The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project? Just email Brian Long, our Delphi Clinic Editor, on clinic@blong.com

BDE Administrator Amnesia

Q After installing the 32-bit version of the BDE on a client's Windows 95 machine, the BDE Administrator seems to have a problem. On the `Configuration` tab, under `Configuration, Drivers, Native` it shows nothing at all. The BDE seems to work okay in that Paradox tables can be accessed, but I can't change any of the settings. I am sure it is something simple but I can't figure it out.

A In the BDE Administrator try playing around with the options in `Object | Options`. I can make all of my drivers 'disappear' by unchecking the session, persistent and virtual checkboxes. I think `Persistent` is the one that's important here. These checkboxes map onto a Windows registry setting that can be viewed by running REGEDIT.EXE and navigating through to

```
My Computer\HKEY_LOCAL_MACHINE\
  Software\Borland\
  Database Engine
```

and selecting the `ViewMode` entry. If all three checkboxes are selected, this entry will have a value of `SPV`, meaning that all `Session` aliases, `Persistent` aliases and `Virtual` aliases will be listed in the BDE Administrator.

Bad BDE Installation

Q I have installed Delphi 2, C++Builder 1, Delphi 3 and C++Builder 3 over the past couple of years. During the last 32-bit installation, I got an error at the end regarding a failure to start up the BDE. Now, every application that tries to use the BDE fails with the message *An error occurred while attempting to initialise the Borland Database Engine (error $2109)*. I can't find out how to fix it. What is wrong?

A I have also had this problem, and it is caused by the way the BDE is set up in the registry. The location of all the BDE DLLs is stored in the registry key

```
HKEY_LOCAL_MACHINE\SOFTWARE\
  Borland\Database Engine\
  DLLPath
```

Each successive InstallShield installation adds the path chosen by the user for the BDE (and the SQL drivers if installing the Client/Server version) to the end of the `DLLPath` key. During Borland product installations. I always install the main BDE files in the default c:\Program Files\Common Files\BDE directory, but the SQL drivers get placed in c:\Program Files\Common Files\BDE\SQL. This gives me a `DLLPath` key value made of these two directories, plus a semicolon: 71 characters.

With each new product installation, this same 71 character path is added onto the end of `DLLPath` (along with another semicolon separator), making 143 characters, then 215 characters and then 287 characters. It is at this point that the BDE will fail to initialise, because the supplied search path is bigger in length than the Windows constant `MAX_PATH` (260). Windows objects to search paths longer than `MAX_PATH` and so the BDE DLLs are not located.

So the answer is to tidy up the `DLLPath` key and remove duplication. Really, InstallShield should only add directories if they are not already in the `DLLPath` key, so it is all due to a limitation of the product installer.

Dynamic Fonts

Q Do you know of any way in which I can load fonts at runtime and then assign them to labels?

A You will need to call `AddFontResource('TheFontFileName')` and remember to call `RemoveFontResource` later. These two functions are Windows API routines. To be polite and courteous you will also need to use

```
SendMessage(HWnd_Broadcast,
  wm_FontChange, 0, 0)
```

after calling each of the two font APIs. Assuming you know (by previous examination) what the font names are, you can then assign them to the `Font.Name` property of any appropriate component.

Cached Updates Problem

Q When using cached updates with SQL, Delphi creates files such as Del1.MB and PdoxUsrs.Lck in the same directory as the executable. If two users try to use the same executable at the same time I get an error saying that the directory is controlled by another lock file. This is because, for both users, Delphi tries to create a lock file in the same directory. Is there a way to tell Delphi where to put the lock and .MB files?

