Beating The System: Taming The Windows Desktop, Part 3

by Dave Jewell

Let's briefly review the story so far. Two months ago, I introduced you to DIPSLIB.DLL, created by Jeffrey Richter, describing how it worked and enlightening you with the joys of shared data segments. I used a slightly massaged version of this DLL to communicate with a host application written in Delphi, thus demonstrating how to save and restore the current desktop layout. Although that was great as far as it went, the need to use a DLL written in C/C++ was something of a thorn in my flesh, and I eventually realised how to completely eliminate the need for a shared data segment, thereby making it possible to rewrite the DLL using Delphi. At the same time, I wanted to get away from the all-or-nothing approach to saving the desktop layout. My aim was to create something that gave much finer control over individual desktop items and, as part of this strategy, we looked at the wondrous `WM_COPYDATA` message, which I'm now using to retrieve information on individual desktop items.

This month, we'll round off this look at programmatic desktop management by adding a lot more functionality to the DLL, and we'll rewrite the code on the application side, packaging everything into a reusable component that can be easily used from any Delphi application.

Introducing TDesktopManager

`TDesktopManager` is the name of our new Delphi component. The most straightforward approach here is to make `TDesktopManager` a non-visual component (ie, inherit from `TComponent`), but if you want to create a visual component that graphically displays a tiny real-time representation of the desktop, allows you to drag tiny desktop icons around in this control and have the real desktop items follow suit then... err... you're probably a very strange person.

Because of the requirement to inject a DLL into the address space of Windows Explorer, I decided to add an `Active` property to the component. This `Boolean` property (which is `False` by default) determines whether or not the DLL is currently injected into Explorer's process space. Setting it to `True` will automatically inject the DLL and setting it to `False` will obviously do the reverse.

I chose to do things this way because I felt it would be rather inflexible if the mere act of placing a `TDesktopManager` component caused the DLL to be needlessly injected regardless of whether or not one made use of the component, and for the whole length of time that the host application is running. After all, the whole point of the Win32 protection mechanism is to prevent one badly-behaved application from damaging others, or crashing the entire operating system. Murphy's First Law dictates that if something can go wrong, it will, so it makes sense to be able to programmatically control when those barriers are raised and lowered. While the component is in the inactive state, it obviously can't use the DLL to communicate with the desktop listview control.

Listing 1 shows the `SetActive` method, which forms the 'setter' routine for the `Active` property. Having established that the `Active` property is being set, the code first checks to see if the hidden dialog can be found and bottles out with an appropriate exception if so. As I've discussed before, it's critically important that the DLL shouldn't be allowed to hook Explorer more than once: if it does, deeply bad things will happen.

The code presented here has been tested and found to be very robust under Windows 2000, Windows ME and (according to a reader who kindly emailed me) NT 4.0. However, you should appreciate that if *some other utility* has already hooked Explorer in a similar fashion, then all bets are off. While developing this code, I experienced a number of crashes which went away as soon as I removed the *On Display* component of MicroAngelo 5.0 from my system.

➤ *Listing 1*

```
procedure TDesktopManager.SetActive (Value: Boolean);
var Msg: TMsg;
begin
  if fActive <> Value then begin
    if Value then begin
      if FindWindow (Nil, 'Delphi Desktop 2001') <> 0 then
        raise EDesktopManager.Create(
          'Another instance of ' + ClassName + ' is already active.')
      else begin
        DeskManagerLoad (fAppWindow);
        GetMessage (Msg, fAppWindow, DM_DLLReady, DM_DLLReady);
        fDeskWin := FindWindow (Nil, 'Delphi Desktop 2001');
        if fDeskWin = 0 then raise
          EDesktopManager.Create('DLL initialization failed');
        fActive := True;
      end;
    end else begin
      SendMessage (fDeskWin, wm_Close, 0, 0);
      while IsWindow (fDeskWin) do
        Application.ProcessMessages;
      DeskManagerUnload;
      fActive := False;
    end;
  end;
end;
```

On Display, in case you don't know, is a desktop utility for controlling icon spacing, foreground and background text colours, and various other characteristics of your desktop. I felt sure that, somehow, my code and the *On Display* code were both trying to do the same thing and causing problems. It's really a similar problem to that of the old DOS days when two TSRs tried to hook the same interrupt. Caveat programmer!

