The Delphi CLINIC

Edited by Brian Long

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

Error Starting Kylix

Q I get runtime error 230 when trying to start Kylix. Do you know what might cause this?

A The most likely cause I have heard is that your JPEG library has been compiled incorrectly. Whilst the library is apparently the correct version (since Kylix installed correctly), one of the values defined in the library is causing problems with the JPEG image used in the Kylix splash screen.

To find out if this is true, try passing the `-ns` parameter to the startkylix script which disables the splash screen and see if the IDE starts up OK. Assuming it does, you can perform a further test by choosing `Help | About` to see the `About` box. This dialog also has a JPEG image and, if the JPEG library is indeed compiled incorrectly, you could well get *JPEG Error #21*.

You will need to obtain the JPEG library source, possibly from your Linux installation CDs, or from www.ijg.org. Look in the file jpeglib.h and locate the symbol `D_MAX_BLOCKS_IN_MCU` (the JPEG decompressor's limit on blocks per MCU). If it is being defined as anything other than 10, change it back to 10 (it is apparently often changed to higher values, such as 64).

Now you can follow the directions that come with the library source code to recompile and install it. This should remedy the problem.

An alternative solution would be to install the correctly compiled library from the Kylix CD. You can find the RPM file in the CD's patches/jpeg6.2.0 directory. Your current library can be updated by navigating to the CD directory and running:

```
rpm -Uhv
  libjpeg-6.2.0-62.i386.rpm
```

Linux Equivalents Of Win32 APIs

Q Can you tell me what the equivalents of the following Win32 APIs are in the Linux environment? I use these APIs quite frequently: `LoadLibrary`, `FreeLibrary`, `GetProcAddress`, `GetComputerName`, `GetUserName`, and `OutputDebugString`.

A Sounds like you could do with ordering *The Tomes of Kylix: The Linux API* by Glenn A Stephens, published by Wordware, which should hopefully be available by the time you read this. The book should cover most of the standard Linux API, showing how to use it in Kylix programs and I am certainly looking forward to reading a copy myself.

Anyway, back to the question. Some of these Win32 APIs are reasonably easy to translate over to the equivalent Linux calls. For example, you can see how to do the library manipulation by scouring the Kylix source code (Kylix applications have to load library files sometimes, after all). In fact, the Kylix version of SysUtils.pas obligingly provides us with implementations of `LoadLibrary`, `FreeLibrary` and `GetProcAddress` which call down to the appropriate Linux APIs (see Listing 1).

`SysUtils` also has a few more translated Win32 APIs up its sleeve, including `GetModuleHandle`, `GetModuleName` and `Sleep` as can be seen in Listing 2.

The other APIs in question include `GetComputerName`, `GetUserName` and `OutputDebug String`. The equivalent of `GetComputerName` is `gethostname` (see Listing 3). `GetUserName` is a bit trickier, though.

On Linux, there is a `getlogin` routine that returns the user logged in on the controlling terminal, but this routine is apparently ill-advised for security-related