A Try setting a value for the `Session.PrivateDir` property that equates to a local path. The BDE will place the cached update file (DEL1.MB) and any other temporary files (such as lock files

```
procedure TForm1.NewItem1Click(Sender: TObject);
begin
  { This loses the maximised MDI child system buttons }
  Menu.Items.Insert(1, NewItem('New 1', 0, False, True, nil,
    0, ''));
end;
procedure TForm1.NewItem2Click(Sender: TObject);
var Flag: Boolean;
begin
  Flag := ActiveMDIChild.WindowState = wsMaximized;
  if Flag then begin
    { Causes some flicker outside the app }
    LockWindowUpdate(Handle);
    ActiveMDIChild.WindowState := wsNormal;
  end;
  try
    Menu.Items.Insert(1, NewItem('New 2', 0, False, True,
      nil, 0, ''))
  finally
    if Flag then begin
      ActiveMDIChild.WindowState := wsMaximized;
      LockWindowUpdate(0)
    end
  end
end;
procedure TForm1.NewItem3Click(Sender: TObject);
{$ifdef Win32}
var Flag, Animation: Boolean;
  function GetAnimation: Boolean;
  var Info: TAnimationInfo;
  begin
    Info.cbSize := SizeOf(TAnimationInfo);
    if SystemParametersInfo(SPI_GETANIMATION, SizeOf(Info),
      @Info, 0) then
      Result := Info.iMinAnimate <> 0
    else
      Result := False;
  end;
  procedure SetAnimation(Value: Boolean);
  var Info: TAnimationInfo;
  begin
    Info.cbSize := SizeOf(TAnimationInfo);
    BOOL(Info.iMinAnimate) := Value;
    SystemParametersInfo(SPI_SETANIMATION, SizeOf(Info),
      @Info, 0);
  end;
{$endif}
begin
{$ifdef Win32}
  Flag := ActiveMDIChild.WindowState = wsMaximized;
  Animation := GetAnimation;
  if Flag then begin
    { Causes some flicker inside the app }
    if Animation then
      SetAnimation(False);
    ActiveMDIChild.WindowState := wsNormal
  end;
  try
    Menu.Items.Insert(1, NewItem('New 3', 0, False, True,
      nil, 0, ''))
  finally
    if Flag then begin
      ActiveMDIChild.WindowState := wsMaximized;
      if Animation then SetAnimation(True)
    end
  end
{$else}
  ShowMessage('This option is Win32 only')
{$endif}
end;
```

➤ *Listing 1*

in the specified directory, thus avoiding the clash between multiple users. This can be done in your main form's `OnCreate` event handler, or with the Object Inspector if a `TSession` component is used.

Erratic MDI Menu

Q I have encountered a problem using Delphi (and also C++Builder) when producing an MDI application. My program occasionally needs to insert a menu item into the main menu. This is fine except when an MDI child window is in a maximised state. Under these circumstances the menu item is successfully added to the main menu, but the minimise, maximise and close buttons for the child window then disappear. Have you encountered this problem before or do you know of a solution?

A I hadn't seen this problem before but it seems to happen in all versions of Delphi. The solution appears to be to minimise the child window before adding the menu item and then maximise it again afterwards. Obviously this minimising and maximising must only take place if the child *is* maximised to begin with.

In Delphi 1, you can avoid the noticeable screen action that this will cause using the `LockWindowUpdate` Windows API. In Delphi 2 and 3 you could alternatively turn the window animation off for the duration. This is the zooming effect you get when minimising and maximising most windows in Windows 95 and Windows NT. This latter option might be desirable as some find the flickering caused by the redraw operations caused by `LockWindowUpdate` to be excessive.

Listing 1 shows the three possibilities including the one that causes the problem. The code can be found in the project MDIMenus.Dpr on the disk. I callously poached the animation enabling and disabling code from the `Forms` unit. The VCL uses it when minimising an application. In Delphi 2 and 3 you might notice that when you minimise the main form, all forms disappear into one icon on the task bar. In fact, all forms get hidden, the task bar icon is a minimised version of the Application window, displayed with no zooming.

Shipping Bitmaps

Q I need to be able to copy a bitmap to the canvas of a form. This bitmap may be one of several that can be copied at runtime. I initially thought of keeping these bitmaps in a resource file bound to the EXE, so that one could be loaded and copied when required. The problem with this approach is that the size of the final EXE would be very large, even if I don't include bitmaps for higher resolutions. My question is can bitmaps (or other resource items) be stored in a DLL that is dynamically linked to the EXE? If so, how?

A The answer to the first question is yes. The answer to the second takes up the rest of this entry. Bitmaps and any other arbitrary resources can be bound to an application executable, one of its DLL code libraries, a resource DLL with practically no code, or to a Delphi package. Before worrying about how to do it with packages in Delphi 3 and above, let's check out normal DLLs.

To load an arbitrary DLL, you call `LoadLibrary`, storing the returned module handle. If this value is less than `HInstance_Error` then it signifies an error. The library must later be unloaded from memory with `FreeLibrary`.