Assuming that the call to `FindWindow` returns zero, the `DeskManagerLoad` routine is called as described last month. You'll notice that I've slightly re-jigged the subsequent call to `GetMessage` in the interests of robustness. Rather than just calling `GetMessage` to wait for any incoming message on the current thread, the call now waits for one specific message, `DM_DLLReady`, to be received by `fAppWindow`, about which more below. In order to achieve this, the code inside `GetMsgProc` (within the DLL) now posts to a specific window rather than using `PostThreadMessage`:

```
PostMessage(AppWindow,
  DM_DLLReady, 0, 0);
```

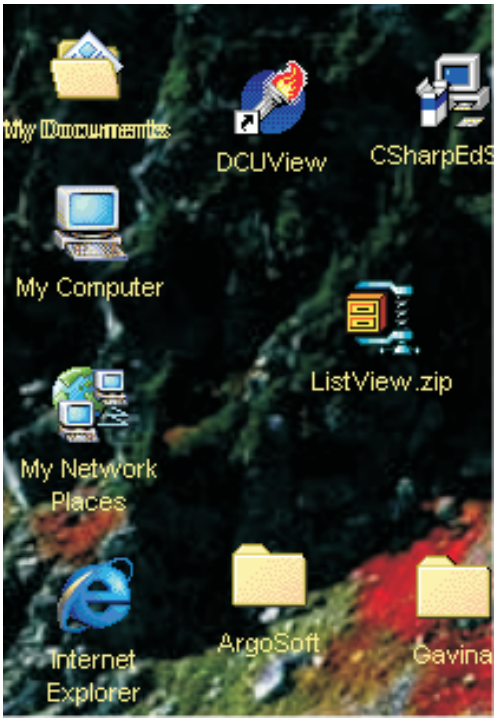
The Deeply Fragrant AllocateHWnd Function

Whenever you implement a non-visual Delphi component that needs to receive messages from some other window, you're faced with an interesting dilemma. There's a fairly obvious requirement: a window to receive the messages. But which window to use? Even non-visual components are generally sat on a form, and you can easily use the `Owner` property to find the form itself. However, this means that you've got to start hooking the message handling of your owner form in order to retrieve the messages you want. It can be done, but unless you're careful it can get quite ugly. Is there an easier way?

As it happens, there is. Let me introduce you to `AllocateHWnd`, a very useful but little-known function in FORMS.PAS. It's not too surprising that this function is largely undiscovered, since the Borlandites seem to have removed all reference to it from the Delphi 5 online help! However, you'll quickly forgive those denizens of Scott's Valley once you realise what a wondrous little gem they've written for you:

```
function AllocateHWnd(
  Method: TWndMethod): HWND;
```

Ordinarily, if you were creating a custom window for exclusive use by a component, you'd have to first register a window class, and then you'd need to create the window; oh, and you'd also have to set up a window message handling function, etc, etc. All very tedious using the stone knives and bearskins provided by the Windows API. Fortunately, `AllocateHWnd` does the whole thing for you in one easy step: just call this routine and it will give you back a genuine window handle whose Windows procedure is automatically set to the specified method. Ordinarily, a window procedure can't be a method of a Delphi class because of different calling conventions, the implicit `Self` parameter, and other issues. However, `AllocateHWnd` calls another auto-magical routine, `MakeObjectInstance`, which transmogrifies an ordinary method into something that can be used as a window procedure at the API level. To see how this all works, look at Listing 2.



➤ *Figure 1: Here's an example of what will happen if you don't force the desktop window to perform a redraw when changing a caption or repositioning an item. As you can see, the caption for 'My Documents' is in glorious stereo!*

```
constructor TDesktopManager.Create (AOwner: TComponent);
begin
  Inherited Create (AOwner);
  fAppWindow := AllocateHWnd (ReceiverHook);
end;
destructor TDesktopManager.Destroy;
begin
  // Bow out gracefully...
  SetActive (False);
  DeallocateHWnd (fAppWindow);
  Inherited Destroy;
end;
```

➤ *Listing 2*

`fAppWindow` is a private field of the `TDesktopManager` class and is initialised in the component's constructor, passing to `AllocateHWnd` the address of the `ReceiverHook` method which handles `WM_COPYDATA` responses returned from the DLL. In a similar vein, the destructor first sets the component as inactive, causing the DLL to be released from Explorer's address space. The private window is then destroyed and the inherited destructor is called. With a component such as this, cleanup code is obviously very important, if the application were able to exit without calling `DeskManagerUnload`, then it would definitely be tears before bedtime.

