The Delphi CLINIC

Edited by Brian Long

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

GUID Equality

Q I have two TGuid variables in a COM application and am trying to compare them to see if they are the same. I thought GUIDs were strings, so why won't Delphi let me do a simple comparison?

A **G**lobally **U**nique **ID**entifiers, or GUIDs, are represented in one of two ways, either as a string or as a TGuid record. The TGuid record is defined in the Ole2 unit of all 32-bit versions of Delphi and also in the System unit of Delphi 3 and 4. It looks like this:

```
type
  PGUID = ^TGUID;
  TGUID = record
    D1: Integer;
    D2: Word;
    D3: Word;
    D4: array[0..7] of Byte;
  end;
```

So a GUID is actually a 16 byte, or 128 bit, number used to uniquely identify a COM class, interface or type library. But because of the inherent inconvenience of working with the record structure above, there is a standard alternative textual representation. You can translate between a string and a TGuid record using the GUIDToString and StringToGUID functions from the ComObj unit in Delphi 3 and 4, or ClassIDToString and StringToClassID from Delphi 2's OleAuto unit.

In most development tools, it is down to the programmer to translate between the two, but Delphi does its usual trick of stepping in and doing stuff for you. What this means is that instead of defining some TGuid constant in this cumbersome manner:

```
const
  AGuid: TGuid =
    (D1: $2835826F;
     D2: $7E60;
     D3: $11D0;
     D4: ($9F, $EA, $A4, $2B,
          $0, $C1, $0, $0));
```

you can simplify matters like this:

```
const
  AGuid: TGuid =
  '{2835826F-7E60-11D0-9FEA-A42B00C10000}';
```

Despite what it looks like, this does not set a TGuid constant to a string value. Instead, Delphi calls StringToGUID on your behalf (or maybe the underlying API CLSIDFromString) and sets the constant to the resulting GUID record. So, now that we know that GUIDs are 16 byte numbers, not strings, the correct form of comparison can be ascertained. Since Delphi cannot compare two records for equality, you could use a call to CompareMem, to verify the data in the two records are the same. Alternatively you can use a dedicated API for the job, declared in the ActiveX unit. Listing 1 shows a couple of snippets from that unit. The IsEqualGUID API is made available under three different names: IsEqualGUID, IsEqualIID and IsEqualCLSID.

TStringLists And Files

Q I wanted to use a TStringList and use its LoadFromFile method to read from disk, but it does not read the entire file, stopping about halfway. Even on smaller files (400 lines) it only reads about 250 lines. Is this my problem or a Delphi 4 bug? I have checked that there are no DOS end-of-file markers (ASCII character 27) in the file. The annoying part is that I can read the file manually, using ReadLn, with no problems.

```
function IsEqualGUID(const guid1, guid2: TGUID): Boolean; stdcall;
function IsEqualIID(const iid1, iid2: TIID): Boolean; stdcall;
function IsEqualCLSID(const clsid1, clsid2: TCLSID): Boolean; stdcall;
…
implementation
…
function IsEqualGUID;    external ole32 name 'IsEqualGUID';
function IsEqualIID;     external ole32 name 'IsEqualGUID';
function IsEqualCLSID;   external ole32 name 'IsEqualGUID';
```

➤ *Above: Listing 1, ActiveX unit.*

➤ *Below: Listing 2*

```
procedure TStrings.SetTextStr(const Value: string);
var
  P, Start: PChar;
  S: string;
begin
  BeginUpdate;
  try
    Clear;
    P := Pointer(Value);
    if P <> nil then
      while P^ <> #0 do begin
        Start := P;
        while not (P^ in [#0, #10, #13]) do Inc(P);
        SetString(S, Start, P - Start);
        Add(S);
        if P^ = #13 then Inc(P);
        if P^ = #10 then Inc(P);
      end;
  finally
    EndUpdate;
  end;
end;
```

```
procedure ReadIntoStrings(List: TStrings; const FileName: TFileName);
var
  TF: TextFile;
  S: String;
begin
  List.BeginUpdate;
  try
    AssignFile(TF, FileName);
    Reset(TF);
    try
      while not EOF(TF) do begin
        ReadLn(TF, S);
        List.Add(S)
      end
    finally
      CloseFile(TF)
    end
  finally
    List.EndUpdate
  end
end;
```

➤ *Above: Listing 3*

➤ *Below: Listing 4*

```
module Server
{
  interface ITest
  {
    double Get_DateAndTime();
  };

  interface TestFactory
  {
    ITest CreateInstance(in string InstanceName);
  };
};
```

A Your files probably have character #0 in the last line successfully read. Unfortunately, because of the way the TStringList class works, it stops on character #0 . The relevant VCL code is in Listing 2. Notice the loop that stops when #0 is encountered.

