The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project?

Just email Brian Long, our Delphi Clinic Editor, on clinic@blong.com

Unexpected COM Error

Q I have an out-of-process Automation server which is being accessed from an Automation client. A timer in the client regularly calls functionality in the server. If I right-click on the client's icon in the task bar (to produce the application system menu) I get an exception each time my timer ticks, saying: *'An outgoing call cannot be made since the application is dispatching an input-synchronous call'*. What does this mean?

A This error has cropped up before in *The Delphi Clinic*. Back in Issue 14 a reader wondered why a Delphi 2 Automation client gave a misleading message under exactly the same circumstances. The message, *'Method xxxx is not supported by OLE object'*, was displayed by Delphi 2 when any error-indicating `HResult` was returned from an attempt to call the Automation server's `IDispatch.GetIDsOfNames` method, regardless of whether it really was a `DISP_E_UNKNOWNNAME` error or not. This was done because the default Windows descriptive message for the `DISP_E_UNKNOWNNAME` of *'Unknown name'* was considered too terse.

However, there are more error-indicating `HResult` values that can be returned by `IDispatch.GetIDsOfNames` than just `DISP_E_UNKNOWNNAME`. Under the circumstances outlined in the question, the error actually produced from a call to any COM interface method is `RPC_E_CANTCALLOUT_ININPUTSYNCCALL`. The problem is all to do with COM threading models and synchronisation of COM method calls.

In summary, the facts are as follows. The client application has its primary thread running in a single-threaded apartment (STA). For more information on apartments, see my *Multi-Threading And COM* article in this issue. The client thread enters the apartment by using ComObj either explicitly or implicitly, and ComObj by default enters an STA, unless you specify otherwise with the `CoInitFlags` variable.

Since the client and server are in different processes, the client application talks to a COM proxy object that represents the real server COM object. Proxy objects operate on cross-process COM method calls by turning them into Windows messages with appropriate parameter data. The message is posted to a COM-managed window in the server application to be dealt with, by turning it into a real method call within that process.

When the method finishes executing, a message is delivered back to the client, allowing it to terminate a COM-managed modal message loop that stops any more client thread code executing. Notice that the client thread code relies upon the server's thread being alive and awake in order for the message to be processed, so the client can ultimately continue.

Consider the possible scenario of the server's thread not being available. Maybe it is frozen by a call to `Sleep`, `WaitForSingleObject`, or even `SendMessage`. In these cases, the server will not immediately respond, the response time being determined by when the server thread is able to continue working. This would equate to after `Sleep` has finished its delay, when the object waited for by `WaitForSingleObject` has become signalled, or the window procedure or message handler processing the message has finished processing it and returned.

Now consider the possibility that the server process could have sent a message (with `SendMessage`) to the client's thread. The client may take some time to process the message, meaning the window procedure takes some time to return. Consider also what happens if the client message handler still processes message queue messages, say with calls to `Application.ProcessMessages`.

In this case, the timer object's message will still be processed. If the timer's event handler makes a COM method call under these circumstances, there is the possibility of a deadlock. If the timer event handler calls the COM object whilst the server is blocked waiting for the client to finish processing a sent message, the client will also block waiting for the server to process the COM method call. This means that they are both blocked waiting for each other and neither will back down.

To avoid this situation, COM checks to see if the calling thread is still processing a sent message before allowing a COM method call to be made. If it is processing a sent

➤ *Listing 1: Avoiding errors in COM STA client code.*

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  try
    if not InSendMessage then
      Caption := DateTimeToStr(Clock.CurrentDateAndTime)
  except
    on Exception do
      Timer1.Enabled := False;
      raise
    end
  end;
end;
```

message, an STA client will be given the RPC_E_CANTCALLOUT_ININPUTSYNCCALL error, regardless of who sent the message (that information seems hard to track down).

This is the error seen by the questioner, and this is caused because a right click on the client task bar button sends a message to its Application object to start a system menu processing loop. This message appears to take some time to process, the amount of time that the menu is displayed on the screen. As soon as the menu disappears, the Application message handler returns and so STA-generated COM method calls can be resumed.

You can avoid the error quite simply. If you are writing code that may potentially be called whilst the taskbar button's popup menu is being displayed, then only execute the COM calls if the InSendMessage API returns False. InSendMessage makes a simple check to see if the calling thread is still dealing with a message sent by SendMessage. You can see some simple COM code in Listing 1 that uses InSendMessage to avoid calling a COM method under problem circumstances.

To prove that your application will be in the middle of a SendMessage call whilst a task bar system menu is produced, try the following. Make a new application with a timer component on it. Set the timer's Interval property to 10 milliseconds. Now make an OnTimer event handler for the timer and add in the code from Listing 2.

