Windows 95 Tray Icons

by Marco Cantù

In the DOS era there were TSR programs and in Windows 3.1 some applications run in an iconized form, but in Windows 95 we have *tray icons*.

In the lower right corner of the screen, near the clock, there is some space (the TaskBar tray) you can use to show your programs or utilities. This implies removing the standard TaskBar icon and providing a popup menu when the user right clicks on the icon. Besides this, there is just one API function involved, `Shell_NotifyIcon`. So, at first sight, writing a similar program in Delphi 2 seems quite simple. The problem is that there are a couple of tricks you have to use to make the application behave properly...

The Shell_NotifyIcon Function

The only API function involved is very simple. It has two parameters: a pointer to a `TNotifyIconData` data structure and a flag indicating whether you want to add, remove or modify the icon. The first field of the data structure (`cbSize`) is its size (which is used by the system to determine the version), then there is the handle of the window the TrayIcon should send notifications to (`hWnd`) and the identifier of the icon (`uID`), useful if an application has several icons. Then comes the `uFlags` field, with three possible flags (`nif_Message`, `nif_Icon` and `nif_Tip`), indicating which of the final three fields is valid: `uCallbackMessage` (the `wm_User` message sent to the handle to notify a user action on the icon), `hIcon` (the icon to display) and `szTip` (the text of the tip displayed when the mouse moves over the icon).

In practice, besides the icon and the tip, the other fields of this structure indicate the window to notify and the message to send to it. In fact, when the user interacts with the tray icon, it sends back to the given window a user-defined message passing as parameters the action performed by the user on the icon (typically a mouse message) and the id given to the icon.

An Example Of The Call

In Listing 1 you can see an example of how to call the function to create a new tray icon and in Figure 1 you can see its effect. This code is preceded by the definition of the message constant:

```
const
  wm_IconMessage = wm_User;
```

How do we handle a message? Simply by adding a new message handler method to the form receiving the message:

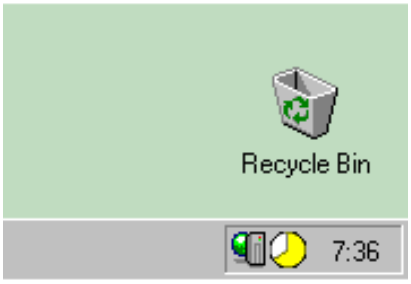
```
public
  procedure IconTray(
    var Msg: TMessage);
    message wm_IconMessage;
```

Inside the message handler we can perform different actions, depending on the user operation (passed in `lParam`):

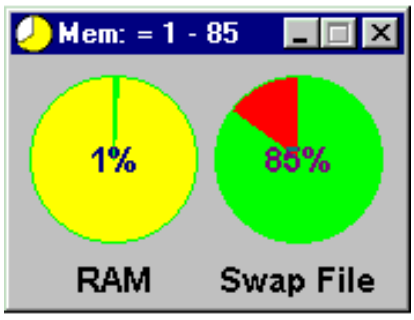
```
if Msg.lParam = wm_rbuttondown
  then...
```

A Full-Blown Example

Instead of just showing you some small code excerpts, I decided to build a full blown program, extending an example from my book *Mastering Delphi 2*. The example program basically shows the amount of free memory using two gauge controls, as you can see in Figure 2.



➤ *Figure 1: A custom tray icon*



➤ *Figure 2: The original main window of the example*

The program is based on a timer, which repeatedly calls the API function `GlobalMemoryStatus`, used to retrieve the amount of physical and free RAM, and the amount of physical and free total memory (swap file plus RAM). The API function `GlobalMemoryStatus` fills a `TMemoryStatus` structure with these and other values you can find in the Win32 API help file.

In Listing 2 you can see the source code of the original version of the example. It determines the `Progress` property of the two gauges, sets the caption of the label, then chooses a proper icon.

➤ *Listing 1: Calling Shell_NotifyIcon*

```
var nid: TNotifyIconData;
begin
  nid.cbSize := sizeof(nid);
  nid.wnd := Handle;
  nid.uID := 1; // icon ID
  nid.uCallBackMessage := wm_IconMessage;
  nid.hIcon := Icon.Handle;
  nid.szTip := 'Free memory';
  nid.uFlags := nif_Message or
    nif_Icon or nif_Tip;
  Shell_NotifyIcon(NIM_ADD, @nid);
```

The green color indicates free RAM, the yellow icon indicates full RAM but available virtual memory, the red one indicates that the free memory is dangerously low (which means the swap file has filled your hard disk).