You'll have to do it manually, I reckon. Maybe call the BeginUpdate method of your string list, then do a loop with ReadLns and Adds, followed at the end by an EndUpdate call, as shown in Listing 3. Unfortunately, though, in Delphi 1 this will only read lines up to 255 characters in length.

CORBA Limitation?

Q My understanding (from previous experience) is that when writing CORBA objects, you first define the object using the CORBA Interface Definition Language (IDL). People who wish to use your CORBA objects then take the IDL file and pass it to an appropriate translator which generates basic stub and skeleton classes in the syntax of the development tool you are using. These classes can then be used in applications to represent the CORBA objects that may be elsewhere on the network, possibly written in various other languages. Where is the IDL to Delphi translation tool?

A Bob Swart started getting into the ins and outs of CORBA in Issue 43, and briefly talked about CORBA stub and skeleton classes in Delphi, so you might like to refer to that article for some background information *[And the article in this issue too. Ed]*.

The CORBA support offered by Delphi 4 is incomplete with regard to using CORBA objects written in other languages. In short, it does not come equipped with an IDL2PAS tool. This is in contrast to C++Builder 4, which is supplied with IDL2CPP.EXE. But, more generally, the CORBA support is inherently limited. Inprise supply VisiBroker for C++ (as delivered in C++Builder 4) and VisiBroker for Java. There is no VisiBroker for Delphi yet. Delphi comes with VisiBroker for C++ with a wrapper DLL (ORBPAS.DLL).

When writing CORBA server objects in Delphi, you typically use the type library editor (but you can use the Add To Interface... option available on the Edit menu, or the editor's context menu, when looking at the CORBA object class). The type library editor is more than capable of generating an IDL file to represent what you have developed, using the Export button. By default it will write out a file conforming to Microsoft's variant of the language, but the dropdown list next to the button allows you to manufacture a CORBA IDL file. It also happily generates the stub-and-skeleton unit, containing CORBA stub and skeleton classes.

However, there is no given support for taking an arbitrary IDL file and turning that into stub and skeleton classes. The *Delphi 4 Developer's Guide* has something to say on this in Chapter 27. The section entitled *Using stubs* says the following, which you can also find in the Delphi 4 help by looking up *CORBA, stubs* then choosing *Using stubs* from the list:

'If you are writing a client for a CORBA server that was not written using Delphi, you must write your own descendant of TCorbaStub to provide marshalling support for your client. You must then register this stub class with the global CORBAStubManager. Finally, to instantiate the stub class and get the server interface, you can call the global BindStub procedure to obtain an interface which you then pass to the CORBA stub manager's CreateStub method.'

So the bottom line is this. If you want to have a stub class to use in order to access the server object, it is currently down to you to do this, following the brief guidelines mentioned above. So how does this procedure work? Well, if you try to follow the help directly you wouldn't get too far. Unfortunately, the BindStub procedure requires a parameter with a type of IOrb, and the relevant value is kept tucked away in the private section of a TOrb object. This isn't the end of the world, though, there is a routine called CorbaFactoryCreateStub that calls BindStub and CreateStub for you. Bearing this in mind, let's press on.

For an example, I wrote a simple CORBA server in Delphi 4 (Server.Dpr) and exported the IDL file (Server.Idl in Listing 4). At this point I have a compiled CORBA executable with the appropriate

IDL file, and will pretend that the CORBA server was written in some other language, to match the question.

The prime object in the server implements an interface called ITest with one method Get_DateAndTime which returns a double. In fact, when I defined the interface in Delphi's type library editor, I made a read-only property called DateAndTime, defined to be a TDateTime, but CORBA does not understand properties or TDateTime types.

