Multi-Threading And COM

by Brian Long

Growing numbers of Delphi developers are getting into COM. This is a good thing, as COM is providing more of the underpinnings of Windows application development.

There is a growing amount of information for Delphi developers on COM: I've listed some references at the end. However, most of this deals with COM in simple, non-threaded applications; there is little on multi-threaded COM applications for the Delphi developer (but see Reference 1).

This article aims to provide an understanding of the concepts, terminology and issues involved in multi-threaded COM programming. It does not try and teach you safe multi-threaded programming, which is a separate topic. Instead, it assumes you have an appreciation of the general issues of multi-threaded programming, such as the necessity to protect or synchronise access to resources accessible from more than one thread. If you need more information on this, see References 6 and 7.

I should mention that Delphi 3 did not have built-in support for thread-safe COM objects. Delphi 4 added some support, although it needed a little hand-holding. Delphi 5 was the first version to cater automatically for thread-aware COM objects in the RTL.

Apartments

No, I'm not about to launch into some spiel about American city dwellings. Instead, an *apartment* is an important thread-related concept in COM. When you read about multi-threaded COM programming you might find it all rather confusing, due to the constant references to various types of apartments, so I will start by investigating what an apartment is.

As Windows developers, you are probably comfortable with the way parts of applications are managed by Windows. When a program is launched, Windows creates a process 'object' containing information about that process. I placed the word *object* in quotes because, whilst it may well be stored as an object, programmers do not have access to it in such a form.

Each process object has a unique identifying number, or process identifier (a `DWord` or unsigned 32-bit number). Applications can also gain varying degrees of access to the process object by getting a handle to it, which is called a process handle (a `THandle`, which again is a 32-bit number).

When the program initiates any threads, be it the first (or primary) thread of an application, or additional (secondary) threads, Windows similarly manages them with thread objects. These contain the state information of the thread, or the thread's context. These thread objects also have unique identifiers (thread identifiers) and can be accessed through a thread handle.

It perhaps should be mentioned at this point that the Delphi VCL is almost entirely *not* thread-safe. You should therefore use some form of synchronisation to access any VCL objects from any thread other than the primary thread. In a normal thread class (inherited from `TThread`), you would use the `Synchronize` method.

When an application thread creates a window, Windows also manages that via some internal data structure containing information about the window, which we refer to with a window handle (a `HWnd` or `THandle`).

Processes, threads and windows are old hat as Windows-managed entities. Apartments are not as common to most, despite the fact that any application dealing with COM contains at least one of them. Apartments also reside in an executable and are managed by Windows, but we programmers do not have easy access to them via handles or identifiers. Nevertheless, they get created (by COM) and are managed for us by the OS.

The job of an apartment is to control how interface method calls are dispatched to COM objects in a multi-threaded environment. If a COM object has not been written with the ability to deal with simultaneous access from multiple threads (in other words is not thread-safe) the apartment mechanism will ensure that the object receives only one method call at a time. If the object has been written to be thread-safe, the apartment permits simultaneous access from multiple threads.

Apartment Types

The apartment therefore acts as a filter or funnel that ensures the COM object gets called in a manner that it can deal with. As was subtly implied by the previous paragraph there are two types of apartment: one that enforces serialised method calls (ensures only one method call executes at any time) and one that does not.

The first type is referred to as a *Single-Threaded Apartment*, or STA. Old Microsoft documentation referred to the STA model simply as the *apartment-threaded model*. The STA was introduced in Windows NT 3.51 and Windows 95. The second is called a *Multi-Threaded Apartment*, or MTA. The MTA model is also known as the *free-threaded model*. The MTA was introduced in Windows 95 with DCOM installed and in NT 4.0.

There is also a third apartment type, the *Thread-Neutral Apartment*, or TNA, introduced by COM+ on Windows 2000, but I'll ignore that for the time being, to keep things as simple as possible.

A COM object must belong to an apartment, be it an STA or an MTA. Moreover, it can only belong to one apartment. A COM object can advertise what type of apartment it is designed to work with by its *threading model*. In Delphi's COM

Object Wizard you can choose an object's threading model. In-process COM objects advertise their threading model in the Windows registry, as we will see later.

Any thread that accesses a COM object also belongs to exactly one apartment at a time. An apartment contains one or more threads in an application that can interact with COM (for example by calling COM object methods). If a thread in an apartment creates a COM object, that object belongs to the same apartment, although technically this may not always be the case.

An apartment can be shared by any number of COM objects, which will all be accessible by clients in a manner dictated by the apartment type. Once instantiated, COM objects can never move to other apartments, and so an apartment is part of the object's identity.

An STA is so named because only one thread belongs to it. However, an arbitrary number of threads can belong to an MTA. The enforced rule is that *only threads in a COM object's apartment are able to directly call its methods*. If an object needs to be accessed by a thread in another apartment, either in the same process or in a different one, the interface reference must be marshaled to the apartment. This concept will be detailed later.

Before moving on to the next section I must acknowledge that if this is all new to you, then it probably sounds very woolly, imprecise, up in the air and difficult to relate to. Also, all this nebulous talk of apartments, with no real details, may be prompting the popular *'Er, so what?'* response. This is perfectly understandable.

As we proceed, real details will be revealed, with the intention of solidifying this background information. If the first portion of this article makes little sense on first reading, please be patient. Read through to the end, then start again. You should find that it all becomes much clearer when read through a second time.

It's a chicken and egg situation really. You need the background information to appreciate the real details later, but you also need all those details in order to appreciate what the background is telling you. Take a deep breath, let's get back to the story...

How Apartments Are Created

Whilst a process must have at least one thread in order to work, it can have zero or more apartments. Remember that a thread which calls COM routines must belong to an apartment. Any thread can access COM objects, with the caveat that the thread must initialise the Windows COM subsystem to do so. So an application with no apartments does not interact with COM.