➤ *Listing 1: Dynamic library manipulation in Linux.*

```
function LoadLibrary(ModuleName: PChar): HMODULE;
begin
  Result := HMODULE(dlopen(ModuleName, RTLD_LAZY));
end;
function FreeLibrary(Module: HMODULE): LongBool;
begin
  Result := LongBool(dlclose(Pointer(Module)));
end;
function GetProcAddress(Module: HMODULE; Proc: PChar):
  Pointer;
var
  Info: TDLInfo;
  Error: PChar;
  ModHandle: HMODULE;
begin
  // dlsym doesn't clear error state when function succeeds
  dlerror;
  Result := dlsym(Pointer(Module), Proc);
  Error := dlerror;
  if Error <> nil then
    Result := nil
  else if dladdr(Result, Info) <> 0 then begin
{ In glibc 2.1.3 and earlier, dladdr returns a nil dli_fname
    for addresses in the main program file.  In glibc 2.1.91
    and later, dladdr fills in the dli_fname for addresses
    in the main program file, but dlopen will segfault when
    given the main program file name.
    Workaround:  Check the symbol base address against the
    main program file's base address, and only call dlopen
    with a nil filename to get the module name of the main
    program. }
    if Info.dli_fbase = ExeBaseAddress then
      Info.dli_fname := nil;
    ModHandle := HMODULE(dlopen(Info.dli_fname, RTLD_LAZY));
    if ModHandle <> 0 then begin
      dlclose(Pointer(ModHandle));
      if ModHandle <> Module then
        Result := nil;
    end;
  end else Result := nil;
end;
```

```
function GetModuleHandle(ModuleName: PChar): HMODULE;
  function CheckModuleName(linkmap: plink_map): Boolean;
  var
    BaseName: PChar;
  begin
    Result := True;
    if ((ModuleName = nil) and ((linkmap.l_name = nil) or
      (linkmap.l_name[0] = #0))) or
      ((ModuleName[0] = PathDelim) and (StrComp(ModuleName,
      linkmap.l_name) = 0)) then begin
      Result := False;
      Exit;
    end else begin
      // Locate the start of the actual filename
      BaseName := StrRScan(linkmap.l_name, PathDelim);
      if BaseName = nil then
        BaseName := linkmap.l_name
      // The filename is actually located at BaseName+1
      else Inc(BaseName);
      if StrComp(ModuleName, BaseName) = 0 then begin
        Result := False;
        Exit;
      end;
    end;
  end;
begin
  Result := InitModule(ScanLinkMap(@CheckModuleName));
end;
function GetModuleName(Module: HMODULE): string;
var
  ModName: array[0..MAX_PATH] of Char;
begin
  SetString(Result, ModName,
    GetModuleFileName(Module, ModName, SizeOf(ModName)));
end;
procedure Sleep(milliseconds: Cardinal);
begin
  usleep(milliseconds * 1000);  // usleep is in microseconds
end;
```

➤ *Listing 2: Some more Win32 APIs written for Linux, from SysUtils.*

purposes; it is too easy for `getlogin` to be fooled:

```
ShowMessage(getlogin());
```

You could also try getting the value of the `LOGNAME` environment variable, but again, that can be changed by anyone in advance:

```
ShowMessage(
  GetEnvironmentVariable(
  'LOGNAME'));
```

It is more reliable to use `getuid`, which returns the real user id (uid) of the current process. This value can then be fed to `getpwuid`, which returns a pointer to a record containing information on the uid from the file /etc/passwd. The record contains the corresponding user name for the uid:

```
ShowMessage(
  getpwuid(getuid)^.pw_name);
```

To find the uid of the effective user, in other words, the uid that the current program is running under the guise of, you could try `cuserid`. However, the online manual page for it warns us off this call as well: *Nobody knows precisely what* `cuserid()` *does, avoid it in portable programs, avoid it altogether, use* `getpwuid(geteuid())` *instead, if that is what you meant. DO NOT USE* `cuserid()`. The comments suggest calling `geteuid`, which returns the effective user id of the current process (not necessarily the logged in uid), as shown in Listing 4.

```
{$ifdef LINUX}
function GetComputerName(lpBuffer: PChar; var nSize: DWord): Bool;
begin
  Result := not Bool(gethostname(lpBuffer, nSize))
end;
{$endif}
function ComputerName: String;
var
  Buf: array[0..MAX_COMPUTERNAME_LENGTH] of Char;
  BufLen: DWord;
begin
  BufLen := SizeOf(Buf);
  if not GetComputerName(Buf, BufLen) then
    RaiseLastOSError; //Use RaiseLastWin32Error in Delphi 5
  Result := Buf
end;
```

➤ *Listing 3: The Linux version of GetComputerName.*

```
{$ifdef LINUX}
function GetUserName(lpBuffer: PChar; var nSize: DWord): Bool;
var
  Name: PChar;
begin
  Result := False;
  Name := getpwuid(geteuid())^.pw_name;
  if StrLen(Name) < nSize then begin
    StrCopy(lpBuffer, Name);
    Result := True
  end else
    nSize := Succ(StrLen(Name))
end;
{$endif}
function UserName: String;
var
  Buf: array[0..256] of Char;
  BufLen: DWord;
begin
  BufLen := SizeOf(Buf);
  if not GetUserName(Buf, BufLen) then
    RaiseLastOSError; //Use RaiseLastWin32Error in Delphi 5
  Result := Buf
end;
```

➤ *Listing 4: The Linux version of GetUserName.*

Next on the list is `OutputDebugString` which, in Windows, sends a message to the debugger (visible in the Event Log debugger window), if one is in control of the program. If no debugger is present, the call does nothing. Unfortunately, Linux does not have an equivalent way of communicating with a debugger, but it does have a way of recording messages of interest in the system message log file (typically /var/log/messages) via the `syslog` API.

