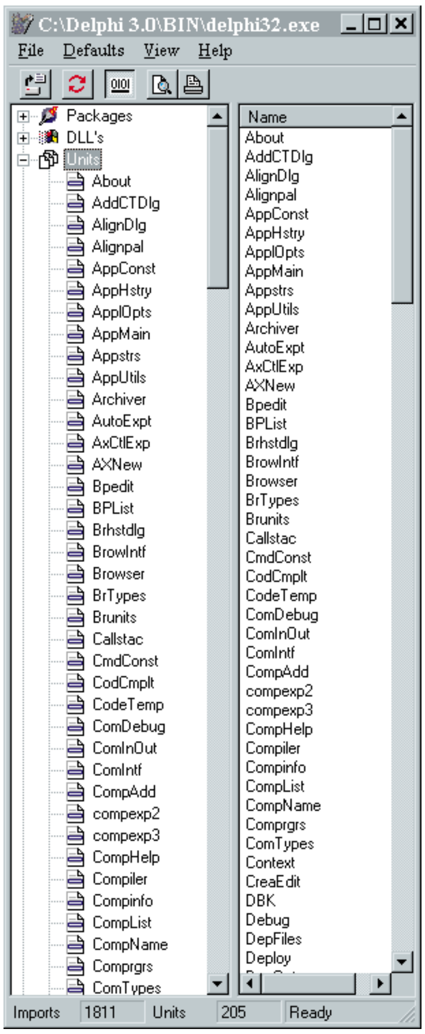
Beating the System: EXE Sniffing, The Story Continues...

by Dave Jewell

You may remember that in last month's column I presented a little utility that could be used to determine the DLLs (and in the case of Delphi 3 executables, the packages) required to run a particular application. At the time, I thought that was pretty much the end of the story. However, not long after I wrote that article I decided to install the latest version of Merlin onto my PC. Merlin, in case you haven't come across it before, is a set of Delphi add-ons which will integrate into 32-bit versions of the Delphi IDE *[See also Brian Long's article on writing Delphi 3 add-ins in this issue. Editor]*. Merlin gives you access to a number of new productivity aids and file viewers from within the IDE and even provides a means of accessing these add-ons (the proper name is *Merlin Wizards*) directly from the Windows shell. This means that, for example, you can browse the resources inside an EXE file simply by right-clicking it from the Explorer and then selecting `Resource Explorer` from the popup menu. If you haven't tried Merlin, I heartily recommend it. You can download a shareware version from the Merlin site at www.boots.com/merlin.

OK, so what's this got to do with last month's column? Well, as I was browsing through the Merlin Wizards, I was fascinated by the capabilities of the Executable Viewer wizard which is capable of showing not only the packages and DLLs required by a particular program, but even shows a list of units compiled into the executable! For example, take a look at Figure 1, which shows some of the 205 units compiled into the Delphi 3.0 IDE.

It should be obvious that being able to see the units linked into an application is potentially very useful. Suppose you discover that one of your tried and trusted units has a rare but fatal bug in it. You fix the unit, but you can't remember which of your applications uses it. Or maybe you work in an environment where it's absolutely essential to know that every piece of code linked into a product has been purchased legitimately.



```
Packages                Name
DLL's                   About
Units                   AddCTDlg
    About               AlignDlg
    AddCTDlg            Alignpal
    AlignDlg            AppConst
    Alignpal            AppHstry
    AppConst            ApplOpts
    AppHstry            AppMain
    ApplOpts            Appstrs
    AppMain             AppUtils
    Appstrs             Archiver
    AppUtils            AutoExpt
    Archiver            AxCtlExp
    AutoExpt            AXNew
    AxCtlExp            Bpedit
    AXNew               BPList
    Bpedit              Brhstdlg
    BPList              BrowIntf
    Brhstdlg            Browser
    BrowIntf            BrTypes
    Browser             Brunits
    BrTypes             Callstac
    Brunits             CmdConst
    Callstac            CodCmplt
    CmdConst            CodeTemp
    CodCmplt            ComDebug
    CodeTemp            ComInOut
    ComDebug            ComIntf
    ComInOut            CompAdd
    ComIntf             compexp2
    CompAdd             compexp3
    compexp2            CompHelp
    compexp3            Compiler
    CompHelp            Compinfo
    Compiler            CompList
    Compinfo            CompName
    CompList            Comprgrs
    CompName            ComTypes
    Comprgrs            Context
    ComTypes            CreaEdit
                        DBK
                        Debug
                        DepFiles
                        Deploy

Imports  1811  Units  205  Ready
```