This implies that a thread enters an apartment when COM is initialised by that thread (we'll see later how the apartment type is determined). When a thread enters an apartment, COM ensures the apartment exists, creating it if need be.

COM stores information about the apartment in the TLS (the thread's local storage area). If that thread then proceeds to create a COM object instance, that COM object belongs to that thread's apartment. The thread leaves the apartment by shutting down COM before terminating.

An application is able to have at most one MTA, but can have as few or as many STAs as you choose. Any thread that enters an STA creates a new apartment. Any thread that enters an MTA will either create the application's sole MTA (if it is the first thread to enter it) or enter the existing MTA created by some other thread entering it earlier.

When you play with COM in Delphi, you might not realise it but COM is implicitly initialised in order for you to do so. Typically you will have `ComObj` in your `uses` clause, or maybe `ComServ` (which itself uses `ComObj`). The initialisation section of `ComObj` (in Delphi 4 and later) ensures that the project source's `Application.Initialize` will cause COM to be initialised for your primary thread, and the finalisation section closes down COM.

This means that any Delphi application that uses `ComObj` or `ComServ` sets up some type of apartment. It also means that you must ensure that the `Application.Initialize` call remains at the beginning of the code in the project file (.DPR file) for this to work correctly.

The implication of all the above is that all COM applications are made up of one or more apartments, be they COM clients or COM servers. Any process that makes COM calls will involve at least one apartment.

In the case of a client talking to an out-of-process COM server, both the client and server will have at least one apartment (they each have at least one thread doing COM things).

With a client talking to an in-process server, there is only one process in the equation. Nominally you might assume that there would be one or more apartments. However, if the client apartment type does not match the thread capabilities of the COM object, COM will helpfully set up a thread of its own in a new apartment, thereby giving a minimum of at least two apartments (we will come back to this later).

Why Have Apartments?

This is a good question to look at again at this point. What is the purpose of an apartment? Well, when you develop a COM object you are at liberty to go to town building in thread safety at every appropriate point (synchronising access to any shared resources and so on) to enable your object to be safely accessed by many concurrent threads. Alternatively, you may choose to write your COM object in a thread-ignorant way, avoiding the concurrency issues altogether. It's your choice.

However, since registered COM objects can be accessed by any COM client application, single-threaded or multi-threaded, thought needs to be given to what might happen. The client may, for example, have multiple threads of its own, each of which will make calls to the object. Or there could be many client applications running, all of which communicate with the same COM object. Either way, if your COM object is not thread-safe, this could prove problematic. Or it could if COM didn't solve the problem on your behalf.

If your COM object is not thread-safe, it should be created in a single-threaded apartment. If it is thread-safe, it can be created in a multi-threaded apartment. In the case of a non thread-safe object in an STA, the apartment itself ensures that method calls come in one at a time from the various threads around the system, regardless of how simultaneously the calls are made. On the other hand, an MTA does not control the order in which method calls arrive, leaving your thread-safe COM object able to service as many concurrent calls as it can.

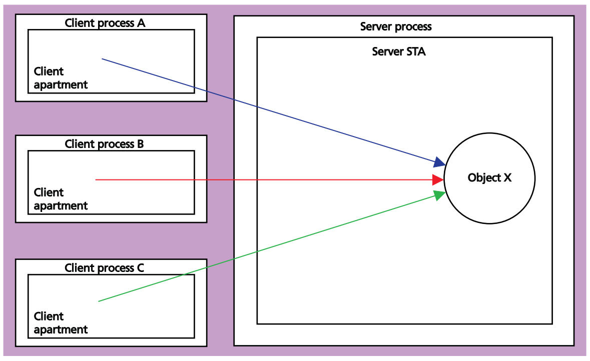
How Does An STA Work?

It is all very well claiming that an STA serialises COM object method calls, but it might help firm things up in your mind if you understand how this magic is performed.

COM sets up a *proxy object* when a COM client wants to talk to a COM object in a different apartment or process (see Reference 5). The proxy is set up to look exactly like the target object, so the client application is not made aware of the deception. The client's interface reference actually points to the proxy.

In order for COM to do this, the type library that describes the interface must be registered so COM can learn how the interface is laid out. In DCOM applications, this means the server's type library must be registered on the client machine.

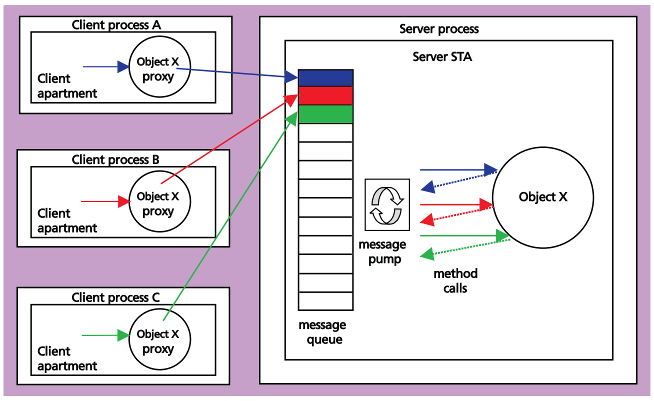
When the client calls one of the COM object methods it actually talks to the proxy. The proxy then deals with communicating the method call across to the real object in the server. Since the proxy is associated with the COM object's apartment, it knows whether method calls should be serialised or not. If the COM object resides in an STA then the proxy must somehow ensure that method calls arrive at the real object one at a time.



➤ *Figure 1: Three client processes simultaneously calling methods of an object in an STA.*

To do this, COM creates a hidden window for each STA. Each method call is translated into a Windows message (with all the pertinent parameter data packaged up). The message is posted to the hidden thread window with `PostThreadMessage`, thereby ending up in the thread's message queue along with any other calls to methods of this or other objects in the same STA. All method calls made by threads not belonging to the COM object's STA are dealt with in this way.