```
procedure TForm1.Timer1Timer(Sender: TObject);
const
  FormColors: array[Boolean] of TColor = (clLime, clRed);
begin
  Color := FormColors[ InSendMessage ]
end;
```

➤ *Listing 2: Proving that SendMessage is the culprit.*

```
procedure TLogoAppForm.FileSend1Execute(Sender: TObject);
var
  MapiMessage: TMapiMessage;
  MError: Cardinal;
begin
  with MapiMessage do begin
    ulReserved := 0;
    lpszSubject := nil;
    lpszNoteText := PChar(RichEdit1.Lines.Text);
    lpszMessageType := nil;
    lpszDateReceived := nil;
    lpszConversationID := nil;
    flFlags := 0;
    lpOriginator := nil;
    nRecipCount := 0;
    lpRecips := nil;
    nFileCount := 0;
    lpFiles := nil;
  end;
    MError := MapiSendMail(0, 0, MapiMessage,
    MAPI_DIALOG or MAPI_LOGON_UI or MAPI_NEW_SESSION, 0);
  if MError <> 0 then
    MessageDlg(SSendError, mtError, [mbOK], 0);
end;
```

➤ *Listing 3: The Delphi wizard MAPI code.*

This ensures that the form is green unless the application is processing a sent message, whereupon it goes red. Try running it and the form will be green. If you right click on the task bar button, it will immediately go red until the system menu disappears.

I have not been able to reproduce the questioner's error on Windows 2000, which may imply that COM makes more stringent checks before producing such an error on that platform.

Delphi And Email

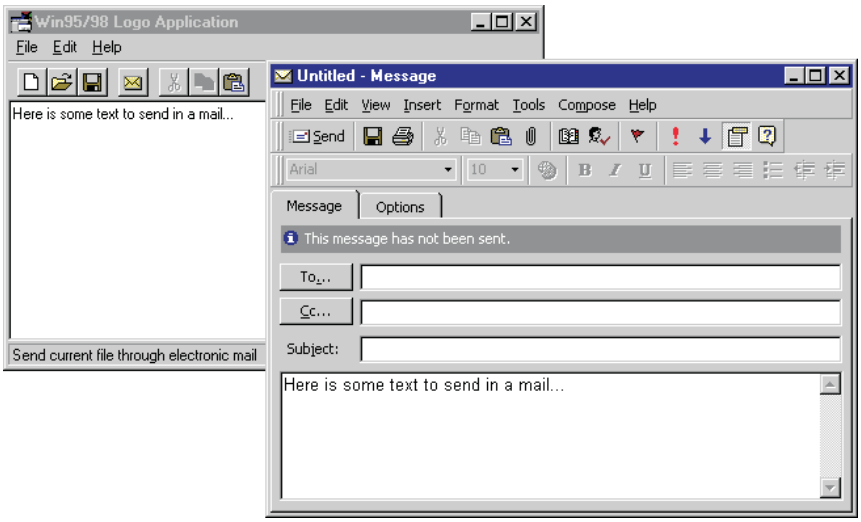
Q What is the best way to send an email with an attachment with it from a Delphi 5 application?

A I wouldn't know how to quantify which is the best way, but let's look at a couple of options available to Delphi 2 and later.

Firstly, you could use MAPI (Microsoft's Mail API). You can get a good start with sending a message using MAPI by using one of the Delphi wizards. Choose File | New... and go to the Projects page and then invoke the *Win95 Logo Application* or *Win95/98 Logo Application*, depending on which version of Delphi is installed. This makes an application in a directory of your choosing that includes a File | Send menu item.

The implementation of this menu item gathers up the text entered in a rich edit control and sets it as a field in a MAPI message record (see Listing 3). It does not concern itself with specifying the sender or intended recipient, or of

➤ *Figure 1: The Delphi wizard generated app sending an email.*



