트레이 아이콘 어플리케이션의 제작

(Creating Tray Icon Application)

윈도우 95 와 NT 4.0 에는 작업바의 우측에 작업 트레이라는 부분이 있다. 여기에 있는 아이콘은 현재 데스크 탑에서 실행되고 있는 윈도우를 가지고 있는 어플리케이션이라기 보다는 도스 시절에 있었던 램상주 프로그램과 비슷한 역할을 하는 것이 많다.
이번 장에서는 트레이에 상주하는 어플리케이션을 제작하는 방법에 대해서 알아보자.

Shell_NotifyIcon

트레이 아이콘을 사용하는 어플리케이션에서는 Shell_NotifyIcon 이라는 API 함수를 사용하게 된다. 이 함수에는 파라미터가 둘 있는데, TNotifyIconData 구조체에 대한 포인터와 트레이에 작업을 원하는 플래그를 설정한다. 플래그에는 아이콘을 추가, 삭제, 수정 등의 기능을 설정할 수 있다. TNotifyIconData 구조체는 다음과 같이 정의되어 있다.

```
type
  TNotifyIconData = record
    cbSize: DWORD;
    Wnd: HWND;
    uID: UINT;
    uFlags: UINT;
    uCallbackMessage: UINT;
    hIcon: HICON;
    szTip: array[0..63] of AnsiChar;
  end;
```

여기서 cbSize 는 구조체의 크기이며, Wnd 는 트레이 아이콘이 메시지를 보낼 윈도우에 대한 핸들이다. uID 는 아이콘을 확인하기 위한 것으로 어플리케이션에 트레이 아이콘이 여러 개 있을 때 사용된다. uFlag 필드에는 세 가지를 사용할 수 있다. NIF_MESSAGE, NIF_ICON, NIF_TIP 이 그것으로 어느 것이 유효한 것인지 결정하는 플래그이다.
uCallbackMessage 필드는 사용자의 아이콘에 대한 동작에 대해 윈도우(hWnd)로 보내어지는 사용자 정의 메시지의 갯수를 나타내며, hIcon 은 트레이 영역 안에 표시할 아이콘의 핸들이다. 마지막으로 szTip 필드에는 작업 표시줄의 아이콘 위로 마우스 커서가 놓일 때 보여주는 툴팁 문자열에 대한 텍스트이다.

이 구조체는 ShellAPI.pas 유닛에 선언되어 있으므로, 실제 사용을 하기 위해서 interface 섹션의 uses 문에 ShellAPI 를 추가해서 사용한다.

기본 원칙

트레이 아이콘 어플리케이션이 동작하는 방식에 대해 생각해보자.
제일 먼저 폼이 생성될 때 작업 트레이 부분에 TNotifyIconData 구조체의 내용을 적절히 채워서 아이콘을 등록해야 할 것이다. 그리고, 아이콘이나 툴팁 등의 변화가 있을 경우 이를 적절하게 반영해 주어야 한다.
그리고, 콜백 메시지를 등록해서 일단 트레이 아이콘 어플리케이션에 해당 되는 메시지일 경우에는 이 메시지를 처리해 주는 루틴을 만들어 주어야 한다. 이때 트레이 아이콘을 더블 클릭할 경우에는 디자인한 폼을 보여주도록 해야 할 것이며, 폼이 보여진 후 최소화 버튼을 눌렀을 때에는 다시 트레이 아이콘으로 복귀하도록 해야 할 것이다. 그리고, 트레이 아이콘에서 오른쪽 버튼을 클릭했을 때에는 디자인한 팝업 메뉴를 보여주고 메뉴를 실행할 수 있어야 한다. 또한, 작업이 끝나면 트레이 아이콘을 제거해야 한다.
마지막으로 염두에 두어야 할 것은 어플리케이션이 처음 시작할 때 어플리케이션 폼이 바로 트레이 아이콘으로 최소화해서 위치하게 해야 한다.
이를 간략하게 정리하면 다음과 같다.