There is a requirement that the thread in an STA have a message loop (or *message pump*) to sequentially pull out each message and send it to the target window's window procedure. The primary thread of any Delphi application already has one of these in the `Application` object. It is this message loop that is responsible for taking all the usual user-generated mouse and keyboard messages and directing them to the right forms and controls. Any additional threads that wish to initialise COM and enter a new STA must also have a message loop (we will get onto the details later when we look at creating new STAs).



➤ *Figure 2: What really happens when simultaneous STA method calls are made.*

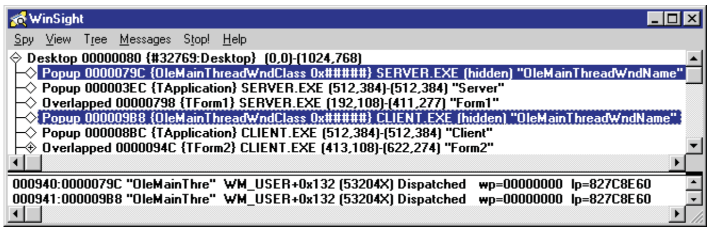
Since all the method calls end up as messages in a queue, the message loop will pull each message from the queue one at a time, and translate them into method calls one at a time. Each message is individually pulled from the queue and given to the hidden COM window's window procedure, which extracts all the parameters and calls the object directly. Since the message loop is part of the STA's thread this is perfectly permissible.

The fact that STA COM object method calls are made via a message queue and message loop inherently means that all STA COM object method calls will be serialised.

Figure 1 shows the surface view of several COM clients making simultaneous calls to an object in an STA in some other process. To the clients, things are very simple. They make method calls. The method calls execute. However, as explained above, things are more complex under the hood, in order to ensure the method calls occur one after another.

Figure 2 shows a more realistic view of things. Each client makes a method call through the supplied COM proxy object. The COM proxy posts a message into the server STA hidden COM window message queue. The message pump pulls each one out in turn, resulting in a method call, but each method is called from the STA's single thread in a serial fashion.

Figure 3 shows WinSight displaying all the windows that are part of a simple COM client and server set-up (the server is an out-of-process executable). Each application has a `TApplication` window



```
Desktop 00000080 {#32769:Desktop}  (0,0)-(1024,768)
  Popup 0000079C {OleMainThreadWndClass 0x#####} SERVER.EXE (hidden) "OleMainThreadWndName"
  Popup 000003EC {TApplication} SERVER.EXE (512,384)-(512,384) "Server"
  Overlapped 00000798 {TForm1} SERVER.EXE (192,108)-(411,277) "Form1"
  Popup 000009B8 {OleMainThreadWndClass 0x#####} CLIENT.EXE (hidden) "OleMainThreadWndName"
  Popup 000008BC {TApplication} CLIENT.EXE (512,384)-(512,384) "Client"
  Overlapped 0000094C {TForm2} CLIENT.EXE (413,108)-(622,274) "Form2"

000940:0000079C "OleMainThre"  WM_USER+0x132 (53204X) Dispatched   wp=00000000  lp=827C8E60
000941:000009B8 "OleMainThre"  WM_USER+0x132 (53204X) Dispatched   wp=00000000  lp=827C8E60
```

➤ *Figure 3: WinSight showing hidden STA windows.*

and a form window (normal for a Delphi app). However, since both the client and server processes initialise COM, they both have an apartment. COM creates a hidden window for each apartment with a window class of `OleMainThreadWndClass 0x#####`. The caption of each window is `OleMainThreadWndName`, implying the window corresponds to the main (indeed only) thread in each apartment, which in this case are both STAs.

The lower portion of Figure 3 shows a pair of messages involved in the STA to STA method call. The first message is posted to the server's hidden COM window to produce a serialised method call. When it is complete, the second message is posted back to the client's hidden window to allow it to continue. The client STA thread is effectively blocked by a COM-provided message loop until the method call returns and so the return message arrives.

So objects in any given STA will never suffer concurrent access, since only one thread will ever be in the apartment. However, there can be multiple STAs in a process, so global data must still be protected with synchronisation of some sort, such as a semaphore, mutex or critical section.

How Does An MTA Work?

This might not sound like such a sensible question right now. The idea with an MTA is that the objects within it can be called from many threads simultaneously, so what is there to know? Well, at this point, remember the rule I stated earlier: only threads in a COM object's apartment are able to directly call its methods.

Consider a multi-threaded client application talking to a thread-safe object in an out-of-process server. The client asks COM to create the object, so COM starts the server, whose main thread enters an MTA and creates the COM object. Since the client is in another process, COM then provides a proxy object to the client application.

Now, several of the client's threads, all of which are in an MTA, wish to call upon the object. However, those threads are not in the object's apartment. They are in an apartment in the client. In fact the server only has one thread of its own. So what does COM do to help out? It provides threads from the RPC (Remote Procedure Call) thread pool which then enter the server's MTA. Each client call to one of the object's methods is executed on one of these pooled threads. That way, it doesn't matter how few explicitly created threads are running in the server.

Some liaising is still done with a hidden COM window during method calls, but the MTA threads need not have a message pump, as calls are not being serialised. Figure 4 shows WinSight examining a client with an STA communicating with an MTA out-of-process object on Windows 95. You can see there are still messages posted back and forth, and that the MTA window has a different caption and has different behaviour. In fact it also has a different window class on other Win32 platforms.

Of course, things are rather different with an in-process server. If an MTA client creates an object in an MTA server, the MTA client threads will be permitted to directly call the object methods.

Since MTA object methods can be called concurrently on various threads, global data and instance data must be accessed with care, using synchronisation mechanisms. Care must also be taken with thread-local variables (those declared with `threadvar` rather than `var`). Since the MTA uses arbitrary threads from the RPC thread pool, each time a method is called, the thread-local variables will have different values. This can be useful in some situations, but is usually unexpected.