Given a module handle, you can load a bitmap resource from it with `LoadBitmap`, passing the handle as the first parameter. This handle can be the `HInstance` variable in an application, which indicates the .EXE itself (although it is safest to

```
#include "ResConst.Pas"
bmpAthena   BITMAP "c:\delphi\Images\Splash\16Color\Athena.Bmp"
bmpChemical BITMAP "c:\delphi\Images\Splash\256Color\Chemical.Bmp"
```

➤ *Listing 2*

```
unit ResConst;
interface
const
  bmpAthena = 1;
  bmpChemical = 2;
implementation
end.
```

➤ *Listing 3*

use `MainInstance` in Delphi 3 onwards as I'll explain later), or a DLL's module handle. The second parameter for `LoadBitmap` is the bitmap resource identifier, used when manufacturing the resource file. `LoadBitmap` returns a bitmap handle which can be assigned to the `Handle` property of a `TBitmap`.

Information on manufacturing a resource file using a resource script can be found in the *Playing Videos* entry in Issue 32's *Delphi Clinic*. The only extra thing to mention is that the command-line resource compiler is called BRCC.EXE in Delphi 1. An example of a resource script that might be used is shown in Listing 2 (ResBmps.RC), where the referenced ResConst.Pas file is in Listing 3. Obviously you might need to change the paths.

Of course there is no real necessity to use the resource compiler when just setting up bitmap resources. You can use Delphi's image editor or Inprise's Resource Workshop instead. However, using these tools might mean that your resource identifier constants can get out of sync with the actual resources.

Once the resource script is compiled, you should have a binary ResBmps.Res file ready for linking into your resource DLL. The source for the resource DLL project is the barest skeleton to compile, and links the resource file in. Compiling this gives ResLib.Dll:

```
library ResLib;
{$R ResBmps.Res}
Begin
end.
```

A sample project that makes use of this DLL, BmpUser.Dpr, is on this month's disk. The important code from BmpUserU.Pas is in Listing 4. Two buttons are used to load one of the two bitmaps into a `TImage` control. Since I gave the two bitmap resources numeric identifiers, I have to either use `MakeIntResource` or a `PChar` typecast (see the two button `OnClick` handlers in Listing 4) to get a successful compilation. Figure 1 shows the program accessing the bitmaps.

So using a DLL is easy enough. But what about storing resources in a package? Well technically speaking, a package is just a special case of a DLL with a .DPL file extension, but when you compile with Delphi 3 packages, that knowledge is not useful. The contents of all the units that get compiled into a package are available to the application using the package just as if they were being compiled directly into that application. Also, to make a new package, you do not create a DLL project, you ask Delphi to create a package instead.

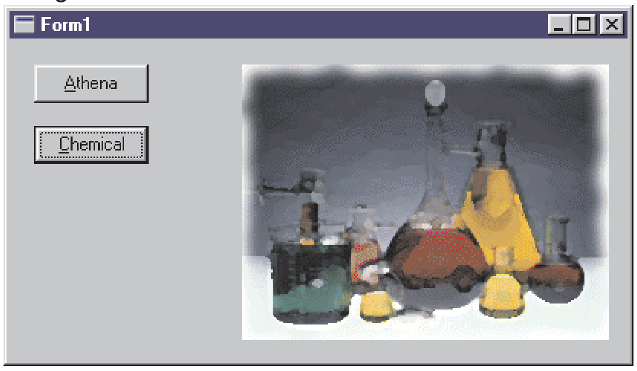
Because of all this, the steps to getting access to resources in packages are a little different. Assuming you already have the compiled resource file and constant declaration unit as described above, we proceed like this.

Make a new package (`File | New... | Package`) and give it a file name, location and optional description. I have supplied one called `ResPkg.Dpk`. Now, to provide somewhere to bind the resources from, make a new unit and put the appropriate compiler directive in:

```
{$R 'ResBmps.RES'}
```

This unit can then be added to the package with the package editor's

➤ *Figure 1*



➤ *Listing 4*

```
type
  TForm1 = class(TForm)
    ...
  public
    DLLHandle: THandle;
  end;
...
uses ResConst;
...
procedure TForm1.FormCreate(Sender: TObject);
begin
  DLLHandle := LoadLibrary('ResLib.Dll');
  if DLLHandle < HInstance_Error then
 {$ifdef Win32}
   RaiseLastWin32Error;
 {$else}
   raise Exception.Create('Could not load resource DLL')
 {$endif}
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  FreeLibrary(DLLHandle)
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  Image1.Picture.Bitmap.Handle :=
    LoadBitmap(DLLHandle, MakeIntResource(bmpAthena))
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
  Image1.Picture.Bitmap.Handle :=
    LoadBitmap(DLLHandle, PChar(bmpChemical))
end;
```

Add button. My unit is called ResPkgU.Pas and when added, makes Delphi's package editor look like Figure 2. Since this package has no use at design-time, press the Options button, deselect design-time package and then select runtime package.