1. 트레이 아이콘 등록
2. 트레이 아이콘 변화를 반영
3. 메시지 처리 루틴 제작
4. 팝업 메뉴와 폼의 최소화(minimize), 복귀(restore) 메시지 처리
5. 트레이 아이콘 제거
6. 어플리케이션 시작시 트레이 아이콘으로 시작할 것

먼저 트레이 아이콘을 등록하는 부분에 대해서 알아보자.
다음의 프로시저는 전형적인 트레이 아이콘을 등록하는 코드를 담고 있다.

```
procedure AddTrayIcon;
var
  IconData: TNotifyIconData;
begin
  with IconData do
  begin
    cbSize := SizeOf(IconData);
```

```
    Wnd := Handle;
    uID := 0;
    uFlags := NIF_ICON or NIF_MESSAGE or NIF_TIP;
    uCallBackMessage := WM_MyCallBack;
    hIcon := LoadIcon(hInstance, 'MyIcon');
    szTip := '시계';
  end;
  Shell_NotifyIcon(NIM_ADD, @IconData);
end;
```

내용은 앞에서 설명한 TNotifyIcon 형의 구조체 변수에 적절한 데이터 필드를 입력하고, 마지막에 Shell_NotifyIcon API 함수를 호출하는 것이다.
여기서 주의 깊게 보아야 할 부분이 세군데 있다.
첫째는 uFlags 필드로 여기에서 추가 또는 수정을 할 때 이용할 플래그를 결정한다. NIF_ICON 은 트레이 아이콘에 대한 정보를, NIF_TIP 은 트레이 아이콘에 대한 툴팁 힌트에 대한 정보를 나타낸다. 마지막으로 NIF_MESSAGE 는 사용되는 사용자 정의 메시지에 대한 것이다. 보통 처음 트레이 아이콘을 추가할 때에는 이 세가지 정보를 모두 등록해야 하므로 NIF_ICON, NIF_TIP, NIF_MESSAGE 를 사용하게 된다.
둘째는 uCallBackMessage 필드로 여기에는 메시지 처리를 하게될 사용자 정의 메시지를 대입한다. 보통 처음에 상수 선언을 해서 대입을 하게 된다.
세째로는 Shell_NotifyIcon API 함수를 호출할 때 사용하는 파라미터로, 지금과 같이 아이콘을 추가할 때에는 NIM_ADD, 아이콘에 대한 정보를 수정할 때에는 NIM_MODIFY, 아이콘을 제거할 때에는 NIM_DELETE 를 각각 호출한다.
이렇게 등록된 트레이 아이콘에 대한 정보를 수정할 때에는 다음과 같이 하면 된다.

```
Shell_NotifyIcon(NIM_MODIFY, @IconData);
```

즉, 바뀌게 되는 내용을 TNotifyIconData 형 변수의 필드에 대입하고 NIM_MODIFY 로 Shell_NotifyIcon API 함수를 호출한다.
트레이 아이콘을 제거할 때에도 다음과 같이 해주면 해결 된다.

```
Shell_NotifyIcon(NIM_DELETE, @IconData);
```

메시지 처리

메시지를 처리하기 위해서 먼저 사용할 사용자 정의 메시지와 이를 처리할 콜백 함수를 선

언해야 한다.

```
const
  WM_CallbackMessage = WM_User + 100;

procedure WndProc(var Message: TMessage);
```

이 함수는 어플리케이션에 대한 메시지를 전담해서 처리하게 된다. 이 함수를 다음과 같이 사용해서 메시지를 처리한다. 다음의 코드는 필자가 제작한 트레이 아이콘 제작 컴포넌트(TTrayIcon) 소스의 일부이다.

```
procedure TTrayIcon.WndProc(var Message: TMessage);
begin
  try
    with Message do
      case Msg of
        WM_QueryEndSession: Message.Result := 1;
        WM_EndSession:
          if TWmEndSession(Message).EndSession then EndSession;
        WM_CallbackMessage:
          case Message.lParam of
            WM_LBUTTONDBLCLK: DblClick;
            WM_LBUTTONUP: Click(mbLeft);
            WM_RBUTTONUP: Click(mbRight);
          end;
      else Result := DefWindowProc(FHandle, Msg, wParam, lParam);
    end;
  except
    Application.HandleException(Self);
  end;
end;
```

즉, 반드시 처리해 주어야 하는 메시지는 자신이 TNotifyIconData 형의 변수 필드에 등록한 메시지로 여기서는 WM_CallbackMessage 가 된다. 보통 그 중에서도 마우스 더블 클릭과 좌우 버튼 클릭에 대한 메시지만 처리하면 된다.
그 밖의 메시지 중 WM_EndSession 은 트레이 아이콘 어플리케이션을 종료하고자 할 때