At this point, it's time to start adding the bells and whistles. I began by adding some simple functionality to get and set the text colour and background colour

used to display icon captions on the desktop. This was implemented via the two property declarations shown here:

```
property TextColor: TColorRef
  index 0 read GetColor
  write SetColor stored False;
property BackgroundColor:
  TColorRef index 1
  read GetColor write SetColor
  stored False;
```

I made a decision that `TDesktopManager` would implement as much functionality as possible without the need to set the `Active` property and inject the DLL. Based on what we've learnt so far, this means that anything which doesn't require passing pointers to/from the desktop listview control can be implemented without worrying about the `Active` flag. If you try and do something which requires the active state (eg retrieving the caption of a desktop item) then an `EDesktopManager` exception will be raised. To help enforce this distinction, I placed the 'always-available' properties in the component's published section so that (type permitting!) they're visible to the Object Inspector. Those properties which require the active state have `public` access.

While writing the `SetColor` routine (see Listing 3), I discovered an interesting quirk. As you can see from the code, whenever the foreground or background text colour is changed, the `RedrawItems` routine is called to force the desktop to redraw each item (the code to do this was given last month). However, Microsoft document the fact that if you use a special colour value of `CLR_NONE` (which is actually `$FFFFFFFF` and therefore not really a colour at all) it's interpreted as a transparent background colour, thus giving a very attractive effect when you've got a background bitmap on your desktop. Unfortunately, there seems to be a bug in the listview control, such that setting `CLR_NONE` as the background colour causes the redrawing code to be ignored. After a little experimentation, I added a 'hit-me-with-a-big-stick' flag (!) to `RedrawItems`. This forces a redraw by hiding and then immediately showing the desktop window. Naturally, I only use this option where strictly necessary in order to minimise possible flicker problems.

➤ *Listing 3*

```
procedure TDesktopManager.SetColor (Index: Integer; Value: TColorRef);
begin
  if GetColor (Index) <> Value then begin
    case Index of
      0:  ListView_SetTextColor (DeskManagerListView, Value);
      1:  ListView_SetTextBkColor (DeskManagerListView, Value);
    end;
    RedrawItems ((Value = CLR_None) and (Index = 1));
  end;
end;
```

➤ *Above: Listing 4*

```
function TDesktopManager.GetPosition (Index: Integer): TPoint;
begin
  NeedActive;
  if (Index < 0) or (Index >= GetItemCount) then
    Result := Point (-1, -1)
  else
    Result := PPoint (PendOnDesktopMessage (DM_GetItemPosition, Index, 0))^;
end;
```

➤ *Below: Listing 5*

```
function HandleGetItemPosition (DlgWnd: hWnd; Index: Integer): Integer;
var
  pt: TPoint;
  Count: Integer;
  cds: TCopyDataStruct;
begin
  Result := 0;
  Count := SendMessage (DeskManagerListView, lvm_GetItemCount, 0, 0);
  if (Index >= 0) and (Index < Count) then begin
    ListView_GetItemPosition (DeskManagerListView, Index, pt);
    cds.dwData := DM_GetItemPosition;
    cds.cbData := sizeof (pt);
    cds.lpData := @pt;
    Result := SendMessage (AppWindow, wm_CopyData, DlgWnd, Integer (@cds));
  end;
end;
```

Location, Location, Location!

My next job was to add some facility for retrieving the captions and positions of a specified item. This was done through the property declarations given below:

```
property Caption[
  Index: Integer]: String
  read GetCaption
  write SetCaption;
property Position[
  Index: Integer]: TPoint
  read GetPosition
  write SetPosition;
```

With some minor tweaking, the `GetCaption` routine is essentially the same as the code I showed you last month, so I won't go over the same ground again. The `GetPosition` routine is as shown in Listing 4. `NeedActive` is a frequently used method which simply checks to see if the `Active` property is set, raising an exception if not.

The general principle is the same: validate the index argument and then call the `PendOnDesktopMessage` (also discussed last time round) to send the `DM_GetItemPosition` message to the DLL, spinning until we get a response. Within the DLL code, the `DM_GetItemPosition` message is fielded by `HandleGetItemPosition`, shown in Listing 5.

In this case, I suppose I could have simplified things even further by merely returning the positional information as the message result of `DM_GetItemPosition`. In other words, pack the `x` and `y` coordinates retrieved from `ListView_GetItemPosition` into a single 32-bit result. It should be noted, though, that since the underlying `LVM_GETITEMPOSITION` message requires a pointer to a `TPoint`, the use of the DLL can't be avoided in this case.