What you have now, if you press the package editor's Compile button, is a .DPL file containing your two bitmap resources. This package is going to be used by an application that we will now create.

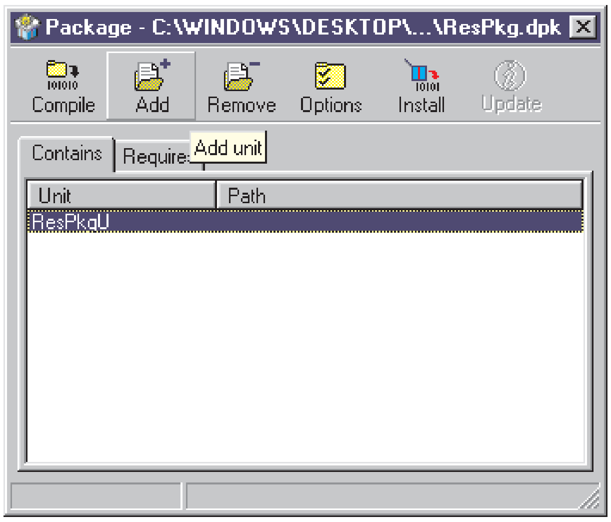
Choose File | New Application, and add a pair of buttons and an image control onto the form (like in Figure 2). Bring up the project options dialog and on the Packages page, ensure the Build with runtime packages checkbox is checked. Now, to make Delphi do the right thing with respect to your package, press the Add... button in the Runtime packages group on the same page and locate the .DCP file generated for your package when it was compiled.

With respect to getting access to the bitmap resources, we will not need to call LoadLibrary on our package. So long as we refer to something in the package, Delphi will ensure it gets loaded while the application is loading. The question is how do we find the module handle for our package?

I mentioned the HInstance variable earlier. This is declared in the System unit, the unit that is implicitly used by all Delphi files. HInstance represents the module handle of the binary file that contains HInstance. In our application code, HInstance represents the .EXE module handle. In code compiled within the package, HInstance represents the .DPL module handle. HInstance is therefore ambiguous, meaning different things depending upon where you access it. This is why, if you want the .EXE module handle, you should use MainInstance which always returns the main application module handle. In the case in point here, we need to access the package module handle from the application. The easiest way to do that is to add a variable into the package unit that gets set to the package's own HInstance variable value. My ResPkgU.Pas file is shown in its full glory in Listing 5.

This now means that as long as we add ResPkgU to a uses clause in the application we are working on, we can refer to ResInstance and it will give us the required module handle. This reference to something in the package also ensures that Delphi will definitely cause the package to be loaded along with the program. Had we not referenced anything, the package would have been ignored and not loaded.

The useful code from the second sample project, BmpUser2.Dpr, is in Listing 6.



➤ *Figure 2*

```
unit ResPkgU;
interface
var
  ResInstance: Integer = 0;
implementation
{$R 'ResBmps.RES'}
initialization
  ResInstance := HInstance
end.
```

➤ *Listing 5*

Graphic File Extensions

Q Delphi 1, 2 and 3 all support a given set of graphic file types: icons (.ICO), bitmaps (.BMP) and metafiles (.WMF). Delphi 2 added support for enhanced metafiles (.EMF) and Delphi 3 has given support for JPEGs (.JPG and .JPEG). Since it is possible for other graphic formats to be installed, is there a way of getting a list of supported graphic file extensions, for the purposes of using it in an open file dialog?

A Up until Delphi 3, the appropriate variable remained unavailable. All the registered file formats are stored in a linked list called FileFormatList. This is a variable hidden in the implementation part of the Graphics unit. So with Delphi 1 and 2, you will have to do the work manually. But with Delphi 3, things have opened up.

The GraphicFileMask function returns the file specification associated with a given graphic class. So GraphicFileMask(TBitmap) returns the string *.bmp and you get the string *.emf;*.wmf when you call GraphicFileMask(TMetafile). However, if you pass in the base graphic class, TGraphic, to GraphicFileMask, it gives you a compound string containing the extensions of all the supported graphic formats. If the JPEG unit has been added to any uses clause in your project, or is contained in a required package in your application, the string will by default be

➤ *Figure 6*

```
uses
  ResConst, ResPkgU;
procedure TForm1.Button1Click(Sender: TObject);
begin
  Image1.Picture.Bitmap.Handle :=
    LoadBitmap(ResInstance, MakeIntResource(bmpAthena))
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
  Image1.Picture.Bitmap.Handle :=
    LoadBitmap(ResInstance, PChar(bmpChemical))
end;
```

`*.jpeg;*.jpg;*.bmp;*.ico;*.emf;*.wmf` otherwise it will be `*.bmp;*.ico;*.emf;*.wmf`.