➤ *Figure 1: The Executable Viewer Wizard showing some of the 205 units compiled into the Delphi 3 IDE*

If you were feeling adventurous, you could take the file-searching code that I discussed last month, merge it with the code I'm presenting here, and thus come up with a utility that searches all Delphi executables on your hard disk to see if they were linked with a particular unit.

In the end, curiosity got the better of me and I did a little 'investigative journalism' in an attempt to figure out how the Merlin code worked its magic. The answer turned out to be relatively straightforward. If you have access to the Delphi 3.0 VCL source code, go into the \SOURCE\RTL\SYS\ directory and open SYSUTILS.PAS. In there (line 5706, or thereabouts) you'll see the declaration for a set of data structures that define a 'package info' resource. Whenever you compile an application or package that requires other packages, this special resource gets generated, and it contains a lot of interesting information.

Introducing The PACKAGEINFO Resource

What's the idea behind this special package info resource? Essentially, it forms a fundamental part of the run-time support for implementing packages in Delphi 3.0. When you run an application that uses packages, the run-time library examines this resource in order to determine which units are contained within the program, and what needs to be linked in from other packages. The package info resource also exists in packages themselves (DPL files) and lists what units are contained within a specific package, as well as what *other* packages are required by this package. The package info resource corresponds to a resource named `PACKAGEINFO` of type `RT_RCDATA` and it's generated by the compiler when you compile an application or package.

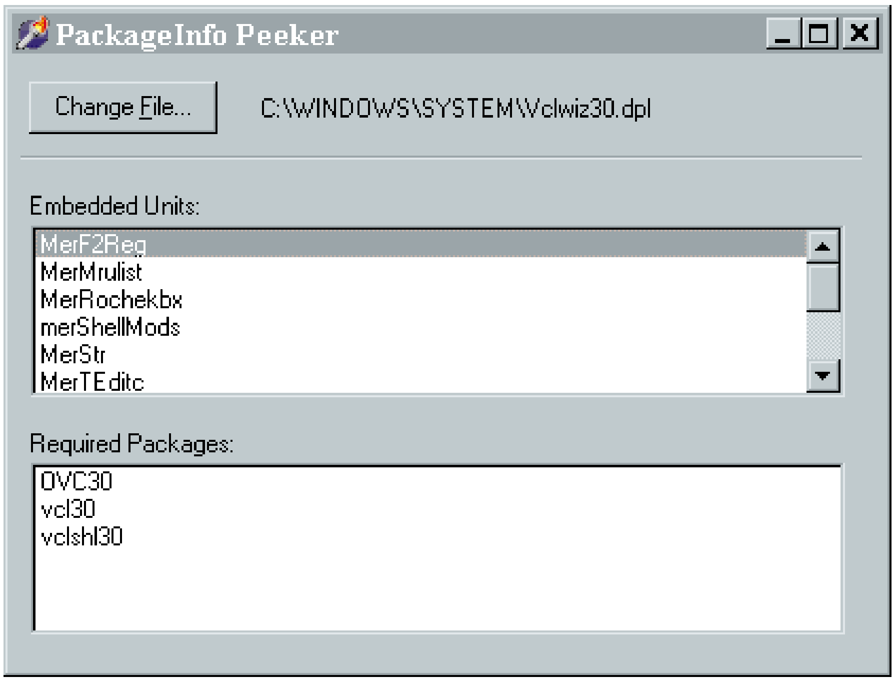
Listing 1 is my first attempt at writing a program to examine the contents of a given package resource. Unlike last month's

```
unit PackPeek;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls;
type
  TForm1 = class(TForm)
    OpenDialog: TOpenDialog;
    CurrentFile: TLabel;
    Button1: TButton;
    Bevel1: TBevel;
    UnitList: TListBox;
    Label1: TLabel;
    PackageList: TListBox;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
  private
  public
  end;
var Form1: TForm1;
implementation
 {$R *.DFM}
procedure PackageCallback (const ModuleName: string;
  NameType: TNameType; Flags: Byte; Param: TForm1);
begin
  with Param do begin
    if NameType = ntContainsUnit then
      UnitList.Items.Add (ModuleName)
    else
      PackageList.Items.Add (ModuleName);
  end;
end;
procedure TForm1.Button1Click (Sender: TObject);
var
  hLib: hModule;
  PackageFlags: Integer;
begin
  if OpenDialog.Execute then begin
    UnitList.Clear;
    PackageList.Clear;
    CurrentFile.Caption := FormatPathToFit(
      OpenDialog.FileName, Canvas, CurrentFile.Width);
    hLib := LoadLibrary (PChar (OpenDialog.FileName));
    if hLib <> 0 then try
      { If we get here, it's a 32-bit executable }
      try
        GetPackageInfo (hLib, Self, PackageFlags,
          @PackageCallback);
      except
        { If executable has no PackageInfo resource,
          just bow out }
          Exit;
      end;
    finally
      FreeLibrary (hLib);
    end;
  end;
end;
end.
```

