# 24장 윈도우즈 쉘의 확장

* 시스템 트레이 다루기
* 애플리케이션 데스크탑 툴바
* 바로 가기 만들기
* 쉘 확장 기능

윈도우즈 95에 이르러서 윈도우즈 쉘(윈도우즈 운영체제의 포괄적인 사용자 인터페이스)은 윈도우즈 3.1에 비해 크게 변했으며, 윈도우즈 NT 3.51 이후의 윈도우즈 NT들이나 윈도우즈 98도 기본적으로 윈도우즈 95의 쉘을 따르고 있다. 특히, 윈도우즈 3.1의 프로그램 관리자에 비해 윈도우즈 95의 쉘은 사용자의 입맛에 맞게 조정할 수 있는 부분이 훨씬 많아졌다는 점을 강조하고 싶다. 다만, 윈도우즈 쉘의 막강한 기능에 비해 개발자에게 제공되는 정보는 상당히 제한적이라서 멋진 아이디어가 있다고 해도 그것을 구현하기는 상대적으로 어려운 현실이다. 그래서 이번 장에서는 시스템 트레이와 애플리케이션 데스크탑 툴바, 쉘 링크, 쉘 확장자 같은 윈도우즈 쉘의 여러 요소들을 다루는 기법에 대해서 설명해 보겠다.

# 시스템 트레이 다루기

그럼 윈도우즈의 시스템 트레이 아이콘을 캡슐화하는 델파이 컴포넌트를 만드는 것으로 이번 장을 시작해 보자. 이번에 만들 컴포넌트의 이름은 TTrayNotification으로 트레이 통지 아이콘을 생성하거나 트레이 통지 아이콘이 제대로 작동하는 데 필요한 여러 API 함수들을 델파이 컴포넌트에 캡슐화한 것이다. 트레이 통지 아이콘(tray-notification icon)이란, 윈도우즈의 작업 표시줄 오른쪽 끝에 있는 시스템 트레이 영역 안에 나타나는 작은 아이콘들을 말한다(그림 24.1).

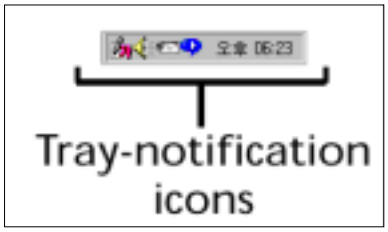



**그림 24.1**
트레이 통지 아이콘들

## 관련 API 함수

믿기 어려울 수도 있겠지만, 트레이 통지 아이콘을 추가하거나 수정하고 제거하는 모든 작업은 Shell_NotifyIcon()이라는 하나의 API 함수로 수행한다. 이 함수나 윈도우

즈 쉘에 관련된 다른 함수들은 모두 ShellAPI 유닛에 정의되어 있다. Shell_NotifyIcon()의 선언은 다음과 같다.

```
function Shell_NotifyIcon(dwMessage: DWORD; lpData:
      PNotifyIconData): BOOL; stdcall;
```

dwMessage 인자는 아이콘에 대해 어떤 작업을 수행할 것인지를 뜻한다. 이 인자에 넣을 수 있는 값은 표 24.1에 나와 있다.

표 24.1 dwMessage 인자의 값들

| 상수 | 값 | 의미 |
|---|---|---|
| NIM_ADD | 0 | 아이콘을 시스템 트레이에 추가한다. |
| NIM_MODIFY | 1 | 기존 아이콘의 속성을 수정한다. |
| NIM_DELETE | 2 | 시스템 트레이에서 아이콘을 제거한다. |

lpData 인자는 TNotifyIconData 레코드에 대한 포인터이다. 이 레코드의 정의는 다음과 같다.

```
type
  PNotifyIconDataA = ^TNotifyIconDataA;
  PNotifyIconData = PNotifyIconDataA;
  {$EXTERNALSYM _NOTIFYICONDATAA}
  _NOTIFYICONDATAA = record
    cbSize: DWORD;
    Wnd: HWND;
    uID: UINT;
    uFlags: UINT;
    uCallbackMessage: UINT;
    hIcon: HICON;
    szTip: array [0..63] of AnsiChar;
  end;
```

cbSize 필드에는 레코드의 크기를 넣는데, 레코드의 크기는 SizeOf(TNotifyIcodData)를 이용해서 얻어야 한다.

Wnd는 트레이 통지 아이콘의 "콜백" 메시지를 받을 윈도우의 핸들이다(콜백이란 말에 따옴표를 한 것은 엄밀한 의미에서의 콜백이 아니기 때문이다. 어쨌든 Win32 문서에서도 트레이 통지 아이콘이 윈도우에게 메시지를 보내는 것을 콜백이라는 용어로 표현하고 있다).

uID는 개발자가 부여하는 고유한 ID 번호이다. 하나 이상의 아이콘을 시스템 트레이에 등록하는 경우, 이 값을 이용해서 각각의 아이콘들을 식별하면 된다.

uFlags는 TNotifyIconData 레코드의 필드들 중 어떤 것을 Shell_NotifyIcon() 함수로 활성화할 것인지, 다시 말하면 dwMessage 인자로 지정된 작업을 아이콘의 어떤 속성에 적용할 것인지를 지정하는 인자이다. 표 24.2는 이 인자에 조합해서 넣을 수 있는 플래그들을 정리한 것이다.



표 24.2 uFlags에 조합해 넣을 수 있는 플래그들

| 상수 | 값 | 의미 |
|---|---|---|
| NIF_MESSAGE | 0 | uCallbackMessage 필드에 적용 |
| NIF_ICON | 2 | hIcon 필드에 적용 |
| NIF_TIP | 4 | szTip 필드에 적용 |

uCallbackMessage에는 Wnd 인자로 지정한 윈도우에 보내질 윈도우즈 메시지값을 넣는다. 일반적으로, 이 필드에는 RegisterWindowMessage()로 얻은 메시지 ID나 WM_USER를 이용해서 정의한 메시지 ID 상수를 넣는다. 이 메시지의 lParam에는 uID 필드에 지정한 값이 들어가며, wParam 인자에는 트레이 통지 아이콘에 대해 발생한 마우스 메시지가 들어가게 된다.

hIcon에는 시스템 트레이에 나타날 아이콘의 핸들을 넣는다.

szTip에는 마우스 포인터를 아이콘에 놓으면 나타나는 풍선 도움말을 뜻하는 널 종료 문자열을 넣는다.

이번 예제인 TTryaNotifyIcon 컴포넌트는 Shell_NotifyIcon() 함수를 SendTrayMessage()라는 메소드로 캡슐화해서 제공하는데, 이 메소드는 다음과 같다.

```
procedure TTrayNotifyIcon.SendTrayMessage(Msg: DWORD; Flags: UINT);
{ 이 메소드는 API 함수 Shell_NotifyIcon()을 호출한다. }
begin
  { 레코드를 적절한 값으로 채운다. }
  with Tnd do
  begin
    cbSize := SizeOf(Tnd);
    StrPLCopy(szTip, PChar(FHint), SizeOf(szTip));
    uFlags := Flags;
    uID := UINT(Self);
    Wnd := IconMgr.HWindow;
    uCallbackMessage := DDGM_TRAYICON;
    hIcon  := ActiveIconHandle;
  end;
  Shell_NotifyIcon(Msg, @Tnd);
end;
```

이 메소드에서 szTip에는 FHint라는 컴포넌트의 내부 저장 필드의 값을 넣는다.

uID에는 Self를 가리키는 참조를 넣는다. 이렇게 하면 통지 아이콘이 보낸 윈도우즈 메시지에서 그 아이콘을 등록한 컴포넌트를 쉽게 식별해 낼 수 있다.

Wnd에는 IconMgr.HWindow를 넣는데, IconMgr은 TIconManager형의 전역 변수이다. TIconMgr의 구현에 대해서는 잠시 후에 설명하겠다. 일단은 시스템 트레이가 보내는 모든 메시지들을 IconMgr의 윈도우가 일차적으로 처리한다는 점만 기억하기 바란다.

uCallbackMessage에는 DDGM_TRAYICON을 넣는다. DDGM_TRAYICON은 RegisterWindow Message() API 함수를 통해서 얻은 메시지 ID이다. RegisterWindowMessage()로 얻은 메시지 ID는 운영체제 전반에서 고유한 값을 가진다. 다음은 DDGM_TRAYICON을 얻는 데 관련된 코드이다.

```
const
  { 윈도우즈 메시지를 등록하는 데 필요한 문자열 }
  TrayMsgStr = 'DDG.TrayNotifyIconMsg';

initialization
  { 트레이 콜백에 사용할 고유한 윈도우즈 메시지 ID를 얻는다. }
  DDGM_TRAYICON := RegisterWindowMessage(TrayMsgStr);
```

hIcon에는 ActiveIconHandle() 메소드의 리턴값을 넣는다. 이 메소드는 컴포넌트의 Icon 속성이 가리키고 있는 아이콘의 핸들을 돌려주는 함수이다.

## 메시지 처리

이 컴포넌트에서 모든 시스템 통지 메시지들은 전역 객체인 IconMgr의 윈도우로 보내진다고 했었다. IconMgr 객체는 컴포넌트의 initialization부에서 생성되며, finalization부에서 해제된다. 다음과 같다.

```
initialization
  { 트레이 콜백에 사용할 고유한 윈도우즈 메시지 ID를 얻는다. }
  DDGM_TRAYICON := RegisterWindowMessage(TrayMsgStr);
  IconMgr := TIconManager.Create;
finalization
  IconMgr.Free;
```

IconMgr의 클래스인 TIconManager의 정의는 다음과 같다.

```
type
  TIconManager = class
  private
    FHWindow: HWnd;
    procedure TrayWndProc(var Message: TMessage);
  public
    constructor Create;
    destructor Destroy; override;
    property HWindow: HWnd read FHWindow write FHWindow;
  end;
```

시스템 트레이의 메시지들을 받을 윈도우는 이 컴포넌트의 생성자에서 AllocateHWnd() 함수를 통해서 만들어진다.

```
constructor TIconManager.Create;
begin
  FHWindow := AllocateHWnd(TrayWndProc);
end;
```

TrayWndProc() 메소드가 생성자에서 만들어진 윈도우의 윈도우 프로시저 역할을 하는데, 이 메소드에 대해서는 잠시 후에 다시 이야기하겠다.

## 아이콘과 풍선 도움말

컴포넌트 사용자가 아이콘과 풍선 도움말을 쉽게 설정할 수 있으려면 Icon이나 Hint 등의 이름을 가지는 속성 형태로 제공하는 것이 좋다. 그리고 Icon 속성을 TIcon형으로 선언하여 만들면 객체 조사기에서 대화상자 형태의 속성 편집기를 통해서 아이콘을 쉽게 설정할 수 있게 된다. 트레이 아이콘은 설계 시점뿐만 아니라 실행 시점에서도 변경할 수 있어야 한다. Icon 속성의 쓰기 메소드에 SendTrayMessage()를 이용해서 트레이 통지 아이콘의 아이콘 핸들을 바꾸는 코드(NIM_MODIFY 메시지를 사용한다)를 넣기만 하면 설계 시점에서든 실행 시점에서든 Icon 속성의 값을 바꾸는 것만으로 시스템 트레이에 나타나는 아이콘을 변경할 수 있게 된다. Hint 속성 역시 마찬가지 기법을 적용한다.

다음은 Icon과 Hint 속성의 쓰기(write) 메소드이다.

```
procedure TTrayNotifyIcon.SetIcon(Value: TIcon);
{ Icon 속성의 쓰기 메소드 }
begin
  FIcon.Assign(Value);  // 새 아이콘을 설정한다.
  { 시스템 트레이의 아이콘을 변경한다. }
  if FIconVisible then SendTrayMessage(NIM_MODIFY, NIF_ICON);
end;


procedure TTrayNotifyIcon.SetHint(Value: string);
{ Hint 속성의 쓰기 메소드 }
begin
  if FHint <> Value then
  begin
    FHint := Value;
    if FIconVisible then
      { 시스템 트레이 아이콘의 풍선 도움말을 변경한다. }
      SendTrayMessage(NIM_MODIFY, NIF_TIP);
  end;
end;
```

## 마우스 클릭 처리

시스템 트레이를 다룰 때 가장 신경을 써야 하는 부분이 트레이 통지 아이콘의 마우스 클릭 처리이다. 일반적으로, 시스템 트레이의 통지 아이콘들은 마우스 클릭에 대해 다음과 같은 반응을 한다.

- 마우스를 한 번 클릭하면 어떤 윈도우가 나타난다.
- 마우스를 더블클릭하면 다른 어떤 윈도우(등록 정보나 옵션 대화상자 등)가 나타난다.
- 마우스를 오른쪽 클릭하면 팝업 메뉴가 나타난다.

문제는 더블클릭을 잡아내기가 어렵다는 것이다. 더블클릭을 하기 전에 단일클릭 이벤트가 발생해 버리기 때문이다.

윈도우즈 메시지의 관점에서 볼 때 사용자가 기본 마우스 버튼(일반적으로 왼쪽)을 더블클릭하면 포커스를 가진 윈도우는 WM_LBUTTONDOWN 메시지와 WM_LBUTTONDBLCLK 메시지를 받는다. 즉, 단일클릭 메시지가 먼저 오고, 그 다음에 더블클릭 메시지가 오는 것이다. 따라서, 단일클릭 메시지 이후에 더블클릭 메시지가 올 것인지 아닌지를 판단할 수 있어야 더블클릭을 처리할 수 있다.

가장 간단한 해결책은 WM_LBUTTONDOWN 메시지를 받아서 즉시 처리하는 대신 WM_LBUTTONDBLCLK 메시지가 오기를 잠시 기다려서 정해진 시간 안에 WM_LBUTTONDBLCLK가 오면 더블클릭으로 처리하고, 안오면 단일클릭으로 처리하는 것이다. 대기 시간은 개발자가 임의로 정의하기보다는 사용자가 제어판의 마우스 애플릿에서 설정한 값을 사용하는 것이 좋다. 제어판에 설정된 더블클릭 시간은 API 함수 GetDoubleClickTime()로 얻으면 된다. 이 함수는 인자가 없으며, 리턴값은 밀리 초 단위의 더블클릭 대기 시간이다. 그리고 일정 시간을 기다리는 알고리즘은 TTimer로 구현하면 된다. 다음은 TTrayNotifiy 컴포넌트의 생성자 코드 중 TTimer의 인스턴스를 생성하고 초기화하는 부분이다.

```
FTimer := TTimer.Create(Self);
with FTimer do
begin
  Enabled := False;
  Interval := GetDoubleClickTime;
  OnTimer := OnButtonTimer;
end;
```

타이머 이벤트가 발생하면 OnButtonTimer()가 실행된다. 이 메소드에 대해서는 잠시 후에 이야기하겠다.

그럼 이번에는 시스템 트레이가 보내는 통지 메시지를 처리하는 IconMgr의 TrayWndProc() 메소드를 살펴보자. 다음과 같다.

```
procedure TIconManager.TrayWndProc(var Message: TMessage);
{ 이 메소드는 컴포넌트의 문맥 안에서 트레이의 콜백 메시지들을 처리하기
  위한 윈도우 프로시저이다. }
var
  Pt: TPoint;
  TheIcon: TTrayNotifyIcon;
begin
  with Message do
  begin
    { 이 컴포넌트가 등록한 트레이 아이콘의 메시지이면 }
    if (Msg = DDGM_TRAYICON) then
    begin
      TheIcon := TTrayNotifyIcon(WParam);
      case lParam of
        { 첫 번째 마우스 클릭이면 타이머를 활성화시킨다. }
        { 일정 시간동안 더블클릭이 일어나지 않으면 }
        { 타이머 이벤트 처리 메소드는 OnClick()을 호출한다. }
        WM_LBUTTONDOWN: TheIcon.FTimer.Enabled := True;
        { 더블클릭이면 FNoShowClick을 True로 설정한다. 이렇게 }
        { 하면 단일클릭 이벤트는 발생하지 않는다. }
        WM_LBUTTONDBLCLK:
          begin
            TheIcon.FNoShowClick := True;
            if Assigned(TheIcon.FOnDblClick) then
                  TheIcon.FOnDblClick(Self);
          end;
        WM_RBUTTONDOWN:
          begin
            if Assigned(TheIcon.FPopupMenu) then
            begin
              { SetForegroundWindow()를 먼저 호출해야 한다. }
```

```
            SetForegroundWindow(IconMgr.HWindow);
            { 현재 커서 위치에서 팝업 메뉴를 띄운다. }
            GetCursorPos(Pt);
            TheIcon.FPopupMenu.Popup(Pt.X, Pt.Y);
            { 작업을 전환하려면 메시지를 발송해야(post) 한다. }
            PostMessage(IconMgr.HWindow, WM_USER, 0, 0);
          end;
        end;
      end;
    end
    else
      { 등록된 트레이 콜백 메시지가 아니면 DefWindowProc()를 호출한다. }
      Result := DefWindowProc(FHWindow, Msg, wParam, lParam);
  end;
end;
```

단일클릭 메시지이면 타이머를 활성화시키며, 더블클릭 메시지이면 더블클릭이 일어났다는 뜻의 플래그를 설정한 다음, 컴포넌트의 OnDlbClick 이벤트를 발생시킨다. 오른쪽 클릭 메시지이면 컴포넌트의 PopupMenu 속성으로 지정된 팝업 메뉴를 띄운다. 그럼 OnButtonTimer() 메소드를 살펴보자.

```
procedure TTrayNotifyIcon.OnButtonTimer(Sender: TObject);
begin
  { 이 이벤트는 클릭당 한 번만 일어나면 되므로 타이머를 비활성화시킨다.  }
  FTimer.Enabled := False;
  { 더블클릭이 일어나지 않았으면 단일클릭 이벤트를 발생시킨다. }
  if (not FNoShowClick) and Assigned(FOnClick) then
    FOnClick(Self);
  FNoShowClick := False;   // 플래그를 초기화한다.
end;
```

이 메소드는 우선 마우스 클릭당 한 번만 이 이벤트가 발생하도록 하기 위해서 타이머를 비활성화시킨다. 그런 다음, FNoShowClick 플래그를 점검한다. 더블클릭 메시지가 이미 도착한 경우에는 이 플래그가 True이다. 따라서, 이 플래그가 True이면 OnClick 이벤트를 발생시키지 않는다.

## 작업 표시줄에서 애플리케이션 숨기기

대부분의 트레이 통지 애플리케이션들은 작업 표시줄에 자신의 버튼을 만들지 않는다. 이러한 기능을 구현하기 위해서 TTrayNotifyIcon 컴포넌트는 HideTask라는 속성을 제공한다. 컴포넌트 사용자는 이 속성을 통해서 애플리케이션을 작업 표시줄에 나타나게 할지를 결정하게 된다. 이 속성의 쓰기 메소드가 아래에 나와 있다. ShowWindow() API 함수의 두 번째 인자는 애플리케이션을 숨길 것인지 아닌지를 결정하는 것이다.

```
procedure TTrayNotifyIcon.SetHideTask(Value: Boolean);
{ HideTask 속성의 쓰기 메소드 }
const
  { 애플리케이션을 표시할 것인지 아닌지를 결정하는 플래그들을 선언한다. }
  ShowArray: array[Boolean] of integer = (sw_ShowNormal, sw_Hide);
begin
  if FHideTask <> Value then
  begin
    FHideTask := Value;
    { 설계 시점이면 아무런 일도 하지 않는다. }
    if not (csDesigning in ComponentState) then
      ShowWindow(Application.Handle, ShowArray[FHideTask]);
  end;
end;
```

목록 24.1은 TTrayNotifyIcon 컴포넌트와 TIconManager 객체가 정의되어 있는 TrayIcon.pas 유닛의 소스 코드이다.