The way a Delphi-built CORBA server operates is that the main CORBA object does not exist by default. When a client application wants to talk to the object, it does so via a factory object which is always up and running. The factory object implements an interface called TestFactory, with a method called CreateInstance whose job is to create an instance of the object that implements ITest.

So in fact there are two CORBA objects in a Delphi CORBA server. The factory that always exists, and the real object that gets created by the factory object. Sometimes, CORBA servers will dispense with the factory idea and have the main object directly available. You should take this into consideration when trying to talk to objects written in other languages, but note that Delphi always uses factory objects and generates appropriate support code in the stub/skeleton unit (the ProjectName_TLB file).

On the disk are supplied three versions of a client application, showing different ways to access the server object, working from the IDL file and developing some rules along the way.

Firstly, we will try and develop a stub unit that mimics, to a certain extent, the normal stub-and-skeleton unit generated via the type library editor. So, the steps are as follows.

First, make a new unit and save it under a suitable name, (Server_Stub.Pas in my case). Next add CorbaObj and OrbPas into the uses clause of the interface part of the unit.

Now define interface types to represent the main interfaces from the IDL file. If you wish, you can add in any properties that you feel the IDL file is missing into the interface type. Listing 5 defines a corresponding interface called ITest with an extra property.

Define descendants from TCorbaStub that can do the appropriate marshalling of data from the client to the server, and back again. You can get a good idea of how this code should look by mocking up a CORBA object in Delphi that mirrors the one you are trying to access (in its methods and properties) and looking at the stub-and-skeleton unit so generated. The prime difference is that you should use TCorbaStub, whereas Delphi will use TCorbaDispatchStub. Have a look at TTestStub in Listing 5, in particular the implementation of Get_DateAndTime. You will find it is not dissimilar to the stub class that is part of the server project on the disk subdirectory.

The stub classes must be registered with the CorbaStubManager variable in the initialisation section of the unit. The interfaces should be registered with the CorbaInterfaceIDManager variable in the initialisation section of the unit. In truth, this step may not be required for your particular application, but we will include it in the steps.

Now define utility classes to act as local representations of each CORBA class factory. The TTestCorbaFactory class in Listing 5 has a CreateInstance class method that calls the aforementioned CorbaFactoryCreateStub routine to connect to the factory and get an instance of the desired CORBA object. This requires you to know the Repository ID of the factory class (which is 'IDL:Server/TestFactory:1.0' in my case). If you look carefully at the one used in Listing 5, you can see that it is fairly straightforward, if a little lengthy. Alternatively, you can run a copy of the server application and then run the VisiBroker Smart Finder, which will locate all CORBA objects running under the VisiBroker ORB, displaying their Repository IDs.

At this point, you can access your target CORBA server in exactly the same way as you would with any Delphi-written CORBA server. I'll grant you that the server used in this example was originally written in Delphi, but I was pretending that it wasn't ☺. The project Client.Dpr talks to the server using the code shown in Listing 6.

That's one out of the way. Let's try another approach now. This time, the list of steps is a little different. The primary issue is to consider the factory object as a standard CORBA object and to

➤ *Listing 5*

```
unit Server_Stub;
interface
uses
  CorbaObj, OrbPas;
type
  ITest = interface
    ['{1E4B84EE-D627-11D2-96EC-0060978E1359}']
    function Get_DateAndTime: TDateTime;
  end;
  TTestStub = class(TCorbaStub, ITest)
    function Get_DateAndTime: TDateTime;
  end;
  TTestCorbaFactory = class
    class function CreateInstance(const InstanceName:
String): ITest;
  end;
implementation
function TTestStub.Get_DateAndTime: TDateTime;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin;
  FStub.CreateRequest('Get_DateAndTime', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
  Result := InBuf.GetDouble;
end;
class function TTestCorbaFactory.CreateInstance(
  const InstanceName: String): ITest;
begin
  Result := CorbaFactoryCreateStub(
   'IDL:Server/TestFactory:1.0', 'Test', InstanceName, '',
   ITest) as ITest
end;
initialization
  CorbaStubManager.RegisterStub(ITest, TTestStub);
  CorbaInterfaceIDManager.RegisterInterface(ITest,
'IDL:Server/TestFactory:1.0');
end.
```

define an interface and stub class for it.

