHIFT';  vkCode: vk_Shift),
    (Key: 'ESCAPE'; vkCode: vk_Escape));


function FindKeyInArray(Key: TKeyString; var Code: Byte): Boolean;

implementation
```

```
uses SysUtils;

function FindKeyInArray(Key: TKeyString; var Code: Byte): Boolean;
{ Key로 넘겨 받은 토큰에 해당하는 키를 배열에서 찾아서 그 키의 가상 키 코드를
 참조 전달 방식인 Code 인자로 돌려준다. }
var
  i: word;
begin
  Result := False;
  for i := Low(KeyDefArray) to High(KeyDefArray) do
    if UpperCase(Key) = KeyDefArray[i].Key then begin
      Code := KeyDefArray[i].vkCode;
      Result := True;
      Break;
    end;
end;

end.
```

SendKey()는 넘겨 받은 문자열을 개별적인 키 입력들로 분석한 다음, 각각의 키 입력을 wm_KeyUp과 wm_KeyDown 메시지가 포함된 메시지 구조체로 만든다. 그 메시지들을 wh_HournalPlayBack 훅을 통해서 윈도우즈에 차례로 재현하면 키 입력들이 재현된다.

키 입력 만들기

문자열에서 개별키 입력들을 뽑아낸 후 키 눌림/떨어짐 메시지(wm_KeyUp/wm_KeyDown 또는 wm_SysKeyUp/wm_SysKeyDown)와 가상키 코드를 MakeMessage()라는 프로시저에 보내서 메시지 구조체를 만들고, 만들어진 메시지 구조체를 MessageList라는 메시지 목록에 추가한다. 여기서 사용하는 메시지 구조체, 즉 레코드는 표준 TMessage 레코드와는 다르며, 5장에서 말한 TMsg 레코드와도 다르다. 이 메시지 레코드는 TEventMsg라는 것으로, 시스템 큐의 메시지를 뜻하는 것이다. 이 레코드의 정의는 다음과 같다.

```
type
  { 저널링에 사용하는 메시지 구조체 }
  PEventMsg = ^TEventMsg;
  TEventMsg = packed record
    message: UINT;
    paramL: UINT;
    paramH: UINT;
    time: DWORD;
    hwnd: HWND;
  end;
```

표 13.3은 TEventMsg의 필드들에 대한 설명이다.



표 13.3 TEventMsg의 필드들

TEventMsg 필드	값
message	메시지 상수. 키보드 메시지일 때에는 wm_(Sys)KeyUp이나 wm(Sys)KeyDown이고, 마우스 메시지일 때에는 wm_xBuutonUp이나 wm_xButtonDown, 또는 wm_MouseMove이다.
paramL	message가 키보드 메시지인 경우, 이 필드에 가상키 코드가 들어간다. message가 마우스 메시지인 경우, 이 필드에는 마우스 커서의 x좌표(화면 단위)가 들어간다.
paramH	message가 키보드 메시지인 경우, 이 필드에 키의 스캔 코드가 들어간다. message가 마우스 메시지인 경우, 이 필드에는 마우스 커서의 y좌표가 들어간다.
time	메시지가 발생한 시간(시스템 시간 단위)
hwnd	메시지를 보낼 윈도우의 핸들. wh_JournalPlayback 훅에서는 이 필드를 사용하지 않는다.

keyDefs 유닛에는 가상키 코드만 정의되어 있으므로 가상키 코드를 스캔 코드로 변환해야 하는데, 다행히 Win32 API 중에 그런 작업을 하는 MapVirtualKey()라는 함수가 있다. 다음은 MakeMessage() 프로시저의 소스 코드이다.

```
procedure MakeMessage(vKey: byte; M: Cardinal);
{ 이 프로시저는 키 입력을 흉내내는 TEventMsg 레코드를 만들고, 그것을
  메시지 목록에 추가한다. }
var
  E: PEventMsg;
begin
  New(E);              // 메시지 레코드의 메모리를 할당
  with E^ do begin
    message := M;      // message 필드를 설정
    paramL := vKey;   // paranH에 가상키 코드를 넣는다.
    paramH := MapVirtualKey(vKey, 0);  // ParamH에 스캔 코드를 넣는다.
    time := GetTickCount;              // time을 설정한다.
    hwnd := 0;                          // 필요 없음.
  end;
  MessageList.Add(E);
end;
```

SendKey()가 받은 문자열을 모두 메시지로 변환해서 목록을 완성한 다음에는 키 입력을 재현한다. 키 입력을 재현하는 프로시저는 StartPlayback()이다. StartPlayback()은 우선 메시지 목록의 첫 번째 메시지를 전역 버퍼에 넣는다. 그리고 몇 번째 메시지까지 재현했는지를 뜻하는 전역 변수와 <Ctrl>, <Alt>, <Shift>키의 상태를 뜻하는 플래그들을 설정한다. 그런 다음, 훅을 설정한다. StartPlayback()의 코드는 다음과 같다.

```
procedure StartPlayback;
{ 전역 변수들을 초기화하고 훅을 설치한다. }
begin
  { hc_Skip 전에 hc_GetNext를 받을 수 있으므로 우선 메시지 목록의 첫 번째
      메시지를 버퍼에 넣는다. }
  MessageBuffer := TEventMsg(MessageList.Items[0]^);
  { 메시지 카운터를 초기화한다. }
  MsgCount := 0;
  { <Alt>, <Ctrl>, <Shift>키의 상태를 나타내는 플래그 변수들을 초기화 }
  AltPressed := False;
  ControlPressed := False;
  ShiftPressed := False;
```

```
  { 훅을 설치한다. }
  HookHandle := SetWindowsHookEx(wh_JournalPlayback, Play, hInstance, 0);
  if HookHandle = 0 then
    raise ESKSetHookError.Create('Couldn''t set hook')
  else
    Playing := True;
end;
```

SetWindowsHookEx()의 두 번째 인자가 훅 함수이다. 이 프로시저는 Play라는 함수를 훅 함수로 설치한다. Play()의 선언은 다음과 같다.

```
function Play(Code: integer; wParam, lParam: Longint): Longint; stdcall;
```

다음은 이 함수의 인자들에 대한 설명이다.

인자	값
Code	hc_GetNext는 목록의 다음 메시지를 준비하라는 뜻이다. 목록에서 다음 메시지를 전역 버퍼에 넣으면 된다. hc_Skip은 다음 메시지를 가리키는 포인터를 lParam 인자에 넣으라는 뜻이다. 그 이외의 값이면 CallNextHookEx()를 호출해서 인자들을 다음 훅에 넘겨야 한다.
wParam	사용하지 않음.
lParam	Code가 hc_Skip인 경우, 다음 메시지(TEventMsg 레코드)를 가리키는 포인터를 lParam에 넣는다.
리턴값	Code가 hc_GetNext이면 0을 돌려준다. Code가 hc_Skip이면 메시지가 처리되기까지의 제한 시간(시스템 틱 단위)을 돌려준다. 코드가 그 이외의 값이면 CallNextHookEx()의 리턴값을 돌려줘야 한다.

목록 13.7은 SendKey.pas 유닛의 전체 소스 코드이다.

목록 13.7 Ch13\Sendkeys\SendKey.pas

```
unit SendKey;

interface

uses
```

```
 SysUtils, Windows, Messages, Classes, KeyDefs;

type
  { 에러 코드 }
  TSendKeyError = (sk_None, sk_FailSetHook, sk_InvalidToken,
      sk_UnknownError, sk_AlreadyPlaying);
  { 가상키 코드들을 원소로 집합형 }
  TvkKeySet = set of vk_LButton..vk_Scroll;