목록 24.1 Comps\TrayIcon.pas

```
unit TrayIcon;

interface

uses Windows, SysUtils, Messages, ShellAPI, Classes, Graphics, Forms, Menus,
  StdCtrls, ExtCtrls;

type
  ENotifyIconError = class(Exception);
```

```
TTrayNotifyIcon = class(TComponent)
private
  FDefaultIcon: THandle;
  FIcon: TIcon;
  FHideTask: Boolean;
  FHint: string;
  FIconVisible: Boolean;
  FPopupMenu: TPopupMenu;
  FOnClick: TNotifyEvent;
  FOnDblClick: TNotifyEvent;
  FNoShowClick: Boolean;
  FTimer: TTimer;
  Tnd: TNotifyIconData;
  procedure SetIcon(Value: TIcon);
  procedure SetHideTask(Value: Boolean);
  procedure SetHint(Value: string);
  procedure SetIconVisible(Value: Boolean);
  procedure SetPopupMenu(Value: TPopupMenu);
  procedure SendTrayMessage(Msg: DWORD; Flags: UINT);
  function ActiveIconHandle: THandle;
  procedure OnButtonTimer(Sender: TObject);
protected
  procedure Loaded; override;
  procedure LoadDefaultIcon; virtual;
  procedure Notification(AComponent: TComponent;
    Operation: TOperation); override;
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
published
  property Icon: TIcon read FIcon write SetIcon;
  property HideTask: Boolean read FHideTask write SetHideTask default
           False;
  property Hint: String read FHint write SetHint;
  property IconVisible: Boolean read FIconVisible write SetIconVisible
           default False;
  property PopupMenu: TPopupMenu read FPopupMenu write SetPopupMenu;
  property OnClick: TNotifyEvent read FOnClick write FOnClick;
  property OnDblClick: TNotifyEvent read FOnDblClick write FOnDblClick;
```

```
  end;

implementation

{ TIconManager }
{ 이 클래스는 트레이 아이콘 메시지를 처리하는 숨겨진 윈도우를 생성한다. }
type
  TIconManager = class
  private
    FHWindow: HWnd;
    procedure TrayWndProc(var Message: TMessage);
  public
    constructor Create;
    destructor Destroy; override;
    property HWindow: HWnd read FHWindow write FHWindow;
  end;

var
  IconMgr: TIconManager;
  DDGM_TRAYICON: Cardinal;

constructor TIconManager.Create;
begin
  FHWindow := AllocateHWnd(TrayWndProc);
end;

destructor TIconManager.Destroy;
begin
  if FHWindow <> 0 then DeallocateHWnd(FHWindow);
  inherited Destroy;
end;

procedure TIconManager.TrayWndProc(var Message: TMessage);
{ 이 메소드는 컴포넌트의 문맥 안에서 트레이의 콜백 메시지들을 처리하기 위한
  윈도우 프로시저이다. }
var
  Pt: TPoint;
  TheIcon: TTrayNotifyIcon;
begin
```

```
  with Message do
  begin
    { 이 컴포넌트가 등록한 트레이 아이콘의 메시지이면 }
    if (Msg = DDGM_TRAYICON) then
    begin
      TheIcon := TTrayNotifyIcon(WParam);
      case lParam of
        { 첫 번째 마우스 클릭이면 타이머를 활성화시킨다. }
        { 일정 시간동안 더블클릭이 일어나지 않으면 }
        { 타이머 이벤트 처리 메소드는 OnClick()을 호출한다. }
        WM_LBUTTONDOWN: TheIcon.FTimer.Enabled := True;
        { 더블클릭이면 FNoShowClick을 True로 설정한다. }
        { 이렇게 하면 단일클릭 이벤트는 발생하지 않는다. }
        WM_LBUTTONDBLCLK:
          begin
            TheIcon.FNoShowClick := True;
            if Assigned(TheIcon.FOnDblClick) then
                  TheIcon.FOnDblClick(Self);
          end;
        WM_RBUTTONDOWN:
          begin
            if Assigned(TheIcon.FPopupMenu) then
            begin
              { SetForegroundWindow()를 먼저 호출해야 한다. }
              SetForegroundWindow(IconMgr.HWindow);
              { 현재 커서 위치에서 팝업 메뉴를 띄운다. }
              GetCursorPos(Pt);
              TheIcon.FPopupMenu.Popup(Pt.X, Pt.Y);
              { 작업을 전환하려면 메시지를 발송해야(post) 한다. }
              PostMessage(IconMgr.HWindow, WM_USER, 0, 0);
            end;
          end;
      end;
    end
    else
      { 등록된 트레이 콜백 메시지가 아니면 DefWindowProc()를 호출한다. }
      Result := DefWindowProc(FHWindow, Msg, wParam, lParam);
  end;
end;
```

```
{ TTrayNotifyIcon }

constructor TTrayNotifyIcon.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FIcon := TIcon.Create;
  FTimer := TTimer.Create(Self);
  with FTimer do
  begin
    Enabled := False;
    Interval := GetDoubleClickTime;
    OnTimer := OnButtonTimer;
  end;
  { 기본 아이콘을 미리 마련해 둔다. }
  LoadDefaultIcon;
end;

destructor TTrayNotifyIcon.Destroy;
begin
  if FIconVisible then SetIconVisible(False);    // 아이콘을 파괴
  FIcon.Free;                                    // 기타 인스턴스들을 해제
  FTimer.Free;
  inherited Destroy;
end;

function TTrayNotifyIcon.ActiveIconHandle: THandle;
{ 활성화된 아이콘 핸들을 돌려준다. }
begin
  { 아이콘이 하나도 로드되어 있지 않으면 기본 아이콘의 핸들을 돌려준다. }
  if (FIcon.Handle <> 0) then
    Result := FIcon.Handle
  else
    Result := FDefaultIcon;
end;

procedure TTrayNotifyIcon.LoadDefaultIcon;
{ 기본 아이콘으로 사용할 윈도우즈 아이콘을 로드하는 메소드이다. Icon 속성에
  아무런 아이콘도 설정되어 있지 않으면 컴포넌트는 이 아이콘을 사용한다. }
begin
```

```
  FDefaultIcon := LoadIcon(0, IDI_WINLOGO);
end;

procedure TTrayNotifyIcon.Loaded;
{ 스트림으로부터 컴포넌트가 로드된 후에 호출된다. }
begin
  inherited Loaded;
  { 아이콘을 표시해야 하는 것으로 설정되어 있으면 아이콘을 트레이에 추가한다.}
  if FIconVisible then
    SendTrayMessage(NIM_ADD, NIF_MESSAGE or NIF_ICON or NIF_TIP);
end;

procedure TTrayNotifyIcon.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (Operation = opRemove) and (AComponent = PopupMenu) then
    PopupMenu := nil;
end;

procedure TTrayNotifyIcon.OnButtonTimer(Sender: TObject);
{ 더블클릭의 두 번째 클릭을 처리하기 위한 타이머 이벤트이다.
  일단 마우스가 한 번 클릭되면 타이머가 활성화되며, 일정 시간이 지나면 이 이벤트
  처리 메소드가 수행된다. 이 메소드가 수행되기 전까지 두 번째 클릭이 발생했으면
  FNoShowClick은 True인 상태이다.
  FNoShowClick이 True이면 단일클릭을 뜻하는 OnClick 이벤트를 발생시키지
  않는다. }
begin
  { 이 이벤트는 클릭당 한 번만 일어나면 되므로 타이머를 비활성화시킨다. }
  FTimer.Enabled := False;
  { 더블클릭이 일어나지 않았으면 단일클릭 이벤트를 발생시킨다. }
  if (not FNoShowClick) and Assigned(FOnClick) then FOnClick(Self);
  FNoShowClick := False;   // 플래그를 초기화한다.
end;

procedure TTrayNotifyIcon.SendTrayMessage(Msg: DWORD; Flags: UINT);
{ 이 메소드는 API 함수 Shell_NotifyIcon()을 호출한다. }
begin
  { 레코드를 적절한 값으로 채운다. }
```

```
  with Tnd do
  begin
    cbSize := SizeOf(Tnd);
    StrPLCopy(szTip, PChar(FHint), SizeOf(szTip));
    uFlags := Flags;
    uID := UINT(Self);
    Wnd := IconMgr.HWindow;
    uCallbackMessage := DDGM_TRAYICON;
    hIcon  := ActiveIconHandle;
  end;
  Shell_NotifyIcon(Msg, @Tnd);
end;

procedure TTrayNotifyIcon.SetHideTask(Value: Boolean);
{ HideTask 속성의 쓰기 메소드 }
const
  { 애플리케이션을 표시할 것인지 아닌지를 결정하는 플래그들을 선언한다. }
  ShowArray: array[Boolean] of integer = (sw_ShowNormal, sw_Hide);
begin
  if FHideTask <> Value then
  begin
    FHideTask := Value;
    { 설계 시점이면 아무런 일도 하지 않는다. }
    if not (csDesigning in ComponentState) then
      ShowWindow(Application.Handle, ShowArray[FHideTask]);
  end;
end;

procedure TTrayNotifyIcon.SetHint(Value: string);
{ Hint 속성의 쓰기 메소드 }
begin
  if FHint <> Value then
  begin
    FHint := Value;
    if FIconVisible then
      { 시스템 트레이의 아이콘의 풍선 도움말을 변경한다. }
      SendTrayMessage(NIM_MODIFY, NIF_TIP);
  end;
end;
```

```
procedure TTrayNotifyIcon.SetIcon(Value: TIcon);
{ Icon 속성의 쓰기 메소드 }
begin
  FIcon.Assign(Value);  // 새 아이콘을 설정한다.
  { 시스템 트레이의 아이콘을 변경한다. }
  if FIconVisible then SendTrayMessage(NIM_MODIFY, NIF_ICON);
end;

procedure TTrayNotifyIcon.SetIconVisible(Value: Boolean);
{ IconVisible 속성의 쓰기 메소드 }
const
  { 트레이 통지 아이콘을 추가할 것인지 제거할 것인지를 뜻하는 플래그들 }
  MsgArray: array[Boolean] of DWORD = (NIM_DELETE, NIM_ADD);
begin
  if FIconVisible <> Value then
  begin
    FIconVisible := Value;
    { 아이콘을 적절히 설정한다. }
    SendTrayMessage(MsgArray[Value], NIF_MESSAGE or NIF_ICON or NIF_TIP);
  end;
end;

procedure TTrayNotifyIcon.SetPopupMenu(Value: TPopupMenu);
{ PopupMenu 속성의 쓰기 메소드 }
begin
  FPopupMenu := Value;
  if Value <> nil then Value.FreeNotification(Self);
end;

const
  { 윈도우즈 메시지를 등록하는 데 필요한 문자열 }
  TrayMsgStr = 'DDG.TrayNotifyIconMsg';

initialization
  { 트레이 콜백에 사용할 고유한 윈도우즈 메시지 ID를 얻는다. }
  DDGM_TRAYICON := RegisterWindowMessage(TrayMsgStr);
```

```
    IconMgr := TIconManager.Create;
  finalization
    IconMgr.Free;
  end.
```

그림 24.2는 TTrayNotifyIcon 컴포넌트를 이용해서 시스템 트레이에 추가된 아이콘의 모습이다.



**그림 24.2**
TTrayNotifyIcon
컴포넌트가 추가한
아이콘

그런데 트레이 아이콘은 컴포넌트의 생성자에서 시스템 트레이에 추가되며, 컴포넌트의 생성자는 설계 시점에서도 실행되므로 설계 시점에서도 시스템 트레이에 아이콘이 추가된다는 점을 기억하기 바란다.

## 예제 트레이 애플리케이션

그럼 TTrayNotifyIcon 컴포넌트를 사용하는 간단한 예제 애플리케이션을 살펴보자. 그림 24.3은 이번 예제 프로젝트인 TrayDemo의 메인 폼이며, 목록 24.2는 메인 폼 유닛의 소스 코드이다.

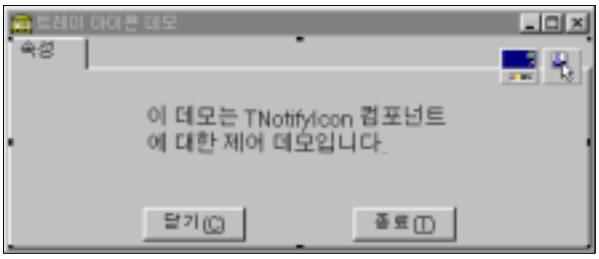



**그림 24.3**
예제 트레이 응용 프로그램의 메인 폼

목록 24.2 Ch24\TrayDemo\Main.pas

```
unit main;

interface

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

type
  TMainForm = class(TForm)
    pmiPopup: TPopupMenu;
    pgclPageCtl: TPageControl;
    TabSheet1: TTabSheet;
    btnClose: TButton;
    btnTerm: TButton;
    Terminate1: TMenuItem;
    Label1: TLabel;
    N1: TMenuItem;
    Propeties1: TMenuItem;
    TrayNotifyIcon1: TTrayNotifyIcon;
    procedure NotifyIcon1Click(Sender: TObject);
    procedure NotifyIcon1DblClick(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure btnTermClick(Sender: TObject);
    procedure btnCloseClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}
```

```
procedure TMainForm.NotifyIcon1Click(Sender: TObject);
begin
  ShowMessage('단일클릭');
end;

procedure TMainForm.NotifyIcon1DblClick(Sender: TObject);
begin
  Show;
end;

procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Action := caNone;
  Hide;
end;

procedure TMainForm.btnTermClick(Sender: TObject);
begin
  Application.Terminate;
end;

procedure TMainForm.btnCloseClick(Sender: TObject);
begin
  Hide;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  TrayNotifyIcon1.IconVisible := True;
end;

end.
```

# 애플리케이션 데스크탑 툴바

애플리케이션 데스크탑 툴바, 줄여서 흔히 AppBar(이하 앱바)라고 하는 것은 화면의 가장자리에 붙는(dock) 윈도우를 말한다. 윈도우즈의 작업 표시줄이 바로 앱바의 대표적인 예이다. 그림 24.4가 윈도우즈의 작업 표시줄인데, 이것은 시작 버튼과 시스템 트레이, 그리고 몇 개의 버튼들이 들어 있는 앱바 윈도우라고 할 수 있다.



**그림 24.4**
윈도우즈 쉘의 작업 표시줄

앱바는 화면 가장자리에 붙는다는 특징 말고도 자동 숨김이라던가 드래그 앤 드롭 기능도 가질 수 있다. 실제로 작업 표시줄은 그런 기능들을 가지고 있다. 그런데 앱바 역시 트레이 통지 아이콘처럼 단 하나의 API 함수로 제어할 수 있다. 다만, 그 API 함수는 앱바 자체를 제어하기 위한 함수라기 보다는 시스템에 앱바를 등록하거나 등록된 앱바에 대한 정보를 얻기 위한 것이라고 하는 편이 더 정확할 것이다. 앱바의 행동은 대부분 쉘이 관리하기 때문이다.

## 관련 API 함수

앱바에 관련된 단 하나의 API 함수는 다름 아닌 SHAppBarMessage()인데, ShellAPI 유닛에는 다음과 같이 선언되어 있다.

```
unction SHAppBarMessage(dwMessage: DWORD; var pData: TAppBarData): UINT;
  stdcall;
```

이 함수의 첫 번째 인자인 dwMessage에 들어갈 수 있는 값은 표 24.3과 같다.

**표 24.3** 앱바 메시지들

| 상 수 | 값 | 의 미 |
|---|---|---|
| ABM_NEW | $0 | 새 앱바를 등록하고 새 콜백 메시지를 지정한다. |
| ABM_REMOVE | $1 | 기존의 앱바를 제거한다. |
| ABM_QUERYPOS | $2 | 앱바의 위치와 크기를 얻는다. |
| ABM_SETPOS | $3 | 앱바의 위치와 크기를 새로 설정한다. |

| 상 수 | 값 | 의 미 |
|---|---|---|
| ABM_GETSTATE | $4 | 윈도우즈 작업 표시줄의 '자동 숨김' 여부나 '항상 위' 여부를 얻는다. |
| ABM_GETTASKBARPOS | $5 | 윈도우즈 작업 표시줄의 위치를 얻는다. |
| ABM_ACTIVATE | $6 | 쉘에게 새 앱바가 생성되었음을 알린다. |
| ABM_GETAUTOHIDEBAR | $7 | 화면의 특정 가장자리에 붙어 있는 자동 숨김 앱바의 핸들을 얻는다. |
| ABM_SETAUTOHIDEBAR | $8 | 화면의 특정 가장자리에 붙일 자동 숨김 앱바를 등록한다. |
| ABM_WINDOWPOSCHANGED | $9 | 앱바의 위치가 변했음을 쉘에게 알린다. |

SHAppBarMessage()의 pData 인자는 TAppBarData형의 레코드인데, TAppBarData는 ShellAPI에 다음과 같이 정의되어 있다.

```
type
  PAppBarData = ^TAppBarData;
  TAppBarData = record
    cbSize: DWORD;
    hWnd: HWND;
    uCallbackMessage: UINT;
    uEdge: UINT;
    rc: TRect;
    lParam: LPARAM; { message specific }
  end;
```

cbSize 필드에는 이 레코드의 크기를 넣으며, hWnd 필드에는 지정된 앱바의 윈도우 핸들을 넣는다. uCallBackMessage에는 쉘이 앱바 윈도우에게 보낼 통지 메시지 ID를 넣으며, rc는 앱바의 크기와 위치를 담는다. 그리고 lParam은 개발자가 사용할 추가적인 정보를 담기 위한 것이다.



SHAppBarMessage() API 함수나 TAppBarData형(Win32에서는 APPBARDATA)에 대한 좀더 자세한 정보는 Win32 도움말이나 델파이 도움말을 참고하기 바란다.

## TAppBar : 앱바 폼

SHAppBarMessage()는 별로 복잡한 API 함수는 아니지만, 그래도 델파이 사용자답게 이 함수를 캡슐화하는 VCL 폼을 만들어 보자. TCustomForm 클래스를 상속해서 앱바 관련 API를 감싸는 컴포넌트를 만들어 두면 언제라도 다시 사용할 수 있다. 그런데 TCustomForm은 폼이기 때문에 TCustomForm을 상속한 컴포넌트 역시 폼이 되며, 따라서 폼 디자이너에서 폼 위의 한 컴포넌트가 아니라 폼 자체로 작동한다.

TAppBar 컴포넌트가 수행하는 대부분의 작업은 TAppBarData 레코드를 적절히 채우고 SHAppBarMessage() API를 호출하는 것이다. TAppBar 컴포넌트에는 TAppBarData 형의 FABD라는 내부 필드가 있다. 생성자에서는 FABD를 적절히 채워서 SendAppBarMsg()를 호출한다. 이때 FABD의 cbSize 필드에 레코드 크기를 넣고, uCallbackMessage 필드에는 RegisterWindowMessage() API로 얻은 메시지 IDYgw 넣으며, hWnd 필드에는 폼의 현재 윈도우 핸들을 넣는다. SendAppBarMessage()는 단순히 SHAppBarMessage()의 래퍼(wrapper) 메소드인데, 정의는 다음과 같다.

```
function TAppBar.SendAppBarMsg(Msg: DWORD): UINT;
begin
  Result := SHAppBarMessage(Msg, FABD);
end;
```

앱바가 성공적으로 생성되면 SetAppBarEdge() 메소드를 호출해서 앱바의 초기 위치를 설정한다. SetAppBarEdge() 메소드는 SetAppBarPos() 메소드에 적절한 플래그를 넣어서 앱바를 화면의 특정 가장자리에 붙인다. 화면 가장자리를 나타내는 플래그에는 ABE_TOP, ABE_BOTTOM, ABE_LEFT, ABE_RIGHT가 있다. 이름만으로도 어떤 의미인지 알 수 있을 것이다. 다음은 SetAppBarBos()의 코드이다.

```
procedure TAppBar.SetAppBarPos(Edge: UINT);
begin
  FABD.uEdge := Edge;       // 가장자리를 지정한다.
  with FABD.rc do
  begin
    // 전체 화면 좌표들을 설정한다.
    Top := 0;
    Left := 0;
    Right := Screen.Width;
    Bottom := Screen.Height;
```

```
    // ABM_QUERYPOS를 보내서 가장자리의 위치 정보를 얻는다.
    SendAppBarMsg(ABM_QUERYPOS);
    // 인자로 지정된 가장자리에 기반해서
    // ABM_QUERYPOS로 얻은 위치를 적절히 수정한다.
    case Edge of
      ABE_LEFT: Right := Left + DEF_APPBAR_WIDTH;
      ABE_RIGHT: Left := Right - DEF_APPBAR_WIDTH;
      ABE_TOP: Bottom := Top + DEF_APPBAR_HEIGHT;
      ABE_BOTTOM: Top := Bottom - DEF_APPBAR_HEIGHT;
    end;
    // 앱바의 위치와 크기(사각형 영역)를 설정한다.
    SendAppBarMsg(ABM_SETPOS);
  end;
  // 폼의 BoundRect 속성의 값을 쉘에 넘겨준 위치/크기에 일치시킨다.
  BoundsRect := FABD.rc;
end;
```

이 메소드는 우선 FADB의 uEdge 필드를 Edge 인자의 값으로 설정한다. 그런 다음, rc 필드를 전체 화면 좌표로 설정하고 ABM_QUERYPOS 메시지를 보낸다. 그러면 rc는 uEdge로 지정된 가장자리의 위치/크기 정보로 갱신된다. 그런 다음에는 앱바가 제대로 나타날 수 있도록 rc의 사각형 영역 좌표를 적절히 설정한다. rc에 앱바의 정확한 사각형 영역 좌표를 설정한 후에는 ABM_SETPOST 메시지를 보내서 쉘에게 앱바의 위치와 크기를 알려준다. 그런 다음 rc, 즉 사각형 영역 좌표를 폼의 BoundsRect 속성에 넣는다.