➤ *Listing 1*

effort, this code is 32-bit only. That's because it uses a package-specific routine that wasn't implemented in previous versions of Delphi, more on that in a moment. Also, bear in mind that, for reasons of space, this isn't a complete code listing. I've removed the `FormatPathToFit` routine because this code is almost exactly the same as it was last month, repeating it would be a waste of space. There's one minor change though: because we're now in 32-bit land, we can't directly assign to byte zero of a string, so you'll need to find the offending piece of code and replace it with this:

```
if Idx > 0 then
  SetLength(Name, Idx-1);
```



C:\WINDOWS\SYSTEM\Vclwiz30.dpl

Embedded Units:
MerF2Reg
MerMrulist
MerRochekbx
merShellMods
MerStr
MerTEditc

Required Packages:
OVC30
vcl30
vclshl30

➤ *Figure 2: My first attempt at a PACKAGEINFO resource explorer*

Don't worry if you haven't got last month's issue: the complete code is included on this month's disk. You can see the program running in Figure 2. It's not as pretty as the Merlin EXE file viewer, but it makes the point that all the information on linked-in units is there for the taking. Assuming, of course, that the executable was compiled with Delphi 3!

So how does the code work? The `Button1Click` routine is the meat of this program. Unlike last time, I haven't gone down the route of encapsulating everything into a callable class library, because you'll undoubtedly have your own ideas about how you want to integrate your code with last month's offering. Once we've got a filename from the `TOpenDialog` component, the fun starts when we call the `LoadLibrary` routine to get a module handle to the executable. Despite the name of this API call, it will actually give us a module handle for an executable file as well as for a DLL.

I believe this wasn't the case with 16-bit Windows, but now you can pass the name of any valid 32-bit executable (EXE or DLL) to the `LoadLibrary` routine.

Because this is an API call rather than a Delphi call, no exception will be generated if the call fails. Instead, the routine will quietly return a module handle of zero. What this most likely means is that the specified file couldn't be found, or it wasn't a Win32 executable. If that happens, then the routine quietly exits. The real key to the program's functionality is in the call to `GetPackageInfo`, a new routine in the Delphi 3 `SYSUTILS` unit. This routine takes the supplied module handle and searches the module for the aforementioned `PACKAGEINFO` resource, calling the application-supplied enumeration routine (in this case called PackageCallback) for every entity contained in the resource. Since GetPackageInfo is a Delphi routine, it *does* generate an exception if it fails, which it will do if the executable in question doesn't contain a PACKAGEINFO resource. Possible exceptions are trapped by a try-except clause whereupon the routine simply exits leaving the two form listboxes empty.

These two listboxes get filled by the PackageCallback routine. Because we've passed Self as the second parameter to GetPackageInfo, we can recover a pointer to the form as the final parameter to the enumeration routine. Using this, we just use NameType to determine whether we're being passed a package name or a unit name, and add the passed string to the appropriate listbox. See the Delphi 3 documentation on GetPackageInfo for more details.