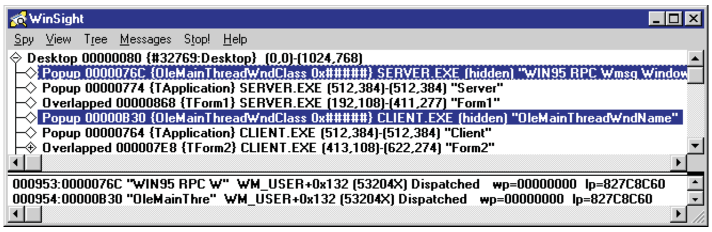
This point has a further, not necessarily obvious, ramification. MTA-based objects cannot synchronise data access by holding locks that have *thread affinity*, such as critical sections or mutexes. Since the thread that creates a mutex must be the same thread to release the mutex (and the same for a critical section), the arbitrary thread selection nature of an MTA prohibits their effectiveness, unless used entirely within a single method's execution.

Choosing Apartment Types

To initialise COM (and enter an apartment) a thread calls `OleInitialize`, `CoInitialize` or `CoInitializeEx`. `CoInitialize` and `OleInitialize` allow a thread to enter an STA. `CoInitializeEx` allows a thread to enter an STA if `COINIT_APARTMENTTHREADED` is passed as the second parameter or an MTA if `COINIT_MULTITHREADED` is passed (the default).

Note that `ComObj` performs this sort of COM initialisation for the primary thread automatically by its inclusion in a `uses` clause. For other threads to do COM work, they must explicitly initialise COM (and therefore enter an apartment). So, for example, to initialise COM and allow a (non-primary) thread to enter an MTA, you could use the code in Listing 1.

`CoInitializeEx` only exists on Windows 95 if DCOM for Windows



```
Desktop 00000080 {#32769:Desktop}  (0,0)-(1024,768)
  Popup 0000076C {OleMainThreadWndClass 0x#####} SERVER.EXE (hidden) "WIN95 RPC Wmsg Window
  Popup 00000774 {TApplication} SERVER.EXE (512,384)-(512,384) "Server"
  Overlapped 00000868 {TForm1} SERVER.EXE (192,108)-(411,277) "Form1"
  Popup 00000B30 {OleMainThreadWndClass 0x#####} CLIENT.EXE (hidden) "OleMainThreadWndName"
  Popup 00000764 {TApplication} CLIENT.EXE (512,384)-(512,384) "Client"
  Overlapped 000007E8 {TForm2} CLIENT.EXE (413,108)-(622,274) "Form2"

000953:0000076C "WIN95 RPC W"  WM_USER+0x132 (53204X) Dispatched   wp=00000000  lp=827C8C60
000954:00000B30 "OleMainThre"  WM_USER+0x132 (53204X) Dispatched   wp=00000000  lp=827C8C60
```

➤ *Figure 4: A client STA window and a server MTA window.*

95 has been installed, so care must be taken when calling it to ensure it exists. Note the code fully qualifies the reference to the CoInitializeEx variable in ComObj to ensure the compiler doesn't directly call the routine via its import declaration in the ActiveX unit. If it did, and the API was not available (which would be the case on Windows 95 without DCOM installed), an unpleasant fatal error would appear at program startup. If the CoInitializeEx API is not available, it falls back on CoInitialize.

A call to CoUninitialize causes a thread to exit its apartment and close down access to COM. This is done with consideration for pending COM calls stuck in a COM window message queue. If such messages are pending, COM enters a modal message loop dispatching those messages (but ignoring non-COM messages) to ensure no COM method calls are lost.

COM client applications and out-of-process COM server applications use this approach to ensure that each thread that needs to access COM objects is in an appropriate apartment. But things are a little different for in-process COM servers (DLLs or OCXs).

Since DLLs are loaded into the address space of the calling application, and the code in a DLL is accessed on the client application's thread, there might seem to be a possible conflict of thread awareness. If the DLL does not create threads of its own, how can it set up an appropriate apartment to ensure that calls are serialised (or not)?

One option would be for the DLL code to initialise COM regardless. However, a thread cannot actively be in two or more different apartment types at the same time, so if CoInitializeEx is called twice on one thread with different apartment type flags, the second call will cause an RPC_E_CHANGED_MODE error. So this is not an option.

DLLs must *never* initialise or shut down COM in their own initialisation or finalisation code. This is because of how Windows implicitly loads DLLs: it may cause an infinite loop or Access Violation, particularly when using DCOM across multiple machines.

➤ *Listing 1: Initialising COM for an MTA.*

```
uses ComObj, ActiveX;
...
if Assigned(ComObj.CoInitializeEx) then
  OleCheck(ComObj.CoInitializeEx(nil, COINIT_MULTITHREADED))
else
  CoInitialize(nil)
```

In-Proc COM Object Threading Model

The answer is that COM has already thought of this. When an in-process server is registered, each COM object has an extra value added into its registry data by the class factory advertising the appropriate threading model. When COM loads the in-process server and prepares to create a COM object from it, it first checks the ThreadingModel registry value to see what threading support the object has. Based upon what it finds, it will do one of two things.

If the client thread's apartment type matches the COM object's advertised apartment support, it does nothing at all, allowing the client to talk directly to the object. However, if the two apartment models clash, COM creates a thread of its own and enters an appropriate apartment for the COM object. The client thread is given a proxy and things go back to how they are with an out-of-process COM server.

Two COM objects that reside in the same process may belong to different apartments and therefore have different concurrency restrictions. This general arrangement caters for in-process servers with differing thread awareness to work together in the same process. A process with an MTA and one or more STAs is sometimes called a *mixed-model process*.

As you will be starting to appreciate, COM goes to a lot of trouble behind the scenes to ensure client apps and COM objects avoid conflict and work in harmony.