쉘이 앱바에게 보내는 통지 메시지(FADB.uCallbackMessage로 등록한 것)는 FABD.hWnd로 지정된 윈도우가 받는다. 다음은 그 윈도우의 윈도우 프로시저(메시지 처리 프로시저)로 사용하는 WndProc() 메소드의 코드이다.

```
procedure TAppBar.WndProc(var M: TMessage);
var
  State: UINT;
  WndPos: HWnd;
begin
  if M.Msg = AppBarMsg then
  begin
    case M.WParam of
```

```
    // '항상 위'나 '자동 숨김' 상태가 변경되면 이 메시지가 온다.
    ABN_STATECHANGE:
      begin
        // 앱바가 아직 ABS_ALWAYSONTOP 상태인지 점검
        State := SendAppBarMsg(ABM_GETSTATE);
        if ABS_ALWAYSONTOP and State = 0 then
          SetTopMost(False)
        else
          SetTopMost(True);
      end;
     // 어떤 전체 화면 애플리케이션이 실행되었거나 이전의
     // 전체 화면 애플리케이션이 종료되었을 때 이 메시지가 온다.
    ABN_FULLSCREENAPP:
      begin
        // 앱바의 겹침 순서(z-order)를 적절히 설정한다.
        State := SendAppBarMsg(ABM_GETSTATE);
        if M.lParam <> 0 then begin
          if ABS_ALWAYSONTOP and State = 0 then
            SetTopMost(False)
          else
            SetTopMost(True);
        end
        else
          if State and ABS_ALWAYSONTOP <> 0 then
            SetTopMost(True);
      end;
         // 앱바의 위치/크기에 영향을 미치는 사건이 발생하면 이 메시지가 온다.

    ABN_POSCHANGED:
      begin
        // 작업 표시줄이나 다른 앱바의 크기나 위치가 변경되었으므로
        // 이 앱바의 크기/위치를 다시 설정한다.
        SetAppBarPos(FABD.uEdge);
      end;
  end;
```

```
  end
  else
    inherited WndProc(M);
end;
```

이 메소드는 앱바에게 보내진 여러 메시지들을 적절히 처리하는 역할을 한다. 목록 24.3은 TAppBar 폼을 정의하는 AppBars.pas 유닛의 전체 코드이다.

목록 24.3 Comps\AppBars.pas

```
unit AppBars;

interface

uses Windows, Messages, SysUtils, Forms, ShellAPI, Classes, Controls;

type
  TAppBarEdge = (abeTop, abeBottom, abeLeft, abeRight);

  EAppBarError = class(Exception);

  TAppBar = class(TCustomForm)
  private
    FABD: TAppBarData;
    FDockedHeight: Integer;
    FDockedWidth: Integer;
    FEdge: TAppBarEdge;
    FOnEdgeChanged: TNotifyEvent;
    FTopMost: Boolean;
    procedure WMActivate(var M: TMessage); message WM_ACTIVATE;
    procedure WMWindowPosChanged(var M: TMessage); message
          WM_WINDOWPOSCHANGED;
    function SendAppBarMsg(Msg: DWORD): UINT;
    procedure SetAppBarEdge(Value: TAppBarEdge);
    procedure SetAppBarPos(Edge: UINT);
    procedure SetTopMost(Value: Boolean);
    procedure SetDockedHeight(const Value: Integer);
    procedure SetDockedWidth(const Value: Integer);
```

```
protected
  procedure CreateParams(var Params: TCreateParams); override;
  procedure CreateWnd; override;
  procedure DestroyWnd; override;
  procedure WndProc(var M: TMessage); override;
public
  constructor CreateNew(AOwner: TComponent; Dummy: Integer = 0);
        override;
  property DockManager;
published
  property Action;
  property ActiveControl;
  property AutoScroll;
  property AutoSize;
  property BiDiMode;
  property BorderWidth;
  property Color;
  property Ctl3D;
  property DockedHeight: Integer read FDockedHeight write SetDockedHeight
    default 35;
  property DockedWidth: Integer read FDockedWidth write SetDockedWidth
    default 40;
  property UseDockManager;
  property DockSite;
  property DragKind;
  property DragMode;
  property Edge: TAppBarEdge read FEdge write SetAppBarEdge default
        abeTop;
  property Enabled;
  property ParentFont default False;
  property Font;
  property HelpFile;
  property HorzScrollBar;
  property Icon;
  property KeyPreview;
  property ObjectMenuItem;
  property ParentBiDiMode;
  property PixelsPerInch;
  property PopupMenu;
```

```
property PrintScale;
property Scaled;
property ShowHint;
property TopMost: Boolean read FTopMost write SetTopMost default
      False;
property VertScrollBar;
property Visible;
property OnActivate;
property OnCanResize;
property OnClick;
property OnClose;
property OnCloseQuery;
property OnConstrainedResize;
property OnCreate;
property OnDblClick;
property OnDestroy;
property OnDeactivate;
property OnDockDrop;
property OnDockOver;
property OnDragDrop;
property OnDragOver;
property OnEdgeChanged: TNotifyEvent read FOnEdgeChanged
  write FOnEdgeChanged;
property OnEndDock;
property OnGetSiteInfo;
property OnHide;
property OnHelp;
property OnKeyDown;
property OnKeyPress;
property OnKeyUp;
property OnMouseDown;
property OnMouseMove;
property OnMouseUp;
property OnMouseWheel;
property OnMouseWheelDown;
property OnMouseWheelUp;
property OnPaint;
property OnResize;
property OnShortCut;
```

```
    property OnShow;
    property OnStartDock;
    property OnUnDock;
  end;

implementation

var
  AppBarMsg: UINT;

constructor TAppBar.CreateNew(AOwner: TComponent; Dummy: Integer);
begin
  FDockedHeight := 35;
  FDockedWidth := 40;
  inherited CreateNew(AOwner, Dummy);
  ClientHeight := 35;
  Width := 100;
  BorderStyle := bsNone;
  BorderIcons := [];
  // TAppBarData 레코드를 초기화한다.
  FABD.cbSize := SizeOf(FABD);
  FABD.uCallbackMessage := AppBarMsg;
end;

procedure TAppBar.WMWindowPosChanged(var M: TMessage);
begin
  inherited;
  // 쉘에게 앱바의 위치가 변경되었음을 알려야 한다.
  SendAppBarMsg(ABM_WINDOWPOSCHANGED);
end;

procedure TAppBar.WMActivate(var M: TMessage);
begin
  inherited;
  // 쉘에게 앱바 윈도우가 활성화되었음을 알려야 한다.
  SendAppBarMsg(ABM_ACTIVATE);
end;

procedure TAppBar.WndProc(var M: TMessage);
```

```
var
  State: UINT;
begin
  if M.Msg = AppBarMsg then
  begin
    case M.WParam of
      // '항상 위'나 '자동 숨김' 상태가 변경되면 이 메시지가 온다.
      ABN_STATECHANGE:
        begin
          // 앱바가 아직 ABS_ALWAYSONTOP 상태인지 점검
          State := SendAppBarMsg(ABM_GETSTATE);
          if ABS_ALWAYSONTOP and State = 0 then
            SetTopMost(False)
          else
            SetTopMost(True);
        end;
      // 어떤 전체 화면 애플리케이션이 실행되었거나 이전의
      // 전체 화면 애플리케이션이 종료되었을 때 이 메시지가 온다.
      ABN_FULLSCREENAPP:
        begin
          // 앱바의 겹침 순서(z-order)를 적절히 설정한다.
          State := SendAppBarMsg(ABM_GETSTATE);
          if M.lParam <> 0 then begin
            if ABS_ALWAYSONTOP and State = 0 then
              SetTopMost(False)
            else
              SetTopMost(True);
          end
          else
            if State and ABS_ALWAYSONTOP <> 0 then
              SetTopMost(True);
        end;
          // 앱바의 위치/크기에 영향을 미치는 사건이 발생하면 이 메시지가 온다.
      ABN_POSCHANGED:
          // 작업 표시줄이나 다른 앱바의 크기나 위치가 변경되었으므로
          // 이 앱바의 크기/위치를 다시 설정한다.
        SetAppBarEdge(FEdge);
    end;
  end
```

```
  else
    inherited WndProc(M);
end;

function TAppBar.SendAppBarMsg(Msg: DWORD): UINT;
begin
  // 설계 시점에서는 앱바를 조작하지 않는 것이 더 깔끔하다.
  if csDesigning in ComponentState then Result := 0
  else Result := SHAppBarMessage(Msg, FABD);
end;

procedure TAppBar.SetAppBarPos(Edge: UINT);
begin
  if csDesigning in ComponentState then Exit;
  FABD.uEdge := Edge;       // 가장자리를 지정한다.
  with FABD.rc do
  begin
    // 전체 화면 좌표들을 설정한다.
    Top := 0;
    Left := 0;
    Right := Screen.Width;
    Bottom := Screen.Height;
    // ABM_QUERYPOS를 보내서 가장자리의 위치 정보를 얻는다.
    SendAppBarMsg(ABM_QUERYPOS);
    // 인자로 지정된 가장자리에 기반해서
      // ABM_QUERYPOS로 얻은 위치를 적절히 수정한다.
    case Edge of
      ABE_LEFT: Right := Left + FDockedWidth;
      ABE_RIGHT: Left := Right - FDockedWidth;
      ABE_TOP: Bottom := Top + FDockedHeight;
      ABE_BOTTOM: Top := Bottom - FDockedHeight;
    end;
    // 앱바의 위치와 크기를 설정한다.
    SendAppBarMsg(ABM_SETPOS);
  end;
  // 폼의 BoundRect 속성의 값을 쉘에 넘겨준 위치/크기에 일치시킨다.
  BoundsRect := FABD.rc;
end;
```

```
procedure TAppBar.SetTopMost(Value: Boolean);
const
  WndPosArray: array[Boolean] of HWND = (HWND_BOTTOM, HWND_TOPMOST);
begin
  if FTopMost <> Value then
  begin
    FTopMost := Value;
    if not (csDesigning in ComponentState) then
      SetWindowPos(Handle, WndPosArray[Value], 0, 0, 0, 0, SWP_NOMOVE or
        SWP_NOSIZE or SWP_NOACTIVATE);
  end;
end;

procedure TAppBar.CreateParams(var Params: TCreateParams);
begin
  inherited CreateParams(Params);
  if not (csDesigning in ComponentState) then
  begin
    Params.ExStyle := Params.ExStyle or WS_EX_TOPMOST or WS_EX_WINDOWEDGE;
    Params.Style := Params.Style or WS_DLGFRAME;
  end;
end;

procedure TAppBar.CreateWnd;
begin
  inherited CreateWnd;
  FABD.hWnd := Handle;
  if not (csDesigning in ComponentState) then
  begin
    if SendAppBarMsg(ABM_NEW) = 0 then
      raise EAppBarError.Create(' AppBar 생성 실패');
    // 위치를 초기화한다.
    SetAppBarEdge(FEdge);
  end;
end;

procedure TAppBar.DestroyWnd;
begin
  // 쉘에게 앱바가 파괴됨을 알려야 한다.
```

```
  SendAppBarMsg(ABM_REMOVE);
  inherited DestroyWnd;
end;

procedure TAppBar.SetAppBarEdge(Value: TAppBarEdge);
const
  EdgeArray: array[TAppBarEdge] of UINT =
    (ABE_TOP, ABE_BOTTOM, ABE_LEFT, ABE_RIGHT);
begin
  SetAppBarPos(EdgeArray[Value]);
  FEdge := Value;
  if Assigned(FOnEdgeChanged) then FOnEdgeChanged(Self);
end;

procedure TAppBar.SetDockedHeight(const Value: Integer);
begin
  if FDockedHeight <> Value then
  begin
    FDockedHeight := Value;
    SetAppBarEdge(FEdge);
  end;
end;

procedure TAppBar.SetDockedWidth(const Value: Integer);
begin
  if FDockedWidth <> Value then
  begin
    FDockedWidth := Value;
    SetAppBarEdge(FEdge);
  end;
end;

initialization
  AppBarMsg := RegisterWindowMessage('DDG AppBar 메시지');
 end.
```

## TAppBar의 활용

부록 CD의 패키지들을 제대로 설치했다면 File | New 메뉴의 New Items 대화상자에 DDG 페이지가 나타날 것이다. 그 페이지에서 AppBar를 선택하면 TAppBar가 포함된 유닛을 생성하는 마법사가 뜰 것이다.



TAppBar 유닛을 생성하는 마법사에 대한 설명은 26장, "델파이의 Open Tools API 활용"에 나온다. 이번 장에서는 마법사가 내부적으로 어떻게 작동하는지에 대해서는 일단 접어두고, TAppBar에만 주목하기 바란다. 지금은 그냥 마법사를 통해서 TAppBar가 포함된 폼과 유닛이 자동으로 생성된다고만 알아두면 된다.

이번 예제 애플리케이션은 TAppBar를 이용해서 Open, Save, Cut, Copy, Paste 같은 편집용 버튼들이 포함된 애플리케이션 툴바(앱바)를 만든다. 그 버튼들은 메인 폼의 TMemo에 입력된 내용들을 적절히 조작하는 역할을 한다. 목록 24.4는 앱바가 포함된 유닛의 소스 코드이며, 그림 24.5는 활성화된 앱바가 화면 아래에 붙어 있는 모습이다.



Ch24\AppBar\ApBarFrm.pas

```
unit ApBarFrm;

interface

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

type
  TAppBarForm = class(TAppBar)
    sbOpen: TSpeedButton;
```

```
    sbSave: TSpeedButton;
    sbCut: TSpeedButton;
    sbCopy: TSpeedButton;
    sbPaste: TSpeedButton;
    OpenDialog: TOpenDialog;
    pmPopup: TPopupMenu;
    Top1: TMenuItem;
    Bottom1: TMenuItem;
    Left1: TMenuItem;
    Right1: TMenuItem;
    N1: TMenuItem;
    Exit1: TMenuItem;
    procedure Right1Click(Sender: TObject);
    procedure sbOpenClick(Sender: TObject);
    procedure sbSaveClick(Sender: TObject);
    procedure sbCutClick(Sender: TObject);
    procedure sbCopyClick(Sender: TObject);
    procedure sbPasteClick(Sender: TObject);
    procedure Exit1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormEdgeChanged(Sender: TObject);
  private
    FLastChecked: TMenuItem;
    procedure MoveButtons;
  end;

var
  AppBarForm: TAppBarForm;

implementation

uses Main;

{$R *.DFM}

{ TAppBarForm }

procedure TAppBarForm.MoveButtons;
// 이 메소드는 복잡해 보이긴 하지만, 수행하는 작업은 사실 단순한 것이다.
```

```
// 현재 앱바의 위치에 맞게 각각의 스피드 버튼들의 위치를 적절히 옮기는 것일
// 뿐이다.
var
  DeltaCenter, NewPos: Integer;
begin
  if Edge in [abeTop, abeBottom] then
  begin
    DeltaCenter := (ClientHeight - sbOpen.Height) div 2;
    sbOpen.SetBounds(10, DeltaCenter, sbOpen.Width, sbOpen.Height);
    NewPos := sbOpen.Width + 20;
    sbSave.SetBounds(NewPos, DeltaCenter, sbOpen.Width, sbOpen.Height);
    NewPos := NewPos + sbOpen.Width + 10;
    sbCut.SetBounds(NewPos, DeltaCenter, sbOpen.Width, sbOpen.Height);
    NewPos := NewPos + sbOpen.Width + 10;
    sbCopy.SetBounds(NewPos, DeltaCenter, sbOpen.Width, sbOpen.Height);
    NewPos := NewPos + sbOpen.Width + 10;
    sbPaste.SetBounds(NewPos, DeltaCenter, sbOpen.Width, sbOpen.Height);
  end
  else
  begin
    DeltaCenter := (ClientWidth - sbOpen.Width) div 2;
    sbOpen.SetBounds(DeltaCenter, 10, sbOpen.Width, sbOpen.Height);
    NewPos := sbOpen.Height + 20;
    sbSave.SetBounds(DeltaCenter, NewPos, sbOpen.Width, sbOpen.Height);
    NewPos := NewPos + sbOpen.Height + 10;
    sbCut.SetBounds(DeltaCenter, NewPos, sbOpen.Width, sbOpen.Height);
    NewPos := NewPos + sbOpen.Height + 10;
    sbCopy.SetBounds(DeltaCenter, NewPos, sbOpen.Width, sbOpen.Height);
    NewPos := NewPos + sbOpen.Height + 10;
    sbPaste.SetBounds(DeltaCenter, NewPos, sbOpen.Width, sbOpen.Height);
  end;
end;

procedure TAppBarForm.Right1Click(Sender: TObject);
begin
  FLastChecked.Checked := False;
  (Sender as TMenuItem).Checked := True;
  case TMenuItem(Sender).Caption[2] of
    'T': Edge := abeTop;
```

```
    'B': Edge := abeBottom;
    'L': Edge := abeLeft;
    'R': Edge := abeRight;
  end;
  FLastChecked := TMenuItem(Sender);
end;

procedure TAppBarForm.sbOpenClick(Sender: TObject);
begin
  if OpenDialog.Execute then
    MainForm.FileName := OpenDialog.FileName;
end;

procedure TAppBarForm.sbSaveClick(Sender: TObject);
begin
  MainForm.memEditor.Lines.SaveToFile(MainForm.FileName);
end;

procedure TAppBarForm.sbCutClick(Sender: TObject);
begin
  MainForm.memEditor.CutToClipboard;
end;

procedure TAppBarForm.sbCopyClick(Sender: TObject);
begin
  MainForm.memEditor.CopyToClipboard;
end;

procedure TAppBarForm.sbPasteClick(Sender: TObject);
begin
  MainForm.memEditor.PasteFromClipboard;
end;

procedure TAppBarForm.Exit1Click(Sender: TObject);
begin
  Application.Terminate;
end;

procedure TAppBarForm.FormCreate(Sender: TObject);
```

```
begin
  FLastChecked := Top1;
end;

procedure TAppBarForm.FormEdgeChanged(Sender: TObject);
begin
  MoveButtons;
end;

 end.
```

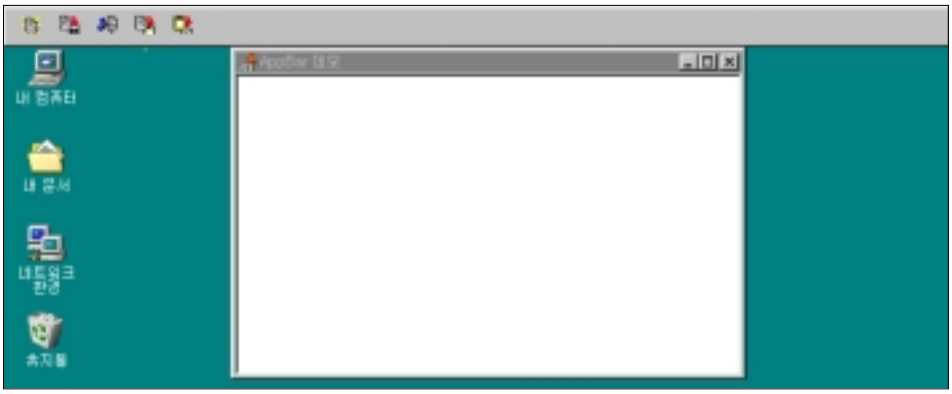


**그림 24.5**
앱바가 활성화된 모습

# 바로 가기 만들기

윈도우즈 쉘은 다양한 인터페이스를 제공하며, 개발자는 그 인터페이스들을 이용해서 쉘의 여러 요소들이나 기능들을 사용한다. 델파이의 경우, 쉘이 제공하는 여러 인터페이스들이 ShlObj 유닛에 정의되어 있다. 각각의 인터페이스들을 모두 설명하자면 새로운 책을 하나 더 써야 할지도 모르므로 이번 장에서는 가장 유용한(그리고 가장 자주 쓰이는) 인터페이스인 IShellLink에 대해서 알아보겠다.