```
procedure SendMail(const Subject, MessageText, MailFromName,
  MailFromAddress, MailToName, MailToAddress: String;
  const Attachments: array of String);
var
  MAPIError: DWord;
  MapiMessage: TMapiMessage;
  Originator, Recipient: TMapiRecipDesc;
  Files, FilesTmp: PMapiFileDesc;
  FilesCount: Integer;
begin
  FillChar(MapiMessage, Sizeof(TMapiMessage), 0);
  MapiMessage.lpszSubject := PChar(Subject);
  MapiMessage.lpszNoteText := PChar(MessageText);
  FillChar(Originator, Sizeof(TMapiRecipDesc), 0);
  Originator.lpszName := PChar(MailFromName);
  Originator.lpszAddress := PChar(MailFromAddress);
  MapiMessage.lpOriginator := @Originator;
  MapiMessage.nRecipCount := 1;
  FillChar(Recipient, Sizeof(TMapiRecipDesc), 0);
  Recipient.ulRecipClass := MAPI_TO;
  Recipient.lpszName := PChar(MailToName);
  Recipient.lpszAddress := PChar(MailToAddress);
  MapiMessage.lpRecips := @Recipient;
  MapiMessage.nFileCount :=
    High(Attachments) - Low(Attachments) + 1;
  Files := AllocMem(SizeOf(TMapiFileDesc) *
    MapiMessage.nFileCount);
  MapiMessage.lpFiles := Files;
  FilesTmp := Files;
  for FilesCount := Low(Attachments) to
    High(Attachments) do begin
    FilesTmp.nPosition := $FFFFFFFF;
    FilesTmp.lpszPathName := PChar(Attachments[FilesCount]);
    Inc(FilesTmp)
  end;
  try
    MAPIError := MapiSendMail(0,
      Application.MainForm.Handle,
      MapiMessage, MAPI_LOGON_UI or MAPI_NEW_SESSION, 0);
    if MAPIError <> 0 then
      MessageDlg(LoadStr(sSendError), mtError, [mbOK], 0)
  finally
    FreeMem(Files)
  end
end;
procedure TLogoAppForm.FileSend(Sender: TObject);
begin
  SendMail('Subject', 'Message text',
    'The Delphi Clinic', 'clinic@blong.com',
    'The Delphi Magazine', 'chrisf@itecuk.com',
    ['c:\autoexec.bat'])
end;
```

➤ *Listing 4: Setting up a full MAPI message.*

setting up an attachment, but acts as a good starting point. In fact the code invokes your installed MAPI client (typically Microsoft Outlook, if installed) and asks it to let you finish setting up the mail, as you can see in Figure 1.

If you want the whole thing to be controlled by the application, then you can set up more of the MAPI message record yourself. Listing 4 shows a `SendMail` procedure that can be used to set up a MAPI message with the sender's details, one recipient and any number of attachments.

You can see that the sender and recipients are described in a `TMapiRecipDescrip` record and each file attachment is described in a `TMapiFileDesc` record. The code allocates memory for as many file description records as are required. It could be modified to also allocate space for an arbitrary number of message recipients, but I tried to keep it as simple as possible.

Note that MAPI can be controlled using Automation, but this code was written using APIs to make it more efficient.

Another possible approach would be to automate Microsoft Outlook, if installed, to send an email (useful if your users already use Outlook and want to keep the same UI). Care should be taken when doing this as undesirable results can be achieved (I refer to various examples of worm viruses that have circulated recently, most notably the Love Bug virus).