  { 예외 정의 }
  ESendKeyError = class(Exception);
  ESKSetHookError = class(ESendKeyError);
  ESKInvalidToken = class(ESendKeyError);
  ESKAlreadyPlaying = class(ESendKeyError);

function SendKeys(S: String): TSendKeyError;
procedure WaitForHook;
procedure StopPlayback;

var
  Playing: Boolean = False;

implementation

uses Forms;

type
  { TList의 상속 객체. 소멸자 Destroy를 재정의한다. }
  TMessageList = class(TList)
  public
    destructor Destroy; override;
  end;

const
  { 유효한 키들 }
  vkKeySet: TvkKeySet = [Ord('A')..Ord('Z'), vk_Menu, vk_F1..vk_F12];

destructor TMessageList.Destroy;
var
```

```
  i: longint;
begin
 { 목록 자체를 파괴하기 전에 모든 메시지 레코드들을 해제해야 한다. }
  for i := 0 to Count - 1 do
    Dispose(PEventMsg(Items[i]));
  inherited Destroy;
end;

var
  { DLL의 전역 변수들 }
  MsgCount: word = 0;
  MessageBuffer: TEventMsg;
  HookHandle: hHook = 0;
  MessageList: TMessageList = Nil;
  AltPressed, ControlPressed, ShiftPressed: Boolean;

procedure StopPlayback;
{ 훅을 제거하고 메모리를 해제한다. }
begin
  { 훅이 살아 있으면 제거한다. }
  if Playing then
    UnhookWindowsHookEx(HookHandle);
  MessageList.Free;
  Playing := False;
end;

function Play(Code: integer; wParam, lParam: Longint): Longint; stdcall;
{ JournalPlayback 훅의 콜백 함수. 하드웨어 이벤트가 발생하면 윈도우즈는 이
함수를 호출한다. Code 인자의 값에 따라 적절한 처리를 한다. }
begin
  case Code of

    hc_Skip: begin
      { hc_Skip은 목록에서 다음 메시지를 꺼내라는 뜻이다. 마지막 메시지까지
          처리했으면 JournalPlayback 훅을 제거하는 메소드를 호출한다. 훅 함수
안에서 훅을 제거하면 안된다. }
      { 메시지 카운터를 증가시킨다. }
      inc(MsgCount);
      { 모든 메시지들을 처리했는지 검사한다. }
```

```
      if MsgCount >= MessageList.Count then
        StopPlayback
      else
      { 목록의 다음 메시지를 버퍼에 넣는다. }
      MessageBuffer := TEventMsg(MessageList.Items[MsgCount]^);
      Result := 0;
    end;

    hc_GetNext: begin
      { hc_GetNext는 메시지를 재현하는 데 필요한 값을 wParam과 lParam에
        넣으라는 뜻이다. 리턴값은 윈도우즈가 메시지를 재현하기까지 소비할 제한
        시간을 의미한다. 즉시 처리하도록 0을 돌려준다. }
      { 버퍼의 메시지를 메시지 큐에 넣는다. }
      PEventMsg(lParam)^ := MessageBuffer;
      Result := 0  { process immediately }
    end

    else
      { Code가 hc_Skip이나 hc_GetNext가 아니면 다음 훅에게 인자들을 넘긴다. }
      Result := CallNextHookEx(HookHandle, Code, wParam, lParam);
  end;
end;

procedure StartPlayback;
{ 전역 변수들을 초기화하고 훅을 설치한다. }
begin
 { hc_Skip 전에 hc_GetNext를 받을 수 있으므로 우선 메시지 목록의 첫 번째
   메시지를 버퍼에 넣는다. }
  MessageBuffer := TEventMsg(MessageList.Items[0]^);
 { 메시지 카운터를 초기화한다. }
  MsgCount := 0;
 { <Alt>, <Ctrl>, <Shift>키의 상태를 나타내는 플래그 변수들을 초기화 }
  AltPressed := False;
  ControlPressed := False;
  ShiftPressed := False;
  { 훅을 설치한다. }
  HookHandle := SetWindowsHookEx(wh_JournalPlayback, Play, hInstance, 0);
  if HookHandle = 0 then
    raise ESKSetHookError.Create('훅을 잡을 수 없습니다.')
```

```
  else
    Playing := True;
end;

procedure MakeMessage(vKey: byte; M: Cardinal);
{ 이 프로시저는 키 입력을 흉내내는 TEventMsg 레코드를 만들고 그것을
 메시지 목록에 추가한다. }
var
  E: PEventMsg;
begin
  New(E);                        // 메시지 레코드의 메모리를 할당
  with E^ do begin
    message := M;               // message 필드를 설정
    paramL := vKey;            // paranH에 가상키 코드를 넣는다.
    paramH := MapVirtualKey(vKey, 0);  // ParamH에 스캔 코드를 넣는다.
    time := GetTickCount;               // time을 설정한다.
    hwnd := 0;                          // 필요 없음.
  end;
  MessageList.Add(E);
end;

procedure KeyDown(vKey: byte);
{ KeyDown 메시지를 만든다. }
begin
  { <Ctrl>키가 눌린 경우에는 "Sys"키 메시지를 발생시키지 말아야 한다
     (윈도우즈의 이상한 점이다). }
  if AltPressed and (not ControlPressed) and  (vKey in vkKeySet) then
    MakeMessage(vKey, wm_SysKeyDown)
  else
    MakeMessage(vKey, wm_KeyDown);
end;

procedure KeyUp(vKey: byte);
{ KeyUp 메시지를 만든다. }
begin
  { Ctrl 키가 눌린 경우에는 "Sys" 키 메시지를 발생시키지 말아야 한다.
     (윈도우즈의 이상한 점이다) }
  if AltPressed and (not ControlPressed) and (vKey in vkKeySet) then
    MakeMessage(vKey, wm_SysKeyUp)
```

```
  else
    MakeMessage(vKey, wm_KeyUp);
end;

procedure SimKeyPresses(VKeyCode: Word);
{ 이 함수는 인자로 지정된 키의 키 입력(키 눌림/뗌)을 재현한다. <Alt>, <Ctrl>,
<Shift>키의 눌림도 함께 처리한다. }
begin
  { 플래그에 따라 <Alt>키를 누른다. }
  if AltPressed then
    KeyDown(vk_Menu);
  { 플래그에 따라 <Ctrl>키를 누른다. }
  if ControlPressed then
    KeyDown(vk_Control);
  { <Shift>키를 눌러야 하는지 검사 }
  if (((Hi(VKeyCode) and 1) <> 0) and (not ControlPressed)) or ShiftPressed
     then KeyDown(vk_Shift);    { <Shift>키를 누른다. }
  KeyDown(Lo(VKeyCode));  { 인자로 지정된 키를 누른다. }
  KeyUp(Lo(VKeyCode));    { 키를 뗀다. }
  { <Shift>키가 눌렸는지 검사 }
  if (((Hi(VKeyCode) and 1) <> 0) and (not ControlPressed)) or ShiftPressed
then
    KeyUp(vk_Shift);      { <Shift>키를 뗀다. }
  { Shift 플래그가 True이면 False로 바꾼다. }
  if ShiftPressed then begin
    ShiftPressed := False;
  end;
  { <Ctrl>키가 눌렸으면 뗀 후 플래그를 False로 한다. }
  if ControlPressed then begin
    KeyUp(vk_Control);
    ControlPressed := False;
  end;
  { <Alt>키가 눌렸으면 뗀 후 플래그를 False로 한다. }
  if AltPressed then begin
    KeyUp(vk_Menu);
    AltPressed := False;
  end;
end;
```

```
procedure ProcessKey(S: String);
{ 이 프로시저는 문자열 S의 각 문자들을 키 입력 메시지로 만들어서 목록에
추가한다. }
var
  KeyCode: word;
  Key: byte;
  index: integer;
  Token: TKeyString;
begin
  index := 1;
  repeat
    case S[index] of

      KeyGroupOpen : begin
      { 특수키 토큰임. }
        Token := '';
        inc(index);
        while S[index] <> KeyGroupClose do begin
          { 토큰의 끝()을 만날 때까지 토큰에 문자를 추가한다. }
          Token := Token + S[index];
          inc(index);
          { 토큰의 길이가 7자 이상이면 잘못된 토큰이다. }
          if (Length(Token) = 7) and (S[index] <> KeyGroupClose) then
            raise ESKInvalidToken.Create('닫는 괄호가 없습니다.');
        end;
              { 특수키가 정의된 배열에서 토큰을 찾는다. 찾으면 Key 인자에
                  그 키의 가상키 코드가 들어간다. }
        if not FindKeyInArray(Token, Key) then
          raise ESKInvalidToken.Create('잘못된 토큰');
        { 키 입력을 재현한다. }
        SimKeyPresses(MakeWord(Key, 0));
      end;