Killing Two Birds With One Stone

If you look at the PackageCallback routine, you'll see that it takes a Byte parameter called Flags. In the 'Mark One' version of this software, we haven't taken any notice of this parameter, but it actually contains some useful information. When NameType indicates that a required package is being enumerated, then the Flags parameter is zero, but if a unit is being enumerated, then the Flags parameter contains interesting information such as whether this is the main unit, and whether the $WEAKPACKAGEUNIT flag was used when compiling this particular code.

Although it would be very easy to modify the existing PackageCallback routine to make use of the Flags byte, I'm going to take a different approach. Rather than continuing to call the GetPackageInfo routine, we'll dispense with it and take responsibility for directly loading the PACKAGEINFO resource, and parsing it ourselves. The reason I'm doing things this way is because I want to show you how easy it is to access 32-bit resources contained in another file. Several

➤ *Listing 2*

```
unit PackPeek;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls;
type
  TForm1 = class(TForm)
    OpenDialog: TOpenDialog;
    CurrentFile: TLabel;
    Button1: TButton;
    Panel1: TPanel;
    MainUnit: TLabel;
    PackageUnit: TLabel;
    WeakPackageUnit: TLabel;
    ImplicitImport: TLabel;
    Label1: TLabel;
    UnitList: TListBox;
    Panel2: TPanel;
    PackageList: TListBox;
    Label2: TLabel;
    Panel3: TPanel;
    NeverBuild: TLabel;
    DesignTime: TLabel;
    RunTime: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure UnitListClick(Sender: TObject);
  private
  public
  end;
var Form1: TForm1;
implementation
 {$R *.DFM}
function BoolCaption (Flags, Mask: Byte; const RootCaption:
ShortString): ShortString;
begin
  Result := RootCaption + ': ';
  if (Flags and Mask) <> 0 then
    Result := Result + 'Yes'
  else
    Result := Result + 'No';
end;
procedure TForm1.Button1Click (Sender: TObject);
var
  hLib: hModule;
  rs: TResourceStream;
  UnitFlags: Byte;
  Idx, PackageFlags, ContainsCount, RequiresCount: Integer;
  function rsReadByte: Byte;
  begin
    rs.Read (Result, sizeof (Result));
  end;
  function rsReadInteger: Integer;
  begin
    rs.Read (Result, sizeof (Result));
  end;
  function rsReadString: ShortString;
  var Ch: Char;
  begin
    Result := '';
    repeat
      Ch := Char (rsReadByte);
      if Ch <> #0 then
        Result := Result + Ch;
    until Ch = #0;
  end;
begin
  if OpenDialog.Execute then begin
    UnitList.Clear;
    PackageList.Clear;
    CurrentFile.Caption := FormatPathToFit(
      OpenDialog.FileName, Canvas, CurrentFile.Width);
    hLib := LoadLibrary(PChar(OpenDialog.FileName));
    if hLib <> 0 then try
      { If we get here, it's a 32-bit executable }
      try
        rs := TResourceStream.Create (hLib, 'PACKAGEINFO',
          rt_rCData);
      except
        { If executable has no PackageInfo resource,
          just bow out }
        Exit;
      end;
      { Ok, we've got resource stream, now interpret data }
      PackageFlags := rsReadInteger;
      NeverBuild.Caption :=
        BoolCaption (PackageFlags, 1, 'Never-Build');
      DesignTime.Caption :=
        BoolCaption (PackageFlags, 2, 'Design-Time');
      RunTime.Caption :=
        BoolCaption (PackageFlags, 4, 'Run-Time');
      RequiresCount := rsReadInteger;
      if RequiresCount <> 0 then
        for Idx := 0 to RequiresCount - 1 do begin
          rsReadByte;
          PackageList.Items.Add (rsReadString);
        end;
      ContainsCount := rsReadInteger;
      if ContainsCount <> 0 then begin
        for Idx := 0 to ContainsCount - 1 do begin
          UnitFlags := rsReadByte;
          rsReadByte;
          UnitList.Items.AddObject(rsReadString,
            TObject(UnitFlags));
        end;
        UnitList.ItemIndex := 0;
        UnitListClick (Self);
      end;
      rs.Free;
    finally
      FreeLibrary (hLib);
    end;
  end;
end;
procedure TForm1.UnitListClick(Sender: TObject);
var Flags: Byte;
begin
  if UnitList.ItemIndex <> -1 then begin
    Flags :=
      Byte(UnitList.Items.Objects[UnitList.ItemIndex]);
    MainUnit.Caption := BoolCaption(Flags, 1, 'Main Unit');
    PackageUnit.Caption := BoolCaption(Flags, 2,
      'Package unit (DPK source)');
    WeakpackageUnit.Caption :=
      BoolCaption (Flags, 4, '$WEAKPACKAGE directive');
    ImplicitImport.Caption :=
      BoolCaption (Flags, 16, 'Implicitly Imported');
  end;
end;
```