The `syslog` API communicates with the system logging utility, adding a line of text to the end of the system message log for each call, prefixed with the date, time, host name and application name. The declaration of the routine in C syntax looks like:

```
void syslog(int priority,
  char *format, ...)
```

where the ellipsis at the end of the argument list implies the routine takes a variable number of arguments (much like C's `printf` and `sprintf` routines).

Historically, Object Pascal has had no way of writing a corresponding declaration and as such, calling C routines with variable numbers of arguments was beyond the scope of Object Pascal programmers. However, starting with Kylix, we can now form the appropriate import declaration using a new directive dedicated to accessing these types of external C routines:

```
procedure syslog(__pri:
  Integer; __fmt: PChar);
  cdecl; varargs;
```

The idea is to pass one or more constants (combined with the `or` operator) as the first parameter. The second parameter is then a string which may contain formatting characters compatible with C's `sprintf` (which are much the same as those used by Object Pascal's `Format` function). If formatting characters are used, the values to be used in their place are passed as additional arguments.

Listing 5 shows three calls being made to `syslog`: one with no formatting characters, a second with one, and a third with two. As you can see, sufficient extra arguments have been passed to ensure each formatting string gets a value. Listing 5 also shows the tail end of /var/log/messages, which contains the generated output. Note that this file is marked as only accessible by `root`, so you will need to log in as `root` or become `root` (using `su`) in order to read the file.

➤ *Listing 5: Some output in the system message log file.*

```
syslog(LOG_USER or LOG_INFO, 'Hello world');
syslog(LOG_USER or LOG_INFO, 'User %s running the app',
  getpwuid(geteuid())^.pw_name);
syslog(LOG_USER or LOG_INFO, 'User %s logged in on host %s',
  getpwuid(getuid())^.pw_name, ComputerName);
-----------------
Apr 22 21:48:23 parallelipiped TestApp: Hello world
Apr 22 21:48:23 parallelipiped TestApp: User blong running the app
Apr 22 21:48:23 parallelipiped TestApp: User blong logged in on host
parallelipiped
```

Dynamically Choosing COM Objects

Q I am designing an application where the main program needs to talk to a COM object via an interface, but the COM object can potentially be implemented in different ways to do different jobs. The goal is to have several versions of the same COM object available on a machine and, depending on some setting (maybe in the registry), decide which one to use at runtime. Do you have any recommendations about how I should set about implementing this architecture?

A I can see two approaches to this problem, either one of which should work just fine. Both approaches involve setting up a small type library which defines the interface. Then, for each possible implementation of the COM object, you create an ActiveX Library (as Delphi calls it, but really I mean an in-process COM server project).

The first solution involves each ActiveX Library implementing a COM object, along with a type library of its own. The COM object will be made to implement the interface defined in the original type library, by making each new type library refer to the original one. Each created COM object will have a coclass defined in its type library with a unique `ClassID` (coclass identifier), registered with the system in the normal way. The calling program can choose between any of these registered `ClassID`s when it needs to talk to a COM object.

In a sense, this solution is not an accurate resolution of the stated problem. Since each COM object has a unique `ClassID`, technically they are completely different COM objects, rather than multiple implementations of the same one.

The second solution also involves creating a COM object in an ActiveX Library for each possible implementation. However, this time they will not have their own type libraries and also will not be registered with the system. Additionally, each COM class will use the same `ClassId`, which will be known to the calling application. I feel that this more accurately fits the problem description.

When the program needs to talk to one of the COM objects, it will dynamically load up the ActiveX Library (which is just a DLL, really) and manually create the COM object in the same way that COM would normally do on your behalf. When the COM object is finished with, the ActiveX Library will be freed.