      AltKey : begin
        { <Alt>키 플래그를 True로 한다. }
        AltPressed := True;
      end;

      ControlKey : begin
```

```
        { <Ctrl>키 플래그를 True로 한다. }
        ControlPressed := True;
      end;

      ShiftKey : begin
        { <Shift>키 플래그를 True로 한다. }
        ShiftPressed := True;
      end;

      else begin
      { 영문자나 소문자 등 일반키인 경우 }
              { 문자를 워드로 변환한다. 이때 상위 바이트에는 <Shift>키의
                상태를 넣고, 하위 바이트에는 그 문자에 해당하는 가상키 코드를
                넣는다. }
        KeyCode := vkKeyScan(S[index]);
        { 키 입력을 재현한다. }
        SimKeyPresses(KeyCode);
      end;
    end;
    inc(index);
  until index > Length(S);
end;

procedure WaitForHook;
begin
  repeat Application.ProcessMessages until not Playing;
end;

function SendKeys(S: String): TSendKeyError;
{ 개발자가 호출할 실제 함수이다. S로 받은 문자열을 해석해서 키 눌림/키 떨어짐
 메시지들의 목록을 만들고, JournalPlayback 훅을 설치한 후 키 입력 메시지들
 을 재현한다. }
begin
  Result := sk_None;         // 성공이라고 가정
  try
    if Playing then raise ESKAlreadyPlaying.Create('');
    MessageList := TMessageList.Create;  // 메시지 목록을 생성
    ProcessKey(S);           // 문자열을 메시지 목록으로 변환
    StartPlayback;           // 훅을 설치하고 키 입력을 재현
```

```
  except
    { 예외가 발생하면 메시지 목록을 해제하고 에러 코드를 돌려준다. }
    on E:ESendKeyError do begin
      MessageList.Free;
      if E is ESKSetHookError then
        Result := sk_FailSetHook
      else if E is ESKInvalidToken then
        Result := sk_InvalidToken
      else if E is ESKAlreadyPlaying then
        Result := sk_AlreadyPlaying;
    end
    else
      { 나머지 모든 예외들을 처리 }
      Result := sk_UnknownError;
  end;
end;

end.
```

SendKeys()의 활용 예제

그럼, SendKeys() 함수를 어떻게 써먹는지를 보여주는 TestSend라는 예제를 소개하겠다. 이 예제의 메인 폼에는 두 개의 TEdit 컴포넌트와 네 개의 TButton 컴포넌트가 있다(그림 13.4).

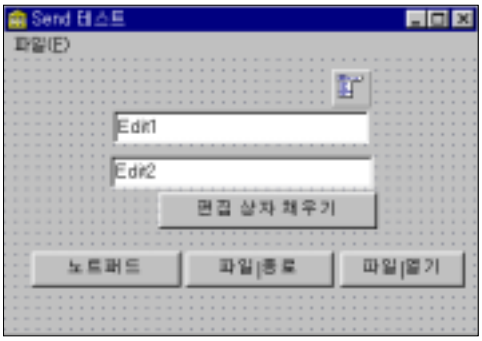


그림 13.4
TestSend 예제의 메인 폼

목록 13.8은 TestSend의 메인 폼 유닛인 Main.pas의 소스 코드이다.

목록 13.8 Main.pas의 소스 코드

```
unit Main;

interface

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

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    Button1: TButton;
    Button2: TButton;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Open1: TMenuItem;
    Exit1: TMenuItem;
    Button4: TButton;
    Button3: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Open1Click(Sender: TObject);
    procedure Exit1Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;

implementation

{$R *.DFM}

uses SendKey, KeyDefs;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.SetFocus;                          // Edit1에 포커스를 설정
  SendKeys('^{DELETE}I Love....');      // Edit1에 키들을 보낸다.
  WaitForHook;                             // 키 입력 재현을 준비
  Perform(wm_NextDlgCtl, 0, 0);       // Edit2로 옮긴다.
  SendKeys('Delphi 5 '); // Edit2에 키들을 보낸다.
end;

procedure TForm1.Button2Click(Sender: TObject);
var
  H: hWnd;
  PI: TProcessInformation;
  SI: TStartupInfo;
begin
  FillChar(SI, SizeOf(SI), 0);
  SI.cb := SizeOf(SI);
  { Invoke notepad }
  if CreateProcess(nil, 'notepad', nil, nil, False, 0, nil, nil, SI, PI)
 then
  begin
    { 메모장이 키 입력을 받을 준비가 될 때까지 기다린다. }
    WaitForInputIdle(PI.hProcess, INFINITE);
    H := FindWindow('Notepad', nil);     // 메모장의 윈도우를 찾는다.
    if SetForegroundWindow(H) then       // 메모장을 맨 앞으로
SendKeys(' Hello from SendKeys Example!{ENTER}');  // 키들을 보낸다.
  end
  else
    MessageDlg(Format('Notepad 호출 에러.  에러 코드 %d',
              [GetLastError]), mtError, [mbOk], 0);
```

```
end;
procedure TForm1.Open1Click(Sender: TObject);
begin
  ShowMessage('Open');
end;

procedure TForm1.Exit1Click(Sender: TObject);
begin
  Close;
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  WaitForInputIdle(GetCurrentProcess, INFINITE);
  SendKeys('@fx');
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  WaitForHook;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  WaitForInputIdle(GetCurrentProcess, INFINITE);
  SendKeys('@fo');
end;

 end.
```

Button1(File Edit)를 클릭하면 SendKeys()를 통해 다음과 같은 일을 수행한다. 우선, <Ctrl+D>키로 Edti1의 내용을 지운 다음 "I love..."를 Edit1에 입력하고, 탭 문자를 입력한다. 그리고 Edit2로 포커스를 옮긴 후 역시 SendKeys()를 통해서 <Shift+D>, "Delphi 5"를 Edit2에 입력한다.

Buttion2의 OnClick 이벤트 처리 프로시저 역시 주목할 만하다. 이 프로시저는 CreateProcess API 함수를 이용해서 윈도우즈 메모장(NotePad)의 인스턴스를 띄운다.

그런 다음, WaitForInputIdle() API 함수를 호출해서 메모장이 키 입력을 받아들일 때까지 프로그램 실행을 지연시킨다. 그리고 SendKeys()를 이용해서 메모장에 텍스트를 입력한다.

C/C++ OBJ 파일 활용

델파이의 컴파일러와 링커는 다른 컴파일러로 만들어낸 표준 OBJ 파일을 델파이 프로그램에 직접 링크시킬 수 있다. OBJ 파일을 링크시키려면 오브젝트 파스칼 코드 안에 다음과 같이 $L이나 $LINK 지시자로 OBJ 파일의 이름을 지정해야 한다.

```
{$L filename.obj}
```

OBJ 파일 안에 있는 함수를 호출하려면 external 지시자를 이용해서 그 함수를 정의해 줘야 한다. 델파이 컴파일러는 external 지시자를 만나면 컴파일 시점이 아니라 링크 시점에서 그 함수의 이름을 해석한다. 다음은 OBJ 파일의 FOO라는 인자도 리턴 값도 없는 프로시저를 정의하는 예이다.

```
procedure FOO; external;
```

그러나 OBJ 파일을 사용할 때에는 몇 가지 제한 사항이 따른다. 다음이 제한 사항들이다.

- 오브젝트 파스칼은 OBJ 파일에 담긴 코드만을 링크할 수 있으며, 데이터는 직접 사용하지 못한다(잠시 후에 OBJ 안의 데이터를 얻는 방법을 설명하겠다). 그러나 OBJ 파일에서 파스칼 데이터를 사용할 수는 있다.
- 오브젝트 파스칼은 LIB(정적 라이브러리) 파일을 링크할 수 없다.
- C++ RTL의 불명확한 참조 구조 때문에 C++ 클래스를 담고 있는 OBJ 파일은 링크할 수 없다. C++ RTL을 OBJ에서 분리시켜서 참조 문제를 해결할 수는 있지만, 노력만큼의 성과를 얻기 힘들다.
- OBJ 파일은 인텔 OMF 형식이어야 한다. 볼랜드 C++는 인텔 OMF 형식의 OBJ 파일을 만들지만, 마이크로소프트 C++ 컴파일러는 COFF 형식의 OBJ 파일을 만든다.



델파이 3 컴파일러부터 OBJ 대 OBJ 참조가 가능해졌다. 그 이전의 델파이에서는 브젝트 파스칼이 링크할 OBJ 파일이 다른 OBJ 파일에 있는 코드를 참조해서는 안 되었지만, 델파이 3 이후부터는 그러한 제약이 없어졌다.

함수의 호출

예를 들어서, ccode.obj라는 C++ OBJ 파일에 다음과 같은 선언의 함수가 있다고 하자.