```
procedure SendMail(const Subject, MessageText, MailFromAddress,
  MailToAddress: String; const Attachments: array of String);
var
  Outlook, Mail, Recipient: Variant;
  I: Integer;
const
  olMailItem = 0;
  olOriginator = $00000000;
  olTo = $00000001;
begin
  Outlook := CreateOleObject('Outlook.Application');
  Mail := Outlook.CreateItem(olMailItem);
  Recipient := Mail.Recipients.Add(MailToAddress);
  Recipient.Type := olTo;
  Recipient := Mail.Recipients.Add(MailFromAddress);
  Recipient.Type := olOriginator;
  for I := 1 to Mail.Recipients.Count do
    Mail.Recipients[I].Resolve;
  Mail.Subject := Subject;
  Mail.Body := MessageText;
  for I := Low(Attachments) to High(Attachments) do
    Mail.Attachments.Add(Attachments[I]);
  Mail.Save;
  Mail.Send
end;
```

➤ *Listing 5: Sending an email by automating Outlook.*

I discussed automating Outlook to some extent in *The Delphi Clinic* back in Issue 35, so I would recommend having a read through that past coverage, but it did not get as far as sending an email. However, a small amount of knowledge of the Outlook Automation interfaces will allow you to concoct something like Listing 5 (from the Outlook.dpr project).

The code starts an Outlook session, and then creates a mail object with `CreateItem`. The object then has its various properties filled in, including recipient, sender, message and attachments. When this has been done, the message is ready to send. Instead of directly sending the message, you could ask Outlook to display the pending mail, so the user can finish off anything that needs doing. This can be done by calling `Mail.Display` instead of `Mail.Save` and `Mail.Send`.

The FastNet components also offer an option for sending mail. Delphi 4 (and later) has native versions of these components, including the `TNMSMTP` component for sending an email. You can find a demo project for this in Delphi's Demos directory. Delphi 4 has an SMTP directory underneath the Internet directory, whereas Delphi 5 renames Internet to FastNet.

[Another SMTP component worth considering is TSMTPClient, which is included in Frank Piette's excellent free internet component suite (downloadable from www.rtfm.be/fpiette) and supports file attachment using the MIME format. Example programs are included and you get source code too. Ed]

Component Construction

Q I wonder if you could help me with a problem we are having porting a project from Delphi 1 to Delphi 4. As I understood it, a Delphi 4 form is constructed in the following order:

1. The `TForm` constructor starts.
2. Each component on the form is constructed.
3. Each component on the form loads its published properties and hooks in its events.
4. The `TForm` constructor is almost finished so the `OnCreate` event is triggered.

However, in a form in our project we seem to be getting the events hooked in and one of a component's published properties assigned (a `TStrings` property) in phase 2. This is causing a problem as the component's `OnChange` event is being called which references another (as yet uncreated) component, thereby causing an Access Violation.

If we remove the design-time event hooks from the component then everything is fine, but we are rather perturbed as the chain of events (pun unintended) is not as we thought it was. Could you please clear up the exact order of things inside a form's constructor?

A The first thing that happens in a form's `Create` constructor is that another constructor is called. `CreateNew` creates a simple `TForm` object with no DFM form resource accessed in any way. Then `Create` proceeds to include `fsCreating` in its `FormState` property.

Assuming the form resource can be found (which should have been linked into the executable) it is now read in piece by piece using a stream that reads directly from the resource. You can see the form resource in textual form by right clicking on any form designer and choosing `View As Text`. Listing 6 shows a simple form resource displayed in textual form. It represents a form with a button dropped on it.

What you see in the form resource are a number of nested sections describing components and their property values (those with non-default values, or those for which there is no specified default). The outermost section describes the form (or maybe data module or web module, but we will call it a form for simplicity) and contains sections for each child control and non-visual component. For any component that is a windowed control and has other child components, additional sections will be nested within. We will need to look into how these sections are processed to better understand the form streaming mechanism.

If the form has been set up using Visual Form Inheritance (VFI) the code will first process the base form class resource in and work its way back to the actual class form resource one layer at a time. However, for each level the procedure is much the same.

As the code starts reading through these nested object sections, it reads the component class name and passes it to `FindClass` to get a class reference through which it can create the component instance. Next, `csLoading` and `csReading` are included in the component's `ComponentState` property. The component name is then read and applied followed by all the other component properties in turn.

The exceptions to this pattern include the outermost object, which is the form. The form has already been created earlier in the constructor and so the form class and name are skipped during the stream read. The same applies when dealing with VFI. If a component has already been created thanks to an ancestor form class resource, it is not recreated when modified properties are read in from a descendant form resource.

As the properties are read, special consideration is given to event properties and object reference properties. In the case of event properties, the form's published method table is used to connect the property to the address of the relevant method in the form, as described in my *Fatal Startup Error* article from Issue 30.

Problems could potentially arise when the streaming code reads in properties that are supposed to be connected to other components. What if the linked-to component has not been created by the time a property refers to it? Fortunately such technical hitches have been foreseen and the streaming code uses a fix-up list for any object reference properties that are encountered. All these fix-ups are processed (meaning the pending object references are 'fixed up') after all components on the form have been constructed and otherwise fully read in.

Given that Delphi 2 and later allow you to connect a property to a component on another form (which itself may not be created), some fix-ups may remain unresolved at this point. These are left in the fix-up list to be attempted again when the next form has finished initialising its components. This continues after each new form is created until (hopefully) all fix-ups are dealt with.

When the properties are all read in, `csReading` is removed from the `ComponentState` property. Next the new component is assigned to the relevant object reference data field in the form using the form's published field table (also discussed in the article from Issue 30). At this point the aforementioned fix-ups are processed.

When all components on the form have been constructed and had their properties read in,

➤ *Listing 6: A form resource viewed as text.*