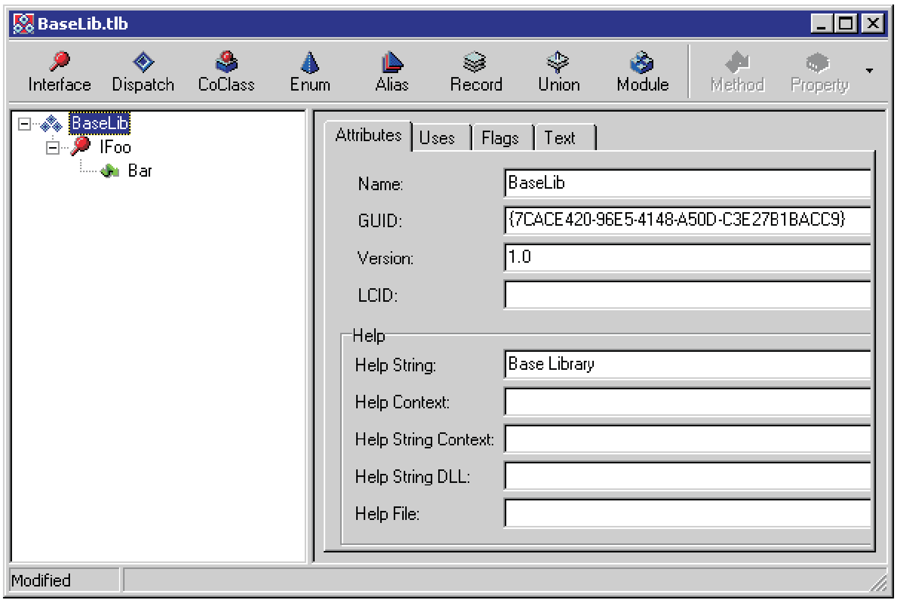
Let's look at implementing the common parts first, then we'll follow both solutions in detail. The first thing needed is a type library so choose `File | New...`, then from the `ActiveX` page of the dialog choose `Type Library`. Save the file as BaseLib.tlb and set its `Help String` attribute on the Type Library Editor's `Attributes` page to `Base Library`. Now add a made-up interface called `IFoo`, with its parent interface set to `IUnknown` (instead of the default `IDispatch`) with a single method called `Bar` (no parameters are needed). Figure 1 shows what we should have.

Next, we need some ActiveX Library projects, which we also select from the `ActiveX` page of the `File | New...` dialog. On this month's disk you can find two such projects, called ComServer1.dpr and ComServer2.dpr.

Now we can try out the first solution, which requires the type library to be registered with the system. This can be done with Delphi's TRegSvr utility:

```
tregsvr BaseLib.tlb
```

In each project we need a COM object. So, starting with ComServer1.dpr, select `COM Object` from the `ActiveX` page of the `File | New...` dialog and in the wizard that pops up, give it a class name of `ComClass1`. When you press `OK`, this COM Server gets its own type library containing an interface



Name: BaseLib
GUID: {7CACE420-96E5-4148-A50D-C3E27B1BACC9}
Version: 1.0
LCID:
Help
Help String: Base Library
Help Context:
Help String Context:
Help String DLL:
Help File:

➤ *Figure 1: Setting up the base type library.*

called IComClass1 and a coclass called ComClass1, set up to implement IComClass1. The goal is to remove the default interface from the type library and change the coclass to implement IFoo instead.

The way to get a coclass to implement an interface defined in another type library was described in the *Type Library Corner Cutting* entry in *The Delphi Clinic* in Issue 51, November 1999. Having deleted the interface from the type library, select the type library node at the root of the tree in the Type Library Editor's object list pane. On the Uses page on the right you will see a list of all the type libraries referenced by this one. We need to add our base type library to this list, so right-click and choose Show All Type Libraries and put a checkmark the Base Library entry.

Now that this type library knows about the base type library you can get the coclass to implement IFoo. Select the ComClass1 coclass, then select the Implements page on the right-hand side of the Type Library Editor. Now right-click, choose Insert Interface, and choose IFoo from the list. A press of the Type Library Editor's Refresh button finishes that side of things, so we can now go to the source.

One final change is needed in the source code of the Delphi class that represents the coclass, currently sitting in an unsaved unit (save it as ComClass1Impl.pas). The TComClass1 class is still set up on the understanding that it will be implementing IComClass1, which of course no longer exists, as well as IFoo. You should remove IComClass1 from the list of implemented interfaces.

In order for the compiler to know what IFoo is, you should also add the type library import unit for the base type library to the uses clause. This unit was automatically generated when you saved the base type library, and was called BaseLib_TLB.pas. The project should compile successfully now with this bare COM class, so all that is needed is some code in the TComClass1.Bar method. A side effect of using a type library to specify that the COM class implements IFoo is that this class already has the IFoo methods (which amount to a single method in this case) declared and implemented with empty methods. Listing 6 shows the whole unit.