```
int __fastcall SAYHELLO(char * hellostr)
```

델파이 안에서 위의 함수를 호출하려면 우선 다음과 같이 $L이나 $LINK 지시자로 OBJ 파일을 EXE에 링크하도록 지시해야 한다.

```
{$L ccode.obj}
```

그런 다음, C++ 함수를 오브젝트 파스칼의 문법에 맞게 정의한다.

```
function SayHello(Text: PChar): Integer; external;
```



C++의 __fastcall 지시자는 오브젝트 파스칼의 기본 호출 규약인 register에 해당하는 것이다. C++ 함수를 오브젝트 파스칼에서 사용할 때 호출 규약을 정확히 지정하지 않으면 에러가 난다. 호출 규약은 인자나 리턴값, 함수 주소의 처리 방식을 뜻하는 것이므로 반드시 일치시켜야 한다. 다음은 오브젝트 파스칼의 호출 규약에 해당하는 C++의 호출 규약들을 정리한 것이다.

오브젝트 파스칼	C++
register*	__fastcall
pascall	__pascal
cdecl	__cdecl*
stdcall	__stdcall

*는 그 언어의 기본 호출 규약이다.

네임 맹글링

기본적으로, C++ 컴파일러는 extern "C" 수정자를 이용해서 함수 이름을 명시적으로 선언하지 않으면 함수의 이름을 나름대로의 규칙에 따라 변환한다. 반면, 오브젝트 파스칼 컴파일러는 함수 이름에 손을 대지 않는다. 예를 들어서, 델파이의 TDUMP 유틸리티로 ccode.obj 안에 들어 있는 SAYHELLO 함수의 심벌 이름을 보면 @SAYHELLO $qqrpc로 되어 있다. 그런데 오브젝트 파스칼이 도입하는 함수 이름은 SAYHELLO이다(오브젝트 파스칼은 심벌을 모두 대문자로 만든다).

두 이름이 다른데도 오브젝트 파스칼이 성공적으로 함수를 링크시킬 수 있는 이유는 델파이 링커가 변환된 부분(@와 $ 이후의 모든 글자)을 그냥 무시하기 때문이다. 그러나 "그냥 무시하기" 때문에 문제가 생길 수도 있다.

C++가 함수 이름을 변환하는 이유는 함수 중복(overloading, 동일한 이름의 함수가 서로 다른 인자 구성을 가질 수 있는 객체 지향적 기능)을 지원하기 위한 것이다. 만일 여러 번 중복 정의된 함수들 중 어떤 하나를 델파이로 불러서 쓴다면 델파이가 링크한 함수가 정확히 그 함수인지는 보장할 수 없다. 따라서, 중복 정의된 함수를 OBJ 파일로부터 링크하는 것은 좋지 않다.



C++ 소스 파일(.CPP)의 함수들은 extern "C" 수정자로 선언하지 않는 한 이름이 변환된다. 컴파일러의 명령행 인자를 적절히 지정하면 이름 변환을 하지 못하게 할 수도 있다.

데이터의 공유

앞에서도 말했지만, OBJ 파일에서 델파이 데이터를 사용하는 것은 가능하다. 즉, C++ 코드에서 델파이의 데이터를 참조할 수 있다는 뜻이다. 우선, 공유할 전역 변수를 오브젝트 파스칼 안에서 선언한다(밑줄이 중요하다).

```
var
  _GLOBALVAR: PChar = '이것은 델파이 문자열';
```

변수를 꼭 초기화해야 하는 것은 아니다. 단지, 예를 들기 위한 것이다.

C++ 모듈 안에서는 extern 수정자를 이용해서 동일한 이름의 변수를 선언한 다음 사용하면 된다.

```
extern char * GLOBALVAR;
```



기본적으로, 볼랜드 C++ 컴파일러는 외부 심벌을 만들어낼 때 밑줄을 붙인다(즉, GLOBALVAR는 _GLOBALVAR가 된다). 이 문제를 해결하려면 다음 중 한 가지 방법을 사용하면 된다.

- 컴파일러 실행시 명령행 인자를 이용해서 밑줄을 아예 붙이지 못하게 한다(볼랜드 C++ 컴파일러의 경우, -u-).
- 아니면, 오브젝트 파스칼에서 아예 밑줄을 붙여서 변수를 선언한다.

반대로 OBJ, 즉 C++에서 선언된 전역 데이터를 오브젝트 파스칼에서 직접 사용하는 것은 불가능하지만, 해결책이 없는 것은 아니다. 우선, C++ 코드에서 전역 데이터를 선언할 때 앞에 __export 지시자를 붙여서 외부에 공개하도록 해야 한다. 예를 들어서, char 배열을 공유하려면 다음과 같이 한다.

```
char __export C_VAR[128];
```

오브젝트 파스칼쪽에서는 external 지시자를 써주며, 변수가 아닌 프로시저로 선언한다(역시 이름 앞에 밑줄을 붙인다).

```
procedure _C_VAR; external; // OBJ 데이터를 참조하기 위한 트릭
```

이렇게 하면 일단 _C_VAR를 가리키는 포인터는 얻을 수 있게 된다. 이제 프로시저로 선언된 _C_VAR를 char의 배열로 다룰 수 있게 만들어야 한다. 다음은 배열의 값을 얻는 방법을 보여주는 예이다.

```
type
  PCharArray = ^TCharArray;
  TCharArray = array[0..127] of char;

function GetCArray: string;
var
  A: PCharArray;
begin
  A := PCharArray(@_C_VAR);
  Result := A^;
end;
```

그리고 다음 코드는 배열의 값을 설정하는 예이다.

```
procedure SetCArray(const S: string);
var
  A: PCharArray;
begin
  A := PCharArray(@_C_VAR);
  StrLCopy(A^, PChar(S), SizeOf(TCharArray));
end;
```

델파이 RTL을 통한 C++ RTL 참조 처리

C++ RTL을 참조하는 코드가 담긴 OBJ 파일을 델파이에 링크하는 것은 간단한 문제가 아니다. C++ RTL은 보통 LIB 파일에 들어 있으며, 델파이는 LIB 파일을 링크하지 못하기 때문이다.

가장 간단하면서도 어려운 해결책은 RTL LIB 파일을 만들어낸 소스 코드에서 OBJ 파일이 사용하는 함수들의 정의를 복사해서 OBJ 파일의 소스 코드에 집어넣는 것이지만, 사실 그런 방법은 생각하기도 싫다. 소스 코드를 얻을 수 있다고 해도, 함수들의 참조 관계를 따지려면 너무 복잡할 뿐만 아니라, OBJ 파일의 크기도 너무 커지기 때문이다.

좀더 똑똑한 방법은, OBJ 파일이 사용하는 RTL 함수들의 선언만 따로 헤더 파일(.h나 .hpp)로 만든 다음, 그 함수들을 오브젝트 파스칼 코드 안에서 실제로 구현하는 것이다. 예를 들어서, C++ 코드에서 API 함수 MessageBox()를 호출한다고 하자. 그러자면 코드에 #include window.h를 포함시켜야 하며, OBJ 파일을 만들 때 MessageBox()가 구현되어 있는 Win32 라이브러리를 링크해야 한다. 그러나 다음과 같이 C++ 코드에서 MessageBox()를 다시 정의한다고 하자.