Astute readers might naturally wonder why it is that you can (for example) send a `WM_GETTEXT` message to a button control belonging to another process and successfully retrieve the caption of that button, despite the fact that you've

passed a pointer as part of the message! The reason is that, when Microsoft implemented 32-bit Windows, they added low-level code to the system which silently copies the required information from the address space of one process to that of another. This low-level code was intended to smooth the transition from the 16-bit to the 32-bit platform, and doesn't apply to the newer 'common controls' which include listview, treeview, status bar and others.

Of course, we also need to be able to set the position of a desktop item, since programmatically restoring the desktop layout is one of the primary aims of this discussion. Fortunately, this is dead easy, since Microsoft didn't use pointers in their implementation of `LVM_SETITEMPOSITION`, choosing instead to pack the `x` and `y` coordinates into `lParam`. See Listing 6.

Of course, this leaves the thorny issue of how to programmatically change the caption (rather than position) of a desktop item. Why thorny? If you look at the source code for the `ListView_SetItemTextA` routine (see COMMCTRL.PAS) you'll see that we have to pass a pointer to the required new caption string. So far, we've managed to avoid passing pointers to the hidden dialog window (which, you'll recall, is running in Explorer's address space) by the simple expedient of using `WM_COPYDATA` to retrieve responses from the DLL code. But that technique was for retrieving data *from* the DLL. In this case, we're trying to pass data *to* the DLL. Disaster!

Fortunately for us, the solution is quite simple. Just as we have used `WM_COPYDATA` to send data from the DLL, we can use exactly the same mechanism to send it to the DLL. Listing 7 gives the source code to the `SetCaption` routine that I eventually came up with.

Part way through the development of this routine, I suddenly realised that I wasn't passing `Index` (the number of the desktop item that we're interested in) through to the DLL. Duh! Unfortunately, this presents a problem because `wParam` and `lParam` are already used for the purposes of the `WM_CopyData` message itself. I therefore had no option but to include the index value into the data block which is sent to the DLL. Thus, the `dwData` field of the `TCopyDataStruct` record tells the DLL that this is a `DM_SetItemText` command. The `lpData` field points to a chunk of data which is made up of the 32-bit index value followed by the new caption string.

Regular readers will, as ever, note my fondness for the `absolute` keyword, which is very useful in cases such as this and avoids a lot of messy casting and other shenanigans. During the development of the new Kylix compiler, a certain Borlandite (who had best be nameless!) was in favour of removing `absolute` from the language specification. Myself and a number of others strongly objected, and I'm glad to say that common sense prevailed.

You'll notice that, as with the colour-changing code mentioned earlier, we need to call another new routine, `RedrawSingleItem`, to persuade the listview control to cleanly draw the new caption. If you don't do this, then the listview control will simply draw the new caption string on top of the old one when it first receives the `LVM_SetItemText` message!! Pathetic, but true! I had hoped that the `LVM_RedrawItems` message sent from `RedrawSingleItem` would be enough to force a redraw, but once again, it's necessary to hide and then show the desktop.

When it came to test out the code which repositions desktop icons, I discovered that, here again, Microsoft's redrawing code simply doesn't work and you're forced to hide/show the desktop window. This incensed me sufficiently that I decided to nail the problem once and for all. If you look at the code on this month's disk, you'll see that I've implemented a new message called `DM_GetItemRect`. This retrieves the bounding rectangle of the desktop icon (including the text caption area) by sending a `ListView_GetItemRect` message to the desktop listview control. Since this involves a pointer to a rectangle, it has to be done within the DLL. With access to the icon's bounding rectangle, we can then implement a small... uhh... kludge, whereby the icon we're interested in is moved off-screen, the previously-occupied bounding rectangle is invalidated and redrawn, and the icon is then moved back to where we want it to be before being redrawn again. This little hackette works well for the code to change captions and reposition icons, but it's obviously less appropriate when the foreground/background colours of the text captions are changed, because this affects all the items on the desktop.