The first test COM server is complete so it needs to be registered, either from the command line with TRegSvr or with the IDE's Run | Register ActiveX Server menu item.

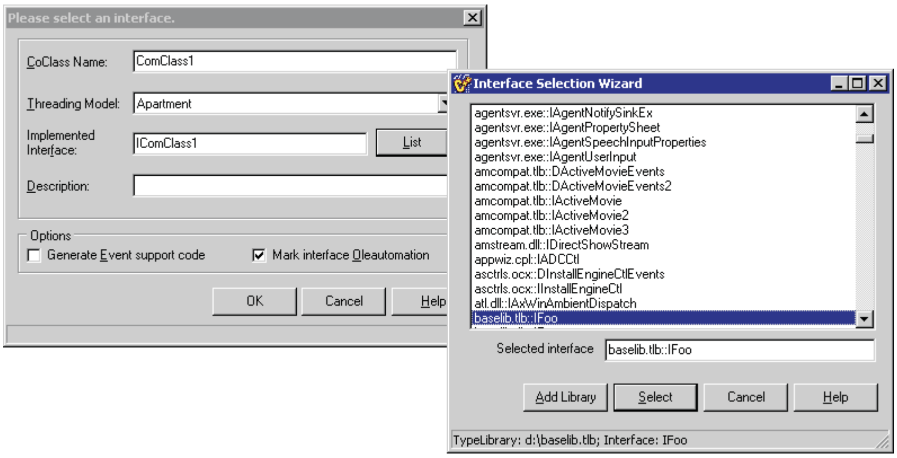
Incidentally, this rigmarole of setting up a COM class to implement an interface from a different type library is much simplified in Delphi 6 (which may be out by the time this is printed and will doubtless borrow this nicety from C++Builder 5). The COM Object wizard will have a List button that shows you a list of all the interfaces defined in registered type libraries, and which you can choose from (see Figure 2).

You now need to go through this whole procedure of making a new COM object, deleting the interface from the type library, using the base type library, making the coclass implement IFoo and fixing the source code for the other ActiveX library we generated earlier.

Both these COM servers had type libraries manufactured in them, and both type libraries will have type library import units generated for them automatically, called ComServer1_TLB.pas and ComServer2_TLB.pas respectively. A test project can create

➤ *Listing 6: The COM class implementation unit.*

```
unit ComClass1Impl;
interface
uses
  Windows, ActiveX, Classes, ComObj, ComServer1_TLB, StdVcl, BaseLib_TLB;
type
  TComClass1 = class(TTypedComObject, IFoo)
  protected
    procedure Bar; safecall;
    {Declare IComClass1 methods here}
  end;
implementation
uses
  ComServ, Dialogs;
procedure TComClass1.Bar;
begin
  ShowMessage('Hello from an instance of TComClass1 in ComServer1.dll')
end;
initialization
  TTypedComObjectFactory.Create(ComServer, TComClass1, Class_ComClass1,
    ciMultiInstance, tmApartment);
end.
```



Please select an interface.

CoClass Name: ComClass1

Threading Model: Apartment

Implemented Interface: IComClass1

Description:

Options

☐ Generate Event support code ☑ Mark interface Oleautomation

Interface Selection Wizard

agentsvr.exe::IAgentNotifySinkEx
agentsvr.exe::IAgentPropertySheet
agentsvr.exe::IAgentSpeechInputProperties
agentsvr.exe::IAgentUserInput
amcompat.tlb::DActiveMovieEvents
amcompat.tlb::DActiveMovieEvents2
amcompat.tlb::IActiveMovie
amcompat.tlb::IActiveMovie2
amcompat.tlb::IActiveMovie3
amstream.dll::IDirectShowStream
appwiz.cpl::IADCCtl
asctrls.ocx::DInstallEngineCtlEvents
asctrls.ocx::IInstallEngineCtl
atl.dll::IAxWinAmbientDispatch
baselib.tlb::IFoo