For the purposes of supporting open dialogs (and open picture dialogs), there is another function called `GraphicFilter`. Again, you pass a specific or generic graphic class to the function but this time it returns a filter string that can be assigned to the `Filter` property of an appropriate dialog. Figure 3 shows such a generated filter being used and Listing 7 shows the straightforward code that makes it work.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  OpenDlg.Filter := GraphicFilter(TGraphic)
end;
```

➤ *Listing 7*

Office Integration

Q I have been looking through *The Delphi Magazine* for any tips on Microsoft Office 97 integration, specifically Microsoft Outlook. I have this code fragment (Listing 8) that works well (when translated) in Visual Basic, but does not in Delphi 2 and 3. It simply makes a call to Outlook to create a new contact, which it does, but the information (name, telephone number, etc) does not get passed over. The contact remains blank.

```
var OutLook: Variant;
...
  OutLook := CreateOleObject('OutLook.Application');
  OutLook.CreateItem(2);
  OutLook.CreateItem(2).LastName := edtLastName.Text;
  OutLook.CreateItem(2).FirstName := edtFirstName.Text;
  OutLook.CreateItem(2).BusinessTelephoneNumber:=edtBusTelNo.Text;
  OutLook.CreateItem(2).Save;
```

➤ *Listing 8*

A Delphi 3 introduced type library support, which is usually of great help when trying to get some Automation code working. Before we look at what this means for this problem, I just need to emphasise a few basic points about Automation support as implemented in Delphi.

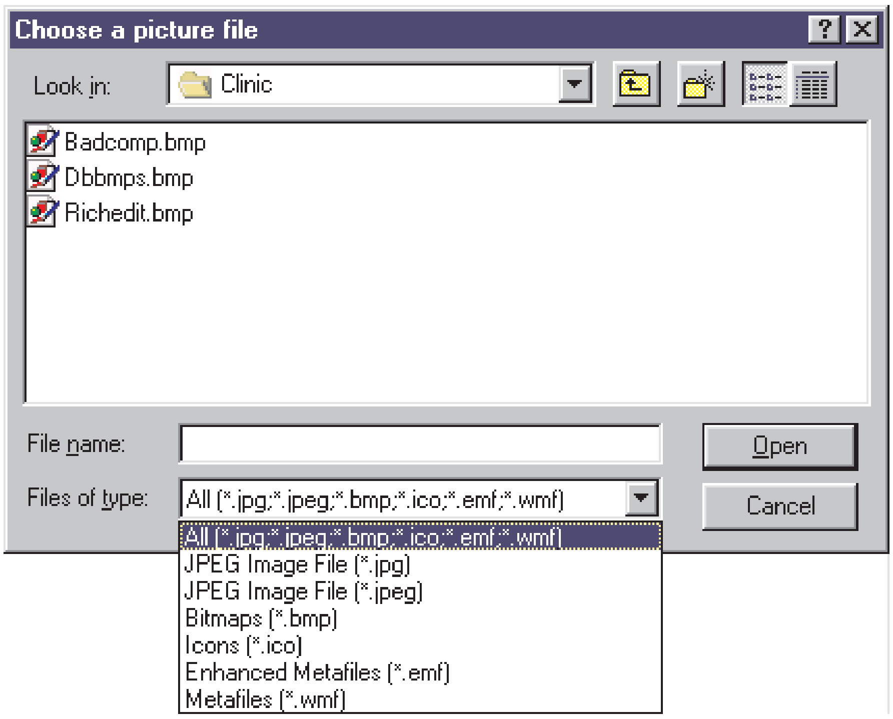
Delphi 2 introduced support for Automation (or OLE Automation as it was called then). The (also new) `Variant` data type was used as the prime controlling force of the Automation mechanism. When you call `CreateOleObject` (from the `OleAuto` unit) and pass it an appropriate `ProgID` (for example `OutLook.Application`) some Delphi code calls some Windows code that attempts to manufacture an instance of the object described by the `ProgID`. If an instance of the appropriate program is already running, that object might well be managed by that program instance. If no instance of the program is running, then the program is launched. Things differ slightly if the Automation object code lives in a DLL but we will ignore complications right now.

When Delphi 2's `CreateOleObject` completes, assuming it was successful, it returns a `Variant` containing appropriate information to allow communication with the object to continue. To talk to the object you do it via the `Variant`. You treat the `Variant` like it is the object in question, accessing properties, calling methods and so on, and some Delphi magic makes it all work out nicely.

Note that in contrast to a Delphi method call or property read/write, the Delphi compiler will not try and verify what you are accessing via the `Variant`. Instead, the compiler will package it up into a data block called a 'call descriptor' and at runtime pass it over to the Automation object in the server. The server will then look it up and see if it is acceptable. If the method is invalid you will get an exception at runtime rather than a compile-time error. This is referred to as 'late binding'.

A `Variant` can hold values of many different types and indeed can hold arrays inside of itself. The data type stored in a `Variant` when talking to an Automation server is an `IDispatch` reference. `IDispatch` is an interface. You might need to refer to the COM series by Dave Jewell if unfamiliar with this form of terminology, but suffice it to say that an Automation server is an object that implements the `IDispatch` interface. The fact that the `Variant` variable holds an `IDispatch` reference means that it is

➤ *Figure 3*



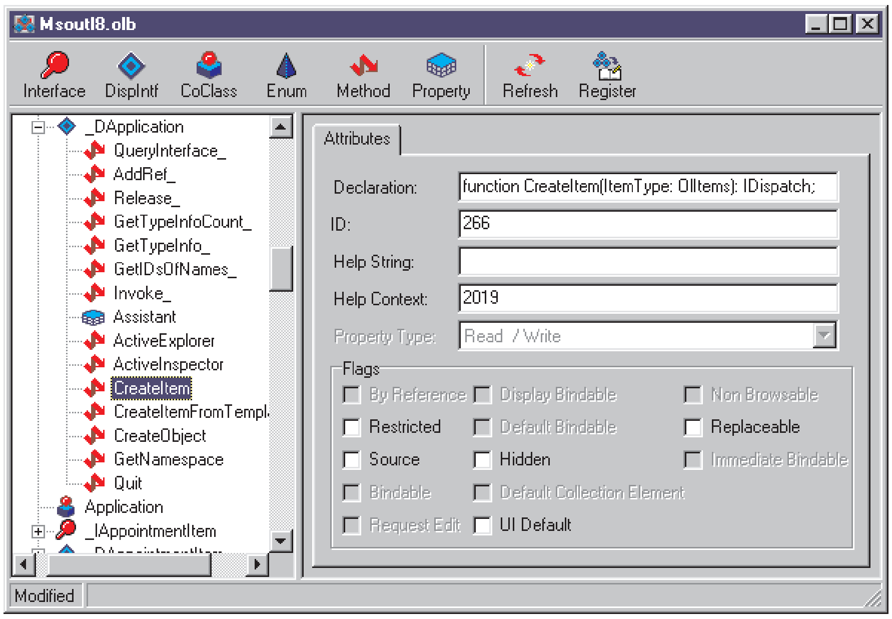
connected to an Automation object. Any object that implements the `IDispatch` interface can be assumed to be an Automation server.