```
extern "C" int __stdcall MessageBox(long, char *, char *, long);
```

이렇게 하면 델파이 프로젝트를 컴파일해서 실행 파일을 만들 때 오브젝트 파스칼 링커는 C++ RTL, 즉, Win32 라이브러리가 아닌 델파이의 라이브러리에서 MessageBox()를 찾아서 링크한다. 즉 Windows.pas 유닛에 정의된 함수를 링크하는 것이다. 이렇게 하면 C++ RTL을 사용하는 OBJ 파일도 델파이로 링크할 수 있다.

목록 13.9는 지금까지 이야기한 기법을 보여주는 간단한 C++ 모듈이다.

목록 13.9 Ch13\CallC\ccode.cpp

```
#include "PasStng.h"

// 전역 변수
extern char * GLOBALVAR;

// 공개할 데이터
char __export C_VAR[128];
```

```
// 외부 함수
extern "C" int __stdcall MessageBox(long, char *, char *, long);

// 함수 선언
int  __fastcall SAYHELLO(char *);
int  __fastcall SAYHELLO(int);

// 함수 정의
int  __fastcall SAYHELLO(char * hellostr)
{
  char a[64];
  memset(a, 64, 0);
  strcat(a, hellostr);
  strcat(a, " -> Borland C++builder4");
  MessageBox(0, a, GLOBALVAR, 0);
  return 0;
}

int  __fastcall SAYHELLO(int)
{
  return 0;
 }
```

MessageBox() 같은 Win32 API 함수뿐만 아니라, C++ 표준 라이브러리 함수들도 LIB 파일에 들어 있다. PasStng.h는 string.h에 선언된 문자열/메모리 관련 함수들 중 자주 사용하는 것들을 모아둔 헤더 파일이다. 목록 13.10이 PasStng.h의 코드이다.

목록 13.10 Ch13\CallC\PasStng.h

```
// PasStng.h
// 이 모듈은 C++ RTL에 포함된 문자열/메모리 관련 함수들을 오브젝트 파스칼이
// 처리할 수 있도록 만들기 위해서 string.h의 일부 함수들을 공개(extern)
// 시킨 헤더 파일임.

#ifndef PASSTNG_H
#define PASSTNG_H
```

```
#ifndef _SIZE_T
#define _SIZE_T
typedef unsigned size_t;
#endif

#ifdef __cplusplus
extern "C" {
#endif

extern char * __cdecl strcat(char *dest, const char *src);
extern int __cdecl stricmp(const char *s1, const char *s2);
extern size_t __cdecl strlen(const char *s);
extern char * __cdecl strlwr(char *s);
extern char * __cdecl strncat(char *dest, const char *src, size_t maxlen);
extern void * __cdecl memcpy(void *dest, const void *src, size_t n);
extern int __cdecl strncmp(const char *s1, const char *s2, size_t  maxlen);
extern int __cdecl strncmpi(const char *s1, const char *s2, size_t n);
extern void * __cdecl memmove(void *dest, const void *src, size_t n);
extern char * __cdecl strncpy(char *dest, const char *src, size_t maxlen);
extern void * __cdecl memset(void *s, int c, size_t n);
extern int __cdecl strnicmp(const char *s1, const char *s2, size_t maxlen);
extern void __cdecl movmem(const void *src, void *dest, unsigned length);
extern void __cdecl setmem(void *dest, unsigned length, char value);
extern char * __cdecl stpcpy(char *dest, const char *src);
extern int __cdecl strcmp(const char *s1, const char *s2);
extern char * __cdecl strstr(char *s1, const char *s2);
extern int __cdecl strcmpi(const char *s1, const char *s2);
extern char * __cdecl strupr(char *s);
extern char * __cdecl strcpy(char *dest, const char *src);

#ifdef __cplusplus
}
#endif

#endif  // PASSTNG_H
```

MessageBox() 같은 Win32 API와는 달리, 이런 함수들은 오브젝트 파스칼 RTL에 구현되어 있지 않으므로 위의 함수들을 오브젝트 파스칼 코드로 직접 구현해야 한다. 목록 13.11은 위의 함수들을 오브젝트 파스칼로 구현한 PasStrng.pas의 소스 코드이다.

목록 13.11 Ch13\CallC\PasStrng.pas

```
unit PasStrng;
{ 이 유닛은 C++ RTL인 String.h에서 자주 사용되는 함수들을 }
{ 오브젝트 파스칼로 구현한 것임. }
{ Copyright (c) 1997 Steve Teixeira  }

interface

uses Windows;

function _strcat(Dest, Source: PChar): PChar; cdecl;
procedure _memset(P: Pointer; Count: Integer; value: DWORD); cdecl;
function _stricmp(P1, P2: PChar): Integer; cdecl;
function _strlen(P1: PChar): Integer; cdecl;
function _strlwr(P1: PChar): PChar; cdecl;
function _strncat(Dest, Source: PChar; MaxLen: Integer): PChar; cdecl;
function _memcpy(Dest, Source: Pointer; Len: Integer): Pointer;
function _strncmp(P1, P2: PChar; MaxLen: Integer): Integer; cdecl;
function _strncmpi(P1, P2: PChar; MaxLen: Integer): Integer; cdecl;
function _memmove(Dest, Source: Pointer; Len: Integer): Pointer;
function _strncpy(Dest, Source: PChar; MaxLen: Integer): PChar; cdecl;
function _strnicmp(P1, P2: PChar; MaxLen: Integer): Integer; cdecl;
procedure _movmem(Source, Dest: Pointer; MaxLen: Integer); cdecl;
procedure _setmem(Dest: Pointer; Len: Integer; Value: Char); cdecl;
function _stpcpy(Dest, Source: PChar): PChar; cdecl;
function _strcmp(P1, P2: PChar): Integer; cdecl;
function _strstr(P1, P2: PChar): PChar; cdecl;
function _strcmpi(P1, P2: PChar): Integer; cdecl;
function _strupr(P: PChar): PChar; cdecl;
function _strcpy(Dest, Source: PChar): PChar; cdecl;

implementation
```

```
uses SysUtils;

function _strcat(Dest, Source: PChar): PChar;
begin
  Result := SysUtils.StrCat(Dest, Source);
end;

function _stricmp(P1, P2: PChar): Integer;
begin
  Result := StrIComp(P1, P2);
end;

function _strlen(P1: PChar): Integer;
begin
  Result := SysUtils.StrLen(P1);
end;

function _strlwr(P1: PChar): PChar;
begin
  Result := StrLower(P1);
end;

function _strncat(Dest, Source: PChar; MaxLen: Integer): PChar;
begin
  Result := StrLCat(Dest, Source, MaxLen);
end;

function _memcpy(Dest, Source: Pointer; Len: Integer): Pointer;
begin
  Move(Source^, Dest^, Len);
  Result := Dest;
end;

function _strncmp(P1, P2: PChar; MaxLen: Integer): Integer;
begin
  Result := StrLComp(P1, P2, MaxLen);
end;

function _strncmpi(P1, P2: PChar; MaxLen: Integer): Integer;
```

```
begin
  Result := StrLIComp(P1, P2, MaxLen);
end;

function _memmove(Dest, Source: Pointer; Len: Integer): Pointer;
begin
  Move(Source^, Dest^, Len);
  Result := Dest;
end;

function _strncpy(Dest, Source: PChar; MaxLen: Integer): PChar;
begin
  Result := StrLCopy(Dest, Source, MaxLen);
end;

procedure _memset(P: Pointer; Count: Integer; Value: DWORD);
begin
  FillChar(P^, Count, Value);
end;

function _strnicmp(P1, P2: PChar; MaxLen: Integer): Integer;
begin
  Result := StrLIComp(P1, P2, MaxLen);
end;

procedure _movmem(Source, Dest: Pointer; MaxLen: Integer);
begin
  Move(Source^, Dest^, MaxLen);
end;

procedure _setmem(Dest: Pointer; Len: Integer; Value: Char);
begin
  FillChar(Dest^, Len, Value);
end;

function _stpcpy(Dest, Source: PChar): PChar;
begin
  Result := StrCopy(Dest, Source);
end;
```