months ago, I showed you how to do this with 16-bit resources, but I never got around to showing how it works with 32-bit resources. It turns out that (thanks to some extensions to the Win32 API, and some nice VCL functionality) it's very much easier to "get at" 32-bit resources than it is to work with 16-bit ones. By writing the Mark Two version of our package info browser in this way, I can kill two birds with one stone.

The revised code is shown in Listing 2. As before, I've removed the `FormatPathToFit` routine for the sake of brevity. This time round, the `Button1Click` routine is quite a bit longer because of the additional work involved. As before, we get a new filename from the user, clear any needed list boxes and then call `LoadLibrary` to get a module handle for the specified file.

That's where the similarity ends. Having got a module handle, we can make use of a special VCL class, `TResourceStream`, to open a stream associated with a particular resource in the designated module. We do this by specifying the module handle, the name of the resource and the resource type. You may remember from our original discussion of resources that a resource name or resource type can be specified either as a string, or as a numeric value. If you want to use a numeric value, then you must cast it to a `PChar` before passing to `TResourceStream.Create`. In fact, that's essentially what the pre-defined `rt_RCData` value does. Bear in mind that this function will throw an exception if no resource of the specified name and type can be found, so we need to add some code to handle the exception, in this case simply by "eating" the exception and bowing out gracefully.

At this point, we've got a stream. Because `TResourceStream` is derived from `TStream`, you can do all the usual stream-based stuff with it, except that, of course, you're dealing with some other application's resource data instead of just an ordinary file. Before you ask: yes, it would be great if you could use this nice stream-based interface to modify an existing resource but, no, you can't! As a matter of fact, there is a `TResourceStream.Write` method, but if you call it, you'll just trigger an `EStreamError` exception. At the Windows API level, Microsoft have added a number of routines which enable an application to add, delete or modify resources in another executable, but, at the present time, Borland haven't added the necessary code to the `TResourceStream` class. Maybe this would be a nice project for a future edition of *Beating The System*, but no promises!

Hint: If you want to do the job yourself, take a look at the Win32 API information on `UpdateResource`, `BeginUpdateResource` and `EndUpdateResource`. The simplest approach would be to cache any data written to the `TResourceStream`, marking the stream as 'dirty.' At the time the stream is closed, the code could do the necessary updating of the file in question.

Before we can make sense of the resource data, we have to know the format of the resource. In the case of `PACKAGEINFO`, this is quite straightforward, because Borland have thoughtfully provided all the necessary information in the source code for the Delphi 3 `SYSUTILS` unit. The resource starts off with 32 bits of flag information that describe certain characteristics of the package, or application. These flags indicate whether the package uses explicit rebuild, whether the package can be used at design time, and whether or not it's a run-time only package. The complete set of package flags (taken from the `SYSUTILS` unit) is given below:

```
{ Package Info flags }
pfNeverBuild      = $00000001;
pfDesignOnly      = $00000002;
pfRunOnly         = $00000004;
pfModuleTypeMask  = $C0000000;
pfExeModule       = $00000000;
pfPackageModule   = $40000000;
pfLibraryModule   = $80000000;
```

This 32-bit flag information is immediately followed by a 32-bit count of the number of packages required by this package. As far as I can tell, if the `PACKAGEINFO` resource is inside an application (EXE file), then this count is always zero. That's because the import information in the PE file header (the stuff we looked at last month) provides all the needed information on what packages are required by the application at run-time. However, if the `PACKAGEINFO` resource is inside a package (DPL file), then the count will indicate what additional packages are required by *this* package.