ComObj And Apartments

The ComObj unit is quite canny about its initialisation code. It first checks to see whether the current module is a DLL or EXE. If it's a DLL it does nothing. If it is an EXE it makes use of a ComObj global variable called CoInitFlags to determine what to do next with regard to initialising COM. CoInitFlags is pre-set by COM object class factories based upon the threading models of the COM objects in the out-of-process server.

CoInitFlags defaults to -1, which means CoInitialize will be called causing the primary thread to enter an STA. Any COM objects you create have associated class factories constructed in their unit initialisation sections. If the COM object's threading model suggests it works in an MTA then CoInitFlags is set to COINIT_MULTITHREADED. If the COM object claims support for an STA then CoInitFlags is set to COINIT_APARTMENTTHREADED (assuming it hasn't already been set to COINIT_MULTITHREADED).

CoInitFlags can also be manually set in the project source before the call to Application.Initialize if you want to change the automatic value or set a specific value. For example, you may wish to add in the flag COINIT_SPEED_OVER_MEMORY using the or operator, which causes COM to optimise for speed rather than memory. To do this, add ComObj and ActiveX into the project file's uses clause and insert this statement before Application.Initialize:

```
CoInitFlags := CoInitFlags or
  COINIT_SPEED_OVER_MEMORY;
```

Assuming CoInitializeEx can be found, CoInitFlags is passed to it, otherwise CoInitialize is called. Usefully, a procedural variable called CoInitializeEx is set to point to the routine if it exists (or nil if not). This variable was used in Listing 1.

Delphi 4 And CoInitFlags

If you are using Delphi 4, you should be aware that CoInitFlags works a little differently. The class factories do not set this variable with appropriate flags, meaning that the job is left to you. You must assign the appropriate threading model flag to `CoInitFlags` before `Application.Initialize` in the project source file.

The COM Object Wizard

Now that we have some background on the subject, we should look at exactly what the `Threading Model` attribute on the various COM-related wizards does. You can specify a target threading model when making a new COM object, Automation object, ActiveX control, ActiveForm and Active Server object. There are four options to choose from in Delphi 5: Single, Apartment, Free and Both, all explained in Table 1. Having specified a threading model for your object, it is your responsibility to ensure your object adheres to that threading model.

Inter-Apartment Interface Passing

I have mentioned a couple of times that a COM object can only be directly called from a thread in its own apartment. Threads in other apartments must talk to a COM proxy, which will be automatically created when the interface reference is marshaled across from the original thread.

It is an error for code in one thread to pass an interface reference directly to a thread in a different apartment. If the reference points to a COM proxy, an `RPC_E_WRONG_THREAD` error will be generated when a method call is made through it. If the reference is a direct pointer to the object, COM has no way to detect the problem but it is still an error.

Any interface reference received by a client application from an API call or method is valid for all threads in the caller's apartment, regardless of whether it is a direct reference to the object or a reference to a COM proxy. So interface references can readily be passed between any threads in an MTA.

So, how do you pass an interface reference from a thread in one apartment to a thread in another apartment? Or, how is the interface reference *marshaled* to the other thread? One way to marshal an interface reference is with the COM APIs `CoMarshalInterThreadInterfaceInStream` and `CoGetInterfaceAndReleaseStream` as described by Steve Teixeira (see Reference 5). These APIs work using a memory stream managed by COM. The first one adds the interface reference to the stream and the second one unmarshals an interface reference to a proxy object from the stream.

The only potential problem with these routines is that they only allow you to unmarshal the interface reference into one thread. If you need to give your interface reference to several threads in various apartments, they are not really helpful. This is where the Global Interface Table, or GIT, comes in.

COM manages one GIT per process and it is accessed through the `IGlobalInterfaceTable` interface (see Listing 2 for an appropriate definition, which can be found in the GITIntf.pas unit on the disk). You can register an arbitrary interface pointer as globally accessible with the `RegisterInterfaceInGlobal` method, which returns a unique identifying number (a cookie). `GetInterfaceFromGlobal` is used by any thread in the process to access an interface reference using the appropriate cookie and `RevokeInterfaceFromGlobal` stops it being available.

The GIT requires Windows NT 4 with Service Pack 3 or later,

➤ *Table 1: Threading models.*

Single	This attribute should be used in in-process servers only. Use of it in an out-of-process server will give the same result as Apartment. You use this if your whole server is completely unaware of thread issues and expects each of its objects to be accessed by the client's primary (STA) thread that created it. It is really for backward compatibility as you are unable to have multiple apartments in the server. Client method calls are all serialised through the client STA. The main client STA will receive an interface reference to the object, whereas any other STA threads will receive references to COM proxies. Because single-threaded objects are loaded into the main client STA (the first STA created in the client), it is important that the main STA remain active until the process terminates. The class factory for one of these objects will not add a `ThreadingModel` value into its registry data.
Apartment	This should be used for objects that are not thread-safe or re-entrant, but which can live in a multi-threaded server within an STA. In other words, whilst COM will ensure that method calls occur synchronously, the object must be careful to synchronise access to any global data in the server, as objects in other threads may access it simultaneously. The class factory for an in-process server will add a `ThreadingModel` value of *Apartment* into the registry during registration. An out-of-process server sets `CoInitFlags` to `COINIT_APARTMENTTHREADED` (if it is not already set to `COINIT_MULTITHREADED`). ActiveX controls and other COM objects with a GUI are typically marked with this attribute. This allows the UI synchronisation to happen without effort on the part of the COM object developer.
Free	This should be used for objects that are thread-safe and re-entrant, and can reside in an MTA. COM will permit multiple threads to access the object simultaneously. The class factory for an in-process server will add a `ThreadingModel` value of *Free* into the registry during registration. An out-of-process server sets `CoInitFlags` to `COINIT_MULTITHREADED`. Objects used in DCOM systems are often marked with this attribute.
Both	Objects that are thread-safe and can live in an MTA, but which can also work happily in an STA, should use this attribute. This gives more flexibility than either of the two previous attributes. The class factory for an in-process server will add a `ThreadingModel` value of *Both* into the registry during registration. An out-of-process server sets `CoInitFlags` to `COINIT_MULTITHREADED`. The implication of this is that specifying the Both-threading model only has any real impact in an in-process server. Additional comments about this threading model are coming up.