```
function _strcmp(P1, P2: PChar): Integer;
begin
  Result := StrComp(P1, P2);
end;

function _strstr(P1, P2: PChar): PChar;
begin
  Result := StrPos(P1, P2);
end;

function _strcmpi(P1, P2: PChar): Integer;
begin
  Result := StrIComp(P1, P2);
end;

function _strupr(P: PChar): PChar;
begin
  Result := StrUpper(P);
end;

function _strcpy(Dest, Source: PChar): PChar;
begin
  Result := StrCopy(Dest, Source);
end;

 end.
```



이 기법의 핵심은 OBJ 안의 코드가 실제 C++ RTL 대신, 오브젝트 파스칼의 RTL이나 코드를 참조하도록 만드는 것이다. Win32 API라면 거의 모두 Windows.pas 유닛에 정의되어 있으므로 쉽게 적용할 수 있을 것이다.

C++ 클래스 활용법

OBJ 파일 안에 담긴 C++ 클래스를 델파이에서 사용하는 것은 불가능하다. 그러나 DLL에 담긴 C++ 클래스는 어느 정도 사용할 수 있다. "어느 정도"라고 한 이유는, 델파이에서 사용할 수 있는 것은 C++ 클래스가 공개하는 가상 메소드뿐이기 때문이다. 오브젝트 파스칼과 C++ 모두 가상 인터페이스에 대해 COM 규약을 따르기 때문에 가상 메소드라면 공유할 수 있다(COM에 대해서는 23장, "COM과 ActiveX"를 볼 것).

목록 13.12는 클래스 정의가 담긴 C++ 모듈 cdll.cpp의 소스 코드이다. 멤버 함수가 아닌 두 함수, 즉 CLASS로 시작하는 두 함수를 주목하기 바란다. 하나는 새 객체를 만들고 그 객체에 대한 참조를 돌려주는 것이며, 또 하나는 인자로 받은 참조를 해제하는 함수이다. 오브젝트 파스칼에서 이 두 함수를 통해 객체를 생성하거나 해제한다.

목록 13.12 Ch13\CallC\cdll.cpp

```
#include <windows.h>
#include <vcl.h>
#pragma hdrstop
int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void*)
{
        return 1;
}

// 객체
class TFoo
{
  virtual int function1(char *);
  virtual int function2(int);
};

// 멤버 함수
int TFoo::function1(char * str1)
{
  MessageBox(NULL, str1, "Hello from C++ DLL", MB_OK);
  return 0;
}
```

```
int TFoo::function2(int i)
{
  return i * i;
}

#ifdef __cplusplus
extern "C"  {
#endif

// 함수 선언
TFoo * __declspec(dllexport) ClassFactory(void);
void __declspec(dllexport) ClassKill(TFoo *);

TFoo * __declspec(dllexport) CLASSFACTORY(void)
{
  TFoo * Foo;
  Foo = new TFoo;
  return Foo;
}

void __declspec(dllexport) CLASSKILL(TFoo * Foo)
{
  delete Foo;
}


#ifdef __cplusplus
}
 #endif
```

델파이에서 이 객체를 사용하려면 두 가지 일을 해줘야 한다. 우선, 클래스 인스턴스를 생성하고 파괴하는 C++의 함수를 도입(import)해야 한다. 그리고 C++ 클래스를 캡슐화하는 가상 추상(virtual abstract) 오브젝트 파스칼 클래스를 정의해야 한다. 다음은 위의 C++ 클래스에 대한 오브젝트 파스칼의 래퍼(wrapper) 클래스이다.

```
type
  TFoo = class
       function Function1(Str1: PChar): Integer; virtual; cdecl; abstract;
       function Function2(i: Integer): Integer; virtual; cdecl; abstract;
  end;
function ClassFactory: TFoo; cdecl; external 'cdll.dll' name
'_CLASSFACTORY';
function ClassKill(Foo: TFoo); cdecl; external 'cdll.dll' name
'_CLASSKILL';
```



C++ 클래스의 오브젝트 파스칼 래퍼를 정의할 때 함수의 이름에 대해서는 걱정할 필요가 없다. 함수가 내부적으로 호출되는 과정에서는 이름이 중요하지 않기 때문이다. 모든 호출은 내부적으로 가상 메소드 테이블(Virtual Method Table, VMT)을 통해서 처리된다. 따라서, 정작 중요한 것은 함수가 선언되는 순서이다. C++ 클래스에서 함수들이 선언된 순서를 오브젝트 파스칼에서도 그대로 지키는 것이 핵심이다.

목록 13.13은 지금까지 나온 모든 C++ 관련 기법들을 보여주는 예제인 CallC.dpr 프로젝트의 메인 폼 유닛 Main.pas의 소스 코드이다. 메인 폼의 모습은 그림 13.5에 나와 있다.



Ch13\CallC\Main.pas

```
unit Main;

interface

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

```
type
  TMainForm = class(TForm)
    Button1: TButton;
    Button2: TButton;
    FooData: TEdit;
    Button3: TButton;
    Button4: TButton;
    SetCVarData: TEdit;
    GetCVarData: TEdit;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
  private
    { Private 선언부 }
  public
    { Public 선언부 }
  end;

var
  MainForm: TMainForm;
  _GlobalVar: PChar = '이것은 Delphi 문자열입니다.';

implementation

uses PasStrng;

{$R *.DFM}

{$L ccode.obj}

type
  TFoo = class
    function Function1(Str1: PChar): integer; virtual; cdecl; abstract;
    function Function2(i: integer): integer; virtual; cdecl; abstract;
  end;

  PCharArray = ^TCharArray;
  TCharArray = array[0..127] of char;
```

```
// OBJ 파일에서 도입하는 루틴들
function SayHello(Text: PChar): integer; external;
procedure SetCVar(Value: Integer); external;
procedure _C_VAR; external;  // OBJ 데이터를 공유하기 위한 기법

// DLL 파일에서 도입하는 루틴들
function ClassFactory: TFoo; cdecl; external 'cdll.dll' name
'_CLASSFACTORY';
procedure ClassKill(Foo: TFoo); cdecl; external 'cdll.dll' name
'_CLASSKILL';

procedure TMainForm.Button1Click(Sender: TObject);
begin
  SayHello('안녕 Delphi5');
end;

procedure TMainForm.Button2Click(Sender: TObject);
var
  Foo: TFoo;
begin
  Foo := ClassFactory;
  Foo.Function1('최고의 툴.');
  FooData.Text := IntToStr(Foo.Function2(10));
  ClassKill(Foo);
end;

function GetCArray: string;
var
  A: PCharArray;
begin
  A := PCharArray(@_C_VAR);
  Result := A^;
end;

procedure SetCArray(const S: string);
var
  A: PCharArray;
begin
  A := PCharArray(@_C_VAR);
```

```
  StrLCopy(A^, PChar(S), SizeOf(TCharArray));
end;

procedure TMainForm.Button3Click(Sender: TObject);
begin
  SetCArray(SetCVarData.Text);
end;

procedure TMainForm.Button4Click(Sender: TObject);
begin
  GetCVarData.Text := GetCArray;
end;

end.
```

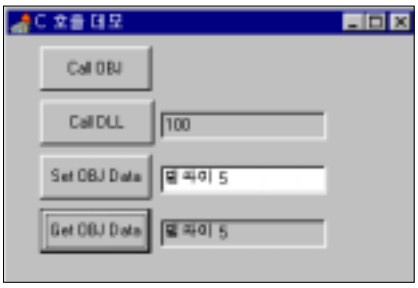


그림 13.5
CallC 프로젝트의 메인 폼



C++ 코드/클래스의 활용에 대해 여기까지 설명한 내용은 어찌 보면 아주 제한적인 기법일 뿐이다. 좀더 본격적으로 C++와 델파이를 통합시키려면 COM 객체를 사용하는 것이 좋다. COM에 대해서는 23장을 참고하기 바란다.

썽킹에 대해

이전에 주로 사용하던 16비트 DLL이나 라이브러리가 있는데, 그것을 32비트로 전환하기가 힘들다면 32비트 애플리케이션에서 16비트 코드를 호출하는 방법을 찾아야 한다. 그처럼 32비트 애플리케이션에서 16비트 코드를 호출하거나, 아니면 16비트 애플리케이션에서 32비트 코드를 호출하는 과정을 썽킹(thunking)이라고 부른다.