IShellLink는 바로 가기(shell link 또는 shortchut) 항목을 만들기 위한 인터페이스이다. 바로 가기가 어떤 것인지는 모두 다 알고 있을 것이다. 바탕 화면의 아이콘들이나 시작 메뉴의 여러 아이콘들은 대부분 바로 가기 항목들이다. 또한, 어떤 항목을 마우스의 오른쪽 버튼을 클릭해서 나오는 팝업 메뉴에 있는 '보내기'라던가 시작 메뉴의 '문서' 같은 메뉴 항목들도 바로 가기를 이용한다. IShellLink 인터페이스는 ShlObj에 다음과 같이 정의되어 있다.

```
const

type
  IShellLinkA = interface(IUnknown) { sl }
    [SID_IShellLinkA]
    function GetPath(pszFile: PAnsiChar; cchMaxPath: Integer;
      var pfd: TWin32FindData; fFlags: DWORD): HResult; stdcall;
    function GetIDList(var ppidl: PItemIDList): HResult; stdcall;
    function SetIDList(pidl: PItemIDList): HResult; stdcall;
    function GetDescription(pszName: PAnsiChar; cchMaxName: Integer):
HResult; stdcall;
    function SetDescription(pszName: PAnsiChar): HResult; stdcall;
    function GetWorkingDirectory(pszDir: PAnsiChar; cchMaxPath: Integer):
HResult; stdcall;
    function SetWorkingDirectory(pszDir: PAnsiChar): HResult; stdcall;
    function GetArguments(pszArgs: PAnsiChar; cchMaxPath: Integer):
HResult; stdcall;
    function SetArguments(pszArgs: PAnsiChar): HResult; stdcall;
    function GetHotkey(var pwHotkey: Word): HResult; stdcall;
    function SetHotkey(wHotkey: Word): HResult; stdcall;
    function GetShowCmd(out piShowCmd: Integer): HResult; stdcall;
    function SetShowCmd(iShowCmd: Integer): HResult; stdcall;
    function GetIconLocation(pszIconPath: PAnsiChar; cchIconPath: Integer;
      out piIcon: Integer): HResult; stdcall;
    function SetIconLocation(pszIconPath: PAnsiChar; iIcon: Integer):
HResult; stdcall;
    function SetRelativePath(pszPathRel: PAnsiChar; dwReserved: DWORD):
HResult; stdcall;
    function Resolve(Wnd: HWND; fFlags: DWORD): HResult; stdcall;
    function SetPath(pszFile: PAnsiChar): HResult; stdcall;
  end;
```



IShellLink와 그의 모든 메소드들에 대해서는 Win32 도움말에 자세히 나와 있으므로, 이번 장에서는 필요한 부분만 설명하기로 한다.

## IShellLink 인스턴스 얻기

앞에서 나온 트레이 통지 아이콘이나 앱바는 API 함수를 사용했던 것에 비해 바로가기는 COM의 인터페이스를 통해서 다룬다. IShellLink 인터페이스는 윈도우즈 쉘이 구현하고 있으며, COM 함수인 CoCreateInstance() 함수를 이용해서 인터페이스 인스턴스를 얻어야 한다. 다음은 IShellLink의 인스턴스를 얻는 예이다.

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



COM이나 OLE 함수들을 사용하기 전에는 CoInitialize() 함수를 호출해서 COM 라이브러리를 활성화해야 한다는 점을 명심하기 바란다. 그리고 COM을 모두 사용한 후에는 CoUninitalize()를 호출해서 마무리를 해줘야 한다. 단, ComObj를 사용하는 델파이 애플리케이션의 경우, Application.Initialize()에서 자동적으로 CoInitialize()가 호출되며, CoUninitialize()는 ComObj 유닛의 finalization부에서 자동적으로 호출되므로 크게 신경 쓰지 않아도 된다.

## IShellLink의 사용법

윈도우즈 사용자라면 누구나 한 번쯤은 바탕화면이나 탐색기 등에서 바로 가기를 만들어 본 적이 있을 것이다. 바로 가기의 실제 모습은 확장자가 lnk인 특별한 종류의 파일이다. 윈도우즈가 부팅되면 윈도우즈는 여러 쉘 폴더에서 LNK 파일들을 찾는다. 쉘 폴더란 특별 폴더라고도 하며, '시작'(\시작 메뉴)이나 '문서'(\Recent)와 같은 특별한 용도의 폴더들을 말한다. 쉘 폴더의 이름(영문)과 실제 폴더 위치는 시스템 레지스트리에 등록되어 있다(대부분 HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders키 아래에 등록되어 있다).

어떤 특별한 쉘 폴더에 바로 가기를 만든다는 것은 결국 쉘 폴더의 실제 디렉토리에 LNK 파일을 만든다는 뜻이 된다. 그런데 레지스트리를 뒤지지 않고도 쉘 폴더의 실제 경로를 알아낼 수 있다. SHGetSpecialFolderPath()라는 함수를 이용하면 되는데, 이 함수의 선언은 다음과 같다.

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

hwndOwner 인자는 이 함수가 띄울 대화상자들의 소유자 역할을 할 윈도우의 핸들이다.

lpszPath 인자는 널 종료 문자열 버퍼를 가리키는 포인터이다. 함수가 성공적으로 수행되면 그 문자열 버퍼에 경로 이름이 들어간다. 문자열 버퍼의 최대 크기는 MAX_PATH 문자이다.

nFolder는 경로를 알아내고자 하는 특별 폴더의 ID이다. 표 24.4는 이 인자에 넣을 수 있는 값들을 정리한 것이다.

fCreate 인자는 폴더가 없는 경우에 그 폴더를 새로 만들 것인지를 뜻하는 플래그이다.

표 24.4 nFolder에 넣을 수 있는 값들

| 상 수 | 의 미 |
|---|---|
| CSIDL_ALTSTARTUP | 지역화 되지 않은(nonlocalized) '시작 프로그램(StartUp)' 그룹에 해당하는 디렉토리 |
| CSIDL_APPDATA | 애플리케이션 관련 데이터의 범용 저장소 역할을 하는 디렉토리 |
| CSIDL_BITBUCKET | '휴지통'에 담긴 파일 객체들이 있는 디렉토리. 이 디렉토리의 위치는 레지스트리에 저장되어 있지 않다. 일반적인 경우, 각 드라이브의 Recycled 디렉토리가 휴지통 디렉토리이다. 디렉토리 속성을 '숨김'과 '시스템'으로 설정해서 일반 사용자들로부터 감춰 놓았을 뿐이다. |
| CSIDL_COMMON_ALTSTARTUP | 지역화되지 않은 '시작 프로그램' 항목에 해당하는 디렉토리(모든 사용자들에 적용) |
| CSIDL_COMMON_DESKTOPDIRECTORY | 바탕화면에 나타나는 파일이나 폴더들이 들어 있는 디렉토리(모든 사용자들에 적용) |
| CSIDL_COMMON_FAVORITES | '즐겨 찾기(Favorites)'에 해당하는 디렉토리(모든 사용자들에 적용) |
| CSIDL_COMMON_PROGRAMS | 시작 메뉴의 '프로그램' 항목에 해당하는 디렉토리(모든 사용자들에 적용) |
| CSIDL_COMMON_STARTMENU | '시작(Start)' 메뉴에 해당하는 디렉토리(모든 사용자들에 적용) |
| CSIDL_COMMON_STARTUP | '시작 프로그램' 항목에 해당하는 디렉토리(모든 사용자들에 적용) |
| CSIDL_CONTROLS | 제어판 애플릿들에 해당하는 아이콘들이 담긴 가상 폴더 |
| CSIDL_COOKIES | 인터넷 쿠키들이 저장되는 디렉토리 |
| CSIDL_DESKTOP | 이름 공간(namespace)의 루트에 있는 윈도우즈 바탕화면(Desktop) 폴더(가상 폴더임) |
| CSIDL_DESKTOPDIRECTORY | '바탕 화면'에 담긴 파일들이 저장되는 실제 디스크 디렉토리 |
| CSIDL_DRIVES | 지역 컴퓨터를 구성하는 모든 것들(저장 장치, 프린터, 제어판 등)을 담고 있는 '내 컴퓨터(My Computer)'에 해당하는 가상 폴더. 연결된 네트웍 드라이브들도 포함할 수 있다. |
| CSIDL_FAVORITES | '즐겨 찾기(Favorites)'에 해당하는 디렉토리 |
| CSIDL_FONTS | '글꼴'에 해당하는 가상 폴더 |
| CSIDL_HISTORY | 인터넷 캐시가 담겨 있는 '열어 본 URL 폴더(History)'에 해당하는 디렉토리 |
| CSIDL_INTERNET | '인터넷'에 해당하는 뜻하는 가상 폴더 |
| CSIDL_INTERNET_CACHE | 임시적인 인터넷 파일들이 저장되는 '인터넷 캐시 폴더'에 해당하는 디렉토리 |
| CSIDL_NETHOOD | '네트웍 환경(Network Neighborhood)'에 나타나는 객체들이 저장되는 실제 디렉토리 |
| CSIDL_NETWORK | 네트웍 계층 구조의 최상위를 뜻하는 '네트웍 환경' 가상 폴더 |
| CSIDL_PERSONAL | '내 문서'에 해당하는 디렉토리 |
| CSIDL_PRINTERS | 설치된 프린터들을 담고 있는 가상 폴더 |
| CSIDL_PRINTHOOD | 프린터 연결들이 저장되는 디렉토리 |
| CSIDL_PROGRAMS | 시작 메뉴의 '프로그램' 항목에 해당하는 디렉토리 |
| CSIDL_RECENT | 시작 메뉴의 '문서' 항목에 해당하는 디렉토리 |
| CSIDL_SENDTO | 팝업 메뉴의 '보내기' 항목에 해당하는 디렉토리 |
| CSIDL_STARTMENU | '시작' 메뉴에 해당하는 디렉토리 |
| CSIDL_STARTUP | '시작 프로그램' 항목에 해당하는 디렉토리. 사용자가 시스템에 로그인하거나(윈도우즈 NT) 시스템이 시동되면(윈도우즈 95/98) '시작 프로그램' 그룹에 있는 항목들이 자동으로 실행된다. |
| CSIDL_TEMPLATES | 팝업 메뉴의 '새로 만들기'에 나타나는 템플릿들이 저장되는 디렉토리 |

### 바로 가기 만들기

IShellLink 인터페이스는 바로 가기 객체를 캡슐화한 것이지만, 이 인터페이스의 정의 자체에는 디스크상의 파일을 다루는 기능이 정의되어 있지 않다. 다만, IShellLink 인터페이스를 구현하려면 IPersistFile 인터페이스를 지원해야 하는데, IPersistFile 인터페이스에 디스크 파일 관련 메소드들이 정의되어 있다. IPersistFile의 정의는 ActiveX.pas에 다음과 같이 되어 있다.

```
type
  IPersistFile = interface(IPersist)
    ['{0000010B-0000-0000-C000-000000000046}']
    function IsDirty: HResult; stdcall;
    function Load(pszFileName: POleStr; dwMode: Longint): HResult;
      stdcall;
    function Save(pszFileName: POleStr; fRemember: BOOL): HResult;
      stdcall;
    function SaveCompleted(pszFileName: POleStr): HResult;
      stdcall;
    function GetCurFile(out pszFileName: POleStr): HResult;
      stdcall;
  end;
```



Win32 도움말에 IPersistFile이나 그의 메소드들에 대한 자세한 내용이 나와 있다.

IShellLink를 구현하는 클래스는 IPersistFile도 구현하므로 IShellLink 인스턴스에 As 연산자를 적용해서(내부적으로는 QueryInterface가 수행된다) IPersistFile 인스턴스를 얻을 수 있다. 다음이 그 예이다.

```
var
  SL: IShellLink;
  PF: IPersistFile;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
      IShellLink, SL);
  // SL과 PF를 사용한다.
end;
```

이전에도 말했지만, 일단 인터페이스 인스턴스를 얻은 다음부터는 보통 오브젝트 파스칼 객체들과 마찬가지 방식으로 속성이나 메소드를 사용하면 된다. 다음은 메모장 프로그램에 대한 바로 가기를 만드는 예이다.

```
procedure MakeNotepad;
const
  // 주의 : 다음 경로에 있다고 가정한다.
  AppName := 'c:\windows\notepad.exe';
var
  SL: IShellLink;
  PF: IPersistFile;
  LnkName: WideString;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
      IShellLink, SL);
  { IShellLink를 구현한 클래스는 IPersistFile도 구현하고 있어야 한다. }
  PF := SL as IPersistFile;
  OleCheck(SL.SetPath(PChar(AppName))); // 바로 가기의 경로 이름을 설정
  { 바로 가기 파일의 경로 이름을 설정한다. }
  LnkName := GetFolderLocation('Desktop') + '\' +
      ChangeFileExt(ExtractFileName(AppName), '.lnk');
  PF.Save(PWideChar(LnkName), True);         // 바로 가기 파일을 저장한다.
end;
```

위의 프로시저는 IShellLink의 SetPath() 메소드를 이용해서 바로 가기가 가리킬 실행 파일(이번 경우 메모장)이나 문서의 경로 이름을 설정한다. 그런 다음, 바로 가기 파일(.lnk)이 저장될 경로 이름을 설정한다. 이때 GetFolderLocation('Desktop')을 이용해

서 바탕화면에 해당하는 디렉토리 경로를 얻고, ChangeFileExt() 함수를 이용해서 메모장 실행 파일의 확장자를 .lnk로 변경한다(역주 : GetFolderLocation()은 위의 코드가 들어 있는 \Ch24\NpdLink\Main.pas 유닛에 정의된 함수이며, ChangeFileExt()는 SysUtils 유닛에 정의되어 있는 함수이다). 이렇게 해서 만들어진 바로 가기 파일의 경로 이름이 LnkName에 저장되며, IPersist의 Save() 메소드에 LnkName을 넣어서 바로 가기 파일을 저장한다. 이 함수가 종료되면 SL과 PF 인스턴스는 자동으로 해제된다.

### 바로 가기의 정보 얻기/설정

IShellLink에는 GetXXX()와 SetXXX() 형태의 메소드들이 정의되어 있는데, 그 메소드들은 바로 가기의 다양한 정보를 얻거나 설정하는 데 사용하는 메소드들이다. 그럼 바로 가기가 담고 있는 여러 가지 정보들을 일괄적으로 다루기 위한 레코드를 정의해 보자.

```
type
  TShellLinkInfo = record
       PathName: string;
       Arguments: string;
       Description: string;
       WorkingDirectory: string;
       IconLoaction: string;
       IconIndex: Integer;
       ShowCmd: Integer;
       Hotkey: Word;
  end;
```

윈도우즈 쉘에서 새 바로 가기를 만들 때 나오는 대화상자에서 설정하는 여러 가지 것들이 반영되어 있는 레코드임을 알 수 있을 것이다. 그럼 이 레코드를 사용해서 바로 가기의 여러 정보들을 얻거나 설정하는 함수들을 만들어 보자. 목록 24.5는 바로 가기를 다루는 여러 함수들이 정의되어 있는 WinShell.pas 유닛의 소스 코드이다(부록 CD의 \Utils 디렉토리에 있다).