The first change I've made is to show the amount of free and total memory:

```
Label1.Caption := Format(
  'RAM:'#13'%s'#13'(%s)',
  [FmtMem(MemInfo.dwAvailPhys),
  FmtMem(MemInfo.dwTotalPhys)]);
```

The same code is applied to the other label as well. The `FmtMem` function is a custom routine I've written to make the information more readable, adding the `KB` or `MB` strings to the proper value:

```
function FmtMem(N: Integer):
  string;
begin
  if N > 1024*1024 then
    FmtMem :=
      Format('%.1f MB',
      [n / (1024*1024)])
  else
    FmtMem :=
      Format('%.1f KB',
      [n / 1024])
end;
```

You can see an example of the updated output in Figure 3. But the key change I've made to the `Timer1Timer` method is the code which updates the tray icon. Since the program already selects an icon for the form, I've updated it simply by selecting the new icon and a proper message for the tray:

```
nid.hIcon := Icon.Handle;
strcopy(nid.szTip,
  PChar(Caption));
nid.uFlags :=
  nif_Icon or nif_Tip;
Shell_NotifyIcon(
  NIM_MODIFY, @nid);
```

This code simply updates the `nid` structure with the icon and the caption (after a cast to `PChar`) then updates the icon. In the example I declare the nid structure as a field of the form, so that I can fill it when the form is created with the code in Listing 1 (showing the tray icon for the first time), then update only the few fields that change.

```
procedure TMemForm.Timer1Timer(Sender: TObject);
var MemInfo : TMemoryStatus;
begin
  MemInfo.dwLength := Sizeof(MemInfo);
  GlobalMemoryStatus(MemInfo);
  RamGauge.Progress := MemInfo.dwAvailPhys div
    (MemInfo.dwTotalPhys div 100);
  VirtualGauge.Progress := MemInfo.dwAvailPageFile div
    (MemInfo.dwTotalPageFile div 100);
  Caption := Format('Memory: = %d - %d',
    [RamGauge.Progress, VirtualGauge.Progress]);
  // set icon color
  if RamGauge.Progress > 5 then
    Icon.Handle := LoadIcon(HInstance, 'GREEN')
  else if VirtualGauge.Progress > 20 then
    Icon.Handle := LoadIcon(HInstance, 'YELLOW')
  else
    Icon.Handle := LoadIcon(HInstance, 'RED');
end;
```

➤ *Listing 2: The original version of the Timer response method*

When the program terminates, in the `OnDestroy` event handler the icon is removed with yet another call to `Shell_NotifyIcon`:

```
nid.uFlags := 0;
Shell_NotifyIcon(NIM_DELETE,
  @nid);
```

If you forget to remove the icon, it simply disappears the first time you move the mouse over it, but it is certainly better to remove it when the application is closed.

Hiding The Main Window

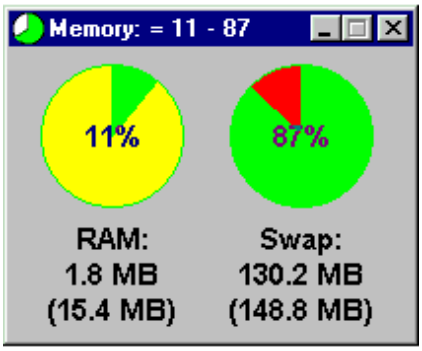
The main form of this application (shown in Figure 3) should become visible only when the user double-clicks on the icon in the tray. This is not too difficult to accomplish. Simply add a line to the source code of the project:

```
Application.ShowMainForm :=
  False;
```

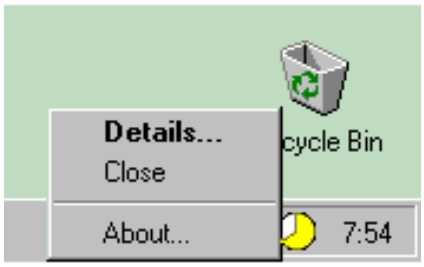
before you call the `CreateForm` method. Then use these two lines to show the form:

```
procedure TMemForm.Details1Click(
  Sender: TObject);
begin
  ShowWindow(
    Handle, sw_ShowNormal);
  SetForegroundWindow(Handle);
end;
```

The second line is required when



➤ *Figure 3: The updated output with more details*



➤ *Figure 4: The popup menu of the tray icon*

another application is active and you click the icon. This method is associated with the default item of a popup menu component I've added to the form. The other items are `Close` and `About`. The popup menu is not connected to the form, but displayed when a user right-clicks on the icon, as you can see in Figure 4. When the user double clicks on the icon the form is immediately displayed, as you can see in the code in Listing 3. By the way, these two choices (right click for the local menu and left double click

for the default action are mandatory in the *Windows User Interface* guidelines).

Again we have to call the SetForegroundWindow function, but this time it serves to display the popup menu properly (that is, in front of everything else). Try commenting out this line of code and you'll see what happens: the popup menu of the tray icon remains on the screen even when you activate another program.