So, make a new unit and save it as a suitable name (Server_Stub2.Pas in my case). Add `CorbaObj` and `OrbPas` into the uses clause of the interface part of the unit.

Now define interface types to represent all the interfaces from the IDL file, including the factory interfaces. Listing 7 from Client2.Dpr defines interfaces called `ITest` and `ITestFactory`. Define descendants from `TCorbaStub` that can do the appropriate marshalling of data from the client to the server, and back again, one per defined interface. Listing 7 has `TTestStub` and `TFactoryStub`.

The stub classes must be registered with the `CorbaStubManager` variable in the `initialisation` section of the unit.

The defined interfaces should be registered with the `Corba InterfaceIDManager` variable in the `initialisation` section of the unit. Again, not all the interfaces may need registering, but it is easiest to do all of them.

Since Inprise have not supplied an IDL2PAS converter, you can do it yourself as I have described above, which is a little tedious. However, an alternative exists on the internet. An enterprising developer called Kevin Smith has written his own version of IDL2PAS which can be located at:

```
www.sotainter.net/users/krsmes/
  corba/idl2pas.zip
```

```
uses Server_Stub;
…
Server: ITest;
…
Server := TTestCorbaFactory.CreateInstance('');
Label1.Caption := DateTimeToStr(Server.Get_DateAndTime);
```

➤ *Listing 6*

This is a good intermediate solution until Borland supply an official tool. Such a tool may arrive in Delphi 5: various Inprise people have been seen strongly suggesting this fact in online internet chat sessions.

Another part of the *Delphi 4 Developer's Guide* in the section called *Writing CORBA clients*, which is in the help if you look up *CORBA clients*, says:

'You may want to use dynamic binding when writing CORBA clients for servers that are not written in Delphi. This way, you do not need to write your own stub class for marshalling interface calls.'

This is referring to the *Dynamic Invocation Interface*, or DII. This allows you to access a CORBA server object using late binding, and so you find out only at runtime if you spelt the method names correctly. Using interfaces and stub classes allows early bound access to the object and compile-time verification that what you are calling is correct.

This situation very much parallels the two ways of accessing a COM Automation object. You can use a `Variant` for late bound Automation, or use the interfaces in the type library import unit for early bound Automation.

For DII to work, the IDL file must be present in an Interface Repository that is running when the client attempts to access the server object. To lodge an IDL file in an interface repository, use the IREP tool, which can be launched as a console or GUI application. Something like this should do the job:

```
IRep -console MyRepository
  Server.Idl
```

Once the IDL file is in the repository, you can proceed. Again, my example server has a factory object that is always running, and is used to get an instance of the target CORBA object. Since this approach relies on no interface types being available, we represent both the server and factory objects using a variable of type `TAny`. This type is defined in the `CorbaObj` unit as a `Variant`, and operates similarly to a `Variant` in COM Automation.

To access the factory object, you call `CorbaBind`, passing the interface repository along. Assuming this succeeds, you can then call the factory method `CreateInstance` through the `TAny` variable and it will return a new `TAny` variable that you can store in the Server variable. So Listing 6 now turns into Listing 8.

➤ *Listing 7*

```
unit Server_Stub2;
interface
uses CorbaObj, OrbPas;
type
  ITest = interface
    ['{6B0BEBC1-40B4-11D2-8684-0020182CD6A0}']
    function Get_DateAndTime: TDateTime;
  end;
  TTestStub = class(TCorbaStub, ITest)
    function Get_DateAndTime: TDateTime;
  end;
  ITestFactory = interface
    ['{2270CD21-D63E-11D2-96EC-0060978E1359}']
    function CreateInstance(const InstanceName: String):
      ITest;
  end;
  TTestFactoryStub = class(TCorbaStub, ITestFactory)
    function CreateInstance(const InstanceName: String):
      ITest;
  end;
implementation
function TTestStub.Get_DateAndTime: TDateTime;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('Get_DateAndTime', True, OutBuf);
  FStub.Invoke(OutBuf, InBuf);
  Result := InBuf.GetDouble;
end;
function TTestFactoryStub.CreateInstance(const InstanceName:
  String): ITest;
var
  OutBuf: IMarshalOutBuffer;
  InBuf: IMarshalInBuffer;
begin
  FStub.CreateRequest('CreateInstance', True, OutBuf);
  OutBuf.PutText(PChar(InstanceName));
  FStub.Invoke(OutBuf, InBuf);
  Result := UnmarshalObject(InBuf, ITest) as ITest;
end;
initialization
  CorbaStubManager.RegisterStub(ITest, TTestStub);
  CorbaInterfaceIDManager.RegisterInterface(ITest,
    'IDL:Server/Test:1.0');
  CorbaStubManager.RegisterStub(ITestFactory,
    TTestFactoryStub);
  CorbaInterfaceIDManager.RegisterInterface(
    ITestFactory, 'IDL:Server/TestFactory:1.0');
end.
```