목록 24.5 Utils\WinShell.pas

```
unit WinShell;

interface

uses SysUtils, Windows, Registry, ActiveX, ShlObj;

type
  EShellOleError = class(Exception);

  TShellLinkInfo = record
    PathName: string;
    Arguments: string;
    Description: string;
    WorkingDirectory: string;
    IconLocation: string;
    IconIndex: integer;
    ShowCmd: integer;
    HotKey: word;
  end;

  TSpecialFolderInfo = record
    Name: string;
    ID: Integer;
  end;

const
  SpecialFolders: array[0..29] of TSpecialFolderInfo = (
    (Name: 'Alt Startup'; ID: CSIDL_ALTSTARTUP),
    (Name: 'Application Data'; ID: CSIDL_APPDATA),
    (Name: 'Recycle Bin'; ID: CSIDL_BITBUCKET),
    (Name: 'Common Alt Startup'; ID: CSIDL_COMMON_ALTSTARTUP),
    (Name: 'Common Desktop'; ID: CSIDL_COMMON_DESKTOPDIRECTORY),
    (Name: 'Common Favorites'; ID: CSIDL_COMMON_FAVORITES),
    (Name: 'Common Programs'; ID: CSIDL_COMMON_PROGRAMS),
    (Name: 'Common Start Menu'; ID: CSIDL_COMMON_STARTMENU),
    (Name: 'Common Startup'; ID: CSIDL_COMMON_STARTUP),
```

```
    (Name: 'Controls'; ID: CSIDL_CONTROLS),
    (Name: 'Cookies'; ID: CSIDL_COOKIES),
    (Name: 'Desktop'; ID: CSIDL_DESKTOP),
    (Name: 'Desktop Directory'; ID: CSIDL_DESKTOPDIRECTORY),
    (Name: 'Drives'; ID: CSIDL_DRIVES),
    (Name: 'Favorites'; ID: CSIDL_FAVORITES),
    (Name: 'Fonts'; ID: CSIDL_FONTS),
    (Name: 'History'; ID: CSIDL_HISTORY),
    (Name: 'Internet'; ID: CSIDL_INTERNET),
    (Name: 'Internet Cache'; ID: CSIDL_INTERNET_CACHE),
    (Name: 'Network Neighborhood'; ID: CSIDL_NETHOOD),
    (Name: 'Network Top'; ID: CSIDL_NETWORK),
    (Name: 'Personal'; ID: CSIDL_PERSONAL),
    (Name: 'Printers'; ID: CSIDL_PRINTERS),
    (Name: 'Printer Links'; ID: CSIDL_PRINTHOOD),
    (Name: 'Programs'; ID: CSIDL_PROGRAMS),
    (Name: 'Recent Documents'; ID: CSIDL_RECENT),
    (Name: 'Send To'; ID: CSIDL_SENDTO),
    (Name: 'Start Menu'; ID: CSIDL_STARTMENU),
    (Name: 'Startup'; ID: CSIDL_STARTUP),
    (Name: 'Templates'; ID: CSIDL_TEMPLATES));

function CreateShellLink(const AppName, Desc: string; Dest: Integer):
 string;
function GetSpecialFolderPath(Folder: Integer; CanCreate: Boolean):
 string;
procedure GetShellLinkInfo(const LinkFile: WideString;
 var SLI: TShellLinkInfo);
procedure SetShellLinkInfo(const LinkFile: WideString; const SLI:
TShellLinkInfo);

implementation

uses ComObj;

function GetSpecialFolderPath(Folder: Integer; CanCreate: Boolean):
 string;
var
  FilePath: array[0..MAX_PATH] of char;
begin
```

```
  { 인자로 지정된 특별 폴더의 경로를 얻는다. }
  SHGetSpecialFolderPath(0, FilePath, Folder, CanCreate);
  Result := FilePath;
end;

function CreateShellLink(const AppName, Desc: string; Dest: Integer):
 string;
{ AppName으로 지정된 애플리케이션이나 문서에 대한 바로 가기를 만든다.
  Desc는 바로 가기 이름, 즉 바로 가기 아이콘에 나타낼 문자열이며, Dest는 바로
  가기 파일을 저장할 특별 폴더를 뜻하는 정수값이다. 각 정수값에 해당하는 특별
  폴더들은 이 유닛의 앞부분에 선언되어 있다(SpecialFolder 상수 배열).
  이 함수의 리턴값은 만들어진 바로 가기 파일의 전체 경로 이름이다. }
var
  SL: IShellLink;
  PF: IPersistFile;
  LnkName: WideString;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
    IShellLink, SL));
 { IShellLink를 구현한 클래스는 IPersistFile도 구현하고 있다. 따라서,
   IShellLink에 as를 적용하면 IPersistFile의 인스턴스를 얻을 수 있다. }
  PF := SL as IPersistFile;
 OleCheck(SL.SetPath(PChar(AppName))); // 바로 가기의 경로 이름을 설정
  if Desc <> '' then
    OleCheck(SL.SetDescription(PChar(Desc))); // 바로 가기 이름을 설정한다.
 { 바로 가기 파일의 경로 이름을 설정한다. }
  LnkName := GetSpecialFolderPath(Dest, True) + '\' +
             ChangeFileExt(AppName, 'lnk');
  PF.Save(PWideChar(LnkName), True);          // 바로 가기 파일을 저장한다.
  Result := LnkName;
end;

procedure GetShellLinkInfo(const LinkFile: WideString;
 var SLI: TShellLinkInfo);
{ 인자로 지정된 바로 가기의 정보를 얻는다. }
var
  SL: IShellLink;
  PF: IPersistFile;
  FindData: TWin32FindData;
  AStr: array[0..MAX_PATH] of char;
```

```
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
    IShellLink, SL));
 { IShellLink를 구현한 클래스는 IPersistFile도 구현하고 있다. 따라서,
   IShellLink에 as를 적용하면 IPersistFile의 인스턴스를 얻을 수 있다. }
  PF := SL as IPersistFile;
  { IPersistFile 인스턴스에 파일을 로드한다. }
  OleCheck(PF.Load(PWideChar(LinkFile), STGM_READ));
  { Resolve() 메소드를 호출해서 바로 가기의 정보를 얻을 준비를 한다. }
  OleCheck(SL.Resolve(0, SLR_ANY_MATCH or SLR_NO_UI));
  { 모든 정보를 얻는다. }
  with SLI do
  begin
    OleCheck(SL.GetPath(AStr, MAX_PATH, FindData, SLGP_SHORTPATH));
    PathName := AStr;
    OleCheck(SL.GetArguments(AStr, MAX_PATH));
    Arguments := AStr;
    OleCheck(SL.GetDescription(AStr, MAX_PATH));
    Description := AStr;
    OleCheck(SL.GetWorkingDirectory(AStr, MAX_PATH));
    WorkingDirectory := AStr;
    OleCheck(SL.GetIconLocation(AStr, MAX_PATH, IconIndex));
    IconLocation := AStr;
    OleCheck(SL.GetShowCmd(ShowCmd));
    OleCheck(SL.GetHotKey(HotKey));
  end;
end;

procedure SetShellLinkInfo(const LinkFile: WideString; const SLI:
TShellLinkInfo);
{ 인자로 지정된 바로 가기에 인자로 지정된 정보들을 설정한다.}
var
  SL: IShellLink;
  PF: IPersistFile;
begin
  OleCheck(CoCreateInstance(CLSID_ShellLink, nil, CLSCTX_INPROC_SERVER,
    IShellLink, SL));
 { IShellLink를 구현한 클래스는 IPersistFile도 구현하고 있다. 따라서,
   IShellLink에 as를 적용하면 IPersistFile의 인스턴스를 얻을 수 있다. }
  PF := SL as IPersistFile;
```

```
    { IPersistFile 인스턴스에 파일을 로드한다. }
    OleCheck(PF.Load(PWideChar(LinkFile), STGM_SHARE_DENY_WRITE));
    { Resolve() 메소드를 호출해서 바로 가기의 정보를 설정할 준비를 한다. }
    OleCheck(SL.Resolve(0, SLR_ANY_MATCH or SLR_UPDATE or SLR_NO_UI));
    { 모든 정보들을 설정한다. }
    with SLI, SL do
    begin
      OleCheck(SetPath(PChar(PathName)));
      OleCheck(SetArguments(PChar(Arguments)));
      OleCheck(SetDescription(PChar(Description)));
      OleCheck(SetWorkingDirectory(PChar(WorkingDirectory)));
      OleCheck(SetIconLocation(PChar(IconLocation), IconIndex));
      OleCheck(SetShowCmd(ShowCmd));
      OleCheck(SetHotKey(HotKey));
    end;
    PF.Save(PWideChar(LinkFile), True);   // 바로 가기 파일을 저장한다.
  end;

  end.
```

주석만으로는 IShellLink 인터페이스의 Resolve() 메소드에 대한 설명이 부족한 것 같다. 이 메소드는 IPersisFile 인스턴스에 로드된 바로 가기 파일에서 바로 가기에 관한 정보들을 뽑아서 IShellLink 인스턴스에 채워넣는 작업을 수행한다.



목록 24.5의 GetShellLinkInfo() 함수에서 지역 배열 변수 AStr를 주목하기 바란다. 임시로 사용할 변수에 string을 이용하는 대신 char의 배열을 사용한 이유는, string 변수에 SetLength()를 사용해서 메모리를 여러 번 할당하다 보면 애플리케이션의 힙이 좀 심하게 단편화될 수 있기 때문이다. Astr처럼 char 배열을 사용하면 그런 염려가 없다. 그리고 char 배열은 선언시에 길이를 한 번만 정해주면 되기 때문에 동적으로 할당/해제가 일어나는 string보다 더 빠르다.

## 예제 애플리케이션

그럼 위에서 나온 함수들을 실제로 사용하는 예제 애플리케이션을 살펴보자. 부록 CD의 ShellLink 프로젝트가 바로 그것인데, 이 프로젝트의 메인 폼이 그림 24.6에 나와 있다.

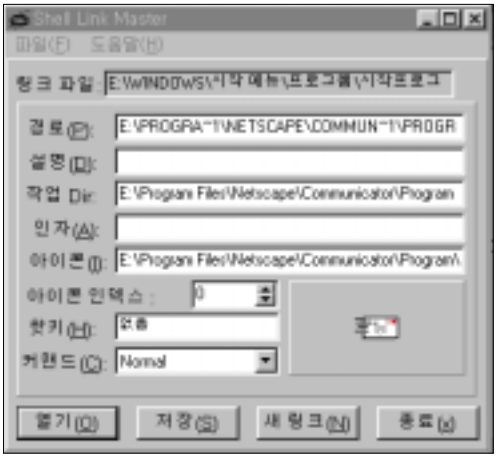



**그림 24.6**
ShellLink의 메인 폼

목록 24.6은 이 프로젝트의 메인 폼 유닛인 Main.pas의 소스 코드이고, 목록 24.7과 24.8은 대화상자로 사용할 두 폼의 유닛인 NewLinkU.pas와 PickU.pas의 소스 코드이다.

**목록 24.6** Ch24\Shellink\Main.pas

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ComCtrls, ExtCtrls, Spin, WinShell, Menus;

type
  TMainForm = class(TForm)
    Panel1: TPanel;
```

```
    btnOpen: TButton;
    edLink: TEdit;
    btnNew: TButton;
    btnSave: TButton;
    Label3: TLabel;
    Panel2: TPanel;
    Label1: TLabel;
    Label2: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    Label8: TLabel;
    Label9: TLabel;
    edIcon: TEdit;
    edDesc: TEdit;
    edWorkDir: TEdit;
    edArg: TEdit;
    cbShowCmd: TComboBox;
    hkHotKey: THotKey;
    speIcnIdx: TSpinEdit;
    pnlIconPanel: TPanel;
    imgIconImage: TImage;
    btnExit: TButton;
    MainMenu1: TMainMenu;
    File1: TMenuItem;
    Open1: TMenuItem;
    Save1: TMenuItem;
    NewLInk1: TMenuItem;
    N1: TMenuItem;
    Exit1: TMenuItem;
    Help1: TMenuItem;
    About1: TMenuItem;
    edPath: TEdit;
    procedure btnOpenClick(Sender: TObject);
    procedure btnNewClick(Sender: TObject);
    procedure edIconChange(Sender: TObject);
    procedure btnSaveClick(Sender: TObject);
    procedure btnExitClick(Sender: TObject);
```

```
    procedure About1Click(Sender: TObject);
  private
    procedure GetControls(var SLI: TShellLinkInfo);
    procedure SetControls(const SLI: TShellLinkInfo);
    procedure ShowIcon;
    procedure OpenLinkFile(const LinkFileName: String);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

uses PickU, NewLinkU, AboutU, CommCtrl, ShellAPI;

type
  THotKeyRec = record
    Char, ModCode: Byte;
  end;

procedure TMainForm.SetControls(const SLI: TShellLinkInfo);
{ SLI에 담긴 값들을 이용해서 메인 폼의 여러 컨트롤들을 갱신한다. }
var
  Mods: THKModifiers;
begin
  with SLI do
  begin
    edPath.Text := PathName;
    edIcon.Text := IconLocation;
     { 아이콘의 경로 이름이 설정되어 있지 않고 바로 가기가 실행 파일을 가리키는
       것이라면 그 실행 파일의 경로 이름을 아이콘의 경로 이름으로 한다. 실행 파일
       에는 하나 이상의 아이콘들이 들어 있을 수 있지만, 아이콘 경로 이름이 비어
       있는 경우에는 아이콘 번호는 의미가 없다. }
    if (IconLocation = '') and
      (CompareText(ExtractFileExt(PathName), 'EXE') = 0) then
      edIcon.Text := PathName;
    edWorkDir.Text := WorkingDirectory;
```

```
    edArg.Text := Arguments;
    speIcnIdx.Value := IconIndex;
    edDesc.Text := Description;
    { SW_* 상수들은 1부터 시작하므로 1을 빼준다. }
    cbShowCmd.ItemIndex := ShowCmd - 1;
    { 단축키의 아스키 문자는 하위 바이트에 들어 있다. }
    hkHotKey.HotKey := Lo(HotKey);
    { 상위 바이트에는 단축키의 조합 정보가(Alt나 Ctrl 등) 들어 있다. }
    Mods := [];
    if (HOTKEYF_ALT and Hi(HotKey)) <> 0 then include(Mods, hkAlt);
    if (HOTKEYF_CONTROL and Hi(HotKey)) <> 0 then include(Mods, hkCtrl);
    if (HOTKEYF_EXT and Hi(HotKey)) <> 0 then include(Mods, hkExt);
    if (HOTKEYF_SHIFT and Hi(HotKey)) <> 0 then include(Mods, hkShift);
    { 얻어낸 조합 정보를 설정한다. }
    hkHotKey.Modifiers := Mods;
  end;
  ShowIcon;
end;

procedure TMainForm.GetControls(var SLI: TShellLinkInfo);
{ 메인 폼의 컨트롤들에 설정된 값을 이용해서 SLI를 채운다. }
var
  CtlMods: THKModifiers;
  HR: THotKeyRec;
begin
  with SLI do
  begin
    PathName := edPath.Text;
    IconLocation := edIcon.Text;
    WorkingDirectory := edWorkDir.Text;
    Arguments := edArg.Text;
    IconIndex := speIcnIdx.Value;
    Description := edDesc.Text;
    { SW_* 상수들은 1부터 시작하므로 1을 빼준다. }
    ShowCmd := cbShowCmd.ItemIndex + 1;
     { 단축키를 얻는다. }
    word(HR) := hkHotKey.HotKey;
    { 조합키(Ctrl, Alt 등) 정보를 얻는다. }
    CtlMods := hkHotKey.Modifiers;
```

```
    with HR do begin
      ModCode := 0;
      if (hkAlt in CtlMods) then ModCode := ModCode or HOTKEYF_ALT;
      if (hkCtrl in CtlMods) then ModCode := ModCode or HOTKEYF_CONTROL;
      if (hkExt in CtlMods) then ModCode := ModCode or HOTKEYF_EXT;
      if (hkShift in CtlMods) then ModCode := ModCode or HOTKEYF_SHIFT;
    end;
    HotKey := word(HR);
  end;
end;

procedure TMainForm.ShowIcon;
{ 바로 가기의 아이콘을 뽑아서 imgIconImage에 표시한다. }
var
  HI: THandle;
  IcnFile: string;
  IconIndex: word;
begin
  { 아이콘 파일의 이름을 얻는다. }
  IcnFile := edIcon.Text;
  { 없으면 바로 가기가 가리키는 실행 파일을 이용한다. }
  if IcnFile = '' then
    IcnFile := edPath.Text;
  { 파일이 존재하는지 확인한다. }
  if FileExists(IcnFile) then
  begin
    IconIndex := speIcnIdx.Value;
    { 파일에서 아이콘을 뽑아낸다. }
    HI := ExtractAssociatedIcon(hInstance, PChar(IcnFile), IconIndex);
    { 아이콘 핸들을 imgIcomImage에 배정한다. }
    imgIconImage.Picture.Icon.Handle := HI;
  end;
end;

procedure TMainForm.OpenLinkFile(const LinkFileName: string);
{ 바로 가기 파일을 열어서 정보들을 얻은 다음, 메인 폼의 컨트롤들에 표시한다. }
var
  SLI: TShellLinkInfo;
begin
```

```
  edLink.Text := LinkFileName;
  try
    GetShellLinkInfo(LinkFileName, SLI);
  except
    on EShellOleError do
      MessageDlg('Error occurred while opening link', mtError, [mbOk], 0);
  end;
  SetControls(SLI);
end;

procedure TMainForm.btnOpenClick(Sender: TObject);
{ OpenBtn 버튼의 OnClick 이벤트 처리 메소드 }
var
  LinkFile: String;
begin
  if GetLinkFile(LinkFile) then
    OpenLinkFile(LinkFile);
end;

procedure TMainForm.btnNewClick(Sender: TObject);
{ NewBtn 버튼의 OnClick 이벤트 처리 메소드 }
var
  FileName: string;
  Dest: Integer;
begin
  if GetNewLinkName(FileName, Dest) then
    OpenLinkFile(CreateShellLink(FileName, '', Dest));
end;

procedure TMainForm.edIconChange(Sender: TObject);
{ edIcon과 speIcnIdx의 OnChange 이벤트 처리 메소드  }
begin
  ShowIcon;
end;

procedure TMainForm.btnSaveClick(Sender: TObject);
{ SaveBtn 버튼의 OnClick 이벤트 처리 메소드 }
var
  SLI: TShellLinkInfo;
```

```
begin
  GetControls(SLI);
  try
    SetShellLinkInfo(edLink.Text, SLI);
  except
    on EShellOleError do
      MessageDlg('Error occurred while setting info', mtError, [mbOk], 0);
  end;
end;

procedure TMainForm.btnExitClick(Sender: TObject);
{ ExitBtn 버튼의 OnClick 이벤트 처리 메소드 }
begin
  Close;
end;

procedure TMainForm.About1Click(Sender: TObject);
{ Help|About 메뉴 항목의 OnClick 이벤트 처리 메소드 }
begin
  AboutBox;
end;

 end.
```

목록 24.7 Ch24\Shellink\NewLinkU.pas

```
unit NewLinkU;

interface

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

type
```

```
  TNewLinkForm = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    edLinkTo: TEdit;
    btnOk: TButton;
    btnCancel: TButton;
    cbLocation: TComboBox;
    sbOpen: TSpeedButton;
    OpenDialog: TOpenDialog;
    procedure sbOpenClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  end;

function GetNewLinkName(var LinkTo: string; var Dest: Integer): Boolean;

implementation

uses WinShell;

{$R *.DFM}

function GetNewLinkName(var LinkTo: string; var Dest: Integer): Boolean;
{ 새 바로 가기의 파일 이름과 대상 폴더를 얻는다. Result가 True인 경우에만
  인자들의 값을 변경한다. }
begin
  with TNewLinkForm.Create(Application) do
  try
    cbLocation.ItemIndex := 0;
    Result := ShowModal = mrOk;
    if Result then
    begin
      LinkTo := edLinkTo.Text;
      Dest := cbLocation.ItemIndex;
    end;
  finally
    Free;
  end;
end;
```

```
procedure TNewLinkForm.sbOpenClick(Sender: TObject);
begin
  if OpenDialog.Execute then
    edLinkTo.Text := OpenDialog.FileName;
end;

procedure TNewLinkForm.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  for I := Low(SpecialFolders) to High(SpecialFolders) do
    cbLocation.Items.Add(SpecialFolders[I].Name);
end;

end.
```




Ch24\Shellink\PickU.pas

```
unit PickU;

interface

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

type
  TLinkForm = class(TForm)
    lbLinkFiles: TFileListBox;
    btnOk: TButton;
    btnCancel: TButton;
    cbLocation: TComboBox;
    Label1: TLabel;
    procedure lbLinkFilesDblClick(Sender: TObject);
    procedure cbLocationChange(Sender: TObject);
```

```
    procedure FormCreate(Sender: TObject);
  end;

function GetLinkFile(var S: String): Boolean;

implementation

{$R *.DFM}

uses WinShell, ShlObj;

function GetLinkFile(var S: String): Boolean;
{ 바로 가기 파일 이름을 S에 넣어서 돌려준다. Result가 True일 때에만 S를
  변경한다. }
begin
  with TLinkForm.Create(Application) do
    try
          { 일단 첫 번째 특별 폴더를 선택한다. }
      cbLocation.ItemIndex := 0;
      cbLocationChange(nil);
          { 폼을 모달로 띄운다. }
      Result := ShowModal = mrOk;
      { 바로 가기 파일의 전체 경로 이름을 S에 넣는다. }
      if Result then
        S := lbLinkFiles.Directory + '\' +
                  lbLinkFiles.Items[lbLinkFiles.ItemIndex];
    finally
      Free;
    end;
end;

procedure TLinkForm.lbLinkFilesDblClick(Sender: TObject);
begin
  ModalResult := mrOk;
end;

procedure TLinkForm.cbLocationChange(Sender: TObject);
var
  Folder: Integer;
```