Immediately following the required package count is an array of data structures, one for each of the required packages. This data structure is very simple and comprises a single-byte hash code, followed by the C-style (zero-terminated) name of a particular package. The hash code is used by the Delphi run-time library to distinguish between different packages without having to perform name comparisons.

Immediately following these data structures is another 32-bit count, this time indicating the number of units which are contained within the application or package. As before, this is followed by an array of data structures, one for each contained unit. This time, the data structure comprises a flags byte, another hash code (with a similar meaning to the per-package hash code mentioned earlier) and a C-style name for each unit.

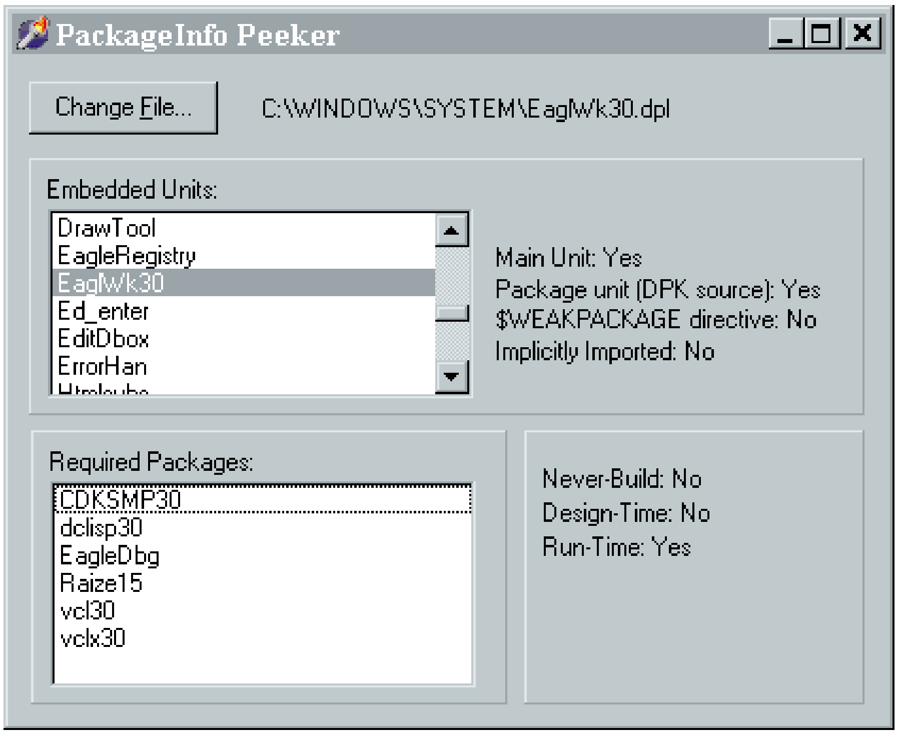
The flags byte may contain a combination of the bit flags given below. Again, these are taken from the `SYSUNITS` unit:

```
{ Unit info flags }
ufMainUnit     = $01;
ufPackageUnit  = $02;
ufWeakUnit     = $04;
ufOrgWeakUnit  = $08;
ufImplicitUnit = $10;
```

Armed with all the above information, we can now write code to parse the `PACKAGEINFO` resource for ourselves. You can see that I've written three small nested functions, `rsReadByte`, `rsReadInteger` and `rsReadString`, which read information from the previously opened resource stream. Using these functions, the Button1Click routine reads the package resource's flags information and then calls a small helper routine, BoolCaption to set up the captions of three TLabel controls. Next, it reads the list of required packages and adds the name of each package to the PackageList list box. Note that this code discards the hash code because it's not really of any use to us at this point. Similarly, the code then reads the list of contained units, adding the name of each unit to the UnitList list box. For each unit name that's added, the corresponding flag byte is added at the same time by coercing it to a TObject.