Windows 95 with DCOM 1.2 (or possibly 1.1, as the Platform SDK refers to both versions) or Windows 98. Unfortunately, Delphi 5 has no definition of this interface. For these reasons the code presented in this article will use the API pair rather than the GIT.

Creating STAs

In the client application, the primary thread can enter an STA by simply adding ComObj into the `uses` clause, or by calling `OleInitialize(nil)` or `CoInitialize(nil)`. Since the client's primary thread already has a message pump, all requirements are then met. An out-of-process server with a user interface also has a message pump in the primary thread, which will have entered an STA upon startup thanks to `ComObj`.

The trickier part is making the server create additional STAs. Since COM objects in a server are ultimately instantiated by their class factory we'll need to modify the class factory. Instead of creating objects on the same thread, the new class factory must create a new thread that enters a new STA and creates the object there.

There are three standard class factories that we can inherit from. `TComObjectFactory` creates COM objects that have no associated type information, `TTypedComObjectFactory` creates COM objects that do have type information (typically in a type library) and finally `TAutoObjectFactory` creates Automation objects. Arbitrarily, I will look specifically at `TTypedComObjectFactory`, which means I will need to test it with COM objects that have type information.

Each class factory implements the `IClassFactory` interface, which defines the `CreateInstance` method. It is this method that is responsible for bringing a COM object to life, so that is where the interesting code will be.

Listing 3 shows the factory class and the apartment thread class. When asked to instantiate a new COM object, the factory first checks it is running in an out-of-process server and that the object claims to support the apartment threading model. If this is the case it creates a new instance of the apartment thread class. It then has to wait until the secondary thread has created the COM object, which will be indicated by a Windows semaphore being released.

This will only work in Delphi 5, as Delphi 4's class factory did not surface the threading model through a convenient property. For Delphi 4, you would need to modify the code to assume an apartment threading model, and then make sure that you only use it in conjunction with apartment-threaded COM objects.

Whilst the factory is waiting, the thread enters an STA by initialising COM, then creates the COM object. Assuming nothing goes wrong, it marshals the interface reference into a memory stream and uses the semaphore to tell the factory that the object is ready. The factory can then read the interface reference (for a COM proxy) from the stream and pass it back to the client.

Whilst this is going on, the STA thread starts a message loop, dispatching messages to the hidden COM window as they come in. It continues to do this until either a `wm_Quit` message arrives (whereby `GetMessage` returns `False`) or the COM object is found to be done with. This is the case when its reference count is at 1 (the only connection to it is the thread's interface reference), at which point the thread terminates.

One point that should be made is that the implementation of `_Release` uses the `InterlockedDecrement` API. The Microsoft documentation states that on Windows 98 and NT it returns the decremented number. On Windows 95 it claims to return a positive number (assuming the decremented value is above zero) but not necessarily the actual decremented number, so the test may prove problematic.

```
const
  CLSID_StdGlobalInterfaceTable : TGUID =
    '{00000323-0000-0000-C000-000000000046}';
type
  IGlobalInterfaceTable = interface(IUnknown)
    ['{00000146-0000-0000-C000-000000000046}']
    function RegisterInterfaceInGlobal(pUnk: IUnknown; const riid: TIID;
      out dwCookie: DWord): HResult; stdcall;
    procedure RevokeInterfaceFromGlobal(dwCookie: DWord); safecall;
    function GetInterfaceFromGlobal(dwCookie: DWord; const riid: TIID; out ppv):
      HResult; stdcall;
  end;
function GIT: IGlobalInterfaceTable;
const GITIntf: IGlobalInterfaceTable = nil;
begin
  if not Assigned(GITIntf) then
    GITIntf := CreateComObject(CLSID_StdGlobalInterfaceTable) as
      IGlobalInterfaceTable;
  Result := GITIntf
end;
```

➤ *Listing 2: The Global Interface Table interface.*

A workaround often used is to change the code to check for a value of 0 returned from `_Release` on systems running Windows 95, as shown in Listing 4. Tests have shown such a change to work well.

This class factory and STA thread class are very similar to the ones that can be found in the VCL's `VCLCom` unit. This unit is used by the implementation units of remote data modules, MTS data modules and CORBA data modules. The idea of the unit is to overcome performance limitations when many clients try accessing apartment-threaded implementations of these COM objects by creating each apartment-threaded object in a separate STA.

Because of the Windows 95 issue with the `InterlockedDecrement` API, you might find that apartment-threaded MIDAS data modules will give the odd problem when hosted on Windows 95. If you encounter this, the most straightforward workaround is to move the server application to Windows NT, which has never suffered this problem.

A pair of simple projects that use the code in Listing 3 are on the disk as MultipleSTAClient.dpr and MultipleSTAServer.dpr. They are set up as normal, with a COM object in the server project, but the new class factory unit (STAThread.pas) is used in the COM object implementation unit.

Additionally, the reference to the normal class factory (`TTypedComObjectFactory`) in the unit's initialisation section is replaced with a reference to `TTypedComObjectFactory2` to allow the STAs to be created.

The client project can create up to 7 instances of the COM object using a bunch of checkboxes. Figure 5 shows Delphi's Threads Window displaying the server's primary thread along with 7 additional STA threads created by the class factory.

This is actually not the best implementation by a long shot. If 35 objects are needed in quick succession, 35 new threads will be created in addition to the server's primary STA thread, which might be excessive. Some form of thread pool would be advisable in a real application.