```
begin
  { 선택된 위치의 경로를 얻는다. }
  Folder := SpecialFolders[cbLocation.ItemIndex].ID;
  lbLinkFiles.Directory := GetSpecialFolderPath(Folder, False);
end;

procedure TLinkForm.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  for I := Low(SpecialFolders) to High(SpecialFolders) do
    cbLocation.Items.Add(SpecialFolders[I].Name);
end;

end.
```

# 쉘 확장 기능

현대적 운영체제의 특징 중 하나는 쉘, 즉 최종 사용자가 운영체제(특히, 파일 객체들)를 다루기 위해 접하는 사용자 인터페이스 요소들을 얼마든지 직접 다룰 수 있도록 하는 것이다. 쉘 확장 기능(shell extension)은 윈도우즈의 쉘을 구성하는 요소이며, 개발자 역시 적절한 COM 인터페이스를 통해서 자신만의 쉘 확장 기능을 구현, 등록할 수 있다. 쉘 확장 기능들은 프로세스 내부 COM 서버로 구현해야 한다.




쉘 확장 기능들은 COM 서버이므로 쉘 확장 기능을 제대로 다루려면 COM에 대한 기본적인 이해가 있어야 한다. 만일 23장, "COM과 ActiveX"을 제대로 읽지 않은 독자라면 다시 한 번 읽어보기 바란다.

쉘의 여러 측면에 해당하는 여러 종류의 쉘 확장 기능들이 있는데, 흔히 "XXX 핸들러" 같은 이름으로 부르기도 하며, 각 "핸들러"들은 하나나 그 이상의 COM 인터페이스들을 구현한 것이다. 다음은 쉘이 지원하는 여러 가지 쉘 확장 기능들에 대한 설명이다.

- 복사 훅 핸들러(copy hook handler)는 ICopyHook 인터페이스를 구현한다. 이 쉘 확장 기능은 폴더가 복사/삭제/이동되거나 이름이 변경될 때 작동한다. 이것을 이용하면 특정 폴더를 삭제하지 못하게 하는 등의 작업을 할 수 있다.
- 문맥 메뉴 핸들러(context menu handler)는 IContextMenu와 IShellExtInit 인터페이스를 구현한다. 쉘에서 어떤 파일을 오른쪽 클릭할 때 나타나는 메뉴 항목들에 관련된 확장 기능이다.
- 드래그 앤 드롭 핸들러(drag-and-drop handler) 역시 IContextMenu와 IShell ExtInit 인터페이스를 구현한다. 이 핸들러의 구현은 문맥 메뉴 핸들러와 거의 비슷하나 사용자가 어떤 객체를 끌어서 새로운 위치에 놓는 작업을 처리하기 위한 확장 기능이다.
- 아이콘 핸들러(icon handler)는 IExtractIcon과 IPersistFile 인터페이스를 구현한다. 아이콘 핸들러는 같은 종류의 파일 객체들을 서로 다른 아이콘으로 표시하는 등의 작업을 처리하기 위한 것이다.
- 등록 정보 핸들러(property sheet handler)는 IShellPropSheetExt와 IShell ExtInit 인터페이스를 구현한다. 파일 등의 어떤 쉘 객체의 등록 정보 대화상자에 나타나는 정보들을 다루는 쉘 확장 기능이다.
- 드롭 대상 핸들러(drop target handler)는 IDropTarget과 IPersistFile 인터페이스를 처리한다. 어떤 쉘 객체를 다른 객체에 끌어다 놓을 때 발생하는 일들을 처리하기 위한 것이다.
- 데이터 객체 핸들러(data object handler)는 IDataObject와 IPersistFile 인터페이스를 구현한다. 파일을 끌어다 놓거나 파일을 복사하고 붙여넣는 작업에 관련된 것들을 처리하기 위한 것이다.

## 쉘 확장 기능의 디버깅

쉘 확장 기능은 애플리케이션의 프로세스가 아니라 쉘 자체의 프로세스에서 실행되므로 일반적인 방식으로는 디버깅하기가 어렵다. 쉘 프로세스를 디버거에 붙여야만 디버깅이 가능한 것이다.
그렇다면 윈도우즈 쉘의 프로세스를 디버거에 붙이는 방법이 존재할까? 쉘의 실체는 explorer.exe라는 하나의 실행 파일(보통 실행 파일과 그리 다르지 않다)이다. 그러나 explorer는 일반 실행 파일과는 좀 다른 특성을 가지고 있다는 점도 기억하기 바란다. 윈도우즈가 부팅되면 explorer.exe가 자동으로 실행되는데, 첫 번째로 실행된 explorer.exe는 쉘을 띄우지만, 그 이후에 사용자가 직접 explorer.exe를 실행시키면 쉘이 하나 더 만들어지는 것이 아니라 "탐색기" 창이 쉘에 추가될 뿐이다.

쉘은 윈도우즈 자체가 아니라 하나의 실행 파일이 띄운 어떤 것이므로 윈도우즈를 종료시키지 않고도 쉘만 종료시킬 수 있다. 다음은 델파이에서 쉘 확장 기능 프로젝트를 디버깅하는 방법인데, 쉘만 종료시키는 기법을 사용한다.

1. Run | Parameter 대화상자에서 디버깅하고자 하는 쉘 확장 기능 프로젝트의 호스트 애플리케이션을 explore.exe로 지정한다. 반드시 전체 경로 이름을 지정해야 한다(즉 c:\windows\explorer.exe).
2. 윈도우즈 쉘의 시작 메뉴에서 시스템 종료를 선택한다. 그러면 Windows 종료 대화상자가 나타난다.
3. 이때 〈Ctrl+Alt+Shift〉키를 누른 채로 취소 버튼을 클릭한다. 그러면 윈도우즈가 종료되지 않은 상태에서 쉘만 종료된다(작업 표시줄이나 바탕 화면의 아이콘들이 사라지면 쉘이 종료된 것이다).
4. 〈Alt+Tab〉키를 이용해서 델파이로 다시 돌아온 다음에 쉘 확장 기능을 실행시킨다. 그러면 explorer.exe가 실행되는데, 쉘이 없는 상태에서 실행되는 것이므로 쉘을 다시 만든다. 결과적으로, 새로 만들어진 쉘은 델파이 디버거의 프로세스 안에서 실행되게 된다. 따라서, 보통 프로젝트를 디버깅하듯이 쉘 확장 기능에 중단점 등을 설정할 수 있다.
5. 쉘이 없으면 작업 표시줄도 없고 시작 메뉴도 없으므로 시작 메뉴의 프로그램 종료로 윈도우즈를 종료할 수는 없다. 쉘이 없는 상태에서 윈도우즈를 종료하려면 〈Ctrl+Alt+Del〉키를 눌러서 프로그램 종료 대화상자를 띄운 후에 시스템 종료 버튼을 클릭하면 된다.

그럼 위에서 말한 여러 가지 핸들러들 중 복사 훅 핸들러, 문맥 메뉴 핸들러, 그리고 아이콘 핸들러 등을 다루는 예제들을 살펴보자.

## COM 객체 마법사

쉘 확장 기능의 여러 핸들러들에 대한 내용으로 들어가기 전에 우선 예제 프로젝트를 만들 때 공통적으로 적용되는 부분에 대해서 이야기를 해야 할 것 같다. 쉘 확장 기능은 프로세스 내부 COM 서버이므로 쉘 확장 기능을 만들 때에는 델파이가 제공하는 COM 서버 지원 기능을 사용하는 것이 좋다. 다음은 이번 장에 나오는 예제들을 비롯해서 쉘 확장 기능 프로젝트를 만들 때 공통적으로 적용되는 과정이다.

1. New Items 대화상자의 ActiveX 페이지에서 ActiveX Library를 선택한다. 그러면 새로운 COM 서버 DLL 프로젝트가 생성된다.

2. New Items 대화상자의 ActiveX 페이지에서 COM Object를 선택하면 COM Object Wizard 대화상자가 나타난다. 이 대화상자에서 쉘 확장 기능의 이름과 설명을 입력한 다음, Apartment threading 모델을 선택한다. OK를 클릭하면 COM 객체에 대한 뼈대 코드가 포함된 새 유닛이 만들어진다.

## 복사 훅 핸들러

어떤 폴더의 복사, 삭제, 이동, 이름 변경 같은 사건이 일어나면 복사 훅 핸들러에게 통지가 온다. 복사 훅 핸들러의 구현에 따라서는 특정 작업이 일어나지 못하게 막을 수도 있다. 단, 복사 훅 핸들러는 폴더 단위나 프린터 객체에 수행되는 작업에서만 호출되며, 개별 파일에 대한 작업에서는 호출되지 않는다.

복사 훅 핸들러를 만드는 첫 번째 단계는 TComObject를 상속하는 객체에서 ICopyHook 인터페이스를 구현하는 것이다. ICopyHook 인터페이스는 ShlObj 유닛에 다음과 같이 정의되어 있다.

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

### CopyCallback() 메소드

위의 정의에서도 알 수 있듯이 ICopyHook에는 CopyCallback()이라는 메소드 밖에 없다. 이 메소드는 쉘의 어떤 폴더에 대해 복사/삭제/이동/이름 변경 등의 작업이 일어나려고 할 때 호출되는 콜백 함수이다. 그럼 각각의 인자에 대해 알아보자.

Wnd는 복사 훅 핸들러가 표시할 윈도우들의 부모 역할을 할 윈도우의 핸들이다. wFunc 인자는 수행하려고 하는 작업을 뜻한다. 이 인자에 사용할 수 있는 값은 표 24.5에 나와 있다.

표 24.5 Copycallback()의 wFunc에 들어갈 수 있는 값들

| 상수 | 값 | 의미 |
|---|---|---|
| FO_COPY | $2 | pszSrcFile로 지정된 파일이 pszDestFile로 지정된 폴더에 복사되려고 함. |
| FO_DELETE | $3 | pszSrcFile로 지정된 파일이 삭제되려고 함. |
| FO_MOVE | $1 | pszSrcFile로 지정된 파일이 pszDestFile로 지정된 폴더로 이동하려고 함. |
| FO_RENAME | $4 | pszSrcFile로 지정된 파일의 이름이 변경되려고 함. |
| PO_DELETE | $13 | pszSrcFile로 지정된 프린터가 삭제되려고 함. |
| PO_PORTCHANGE | $20 | 프린터 포트를 변경하려고 함. 이때 pszSrcFile에는 널 문자로 분리된 두 개의 널 종료 문자열이 들어 있다. 첫 번째 문자열은 프린터 이름이며, 또 한 문자열은 포트 이름들이다. pszSrcFile 들어 있는 포트 이름은 현재 프린터 포트의 이름이며, 바꾸고자 하는 새 프린터 포트 이름은 pszDestFile에 들어 있다. |
| PO_RENAME | $14 | pszSrcFile로 지정된 프린터의 이름이 변경되려고 함. |
| PO_REN_PORT | $34 | PO_RENAME과 PO_PORTCHANGE가 합쳐진 것이다. |

wFlags에는 작업의 특성을 뜻하는 플래그들이 들어 있다. 이 인자에 포함될 수 있는 플래그들은 표 24.6과 같다.

표 24.6 CopyCallback()의 wFlags 인자에 포함될 수 있는 플래그들

| 상수 | 값 | 의미 |
|---|---|---|
| FOF_ALLOWUNDO | $40 | 실행 취소(Undo) 정보를 보존한다(가능한 경우에만). |
| FOF_MULTIDESTFILES | $1 | SHFileOperation() 함수가 대상 파일들을 하나 이상 지정했음(원본 파일 하나당 하나의 대상 파일). 일반적으로, 복사 훅 핸들러는 이 값을 무시한다. |
| FOF_NOCONFIRMATION | $10 | 모든 대화상자에서 "모두(Yes to All)"을 선택하도록 한다. |
| FOF_NOCONFIRMMKDIR | $200 | 수행할 작업이 새 디렉토리를 만들어야 하는 경우, 사용자에게 확인하지 않고 디렉토리를 만든다. |
| FOF_RENAMEONCOLLISION | $8 | 복사, 이동, 이름 변경 작업을 수행할 때 대상 디렉토리에 같은 이름의 파일이 있으면 새로운 이름을 자동으로 만든다("사본 - ..." 등). |
| FOF_SILENT | $4 | 진행 상태 대화상자(progress dialog)를 표시하지 않는다. |
| FOF_SIMPLEPROGRESS | $100 | 진행 상태 대화상자를 표시하되 작업중인 파일 이름은 표시하지 않는다. |

pszSourceFile은 원본 폴더의 이름이며, dwSrcAttrib에는 원본 폴더의 특성들(읽기 전용, 숨김 등)이 들어 있다. 그리고 pszDestFile은 대상 폴더의 이름이며, dwDestAttrib에는 대상 폴더의 특성들이 들어 있다.

다른 인터페이스 메소드들과는 달리 이 메소드는 OLE 결과 코드를 돌려주지 않는다. 이 메소드가 돌려주는 리턴값의 의미는 표 24.7과 같다. 이 값들은 Windows 유닛에 정의되어 있다.

표 24.7 Copycallback()의 리턴값

| 상수 | 값 | 의미 |
|---|---|---|
| IDYES | 6 | 작업을 허용함. |
| IDNO | 7 | 이 파일에 대해서는 작업을 허용하지 않음. 다른 작업은 계속 진행할 수 있다(여러 개의 파일들을 복사하는 경우를 상상하기 바란다). |
| IDCANCEL | 2 | 현재 작업뿐만 아니라 나머지 작업들도 모두 취소시킨다. |

### ICopyHook을 구현하는 TCopyHook 객체

다음은 ICopyHook 인터페이스를 구현하는 COM 객체 TCopyHook인데, CopyCallback()만 구현하는 간단한 객체이다.

```
type
  TCopyHook = class(TComObject, ICopyHook)
  protected
      function CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
      pszSrcFile: PAnsiChar;
      dwSrcAttribs: DWORD; pszDestFile: PAnsiChar; dwDestAttribs: DWORD):
UINT; stdcall;
end;
```

그럼 CopyCallback()을 아주 간단하게 구현해 보자. 그냥 메시지 상자를 띄워서 사용자에게 다시 확인을 받도록 하겠다. 조금만 응용하면 좀더 의미 있는 작업을 수행하게 개선할 수 있을 것이다. 그리고 표 24.7을 보면 알겠지만, CopyCallback()의 리턴값은 MessageBox()의 리턴값과 동일한 의미이므로 Result에 MessageBox()의 리턴값을 직접 넣어주면 된다.

```
function TCopyHook.CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
  pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD; pszDestFile: PAnsiChar;
  dwDestAttribs: DWORD): UINT;
const
  MyMessage: string = ' ''%s''를 정말 변경하실래요?';
begin
  // 작업을 진행할 것인지 확인한다.
  Result := MessageBox(Wnd, PChar(Format(MyMessage, [pszSrcFile])),
  'D4DG Shell Extension', MB_YESNO);
end;
```

Tip

델파이의 MessageDlg()나 ShowMessage() 대신 API 함수 MessageBox()를 사용한 이유는 성능과 크기 때문이다. MessageDlg()나 ShowMessage()를 사용하려면 Dialogs나 Forms 유닛을 포함시켜야 하므로 컴파일된 DLL의 크기도 커질 뿐만 아니라 속도도 감소된다. uses절에 Dialogs나 Forms 유닛이 포함되지 않은 상태에서 컴파일하면 예제 DLL의 크기는 70K 정도 밖에 안된다.

CopyCallback()을 구현하는 것만으로 TCopyHook 객체의 작성이 끝났다. ICopyHook 자체가 워낙 간단하기 때문이다. 인터페이스의 구현은 이렇게 간단하게 끝났지만 아직 이 쉘 확장 기능을 시스템 레지스트리에 등록하는 과정이 남아 있다.

### 등록

쉘 확장 기능은 COM 서버이므로 기본적인 COM 서버 등록 과정을 거쳐야 한다. 복사 훅 핸들러로서의 등록 과정은 레지스트리의 HKEY_CLASSES_ROOT\Directory\shellex\CopyHookHandlers 아래에 자신의 항목을 추가하는 것이다.

그리고 윈도우즈 NT의 경우에는 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Shell Extensions\Approved 아래에도 항목을 만들어야 한다.

레지스트리에 쉘 확장 기능을 등록하는 방법은 여러 가지가 있다. REG 파일을 사용해도 되고, 설치 프로그램을 사용해도 된다. 그리고 쉘 확장 기능 DLL 자체에 등록 루틴을 넣을 수도 있다. 가장 깔끔한 방법은 DLL 안에 스스로 자신을 등록하는 코드를 포함시키는 것이다. 추가적인 파일이 필요 없으므로 관리나 배포에 좋다.

23장에서 COM 객체들은 항상 클래스 팩토리로부터 생성된다고 했었다. 델파이로 COM 객체를 만드는 경우, 클래스 팩토리 객체는 자신이 만드는 COM 객체의 등록까지 책임을 진다. 만일 COM 객체가 또 다른 레지스트리 항목을 만들어야 한다면 (쉘 확장 기능처럼) 클래스 팩토리의 UpdateRegistry()를 적절히 재정의(override)해야 한다. 목록 24.9는 TCopyHook 객체를 정의하는 코드와 그 객체를 생성하는 클래스 팩토리를 적절히 재정의하는 코드가 들어 있는 CopyMain.pas 유닛의 전체 코드이다.

목록 24.9 CopyMain.pas의 소스 코드

```
unit CopyMain;

interface

uses Windows, ComObj, ShlObj;

type
  TCopyHook = class(TComObject, ICopyHook)
  protected
    function CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
          pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD;
          pszDestFile: PAnsiChar; dwDestAttribs: DWORD): UINT; stdcall;
  end;

  TCopyHookFactory = class(TComObjectFactory)
  protected
    function GetProgID: string; override;
    procedure ApproveShellExtension(Register: Boolean; const ClsID:
          string); virtual;
  public
    procedure UpdateRegistry(Register: Boolean); override;
  end;

implementation

uses ComServ, SysUtils, Registry;

{ TCopyHook }

// 폴더에 대한 어떤 작업이 발생하면 쉘이 이 함수를 호출한다.
function TCopyHook.CopyCallback(Wnd: HWND; wFunc, wFlags: UINT;
  pszSrcFile: PAnsiChar; dwSrcAttribs: DWORD; pszDestFile: PAnsiChar;
  dwDestAttribs: DWORD): UINT;
const
  MyMessage: string = '"%s"를 정말 변경하실래요?';
begin
```

```
  // 사용자에게 작업을 확인받는다.
  Result := MessageBox(Wnd,  PChar(Format(MyMessage, [pszSrcFile])),
    'D5DG 셸 확장', MB_YESNO);
end;

{ TCopyHookFactory }

function TCopyHookFactory.GetProgID: string;
begin
  // 쉘 확장 기능에는 ProgID가 필요하지 않다.
  Result := '';
end;

procedure TCopyHookFactory.UpdateRegistry(Register: Boolean);
var
  ClsID: string;
begin
  ClsID := GUIDToString(ClassID);
  inherited UpdateRegistry(Register);
  ApproveShellExtension(Register, ClsID);
  if Register then
    // 레지스트리의 CopyHookHandlers 항목에 쉘 확장 기능의 CLSID를 추가한다.
    CreateRegKey('directory\shellex\CopyHookHandlers\' + ClassName, '',
      ClsID)
  else
    DeleteRegKey('directory\shellex\CopyHookHandlers\' + ClassName);
end;

procedure TCopyHookFactory.ApproveShellExtension(Register: Boolean;
  const ClsID: string);
// 윈도우즈 NT에서도 쉘 확장 기능이 제대로 작동하려면 다음 레지스트리 항목을
// 추가해야 한다.
const
  SApproveKey = 'SOFTWARE\Microsoft\Windows\CurrentVersion\Shell
     Extensions\Approved';
begin
  with TRegistry.Create do
    try
      RootKey := HKEY_LOCAL_MACHINE;
```

```
      if not OpenKey(SApproveKey, True) then Exit;
      if Register then WriteString(ClsID, Description)
      else DeleteValue(ClsID);
    finally
      Free;
    end;
end;

const
  CLSID_CopyHook: TGUID = '{66CD5F60-A044-11D0-A9BF-00A024E3867F}';

initialization
  TCopyHookFactory.Create(ComServer, TCopyHook, CLSID_CopyHook,
    'D4DG_CopyHook', 'D5DG Copy Hook 셸 확장 예제',
    ciMultiInstance, tmApartment);
end.
```

코드를 보면 알겠지만, TCopyHookFactory의 인스턴스는 보통 때와는 달리 유닛의 initialization부에서 생성된다. 그림 24.7은 이 복사 훅 핸들러 DLL을 등록한 후에 어떤 폴더의 이름을 변경하려고 하면 나타나는 메시지 상자이다.

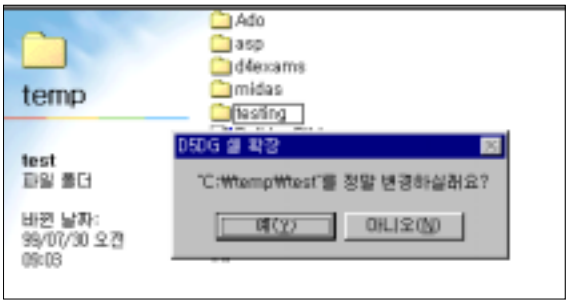



**그림 24.7**
복사 훅 처리자의 실행 모습

## 문맥 메뉴 핸들러

문맥 메뉴 핸들러(Context Menu Handler)란, 쉘에서 어떤 파일을 오른쪽 클릭했을 때 나타나는 팝업 메뉴에 어떤 항목을 추가하거나 선택된 항목을 처리하기 위한 쉘 확장 기능이다. 그림 24.8이 쉘 문맥 메뉴의 예이다.

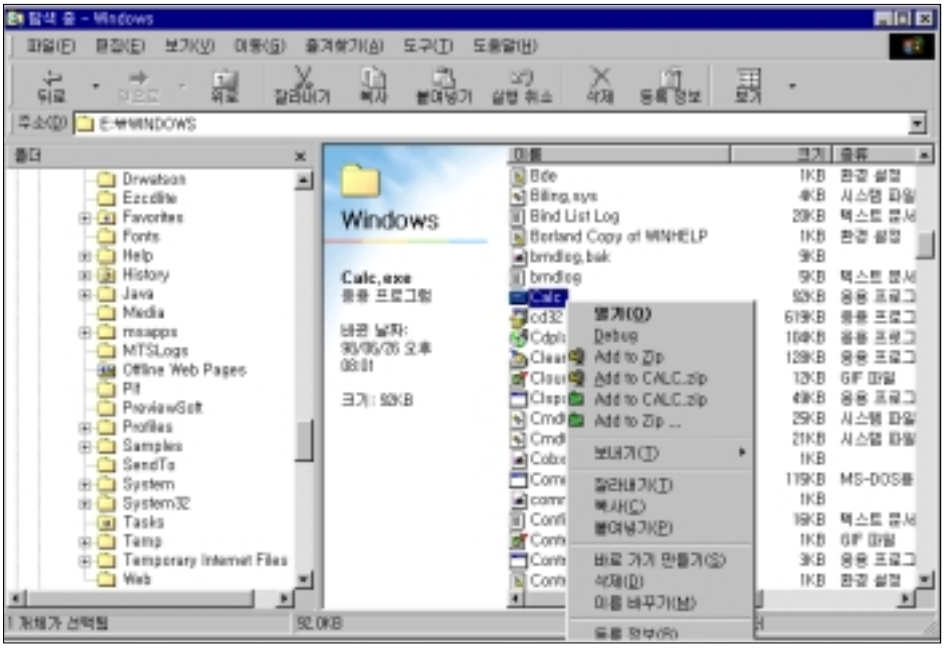



**그림 24.8**
Cale 파일을 오른쪽 클릭했을 때 나타나는 문맥 메뉴

문맥 메뉴 핸들러를 만들려면 IShellExtInit과 IContextMenu 인터페이스를 구현해야 한다. 이번 예제에서는 볼랜드 패키지 라이브러리(Borland Package Library, BPL) 파일에 대한 문맥 메뉴를 만들어 보겠다. 쉘에서 .bpl 파일을 오른쪽 클릭하면 팝업 메뉴에 그 bpl 파일에 대한 정보를 얻을 수 있는 항목이 나타나도록 하는 것이다. 이번에 만드는 문맥 메뉴 핸들러 객체의 이름은 TContextMenu이며, 복사 훅 핸들러와 마찬가지로 TComObject를 상속한다.

### IShellExtInit의 구현

IShellExtInit 인터페이스는 쉘 확장 기능을 초기화하기 위한 것으로 ShlObj 유닛에 다음과 같이 정의되어 있다.