The UnitListClick routine is called in response to a click on the UnitList list box. Each time the selected unit changes, the flag byte for the selected unit is retrieved from the Objects list and then the BoolCaption routine is called to update the various 'per-unit' TLabel items. See Figure 3 for the overall result.

Figure 3: Here's the all-singin', all-dancin', Mark 2 version. I'm looking at one of the run-time only DPL files installed in my Windows\System directory. You can see that it, in turn, depends on Raize Components.


PackageInfo Peeker

Change File...

C:\WINDOWS\SYSTEM\EaglWk30.dpl

Embedded Units:

DrawTool
EagleRegistry
EaglWk30
Ed_enter
EditDbox
ErrorHan

Main Unit: Yes
Package unit (DPK source): Yes
$WEAKPACKAGE directive: No
Implicitly Imported: No

Required Packages:

CDKSMP30
dclisp30
EagleDbg
Raize15
vcl30
vclx30

Never-Build: No
Design-Time: No
Run-Time: Yes

Resource Snooping For Profit And Pleasure

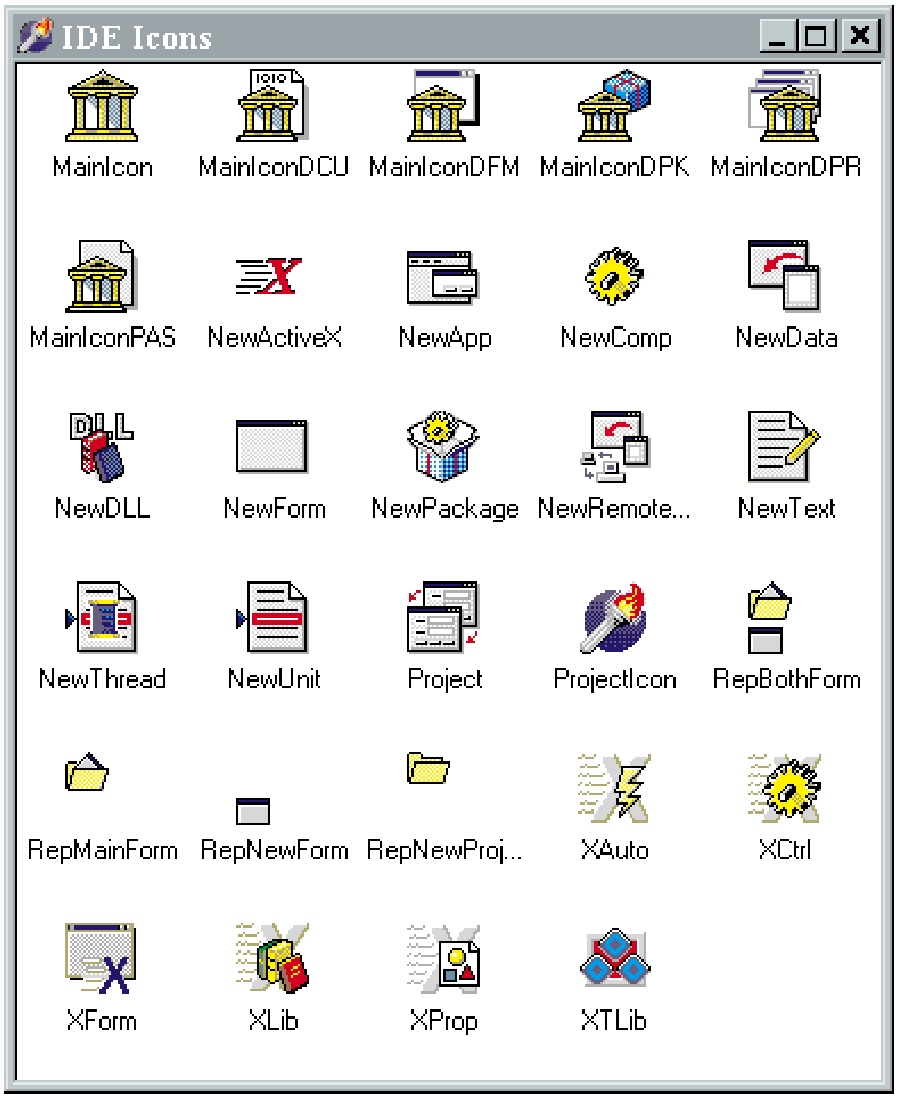
I wouldn't want to leave this subject without emphasising that many of the Windows API standard resource-handling routines will work just as well on someone else's executable as they will when accessing resources within your own application: the key is to retrieve a module handle for the file of interest. To illustrate the point, take a look at the code shown in Listing 3. This is a small program that opens the Delphi 3.0 IDE executable, reads all the icons from the file and loads them into a TImageList control. The various icons are then displayed in a TListView, the result being as shown in Figure 4. This is a very simple program and it assumes that Delphi is installed into the directory shown in the code listing. If it isn't, then the results won't be terribly impressive!