As you can see, though, creating multiple STAs involves a bit of fiddling around. There is also the problem that passing these interface references around requires marshaling which has an overhead (all those proxy objects being created). Having multiple threads in an MTA is much easier to achieve as those threads do not need a message pump.

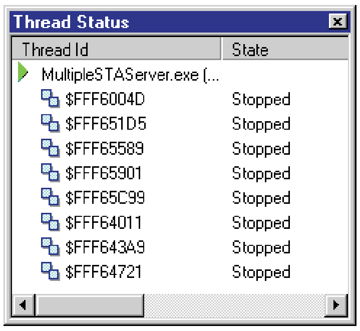
Apartment Interaction

Before finishing off, we should summarise how various apartment types interact between clients and both in-process and out-of-process servers.

In the case of an out-of-process server, the client will always be given a proxy and so COM will always sit in between. If the object is in an STA, COM serialises the calls. If the object is in an MTA, COM executes the client calls on RPC threads.

When the server is in-process, COM will interject between the client apartment and the server object only if there is a mismatch in threading support, otherwise it will allow the code in the client apartment thread to talk directly to the object. When COM does this

➤ *Figure 5: Multiple STAs in a process.*



```
Thread Status
Thread Id                          State
MultipleSTAServer.exe (...
    $FFF6004D                      Stopped
    $FFF651D5                      Stopped
    $FFF65589                      Stopped
    $FFF65901                      Stopped
    $FFF65C99                      Stopped
    $FFF64011                      Stopped
    $FFF643A9                      Stopped
    $FFF64721                      Stopped
```

➤ *Listing 3: A class factory that creates new STAs.*

```
type
  TTypedComObjectFactory2 =
    class(TTypedComObjectFactory, IClassFactory)
  protected
    //Create the COM object in a separate thread
    function CreateInstance(const UnkOuter: IUnknown;
      const IID: TGUID; out Obj): HResult; stdcall;
  end;
  TApartmentThread = class(TThread)
  private
    FFactory: IClassFactory2;
    FUnkOuter: IUnknown;
    FIID: TGuid;
    FSemaphore: THandle;
    FStream: Pointer;
    FCreateResult: HResult;
  protected
    procedure Execute; override;
  public
    constructor Create(Factory: IClassFactory2;
      UnkOuter: IUnknown; IID: TGuid);
    destructor Destroy; override;
    property Semaphore: THandle read FSemaphore;
    property CreateResult: HResult read FCreateResult;
    property ObjStream: Pointer read FStream;
  end;
function TTypedComObjectFactory2.CreateInstance(
  const UnkOuter: IUnknown; const IID: TGUID; out Obj):
HResult;
begin
  //Verify we are not an in-proc server and
  //that the object is STA-ready
  if not IsLibrary and (ThreadingModel = tmApartment) then
    begin
    LockServer(True);
    try
      //Create thread
      with TApartmentThread.Create(Self, UnkOuter, IID) do
        begin
        //Wait for thread to create the COM object
        if WaitForSingleObject(Semaphore, INFINITE) =
          WAIT_OBJECT_0 then begin
          Result := CreateResult;
          if Result <> S_OK then
            Exit;
          //If all is well, unmarshal interface from stream
          Result := CoGetInterfaceAndReleaseStream(
            IStream(ObjStream), IID, Obj);
        end else
          Result := E_FAIL
      end
    finally
      LockServer(False)
    end
  end else
    //In-proc servers and non-STA objects get created as
    // normal
    Result := inherited CreateInstance(UnkOuter, IID, Obj);
end;
constructor TApartmentThread.Create(Factory: IClassFactory2;
  UnkOuter: IUnknown; IID: TGuid);
begin
  inherited Create(True);
  FFactory := Factory;
  FUnkOuter := UnkOuter;
  FIID := IID;
  //Create the synchronisation device
  FSemaphore := CreateSemaphore(nil, 0, 1, nil);
  FreeOnTerminate := True;
  //After setting all thread attributes, let thread start
  Resume
end;
destructor TApartmentThread.Destroy;
begin
  CloseHandle(FSemaphore);
  inherited Destroy;
end;
procedure TApartmentThread.Execute;
var
  Msg: TMsg;
  Unk: IUnknown;
begin
  try
    CoInitialize(nil);    //Enter STA
    try
      //Create object
      FCreateResult := FFactory.CreateInstanceLic(
        FUnkOuter, nil, FIID, '', Unk);
      FUnkOuter := nil;
      FFactory := nil;
      //Marshal interface reference into stream
      if FCreateResult = S_OK then
        CoMarshalInterThreadInterfaceInStream(FIID, Unk,
          IStream(FStream));
      //Allow factory to read the interface reference
      ReleaseSemaphore(FSemaphore, 1, nil);
      if FCreateResult = S_OK then
        //Start the message pump
        while GetMessage(Msg, 0, 0, 0) do begin
          DispatchMessage(Msg);
          //See if only connection to this object is ours
          //If it is, then this thread's work is done
          Unk._AddRef;
          if Unk._Release = 1 then
            Break;
        end;
    finally
      Unk := nil;
      //Leave the STA
      CoUninitialize;
    end;
  except
    // No exceptions should go unhandled
  end;
end;
```

form of interjection it uses another thread from the RPC thread pool and enters an appropriate apartment for the object. The client apartment gets a proxy.

Efficiency Considerations

When writing in-process COM objects you must consider not only how much time you will put into ensuring thread safety, but also issues of efficiency in inter-apartment communication.

Firstly, whilst STA-compliant objects are easier to write, if many clients call into them the calls will be serialised and may cause unacceptable delay. This includes calls to multiple objects created within the same STA. Since there is only one thread in the STA all method calls to all objects will be serialised. Examples of where this might be the case include middle-tier COM servers or web servers that have many client applications.

If the client applications all fire off method calls to the server application, the calls will be executed one at a time, in series. Taking the time to make your object thread-safe may be a good investment, allowing multiple client calls to execute concurrently.