사실, 32비트의 세계와 16비트의 세계는 CPU 차원에서 다른 세계라고 할 수 있으며, 두 세계를 넘나든다는 것은 전혀 자연스럽지 못한 일이다. 그래서 Win32 프로그래밍의 여러 분야 중 가장 어려운 부분을 썽킹이라고 하는 개발자들도 많이 있다.



16/32비트의 경계를 넘나들 때 썽킹 이외에도 자동화(automation, 23장에 나옴)를 이용하는 방법도 있다. 그러한 일을 가능하게 하는 것은 자동화의 IDispatch 인터페이스이다.

Win32는 공통(universal), 일반(generic), 플랫(flat)이라는 세 종류의 썽킹을 제공한다. 각각의 썽킹은 나름대로의 장·단점을 가지고 있다.

- 공통 썽킹은 Win32s 플랫폼에서만 가능하다. 공통 썽킹은 16비트 애플리케이션이 Win32 DLL을 로드하고 호출할 수 있도록 만드는 것이다. 그러나 이러한 썽킹은 델파이가 공식적으로 지원하지는 않는 Win32s 플랫폼에서만 일어날 수 있기 때문에 이 책에서는 다루지 않겠다.
- 일반 썽킹은 윈도우즈 9x나 윈도우즈 NT에서 실행되는 16비트 애플리케이션이 Win32 DLL을 호출할 수 있도록 만드는 것이다. 이것은 Win32의 대표적인 운영체제인 윈도우즈 9x와 윈도우즈 NT에서 모두 사용할 수 있으며, API만으로 가능하기 때문에 가장 바람직하다. 델파이2가 나온 이후로 이 썽킹에 대해서는 많은 잡지와 많은 서적에서 다루었으므로 이 글에서는 다루지 않겠다.
- 플랫 썽킹은 Win32 애플리케이션이 16비트 DLL을 호출하거나, 16비트 애플리케이션이 Win32 DLL을 호출할 수 있도록 만드는 것이다. 그러나 플랫 썽킹은 윈도우즈 95에서만 가능하며, 32비트와 16비트 양쪽에 링크될 특별한 OBJ 파일을 생성하는 썽크(thunk) 컴파일러가 필요하다. 호환성도 떨어지고, 추가적인 도구도 있어야 하기 때문에 이 책에서는 플랫 썽킹을 다루지 않겠다.

그리고 WM_COPYDATA 윈도우즈 메시지를 이용해서 32비트와 16비트 프로세스 사이에 데이터를 공유하는 기법도 있다. 특히, 윈도우즈 NT의 경우에 WM_COPYDATA를 이용해서 상당히 직접적으로 16비트 코드에 접근할 수 있기 때문에 이번 장에서는 WM_COPYDATA 기법에 대해서 이야기하겠다.

WM__COPYDATA

윈도우즈 9x에서는 플랫 썽킹을 통해서 Win32 애플리케이션에서 16비트 DLL을 호출할 수 있다. 그러나 윈도우즈 NT에서는 Win32 애플리케이션에서 16비트 코드를 직접 호출하는 방법이 없다. 그렇다면 윈도우즈 NT에서 32비트와 16비트 프로세서 사이에 데이터를 공유하는 최고의 해결책은 무엇일까? 그리고 윈도우즈 NT와 윈도우즈 9x에서 모두 작동하는 16비트-32비트 데이터 공유 기법은 없는 것일까?

두 질문의 답은 모두 WM_COPYDATA이다. WM_COPYDATA는 프로세스들 사이에서 이진 데이터를 전송할 때 사용하는 것으로, 16비트와 32비트 프로세스 사이에서도 이진 데이터를 전송할 수 있다. 어떤 윈도우가 WM_COPYDATA 메시지를 받은 경우, 그 메시지의 wParam에는 데이터를 보낸 윈도우의 핸들이 들어 있고, lParam에는 TCopyDataStruct 레코드를 가리키는 포인터가 들어 있다. TCopyDataStruct 레코드의 정의는 다음과 같다.

```
type
  PCopyDataStruct = ^TCopyDataStruct;
  TCopyDataStruct = packed record
    dwData: DWORD;
    cbData: DWORD;
    lpData: Pointer;
  end;
```

dwData 필드에는 32비트짜리 사용자 정의 정보가 담겨 있다. cbData에는 lpData가 가리키는 버퍼의 크기가 담겨 있다. lpData는 전달한 데이터가 담긴 버퍼를 가리키는 포인터이다. 32비트와 16비트 사이에서 이 메시지를 주고 받은 경우, 윈도우즈는 lpData 포인터에 대해 자동으로 16비트 포인터(16:16)/32비트 포인터(0:32) 변환을 수행한다. 데이터를 받은 프로세스의 주소 공간에 lpData가 가리키는 데이터를 적절히 맵핑시키는 작업도 윈도우즈가 수행해 준다.

참고

작은 양의 정보만을 주고 받을 때에는 WM_COPYDATA가 적격이지만, 많은 양의 데이터를 16과 32비트 프로세스 사이에서 주고 받으려면 자동화를 사용하는 것이 좋다. 자동화에 대해서는 23장에서 설명한다.

Tip

WM_COPYDATA로 함수를 직접 호출할 수는 없다. 대신, 16비트 DLL의 함수 진입을 감싸는 16비트 애플리케이션을 만들고, 그 애플리케이션이 WM_COPYDATA를 이용해서 32비트 애플리케이션과 함수의 인자나 리턴값을 주고 받게 만들면 된다.

WM_COPYDATA의 사용법을 보여주기 위해 두 개의 프로젝트를 만들었는데, 첫 번째 것은 32비트 애플리케이션이다. 이 애플리케이션은 메인 폼의 메모 컨트롤에 입력된 텍스트를 두 번째 프로젝트인 16비트 애플리케이션에게 보낸다. 두 애플리케이션이 정보를 주고 받은 과정은 다음과 같다.

1. 윈도우 메시지 등록을 통해서 애플리케이션간의 정보 공유에 사용할 고유한 메시지 ID를 얻는다.
2. Win32 애플리케이션에서 그 메시지를 시스템 전체에 뿌린다(broadcast). 그 메시지의 wParam에는 Win32 애플리케이션의 메인 윈도우의 핸들이 들어 있다.
3. 뿌린 메시지를 받은 16비트 애플리케이션은 등록된 메시지를 Win32 애플리케이션에 보내는데, 이때 wParam에 자신의 메인 윈도우의 핸들을 넣어서 보낸다.
4. 돌아온 메시지를 받은 Win32 애플리케이션은 16비트 애플리케이션의 메인 폼의 핸들을 얻었으므로 그 핸들을 이용해서 16비트 애플리케이션에게 WM_COPYDATA 메시지와 메모 컨트롤의 텍스트를 보낸다.

목록 13.14는 두 프로젝트가 공유하는 RegMsg.pas 유닛의 소스 코드이다.

목록 13.14 Ch13\CopyData\RegMsg.pas

```
unit RegMsg;
interface
var
  DDGM_HandshakeMessage: Cardinal;
implementation
uses WinProcs;
const
  HandshakeMessageStr: PChar = 'DDG.CopyData.Handshake';
initialization
  DDGM_HandshakeMessage := RegisterWindowMessage(HandshakeMessageStr);
end.
```

32비트 프로젝트인 CopyData.dpr의 메인 폼 유닛인 CopyMain.pas의 소스 코드는 목록 13.15에 나와 있다. 이 유닛은 데이터의 공유를 시작하고, 데이터를 보내는 역할을 한다.