```
object Form1: TForm1
  Left = 192
  Top = 107
  Width = 112
  Height = 71
  Caption = 'Form1'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13
  object Button1: TButton
    Left = 8
    Top = 8
    Width = 75
    Height = 25
    Caption = 'Button1'
    TabOrder = 0
  end
end
```

the Loaded method for each component is called in turn. The default implementation of Loaded removes csLoading from ComponentState. Loaded can be used by components to perform actions that rely on the values of several properties that could have been stored in any arbitrary order. For example, TTable waits until Loaded in order to deal with an Active property having a stored property of True. This saves problems if the Active property is read in before DatabaseName, TableName and so on.

Now that the form's constructor is nearly over, fsCreating is removed from FormState. In Delphi 1, 2 and 3 the OnCreate event is now triggered just before the constructor exits and the same is true in Delphi 4 and later if the OldCreateOrder property is True. If OldCreateOrder is False (the default), Delphi 4 and later wait until the form's AfterConstruction to trigger OnCreate.

So to summarise the order of events:

1. The TForm constructor starts.
2. The form resource is traversed, constructing each component in turn and setting its properties and events, and assigning it to the corresponding form data field before moving onto the next component.
3. Any fix-ups recorded for object reference properties are tied up.
4. Each constructed component's Loaded method is called.
5. The TForm constructor finalises and the OnCreate event is called.

The problem is likely to be caused by not realising each property is assigned individually, causing the corresponding property writer to execute. If you have properties whose assignments will trigger the OnChange event, perhaps you should check if csReading is in ComponentState before going ahead and triggering it.

Code Completion

Q When I use Class Completion in Delphi 5 to generate method implementations, it sometimes inserts the inherited keyword in the method body. What criteria does it use to decide when to do this?

A Class Completion, as introduced in Delphi 4, is a time saver for implementing the basic skeleton of procedure methods, function methods and properties (typically involving methods as well). Ignoring properties, Class Completion can work from a method declaration and enter the implementation skeleton for you or take a method implementation and insert an appropriate declaration in the class. All you do is press Shift+Ctrl+C when the cursor is in a class definition or method, or right-click and choose Complete class at cursor.

If a method declaration involves the reserved word override, it is clear that this is a polymorphic method, which will be declared in one of the ancestor classes with an identical signature, marked with either virtual, dynamic or override.

Often when overriding an existing polymorphic method, a developer will wish to chain back to the original definition by using the inherited keyword, before executing their own code. This way, the behaviour of a polymorphic method in a class can be extended in descendant classes.

This is done using the inherited keyword, optionally in conjunction with the method name and parameters. Take this method declaration:

```
procedure Foo(Bar: Integer);
  override;
```

In the implementation, the developer can call the original version of the routine inherited from the ancestor class like this:

```
inherited Foo(Bar);
```

Of course, alternative parameters can be passed if desired. However, assuming the same parameters are to be passed to the ancestor method, this statement can be abbreviated to inherited.

```
destructor TForm1.Destroy;
begin
  inherited;
end;
```

➤ *Listing 7: A destructor generated by Class Completion.*

Class Completion will insert the reserved word inherited in the implementation of any overridden polymorphic procedure method, followed by a blank line where you can start entering your code. If you do not want to call the inherited version, you can remove the call. Note that it uses the abbreviated version to save having to enter references to the method name along with all the parameters.

This simple template is used regardless of the intent of the method and so is also the case in overridden destructors (which are procedures with an extra syntactic implication of ultimately releasing the object instance's allocated memory space). This explains why a destructor generated by Class Completion looks like Listing 7. Normally with destructors, you execute your own tidying up code before chaining back to the inherited destructor. However, Code Completion does not realise it is dealing with a destructor.

Having looked so far at procedure methods, let's now consider function methods. Functions must return a result, and so a common pattern for the implementation of an overridden function method is to precede your extra code with:

```
Result := inherited Foo(Bar);
```

However, there is no valid abbreviation for this, so Class Completion doesn't bother inserting anything. The Delphi 4 and 5 version is not written with enough flexibility to generate the required statement so it does nothing. Maybe Delphi 6 will have a more intelligent Class Completion...