발생하는 메시지 이다. 여기서는 필자가 따로 제작한 EndSession 이라는 프로시저를 호출한다. EndSession 프로시저에서는 다음과 같이 등록된 트레이 아이콘을 제거하는 코드를 사용한다.

```
procedure TTrayIcon.EndSession;
begin
  Shell_NotifyIcon(NIM_DELETE, @FIconData);
end;
```

트레이 아이콘 컴포넌트(TTrayIcon)의 제작

트레이 아이콘 어플리케이션에 대한 확실한 이해를 돕기 위해 델파이 폼에 올려 놓고 모든 어플리케이션을 바로 트레이 아이콘 어플리케이션으로 전환시킬 수 있는 컴포넌트를 하나 제작해 보았다.
이 컴포넌트를 사용하면 쉽게 트레이 아이콘 어플리케이션을 만들 수 있다. 지면 관계상 모든 소스를 다 설명할 수는 없지만 중요한 부분을 골라서 설명하겠다.
기본적인 테크닉은 앞에서 설명한 Shell_NotifyIcon 함수를 이용해서 TNotifyIconData 구조체를 등록, 수정, 삭제하고 메시지를 처리하는 것이다.
먼저 컴포넌트를 설계하도록 하자. 이 컴포넌트에서는 트레이 아이콘의 형태를 지정할 수 있는 Icon, 툴팁을 지정할 수 있는 Hint, 그리고 트레이 아이콘을 오른쪽 버튼 클릭할 때 보여줄 팝업 메뉴를 설정할 때 사용할 PopupMenu 프로퍼티를 제공하도록 하자. 또한 이벤트로는 트레이 아이콘을 클릭할 때 발생하는 OnClick, 더블 클릭할 때 발생하는 OnDblClick, 그리고 폼을 트레이 아이콘에 최소화하거나 정상 형태로 복귀할 때 각각 발생하는 OnHide, OnShow 이벤트를 제공하도록 한다.
다음은 TTrayIcon 컴포넌트의 선언 부분이다.

```
  TTrayIcon = class(TComponent)
  private
    FHandle: HWnd;
    FIconData: TNotifyIconData;
    FIcon: TIcon;
    FHint: string;
    FPopupMenu: TPopupMenu;
    FOnClick: TMouseEvent;
    FOnDblClick: TNotifyEvent;
    FOnHide: TNotifyEvent;
```

```
    FOnShow: TNotifyEvent;
    procedure SetHint(const Hint: string); virtual;
    procedure SetIcon(Icon: TIcon); virtual;
    procedure WndProc(var Message: TMessage);
  protected
    procedure DoMenu; virtual;
    procedure Click(Button: TMouseButton); virtual;
    procedure DblClick; virtual;
    procedure EndSession; virtual;
    procedure Changed; virtual;
  public
    constructor Create(Owner: TComponent); override;
    destructor Destroy; override;
    procedure Hide(Sender: TObject); virtual;
    procedure Show(Sender: TObject); virtual;
  published
    property Hint: string read FHint write SetHint;
    property Icon: TIcon read FIcon write SetIcon;
    property PopupMenu: TPopupMenu read FPopupMenu write FPopupMenu;
    property OnClick: TMouseEvent read FOnClick write FOnClick;
    property OnDblClick: TNotifyEvent read FOnDblClick write FOnDblClick;
    property OnHide: TNotifyEvent read FOnHide write FOnHide;
    property OnShow: TNotifyEvent read FOnShow write FOnShow;
  end;
```

한번쯤 컴포넌트를 직접 만들어보신 분이라면 그다지 어렵지 않게 이해할 것이다.
그렇지만, 앞에서 설명한 3 개의 프로퍼티와 4 개의 이벤트와 직접 연관되지는 않지만 내부적으로 사용되는 데이터 필드와 메소드를 소개하면 다음과 같다.
FHandle 은 WndProc 콜백 함수가 사용될 어플리케이션 윈도우에 대한 핸들을 저장한다. 또한, FIconData 는 트레이 아이콘을 등록, 수정, 삭제할 때 사용할 TNotifyIconData 구조체 변수이다.
protected 섹션에 정의된 메소드 들중 Click 과 DblClick 은 OnClick, OnDblClick 이벤트를 실제 구현할 메소드 이다. EndSession 은 앞에서 잠깐 설명한 바 있으므로 생략한다.
Changed 메소드는 Set 메소드에서 프로퍼티에 저장된 TNotifyIconData 구조체 필드의 내용을 실제로 반영하는 메소드로 이 메소드와 Set 메소드들은 다음과 같이 구현된다.

```
procedure TTrayIcon.Changed;
begin
  if not (csDesigning in ComponentState) then
    Shell_NotifyIcon(NIM_MODIFY, @FIconData);
end;

procedure TTrayIcon.SetHint(const Hint: string);
begin
  if FHint <> Hint then
  begin
    FHint := Hint;
    StrPLCopy(FIconData.szTip, Hint, SizeOf(FIconData.szTip) - 1);
    if Hint <> '' then
      FIconData.uFlags := FIconData.uFlags or NIF_TIP
    else
      FIconData.uFlags := FIconData.uFlags and not NIF_TIP;
    Changed;
  end;
end;

procedure TTrayIcon.SetIcon(Icon: TIcon);
begin
  if FIcon <> Icon then
  begin
    FIcon.Assign(Icon);
    FIconData.hIcon := Icon.Handle;
    Changed;
  end;
end;
```