Simple as it is, it demonstrates how easy it is to access resources outside your own code. Instead of loading the icons into an image list, it would have been just as simple to call the SaveToFile method on the TIcon object, thus making a permanent copy of each retrieved icon.

Listing 3

```
unit uicon;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, ComCtrls, StdCtrls;
type
  TForm1 = class(TForm)
    ListView1: TListView;
    ImageList1: TImageList;
    procedure FormCreate(Sender: TObject);
  private
  public
  end;
var
  Form1: TForm1;
implementation
 {$R *.DFM}
const
  IconNames: array [0..28] of PChar = (
    'MainIcon',      'MainIconDCU',    'MainIconDFM',
    'MainIconDPK',   'MainIconDPR',    'MainIconPAS',
    'NewActiveX',    'NewApp',         'NewComp',
    'NewData',       'NewDLL',         'NewForm',
    'NewPackage',    'NewRemoteData',  'NewText',
    'NewThread',     'NewUnit',        'Project',
    'ProjectIcon',   'RepBothForm',    'RepMainForm',
    'RepNewForm',    'RepNewProject',  'XAuto',
    'XCtrl',         'XForm',          'XLib',
    'XProp',         'XTLib' );
```

```
procedure TForm1.FormCreate(Sender: TObject);
var Idx: integer;
    hLib: hModule;
procedure AddIcon (IconName: PChar);
var Icon: TIcon;
    Idx: Integer;
    ListItem: TListItem;
begin
  Icon := TIcon.Create;
  try
    Icon.Handle := LoadIcon (hLib, IconName);
    Idx := ImageList1.AddIcon (Icon);
    ListItem := ListView1.Items.Add;
    ListItem.Caption := IconName;
    ListItem.ImageIndex := Idx;
  finally
    Icon.Free;
  end;
end;
begin
  hLib := LoadLibrary('c:\delphi 3.0\bin\delphi32.exe');
  if hLib <> 0 then try
    for Idx := Low(IconNames) to High(IconNames) do
      AddIcon (IconNames [Idx]);
  finally
    FreeLibrary (hLib);
  end;
end;
end.
```



➤ *Figure 4: This small demo shows how easy it is to access the resources inside another file.*

Of course, there's one difficulty here, which I've carefully glossed over! Specifically, how do you tell programmatically what icons (or other resources, for that matter) are contained within a particular file? When I wrote the code for Listing 3, I used a resource editing program to peek inside the Delphi IDE, jot down the names of all the icon resources found there, and then use these names as the basis for my `IconNames` array. It would have been a lot handier if one could just open an arbitrary file, and access whatever resources it might contain. Surprisingly, it's quite easy to do this under Win32. There are two routines you need. Firstly, you use `EnumResourceTypes` to enumerate all the resources contained within a particular file. Secondly, you use `EnumResourceNames` to enumerate all the individual resources of a particular type. I haven't done this for you, but you should find it very easy to make the necessary changes to Listing 3. Do remember, though, that you need to work with `RT_GROUP_ICON` resources and not the lower-level `RT_ICON` type.

All three of the sample programs developed in this month's code are included on the disk, but I've compiled the executables to use packages so as to minimise space.

Dave Jewell is a freelance consultant/programmer and technical journalist specialising in system-level Windows and DOS work. He is the author of *Instant Delphi Programming* published by Wrox Press. You can contact Dave as Dave@HexManiac.com.