```
type
  IShellExtInit = interface(IUnknown)
    [SID_IShellExtInit]
    function Initialize(pidlFolder: PItemIDList; lpdobj: IDataObject;
      hKeyProgID: HKEY): HResult; stdcall;
  end;
```

이 인터페이스의 유일한 메소드인 Initialize()는 쉘 확장 기능(이번 예제에서는 문맥 메뉴 핸들러)을 초기화하기 위한 것이다. 그럼 각각의 인자들에 대해서 알아보자.

pidFolder는 PItemIDList형의 포인터인데, 이 포인터 인자가 가리키는 것은 문맥 메뉴가 적용될 폴더(즉, 사용자가 오른쪽 클릭을 한 객체가 있는 폴더)에 대한 정보가 담겨 있는 레코드이다. lpdobj는 IDataObject 인터페이스 인스턴스인데, 이 인스턴스를 이용해서 작업을 수행할 객체를 얻는다. hkeyProgID에는 파일 객체나 폴더의 레지스트리키가 담겨 있다.

아래의 코드는 이 메소드를 구현한 것이다. 좀 복잡해 보이는 코드인데, 크게 나누면 세 가지 작업을 하는 코드이다. 우선, lpobj.GetData()를 호출해서 IDataObject 인스턴스의 데이터를 얻는다. 그런 다음, DragQueryFile()을 두 번 호출해서 데이터로부터 파일 개수와 파일 이름을 얻는다. 마지막으로, 파일 이름을 객체의 FFileName 필드에 넣는다.

```
function TContextMenu.Initialize(pidlFolder: PItemIDList; lpdobj:
  IDataObject; hKeyProgID: HKEY): HResult;
var
  Medium: TStgMedium;
  FE: TFormatEtc;
begin
  try
    // lpdobj가 nil이면 호출 결과를 실패로 만들고 종료한다.
    if lpdobj = nil then
    begin
      Result := E_FAIL;
      Exit;
    end;
```

```
    with FE do
    begin
      cfFormat := CF_HDROP;
      ptd := nil;
      dwAspect := DVASPECT_CONTENT;
      lindex := -1;
      tymed := TYMED_HGLOBAL;
    end;
       // IDataObject 인스턴스가 가리키고 있는 데이터를 CF_HDROP 형식의
       // HGLOBAL 저장 매체로 변환한다.
    Result := lpdobj.GetData(FE, Medium);
    if Failed(Result) then Exit;
    try
           // 하나의 파일만 선택된 경우, 파일 이름을 뽑아서 szFile에 넣는다.
           // 아니면 호출 결과를 실패로 만들고 종료한다.
      if DragQueryFile(Medium.hGlobal, $FFFFFFFF, nil, 0) = 1 then
      begin
        DragQueryFile(Medium.hGlobal, 0, FFileName, SizeOf(FFileName));
        Result := NOERROR;
      end
      else
        Result := E_FAIL;
    finally
      ReleaseStgMedium(medium);

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

## IContextMenu의 구현

IContextMenu 인터페이스는 쉘의 파일에 관련된 문맥 메뉴, 즉 팝업 메뉴를 조작하기 위한 것이다. 이 인터페이스는 ShlObj 유닛에 다음과 같이 정의되어 있다.

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

문맥 메뉴 핸들러가 IShellExtInit 인터페이스를 통해서 초기화된 후에는 IContextMenu.QueryContextMenu()가 호출된다. 이 메소드의 첫 번째 인자는 메뉴 핸들이며, 두 번째 인자는 팝업 메뉴의 몇 번째 항목에 첫 번째 인자로 지정할 메뉴를 삽입할 것인지를 뜻한다. 세 번째와 네 번째 인자는 메뉴 항목 ID의 최소값과 최대값이며, 마지막 인자는 메뉴의 특성을 뜻하는 플래그이다. 다음은 이번 예제인 TContextMenu가 구현한 QueryContextMenu() 메소드인데, Menu 인자로 넘겨 받은 메뉴에 "Package Info..."라는 항목을 추가한다(이 메소드는 삽입된 마지막 메뉴 항목의 색인 번호에 1을 더한 값을 돌려줘야 한다는 점을 기억하기 바란다).

```
function TContextMenu.QueryContextMenu(Menu: HMENU; indexMenu, idCmdFirst,
  idCmdLast, uFlags: UINT): HResult;
begin
  FMenuIdx := indexMenu;
  // 문맥 메뉴에 메뉴 항목을 하나 추가한다.
  InsertMenu (Menu, FMenuIdx, MF_STRING or MF_BYPOSITION, idCmdFirst,
    '패키지 정보...');
  // 삽입된 마지막 메뉴 항목의 색인 번호에 1을 더한 값을 돌려줘야 한다.
  Result := FMenuIdx + 1;
end;
```

그 다음으로 쉘은 GetCommandString() 메소드를 호출한다. 쉘은 이 메소드를 통해서 메뉴 항목에 대한 국가별 명령 문자열(열기, 편집 등)이나 도움말 문자열을 얻는다. 이 메소드의 각 인자들의 의미는 왼쪽부터 메뉴 항목의 오프셋, 얻고자 하는 정보의 종류를 뜻하는 플래그, 아직 사용되지 않는 인자, 그리고 문자열 버퍼와 버퍼의 크기이다.

다음은 이번 예제인 TContextMenu에서 GetCommandString() 메소드를 구현한 코드인데, 메뉴 항목에 대한 도움말 문자열을 돌려준다(도움말 문자열은 탐색기나 폴더 창의 상태 표시줄에 나타난다).

```
function TContextMenu.GetCommandString(idCmd, uType: UINT; pwReserved:
  PUINT; pszName: LPSTR; cchMax: UINT): HRESULT;
begin
  Result := S_OK;
  try
    // 이 문맥 메뉴 핸들러가 처리해야 할 메뉴 항목인지, 그리고 쉘이 도움말 문자열을
   // 요구하는 것인지 확인한다.
    if (idCmd = FMenuIdx) and ((uType and GCS_HELPTEXT) <> 0) then
      // 메뉴 항목에 대한 도움말 문자열을 돌려준다.
      StrLCopy(pszName, '선택된 패키지에 대한 정보 얻음.', cchMax)
    else
      Result := E_INVALIDARG;
  except
    Result := E_UNEXPECTED;
  end;
end;
```

여기까지는 아직 사용자가 메뉴 항목을 클릭하지 않은 상태이다. QueryContextMenu()에서 추가한 메뉴 항목을 사용자가 클릭하면 InvokeCommand() 메소드가 호출된다. 이 메소드의 인자는 TCMInvokeCommandInfo 레코드형인데, 이 레코드는 ShlObj 유닛에 다음과 같이 정의되어 있다.

```
type
  PCMInvokeCommandInfo = ^TCMInvokeCommandInfo;
  TCMInvokeCommandInfo = packed record
    cbSize: DWORD;        { sizeof(CMINVOKECOMMANDINFO)이어야 함 }
    fMask: DWORD;         { CMIC_MASK_* 플래그들의 조합 }
    hwnd: HWND;           { NULL도 가능(소유자 윈도우가 없는 경우) }
    lpVerb: LPCSTR;       { MAKEINTRESOURCE(idOffset) 문자열 }
    lpParameters: LPCSTR; { NULL도 가능(인자가 없는 경우) }
    lpDirectory: LPCSTR;  { NULL도 가능(지정된 디렉토리가 없는 경우)}
    nShow: Integer;       { ShowWindow() API의 SW_ 값들 중 하나 }
    dwHotKey: DWORD;
    hIcon: THandle;
  end;
```

lpVerb 필드의 하위 워드에는 선택된 메뉴 항목의 색인이 들어 있다. 다음은 이번 예제 TContextMenu에서 구현한 InvokeCommand()이다.

```
function TContextMenu.InvokeCommand(var lpici: TCMInvokeCommandInfo):
  HResult;
begin
  Result := S_OK;
  try
    // 쉘이 아니라 애플리케이션이 이 메소드를 호출한 경우에는 호출을 실패로 만든다.
    if HiWord(Integer(lpici.lpVerb)) <> 0 then
    begin
      Result := E_FAIL;
      Exit;
    end;
    // lpici.lpVerb로 지정된 명령을 수행한다.

    // 잘못된 메뉴 항목이면 E_INVALIDARG를 돌려준다.
    if LoWord(lpici.lpVerb) = FMenuIdx then
      ExecutePackInfoApp(FFileName, lpici.hwnd)
    else
      Result := E_INVALIDARG;
  except
    MessageBox(lpici.hwnd, '패키지 정보 획득 실패.',
          'Error', MB_OK or MB_ICONERROR);
    Result := E_FAIL;
  end;
end;
```

이 메소드는 ExcutePackInfoApp() 함수를 호출해서 PackInfo.exe를 띄운다. PackInfo.exe는 BPL 파일에 대한 정보를 보여주는 애플리케이션인데, 13장, "고난도 기법들"의 예제 프로젝트로 나온 것이니 자세한 설명은 13장을 참고하기 바란다.

### 등록

문맥 메뉴 핸들러는 시스템 레지스트리의 HKEY_CLASSES_ROOT\<파일 종류>\shellex\ContextMenuHandler 아래에 등록된다. 이번 예제 문맥 메뉴 핸들러 역시 클래스 팩토리의 메소드들을 재정의해서 DLL이 자신을 스스로 등록하도록 했다. 목록

24.10은 TContextMenu와 등록 관련 메소드들이 포함된 ContMain.pas의 소스 코드이다. 그림 24.9는 BPL 파일을 오른쪽 클릭할 때 나오는 문맥 메뉴에 새로운 메뉴 항목이 추가된 모습이고, 그림 24.10은 패키지 정보...를 선택해서 실행된 PackInfo.exe의 윈도우에 패키지의 정보들이 나타난 모습이다.

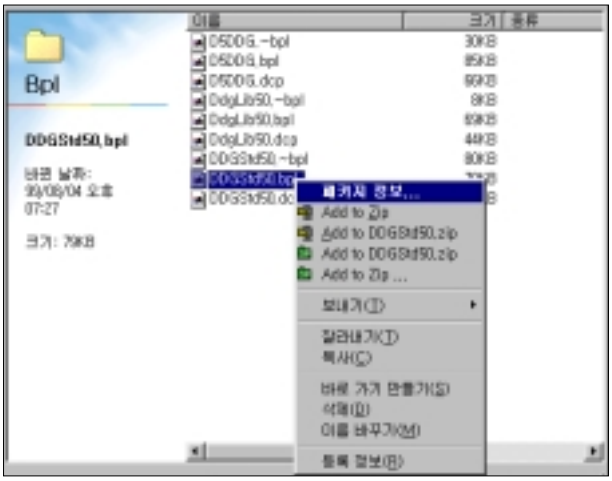



**그림 24.9**
문맥 메뉴 처리자의 실행 모습




**그림 24.10**
문맥 메뉴 처리자를 통해서 패키지 정보 프로그램을 띄운 모습

목록 24.10 Ch24\ContMenu\ContMain.pas

```
unit ContMain;

interface

uses Windows, ComObj, ShlObj, ActiveX;

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

  TContextMenuFactory = class(TComObjectFactory)
  protected
    function GetProgID: string; override;
    procedure ApproveShellExtension(Register: Boolean; const ClsID:
          string); virtual;
  public
    procedure UpdateRegistry(Register: Boolean); override;
  end;

implementation

uses ComServ, SysUtils, ShellAPI, Registry;
```

```
procedure ExecutePackInfoApp(const FileName: string; ParentWnd: HWND);
const
  SPackInfoApp = '%sPackInfo.exe';
  SCmdLine = '"%s" %s';
  SErrorStr = 'PackInfo 실행 실패:'#13#10#13#10;
var
  PI: TProcessInformation;
  SI: TStartupInfo;
  ExeName, ExeCmdLine: string;
  Buffer: array[0..MAX_PATH] of char;
begin
  // 이 DLL의 경로 이름을 얻는다. PackInfo.exe는 DLL과 같은 디렉토리에 있는
  // 것으로 가정한다.
  GetModuleFileName(HInstance, Buffer, SizeOf(Buffer));
  ExeName := Format(SPackInfoApp, [ExtractFilePath(Buffer)]);
  ExeCmdLine := Format(SCmdLine, [ExeName, FileName]);
  FillChar(SI, SizeOf(SI), 0);
  SI.cb := SizeOf(SI);
  if not CreateProcess(PChar(ExeName), PChar(ExeCmdLine), nil, nil, False,
    0, nil, nil, SI, PI) then
    MessageBox(ParentWnd, PChar(SErrorStr + SysErrorMessage
          (GetLastError)), '에러', MB_OK or MB_ICONERROR);
end;

{ TContextMenu }

{ TContextMenu.IContextMenu }

function TContextMenu.QueryContextMenu(Menu: HMENU; indexMenu, idCmdFirst,
  idCmdLast, uFlags: UINT): HResult;
begin
  FMenuIdx := indexMenu;
  // 문맥 메뉴에 메뉴 항목을 하나 추가한다.
  InsertMenu (Menu, FMenuIdx, MF_STRING or MF_BYPOSITION, idCmdFirst,
    '패키지 정보...');
  // 삽입된 마지막 메뉴 항목의 색인 번호에 1을 더한 값을 돌려줘야 한다.
  Result := FMenuIdx + 1;
end;
```

```
function TContextMenu.InvokeCommand(var lpici: TCMInvokeCommandInfo):
 HResult;
begin
  Result := S_OK;
  try
    // 애플리케이션이 이 메소드를 호출한 것이어서는 안된다.
    if HiWord(Integer(lpici.lpVerb)) <> 0 then
    begin
      Result := E_FAIL;
      Exit;
    end;
    // lpici.lpVerb로 지정된 명령을 수행한다.
    // 인자 개수가 유효하지 않으면 E_INVALIDARG를 돌려준다.
    if LoWord(lpici.lpVerb) = FMenuIdx then
      ExecutePackInfoApp(FFileName, lpici.hwnd)
    else
      Result := E_INVALIDARG;
  except
    MessageBox(lpici.hwnd, '패키지 정보 획득 에러.', '에러',
      MB_OK or MB_ICONERROR);
    Result := E_FAIL;
  end;
end;

function TContextMenu.GetCommandString(idCmd, uType: UINT; pwReserved: PUINT;
  pszName: LPSTR; cchMax: UINT): HRESULT;
begin
  Result := S_OK;
  try
  // 이 문맥 메뉴 핸들러가 처리해야 할 메뉴 항목인지 그리고 쉘이 도움말 문자열을
  // 요구하는 것인지 확인한다.
    if (idCmd = FMenuIdx) and ((uType and GCS_HELPTEXT) <> 0) then
      // 메뉴 항목에 대한 도움말 문자열을 돌려준다.
      StrLCopy(pszName, 'Get information for the selected package.',
               cchMax)
    else
      Result := E_INVALIDARG;
  except
```

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

{ TContextMenu.IShellExtInit }

function TContextMenu.Initialize(pidlFolder: PItemIDList; lpdobj:
 IDataObject; hKeyProgID: HKEY): HResult;
var
  Medium: TStgMedium;
  FE: TFormatEtc;
begin
  try
    // lpdobj가 nil이면 호출 결과를 실패로 만들고 종료한다.
    if lpdobj = nil then
    begin
      Result := E_FAIL;
      Exit;
    end;
    with FE do
    begin
      cfFormat := CF_HDROP;
      ptd := nil;
      dwAspect := DVASPECT_CONTENT;
      lindex := -1;
      tymed := TYMED_HGLOBAL;
    end;
      // IDataObject 인스턴스가 가리키고 있는 데이터를 CF_HDROP 형식의
      // HGLOBAL 저장 매체로 변환한다.
    Result := lpdobj.GetData(FE, Medium);
    if Failed(Result) then Exit;
    try
          // 하나의 파일만 선택된 경우, 파일 이름을 뽑아서 szFile에 넣는다.
          // 아니면 호출 결과를 실패로 만들고 종료한다.
      if DragQueryFile(Medium.hGlobal, $FFFFFFFF, nil, 0) = 1 then
      begin
        DragQueryFile(Medium.hGlobal, 0, FFileName, SizeOf(FFileName));
        Result := NOERROR;
      end
```

```
      else
        Result := E_FAIL;
    finally
      ReleaseStgMedium(medium);
    end;
  except
    Result := E_UNEXPECTED;
  end;
end;

{ TContextMenuFactory }

function TContextMenuFactory.GetProgID: string;
begin
  // 문맥 메뉴 핸들러 쉘 확장 기능에는 ProgID가 필요하지 않다.
  Result := '';
end;

procedure TContextMenuFactory.UpdateRegistry(Register: Boolean);
var
  ClsID: string;
begin
  ClsID := GUIDToString(ClassID);
  inherited UpdateRegistry(Register);
  ApproveShellExtension(Register, ClsID);
  if Register then
  begin
    // .bpl 파일을 등록한다.
    CreateRegKey('.bpl', '', 'BorlandPackageLibrary');
    // 이 DLL을 .bpl 파일의 문맥 메뉴 핸들러로 등록한다.
    CreateRegKey('BorlandPackageLibrary\shellex\ContextMenuHandlers\' +
      ClassName, '', ClsID);
  end
  else begin
    DeleteRegKey('.bpl');
    DeleteRegKey('BorlandPackageLibrary\shellex\ContextMenuHandlers\' +
      ClassName);
  end;
end;
```

```
procedure TContextMenuFactory.ApproveShellExtension(Register: Boolean;
  const ClsID: string);
// 이 문맥 메뉴 핸들러가 윈도우즈 NT에서도 제대로 작동하려면 아래의
// 레지스트리 항목을 등록해야 한다.
const
  SApproveKey = 'SOFTWARE\Microsoft\Windows\CurrentVersion\' +
     'Shell Extensions\Approved';
begin
  with TRegistry.Create do
    try
      RootKey := HKEY_LOCAL_MACHINE;
      if not OpenKey(SApproveKey, True) then Exit;
      if Register then WriteString(ClsID, Description)
      else DeleteValue(ClsID);
    finally
      Free;
    end;
end;

const
  CLSID_CopyHook: TGUID = '{7C5E74A0-D5E0-11D0-A9BF-E886A83B9BE5}';

initialization
  TContextMenuFactory.Create(ComServer, TContextMenu, CLSID_CopyHook,
    'D4DG_ContextMenu', 'D5DG 컨텍스트 메뉴 셸 확장 예제',
    ciMultiInstance, tmApartment);
end.
```

## 아이콘 핸들러

아이콘 핸들러는 쉘의 파일 객체에 표시되는 아이콘을 처리하는 쉘 확장 기능으로 이것을 이용하면 같은 종류의 파일들을 서로 다른 아이콘들로 표시할 수 있다. 이번 예제 아이콘 핸들러인 TIconHanlder는 볼랜드 패키지 라이브러리 파일(.bpl)을 내부적 종류에 따라 서로 다른 아이콘으로 표시하는 역할을 한다. 즉, 같은 .bpl 파일이라도 설계 패키지인지, 런타임 패키지인지, 아니면 둘 다인지에 따라 서로 다른 아이콘으로 표시하는 것이다.

### 패키지 종류 얻기

아이콘 핸들러의 인터페이스 구현에 앞서서 우선 패키지 라이브러리 파일의 종류를 돌려주는 메소드부터 살펴보자. 이 메소드는 다음과 같이 정의된 TPackType 열거형의 값을 돌려준다.