The object in question (as implemented in Microsoft Outlook) will have a number of properties and methods, and indeed you have found a certain amount of information on one of these, `CreateItem`. For me (with no experience of programming OutLook) to examine this problem, I need some information on how to call `CreateItem`. A subroutine declaration would be nice.

This is where the type library support of Delphi 3 comes in. Incidentally, in Delphi 3 Automation (and COM) support was re-implemented. `CreateOleObject` still exists, but now in the `ComObj` unit. Also, instead of returning a `Variant` it now returns an `IDispatch` directly (which you can still assign to a `Variant` if you wish).

COM objects in general (of which Automation objects are a subset) can offer supporting type libraries. Type libraries are binary files which describe the COM objects (and other sundry bits and pieces) available in a language independent manner. Development tools such as Delphi 3 and Borland C++Builder 3 can read these type libraries and generate local language declarations for the interfaces implemented by the COM objects. So Delphi 3 can generate a unit containing Delphi Pascal interface declarations for all the objects exposed by Microsoft Outlook. This can be used to find what parameters are taken and what data types get returned, but can also be used in your program. Indeed, these can be used by the Code Completion and Code Parameters windows in Delphi 3 to ensure you get things pretty much right. When talking to Automation objects, these declarations allow rather more efficient 'early binding' where the objects are called more directly than when using a `Variant`.

A type library of another program can be opened in one of two ways. One way would be simply to



_DApplication
QueryInterface_
AddRef_
Release_
GetTypeInfoCount_
GetTypeInfo_
GetIDsOfNames_
Invoke_
Assistant
ActiveExplorer
ActiveInspector
CreateItem
CreateItemFromTempl
CreateObject
GetNamespace
Quit
Application
_IAppointmentItem

Declaration: function CreateItem(ItemType: OlItems): IDispatch;
ID: 266
Help String:
Help Context: 2019
Property Type: Read / Write

Flags: By Reference, Display Bindable, Non Browsable, Restricted, Default Bindable, Replaceable, Source, Hidden, Immediate Bindable, Bindable, Default Collection Element, Request Edit, UI Default

Modified

➤ *Figure 4*

have a look at what is in it, which can be done using the type library editor. The other is to ask Delphi to generate a new unit containing Pascal versions of everything in the type library it recognises and then to add it to the current project. This latter option would allow you to programmatically refer to things defined in the type library.

To simply browse Outlook's type library, choose `File | Open...` and then drop down the `Files of type:` combobox and choose `Type Library`. Now navigate to where your Microsoft Office package is installed and find the appropriate library. Mine is at C:\Program Files\Microsoft Office\Office\MSOutl8.Olb. When you press `Open`, you are taken to the type library editor (which looks something like Figure 4).

Very much like the form designer and form unit can be switched between with `F12` (as can a package source file and the package editor) this keystroke also toggles between the type library editor and the type library's import unit. So `F12` in the type library viewer does show you a Pascal import unit for Outlook, but it is not part of the project and it is not really a standalone file. Because of the way it was opened, the Pascal code is simply a view on the type library. This is not an advisable approach as Delphi probably won't be able to represent everything in the type library in Pascal terms. Additionally, Delphi will quite happily try and save a modified version of the type library if you inadvertently choose the `File | Save` option (however, Delphi will tell you that you will lose some information first).