즉, Changed 메소드는 어플리케이션이 델파이의 디자인 모드에 있지 않으면 FIconData 변수의 필드 값을 Shell_Notify 함수를 이용해서 반영하는 것이다.
그리고, 각각의 Set 메소드 들은 먼저 FIconData 변수에 해당 필드와 프로퍼티로 나타내기 위해서 필드 데이터(FIcon, FHint)에 값을 저장하고 Changed 메소드를 호출한다.
DoMenu 메소드는 팝업 메뉴가 선택되었을 때 메뉴를 보여주는 메소드로 Click 메소드에 의해서 호출되며 다음과 같이 구현된다.

```
procedure TTrayIcon.DoMenu;
var
  Pt: TPoint;
begin
  if (FPopupMenu <> nil) and not IsWindowVisible(Application.Handle) then
  begin
    GetCursorPos(Pt);
    FPopupMenu.Popup(Pt.X, Pt.Y);
  end;
end;

procedure TTrayIcon.Click(Button: TMouseButton);
var
  MousePos: TPoint;
begin
  GetCursorPos(MousePos);
  if (Button = mbRight) then DoMenu;
  if Assigned(FOnClick) then FOnClick(Self, Button, [], MousePos.X, MousePos.Y);
end;
```

그다지 어렵지 않은 코드이므로 자세한 설명은 생략하도록 한다.
컴포넌트를 구현한 다른 부분들은 이달에 디스켓으로 제공되는 소스를 직접 참고하기 바라며, 마지막으로 Create 생성자를 구현하는 코드에 대해서 살펴보자.

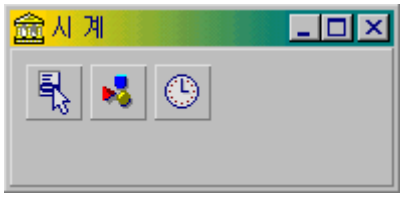
```
constructor TTrayIcon.Create(Owner: TComponent);
begin
  inherited Create(Owner);
  FIcon := TIcon.Create;
  FIcon.Assign(Application.Icon);
  FHandle := AllocateHwnd(WndProc);
  if not (csDesigning in ComponentState) then
  begin
    FillChar(FIconData, SizeOf(FIconData), 0);
    with FIconData do
    begin
```

```
        cbSize := SizeOf(FIconData);
        Wnd := FHandle;
        hIcon := Icon.Handle;
        uFlags := NIF_ICON or NIF_MESSAGE;
        uCallbackMessage := WM_CallbackMessage;
      end;
      StrPLCopy(FIconData.szTip, Application.Title, SizeOf(FIconData.szTip) - 1);
      if Application.Title <> '' then
        FIconData.uFlags := FIconData.uFlags or NIF_TIP;
      if not Shell_NotifyIcon(NIM_ADD, @FIconData) then
        raise EOutOfResources.Create('트레이 아이콘을 생성할 수 없습니다 !');
      Application.OnMinimize := Hide;
      Application.OnRestore := Show;
    end;
end;
```

즉, 초기 아이콘으로 어플리케이션 객체의 아이콘을 그대로 사용하게 하였다. 또한, 등록하는 윈도우를 AllocateHwnd 함수를 이용해서 이 컴포넌트의 WndProc 함수가 소속된 윈도우를 찾는다. 그리고, 에러 처리 부분을 추가 했으며 어플리케이션 객체가 최소화 하거나 복귀할 때에 TTrayIcon 컴포넌트의 Hide, Show 메소드에서 처리하도록 한다.