Selected interface: baselib.tlb::IFoo

TypeLibrary: d:\baselib.tlb; Interface: IFoo

➤ *Figure 2: Implementing a registered interface in C++Builder 5.*

```
uses
  BaseLib_TLB, ComServer1_TLB, ComServer2_TLB;
procedure TForm1.Button1Click(Sender: TObject);
var
  Foo: IFoo;
begin
  Foo := CoComClass1.Create;
  Foo.Bar
end;
procedure TForm1.Button2Click(Sender: TObject);
var
  Foo: IFoo;
begin
  Foo := CoComClass2.Create;
  Foo.Bar
end;
```

➤ *Listing 7: Create instances of COM objects implementing the same interface.*

```
library ComServer1;
uses
  ComServ;
exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;
{$R *.RES}
begin
end.
```

➤ *Listing 8: An ActiveX Library project.*

either COM object by using these two import units and using the helper classes defined therein. Listing 7 shows some code from the ComClient.dpr test project. Note that it also uses BaseLib_TLB.pas for the IFoo interface type definition.

You can see this project creating one of the two COM Objects and calling the Bar method in Figure 3.

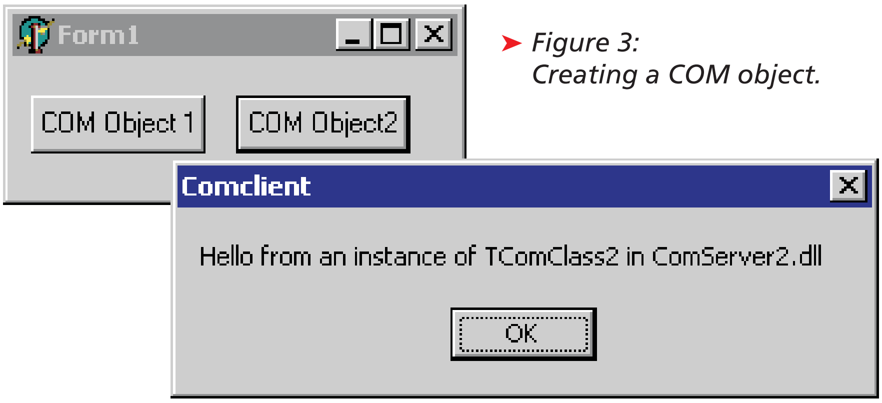
CoComClass1 is a small helper class which is defined in ComServer1_TLB.pas; it asks COM to make an instance of the corresponding coclass (ultimately this will be an instance of TComClass1 inside ComServer1.dll). It gets an IUnknown interface back and queries it for IFoo support. Assuming the COM object claims to support IFoo, an IFoo reference is returned. The Create method is a class method rather than a constructor, so you don't actually create an instance of CoComClass1 (which would require you to destroy it). You just call one of its methods, saving you from talking directly to the COM API when creating an instance of this type of coclass.

The same is true for CoComClass2 being a helper class representing TComClass2 defined in the ComServer2_TLB.pas import unit.

Well, that was the first solution. Your normal registered COM approach, but needing to tweak the automatically generated type library and COM class implementation unit, and make each new COM server reference the base type library. Now let's see how the other solution differs.

Just as before, we start with a base type library defining the common interface, IFoo, although this time the type library need not be registered. We also start with a pair of empty ActiveX Library projects. Notice the four exported routines from the project source (see Listing 8). DllRegisterServer and DllUnregisterServer are exported to (as their names suggest) facilitate registering and unregistering the COM server. These routines are called by the Register ActiveX Server and Unregister ActiveX Server items on the IDE's Run menu, and also by TRegSvr and Windows' own RegSvr32 (see *OCX Deployment* in *The Delphi Clinic* in Issue 19, March 1997, for more information).

The other two routines are called by COM when a client application requests for a COM object to be created. COM is given the target ClassID and looks it up in the registry to find which COM server contains it. If the server is a DLL, COM loads it into memory and calls DllGetClassObject. This routine takes the ClassID and returns a reference to the class factory for the COM object. The class factory implements the IClassFactory interface which is returned. COM then calls the class factory's CreateInstance method whose job is to construct an instance of the COM class and return it.



Form1

Comclient

Hello from an instance of TComClass2 in ComServer2.dll

➤ *Figure 3: Creating a COM object.*

The `DllCanUnloadNow` routine is called by COM to check if there are any COM objects still alive in the server DLL, before possibly unloading it.