```
TPackType = (ptDesign, ptDesignRun, ptNone, ptRun);
```

다음은 메소드의 코드이다.

```
function TIconHandler.GetPackageType: TPackType;
var
  PackMod: HMODULE;
  PackFlags: Integer;
begin
  // 패키지를 실제로 로드할 필요는 없으므로, LoadLibrayrEx 함수에
  // LOAD_LIBRARY_AS_DATAFILE 플래그를 지정해서 필요한 데이터만 뽑아낸다.
  PackMod := LoadLibraryEx(PChar(FFileName), 0, LOAD_LIBRARY_AS_DATAFILE);
  if PackMod = 0 then
  begin
    Result := ptNone;
    Exit;
  end;
  try
    GetPackageInfo(PackMod, nil, PackFlags, PackInfoProc);
  finally
    FreeLibrary(PackMod);
  end;
  // 설계 시점과 런타임 플래그만 남긴 후 플래그에 따라 적절한
  // 리턴값을 돌려준다.
  case PackFlags and (pfDesignOnly or pfRunOnly) of
    pfDesignOnly: Result := ptDesign;
    pfRunOnly: Result := ptRun;
    pfDesignOnly or pfRunOnly: Result := ptDesignRun;
  else
    Result := ptNone;
  end;
end;
```

이 메소드는 SysUtils 유닛의 GetPackageInfo()를 호출해서 패키지 종류를 뜻하는 플래그를 얻는다. 이 메소드는 성능을 최적화하기 위해서 델파이의 LoadPackage() 프로시저 대신 API 함수 LoadLibraryEx()를 이용해서 패키지 라이브러리를 로드한다. LoadPackage() 프로시저는 LoadLibrary() API 함수를 호출해서 BPL을 로드한 후 InitializePackage()를 호출해서 패키지의 각 유닛들의 초기화 코드를 수행한다. 이 메소드에 필요한 것은 패키지가 실제로 초기화되는 것이 아니라 BPL 파일에 링크된 리소스에 들어 있는 패키지 종류 플래그들이므로 LoadPackage()를 사용할 필요가 없다. 그냥 LoadLibraryEx() 함수에 LOAD_LIBRARY_AS_DATAFILE 플래그를 넣어서 호출하면 된다.

### 아이콘 핸들러 인터페이스

앞에서 말했듯이 아이콘 핸들러는 IExtractInoc(ShlObj 유닛에 정의되어 있음) 인터페이스와 IPersistFile(ActiveX 유닛에 정의되어 있음) 인터베이스를 모두 지원해야 한다. 두 인터페이스의 정의는 다음과 같다.

```
type
  IExtractIconA = interface(IUnknown)
    [SID_IExtractIconA]
    function GetIconLocation(uFlags: UINT; szIconFile: PAnsiChar; cchMax:
    UINT;  out piIndex: Integer; out pwFlags: UINT): HResult; stdcall;
    function Extract(pszFile: PAnsiChar; nIconIndex: UINT;
     out phiconLarge, phiconSmall: HICON; nIconSize: UINT): HResult; stdcall;
  end;

type
  IPersistFile = interface(IPersist)
      ['{0000010B-0000-0000-C000-000000000046}']
      function IsDirty: HResult; stdcall;
      function Load(pszFileName: POleStr; dwMode: Longint): HResult;
          stdcall;
      function Save(pszFileName: POleStr; fRemember: BOOL): HResult;
          stdcall;
      function SaveCompleted(pszFileName: POleStr): HResult; stdcall;
      function GetCurFile(out pszFileName: POleStr): HResult; stdcall;
  end;
```

이전에 나왔던 인터페이스들보다는 훨씬 더 복잡한 것 같지만, 실제로 구현해야 할 메소드는 단 두 개뿐이다. 아이콘 핸들러가 구현해야 할 첫 번째 메소드는 IPersistFile.Load()이다. 이 메소드는 쉘이 쉘 확장 기능을 초기화하기 위해서 호출되는데, pszFileName 인자로 넘겨 받은 파일 이름은 다른 메소드에서 사용해야 하므로 보존해 두어야 한다. 다음은 이번 예제 아이콘 핸들러인 TIconHandler에서 구현한 Load() 메소드이다.

```
function TIconHandler.Load(pszFileName: POleStr; dwMode: Longint): HResult;
begin
  // 쉘이 아이콘 핸들러를 초기화하기 위해 호출하는 메소드이다.
  // pszFileName로 넘겨 받은 파일 이름을 보존해 두어야 한다.
  FFileName := pszFileName;
  Result := S_OK;
end;
```

구현해야 할 두 번째 메소드는 IExtractIcon.GetIconLocation()이다. 그럼 이 메소드의 인자들을 알아보자.

uFlags는 화면에 표시할 아이콘의 종류를 뜻하는데, 값은 0이나 GIL_FORSHELL 아니면 GIL_OPENICON이다. GIL_FORSHELL은 아이콘을 쉘 폴더 창에 표시한다는 뜻이고, GIL_OPENICON은 "열린" 상태의 아이콘을 표시한다는 뜻이다. 윈도우즈 탐색기의 트리뷰에 나타나는 폴더 아이콘들을 연상하면 된다. "열린" 상태와 "닫힌" 상태의 아이콘을 모두 가지고 있는 경우에만 이 값을 처리하면 된다.

szIconFile은 아이콘의 위치를 넣을 버퍼이다. 메소드에서 이 인자에 적절한 아이콘 경로를 넣어야 한다. cchMax는 버퍼의 크기이다. piIndex에는 아이콘의 색인을 넣어야 하는데, 아이콘 위치에 대해 이야기할 때 자세히 설명하겠다. pwFlags는 아이콘의 처리 방식을 지정하기 위한 플래그인데, 여기에 넣을 수 있는 값은 표 24.8에 나와 있다.

표 24.8 GetIconLocation()의 pwFlags 인자에 넣을 수 있는 값들

| 플래그 | 의미 |
|---|---|
| GIL_DONTCACHE | 이 아이콘에 사용할 비트맵은 캐시에 저장하지 않도록 한다. 이후 운영체제에서 GIL_DONTCACHELOCATION 플래그가 도입될 수도 있으므로 이 플래그를 지정해 주는 것이 중요하다. |
| GIL_NOTFILENAME | 아이콘 위치가 일반적인 파일 이름/색인 쌍을 의미하지 않는다. 호출자(쉘)는 이 핸들러 객체의 IExtractIcon.Extract() 메소드를 이용해서 아이콘을 뽑아내야 한다. |
| GIL_PERCLASS | 이 클래스의 모든 객체들은 동일한 아이콘을 사용한다. 이 값 역시 별 의미가 없다. 하나의 파일 종류에 대해서 한 종류의 아이콘만 사용한다면 굳이 아이콘 핸들러를 구현할 필요가 없기 때문이다. |
| GIL_PERINSTANCE | 이 클래스의 각 객체들이 서로 다른 아이콘을 사용한다. Setup.exe처럼 이름도 같고 확장자도 같지만, 서로 다른 아이콘을 사용하는 경우를 처리하기 위해서 쉘이 내부적으로 사용하는 플래그이다. 개발자가 아이콘 핸들러를 만들 때에는 이 플래그를 사용하지 않는 것이 일반적이다. |
| GIL_SIMULATEDOC | 호출자는 지정된 아이콘을 사용해서 문서 아이콘을 만들어야 한다. |

이번 예제 TIconHandler가 구현한 GetIconLocation()은 다음과 같다.

```
function TIconHandler.GetIconLocation(uFlags: UINT; szIconFile: PAnsiChar;
  cchMax: UINT; out piIndex: Integer; out pwFlags: UINT): HResult;
begin
  Result := S_OK;

  try
    // 쉘에게 이 DLL의 아이콘을 사용하라고 알려준다.
    GetModuleFileName(HInstance, szIconFile, cchMax);
    // BPL 파일의 종류에 따라 다른 아이콘을 사용할 것이므로
    // 쉘이 아이콘 비트맵을 캐시에 저장하지 않도록 한다.
    pwFlags := GIL_DONTCACHE or GIL_PERINSTANCE;
    // 아이콘 색인과 TPackType값을 일치시킨다.
    piIndex := Ord(GetPackageType);
  except
    // 예외가 발생하면 기본 패키지 아이콘을 사용한다.
    piIndex := Ord(ptNone);
  end;
end;
```

아이콘들은 이 아이콘 핸들러 DLL의 리소스에 들어 있으므로 szIconFile 버퍼에 이 DLL의 경로 이름을 넣어서 쉘에 돌려준다. 그리고 아이콘들이 리소스에 들어 있는 순서는 패키지 종류를 뜻하는 TPackType 열거값들의 순서와 일치하므로 GetPackageType()로 얻은 TPackType값의 순서를 그냥 piIndex에 넣으면 된다.

### 등록

아이콘 핸들러는 레지스트리의 HKEY_CLASSES_ROOT\<파일 종류>\shellex\IconHandler 키 아래에 등록한다. 앞의 예제들과 마찬가지로 이번 아이콘 핸들러 역시 TComObjectFactory의 여러 메소드들을 재정의해서 레지스트리에 등록한다. 목록 24.11에 이번 예제 아이콘 핸들러의 전체 소스 코드가 나와 있다.

그림 24.11은 서로 다른 종류의 BPL 패키지가 담겨 있는 폴더의 모습이다. 같은 bpl 파일이라도 서로 다른 아이콘으로 나타나 있음을 주목하기 바란다.

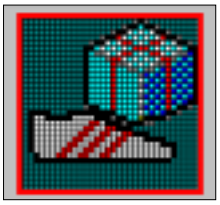


**그림 24.11**
아이콘 핸들러를
사용한 결과 나타나는
아이콘

**목록 24.11** Ch24\IconHand\IconMain.pas

```
unit IconMain;

interface

uses Windows, ActiveX, ComObj, ShlObj;

type
  TPackType = (ptDesign, ptDesignRun, ptNone, ptRun);
```

```
  TIconHandler = class(TComObject, IExtractIcon, IPersistFile)
  private
    FFileName: string;
    function GetPackageType: TPackType;
  protected
    // IExtractIcon의 메소드들
    function GetIconLocation(uFlags: UINT; szIconFile: PAnsiChar; cchMax:
          UINT; out piIndex: Integer; out pwFlags: UINT): HResult; stdcall;
    function Extract(pszFile: PAnsiChar; nIconIndex: UINT;
      out phiconLarge, phiconSmall: HICON; nIconSize: UINT): HResult;
              stdcall;
    // IPersist의 메소드들
    function GetClassID(out classID: TCLSID): HResult; stdcall;
    // IPersistFile의 메소드들
    function IsDirty: HResult; stdcall;
    function Load(pszFileName: POleStr; dwMode: Longint): HResult;
          stdcall;
    function Save(pszFileName: POleStr; fRemember: BOOL): HResult;
          stdcall;
    function SaveCompleted(pszFileName: POleStr): HResult; stdcall;
    function GetCurFile(out pszFileName: POleStr): HResult; stdcall;
  end;

  TIconHandlerFactory = class(TComObjectFactory)
  protected
    function GetProgID: string; override;
    procedure ApproveShellExtension(Register: Boolean; const ClsID:
          string); virtual;
  public
    procedure UpdateRegistry(Register: Boolean); override;
  end;

implementation

uses SysUtils, ComServ, Registry;

{ TIconHandler }

procedure PackInfoProc(const Name: string; NameType: TNameType; Flags:
```

```
Byte;
  Param: Pointer);
begin
  // 패키지 종류를 뜻하는 플래그만 얻으면 되므로 이 메소드는 구현할 필요가 없다.
end;

function TIconHandler.GetPackageType: TPackType;
var
  PackMod: HMODULE;
  PackFlags: Integer;
begin
  // 패키지를 실제로 로드할 필요는 없으므로 LoadLibrayrEx 함수에
  // LOAD_LIBRARY_AS_DATAFILE 플래그를 지정해서 필요한 데이터만
  // 뽑아낸다.
  PackMod := LoadLibraryEx(PChar(FFileName), 0, LOAD_LIBRARY_AS_DATAFILE);
  if PackMod = 0 then
  begin
    Result := ptNone;
    Exit;
  end;
  try
    GetPackageInfo(PackMod, nil, PackFlags, PackInfoProc);
  finally
    FreeLibrary(PackMod);
  end;
  // 설계시점과 런타임 플래그만 남긴 후 플래그에 따라 적절한
  // 리턴값을 돌려준다.
  case PackFlags and (pfDesignOnly or pfRunOnly) of
    pfDesignOnly: Result := ptDesign;
    pfRunOnly: Result := ptRun;
    pfDesignOnly or pfRunOnly: Result := ptDesignRun;
  else
    Result := ptNone;
  end;
end;

{ TIconHandler.IExtractIcon }

function TIconHandler.GetIconLocation(uFlags: UINT; szIconFile: PAnsiChar;
```

```
  cchMax: UINT; out piIndex: Integer; out pwFlags: UINT): HResult;
begin
  Result := S_OK;
  try
    // 쉘에게 이 DLL의 아이콘을 사용하라고 알려준다.
    GetModuleFileName(HInstance, szIconFile, cchMax);
    // BPL 파일의 종류에 따라 다른 아이콘을 사용할 것이므로
    // 쉘이 아이콘 비트맵을 캐시에 저장하지 않도록 한다.
    pwFlags := GIL_DONTCACHE or GIL_PERINSTANCE;
    // 아이콘 색인과 TPackType값을 일치시킨다.
    piIndex := Ord(GetPackageType);
  except
    // 예외가 발생하면 기본 패키지 아이콘을 사용한다.
    piIndex := Ord(ptNone);
  end;
end;

function TIconHandler.Extract(pszFile: PAnsiChar; nIconIndex: UINT;
  out phiconLarge, phiconSmall: HICON; nIconSize: UINT): HResult;
begin
 // 이 메소드는 아이콘 위치가 파일 이름/색인 쌍을 의미하지 않는 경우에
 // 아이콘을 쉘에 직접 제공하기 위한 것이다. 이번 예제에서는 DLL에
 // 아이콘들이 들어 있으므로 그냥 S_FALSE를 돌려주면 된다.
  Result := S_FALSE;
end;

{ TIconHandler.IPersist }

function TIconHandler.GetClassID(out classID: TCLSID): HResult;
begin
  // 아이콘 핸들러에서는 이 메소드가 호출되지 않는다.
  Result := E_NOTIMPL;
end;

{ TIconHandler.IPersistFile }

function TIconHandler.IsDirty: HResult;
begin
  // 아이콘 핸들러에서는 이 메소드가 호출되지 않는다.
```

```
  Result := S_FALSE;
end;

function TIconHandler.Load(pszFileName: POleStr; dwMode: Longint): HResult;
begin
  // 쉘이 아이콘 핸들러를 초기화하기 위해 호출하는 메소드이다.
  // pszFileName으로 넘겨 받은 파일 이름을 보존해 두어야 한다.
  FFileName := pszFileName;
  Result := S_OK;
end;

function TIconHandler.Save(pszFileName: POleStr; fRemember: BOOL): HResult;
begin
  // 아이콘 핸들러에서는 이 메소드가 호출되지 않는다.
  Result := E_NOTIMPL;
end;
function TIconHandler.SaveCompleted(pszFileName: POleStr): HResult;
begin
  // 아이콘 핸들러에서는 이 메소드가 호출되지 않는다.
  Result := E_NOTIMPL;
end;

function TIconHandler.GetCurFile(out pszFileName: POleStr): HResult;
begin
  // 아이콘 핸들러에서는 이 메소드가 호출되지 않는다.
  Result := E_NOTIMPL;
end;

{ TIconHandlerFactory }

function TIconHandlerFactory.GetProgID: string;
begin
  // 아이콘 핸들러에서는 ProgID가 필요 없다.
  Result := '';
end;

procedure TIconHandlerFactory.UpdateRegistry(Register: Boolean);
var
  ClsID: string;
```

```
begin
  ClsID := GUIDToString(ClassID);
  inherited UpdateRegistry(Register);
  ApproveShellExtension(Register, ClsID);
  if Register then
  begin
    // .bpl 파일을 등록한다.
    CreateRegKey('.bpl', '', 'BorlandPackageLibrary');
    // 이 DLL을 .bpl 파일의 아이콘 핸들러로 등록한다.
    CreateRegKey('BorlandPackageLibrary\shellex\IconHandler', '', ClsID);
  end
  else begin
    DeleteRegKey('.bpl');
    DeleteRegKey('BorlandPackageLibrary\shellex\IconHandler');
  end;
end;

procedure TIconHandlerFactory.ApproveShellExtension(Register: Boolean;
  const ClsID: string);
// 이 아이콘 핸들러가 윈도우즈 NT에서도 제대로 작동하려면 아래의
// 레지스트리 항목을 등록해야 한다.
const
  SApproveKey = 'SOFTWARE\Microsoft\Windows\CurrentVersion\' +
     'Shell Extensions\Approved';
begin
  with TRegistry.Create do
    try
      RootKey := HKEY_LOCAL_MACHINE;
      if not OpenKey(SApproveKey, True) then Exit;
      if Register then WriteString(ClsID, Description)
      else DeleteValue(ClsID);
    finally
      Free;
    end;
end;

const
  CLSID_IconHandler: TGUID = '{ED6D2F60-DA7C-11D0-A9BF-90D146FC32B3}';
```

```
initialization
  TIconHandlerFactory.Create(ComServer, TIconHandler, CLSID_IconHandler,
    'D5DG_IconHandler', 'D5DG 아이콘 핸들러 쉘 확장 예제',
    ciMultiInstance, tmApartment);
end.
```

# 요약

이번 장에서는 윈도우즈 쉘의 기능을 확장하는 다양한 기법들을 소개했다. 트레이 통지 아이콘과 앱바, 바로 가기에 대해 이야기했으며, 여러 가지 쉘 확장 기능들에 대해서도 이야기했다. 23장의 COM과 ActiveX에 대한 내용들이 이번 장을 통해서 좀 더 구체화되었을 것이다. 웹 사이트 중 http://members.tripod.com/~delphipower/에 보면 여러 가지 쉘 확장 예제들을 살펴볼 수 있다. 다음 장에서는 COM 기술의 결정판이라고 할 수 있는 ActiveX 컨트롤의 작성에 대해서 이야기하겠다.