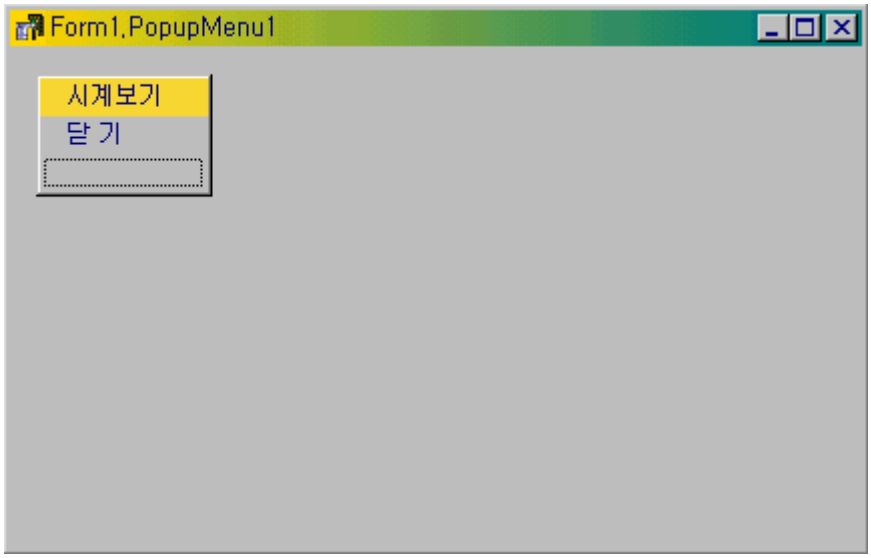
TTrayIcon 컴포넌트를 이용한 트레이 아이콘 어플리케이션의 제작

그럼 이제 TTrayIcon 컴포넌트를 이용해서 실제로 트레이 아이콘 어플리케이션을 제작해 보자. 먼저 이달의 디스켓으로 제공되는 TrayIcon.zip 파일의 압축을 풀고 TrayIcon.pas 유닛을 이용해서 컴포넌트를 설치한다. 그러면 'Samples' 팔레트에 TTrayIcon 컴포넌트가 추가될 것이다.
새로운 어플리케이션을 시작하고, 폼을 다음과 같이 디자인 한다.



즉, TPopupMenu, TPanel, TTimer, TTrayIcon 컴포넌트를 하나씩 올려 놓는다. 이때 Panel1 의 Align 프로퍼티를 alClient 로 설정해서 폼에 꽉 차도록 한다. 또한, Caption 프

로퍼티는 시간을 알려주게 할 것이므로 그냥 비워둔다.
그리고 팝업 메뉴에는 '시계보기'와 '닫 기'를 다음 그림과 같이 설정한다.



이제 TrayIcon1 컴포넌트의 프로퍼티를 설정해 보자. 디자인 시에는 PopupMenu 프로퍼티만 콤보 박스를 이용해서 PopupMenu1 으로 설정하면 된다.
그리고 팝업 메뉴들에 대한 이벤트 핸들러를 다음과 같이 작성한다.

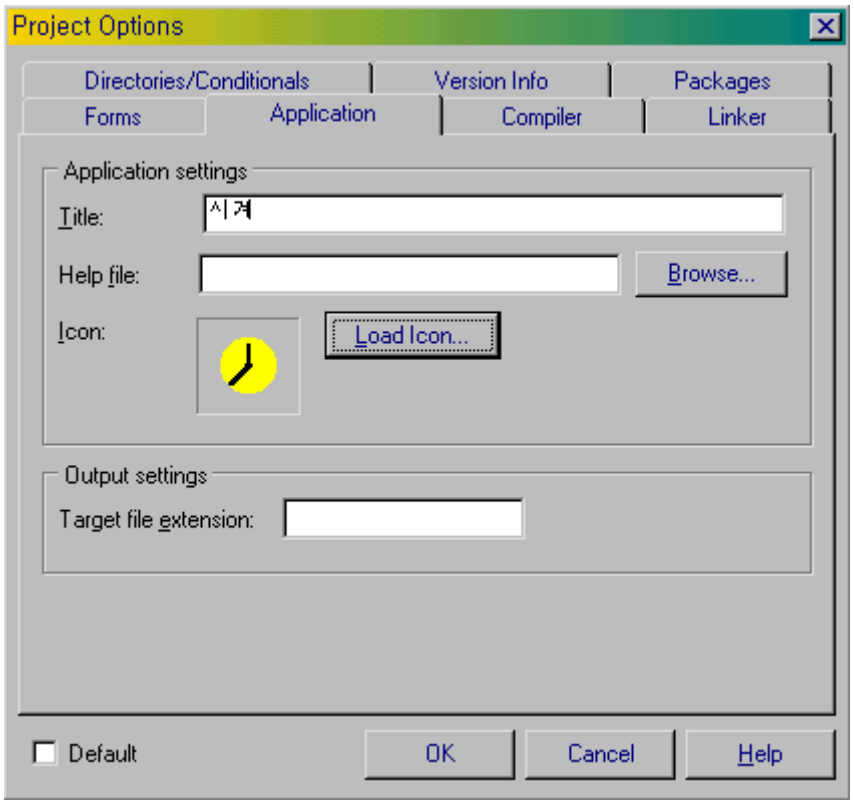
```
procedure TForm1.N1Click(Sender: TObject);
begin
  Application.Restore;
end;

procedure TForm1.N2Click(Sender: TObject);
begin
  Form1.Close;
end;
```