The Close menu command closes the form. However, I want to be able to simply hide the form when the user clicks on its Close button. The solution? I've set a Closing flag to False at the beginning, then I've written this code for the OnClose event handler:

```
procedure TMemForm.FormClose(
  Sender: TObject;
  var Action: TCloseAction);
begin
  if not Closing then begin
    Action := caNone;
    ShowWindow(Handle, sw_Hide);
  end;
end;
```

Strangely enough, setting Action to caHide terminates the program, so I call ShowWindow to hide the form. When I really want to close the form, I set Closing to True, as in the handler of the OnClick event of the Close menu item:

```
procedure TMemForm.Close1Click(
  Sender: TObject);
begin
  Closing := True;
  Close;
end;
```

Hiding The TaskBar Icon

So far we've been able to show a tray icon, update it when the system status changes (using a timer) and let a user select a couple of commands from a popup menu connected with the icon. And we've been able to fix a Windows 95 glitch by calling the function SetForegroundWindow function before the popup menu is displayed.

However, there is still one big problem. When you run the program the main form is hidden but the application window is visible, resulting in an application icon in the task bar, plus the icon in the tray area of the TaskBar. This is not what these kind of applications generally do.

```
procedure TMemForm.IconTray(var Msg: TMessage);
var Pt: TPoint;
begin
  if Msg.lParam = wm_rbuttondown then begin
    GetCursorPos(Pt);
    SetForegroundWindow(Handle);
    PopupMenu1.Popup(Pt.x, Pt.y);
  end;
  if Msg.lParam = wm_lbuttondblclk then
    Details1Click(self);
end;
```

➤ *Listing 3: The handler method of the icon 'callback' message*

How can we hide the TaskBar icon, representing the application? The first idea is to hide its window (Application.Handle) but this means the TaskBar icon is created and displayed first, then it is removed. You'll probably notice only a small flicker, but this is far from professional.

An alternative is to disable the creation of the application window, by setting the global variable IsLibrary to True. In fact, looking at the VCL source code in the constructor TApplication.Create you can see the following code:

```
if not IsLibrary then
  CreateHandle;
```

So we can make Delphi think we are a DLL and do not require a main window. This is a risky trick and I've found some strange behaviour when you close the program (the OnDestroy event handler is not called). For this reason I've decided to set IsLibrary to True and then set it back to False as soon as possible. But the real problem is to set this global variable to True before the global Application object is created. This happens in the initialization section of the Controls unit. So the question becomes, how can we execute some code before this unit is initialized? Writing something like:

```
IsLibrary := True;
Application.CreateForm(
  TMemForm, MemForm);
Application.Run;
```

doesn't help at all. In fact units are initialized before Delphi executes the code of the project file. The solution lies in the fact that units are initialized in the order they are listed in the project file. By default this is:

```
uses
  Forms,
  Resform in
    'RESFORM.PAS' {MemForm};
```

However, ResForm includes Forms (which includes Controls) so even in the initialization code of that unit it is already too late. To solve the problem I've added a new unit to the program, with this plain code:

```
unit RunFirst;
interface
implementation
initialization
  IsLibrary := True;
end.
```

Then I've changed the source code of the project file as shown in Listing 4. Notice that the RunFirst unit is listed first in the uses clause (to initialize it first) and that I set the IsLibrary variable back to False immediately.

Although I cannot swear everything is completely OK, I haven't seen any drawbacks in disabling the creation of the application window.

The effect of this is that as soon as the main form becomes visible, its icon is added to the task bar, something you usually don't see in a Delphi application. This is probably not the standard (tray icon windows usually have no entry in the task bar even when visible), but I think I do like it, so I've left the code as it is.

```
program Mem;
uses
  RunFirst in 'RunFirst.pas',
  Forms, Windows,
  Resform in 'RESFORM.PAS' {MemForm};
{$R *.RES}
begin
  Application.ShowMainForm := False;
  IsLibrary := False;
  Application.CreateForm(TMemForm, MemForm);
  Application.Run;
end.
```

➤ *Listing 4: Updated project file*

Conclusion

Writing tray icon applications is probably the simplest thing you can do to customize the Windows 95 shell (and the new Windows NT 4.0 shell as well). There are many more things you can do that I'm exploring (as you've probably guessed!) for a new advanced book I'm writing.

For sure Delphi allows you to do everything, but at times there are some hurdles you have to jump over (or run around) to make things work properly. Needless to say these challenges can be fun and are a good way of exploring the Delphi architecture in detail.

Marco Cantù is a writer and consultant who lives in Italy (when he's not travelling, of course). His last book *Mastering Delphi 2 for Windows 95/NT* (published by SYBEX) is only 1000 pages so he's collecting new material for a follow-up. You can reach Marco at 100273.2610@compuserve.com or visit his home page at:
http://ourworld.compuserve.com/homepages/marcocantu