목록 13.15 Ch13\CopyData\CopyMain.pas

```
unit CopyMain;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls, Menus;
type
  TMainForm = class(TForm)
    DataMemo: TMemo;
    BottomPnl: TPanel;
    BtnPnl: TPanel;
    CloseBtn: TButton;
    CopyBtn: TButton;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    CopyData1: TMenuItem;
```

```
    N1: TMenuItem;
    Exit1: TMenuItem;
    Help1: TMenuItem;
    About1: TMenuItem;
    procedure CloseBtnClick(Sender: TObject);
    procedure FormResize(Sender: TObject);
    procedure About1Click(Sender: TObject);
    procedure CopyBtnClick(Sender: TObject);
  private
    { Private declarations }
  protected
    procedure WndProc(var Message: TMessage); override;
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;
implementation
{$R *.DFM}
uses AboutU, RegMsg;

// 다음의 선언은 Windows 유닛에 있는 BroadcastSystemMessage() 선언
// 에러 때문에 필요하다.
function BroadcastSystemMessage(Flags: DWORD; Recipients: PDWORD;
  uiMessage: UINT; wParam: WPARAM; lParam: LPARAM): Longint; stdcall;
  external 'user32.dll';

var
  Recipients: DWORD = BSM_APPLICATIONS;
procedure TMainForm.WndProc(var Message: TMessage);
var
  DataBuffer: TCopyDataStruct;
  Buf: PChar;
  BufSize: Integer;
begin
  if Message.Msg = DDGM_HandshakeMessage then begin
    { 버퍼를 할당한다. }
    BufSize := DataMemo.GetTextLen + (1 * SizeOf(Char));
```

```
    Buf := AllocMem(BufSize);
    { 메모 컴포넌트의 텍스트를 버퍼에 넣는다. }
    DataMemo.GetTextBuf(Buf, BufSize);
    try
      with DataBuffer do begin
        { 안전을 위해서 dwData에 등록된 메시지를 넣는다. }
        dwData := DDGM_HandshakeMessage;
        cbData := BufSize;
        lpData := Buf;
      end;
      { 주의 : WM_COPYDATA 메시지는 반드시 Send로 보내야 한다.}
      SendMessage(Message.wParam, WM_COPYDATA, Handle,
                                        Longint(@DataBuffer));
    finally
      FreeMem(Buf, BufSize);
    end;
  end
  else
    inherited WndProc(Message);
end;

procedure TMainForm.CloseBtnClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.FormResize(Sender: TObject);
begin
  BtnPnl.Left := BottomPnl.Width div 2 - BtnPnl.Width div 2;
end;

procedure TMainForm.About1Click(Sender: TObject);
begin
  AboutBox;
end;

procedure TMainForm.CopyBtnClick(Sender: TObject);
begin
  { 시스템 전체에 메시지를 뿌린다. }
```

```
  BroadcastSystemMessage(BSF_IGNORECURRENTTASK or BSF_POSTMESSAGE,
    @Recipients, DDGM_HandshakeMessage, Handle, 0);
end;

end.
```

목록 13.16은 16비트 프로젝트인 ReadData.dpr의 메인 폼 유닛인 ReadMain.pas의 소스 코드이다. 이 유닛은 CopyData 프로젝트의 메시지에 응답을 하며, CopyData 프로젝트가 보낸 데이터를 받는다.



Ch13\CopyData\ReadMain.pas

```
unit Readmain;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Menus, StdCtrls;
{ WM_COPYDATA 윈도우즈 메시지는 16비트의 Messages 유닛에 정의되어 있지는 않지만,
 윈도우즈 NT나 윈도우즈 95에서 실행되는 16비트 애플리케이션에서도 제대로
 작동한다. 이 메시지에 대한 자세한 내용은 Win32 API 도움말을 참고하기 바란다. }
const
  WM_COPYDATA = $004A;

type
  TMainForm = class(TForm)
    ReadMemo: TMemo;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Exit1: TMenuItem;
    Help1: TMenuItem;
    About1: TMenuItem;
    procedure Exit1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure About1Click(Sender: TObject);
  private
    procedure OnAppMessage(var M: TMsg; var Handled: Boolean);
    procedure WMCopyData(var M: TMessage); message WM_COPYDATA;
```

```
  end;
var
  MainForm: TMainForm;
implementation
{$R *.DFM}
uses RegMsg, AboutU;
type
 { TCopyDataStruct 레코드형은 WinTypes 유닛에 정의되어 있지 않으므로 직접
 정의해 줘야 한다. 물론, 윈도우즈 NT나 윈도우즈 95에서 실행되는 16비트 응용
 프로그램에서는 아무 문제 없이 작동한다. WM_COPYDATA 메시지의 lParam에는
 이 레코드에 대한 포인터가 들어간다. }
  PCopyDataStruct = ^TCopyDataStruct;
  TCopyDataStruct = record
    dwData: DWORD;
    cbData: DWORD;
    lpData: Pointer;
  end;

procedure TMainForm.OnAppMessage(var M: TMsg; var Handled: Boolean);
{ Application 객체의 OnMessage 이벤트 처리 프로시저 }
begin
 { DDGM_HandshakeMessage는 모든 애플리케이션들에게 뿌려진다. 이 메시지의
     wParam에는 메시지를 뿌린 윈도우의 핸들이 들어 있다. 이 메시지를 받으면 이
     메시지를 보낸 윈도우에게 다시 이 메시지를 보내는데, 이때 wParam에 자신의
     윈도우 핸들을 넣는다. }
  if M.Message = DDGM_HandshakeMessage then begin
    PostMessage(M.wParam, DDGM_HandshakeMessage, Handle, 0);
    Handled := True;
  end;
end;
procedure TMainForm.WMCopyData(var M: TMessage);
{ WM_COPYDATA 메시지를 처리하는 프로시저 }
begin
  { wParam을 검사해서 데이터를 교류할 32비트 애플리케이션이 맞는지 확인 }
  if PCopyDataStruct(M.lParam)^.dwData = DDGM_HandshakeMessage then
    { WM_COPYDATA 메시지이면 lParam이 가리키는 내용을 메모 컴포넌트에 넣는다.}
    ReadMemo.SetTextBuf(PChar(PCopyDataStruct(M.lParam)^.lpData));
end;
```

```
procedure TMainForm.Exit1Click(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  Application.OnMessage := OnAppMessage;
end;

procedure TMainForm.About1Click(Sender: TObject);
begin
  AboutBox;
end;

 end.
```

그림 13.6은 두 애플리케이션이 제대로 작동하고 있는 모습이다.

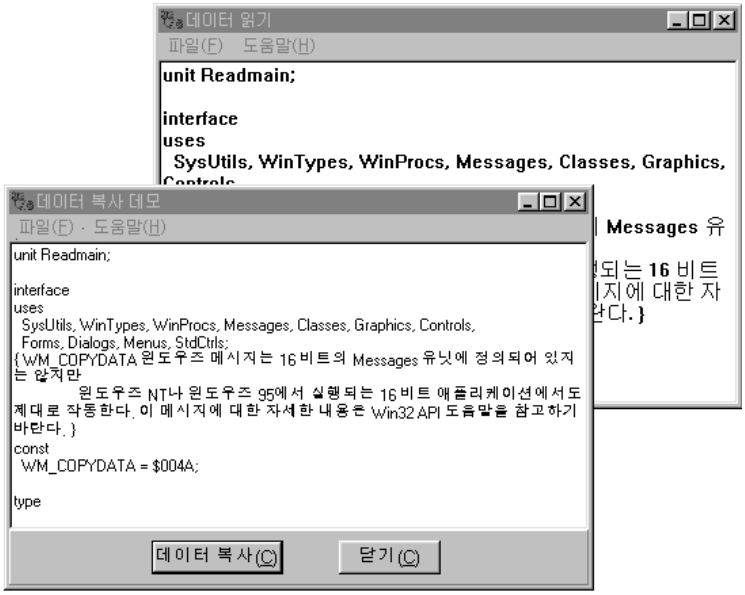


그림 13.6
WM_COPYDATA를 통한 데이터 공유

요약

이번 장은 정말 어려운 내용들을 모자라는 말솜씨로 힘겹게 이야기한 장이었다. 윈도우 프로시저의 서브클래싱, 중복 실행 방지, 윈도우즈 후킹, BASM 프로그래밍, C++의 OBJ 파일과 클래스 활용, 썽킹, 그리고 WM_COPYDATA에 이르기까지 그야말로 고수들의 기법들을 이야기했다. 잘 이해하지 못하는 내용이 나왔더라도, 일단은 그냥 넘어가기 바란다. 그러나 언젠가는 필요하게 될 내용들일 수도 있으므로 이번 장에 나온 개념과 제목들만이라도 머리 속에 담아두기 바란다. 이왕 어려운 내용들을 탐험한 김에, 다음 장에서는 운영체제 깊숙히 들어가서 프로세스나 스레드, 모듈에 대한 정보를 얻는 방법들을 알아보겠다.