If you wish to add a Pascal version of the Outlook type library into your project, we can do the following. Choose `Project | Import Type Library...` and select the Microsoft Outlook entry, mine says `Microsoft Outlook 8.0 Object Library (Version 8.0)`. If the type library is not present in the list (which means it has not been registered by its application) you can press the `Add...` button and go and find it manually. Finally press `OK`. This generates a file Outlook_TLB.Pas with a lot of interface definitions in. This unit is saved (by default) in Delphi's Imports directory and added to the project. In addition, because Outlook's type library refers to two other Microsoft type libraries, you will also get the `Office_TLB` and `MSForms_TLB` units generated.

The object interfaces surfaced by Outlook tend to support Automation. Because of this, for any interface you will find one entry prefixed with `_I` (an interface for COM programming) and another entry prefixed with `_D` (a dispatch interface for Automation programming). Since the code in Listing 11 is trying to create an OutLook.Application object for Automation purposes, we should be looking in Outlook's type library for a _DApplication dispatch interface.

Using either the type library viewer (Figure 4) or the type library import unit, you should be able to locate this item. The code declaration is shown in Listing 9.

The CreateItem function takes a parameter from the OlItems enumeration (whose values again can be found in the type library viewer or import unit, the symbol olContactItem equates to the value of 2 that you are using) and returns an IDispatch. In other words, when you call CreateItem, you get another Automation object returned. The value passed into CreateItem dictates what type of object is returned, but if you pass in a value that asks for a contact, it makes sense that the object returned will be related to contacts very strongly. From browsing what is available, the likely dispatch interface type is _DContactItem, which is borne out by the fact that you refer to properties called FirstName and LastName. _DContactItem has these properties (amongst many others, see Listing 10).

This contact item object should be kept around in a separate variable and talked to in its own right. Your code in Listing 9 is calling CreateItem five times, making five separate contact items. The last one created is the one that you save, and so it has no details set and so remains blank. Not having a copy of Visual Basic I cannot verify how well or badly this code style would work, but given the type library information I would be surprised if it worked exactly as required.

The code in Listing 11 (from Out_Look.Dpr) seems to work a bit more satisfactorily. Notice that it caters for both Delphi 2 and 3.

Before closing this section I feel I should mention some problems I bumped into when looking into this question. Firstly, when the OutLook_TLB.Pas file is added to the project, it causes a compilation error. OutLook_TLB defines a symbol in its interface section called Application. In the project source file, there is ambiguity between Application as defined in the Forms unit and Application as defined in the OutLook_TLB unit. Because units added to a project are added at the bottom of the project file's uses clause, and because the compiler always looks at the last unit in a uses clause first, the wrong Application symbol is used. So I had to modify my project file to allow it to compile. I could have moved the Forms unit to be the last unit in the uses clause, but I took the option of fully qualifying all references to the Application symbol that resides in the Forms unit, see Listing 12.

The second problem is to do with how Microsoft Office 97 acts as a good COM object. Or should I say doesn't act as a good COM object... As Dave Jewell as has explained in his series *Delphi Meets COM* (which started in Issue 28), COM objects are lifetime-managed. Whenever you get a reference to a COM object (for example, an Automation server), Delphi will ensure that its reference count is incremented. Whenever you