즉, '시계 보기' 메뉴일 경우에는 Application 객체의 Restore 메소드를 사용해서 폼을 보여주게 하고, '닫 기'일 경우에는 폼의 Close 메소드를 호출해서 어플리케이션을 종료한다.
이때 Application.Restore 메소드는 TTrayIcon 컴포넌트의 Show 메소드를 호출하도록 변경되어 있으므로 TrayIcon1.Show 와 같은 기능을 하게 된다.
그리고, Timer 의 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Timer1Timer(Sender: TObject);
```

```
begin
  Panel1.Caption := TimeToStr(Time);
  TrayIcon1.Hint := TimeToStr(Time);
end;
```

현재 시각을 문자열로 변환해서 Panel1 과 트레이 아이콘의 툴팁으로 설정한다.
이제 사용할 어플리케이션 아이콘과 초기 툴팁을 결정하기 위해 Project|Options ... 메뉴를 클릭한 후 Application 탭을 선택하고, 다음 그림과 같이 설정한다. 이때 사용한 아이콘 파일은 필자가 이미지 에디터로 급히 그린 것이라 그다지 아름답지 못하므로 독자 여러분들의 너그러운 양해를 바란다. 원래 계획으로는 필자가 트레이 아이콘을 이용해서 움직이는 공룡 캐릭터와 타이머를 이용해서 자그마한 다마고치 예제를 제공하려고 했으나, 필자의 조악한 그림 실력으로 인해 이렇게 시계 예제로 전환하였다. 관심 있는 분들은 TTrayIcon 컴포넌트를 이용해서 쉽게 구현할 수 있을 것이다.

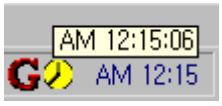


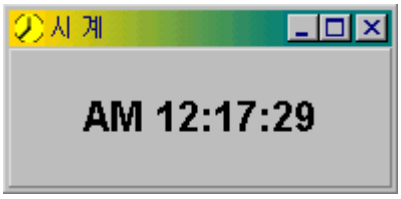
이제 마지막으로 처음에도 언급 했듯이 어플리케이션이 시작할 때 트레이 아이콘 상태로 시작하도록 하자. 필자도 이 방법을 찾기 위해 별별 방법을 다 써 보았는데, 해답은 생각보다 가까운 곳에 있었다. 즉, 폼의 OnActivate 이벤트에서 Application.Minimize 를 호출해

주면 된다.

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  Application.Minimize;
end;
```

이제 시계 예제가 완성되었다. 프로그램을 실행하면 다음 그림과 같이 동작하는 화면들을 얻을 수 있을 것이다.





정 리 (Summary)

트레이 아이콘 어플리케이션을 만드는 방법은 아마도 윈도우 95 와 윈도우 NT 4.0 쉘을 수정해서 사용하는 방법 중에서 가장 유용하고 간단하다고 말할 수 있다. 특히 도스 시절의 램상주 프로그램처럼 간단한 유틸리티를 제작할 때에는 그 활용도가 매우 높다고 할 수 있다. 이미 많은 수의 유틸리티 프로그램들이 제공되고 있는데, 이들 중에는 시스템을 모니터하거나, 경고를 주는 등의 유용한 유틸리티 들이 있으며 대부분의 경우 트레이 아이콘 어플리케이션의 형태로 제공되고 있다.