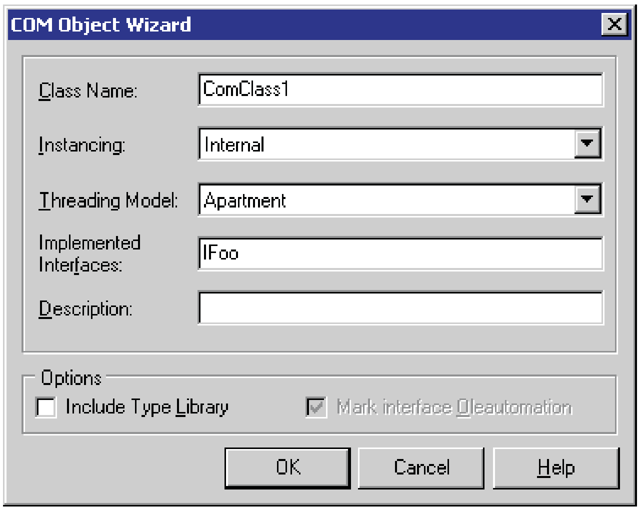
This second solution will replicate some of this behaviour to show another way of creating different objects that implement the same interface. Each COM object will use the same ClassID, defined in a common unit. The calling program need know only the ClassID and the name of the DLL in order to access an object implementing `IFoo`.

The first thing we need is a new unit, which will be shared by both COM servers and also the calling program. The only thing to go in the unit is the common ClassID shared by both COM objects (see Listing 9).

Now, in ComServer1.dpr, invoke the COM Object Wizard again, through the `File | New...` dialog. Set the `Instancing` value to be *Internal*, meaning that it will not be registered, uncheck the `Include Type Library` checkbox and specify `IFoo` will be implemented (see Figure 4).

The resultant unit can be saved as ComClass1Impl.pas and then we can make some changes to it. Firstly, `BaseLib_TLB` must be added to the `uses` clause so the definition of `IFoo` can be found. Next, since we do not have a type library in the project, we must manually enter the declaration of the `Bar` method.

➤ *Figure 4: Setting up a non-registered COM object.*



COM Object Wizard

Class Name: ComClass1

Instancing: Internal

Threading Model: Apartment

Implemented Interfaces: IFoo

Description:

Options

□ Include Type Library ☑ Mark interface Oleautomation

```
unit CommonUnit;
interface
const
  SharedClassID: TGUID = '{1D02060B-0A08-4E8F-A57A-CCAF038445AF}';
implementation
end.
```

➤ *Listing 9: The shared ClassID.*

```
unit ComClass1Impl;
interface
uses
  BaseLib_TLB, Windows, ActiveX, Classes, ComObj;
type
  TComClass1 = class(TComObject, IFoo)
  protected
    {Declare IFoo methods here}
    procedure Bar; safecall;
  end;
//const
//  Class_ComClass1: TGUID = '{1D02060B-0A08-4E8F-A57A-CCAF038445AF}';
implementation
uses
  ComServ, Dialogs, CommonUnit;
{ TComClass1 }
procedure TComClass1.Bar;
begin
  ShowMessage('Hello from a TComClass1 in ComServer1.dll')
end;
initialization
  TComObjectFactory.Create(ComServer, TComClass1, SharedClassID,
    'ComClass1', '', ciInternal, tmApartment);
end.
```

➤ *Listing 10: The non-registered COM class.*

Fortunately, with the declaration entered, a press of `Shift+Ctrl+C` will enter the method body. The method can be made to produce a simple message box as before.

Next, you should remove the `ClassID` automatically entered by the wizard, and add `CommonUnit` to the `uses` clause, so our shared `ClassID` can be referenced. The final change is to the class factory object being created in the initialisation section. The third parameter is the `ClassID` to associate with `TComClass1`, and so should be changed from `Class_ComClass1` to `SharedClassID`. The changed unit can be seen in Listing 10.

Now repeat these steps of making and fixing the COM object in ComServer2.dpr and we can move onto the calling program.

To call either of these COM servers, the test program must again use `BaseLib_TLB`. Much like the previous test program, this one has a pair of buttons. Each one calls a utility routine, `GetIFoo`, to obtain an `IFoo` interface reference from a COM object, passing in the name of the DLL that contains it. The code for `GetIFoo` is shown in Listing 11.

To keep a track of all the loaded DLLs, `GetIFoo` maintains a string list containing the DLL names and also their handles. This allows the corresponding `TidyUpDllList` routine to safely unload the DLLs later.