```
_DApplication = dispinterface
  ['{00063001-0000-0000-C000-000000000046}']
  property Assistant: Assistant readonly dispid 276;
  function ActiveExplorer: Explorer; dispid 273;
  function ActiveInspector: Inspector; dispid 274;
  function CreateItem(ItemType: OlItems): IDispatch; dispid 266;
  function CreateItemFromTemplate(const TemplatePath: WideString;
    InFolder: OleVariant): IDispatch; dispid 267;
  function CreateObject(const ObjectName: WideString): IDispatch; dispid 277;
  function GetNamespace(const Type_: WideString): NameSpace; dispid 272;
  procedure Quit; dispid 275;
end;
```

➤ *Listing 9*

```
_DContactItem = dispinterface
  ['{00063021-0000-0000-C000-000000000046}']
...
  property BusinessAddress: WideString dispid 32795;
  property BusinessAddressCity: WideString dispid 32838;
  property BusinessAddressCountry: WideString dispid 32841;
  property BusinessAddressPostalCode: WideString dispid 32840;
  property BusinessAddressState: WideString dispid 32839;
  property BusinessAddressStreet: WideString dispid 32837;
  property BusinessFaxNumber: WideString dispid 14884;
  property BusinessTelephoneNumber: WideString dispid 14856;
...
  property FirstName: WideString dispid 14854;
...
  property LastName: WideString dispid 14865;
...
  procedure Save; dispid 61512;
...
end;
```

➤ *Listing 10*

```
{$ifndef Ver90}
uses
  ComObj, Outlook_TLB;
{$else}
uses
  OleAuto;
const
  olContactItem = 2;
{$endif}
{$R *.DFM}
procedure TForm1.Button1Click(Sender: TObject);
var
  OutLook, Contact: Variant;
begin
  OutLook := CreateOleObject('OutLook.Application');
  Contact := OutLook.CreateItem(olContactItem);
  Contact.LastName := edtLastName.Text;
  Contact.FirstName := edtFirstName.Text;
  Contact.BusinessTelephoneNumber := edtBusTelNo.Text;
  Contact.Save;
  Contact.Display(True);
  //  OutLook.Quit
end;
```

➤ *Listing 11*

drop your connection to the server, Delphi will decrement the reference count. If the reference count gets back down to zero, the COM object should dispose of itself.

What this tends to mean is that if you start a conversation with Word (for example), and Word was not already running, then when you finish the conversation, Word should shut itself. This worked fine with Word 6 and Word 95. However Word 97, and all the other Office 97 applications, seem to have given up this convention. The launched application remains steadfastly running when a Delphi application finishes talking to it. I have asked several people who I thought might be able to explain this, but so far no satisfactory explanation has been forthcoming.

So, to summarise, your Outlook application will still be running when the Automation code has finished. One way to approach this problem is to tell Outlook to explicitly close itself (see the commented out call to Outlook's `Quit` method in Listing 12), but this isn't really the COM way. If anyone knows how to get Office 97 to behave like a good COM citizen, please let me know.

For more information on using Automation with Microsoft Outlook, you should refer to the Microsoft Developer Network Library CD-ROM, or the equivalent area on Microsoft's website. Doing a search for 'Automation and Outlook' comes up with many topics which include several that discuss using VB to automate Outlook. You should be able to get the general idea of what properties and methods do various jobs by browsing through these.

```
program Out_Look;
uses
  Forms,
  OutLookU in 'OutLookU.pas' {Form1},
  Outlook_TLB in 'Outlook_TLB.pas';
{$R *.RES}
begin
  Forms.Application.Initialize;
  Forms.Application.CreateForm(TForm1, Form1);
  Forms.Application.Run;
end.
```

➤ *Listing 12*

Acknowledgements

Thanks to Steve Axtell from the Inprise European Technical Team for the database information used in this month's column.