➤ *Listing 6*

```
procedure TDesktopManager.SetPosition (Index: Integer; Value: TPoint);
begin
  if (Index >= 0) and (Index < GetItemCount) then
    ListView_SetItemPosition (DeskManagerListView, Index, Value.x, Value.y);
end;
```

Putting It All Together

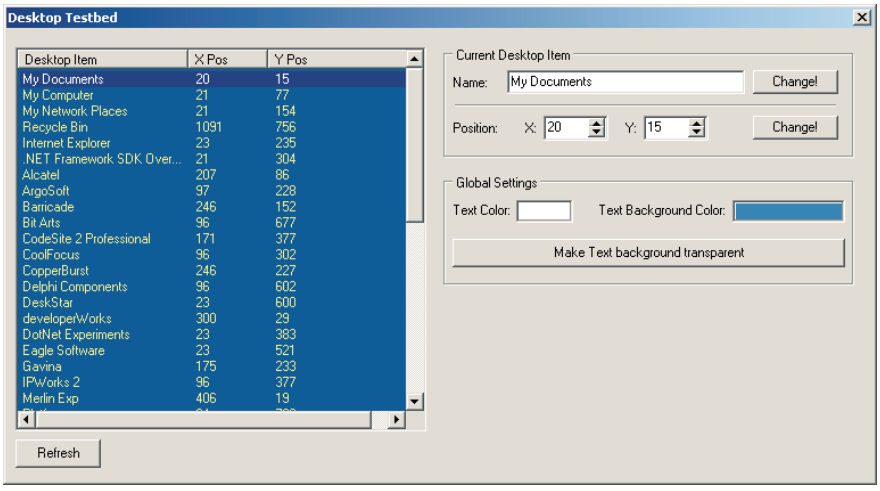
At this point, we've got reliable, robust code to change the text

➤ *Listing 7*

```
procedure TDesktopManager.SetCaption (Index: Integer; const Value: String);
var
  cds: TCopyDataStruct;
  buff: array [0..255] of Char;
  Idx: Integer absolute buff;
begin
  NeedActive;
  if (Index >= 0) and (Index < GetItemCount) then begin
    Idx := Index;
    StrPCopy (@buff [sizeof (Integer)], Value);
    cds.dwData := DM_SetItemText;
    cds.cbData := Length (Value) + 1 + sizeof (Integer);
    cds.lpData := @buff;
    SendMessage (fDeskWin, WM_CopyData, fAppWindow, Integer (@cds));
    RedrawSingleItem (Index, True);
  end;
end;
```

colour, icon captions and icon positions. We're now set fair to implement a nice little desktop layout save/restore utility. But first, I wanted to test everything out by developing a little testbed utility. This program, called DESKTESTER.EXE, can be seen running in Figure 2. As you can see, it provides facilities for viewing the captions of all available desktop items, along with their position within the listview. You can refresh the display at any time by clicking the `Refresh` button, and you can use the controls on the right hand side of the window to change things such as the name of a desktop item, its position, or the 'item-wide' colour settings. There is also a button you can click which will set a transparent background colour for the item captions. If you do this, then the *Text Background Color* panel will display the word 'transparent' rather than showing a specific colour.

Although the code for this little testbed program is reasonably straightforward, there are some important points to note here. Firstly, when you check through the code, you'll see that I set the component's `Active` property only when needed, and clear it afterwards. In other words, the DLL remains injected for only as long as it's needed. Since the DLL is already mapped into the process space of the testbed application (by virtue of the `external` references to it in DesktopManager.Pas) the process of injecting and 'un-injecting' is actually very fast. By minimising the amount of time that the DLL sits in Explorer's address space, we minimise the likelihood of unpleasant clashes with other software that uses similar techniques.



➤ *Figure 2: The testbed program serves as a vehicle for testing out the functionality of the TDesktopManager/DLL combination. Please note the comments in the text concerning re-entrancy and the disabling of the 'Refresh' button.*

While on the subject of the `Active` flag, I eventually realised that you really need to have it set to `False` before running a program from within the IDE. If you don't do this, Delphi will try to create a runtime instance of `TDesktopManager`, and fail, because there is already an active instance of the component on the design-time form. This being the case, it really makes sense to turn `Active` into a `public` rather than `published` property so that it can't be inadvertently set to `True` at design-time.

Something else you should be cautious about is changing the name of 'well-known' desktop items such as `My Documents`, `My Computer`, `Recycle Bin`, and so forth. The code I've developed here will certainly allow you to do this, but if so, it's anybody's guess how other software reacts to the disappearance of some expected desktop item. I'd err on the side of caution here, if I were you.