CORBA And DUN

Q I sometimes run CORBA client applications on a laptop, when it is plugged into the network. The laptop runs Windows 95 and has a modem, which is accessed through Windows 95 Dial-Up Networking (DUN). The problem I get is that whenever my programs are locating a remote object, the modal dial-up networking connection dialog pops up. My program hangs until I cancel the dialog and clearly this requires user interaction. Can I stop the dialog popping up in the first place?

A I have encountered the same problem myself when I started looking at CORBA last year. In brief, I found the cause of the problem, and came up with what I felt was an adequate solution.

If the Windows AutoDial facility is enabled, the dialog will pop up when your CORBA client does some type of network search (excuse the lack of technicality here, networking is not my forté). My approach to the problem is to have a look at the corresponding registry setting and see if AutoDial is enabled. If it is, I disable it, but just temporarily. Then the CORBA subsystem can be initialised and the setting can be restored if it was changed.

The three CORBA client projects on the disk include a self-contained unit called NoDUNBox.Pas that helps eliminate this problem. Just add it to your CORBA client projects and hopefully the problem should disappear. You can see the important parts of the code in Listing 9.

➤ *Listing 9*

```
uses Registry, CorbaObj;
const
  Path = 'Software\Microsoft\Windows\CurrentVersion\Internet Settings';
  EnableAutoDial = 'EnableAutoDial';
  DisableIt: Longint = 0;
var
  EnableIt: Longint;
  ChangeIt: Boolean;
initialization
  //We're gonna play with the registry
  with TRegistry.Create() do
  try
    //Flag for later on
    ChangeIt := False;
    //Do we have these Internet settings?
    if OpenKey(Path, False) then begin
      //Check the AutoConnect option
      ReadBinaryData(EnableAutoDial, EnableIt, SizeOf(EnableIt));
      //If it is on, we will turn it off
      ChangeIt := EnableIt <> DisableIt;
      if ChangeIt then
        WriteBinaryData(EnableAutoDial, DisableIt, SizeOf(DisableIt));
    end;
    //We call this now to prevent a minor delay later on
    //when CORBA stuff actually happens, but primarily so we
    //can prevent the Dial-Up Networking box if necessary
    CorbaInitialize;
    //If we changed something, put it back to how it was
    if ChangeIt then
      WriteBinaryData(EnableAutoDial, EnableIt, SizeOf(EnableIt))
  finally
    Free
  end
end.
```

Before leaving the discussion of CORBA applications on laptop machines, it should be noted that CORBA needs an active TCP/IP stack to operate correctly. If you remove a laptop's network card, the chances are that your server applications will no longer start, due to the lack of a TCP/IP stack. If you have a modem on the machine, you can set an active stack as follows.

Go to the Network Properties dialog, and modify the TCP/IP settings for your Dial-Up Adapter. The idea is to ensure that the `Specify an IP address:` radio button is selected, and to give yourself an IP address. This will allow the various required network messages to work. Of course you will need to remember to reset the option before dialling out to your ISP, but so far this is the best I can offer.

```
uses CorbaObj;
…
Factory: TAny;
Server: TAny;
…
Factory := CorbaBind('IDL:Server/TestFactory:1.0');
Server := Factory.CreateInstance('');
Label1.Caption := DateTimeToStr(Server.Get_DateAndTime);
```

➤ *Listing 8*

One final comment to make is that TCP/IP is definitely required for VisiBroker applications. Make sure you set it up well. This includes adding entries into the HOSTS file, which is somewhere in your Windows directory tree, that describe the names and IP addresses of the machines that are typically found on your local network, and are frequently accessed by CORBA clients talking to CORBA servers.