`GetIFoo` declares a function variable (a typed function pointer) that is assigned the address of the DLL's `DllGetClassObject` routine, if found. It is then called to get a reference to a factory object, whose `CreateInstance` method is then called to get the target COM Object.

At the end of the program, the `TidyUpDllList` routine is called, and that also declares a function variable, this time for `DllCanUnloadNow`. For each DLL in the list, the routine is called to verify there are no COM objects still being maintained by the COM server, before unloading the DLL from memory.

```
uses
  BaseLib_TLB, CommonUnit, ActiveX, ComObj;
var
  DllList: TStringList;
function GetIFoo(const DllName: String): IFoo;
var
  Idx: Integer;
  Dll: THandle;
  DllGetClassObject: function (const CLSID, IID: TGUID;
    var Obj): HResult; stdcall;
  FooFactory: IClassFactory;
begin
  //Is DLL already in list?
  Idx := DllList.IndexOf(DllName);
  //If not, load it and add it
  if Idx = -1 then begin
    Dll := LoadLibrary(PChar(DllName));
    if Dll = 0 then
      RaiseLastWin32Error;
    DllList.AddObject(DllName, Pointer(Dll));
  end else
    //else locate it
    Dll := THandle(DllList.Objects[Idx]);
  //Find the key function
  DllGetClassObject :=
    GetProcAddress(DLL, 'DllGetClassObject');
  if not Assigned(@DllGetClassObject) then
    RaiseLastWin32Error;
  //Call it to get the class factory
  OleCheck(DllGetClassObject(SharedClassID, IClassFactory,
    FooFactory));
  //Ask the class factory the COM object
  if Assigned(FooFactory) then
    OleCheck(FooFactory.CreateInstance(nil, IFoo, Result));
end;
procedure TidyUpDllList;
var
  I: Integer;
  DllCanUnloadNow: function: HResult; stdcall;
begin
  for I := 0 to DllList.Count - 1 do begin
    DllCanUnloadNow :=
      GetProcAddress(THandle(DllList.Objects[0]),
      'DllCanUnloadNow');
    if Assigned(@DllCanUnloadNow) and
      (DllCanUnloadNow = S_OK) then
      Freelibrary(THandle(DllList.Objects[0]));
  end;
  DllList.Free;
  DllList := nil
end;
```

➤ *Listing 11: Asking a class factory to create a COM object.*

That concludes the second possible solution to the problem, which requires absolutely no registration of COM servers or type libraries. Both solutions are in the usual place on this month's disk, but each has its own subdirectory, COMSolution1 or COMSolution2.

Type Library Editor Quirk

Q Have you ever seen the Type Library Editor appending underscore characters onto the end of interfaces? This occasionally happens to me when building COM servers, but I have never found out what causes it to happen. Any ideas?

A I know that there is a large potential for type libraries which are being imported to use identifiers which clash with identifiers already defined in Delphi. To help overcome this problem, the type library importer uses a text file called TLBIMP.SYM. The text file is formatted like an INI file and contains lists of known type library identifiers that need to be modified when encountered.

Full details of how this file is used can be found in *Remapping Names Defined In Type Libraries*, by Robert West of Borland R&D at

```
http://community.borland.com/
  article/ 0,1410,6328,00.html
```

Additionally, identifiers that are encountered in type libraries that match language reserved words (such as `type`, `unit` and `String`) are automatically modified by suffixing them with trailing underscores. However, both of these factors only affect the case when an existing type library has been imported with `Project | Import Type Library...`, or the TLIBIMP command-line tool.

The questioner is experiencing these modified identifiers when just building a COM server application. The only time I have heard of this problem is when a coclass is given the same name as the type library itself (which matches the project name by default).

When testing this idea, I found it difficult to see the symptom. Asking for a COM object with a coclass name set the same as the project (and therefore the same as the default type library name) results in an error: *The project already contains a form or module named XXX*.

Changing the type library name beforehand, and asking for a COM object with the same name gave an Access Violation and, whilst the coclass and interface are added to the type library, the unit containing the Delphi COM class was not manufactured.

However, after jigging around with things for a bit, I managed to get to see the symptom. The type library had the coclass and interface with the names I requested, and the type library import unit appended an underscore onto anything related to the coclass name.

Acknowledgements

Thanks are due to Martin Curtis for helping me find the cause of the type library identifier modification.