When working with the desktop, it's also important to realise that the number, names and positions of desktop items can potentially change at any time. For this reason, you can't simply retrieve the caption of an item, allow the user to spend five minutes typing in a new name, and then write the changed caption out to what (you hope!) is the same item. Instead, you need to add sanity checks to ensure that the item that's changed is the item you want.

This is illustrated by the code in Listing 8. Every time the listview control (the one in the testbed, not the one on the desktop!) has its selection changed, the `OriginalIdx`, `OriginalName` and `OriginalPos` fields are updated to reflect the index, initial name and initial position of the currently selected item. When it's time to alter an item, the code verifies that we're still dealing with the same control and that it's at the same position we expected it to be! As an alternative to this approach, you might care to look at the `LVM_FindItem` message which instructs the listview control to perform an item search at the API level.

Right then, on with the show. Figure 3 shows a little desktop save/restore program which I wrote using `TDesktopManager`. This program is called `DeskLayout` and is included on this month's disk along with everything else (see the

➤ *Listing 8*

```
procedure TForm1.ChangeCaptionClick(Sender: TObject);
begin
  dm.Active := True;
  try
  // Check that the target item still exists with same index
  if dm.Caption [OriginalIdx] = OriginalName then
    if dm.Position [OriginalIdx].x = OriginalPos.x then
      if dm.Position [OriginalIdx].y = OriginalPos.y then begin
        dm.Caption [OriginalIdx] := CurName.Text;
        RefreshBtn.Click;
      end;
  finally
    dm.Active := False;
  end;
end;
```

```
Names := TStringList.Create;
try
  // Get *current* desktop status for index mapping
  for Idx := 0 to dm.ItemCount - 1 do Names.Add (dm.Caption [Idx]);
  // Now do the biz
  for Idx := 0 to FileCount - 1 do begin
    // Read item name - pascal format
    fs.Read (Len, sizeof (Len));
    SetLength (ItemName, Len);
    fs.Read (ItemName [1], Len);
    fs.Read (ItemPos, sizeof (ItemPos));
    // Does this item still exist?
    NewIdx := Names.IndexOf (ItemName);
    if NewIdx <> -1 then
      if (dm.Position [NewIdx].x <> ItemPos.x) or
        (dm.Position [NewIdx].y <> ItemPos.y) then
        dm.Position [NewIdx] := ItemPos;
  end;
finally
  Names.Free;
end;
```

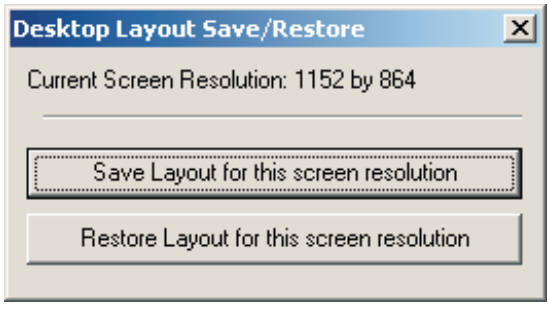
➤ *Listing 9*

Files Roundup section). The operation of this program is pretty straightforward: when you click the `Save Layout` button it saves the position of all current desktop items into a file in the same directory as the executable. If you wanted to, you could embellish things with `File Open/Save` dialogs, set up a file association for the desk layout file, etc, etc, but I wanted to keep things as simple as possible. When you click the `Restore Layout` button, the desktop layout is restored to whatever configuration it had on the last save.

As I mentioned two months ago, when I started this foray into desktop listview access, one of the biggest problems with the brain-dead Windows desktop is the way in which it 'thoughtfully' rearranges your desktop items every time the Windows screen resolution is saved. This is a particularly irritating problem for technical journalists, because we are frequently asked to provide screenshots at a specific pixel resolution, (eg 800 by 600) but few people would want to habitually use that sort of resolution these days! In order to address this particular problem, I wrote `DeskLayout` such that it would save/load a different file depending on the current screen resolution in effect. In other words, if you use 1152 by 864 pixels, then the 'Save Layout' code in `DeskLayout` will spit out a file called 1152x864.desk and this is the filename that the 'Restore Layout' code will look for when operating at the same resolution. Using this simple technique, the program can easily maintain a preferred desktop layout for each screen resolution that you use.