Paradox Update

The Issue 42 article *Paradox File Corruption* has now been updated thanks to several readers who pointed out that my application had a few small errors. The article documented various registry entries as `DWord` values, but the program entered them as binary values, which has a tendency to upset Windows NT and cause entries to be added into the system log mentioning that they are of the wrong type. The article has now been updated (with the source) and is at www.itecuk.com/delmag in the *What's New* area.

NT Service Wizard Update

In Issue 43's Clinic, in *Delphi 4 Disservice*, I referred you to a 3rd party NT Service wizard written by Jeff Overcash that can be used in Delphi 4 Professional. An official version of this wizard has now been released and can be found at

```
www.borland.com/
  devsupport/bcppbuilder/
  file_supplements.html
```

which works with both Delphi 4 Professional and C++Builder 4 Professional. Thanks to Wyatt Wong for this information.

Long Filename Update

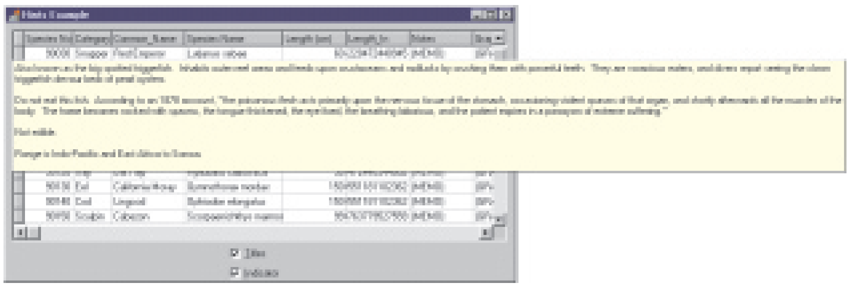
In Issue 44's Clinic, Listing 7 used `SHGetFileInfo` with a `SHGFI_DISPLAYNAME` parameter to translate from a short file name to a long filename. As Ian Carter pointed out to me, the `SHGFI_DISPLAYNAME` flag gets exactly that: a display name. If the user has the *Hide MS-DOS file extensions for file types that are registered* option set in Windows Explorer, the display name returned by `SHGetFileInfo` will not contain an extension for various file types. This will cause problems if the resultant pathname is passed to a routine like `ShellExecute`, which will not be able to find the file.

Acknowledgments

Thanks are due to Inprise's Steve Axtell for some TCP/IP tips mentioned this month and also to Bob Swart for some CORBA-related comments.

Update: Tooltips Under Your Control

After publishing the *Tooltips Under Your Control* article in Issue 43, a reader sent me a mail asking if it was possible to have a tooltip be displayed when the mouse moves over a memo field. Of course in a `DBGrid`, memos normally display as (MEMO) but the request was for the tooltip to display the contents of the memo field. This seems to be a very simple change in Delphi 4: if the field is a memo, extract the `AsString` property of the field instead of the `DisplayText` property. Unfortunately with Delphi 1, 2 and 3, they all seem to have different problems with this simple request. For example, in Delphi 1 a `TMemoField`'s `AsString` property extracts (MEMO) instead of the real memo contents, and also Delphi 1 strings can only be 255 characters in length. So, having checked into the problem with each version, I now have a new version of the code that works as requested. Another sample project called NewHints.Dpr is supplied on this month's disk (along with the modified component unit), and it is shown running in the screen shot below.



Hints Example

Species No	Category	Common_Name	Species Name	Length (cm)	Length_In	Notes
90030	Snapper	Red Emperor	Lutjanus sebae	[illegible]	[illegible]	(MEMO)
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	(MEMO)
90130	Eel	California Moray	Gymnothorax mordax	150	[illegible]	(MEMO)
90140	Cod	Lingcod	Ophiodon elongatus	150	[illegible]	(MEMO)
90150	Sculpin	Cabezon	Scorpaenichthys marmoratus	[illegible]	[illegible]	(MEMO)

[illegible]