But if you take time to cover re-entrancy issues and protect instance data and decide to mark your object as MTA-compliant (using the *Free* attribute in the wizard), you should carefully consider which COM client application threads might call your object.

Out-of-process server objects will be called by clients that use COM proxies regardless of the client apartment, but in-process objects are a different matter. If many of the expected clients are to be STAs, many proxy objects will be created on your behalf and so a lot of implicit marshaling and thread-switching will occur.

The Both-Threading Model

In this scenario, it might be better to mark your object as *Both* in the wizard, meaning it will work both in an STA and an MTA. This removes COM's obligation to make COM proxies and means that your object can be created directly in the client STA or MTA, thereby increasing efficiency.

Using this threading model for an in-process object must also be done with care and forethought. Firstly, the object must be written in a thread-safe manner, because it can be directly instantiated in an MTA environment. Secondly, you must be very careful about accepting any callback references (interface references to objects who call you and expect you to call them back at some point later) as method parameters.

Let's say your object is created by an object in an MTA. Both your object and the creator are in the same apartment, and so your creator has an interface to you, not to a proxy. It calls one of your methods passing a reference to itself, which will also be a direct pointer. It is not a problem for you to call one of your caller's methods using this reference, as you are both in the same apartment.

Now let's say that the creator passes to another thread in the same MTA an interface reference to you. If that thread invokes a method that needs to call via the callback, this is still fine as both threads are in the same apartment. So where is the problem?

The problem comes in when the object is created in an STA. Again, the creator has a direct interface reference and if it calls one of your methods passing in a callback, this will also be a direct interface reference. No problem so far, as there is only one apartment involved. But let's assume the creating thread marshals its interface reference to your object to another thread, in another apartment. If your object uses the callback (which was not a proxy), we break one of the rules of COM. We are calling an STA object that lives in one thread from a completely different thread.

So, to overcome this problem, you must marshal any interface references that come into your object, but this is only necessary if you are in an STA. To do this in an MTA would lead to unnecessary marshaling, thanks to the proxy's existence.

```
function FinalRefCount: Integer;
begin
  //Return 0 on Win95 (Windows 4.0)
  if (Win32Platform =
    VER_PLATFORM_WIN32_WINDOWS) and
    (Win32MajorVersion = 4) and
    (Win32MinorVersion = 0) then
    Result := 0
  else
    Result := 1
end;
...
//The fixed code
Unk._AddRef;
if Unk._Release = FinalRefCount
  then Break;
```

➤ *Listing 4: Making Listing 3 work on Windows 95.*

So the *Both*-threading model should really only be used for objects whose interface methods never receive callback interfaces as parameters. If there is any uncertainty, don't use *Both*.

MTS, COM+, Windows 2000

When writing MTS objects, you can forget about the MTA model, as it is incompatible with MTS. Instead, the STA model is favoured (each STA object is created in a new thread in a new STA). MTS uses the concept of an *activity*, which is a group of one or more objects that do something on behalf of a client. Each activity has one single logical thread running through it (which enforces call serialisation but without a Windows message queue), although the objects can be distributed across multiple processes.

Activities came about to help COM objects become more scaleable. STAs alone are not scaleable without a lot of work pooling a number of STA threads. MTAs already pool a number of threads, but the concurrent access issue, along with the lack of thread affinity (discussed earlier), poses headaches for developers.

COM+, introduced in Windows 2000, introduces another threading model called the *Thread-Neutral Apartment* (TNA). Any process can have at most one TNA and in-process COM objects advertise a desire to run in a TNA with a `ThreadingModel` registry key of `Neutral`. A TNA is not a place where an object lives, but MTA and STA threads that create a TNA object get a proxy that allows method calls without involving a thread switch (therefore rather efficient).

A TNA object allows its methods to execute on multiple threads, but only one thread will be executing any method of a given object instance at any time. It does not require synchronisation of access to its instance data, unlike an MTA.

For UI-less objects, TNA is the preferred model under COM+, although objects with a UI should still use STAs.

Together, MTS and COM+ provide advanced and efficient thread management along with pooling logic which enable developers to worry less about the low-level intricacies of COM threading.

C++Builder 5 supports the development of COM+ objects and also supports the TNA: we can assume Delphi 6 will have this too.

References

1. *www.techvanguards.com/ com/com.htm*. Binh Ly's COM programming web pages, containing much of the scarce information on advanced COM issues such as concurrency and callbacks.

2. *Delphi COM Programming* by Eric Harmon, published by Macmillan Technical Publishing. This book is a useful COM reference text, although it contains no information on COM threading issues.

3. *Essential COM* by Don Box, published by Addison-Wesley.

4. *Effective COM* by Don Box et al, published by Addison-Wesley.

5. *COM Corner: More Callbacks* by Steve Teixeira in *The Delphi Magazine*, Issue 58 (June 2000). Discusses marshaling interfaces for callbacks.

6. *Win32 Multithreaded Programming* by Aaron Cohen, Mike Woodring, Ronald Pertruska, published by O'Reilly & Associates.

7. *Multithreading Applications in Win32: The Complete Guide To Threads* by Jim Beveridge and Robert Wiener, published by Addison-Wesley.

8. *Microsoft Developer Network Library CD*. There are many articles covering multi-threaded COM to be found on this library.

Summary

Hopefully, having hung on to the end, you now have a much better understanding of what COM does to help thread-safe and thread-ignorant code to work together in harmony without any outside help. Next time someone drops the word *apartment* into a conversation about COM, feel free to join in.

Acknowledgements

Thanks to Danny Thorpe, Binh Ly, Roy Nelson, Guy Smith-Ferrier, Geoff Lawrence and Malcolm Matthews for their feedback.

Brian Long is a UK-based freelance consultant and trainer. He spends most of his time running Delphi and C++Builder training courses for his clients, and doing problem-solving work for them. Brian is at brian@blong.com