I won't bore you with a lengthy description of all the source code in the `DeskLayout` program; bearing in mind the previous discussion, it's all pretty straightforward stuff. Instead, we will just focus on one code snippet: see Listing 9. This shows the heart of the 'Restore Layout' code. At this point, `fs` is a `TFileStream` object which references the desk layout file being restored. The file starts off with a special signature string that's used to authenticate the file. (You might perhaps wonder why this signature string finishes with a sequence of `CR`, `LF` and `EOF` characters. This is actually an old Borland trick: try displaying the contents of the file using `TYPE` from a DOS command line and you will see what I mean.)

➤ *Figure 3: The desktop layout load/save demo creates differently named files according to the screen resolution being used, making it easy to keep layouts for different resolutions.*



Following the authentication string is a count of the number of items stored in the file. For each item, we store the desktop item name as a Pascal string, followed by a `TPoint` data structure for the item position. When streaming strings into a file, it's generally *much* easier to store them as Pascal strings with a preceding length byte. This allows the 'reader' to read the length byte first, and then know how many more bytes to read or skip, as appropriate. If you want, you can even read from a stream directly into a dynamic string. The code in Listing 9 illustrates how to do this, but be sure to set the string length first!

You will also see that the code reads the current list of item names directly from a `TDesktopManager` component into a stringlist. This is essential, because not only have the item positions possibly changed, but also the ordering of the various items may have changed. Don't assume that the saved item 35 corresponds to the current item 35. By doing a lookup on an up-to-date list of desktop names, we know that we are referencing the correct item. If any item in the saved layout cannot be found in the current desktop configuration, then it is ignored.

I had feared that the desktop layout 'Restore' function might introduce quite a bit of flicker, if a large number of items were being moved around. Thankfully, however, this has proved not to be the case. Obviously the extra time that I spent bludgeoning Microsoft's redrawing code into submission was worthwhile!

One thing which you will notice in Jeff Richter's original code is that he ensures the desktop listview control has the Auto-Arrange feature turned off before doing the desktop restore. This seemed slightly odd to me because, if you have got Auto-Arrange turned on, you are not likely to be much interested in maintaining a specific desktop layout anyway, but then again perhaps I am missing something subtle here!

File/s	Description
DeskManager.dpr DeskManagerDlg.res	Source code for the DLL itself. The all-important .RES file contains the resource template for the hidden dialog window.
DesktopManager.pas	Source for the TDesktopManager component. Compile it and install into a package of your choice.
DeskManMessages.pas	Message definitions used by both the DLL and the component.
DeskTester.dpr DeskTester.exe DeskTesterForm.pas	Source code and executable for the testbed application. This is compiled as a packaged Delphi 5 application.
DeskLayout.dpr DeskLayout.exe DeskLayoutForm.pas	Source code and executable for the desktop layout save/restore demo. This is compiled as a packaged Delphi 5 application.

➤ *Table 1*

Conclusions

There are other facilities you could add to the DeskLayout program, such as the ability to save and restore text foreground and background colours, along with the positioning information. Or you could write the application so as to place an icon in the Windows tray area, for instant access to the program's functionality. Regarding `TDesktopManager` itself, you could add a lot more functionality, such as spacing changes, auto-arrange (if you must!), or maybe add the code needed to retrieve and change the bitmap associated with each desktop item. There is plenty of scope for the adventurous here!

Finally, I should warn you that there is a subtle 'feature' in the way that I've implemented the `PendOnDesktopMessage` routine. Because of the way in which it calls `Application.ProcessMessages` internally, there exists the possibility of triggering some new call on the DLL while the application is waiting to receive a `WM_COPYDATA` response. If you want to see this happen, try removing the outermost `try..finally` block from the `RefreshBtnClick` method in DeskTesterForm.Pas, then rebuild and run the testbed application. If you now try clicking the `Refresh` button twice in rapid succession (ie while it's still processing the first click) then you'll hang the program. I shamelessly cheated by disabling the `Refresh` button to prevent this eventuality, but a cleaner solution might be to implement an 'I'm busy, go away' exception which is triggered if a busy state is detected on entry to critical code.

A final hint: within the `TDesktopManager` implementation, you can easily check if the control is busy by examining the state of the `fGotResponse` field. The way I've written the code, this field has the value -1 whenever we're spinning on `Application.ProcessMessages`.

Files Roundup

Because of the relatively large number of source files this month, Table 1 gives a quick summary of what's what.

Dave is a freelance consultant, programmer and technical journalist specialising in system-level Windows programming, and cross-platform issues. He is the Technical Editor of *The Delphi Magazine*. You can contact Dave at TechEditor@itecuk.com